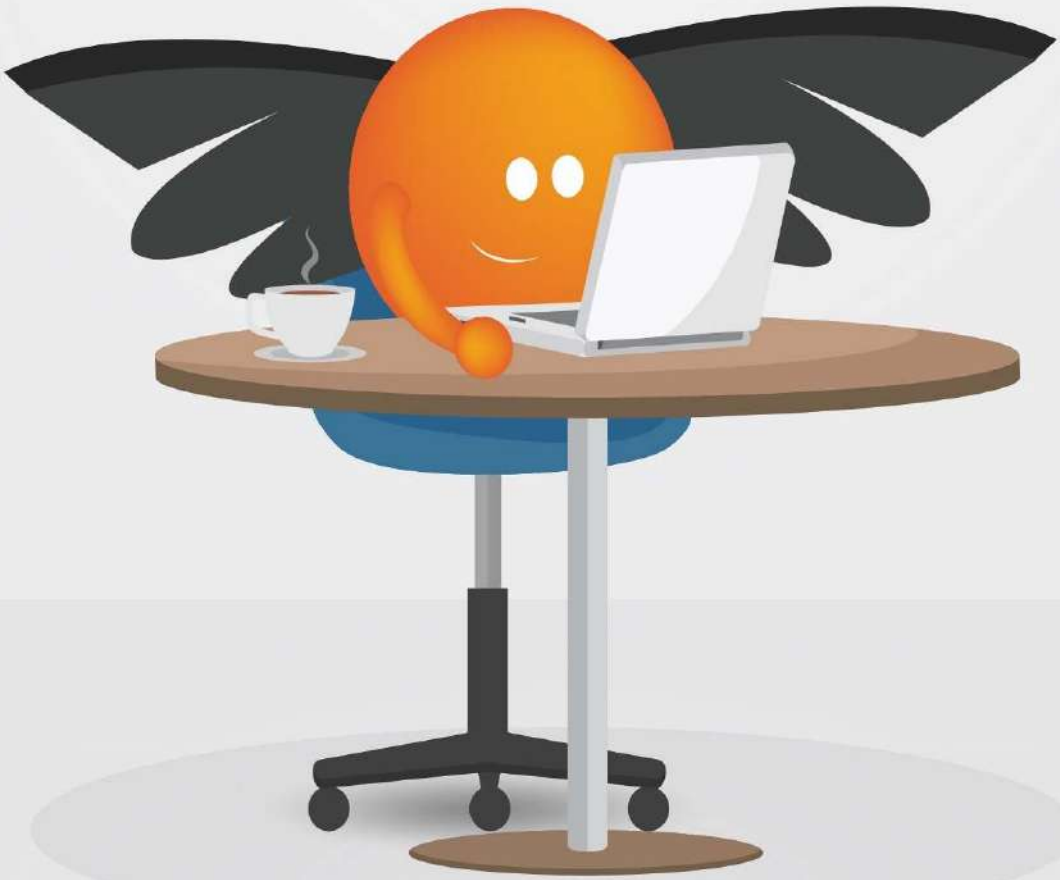
Universidad de los Andes
Facultad de Ingeniería

Departamento de Ingeniería de Sistemas y Computación

FUNDAMENTOS DE PROGRAMACIÓN

APRENDIZAJE ACTIVO BASADO EN CASOS



JORGE A. VILLALOBOS S. / RUBBY CASALLAS G.

Tabla de contenido

Fundamentos de Programación

Prefacio

Objetivos

Este libro es uno de los resultados del proyecto Cupi2, un proyecto de actualización curricular de la Universidad de los Andes (Bogotá, Colombia), cuyo principal propósito es encontrar mejores formas de enseñar/aprender a resolver problemas usando como herramienta un lenguaje de programación de computadores.

Este libro tiene como objetivo servir de herramienta fundamental en el proceso de enseñaza/aprendizaje de un primer curso de programación, usando un enfoque novedoso desde el punto de vista pedagógico, y moderno desde el punto de vista tecnológico.

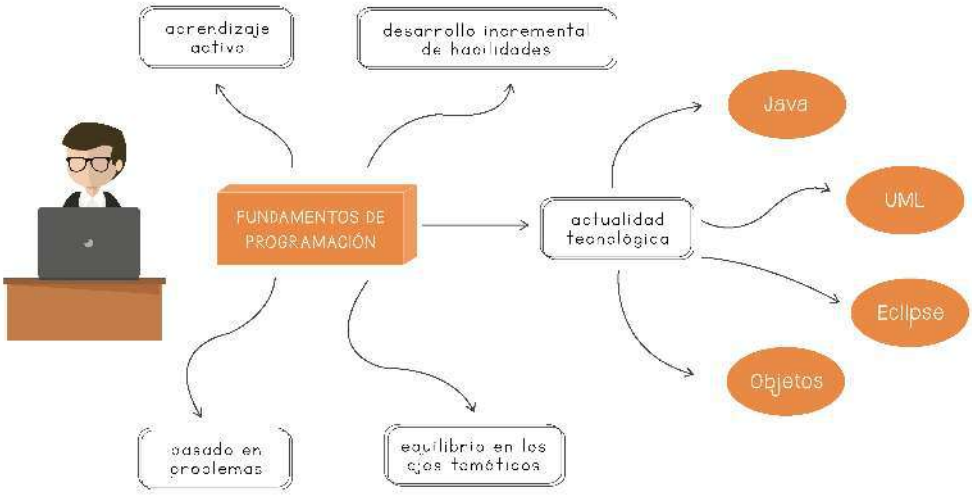
Queremos que el libro sea una herramienta de trabajo dentro de un proceso de aprendizaje, en el que el lector debe ser su principal protagonista. Por esta razón, a lo largo de los niveles que conforman el libro, se le irá pidiendo al lector que realice pequeñas tareas a medida que se presenta la teoría y, luego, que resuelva problemas completos directamente sobre el libro.

El Público Destinatario

El libro está dirigido a estudiantes que toman por primera vez un curso de programación de computadores, sin importar el programa de estudios que estén siguiendo. Esto quiere decir que para utilizar el libro no se necesita ninguna formación específica previa, y que las competencias generadas con este texto se pueden enmarcar fácilmente dentro de cualquier perfil profesional.

El Enfoque del Libro

La estrategia pedagógica diseñada para este libro gira alrededor de cinco pilares, los cuales se ilustran en la siguiente figura.



Aprendizaje activo: La participación activa del lector dentro del proceso de aprendizaje es un elemento fundamental en este tema, puesto que, más que presentar un amplio conjunto de conocimientos, el libro debe ayudar a generar las competencias o habilidades necesarias para utilizarlos de manera efectiva. Una cosa es entender una idea, y otra muy distinta lograr utilizarla para resolver un problema.

Desarrollo incremental de habilidades: Muchas de las competencias necesarias para resolver un problema usando un lenguaje de programación se generan a partir del uso reiterado de una técnica o metodología. No es suficiente con que el lector realice una vez una tarea aplicando los conceptos vistos en el curso, sino que debe ser capaz de utilizarlos de distintas maneras en distintos contextos.

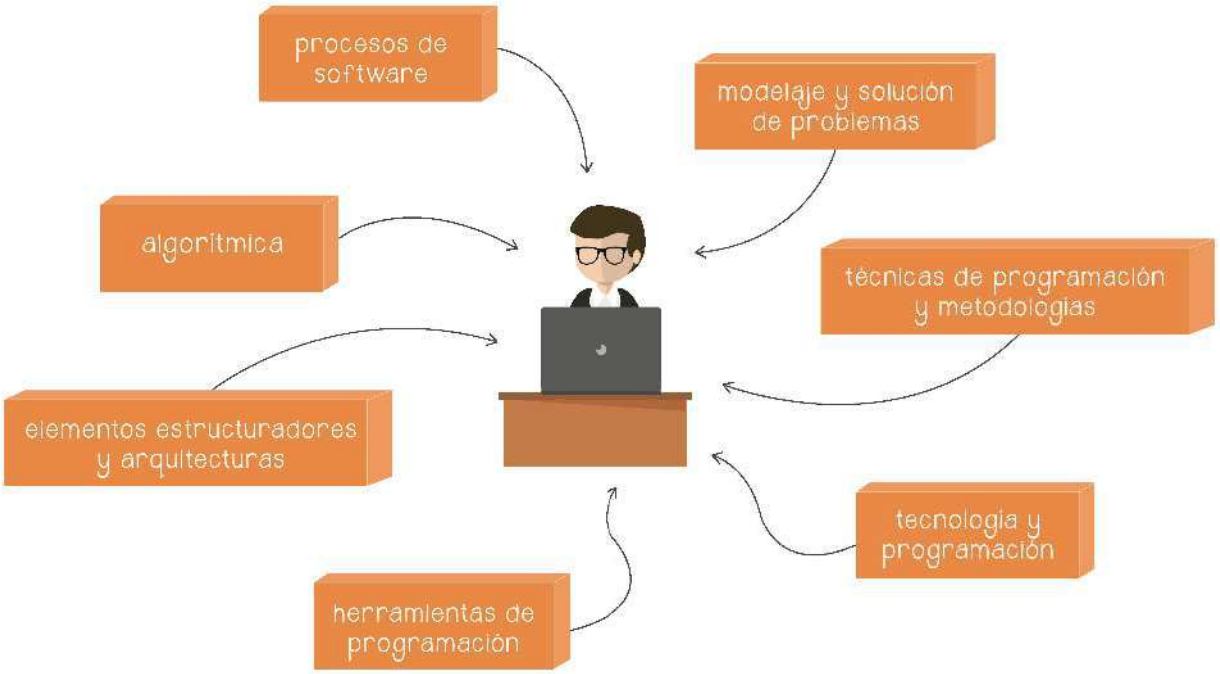
Equilibrio en los ejes temáticos: La solución de un problema usando un lenguaje de programación incluye un conjunto de conocimientos y habilidades de varios dominios. Dichos dominios son los que en la siguiente sección denominamos ejes conceptuales. Este curso intenta mantener un equilibro entre dichos ejes, mostrando así al lector que es en el adecuado uso de las herramientas y técnicas que provee cada uno de los ejes, donde se encuentra la manera correcta de escribir un programa de computador.

Basado en problemas: El libro gira alrededor de 24 problemas completos, cuya solución requiere el uso del conjunto de conceptos y técnicas presentadas en el libro. La mitad de los problemas se utilizan como casos de estudio y la otra mitad, como hojas de trabajo.

Actualidad tecnológica: En este libro se utilizan los elementos tecnológicos actuales, entre los cuales se encuentran el lenguaje de programación Java, el lenguaje de modelado UML, el ambiente de desarrollo de programas Eclipse y las técnicas de la programación orientada por objetos

Los Ejes Conceptuales de la Programación

Para resolver un problema utilizando como herramienta un lenguaje de programación, se necesitan conocimientos y habilidades en siete dominios conceptuales (llamados también ejes temáticos), los cuales se resumen en la siguiente figura:



Modelado y solución de problemas: Es la capacidad de abstraer la información de la realidad relevante para un problema, de expresar dicha realidad en términos de algún lenguaje y proponer una solución en términos de modificaciones de dicha abstracción. Se denomina "análisis" al proceso de crear dicha abstracción a partir de la realidad, y "especificación del problema" al resultado de expresar el problema en términos de dicha abstracción.

Algorítmica: Es la capacidad de utilizar un conjunto de instrucciones para expresar las modificaciones que se deben hacer sobre la abstracción de la realidad, para llegar a un punto en el cual el problema se considere resuelto. Se denomina "diseño de un algoritmo" al proceso de construcción de dicho conjunto de instrucciones.

Tecnología y programación: Son los elementos tecnológicos necesarios (lenguaje de programación, lenguaje de modelado, etc.) para expresar, en un lenguaje comprensible por una máquina, la abstracción de la realidad y el algoritmo que resuelve un problema sobre dicha abstracción. Programar es la habilidad de utilizar dicha tecnología para que una máquina sea capaz de resolver el problema.

Herramientas de programación: Son las herramientas computacionales (compiladores, editores, depuradores, gestores de proyectos, etc.) que permiten a una persona desarrollar un programa. Se pueden considerar una implementación particular de la tecnología.

Procesos de software: Es el soporte al proceso de programación, que permite dividir el trabajo en etapas claras, identificar las entradas y las salidas de cada etapa, garantizar la calidad de la solución, y la capacidad de las personas involucradas y estimar en un futuro el esfuerzo de desarrollar un programa. Aquí se incluye el ciclo de vida de un programa, los formularios, la definición de los entregables, el estándar de documentación y codificación, el control de tiempo, las técnicas de inspección de código, las técnicas de pruebas de programas, etc.

Técnicas de programación y metodologías: Son las estrategias y guías que ayudan a una persona a crear un programa. Se concentran en el cómo hacer las cosas. Definen un vocabulario sobre la manera de trabajar en cada una de las facetas de un programa, y están constituidas por un conjunto de técnicas, métricas, consejos, patrones, etc. para que un programador sea capaz de pasar con éxito por todo el ciclo de vida de desarrollo de una aplicación.

Elementos estructuradores y arquitecturas: Definen la estructura de la aplicación resultante, en términos del problema y de los elementos del mundo del problema. Se consideran elementos estructuradores las funciones, los objetos, los componentes, los servicios, los modelos, etc. Este eje se concentra en la forma de la solución, las responsabilidades de cada uno de los elementos, la manera como esos elementos se comunican, etc.

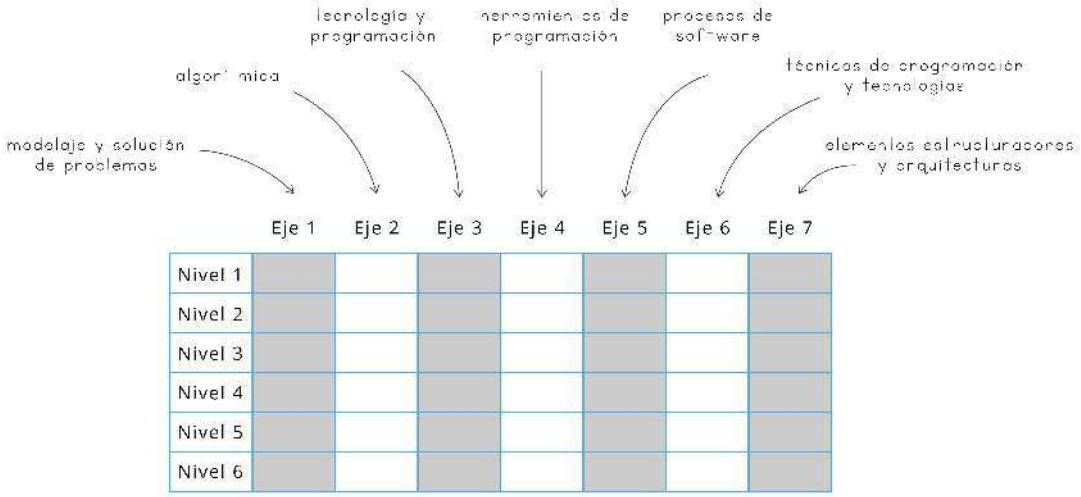
La Estructura del Libro

El libro sigue una estructura de niveles, en el cual se introducen los conceptos de manera gradual en los distintos ejes alrededor de los cuales gira la programación. Para hacerlo, se utilizan diversos casos de estudio o problemas, que le dan contexto a los temas y permiten ayudar a generar las habilidades necesarias para que el lector utilice de manera adecuada los conceptos vistos.

Los 6 niveles en los cuales se encuentra dividido el libro se muestran en la siguiente figura:

Nivel 1 - Problemas, Soluciones y Programas
Nivel 2 - Definición de Situaciones y Manejo de Casos
Nivel 3 - Manejo de Grupos de Atributos
Nivel 4 - Definición y Cumplimiento de Responsabilidades
Nivel 5 - Construcción de la Interfaz Gráfica
Nivel 6 - Manejo de Estructuras de dos Dimensiones y Persistencia

En cada uno de dichos niveles, se presentan de manera trasversal los elementos de los siete ejes conceptuales, dando lugar a una estructura como la que se presenta a continuación:



El contenido de cada uno de los niveles se resume de la siguiente manera:

Nivel 1. Problemas, Soluciones y Programas:

Se explica el proceso global de solución de un problema con un programa de computador. Esto incluye las etapas que deben seguirse para resolverlo y los distintos elementos que se deben ir produciendo a medida que se construye la solución. Se analizan problemas simples a través de la especificación de los servicios que el programa debe ofrecer y a través de un modelo conceptual del mundo del problema. Se explica la estructura de un programa de computador y el papel que desempeña cada uno de los elementos que lo componen. Se introduce el lenguaje de programación Java y los elementos necesarios para que el estudiante complete un programa utilizando expresiones simples, asignaciones y llamadas de métodos. Se utiliza un ambiente de desarrollo de programas y un espacio de trabajo predefinido, para completar una solución parcial a un problema.

Nivel 2. Definiendo Situaciones y Manejando Casos:

Se extienden los conceptos de modelado de las características de un objeto, utilizando nuevos tipos simples de datos y la técnica de definir constantes para representar los valores posibles de un atributo. Se utilizan expresiones como medio para identificar una situación posible en el estado de un objeto y para indicar la manera de modificar dicho estado. Se explican las instrucciones condicionales simples y compuestas como parte del cuerpo de un método, de manera que sea posible considerar distintos casos posibles en la solución de un problema. Se presenta de manera informal, una forma para identificar los métodos de una clase, utilizando para esto la técnica de agrupar los métodos por tipo de responsabilidad que tienen: construir, modificar o calcular.

Nivel 3: Manejando Grupos de Atributos:

Se explica la forma de utilizar las estructuras contenedoras de tamaño fijo como elementos de modelado de una característica de un elemento del mundo, que permiten almacenar una secuencia de valores (simples u objetos) y las estructuras contenedoras de tamaño variable como elementos de modelado que permiten manejar atributos cuyo valor es una secuencia de objetos. Se introducen las instrucciones repetitivas en el contexto del manejo de secuencias. Se extienden conceptos sobre el ambiente de desarrollo, en particular, se explica la forma de crear una clase completa en Java utilizando Eclipse. Se expone la forma de utilizar la documentación de un conjunto de clases escritas por otros y la forma de servirse de dicha documentación para poder incorporar y usar adecuadamente dichas clases en un programa que se está construyendo.

Nivel 4: Definición y Cumplimiento de Responsabilidades:

En este nivel se hace énfasis en la asignación de responsabilidades a las clases que representan la solución de un problema, utilizando técnicas simples. Se explica la técnica metodológica de dividir y conquistar para resolver los requerimientos funcionales de un problema y realizar la asignación de responsabilidades. Se estudia el concepto de contratos de los métodos tanto para poderlos definir como para poderlos utilizar en el momento de

invocar el método. Se enseña la forma de utilizar la clase Exception de Java para manejar los problemas asociados con la violación de los contratos. Se presenta la forma de documentar los contratos de los métodos utilizando la sintaxis definida por la herramienta javadoc. Se profundiza en el manejo del ambiente de desarrollo y el lenguaje Java, con el propósito de que el estudiante pueda escribir una clase completa del modelo del mundo, siguiendo una especificación dada en términos de un conjunto de contratos.

Nivel 5: Construyendo la Interfaz Gráfica:

El tema principal de este nivel es la construcción interfaces usuario simples. Se presenta la importancia que tiene la interfaz de usuario dentro de un programa de computador, teniendo en cuenta que es el medio de comunicación entre el usuario y el modelo del mundo. Se propone una arquitectura para un programa simple, repartiendo de manera adecuada las responsabilidades entre la interfaz de usuario, el modelo del mundo y las pruebas unitarias. Se enfatiza la importancia de mantener separadas las clases de esos tres dominios.

Nivel 6: Manejo de Estructuras de dos Dimensiones y Persistencia:

Se explica cómo utilizar el concepto de matriz como elemento de modelado que permite agrupar los elementos del mundo en una estructura contenedora de dos dimensiones de tamaño fijo. Se identifican los patrones de algoritmo para manejo de matrices, dada la especificación de un método. Se presenta la manera de utilizar un esquema simple de persistencia para el manejo del estado inicial de un problema. Por último, se resume el proceso de construcción de un programa seguido en el libro.

Las Herramientas y Recursos de Apoyo

Este libro es un libro de trabajo para el estudiante, donde puede realizar sus tareas y ejercicios asociados con cada nivel. Consideramos la página web del curso como parte integral del mismo. Todos los casos de estudio que se utilizan en los distintos niveles están resueltos e incluidos en dicho sitio web, así como las hojas de trabajo. Además, cada una de estas soluciones contiene puntos de extensión para que el profesor pueda diseñar ejercicios adicionales con sus estudiantes. Es importante que el profesor motive a los estudiantes a consultar la página web al mismo tiempo que lee el libro.

En la página web se encuentran tres tipos de elementos: (1) los programas de los casos de estudio, (2) los programas de las hojas de trabajo y (3) los entrenadores sobre ciertos conceptos. La página ha sido construida de manera que sea fácil navegar por los elementos que lo constituyen.

Todo el contenido de apoyo, lo mismo que otros materiales de apoyo al profesor, se puede encontrar en el sitio web del proyecto: http://cupi2.uniandes.edu.co

Licencias de Uso y Marcas Registradas

A lo largo de este libro hacemos mención a distintas herramientas y productos comerciales, todos los cuales tienen sus marcas registradas. Estos son: Microsoft Windows®, Microsoft Word®, Enterprise Architect®, Java®, Mozilla Firefox®, Eclipse®, JUnit®, Adobe Acrobat Reader®, Mac Apple Inc.®.

Todas las herramientas, programas, entrenadores y demás materiales desarrollados como soporte y complemento del libr, se distribuyen bajo la licencia “Academic Free License v. 2.1” que se rige por lo definido en: http://opensource.org/licenses/

Agradecimientos

Agradecemos a todas las personas, profesores y estudiantes, que han ayudado a que este libro se vuelva una realidad. En particular, queremos agradecer a Katalina Marcos por su valiosa ayuda y apoyo a todo lo largo del proceso.

También queremos reconocer el trabajo de Mario Sánchez y Pablo Bravo, nuestros incansables colaboradores. Ellos nos ayudaron en la construcción de muchos de los ejercicios y ejemplos alrededor de los cuales gira este libro. Gracias por su ayuda y su permanente buen humor.
Una mención especial merecen los profesores y estudiantes que durante el último año participaron en las secciones de prueba, usadas para validar el enfoque pedagógico propuesto, y quienes utilizaron como material de trabajo los primeros borradores de este libro. En particular, queremos reconocer el trabajo de Marcela Hernández, quien participó activamente en la revisión del borrador del libro y quien construyó una parte del material de apoyo a profesores que se encuentra disponible en el sitio WEB del proyecto.

Agradecemos la ayuda que recibimos de parte de LIDIE (Laboratorio de Investigación en Informática Educativa de la Universidad de los Andes), en las tareas de seguimiento, validación, diseño de las estrategias pedagógicas y diseño de los iconos y figuras que ilustran este libro.

Sobre los Autores

Jorge A. Villalobos, Ph.D

Obtuvo un doctorado en la Universidad Joseph Fourier (Francia), un Master en Informática en el Instituto Nacional Politécnico de Grenoble (Francia) y el título de Ingeniero en la Universidad de los Andes (Colombia). Actualmente es profesor asociado del Departamento

de Ingeniería de Sistemas de la Universidad de los Andes, en donde coordina el grupo de investigación en Construcción de Software y el proyecto Cupi2. Ha trabajado como investigador visitante en varias universidades europeas (España y Francia) y es el autor de los libros “Diseño y Manejo de Estructuras de Datos en C” (1996) y “Estructuras de Datos: Un Enfoque desde Tipos Abstractos” (1990).

Rubby Casallas, Ph.D

Obtuvo un doctorado en la Universidad Joseph Fourier (Francia), es Especialista en Sistemas de Información en la Organización de la Universidad de los Andes e Ingeniera de Sistemas de la misma Universidad. Ha sido profesora de la Universidad del Valle y del Rochester Institute Of Technology. Actualmente trabaja como profesora asociada del Departamento de Ingeniería de Sistemas de la Universidad de los Andes. Es coautora del libro “Introducción a la Programación” (1987).

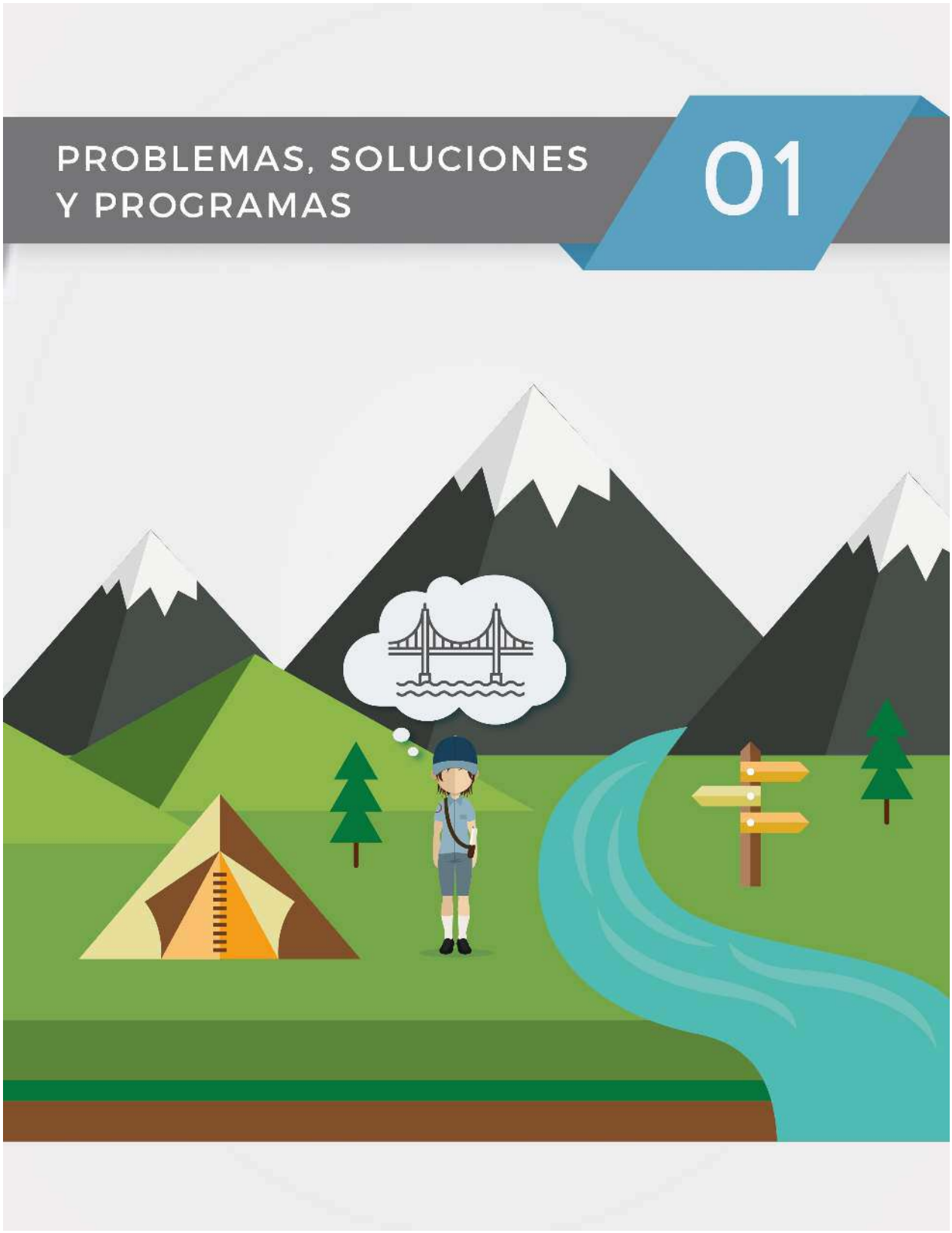
Dedicatoria

> A Vicky, por la alegría con la que me ha acostumbrado a vivir.

-Jorge

> A Irene y Jorge Esteban por el futuro que representan.

-Rubby


PROBLEMAS, SOLUCIONES
Y PROGRAMAS
01

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Explicar el proceso global de solución de un problema usando un programa de computador. Esto incluye las etapas que debe seguir para resolverlo y los distintos elementos que debe ir produciendo a medida que construye la solución.
- Analizar un problema simple que se va a resolver usando un programa de computador, construyendo un modelo con los elementos que intervienen en el problema y especificando los servicios que el programa debe ofrecer.
- Explicar la estructura de un programa de computador y el papel que desempeña cada uno de los elementos que lo componen.
- Completar una solución parcial a un problema (un programa incompleto escrito en el lenguaje Java), usando expresiones simples, asignaciones e invocaciones a métodos. Esto implica entender los conceptos de parámetro y de creación de objetos.
- Utilizar un ambiente de desarrollo de programas y un espacio de trabajo predefinido, para completar una solución parcial a un problema.

2. Motivación

La computación es una disciplina joven comparada con las matemáticas, la física o la ingeniería civil. A pesar de su juventud, nuestra vida moderna depende de los computadores. Desde la nevera de la casa, hasta el automóvil y el teléfono celular, todos requieren de programas de computador para funcionar. Se ha preguntado alguna vez, ¿cuántas líneas de código tienen los programas que permiten volar a un avión? La respuesta es varios millones.

El computador es una herramienta de trabajo, que nos permite aumentar nuestra productividad y tener acceso a grandes volúmenes de información. Es así como, con un computador, podemos escribir documentos, consultar los horarios de cine, bajar música de Internet, jugar o ver películas. Pero aún más importante que el uso personal que le podemos dar a un computador, es el uso que hacen de él otras disciplinas. Sería imposible sin los computadores llegar al nivel de desarrollo en el que nos encontramos en disciplinas como la biología (¿qué sería del estudio del genoma sin el computador?), la medicina, la ingeniería mecánica o la aeronáutica. El computador nos ayuda a almacenar grandes cantidades de información (por ejemplo, los tres mil millones de pares de bases del genoma humano, o los millones de píxeles que conforman una imagen que llega desde un satélite) y a manipularla a altas velocidades, para poder así ejecutar tareas que hasta hace sólo algunos años eran imposibles para nosotros.

El usuario de un programa de computador es aquél que, como parte de su trabajo o de su vida personal, utiliza las aplicaciones desarrolladas por otros para resolver un problema. Todos nosotros somos usuarios de editores de documentos o de navegadores de Internet, y los usamos como herramientas para resolver problemas. Un programador, por su parte, es la persona que es capaz de entender los problemas y necesidades de un usuario y, a partir de dicho conocimiento, es capaz de construir un programa de computador que los resuelva (o los ayude a resolver). Vista de esta manera, la programación se puede considerar fundamentalmente una actividad de servicio para otras disciplinas, cuyo objetivo es ayudar a resolver problemas, construyendo soluciones que utilizan como herramienta un computador.

Cuando el problema es grande (como el sistema de información de una empresa), complejo (como crear una visualización tridimensional de un diseño) o crítico (como controlar un tren), la solución la construyen equipos de ingenieros de software, entrenados especialmente para asumir un reto de esa magnitud. En ese caso aparecen también los arquitectos de software, capaces de proponer una estructura adecuada para conectar los componentes del programa, y un conjunto de expertos en redes, en bases de datos, en el

negocio de la compañía, en diseño de interfaces gráficas, etc. Cuanto más grande es el problema, más interdisciplinaridad se requiere. Piense que en un proyecto grande, puede haber más de 1000 expertos trabajando al mismo tiempo en el diseño y construcción de un programa, y que ese programa puede valer varios miles de millones de dólares. No en vano, la industria de construcción de software mueve billones de dólares al año.

Independiente del tamaño de los programas, podemos afirmar que la programación es una actividad orientada a la solución de problemas. De allí surgen algunos de los interrogantes que serán resueltos a lo largo de este primer nivel: ¿Cómo se define un problema? ¿Cómo, a partir del problema, se construye un programa para resolverlo? ¿De qué está conformado un programa? ¿Cómo se construyen sus partes? ¿Cómo se hace para que el computador entienda la solución?

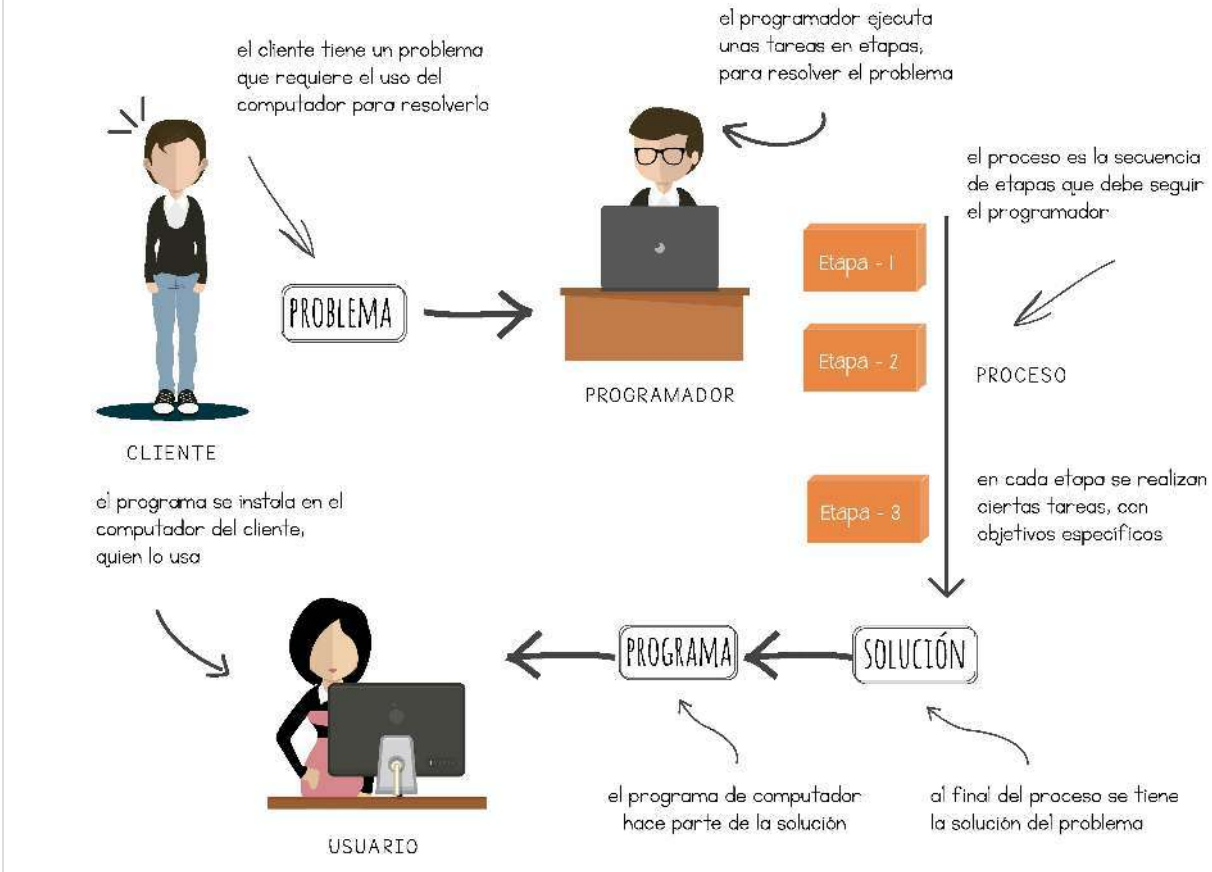
Bienvenidos, entonces, al mundo de la construcción de programas. Un mundo en constante evolución, en donde hay innumerables áreas de aplicación y posibilidades profesionales.

3. Problemas y Soluciones

Sigamos el escenario planteado en la figura 1.1, el cual resume el ciclo de vida de construcción de un programa y nos va a permitir introducir la terminología básica que necesitamos:

- **Paso 1**: Una persona u organización, denominada el **cliente**, tiene un problema y necesita la construcción de un programa para resolverlo. Para esto contacta una empresa de desarrollo de software que pone a su disposición un **programador**.
- **Paso 2**: El programador sigue un conjunto de etapas, denominadas el **proceso**, para entender el problema del cliente y construir de manera organizada una **solución** de buena calidad, de la cual formará parte un **programa**.
- **Paso 3**: El programador instala el programa que resuelve el problema en un computador y deja que el **usuario** lo utilice para resolver el problema. Fíjese que no es necesario que el cliente y el usuario sean la misma persona. Piense por ejemplo que el cliente puede ser el gerente de producción de una fábrica y, el usuario, un operario de la misma.

Fig. 1.1 Proceso de solución de un problema



- En la primera sección nos concentramos en la definición del problema, en la segunda en el proceso de construcción de la solución y, en la tercera, en el contenido y estructura de la solución misma.

3.1. Especificación de un Problema

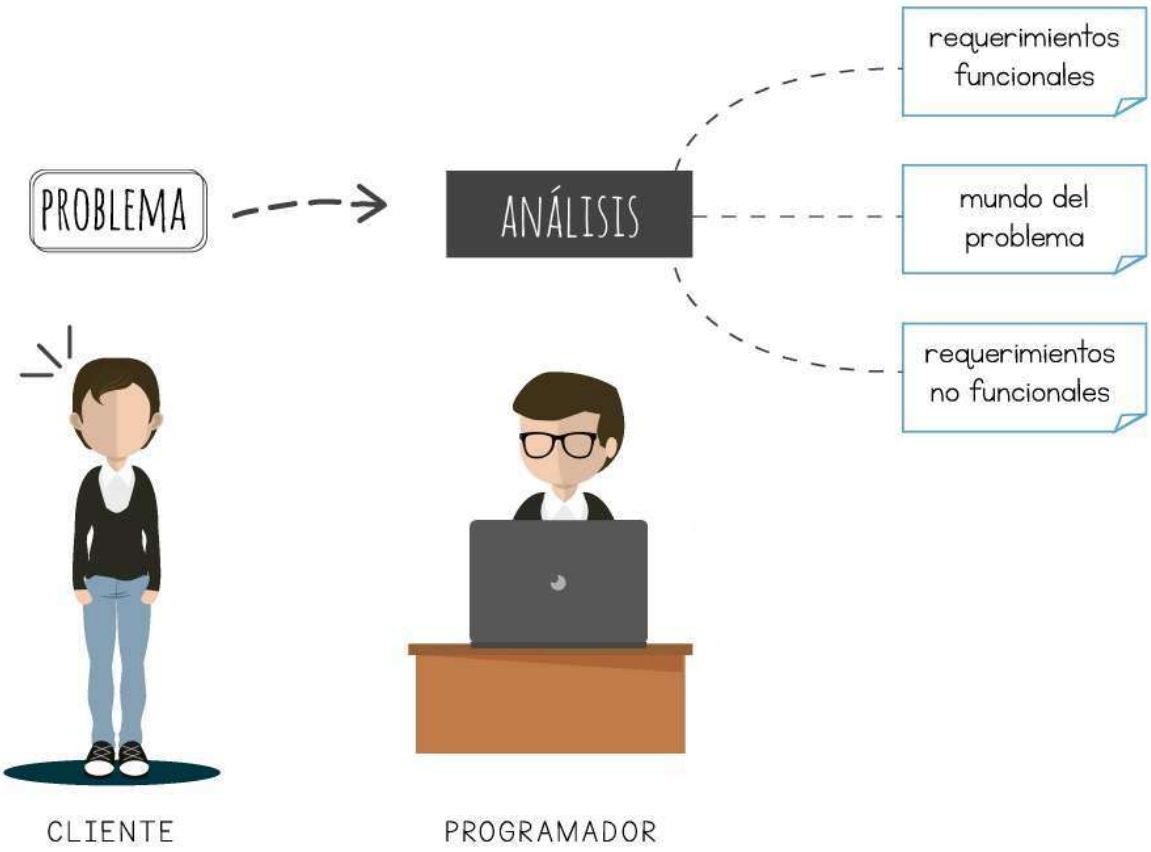
Partimos del hecho de que un programador no puede resolver un problema que no entiende. Por esta razón, la primera etapa en todo proceso de construcción de software consiste en tratar de entender el problema que tiene el cliente, y expresar toda la información que él suministre, de manera tal que cualquier otra persona del equipo de desarrollo pueda entender sin dificultad lo que espera el cliente de la solución. Esta etapa se denomina **análisis** y la salida de esta etapa la llamamos la **especificación** del problema.

Para introducir los elementos de la especificación, vamos a hacer el paralelo con otras ingenierías, que comparten problemáticas similares. Considere el caso de un ingeniero civil que se enfrenta al problema de construir una carretera. Lo primero que éste debe hacer es tratar de entender y especificar el problema que le plantean. Para eso debe tratar de identificar al menos tres aspectos del problema: (1) los requerimientos del usuario (entre qué puntos quiere el cliente la carretera, cuántos carriles debe tener, para qué tipo de tráfico debe ser la carretera), (2) el mundo en el que debe resolverse el problema (el tipo de terreno, la cantidad de lluvia, la temperatura), y (3) las restricciones y condiciones que plantea el cliente (el presupuesto máximo, que las pendientes no sobrepasen el 5%). Sería una pérdida de tiempo y de recursos para el ingeniero civil, intentar construir la carretera si no ha entendido y definido claramente los tres puntos antes mencionados. Y más que tiempo y recursos, habrá perdido algo muy importante en una profesión de servicio como es la ingeniería, que es la confianza del cliente.

En general, todos los problemas se pueden dividir en estos tres aspectos. Por una parte, se debe identificar lo que el cliente espera de la solución. Esto se denomina un **requerimiento funcional**. En el caso de la programación, un requerimiento funcional hace referencia a un servicio que el programa debe proveer al usuario. El segundo aspecto que conforma un problema es el **mundo** o **contexto** en el que ocurre el problema. Si alguien va a escribir un programa para una empresa, no le basta con entender la funcionalidad que éste debe tener, sino que debe entender algunas cosas de la estructura y funcionamiento de la empresa. Por ejemplo, si hay un requerimiento funcional de calcular el salario de un empleado, la descripción del problema debe incluir las normas de la empresa para calcular un salario. El tercer aspecto que hay que considerar al definir un problema son los **requerimientos no funcionales**, que corresponden a las restricciones o condiciones que impone el cliente al programa que se le va a construir. Fíjese que estos últimos se utilizan para limitar las

soluciones posibles. En el caso del programa de una empresa, una restricción podría ser el tiempo de entrega del programa, o la cantidad de usuarios simultáneos que lo deben poder utilizar. En la figura 1.2 se resumen los tres aspectos que conforman un problema.

Fig. 1.2 Aspectos que hacen parte del análisis de un problema



- Analizar un problema es tratar de entenderlo. Esta etapa busca garantizar que no tratemos de resolver un problema diferente al que tiene el cliente.
- Descomponer el problema en sus tres aspectos fundamentales, facilita la tarea de entenderlo: en cada etapa nos podemos concentrar en sólo uno de ellos, lo cual simplifica el trabajo.
- Esta descomposición se puede generalizar para estudiar todo tipo de problemas, no sólo se utiliza en problemas cuya solución sea un programa de computador.
- Además de entender el problema, debemos expresar lo que entendemos siguiendo algunas convenciones.
- Al terminar la etapa de análisis debemos generar un conjunto de documentos que contendrán nuestra comprensión del problema. Con dichos documentos podemos validar nuestro trabajo, presentándoselo al cliente y discutiendo con él.

Ejemplo 1

Objetivo: Identificar los aspectos que hacen parte de un problema.

El problema: una empresa de aviación quiere construir un programa que le permita buscar una ruta para ir de una ciudad a otra, usando únicamente los vuelos de los que dispone la empresa. Se quiere utilizar este programa desde todas las agencias de viaje del país.

Cliente	La empresa de aviación.
Usuario	Las agencias de viaje del país.
Requerimiento funcional	R1: dadas dos ciudades C1 y C2, el programa debe dar el itinerario para ir de C1 a C2, usando los vuelos de la empresa. En este ejemplo sólo hay un requerimiento funcional explícito. Sin embargo, lo usual es que en un problema haya varios de ellos.
Mundo del problema	En el enunciado no está explícito, pero para poder resolver el problema, es necesario conocer todos los vuelos de la empresa y la lista de ciudades a las cuales va. De cada vuelo es necesario tener la ciudad de la que parte, la ciudad a la que llega, la hora de salida y la duración del vuelo. Aquí debe ir todo el conocimiento que tenga la empresa que pueda ser necesario para resolver los requerimientos funcionales.
Requerimiento no funcional	El único requerimiento no funcional mencionado en el enunciado es el de distribución, ya que las agencias de viaje están geográficamente dispersas y se debe tener en cuenta esta característica al momento de construir el programa.

Tarea 1:

Objetivo: Identificar los aspectos que forman parte de un problema.

El problema: un banco quiere crear un programa para manejar sus cajeros automáticos. Dicho programa sólo debe permitir retirar dinero y consultar el saldo de una cuenta. Identifique y discuta los aspectos que constituyen el problema. Si el enunciado no es explícito con respecto a algún punto, intente imaginar la manera de completarlo.

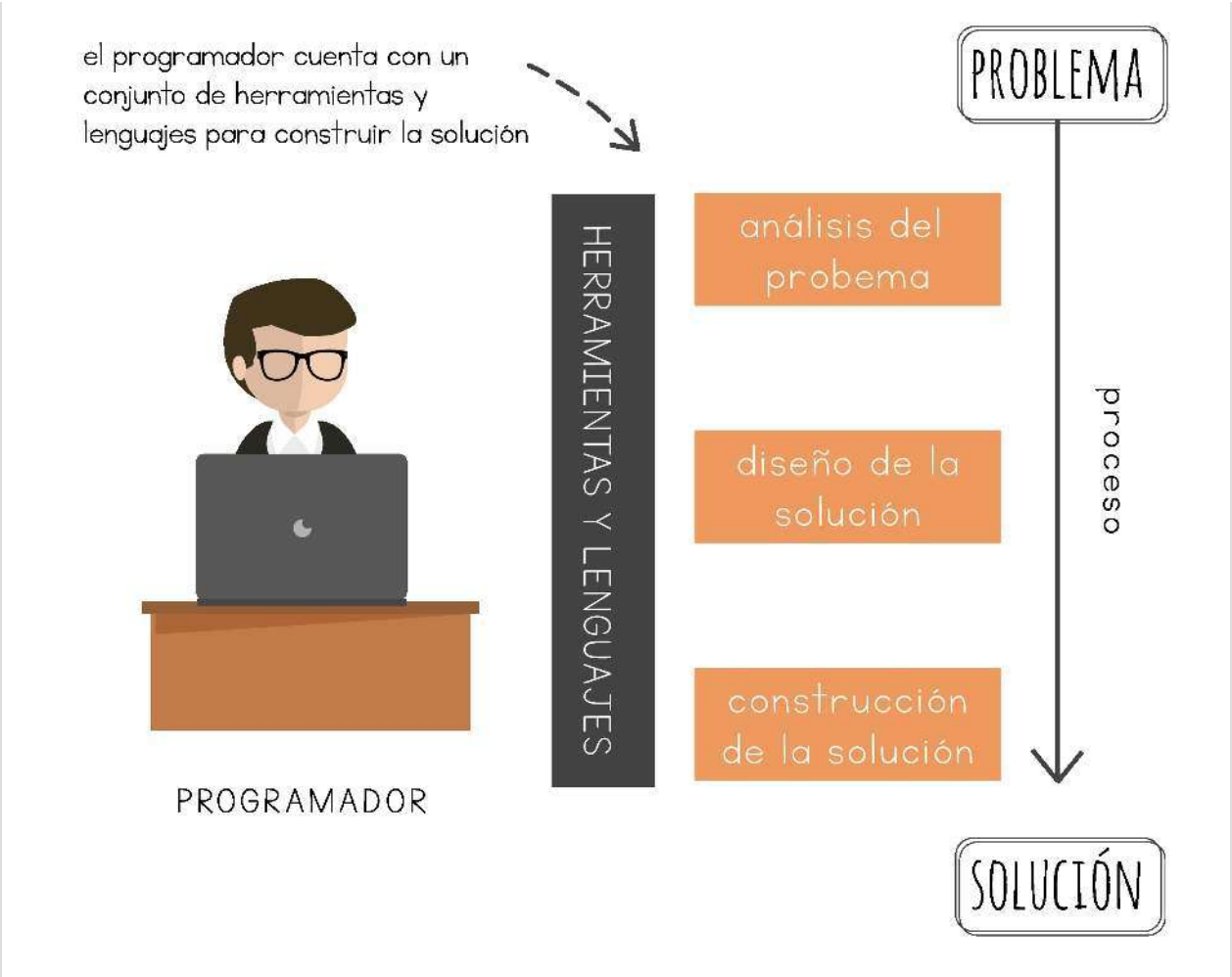
Cliente	
Usuario	
Requerimiento funcional	
Mundo del problema	
Requerimiento no funcional	

> Analizar un problema significa entenderlo e identificar los tres aspectos en los cuales siempre se puede descomponer: los requerimientos funcionales, el mundo del problema y los requerimientos no funcionales. Esta división es válida para problemas de cualquier tamaño.

3.2. El Proceso y las Herramientas

Entender y especificar el problema que se quiere resolver es sólo la primera etapa dentro del
proceso de desarrollo de un programa. En la figura 1.3 se hace un resumen de las principales etapas que constituyen el proceso de solución de un problema. Es importante que el lector entienda que si el problema no es pequeño (por ejemplo, el sistema de información de una empresa), o si los requerimientos no funcionales son críticos (por ejemplo, el sistema va a ser utilizado simultáneamente por cincuenta mil usuarios), o si el desarrollo se hace en equipo (por ejemplo, veinte ingenieros trabajando al mismo tiempo), es necesario adaptar las etapas y la manera de trabajar que se plantean en este libro. En este libro sólo abordamos la problemática de construcción de programas de computador para resolver problemas pequeños.

Fig. 1.3 Principales etapas del proceso de solución de problemas



- La primera etapa para resolver un problema es analizarlo. Para facilitar este estudio, se debe descomponer el problema en sus tres partes.
- Una vez que el problema se ha entendido y se ha expresado en un lenguaje que se pueda entender sin ambigüedad, pasamos a la etapa de diseño. Aquí debemos imaginarnos la solución y definir las partes que la van a componer. Es muy común comenzar esta etapa definiendo una estrategia.
- Cuando el diseño está terminado, pasamos a construir la solución.

El proceso debe ser entendido como un orden en el cual se debe desarrollar una serie de actividades que van a permitir construir un programa. El proceso planteado tiene tres etapas principales, todas ellas apoyadas por herramientas y lenguajes especiales:

- **Análisis del problema**: el objetivo de esta etapa es entender y especificar el problema que se quiere resolver. Al terminar, deben estar especificados los requerimientos funcionales, debe estar establecida la información del mundo del problema y deben estar definidos los requerimientos no funcionales.
- **Diseño de la solución**: el objetivo es detallar, usando algún lenguaje (planos, dibujos, ecuaciones, diagramas, texto, etc.), las características que tendrá la solución antes de ser construida. Los diseños nos van a permitir mostrar la solución antes de comenzar el proceso de fabricación propiamente dicho. Es importante destacar que dicha

especificación es parte integral de la solución.

- **Construcción de la solución**: tiene como objetivo **implementar** el programa a partir del diseño y probar su correcto funcionamiento.

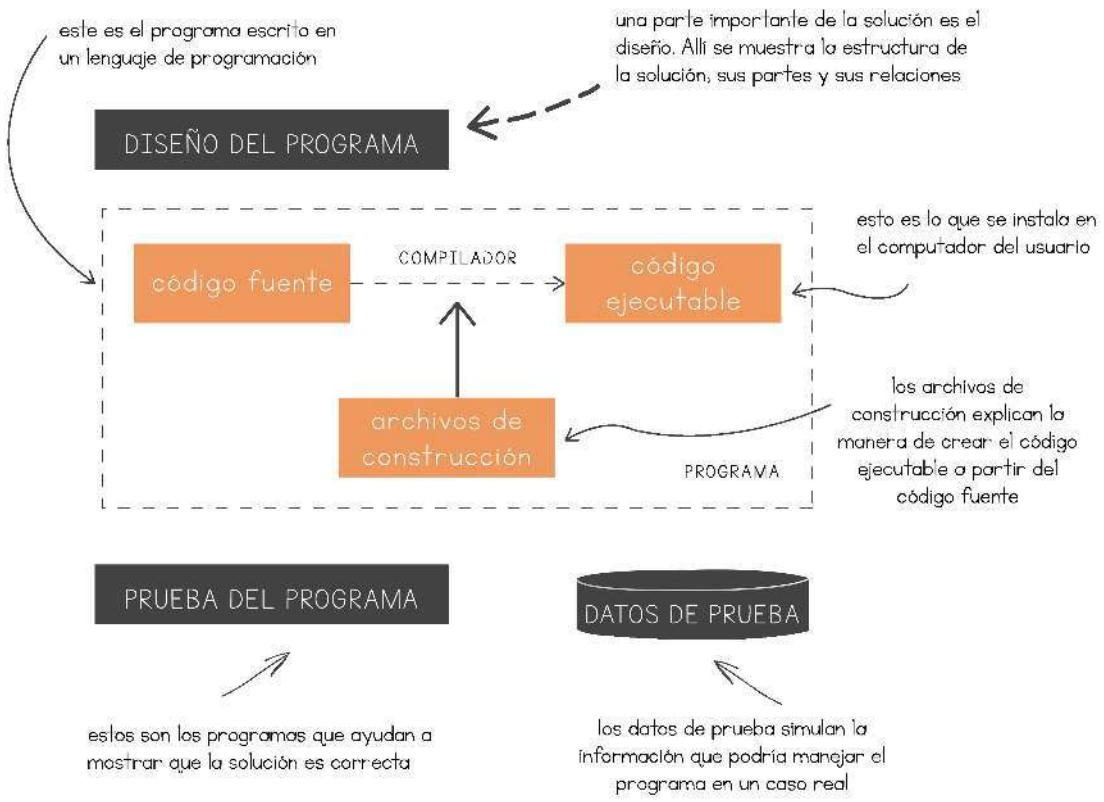
Cada una de las etapas de desarrollo está apoyada por herramientas y lenguajes, que van a permitir al programador expresar el producto de su trabajo. En la etapa de construcción de la solución es conveniente contar con un **ambiente de desarrollo** que ayuda, entre otras cosas, a editar los programas y a encontrar los errores de sintaxis que puedan existir.

> Las etapas iniciales del proceso de construcción de un programa son críticas, puesto que cuanto más tarde se detecta un error, más costoso es corregirlo. Un error en la etapa de análisis (entender mal algún aspecto del problema) puede implicar la pérdida de mucho tiempo y dinero en un proyecto. Es importante que al finalizar cada etapa, tratemos de asegurarnos de que vamos avanzando correctamente en la construcción de la solución.

3.3. La Solución a un Problema

La solución a un problema tiene varios componentes, los cuales se ilustran en la figura 1.4. El primero es el **diseño** (los planos de la solución) que debe definir la estructura del programa y facilitar su posterior mantenimiento. El segundo elemento es el **código fuente** del programa, escrito en algún **lenguaje de programación** como Java, C, C# o C++. El código fuente de un programa se crea y edita usando el ambiente de desarrollo mencionado en la sección anterior.

Fig. 1.4 Elementos que forman parte de la solución de un problema



Existen muchos tipos de lenguajes de programación, entre los cuales los más utilizados en la actualidad son los llamados lenguajes de **programación orientada a objetos**. En este libro utilizaremos Java que es un lenguaje orientado a objetos muy difundido y que iremos presentando poco a poco, a medida que vayamos necesitando sus elementos para resolver problemas.

> Un programa es la secuencia de instrucciones (escritas en un lenguaje de programación) que debe ejecutar un computador para resolver un problema.

El tercer elemento de la solución son los archivos de construcción del programa. En ellos se explica la manera de utilizar el código fuente para crear el **código ejecutable**. Este último es el que se instala y ejecuta en el computador del usuario. El programa que permite traducir el código fuente en código ejecutable se denomina **compilador**. Antes de poder construir nuestro primer programa en Java, por ejemplo, tendremos que conseguir el respectivo compilador del lenguaje.

El último elemento que forma parte de la solución son las **pruebas**. Allí se tiene un programa que es capaz de probar que el programa que fue entregado al cliente funciona correctamente. Dicho programa funciona sobre un conjunto predefinido de datos, y es

capaz de validar que para esos datos predefinidos (y que simulan datos reales), el programa funciona bien.

> La solución de un problema tiene tres partes: (1) el diseño, (2) el programa y (3) las pruebas de corrección del programa. Estos son los elementos que se deben entregar al cliente. Es común que, además de los tres elementos citados anteriormente, la solución incluya un manual del usuario, que explique el funcionamiento del programa.

Si por alguna razón el problema del cliente evoluciona (por ejemplo, si el cliente pide un nuevo requerimiento funcional), cualquier programador debe poder leer y entender el diseño, añadirle la modificación pedida, ajustar el programa y extender las pruebas para verificar la nueva extensión.

La figura 1.5 muestra dos mapas conceptuales (parte a y parte b) que intentan resumir lo visto hasta el momento en este capítulo.

Fig. 1.5 (a) Mapa conceptual de la solución de un problema con un computador

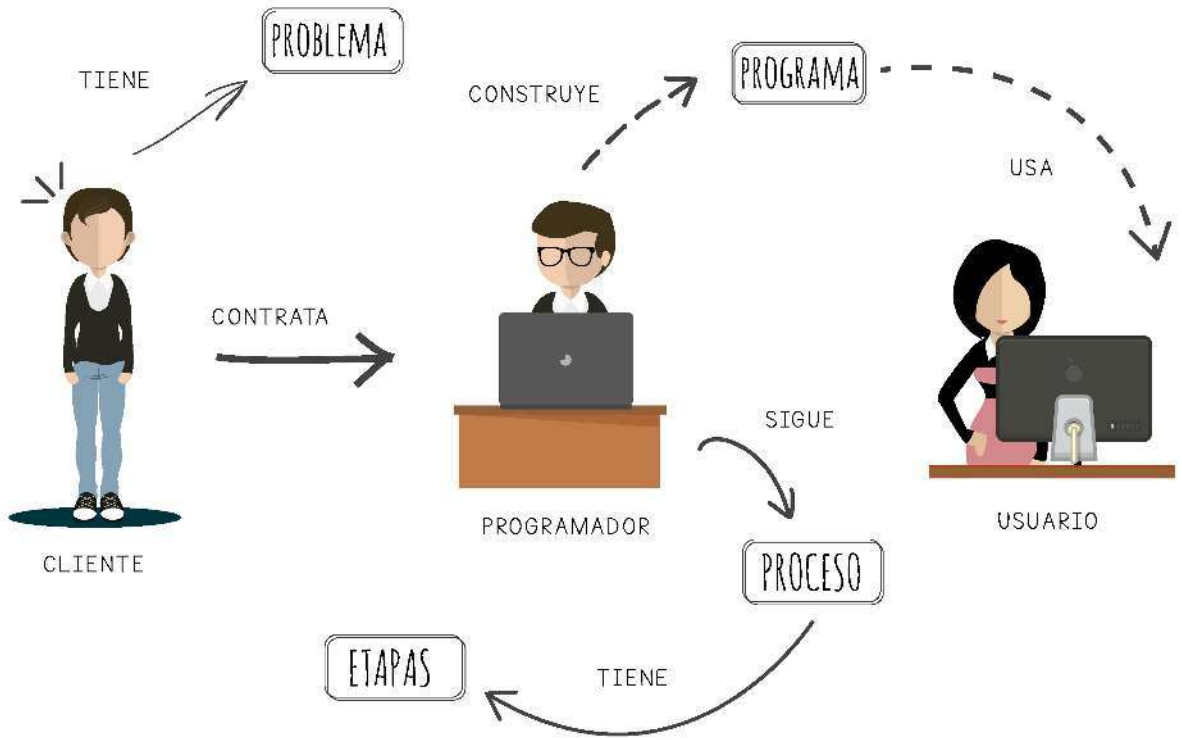
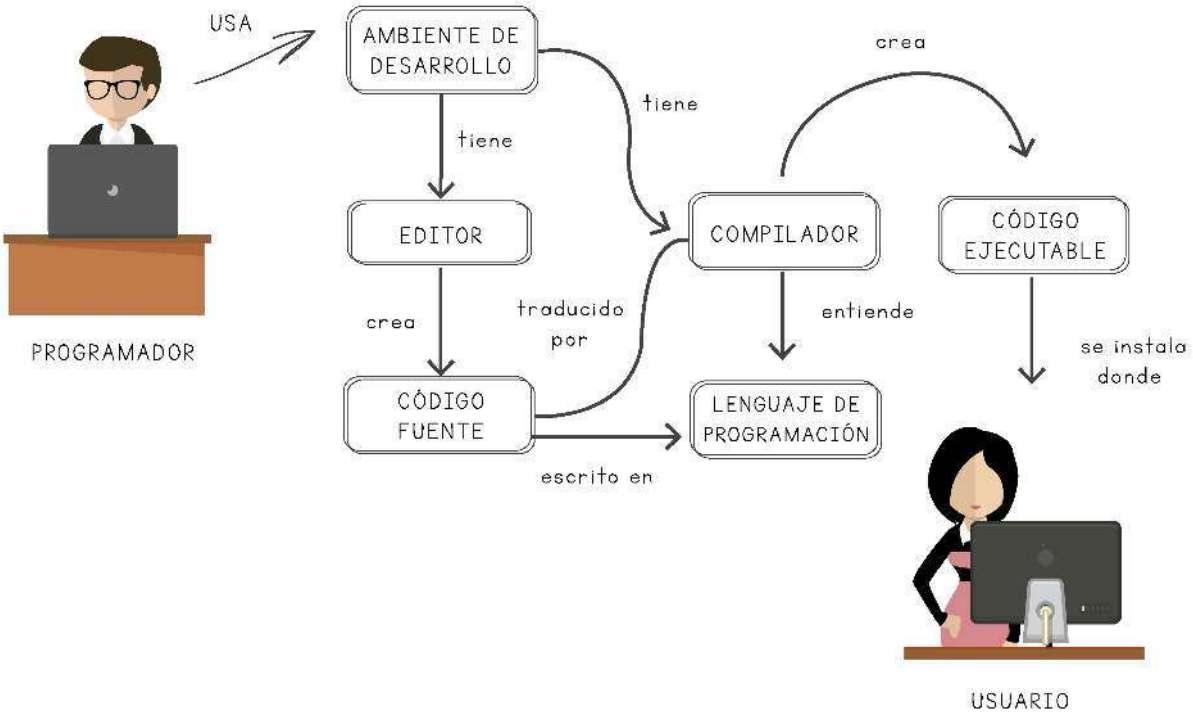


Fig. 1.5 (b) Mapa conceptual de la construcción de un programa



4. Casos de Estudio

Los tres casos de estudio que se presentan a continuación serán utilizados en el resto del capítulo para ilustrar los conceptos que se vayan introduciendo. Puede encontrar estos casos de estudio en la página web. Se recomienda leerlos detenidamente antes de continuar y tratar de imaginar el funcionamiento de los programas que resuelven los problemas, utilizando para esto las figuras que se muestran. Al final del capítulo encontrará otros casos de estudio diferentes, con las respectivas hojas de trabajo para desarrollarlos.

4.1 Caso de Estudio Nº 1: Un Empleado

Para este caso de estudio vamos a considerar un programa que administra la información de un empleado.

El empleado tiene un nombre, un apellido, un género (masculino o femenino), una fecha de nacimiento y una imagen asociada (su foto). Además, tiene una fecha de ingreso a la empresa en la que trabaja y un salario básico asignado.

Desde el programa se debe poder realizar las siguientes operaciones: (1) calcular la edad actual del empleado, (2) calcular la antigüedad en la empresa, (3) calcular las prestaciones a las que tiene derecho en la empresa, (4) cambiar el salario del empleado, y (5) cambiar el empleado.

4.2. Caso de Estudio Nº 2: Un Simulador Bancario

Una de las actividades más comunes en el mundo financiero es la realización de simulaciones que permitan a los clientes saber el rendimiento de sus productos a través del tiempo, contemplando diferentes escenarios y posibles situaciones que se presenten.

Se quiere crear un programa que haga la simulación en el tiempo de la cuenta bancaria de un cliente. Un cliente tiene un nombre y un número de cédula, el cual identifica la cuenta. Una cuenta, por su parte, está constituida por tres productos financieros básicos: (1) una cuenta de ahorros, (2) una cuenta corriente y (3) un certificado de depósito a término (CDT). Estos productos son independientes y tienen comportamientos particulares.

El saldo total de la cuenta es la suma de lo que el cliente tiene en cada uno de dichos productos. En la cuenta corriente, el cliente puede depositar o retirar dinero. Su principal característica es que no recibe ningún interés por el dinero que se encuentre allí

depositado. En la cuenta de ahorros, el cliente recibe un interés mensual del 0,6% sobre el saldo. Cuando el cliente abre un CDT, define la cantidad de dinero que quiere invertir y negocia con el banco el interés mensual que va a recibir. A diferencia de la cuenta corriente o la cuenta de ahorros, en un CDT no se puede consignar ni retirar dinero. La única operación posible es cerrarlo, en cuyo caso, el dinero y sus intereses pasan a la cuenta corriente.

Se quiere que el programa permita a una persona simular el manejo de sus productos bancarios, dándole las facilidades de: (1) hacer las operaciones necesarias sobre los productos que conforman la cuenta, y (2) avanzar mes por mes en el tiempo, para que el cliente pueda ver el resultado de sus movimientos bancarios y el rendimiento de sus inversiones.

- Con el botón marcado como "Avanzar mes" el usuario puede avanzar un mes en la simulación y ver los resultados de sus inversiones.
- Con los seis botones de la parte derecha de la ventana, el usuario puede simular el manejo que va a hacer de los productos que forman parte de su cuenta bancaria.

- En la parte media de la ventana, aparecen el saldo que tiene en cada producto y el interés que está ganando en cada caso.

4.3. Caso de Estudio Nº 3: Un Triángulo

En este caso se quiere construir un programa que permita manejar un triángulo. Esta figura geométrica está definida por tres puntos, cada uno de los cuales tiene dos coordenadas X, Y. Un triángulo tiene además un color para las líneas y un color de relleno. Un color por su parte, está definido por tres valores numéricos entre 0 y 255 (estándar RGB por Red-Green-Blue). El primer valor numérico define la intensidad en rojo, el segundo en verde y el tercero en azul. Más información sobre esta manera de representar los colores la puede encontrar por Internet. ¿Cuál es el código RGB del color negro? ¿Y del color blanco?

El programa debe permitir: (1) visualizar el triángulo en la pantalla, (2) calcular el perímetro del triángulo, (3) calcular el área del triángulo, (4) calcular la altura del triángulo, (5) cambiar el color del triángulo y (6) cambiar las líneas del triángulo.

- Con los tres botones de la izquierda, el usuario puede cambiar los puntos que definen el triángulo, el color de las líneas y el color del fondo.
- En la zona marcada como "Medidas en pixeles", el usuario puede ver el perímetro, el

área y la altura del triángulo (en píxeles).

- En la parte derecha aparece dibujado el triángulo descrito por sus tres puntos.

5. Comprensión y Especificación del Problema

Ya teniendo claras las definiciones de problema y sus distintos componentes, en esta sección vamos a trabajar en la parte metodológica de la etapa de análisis. En particular, queremos responder las siguientes preguntas: ¿cómo especificar un requerimiento funcional?, ¿cómo saber si algo es un requerimiento funcional?, ¿cómo describir el mundo del problema? Dado que el énfasis de este libro no está en los requerimientos no funcionales, sólo mencionaremos algunos ejemplos sencillos al final de la sección.

> Es imposible resolver un problema que no se entiende.

La frase anterior resume la importancia de la etapa de análisis dentro del proceso de solución de problemas. Si no entendemos bien el problema que queremos resolver, el riesgo de perder nuestro tiempo es muy alto.

A continuación, vamos a dedicar una sección a cada uno de los elementos en los cuales queremos descomponer los problemas, y a utilizar los casos de estudio para dar ejemplos y generar en el lector la habilidad necesaria para manejar los conceptos que hemos ido introduciendo. No más teoría por ahora y manos a la obra.

5.1. Requerimientos Funcionales

Un requerimiento funcional es una operación que el programa que se va a construir debe proveer al usuario, y que está directamente relacionada con el problema que se quiere resolver. Un requerimiento funcional se describe a través de cuatro elementos:

- Un identificador y un nombre.
- Un resumen de la operación.
- Las entradas (datos) que debe dar el usuario para que el programa pueda realizar la operación.
- El resultado esperado de la operación. Hay tres tipos posibles de resultado en un requerimiento funcional: (1) una modificación de un valor en el mundo del problema, (2) el cálculo de un valor, o (3) una mezcla de los dos anteriores.

Ejemplo 2

Objetivo: Ilustrar la manera de documentar los requerimientos funcionales de un problema.

En este ejemplo se documenta uno de los requerimientos funcionales del caso de estudio del empleado. Para esto se describen los cuatro elementos que lo componen.

Nombre	**R1: Actualizar el salario básico del empleado**	**Es conveniente asociar un identificador con cada requerimiento, para poder hacer fácilmente referencia a él. En este caso el identificador es R1. Es aconsejable que el nombre de los requerimientos corresponda a un verbo en infinitivo, para dar una idea clara de la acción asociada con la operación. En este ejemplo el verbo asociado con el requerimiento es "actualizar".**
Resumen	Permite modificar el salario básico del empleado	El resumen es una frase corta que explica sin mayores detalles el requerimiento funcional.
Entradas	Nuevo salario	Las entradas corresponden a los valores que debe suministrar el usuario al programa para poder resolver el requerimiento. En el requerimiento del ejemplo, si el usuario no da como entrada el nuevo salario que quiere asignar al empleado, el programa, no podrá hacer el cambio. Un requerimiento puede tener cero o muchas entradas. Cada entrada debe tener un nombre que indique claramente su contenido. No es buena idea utilizar frases largas para definir una entrada.
Resultado	El salario del empleado ha sido actualizado con el nuevo salario	El resultado del requerimiento funcional de este ejemplo es una modificación de un valor en el mundo del problema: el salario del empleado cambió. Un ejemplo de un requerimiento que calcula un valor podría ser aquél que informa la edad del empleado. Fíjese que el hecho de calcular esta información no implica la modificación de ningún valor del mundo del problema. Un ejemplo de un requerimiento que modifica y calcula a la vez, podría ser aquél que modifica el salario del empleado y calcula la nueva retención en la fuente.

En la etapa de análisis, el cliente debe ayudarle al programador a concretar esta información. La responsabilidad del programador es garantizar que la información esté completa y que sea clara. Cualquier persona que lea la especificación del requerimiento debe entender lo mismo.

Para determinar si algo es o no un requerimiento funcional, es conveniente hacerse tres preguntas:

- ¿Poder realizar esta operación es una de las razones por las cuales el cliente necesita construir un programa? Esto descarta todas las opciones que están relacionadas con el manejo de la interfaz ("poder cambiar el tamaño de la ventana", por ejemplo) y todos los requerimientos no funcionales, que no corresponden a operaciones sino a

restricciones.

- ¿La operación no es ambigua? La idea es descartar que haya más de una interpretación posible de la operación.
- ¿La operación tiene un comienzo y un fin? Hay que descartar las operaciones que implican una responsabilidad continua (por ejemplo, "mantener actualizada la información del empleado") y tratar de buscar operaciones puntuales que correspondan a acciones que puedan ser hechas por el usuario.

> Un requerimiento funcional se puede ver como un servicio que el programa le ofrece al usuario para resolver una parte del problema.

Ejemplo 3

Objetivo: Ilustrar la manera de documentar los requerimientos funcionales de un problema.

A continuación se presenta otro requerimiento funcional del caso de estudio del empleado, para el cual se especifican los cuatro elementos que lo componen.

Nombre	**R2: Cambiar el empleado**	**Asociamos el identificador R2 con el requerimiento. En la mayoría de los casos el identificador del requerimiento se asigna siguiendo alguna convención definida por la empresa de desarrollo. Utilizamos el verbo "cambiar" para describir la operación que se quiere hacer.**
Resumen	Permite al usuario cambiar la información del empleado: datos personales y datos de vinculación a la empresa.	Describimos la operación, dando una idea global del tipo de información que se debe ingresar y del resultado obtenido.
Entradas	1) Nombre del empleado. 2) Apellido del empleado. 3) Género del empleado. 4) Fecha de nacimiento. 5) Fecha de ingreso a la compañía. 6) Salario básico. 6) Imagen del empleado.	En este caso se necesitan siete entradas para poder realizar el requerimiento. Esta información la debe proveer el usuario al programa. Note que no se define la manera en que dicha información será ingresada por el usuario, puesto que eso va a depender del diseño que se haga de la interfaz, y será una decisión que se tomará más tarde en el proceso de desarrollo. Fíjese que tampoco se habla del formato en el que va a entrar la información. Por ahora sólo se necesita entender, de manera global, lo que el cliente quiere que el programa sea capaz de hacer.
Resultado	La información del empleado ha sido actualizada.	La operación corresponde de nuevo a una modificación de algún valor del mundo, puesto que con la información obtenida como entrada se quieren modificar los datos del empleado.

Tarea 2

Objetivo: Crear habilidad en la identificación y especificación de requerimientos funcionales. Para el caso de estudio 2, un simulador bancario, identifique y especifique tres requerimientos funcionales.

Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

Tarea 3

Objetivo: Crear habilidad en la identificación y especificación de requerimientos funcionales.

Para el caso de estudio 3, un programa para manejar un triángulo, identifique y especifique tres requerimientos funcionales.

Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

5.2. El Modelo del Mundo del Problema

En este segundo componente del análisis, el objetivo es entender el mundo en el que ocurre el problema y recopilar toda la información necesaria para que el programador pueda escribir el programa. Suponga por ejemplo que existe un requerimiento de calcular los días de vacaciones a los que tiene derecho el empleado. Si durante la etapa de análisis no se recoge la información de la empresa que hace referencia a la manera de calcular el número de días de vacaciones a los cuales un empleado tiene derecho, cuando el programador trate de resolver el problema se va a dar cuenta de que no tiene toda la información que

necesita. Ya no nos vamos a concentrar en las opciones que el cliente quiere que tenga el programa, sino nos vamos a concentrar en entender cómo es el mundo en el que ocurre el problema. En el caso de estudio del empleado, el objetivo de esta parte sería entender y especificar los aspectos relevantes de la empresa.

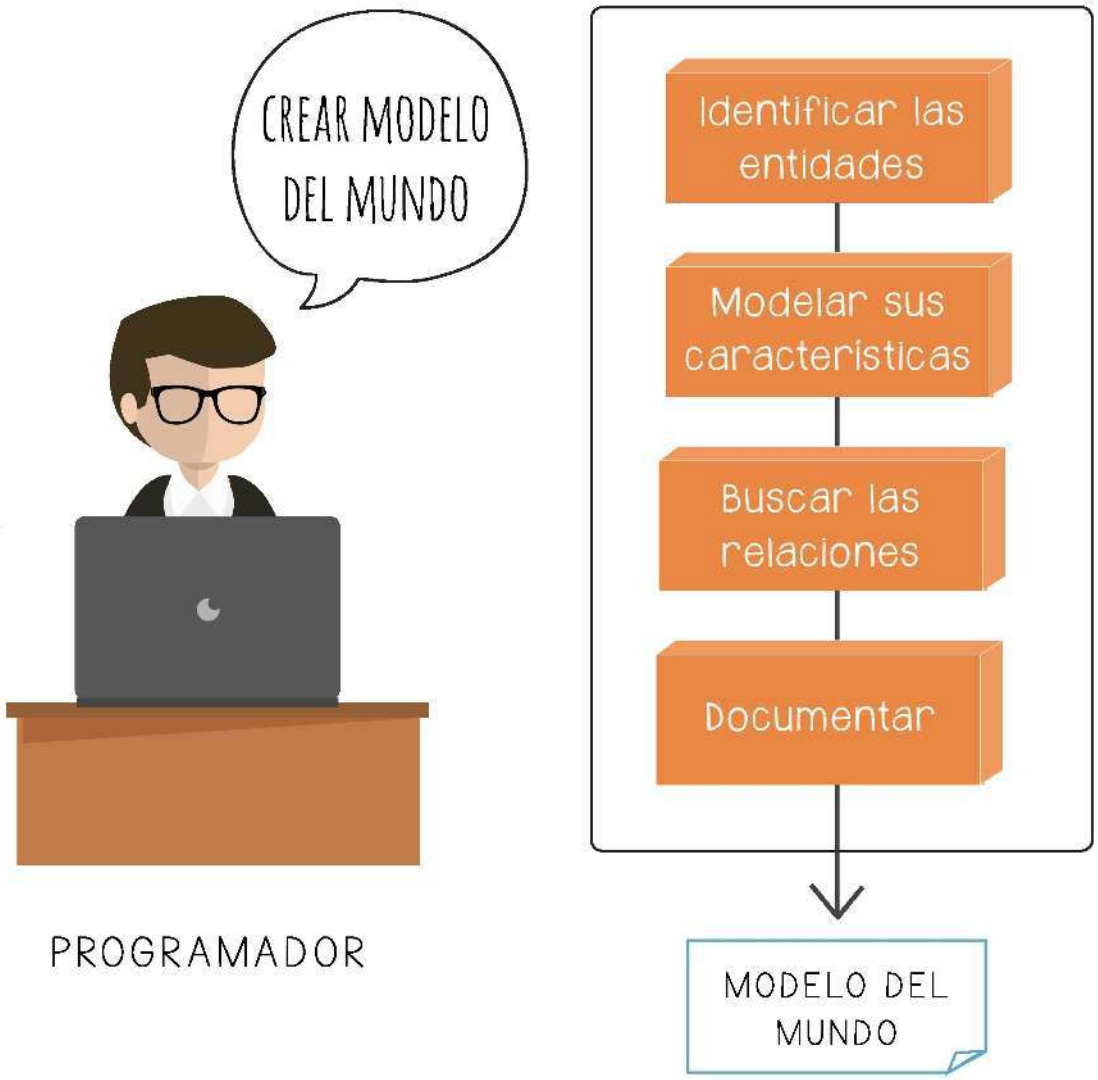
Como salida de esta actividad, se debe producir lo que se denomina un **modelo del mundo** del problema, en el cual hayamos identificado todos los elementos del mundo que participan en el problema, con sus principales características y relaciones. Este modelo será utilizado en la etapa de diseño para definir los elementos básicos del programa.

Esta actividad está basada en un proceso de "observación" del mundo del problema, puesto que los elementos que allí aparecen ya existen y nuestro objetivo no es opinar sobre ellos (o proponer cambiarlos), sino simplemente identificarlos y describirlos para que esta información sea utilizada más adelante.

En la figura 1.6 se resumen las cuatro actividades que debe realizar el programador para construir el modelo del mundo. En la primera, se identifica lo que denominamos las **entidades** del mundo, en la segunda se documentan las **características** de cada una de ellas, en la tercera se definen las **relaciones** que existen entre las distintas entidades y, finalmente, se documenta la información adicional (reglas, restricciones, etc.) que se tenga sobre las entidades.

Para expresar el modelo del mundo utilizaremos la sintaxis definida en el diagrama de clases del lenguaje de modelos UML (Unified Modeling Language). Dicho lenguaje es un estándar definido por una organización llamada OMG (Object Management Group) y utilizado por una gran cantidad de empresas en el mundo para expresar sus modelos.

Fig. 1.6 Actividades en la elaboración del modelo del mundo



5.2.1. Identificar las Entidades

Esta primera actividad tiene como objetivo identificar los elementos del mundo que intervienen en el problema. Dichos elementos pueden ser concretos (una persona, un vehículo) o abstractos (una cuenta bancaria). Por ahora únicamente queremos identificar estos elementos y asociarles un nombre significativo. Una primera pista para localizarlos es buscar los sustantivos del enunciado del problema. Esto sirve en el caso de problemas pequeños, pero no es generalizable a problemas de mayor dimensión.

En programación orientada a objetos, las entidades del mundo se denominan **clases**, y serán los elementos básicos del diseño y la posterior implementación.

> Una convención es una regla que no es obligatoria en el lenguaje de programación, pero que suelen respetar los programadores que utilizan el lenguaje. Por ejemplo, por convención, los nombres de las clases comienzan por mayúsculas.
>
> Seguir las convenciones hace que sea más fácil entender los programas escritos por otras personas. También ayuda a construir programas más "elegantes".

Para el primer caso de estudio, hay dos entidades en el mundo del problema: la clase Empleado y la clase Fecha. Esta última se emplea para representar el concepto de fecha de nacimiento y fecha de ingreso a la empresa. Si lee con detenimiento el enunciado del caso, se podrá dar cuenta de que éstos son los únicos elementos del mundo del problema que se mencionan. Lo demás corresponde a características de dichas entidades (el nombre, el apellido, etc.) o a requerimientos funcionales.

En el ejemplo 4 se identifican las entidades del caso de estudio del simulador bancario y se describe el proceso que se siguió para identificarlas.

Ejemplo 4

Objetivo: Ilustrar la manera de identificar las entidades (llamadas también clases) del mundo del problema.

En este ejemplo se identifican las entidades que forman parte del mundo del problema para el caso 2 de este nivel: un simulador bancario.

Entidad	Descripción
SimuladorBancario	Es la entidad más importante del mundo del problema, puesto que define su frontera (todo lo que está por fuera de la cuenta bancaria no nos interesa). Es buena práctica comenzar la etapa de análisis tratando de identificar la clase más importante del problema. Cuando el nombre de la entidad es compuesto, se usa por convención una letra mayúscula al comienzo de cada palabra. En otra época se utilizaban el carácter "_" para separar las palabras (Cuenta_Bancaria) pero eso está pasado de moda.
CuentaCorriente	Este es otro concepto que existe en el mundo del problema. Según el enunciado una cuenta corriente forma parte de una cuenta bancaria, luego esta entidad está "dentro" de la frontera que nos interesa. Por ahora no nos interesan los detalles de la cuenta corriente (por ejemplo si tiene un saldo o si paga intereses). En este momento sólo queremos identificar los elementos del mundo del problema que están involucrados en los requerimientos funcionales.
CuentaAhorros	Este es el tercer concepto que aparece en el mundo del problema. De la misma manera que en el caso anterior, una cuenta bancaria "incluye" una cuenta de ahorros. Los nombres asignados a las clases deben ser significativos y dar una idea clara de la entidad del mundo que representan. No se debe exagerar con la longitud del nombre, porque de lo contrario los programas pueden resultar pesados de leer.
CDT	El nombre de esta clase se encuentra en mayúsculas, porque es una sigla. Otro nombre para esta clase habría podido ser el nombre completo del concepto: CertificadoDepositoTermino. En el lenguaje Java no es posible usar tildes en los nombres de los clases, así que nunca veremos una clase llamada CertificadoDepósitoTérmino.

Tarea 4

Objetivo: Identificar las entidades del mundo para el caso de estudio 3: un programa que maneje un triángulo.

Lea el enunciado del caso y trate de guiarse por los sustantivos para identificar las entidades del mundo del problema.

	Nombre	Descripción
Entidad		
Entidad		
Entidad		

Punto de reflexión: ¿Qué pasa si no identificamos bien las entidades del mundo?

Punto de reflexión: ¿Cómo decidir si se trata efectivamente de una entidad y no sólo de una característica de una entidad ya identificada?

5.2.2. Modelar las Características

Una vez que se han identificado las entidades del mundo del problema, el siguiente paso es identificar y modelar sus características. A cada característica que vayamos encontrando, le debemos asociar (1) un nombre significativo y (2) una descripción del conjunto de valores que dicha característica puede tomar.

En programación orientada a objetos, las características se denominan **atributos** y, al igual que las clases, serán elementos fundamentales tanto en el diseño como en la implementación. El nombre de un atributo debe ser una cadena de caracteres no vacía, que empiece con una letra y que no contenga espacios en blanco.

> Por convención, el nombre de los atributos comienza por una letra minúscula. Si es un nombre compuesto, se debe iniciar cada palabra simple con mayúscula.

En el lenguaje UML, una clase se dibuja como un cuadrado con tres zonas (ver ejemplo 5): la primera de ellas con el nombre de la clase y, la segunda, con los atributos de la misma. El uso de la tercera zona la veremos más adelante, en la etapa de diseño.

Ejemplo 5

Objetivo: Mostrar la manera de identificar y modelar los atributos de una clase.

En este ejemplo se identifican las características de las clases Empleado y Fecha para el caso de estudio del empleado.

Clase: Empleado

Atributo	Valores Posibles	Comentarios
nombre	Cadena de caracteres	La primera característica que aparece en el enunciado es el nombre del empleado. El valor de este atributo es una cadena de caracteres (por ejemplo, "Juan"). Seleccionamos "nombre" como nombre del atributo. Es importante que los nombres de los atributos sean significativos (deben dar una idea clara de lo que una característica representa), para facilitar así la lectura y la escritura de los programas.
apellido	Cadena de caracteres	El segundo atributo es el apellido del empleado. Al igual que en el caso anterior, el valor que puede tomar este atributo es una cadena de caracteres (por ejemplo, "Pérez"). Como nombre del atributo seleccionamos "apellido". El nombre de un atributo debe ser único dentro de la clase (no es posible dar el mismo nombre a dos atributos).
género	Masculino o Femenino	Esta característica puede tomar dos valores: masculino o femenino. En esta etapa de análisis basta con identificar los valores posibles. Es importante destacar que los valores posibles de este atributo (llamado "género") no son cadenas de caracteres. No nos interesan las palabras en español que pueden describir los valores posibles de esta característica, sino los valores en sí mismos.
salario	Valores reales positivos	El salario está expresado en pesos y su valor es un número real positivo.

Clase: Fecha

Atributo	Valores Posibles	Comentarios
dia	Valores enteros entre 1 y 31	La primera característica de una fecha es el día y puede tomar valores enteros entre 1 y 31. En los nombres de las variables no puede haber tildes, por lo que debemos contentarnos con el nombre "dia" (sin tilde) para el atributo.
mes	Valores enteros entre 1 y 12	La segunda característica es el mes. Aquí se podrían listar los meses del año como los valores posibles (por ejemplo, enero, febrero, etc.), pero por simplicidad vamos a decir que el mes corresponde a un valor entero entre 1 y 12.
anio	Valores enteros positivos	La última característica es el año. Debe ser un valor entero positivo (por ejemplo, 2001). Aquí nos encontramos de nuevo con un problema en español: los nombres de los atributos no pueden contener la letra "ñ". En este caso resolvimos reemplazar dicha letra y llamar el atributo "anio" que da aproximadamente el mismo sonido.

Con las tres características anteriores queda completamente definida una fecha. Esa es la pregunta que nos debemos hacer cuando estamos en esta etapa: ¿es necesaria más información para describir la entidad que estamos representando? Si encontramos una

característica cuyos valores posibles no son simples, como números, cadenas de caracteres, o una lista de valores, nos debemos preguntar si dicha característica no es más bien otra entidad que no identificamos en la etapa anterior. Si es el caso, simplemente la debemos agregar.

> Es importante que antes de agregar un atributo a una clase, verifiquemos que dicha característica forma parte del problema que se quiere resolver. Podríamos pensar, por ejemplo, que la ciudad en la que nació el empleado es uno de sus atributos. ¿Cómo saber si lo debemos o no agregar? La respuesta es que hay que mirar los requerimientos funcionales y ver si dicha característica es utilizada o referenciada desde alguno de ellos.

Tarea 5

Para cada una de las cuatro entidades identificadas en el caso de estudio del simulador bancario, identifique los atributos, sus valores posibles, y escriba la clase en UML. No incluya las relaciones que puedan existir entre las clases, ya que eso lo haremos en la siguiente etapa del análisis. Por ahora trate de identificar las características de las entidades que son importantes para los requerimientos funcionales.

Clase: SimuladorBancario

Atributo	Valores Posibles

Diagrama UML:

Clase: CuentaCorriente

Atributo	Valores Posibles

Diagrama UML:

Clase: CuentaAhorros

Atributo	Valores Posibles

Diagrama UML:

Clase: CDT

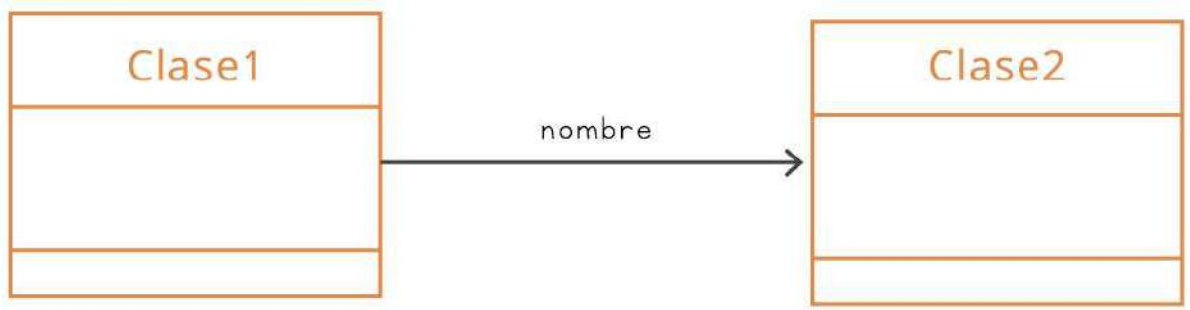
Atributo	Valores Posibles

Diagrama UML:

5.2.3. Las Relaciones entre las Entidades

En esta actividad, debemos tratar de identificar las relaciones que existen entre las distintas entidades del mundo y asignarles un nombre. Las relaciones se representan en UML como flechas que unen las cajas de las clases (ver figura 1.7) y se denominan usualmente **asociaciones**. El diagrama de clases en el cual se incluye la representación de todas las entidades y las relaciones que existen entre ellas se conoce como el **modelo conceptual**, porque explica la estructura y las relaciones de los elementos del mundo del problema.

Fig. 1.7 Sintaxis en UML para mostrar una asociación entre dos clases

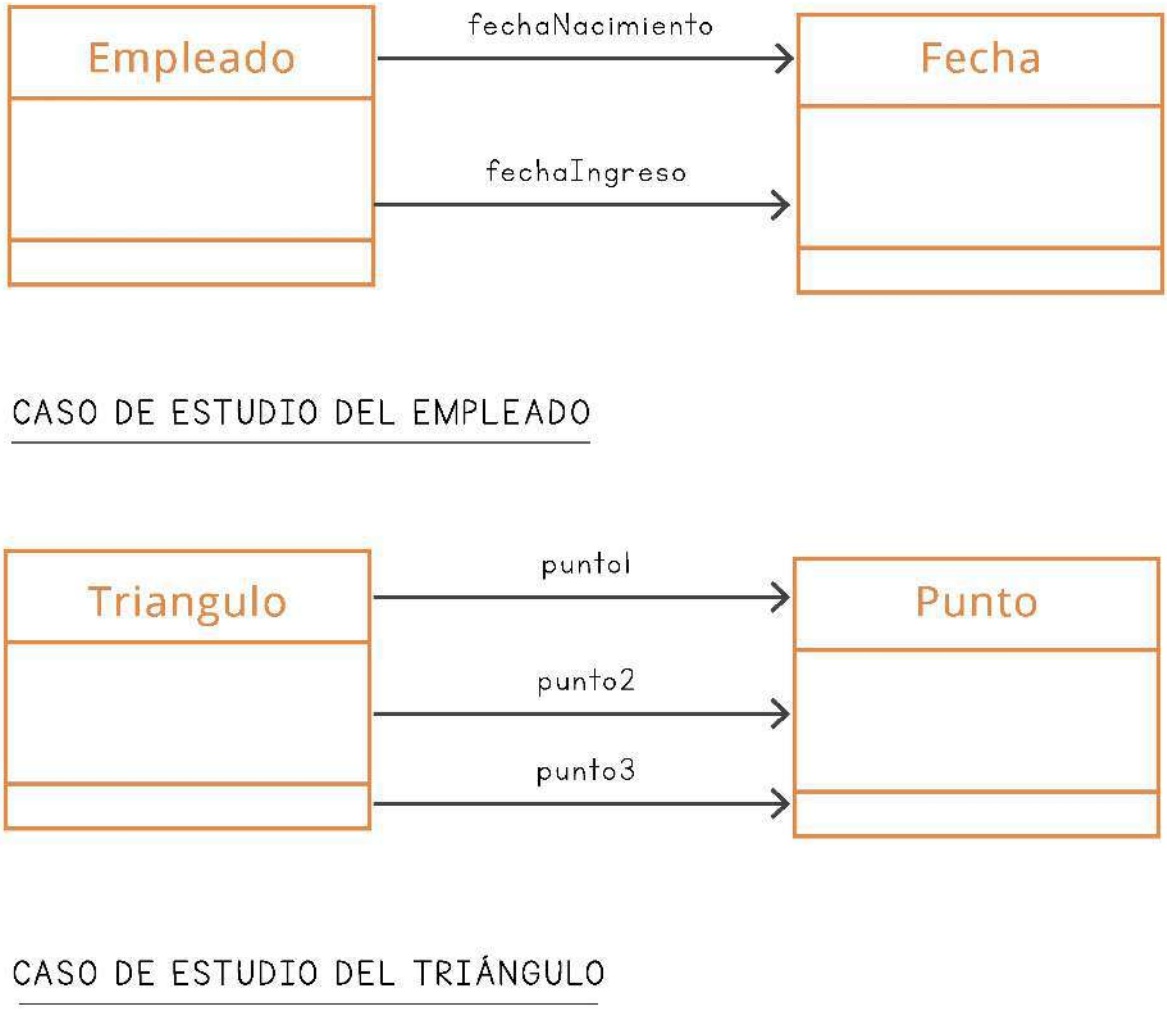


- El modelo presentado en la figura dice que hay dos entidades en el mundo (llamadas Clase1 y Clase2), y que existe una relación entre ellas.
- También explica que para la Clase1, la Clase2 representa algo que puede ser descrito con el nombre que se coloca al final de la asociación. La selección de dicho nombre es fundamental para la claridad del diagrama.

> El nombre de la asociación sigue las mismas convenciones del nombre de los atributos y debe reflejar la manera en que una clase utiliza a la otra como parte de sus características.

Es posible tener varias relaciones entre dos clases, y por eso es importante seleccionar bien el nombre de cada asociación. En la figura 1.8 se muestran las asociaciones entre las clases del caso de estudio del empleado y del caso de estudio del triángulo. En los dos casos existe más de una asociación entre las clases, cada una de las cuales modela una característica diferente.

Fig. 1.8 Diagrama de clases para representar el modelo del mundo



Caso de Estudio del Empleado:

- La primera asociación dice que un empleado tiene una fecha de nacimiento y que esta fecha es una entidad del mundo, representada por la clase Fecha.
- La segunda asociación hace lo mismo con la fecha de ingreso del empleado a la empresa.
- La dirección de la flecha indica la entidad que "contiene" a la otra. El empleado tiene una fecha, pero la fecha no tiene un empleado.

Caso de Estudio del Triángulo:

- Un triángulo tiene tres puntos, cada uno de los cuales define una de sus aristas. Cada punto tiene un nombre distinto (punto1, punto2 y punto3), el cual se asigna a la asociación.
- Note que este diagrama está incompleto, puesto que no aparece la clase Color (para representar el color de las líneas y el relleno del triángulo), ni las asociaciones hacia ella.

- La clase Punto seguramente tiene dos atributos para representar las coordenadas en cada uno de los ejes, pero eso no lo incluimos en el diagrama para simplificarlo.

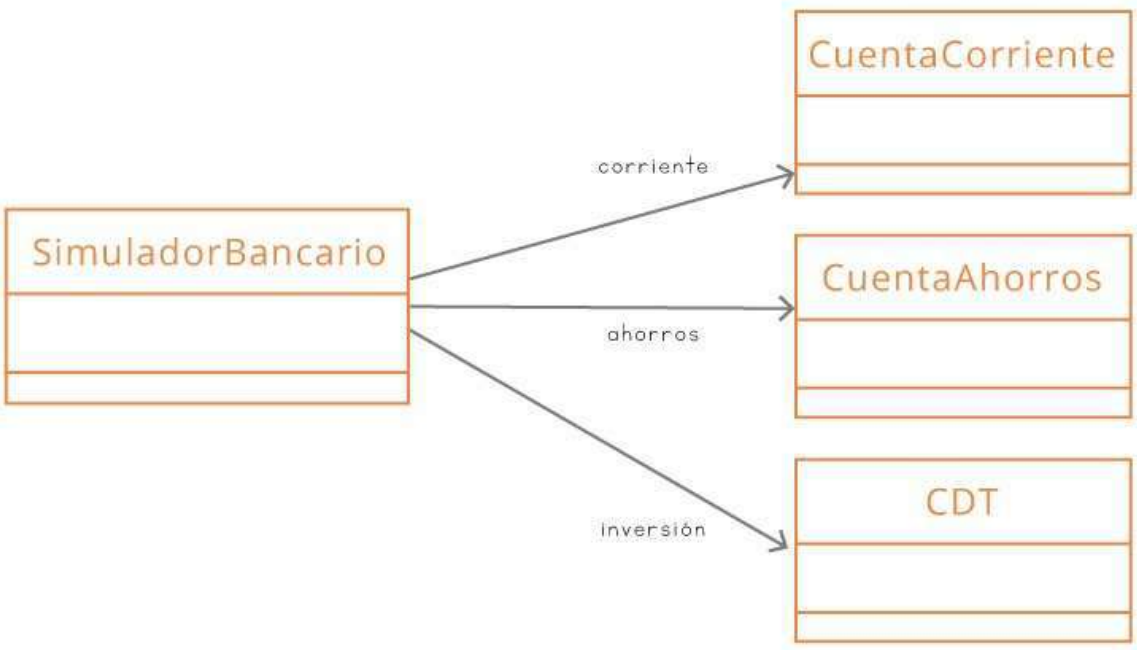
Volveremos a abordar el tema de las relaciones entre entidades en los niveles posteriores, así que por ahora sólo es importante poder identificar las relaciones para casos muy simples. En el ejemplo 6 se muestran y explican las relaciones que existen entre las entidades del caso del simulador bancario.

> Una asociación se puede ver como una característica de una entidad cuyo valor está representado por otra clase.

Ejemplo 6

Objetivo: Presentar el diagrama de clases, como una manera de mostrar el modelo de una realidad.

A continuación se muestra el diagrama de clases del modelo del mundo, para el caso del simulador
bancario.



- La relación entre la clase SimuladorBancario y la clase CuentaCorriente se llama

"corriente" y refleja el hecho de que una cuenta bancaria tiene una cuenta corriente como parte de ella.
- Fíjese que las flechas tienen una dirección. Dicha dirección establece qué entidad utiliza a la otra como parte de sus características.
- Si lee de nuevo el enunciado, se dará cuenta de que el diagrama de clases se limita a expresar lo mismo que allí aparece, pero usando una sintaxis gráfica, que tiene la ventaja de no ser ambigua.

5.3. Los Requerimientos no Funcionales

En la mayoría de los casos, la solución que se va a construir debe tener en cuenta las restricciones definidas por el cliente, que dependen, en gran medida, del contexto de utilización del programa. Para el caso del empleado, por ejemplo, el cliente podría pedir que el programa se pueda usar a través de un teléfono celular, o desde un navegador de Internet, o que el tiempo de respuesta de cualquier consulta sea menor a 0,0001 segundos.

Los requerimientos no funcionales están muchas veces relacionados con restricciones sobre la tecnología que se debe usar, el volumen de los datos que se debe manejar o la cantidad de usuarios. Para problemas grandes, los requerimientos no funcionales son la base para el diseño del programa. Piense en lo distinto que será un programa que debe trabajar con un único usuario, de otro que debe funcionar con miles de ellos simultáneamente.

En el contexto de este libro, dados los objetivos y el tamaño de los problemas, sólo vamos a considerar los requerimientos no funcionales de interacción y visualización, que están ligados con la interfaz de los programas.

> En este punto el lector debería ser capaz de leer el enunciado de un problema sencillo y, a partir de éste, (1) especificar los requerimientos funcionales, (2) crear el modelo del mundo del problema usando UML y (3) listar los requerimientos no funcionales.

6. Elementos de un Programa

En esta parte del capítulo presentamos los distintos elementos que forman parte de un programa. No pretende ser una exposición exhaustiva, pero sí es nuestro objetivo dar una visión global de los distintos aspectos que intervienen en un programa.

En algunos casos la presentación de los conceptos es muy superficial. Ya nos tomaremos el tiempo en los niveles posteriores de profundizar poco a poco en cada uno de ellos. Por ahora lo único importante es poderlos usar en casos limitados. Esta manera de presentar los temas nos va a permitir generar las habilidades de uso de manera incremental, sin necesidad de estudiar toda la teoría ligada a un concepto antes de poder usarlo.

6.1. Algoritmos e Instrucciones

Los algoritmos son uno de los elementos esenciales de un programa. Un **algoritmo** se puede ver como la solución de un problema muy preciso y pequeño, en el cual se define la secuencia de instrucciones que se debe seguir para resolverlo. Imagine, entonces, un programa como un conjunto de algoritmos, cada uno responsable de una parte de la solución del problema global.

Un algoritmo, en general, es una secuencia ordenada de pasos para realizar una actividad. Suponga, por ejemplo, que le vamos a explicar a alguien lo que debe hacer para viajar en el metro parisino. El siguiente es un algoritmo de lo que esta persona debe hacer para llegar a una dirección dada:

1. Compre un tiquete de viaje en los puntos de venta que se encuentran a la entrada de cada una de las estaciones del metro.
2. Identifique en el mapa del metro la estación donde está y el punto adonde necesita ir.
3. Localice el nombre de la estación de metro más cercana al lugar de destino.
4. Verifique si, a partir de donde está, hay alguna línea que pase por la estación destino.
5. Si encontró la línea, busque el nombre de la misma en la dirección de destino.
6. Suba al metro en el andén de la línea identificada en el paso anterior y bájese en la estación de destino.

Tarea 6

Objetivo: Reflexionar sobre el nivel de precisión que debe ser usado en un algoritmo para evitar ambigüedades.

Suponga que usted es la persona que va a utilizar el algoritmo anterior, para moverse en el metro de París. Identifique qué problemas podría tener con las instrucciones anteriores. Piense por ejemplo si están completas.

¿Se prestan para que se interpreten de maneras distintas? ¿Estamos suponiendo que quién lo lee usa su "sentido común", o cualquier persona que lo use va a resolver siempre el problema de la misma manera?

Utilice este espacio para anotar sus conclusiones:


Tarea 7

Objetivo: Entender la complejidad que tiene la tarea de escribir un algoritmo.

Esta tarea es para ser desarrollada en parejas:

1. En el primer cuadrante haga un dibujo simple.
2. En el segundo cuadrante escriba las instrucciones para explicarle a la otra persona cómo hacer el dibujo.
3. Lea las instrucciones a la otra persona, quien debe intentar seguirlas sin ninguna ayuda adicional.
4. Compare el dibujo inicial y el dibujo resultante.

Dibujo:	**Algoritmo**
	

Haga una síntesis de los resultados obtenidos:

Cuando es el computador el que sigue un algoritmo (en el caso del computador se habla de **ejecutar**), es evidente que las instrucciones que le demos no pueden ser como las definidas en el algoritmo del metro de París. Dado que el computador no tiene nada parecido al "sentido común", las instrucciones que le definamos deben estar escritas en un lenguaje que no dé espacio a ninguna ambigüedad (imaginemos al computador de una nave espacial diciendo "es que yo creí que eso era lo que ustedes querían que yo hiciera"). Por esta razón los algoritmos que constituyen la solución de un problema se deben traducir a un lenguaje increíblemente restringido y limitado (pero a su vez poderoso si vemos todo lo que con él podemos hacer), denominado un **lenguaje de programación**. Todo lenguaje de programación tiene su propio conjunto de reglas para decir las cosas, denominado la **sintaxis** del lenguaje.

Existen muchos lenguajes de programación en el mundo, cada uno con sus propias características y ventajas. Como dijimos anteriormente, en este libro utilizaremos el lenguaje de programación Java que es un lenguaje de propósito general (no fue escrito para resolver problemas en un dominio específico), muy utilizado hoy en día en el mundo entero, tanto a nivel científico como empresarial.

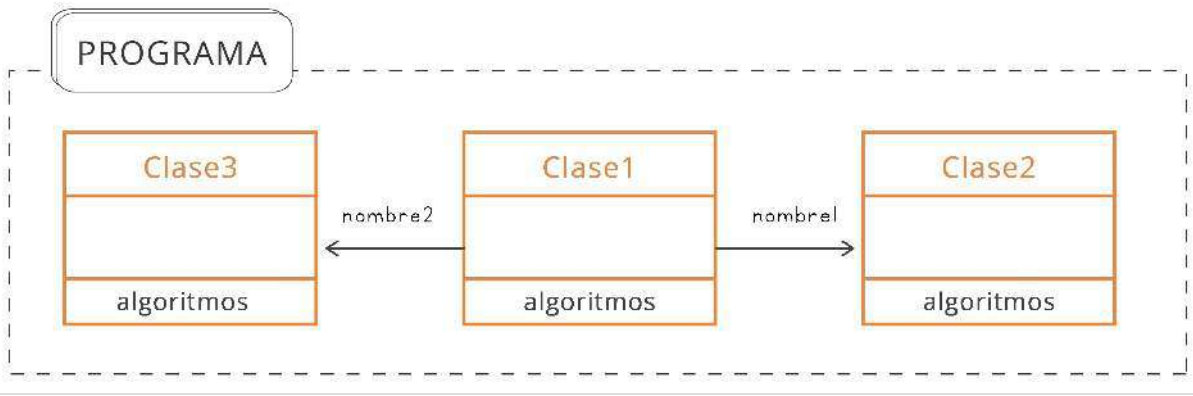
> Un programa de computador está compuesto por un conjunto de algoritmos, escritos en un lenguaje de programación. Dichos algoritmos están estructurados de tal forma que, en conjunto, son capaces de resolver el problema.

6.2. Clases y Objetos

Las clases son los elementos que definen la estructura de un programa. Tal como vimos en la etapa de análisis, las clases representan entidades del mundo del problema (más adelante veremos que también pueden pertenecer a lo que denominaremos el mundo de la solución). Por ahora, y para que se pueda dar una idea de lo que es un programa completo,

imagine que los algoritmos están dentro de las clases, y que son estas últimas las que establecen la manera en que los algoritmos colaboran para resolver el problema global (ver figura 1.9). Esta visión la iremos refinando a medida que avancemos en el libro, pero por ahora es suficiente para comenzar a trabajar.

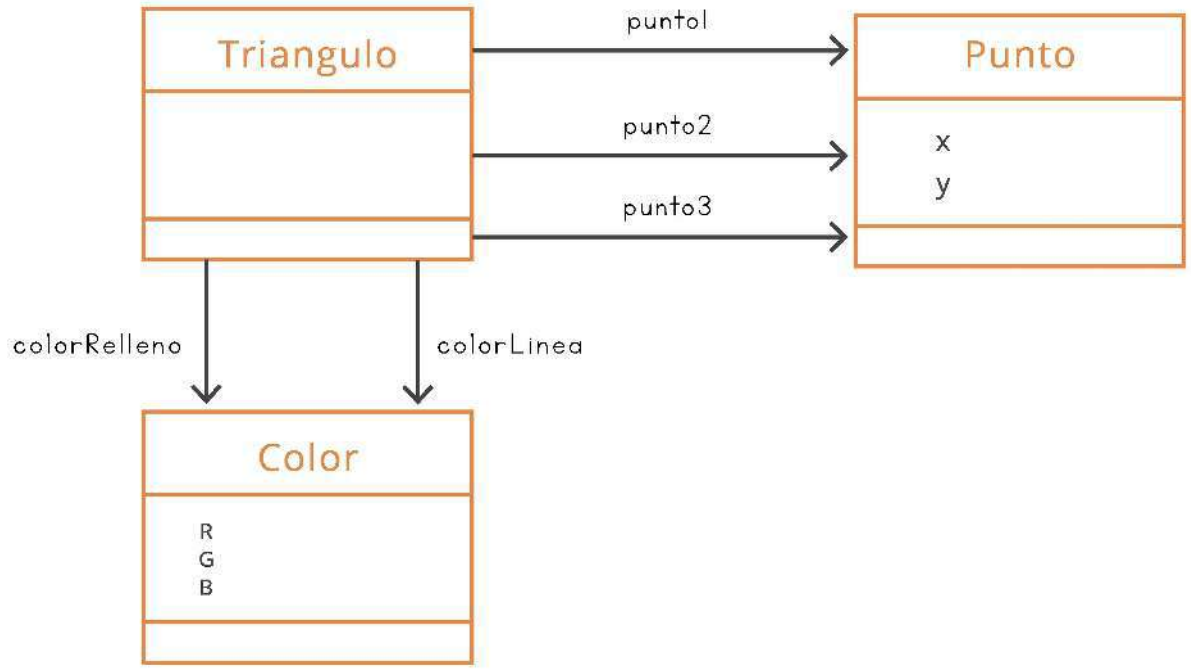
Fig. 1.9 Visión intuitiva de la estructura de un programa



Hasta ahora es claro que en un programa hay una clase por cada entidad del mundo del problema. Pero, ¿qué pasa si hay varias "instancias" (es decir, varios ejemplares) de alguna de esas entidades? Piense por ejemplo que en vez de crear un programa para manejar un empleado, como en el primer caso de estudio, resolvemos hacer un programa para manejar todos los empleados de una empresa. Aunque todos los empleados tienen las mismas características (nombre, apellido, etc.), cada uno tiene valores distintos para ellas (cada uno va a tener un nombre y un apellido distinto). Es aquí donde aparece el concepto de **objeto**, la base de toda la programación orientada a objetos. Un objeto es una instancia de una clase (la cual define los atributos que debe tener) que tiene sus propios valores para cada uno de los atributos. El conjunto de valores de los atributos se denomina el **estado del objeto**. Para diferenciar las clases de los objetos, se puede decir que una clase define un tipo de elemento del mundo, mientras que un objeto representa un elemento individual.

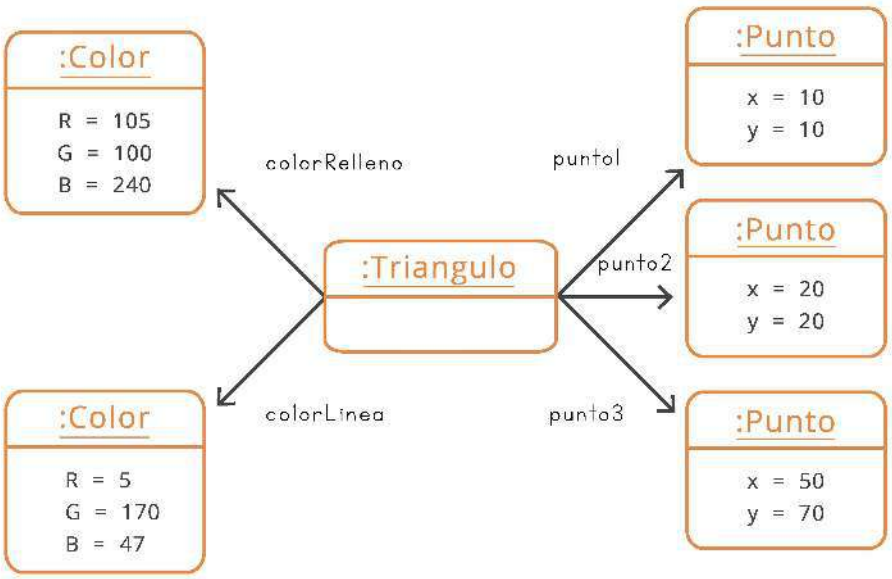
Piense por ejemplo en el caso del triángulo. Cada uno de los puntos que definen las aristas de la figura geométrica son objetos distintos, todos pertenecientes a la clase Punto. En la figura 1.10 se ilustra la diferencia entre clase y objeto para el caso del triángulo. Fíjese que la clase Punto dice que todos los objetos de esa clase deben tener dos atributos (x, y), pero son sus instancias las que tienen los valores para esas dos características.

Fig. 1.10a Diferencia entre clases y objetos para el caso de estudio del triángulo



- La clase Triangulo tiene tres asociaciones hacia la clase Punto (`punto1` , `punto2` y `punto3`). Eso quiere decir que cada objeto de la clase Triangulo tendrá tres objetos asociados, cada uno de ellos perteneciente a la clase Punto.
- Lo mismo sucede con las dos asociaciones hacia la clase Color: debe haber dos objetos de la clase Color por cada objeto de la clase Triangulo.
- Cada triángulo será entonces representado por 6 objetos conectados entre sí: uno de la clase Triangulo, tres de la clase Punto y dos de la clase Color.

Fig. 1.10b Diferencia entre clases y objetos para el caso de estudio del triángulo



- Cada uno de los objetos tiene asociado el nombre que se definió en el diagrama de clases.
- El primer punto del triángulo está en las coordenadas (10, 10).
- El segundo punto del triángulo está en las coordenadas (20, 20).
- El tercer punto del triángulo está en las coordenadas (50, 70).
- Las líneas del triángulo son del color definido por el código RGB de valor (5, 170, 47). ¿A qué color corresponde ese código?
- Este es sólo un ejemplo de todos los triángulos que podrían definirse a partir del diagrama de clases.
- En la parte superior de cada objeto aparece la clase a la cual pertenece.

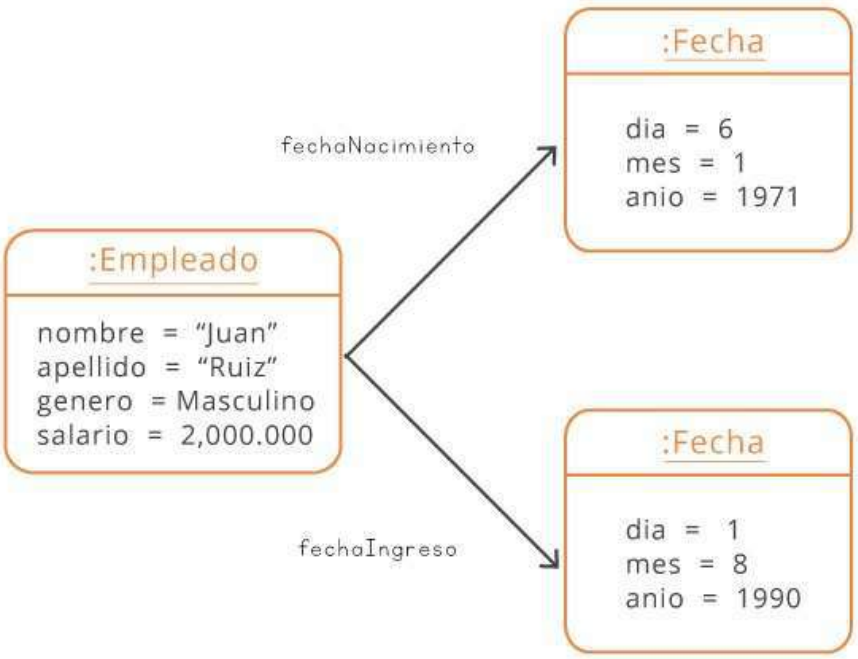
Para representar los objetos vamos a utilizar la sintaxis propuesta en UML (diagrama de objetos), que consiste en cajas con bordes redondeados, en la cual hay un valor asociado con cada atributo. Podemos pensar en un diagrama de objetos como un ejemplo de los objetos que se pueden construir a partir de la definición de un diagrama de clases. En el ejemplo 7 se ilustra la manera de visualizar un conjunto de objetos para el caso del empleado.

Ejemplo 7

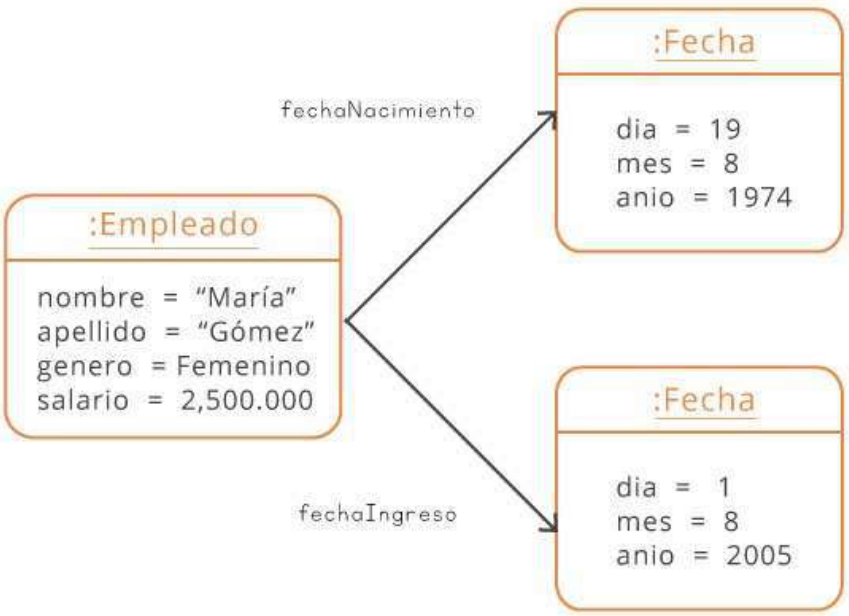
Objetivo: Ilustrar utilizando una extensión del caso de estudio 1 la diferencia entre los conceptos de clase y objeto.

La extensión consiste en suponer que el programa debe manejar todos los empleados de una empresa, en lugar de uno solo de ellos.

- Cada objeto de la clase Empleado tendrá un valor para cada uno de sus atributos y un objeto para cada una de sus asociaciones.
- Esta clase define los atributos de todos los empleados de la empresa.
- De manera intuitiva, una clase puede verse como un molde a partir del cuál sus objetos son construidos.
- Cada empleado será representado con tres objetos: uno de la clase Empleado y dos de la clase Fecha.



- Este es el primer ejemplo de un empleado de la empresa. Se llama Juan Ruiz, nació el 6 de enero de 1971, comenzó a trabajar en la empresa el 1 de agosto de 1990 y su salario es de dos millones de pesos.
- Durante la ejecución de un programa pueden aparecer tantos objetos como sean necesarios, para representar el mundo del problema. Si en la empresa hay 500 empleados, en la ejecución del programa habrá 1500 objetos representándolos (3 objetos por empleado).



- Este grupo de objetos representa otro empleado de la empresa.
- Note que cada empleado tiene sus propios valores para los atributos y que lo único que comparten los dos empleados es la clase a la cual pertenecen, la cual establece la lista de atributos que deben tener.

6.3. Java como Lenguaje de Programación

Existen muchos lenguajes de programación en el mundo. Los hay de distintos tipos, cada uno adaptado a resolver distintos tipos de problemas. Tenemos los lenguajes funcionales como LISP o CML, los lenguajes imperativos como C, PASCAL o BASIC, los lenguajes lógicos como PROLOG y los lenguajes orientados a objetos como Java, C# y SMALLTALK.

Java es un lenguaje creado por Sun Microsystems en 1995, muy utilizado en la actualidad en todo el mundo, sobre todo gracias a su independencia de la plataforma en la que se ejecuta. Java es un lenguaje de propósito general, con el cual se pueden desarrollar desde pequeños programas para resolver problemas simples hasta grandes aplicaciones industriales o de apoyo a la investigación.

En esta sección comenzamos a estudiar la manera de expresar en el lenguaje Java los elementos identificados hasta ahora. Comenzamos por las clases, que son los elementos fundamentales de todos los lenguajes orientados a objetos. Lo primero que debemos decir es que un programa en Java está formado por un conjunto de clases, cada una de ellas descrita siguiendo las reglas sintácticas exigidas por el lenguaje.

Cada clase se debe guardar en un archivo distinto, cuyo nombre debe ser igual al nombre de la clase, y cuya extensión debe ser .java. Por ejemplo, la clase Empleado debe estar en el archivo Empleado.java y la clase Fecha en la clase Fecha.java.

> Un programa escrito en Java está formado por un conjunto de archivos, cada uno de los cuales contiene una clase. Para describir una clase en Java, se deben seguir de manera estricta las reglas sintácticas del lenguaje.

Ejemplo 8

Objetivo: Mostrar la sintaxis básica del lenguaje Java para declarar una clase.

Utilizamos el caso de estudio del empleado para introducir la sintaxis que se debe utilizar para declarar una clase.

Archivo: Empleado.java

Clase: Empleado

```
public class Empleado
{
    // Aquí va la declaración de la clase Empleado
}
```

Archivo: Fecha.java

Clase: Fecha

```
public class Fecha
{
    // Aquí va la declaración de la clase Fecha
}
```

En el lenguaje Java, todo lo que va entre dos corchetes ("{" y "}") se llama un bloque de instrucciones. En particular, entre los corchetes de la clase del ejemplo 8 va la **declaración** de la clase. Allí se deben hacer explícitos tanto los atributos como los algoritmos de la clase. También es posible agregar **comentarios**, que serán ignorados por el computador,

pero que le sirven al programador para indicar algo que considera importante dentro del código. En Java, una de las maneras de introducir un comentario es con los caracteres //, tal como se muestra en el ejemplo 8.

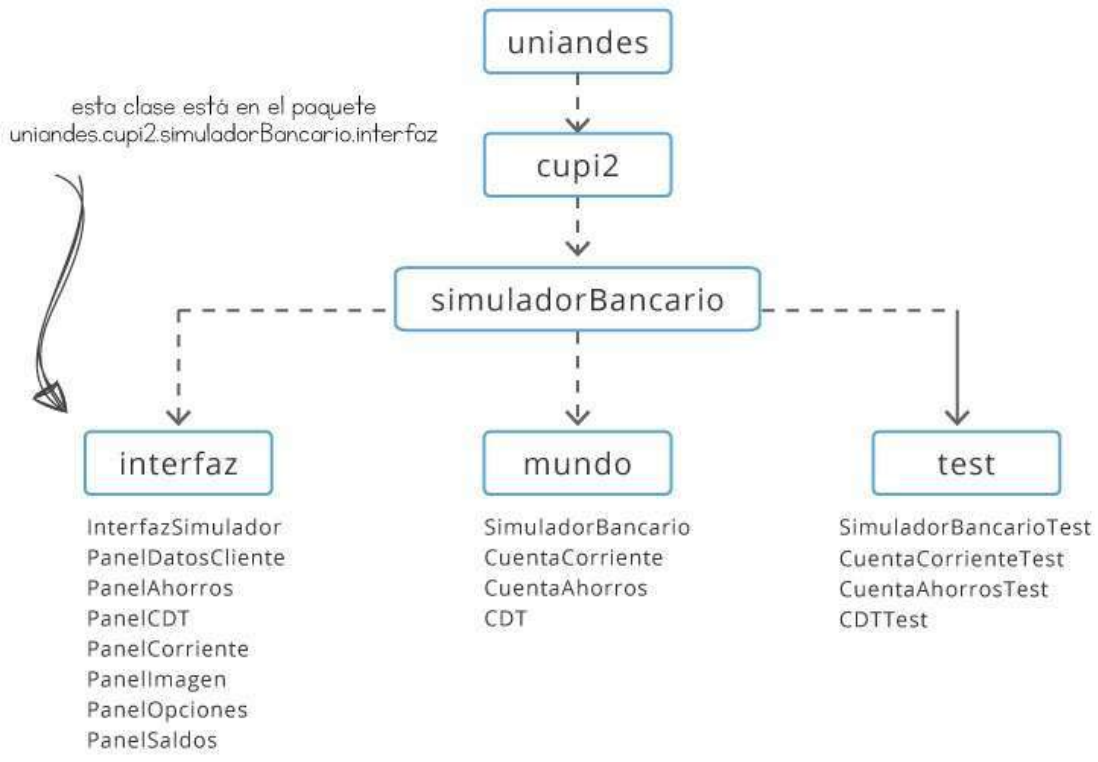
El programa del simulador bancario, por ejemplo, consta de 16 clases distribuidas de la siguiente manera:

- 4 clases para el modelo del mundo, almacenadas en los archivos SimuladorBancario.java, CuentaCorriente.java, CuentaAhorros.java y CDT.java.
- 8 clases para la interfaz usuario, en 8 archivos .java.
- 4 clases para las pruebas del programa, en 4 archivos .java.

Es aconsejable en este momento mirar en la sección 8 de este capítulo la localización de dichos archivos en la página web. Vale la pena también dar una mirada al contenido de los archivos que vamos mencionando en esta parte.

Puesto que un programa puede estar compuesto por miles de clases, Java tiene el concepto de paquete, a través del cual es posible estructurar las clases por grupos jerárquicos. Esto facilita su localización y manejo. En la figura 1.11 se muestra la estructura de paquetes del caso del simulador bancario.

Fig. 1.11 Ejemplo de la estructura de paquetes del caso de estudio del simulador bancario



- Las dieciséis clases del programa se dividen en 3 paquetes: uno con las clases de la interfaz de usuario (aquellas que implementan la ventana y los botones), uno con el modelo del mundo y un último paquete con las pruebas.
- El nombre completo de una clase es el nombre del paquete en el que se encuentra, seguido del nombre de la clase.

Toda clase en Java debe comenzar por la definición del paquete en el cual está situada la clase, como se muestra en el siguiente fragmento de programa del caso de estudio del empleado:

```
package uniandes.cupi2.empleado;

/**
 *    Esta clase representa un empleado
 */
public class Empleado
{

}
```

- El nombre del paquete es una secuencia de identificadores separados por un punto.
- uniandes.cupi2.empleado.Empleado es el nombre completo de la clase.
- En el momento de desarrollar un programa se deben establecer los paquetes que se van a utilizar. En nuestro caso, el nombre del paquete está conformado por el nombre de la institución (uniandes), seguida por el nombre del proyecto (cupi2) y luego el nombre del ejercicio del cual forma parte la clase (empleado).
- Cada empresa de desarrollo sigue sus propias convenciones para definir los nombres de los paquetes.

En todo lenguaje de programación existen las que se denominan **palabras reservadas**. Dichas palabras no las podemos utilizar para nombrar nuestras clases o atributos. Hasta el momento hemos visto las siguientes palabras reservadas: `package`, `public` y `class`.

> Un elemento de una clase se declara `public` cuando queremos que sea visible desde otras clases.

En el ejemplo anterior se puede apreciar otra manera de incluir un comentario dentro de un programa: se utilizan los símbolos `/**` para comenzar y los símbolos `*/` para terminar. El comentario puede extenderse por varios renglones sin ningún problema, a diferencia de los comentarios que comienzan por los símbolos `//` que terminan cuando se acaba el renglón. Los comentarios que se introducen como aparece en el ejemplo sirven para describir los principales elementos de una clase y tienen un uso especial que se verá más adelante en el libro.

6.4. Tipos de Datos

Cada lenguaje de programación cuenta con un conjunto de tipos de datos a través de los cuales el programador puede representar los atributos de una clase. En este nivel nos vamos a concentrar en dos tipos simples de datos: los enteros (tipo `int`), que permiten modelar características cuyos valores posibles son los valores numéricos de tipo entero (por ejemplo, el día en la clase Fecha), y los reales (tipo `double`), que permiten representar valores numéricos de tipo real (por ejemplo, el interés de una cuenta de ahorros). También vamos a estudiar un tipo de datos para manejar las cadenas de caracteres (tipo `String`), que permite representar dentro de una clase una característica como el nombre de una persona o una dirección. En los siguientes niveles, iremos introduciendo nuevos tipos de datos a medida que los vayamos necesitando.

En Java, en el momento de declarar un atributo, es necesario declarar el tipo de datos al cual corresponde, utilizando la sintaxis que se ilustra en el ejemplo que se muestra a continuación:

```
package uniandes.cupi2.empleado;

/**
* Esta clase representa un empleado
*/
public class Empleado
{
    //-------------------------------
    // Atributos
    //-------------------------------
    private String nombre;
    private String apellido;
    private double salario;
    ...
}
```

- Inicialmente se declaran los atributos nombre y apellido, de tipo `String` (cadenas de caracteres).
- Los atributos se declaran como privados (`private`) para evitar su manipulación desde fuera de la clase.
- El atributo salario se declara de tipo double, puesto que es un valor real.
- Con las tres declaraciones que aparecen en el ejemplo, el computador entiende que cualquier objeto de la clase Empleado debe tener valores para esas tres características.
- Sólo quedó pendiente por decidir el tipo del atributo genero, que no corresponde a ninguno de los tipos vistos; eso lo haremos más adelante.

Para modelar el atributo "genero", debemos utilizar alguno de los tipos de datos con los que cuenta el lenguaje. Lo mejor en este caso es utilizar un atributo de tipo entero y usar la convención de que si dicho atributo tiene el valor 1 se está representando un empleado con género masculino y, si es 2, un empleado con género femenino. Este proceso de asociar valores enteros y una convención para interpretarlos es algo que se hace cada vez que los valores posibles de un atributo no corresponden directamente con los de algún tipo de datos. Fíjese que una cosa es el valor que usamos (que es arbitrario) y otra la interpretación que hacemos de ese valor. Ese punto será profundizado en el nivel 2.

```
public class Empleado
{
    ...

    /**
     * 1 = masculino, 2 = femenino
     */
    private int genero;

    ...

}
```

- Al declarar un atributo para el cual se utilizó una convención especial para representar los valores posibles, es importante agregar un comentario en la declaración del mismo, explicando la interpretación que se debe dar a cada valor.
- En el ejemplo, decidimos representar con un 1 el valor masculino, y con un 2 el valor femenino.

> El tipo de un atributo determina el conjunto de valores que éste puede tomar dentro de los objetos de la clase, lo mismo que las operaciones que se van a poder hacer sobre dicha característica.

En el diagrama de clases de UML, por su parte, usamos una sintaxis similar para mostrar los atributos. En la figura 1.12 aparece la manera en que se incluyen los atributos y su tipo en el caso de estudio del empleado. Dependiendo de la herramienta que se utilice para definir el diagrama de clases, es posible que la sintaxis varíe levemente.

Fig. 1.12 Ejemplo de la declaración en UML de los atributos de la clase Empleado

Lo único que nos falta incluir en el código Java es la declaración de las asociaciones. Para esto, vamos a utilizar una sintaxis similar a la presentada anteriormente utilizando el nombre de la asociación como nombre del atributo y el nombre de la clase como su tipo, tal como se presenta en el siguiente fragmento de código:

```
package uniandes.cupi2.empleado;

public class Empleado
{
    //-----------------------------
    // Atributos
    //-----------------------------
    private String nombre;
    private String apellido;
    private double salario;
    private int genero;

    private Fecha fechaNacimiento;
    private Fecha fechaIngreso;

}
```

- Las asociaciones hacia la clase Fecha las declaramos como hicimos con el resto de atributos, usando el nombre de la asociación como nombre del atributo.
- El tipo de la asociación es el nombre de la clase hacia la cual está dirigida la flecha en el diagrama de clases.
- El orden de declaración de los atributos no es importante.

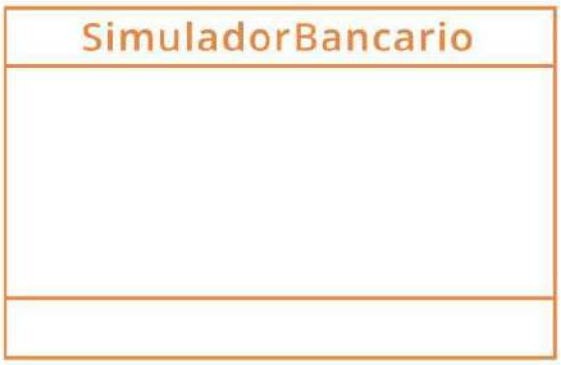
En la figura 1.13 aparece el diagrama de clases completo del caso de estudio del empleado.

Fig. 1.13 Representación de la clase Empleado en UML

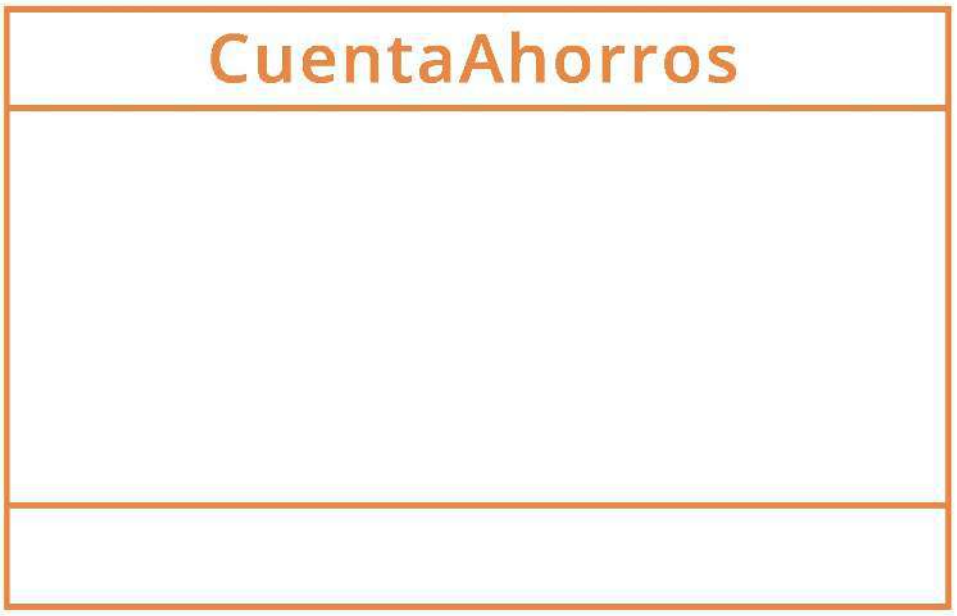
Tarea 8

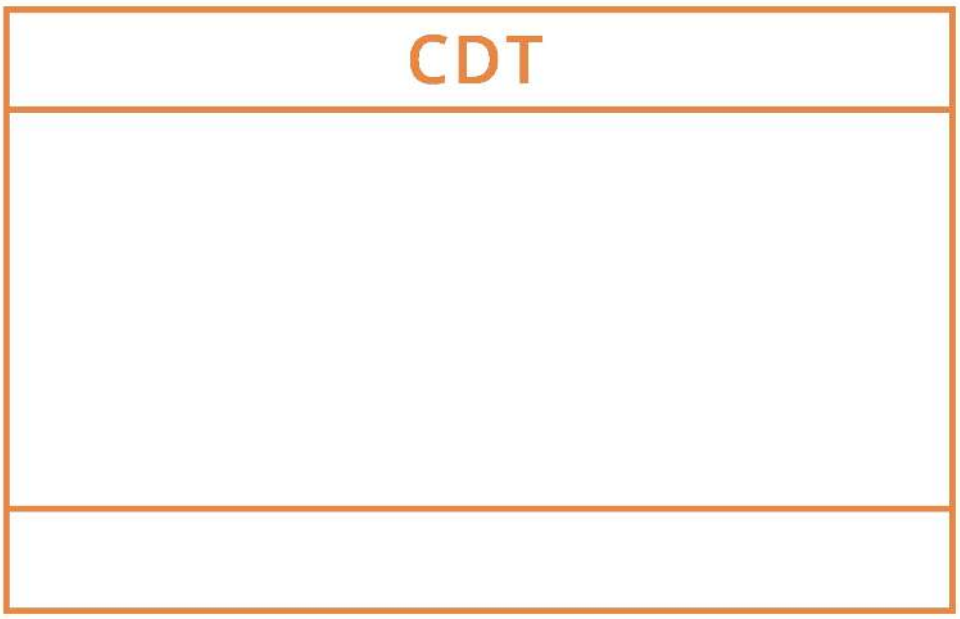
Objetivo: Crear habilidad en la definición de los tipos de datos para representar las características de una clase.

Escriba en Java y en UML las declaraciones de los atributos (y las asociaciones) para las cinco clases del caso de estudio del simulador bancario.

Declaracíon en Java	Descripción de la clase en UML
	

Declaracíon en Java	Descripción de la clase en UML
	

Declaracíon en Java	Descripción de la clase en UML
	

Declaracíon en Java	Descripción de la clase en UML
	

6.5. Métodos

Después de haber definido los atributos de las clases en Java, sigue el turno para lo que hemos llamado hasta ahora "los algoritmos" de la clase. Cada uno de esos algoritmos se denomina un **método**, y pretende resolver un problema puntual, dentro del contexto del

problema global que se quiere resolver. También se puede ver un método como un servicio que la clase debe prestar a las demás clases del modelo (o a ella misma si es el caso), para que ellas puedan resolver sus respectivos problemas.

Un método está compuesto por cuatro elementos:

- Un **nombre** (por ejemplo, cambiarSalario, para el caso de estudio del empleado, que serviría para modificar el salario del empleado).
- Una **lista de parámetros**, que corresponde al conjunto de valores (cada uno con su tipo) necesarios para poder resolver el problema puntual (Si el problema es cambiar el salario del empleado, por ejemplo, es necesario que alguien externo al empleado dé el nuevo salario. Sin esa información es imposible escribir el método). Para definir los parámetros que debe tener un método, debemos preguntarnos ¿qué información, que no tenga ya el objeto, es indispensable para poder resolver el problema puntual?
- Un **tipo de respuesta**, que indica el tipo de datos al que pertenece el resultado que va a retornar el método. Si no hay una respuesta, se indica el tipo `void`.
- El **cuerpo del método**, que corresponde a la lista de instrucciones que representa el algoritmo que resuelve el problema puntual.

Típicamente, una clase tiene entre cinco y veinte métodos (aunque hay casos en los que tiene decenas de ellos), cada uno capaz de resolver un problema puntual de la clase a la cual pertenece. Dicho problema siempre está relacionado con la información que contiene la clase. Piense en una clase como la responsable de manejar la información que sus objetos tienen en sus atributos, y los métodos como el medio para hacerlo. En el cuerpo de un método se explica entonces la forma de utilizar los valores de los atributos para calcular alguna información o la forma de modificarlos si es el caso.

> El encabezado del método (un método sin el cuerpo) se denomina su **signatura**.

Ejemplo 9

Objetivo: Mostrar la sintaxis que se usa en Java para declarar un método.

Usamos para esto el caso de estudio del empleado, con tres métodos sin cuerpo, suponiendo que cada uno debe resolver el problema que ahí mismo se describe. La declaración que aquí se muestra hace parte de la declaración de la clase (los métodos van después de la declaración de los atributos).
Se deja un cuarto método al final como tarea para el lector; en este caso, a partir de la descripción, debe determinar los parámetros, el retorno y la signatura del método.

```
public void cambiarSalario( double pNuevoSalario)
{
    // Aquí va el cuerpo del método
}
```

Nombre: cambiarSalario

Parámetros: pNuevoSalario de tipo real. Si no se entrega este valor como parámetro es imposible cambiar el salario del empleado. Note que al definir un parámetro se debe dar un nombre al valor que se espera y un tipo.

Retorno: ninguno (`void`) puesto que el objetivo del método no es calcular ningún valor, sino modificar el valor de un atributo del empleado.

Descripción: cambia el salario del empleado, asignándole el valor que se entrega como parámetro.

```
public double darSalario( )
{
    // Aquí va el cuerpo del método
}
```

Nombre: darSalario

Parámetros: ninguno, puesto que con la información que ya tienen los objetos de la clase Empleado es posible resolver el problema.

Retorno: el salario actual del empleado, de tipo real. En la signatura sólo se dice el tipo de datos que se va a retornar, pero no se dice cómo se retornará.

Descripción: retorna el salario actual del empleado.

```
public double calcularPrestaciones( )
{
    // Aquí va el cuerpo del método
}
```

Nombre: calcularPrestaciones

Parámetros: ninguno. Al igual que en el método anterior, no se necesita información externa al empleado para poder calcular sus prestaciones.

Retorno: las prestaciones anuales a las que tiene derecho el empleado. Las prestaciones, al igual que el salario, son un número real.

Descripción: retorna el valor de las prestaciones anuales a las que tiene derecho el empleado.

Nombre: aumentarSalario

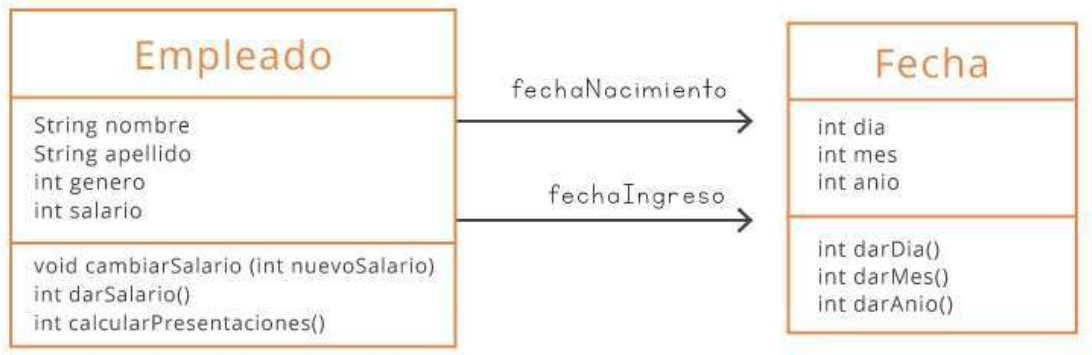
Parámetros:

Retorno:

Descripción: aumenta el salario del empleado en un porcentaje que corresponde a la inflación anual del país.

¿Cuáles son los métodos que se deben tener en una clase? Esa es una pregunta que se contestará en niveles posteriores. Por ahora, supongamos que la clase tiene ya definidos los métodos que necesita para poder resolver la parte del problema que le corresponde y trabajemos en el cuerpo de ellos. En el diagrama de clases de UML, se utiliza la tercera zona de la caja de una clase para poner las signaturas de los métodos, tal como se ilustra en la figura 1.14.

Fig. 1.14 Sintaxis en UML para mostrar las signaturas de los métodos de una clase



Tarea 9

Objetivo: Escribir y entender en Java la signatura de algunos métodos del caso de estudio del simulador bancario.

Complete la siguiente información, ya sea escribiendo la signatura del método que se describe, o interpretando la signatura que se da. Todos los métodos de esta tarea son de la clase CuentaAhorros.

```
public void consignarValor( double pValor )
{

}
```

Nombre:	
Parámetros:	
Retorno:	
Descripción:	

Nombre:	**darSaldo**
Parámetros:	ninguno.
Retorno:	valor de tipo real.
Descripción:	retorna el saldo de la cuenta de ahorros.

Signatura del Método:

Nombre:	retirarValor
Parámetros:	valor de tipo entero, que indica el monto que se quiere retirar de la cuenta de ahorros.
Retorno:	ninguno.
Descripción:	retira de la cuenta de ahorros el valor que se entrega como parámetro.

Signatura del Método:

Nombre:	darInteresMensual
Parámetros:	ninguno.
Retorno:	valor de tipo real.
Descripción:	retorna el interés mensual que paga una cuenta de ahorros.

Signatura del Método:

Nombre:	actualizarSaldoPorPasoMes
Parámetros:	ninguno.
Retorno:	ninguno.
Descripción:	actualiza el saldo de la cuenta de ahorros simulando que acaba de transcurrir un mes y que se deben agregar los correspondientes intereses ganados.

Signatura del Método:

6.6. La Instrucción de Retorno

En el cuerpo de un método van las instrucciones que resuelven un problema puntual o prestan un servicio a otras clases. El computador obedece las instrucciones, una después de otra, hasta llegar al final del cuerpo del método. Hay instrucciones de diversos tipos, la más sencilla de las cuales es la instrucción de retorno (`return`). Con esta instrucción le decimos al método cuál es el resultado que debe dar como solución al problema. Por ejemplo, si el problema es dar el salario del empleado, la única instrucción que forma parte del cuerpo de dicho método indica que el valor se encuentra en el atributo "salario". En el siguiente fragmento de programa se ilustra el uso de la instrucción de retorno.

```
public class Empleado
{
    //--------------------------------
    // Atributos
    //--------------------------------
    private String nombre;
    private String apellido;
    private double salario;
    private int genero;
    private Fecha fechaNacimiento;
    private Fecha fechaIngreso;

    //--------------------------------
    // Métodos
    //--------------------------------
    public double darSalario( )
    {
        return salario;
    }
}
```

- Tal como se había presentado antes, la declaración de la clase comienza con la declaración de cada uno de sus atributos (incluidas las asociaciones). Note que no hay diferencia sintáctica entre declarar algo de tipo entero (`genero`) y una asociación hacia la clase Fecha (`fechaIngreso`).
- Después de los atributos, viene la declaración de cada uno de los métodos de la clase. Cada método tiene una signatura y un cuerpo.
- Los métodos que van a ser utilizados por otras clases se deben declarar como públicos.
- En el cuerpo del método se deben incluir las instrucciones para resolver el problema puntual que se le plantea. El cuerpo de un método puede tener cualquier número de instrucciones.
- En el cuerpo de un método únicamente se puede hacer referencia a los atributos del objeto para el cual se está resolviendo el problema y a los parámetros, que representan la información externa al objeto que se necesita para resolver el problema puntual.
- En el caso del método cuyo problema puntual consiste en calcular el salario del empleado, la solución consiste en retornar el valor que se encuentra en el respectivo atributo. Fácil, ¿no?
- Es buena idea utilizar comentarios para separar la "zona" de declaración de atributos y la "zona" de declaración de métodos. Esta separación en zonas va a facilitar su posterior localización.

Todo método que declare en su signatura que va a devolver un resultado (todos los métodos que no son de tipo `void`) debe tener en su cuerpo una instrucción de retorno.

Cuando alguien llama un método sobre un objeto, éste "busca" dicho método en la clase a la cual pertenece y ejecuta las instrucciones que allí aparecen, utilizando sus propios atributos. Por esa razón, en el cuerpo de los métodos se puede hacer referencia a los atributos del objeto sin riesgo de ambigüedad, puesto que siempre se trata de los atributos del objeto al cual se le invocó el método. En el ejemplo anterior, si alguien invoca el método darSalario( ) sobre un objeto de la clase Empleado, dicho objeto va a su clase para establecer lo que debe hacer y la clase le explica que debe retornar el valor de su propio atributo llamado salario.

6.7. La Instrucción de Asignación

Los métodos que no están hechos para calcular un valor, sino para modificar el estado del objeto, utilizan la instrucción de asignación (`=`) para definir el nuevo valor que debe tener el atributo. Si existiera, por ejemplo, un método para duplicar el salario de un empleado, el siguiente sería el cuerpo de dicho método:

```
public class Empleado
{
    ...

    public void duplicarSalario( )
    {
        salario = salario * 2;
    }
}
```

En la parte izquierda de la asignación va el atributo que va a ser modificado (más adelante se extenderá a otros elementos del lenguaje, pero por ahora puede suponer que sólo se hacen asignaciones sobre los atributos). En la parte derecha va una **expresión** que indica el nuevo valor que debe guardarse en el atributo. Pueden formar parte de una expresión los atributos (incluso el que va a ser modificado), los parámetros y los valores constantes (como el 2 en el ejemplo anterior). Los elementos que forman parte de una expresión se denominan **operandos**. Adicionalmente en la expresión están los **operadores**, que indican cómo calcular el valor de la expresión. Los operadores aritméticos son la suma (`+`), la resta (`-`), la multiplicación (`*`) y la división (`/`).

En el siguiente fragmento de código vemos algunos métodos de la clase Empleado, que dan una idea del uso de la asignación, el retorno de valores y las expresiones:

```
public class Empleado
{
    ...
    public void cambiarSalario( double pNuevoSalario )
    {
        salario  =  pNuevoSalario;
    }

    public double calcularSalarioAnual( )
    {
        return  salario  *  12;
    }
}
```

- El primer método cambia el salario del empleado, asignándole el valor recibido como parámetro. Recuerde que siempre se asigna a la variable que aparece en la parte izquierda el valor que aparece en la parte derecha.
- El segundo método calcula el total al año que recibe el empleado por concepto de salario.

6.8. La Instrucción de Llamada de un Método

En algunos casos, como parte de la solución del problema, es necesario llamar un método de un objeto con el cual existe una asociación. Suponga que un empleado necesita saber el año en el que él ingresó a la empresa. Esa información la tiene el objeto de la clase Fecha que está siendo referenciado por su atributo `fechaIngreso`. Puesto que la clase Empleado no tiene acceso directo a los atributos de la clase Fecha, debe llamar el método de dicha clase que presta ese servicio (o que sabe resolver ese problema puntual). La sintaxis para hacerlo y el proceso de llamada (o invocación) se ilustran a continuación:

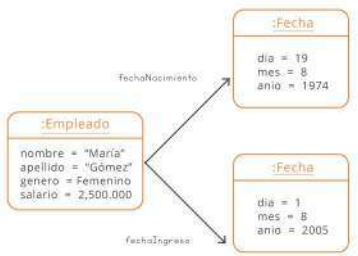


Diagrama de objetos para ilustrar la llamada del método: la empleada María Gómez ingresó a la empresa a trabajar en el año 2005.

```
public class Empleado
{
    ...

    public void miProblema( )
    {
        int valor = fechaIngreso.darAnio( );
        ...
    }
}
```

- Dentro de un método de la clase Empleado se necesita saber el año de ingreso a la empresa.
- Invocamos el método darAnio( ) sobre el objeto de la clase Fecha que representa la fecha de ingreso. Ese método debe retornar 2005 si el diagrama de objetos es el mostrado en la figura anterior.
- Para pedir un servicio a través de un método, debemos dar el nombre de la asociación, el nombre del método que queremos usar y un valor para cada uno de los parámetros que hay en su signatura (ninguno en este caso).
- El resultado de la llamada del método lo guardamos en una variable llamada valor, de tipo entero. Un poco más adelante se explica el uso de las variables.

```
public class Fecha
{
   ...

   public int darAnio( )
   {
       return anio;
   }
}
```

- El método darAnio( ) de la clase Fecha se contenta con retornar el valor que aparezca en el atributo " `anio` " del objeto sobre el cual se hace la invocación.

Con la referencia al objeto y el nombre del método, el computador localiza el objeto y llama el método pedido pasándole la información para los parámetros. Luego espera que se ejecuten todas las instrucciones del método y trae la respuesta en caso de que haya una.

De la misma manera que un objeto puede invocar un método de otro objeto con el cual tiene una asociación, también puede, dentro de uno de sus métodos, invocar otro método de su misma clase. ¿Para qué puede servir eso? Suponga que tiene un método cuyo problema se vería simplificado si utiliza la respuesta que calcula otro método. ¿Por qué no utilizarlo? Esta idea se ilustra en el siguiente fragmento de código:

```
public class Empleado
{
    ...

    public double calcularSalarioAnual( )
    {
        return  salario  *  12;
    }

    public double calcularImpuesto( )
    {
        double total = calcularSalarioAnual( );
        return total * 19.5 / 100;
    }
}
```

- Suponga que queremos calcular el monto de los impuestos que debe pagar el empleado en un año. Los impuestos se calculan como el 19,5% del total de salarios recibidos en un año.
- Si ya tenemos un método que calcula el valor total del salario anual, ¿por qué no lo utilizamos como parte de la solución? Eso nos va a permitir disminuir la complejidad del problema puntual del método, porque nos podemos concentrar en la parte que "nos falta" para resolverlo.
- Para invocar un método sobre el mismo objeto, basta con utilizar su nombre sin necesidad de explicar sobre cuál objeto queremos hacer la llamada. Por defecto se hace sobre él mismo.
- Note que utilizamos una variable (`total`) como parte del cuerpo del método. Una variable se utiliza para almacenar valores intermedios dentro del cuerpo de un método. Una variable debe tener un nombre y un tipo, y sólo puede utilizarse dentro del método dentro del cual fue declarada. En el siguiente capítulo volveremos a tratar el tema de las variables.

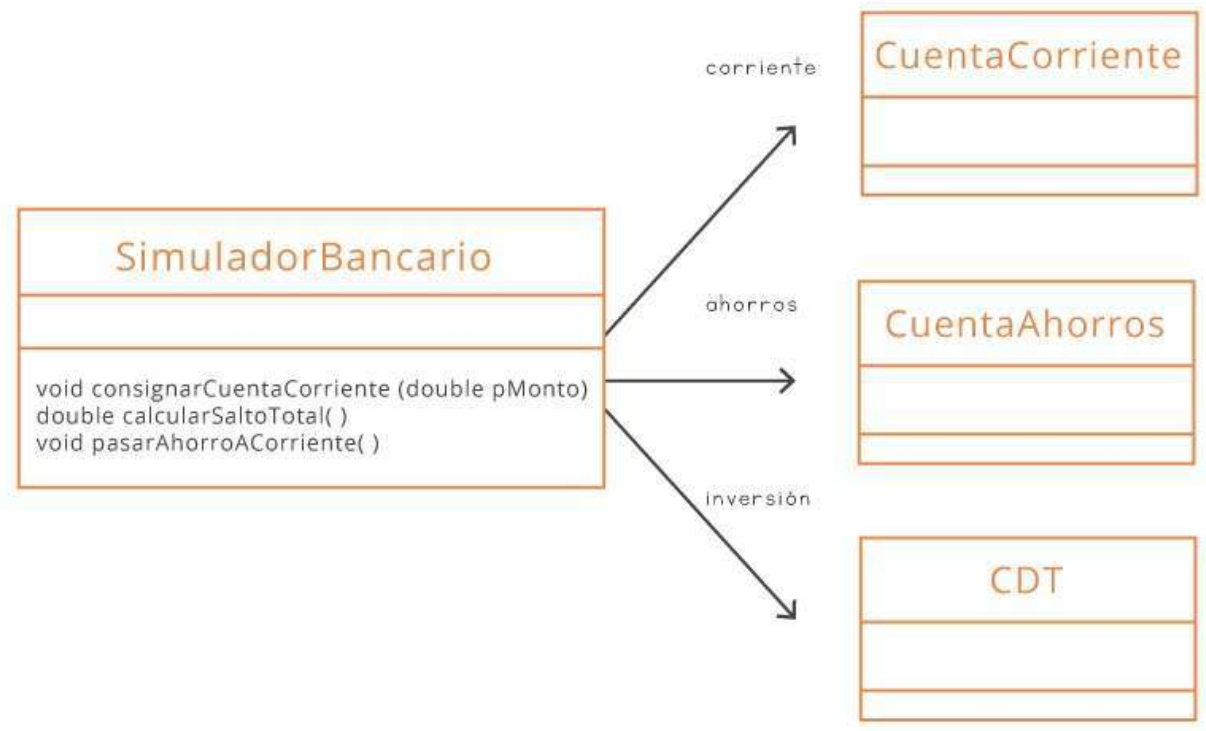
Ejemplo 10

Objetivo: Ilustrar la construcción de los métodos de una clase.

Para el caso de estudio del simulador bancario, en este ejemplo se muestra el código de algunos métodos, en donde se pueden apreciar los distintos tipos de instrucción que hemos visto hasta ahora.

```
package uniandes.cupi2.simuladorBancario.mundo;

public class SimuladorBancario
{
    //-----------------------------------
    // Atributos
    //-----------------------------------
    private String cedula;
    private String nombre;

    private CuentaCorriente corriente;
    private CuentaAhorros ahorros;
    private CDT inversion;
    private int mesActual;
    ...
}
```

- Declaración de los atributos de la clase que representa la cuenta bancaria. Note de nuevo la manera en que se declaran las relaciones con otras clases (como atributos, cuyo nombre corresponde al nombre de la asociación).



```
public void consignarCuentaCorriente( double pMonto )
{
    corriente.consignarMonto( pMonto );
}
```

- Para depositar en la cuenta corriente un valor que llega como parámetro, la cuenta bancaria pide dicho servicio al objeto que representa la cuenta corriente, usando la asociación que hay entre los dos y el método consignarMonto() de la clase CuentaCorriente.

```
public double calcularSaldoTotal( )
{
    return  corriente.darSaldo( ) +
            ahorros.darSaldo( ) +
            inversión.calcularValorPresente( pMesActual );
}
```

- Para calcular y retornar el saldo total de la cuenta bancaria, el método pide a cada uno de los productos que la componen que calcule su valor actual. Luego, suma dichos valores y los retorna como el resultado. Fíjese que una expresión puede estar separada en varias líneas, mientras no aparezca el símbolo ";" de final de una instrucción.
- Para calcular el valor presente del CDT se le debe pasar como parámetro el mes en el que va la simulación.

```
public void pasarAhorroACorriente( )
{
    double temp =  ahorros.calcularSaldo( );
    ahorros.retirar( temp );
    corriente.consignarValor( temp );
}
```

- Este método pasa todo el dinero depositado en la cuenta de ahorros a la cuenta corriente. Fíjese que es indispensable utilizar una variable (`temp`) para almacenar el valor temporal que se debe mover. ¿Se podría hacer sin esa variable? Las variables se declaran dentro del método que la va a utilizar y se pueden usar dentro de las expresiones que van en el cuerpo del método.

> Si hay necesidad de convertir un valor real en un valor entero, se puede usar el operador de conversión `(int)`. Dicho operador se utiliza de la siguiente manera:
> `int respuesta = ( int )( 1000 / 33 );`
> En ese caso, el computador primero evalúa la expresión y luego elimina las cifras decimales.

Tarea 10

Objetivo: Escribir el cuerpo de algunos métodos simples.

Escriba el cuerpo de los métodos de la clase CuentaBancaria (caso de estudio 2) cuya signatura aparece a continuación. Utilice los nombres de los atributos que aparecen en la declaración de la clase. Suponga que existen los métodos que necesite en las clases CuentaCorriente, CuentaAhorros y CDT.

```
public void ahorrar( double pMonto)
{



}
```

Pasa de la cuenta corriente a la cuenta de ahorros el valor que se entrega como parámetro (suponiendo que hay suficientes fondos).

```
public void retirarAhorro( double pMonto)
{


}
```

Retira un valor dado de la cuenta de ahorros (suponiendo que hay suficientes fondos).

```
public double darSaldoCorriente( )
{


}
```

Retorna el saldo que hay en la cuenta corriente. No olvide que éste es un método de la clase CuentaBancaria.

```
public void retirarTodo( )
{


}
```

Retira todo el dinero que hay en la cuenta corriente y en la cuenta de ahorros.

```
public void duplicarAhorro( )
{


}
```

Duplica la cantidad de dinero que hay en la cuenta de ahorros.

```
public void avanzarMesSimulacion( )
{


}
```

Avanza un mes la simulación de la cuenta bancaria.

> Dentro de un método:
>
> - Para hacer referencia a un atributo basta con utilizar su nombre (`salario`).
> - Para invocar un método sobre el mismo objeto, se debe dar únicamente el nombre del método y la lista de valores para los parámetros (cambiarSalario(2000000)).
> - Para invocar un método sobre un objeto con el cual se tiene una asociación, se debe dar el nombre de la asociación, seguido de un punto y luego la lista de valores para los parámetros (fechaIngreso.darDia( )).

6.9. Llamada de Métodos con Parámetros

Este tema se profundizará en los capítulos posteriores. Por ahora sólo queremos dar una idea global del proceso de llamada de un método con parámetros. Para eso vamos a contestar siete preguntas:

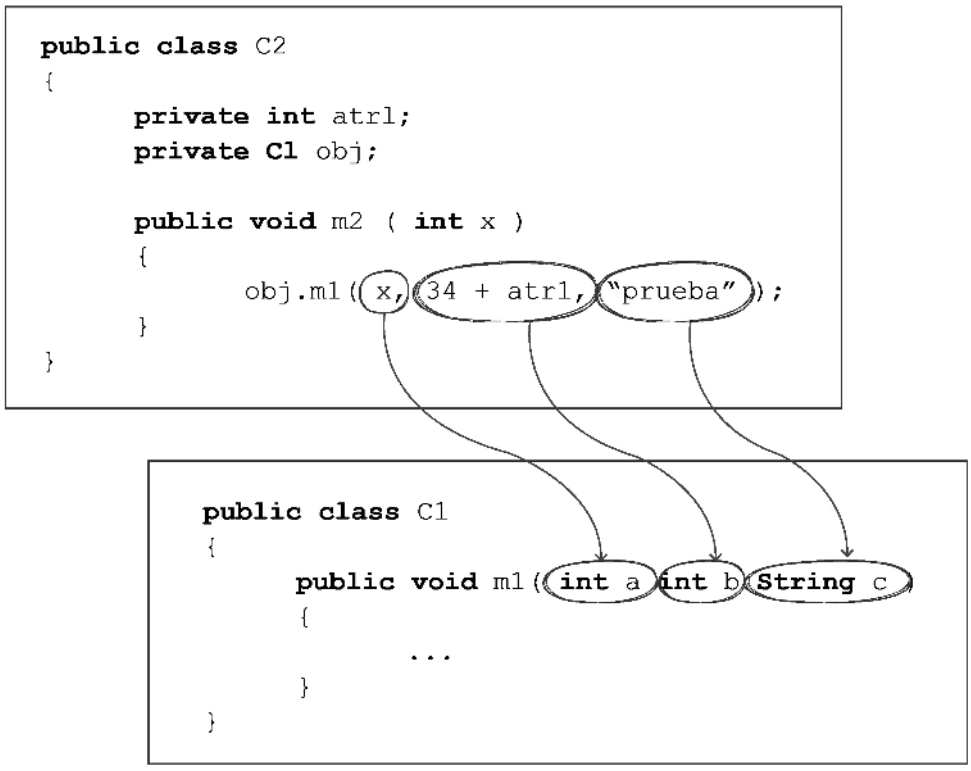
- ¿Cuándo necesita parámetros un método? Un método necesita parámetros cuando la información que tiene el objeto en sus atributos no es suficiente para resolver el problema que le plantean.
- ¿Cómo se declara un parámetro? En la signatura del método se define el tipo de dato del parámetro y se le asocia un nombre. Es conveniente que este nombre dé una idea clara del valor que se va a recibir por ese medio.
- ¿Cómo se utiliza el valor del parámetro? Basta con utilizar el nombre del parámetro en el cuerpo del método, de la misma manera en que se utilizan los atributos.
- ¿Se puede utilizar el parámetro por fuera del cuerpo del método? No. En ningún caso.
- Aquel que hace la llamada del método, ¿cómo hace para definir los valores de los parámetros? En el momento de hacer la llamada, se deben pasar tantos valores como

parámetros está esperando el método. Esos valores pueden ser constantes (por ejemplo, 500), atributos del objeto que hace la llamada (por ejemplo, `salario`), parámetros del método desde el cual se hace la llamada (por ejemplo, `pNuevoSalario`), o expresiones que mezclen los tres anteriores (por ejemplo, `salario + pNuevoSalario * 500`).

- ¿Cómo se hace la relación entre esos valores y los parámetros? Los valores se deben pasar teniendo en cuenta el orden en el que se declararon los parámetros. Eso se ilustra en la figura 1.15.

- ¿Qué sucede si se pasan más (o menos) valores que parámetros? El compilador informa que hay un error en la llamada. Lo mismo sucede si los tipos de datos de los valores no coinciden con los tipos de datos de los parámetros.

Fig. 1.15 Llamada de un método con parámetros



- Tenemos una clase C1, con un método m1() que tiene tres parámetros.
- Tenemos una clase C2, con un atributo de la clase C1. Desde allí vamos a llamar el método m1() de la primera clase.
- Debemos pasarle 3 valores en el momento de invocar el método. El primer valor es el parámetro x del método m2( ). El segundo valor es una expresión que incluye una constante y un atributo. El tercer valor es una constante de tipo cadena de caracteres.

- Al hacer la llamada se hace la correspondencia uno a uno entre los valores y los parámetros.
- Después de hacer la correspondencia se calcula cada valor y se le asigna al respectivo parámetro. Esta copia del valor se hace para todos los tipos simples de datos.
- Una vez que se han inicializado los parámetros se inicia la ejecución del método.

6.10. Creación de Objetos

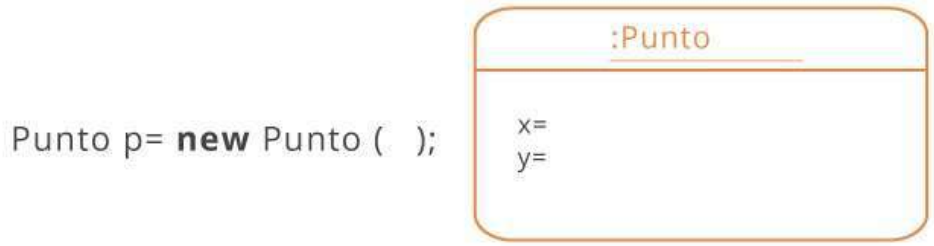
La creación de objetos es un tema que será abordado nuevamente en el segundo nivel. Sin embargo se explicará la creación de objetos porque es indispensable para entender la estructura de un programa completo. Para esto empezaremos contestando algunas preguntas.

- ¿Quién crea los objetos del modelo del mundo? Típicamente, el proceso lo inicia la interfaz de usuario, creando una instancia de la clase más importante del modelo. Lo que sigue, depende del diseño que se haya hecho del programa.

- ¿Cómo se guarda un objeto que acaba de ser creado? Más que guardar un objeto se debe hablar de referenciar. Una referencia a un objeto se puede guardar en cualquier atributo o variable del mismo tipo.

Un objeto se crea utilizando la instrucción `new` y dando el nombre de la clase de la cual va a ser una instancia. Para crear un empleado, por triángulo, se usa la expresión `new Triangulo( )`. Al ejecutar esta instrucción, el computador se encarga de buscar la declaración de la clase y asignar al objeto un nuevo espacio en memoria en donde pueda almacenar los valores de todos sus atributos. Como no es responsabilidad del computador darle un valor inicial a los atributos, éstos quedan en un valor que se puede considerar indefinido, tal como se sugiere en la figura 1.16, en donde se muestra la sintaxis de la instrucción `new` y el efecto de su uso.

Fig. 1.16 Creación de un objeto usando la instrucción new



- El resultado de ejecutar la instrucción del ejemplo es un nuevo objeto, con sus atributos no inicializados.
- Dicho objeto está "referenciado" por p, que puede ser un atributo o una variable de tipo Punto.

Para inicializar los valores de un objeto, las clases permiten la definición de métodos constructores, los cuales son invocados automáticamente en el momento de ejecutar la instrucción de creación. Un método constructor tiene dos reglas fundamentales:

1. Se debe llamar igual que la clase.
2. No puede tener ningún tipo de retorno, puesto que su único objetivo es dar un valor inicial a los atributos.

El siguiente es un ejemplo de un método constructor para la clase Punto:

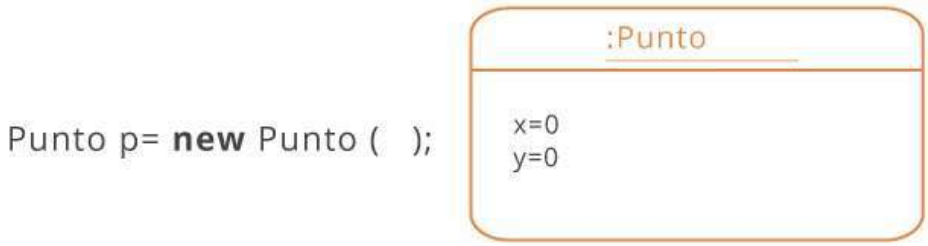
```
public Punto(  )
{
    x = 0;
    y = 0;
}
```

Un método constructor tiene el mismo nombre de la clase (así lo puede localizar el compilador) y no tiene ningún tipo de retorno.

- El método constructor del ejemplo le asigna valores iniciales por defecto a todos los atributos del objeto.
- Un método constructor no se puede llamar directamente, sino que es invocado automáticamente cada vez que se crea un nuevo objeto de la clase.

El método constructor anterior le asigna un valor por defecto a cada uno de los atributos del objeto, evitando así tener valores indefinidos. El hecho de incluir este método constructor en la declaración de la clase hace que éste siempre se invoque como parte de la respuesta del computador a la instrucción `new` . En la figura1.17 se ilustra la creación de un objeto de una clase que tiene un método constructor.

Fig. 1.17 Creación de un objeto cuya clase tiene un método constructor.



Puesto que en muchos casos los valores por defecto no tienen sentido (no todos los puntos pueden tener coordenadas 0,0), es posible agregar parámetros en el constructor, lo que obliga a todo aquel que quiera crear una nueva instancia de esa clase a definir dichos valores iniciales.

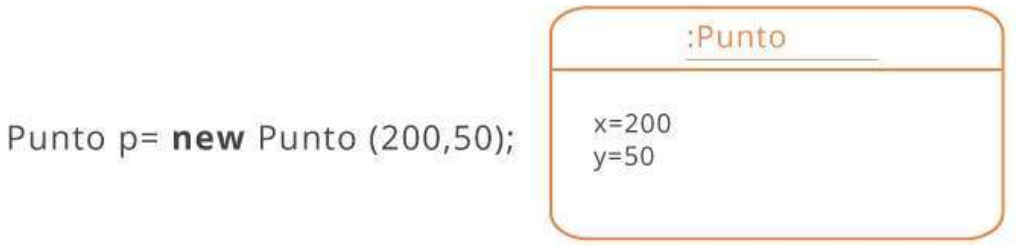
En el siguiente ejemplo, se muestra un constructor que recibe por parámetro las coordenadas que se desea asignar al punto desde su creación:

```
public Punto( double pX, double pY )
{
    x = pX;
    y = pY;
}
```

- Este constructor exige 2 parámetros, de tipo real, para poder inicializar los objetos de la clase Punto.
- En el constructor se asignan los valores de los parámetros a los atributos.

En la figura 1.18 se ilustra la creación de un objeto de una clase que usa el método constructor con parámetros definido arriba.

Fig. 1.18 Creación de un objeto a partir de un constructor con parámetros.



- El objeto creado se ubica en alguna parte de la memoria del computador. Dicho objeto es referenciado por el atributo o la variable llamada " `p` ".

Debido a que es necesario que el triángulo tenga 3 puntos, su método constructor debe incluir la creación de los 3 puntos, como se muestra a continuación:

```
public Triangulo(    )
{
    // Inicializa los puntos
    punto1 = new Punto( 200, 50 );
    punto2 = new Punto( 300, 200 );
    punto3 = new Punto( 100, 200 );
}
```

Tarea 11

Objetivo: Generar habilidad en el uso de los constructores de las clases.

Complete el constructor de la clase Color, de manera que reciba por parámetro los valores que se desea asignar a cada uno de sus atributos y los inicialice.

```
public Color(                                    )
{




}
```

Complete el constructor de la clase Triangulo para que inicialice el color de relleno y el color de las líneas, usando el constructor creado arriba. (Tenga en cuenta que los valores de cada componente del color se deben inicializar con un entero etre 0 y 255).

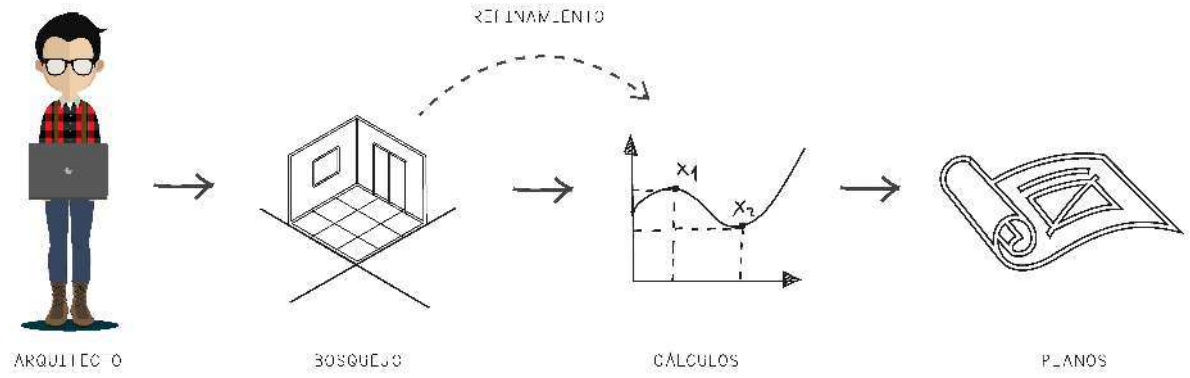
```
public Triangulo(    )
{
    // Inicializa los puntos
    punto1 = new Punto( 200, 50 );
    punto2 = new Punto( 300, 200 );
    punto3 = new Punto( 100, 200 );




}
```

7. Diseño de la Solución

En esta sección se da una visión global de la etapa de diseño, la segunda etapa del proceso de desarrollo de un programa.

Si hacemos el paralelo con el trabajo de un arquitecto que construye un edificio, podemos imaginar que éste, una vez que ha terminado de entender lo que el cliente quiere, empieza la etapa de diseño del edificio. La figura 1.19 pretende mostrar que la actividad de diseño se suele desarrollar a través de refinamientos sucesivos: el arquitecto primero hace un bosquejo de lo que quiere construir, luego hace los cálculos necesarios para verificar si esta solución es viable (debe por ejemplo estimar los materiales y el costo de mano de obra). Si llega a la conclusión de que no cumple por alguna razón las restricciones impuestas por el cliente (o se le ocurre una manera mejor de hacerlo), realiza los ajustes del caso y repite de nuevo la etapa de cálculos. La actividad termina cuando el arquitecto decide que encontró una buena solución al problema. En ese momento comienza a elaborar un conjunto de planos que van a ser utilizados como guía para la construcción del edificio.

Fig. 1.19 El diseño es una actividad iterativa hasta encontrar una solución

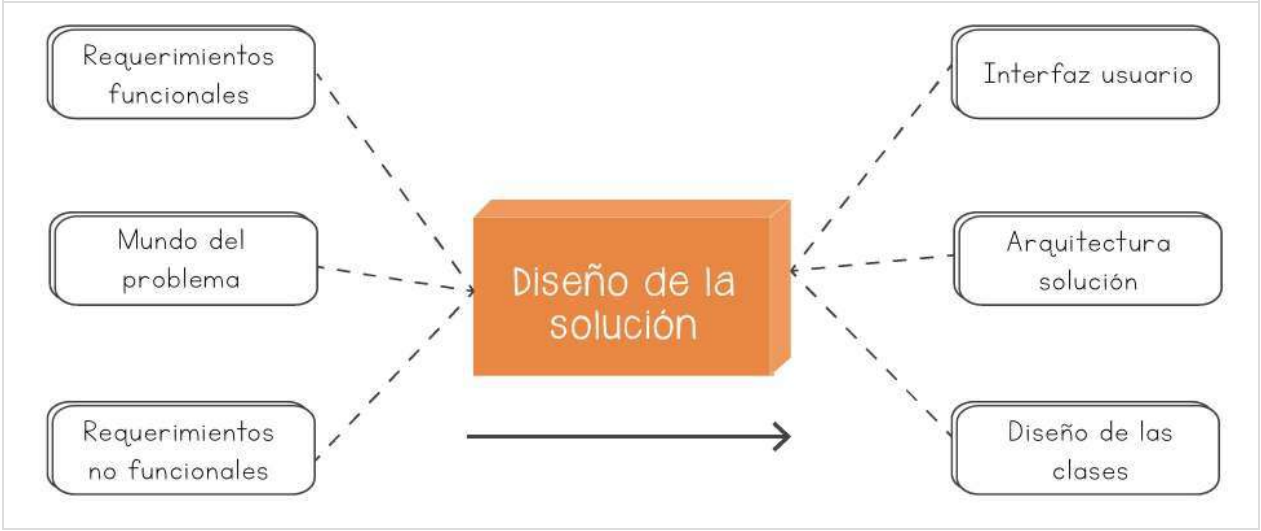


En el caso de la construcción de un programa, la actividad de diseño sigue el mismo esquema: nuestro bosquejo inicial es el modelo conceptual del mundo del problema, nuestros cálculos consisten en verificar los requerimientos no funcionales y calcular el costo de implementación, y nuestros planos son, entre otros, diagramas detallados escritos en UML. En cada refinamiento introducimos o ajustamos algunos de los elementos del programa y así nos vamos aproximando a una solución adecuada.

Como se muestra en la figura 1.20, los documentos de diseño (nuestros "planos") deben hacer referencia al menos a tres aspectos:

1. El diseño de la interfaz de usuario.
2. La arquitectura de la solución.
3. El diseño de las clases.

Fig. 1.20 Entradas y salidas de la etapa de diseño



- Como entrada tenemos el análisis del problema, dividido en tres partes: requerimientos funcionales, mundo del problema y requerimientos no funcionales.
- La salida es el diseño del programa, que incluye la interfaz de usuario, la arquitectura y el diseño de las clases.

7.1. La Interfaz de Usuario

La interfaz de usuario es la parte de la solución que permite que los usuarios interactúen con el programa. A través de la interfaz, el usuario puede utilizar las operaciones del programa que implementan los requerimientos funcionales. La manera de construir esta interfaz será el tema del nivel 5 de este libro. Hasta entonces, todas las interfaces que se necesitan para completar los programas de los casos de estudio serán dadas.

7.2. La Arquitectura de la Solución

En general, cuando se quiere resolver un problema, es bueno contar con mecanismos que ayuden a dividirlo en problemas más pequeños. Estos problemas son menos complejos que el problema original y, por lo tanto, más fáciles de resolver.

Por ejemplo, si se quiere construir un aeropuerto, al plantear la solución, los diseñadores identifican sus grandes partes: las pistas de aterrizaje, las salas de llegada y salida de pasajeros, la torre de control, etc. Luego tratan de diseñar esas partes por separado, sabiendo que cada diseño es más sencillo que el diseño completo del aeropuerto. Lo importante es después poder pegar los pedazos de solución. Para eso es importante tener un diseño de alto nivel en el que aparezcan a grandes rasgos los elementos que conforman la solución. Eso es lo que en programación se denomina la **arquitectura de la solución**. En

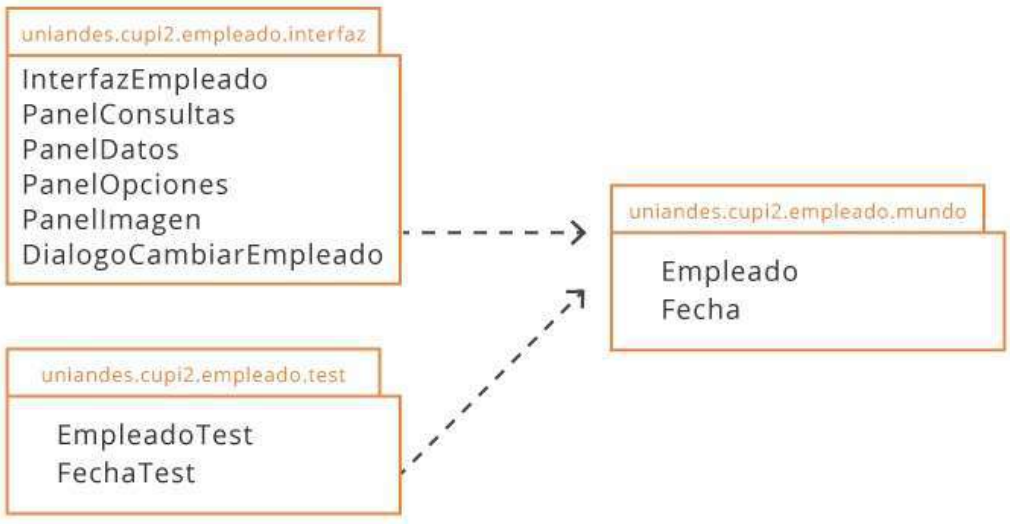
el caso de los problemas que tratamos en este libro, dado que son pequeños y su complejidad es baja, nos vamos a contentar con identificar los paquetes y las clases que van en cada uno de ellos. Luego, nos dedicaremos a trabajar en las clases de cada paquete, para finalmente armar la solución completa.

En los problemas en los que vamos a trabajar a lo largo del libro, se pueden identificar 3 grandes grupos de clases:

1. Las clases que implementan la interfaz de usuario.
2. Las clases que implementan el modelo del mundo.
3. Las clases que implementan las pruebas.

Cada uno de estos grupos va a ir en un paquete distinto. Esta manera de separar la aplicación en estos tres paquetes la vamos a llamar la **arquitectura básica** y la estaremos utilizando en la gran mayoría de los casos de estudio de este libro. La figura 1.21 ilustra la arquitectura de la solución para el caso de estudio del empleado, en la cual se puede apreciar que hay tres paquetes, que cada uno tiene en su interior un grupo de clases, y que estos paquetes están relacionados (la relación está indicada por las flechas punteadas).

Fig. 1.21 Arquitectura de paquetes del caso de estudio del empleado



- En el diagrama de paquetes se puede leer que alguna clase del paquete

`uniandes.cupi2.empleado.interfaz` utiliza algún servicio de una clase del paquete `uniandes.cupi2.empleado.mundo`. En este diagrama no se entra en detalles sobre cuál clase es la que tiene la relación.

- El diagrama de paquetes es muy útil para darse una idea de la estructura del programa. En este nivel sólo estamos interesados en mirar por dentro el paquete con las clases del mundo. En niveles posteriores nos interesaremos por las demás clases.

Sin entrar por ahora en mayores detalles, podemos decir que en el paquete de la interfaz estarán las clases que implementan los elementos gráficos y de interacción, lo mismo que las clases que implementan los requerimientos funcionales y las clases que crean las instancias del modelo del mundo. Es allí donde están agrupadas todas esas responsabilidades. Este es el tema del nivel 5 de este libro. Por ahora, paciencia...

7.3. El Diseño de las Clases

El objetivo de esta parte de la etapa de diseño es mostrar los detalles de cada una de las clases que van a hacer parte del programa. Para esto vamos a utilizar el diagrama de clases de UML, con toda la información que presentamos en las secciones anteriores (clases, atributos y signaturas de los métodos). En el nivel 4, veremos la manera de precisar las responsabilidades y compromisos de cada uno de los métodos (exactamente qué debe hacer cada método), de manera que la persona que vaya a implementar los métodos no deba guiarse únicamente por los nombres de los mismos.

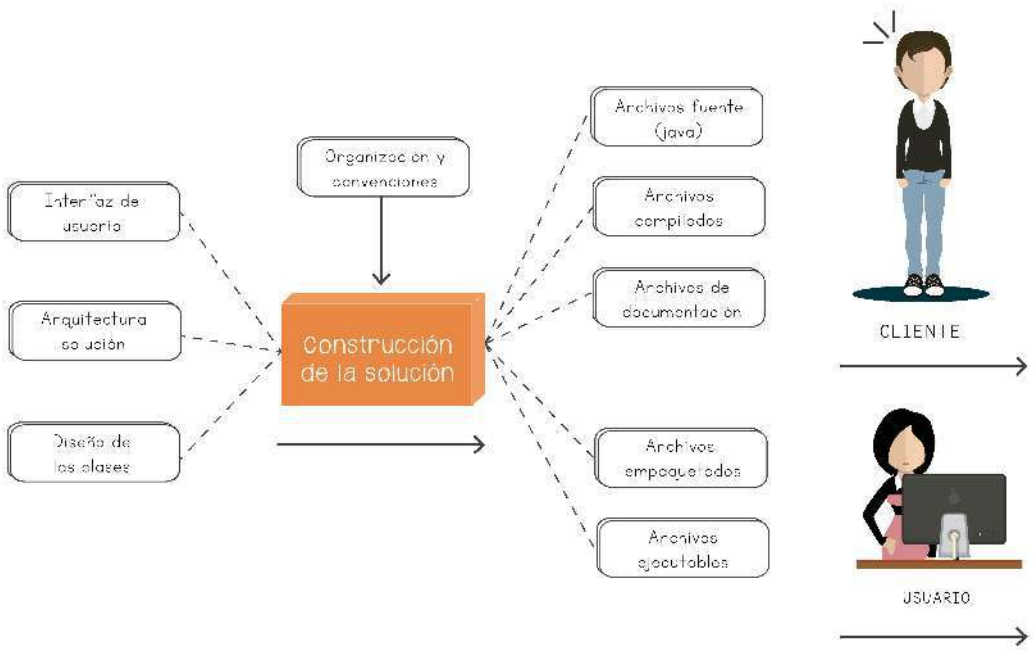
8. Construcción de la Solución

8.1. Visión Global

En la etapa de construcción de la solución debemos escribir todos los elementos que forman parte del programa que fue diseñado en la etapa anterior, y que resuelve el problema planteado por el cliente. Dicho programa será instalado en el computador del usuario y luego ejecutado.

En la figura 1.22 aparecen las entradas y las salidas de esta etapa. Allí se puede apreciar que un programa consta de un conjunto estructurado de archivos de distintos tipos (no sólo están los archivos de las clases Java). La descripción de todos ellos se hará en la **sección 8.2**. También se puede ver que la etapa de construcción debe seguir ciertas reglas de organización, las cuales varían de empresa a empresa de desarrollo de software, y que deben hacerse explícitas antes de comenzar el trabajo. Estas reglas de organización son el tema de la **sección 8.3**. Al terminar la etapa de construcción, algunos archivos empaquetados y algunos archivos ejecutables irán al computador del usuario, pues en ellos queda el programa listo para su uso. El resto de los archivos se entregan al cliente, quien los podrá utilizar en el futuro para darle mantenimiento al programa, permitiendo así incluir nuevas opciones y dando al cliente la oportunidad de adaptar el programa a los cambios que puedan aparecer en el mundo del problema.

Fig. 1.22 Entradas y salidas de la etapa de construcción de la solución



8.2. Tipos de Archivos

Dentro de cada uno de los proyectos de desarrollo en Java incluidos en este libro, aparecen nueve tipos distintos de archivos, los cuales contienen partes de la solución. A continuación se describe cada uno de ellos:

Tipo de archivo	¿Qué contiene?	¿Cómo se usa?	¿Cómo se construye?
.class	Es un archivo que contiene el código compilado de una clase Java. El compilador genera este archivo, que después podrá ser ejecutado. En el proyecto habrá un archivo .class por cada archivo .java.	Lo usa el computador para ejecutar un programa.	Se construye llamando el compilador del lenguaje, e indicándole el archivo .java que debe compilar.
.docx	Es un archivo que tiene parte de la especificación del problema (el enunciado general y los requerimientos funcionales). Tiene el formato usado por Microsoft Word®.	Se requiere tener instalado en el computador la aplicación Microsoft Word®. Para abrirlo basta con hacer doble clic en el archivo desde el explorador de archivos.	Se crea y modifica desde la aplicación Microsoft Word®.
.html	Es un archivo con la documentación de una clase, generada automáticamente por la utilidad Javadoc.	Se requiere tener instalado en el computador un navegador de Internet. Para abrirlo basta con hacer doble clic en el archivo desde el explorador de archivos.	Lo crea automáticamente la aplicación Javadoc, que extrae y organiza la documentación de una clase escrita en Java.
.jar	Es un archivo en el que están empaquetados todos los archivos .class de un programa. Su objetivo es facilitar la instalación de un programa en el computador de un usuario. En lugar de tener que copiar cientos de archivos .class se empaquetan todos ellos en un solo .jar.	Lo usa el computador para ejecutar un programa.	Se construye utilizando la utilidad jar que viene con el compilador de Java.

.java	Es un archivo con la implementación de una clase en Java.	Se le pasa al compilador para que cree a partir de él un .class, que será posteriormente ejecutado por el computador.	Desde cualquier editor de texto. En nuestro caso, el ambiente de desarrollo Eclipse va a permitir editar este tipo de archivo, dándonos ayudas para detectar errores de sintaxis.
.eap	Es un archivo con los diagramas de clases y de arquitectura del programa. Están escritos en el formato de Enterprise Architect®.	Se requiere tener instalado en el computador la aplicación Enterprise Architect®. Para abrirlo basta con hacer doble clic en el explorador de archivos.	Se crea, modifica e imprime desde la aplicación Enterprise Architect®.
.jpeg/png	Son archivos que contienenuna imagen. Los usamos para mostrar los distintos diagramas del programa. Esto permite visualizar el diseño a aquellos que no cuenten con el programa Enterprise Architect®.	Cualquier programa de imágenes (incluso los navegadores de Internet) pueden leer estos archivos.	Se crean con cualquier editor de imágenes.
.zip	Es un archivo que empaqueta un conjunto de archivos. Tiene la ventaja de que los almacena de manera comprimida y hace que ocupen menos espacio.	Muchas herramientas en el mercado permiten manejar este tipo de archivos. Si tiene alguna de ellas instalada en su computador, un doble clic desde el explorador de archivos iniciará la aplicación.	Se construyen utilizando las mismas herramientas que permiten extraer de allí los archivos que contienen.

8.3. Organización de los Elementos de Trabajo

Sigamos con el paralelo que estábamos haciendo con el edificio. Una vez terminados los planos debemos pasar a la etapa de construcción. Antes de empezar a abrir el hueco para los cimientos y de comprar los materiales que se necesitan, es necesario fijar todas las normas de organización. Lo primero es decidir dónde se va a poner cada elemento para la construcción: dónde van los ladrillos, dónde va el cemento, etc. Luego, cómo vamos a llamar las cosas. Si hay varios tipos de puertas, por ejemplo, nos debemos poner de acuerdo en la manera de etiquetarlas. Esto último es lo que se denomina una convención. Tanto la organización como las convenciones no son universales y en cada edificio que se va a construir pueden cambiar. Lo importante es que antes de iniciar la construcción todo el mundo esté informado y se comprometa a respetar dichas normas.

Para la construcción de un programa se sigue la misma idea: se define una organización (siempre debemos saber dónde buscar un elemento de la solución) y un conjunto de convenciones (por ejemplo, el archivo en el que están los requerimientos funcionales siempre se va a llamar de la misma manera). Nuestros elementos están siempre en archivos, y nuestra estructura de organización de basa en el sistema de directorios.

> En esta sección presentamos la organización y las convenciones que utilizamos en los proyectos de construcción de los programas de los casos de estudio. Todos los proyectos de este libro las siguen y, aunque no son universales, reflejan las prácticas comunes de los equipos de desarrollo de software.

8.3.1 Proyectos y Directorios

Un proyecto de desarrollo va siempre en un directorio, cuyo nombre indica su contenido. En nuestro caso el nombre del directorio comienza por el nivel, seguido del nombre del caso de estudio (por ejemplo, n1_empleado).

Dentro del directorio principal, se encuentran siete directorios, con el contenido que se muestra en la figura 1.23.

Fig. 1.23 Estructura de directorios dentro de un proyecto de desarrollo

Comencemos entonces a recorrer cada uno de estos directorios, utilizando para esto el proyecto de desarrollo del caso de estudio del empleado. En la tarea 11 se dan los pasos para poder comenzar este recorrido.

Tarea 12

Objetivo: Preparar la organización para iniciar el recorrido por los elementos de un proyecto de desarrollo, utilizando como ejemplo el caso de estudio del empleado.

Siga los pasos que se enuncian a continuación:

1. Descargue de aquí al disco de su computador el proyecto de nivel 1 llamado n1_empleado. Descomprímalo (está en formato zip) y recorra los directorios internos utilizando el explorador de archivos.
2. Verifique que en su computador se encuentre instalado el compilador de Java. Si no está instalado, vaya al **anexo A** del libro y siga las instrucciones para instalarlo. Algunos programas del libro están escritos para versiones de Java posteriores a la versión 1.4.
3. Verifique que en su computador se encuentre instalado el ambiente de desarrollo

Eclipse. Si no está instalado, vaya al **anexo B** del libro y siga las instrucciones para instalarlo.

8.3.2. El Directorio source

En este directorio encontrará los archivos fuente, en los que está la implementación en Java de cada una de las clases. Cada clase está en un archivo distinto, dentro de un directorio que refleja la jerarquía de paquetes. Esta relación entre paquetes y directorios es la que permite al compilador encontrar las clases en el espacio de trabajo. En la figura 1.24 se ilustra esta relación.

Fig. 1.24 Relación entre los paquetes y la jerarquía de directorios

Tarea 13

Objetivo: Recorrer los archivos fuente de un programa y ver la relación entre la jerarquía de directorios y la estructura de paquetes.

Siga los pasos que se dan a continuación.

1. Abra el explorador de archivos y sitúese en el directorio source del proyecto que instaló en la tarea 11.

2. Entre al directorio "uniandes". Dentro de éste entre al directorio "cupi2" y luego al directorio "empleado". Allí deben aparecer los directorios "interfaz" y "mundo". Entre en cualquiera de ellos y utilice el bloc de notas para ver el contenido de un archivo .java. Es importante decir que si se mueve un archivo a otro directorio, o se cambia el paquete al que pertenece sin desplazar físicamente el archivo al nuevo directorio, el programa no se va a compilar correctamente.

8.3.3. El Directorio classes

En este directorio están todos los archivos .class. Tiene la misma jerarquía de directorios que se usa para los archivos fuente. No es muy interesante su contenido, porque para poder ver estos archivos por dentro se necesitan editores especiales. Si intenta abrir uno de estos archivos con editores de texto normales, va a obtener unos caracteres que aparentemente no tienen ningún sentido.

Estos archivos tienen por dentro el bytecode (código binario) producto de compilar la correspondiente clase Java.

8.3.4. El Directorio test

En este directorio están todos los archivos que hacen las pruebas automáticas del programa. Por ahora lo único importante es saber que en su interior hay varios directorios, con archivos .class, .jar y .java. En un nivel posterior entraremos a mirar este directorio.

8.3.5. El Directorio docs

En este directorio hay dos subdirectorios:

- **specs**: contiene todos los documentos de diseño. Allí encontrará: (1) el archivo Descripción.docx, con el enunciado del caso de estudio, (2) el archivo RequerimientosFuncionales.docx con la especificación de los requerimientos funcionales, (3) el archivo Modelo.eap con los diagramas de clases del diseño y (4) un conjunto de archivos .jpg con las imágenes de los distintos diagramas de clases.
- **api**: contiene los archivos de la documentación de las clases del programa. Estos archivos sólo se verán a partir del nivel 4.

8.3.6. El Directorio lib

En este directorio encontrará el archivo empaquetado para instalar en el computador del usuario. En el caso de estudio del empleado dicho archivo se llama empleado.jar. Este archivo tiene la misma estructura interna de un archivo .zip, así que si desea ver su

contenido puede utilizar cualquiera de los programas que permiten manejar esos archivos. En su interior deberá encontrar todos los archivos .class del proyecto.

8.3.7 El Directorio data

Este directorio contiene archivos con información que utiliza el programa, ya sea para almacenar datos (si tuviéramos una base de datos estaría en ese directorio) o para leerlos (por ejemplo, en el caso de estudio del empleado, allí se guarda la foto en un archivo con formato jpeg).

8.4. Eclipse: Un Ambiente de Desarrollo

Un ambiente (o entorno) de desarrollo es una aplicación que facilita la construcción de programas. Principalmente, debe ayudarnos a escribir el código, a compilarlo y a ejecutarlo. Eclipse es un ambiente de múltiples usos, uno de los cuales es ayudar al desarrollo de programas escritos en Java. Es una herramienta de uso gratuito, muy flexible y adaptable a las necesidades y gustos de los programadores.

Tarea 14

Objetivo: Estudiar tres funcionalidades básicas del ambiente de desarrollo:

1. Cómo abrir un proyecto que ya existe (como el del caso de estudio).
2. Cómo leer y modificar los archivos de las clases Java.
3. Cómo ejecutar el programa.

Siga los pasos que se enuncian a continuación.

1. ¿Cómo abrir en Eclipse el programa n1_empleado? Puede hacerlo de dos formas:

Opción 1: Creando el proyecto directamente en la estructura de directorios

- Descomprima el archivo .zip que contiene el proyecto (por ejemplo en C:/temp/).
- Cree un proyecto Java en Eclipse (menú *File/New/Java Project*), con la ruta del directorio (C:/temp/n1_empleado) y el nombre del proyecto (n1_empleado).
- Puede aceptar la creación ahora (botón "*Finish*"), o navegar a la siguiente ventana ("*Next*") para ver las propiedades del proyecto.

Opción 2: Importando el proyecto de la estructura de directorios:

- Descomprima el archivo .zip que contiene el proyecto (por ejemplo en C:/temp/)
- Elija la opción de importación (menú *File/Import*…). En el diálogo en el que le preguntan la fuente de la importación seleccione "Existing Project into Workspace".

- Seleccione la carpeta del proyecto (C:/temp/n1_empleado) y finalice.

2. ¿Cómo explorar en Eclipse el contenido de un proyecto abierto?

- Utilice la vista llamada navegador. Si la vista no está disponible, búsquela en el menú *Window/Show View/ Navigator*.
- Revise la estructura de directorios del proyecto n1_empleado y recuerde el contenido de cada uno de ellos (puede ocurrir que algunos directorios no contengan archivos en el proyecto que está explorando).

3. ¿Cómo explorar en Eclipse un proyecto Java que esté abierto?

- Utilice la vista llamada "Package Explorer". Si la vista anterior no está disponible, búsquela en el menú *Window/ Show View/Package Explorer*.
- Revise las propiedades del proyecto. Puede editar las propiedades haciendo clic derecho sobre el proyecto o mediante el menú *Project/Properties*.
- Seleccione de la ventana de propiedades (de las opciones que aparecen a la izquierda) las opciones de construcción de Java ("*Java Build Path*") y revise la configuración del proyecto.
- Observe la estructura de paquetes del proyecto.

4. ¿Cómo editar una clase Java?

- Utilizando la vista llamada "*Package Explorer*" localice el directorio con los archivos fuente del proyecto.
- Dando doble clic sobre cualquiera de los archivos que allí se encuentran (Empleado.java, por ejemplo), el editor lo abre y permite al programador que lo modifique.
- Agregue un comentario en algún punto de la clase Empleado, teniendo cuidado de no afectar el contenido del archivo, y sálvelo de nuevo con la opción del menú *File/Save*.
- Cierre el archivo después de haberlo salvado.

5. ¿Cómo ejecutar el programa en un proyecto abierto en Eclipse?

- Utilizando la vista llamada "Package Explorer" localice el directorio con los archivos fuente del proyecto.
- Localice la clase InterfazEmpleado en el paquete que contiene las clases de la interfaz. Cada programa en Java tiene una clase por la cual comienza la ejecución. Siempre se debe localizar esta clase para poder iniciar el programa.
- Elija el comando "*Run/Java Application*". Puede hacerlo desde la barra de herramientas, el menú principal o el menú emergente que aparece al hacer clic derecho sobre la clase.
- Con este comando el programa comienza su ejecución. El programa y Eclipse siguen funcionando simultáneamente. Para terminar el programa, basta con cerrar su ventana.

- Localice la vista llamada consola. Si la vista no está disponible, búsquela en el menú *Window/Show View/Console*. Allí pueden aparecer algunos mensajes de error de ejecución. En esa vista hay un botón rojo pequeño, que permite terminar la ejecución del programa.

> Debe estar claro que el ambiente de desarrollo es una herramienta para el programador, y que lo normal es que dicho ambiente no esté instalado en el computador del usuario.

9. Hojas de Trabajo

9.1 Hoja de Trabajo Nº 1: Una Encuesta

Descargue esta hoja de trabajo a través de los siguientes enlaces: Descargar PDF | Descargar Word.

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

Se quiere crear una aplicación que permita realizar encuesta de opinión de un curso y manejar sus resultados. La encuesta consiste en una única pregunta, en la cual se le pide a la persona que califique la calidad de un curso dando un valor entre 0 y 10.

Se desea poder conocer los resultados de la en cuenta para diferentes sectores demográficos. Para esto se tendrá en cuenta el rango el rango de edad y el estado civil de la persona que puede ser soltero(a) o casado(a). En la encuesta se dividieron las personas en 3 rangos de edad: (1) menores de 18, (2) entre 18 y 54, y (3) con 55 o más años.

En el momento de hacer la pregunta, la persona debe seleccionar su rango de edad, informar si es soltera o casada y agregar una nueva opinión a la encuesta.

El programa debe informar el promedio total de la encuesta. Esto es, debe promediar todas las notas dadas y presentar el resultado en pantalla. También debe debe ser capaz de informar valores parciales de la encuesta. En ese caso se debe especificar un rango de edad y un estado civil. El programa presenta por pantalla el promedio de las calificaciones del curso dadas por todas las personas que cumplen el perfil pedido. Puede suponer que en el momento de calcular los resultados hay por lo menos una persona de cada perfil.

La interfaz de usuario de este programa es la que se muestra a continuación:


Encuesta del curso
LA ENCUESTA
Agregar opinión a encuesta
Bienvenido(a) a nuestra encuesta!
Por favor, seleccione su rango de edad:
0-17 años
>>
Opciones
Opción 1
Opción 2


Encuesta del curso
LA ENCUESTA
Agregar opinión a encuesta
Bienvenido(a) a nuestra encuesta!
<<
Su estado civil es:
Casado(a)
>>
Opciones
Opción 1
Opción 2


Encuesta del curso
LA ENCUESTA
Agregar opinión a encuesta
Bienvenido(a) a nuestra encuesta!
<<
Califique de 0 a 10 el curso:
4
Enviar
0
5
10
Opciones
Opción 1
Opción 2

Requerimientos funcionales. Describa tres requerimientos funcionales de la aplicación 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	

Entidades del mundo. Identifique las entidades del mundo y descríbalas brevemente.

Entidad	Descripción

Características de las entidades. Identifique las características de cada una de las entidades y escriba la clase en UML con el tipo de datos adecuado.

Entidad 1

Atributo	Valores Posibles

Diagrama UML

Entidad 2

Atributo	Valores Posibles
	
	
	

Diagrama UML


Relaciones entre entidades. Dibuje las entidades en UML (sin atributos ni métodos) y las relaciones que existan entre ellas.

Métodos de las entidades. Lea las siguientes descripciones de métodos y escriba su implementación en el lenguaje Java.

Método 1

Clase	RangoEdad
Nombre	darNumeroCasados
Parámetros	Ninguno.
Retorno	El número de personas casadas que respondieron la encuesta, en el rango de edad de la clase.
Descripción	Retorna el número de personas casadas que respondieron la encuesta, en el rango de edad de la clase.

Implementación en Java


Método 2

Clase	RangoEdad
Nombre	darTotalOpinionCasados
Parámetros	Ninguno.
Retorno	La suma de todas las opiniones de los encuestados casados en el rango de edad de la clase.
Descripción	Retorna la suma de todas las opiniones de los encuestados casados en el rango de edad de la clase.

Implementación en Java


Método 3

Clase	RangoEdad
Nombre	calcularPromedio
Parámetros	Ninguno.
Retorno	El promedio de la encuesta en el rango de edad de la clase.
Descripción	Retorna el promedio de la encuesta en el rango de edad de la clase. Para esto suma todas las opiniones y divide por el número total de encuestados.

Implementación en Java


Método 4

Clase	RangoEdad
Nombre	agregarOpinionCasado
Parámetros	Opinión del encuestado.
Retorno	Ninguno.
Descripción	Añade la opinión de una persona casada en el rango de edad que representa la clase.

Implementación en Java


Método 5

Clase	RangoEncuesta
Nombre	darPromedioCasados
Parámetros	Ninguno.
Retorno	El promedio de la encuesta en el rango de edad de la clase considerando sólo los casados.
Descripción	Retorna el promedio de la encuesta en el rango de edad de la clase. Para esto suma todas las opiniones de los casados y divide por el número total de ellos.

Implementación en Java


Método 6

Clase	Encuesta
Nombre	agregarOpinionRango1Casado
Parámetros	Opinión del encuestado.
Retorno	Ninguno.
Descripción	Añade la opinión de una persona casada en el rango de edad 1 de la encuesta.

Implementación en Java


Método 7

Clase	Encuesta
Nombre	agregarOpinionRango2Soltero
Parámetros	(1) estado civil, (2) opinión.
Retorno	Ninguno.
Descripción	Añade la opinión de una persona soltera en el rango de edad 2 de la encuesta.

Implementación en Java


Método 8

Clase	Encuesta
Nombre	calcularPromedio
Parámetros	Ninguno.
Retorno	El promedio de la encuesta en todos los rangos de edad.
Descripción	Retorna el promedio de la encuesta en todos los rangos de edad. Para esto suma todas las opiniones y divide por el número total de encuestados.

Implementación en Java


Método 9

Clase	Encuesta
Nombre	darPromedioCasados
Parámetros	Ninguno.
Retorno	El promedio de la encuesta en todos los rangos de edad de la clase, considerando sólo los casados.
Descripción	Retorna el promedio de la encuesta en todos los rangos de edad. Para esto suma todas las opiniones de los casados y divide por el número total de ellos.

Implementación en Java


9.2 Hoja de Trabajo Nº 2: Una Alcancía

Descargue esta hoja de trabajo a través de los siguientes enlaces: Descargar PDF | Descargar Word.

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

Se quiere construir un programa para manejar una alcancía. En la alcancía es posible guardar monedas de distintas denominaciones: $50, $100, $200, $500 y $1000. No se guardan billetes o monedas de otros valores.

El programa debe dar las siguientes opciones: (1) agregar una moneda de una de las denominaciones que maneja, (2) informar cuántas monedas tiene de cada denominación, (3) calcular el total de dinero ahorrado y (4) romper la alcancía, vaciando su contenido.

La interfaz de usuario de este programa es la que se muestra a continuación:

Requerimientos funcionales. Describa tres requerimientos funcionales de la aplicación que haya identificado en el enunciado.

Requerimiento Funcional 1

Nombre	R1 – Guardar una moneda de $50 en la alcancía.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	R2 – Contar el número de monedas de $50 que hay en la alcancía.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 3

Nombre	R3 – Calcular el total de dinero ahorrado en la alcancía.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 4

Nombre	R4 – Romper la alcancía.
Resumen	
Entradas	
Resultado	

Entidades del mundo. Identifique las entidades del mundo y descríbalas brevemente.

Entidad	Descripción
	

Características de las entidades. Identifique las características de cada una de las entidades y escriba la clase en UML con el tipo de datos adecuado.

Entidad 1

Atributo	Valores Posibles

Diagrama UML


Métodos de las entidades. Complete las siguientes descripciones de métodos y escriba su implementación en el lenguaje Java.

Método 1

Clase	Alcancia
Nombre	AgregarMoneda50
Parámetros	
Retorno	
Descripción	

Implementación en Java

Método 2

Clase	Alcancia
Nombre	AgregarMoneda500
Parámetros	
Retorno	
Descripción	

Implementación en Java

Método 3

Clase	Alcancia
Nombre	darTotalDinero
Parámetros	
Retorno	
Descripción	

Implementación en Java


Método 4

Clase	Alcancia
Nombre	darNumeroMonedas100
Parámetros	
Retorno	
Descripción	

Implementación en Java

Método 5

Clase	**Alcancia**
Nombre	romperAlcancia
Parámetros	
Retorno	
Descripción	

Implementación en Java

02 DEFINICIÓN DE SITUACIONES Y MANEJO DE CASOS

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Modelar las características de un objeto, utilizando nuevos tipos simples de datos y la técnica de definir constantes y enumeraciones para representar los valores posibles de un atributo.
- Utilizar expresiones como medio para identificar una situación posible en el estado de un objeto y para indicar la manera de modificar dicho estado.
- Utilizar las instrucciones condicionales simples y compuestas como parte del cuerpo de un método, para poder considerar distintos casos en la solución de un problema.
- Identificar de manera informal los métodos de una clase, utilizando para esto la técnica de agrupar los métodos por tipo de responsabilidad que tienen: construir, modificar o calcular.

2. Motivación

En el nivel anterior se introdujo la noción de un programa como la solución a un problema planteado por un cliente. Para construir dicho programa, se presentaron y utilizaron los elementos conceptuales, tecnológicos y metodológicos necesarios para enfrentar problemas triviales. A medida que los problemas comienzan a ser más complejos, es preciso ir extendiendo dichos elementos. En este nivel vamos a introducir nuevos elementos en tres direcciones:

1. Nuevas maneras de modelar una característica.
2. La posibilidad de considerar casos alternativos en el cuerpo de un método.
3. Algunas técnicas para identificar los métodos de una clase.

En los siguientes párrafos se muestra la necesidad de estas extensiones dentro del proceso de desarrollo de programas.

¿Por qué necesitamos nuevas maneras de modelar una característica? Aunque con los tipos de datos para manejar enteros, reales y cadenas de caracteres se puede cubrir un amplio espectro de casos, en este nivel veremos nuevos tipos de datos y nuevas técnicas para representar las características de las clases. También aprovecharemos para profundizar en los tipos de datos estudiados en el nivel anterior.

¿Por qué es necesario poder considerar casos en el cuerpo de un método? Con las instrucciones que se presentaron en el nivel anterior, sólo es posible asignar un valor a un atributo, pedir un servicio a un objeto con el cual se tiene una asociación, o retornar un resultado. Por ejemplo, si en el caso del empleado del nivel 1 existiera una norma de la empresa por la que se diera una bonificación en el salario a aquellos empleados que llevan más de 10 años trabajando con ellos, sería imposible incluirla en el programa. No habría manera de verificar si el empleado cumple con esa condición para sumarle la bonificación al salario. Allí habría dos casos distintos, cada uno con un algoritmo diferente para calcular el salario.

¿Por qué necesitamos técnicas para clasificar los métodos de una clase? Uno de los puntos críticos de la programación orientada por objetos es lo que se denomina la asignación de responsabilidades. Dado que la solución del problema se divide entre muchos algoritmos repartidos por todas las clases (que pueden ser centenares), es importante tener clara la manera de definir quién debe hacer qué. En el nivel 4 nos concentraremos en discutir en detalle este punto; por el momento, vamos a sentar las bases para poder avanzar en esa dirección.

Además de los nuevos elementos antes mencionados, en este nivel trataremos de reforzar y completar algunas de las habilidades generadas en el lector en el nivel anterior. La programación, más que una actividad basada en el conocimiento de enormes cantidades de conceptos y definiciones, es una actividad de habilidades, utilizables en múltiples contextos. Por eso, en la estructura de este libro, se le da mucha importancia a las tareas, cuyo objetivo es trabajar en la manera de usar los conceptos que se van viendo.

3. El Primer Caso de Estudio

En este caso, tenemos un programa que permite manejar el inventario de una pequeña tienda, conocer cuánto dinero hay en caja y tener un control de estadísticas de venta.

La tienda maneja cuatro productos, para cada uno de los cuales se debe manejar la siguiente información:

1. Nombre. No pueden haber dos productos con el mismo nombre.
2. Tipo (puede ser un producto de papelería, de supermercado o de droguería).
3. Cantidad actual del producto en la tienda (número de unidades disponibles para la venta que hay en la bodega).
4. Cantidad mínima para abastecimiento (número de productos por debajo del cual se puede hacer un nuevo pedido al proveedor).
5. El precio base de venta por unidad.

Para calcular el precio final de cada producto, se deben sumar los impuestos que define la ley (IVA). Dichos impuestos dependen del tipo del producto, de la siguiente manera:

- Papelería: 16%
- Supermercado: 4%
- Droguería: 12%.

Eso quiere decir que si un lápiz tiene un precio base de $10, el precio final será de $11,6 considerando que un lápiz es un producto de papelería, y sobre estos se debe pagar el 16% de impuestos.

El programa de manejo de esta tienda debe permitir las siguientes operaciones:

1. Vender un producto.
2. Abastecer la tienda con un producto.
3. Cambiar un producto.
4. Calcular estadísticas de ventas la tienda. Dichas estadísticas son: (a) el producto más vendido, (b) El producto menos vendido, (c) la cantidad total de dinero obtenido por las ventas de la tienda, (d) la cantidad de dinero promedio obtenido por unidad de producto vendida.

3.1. Comprensión del Problema

Tal como planteamos en el nivel anterior, el primer paso para poder resolver un problema es entenderlo. Este entendimiento lo mostramos descomponiendo el problema en tres aspectos: los requerimientos funcionales, el modelo conceptual y los requerimientos no funcionales. En la primera tarea de este nivel trabajaremos los dos primeros puntos.

Tarea 1

Objetivo: Entender el problema del caso de estudio de la tienda.

1. Lea detenidamente el enunciado del caso de estudio de la tienda.
2. Identifique y complete la documentación de los cuatro requerimientos funcionales.
3. Construya un primer diagrama de clases con el modelo conceptual, en el que sólo aparezcan las clases, las asociaciones y los atributos sin tipo.

Requerimiento Funcional 1

Nombre	R1 – Vender un producto.
Resumen	Permite vender una cantidad dada de unidades de un producto.
Entradas	(1) el nombre del producto, (2) la cantidad de unidades.
Resultado	Si había suficiente cantidad de producto en bodega, se vende (disminuye en bodega) la cantidad total pedida por el cliente. Si no, se vende (disminuye en bodega) la cantidad total existente en bodega. Se guarda en la caja de la tienda el dinero resultado de la venta. Se informa al usuario la cantidad de unidades vendidas.

Requerimiento Funcional 2

Nombre	R2 – Abastecer la tienda con un producto.
Resumen	Se abastece la tienda con la cantidad de unidades indicada por el usuario. El abastecimiento sólo se puede realizar si la cantidad de productos en bodega es menor que la cantidad mínima del producto.
Entradas	
Resultado	

Requerimiento Funcional 3

Nombre	R3 – Cambiar un producto.
Resumen	Permite cambiar la información de un producto vendido en la tienda.
Entradas	1) Nombre actual, 2)Nuevo nombre, 3) Tipo, 4) Valor unitario, 5) Cantidad en bodega, 6) Cantidad mínima.
Resultado	Se actualiza la información del producto.

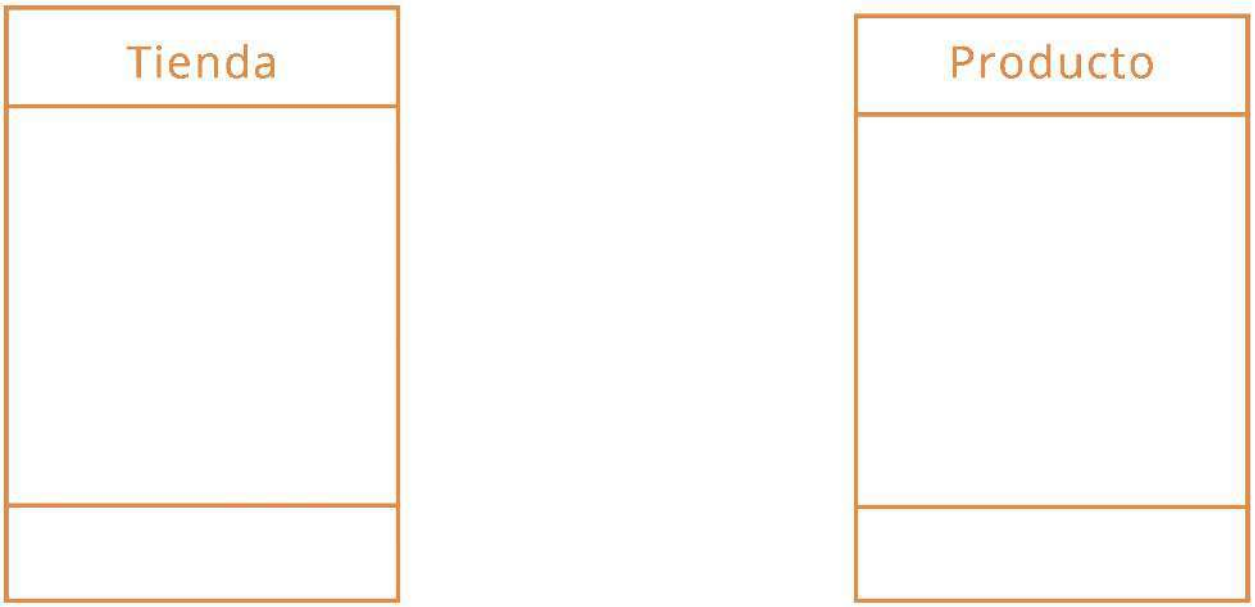
Requerimiento Funcional 4

Nombre	R4 – Calcular estadísticas de ventas.
Resumen	Calcula y muestra las siguientes estadísticas: (a) el producto más vendido; (b) el producto menos vendido; (c) la cantidad total de dinero obtenido por las ventas de la tienda; (d) el promedio de ventas de la tienda.
Entradas	Ninguna.
Resultado	Se muestra la información estadística de ventas.

Modelo conceptual:

En el enunciado se identifican dos entidades: la tienda y el producto. Defina los atributos de cada una de ellas, sin especificar por ahora su tipo.

Dibuje las asociaciones entre las clases y asigne a cada asociación un nombre y una dirección.



3.2 Definición de la Interfaz de Usuario

El diseño de la interfaz de usuario es una de las actividades que debemos realizar como parte del diseño de la solución al problema. En la figura 2.1 presentamos el diseño que decidimos para la interfaz del caso de estudio.

Fig. 2.1 Interfaz de usuario para el caso de estudio de la tienda

- La ventana del programa tiene dos zonas: en la primera aparece la información de los productos de la tienda. Allí se tiene el nombre de cada producto, la cantidad disponible en la bodega de la tienda, el IVA que se debe pagar por el producto, su precio antes de impuestos y si ya se debe hacer o no un pedido.
- En esta zona también tenemos tres botones, cada uno asociado con un requerimiento funcional. Desde allí podemos vender el producto a un cliente, abastecer la tienda con el producto, o modificar la información del producto.
- Cuando el usuario selecciona las opciones Vender o Abastecer, la aplicación presenta un diálogo en el que el usuario puede indicar el número de unidades deseadas.
- Cuando el usuario selecciona la opción Cambiar, la aplicación presenta un diálogo en el que el usuario puede ingresar la nueva información del producto.
- En la última de las zonas se encuentran los botones que permiten pedir la información estadística correspondiente al último requerimiento funcional.

4. Nuevos Elementos de Modelado

4.1. Tipos Simples de Datos

En esta sección presentamos dos nuevos tipos simples de datos (`boolean` y `char`) y volvemos a estudiar algunos aspectos de los tipos introducidos en el capítulo anterior.

Comenzamos con el tipo `double` . Para facilitar el modelado de las características que toman valores reales, la mayoría de los lenguajes de programación proveen un tipo simple denominado `double` . En el caso de estudio de la tienda usaremos un atributo de este tipo para modelar el precio de cada producto. Esto nos va a permitir tener un producto cuyo precio sea, por ejemplo, $23,12 (23 pesos y 12 centavos).

Se denomina **literal** de un tipo de datos a un valor constante de dicho tipo. En la siguiente tabla se dan algunos ejemplos de la manera de escribir literales para los tipos de datos estudiados. A medida que vayamos viendo nuevos tipos, iremos introduciendo la sintaxis que utilizan.

Tipo en Java	Ejemplo de literales	Comentarios
entero (`int`)	564, -12	Los literales de tipo entero se expresan como una secuencia de dígitos. Si el valor es negativo, dicha secuencia va precedida del símbolo "-".
real (`double`)	564.78, -98.3	Los literales de tipo real se expresan como una secuencia de dígitos. Para separar la parte entera de la parte decimal se utiliza el símbolo ".".
cadena de caracteres (`String`)	"esta es una cadena","", ""	Los literales de tipo cadena de caracteres van entre comillas dobles. Dos comillas dobles seguidas indican una cadena de caracteres vacía. Es distinta una cadena vacía que una cadena que sólo tiene un carácter de espacio en blanco.

En el ejemplo 1 se muestra la manera de declarar y manipular los atributos de tipo `double` usando el caso de estudio de la tienda. También se presenta la manera de convertir los valores reales a valores enteros.

Ejemplo 1

Objetivo: Repasar la manera de manejar atributos de tipo `double` en el lenguaje de programación Java, usando el caso de estudio de la tienda.

```
public class Producto
{
    private  double  valorUnitario;
}
```

- Declaración del atributo valorUnitario dentro de la clase Producto, para representar el precio del producto por unidad, antes de impuestos (sin IVA).
- Como de costumbre, el atributo lo declaramos privado, para evitar que sea manipulado desde fuera de la clase.

Las siguientes instrucciones pueden ir como parte de cualquier método de la clase Producto:

```
valorUnitario = 23.12;
```

- En cualquier método de la clase se puede asignar un literal de tipo real al atributo.

```
int valorPesos = ( int ) valorUnitario;
```

- Si en la variable valor valorPesos queremos tener la parte entera del precio del producto, utilizamos el operador de conversión (`int`). Este operador permite convertir valores reales a enteros.
- El operador (`int`) incluye los paréntesis y debe ir antes del valor que se quiere convertir.
- Si no se incluye el operador de conversión, el compilador va a señalar un error ("*Type mismatch: cannot convert from double to int*").

```
valorUnitario = valorUnitario / 1.07;
```

Para construir una expresión aritmética de valor real, se pueden usar los operadores de suma(`+`), resta (`-`), multiplicación (`*`) y división (`/`).

```
int valorPesos = 17 / 3;
```

- La división entre valores enteros da un valor entero. En el caso del ejemplo, después de la asignación, la variable valorPesos tendrá el valor 5.
- El lenguaje Java decide en cada caso (dependiendo del tipo de los operandos) si utiliza la división entera o la división real para calcular el resultado.

Un operador que se utiliza frecuentemente en problemas aritméticos es el operador módulo (`%`). Este operador calcula el residuo de la división entre dos valores, y se puede utilizar tanto en expresiones enteras como reales. La siguiente tabla muestra el resultado de aplicar dicho operador en varias expresiones.

Expresión	Valor	Comentarios
$4\%4$	0	El residuo de dividir 4 por 4 es cero. El resultado de este operador se puede ver como lo que "sobra" después de hacer la división entera.
$14\%3$	2	El resultado de la expresión es 2, puesto que al dividir 14 por 3 se obtiene como valor entero 4 y "sobran" 2.
$17\%3$	2	En esta expresión el valor entero es 5 (5 * 3 es 15) y "sobran" de nuevo 2.
$3\%17$	3	La división entera entre 3 y 17 es cero, así que "sobran" 3.
$4.5\%2.2$	0.1	El operador % se puede aplicar también a valores reales. En la expresión del ejemplo, 2.2. está 2 veces en 4.5 y "sobra" 0.1.

Otro tipo simple de datos que encontramos en los lenguajes de programación es el que permite representar valores lógicos (verdadero o falso). El nombre de dicho tipo es `boolean` . Imagine, por ejemplo, que en la tienda queremos modelar una característica de un producto que dice si es subsidiado o no por el gobierno. De esta característica sólo nos interesaría saber si es verdadera o falsa (los únicos valores posibles), para saber si hay que aplicar o no el respectivo descuento. Este tipo de características se podría modelar usando un entero y una convención sobre la manera de interpretar su valor (por ejemplo, 1 es verdadero y 2 es falso). Es tan frecuente encontrar esta situación que muchos lenguajes resolvieron convertirlo en un nuevo tipo de datos y evitar así tener que usar otros tipos para representarlo.

El tipo `boolean` sólo tiene dos literales: **true** y **false**. Estos son los únicos valores constantes que se le pueden asignar a los atributos o variables de dicho tipo.

Ejemplo 2

Objetivo: Mostrar la manera de manejar atributos de tipo `boolean` en el lenguaje de programación Java.

En este ejemplo se utiliza una extensión del caso de estudio de la tienda, para mostrar la sintaxis de declaración y el uso de los atributos de tipo `boolean` .

```
public class Producto
{
    private boolean subsidiado;
}
```

- Aquí se muestra la declaración del atributo "subsidiado" dentro de la clase Producto.
- Dicha característica no forma parte del caso de estudio y únicamente se utiliza en este ejemplo para ilustrar el uso del tipo de datos `boolean`.

Las siguientes instrucciones pueden ir como parte de cualquier método de la clase Producto:

```
subsidiado = true;
subsidiado = false;
```

- Los únicos valores que se pueden asignar a los atributos de tipo `boolean` son true y false. Los operadores que nos permitirán crear expresiones con este tipo de valores, los veremos más adelante.

```
boolean valorLogico = subsidiado;
```

- Es posible tener variables de tipo `boolean`, a las cuales se les puede asignar cualquier valor de dicho tipo.

El último tipo simple de dato que veremos en este capítulo es el tipo `char`, que sirve para representar un carácter. En el ejemplo 3 se ilustra la manera de usarlo dentro del contexto del caso de estudio. Un valor de tipo `char` se representa internamente mediante un código numérico llamado UNICODE.

Ejemplo 3

Objetivo: Mostrar la manera de manejar atributos de tipo `char` en el lenguaje de programación Java, usando una extensión del caso de estudio de la tienda.

Suponga que los productos de la tienda están clasificados en tres grupos: A, B y C, según su calidad. En este ejemplo se muestra una manera de representar dicha característica usando un atributo de tipo `char`.

```
public class Producto
{
    private char calidad;
}
```

- Aquí se muestra la declaración del atributo "calidad" dentro de la clase Producto. Dicha característica será representada con un carácter que puede tomar como valores 'A', 'B' o 'C'.

Las siguientes instrucciones pueden ir como parte de cualquier método de la clase Producto:

```
calidad  =  'A';
calidad  =  'B';
```

- Los literales de tipo `char` se expresan entre comillas sencillas. En eso se diferencian de los literales de la clase String, que van entre comillas dobles.

```
calidad  =  67;
```

- Lo que aparece en este ejemplo es poco usual: es posible asignar directamente un código UNICODE a un atributo de tipo `char`. El valor 67, por ejemplo, es el código interno del carácter 'C'. El código interno del carácter 'c' (minúscula) es 99. Cada carácter tiene su propio código interno, incluso los que tienen tilde (el código del carácter 'á' es 225).

```
char valorCaracter = calidad;
```

- Es posible tener variables de tipo `char`, a las cuales se les puede asignar cualquier valor de dicho tipo.

4.2. Constantes para Representar Valores Inmutables

En muchos problemas encontramos algunos valores que no van a cambiar durante la ejecución del programa (inmutables). Considere el caso de la tienda, en el que el valor del precio final del producto depende de los impuestos definidos por la ley. Según lo que vimos en el nivel anterior, cada vez que necesitemos el valor del IVA de los productos de papelería, debemos escribir su valor numérico (0.16). Para facilitar la lectura y escritura del código, los lenguajes de programación permiten asociar un nombre significativo al valor, para así reemplazar el valor numérico dentro del código. Estos nombres asociados se denominan **constantes**.

Estas constantes pueden ser de cualquier tipo de datos (por ejemplo, puede haber una constante de tipo String o `double`) y se les debe fijar su valor desde la declaración. Dicho valor no puede ser modificado en ningún punto del programa.

El ejemplo 4 desarrolla esa idea con el caso de la tienda y muestra la sintaxis en Java para declarar y usar constantes.

Ejemplo 4

Objetivo: Mostrar el uso de constantes para representar los valores inmutables, usando el caso de estudio de la tienda.

En este ejemplo ilustramos el uso de constantes para representar los posibles valores del IVA de los productos.

```
public class Producto
{
    //--------------------------------------
    // Constantes
    //--------------------------------------

    private final static double IVA_PAPELERIA = 0.16;
    private final static double IVA_SUPERMERCADO = 0.04;
    private final static double IVA_FARMACIA = 0.12;
    ...
}
```

- Declaramos tres constantes que tienen los valores posibles del IVA en el problema: 16%, 12% y 4%. Estas constantes se llaman IVA_FARMACIA, IVA_PAPELERIA e IVA_SUPERMERCADO.
- Son constantes de tipo `double`, puesto que de ese tipo son los valores inmutables que queremos representar.
- Las constantes se declaran privadas si no van a ser usadas por fuera de la clase.
- Para inicializar una constante, se debe elegir un literal del mismo tipo de la constante, o una expresión.
- Dentro de la declaración de la clase, se agrega una zona para declarar las constantes. Es conveniente situar esa zona antes de la declaración de los atributos.

Las siguientes instrucciones pueden ir como parte de cualquier método de la clase Producto:

```
precio = valorUnitario * ( 1 + IVA_SUPERMERCADO );
precio = valorUnitario * ( 1 + 0.04 );
```

- Las constantes sólo sirven para reemplazar el valor que representan. Las dos instrucciones del ejemplo son equivalentes y permiten calcular el precio al consumidor, aplicándole un IVA del 4% al precio de base del producto.
- La ventaja de las constantes es que cuando alguien lee el programa entiende a qué corresponde el valor 0.04 (puesto que también podría corresponder a los intereses o algún otro tipo de impuesto).

Esta práctica de definir constantes en sustitución de aquellos valores que no cambian durante la ejecución tiene muchas ventajas y es muy apreciada cuando hay necesidad de hacer el mantenimiento a un programa. Suponga, por ejemplo, que el gobierno autoriza un incremento en los impuestos y, ahora, el impuesto sobre los productos de supermercado pasa del 4% al 6%. Si dentro del programa siempre utilizamos la constante IVA_SUPERMERCADO para referirnos al valor del impuesto sobre los productos de supermercado, lo único que debemos hacer es reemplazar el valor 0.04 por 0.06 en la declaración de la constante. Si por el contrario, en el código del programa no utilizamos el nombre de la constante sino el valor, tendríamos que ir a buscar todos los lugares en el código donde aparece el valor 0.04 (que hace referencia al impuesto sobre los productos de supermercado) y reemplazarlo por 0.06. Si hacemos lo anterior, fácilmente podemos pasar por alto algún lugar e introducir así un error en el programa.

> Por convención, las constantes siempre van en mayúsculas. Si el nombre de la constante contiene varias palabras, es usual separarlas con el carácter "_". Por ejemplo podríamos tener una constante llamada PRECIO_MAXIMO.

Imaginémonos una nueva constante en la clase producto que define el precio máximo que puede tener un producto.

```
public class Producto
{
    //--------------------------------------
    // Constantes
    //--------------------------------------

    public final static double PRECIO_MAXIMO = 500000.0;
}
```

El siguiente método podría pertenecer a la clase Tienda:

```
public class Tienda
{
    public double ejemplo( )
    {
        return Producto.PRECIO_MAXIMO;
    }
}
```

- Por fuera de la clase Producto, las constantes pueden usarse indicando la clase en la cual fueron declaradas (siempre y cuando hayan sido declaradas como `public` en esa clase).

4.3. Enumeraciones para Definir el Dominio de un Atributo

Considere el caso de la tienda, en el que queremos modelar la característica de tipo de producto, el cual puede ser de tres tipos distintos: supermercado, papelería o droguería. En el nivel anterior vimos que es posible utilizar el tipo entero para representar esta característica, y asociar un número con cada uno de los valores posibles. Sin embargo, para estos casos, los lenguajes de programación permiten agrupar estos posibles valores de la característica, asignando solamente un nombre significativo para cada uno de ellos, sin asignarles ningún valor. Estas agrupaciones de valores de datos se denominan **enumeraciones**. De esta forma, dentro de los métodos podemos usar los nombres existentes en una enumeración.

El ejemplo 5 desarrolla esa idea con el caso de la tienda y muestra la sintaxis en Java para declarar y usar enumeraciones.

Ejemplo 5

Objetivo: Mostrar el uso de enumeraciones para representar los valores posibles de alguna característica.

Usando el caso de estudio de la tienda, en este ejemplo se muestra una manera de crear una enumeración para representar la
característica de tipo de producto.

```
public class Producto
{
    //--------------------------------------
    // Enumeraciones
    //--------------------------------------
    public enum Tipo
    {
        PAPELERIA,
        SUPERMERCADO,
        FARMACIA
    }

    //--------------------------------------
    // Atributos
    //--------------------------------------
    private Tipo tipo;
    ...
}
```

- Se declara una enumeración llamada Tipo, para modelar el conjunto de nombres que podrán representar un tipo de producto.
- Dentro de la declaración del tipo, se agregan los nombres significativos de los tres tipos de producto existentes: PAPELERIA, SUPERMERCADO y FARMACIA.
- Se declara un atributo llamado "tipo" dentro de la clase Producto, para representar esa característica. El tipo asignado a este atributo es la enumeración que creamos arriba.
- Dentro de la declaración de la clase, se agrega una zona para declarar las constantes. Es conveniente situar esa zona antes de la declaración de los atributos.

Para poder usar una enumeración, se debe escribir el nombre de la enumeración y después llamar el valor que se desea asignar. Las siguientes instrucciones pueden ir como parte de cualquier método de la clase Producto:

```
tipo = Tipo.PAPELERIA;
tipo = Tipo.SUPERMERCADO;
tipo = Tipo.FARMACIA;
```

Cualquiera de esas tres asignaciones define el tipo de un producto (no las tres a la vez, por supuesto). La ventaja de usar una enumeración (PAPELERIA) en lugar de un valor numérico es que el programa resultante es mucho más fácil de leer y entender.

El siguiente método podría pertenecer a la clase Tienda:

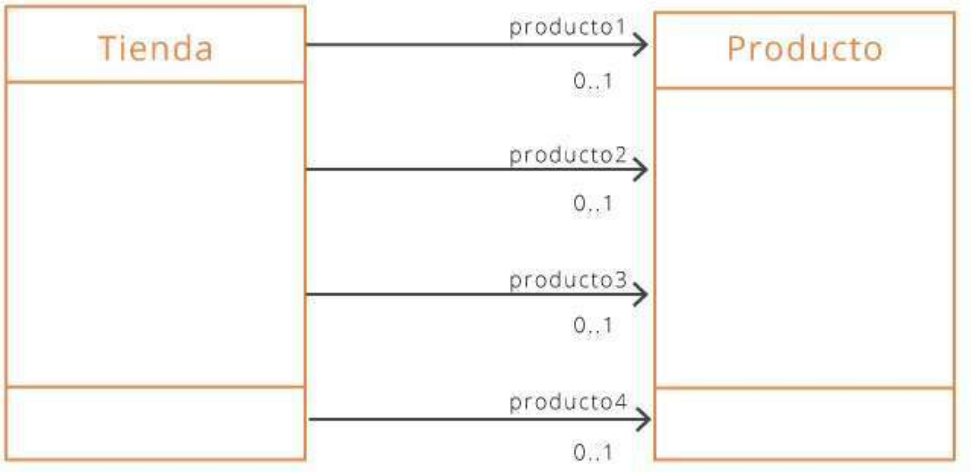
```
public void ejemplo( )
{
    ...
    tipoVenta = Tipo.PAPELERIA;
    tipoCompra = Tipo.SUPERMERCADO;
    ...
}
```

- Por fuera de la clase Producto, las enumeraciones se llaman de la misma manera que se llamaba dentro de la clase donde fueron declaradas (siempre y cuando hayan sido declaradas como public en esa clase).
- En el ejemplo estamos suponiendo que `tipoVenta` y `tipoCompra` son atributos de la clase Tienda.

4.4. Manejo de Asociaciones Opcionales

Supongamos que queremos modificar el enunciado del caso de la tienda, para que el programa pueda manejar 1, 2, 3 ó 4 productos. Lo primero que debemos hacer entonces es modificar el diagrama de clases, para indicar que las asociaciones pueden o no existir. Para esto usamos la sintaxis de UML que se ilustra en la figura 2.2, y que dice que las asociaciones son opcionales. Esta característica se denomina **cardinalidad** de la asociación y se verá más a fondo en el nivel 3. Por ahora podemos decir que la cardinalidad define el número de instancias de una clase que pueden manejarse a través de una asociación. En el caso de una asociación opcional, la cardinalidad es 0..1 (para expresar la cardinalidad, se usan dos números separados con dos puntos), puesto que a través de la asociación puede manejarse un objeto de la otra clase o ningún objeto.

Fig. 2.2 Diagrama de clases con asociaciones opcionales

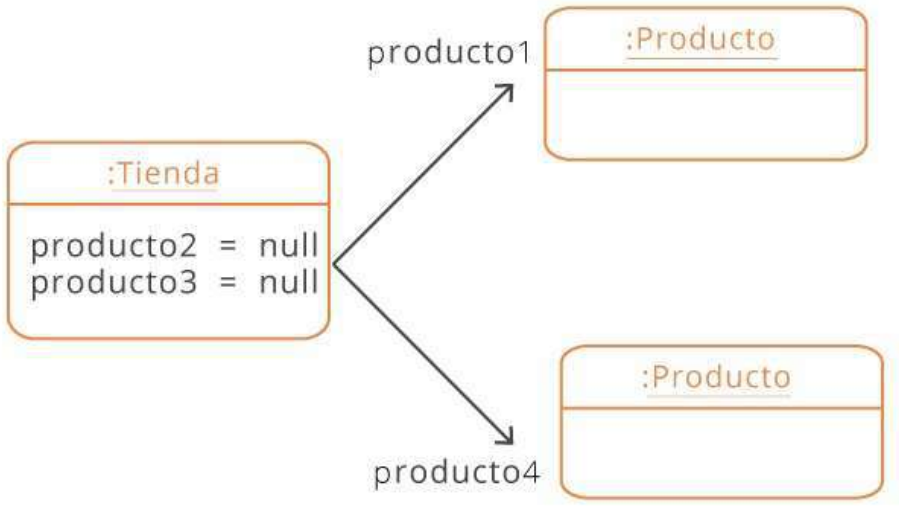


- La cardinalidad de la asociación llamada producto1 entre la clase Tienda y la clase Producto es cero o uno (0..1), para indicar que puede o no existir el objeto que representa la asociación producto1. Lo mismo sucede con cada una de las demás asociaciones.
- Si en el diagrama no aparece ninguna cardinalidad en una asociación, se interpreta como que ésta es 1 (existe exactamente un objeto de la otra clase).
- En la figura 2.3 aparece un ejemplo de un diagrama de objetos para este diagrama de clases.

Dentro de un método, para indicar que el objeto correspondiente a una asociación que no está presente (que no hay, por ejemplo, un objeto de la clase Producto para la asociación producto1) se utiliza el valor especial `null` (`producto1 = null;`). En la figura 2.3 se muestra un ejemplo de un diagrama de objetos para el modelo conceptual de la figura anterior.

> Cuando se intenta llamar un método a través de una asociación cuyo valor es null, el computador muestra el error: *NullPointerException*.

Fig. 2.3 Diagrama de objetos con asociaciones opcionales



El ejemplo anterior lo utilizamos únicamente para ilustrar la idea de una asociación opcional. En el resto del capítulo seguiremos trabajando con el caso inicial, en el cual todas las asociaciones entre la clase Tienda y la clase Producto tienen cardinalidad 1, tal como se muestra en el ejemplo 6.

Ejemplo 6

Objetivo: Mostrar las declaraciones de las clases Tienda y Producto que vamos a usar en el resto del capítulo.

En este ejemplo se muestra un diseño posible para las clases del caso de estudio de la tienda. Se presenta tanto el diagrama de clases en UML como las respectivas declaraciones en Java. En el diseño se incluyen los métodos de cada una de las clases.

El diagrama de clases consta de las clases Tienda y Producto, con 4 asociaciones entre ellos (todas de cardinalidad 1). Para cada clase se muestran los atributos que modelan las características importantes para el problema. Entre los principales atributos de la clase Producto están su nombre, su tipo, su valor unitario antes de impuestos, etc.

```
public class Tienda
{
    //---------------------------------
    // Atributos
    //---------------------------------
    private Producto producto1;
    private Producto producto2;
    private Producto producto3;
    private Producto producto4;
    private double dineroEnCaja;
    ...

    //-----------------------------------------
    //Métodos
    //-----------------------------------------
    public Producto darProducto1( ) { ... }
    public Producto darProducto2( ) { ... }
    public Producto darProducto3( ) { ... }
    public Producto darProducto4( ) { ... }
    public double darDineroEnCaja( ) { ... }

}
```

- Se modelan los 4 productos, unidos a la tienda con las asociaciones llamadas producto1, producto2, producto3 y producto4. Fíjese que las asociaciones y los atributos se declaran siguiendo la misma sintaxis. El dinero total que hay en caja de la tienda se modela con un atributo de tipo `double`.
- Esta es la lista de signaturas de algunos de los métodos de la clase Tienda que utilizaremos en la siguiente sección. Esta lista se irá completando poco a poco, a medida que avancemos en el capítulo.

```
public class Producto
{
    //-----------------------------------------
    // Enumeraciones
    //-----------------------------------------
    /**
     * Enumeradores para los tipos de producto.
     */
    public enum Tipo
    {
        PAPELERIA,
        SUPERMERCADO,
        DROGUERIA
    }
    //-----------------------------------------
    // Constantes
    //-----------------------------------------
    private final static double IVA_PAPELERIA = 0.16;
    private final static double IVA_SUPERMERCADO = 0.04;
    private final static double IVA_DROGUERIA = 0.12;

    //-----------------------------------------
    // Atributos
    //-----------------------------------------
    private String nombre;
    private Tipo tipo;
    private double valorUnitario;
    private int cantidadBodega;
    private int cantidadMinima;
    private int cantidadUnidadesVendidas;

    //-----------------------------------------
    //Métodos
    //-----------------------------------------
    public String darNombre( ) { ... }
    public Tipo darTipo( ) { ... }
    public double darValorUnitario( ) { ... }
    public int darCantidadBodega( ) { ... }
    public int darCantidadMinima( ) { ... }
    public int darCantidadUnidadesVendidas( ) { ... }
}
```

- En la clase Producto, se declaran primero las constantes para representar los valores de modelado de los atributos. Luego, las constantes que representan valores inmutables.
- En la segunda zona va la declaración de los atributos de la clase.
- En la tercera zona se observa la lista de signaturas de algunos de los métodos de la clase Producto que utilizaremos en la siguiente sección. Esta lista se irá completando poco a poco, a medida que avancemos en el capítulo.

5. Expresiones

5.1. Algunas Definiciones

Una **expresión** es la manera en que expresamos en un lenguaje de programación algo sobre el estado de un objeto. Es el medio que tenemos para decir en un programa algo sobre el mundo del problema. En el nivel anterior vimos las expresiones aritméticas, que permitían definir la manera en que debía ser modificado el estado de un elemento del mundo, usando sumas y restas.

Las expresiones aparecen dentro del cuerpo de los métodos y están formadas por **operandos** y **operadores**. Los operandos pueden ser atributos, parámetros, literales, constantes o llamadas de métodos, mientras que los operadores son los que indican la manera de calcular el valor de la expresión. Los operadores que se pueden utilizar en una expresión dependen del tipo de los datos de los operandos que allí aparezcan.

> En algunos casos es indispensable utilizar paréntesis para evitar la ambigüedad en las expresiones. Por ejemplo, la expresión 10 – 4 – 2 puede ser interpretada de dos maneras, cada una con un resultado distinto: 10 – (4 – 2) = 8, o también (10 – 4) – 2 = 4. Es buena idea usar siempre paréntesis en las expresiones, para estar seguros de que la interpretación del computador es la que nosotros necesitamos.

Ejemplo 7

Objetivo: Ilustrar la manera de usar expresiones aritméticas para hablar del estado de un objeto.

Suponga que estamos en un objeto de la clase Producto. Vamos a escribir e interpretar algunas expresiones aritméticas simples.

La expresión...	Se interpreta como...
`valorUnitario * 2`	El doble del valor unitario del producto.
`cantidadBodega - cantidadMinima`	La cantidad del producto que hay que vender antes de poder hacer un pedido.
`valorUnitario * ( 1 + ( IVA_PAPELERIA / 2 ) )`	El precio final al consumidor si el producto debe pagar el IVA de los productos de papelería (16%) y sólo paga la mitad de éste.
`cantidadUnidadesVendidas * 1.1`	La cantidad de unidades vendidas del producto, inflado en un 10%.

5.2. Operadores Relacionales

Los lenguajes de programación cuentan siempre con operadores relacionales, los cuales permiten determinar un valor de verdad (verdadero o falso) para una situación del mundo. Si queremos determinar, por ejemplo, si el valor unitario antes de impuestos de un producto es menor que $10.000, podemos utilizar (dentro de la clase Producto) la expresión:

```
valorUnitario < 10000
```

Los operadores relacionales son seis, que se resumen en la siguiente tabla:

Significado	Símbolo	Ejemplo
Es igual que	`==`	`valorUnitario == 55.75`
Es diferente de	`!=`	`tipo != Tipo.PAPELERIA`
Es menor que	`<`	`cantidadBodega < 120`
Es mayor que	`>`	`cantidadBodega > cantidadMinima`
Es menor o igual que	`<=`	`valorUnitario <= 100.0`
Es mayor o igual que	`>=`	`valorUnitario >= 100.0`

Ejemplo 8

Objetivo: Ilustrar la manera de usar operadores relacionales para describir situaciones de un objeto (algo que es verdadero o falso).

Suponga que estamos en un objeto de la clase Producto. Vamos a escribir e interpretar algunas expresiones que usan operadores relacionales.

La expresión...	Se interpreta como..
`tipo == Tipo.DROGUERIA`	¿El producto es de droguería?
`cantidadBodega > 0`	¿Hay disponibilidad del producto en la bodega?
`totalProductosVendidos > 0`	¿Se ha vendido alguna unidad del producto?
`cantidadBodega <= cantidadMinima`	¿Ya es posible hacer un nuevo pedido del producto?

5.3. Operadores Lógicos

Los operadores lógicos nos permiten describir situaciones más complejas, a partir de la composición de varias expresiones relacionales o de atributos de tipo `boolean`. Los operadores lógicos son tres: `&&` (y), `||` (o), `!` (no), y el resultado de aplicarlos se

resume de la siguiente manera:

- operando1 && operando2 es cierto, si ambos operandos son verdaderos.
- operando1 || operando2 es cierto, si cualquiera de los dos operandos es verdadero.
- !operando es cierto, si el operando es falso.

> Los operadores `&&` y `||` se comportan de manera un poco diferente a todos los demás. La expresión en la que estén sólo se evalúa de izquierda a derecha hasta que se establezca si es verdadera o falsa. El computador no pierde tiempo evaluando el resto de la expresión si ya sabe cual será su resultado.

Ejemplo 9

Objetivo: Ilustrar la manera de usar operadores lógicos para describir situaciones de un objeto (algo que es cierto o falso).

Suponga que estamos en un objeto de la clase Producto. Vamos a escribir e interpretar algunas expresiones que usan operadores lógicos.$x = y$

La expresión...	Se interpreta como...
`tipo == Tipo.SUPERMERCADO && cantidadUniadesVendidas== 0`	¿El producto es de supermercado y no se ha vendido ninguna unidad? En este caso, si el producto no es de supermercado o ya se ha vendido alguna unidad, la expresión es falsa.
`valorUnitario >= 10000 && valorUnitario <= 20000 && tipo == Tipo.DROGUERIA`	¿El producto vale entre $10.000 y $20.000 y, además, es un producto de droguería?
`!( tipo == Tipo.PAPELERIA )`	¿El producto no es de papelería? Note que esta expresión es equivalente a la expresión que va en la siguiente línea. Y también es equivalente a (tipo != Tipo.PAPELERIA).
`tipo == Tipo.SUPERMERCADO || tipo == Tipo.DROGUERIA`	¿El producto es de supermercado o de droguería?

Operadores sobre Cadenas de Caracteres

El tipo String nos sirve para representar cadenas de caracteres. A diferencia de los demás tipos de datos vistos hasta ahora, este tipo no es simple, sino que se implementa mediante una clase especial en Java. Esto implica que, en algunos casos, para invocar sus operaciones debemos utilizar la sintaxis de llamada de métodos.

Existen muchas operaciones sobre cadenas de caracteres, pero en este nivel sólo nos vamos a interesar en el operador de concatenación (`+`), en el de comparación (`equals`) y en el de extracción de un carácter (`charAt`).

El primer operador (`+`) sirve para pegar dos cadenas de caracteres, una después de la otra. Por ejemplo, si quisiéramos tener un método en la clase Producto que calculara el mensaje que se debe mostrar en la publicidad de la tienda, tendría la siguiente forma:

```
public String darPublicidad( )
{
    return "Compre el producto " + nombre + " por solo $" + valorUnitario;
}
```

- Si alguno de los operandos no es una cadena de caracteres (como es el caso del atributo de tipo real `valorUnitario`) el compilador se encarga de convertirlo a cadena. No es necesario hacer una conversión explícita porque el compilador lo hace automáticamente por nosotros, para todos los tipos simples de datos.
- Al ejecutar este método, retornará una cadena con algo del siguiente estilo: Compre el producto cuaderno por solo $100.50.

La segunda operación que nos interesa en este momento es la comparación de cadenas de caracteres. A diferencia de los tipos simples, en donde se utiliza el operador `==` , para poder comparar dos cadenas de caracteres es necesario llamar el método `equals` de la clase String. Por ejemplo, si queremos tener un método en la clase Producto que reciba como parámetro una cadena de caracteres e informe si el nombre del producto es igual al valor recibido como parámetro, éste sería más o menos así:

```
public boolean esIgual( String pBuscado )
{
    return nombre.equals( pBuscado );
}
```

- Se usa la sintaxis de invocación de métodos para poder utilizar el método equals. La razón es que `String` es una clase, y se deben respetar las reglas de llamada de un método (`int` , `double` y `boolean` no son clases, y por esta razón se puede utilizar el operador `==` directamente).
- El retorno del método equals es de tipo boolean, razón por la cual lo podemos retornar directamente como respuesta del método que queremos construir.
- En el ejemplo, el método equals se invoca sobre el atributo de la clase Producto llamado "nombre" y se le pasa como parámetro el valor recibido en "buscado".

La última operación que vamos a estudiar en este nivel nos permite "obtener" un carácter de una cadena. Para esto debemos dar la posición dentro de la cadena del carácter que nos interesa, e invocar el método *chatAt* de la clase String, tal como se muestra en los siguientes ejemplos. Nótese que el primer carácter de una cadena se encuentra en la posición 0.

Suponga que tenemos dos cadenas de caracteres, declaradas de la siguiente manera:

```
String cad1 = "la casa es roja";
String cad2 = "La Casa es Roja";
```

La expresión...	Tiene el valor..	Comentarios...
`cad1.equals(cad2)`	false	La expresión es falsa, porque la comparación se hace teniendo en cuenta las mayúsculas y las minúsculas.
`cad1.equalsIgnoreCase(cad2)`	true	Con este método de la clase String podemos comparar dos cadenas de caracteres, ignorando si son mayúsculas o minúsculas.
`cad1 + " y verde"`	"la casa es roja y verde"	Se debe prever un espacio en blanco entre las cadenas, si no queremos que queden pegadas.
`cad1.charAt(1)`	'a'	Los caracteres de la cadena se comienzan a numerar desde cero.
`cad2.charAt(2)`	' '	El espacio en blanco es el tercer carácter de la cadena. Debe quedar claro que no es lo mismo el carácter ' ' que la cadena de caracteres " ". El primero es un literal de tipo `char`, mientras que el segundo es un literal de la clase `String`.

> Si en una expresión aritmética no se usan paréntesis para definir el orden de evaluación, Java aplicará a los operadores un orden por defecto. Dicho orden está asociado con una prioridad que el lenguaje le asigna a cada operador.
> Básicamente, las reglas se pueden resumir de la siguiente manera:
>
> - Primero se aplican los operadores de multiplicación y división, de izquierda a derecha.
> - Después se aplican los operadores de suma y resta, de izquierda a derecha.

Supongamos que tenemos dos variables var1 y var2, con valores 10 y 5 respectivamente.

La expresión...	Tiene el valor...	Comentarios...
`var1 – var2 - 10`	-5	Aplica el operador de resta de izquierda a derecha.
`var1 – ( var2 – 10 )`	15	Los paréntesis le dan un orden de evaluación distinta a la expresión: 10 – (5 – 10) = 10 –(-5) = 10 + 5 = 15.
`var1 * var2 / 5`	10	En esta expresión se hace primero la multiplicación y luego la división: (10 * 5) / 5 = 50 / 5 = 10. Esto es así porque ambos operadores tienen la misma prioridad, de modo que se evalúan de izquierda a derecha.
`var1 * ( var2 / 10 )`	5	Los paréntesis le dan un orden de evaluación distinto a la expresión: 10 *(5 / 10)* = *10* 0.5 = 5.
`var1 - var2 + 10`	15	En esta expresión se hace primero la resta y después la suma (aplica los operadores suma y resta de izquierda a derecha, puesto que ambos tienen la misma prioridad).
`var1 + var2 * 10`	60	En esta expresión se hace primero la multiplicación, puesto que ese operador tiene más prioridad que la suma.
`var1 + var2 * 10 - 5`	55	En esta expresión se hace primero la multiplicación, luego la suma y, finalmente, la resta.
`var1 + var2 * 10 / 5`	20	En esta expresión se hace primero la multiplicación, luego la división y, finalmente, la suma. Debe ser clara, en este punto, la importancia de los paréntesis en las expresiones.

Llegó el momento de comenzar a trabajar en el caso de la tienda, así que de nuevo manos a la obra.

Tarea 2

Objetivo: Generar habilidad en la construcción e interpretación de expresiones, utilizando el caso de estudio de la tienda.

Utilizando las declaraciones hechas en la sección anterior para las clases Tienda y Producto y el escenario propuesto a continuación, resuelva los ejercicios que se plantean más adelante.

Escenario:

Suponga que en la tienda del caso de estudio se tienen a la venta los siguientes productos:

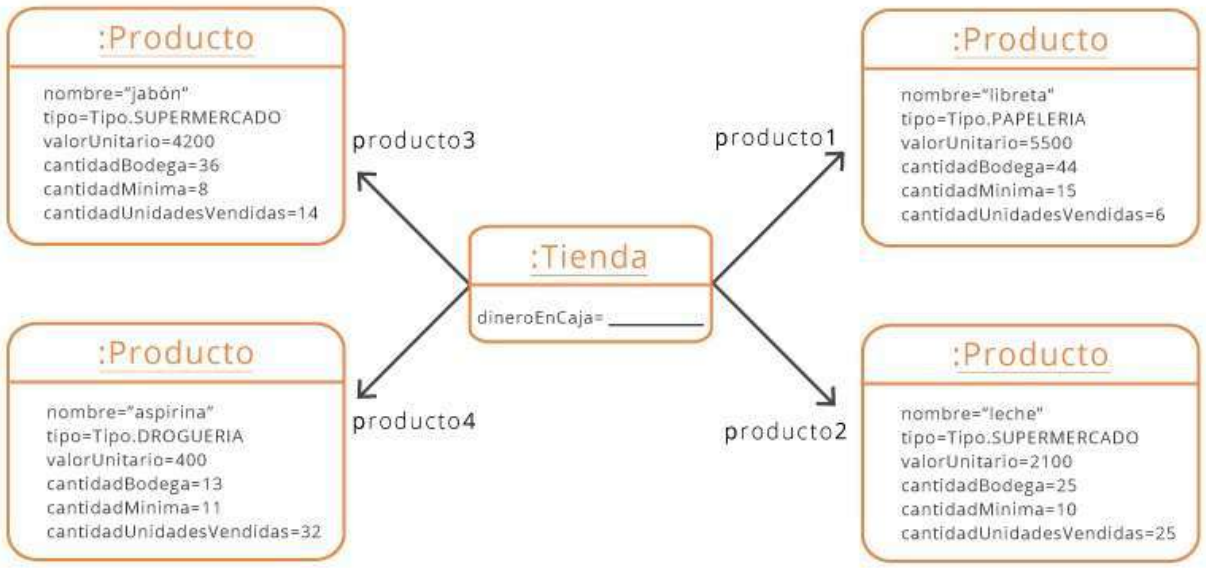
1. Libreta de apuntes, producto de papelería, a $5.500 pesos la unidad.
2. Leche en bolsa de 1 litro, producto de supermercado, a $2.100 pesos.
3. Jabón en polvo, producto de supermercado, a $4.200 el kilo.
4. Aspirina, producto de droguería, a $400 la unidad.

Suponga además, que ya se han vendido en la tienda 6 libretas, 25 bolsas de leche, 14 bolsas de jabón y 32 aspirinas.

Por último tenemos la siguiente tabla para resumir el inventario de unidades de la tienda y la cantidad por debajo de la cual se puede hacer un abastecimiento.

Producto	Cantidad en bodega	Cantidad mínima
libreta	44	15
leche	25	10
jabón	36	8
aspirina	13	11

En el siguiente diagrama de objetos puede ver el estado actual de la tienda. Complete la cantidad de dinero en caja que tiene la tienda, teniendo en cuenta las ventas que ya se realizaron.



Parte I – Evaluación de Expresiones (operadores aritméticos):

Para el objeto...	la expresión...	toma el valor...
leche	`cantidadBodega - cantidadMinima`	15
aspirina	`valorUnitario * cantidadBodega`	
jabón	`( cantidadUnidadesVendidas+ cantidadBodega ) * ( valorUnitario + valorUnitario * IVA_SUPERMERCADO )`	
libreta	`valorUnitario * cantidadBodega / cantidadUnidadesVendidas * valorUnitario`	
leche	`valorUnitario * cantidadUnidadesVendidas * IVA_SUPERMERCADO`	
aspirina	`valorUnitario * ( 1 + IVA_DROGUERIA ) * cantidadUnidadesVendidas`	
la tienda	`( producto1.darValorUnitario( ) + producto2.darValorUnitario( ) + producto3.darValorUnitario( ) + producto4.darValorUnitario( ) ) / 4`	3050.0
la tienda	`( producto1.darCantidadBodega( ) - producto1.darCantidadMinima( ) ) * ( producto1.darValorUnitario( ) * ( 1 + producto1.darIVA( ) ) )`	
la tienda	`dineroEnCaja - ( producto2.darCantidadMinima( ) * producto2.darValorUnitario( ) )`	
la tienda	`producto3.darCantidadUnidadesVendidas( ) * ( 1 + producto3.darIVA( ) )`	

Parte II – Evaluación de Expresiones (operadores relacionales):

Para el objeto...	la expresión...	toma el valor...
libreta	`tipo == Tipo.PAPELERIA`	true
libreta	`tipo != Tipo.DROGUERIA`	
leche	`cantidadMinima >= cantidadBodega`	
jabón	`valorUnitario <= 10000`	
aspirina	`cantidadBodega - cantidadMinima != cantidadUnidadesVendidas`	
jabón	`cantidadBodega * valorUnitario == cantidadUnidadesVendidas * IVA_PAPELERIA`	
la tienda	`producto1.darCantidadUnidadesVendidas( ) + producto2.darCantidadUnidadesVendidas( ) > producto3.darCantidadUnidadesVendidas( )`	true
la tienda	`dineroEnCaja <= producto4.darCantidadUnidadesVendidas( ) * ( ( 1 + producto4.darIVA( ) ) * producto4.darValorUnitario( ) )`	
la tienda	`( producto1.darCantidadBodega( ) + producto2.darCantidadBodega( ) + producto3.darCantidadBodega( ) + producto4.darCantidadBodega( ) ) <= 1000`	
la tienda	`dineroEnCaja * producto1.darIVA( ) > producto1.darCantidadUnidadesVendidas( ) * producto1.darValorUnitario( )`	

Parte III – Evaluación de Expresiones (operadores lógicos):

Para el objeto...	la expresión...	toma el valor...
leche	`!(tipo == Tipo.PAPELERIA || tipo == Tipo.DROGUERIA )`	true
jabón	`tipo == Tipo.SUPERMERCADO && valorUnitario <= 10000`	
aspirina	`cantidadBodega > cantidadMinima && cantidadBodega < cantidadUnidadesVendidas`	
libreta	`valorUnitario >= 1000 && valorUnitario <= 5000`	
leche	`tipo != Tipo.PAPELERIA && tipo != Tipo.SUPERMERCADO`	
aspirina	`tipo == Tipo.PAPELERIA && valorUnitario > 50 && !(cantidadMinima < cantidadBodega )`	
la tienda	`producto1.darTipo( ) == Tipo.PAPELERIA && producto2.darTipo( ) == Tipo.SUPERMERCADO && producto3.darTipo( ) != Tipo.DROGUERIA && producto4.darTipo( ) == Tipo.SUPERMERCADO`	false
la tienda	`( dineroEnCaja / producto1.darValorUnitario( ) ) >= producto1.darCantidadMinima( )`	
la tienda	`(( producto2.darCantidadBodega() + producto2.darCantidadBodega())/10 < 100 ) && (( producto2.darCantidadBodega()+producto2.darCantidadBodega())/10 >= 50 )`	
la tienda	`dineroEnCaja * 0.1 <= producto3.darValorUnitario( ) * ( 1 + producto3.darIVA( ) )`	

Parte IV – Creación de Expresiones (operadores aritméticos):

En un método de la clase...	para obtener..	se usa la expresión...
Producto	Valor de venta de un producto con IVA del 16%	`valorUnitario * ( 1 + IVA_PAPELERIA )`
Producto	Número de unidades que se deben vender para alcanzar la cantidad mínima.	
Producto	Número de veces que se ha vendido la cantidad mínima del producto.	
Producto	Número de unidades sobrantes si se arman paquetes de 10 con lo disponible en bodega.	
Tienda	Dinero en caja de la tienda incrementado en un 25%	`dineroEnCaja * 1.25`
Tienda	Total del IVA a pagar por las unidades vendidas de todos los productos.	
Tienda	El número de unidades del producto 3 que se pueden pagar (a su valor unitario) con el dinero en caja de la tienda.	
Tienda	El número de estantes de 50 posiciones que se requieren para almacenar las unidades en bodega de todos los productos (suponga que cada unidad de producto ocupa una posición).	

Parte V – Creación de Expresiones (operadores relacionales):

En un método de la clase...	para obtener..	se usa la expresión...
Producto	¿La cantidad en bodega es mayor o igual al doble de la cantidad mínima?	`cantidadBodega >= 2 * cantidadMinima`
Producto	¿El tipo no es PAPELERIA?	
Producto	¿El total de productos vendidos es igual a la cantidad en bodega?	
Producto	¿El nombre del producto comienza por el carácter 'a'?	
Tienda	¿El nombre del producto 2 es "aspirina"?	`producto2.darNombre().equals("aspirina")`
Tienda	¿La cantidad mínima del producto 4 es una quinta parte de la cantidad de productos vendidos?	
Tienda	¿El valor obtenido por los productos vendidos (incluyendo el IVA) es menor a un tercio del dinero en caja?	
Tienda	¿El promedio de unidades vendidas de todos los productos es mayor al promedio de unidades en bodega de todos los productos?	

Parte VI – Creación de Expresiones (operadores lógicos):

En un método de la clase...	para obtener..	se usa la expresión...

Producto	¿El tipo de producto es SUPERMERCADO y su valor unitario es menor a $3.000?	`tipo == Tipo.SUPERMERCADO && valorUnitario < 3000`
Producto	¿En la cantidad en bodega o en la cantidad de productos vendidos está al menos 2 veces la cantidad mínima?	
Producto	¿El tipo no es DROGUERIA y el valor está entre 1000 y 3500 incluyendo ambos valores?	
Producto	¿El tipo es PAPELERIA y la cantidad en bodega es mayor a 10 y el valor unitario es mayor o igual a $3.000?	
Tienda	¿El tipo del producto 1 no es ni DROGUERIA ni PAPELERIA y el total de unidades vendidas de todos los productos es menor a 30?	`producto1.darTipo( ) != Tipo.DROGUERIA && producto1.darTipo( ) != Tipo.PAPELERIA && ( producto1.darProductosVendidos( ) + producto2.darProductosVendidos( ) + producto3.darProductosVendidos( ) + producto4.darProductosVendidos( ) ) < 30`
Tienda	¿Con el valor en caja de la tienda se pueden pagar 500 unidades del producto 1 ó 300 unidades del producto 3 (al precio de su valor unitario)?	
Tienda	¿Del producto 4, el tope mínimo es mayor a 10 y la cantidad en bodega es menor o igual a 25?	

Tienda	¿El valor unitario de los productos 1 y 2 está entre 200 y 1000 sin incluir dichos valores?	

5.5. Manejo de Variables

El objetivo de las **variables** es permitir manejar cálculos parciales en el interior de un método. Las variables se deben declarar (darles un nombre y un tipo) antes de ser utilizadas y siguen la misma convención de nombres de los atributos. Las variables se crean en el momento en el que se declaran y se destruyen automáticamente al llegar al final del método que las contiene. Por esta razón es imposible utilizar el valor de una variable por fuera del método donde fue declarada.

Se suelen usar variables por tres razones principales:

1. Porque es necesario calcular valores intermedios.
2. Por eficiencia, para no pedir dos veces el mismo servicio al mismo objeto.
3. Por claridad en el código.

A continuación se muestra un ejemplo de un método de la clase Tienda que calcula la cantidad disponible del primer producto y luego vende esa misma cantidad de todos los demás.

```
public void venderDeTodo( )
{
    int cuanto = producto1.darCantidadBodega( );
    producto2.vender( cuanto );
    producto3.vender( cuanto );
    producto4.vender( cuanto );
}
```

- Se declara al comienzo del método una variable de tipo entero llamada "`cuanto`", y se le asigna la cantidad que hay en bodega del producto 1 de la tienda.
- La declaración de la variable y su inicialización se pueden hacer en instrucciones separadas (no hay necesidad de inicializar las variables en el momento de declararlas). La única condición que verifica el compilador es que antes de usar una variable ya haya sido inicializada.
- En este método se usa la variable "`cuanto`" por eficiencia y por claridad (no calculamos el mismo valor tres veces sino sólo una).

5.6 Otros Operadores de Asignación

El operador de asignación visto en el nivel anterior permite cambiar el valor de un atributo de un objeto, como una manera de reflejar un cambio en el mundo del problema. Vender 5 unidades de un producto, por ejemplo, se hace restando el valor 5 del atributo `cantidadBodega`.

En este nivel vamos a introducir cuatro nuevos operadores de asignación, con la aclaración de que sólo es una manera más corta de escribir las asignaciones, las cuales siempre se pueden escribir con el operador del nivel anterior.

- Operador `++`. Se aplica a un atributo entero, para incrementarlo en 1. Por ejemplo, para indicar que se agregó una unidad de un producto a la bodega (en la clase Producto), se puede utilizar cualquiera de las siguientes versiones del mismo método.

```
public void agregarNuevaUnidadBodega( )
{
    cantidadBodega++;
}
```

- El operador de incremento se puede ver como un operador de asignación en el cual se modifica el valor del operando sumándole el valor 1.
- El uso de este operador tiene la ventaja de generar expresiones un poco más compactas.

```
public void agregarNuevaUnidadBodega( )
{
    cantidadBodega = cantidadBodega + 1;
}
```

- Esta segunda versión del método tiene la misma funcionalidad, pero utiliza el operador de asignación normal.
- Operador `--`. Se aplica a un atributo entero, para disminuirlo en 1. Se utiliza de manera análoga al operador de incremento.
- Operador `+=`. Se utiliza para incrementar un atributo en cualquier valor. Por ejemplo, el método para hacer un pedido de una cierta cantidad de unidades para la bodega, puede escribirse de las dos maneras que se muestran a continuación. Debe quedar claro que la instrucción `var++` es equivalente a `var += 1`, y equivalente a su vez a `var = var + 1`.

```
public void pedir( int pNum )
{
    cantidadBodega += pNum;
}
```

- Este método de la clase Producto permite hacer un pedido de "pNum" unidades y agregarlas a la bodega.
- El operador `+=` se puede ver como una generalización del operador `++`, en el cual el incremento puede ser de cualquier valor y no sólo igual a 1.

```
public void pedir( int pNum )
{
    cantidadBodega = cantidadBodega + pNum;
}
```

- Esta segunda versión del método tiene la misma funcionalidad, pero utiliza el operador de asignación normal.
- La única ventaja de utilizar el operador `+=` es que se obtiene un código un poco más compacto. Usarlo o no usarlo es cuestión de estilo de cada programador.
- Operador `-=`. Se utiliza para disminuir un atributo en cualquier valor. Se utiliza de manera análoga al operador `+=`.

Tarea 3

Objetivo: Generar habilidad en la utilización de las asignaciones y las expresiones como un medio para transformar el estado de un objeto.

Para las declaraciones de las clases Tienda y Producto dadas anteriormente y, teniendo en cuenta el escenario planteado más adelante, escriba la instrucción o las instrucciones necesarias para modificar el estado, siguiendo la descripción que se hace en cada caso.

Escenario

Suponga que en la tienda del caso de estudio se tienen a la venta los siguientes productos:

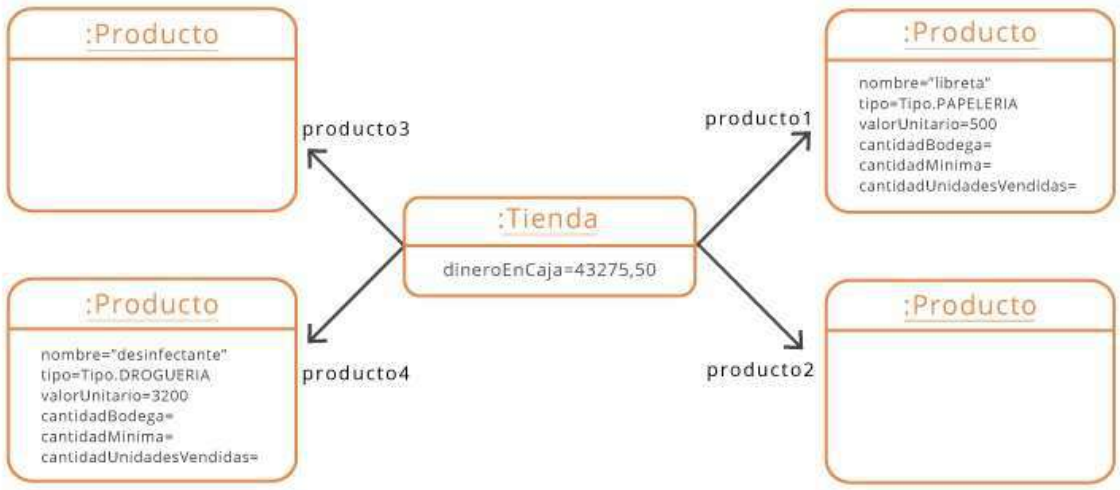
1. Lápiz, producto de papelería, con un valor base de $500 pesos la unidad.
2. Borrador, producto de papelería, a $300 pesos.
3. Kilo de café, producto de supermercado, a $5.600 la unidad.
4. Desinfectante, producto de droguería, a $3.200 la unidad.

Suponga además, que se han vendido 15 lápices, 5 borradores, 7 kilos de café y 12 frascos de desinfectante, y que en la caja de la tienda hay en este momento $43.275,50.

Por último tenemos la siguiente tabla para resumir el inventario de unidades de la tienda y el tope mínimo que se debe alcanzar para poder hacer un nuevo pedido:

Producto	Cantidad en bodega	Tope mínimo
lapiz	30	9
borrador	15	5
café	20	10
desinfectante	12	11

Complete el diagrama de objetos que aparece a continuación, con la información del escenario:



Signatura de los métodos de la clase Tienda:

```
//---------------------------------
// Signaturas de métodos
//---------------------------------
public Producto darProducto1( )
public Producto darProducto2( )
public Producto darProducto3( )
public Producto darProducto4( )
public double darDineroEnCaja( )
```

Signatura de los métodos de la clase Producto:

```
//---------------------------------
// Signaturas de métodos
//---------------------------------
public String darNombre( )
public int darTipo( )
public double darValorUnitario( )
public int darCantidadBodega( )
public int darCantidadMinima( )
public int darCantidadUnidadesVendidas( )
public double darIVA( )
public int vender( int pCantidad )
public void abastecer( int pCantidad  )
```

En un método de la clase...	la siguiente modificación de estado..	se logra con las siguientes instruciones...
Producto	Se vendieron 5 unidades del producto (suponga que hay suficientes).	`cantidadUnidadesVendidas += 5; cantidadBodega -= 5;`
Producto	El valor unitario se incrementa en un 10%	
Producto	Se incrementa en uno la cantidad mínima para hacer pedidos.	
Producto	El producto ahora se clasifica como de SUPERMERCADO	
Producto	Se cambia el nombre del producto. Ahora se llama "teléfono".	

En un método de la clase...	la siguiente modificación de estado..	se logra con las siguientes instruciones...
Tienda	Se asigna al dinero en caja de la tienda la suma de los valores unitarios de los cuatro productos.	`dineroEnCaja = producto1.darValorUnitario( ) + producto2.darValorUnitario( ) + producto3.darValorUnitario( ) + producto4.darValorUnitario( );`

Tienda	Se venden 4 unidades del producto 3 (suponga que están disponibles).	
Tienda	Se disminuye en un 2% el dinero en la caja.	
Tienda	Se abastece la tienda con la mitad de la cantidad mínima de cada producto, suponiendo que la cantidad en bodega de todos los productos es menor a la cantidad mínima.	
Tienda	Se pone en la caja el dinero correspondiente a las unidades vendidas de todos los productos de la tienda.	
Una clase de la interfaz de usuario	Se vende una unidad de cada uno de los productos de la tienda. Recuerde que este método está por fuera de la clase Tienda, y que por lo tanto no puede utilizar sus atributos de manera directa.	

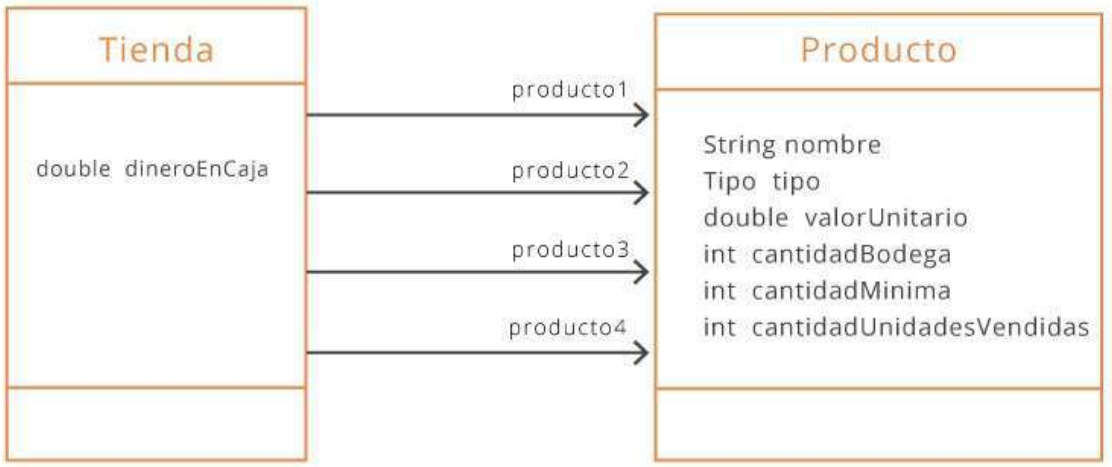
> Antes de comenzar a escribir el cuerpo de un método, es importante tener en cuenta la clase en la cual éste se encuentra. No olvide que dependiendo de la clase en la que uno se encuentre, las cosas se deben decir de una manera diferente. En unos casos los atributos se pueden manipular directamente y, en otros, es indispensable llamar un método para cambiar el estado (para que la modificación la realice el objeto al que pertenece el atributo).

6. Clases y Objetos

6.1. Diferencia entre Clases y Objetos

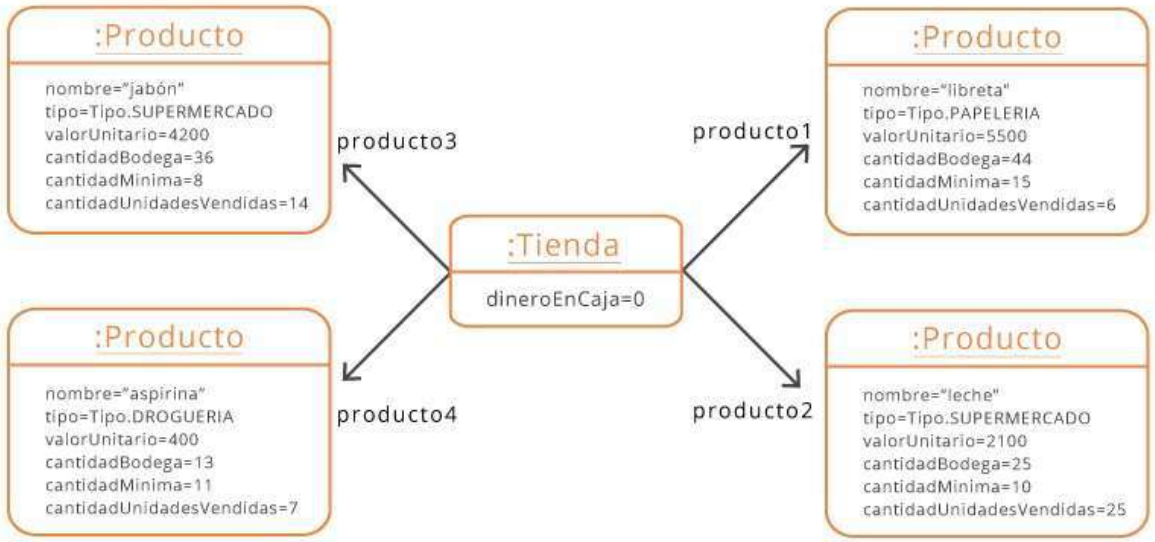
Aunque los conceptos de clase y objetos son muy diferentes, el hecho de usarlos indistintamente en algunos contextos hace que se pueda generar alguna confusión al respecto. En la figura 2.4a y figura 2.4b se muestra, para el caso de la tienda, el correspondiente diagrama de clases y un ejemplo de un posible diagrama de objetos. Allí se puede apreciar que la clase Tienda describe todas las tiendas imaginables que vendan 4 productos.

Fig. 2.4a Modelo de clases



- Diagrama de clases para el caso de estudio de la tienda.
- El diagrama sólo dice, por ejemplo, que producto1 debe ser un producto.

Fig. 2.4b Modelo de objetos



- Fíjese como cada asociación del diagrama de clases debe tener su propio objeto en el momento de la ejecución.

Una clase no habla de un escenario particular, sino del caso general. Nunca dice cuál es el valor de un atributo, sino que se contenta con afirmar cuáles son los atributos (nombre y tipo) que deben tener los objetos que son instancias de esa clase. Los objetos, por su parte, siempre pertenecen a una clase, en el sentido de que cumplen con la estructura de atributos que la clase exige. Por ejemplo, puede haber miles de tiendas diferentes, cada una de las cuales vende distintos productos a distintos precios. Piense que cada vez que instalamos el programa del caso de estudio en una tienda distinta, el dueño va a querer que los objetos que se creen para representarla reflejen el estado de su propia tienda.

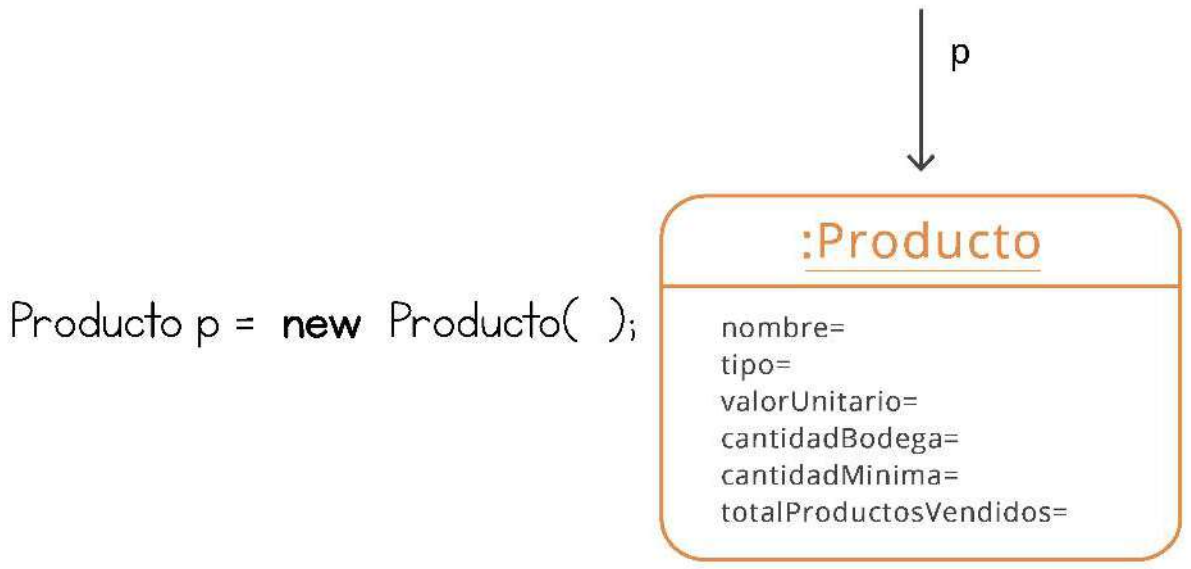
Los métodos de una clase, por su parte, siempre están en ella y no copiados en cada uno de sus objetos. Por esta razón cada objeto debe saber a qué clase pertenece, para poder buscar en ella los métodos que puede ejecutar. Los métodos están escritos de manera que se puedan utilizar desde todos los objetos de la clase. Cuando un método de la clase Tienda dice `producto1.darNombre()`, le está pidiendo a una tienda particular que busque en su propio escenario el objeto al cual se llega a través de la asociación producto1, y le pida a éste su nombre usando el método que todos los productos tienen para hacerlo. En este

sentido se puede decir que los métodos son capaces de resolver los problemas en abstracto, y que cada objeto los aplica a su propio escenario para resolver su problema concreto.

6.2. Creación de Objetos de una Clase

Recordemos la creación de objetos visto en el nivel anterior. Un objeto se crea utilizando la instrucción `new` y dando el nombre de la clase de la cual va a ser una instancia. Todas las clases tienen un método constructor por defecto, sin necesidad de que el programador tenga que crearlo. Como no es responsabilidad del computador darle un valor inicial a los atributos, cuando se usa este constructor, éstos quedan en un valor que se puede considerar indefinido. En la figura 2.5 se muestra el resultado del llamado a este constructor.

Fig. 2.5 Creación de un objeto usando la instrucción new



- El resultado de ejecutar la instrucción del ejemplo es un nuevo objeto, con sus atributos no inicializados.
- Dicho objeto está "referenciado" por p, que puede ser un atributo o una variable de tipo Producto.

Para inicializar los valores de un objeto, se debe definir en la clase un constructor propio. En el siguiente ejemplo trabajaremos los conceptos vistos en el capítulo anterior, usando el caso de la tienda.

Ejemplo 10

Se hace la inicialización de los atributos de los objetos de la clase.

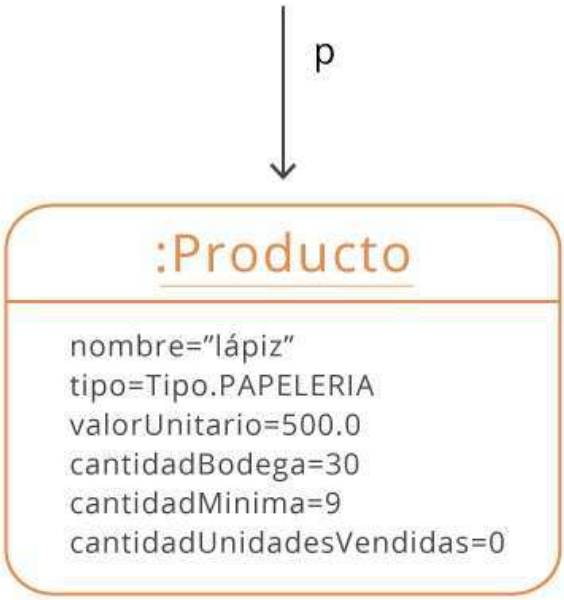
En este ejemplo mostramos los constructores de las clases Tienda y Producto, así como la manera de pedir la creación de un objeto de cualquiera de esos dos tipos.

```
public Producto(Tipo pTipo, String pNombre, double pValorUnitario, int pCantidadBodega
, int pCantidadMinima)
{
    tipo = pTipo;
    nombre = pNombre;
    valorUnitario = pValorUnitario;
    cantidadBodega = pCantidadBodega;
    cantidadMinima = pCantidadMinima;
    cantidadUnidadesVendidas = 0;
}
```

- El constructor exige 5 parámetros para poder inicializar los objetos de la clase Producto.
- En el constructor se asignan los valores de los parámetros a los atributos.

```
Producto p=new Producto(Tipo.PAPELERIA, "lápiz", 500.0, 30, 9);
```

- Este es un ejemplo de la manera de crear un objeto cuando el constructor tiene parámetros.



- Este es el objeto que se crea con la llamada anterior.
- El objeto creado se ubica en alguna parte de la memoria del computador. Dicho objeto es referenciado por el atributo o la variable llamada " `p` ".

```
public Tienda(  )
{
    producto1 = new Producto( Tipo.PAPELERIA, "Lapiz", 550.0, 18, 5 );
    producto2 = new Producto( Tipo.DROGUERIA, "Aspirina", 109.5, 25, 8 );
    producto3 = new Producto( Tipo.PAPELERIA, "Borrador", 207.3, 30, 10 );
    producto4 = new Producto( Tipo.SUPERMERCADO, "Pan", 150.0, 15, 20 );
    dineroEnCaja = 0;
}
```

- Puesto que es necesario que la tienda tenga 4 productos, su método constructor debe ser como el que se presenta. Supone que en la caja de la tienda no hay dinero al comenzar el programa.

Vamos a practicar la creación de escenarios usando los métodos constructores de las clases Tienda y Producto. En el programa del caso de estudio, la responsabilidad de crear el estado de la tienda sobre la cual se trabaja está en la clase principal del mundo (clase Tienda). En una situación real, dichos valores deberían leerse de un archivo o de una base

de datos, pero en nuestro caso se utilizará un escenario predefinido. Si quiere modificar los datos de la tienda sobre los que trabaja el programa, puede darle otros valores en el momento de construir las instancias.

Tarea 4

Objetivo: Generar habilidad en el uso de los constructores de las clases para crear escenarios

Cree los escenarios que se describen a continuación, dando la secuencia de instrucciones que los construyen. Suponga que dicha construcción se hace desde una clase externa a las clases Tienda y Producto.

Escenario 1

Una nueva tienda acaba de abrir y quiere usar el programa del caso de estudio con los siguientes productos:

1. Frasco de jarabe (para la gripe), producto de droguería, con un valor base de $7.200 pesos.
2. Botella de alcohol, producto de droguería, a $2.800 pesos la unidad.
3. Kilo de queso, producto de supermercado, a $4.100 la unidad.
4. Resaltador, producto de papelería, a $3.500 la unidad.

La siguiente tabla resume el inventario inicial de la tienda y el tope mínimo que se debe alcanzar para poder hacer un nuevo pedido. Suponga que el valor inicial en caja es cero pesos.

Producto	Cantidad en bodega	Tope mínimo
jarabe	14	10
alcohol	12	8
queso	10	4
resaltador	20	10

Código

```
public Tienda( )
{
    producto1 = new Producto( Tipo.DROGUERIA, "jarabe", 7200.0, 14, 10 );
    producto2 = new Producto( Tipo.DROGUERIA, "alcohol", 2800.0, 12, 8 );
    producto3 = new Producto( Tipo.SUPERMERCADO, "queso", 4100.0, 10, 4 );
    producto4 = new Producto( Tipo.PAPELERIA, "resaltador", 3500.0, 20, 10 );
}
```

Escenario 2

Una nueva tienda acaba de abrir y quiere usar el programa del caso de estudio con los siguientes productos:

1. Kilo de arroz, producto de supermercado, con valor base de $1.200 pesos.
2. Caja de cereal, producto de supermercado, a $7.500 pesos.
3. Resma de papel, producto de papelería, a $20.000 pesos la unidad.
4. Bolsa de algodón, producto de droguería, a $4.800 pesos.

La siguiente tabla resume el inventario inicial de la tienda y el tope mínimo que se debe alcanzar para poder hacer un nuevo pedido. Suponga que el valor inicial en caja es cero pesos.

Producto	Cantidad en bodega	Tope mínimo
arroz	6	7
cereal	5	5
papel	50	2
algodón	12	6

Código

Escenario 3

Una nueva tienda acaba de abrir, y quiere usar el programa del caso de estudio con los siguientes productos:

1. Litro de aceite, producto de supermercado, con un valor base de $6.500 pesos la unidad.
2. Crema dental, producto de supermercado, a $5.100 pesos.
3. Kilo de pollo, producto de supermercado, a $13.800 pesos la unidad.
4. Protector solar, producto de droguería, a $16.000 la unidad.

La siguiente tabla resume el inventario inicial de la tienda y el tope mínimo que se debe alcanzar para poder hacer un nuevo pedido. Suponga que el valor inicial en caja es cero pesos.

Producto	Cantidad disponible	Tope mínimo
aceite	13	10
crema dental	20	15
pollo	6	5
protector solar	3	3

Código

7. Instrucciones Condicionales

7.1. Instrucciones Condicionales Simples

Una instrucción condicional nos permite plantear la solución a un problema considerando los distintos casos que se pueden presentar. De esta manera, podemos utilizar un algoritmo distinto para enfrentar cada caso que pueda existir en el mundo. Considere el método de la clase Producto que se encarga de vender una cierta cantidad de unidades presentes en la bodega. Allí, se pueden presentar dos casos posibles, cada uno con una solución distinta: el primer caso es cuando la cantidad que se quiere vender es mayor que la cantidad disponible en la bodega (el pedido es mayor que la disponibilidad) y el segundo es cuando hay suficientes unidades del producto en la bodega para hacer la venta. En cada una de esas situaciones la solución es distinta y el método debe tener un algoritmo diferente.

Para construir una instrucción condicional, se deben identificar los casos y las soluciones, usando algo parecido a la tabla que se muestra a continuación:

Caso 1:

Expresión que describe el caso:

```
cantidad > cantidadBodega
```

Algoritmo para resolver el problema en ese caso:

```
// Vende todas las unidades disponibles
cantidadUnidadesVendidas += cantidadBodega;
cantidadBodega = 0;
```

Caso 2:

Expresión que describe el caso:

```
cantidad <= cantidadBodega
```

Algoritmo para resolver el problema en ese caso:

```
// Vende lo pedido por el usuario
cantidadUnidadesVendidas += cantidad;
cantidadBodega  -=  cantidad;
```

En el primer caso la solución es vender todo lo que hay en la bodega. En el segundo, vender lo pedido como parámetro. En Java existe la instrucción condicional `if-else`, que permite expresar los casos dentro de un método. La sintaxis en Java de dicha instrucción se ilustra en el siguiente fragmento de programa:

```
public class Producto
{
    ...
    public void vender( int pCantidad )
    {
        if( pCantidad > cantidadBodega )
        {
            totalProductosVendidos += cantidadBodega;
            cantidadBodega  =  0;
        }
        else
        {
            totalProductosVendidos += pCantidad ;
            cantidadBodega -= pCantidad ;
        }
    }
}
```

- En lugar de una sola secuencia de instrucciones, se puede dar una secuencia para cada caso posible. El computador sólo va a ejecutar una de las dos secuencias.
- Es como si el método escogiera el algoritmo que debe utilizar para resolver el problema puntual que tiene, identificando la situación en la que se encuentra el objeto.
- La condición caracteriza los dos casos que se pueden presentar. Note que con una sola condición debemos separar los dos casos.
- Los paréntesis alrededor de la condición son obligatorios.
- La condición es una expresión lógica, construida con operadores relacionales y lógicos.
- La parte del " `else` ", incluida la segunda secuencia de instrucciones, es opcional. Si no se incluye, eso querría decir que para resolver el segundo caso no hay que hacer nada.

La instrucción `if-else` tiene tres elementos:

1. Una condición que corresponde a una expresión lógica capaz de distinguir los dos casos (su evaluación debe dar verdadero si se trata del primer caso y falso si se trata del segundo).
2. La solución para el primer caso.
3. La solución para el segundo caso. Al encontrar una instrucción condicional, el computador evalúa primero la condición y decide a partir de su resultado cuál de las dos soluciones ejecutar. Nunca ejecuta las dos.

Si el algoritmo que resuelve uno de los casos sólo tiene una instrucción, es posible eliminar los corchetes, como se ilustra en el ejemplo 11. Allí también se puede apreciar que una instrucción condicional es sólo una instrucción más dentro de la secuencia de instrucciones que implementan un método.

Ejemplo 11

En este ejemplo se presentan algunos métodos de la clase Producto, para mostrar la sintaxis de la instrucción `if-else` de Java. Los métodos aquí presentados no son necesariamente los que escribiríamos para implementar los requerimientos funcionales del caso de estudio, pero sirven para ilustrar distintos aspectos de las instrucciones condicionales.

```
public boolean haySuficiente( int pCantidad )
{
    boolean suficiente;
    if( pCantidad <= cantidadBodega )
        suficiente = true;
    else
        suficiente = false;
    return suficiente;
}
```

- Como la secuencia de instrucciones de cada caso tiene una sola instrucción, se pueden eliminar los corchetes.
- Fíjese que dejamos el resultado de cada caso en la misma variable, de manera que al hacer el retorno del método siempre se encuentre allí el resultado. ¿Qué hace este método?
- Una instrucción condicional se puede ver como otra instrucción más del método. Puede haber instrucciones antes y después de ella.

```
public boolean haySuficiente( int pCantidad )
{
      return pCantidad <= cantidadBodega;
}
```

- El método anterior también se podría escribir de esta manera, un poco más sencilla. ¿Qué ganamos escribiéndolo así?

```
public double darPrecioPapeleria( boolean conIVA )
{
    double precioFinal = valorUnitario;

    if( conIVA )
        precioFinal = precioFinal * ( 1+IVA_PAPELERIA );

    return precioFinal;
}
```

- Si en el segundo de los casos de una instrucción condicional no es necesario hacer nada, no se debe escribir ninguna instrucción, tal como se muestra en el ejemplo.
- ¿Está claro el problema que resuelve el método?

```
public void ajustarPrecio( )
{
    if( totalProductosVendidos < 100 )
    {
        valorUnitario = valorUnitario * 80 / 100;
    }
    else
    {
        valorUnitario = valorUnitario * 1.1;
    }
}
```

- En este método, si se han vendido menos de 100 unidades, se hace un descuento del 20% en el precio del producto.
- Si se han vendido 100 o más unidades, se aumenta en un 10% el precio.
- En las instrucciones condicionales, incluso si sólo hay una instrucción para resolver cada caso, es buena idea utilizar los corchetes para facilitar la lectura del código, es buena idea utilizar los corchetes. En algunos casos, incluso, son indispensables para evitar ambigüedades.

> Tenga cuidado de no escribir un ";" después de la condición, porque el computador lo va a interpretar como si la solución al caso fuera no hacer nada (una instrucción vacía).

7.2 Condicionales en Cascada

Cuando el problema tiene más de dos casos, es necesario utilizar una cascada (secuencia) de instrucciones `if-else`, en donde cada condición debe indicar sin ambigüedad la situación que se quiere considerar. Suponga por ejemplo que queremos calcular el IVA de

un producto. Puesto que el valor que se paga de impuestos por un producto depende de su tipo, es necesario considerar los tres casos siguientes:

Caso 1

Expresión que describe el caso:

```
( tipo == Tipo.SUPERMERCADO )
```

Algoritmo para resolver el problema en ese caso:

```
return IVA_SUPERMERCADO;
```

Caso 2

Expresión que describe el caso:

```
( tipo == Tipo.DROGUERIA )
```

Algoritmo para resolver el problema en ese caso:

```
return IVA_DROGUERIA;
```

Caso 3

Expresión que describe el caso:

```
( tipo == Tipo.PAPELERIA )
```

Algoritmo para resolver el problema en ese caso:

```
return IVA_PAPELERIA;
```

El método de la clase Producto para determinar el IVA que hay que pagar sería de la siguiente forma (no es la única solución, como veremos más adelante):

```
public double darIVA( )
{
    if( tipo == Tipo.PAPELERIA )
    {
        return IVA_PAPELERIA;
    }
    else if( tipo == Tipo.SUPERMERCADO )
    {
        return IVA_SUPERMERCADO;
    }
    else
    {
        return IVA_DROGUERIA;
    }
}
```

- Para representar los tres casos posibles, utilizamos una instrucción condicional en el "else" del primer caso. Esa manera de encadenar las instrucciones condicionales para poder considerar cualquier número de casos se denomina "en cascada".
- Una instrucción condicional puede ir en cualquier parte donde pueda ir una instrucción del lenguaje Java. Esto lo retomaremos en capítulos posteriores.

```
public double darIVA( )
{
    double resp = 0.0;

    if( tipo == Tipo.PAPELERIA )
    {
        resp = IVA_PAPELERIA;
    }
    else if( tipo == Tipo.SUPERMERCADO )
    {
        resp = IVA_SUPERMERCADO;
    }
    else
    {
        resp = IVA_DROGUERIA;
    }

    return resp;
}
```

En esta segunda solución del método, en lugar de hacer un retorno en cada caso, guardamos la respuesta en una variable y luego la retornamos al final.

> Al usar varias instrucciones if en cascada hay que tener cuidado con la ambigüedad que puede surgir con la parte else. Es mejor usar siempre corchetes para asegurarse de que el computador lo va a interpretar de la manera adecuada.

Tarea 5

Objetivo: Practicar el uso de las instrucciones condicionales simples para expresar el cambio de estado que debe hacerse en un objeto, en cada uno de los casos identificados.

Escriba el código de cada uno de los métodos descritos a continuación. Tenga en cuenta la clase en la cual está el método y la información que se entrega como parámetro.

Para la clase: Producto

Aumentar el valor unitario del producto, utilizando la siguiente política: si el producto cuesta menos de $1000, aumentar el 1%. Si cuesta entre $1000 y $5000, aumentar el 2%.
Si cuesta más de $5000 aumentar el 3%.

```
public void subirValorUnitario( )
{









}
```

Recibir un pedido, sólo si en bodega se tienen menos unidades de las indicadas en el tope mínimo. En caso contrario el método no debe hacer nada.

```
public void hacerPedido( int pCantidad )
{









}
```

Modificar el precio del producto, utilizando la siguiente política: si el producto es de droguería
o papelería debe disminuir un 10%. Si es de supermercado debe aumentar un 5%.

```
public void cambiarValorUnitario( )
{









}
```

Para la clase: Tienda

Vender una cierta cantidad del producto cuyo nombre es igual al recibido como parámetro. El método retorna el número de unidades efectivamente vendidas. Suponga que el nombre que se recibe como parámetro corresponde a uno de los productos de la tienda. Utilice el método vender de la clase Producto como parte de su solución.

```
public int venderProducto( String pNombreProducto, int pCantidad )
{









}
```

Calcular el número de productos de papelería que se venden en la tienda.

```
public int cuantosPapeleria( )
{









}
```

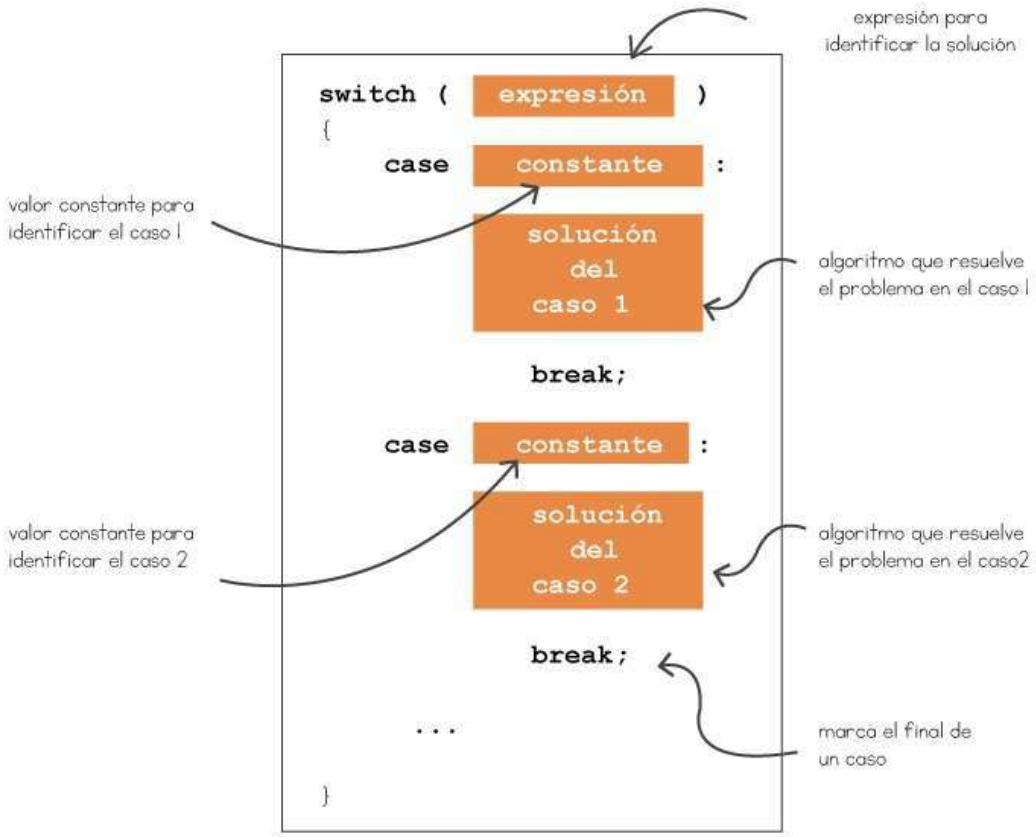
7.3. Instrucciones Condicionales Compuestas

Una instrucción condicional compuesta (`switch`) es una manera alternativa de expresar la solución de un problema para el cual existe un conjunto de casos, cada uno con un algoritmo distinto para resolverlo. Esta instrucción tiene la restricción de que cada caso debe estar identificado con un valor de tipo entero, String o del tipo de una enumeración.

En la instrucción `switch` va inicialmente la expresión (de tipo entero, String o del tipo de una enumeración) que se desea evaluar para identificar el caso que se está presentando. Esta expresión debe ir entre paréntesis. Después de dicha expresión, se escriben los bloques de instrucciones para cada uno de los casos identificados. Cada bloque empieza con la instrucción `case` seguida del valor de la constante que identifica el caso. Después se ponen las instrucciones del caso y al final de estas se debe colocar la instrucción `break` para indicar el fin del conjunto de instrucciones. En la figura 2.6 se ilustra la estructura de una instrucción condicional compuesta.

En el ejemplo 12 se presenta la solución del método que calcula el IVA de un producto, usando una instucción condicional compuesta.

Fig. 2.6 Estructura de la instrucción switch de Java



Ejemplo 12

Objetivo: Ilustrar el uso de instrucciones condicionales compuestas.

En este ejemplo se presenta el método que calcula el IVA que debe pagar un producto, dependiendo de su tipo.

```
public double darIVA( )
{
    double iva = 0.0;

    switch( tipo )
    {
        case PAPELERIA:
        {
            iva = IVA_PAPELERIA;
            break;
        }
        case SUPERMERCADO:
        {
            iva = IVA_SUPERMERCADO;
            break;
        }
        case DROGUERIA:
        {
            iva = IVA_DROGUERIA;
            break;
        }
    }

    return iva;
}
```

- Este método de la clase Producto tiene tres casos posibles, cada uno identificado con un valor de una enumeración (candidato ideal para la instrucción switch).
- La expresión que va a permitir distinguir los casos se construye simplemente con el atributo "tipo".
- Cada caso se introduce con la palabra reservada de Java "case" y se cierra con un "break". Después de la instrucción "case" va el valor que identifica el caso. En nuestro ejemplo, el valor se identifica con las constantes que representan los tres tipos posibles de productos: PAPELERIA, SUPERMERCADO o DROGUERIA.

Dado que siempre es posible escribir una instrucción condicional compuesta como una cascada de condicionales simples, la pregunta que nos debemos hacer es ¿cuándo usar una instrucción condicional compuesta? La respuesta es que siempre que se pueda utilizar la instrucción `switch` en lugar de una cascada de `if` es conveniente hacerlo, por dos razones:

1. Eficiencia, ya que de este modo sólo se evalúa una vez la expresión aritmética, mientras que en la cascada se evalúan una a una las condiciones, descartándolas.
2. Claridad en el programa, porque es más fácil de leer y mantener un programa escrito de esta manera.

Y la segunda pregunta es, ¿cuándo no intentar usar una instrucción condicional compuesta? La respuesta es: cuando los casos no están identificados por valores enteros, Strings o enumeraciones. Si se tienen, por ejemplo, un valor real como identificadores de los casos, la única opción es usar una cascada de instrucciones `if-else`.

Tarea 6

Objetivo: Utilizar instrucciones condicionales para expresar un conjunto de casos y soluciones
asociadas con los mismos.

Escriba el código de cada uno de los métodos descritos a continuación. Tenga en cuenta la clase en la cual está el método y la información que se entrega como parámetro.

Para la clase: Producto

Dar el nombre del tipo del producto. Por ejemplo, si el producto es de tipo SUPERMERCADO, el método debe retornar la cadena: "El producto es de supermercado".

```
public String nombreTipoProducto( )
{









}
```

Aumentar el precio del producto, siguiendo esta regla: si es un producto de droguería debe aumentar el 1%, si es de supermercado el 3% y si es de papelería el 2%.

```
public void aumentarValorUnitario( )
{









}
```

Para la clase: Tienda

Retornar el precio final del producto identificado con el número que se entrega como parámetro. Por ejemplo, si pNumeroProducto es 3, debe retornar el precio del tercer producto (`producto3`). Suponga que el valor que se entrega como parámetro es mayor o igual a 1 y menor o igual a 4.

```
public double darPrecioProducto( int pNumeroProducto )
{









}
```

Este método debe hacer lo mismo que el anterior, pero en lugar de recibir como parámetro el número del producto, recibe su nombre. Puede suponer que el nombre que se entrega como parámetro corresponde a un producto perteneciente a la tienda.

```
public double darPrecioProducto( String pNombreProducto )
{









}
```

8. Responsabilidades de una Clase

8.1. Tipos de Método

Los métodos en una clase se clasifican en tres tipos, según la operación que realicen:

- **Métodos constructores:** tienen la responsabilidad de inicializar los valores de los atributos de un objeto durante su proceso de creación.
- **Métodos modificadores:** tienen la responsabilidad de cambiar el estado de los objetos de la clase. Son los responsables de "hacer".
- **Métodos analizadores:** tienen la responsabilidad de calcular información a partir del estado de los objetos de la clase. Son los responsables de "saber".

8.2. ¿Cómo Identificar las Responsabilidades?

En esta parte sólo veremos algunas guías intuitivas respecto de cómo identificar las responsabilidades de una clase. Utilizamos dos estrategias complementarias que se pueden utilizar en cualquier orden y que se resumen a continuación:

- Una clase es responsable de administrar la información que hay en sus atributos. Por esta razón se debe tratar de buscar el conjunto de servicios que reflejen las operaciones típicas del elemento del mundo que la clase representa.

- Una clase es responsable de ayudar a sus vecinos del modelo del mundo y colaborar con ellos en la solución de sus problemas. En este caso la pregunta que nos debemos hacer es, ¿qué servicios necesitan las demás clases que les preste la clase que estamos diseñando? A partir de la respuesta a esta pregunta, iremos agregando servicios hasta que el problema global tenga solución.

Para las dos estrategias es conveniente hacer el recorrido por tipo de método, diseñando primero los constructores, luego los modificadores y, finalmente, los analizadores. En el nivel 4 de este libro retomaremos este problema de asignar responsabilidades a las clases.

Una vez que se han definido los servicios que va a prestar una clase, debemos definir los parámetros y el tipo de retorno. Para definir los parámetros de un método, debemos preguntarnos cuál es la información externa a la clase que se necesita para poder prestar el servicio. Para definir el tipo de retorno debemos preguntarnos qué información está esperando aquél que solicitó el servicio.

Tarea 7

Objetivo: Identificar y describir los métodos que representan las principales responsabilidades de una clase.

Para el caso de estudio que se presenta a continuación construya el diagrama de clases e identifique los principales métodos.

Una empresa de transporte tiene 3 camiones para llevar carga de una ciudad a otra del país. De cada camión se tiene su matrícula (6 caracteres), su capacidad (en kilogramos) y el consumo de gasolina por kilómetro (un valor real en litros/kilómetro) y la carga actual (en kilogramos). Se quiere construir un programa que permita optimizar el uso de los camiones. Para esto debe tener una única opción que determina cuál es el mejor camión para transportar una cierta carga entre dos ciudades. El mejor camión es aquél que, siendo capaz de transportar la carga, consume la mínima cantidad de gasolina.

Requerimiento funcional

Nombre	
Resumen	
Entradas	
Resultado	

Diagrama de clases:

Clase EmpresaTransporte

Nombre del método:

Tipo de método	
Responsabilidad	
Parámetros	
Retorno	

Nombre del método:

Tipo de método	
Responsabilidad	
Parámetros	
Retorno	

Nombre del método:

Tipo de método	
Responsabilidad	
Parámetros	
Retorno	

Nombre del método:

Tipo de método	
Responsabilidad	
Parámetros	
Retorno	

Clase Camion

Nombre del método:

Tipo de método	
Responsabilidad	
Parámetros	
Retorno	

Nombre del método:

Tipo de método	
Responsabilidad	
Parámetros	
Retorno	

Nombre del método:

Tipo de método	
Responsabilidad	
Parámetros	
Retorno	

Nombre del método:

Tipo de método	
Responsabilidad	
Parámetros	
Retorno	

9. Eclipse: Nuevas Opciones

En esta sección se cubren los siguientes temas:

- Uso de Eclipse para formatear una clase (concepto de *profile*). Se presentan las ventajas de mantener un correcto formato en los programas.
- Uso de Eclipse para localizar una declaración.
- Uso de Eclipse para localizar todos los clientes de un método (aquellos que lo usan).
- Uso de Eclipse para cambiar el nombre de un atributo, variable o método. Ventajas de hacerlo de esta manera y riesgo si hay errores de compilación.
- La siguiente tarea le propondrá una secuencia de acciones, que pretenden mostrarle la manera de hacer lo anteriormente mencionado en el ambiente de desarrollo Eclipse.

Tarea 8

Objetivo: Trabajar en Eclipse sobre la solución del caso de estudio, mostrando las nuevas opciones del ambiente de desarrollo que se introducen en este nivel.

Localice en el sitio del proyecto Cupi2 la solución del caso de estudio de este nivel. Copie en un directorio de trabajo dicha solución. Ejecute Eclipse y cree un nuevo proyecto que la contenga. Siga los pasos que se dan a continuación:

Paso I: ejecutar la aplicación

1. La clase InterfazTienda es la clase principal del programa. Localícela y selecciónela en el explorador de paquetes. Si tiene dificultades en esto, consulte la manera de hacerlo en el capítulo anterior.
2. Para ejecutar la clase principal de un programa, seleccione el comando *Run as Java Application*. Puede hacerlo desde la barra de herramientas, el menú principal o el menú emergente que aparece al hacer clic derecho sobre la clase.

Paso II: dar formato al código fuente

1. Localice el *profile* (perfil) de formato en el sitio del proyecto Cupi2. Puede encontrarlo bajo el título "*Perfil Cupi2 para Eclipse*". El *profile* reúne un conjunto de preferencias de formato en el código fuente, tales como indentación, posición de los corchetes, manejo de las líneas en blanco, comentarios, etc.
2. Instale el *profile* en Eclipse. Para esto seleccione la opción *Window/Preferences* del menú principal. En la ventana que aparece localice la zona *Java/Code Style/Code*

Formatter. Utilice el botón *Import*... para cargar el archivo mencionado en el punto anterior.

3. El hecho de cargar un *profile* no cambia automáticamente el formato de las clases. Seleccione y abra la clase Producto en el explorador de paquetes. Cambie el formato de los métodos. Elimine algunos espacios en las expresiones o cambie la indentación de las instrucciones.
4. Ahora aplíquele formato a la clase Producto seleccionando la opción *Source/Format* en el menú emergente del clic derecho o con *ctrl+mayús+F*. El formato ayuda a organizar el código fuente, mejorando su legibilidad y consistencia. Cuando se aplica formato a una clase, Eclipse utiliza la información que aparece en el *profile* que esté activo. Note cómo el programa recupera su estado inicial.
5. Para darle formato a una sola sección de la clase, seleccione la sección y aplíquele el formato como en el paso anterior. Adquiera la buena práctica de aplicar el formato a todos sus proyectos antes de entregarlos.

Paso III: localizar rápidamente el código fuente de una clase o método

1. Seleccione y abra la clase Tienda en el explorador de paquetes. Localice la declaración de cualquiera de los atributos de la clase Producto (producto1, producto2, producto3 o producto4). Oprima la tecla *ctrl* y al mismo tiempo ubique el cursor sobre la palabra "Producto" en la declaración. La palabra "Producto" se resalta con un subrayado.
2. Haga clic sobre la palabra resaltada: se abrirá la clase Producto para ser consultada. En general, es posible localizar la declaración de cualquier elemento del programa utilizando esta misma interacción. Basta con posicionarse sobre el elemento cuya declaración queremos consultar y con la combinación *ctrl+clic* llegamos a dicho punto del programa.
3. Si desea puede ver el video explicativo en el sitio del proyecto.

Paso IV: localizar rápidamente los lugares donde se invoca un método

1. Seleccione y abra la clase Producto en el explorador de paquetes. Localice en el editor la declaración de cualquiera de los métodos o atributos de esta clase.
2. Busque todos los lugares del programa en donde se invoca dicho método, seleccionando la opción Navigate_/ Open Call Hierarchy _en el menú principal o en el menú emergente que aparece al hacer clic derecho sobre el método o con ctrl+alt+h.
3. En la vista de búsqueda de Eclipse se presentan todos los métodos en los que existe un llamado al método o atributo seleccionado.
4. Si selecciona un método, señala el lugar dónde este método fue llamado.

5. Si desea puede ver el video explicativo en el sitio del proyecto.
6. Repita el procedimiento anterior con el método constructor de la clase Tienda, para llegar hasta la clase de la interfaz de usuario que crea la tienda.

7. https://sicuaplus.uniandes.edu.co/bbcswebdav/users/cupitaller/Videos/Jerarquia_llamados.mp4

Paso V: cambiar los elementos de una clase

1. Seleccione y abra la clase Producto en el explorador de paquetes. Localice la declaración del atributo valorUnitario en dicha clase.
2. Cambie el nombre de este atributo a valorUnidad, seleccionando la opción *Refactor/Rename* en el menú principal o en el menú emergente que aparece al hacer clic derecho sobre el atributo. Esta operación realiza la modificación en todos los puntos del programa en los cuales se utiliza dicho atributo. La ventaja de hacer de esta manera los cambios es que el compilador ayuda a no cambiar por error otros elementos del programa.
3. Localice el método abastecer en la clase Producto. Cambie el nombre del parámetro pCantidad a pNumeroUnidades, de la misma manera que en el punto anterior. Esta misma técnica sirve para cambiar los nombres de los métodos. Si en algún punto del programa hay errores de compilación, es un poco arriesgado hacer los cambios de nombre mencionados en esta parte, ya que dichos errores pueden confundir al compilador y llevar a Eclipse a dejar de hacer algunos cambios necesarios.
4. Si desea puede ver el video explicativo en el sitio del proyecto.

10. Hojas de Trabajo

10.1 Hoja de Trabajo Nº 1: Un Estudiante

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 quiere que haga el programa y las restricciones para desarrollarlo.

Se desea construir una aplicación para el manejo de información de los cursos que está tomando un estudiante. El estudiante toma solo 4 cursos en el semestre. Los datos personales del estudiante que maneja la aplicación son código, nombre y apellido.

De cada curso se conoce:

- Código. Es el identificador del curso y no pueden haber dos cursos con el mismo código.
- Nombre.
- Departamento. Puede ser Matemáticas, Física, Sistemas o Biología.
- Cantidad de créditos.
- Nota obtenida en el curso. Este valor debe estar entre 1.5 y 5.

Para poder calcular el promedio del estudiante, se deben ponderar las notas, teniendo en cuenta la cantidad de créditos de las materias. Para esto, para cada curso se debe multiplicar la nota del curso con su cantidad de créditos, sumar estos valores y dividir esta suma por la cantidad total de créditos vistos por el estudiante. Por ejemplo, si el estudiante ha terminado dos materias, “Cálculo 1” y “Física 1”, la primera de 4 créditos y la segunda de tres, con las siguientes notas:

- Cálculo 1: 4,5
- Física 1: 3,5

El promedio del estudiante es:

- (4,5 x 4 + 3,5 x 3) / 7 = 4,07

Adicionalmente, se quiere poder saber si un estudiante está en prueba académica o si es candidato para beca. Para esto se debe tener en cuenta las siguientes reglas.

- Se considera que un estudiante está en prueba académica si su promedio es inferior a 3.25.

- Se considera que un estudiante es candidato a beca si su promedio es igual o superior a 4.75.

La aplicación debe permitir: (1) visualizar la información del estudiante, (2) visualizar la información de los cursos, (3) modificar la información de un curso, (4) asignar una nota a un curso (5) calcular el promedio del estudiante (6) indicar si el estudiante está en prueba académica, (7) indicar si el estudiante es candidato a beca.

La interfaz del programa es la siguiente:

Requerimientos funcionales. Especifique los siete requerimientos funcionales descritos en el enunciado.

Requerimiento Funcional 1

Nombre	**R1 – Visualizar la información del estudiante.**
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	**R2 – Visualizar la información de los cursos.**
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 3

Nombre	R3 – Modificar la información de un curso.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 4

Nombre	R4 – Asignar una nota a un curso.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 5

Nombre	R5 – Calcular el promedio del estudiante.
Resumen	
Entradas	
Resultado	

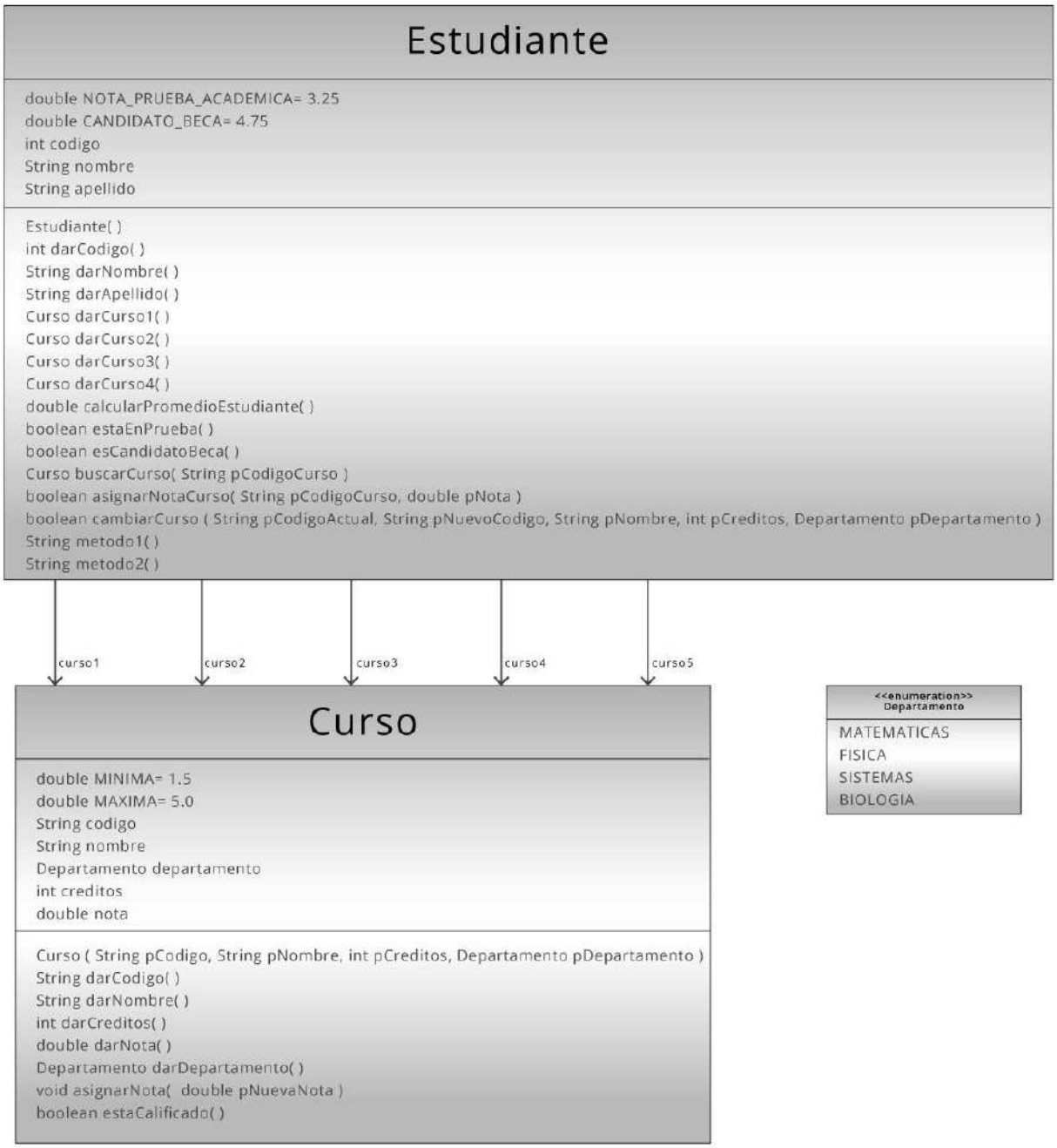
Requerimiento Funcional 6

Nombre	R6 – Indicar si el estudiante está en prueba académica.
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 7

Nombre	**R7 – Indicar si el estudiante es candidato a beca.**
Resumen	
Entradas	
Resultado	

Modelo conceptual. Estudie el siguiente modelo conceptual



Declaración de las clases. Complete las declaraciones de las siguientes clases.

```
public class Estudiante
{
    // ------------------------------------------------
    // Atributos
    // ------------------------------------------------




}
```

```
public class Curso
{
    // ------------------------------------------------
    // Atributos
    // ------------------------------------------------




}
```

Creación de Expresiones. Para cada uno de los siguientes enunciados, escriba la expresión que lo representa. Tenga en cuenta la clase dada para determinar los elementos disponibles.

Curso	**¿El nombre del curso es “Cálculo 1”?**	
Curso	¿El curso ya tiene una nota asignada?	
Curso	¿El curso tiene más de tres créditos?	
Curso	¿El curso fue aprobado?	
Estudiante	¿El código del estudiante es "1234"?	
Estudiante	¿El primer curso tiene una nota asignada?	
Estudiante	¿El segundo curso pertenece al departamento de matemáticas?	
Estudiante	¿Cuál es el promedio del estudiante?	

Desarrollo de métodos. Escriba el código de los métodos indicados.

Método 1

Clase: Curso

Descipción: Retorna el código del curso.

```
public String darCodigo( )
{




}
```

Método 2

Clase: Curso

Descipción: Indica si el curso ya fue calificado (tiene una nota distinta de cero).

```
public boolean estaCalificado( )
{




}
```

Método 3

Clase: Estudiante

Descipción: Retorna el nombre del estudiante.

```
public String darNombre( )
{




}
```

Método 4

Clase: Estudiante

Descipción: Indica si el estudiante ya tiene los cuatro cursos pertenecen al mismo departamento.

```
public boolean pertenecenMismoDepartamento( )
{




}
```

Método 5

Clase: Estudiante

Descipción: Calcula el promedio de los cursos que ya tienen nota. Si ningún curso tiene nota asignada, retorna cero.

```
public double calcularPromedioEstudiante( )
{









}
```

Método 6

Clase: Estudiante

Descipción: Busca y retorna el curso que tiene el código que se recibe como parámetro. Si ningún curso tiene dicho código, el método retorna null.

```
public Curso buscarCurso( String pCodigoCurso )
{









}
```

Método 7

Clase: Estudiante

Descipción: Indica si el estudiante se encuentra en prueba académica. Retorna verdadero si está en prueba académica, false de lo contrario.

```
public boolean estaEnPrueba( )
{









}
```

MANEJO DE GRUPOS DE ATRIBUTOS

03

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Utilizar las estructuras contenedoras de tamaño fijo como elementos para modelar una característica de un elemento del mundo que permiten almacenar una secuencia de valores (simples u objetos).
- Utilizar las estructuras contenedoras de tamaño variable como elementos de modelado que permiten manejar atributos cuyo valor es una secuencia de objetos.
- Utilizar las instrucciones iterativas para manipular estructuras contenedoras y entender que dichas instrucciones se pueden utilizar en otro tipo de problemas.
- Crear una clase completa en Java utilizando el ambiente de desarrollo Eclipse.
- 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

Cuando nos enfrentamos a la construcción del modelo conceptual del mundo del problema, en muchas ocasiones nos encontramos con el concepto de colección o grupo de cosas de la misma clase. Por ejemplo, si retomamos el caso de estudio del empleado presentado en el nivel 1 y lo generalizamos a la administración de todos los empleados de la universidad, es claro que en alguna parte del diagrama de clases debe aparecer el concepto de grupo de empleados. Además, cuando planteemos la solución, tendremos que definir un método en alguna clase para añadir un nuevo elemento a ese grupo (ingresó un nuevo empleado a la universidad) o un método para buscar un empleado de la universidad (por ejemplo, quién es el empleado que tiene mayor salario). De manera similar, si retomamos el caso de estudio del nivel 2 sobre la tienda, lo natural es que una tienda manipule un número arbitrario de productos, y no sólo cuatro de ellos como se definió en el ejemplo. En ese caso, la tienda debe poder agregar un nuevo producto al grupo de los que ya vende, buscar un producto en su catálogo, etc.

En este capítulo vamos a introducir dos conceptos fundamentales de la programación:

1. Las estructuras contenedoras, que nos permiten manejar atributos cuyo valor corresponde a una secuencia de elementos.
2. Las instrucciones repetitivas, que son instrucciones que nos permiten manipular los elementos contenidos en dichas secuencias.

Además, en este nivel estudiaremos la manera de crear objetos y agregarlos a una contenedora, la manera de crear una clase completa en Java y la forma de leer la descripción de un conjunto de clases desarrolladas por otros, para ser capaces de utilizarlas en nuestros programas.

Vamos a trabajar sobre varios casos de estudio que iremos introduciendo a lo largo del nivel.

3. Caso de Estudio Nº 1: Las Notas de un Curso

Considere el problema de administrar las calificaciones de los alumnos de un curso, en el cual hay doce estudiantes, de cada uno de los cuales se tiene la nota definitiva que obtuvo (un valor entre 0,0 y 5,0).

Se quiere construir un programa que permita:

1. Cambiar la nota de un estudiante.
2. Calcular el promedio del curso.
3. Establecer el número de estudiantes que está por encima de dicho promedio.

En la figura 3.1 aparece la interfaz de usuario que se quiere que tenga el programa.

Fig. 3.1 Interfaz de usuario del programa del primer caso de estudio

- En la ventana del programa aparece la nota de cada uno de los doce estudiantes del

curso. La nota con la que comienzan es siempre cero.
- Con el respectivo botón es posible modificar la nota. Al oprimirlo, aparece una ventana de diálogo en la que se pide la nueva nota.
- En la parte de abajo de la ventana se encuentran los botones que implementan los requerimientos funcionales: calcular el promedio e indicar el número de estudiantes que están por encima de dicha nota.

3.1. Comprensión de los Requerimientos

Requerimiento funcional 1

Nombre	R1 – Cambiar nota.
Resumen	Cambia la nota de uno de los estudiantes que pertenece a la lista del curso.
Entradas	(1) Número del estudiante, (2) nota del estudiante
Resultado	Se muestra la nueva nota del estudiante. En caso de que no cumpla el formato de número decimal con punto como separador, se muestra un mensaje de error.

Requerimiento funcional 2

Nombre	R2 – Calcular promedio de notas.
Resumen	Calcula el promedio de notas de la lista de estudiantes.
Entradas	Ninguna.
Resultado	Se muestra un mensaje con el promedio calculado.

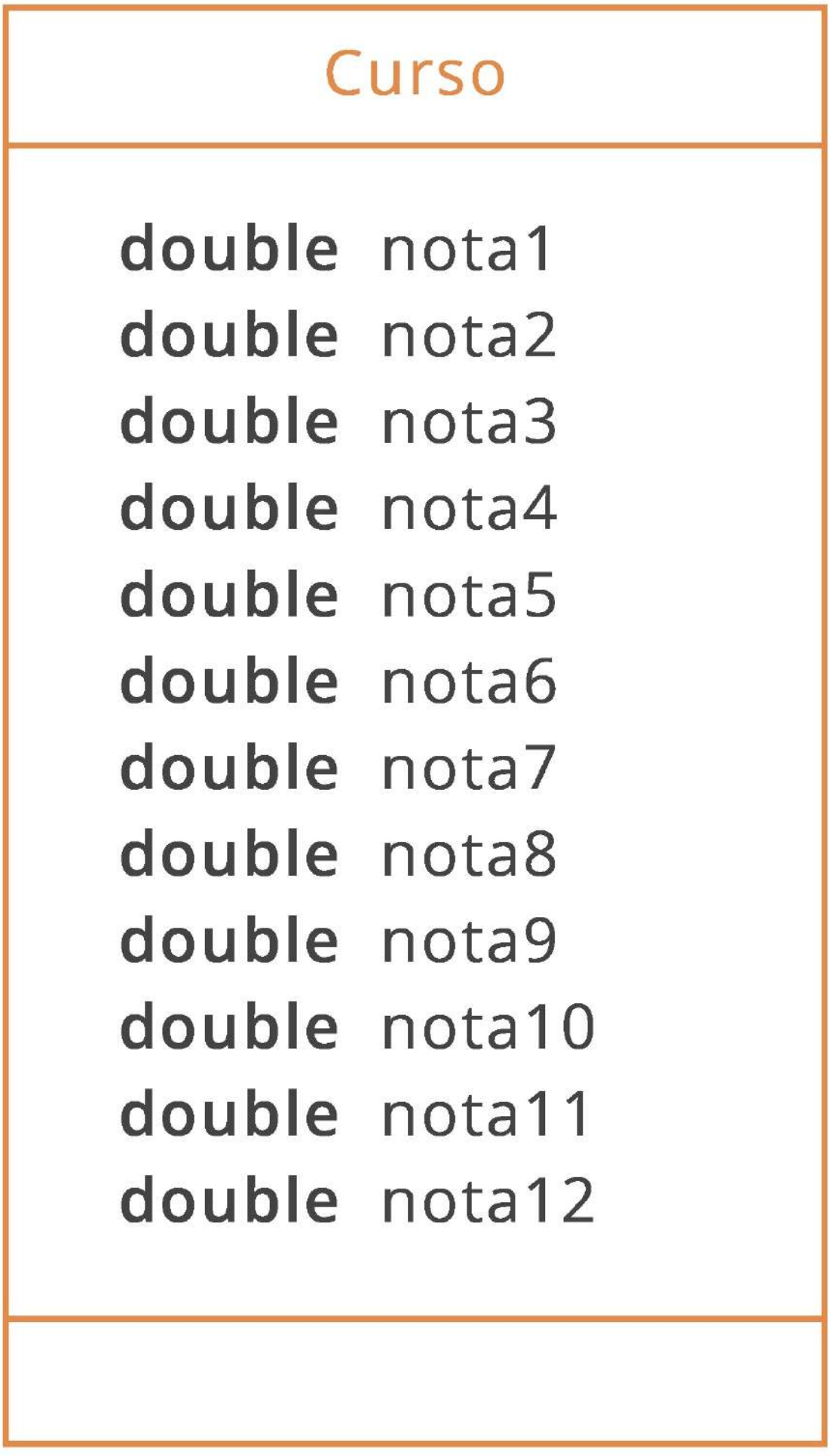
Requerimiento funcional 3

Nombre	R3 – Calcular la cantidad de estudiantes por encima del promedio.
Resumen	Calcula la cantidad de estudiantes que tienen una nota registrada mayor al promedio calculado.
Entradas	Ninguna.
Resultado	Se muestra un mensaje con la cantidad de estudiantes por encima del promedio.

3.2. Comprensión del Mundo del Problema

Dado el enunciado del problema, el modelo conceptual se puede definir con una clase llamada Curso, la cual tendría doce atributos de tipo `double` para representar las notas de cada uno de los estudiantes, tal como se muestra en la figura 3.2.

Fig. 3.2 Modelo conceptual de las calificaciones de los estudiantes

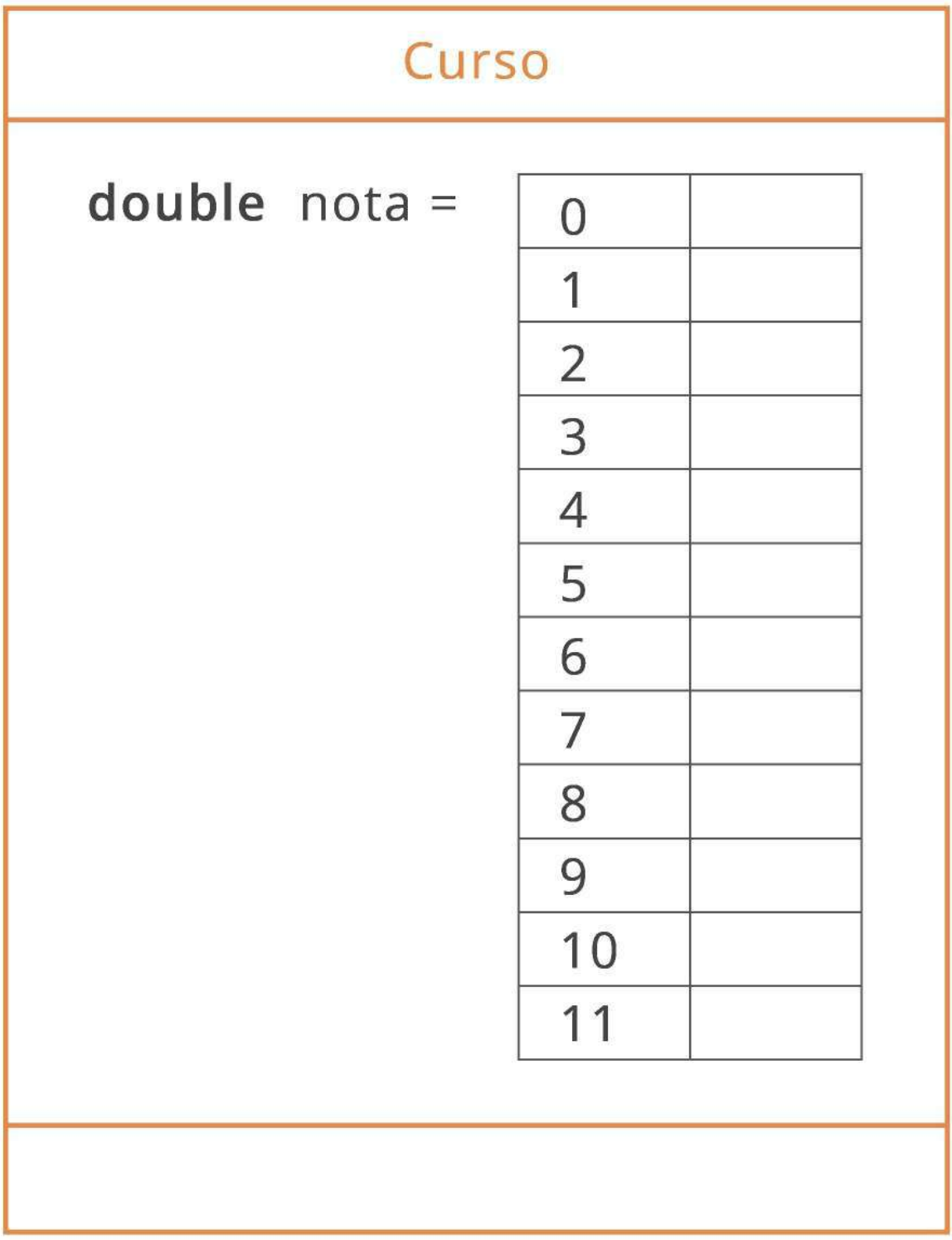

Curso
double nota1
double nota2
double nota3
double nota4
double nota5
double nota6
double nota7
double nota8
double nota9
double nota10
double nota11
double nota12

Aunque este modelado es correcto, los métodos necesarios para resolver el problema resultarían excesivamente largos y dispendiosos. Cada expresión aritmética para calcular cualquier valor del curso tomaría muchas líneas de código. Además, imagine si en vez de 12 notas tuviéramos que manejar 50 ó 100. Terminaríamos con algoritmos imposibles de leer y de mantener. Necesitamos una manera mejor de hacer este modelado y ésta es la motivación de introducir el concepto de estructura contenedora.

4. Contenedoras de Tamaño Fijo

Lo ideal, en el caso de estudio, sería tener un sólo atributo (llamado por ejemplo notas), en donde pudiéramos referirnos a uno de los valores individuales por un número que corresponda a su posición en el grupo (por ejemplo, la quinta nota). Ese tipo de atributos que son capaces de agrupar una secuencia de valores se denominan contenedoras y la idea se ilustra en la figura 3.3. Vale la pena aclarar que la sintaxis usada en la figura no corresponde a la sintaxis de UML, sino que solamente la usamos para ilustrar la idea de una estructura contenedora.

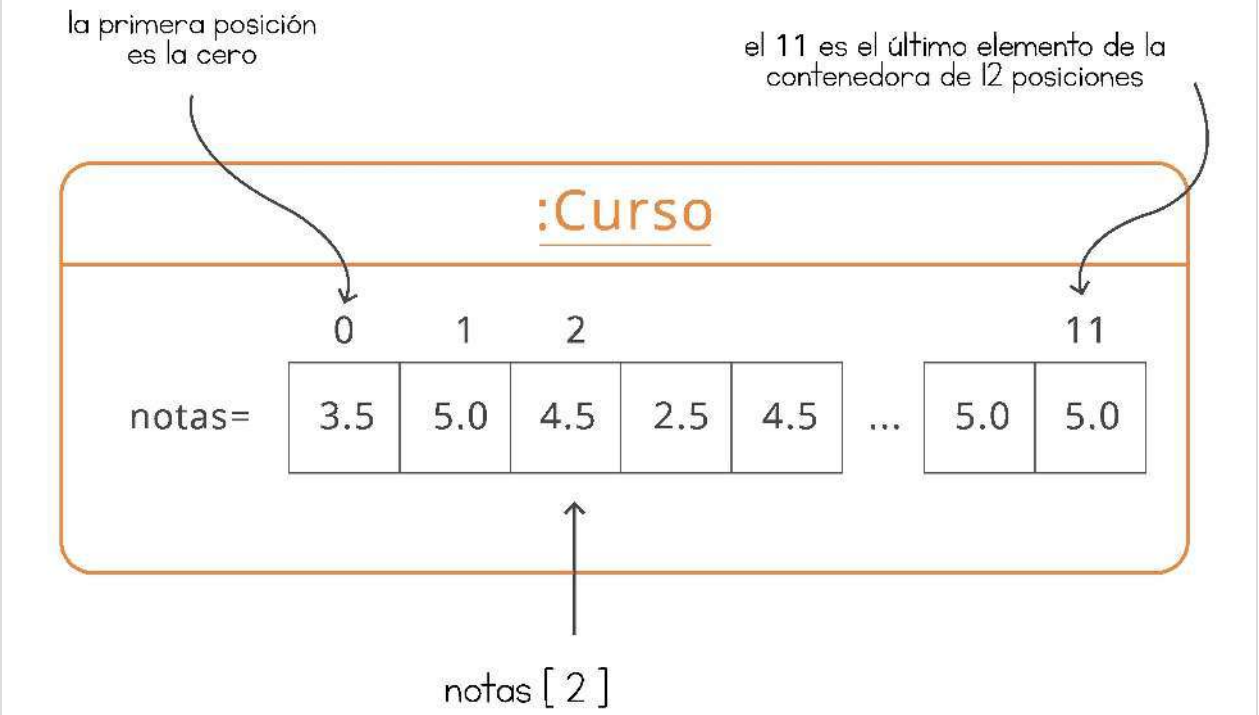
Fig. 3.3 Modelo conceptual de las calificaciones con una contenedora



- En lugar de tener 12 atributos de tipo real, vamos a tener un sólo atributo llamado "notas" el cual contendrá en su interior las 12 notas que queremos representar.
- Cada uno de los elementos del atributo "notas" se puede referenciar utilizando la sintaxis notas[x], donde x es el número del estudiante a quien corresponde la nota (comenzando en 0).
- Con esta representación podemos manejar de manera más simple y general el grupo de notas de los estudiantes.

Un objeto de la clase Curso se vería como aparece en la figura 3.4. Allí se puede apreciar que las posiciones dentro de una contenedora se comienzan a numerar a partir del valor 0 y que los elementos individuales se referencian a través de su posición. Cada nota va en una posición distinta de la contenedora de tipo `double` llamada notas.

Fig. 3.4 – Representación gráfica de un arreglo



En las secciones que siguen veremos la manera de declarar (en UML y en Java) un atributo que corresponda a una contenedora, y a manipular los valores allí incluidos.

4.1 Declaración de un Arreglo

En Java, las estructuras contenedoras de tamaño fijo se denominan arreglos (arrays en inglés), y se declaran como se muestra en el ejemplo 1. Los arreglos se utilizan para modelar una característica de una clase que corresponde a un grupo de elementos, de los cuales se conoce su número. Si no supiéramos, por ejemplo, el número de estudiantes del curso en el caso de estudio, deberíamos utilizar una contenedora de tamaño variable, que es el tema de una sección posterior de este capítulo.

Ejemplo 1

Objetivo: Mostrar la sintaxis usada en Java para declarar un arreglo.

En este ejemplo se hace la declaración del arreglo de notas, como parte de la clase Curso del caso de estudio.

```
public class Curso
{
    //-----------------------------------
    // Constantes
    //-----------------------------------
    public final static int TOTAL_EST = 12;

    //-----------------------------------
    // Atributos
    //-----------------------------------
    private double[] notas;
    ...
}
```

- Es conveniente declarar el número de posiciones del arreglo como una constante (`TOTAL_EST`). Eso facilita realizar más tarde modificaciones al programa. Si en vez de 12 hay que manejar 15 estudiantes, bastaría con cambiar dicho valor.
- En el momento de declarar el atributo " `notas` ", usamos la sintaxis " `[]` " para indicar que va a contener un grupo de valores.
- El tamaño del arreglo será determinado en el momento de la inicialización del arreglo, en el método constructor. Por ahora no hay que decir nada al respecto.
- En la declaración le decimos al compilador que todos los elementos del arreglo son de tipo `double` .
- Recuerde que los elementos de un arreglo se comienzan a referenciar a partir de la posición 0.

4.2 Inicialización de un Arreglo

Al igual que con cualquier otro atributo de una clase, es necesario inicializar los arreglos en el método constructor antes de poderlos utilizar. Para hacerlo, se debe definir el tamaño del arreglo, es decir el número de elementos que va a contener. Esta inicialización es obligatoria, puesto que es en ese momento que le decimos al computador cuántos valores debe manejar en el arreglo, lo que corresponde al espacio en memoria que debe reservar. Veamos en el ejemplo 2 cómo se hace esto para el caso de estudio.

> Si tratamos de acceder a un elemento de un arreglo que no ha sido inicializado, vamos a obtener el error de ejecución: *java.lang.NullPointerException*

Ejemplo 2

Objetivo: Mostrar la manera de inicializar un arreglo en Java.

En este ejemplo mostramos, en el contexto del caso de estudio, la manera de inicializar el arreglo de notas dentro del constructor de la clase Curso.

```
public Curso( )
{
    notas = new double[ TOTAL_EST ] ;
}
```

- Se utiliza la instrucción `new` como con cualquier otro objeto, pero se le especifica el número de valores que debe contener el arreglo (TOTAL_EST, que es una constante de valor 12).
- Esta construcción reserva el espacio para el arreglo, pero el valor de cada uno de los elementos del arreglo sigue siendo indefinido. Esto lo arreglaremos más adelante.

El lenguaje Java provee un operador especial (`length`) para los arreglos, que permite consultar el número de elementos que éstos contienen. En el caso de estudio, la expresión `notas.length` debe dar el valor 12, independientemente de si los valores individuales ya han sido o no inicializados, puesto que en el método constructor de la clase se reservó dicho espacio de memoria.

4.3. Acceso a los Elementos del Arreglo

Un índice es un valor entero que nos sirve para indicar la posición de un elemento en un arreglo. Los índices van desde 0 hasta el número de elementos menos 1. En el caso de estudio, la primera nota tiene el índice 0 y la última, el índice 11. Para tomar o modificar el valor de un elemento particular de un arreglo necesitamos dar su índice, usando la sintaxis que aparece en el siguiente método de la clase Curso y que, en el caso general, se puede resumir como `<arreglo>[<índice>]` .

```
public void noHaceNadaUtil( double valor )
{
    int indice = 10;
    notas[ 0 ] = 3.5;
    if( valor < 2.5 && notas.length == TOTAL_EST )
    {
        notas[ indice ] = notas[ 0 ];
        notas[ 0 ] = valor + 1.0;
    }
    else
    {
        notas[ indice ] = notas[ 0 ] - valor;
    }
}
```

- Este método sólo lo utilizamos para ilustrar la sintaxis que se utiliza en Java para manipular los elementos de un arreglo.
- Para asignar un valor a una casilla del arreglo, usamos la sintaxis `notas[ x ] = valor`, donde x es el índice que nos indica una posición.
- Para obtener el valor de una casilla, usamos la misma sintaxis (`notas[ x ]`) y para conocer el número de casillas del arreglo usamos `notas.length` .

De esta manera podemos asignar cualquier valor de tipo `double` a cualquiera de las casillas del arreglo, o tomar el valor que allí se encuentra.

> Cuando dentro de un método tratamos de acceder una casilla con un índice no válido (menor que 0 o mayor o igual que el número de casillas), obtenemos el error de ejecución: *java.lang.ArrayIndexOutOfBoundsException*

Es importante destacar que, hasta este momento, lo único que hemos ganado con la introducción de los arreglos es no tener que usar atributos individuales para representar una característica que incluye un grupo de elementos. Es más cómodo tener un sólo atributo con todos esos elementos en su interior. Las verdaderas ventajas de usar arreglos las veremos a continuación, al introducir las instrucciones repetitivas.

5. Instrucciones Repetitivas

5.1. Introducción

En muchos problemas notamos una regularidad que sugiere que su solución puede lograrse repitiendo un paso que vaya transformando gradualmente el estado del mundo modelado y acercándose a la solución. Instintivamente es lo que hacemos cuando subimos unas escaleras: repetimos el paso de subir un escalón hasta que llegamos al final. Otro ejemplo posible es si suponemos que tenemos en una hoja de papel una lista de palabras sin ningún orden y nos piden buscar si la palabra "casa" está en la lista. El algoritmo que seguimos para realizar está tarea puede ser descrito de la siguiente manera:

1. Verifique si la primera palabra es igual a "casa".
2. Si lo es, no busque más. Si no lo es, busque la segunda palabra.
3. Verifique si la segunda palabra es igual a "casa".
4. Si lo es, no busque más. Si no lo es, busque la tercera palabra.
5. Repita el procedimiento palabra por palabra, hasta que la encuentre o hasta que no haya más palabras para buscar.

Tarea 1

Objetivo: Explicar el significado de la instrucción repetitiva y usarla para definir un algoritmo que resuelva un problema simple.

Suponga que en el ejemplo anterior, ya no queremos buscar una palabra sino contar el número total de letras que hay en todas las palabras de la hoja.

Escriba el algoritmo para resolver el problema:

5.2. Calcular el Promedio de las Notas

Para resolver el segundo requerimiento del caso de estudio (R2 - calcular el promedio de las notas), debemos calcular la suma de todas las notas del curso para luego dividirlo por el número de estudiantes. Esto se puede hacer con el método que se muestra a continuación:

```
public double promedio( )
{
    double suma = notas[ 0 ] + notas[ 1 ] + notas[ 2 ] +
                  notas[ 3 ] + notas[ 4 ] + notas[ 5 ] +
                  notas[ 6 ] + notas[ 7 ] + notas[ 8 ] +
                  notas[ 9 ] + notas[ 10 ] + notas[ 11 ];
    return suma / TOTAL_EST;
}
```

- Primero sumamos las notas de todos los estudiantes y guardamos el valor en la variable suma.
- El promedio corresponde a dividir dicho valor por el número de estudiantes, representado con la constante `TOTAL_EST` .

Si planteamos el problema de manera iterativa, podemos escribir el mismo método de la siguiente manera, en la cual, en cada paso, acumulamos el valor del siguiente elemento:

```
public double promedio( )
{
    double suma = 0.0;
    int indice = 0;

    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];
    indice++;
    suma += notas[ indice ];

    return suma / TOTAL_EST;
}
```

- Esta solución también calcula el promedio del curso, pero en lugar de hacer referencia directa a las doce casillas del arreglo, utiliza un índice que va desplazando desde 0 hasta 11.
- Por supuesto que es más clara la solución anterior, pero queremos utilizar este ejemplo para introducir las instrucciones iterativas, que expresan esta misma idea de "desplazar" un índice, pero usando una sintaxis mucho más compacta.
- Lo primero que debemos notar es que vamos a ejecutar 12 veces (`TOTAL_EST` veces para ser exactos) un grupo de instrucciones.
- Ese grupo de instrucciones es: `suma += notas[ indice ]; indice++` ;
- Después de ejecutar 12 veces esas dos instrucciones, en la variable suma tendremos el valor total, listo para dividirlo por el número de estudiantes.
- El índice comienza teniendo el valor 0 y termina teniendo el valor 11. De esta manera, cada vez que hacemos referencia al elemento `notas[indice]` , estamos hablando de una casilla distinta del arreglo.

Allí repetimos 12 veces una pareja de instrucciones, una vez por cada elemento del arreglo. Basta un poco de reflexión para ver que lo que necesitamos es poder decir que esas dos instrucciones se deben repetir tantas veces como notas haya en el arreglo. Las instrucciones repetitivas nos permiten hacer eso de manera sencilla. En el siguiente método se ilustra el uso de la instrucción `while` para el mismo problema del cálculo del promedio.

```
public double promedio( )
{
    double suma = 0.0;
    int indice = 0;
    while( indice < TOTAL_EST )
    {
        suma += notas[ indice ];
        indice++;
    }
    return suma / TOTAL_EST;
}
```

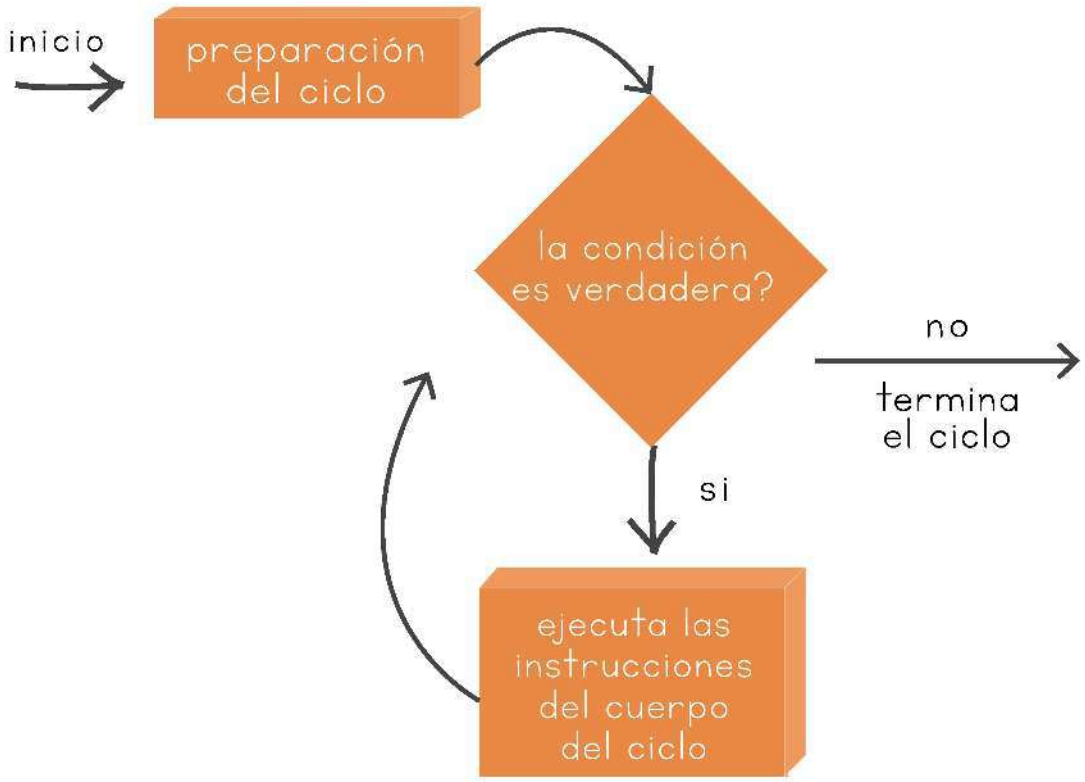
- La estructura del método sigue siendo la misma, con la única diferencia de que en lugar de repetir 12 veces la pareja de instrucciones, las incluimos dentro de la instrucción `while`, que se encarga de ejecutar repetidamente las instrucciones que tiene en su interior.
- La instrucción `while` sirve para decirle al computador que "mientras que" una condición se cumpla, siga ejecutando las instrucciones que están por dentro.
- La condición en el ejemplo es `indice < TOTAL_EST`, que equivale a decirle que "mientras que" el índice no llegue a 12, vuelva a ejecutar la pareja de instrucciones que tiene asociadas.

Ahora veremos las partes de las instrucciones repetitivas y su significado.

5.3. Componentes de una Instrucción Repetitiva

La figura 3.5 ilustra la manera en que se ejecuta una instrucción repetitiva. Primero, y por una sola vez, se ejecutan las instrucciones que vamos a llamar de inicio o preparación del ciclo. Allí se le da el valor inicial al índice y a las variables en las que queremos acumular los valores durante el recorrido. Luego, se evalúa la condición del ciclo. Si es falsa, se ejecutan las instrucciones que se encuentran después del ciclo. Si es verdadera, se ejecutan las instrucciones del cuerpo del ciclo para finalmente volver a repetir el mismo proceso. Cada repetición, que incluye la evaluación de la condición y la ejecución del cuerpo del ciclo, recibe el nombre de **iteración** o **bucle**.

Fig. 3.5 Ejecución de una instrucción repetitiva



Usualmente en un lenguaje de programación hay varias formas de escribir una instrucción repetitiva. En Java existen varias formas, pero en este libro sólo vamos a presentar dos de ellas: la instrucción `for` y la instrucción `while`.

5.3.1. Las Instrucciones for y while

Una instrucción repetitiva con la instrucción `while` se escribe de la siguiente manera:

```
<inicio>
while( <condición> )
{
    <cuerpo>
    <avance>
}
```

- Las instrucciones de preparación del ciclo van antes de la instrucción repetitiva.
- La condición que establece si se debe repetir de nuevo el ciclo va siempre entre paréntesis.
- El avance del ciclo es una parte opcional, en la cual se modifican los valores de algunos de los elementos que controlan la salida del ciclo (avanzar el índice con el que

se recorre un arreglo sería parte de esta sección).

Una instrucción repetitiva con la instrucción `for` se escribe de la siguiente manera:

```
<inicio1>
for( <inicio2>; <condición>; <avance> )
{
    <cuerpo>
}
```

- El inicio va separado en dos partes: en la primera, va la declaración y la inicialización de las variables que van a ser utilizadas después de terminado el ciclo (la variable `suma`, por ejemplo, en el método del promedio). En la segunda parte de la zona de inicio van las variables que serán utilizadas únicamente dentro de la instrucción repetitiva (la variable `índice`, por ejemplo, que sólo sirve para desplazarse recorriendo las casillas del arreglo).
- La segunda parte del inicio, lo mismo que el avance del ciclo, se escriben en el encabezado de la instrucción `for`.

Ejemplo 3

Objetivo: Mostrar la manera de utilizar la instrucción iterativa `for`.

En este ejemplo se presenta una implementación del método que calcula el promedio de notas del caso de estudio, en la cual se utiliza la instrucción `for`.

```
public double promedio( )
{
    double suma = 0.0;
    for(int indice = 0; indice < TOTAL_EST; indice++ )
    {
        suma += notas[ indice ];
    }
    return suma / TOTAL_EST;
}
```

- Puesto que la variable "`suma`" será utilizada por fuera del cuerpo del ciclo, es necesario declararla antes del for.
- La variable "`indice`" es interna al ciclo, por eso se declara dentro del encabezado.
- El avance del ciclo consiste en incrementar el valor del "`indice`".
- En este ejemplo, los corchetes del `for` son opcionales, porque sólo hay una instrucción dentro del cuerpo del ciclo.

Vamos a ver en más detalle cada una de las partes de la instrucción y las ilustraremos con algunos ejemplos.

5.3.2. El Inicio del Ciclo

El objetivo de las instrucciones de inicio o preparación del ciclo es asegurarnos de que vamos a empezar el proceso repetitivo con las variables de trabajo en los valores correctos. En nuestro caso, una variable de trabajo la utilizamos como índice para movernos por el arreglo y la otra para acumular la suma de las notas:

- La suma antes de empezar el ciclo debe ser cero: `double suma = 0.0;`
- El índice a partir del cual vamos a iterar debe ser cero: `int indice = 0;`

5.3.3. La Condición para Continuar

El objetivo de la condición del ciclo es identificar el caso en el cual se debe volver a hacer una nueva iteración. Esta condición puede ser cualquier expresión lógica: si su evaluación da verdadero, significa que se deben ejecutar de nuevo las instrucciones del ciclo. Si es falsa, el ciclo termina y se continúa con la instrucción que sigue después de la instrucción repetitiva.

Típicamente, cuando se está recorriendo un arreglo con un índice, la condición del ciclo dice que se debe volver a iterar mientras el índice sea menor que el número total de elementos del arreglo. Para indicar este número, se puede utilizar la constante que define su tamaño (`TOTAL_EST`) o el operador que calcula el número de elementos de un arreglo (`notas.length`).

> Dado que los arreglos comienzan en 0, la condición del ciclo debe usar el operador `<` y el número de elementos del arreglo. Son errores comunes comenzar los ciclos con el índice en 1 o tratar de terminar con la condición `indice <= notas.length`.

5.3.4. El Cuerpo del Ciclo

El cuerpo del ciclo contiene las instrucciones que se van a repetir en cada iteración. Estas instrucciones indican:

- La manera de modificar algunas de las variables de trabajo para ir acercándose a la solución del problema. Por ejemplo, si el problema es encontrar la suma de las notas de todos los estudiantes del curso, con la instrucción `suma += notas[indice]` agregamos un nuevo valor al acumulado.
- La manera de modificar los elementos del arreglo, a medida que el índice pasa por cada casilla. Por ejemplo, si queremos sumar una décima a todas las notas, lo

hacemos con la instrucción `notas[indice] += 0.1`.

5.3.5. El Avance del Ciclo

Cuando se recorre un arreglo, es necesario mover el índice que indica la posición en la que estamos en un momento dado (`indice++`). En algún punto (en el avance o en el cuerpo) debe haber una instrucción que cambie el valor de la condición para que finalmente ésta sea falsa y se detenga así la ejecución de la instrucción iterativa. Si esto no sucede, el programa se quedará en un ciclo infinito.

> Si construimos un ciclo en el que la condición nunca sea falsa (por ejemplo, si olvidamos escribir las instrucciones de avance del ciclo), el programa dará la sensación de que está bloqueado en algún lado, o podemos llegar al error: *java.lang.OutOfMemoryError*

Tarea 2

Objetivo: Practicar el desarrollo de métodos que tengan instrucciones repetitivas.

Para el caso de estudio de las notas de los estudiantes escriba los métodos de la clase Curso que resuelven los problemas planteados.

Calcular el número de estudiantes que sacaron una nota entre 3,0 y 5,0:

```
public int calcularCantidadAprobados( )
{









}
```

Calcular la mayor nota del curso:

```
public double calcularMayorNota( )
{









}
```

Contar el número de estudiantes que sacaron una nota inferior a la del estudiante que está en la posición del arreglo que se entrega como parámetro. Suponga que el parámetro `pPosEst` tiene un valor comprendido entre `0` y `TOTAL_EST - 1`.

```
public int calcularCantidadNotasInferioresA( int pPosEst )
{









}********
```

Aumentar el 5% todas las notas del curso, sin que ninguna de ellas sobrepase el valor 5,0:

```
public void hacerCurva( )
{









}
```

5.4. Patrones de Algoritmo para Instrucciones Repetitivas

Cuando trabajamos con estructuras contenedoras, las soluciones de muchos de los problemas que debemos resolver son similares y obedecen a ciertos esquemas ya conocidos (¿cuántas personas no habrán resuelto ya los mismos problemas que estamos aquí resolviendo?). En esta sección pretendemos identificar tres de los patrones que más se repiten en el momento de escribir un ciclo, y con los cuales se pueden resolver todos los problemas del caso de estudio planteados hasta ahora. Lo ideal sería que, al leer un problema que debemos resolver (el método que debemos escribir), pudiéramos identificar el patrón al cual corresponde y utilizar las guías que existen para resolverlo. Eso simplificaría enormemente la tarea de escribir los métodos que tienen ciclos.

Un patrón de algoritmo se puede ver como una solución genérica para un tipo de problemas, en la cual el programador sólo debe resolver los detalles particulares de su problema específico.

En esta sección vamos a introducir tres patrones que se diferencian por el tipo de recorrido que hacemos sobre la secuencia.

5.4.1. Patrón de Recorrido Total

En muchas ocasiones, para resolver un problema que involucra una secuencia, necesitamos recorrer todos los elementos que ésta contiene para lograr la solución. En el caso de estudio de las notas tenemos varios ejemplos de esto:

- Calcular la suma de todas las notas.
- Contar cuántos en el curso obtuvieron la nota 3,5.
- Contar cuántos estudiantes aprobaron el curso.
- Contar cuántos en el curso están por debajo del promedio (conociendo este valor).
- Aumentar en 10% todas las notas inferiores a 2,0.

¿Qué tienen en común los algoritmos que resuelven esos problemas? La respuesta es que la solución requiere siempre un recorrido de todo el arreglo para poder cumplir el objetivo que se está buscando: debemos pasar una vez por cada una de las casillas del arreglo. Esto significa:

1. Que el índice para iniciar el ciclo debe empezar en cero.
2. Que la condición para continuar es que el índice sea menor que la longitud del arreglo.
3. Que el avance consiste en sumarle uno al índice.

Esa estructura que se repite en todos los algoritmos que necesitan un recorrido total es lo que denominamos el **esqueleto del patrón**, el cual se puede resumir con el siguiente fragmento de código:

```
for( int indice = 0; indice < arreglo.length; indice++ )
{
    <cuerpo>
}
```

- Es común que en lugar de la variable " `indice` " se utilice una variable llamada " `i` ". Esto hace el código un poco más compacto.
- En lugar del operador " `length` ", se puede utilizar también la constante que indica el número de elementos del arreglo.
- Los corchetes del " `for` " sólo son necesarios si el cuerpo tiene más de una instrucción.

Lo que cambia en cada caso es lo que se quiere hacer en el cuerpo del ciclo. Aquí hay dos variantes principales. En la primera, algunos de los elementos del arreglo van a ser modificados siguiendo una regla (por ejemplo, aumentar en 10% todas las notas inferiores a 2,0). Lo único que se hace en ese caso es reemplazar el del esqueleto por las instrucciones que hacen la modificación pedida a un elemento del arreglo (el que se encuentra en la posición indice). Esa variante se ilustra en el ejemplo 4.

Ejemplo 4

Objetivo: Mostrar la primera variante del patrón de recorrido total.

En este ejemplo se presenta la implementación del método de la clase Curso que aumenta en 10% todas las notas inferiores a 2,0.

```
public void hacerCurva( )
{
    for( int i = 0; i < notas.length; i++ )
    {
        if( notas[ i ] < 2.0 )
        {
            notas[ i ] = notas[ i ] * 1.1;
        }
    }
}
```

- El esqueleto del patrón de algoritmo de recorrido total se copia dentro del cuerpo del método.
- Se reemplaza el cuerpo del patrón por la instrucción condicional que hace la modificación pedida.

- En el cuerpo se indica la modificación que debe sufrir el elemento que está siendo referenciado por el índice con el que se recorre el arreglo.

La segunda variante corresponde a calcular alguna propiedad sobre el conjunto de elementos del arreglo (por ejemplo, contar cuántos estudiantes aprobaron el curso). Esta variante implica cuatro decisiones que definen la manera de completar el esqueleto del patrón:

1. Cómo acumular la información que se va llevando a medida que avanza el ciclo.
2. Cómo inicializar dicha información.
3. Cuál es la condición para modificar dicho acumulado en el punto actual del ciclo.
4. Cómo modificar el acumulado.

En el ejemplo 5 se ilustra esta variante.

Ejemplo 5

Objetivo: Mostrar la segunda variante del patrón de recorrido total.

En este ejemplo se presenta la aplicación del patrón de algoritmo de recorrido total, para el problema de contar el número de estudiantes que aprobaron el curso.

- ¿Cómo acumular información?

Vamos a utilizar una variable de tipo entero llamada `vanAprobando`, que va llevando durante el ciclo el número de estudiantes que aprobaron el curso.

- ¿Cómo inicializar el acumulado?

La variable `vanAprobando` se debe inicializar en 0, puesto que inicialmente no hemos encontrado todavía ningún estudiante que haya pasado el curso.

- ¿Condición para cambiar el acumulado?

Cuando `notas[ indice ]` sea mayor o igual a 3,0, porque quiere decir que hemos encontrado otro estudiante que pasó el curso.

- ¿Cómo modificar el acumulado?

El acumulado se modifica incrementándolo en 1.

```
public int darCantidadAprobados( )
{
    int vanAprobando = 0;
    for( int i = 0; i < notas.length; i++ )
    {
        if( notas[ i ] >= 3.0 )
        {
            vanAprobando++;
        }
    }
    return vanAprobando;
}
```

- Las cuatro decisiones tomadas anteriormente van a definir la manera de completar el esqueleto del algoritmo definido por el patrón.
- Las decisiones 1 y 2 definen el inicio del ciclo.
- Las decisiones 3 y 4 ayudan a construir el cuerpo del mismo.

A continuación se muestra cómo sería el método anterior utilizando la instrucción for-each.

```
public int darCantidadAprobados( )
{
    int vanAprobando = 0;
    for( Double nota: notas )
    {
        if( nota >= 3.0 )
        {
            vanAprobando++;
        }
    }
    return vanAprobando;
}
```

> En resumen, si el problema planteado corresponde al patrón de recorrido total, se debe identificar la variante y luego tomar las decisiones que definen la manera de completar el esqueleto.

Tarea 3

Objetivo: Generar habilidad en el uso del patrón de algoritmo de recorrido total.

Escriba los métodos de la clase Curso que resuelven los siguientes problemas, los cuales corresponden a las dos variantes del patrón de algoritmo de recorrido total.

Escriba un método para modificar las notas de los estudiantes de la siguiente manera: a todos los que obtuvieron más de 4,0, les quita 0,5. A todos los que obtuvieron menos de 2,0, les aumenta 0,5. A todos los demás, les deja la nota sin modificar:

```
public void cambiarNotas( )
{









}
```

Escriba un método que retorne la menor nota del curso:

```
public double darMenorNota ( )
{









}
```

Escriba un método que indique en cuál rango se encuentra la mayoría de las notas del curso. Los rangos están definidos de la siguiente manera: rango 1 de 0,0 a 1,99, rango 2 de 2,0 a 3,49, rango 3 de 3,5 a 5,0. El método debe retornar el número del rango.

```
public int darRangoConMasNotas( )
{









}
```

5.4.2. Patrón de Recorrido Parcial

En algunos problemas de manejo de secuencias no es necesario recorrer todos los elementos para lograr el objetivo propuesto. Piense en la solución de los siguientes problemas:

- Informar si algún estudiante obtuvo la nota 5,0.
- Buscar el primer estudiante con nota igual a cero.
- Indicar si más de 3 estudiantes perdieron el curso.
- Aumentar el 10% en la nota del primer estudiante que haya sacado más de 4,0.

En todos esos casos hacemos un recorrido del arreglo, pero éste debe terminar tan pronto hayamos resuelto el problema. Por ejemplo, el método que informa si algún estudiante obtuvo cinco en la nota del curso debe salir del proceso iterativo tan pronto localice el primer estudiante con esa nota. Sólo si no lo encuentra, va a llegar hasta el final de la secuencia.

Un **recorrido parcial** se caracteriza porque existe una condición que debemos verificar en cada iteración para saber si debemos detener el ciclo o volver a repetirlo.

En este patrón, debemos adaptar el esqueleto del patrón anterior para que tenga en cuenta la condición de salida, de la siguiente manera:

```
boolean termino = false;

for( int i = 0; i < arreglo.length && !termino; i++ )
{
    <cuerpo>

    if( <ya se cumplió el objetivo> )
    {
        termino = true;
    }
}
```

- Primero, declaramos una variable de tipo `boolean` para controlar la salida del ciclo, y la inicializamos en false.
- Segundo, en la condición del ciclo usamos el valor de la variable que acabamos de definir: si su valor es verdadero, no debe volver a iterar.
- Tercero, en algún punto del ciclo verificamos si el problema ya ha sido resuelto (si ya se cumplió el objetivo). Si ése es el caso, cambiamos el valor de la variable a verdadero.

```
for( int i = 0; i < arreglo.length && !<condición>; i++ )
{
    <cuerpo>
}
```

Este patrón de esqueleto es más simple que el anterior, pero sólo se debe usar si la expresión que indica que ya se cumplió el objetivo del ciclo es sencilla.

> Cuando se aplica el patrón de recorrido parcial, el primer paso que se debe seguir es identificar la condición que indica que el problema ya fue resuelto. Con esa información se puede tomar la decisión de cuál esqueleto de algoritmo es mejor usar.

Ejemplo 6

Objetivo: Mostrar el uso del patrón de recorrido parcial para resolver un problema.

En este ejemplo se presentan tres soluciones posibles al problema de decidir si algún estudiante obtuvo cinco en la nota del curso.

```
public boolean hayAlguienConCinco( )
{
    boolean termino = false;

    for( int i = 0; i < notas.length && !termino; i++ )
    {
        if( notas[ i ] == 5.0 )
        {
            termino = true;
        }
    }

    return termino;
}
```

- La condición para no seguir iterando es que se encuentre una nota igual a 5,0 en la posición `i`.
- Al final del método, se retorna el valor de la variable "`termino`", que indica si el objetivo se cumplió. Esto funciona en este caso particular, porque dicha variable dice que en el arreglo se encontró una nota igual al valor buscado.

```
public boolean hayAlguienConCinco( )
{
    int i = 0;
    while( i < notas.length && notas[ i ] != 5.0 )
    {
        i++;
    }
    return i < notas.length;
}
```

- Esta es la segunda solución posible, y evita el uso de la variable "`termino`", pero tiene varias consecuencias sobre la instrucción iterativa.
- En lugar de la instrucción `for` es más conveniente usar la instrucción `while`.
- La condición de continuación en el ciclo es que la i-ésima nota sea diferente de 5,0.
- El método debe retornar verdadero si la variable `i` no llegó hasta el final del arreglo, porque esto querría decir que encontró en dicha posición una nota igual a cinco.

```
public boolean hayAlguienConCinco( )
{
    for( int i = 0; i < notas.length; i++ )
    {
        if( notas[ i ] == 5.0 )
        {
            return true;
        }
    }
    return false;
}
```

- Esta es la tercera solución posible. Si dentro del ciclo ya tenemos la respuesta del método, en lugar de utilizar la condición para salir del ciclo, la usamos para salir de todo el método.
- En la última instrucción retorna falso, porque si llega a ese punto quiere decir que no encontró ninguna nota con el valor buscado.
- Esta manera de salir de un ciclo, terminando la ejecución del método en el que éste se encuentra, se debe usar con algún cuidado, puesto que se puede producir código difícil de entender.

> Hay muchas soluciones posibles para resolver un problema. Un patrón de algoritmo sólo es una guía que se debe adaptar al problema específico y al estilo preferido del programador.

Para el patrón de recorrido parcial aparecen las mismas dos variantes que para el patrón de recorrido total (ver ejemplo 7):

- En la primera variante se modifican los elementos del arreglo hasta que una condición se cumpla (por ejemplo, encontrar las tres primeras notas con 1,5 y asignarles 2,5). En ese caso, en el cuerpo del método va la modificación que hay que hacerle al elemento que se encuentra en el índice actual, pero se debe controlar que cuando haya llegado a la tercera modificación termine el ciclo.
- En la segunda variante, se deben tomar las mismas cuatro decisiones que se tomaban con el patrón de recorrido total, respecto de la manera de acumular la información para calcular la respuesta que está buscando el método.

Ejemplo 7

Objetivo: Mostrar el uso del patrón de recorrido parcial, en sus dos variantes.

En este ejemplo se presentan dos métodos de la clase Curso, en cada uno de los cuales se ilustra una de las variantes del patrón de recorrido parcial.

Encontrar las primeras tres notas iguales a 1,5 y asignarles 2,5:

```
public void subirNotas( )
{
    int numNotas  =  0;
    for( int i =  0; i <  notas.length &&  numNotas  <  3; i++ )
    {
        if( notas[ i ] == 1,5 )
        {
            numNotas++;
            notas[ i ] =  2,5;
        }
    }
}
```

Este método corresponde a la primera variante, porque hace una modificación de los elementos del arreglo hasta que una condición se cumpla. En el método del ejemplo, debemos contar el número de modificaciones que hacemos, para detenernos al llegar a la tercera.

Retornar la posición en la secuencia de la tercera nota con valor 5,0. Si dicha nota no aparece al menos 3 veces, el método debe retornar el valor –1:

```
public int darTercerCinco( )
{
    int cuantosCincos = 0;
    int posicion = -1;
    for( int i =  0; i < notas.length && posicion == -1; i++ )
    {
        if( notas[ i ] == 5,0 )
        {
            cuantosCincos++;
            if( cuantosCincos == 3 )
            {
                posicion = i;
            }
        }
    }
    return posicion;
}
```

- ¿Cómo acumular información? En este caso necesitamos dos variables para acumular la información: la primera para llevar el número de notas iguales a 5,0 que han aparecido (`cuantosCincos`), la segunda para indicar la posición de la tercera nota 5,0 (`posicion`).
- ¿Cómo inicializar el acumulado? La variable `cuantosCincos` debe comenzar en 0. La variable `posicion` debe comenzar en menos 1.

- ¿Condición para cambiar el acumulado? Si la nota actual es 5,0 debemos cambiar nuestro acumulado.
- ¿Cómo modificar el acumulado? Debe cambiar la variable `cuantosCincos`, incrementándose en 1. Si es el tercer 5,0 de la secuencia, la variable `posicion` debe cambiar su valor, tomando el valor del índice actual.

Tarea 4

Objetivo: Generar habilidad en el uso del patrón de algoritmo de recorrido parcial.

Escriba los métodos de la clase Curso que resuelven los siguientes problemas, los cuales corresponden a las dos variantes del patrón de algoritmo de recorrido parcial.

Reemplazar todas las notas del curso por 0,0, hasta que aparezca la primera nota superior a 3,0.

```
public void cambiarNotasACero( )
{









}
```

Calcular el número mínimo de notas del curso necesarias para que la suma supere el valor 30, recorriéndolas desde la posición 0 en adelante. Si al sumar todas las notas no se llega a ese valor, el método debe retornar –1.

```
public int sumadasDanTreinta( )
{









}
```

5.4.3. Patrón de Doble Recorrido

El último de los patrones que vamos a ver en este capítulo es el de doble recorrido. Este patrón se utiliza como solución de aquellos problemas en los cuales, por cada elemento de la secuencia, se debe hacer un recorrido completo. Piense en el problema de encontrar la nota que aparece un mayor número de veces en el curso. La solución evidente es tomar la primera nota y hacer un recorrido completo del arreglo contando el número de veces que ésta vuelve a aparecer. Luego, haríamos lo mismo con los demás elementos del arreglo y escogeríamos al final aquélla que aparezca un mayor número de veces.

El esqueleto básico del algoritmo con el que se resuelven los problemas que siguen este patrón es el siguiente:

```
for( int indice1 = 0; indice1 < arreglo.length; indice1++ )
{
    for( int indice2 = 0; indice2 < arreglo.length; indice2++ )
    {
        <cuerpo del ciclo interno>
    }

    <cuerpo del ciclo externo>
}
```

- El ciclo de afuera está controlado por la variable " `indice1` ", mientras que el ciclo interno utiliza la variable " `indice2` ".
- Dentro del cuerpo del ciclo interno se puede hacer referencia a la variable " `indice1` ".

Las variantes y las decisiones son las mismas que identificamos en los patrones anteriores. La estrategia de solución consiste en considerar el problema como dos problemas independientes, y aplicar los patrones antes vistos, tal como se muestra en el ejemplo 8.

Ejemplo 8

Objetivo: Mostrar el uso del patrón de algoritmo de recorrido total con doble recorrido.

En este ejemplo se muestra el método de la clase Curso que retorna la nota que aparece un mayor número de veces. Para escribirlo procederemos por etapas, las cuales se describen en la parte derecha.

```
public double darNotaMasRecurrente( )
{
    double notaMasRecurrente = 0.0;

    for( int i = 0; i < notas.length; i++ )
    {
        for( int j = 0; j < notas.length; j++ )
        {

            //Por completar

        }
    }

    return notaMasRecurrente ;
}
```

- Primera etapa: armar la estructura del método a partir del esqueleto del patrón.
- Utilizamos las variables `i` y `j` para llevar los índices en cada uno de los ciclos.
- Decidimos que el resultado lo vamos a dejar en una variable llamada `notaMasRecurrente`, la cual retornamos al final del método.
- Una vez construida la base del método, identificamos los dos problemas que debemos resolver en su interior: (1) contar el número de veces que aparece en el arreglo el valor que está en la casilla i; (2) encontrar el mayor valor entre los que son calculados por el primer problema.

```
public double darNotaMasRecurrente( )
{
    double notaMasRecurrente = 0.0;

    for( int i = 0; i < notas.length; i++ )
    {
        double notaBuscada = notas[ i ];
        int contador = 0;

        for( int j = 0; j < notas.length; j++ )
        {
            if( notas[ j ] == notaBuscada )
            {
                contador++;
            }
        }

        //Por completar
    }
    return notaMasRecurrente ;
}
```

- Segunda etapa: Resolvemos el primero de los problemas identificados, usando para eso el ciclo interno.
- Para facilitar el trabajo, vamos a dejar en la variable `notaBuscada`, la nota para la cual queremos contar el número de ocurrencias. Dicha variable la inicializamos con la nota de la casilla i.
- Usamos una segunda variable llamada `contador` para acumular allí el número de veces que aparezca el valor buscado dentro del arreglo. Dicho valor será incrementado cuando `notaBuscada == notas[ j ]`.
- Al final del ciclo, en la variable contador quedará el número de veces que el valor de la casilla i aparece en todo el arreglo.

```
public double darNotaMasRecurrente( )
{

    double notaMasRecurrente = 0.0;
    int cantidadOcurrencias= 0;

    for( int i = 0; i < notas.length; i++ )
    {
        double notaBuscada = notas[ i ];
        int contador = 0;

        for( int j = 0; j < notas.length; j++ )
        {
            if( notas[ j ] == notaBuscada )
            {
                contador++;
            }
        }

        if( contador > cantidadOcurrencias)
        {
            notaMasRecurrente = notaBuscada;
            cantidadOcurrencias= contador;
        }
    }

    return notaMasRecurrente ;

}
```

- Tercera etapa: Usamos el ciclo externo para encontrar la nota que más veces aparece.
- Usamos para eso dos variables: `notaMasRecurrente` que indica la nota que hasta el momento más veces aparece, y `cantidadOcurrencias` para saber cuántas veces aparece dicha nota.
- Luego definimos el caso en el cual debemos cambiar el acumulado: si encontramos un valor que aparezca más veces que el que teníamos hasta el momento (`contador >`

`cantidadOcurrencias`) debemos actualizar los valores de nuestras variables.

En general, este patrón dice que para resolver un problema que implique un doble recorrido, primero debemos identificar los dos problemas que queremos resolver (uno con cada ciclo) y, luego, debemos tratar de resolverlos independientemente, usando los patrones de recorrido total o parcial.

Si para resolver un problema se necesita un tercer ciclo anidado, debemos escribir métodos separados que ayuden a resolver cada problema individualmente, tal como se plantea en el nivel 4, porque la solución directa es muy compleja y propensa a errores.

Tarea 5

Objetivo: Generar habilidad en el uso del patrón de algoritmo de doble recorrido.

Escriba el método de la clase Curso que resuelve el siguiente problema, que corresponde al patrón de algoritmo de doble recorrido.

Calcular una nota del curso (si hay varias que lo cumplan puede retornar cualquiera) tal que la mitad de las notas sean menores o iguales a ella.

```
public double notaMediana( )
{









}
```

6. Caso de Estudio Nº 2: Reservas en un Vuelo

Un cliente quiere que construyamos un programa para manejar las reservas de un vuelo. Se sabe que el avión tiene 50 sillas, de las cuales 8 son de clase ejecutiva y las demás de clase económica. Las sillas ejecutivas se acomodan en filas de cuatro, separadas en el medio por el corredor. Las sillas económicas se acomodan en filas de seis, tres a cada lado del corredor.

Cuando un pasajero llega a solicitar una silla, indica sus datos personales y sus preferencias con respecto a la posición de la silla en el avión. Los datos del pasajero que le interesan a la aerolínea son el nombre y la cédula. Para dar la ubicación deseada, el pasajero indica la clase y la ubicación de la silla. Esta puede ser, en el caso de las ejecutivas, ventana y pasillo, y en el de las económicas, ventana, pasillo y centro. La asignación de la silla en el avión se hace en orden de llegada, tomando en cuenta las preferencias anteriores y las disponibilidades.

La interfaz de usuario del programa a la que se llegó después de negociar con el cliente se muestra en la figura 3.6.

Fig. 3.6 Interfaz de usuario para el caso de estudio del avión

- En la parte superior del avión aparecen las 8 sillas ejecutivas.
- En la parte inferior, aparecen las 42 sillas económicas, con un corredor en la mitad.
- Se ofrecen las distintas opciones del programa a través de los botones que se pueden observar en la parte superior de la ventana.
- Cuando una silla está ocupada, ésta aparecerá indicada en el dibujo del avión con un color especial.
- Cada silla tiene asignado un número que es único. La silla 7, por ejemplo, está en primera clase, en el corredor de la segunda fila.

6.1. Comprensión de los Requerimientos

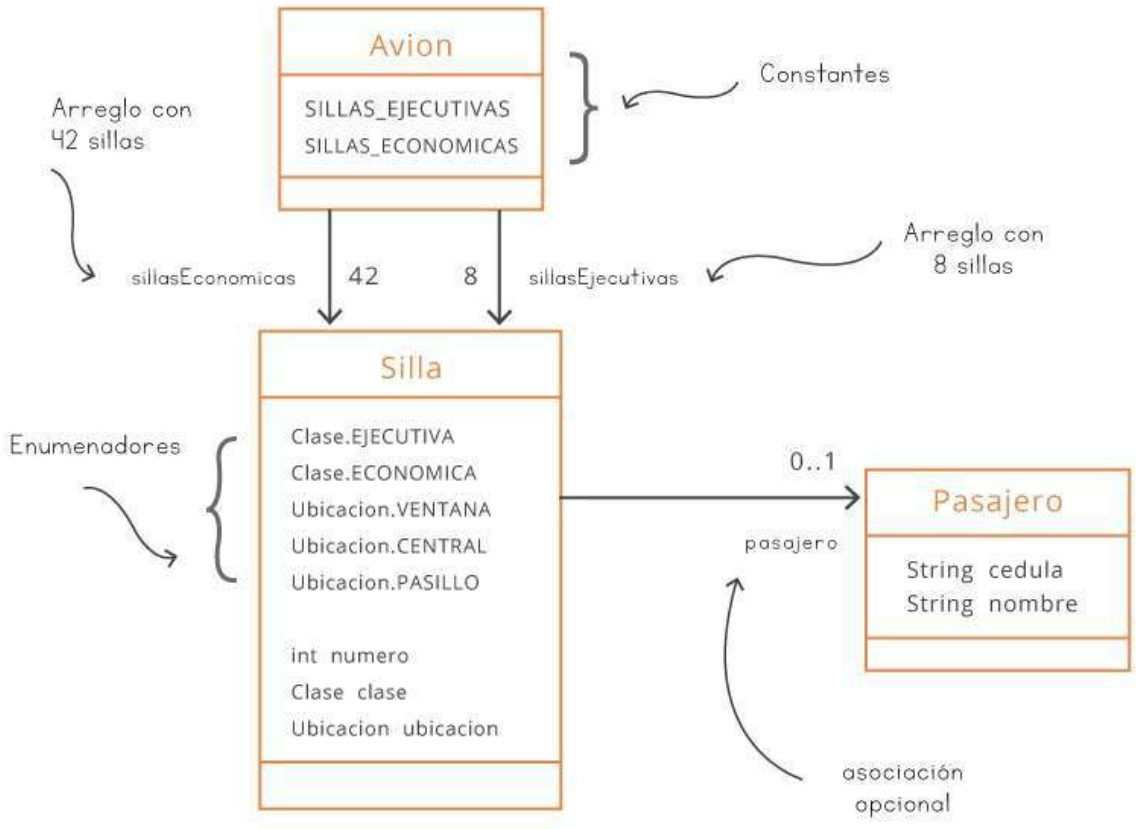
Nos vamos a concentrar en el siguiente requerimiento funcional:

Nombre	R1 - Asignar una silla a un pasajero.
Resumen	Asigna una silla a un pasajero según sus preferencias. Estas son clase (Ejecutiva o Económica) y ubicación (Ventana, Centro o Pasillo).
Entradas	(1) nombre del pasajero, (2) cédula del pasajero, (3) clase de la silla, (4) ubicación de la silla.
Resultados	Se marca como asignada una de las sillas disponibles en el avión, dependiendo de la clase y ubicación elegida. En caso de que todas las sillas estén asignadas, se muestra un mensaje de error.

6.2. Comprensión del Mundo del Problema

Podemos identificar tres entidades distintas en el mundo: avión, silla y pasajero. Lo cual nos lleva al diagrama de clases que se muestra en la figura 3.7.

Fig. 3.7 Diagrama de clases para el caso de estudio del avión



En este diagrama se puede leer lo siguiente:

- Una silla puede ser ejecutiva o económica (un enumerador con las dos constantes definidas para la posible clase de la Silla), puede estar localizada en pasillo, corredor o centro (un enumerador con tres constantes definidas para la posible ubicación de la Silla), y tiene un identificador único que es un valor numérico.
- Entre Silla y Pasajero hay una asociación opcional (0..1). Si la asociación está presente se interpreta como que la silla está ocupada y se conoce el pasajero que allí se encuentra. Si no está presente (vale null) se interpreta como que la silla está disponible.
- Un pasajero se identifica con la cédula y tiene un nombre.
- Un avión tiene 8 sillas ejecutivas (constante SILLAS_EJECUTIVAS de la clase Avion) y 42 sillas económicas (constante SILLAS_ECONOMICAS de la clase Avion). Fíjese cómo se expresa la cardinalidad de una asociación en UML.

6.3. Diseño de la Solución

Vamos a dividir el proyecto en 3 paquetes, siguiendo la arquitectura planteada en el primer nivel del libro. Los paquetes son:

```
uniandes.cupi2.avion.interfaz
uniandes.cupi2.avion.test
uniandes.cupi2.avion.mundo
```

La principal decisión de diseño del programa se refiere a la manera de representar el grupo de sillas del avión. Para esto vamos a manejar dos arreglos de objetos. Uno con 8 posiciones que tendrá los objetos de la clase Silla que representan las sillas de la clase ejecutiva, y otro arreglo de 42 posiciones con los objetos para representar las sillas económicas.

En las secciones que siguen presentaremos las distintas clases del modelo del mundo que constituyen la solución. Comenzamos por la clase más sencilla (la clase Pasajero) y terminamos por la clase que tiene la responsabilidad de manejar los grupos de atributos (la clase Avion), en donde tendremos la oportunidad de utilizar los patrones de algoritmo vistos en las secciones anteriores.

6.4. La Clase Pasajero

Tarea 6

Objetivo: Hacer la declaración en Java de la clase Pasajero.

Complete la declaración de la clase Pasajero, incluyendo sus atributos, el constructor y los métodos que retornan la cédula y el nombre. Puede guiarse por el diagrama de clases que aparece en la figura 3.7.

```
public class Pasajero
{

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









    //-----------------------------------
    // Constructor
    //-----------------------------------
    public Pasajero( String pCedula, String pNombre )
    {









    }

    //-----------------------------------
    // Métodos
    //-----------------------------------
    public String darCedula( )
    {









    }
    public String darNombre( )
    {
```

```
    }
}
```

6.5. La Clase Silla

Tarea 7

Objetivo: Completar la declaración de la clase Silla.

Complete las declaraciones de los atributos y las enumeraciones de la clase Silla y desarrolle los métodos que se le piden para esta clase.

```
public class Silla
{
    //-----------------------------------------
    // Enumeraciones
    //-----------------------------------------

    /**
      * Enumeradores para las clases de silla.
      */
     public enum Clase
     {
        /**
          * Representa la clase ejecutiva.
          */
          EJECUTIVA,

          /**
           * Representa la clase económica.
           */
          ECONOMICA
    }

    /**
      * Enumeradores para las ubicaciones de las sillas.
      */
     public enum Ubicacion
     {
        /**
          * Representa la ubicación ventana.
          */
         VENTANA,

         /**
          * Representa la ubicación centro.
          */


         /**
          * Representa la ubicación pasillo.
          */

    }

    ...
}
```

- Se declara un enumerador con dos constantes para el atributo clase de la silla (EJECUTIVA, ECONOMICA).
- Se declara un enumerador con tres constantes para representar las tres ubicaciones posibles de una silla (VENTANA, CENTRAL, PASILLO).

```
public class Silla
{
    ...

    //-----------------------------------------
    // Atributos
    //-----------------------------------------
    private int numero;
    private Clase clase;
    private Ubicacion ubicacion;
    private Pasajero pasajero;

    ...
}
```

- Se declaran en la clase cuatro atributos: (1) el número de la silla, (2) la clase de la silla, (3) su ubicación y (4) el pasajero que opcionalmente puede ocupar la silla.
- El atributo " `pasajero` " debe tener el valor `null` si no hay ningún pasajero asignado a la silla.

```
public Silla( int pNumero, Clase pClase, Ubicacion pUbicacion )
{
    numero = pNumero;
    clase = pClase;
    ubicacion = pUbicacion;
    pasajero = null;
}
```

- En el constructor se inicializan los atributos a partir de los valores que se reciben como parámetro.
- Se inicializa el atributo `pasajero` en `null` , para indicar que la silla se encuentra vacía.

```
public class Silla
{
    ...
    public void asignarPasajero( Pasajero pPasajero )
    {









    }
    ...
}
```

- Asigna la silla al pasajero "pPasajero".

```
public class Silla
{
    ...
    public void desasignarSilla ( )
    {









    }
    ...
}
```

- Quita al pasajero que se encuentra en la silla, dejándola desocupada.

```
public class Silla
{
    ...
    public boolean sillaAsignada( )
    {









    }
    ...
}
```

- Informa si la silla está ocupada.

```
public class Silla
{
    ...
    public int darNumero( )
    {









    }
    ...
}
```

- Retorna el número de la silla.

```
public class Silla
{
    ...
    public Clase darClase( )
    {









    }
    ...
}
```

- Retorna la clase de la silla.

```
public class Silla
{
    ...
    public Ubicacion darUbicacion( )
    {

    }
    ...
}
```

- Retorna la ubicación de la silla.

```
public class Silla
{
    ...
    public Pasajero darPasajero( )
    {

    }
    ...
}
```

- Retorna el pasajero de la silla.

6.6. La Clase Avion

Ejemplo 9

Objetivo: Mostrar las declaraciones y el constructor de la clase Avion.

En este ejemplo se presentan las declaraciones de los atributos y las constantes de la clase Avion, lo mismo que su método constructor.

```
public class Avion
{
    //-------------------------------------------
    // Constantes
    //-------------------------------------------
    public final static int SILLAS_EJECUTIVAS = 8;
    public final static int SILLAS_ECONOMICAS = 42;
    ...

}
```

- Con dos constantes representamos el número de sillas de cada una de las clases.

```
public class Avion
{
    ...
    //-------------------------------------------
    // Atributos
    //-------------------------------------------
    private Silla[] sillasEjecutivas;
    private Silla[] sillasEconomicas;

    ...
}
```

- La clase Avion tiene dos contenedoras de tamaño fijo de sillas: una, de 42 posiciones, con las sillas de clase económica, y otra, de 8 posiciones, con las sillas de clase ejecutiva.
- Se declaran los dos arreglos, utilizando la misma sintaxis que utilizamos en el caso de las notas del curso.
- La única diferencia es que, en lugar de contener valores de tipo simple, van a contener objetos de la clase Silla.

A continuación aparece un fragmento del constructor de la clase. En las primeras dos instrucciones del constructor, creamos los arreglos, informando el número de casillas que deben contener. Para eso usamos las constantes definidas en la clase.

Después de haber reservado el espacio para los dos arreglos, procedemos a crear los objetos que representan cada una de las sillas del avión y los vamos poniendo en la respectiva casilla.

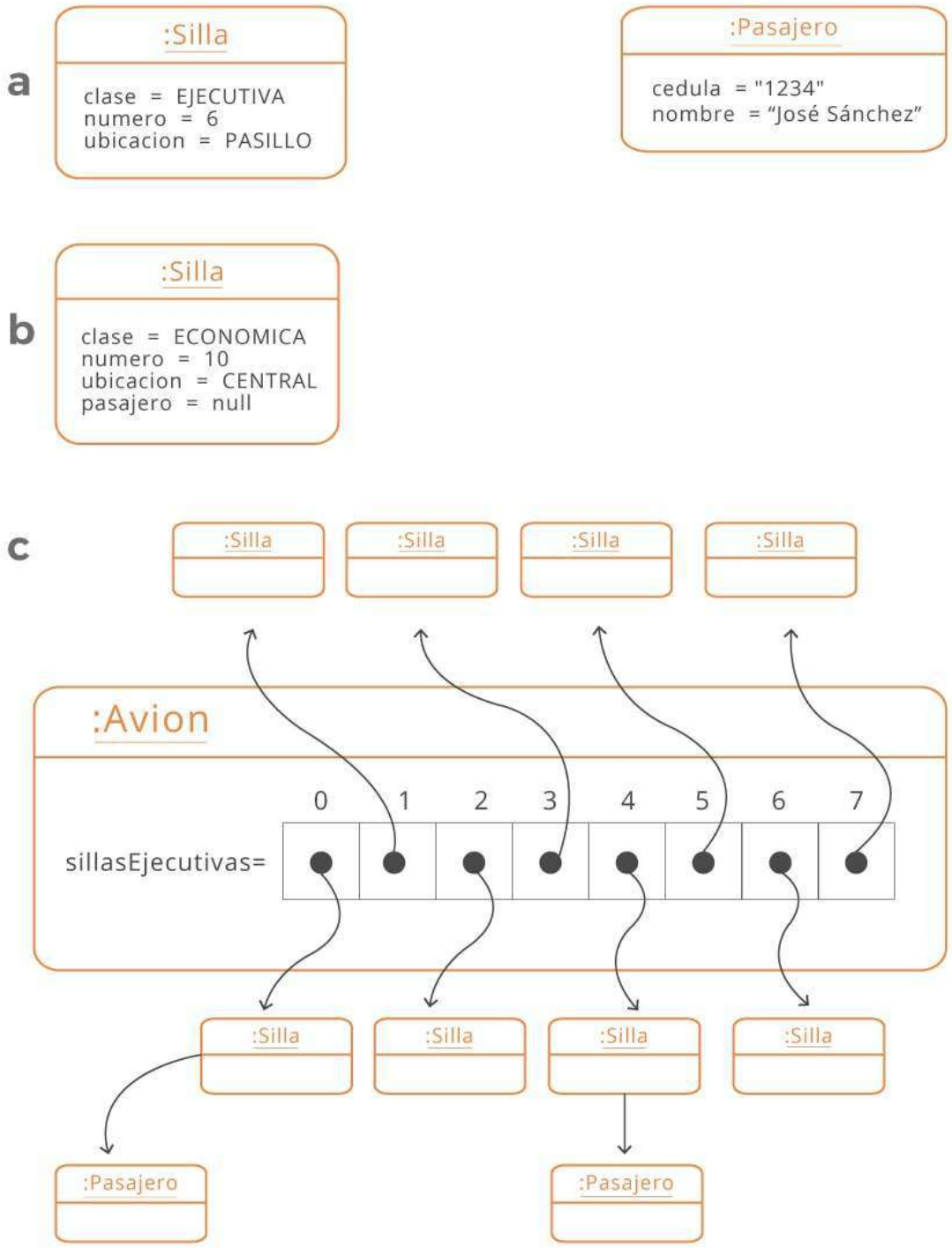
Esta inicialización se podría haber hecho con varios ciclos, pero el código resultaría un poco difícil de explicar.

```
public Avion( )
{
    sillasEjecutivas = new Silla[ SILLAS_EJECUTIVAS ];
    sillasEconomicas = new Silla[ SILLAS_ECONOMICAS ];

    // Creación de las sillas de clase ejecutiva
    sillasEjecutivas[ 0 ] = new Silla( 1, Clase.EJECUTIVA, Ubicacion.VENTANA );
    sillasEjecutivas[ 1 ] = new Silla( 2, Clase.EJECUTIVA, Ubicacion.PASILLO );
    sillasEjecutivas[ 2 ] = new Silla( 3, Clase.EJECUTIVA, Ubicacion.PASILLO );
    sillasEjecutivas[ 3 ] = new Silla( 4, Clase.EJECUTIVA, Ubicacion.VENTANA );
    sillasEjecutivas[ 4 ] = new Silla( 5, Clase.EJECUTIVA, Ubicacion.VENTANA );
    sillasEjecutivas[ 5 ] = new Silla( 6, Clase.EJECUTIVA, Ubicacion.PASILLO );
    sillasEjecutivas[ 6 ] = new Silla( 7, Clase.EJECUTIVA, Ubicacion.PASILLO );
    sillasEjecutivas[ 7 ] = new Silla( 8, Clase.EJECUTIVA, Ubicacion.VENTANA );

    // Creación de las sillas de clase económica
    sillasEconomicas[ 0 ] = new Silla( 9, Clase.ECONOMICA, Ubicacion.VENTANA );
    sillasEconomicas[ 1 ] = new Silla( 10, Clase.ECONOMICA, Ubicacion.CENTRAL );
    sillasEconomicas[ 2 ] = new Silla( 11, Clase.ECONOMICA, Ubicacion.PASILLO );
    ...
}
```

Ya con las declaraciones hechas y con el constructor implementado, estamos listos para comenzar a desarrollar los distintos métodos de la clase. Pero antes de empezar, queremos hablar un poco de las diferencias que existen entre un arreglo de valores de tipo simple (como el del caso de estudio de las notas) y un arreglo de objetos (como el del caso del avión).

Para empezar, en la figura 3.8a se muestra una instancia de la clase Silla ocupada por un pasajero. En la figura 3.8b se muestra un objeto de la clase Silla que se encuentra vacía. En la figura 3.8c se ilustra un posible contenido del arreglo de sillas ejecutivas (usando un diagrama de objetos).

Fig. 3.8 Ejemplo del contenido del arreglo de sillas ejecutivas



- **Figura 3.8a:** en la silla de primera clase número 6, situada en el corredor, está sentado el Sr. José Sánchez con cédula No. 1234.
- **Figura 3.8b**: la silla de clase económica número 10, situada en el centro, está desocupada.

- **Figura 3.8c:** cada casilla del arreglo tiene un objeto de la clase Silla (incluso si la silla está desocupada).
- Las sillas ocupadas tienen una asociación con el objeto que representa al pasajero que la ocupa. * En los arreglos de objetos se almacenan referencias a los objetos, en lugar de los objetos mismos.
- Con la sintaxis `sillasEjecutivas[x]` podemos hacer referencia al objeto de la clase Silla que se encuentra en la casilla x.
- Si queremos llegar hasta el pasajero que se encuentra en alguna parte del avión, debemos siempre pasar por la silla que ocupa. No hay otra manera de "navegar" hasta él.

Ya teniendo una visualización del diagrama de objetos del caso de estudio, es más fácil contestar las siguientes preguntas:

¿Cómo se llama un método de un objeto que está en un arreglo?	**Por ejemplo, dentro de la clase Avion, para preguntar si la silla que está en la posición 0 del arreglo de sillas ejecutivas está ocupada, se utiliza la sintaxis: `sillasEjecutivas[0].sillaAsignada()` .Esta sintaxis es sólo una extensión de la sintaxis que ya veníamos utilizando. Lo único que se debe tener en cuenta es que cada vez que hacemos referencia a una casilla, estamos hablando de un objeto, más que de un valor simple.**
¿Los objetos que están en un arreglo se pueden guardar en una variable?	Tanto las variables como las casillas de los arreglos guardan únicamente referencias a los objetos. Si se hace la siguiente asignación: `Silla sillaTemporal = sillasEjecutivas[0];` tanto la variable `sillaTemporal` como la casilla 0 del arreglo estarán haciendo referencia al mismo objeto. Debe quedar claro que el objeto no se duplica, sino que ambos nombres hacen referencia al mismo objeto.
¿Qué pasa con el objeto que está siendo referenciado desde una casilla si asigno `null` a esa posición del arreglo?	Si guardó una referencia a ese objeto en algún otro lado, puede seguir usando el objeto a través de dicha referencia. Si no guardó una referencia en ningún lado, el recolector de basura de Java detecta que ya no lo está usando y recupera la memoria que el objeto estaba utilizando. ¡Adiós objeto!

Ejemplo 10

Objetivo: Mostrar la sintaxis que se usa para manipular arreglos de objetos.

En este ejemplo se muestra el código de un método de la clase Avion que permite eliminar todas las reservas del avión. No forma parte de los requerimientos funcionales, pero nos va a permitir mostrar una aplicación del patrón de recorrido total.

```
public void eliminarReservas( )
{
    for( int i = 0; i < SILLAS_EJECUTIVAS; i++ )
    {
        sillasEjecutivas[ i ].desasignarSilla( );
    }

    for( int i = 0; indice < SILLAS_ECONOMICAS; i++ )
    {
        sillasEconomicas[ i ].desasignarSilla( );
    }
}
```

- Este método elimina todas las reservas que hay en el avión.
- Note que podemos utilizar la misma variable como índice en los dos ciclos. La razón es que en la instrucción `for`, al terminar de ejecutar el ciclo, se destruyen las variables declaradas dentro de él y, por esta razón, podemos volver a utilizar el mismo nombre para la variable del segundo ciclo.
- El método utiliza el patrón de recorrido total dos veces, una por cada uno de los arreglos del avión.

Ya vimos toda la teoría concerniente al manejo de los arreglos (estructuras contenedoras de tamaño fijo). Lo que sigue es aplicar los patrones de algoritmo que vimos unas secciones atrás, para implementar los métodos de la clase Avion.

Tarea 8

Objetivo: Desarrollar los métodos de la clase Avión que nos permitan implementar los requerimientos funcionales del caso de estudio.

Para cada uno de los problemas que se plantean a continuación, escriba el método que lo resuelve. No olvide identificar primero el patrón de algoritmo que se necesita y usar las guías que se dieron en secciones anteriores.

Calcular el número de sillas ejecutivas ocupadas en el avión:

```
public int contarSillasEjecutivasOcupadas( )
{









}
```

Localizar la silla en la que se encuentra el pasajero identificado con la cédula que se entrega como parámetro. Si no hay ningún pasajero en clase ejecutiva con esa cédula, el método retorna `null` .

```
public Silla buscarPasajeroEjecutivo( String pCedula )
{









}
```

Localizar una silla económica disponible, en una localización dada (ventana, centro o pasillo). Si no existe ninguna, el método retorna `null` :

```
public Silla buscarSillaEconomicaLibre( Ubicacion pUbicacion )
{









}
```

Asignar al pasajero que se recibe como parámetro una silla en clase económica que esté libre (en la ubicación pedida). Si el proceso tiene éxito, el método retorna verdadero. En caso contrario, retorna falso:

```
public boolean asignarSillaEconomica( Ubicacion pUbicacion, Pasajero pPasajero )
{









}
```

Anular la reserva en clase ejecutiva que tenía el pasajero con la cédula dada. Retorna verdadero
si el proceso tiene éxito:

```
public boolean anularReservaEjecutivo( String pCedula )
{









}
```

Contar el número de puestos disponibles en una ventana, en la zona económica del avión:

```
public int contarVentanasEconomica( )
{









}
```

Informar si en la zona económica del avión hay dos personas que se llamen igual. Patrón de doble
recorrido:

```
public boolean hayDosHomonimosEconomica( )
{









}
```

6.7. La instrucción for-each

El **esqueleto del patrón** de recorrido total también puede definirse con la instrucción **for-each**, la cual es una variación de la instrucción for que se puede resumir en el siguiente fragmento de código:

```
for( NombreClase elemento: arreglo )
{
    <cuerpo>
}
```

La instrucción for-each permite recorrer todos los elementos de un arreglo. De esta manera, para cada objeto existente en el arreglo, se ejecutan las instrucciones que se encuentran en el cuerpo del ciclo. En cada iteración, la variable elemento va a referenciar al objeto actual,

permitiendo que se hagan las operaciones necesarias sobre este. Cabe resaltar que en el for-each no es necesario utilizar un índice, ya que la instrucción se encarga de pasar por cada uno de los elementos de forma automática. Es por esto que la instrucción for-each se utiliza principalmente en problemas que requieran un recorrido sobre todos los elementos del arreglo (recorrido total).

Ejemplo 11

Objetivo: Mostrar la sintaxis que se usa para la instrucción for-each.

En este ejemplo se muestra el código de un método de la clase Avion, el cual permite contar la cantidad de sillas económicas ocupadas, con el fin mostrar una aplicación del patrón de recorrido total utilizando la instrucción for-each. Si no hay ninguna silla económica ocupada, el método retorna cero.

A continuación se muestra el método utilizando la instrucción for:

```
public int contarSillasEconomicasOcupadas( )
{
    int contador = 0;
    Silla silla = null;
    for( int i = 0; i < SILLAS_ECONOMICAS; i++ )
    {
        silla = sillasEconomicas[ i ];
        if( silla.sillaAsignada( ) )
        {
            contador++;
        }
    }
    return contador;
}
```

La implementación del método utilizando la instrucción for-each es la siguiente:

```
public int contarSillasEconomicasOcupadas( )
{
    int contador = 0;
    for( Silla sillaEconomica : sillasEconomicas )
    {
        if( sillaEconomica.sillaAsignada( ) )
        {
            contador++;
        }
    }
    return contador;
}
```

Tarea 9

Objetivo: Desarrollar los métodos de la clase Avión que nos permitan implementar los requerimientos funcionales del caso de estudio utilizando la instrucción for-each.

Para cada uno de los problemas que se plantean a continuación, escriba el método que lo resuelve. En todos los casos son problemas que requieren un recorrido total y que se deben resolver utilizando la instrucción for-each.

Calcular el número de sillas económicas libres en el avión:

```
public int contarSillasEconomicasLibres( )
{









}
```

Contar el número de puestos disponibles en el pasillo, en la zona ejecutiva del avión:

```
public int contarPasilloEjecutivas( )
{









}
```

Desocupar avión. Se encarga de desocupar todas las sillas del avíon:

```
public void desocuparAvion( )
{









}
```

7. Caso de Estudio Nº 3: Tienda de Libros

Se quiere construir una aplicación que permita administrar una tienda de libros. La tienda tiene un catálogo de libros, que son los libros que desea poner a la venta. La aplicación permite abastecer la tienda con ejemplares de los libros del catálogo y venderlos. Adicionalmente permite saber cuánto dinero se tiene en caja, empezando con una inversión inicial de $1.000.000.

De cada libro se conoce:

- ISBN. Identificador del libro. No pueden existir dos libros en la tienda con el mismo ISBN.
- Título. El nombre del libro.
- Imagen. La imagen del libro.
- Precio de compra: Valor pagado por la compra de cada ejemplar en la tienda.
- Precio de venta: Valor por el cual se vende cada ejemplar del libro.
- Cantidad actual. Cantidad actual de ejemplares que tiene la tienda. Solo puede ser modificada mediante la venta o el abastecimiento.

Adicionalmente, de cada libro se conocen todas las transacciones que se han realizado sobre él. De cada transacción se conoce:

- El tipo de transacción. Puede ser venta o abastecimiento.
- La fecha de realización.
- La cantidad de ejemplares incluidos en la transacción.

El abastecimiento de libros permite aumentar la cantidad actual de ejemplares del libro y registrar una transacción de tipo abastecimiento.

La venta de libros permite disminuir la cantidad actual de ejemplares del libro y registrar una transacción de venta. Esta transacción solo se podrá realizar si la cantidad actual de ejemplares es mayor a la cantidad que se quiere vender.

En la figura 3.9 aparece la interfaz de usuario que se tiene prevista para el programa que se va a construir.

Fig. 3.9 Interfaz de usuario de la tienda de libros

- La interfaz está dividida en cuatro zonas: una para mostrar el dinero que hay en la caja, una para que el usuario pueda ver el listado de libros disponibles en el catálogo (donde también puede registrar nuevos libros), una para mostrar la información de un libro del catálogo, y una para las búsquedas y consultas realizadas sobre el catálogo de libros.
- En la imagen del ejemplo, aparecen tres libros en el catálogo. Para agregar libros a la tienda, se usa el botón Registrar.
- Al abastecimiento de libro se hace a través del botón Abastecer, la venta de libros a través del botón Vender y la eliminación de un libro a través del botón Eliminar.
- En la zona de consultas y búsquedas se puede buscar un libro por ISBN o título, y consultar el libro más económico, el más costoso y el más vendido.

7.1. Comprensión de los Requerimientos

Los requerimientos funcionales de este caso de estudio son 10:

1. Registrar un libro en el catálogo.
2. Eliminar un libro del catálogo.
3. Buscar un libro por título.
4. Buscar un libro por ISBN.
5. Abastecer ejemplares de un libro.
6. Vender ejemplares de un libro.
7. Calcular la cantidad de transacciones de abastecimiento de un libro particular.
8. Buscar el libro más costoso.
9. Buscar el libro menos costoso.
10. Buscar el libro más vendido.

Tarea 10

Objetivo: Entender el problema del caso de estudio.

Lea detenidamente el enunciado del caso de estudio y complete la documentación de los primeros tres requerimientos funcionales.

Requerimiento funcional 1

Nombre	R1 - Registrar un libro en el catálogo.
Resumen	Registra un libro en el catálogo con su título, código ISBN, precio de compra y precio de venta. La cantidad actual de ejemplares en el momento de registro es cero y el libro se crea sin transacciones registradas. El resultado es el nuevo libro creado en caso de que si se haya podido registrar, en caso contrario, el resultado debe ser es nulo.
Entradas	(1) título del libro, (2) ISBN del libro, (3) precio de compra del libro, (4) precio de venta del libro,(5) imagen del libro.
Resultado	El catálogo ha sido actualizado y contiene el nuevo libro.

Requerimiento funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

7.2. Comprensión del Mundo del Problema

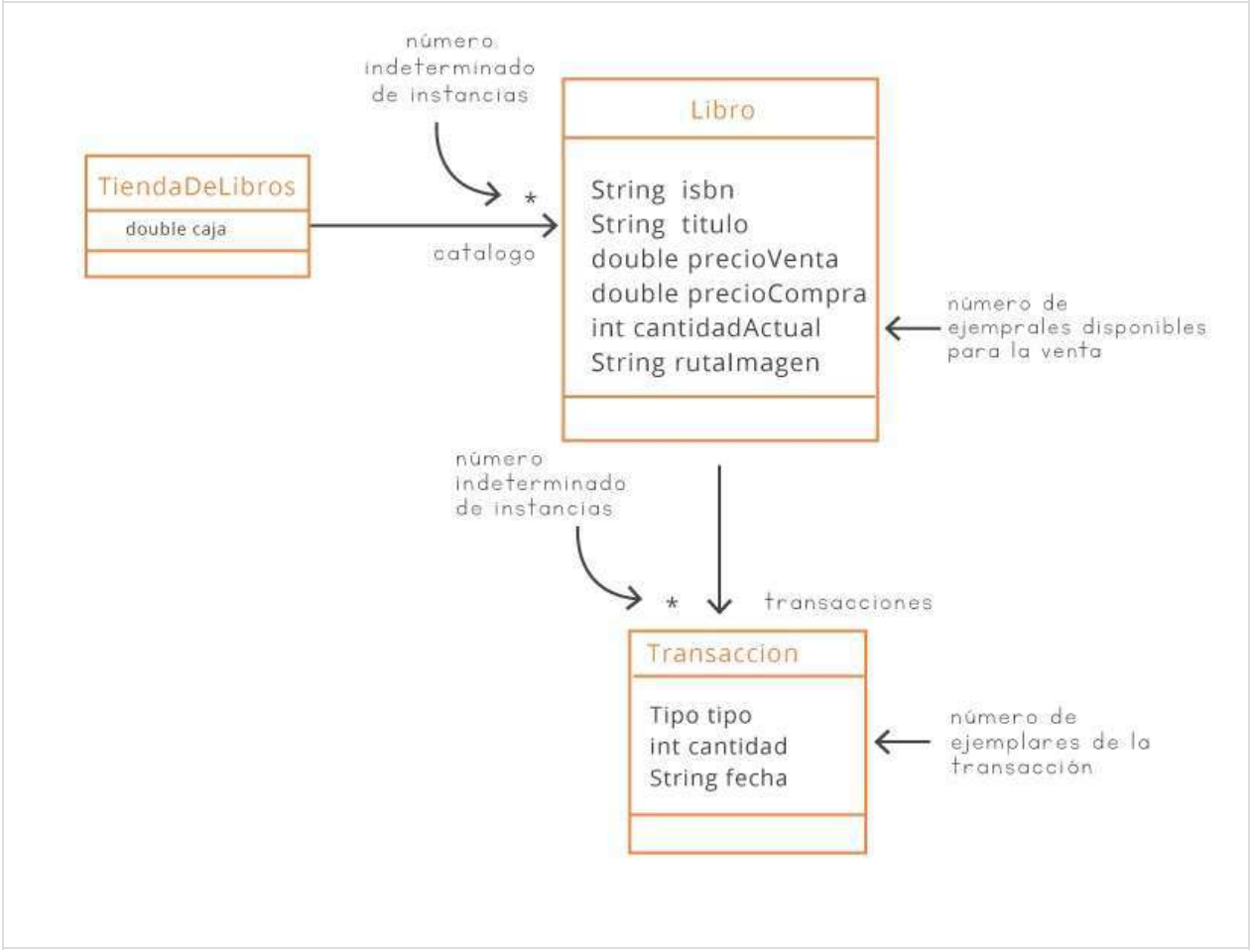
En el mundo del problema podemos identificar tres entidades (ver figura 3.10):

- La tienda de libros (clase TiendaDeLibros)
- Un libro (clase Libro)
- Una transacción (clase Transaccion)

Todas las características de las entidades identificadas en el modelo conceptual se pueden modelar con los elementos que hemos visto hasta ahora en el libro, con excepción del grupo de libros del catálogo y el listado de transacciones de un libro. La dificultad que tenemos es que no podemos predecir la cardinalidad de dicho grupo de elementos y, por esta razón, el modelado con arreglos puede no ser el más adecuado.

¿En qué se diferencia del caso del avión? La diferencia radica en que el avión tiene unas dimensiones predefinidas (42 sillas en clase económica y 8 en clase ejecutiva) que no van a cambiar durante la ejecución del programa (no existe un requerimiento de agregar una silla al avión). En el caso de la tienda de libros, se plantea que el catálogo puede tener cualquier cantidad de libros y que un libro puede tener cualquier cantidad de transacciones. Si usáramos arreglos para representar dicha información, ¿de qué dimensión deberíamos crearlos? ¿Qué hacemos si se llena el arreglo de libros del catálogo?

Fig. 3.10 Modelo conceptual para el caso de estudio de la tienda de libros



La solución a ese problema será el tema de esta parte final del nivel, en la cual presentamos las contenedoras de tamaño variable, la manera en que se usan a nivel de modelado del mundo y la forma en que se incorporan en los programas escritos en Java.

Por ahora démosle una mirada al diagrama de clases de la figura 3.10 y recorramos cada una de las entidades identificadas:

- Una tienda de libros tiene un catálogo (así se llama la asociación), que corresponde a un grupo de longitud indefinida de libros (representado por el *).
- Un libro tiene cinco atributos: un título, un ISBN, un precio de compra, un precio de venta y una imagen.
- Un libro tiene un grupo de transacciones (así se llama la asociación) de longitud

indefinida. Cada transacción es de tipo abastecimiento o venta.

- Cada transacción tiene el tipo (abastecimiento o venta), la cantidad de ejemplares y la fecha.

8. Contenedoras de Tamaño Variable

En muchos problemas necesitamos representar grupos de atributos para los cuales no conocemos su tamaño máximo. En el caso de la tienda de libros, por ejemplo, el catálogo podría tener 100 ó 10.000 libros distintos. Para poder representar y manejar ese tipo de características, tenemos las contenedoras de tamaño variable.

En el diagrama de clases de UML, las asociaciones que tienen dicha característica se representan con una cardinalidad indefinida, usando los símbolos * o 0..N, tal como se mostró en la figura 3.10.

Para implementarlas en Java, no existen elementos en el lenguaje como los arreglos, sino que es necesario utilizar algunas clases que fueron construidas con este fin.

¿Cuál es la diferencia? La principal diferencia es que para manipular las contenedoras de tamaño variable debemos utilizar la misma sintaxis que utilizamos para manejar cualquier otra clase. No hay una sintaxis especial para obtener un elemento (como `[ ]` en los arreglos), ni contamos con operadores especiales (`length`).

En Java existen varias clases que nos permiten manejar contenedoras de tamaño variable, todas ellas disponibles en el paquete llamado `java.util` . En este libro vamos a utilizar la clase ArrayList, que es eficiente e incluye toda la funcionalidad necesaria para manipular grupos de objetos. La mayor restricción que vamos a encontrar es que no permite manejar grupos de atributos de tipo simple, sino únicamente grupos de objetos. En este nivel vamos a estudiar únicamente los principales métodos de esa clase, aquéllos que ofrecen las funcionalidades típicas para manejar esta clase de estructuras. Si desea conocer la descripción de todos los métodos disponibles, lo invitamos a consultar la documentación que aparece en el sitio web del lenguaje Java.

> Por simplicidad, vamos a llamar vector a cualquier implementación de una estructura contenedora de tamaño variable.

Al igual que con los arreglos, comenzamos ahora el recorrido para estudiar la manera de declarar un atributo de la clase ArrayList, la manera de tener acceso a sus elementos, la forma de modificarlo, etc. Para esto utilizaremos el caso de estudio de la tienda de libros.

8.1. Declaración de un Vector

Puesto que un vector es una clase común y corriente de Java, la sintaxis para declararlo es la misma que hemos utilizado en los niveles anteriores. En el ejemplo 11 se explican las declaraciones de las clases TiendaLibros y Libro.

Ejemplo 12

Objetivo: Mostrar la sintaxis usada en Java para declarar un vector.

En este ejemplo se muestran las declaraciones de las clases TiendaLibros y Libro, las cuales contienen atributos de tipo vector.

```
package uniandes.cupi2.tiendadelibros.mundo;

import java.util.*;

public class TiendaDeLibros
{
    //---------------------------------
    // Atributos
    //---------------------------------
    private ArrayList<Libro> catalogo;
    private double caja;
    ...
}
```

- Para poder usar la clase ArrayList es necesario importar su declaración, indicando el paquete en el que ésta se encuentra (`java.util`). Esto se hace con la instrucción `import` de Java.
- Dicha instrucción va después de la declaración del paquete de la clase y antes de su encabezado.
- En la clase TiendaDeLibros se declaran dos atributos: el catálogo, que es un vector, y el dinero que hay en la caja, que es de tipo double.
- Al declarar un vector, se indica el tipo de objetos que se van a guardar en él, usando la sintáxis `<NombreDeLaClase>` . En el caso del catálogo, se indica que el catálogo es un vector de libros.

```
package uniandes.cupi2.tiendadelibros.mundo;

import java.util.*;

public class Libro
{
    //---------------------------------
    // Atributos
    //---------------------------------
    private ArrayList<Transaccion> transacciones;
    ...
}
```

- En la clase Libro se declara el grupo de transacciones como un vector.
- Se debe de nuevo importar el paquete en donde se encuentra la clase ArrayList, usando la instrucción `import`.
- Fíjese que la declaración de un vector utiliza la misma sintaxis que se usa para declarar cualquier otro atributo de la clase.

8.2 Inicialización y Tamaño de un Vector

En el constructor es necesario inicializar los vectores, al igual que hacemos con todos los demás atributos de una clase. Hay dos diferencias entre crear un arreglo y crear un vector:

- En los vectores se utiliza la misma sintaxis de creación de cualquier otro objeto (`new ArrayList<NombreDeLaClase>()`) teniendo que agregar el nombre de las clase a la que pertenecen los objetos que se van a agregar al vector, mientras que los arreglos utilizan los `[ ]` para indicar el tamaño (`new NombreDeLaClase[TAMANIO]`).
- En los vectores no es necesario definir el número de elementos que va a tener, mientras que en los arreglos es indispensable hacerlo.

Ejemplo 13

Objetivo: Mostrar la manera de inicializar un vector.

En este ejemplo se muestran los métodos constructores de las clases TiendaLibros y Libro, las cuales contienen atributos de tipo vector.

```
public TiendaDeLibros( )
{
    catalogo = new ArrayList<Libro>( );
}
```

- No hay necesidad de especificar el número de elementos que el vector va a contener.

```
public Libro( )
{
    transacciones = new ArrayList<Transaccion>( );
}
```

- Al crear un vector se reserva un espacio variable para almacenar los elementos que vayan apareciendo. Inicialmente hay 0 objetos en él.

Dos métodos de la clase ArrayList nos permiten conocer el número de elementos que en un momento dado hay en un vector:

- `isEmpty( )` : es un método que retorna verdadero si el vector no tiene elementos y falso en caso contrario. Por ejemplo, en la clase Libro, después de llamar el constructor, la invocación del método `transacciones.isEmpty( )` retorna verdadero.
- `size( )` : es un método que retorna el número de elementos que hay en el vector. Para el mismo caso planteado anteriormente, `transacciones.size( )` es igual a 0.

Si adaptamos el esqueleto de los patrones de algoritmo para el manejo de vectores, lo único que va a cambiar es la condición para continuar en el ciclo. En lugar de usar la operación `length` de los arreglos, debemos utilizar el método size( ) de los vectores, tal como se muestra en el siguiente fragmento de método de la clase TiendaDeLibros.

```
public void esqueleto( )
{
    for( int i = 0; i < catalogo.size(); i++ )
    {
        // cuerpo del ciclo
    }
}
```

Las instrucción for-each para los vectores funciona de forma similar que para los arreglos como se muestra en el siguiente fragmento de código.

```
public void esqueleto( )
{
    for( Libro libro: catalogo )
    {
        // cuerpo del ciclo
    }
}
```

Las posiciones en los vectores, al igual que en los arreglos, comienzan en 0.

La condición para continuar en el ciclo se escribe utilizando el método `size( )` de la clase ArrayList, en lugar del operador `length` de los arreglos. Note que los paréntesis son necesarios.

La siguiente tabla ilustra el uso de los métodos de manejo del tamaño de un vector en el caso de estudio:

Clase	Expresión	Interpretación
TiendaDeLibros	`catalogo.size( )`	Número de libros disponibles en el catálogo.
TiendaDeLibros	`catalogo.size( ) == 10`	¿Hay 10 libros en el catálogo?
TiendaDeLibros	`catalogo.isEmpty( )`	¿Está vacío el catálogo?
Libro	`transacciones.size( )`	Número de transacciones del libro.

8.3. Acceso a los Elementos de un Vector

Los elementos de un vector se referencian por su posición en la estructura, comenzando en la posición cero. Para esto se utiliza el método get(pos), que recibe como parámetro la posición del elemento que queremos recuperar y nos retorna el objeto que allí se encuentra.

Ejemplo 14

Objetivo: Ilustrar el uso del método que nos permite recuperar un objeto de un vector.

En este ejemplo se ilustra el uso del método de acceso a los elementos de un vector. Vamos a suponer que en la clase Libro existe el método darPrecioVenta( ), que retorna el precio de venta del libro. Este método suma el precio de venta de todos los libros del catálogo.

```
public int inventario( )
{
    int sumaPrecios = 0;
    for( int i = 0; i < catalogo.size( ); i++ )
    {
        Libro libro = catalogo.get( i );
        sumaPrecios += libro.darPrecioVenta( );
    }
    return sumaPrecios;
}
```

- Con la instrucción `get( i )` de los vectores se puede acceder a la referencia del objeto del vector que se encuentra en la posición i.
- Es una buena idea guardar siempre en una variable temporal la referencia al objeto recuperado, para simplificar el código.

> Cuando dentro de un método tratamos de acceder una posición en un vector con un índice no válido (menor que 0 o mayor o igual que el número de objetos que en ese momento se encuentren en el vector), obtenemos el error de ejecución: *java.lang.IndexOutOfBoundsException*.

Recuerde que al utilizar el método get(pos), lo único que estamos obteniendo es una referencia al objeto que se encuentra referenciado desde la posición pos del vector. No se hace ninguna copia del objeto, ni desplaza el objeto a ningún lado.

8.4. Agregar Elementos a un Vector

Los elementos de un vector se pueden agregar al final del mismo o insertar en una posición específica. Los métodos para hacerlo son los siguientes:

- **add(objeto)**: es un método que permite agregar al final del vector el objeto que se pasa como parámetro. No importa cuántos elementos haya en el vector, el método siempre sabe cómo buscar espacio para agregar uno más.
- **add(indice, objeto)**: es un método que permite insertar un objeto en la posición indicada por el índice especificado como parámetro. Esta operación hace que el elemento que se encontraba en esa posición se desplace hacia la posición siguiente, lo mismo que el resto de los objetos en la estructura.

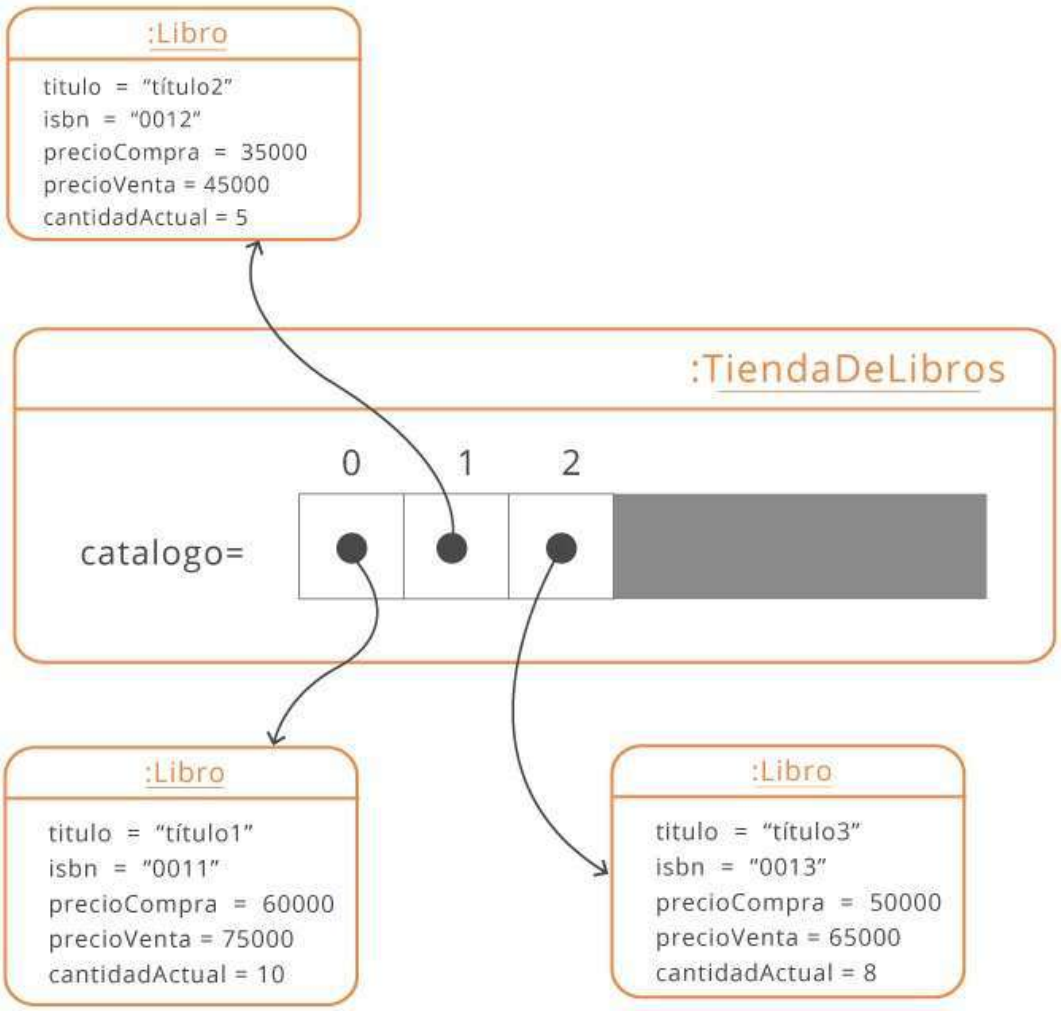
Ejemplo 15

Objetivo: Mostrar el uso del método que agrega objetos a un vector.

En este ejemplo se ilustra el uso de los métodos que permiten agregar elementos a un vector. El siguiente es un método de la clase TiendaDeLibros que añade tres libros al catálogo.

```
public void agregarTresLibros( )
{
    Libro lb1 = new Libro( "título1", "0011", 1000, 1200, "Ruta Imagen 1" );
    Libro lb2 = new Libro( "título2", "0012", 2000, 2400, "Ruta Imagen 2" );
    Libro lb3 = new Libro( "título3", "0013", 3000, 3600, "Ruta Imagen 3" );

    catalogo.add( lb2 );
    catalogo.add( lb3 );
    catalogo.add( 0, lb1 );
}
```

- En el método se crean inicialmente los tres libros. Luego se agrega el segundo de los libros (`lb2`). Como el vector estaba vacío, el nuevo elemento queda en la posición 0 del catálogo. Después se añade el tercer libro (`lb3`), que queda en la posición 1. Finalmente se inserta el primer libro (`lb1`) en la posición 0, lo que desplaza el libro 2 a la posición 1 y el libro 3 a la posición 2.



- En este diagrama de objetos se puede apreciar el estado del catálogo después de ejecutar este método.
- Si usamos el método size( ) para el catálogo, debe responder 3.

- En el dibujo dejamos en gris las casillas posteriores a la 2, para indicar que el vector las puede ocupar cuando las necesite.

8.5. Reemplazar un Elemento en un Vector

Cuando se quiere reemplazar un objeto por otro en un vector, se utiliza el método set( ), que recibe como parámetros el índice del elemento que se debe reemplazar y el objeto que debe tomar ahora esa posición.

Este método es muy útil para ordenar un vector o para clasificar bajo algún concepto los elementos que allí se encuentran. En el ejemplo 15 aparece un método de la clase TiendaDeLibros que permite intercambiar dos libros del catálogo, dadas sus posiciones en el vector que los contiene.

Ejemplo 16

Objetivo: Mostrar la manera de reemplazar un objeto en un vector.

En este ejemplo se ilustra el uso del método que reemplaza un objeto por otro en un vector. El método de la clase TiendaLibros recibe las posiciones en el catálogo de los libros que debe intercambiar.

```
public void intercambiar( int pPosicion1, int pPosicion2 )
{
    Libro libro1 = catalogo.get( pPosicion1 );
    Libro libro2 = catalogo.get( pPosicion2 );

    catalogo.set( pPosicion1 , libro2 );
    catalogo.set( pPosicion2 , libro1 );
}
```

Cuando se intercambian los elementos en cualquier estructura es indispensable guardar al menos uno de ellos en una variable temporal. En este método decidimos usar dos variables por claridad.
En este método suponemos que las dos posiciones dadas son válidas (que son posiciones entre 0 y `catalogo.size( ) -1`).

El método set( ) no hace sino reemplazar la referencia al objeto que se encuentra almacenada en la casilla. Se puede ver simplemente como la manera de asignar un nuevo valor a una casilla.
La referencia que allí se encontraba se pierde, a menos que haya sido guardada en algún otro lugar.

8.6. Eliminar un Elemento de un Vector

De la misma manera que es posible agregar elementos a un vector, también es posible eliminarlos. Piense en el caso de la tienda de libros. Si el usuario decidiera eliminar un libro del catálogo la tienda, nosotros en el programa debemos quitarlo del respectivo vector el objeto que lo representaba. Después de eliminada la referencia a un objeto, esta posición es ocupada por el elemento que se encontraba después de él en el vector.

El método de la clase ArrayList que se usa para eliminar un elemento se llama remove( ) y recibe como parámetro la posición del elemento que se quiere eliminar (un valor entre 0 y el número de elementos menos 1). Al usar esta operación, se debe tener en cuenta que el tamaño de la estructura disminuye en 1, por lo que se debe tener cuidado en el momento de definir la condición de continuación de los ciclos.

Es importante recalcar que el hecho de quitar un objeto de un vector no implica necesariamente su destrucción. Lo único que estamos haciendo es eliminando una referencia al objeto. Si queremos mantenerlo vivo, basta con guardar su referencia en otro lado, por ejemplo en una variable.

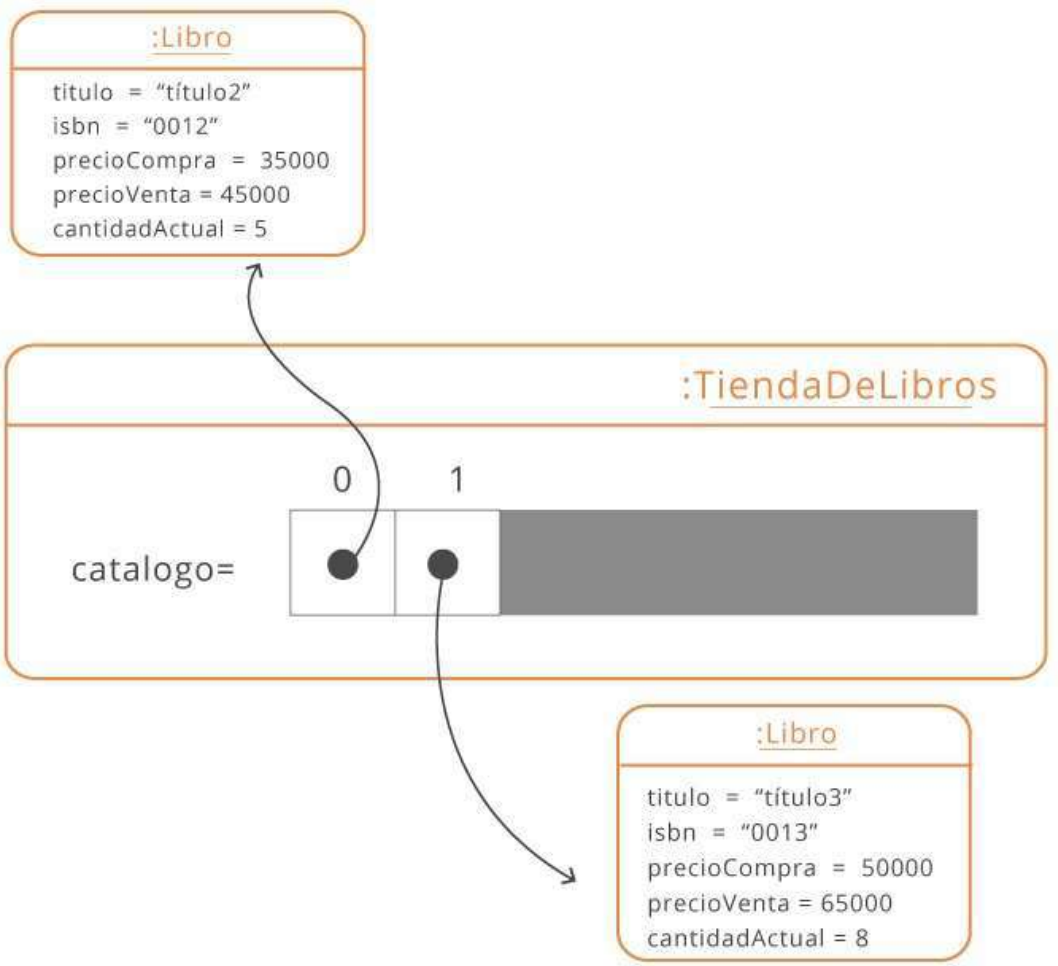
Ejemplo 17

Objetivo: Mostrar la manera de utilizar el método que elimina un objeto de un vector.

En este ejemplo presentamos un método de la clase TiendaDeLibros que elimina el primer libro del catálogo. Ilustramos el resultado usando el diagrama de objetos del ejemplo 14.

```
public void eliminarPrimerLibro( )
{
    catalogo.remove( 0 );
}
```

Este método elimina del catálogo la referencia al primer libro de la tienda. Después de su ejecución, todos los libros se mueven una posición hacia la izquierda en el catálogo.



- Si ejecutamos este método sobre el diagrama de objetos del ejemplo 14, obtenemos el diagrama que aparece en esta figura.
- El libro que estaba en la posición 1 pasa a la posición 0, y el libro de la posición 2 pasa a la posición 1.
- Ahora `catalogo.size( )` es igual a 2.

Ya que hemos terminado de ver los principales métodos con los que contamos para manejar los elementos de un vector, vamos a comenzar a escribir los métodos de la clase del caso de estudio. Comenzamos con las declaraciones de las clases simples y seguimos con los métodos que manejan los vectores.

8.7. Construcción del Programa del Caso de Estudio

8.7.1. La Clase Libro

La clase Libro es responsable de manejar sus seis atributos, abastecer ejemplares, vender ejemplares y retornar el listado de transacciones. Para esto cuenta con un método constructor, cinco métodos analizadores y dos métodos modificadores:

`Libro( String pTitulo, String pISBN, double pPrecioCompra, double pPrecioVenta, String pRutaImagen )`	**Método constructor.**
`String darTitulo( )`	Retorna el título del libro.
`String darIsbn( )`	Retorna el ISBN del libro.
`double darPrecioCompra( )`	Retorna el precio de compra del libro.
`double darPrecioVenta( )`	Retorna el precio de venta del libro.
`String darCantidadActual( )`	Retorna la cantidad de ejemplares del libro.
`String darRutaImagen( )`	Retorna la ruta de la imagen del libro.
`void vender( int pCantidad, String pFecha )`	Vende ejemplares del libro.
`void abastecer( int pCantidad, String pFecha )`	Abastece ejemplares del libro.
`ArrayList<Transaccion> darTransacciones( )`	Retorna las transacciones del libro.

La clase libro es responsable de abastecer y vender ejemplares del libro así como de registrar una transacción por cada abastecimiento o venta que realice el usuario.

> Al igual que en el caso de los arreglos, si antes de usar un vector no lo hemos creado adecuadamente, se va a generar el error de ejecución: *java.lang.NullPointerException*.

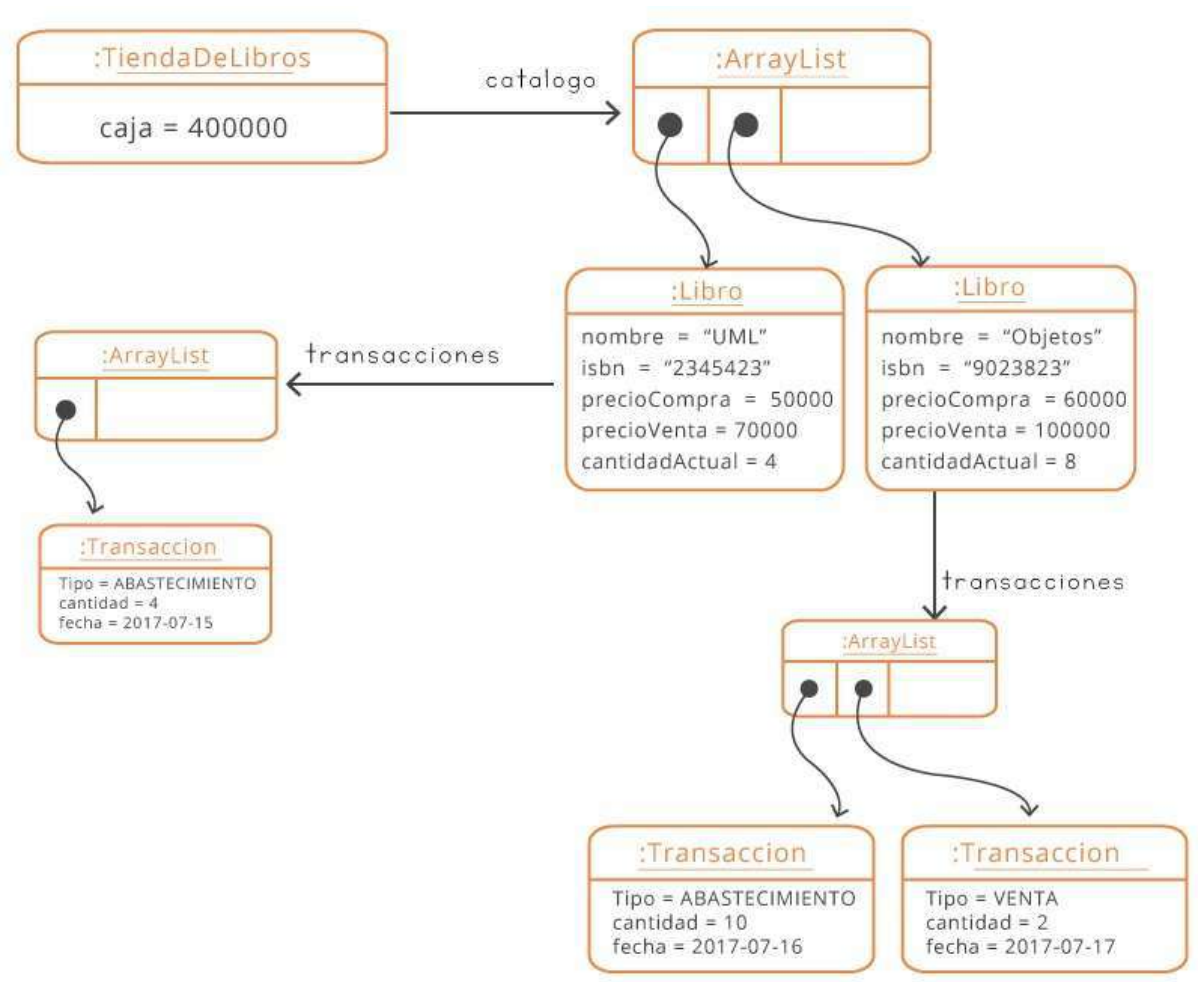
8.7.2. La Clase Transaccion

Cada objeto de la clase Transaccion tiene el tipo de transacción, la cantidad de ejemplares y la fecha en que se realizó la transacción. Aquí es importante resaltar que los objetos de la clase Libro tendrán varias transacciones, como se ilustra en el diagrama de objetos de la figura 3.11.

Los métodos de esta clase se resumen en la siguiente tabla:

`Transaccion( Tipo pTipo, int pCantidad, String pFecha )`	**Método constructor.**
`Tipo darTipo( )`	Retorna el tipo de transacción.
`int darCantidad( )`	Retorna la cantidad de las transacción.
`String darFecha( )`	Retorna la fecha de la transacción.

Fig. 3.11 Diagrama de objetos para ilustrar el caso de la tienda de libros



En la figura 3.11 se puede apreciar el caso en el que el usuario tiene en su catalogo dos libros. El primer libro tiene una transacción y el segundo libro tiene dos transacciones. En este diagrama decidimos mostrar los vectores como objetos externos a las clases que los usan. Esta representación se ajusta más a la realidad que la que usamos en ejemplos anteriores, aunque es menos simple. Ambas maneras de mostrar el diagrama de objetos son válidas. Observe, por ejemplo, que el objeto llamado catalogo es una asociación hacia un objeto de la clase ArrayList, que mantiene las referencias a los objetos que representan los libros.

8.7.3. La Clase TiendaDeLibros

En la tarea 10 vamos a desarrollar algunos de los métodos de la clase TiendaDeLibros. Sus principales responsabilidades se resumen en la siguiente tabla:

`TiendaLibros( )`	**Método constructor.**
`ArrayList<Libro> darCatalogo( )`	Retorna el catálogo de libros.
`double darCaja( )`	Retorna el saldo de la caja.
`void cambiarCaja( double pCaja )`	Cambia el saldo de la caja.
`Libro registrarLibro( String pTitulo, String pIsbn, double pPrecioVenta, double pPrecioCompra, String pRutaImagen )`	Añade un nuevo libro al catálogo a partir de los parámetros recibidos. Si el libro ya está en el catálogo, el método no hace nada.
`Libro buscarLibroPorISBN( String pIsbn )`	Localiza un libro del catálogo dado su ISBN. Si no lo encuentra retorna `null` .
`Libro buscarLibroPorTitulo( String pTitulo )`	Localiza un libro del catálogo dado su título. Si no lo encuentra retorna `null` .
`boolean eliminarLibro( String pIsbn )`	Elimina un libro del catálogo dado su ISBN. Si no lo encuentra retorna false.
`boolean abastecer( String pIsbn, int pCantidad, String pFecha )`	Abastece ejemplares de un libro dado su ISBN. Si no puede abastecer los ejemplares del libro retorna `false` .
`boolean vender( String pIsbn, int pCantidad, String pFecha )`	Vende ejemplares de un libro dado su ISBN. Si no puede vender los ejemplares del libro retorna false.
`Libro darLibroMasCostoso( )`	Retorna el libro con el precio de venta mayor. Si no hay libros en el catálogo retorna `null` .
`Libro darLibroMasEconomico( )`	Retorna el libro con el precio de venta menor. Si no hay libros en el catálogo retorna `null` .
`Libro darLibroMasVendido( )`	Retorna el libro del cuál se han vendido más ejemplares. Si no hay libros en el catálogo retorna `null` .
`int darCantidadTransaccionesAbastecimiento( String pIsbn )`	Retorna el número de transacciones de tipo abastecimiento que se han realizado al libro con el ISBN recibido como parámetro. En caso de que no encuentre el libro o que el libro no tenga transacciones, retorna cero.

Tarea 11

Objetivo: Desarrollar los métodos de la clase TiendaDeLibros que nos permiten implementar los requerimientos funcionales del caso de estudio.

Para cada uno de los problemas que se plantean a continuación, escriba el método que lo resuelve. No olvide identificar primero el patrón de algoritmo que se necesita y usar las guías que se dieron en secciones anteriores.

Localizar un libro en el catálogo, dado su ISBN. Si no lo encuentra, el método debe retornar `null`:

```
public Libro buscarLibroPorISBN( String pIsbn )
{









}
```

Eliminar un libro en el catálogo dado su ISBN. Si el libro no existe o si la cantidad actual de ejemplares es mayor a cero retorna `false`. Utilice el método anterior:

```
public boolean eliminarLibro( String pIsbn )
{









}
```

Agregar un libro en el catálogo, si no existe ya un libro con ese ISBN. Utilice el método `buscarLibroPorISBN`:

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









}
```

Buscar el libro más costoso del catálogo, si el catálogo está vacío retorna `null`:

```
public Libro darLibroMasCostoso( )
{









}
```

Buscar el libro del cuál se han vendido más ejemplares. Si no hay libros en el catálogo, retorna `null:`

```
public Libro darLibroMasVendido( )
{









}
```

Retorna el número de transacciones de tipo abastecimiento que se le han realizado al libro con el ISBN recibido como parámetro. En caso de que no encuentre el libro o que el libro no tenga transacciones, retorna cero.

```
public int darCantidadTransaccionesAbastecimiento( String pIsbn )
{









}
```

Tarea 12

Objetivo: Desarrollar los métodos de la clase Libro.

Para cada uno de los problemas que se plantean a continuación, escriba el método que lo resuelve.

Vender la cantidad de ejemplares del libro recibida como parámetro siempre y cuando la cantidad de ejemplares actual sea menor o igual a la cantidad a vender. La venta implica decrementar el número de ejemplares del libro. Adicionalmente agrega una nueva transacción de tipo venta al listado de transacciones del libro:

```
public boolean vender( int pCantidad, String pFecha )
{









}
```

Abastecer la cantidad de ejemplares del libro recibida como parámetro. El abastecimiento implica incrementar el número de ejemplares del libro. Adicionalmente agrega una nueva transacción de tipo abastecimiento al listado de transacciones del libro:

```
public void abastecer( int pCantidad, String pFecha )
{









}
```

9. Uso de Ciclos en Otros Contextos

Aunque hasta este momento sólo hemos mostrado las instrucciones iterativas como una manera de manejar información que se encuentra en estructuras contenedoras, dichas instrucciones también se usan muy comúnmente en otros contextos. En el ejemplo 17 mostramos su uso para calcular el valor de una función aritmética.

Ejemplo 18

Objetivo: Mostrar el uso de las instrucciones iterativas en un contexto diferente al de manipulación de estructuras contenedoras.

En este ejemplo presentamos la manera de escribir un método para calcular el factorial de un número. La función factorial aplicada a un número entero n (en matemáticas a ese valor se le representa como n!) se define como el producto de todos los valores enteros positivos menores o iguales al valor en cuestión. Planteado de otra manera, tenemos que:

- factorial(0) es igual a 1.
- factorial(1) es igual a 1.
- factorial(n) = n * factorial(n – 1).

Por ejemplo, factorial(5) = 5 * 4 * 3 * 2 * 1 = 120

Si queremos construir un método capaz de calcular dicho valor, podemos utilizar una instrucción iterativa, como se muestra a continuación.

```
package  uniandes.cupi2.matematicas;
public class Matematica
{
    public  static int factorial( int pNum  )
    {
        int acum = 1;

        if( pNum > 0 )
        {
            for( int i = 1; i <= num; i++ )
            {
                acum = acum * i;
            }
        }
    }
    return acum;
}
```

El método lo declaramos de manera especial (`static`) y su modo de uso es como aparece más
abajo en este mismo ejemplo.

El primer caso que tenemos es que el valor del parámetro sea 0. La respuesta en ese caso es 1.
Hasta ahí es fácil.

En el caso general, debemos multiplicar todos los valores desde 1 hasta el valor que recibimos
como parámetro e ir acumulando el resultado en una variable llamada " `acum` ". Al final el método retorna dicho valor.

Esta solución no es otra que el patrón de recorrido total aplicado a la secuencia de números. Aunque no estén almacenados en un arreglo, se pueden imaginar uno después del otro, con el índice recorriéndolos de izquierda a derecha. Este uso de las instrucciones iterativas no tiene una teoría distinta a la vista en este capítulo.

```
int fact = Matematica.factorial( i );
```

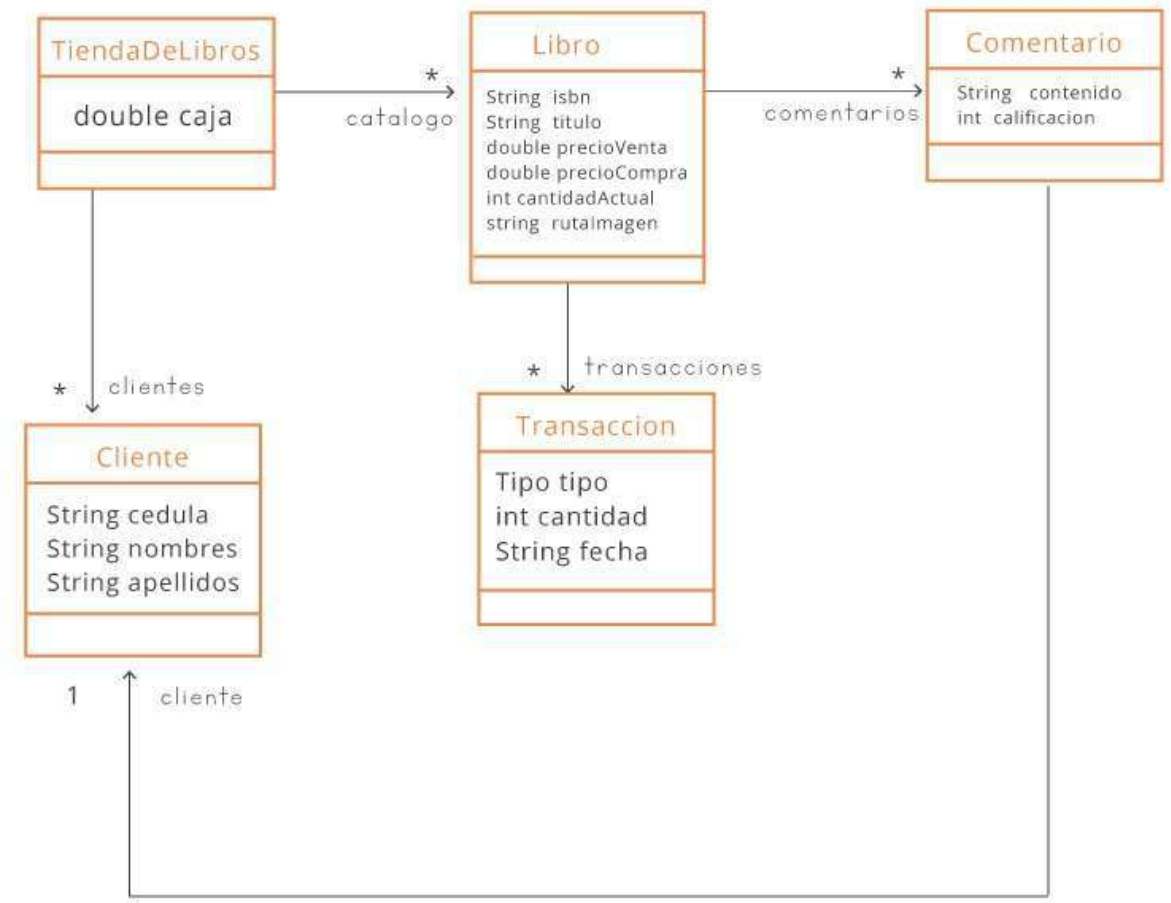
La llamada del método se hace utilizando esta sintaxis. Como es una función aritmética que no
está asociada con ningún elemento del mundo, debemos usar el nombre de la clase para hacer la invocación.

Creación de una Clase en Java

Tarea 13

Objetivo: Agregar una nueva clase en un programa escrito en Java.

En esta tarea vamos a extender el caso de estudio de la tienda de libros, agregando dos clases nuevas, en un paquete distinto a los ya definidos. Siga los pasos que se detallan a continuación:



Este es el diagrama de clases que queremos construir. Hay dos clases adicionales: una para modelar los clientes de la tienda de libros y otra con comentarios que se hacen opcionalmente sobre cada libro. Tome nota de las nuevas asociaciones que aparecen.

1. Ejecute Eclipse y abra el proyecto de la tienda de libros. Localice el directorio en el cual se guardan los programas fuente.
2. Vamos a crear los archivos de las clases Comentario y Cliente en un nuevo paquete llamado `uniandes.cupi2.tiendadelibros.extension`. Para esto, debemos crear primero el paquete. Para crear un paquete en Java, seleccione la opción *File/New/Package* del

menú principal o la opción *New/Package* del menú emergente que aparece al hacer clic derecho sobre el directorio de fuentes.

3. Una vez creado el paquete, podemos crear la clase allí dentro, seleccionando la opción *File/New/Class* del menú principal o la opción *New/Class* del menú emergente que aparece al hacer clic derecho sobre el paquete de clases elegido. En la ventana que abre el asistente de creación de clases, podemos ver el directorio de fuentes y el paquete donde se ubicará la clase. Allí debemos teclear el nombre de la clase. Al oprimir el botón *Finish*, el editor abrirá la clase y le permitirá completarla con sus atributos y métodos. Siguiendo el proceso antes mencionado, cree las clases Cliente y Comentario incluyendo sus atributos.
4. El siguiente paso es agregar los atributos que van a representar las asociaciones hacia esas clases. Abra para esto la clase Libro. Agregue el atributo de tipo vector que representa la asociación hacia la clase Comentario tal como se describe en el diagrama de clases. ¿Por qué el compilador no reconoce la nueva clase? Sencillamente porque está en otro paquete, el cual debemos importar. Añada la instrucción para importar las clases del nuevo paquete. Esta importación puede hacerla manualmente o utilizando el comando Control+Mayús+O para que el editor agregue automáticamente todas las importaciones que necesite.
5. Agregue el atributo clientes a la clase TiendaDeLibros, representándolo como un vector. Es necesario que importe la clase Cliente al momento de declarar el vector puesto que es la primera vez que hacemos referencia directa a esta clase.
6. En el constructor de la clase TiendaDeLibros, inicialice el vector de clientes.
7. En el constructor de la clase Libro, inicialice el vector de comentarios.
8. Las clases antes mencionadas también se habrían podido crear desde cualquier editor de texto simple (por ejemplo, el bloc de notas). Basta con crear el archivo, salvarlo en el directorio que representa el paquete y, luego, entrar a Eclipse y utilizar la opción *Refresh* del menú emergente que aparece al hacer clic derecho sobre el proyecto.
9. En la clase Comentario agregue el constructor que recibe como parámetros el contenido, la calificación y el objeto del cliente que realizó el comentario. Agregue tres métodos para recuperar el contenido del comentario, la calificación otorgada y el cliente.
10. En la clase Cliente escriba el constructor que recibe como parámetros la cédula, los nombres y los apellidos. Agregue tres métodos para recuperar la cédula, los nombres y los apellidos.
11. En la clase Libro, añada un método que agregue un comentario al libro y otro que retorne el vector con todos los comentarios del libro.
12. En la clase TiendaDeLibros, añada los siguientes métodos: (a) un método para agregar un nuevo cliente, (b) un método para buscar un cliente dado su número de cédula, (c) un método para calcular la calificación promedio de un libro dado su ISBN, (d) un método que calcule el número total de libros del catálogo que tienen al menos

comentario, y (e) un método que agregue un nuevo comentario a un libro. Este último método recibe como parámetros el ISBN del libro, la cédula del cliente, y el contenido y la calificación del comentario.

10. Hojas de Trabajo

10.1. Hoja de Trabajo Nº 1: Un Parqueadero

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 quiere que haga el programa y las restricciones para desarrollarlo.

Se quiere construir una aplicación para administrar un parqueadero (lugar de estacionamiento para carros). Dicho parqueadero tiene 40 puestos, numerados del 1 al 40. En cada puesto se puede parquear un sólo carro (que representaremos con una clase llamada Carro), el cual se identifica por su placa. El parqueadero tiene una tarifa por hora o fracción de hora, puede ser cambiada por el administrador.

De cada vehículo aparcado se debe conocer la hora en la que entró, que corresponde a un valor entre 6 y 21, dado que el parqueadero está abierto entre 6 de la mañana y 9 de la noche.

Se espera que la aplicación que se quiere construir permita hacer lo siguiente:

1. Ingresar un carro al parqueadero. Se debe indicar el puesto en el que se debe parquear (si hay cupo).
2. Dar salida a un carro del parqueadero. Se debe indicar cuánto debe pagar.
3. Informar los ingresos del parqueadero.
4. Consultar la cantidad de puestos disponibles.
5. Avanzar una hora en el reloj del parqueadero.
6. Cambiar la tarifa del parqueadero.

La siguiente es la interfaz de usuario propuesta para el programa, donde los puestos ocupados deben aparecen con un vehículo.

Requerimientos funcionales. Describa los seis requerimientos funcionales de la aplicación 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

Nombre	
Resumen	
Entradas	
Resultado	

Modelo del mundo. Complete el diagrama de clases con los atributos, las constantes y las asociaciones.

Diagrama UML: Parqueadero

Diagrama UML: Puesto

Diagrama UML: Carro

Declaración de arreglos. Para las siguientes clases, escriba la declaración de los atributos indicados en el comentario (como contenedoras del tipo dado), así como las constantes necesarias para manejarlos.

```
public class Parqueadero
{
    //-----------------------------------------------
    // Constantes
    //-----------------------------------------------
    /**
     * Indica el número de puestos en el parqueadero
     */


    //-----------------------------------------------
    // Atributos
    //-----------------------------------------------
    /**
     * Arreglo de puestos
     */


}
```

Inicialización de arreglos. Escriba el constructor de la clase para inicializar las contenedoras declaradas en el punto anterior.

```
public Parqueadero( )
{




}
```

Patrones de algoritmos. Desarrolle los siguientes métodos de la clase Parqueadero, identificando el tipo de patrón de algoritmo al que pertenece y siguiendo las respectivas guías

Método 1

Contar y retornar el número total de puestos ocupados.

```
public int darTotalPuestosOcupados( )
{









}
```

Método 2

Informar si en el parqueadero hay un carro cuya placa comience con la letra dada como parámetro.

```
public boolean existePlacaIniciaCon( char pLetra )
{









}
```

Método 3

Retornar el número de carros en el parqueadero que llegaron antes del mediodía.

```
public int darTotalCarrosIngresoManana( )
{









}
```

Método 4

Retornar el último carro en ingresar al parqueadero. Si el parqueadero está vacío, retorna null.

```
public Carro darCarroLlegadaMasReciente( )
{









}
```

Método 5

Informar si en algún lugar del parqueadero hay dos puestos libres consecutivos. Esto se hace cuando el vehículo que se quiere aparcar es muy grande.

```
public boolean hayDosPuestosLibresConsecutivos( )
{









}
```

Método 6

Informar si hay dos carros en el parqueadero con la misma placa.

```
public boolean hayPlacasRepetidas( )
{









}
```

10.2 Hoja de Trabajo Nº 2: Lista de Contactos

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 quiere que haga el programa y las restricciones para desarrollarlo.

Se quiere construir un programa para manejar la lista de contactos de una persona. Un contacto tiene nombre, apellido, una dirección, un correo electrónico, varios teléfonos y un conjunto de palabras clave que se utilizan para facilitar su búsqueda. El nombre completo (nombre + apellido) de cada contacto debe ser único. Tanto el nombre como el apellido se usan como palabras clave para las búsquedas.

En el programa de contactos se debe poder:

1. Agregar un nuevo contacto.
2. Eliminar un contacto ya existente.
3. Ver la información detallada de un contacto.
4. Modificar la información de un contacto.
5. Buscar contactos usando las palabras clave.

La siguiente es la interfaz de usuario propuesta para el programa de la lista de contactos.

Requerimientos funcionales. Describa los cinco requerimientos funcionales de la aplicación 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	

Modelo del mundo. Complete el diagrama de clases con los atributos, las constantes y las asociaciones.

Diagrama UML: ListaDeContactos

Diagrama UML: Contacto

Declaración de arreglos. Para las siguientes clases, escriba la declaración de los atributos indicados en el comentario (como contenedoras del tipo dado).

```
public class Contacto
{
    //-----------------------------------------------------
    // Atributos
    //-----------------------------------------------------
    private String nombre;
    private String apellido;
    private String direccion;
    private String correo;

    /**
     * Lista de teléfonos del contacto.
     */


    /**
     * Lista de palabras clave del contacto.
     */


}
```

```
public class ListaDeContactos
{
    //-----------------------------------------------------
    // Atributos
    //-----------------------------------------------------
    /**
     * Lista de contactos.
     */


}
```

Inicialización de arreglos. Escriba el constructor de las clases dadas.

```
public Contacto ( )
{









}
```

```
public ListaDeContactos ( )
{









}
```

Patrones de algoritmos. Desarrolle los siguientes métodos de la clase indicada, identificando el tipo de patrón de algoritmo al que pertenece y siguiendo las respectivas guías.

Metodo

Clase: Contacto

Contar el número de palabras clave que empiezan por la letra dada como parámetro.

```
public int darTotalPalabrasInicianCon( char pLetra )
{









}
```

Metodo 2

Clase: Contacto

Informar si el contacto tiene algún teléfono que comienza por el prefijo dado como parámetro.

```
public boolean existeTelefonoIniciaCon( String pPrefijo )
{









}
```

Metodo 3

Clase: Contacto

Retornar la primera palabra clave que termina con la cadena dada.

```
public String darPalabraTerminaCon( String pCadena )
{









}
```

Metodo 4

Clase: Contacto

Contar el número de palabras clave que son prefijo (parte inicial) de otras palabras clave.

```
public int darTotalPalabrasPrefijo( )
{









}
```

Metodo 5

Clase: ListaDeContacto

Contar el número de contactos cuyo nombre es igual al recibido como parámetro.

```
public int darTotalContactosConNombre( String pNombre )
{









}
```

Metodo 6

Clase: ListadeContactos

Informar si hay dos contactos en la lista con la misma dirección de correo electrónico.

```
public boolean hayCorreosRepetidos( )
{









}
```

Metodo 7

Clase: ListadeContactos

Retornar el contacto con el mayor número de palabras clave.

```
public Contacto darContactoConMasPalabras( )
{









}
```

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 pPrecio
Compra, 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.

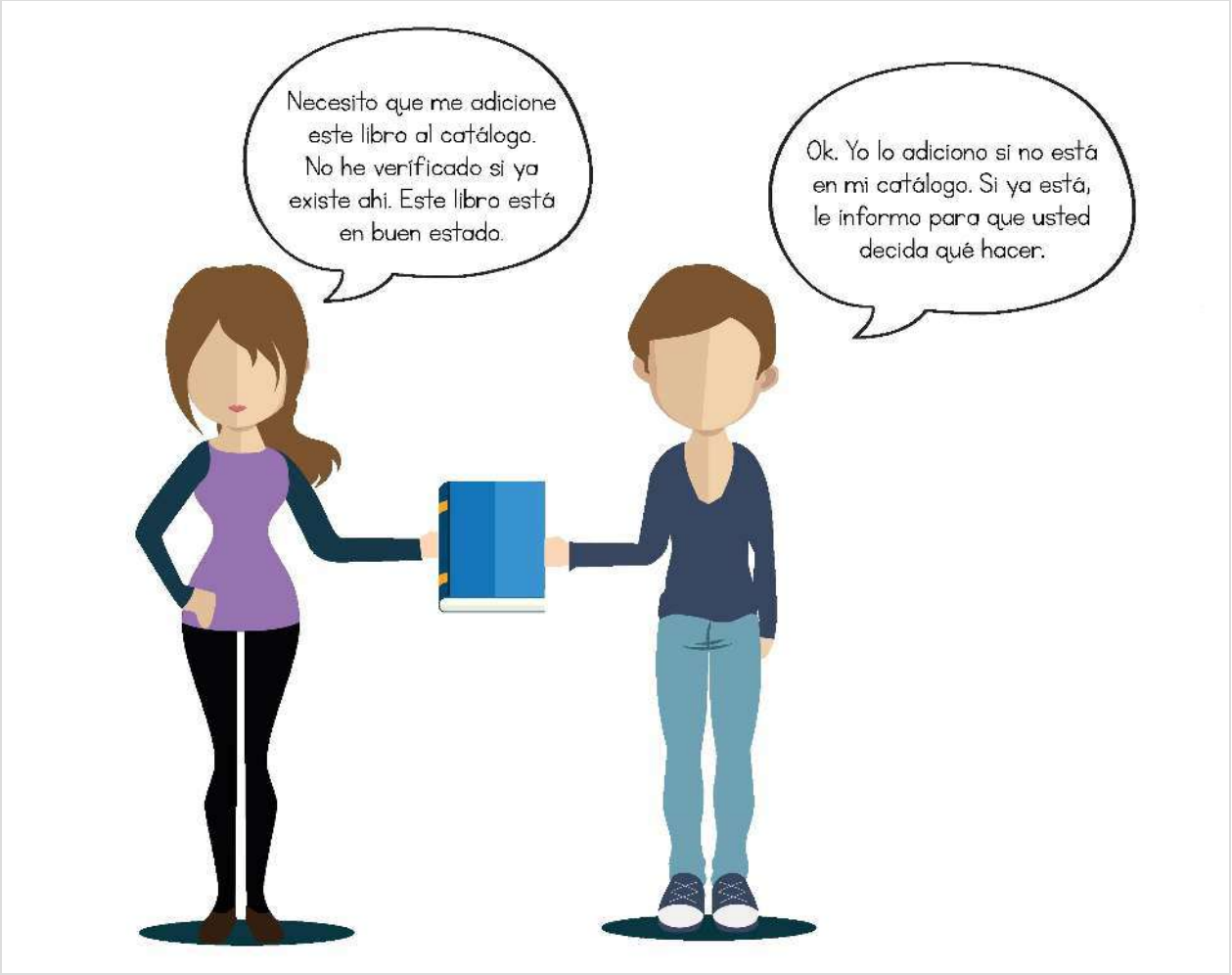
Este otro extremo parece un poco exagerado, puesto que algunas verificaciones pueden tomar mucho tiempo, ser costosas e inútiles. ¿Cómo tomar entonces la decisión de qué validar y qué suponer? La respuesta es que lo importante no es cuál de las soluciones tomemos. Lo importante es que aquello que decidamos 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 Nº 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 "Agregar 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	R1 - Agregar una persona autorizada por un socio.
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	R2 - Pagar una factura.
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

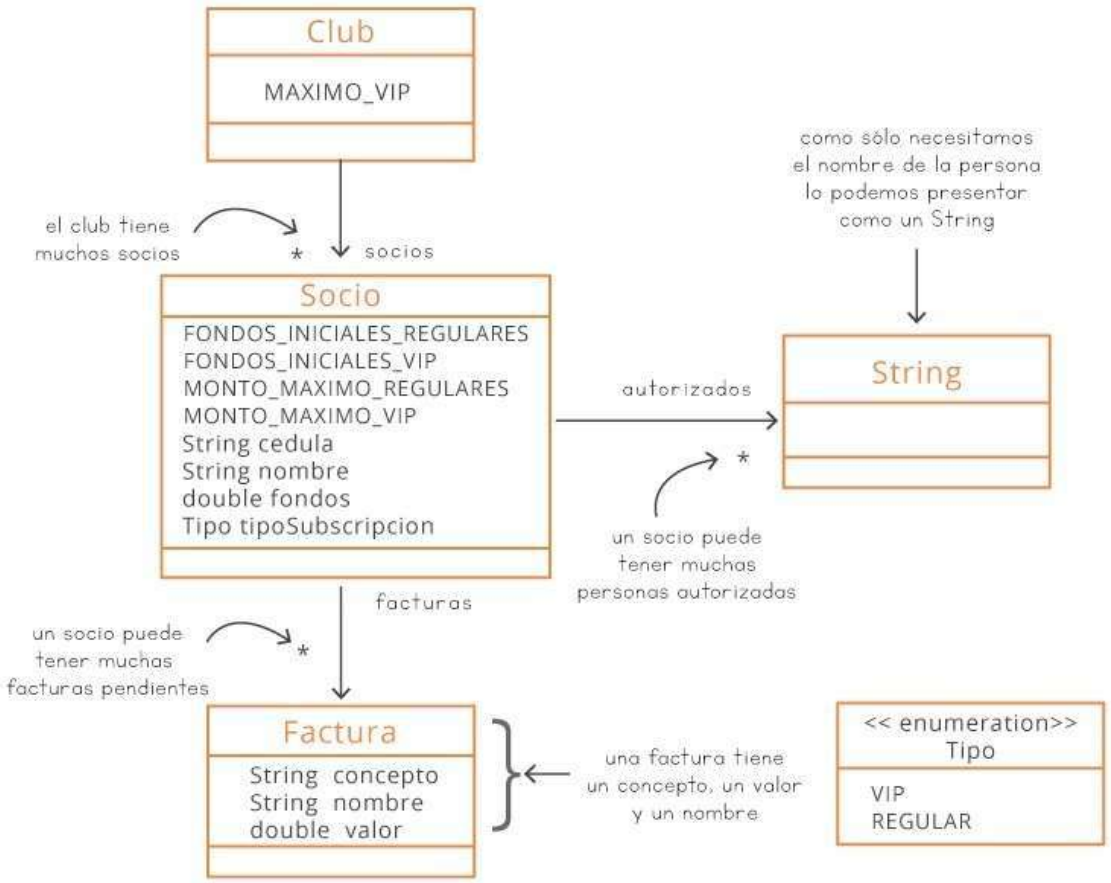
Nombre	R5 - Aumentar los fondos de la cuenta de un socio.
Resumen	
Entradas	
Resultado	

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.

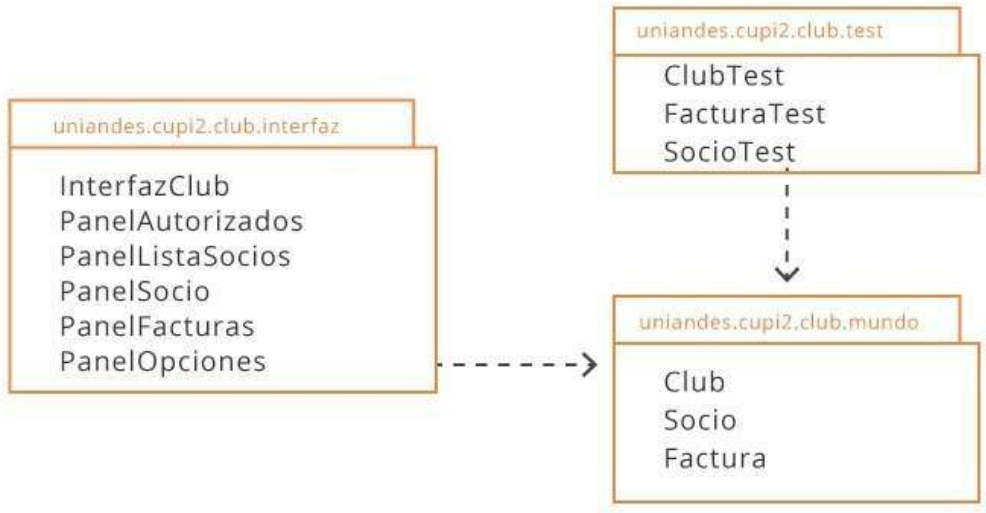
Fig. 4.3 Modelo conceptual del caso de estudio del club



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
    // ------------------------------------------------------------------
```

```java
    /**
     * 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)).

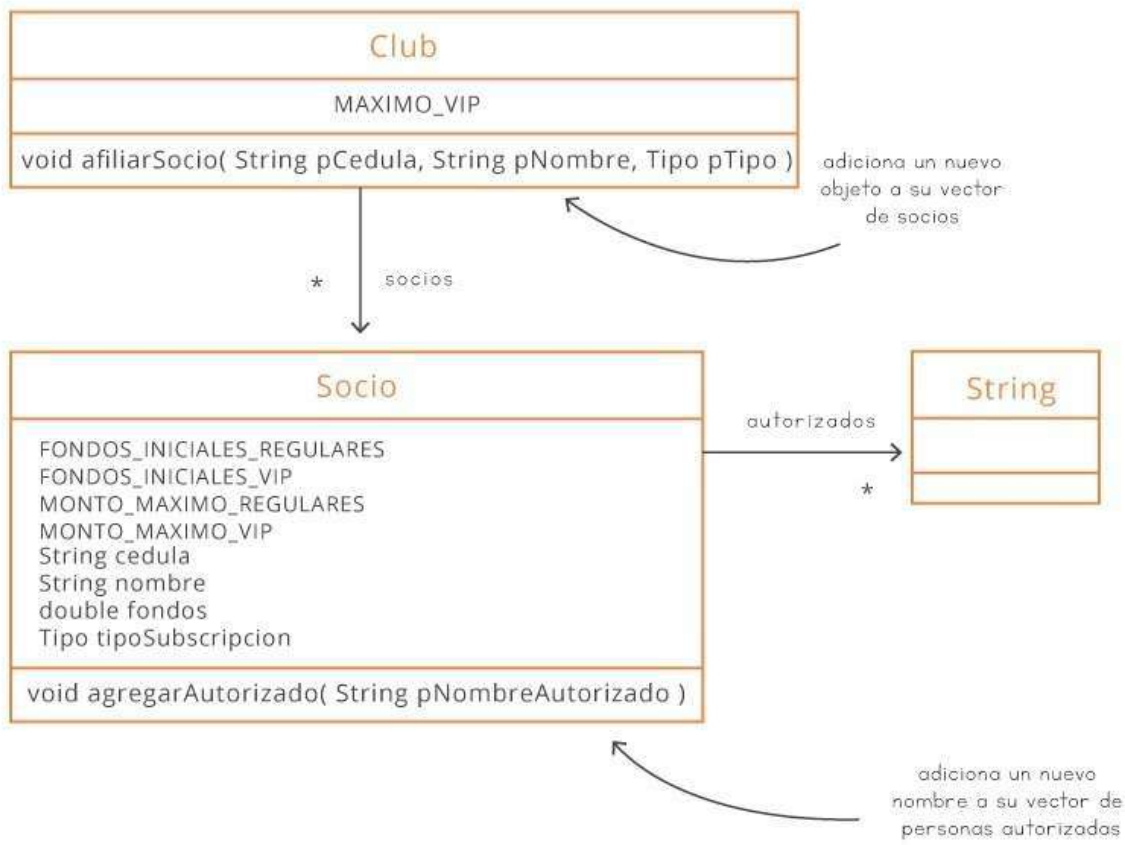
> La técnica del experto define quién es responsable de hacer algo, pero son las reglas del mundo las que nos dicen cómo cumplir con dicha responsabilidad.

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.

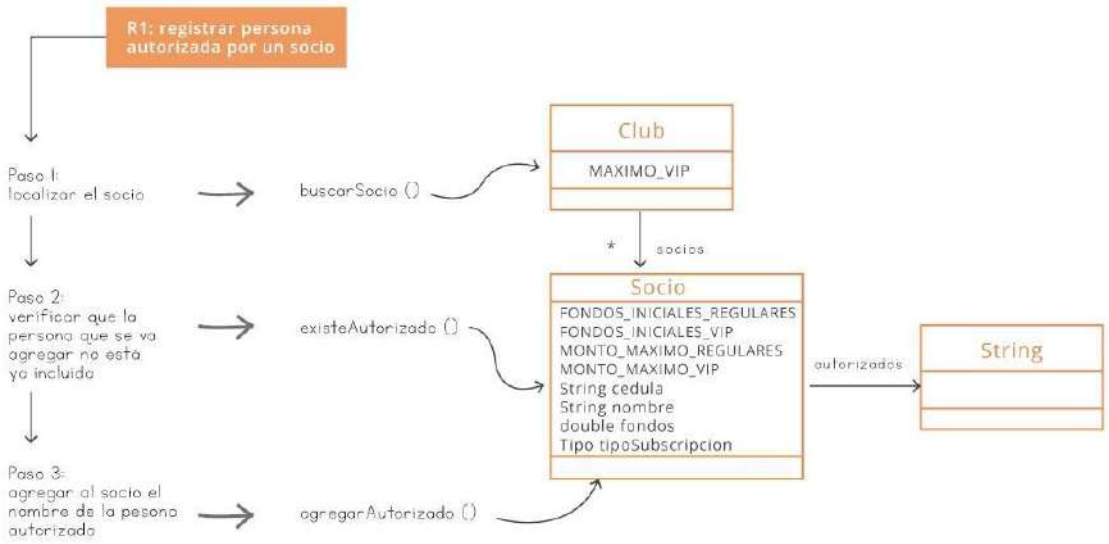
Haga la descomposición en pasos del requerimiento funcional de realizar un consumo en el club.

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 res- ponsabilidad 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 Exceptio
n
{
  ...
}
```

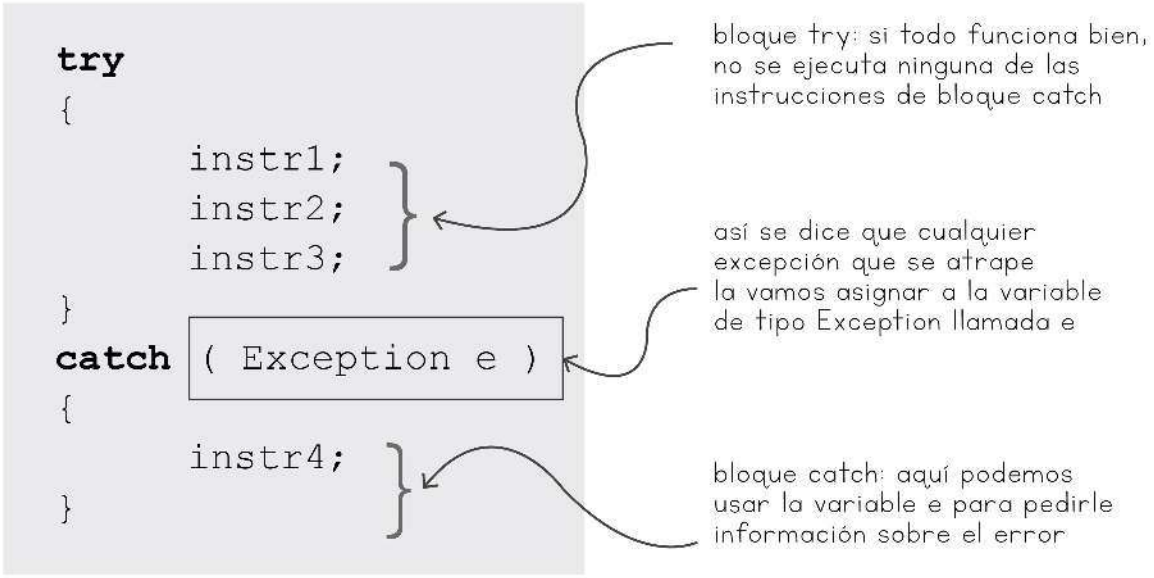
- Con esta declaración el método advierte a todos aquellos que lo usan de que puede producirse una excepción al invocarlo. Los métodos que hacen la invocación pueden decidir atraparla o dejarla pasar.
- No es necesario hacer un `import` de la clase Exception, puesto que esta clase está en un paquete 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 traía 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 Exceptio
n
{

  // 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( ) );
}
```

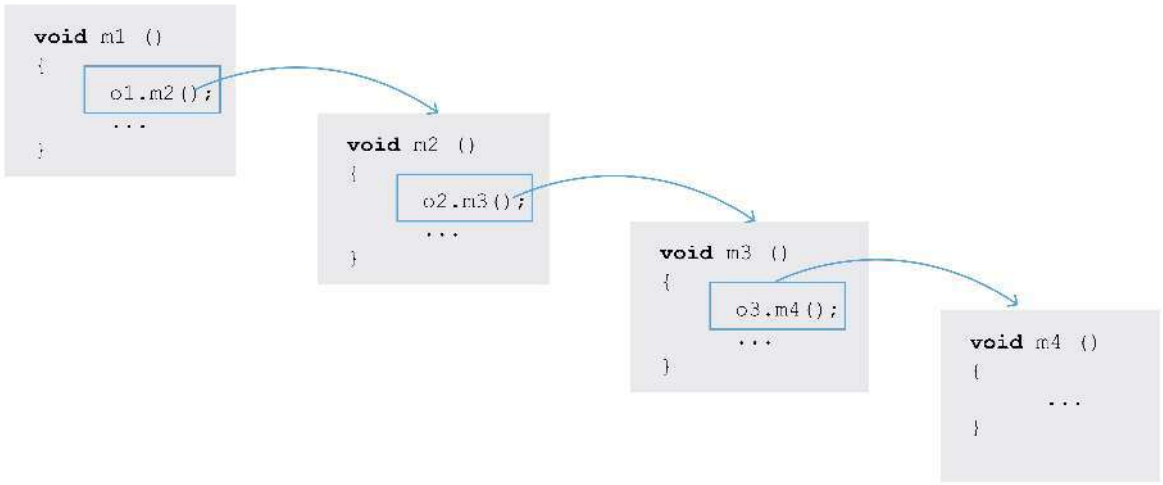
Obtendremos la ventana de advertencia al usuario que aparece en la figura 4.8.

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

5.4. Recuperación de una Situación Anormal

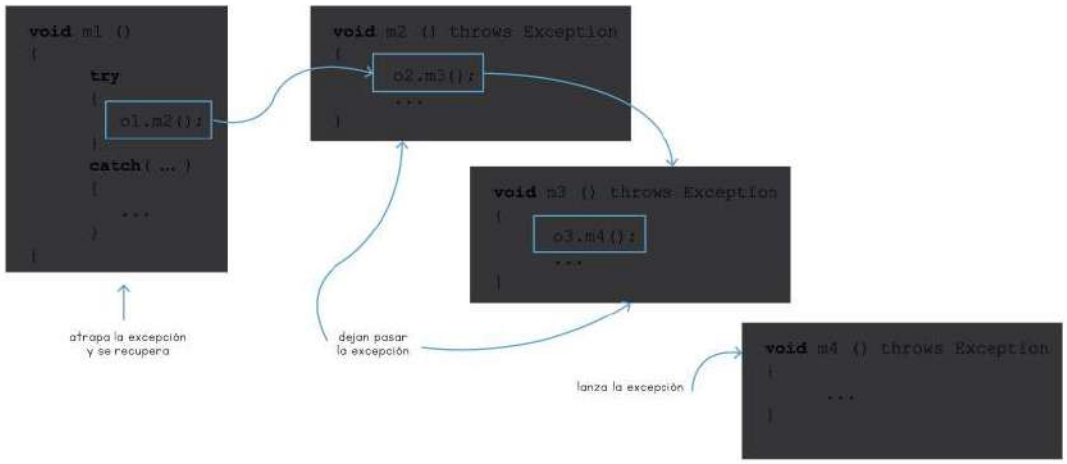
Cuando se está ejecutando un método, puede pasar que desde su interior se invoque otro método y, desde el interior de éste, otro y así sucesivamente. En la figura 4.9 mostramos un ejemplo de la ejecución de un método m1() que invoca un método m2(), el cual llama a m3() y este último a m4().

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 Exceptio
n
```

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 `pNombre` .	
Suposiciones sobre el parámetro `pConcepto` .	
Suposiciones sobre el parámetro `pValor` .	
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 ineficientes. 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 eje- cución 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 alcanz
ado 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 subscripción 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 subscripción VIP pero el club ha alcanz
ado el límite.
*/
public void afiliarSocio( String pCedula, String pNombre, Tipo pTipo ) throws Exceptio
n
{
}
```

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

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 Exceptio
n
```

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
```

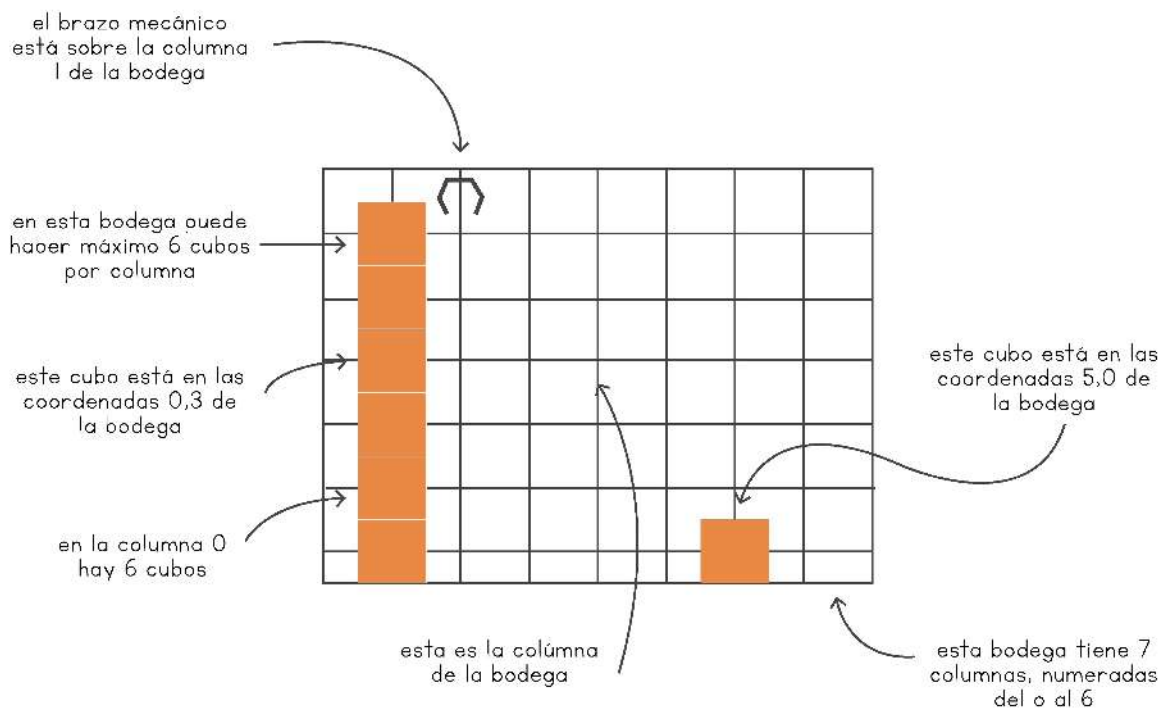
```
void pagarFactura( int pIndiceFactura ) throws Exception
```

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

8. Caso de Estudio Nº 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.

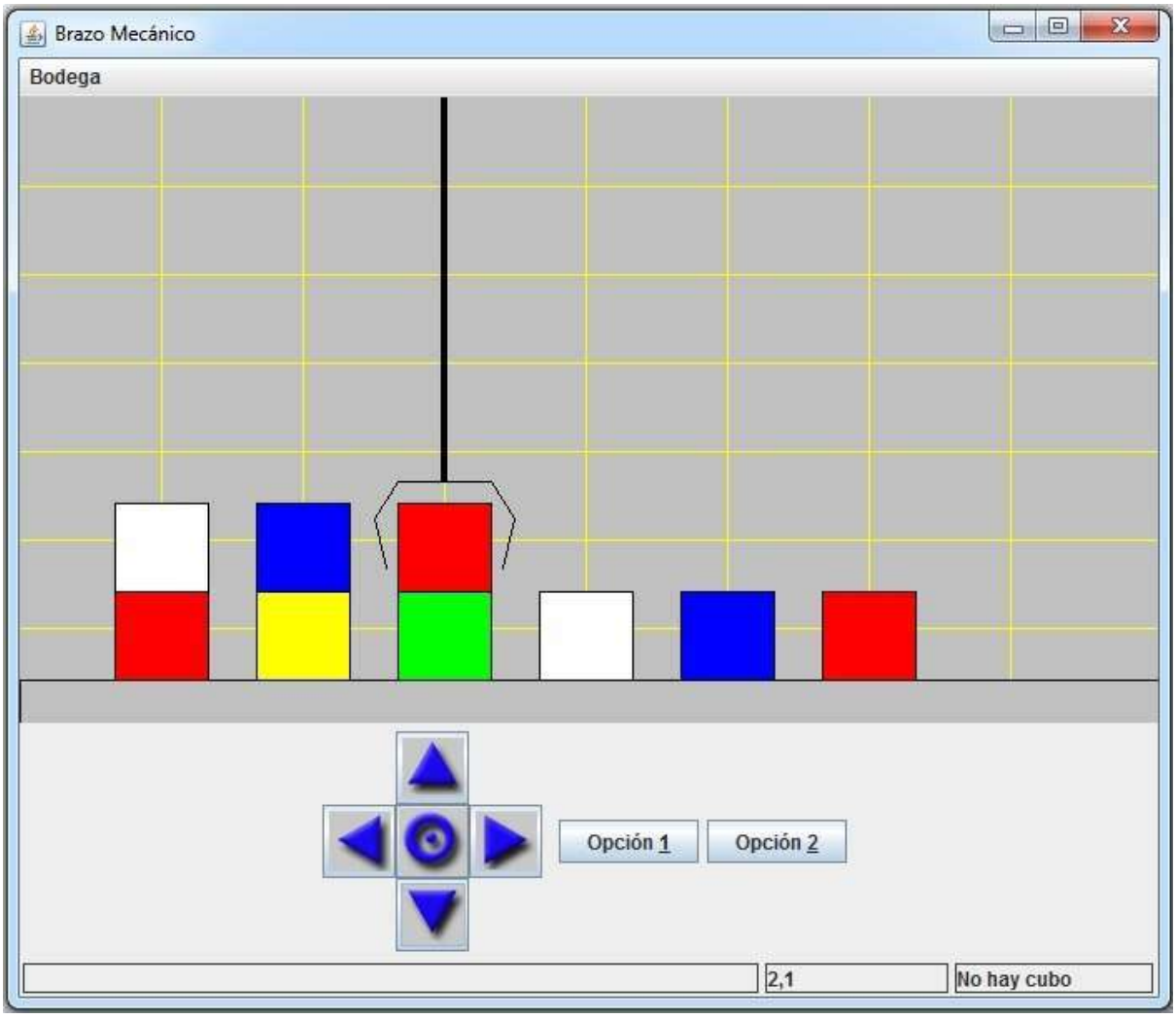
El brazo mecánico está suspendido del techo de la bodega y puede moverse a lo largo de las columnas, al igual que puede subir y bajar. El brazo puede cargar un cubo a la vez y solamente puede tomarlo si se coloca en la misma posición del cubo que quiere agarrar.

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.

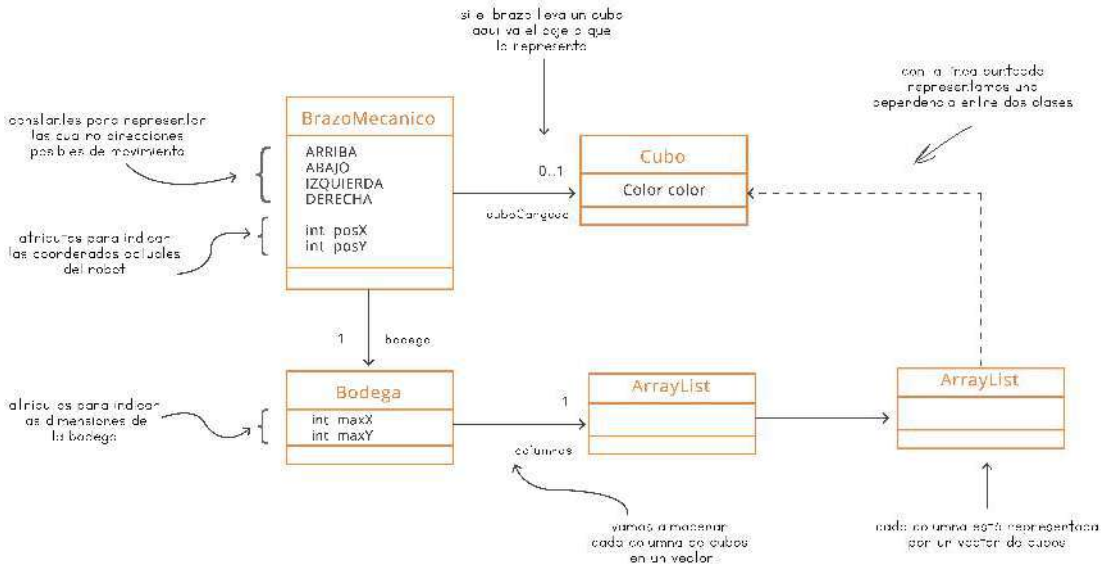
Vamos a utilizar este caso de estudio para generar habilidad en el uso de las nociones de asignación de responsabilidades, contratos y excepciones. Primero, vamos a explicar la manera en que diseñamos e implementamos el mundo del brazo mecánico y luego vamos a resolver algunos problemas en ese mundo.

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. Comprensión 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.

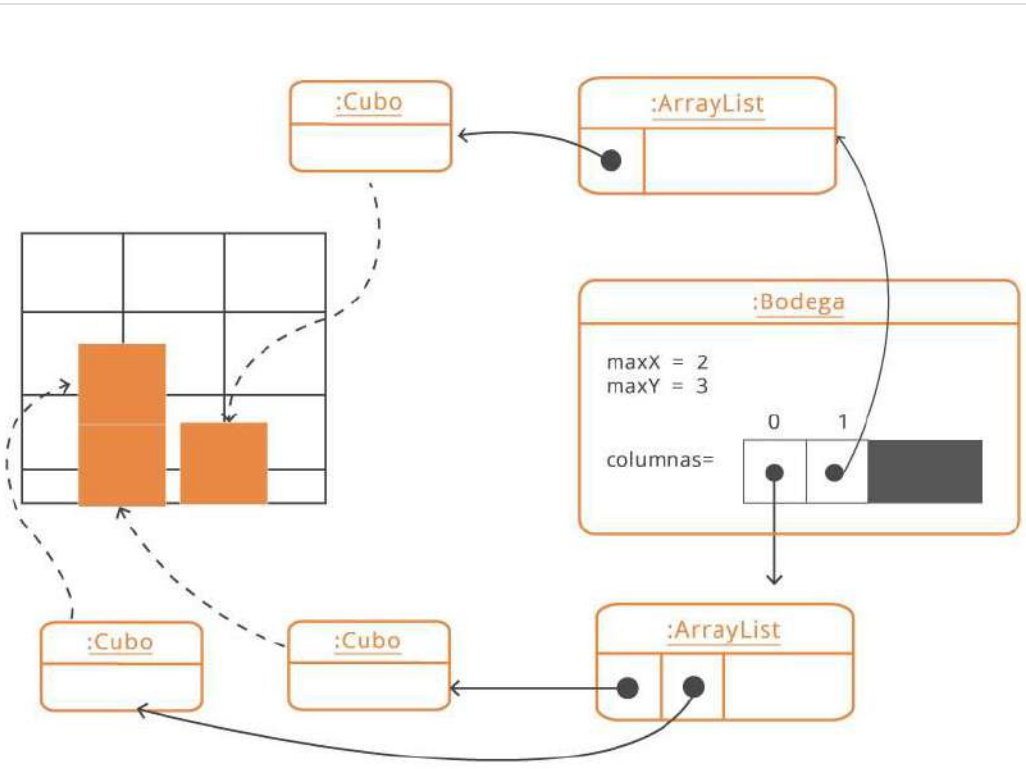
```
public class BrazoMecanico
{

  //-----------------------------
  // 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.

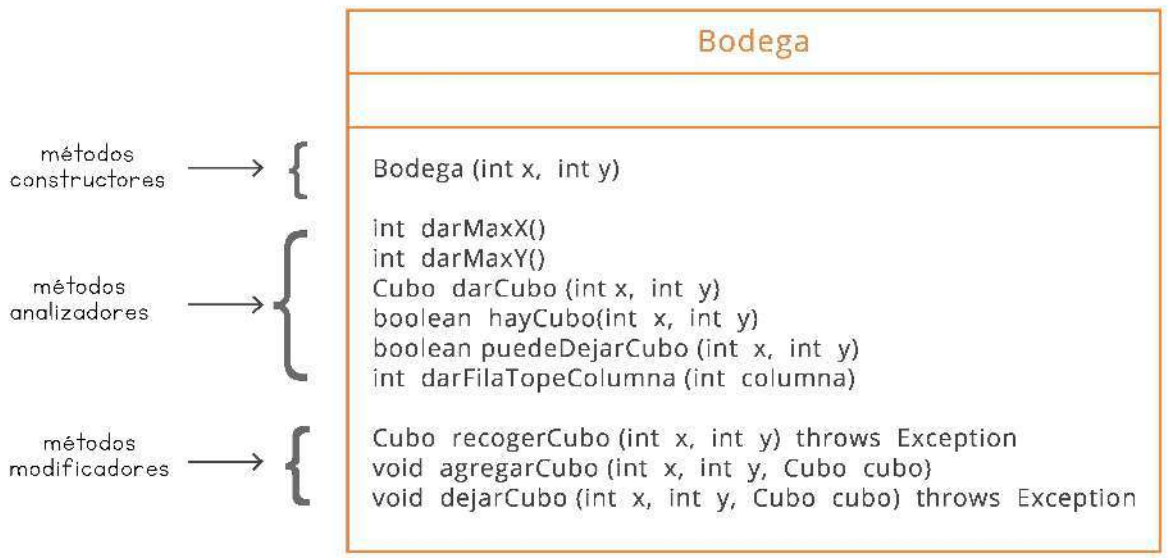
8.2. Comprender la Asignación de Responsabilidades y los Contratos

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:

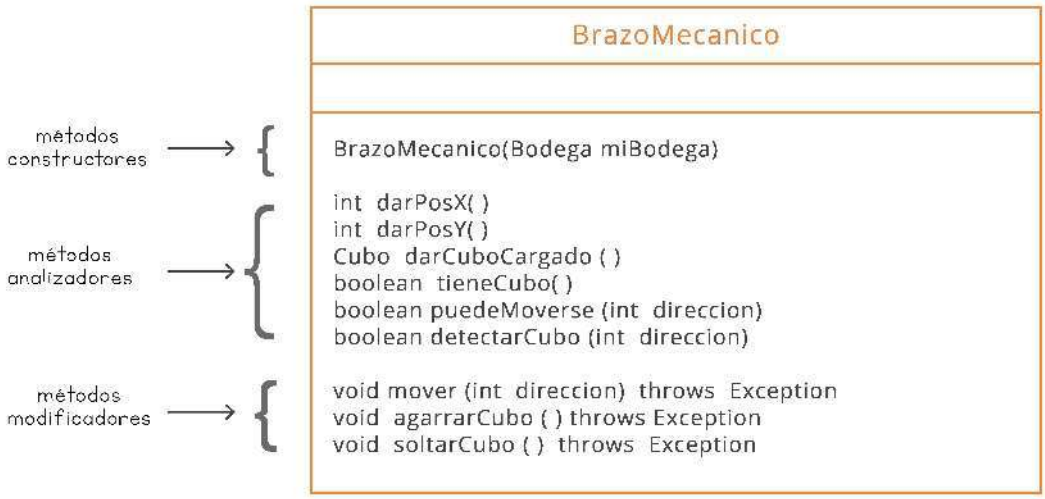
- **Ubicación:** el brazo sabe dónde se encuentra ubicado dentro de la bodega (`posX` , `posY`). Por esta razón, tiene la responsabilidad de informar sobre su posición: `darPosX()` , `darPosY()` .
- **Relación con un cubo:** tiene una asociación de cardinalidad opcional con un cubo, que representa la 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.

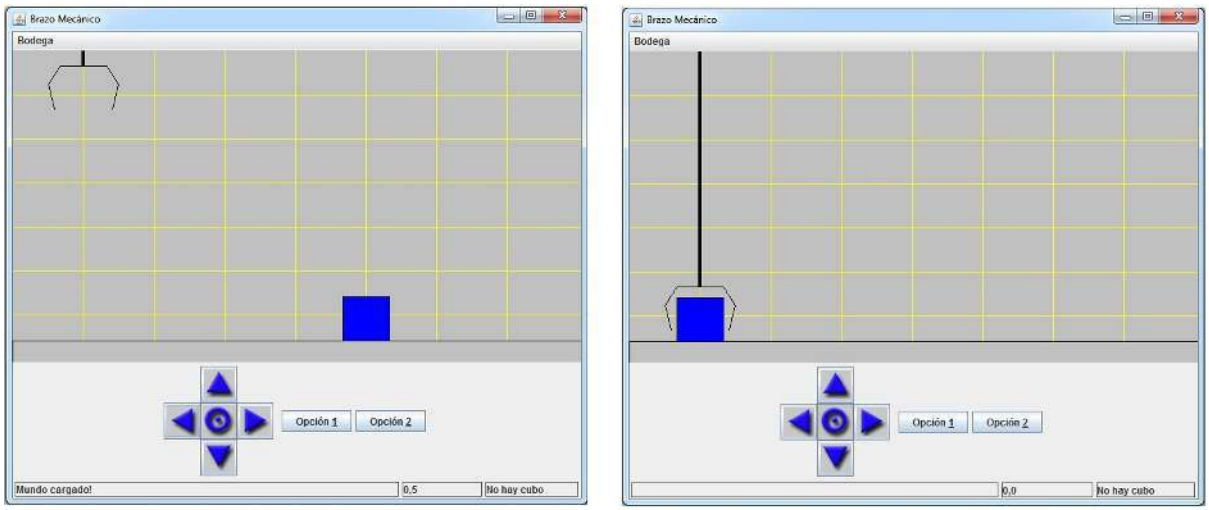
8.3.1. Reto 1

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).

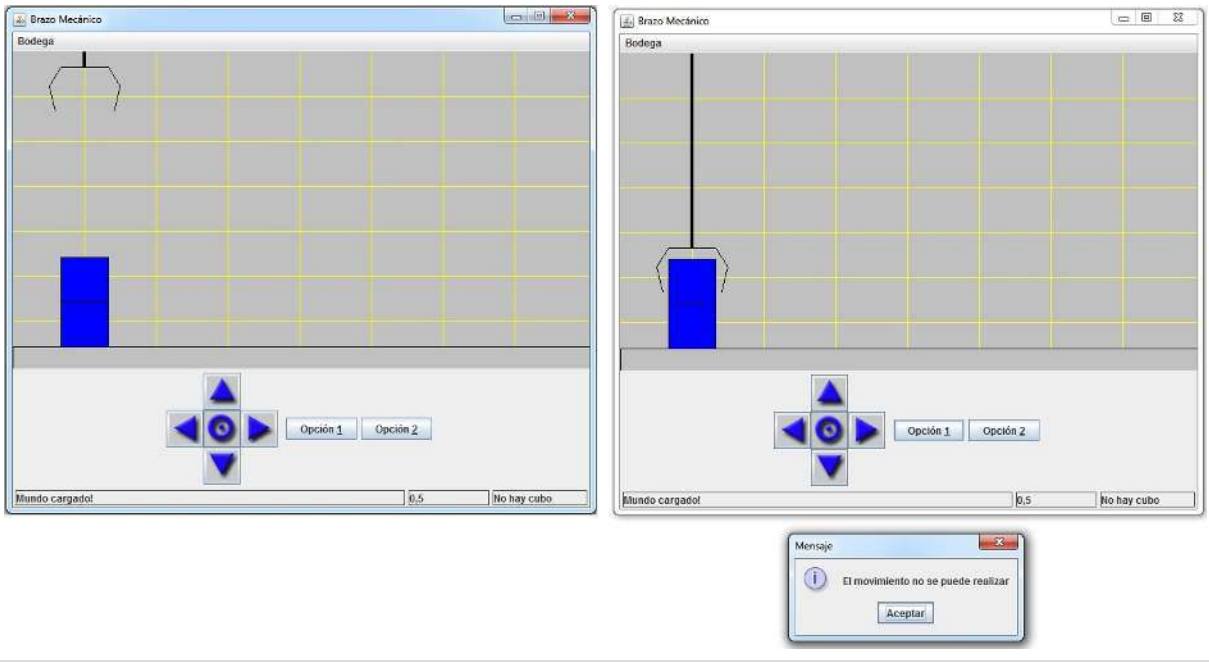
Meta 1: Para lograr la meta 1, debemos mover el brazo hasta que su posición en el eje Y sea igual a 0 (es decir, el piso). Esto lo podemos lograr con una instrucción repetitiva para la que podemos utilizar el patrón de recorrido total.

```
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:

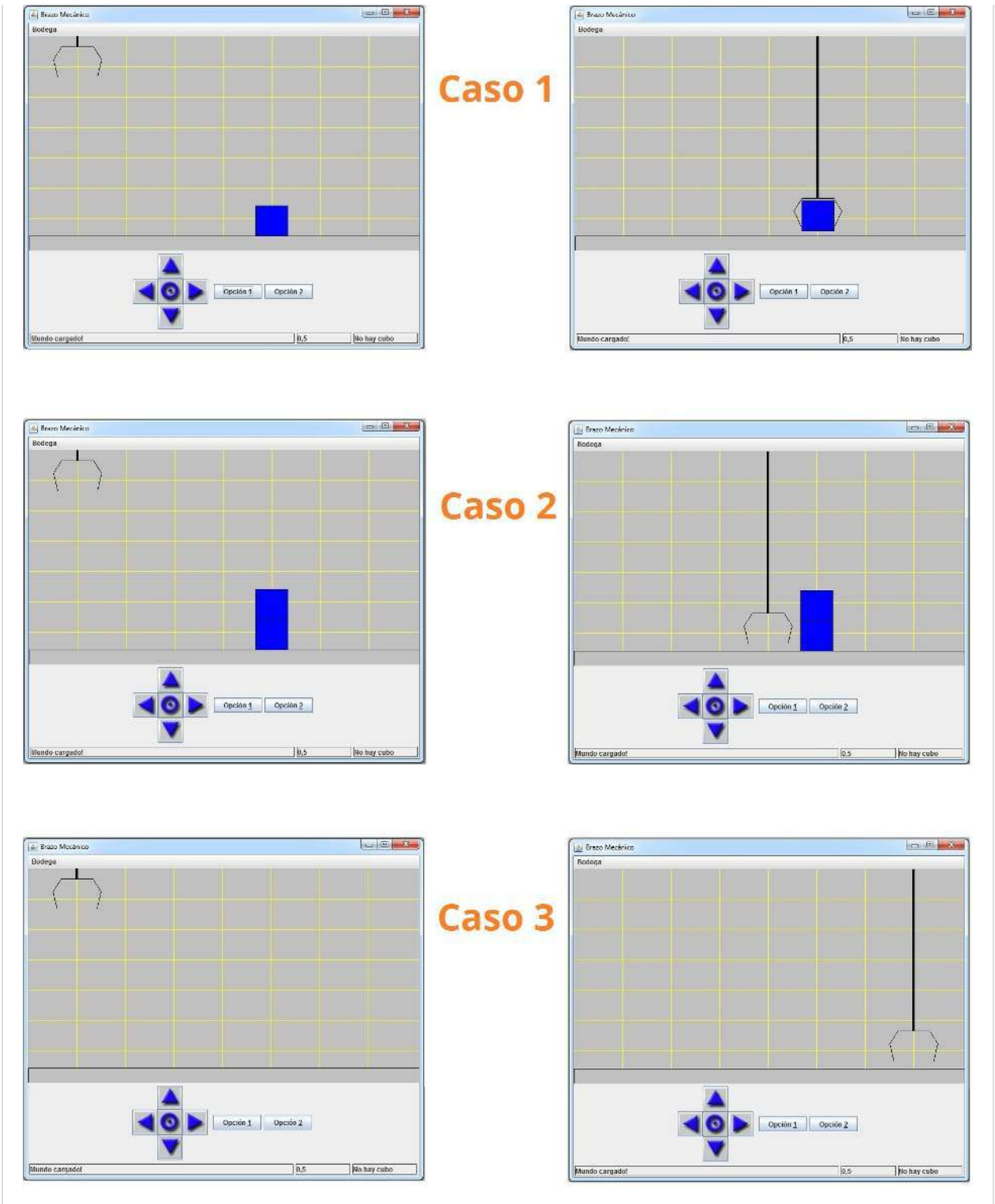
```
/**
* Busca y agarra el único cubo que hay en el mundo.
*
* <b>pre:</b> El brazo está en la posición 0,0 de la bodega.
* <b>post:</b> El brazo está en la posición donde
* se encuentra el cubo y lo está agarrando
*
* @throws Exception Si no encontró un cubo o si el brazo se
* estrelló contra una pila 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



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
  }
}
```

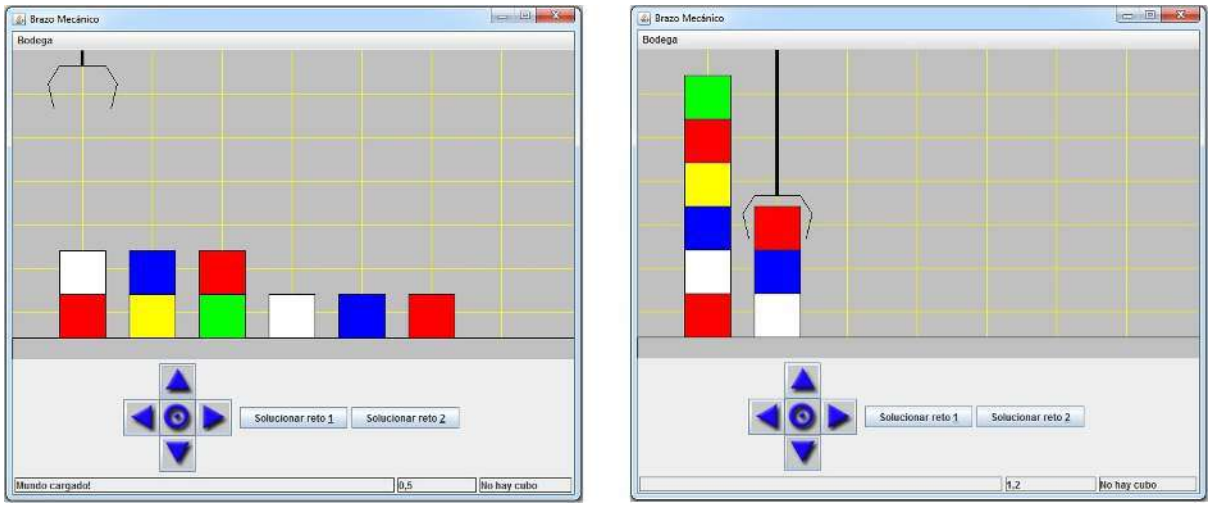
- La solución corresponde de nuevo a una instrucción repetitiva donde se puede aplicar el patrón de recorrido total.
- Dado que el método exige en su precondición que el camino hasta el origen esté despejado y que el cubo esté efectivamente agarrado por el brazo, no existe ninguna posibilidad de que 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 Nº 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

Nombre	
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.

Incrementar el saldo de la tarjeta CINEMAS.	**1. Buscar la tarjeta CINEMAS por su código (Cine).** **2. Aumentar el valor de saldo de la tarjeta (Tarjeta).**
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.

Dado un valor numérico, incrementar el saldo de una tarjeta.	**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.

Clase: Cine	**Método: Buscar una tarjeta dado su código.**
Signatura	Tarjeta buscarTarjeta(String pCodigo)
Precondición sobre el objeto:	El vector 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

Clase: Cine	**Método: Crear una tarjeta.**
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

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

Clase: Tarjeta	**Método: 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:	

Clase: Tarjeta	**Método: 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:	

Clase: Reserva	**Método: 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:	

Clase: Reserva	**Método: 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:	

Clase: Silla	**Método: 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 Nº 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.

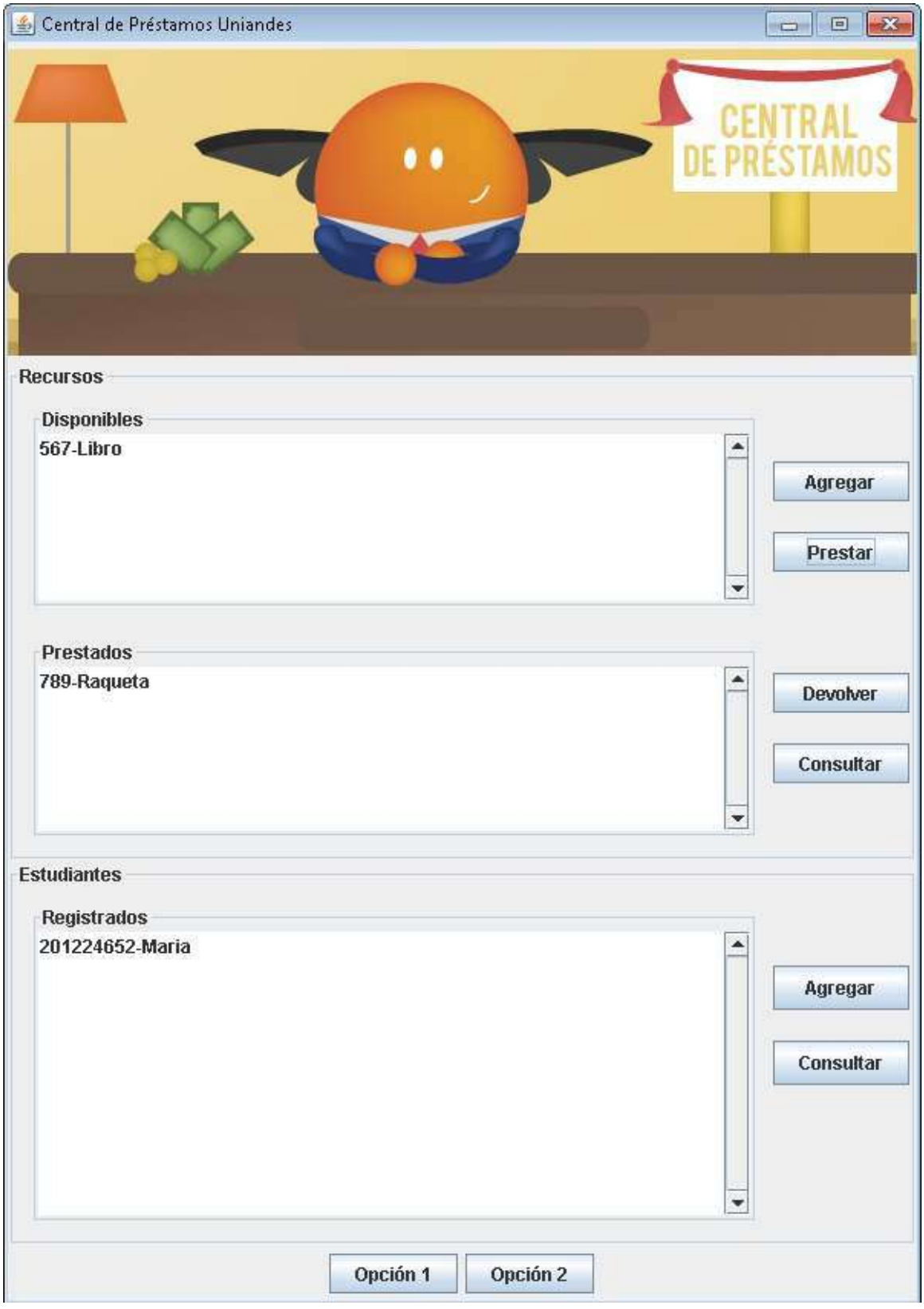
Los recursos pueden ser de cualquier naturaleza, se identifican con un código y tienen además un nombre. Los códigos son únicos, pero los nombres pueden repetirse. Cada recurso que se quiera prestar a los estudiantes debe ser registrado en la aplicación. Un recurso se puede prestar sólo si está disponible, 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

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

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.	**1. Buscar el recurso que se va a prestar. (CentralPrestamos)** **2. Validar si el recurso está disponible. (Recurso)** **3. Buscar al estudiante a quién se le prestará el recurso. (CentralPrestamos)** **4. Asignar el recurso al estudiante. (Recurso)** **5. Agregar el recurso a los recursos prestados al estudiante. (Estudiante)**
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.	**a. El código del recurso es inválido.** **b. El nombre del recurso es nulo o es una cadena vacía** **c. El código del recurso ya ha sido registrado**
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.

Clase: CentralPrestamos	**Método: 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.

Clase: Estudiante	**Método: 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:	

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

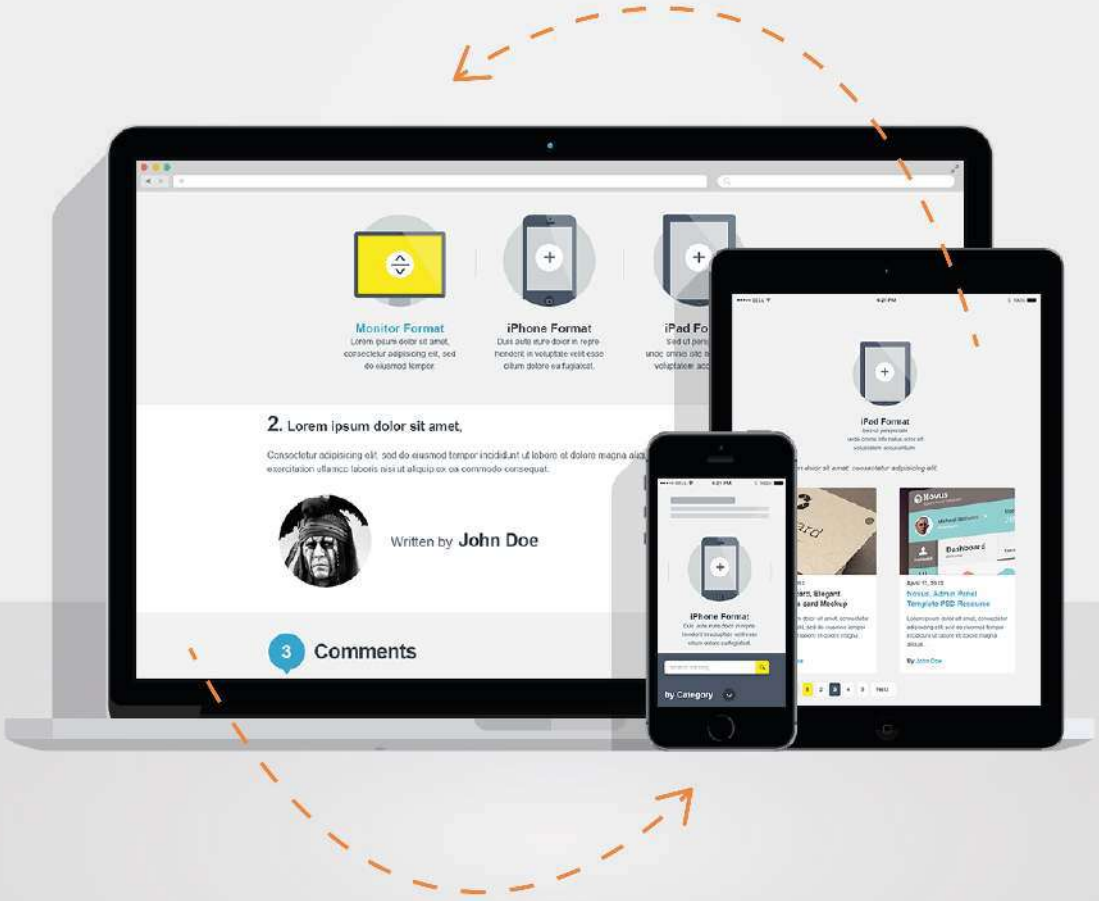
Clase: CentralPrestamos	**Método: Registrar un nuevo recurso en la central de préstamos a partir de su nombre y código.**
Signatura	
Precondición sobre el objeto:	
Precondición sobre los parámetros:	
Postcondición sobre el objeto:	
Postcondición sobre el retorno:	
Excepciones:	

Clase: CentralPrestamos	**Método: 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:	

Clase: CentralPrestamos	**Método: 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:	

CONSTRUCCIÓN DE LA INTERFAZ GRÁFICA

05



1. Objetivos Pedagógicos

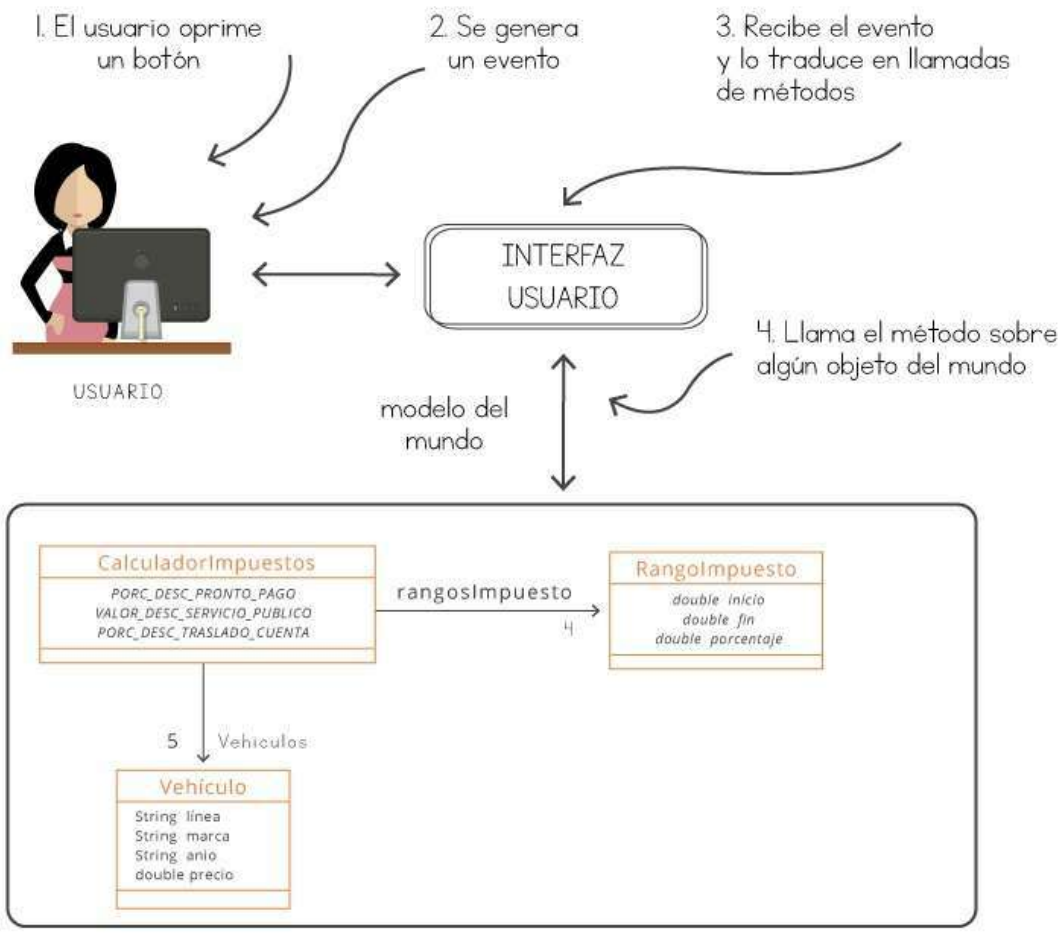
Al final de este nivel el lector será capaz de:

- Explicar la importancia de la interfaz de usuario dentro de un programa de computador, teniendo en cuenta que es el medio de comunicación entre el usuario y el modelo del mundo.
- Proponer una arquitectura para un programa simple, repartiendo de manera adecuada las responsabilidades entre la interfaz de usuario, el modelo del mundo y las pruebas unitarias. El lector deberá poder explicar la importancia de mantener separadas las clases de esos tres dominios.
- Construir las clases que implementan una interfaz de usuario sencilla e integrarlas con las clases que implementan el modelo del mundo del problema.

2. Motivación

La interfaz de usuario es el medio de comunicación entre el usuario y el modelo del mundo, tal como se sugiere en la figura 5.1. A través de la interfaz, el usuario expresa las operaciones que desea realizar sobre el modelo del mundo y, por medio de la misma interfaz, el usuario puede apreciar el resultado de sus acciones. Es un medio que permite la comunicación en los dos sentidos. La interfaz de usuario ideal es aquella en la que la persona siente que está visualizando e interactuando directamente con los elementos del modelo del mundo, y que esto se hace a través de un proceso sencillo y natural.

Fig. 5.1 La interfaz de usuario como medio de comunicación



Siempre que utilizamos un programa de computador, esperamos que sea agradable y fácil de utilizar. Aunque es difícil dar una definición precisa de lo que significa agradable, hay condiciones mínimas que influyen en esta percepción, que se relacionan con la combinación de colores, la organización de los elementos en las ventanas, los gráficos, los

tipos de letra, etc. La propiedad de facilidad de uso, por su parte, está más relacionada con el hecho de que los elementos de interacción se comporten de forma intuitiva (por ejemplo, si existe un botón con la etiqueta "cancelar" se espera que aquello que se está realizando se suspenda al oprimir este botón) y también, con la cantidad de conocimiento que el usuario debe tener para utilizar el programa.

- La interfaz es el medio para que el usuario pueda interactuar con el modelo del mundo.
- Es también una ventana para que el usuario pueda visualizar el estado del mundo.
- La interfaz debe ser amigable y fácil de usar, de manera que el usuario se sienta cómodo utilizando el programa y no cometa errores debido a cuestiones que no son claras para él.
- La interfaz debe ser capaz de interpretar las acciones de los usuarios (expresadas como eventos) y llamar los métodos que ejecutan lo que él pide.

La razón de darle importancia a este aspecto es muy sencilla: si el usuario no se siente cómodo con el programa, no lo va a utilizar o lo va a utilizar de manera incorrecta. En la mayoría de los proyectos, se dedica igual cantidad de esfuerzo a la construcción de la interfaz que al desarrollo del modelo del mundo.

Hay dos aspectos de gran importancia en el diseño de la interfaz: el primer aspecto tiene que ver con el diseño funcional y gráfico (los colores que se deben usar, la distribución de los elementos gráficos, etc.). En eso debe participar la mayoría de las veces un diseñador gráfico y es un tema que está por fuera del alcance de este libro.

El segundo aspecto es la parte de la organización de las clases que van a conformar la interfaz y, de nuevo, este aspecto tiene que ver con la asignación de responsabilidades que discutimos en el nivel anterior.

Hay muchas formas distintas de estructurar una interfaz gráfica. Podríamos, por ejemplo, construir una sola clase con todos los elementos que el usuario va a ver en la pantalla y todo el código relacionado con los servicios para recibir información, presentar información, etc. El problema de esta solución es que sería muy difícil de construir y de mantener. Un buen diseño en este caso se refiere a una estructura clara, fácil de mantener y que sigue reglas que facilitan localizar los elementos que en ella participan. Esa es la principal preocupación de este nivel: cómo estructurar la interfaz de usuario y cómo comunicarla con las clases del modelo del mundo, sin mezclar en ningún momento las responsabilidades de esos dos componentes de un programa. De algún modo las acciones del usuario se deben convertir en eventos, los cuales deben ser interpretados por algún elemento de la interfaz y traducidos en llamadas a métodos de los objetos del modelo del mundo (ver figura 5.1).

Para la construcción de las interfaces de usuario, los lenguajes de programación proveen un conjunto de clases y mecanismos ya implementados, que van a facilitar, en gran medida, el trabajo del programador. Dicho conjunto se denomina un framework o, también, biblioteca

gráfica. Construir una interfaz de usuario se convierte entonces en el uso adecuado y en la especialización de los elementos que allí aparecen disponibles. Nosotros trabajaremos en este nivel sobre swing y awt, el framework sobre el que se basan la mayoría de las interfaces gráficas escritas en Java.

3. El Caso de Estudio

En este caso de estudio queremos construir un programa que permita a una persona calcular el valor de los impuestos que debe pagar por un automóvil. Para esto, el programa debe tener en cuenta el valor del vehículo y los descuentos que contempla la ley.

Un vehículo se caracteriza por una marca (por ejemplo, Peugeot, Mazda), una línea (por ejemplo, 206, 307, Allegro), un modelo, que corresponde al año de fabricación (por ejemplo, 2016, 2017), y un precio.

Para calcular el valor de los impuestos se establecen ciertos rangos, donde cada uno tiene asociado un porcentaje que se aplica sobre el valor del vehículo. Por ejemplo, si se tiene que los vehículos con precio entre 0 y 30 millones deben pagar el 1,5% del valor del vehículo como impuesto anual, un automóvil avaluado en 10 millones debe pagar $150.000 al año. La siguiente tabla resume el porcentaje de impuestos para los cuatro rangos de valores en que han sido divididos los automóviles.

- Entre 0 y 30 millones, pagan el 1,5% de impuesto.
- Más de 30 millones y hasta 70 millones, pagan el 2,0% de impuesto.
- Más de 70 millones y hasta 200 millones, pagan el 2,5% de impuesto.
- Más de 200 millones, pagan el 4% de impuesto.

Esta tabla se debe poder cambiar sin necesidad de modificar el programa, lo cual implica que pueden aparecer nuevos rangos, modificarse los límites o cambiar los porcentajes.

En el caso que queremos trabajar, están definidos tres tipos de descuentos:

1. Descuento por pronto pago (10% de descuento en el valor del impuesto si se paga antes del 31 de marzo).
2. Descuento para vehículos de servicio público ($50.000 de descuento en el impuesto anual).
3. Descuento por traslado del registro de un automóvil a una nueva ciudad (5% de descuento en el pago).

Estos descuentos se aplican en el orden en el que acabamos de presentarlos. Por ejemplo, si el vehículo debe pagar $150.000 de impuestos, pero tiene derecho a los tres descuentos, debería pagar $80.750, calculados de la siguiente manera:

- 150.000 – 15.000 = 135.000 (Primer descuento: 150.000 * 10% = 15.000)
- 135.000 – 50.000 = 85.000 (Segundo descuento: 50.000)
- 85.000 – 4.250 = 80.750 (Tercer descuento: 85.000 * 5% = 4.250)

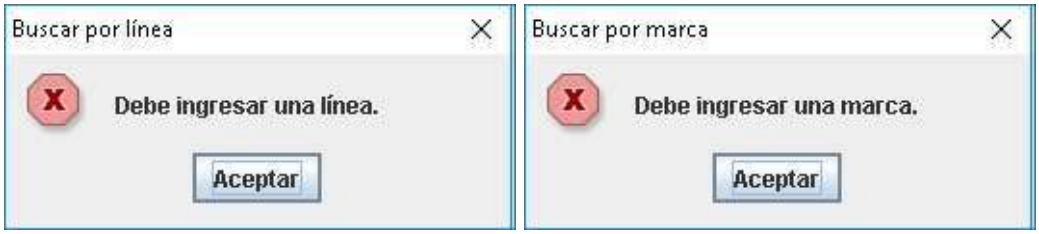
El diseño de la interfaz de la aplicación (figura 5.2) trata de organizar los elementos del problema en zonas de trabajo fáciles de entender y utilizar por el usuario. Como se puede ver, después de la la imagen con el nombre de la aplicación hay una zona que tiene como objetivo mostrar la información sobre un vehículo y permite navegar por los vehículos existentes en la aplicación. Después, hay una zona que permite buscar un vehículo por línea o marca y encontrar el vehículo más caro. Luego hay una zona que permite seleccionar los descuentos que se desean aplicar. Finalmente, en la parte inferior se tiene una zona donde se ofrece la opción de calcular el impuesto a pagar por el vehículo actual, así como 2 opciones adicionales.

Fig. 5.2 Diseño de la interfaz de usuario del caso de estudio

- La ventana del programa está dividida en cinco zonas: en la primera está la imagen con el título, en la segunda los datos del vehículo, en la tercera las opciones de búsqueda, en la cuarta la selección de descuentos, y en la quinta las opciones que provee la aplicación.
- El botón << permite visualizar el primer vehículo de la lista.
- El botón < permite visualizar el vehículo anterior.
- El boton > permite visualizar el vehículo siguiente.
- El botón >> permite visualizar el último vehículo de la lista.
- El botón Buscar por línea permite visualizar el primer vehículo que encuentre con la línea ingresada por el usuario.
- El botón Buscar por marca permite visualizar el primer vehículo que encuentre con la marca ingresada por el usuario.
- El botón Buscar vehículo más caro permite visualizar el vehículo con mayor valor.
- En la zona de descuentos, el usuario debe seleccionar los descuentos que quiere aplicar.
- El botón calcular muestra un mensaje con el valor total que debe pagar el usuario por el vehículo mostrado actualmente, incluyendo los descuentos seleccionados.
- Los botones Opción 1 y Opción 2 todavía no tienen una funcionalidad asignada.
- Fíjese que cada zona (menos la primera) tiene un borde y un título.

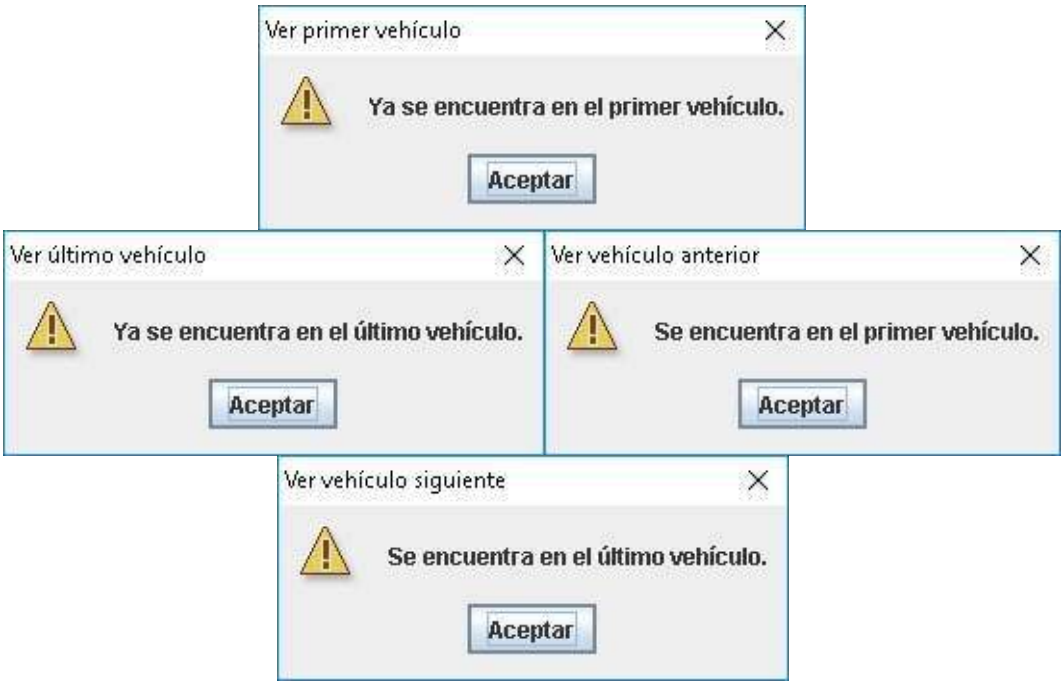
Si el usuario intenta hacer hacer una búsqueda sin haber dado la información necesaria, el programa debe informarle del problema utilizando una ventana de diálogo, como se muestra en la figura 5.3.

Fig. 5.3 Mensaje de la interfaz al usuario



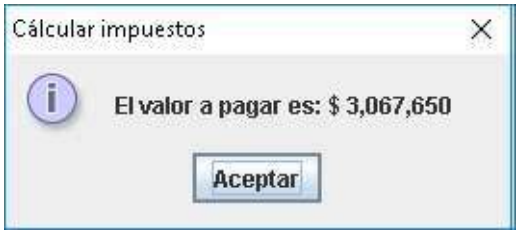
Si el usuario trata de navegar una consulta sobre un vehículo cuya información no está registrada en el programa, se debe presentar la advertencia que aparece en la figura 5.4.

Fig. 5.4 – Mensaje de la interfaz al usuario



Cuando el usuario selecciona la opción calcular, se debe presentar un mensaje con el valor a pagar, como se muestra en la figura 5.5.

Fig. 5.5 – Mensaje de la interfaz al usuario



3.1. Comprensión de los requerimientos

A partir de la descripción del caso de estudio, podemos identificar al menos seis requerimientos funcionales:

1. Navegar entre vehículos.
2. Buscar un vehículo por línea.
3. Buscar un vehículo por marca.
4. Buscar el vehículo más caro.
5. Calcular el impuesto de un carro.
6. Visualizar la información de un vehículo.

Tarea 1

Objetivo: Entender los requerimientos funcionales del caso de estudio.

Lea detenidamente el enunciado del caso de estudio, identifique los seis requerimientos funcionales y complete su documentación.

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

Nombre	
Resumen	
Entradas	
Resultado	

3.2. Comprensión del Mundo del Problema

En el modelo conceptual aparecen tres entidades con la estructura que se muestra en la figura 5.6. Dichas entidades son:

1. El calculador de impuestos (clase CalculadorImpuestos).
2. El vehículo (claseVehiculo).
3. Un rango de precios al que se le asocia un porcentaje de impuestos (clase RangoImpuesto).

Fig. 5.6 Diagrama de clases del caso de estudio

Tarea 2

Objetivo: Entender la estructura y las entidades del modelo conceptual del caso de estudio.

Lea de nuevo el enunciado del problema y estudie el diagrama de clases de UML que aparece en la figura 5.6. Para cada clase describa las constantes, los atributos y las asociaciones que aparecen en el diagrama.

Clase CalculadorImpuestos:

Constantes:

Asociaciones:

Clase Vehículo:

Atributos:

Clase RangoImpuesto:

Atributos:

3.3. Definición de los Contratos

Describimos a continuación los contratos de los principales métodos de la clase CalculadorImpuestos. Estos son los métodos que invocaremos desde la interfaz para pedir los servicios que solicite el usuario, pasándoles como parámetros la información que éste ingrese.

Método constructor:

```
/**
* Crea un calculador de impuestos, cargando la información de dos archivos.<br>
* <b>post: </b> Se inicializaron los arreglos de vehículos y rangos.<br>
*               Se cargaron los datos correctamente a partir de los archivos.<br>
* @throws Exception Si hay algún error al tratar de leer los archivos.
*/
public CalculadorImpuestos( ) throws Exception
{ ... }
```

Leyendo este contrato podemos deducir tres cosas: el constructor sabe dónde encontrar sus archivos para leerlos (no es nuestro problema definir su localización), el contenido de dichos archivos debe ser correcto (el método no va a hacer ninguna verificación interna) y si hay algún error físico de lectura de los archivos, va a lanzar una excepción que deberá atrapar quien llame al constructor.

En algún punto de la interfaz de usuario, con la instrucción que aparece a continuación, vamos a construir un objeto que representará el modelo del mundo. Dicha instrucción debe encontrarse dentro de un try-catch que nos permita atrapar la excepción que puede generarse.

```
CalculadorImpuestos calculador = new CalculadorImpuestos( );
```

Calcular pago de impuesto:

```
/**
* Calcula el pago de impuesto que debe hacer el vehículo actual.
* <b>pre:</b> Las listas de rangos y vehículos están inicializadas.
* @param pDescProntoPago  Indica  si aplica  el descuento  por  pronto  pago.
* @param pDescServicioPublico Indica si aplica el descuento por servicio público.
* @param pDescTrasladoCuenta Indica si aplica el descuento por traslado de cuenta.
* @return Valor por pagar de acuerdo con las características del vehículo y los
* descuentos que se pueden aplicar. Si no encuentra un rango para el modelo devuelve 0
.
*/
double calcularPago( boolean pDescProntoPago, boolean pDescServicioPublico, boolean pD
escTrasladoCuenta )
{ ... }
```

Este caso está orientado a la construcción de la interfaz del programa, por lo que suponemos que ya se han implementado el modelo conceptual y las pruebas unitarias del mismo.

En las próximas secciones vamos a estudiar:

1. Cómo se organizan los elementos gráficos de la interfaz de usuario en clases Java.
2. Cómo se asignan las responsabilidades.
3. Cómo se maneja la interacción con el usuario.

Todo esto se ilustrará con el programa del caso de estudio, el cual construiremos paso a paso, dando respuesta a los tres puntos planteados anteriormente.

4. Construcción de Interfaces Gráficas

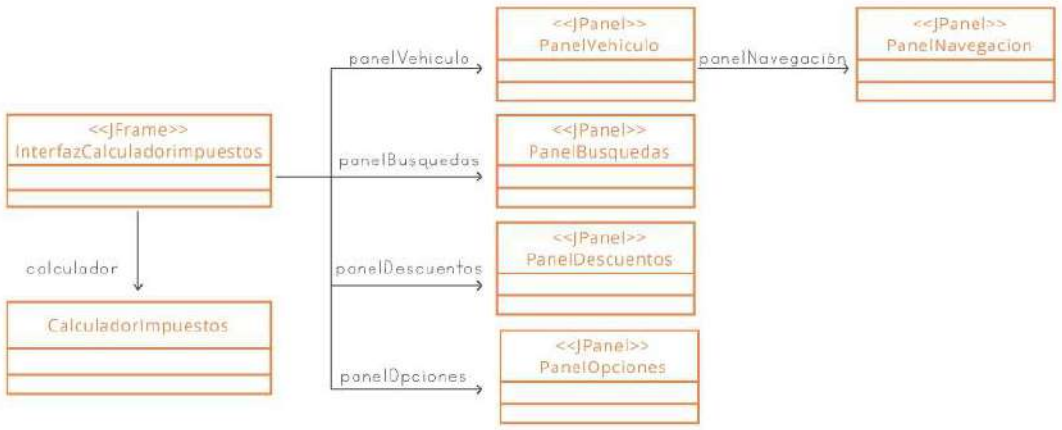
En este nivel vamos a estudiar una manera de construir interfaces de usuario para problemas pequeños. El diseño gráfico de estas interfaces incluye una ventana en la que aparece un formulario sencillo, el cual cuenta con algunos campos de edición y algunos botones para activar los requerimientos funcionales. Muchos de los elementos que se necesitan para crear una interfaz un poco más completa están por fuera del tema de este libro. El objetivo es estudiar únicamente lo indispensable para hacer una interfaz de usuario elemental. En particular, quedan por fuera todos los elementos de visualización e interacción que permiten manejar grupos de valores, de manera que, en algunos de los casos de estudio del libro, la interacción puede parecer un poco artificial.

Existen herramientas que permiten crear parcialmente el código en Java de la interfaz a partir de una descripción de la misma que se crea usando un editor gráfico. Pero en este nivel vamos a construir manualmente todos los elementos, puesto que es el único medio que tenemos de explorar a fondo la arquitectura del programa.

La buena noticia es que en la interfaz de usuario vamos a trabajar usando los mismos elementos e ideas que hemos utilizado hasta ahora. Allí vamos a encontrar clases, métodos, asociaciones, instrucciones iterativas, etc., y los vamos a expresar por medio de los mismos formalismos que hemos venido utilizando. El diagrama de clases de la interfaz, por ejemplo, se expresará en UML. La diferencia es que en lugar de trabajar con las entidades del mundo del problema, vamos a trabajar con las entidades del mundo gráfico y de interacción. En vez de tener conceptos como estudiante, tienda y banco, vamos a tener entidades como ventana, botón, campo de texto, etc. Por lo demás, es aplicar lo que ya hemos aprendido a un mundo con otro tipo de elementos.

Nuestra estrategia de diseño consiste en identificar los elementos de la interfaz que tienen un propósito común y, para cada grupo de elementos, crear una clase. Si revisamos el diseño gráfico de la interfaz en la figura 5.2 podemos ver que ésta estará compuesta de seis clases principales: una que represente la ventana principal, otra que represente la zona de información del vehículo, una zona para la navegación, una zona para las búsquedas, una zona de descuentos y la última para la zona de las opciones. En la figura 5.7 aparece el diagrama de clases de la interfaz de usuario del caso de estudio. En esa figura se muestra la asociación que existe entre una clase de la interfaz (la clase InterfazImpuestosCarro) y una clase del mundo del problema (la clase CalculadorImpuestos). Es usando dicha asociación que vamos a hacer las llamadas hacia el modelo del mundo.

Fig. 5.7 Diagrama de clases de la interfaz de usuario para el caso de estudio



El objetivo de este nivel es explicar la manera de construir el diagrama de clases de la interfaz de usuario y, posteriormente, implementarlo en Java usando swing y awt.

Hay dos aspectos prácticos que debemos tratar antes de seguir adelante: el primero es que las clases del framework swing están en el paquete `javax.swing` y en el paquete `java.awt`. Esto significa que estos paquetes o alguno de sus subpaquetes deben ser importados cada vez que se quiera incluir un elemento gráfico. El segundo aspecto es que algunas de las clases de swing han ido cambiando según la versión del lenguaje. Lo que aparece en este libro vale para las versiones posteriores a Java 5. En todo caso, la adaptación a las versiones anteriores es trivial, y en la mayoría de los casos se reduce a una simple transformación en las llamadas de los métodos.

Comencemos con la tarea 3, en la que el lector debe tratar de identificar las entidades del mundo de la interfaz.

Tarea 3

Objetivo: Identificar intuitivamente las entidades, los atributos, las asociaciones y los métodos que forman parte de una interfaz gráfica.

1. Enumere al menos 8 elementos gráficos que pueden aparecer en una interfaz gráfica cualquiera (piense en elementos como ventana, botón, etiqueta, etc.).
2. Dibuje el diagrama de clases relacionando los elementos antes identificados por medio de asociaciones.
3. Complete la descripción de cada clase con los principales atributos que debería tener.
4. Agregue al diagrama de clases las signaturas de los métodos que reflejen las principales responsabilidades de cada uno de ellos.

Identifique al menos 8 elementos de una interfaz de usuario. Piense en los elementos que forman parte de la interfaz de un programa:

Para la clase que representa la ventana principal del programa, trate de identificar sus atributos. Guíese por las características que debe tener.

Dibuje el diagrama de clases con los elementos de una interfaz de usuario cualquiera:

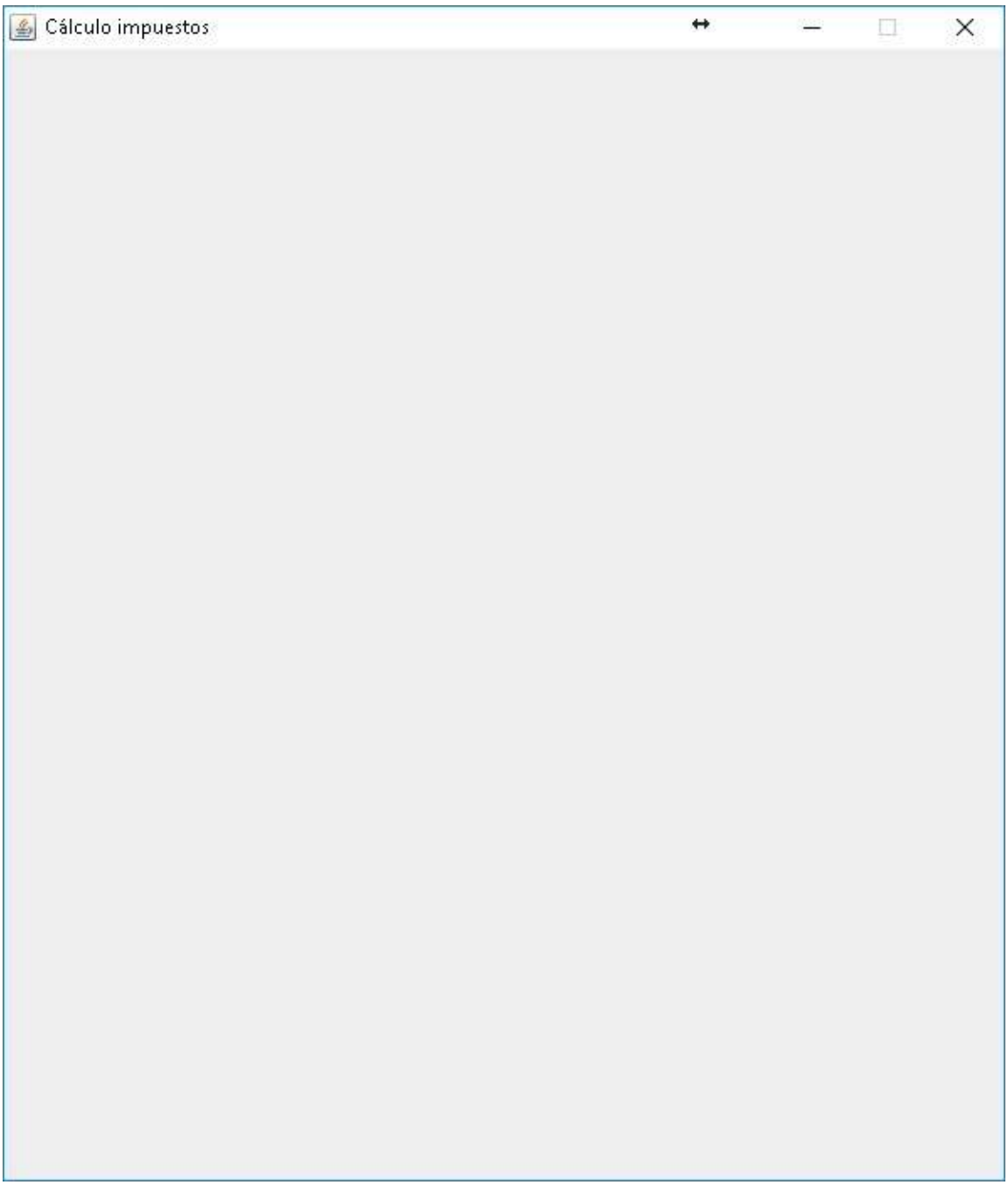
Puede usar el siguiente diagrama como guía:

5. Elementos Gráficos Estructurales

5.1. Creación de la Ventana Principal

La ventana principal de la interfaz de usuario es la encargada de contener todos los elementos de visualización e interacción, por medio de los cuales el usuario va a utilizar el programa. Su única función es servir como marco para los demás elementos de la interfaz. Típicamente, la ventana tiene en la parte superior derecha los controles para cerrar el programa, minimizar y cambiar de tamaño. Cuenta también con una zona para presentar un título, como se ilustra en la figura 5.8.

Fig. 5.8 Visualización de la ventana principal



Una ventana es el primer ejemplo de lo que se denomina un **contenedor gráfico**. Al igual que con las estructuras contenedoras que manejábamos en el modelo del mundo, un contenedor gráfico está hecho para incluir dentro de él otros elementos gráficos más sencillos: es un medio para agrupar y estructurar componentes de visualización e interacción. De alguna manera, dentro de una ventana, vamos a poder incluir las zonas de texto, los menús, los iconos, etc.

Una ventana es un objeto de una clase que se ha declarado de una manera particular (clase InterfazImpuestosCarro en el caso de estudio). Esta ventana principal va a contener la imagen con el título y cuatro de las zonas de trabajo que mencionamos antes y sus responsabilidades principales están relacionadas con la creación y organización visual de las zonas de trabajo.

La clase que representa la ventana principal (InterfazImpuestosCarro en nuestro ejemplo), al igual que cualquier clase del modelo del mundo, debe estar declarada en su propio archivo Java, siguiendo las mismas reglas definidas en los niveles anteriores. La única diferencia es que, como la clase pertenece a otro mundo distinto (el mundo gráfico), la vamos a situar en otro paquete. En el caso de estudio, por ejemplo, todas las clases de la interfaz van a estar en el paquete `uniandes.cupi2.impuestosCarro.interfaz`.

Para que la ventana principal tenga el comportamiento estándar de una ventana, como minimizarse, cerrarse o moverse cuando el usuario la arrastra, debemos indicar que nuestra clase es una extensión de una clase de un tipo particular llamado JFrame. Esta es una clase predefinida del framework swing, que tiene ya implementados los métodos para que la ventana se comporte de la manera esperada y no nos toca a nosotros, cada vez que hacemos una ventana, escribir el código para que se pueda mover, cerrar, etc.

Ejemplo 1

Objetivo: Presentar la manera de declarar en Java la clase que implementa la ventana de una interfaz de usuario.

En este ejemplo presentamos la declaración de la clase InterfazImpuestosCarro, la cual va a implementar la ventana de la interfaz para el caso de estudio. El código que se presenta en este ejemplo debe ir dentro del archivo InterfazImpuestosCarro.java, el cual se irá completando en los ejemplos de las secciones siguientes.

Al final del ejemplo estudiamos la representación de la clase en UML.

```
package uniandes.cupi2.impuestosCarro.interfaz;

import java.awt.*;
import javax.swing.*;

import uniandes.cupi2.impuestosCarro.mundo.*;

/**
* Interfaz de cálculo de impuestos de vehículos
*/
public class InterfazImpuestosCarro extends JFrame
{
  ...
}
```

- La clase se declara dentro del paquete de las clases de la interfaz de usuario.
- Se importan las clases swing de los dos paquetes indicados (swing y awt).
- Se importan las clases del modelo del mundo. Debido a que están en un paquete

distinto, es indispensable especificar su posición.
- La clase se declara con la misma sintaxis de las clases del modelo del mundo. La única diferencia es que se agrega en la declaración el término extends JFrame para indicar que es una ventana.
- JFrame es la clase en swing que implementa las ventanas.

- En UML vamos a utilizar lo que se denominan estereotipos para representar las clases de la interfaz. Eso quiere decir que, en cada clase, se hace explícita la clase del framework swing que esa clase está extendiendo.
- Al extender la clase JFrame, tenemos derecho a utilizar dentro de nuestra clase todos sus métodos.

Las preguntas ahora son dos: ¿cómo hacemos para poner los elementos gráficos dentro de una ventana? y ¿cómo hacemos para modificar sus características? La respuesta a estas dos preguntas es la misma: tenemos un conjunto de métodos implementados en la clase JFrame, que podemos utilizar para cambiar el estado de la ventana. Con estos métodos, vamos a poder cambiar el título de la ventana, su tamaño o agregar en su interior otros componentes gráficos.

Algunos de los principales métodos que podemos usar con una ventana son los siguientes (la lista completa se puede encontrar en la documentación de la clase JFrame):

- **setSize(ancho, alto):** este método permite cambiar el alto y el ancho de la ventana. Los valores de los parámetros se expresan en píxeles.
- **setResizable(cambiable):** indica si el usuario puede o no cambiar el tamaño de la ventana.
- **setTitle(titulo):** cambia el título que se muestra en la parte superior de la ventana.
- **setDefaultCloseOperation(EXIT_ON_CLOSE):** indica que la aplicación debe terminar su ejecución en el momento en el que se cierre la ventana. `EXIT_ON_CLOSE` es una constante de la clase.
- **setVisible(esVisible):** hace aparecer o desaparecer la ventana de la pantalla,

dependiendo del valor lógico que se le pase como parámetro.
- **add(componente):** permite agregar un componente gráfico a la ventana. En la siguiente sección abordaremos el tema de cómo explicarle a la ventana la "posición" dentro de ella donde queremos añadir el componente.

La configuración de las características de la ventana (tamaño, zonas, etc.) se debe hacer en el método constructor de la clase, tal como se muestra en el ejemplo 2. Lo único que nos falta en la ventana es agregar una asociación con las clases del modelo del mundo, de tal forma que sea posible traducir los eventos que genere el usuario en llamadas a los métodos que manipulan los objetos del mundo. Esto lo hacemos agregando una asociación en la ventana hacia uno o más objetos del mundo del problema.

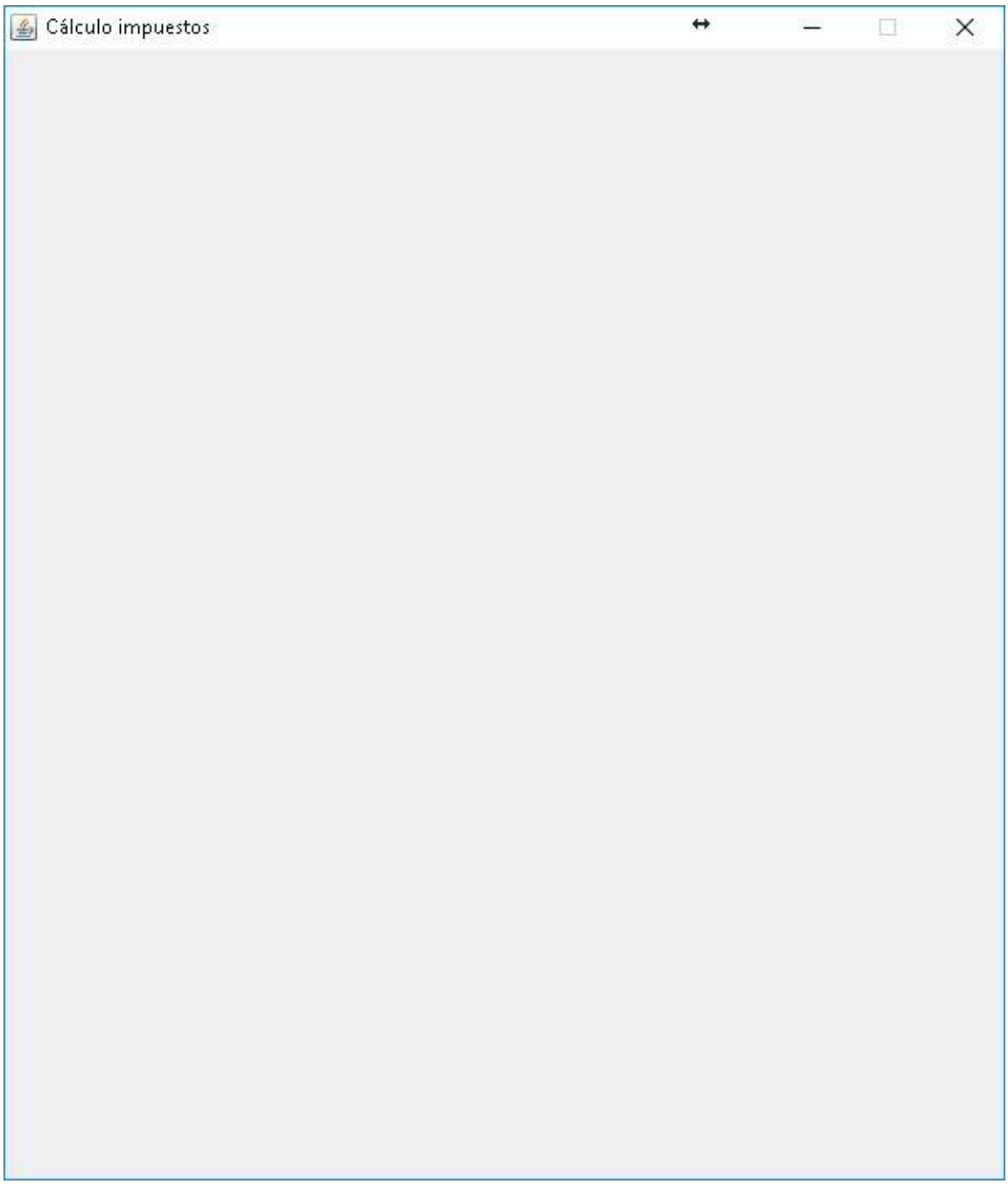
Ejemplo 2

Objetivo: Mostrar la manera de definir la configuración básica de una ventana.

En este ejemplo se muestra parte del método constructor de la clase que implementa la ventana, lo mismo que la manera de declarar un atributo para representar la asociación con el modelo del mundo.

```
public class InterfazImpuestosCarro extends JFrame
{
  private CalculadorImpuestos calculador;

  public InterfazImpuestosCarro( )
  {
    setTitle( "Cálculo impuestos" );
    setSize( 600, 700 );
    setResizable( false );
    setDefaultCloseOperation( EXIT_ON_CLOSE );
    ...
  }
}
```

- En el método constructor de la clase de la ventana definimos su configuración: título, tamaño, evitamos que el usuario la cambie de tamaño y definimos que cuando el usuario cierre la ventana el programa debe terminar.
- La llamada de los métodos se hace como si fueran de nuestra propia clase, puesto que pertenecen a la clase JFrame y nuestra clase la extiende.
- Si se incluye en la clase InterfazImpuestosCarro el constructor que aparece en este ejemplo y se ejecuta el programa, veremos aparecer en la pantalla la imagen que se muestra más abajo.



- La imagen corresponde a una ventana que tiene 600 píxeles de ancho y 700 píxeles de alto.
- Por ahora no agregamos los componentes internos de la ventana, hasta que no tratemos el tema de distribución gráfica.
- Como atributo de la ventana definimos una asociación a un objeto de la clase CalculadorImpuestos. Ya veremos más adelante cómo se inicializa y cómo lo utilizamos para implementar los requerimientos funcionales.

5.2. Distribución Gráfica de los Elementos

El siguiente problema que debemos enfrentar en la construcción de la ventana es la distribución de los componentes gráficos que va a tener en su interior. Para manejar esto, Java incluye el concepto de distribuidor gráfico (layout), que es un objeto que se encarga de hacer esa tarea por nosotros. Lo que hacemos entonces en la ventana, o en cualquier otro contenedor gráfico que tengamos, es crear y asociarle un objeto que se encargue de hacer este proceso; es decir que nosotros nos contentamos con agregar los componentes y dejamos que este objeto que creamos se encargue de situarlos en alguna parte de la ventana.

En el framework swing existe ya un conjunto de distribuidores gráficos listos para utilizar. En este nivel veremos dos de los más simples que existen, los cuales están implementados en las clases BorderLayout y GridLayout. Para asociar uno de estos distribuidores con cualquier contenedor gráfico, se utiliza el método setLayout( ), al cual se le debe pasar como parámetro una instancia de la clase que queremos que maneje la presentación gráfica de los elementos que contiene. En el caso de estudio, basta con agregarle al constructor de la ventana la siguiente llamada:

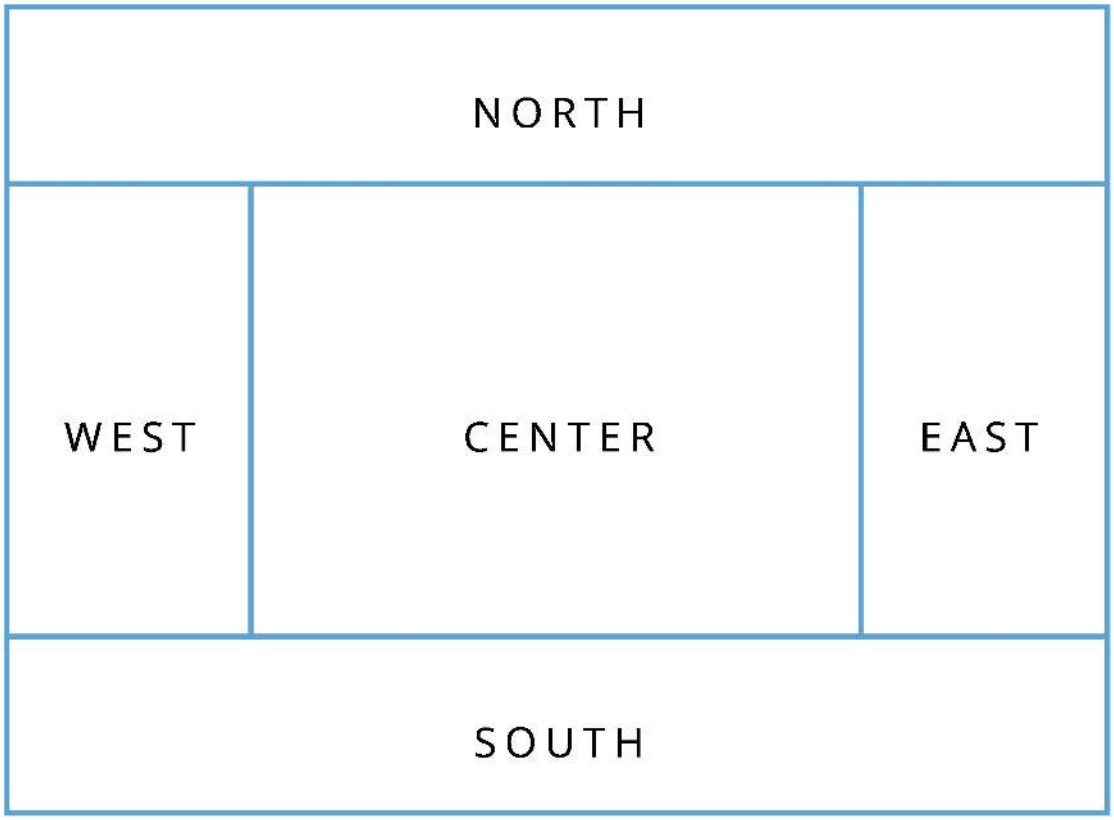
```
setLayout( new BorderLayout( ) );
```

- Si agregamos esta instrucción dentro del constructor de la ventana que vimos en el ejemplo 2, cada vez que agreguemos en ella un componente gráfico, será la instancia de la clase BorderLayout que acabamos de crear la que se encargue de situar el nuevo elemento dentro de la ventana.

5.2.1. Distribuidor en los Bordes

El distribuidor en los bordes (BorderLayout) divide el espacio del contenedor gráfico en cinco zonas, tal como muestra la figura 5.9. Cada una de ellas se identifica con una constante definida dentro de la clase (`NORTH` , `CENTER` , `SOUTH` , `WEST` , `EAST`).

Fig. 5.9 Distribuidor en los bordes (BorderLayout)



Si asociamos este distribuidor con un contenedor gráfico, cuando agreguemos un elemento deberemos pasar como parámetro la zona que queremos que éste ocupe. Por ejemplo, si quisiéramos situar un componente gráfico llamado panelVehiculo en la zona norte de la ventana del caso de estudio, deberíamos agregar en el constructor de la clase la siguiente instrucción:

```
add( panelVehiculo, BorderLayout.NORTH );
```

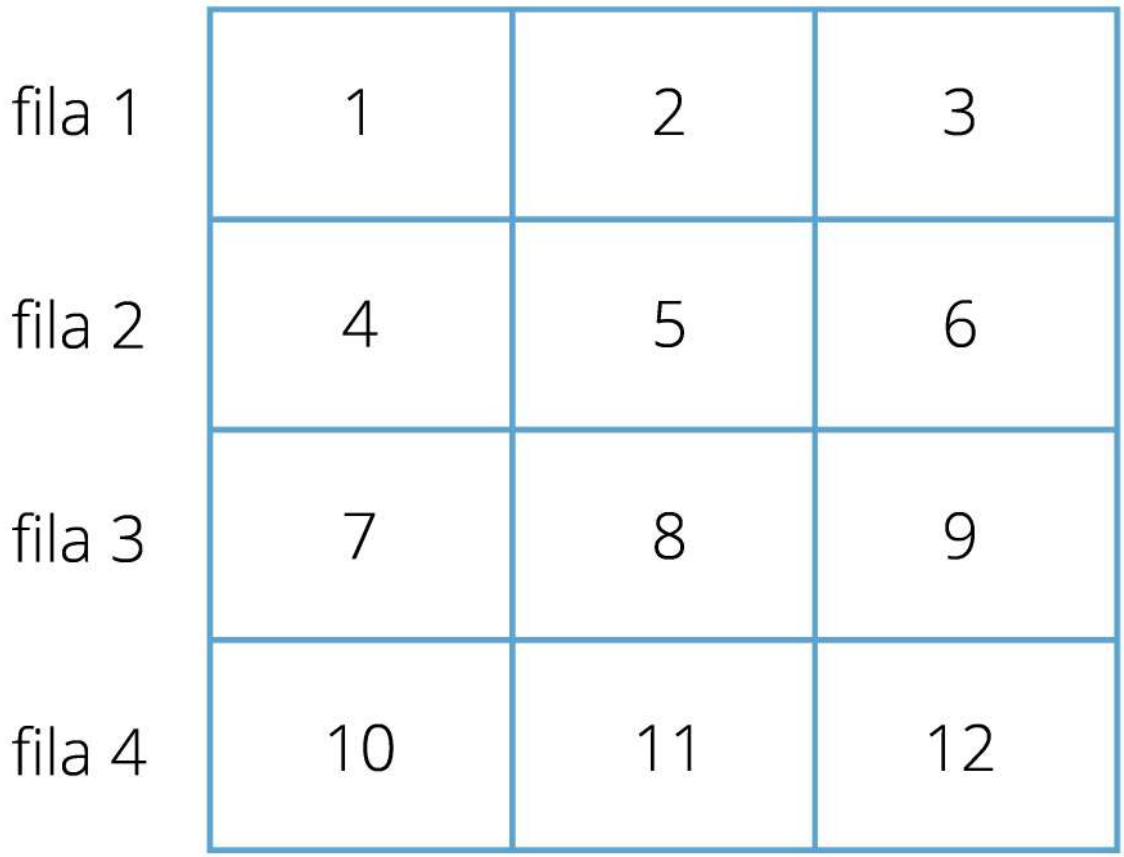
- Con esta instrucción agregamos un componente gráfico llamado panelVehiculo en la zona norte de un contenedor gráfico que tiene asociado un distribuidor en los bordes.
- Fíjese cómo se referencia la constante `NORTH` de la clase BorderLayout.

Es importante resaltar que este distribuidor utiliza el tamaño definido por cada uno de los componentes que va a albergar (cada uno tiene un ancho y un alto en pixeles) para reservarles espacio en el contenedor gráfico, y asigna todo el espacio sobrante para el componente que se encuentre en la zona del centro. Nosotros usaremos este distribuidor gráfico para construir la interfaz de usuario del caso de estudio, de manera que esto último lo veremos en detalle más adelante.

5.2.2. Distribuidor en Malla

Para usar el distribuidor en malla, se debe indicar en su constructor el número de filas y de columnas que va a tener, las cuales van a establecer las zonas en las que estará dividido el contenedor gráfico, tal como se muestra en la figura 5.10 para un distribuidor de 4 filas y 3 columnas.

Fig. 5.10 Distribuidor en malla (GridLayout) con orden de llenado



- Además de definir una estructura en filas y columnas, el distribuidor en malla define un orden de llenado.
- La primera zona que se va a ocupar es la que se encuentra en la primera columna de la primera fila (arriba a la izquierda de la ventana).
- Los componentes deben agregarse secuencialmente, siguiendo el orden de llenado del distribuidor.

Para asociar un distribuidor con un componente gráfico se utiliza la siguiente instrucción, siguiendo con el ejemplo de 4 filas y 3 columnas:

```
setLayout( new GridLayout( 4, 3 ) );
```

- Si esta instrucción se coloca en el constructor de un contenedor gráfico, todos los

elementos que se le agreguen ocuparán en orden cada una de las 12 zonas en las que está dividido.

A diferencia del distribuidor en los bordes, cuando se utiliza un distribuidor en malla no es necesario definir la posición que va a ocupar el componente que se va a incluir, porque estas posiciones son asignadas en orden de llegada: se llena primero toda la fila 1, luego la fila 2 y así sucesivamente. Este distribuidor ignora el tamaño definido para cada componente, ya que hace una distribución uniforme del espacio. En la próxima sección veremos un ejemplo de uso de este distribuidor gráfico.

5.3. Divisiones y Paneles

Dentro de la ventana principal aparecen las divisiones (o paneles), encargadas de agrupar los elementos gráficos por contenido y uso, de tal manera que sea sencillo para el usuario localizarlos y usarlos. Esta manera de estructurar la visualización del programa es muy importante, puesto que de ella depende en gran medida lo fácil e intuitivo que resulte utilizarlo. En la interfaz del caso de estudio (figura 5.2), por ejemplo, tenemos cuatro divisiones dentro de la ventana: en la primera van los datos del vehículo, en la segunda las opciones del búsqueda, en la tercera los descuentos y, en la cuarta, los botones con las opciones.

Cada división se implementa como una clase aparte en el modelo (en nuestro caso, con las clases PanelDescuentos, PanelBusquedas, PanelResultados y PanelVehiculo) y, al igual que la ventana, cada una de ellas es un contenedor gráfico al cual hay que asociarle su propio distribuidor (layout) y al cual se le pueden agregar en su interior otros componentes gráficos. En el caso del panel Vehículo, se puede observar que contiene adicionalmente el PanelNavegacion. En el constructor de la ventana se debe crear una instancia de cada una de las divisiones o paneles y luego agregarlas a la ventana. Este proceso se ilustra en el ejemplo 3.

Ejemplo 3

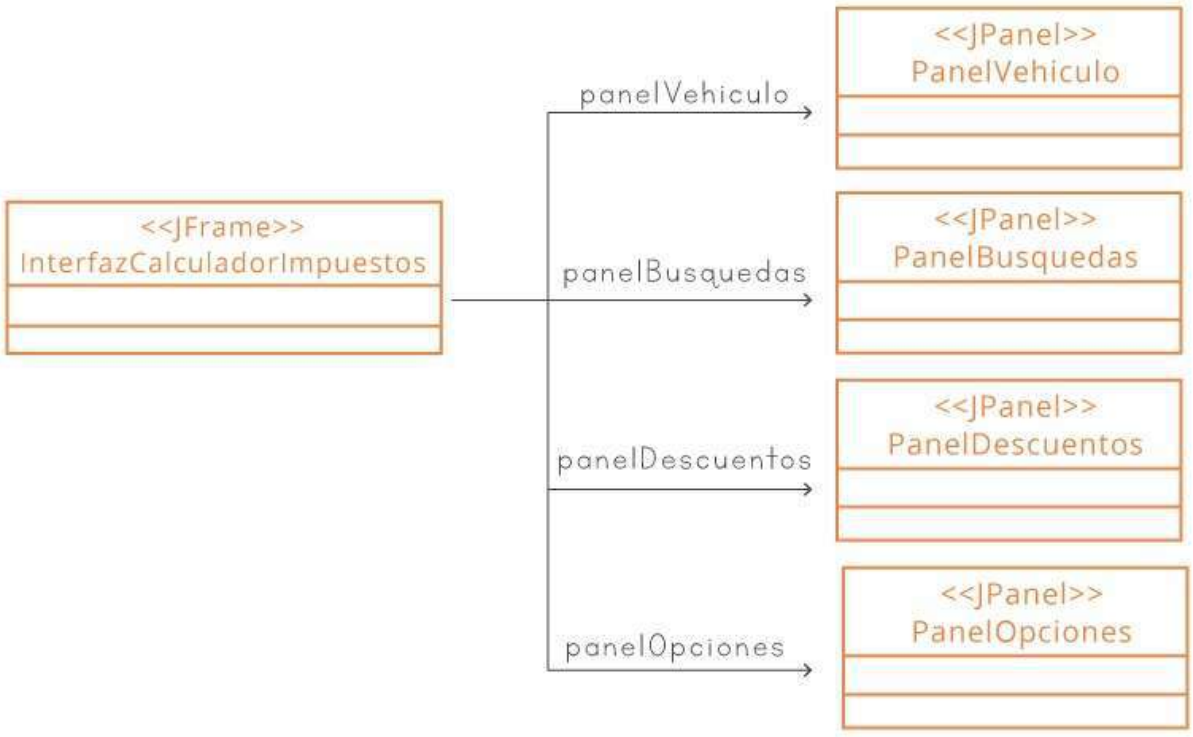
Objetivo: Mostrar la manera de agregar paneles a una ventana.

En este ejemplo se muestra el método constructor de la clase InterfazImpuestosCarro, en donde se crean las instancias de los cuatro paneles y luego se agregan a la ventana en una de las zonas del distribuidor en los bordes. Aquí se debe suponer que las clases que implementan cada una de las divisiones ya fueron creadas y sus nombres son: PanelBusqueda, PanelDescuentos, PanelOpciones y PanelVehiculo. Note que las asociaciones con los paneles se declaran como cualquier otra asociación en Java.

```
public class InterfazImpuestosCarro extends JFrame
{
    private CalculadorImpuestos calculador;

    private PanelVehiculo panelVehiculo;
    private PanelBusquedas panelBusquedas;
    private PanelDescuentos panelDescuentos;
    private PanelOpciones panelOpciones;


    public InterfazImpuestosCarro( )
    {
        setTitle( "Cálculo impuestos" );
        setSize( 290, 300 );
        setResizable( false );
        setDefaultCloseOperation( EXIT_ON_CLOSE );
        setLayout( new BorderLayout( ) );

        JPanel centro = new JPanel( );
        centro.setLayout( new BorderLayout( ) );
        add( centro, BorderLayout.CENTER );

        panelVehiculo = new PanelVehiculo( this );
        centro.add( panelVehiculo, BorderLayout.CENTER );

        panelBusquedas= new PanelBusquedas( this );
        centro.add( panelBusquedas, BorderLayout.SOUTH );

        JPanel sur = new JPanel( );
        sur.setLayout( new BorderLayout( ) );
        add( sur, BorderLayout.SOUTH );

        panelDescuentos = new PanelDescuentos( );
        sur.add( panelDescuentos, BorderLayout.CENTER );

        panelOpciones = new PanelOpciones( this );
        sur.add( panelOpciones, BorderLayout.SOUTH );

    }
}
```

- En la ventana se declara un atributo por cada una de las divisiones o paneles.
- En el constructor se asocia con la ventana un distribuidor en los bordes.
- Esta es una versión provisional del constructores, que después cambiaremos levemente a medida que vayamos conociendo nuevos elementos.
- Se crea una instancia de cada una de las divisiones y se agrega en una posición de las definidas en el distribuidor en los bordes.
- Por el momento no agregaremos nada en el norte, debido a que este espacio se reservará para la imagen con el título de la aplicación.

- Debido a que hay más paneles que divisiones disponibles, se crean dos paneles auxiliares llamados centro y sur.
- El panel con la información del vehículo va en el centro del panel centro.
- El panel para las búsquedas va en el sur del panel centro.
- El panel con la información de los descuentos va en el centro del panel sur.
- El panel con las opciones va en el sur del panel sur.
- Las zonas este y oeste quedan sin ningún componente en ellas, por lo que el distribuidor no les asigna ningún espacio en la ventana.



- Con el método constructor definido hasta el momento, hemos creado las asociaciones que se muestran en el diagrama de clases de la figura.
- Se omite la asociación hacia el modelo del mundo para concentrarnos únicamente en los elementos gráficos.

Para la construcción de las clases que representan las divisiones, se sigue un proceso muy similar al que seguimos con la ventana, ya que todas comparten el hecho de ser contenedores gráficos. Ahora, en lugar de la clase JFrame, que representa las ventanas en swing, vamos a utilizar la clase JPanel, que representa las divisiones o paneles.

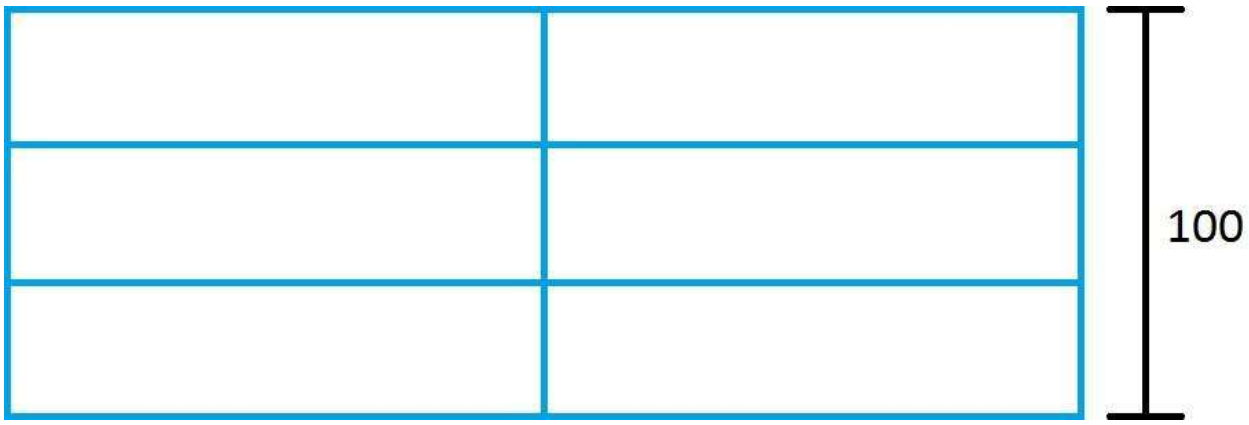
Una diferencia importante es que ahora usamos el método setPreferredSize(dimension) para definir el tamaño de las divisiones (en el ejemplo 4 se explica su utilización en más detalle). Esta información es facultativa; el distribuidor gráfico decide si hace uso de ella, si sólo la utiliza parcialmente o si sencillamente la ignora.

Ejemplo 4

Objetivo: Mostrar la manera de declarar los paneles de una ventana.

En este ejemplo se muestra la declaración en Java de las clases que implementan los paneles de la ventana principal de la interfaz de usuario en el caso de estudio.

```
public class PanelBusquedas extends JPanel
{
    public PanelBusquedas( )
    {
        setLayout( new GridLayout(3,2) );
        setPreferredSize( new Dimension( 0, 100) );
    }
}
```

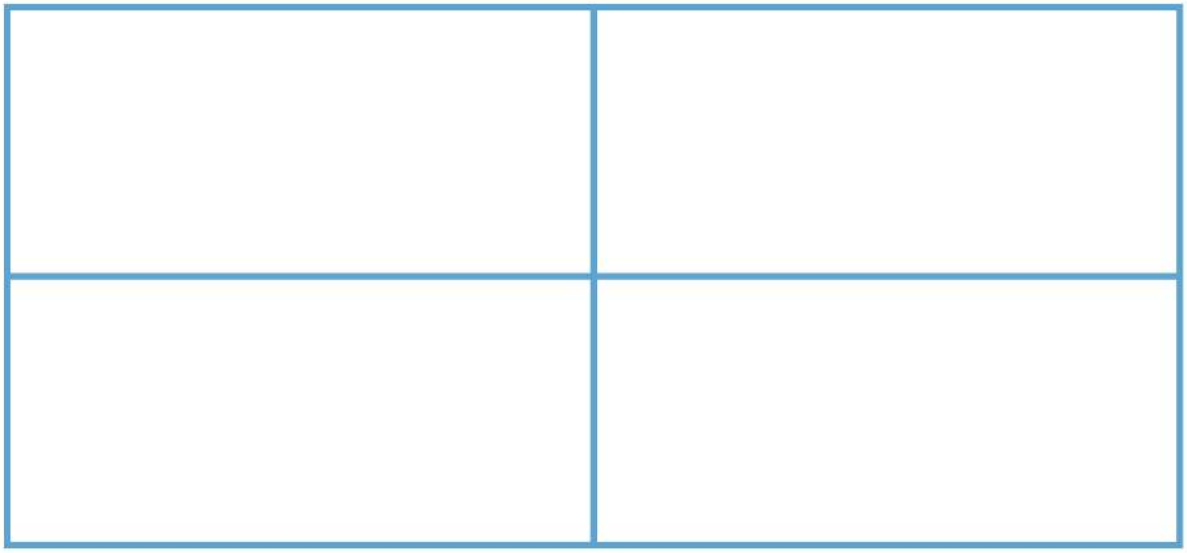


El panel con las opciones de búsqueda estará dividido en 6 zonas (3 filas y 2 columnas en cada una).

En el momento de definir la dimensión del panel es importante declarar el alto que queremos que tenga (100 píxeles), puesto que este valor es utilizado por el distribuidor en los bordes para reservarle espacio al panel, y ése es el distribuidor que usamos en la ventana principal para situar este panel en la ventana. Fíjese cómo se utiliza la clase Dimension para definir el tamaño del panel.

Al definir la dimensión del panel, pasamos 0 píxeles como ancho. Allí habríamos podido escribir cualquier valor, porque de todas maneras el distribuidor lo ignorará y le asignará como ancho todo el espacio disponible en la ventana, descontando el espacio necesario para los componentes del este y del oeste.

```
public class PanelDescuentos extends JPanel
{
    public PanelDescuentos( )
    {
        setLayout( new GridLayout( 2,2 ) );
    }
}
```



El panel con la información de los descuentos estará dividido en cuatro zonas (dos filas y dos columnas en cada una). Aquí no es importante definir la dimensión del panel, porque el distribuidor de la ventana en la cual va a estar situado le asignará todo el espacio disponible después de haber colocado los otros paneles.

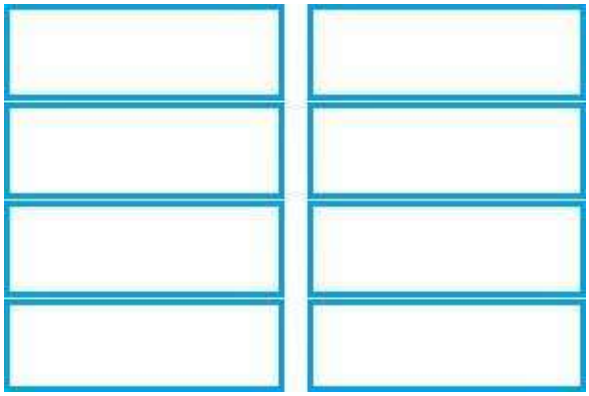
```
public class PanelOpciones JPanel
{
    public PanelOpciones ( )
    {
        setLayout( new GridLayout( 1, 3 ) );
    }
}
```

El panel con las opciones del programa estará dividido en 3 zonas(una fila y tres columnas). Aquí tampoco se debe definir la dimensión del panel.

```
public  class PanelVehiculo(  ) extends JPanel
{
    public  class PanelVehiculo(  )
    {
        setLayout( new BorderLayout( ) );

        JPanel informacion = new JPanel( );
        informacion.setLayout( new GridLayout( 4, 2, 10, 5 ) );
        add( informacion, BorderLayout.CENTER );

        PanelNavegacion panelNavegacion = new PanelNavegacion( );
        add( panelNavegacion, BorderLayout.SOUTH );
    }
}
```

El panel con la información del vehículo es más complejo que los paneles anteriores. Por esta razón, se deben utilizar nuevamente paneles auxiliares que permitan ajustar todos los elementos en esa distribución. Como podemos ver en la imagen anterior, se divide el panel en 3 zonas, una con la imagen del vehículo, otra con los datos del vehículo y otra con las opciones de navegación.

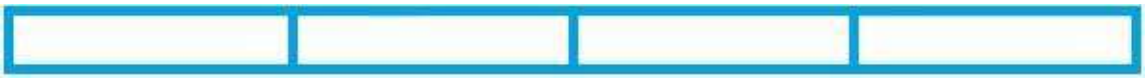
- Se usa un BorderLayout para la distribución general del panel.
- Por ahora no agregaremos nada en el oeste, debido a que este espacio se reservará para la imagen del vehículo.
- Se crea un panel auxiliar para llamado información para poner la información del vehículo, y se ubica en el centro del panel.
- El panel informacion usa un GridLayout, con 4 filas y 2 columnas. Se agregan 3 parámetros adicionales al constructor del distribuidor gráfico, para indicar los espacios, en pixeles, entre cada uno de sus zonas. En este caso, se deja un espacio de 10 pixeles entre las columnas y un espacio de 5 pixeles entre las filas.



- Para el panel de navegación, vamos a crear una nueva clase, llamada PanelNavegacion, debido a que tiene funciones diferentes que veremos más adelante.
- Las zonas este y norte quedan sin ningún componente en ellas, por lo que el distribuidor no les asigna ningún espacio en la ventana.

```
public class PanelNavegacion extends JPanel
{
    public PanelNavegacion( )
    {
        setLayout( new GridLayout( 1, 4 ) );
    }
}
```

El panel con las opciones de navegación del programa estará dividido en 4 zonas(una fila y 4 columnas).



Con esta clase completamos seis clases en el paquete de la interfaz: una para la ventana, cuatro para los paneles en los cuales la ventana está dividida y una para el panel de navegación.

La clase JPanel dispone de una amplia variedad de métodos para manejar sus propiedades. Si quiere modificar el color, por ejemplo, pruebe alguna de las siguientes instrucciones dentro del respectivo método constructor. En la documentación de la clase encontrará la lista de servicios que ofrece dicha clase.

```
setForeground( Color.RED );
setBackground( Color.WHITE );
```

Para facilitar la identificación de las divisiones dentro de la ventana, tenemos el concepto de borde, que se maneja como un objeto que se asocia con el panel. La creación de los bordes se hace de manera de la misma manera como se crean otros objetos (se utiliza el método

`new`) y la asociación con el panel se realiza de la manera que se muestra en el ejemplo 5.

Ejemplo 5

Objetivo: Mostrar la manera de crear un borde en un panel.

En este ejemplo se muestra la creación de los bordes para las tres divisiones del programa del caso de estudio. De todos los tipos de borde disponibles en swing, vamos a utilizar el borde con título, el cual permite que, además de marcar las divisiones, podamos asociar una cadena de caracteres que indique el contenido de cada uno de los paneles.

A continuación se presentan las instrucciones que se deben agregar a los métodos constructores de los paneles para asociarles los bordes necesarios. Al final se muestra la imagen de la interfaz que se ha construido hasta el momento.

```
public PanelVehiculo( )
{
  ...
  TitledBorder border = new TitledBorder( "Datos del vehículo" );
  border.setTitleColor( Color.BLUE );
  setBorder( border );
}

public PanelBusquedas( )
{
  ...
  TitledBorder border = new TitledBorder( "Búsquedas" );
  border.setTitleColor( Color.BLUE );
  setBorder( border );
}



public PanelDescuentos( )
{
  ...
  TitledBorder border = new TitledBorder( "Descuentos" );
  border.setTitleColor( Color.BLUE );
  setBorder( border );
}

public PanelOpciones( )
{
  ...
  TitledBorder border = new TitledBorder( "Opciones" );
  border.setTitleColor( Color.BLUE );
  setBorder( border );
}
```

Es conveniente utilizar alguna convención clara para nombrar las clases de los componentes gráficos. En este libro las clases que implementan las divisiones de las ventanas comenzarán por la cadena "Panel", seguidas de una descripción de su contenido. Con esta convención podemos fácilmente localizar las clases involucradas en algún aspecto de la interfaz.

Como lo vimos anteriormente, cuando en una ventana necesitamos cuatro o más divisiones en sentido vertical y horizontal y queremos utilizar el distribuidor en bordes, lo único que debemos hacer es agregar divisiones adicionales dentro de uno de los paneles, aprovechando que éstos son contenedores gráficos y pueden contener en su interior cualquier tipo de componente gráfico.

En este punto ya se tienen los conceptos indispensables para comenzar a utilizar los entrenadores de construcción de interfaces de usuario, uno de los cuales permite manipular interactivamente los distribuidores gráficos vistos en este capítulo.

5.4. Etiquetas y Zonas de Texto

Una vez que hemos terminado de estructurar las divisiones y hemos asociado con cada una de ellas un distribuidor gráfico, podemos comenzar a agregar los elementos gráficos y de interacción. Vamos a empezar por dos de los componentes gráficos más simples, que permiten una comunicación básica con el usuario: las etiquetas y las zonas de texto.

Las etiquetas permiten agregar un texto corto o imágenes como parte de la interfaz, la mayor parte de las veces con el fin de explicar algún elemento de interacción, por ejemplo una zona de texto. Las etiquetas (labels) son objetos de la clase JLabel en swing, que se crean pasando en el constructor el texto que deben contener. Dicha clase cuenta con diversos métodos, entre los cuales tenemos los siguientes:

- **setText(etiqueta):** permite cambiar el texto de la etiqueta.
- **setForeground(color):** permite cambiar el color de la etiqueta. Como color se puede pasar cualquiera de las constantes de la clase Color (`BLACK` , `RED` , `GREEN` , `BLUE` , etc.), o definir un nuevo color utilizando los tres índices ROJO-VERDE-AZUL del estándar RGB.

Para agregar una etiqueta a un panel, se siguen cuatro pasos:

1. Declarar en el panel un atributo de la clase JLabel.
2. Agregar la instrucción de creación de la etiqueta (`new`).
3. Utilizar los métodos de la clase para configurar los detalles de visualización deseados.
4. Utilizar la instrucción `add` del panel para agregarla en la zona que le corresponda.

Estos cuatro pasos son los mismos para cualquier componente gráfico que se quiera incorporar a una división. En el ejemplo 6 aparece el código necesario para crear todas las etiquetas de la interfaz del caso de estudio.

También es importante definir una convención de nombres para los atributos, que permita distinguir el tipo de componente gráfico al que corresponde. Nuestra convención es que el nombre de los atributos que representan las etiquetas comienza por la cadena "lab", mientras que aquellos que representan una zona de texto comienzan por "txt".

Las zonas de texto (objetos de la clase JTextField) cumplen dos funciones en una interfaz. Por una parte, permiten al usuario ingresar la información correspondiente a las entradas de los requerimientos funcionales (por ejemplo, la marca del vehículo) y, por otra, obtener las

respuestas calculadas por el programa (por ejemplo, el monto que se debe pagar por impuestos). Los siguientes métodos permiten configurar y manipular las zonas de texto:

- **getText( ):** retorna la cadena de caracteres ingresada por el usuario dentro de la zona de texto. Independientemente de si el usuario ingresó un valor numérico o una secuencia de letras, todo lo que el usuario ingresa se maneja y retorna como una cadena de caracteres. Más adelante veremos cómo convertirla a un número cuando así lo necesitemos.
- **setText(texto):** presenta en la zona de texto la cadena que se pasa como parámetro. Este método se usa frecuentemente para mostrar los resultados de un cálculo hecho por el programa.
- **setEditable(editable):** indica si el contenido de la zona de texto puede ser modificado por el usuario. En el caso de las zonas de texto utilizadas para mostrar resultados, es común impedir que el usuario modifique el valor allí contenido.
- **setForeground(color):** define el color de los caracteres que aparecen en la zona de texto. De la misma manera que con las etiquetas, aquí se pueden usar las constantes de la clase Color o crear otro color distinto.
- **setBackground(color):** define el color del fondo de la zona de texto.

Ejemplo 6

Objetivo: Mostrar la manera de agregar componentes gráficos simples a un panel.

Este ejemplo muestra la manera de añadir los componentes gráficos al primer panel de la interfaz del caso de estudio. Inicialmente, se presenta el contenido final esperado. Luego se muestran las instrucciones que se deben agregar al método constructor del primer panel para lograr el objetivo. Lo único que no se agrega en este momento es el panel con los botones de navegación, que es tema de una sección posterior.

```
public class PanelVehiculo extends JPanel
{
    //--------------------------------------
    // Atributos
    //--------------------------------------
    private JTextField txtMarca;
    private JTextField txtLinea;
    private JTextField txtModelo;
    private JTextField txtValor;
    private JLabel labImagen;
}
```

Paso 1: se declara un atributo en la clase por cada componente gráfico cuyo valor cambiará después de creado, que se quiera incluir en el panel.

Si vemos la imagen anterior, podemos ver que tendremos 4 etiquetas (Marca, Línea, Modelo y Valor) con texto, 4 zonas de texto asociadas y , una etiqueta con la imagen del vehículo. Como el valor de las etiquetas nunca cambia, no la agregamos como atributo.

Es conveniente asociar parejas de nombres para indicar que los componentes están relacionados entre sí. Por ejemplo los nombres `txtMarca` y `labMarca` indican que se trata de dos componentes relacionados con el mismo concepto (la marca del vehículo).

```
public PanelVehiculo( )
{
    ...
    labImagen = new JLabel( );

    JLabel labMarca = new JLabel( "Marca" );
    txtMarca = new JTextField( );

    JLabel labLinea = new JLabel( "Línea" );
    txtLinea = new JTextField( );

    JLabel labModelo = new JLabel( "Modelo" );
    txtModelo = new JTextField( );

    JLabel labValor = new JLabel( "Valor" );
    txtValor = new JTextField( );
    ...
```

Note que las campos de texto y la etiqueta con la imagen todavía no tienen ningún valor, porque este depende del vehículo que se quiera visualizar.

Paso 2: en el constructor del panel se crean los objetos que representan cada uno de los componentes gráficos.

En los constructores de algunos elementos gráficos es posible configurar algunas de las características que queremos que tenga. Estas instrucciones se escriben después de las instrucciones de definición del distribuidor gráfico y del borde.

```
    txtValor.setEditable( false );
    txtValor.setForeground( Color.BLUE );
    txtValor.setBackground( Color.WHITE );
```

En este caso, se indica el campo de texto `txtValor` no puede ser editado por el usuario, que el color del texto de la etiqueta es azul y el color de fondo blanco.

Paso 3: utilizando los métodos de cada clase se configura el componente.

Aquí sólo van las características que no hayan podido ser definidas en la creación del objeto.

```
    add( labImagen, BorderLayout.WEST );

    informacion.add( labMarca );
    informacion.add( txtMarca );
    informacion.add( labLinea );
    informacion.add( txtLinea );
    informacion.add( labModelo );
    informacion.add( txtModelo );
    informacion.add( labValor );
    informacion.add( txtValor  );
}
```

Paso 4: se añaden al panel los componentes gráficos creados. La imagen del vehículo se agrega en el oeste del panel, que es la zona que se había reservado para esto. El resto de las etiquetas y los campos de texto se agregan en el panel de información, se teniendo cuidado de agregarlos en el orden utilizado por el distribuidor gráfico (de izquierda a derecha y de arriba a abajo).

5.5. Formateo de Datos Numéricos

En algunas ocasiones, es importante formatear de manera adecuada los valores numéricos en el momento de presentárselos al usuario. Si el valor de los impuestos del vehículo actual es 1615500,120023883, debemos buscar la manera de que en la zona de texto aparezca algo del estilo "$ 1.615.500,00". Esto se logra con el código que se presenta a continuación, en el cual suponemos que en la variable precio, de tipo real, está el valor que queremos presentar y que la zona de texto en donde debe aparecer se llama txtValor:

```
DecimalFormat df = (DecimalFormat) NumberFormat.getInstance(  );

df.applyPattern( "$ ###.###,##" );
String strPrecio= df.format( precio);
txtValor.setText( strPrecio);
```

- DecimalFormat es una clase que hace este tipo de formateo. Se encuentra en el paquete java.text.
- En la primera línea se obtiene una instancia de dicha clase.
- En la segunda línea se define el formato que queremos utilizar. Marcamos con # los espacios ocupados por los dígitos que forman parte del número.
- En la tercera línea aplicamos el formato al valor que se encuentra en la variable llamada "precio".
- En la última línea colocamos la respuesta en la zona de texto llamada txtValor, utilizando el método setText().

5.6. Selección de Opciones

El framework swing provee un componente gráfico que permite al usuario seleccionar o no una opción. En el caso de estudio lo utilizamos para que el usuario seleccione los descuentos a los que tiene derecho. Con estos controles el usuario sólo puede decir "sí" o "no". El manejo de estos componentes gráficos sigue las mismas reglas explicadas en la sección anterior, tal como se muestra en el ejemplo 7.

Estos componentes son manejados por la clase JCheckBox, cuyos principales métodos son los siguientes:

- **isSelected():** retorna un valor lógico que indica si el usuario seleccionó la opción (verdadero si la opción fue escogida y falso en caso contrario).
- **setSelected(seleccionado):** marca como seleccionado o no el control, dependiendo del valor lógico del parámetro.

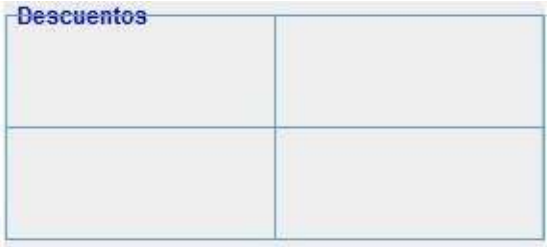
> Por convención utilizaremos el prefijo "cb" para los nombres de los atributos que representen este tipo de componentes gráficos (JCheckBox).

Ejemplo 7

Objetivo: Mostrar el manejo de los componentes de selección de opciones.

Este ejemplo muestra el manejo del componente JCheckBox en el contexto del caso de estudio. Vamos a utilizar tres objetos de esa clase en el segundo de los paneles, para que el usuario pueda escoger los descuentos a los que tiene derecho. Comenzamos mostrando

la imagen esperada en la interfaz y el distribuidor gráfico instalado sobre la división, de manera que sea claro el orden en el que los componentes se deben agregar.




```
public class PanelDescuentos extends JPanel
{
    private JCheckBox cbPPago;
    private JCheckBox cbSPublico;
    private JCheckBox cbTCuenta;
    ...
}
```

- Declaración como atributos de los tres componentes gráficos de selección de opciones.

```
public PanelDescuentos( )
{
    ...

    cbPPago = new JCheckBox( "Pronto pago" );
    cbSPublico = new JCheckBox( "Servicio público" );
    cbTCuenta = new JCheckBox( "Traslado de cuenta" );

    add( cbPPago );
    add( cbTCuenta );
    add( cbSPublico );

}
```

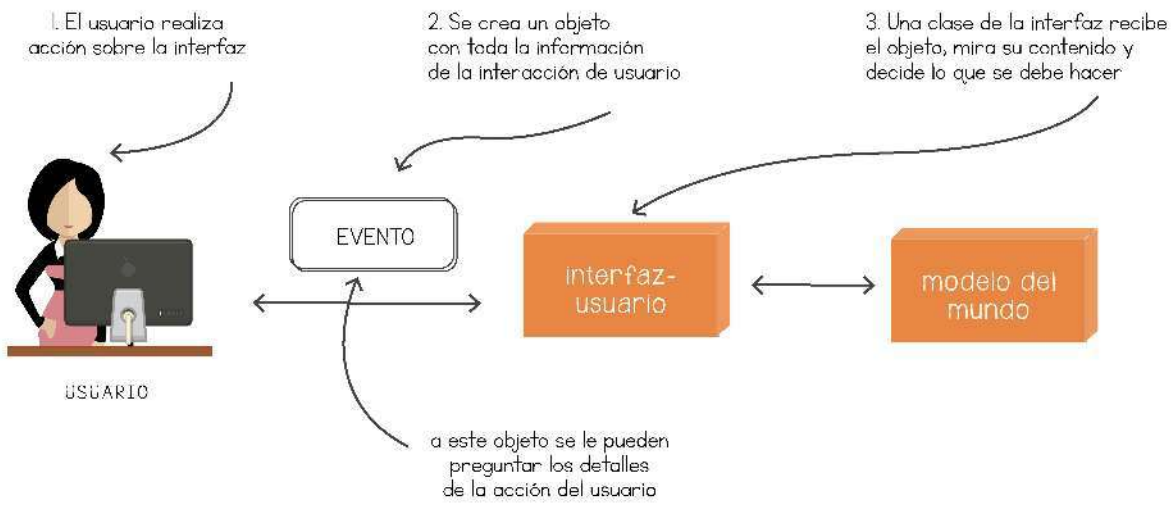
En el constructor de la clase se crean inicialmente los objetos, pasando como parámetro el nombre que se debe asociar con cada opción.

Luego se agregan los objetos al panel, siguiendo el orden pedido por el distribuidor (por filas, de arriba hacia abajo y de izquierda a derecha).

6. Elementos de Interacción

Existen muchos mecanismos de interacción mediante los cuales el usuario puede expresar sus órdenes a la interfaz. Desde hacer clic en algún punto de la ventana, hasta arrastrar un icono de una zona a otra de un panel. Todas estas acciones del usuario son convertidas en eventos en Java y son manipuladas mediante objetos. Esto quiere decir que cada vez que el usuario hace algo sobre la ventana del programa, esta acción se convierte en un objeto (llamado un evento) que contiene toda la información para describir lo que el usuario hizo. De esta manera, podemos tomar dicho objeto, estudiar su contenido y hacer que el programa reaccione como se supone debe hacerlo, de acuerdo con la acción del usuario. Por ejemplo. si en el evento aparece que el usuario oprimió un botón, debemos ejecutar la respectiva reacción, que puede incluir cambiar o consultar algo en el modelo del mundo. La figura 5.11 ilustra la idea anterior.

Fig. 5.11 Relación entre un evento y la llamada de un método



En este libro únicamente estudiamos la interacción usando botones, posiblemente el mecanismo más simple que existe para que el usuario exprese sus órdenes. Dichos botones son componentes gráficos que pertenecen a la clase JButton. Estos componentes se declaran y agregan a los paneles como cualquier otro, tal como se muestra en el ejemplo 8.

Ejemplo 8

Objetivo: Mostrar la manera de agregar botones a un panel.

En este ejemplo se presenta la manera de declarar y agregar los botones panel del caso de estudio usado para las búsquedas. Una vez instalado como aparece en el ejemplo, los botones se pueden oprimir, pero no reaccionan de ninguna manera.

Ese es el tema que sigue: ¿cómo asociar una reacción con un evento de un botón? En la imagen que se presenta a continuación aparece la visualización esperada del panel.

```
public class PanelBusquedas extends JPanel
{
    //--------------------------------------
    // Atributos
    //--------------------------------------
    private JTextField txtLinea;

    /**
     * Campo de texto para la marca.
     */
    private JTextField txtMarca;
    private JButton btnBuscarLinea;
    private JButton btnBuscarMarca;
    private JButton btnBuscarCaro;

    //--------------------------------------
    // Constructor
    //--------------------------------------
    public PanelBusquedas ( )
    {
        ...

        txtLinea = new JTextField( );
        add( txtLinea );

        btnBuscarLinea = new JButton( BUSCAR_POR_LINEA );
        add( btnBuscarLinea );

        txtMarca = new JTextField( );
        add( txtMarca );

        btnBuscarMarca = new JButton( BUSCAR_POR_MARCA );
        add( btnBuscarMarca );

        add( new JLabel( ) );

        btnBuscarCaro = new JButton( BUSCAR_MAS_CARO );
        add( btnBuscarCaro );
    }
}
```

- Se declara un atributo por cada componente botón: dos zona de texto para ingresar lo que se desea buscar y tres botones.
- El prefijo utilizado para los botones en este ejemplo es "btn".
- Las instrucciones que aquí se muestran deben venir después de aquellas que asocian el distribuidor gráfico y el borde.
- Se crean los objetos que implementan los componentes gráficos y se inicializan.
- Al crear un botón, se define la etiqueta que va a aparecer sobre él.
- Fíjese que agregamos una etiqueta "vacía" para obtener la visualización deseada.

Hay tres pasos que se deben seguir para decidir la manera de manejar un evento con un botón de la interfaz, los cuales se explican a continuación:

- Decidir el nombre del evento. A los eventos de los botones se les asocia un nombre por medio del cual se van a poder identificar más adelante. El nombre es una cadena de caracteres y es muy conveniente definir dicha cadena como una constante. Para el caso de estudio, los nombres de los eventos se asocian de la siguiente manera con los dos botones:

```
public class PanelBusquedas extends JPanel
{
    //--------------------------------------
    // Constantes
    //--------------------------------------
    public final static String BUSCAR_POR_LINEA = "Buscar por línea";
    public final static String BUSCAR_POR_MARCA = "Buscar por marca";
    public final static String BUSCAR_MAS_CARO = "Buscar vehículo más Caro";

    public PanelBusquedas ( )
    {
        ...
        btnBuscarLinea.setActionCommand( BUSCAR_POR_LINEA );
        btnBuscarMarca.setActionCommand( BUSCAR_POR_MARCA );
        btnBuscarCaro.setActionCommand( BUSCAR_MAS_CARO );
    }
}
```

- Implementar el método que va a atender el evento. Para atender el evento, el panel que contiene el botón debe agregar una declaración en el encabezado de la clase (`implements ActionListener`) e implementar un método especial llamado actionPerformed, que recibe como parámetro el evento ocurrido en el panel. Dicho evento es un objeto de la clase ActionEvent. Estos puntos se ilustran en el siguiente código, en el cual se muestra también la manera de obtener el nom- bre del evento ocurrido, a partir del objeto que lo representa.

```
public class PanelBusquedas extends JPanel implements ActionListener
{

    public void actionPerformed( ActionEvent pEvento )
    {
        String comando = evento.getActionCommand( );

        if( comando.equals( BUSCAR_MAS_CARO ) )
        {
            // Reacción al evento de BUSCAR_MAS_CARO
        }
        else if( comando.equals( BUSCAR_POR_LINEA ) )
        {
            // Reacción al evento de BUSCAR_POR_LINEA
        }
        else if( comando.equals( BUSCAR_POR_MARCA ) )
        {
            // Reacción al evento de BUSCAR_POR_MARCA

        }
    }
}
```

- La clase del panel debe incluir en su encabezado la declaración implements ActionListener.
- Esa misma clase debe implementar un método con la signatura planteada en el ejemplo.
- Con el método getActionCommand podemos saber el nombre del evento ocurrido.

Cada vez que el usuario oprime un botón en un panel, se ejecuta su método actionPerformed. El contenido exacto de dicho método se estudiará en una sección posterior, puesto que hay decisiones de nivel de arquitectura que todavía no hemos tomado. Pero a grandes rasgos se puede decir que ese método debe utilizar el nombre del evento ocurrido para decidir la acción que debe tomar.

- Declarar que el panel es el responsable de atender los eventos de sus botones. Para esto se utiliza el método addActionListener, pasando como referencia el panel. Puesto que esto se debe ejecutar en el constructor del mismo panel, utilizamos la variable this que provee el lenguaje Java para hacer referencia al objeto que está ejecutando un método. De esta manera podemos decir dentro del constructor del panel que quien va a atender los eventos del botón es el panel mismo. El código es el siguiente:

```
public class PanelBusquedas extends JPanel implements ActionListener
{
    public PanelResultados( )
    {
        ...

        btnBuscarLinea.addActionListener( this );
        btnBuscarMarca.addActionListener( this );
        btnBuscarCaro.addActionListener( this );
    }
}
```

- Con el método addActionListener, el botón declara que es el panel quien va a atender sus eventos.
- La variable this siempre referencia al objeto que está ejecutando un método.

Con eso completamos el manejo de eventos relacionados con los botones y sólo queda pendiente el cuerpo exacto del método que atiende los eventos.

A continuación mostramos el contenido completo de la clase PanelBusquedas , para dar una visión global de su contenido:

```
public class PanelBusquedas extends JPanel implements ActionListener
{
    // -----------------------------------------------------------------
    // Constantes
    // -----------------------------------------------------------------
    public final static String BUSCAR_POR_LINEA = "Buscar por línea";
    public final static String BUSCAR_POR_MARCA = "Buscar por marca";
    public final static String BUSCAR_MAS_CARO = "Buscar vehículo más Caro";

    // -----------------------------------------------------------------
    // Atributos de la interfaz
    // -----------------------------------------------------------------

    private JTextField txtLinea;
    private JTextField txtMarca;
    private JButton btnBuscarLinea;
    private JButton btnBuscarMarca;
    private JButton btnBuscarCaro;

    // -----------------------------------------------------------------
    // Constructores
    // -----------------------------------------------------------------

    public PanelBusquedas( InterfazImpuestosCarro pPrincipal )
    {
        principal = pPrincipal;
        setLayout( new GridLayout( 3, 2 ) );
        TitledBorder border =  new TitledBorder( "Búsquedas" );
```

```
        border.setTitleColor( Color.BLUE );
        setBorder( border );
        setBorder( border);

        txtLinea = new JTextField( );
        add( txtLinea );

        btnBuscarLinea = new JButton( BUSCAR_POR_LINEA );
        btnBuscarLinea.addActionListener( this );
        btnBuscarLinea.setActionCommand( BUSCAR_POR_LINEA );
        add( btnBuscarLinea );

        txtMarca = new JTextField( );
        add( txtMarca );

        btnBuscarMarca = new JButton( BUSCAR_POR_MARCA );
        btnBuscarMarca.addActionListener( this );
        btnBuscarMarca.setActionCommand( BUSCAR_POR_MARCA );
        add( btnBuscarMarca );

        add( new JLabel( ) );

        btnBuscarCaro = new JButton( BUSCAR_MAS_CARO );
        btnBuscarCaro.addActionListener( this );
        btnBuscarCaro.setActionCommand( BUSCAR_MAS_CARO );
        add( btnBuscarCaro );

    }

    public void actionPerformed( ActionEvent pEvento )
    {
        String comando = pEvento.getActionCommand( );

        if( comando.equals( BUSCAR_MAS_CARO ) )
        {
            // Por definir

        }
        else if( comando.equals( BUSCAR_POR_LINEA ) )
        {
            // Por definir

        }
        else if(comando.equals( BUSCAR_POR_MARCA ) )
        {
            // Por definir
        }
    }
}
```

Si una clase incluye la declaración implements ActionListener y no implementa el método actionPerformed (o si lo implementa con otra signatura), se obtiene el siguiente error de compilación:

> Class must implement the inherited abstract method ActionListener.
> actionPerformed(ActionEvent)

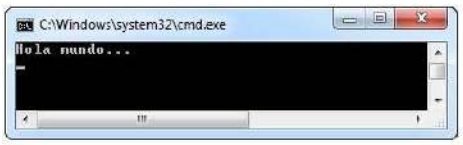
7. Mensajes al Usuario y Lectura Simple de Datos

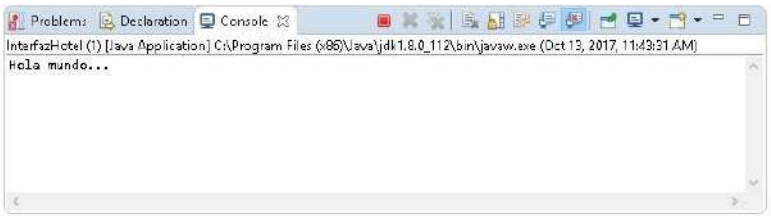
7.1. Mensajes en la Consola

Para presentar un mensaje en la ventana de comandos del sistema operativo, se utiliza la instrucción `System.out.println(cadena)`. Es poco usual enviarle mensajes al usuario a esa ventana, pero en algunos casos (errores fatales, por ejemplo), esto es indispensable.

Si el programa se está ejecutando en un ambiente de desarrollo como Eclipse, los mensajes aparecerán en una ventana especial llamada consola.

```
System.out.println( "Hola mundo..." );
```





Se puede utilizar esta instrucción, en cualquier clase de la interfaz, para enviar un mensaje al usuario a la ventana de comandos del sistema operativo.

Si durante la ejecución de un programa se lanza una excepción que no es atrapada por ninguna clase de la interfaz, la acción por defecto es generar una secuencia de mensajes en la ventana de comandos, con información relativa al error.

7.2. Mensajes en una Ventana

El paquete swing incluye una clase JOptionPane que, entre sus múltiples usos, tiene un método para enviarle mensajes al usuario en una pequeña ventana emergente. Esto es muy útil en caso de error en las entradas del usuario o con el fin de mostrarle un resultado puntual de una consulta. La sintaxis de uso es la que se muestra en el ejemplo 9.

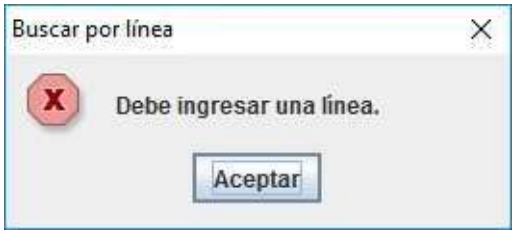
Ejemplo 9

Objetivo: Mostrar la manera de presentar un mensaje a un usuario, usando una ventana simple de diálogo.

Este ejemplo muestra la manera de enviarle mensajes al usuario, abriendo una nueva ventana y esperando hasta que el usuario oprima el botón para continuar. En cada imagen aparece la ventana que se va a mostrar al usuario y, debajo, la instrucción que ordena hacerlo. Esta instrucción debe ir dentro de un método de un panel.

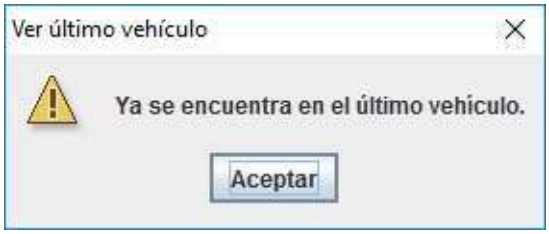
Mensaje de error:

```
JOptionPane.showMessageDialog( principal, "Debe ingresar una línea.", "Buscar por línea", JOptionPane.ERROR_MESSAGE );
```



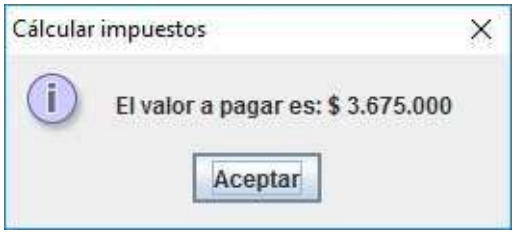
Mensaje de advertencia:

```
JOptionPane.showMessageDialog( this , "Ya se encuentra en el último vehículo.", "Ver último vehículo" , JOptionPane.WARNING_MESSAGE );
```



Mensaje de información:

```
JOptionPane.showMessageDialog( this , "El valor a pagar es: $3.675.000" , "Cálculo de Impuestos" , JOptionPane.INFORMATION_MESSAGE );
```



7.3. Pedir Información al Usuario

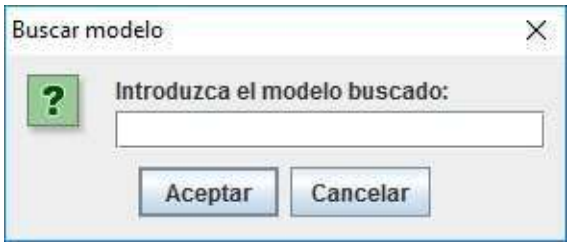
Cuando la información que se necesita como entrada de un requerimiento funcional es muy sencilla (un nombre o un valor numérico), se puede utilizar un método de la clase JOptionPane que abre una ventana de diálogo y luego retorna la cadena tecleada por el usuario. Su uso se ilustra en el ejemplo 10. Si la información que se necesita de parte del usuario es más compleja, se debe utilizar un cuadro de diálogo más elaborado, en el cual irían los componentes gráficos necesarios para recoger la información.

Ejemplo 10

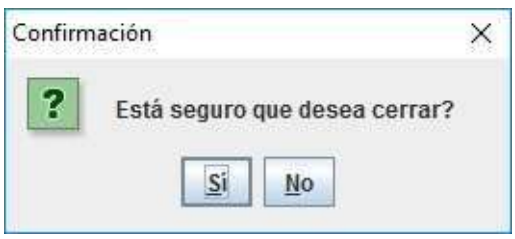
Objetivo: Mostrar la manera de pedir información simple al usuario.

Este ejemplo muestra la manera de pedir al usuario que teclee alguna información en una nueva ventana de diálogo. Al igual que en el ejemplo anterior, en cada imagen aparece la visualización de la ventana y, debajo, el código en Java que la presenta y que recupera el valor tecleado. Esta instrucción debe ir dentro de un método de un panel o ventana.

```
String strModelo = JOptionPane.showInputDialog( this , "Introduzca el modelo buscado:"
, "Buscar modelo", JOptionPane.QUESTION_MESSAGE );

if( strModelo != null )
{
  // el usuario tecleó algo
}
```




```
int resp = JOptionPane.showConfirmDialog( this , "Está seguro que desea cerrar?" , "Co
nfirmación" , JOptionPane.YES_NO_OPTION ); );

if( resp == JOptionPane.YES_OPTION )
{
  // el usuario seleccionó Sí
}
```



7.4. Validación y formateo de datos

Cuando el usuario ingresa alguna información, la interfaz tiene muchas veces la responsabilidad de convertirla al formato y al tipo adecuados para poder manipularla (por ejemplo, convertir una cadena en una variable de tipo entero o pasar una cadena a minúsculas). De la misma manera, si el usuario tecleó un contenido que no corresponde a lo esperado (ingresó una letra cuando se esperaba un número), la interfaz debe advertir al usuario de su error. Vamos entonces por partes para ver cómo manejar cada uno de los casos.

Para convertir una cadena de caracteres (que sólo contenga dígitos) en un número, se utiliza el método de la clase Integer llamado parseInt, usando la sintaxis que se muestra a continuación. Dicho método lanza una excepción cuando la cadena que se pasa como parámetro no se puede convertir en un valor entero. En la sección de recuperación de la excepción (sección catch) podría incluirse un mensaje al usuario.

```
String strModelo = JOptionPane.showInputDialog( this , "Introduzca el modelo buscado:"
, "Buscar por modelo", JOptionPane.QUESTION_MESSAGE );

if( strModelo != null )
{
    try
    {
        int nModelo = Integer.parseInt( strModelo );
    }
    catch( Exception e )
    {
        JOptionPane.showMessageDialog( principal, "Debe ingresar un valor numérico.",
"Buscar por modelo", JOptionPane.ERROR_MESSAGE );
    }
}
```

- Suponga que queremos convertir el modelo del vehículo que ingresó el usuario en el valor entero correspondiente (si ingresó la cadena "2016", queremos obtener el valor entero 2016).
- Lo primero que hacemos es tomar la cadena de caracteres ingresada por el usuario en el JInputDialog.
- Luego intentamos convertir dicha cadena en el valor entero correspondiente.
- En este ejemplo, si se produce una excepción, le presentamos un mensaje al usuario indicándolo .
- Este esquema de conversión es típico de las interfaces gráficas, puesto que no estamos seguros del tipo de datos de lo que ingresó el usuario y, en algunos casos, es conveniente verificarlo antes de continuar.

La clase String, por su parte, nos ofrece los siguientes métodos para transformar la cadena tecleada por el usuario:

- **toLowerCase( ):** convierte todos los elementos de una cadena de caracteres a minúsculas.
- **toUpperCase( ):** convierte todos los elementos de una cadena de caracteres a mayúsculas.
- **trim( ):** elimina todos los caracteres en blanco del comienzo y el final de la cadena.

En la siguiente tabla se muestran algunos ejemplos del uso de los métodos anteriores:

String ejemplo = " La Casa ";	**// valor inicial de la cadena**
String minusculas = ejemplo.toLowerCase( );	minusculas.equals(" la casa ")
String mayusculas = ejemplo.toUpperCase( );	mayusculas.equals(" LA CASA ")
String sinBlancos = ejemplo.trim( );	sinBlancos.equals("La Casa")

8. Arquitectura y Distribución de Responsabilidades

8.1. ¿Por dónde Comienza la Ejecución de un Programa?

Un método que no hemos mencionado hasta ahora y que, sin embargo, es el punto por donde comienza siempre la ejecución de un programa, es el método main(). Este método se implementa en la clase de la ventana principal del programa y tiene la sintaxis que se muestra a continuación. Su principal tarea es crear una instancia de la ventana y hacerla visible en la pantalla.

```
//-----------------------------------------------------
// Programa principal
//-----------------------------------------------------
public static void main( String[] args )
{
    try
    {
        InterfazImpuestosCarro vent = new InterfazImpuestosCarro( );
        vent.setVisible( true );
    }
    catch( Exception e )
    {
        JOptionPane.showMessageDialog( null, e.getMessage(), "Calculador impuestos", J
OptionPane.ERROR_MESSAGE);
    }
}
```

- Este método debe ir en la clase que implementa la ventana principal. Su objetivo es establecer la manera de comenzar la ejecución del programa, creando una instancia de la ventana y haciéndola visible.
- La signatura del método debe ser idéntica a la que aparece en el ejemplo.

8.2. ¿Quién Crea el Modelo del Mundo?

La responsabilidad de crear el modelo del mundo (los objetos que lo van a representar) es de la interfaz. En la arquitectura que presentamos en este libro, nosotros asignamos esta responsabilidad al constructor de la ventana principal. Allí se deben realizar todas las

acciones necesarias para que los objetos del modelo del mundo (uno o varios) sean creados, inicializados y almacenados en atributos de dicha clase. A continuación se muestra, para el caso de estudio, la creación del objeto que representa el calculador de impuestos.

```
public class InterfazImpuestosCarro extends JFrame
{
    //-----------------------------------------------
    // Atributos
    //-----------------------------------------------
    private CalculadorImpuestos calculador;

    private PanelVehiculo panelVehiculo;
    private PanelDescuentos panelDescuentos;
    private PanelOpciones panelOpciones;
    private PanelBusquedas panelConsultas;

    //-----------------------------------------------
    // Constructor
    //-----------------------------------------------
    public InterfazImpuestosCarro( ) throws Exception
    {
        calculador = new CalculadorImpuestos( );
        ...
    }
}
```

- En este caso la creación es simple, pues el constructor del calculador de impuestos tiene la responsabilidad de abrir los archivos con la información y crear los objetos necesarios para representarla.
- Definimos un atributo de la clase CalculadorImpuestos, y allí guardamos la asociación que nos va a permitir "hablar" con el modelo del mundo (ver diagrama de clases).
- En el constructor dejamos pasar las excepciones generadas en la construcción del modelo del mundo. Dejamos al programa principal la responsabilidad de atraparlas y enviarle el mensaje respectivo al usuario.
- No poder construir el modelo del mundo (p.ej. no poder abrir los archivos con los valores que utiliza la calculadora) lo consideramos un error fatal, y por esa razón no existe ninguna manera de recuperarse.
- Se puede ver la ventana principal como la clase que va a coordinar el trabajo entre los paneles y el modelo del mundo.

8.3. ¿Qué Métodos Debe Tener un Panel?

Una pregunta que debemos responder en este punto es cuáles son los métodos que debe tener un panel, ya que hasta este momento sólo tenemos un método constructor y un método para atender los eventos. La respuesta es que los paneles tienen dos grandes responsabilidades además de las ya estudiadas:

1. Proveer los métodos indispensables para permitir el acceso a la información tecleada por el usuario. Considere la interfaz de usuario del caso de estudio, en la cual en el primer panel está la información del vehículo. Puesto que el tercer panel va a necesitar esta información para poder calcular los impuestos, es responsabilidad del panel que tiene la información proveer un conjunto de métodos que garantice que aquellos que requieran la información puedan tener acceso a ella. Eso no quiere decir que haya que construir un método por cada zona de texto. Lo que quiere decir es que se debe establecer qué información se necesita manejar desde fuera del panel y crear los métodos respectivos. El programador debe decidir si estos métodos son responsables de hacer las conversiones o si esta labor se deja a aquellos que van a utilizar la información.
2. Proveer los métodos para refrescar la información presentada en el panel. Si en un panel se presenta información que depende del estado del modelo del mundo, debemos implementar los servicios necesarios para poder actualizarla. Por ejemplo, en el panel de información del vehículo, debemos tener un método que pueda modificar la información del vehículo actual. Estos métodos se denominan de **refresco** y su objetivo es permitir actualizar el contenido de los componentes gráficos del panel. El ejemplo 11 ilustra esta responsabilidad.

Ejemplo 11

Objetivo: Mostrar los métodos que debe implementar un panel, para prestar servicios a los demás elementos de la interfaz.

Este ejemplo muestra los métodos de acceso a la información y de refresco para la clase que implementa el panel con la información del vehículo.

```
public class PanelVehiculo extends JPanel implements ActionListener
{
    //-----------------------------------------------
    // Métodos de refresco
    //-----------------------------------------------
    public void actualizar( String pMarca, String pLinea, String pAnio, double pPrecio
, String pRutaImagen)
    {
        txtMarca.setText( pMarca );
        txtLinea.setText( pLinea );
        txtModelo.setText( pAnio );
        DecimalFormat df = ( DecimalFormat )NumberFormat.getInstance( );
        df.applyPattern( "$ ###,###.##" );
        txtValor.setText( df.format( pPrecio ) );
        labImagen.setIcon( new ImageIcon( new ImageIcon( "./data/imagenes/" + pRutaIma
gen ).getImage( ).getScaledInstance( 280, 170, Image.SCALE_DEFAULT ) ) );
    }
}
```

La clase tiene un método de refresco que permite cambiar la información del vehículo. De esta manera, utilizando el método `setText()` actualiza la información de los campos de texto y con el método `setIcon()` cambia la imagen mostrada en la etiqueta labImagen.

8.4. ¿Quién se Encarga de Hacer Qué?

Si recapitulamos lo que llevamos hasta este momento, podemos decir que ya sabemos:

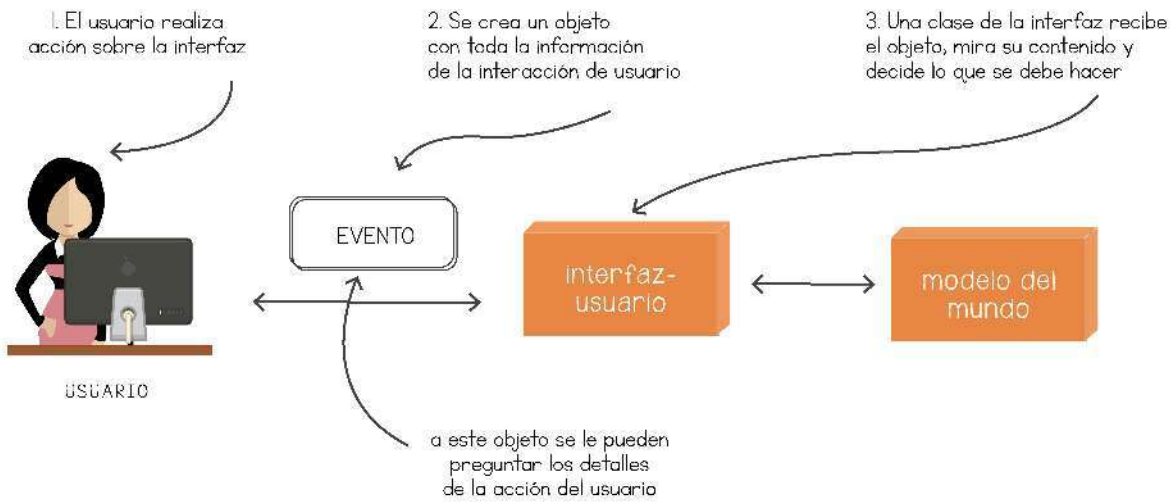
- Crear la ventana de la interfaz, con sus paneles y sus componentes gráficos.
- Obtener de los componentes gráficos la información suministrada por el usuario.
- Asociar un nombre con el evento que genera cada botón y escribir el método que lo atiende.
- Convertir la información que teclea el usuario a otros tipos de datos.
- Presentar al usuario mensajes con información simple.
- Escribir el método que inicia la ejecución del programa.
- Crear el modelo del mundo en el constructor de la ventana y guardar una asociación hacia él.
- Escribir en los paneles los métodos de servicio (refresco y acceso a la información).

Lo único que nos falta en este momento es definir la manera de utilizar todo lo anterior para implementar los requerimientos funcionales. Para esto, debemos definir las responsabilidades y compromisos de cada uno de los participantes, de manera que siempre sepamos quién debe hacer qué, y en qué orden. A esto lo denominaremos el **protocolo** de la arquitectura. Sobre este punto debemos decir que hay muchas soluciones posibles y que la arquitectura que utilizamos a lo largo de este libro es sólo una manera de estructurar y

repartir las responsabilidades. Tiene la ventaja de facilitar la localización de cada uno de los componentes del programa, aumentando su claridad y simplificando su mantenimiento, dos puntos fundamentales a la hora de escribir un programa de computador.

La arquitectura que usamos se basa en la idea de que los requerimientos funcionales se implementan en la ventana principal (un método por requerimiento) y que es allí donde se coordinan todas las acciones, tanto de los elementos que se encuentran en los paneles como de los elementos del modelo del mundo. En la figura 5.11 aparece el protocolo con los seis pasos básicos para reaccionar a un evento generado por el usuario.

Fig. 5.11 Los seis pasos del protocolo de la arquitectura de un programa



Veamos ahora paso por paso el protocolo, para explicar la figura anterior. Los números asociados con las flechas indican el orden en el que las acciones se llevan a cabo.

Paso 1: el usuario genera un evento oprimiendo un botón en uno de los paneles de la interfaz. Dicho evento se convierte en un objeto que lleva toda la información relacionada con la acción del usuario.

- Debe reaccionar el panel que contiene el botón.

Paso 2: el panel reacciona al evento con su método actionPerformed, el cual debe solicitar a la ventana principal que ejecute el requerimiento funcional pedido por el usuario.

- El panel debe pasarle toda la información que tiene en su interior y que se necesita como entrada del requerimiento funcional.
- Si hay necesidad de convertir la información ingresada por el usuario a un tipo específico de datos, es responsabilidad del panel hacerlo.
- Un requerimiento funcional se implementa como un método en la ventana.

Paso 3: la ventana principal completa la información necesaria para poder cumplir con el requerimiento funcional, pidiéndola a los demás paneles.

- Puesto que el método que implementa el requerimiento funcional es responsable de que se cumplan las precondiciones de los métodos del modelo del mundo, en este punto debe hacer todas las verificaciones necesarias y, en caso de que surja un problema, puede cancelar la reacción y notificar al usuario de lo sucedido.
- Para realizar este paso, desde el método que implementa el requerimiento funcional se invocan los métodos de acceso a la información de los demás paneles.

Paso 4: se pide al modelo del mundo que haga una modificación (basada en los valores ingresados por el usuario) o que calcule algún valor.

- Se utiliza en este paso la asociación (o las asociaciones) que tiene la interfaz hacia el modelo del mundo, para invocar el o los métodos que van a ayudar a implementar el requerimiento funcional. Cualquier excepción lanzada por los métodos del modelo del mundo debería ser atrapada en este punto.
- Si sólo se está pidiendo al modelo del mundo que calcule un valor (por ejemplo, calcular el avalúo del vehículo), al final de este paso ya se tiene toda la información necesaria para iniciar el proceso de refresco.
- Si se pidió una modificación del modelo del mundo, se debe ejecutar el paso 5.

Paso 5: si en el paso anterior se pidió una modificación al modelo del mundo, se llaman aquí los métodos que retornan los nuevos valores que se deben presentar.

Para saber qué métodos invocar, se debe establecer qué partes de la información mostrada al usuario deben ser recalculadas.

Paso 6: se pide a todos los paneles que tienen información que pudo haber cambiado que actualicen sus valores.

Para eso se utilizan los métodos de refresco implementados por los paneles.
Hay muchos modelos distintos para mantener la información de la interfaz sincronizada con el estado del modelo del mundo. El que se plantea aquí puede ser muy ineficiente en problemas grandes, por lo que insistimos en que esta arquitectura sólo debe ser utilizada en problemas pequeños.

8.5. ¿Cómo Hacer que los Paneles Conozcan la Ventana?

De acuerdo con el protocolo antes mencionado, todos los paneles que tengan botones (llamados paneles activos) deben tener una asociación hacia la ventana principal, de manera que sea posible ejecutar los métodos que implementan los requerimientos funcionales. Esto hace que los constructores de los paneles que tienen botones deban modificar un poco su estructura, tal como se muestra en el ejemplo 12.

```
public class PanelBusquedas extends JPanel implements ActionListener
{
    private InterfazImpuestosCarro principal;

    public PanelBusquedas( InterfazImpuestosCarro pPrincipal )
    {
        principal = pPrincipal;
        ...
    }
}
```

- Modificación de la clase que implementa el panel de búsquedas, para incluir una asociación a la ventana principal.
- En el atributo llamado "principal" almacenamos la referencia a la ventana principal, recibida como parámetro.

```
public class PanelOpciones extends JPanel implements ActionListener
{
    private InterfazImpuestosCarro principal;

    public PanelOpciones ( InterfazImpuestosCarro pPrincipal )
    {
        principal = pPrincipal;
        ...
    }
}
```

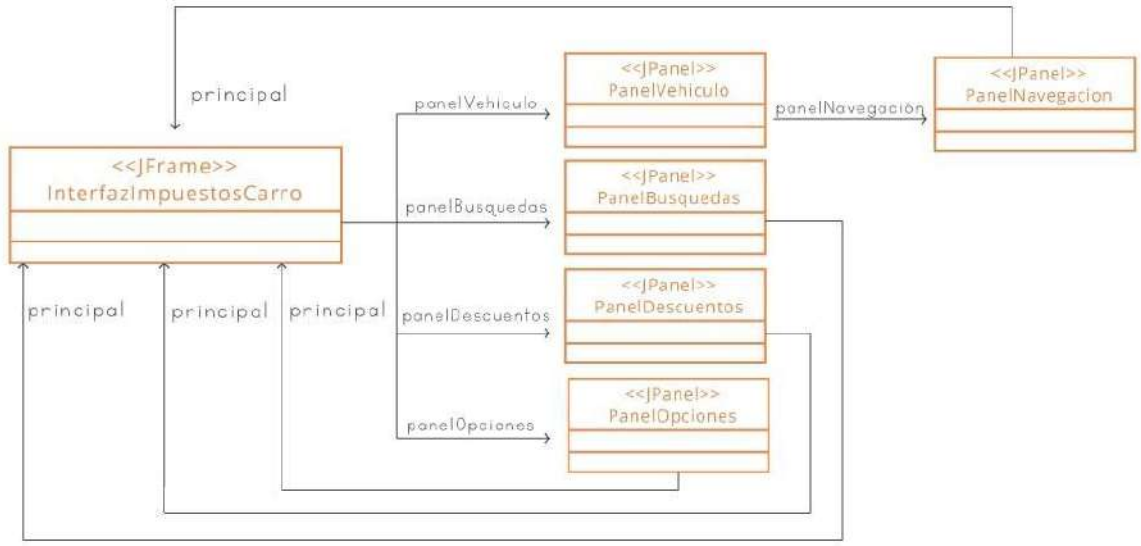
- Modificación de la clase que implementa el panel de búsquedas para incluir una asociación a la ventana principal.
- En el constructor de la ventana, cuando se cree este panel, se debe pasar como parámetro una referencia a la ventana de la interfaz.

```
public class PanelNavegacion extends JPanel implements ActionListener
{
    private InterfazImpuestosCarro principal;

    public PanelNavegacion ( InterfazImpuestosCarro pPrincipal )
    {
        principal = pPrincipal;
        ...
    }
}
```

- Modificación de la clase que implementa el panel de navegación para incluir una asociación a la ventana principal.
- En el constructor de PanelVehiculo, cuando se cree este panel, se debe pasar como parámetro una referencia a la ventana de la interfaz.

```
public class PanelVehiculo extends JPanel implements  ActionListener
{
    public PanelVehiculo( InterfazImpuestosCarro pPrincipal )
    {
        add( new PanelNavegacion( pPrincipal ), BorderLayout.SOUTH );
        ...
    }
}
```

- Modificación de la clase que implementa panel con la información del vehículo, para incluir una asociación a la ventana principal.
- En este caso no se crea el atributo llamado "principal" porque la referencia a la ventana principal sólo se utiliza para crear el panel de navegación.



8.6. ¿Cómo Implementar los Requerimientos Funcionales?

Lo único que nos hace falta ahora es implementar los métodos de los requerimientos funcionales. Estos métodos deben formar parte de la clase de la ventana principal de la interfaz, y tienen como objetivo coordinar los paneles y el modelo del mundo para lograr lo pedido por el cliente. En el ejemplo 13 se muestra la estructura de dichos métodos.

Ejemplo 13

Objetivo: Ilustrar la construcción de los métodos que implementan los requerimientos funcionales.

En este ejemplo se muestran los dos métodos de la clase InterfazImpuestosCarro que implementan los requerimientos funcionales del caso de estudio.

```
 public void buscarPorLinea( String pLinea )
 {
     // 1
     Vehiculo respuesta = calculador.buscarVehiculoPorLinea( pLinea );
     if( respuesta == null )
     {
         // 2
         JOptionPane.showMessageDialog( this, "No se encontró ningún vehículo de esta
línea", "Buscar por línea", JOptionPane.ERROR_MESSAGE );
     }
     else
     {
         // 3
         panelVehiculo.actualizar( respuesta.darMarca( ), respuesta.darLinea( ), respu
esta.darAnio( ), respuesta.darPrecio( ), respuesta.darRutaImagen( ) );
     }
}
```

- Método de la ventana principal que atiende el requerimiento funcional de mostrar el vehículo con la línea dada.
- En el paso 1 se le pide al modelo del mundo que busque el vehículo con la línea dada.
- Si no se encontró un vehículo de la línea dada (respuesta == null), se muestra un mensaje al usuario indicándolo.
- En caso contrario, se actualiza la información del PanelVehiculo con la del vehículo encontrado, usando el método actualizar de este panel.

```
  public void calcularImpuestos( )
  {
      // 1
      boolean descProntoPago = panelDescuentos.hayDescuentoProntoPago( );
      boolean descServicioPublico = panelDescuentos.hayDescuentoServicioPublico( );
      boolean descTrasladoCuenta = panelDescuentos.hayDescuentoTrasladoCuenta( );

      // 2
      double pago = calculador.calcularPago( descProntoPago, descServicioPublico, desc
TrasladoCuenta );

      // 3
      DecimalFormat df = ( DecimalFormat )NumberFormat.getInstance( );
      df.applyPattern( "$ ###,###.##" );

      // 4
      JOptionPane.showMessageDialog( this, "El valor a pagar es: " + df.format( pago )
, "Cálcular impuestos", JOptionPane.INFORMATION_MESSAGE );
  }
```

- Método de la ventana principal que atiende el requerimiento funcional de calcular el valor que se debe pagar de impuestos.

- En el paso 1 se pide toda la información de los descuentos que se requiere para calcular el pago.
- En el paso 2 se pide al modelo del mundo que calcule el valor que se debe pagar de impuestos.
- En el paso 3 se crea el formato en el cual se desea visualizar la información.
- En al paso 4 se muestra un mensaje al usuario con la el valor que se debe pagar por los impuestos del vehículo actual.

9. Ejecución de un Programa en Java

Para ejecutar un programa en Java es necesario especificar desde la ventana de comandos del sistema operativo el nombre del archivo jar que contiene el código compilado del programa y el nombre completo de la clase principal por la cual debe comenzar la ejecución (la clase que tiene el método main). (Si el programa no está empaquetado en un archivo jar, hay que dar solamente el nombre de la clase principal.) Para el caso de estudio, el comando de ejecución es el siguiente (en una sola línea):

```
java  -classpath  ./lib/impuestosCarro.jar uniandes.cupi2.impuestosCarro.interfaz.Inte
rfazImpuestosCarro
```

> Si el computador no encuentra el archivo jar, o si dentro de éste no encuentra la clase que se le especificó en el comando de ejecución, aparece en la ventana de comandos del sistema operativo el error: *java.lang.NoClassDefFoundError*.

10. Hojas de Trabajo

10.1 Hoja de Trabajo Nº 1: Granja de traducciones

Descargue esta hoja de trabajo a través de los siguientes enlaces: Descargar PDF | Descargar Word.

Enunciado. Lea detenidamente el siguiente enunciado sobre el cual se desarrollará la presente hoja de trabajo.

Se quiere crear una aplicación que ayude a aprender los nombres de los animales de la granja en inglés. Cada vez que se inicia una nueva jugada, aparece el nombre de un animal en inglés, y el usuario debe seleccionar la imagen del animal. Posteriormente, la aplicación muestra cuál era la respuesta correcta. Por cada respuesta correcta, el usuario obtiene 20 puntos.

Se espera que la aplicación permita: (1) iniciar una jugada, (2) seleccionar un animal, (3) visualizar la traducción correcta, (4) visualizar el puntaje del jugador.

La siguiente es la interfaz de usuario que se quiere construir, en la cual se identifican tres zonas:

Requerimientos funcionales. Identifique y especifique los cuatro requerimientos funcionales de la aplicación.

Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

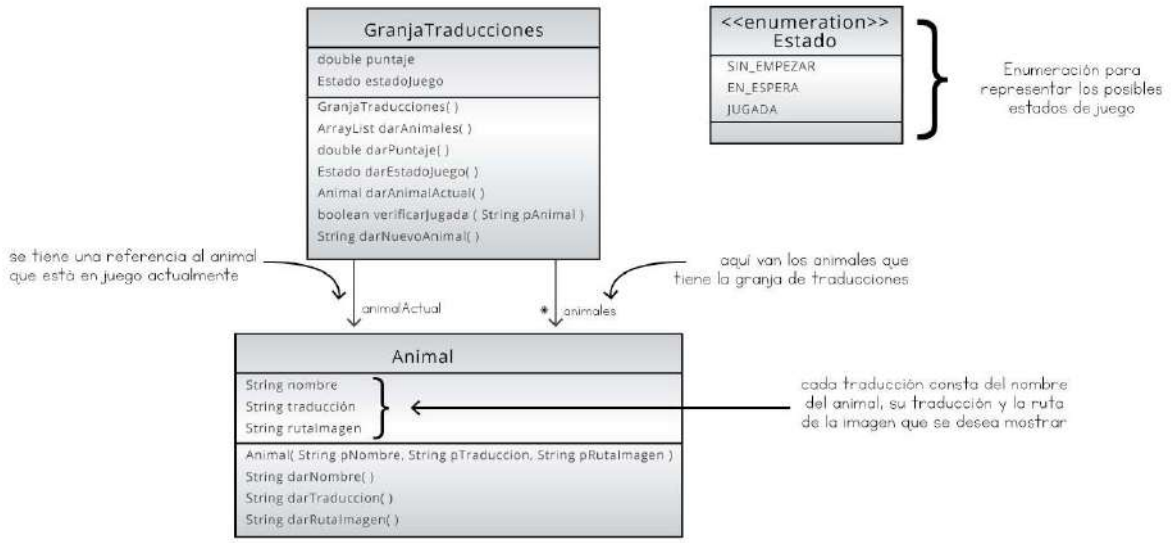
Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 4

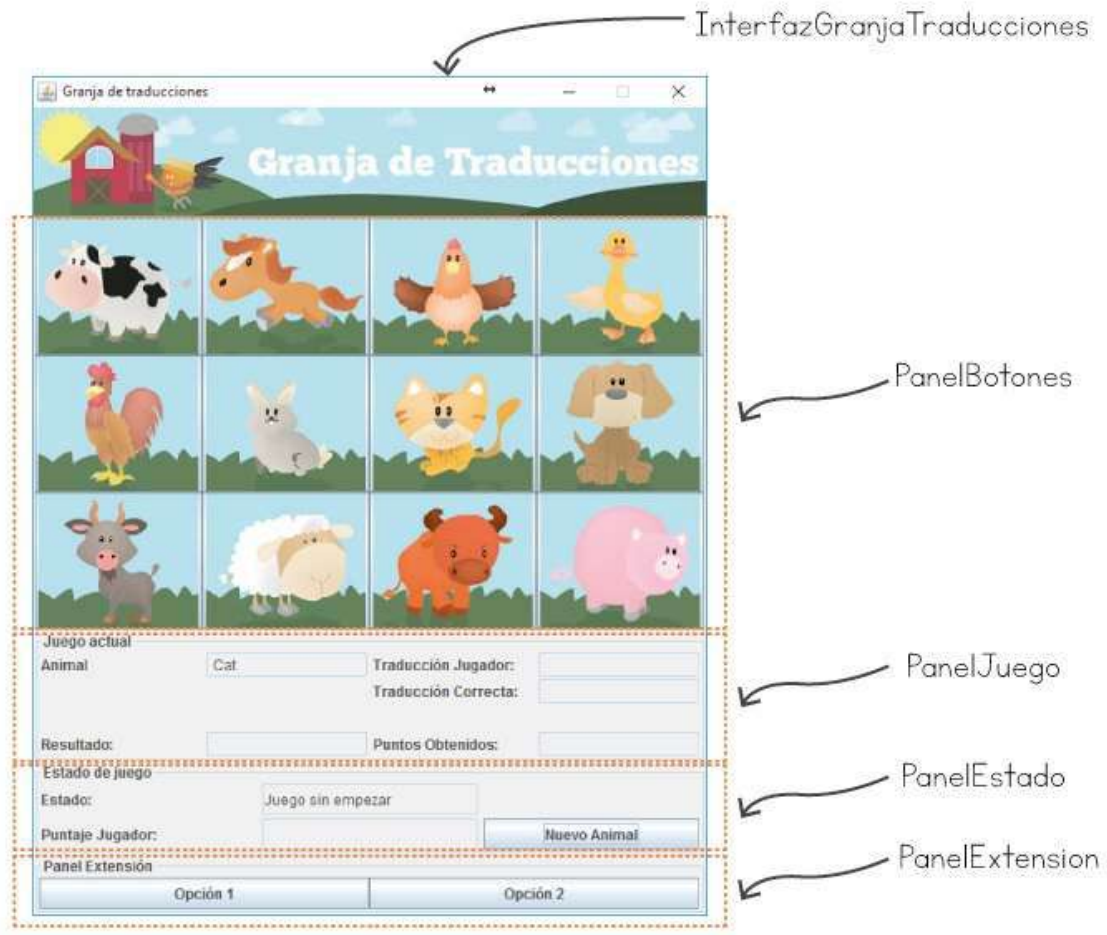
Nombre	
Resumen	
Entradas	
Resultado	

Modelo del mundo. Estudie el diagrama de clases que implementa el modelo del mundo y los métodos de cada una de las clases.



Nombre método	**Descripción**
GranjaTraducciones()	Crea una granja con sus traducciones.
Animal darAnimales( )	Retorna la lista de animales de la granja.
double darPuntaje( )	Retorna el puntaje del animal actual.
Estado darEstadoJuego( )	Retorna el estado actual del juego.
Animal darAnimalActual( )	Retorna el animal actual.
boolean verificarJugada(String pAnimal)	Verifica si la traducción del animal ingresado corresponde con la traducción del animal actual.
String darNuevoAnimal()	Retorna el nombre del nuevo animal seleccionado aleatoriamente.

Interfaz por construir. Observe la estructura de la interfaz que se desea construir y los nombres de las clases que se deben asociar con sus partes.



Arquitectura de la interfaz. Dibuje en UML el diagrama de las clases (sin atributos ni métodos) que conformarán la interfaz. Utilice los estereotipos para indicar sí es un JFrame o JPanel. Dibuje también las clases del mundo con las que se relacionan.

Diagrama UML


Construcción de la interfaz. Siga los siguientes pasos para construir la interfaz dada.

1	Cree el paquete para las clases de la interfaz (**uniandes.cupi2.granjaTraducciones.interfaz**).
2	Cree la clase **InterfazGranjaTraducciones** como extensión de **JFrame**. Escriba el método **main()**, encargado de iniciar la ejecución del programa. Incluya los atributos para representar el modelo del mundo, la imagen del título y los paneles que lo conforman. Defina el tamaño de la ventana como 565x 700. Asocie con la ventana un distribuidor en los bordes. Cree cada uno de los paneles y añádalos adecuadamente a la ventana.
3	Cree la clase **PanelBotones** como una extensión de la clase **JPanel** que implementa **ActionListener**. Declare como atributo una contenedora de botones. Implemente un constructor que reciba como parámetro una referencia a la ventana del programa y la lista de animales. Asocie con el panel un distribuidor en malla de 3 x 4. Cree todos los botones, asociando como comando el nombre del animal, y asignado la imagen asociada al animal. Escriba el esqueleto del método **actionPerformed()**.
4	Cree la clase **PanelJuego** como extensión de **JPanel**. Declare los atributos para manejar los componentes gráficos que se encuentran en su interior. Deshabilite la posibilidad de escribir en las zonas de texto. Asocie con el panel un distribuidor en malla de 4 x 4.

5	Cree la clase **PanelEstado** como una extensión de la clase **JPanel** que implementa **ActionListener**. Declare una constante para identificar el evento que va a generar el botón del panel. Declare los atributos para manejar los componentes graficos que se encuentran en su interior. Implemete un constructor que reciba como parámetro una referencia a la venatana del programa. Asocie con el panel un distribuidor en malla de 3 x 3. Escriba el esqueleto del método **actionPerformed()**.
6	Cree la clase **PanelExtension** como una extensión de la clase **JPanel** que implementa **ActionListener**. Declare una constante para identificar los eventos que van a generar los botones del panel. Declare los atributos para manejar los componentes graficos que se encuentran en su interior. Implemete un constructor que reciba como parámetro una referencia a la venatana del programa. Asocie con el panel un distribuidor en malla de 1 x 2. Escriba el esqueleto del método **actionPerformed()**.
7	En las clases de los paneles, escriba los métodos de refresco de la información. Incluya en los métodos de refresco el servicio de “borrar” el contenido de los campos una vez que se haya ejecutado una operación
8	En la clase **InterfazGranjaTraducciones**, escriba un método por cada uno de los requerimientos funcionales. Defina la signatura de manera que reciba como parámetro toda la información de la que dispone el panel que va a hacer la invocación.
9	Complete el método **actionPerformed()** en las clases **PanelBotones**, **PanelEstado** y **PanelExtension**, haciendo las llamadas respectivas a los métodos de la ventana principal que implementan los requerimientos funcionales.
10	Complete todos los detalles que falten en la interfaz, para obtener la visualización y el funcionamiento descritos en el enunciado. Pruebe cada una de las opciones del programa.

10.2 Hoja de Trabajo Nº 2: Examen

Descargue esta hoja de trabajo a través de los siguientes enlaces: Descargar PDF | Descargar Word.

Enunciado. Lea detenidamente el siguiente enunciado sobre el cual se desarrollará la presente hoja de trabajo.

Se quiere construir una aplicación que permita simular un examen de geografía, donde se preguntan las capitales de diferentes países. El examen tiene 8 preguntas. De cada pregunta se muestra el número de la pregunta, el enunciado, la bandera del país cuya capital se está preguntando y las 4 posibles respuestas. Cuando se elige una respuesta, se muestra la respuesta seleccionada por el usuario, la respuesta correcta y los puntos obtenidos.

La aplicación carga la información de los países desde un archivo, y selecciona aleatoriamente las preguntas del examen y sus posibles respuestas (el programa no implementa la forma de modificarlas). Esto quiere decir que cada vez que se inicia un nuevo examen, las preguntas serán diferentes.

La aplicación debe permitir: navegar entre las preguntas del examen, visualizar la información de una pregunta, responder una pregunta, terminar un examen, iniciar un nuevo examen y visualizar el progreso de las preguntas.

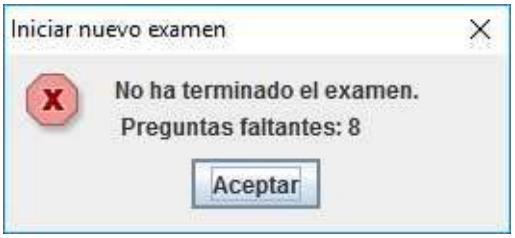
La siguiente es la interfaz de usuario que se quiere construir, en la cual se identifican cuatro zonas:

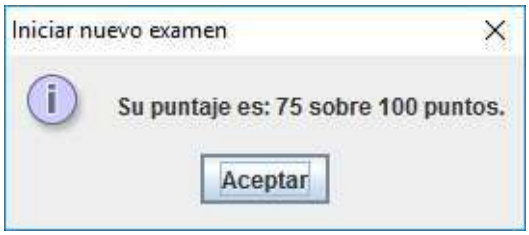
Los siguientes son los mensajes que hay que presentar al usuario, como resultado de su interacción con el programa:

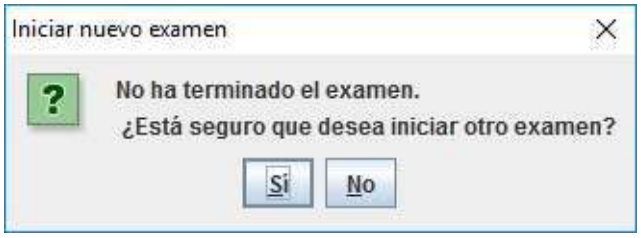
Este mensaje aparece cuando el usuario oprime el botón para retroceder en la lista de preguntas y está situado en el primero:

Este mensaje aparece cuando el usuario oprime el botón para avanzar en la lista de preguntas y está situado en el último:

Este mensaje aparece cuando el usuario intenta terminar el examen, pero no ha respondido todas las preguntas.



Este mensaje aparece cuando el usuario termina un examen, cuyas respuestas fueron respondidas en su totalidad. 

Este mensaje de confirmación aparece cuando intenta iniciar un nuevo examen, pero no ha respondido todas las preguntas. 

Requerimientos funcionales. Identifique y especifique los requerimientos funcionales de la aplicación.

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	

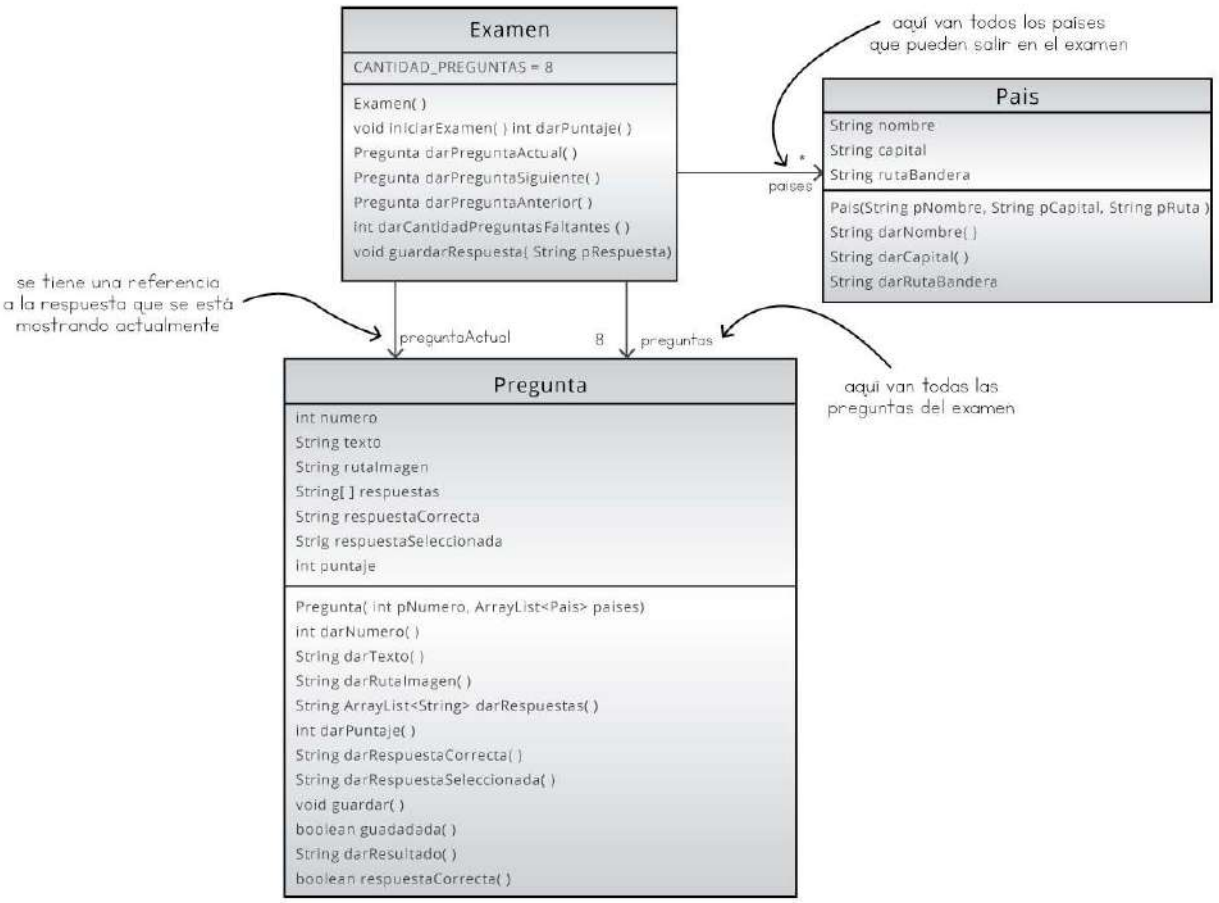
Requerimiento Funcional 5

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 6

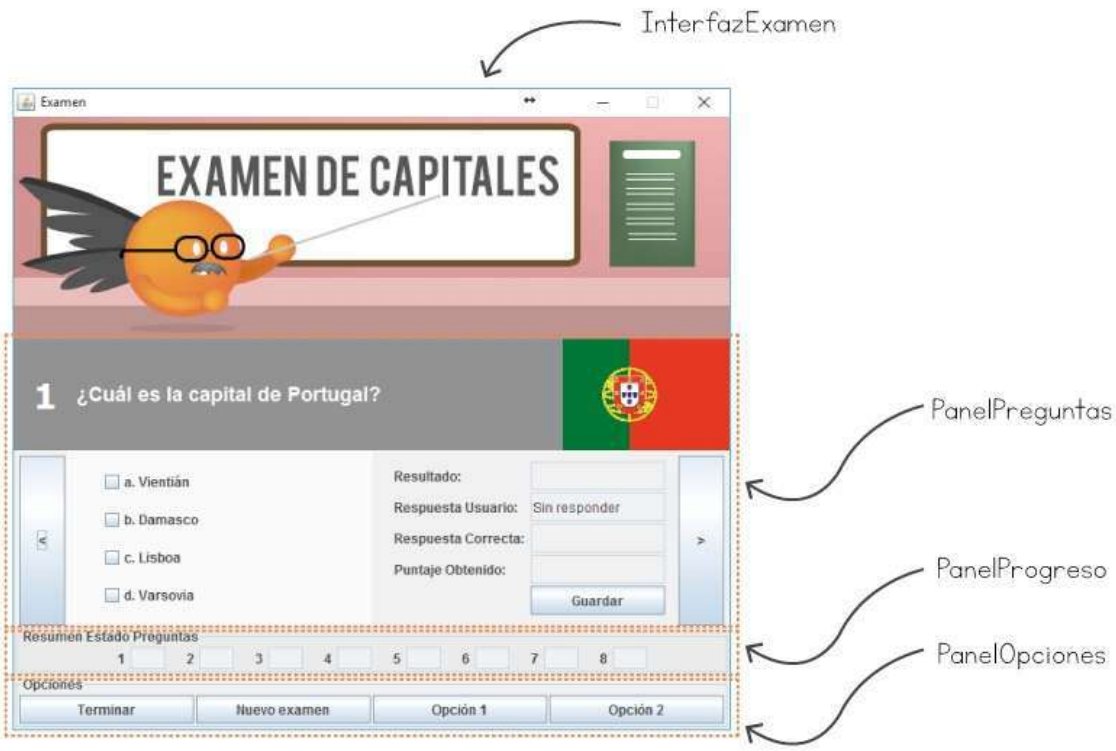
Nombre	
Resumen	
Entradas	
Resultado	

Modelo del mundo. Estudie el diagrama de clases que implementa el modelo del mundo y los métodos de cada una de las clases.



Nombre método	**Descripción**
Examen()	Crea el examen, cargando la información de los países de un archivo. Si hay algún problema en el momento de leer el archivo, lanza una excepción.
void iniciarExamen( )	Genera preguntas con sus respuestas seleccionando países de la lista aleatoriamente.
int darPuntaje( )	Retorna el puntaje de la pregunta actual.
Pregunta darPreguntaActual( )	Retorna la pregunta actual.
Pregunta darPreguntaAnterior( )	Retorna la pregunta anterior y actualiza la nueva pregunta actual. Si ya se encuentra en la primera pregunta, lanza una excepción.
Pregunta darPreguntaSiguiente( )	Retorna la pregunta siguiente y actualiza la nueva pregunta actual. Si ya se encuentra en la última pregunta, lanza una excepción.
int darCantidadPreguntasFaltantes( )	Retorna la cantidad de preguntas sin responder.
void guardarRespuesta(String pRespuesta)	Guarda la respuesta seleccionada por el usuario en la pregunta actual.

Interfaz por construir. Observe la estructura de la interfaz que se desea construir y los nombres de las clases que se deben asociar con sus partes.



Arquitectura de la interfaz. Dibuje en UML el diagrama de clases (sin atributos ni métodos) que conformarán la interfaz. Utilice los estereotipos para indicar sí es un JFrame o JPanel. Dibuje también las clases del mundo con las que se relacionan.

Diagrama UML


Construcción de la interfaz. Siga los siguientes pasos para construir la interfaz dada.

1	Cree el paquete para las clases de la interfaz (**uniandes.cupi2.sinonimos.interfaz**).
2	Cree la clase **InterfazExamen** como extensión de **JFrame**. Escriba el método **main()**, encargado de iniciar la ejecución del programa. Incluya los atributos para representar el modelo del mundo, así como los elementos y los paneles que lo conforman. Defina el tamaño de la ventana como 400 x 180. Asocie con la ventana un distribuidor en los bordes. Cree cada uno de los paneles y añádalos adecuadamente a la ventana.
3	Cree la clase **PanelPregunta** como una extensión de la clase **JPanel** que implementa **ActionListener**. Declare las constantes para identificar los eventos que van a generar los botones del panel. Declare los atributos para manejar los componentes gráficos que se encuentran en su interior. Implemente un constructor que reciba como parámetro una referencia a la ventana del programa. Asocie con el panel un distribuidor en los bordes. Cree los paneles auxiliares necesarios para poder distribuir los elementos de la forma esperada. Tenga en cuenta que los elementos de tipo JCheckBox también deben llamar al método **actionPerformed( )**, y por ende también deben tener un comando asociado. Escriba el esqueleto del método **actionPerformed()**.
4	Cree la clase **PanelProgreso** como una extensión de **JPanel** que implementa **ActionListener**. Declare los atributos para manejar los componentes gráficos que se encuentran en su interior. Deshabilite la posibilidad de escribir en las zonas de texto. Asocie con el panel un distribuidor en grilla. Escriba el esqueleto del método **actionPerformed()**. En dicho método no vamos a llamar ningún método de la ventana.
5	Cree la clase **PanelOpciones** como una extensión de la clase **JPanel** que implementa **ActionListener**. Declare las constantes para identificar los eventos de los botones. Declare los atributos para manejar los componentes gráficos que se encuentran en su interior. Implemente un constructor que reciba como parámetro una referencia a la ventana del programa. Asocie con el panel un distribuidor en malla. Escriba el esqueleto del método **actionPerformed()**.
6	En las clases de los tres paneles, escriba los métodos de refresco de la información y los métodos de acceso a la información.
7	En la clase **InterfazExamen**, escriba un método para implementar cada requerimiento funcional. Asegúrese de validar los datos y manejar las excepciones de manera que presente los mensajes descritos en el enunciado.
8	Complete el método **actionPerformed()** en las clases **PanelPregunta** y **PanelOpciones**, de modo que haga la llamada a los método de la ventana principal correspondientes. Recuerde que en este método también se deben ejecutar las acciones necesarias para que cuando un usuario seleccione un CheckBox, no haya ninguna otra casilla seleccionada.
9	Complete todos los detalles que falten en la interfaz, para obtener la visualización y el funcionamiento descritos en el enunciado. Pruebe cada una de las opciones del programa.


06
MANEJO DE ESTRUCTURAS DE DOS DIMENSIONES Y PERSISTENCIA

1. Objetivos Pedagógicos

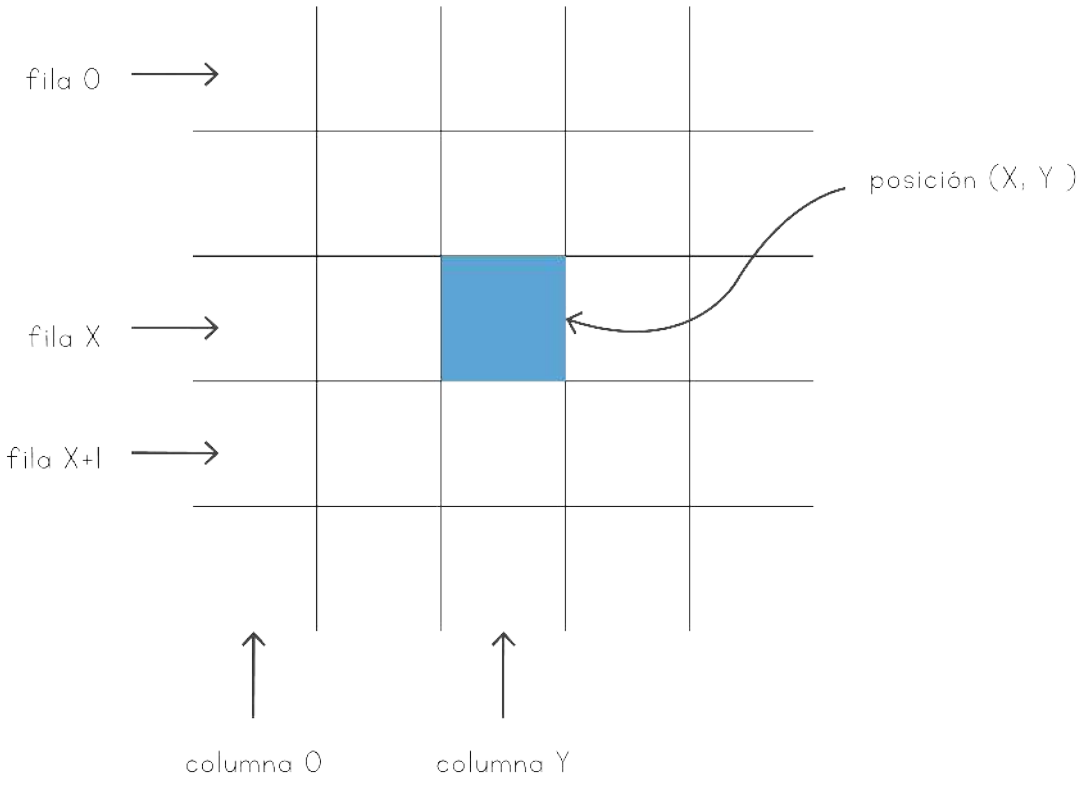
Al final de este nivel el lector será capaz de:

- Utilizar el concepto de matriz como elemento de modelado para agrupar los elementos del mundo en una estructura contenedora de dos dimensiones de tamaño fijo.
- Identificar los patrones de algoritmo para manejo de matrices, dada la especificación de un método.
- Utilizar el esqueleto del patrón de algoritmo y los pasos asociados como medio para escribir un algoritmo para manipular una matriz.
- Utilizar un esquema simple de persistencia para manejar el estado inicial de un problema.
- Desarrollar un programa completo, teniendo una visión global de las etapas del proceso que se debe seguir para resolver un problema usando un computador.

2. Motivación

Si fuésemos a diseñar un programa para simular un juego de ajedrez, podemos imaginar el tablero de juego en la forma de una cuadrícula compuesta por 8 filas y 8 columnas. En ese escenario, quisiéramos tener una estructura que nos permitiera hacer la manipulación de las diferentes fichas del tablero de juego, utilizando la posición de la fila y la posición de la columna en el que está ubicada cada ficha como en el plano cartesiano que se muestra en la figura. 6.1. Una estructura que nos permitiera referirnos directamente a una ficha por sus coordenadas: la ficha que se encuentra en la posición (fila, columna).

Fig. 6.1 Plano cartesiano y una estructura matricial



Hay muchos otros casos en donde esta idea de tener una estructura contenedora de dos dimensiones es muy útil y representa, de manera natural, un grupo de elementos del mundo del problema. Supongamos, por ejemplo, que queremos manipular imágenes fotográficas. Una imagen fotográfica puede entenderse como una colección de puntos en un plano cartesiano. Cada punto representa un píxel de la imagen. Si necesitamos construir un programa para manipular imágenes fotográficas, que sea capaz de cambiar los colores,

aplicar un filtro, etc. sería muy conveniente poder contar con una estructura de modelado que nos permitiera manipular los puntos de la imagen como en el plano cartesiano: el píxel que está en las coordenadas (x, y).

En este nivel vamos a estudiar la manera de definir, crear y manipular estructuras contenedoras de dos dimensiones. Estas estructuras se llaman **matrices**. Utilizaremos inicialmente un caso de estudio que corresponde a la construcción de un programa que permite hacer manipulaciones simples sobre imágenes fotográficas. Veremos también la forma de adaptar los patrones de algoritmo para el caso de las matrices, de tal manera que podamos guiarnos para su construcción por las ideas presentadas en el nivel 3.

Después estudiaremos y plantearemos una solución al problema de cómo predefinir el estado inicial de un programa. En muchos de nuestros programas, quisiéramos que la información que define el estado inicial pudiera ser leída desde un archivo, creado con herramientas externas a nuestro programa (como un editor de texto). Por ejemplo, en el caso de la tienda de libros que presentamos en el nivel 2, la configuración inicial del catálogo de libros se podría leer desde un archivo. Esto facilitaría adaptar el programa a distintos contextos de la tienda sin necesidad de cambiar el funcionamiento del mismo.

Finalizaremos este nivel dando una visión global del proceso que se debe seguir para resolver un problema usando un computador. Allí veremos de manera esquemática las etapas que se deben seguir y los puntos más importantes que se deben tener en cuenta en cada una de ellas.

3. Caso de Estudio Nº 1: Un Visor de Imágenes

Se quiere construir una aplicación que permita la visualización de imágenes en formato BMP (BitMaP) de diferentes dimensiones. El formato BMP es probablemente el formato de imágenes más simple que existe y consiste en guardar la información del color de cada píxel o punto que conforma la imagen. El color de un píxel se expresa en el sistema RGB (Red-Green-Blue), donde el color se forma mediante la combinación de tres componentes (rojo, verde y azul) cada uno de los cuales es representado por un número que indica la proporción del color del componente en el color resultante.

Además de mostrar la imagen, el programa debe ofrecer servicios de transformación de la imagen. Por ejemplo, debe poder transformar la imagen en su negativa, polarizar o aplicar un filtro sobre la imagen, invertir la imagen, rotarla, etc. La interfaz de usuario que utilizaremos para este problema se muestra en la figura 6.2.

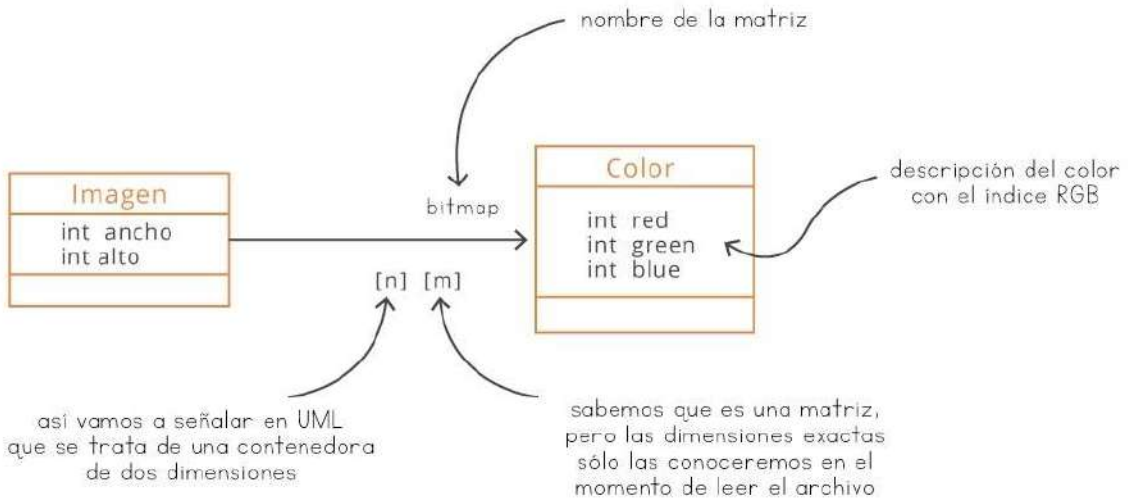
Fig. 6.2 Interfaz de usuario para el visor de imágenes

3.1. Comprensión del Mundo del Problema

Si estudiamos el mundo del problema, el único elemento que encontramos es la imagen. Una imagen contiene una colección de píxeles. Esta colección está organizada en forma de una matriz de dos dimensiones donde cada posición contiene la información sobre el color del píxel. El tamaño de la imagen está limitado a un número de alto x ancho pixeles.

La figura 6.3 muestra el modelo conceptual del problema. Nótese que estamos modelando una asociación llamada bitmap que representa una estructura única que nos permite modelar la matriz de colores.

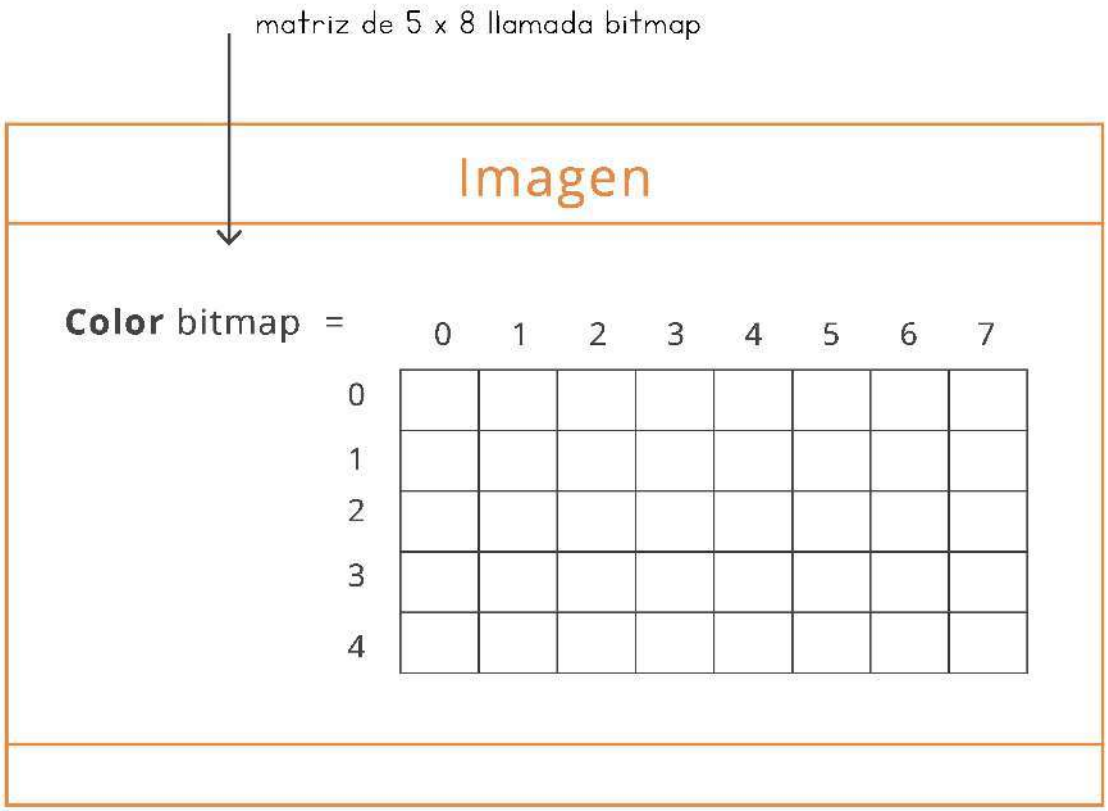
Fig. 6.3 Modelo conceptual del caso de visor de imágenes



4. Contenedoras de dos Dimensiones: Matrices

Una matriz es una estructura contenedora de dos dimensiones, de tamaño fijo, cuyos elementos son referenciados utilizando dos índices: el índice de la fila y el índice de la columna. Este tipo de estructuras se utiliza cuando en el mundo del problema hay características que se adaptan a esta representación bidimensional. Para hacer el paralelo con la visualización que usamos en el nivel 3 para mostrar la idea de un arreglo, en la figura 6.4 presentamos una manera de imaginar una clase que tiene un atributo que corresponde a una matriz.

Fig. 6.4 Visualización de una matriz como un atributo de una clase



En las secciones que siguen, veremos la manera de declarar, crear y manipular contenedoras de dos dimensiones de tamaño fijo en el lenguaje de programación Java.

4.1. Declaración de una Matriz

En Java, las estructuras contenedoras de dos dimensiones de tamaño fijo se denominan **matrices** y se declaran como se muestra en el ejemplo 1.

Ejemplo 1

Objetivo: Mostrar la manera de declarar una matriz en Java.

En este ejemplo se presenta la declaración en Java de la matriz que representa la imagen en el caso de estudio.

```
public class Imagen
{
    //--------------------------------------
    // Atributos
    //--------------------------------------

    private int ancho;

    private int alto;

    private Color[][] bitmap;
}
```

Es conveniente declarar el número de columnas (ancho) y el número de filas (ancho) como atributos. Esto va a facilitar realizar posteriores modificaciones al programa.

La declaración del atributo `bitmap` indica que es una matriz de dos dimensiones de tamaño fijo (el valor exacto del tamaño será determinado en el momento de la inicialización de la matriz) y cuyos elementos son todos de tipo Color.

La clase Color es una clase predefinida de Java que permite manejar colores en formato RGB. Esta clase de encuentra en el paquete `java.awt`. En nuestros ejemplos utilizamos algunos de los servicios que ofrece esa clase.

4.2. Inicialización de una Matriz

Al igual que con cualquier otro atributo de una clase, es necesario inicializar la matriz antes de poderla utilizar. Para hacerlo, se deben definir las dimensiones de la matriz. Esta inicialización es obligatoria, puesto que es entonces cuando le decimos al computador cuántos valores se van a manejar en la matriz, lo que corresponde al espacio en memoria que debe reservar. Veamos en el ejemplo 2 cómo se hace esto para el caso de estudio.

Ejemplo 2

Objetivo: Mostrar la manera de crear una matriz en Java.
En este ejemplo se presenta el constructor de la clase Imagen, que tiene la responsabilidad de crear la matriz que va a contener los píxeles.

```
// Constructor
//-------------------------------------
public Imagen( )
{
    ancho = 400;
    alto = 300;
    bitmap = new Color[ alto ][ ancho ];
}
```

- Se utiliza la instrucción `new`, pero en este caso se indican dos dimensiones de la matriz, en nuestro caso de ejemplo 300 filas (alto) cada una con 400 columnas (ancho).
- Aunque el espacio ya queda reservado con la instrucción `new`, el valor de cada uno de los elementos del arreglo sigue siendo indefinido. Esto lo arreglaremos más adelante. Recuerde que siempre van primero las filas y luego las columnas.

El lenguaje Java provee un operador especial (`length`), que permite consultar el número de filas que tiene una matriz. En el caso de ejemplo, la expresión `bitmap.length` debe dar el valor 300 que corresponde al número de filas, independientemente de si las casillas individuales ya han sido o no inicializadas. De la misma manera el operador `length` nos permite preguntar el número de columnas de la matriz. La expresión `bitmap[0].length` debe dar el valor 400, que corresponde al número de columnas en la fila 0. Como en nuestro caso todas las filas tienen el mismo número de columnas, esa expresión nos puede servir para establecer la segunda dimensión de la matriz.

4.3. Acceso a los Elementos de una Matriz

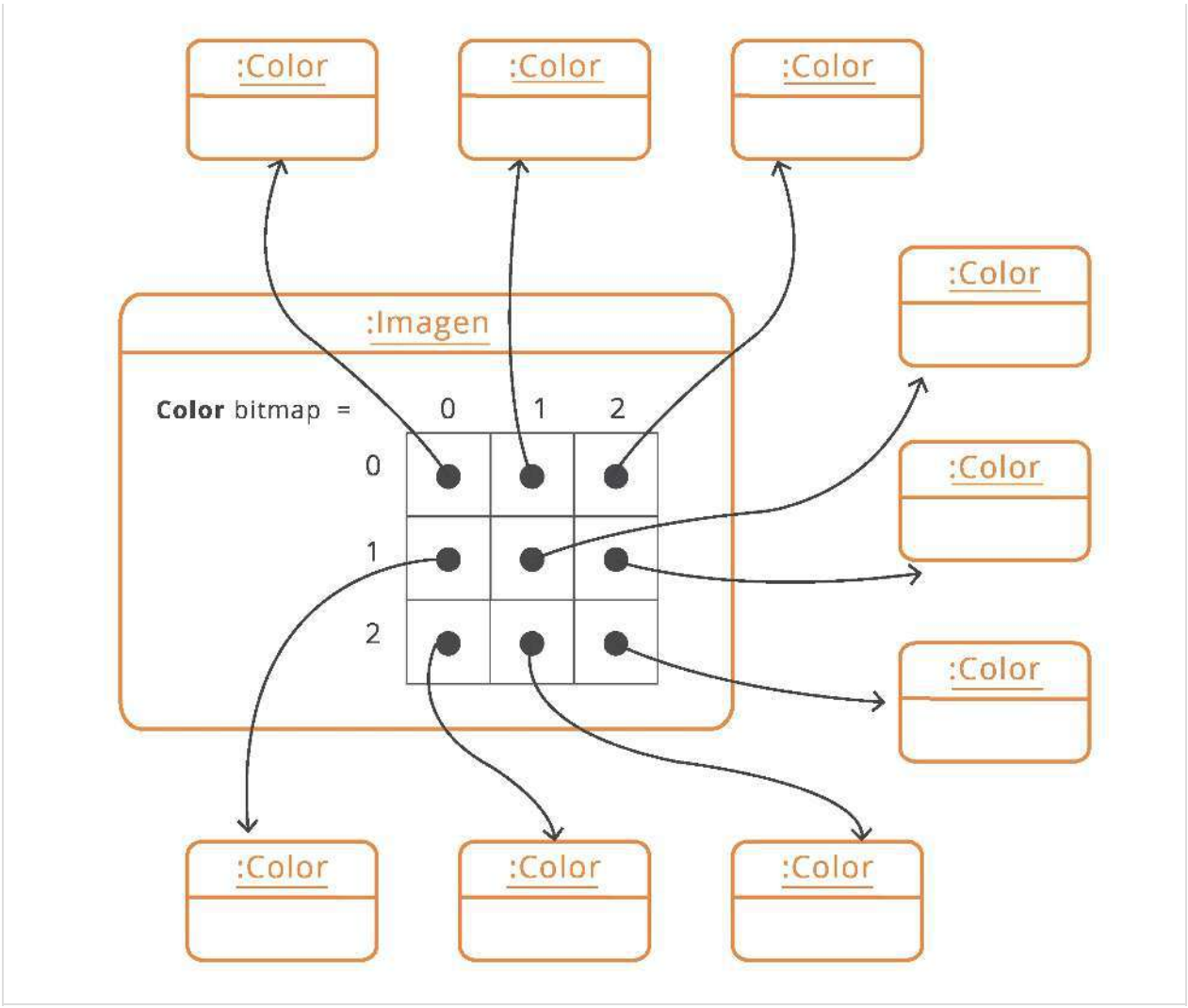
Para acceder a una posición de una matriz necesitamos dos **índices**, uno para indicar la fila y el otro para indicar la columna (por ejemplo, con la sintaxis `bitmap[5][6]` hacemos referencia al elemento de la casilla que está en la fila 5 en la columna 6). Recuerde que un índice es un valor entero y sus valores van desde 0 hasta el número de elementos de la dimensión correspondiente menos 1. Para tomar o modificar el valor de un elemento particular de una matriz necesitamos dar los dos índices. El siguiente ejemplo inicializa todos los elementos de bitmap en la clase Imagen con el color azul.

```
public void imagenAzul( )
{
    for( int i = 0; i < alto; i++ )
    {
        for( int j = 0; j < ancho; j++ )
        {
            bitmap[ i ][ j ] = new Color( 0, 0, 255 );
        }
    }
}
```

- Este método recorre la matriz inicializando las casillas con objetos de la clase Color cuyo valor representa el azul.
- Debemos saber que el color azul en el formato RGB se representa por los valores 0, 0, 255.
- Con la sintaxis `bitmap[ i ][ j ]` hacemos referencia a la casilla que se encuentra en la fila i columna j.
- Fíjese que en cada casilla queda una referencia a un objeto distinto de la clase Color (120.000 objetos distintos, si la imagen es de 300 x 400).

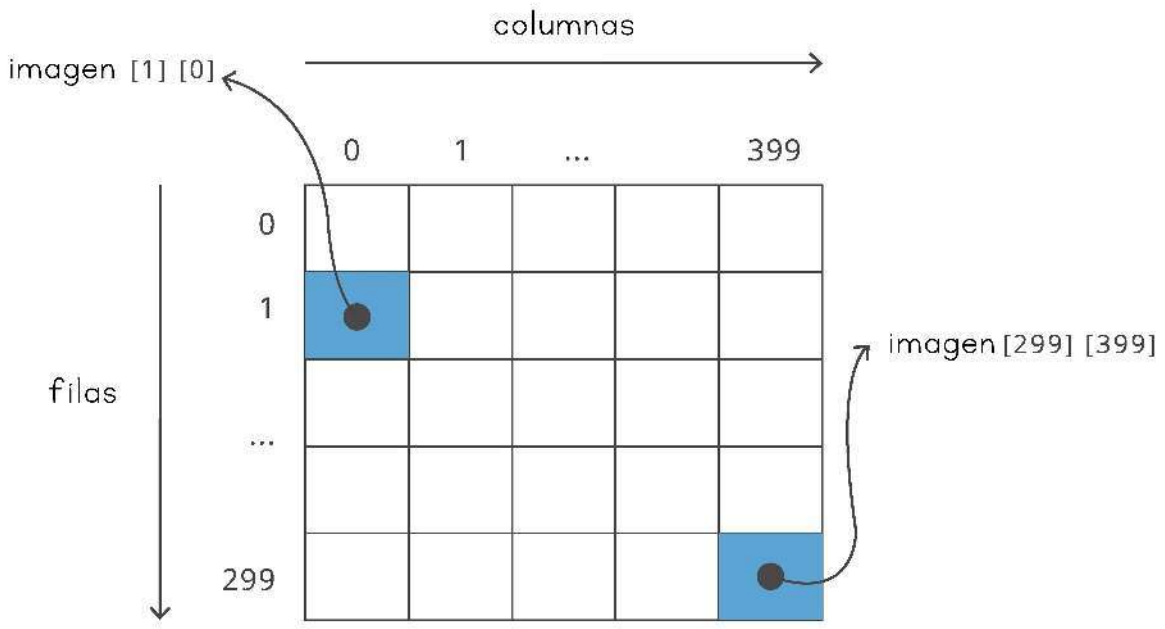
En la figura 6.5 se muestra el diagrama de objetos después de haber ejecutado el método anterior, suponiendo que la imagen es de 3 x 3.

Fig. 6.5 Diagrama de objetos para una imagen completamente azul de 3 x 3



Note que en el método del ejemplo anterior con el primer ciclo recorremos las filas empezando por la correspondiente al índice cero y terminando en la fila `alto-1` (vamos de arriba hacia abajo recorriendo las filas, como se muestra en la figura 6.6). Una vez que se fija una fila, el segundo ciclo nos permite recorrer las columnas de esa fila. Este recorrido se hace desde la columna 0 hasta la columna `ancho-1`. Note que cada vez que se termina con una fila, el ciclo interior vuelve a ejecutarse desde el principio e inicializa la columna en cero.

Fig. 6.6 Recorrido de la matriz con la imagen



El algoritmo anterior también se podría escribir utilizando la instrucción `while`, como se presenta a continuación:

```
public void imagenAzul( )
{
    int i = 0;

    while( i < alto )
    {
        int j = 0;

        while( j < ancho )
        {
            bitmap[ i ][ j ] = new Color( 0, 0, 255 );
            j++;
        }

        i++;
    }
}
```

- Este método hace la misma inicialización del ejemplo anterior, pero utiliza la instrucción while en lugar de la instrucción `for`.
- Con el índice i recorremos las filas, mientras que con el índice j recorremos las columnas.
- Dentro del ciclo interno, recorremos todas las columnas de la fila i (allí j va cambiando para pasar por todas las columnas de la matriz).

- En la condición del primer ciclo podría remplazarse el atributo alto por `bitmap.length`. Ambas expresiones hacen referencia al número de filas de la matriz.

> En la sintaxis de acceso a un elemento se pasa primero la fila en la que se encuentra y después la columna. Tanto las filas como las columnas se comienzan a numerar desde cero.
>
> Cuando dentro de un método tratamos de acceder a una casilla con un par de índices no válidos (al menos uno de ellos es menor que 0 o mayor que el máximo índice permitido para la dimensión correspondiente), obtenemos el error de ejecución: *java.lang.ArrayIndexOutOfBoundsException*

4.4. Comparar los Elementos de una Matriz

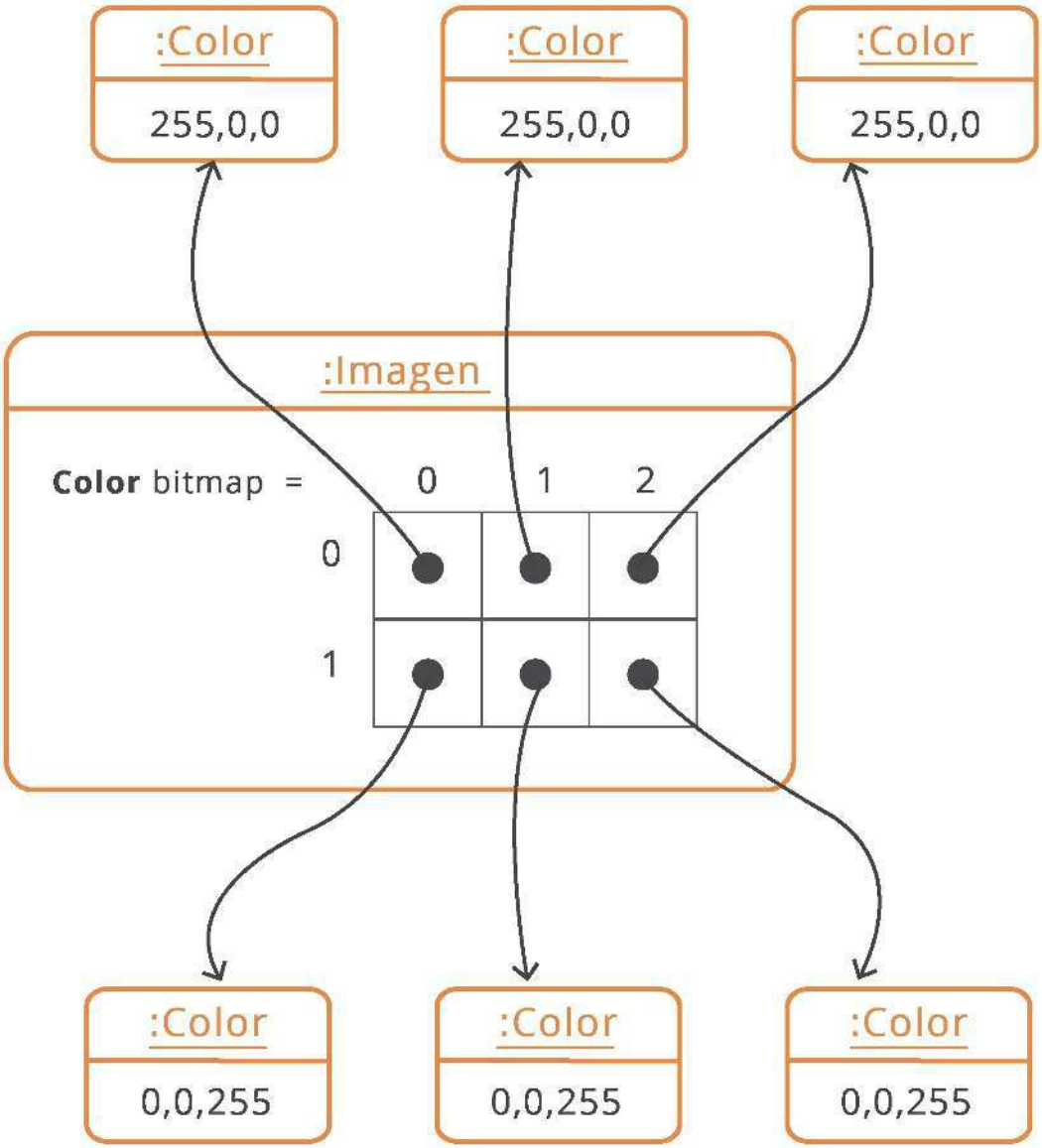
Si los elementos de una matriz son de un tipo simple (enteros, reales, etc.), se comparan utilizando el operador `==` que estudiamos en el segundo nivel. Después de todo, el estar almacenados en una matriz no cambia el hecho de que sean valores simples, y por lo tanto se deben seguir manipulando de la misma manera que hemos venido utilizando hasta ahora.

Cuando se trata de referencias a objetos hay que tener un poco más de cuidado. Si utilizamos el operador `==` estamos preguntando si las dos referencias señalan al mismo objeto físico y, a veces, no es eso lo que queremos saber. Para establecer si son iguales, aunque no estén referenciando el mismo objeto, se utiliza el método `equals`: piense por ejemplo que dos objetos pueden representar el color azul sin necesidad de ser el mismo objeto. Esta idea se ilustra en ejemplo 3. Si miramos un poco hacia atrás, esa es la razón por la cual siempre hemos comparado las cadenas de caracteres utilizando dicho método, en lugar del operador `==`. No nos importa que estén referenciando el mismo objeto, sino que contengan la misma cadena de caracteres.

Ejemplo 3

Objetivo: Mostrar la manera de comparar los elementos de una matriz, cuando dichos elementos son objetos.

En este ejemplo se muestra la diferencia entre comparar dos referencias a objetos utilizando el operador `==` y el método `equals`. También se ilustra la consecuencia de asignar a una variable una referencia a un objeto que ya está en una casilla de una matriz.



- Comenzamos este ejemplo mostrando un diagrama de objetos con una imagen de 2 x 3, cuya primera fila está coloreada de rojo (255,0,0) y la segunda de azul (0,0,255).
- Cada casilla tiene un objeto diferente que representa el color que allí aparece.
- La expresión `bitmap[0][0] == bitmap[0][1]` es falsa. Ambas referencias llevan a objetos que representan el color rojo, pero son objetos distintos.
- La expresión `bitmap[0][0].equals(bitmap[0][1])` es verdadera. Ambas referencias llevan a objetos que representan el color rojo y el método no tiene en cuenta que sean instancias distintas.
- La expresión `bitmap[0][0].equals(bitmap[1][0])` es falsa. El primer objeto representa el color rojo, mientras que el segundo representa el color azul.
- Si hacemos la siguiente asignación: `Color temp = bitmap[0][0]`, tenemos que tanto la variable temp como la casilla de coordenadas 0,0 referencian el mismo objeto. En ese caso la comparación `temp == bitmap[0][0]` es verdadera, lo mismo que la expresión `temp.equals(bitmap[0][0])`.

> El método equals() no está definido de manera adecuada en todas las clases. Algunas como String o Color sí lo tienen. Otras (como por ejemplo las que hemos desarrollado a lo largo de este libro), no lo tienen bien definido y si vamos a usar el método con esas clases nos tocaría implementarlo.

En este punto podemos retomar de nuevo la discusión planteada en la sección 4.3 sobre la imagen completamente azul: en vez de los miles de objetos para representar los píxeles (todos de color azul), ¿es posible utilizar un solo objeto con dicho fin? ¿Es posible que las 120.000 casillas de la matriz referencien todas el mismo objeto? La respuesta es que en este caso es posible, pero que dicha aproximación no se puede generalizar. En este caso lo podemos hacer porque la clase Color no tiene ningún método que permita a sus instancias modificar su valor. Si alguien quiere cambiar el color de un píxel debe crear un nuevo objeto de esa clase para representarlo. Esto tiene como consecuencia que, en el caso de estudio, sí podemos compartir el objeto azul de la clase Color desde todos los puntos de la imagen, ya que nadie puede cambiarlo. Si existiera un método en dicha clase que permitiera, por ejemplo, hacer más rojo un color, el hecho de utilizar un solo objeto compartido por todos haría que al cambiar de color un solo píxel, el cambio se traslade a todos los otros píxeles de la imagen que están siendo representados por el mismo objeto.

Tarea 1

Objetivo: Ilustrar la manera de escribir un algoritmo para manipular una matriz.

Complete el siguiente método de la clase Imagen. No olvide que para preguntar si dos colores son iguales, se debe utilizar el método equals de la clase Color.

```
/**
*   Devuelve el número de píxeles en la imagen cuyo color es el dado como parámetro.
*   @param pColorBuscado Objeto por el que se quiere preguntar. pColorBuscado != null.
*   @return Número de puntos en la matriz cuyo color es igual al dado.
*/
public int cuantosPixelColor( Color pColorBuscado )
{









}
```

4.5. Patrones de Algoritmo para Recorrido de Matrices

Las soluciones de muchos de los problemas que debemos resolver sobre matrices son similares entre sí y obedecen a ciertos esquemas ya conocidos. En esta sección pretendemos adaptar algunos de los patrones que estudiamos en el nivel 3 al caso de las matrices. De nuevo, lo ideal es que al leer un problema que debemos resolver (el método que debemos escribir), podamos identificar el patrón al cual corresponde y utilizar las guías que existen para resolverlo. Eso simplificaría enormemente la tarea de escribir los métodos que tienen ciclos y que trabajan sobre estructuras de matrices.

4.5.1. Patrón de Recorrido Total

Este patrón se aplica en las situaciones donde debemos recorrer todos los elementos que contiene la matriz para lograr la solución. En el caso de estudio de la imagen tenemos varios ejemplos de esto:

- Contar cuántos puntos en la imagen son de color rojo.
- Cambiar el color de todos los puntos en la imagen haciéndolos más oscuros.
- Cambiar cada color de la imagen por su negativo.
- Contar cuántos puntos en la imagen tienen la componente roja distinta de cero.

Para la solución de cada uno de esos problemas, se requiere siempre un recorrido de toda la matriz para poder cumplir el objetivo que se está buscando. Un primer ciclo para recorrer las filas y, luego, un ciclo por cada una de ellas para recorrer sus columnas.

Para lograr el recorrido total, tenemos que:

1. El índice para iniciar el primer ciclo debe empezar en cero.
2. La condición para continuar es que el índice sea menor que el número de filas de la matriz.
3. El avance consiste en sumarle uno al índice.
4. El cuerpo del segundo ciclo contiene el recorrido de las columnas y debe ser tal que (a) el índice debe comenzar en cero, (b) la condición para continuar es que el índice sea menor que el número de columnas de la matriz, (c) el avance consiste en sumarle uno al índice. Esa estructura que se repite en todos los algoritmos que necesitan un recorrido total es lo que denominamos el **esqueleto del patrón**, el cual se puede resumir con el siguiente fragmento de código:

```
for( int i = 0; i < NUM_FILAS; i++ )
{
    for( int j = 0; j < NUM_COLUMNAS; j++ )
    {
        <cuerpo del ciclo>
    }
}
```

- El patrón consiste en dos ciclos anidados: el primero para recorrer las filas, el segundo para recorrer las columnas de cada fila.

Lo que cambia en cada caso es lo que se quiere hacer en el cuerpo del ciclo. Aquí hay dos variantes principales. En la primera, todos los elementos de la matriz van a ser modifiados siguiendo alguna regla (por ejemplo, oscurecer el color de todos los puntos). Lo único que se hace en ese caso es remplazar el cuerpo del ciclo en el esqueleto por las instrucciones que hacen la modificación pedida para un elemento de la matriz. Damos un ejemplo de aplicación en el siguiente código (método de la clase Imagen), que oscurece una imagen:

```
for( int i = 0; i < alto; i++ )
{
    for( int j = 0; j < ancho; j++ )
    {
        bitmap[ i ][ j ] = bitmap[ i ][ j ].darker();
    }
}
```

- Partimos del esqueleto del patrón. Sólo cambiamos el cuerpo del segundo ciclo, para explicar la manera de modificar cada una de las casillas de la matriz.
- Toda modificación que hagamos allí para la casilla de coordenadas i, j, la estaremos haciendo para cada uno de los elementos de la estructura.
- El método darker() crea una nueva instancia de la clase Color, más oscura que el objeto que recibe la llamada.

La segunda variante del patrón es cuando se quiere calcular alguna propiedad sobre el conjunto de elementos de la matriz (por ejemplo, contar cuántos puntos tienen el componente rojo igual a cero). Como vimos en el nivel 3, esta variante implica cuatro decisiones que definen la manera de completar el esqueleto del patrón:

1. Cómo acumular la información que se va llevando a medida que avanza el segundo ciclo.
2. Cómo inicializar dicha información.
3. Cuál es la condición para modificar dicho acumulado en el punto actual del ciclo.
4. Cómo modificar el acumulado. Veamos esos puntos para resolver el problema de contar cuántos puntos tienen el componente rojo igual a cero.

¿Cómo acumular información? Vamos a utilizar una variable de tipo entero llamada `cuantosRojoCero` que va llevando el número de puntos que tienen el componente rojo en cero.

¿Cómo inicializar el acumulado? La variable `cuantosRojoCero` se debe inicializar en 0, antes de la primera iteración del ciclo exterior.

¿Condición para cambiar el acumulado? Cuando el método `getRed()` del objeto Color que se encuentra en `bitmap[i][j]` sea igual a 0.

¿Cómo modificar el acumulado? El acumulado se modifica incrementándolo en 1.

El método resultante es el siguiente:

```
public int rojoCero( )
{
    int cuantosRojoCero = 0;

    for( int i = 0; i < alto; i++ )
    {
        for( int j = 0; j < ancho; j++ )
        {
            if( bitmap[ i ][ j ].getRed( ) == 0 )
            {
                cuantosRojoCero++;
            }
        }
    }

    return cuantosRojoCero;
}
```

- Este método de la clase Imagen permite calcular el número de píxeles de la imagen cuyo componente rojo es cero.
- El método `getRed()` de la clase Color retorna el índice de rojo que tiene el objeto sobre el que se invoca el método. En este caso corresponde al color del objeto que se encuentra referenciado en la casilla (i,j).
- Si dicho método retorna el valor 0, debemos incrementar la variable en la que vamos acumulando el resultado.

Tarea 2

Objetivo: Generar habilidad en el uso del patrón de recorrido total para escribir un método que manipula una matriz.

Escriba los métodos de la clase Imagen que resuelven los siguientes problemas, que corresponden a las dos variantes del patrón de algoritmo de recorrido total.

Escriba un método que modifique los puntos de la matriz convirtiéndolos en sus negativos. El negativo se calcula restándole 255 a cada componente RGB del color y tomando el valor absoluto del resultado.

```
public void negativoImagen( )
{









}
```

Escriba un método que indique cuál es la tendencia de color de la imagen. Esto se calcula de la siguiente manera: un píxel tiene un color de tendencia roja, si su índice es mayor que los otros dos. Lo mismo sucede con los demás colores. Este método retorna 0 si la imagen no tiene ninguna tendencia, 1 si la tendencia es roja, 2 si la tendencia es verde y 3 si la tendencia es azul.

```
public int calcularTendencia( )
{









}
```

4.5.2. Patrón de Recorrido Parcial

Como vimos con los arreglos y con los vectores, algunos problemas de manejo de estructuras contenedoras no exigen recorrer todos los elementos para lograr el objetivo propuesto. Piense por ejemplo en el problema de saber si hay algún punto negro (0, 0, 0) en la imagen. En ese caso hacemos un recorrido que puede terminar cuando encontremos el primer punto negro o cuando lleguemos al final de la matriz sin haberlo encontrado. Un recorrido parcial se caracteriza porque existe una condición que debemos verificar en cada iteración para saber si debemos detener el ciclo o volverlo a repetir.

En este patrón debemos tener en cuenta la condición de salida de la siguiente manera:

```
boolean termino = false;

for( int i = 0; i < NUM_FILAS && !termino; i++ )
{

    for( int j = 0; j < NUM_COLUMNAS && !termino; j++ )
    {
        <cuerpo del ciclo>

        if( <problema terminado> )
        {
            termino = true;
        }
    }
}
```

- Este esqueleto es una variante del que utilizamos en el caso de los arreglos, con la diferencia de que utilizamos la variable `termino` para hacerlo salir de los dos ciclos a la vez.
- Tal como vimos en el nivel 3, la variable **termino** se puede reemplazar por cualquier condición que indique el punto en el que el problema ya ha sido resuelto.

Hay casos en los cuales se deben utilizar dos variables distintas para controlar la salida de cadauno de los ciclos de manera independiente. En ese caso se trata simplemente de aplicar el patrón de recorrido parcial de los arreglos de manera anidada, tal como se muestra en el siguiente esqueleto de algoritmo:

```
boolean termino1 = false;

for( int i = 0; i < NUM_FILAS && !termino1; i++ )
{

    boolean termino2 = false;

    for( int j = 0; j < NUM_COLUMNAS && !termino2; j++ )
    {
        <cuerpo del ciclo>

        if( <problema interno terminado> )
        {
            termino2 = true;
        }
    }

    if( <problema externo terminado> )
    {
        termino1 = true;
    }
}
```

- Con la variable `termino1` manejamos el recorrido parcial del ciclo externo. Cuando el problema que se quiere resolver con ese ciclo se da por resuelto, la variable cambia de valor y termina la instrucción repetitiva.
- Con la variable `termino2` hacemos lo mismo con el ciclo interno.
- De nuevo, las variables `termino1` y `termino2` se pueden reemplazar por expresiones lógicas que determinen si el objetivo de cada ciclo ya ha sido alcanzado.

En el ejemplo 4 se ilustra el uso de los dos esqueletos de algoritmo para resolver problemas de manipulación de matrices.

Ejemplo 4

Objetivo: Mostrar dos problemas de matrices que se resuelven utilizando los dos esqueletos planteados anteriormente.

En este ejemplo se presentan dos métodos de la clase Imagen cuya solución sigue el patrón de recorrido parcial de matrices.

```
public boolean hayPuntoNegro( )
{
    boolean termino = false;

    for( int i = 0; i < alto && !termino; i++ )
    {
        for( int j = 0; j < ancho && !termino; j++ )
        {
            if( bitmap[ i ][ j ].equals( Color.BLACK ) )
            {
                termino = true;
            }
        }
    }

    return termino;
}
```

- Este método nos permite saber si hay al menos un punto negro en la imagen.
- En este método, la condición para dar por resuelto el problema es que se encuentre en la casilla actual (i,j) un píxel negro. Ahí sabemos que la respuesta es verdadera, y queremos salir del ciclo interno y del ciclo externo a la vez.
- Si al llegar al final de todo el recorrido no hemos encontrado ningún píxel negro, debemos retornar falso.

```
public boolean muchasFilasConPixelNegro( )
{
    boolean termino1 = false;
    int numFilas = 0;

    for( int i = 0; i < alto && !termino1; i++ )
    {
        boolean termino2 = false;

        for( int j = 0; j < ancho && !termino2; j++ )
        {
            if( bitmap[ i ][ j ].equals( Color.BLACK ) )
            {
                numFilas++;
                termino2 = true;
            }
        }

        if( numFilas > 50 )
        {
            termino1 = true;
        }
    }

    return termino1;
}
```

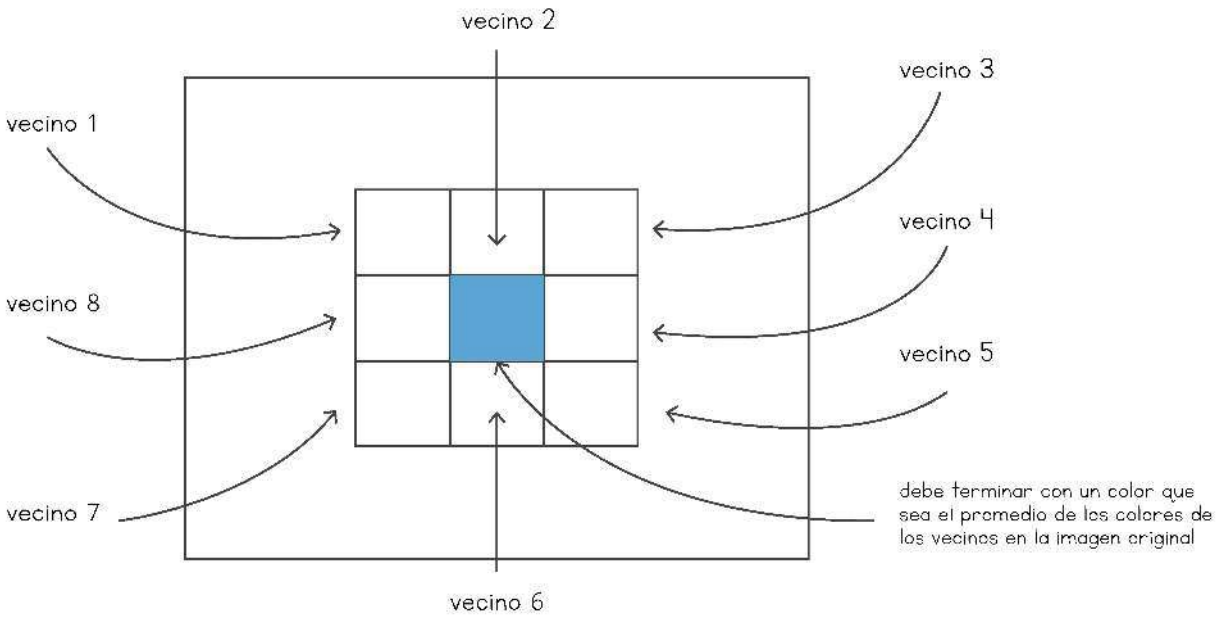
- Este método indica si hay más de 50 filas en la imagen con un píxel negro.
- El objetivo del ciclo exterior se cumple si se encuentran más de 50 filas con un píxel negro. La variable `termino1` debe cambiar de valor en ese caso y hacer que se termine la iteración.
- El objetivo del ciclo interior es encontrar un píxel negro en la fila i. Tan pronto lo encuentre, debe usar la variable `termino2` para dejar de iterar.
- Puesto que el problema planteado a cada ciclo termina en un momento distinto, no podemos utilizar una sola variable como habíamos hecho en el método anterior.
- En lugar de retornar el valor de la variable `termino1`, habríamos podido retornar la expresión `numFilas > 50`.

Tarea 3

Objetivo: Escribir algunos métodos para manipular matrices.

Desarrolle los métodos de la clase Imagen que resuelven los siguientes problemas. En cada caso, identifique el patrón de algoritmo que va a utilizar.

En el proceso de adquisición de una imagen, ésta puede quedar con una serie de errores los cuales hacen que se vea de mala calidad. Para corregir estos errores existe un algoritmo de filtrado, que se basa en cálcular un nuevo valor para cada píxel de la imagen. Este valor se calcula como el promedio de los 8 vecinos del píxel en la imagen original, sobre cada uno de los componentes RGB. En este proceso no se incluyen los bordes de la imagen, puesto que no tienen los 8 vecinos necesarios. Este método de la clase Imagen debe retornar una matriz con una copia de la imagen filtrada.




```
public Color[][] imagenFiltrada( )
{









}
```

En algunos contextos (en robótica, por ejemplo), en lugar del color exacto de cada píxel nos interesa solamente distinguir el fondo de la imagen (en blanco) de otros elementos que puedan aparecer (un obstáculo para el robot, por ejemplo). Escriba un método de la clase Imagen que modifique la matriz de píxeles de la siguiente manera: si la suma de los tres componentes RGB de un píxel es menor que 100, lo debe reemplazar por el color blanco (255,255,255). En caso contrario lo reemplaza por el color negro (0,0,0).

```
public void binarizar( )
{









}
```

Escriba un método de la clase Imagen que sea capaz de rotarla 90 grados a la derecha.

```
public void rotar90AlaDerecha( )
{









}
```

4.5.3. Otros Algoritmos de Recorrido

En el ejemplo 5 mostramos la manera de adaptar los patrones que hemos visto a algunos problemas típicos de manejo de matrices.

Ejemplo 5

Objetivo: Mostrar algunos problemas de matrices que pueden ser resueltos adaptando los patrones que hemos visto.

En este ejemplo se presentan tres métodos de la clase Imagen, cuya solución puede ser explicada a través de la adaptación de alguno de los patrones que hemos visto en este libro.

```
public int contarVerdes( int pNumFila )
{
    int numVerdes = 0;

    for( int i = 0; i < ancho; i++ )
    {
        if( bitmap[pNumFila][i].getGreen() == 255 )
        {
            numVerdes++;
        }
    }

    return numVerdes;
}
```

- En este método vamos a contar el número de píxeles de la fila `pNumFila` cuyo componente verde es el máximo posible.
- En este ejemplo queremos recorrer una fila de la matriz, cuyo índice se recibe como parámetro. El hecho de utilizar una sola fila hace que pasemos al contexto de las contenedoras de una sola dimensión y que apliquemos los patrones estudiados en el nivel 3.
- Aplicamos entonces el patrón de recorrido total sobre el arreglo representado por la fila dada. La única diferencia es que para indicar un elemento debemos usar la sintaxis `bitmap[pNumFila][i]`.

```
public int darSumaAzulColumna( int pNumColumna )
{
    int acumAzul = 0;

    for( int i = 0; i < alto; i++ )
    {
        acumAzul += bitmap[i][pNumColumna].getBlue();
    }
    return acumAzul;
}
```

- Este método calcula la suma del valor azul de todos los píxeles de la columna que recibe como parámetro.
- Basta con ver la columna número `pNmColumna` como un arreglo de longitud alto (el número de filas).
- Cada elemento se debe referenciar con la sintaxis `bitmap[i][pNumColumna]`.

```
public boolean negroEnDiagonal( )
{
    for( int i = 0; i < alto && i < ancho; i++ )
    {
        if( bitmap[ i ][ i ].equals( Color.BLACK ) )
        {
            return true;
        }
    }
    return false;
}
```

- Este método indica si hay un píxel negro sobre la diagonal de la imagen que comienza en el punto (0,0).
- Para este problema, vamos a imaginar el arreglo compuesto por los elementos de la diagonal: (0,0), (1,1), (2,2), etc.
- Luego, aplicamos el patrón de recorrido parcial de los arreglos. La única diferencia es que, al avanzar, debemos hacerlo a la vez sobre las dos dimensiones, de manera que nos movamos por la diagonal.

5. Caso de Estudio Nº 2: Campeonato de Fútbol

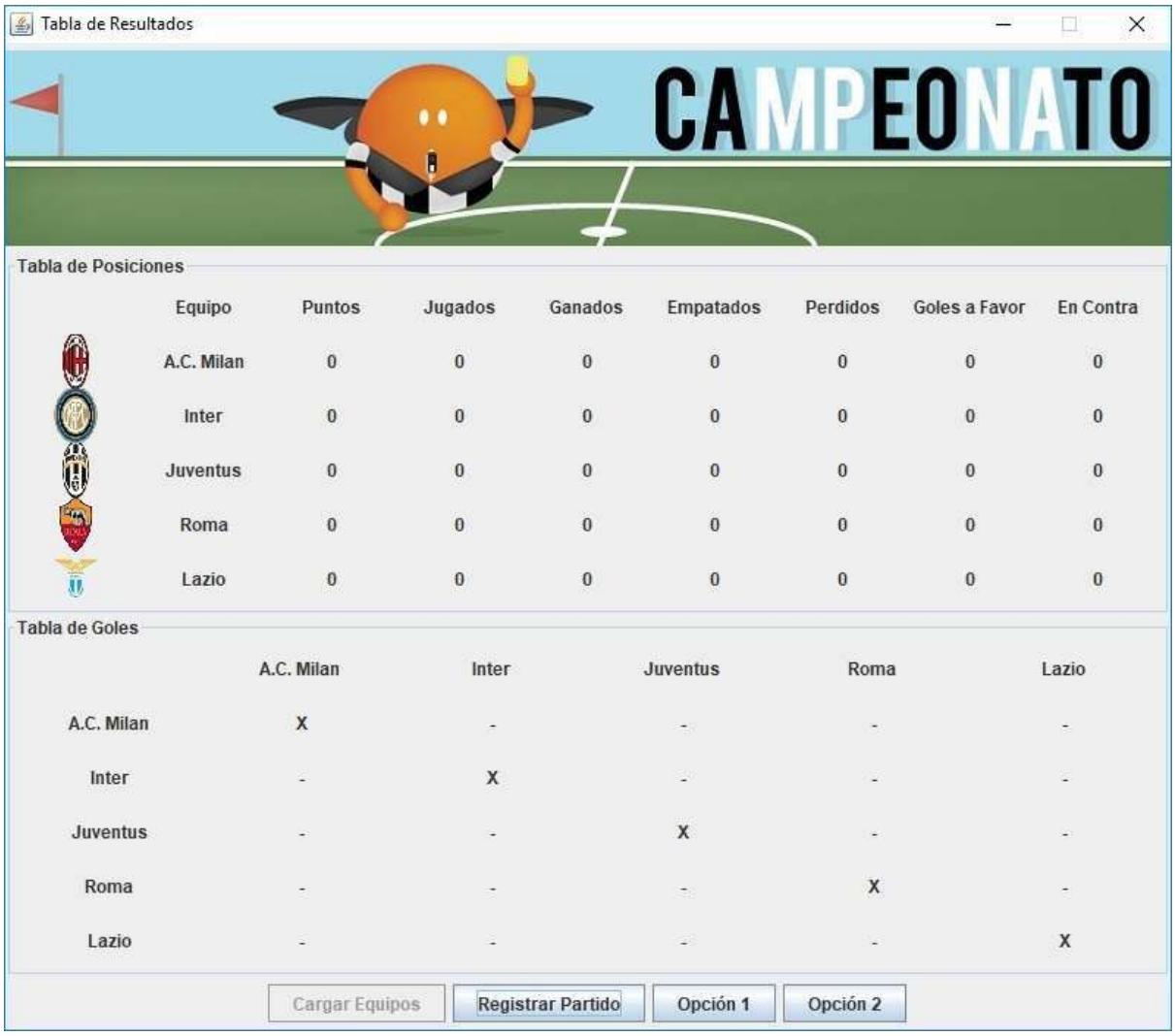
En este caso se quiere construir una aplicación para manejar los resultados de los partidos en un campeonato de fútbol. En el campeonato hay varios equipos y cada uno de ellos puede jugar contra cada uno de los otros equipos una sola vez.

La información de los equipos que participan del campeonato está definida en un archivo que la aplicación debe leer para construir el estado inicial. El formato de dicho archivo se explicará más adelante.

En el programa se debe permitir registrar el resultado de cualquier partido del campeonato y, con base en esa información, se debe mostrar la tabla de resultados, en la que se indique cuántos goles le hizo cada equipo a cada uno de los otros con los que ha jugado. También se debe mostrar la tabla de posiciones, indicando para cada equipo el número de puntos (Puntos), los partidos jugados (Jugados), los partidos ganados (Ganados), los partidos empatados (Empatados), los partidos perdidos (Perdidos), los goles a favor (Goles a Favor) y los goles en contra (En Contra).

La interfaz de usuario que hemos diseñado para esta aplicación es la que se muestra en la figura 6.7.

Fig. 6.7 Interfaz de usuario del caso de estudio del campeonato de fútbol



En esta interfaz se muestra permanentemente la tabla de goles y la tabla de posiciones de los equipos. Usando el botón Registrar Partido se ingresa el resultado de alguno de los partidos del campeonato. Con el botón Cargar Equipos se permite al usuario leer de un archivo los nombres de los equipos inscritos en el campeonato. El programa debe funcionar para cualquier número de equipos, pero una vez que se haya leído el archivo con los nombres, éstos no se pueden cambiar.

5.1. Comprensión de los Requerimientos

Los requerimientos funcionales de este caso de estudio son los siguientes:

1. Cargar equipos.
2. Registrar un resultado.
3. Mostrar tabla de goles.
4. Mostrar tabla de posiciones.

Requerimiento funcional 1

Nombre:	R1 - Cargar equipos.
Resumen:	Carga los equipos que van a tomar parte en el campeonato a través de un archivo de propiedades. La tabla de posiciones y tabla de goles se reinician.
Entradas:	Archivo de propiedades con los datos de los equipos.
Resultado:	Se muestran los equipos cargados y las tablas de goles y posiciones reiniciadas.

Requerimiento funcional 2

Nombre:	R2 - Registrar un resultado.
Resumen:	Registra el resultado de un partido en la tabla de goles y de posiciones. Si los equipos del partido ya tienen registrado un resultado para el mismo o si es un partido inválido (un equipo contra sí mismo) no se hace el registro de datos.
Entradas:	(1) equipo 1, (2) equipo 2, (3) goles del equipo 1 (4) goles del equipo 2.
Resultado:	Se actualiza la tabla de goles con los goles efectuados por los dos equipos y la tabla de posiciones con el partido jugado.

Requerimiento funcional 3

Nombre:	R3 - Mostrar tabla de goles.
Resumen:	Muestra la tabla de goles: para cada equipo se muestra el número de goles que le hizo a cada uno de los otros equipos.
Entradas:	Ninguna.
Resultado:	Se muestra la tabla de goles con los partidos registrados.

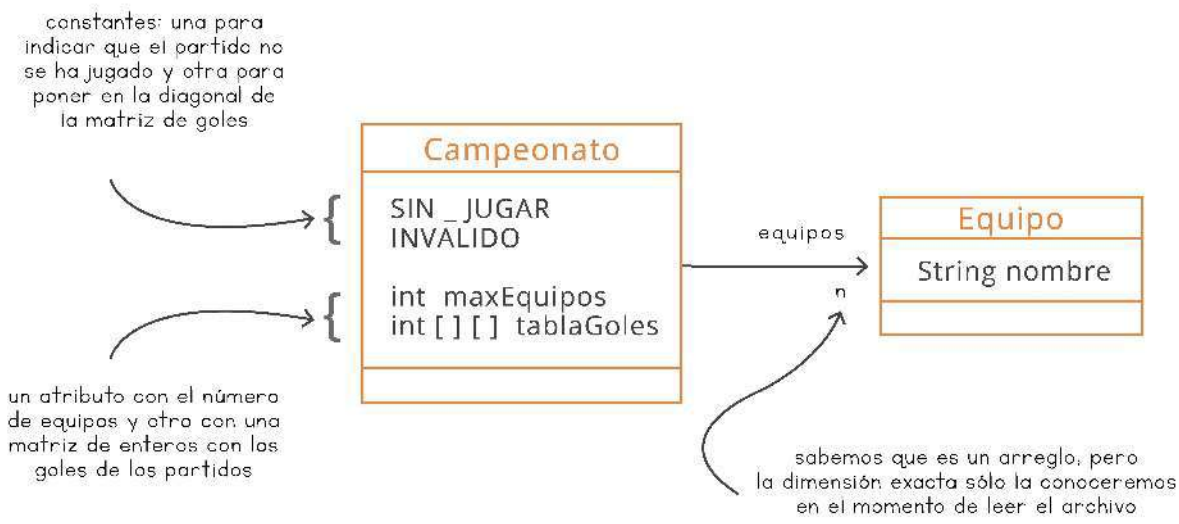
Requerimiento funcional 4

Nombre:	R4 - Mostrar tabla de posiciones.
Resumen:	Muestra la tabla de posiciones del campeonato. Para cada equipo se muestra el número de puntos, los partidos jugados, ganados, empatados y perdidos y el número de goles a favor y en contra.
Entradas:	Ninguna.
Resultado:	Se muestra la tabla de posiciones con los partidos registrados.

5.2. Comprensión del Mundo del Problema

En el mundo del problema podemos identificar dos entidades (ver figura 6.8): el campeonato y los equipos. La tabla de resultados la vamos a representar como una matriz de enteros, en la cual en la casilla (X, Y) está el número de goles que le hizo el equipo X al equipo Y. Si no han jugado, en dicha casilla almacenamos la constante `SIN_JUGAR`. En la diagonal ponemos el valor `INVALIDO` para indicar que un equipo no puede jugar contra sí mismo. El campeonato tiene un arreglo de equipos, cada uno de los cuales almacena su nombre.

Fig. 6.8 Modelo conceptual del campeonato de fútbol



5.3. Diseño de las Clases

5.3.1. Declaración de los Atributos y las Asociaciones

A continuación mostramos la manera de declarar en Java las clases involucradas en el problema, con una explicación de cómo se representa la información. De los métodos sólo mostramos algunas de las signaturas que utilizaremos más adelante.

```
public class Equipo
{
    //---------------------------------------
    // Atributos
    //---------------------------------------
    private String nombre;
    //---------------------------------------
    // Metodos
    //---------------------------------------
    public Equipo( String nombreEquipo ) {...}
    public String darNombre( ) {...}
    public String toString( ) {...}
}
```

- La clase Equipo tiene un único atributo que contiene su nombre.
- La clase cuenta con un constructor, que recibe como parámetro el nombre del equipo, y dos métodos: uno que retorna el nombre del equipo y otro que retorna un texto para representar el equipo como una cadena de caracteres.

```
public class Campeonato
{
    //---------------------------------------
    // Constantes
    //---------------------------------------
    public static final int SIN_JUGAR = -1;
    public static final int INVALIDO = -2;

    //---------------------------------------
    // Atributos
    //---------------------------------------
    private int maxEquipos;
    private int[][] tablaGoles;
    private Equipo[] equipos;
}
```

- Una decisión importante que debemos tomar al diseñar la clase es la manera de representar los equipos y la tabla de goles. Dado que el número de equipos que participan en el campeonato no cambia y que ésta es una información que vamos a leer del archivo de entrada, podemos modelar los equipos como un arreglo de tamaño fijo (equipos).
- La tabla de goles es una estructura de dos dimensiones en donde el número de columnas es igual al número de filas, y este valor corresponde al número de equipos que están participando en el campeonato. Dado que la información de los goles es un valor numérico los elementos serán de tipo entero.
- Interpretaremos la tabla de la siguiente manera: (a) `tablaGoles[equipo1][equipo2]` indicará el número de goles que el equipo1 le hizo al equipo2; (b) `tablaGoles[equipo2]`

`[equipo1]` indicará el número de goles que el equipo2 le hizo al equipo1.
- La constante `SIN_JUGAR` indica que el partido no se ha jugado todavía.
- La constante INVALIDO sólo se usa en la diagonal de la matriz (un equipo no puede jugar contra sí mismo).
- En el atributo `maxEquipos` almacenamos el número de equipos inscritos en el campeonato.

Dicho valor no debe cambiar después de ser cargado del archivo.

5.3.2. Asignación de Responsabilidades

Dado que la clase Campeonato contiene la información de los equipos y de los goles de los partidos jugados, esta clase es responsable de:

1. Dar la información sobre los equipos.
2. Dar la información sobre la tabla de goles.
3. Dar la información sobre la tabla de posiciones.
4. Cargar de un archivo la información del campeonato y guardarla en el arreglo de equipos.
5. Registrar el resultado de un partido.

Las cinco responsabilidades anteriores nos van a guiar en la definición de los métodos de la clase Campeonato. En la siguiente sección nos vamos a concentrar en el problema de cargar los datos del campeonato a partir de la información registrada en un archivo. Esto nos va a permitir que siempre que ejecutemos el programa encontremos el mismo estado inicial. El problema general de la persistencia, o sea, el hecho de guardar en un archivo los cambios hechos en el estado del modelo del mundo (el campeonato en nuestro caso) está fuera del alcance de este libro. En la siguiente sección estudiaremos un mecanismo simple de lectura de la información inicial de un programa desde un tipo especial de archivos en Java llamados archivos de **propiedades** (*properties*).

6. Persistencia y Manejo del Estado Inicial

En varios de los casos de estudio de este libro, hemos utilizado archivos de datos para configurar el estado inicial de la aplicación. Por ejemplo, en el caso de estudio del empleado (nivel 1) teníamos en un archivo su fotografía. En el caso de estudio de la tienda (nivel 2) teníamos en un archivo la imagen de cada producto. En este nivel, el visor de imágenes utiliza un archivo para leer la imagen que será manipulada por la aplicación. Todos esos ejemplos tienen en común que la información del archivo se emplea para inicializar el estado de la aplicación. En ningún caso hemos guardado resultados del programa en un archivo para hacerlos persistentes cuando la aplicación termine. Este problema de hacer persistir los cambios que hagamos en el estado del mundo está fuera del alcance de este libro.

En esta sección estudiaremos una forma sencilla de leer datos de un archivo, con el propósito de configurar el estado inicial de los elementos del modelo del mundo. Vamos a estudiar los conceptos básicos y luego resolveremos el requerimiento funcional de cargar la información del campeonato desde un archivo.

6.1. El Concepto de Archivo

El concepto de archivo no es nuevo para nosotros. Desde el primer caso de estudio de este libro hemos utilizado archivos: archivos de texto como los que contienen el código Java, archivos html como los que contienen la documentación del programa, archivos mdl con los diagramas de clases, etc. Los directorios en donde guardamos los archivos con los datos y todos los de- más directorios que manejamos en los proyectos son a su vez archivos.

De manera general, podemos definir un **archivo** como una entidad que contiene información que puede ser almacenada en la memoria secundaria del computador (el disco duro o un CD). Todo archivo tiene un nombre que permite identificarlo de manera única dentro del computador, el cual está compuesto por dos partes: la ruta (*path*) y el nombre corto. La ruta describe la estructura de directorios dentro de los cuales se encuentra el archivo, empezando por el nombre de alguno de los discos duros del computador. Veamos en la siguiente tabla un ejemplo que ilustre lo anterior:

Nombre completo:	**c:/dev/uniandes/cupi2/empleado/mundo/Empleado.java**
Nombre corto:	Empleado.java
Extensión o apellido:	.java
Ruta o camino:	c:/dev/uniandes/cupi2/empleado/mundo/

El carácter '/' es llamado el separador de nombres de archivos (file separator). Este separador depende del sistema operativo en el que estemos trabajando. Por ejemplo, en Windows se suele utilizar como separador el carácter '\' (backslash) mientras que en Unix y Linux se utiliza el carácter '/' (slash).

La extensión que opcionalmente acompaña el nombre del archivo es una convención para indicar el tipo de información que hay dentro del archivo. El tipo de información dentro del archivo determina el programa con el que el archivo puede ser manipulado. Por ejemplo, los archivos de texto pueden ser manipulados por editores de texto, los archivos con extensión .xls deben ser manipulados por el programa Microsoft Excel, etc.

Desde nuestros programas en Java podemos acceder y leer información de los archivos del disco, siempre y cuando conozcamos su nombre para poder localizarlo y, además, conozcamos el tipo de información que el archivo contiene para poderla leer. Los archivos que manejaremos en nuestros programas tienen un formato especial que llamamos de propiedades (properties). Apoyándonos en algunas clases de utilidad que Java nos ofrece, vamos a poder leer información desde estos archivos de una manera muy sencilla.

Las clases Java que permiten manejar archivos desde un programa se encuentran definidas en el paquete `java.io`, mientras que la clase que maneja las propiedades está en el paquete `java.util`.

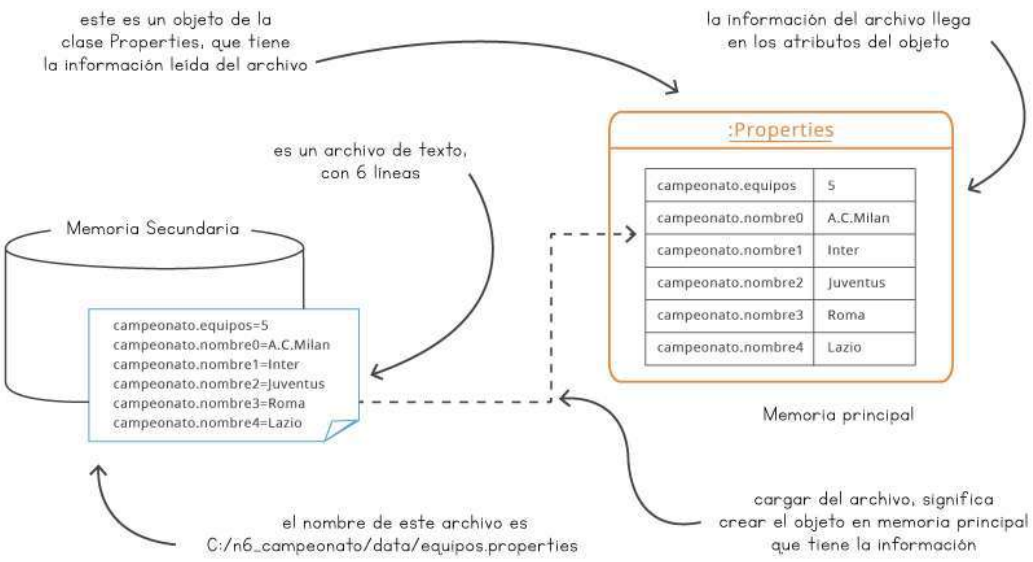
6.2. Leer Datos como Propiedades

Una propiedad se define como una pareja nombre = valor. Por ejemplo, para expresar en un archivo que la propiedad llamada campeonato.equipos tiene el valor 5, se usa la sintaxis:

```
campeonato.equipos = 5
```

En Java existe una clase llamada Properties que representa un conjunto de propiedades persistentes. Por persistentes queremos decir que estas propiedades pueden ser almacenadas en un archivo en memoria secundaria y leídas a la memoria del programa desde un archivo que ha sido escrito siguiendo las convenciones de nombre = valor. En la figura 6.9 se ilustra la correspondencia que queremos hacer entre un archivo llamado equipos.properties y un objeto de la clase Properties en memoria principal.

Fig. 6.9 Asociación entre un archivo y el objeto Properties en memoria principal



El archivo es un archivo de texto que contiene una lista de propiedades. Cada propiedad es una línea del archivo y está definida por un nombre, el operador `=` y el valor de la propiedad (sin necesidad de comillas). Si en nuestro programa, el objeto de la clase Properties está referenciado desde una variable llamada `pDatos`, una vez leído el archivo en memoria, podemos utilizar los métodos de dicha clase para obtener el valor de los elementos. Por ejemplo, si queremos saber el valor de la propiedad campeonato.nombre0, podemos utilizar el siguiente método, cuya respuesta será la cadena "A.C.Milan".

```
String nombre = pDatos.getProperty ( "campeonato.nombre0" );
```

Para completar el ejemplo, necesitamos aprender varias cosas. Primero necesitamos saber cómo localizar el archivo en el disco, luego hacer la asociación entre el archivo físico y un objeto en el programa que lo represente, y después, leer o cargar el contenido del archivo en el objeto Properties de nuestro programa. En las siguientes secciones veremos en detalle cada uno de estos pasos.

> Por convención, para los nombres de las propiedades utilizamos una secuencia de palabras en minúsculas, separadas por un punto.

6.3. Escoger un Archivo desde el Programa

Como explicamos en la sección de definición de un archivo, el nombre físico de un archivo depende del sistema operativo en el que nuestro programa esté trabajando, en particular porque el carácter de separación de directorios puede cambiar entre los diferentes sistemas

operativos. Por esta razón, para no depender del sistema operativo, en Java se puede hacer una abstracción de este nombre específico y convertirlo en un nombre independiente utilizando la clase File.

Para crear un objeto de la clase File que contenga la representación abstracta del archivo físico, debemos crear una instancia de dicha clase, usando la sintaxis que se muestra a continuación:

```
File archivoDatos = new File( "C:\n6_campeonato\data\equipos.properties" );
```

> Si invocamos el constructor de la clase File con una cadena vacía (`null`), se disparará la excepción: *java.lang.NullPointerException*

La clase File nos ofrece varios servicios muy útiles, como métodos para saber si el archivo existe, preguntar por las características del archivo, crear un archivo vacío, renombrar un archivo y muchas otras más. En este nivel no las vamos a estudiar en detalle pero el lector interesado puede consultar la documentación de la clase.

Con la instrucción del ejemplo anterior, obtenemos una variable llamada `archivoDatos` que está haciendo referencia a un objeto de la clase File que representa en abstracto el archivo que queremos leer. Lo anterior es suficiente si conocemos con anticipación el nombre del archivo de donde queremos cargar la información. Pero si, como en el caso de estudio, queremos que sea el cliente quien seleccione el archivo que quiere abrir, debemos utilizar otra manera de construir dicho objeto. Esto se ilustra en el ejemplo 6.

Ejemplo 6

Objetivo: Mostrar la manera de permitir al usuario escoger un archivo de manera interactiva.

En este ejemplo se presenta el código que permite a un programa preguntarle al usuario el archivo a partir del cual quiere leer alguna información.

```
public class InterfazCampeonato extends JFrame
{
    public void cargarEquipos( )
    {
        ...
    }
}
```

- El método `cargarEquipos()` de la clase InterfazCampeonato es responsable de preguntar al usuario el archivo del cual quiere cargar la información del campeonato.

- Veamos paso a paso la construcción de dicho método, comenzando por la manera de presentar la ventana de archivos disponibles en el computador y, luego, recuperar la selección que haya hecho el usuario.


```
public class InterfazCampeonato extends JFrame
{
    public void cargarEquipos( )
    {
        ...

        JFileChooser fc = new JFileChooser( "./data" );
        fc.setDialogTitle("Abrir archivo de campeonato");
        ...
    }
    ...
}
```

- Lo primero que debemos hacer en el método es utilizar la clase FileChooser, que permite seleccionar un archivo. Creamos una instancia de dicha clase, pasándole en el constructor el directorio por el cual queremos comenzar la búsqueda de los archivos. En nuestro caso, indicamos que es el directorio llamado data.
- En la segunda instrucción de esta parte, cambiamos el título de la ventana.

```
public class InterfazCampeonato extends JFrame
{
    public void cargarEquipos( )
    {
        ...

        File archivoCampeonato = null;
        int resultado = fc.showOpenDialog( this );

        if( resultado == JFileChooser.APPROVE_OPTION )
        {
            archivoCampeonato = fc.getSelectedFile( );
        }

        ...
        // Aquí debe ir la lectura del archivo
    }
}
```

- Con el método `showOpenDialog` hacemos que la ventana de selección de archivos se abra.
- Mientras el usuario no seleccione un archivo o cancele la operación, el método queda bloqueado en ese punto.
- El método `showOpenDialog` retorna un valor entero que describe el resultado de la operación.
- Con el método `getSelectedFile` obtenemos el objeto de la clase File que describe el archivo escogido por el usuario (sólo si el usuario no canceló la operación).

El código del ejemplo 6 está incompleto, porque hasta ahora sólo hemos obtenido un objeto de la clase File que representa el archivo que el usuario quiere cargar en memoria. En la próxima sección veremos cómo realizar la lectura propiamente dicha.

6.4. Inicialización del Estado de la Aplicación

Para cargar el estado inicial del campeonato, debemos leer del archivo de propiedades la información sobre el número de equipos que van a participar (propiedad llamada "campeonato.equipos") y el nombre de los equipos (propiedades llamadas "campeonato.equipo" seguido de un índice que comienza en cero). Con dicha información podremos inicializar nuestro arreglo de equipos y, también, la matriz que representa la tabla de goles. El constructor de la clase Campeonato será el encargado de hacer esta inicialización, que vamos a dividir en tres subproblemas para los que hemos identificado tres metas intermedias:

- **Meta 1:** Cargar la información del archivo en un objeto Properties.

- **Meta 2:** Inicializar el arreglo de equipos con base en la información leída.
- **Meta 3:** Inicializar la matriz que representa la tabla de goles.
- La primera de estas metas se logra con los métodos explicados en el ejemplo 7.

Ejemplo 7

Objetivo: Mostrar la manera de crear un objeto de la clase Properties a partir de la información de un archivo.

En este ejemplo se muestra el código del constructor de la clase Campeonato, en términos de los métodos que resuelven cada una de las metas intermedias. Luego se muestra el método privado que logra la primera de ellas. Los demás métodos serán presentados más adelante.

```
public class Campeonato
{
    //-----------------------------------
    // Atributos
    //-----------------------------------
    private int maxEquipos;
    private int[][] tablaGoles;
    private Equipo[] equipos;

    //-----------------------------------
    // Constructor
    //-----------------------------------
    public Campeonato( File pArchivo ) throws Exception
    {
        Properties datos = cargarInfoCampeonato( pArchivo );
        inicializarEquipos( datos );
        inicializarTablaGoles( );
    }
}
```

- El constructor recibe como parámetro el objeto de la clase File que describe el archivo con la información.
- Dicho objeto viene desde la interfaz del programa (obtenido con el método del ejemplo 6).
- El constructor lanza una excepción si encuentra un problema al leer el archivo o si el formato interno del mismo es inválido.
- El primer método carga la información del archivo en un objeto llamado datos.
- El segundo método recibe dicho objeto e inicializa el arreglo de equipos.
- El tercer método aprovecha la información dejada en los atributos, para crear la matriz con la tabla de goles.

```
private Properties cargarInfoCampeonato( File pArchivo ) throws Exception
{
    Properties datos = new Properties( );
    FileInputStream in = new FileInputStream( pArchivo );

    try
    {
        datos.load( in );
        in.close( );
    }
    catch( Exception e )
    {
        throw new Exception( "Formato inválido" );
    }

    return datos;
}
`
```

- Este método recibe un objeto de la clase File.
- Lo primero que hacemos es crear un objeto de la clase Properties (llamado `datos`) en el cual vamos a dejar el resultado del método.
- Luego creamos un objeto de la clase FileInputStream que nos ayuda a hacer la conexión entre la memoria secundaria y el programa.
- La clase FileInputStream sirve para crear una especie de "canal" por donde los datos serán transmitidos. Para construir este objeto y asociarlo con el archivo seleccionado por el usuario, usamos el objeto de la clase File que recibimos como parámetro.
- Si el archivo referenciado por `pArchivo` no existe al tratar de crear la instancia de la clase FileInputStream se lanza una excepción.
- Luego, usamos el método `load` de la clase Properties, pasándole como parámetro el "canal de lectura". Dicho método lanza una excepción si encuentra que el formato del archivo no es el esperado (no está formado por parejas de la forma nombre = valor). Allí atrapamos la excepción y la volvemos a lanzar con un mensaje significativo para nuestro programa.
- Finalmente cerramos el "canal de lectura" con el método close.

```
private void inicializarTablaGoles( )
{
    tablaGoles = new int[ maxEquipos ][ maxEquipos ];

    for( int i = 0; i < maxEquipos; i++ )
    {
        for( int j = 0; j < maxEquipos; j++ )
        {
            if( i != j )
            {
                tablaGoles[ i ][ j ] = SIN_JUGAR;
            }
            else
            {
                tablaGoles[ i ][ j ] = INVALIDO;
            }
        }
    }
}
```

- Este es el método que logra la tercera meta planteada en el constructor.
- Crea inicialmente una matriz que tiene una fila y una columna por cada equipo en el campeonato (es una matriz cuadrada).
- Luego inicializa cada una de las casillas de la matriz de enteros (patrón de recorrido total), usando para esto las constantes definidas en la clase.
- En la diagonal deja el valor INVALIDO.

6.5. Manejo de los Objetos de la Clase Properties

Para resolver la segunda meta, debemos implementar el método inicializarEquipos cuyo objetivo es inicializar el arreglo de equipos a partir de la información que recibe como parámetro de entrada. Para hacer esto necesitamos acceder al valor de las propiedades individuales que vienen en el objeto Properties. Esto se hace usando el método getProperty de la clase Properties, pasando como parámetro el nombre de la propiedad que queremos obtener (por ejemplo,"campeonato.equipos"). Veamos el código en el siguiente ejemplo.

Ejemplo 8

Objetivo: Mostrar la manera de acceder a las propiedades que forman parte de un objeto de la clase Properties.

En este ejemplo se muestra el código del método que implementa la segunda meta intermedia del constructor de la clase Campeonato.

```
private void inicializarEquipos( Properties pDatos )
{
    String strNumeroEquipos = pDatos.getProperty( "campeonato.equipos" );
    maxEquipos = Integer.parseInt( strNumeroEquipos );
    equipos = new Equipo[ maxEquipos ];

    for( int i = 0; i < maxEquipos; i++ )
    {
        String nombreEquipo = datos.getProperty( "campeonato.nombre" + i);
        equipos[ i ]= new Equipo( nombreEquipo );
    }
}
```

- Comenzamos obteniendo la propiedad que define el número de equipos del campeonato (llamada "campeonato.equipos"). El valor de una propiedad siempre es una cadena de caracteres.
- Luego, convertimos la respuesta que obtenemos en un entero, usando el método parseInt. Por ejemplo, convertimos la cadena "5" en el entero de valor 5. Note que dejamos el resultado en el atributo `maxEquipos` previsto para tal fin.
- Creamos después el arreglo de equipos, reservando suficiente espacio para almacenar los objetos de la clase Equipo que van a representar cada uno de ellos.
- En un ciclo recuperamos los nombres de los equipos (a partir de las propiedades), y con esa información vamos creando los objetos de la clase Equipo que los representan y los vamos guardando secuencialmente en las casillas del arreglo.
- Los nombres de los equipos vienen en las propiedades "campeonato. nombre0", "campeonato.nombre1", etc., razón por la cual calculamos dicho nombre dentro del ciclo, agregando al final el índice en el que va la iteración.

7. Completar la Solución del Campeonato

En esta sección vamos a mostrar los métodos que nos van a permitir implementar los requerimientos funcionales no cubiertos hasta ahora, y vamos a trabajar en la construcción de algunos métodos que, aunque no forman parte de los requerimientos, ayudarán al lector a generar habilidad en el uso de los patrones de algoritmo.

7.1. Registrar el Resultado de un Partido

Retomando las clases y sus responsabilidades, hemos establecido que la clase Campeonato tiene la información sobre los equipos que están jugando y sobre la tabla de goles. Veamos cómo podemos resolver el requerimiento de registrar el resultado de un partido. Este método se compromete en su contrato a realizar la actualización de la tabla de goles o a disparar una excepción si los datos entregados no son válidos. El código de la solución se muestra en el ejemplo 9.

Ejemplo 9

Objetivo: Mostrar el método que implementa el requerimiento funcional de registrar el resultado de un nuevo partido.

En este ejemplo se muestra el método de la clase Campeonato encargado de incluir el resultado del partido jugado por dos equipos Si los datos de entrada son inválidos, el método lanza una excepción.

```
public void registrarResultado( int pEquipo1, int pEquipo2, int pGol1, int pGol2 ) thr
ows Exception
{
    if( pEquipo1 < 0 || pEquipo1 >= maxEquipos || pEquipo2 < 0 || pEquipo2 >= maxEquip
os )
    {
        throw new Exception( "Equipos incorrectos" );
    }

    if( pEquipo1 == pEquipo2 )
    {
        throw new Exception( "Son el mismo equipo" );
    }

    if( pGol1 < 0 || pGol2 < 0 )
    {
        throw new Exception( "Número de goles inválido" );
    }

    if( tablaGoles[ pEquipo1 ][ pEquipo2 ] != SIN_JUGAR || tablaGoles[ pEquipo2 ][ pEq
uipo1 ] != SIN_JUGAR )
    {
        throw new Exception( "Partido ya jugado" );
    }

    tablaGoles[ pEquipo1 ][ pEquipo2 ] = pGol1;
    tablaGoles[ pEquipo2 ][ pEquipo1 ] = pGol2;
}
```

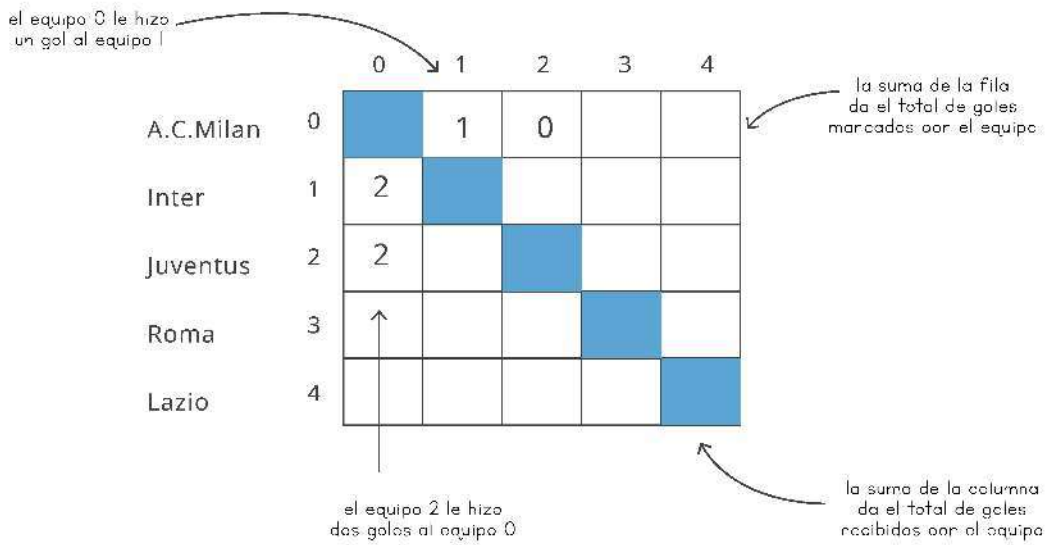
- El método supone que la matriz de goles ya fue inicializada (esto forma parte del contrato, en la parte de precondición).
- `pEquipo1` es el índice dentro de la matriz que identifica el primer equipo.
- `pEquipo2` es el índice dentro de la matriz que identifica el segundo equipo.
- `pGol1` es el número de goles marcados por el primer equipo (`pEquipo1`).
- `pGol2` es el número de goles marcados por el segundo equipo (`pEquipo2`).
- La mayor parte del método se dedica a validar la información recibida en los parámetros de entrada.
- Cuando los valores de los parámetros han sido validados, debemos actualizar las posiciones de la matriz que representan el partido entre los equipos `pEquipo1` y `pEquipo2` .

7.2. Construir la Tabla de Posiciones

De acuerdo con la definición de la tabla de posiciones, por cada equipo del campeonato debemos informar sus partidos jugados, partidos ganados, partidos empatados, partidos perdidos, goles a favor y goles en contra. Todos estos datos se pueden calcular a partir de la matriz que tiene la tabla de resultados, y eso es lo que haremos en esta sección.

En la figura 6.10 se muestra un escenario posible del campeonato: se han jugado dos partidos, en los cuales A.C. Milán perdió contra el Inter por un marcador de 1 a 2, y también perdió contra el Juventus recibiendo dos goles y no marcando ninguno. ¡Mal inicio de temporada para el A.C. Milán! En dicho escenario, el índice del A.C. Milán es el cero, mientras que el índice del Juventus es el 2.

Fig. 6.10 Tabla de goles del campeonato italiano



En la siguiente tabla se resumen algunas de las características de la tabla:

`tablaGoles[ 2 ][ 0 ] > tablaGoles[ 0 ][ 2 ]`	**Indica que el equipo con índice 2 le ganó el partido al equipo con índice 0.**
`tablaGoles[ 2 ][ 0 ] == tablaGoles[ 0 ][ 2 ]`	Indica que los equipos 0 y 2 empataron en el partido que jugaron.
La suma de todas las casillas de la fila 0	Indica el número total de goles marcados por el equipo 0 en todo el campeonato.
La suma de todas las casillas de la columna 0	Indica el número total de goles recibidos por el equipo 0 en todo el campeonato.
`tablaGoles[ i ][ i ] == INVALIDO`	Las casillas de la diagonal siempre van a tener el valor INVALIDO. Dichas casillas se deben ignorar en el momento de calcular los valores mencionados anteriormente.
Si `tablaGoles[ 2 ][ 0 ] == SIN_JUGAR`, entonces `tablaGoles[ 0 ][ 2 ] == SIN_JUGAR`	Si en la casilla (i, j) no hay un resultado, en la casilla simétrica (j, i) tampoco puede haberlo.

Tarea 4

Objetivo: Construir los métodos que nos van a permitir calcular la información de los equipos.

Escriba los métodos de la clase Campeonato que resuelven los problemas que se mencionan a continuación. Identifique el patrón de algoritmo que se debe aplicar en cada caso.

Calcular el número total de partidos ganados por el equipo que se recibe como parámetro.

```
public int partidosGanados( int pEquipo )
{









}
```

Calcular el número total de partidos empatados por el equipo que se recibe como parámetro.

```
public int partidosEmpatados( int pEquipo )
{









}
```

Calcular el número total de partidos jugados por el equipo que se recibe como parámetro.

```
public int partidosJugados( int pEquipo )
{









}
```

Calcular el número total de goles marcados por el equipo que se recibe como parámetro.

```
public int golesAFavor( int pEquipo )
{









}
```

Calcular el número total de puntos del equipo que se recibe como parámetro. Tenga en cuenta que un equipo recibe 3 puntos por cada partido ganado y un punto por cada partido empatado.

```
public int calcularTotalPuntos( int pEquipo )
{









}
```

7.3 Implementación de otros Métodos sobre Matrices

Tarea 5

Objetivo: Construir algunos métodos adicionales al caso de estudio, que ayuden a generar habilidad en la construcción de algoritmos para manejar matrices.

Escriba los métodos de la clase Campeonato que se describen a continuación. Identifique en cada caso el patrón de algoritmo que debe utilizar.

Retornar el índice del equipo que va ganando el campeonato. Si hay dos equipos con el mismo número de puntos, gana aquél cuya diferencia de goles (goles anotados menos goles recibidos) sea mayor.

```
public int calcularGanador( )
{









}
```

Calcular el número de partidos que faltan por jugar en el campeonato.

```
public int calcularPorJugar( )
{









}
```

Calcular el mayor número de goles marcados en un partido del campeonato (sumando los goles de los dos equipos).

```
public int calcularTotalGoles( )
{









}
```

Calcular el número de partidos del campeonato cuyo marcador fue cero a cero.

```
public int calcularTotalCeroACero( )
{









}
```

8. Proceso de Construcción de un Programa

Vamos a terminar este nivel con un resumen del proceso de construcción de un programa. Las actividades que se necesitan para construir un programa las hemos venido definiendo y practicando a lo largo de todo el libro. En los distintos niveles, dependiendo del tema tratado, hemos hecho énfasis en algunas de las tareas. Es importante recordar que este proceso de construcción de programas está pensado para construir programas pequeños (pocos requerimientos, pocas clases e interfaces gráficas simples) que, básicamente, pueden ser resueltos por un sólo desarrollador. Para programas más grandes en donde sea necesario que participen más desarrolladores, se requieren procesos distintos y actividades extra, relacionadas con la coordinación y sincronización del trabajo y, en general, con el manejo de la complejidad adicional que resulta de una mayor cantidad de requerimientos y del elevado número de clases necesarias para conformar la solución final.

El proceso de construcción de un programa es el conjunto de actividades que debemos seguir para terminar con éxito nuestra tarea. Éxito significa que al final tenemos un programa que funciona correctamente de acuerdo con los requerimientos, tiene su documentación completa (modelo del mundo, diseño de la interfaz, etc.) y, además, el código está documentado con los contratos y con los comentarios adicionales que permitirán a cualquier persona, más adelante, entenderlo y darle mantenimiento.

El proceso que hemos seguido se compone de tres actividades principales: análisis del problema, diseño de la solución y construcción de la solución. Lo importante de estas actividades es comprender cuál es su objetivo y qué artefactos debemos producir en cada una de ellas. Veamos una rápida síntesis de esas actividades.

8.1. Análisis del Problema

Objetivo:

- Entender el problema y poder explicar a otros nuestro entendimiento, siguiendo un conjunto de convenciones.

Resultados:

- Los requerimientos funcionales quedan consignados en un documento donde se identifican los servicios que el programa debe ofrecer al usuario. Cada uno de ellos debe tener una pequeña descripción que resuma el objetivo, la información de entrada

(suministrada por el usuario) y el resultado (producido por el programa).

- El modelo conceptual del mundo del problema es una simplificación de la realidad en la cual ocurre el problema. Este modelo lo expresamos en un diagrama de clases escrito en el lenguaje UML. En un diagrama de clases aparecen las entidades del mundo que participan en el problema, los atributos que permiten expresar su estado y las relaciones (llamadas asociaciones) existentes entre las entidades. Las asociaciones pueden tener un nombre y una cardinalidad. Esta última expresa el número de instancias involucradas en la relación entre las entidades.
- Los requerimientos no funcionales son las restricciones y condiciones que impone el cliente sobre el programa que se va a construir. Casi siempre hacen referencia al tipo de persistencia de la información, a las características de la interfaz de usuario, al manejo de la seguridad, etc. En este libro no tocamos este tema, dado que los problemas sobre los cuales trabajamos son pequeños, y los requerimientos no funcionales no influyen sobre la arquitectura de la solución.

8.2. Diseño de la Solución

Objetivo:

- Detallar las características que tendrá la solución, antes de ser construida. Los diseños nos van a permitir mostrar la solución antes de comenzar el proceso de fabricación propiamente dicho.

Resultados:

- La interfaz de usuario es la parte de la solución que permite que el usuario interactúe con el programa. Diseñarla significa que debemos producir dos artefactos: la visualización y el modelo conceptual de las clases que la van a componer (expresado en UML).
- La arquitectura nos ayuda a descomponer la solución en partes y a identificar sus relaciones. En los ejemplos de este libro, hemos utilizado un diagrama de paquetes para mostrar los tres componentes de la aplicación: la interfaz de usuario, el mundo y las pruebas.
- El diseño de las clases involucra la actividad más difícil de todas las que hemos visto en este libro. Esta actividad es la de asignación de responsabilidades. Como guía en la asignación de responsabilidades podemos utilizar los requerimientos funcionales para identificar los servicios esperados de cada clase. Tratamos de descomponer los requerimientos en servicios puntuales y, luego, de acuerdo con la técnica básica del experto, decidimos qué clases deben resolver cada uno de los métodos identificados. Al interior de cada clase diseñamos luego sus métodos, definiendo su contrato y su signatura.

8.3. Construcción de la Solución

Objetivo:

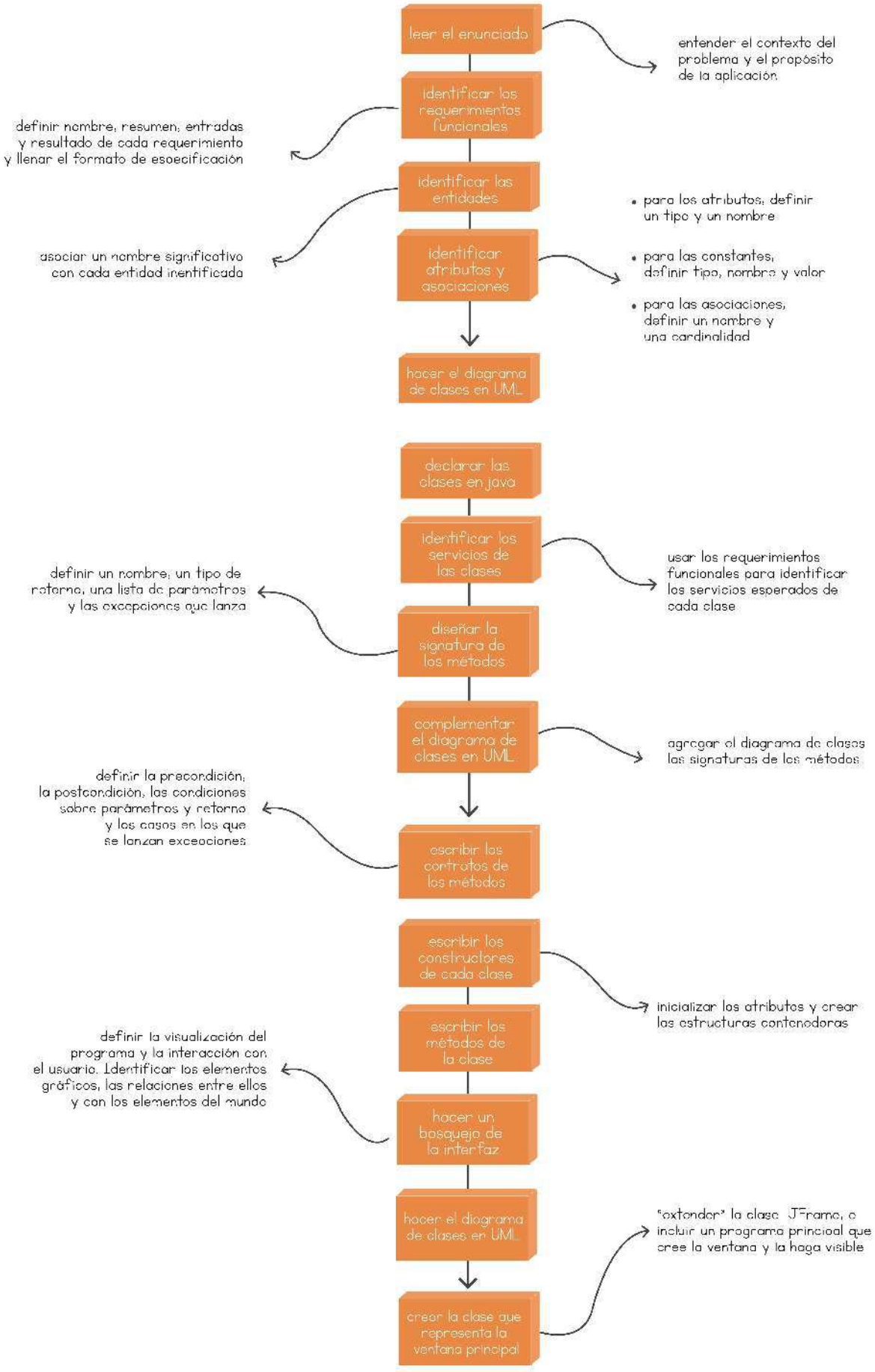
- Escribir el código en el lenguaje de programación (en nuestro caso Java), que implementa el diseño que definimos en la etapa anterior.

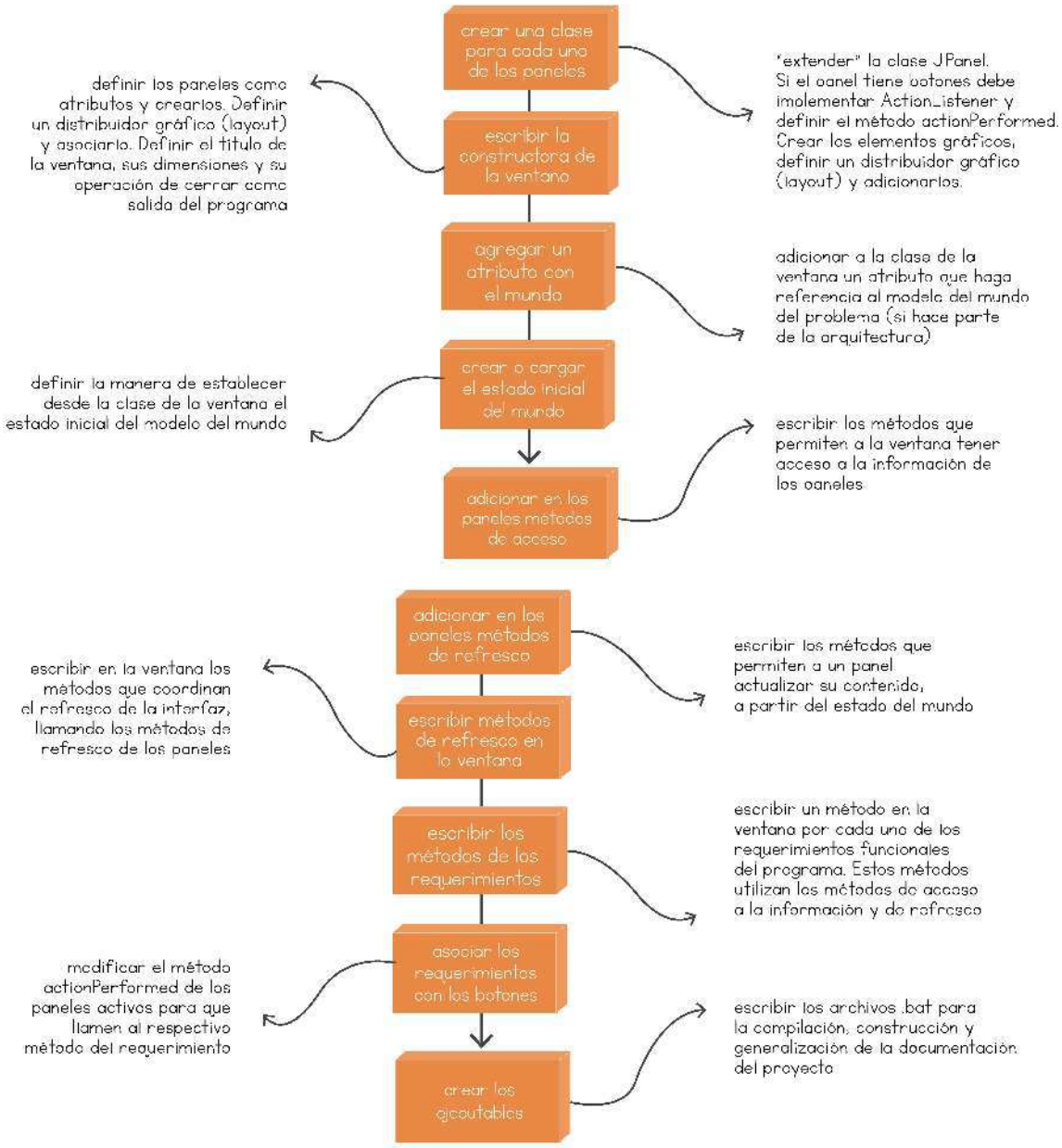
Resultados:

- El código de todas las clases, con sus contratos y comentarios.
- Para saber si hemos terminado nuestra tarea de construcción del programa, debemos probarlo. Además de las pruebas manuales que podemos realizar sobre él es importante contar con pruebas automáticas. Dichas pruebas son también clases Java que se encuentran definidas en el paquete de pruebas.

8.4. Una Visión Gráfica del Proceso

En esta parte resumimos gráficamente las principales tareas que constituyen el proceso de desarrollo de un programa. La idea es que a partir del enunciado del problema, el lector pueda seguirlo paso por paso. Todas estas tareas están enmarcadas dentro de las tres grandes etapas mencionadas anteriormente.


leer el enunciado
entender el contexto del problema y el propósito de la aplicación
identificar los requerimientos funcionales
definir nombre, resumen, entradas y resultado de cada requerimiento y llenar el formato de especificación
identificar las entidades
asociar un nombre significativo con cada entidad identificada
identificar atributos y asociaciones
para los atributos, definir un tipo y un nombre
para las constantes, definir tipo, nombre y valor
para las asociaciones, definir un nombre y una cardinalidad
hacer el diagrama de clases en UML
declarar las clases en java
identificar los servicios de las clases
usar los requerimientos funcionales para identificar los servicios esperados de cada clase
diseñar la signatura de los métodos
definir un nombre, un tipo de retorno, una lista de parámetros y las excepciones que lanza
complementar el diagrama de clases en UML
agregar al diagrama de clases las signaturas de los métodos
escribir los contratos de los métodos
definir la precondición, la postcondición, las condiciones sobre parámetros y retorno y los casos en los que se lanzan excepciones
escribir los constructores de cada clase
inicializar los atributos y crear las estructuras contenedoras
escribir los métodos de la clase
hacer un bosquejo de la interfaz
definir la visualización del programa y la interacción con el usuario. Identificar los elementos gráficos, las relaciones entre ellos y con los elementos del mundo
hacer el diagrama de clases en UML
crear la clase que representa la ventana principal
"extender" la clase JFrame, e incluir un programa principal que cree la ventana y la haga visible


crear una clase para cada uno de los paneles
"extender" la clase JPanel. Si el panel tiene botones debe implementar Action_listener y definir el método actionPerformed. Crear los elementos gráficos, definir un distribuidor gráfico (layout) y adicionarlos.
escribir la constructora de la ventana
definir los paneles como atributos y crearlos. Definir un distribuidor gráfico (layout) y asociarlo. Definir el título de la ventana, sus dimensiones y su operación de cerrar como salida del programa
agregar un atributo con el mundo
adicionar a la clase de la ventana un atributo que haga referencia al modelo del mundo del problema (si hace parte de la arquitectura)
crear o cargar el estado inicial del mundo
definir la manera de establecer desde la clase de la ventana el estado inicial del modelo del mundo
adicionar en los paneles métodos de acceso
escribir los métodos que permiten a la ventana tener acceso a la información de los paneles
adicionar en los paneles métodos de refresco
escribir los métodos que permiten a un panel actualizar su contenido, a partir del estado del mundo
escribir métodos de refresco en la ventana
escribir en la ventana los métodos que coordinan el refresco de la interfaz, llamando los métodos de refresco de los paneles
escribir los métodos de los requerimientos
escribir un método en la ventana por cada uno de los requerimientos funcionales del programa. Estos métodos utilizan los métodos de acceso a la información y de refresco
asociar los requerimientos con los botones
modificar el método actionPerformed de los paneles activos para que llamen al respectivo método del requerimiento
crear los ejecutables
escribir los archivos .bat para la compilación, construcción y generalización de la documentación del proyecto

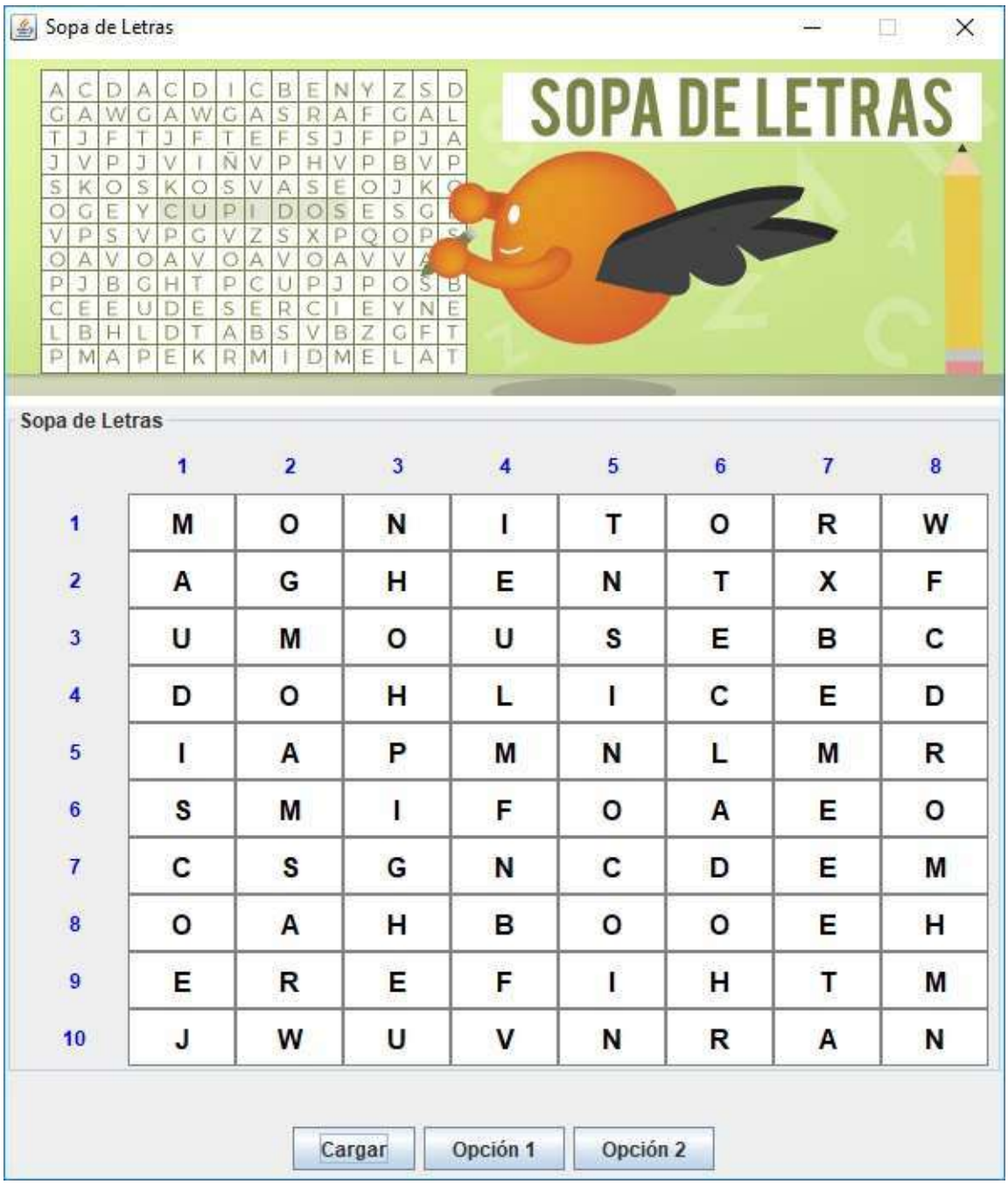
9. Hojas de Trabajo

9.1. Hoja de Trabajo Nº 1: Sopa de Letras

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 quiere de la aplicación y las restricciones para desarrollarla.

Se quiere construir un programa para el juego de la sopa de letras. En este juego hay un tablero que tiene una serie de letras organizadas en filas y columnas. Algunas de estas letras forman palabras que el jugador debe encontrar. Las palabras pueden estar dispuestas en modo horizontal, vertical o diagonal y pueden escribirse también en sentido contrario al normal (de derecha a izquierda o de abajo hacia arriba). Para cada sopa de letras hay una serie de palabras que deben buscarse. Cuando el jugador las encuentra todas, hay que avisarle que ganó el juego. La interfaz de usuario del programa es la que aparece en la siguiente figura. Las letras que forman parte de las palabras ya encontradas deben aparecer en otro color.



Tanto las dimensiones de la sopa de letras como las palabras que contiene se deben cargar desde un archivo de propiedades (seleccionado por el usuario durante la ejecución del programa), con las siguientes características:

- La propiedad sopaDeLetras.columnas define el número de columnas.
- La propiedad sopaDeLetras.filas define el número de filas.
- La propiedad sopaDeLetras.numPalabras define el número de palabras presentes en la sopa.
- La propiedad sopaDeLetras.palabra1 define la primera palabra que aparece en la sopa.
- La propiedad sopaDeLetras.fila1 define el contenido de la primera fila de la sopa.

El siguiente es un ejemplo de un posible archivo para describir la situación inicial del juego. En este tipo de archivos las líneas que comienzan por el símbolo # se interpretan como comentarios.

#letras

sopaDeLetras.columnas=8

sopaDeLetras.filas=10

sopaDeLetras.fila1=M O N I T O R W

sopaDeLetras.fila2=A G H E N T X F

sopaDeLetras.fila3=U M O U S E B C

sopaDeLetras.fila4=D O H L I C E D

sopaDeLetras.fila5=I A P M N L M R

sopaDeLetras.fila6=S M I F O A E O

sopaDeLetras.fila7=C S G N C D E M

sopaDeLetras.fila8=O A H B O O E H

sopaDeLetras.fila9=E R E F I H T M

sopaDeLetras.fila10=J W U V N R A N

#palabras

sopaDeLetras.numPalabras=7

sopaDeLetras.palabra1=MONITOR

sopaDeLetras.palabra2=MOUSE

sopaDeLetras.palabra3=DISCO

sopaDeLetras.palabra4=TECLADO

sopaDeLetras.palabra5=CDROM

sopaDeLetras.palabra6=MODEM

sopaDeLetras.palabra7=WEBCAM

Requerimientos funcionales. Describa los dos requerimientos funcionales de la aplicación.

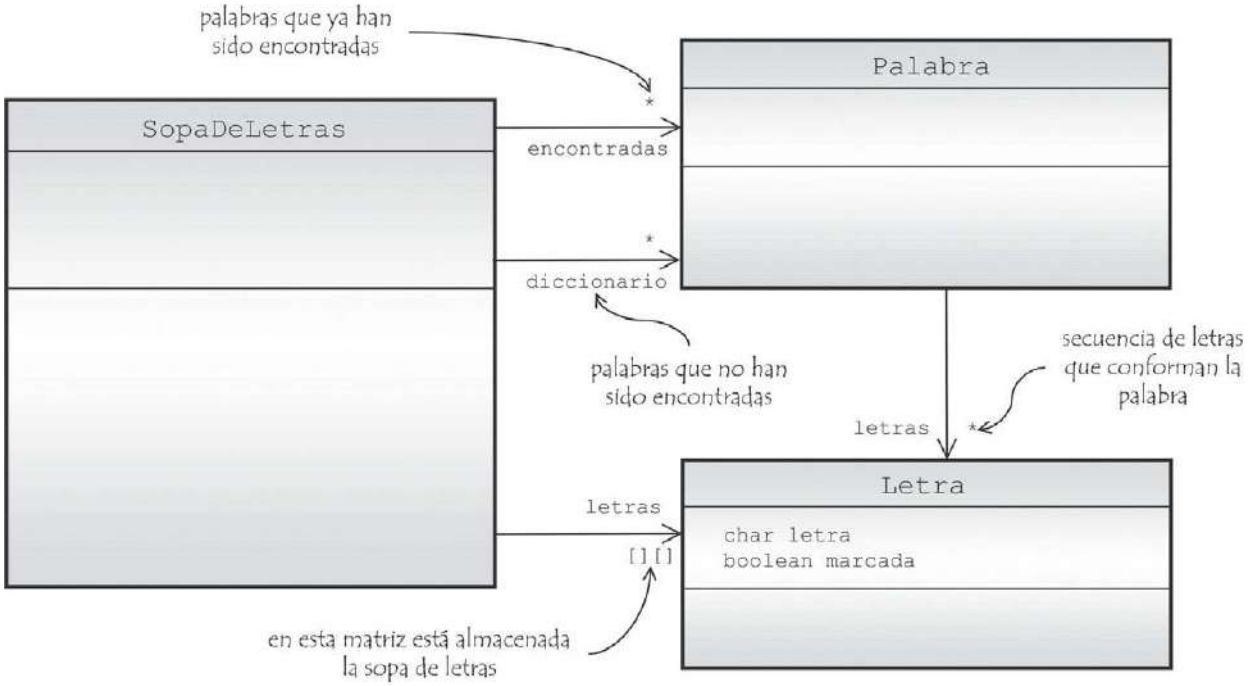
Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

Modelo del mundo. Estudie y complete el modelo con los atributos, constantes, asociaciones entre las clases y principales métodos.



Declaración de clases. Para las siguientes clases, escriba en Java la declaración de sus atributos y sus asociaciones.

```
public class SopaDeLetras
{



}
```

```
public class Palabra
{



}
```

```
public class Letra
{



}
```

Inicialización de matrices. Escriba el constructor de la clase SopaDeLetras, que carga la información de un archivo de propiedades, cuya representación abstracta se entrega como parámetro. Si hay problemas en el proceso, lanza una excepción.

```
public SopaDeLetras( File pArchivoSopa ) throws Exception
{









}
```

Desarrollo de métodos. Desarrolle los siguientes métodos de la clase SopaDeLetras, identificando el patrón de algoritmo al que corresponde cada uno.

Metodo 1

Retornar el número de palabras que ya se han encontrado en la sopa de letras.

```
public int darPalabrasEncontradas( )
{




}
```

Metodo 2

Contar el número de vocales que hay en la sopa de letras.

```
public int totalVocales( )
{




}
```

Metodo 3

Retornar la cadena con los caracteres que se encuentran entre dos columnas (pColumna1 y pColumna2) de una misma fila (pFila). Puede suponer que los valores que se entregan como parámetros son todos válidos. Puede suponer que pColumna2 es mayor que pColumna1.

```
public String darPalabraEnFila( int pFila, int pColumna1, int pColumna2 )
{




}
```

Metodo 4

Retornar la cadena con los caracteres que se encuentran entre dos filas (pFila1 y pFila2) de una misma columna (pColumna). Puede suponer que los valores que se entregan como parámetros son todos válidos. Puede suponer que pFila2 es mayor que pFila1.

```
public String darPalabraEnColumna( int pColumna, int pFila1, int pFila2 )
{









}
```

Metodo 5

Retornar la cadena con los caracteres que se encuentran en diagonal entre dos filas (pFila1 y pFila2). La diagonal comienza en la columna que se recibe como parámetro y desciende de izquierda a derecha. Puede suponer que los valores que se entregan como parámetros son todos válidos. Puede suponer que pFila2 es mayor que pFila1.

```
public String darPalabraEnDiagonal(int pColumna, int pFila1, int pFila2 )
{









}
```

Metodo 6

Retornar una cadena de caracteres formada con todas las letras que no forman parte de las palabras encontradas. Las letras se deben agregar a la respuesta de izquierda a derecha, de arriba abajo.

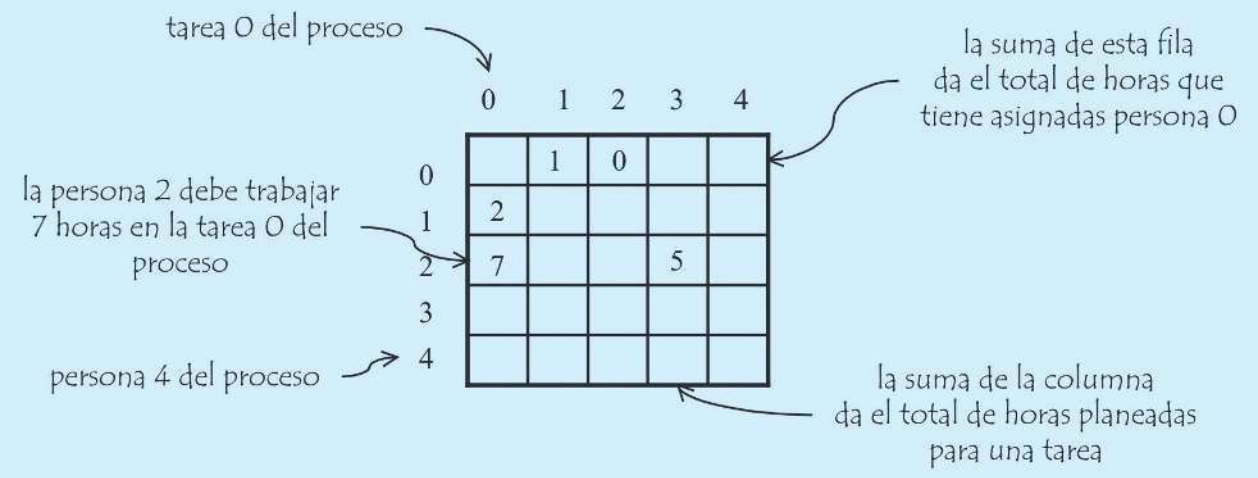
```
public String darMensajeSecreto( )
{









}
```

9.2 Hoja de Trabajo Nº 2: Asignación de Tareas

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.

En todo proceso es importante la asignación de tareas, actividad en la cual se definen los recursos (en particular personas) que necesita cada tarea para poderse llevar a cabo. Se quiere construir una aplicación que permita manejar la asignación de tareas para organizar una fiesta, de forma similar a una planilla (en las columnas están las tareas que se deben realizar y en las filas las personas disponibles para hacerlo). Las tareas y las personas ya están definidas desde el comienzo del programa (se cargan de un archivo de propiedades). En cada casilla de la planilla va el número de horas que dicha persona debe dedicarle a la respectiva tarea, como se muestra en la siguiente figura:



La aplicación debe permitir que se asigne un determinado número de horas de trabajo de una tarea a una persona. Si a una persona ya se le ha asignado un número de horas en una tarea, es posible reasignar (cambiar) ese tiempo. Además, a partir de esta asignación, se quieren realizar algunos cálculos:

Para cada tarea es importante saber:

- El número de personas asignadas (las que tienen más de 0 horas asignadas para la tarea
- El total de horas asignadas.
- La persona con más horas asignadas a la tarea.
- El promedio de horas por persona.
- El porcentaje de trabajo que representa una tarea respecto del total de tareas.

Para cada persona es importante saber:

- El número de tareas asignadas (aquellas para las que la persona tiene más de 0 horas asignadas).
- El total de horas asignadas.
- La tarea para la que tiene el mayor número de horas asignadas.
- Si es la persona con el mayor número de horas asignadas.
- El promedio de horas por tarea.

La interfaz de usuario del programa de asignación de tareas es la que aparece en la siguiente figura:

La información de tareas y personas de la aplicación está consignada en el archivo de propiedades llamado data/datosPlanilla.dat. Un ejemplo de dicho archivo es el siguiente:

#tareas

tareas.numero=6

tareas.tarea1.nombre=Inflar globos

tareas.tarea2.nombre=Preparar la torta

tareas.tarea3.nombre=Repartir las invitaciones

tareas.tarea4.nombre=Hacer el playlist de música

tareas.tarea5.nombre=Decorar la sala

tareas.tarea6.nombre=Instalar equipo de sonido

#personas

personas.numero=4

personas.persona1.nombre=Pedro

personas.persona2.nombre=Juan

personas.persona3.nombre=Carolina

personas.persona4.nombre=Andrés

En la propiedad tareas.numero se indica el número de tareas que manejará la aplicación. Luego, para nombrar las tareas, deben aparecer tantas propiedades como este número indica. Estas propiedades son de la forma *tareas.tarea<contador>.nombre*, donde el contador es un número que va desde uno hasta el número de tareas indicado.

En la propiedad personas.numero se indica el número de personas que manejará la aplicación. Luego, para nombrar a las personas, deben aparecer tantas propiedades como este número indica. Estas propiedades son de la forma *personas.persona<contador>*.nombre, donde el contador es un número que va desde uno hasta el número de personas indicado.

Requerimientos funcionales. Describa algunos de los más importantes requerimientos funcionales.

Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

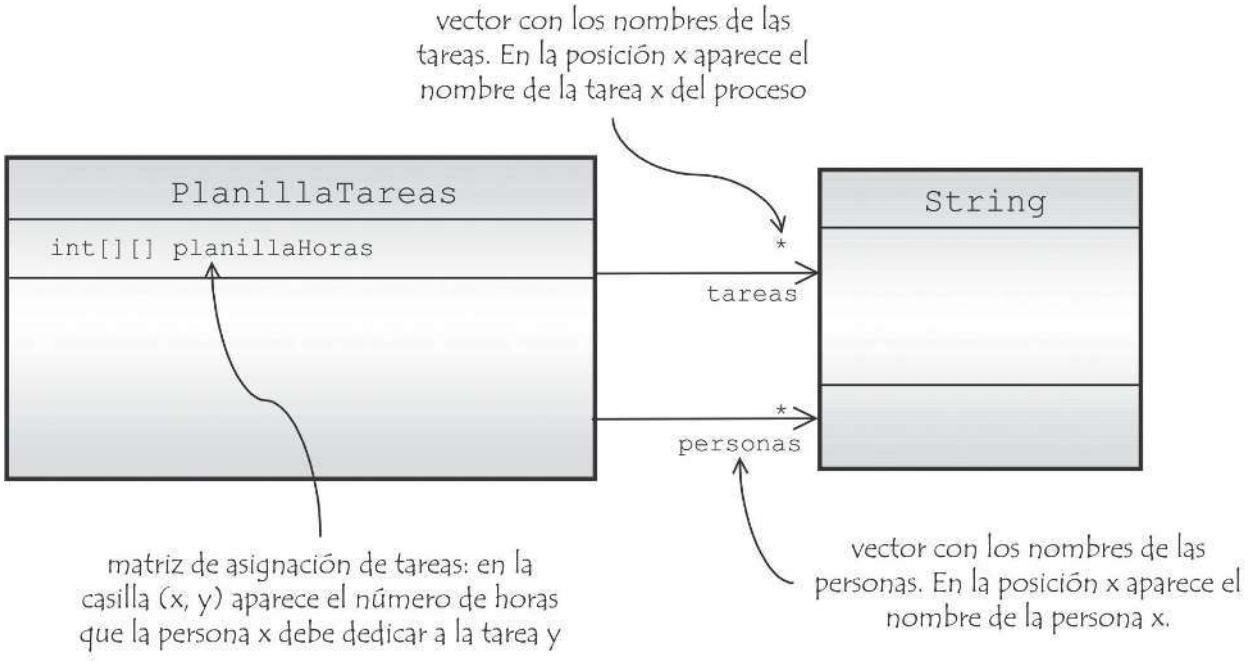
Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

Requerimiento Funcional 4

Nombre	
Resumen	
Entradas	
Resultado	

Modelo el mundo. Estudie y complete el modelo con los atributos, constantes, asociaciones entre las clases y principales métodos.



Declaración de las clases. Para la siguiente clase escriba en Java la declaración de sus atributos y sus asociaciones.

```
public class PlanillaTareas
{









}
```

Inicialización de matrices. Escriba el constructor de la clase PlanillaTareas, que carga la información de un archivo de propiedades, cuyo nombre completo se entrega como parámetro, y arma la matriz que representa la planilla. Si hay problemas en el proceso, lanza una excepción.

```
public PlanillaTareas( String pArchivo ) throws Exception
{

















}
```

Desarrollo de métodos. Desarrolle los siguientes métodos de la clase PlanillaTareas, identificando el patrón de algoritmo al que corresponde cada uno.

Metodo 1

Contar el total de tareas que no tienen ninguna asignada.

```
public int totalTareasSinAsignar( )
{









}
```

Metodo 2

Contar el número de tareas en las que participa la persona cuyo nombre se da como parámetro.

```
public int totalTareasParticipa( String pNombre )
{









}
```

Metodo 3

Decir si existe al menos una tarea en la que participen todas las personas.

```
public boolean existeTareaTodosParticipan( )
{









}
```

Metodo 4

Retornar el nombre de la persona que más tiempo tiene asignado en la tarea que se da como parámetro.

```
public String personaMasParticipa( String pTarea )
{









}
```

Metodo 5

Retornar el nombre de la tarea en la que más tiempo tiene asignado la persona cuyo nombre se da como parámetro.

```
public String tareaMasParticipa( String pNombre )
{









}
```

Metodo 6

Retornar la suma de horas asignadas que tienen las personas que se encuentran en un rango de filas descrito por los índices recibidos como parámetros.

```
public int sumarHorasPersonasEntre( int pIndiceInicial, int pIndiceFinal )
{









}
```

Metodo 7

Calcular el promedio de horas asignadas a todas las personas.

```
public double darPromedioTiempoAsignadoPersonas( )
{









}
```

ANEXOS

A. El Lenguaje Java

1. Instalación de las Herramientas

1.1. ¿Qué se Necesita para Empezar?

Hay dos herramientas básicas que el lector debe instalar en su computador, antes de empezar a crear el ambiente de desarrollo necesario para construir programas. Estas herramientas son:

1. Un navegador de Internet.
2. Un programa que permita extraer el contenido de un archivo con formato zip.

> Antes de continuar, asegúrese de que cuenta en su computador con un navegador de Internet y con un programa para extraer el contenido de un archivo con formato zip.

1.2. ¿Dónde Encontrar los Instaladores de las Herramientas?

Para crear el ambiente de desarrollo se necesitan algunas herramientas, las cuales se pueden obtener en:

El sitio web del proyecto CUPI2:

En la dirección http://cupi2.uniandes.edu.co.

El sitio web de los fabricantes de los programas:

En las siguientes direcciones de Internet puede encontrar las últimas versiones de los instaladores:

- Lenguaje Java: http://java.sun.com/
- Ambiente de desarrollo Eclipse: http://www.eclipse.org/

En el primer enlace busque el instalador de la herramienta llamada "Java 2 Platform, Standard Edition (J2SE)".

> Verifique que ha localizado el instalador de Java. Este viene en un archivo .exe que permite hacer la instalación de la máquina virtual y del compilador (jdk-1_5_0-rc-windows-i586.exe) y en un archivo .zip que trae la documentación (jdk-1_5_0-rc-doc.zip). Los nombres exactos de dichos archivos pueden variar dependiendo de las versiones.
>
> Verifique que ha localizado el instalador de Eclipse. Este instalador viene en un archivo .zip (eclipse-SDK-3.0-win32.zip). El nombre exacto de dicho archivo puede variar, dependiendo de la versión que vaya a instalar.

1.3. ¿Cómo Instalar las Herramientas?

Java 2 Standard Edition (J2SE)

- Ejecute el instalador y responda a las preguntas que éste hace durante el proceso. En particular, debe escoger un directorio en el disco duro para instalar las herramientas del lenguaje.
- Extraiga el contenido del archivo .zip que trae la documentación de Java, utilizando la herramienta que tenga disponible para tal fin.
- Modifique la variable de ambiente del sistema operativo llamada PATH, para que incluya el subdirectorio bin del directorio en el cual quedaron instaladas las herramientas del lenguaje.

Eclipse SDK

- Extraiga el contenido del archivo .zip en el directorio en el que quiera que quede instalado el ambiente de desarrollo Eclipse.

Busque en el directorio en el que instaló el ambiente Eclipse un archivo llamado eclipse.exe. Ejecútelo para iniciar dicha aplicación.

Abra una ventana de comandos del sistema operativo. Ejecute el comando

```
java –version
```

La máquina virtual de Java debe contestar algo parecido al siguiente mensaje:

```
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-b64) Java HotSpot(TM) Cl
ient VM  (build 1.5.0-b64, mixed mode)
```

Abra una ventana de comandos del sistema operativo. Ejecute el comando

```
javac
```

El compilador del lenguaje Java debe contestar algo parecido al siguiente mensaje:

```
javac: no source files
Usage: javac <options> <source files>
```

Abra una ventana de comandos del sistema operativo. Ejecute el comando

```
javac -version
```

El compilador del lenguaje Java debe contestar algo parecido al siguiente mensaje:

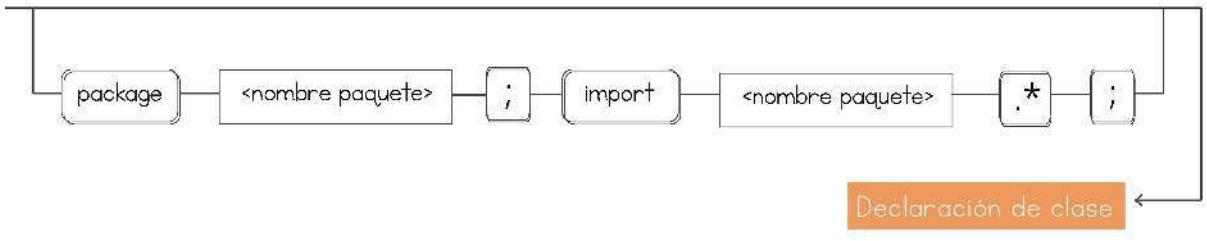
```
javac 1.5.0-rc
```

Si las tres acciones anteriores funcionan correctamente, quiere decir que tanto Java como el ambiente de desarrollo Eclipse quedaron instalados correctamente en su computador. Si tiene algún problema en el proceso de instalación, le recomendamos buscar en el sitio web del proyecto los tutoriales respectivos.

2. Diagramas de Sintaxis del Lenguaje Java

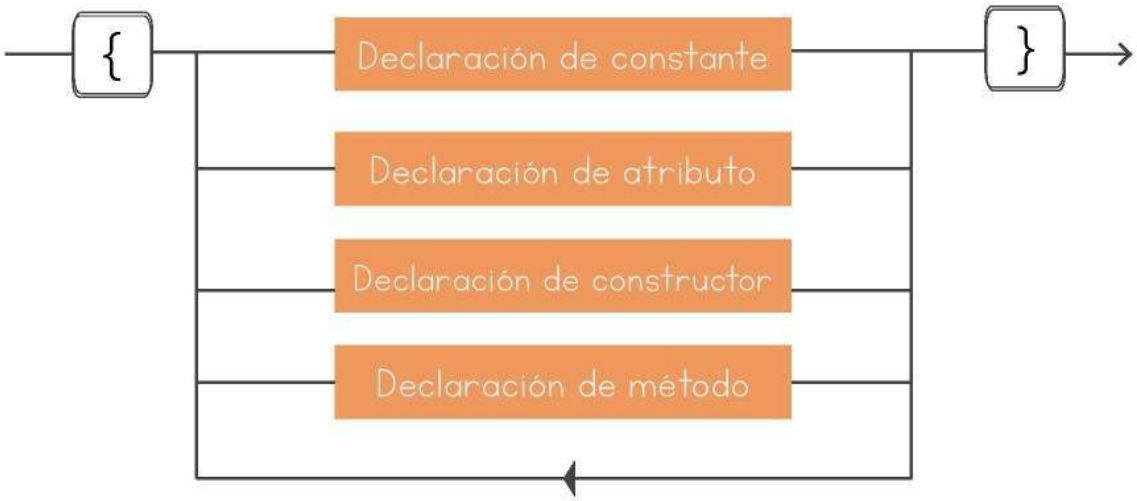
La sintaxis resumida en este anexo corresponde únicamente al subconjunto del lenguaje Java estudiado en este libro, junto con ciertas buenas prácticas de programación. En algunos casos se hicieron algunas simplificaciones en la sintaxis, de manera que más que una especificación formal del lenguaje debe tomarse como una guía informal de uso.

Unidad de compilación

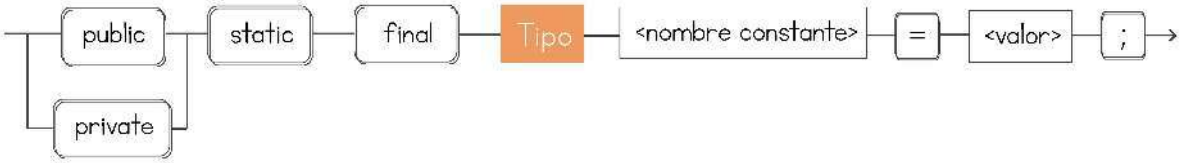


Declaración de clase

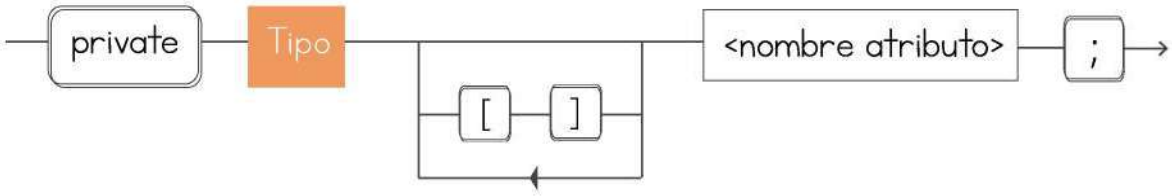
Cuerpo de clase



Declaración de constante

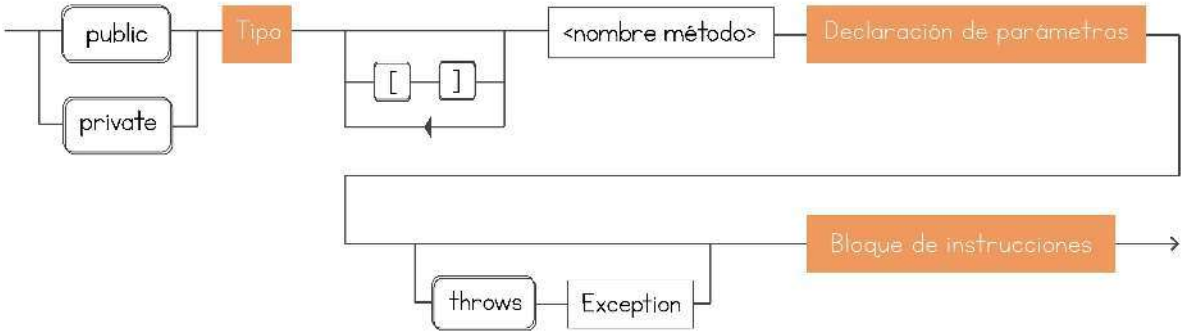


Declaración de atributo



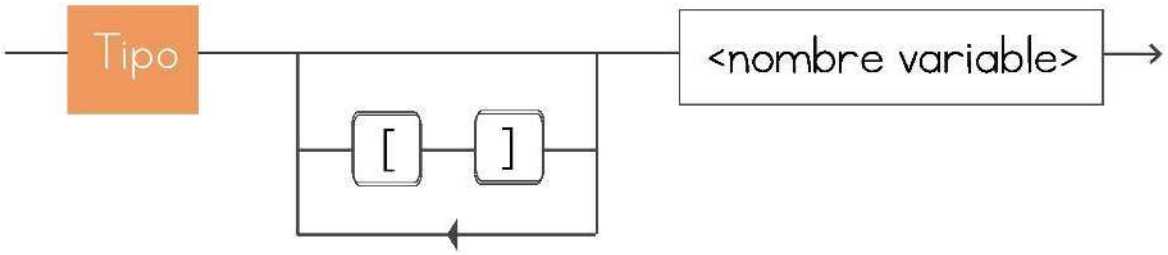
Declaración de constructor

Declaración de método

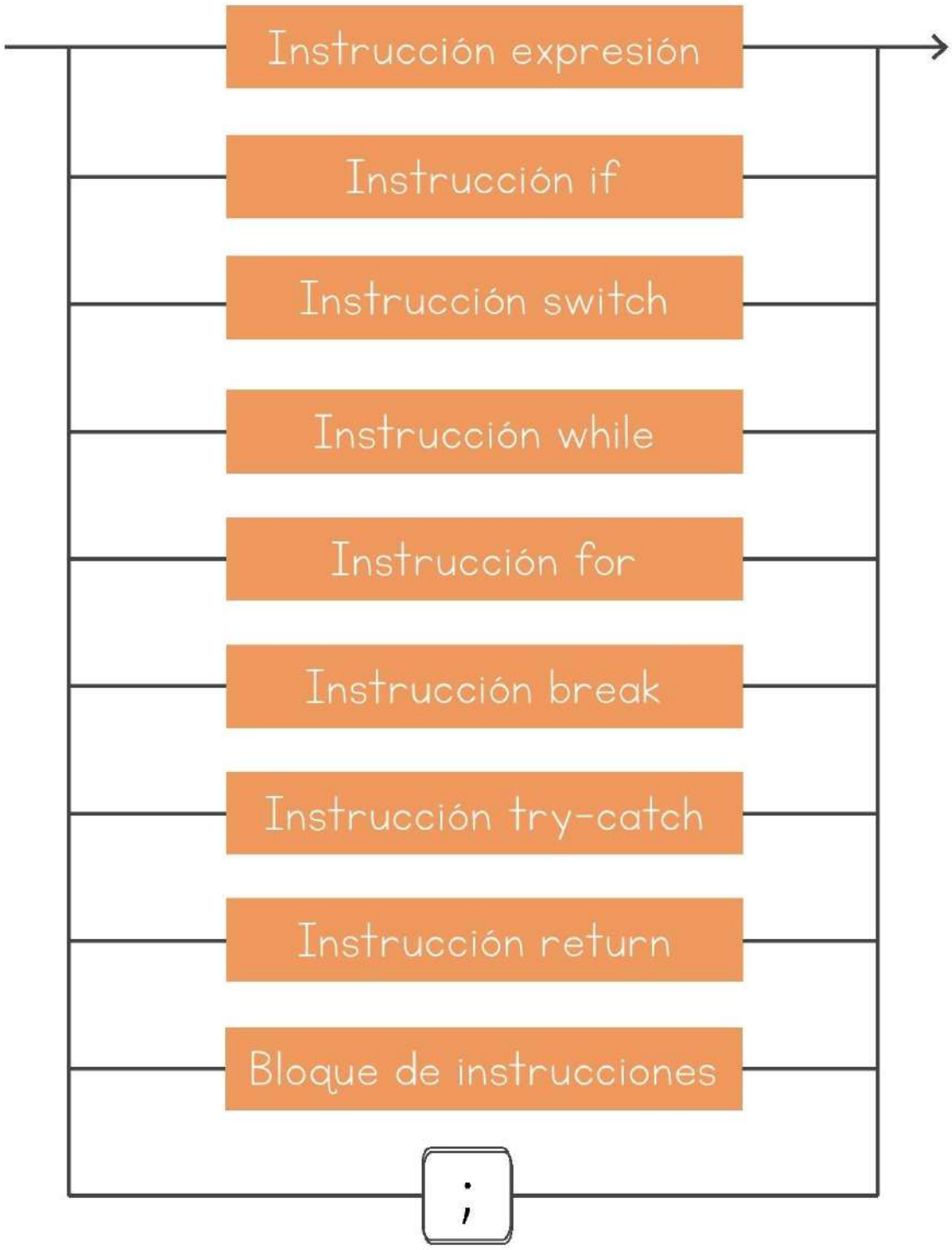


Declaración de parámetros

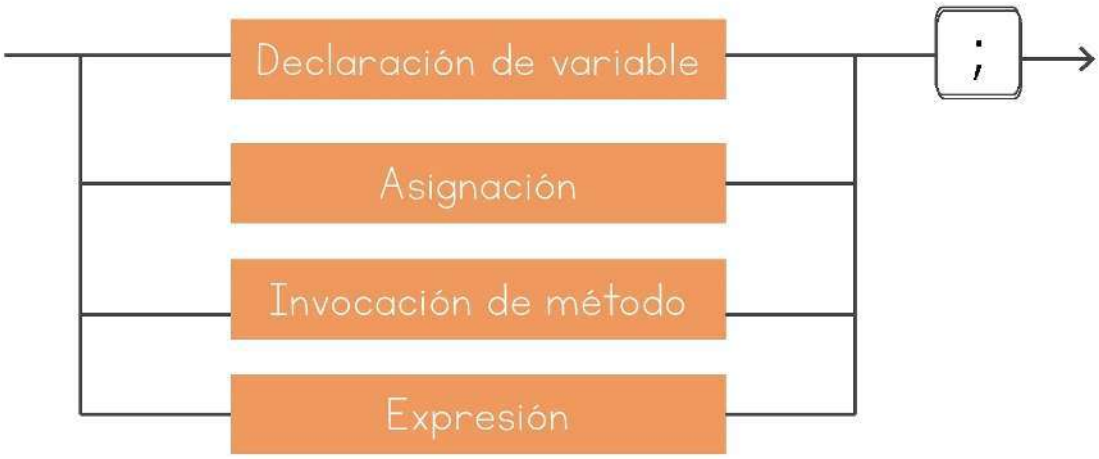
Declaración de variable



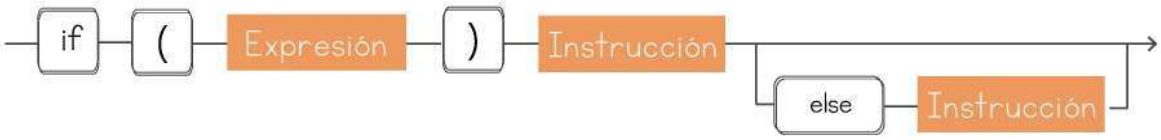
Instrucción



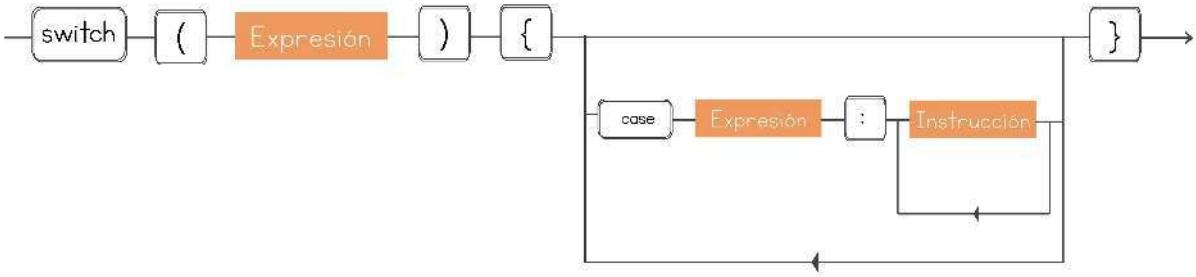
Instrucción Expresión



Instrucción if



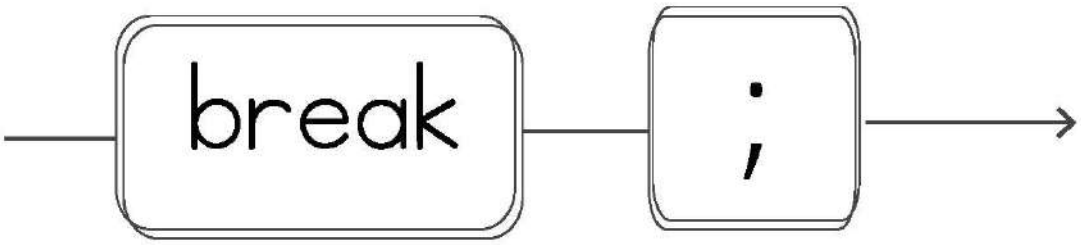
Instrucción switch



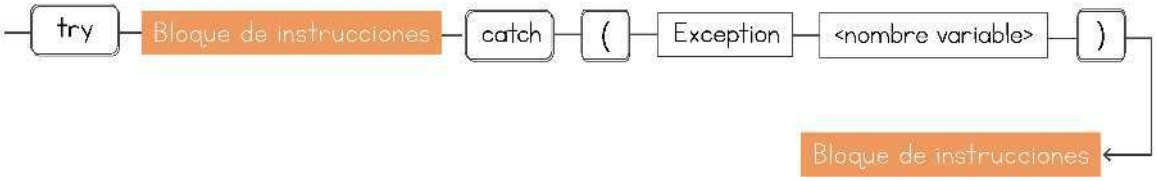
Instrucción while:

Instrucción for:

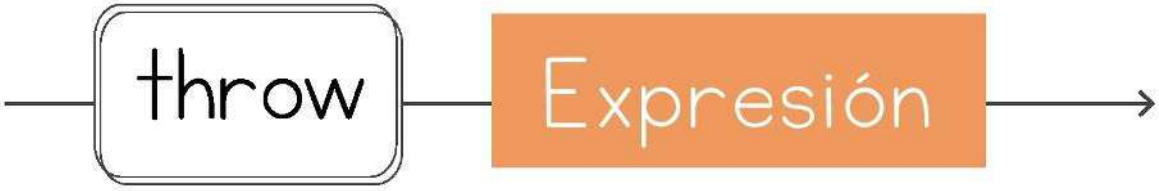
Instrucción break



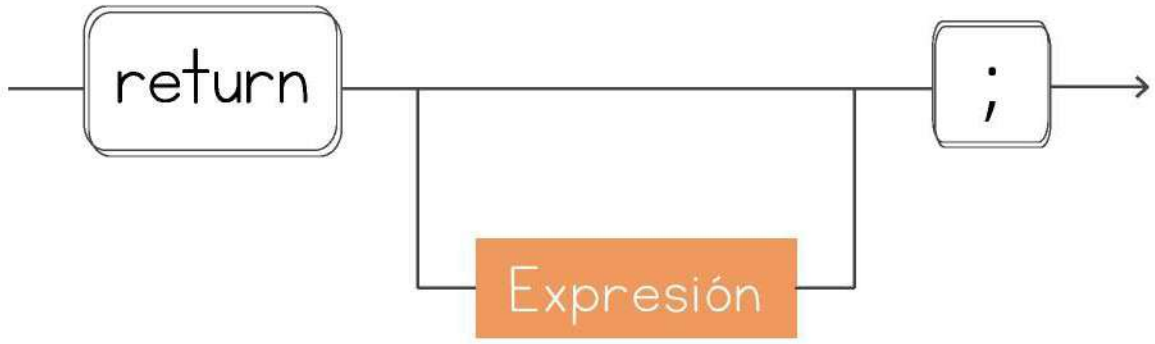
Instrucción try-catch



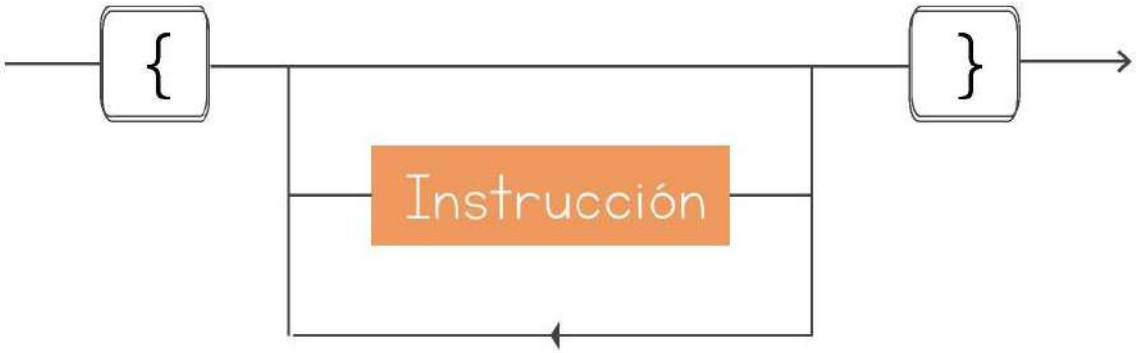
Instrucción throw



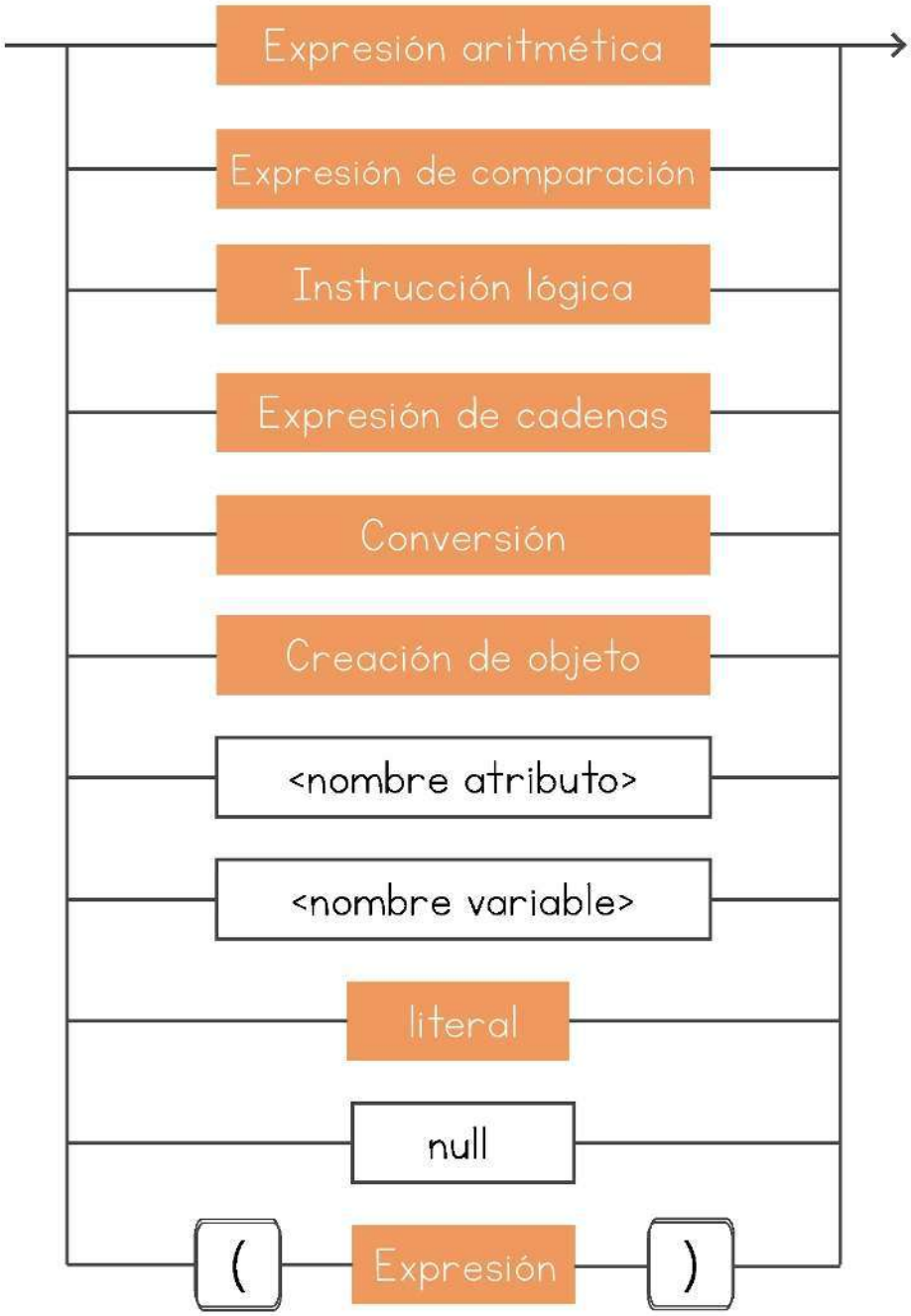
Instrucción return



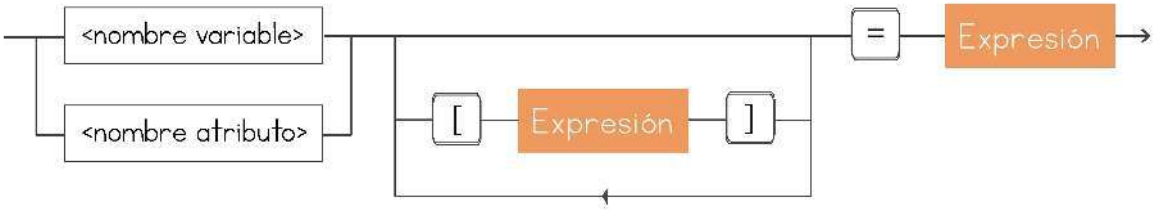
Bloque de instrucciones



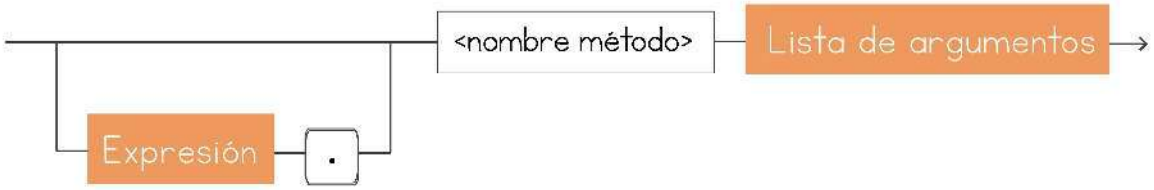
Expresión


Expresión aritmética
Expresión de comparación
Instrucción lógica
Expresión de cadenas
Conversión
Creación de objeto
<nombre atributo>
<nombre variable>
literal
null
(
Expresión
)

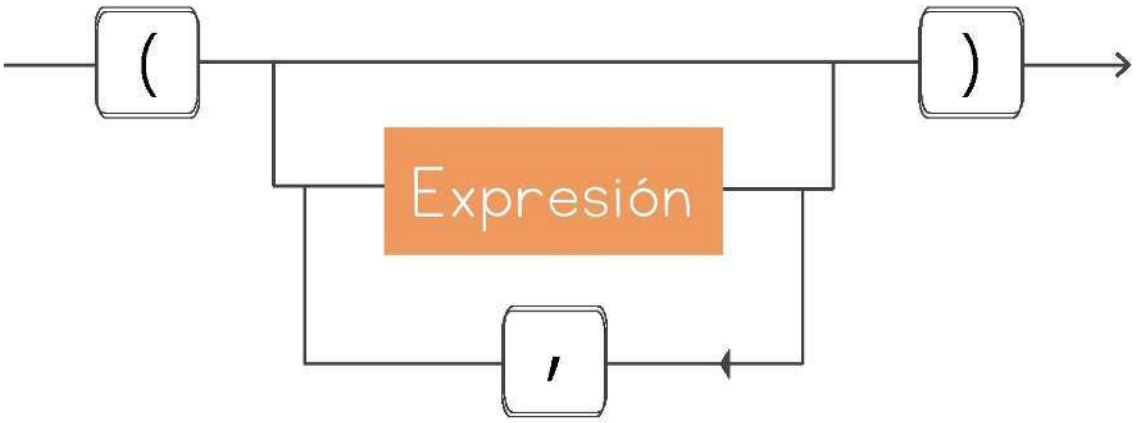
Asignación



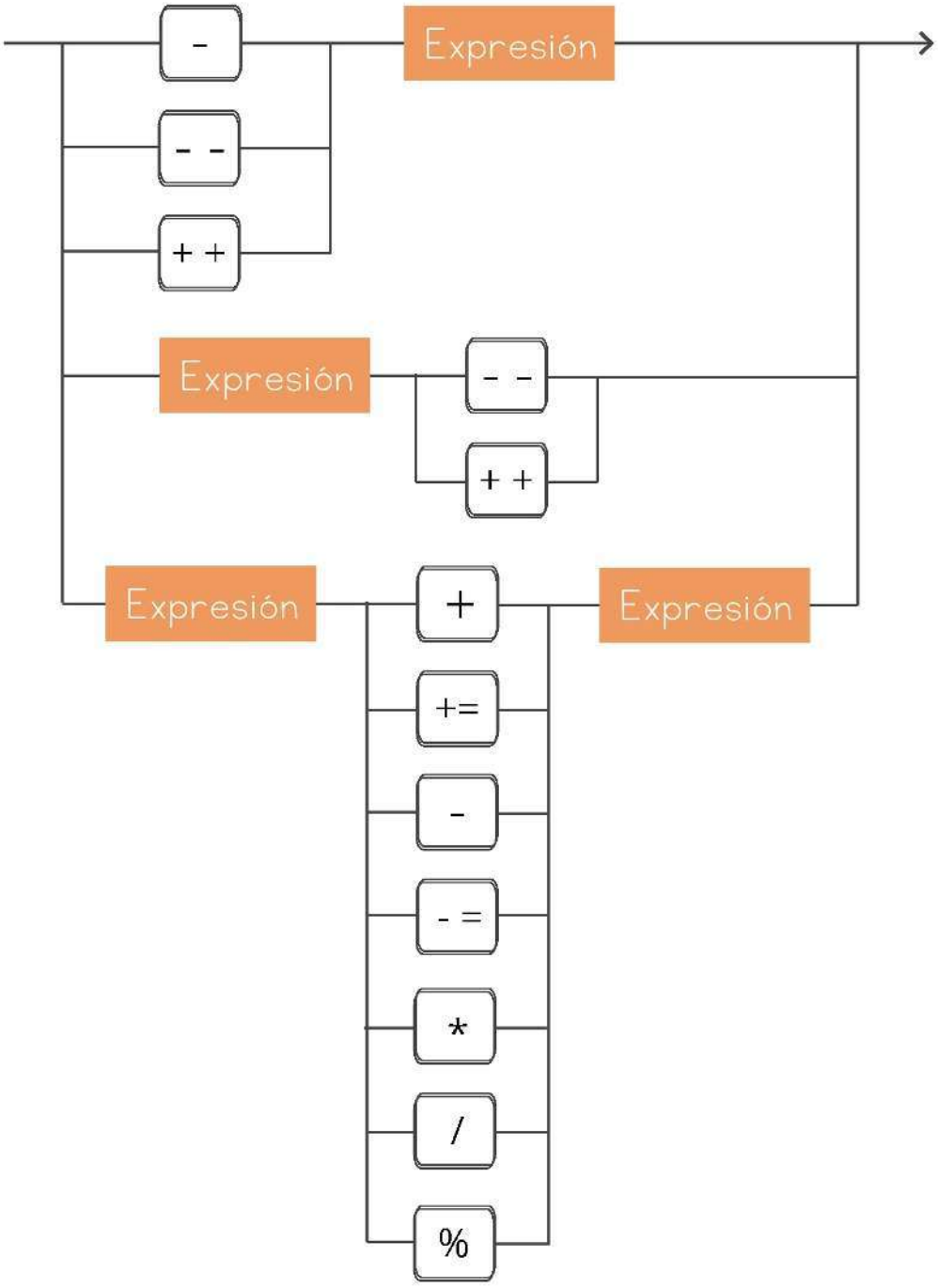
Invocación de método



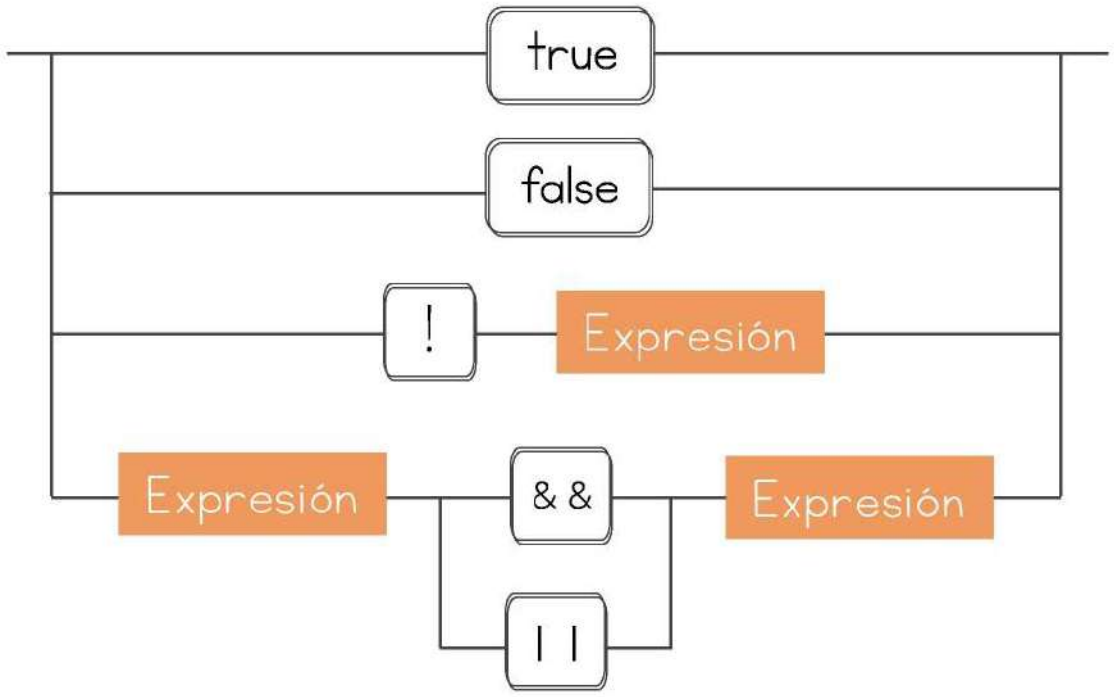
Lista de argumentos



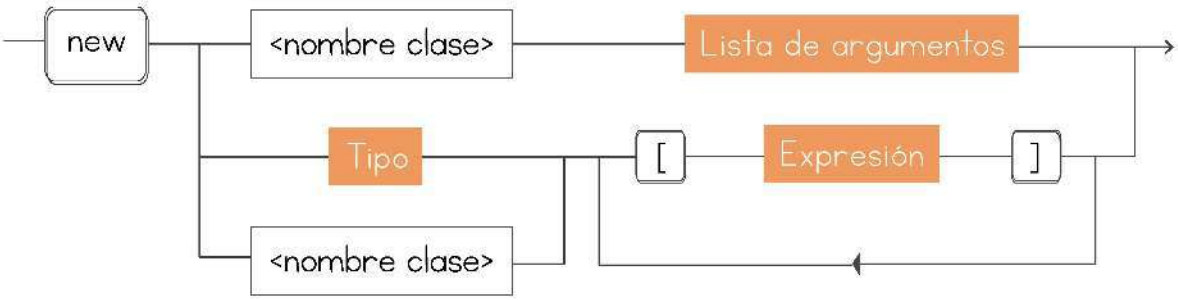
Expresión aritmética



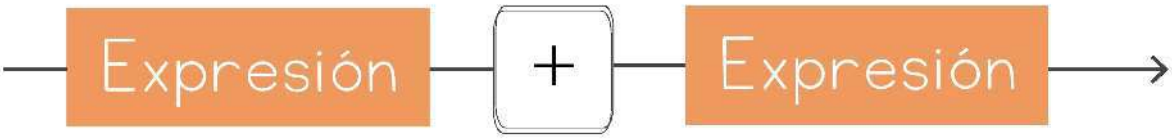
Expresión lógica



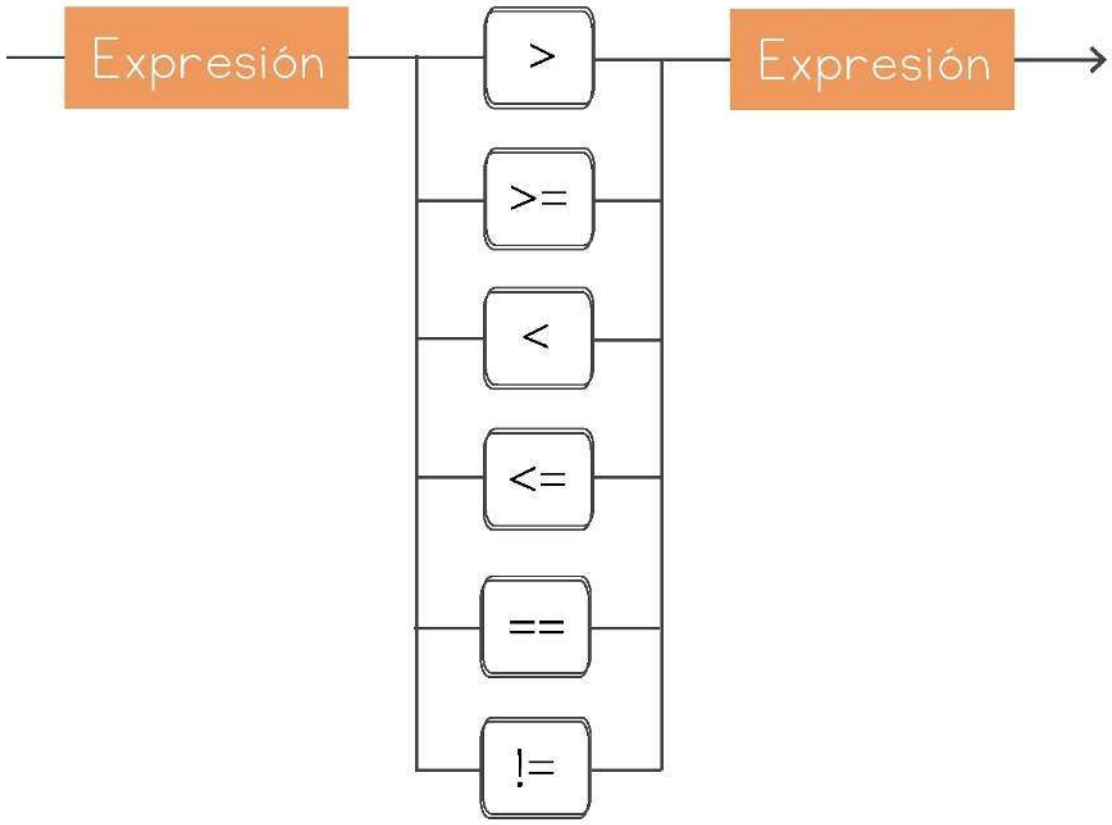
Creación de objeto



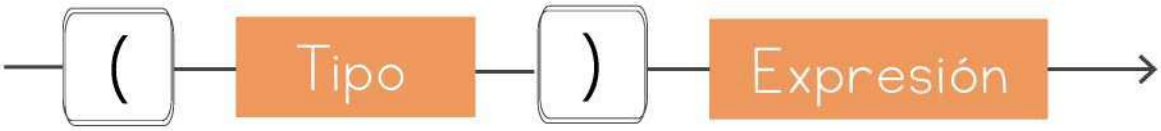
Expresión de cadenas



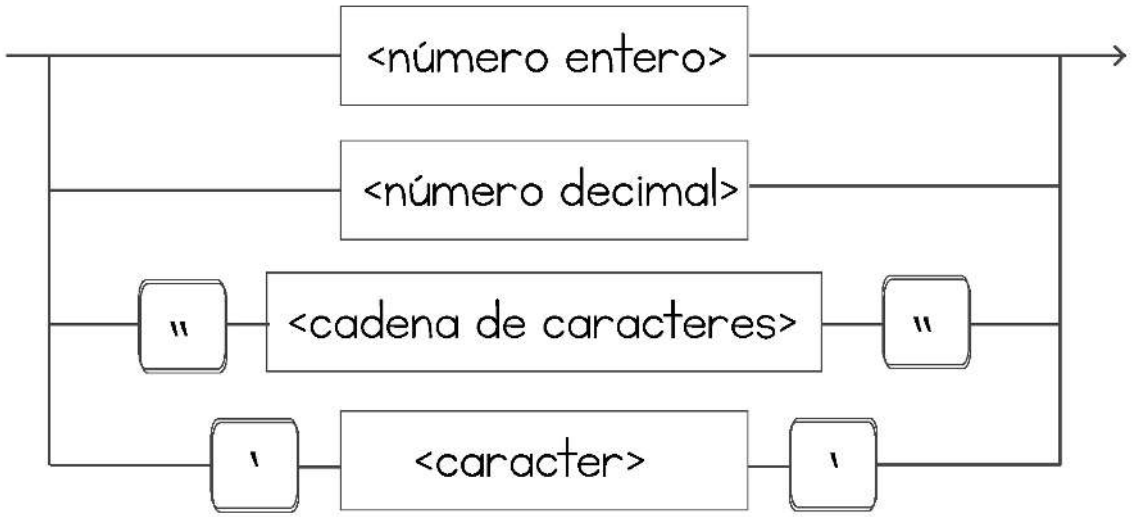
Expresión de comparación



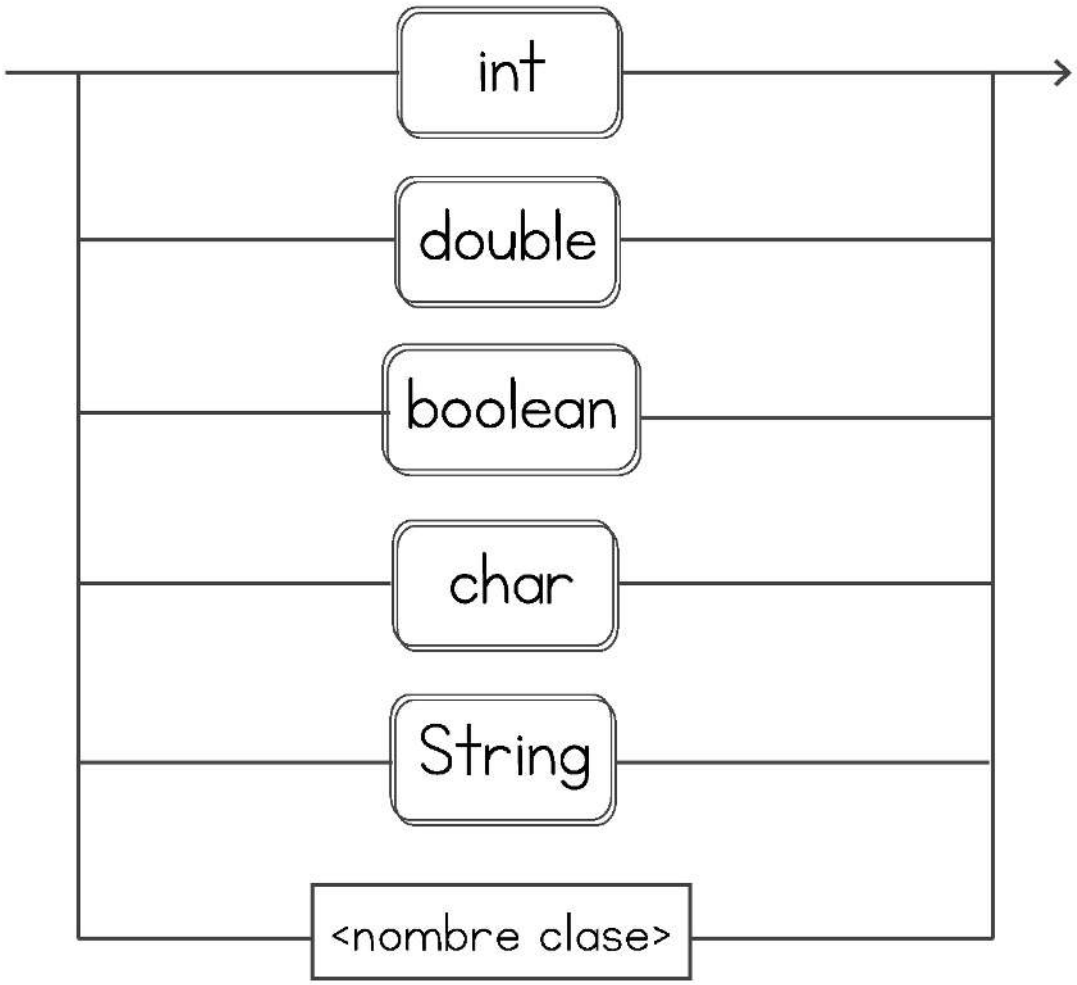
Conversión



Literal


<número entero>
<número decimal>
"
<cadena de caracteres>
"
'
<caracter>
'

Tipo



C. Resumen de Comandos de Windows

1. Comandos Ejecutables de Windows

A continuación encontrará un subconjunto de los comandos de Windows que se pueden ejecutar en la consola o intérprete de comandos del sistema operativo. Varios de estos comandos son utilizados en los archivos ejecutables (archivos.bat) de los ejemplos que se desarrollan a lo largo de este libro.

Para obtener la lista completa de los comandos válidos utilice el comando help y, para obtener mayor información de un comando en particular, utilice

```
help  <comando>
```

Comando:

```
CD o CHDIR
```

Muestra el nombre del directorio actual o permite cambiar de directorio.

CD

Muestra el nombre del directorio actual.

CD

Cambia el directorio actual.

Comando:

```
 CLS
```

Limpia el contenido de la pantalla.

CLS

Comando:

```
CMD
```

Inicia una nueva ventana del intérprete de comandos.

CMD

Inicia un nuevo intérprete.

CMD /C

Inicia un nuevo intérprete, ejecuta el comando y termina.

CMD /K

Inicia un nuevo intérprete, ejecuta el comando y permanece activo.

Comando:

```
COPY
```

Copia un archivo a un directorio de destino.

COPY

Copia el archivo origen en el destino. puede ser el nombre de un directorio o de un archivo.

Comando:

```
DATE
```

Muestra o cambia la fecha del sistema.

DATE /T

Muestra la fecha del sistema.

DATE

Muestra la fecha del sistema y permite cambiarla.

Comando:

```
DEL o ERASE
```

Borra uno o más archivos.

DEL

Borra cada uno de los archivos especificados en la lista de nombres.

puede incluir nombres de directorios y comodines para borrar varios archivos.

Comando:

```
DIR
```

Muestra el contenido (archivos y subdirectorios) de un directorio.

DIR

Muestra el contenido del directorio actual.

DIR

Muestra el contenido del directorio indicado.

Comando:

```
ECHO
```

Muestra un mensaje y permite activar y desactivar la salida del mismo comando ECHO.

ECHO ON

Activa la salida de mensajes del comando.

ECHO OFF

Desactiva la salida de mensajes del comando.

ECHO

Muestra el mensaje en la consola.

Comando:

```
EXIT
```

Termina el intérprete de comandos, o un programa de comandos (archivo .bat).

EXIT

Sale de la ventana del intérprete de comandos.

EXIT /B

Sale de un programa de comandos (archivo .bat) sin salir de la ventana del intérprete.

Comando:

```
FIND
```

Busca una cadena de texto en uno o más archivos del sistema.

FIND ""

Busca la cadena dada en los archivos especificados por

. puede contener comodines para especificar más fácilmente los archivos y directorios en los que se quiere hacer la búsqueda.

Comando:

```
HELP
```

Brinda la información de ayuda para los comandos de Windows.

HELP

Lista todos los comandos junto con una descripción abreviada.

HELP

Muestra la ayuda detallada de un comando en particular.

Comando:

```
MD o MKDIR
```

Crea un directorio o una ruta de directorios.

MD

Crea el directorio o la ruta de directorios indicada en . Si para ello hace falta crear directorios intermedios, este comando se encargará de ello.

Comando:

```
MORE
```

Muestra por partes en la pantalla el contenido de un archivo o la salida de un comando.

MORE

Muestra los archivos incluidos en la lista haciendo una pausa cada vez que se llena la pantalla.

comando | MORE

Muestra la salida del comando haciendo una pausa cada vez que se llena la pantalla.

Comando:

```
MOVE
```

Mueve archivos y cambia el nombre de archivos y directorios.

MOVE

Cambia de nombre el archivo o el directorio.

MOVE

Mueve el archivo al destino indicado.

Comando:

```
PATH
```

Muestra o establece la ruta de búsqueda de los archivos ejecutables.

PATH

Muestra la ruta de búsqueda de los archivos ejecutables.

PATH

Establece las rutas de búsqueda. Diferentes rutas pueden separarse con el carácter ‘;’. Puede utilizar la variable %PATH% para agregar las nuevas rutas a las establecidas con anterioridad.

PATH ;

Borra todas las rutas de búsqueda establecidas.

Comando:

```
PAUSE
```

Suspende la ejecución de un programa de comandos y espera que el usuario oprima una tecla para continuar.

PAUSE

Suspende el proceso actual del programa y presenta el mensaje "Presione una tecla para continuar...".

Comando:

```
PROMPT
```

Cambia el símbolo del sistema que se muestra en el intérprete de comandos.

PROMPT

Cambia el símbolo del sistema al texto indicado. Existen códigos para incluir caracteres especiales.

Comando:

```
RD o RMDIR
```

Elimina un directorio.

RD

Elimina el directorio si está vacío.

RD /S

Elimina el árbol de directorios cuya raíz es .

RD /S /Q

Elimina el árbol de directorios cuya raíz es sin pedir confirmación.

Comando:

```
REM
```

Inicia un comentario en los archivos de programas de comandos (archivos .bat).

REM

Introduce el comentario indicado.

Comando:

```
REN o RENAME
```

Cambia el nombre de un archivo.

REN

Cambia el nombre del archivo.

Comando:

```
SET
```

Muestra, cambia o elimina las variables de entorno del intérprete de comandos.

SET

Lista todas las variables del entorno y los valores que tienen asignados.

SET

Muestra el valor asignado a .

SET =

Establece la cadena dada como valor de la variable indicada.

Comando:

```
START
```

Inicia una nueva ventana del intérprete de comandos.

START

Abre una nueva ventana sin ejecutar ningún programa o comando.

START

Abre una nueva ventana y ejecuta el comando indicado.

START

Abre una nueva ventana y ejecuta el archivo ejecutable indicado.

Comando:

```
TIME
```

Muestra o cambia la hora del sistema.

TIME /T

Muestra la hora del sistema.

TIME

Muestra la hora del sistema y permite cambiarla.

Comando:

```
TITLE
```

Establece el título de la ventana del intérprete de comandos.

TITLE

Cambia el título de la ventana al indicado.

Comando:

```
TYPE
```

Muestra el contenido de uno o más archivos de texto.

TYPE

Muestra el contenido de los archivos incluidos en la lista.

Comando:

```
VER
```

Muestra la versión del sistema operativo Windows.

VER

Muestra la versión de Windows.

Comando:

```
XCOPY
```

Copia árboles de archivos y directorios.

XCOPY

Copia los archivos incluidos en el directorio de origen al directorio de destino.

XCOPY /S

Copia todo el contenido (directorios y archivos) del directorio de origen al directorio de destino.

D. Tabla de Códigos UNICODE

La siguiente tabla muestra los principales caracteres UNICODE usados en Java, con su respectivo valor numérico.

33:	!	**34:**	"	**35:**	#	**36:**	$	**37:**	%
38:	&	**39:**	'	**40:**	(	**41:**	)	**42:**	*
43:	+	**44:**	,	**45:**		**46:**	.	**47:**	/
48:	0	**49:**	1	**50:**	2	**51:**	3	**52:**	4
53:	5	**54:**	6	**55:**	7	**56:**	8	**57:**	9
58:	:	**59:**	;	**60:**	<	**61:**	=	**62:**	>
63:	?	**64:**	@	**65:**	A	**66:**	B	**67:**	C
68:	D	**69:**	E	**70:**	F	**71:**	G	**72:**	H
73:	I	**74:**	J	**75:**	K	**76:**	L	**77:**	M
78:	N	**79:**	O	**80:**	P	**81:**	Q	**82:**	R
83:	S	**84:**	T	**85:**	U	**86:**	V	**87:**	W
88:	X	**89:**	Y	**90:**	Z	**91:**	[	**92:**	\
93:	]	**94:**	^	**95:**	_	**96:**	`	**97:**	a
98:	b	**99:**	c	**100:**	d	**101:**	e	**102:**	f
103:	g	**104:**	h	**105:**	i	**106:**	j	**107:**	k
108:	l	**109:**	m	**110:**	n	**111:**	o	**112:**	p
113:	q	**114:**	r	**115:**	s	**116:**	t	**117:**	u
118:	v	**119:**	w	**120:**	x	**121:**	y	**122:**	z
123:	{	**124:**	|	**125:**	}	**126:**	~	**161:**	¡
162:	¢	**163:**	£	**164:**	¤	**165:**	¥	**166:**	¦
167:	§	**168:**	¨	**169:**	©	**170:**	ª	**171:**	«
172:	¬	**173:**		**174:**	®	**175:**	¯	**176:**	°
177:	±	**178:**	²	**179:**	³	**180:**	´	**181:**	µ
182:	¶	**183:**	·	**184:**	¸	**185:**	¹	**186:**	º
187:	»	**188:**	¼	**189:**	½	**190:**	¾	**191:**	¿
192:	À	**193:**	Á	**194:**	Â	**195:**	Ã	**196:**	Ä

197:	Å	**198:**	Æ	**199:**	Ç	**200:**	È	**201:**	É
202:	Ê	**203:**	Ë	**204:**	Ì	**205:**	Í	**206:**	Î
207:	Ï	**208:**	Ð	**209:**	Ñ	**210:**	Ò	**211:**	Ó
212:	Ô	**213:**	Õ	**214:**	Ö	**215:**	×	**216:**	Ø
217:	Ù	**218:**	Ú	**219:**	Û	**220:**	Ü	**221:**	Ý
222:	Þ	**223:**	ß	**224:**	à	**225:**	á	**226:**	â
227:	ã	**228:**	ä	**229:**	å	**230:**	æ	**231:**	ç
232:	è	**233:**	é	**234:**	ê	**235:**	ë	**236:**	ì
237:	í	**238:**	î	**239:**	ï	**240:**	ð	**241:**	ñ
242:	ò	**243:**	ó	**244:**	ô	**245:**	õ	**246:**	ö
247:	÷	**248:**	ø	**249:**	ù	**250:**	ú	**251:**	û
252:	ü	**253:**	ý	**254:**	þ	**255:**	ÿ	**338:**	Œ
339:	œ	**352:**	Š	**353:**	š	**376:**	Ÿ	**381:**	Ž