

!

*

Introducción a las Estructuras de Datos

APRENDIZAJE ACTIVO BASADO EN CASOS

>

/

{

=



Dogram Code



Jorge A. Villalobos S.



Dogram Code

Accede a Gratis a la Biblioteca Online +300 Libros en PDF

<https://dogramcode.com/biblioteca>

Únete al canal de Telegram

https://t.me/bibliotecagratis_dogramcode

Únete al Grupo de Facebook

<https://www.facebook.com/groups/librosyrecursosdeprogramacion>

Introducción a las **Estructuras** **de Datos**

APRENDIZAJE ACTIVO BASADO EN CASOS

Un Enfoque Moderno Usando Java, UML, Objetos y Eclipse

Introducción a las **Estructuras** **de Datos**

APRENDIZAJE ACTIVO BASADO EN CASOS

Un Enfoque Moderno Usando Java, UML, Objetos y Eclipse

JORGE A. VILLALOBOS S.

Universidad de los Andes
Bogotá - Colombia



COLOMBIA • CHILE • ARGENTINA • BRASIL • COSTA RICA • ESPAÑA
GUATEMALA • MÉXICO • PERÚ • PUERTO RICO • VENEZUELA

Datos de catalogación bibliográfica

Villalobos S., Jorge A.

Introducción a las Estructuras de Datos.

Aprendizaje Activo Basado en Casos – 1^a edición

Pearson Educación de Colombia Ltda., 2008

ISBN: 978-958-699-104-9

Formato: 21 x 27 cm

Páginas: 502

Autor: Jorge A. Villalobos S.

Editora: María Fernanda Castillo

fernanda.castillo@pearsoned.cl

Corrección de estilo: Daniel Soria / Óscar Saldaña / Alessandra Canessa

Diagramación: Víctor Goyburo

PRIMERA EDICIÓN, 2008

D.R. © 2008 por Pearson Educación de Colombia Ltda.

Carrera 65B N° 13 - 62,

Zona Industrial, Bogotá - Colombia

Prentice Hall es una marca registrada de **Pearson Educación de México S. A. de C. V.**

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

ISBN: 978-958-699-104-9

Impreso en Colombia / Printed in Colombia

A mis padres.

Con un inmenso gracias...

JORGE

Tabla de Contenido

Prefacio	XVII
----------------	------

Nivel 1 - Búsqueda, Ordenamiento y Pruebas Automáticas

1. Objetivos Pedagógicos	1
2. Motivación.....	2
3. Caso de Estudio N° 1: Un Traductor de Idiomas	2
3.1. Objetivos de Aprendizaje	4
3.2. Comprensión de los Requerimientos	4
3.3. Comprensión del Mundo del Problema	6
3.4. Invariantes de Clase y Noción de Corrección del Modelo	8
3.5. Asignación de Responsabilidades e Implementación de Métodos.....	16
3.6. Pruebas Unitarias Automáticas	20
3.6.1. Introducción y Motivación	21
3.6.2. Visión Conceptual de las Pruebas	21
3.6.3. JUnit: Un Framework de Construcción de Pruebas.....	26
4. Caso de Estudio N° 2: Un Manejador de Muestras	32
4.1. Objetivos de Aprendizaje	33

4.2. Comprensión de los Requerimientos	34
4.3. Arquitectura de la Solución	36
4.4. Algoritmos de Ordenamiento en Memoria Principal	41
4.4.1. Ordenamiento por Selección.....	41
4.4.2. Ordenamiento por Intercambio (Burbuja).....	45
4.4.3. Ordenamiento por Inserción	48
4.4.4. ¿Y Cuándo Ordenar?	51
4.5. Algoritmos de Búsqueda en Memoria Principal	52
4.5.1. Búsqueda de un Elemento	52
4.5.2. Búsqueda en Estructuras Ordenadas	55
4.6. Generación de Datos y Medición de Tiempos	57
4.7. Pruebas Unitarias Automáticas	58
5. Caso de Estudio N° 3: Una Exposición Canina	62
5.1. Objetivos de Aprendizaje	63
5.2. Comprensión de los Requerimientos	63
5.3. Arquitectura de la Solución	65
5.4. Comparación de Objetos por Múltiples Criterios.....	66
5.5. Métodos de Ordenamiento y Búsqueda de Objetos	69
5.6. Manejo de Grupos de Valores en la Interfaz de Usuario	71
6. Glosario de Términos.....	74
7. Hojas de Trabajo	75
7.1. Hoja de Trabajo N° 1: Bolsa de Empleo.....	75
7.2. Hoja de Trabajo N° 2: Venta de Vehículos	85

Nivel 2 - **Archivos, Serialización y Tipos de Excepción**

1. Objetivos Pedagógicos	97
2. Motivación.....	98
3. Caso de Estudio N°1: Un Explorador de Archivos	99
3.1. Objetivos de Aprendizaje	100
3.2. Comprensión de los Requerimientos	101
3.3. Modelo del Mundo del Problema	102

3.4. Expresiones Condicionales en Java	103
3.5. Manipulación Básica de Archivos	104
3.6. Lectura de Archivos	107
3.7. Manipulación de Cadenas de Caracteres.....	110
3.8. Escritura de Archivos	115
3.9. Pruebas Unitarias con Archivos	117
3.10. El Componente JTextArea	121
3.11. Extensión del miniExplorer	122
4. Visión Global de la Persistencia	125
5. Caso de Estudio N°2: Una Tienda Virtual de Discos	126
5.1. Objetivos de Aprendizaje	128
5.2. Comprensión de los Requerimientos	129
5.3. Diseño e Implementación de las Excepciones	130
5.4. Arquitectura y Responsabilidades en el Modelo del Mundo.....	137
5.5. Reportes y Otras Salidas en Archivos.....	145
5.6. Importación de Datos desde Archivos.....	149
5.7. Construcción de Pruebas Unitarias.....	152
5.8. Persistencia Simple por Serialización	153
5.9. El Componente JComboBox.....	159
6. Uso Básico del Depurador en Eclipse	161
7. Glosario de Términos	163
8. Hojas de Trabajo	164
8.1. Hoja de Trabajo N° 1: Campeonato de Fórmula 1	164
8.2. Hoja de Trabajo N° 2: Mundial de Fútbol	169

Nivel 3 - Estructuras Lineales Enlazadas

1. Objetivos Pedagógicos	175
2. Motivación.....	176
3. Caso de Estudio N° 1: Una Central de Pacientes	177
3.1. Objetivos de Aprendizaje	178

3.2. Comprensión de los Requerimientos	178
3.3. Referencias y Ciclo de Vida de los Objetos.....	180
3.4. Del Análisis al Diseño	185
3.5. Estructuras Lineales Enlazadas.....	188
3.6. Algorítmica Básica	191
3.6.1. Localización de Elementos y Recorridos	191
3.6.2. Supresión de Elementos	196
3.6.3. Inserción de Elementos	199
3.7. Patrones de Algoritmo	204
3.8. Nuevos Componentes Gráficos	206
4. Caso de Estudio N° 2: Reservas en una Aerolínea	210
4.1. Objetivos de Aprendizaje	211
4.2. Diagramas de Casos de Uso	212
4.3. Comprensión de los Requerimientos	213
4.4. Del Análisis al Diseño	215
4.5. Arreglos de Constantes.....	219
4.6. Manejo de Fechas y Formatos	220
4.7. Estructuras Ordenadas Dblemente Enlazadas	221
4.8. Nuevos Distribuidores Gráficos	229
4.8.1. El Distribuidor Secuencial	229
4.8.2. El Distribuidor GridBagLayout.....	231
5. Glosario de Términos	234
6. Hojas de Trabajo	235
6.1. Hoja de Trabajo N° 1: Una Agenda	235
6.2. Hoja de Trabajo N° 2: Un Manejador de ADN	244
Nivel 4 - Mecanismos de Reutilización y Desacoplamiento	
1. Objetivos Pedagógicos	253
2. Motivación	254
3. El Caso de Estudio N° 1: Un Editor de Dibujos	255
3.1. Objetivos de Aprendizaje	257

3.2. Comprensión de los Requerimientos	258
3.3. Interfaces: Compromisos Funcionales.....	260
3.4. Referencias de Tipo Interfaz	262
3.5. Construcción de una Clase que Implementa una Interfaz.....	263
3.6. Interfaces para Contenedoras	265
3.7. Iteradores para Recorrer Secuencias.....	268
3.8. Implementación del Editor de Dibujos	271
3.9. Otras Interfaces Usadas Anteriormente	273
3.10. La Herencia como Mecanismo de Reutilización.....	274
3.10.1. Superclases y Subclases	276
3.10.2. Elementos Privados y Elementos Protegidos	283
3.10.3. Acceso a Métodos de la Superclase	285
3.10.4. La Clase Object	288
3.10.5. Extensión de Interfaces	292
3.11. Uso del Mecanismo de Herencia en Niveles Anteriores	293
3.12. Los Componentes de Manejo de Menús.....	294
3.13. Manejo de Eventos del Ratón	299
3.14. Dibujo Básico en Java	302
3.14.1. Modelo Gráfico de Java	302
3.14.2. Manejo de la Superficie de Dibujo.....	304
3.14.3. Manejo de Formas Geométricas	306
3.15. Evolución del Editor.....	310
4. Glosario de Términos	315
5. Hojas de Trabajo	316
5.1. Hoja de Trabajo N° 1: Mapa de la Ciudad.....	316

Nivel 5 - Estructuras y Algoritmos Recursivos

1. Objetivos Pedagógicos	327
2. Motivación.....	328
3. Caso de Estudio N°1: Un Directorio de Contactos.....	330
3.1. Objetivos de Aprendizaje	331
3.2. Comprensión de los Requerimientos	331

3.3. Árboles Binarios Ordenados.....	332
3.3.1. Algunas Definiciones.....	334
3.3.2. Proceso de Búsqueda.....	335
3.3.3. Proceso de Inserción	337
3.3.4. Proceso de Supresión.....	338
3.4. Del Análisis al Diseño	340
3.5. Algoritmos Recursivos.....	345
3.5.1. El Algoritmo de Búsqueda	346
3.5.2. El Algoritmo de Inserción	348
3.5.3. Algoritmos para Calcular Propiedades de un Árbol.....	351
3.5.4. El Algoritmo de Supresión	356
3.5.5. El Algoritmo de Recorrido en Inorden	358
3.5.6. Verificación del Invariante	361
3.5.7. Patrones de Algoritmo para Árboles Binarios	362
4. Caso de Estudio N° 2: Organigrama de una Empresa.....	364
4.1. Objetivos de Aprendizaje	365
4.2. Comprensión de los Requerimientos	365
4.3. Árboles n-arios.....	367
4.4. Del Análisis al Diseño	368
4.5. Algorítmica de Árboles n-arios.....	371
4.5.1. Los Algoritmos de Búsqueda	371
4.5.2. Algoritmos para Calcular Propiedades	376
4.5.3. Los Algoritmos de Inserción y Supresión	380
4.5.4. Los Algoritmos de Modificación.....	383
4.5.5. El Algoritmo de Recorrido en Preorden	384
4.5.6. Patrones de Algoritmo para Árboles n-arios	385
4.5.7. Extensiones al Caso de Estudio	386
5. Glosario de Términos.....	388
6. Hojas de Trabajo	389
6.1. Hoja de Trabajo N° 1: Juego 8-Puzzle.....	389
6.2. Hoja de Trabajo N° 2: Juego Genético	397

Nivel 6 - **Bases de Datos y Distribución Básica**

1. Objetivos Pedagógicos	407
2. Motivación.....	408
3. Caso de Estudio N° 1: Juego Batalla Naval.....	409
3.1. Objetivos de Aprendizaje	412
3.2. Comprensión de los Requerimientos	412
3.3. Modelo del Mundo del Problema.....	414
3.4. Primer Borrador de Arquitectura.....	416
3.5. Canales de Comunicación entre Programas.....	417
3.5.1. El Proceso de Conexión.....	418
3.5.2. El Proceso de Escritura	420
3.5.3. El Proceso de Lectura	421
3.5.4. Los Protocolos de Comunicación	423
3.6. Primer Refinamiento de la Arquitectura	424
3.7. Concurrencia en un Programa.....	435
3.8. Introducción a las Bases de Datos.....	441
3.8.1. Estructura y Definiciones	441
3.8.2. El Lenguaje de Consulta SQL.....	442
3.8.3. El Framework JDBC.....	446
3.8.4. Persistencia en el Caso de Estudio	451
3.9. Extensiones al Caso de Estudio	452
4. Glosario de Términos	454
5. Hojas de Trabajo	455
5.1. Hoja de Trabajo N° 1: Registro de Ventas de un Almacén	455
5.2. Hoja de Trabajo N° 2: Mensajería Instantánea	462

Anexo A - **Tabla de Códigos UNICODE**

1. Tabla de Códigos.....	477
---------------------------------	------------

Prefacio

Objetivos

Este libro tiene como objetivo introducir las estructuras de datos básicas, que permiten almacenar la información de un problema de una forma adecuada, teniendo en cuenta criterios como eficiencia y facilidad de evolución. Se incluyen, entre otros temas, el manejo de estructuras lineales enlazadas (listas), estructuras recursivas simples (árboles binarios y árboles n-arios) y archivos de texto. Se hace especial énfasis en las herramientas de apoyo a la construcción de programas correctos, y en algunos elementos de la programación orientada a objetos que facilitan la reutilización y el desacoplamiento de los distintos componentes de un programa. Se estudian los principales algoritmos de búsqueda y ordenamiento en memoria principal, se genera la habilidad de construir algoritmos recursivos y se aborda el problema de incluir los requerimientos no funcionales de persistencia y distribución en un programa.

Esto nos permite dar una visión integral de la problemática de representar y manipular la información de un problema, como parte de un proceso de desarrollo de software.

El Público Objetivo

Este libro está dirigido a estudiantes que se encuentran cursando un segundo o tercer curso en el tema de programación, y que son capaces de construir programas de computador en el lenguaje Java para resolver problemas simples.

Requisitos

Para aprovechar de manera adecuada este libro, es conveniente que el lector se encuentre familiarizado con los siguientes temas: programación básica en Java, sintaxis del diagrama de clases de UML, conceptos de programación orientada a objetos, habilidad en el uso de un ambiente integrado de desarrollo (IDE) como Eclipse y conceptos básicos de ingeniería de software.

Si el lector considera que no cumple alguno de los requisitos antes mencionados, se recomienda consultar el libro "Fundamentos de Programación: Aprendizaje Activo Basado en Casos", en donde se tratan de manera incremental los temas básicos de programación utilizando el lenguaje Java. De la misma manera se

recomienda consultar y utilizar los recursos públicos disponibles en el sitio <http://cupi2.uniandes.edu.co>.

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 equilibrio 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 que se encuentra la manera correcta de escribir un programa de computador.

- **Basado en problemas:** El libro gira alrededor de 22 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 modelaje UML, el ambiente de desarrollo de programas Eclipse, las técnicas de la programación orientada por objetos, el framework de pruebas JUnit y el manejador de bases de datos DERBY.

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 en que esos elementos se comunican, etc.

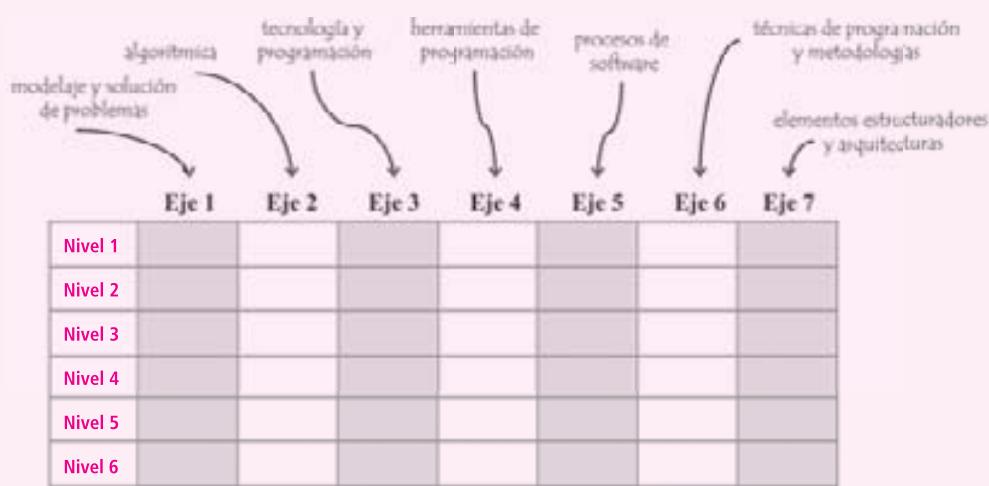
La Estructura del Libro

El libro sigue una estructura de niveles, en la 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 seis 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 transversal los elementos de los siete ejes conceptuales, dando lugar a una estructura como la que se presenta a continuación:



Las Herramientas y Recursos de Apoyo

Éste es un libro de trabajo para el estudiante, donde puede realizar sus tareas y ejercicios asociados con cada nivel. Consideramos el CD que acompaña al libro como parte integral del mismo. Todos los casos de estudio que se utilizan en los distintos niveles están resueltos e incluidos en dicho CD, 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 el CD al mismo tiempo que lee el libro.

En el CD se encuentran cuatro tipos de elementos: (1) los programas de los casos de estudio, (2) los programas de las hojas de trabajo, (3) los instaladores de algunas herramientas de desarrollo y (4) los entrenadores sobre ciertos conceptos. El CD ha sido construido de manera que sea fácil navegar por los elementos que lo constituyen.

Todo el contenido del CD de apoyo, lo mismo que otros materiales de apoyo al profesor, se pueden 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 de distintas herramientas y productos comerciales, todos los cuales tienen sus marcas registradas. Éstos son: Microsoft Windows®, Microsoft Word®, Rational ROSE®, Java®, Mozilla Firefox®, Eclipse®, JUnit® y Derby®.

En el CD que acompaña al libro van las licencias de uso de las herramientas que allí incluimos.

Todas las herramientas, programas, tutoriales y demás materiales desarrollados como soporte y complemento del libro (los cuales se encuentran en el CD), se distribuyen bajo la licencia "Academic Free License v. 2.1", que se rige por lo definido en: <http://opensource.org/licenses/>.

Agradecimientos

Este libro es el producto del trabajo de un gran número de personas, todas ellas con un alto nivel de compromiso y profesionalismo. Como primera medida quiero reconocer el apoyo directo que obtuve de Katalina Marcos, Marcela Hernández, Rubby Casallas, Mario Sánchez, Milena Vela, Daniel Romero, Pablo Márquez, Manuel Muñoz y Carlos Vega. Sin ellos este libro no sería una realidad. Gracias por su dedicación y por el corazón que siempre le pusieron al proyecto.

También debo reconocer y agradecer el trabajo de los profesores que han dictado el curso utilizando borradores de este libro, quienes han hecho aportes y comentarios que han resultado muy valiosos.

Con respecto a la editorial, quiero agradecer a Peter Vargas, Julio César Beltrán y Fernanda Castillo, por creer en el proyecto, y por haber llevado a cabo un excelente proceso de edición.

Por último quiero agradecer a Vicky, por su paciencia y apoyo en la elaboración de este libro. Siempre con las palabras de aliento necesarias para salir adelante.

Sobre el Autor

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 codirige 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 "Fundamentos de Programación: Aprendizaje Activo Basado en Casos" (2006), "Diseño y Manejo de Estructuras de Datos en C" (1996), "Estructuras de Datos: Un Enfoque desde Tipos Abstractos" (1990).

Iconos y Convenciones Tipográficas

En esta sección presentamos los iconos utilizados a lo largo del libro y el significado que queremos asociarles.



Es una referencia a algo que se puede encontrar en el CD que acompaña al libro o en el sitio web del proyecto.



Es una tarea que se deja al lector. Toda tarea tiene un objetivo y un enunciado. El primero establece la intención, mientras el segundo contiene las instrucciones que el lector debe seguir para resolver la tarea.



Éste es el símbolo que usamos para introducir un ejemplo. En los ejemplos siempre se hace explícito su objetivo y su alcance.



Periódicamente hacemos una síntesis de las principales ideas que se han presentado en cada nivel y usamos este símbolo para indicar cuáles son las definiciones e ideas que el lector debe retener.



Este símbolo de advertencia lo asociamos a puntos en los cuales el lector debe tener cuidado, ya sea porque es una fuente frecuente de problemas o porque es un concepto que se presta a distintas interpretaciones.



Corresponde a una tarea de programación, en la cual el lector debe realizar alguna modificación o extensión a uno de los casos de estudio del libro.



Este símbolo lo usamos para pedir al lector que verifique alguna cosa.



Al final de cada nivel hay dos hojas de trabajo, cada una sobre un problema distinto. Cada hoja de trabajo está estructurada como una secuencia de tareas, las cuales deben ayudar a reforzar los conceptos y técnicas introducidos a lo largo del nivel.



El glosario es una sección presente al final de cada nivel, donde se le pide al lector que escriba las principales definiciones que se han visto en el capítulo. Esto permite al lector explicar en sus propias palabras los principales términos estudiados.



Usamos este símbolo para resumir las convenciones que se recomienda seguir al escribir un programa en el lenguaje Java. Una convención es una regla que, sin ser obligatoria, ayuda a escribir programas de mejor calidad.

Usamos las siguientes convenciones tipográficas:

Negrita. Se usa para resaltar un concepto importante que se acaba de introducir en el texto.

Itálica. Las palabras en itálica corresponden a términos en inglés, cuya traducción a español consideramos innecesaria, debido a que el lector va a encontrar esas palabras en dicho idioma en las herramientas de desarrollo. En algunos casos, al lado del término en español, colocamos también la traducción en inglés.

Courier. Este tipo de letra lo usamos en los programas que aparecen en el libro y, dentro del texto, para hacer referencia a los elementos de un programa. Dichas palabras no respetan, en algunos casos, las reglas del idioma español, puesto que es una cita textual de un elemento de un programa, el cual tiene reglas distintas de escritura (no acepta tildes o el carácter "ñ").

Courier. Dentro de los programas, este tipo de letra en negrita lo usamos para identificar las palabras reservadas del lenguaje Java.

Comentario. Todos los comentarios que hacemos en los ejemplos, las figuras o los programas van en este tipo de letra. Queremos así distinguir lo que es teoría de las explicaciones adicionales asociadas con un ejemplo en particular.

Nivel 1

Búsqueda, Ordenamiento y Pruebas Automáticas

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Escribir el invariante de una clase e implementar los métodos necesarios para su verificación, utilizando la instrucción `assert` de Java.
- Desarrollar las clases y los métodos necesarios para implementar las pruebas unitarias automáticas, que ayudan a comprobar el correcto funcionamiento de un programa.
- Entender la importancia de construir programas correctos y la manera como los invariantes, los contratos de los métodos y las pruebas unitarias son fundamentales en este propósito.
- Implementar una interfaz de usuario en la cual se despliegue y se permita la interacción con un grupo de objetos, utilizando algunos de los componentes gráficos básicos que ofrece el lenguaje Java.
- Implementar, adaptar y utilizar algunos algoritmos clásicos de búsqueda de información sobre una estructura ordenada o desordenada.
- Implementar, adaptar y utilizar algunos de los algoritmos no recursivos de ordenamiento de información.

2. Motivación

Con este nivel comenzamos una nueva etapa en el proceso de aprendizaje de programación de computadores, en la cual nos encontraremos con nuevos problemas que nos van a llevar a estudiar otras herramientas, instrucciones y técnicas para resolverlos.

El primer problema al que nos vamos a enfrentar es al problema de la corrección de lo que desarrollemos. ¿Cómo estar seguros de que, en el momento de entregarle al cliente un programa, éste funciona adecuadamente? ¿Qué herramientas tenemos que nos ayuden a desarrollar programas correctos, garantizando que siempre vamos a construir programas de alta calidad? Esas dos preguntas deben ser contestadas por cualquier programador que espere que sus desarrollos sean utilizados en un contexto real. Los clientes están cada vez menos dispuestos a pagar por soluciones que tienen errores o que no tienen el funcionamiento esperado. ¿Qué tal un error en el programa de facturación de una empresa? ¿Cuánto dinero e imagen le puede costar eso a una compañía? En un mundo con tan alto nivel de competencia, no hay espacio para los desarrollos defectuosos y de baja calidad, ni para los programadores que no puedan garantizar la corrección de sus trabajos.

En este nivel veremos dos elementos fundamentales en la construcción de programas correctos, que deben integrarse al proceso de desarrollo de programas que hemos presentado hasta ahora: los invariantes de las clases y las pruebas unitarias automáticas. Los primeros sirven para verificar que, durante la ejecución del programa, la información que hay en los objetos del modelo del mundo sea coherente. Las pruebas unitarias, por su parte, son unos pequeños programas que se desarrollan aparte y que son capaces de probar el correcto funcionamiento de los métodos del modelo del mundo, dentro de contextos controlados.

El segundo problema crítico que enfrentan los programadores es la eficiencia de los programas que escriben. Un usuario, en general, espera que el programa sea capaz de contestar sus requerimientos

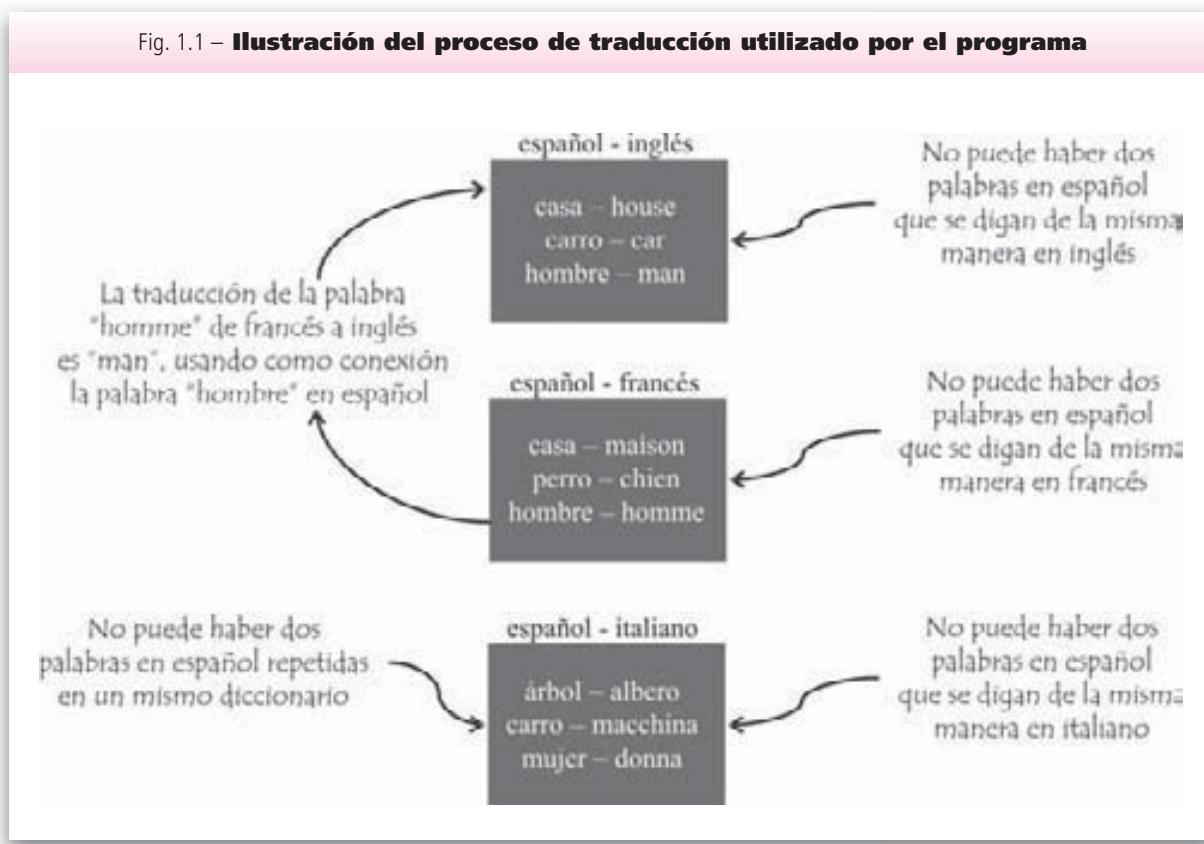
rápidamente. ¿Cómo construir entonces programas que sean eficientes? La respuesta a esta pregunta es bastante más difícil de contestar, puesto que intervienen muchos factores que iremos viendo a lo largo de los niveles que siguen. Por ahora vamos únicamente a abordar el problema de buscar eficientemente información en un grupo de valores, puesto que dicha actividad es fundamental en muchos programas, y una buena cantidad del tiempo de ejecución se va en localizar algo que el usuario quiere consultar. Si ese es el caso, resulta muy conveniente ordenar la información que maneja el programa (objetos o valores simples) para que sea fácil de encontrar. Tanto el tema de búsqueda como el de ordenamiento los estudiaremos en este nivel.

Otro problema que se le presenta con frecuencia al programador es la construcción de la interfaz de usuario. Más de la mitad de los problemas que reportan los clientes (y que muchas veces llevan al fracaso de un proyecto) tienen que ver con la dificultad de uso del programa. En este nivel veremos los componentes gráficos de una interfaz de usuario que permiten visualizar e interactuar con un grupo de elementos. Esto va a ayudar a construir aplicaciones en donde el objetivo central sea la búsqueda y consulta de información.

3. Caso de Estudio N° 1: Un Traductor de Idiomas

En este primer caso de estudio del nivel, queremos construir un programa de computador que permita traducir palabras del español a otros idiomas (inglés, francés e italiano) y de los otros idiomas entre ellos, haciendo la suposición de que las traducciones son únicas. Para esto el programa debe contar con tres diccionarios, que pueden tener conjuntos de palabras distintas. Esto quiere decir que no necesariamente las mismas palabras que tienen traducción al inglés tienen traducción al francés y viceversa. Lo anterior se ilustra en la figura 1.1, en la cual aparecen tres diccionarios: español-inglés, español-francés y español-italiano.

Fig. 1.1 – Ilustración del proceso de traducción utilizado por el programa



Para que el programa funcione correctamente hay dos características que deben respetar los diccionarios. La primera es que no hay dos palabras en español repetidas en ningún diccionario (cada palabra tiene una sola traducción en el otro idioma). La segunda es que no hay dos palabras en español con la misma traducción en ningún diccionario (cada palabra tiene una traducción que es única). Con estas dos características, podemos utilizar los tres diccionarios para hacer traducciones entre los cuatro idiomas: si es del español a otro idioma, sencillamente utilizamos el respectivo diccionario. Si es de un idioma distinto al español a cualquier otro idioma, lo que hacemos es encontrar en el diccionario que corresponde la palabra en español de la cual la palabra que buscamos es una traducción, y luego consultamos la traducción de dicha palabra en el otro diccionario. Por ejemplo, si el usuario quiere traducir la palabra "homme" de francés a inglés, primero usamos el diccionario español-francés para determinar que "homme" es la traducción en francés de la palabra "hombre". Luego, usamos el diccionario español-inglés

para buscar la traducción de "hombre" y encontramos la palabra "man".

Se espera que el programa provea las siguientes opciones al usuario, utilizando la interfaz de usuario presentada en la figura 1.2: (1) Agregar una palabra en español y su traducción a cualquiera de los tres idiomas de los cuales el programa tiene un diccionario. Si la palabra en español o su traducción ya existen en ese diccionario, debe informar al usuario que no es posible agregar esta nueva entrada. (2) Buscar la traducción de una palabra escrita en cualquiera de los cuatro idiomas que maneja el traductor, al idioma seleccionado por el usuario. Si el idioma de origen es español, el programa busca simplemente en el diccionario del idioma de destino. Si el idioma de origen es inglés, francés o italiano, el programa busca la palabra en español de la cual esta palabra es una traducción y, si ésta existe, busca en el diccionario del idioma destino la traducción. (3) Informar el número de palabras presentes en cada uno de los tres diccionarios.

Fig. 1.2 – Interfaz de usuario del traductor de idiomas



■ La ventana está dividida en tres zonas: la primera para las consultas, la segunda para agregar palabras y la tercera con información sobre el número de palabras en cada diccionario.

■ Para consultar una palabra el usuario debe teclearla en la primera zona de texto, luego debe seleccionar el "Idioma origen" y el "Idioma destino" y finalmente oprimir el botón "Traducir".

■ Para agregar una palabra a un diccionario, en la segunda zona debe teclear la palabra y su traducción, y luego seleccionar el "Idioma traducción" y oprimir el botón "Aregar".

3.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?

■ Construir un programa siguiendo un proceso disciplinado de desarrollo.

■ Utilizar los mecanismos disponibles para ayudar a construir un programa correcto.

¿Qué tenemos que aprender o reforzar?

■ Revisar y seguir el proceso de desarrollo de programas estudiado en cursos anteriores.

■ Aprender a construir el invariante de una clase y a implementar los métodos que van a permitir verificar que éste se cumpla en todo momento.

■ Aprender a desarrollar las pruebas unitarias automáticas para verificar que los métodos de una clase cumplen con su contrato (en un contexto controlado) y aprender a utilizarlas de manera que apoyen el proceso de desarrollo de programas.

3.2. Comprensión de los Requerimientos



El primer paso cuando se va a resolver un problema es analizarlo, lo cual 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.

Un requerimiento funcional es un servicio u operación que el programa debe proveer al usuario. Se especifica con un nombre, un resumen, unas entradas (qué información debe suministrar el usuario que no tenga ya el programa) y un resultado (qué efecto produce la ejecución de dicho servicio).

Tarea 1

Objetivo: Entender el problema del caso de estudio del traductor.

(1) Lea detenidamente el enunciado del caso de estudio del traductor e (2) identifique y complete la documentación de los tres requerimientos funcionales que allí aparecen.

Requerimiento funcional 1	Nombre	R1 – Agregar una palabra en español a un diccionario
	Resumen	
	Entradas	(1)
		(2)
		(3)
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Consultar la traducción de una palabra
	Resumen	
	Entradas	(1)
		(2)
		(3)
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Consultar el número de palabras en cada diccionario
	Resumen	
	Entradas	
	Resultado	

3.3. Comprensión del Mundo del Problema

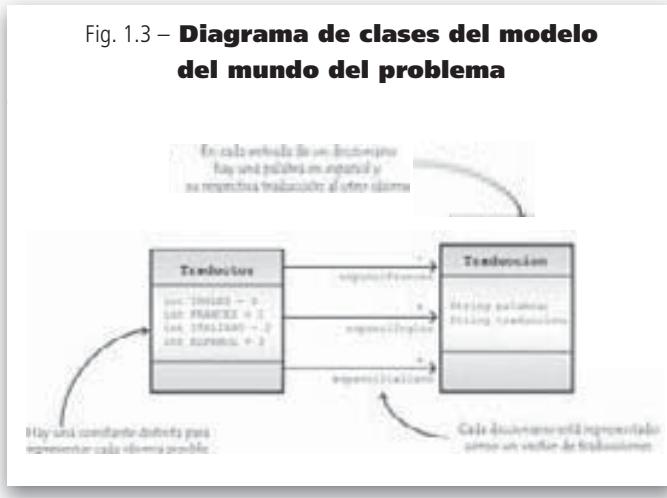


El mundo o contexto en el cual se presenta el problema es el segundo componente que se debe estudiar en la etapa de análisis.

En esta etapa se deben realizar cinco tareas: (1) identificar y nombrar las entidades del mundo del problema, (2) identificar para cada una de ellas sus atributos y tipo, (3) identificar la necesidad de utilizar constantes para modelar algunas características, (4) buscar las relaciones entre las entidades y sus propiedades y (5) escribir el diagrama de clases utilizando el lenguaje UML.

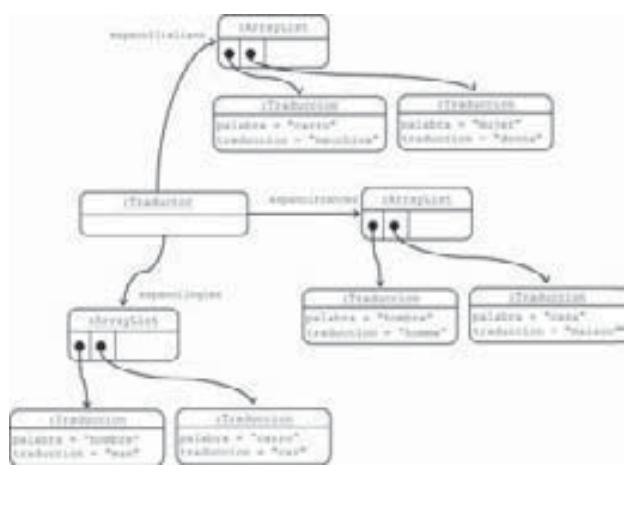
En la figura 1.3 aparece el diagrama de clases del traductor. Allí figuran dos entidades (**Traductor** y **Traducción**) con tres asociaciones entre ellas. Cada una de estas asociaciones representa un diccionario, que está constituido por un grupo de traducciones. Cada traducción, por su parte, es una pareja de la forma palabra-traducción. Para representar los idiomas, agregamos en el modelado cuatro constantes, una por idioma.

Fig. 1.3 – **Diagrama de clases del modelo del mundo del problema**



En algunos casos es conveniente ilustrar el diagrama de clases de un modelo mediante un diagrama de objetos, que no es otra cosa que un ejemplo de cómo podría materializarse el diagrama de clases en un escenario particular. En la figura 1.4 aparece un diagrama de objetos posible para el traductor. Allí figuran, por ejemplo, dos palabras en el diccionario español-francés ("hombre" y "casa") con sus respectivas traducciones ("homme" y "maison"). También aparecen dos palabras en el diccionario español-inglés ("hombre" y "carro") con las palabras del idioma inglés con las que se traducen ("man" y "car"). Finalmente, en el diccionario español-italiano se encuentran las parejas "carro-macchina" y "mujer-donna". Estudie detenidamente el diagrama que se muestra en la figura y vea la manera como se representan los vectores y los objetos contenidos en cada uno de ellos.

Fig. 1.4 – **Diagrama de objetos para ilustrar un estado posible del modelo del mundo**



Después de representar el mundo del problema en términos de clases (entidades), características (atributos) y relaciones entre entidades (asociaciones), se procede a hacer las respectivas declaraciones en el lenguaje Java. Ese es el objetivo de la siguiente tarea.

Tarea 2

Objetivo: Escribir en Java las declaraciones de las clases del modelo del mundo.

Utilizando los diagramas presentados en las figuras 1.3 y 1.4, escriba en Java las declaraciones de las clases. Suponga que éstas van en el paquete `uniandes.cupi2.traductor.mundo`.

```
package uniandes.cupi2.traductor.mundo;

public class Traductor
{

}
```

```
package uniandes.cupi2.traductor.mundo;

public class Traduccion
{

}
```

3.4. Invariantes de Clase y Noción de Corrección del Modelo

La presentación de esta sección la haremos contestando siete preguntas e ilustrando las respuestas con el problema planteado en el caso de estudio.

- ¿Cuál es el problema que se quiere resolver?

Antes de comenzar a hablar de los invariantes de clase, debemos entender cuál es el problema que tenemos y qué consecuencias tiene en el proceso de desarrollo de software. Para eso veamos de nuevo el enunciado del caso de estudio y busquemos qué información importante no quedó plasmada ni en los requerimientos funcionales ni en el modelo del mundo. ¿Dónde quedó la restricción que dice que no puede haber palabras repetidas en los diccionarios? ¿Dónde quedó la propiedad de los diccionarios de que las traducciones son únicas? La respuesta es que no quedó en ninguna parte del análisis que hicimos del problema, y que esa información sólo aparecerá más adelante, cuando hagamos los contratos de los métodos de las clases.

Aquí aparecen por lo menos tres problemas: (1) Falta de documentación de las características del modelo del mundo. El objetivo que nos habíamos planteado de que cualquiera que leyera los documentos producidos en la etapa de análisis entendiera el problema completo no se está cumpliendo. ¡Eso hay que corregirlo! (2) Duplicación de la información en los contratos de los métodos. Si la clase `Traductor` tiene 10 métodos, en la precondición de los contratos de todos ellos debe ir la información con las propiedades de los diccionarios. Por todos lados debemos estar repitiendo que las traducciones son únicas, por ejemplo. Eso no es grave, pero en problemas grandes eso representa una pérdida de tiempo. (3) Poca claridad en la asignación de la responsabilidad de verificar si las propiedades definidas para una clase se cumplen. ¿Debe haber un método de la clase `Traductor` que verifique que las traducciones sean únicas? ¿Siempre hay que llamarlo antes de invocar cualquier otro método de la clase para asegurarnos que se cumple la precondición del contrato?

La respuesta a esta última pregunta es "claro que no" (por razones de eficiencia), pero entonces, ¿cómo hacer para asegurarnos de que siempre se cumplan las propiedades que definimos para una clase? Fíjese que si todos los métodos de la clase se comprometen a dejar sus objetos en un estado que cumpla esas propiedades, no habría necesidad de verificarlo antes de llamar los otros métodos.

En resumen, necesitamos una manera de documentar y definir de manera clara lo que se quiere decir en el párrafo anterior con "las propiedades de una clase", y ajustar el concepto de contrato de un método para que lo tenga en cuenta.

- ¿Qué es el invariante?

El **invariante de una clase** es un conjunto de aserciones (afirmaciones) que indican las propiedades que en todo momento deben cumplir las instancias de esa clase, y que pueden utilizarse como suposiciones dentro de todos los métodos sin necesidad de que aparezcan en las precondiciones y sin necesidad de verificarlos.

Un invariante está compuesto por (1) restricciones sobre los valores que pueden tomar los atributos de la clase, (2) restricciones sobre los valores que pueden tomar los objetos hacia los cuales hay una asociación y (3) relaciones entre los atributos y/o los objetos con los cuales se relaciona.

El hecho de sacar de los contratos las aserciones que se pasan al invariante, nos obliga a replantear la noción de contrato de un método de la siguiente manera:



Todo método (que no sea un constructor) puede suponer al comienzo de su ejecución que se cumplen todas las afirmaciones que aparecen en el invariante de clase y en la precondición del contrato, y se compromete a que después de haber sido ejecutado sobre un objeto, éste cumple todas las afirmaciones del invariante y de la postcondición.

**Ejemplo 1**

Objetivo: Identificar el invariante de las clases del caso de estudio.

En este ejemplo se muestran y se explican las aseraciones que hacen parte del invariante de las clases Traductor y Traducción.

	Invariante	Explicación
Clase Traducción	<ul style="list-style-type: none"> • palabra != null • !palabra.equals("") • traducion != null • !traducion.equals("") 	<ul style="list-style-type: none"> ■ En una traducción la palabra no puede ser nula ni vacía. ■ En una traducción la palabra traducida no puede ser nula ni vacía. ■ Si esas cuatro condiciones se cumplen, consideramos que una traducción es válida, y como tal puede hacer parte de un diccionario. ■ Estas cuatro condiciones no hay necesidad de inclirlas en las precondiciones de los métodos de la clase, sino que se suponen siempre ciertas. ■ Más adelante veremos quién es responsable de garantizar que estas condiciones siempre se cumplan.
Clase Traductor	<ul style="list-style-type: none"> • espanolIngles != null • espanolFrances != null • espanolItaliano != null • En el vector espanolIngles no hay palabras repetidas • En el vector espanolFrances no hay palabras repetidas • En el vector espanolItaliano no hay palabras repetidas • En el vector espanolIngles no hay traducciones repetidas • En el vector espanolFrances no hay traducciones repetidas • En el vector espanolItaliano no hay traducciones repetidas 	<ul style="list-style-type: none"> ■ El invariante de esta clase consta de nueve aseraciones. ■ Las tres primeras son restricciones sobre los valores que pueden tener los tres vectores que representan los diccionarios del traductor. Aquí afirmamos que están convenientemente inicializados. Fíjese que esto va a definir hasta dónde va la responsabilidad de los métodos constructores. ■ Las siguientes aseraciones indican que en un diccionario no hay palabras repetidas. En un invariante se puede utilizar lenguaje natural (español) mientras lo que se diga sea suficientemente preciso. También se puede utilizar notación matemática. Por ejemplo, la cuarta asercción se podría haber escrito así: <pre>espanolIngles_i.palabra != espanolIngles_k.palabra, para todo i != k</pre> ■ Las últimas tres aseraciones dicen que las traducciones son únicas en cada uno de los diccionarios. ■ Todos los métodos de la clase tienen derecho a suponer que todo lo que se afirma en el invariante es cierto cuando el método va a comenzar a ejecutarse.

- ¿Cómo calcular el invariante?

En un invariante hay afirmaciones de dos tipos. Unas que vienen del análisis y que indican algunas propiedades que tiene en el mundo el elemento que está siendo representado por la clase. Es el caso, por ejemplo, de la afirmación de que las traducciones son únicas en un diccionario. Así son los diccionarios que describen el enunciado del caso de estudio. Otras afirmaciones vienen de decisiones de diseño. Cuando decimos en el invariante de la clase `Traductor` que los vectores que representan los diccionarios son distintos de `null`, estamos expresando una decisión de diseño que tiene que ver con quién tiene la responsabilidad de inicializar estos atributos.

Para definir el invariante se debe revisar inicialmente el enunciado y sacar de allí toda la información de restricciones o relaciones que puedan aparecer en los elementos del mundo. Luego, se incluyen las aserciones correspondientes al modelado de las características. Si, por ejemplo, un atributo es un entero que sólo puede tomar como valor algunas constantes que definimos especialmente con ese propósito, se debe incluir esta información en el invariante. Por último, se revisan las responsabilidades de construcción y se decide quién es respon-

sable de inicializar los atributos. Si es el constructor, debe ir al invariante. Si todos los métodos deben verificar si ya están inicializados los atributos y hacerlo en caso de que no sea así, en el invariante se debe simplemente omitir cualquier referencia a la inicialización.

El último punto en esta parte es cómo determinar cuál clase es responsable de hacer las afirmaciones. En el caso de estudio, por ejemplo, ¿por qué la clase `Traducion` no dice que no hay palabras repetidas en el diccionario? La respuesta es simple, y es porque una traducción no sabe que hace parte de un diccionario, luego no puede referirse a las demás traducciones. Es el traductor, quien almacena las traducciones en un vector, el único responsable de establecer la condición de que no puede haber palabras repetidas dentro de ese grupo de objetos.

- ¿Cómo documentar el invariante de una clase?

El invariante de una clase se debe documentar como parte del Javadoc que describe la clase, de manera que cuando se genere la documentación de la clase esta información resulte explícita. La documentación de los invariantes del caso de estudio sería la siguiente:

```
/**
 * Traductor de palabras de español, inglés, francés e italiano.
 * inv:
 *   * espanolIngles != null
 *   * espanolFrances != null
 *   * espanolItaliano != null
 *
 *   * En el vector espanolIngles no hay palabras repetidas
 *   * En el vector espanolFrances no hay palabras repetidas
 *   * En el vector espanolItaliano no hay palabras repetidas
 *
 *   * En el vector espanolIngles no hay traducciones repetidas
 *   * En el vector espanolFrances no hay traducciones repetidas
 *   * En el vector espanolItaliano no hay traducciones repetidas
 */
public class Traductor
{
    ...
}
```

```

/**
 * Representa una palabra y su traducción en otro idioma
 * inv:
 * palabra != null
 * !palabra.equals( "" )
 * traducción != null
 * !traducción.equals( "" )
 */
public class Traducción
{
    ...
}

```

- ¿Cómo verificar el invariante durante la ejecución?

Antes de presentar la manera de escribir los métodos que van a permitir verificar si el invariante se cumple, vamos a presentar una nueva instrucción del lenguaje Java, disponible desde la

versión 1.4, que tiene algunas características que la hacen muy interesante para escribir este tipo de métodos.

La instrucción `assert` de Java permite verificar una aserción. Para esto utiliza la siguiente sintaxis:

assert expresión;

 La expresión debe ser de tipo lógico. Si al evaluar la expresión da verdadero, el programa continúa normalmente. Si la evaluación de la expresión da falso, el programa lanza un tipo especial de excepción llamado `AssertionError` que hace que el programa termine si nadie lo atrapa.

Por ejemplo: `assert palabra != null;`

assert expresión1 : expresión2;

 Este caso es una variante del anterior, en el cual la expresión2 debe ser de tipo cadena de caracteres. La única diferencia es que al lanzar la excepción con el error, se le asocia el resultado de evaluar la expresión2.

 Por ejemplo: `assert palabra != null : "palabra inválida";`

Utilizar esta instrucción tiene varias ventajas, entre las cuales está que Java permite activar y desactivar la verificación de las aserciones de manera muy sencilla, de tal forma que cuando esté desactivada la verificación (por ejemplo cuando se instala el programa en el computador del usuario después de estar seguros de que está correcto) el computador no pierde tiempo haciendo

verificaciones que en ese momento ya son inútiles. Más adelante veremos cómo activar y desactivar esta opción desde Eclipse y desde el programa .bat que lo ejecuta.

Existen muchas formas posibles de escribir los métodos para verificar el invariante de una clase. En todos los

casos de estudio del libro vamos a utilizar una manera particular de hacerlo, lo que nos va a permitir guiar metodológicamente el proceso de escribirlo y facilitar la localización de los métodos que lo hacen.

Vamos a seguir tres pasos, los cuales serán ilustrados en el ejemplo 2 usando el caso de estudio del traductor:

- (1) En cada clase que tenga un invariante que se deba verificar, escribimos un método con la siguiente firma: `private void verificarInvariante()`.

En dicho método utilizamos una vez la instrucción `assert` para cada aserción del invariante. Si la expresión es simple, la colocamos directamente en la instrucción. Si es compleja, desarrollamos un método privado de tipo lógico que haga la verificación.

- (2) Al final de cada constructor y al final de cada método modificador agregamos la llamada al método `verificarInvariante()`, ya que son ellos los únicos que cambian el estado del objeto.

Ejemplo 2



Objetivo: Implementar los métodos para verificar el invariante de las clases del caso de estudio.

En este ejemplo mostramos la manera de utilizar la instrucción `assert`.

```
public class Traducion
{
    ...
    // -----
    // Invariante
    // -----
    private boolean palabraEsValida()
    {
        return palabra != null && !palabra.equals( "" );
    }

    private boolean traducionEsValida()
    {
        return traducion != null && !traducion.equals( "" );
    }

    private void verificarInvariante()
    {
        assert palabraEsValida() : "La palabra es inválida";
        assert traducionEsValida() : "La traducción es inválida";
    }
}
```

Además del método `verificarInvariante()` hay dos métodos privados: uno para verificar que la palabra sea válida y el otro para hacer lo mismo con la traducción.

Estos métodos deberían quedar en la parte de abajo de la clase, en una sección especial que indique que son utilizados para verificar el invariante.

La ventaja de separar el cálculo del invariante en métodos privados es que todos resultan suficientemente simples, como para evitar cometer errores.

```

public class Traductor
{
    ...

    // -----
    // Invariante
    // -----

    private void verificarInvariante( )
    {

        assert espanolIngles != null :
            "Diccionario español-inglés sin inicializar";
        assert espanolFrances != null :
            "Diccionario español-francés sin inicializar";
        assert espanolItaliano != null :
            "Diccionario español-italiano sin inicializar";

        assert !hayPalabrasRepetidas( INGLES ) :
            "Palabras repetidas en el diccionario de inglés";
        assert !hayPalabrasRepetidas( FRANCES ) :
            "Palabras repetidas en el diccionario de francés";
        assert !hayPalabrasRepetidas( ITALIANO ) :
            "Palabras repetidas en el diccionario de italiano";

        assert !hayTraduccionesRepetidas( INGLES ) :
            "Traducciones repetidas en el diccionario de inglés";
        assert !hayTraduccionesRepetidas( FRANCES ) :
            "Traducciones repetidas en el diccionario de francés";
        assert !hayTraduccionesRepetidas( ITALIANO ) :
            "Traducciones repetidas en el diccionario de italiano";
    }
}

```

 Dado que el invariante de esta clase es un poco más complejo, vamos a ir por partes. Inicialmente vamos a desarrollar el método que verifica el invariante y luego los métodos auxiliares que se necesitan. Estos últimos aparecerán a continuación, como parte de la tarea 3.

 Para el desarrollo de estos métodos vamos a privilegiar la claridad sobre la eficiencia, ya que es muy importante que los métodos sean correctos y además sabemos que nunca se van a ejecutar cuando el programa sea entregado al usuario.

Tarea 3

Objetivo: Escribir los métodos auxiliares para verificar el invariante de la clase Traductor.

Escriba los métodos cuyo contrato se especifica a continuación. Suponga que en la clase Traducción existen los métodos `darPalabra()` y `darTraducción()`, que retornan respectivamente la palabra y la traducción de un objeto de esa clase.

```
/**  
 * Indica si hay palabras repetidas en el diccionario del idioma dado.  
 * @param idTrad Idioma del diccionario. idTrad pertenece a {FRANCES, INGLES, ITALIANO}  
 * @return true si hay al menos una palabra repetida o false en caso contrario.  
 */  
private boolean hayPalabrasRepetidas( int idTrad )  
{  
  
}  
  
/**  
 * Indica si hay palabras con la misma traducción en el diccionario del idioma dado.  
 * @param idTrad Idioma del diccionario. idTrad pertenece a {FRANCES, INGLES, ITALIANO}  
 * @return true si hay dos palabras con la misma traducción o false en caso contrario.  
 */  
private boolean hayTraduccionesRepetidas( int idTrad )  
{  
  
}
```

- ¿Cuándo vale la pena verificar el invariante en ejecución?

El invariante debe ser visto como una herramienta para el programador y no como una carga adicional en el proceso de desarrollo. Si los métodos necesarios para calcular el invariante son demasiado complejos, se debe buscar una solución de compromiso y verificar únicamente una parte de éste. Hay que tratar de balancear la ganancia de calcularlo contra el esfuerzo de hacerlo, y buscar un punto intermedio.



Se debe tener mucho cuidado en el desarrollo de los métodos que calculan el invariante, porque si tienen algún efecto de borde (modifican algo del estado del objeto) pueden ser una fuente de inestabilidad en los programas.

- ¿Cómo usar el invariante como una herramienta de desarrollo?

La verificación del invariante le permite al programador estar seguro, en todo momento de la ejecución, de que la información que maneja en el modelo del mundo no ha perdido consistencia. Con esa certeza, la programación se simplifica, puesto que, en caso de un error en un método, sabemos que el problema es local y podemos concentrar allí la búsqueda del problema, ya que sabemos que cuando comenzó el método, el invariante se cumplía. Esta posibilidad de focalizar el desarrollo y la

depuración en un sólo método nos permite ir construyendo las clases de manera incremental.

Cuando durante la verificación del invariante se detecta un error, la ejecución del programa se detiene, permitiéndonos así detectar el método defectuoso. Este mecanismo, por supuesto, debe estar desactivado cuando se le entrega el programa al cliente.



En Eclipse, para poder utilizar la instrucción `assert`, asegúrese de que el compilador de Java seleccionado corresponda a una versión igual o posterior a la 1.4. Usualmente eso es así por defecto, pero si ve que el compilador informa de un error en todas las instrucciones `assert` del programa, trate de cambiar en las propiedades del proyecto el compilador seleccionado.

Por defecto, durante la ejecución de un programa no se verifican las instrucciones `assert`. Para hacer que la máquina virtual de Java las ejecute, se debe incluir en la llamada del programa la opción `-ea` (`-enableassertions`). En Eclipse esto se logra agregando en la ventana de ejecución de un programa, en la parte de parámetros para la máquina virtual (`VM arguments`), la opción `-ea`. Es posible seleccionar las clases o paquetes para los cuales se quieren ejecutar las aserciones, usando la sintaxis mostrada a continuación:

`-ea`

`-ea:uniandes.cupi2.traductor.mundo.Traducion`

`-ea:uniandes.cupi2.traductor.mundo...`

Las aserciones de todas las clases del programa serán ejecutadas.

Se activan únicamente las aserciones de la clase Traducion. Las instrucciones assert de las demás clases del proyecto no son ejecutadas.

Con esta opción, en la llamada de un programa, se ejecutan las instrucciones assert de todas las clases del paquete uniandes.cupi2.traductor.mundo (en nuestro caso las clases Traductor y Traducion).

```
-ea -da:uniandes.cupi2.traductor.mundo.Traductor
```

- Las aserciones de todas las clases del programa serán ejecutadas, con excepción de la clase Traductor. La opción `-da` (`-disableassertions`) desactiva la ejecución de la instrucción assert en una clase o paquete particular.

Fíjese que incluso si apagamos la ejecución de las aserciones, la llamada del método `verificarInvariant()` se va a seguir haciendo, aunque en su interior no haya ninguna instrucción que se ejecute. La decisión de crear este método para encapsular allí toda la verificación del invariante la tomamos para evitar llenar todos los métodos modificadores de la clase con la misma secuencia de instrucciones `assert`. Puede ser un poco menos eficiente, pero mucho más claro y fácil de mantener.

3.5. Asignación de Responsabilidades e Implementación de Métodos



La asignación de responsabilidades a las clases (o sea, qué métodos debe tener cada clase y qué contrato implementa cada uno de ellos) es una de las etapas críticas en el proceso de diseño. Para hacerla, se debe seguir una secuencia de tareas que se pueden resumir de la siguiente manera: (1) aplicar la técnica del experto o la técnica de descomposición de requerimientos para identificar los servicios que debe ofrecer cada clase, (2) diseñar la firma de los métodos necesarios para satisfacer los servicios esperados, (3) escribir el contrato de los métodos, identificando sus precondiciones y postcondiciones, y (4) hacer las declaraciones en Java de los métodos.

Ejemplo 3



Objetivo: Entender la asignación de responsabilidades y los contratos de los métodos en el caso de estudio.

En este ejemplo podemos ver los contratos de los métodos y la explicación de la manera como se asignó cada responsabilidad a cada uno de ellos. En algunos casos aparece el código que implementa el método, para ilustrar los puntos en los cuales se debe verificar el invariante de las clases.

Para estudiar el código completo de las clases, debe consultar el CD que acompaña al libro.

```
public class Traduccion
{
    // -----
    // Atributos
    // -----
    private String palabra;
    private String traduccion;
```

- Comenzamos por la clase Traducción, que es la más simple. Sólo tiene dos atributos: uno con la palabra y otro con su traducción en algún otro idioma.

```
/**
 * Crea la traducción de una palabra.
 * post: Se creó la traducción de la palabra especificada.
 * @param pal es la palabra. pal != null y pal != ""
 * @param trad es la traducción. trad != null, trad != ""
 */
public Traducion( String pal, String trad )
{
    palabra = pal;
    traducion = trad;

    // Verifica el invariante
    verificarInvariante( );
}
```

 La primera responsabilidad de la clase es crear instancias correctas de la misma.

 En la precondición exigimos que tanto la palabra como la traducción sean válidas. No estamos obligados entonces a hacer ninguna verificación.

 Al final validamos que el invariante de la clase se cumpla.

```
/**
 * Retorna la palabra base de la traducción.
 * @return La palabra base de la traducción
 */
public String darPalabra( )
{
    return palabra;
}
```

 La técnica del experto nos pide que tengamos un método para retornar la información que manejamos y que es necesaria en otras clases

```
/**
 * Retorna la traducción de la palabra.
 * @return La traducción de la palabra
 */
public String darTraducion( )
{
    return traducion;
}
```

 Por la técnica del experto tenemos también este método.

 No hay métodos de modificación en esta clase, porque no hay ningún requerimiento funcional en el caso de estudio que lo requiera.

```

public class Traductor
{
    // -----
    // Constantes
    // -----


    public final static int INGLES = 0;
    public final static int FRANCES = 1;
    public final static int ITALIANO = 2;
    public final static int ESPANOL = 3;

    // -----
    // Atributos
    // -----
}

```

```

private ArrayList espanolIngles;
private ArrayList espanolFrances;
private ArrayList espanolItaliano;

```

```

/**
 * Crea el traductor con los diccionarios vacíos.
 * post: Se creó el traductor con los diccionarios vacíos.
 */
public Traductor( )
{
    espanolIngles = new ArrayList( );
    espanolFrances = new ArrayList( );
    espanolItaliano = new ArrayList( );

    // Verifica el invariante
    verificarInvariant();
}

```

Ahora pasamos a la clase Traductor, la cual declara cuatro constantes para representar los posibles idiomas que maneja.

Como atributos tiene los tres diccionarios, que son vectores (ArrayList) que contienen objetos de la clase Traducción.

El constructor debe crear instancias válidas de la clase (deben cumplir el invariante a la salida del constructor), que satisfagan también la postcondición.

Por eso el constructor debe asumir la responsabilidad de crear los vectores que representan los diccionarios (el invariante dice que no pueden ser nulos y la postcondición dice que deben estar vacíos).

- Usando la técnica de descomposición de requerimientos, encontramos que la clase Traductor debe asumir tres grandes responsabilidades. La primera, permitir el ingreso de nuevas palabras. La segunda, buscar la traducción de una palabra. La tercera, calcular el número de palabras que hay en cada diccionario. A continuación se muestran los contratos de estos tres métodos.
- Decidimos no lanzar excepciones desde los métodos, sino retornar valores lógicos que indican si la operación se llevó a cabo sin problemas.
- Para la implementación de estos métodos es conveniente utilizar la técnica de dividir y conquistar (estudiada en cursos anteriores) y crear un conjunto de métodos privados que resuelvan partes del problema.

```
/**
 * Agrega al diccionario una traducción de una palabra en español a un idioma dado.
 * post: La traducción fue adicionada al diccionario del idioma especificado.
 * @param pal es la palabra. pal != null y pal != ""
 * @param trad es la traducción. trad != null y trad != ""
 * @param idDestino es el idioma destino. idDestino pertenece a {FRANCES, INGLES, ITALIANO}
 * @return true si la palabra pudo ser adicionada al diccionario o false en caso contrario.
 */
public boolean agregarTraducion( String pal, String trad, int idDestino )
{
}
```

```
/**
 * Dada una palabra, el idioma en el que está y el idioma al que se quiere traducir, éste
 * método retorna la traducción correspondiente.
 * @param pal es la palabra. pal != null
 * @param idOrigen Idioma de origen. idOrigen pertenece a {FRANCES, INGLES, ITALIANO, ESPANOL}
 * @param idDestino Idioma destino. idDestino pertenece a {FRANCES, INGLES, ITALIANO, ESPANOL}
 * @return Traducción de la palabra en el idioma destino. Si no existe, retorna null.
 */
public Traducion traducir( String pal, int idOrigen, int idDestino )
{
}
```

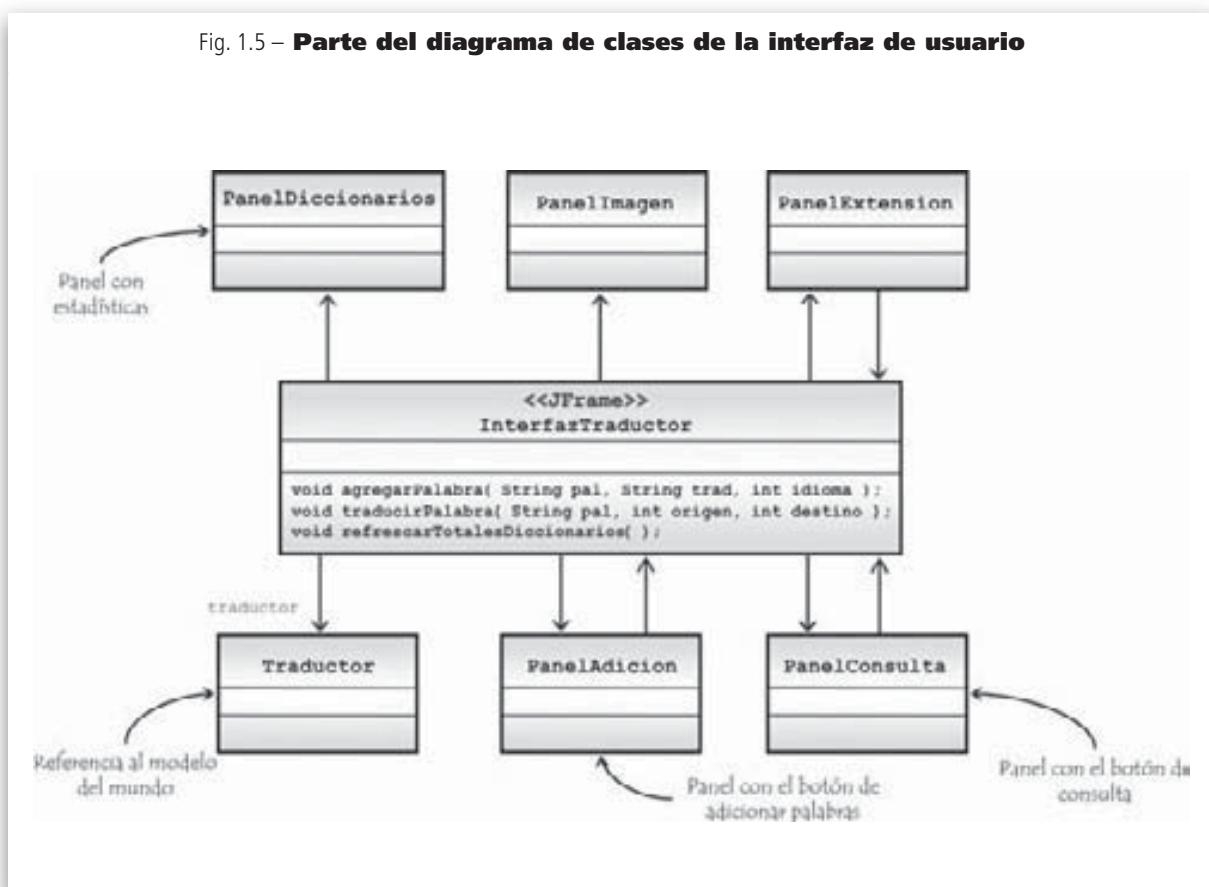
```
/**
 * Retorna el número de palabras del diccionario de un idioma dado.
 * @param idDestino Idioma destino. idDestino pertenece a {FRANCES, INGLES, FRANCES}
 * @return Número de palabras en el diccionario
 */
public int darTotalPalabrasTraducidas( int idDestino )
{
}
```

Después de haber repartido las responsabilidades entre las clases del modelo del mundo y de haber escrito los respectivos contratos, sólo nos queda escribir un método, en la clase principal de la interfaz de usuario, por cada requerimiento funcional, de manera que todos los paneles activos deleguen en esos métodos las solicitudes hechas en ejecución por los usuarios.

En la figura 1.5 aparece una parte del diagrama

de clases de la interfaz de usuario. Allí se muestra la relación entre la interfaz y el modelo del mundo (asociación hacia la clase `Traductor`), además de cinco paneles: tres activos (con botones para el usuario) y dos pasivos (que sólo despliegan información). También se puede apreciar la firma de los tres métodos de la ventana principal, mediante los cuales se van a ejecutar los requerimientos funcionales del programa.

Fig. 1.5 – **Parte del diagrama de clases de la interfaz de usuario**



3.6. Pruebas Unitarias Automáticas

Después de haber dividido las responsabilidades entre todas las clases y de haber definido los contratos de los métodos comienza el proceso de implementación y pruebas, en el cual debemos escribir el cuerpo de

los métodos y verificar que cumplan efectivamente su contrato. La manera de probar el código que vayamos desarrollando es el tema de esta sección. En ella comenzaremos con una visión abstracta del problema y terminaremos mostrando cómo utilizar la herramienta `JUnit` para construir las pruebas unitarias automáticas de un programa escrito en Java.

Por ahora comenzaremos con los elementos básicos de dicho proceso y lo iremos extendiendo en los niveles posteriores, a medida que vayamos estudiando nuevos temas.

3.6.1. Introducción y Motivación

¿Por qué no esperar a terminar de escribir todo el programa antes de comenzar a probarlo? Esa es la pregunta que puede surgir en este momento y la respuesta es: tiempo y garantía de calidad. Uno de los problemas más complicados del proceso de corrección de un error es localizarlo. Si ya tenemos miles de líneas de código implementadas y detectamos un defecto en el funcionamiento del programa, ¿por dónde comenzarlo a buscar? ¿Cómo saber que no es la suma de muchos errores en distintas partes del programa? ¿Qué hacer si el programa es inestable, en el sentido de que, para las mismas entradas, a veces funciona y a veces no? E incluso, si en las pruebas no detectamos ningún problema, ¿cómo podemos garantizar que ya probamos todos los casos posibles y así garantizar que el programa es correcto?

Durante muchos años las pruebas de los programas se hicieron en una etapa posterior a la etapa de desarrollo, cuando el programa ya estaba terminado, y se basaban en guiones que ejecutaban las personas sobre la interfaz de usuario para verificar que los requerimientos funcionales eran satisfechos por el programa. Sin embargo, a medida que los problemas han ido creciendo y que los contextos en los que se usan los programas se han vuelto críticos (equipos médicos o de control), ese enfoque ha demostrado ser insuficiente. Debemos encontrar una manera más efectiva de probar los programas que desarrollemos.

Es tratando de encontrar una respuesta a los problemas antes mencionados que aparecieron las **pruebas automáticas**. En ellas, el encargado de ejecutar las pruebas y verificar que los resultados sean

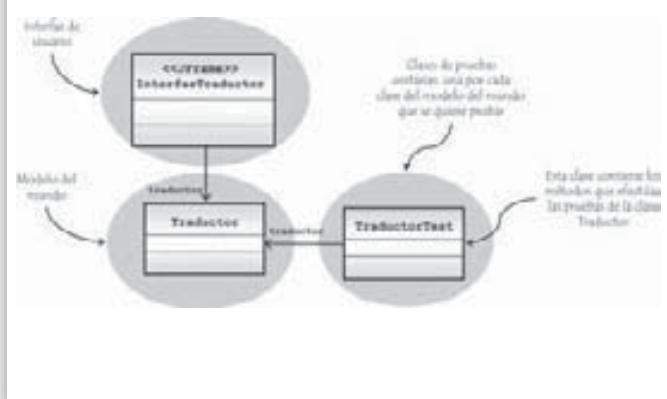
los esperados es un programa, el cual trabaja sobre un conjunto predefinido de escenarios y casos. Este enfoque tiene la ventaja de que el mismo proceso de pruebas se puede repetir múltiples veces (si algo falla podemos volver a comenzar desde cero a bajo costo), pero sigue teniendo el problema de la localización de los errores. Surgen entonces las **pruebas unitarias automáticas**, en las cuales no nos lanzamos a probar un programa completo, sino a probar individualmente cada una de sus clases. La hipótesis es que si todas las clases cumplen con sus compromisos y contratos, el programa debe funcionar correctamente. Y, si algo falla, sabremos inmediatamente qué clase fue, y cuál de sus métodos no está cumpliendo el contrato para el cual fue construido.

Aunque no se puede decir que las pruebas unitarias automáticas sean la solución a todos los problemas de corrección, sí es claro que son una excelente herramienta que permite mejorar la calidad de los programas y facilitar el proceso de desarrollo. Por eso es importante no ver los programas que hacen las pruebas como un producto adicional o como una carga extra, sino como una ayuda al proceso de construcción de programas.

3.6.2. Visión Conceptual de las Pruebas

Para facilitar su evolución y desarrollo, vamos a implementar una clase de prueba por cada una de las clases del modelo del mundo que estemos interesados en probar, la cual tendrá siempre una asociación hacia la clase objeto de la verificación. En la figura 1.6 damos una idea global de la arquitectura que va a tener el programa y la manera como se conectan la interfaz de usuario y las clases de prueba al modelo del mundo. Allí se puede apreciar que cada clase de prueba (`TraductorTest`) tiene una relación directa con la clase que quiere probar (`Traductor`), sin pasar por las clases que implementan la interfaz de usuario (`InterfazTraductor`).

Fig. 1.6 – **Las clases de prueba en la arquitectura de un programa**

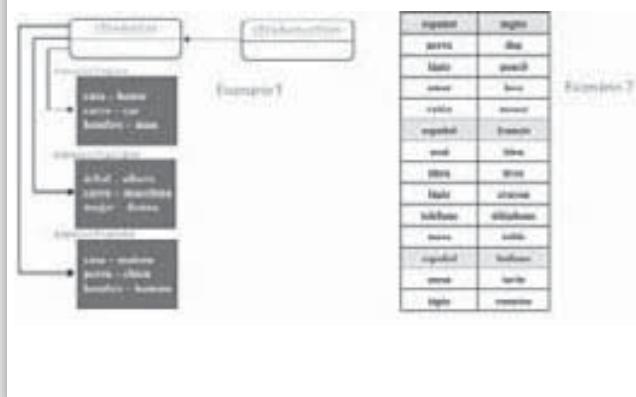


Las clases de prueba las vamos a almacenar en un paquete distinto (en el caso de estudio están en el paquete `uniandes.cupi2.traductor.test`) para distinguirlas de las clases del programa, y físicamente las vamos a colocar en un directorio aparte (`test\source`) del que usamos para el modelo del mundo y la interfaz (`source`).

Hay dos componentes fundamentales que se deben tener en cuenta en el momento de construir una clase de prueba: los escenarios y los casos de prueba. Un **escenario** es un objeto de la clase que se quiere probar, el cual tiene un estado conocido por nosotros (sabemos el valor de cada uno de los atributos y conocemos el estado de los objetos con los cuales está asociado). Un escenario se puede representar con un diagrama de objetos, tal como se mostró en la figura 1.4 para el caso del traductor, aunque más adelante introduciremos una sintaxis un poco más simple. La primera responsabilidad de una clase de prueba es saber construir distintos escenarios sobre los cuales podamos probar el correcto funcionamiento de los métodos. Vamos a escribir un método por cada escenario que queramos construir. Un escenario siempre debe cumplir con el invariante de la clase.

En la figura 1.7 mostramos dos escenarios distintos para las pruebas de la clase `Traductor`. Cualquier sintaxis es válida para expresar un escenario, en la medida en que contenga suficiente información para establecer el estado del objeto. Es importante construir los escenarios de manera que representen todas las situaciones posibles en las que se pueda encontrar el modelo del mundo. En el caso del traductor, por ejemplo, sería conveniente definir un escenario en el cual los diccionarios estén vacíos.

Fig. 1.7 – **Dos escenarios posibles para las pruebas de la clase Traductor**



Una vez definidos los escenarios e implementados los métodos que son capaces de construirlos, pasamos a definir los casos de prueba, asociados con cada uno de los métodos de la clase que queremos probar. Un caso de prueba asocia un escenario, un método y unos valores de entrada para los parámetros del método con un resultado, tal como se muestra en el ejemplo 4. Es necesario que tanto el escenario como los valores de los parámetros cumplan la precondition del método que se quiere probar.

Ejemplo 4

Objetivo: Ilustrar la idea de un caso de prueba para un método.

En este ejemplo se muestran tres casos de prueba para el método `agregarTraduccion()` de la clase `Traductor`.

Clase	Método	Escenario	Valores de entrada	Resultado
Traductor	<code>agregarTraduccion()</code>	Fig. 1.7 Escenario1	pal = azul trad = bleu idDestino = FRANCES	Verdadero. La palabra y la traducción se agregaron al diccionario español-francés.
Traductor	<code>agregarTraduccion()</code>	Fig. 1.7 Escenario1	pal = casa trad = maison idDestino = FRANCES	Falso. No se ha modificado el diccionario español-francés.
Traductor	<code>agregarTraduccion()</code>	Fig. 1.7 Escenario1	pal = coche trad = car idDestino = INGLES	Falso. No se ha modificado el diccionario español-inglés.

Es usual que los casos de prueba se planteen de manera incremental y con un propósito compartido y definido, más que como un conjunto independiente de verifica-

ciones al azar. En el ejemplo 5 mostramos la información adicional que debe hacer parte de la documentación de un caso de prueba.

Ejemplo 5

Objetivo: Ilustrar la idea de un caso de prueba para un método.

En este ejemplo se documentan algunos casos de prueba para el método `agregarTraduccion()` de la clase `Traductor`.

Prueba No. 1	Objetivo: Probar que el método es capaz de agregar palabras y traducciones válidas a cualquiera de los diccionarios.			
Traductor	<code>agregarTraduccion()</code>	Fig. 1.7 Escenario1	pal = azul trad = bleu idDestino = FRANCES	Verdadero. La palabra y la traducción se agregaron al diccionario español-francés.
Traductor	<code>agregarTraduccion()</code>	-	pal = azul trad = blue idDestino = INGLES	Verdadero. La palabra y la traducción se agregaron al diccionario español-inglés.
Traductor	<code>agregarTraduccion()</code>	-	pal = mesa trad = tavlo idDestino = ITALIANO	Verdadero. La palabra y la traducción se agregaron al diccionario español-italiano.

Prueba No. 2	Objetivo: Probar que el método no permite agregar palabras repetidas en español a sus diccionarios.			
Traductor	agregarTraducion()	Fig. 1.7 Escenario1	pal = casa trad = maison idDestino = FRANCES	Falso. No se ha modificado el diccionario español-francés, porque la palabra "casa" ya existe en español en ese diccionario.
Traductor	agregarTraducion()	-	pal = casa trad = house idDestino = INGLES	Falso. No se ha modificado el diccionario español-inglés, porque la palabra "casa" ya existe en español en ese diccionario.
Prueba No. 3	Objetivo: Probar que el método no permite agregar traducciones repetidas a sus diccionarios.			
Traductor	agregarTraducion()	Fig. 1.7 Escenario1	pal = coche trad = car idDestino = INGLES	Falso. No se ha modificado el diccionario español-inglés, porque la traducción "car" ya está asignada a otra palabra en el diccionario.

Ya podemos comenzar a definir las pruebas para nuestro caso de estudio, usando para eso uno de los escenarios definidos anteriormente.



Tarea 4

Objetivo: Definir las pruebas para el método `traducir()` de la clase `Traductor`.

(1) Lea detenidamente el contrato del método `traducir()` de la clase `Traductor` e identifique los valores de entrada y el resultado esperado; (2) para las tres pruebas cuyos objetivos se plantean más adelante, proponga un conjunto de casos de prueba, usando el segundo escenario definido en la figura 1.7.

Prueba No. 1	Objetivo: Probar que el método es capaz de encontrar correctamente la traducción de una palabra del español a cualquiera de los otros idiomas.			
Clase	Método	Escenario	Valores de entrada	Resultado
Traductor	Traducir()	Fig. 1.7 Escenario2		
Traductor	Traducir()	-		
Traductor	Traducir()	-		

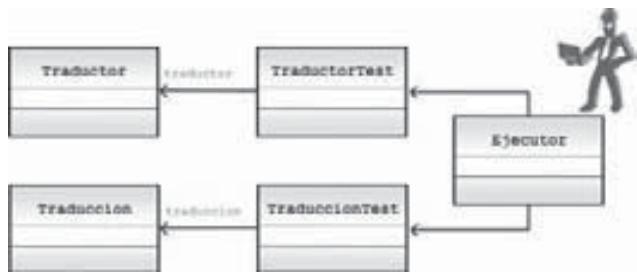
Prueba No. 2		Objetivo: Probar que el método es capaz de encontrar correctamente la traducción de una palabra de un idioma distinto al español a cualquiera de los otros idiomas.		
Traductor	Traducir()	Fig. 1.7 Escenario2		
Traductor	Traducir()	-		
Traductor	Traducir()	-		
Prueba No. 3		Objetivo: Probar que el método no encuentra la traducción para palabras que no están en el diccionario.		
Traductor	Traducir()	Fig. 1.7 Escenario2		
Traductor	Traducir()	-		
Traductor	Traducir()	-		



Una clase de prueba está asociada a una clase del modelo del mundo. Tiene dos tipos de métodos: unos para crear escenarios y otros para verificar que cada uno de los métodos cumpla con su contrato. Las pruebas se diseñan considerando las distintas situaciones a las que se debe enfrentar el método.

Por último, necesitamos a alguien que tome cada clase de prueba que nosotros señalemos, ejecute las verificaciones que ella contiene y genere un reporte con los resultados. Dicha clase la llamaremos el **ejecutor de pruebas** y, a menos que utilicemos alguna herramienta que ya tenga uno (como es el caso de JUnit), debemos desarrollarlo. La idea del ejecutor de pruebas se ilustra en la figura 1.8. Allí aparecen las clases de prueba para el traductor.

Fig. 1.8 – **El ejecutor de pruebas en el caso de estudio**



3.6.3. JUnit: Un Framework de Construcción de Pruebas

Aunque es posible construir las pruebas directamente en Java, es mucho más sencillo si utilizamos un

conjunto de clases que han sido diseñadas con el objeto de ayudarnos en esta labor. Las clases que nosotros vamos a utilizar en este libro hacen parte del *framework JUnit*, el cual cuenta también con un ejecutor de pruebas. La única condición, para poder aprovechar todas las ventajas de esta herramienta, es cumplir con ciertas reglas y convenciones que ella impone, y que veremos en esta sección:

- Las clases de prueba deben declararse con un `extends TestCase` en su encabezado.
- Se deben importar las clases del paquete `junit.framework`.
- Los métodos que implementan las pruebas deben ser de tipo `void` y su nombre debe comenzar por el prefijo “`test`”.

Ejemplo 6



Objetivo: Mostrar el esqueleto de una clase de prueba usando JUnit.

En este ejemplo se presenta el esqueleto de la clase `TraductorTest` que contiene las pruebas de la clase `Traductor`.

```

package uniandes.cupi2.traductor.test;

import junit.framework.TestCase;
import uniandes.cupi2.traductor.mundo.Traductor;

public class TraductorTest extends TestCase
{
    // -----
    // Atributos
    //

    private Traductor traductor;

    public void testAgregarTraduccion( )
    {
        ...
    }
}
  
```

■ Se debe importar la clase `TestCase` de JUnit, lo mismo que la clase del modelo del mundo que queremos probar. Se debe importar porque esta clase está localizada en un paquete distinto.

■ La clase la llamamos `TraductorTest` para que sea evidente la clase del mundo que pretende probar.

■ En el encabezado incluimos la declaración “`extends TestCase`” para indicar que es una clase de prueba que va a utilizar los métodos definidos en la clase `TestCase` que hace parte de JUnit.

■ Se declara una asociación hacia la clase que se quiere probar. Dicha asociación será utilizada para referenciar en todo momento el escenario de prueba.

■ En este ejemplo sólo aparece un método de prueba. Es de tipo `void` y su nombre comienza por el prefijo “`test`”. El nombre del método debe dar una idea de la prueba que se quiere hacer con él.

Los métodos para construir escenarios los vamos a definir como privados y van a comenzar por el prefijo “`setupEscenario`” y un número. Estos métodos deben encargarse de crear una instancia del mode-

lo del mundo e invocar los métodos necesarios para llevarlo al estado que se definió en su diseño. Esto se muestra en el ejemplo 7.

Ejemplo 7



Objetivo: Mostrar el método de creación de un escenario.

En este ejemplo se presenta el método de la clase `TraductorTest` encargado de crear el primer escenario definido en la figura 1.7.

```
public class TraductorTest extends TestCase
{
    private Traductor traductor;

    private void setupEscenario1( )
    {
        traductor = new Traductor( );

        traductor.agregarTraduccion( "casa", "house", Traductor.INGLES );
        traductor.agregarTraduccion( "carro", "car", Traductor.INGLES );
        traductor.agregarTraduccion( "hombre", "man", Traductor.INGLES );

        traductor.agregarTraduccion( "casa", "maison", Traductor.FRANCES );
        traductor.agregarTraduccion( "perro", "chien", Traductor.FRANCES );
        traductor.agregarTraduccion( "hombre", "homme", Traductor.FRANCES );

        traductor.agregarTraduccion( "árbol", "albero", Traductor.ITALIANO );
        traductor.agregarTraduccion( "carro", "macchina", Traductor.ITALIANO );
        traductor.agregarTraduccion( "mujer", "donna", Traductor.ITALIANO );
    }

    ...
}
```



El objetivo del método es crear el escenario 1 definido en la figura 1.7, y dejarlo en el atributo “`traductor`”.

Para escribir los métodos de prueba, la clase `TestCase` nos provee las siguientes funcionalidades:

- `assertTrue(mensaje, condición)`: Este método evalúa la condición. Si es verdadera, continúa normalmente la ejecución. Si es falsa, lanza la excepción `AssertionFailedError` asociándole el mensaje que llega como parámetro. Dicho mensaje será utilizado por el ejecutor de pruebas para generar el reporte del error. El mensaje es opcional en este método y en todos los que siguen.
- `assertFalse(mensaje, condición)`: Funciona de manera similar al anterior, pero lanza la excepción si la condición no es falsa.
- `assertNull(mensaje, objeto)`: Este método lanza la excepción `AssertionFailedError` si el objeto que llega como parámetro no es `null`.
- `assertNotNull(mensaje, objeto)`: Este método lanza la excepción `AssertionFailedError` si el objeto que llega como parámetro es `null`.

- `assertEquals(mensaje, esperado, actual)`: Este método verifica si `esperado == actual`. Si eso es cierto, continúa normalmente la ejecución. Si es falso, lanza la excepción `AssertionFailedError`. Los parámetros `esperado` y `actual` deben ser de tipo `int`, `long` o `char`.
- `assertEquals(mensaje, esperado, actual, delta)`: Este método se utiliza para comparar valores reales, en los cuales no es posible verificar igualdad total, debido a las aproximaciones que hacen los computadores. En ese caso verifica si `esperado == actual`, pero les da un `delta` de error (si no son iguales, la diferencia entre los dos no puede superar este valor).



Tenga cuidado de no confundir los métodos anteriores con la instrucción `assert` de Java. Éstos son métodos de la clase `TestCase` de JUnit, que podemos utilizar gracias a la declaración `"extends"` que usamos. Estos métodos sólo se pueden usar dentro de las clases de prueba.

Ejemplo 8



Objetivo: Mostrar un método de prueba en el caso de estudio.

En este ejemplo se presenta el método de la clase `TraductorTest` encargado de probar que las palabras se agregan de manera correcta a los diccionarios, si no hay ningún conflicto.

```
public class TraductorTest extends TestCase
{
    private Traductor traductor;

    private void setupEscenario3()
    {
        traductor = new Traductor();
    }
}
```

■ Vamos a trabajar sobre un tercer escenario (diccionarios vacíos), puesto que sería un error partir de un escenario que necesita el método que vamos a probar para poderse construir.

■ El método va a probar el caso en el cual sí es posible agregar palabras a cada uno de los diccionarios.

```
public void testAgregarTraducion()
{
    setupEscenario3();

    // Verificamos que el método de inserción retorne siempre true
    assertTrue( traductor.agregarTraducion( "perro", "dog", Traductor.INGLES ) );
    assertTrue( traductor.agregarTraducion( "blanco", "white", Traductor.INGLES ) );
    assertTrue( traductor.agregarTraducion( "casa", "house", Traductor.INGLES ) );
}
```

```

assertTrue( traductor.agregarTraducion( "casa", "maison", Traductor.FRANCES ) );
assertTrue( traductor.agregarTraducion( "libro", "livre", Traductor.FRANCES ) );
assertTrue( traductor.agregarTraducion( "azul", "bleu", Traductor.FRANCES ) );

assertTrue( traductor.agregarTraducion( "mesa", "tavlo", Traductor.ITALIANO ) );
assertTrue( traductor.agregarTraducion( "hoja", "foglia", Traductor.ITALIANO ) );
assertTrue( traductor.agregarTraducion( "revista", "rivista", Traductor.ITALIANO ) );

// En cada diccionario debe haber tres palabras con sus traducciones
assertEquals( 3, traductor.darTotalPalabrasTraducidas( Traductor.INGLES ) );

assertEquals( 3, traductor.darTotalPalabrasTraducidas( Traductor.FRANCES ) );
assertEquals( 3, traductor.darTotalPalabrasTraducidas( Traductor.ITALIANO ) );

// Realiza la búsqueda de traducciones de español a inglés
Traducion traducion;

traducion = traductor.traducir( "blanco", Traductor.ESPAÑOL, Traductor.INGLES );
assertEquals( "white", traducion.darTraducion( ) );

traducion = traductor.traducir( "perro", Traductor.ESPAÑOL, Traductor.INGLES );
assertEquals( "dog", traducion.darTraducion( ) );

traducion = traductor.traducir( "casa", Traductor.ESPAÑOL, Traductor.INGLES );
assertEquals( "house", traducion.darTraducion( ) );

// Realiza la búsqueda de traducciones de español a francés
traducion = traductor.traducir( "azul", Traductor.ESPAÑOL, Traductor.FRANCES );
assertEquals( "bleu", traducion.darTraducion( ) );

traducion = traductor.traducir( "libro", Traductor.ESPAÑOL, Traductor.FRANCES );
assertEquals( "livre", traducion.darTraducion( ) );

traducion = traductor.traducir( "casa", Traductor.ESPAÑOL, Traductor.FRANCES );
assertEquals( "maison", traducion.darTraducion( ) );

// Realiza la búsqueda de traducciones de español a italiano
traducion = traductor.traducir( "mesa", Traductor.ESPAÑOL, Traductor.ITALIANO );
assertEquals( "tavlo", traducion.darTraducion( ) );

traducion = traductor.traducir( "hoja", Traductor.ESPAÑOL, Traductor.ITALIANO );
assertEquals( "foglia", traducion.darTraducion( ) );

traducion = traductor.traducir( "revista", Traductor.ESPAÑOL, Traductor.ITALIANO );
assertEquals( "rivista", traducion.darTraducion( ) );
}
}

```

En el método anterior pudimos observar algo que va a ser recurrente en la construcción de las pruebas: los métodos se necesitan entre sí para probarse. La única manera de saber si la palabra y su traducción se agregaron efecti-

vamente en un diccionario es buscando allí la palabra y verificando que el programa le retorna la traducción que le acabamos de dar. Esto nos va a suceder sobre todo cuando vayamos a probar los métodos modificadores.

Tarea 5

Objetivo: Escribir el método para probar la búsqueda de palabras en el traductor.

Utilice el escenario 1 (figura 1.7), definido anteriormente, para probar que el método de traducción funciona de manera efectiva. Utilice el diseño de la prueba que hizo en la tarea 4. Separe el código en secciones, en donde sea claro el objetivo de cada parte del método.

Ahora que ya sabemos escribir las pruebas, sólo nos resta ejecutarlas, utilizando para esto el ejecutor que nos provee JUnit. La clase del *framework* que implementa el ejecutor se llama `junit.swingui.TestRunner` y le debemos pasar como parámetro la clase (el nombre completo con el paquete en el que se encuentra) que contiene las pruebas que vamos a ejecutar (`uniandes.cupi2.traductor.test.TraductorTest`).



Tarea 6

Objetivo: Ejecutar las pruebas del caso de estudio y localizar en los respectivos directorios los elementos necesarios para que funcione.

Siga las instrucciones que se dan a continuación, con el fin de ejecutar las pruebas del traductor.

1. Copie del CD el ejemplo `n7_traductor.zip` en algún lugar del disco duro y descomprímalo.
2. Localice en el subdirectorio "bin" el archivo "build.bat" y ejecútelo.
3. Haga lo mismo con el archivo "buildTest.bat" que se encuentra en ese mismo directorio. Con este archivo compilamos las clases de prueba y creamos el archivo "traductorTest.jar". Vaya al directorio "test\lib" y verifique que allí se encuentra este archivo, junto con el archivo "junit.jar". En este último archivo se encuentran todas las clases de JUnit.
4. Edite en el directorio "bin" el archivo "runTest.bat" y revise la manera en que se invoca el ejecutor de JUnit. Fíjese en la manera como le decimos al ejecutor la clase que contiene las pruebas.
5. Ejecute el archivo "runTest.bat" y mire en detalle el reporte que genera JUnit, luego de haber ejecutado las pruebas. ¿Cuántos métodos de prueba ejecutó? ¿Cuántos fueron exitosos?
6. Lance Eclipse y cree el proyecto `n7_traductor`. Localice el código fuente de las pruebas, en el directorio "test\source". Edite el archivo `TraductorTest` y estudie la manera en que se crean los escenarios. ¿Cuántos escenarios crea esa clase? ¿En cuál escenario están todos los diccionarios vacíos?
7. En la clase `TraductorTest` localice los métodos de prueba. ¿Cuántos son? ¿Qué objetivo tiene cada uno de ellos?
8. Vaya a la vista del explorador de paquetes (*Package Explorer*). Localice allí el archivo `TraductorTest`. Vamos a utilizar la facilidad que nos da Eclipse para ejecutar las pruebas desde su interior. Con el botón derecho del ratón sobre el archivo, seleccione la opción **Run As \ JUnit Test**. ¿Cómo despliega Eclipse el reporte de las pruebas? Modifique algo en la clase de prueba, de manera que aparezca que una de las pruebas falla. ¿Qué información nos da JUnit para localizar el punto que generó el error? ¿Qué otra información nos suministra JUnit en el reporte?
9. En la vista del explorador de paquetes, seleccione el proyecto `n7_traductor` y con el botón derecho escoja la opción **Properties**. Allí aparece una ventana con las propiedades del proyecto. Busque en la parte izquierda la entrada llamada **Java Build Path** y haga clic sobre ella. Seleccione la pestaña llamada **Source** y verifique que aparezca el directorio `test\source` allí incluido. De esa manera le decimos a Eclipse que en ese directorio también hay código fuente. Seleccione ahora la pestaña **Libraries**. Allí aparece que el archivo `junit.jar` se encuentra en el directorio `test\lib`. De esa forma le contamos a Eclipse en dónde debe buscar otras clases ya compiladas.
10. Cierre la ventana de propiedades del proyecto y ejecute el programa. Verifique que todo funciona según la especificación. Imagine cómo sería un guión de pruebas para que un usuario pudiera verificar la correcta implementación de todos los requerimientos funcionales.
11. Vaya al CD que acompaña al libro y localice el menú de recursos. Busque el Javadoc de JUnit. Mire la documentación de las clases `TestCase` y `Assert`. Estudie los otros métodos disponibles en JUnit para la construcción de pruebas.

4. Caso de Estudio Nº 2: Un Manejador de Muestras

En este segundo caso del nivel vamos a desarrollar un programa que nos permita manejar una muestra de datos. Una muestra es una secuencia de valores enteros que se encuentran en un rango dado (por simplici-

dad vamos a suponer que el límite inferior del rango es siempre uno). Piense por ejemplo en los resultados de una encuesta, en la cual las personas califican el desempeño del presidente con un valor entre 1 y 10. O piense también en los números de la tarjeta de identidad de todos los niños nacidos en el país entre 1999 y 2001. Las operaciones que nos interesan sobre una

Fig. 1.9 – **Interfaz de usuario del manejador de muestras**



Para iniciar el programa debemos crear una muestra, usando el botón **Nueva Muestra**. Allí definimos el tamaño y el límite superior. El programa contesta creando la muestra de manera aleatoria y presentando los primeros resultados estadísticos (mínimo, máximo y promedio).

Inicialmente todos los botones que sólo funcionan sobre una muestra ordenada se encuentran desactivados.

Es el caso, por ejemplo, de la búsqueda binaria.

Luego, se debe ordenar la muestra utilizando cualquiera de los algoritmos disponibles en el programa. Después de esto se activan las opciones que no se encontraban disponibles.

En este momento se pueden utilizar todas las opciones del programa: ejecutar las búsquedas, determinar cuántos elementos de la muestra se encuentran en un rango dado, cuántas veces aparece un valor dado, cuántos valores distintos hay en la muestra o cuál es el valor que más veces se presenta en la muestra. Para todos ellos se presenta el respectivo resultado, acompañado del tiempo que le tomó al computador calcularlo.

Es posible ordenar varias veces la muestra, usando los distintos algoritmos.

muestra tienen que ver con ordenamientos y búsqueda de información, y con el tiempo que utilizan los distintos algoritmos disponibles para cumplir dichas tareas.

El programa que vamos a desarrollar debe ofrecer las siguientes opciones: (1) crear una nueva muestra de manera aleatoria, de un tamaño y con un límite superior definidos por el usuario (el usuario podría, por ejemplo, pedir una muestra de 200.000 elementos, en el rango 1 a 200.000); (2) ordenar la muestra utilizando uno de los siguientes algoritmos: inserción, selección o intercambio (burbuja), y mostrar el tiempo que gasta el computador en ejecutar dicha tarea; (3) mostrar la eficiencia de los algoritmos de búsqueda secuencial y binaria. Para esto, el programa debe calcular el tiempo que utiliza en buscar cada uno de los elementos del rango de la muestra y dividirlo por el número de elementos que buscó. Por ejemplo, si la muestra está en el rango de 1 a 200.000, el programa debe buscar cada uno de esos valores en la muestra, calcular el tiempo total del proceso y dividir por 200.000; (4) calcular el número de elementos que se

encuentran en un rango dentro de la muestra ordenada (por ejemplo, determinar cuántos valores de la muestra están entre 50.000 y 100.000), (5) calcular el número de veces que aparece un valor dado en la muestra ordenada (por ejemplo, cuántas veces aparece en la muestra el valor 30.000), (6) calcular el número de valores distintos en la muestra ordenada, (7) encontrar el valor que más veces aparece en la muestra ordenada y (8) dar información estadística de la muestra no ordenada, incluyendo el mayor valor, el menor valor y el promedio.

La interfaz de usuario del programa se muestra en la figura 1.9, en donde aparecen los resultados de la ejecución para una muestra de 200.000 datos, con valores en un rango de 1 a 200.000. Allí podemos apreciar que hay diferencias considerables de tiempo al ordenar la muestra utilizando los distintos algoritmos disponibles en el programa. También se puede ver la diferencia de tiempo que existe entre una búsqueda secuencial y una búsqueda binaria. Casi 1.000 veces más veloz la segunda que la primera.

4.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none"> ■ Implementar los algoritmos de ordenamiento por selección, inserción e intercambio para tipos simples de datos. 	<ul style="list-style-type: none"> ■ Aprender cómo funcionan esos tres algoritmos de ordenamiento.
<ul style="list-style-type: none"> ■ Implementar los algoritmos de búsqueda que se piden en el enunciado. 	<ul style="list-style-type: none"> ■ Aprender en qué consiste la búsqueda binaria. Para las demás búsquedas ya tenemos los conocimientos y habilidades necesarios.
<ul style="list-style-type: none"> ■ Generar valores aleatorios en un rango. 	<ul style="list-style-type: none"> ■ Estudiar el método <code>Math.random()</code> de Java y adaptarlo para que genere valores en un rango dado.
<ul style="list-style-type: none"> ■ Calcular el tiempo de ejecución de un método. 	<ul style="list-style-type: none"> ■ Estudiar los métodos que ofrece Java para manejo de tiempo y la manera en que se pueden utilizar para medir el tiempo de ejecución de una parte de un programa.
<ul style="list-style-type: none"> ■ Construir un programa que funcione correctamente. 	<ul style="list-style-type: none"> ■ Practicar la utilización de invariantes y la construcción de pruebas unitarias.

4.2. Comprensión de los Requerimientos

Como primer paso para la construcción del programa, debemos identificar los requerimientos funcionales y especificarlos.

Tarea 7



Objetivo: Entender el problema del caso de estudio del manejador de muestras.

(1) Lea detenidamente el enunciado del caso de estudio del manejador de muestras e (2) identifique y complete la documentación de los ocho requerimientos funcionales que allí aparecen.

Requerimiento funcional 1	Nombre	R1 – Crear una nueva muestra
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Ordenar la muestra
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Calcular el tiempo de ejecución de los algoritmos de búsqueda
	Resumen	

Requerimiento funcional 3	Entradas	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Calcular el número de elementos en un rango
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Calcular el número de veces que aparece un valor
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 6	Nombre	R6 – Calcular el número de valores distintos
	Resumen	

Requerimiento funcional 6	Entradas	
	Resultado	
Requerimiento funcional 7	Nombre	R7 – Encontrar el valor que más veces aparece
	Resumen	
Requerimiento funcional 8	Entradas	
	Resultado	
	Nombre	R8 – Dar información estadística de la muestra
	Resumen	
	Entradas	
	Resultado	

4.3. Arquitectura de la Solución

En esta sección vamos a presentar los tres diagramas que definen la arquitectura de la solución propuesta: el diagrama de clases del modelo del mundo, el diagrama de clases de la interfaz de usuario y el diagrama de clases de las pruebas unitarias. En el modelo del mundo hay

dos clases. La primera, llamada `Muestra`, representa una secuencia no ordenada de valores en un rango. Tiene en su interior tres atributos, tal como se muestra en la figura 1.10: un arreglo con los valores de la muestra (`valores`), el cual tiene reservado el espacio definido en el constructor, pero que sólo tiene un número dado de elementos presentes (`tamaño`), todos estos valores

en el rango 1..`limiteSuperior`. En la figura 1.11 aparece un posible diagrama de objetos, en donde se puede observar una instancia de esta clase.

La segunda clase, llamada `MuestraOrdenada`, representa una secuencia ordenada de valores. Esta clase sólo tiene como atributo un arreglo que almacena los valores de la muestra. Dicho arreglo se encuentra completamente lleno y la clase no tendrá métodos para agregar valores, eliminarlos o modificarlos, puesto que

no hay ningún requerimiento del caso que así lo exija.

En el diagrama de clases se puede ver una asociación marcada con una línea punteada, que indica que las dos clases tienen una dependencia, aunque no exista una asociación directa entre ellas. En este caso la dependencia consiste en que vamos a tener un método de la clase `Muestra` que es capaz de crear instancias de la clase `MuestraOrdenada`. En la figura 1.11 se ilustra esto con un diagrama de objetos.

Fig. 1.10 – **Diagrama de clases del modelo del mundo**

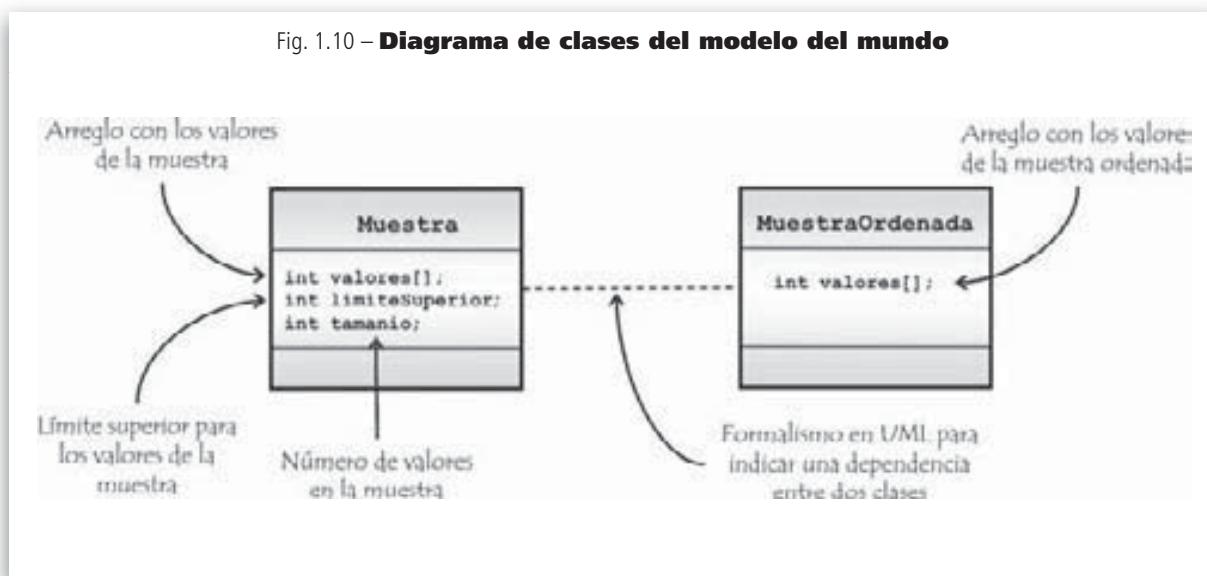
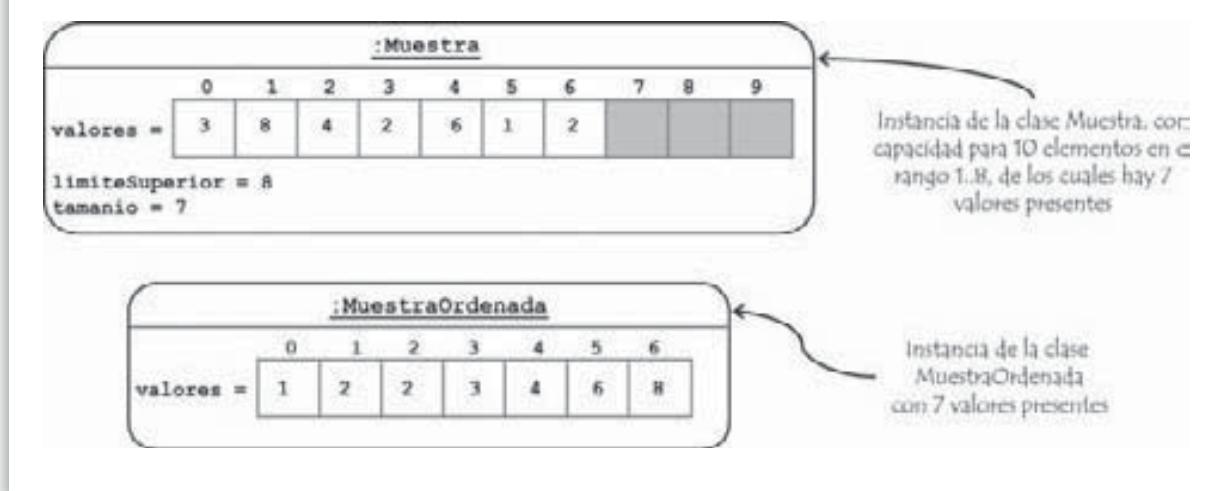


Fig. 1.11 – **Diagrama de objetos como un ejemplo de un estado posible del modelo del mundo**



En la siguiente tarea pediremos al lector que defina el invariante de las clases antes descritas y que implemente los métodos que nos van a permitir su verificación durante la ejecución.

Tarea 8

Objetivo: Definir el invariante de cada una de las clases del caso de estudio e implementar el método que lo verifica.

Para las clases `Muestra` y `MuestraOrdenada`, siga los pasos que se proponen a continuación:

1. Defina el invariante de cada una de las clases del caso de estudio:

Clase	Atributo	Invariante análisis	Invariante diseño
Muestra	valores		
Muestra	limiteSuperior		
Muestra	tamanio		
MuestraOrdenada	valores		

2. Escriba el método en cada clase que verifica en ejecución que el invariante se cumple:

```
public class Muestra
{
    private void verificarInvariante()
    {
        }
}
```

```
public class MuestraOrdenada
{
    private void verificarInvariante()
    {
        }
}
```

En la siguiente tabla hacemos un resumen de los principales métodos públicos de las dos clases del caso de estudio. Para ver los contratos exactos se recomienda consultar el CD que acompaña al libro.

Clase Muestra:	
<code>Muestra(int capacidad, int limite)</code>	Crea una muestra vacía (sin elementos), en un rango y con una capacidad dados como parámetro.
<code>void agregarDatos(int valor)</code>	Agrega un valor al final de la muestra. La precondition afirma que hay espacio en la muestra para agregar el nuevo valor.
<code>void generarValores()</code>	Genera de manera aleatoria valores para la muestra, llenándola completamente hasta su capacidad máxima.
<code>int darTamanio()</code>	Retorna el número de elementos presentes en la muestra.
<code>int darCapacidad()</code>	Retorna el número máximo de elementos que puede contener la muestra.
<code>int darLimiteSuperior()</code>	Retorna el límite superior de valores de la muestra.
<code>int darMaximo()</code>	Retorna el máximo valor presente en la muestra. Si la muestra está vacía retorna el valor 0.
<code>int darMinimo()</code>	Retorna el mínimo valor presente en la muestra. Si la muestra está vacía retorna el valor 0.
<code>double darPromedio()</code>	Retorna el promedio de los valores presentes en la muestra. Si la muestra está vacía retorna el valor 0.
<code>MuestraOrdenada ordenarSeleccion()</code>	Crea y retorna una instancia de la clase MuestraOrdenada con todos los elementos de la muestra, utilizando para esto el algoritmo de ordenamiento por selección.
<code>MuestraOrdenada ordenarBurbuja()</code>	Crea y retorna una instancia de la clase MuestraOrdenada con todos los elementos de la muestra, utilizando para esto el algoritmo de ordenamiento por intercambio.
<code>MuestraOrdenada ordenarInsercion()</code>	Crea y retorna una instancia de la clase MuestraOrdenada con todos los elementos de la muestra, utilizando para esto el algoritmo de ordenamiento por inserción.
<code>boolean buscarSecuencial(int valor)</code>	Retorna verdadero si el valor que llega como parámetro se encuentra presente en la muestra.

Clase MuestraOrdenada:

MuestraOrdenada(int[] vals)

■ Crea una muestra ordenada a partir de la información que recibe en un arreglo de valores enteros. La precondition dice que dicho arreglo no es nulo y que viene ordenado ascendenteamente.

int darTamanio()

■ Retorna el número de elementos de la muestra ordenada (que en este caso es igual al tamaño del arreglo de valores).

int contarOcurrencias(int valor)

■ Calcula y retorna el número de veces que aparece un valor dado en la muestra ordenada.

int contarValoresDistintos()

■ Calcula y retorna el número de valores distintos que hay en la muestra ordenada.

int contarElementosEnRango(int inf, int sup)

■ Calcula y retorna el número de elementos de la muestra ordenada que están en el rango inf...sup, incluyendo los dos límites.

int darValorMasFrecuente()

■ Calcula y retorna el valor que más veces aparece en la muestra ordenada.

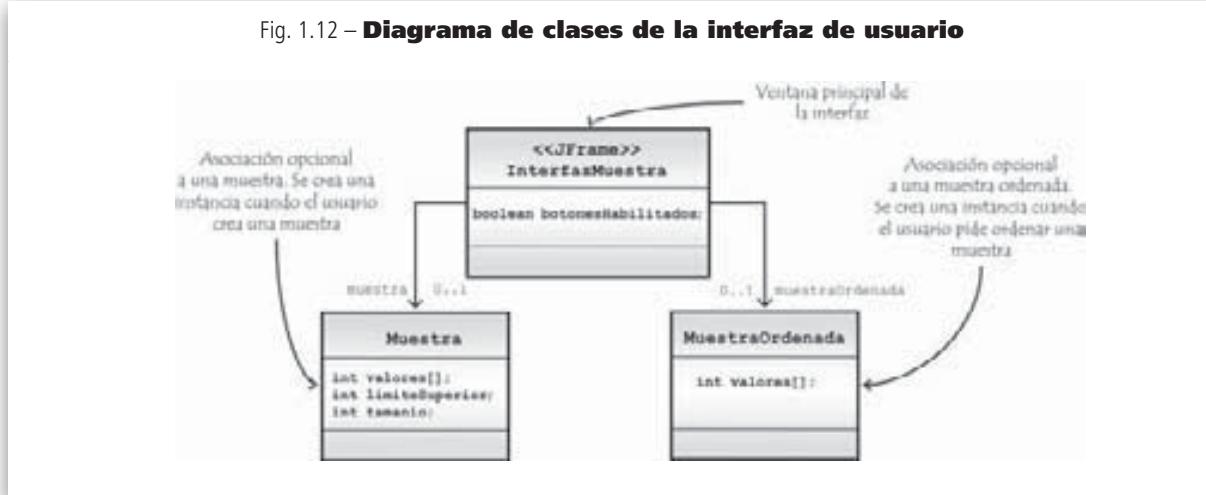
boolean buscarBinario(int valor)

■ Retorna verdadero si el valor que llega como parámetro se encuentra presente en la muestra ordenada, utilizando para esto el algoritmo de búsqueda binaria.

En la figura 1.12 aparece una parte del diagrama de clases de la interfaz de usuario, en la que se puede apreciar su relación con el modelo del mundo. En este

diagrama se puede ver que existe una asociación opcional (0..1) hacia la clase **Muestra** y otra asociación también opcional hacia la clase **MuestraOrdenada**.

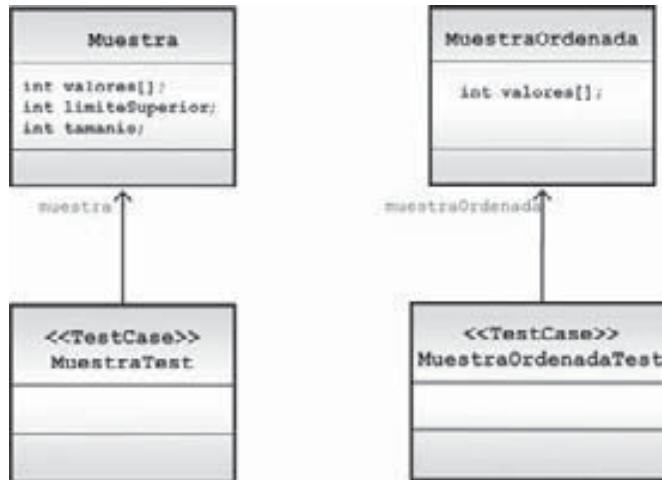
Fig. 1.12 – **Diagrama de clases de la interfaz de usuario**



Tan pronto el usuario genera una muestra, se crea la primera instancia. En el momento de utilizar cualquiera de los métodos de ordenamiento, se crea la instancia de la segunda clase. Existe también un atributo de tipo lógico (`botonesHabilitados`) en la clase de la ventana principal, que indica si se pueden ofrecer al usuario las opciones que trabajan sobre una muestra ordenada, cuyo valor se vuelve verdadero después de haber ordenado la muestra.

Para las pruebas unitarias construiremos dos clases (`MuestraTest` y `MuestraOrdenadaTest`), relacionadas con el modelo del mundo como aparece en la figura 1.13. Los escenarios y los casos de prueba son tema de una sección posterior. Por ahora nos contentamos con definir la parte estructural de la solución, e identificar las responsabilidades de cada uno de los componentes de ella.

Fig. 1.13 – **Diagrama de clases de las pruebas**



4.4. Algoritmos de Ordenamiento en Memoria Principal

Ahora que ya está completamente definida la arquitectura de la solución, nos podemos concentrar en la parte algorítmica, la cual constituye el principal objetivo de este caso de estudio. En esta sección trabajaremos sobre tres de los algoritmos de ordenamiento más simples que existen y veremos la manera de usarlos para escribir los métodos de la clase `Muestra`, especificados en la sección anterior.

Todos los algoritmos de ordenamiento que presentamos en esta sección manejan un orden ascendente y trabajan en una dirección determinada (de izquierda a

derecha o de derecha a izquierda). Los cambios que se deben hacer para ordenar un arreglo de valores descendente o para trabajar en la dirección contraria son mínimos, por lo que no serán tratados aquí de manera aparte.

4.4.1. Ordenamiento por Selección

Aunque todos los algoritmos de ordenamiento satisfacen el mismo contrato, existen distintas maneras de cumplirlo, cada una de ellas con pequeñas ventajas y desventajas. El primero de los algoritmos que vamos a estudiar es el que utiliza una técnica denominada de selección, y se basa en la idea que se ilustra en el ejemplo 9.

**Ejemplo 9**

Objetivo: Mostrar el funcionamiento del algoritmo de ordenamiento por selección.

En este ejemplo se muestra cada una de las etapas por las que pasa el algoritmo de ordenamiento por selección para ordenar ascendente un arreglo de valores de tipo simple.

58	55	24	81	21	87	54	75	57	88
21	55	24	81	58	87	54	75	57	88
21	24	55	81	58	87	54	75	57	88
21	24	54	81	58	87	55	75	57	88
21	24	54	55	58	87	81	75	57	88
21	24	54	55	57	87	81	75	58	88
21	24	54	55	57	58	81	75	87	88
21	24	54	55	57	58	75	81	87	88
21	24	54	55	57	58	75	81	87	88
21	24	54	55	57	58	75	81	87	88

■ En este ejemplo tenemos inicialmente un arreglo no ordenado de 10 posiciones (primera fila de la figura).

■ Despues de la primera iteración del algoritmo, queremos que el menor elemento (21) quede en la primera posición. Para esto buscamos el menor valor de todo arreglo y lo intercambiamos con el que está en la primera posición (58). Así obtenemos el arreglo que aparece en la segunda fila del ejemplo.

■ Luego, repetimos el mismo proceso pero comenzando desde el segundo elemento del arreglo. Buscamos el menor (24) y lo remplazamos con el primer elemento que se encuentra en la parte sin ordenar (55).

■ El proceso continúa hasta que todo el arreglo queda ordenado, tal como se ilustra en la siguiente secuencia de pasos.

58	55	24	81	21	87	54	75	57	88
21	55	24	81	58	87	54	75	57	88

■ Primera iteración: Todo el arreglo se encuentra sin ordenar. Buscamos el menor elemento (21) y lo intercambiamos con el primer elemento de la parte no ordenada (58).

21	55	24	81	58	87	54	75	57	88
21	24	55	81	58	87	54	75	57	88

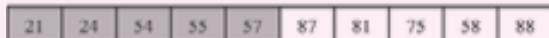
■ Segunda iteración: Ya hay un elemento ordenado (21). Buscamos el menor de la parte sin ordenar (24) y lo intercambiamos con el primer elemento del arreglo que no está ordenado (55).

21	24	55	81	58	87	54	75	57	88
21	24	54	81	58	87	55	75	57	88

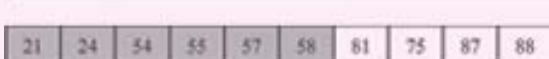
■ Tercera iteración: Ya hay dos elementos ordenados en el arreglo (21, 24). Intercambiemos el menor de la parte restante (54) con el primero de la misma (55).

21	24	55	81	58	87	54	75	57	88
21	24	54	81	58	87	55	75	57	88

■ Cuarta iteración: Incluimos en la parte ordenada el elemento 55, intercambiándolo por el 81. Con eso completamos ya cuatro elementos ordenados ascendenteamente.



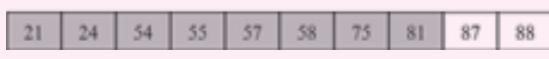
Quinta iteración: Repetimos el mismo proceso anterior para intercambiar los elementos 57 y 58. Fíjese que en todo momento los valores de la parte ordenada son menores que todos los que se encuentran en la parte no ordenada.



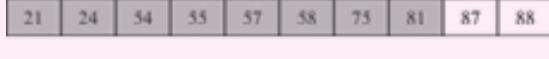
Sexta iteración: Ya hay cinco elementos ordenados. Seleccionamos el menor de la parte no ordenada (58) y lo intercambiamos con el primero de la parte no ordenada (87).



Séptima iteración: Intercambiamos los elementos 75 y 81 para avanzar en el proceso de ordenamiento.



Octava iteración: Intercambiamos los elementos 87 y 81 para completar ocho elementos ordenados en el arreglo.



Novena iteración: En este caso no hay que hacer ningún intercambio, puesto que el menor de la parte no ordenada (87) ya se encuentra en la primera posición.

Cuando hayamos ordenado todos los elementos menos uno terminamos el proceso, puesto que el restante (88) es necesariamente mayor que todos los que ya fueron incluidos en la parte ordenada.

A continuación se muestra el código del método de la clase **Muestra** encargado de crear una instancia

de la clase **MuestraOrdenada** usando la técnica de ordenamiento por selección:

```
public MuestraOrdenada ordenarSeleccion( )
{
    int[] arreglo = darCopiaValores();
    for( int i = 0; i < tamanio - 1; i++ )
    {
        int menor = arreglo[ i ];
        int cual = i;
        for( int j = i + 1; j < tamanio; j++ )
        {
```

Inicialmente crea una copia de los valores de la muestra, para ordenarlos, usando el método `darCopiaValores()`.

El ciclo externo del método lleva el índice "i" señalando el punto en el cual comienza la parte sin ordenar del arreglo. Comienza en 0 y termina cuando llega a la penúltima posición.

```

        if( arreglo[ j ] < menor )
    {
        menor = arreglo[ j ];
        cual = j;
    }
}
int temp = arreglo[ i ];
arreglo[ i ] = menor;
arreglo[ cual ] = temp;
}
return new MuestraOrdenada( arreglo );
}

```



El objetivo del ciclo interior es buscar el menor valor de la parte sin ordenar. Dicha parte comienza en la posición “*i*” y va hasta el final del arreglo. Al final del ciclo, deja en la variable “menor” el menor valor encontrado y en la variable “cual” su posición dentro del arreglo.



Después de haber localizado el menor elemento, lo intercambia con el que se encuentra en la casilla “*i*” del arreglo. Para esto utiliza una variable temporal “temp”.



Cuando finalmente ha terminado de ordenar el arreglo, crea una instancia de la clase MuestraOrdenada.

Tarea 9



Objetivo: Utilizar la técnica de ordenamiento por selección, para ordenar ascendentemente una secuencia de valores.

Suponiendo que los valores que se encuentran en la siguiente tabla corresponden a una secuencia de números enteros que quiere ordenar, muestre el avance en cada una de las iteraciones para el caso en el cual utilice la técnica de ordenamiento por selección. Marque claramente en cada iteración la parte que ya está ordenada y el menor elemento de la parte por ordenar.



El algoritmo de ordenamiento por selección se basa en la idea de tener dividido el arreglo que se está ordenando en dos partes: una, con un grupo de elementos ya ordenados, que ya encontraron su posición final. La otra, con los elementos que no han sido todavía ordenados. En cada iteración, localizamos el menor elemento de la parte no ordenada, lo intercambiamos con el primer elemento de esta misma región e indicamos que la parte ordenada ha aumentado en un elemento.

4.4.2. Ordenamiento por Intercambio (Burbuja)

Otra técnica que se usa frecuentemente para ordenar

secuencias de valores se basa en la idea de ir intercambiando todo par de elementos consecutivos que no se encuentren ordenados, tal como se ilustra en el ejemplo 10.

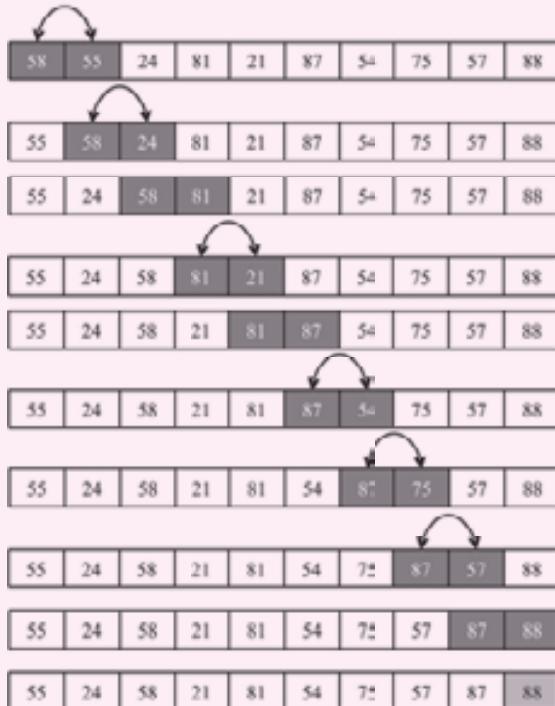


Ejemplo 10 **Objetivo:** Mostrar el funcionamiento del algoritmo de ordenamiento por intercambio (también llamado ordenamiento de burbuja).

En este ejemplo se muestra cada una de las etapas por las que pasa el algoritmo de ordenamiento de burbuja para ordenar ascendenteamente un arreglo de valores de tipo simple.

58	55	24	81	21	87	54	75	57	88
55	24	58	21	81	54	72	57	87	88
24	55	21	58	54	75	57	81	87	88
24	21	55	54	58	57	72	81	87	88
21	24	54	55	57	58	72	81	87	88
21	24	54	55	57	58	72	81	87	88
21	24	54	55	57	58	72	81	87	88
21	24	54	55	57	58	72	81	87	88
21	24	54	55	57	58	72	81	87	88
21	24	54	55	57	58	72	81	87	88

- En este ejemplo, tenemos inicialmente un arreglo no ordenado de 10 posiciones (primera fila de la figura).
- En la primera iteración recorremos el arreglo de izquierda a derecha intercambiando todo par de valores consecutivos que no se encuentren ordenados. Al final de la primera iteración, el mayor de los elementos (88) debe quedar en la última posición del arreglo, mientras todos los demás valores han avanzado un poco hacia su posición final.
- El valor 87, por ejemplo, alcanzó a avanzar tres posiciones hacia su posición definitiva.
- Este mismo proceso se repite para la parte del arreglo que todavía no está ordenada (en blanco en la figura).
- El proceso termina cuando sólo queda un elemento en la zona no ordenada del arreglo (al comienzo), lo cual requiere nueve iteraciones en nuestro ejemplo (número de elementos menos uno).



- En esta figura aparecen todos los intercambios que tienen lugar en la primera iteración.
- Si vamos en la posición "i" del arreglo, y el valor en esa posición es mayor que el valor de la posición "i+1", los intercambiamos.
- El proceso en esta iteración va hasta el último elemento del arreglo. A medida que vayamos avanzando en la ejecución del algoritmo, iremos acortando el punto hasta el cual hay que hacer los intercambios.
- Al final de la primera iteración, el último elemento ya se encuentra ordenado.
- Este método se denomina de burbuja, porque hace que los elementos vayan subiendo poco a poco hasta ocupar su posición final.
- Al final de cada iteración, en la parte final del arreglo están los elementos ya ordenados, los cuales además son mayores que todos los elementos que faltan por ordenar.



El algoritmo de ordenamiento por intercambio (conocido también como ordenamiento de burbuja) se basa en la idea de intercambiar todo par de elementos consecutivos que no se encuentren en orden. Al final de cada pasada haciendo este intercambio, un nuevo elemento queda ordenado y todos los demás elementos se acercaron a su posición final.

A continuación se muestra el código del método de la clase `Muestra` encargado de crear una instancia de

la clase `MuestraOrdenada` usando la técnica de ordenamiento por intercambio (burbuja):

```
public MuestraOrdenada ordenarBurbuja()
{
    int[] arreglo = darCopiaValores();
    for( int i = tamano; i > 0; i-- )
    {
        for( int j = 0; j < i - 1; j++ )
        {
            if( arreglo[ j ] > arreglo[ j + 1 ] )
            {
                ...
            }
        }
    }
}
```

- Inicialmente crea una copia de los valores de la muestra, para ordenarlos, usando el método `darCopiaValores()`.
- El ciclo externo va controlando con la variable "i" el punto hasta el cual hay que llevar el proceso de intercambio. Inicialmente va hasta el final de la secuencia, y va disminuyendo hasta que sólo queda un elemento (termina cuando `i==0`).

```
        int temp = arreglo[ j ];
        arreglo[ j ] = arreglo[ j + 1 ];
        arreglo[ j + 1 ] = temp;
    }
}
}

return new MuestraOrdenada( arreglo );
}
```

- En el ciclo interno se lleva a cabo el proceso de intercambio con la variable "j", comenzando desde la posición 0 hasta llegar al punto anterior al límite marcado por la variable "i". Allí compara (y si es necesario intercambia) los valores de las posiciones "j" y "j+1".
 - Cuando finalmente ha terminado de ordenar el arreglo, crea una instancia de la clase MuestraOrdenada.

Tarea 10



Objetivo: Utilizar la técnica de ordenamiento por intercambio (burbuja), para ordenar ascendente mente una secuencia de valores.

Suponiendo que los valores que se encuentran en la siguiente tabla corresponden a una secuencia de números enteros que quiere ordenar, muestre el avance en cada una de las iteraciones para el caso en el cual utilice la técnica de ordenamiento por intercambio (burbuja). Marque claramente en cada iteración la parte del arreglo que ya se encuentra ordenada.

4.4.3. Ordenamiento por Inserción

Uno de los métodos más naturales para ordenar una secuencia de valores consiste en separar la secuencia en dos grupos: una parte con los valores ordenados

(initialmente con un solo elemento) y otra con los valores por ordenar (initialmente todo el resto). Luego, vamos pasando uno a uno los valores a la parte ordenada, asegurándonos de que se vayan colocando ascendenteamente, tal como se ilustra en el ejemplo 11.

Ejemplo 11


Objetivo: Mostrar el funcionamiento del algoritmo de ordenamiento por inserción.

En este ejemplo se muestra cada una de las etapas por las que pasa el algoritmo de ordenamiento por inserción para ordenar ascendenteamente un arreglo de valores de tipo simple.

58	55	24	81	21	87	54	75	57	88
55	58	24	81	21	87	54	75	57	88
24	55	58	81	21	87	54	75	57	88
24	55	58	81	21	87	54	75	57	88
21	24	55	58	81	87	54	75	57	88
21	24	55	58	81	87	54	75	57	88
21	24	54	55	58	81	87	75	57	88
21	24	54	55	58	75	81	87	57	88
21	24	54	55	57	58	75	81	87	88
21	24	54	55	57	58	75	81	87	88

■ En este ejemplo, tenemos inicialmente un arreglo no ordenado de 10 posiciones (primera fila de la figura).

■ Al inicio, podemos suponer que en la parte ordenada del arreglo hay un elemento (58). No está todavía en su posición final, pero siempre es cierto que una secuencia de un solo elemento está ordenada.

■ Luego tomamos uno a uno los elementos de la parte no ordenada y los vamos insertando ascendenteamente en la parte ordenada. Comenzamos con el 55 y vamos avanzando hasta insertar el valor 88.

■ En la figura, en cada iteración aparece marcado el elemento que acaba de ser insertado.

58	55	24	81	21	87	54	75	57	88
	58	24	81	21	87	54	75	57	88
55	58	24	81	21	87	54	75	57	88

■ Primera iteración: En la parte ordenada de la secuencia sólo está el valor 58. Vamos a insertar a esa parte el valor 55. Debemos buscar el punto en el que dicho valor debería encontrarse para que esa parte siga ordenada y desplazar los elementos necesarios que ya se encuentran allí. En este caso debemos mover el valor 58 a la derecha para abrir el espacio necesario para el 55.

55	58	24	81	21	87	54	75	57	88
24	55	58	81	21	87	54	75	57	88

■ Segunda iteración: La parte ordenada ya tiene los valores 55 y 58. Vamos a insertar allí el valor 24. Repetimos el mismo proceso de desplazamiento y le abrimos espacio a este valor en el punto adecuado, para que esta parte siga ordenada.

24	55	58	 81	21	87	54	75	57	88
----	----	----	--	----	----	----	----	----	----

 Tercera iteración: Insertar el valor 81 a la parte ordenada es simple, puesto que no hay que desplazar ningún valor, ya que el 81 es mayor que todos los valores presentes en la parte ordenada.

24	55	58	81	 21	87	54	75	57	88
----	----	----	----	--	----	----	----	----	----

 Cuarta iteración: Insertamos el valor 21, quedando en la primera posición del arreglo. Allí tenemos que desplazar a la derecha los cuatro valores que ya estaban ordenados.

21	24	55	58	81	 87	54	75	57	88
----	----	----	----	----	--	----	----	----	----

 Quinta iteración: Insertamos el valor 87 sin necesidad de desplazar ningún elemento, puesto que debe quedar al final.

21	24	55	58	81	87	 54	75	57	88
----	----	----	----	----	----	---	----	----	----

 Sexta iteración: Para insertar el valor 54, debemos desplazar los valores que son mayores que él (55, 58, 81 y 87). Los demás (21, 24) no se mueven.

21	24	54	55	58	81	87	 75	57	88
----	----	----	----	----	----	----	--	----	----

 Séptima iteración: Vamos a insertar ahora el valor 75, para lo cual movemos hacia la derecha los valores 81 y 87.

21	24	54	55	58	75	81	87	 57	88
----	----	----	----	----	----	----	----	--	----

 Octava iteración: Para insertar el valor 57 movemos hacia la derecha los valores 58, 75, 81 y 87.

21	24	54	55	57	58	75	81	87	 88
----	----	----	----	----	----	----	----	----	--

 Novena iteración: La última iteración no implica ningún desplazamiento, puesto que el 88 es el mayor valor de todo el arreglo.

Tarea 12

Objetivo: Medir el tiempo de ejecución de los algoritmos de ordenamiento e intentar identificar diferencias entre ellos.

Localice en el CD el ejemplo n7_muestra, cópielo al disco duro y ejecútelo. Siga luego las instrucciones que aparecen a continuación.

Algoritmo	5000	7000	10000	15000	20000	30000	40000	50000
Selección								
Burbuja								
Inserción								



Llene la tabla de tiempos de ejecución de los tres algoritmos de ordenamiento, para muestras de los tamaños allí especificados.

¿Qué se puede concluir de la tabla anterior?



Localice en el CD que acompaña al libro los entrenadores de ordenamiento allí disponibles, para complementar así lo estudiado en esta parte.

4.4.4. ¿Y Cuándo Ordenar?

Hay dos razones principales para ordenar información. La primera, para facilitar la interacción con el usuario. Si tenemos, por ejemplo, una lista de códigos de productos para que el usuario seleccione uno de ellos, es mucho más fácil para él encontrar lo que está buscando si le presentamos esta lista de manera ordenada. En ese caso no tenemos mayores opciones y debemos utilizar nuestros algoritmos de ordenamiento. La segunda razón es para hacer más eficientes los programas. Un método que busca información sobre una estructura que está ordenada gasta mucho menos tiempo que si los valores allí presentes no tienen ningún orden. El problema que se nos presenta aquí es que ordenar la información pue-

de tomar un tiempo considerable; luego la pregunta que nos debemos hacer es: ¿cuándo es conveniente ordenar la información? La respuesta afortunadamente es simple y depende del número de búsquedas que se vayan a hacer. Si es una sola búsqueda, definitivamente no es un buen negocio ordenar antes la información, pero si vamos a hacer miles de ellas sobre el mismo conjunto de datos, la ganancia en tiempo amerita hacer un proceso previo de ordenamiento.

Más adelante veremos algunas estructuras de datos que están hechas para mantener permanentemente ordenada la información que contienen, lo que las hace ideales para manejar información sobre la cual hay búsquedas y modificaciones todo el tiempo.

4.5. Algoritmos de Búsqueda en Memoria Principal

En esta sección vamos a estudiar los algoritmos de búsqueda de un elemento en un arreglo (¿dónde está un elemento dado?) y los algoritmos de búsqueda en general de distintas características en un conjunto de valores (por ejemplo, ¿cuántas veces aparece un valor en un arreglo?), los cuales necesitamos para escribir los métodos que implementan los requerimientos funcionales del caso de estudio.

4.5.1. Búsqueda de un Elemento

El proceso de búsqueda de un elemento en un arreglo depende básicamente de si éste se encuentra ordenado. En caso de que no haya ningún orden en los valores, la única opción que tenemos es hacer una búsqueda secuencial utilizando para esto el patrón de recorrido parcial estudiado en cursos anteriores.

Tarea 13



Objetivo: Escribir el método que busca un elemento en un arreglo.

Para la clase `Muestra` escriba dos versiones del método que busca un elemento: en el primero, haga una búsqueda secuencial suponiendo que la muestra no está ordenada. En el segundo, haga también una búsqueda secuencial pero suponga que la muestra está ordenada ascendente. ¿Qué cambia de un algoritmo a otro?

```
public class Muestra
{
    public boolean buscarSecuencial( int valor )
    {
        }

    }

public class Muestra
{
    public boolean buscarSecuencial( int valor )
    {
        }
}
```

Si el arreglo en el que queremos buscar el elemento se encuentra ordenado, podemos utilizar una técnica muy eficiente de localización que se denomina la **búsqueda binaria**. La idea de esta técnica, que se ilustra en el ejemplo 12, es localizar el elemento que se encuentra en la mitad del arreglo. Si ese elemento es mayor que el valor que estamos buscando, podemos descartar en el proceso de búsqueda la mitad final del arreglo (¿para qué buscar allí si todos los

valores de esa parte del arreglo son mayores que aquél que estamos buscando?). Si el elemento de la mitad es menor que el valor buscado, descartamos la mitad inicial del arreglo. Piense que sólo con lo anterior ya bajamos el tiempo de ejecución del método a la mitad. Y si volvemos a repetir el mismo proceso anterior con la parte del arreglo que no hemos descartado, iremos avanzando rápidamente hacia el valor que queremos localizar.

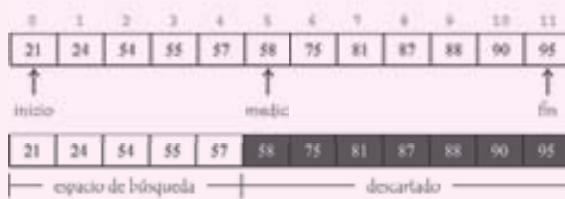
Ejemplo 12



Objetivo: Mostrar el funcionamiento del algoritmo de búsqueda binaria.

En este ejemplo se muestra cada una de las etapas por las que pasa el algoritmo de búsqueda binaria para localizar un elemento en un arreglo ordenado de valores.

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95



Este es el arreglo ordenado sobre el que vamos a hacer las búsquedas. Tiene 12 valores.

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95



En la segunda iteración los valores de fin (4) y medio (2) han sido recalculados para contemplar únicamente la parte del arreglo en la que podría aparecer el valor buscado. Puesto que estamos localizando el 55, y éste es mayor que el valor que se encuentra en la posición del medio (54), descartamos de la búsqueda los elementos que están entre inicio y medio.

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95

Repetimos el proceso recalculando los valores de inicio, fin y medio. Puesto que el valor que se encuentra en la posición del medio es el valor buscado (55), terminamos el proceso.

0	1	2	3	4	5	6	7	8	9	10	11
21	24	54	55	57	58	75	81	87	88	90	95

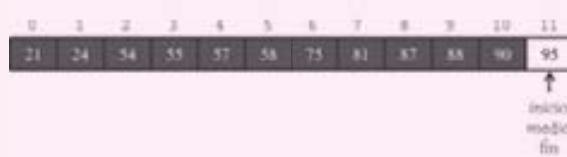
Vamos a buscar ahora un valor inexistente en el arreglo para ver el comportamiento del algoritmo. Busquemos entonces el 92. En la primera iteración descartamos los valores de la primera mitad, puesto que el 92 es mayor que el valor del medio (58).



En la segunda iteración descartamos de nuevo la primera parte de la zona de búsqueda, puesto que el 92 es mayor que el elemento del medio (87).



En la tercera iteración reducimos la zona de búsqueda a un solo elemento (95), puesto que el 92 es mayor que aquél que se encuentra en la posición del medio (90).



Puesto que el único elemento que queda en la zona de búsqueda es el 95 y es diferente del valor buscado, podemos afirmar en este punto que el 92 no está en el arreglo.

Fíjese que en sólo cuatro iteraciones nos pudimos dar cuenta de que un valor no existía en un arreglo de 12 posiciones. Nada mal, ¿no?

A continuación se muestra el código del método de la clase `MuestraOrdenada` encargado de buscar un elemento usando la técnica de búsqueda binaria:

```
public boolean buscarBinario( int valor )
{
    boolean encontre = false;
    int inicio = 0;
    int fin = valores.length - 1;
    while( inicio <= fin && !encontre )
    {
        int medio = ( inicio + fin ) / 2;
        if( valores[ medio ] == valor )
        {
            encontre = true;
        }
        else if( valores[ medio ] > valor )
        {
            fin = medio - 1;
        }
        else
        {
            inicio = medio + 1;
        }
    }
    return encontre;
}
```

Vamos a utilizar tres variables enteras (`inicio`, `medio` y `fin`) para indicar en cada iteración la posición donde comienza la zona de búsqueda, la posición media y la posición en la que termina.

Inicialmente “`inicio`” comienza en 0 y “`fin`” en la última posición del arreglo.

La posición media se calcula sumando las posiciones de `inicio` y `fin`, y dividiendo el resultado entre 2.

Luego, hay tres casos: (1) si el valor es igual al de la mitad, ya lo encontramos; (2) si el valor es menor que el de la mitad, repositionamos la marca final de la zona de búsqueda, y (3) si el valor es mayor, movemos el inicio de la zona de búsqueda.

Este proceso se repite mientras no hayamos encontrado el elemento y mientras exista algún elemento en el interior de la zona de búsqueda.

Tarea 14

Objetivo: Medir el tiempo de ejecución de los algoritmos de búsqueda e intentar identificar diferencias entre ellos.

Localice en el CD el ejemplo `n7_muestra`, cópielo al disco duro y ejecútelo. Siga luego las instrucciones que aparecen a continuación.

Algoritmo	5000	10000	20000	40000	60000	80000	100000	200000
Secuencial								
Binaria								

¿Qué se puede concluir de la tabla anterior?



Llene la tabla de tiempos de ejecución de los dos algoritmos de búsqueda, para muestras de distintos tamaños. Utilice una muestra entre 1 y 200.000.

4.5.2. Búsquedas en Estructuras Ordenadas

En esta sección planteamos, en términos de tareas, los métodos de búsqueda necesarios para completar el programa del caso de estudio.

Tarea 15

Objetivo: Escribir los métodos que se necesitan en el caso de estudio, para calcular algún valor sobre la muestra ordenada.

Escriba los métodos que se piden a continuación en la clase `MuestraOrdenada`. Asegúrese de utilizar en su algoritmo el hecho de que los valores de la muestra se encuentran ordenados, para así tratar de encontrar una solución lo más eficiente posible.

```
public int contarOcurrencias( int valor )
{
    }
```



Cuenta el número de veces que aparece un valor en la muestra ordenada.

```
public int contarValoresDistintos( )  
{  
}  
  
}  
  
public int contarElementosEnRango( int inf, int sup )  
{  
}  
  
}  
  
public int darValorMasFrecuente( )  
{  
}  
  
}  
}
```



Cuenta el número de valores distintos que hay en la muestra ordenada.



Cuenta el número de elementos que hay en un rango de valores (incluidos los extremos).



Retorna el valor más frecuente en la muestra ordenada. Si hay varios números con la misma frecuencia, retorna el menor de ellos.

4.6. Generación de Datos y Medición de Tiempos

El siguiente problema al que nos enfrentamos es la generación aleatoria de los valores de la muestra. Para hacer esto contamos con el siguiente método:

```
public void generarValores( )
{
    for( int i = 0; i < valores.length; i++ )
    {
        double aleatorio = Math.random( ) *
limiteSuperior;

        valores[ i ] = ( int )aleatorio + 1;

        tamanio = valores.length;
    }
}
```

Dos métodos adicionales que pueden ser de utilidad para manipular valores enteros son los siguientes:

- `Math.max(valor1, valor2)` : Retorna el mayor valor entre `valor1` y `valor2`.
- `Math.min(valor1, valor2)` : Retorna el menor valor entre `valor1` y `valor2`.

Por último, necesitamos medir el tiempo que toma en ejecutarse un método. Para hacer esto tenemos los siguientes métodos de Java:

- `System.currentTimeMillis()` : Retorna un valor de tipo `long` con la fecha actual, expresada como el número de milisegundos que han transcurrido desde el 1 de enero de 1970 a las 0 horas.

```
public long medir( )
{
    long t1 = System.currentTimeMillis( );

    muestra.metodo( );

    long t2 = System.currentTimeMillis( );

    return t2 - t1;
}
```

- `Math.random()` : Retorna un valor aleatorio de tipo `double` que es mayor o igual a cero y menor que uno. El método garantiza una distribución (aproximadamente) uniforme en ese rango.

Puesto que necesitamos generar valores enteros entre 1 y un límite superior, agregamos el siguiente método a la clase `Muestra`:

- Al multiplicar el límite superior por el valor aleatorio generado, obtenemos un valor real mayor o igual a cero y menor que el límite superior.
- Al utilizar el operador de conversión a enteros (`int`), el resultado anterior se trunca, dejando sólo un valor entero entre 0 y el límite superior menos 1.
- Sumando 1 a este resultado obtenemos el valor que necesitamos.

- `System.nanoTime()` : Retorna un valor de tipo `long` que mide el tiempo en nanosegundos (la milmillonésima parte de un segundo), con respecto a un sistema no especificado de tiempo. Este método sólo está disponible desde la versión 5 de Java.

Puesto que necesitamos medir el tiempo de ejecución de un método, vamos a utilizar el siguiente fragmento de código cada vez que debamos hacerlo. Para nuestro caso vamos a calcular el tiempo del método en milisegundos, aunque, si es necesaria mayor precisión, podríamos utilizar el método `System.nanoTime()` :

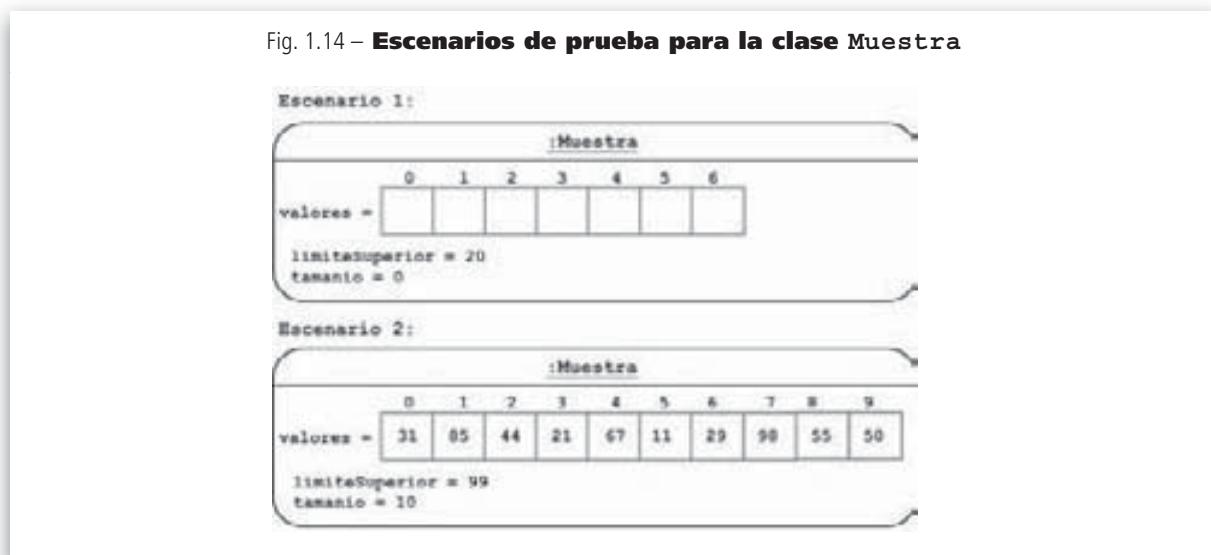
- Suponga que estamos interesados en medir el tiempo que toma la ejecución de un método de la clase `Muestra`.
- Primero calculamos el tiempo antes de comenzar la ejecución y almacenamos este valor en la variable "t1".
- Luego ejecutamos el método y volvemos a medir el tiempo cuando éste termine. Este valor lo almacenamos en la variable "t2".
- Finalmente retornamos la diferencia de los dos tiempos medidos ($t2 - t1$).

4.7. Pruebas Unitarias Automáticas

En esta sección vamos a construir una parte de las pruebas de la clase `Muestra`. Se recomienda consultar el CD que acompaña al libro para estudiar el resto de las pruebas.

Vamos a trabajar sobre dos escenarios, los cuales se muestran en la figura 1.14. El primero de ellos se encuentra vacío y el segundo ya está lleno (llegó a su capacidad, que es de 10 elementos). Sobre el primero haremos las pruebas del método que agrega valores, mientras que sobre el segundo probaremos los métodos de ordenamiento.

Fig. 1.14 – Escenarios de prueba para la clase `Muestra`



Comencemos por declarar la clase de prueba y desarrollar los métodos que crean los dos escenarios.



Tarea 16

Objetivo: Declarar la clase de prueba `MuestraTest` y crear los métodos que construyen los escenarios que vamos a utilizar.

Escriba la declaración completa de la clase `MuestraTest`, siguiendo el diagrama de clases que se presenta en la figura 1.13. Luego, implemente los métodos necesarios para construir los dos escenarios de la figura 1.14. No olvide que en la sección 4.3 están definidos los métodos de la clase `Muestra` que puede utilizar con este fin.

Vamos ahora a definir los casos de prueba para los métodos `agregarDato(valor)`, `ordenarInsercion()`, `buscarSecuencial(valor)` y `generarValores()`. En la implementación de algunos de ellos vamos a utilizar el método `darDato(posicion)`, que nos retorna el valor que se encuentra en una posición válida de la muestra.

Ejemplo 13

Objetivo: Definir las pruebas para algunos métodos de la clase `Muestra`.

(1) Lea detenidamente en la sección 4.3 la descripción de los métodos de la clase `Muestra` e identifique los valores de entrada y el resultado esperado, y (2) estudie las pruebas que se describen a continuación.

Prueba No. 1	Objetivo: Probar que el método <code>agregarDato()</code> es capaz de incluir nuevos valores en la muestra. Puesto que la precondition del método exige que exista espacio en la muestra y que el valor se encuentre en el rango válido, no es mucho lo que se debe probar.	
Escenario 1	Valores de entrada: 12, 5, 7, 1, 10, 1, 19	Resultado: La muestra debe quedar así: [12, 5, 7, 1, 10, 1, 19] El tamaño de la muestra debe ser 7.
Prueba No. 2	Objetivo: Probar que el método <code>generarValores()</code> llena de manera correcta la muestra con valores aleatorios.	
Escenario 1	Valores de entrada: Ninguno	Resultado: El tamaño de la muestra debe ser 7. No hay necesidad de verificar nada más, puesto que el invariante se encarga de ello.
Prueba No. 3	Objetivo: Probar que el método <code>ordenarInsercion()</code> ordena de manera correcta la muestra. Aquí no hay manera de verificar que el método haya aplicado la técnica de ordenamiento por inserción; nos tenemos que contentar con comprobar que al final haya creado una muestra ordenada correcta, con los mismos valores de la muestra original.	
Escenario 2	Valores de entrada: Ninguno	Resultado: La muestra ordenada es: [11, 21, 29, 31, 44, 50, 55, 67, 85, 98]
Prueba No. 4	Objetivo: Probar que el método <code>buscarSecuencial()</code> es capaz de localizar los elementos presentes en la muestra.	
Escenario 2	Valores de entrada: 31 (el primero)	Resultado: Verdadero
Escenario 2	Valores de entrada: 50 (el último)	Resultado: Verdadero
Escenario 2	Valores de entrada: 11 (en la mitad)	Resultado: Verdadero
Escenario 2	Valores de entrada: 40 (inexistente)	Resultado: Falso

Después de definidas las pruebas que queremos realizar, pasamos a implementar los métodos que lo hacen.

**Tarea 17**

Objetivo: Escribir los métodos que ejecutan las pruebas definidas en el ejemplo 12.

Implemente en la clase `MuestraTest` los cuatro métodos que llevan a cabo las pruebas definidas en el ejemplo 13. No olvide invocar el método que crea el escenario respectivo.

```
public class MuestraTest extends TestCase
{
    private Muestra muestra;

    public void testAgregarDato( )
    {

    }

    public void testGenerarValores( )
    {

    }
}
```

```
public void testOrdenarInsercion()  
{
```

```
}
```

```
public void testBuscarSecuencial()  
{
```

```
}
```

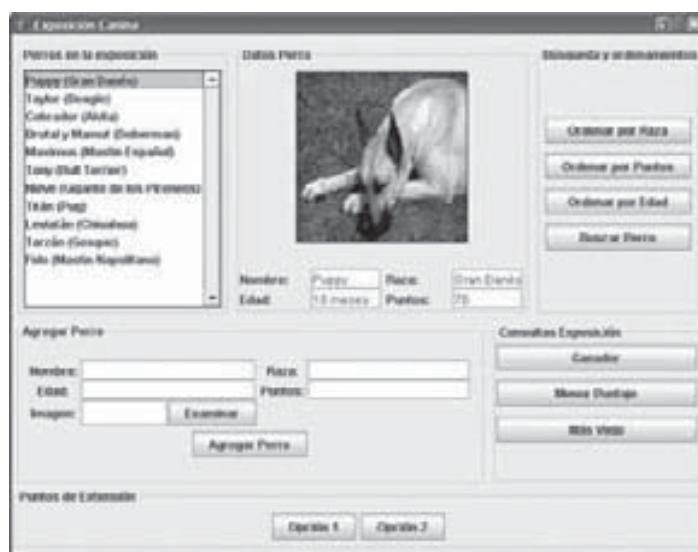
```
}
```

5. Caso de Estudio N° 3: Una Exposición Canina

En el último caso de este nivel vamos a construir un programa para manejar la información de una exposición canina. De cada uno de los perros que participa en la exposición nos interesa registrar su nombre (el cual debe ser único en toda la exposición), su raza, su edad en meses, una imagen y el número de puntos que le asignó el jurado.

En la figura 1.15 aparece la interfaz de usuario que se espera tenga el programa. Éste debe ofrecer los siguientes servicios al usuario: (1) mostrar la lista de los perros registrados en la exposición, ordenada por raza, puntos o edad; (2) mostrar la información de un perro específico, (3) registrar un nuevo perro, (4) localizar un perro por su nombre, (5) buscar al perro ganador de la exposición (el que tiene un mayor puntaje asignado), (6) buscar al perro con menor puntaje y (7) buscar al perro más viejo de todos (con mayor edad).

Fig. 1.15 – Interfaz de usuario para el caso de estudio de la exposición canina



- En la parte izquierda de la ventana aparece la lista de perros registrados en la exposición. Inicialmente no están ordenados.
- Usando los botones que aparecen en la parte derecha de la ventana, se puede ordenar esta lista por distintos conceptos.
- En la parte central aparece la información del perro que se encuentra seleccionado en la lista.
- Para registrar un nuevo perro se deben ingresar sus datos y oprimir el botón **Agregar Perro**.
- Todos los requerimientos de búsqueda aparecen asociados con alguno de los botones de la interfaz.

El programa va a manejar un esquema muy simple de persistencia, en el cual el estado inicial del programa debe cargarse desde un archivo de propiedades

llamado "perros.txt", localizado en el directorio **data**, el cual tiene el formato que se ilustra en el siguiente cuadro:

```
total.perros = 2

perro1.nombre = Puppy
perro1.raza = Gran Danés
perro1.imagen = ./data//gran_danes.jpg
perro1.puntos = 70
perro1.edad = 10

perro2.nombre = Tarzán
perro2.raza = Gosque
perro2.imagen = ./data//gosque.jpg
perro2.puntos = 100
perro2.edad = 137
```

- La primera propiedad del archivo ("total.perros") define el número de perros presentes en el archivo.
- Para cada uno de los perros tenemos cinco datos: nombre, raza, imagen, puntos y edad.
- Cada uno de los datos de un perro se encuentra asociado con una propiedad.

5.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
 Presentar en la interfaz de usuario la lista de los perros de la exposición.	 Estudiar el componente <code>JList</code> de Java, aprender a integrarlo a una interfaz de usuario y a tomar los eventos que sobre los elementos de la lista genere el usuario (clic sobre un perro).
 Ordenar por distintos conceptos un grupo de longitud variable de objetos.	 Adaptar lo visto en el caso anterior para hacer los ordenamientos pedidos, por distintos conceptos y teniendo en cuenta que vamos a manejar objetos en lugar de tipos simples.

5.2. Comprensión de los Requerimientos

Tarea 18										
		Objetivo: Entender el problema del caso de estudio de la exposición canina. (1) Lea detenidamente el enunciado del caso de estudio de la exposición canina e (2) identifique y complete la documentación de los requerimientos funcionales que allí aparecen.								
Requerimiento funcional 1		<table border="1"> <tr> <td data-bbox="374 1012 599 1073">Nombre</td><td data-bbox="599 1012 1348 1073">R1 – Mostrar la lista de perros de la exposición</td></tr> <tr> <td data-bbox="374 1073 599 1216">Resumen</td><td data-bbox="599 1073 1348 1216"></td></tr> <tr> <td data-bbox="374 1216 599 1358">Entradas</td><td data-bbox="599 1216 1348 1358"></td></tr> <tr> <td data-bbox="374 1358 599 1501">Resultado</td><td data-bbox="599 1358 1348 1501"></td></tr> </table>	Nombre	R1 – Mostrar la lista de perros de la exposición	Resumen		Entradas		Resultado	
Nombre	R1 – Mostrar la lista de perros de la exposición									
Resumen										
Entradas										
Resultado										
Requerimiento funcional 2		<table border="1"> <tr> <td data-bbox="374 1531 599 1592">Nombre</td><td data-bbox="599 1531 1348 1592">R2 – Mostrar la información de un perro</td></tr> <tr> <td data-bbox="374 1592 599 1735">Resumen</td><td data-bbox="599 1592 1348 1735"></td></tr> <tr> <td data-bbox="374 1735 599 1859">Entradas</td><td data-bbox="599 1735 1348 1859"></td></tr> </table>	Nombre	R2 – Mostrar la información de un perro	Resumen		Entradas			
Nombre	R2 – Mostrar la información de un perro									
Resumen										
Entradas										

Requerimiento funcional 2	Resultado	
	Nombre	R3 – Registrar un nuevo perro
	Resumen	
	Entradas	
Requerimiento funcional 3	Resultado	
	Nombre	R4 – Localizar un perro por su nombre
	Resumen	
	Entradas	
Requerimiento funcional 4	Resultado	
	Nombre	R5 – Buscar al perro ganador de la exposición
	Resumen	
	Entradas	
Requerimiento funcional 5	Resultado	

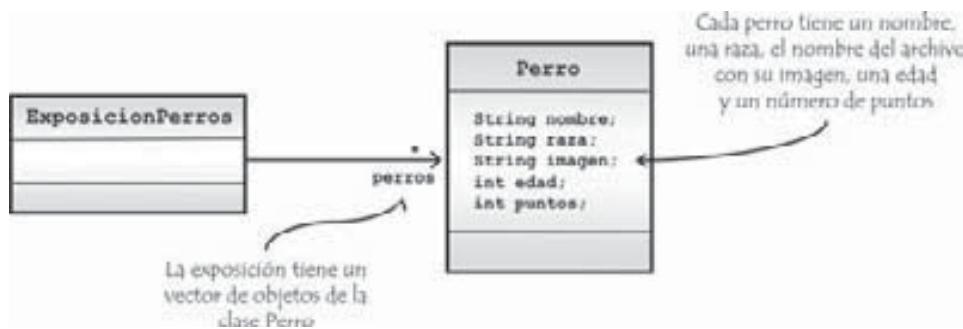
Requerimiento funcional 6	Nombre	R6 – Buscar al perro con menor puntaje
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 7	Nombre	R7 – Buscar al perro más viejo
	Resumen	
	Entradas	
	Resultado	

5.3. Arquitectura de la Solución

El mundo del problema de este caso es muy simple, como se muestra en la figura 1.16. Sólo hay dos enti-

tidades: la primera, que representa un perro (clase *Perro*), con todas sus características, y la segunda, la exposición (clase *ExposicionPerros*), que tiene un vector de perros como su única asociación.

Fig. 1.16 – **Diagrama de clases del modelo del mundo**



Los invariantes de estas clases son:

- Clase ExposiciónPerros:

El vector de perros está inicializado (`perros != null`)

No hay dos perros en el vector que tengan el mismo nombre.

- Clase Perro:

El nombre del perro es una cadena no nula de caracteres (`nombre != null`)

La raza del perro es una cadena no nula de caracteres (`raza != null`)

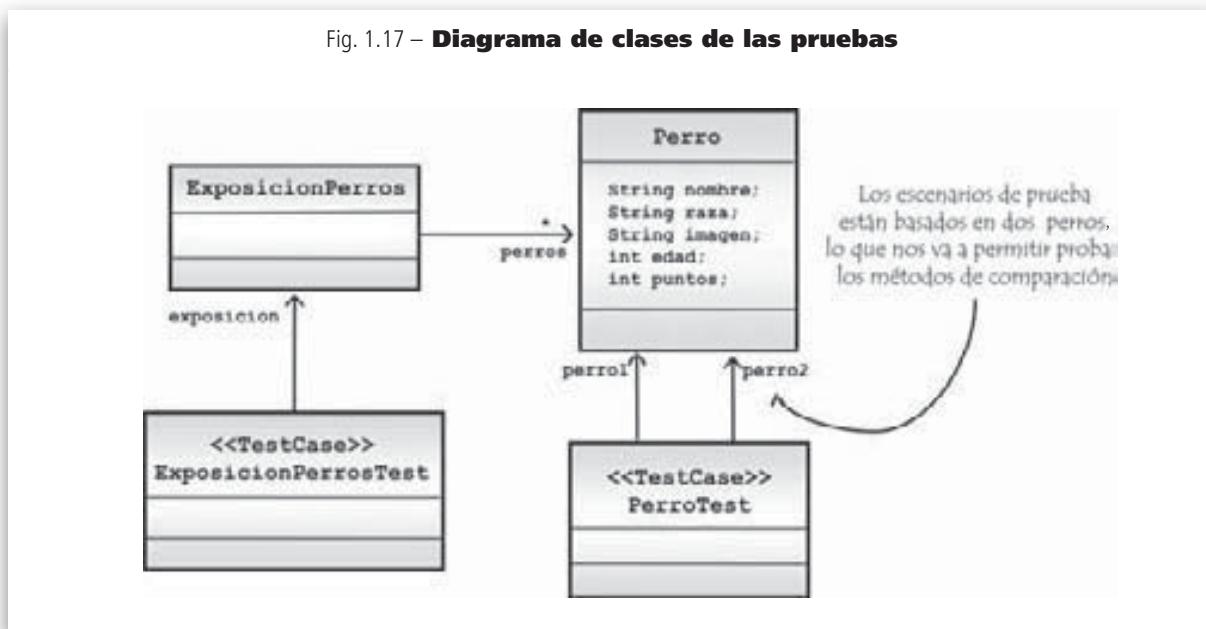
La imagen del perro (nombre del archivo) está dada por una cadena no nula de caracteres (`imagen != null`)

La edad del perro es un valor positivo (`edad > 0`)

El perro tiene un número no negativo de puntos (`puntos >= 0`)

Para las pruebas tendremos dos clases, como se muestra en la figura 1.17: una para probar la clase Perro y otra para probar la clase ExposiciónPerros. El tema de las pruebas no será abordado en este caso de estudio, por lo que se recomienda revisar el contenido del CD para estudiar allí los escenarios utilizados y los casos de prueba definidos.

Fig. 1.17 – **Diagrama de clases de las pruebas**



5.4. Comparación de Objetos por Múltiples Criterios

Tanto los algoritmos de ordenamiento como los algoritmos de búsqueda se basan en la capacidad que ellos deben tener de comparar dos elementos y decidir si son iguales, mayores o menores. Sin eso no podrían trabajar. En el caso de las muestras, en las cuales teníamos valores de tipo simple (enteros), utilizamos los operadores relacionales (`==`, `<`, `>`, `<=`, `>=`, `!=`) que provee Java para tal fin. ¿Cómo hacer

ahora que queremos utilizar los mismos algoritmos, pero aplicados a objetos? ¿Cómo saber si un perro es mayor que otro cuando se quieren ordenar por raza?

Comencemos tratando el problema de la igualdad de objetos. Aquí tenemos dos niveles posibles: el primero se refiere al caso en el cual dos objetos son físicamente el mismo. Si eso es lo que queremos establecer, podemos utilizar el operador `==` de Java. Al decir `obj1 == obj2` estamos preguntando si las

variables `obj1` y `obj2` están haciendo referencia al mismo objeto en memoria. El segundo caso posible es cuándo queremos saber si dos objetos son iguales bajo algún concepto (por ejemplo, si dos perros tienen la misma raza). En ese caso, es necesario escribir un método llamado `equals()`, que indique si dos objetos son iguales sin ser necesariamente el mismo. Así es, por ejemplo, como comparamos dos cadenas de caracteres, ya que la clase `String` define este método. Al decir `cad1.equals(cad2)` estamos tratando de establecer si el “contenido” de `cad1` es igual al “contenido” de `cad2`, aunque sean objetos distintos.

El problema con el que nos encontramos ahora es que el método `equals()` sólo contempla un criterio de orden, y eso podría no ser suficiente en algunos casos. La clase `String`, por ejemplo, definió otro método para verificar igualdad llamado `equalsIgnoreCase()`, que ignora si las letras están en mayúsculas o minúsculas. Lo importante, en cualquier solución que utilicemos, es que sea la misma clase la que implemente estos métodos, pues decidir si un objeto de la clase es igual a otro es su responsabilidad.

Si extendemos el problema a determinar si un objeto es menor o mayor que otro, podemos encontrar ideas de solución en la clase `String`, la cual cuenta con los siguientes métodos:

- `compareTo(cad)` : Compara dos cadenas lexicográficamente, retornando: (1) un valor negativo si el objeto cadena es menor que el parámetro, (2) cero si las dos cadenas son iguales o (3) un valor positivo si el objeto es mayor que el parámetro.
- `compareToIgnoreCase(cad)` : Funciona igual que el método anterior, pero ignora si las letras se encuentran en mayúsculas o minúsculas.

Nosotros vamos a utilizar el mismo enfoque, generalizando la idea de tener un método de comparación por cada criterio de orden que pueda tener un objeto. En el caso de la exposición canina, los perros

se deben saber comparar por nombre, por raza, por edad y por puntos. Para cada uno de esos criterios, la clase debe proveer el respectivo método de comparación. Es así como en la clase `Perro` vamos a encontrar los siguientes métodos:

- `compararPorNombre(perro2)` : Compara dos perros usando como criterio su nombre y retorna: (1) un valor negativo si el objeto es menor que el parámetro `perro2` (en este caso, si el nombre del perro es menor que el nombre del perro que llega como parámetro), (2) cero si los dos perros tienen el mismo nombre o (3) un valor positivo si el objeto es mayor que el parámetro `perro2`.
- `compararPorRaza(perro2)` : Compara dos perros usando como criterio su raza y retorna: (1) un valor negativo si el objeto es menor que el parámetro `perro2`, (2) cero si los dos perros tienen la misma raza o (3) un valor positivo si el objeto es mayor que el parámetro `perro2`.
- `compararPorEdad(perro2)` : Compara dos perros usando como criterio su edad y retorna: (1) un valor negativo si el objeto es menor que el parámetro `perro2`, (2) cero si los dos perros tienen la misma edad o (3) un valor positivo si el objeto es mayor que el parámetro `perro2`.
- `compararPorPuntos(perro2)` : Compara dos perros usando como criterio el número de puntos obtenidos en la exposición y retorna: (1) un valor negativo si el objeto es menor que el parámetro `perro2`, (2) cero si los dos perros tienen el mismo número de puntos o (3) un valor positivo si el objeto es mayor que el parámetro `perro2`.

Fíjese que con sólo modificar estos métodos de comparación, podemos ordenar de manera ascendente o descendente un grupo de perros, sin necesidad de modificar el algoritmo de ordenamiento.

En el ejemplo 14 mostramos la implementación de los métodos de comparación para el programa de la exposición canina.

Ejemplo 14

Objetivo: Escribir los métodos de comparación de una clase usando distintos criterios.

En este ejemplo implementamos los cuatro métodos de comparación que necesita la clase Perro, cuya especificación se dio anteriormente.

```
public int compararPorNombre( Perro p )
{
    return nombre.compareToIgnoreCase( p.nombre );
}
```

- Este método compara dos perros usando como criterio su nombre.
- La solución consiste en usar los nombres de los dos perros y delegar esta responsabilidad al método compareToIgnoreCase () de la clase String.
- Puesto que el parámetro "p" pertenece a la misma clase en la cual estamos definiendo el método (clase Perro), tenemos derecho a hacer referencia de manera directa a sus atributos. En este caso, el atributo "nombre" contiene el nombre del perro sobre el cual se hace la invocación del método y el atributo "p.nombre" hace referencia al nombre del perro que llega como parámetro.

```
public int compararPorRaza( Perro p )
{
    return raza.compareToIgnoreCase( p.raza );
}
```

- Este método compara dos perros usando como criterio su raza.
- Utilizamos el mismo tipo de solución del método anterior, pero usando el atributo que contiene la raza de cada uno de los perros.

```
public int compararPorPuntos( Perro p )
{
    if( puntos == p.puntos )
        return 0;
    else if( puntos > p.puntos )
        return 1;
    else
        return -1;
}
```

- Este método compara dos perros usando como criterio el número de puntos conseguidos en la exposición.
- Si son iguales retorna 0, si el perro tiene más puntos que el parámetro p retorna 1, en cualquier otro caso retorna -1.

```
public int compararPorEdad( Perro p )
{
    if( edad == p.edad )
        return 0;
    else if( edad > p.edad )
        return 1;
    else
        return -1;
}
```

- Este método compara dos perros usando como criterio su edad.
- Si tienen la misma edad retorna 0, si el perro es más viejo que el parámetro p retorna 1, en cualquier otro caso retorna -1.

5.5. Métodos de Ordenamiento y Búsqueda de Objetos

En esta sección nos vamos a dedicar a adaptar los métodos de ordenamiento y búsqueda que vimos en el caso anterior para que puedan trabajar sobre objetos y puedan manejar distintos criterios de

ordenamiento. En el ejemplo 15 mostraremos la adaptación de la técnica de ordenamiento por inserción, aplicada al problema de ordenar los perros por puntaje. Luego plantearemos en términos de tareas al lector la adaptación de los demás algoritmos de ordenamiento y el algoritmo de búsqueda binaria.

Ejemplo 15



Objetivo: Mostrar la adaptación del algoritmo de ordenamiento por inserción al caso en el cual se deban manejar objetos.

En este ejemplo se presenta el método de la clase `ExposicionPerros` que ordena un vector de objetos en el orden definido por el método `compararPorPuntos()` de la clase `Perro`.

```
public void ordenarPorPuntos( )
{
    for( int i = 1; i < perros.size( ); i++ )
    {
        Perro porInsertar = ( Perro )perros.get( i );
        boolean termino = false;

        for( int j = i; j > 0 && !termino; j-- )
        {
            Perro actual = ( Perro )perros.get( j - 1 );

            if( actual.compararPorPuntos( porInsertar ) > 0 )
            {
                perros.set( j, actual );
                perros.set( j - 1, porInsertar );
            }
            else
                termino = true;
        }
        verificarInvarianto( );
    }
}
```

Al tratar de adaptar el algoritmo de ordenamiento por inserción que escribimos antes, vamos a encontrar tres dificultades.

La primera es que la sintaxis para recuperar un elemento es más pesada, por lo que es conveniente agregar variables temporales ("porInsertar" y "actual").

La segunda es que no podemos asignar directamente los valores; luego debemos pasar por el método `set()` de la clase `ArrayList`.

La tercera es que no podemos comparar los objetos con el operador relacional `==`, lo que nos obliga a utilizar el método `compararPorPuntos()`.

Del resto, la traducción es directa.

Tarea 19

Objetivo: Adaptar los métodos de ordenamiento y búsqueda binaria al caso en el cual deben manipular objetos

Escriba los métodos de la clase `ExposicionPerros` que se piden a continuación.

1. Implemente el método que ordena por raza el vector de perros, usando la técnica de intercambio (burbuja).

```
public void ordenarPorRaza( )  
{
```

```
}
```

2. Implemente el método que ordena por edad el vector de perros, usando la técnica de selección.

```
public void ordenarPorEdad( )  
{
```

```
}
```

3. Implemente el método que hace búsqueda binaria por nombre en el vector de perros y retorna la posición en el vector en el que se encuentra o -1 si no hay ningún perro con ese nombre.

```
public int buscarBinarioPorNombre( String nombre )
{
}
```

5.6. Manejo de Grupos de Valores en la Interfaz de Usuario

En las interfaces de usuario más simples se utilizan componentes gráficos para mostrar a la persona información individual. Estos componentes corresponden principalmente a etiquetas y zonas de texto. En algunos casos, como en el de la exposición canina, necesitamos utilizar componentes que permitan visualizar grupos de elementos, ya sea para pedirle al usuario que seleccione algo (el perro del cual desea información) o simplemente para mos-

trarle un conjunto de datos (la lista de perros de la exposición).

En esta sección vamos a estudiar dos componentes gráficos que, manejados en conjunto, nos van a permitir manejar listas de elementos: la clase `JList` y la clase `JScrollPane`. Vamos a presentar el tema contestando cuatro preguntas y utilizando como ejemplo el código de la interfaz de usuario del caso de estudio.

- ¿Cómo crear y configurar una lista gráfica?

```
public class PanelListaPerros extends JPanel
{
    // -----
    // Atributos
    // -----
    private JList listaPerros;
    private JScrollPane scroll;
```

- Para ilustrar la declaración, creación y configuración de la lista gráfica, mostraremos el código del panel que lo contiene.
- Al igual que con los demás componentes gráficos, debemos declarar un atributo por cada uno que queramos incluir en la interfaz.
- Declaramos una lista gráfica (`listaPerros`) y un componente de desplazamiento (`scroll`).

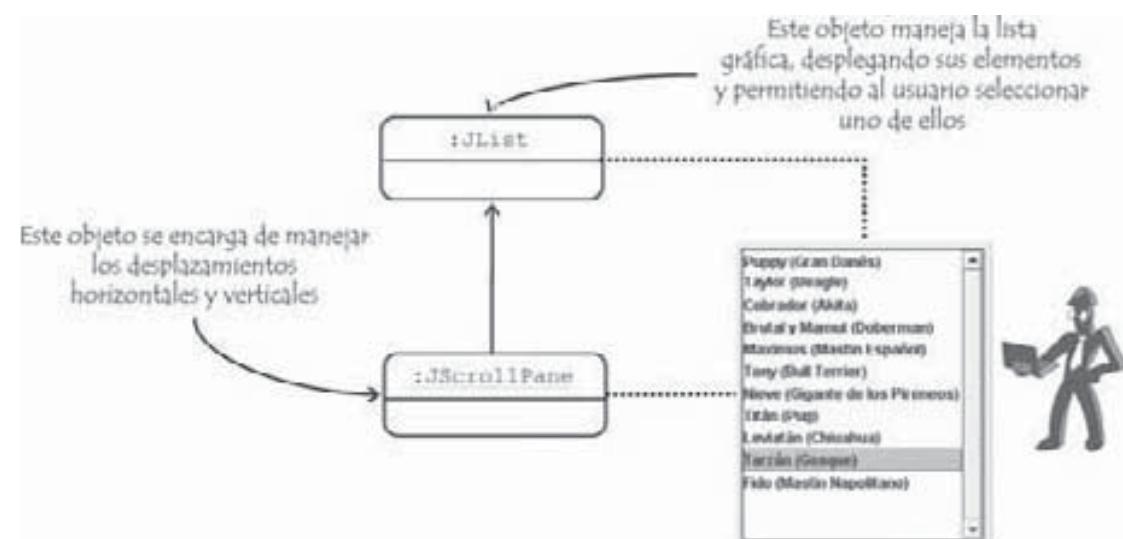
```

public PanelListaPerros( )
{
    ...
    listaPerros = new JList( );
    scroll = new JScrollPane( listaPerros );
    listaPerros.setSelectionMode(
        ListSelectionModel.SINGLE_SELECTION );
    scroll.setHorizontalScrollBarPolicy(
        JScrollPane.HORIZONTAL_SCROLLBAR_NEVER );
    scroll.setVerticalScrollBarPolicy(
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS );
    ...
    add( scroll, BorderLayout.CENTER );
}

```

- En el constructor del panel creamos inicialmente cada una de las instancias.
- Al crear la instancia del componente de desplazamiento pasamos como parámetro la lista gráfica que debe contener. En la figura 1.18 se ilustra la relación entre las instancias de estas dos clases.
- Después de creadas las instancias pasamos a configurarlas. Con el método setSelectionMode() y el parámetro SINGLE_SELECTION indicamos que es una lista gráfica de selección simple (sólo se puede escoger un elemento).
- Con los métodos setHorizontalScrollBarPolicy() y setVerticalScrollBarPolicy() indicamos que no queremos una barra de desplazamiento horizontal, pero sí una barra de desplazamiento vertical.
- Sólo agregamos al panel el componente de desplazamiento, puesto que la lista gráfica ya está incluida dentro de éste.
- Existen muchos otros métodos de configuración en estas dos clases. Invitamos al lector a visitar la respectiva documentación y a ensayar su efecto sobre el programa del caso de estudio.

Fig. 1.18 – Relación entre la clase **JList** y la clase **JScrollPane**



- ¿Cómo asignarle datos a una lista gráfica?

```
public void refrescarLista( ArrayList nuevaLista )
{
    listaPerros.setListData( nuevaLista.toArray( )
);

    listaPerros.setSelectedIndex( 0 );
}
```

■ Vamos a crear un método en la clase del panel, que va a utilizar la información que recibe como parámetro (un vector) para construir la lista de elementos que debe visualizar. Este método se va a llamar desde la ventana principal cada vez que un requerimiento cambie la lista de perros.

■ El método `setListData()` de la clase `JList` recibe como parámetro un arreglo de objetos, y a partir de esta información despliega cada uno de ellos en la interfaz. Utilizamos el método `toArray()`, que convierte un `ArrayList` en un arreglo.

■ Con el método `setSelectedIndex()` le decimos a la lista que el primer elemento que ella contenga debe aparecer como seleccionado.

- ¿Cómo mostrar en la lista la información de un objeto? O planteado de otra manera, ¿cómo puede hacer la lista gráfica para saber la manera de presentar cada uno de los objetos que contiene? (Por ejemplo, si al recibir el vector de perros queremos que muestre de cada uno de ellos el nombre y entre paréntesis la raza.)

```
public class Perro
{
    // -----
    // Atributos
    // -----

    private String nombre;
    private String raza;

    ...

    public String toString( )
    {
        return nombre + " (" + raza + ")";
    }
}
```

■ La respuesta a la última pregunta es que esta responsabilidad la tiene que asumir la clase de los objetos que van a ser incluidos en la lista gráfica. En nuestro caso, la clase `Perro`.

■ La clase `JList` se va a contentar con invocar el método `toString()` sobre cada uno de los objetos que va a presentar en su interior. Si la clase no tiene definido dicho método, se utiliza un método por defecto que tiene Java.

■ En nuestro caso, el método `toString()` de la clase `Perro` retorna el nombre del perro, concatenado con la raza del perro entre paréntesis.

- ¿Cómo tomar el elemento que el usuario seleccionó?

```
public class PanelListaPerros extends JPanel
    implements ListSelectionListener
{
    // -----
    // Atributos
    // -----
```

■ Debemos manejar el panel como un panel activo: esto implica una asociación hacia la ventana principal inicializada en el constructor.

```

private InterfazExposicionCanina principal;
private JList listaPerros;
...

public PanelListaPerros( InterfazExposicionCanina v )
{
    principal = v;
    ...
    listaPerros.addListSelectionListener( this );
    ...
}

public void valueChanged( ListSelectionEvent e )
{
    if( listaPerros.getSelectedValue( ) != null )
    {
        Perro p = ( Perro )listaPerros.getSelectedValue( );
        principal.verDatos( p );
    }
}

```

Para que un panel pueda manejar los eventos de una lista gráfica, debe incluir en su declaración la cláusula “implements ListSelectionListener”, e implementar el método valueChanged(), con la firma exacta que aparece en el ejemplo.

Dicho método será invocado cada vez que el usuario cambie la selección en la lista (ya sea por un clic sobre un elemento o por un desplazamiento con las flechas).

Con el método addListSelectionListener() definimos que es el propio panel quien va a responder a los eventos de la lista.

Allí utilizamos el método getSelectedValue() para tomar el objeto que fue seleccionado por el usuario (no la cadena de caracteres que aparece desplegada, sino el objeto incluido en la lista).

También podemos usar los métodos isEmpty() para saber si hay algún valor seleccionado, clearSelection() para no dejar ningún elemento seleccionado o selectedIndex() para conocer el índice dentro de la lista del elemento seleccionado.

6. Glosario de Términos

GLOSARIO



Complete las siguientes definiciones como una manera de resumir los conceptos aprendidos en el nivel. Asegúrese de entender bien el significado de cada uno de estos términos, puesto que son la base para poder continuar con los niveles que siguen en el libro.

Requerimiento funcional:

Diagrama de clases:

Modelo del mundo:

Diagrama de objetos:

Invariante de clase:

Ordenamiento por inserción:

Instrucción assert:

Ordenamiento por intercambio:

Pruebas unitarias automáticas:

Ordenamiento por selección:

Escenario de prueba:

Búsqueda secuencial:

Caso de prueba:

Búsqueda binaria:

Framework JUnit:

Método Math.random():

Ejecutor de pruebas:

Clase JList:

Clase TestCase:

Clase JScrollPane:

7. Hojas de Trabajo



7.1. Hoja de Trabajo N° 1: Bolsa de Empleo

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

Una empresa del sector financiero quiere construir un programa para manejar su bolsa de empleo. Se espera que dicho programa le dé apoyo al proceso de selección de personal, administrando las hojas de vida de los aspirantes a los diferentes cargos en los cuales la empresa tiene vacantes. Para la bolsa de empleo es importante poder clasificar los mejores aspirantes de

acuerdo a unos criterios previamente definidos (años de experiencia, edad y profesión), haciendo la suposición de que no hay dos aspirantes con el mismo nombre. El programa sólo contempla aspirantes de cuatro profesiones posibles: "administrador", "ingeniero industrial", "contador" y "economista".

La aplicación debe permitir (1) agregar nuevas hojas de vida de aspirantes, (2) mostrar la lista de aspirantes, (3) mostrar la información detallada de un aspirante, (4) buscar por nombre del aspirante, (5) permitir ordenar la lista de aspirantes por los diferentes criterios: años de experiencia, edad y profesión; (6) localizar al aspirante con mayor experiencia, (7) localizar al aspirante más

joven, (8) contratar a un aspirante (eliminarlo de la lista de aspirantes de la bolsa) y (9) eliminar a aquellos aspirantes cuya experiencia sea menor a una cantidad de años especificada.

La interfaz de usuario que debe tener el programa es la siguiente:



La interfaz está dividida en cuatro zonas: la primera muestra una lista con los aspirantes de la bolsa de empleo que se encuentran registrados en la aplicación. La segunda muestra la información del aspirante que se seleccionó de la lista. La tercera zona muestra los campos para agregar un nuevo aspirante a la bolsa. Y por último, la cuarta zona muestra algunas de las operaciones que se pueden hacer sobre los aspirantes registrados y las diferentes opciones de ordenamiento.

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

Nombre:	R1 – Agregar una hoja de vida
Resumen:	
Entradas:	
Resultado:	

Nombre:	R2 – Mostrar la lista de aspirantes de la bolsa de empleo
Resumen:	
Entradas:	
Resultado:	
Nombre:	R3 – Consultar los datos de un aspirante
Resumen:	
Entradas:	
Resultado:	
Nombre:	R4 – Localizar a un aspirante
Resumen:	
Entradas:	
Resultado:	
Nombre:	R5 – Ordenar la lista de aspirantes por distintos conceptos
Resumen:	
Entradas:	
Resultado:	

Nombre:	R6 – Mostrar los datos del aspirante con mayor experiencia
Resumen:	
Entradas:	
Resultado:	
Nombre:	R7 – Mostrar los datos del aspirante más joven
Resumen:	
Entradas:	
Resultado:	
Nombre:	R8 – Contratar a un aspirante
Resumen:	
Entradas:	
Resultado:	
Nombre:	R9 – Eliminar a aspirantes que no tengan los años de experiencia requeridos
Resumen:	
Entradas:	
Resultado:	

Modelo conceptual. Estudie el siguiente modelo conceptual e identifique las entidades, los atributos y los métodos de cada clase, lo mismo que las relaciones entre ellas.



Invariante. Escriba las restricciones y relaciones que existen sobre los atributos y asociaciones de las dos clases del diagrama anterior. Haga explícito si se trata de una restricción proveniente del análisis o del diseño.

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

<p>Este método verifica el invariante de la clase.</p>	<pre>public class Aspirante { private void verificarInvariante() { } }</pre>
<p>Este método retorna 0 si los aspirantes tienen el mismo nombre, 1 si el nombre del aspirante es mayor que el nombre del que viene por parámetro y -1 si el nombre del aspirante es menor que el nombre del que viene por parámetro.</p>	<pre>public class Aspirante { public int compararPorNombre(Aspirante a) { } }</pre>
<p>Este método retorna 0 si los aspirantes tienen los mismos años de experiencia, 1 si el aspirante tiene más experiencia que el que viene por parámetro y -1 si el aspirante tiene menos experiencia que el que viene por parámetro.</p>	<pre>public class Aspirante { public int compararPorAniosExperiencia(Aspirante a) { } }</pre>

```

public class BolsaDeEmpleo
{
    private void verificarInvarianto( )
    {
        }

    private int contarVecesAparece( String nombre )
    {
        }

    }
}

```

Este método verifica el invariante de la clase.

Se apoya en un segundo método que cuenta el número de veces que aparece un nombre dado dentro de la bolsa de empleo.

```

public class BolsaDeEmpleo
{
    public boolean agregarAspirante( String nombre, String profesion,
                                    int experiencia, int edad,
                                    String telefono, String imagen )
    {
        }

    }
}

```

Este método agrega un aspirante a la bolsa de empleo, retornando verdadero si se pudo agregar o falso en caso contrario (porque ya existe alguien con ese mismo nombre). Utilice el constructor de la clase

Aspirante que aparece definido en el diagrama de clases. Utilice también el método auxiliar del punto anterior.

<p>Este método ordena la lista de aspirantes ascendentemente por años de experiencia, utilizando el algoritmo de selección.</p>	<pre>public class BolsaDeEmpleo { public void ordenarPorAniosDeExperiencia() { } }</pre>
<p>Este método utiliza el algoritmo de búsqueda binaria para localizar la posición dentro del vector en la cual se encuentra el aspirante cuyo nombre se recibe como parámetro. Si no hay ningún aspirante con dicho nombre, el método retorna -1.</p>	<pre>public class BolsaDeEmpleo { public int buscarBinarioPorNombre(String nombre) { } }</pre>
<p>Este método permite localizar la posición dentro del vector del aspirante más joven, sin hacer ninguna suposición sobre el orden en el cual éstos se encuentran. Si no hay aspirantes en la bolsa, este método retorna -1.</p>	<pre>public class BolsaDeEmpleo { public int buscarAspiranteMasJoven() { } }</pre>

Pruebas unitarias. Defina un escenario para cada clase y diseñe los casos de prueba para algunos de los métodos implementados en el punto anterior.

Clase Aspirante:	Escenario:		
Método:	<code>int compararPorNombre(Aspirante a)</code>		
Caso No.	Descripción	Valores de entrada	Resultado
Clase BolsaDeEmpleo:	Escenario:		

Método:	<code>int buscarAspiranteMasJoven()</code>		
Caso No.	Descripción	Valores de entrada	Resultado
Método:	<code>int buscarBinarioPorNombre(String nombre)</code>		
Caso No.	Descripción	Valores de entrada	Resultado
Método:	<code>void ordenarPorAniosDeExperiencia()</code>		
Caso No.	Descripción	Valores de entrada	Resultado
Método:	<code>boolean agregarAspirante(String nombre, String profesion, int experiencia, int edad, String telefono, String imagen)</code>		
Caso No.	Descripción	Valores de entrada	Resultado



7.2. Hoja de Trabajo N° 2: Venta de Vehículos

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

Se quiere construir un programa para un negocio de venta de vehículos usados, en el cual se ofrecen a los clientes automóviles, buses, motos y camiones.

Además del tipo de cada vehículo, se conoce la marca, el nombre del modelo, el año de producción, el número de ejes, la cilindrada, el valor y una foto de éste. Por condiciones del negocio, se estableció que no hay dos vehículos que correspondan al mismo modelo y año.

La aplicación debe permitir al usuario las siguientes operaciones: (1) obtener la lista de todos los vehículos que están a la venta, (2) obtener la información detallada de un vehículo dado, (3) agregar un nuevo

vehículo a la venta, (4) ordenar la lista de vehículos por modelo, por marca o por año; (5) hacer una búsqueda usando el modelo y el año del vehículo, (6) comprar un vehículo (eliminarlo de la lista de vehículos que están a la venta), (7) disminuir en un 10% el precio de los vehículos que tienen un valor mayor a una cantidad dada, (8) localizar el vehículo más antiguo, (9) localizar el vehículo más potente (el de más cilindrada) y (10) localizar el vehículo más barato (el de menor precio).

La interfaz de usuario que debe tener el programa es la siguiente:

En la parte superior izquierda aparece la lista de cada uno de los vehículos disponibles. Con los botones que aparecen un poco más abajo, el usuario puede ordenar esta lista por marca, por cilindrada o por antigüedad. También puede buscar un vehículo dando su modelo y su año. En la parte derecha aparecen los datos del vehículo que se encuentra seleccionado en la lista. Allí figura toda la información que se tiene del vehículo. Con los botones del panel llamado

Consultas y Operaciones el usuario puede agregar un nuevo vehículo, localizar el vehículo más económico, el más antiguo o el más potente, al igual que disminuir el precio de todos los vehículos que superan un cierto valor. Allí mismo se encuentra un botón que permite a un usuario comprar el vehículo cuya información se está mostrando, haciendo que este salga de la lista de vehículos disponibles.

Requerimientos funcionales. Especifique los 10 requerimientos funcionales descritos en el enunciado

Nombre:	R1 – Obtener la lista de todos los vehículos que están a la venta
Resumen:	
Entradas:	
Resultado:	
Nombre:	R2 – Obtener la información detallada de un vehículo dado
Resumen:	
Entradas:	
Resultado:	
Nombre:	R3 – Agregar un nuevo vehículo a la venta
Resumen:	
Entradas:	
Resultado:	
Nombre:	R4 – Ordenar la lista de vehículos por modelo, por marca o por año
Resumen:	
Entradas:	
Resultado:	

Nombre:	R5 – Hacer una búsqueda usando el modelo y el año del vehículo
Resumen:	
Entradas:	
Resultado:	
Nombre:	R6 – Comprar un vehículo dado
Resumen:	
Entradas:	
Resultado:	
Nombre:	R7 – Disminuir en un 10% el precio de los vehículos que tienen un valor mayor a una cantidad dada
Resumen:	
Entradas:	
Resultado:	
Nombre:	R8 – Localizar el vehículo más antiguo
Resumen:	
Entradas:	
Resultado:	

Nombre:	R9 – Localizar el vehículo más potente
Resumen:	
Entradas:	
Resultado:	
Nombre:	R10 – Localizar el vehículo más barato
Resumen:	
Entradas:	
Resultado:	

Modelo conceptual. Estudie el siguiente modelo conceptual e identifique las entidades, los atributos y los métodos de cada clase, lo mismo que las relaciones entre ellas.



Invariante. Escriba las restricciones y relaciones que existen sobre los atributos y asociaciones de las dos clases del diagrama anterior. Haga explícito si se trata de una restricción proveniente del análisis o del diseño.

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

```
public class Vehiculo
{
    private void verificarInvariante( )
    {
    }
}
```

Este método verifica el invariante de la clase.

```
public class Vehiculo
{
    public int compararPorCilindrada( Vehiculo v )
    {
    }
}
```

Retorna 0 si los dos vehículos tienen la misma cilindrada, 1 si el vehículo tiene mayor cilindrada que el que viene por parámetro y -1 si el vehículo tiene menor cilindrada que el que viene por parámetro.

```
public class Vehiculo
{
    public int compararPorModelo( Vehiculo v )
    {
    }
}
```

Retorna 0 si los dos vehículos tienen el mismo modelo, 1 si el vehículo tiene un modelo mayor que el que viene por parámetro y -1 si el vehículo tiene un modelo menor que el que viene por parámetro.

Este método verifica el invariante de la clase.

Se apoya en un segundo método que retorna verdadero si hay dos vehículos con los mismos modelo y año, y falso en caso contrario.

```
public class VentaVehiculos
{
    private void verificarInvariante( )
    {
    }

    private boolean buscarVehiculosModeloYAnioRepetido( )
    {
    }
}
```

Este método busca un vehículo según su modelo y año, y retorna la posición del vector en la que se encuentra. Si no encuentra ningún vehículo con ese modelo y año retorna -1.

```
public class VentaVehiculos
{
    public int buscarVehiculo( String modelo, int anio )
    {
    }
}
```

Este método agrega un nuevo vehículo a la venta. Retorna falso en caso de que ya exista un vehículo con el mismo modelo y año. Utiliza el método desarrollado en el punto anterior.

```
public class VentaVehiculos
{
    public boolean agregarVehiculo( String modelo, String marca,
                                    String imagen, String tipo,
                                    int anio, int cilindrada,
                                    int ejes, int valor )
    {
    }
}
```

Este método ordena los vehículos de manera ascendente teniendo en cuenta la cilindrada. Utiliza para hacerlo el algoritmo de inserción.

```
public class VentaVehiculos
{
    public void ordenarPorCilindrada( )
    {
        }
    }
}
```

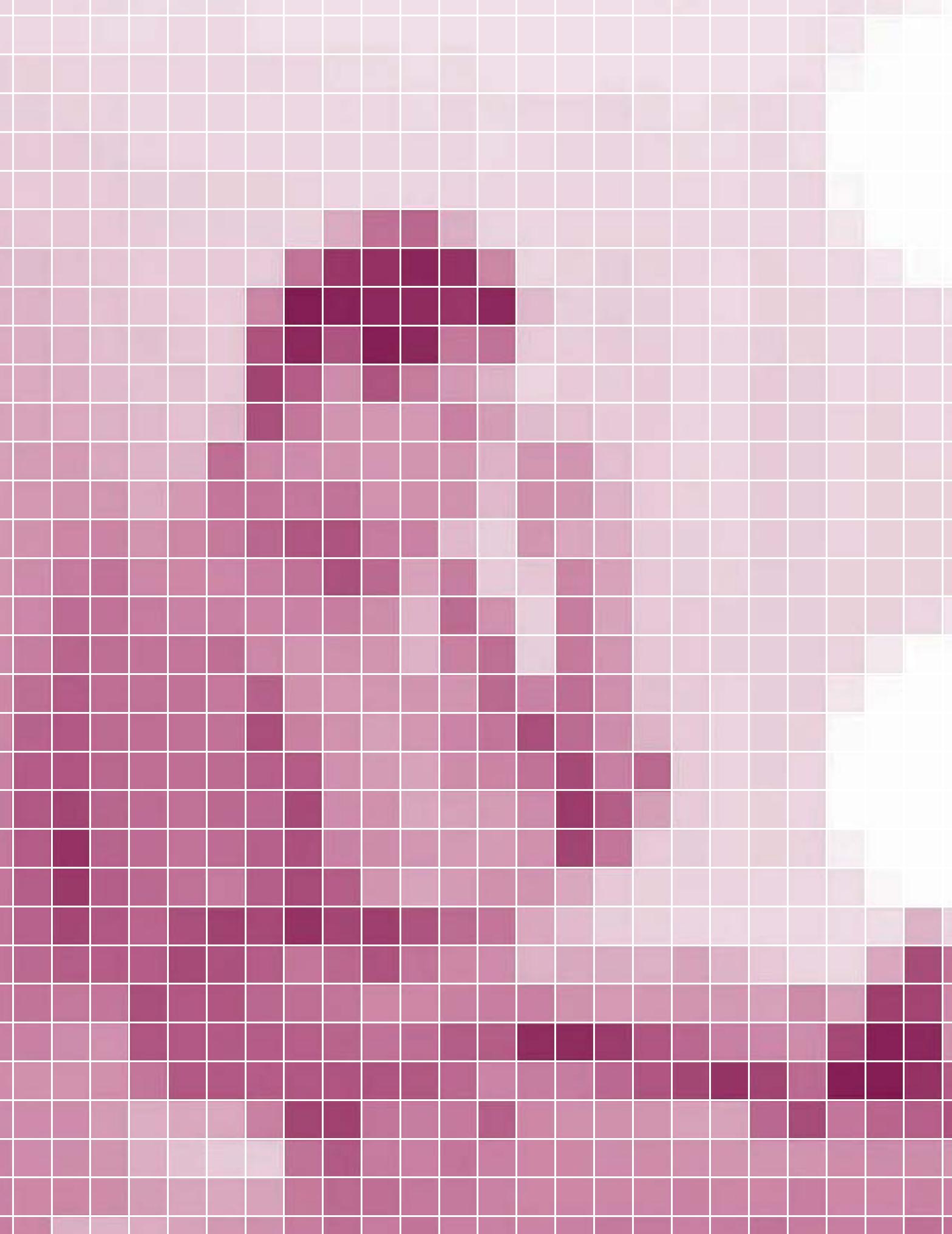
Este método retorna la posición del vehículo más barato de la venta. Para esto hace una búsqueda secuencial, puesto que no puede suponer que la estructura se encuentre ordenada por algún concepto.

```
public class VentaVehiculos
{
    public int buscarVehiculoMasEconomico( )
    {
        }
    }
}
```

Pruebas unitarias. Defina un escenario para cada clase y diseñe los casos de prueba para algunos de los métodos implementados en el punto anterior.

Clase Vehiculo:	Escenario:		
Método:	<code>int compararPorCilindrada(Vehiculo v)</code>		
Caso No.	Descripción	Valores de entrada	Resultado
Clase VentaVehiculos:	Escenario:		
Método:	<code>public boolean agregarVehiculo(String modelo, String marca, String imagen, String tipo, int anio, int cilindrada, int ejes, int valor)</code>		

Caso No.	Descripción	Valores de entrada	Resultado
Método:	<code>public int buscarVehiculoMasEconomico()</code>		
Caso No.	Descripción	Valores de entrada	Resultado
Método:	<code>public void ordenarPorCilindrada()</code>		
Caso No.	Descripción	Valores de entrada	Resultado
Método:	<code>public int buscarVehiculo(String modelo, int anio)</code>		



Nivel 2

Archivos, Serialización y Tipos de Excepción

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Hacer persistir el estado del modelo del mundo del problema al terminar la ejecución de un programa y restaurarlo al volver a ejecutar el mismo.
- Manipular archivos de texto y utilizarlos para implementar algunos requerimientos del cliente.
- Usar e implementar distintos tipos de excepción como parte de un programa, de manera que sea posible clasificar los tipos de error que se pueden presentar y asociarles en el programa distintas maneras de recuperarse ante el problema.
- Construir las pruebas unitarias automáticas para el caso de manejo de archivos, persistencia y excepciones, y utilizarlas como un mecanismo de construcción de programas correctos de manera incremental.
- Utilizar el depurador de Eclipse como una ayuda adicional en el proceso de desarrollo de programas.

2. Motivación

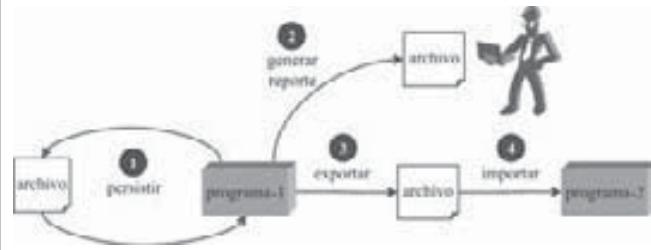
Muchos de los programas que hemos construido hasta ahora tienen una utilidad limitada, debido al hecho de que la información que introducimos en ellos no aparece la siguiente vez que los ejecutamos. ¿De qué puede servir el programa que maneja la información de la exposición canina si cada vez que lanzamos el programa obtenemos la misma lista inicial de perros? ¿Cómo podemos hacer para que la información del modelo del mundo perdure y que, al ejecutar de nuevo el programa, nos encontremos con el mismo estado que dejamos la última vez? Esta propiedad de la información se denomina **persistencia** y es una de las preguntas que contestaremos a lo largo de este nivel.

La persistencia de la información puede ser un problema complicado, sobre todo si ésta debe ser compartida de manera simultánea por un gran número de usuarios, como es el caso de los grandes sistemas de información. Piense, por ejemplo, en la cantidad de datos y en la cantidad de usuarios que un sitio de ventas por Internet puede tener en un momento dado. Para construir ese tipo de programas debemos utilizar algunos conceptos y herramientas que iremos viendo a lo largo del libro y otros que están fuera del alcance del mismo. Por ahora comenzaremos con esquemas muy simples de persistencia, en los cuales utilizamos archivos secuenciales para almacenar y manejar la información de un programa, el cual será usado de manera individual por un usuario.

En el nivel anterior utilizamos un archivo de propiedades como una manera de definir el estado inicial del programa de la exposición canina. Esta aproximación tiene muchas limitaciones para poder ser utilizada como un mecanismo general de persistencia de información. Los archivos secuenciales que vamos a estudiar en este nivel son una forma más general de almacenar y recuperar información de un medio magnético (disco duro, memoria USB, CD, DVD, etc.). En la figura 2.1 mostramos los cuatro usos que le vamos a dar a estos archivos y sobre los cuales trabajaremos más adelante: (1) hacer persistir la información de un programa para que él mismo la pueda utilizar

en el momento de volverse a ejecutar, (2) generar un reporte con información de un programa (crear, por ejemplo, un archivo para auditoría, con las ventas de un día o crear un reporte de los cien productos más vendidos), (3) exportar información para que otro programa la pueda utilizar (crear, por ejemplo, un archivo en un formato que pueda ser leído por un programa de adquisiciones, con los productos de los cuales no hay disponibilidad en bodega) e (4) importar información contenida en un archivo (leer, por ejemplo, de un archivo, en un formato predefinido, los pedidos de un distribuidor en otro país). Los archivos se deben ver como una pieza fundamental de los programas de computador, y uno de los principales objetivos de este nivel es estudiar la manera de manipularlos.

Fig. 2.1 – **Uso de los archivos para compartir total o parcialmente información**

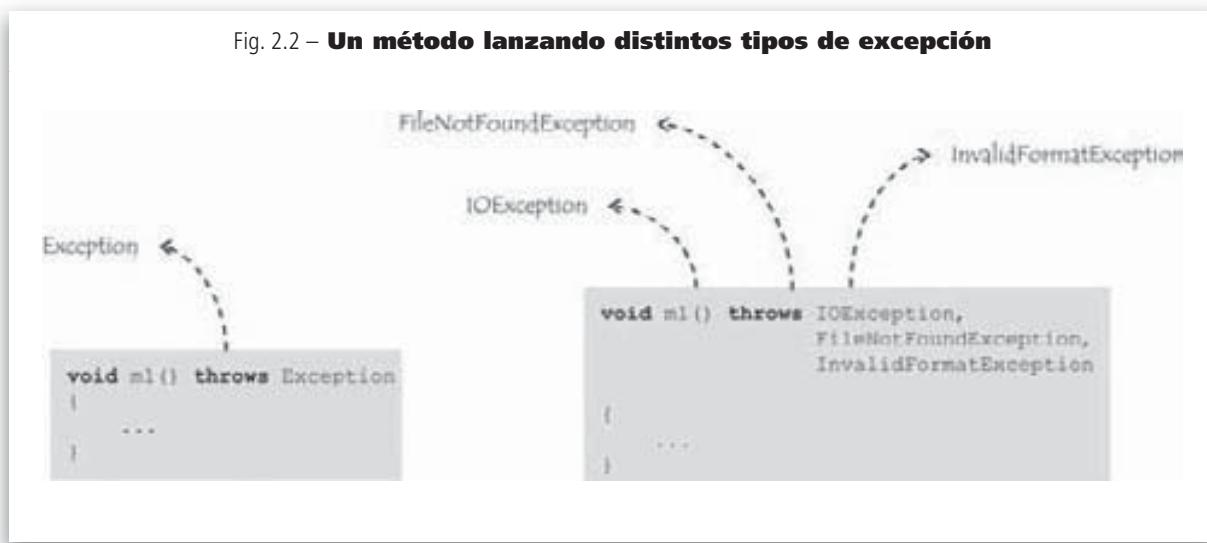


El segundo punto en el que trabajaremos en este nivel tiene que ver con las excepciones, tema que fue introducido en un curso anterior. Allí se presentaron las excepciones como la materialización –en forma de objeto– de un error detectado en algún punto de un programa, el cual podía ser lanzado y atrapado en otras partes del código, permitiendo así su recuperación. Este mecanismo es muy poderoso, pero del modo como fue estudiado no permite distinguir de manera conveniente entre distintos tipos de error y eso, en algunos casos, puede ser un problema. Piense, por ejemplo, en un método que tiene que leer información de un archivo para procesar una lista de pedidos. Allí se pueden presentar tres errores distintos:

(1) el archivo cuyo nombre suministró el usuario no existe, (2) hay un error en el archivo, lo cual nos impide leerlo (el CD está rayado por ejemplo) o (3) la información del archivo no se encuentra en el formato pedido. Si tenemos en cuenta que la recuperación de

cada uno de estos errores se debe hacer de manera distinta y, a lo mejor, en puntos distintos del programa, es conveniente que el método pueda informar cuál de ellos se presentó, en lugar de sólo decir que hubo un problema. Esta idea se ilustra en la figura 2.2.

Fig. 2.2 – **Un método lanzando distintos tipos de excepción**



Al introducir estos nuevos temas (archivos y tipos de excepción), nos vemos en la necesidad de incluir en el proceso de desarrollo de programas la manera de hacer las pruebas unitarias automáticas. Hay que verificar, por ejemplo, que efectivamente sí se están lanzando las excepciones adecuadas cuando el contrato así lo exige. Ese es el último punto que estudiaremos en este nivel.

3. Caso de Estudio N° 1: Un Explorador de Archivos

En este caso queremos construir un programa (miniExplorer) que permita explorar el sistema de archivos almacenados en un computador y hacer búsquedas simples sobre el contenido de los directorios y archivos que allí se encuentran. Dicho programa debe utilizar la interfaz de usuario presentada en la figura 2.3. Debe comenzar la exploración

situado en la raíz del disco C del computador, mostrando los directorios y archivos que allí aparecen.

A partir de dicho momento, el programa debe ofrecer las siguientes opciones: (1) mover el explorador a uno de los subdirectorios del directorio actual que aparecen desplegados en la interfaz. En ese caso se actualiza el contenido de la ventana con la información respectiva. (2) Subir un nivel en la jerarquía de archivos, para llegar al directorio padre del actual. (3) Consultar las propiedades de un archivo que se encuentra desplegado en el explorador. En una caja de diálogo debe aparecer el nombre del archivo, su tamaño y la última fecha de modificación. (4) Crear un archivo de texto en el directorio actual, con un contenido definido por el usuario. Si por alguna razón no puede hacerlo, debe mostrar una ventana de diálogo que explique el motivo. (5) Buscar una palabra en todos los archivos que se encuentran en el directorio actual del explorador. El usuario teclea una palabra y el programa busca y presenta al usuario la lista de los archivos de texto que la incluyen (que tienen al menos una palabra que comienza por la cadena dada por el usuario). Sólo

se consideran archivos de texto aquéllos cuyo nombre termina con el sufijo .txt.

No se espera que el explorador se mantenga sincronizado con el contenido del sistema de archivos del computa-

dor. Esto es, si alguien crea o borra un archivo desde otro programa, no es necesario que el resultado se refleje en nuestra interfaz. Esto sería lo ideal, pero por ahora nos contentamos con algo más simple.

Fig. 2.3 – **Interfaz de usuario del explorador de archivos**



- En todo momento el explorador se encuentra desplegando la información del directorio actual. En la figura, el directorio actual es "C:\".
- En la parte izquierda de la ventana aparecen los archivos (en la parte superior) y los directorios (en la parte inferior) del directorio actual.
- Al hacer clic sobre uno de los archivos se despliega una nueva ventana con información acerca de éste.
- Al hacer clic sobre uno de los directorios, el explorador lo convierte en el directorio actual.
- En la zona central de la ventana aparece la opción de búsqueda de una palabra en todos los archivos de texto del directorio actual.
- En la parte derecha de la ventana aparece la opción para crear un archivo en el directorio actual, con el contenido que sea tecleado en la respectiva zona de texto.
- Con el botón **Subir** podemos navegar hacia arriba en la jerarquía de archivos del computador, hasta llegar a la raíz del sistema de directorios.

3.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?

- Navegar por el sistema de archivos del computador, obteniendo la información de los directorios y archivos presentes.

- Leer el contenido de un archivo de texto.

- Crear un archivo de texto y escribir en él.

¿Qué tenemos que aprender o reforzar?

- Aprender a usar la clase `File` de Java que permite manipular archivos.

- Aprender a usar las clases de Java que permiten la lectura del contenido de un archivo de texto.

- Aprender a usar las clases de Java que permiten crear y escribir en archivos de texto.

 Atrapar los distintos tipos de excepción que genera el <i>framework</i> de archivos de Java, para informar al usuario de la causa del error.	 Generalizar el concepto de excepción visto en niveles anteriores, para poder considerar distintos tipos de error.
 Buscar en las cadenas de caracteres provenientes de un archivo de texto las palabras que comienzan por un cierto prefijo.	 Estudiar la clase <code>String</code> para determinar qué métodos, adicionales a los ya estudiados anteriormente, nos pueden ser útiles para resolver el problema.
 Utilizar los mecanismos disponibles para construir programas correctos.	 Aprender a crear las pruebas unitarias automáticas cuando se manejan archivos en el programa.

3.2. Comprensión de los Requerimientos

Tarea 1



Objetivo: Entender el problema del caso de estudio del explorador de archivos.

(1) Lea detenidamente el enunciado del caso de estudio del explorador de archivos e (2) identifique y complete la documentación de los cinco requerimientos funcionales.

Requerimiento funcional 1	Nombre	R1 – Explorar uno de los subdirectorios
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Subir un nivel en la jerarquía de archivos
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Consultar las propiedades de un archivo
	Resumen	
	Entrada	
	Resultado	

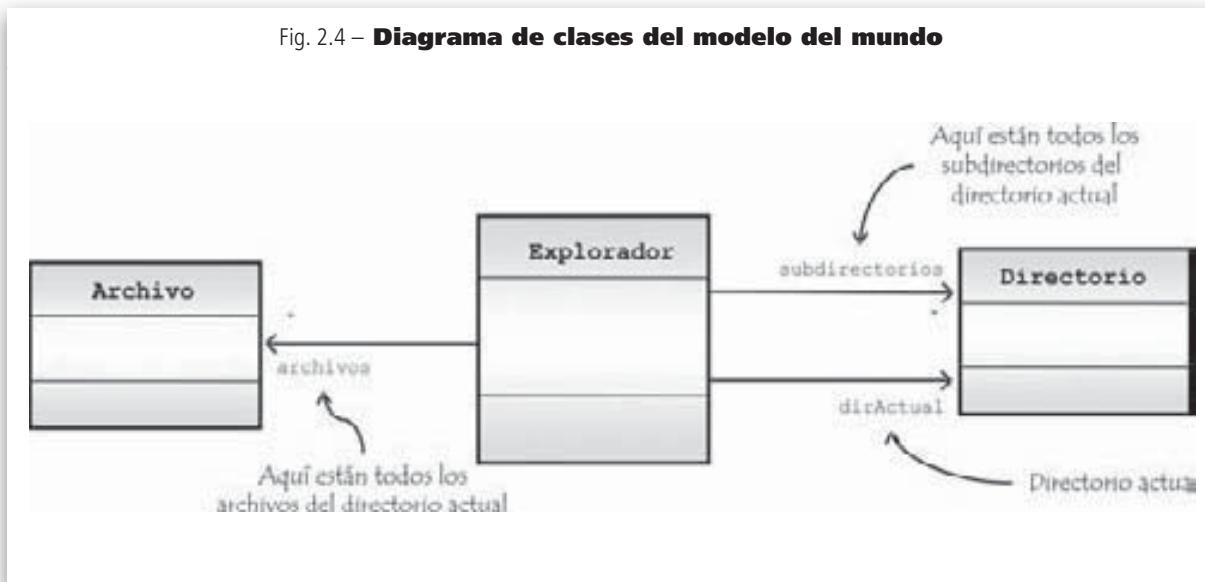
Requerimiento funcional 4	Nombre	R4 – Crear un archivo de texto en el directorio actual
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Buscar una palabra
	Resumen	
	Entrada	
	Resultado	

3.3. Modelo del Mundo del Problema

Para representar el modelo del mundo vamos a manejar tres clases distintas, de acuerdo con el diagrama que aparece en la figura 2.4. Una clase para representar el explorador de archivos (**Explorador**), una clase para representar un directorio (**Directorio**) y una clase para representar un archivo (**Archivo**). La clase

Explorador tiene dos asociaciones hacia la clase **Directorio**: una, para representar el directorio actual, y otra, para mantener el grupo de subdirectorios que allí se encuentran. También tiene una asociación hacia la clase **Archivo**, en la que maneja el grupo de archivos presentes en el directorio actual. De los detalles de las clases **Archivo** y **Directorio** nos preocuparemos más adelante.

Fig. 2.4 – **Diagrama de clases del modelo del mundo**



3.4. Expresiones Condicionales en Java

Antes de comenzar a estudiar los temas que se necesitan para resolver el caso de estudio, vamos a introducir en esta sección una sintaxis compacta que ofrece Java para escribir ciertas expresiones, la

cual nos puede ayudar más adelante para escribir de manera un poco más corta los métodos del caso de estudio.

Considere, por ejemplo, el siguiente fragmento de un método en el cual debemos calcular el valor máximo de dos variables, `var1` y `var2`, y dejar el resultado en una variable llamada `var3`:

```
public void ejemplo( )
{
    ...
    if( var1 > var2 )
        var3 = var1;
    else
        var3 = var2;
    ...
}
```

- Como parte del método, queremos dejar en la variable `var3` el valor máximo entre `var1` y `var2`.
- Para esto utilizamos la instrucción condicional “`if`” y dos asignaciones distintas.
- Este tipo de situaciones es recurrente en los programas, y utiliza al menos cuatro líneas de código.

El lenguaje de programación Java ofrece una sintaxis más compacta para expresar lo mismo. Se denominan **expresiones condicionales** (porque su valor

va a depender de una condición), y utilizan el operador “`?`”, tal como se muestra a continuación para el mismo ejemplo anterior:

```
public void ejemplo( )
{
    ...
    var3 = ( var1 > var2 ) ? var1
    : var2;
    ...
}
```

- En las expresiones condicionales, los paréntesis que rodean la expresión lógica son opcionales, pero facilitan la lectura al programador.
- En el ejemplo, la expresión que se le va a asignar a la variable `var3` depende de una condición (`var1 > var2`). Si la condición es verdadera, la expresión que se debe usar es “`var1`”. Si la condición es falsa, la expresión que se debe usar es “`var2`”.
- Expresamos lo mismo que en el ejemplo anterior, pero de una manera más compacta y fácil de entender.

Ejemplo 1



Objetivo: Ilustrar el uso de las expresiones condicionales.

Este ejemplo muestra algunos usos posibles de las expresiones condicionales, utilizando para esto una clase simple llamada `Ejemplo`.

```
public class Ejemplo
{
    private int valor1;
    private int valor2;
    private String cadena;
    private boolean condicion;
}
```

- La clase `Ejemplo` tiene cuatro atributos: dos enteros (`valor1` y `valor2`), uno de tipo cadena de caracteres (`cadena`) y uno de tipo lógico (`condicion`).
- Supondremos en adelante que los atributos han sido inicializados adecuadamente por el constructor.

```

public int valorAbsoluto( )
{
    return ( valor1 >= 0 ) ? valor1
    : -valor1;
}

public void informeConsola( )
{
    System.out.println( condicion ?
    "OK" : "ERROR" );
}

public void cambiar( )
{
    valor1 = valor2 > 0 ? valor1 +
    1 : valor1 - 1;
}

public String darMinusculas( )
{
    return cadena != null ? cadena.
   toLowerCase( ) : "";
}

```

 Este método calcula y retorna el valor absoluto del atributo "valor1".

 Este método imprime un mensaje por consola, dependiendo del valor del atributo condicion.

 Este método modifica el atributo "valor1": si "valor2" es mayor que cero, lo incrementa en uno. En caso contrario le resta uno.

 Este método convierte una cadena a minúsculas, teniendo en cuenta que si la cadena es nula debe retornar la cadena vacía.



Las expresiones condicionales permiten escribir métodos más compactos, y se deben usar siempre que no comprometan la claridad del código.

3.5. Manipulación Básica de Archivos

Lo primero que debemos aprender es la manera de recuperar información del sistema de 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.

Para representar un archivo o directorio existe una clase llamada *File* en el *framework* que provee Java, que ya utilizamos en el nivel anterior, pero que debemos estudiar un poco más a fondo en este nivel. Los siguientes son algunos de los métodos de esta clase que vamos a necesitar:

- *File(nombreCompleto)* : Es uno de los constructores de la clase. Permite crear una instancia a partir del nombre completo del archivo o directorio que se quiere representar.
- *File(directorio, nombreCorto)* : Este constructor permite crear una instancia a partir del objeto de la clase *File* que representa el directorio en el que debe estar el archivo y su nombre corto.
- *length()* : Este método funciona únicamente para archivos y retorna el número de bytes que un archivo contiene. Si el archivo no existe, el método retorna cero.

- `getAbsolutePath()` : Retorna el nombre completo del archivo o directorio, incluyendo su ruta.
- `getName()` : Retorna el nombre corto del archivo o directorio.
- `lastModified()` : Retorna un valor de tipo `long` que representa el momento en el cual el archivo o directorio fue modificado por última vez. Este valor corresponde al número de milisegundos contados a partir del 1 de enero de 1970. Si el archivo o directorio no existe, retorna cero.
- `createNewFile()` : Crea un nuevo archivo vacío con el nombre especificado por el objeto `File`, sólo si un archivo con ese nombre no existe previamente. Este método retorna verdadero si el archivo fue creado y falso en caso contrario.
- `listFiles()` : Retorna un arreglo de objetos de la clase `File` que contiene todos los elementos (directorios y archivos) presentes en el directorio. Si el objeto no es un directorio, este método retorna `null`.
- `isDirectory()` : Indica si el objeto representa un directorio.
- `isFile()` : Indica si el objeto representa un archivo.

En esta clase se define también la siguiente constante pública:

- `File.separator`: Representa el carácter que en el nombre de un archivo separa los directorios de la ruta y el nombre corto del archivo. Típicamente corresponde a la cadena "/" o "\", dependiendo del sistema operacional. El uso de esta constante ayuda a que el programa que desarrollemos sea portable.

Contando con los métodos anteriores, ya es posible desarrollar algunos de los métodos que necesitamos para el caso de estudio, los cuales se muestran en el ejemplo 2.

Ejemplo 2



Objetivo: Ilustrar el uso de los métodos de la clase `File`.

En este ejemplo se presentan algunos de los métodos que necesitamos para implementar nuestro miniExplorer.

```
public class Archivo
{
    // -----
    // Atributos
    // -----
    private File archivo;

    public Archivo( String pRuta )
    {
        archivo = new File( pRuta );
    }

    public long darTamanio( )
    {
        return archivo.length();
    }
}
```

Vamos a tener una asociación hacia la clase `File`, de manera que podamos aprovechar todas las funcionalidades ofrecidas por esa clase.

El constructor de la clase recibe como parámetro una cadena de caracteres con el nombre completo del archivo y, con este valor, crea la respectiva instancia de la clase `File`.

El método que calcula el número de *bytes* del archivo delega simplemente esta responsabilidad al método `length()` de la clase `File`.

```
public String darRuta( )
{
    return archivo.getAbsolutePath( );
}
```

Utilizamos el método `getAbsolutePath()` para obtener el nombre completo del archivo.

```
public Date darFechaUltimaModificacion( )
{
    return new Date( archivo.lastModified( ) );
}
```

Este método retorna un objeto de la clase `Date` (paquete `java.util`), que representa la fecha y la hora de la última modificación del archivo.

```
public class Explorador
{
    // -----
    // Atributos
    // -----
    private Directorio dirActual;
    private ArrayList subdirectorios;
    private ArrayList archivos;
```

La clase `Explorador` tiene tres asociaciones, de acuerdo con el diagrama de clases antes presentado: una hacia el directorio actual, otra hacia un vector de subdirectorios y la última hacia un vector de archivos.

```
public Explorador( )
{
    dirActual = new Directorio( );
    actualizarInformacion( );
```

El constructor se encarga de inicializar el directorio actual en la raíz del disco C, tal como lo pide el enunciado del caso.

- El método comienza creando los dos vectores vacíos (de subdirectorios y de archivos), invocando el constructor de la clase `ArrayList`. Luego, obtiene un arreglo con todos los elementos presentes en el directorio actual, utilizando el método `listFiles()` de la clase `File`.
- Si el arreglo resultante no es nulo, lo recorre uno por uno preguntando si es un directorio o un archivo. Si es un directorio lo agrega al vector de subdirectorios o, en caso contrario, al de archivos.

```

private void actualizarInformacion( )
{
    subdirectorios = new ArrayList( );
    archivos = new ArrayList( );

    File directorio = new File( dirActual.darRuta( ) );
    File[] elementos = directorio.listFiles( );

    if( elementos != null )
    {
        for( int i = 0; i < elementos.length; i++ )
        {
            if( elementos[ i ].isDirectory( ) )
            {
                subdirectorios.add( new Directorio( elementos[ i ].getAbsolutePath( ) ) );
            }
            else if( elementos[ i ].isFile( ) )
            {
                archivos.add( new Archivo( elementos[ i ].getAbsolutePath( ) ) );
            }
        }
    }
}

```

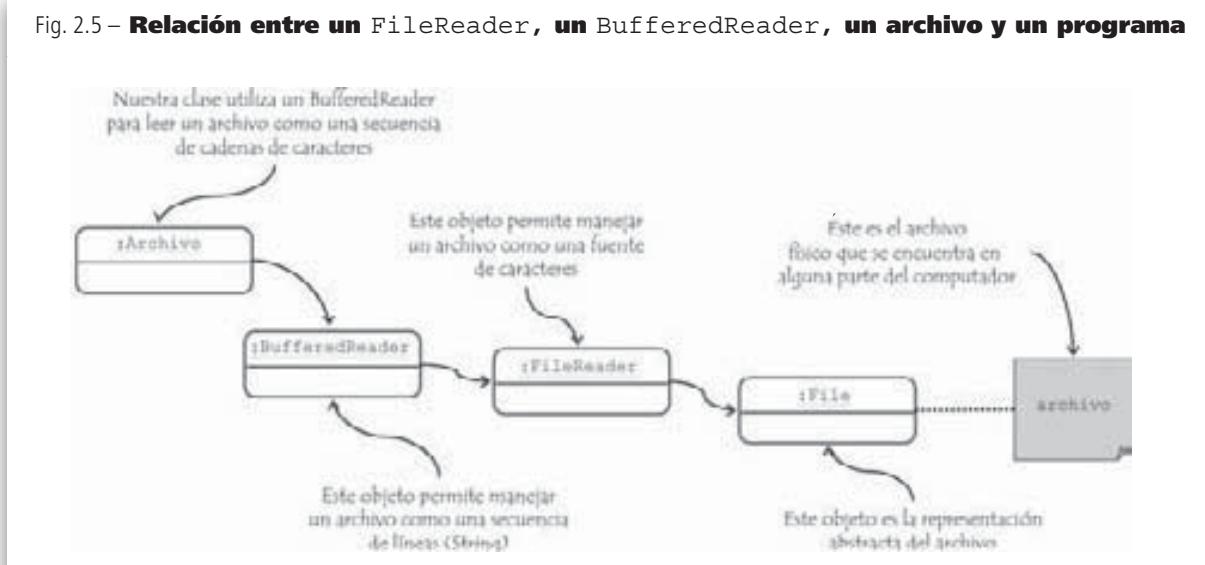
El resto de los métodos los iremos viendo a medida que vayamos presentando los temas correspondientes.

3.6. Lectura de Archivos

Para implementar la funcionalidad de búsqueda en los archivos, necesitamos poder leer el contenido de los mis-

mos como cadenas de caracteres, y luego intentar localizar en dichas cadenas la palabra que el usuario está buscando. La primera parte se estudia en esta sección, la segunda parte, en la sección siguiente.

Fig. 2.5 – Relación entre un FileReader, un BufferedReader, un archivo y un programa



Para leer el contenido de un archivo por líneas, vamos a utilizar dos clases nuevas de Java: la clase `FileReader`, que representa una fuente de caracteres desde un archivo, y la clase `BufferedReader`, que permite manipular dicha fuente como si fuera una secuencia de cadenas de caracteres (`String`). En la figura 2.5 se ilustra la relación entre estas dos clases, el archivo físico y el programa que las utiliza.

Poder manejar un archivo como una secuencia de longitud indeterminada de cadenas de caracteres facilita enormemente la escritura de los métodos, permitiéndonos utilizar un patrón de algoritmo muy simple. Existen otras maneras de leer el contenido de un archivo, pero por lo pronto nos contentaremos con presentar una de ellas.

Comencemos con la clase `FileReader`. De esta clase sólo nos interesan dos métodos: el constructor y el método que lo cierra. Los demás son de muy bajo nivel para nuestro interés actual:

- `FileReader(File)`: Este constructor nos permite crear una instancia de la clase, haciendo referencia al archivo representado por el objeto de la clase `File` que recibe como parámetro.
- `close()`: Este método nos permite cerrar el flujo de entrada de caracteres desde el archivo, anulando la relación que existía con el archivo físico.

De la clase `BufferedReader` nos interesa el método de creación, el método para cerrar la relación con la fuente de entrada de caracteres y el método para leer la siguiente cadena de caracteres presente en el archivo:

- `BufferedReader(FileReader)`: Este constructor nos permite crear una instancia de la clase, asociándola con la fuente de entrada de caracteres que recibe como parámetro.
- `close()`: Este método nos permite cerrar la relación con la fuente de caracteres.
- `readLine()`: Este método lee la siguiente línea del archivo y la retorna en una cadena de caracteres (`String`). Si ya ha llegado al final del archivo, este método retorna `null`. La cadena que retorna no trae al final los caracteres de cambio de línea que se almacenan en el archivo (*line feed* ('\n') o *carriage return* ('\r')).

Las clases que manejan archivos en Java lanzan distintos tipos de excepciones para que el programador pueda definir maneras diferentes de recuperarse al detectar una de estas situaciones. En la siguiente tabla presentamos algunas de las excepciones lanzadas por los métodos antes estudiados:

Clase	Método	Excepción	Causa
File	File(nombreCompleto)	NullPointerException	 El nombre completo del archivo o directorio es nulo.
File	File(directorio, nombreCorto)	NullPointerException	 El nombre corto del archivo o directorio es nulo.
File	createNewFile()	IOException	 Error mientras se trataba de escribir el archivo en memoria secundaria.
FileReader	FileReader(File)	FileNotFoundException	 No existe el archivo al cual hace referencia el parámetro.

FileReader	close()	IOException	 Error mientras se trataba de cerrar la fuente de entrada de caracteres.
BufferedReader	readLine()	IOException	 Error mientras se trataba de leer la siguiente línea del archivo.
BufferedReader	close()	IOException	 Error cerrando la fuente de entrada de cadenas de caracteres.

Tarea 2

Objetivo: Leer la documentación de las clases que permiten el manejo de archivos en Java.

Localice en el CD la documentación (Javadoc) de la clase `File` y conteste las siguientes preguntas:

Clase: <code>File</code>	1. ¿Qué hace el método <code>delete()</code> ? ¿Qué retorna? ¿Qué excepciones lanza?	
	2. ¿Qué hace el método <code>mkdir()</code> ? ¿Qué retorna? ¿Qué excepciones lanza? ¿En qué se diferencia del método <code>mkdirs()</code> ?	
	3. ¿Qué hace el método <code>exists()</code> ? ¿Qué retorna?	
	4. ¿Qué método se puede utilizar para cambiar el nombre de un archivo?	
	5. ¿Qué método se puede utilizar para que sobre un archivo sólo se puedan ejecutar operaciones de lectura?	

Ya estamos listos para escribir el método de lectura de archivos que necesitamos para el miniExplorer.

**Ejemplo 3**

Objetivo: Presentar el patrón de algoritmo de lectura de archivos secuenciales e ilustrar su uso en el programa del caso de estudio.

En este ejemplo se presenta el método que permite leer un archivo para buscar en su interior palabras que comiencen con un prefijo dado.

```
public class Archivo
{
    private File archivo;

    public boolean contienePrefijo( String prefijo )
                                throws IOException
    {
        boolean contiene = false;

        FileReader reader = new FileReader( archivo );
        BufferedReader lector = new BufferedReader( reader );

        String linea = lector.readLine();
        while( linea != null && !contiene )
        {
            contiene = lineaContiene( linea, prefijo );
            linea = lector.readLine();
        }

        lector.close();
        reader.close();

        return contiene;
    }
}
```

- El método recibe como parámetro el prefijo buscado y puede lanzar una excepción del tipo IOException.
- Creamos inicialmente los objetos necesarios para representar los flujos conectados con el archivo.
- Antes de entrar al ciclo leemos la primera línea del archivo y dejamos el resultado en la variable "linea".
- Si la línea leída no es nula (que indicaría que el archivo se terminó) y no ha encontrado el prefijo, procesa el contenido de la línea y luego lee nuevamente del archivo.
- El método lineaContiene() es un método privado de la clase, que veremos más adelante.
- Al final cerramos los dos flujos de entrada que habíamos abierto.

3.7. Manipulación de Cadenas de Caracteres

En la sección anterior llegamos hasta el punto en el cual se leyó el contenido de un archivo secuencial, línea por línea, y dejamos la responsabilidad de buscar la palabra pedida por el usuario a un método privado. En esta sección nos concentraremos en la implementación de éste y de todos los demás métodos necesarios en el caso de estudio, cuya solución se reduce a la manipulación de cadenas de caracteres.

Vamos a utilizar los siguientes diez métodos de la clase `String`:

- `trim()` : Elimina los blancos del comienzo y del final de la cadena, y retorna el resultado.

- `toLowerCase()` : Convierte una cadena a minúsculas y retorna el resultado.
- `toUpperCase()` : Convierte una cadena a mayúsculas y retorna el resultado.
- `indexOf(caracter)` : Retorna la posición en la cadena de la primera aparición (de izquierda a derecha) del carácter que llega como parámetro. Si no aparece, el método retorna -1.
- `lastIndexOf(caracter)` : Retorna la posición en la cadena de la última aparición (de izquierda a derecha) del carácter que llega como parámetro. Si no aparece, el método retorna -1.
- `substring(inicio, fin)` : Retorna una cadena que contiene los caracteres que

se encuentran entre las posiciones `inicio` y `fin-1`. Si cualquiera de los dos límites es inválido, este método lanza la excepción `IndexOutOfBoundsException`. Si sólo hay un parámetro, se toma la cadena que va desde dicho punto hasta el final de la cadena.

- `startsWith(prefijo)` : Indica si la cadena de caracteres comienza por el prefijo que llega como parámetro.
- `endsWith(sufijo)` : Indica si la cadena de caracteres termina con el sufijo que llega como parámetro.

- `split(separador)` : Parte la cadena en un arreglo de palabras, las cuales se encuentran separadas en la cadena por un carácter que llega como parámetro. Este método es mucho más general y poderoso de lo explicado aquí, pero para el uso que le vamos a dar es suficiente con esta explicación.

- `replace(car1, car2)` : Retorna una cadena en la que se han remplazado todas las ocurrencias del carácter `car1` por el carácter `car2`.

En la siguiente tabla se ilustra el uso de los métodos anteriores:

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

```
String cad1 = "la CASA es roja";
String cad2 = "    el perro    es BRAVO  ";
```

La expresión...	retorna...	Comentarios
<code>cad1.toLowerCase()</code>	"la casa es roja"	Retorna una nueva cadena en la que se han cambiado a minúsculas las letras que estaban en mayúsculas, sin alterar el resto de los caracteres.
<code>cad1.toUpperCase()</code>	"LA CASA ES ROJA"	Retorna una nueva cadena en la que se han cambiado a mayúsculas las letras que estaban en minúsculas, sin alterar el resto de los caracteres.
<code>cad2.trim()</code>	"el perro es BRAVO"	Elimina los blancos que hay al comienzo y al final de la cadena. Como en los dos casos anteriores, el método retorna el resultado.
<code>cad1.indexOf('A')</code>	4	El método encuentra en la posición 4 de la cadena la primera ocurrencia del carácter 'A'.
<code>cad1.indexOf('Z')</code>	-1	Retorna el valor -1 puesto que el carácter 'Z' no aparece en la cadena.
<code>cad1.lastIndexOf('A')</code>	6	La última vez que aparece el carácter 'A' es en la posición 6 de la cadena.
<code>cad1.substring(3, 7)</code>	"CASA"	El carácter de la posición de inicio (3) se incluye, pero el carácter de la posición de fin (7) se excluye.

cad1.substring(3)	"CASA es roja"	■ Es equivalente a pedir la cadena que está entre la posición 3 y la longitud total de la cadena.
cad1.startsWith("la")	true	■ Retorna verdadero, pues la cadena comienza por el prefijo "la".
cad1.endsWith("roja")	true	■ Retorna verdadero, pues la cadena termina por el sufijo "roja".
cad1.split(" ")	["la", "CASA", "es", "roja"]	■ Retorna un arreglo de cuatro posiciones de cadenas de caracteres, con cada una de las palabras que encontró en la cadena original, utilizando como separador el carácter blanco (" ").
cad1.split("a")	["l", " CASA es roj"]	■ Retorna un arreglo de dos posiciones de cadenas de caracteres, con las dos palabras que obtiene si utiliza como separador la "a".
cad1.replace(' ', '-')	"la-CASA-es-roja"	■ Retorna una cadena en la que se han remplazado todas las ocurrencias del carácter " " por el carácter "-".

Ejemplo 4

Objetivo: Implementar los distintos métodos del caso de estudio que se resuelven mediante la manipulación de cadenas de caracteres.

En este ejemplo se presentan algunos métodos de las clases Directorio y Archivo.

```
public class Directorio
{
    // -----
    // Constantes
    // -----
    public static final String RAIZ = "C:" + File.separator;
    // -----
    // Atributos
    // -----
    private String ruta;
}

private void verificarInvarianto( )
{
    assert ruta != null : "Ruta nula";
    assert ruta.startsWith( RAIZ ) : "Ruta inválida ";
}
```

■ La clase Directorio define una constante (RAIZ) con la ruta inicial del programa ("C:\") de acuerdo con el enunciado.

■ Utilizamos la constante de la clase File llamada "separator", de manera que funcione igual para LINUX y para WINDOWS.

■ Sólo hay un atributo en esta clase, el cual define la ruta completa del directorio en el sistema de archivos.

■ El invariante de la clase define dos condiciones: la ruta del directorio no es nula y la ruta del directorio comienza con la cadena "C:\".

■ Usamos el método startsWith() para validar la segunda de las condiciones del invariante.

```
public Directorio( )
{
    ruta = RAIZ;
    verificarInvariante( );
}
```

- El constructor de la clase crea el directorio en la raíz del sistema de archivos y luego verifica el invariante.

```
public boolean esRaiz( )
{
    return ruta.equals( RAIZ );
}
```

- Este método indica si el directorio corresponde a la raíz del sistema de archivos. Utiliza el método equals() de la clase String.

```
public String darNombre( )
{
    if( esRaiz( ) )

        return "";

    else
    {
        int posicion = ruta.lastIndexOf( File.separator );

        return ruta.substring( posicion + 1 );
    }
}
```

- Este método retorna el nombre del directorio.
- Si el directorio es la raíz, retorna simplemente una cadena vacía.
- Si no es la raíz, localiza en el atributo "ruta" la última aparición del carácter File.separator, utilizando el método lastIndexOf().
- Con este valor, el método retorna la cadena que comienza en la siguiente posición y va hasta el final, usando el método substring().

```
public void subirNivel( )
{
    if( !esRaiz( ) )

        int posicion = ruta.lastIndexOf( File.separator );

        ruta = ruta.substring( 0, posicion );

        if( ruta.indexOf( File.separator ) == -1 )
        {
            ruta += File.separator;
        }

        verificarInvariante( );
}
```

- Este método permite subir un nivel en la jerarquía de directorios del sistema de archivos.
- Si es la raíz el método no hace nada, puesto que es imposible subir un nivel.
- Si no es la raíz, localiza en el atributo "ruta" la última aparición del carácter File.separator.
- Luego recalcula la ruta del directorio tomando la cadena de caracteres que va desde el comienzo hasta la posición antes calculada.
- Si al hacer esta operación desaparece el único "\\" (porque llegó a la raíz), lo vuelve a agregar.

```
public class Archivo
{
    // -----
    // Atributos
    // -----
    private File archivo;
}
```

Pasamos ahora a la clase Archivo. Allí, como vimos anteriormente, sólo hay una asociación hacia la clase File.

El invariante de la clase debe afirmar dos cosas: que el atributo "archivo" no es nulo y que el nombre completo del archivo (incluida la ruta) comienza por la cadena "C:\".

```
public boolean esTexto( )
{
    String nombre = archivo.getName( );
    return nombre.toUpperCase( ).endsWith( ".TXT" );
}
```

Este método nos sirve para determinar si se trata de un archivo de texto.

Para esto tomamos el nombre del archivo (getName), lo pasamos a mayúsculas (toUpperCase) y vemos si el nombre termina con la cadena ".TXT" (endsWith).

Esto lo hacemos para contemplar a la vez el caso de los archivos que terminan en ".txt" y ".TXT".

```
private boolean lineaContiene( String linea,
                               String prefijo )
{
    linea = limpiarLinea( linea );
    String[] palabras = linea.split( " " );

    for( int i = 0; i < palabras.length; i++ )
    {
        if( palabras[ i ].toLowerCase( ).startsWith(
            prefijo.toLowerCase( ) ) )
            return true;
    }

    return false;
}
```

Este método permite establecer si una cadena de caracteres (linea) tiene al menos una palabra que comienza por un prefijo dado. Este método es llamado desde el método contienePrefijo() presentado anteriormente.

Lo primero que hace es llamar otro método (limpiarLinea), que remplaza en la línea todos los caracteres de puntuación por un carácter blanco.

Con el método split() parte la línea en palabras, utilizando como separador la cadena " ".

Recorre luego el arreglo resultante de palabras y, usando el método startsWith(), verifica si comienza por el prefijo dado.

```

private String limpiarLinea( String linea )
{
    linea = linea.replace( '.', ' ' );
    linea = linea.replace( ',', ' ' );
    linea = linea.replace( ':', ' ' );
    linea = linea.replace( ';', ' ' );
    linea = linea.replace( '!', ' ' );
    linea = linea.replace( '*', ' ' );
    linea = linea.replace( '(', ' ' );
    linea = linea.replace( ')', ' ' );

    return linea.trim( );
}

```

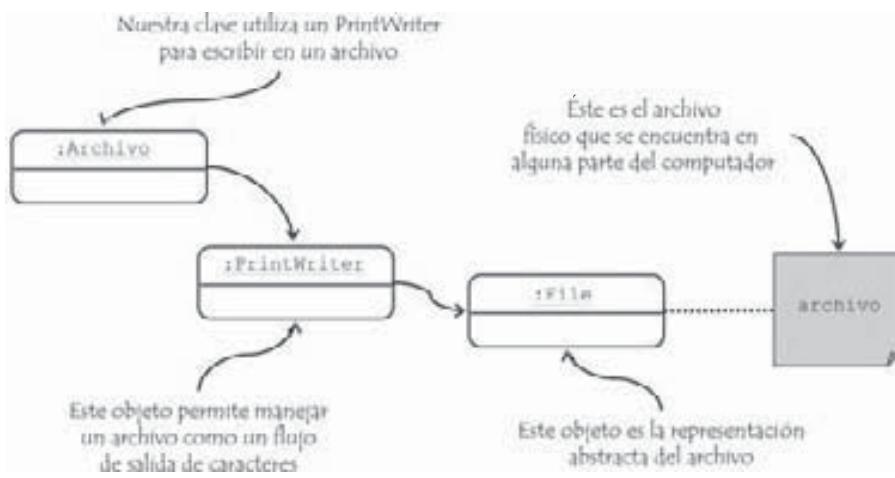
- Utilizando el método replace() remplaza los caracteres de puntuación por un carácter blanco (' ').
- Antes de retornar, utiliza el método trim() para eliminar los blancos que aparecen al comienzo y al final de la cadena.

3.8. Escritura de Archivos

Pasemos ahora al problema de escribir una cadena de caracteres en un archivo. Para esto vamos a utilizar la clase PrintWriter de Java. Un ob-

jeto de esta clase representa un flujo de salida de caracteres o cadenas, el cual está conectado físicamente con un archivo en la memoria secundaria de un computador, tal como se ilustra en la figura 2.6.

Fig. 2.6 – **Un objeto de la clase PrintWriter como una conexión con un archivo**



Puede pensarse como en un punto de conexión entre un programa y un archivo de texto, por el cual se van mandando de manera secuencial las cadenas de caracteres que queremos que se escriban en el archivo. El hecho de ser secuencial hace que sólo sea posible escribir al final del mismo, lo que nos impide borrar parcialmente o sobreescibir su contenido. Si hay que cambiarlo es necesario crear otro y

pasar uno a uno los elementos modificados al nuevo archivo.

La clase PrintWriter nos ofrece, entre otros, los siguientes métodos:

- **PrintWriter(File)**: Crea un nuevo flujo de salida conectado con el archivo descrito por el objeto de la clase File que llega como parámetro.

Si dicho archivo ya existe, borra su contenido y se prepara a escribir en él. Si el archivo no existe, lo crea. En caso de detectar cualquier problema para "conectarse" con el archivo (crearlo, borrarlo, localizarlo, etc.), este método lanza la excepción `FileNotFoundException`.

- `print(cadena)` : Escribe en el flujo de salida la cadena de caracteres que recibe como parámetro.
- `println(cadena)` : Escribe en el flujo de salida la cadena de caracteres que recibe como parámetro y le agrega al final los caracteres necesarios para marcar el final de una línea en un archivo (*line feed* ('\n') o *carriage return* ('\r')).

- `close()` : Cierra la conexión del flujo de salida con el archivo. Mientras esta conexión no se cierre, no es posible abrir un flujo de lectura desde el archivo. Al terminar la ejecución de cualquier programa en Java, se invoca automáticamente este método para todos los flujos de escritura que continúen abiertos.
- `flush()` : En algunos casos el flujo no va enviando permanentemente al archivo los caracteres que le van escribiendo, sino que los acumula en la memoria. Este método obliga al flujo a llevar al archivo todo lo que está en memoria. Antes de cerrarse un flujo (con el método `close()`) hace esto de manera automática. Es muy poco usual que un programa deba invocar este método.



Es indispensable cerrar la conexión entre un flujo de escritura y el archivo que tiene asociado, para que sea posible leerlo o utilizarlo desde otro programa.



Ejemplo 5

Objetivo: Ilustrar el uso de los métodos de la clase `PrintWriter`.

En este ejemplo se muestra el método de la clase `Archivo` del miniExplorer, encargado de escribir una cadena de caracteres en un archivo.

```
public class Archivo
{
    private File archivo;

    public void escribirArchivo( String contenido )
        throws IOException
    {
        PrintWriter escritor = new PrintWriter(archivo);
        escritor.println( contenido );
        escritor.close();
    }
}
```

El método anuncia en su encabezado que puede lanzar la excepción `IOException`.

La primera instrucción crea el flujo de escritura (`PrintWriter`) y lo conecta con el archivo.

La segunda instrucción (`println`) escribe en el archivo la cadena que el usuario tecleó en la interfaz de usuario y que le llega a este método como parámetro.

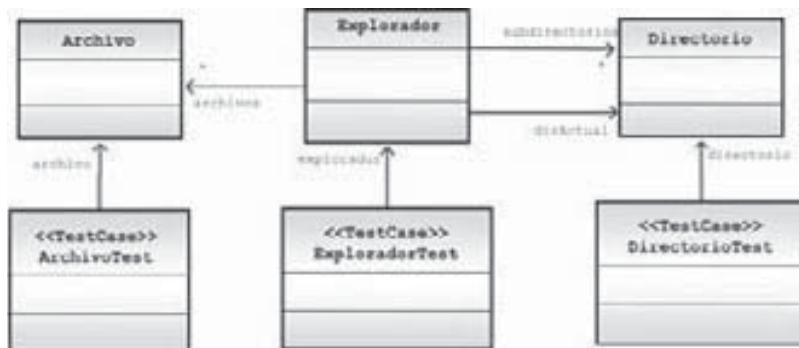
La última instrucción del método cierra el flujo de escritura.

3.9. Pruebas Unitarias con Archivos

Las pruebas unitarias para nuestro miniExplorer son un poco distintas a las que hemos estudiado hasta ahora, por cuanto los escenarios que debemos

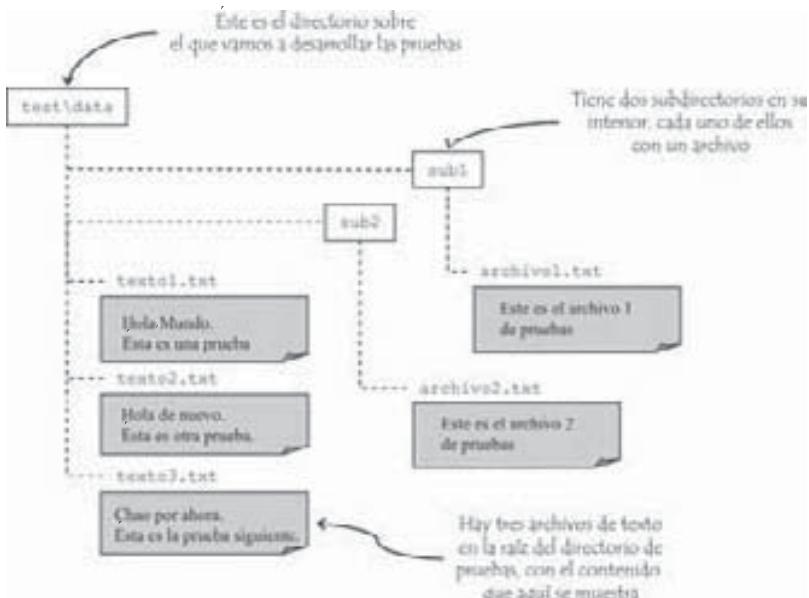
crear tienen una parte física, que incluye la construcción de directorios y archivos, los cuales situaremos dentro del directorio `test\data`. Por ahora vamos a crear tres clases de prueba, tal como se muestra en el diagrama de la figura 2.7, y ya veremos más adelante el detalle de una de ellas.

Fig. 2.7 – **Diagrama de clases de las pruebas unitarias del miniExplorer**



Vamos a trabajar sobre un único escenario físico, que consta de dos subdirectorios (`sub1`, `sub2`) y cinco archivos de texto (`texto1.txt`, `texto2.txt`, `texto3.txt`, `archivo1.txt`, `archivo2.txt`), con la estructura y contenidos mostrados en la figura 2.8.

Fig. 2.8 – **Escenario de prueba para el miniExplorer**



En el ejemplo 6 mostramos el código de la clase `ExploradorTest` que crea el escenario anterior, suponiendo que los archivos descritos anteriormente ya han sido creados en el respectivo directorio.

Ejemplo 6



Objetivo: Mostrar la declaración de la clase `ExploradorTest` y el método que construye el escenario de pruebas de la figura 2.8.

En este ejemplo se muestra la manera de construir el escenario de pruebas.

```
public class ExploradorTest extends TestCase
{
    // -----
    // Atributos
    // -----

    private Explorador explorador;

    private void setupEscenario1( )
    {
        File ruta = new File( "test" + File.separator + "data" );

        String path = ruta.getAbsolutePath();

        if( !path.startsWith( Directorio.RAIZ ) )
        {
            fail( "Ruta inicial inválida" );
        }

        explorador = new Explorador( path );
    }
}
```

Vamos a crear el escenario y a dejarlo en el atributo llamado "explorador".

Lo primero que hacemos es determinar cuál es la ruta absoluta del directorio "test\data". Para esto usamos un objeto de la clase `File`.

Puesto que la precondition del constructor exige que la ruta sea válida y que se encuentre en el disco C:, verificamos esto antes de construir la instancia.

Usamos el método `fail()` en caso de que el usuario haya instalado el programa en un directorio diferente del C:.

Finalmente creamos la instancia pasándole el nombre completo del directorio sobre el que vamos a llevar a cabo las pruebas.

Una vez creado el escenario, pasamos a definir los casos de prueba que vamos a implementar. Un resumen de estos casos aparece en la siguiente tabla:

Prueba No. 1	Objetivo: Probar que el constructor de la clase <code>Explorador</code> es capaz de recuperar correctamente la información del sistema de archivos.	
Método: <code>Explorador()</code>	Valores de entrada: Ninguno	Resultado: El directorio actual es "test\data". En la raíz hay dos subdirectorios llamados "sub1" y "sub2". En la raíz hay tres archivos llamados "texto1.txt", "texto2.txt" y "texto3.txt". No aparece ningún otro directorio o archivo en el explorador.

Prueba No. 2	Objetivo: Probar que los métodos de subir y bajar por los subdirectorios funcionan correctamente.	
Método: <code>irDirectorio()</code>	Valores de entrada: 1	Resultado: El directorio actual es "test\data\sub1". En el directorio actual se encuentra el archivo "archivo1.txt".
Método: <code>subirDirectorio()</code>	Valores de entrada: Ninguno	Resultado: El directorio actual es "test\data".
Método: <code>irDirectorio()</code>	Valores de entrada: 2	Resultado: El directorio actual es "test\data\sub2". En el directorio actual se encuentra el archivo "archivo2.txt".
Método: <code>subirDirectorio()</code>	Valores de entrada: 1	Resultado: El directorio actual es "test\data".
Prueba No. 3	Objetivo: Probar que el método de buscar por prefijo funciona correctamente.	
Método: <code>buscarPorPrefijo()</code>	Valores de entrada: "prue"	Resultado: Debe retornar un vector con los objetos de la clase Archivo que representan los archivos "texto1.txt", "texto2.txt" y "texto3.txt".
Método: <code>buscarPorPrefijo()</code>	Valores de entrada: "nuevo"	Resultado: Debe retornar un vector con el objeto de la clase Archivo que representa el archivo "texto2.txt".
Método: <code>buscarPorPrefijo()</code>	Valores de entrada: "chao"	Resultado: Debe retornar un vector con el objeto de la clase Archivo que representa el archivo "texto3.txt", puesto que en la búsqueda no debe importar si la palabra está en mayúsculas o minúsculas.
Método: <code>buscarPorPrefijo()</code>	Valores de entrada: "nunca"	Resultado: Debe retornar un vector vacío, puesto que ningún archivo contiene palabras con ese prefijo.

Tarea 3

Objetivo: Implementar los casos de prueba de la tabla anterior.

Implemente un método de la clase `ExploradorTest` por cada una de las pruebas planteadas anteriormente.

```
public class ExploradorTest extends TestCase
{
    private Explorador explorador;

    private void setupEscenario1( )
    { ... }

    public void testConstruirEscenario( )
    {

    }

    public void testSubirBajar( )
    {

    }
}
```

```
public void testBuscarPorPrefijo( )
{
}

}
```

3.10. El Componente JTextArea

El único componente nuevo que necesitamos utilizar en la interfaz de usuario es un `JTextArea`. Con este componente podemos tener una zona de texto plano, con múltiples líneas y con capacidades básicas de edición. Este componente se maneja como cualquier otro componente gráfico de Java, con la misma particularidad que vimos en el componente `JList`, que dice que si necesitamos manejar barras de desplazamiento (verticales u horizontales) debemos crear la respectiva instancia dentro de un objeto de la clase `JScrollPane`.

Los siguientes son algunos de los métodos de la clase `JTextArea` que vamos a utilizar:

- `JTextArea()` : Crea una nueva instancia de la clase, con un texto vacío asociado.
- `setLineWrap(cambioLinea)` : Indica al componente si debe manejar los cambios de línea de manera automática. Esto es, si una línea es más larga que el ancho del componente debe mostrarla partida en varias líneas (`cambioLinea` es `true`) o, si es por el contrario, debe presentarla en una sola línea y desplegar la barra horizontal de desplazamiento (`cambioLinea` es `false`).
- `getText()` : Retorna en una cadena de caracteres el contenido del componente.
- `SetText(texto)` : Reemplaza el contenido del componente por la cadena recibida como parámetro.

**Ejemplo 7**

Objetivo: Mostrar el uso de algunos de los métodos del componente `JTextArea`.

En este ejemplo se muestra el constructor de la clase `PanelNuevoArchivo`, encargado de desplegar en la interfaz la zona de texto para que el usuario teclee el contenido del archivo que quiere crear.

```
public PanelNuevoArchivo()
{
    ...
    txtTexto = new JTextArea();
    txtTexto.setLineWrap( true );
    txtTexto.setWrapStyleWord( true );

    scroll = new JScrollPane( txtTexto );
    scroll.setPreferredSize( new Dimension( 220, 220 ) );
    add( scroll, BorderLayout.CENTER );
    ...
}
```

Las tres primeras instrucciones crean el componente y determinan que éste debe manejar de manera automática el cambio de línea, partiendo las palabras cuando encuentre un espacio en blanco.

El resto de instrucciones sirven para crear el componente de barras de desplazamiento con la zona de texto asociada y luego configurarlo y agregarlo en el panel que lo va a contener.

3.11. Extensión del miniExplorer

En esta sección vamos a desarrollar algunas extensiones al explorador de archivos, las cuales serán

planteadas como tareas al lector, para practicar el uso de los métodos de manejo de archivos secuenciales de texto.

Tarea 4

Objetivo: Escribir una extensión al `miniExplorer` que permita pasar a mayúsculas todo el contenido de un archivo de texto.

Modifique el programa `n8_exploradorArchivos` que se encuentra en el CD que acompaña al libro, siguiendo las instrucciones que se dan a continuación:

1. Cree en Eclipse un proyecto a partir del archivo `n8_exploradorArchivos.zip`.
2. Estudie las clases de la interfaz de usuario. Mire la manera en que están conectados los cuatro botones de opciones (en la parte inferior de la ventana) con métodos de la clase `Explorador`. ¿Qué información reciben como parámetro esos métodos? ¿Cómo recupera esa información la clase `InterfazExploradorArchivos`?
3. Agregue en la clase `Archivo` el método `pasarAMayusculas()`, que se encarga de modificar el contenido del archivo pasando a mayúsculas todas las letras que en él aparecen. No olvide que para cambiar el contenido de un archivo se debe crear otro con los cambios, y luego, con los métodos de eliminación y cambio de nombre de un archivo, se puede hacer la sustitución.
4. Asocie con el botón "Opción 1" la funcionalidad antes descrita, modificando el método `metodo1()` de la clase `Explorador`.
5. Verifique que la extensión funciona correctamente.

Tarea 5

Objetivo: Escribir una extensión al mini Explorer que permita la encriptación del contenido de un archivo, utilizando una clave dada.

Modifique el programa `n8_exploradorArchivos` que se encuentra en el CD que acompaña al libro, siguiendo las instrucciones que se dan a continuación:

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n8_exploradorArchivos.zip`.

2. Localice en el directorio **data** el archivo **clave.txt** y edítelo. Debe ver algo del siguiente estilo:



Allí aparece para cada letra 'a' – 'z' y para cada dígito '0' – '9', el carácter con el que debe ser remplazado en el archivo resultado del proceso de encriptación. Por ejemplo, cada vez que aparezca una letra 'a' en el archivo se debe escribir la letra 'e'. La encriptación es un proceso mediante el cual la información es cifrada para que el resultado no pueda ser entendido por otra persona, a menos que conozca la manera de recuperar la información original. En nuestro caso, a menos que la otra persona conozca la clave usada, no podrá leer el contenido del archivo.

3. Agregue en la clase **Archivo** el método `encriptacion()`, que se encarga de crear un nuevo archivo, al cual se le ha aplicado la clave de cifrado antes vista. Si el archivo original se llama, por ejemplo, "arch1.txt", el archivo resultante debe quedar en el mismo directorio y se debe llamar "arch1.cripto.txt".

4. Asocie con el botón **Opción 2** la funcionalidad antes descrita, modificando el método `metodo2()` de la clase **Explorador**.

5. Verifique que la extensión funciona correctamente.

Tarea 6

Objetivo: Escribir una extensión al mini Explorer que permita partir un archivo en una secuencia de archivos más pequeños.

Modifique el programa `n8_exploradorArchivos` que se encuentra en el CD que acompaña al libro, siguiendo las instrucciones que se dan a continuación:

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n8_exploradorArchivos.zip`.

2. Agregue en la clase **Archivo** el método `partir()`, que se encarga de partir un archivo de texto en un grupo de archivos, cada uno de los cuales tiene como máximo 1.024 caracteres. Dichos archivos deben quedar en el mismo directorio del archivo original. Si, por ejemplo, el archivo se llama "arch1.txt", los archivos creados por este método se deben llamar "arch1.parte1.txt", etc. Este proceso de partir un archivo se usa con frecuencia en comunicaciones, para enviar la información contenida en un archivo como una secuencia de paquetes de un tamaño máximo dado.

3. Asocie con el botón **Opción 3** la funcionalidad antes descrita, modificando el método `método3 ()` de la clase `Explorador`.

4. Verifique que la extensión funciona correctamente.

Tarea 7



Objetivo: Escribir una extensión al `miniExplorer` que permita crear un archivo con algunas estadísticas básicas.

Modifique el programa `n8_exploradorArchivos` que se encuentra en el CD que acompaña al libro, siguiendo las instrucciones que se dan a continuación:

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n8_exploradorArchivos.zip`.

2. Agregue en la clase `Archivo` el método `calcularEstadísticas ()`, que se encarga de crear un archivo con las estadísticas que aparecen en la siguiente figura:

```

arch1.estadisticas.txt - Note...
File Edit Format View Help
miniExplorer - Estadísticas
Archivo: arch1.txt
Líneas: 345
Caracteres: 14234
a: 456
e: 563
i: 123
o: 289
u: 90
.: 234
,: 678
/: 12
(: 23
): 23

```

Allí tenemos el nombre del archivo, el número total de líneas del archivo, el número total de caracteres del archivo, el número de veces que aparece cada vocal y el número de veces que aparecen algunos signos de puntuación.

Si el archivo se llama "arch1.txt", el archivo de estadísticas se debe llamar "arch1.estadisticas.txt".

3. Asocie con el botón **Opción 4** la funcionalidad antes descrita, modificando el método `método4 ()` de la clase `Explorador`.

4. Verifique que la extensión funciona correctamente.



En este punto es muy recomendable revisar la solución completa del caso de estudio que se encuentra en el CD, y estudiar las clases de la interfaz, mirando en detalle la manera como se conectan con las clases del modelo del mundo.

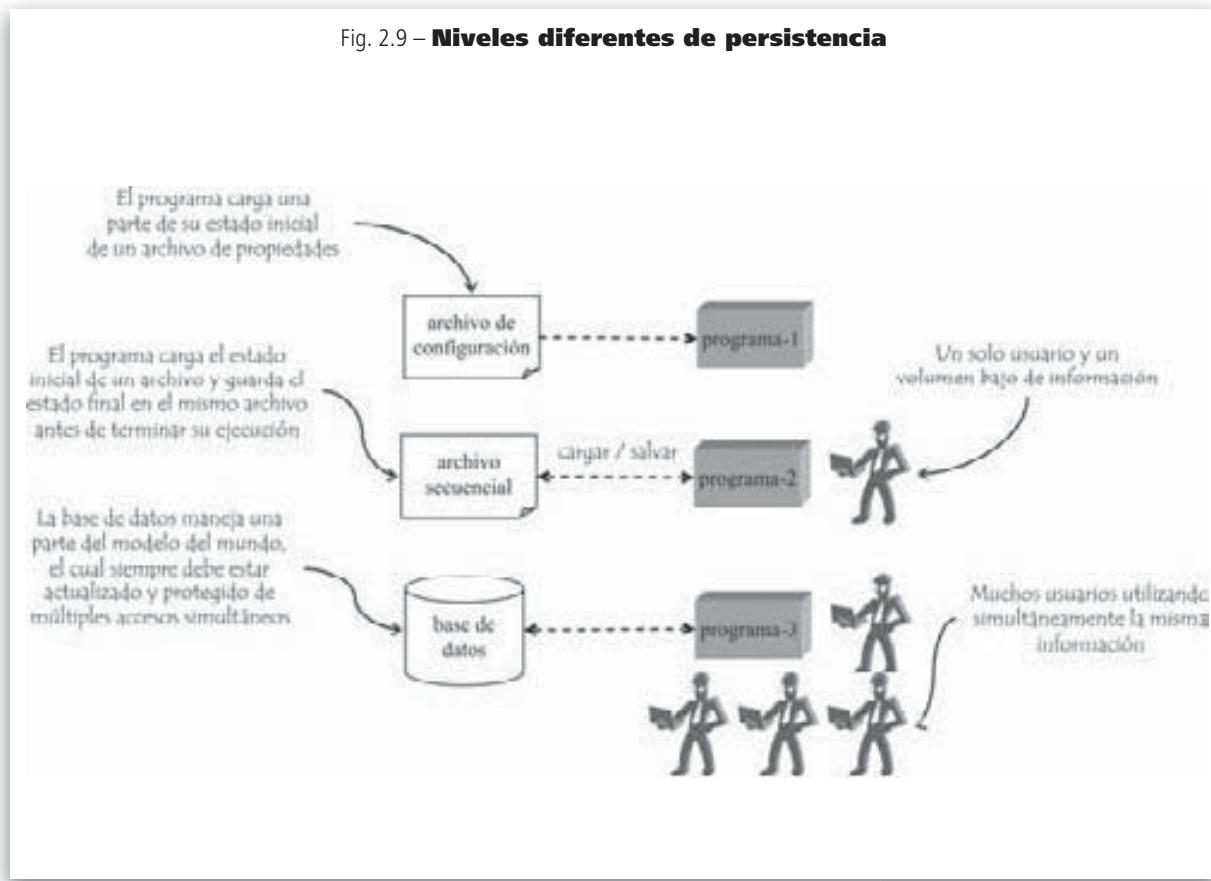
4. Visión Global de la Persistencia

La persistencia es uno de los requerimientos no funcionales más comunes de los programas de computador que se desarrollan. Es baja la utilidad de un programa si no es capaz de garantizar que los datos introducidos por el usuario se van a mantener después de terminada su ejecución. ¿Quién compraría un programa para manejar una exposición canina si toda la información que se agrega va a desaparecer al salir del programa?

La solución al problema de la persistencia puede ser muy simple o muy compleja, dependiendo principalmente de la cantidad de información que se deba almacenar, de la cantidad de usuarios simultáneos que la vayan a utilizar y de lo tolerante que deba ser el programa ante las posibles fallas de la aplicación.

Podemos imaginarnos al menos tres niveles distintos, cada uno con un esquema de solución diferente, los cuales se ilustran en la figura 2.9:

Fig. 2.9 – **Niveles diferentes de persistencia**



- En el primer nivel encontramos los programas de computador que deben partir de un estado inicial dado, pero que no son capaces de hacer persistir los cambios que se hagan durante su ejecución. El ejemplo típico son los programas que utilizan un archivo de propiedades para configurarse.
- En el segundo nivel tenemos los programas que son utilizados por un único usuario a la vez y que sólo hacen persistir la información en un archivo al terminar la ejecución del mismo (o cuando el usuario explícitamente lo ordena). Estos programas se pueden construir mediante la utilización de archivos

secuenciales como los estudiados en el caso anterior. Llamaremos "cargar" (*load*) al proceso de llevar la información del archivo a la memoria principal y "salvar" al proceso contrario.

- En el tercer nivel aparecen los programas cuya información debe ser compartida simultáneamente por varios usuarios, o cuyo volumen es demasiado alto como para poder llevar toda la información a la memoria principal. En ambos casos se deben utilizar **bases de datos**, un tema que será abordado de manera superficial algunos niveles más adelante. En esta solución la dificultad radica en que es necesario mantener toda la información en memoria secundaria actualizada y protegida de múltiples accesos. No es un problema de cargar y salvar información, sino un problema de manejar en la base de datos una parte del modelo del mundo, lo cual implica la aparición de un sinnúmero de nuevos problemas y retos.

5. Caso de Estudio N° 2: Una Tienda Virtual de Discos

En el segundo caso de estudio del nivel estamos interesados en crear una aplicación (*miDiscoTienda*) para el manejo de la información de una tienda virtual de canciones en formato MP3. Esta aplicación debe permitir visualizar un catálogo de discos y canciones que la tienda tiene a la venta. Cada disco tiene un nombre, un artista, un género y una imagen de la carátula del disco. La tienda se especializa en la venta de canciones en formato MP3, por lo que ofrece toda la información relevante de una canción al usuario. Esta información comprende: el nombre de la canción, su precio individual, la duración en minutos y segundos, el tamaño en *megabytes* (MB) y la calidad de la canción, expresada en *kilobytes* por segundo (Kbps). La interfaz de usuario que debe tener el programa se muestra en la figura 2.10.

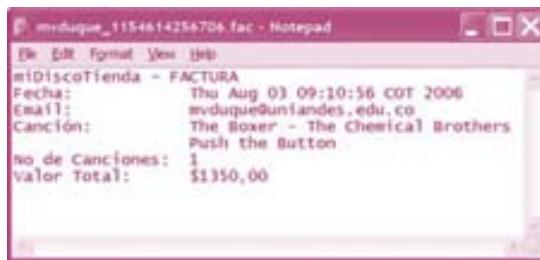
Fig. 2.10 – Interfaz de usuario de la tienda virtual de discos



- Con el botón **Agregar Disco** el usuario puede agregar un disco a la tienda virtual. El programa le pide el nombre del disco, además del resto de la información que maneja.
- En la lista plegable (comboBox) que aparece en la parte izquierda de la ventana se encuentran todos los discos de la tienda.
- Con el botón **Agregar Canción** se puede agregar una canción al disco seleccionado. En la lista plegable de la parte derecha de la ventana aparecen todas las canciones del disco seleccionado.
- Con el botón **Vender Canción** se le vende la canción seleccionada a un usuario, el cual debe teclear su dirección electrónica. El usuario recibe por correo electrónico la canción que acaba de comprar y un archivo con la respectiva factura.
- Con el botón **Cargar Pedido** se procesa el pedido de un usuario a partir de los datos contenidos en un archivo. Dicho archivo debe tener el formato definido en el enunciado.

Se espera que miDiscoTienda cuente con la siguiente funcionalidad: (1) presentar al usuario la información de una canción del catálogo (dados el nombre del disco y el nombre de la canción), (2) agregar un disco al catálogo, el cual estará sin canciones inicialmente; (3) agregar una canción a un disco (dados el nombre del disco y toda la información de la canción), (4) vender una canción a un usuario. En este caso se debe teclear la dirección electrónica del usuario para que le sea enviado por correo el respectivo archivo MP3 con la canción. Por cada venta, el programa debe crear un archivo de texto (en el directorio `data\facturas`), cuyo nombre debe ser parte de la dirección electrónica del usuario (lo que va antes del símbolo "@") seguido de un número único generado por el programa. Dicho archivo también se envía por correo electrónico al usuario. En la figura 2.11 aparece un ejemplo de una factura generada a un usuario por la venta de una canción. El programa debe llevar el registro del número de copias vendidas de cada canción.

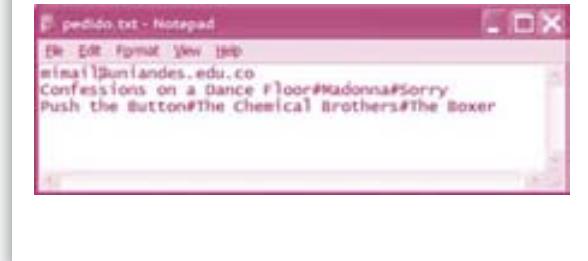
Fig. 2.11 – **Ejemplo de una factura generada por cada venta**



Por último, (5) el programa debe permitir procesar un archivo completo de pedidos de un cliente. En este caso se hace una sola factura. En la figura 2.12 aparece un ejemplo del formato que debe tener un archivo de pedidos para que pueda ser procesado automáticamente por miDiscoTienda. Allí se puede ver que en la pri-

mera línea debe ir la dirección electrónica de aquél que hace el pedido, seguido después de una línea por cada canción que el usuario quiere comprar. De cada canción debe incluir el nombre del disco seguido del carácter "#", el nombre del artista seguido de nuevo del carácter "#" y finalmente el nombre de la canción. Se piden estos tres datos para asegurarse de que no haya ninguna ambigüedad.

Fig. 2.12 – **Ejemplo de un archivo con los pedidos de un usuario**



Para que el programa sea de verdad útil, la información de la tienda debe ser persistente y el proceso debe ser completamente transparente para el usuario. Esto quiere decir que el programa debe ser capaz de guardar la información en un archivo cada vez que el usuario termina la ejecución del mismo y de utilizar dicha información cuando el usuario vuelve a ejecutarlo para reconstruir el estado del modelo del mundo. El programa no debe preguntarle al usuario el nombre del archivo, sino que lo tiene que manejar todo internamente.

En caso de cualquier error en la ejecución del programa, éste debe presentar una caja de diálogo con un mensaje claro que explique la razón del problema. La siguiente es una lista de los errores que se pueden presentar y el tipo de acción que debe realizar el programa:

Errores posibles	Manejo del error
Al intentar agregar un nuevo disco, el programa se da cuenta de que ya existe un disco con ese título.	Mensaje al usuario con el error. El nuevo disco no es agregado.
Al intentar agregar una nueva canción a un disco, el programa se da cuenta de que ya existe una canción con ese nombre.	Mensaje al usuario con el error. La nueva canción no es agregada.
Los datos dados por el usuario no son válidos (el precio no es un valor numérico, o el nombre de la canción es vacío).	Mensaje al usuario con el error. El nuevo disco o la nueva canción no son agregadas.
Al intentar escribir el archivo con la factura, hay un error de entrada / salida.	Mensaje al usuario con el error. La venta se hace de todos modos y se envía la canción al usuario sin la factura.
Al intentar leer el archivo de pedidos, no lo encuentra en el sistema de archivos.	Mensaje al usuario con el error.
Durante la lectura del archivo de pedidos, hay un error de entrada / salida.	Mensaje al usuario con el error. La venta se hace hasta donde se haya podido leer el archivo. Se genera una factura parcial con la venta.
Al intentar leer el archivo de pedidos, el programa se da cuenta de que éste no tiene el formato pedido.	Mensaje al usuario con el error. Si el error impide la venta global, se anula todo el proceso. Si no, se venden las canciones que tengan el formato pedido, y en la factura se señala cuáles canciones no se pudieron incluir en la venta por un problema de formato.
Al intentar cargar el estado inicial del modelo del mundo, hay un error de entrada / salida.	No debe comenzar la ejecución del programa y debe aparecer un mensaje de error al usuario. En un archivo de registro (<i>log</i>) se debe escribir la información del error, incluyendo la hora del problema.
Al intentar salvar el estado final del modelo del mundo, hay un error de entrada / salida.	No debe dejar que el programa termine (a menos que el usuario así lo decida explícitamente) y debe presentar un mensaje de error pidiéndole que lo resuelva. En un archivo de registro (<i>log</i>) se debe escribir la información del error, incluyendo la hora del problema.
Al hacer una venta, el usuario no suministra una dirección electrónica válida.	Mensaje al usuario con el error. La venta no se hace y no se emite ninguna factura.

5.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
 Hacer persistir la información del modelo del mundo de manera transparente para el usuario.	 Estudiar la serialización como un mecanismo que provee Java para hacer persistir información. Aprender a tomar el control del programa tan pronto el usuario decida salir del mismo, para llevar la información del mundo del problema a un archivo.
 Manejar de manera adecuada todos los tipos de error que se pueden presentar en el programa.	 Aprender a crear y atrapar distintos tipos de excepciones para poder informar al usuario de manera adecuada sobre cualquier problema detectado.

 Leer y escribir en archivos secuenciales de texto.	 Repasar la manera de usar las clases que provee Java para manejar archivos y para manipular cadenas de caracteres.
 Manejar listas plegables en la interfaz de usuario.	 Estudiar la clase JComboBox del <i>framework</i> gráfico de Java.

5.2. Comprensión de los Requerimientos

Tarea 8



Objetivo: Entender el problema del caso de estudio de la tienda virtual de canciones MP3.

(1) Lea detenidamente el enunciado del caso de estudio e (2) identifique y complete la documentación de los cinco requerimientos funcionales del programa. Incluya toda la información disponible sobre los formatos de los archivos que se manejan en cada uno de los requerimientos (tanto de entrada como de salida), lo mismo que los errores que se pueden presentar y los mensajes que se deben mostrar al usuario.

Requerimiento funcional 1	Nombre	R1 – Presentar al usuario la información de una canción del catálogo
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Agregar un disco al catálogo
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Agregar una canción a un disco
	Resumen	
	Entrada	
	Resultado	

Requerimiento funcional 4	Nombre	R4 – Vender una canción a un usuario
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Procesar un archivo de pedidos de un cliente
	Resumen	
	Entrada	
	Resultado	

Además de los requerimientos funcionales de un programa (las opciones que se le deben ofrecer al usuario y el comportamiento esperado al usarlas), se deben documentar las condiciones de operación que define el cliente, las cuales se catalogan dentro del grupo de requerimientos no funcionales. Por ejem-

plo, la persistencia del modelo del mundo es más una característica del programa (“la información debe perdurar”) que una nueva funcionalidad que se le ofrece al usuario. Para documentar los requerimientos no funcionales vamos a utilizar el siguiente formato:

Requerimiento no funcional 1	Tipo: Persistencia
	<p>Descripción:</p> <ul style="list-style-type: none"> • La información del modelo del mundo debe ser persistente. • El proceso de hacer persistente la información debe ser transparente para el usuario (no participa en el proceso). • Se deben usar archivos secuenciales para almacenar la información una vez que el programa haya terminado su ejecución.

5.3. Diseño e Implementación de las Excepciones

Vamos a comenzar a trabajar en este caso de estudio por el tema de manejo y recuperación de las situaciones anormales que tiene que saber manejar un

programa. En el enunciado vimos una lista de posibles problemas con los que el programa se puede encontrar y la acción que éste debe seguir. En esta sección vamos a extender el concepto y manejo de excepciones antes estudiados, como el mecanismo de base para lograr el objetivo propuesto.



Una **excepción** es una situación anormal que un programa ha detectado en alguno de sus métodos, y para la cual ha creado un objeto con el fin de representarla. Dicho objeto es creado y lanzado por el método que detecta el problema, y atrapado por algún otro método dentro del programa para informar al usuario o para establecer alguna forma de recuperación.

Anteriormente se estudió la manera de crear y atrapar un tipo de excepción (objetos de la clase `Exception`). Aquí veremos cómo generalizar la idea y crear distintos tipos de excepción, cada uno para indicar la aparición de un problema distinto. Piense, por ejemplo, que en caso de que el usuario intente agregar una canción o un disco que ya existe, nos gustaría crear un objeto de la clase `ElementoExisteException`, o que si el archivo en el que viene el pedido de un cliente tiene un formato incorrecto podamos representar el problema con un objeto de la clase `ArchivoVentaException`. Manejar distintos tipos de excepción nos va a permitir que el método

que las debe atrapar escoja cuáles de ellas le interesan y cuáles debe dejar pasar.

La presentación del tema laharemos contestando cuatro preguntas y utilizando el caso de estudio para ilustrar la teoría:

- ¿Cómo crear un tipo de excepción?

Esta primera pregunta que vamos a contestar se refiere a la manera de diseñar, implementar y documentar una excepción que queremos manejar en nuestro programa. El primer paso consiste en contestar las preguntas que aparecen en el siguiente formato:

Clase	Método	Excepción
<code>ElementoExisteException</code>	<ul style="list-style-type: none"> • Ya existe un disco en la tienda con ese nombre. • Ya existe en el disco una canción con ese nombre. 	<ul style="list-style-type: none"> • Nombre del disco o canción que se está tratando de agregar.
<code>ArchivoVentaException</code>	<ul style="list-style-type: none"> • El formato del archivo de pedidos impide que éste sea procesado completamente. • Error de lectura del archivo con el pedido de canciones. 	<ul style="list-style-type: none"> • Causa del error. • Número de canciones que pudieron ser vendidas y cuyo detalle se encuentra en la factura.
<code>PersistenciaException</code>	<ul style="list-style-type: none"> • Error al leer o escribir el archivo con la información del estado del modelo del mundo. 	<ul style="list-style-type: none"> • Causa del error.

En la primera columna tenemos el nombre que le queremos dar a la excepción, en la segunda, el grupo de errores que queremos representar con esta excepción, y en la tercera, la información que debe contener el objeto que va a representar el error, y que será utilizada para informar al usuario o para establecer algún proceso de recuperación.

El paso siguiente es declarar una clase en Java por cada una de las excepciones que vayamos a manejar en el programa. En el ejemplo 8 se ilustra la declaración de estas clases para la tienda de discos.

**Ejemplo 8**

Objetivo: Hacer la declaración en Java de las excepciones del programa.
En este ejemplo se muestra la declaración de las tres excepciones que se van a manejar en el programa.

```
package uniandes.cupi2.discotienda.mundo;

public class PersistenciaException extends Exception
{
    // -----
    // Constructor
    // -----

    public PersistenciaException( String causa )
    {
        super( causa );
    }
}
```

- Las clases de las excepciones están situadas en el mismo paquete del resto de clases del modelo del mundo.
- Para indicar que es una excepción se debe utilizar la cláusula "extends Exception" en el encabezado.
- En el constructor de la clase vamos a recibir como parámetro el motivo del error que fue detectado ("causa").
- En general, con el término "super" hacemos referencia a la clase de la cual se hace la extensión (en este caso la clase Exception).
- Con la instrucción super(causa) estamos pidiendo que se invoque el constructor de dicha clase pasándole como parámetro la información que acabamos de recibir.

```
package uniandes.cupi2.discotienda.mundo;

public class ArchivoVentaException extends Exception
{
    // -----
    // Atributos
    // -----

    private int cancionesVendidas;

    // -----
    // Constructor
    // -----

    public ArchivoVentaException( String causa,
                                  int ventas )
    {
        super( causa );
        cancionesVendidas = ventas;
    }

    public int darCancionesVendidas( )
    {
        return cancionesVendidas;
    }
}
```

- Declaramos el atributo "cancionesVendidas" de tipo entero, para almacenar el número de canciones que se pudieron vender a pesar del error.
- El constructor pide dos parámetros: uno con la causa y otro con las ventas alcanzadas a hacer.
- Con el primer parámetro se invoca el constructor de la clase Exception (como se mostró anteriormente), mientras el segundo valor es guardado en el parámetro.
- Debemos agregar un método a la clase que permita consultar el número de canciones vendidas (darCancionesVendidas) que retorna un valor de tipo entero.
- Para consultar la causa del error usamos el método de la clase Exception llamado getMessage(), el cual retorna una cadena de caracteres con el mensaje pasado en su constructor.
- Por lo general, se debe escribir un método de consulta por cada atributo que se haya declarado en la clase.

```

package uniandes.cupi2.discotienda.mundo;

public class ElementoExisteException
        extends Exception
{
    // -----
    // Constructor
    // -----

    public ElementoExisteException( String nomElem )
    {
        super( nomElem );
    }
}

```

 Es el mismo caso de la primera excepción mostrada (PersistenciaException), en la cual la información referente al error la podemos almacenar directamente en la excepción de la cual se hizo la extensión.

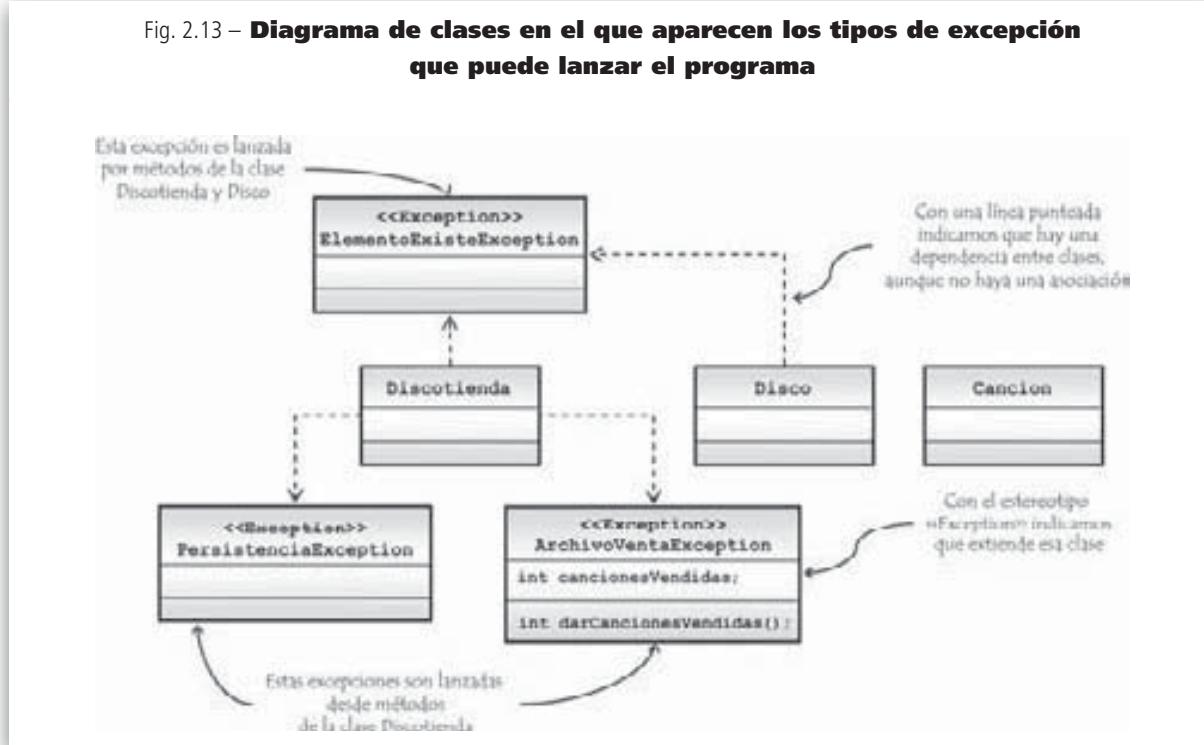


La clase `Exception` es una clase de Java que ofrece múltiples servicios, que se pueden consultar en la documentación. Los más usados son `getMessage()`, que retorna el mensaje con el que fue creada la excepción, y `printStackTrace()`, que imprime en la consola de ejecución toda la traza incluida en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

En la figura 2.13 aparece una parte del diagrama de clases del modelo del mundo, en el que se puede apreciar la manera de documentar las clases que

representan las excepciones y su relación de dependencia con el resto del modelo.

Fig. 2.13 – **Diagrama de clases en el que aparecen los tipos de excepción que puede lanzar el programa**



- ¿Cuándo vale la pena crear un nuevo tipo de excepción?

Hay dos razones principales por las cuales podríamos necesitar un nuevo tipo de excepción. La primera es una razón práctica y responde a la necesidad que tenemos en algunos casos de asociar más información con un error. Puesto que la clase `Exception` cuenta únicamente con una cadena de caracteres para acumular los datos asociados con el problema, debemos crear una nueva clase en caso de que sea necesaria información adicional en el proceso de recuperación o como parte de la información que debe recibir el usuario. Es el caso de nuestra excepción `ArchivoVentaException`, en la cual enviamos como parte del error información sobre el número de canciones que alcanzaron a ser vendidas.

La segunda razón es más conceptual, y responde a la conveniencia de clasificar los errores en grupos, de manera que sea más sencilla su identificación. Un método podría estar interesado en atrapar sólo algunos de los errores que se pue-

den presentar en un momento dado, o podría atraparlos todos pero usar estrategias distintas de recuperación.

- ¿Cómo declarar que un método lanza varios tipos de excepción?

Supongamos que el método de la clase `Discotienda` encargado de vender una lista de canciones que va leyendo de un archivo de texto debe considerar tres tipos distintos de error: el primero, cuando el archivo con los pedidos no existe en el computador (`FileNotFoundException`) ; el segundo, en caso de que el archivo no cumpla con el formato esperado (`ArchivoVentaException`) , y el tercero, cuando hay un error al tratar de escribir la factura en el disco duro (`IOException`) . A continuación aparece la declaración del encabezado de dicho método, que recibe un parámetro con el archivo que tiene los pedidos y otro parámetro con la ruta del directorio en el cual debe dejar el archivo con la factura. Este método retorna el nombre del archivo generado con la factura.

```
public String venderListaCanciones( File archivoPedido, String rutaFactura )
    throws FileNotFoundException, IOException, ArchivoVentaException
{
    ...
}
```

- ¿Cómo atrapar cada tipo de excepción en un punto distinto?

Para atrapar las excepciones seguiremos utilizando la instrucción `try-catch` de Java, pero la extenderemos para poder distinguir el tipo de problema que se presentó. Esta extensión consiste en que la instrucción acepta varios bloques de la

parte `catch`, cada uno asociado con un tipo de excepción distinto. Supongamos que desde la clase de la interfaz del programa queremos llamar el método `venderListaCanciones()`. El código necesario para atrapar cada tipo de excepción en un punto distinto sería el siguiente:

```
public void cargarPedido( )
{
    ...
    try
    {
        String archivoFactura = discotienda.venderListaCanciones( archivo, RUTA_FACTURAS
    );
```

```

}
catch( FileNotFoundException e )
{
    // Si la excepción es de tipo FileNotFoundException sólo se ejecuta el código que
    // aparece en este bloque
}
catch( IOException e )
{
    // Si la excepción es de tipo IOException sólo se ejecuta el código que aparece
    // en esta parte
}
catch( ArchivoVentaException e )
{
    // Si la excepción es de tipo ArchivoVentaException sólo se ejecuta el código que
    // aparece en este bloque
}
...
}

```

Si al ejecutarse la instrucción que se encuentra dentro del bloque `try` se produce una excepción de un tipo dado, el computador comienza a buscar de arriba hacia abajo hasta encontrar el bloque `catch` que le corresponde.

Por esta razón, si no queremos distinguir entre los

distintos tipos de excepción o si queremos agrupar el código de recuperación de más de un tipo, podemos escribir un bloque `catch` para el tipo de excepción `Exception` que incluye a todas las demás. Esto se ilustra en el siguiente fragmento de código.

```

public void cargarPedido( )
{
    ...
try
{
    String archivoFactura = discotienda.venderListaCanciones( archivo, RUTA_FACTURAS );
}
catch( FileNotFoundException e )
{
    // Si la excepción es de tipo FileNotFoundException sólo se ejecuta el código que
    // aparece en este bloque
}
catch( Exception e )
{
    // Si la excepción es de tipo IOException o ArchivoVentaException se ejecuta el
    // código que aparece en esta parte
}
...
}

```



Cuando se coloca el bloque de una excepción después del bloque de otra que ya la incluye (como sería el caso si invertimos los dos bloques `catch` del ejemplo anterior), el compilador presenta el siguiente error:

Unreachable catch block for FileNotFoundException. It is already handled by the catch block for Exception.

En algunos casos es conveniente atrapar un tipo de excepción y lanzar otro distinto con más información o que sea más fácil de tratar por aquéllos que hicieron el llamado. Suponga por ejemplo que estamos en el método que procesa un archivo de pedidos (`venderListaCanciones()`) y detectamos un error de lectura de bajo nivel (un error en el disco duro). En ese caso la instrucción de lectura va a generar una excepción de tipo `IOException`,

la cual debemos atrapar, tal como se muestra en el siguiente fragmento de código. ¿Qué hacer ahí? ¿Por qué no dejarla pasar y que sea la interfaz la que haga el proceso de recuperación? La respuesta es que si la dejamos pasar, perdemos la información del contexto del error que se necesita más adelante (el número de canciones que sí se pudieron vender). Por esa razón debemos lanzar una excepción que incluya esta información.

```
public String venderListaCanciones( File archivoPedido, String rutaFactura )
    throws FileNotFoundException, IOException, ArchivoVentaException
{
    int cancionesVendidas;
    ...
    try
    {
        pedido = lector.readLine();
    }
    catch( IOException e )
    {
        throw new ArchivoVentaException( e.getMessage(), cancionesVendidas );
    }
}
```

Hay un bloque opcional que se puede colocar al final de la secuencia de bloques `catch`, cuya etiqueta es la palabra `finally`, tal como se muestra a continuación:

```
public String venderListaCanciones( File archivoPedido, String rutaFactura )
    throws FileNotFoundException, IOException, ArchivoVentaException
{
    int cancionesVendidas;
    ...
    try
    {
        pedido = lector.readLine();
    }
    catch( IOException e )
    {
        throw new ArchivoVentaException( e.getMessage(), cancionesVendidas );
    }
    finally
    {
        // Siempre ejecuta estas instrucciones, incluso si se lanza una excepción.
        lector.close();
    }
}
```

Las instrucciones que aparecen en ese bloque se ejecutan siempre, independientemente de si se atrapó una excepción o si se terminó de ejecutar el bloque `try` sin ningún inconveniente. Ése es un excelente punto para cerrar los flujos de lectura y escritura, y para verificar el invariante en los métodos constructores y modificadores. El hecho de que se haya producido un error no nos da derecho en ningún caso (a no ser que sea un error fatal y el programa esté a punto de cerrarse) a dejar de cumplir el invariante de la clase.

5.4. Arquitectura y Responsabilidades en el Modelo del Mundo

En esta sección vamos a hacer un recorrido construyendo los elementos del modelo del mundo, aplicando para esto lo estudiado anteriormente.

Tarea 9



Objetivo: Definir e implementar las partes del modelo del mundo que no tienen que ver con manejo de archivos.

Desarrolle las tareas planteadas a continuación, utilizando la información que aparece en el enunciado del caso de estudio.

1. Complete el diagrama de clases del modelo del mundo.



2. Describa el invariante de cada una de las clases.

Discotienda	
Disco	
Cancion	

3. Escriba la declaración en Java de cada una de las clases.

```
public class Discotienda  
{  
  
}
```

```
public class Disco  
{  
  
}  
  
}
```

```
public class Cancion  
{  
  
}  
  
}
```

4. Implemente el método que verifica el invariante de cada una de las clases y los método privados que lo apoyan.

```
public class Discotienda  
{  
    private void verificarInvariante( )  
    {  
  
    }  
  
    private boolean buscarDiscosConElMismoNombre( )  
    {  
  
    }  
}
```

```
public class Disco
{
    private void verificarInvarianto( )
    {

    }

    private boolean buscarCancionesConElMismoNombre( )
    {

    }
}

public class Cancion
{
    private void verificarInvarianto( )
    {

    }
}
```

En la siguiente tabla se resumen los principales métodos de las tres clases del mundo. Para ver los contratos precisos, se recomienda consultar el CD.

Clase Discotienda:	
<code>Discotienda(String nombreArchivoDiscotienda)</code>	■ Es el constructor de la clase. Recibe como parámetro el nombre del archivo en el cual debe manejar la persistencia. Lanza la excepción PersistenciaException en caso de problema. Si el archivo no existe crea una tienda de discos vacía.
<code>Disco darDisco(String nombreDisco)</code>	■ Retorna un disco de la discotienda dado su nombre. Si no lo encuentra, retorna null.
<code>void agregarDisco(String nombreDiscoD, String artistaD, String generoD, String imagenD)</code>	■ Agrega un nuevo disco a la discotienda, dados su nombre, artista, género y el nombre del archivo en donde se encuentra almacenada la imagen de la portada. Lanza la excepción ElementoExisteException si ya existe un disco con el mismo nombre.
<code>void agregarCancionADisco(String nombreDisco, String nombreC, int minutosC, int segundosC, double precioC, double tamanoC, int calidadC)</code>	■ Agrega una nueva canción a un disco, del cual se da su nombre. Llegan como parámetro todos los datos de la canción. Lanza la excepción ElementoExisteException si ya existe una canción con el mismo nombre en ese disco.
<code>String venderCancion(Disco disco, Cancion cancion, String email, String rutaFactura)</code>	■ Registra la venta de una canción y genera la factura en un archivo nuevo. Recibe como parámetros el disco (que debe existir en la tienda), la canción, la dirección electrónica del comprador y el directorio en donde debe dejar el archivo con la factura. Lanza la excepción IOException en caso de error generando la factura. Retorna el nombre del archivo creado con la factura.
<code>ArrayList darDiscos()</code>	■ Retorna un vector con los nombres de todos los discos de la tienda.

<code>String venderListaCanciones(File archivoPedido, String rutaFactura)</code>	 Vende todas las canciones que aparecen en el archivo que viene como parámetro y retorna el nombre del archivo en el que generó la factura. Lanza la excepción FileNotFoundException si el archivo de lectura no existe, la excepción IOException si hay un error en la escritura de la factura y la excepción ArchivoVentaException si hay un problema con el formato del archivo del pedido o un error de lectura.
<code>boolean validarEmail(String email)</code>	 Indica si la dirección electrónica que se recibe como parámetro es válida.

Clase Disco:

<code>Disco(String nombreDiscoD, String artistaD, String generoD, String imagenD)</code>	 Crea un disco con los datos que recibe como parámetro. Inicialmente el disco no tiene ninguna canción. La precondición exige que todos los datos sean válidos.
<code>Cancion darCancion(String nombreC)</code>	 Retorna una canción del disco dado su nombre. Si no la encuentra retorna null.
<code>void agregarCancion(Cancion c)</code>	 Agrega una canción al disco. Lanza la excepción ElementoExisteException si ya existe en el disco una canción con ese nombre.
<code>String darArtista()</code>	 Retorna el artista del disco.
<code>ArrayList darNombresCanciones()</code>	 Retorna un vector con los nombres de las canciones del disco.
<code>String darGenero()</code>	 Retorna el género del disco.
<code>String darNombreDisco()</code>	 Retorna el nombre del disco.
<code>String darImagen()</code>	 Retorna el nombre del archivo que contiene la imagen del disco.
<code>double darPrecioDisco()</code>	 Retorna el precio total del disco (la suma de los precios de sus canciones).
<code>boolean equals(String nombre)</code>	 Indica si el disco tiene el nombre que llega como parámetro.

Clase Cancion:

<code>Cancion(String nombreC, int minutosC, int segundosC, double precioC, double tamanoC, int calidadC, int cantidad)</code>	 Crea una nueva canción con los datos suministrados. La precondition exige que todos los datos sean válidos.
<code>String darNombre()</code>	 Retorna el nombre de la canción.
<code>int darMinutos()</code>	 Retorna la duración en minutos de la canción.
<code>int darSegundos()</code>	 Retorna la duración en segundos de la canción.
<code>double darPrecio()</code>	 Retorna el precio de la canción.
<code>double darTamano()</code>	 Retorna el tamaño de la canción, expresado en <i>megabytes</i> (MB).
<code>int darCalidad()</code>	 Retorna la calidad de la canción, expresada en <i>kilobytes</i> por segundo (Kbps).
<code>int darUnidadesVendidas()</code>	 Retorna el número de unidades vendidas de la canción.
<code>void vender()</code>	 Aumenta en uno la cantidad de unidades vendidas.
<code>boolean equals(String nombreCancion)</code>	 Indica si la canción tiene el nombre que llega como parámetro.

Vamos a plantear ahora como tarea al lector la implementación de algunos de los métodos del caso de estudio que no incluyen manejo de archivos.

Tarea 10

Objetivo: Implementar algunos de los métodos de las clases de la tienda virtual de discos.

Desarrolle los métodos que se indican a continuación. Como parte de la solución utilice únicamente los métodos de las clases antes descritos y los atributos definidos en la tarea 9.

```
public class Cancion
{
    public Cancion( String nombreC, int minutosC, int segundosC, double precioC,
                    double tamanoC, int calidadC, int cantidad )
    {

    }
```

```
public int darUnidadesVendidas( )
{
}

public void vender( )
{
}

public boolean equals( String nombreCancion )
{
}

public class Disco
{
    public Disco( String nombreDiscoD, String artistaD, String generoD, String
ImagenD )
    {

}

public Cancion darCancion( String nombreC )
{
}

public void agregarCancion( Cancion c ) throws ElementoExisteException
{
}

}
```

```
public double darPrecioDisco( )
{
}

}

public class Discotienda
{
    public Disco darDisco( String nombreDisco )
    {

    }

    public void agregarDisco( String nombreDiscoD, String artistaD, String generoD,
                            String imagenD ) throws ElementoExisteException
    {

    }

    public void agregarCancionADisco( String nombreDisco, String nombreC, int minutosC,
                                      int segundosC, double precioC, double tamanoC,
                                      int calidadC ) throws ElementoExisteException
    {
        }

}
```

5.5. Reportes y Otras Salidas en Archivos

En esta sección vamos a estudiar la construcción de métodos que exportan información en un archivo.

Ejemplo 9



Objetivo: Ilustrar la construcción de métodos que exportan información en un archivo.

En este ejemplo implementamos el método de la clase `Discotienda` encargado de vender una canción y generar un archivo con la respectiva factura.

```
public String venderCancion( Disco disco, Cancion cancion,
                           String email, String rutaFactura )
                           throws IOException
{
    cancion.vender();
}
```

■ Éste es el método que vende una canción a un cliente, del cual tenemos su dirección electrónica. Como parámetro recibimos el disco, la canción, la dirección electrónica del cliente y la ruta del directorio en el cual debemos dejar el archivo con la factura.

■ El método lanza la excepción `IOException` si se presenta cualquier error al escribir el archivo.

■ La primera instrucción del método consiste en indicar que se ha vendido una nueva unidad de la canción.

```
int posArroba1 = email.indexOf( "@" );
String login = email.substring( 0, posArroba1 );

String strTiempo =
    Long.toString( System.currentTimeMillis() );

String nombreArchivo = login + "_" +
    strTiempo + ".fac";
```

■ La siguiente tarea del método es definir el nombre del archivo que va a contener la factura. Para esto tomamos la primera parte de la dirección electrónica del usuario y le concatenamos la fecha actual en milisegundos. Así garantizamos que el nombre siempre será único y no vamos a escribir sobre una factura que ya existe.

```
File directorioFacturas = new File( rutaFactura );
if( !directorioFacturas.exists() )
    directorioFacturas.mkdirs();

File archivoFactura = new File( directorioFacturas,
                               nombreArchivo );

PrintWriter out = new PrintWriter( archivoFactura );
```

■ Luego, creamos el archivo de la factura. Para hacerlo verificamos primero que el directorio en donde lo vamos a almacenar existe. Si no es así, pedimos que se creen todos los directorios necesarios usando el método `mkdirs()`.

■ Creamos después el archivo en el directorio de las facturas, definiendo un flujo de salida sobre él.

```

Date fecha = new Date();

out.println( "miDiscoTienda - FACTURA" );
out.println( "Fecha:      " + fecha.toString() );
out.println( "Email:      " + email );

out.println( "Canción:      " + cancion.darNombre() +
            " - " + disco.darArtista() );

out.println( "          " + disco.darNombreDisco() );

out.println( "No de Canciones: 1" );

DecimalFormat df = new DecimalFormat( "$0.00" );
out.println( "Valor Total:    " +
            df.format( cancion.darPrecio() ) );

out.close();

return nombreArchivo;
}

```

Los métodos de creación de reportes siguen siempre un esquema parecido: (1) crear el archivo, (2) escribir la información en el archivo buscando los valores necesarios en los objetos del modelo del

mundo que los tienen y (3) cerrar el archivo. En las siguientes tareas estudiaremos y construiremos otros métodos que van a generar reportes para la tienda de discos.

Tarea 11

Objetivo: Estudiar la generación de la factura en el caso de un pedido recibido en un archivo.

Siga los pasos que se dan a continuación y conteste las preguntas que se le plantean, utilizando el programa `n8_discotienda` que se encuentra en el CD.

1. Descomprima el archivo `n8_discotienda.zip` que se encuentra en el CD y cree el respectivo proyecto en Eclipse. Localice en el proyecto la clase `Discotienda` y edite su código fuente.

2. Busque el método `generarFactura()`, estúdielo detenidamente y conteste las preguntas que se plantean a continuación:

¿Qué información llega como parámetro en los vectores `discos`, `canciones` y `noEncontradas`? ¿Qué relación existe entre ellos?

¿Cómo calcula el método el precio total de la venta y el número de canciones vendidas?

Ya estamos listos para comenzar a escribir. Comenzamos escribiendo el encabezado de la factura, con la fecha y la dirección electrónica del cliente. Para esto utilizamos el método `println()` del flujo de salida.

Luego escribimos el nombre de la canción, el artista y el disco.

Siguiendo con el formato definido en los requerimientos funcionales, incluimos en la factura el número de canciones vendidas (1) y el precio de venta. Para imprimir este último en el formato pedido, utilizamos la clase `DecimalFormat`.

Finalmente cerramos el archivo y retornamos el nombre con el que éste fue creado.

¿Cómo se incluyen en la factura las canciones que no pudieron ser vendidas?	
¿Qué retorna al final el método? ¿Cómo calcula dicho valor?	
3. Ejecute el programa y venda una canción. Localice en el directorio data\facturas la factura de la venta que acaba de hacer y editela. Verifique que el contenido sea correcto.	
4. Busque en el directorio data el archivo pedido.txt . Edítelo y vea un ejemplo de un pedido de una lista de canciones. Usando el botón Cargar Pedido del programa haga la venta de las canciones de dicho archivo. Localice en el directorio data\facturas la factura de la venta que acaba de hacer y editela. Verifique que el contenido sea correcto.	

En las siguientes cuatro tareas vamos a extender la funcionalidad del programa que maneja la tienda de discos, con operaciones de exportación de información en archivos secuenciales de texto.

Tarea 12



Objetivo: Agregar al programa una opción para exportar en un archivo de texto todas las canciones que comienzan por un prefijo dado.

Siga los pasos que se dan a continuación, trabajando sobre el proyecto **n8_discotienda** creado anteriormente.

1. Agregue en la clase **Cancion** un método de retorno lógico llamado **comienzaPrefijo(cadena)**, que retorna verdadero si el nombre de la canción comienza por el prefijo que llega como parámetro.

2. Agregue en la clase **Disco** un método llamado **darCancionesPrefijo(cadena)**, que retorna un **ArrayList** con todas las canciones del disco cuyo nombre comienza por el prefijo que llega como parámetro. Utilice el método construido en el punto anterior.

3. Agregue en la clase **Discotienda** un método llamado **exportarPorPrefijo(cadena, nombreArchivo)**, que crea un archivo en el directorio **data\export** con el nombre que llega como segundo parámetro, y que incluye el nombre de todas las canciones de la tienda de discos cuyo nombre comienza por el prefijo dado. Por cada canción debe aparecer el nombre completo y entre paréntesis el nombre del disco que la contiene. Utilice el método construido en el punto anterior.

4. Asocie con el botón **Opción 1** la funcionalidad anterior.

5. Ejecute el programa y verifique que funciona correctamente.

Tarea 13

Objetivo: Agregar al programa una opción para exportar en un archivo de texto los nombres de los discos de un género específico.

Siga los pasos que se dan a continuación, trabajando sobre el proyecto n8_discotienda creado anteriormente.

1. Agregue en la clase Discotienda un método llamado `exportarGenero(cadena)`, que crea un archivo en el directorio **data\export**, cuyo nombre es el género pedido (por ejemplo, "POP.txt"), que contiene el nombre de todos los discos que tiene la discotienda en el género que llega como parámetro. En el archivo debe aparecer el nombre del disco y el número de canciones que contiene.
2. Asocie con el botón **Opción 2** la funcionalidad anterior.
3. Ejecute el programa y verifique que funciona correctamente.

Tarea 14

Objetivo: Agregar al programa una opción para generar un reporte con las diez canciones más vendidas de la tienda.

Siga los pasos que se dan a continuación, trabajando sobre el proyecto n8_discotienda creado anteriormente.

1. Agregue en la clase Discotienda un método llamado `reporteTopTen()`, que crea un archivo llamado **topten.txt** en el directorio **data\export**, que contiene las diez canciones más vendidas de la tienda. De cada canción debe aparecer el nombre y el número de veces que ha sido vendida.
2. Asocie con el botón **Opción 3** la funcionalidad anterior.
3. Ejecute el programa y verifique que funciona correctamente.

Tarea 15

Objetivo: Agregar al programa una opción para exportar en un archivo toda la información de ventas de un artista.

Siga los pasos que se dan a continuación, trabajando sobre el proyecto n8_discotienda creado anteriormente.

1. Agregue en la clase Cancion un método llamado `darRegalias()`, que calcula las regalías que se le deben pagar al artista por las ventas de dicha canción. Las regalías corresponden al 10% del total de ventas, si se han vendido menos de 25.000 copias, y del 15% si se han vendido más.
2. Agregue en la clase Disco un método llamado `darRegalias()`, que calcula las regalías que se le deben pagar al artista por las ventas de todas las canciones incluidas en ese disco. Utilice el método construido en el punto anterior.
3. Agregue en la clase Discotienda un método llamado `reporteRegalias(artista)`, que crea un archivo con el nombre del artista (por ejemplo, "shakira.txt") en el directorio **data\export**, el cual contiene sus discos, con el valor de las regalías que se le deben pagar por cada uno de ellos. Utilice el método construido en el punto anterior.
4. Asocie con el botón **Opción 4** la funcionalidad anterior.
5. Ejecute el programa y verifique que funciona correctamente.

5.6. Importación de Datos desde Archivos

En esta sección vamos a estudiar los métodos que importan información a partir de un archivo. En el ejemplo 10

mostraremos la implementación del método que lee un pedido de un archivo (requerimiento funcional 5) y luego propondremos una serie de tareas al lector para extender la funcionalidad del programa con operaciones cuya información proviene de archivos secuenciales de texto.

Ejemplo 10



Objetivo: Ilustrar la construcción de métodos que importan información de un archivo.

En este ejemplo implementamos el método de la clase `Discotienda`, encargado de procesar un archivo con una lista de pedidos de un cliente y generar un archivo con la respectiva factura.

```
public String venderListaCanciones( File archivoPedido,
                                    String rutaFactura )

    throws FileNotFoundException, IOException,
           ArchivoVentaException

{
    BufferedReader lector = new BufferedReader(
        new FileReader( archivoPedido ) );

    String email = null;

    try
    {
        email = lector.readLine();
    }
    catch( IOException e )
    {

        throw new ArchivoVentaException( e.getMessage(), 0 );
    }

    if( email == null )
        throw new ArchivoVentaException(
            "El archivo está vacío", 0 );

    if( !validarEmail( email ) )
        throw new ArchivoVentaException(
            "El email indicado no es válido", 0 );
}
```

Este método recibe como parámetro el archivo con el pedido y la ruta del directorio en el cual debe dejar la factura.

El método puede lanzar tres tipos distintos de excepción, según el problema detectado.

Con la primera instrucción, abre el archivo con el pedido. Si no existe, el constructor de la clase `FileReader` lanza la excepción `FileNotFoundException`.

Pasamos a leer la primera línea del archivo de pedidos, en donde debe aparecer la dirección electrónica del cliente.

Si hay un error en la lectura del archivo, lanzamos la excepción `ArchivoVentaException`, pasando como parámetros el mensaje de la excepción producida (usando el método `getMessage()` lo recuperamos) y el valor 0 (el número de canciones vendidas).

Verificamos que la dirección electrónica leída sea válida. En caso de problema lanzamos la respectiva excepción.

Utilizamos el método `validarEmail()` de la clase `Discotienda`, descrito antes.

```

String pedido = null;

try
{
    pedido = lector.readLine();
}
catch( IOException e )
{
    throw new ArchivoVentaException( e.getMessage(), 0 );
}

if( pedido == null )
    throw new ArchivoVentaException(
        "Debe haber por lo menos una canción en el pedido", 0 );

```

Leemos la cadena de caracteres con la primera canción del pedido del cliente. Si hay un error en el proceso, lanzamos la excepción ArchivoVentaException con el mensaje del problema y con 0 canciones vendidas.

```

ArrayList discosFactura = new ArrayList();
ArrayList cancionesFactura = new ArrayList();
ArrayList cancionesNoEncontradas = new ArrayList();

int cancionesVendidas = 0;

```

Si el archivo llegó a su fin lanzamos una excepción diciendo que debería haber por lo menos una canción en el archivo de pedidos.

```

while( pedido != null )
{
    int p1 = pedido.indexOf( "#" );
    if( p1 != -1 )
    {
        String resto1 = pedido.substring( p1 + 1 );
        int p2 = resto1.indexOf( "#" );

        if( p2 != -1 )
        {
            String nombreDisco = pedido.substring( 0, p1 );
            String nombreArtista = resto1.substring( 0, p2 );
            String nombreCancion = resto1.substring( p2 + 1 );

```

Ahora inicializamos las estructuras de datos que vamos a utilizar para crear la factura, las cuales estudiamos en la tarea 11.

Utilizamos ahora el patrón de recorrido de archivos secuenciales, en el cual vamos avanzando línea por línea, procesando cada una de ellas en el cuerpo del ciclo.

La primera tarea consiste en separar la información que se recibe en cada línea (disco#artista#cancion), verificando que el formato sea el adecuado.

```

Disco discoBuscado = darDisco( nombreDisco,
                                nombreArtista, nombreCancion );
if( discoBuscado != null )
{
    Cancion cancionPedida =
        discoBuscado.darCancion( nombreCancion );

    cancionPedida.vender();
    discosFactura.add( discoBuscado );
    cancionesFactura.add( cancionPedida );
    cancionesVendidas++;
}
else
    cancionesNoEncontradas.add( pedido );
}
else
    cancionesNoEncontradas.add( pedido );
}
else
    cancionesNoEncontradas.add( pedido );

```

Con la información recuperada del archivo localizamos el disco y la canción en la discoteca.

Si el formato es inválido o si la canción no se puede localizar, se agrega la información respectiva al vector de canciones no encontradas, que será utilizado luego para la generación de la factura.

Una vez localizada la canción registramos la venta con el método vender() y almacenamos la información para la factura.

```

try
{
    pedido = lector.readLine( );
}
catch( IOException e )
{
    generarFactura( discosFactura, cancionesFactura,
                    cancionesNoEncontradas, email, rutaFactura );

    throw new ArchivoVentaException( e.getMessage( ),
                                    cancionesVendidas );
}
}

lector.close( );

return generarFactura( discosFactura, cancionesFactura,
                      cancionesNoEncontradas, email, rutaFactura );
}

```

Luego de procesada la línea, leemos la siguiente, de acuerdo con el patrón de recorrido de archivos secuenciales.

Si se presenta un error leyendo la siguiente línea, capturamos la excepción, pedimos que se cree la factura para las canciones que ya fueron vendidas y lanzamos la excepción ArchivoVentaException.

Antes de salir del método cerramos el flujo de lectura, pedimos la generación de la factura de venta y retornamos el nombre del archivo.

Tarea 16



Objetivo: Agregar al programa una opción para incluir en el catálogo una lista de canciones que acaban de llegar a la tienda, las cuales vienen descritas en un archivo.

Siga los pasos que se dan a continuación, trabajando sobre el proyecto n8_discotienda creado anteriormente.

1. Cree en la clase Discotienda un método llamado `incluirCatalogo(nombreArchivo)`, que agrega en el catálogo de la tienda las canciones que vienen descritas en el archivo cuyo nombre llega como parámetro. El archivo tiene el siguiente formato y cumple con las siguientes reglas:

- Todas las canciones del archivo pertenecen al mismo disco
- Formato:
 - nombre del disco
 - nombre del artista
 - género del disco
 - nombre del archivo con la carátula (debe encontrarse en el subdirectorio "data/ímágenes")
 - número de canciones que incluye
 - canción1-minutos-segundos-tamaño-calidad-precio
 - canción2-minutos-segundos-tamaño-calidad-precio
 - ...

Diseñe las excepciones necesarias y utilice el método que aparece en el siguiente paso como parte de su solución.

2. Implemente en la clase Cancion un constructor que reciba como parámetro un flujo de lectura abierto, el cual se encuentra listo para leer la siguiente línea del archivo con la información de una canción, de acuerdo con el formato antes descrito. Este constructor debe ser invocado desde el método implementado en el punto anterior, para construir los objetos de la clase Cancion descritos en el archivo.

3. Asocie con el botón **Opción 5** la funcionalidad anterior, utilizando un `JFileChooser` para seleccionar el archivo que trae la información de las nuevas canciones.

4. Ejecute el programa y verifique que funciona correctamente.

Tarea 17



Objetivo: Agregar al programa una opción para vender una canción dada a una lista de clientes, cuya información viene en un archivo.

Siga los pasos que se dan a continuación, trabajando sobre el proyecto `n8_discotienda` creado anteriormente.

1. Cree en la clase `Discotienda` un método llamado `venderGrupo(disco, cancion, nombreArchivo)`, que vende una canción dada (descrita en los parámetros) a una lista de clientes, cuyas direcciones electrónicas están en el archivo que se recibe como parámetro. En dicho archivo aparece una dirección electrónica por línea. El programa debe generar una factura individual por cada venta.

2. Asocie con el botón **Opción 6** la funcionalidad anterior, haciendo las modificaciones que sean necesarias en el programa.

3. Ejecute el programa y verifique que funciona correctamente.

5.7. Construcción de Pruebas Unitarias

El único punto nuevo en el tema de pruebas unitarias automáticas tiene que ver con la manera de probar que las excepciones se lancen de manera correcta al detectarse un error. Para esto, vamos a construir

escenarios y casos de prueba que produzcan el error esperado y vamos a atrapar la excepción y a constatar que tenga la información adecuada. Si por alguna razón el método no lanza la excepción, debemos frenar la ejecución de las pruebas e indicar que hay una falla en las mismas. Esto se ilustra en el ejemplo 11 para la tienda de discos.



Objetivo: Construir las pruebas para verificar que un método lanza las excepciones a las que se compromete en su contrato.

En este ejemplo vamos a mostrar la manera de probar que el método `agregarCancion()` de la clase `Disco` lanza la excepción `ElementoExisteException` cuando la canción que se quiere agregar ya existe.

```
public class DiscoTest extends TestCase
{
    // -----
    // Atributos
    // -----
    private Disco disco1;

    private void setupEscenario1( )
    {
        disco1 = new Disco( "Mi disco1", "artistaPrueba",
                           "Latino", "prueba.jpg" );
    }
}
```

El escenario para la prueba es un disco, cuyo nombre es "Mi disco1", del artista "artistaPrueba", del género "Latino" y cuya carátula se encuentra en el archivo llamado "prueba.jpg".

```

public void testAgregarCancion( )
{
    setupEscenario1( );

    Cancion c1 = new Cancion( "Cancion1", 1, 20, 1.50, 2, 96, 0 );
    Cancion c2 = new Cancion( "Cancion1", 2, 40, 3.45, 2, 96, 0 );
    try
    {
        disco1.agregarCancion( c1 );
    }
    catch( ElementoExisteException e )
    {
        fail( );
    }
    try
    {
        disco1.agregarCancion( c2 );
        fail( );
    }
    catch( ElementoExisteException e )
    {
        assertTrue( true );
    }
}
}

```

La prueba consiste en agregar una primera canción (c1) y verificar que no lanza ninguna excepción. Fíjese que en el bloque catch hacemos que la prueba falle.

Luego intentamos agregar otra canción (c2) con el mismo nombre y verificamos que se lanza la excepción tal como lo exige el contrato. El método fail() es invocado justo después del método que ha debido producir la excepción, de manera que si el método agregarCancion() está bien construido, el programa de prueba no debe nunca ejecutar el método fail().

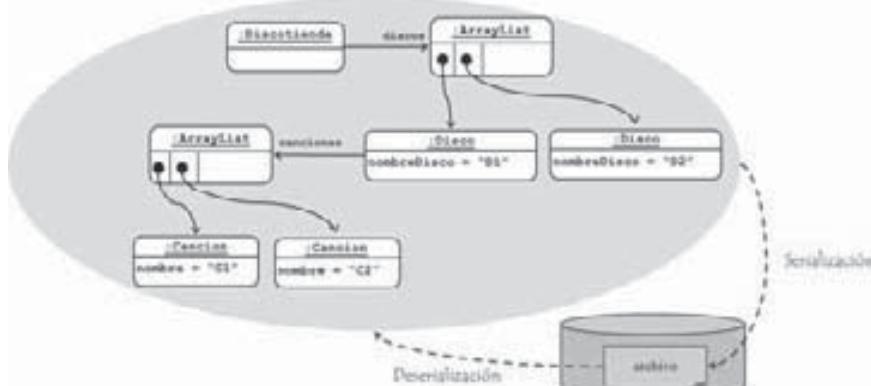
5.8. Persistencia Simple por Serialización

Podemos imaginarnos el problema de hacer persistir la información de un programa como el problema de llevar un grupo de objetos de la memoria principal a un archivo, para luego ser capaces de hacer el proceso contrario y reconstruir el estado del modelo del mundo usando la información traída del disco duro, tal como se sugiere en la figura 2.14. Dicho proceso lo podríamos hacer sin ningún problema utilizando lo es-

tudiado en las secciones anteriores: exportaríamos el estado completo del modelo del mundo en un archivo secuencial de texto y luego lo importaríamos en algún momento durante el proceso de construcción.

Dicho proceso de convertir un grupo de objetos en una secuencia de bytes se denomina **serialización** y es uno de los esquemas más simples de persistencia. El proceso contrario, que permite reconstruir los objetos con su estado y sus asociaciones, se denomina **deserialización**.

Fig. 2.14 – Proceso de serialización / deserialización



El lenguaje Java incluye dentro de su *framework* de manejo de archivos un conjunto de clases que nos van a simplificar enormemente la persistencia de un objeto por serialización. Con una sola instrucción seremos capaces de enviar al disco duro toda la información de la discotienda, y con otra instrucción la recuperaremos en el constructor de la clase. Interesante, ¿no?

Veamos primero el proceso de recuperación a partir de un archivo. Allí participan las clases `ObjectInputStream` y `FileInputStream`, las cuales se encuentran conectadas al archivo como se ilustra en la figura 2.15.



Los objetos de la clase `ObjectInputStream` ven el archivo como una fuente secuencial de entrada de objetos, cada uno de los cuales se lee con el método `readObject()`. En el ejemplo 12 se muestra

el código del constructor de la clase `Discotienda` encargado de reconstruir el estado completo de un objeto de dicha clase.

Ejemplo 12



Objetivo: Mostrar la manera de utilizar las clases de Java que nos ayudan en el proceso de reconstrucción de un objeto por deserialización.

En este ejemplo vamos a mostrar el constructor de la clase `Discotienda`, el cual recibe como parámetro el nombre del archivo a partir del cual debe recuperar el estado de la tienda de discos. La manera de recuperarlo depende siempre de la manera como ha sido almacenado. En este caso el método que hace la serialización guardó en el archivo el vector de discos, puesto que esa información basta para recuperar todo el estado del modelo del mundo.

```

public Discotienda( String nombreArchivoDiscotienda )
                    throws PersistenciaException
{
    archivoDiscotienda = nombreArchivoDiscotienda;
    File archivo = new File( archivoDiscotienda );
    if( archivo.exists( ) )
    {
        try
    
```

Como primera medida, guardamos el nombre del archivo en un atributo, de manera que cuando vayamos a salvar de nuevo el estado lo hagamos en el mismo archivo del cual partimos.

```

{
    ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream( archivo ) );

    discos = ( ArrayList )ois.readObject( );
}

ois.close();
}
catch( Exception e )
{
    registrarError( e );

    throw new PersistenciaException(
        "Imposible restaurar el estado del programa (" +
        e.getMessage( ) + ")" );
}
else
{
    discos = new ArrayList( );
}

verificarInvariantes( );
}

```

El constructor debe considerar dos casos: si el archivo no existe es porque es la primera vez que se ejecuta el programa, luego debe partir de un estado en el cual no hay ningún disco. Basta entonces con crear un ArrayList vacío.

Si el archivo existe, debe partir de allí para reconstruir el estado de la tienda de discos. Crea en ese caso la fuente de entrada de objetos (ois) y la conecta con el archivo.

Con el método de lectura readObject() leemos todo el vector de discos, que ha debido ser guardado por el método que hace la serialización. Con el operador (ArrayList) convertimos el objeto leído en un vector.

Luego cerramos el flujo de lectura puesto que el estado de la discoteca ya ha sido recuperado.

Si hay cualquier problema en la lectura, registramos el error en un archivo de log con el método registrarError (que veremos más adelante) y luego lanzamos la excepción PersistenciaException que diseñamos con este propósito.

Antes de salir del constructor verificamos que se cumpla el invariante de la clase.

Para que un objeto de una clase pueda ser serializado (en nuestro caso se serializan los objetos de las clases Disco y Cancion) es necesario incluir en la declaración de la clase la cláusula implements Serializable tal como se muestra a continuación:

```

import java.io.*;

public class Disco implements Serializable
{
    ...
}

```

```

import java.io.*;

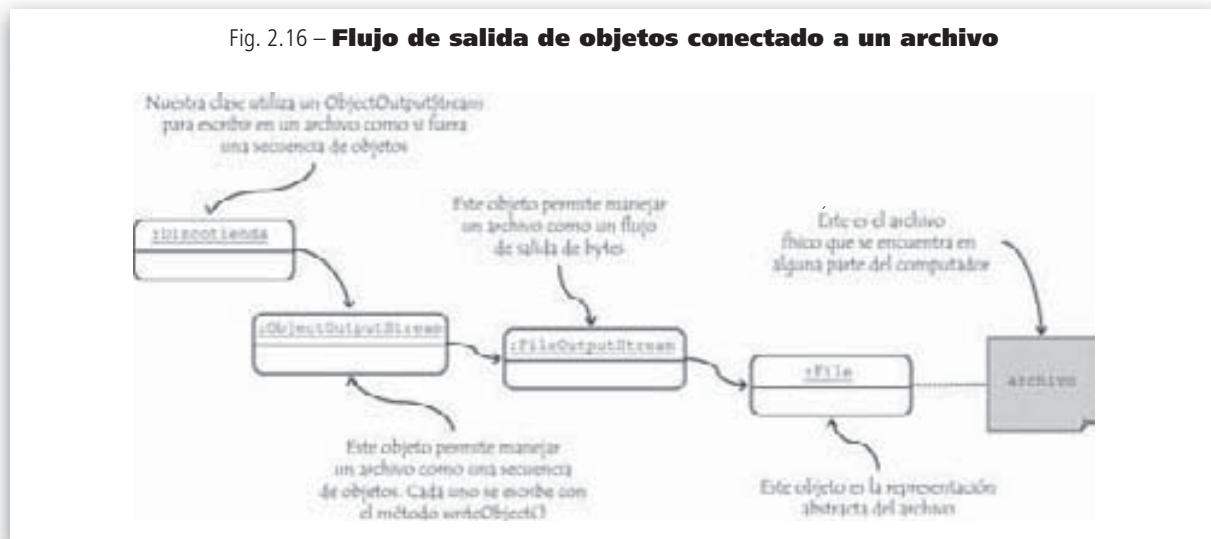
public class Cancion implements Serializable
{
    ...
}

```

Estas clases también deben declarar una constante llamada `serialVersionUID`, que será utilizada por la máquina virtual de Java para asegurarse de que la clase que hace la serialización sea la misma que lleva a cabo el proceso contrario. Dicho valor es guardado en el archivo. Si dicha constante no es declarada, el compilador presenta una advertencia y la genera internamente por nosotros.

Para hacer la serialización vamos a contar con las clases `ObjectOutputStream` y `FileOutputStream` del framework de archivos de Java, las cuales se relacionan con el archivo como se ilustra en la figura 2.16. Con el método `writeObject()` de la clase `ObjectOutputStream` enviamos objetos completos por el flujo de salida, tal como se muestra en el ejemplo 13.

Fig. 2.16 – **Flujo de salida de objetos conectado a un archivo**



Ejemplo 13



Objetivo: Mostrar la manera de utilizar las clases de Java que nos ayudan en el proceso de serialización de un objeto.

En este ejemplo vamos a mostrar el método de la clase `Discotienda` que hace la persistencia.

```
public void salvarDiscotienda( ) throws PersistenciaException
{
    try
    {
        ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream( archivoDiscotienda ) );

        oos.writeObject( discos );
        oos.close( );
    }
    catch( IOException e )
    {
        registrarError( e );
        throw new PersistenciaException( "Error al salvar: " +
            e.getMessage( ) );
    }
}
```

■ Creamos el flujo de salida de objetos (oos), asociándolo con el archivo del cual se cargó el estado inicial.

■ Escribimos en dicho flujo el vector de discos, usando la instrucción `writeObject()` y cerramos luego el flujo.

■ Si se produce una excepción en el proceso de serialización la atrapamos, la registramos en el archivo de log y luego lanzamos nuestra propia excepción.

Dos preguntas que nos quedan por contestar son: (1) ¿cómo tomar el control de un programa cuando el usuario ha decidido terminarlo cerrando su ven-

tana? y (2) ¿cómo evitar que el programa termine si hay un problema salvando el estado del modelo del mundo?



Objetivo: Mostrar la manera de tomar el control de un programa en el momento en el cual el usuario cierra la ventana de la interfaz y estudiar la forma de evitar que un programa termine en caso de un error grave en la persistencia.

En este ejemplo presentamos el método `dispose()` de la clase `InterfazDiscoTienda`.

```
public class InterfazDiscotienda extends JFrame
{
    // -----
    // Atributos
    // -----
    private Discotienda discotienda;

    public void dispose( )
    {
        try
        {
            discotienda.salvarDiscotienda( );

            super.dispose( );
        }
        catch( Exception e )
        {
            setVisible( true );

            int respuesta = JOptionPane.showConfirmDialog( this,
                "Problemas salvando la información:\n" +
                e.getMessage( ) +
                "\n;Quiere cerrar el programa sin salvar?", "Error", JOptionPane.YES_NO_OPTION );

            if( respuesta == JOptionPane.YES_OPTION )
            {
                super.dispose( );
            }
        }
    }
}
```

■ Cuando la ventana de un programa está a punto de cerrarse, invoca un método llamado `dispose()` de la clase `JFrame`, el cual podemos siempre reescribir para agregarle acciones al proceso de terminación de un programa.

■ En este ejemplo mostramos la manera de salvar el estado de la tienda de discos durante el proceso de cierre de la ventana principal del programa.

■ Con la instrucción `super.dispose()` invitamos el método que trae Java para cerrar la ventana, el cual queremos extender.

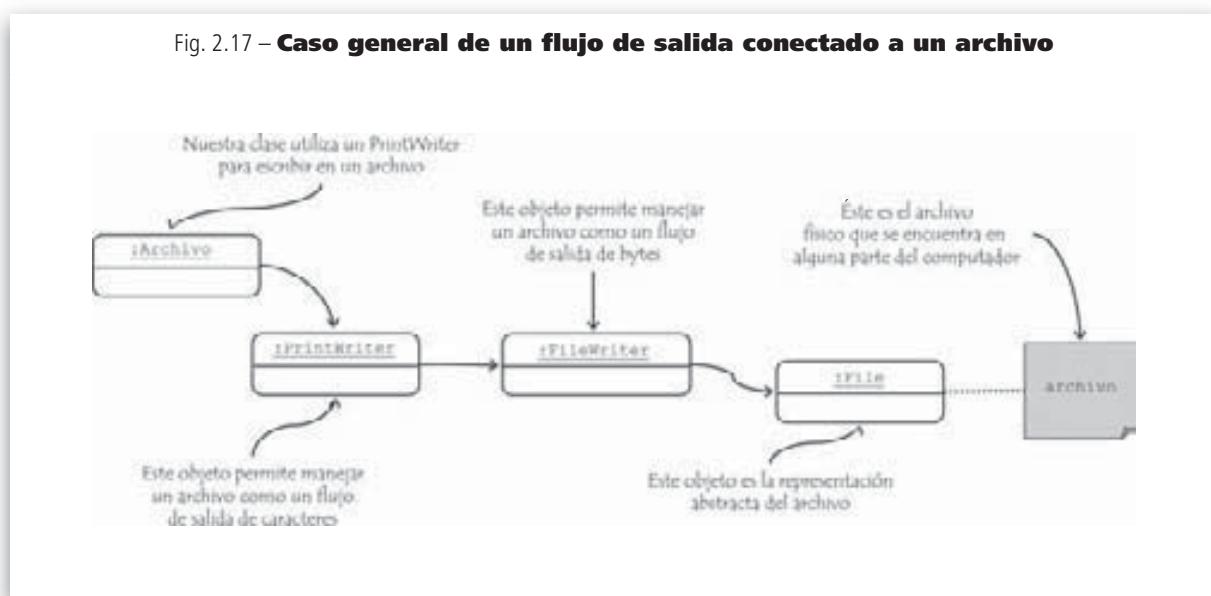
■ Si se presenta una excepción en el proceso de serialización del estado del modelo del mundo, evitamos que la ventana se cierre (no llamamos la instrucción `super.dispose()`) y, en lugar de eso, hacemos que la ventana sea de nuevo visible y le preguntamos al usuario si de verdad quiere salir a pesar de que va a perder toda la información.

■ Si el usuario contesta afirmativamente, llamamos el método `super.dispose()` para que cierre la ventana. En caso contrario la ventana queda abierta.

Pasemos ahora al problema de manejar el archivo de registro de errores (archivo de *log*), que, según los requerimientos del cliente, se debe llevar para dejar una traza escrita de todos los problemas en el manejo de la persistencia. El manejo de este tipo de archivos es un requerimiento típico de los programas comerciales, puesto que, en caso de llegar a una situación anormal, permite que se haga un diagnóstico preciso del problema encontrado.

La única dificultad adicional que tenemos es que este archivo no se debe reinicializar cada vez que creemos un flujo de escritura sobre él, puesto que la idea es que acumule la información histórica. Esto nos obliga a introducir una nueva clase de Java llamada `FileWriter`, que se sitúa como aparece en la figura 2.17.

Fig. 2.17 – Caso general de un flujo de salida conectado a un archivo



La clase `FileWriter` siempre había estado situada en ese punto, pero el constructor de la clase `PrintWriter` nos la había ocultado para facilitarnos la creación de flujos de escritura. Ahora,

debemos hacer explícita la llamada al constructor de dicha clase, para indicar allí que queremos seguir escribiendo al final de lo que ya tiene el archivo. Esto se ilustra en el ejemplo 15.

Ejemplo 15



Objetivo: Mostrar la manera de escribir al final del contenido actual de un archivo.

En este ejemplo presentamos el método que escribe en el archivo de registro de errores, el cual es llamado desde los métodos responsables de hacer la persistencia.

```

public void registrarError( Exception excepcion )
{
    try
    {
        FileWriter out = new FileWriter( LOG_FILE, true );
        PrintWriter log = new PrintWriter( out );
    }
}
  
```

Este método recibe como parámetro la excepción que explica el error detectado.

```

        log.println( "-----" );
        log.println( "Discotienda.java:" + new Date().toString() );
        log.println( "-----" );

        excepcion.printStackTrace( log );

        log.close( );
        out.close( );
    }
catch( IOException e )
{
    excepcion.printStackTrace( );
    e.printStackTrace( );
}
}

```

Al constructor del FileWriter le pasamos como parámetro una constante con el nombre del archivo y el valor “true” en el segundo parámetro, para indicar que, si ya existe, no hay que borrar el contenido que tiene.

Para escribir en el flujo de salida usamos el método printStackTrace(), que espera como parámetro un flujo de salida.

En la parte de abajo aparece un ejemplo del archivo de registro que crea este método.



5.9. El Componente JComboBox

El componente gráfico JComboBox es la implementación que trae Java para manejar en la interfaz una lista plegable. Tiene la ventaja de ocupar menos espacio que una lista (JList), pero no es muy recomendable su uso si la cantidad de elementos que se deben incluir es grande. Los métodos que nos provee dicho componente son los siguientes:

- `JComboBox()` : Es uno de los constructores disponibles para la clase. Crea una lista plegable sin elementos.

- `removeAllItems()` : Elimina todos los elementos de una lista plegable.
- `addItem(item)` : Agrega un elemento al final de una lista plegable.
- `setEditable(editable)` : Define si el valor seleccionado es o no editable.
- `getSelectedItem()` : Retorna el elemento de la lista plegable que se encuentra seleccionado.

Para el manejo de eventos funciona de manera idéntica a los botones, tal como se muestra en el ejemplo 16

Ejemplo 16

Objetivo: Mostrar la manera de usar el componente JComboBox de Java.

En este ejemplo presentamos un fragmento del código del constructor del panel que contiene la lista plegable de discos.

```
public class PanelDiscos extends JPanel implements ActionListener
{
    private static final String CAMBIAR_DISCO = "CambiarDisco";
    ...

    private JComboBox comboDiscos;
    ...

    public PanelDiscos( )
    {
        ...
        comboDiscos = new JComboBox( );
        comboDiscos.setEditable( false );
        comboDiscos.addActionListener( this );
        comboDiscos.setActionCommand( CAMBIAR_DISCO );
        add( comboDiscos, BorderLayout.NORTH );
        ...
    }

    public void refrescarDiscos( ArrayList discos )
    {
        comboDiscos.removeAllItems( );
        for( int i = 0; i < discos.size( ); i++ )
        {
            comboDiscos.addItem( discos.get( i ) );
        }
    }

    public void actionPerformed( ActionEvent evento )
    {
        String comando = evento.getActionCommand( );
        if( CAMBIAR_DISCO.equals( comando ) )
        {
            ...
        }
    }
}
```

■ En el encabezado de la clase declaramos que implementa ActionListener, como hacíamos en el caso de los botones.

■ La lista plegable se declara como un atributo de la clase.

■ La creación del componente gráfico y su configuración se hace en el constructor utilizando los métodos disponibles en la clase JComboBox.

■ Proveemos un método que refresca la lista de discos. Dicho método recibe como parámetro un vector con los nombres de los discos que debe mostrar. El método elimina los elementos que tenía en la lista plegable y agrega uno por uno los nuevos elementos.

■ Para atender los eventos utilizamos el mismo esquema que se usa con los botones (método actionPerformed).



En este punto es muy recomendable estudiar la solución completa del caso de estudio que se encuentra en el CD. Es importante darle una mirada a las pruebas unitarias automáticas y a las clases que implementan la interfaz de usuario.

6. Uso Básico del Depurador en Eclipse

Se denomina **depurar** un programa al proceso de encontrar errores en su código y corregirlos. Para ayudarnos en

esta tarea los ambientes de programación cuentan con algunas operaciones básicas, como detener la ejecución de un programa en un punto dado, para poder consultar allí el estado de algún objeto o de alguna variable. En esta sección estudiaremos algunas de las opciones que nos ofrece el ambiente Eclipse para tal fin.

Tarea 18



Objetivo: Aprender a utilizar las facilidades básicas de depuración que ofrece el ambiente de desarrollo Eclipse.

Siga los pasos que se dan a continuación y vaya contestando las preguntas que se plantean.

1. Si no lo ha hecho antes, descomprima el archivo `n8_discotienda.zip` que se encuentra en el CD y cree el respectivo proyecto en Eclipse. Ejecute la clase `InterfazDiscotienda` (como siempre lo ha hecho) y utilice las distintas opciones que el programa le ofrece.

2. Consulte por Internet la razón por la cual el proceso de depuración se identifica con un pequeño bicho (*bug*). ¿De dónde viene la palabra "debugger"?

3. Localice el método `venderCancion()` de la clase `Discotienda`. Identifique la primera instrucción del método (`cancion.vender();`). En el costado izquierdo del editor de Java establezca un punto de parada (*breakpoint*), haciendo clic derecho sobre el costado (justo al lado de la instrucción) y seleccionando la opción del menú emergente **Toggle Breakpoint**. Debe aparecer un punto para indicar que hay un punto de parada en esa instrucción. Si sitúa el cursor sobre dicho punto, obtendrá información sobre el nombre del método y la línea en la que se encuentra localizado.

4. Localice en la vista **Package Explorer** la clase `InterfazDiscotienda`. Despliegue el menú emergente con el clic derecho y seleccione la opción **Debug As/Java Application**. Cuando aparezca la ventana del programa, utilice el botón **Vender Canción**. ¿Qué sucedió? Describa cada una de las vistas que aparecieron en Eclipse.

Vista "Debug":

Vista "Variables":

Vista "Breakpoints":

Vista "Help":

Vista "Console":

5. Localice en la vista "Debug" la barra de herramientas:



Con dichos botones vamos a controlar la ejecución del programa, que en este momento se encuentra detenida en el punto de parada que colocamos en el método `venderCancion()`. ¿Qué sucede si visualizamos en este momento la ventana del programa?

6. Utilice el botón para continuar la ejecución.

7. Usando la interfaz del programa venda otra canción, dando una dirección electrónica válida. El programa de nuevo se detiene en la primera instrucción del método `venderCancion()`. Utilice ahora el botón  (paso hacia adentro). Describa el punto de la ejecución del programa al cual llegó. ¿Qué hay ahora en la vista de variables?
8. Oprima dos veces más el mismo botón , hasta que la ejecución del programa regrese al método `venderCancion()`. ¿En qué instrucción se encuentra ahora el programa?
9. Utilice ahora el botón  (paso sobre o sin entrar). Fíjese que en lugar de entrar al cuerpo del método `indexOf()` pasó a la siguiente instrucción. Avance con este botón sobre algunas instrucciones del método y mire en la vista de variables la manera como los valores van cambiando. ¿Qué valor tiene la variable `posArroba1`?
10. Utilice el botón  (retornar del método). ¿A qué punto del programa llegó? ¿Qué hay ahora en la vista de variables?
11. Con el botón  se detiene la ejecución del programa. Éste es un botón que se debe utilizar con alguna precaución, porque no sigue el proceso normal de finalización del programa y, por ejemplo en nuestro caso, no se va a llamar el método que hace persistir el estado del modelo del mundo.
12. Elimine el punto de parada del método de venta de canciones (siguiendo el mismo proceso que usó para definirlo) y coloque ahora puntos de parada en la primera línea de los constructores de las clases `Discotienda`, `Disco` y `Cancion`. Ejecute de nuevo el programa en modo depuración usando el botón  de la barra de herramientas. Agregue discos y canciones a la tienda de discos y utilice los botones antes explicados para seguir la ejecución del programa y estudiar la manera como se puede visualizar el estado del modelo del mundo desde la vista de variables.

7. Glosario de Términos

GLOSARIO



Complete las siguientes definiciones como una manera de resumir los conceptos aprendidos en el nivel. Asegúrese de entender bien el significado de cada uno de estos términos, puesto que son la base para poder continuar con los niveles que siguen en el libro.

Persistencia:

Importar información:

Requerimiento no funcional:

Excepción:

Archivo secuencial:

Tipo de excepción:

Flujo de entrada:

Serialización:

Clase `File`:

Depurador:

Flujo de salida:

Lista plegable:

Exportar información:

Expresión condicional:

8. Hojas de Trabajo



8.1. Hoja de Trabajo № 1: Campeonato de Fórmula 1

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

En el campeonato de Fórmula 1 participan varios equipos, cada uno de los cuales tiene dos pilotos. Cada equipo por su parte tiene un nombre y un auto emblema. De cada piloto se conoce el nombre, el número del vehículo y el país de origen. Los pilotos tienen como emblema su casco.

En un campeonato se lleva a cabo una serie de carreras que se corren en diferentes pistas. De cada carrera se conoce el nombre de la carrera (por ejemplo "Gran Premio de San Marino"), el nombre de la pista (por ejemplo "Imola"), la longitud en metros de la pista, el número de vueltas que conforman la carrera y el récord de vuelta actual. Adicionalmente se tiene una imagen que muestra el recorrido.

La información del campeonato debe ser persistente y el proceso debe ser completamente transparente para el usuario. Esto quiere decir que el programa debe ser capaz de guardar la información en un archivo (por serialización) cada vez que el usuario termina la ejecución del mismo, y de utilizar dicha información cuando el usuario vuelve a ejecutarlo para reconstruir el estado del modelo del mundo. El programa no debe preguntarle al usuario el nombre del archivo, sino que lo tiene que manejar todo internamente.

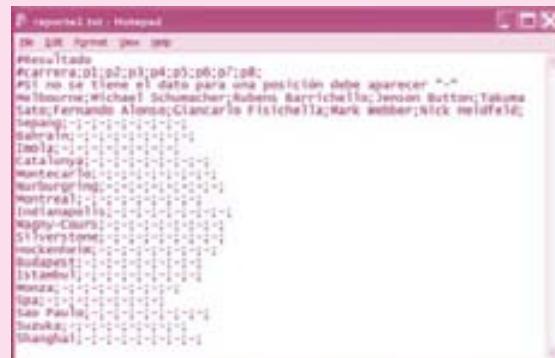
La interfaz de usuario del programa debe ser la que se muestra a continuación:

En cada carrera los pilotos que llegan en los ocho primeros lugares reciben el número de puntos que aparecen en la siguiente tabla:

Posición	Posición
1	10
2	8
3	6
4	5
5	4
6	3
7	2
8	1

Se quiere construir un programa que permita: (1) ingresar los resultados de una carrera, suministrando para esto los nombres de los pilotos que ocuparon los ocho primeros lugares; (2) informar el nombre del

piloto que va ganando el campeonato, teniendo en cuenta todos los resultados registrados, y (3) generar un reporte en un archivo de texto (cuyo nombre debe dar el usuario) con los resultados de todas las carreras del campeonato de Fórmula 1, utilizando para esto el formato que se ilustra en la siguiente figura:



En caso de error en el momento de leer o escribir el contenido de la persistencia, el programa debe agregar en un archivo de registro de errores un reporte completo con la descripción del problema.

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

Requerimiento funcional 1	Nombre	R1 – Ingresar los resultados de una carrera
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Informar el nombre del piloto que va ganando el campeonato
	Resumen	
	Entrada	
	Resultado	

Requerimiento funcional 3	Nombre	R3 – Generar un reporte con los resultados de las carreras
	Resumen	
	Entrada	
	Resultado	

Modelo conceptual. Construya el diagrama de clases, asegurándose de incluir las siguientes entidades: Campeonato, Carrera, Equipo y Piloto.

Invariante de clase. Defina el invariante para cada una de las clases del diagrama anterior.

Clase Campeonato	Clase Carrera	Clase Equipo	Clase Piloto

Requerimientos no funcionales. Identifique los requerimientos no funcionales planteados en el enunciado.

Manejo de archivos. A continuación se proponen varias tareas que van a permitir al lector estudiar la solución planteada al problema.

- Cree un proyecto en Eclipse a partir del archivo `n8_formula1.zip`, el cual se encuentra disponible en el CD que acompaña al libro. Localice la clase `InterfazFormula1` y ejecute desde allí el programa. Recorra la información disponible de carreras y registre un nuevo resultado. Salga del programa. Ejecútelo de nuevo y verifique que la información que fue ingresada anteriormente se encuentra presente. Edite desde Eclipse las clases `Campeonato` e `InterfazFormula1`.

¿En qué método se salva el estado del mundo? ¿Qué estructuras contenedoras hace persistir? ¿Qué método utiliza para hacerlo?

¿Cuál es el objetivo del método `registrarError()`? ¿Qué recibe este método como parámetro? ¿En qué casos es invocado este método?

¿En qué método se lee el estado del mundo? ¿Qué hace este método si no encuentra el archivo con la serialización?

¿Qué método impide que se termine el programa en caso de error en la persistencia? ¿Para qué sirve la instrucción `super.dispose()`?

- Ahora utilice desde el programa la opción de **Generar Reporte Resultados**. Indique el nombre del archivo donde desea guardar el reporte y haga clic en aceptar. Esto generará un reporte con los resultados de todas las carreras. Edite el archivo con el reporte y estudie su contenido. Edite desde Eclipse la clase `Campeonato` y la clase `Carrera`.

<p>¿Qué método de la clase Campeonato se utiliza para generar el reporte? ¿Qué recibe como parámetro este método?</p>	
<p>¿Qué método de la clase Carrera se utiliza para ayudar a generar el reporte? ¿Qué recibe como parámetro este método?</p>	
<p>3. Edite desde Eclipse las clases ElementoNoExisteException y PersistenciaException, las cuales implementan las excepciones que van a lanzar algunos métodos en caso de problemas.</p>	
<p>¿En qué situaciones se lanza cada una de estas excepciones?</p>	
<p>¿Qué métodos lanzan estas excepciones?</p>	
<p>4. Vamos ahora a estudiar los métodos que sirven para verificar el invariante de la clase Campeonato.</p>	
<p>¿Cuántas condiciones se verifican en el invariante?</p>	
<p>Explique el objetivo de cada uno de los cuatro métodos auxiliares que se utilizan.</p>	
<p>5. Por último vamos a estudiar algunos detalles de las clases que implementan el modelo del mundo.</p>	
<p>¿Para qué sirve la constante llamada serialVersionUID? ¿Qué valor se le asignó en cada clase del modelo del mundo? ¿Qué sucede en ejecución si se modifica dicho valor?</p>	
<p>¿Cuántos escenarios de prueba se construyeron para la clase Campeonato? ¿Cómo se construyen esos escenarios?</p>	
<p>¿Cuántos casos de prueba se desarrollaron para la clase Carrera?</p>	



8.2. Hoja de Trabajo N° 2: Mundial de Fútbol

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

Se quiere construir un programa para organizar la información del campeonato mundial de fútbol. La aplicación debe permitir manejar los datos de los equipos que participan y la ficha técnica de sus jugadores. Cada equipo tiene un país de origen, el nombre del director técnico y la ruta de la imagen con la bandera del país que representa. Para cada jugador perteneciente a un equipo se conoce el nombre, la edad, la altura, el peso, el salario, la ruta a la imagen con la foto del jugador y la posición en la que juega: delantero, centrocampista, defensa o arquero.

La interfaz de usuario del programa debe ser la siguiente:



El programa debe contar con las siguientes operaciones: (1) consultar la información de un jugador dados el nombre del equipo al que pertenece y el nombre del jugador, (2) agregar un equipo al mundial, el cual estará sin jugadores inicialmente; (3) agregar un jugador a un equipo (dados el nombre del equipo y toda la

información del jugador), (4) crear un archivo con un reporte, que incluya el valor de la nómina de un equipo dado. Este valor corresponde a la suma de los salarios de todos los jugadores de dicho equipo. Este archivo se debe crear en el directorio **data\reportes**, con el contenido que se ilustra a continuación.

```
Alemania_1174144499609.nómina - Notepad
File Edit Format View Help
Mundial_CUPI2 - Reporte Nómina - Equipo: Alemania
Fecha: Sat Mar 17 10:14:59 CDT 2007
Total Nómina: 78602.8 millones anuales
```

Por último, (5) el programa debe permitir modificar la información de los jugadores del mundial a partir de un archivo de texto. Toda la información de un jugador puede ser modificada exceptuando su nombre y el equipo al que pertenece. Dicho archivo debe tener la estructura que se muestra a continuación:

```
Jugadores.txt - Notepad
[Equipo]
[Nombre]
[Edad]
[Posición]
[Altura]
[Peso]
[Salario]
```

La información del mundial debe ser persistente y el proceso debe ser completamente transparente para el usuario. Esto quiere decir que el programa debe ser capaz de guardar la información (por serialización) en un archivo cada vez que el usuario termina la ejecución del mismo, y de utilizar dicha información cuando el usuario vuelve a ejecutarlo para reconstruir el estado del modelo del mundo. El programa no debe preguntarle al usuario el nombre del archivo, sino que lo tiene que manejar todo internamente. En caso de error en el momento de leer o escribir el contenido de la persistencia, el programa debe agregar en un archivo de registro de errores un reporte completo con la descripción del problema.

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

Requerimiento funcional 1	Nombre	R1 – Consultar la información de un jugador
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Agregar un equipo al mundial
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Ingresar los resultados de una carrera
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Crear un reporte con el valor de la nómina de un equipo dado
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Modificar la información de los jugadores a partir de un archivo de texto
	Resumen	
	Entrada	
	Resultado	

Modelo conceptual. Construya el diagrama de clases, asegurándose de incluir las siguientes entidades: Mundial, Equipo y Jugador.

Invariante de clase. Defina el invariante para cada una de las clases del diagrama anterior.

Clase Mundial	Clase Equipo	Clase Jugador

Requerimientos no funcionales. Identifique los requerimientos no funcionales planteados en el enunciado.

Manejo de archivos. A continuación se proponen varias tareas que van a permitir al lector estudiar la solución planteada al problema.

- Cree un proyecto en Eclipse a partir del archivo n8_mundial.zip, el cual se encuentra disponible en el CD que acompaña al libro. Localice la clase `InterfazMundial` y ejecute desde allí el programa. Recorra la información disponible de equipos y jugadores. Agregue un nuevo equipo con algunos jugadores. Salga del programa. Ejecútelo de nuevo y verifique que la información que fue ingresada anteriormente se encuentra presente. Edite desde Eclipse las clases `Mundial` e `InterfazMundial`.

¿En qué método se salva el estado del mundo? ¿Qué estructuras contenedoras hace persistir? ¿Qué método utiliza para hacerlo?

¿Cuál es el objetivo del método `registrarError()`? ¿Qué recibe este método como parámetro? ¿En qué casos es invocado este método?

¿En qué método se lee el estado del mundo? ¿Qué hace este método si no encuentra el archivo con la serialización?

¿Qué método impide que se termine el programa en caso de error en la persistencia? ¿Para qué sirve la instrucción `super.dispose()`?

- Ahora utilice desde el programa la opción de **Calcular Nómina** sobre alguno de los equipos. Localice en el directorio **data/reportes** el archivo generado. Edite desde Eclipse la clase `Mundial` y la clase `Equipo`.

¿Bajo qué nombre fue creado el archivo? ¿El contenido es coherente con el respectivo requerimiento?

¿Qué método de la clase `Mundial` se utiliza para generar el reporte? ¿Qué recibe como parámetro este método?

¿Qué método de la clase `Equipo` se utiliza para ayudar a generar el reporte? ¿Cómo calcula este método el monto total de la nómina?

- Edite desde Eclipse las clases `ArchivoJugadoresException`, `PersistenciaException` y `ElementoExisteException`, las cuales implementan las excepciones que van a lanzar algunos métodos en caso de problemas.

¿En qué situaciones se lanza cada una de estas excepciones?

¿Qué métodos lanzan estas excepciones?

4. Vamos ahora a estudiar los métodos que sirven para verificar el invariante de la clase Mundial.

¿Cuántas condiciones se verifican en el invariante?

Explique el objetivo del método auxiliar que utiliza.

5. Localice en la carpeta **data** el archivo **jugadores.txt** y ábralo desde Eclipse o desde cualquier otro editor de texto. Analice el contenido del archivo. Utilice en el programa la opción **Modificar Jugadores**, seleccione el archivo anterior y mire la manera como los datos de algunos jugadores fueron actualizados a partir de dicha información. Edite las clases Mundial, Equipo y Jugador.

¿Qué recibe como parámetro el método `modificarInformacionJugadores()`? ¿Qué excepciones lanza y en qué casos?

¿Qué recibe como parámetro el método `modificarJugador()`? ¿Qué responsabilidades tiene este método? ¿Qué excepciones lanza y en qué casos?

¿Qué métodos de la clase Equipo se invocan para modificar la información? ¿Qué parámetros recibe? ¿Qué precondición cumplen dichos parámetros?

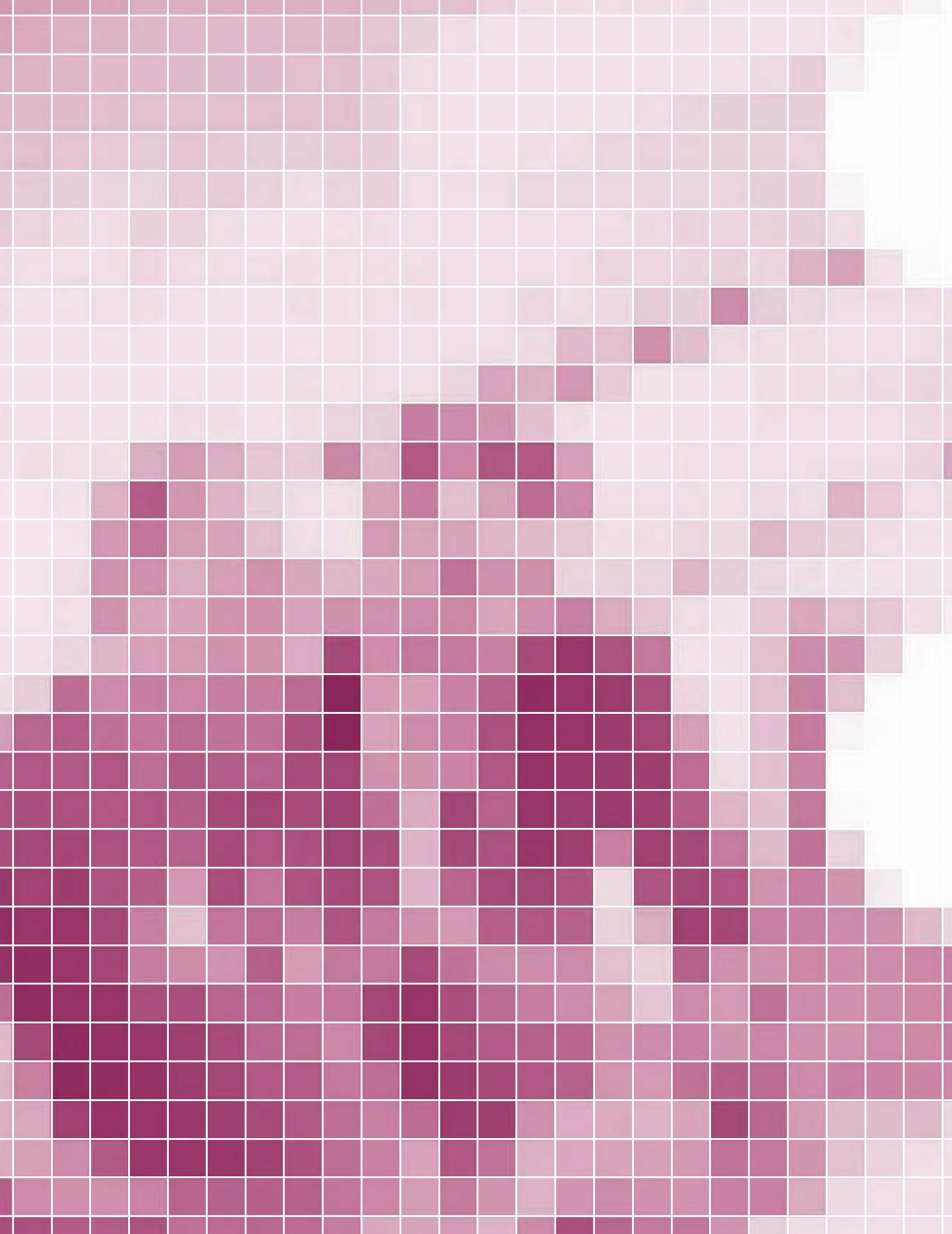
¿Qué métodos de la clase Jugador se invocan desde la clase Equipo para modificar la información? ¿Qué parámetros recibe? ¿Qué precondición cumplen dichos parámetros?

6. Por último vamos a estudiar algunos detalles de las clases que implementan el modelo del mundo.

¿Para qué sirve la constante llamada `serialVersionUID`? ¿Qué valor se le asignó en cada clase del modelo del mundo? ¿Qué sucede en ejecución si se modifica dicho valor?

¿Cuántos escenarios de prueba se construyeron para la clase Mundial? ¿Cómo se construyen esos escenarios?

¿Cuántos casos de prueba se desarrollaron para la clase Equipo?



Nivel 3

Estructuras Lineales Enlazadas

1. Objetivos Pedagógicos

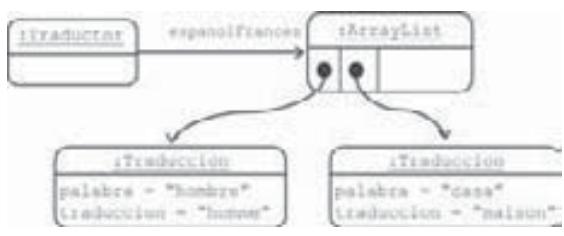
Al final de este nivel el lector será capaz de:

- Utilizar estructuras enlazadas de objetos para modelar grupos de atributos de longitud indeterminada.
- Escribir los algoritmos necesarios para manipular estructuras lineales que almacenan sus elementos enlazándolos entre ellos.
- Construir interfaces de usuario más complejas, utilizando nuevos distribuidores gráficos y nuevos componentes, que van a facilitar el despliegue de información y la interacción con el usuario.

2. Motivación

Si volvemos atrás y recordamos el caso de estudio del primer nivel, en el que queríamos construir un traductor de idiomas, nos encontramos con que cada uno de los diccionarios estaba representado con un vector, tal como se muestra en la figura 3.1. Allí podemos apreciar que un objeto de la clase `Traductor` tenía una referencia a un objeto de la clase `ArrayList`, el cual mantenía una secuencia de referencias a las traducciones.

Fig. 3.1 – **Diagrama de objetos para el caso del traductor de idiomas**



El diagrama de clases para representar un diccionario como una estructura lineal enlazada aparece en la figura 3.3b. Fíjese cómo transformamos el diagrama de clases del análisis (figura 3.3a) para indicar que en lugar de ma-

En esta sección vamos a estudiar una manera alternativa de representar grupos de elementos, en la cual, en lugar de tener un objeto contenedor (un `ArrayList` en el ejemplo anterior), vamos a delegar la responsabilidad a los mismos elementos contenidos de saber quién más pertenece al grupo. Esta idea se ilustra en la figura 3.2 para el mismo caso del traductor de idiomas. En la figura se puede ver que el objeto de la clase `Traductor` sólo conoce la primera de las traducciones, y que cada una de ellas conoce la siguiente traducción del diccionario.

Fig. 3.2 – **Diagrama de objetos para el caso del traductor de idiomas usando una estructura lineal enlazada**



nejar una contenedora (cardinalidad *) vamos a manejar una estructura enlazada. Ahora la clase `Traduccion` tiene una asociación a ella misma llamada `siguiente`, que va a permitir navegar por los elementos del grupo.

Fig. 3.3 – **Diagrama de clases del análisis y diagrama de clases del diseño**

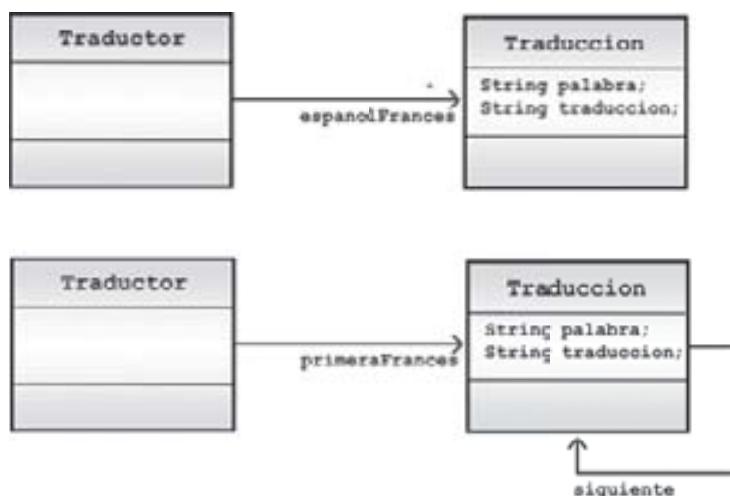


Fig. 3.3a: Aquí aparece una parte del diagrama de clases obtenido en la etapa de análisis: un traductor tiene una asociación hacia un grupo de objetos de la clase `Traduccion`. Dicha asociación se denomina “españolFrances” y contiene todas las traducciones de español a francés.

Fig. 3.3b: En esta figura aparece el diagrama de clases transformado, en el cual la clase `Traductor` sólo tiene una referencia a la primera de las traducciones del diccionario (“`primeraFrances`”) y la clase `Traduccion` se encarga de indicar cuál es la siguiente.

En la primera parte de este nivel estudiaremos la manera de agregar, eliminar y buscar elementos en una estructura enlazada, y de resolver todos los problemas algorítmicos asociados con el manejo de este tipo de representación. En la segunda parte nos concentraremos en lo que se denominan estructuras doblemente enlazadas, en las cuales cada elemento conoce no sólo el siguiente elemento de la secuencia, sino también el anterior.

La idea de enlazar los objetos se puede generalizar a estructuras mucho más complejas y nos va a permitir en niveles posteriores utilizar estructuras recursivas o con múltiples encadenamientos. Por ahora nos vamos a concentrar únicamente en las estructuras lineales y en la algorítmica básica para su manejo.

3. Caso de Estudio N° 1: Una Central de Pacientes

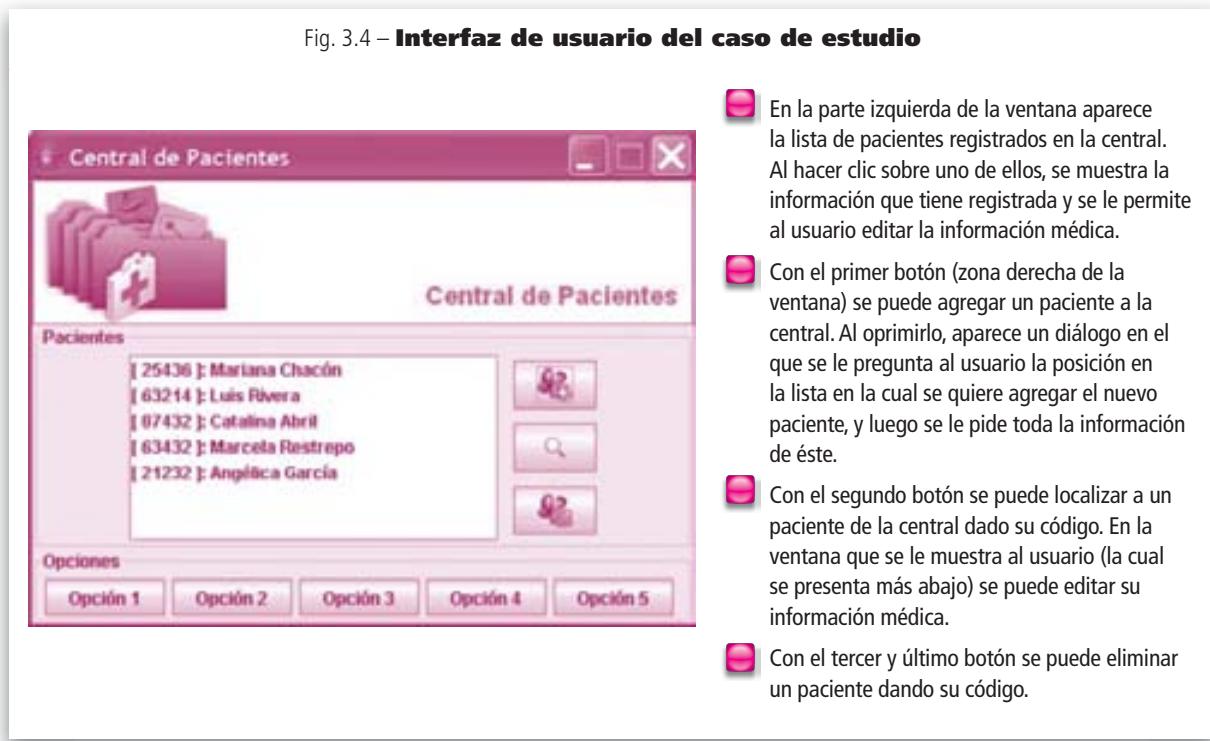
Un cliente nos pide que desarrollemos un programa de computador para administrar la información de

un grupo de pacientes que son atendidos en distintas clínicas de la ciudad. De cada paciente la aplicación debe almacenar un código numérico que es único (76527, por ejemplo), un nombre ("Mariana Chacón"), el nombre de la clínica a la cual fue remitido el paciente ("Clínica Reina Sofía"), la información médica del paciente ("Dolor de oído") y el sexo (Femenino). La interfaz de usuario del programa es la que se presenta en la figura 3.4.

Se espera que el programa sea capaz de agregar un paciente a la central, contemplando cuatro variantes posibles: (1) insertar al comienzo de la lista de pacientes, (2) insertar al final, (3) insertar después de otro paciente (dado su código) o (4) insertar antes de otro paciente (dado su código). El programa también debe permitir (5) eliminar un paciente, (6) consultar sus datos o (7) editar su información médica, para lo cual el usuario debe suministrar el respectivo código del paciente.

Por facilidad de implementación, no se pide que la información sea persistente.

Fig. 3.4 – Interfaz de usuario del caso de estudio





- En esta ventana de diálogo aparece la información de un paciente. Los campos que tienen el nombre, el código, la clínica y el sexo del paciente no se pueden modificar.
- El único campo que se puede cambiar es el que contiene la información médica del paciente. Para que los cambios se hagan efectivos, se debe utilizar el botón **Registrar Cambios**.
- Para la selección del sexo del paciente se utiliza un componente gráfico que impide que se seleccionen las dos opciones al mismo tiempo.

3.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none"> ■ Insertar, eliminar y buscar un paciente en una estructura lineal enlazada. 	<ul style="list-style-type: none"> ■ Estudiar la algorítmica de manejo de las estructuras lineales enlazadas e identificar los patrones de algoritmo que permiten resolver problemas sobre dichas estructuras.
<ul style="list-style-type: none"> ■ Presentar al usuario la información completa de un paciente en una nueva ventana. 	<ul style="list-style-type: none"> ■ Estudiar el componente JDialoG del lenguaje de programación Java.
<ul style="list-style-type: none"> ■ Permitir al usuario seleccionar una opción de un conjunto de opciones excluyentes. 	<ul style="list-style-type: none"> ■ Estudiar los componentes de interacción JRadioButton y ButtonGroup del lenguaje de programación Java.

3.2. Comprensión de los Requerimientos

**Tarea 1**

Objetivo: Entender el problema del caso de estudio.

(1) Lea detenidamente el enunciado del caso de estudio e (2) identifique y complete la documentación de los siete requerimientos funcionales que allí aparecen.

Requerimiento funcional 1	Nombre	R1 – Insertar un paciente al comienzo de la lista
	Resumen	
	Entrada	
	Resultado	

Requerimiento funcional 2	Nombre	R2 – Insertar un paciente al final de la lista
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Insertar un paciente después de otro paciente de la lista
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Insertar un paciente antes de otro paciente de la lista
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Eliminar un paciente de la lista
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 6	Nombre	R6 – Consultar los datos de un paciente
	Resumen	
	Entrada	
	Resultado	

Requerimiento funcional 7	Nombre	R7 – Modificar la información médica de un paciente
	Resumen	
	Entrada	
	Resultado	

3.3. Referencias y Ciclo de Vida de los Objetos

Antes de abordar el tema de las estructuras enlazadas, es importante repasar la diferencia que existe entre un objeto y una referencia a éste. De igual forma, debemos definir claramente lo que quiere decir que un objeto deje de existir, y determinar el momento en el cual el espacio que el objeto ocupa en memoria es recuperado por el computador. Éstos son los dos temas que trataremos en esta sección.

Un objeto es una instancia de una clase y ocupa un espacio en la memoria del computador, en donde almacena físicamente los valores de sus atributos y asociaciones. La clase es la que define el espacio en memoria que necesita cada uno de sus objetos, puesto que conoce la cantidad de información que éstos deben almacenar. Una referencia es un nombre mediante el cual podemos señalar o indicar un objeto particular en la memoria del computador. Una variable de una clase *c1* declarada dentro de un método cualquiera no es otra cosa que una referencia temporal a un objeto de dicha clase. En la figura 3.5 presentamos una sintaxis gráfica que nos va a permitir mostrar referencias a objetos. Allí aparecen cuatro objetos de la clase *c1*. El primero de los objetos es señalado o apuntado por dos referencias llamadas *ref1* y *ref2*. El segundo de los objetos no tiene referencias. El tercer objeto está siendo referenciado por *ref3* y tiene una asociación a un objeto de la misma clase que se materializa como una referencia, la cual, en lugar de estar almacenada en una variable, está almacenada en un atributo.

Fig. 3.5 – Sintaxis gráfica para mostrar referencias a objetos



Cuando usamos la sintaxis *ref1.m1()* estamos pidiendo la invocación del método *m1()* sobre el objeto que se encuentra referenciado por *ref1*. Es importante recordar que los objetos sólo se pueden acceder a través de referencias, lo que implica que si existe un objeto sin referencias hacia él, éste deja de ser utilizable dentro de un programa.

El **recolector de basura** (*garbage collector*) de Java es un proceso de la máquina virtual del lenguaje que recorre periódicamente la memoria del computador buscando y eliminando los objetos que no son referenciados desde ningún punto. Por esta razón, si queremos “destruir” un objeto porque ya no es útil dentro de un programa, sólo debemos asegurarnos de que no existe ninguna referencia hacia él. El resto del trabajo lo hace de manera automática el recolector de basura.

En el ejemplo 1 mostramos la manera como ciertas instrucciones de un programa afectan el estado de la memoria del computador.

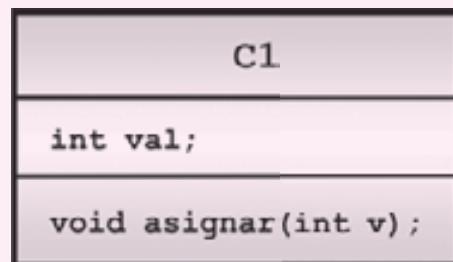
Ejemplo 1

Objetivo: Mostrar la manera como ciertas instrucciones del lenguaje Java manejan referencias y modifican el estado de los objetos.

En este ejemplo utilizamos la clase C1, cuya declaración se encuentra a continuación. Luego, mostramos de manera incremental la forma como las instrucciones de un programa afectan el estado de la memoria del computador. Por último, presentamos algunas características que se deben tener en cuenta en el momento de comparar objetos y referencias.

```
public class C1
{
    private int val;

    public void asignar( int v )
    {
        val = v;
    }
}
```



Instrucción	Estado de la memoria
C1 v1 = new C1();	<p>Diagram illustrating the memory state after the instruction <code>C1 v1 = new C1();</code>. A variable <code>v1</code> points to a new object instance of type <code>C1</code>, which has a <code>val</code> field initialized to <code>null</code>.</p>
C1 v2 = v1;	<p>Diagram illustrating the memory state after the instruction <code>C1 v2 = v1;</code>. Both variables <code>v1</code> and <code>v2</code> now point to the same <code>C1</code> object instance, which still has a <code>val</code> field initialized to <code>null</code>.</p>
C1 v3 = null;	<p>Diagram illustrating the memory state after the instruction <code>C1 v3 = null;</code>. Variable <code>v3</code> points to a <code>null</code> value, while <code>v1</code> and <code>v2</code> still point to the shared <code>C1</code> object instance.</p>
v2.asignar(5); v1 = v3;	<p>Diagram illustrating the memory state after the instructions <code>v2.asignar(5);</code> and <code>v1 = v3;</code>. The <code>C1</code> object instance pointed to by <code>v2</code> now has its <code>val</code> field set to <code>5</code>. Variable <code>v1</code> still points to the original <code>C1</code> object, and <code>v3</code> points to <code>null</code>.</p>
ArrayList a1 = new ArrayList();	<p>Diagram illustrating the memory state after the instruction <code>ArrayList a1 = new ArrayList();</code>. A variable <code>a1</code> points to a new <code>ArrayList</code> object instance, which contains three empty slots.</p>
a1.add(v2);	<p>Diagram illustrating the memory state after the instruction <code>a1.add(v2);</code>. The <code>ArrayList</code> object <code>a1</code> now contains the <code>C1</code> object instance pointed to by <code>v2</code>. The <code>C1</code> object's <code>val</code> field is still <code>5</code>. Variables <code>v2</code> and <code>v3</code> remain pointing to the original <code>C1</code> object.</p>
v1 = new C1(); v1.asignar(1);	<p>Diagram illustrating the final memory state after the instructions <code>v1 = new C1();</code> and <code>v1.asignar(1);</code>. The <code>ArrayList</code> object <code>a1</code> now contains two <code>C1</code> objects. The first <code>C1</code> object (pointed to by <code>v2</code>) has a <code>val</code> of <code>5</code>. The second <code>C1</code> object (pointed to by <code>v1</code>) has a <code>val</code> of <code>1</code>. Variable <code>v3</code> still points to <code>null</code>.</p>

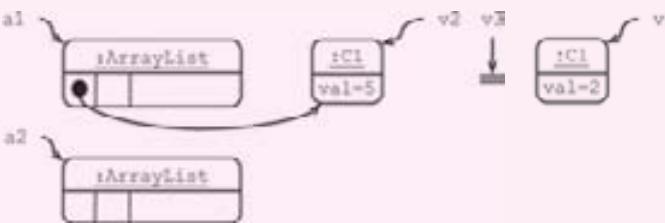
```
v1 = new C1( );
v1.asignar( 2 );
```



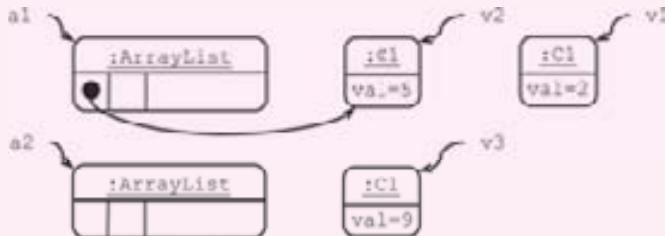
```
// Ejecución del recolector de
// basura de Java
```



```
ArrayList a2 = new ArrayList( );
```



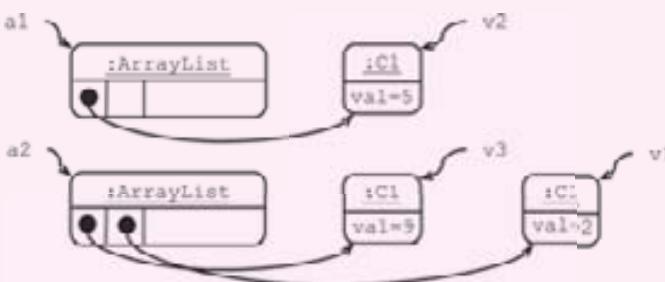
```
v3 = new C1( );
v3.asignar( 9 );
```



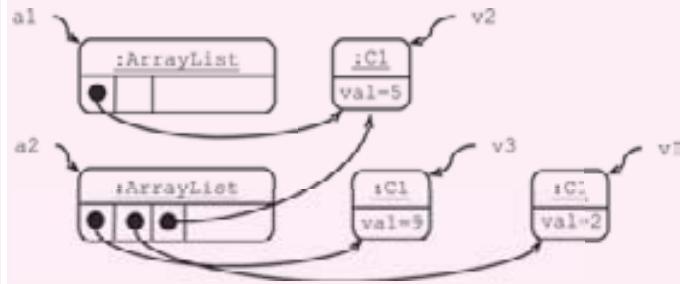
```
a2.add( v3 );
```



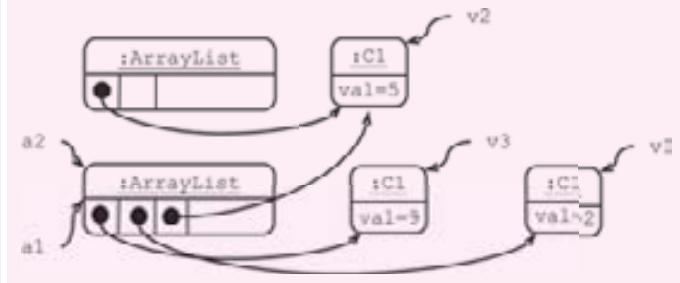
```
a2.add( v1 );
```



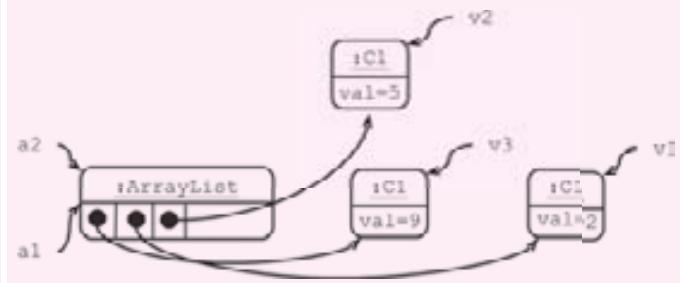
```
a2.add( v2 );
```



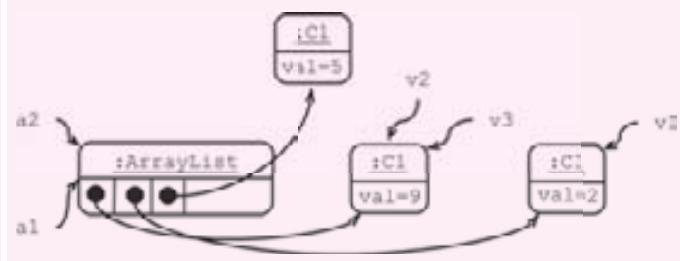
```
a1 = a2;
```



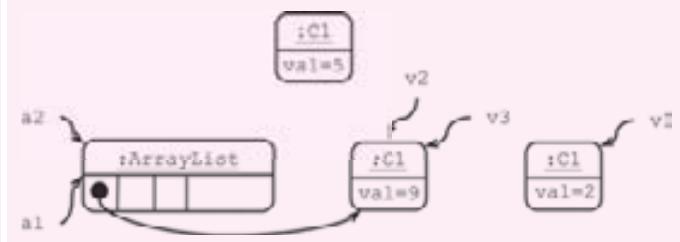
```
// Ejecución del recolector de  
// basura de Java
```



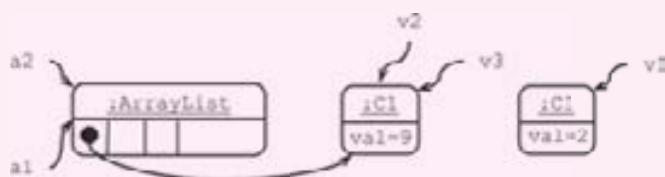
```
v2 = ( C1 )a2.get( 0 );
```



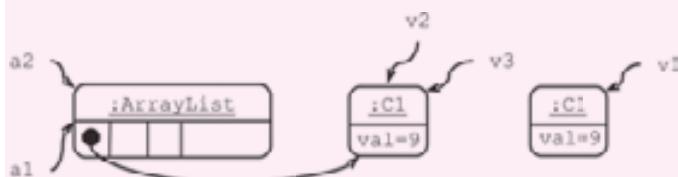
```
a2.remove( 2 );  
a2.remove( 1 );
```



```
// Ejecución del recolector de
// basura de Java
```



```
v1.asignar( 9 );
```



Expresión

Explicación

Aprovechemos ahora el diagrama de objetos al que llegamos con la secuencia anterior de instrucciones para repasar la manera de comparar objetos. En la columna de la izquierda aparece una expresión y en la parte derecha su evaluación.

v2 == v3

Verdadero. Cuando utilizamos el operador == entre referencias, estamos preguntando si físicamente están señalando el mismo objeto.

v1 == v2

Falso. Incluso si los objetos referenciados desde v1 y v2 tienen el mismo valor en su único atributo (9), el operador == sólo verifica si las dos referencias llegan al mismo objeto en memoria. Si queremos que la noción de igualdad sea más general, debemos implementar en la clase C1 el método equals(), tal como se muestra más adelante.

a1.get(0) == v3

Verdadero. El método get() retorna una referencia al mismo objeto referenciado por v3.

a1 == a2

Verdadero. Ambas referencias llegan al mismo objeto de la clase ArrayList.

En algunos casos, nos interesa crear una copia de un objeto con los mismos valores del objeto original. Este proceso se denomina **clonación** y se logra implementando el método `clone()` en la respectiva clase. En

el ejemplo 2 mostramos la manera de extender la clase C1 del ejemplo anterior, con un método de comparación y un método de clonación.

Ejemplo 2

Objetivo: Mostrar la manera de implementar un método de comparación y un método de clonación para la clase del ejemplo anterior.

En este ejemplo extendemos la clase C1 con un método que permite comparar dos instancias de la clase, teniendo en cuenta su estado interno. También mostramos la implementación de un método que permite clonar objetos.

```
public class C1
{
    private int val;

    public void asignar( int v )
    {
        val = v;
    }
    public boolean equals( C1 obj )
    {
        return val == obj.val;
    }

    public Object clone( )
    {
        C1 temp = new C1( );
        temp.val = val;
        return temp;
    }
}
```

Para implementar el método que compara dos objetos de la clase, teniendo en cuenta su estado interno, lo primero que debemos preguntarnos es cuándo se considera que dos objetos de la clase C1 son iguales. En nuestro ejemplo es simple, y es cuando tengan el mismo valor en su único atributo (val). Eso es lo que se refleja en la implementación del método equals(). Fíjese que los atributos del parámetro (obj) se pueden acceder directamente, puesto que dicho objeto pertenece a la misma clase que estamos definiendo.

El método de clonación debe crear una nueva instancia de la clase y pasárle el estado del objeto. En la signatura de este método se debe declarar que retorna un objeto de la clase Object, algo que estudiaremos en el siguiente nivel. Aquí también accedemos directamente al atributo "val" del objeto "temp", sin necesidad de pasar por el método de asignación.

El uso de los métodos antes descritos se ilustra en el siguiente fragmento de programa:

```
C1 v1 = new C1( );
v1.asignar( 9 );
C1 v2 = ( C1 )v1.clone( );
if( v1.equals( v2 ) )
    ...

```

Note que al usar el método clone() debemos aplicar el operador de conversión (C1) antes de hacer la asignación.

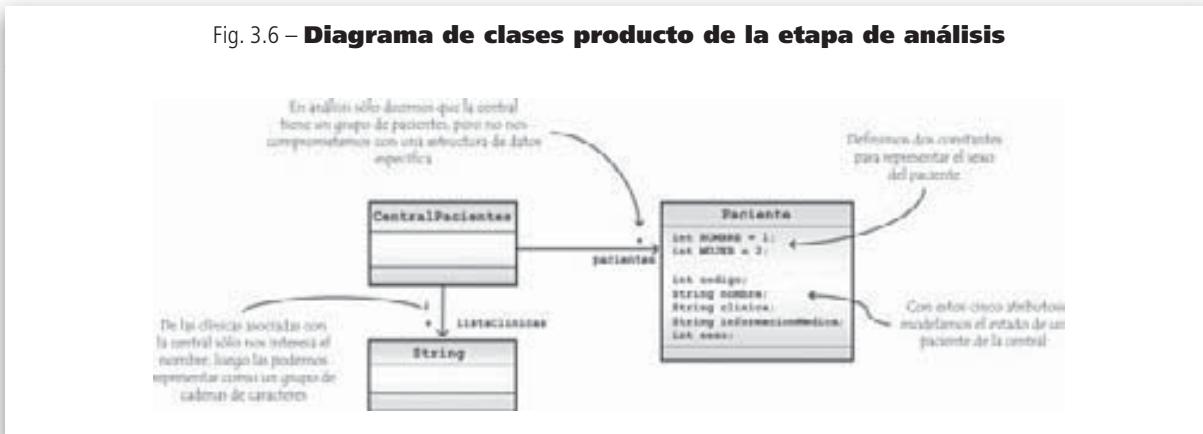
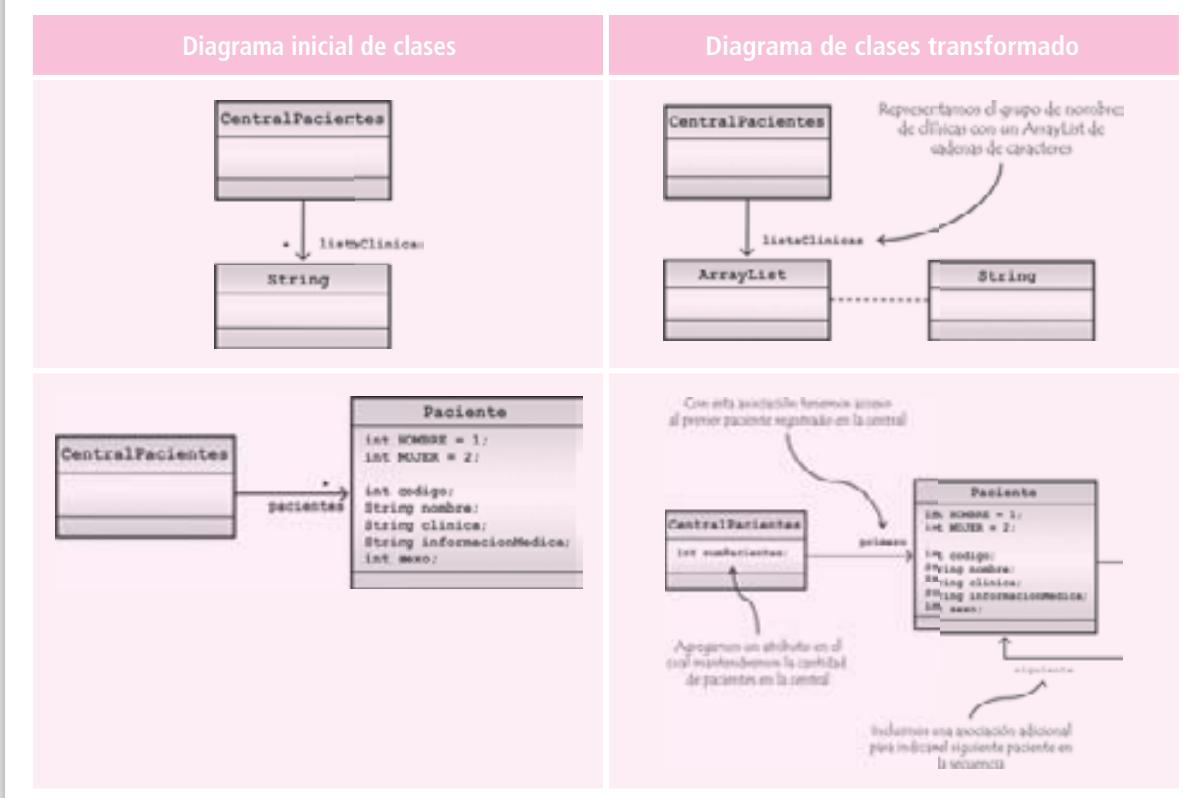
3.4. Del Análisis al Diseño

Al terminar la etapa de análisis de nuestro caso de estudio encontramos dos entidades: la central de pacientes y el paciente, con los atributos y asociaciones que aparecen en la figura 3.6. En dicho diagrama hay dos asociaciones con cardinalidad múltiple (pacientes y listaClinicas), para las cuales

es necesario tomar una decisión de diseño antes de iniciar la implementación. Lo demás corresponde a atributos de tipo simple o a objetos de la clase string, para los cuales no hay ninguna decisión adicional que tomar.

Hasta este momento, el invariante de las dos entidades identificadas en el análisis es el siguiente:

CentralPaciente:	Paciente:
<ul style="list-style-type: none"> Los códigos de los pacientes son únicos en la central 	<ul style="list-style-type: none"> codigo >= 0 nombre != null && nombre != "" clinica != null && clinica != "" informacionMedica != null sexo == HOMBRE o sexo == MUJER

Fig. 3.6 – **Diagrama de clases producto de la etapa de análisis**Fig. 3.7 – **Transformación del diagrama de clases como parte de la etapa de diseño**

El **diseño** es una etapa dentro del proceso de desarrollo de software en la cual transformamos el diagrama de clases resultado de la etapa de análisis con el fin de incorporar los requerimientos no funcionales definidos por el cliente (persistencia o distribución, por ejemplo), lo mismo que algunos criterios de calidad de la solución (eficiencia, claridad, etc.). Aquí, en particular, debemos definir las estructuras concretas de datos con las cuales vamos a representar los agrupamientos de objetos. Para el caso de estudio, las dos decisiones de diseño se pueden resumir en la figura 3.7.

Por un lado, representaremos el agrupamiento de clínicas con un `ArrayList` de cadenas de caracteres, en el que vamos a almacenar el nombre de cada una de ellas. Por otro lado, utilizaremos una estructura encadenada para representar la lista de pacientes registrados en la central. Adicionalmente, vamos a agregar un atributo en la clase `CentralPacientes` con el número de pacientes que allí se encuentran.

Las dos clases se declaran en Java de la siguiente manera:

```
public class CentralPacientes
{
    // -----
    // Atributos
    // -----

    private Paciente primero;
    private int numPacientes;
    private ArrayList listaClinicas;
}
```

```
public class Paciente
{
    // -----
    // Constantes
    // -----

    public final static int HOMBRE = 1;
    public final static int MUJER = 2;

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

    private int codigo;
    private String nombre;
    private String clinica;
    private String informacionMedica;
    private int sexo;
    private Paciente siguiente;
}
```

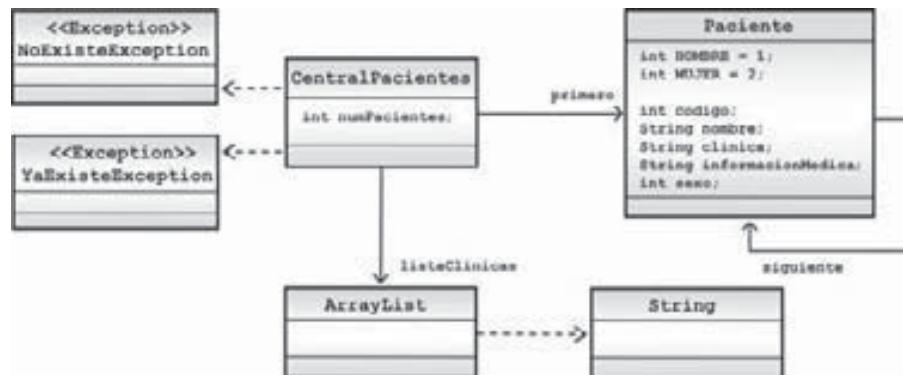
Extendemos ahora el invariante que habíamos calculado en el análisis, para incluir los elementos que aparecieron en la etapa de diseño:

CentralPaciente:	Paciente:
<ul style="list-style-type: none"> Los códigos de los pacientes son únicos en la central <code>numPacientes == longitud de la lista de pacientes</code> <code>listaClinicas != null</code> 	<ul style="list-style-type: none"> <code>codigo >= 0</code> <code>nombre != null && nombre != ""</code> <code>clinica != null && clinica != ""</code> <code>informacionMedica != null</code> <code>sexo == HOMBRE o sexo == MUJER</code>

Pasamos ahora a diseñar las excepciones que deben lanzar los distintos métodos de las dos clases. Para esto incluimos una excepción llamada `YaExisteException`, que indica que se trató de agregar un paciente con un código que ya existe en la central y una excepción

llamada `NoExisteException`, que indica que el paciente dado como referencia ("antes de" o "después de") para insertar el nuevo paciente no existe. Con esto completamos el diagrama de clases que se muestra en la figura 3.8.

Fig. 3.8 – **Diagrama de clases final para el caso de estudio**

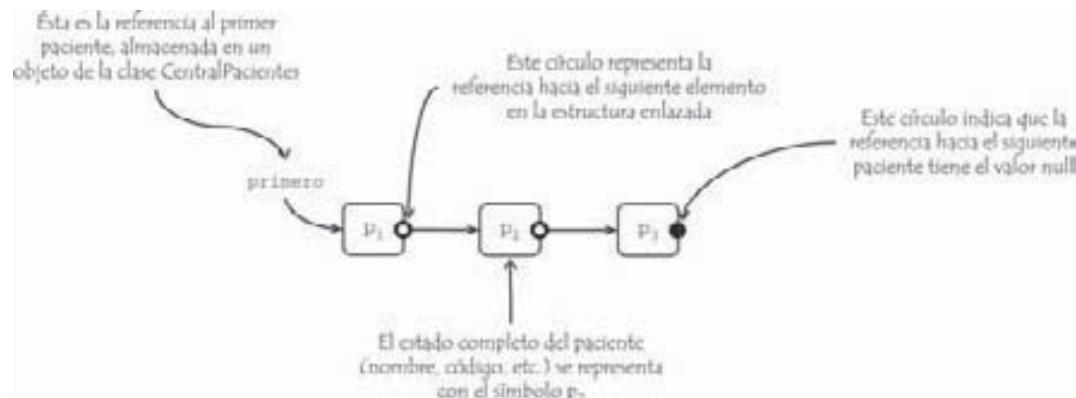


3.5. Estructuras Lineales Enlazadas

Vamos a introducir en esta sección una notación que corresponde a una simplificación de los diagramas de objetos de UML, para mostrar con ella los principales procesos que se tienen cuando se quiere manipular una estructura lineal sencillamente enlazada. Dicha notación se muestra en la figura 3.9, en donde apa-

rece una central con tres pacientes registrados. Puesto que no estamos interesados en los valores exactos de los atributos de cada paciente, sino en el enlazamiento de la estructura, representamos todo el estado de cada paciente mediante un solo símbolo (p_1 , p_2 y p_3). La referencia al primer paciente la llamamos **primero**, que corresponde al nombre de la asociación que tiene la clase `CentralPacientes` hacia la estructura enlazada.

Fig. 3.9 – **Diagrama de objetos simplificado para representar la lista de pacientes**



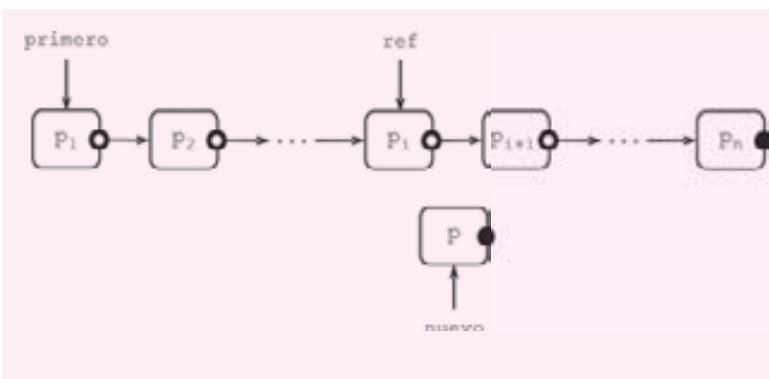
Para representar el caso general, en el cual en la lista hay n pacientes enlazados, utilizaremos la notación que se presenta en la figura 3.10.

Fig. 3.10 – **Caso general del diagrama de objetos simplificado**



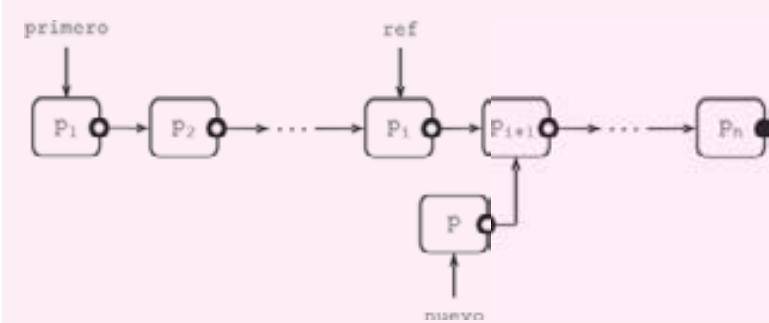
Existen tres procesos fundamentales de modificación de una estructura enlazada, los cuales se ilustran a continuación:

- Insertar un elemento después de otro del cual tenemos una referencia:

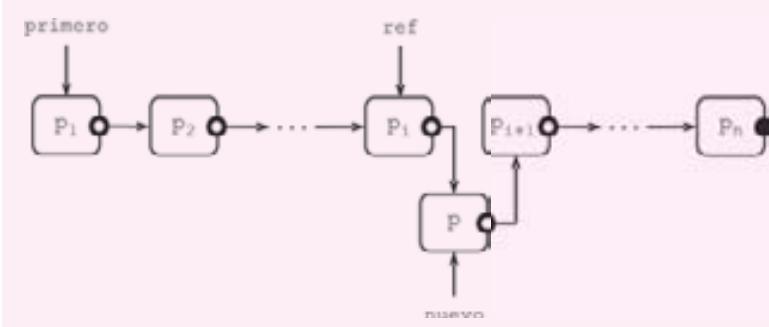


Situación inicial:
tenemos una estructura enlazada, una referencia al punto de inserción (ref) y un nuevo elemento señalado por la referencia "nuevo".

Queremos insertar en la lista el nuevo paciente, después del que se encuentra señalado por la referencia "ref".

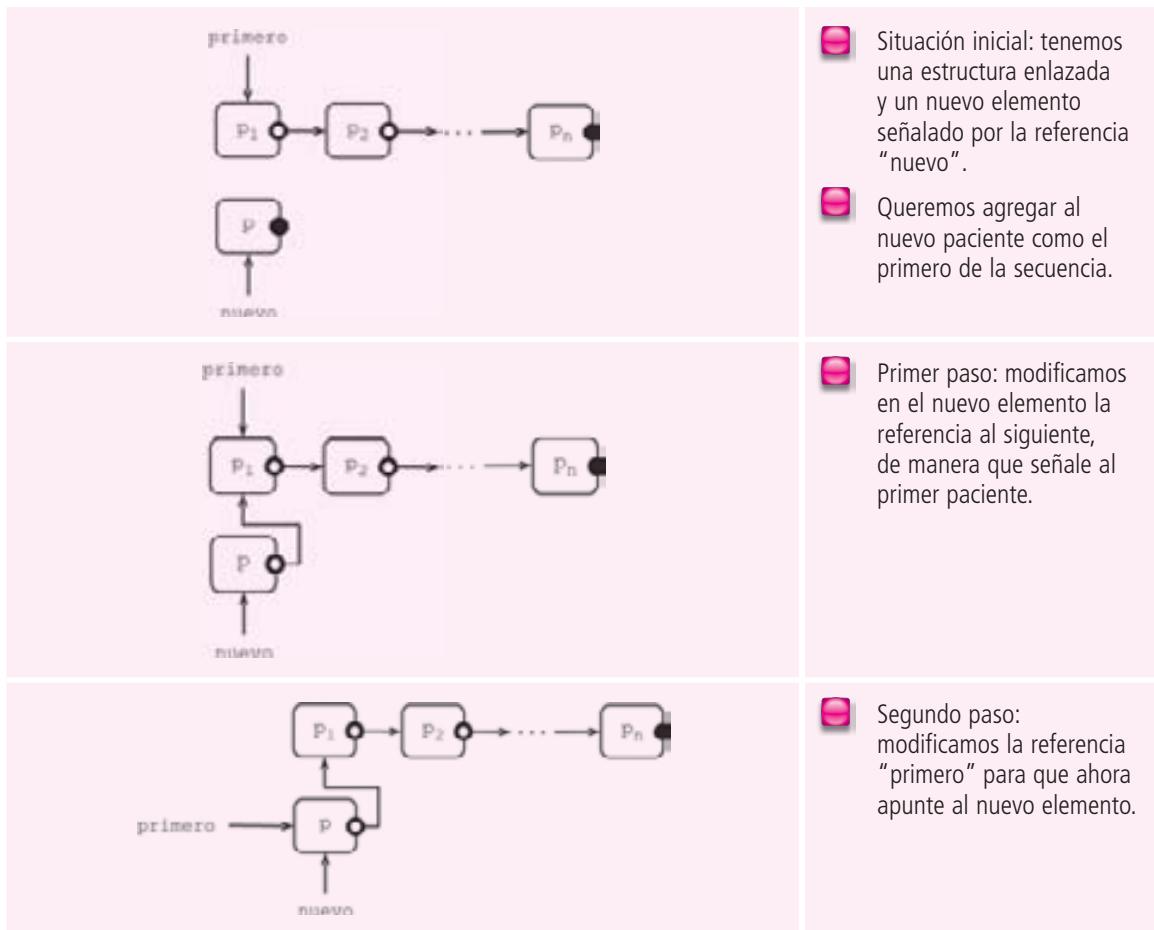


Primer paso: en el nuevo elemento, modificamos la referencia al siguiente paciente, de manera que señale al paciente que está en la posición "i+1".

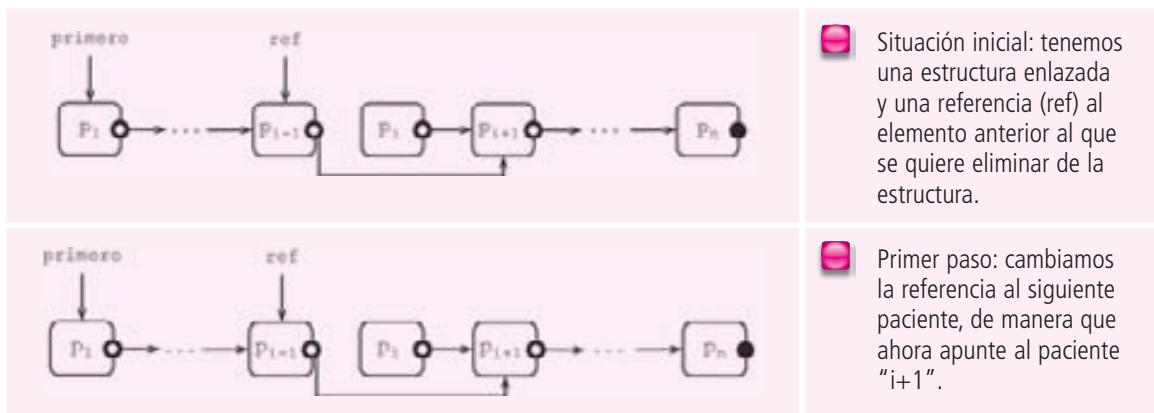


Segundo paso:
modificamos la referencia entre el paciente "i" y el paciente "i+1" de manera que ahora se tenga en cuenta el nuevo elemento.

- Insertar un elemento como el primero de la secuencia:



- Eliminar un elemento, teniendo una referencia al anterior:



En una estructura enlazada se denomina **longitud** al número de elementos que tiene encadenados.

3.6. Algorítmica Básica

Pasemos ahora a desarrollar los métodos que implementan las operaciones básicas de manejo de una estructura enlazada. Comenzamos con las operaciones de localización y después abordaremos la problemática asociada con la inserción y supresión de elementos.

Vale la pena resaltar que no hay una teoría o unas instrucciones especiales para manejar este tipo de estructuras. Simplemente vamos a aprovechar la capacidad que tienen

los lenguajes de programación para manipular referencias a los objetos que se encuentran en su memoria.

3.6.1. Localización de Elementos y Recorridos

Vamos a estudiar tres algoritmos de localización y un algoritmo de recorrido total, los cuales serán presentados a lo largo de los siguientes cuatro ejemplos. Supondremos para su implementación que en la clase Paciente están definidos los siguientes métodos:

Clase Paciente:	
Paciente(int cod, String nom, String clin, String infoMed, int sex)	■ Éste es el constructor de la clase. Recibe toda la información del paciente y crea un nuevo objeto, con el atributo "siguiente" en null. En la precondición exige que la información suministrada sea válida.
int darCodigo()	■ Este método retorna el código del paciente.
String darNombre()	■ Este método retorna el nombre del paciente.
String darClinica()	■ Este método retorna el nombre de la clínica a la cual fue enviado el paciente.
int darSexo()	■ Este método retorna el sexo del paciente.
String darInformacionMedica()	■ Este método retorna la información médica del paciente.
Paciente darSiguiente()	■ Este método retorna el siguiente paciente de la estructura enlazada. Si es el último paciente de la secuencia, retorna null.

**Ejemplo 3**

Objetivo: Implementar el método de la clase `CentralPacientes` que permite localizar un paciente dado su código.

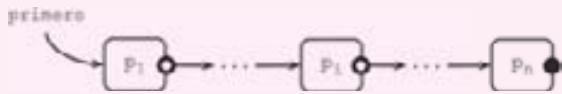
En este ejemplo presentamos el código del método e ilustramos gráficamente su funcionalidad.

```
public Paciente localizar( int codigo )
{
    Paciente actual = primero;

    while( actual != null &&
           actual.darCodigo( ) != codigo )
    {
        actual = actual.darSiguiente( );
    }

    return actual;
}
```

Inicial:



Llamado:

Paciente p = central.localizar(cod);

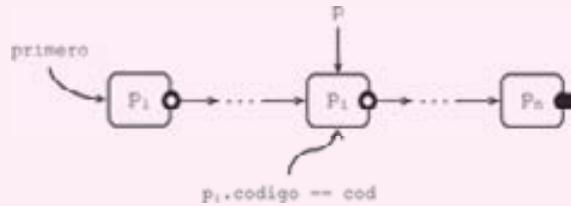
Utilizamos una referencia auxiliar llamada "actual" para hacer el recorrido de la estructura. Dicha referencia comienza apuntando al primer elemento.

El ciclo lo repetimos mientras que la referencia auxiliar sea distinta de null y mientras que no hayamos llegado al elemento que estamos buscando.

El avance del ciclo consiste en mover la referencia al siguiente paciente de la lista.

Al final retornamos el valor en el que termine la referencia auxiliar: si encontró un paciente con el código pedido, lo retorna. En caso contrario, retorna null.

Final:

**Ejemplo 4**

Objetivo: Implementar el método de la clase `CentralPacientes` que permite localizar el último paciente de la estructura enlazada.

En este ejemplo presentamos el código del método e ilustramos gráficamente su funcionalidad.

```
public Paciente localizarUltimo( )
{
    Paciente actual = primero;

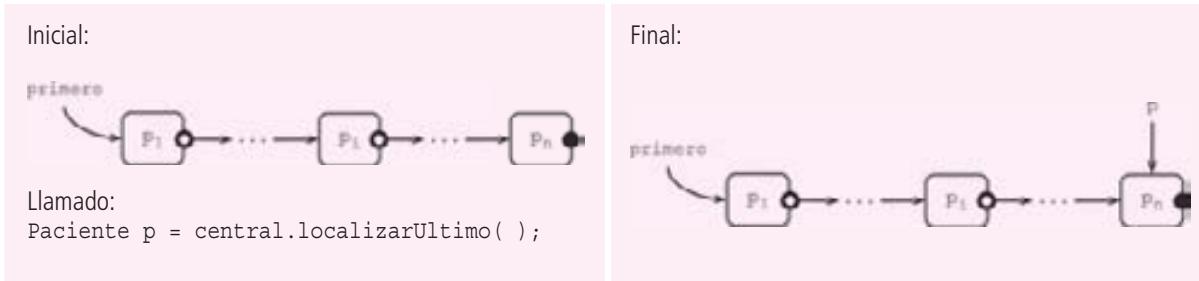
    if( actual != null )
    {
        while( actual.darSiguiente( )!=null )
        {
            actual = actual.darSiguiente( );
        }
    }

    return actual;
}
```

Utilizamos de nuevo una referencia auxiliar llamada "actual". Dicha referencia comienza señalando al primer paciente de la estructura.

Si la estructura se encuentra vacía (`primero == null`) el método retorna null.

En el ciclo avanza elemento por elemento mientras no haya llegado a la última posición.



Objetivo: Implementar el método de la clase `CentralPacientes` que permite localizar al paciente que se encuentra antes de otro paciente del cual recibimos el código como parámetro.

En este ejemplo presentamos el código del método e ilustramos gráficamente su funcionalidad.

```
public Paciente localizarAnterior( int cod )
{
    Paciente anterior = null;

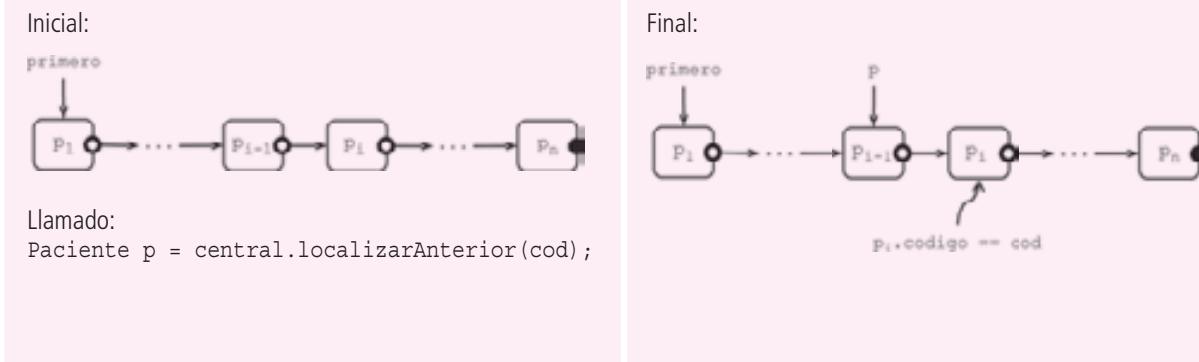
    Paciente actual = primero;

    while( actual != null &&
           actual.darCodigo( ) != cod )
    {
        anterior = actual;

        actual = actual.darSiguiente( );
    }

    return actual != null ? anterior : null;
}
```

- Para este método utilizamos dos referencias auxiliares: una (actual), para indicar el elemento sobre el cual estamos haciendo el recorrido, y otra (anterior), que va una posición atrasada con respecto a la primera.
- Cuando la referencia "actual" llegue al paciente buscado, en la referencia "anterior" tendremos la respuesta del método.
- Dentro del ciclo avanzamos las dos referencias: "anterior" se mueve a la posición que tiene "actual" y "actual" avanza una posición sobre la secuencia de pacientes.



**Ejemplo 6**

Objetivo: Implementar el método de la clase `CentralPacientes` que calcula el número de pacientes de la estructura. Dentro del caso de estudio lo usaremos para verificar el invariante.

En este ejemplo presentamos el código del método y explicamos la estructura del algoritmo de recorrido total de este tipo de estructuras.

```
private int darLongitud()
{
    Paciente actual = primero;

    int longitud = 0;

    while( actual != null )
    {
        longitud++;
        actual = actual.darSiguiente();
    }

    return longitud;
}
```

■ Vamos a utilizar una referencia auxiliar (`actual`) para hacer el recorrido de la estructura. El ciclo termina cuando dicha referencia toma el valor `null`.

■ Declaramos también una variable de tipo entero (`longitud`), en la cual iremos acumulando el número de elementos recorridos.

■ Al final del ciclo, en la variable “`longitud`” tendremos el número total de pacientes registrados en la central.

A continuación planteamos, como tarea al lector, la implementación de cuatro métodos de localización y reco-

rrido, que corresponden a variantes de lo presentado en los ejemplos anteriores:

**Tarea 2**

Objetivo: Implementar algunos métodos de la clase `CentralPacientes` para practicar la algorítmica de localización y recorrido.

Desarrolle los cuatro métodos que se plantean a continuación.

```
public class CentralPacientes
{
    private Paciente primero;

    public int darPacientesEnClinica( String nomClinica )
    {

    }
```

■ Calcula el número de pacientes que fueron enviados a una misma clínica, cuyo nombre es dado como parámetro.

```
public Paciente darPacienteMenorCodigo( )  
{
```

■ Retorna el paciente con el menor código en la central. Si no hay ninguno, retorna null.

```
}
```

```
public boolean hayMasHombres( )  
{
```

■ Indica si hay más hombres que mujeres registrados en la central.

```
}
```

```
public Paciente darUltimaMujer( )  
{
```

■ Retorna la última mujer en la estructura. Si no hay ninguna retorna null.

```
}
```

3.6.2. Supresión de Elementos

Para eliminar un elemento de una estructura enlazada, vamos a dividir las responsabilidades entre

las clases `CentralPacientes` y `Paciente`, tal como se muestra en los siguientes ejemplos:

**Ejemplo 7**

Objetivo: Presentar los métodos de la clase `Paciente` que nos van a servir para eliminar un elemento de la lista.

En este ejemplo explicamos los métodos `cambiarSiguiente()` y `desconectarSiguiente()` de la clase `Paciente`.

```
public class Paciente
{
    // -----
    // Atributos
    // -----

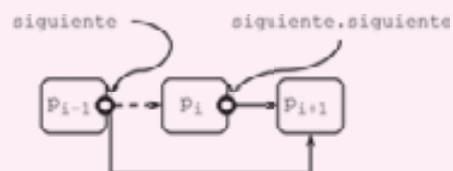
    private Paciente siguiente;

    public void cambiarSiguiente( Paciente pac )
    {
        siguiente = pac;
    }

    public void desconectarSiguiente( )
    {
        siguiente = siguiente.siguiente;
    }
}
```

El primer método cambia la referencia que tiene al siguiente elemento y ahora señala al paciente que llega como parámetro.

El segundo método tiene como precondition que no es el último paciente de la lista. Para desencadenar el siguiente, se contenta con apuntar al que está después del que le sigue, tal como se muestra en la figura:

**Ejemplo 8**

Objetivo: Mostrar el método de la clase `CentralPacientes` que elimina un paciente de la central, dado su código.

En este ejemplo presentamos el método que elimina un elemento de una lista enlazada.

```
public class CentralPacientes
{
    // -----
    // Atributos
    // -----

    private Paciente primero;
    private int numPacientes;

    public void eliminarPaciente( int cod )
            throws NoExisteException
    {
        if( primero == null )
            throw new NoExisteException( cod );
    }
}
```

El método considera tres casos: (1) la lista está vacía, (2) es el primer elemento de la lista o (3) está en un punto intermedio de la lista (incluyendo el final).

En el primer caso, lanzamos una excepción informando que el paciente no fue encontrado.

En el segundo caso, cambiamos la referencia al primero y lo ponemos a apuntar al segundo de la lista.

```

else if( cod == primero.darCodigo( ) )
{
    primero = primero.darSiguiente( );
}

else
{
    Paciente anterior = localizarAnterior( cod );
    if( anterior == null )
        throw new NoExisteException( cod );

    anterior.desconectarSiguiente( );
}

numPacientes--;
verificarInvariant( );
}

```

En el tercer caso, localizamos el paciente anterior al que queremos eliminar, utilizando el método localizarAnterior() desarrollado en el ejemplo 5. Una vez que ha sido localizado, procedemos a pedirle que desencadene al siguiente, utilizando para esto el método desconectarSiguiente() de la clase Paciente.

En el caso anterior, si el paciente con el código pedido no existe, lanzamos una excepción.

Finalmente, disminuimos en uno el número de pacientes de la lista (valor almacenado en el atributo "numPacientes") y verificamos que el objeto continúe cumpliendo con el invariante.



En el sitio web puede localizar la herramienta llamada "Laboratorio de Estructuras de Datos" y resolver algunos de los retos que allí se plantean, como una manera de complementar el trabajo desarrollado hasta este punto del nivel.

En la siguiente tarea trabajaremos en algunos métodos cuyo objetivo es eliminar elementos de una estructura enlazada:

Tarea 3



Objetivo: Implementar algunos métodos de la clase CentralPacientes para practicar la algorítmica de supresión de elementos.

Desarrolle los tres métodos que se plantean a continuación.

```

public class CentralPacientes
{
    private Paciente primero;

```

```
public void eliminarUltimo( )
{
}

public void eliminarHombres( )
{
}

public void eliminarMenorCodigo( )
{
}

}
```

 Elimina al último paciente de la lista.

 Elimina de la lista a todos los pacientes de sexo masculino.

 Elimina de la lista al paciente con el menor código.

3.6.3. Inserción de Elementos

Para insertar un elemento en una estructura enlazada, vamos a dividir las responsabilidades entre las clases `CentralPacientes` y `Paciente`, de la manera como se muestra en los siguientes ejemplos:

Ejemplo 9



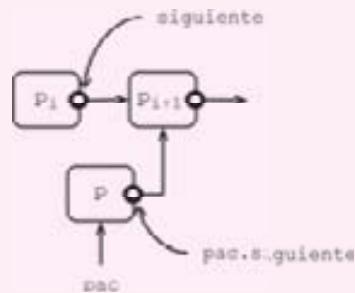
Objetivo: Presentar los métodos de la clase `Paciente` que nos van a servir para enlazar un nuevo elemento en una secuencia.

En este ejemplo explicamos el método `insertarDespues(paciente)` de la clase `Paciente`, el cual permite pedirle a un objeto de dicha clase que cambie su referencia al siguiente elemento.

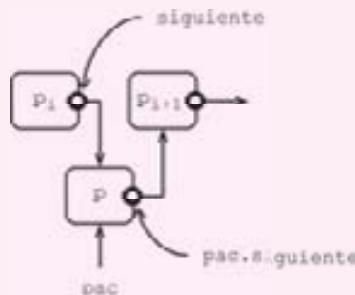
```
public class Paciente
{
    // -----
    // Atributos
    // -----
    private Paciente siguiente;
    public void insertarDespues( Paciente pac )
    {
        pac.siguiente = siguiente;
        siguiente = pac;
    }
}
```

Este método cambia la referencia al siguiente objeto de la estructura, de manera que ahora indique al paciente que llega como parámetro. Eso tiene como efecto que se inserta el nuevo paciente en la secuencia.

Con la primera instrucción hacemos que el nuevo paciente haga referencia al siguiente:



Con la segunda instrucción incluimos al nuevo paciente en la secuencia:



Ahora presentaremos los cuatro métodos de inserción que necesitamos para implementar los requerimientos funcionales del caso de estudio: un método que agregue un paciente como el primero de la lista (ejemplo 10), un método que agregue un paciente al final de la lista (ejemplo 11), un método que incluya

un nuevo paciente después de otro de la lista (ejemplo 12) y un método para insertar un nuevo paciente antes de otro de la lista (ejemplo 13). Con esos cuatro métodos completamos la algorítmica de inserción en estructuras enlazadas.

Ejemplo 10

Objetivo: Presentar el método de la clase `CentralPacientes` que inserta un nuevo paciente antes del primero de la lista.

La precondition del método garantiza que no hay ningún otro paciente en la central con el mismo código del paciente que se va a agregar.

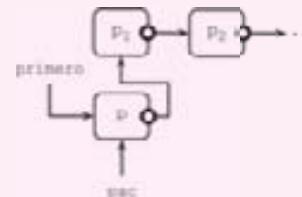
```
public class CentralPacientes
{
    // -----
    // Atributos
    // -----
    private Paciente primero;
    private int numPacientes;

    public void agregarPacienteAlComienzo( Paciente pac )
    {
        if( primero == null )
        {
            primero = pac;
        }
        else
        {
            pac.cambiarSiguiente( primero );
            primero = pac;
        }

        numPacientes++;
        verificarInvarianto();
    }
}
```

El método contempla dos casos: si la lista de pacientes se encuentra vacía (`primero == null`) basta con cambiar el valor de la referencia que señala al primero, de manera que ahora apunte al nuevo paciente.

Si la lista no está vacía, agregamos el nuevo elemento en la primera posición.



Finalmente actualizamos el atributo que contiene el número de elementos de la secuencia y verificamos que la estructura cumpla el invariante.

Ejemplo 11

Objetivo: Presentar el método de la clase `CentralPacientes` que inserta un nuevo paciente al final de la lista.

La precondition del método garantiza que no hay ningún otro paciente en la central con el mismo código del paciente que se va a agregar.

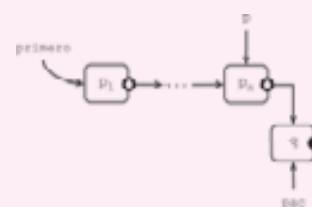
```
public class CentralPacientes
{
    // -----
    // Atributos
    // -----
    private Paciente primero;
    private int numPacientes;

    public void agregarPacienteAlFinal( Paciente pac )
    {
        if( primero == null )
        {
            primero = pac;
        }
        else
        {
            Paciente p = localizarUltimo();
            p.insertarDespues( pac );
        }

        numPacientes++;
        verificarInvarianto();
    }
}
```

Este método considera dos casos posibles: si la lista es vacía (`primero == null`), deja el nuevo nodo como el primero (y también el último, porque es el único paciente).

Si la lista no es vacía, utiliza el método que permite localizar el último elemento de la estructura (`localizarUltimo()`). Una vez que tenemos la referencia "p" señalando al último paciente, le pedimos que agregue al nuevo elemento como siguiente.



Ejemplo 12

Objetivo: Presentar el método de la clase `CentralPacientes` que inserta un nuevo paciente después de un paciente que se encuentra en la lista y del cual recibimos su código como parámetro. La precondition del método garantiza que no hay ningún otro paciente en la central con el mismo código del paciente que se va a agregar.

```
public class CentralPacientes
{
    private Paciente primero;
    private int numPacientes;

    public void agregarPacienteDespuesDe( int cod,
                                         Paciente pac )
                                         throws NoExisteException
    {
        Paciente anterior = localizar( cod );

        if( anterior == null )
            throw new NoExisteException( cod );
        else
            anterior.insertarDespues( pac );

        numPacientes++;
        verificarInvariant();
    }
}
```

- El método recibe como parámetro el código del paciente después del cual debemos hacer la inserción y el nuevo paciente de la central.
- Si no hay ningún paciente con dicho código, el método lanza una excepción.
- El primer paso es localizar al paciente con el código "cod" dentro de la lista y dejar sobre él la referencia "anterior". Si no lo logra localizar, lanza la excepción.
- Luego, le pide al paciente referenciado por la variable "anterior" que incluya al nuevo paciente dentro de la secuencia.
- Finalmente, incrementa el atributo que contiene el número de pacientes y verifica el cumplimiento del invariante.

Ejemplo 13

Objetivo: Presentar el método de la clase `CentralPacientes` que inserta un nuevo paciente antes de un paciente que se encuentra en la lista y del cual recibimos su código como parámetro. La precondition del método garantiza que no hay ningún otro paciente en la central con el mismo código del paciente que se va a agregar.

```
public class CentralPacientes
{
    private Paciente primero;
    private int numPacientes;

    public void agregarPacienteAntesDe( int cod,
                                         Paciente pac )
                                         throws NoExisteException
    {
        if( primero == null )
            throw new NoExisteException( cod );
```

- En este método debemos considerar tres casos distintos. En el primer caso, si la lista es vacía, lanzamos una excepción indicando que el paciente de código "cod" no existe.
- En el segundo caso, sabiendo que la lista no es vacía, establecemos si el paciente de código "cod" es el primero, caso en el cual el nuevo paciente debe ser insertado en la primera posición.

```

else if( cod == primero.darCodigo( ) )
{
    pac.cambiarSiguiente( primero );
    primero = pac;
}
else
{
    Paciente anterior = localizarAnterior( cod );
    if( anterior == null )
        throw new NoExisteException( cod );

    anterior.insertarDespues( pac );
}
numPacientes++;
verificarInvariantes( );
}
}

```

Para el caso general, utilizamos el método que nos permite localizar el anterior a un elemento (`localizarAnterior()`) y dejamos sobre dicho paciente la referencia "anterior". Luego, le pedimos a dicho elemento que incluya al nuevo paciente dentro de la secuencia.

En la siguiente tarea vamos a trabajar sobre algoritmos que modifican los enlazamientos de la estructura lineal de pacientes que manejamos en la central.

Tarea 4



Objetivo: Implementar algunos métodos de la clase `CentralPacientes` para practicar la algorítmica de modificación del enlazamiento en una estructura lineal.

Desarrolle los métodos que se plantean a continuación.

```

public class CentralPacientes
{
    private Paciente primero;

    public void invertir( )
    {

```

■ Invierte la lista de pacientes, dejando al primero de último, al segundo de penúltimo, etc.

```
}
```

```
public void pasarAlComienzoPaciente( int cod )
    throws NoExisteException
{
```

 Mueve al comienzo de la lista al paciente que tiene el código "cod". Si no lo encuentra lanza una excepción.

```
}
```

```
public void ordenar( )
{
```

 Ordena la lista de pacientes de la central ascendente por código. Utiliza el algoritmo de selección.

```
}
```

```

public void pasarAlFinalPacientes( String nomClinica )
{
}

```

- Mueve al final de la lista a todos los pacientes que fueron remitidos a una clínica, cuyo nombre llega como parámetro.

3.7. Patrones de Algoritmo

La mayoría de los algoritmos para manejar estructuras enlazadas corresponden a especializaciones de cuatro patrones distintos de recorrido y localización, los cuales se resumen a continuación. Las variantes corresponden a cálculos o a cambios en los enlaces que se hacen a medida que se avanza en el recorrido, o al terminar el mismo.

```

Nodo actual = primero;
while( actual != null )
{
    ...
    actual = actual.darSiguiente( );
}

```

- Patrón de recorrido total:

El recorrido total es el patrón de algoritmo más simple que hay sobre estas estructuras. Consiste en pasar una vez sobre cada uno de los elementos de la secuencia, ya sea para contar el número de ellos o para calcular alguna propiedad del grupo. El esqueleto del algoritmo es el siguiente:

- Suponemos que los elementos de la estructura pertenecen a una clase llamada Nodo.
- El esqueleto del patrón de recorrido total utiliza una referencia auxiliar (actual) que nos permite desplazarnos desde el primer elemento de la estructura (primero) hasta que se hayan recorrido todos los elementos enlazados (actual==null).
- En cada ciclo se utiliza el método que permite avanzar hacia el siguiente nodo enlazado.

- Patrón de recorrido parcial – Localización del último elemento:

Es muy común en la manipulación de estructuras enlazadas que estemos interesados en localizar el último elemento de la secuencia. Para esto utilizamos el

patrón de recorrido parcial, con terminación al llegar al final de la secuencia. Fíjese que con este esqueleto no hacemos un recorrido total, puesto que siempre se queda el último elemento sin recorrer. Sólo sirve para posicionar una referencia sobre el último elemento.

```

if( actual != null )
{
    Nodo actual = primero;

    while( actual.darSiguiente( ) != null )
    {
        actual = actual.darSiguiente( );
    }
    ...
}

```

- Suponemos de nuevo que los elementos de la estructura enlazada son objetos de la clase Nodo.
- El esqueleto debe considerar aparte el caso en el cual la lista se encuentra vacía.
- En el caso general, utilizamos una referencia auxiliar para hacer el recorrido. La salida del ciclo se hace cuando la referencia al siguiente sea nula (en ese momento la variable "actual" estará señalando el último elemento de la estructura).

- Patrón de recorrido parcial – Hasta que una condición se cumpla sobre un elemento:

Con este patrón de algoritmo podemos recorrer la

estructura hasta que detectemos que una condición se cumple sobre el elemento actual del recorrido. Con este esqueleto se resuelven problemas como el de localizar un valor en una estructura enlazada.

```

Nodo actual = primero;

while( actual != null && !condicion )
{
    actual = actual.darSiguiente( );
}
...

```

- Suponemos de nuevo que los elementos de la estructura enlazada son objetos de la clase Nodo.
- El esqueleto debe considerar aparte el caso en el cual la lista se encuentra vacía.
- En el caso general, utilizamos una referencia auxiliar para hacer el recorrido. La salida del ciclo se hace cuando la referencia al siguiente sea nula (en ese momento la variable "actual" estará señalando al último elemento de la estructura).

- Patrón de recorrido parcial – Hasta que una condición se cumpla sobre el siguiente elemento:

Es muy frecuente encontrar algoritmos que utilizan este patrón, puesto que para hacer ciertas modifica-

ciones sobre una estructura simplemente enlazada es común tener que obtener una referencia al anterior elemento de uno que cumpla una propiedad. Piense, por ejemplo, en el problema de agregar un elemento antes de alguno que tenga un valor dado.

```

Nodo anterior = null;
Nodo actual = primero;

while( actual != null && !condicion )
{
    anterior = actual;
    actual = actual.darSiguiente( );
}
...

```

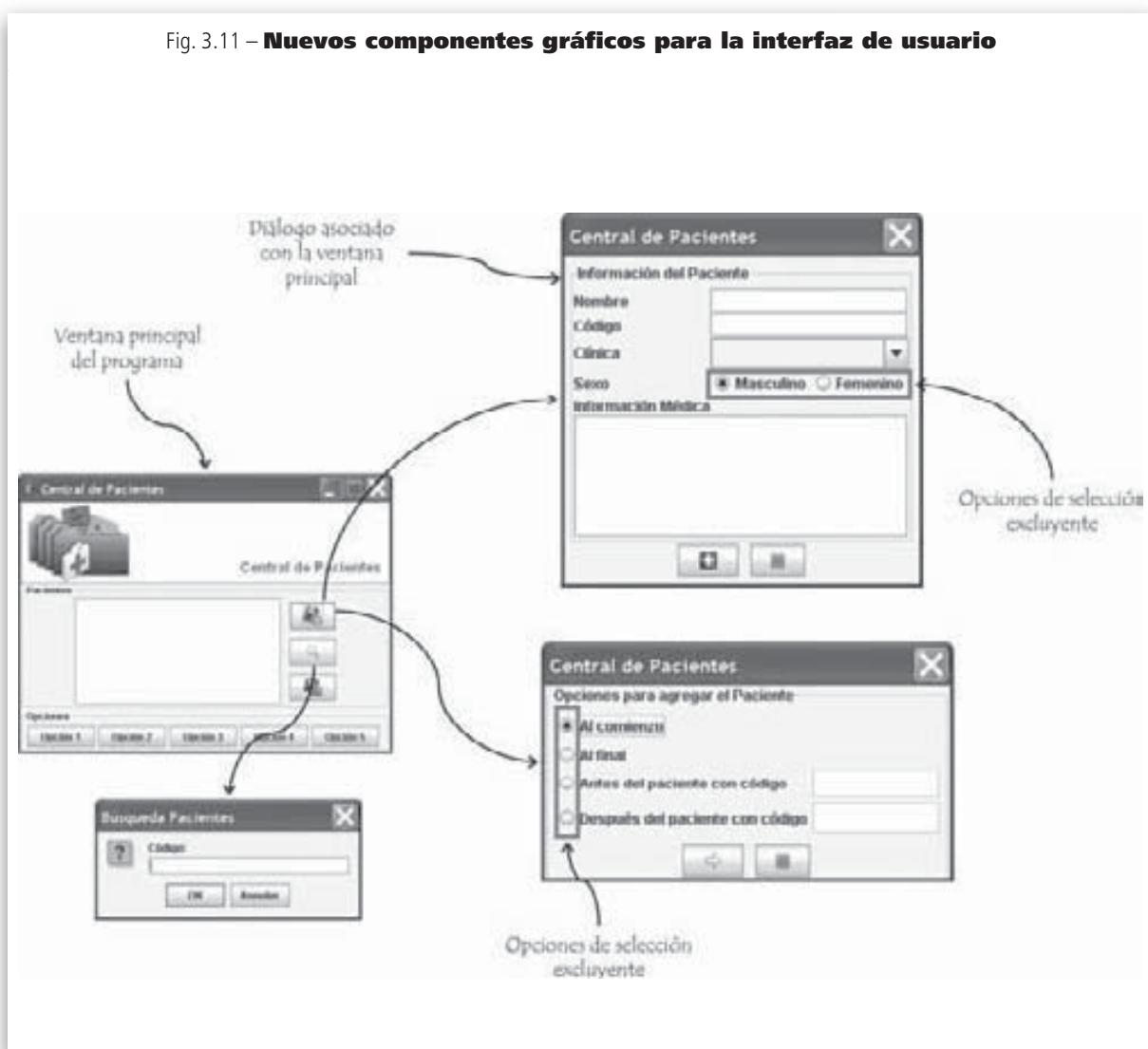
- Este esqueleto utiliza dos referencias: una que tiene como valor el elemento actual del recorrido (actual) y otra que va una posición atrasada con respecto a la primera (anterior).
- Al terminar el ciclo, en la variable "anterior" tenemos una referencia al elemento anterior al que cumple la propiedad.
- Al igual que en el esqueleto anterior, la propiedad se representa con la variable "condicion", pero podría tratarse de cualquier expresión de tipo lógico.

3.8. Nuevos Componentes Gráficos

En esta sección vamos a introducir dos nuevos componentes gráficos para enriquecer las interfaces de usuario. El primero nos permite manejar opciones

de selección exclusiva, como las que se necesitan en el caso de estudio para seleccionar el sexo del paciente. El segundo nos permite crear nuevas ventanas para manejar diálogos puntuales con el usuario. Ambos componentes se ilustran en la figura 3.11.

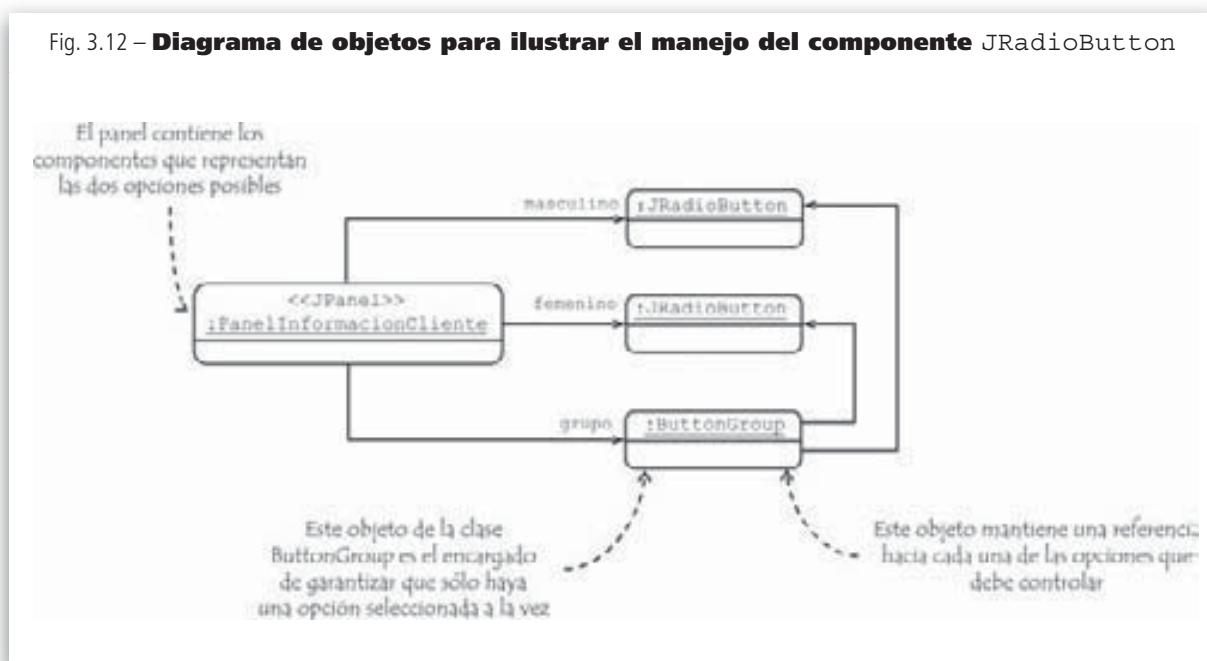
Fig. 3.11 – **Nuevos componentes gráficos para la interfaz de usuario**



Para mostrar en la interfaz de usuario un grupo excluyente de opciones, Java provee dos clases: `JRadioButton` para representar cada una de las opciones y `ButtonGroup` para manejar el grupo. En la figura 3.12 se muestra un diagrama de objetos para ilustrar las relaciones entre estos dos componentes.

Allí se ve cómo las opciones se agregan directamente a los paneles como cualquier otro componente, y que el hecho de que sean excluyentes entre ellas se maneja con un objeto de la clase `ButtonGroup`, el cual se encarga de controlar que sólo haya una de ellas seleccionada en todo momento.

Fig. 3.12 – **Diagrama de objetos para ilustrar el manejo del componente** JRadioButton



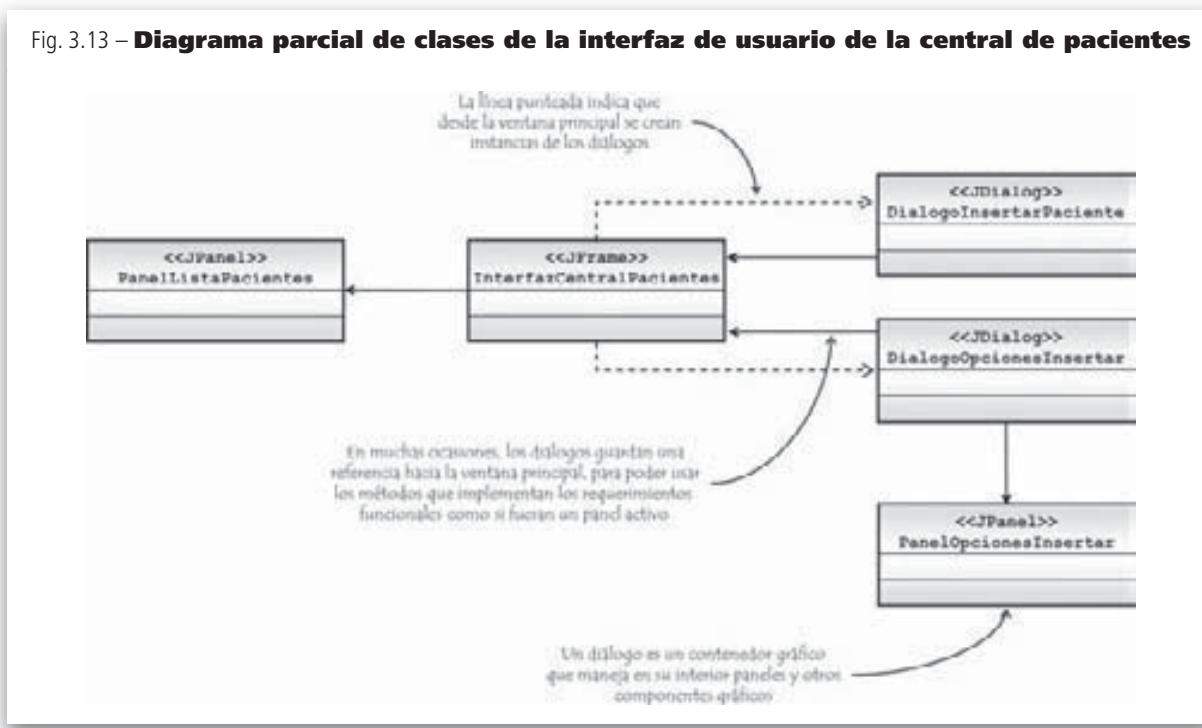
Los principales métodos que provee la clase JRadioButton son los siguientes:

- **JRadioButton(etiqueta)** : Éste es el constructor de la clase. Permite crear una nueva opción cuyo nombre será la etiqueta que recibe como parámetro.
- **JRadioButton(etiqueta, seleccionado)** : Es una variante del constructor anterior, en la cual la opción aparece como seleccionada.
- **setSelected(seleccionado)** : Este método se utiliza para seleccionar o no la opción, dependiendo del valor lógico que se pase como parámetro.
- **setEnabled(habilitado)** : Este método activa o desactiva la opción, dependiendo del valor lógico que se pase como parámetro. Al desactivar la opción se permite que el usuario visualice su valor, pero no se le permite cambiarlo.
- **isSelected()** : Retorna un valor lógico que indica si la opción está seleccionada.

Los dos métodos de la clase ButtonGroup en los cuales estamos interesados son:

- **ButtonGroup()** : Este constructor permite crear un agrupamiento de opciones, inicialmente vacío.
- **add(opcion)** : Con este método se agregan opciones al grupo, el cual garantiza siempre que sólo una de ellas se encuentra seleccionada.

Un diálogo es una ventana auxiliar de la interfaz de usuario, asociada con la ventana principal, en la cual podemos establecer una comunicación puntual con el usuario. Dichas ventanas deben aparecer y desaparecer según las opciones que se vayan seleccionando durante la ejecución del programa. Puesto que un diálogo es un contenedor gráfico, tiene la misma estructura de paneles de la ventana principal y responde a los mismos métodos. En la figura 3.13 aparece una parte del diagrama de clases de la interfaz de usuario para la central de pacientes.

Fig. 3.13 – **Diagrama parcial de clases de la interfaz de usuario de la central de pacientes**

Un diálogo puede ser **modal** o **no modal**, lo que significa que bloquea o no la ventana principal en el momento de aparecer. Bloquear la ventana principal significa que el usuario no podrá interactuar con ninguno de sus elementos. En nuestro caso de estudio todos los diálogos son modales.

En Java los diálogos son implementados por la clase `JDialog`. En este nivel nos vamos a concentrar en algunos métodos de esta clase que sirven para hacer aparecer y desaparecer la ventana, puesto que para el resto de cosas este componente tiene el mismo comportamiento de una ventana normal. Estos métodos son:

- `JDialog(propietaria, esModal)` : Con este constructor podemos crear un diálogo y asociarlo con una ventana propietaria que se recibe como parámetro. Se puede también escoger si el diálogo es o no modal (el segundo parámetro es de tipo lógico). La manera de utilizar este constructor aparece ilustrado en el ejemplo 14.
- `setVisible(mostrar)` : Este método hace aparecer el diálogo en la pantalla cuando se llama con el valor true como parámetro.
- `dispose()` : Vamos a utilizar este método para cerrar el diálogo y pedirle a la máquina virtual de Java que libere todos los recursos que utilizaba.
- `pack()` : Con este método hacemos que el diálogo (y en general cualquier ventana) aplique los distribuidores gráficos y las dimensiones pedidas por el programador, tanto para la ventana como para los componentes gráficos que contiene.

en la pantalla del computador localizada de acuerdo con un componente que se recibe como parámetro. Este componente es típicamente la ventana propietaria del diálogo. Si no se invoca este método, el diálogo va a aparecer en el extremo superior izquierdo de la pantalla.

- `setVisible(mostrar)` : Este método hace aparecer el diálogo en la pantalla cuando se llama con el valor true como parámetro.
- `dispose()` : Vamos a utilizar este método para cerrar el diálogo y pedirle a la máquina virtual de Java que libere todos los recursos que utilizaba.
- `pack()` : Con este método hacemos que el diálogo (y en general cualquier ventana) aplique los distribuidores gráficos y las dimensiones pedidas por el programador, tanto para la ventana como para los componentes gráficos que contiene.

Para crear un diálogo debemos seguir los mismos pasos que se utilizan para crear una ventana, tal como se ilustra en el ejemplo 14.

Ejemplo 14

Objetivo: Mostrar la manera de crear un diálogo desde la ventana principal de un programa.

En este ejemplo mostramos algunos fragmentos simplificados de código, el cual nos permite ilustrar los principales aspectos que se deben tener en cuenta al momento de crear un diálogo.

```
public class InterfazCentralPacientes extends JFrame
{
    private CentralPacientes central;
    ...

    public InterfazCentralPacientes( )
    {
        central = new CentralPacientes( );
        ...
    }

    public void mostrarDialogoInsertarPaciente( )
    {
        DialogoInsertarPaciente dialogo = new DialogoInsertarPaciente(this);
        dialogo.setLocationRelativeTo( this );
        dialogo.setVisible( true );
    }
}

public class DialogoInsertarPaciente extends JDialog
    implements ActionListener
{
    // -----
    // Constantes
    // -----
    private final static String AGREGAR = "Agregar";
    private final static String CANCELAR = "Cancelar";

    // -----
    // Atributos
    // -----
    private InterfazCentralPacientes principal;

    public DialogoInsertarPaciente( InterfazCentralPacientes ventana )
    {
        super( ventana, true );
        principal = ventana;
        ...
        pack();
    }

    public void actionPerformed( ActionEvent e )
    {
        String comando = e.getActionCommand();

        if( comando.equals( AGREGAR ) )
        {
            ...
        }
        else if( comando.equals( CANCELAR ) )
        {
            dispose();
        }
    }
}
```

■ La ventana principal se crea de la misma manera que siempre lo hemos hecho.

■ La diferencia radica en que, al implementar ciertos requerimientos funcionales, creamos y desplegamos un diálogo con el usuario.

■ La clase debe extender de JDialog e implementar la interfaz ActionListener, puesto que va a tener botones.

■ En el constructor, invocamos el constructor de la clase JDialog con la palabra "super", que siempre hace referencia a la clase de la cual se hace la extensión.

■ Si el usuario cancela la operación, sencillamente invocamos el método dispose() que destruye el diálogo.

■ En la opción "agregar" invocamos el respectivo método de la ventana principal.

4. Caso de Estudio N° 2: Reservas en una Aerolínea

Se quiere construir un programa para administrar los vuelos de una aerolínea, que parten de Bogotá hacia las demás ciudades del mundo. El sistema debe ser capaz de manejar cualquier cantidad de ciudades, y cualquier número de vuelos hacia ellas. La aerolínea sólo maneja un tipo de avión, el cual tiene 15 filas (de la 0 a la 14) con cinco sillas cada una. Las sillas se encuentran identificadas por el número de la fila en

la que se encuentran y por una letra de la 'A' a la 'E' (por ejemplo, la silla "7-C"). Para administrar la información de las ciudades y vuelos existen tres requerimientos funcionales: (1) Agregar una nueva ciudad de destino, para lo cual se debe suministrar un nombre y unas coordenadas sobre el mapa. (2) Eliminar una ciudad, la cual debe ser señalada por el usuario en el mapa. (3) Agregar un vuelo desde Bogotá hacia una de las ciudades de destino ya definidas. Para esto el usuario debe señalar la ciudad en el mapa y dar un código para el vuelo, que debe ser único, lo mismo que una fecha y una hora. En la figura 3.14 se presenta la interfaz de usuario que debe tener el programa.

Fig. 3.14 – **Interfaz de usuario del programa de reservas de una aerolínea**



■ En la parte superior izquierda de la ventana aparece un mapa con todas las ciudades registradas en el programa. Cada punto amarillo representa una ciudad.

■ Para seleccionar una ciudad basta con hacer clic sobre ella en el mapa.

■ Una vez seleccionada la ciudad, aparecen, en la zona superior central, la lista de todos los vuelos disponibles hacia ella.

■ Al seleccionar uno de los vuelos, el programa muestra en la parte central de la ventana su código, la fecha del vuelo y el porcentaje de puestos libres que tiene. También presenta el plano del avión, indicando las sillas que se encuentran ocupadas y libres.

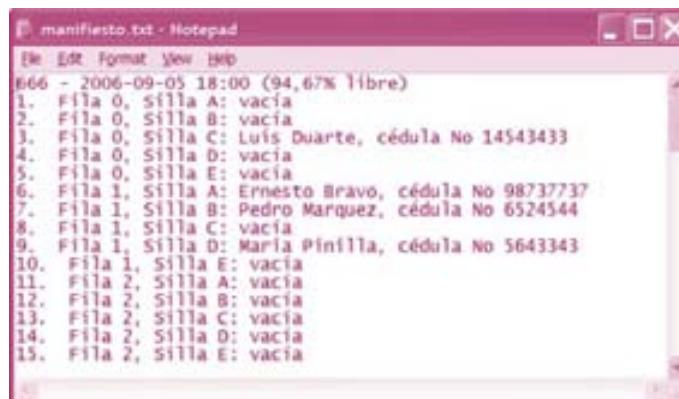
■ Para ingresar un nuevo vuelo, el programa muestra el diálogo que aparece en la figura. Allí se debe dar un código único, una fecha y una hora.

Desde el punto de vista del usuario que va a hacer una reserva, hay tres opciones: (4) Listar todos los vuelos existentes a una ciudad de destino dada, ordenados por fecha y hora. (5) Presentar la información de un vuelo particular. En ese caso se deben mostrar todas las sillas ocupadas y disponibles del vuelo, además del porcentaje libre del mismo. (6) Hacer una reserva en un vuelo. En esta opción el usuario debe dar la ciudad de destino, el vuelo, la identificación de la silla que quiere reservar, el nombre del pasajero y su número de identificación.

Finalmente, desde el punto de vista del encargado de seguridad de la aerolínea, hay una opción que le permite generar un archivo con el manifiesto de embarque, en el cual aparece la información de cada uno de los pasajeros que va a viajar. En la figura 3.15 aparece un ejemplo del formato que debe tener dicho archivo. El usuario debe escoger un directorio y un nombre para almacenar el archivo generado.

El programa debe manejar de manera persistente la información que tiene.

Fig. 3.15 – **Formato del archivo con el manifiesto de embarque**



4.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none"> ■ Manejar estructuras secuenciales sencillamente enlazadas, para almacenar las ciudades a las cuales llegan vuelos de la aerolínea. 	<ul style="list-style-type: none"> ■ Reforzar la algorítmica estudiada en el caso anterior, para agregar y eliminar ciudades. Repasar los patrones de algoritmo para recorrer y localizar elementos sobre estas estructuras.
<ul style="list-style-type: none"> ■ Manejar estructuras secuenciales doblemente enlazadas, para almacenar los vuelos que llegan a una ciudad dada, ordenados ascendentemente por fecha y por hora. 	<ul style="list-style-type: none"> ■ Estudiar las estructuras doblemente enlazadas y la algorítmica necesaria para manejarlas. ■ Estudiar el manejo de estructuras enlazadas ordenadas.
<ul style="list-style-type: none"> ■ Hacer persistir la información de las ciudades, los vuelos y las reservas. 	<ul style="list-style-type: none"> ■ Utilizar la técnica de persistencia por serialización estudiada en el nivel anterior.

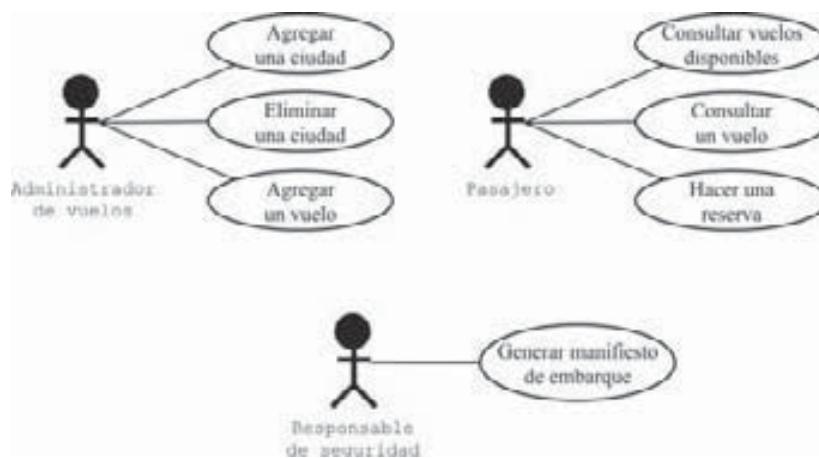
■ Generar un archivo secuencial de texto con información del manifiesto de embarque.	■ Utilizar lo estudiado en el nivel anterior para manejar archivos secuenciales de texto y exportar en ellos información.
■ Distribuir de manera más amigable la información en la interfaz de usuario.	■ Estudiar nuevos distribuidores gráficos, de manera que sea posible mejorar la apariencia y distribución de la información en la interfaz de usuario.
■ Separar los requerimientos funcionales del programa por tipo de usuario.	■ Estudiar los diagramas de casos de uso de UML, que permiten asociar actores con funcionalidades.

4.2. Diagramas de Casos de Uso

En el enunciado es claro que hay tres usuarios distintos que van a utilizar el programa que vamos a construir: el administrador de vuelos, el pasajero y el responsable de seguridad de la aerolínea. Lo ideal en esos casos es construir tres interfaces de usuario distintas, cada una de ellas orientada a un tipo de usuario. En nuestro caso, por simplicidad, vamos a construir una sola interfaz que va a ser compartida por los distintos interesados.

Una primera etapa en la definición de los requerimientos funcionales consiste en identificar a los usuarios potenciales del programa y asociarle a cada uno de ellos las funcionalidades que va a utilizar. Para esto contamos con un diagrama muy simple, que hace parte de UML, y que se denomina el **diagrama de casos de uso**. En la figura 3.16 se muestra este diagrama para el caso de estudio.

Fig. 3.16 – **Diagrama de casos de uso para el programa de la aerolínea**



El diagrama de casos de uso tiene dos elementos principales: el **actor**, que representa un tipo de usuario que va a tener el programa y el **caso de uso** (*use case*), que representa una funcionalidad del programa que va a ser utilizada por un actor. Este diagra-

ma se puede considerar como una especie de "mapa funcional" del programa, y es muy usual que la etapa de análisis de requerimientos funcionales comience por su construcción y validación con el cliente.



El nombre de cada actor debe ser un sustantivo que identifique de manera clara un tipo de usuario. El nombre que se asocia con cada caso de uso debe comenzar por un verbo, que indique la funcionalidad que el programa le debe ofrecer a un actor.

4.3. Comprensión de los Requerimientos

Tarea 5



Objetivo: Entender el problema del caso de estudio.

(1) Lea detenidamente el enunciado del caso de estudio e (2) identifique y complete la documentación de los siete requerimientos funcionales que allí aparecen, haciendo explícito el nombre del actor que va a utilizar cada opción del programa.

Requerimiento funcional 1	Nombre	R1 – Agregar una ciudad	Actor	Administrador de vuelos
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 2	Nombre	R2 – Eliminar una ciudad	Actor	Administrador de vuelos
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 3	Nombre	R3 – Agregar un vuelo	Actor	Administrador de vuelos
	Resumen			
	Entrada			
	Resultado			

Requerimiento funcional 4	Nombre	R4 – Consultar vuelos disponibles	Actor	
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 5	Nombre	R5 – Consultar un vuelo	Actor	
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 6	Nombre	R6 – Hacer una reserva	Actor	
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 7	Nombre	R7 – Generar manifiesto de embarque	Actor	
	Resumen			
	Entrada			
	Resultado			

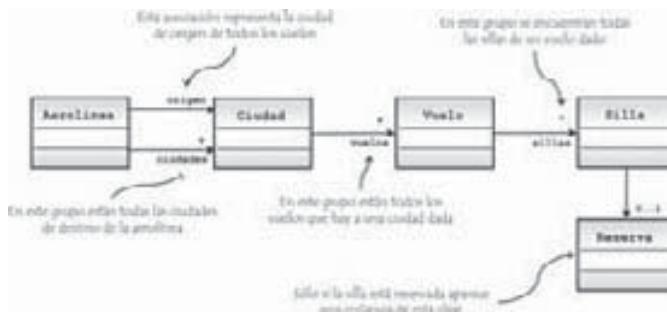
El enunciado del problema define un requerimiento no funcional de persistencia, el cual se resume en el siguiente formato:

Requerimiento no funcional 1	Tipo: Persistencia
	Descripción: <ul style="list-style-type: none">• La información del modelo del mundo debe ser persistente.

4.4. Del Análisis al Diseño

El diagrama de clases obtenido en el análisis del modelo del mundo se muestra en la figura 3.17. Allí encontramos cinco entidades: la aerolínea, la ciudad, el vuelo, la silla y la reserva. Los atributos y los invariantes se piden al lector como parte de la tarea 6.

Fig. 3.17 – **Diagrama de clases del análisis para el caso de la aerolínea**



Cuando comenzemos la etapa de diseño debemos tomar la decisión de cómo representar cada uno de los tres niveles de agrupamiento que maneja el pro-

blema: una aerolínea tiene un grupo de ciudades, una ciudad tiene un grupo de vuelos y cada vuelo tiene un grupo de sillas.

Tarea 6



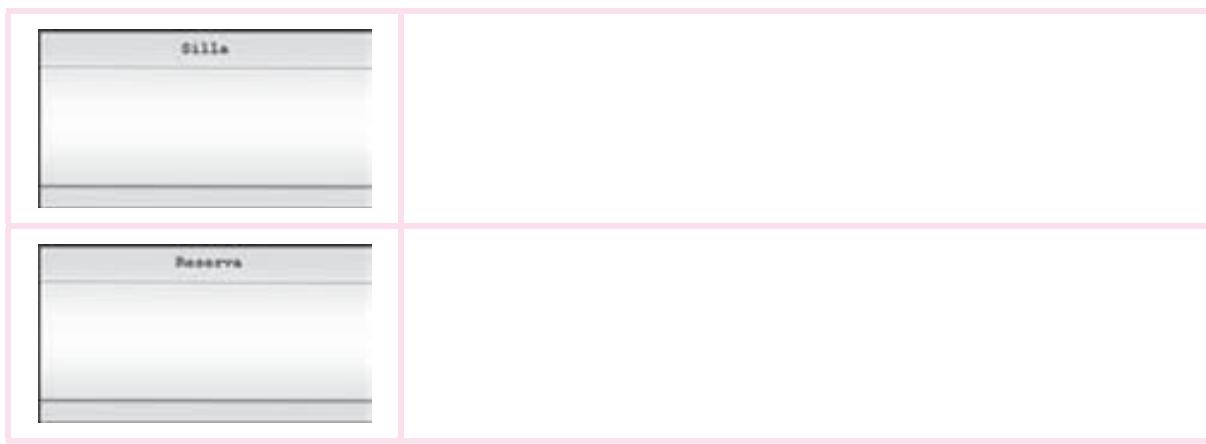
Objetivo: Identificar los atributos y los invariantes de las cinco entidades del diagrama de clases del análisis.

Para cada una de las clases identifique sus atributos (nombre y tipo) y especifique su invariante de la manera más precisa posible.

AEROLÍNEA	
-----------	--

CIUDAD	
--------	--

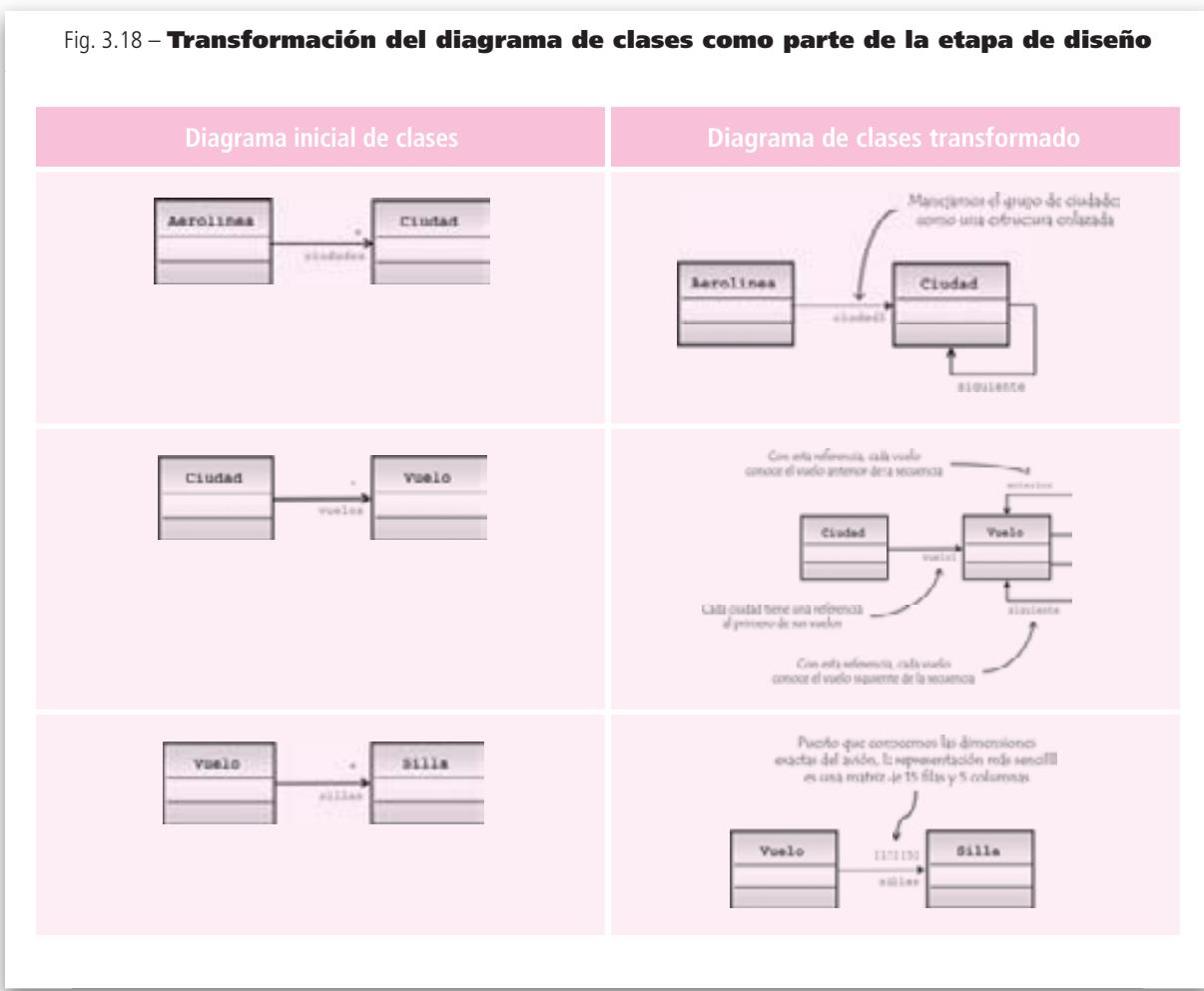
VUELO	
-------	--



Como parte del proceso de transformación del diagrama de clases debemos decidir la manera de representar los tres agrupamientos de objetos antes mencionados, los cuales tienen características distintas.

Estas características nos van a servir para tomar las decisiones correspondientes. En la figura 3.18 se resumen las decisiones de diseño tomadas sobre las cuales haremos la implementación.

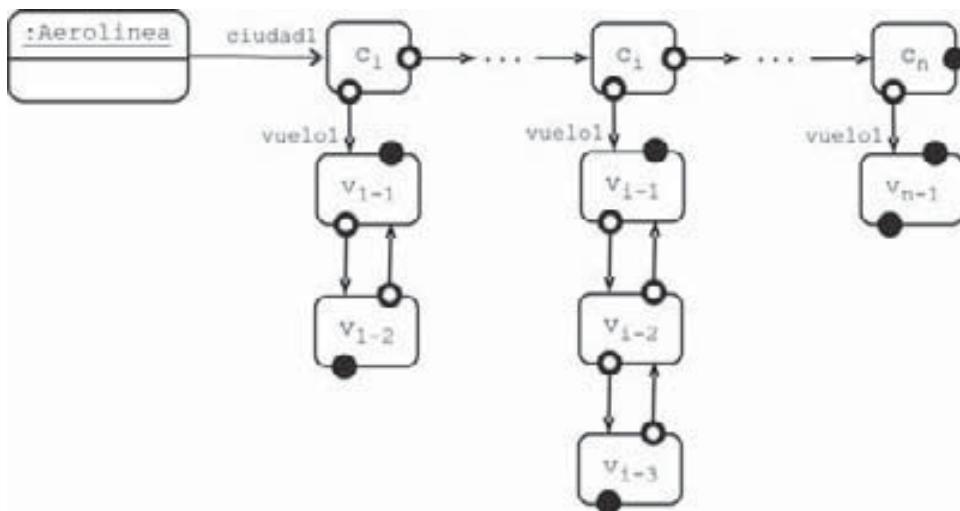
Fig. 3.18 – **Transformación del diagrama de clases como parte de la etapa de diseño**



Un posible diagrama de objetos para ilustrar las decisiones de diseño tomadas se muestra en la figura 3.19 (usando la sintaxis simplificada introducida al comienzo de este nivel). Allí podemos ver que vamos a manejar una estructura enlazada de ciudades

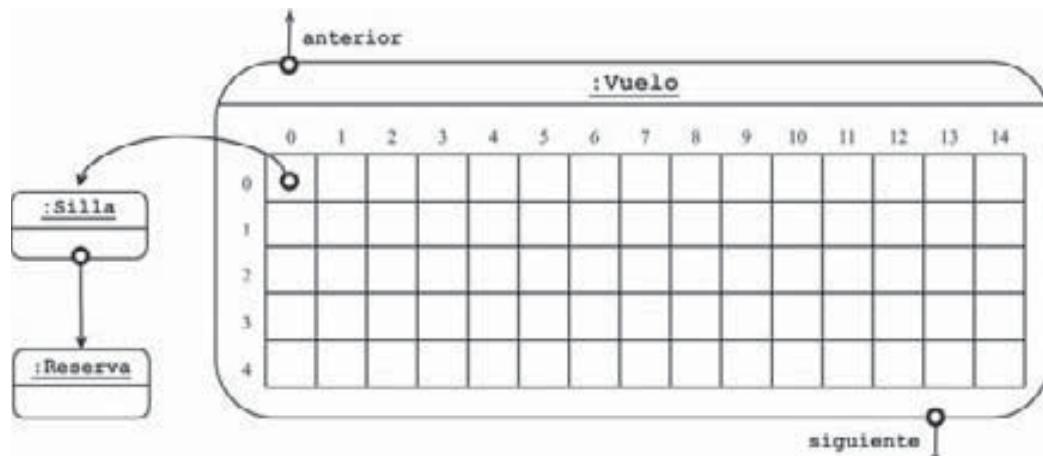
(cada una conoce a la siguiente), y que cada ciudad tiene una referencia al comienzo de una estructura doblemente enlazada de vuelos (cada vuelo conoce al siguiente y al anterior). Los vuelos se encuentran ordenados ascendente por fecha y por hora.

Fig. 3.19 – **Diagrama de objetos para el diseño propuesto para el caso de estudio**



Si miramos en detalle los objetos que representan un vuelo, obtenemos lo que se muestra en la figura 3.20.

Fig. 3.20 – **Detalle del diagrama de objetos para estudiar la representación de un vuelo.**



Vamos a agregar al diseño una clase llamada `AerolineaExpcion`, que va a representar los errores que se pueden presentar en los métodos de las clases que hacen parte del modelo del mundo.


Tarea 7

Objetivo: Declarar las clases para el programa que maneja la aerolínea.

Escriba la declaración en Java de cada una de las seis clases que van a hacer parte del programa del caso de estudio. Tenga en cuenta las decisiones de diseño explicadas en la figura 3.18 y el análisis hecho en la tarea 6.

```
public class Aerolinea
{
```

```
}
```

```
public class Ciudad
{
```

```
}
```

```
public class Silla
{
```

```
}
```

```
public class AerolineaExpcion
{
```

```
}
```

```
public class Vuelo
{
```

```
}
```

```
public class Reserva
{
```

```
}
```



En este punto hemos terminado ya el diseño del programa y estamos listos para comenzar la implementación de los métodos. La buena noticia es que con los patrones de algoritmo estudiados en el caso anterior podemos construir todos los métodos de este nuevo problema.

Lo único adicional es que debemos tener cuidado con el manejo de la referencia adicional hacia el elemento anterior, cuando se trata de una estructura doblemente enlazada.

4.5. Arreglos de Constantes

En Java es posible definir un arreglo de constantes, para los casos en los cuales sea mejor utilizarlas indicando

una posición en una contenedora y no un nombre particular. Esto es muy útil cuando queremos usar las constantes como parte de una instrucción repetitiva, como es el caso que se muestra en el ejemplo 15.

Ejemplo 15



Objetivo: Mostrar el método de la clase `Vuelo` que inicializa las sillas del avión, utilizando un arreglo de constantes.

En este ejemplo mostramos la manera de implementar el método de inicialización de las sillas de un vuelo, de manera que sea fácil adaptarlo en caso de una evolución del problema (que haya aviones con más filas o con más sillas por fila).

```
public class Vuelo
{
    // -----
    // Constantes
    // -----

    public static final char[] LETRAS = new char[]{ 'A', 'B', 'C', 'D', 'E' };

    public static final int NUMERO_FILAS = 15;

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

    private Silla[][] sillas;

    // -----
    // Métodos
    // -----


    private void inicializarSillas( )
    {
        sillas = new Silla[ NUMERO_FILAS ][ LETRAS.length ];

        for( int i = 0; i < NUMERO_FILAS; i++ )
        {
            for( int j = 0; j < LETRAS.length; j++ )
            {
                sillas[ i ][ j ] = new Silla( i, LETRAS[ j ] );
            }
        }
    }
}
```

Definimos un arreglo con las letras asignadas a las sillas de un vuelo. Es un arreglo de constantes que se debe inicializar en la misma declaración. Al fijar la lista de valores constantes, se define de manera implícita el número de casillas del arreglo.

Dentro del ciclo de inicialización, utilizamos la sintaxis `LETRA[j]` para referirnos a la constante que se encuentra en la casilla número "j".

Si tratamos de asignar un valor a una casilla del arreglo, vamos a obtener un error de compilación.

Podemos utilizar el operador `length` para establecer el número de constantes que hay en el arreglo.



En la declaración de un arreglo de constantes se debe definir el valor de cada una de ellas. Para esto se debe dar la lista de valores separados por una coma. Ni los valores ni el tamaño del arreglo se pueden modificar.

4.6. Manejo de Fechas y Formatos

En esta sección vamos a estudiar el manejo de fechas que necesitamos para el problema de la aerolínea. Para esto vamos a presentar algunos de los métodos de las clases `Date` y `SimpleDateFormat`, que nos van a permitir almacenar una fecha (incluido año, mes, día, hora, minutos, segundos y milisegundos) y convertirla en una cadena de caracteres que la represente, siguiendo un formato definido.

La clase `Date` se encuentra en el paquete `java.util` y cuenta con los siguientes métodos:

- `Date()` : Crea un objeto de la clase, que representa el instante de tiempo en el cual fue creado.
- `compareTo(fecha)` : Este método permite comparar dos fechas, retornando un valor negativo si la fecha que recibe el llamado es menor que el parámetro, cero si son iguales y un valor

positivo si la fecha que recibe el llamado es mayor que el parámetro.

- `equals(fecha)` : Este método indica si dos fechas son iguales, verificando no sólo el año, el mes y el día, sino también las horas, los minutos, los segundos y los milisegundos.

Esta clase contaba con muchos otros métodos en las primeras versiones de Java, pero poco a poco sus funcionalidades han sido remplazadas por otras clases, razón por la cual ahora esos métodos están marcados como obsoletos (*deprecated*) y no se deben utilizar.

La clase `SimpleDateFormat` está en el paquete `java.text` y tiene la siguiente funcionalidad:

- `SimpleDateFormat(patrón)` : Este método permite crear un objeto de la clase, especificando el patrón de formato con el que debe manipular las fechas. En la siguiente tabla hay algunos ejemplos de patrones que pueden ser utilizados para dar formato a una fecha.

Patrón	Fecha con el patrón	Explicación
"yyyy-MM-dd HH:mm"	"2006-11-28 08:10"	<ul style="list-style-type: none"> ■ "yyyy" representa el año (4 dígitos). ■ "MM" representa el mes (2 dígitos). ■ "dd" representa el día (2 dígitos). ■ "HH" representa la hora del día (0-23). ■ "mm" representa los minutos (2 dígitos).
"dd.MMM.yy 'at' HH:mm:ss"	"28.nov.06 at 08:18:40"	<ul style="list-style-type: none"> ■ "MMM" representa el mes, con tres letras. ■ "yy" representa el año (sólo 2 dígitos). ■ 'cadena' sirve para incluir cualquier texto como parte de la fecha. ■ "ss" representa los segundos (2 dígitos).
"hh:mm a"	"08:10 AM"	<ul style="list-style-type: none"> ■ "hh" representa la hora del día en formato 1 a 12. ■ "a" indica si la hora es "AM" o "PM".

- **parse(cadena)** : Este método recibe como parámetro una cadena de caracteres que representa una fecha en el formato definido y retorna un objeto de la clase Date. Lanza la excepción ParseException en el caso en el cual la cadena recibida como parámetro no cumpla con el patrón de formato establecido en el constructor.
- **format(fecha)** : Este método recibe como parámetro un objeto de la clase Date y retorna una cadena de caracteres que representa dicha fecha en el formato establecido en el constructor.

En el siguiente fragmento de código se muestra la manera de utilizar estas dos clases para construir un método que permite cambiar de formato una fecha:

```
public String formatoFecha( String cadena ) throws ParseException
{
    SimpleDateFormat formato1 = new SimpleDateFormat( "dd.MMM.yy 'at' HH:mm:ss" );
    Date fecha = formato1.parse( cadena );

    SimpleDateFormat formato2 = new SimpleDateFormat( "yyyy-MM-dd HH:mm" );
    return formato2.format( fecha );
}
```

4.7. Estructuras Ordenadas Dblemente Enlazadas

La mayor parte de esta sección está planteada en términos de tareas, a través de las cuales iremos construyendo todos los métodos necesarios para el caso de estudio y algunas extensiones propuestas.

Tarea 8



Objetivo: Implementar los métodos de la clase Vuelo, que permiten manipular las fechas y el doble enlace entre los elementos.

Desarrolle los métodos que se plantean a continuación.

```
public Date crearFecha( int anio, int mes, int dia, int horas, int minutos )
{
}
```



Crea un objeto de la clase Date, para representar la hora de salida de un vuelo. Recibe como parámetro la información necesaria en forma de valores numéricos.

```

public Silla darSilla( String nombreSilla )
{
}

public Date darFecha( )
{
}

public String darFechaYHora( )
{
}

public void reservarSilla( String nSilla, String nombre, int cedula )
throws AerolineaExpcion
{
}
}

```

Este método retorna la silla del vuelo que tiene el nombre que se pasa como parámetro (por ejemplo "7-C").

Retorna la fecha y hora de salida del vuelo, como un objeto de la clase Date.

Retorna la fecha y hora de salida del vuelo, usando el formato "yyyy-MM-dd HH:mm".

Este método asigna una silla al pasajero cuya información se recibe como parámetro.

Si un pasajero con esa misma cédula ya tiene una reserva en el mismo vuelo, el método lanza una excepción.

Si la silla está ocupada, el método lanza una excepción.

Utilice los métodos de las clases Silla y Reserva que considere necesarios.

```
public void generarManifiestoEmbarque( File archivo ) throws IOException
{
```

■ Genera el manifiesto de embarque del vuelo en el archivo indicado como parámetro.

■ Si hay un error durante el proceso de escritura, el método lanza la excepción IOException.

■ Revise en el enunciado el formato que debe tener este archivo.

■ Utilice los métodos de las clases Silla y Reserva que considere necesarios.

```
}
```

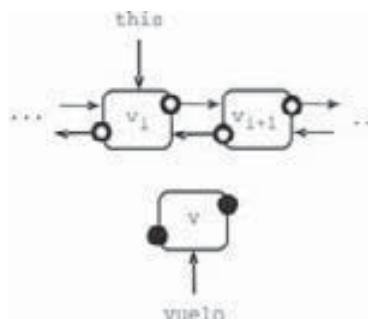
```
public void insertarDespues( Vuelo vuelo )
{
```

■ Inserta el vuelo que llega como parámetro después del actual.

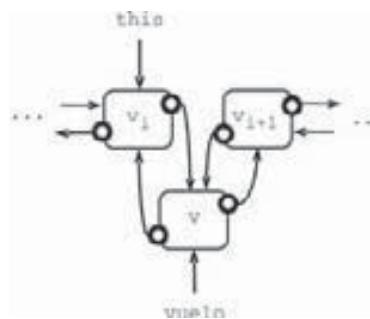
■ No olvide tener en cuenta el doble enlace de la estructura y el caso en el que sea el primero o el último.

```
}
```

Situación inicial:



Situación final:



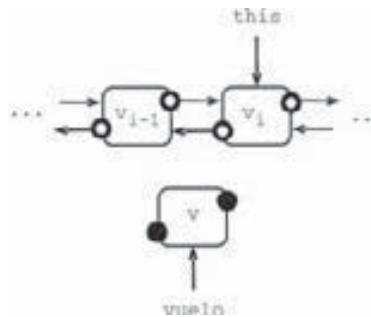
```
public void insertarAntes( Vuelo vuelo )
{
```

■ Inserta el vuelo que llega como parámetro antes del actual.

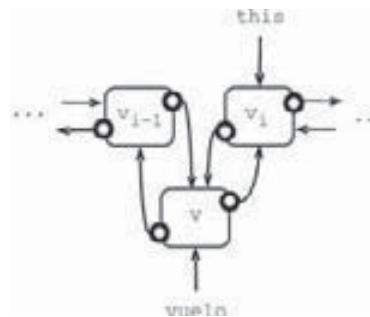
■ No olvide tener en cuenta el doble enlace de la estructura y el caso en el que sea el primero o el último.

```
}
```

Situación inicial:



Situación final:



```
private void verificarInvariante( )
{
}
```

- Verifica que en el vuelo no haya dos reservas con el mismo número de identidad.

Tarea 9

Objetivo: Implementar los métodos de la clase `Ciudad`, que permiten manejar los vuelos que llegan a ella como una estructura ordenada doblemente enlazada, lo mismo que proveer la funcionalidad básica para manejar las ciudades como una estructura con un enlace simple entre ellas.

Desarrolle los métodos que se plantean a continuación.

```
public Vuelo buscarVuelo( int codigo )
{
}
```

- Este método localiza y retorna un vuelo, dado su código. Si no lo encuentra, retorna null.

```
public void agregarVuelo( int codigo, Date fecha )
{
```

Este método agrega un nuevo vuelo a la ciudad, teniendo en cuenta que la lista de vuelos está ordenada ascendente por código.

La precondition del método afirma que no existe otro vuelo con ese mismo código.

Utilice los métodos desarrollados en la tarea anterior.

```
}
```

```
public void agregarDespues( Ciudad ciudad )
{
```

Cambia el enlace hacia la siguiente ciudad de la lista, para incluir la ciudad que llega como parámetro.

```
}
```

```
public void desconectarSiguiente( )
{
```

Cambia el enlace hacia la siguiente ciudad de la lista, para eliminarla de la estructura.

```
}
```

```
public void verificarInvariantes( )
{
```

Verifica que los vuelos de la ciudad se encuentren ordenados de manera ascendente, sin códigos repetidos.

```
}
```

Tarea 10

Objetivo: Implementar los métodos de la clase `Aerolinea` que proveen la funcionalidad necesaria para implementar los requerimientos funcionales del problema.

Desarrolle los métodos que se plantean a continuación.

```

public Ciudad darCiudad( String nombreCiudad )
{
}

public void agregarCiudad( String nombre ) throws AerolineaExpcion
{
}

private Ciudad localizarAnterior( String nombre )
{
}

public void eliminarCiudad( String nombre ) throws AerolineaExpcion
{
}

```

■ Retorna la ciudad con el nombre indicado o null si no la encuentra.

■ Agrega una nueva ciudad, verificando previamente que no existe otra ciudad con el mismo nombre.

■ Busca la ciudad anterior a la ciudad con el nombre especificado.

■ Este método se utiliza más adelante como parte del método que elimina una ciudad.

■ Elimina una ciudad dado su nombre. Si no encuentra una ciudad con ese nombre, lanza una excepción.

```

public void agregarVuelo( Ciudad destino, int codigo, Date fecha )
    throws AerolineaExcepcion
{
}

private void verificarInvariante( )
{
}
}

```

- Agrega un nuevo vuelo a una ciudad.
- Si el código del vuelo ya existe (hacia cualquier ciudad a las cuales vuela la aerolínea), lanza una excepción.
- Si no encuentra una ciudad con el nombre que recibe como parámetro, lanza una excepción.

- Verifica que los códigos de todos los vuelos sean únicos.

Tarea 11



Objetivo: Hacer una extensión al programa del caso de estudio, agregando nuevos requerimientos funcionales que permitan poner en práctica lo estudiado en este nivel.

Modifique el programa `n9_aerolinea` que se encuentra en el CD que acompaña al libro, siguiendo las instrucciones que se dan a continuación:

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n9_aerolinea.zip`.
2. Asocie con el botón **Opción 1** un nuevo requerimiento funcional que elimine todos los vuelos de la aerolínea que se encuentren vacíos. Construya todos los métodos que sean necesarios. Asegúrese de hacer una buena asignación de responsabilidades.
3. Extienda las clases de prueba para verificar que los métodos que acaba de agregar cumplen efectivamente sus contratos.

Tarea 12

Objetivo: Hacer una extensión al programa del caso de estudio, agregando nuevos requerimientos funcionales que permitan poner en práctica lo estudiado en este nivel.

Modifique el programa `n9_aerolinea` que se encuentra en el CD que acompaña al libro, siguiendo las instrucciones que se dan a continuación:

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n9_aerolinea.zip`.
2. Asocie con el botón **Opción 2** un nuevo requerimiento funcional que calcula el número total de reservas en todos los vuelos de la aerolínea. Construya todos los métodos que sean necesarios. Asegúrese de hacer una buena asignación de responsabilidades.
3. Extienda las clases de prueba para verificar que los métodos que acaba de agregar cumplen efectivamente sus contratos.

Tarea 13

Objetivo: Hacer una extensión al programa del caso de estudio, agregando nuevos requerimientos funcionales que permitan poner en práctica lo estudiado en este nivel.

Modifique el programa `n9_aerolinea` que se encuentra en el CD que acompaña al libro, siguiendo las instrucciones que se dan a continuación:

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n9_aerolinea.zip`.
2. Asocie con el botón **Opción 3** un nuevo requerimiento funcional que ordena las ciudades ascendentemente por nombre. Construya todos los métodos que sean necesarios. Asegúrese de hacer una buena asignación de responsabilidades.
3. Extienda las clases de prueba para verificar que los métodos que acaba de agregar cumplen efectivamente sus contratos.

Tarea 14

Objetivo: Hacer una extensión al programa del caso de estudio, agregando nuevos requerimientos funcionales que permitan poner en práctica lo estudiado en este nivel.

Modifique el programa `n9_aerolinea` que se encuentra en el CD que acompaña al libro, siguiendo las instrucciones que se dan a continuación:

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n9_aerolinea.zip`.
2. Asocie con el botón **Opción 4** un nuevo requerimiento funcional que cuenta el número de vuelos que hay en una fecha dada. Construya todos los métodos que sean necesarios. Asegúrese de hacer una buena asignación de responsabilidades.
3. Extienda las clases de prueba para verificar que los métodos que acaba de agregar cumplen efectivamente sus contratos.

**Tarea 15**

Objetivo: Hacer una extensión al programa del caso de estudio, agregando nuevos requerimientos funcionales que permitan poner en práctica lo estudiado en este nivel.

Modifique el programa `n9_aerolinea` que se encuentra en el CD que acompaña al libro, siguiendo las instrucciones que se dan a continuación:

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n9_aerolinea.zip`.
2. Asocie con el botón **Opción 5** un nuevo requerimiento funcional que indique el nombre de la ciudad que tiene el mayor número de vuelos. Construya todos los métodos que sean necesarios. Asegúrese de hacer una buena asignación de responsabilidades.
3. Extienda las clases de prueba para verificar que los métodos que acaba de agregar cumplen efectivamente sus contratos.

4.8. Nuevos Distribuidores Gráficos

En los programas que hemos desarrollado hasta ahora hemos utilizado dos distribuidores gráficos sencillos como parte de la interfaz de usuario: el distribuidor en los bordes (`BorderLayout`) y el distribuidor en malla (`GridLayout`).



Un **distribuidor gráfico** (`layout`) es un objeto en Java que se encarga de asignar una posición, dentro de un contenedor gráfico, a cada uno de los componentes que allí se encuentran.

En esta sección presentaremos otros dos distribuidores gráficos disponibles en Java. El primero, muy simple y fácil de usar, implementado por la clase `FlowLayout`, y el segundo, mucho más poderoso y flexible, implementado por la clase `GridBagLayout`.

4.8.1. El Distribuidor Secuencial

El **distribuidor secuencial** (implementado por la clase `FlowLayout`) sitúa todos los componentes que se encuentran dentro del contenedor gráfico, uno después de otro, de izquierda a derecha, dependiendo del orden de llegada y respetando el tamaño preferido que tienen establecido. Si se termina el espacio en una fila, el distribuidor comienza a usar la siguiente. En el ejemplo 16 presentamos la manera como se utiliza este distribuidor y la visualización que se obtiene.

Ejemplo 16

Objetivo: Mostrar la manera de utilizar un distribuidor secuencial.

En este ejemplo mostramos una clase que implementa una ventana de la interfaz de usuario, en la cual agregamos tres paneles usando un distribuidor secuencial.

```
public class InterfazEjemplo extends JFrame
{
    public InterfazEjemplo()
    {
        setLayout( new FlowLayout( FlowLayout.LEFT ) );
        setTitle( "Ejemplo FlowLayout" );
    }
}
```

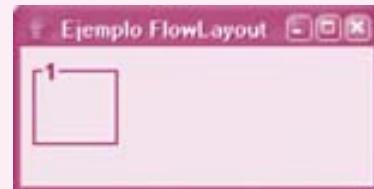


En el constructor de la clase, definimos que vamos a utilizar un distribuidor secuencial, y que queremos que los elementos aparezcan alineados a la izquierda.

```
panel1 = new JPanel();
...
panel1.setPreferredSize( new Dimension(50, 50) );
add( panel1 );
```

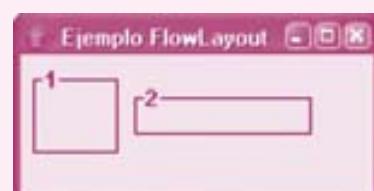
- Si quisieramos que estuvieran centrados o alineados a la derecha, usaríamos las constantes CENTER o RIGHT como parámetro del constructor del distribuidor.

- Si en el constructor no pasamos ningún parámetro, el valor por defecto es CENTER.



```
panel2 = new JPanel();
...
panel2.setPreferredSize( new Dimension(100, 30) );
add( panel2 );
```

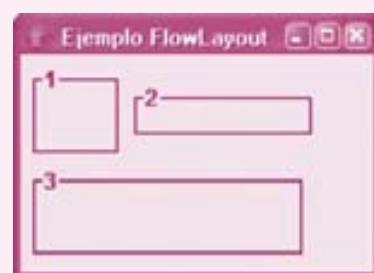
- Creamos el primero de los paneles, definimos su dimensión como de 50 por 50 píxeles y lo agregamos a la ventana.



```
panel3 = new JPanel();
...
panel3.setPreferredSize( new Dimension(150, 50) );
add( panel3 );
}
```

- Creamos el segundo panel de dimensiones 100 por 30 y lo agregamos a la ventana.

- El elemento queda situado a la derecha del anterior panel, respetando sus dimensiones.



- Como no hay espacio en la misma fila, el distribuidor coloca el tercer panel en la segunda fila.



En el CD puede encontrar una herramienta que permite practicar el manejo del distribuidor secuencial, mediante la cual, interactivamente, es posible agregar componente por componente y visualizar el resultado.

4.8.2. El Distribuidor `GridBagLayout`

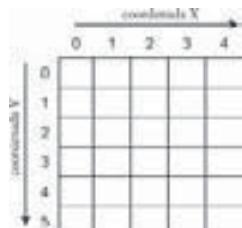
Éste es un distribuidor gráfico muy flexible que permite construir interfaces de usuario con una estructura profesional. Está montado sobre una malla dinámica, en la cual un componente puede utilizar una o más celdas. Tiene la ventaja de poder definir la manera como los componentes deben cambiar de tamaño cuando las dimensiones del contenedor gráfico cambian.

Este distribuidor está implementado por dos clases en Java: la clase `GridBagLayout`, que representa al distribuidor, y la clase `GridBagConstraints`, que describe las características (posición, tamaño, forma de adaptarse a los cambios de tamaño, etc.) de cada uno de los componentes. En el momento de agregar un componente al distribuidor (con el método `add()`) se debe pasar como parámetro una instancia de la clase `GridBagConstraints` con la información antes mencionada.

En esta sección sólo estudiaremos algunas de las características básicas que se pueden utilizar con este distribuidor. Para una presentación completa se recomienda consultar la documentación de la clase.

Un objeto de la clase `GridBagConstraints` tiene definidos (entre otros) los siguientes atributos:

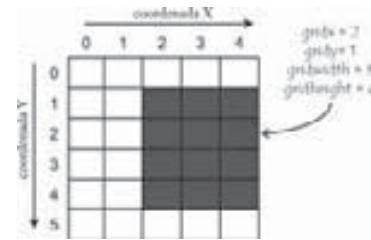
- `gridx`: Define la coordenada X de la celda de la malla (de acuerdo con el sistema de referencia que se muestra a continuación), en la cual comienza la zona que va a ocupar el componente.



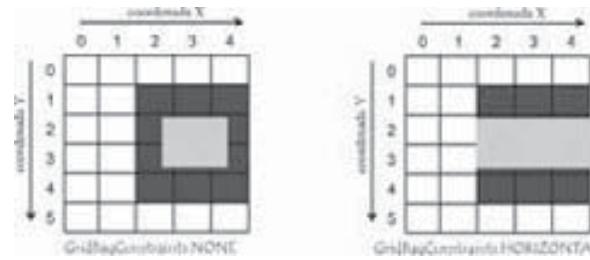
- `gridy`: Define la coordenada Y de la celda de la malla en la cual comienza la zona que va a ocupar el componente. En la coordenada (`gridx`,

`gridy`) se va a encontrar el extremo superior izquierdo del componente.

- `gridwidth`: Número de casillas que va a ocupar el componente en la dirección X.
- `gridheight`: Número de casillas que va a ocupar el componente en la dirección Y.



- `fill`: Sólo se usa cuando el espacio reservado por el componente es mayor que el espacio que el componente necesita. Puede tomar los siguientes valores: `GridBagConstraints.NONE` (es el valor por defecto y corresponde a no hacer nada), `GridBagConstraints.HORIZONTAL` (cambia el tamaño del componente en la dimensión horizontal), `GridBagConstraints.VERTICAL` (cambia el tamaño del componente en la dimensión vertical), o `GridBagConstraints.BOTH` (hace los cambios necesarios en las dos dimensiones para tratar de ocupar todo el espacio disponible en el distribuidor).



- `weightx`: Define la manera como se va a distribuir el espacio extra en la dimensión X, lo cual se usa sobre todo en el caso en el que cambie el tamaño del contenedor gráfico. El distribuidor

gráfico calcula el ancho de la columna como el máximo de este atributo para todas las celdas que la componen. Si un componente define este atributo como cero, no va a recibir ningún espacio extra en caso de que exista. Este valor no es absoluto sino relativo con respecto a los demás valores. Esto quiere decir que si dos elementos comparten la misma fila y uno asigna el valor 3 a este atributo y el otro el valor 7, el primero recibirá el 30% del espacio extra disponible y el segundo el 70%.

- **weighty**: Define la manera como se va a distribuir el espacio extra en la dimensión Y, lo cual se usa sobre todo en el caso en el que cambie el tamaño del contenedor gráfico. Se utiliza de la misma manera que se explicó en el atributo anterior.

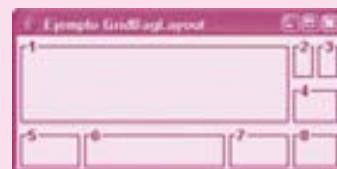
Los atributos de la clase `GridBagConstraints`s son públicos, de manera que es posible modificarlos desde el exterior de la clase, y eso es lo que haremos para simplificar un poco la configuración del distribuidor.

Ejemplo 17



Objetivo: Mostrar la manera de utilizar el distribuidor `GridBagLayout`.

En este ejemplo presentamos la forma de construir una ventana para una interfaz de usuario, con la siguiente distribución de paneles:



En caso de que haya espacio horizontal extra, lo deben aprovechar proporcionalmente los componentes 1, 5, 6 y 7. El espacio vertical extra no debe ser aprovechado. Por ejemplo, si el tamaño de la ventana aumenta, la distribución debe ser como aparece en la siguiente figura:



```
public class InterfazEjemplo3 extends JFrame
{
    public InterfazEjemplo3( )
    {
        setLayout( new GridBagLayout( ) );

        GridBagConstraints c = new GridBagConstraints( );
        ...
    }
}
```

■ Ésta es la declaración de la clase. Allí definimos que la ventana va a utilizar un distribuidor gráfico de la clase `GridBagLayout`.

■ Creamos una instancia de la clase `GridBagConstraints`s que va a definir las características de posición, tamaño y comportamiento de cada uno de los elementos que agreguemos.

```

panel1 = new JPanel( );
// Configuración del panel
...
// Propiedades de localización del componente
...
add( panel1, gbc );

```

Para cada uno de los paneles, vamos a utilizar un fragmento de código parecido al que aparece en el ejemplo.

Primero creamos el panel, luego lo configuramos (le ponemos un borde y un título), después modificamos el objeto de la clase GridBagConstraints para que incluya las propiedades de localización del elemento y finalmente agregamos el panel a la ventana.

En la siguiente tabla se presentan los valores que deben tener los atributos de configuración para cada uno de los paneles:

	panel 1	panel 2	panel 3	panel 4	panel 5	panel 6	panel 7	panel 8
gridx	0	3	4	3	0	1	2	3
gridy	0	0	0	2	3	3	3	3
gridwidth	3	1	1	2	1	1	1	2
gridheight	3	1	1	1	1	1	1	1
fill	BOTH	NONE	NONE	HORIZONTAL	HORIZONTAL	HORIZONTAL	HORIZONTAL	HORIZONTAL
weightx	1	0	0	0	0.2	0.6	0.2	0
weighty	0	0	0	0	0	0	0	0

```

panel6 = new JPanel( );
...
c.gridx = 1;
c.gridy = 3;
c.weightx = 0.6;
c.weighty = 0;
c.gridwidth = 1;
c.gridheight = 1;
c.fill = GridBagConstraints.HORIZONTAL;

add( panel6, c );

```

Este es el código completo de configuración del panel 6.



En el CD puede encontrar una herramienta que permite practicar el manejo del distribuidor GridBagConstraints, mediante la cual, interactivamente, es posible agregar componente por componente y visualizar el resultado.

5. Glosario de Términos

GLOSARIO

Complete las siguientes definiciones como una manera de resumir los conceptos aprendidos en el nivel. Asegúrese de entender bien el significado de cada uno de estos términos, puesto que son la base para poder continuar con los niveles que siguen en el libro.

Estructura lineal sencillamente enlazada:

Estructura lineal doblemente enlazada:

Componente `JDialog`:

Diálogo modal y no modal:

Componente `JRadioButton`:

Componente `ButtonGroup`:

Referencia a un objeto:

Recolector de basura:

Clonación de un objeto:

Diseño de un programa:

Transformación del diagrama de clases:

Patrón de recorrido total:

Patrón de recorrido parcial:

Distribuidor gráfico:

Diagrama de casos de uso:

Actor:

Caso de uso:

Clase `Date`:

Clase `SimpleDateFormat`:

Clase `FlowLayout`:

Clase `GridBagLayout`:

6. Hojas de Trabajo



6.1. Hoja de Trabajo N° 1: Una Agenda

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

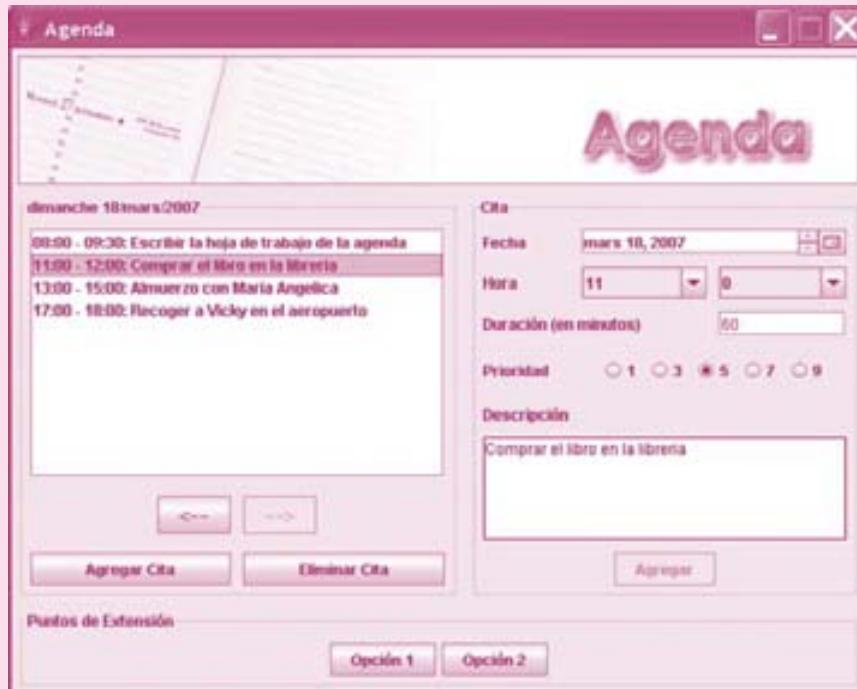
Se quiere construir un programa para manejar la información de una agenda de citas y/o tareas diarias de una persona. Las citas se crean para un día determinado con la siguiente información: hora de inicio (horas y minutos), duración en minutos de la cita, prioridad (1, 3, 5, 7 ó 9) y descripción de la cita.

En la aplicación, el usuario debe poder: (1) agregar una nueva cita (dando toda la información relacionada), (2)

eliminar una cita (especificando la fecha y la hora de inicio), (3) consultar todas las citas de un día organizadas por hora, (4) avanzar hasta el siguiente día en la agenda que tenga una cita, (5) retroceder hasta el anterior día en la agenda que tenga una cita y (6) consultar los datos de una cita (especificando la fecha y la hora de inicio).

Sólo se pueden hacer citas entre las 8:00 a.m. y las 7:00 p.m.

La interfaz de usuario del programa debe ser la siguiente:



La información de la agenda debe ser persistente y el proceso debe ser completamente transparente para el usuario. El programa debe ser capaz de guardar la información en un archivo, usando el formato vCalendar, de manera que se garantice compatibilidad con otros programas que existen en el mercado y que utilizan dicho formato para almacenar su información.

Para la representación de la información de la agenda, se ha decidido utilizar una estructura doblemente enlazada con un elemento por cada uno de los días en los cuales hay por lo menos una cita (ordenada ascendentemente por fecha), y, para cada uno de ellos, una lista sencillamente enlazada con las citas ordenadas ascendentemente por hora. Esto va a facilitar el proceso de inserción y supresión de elementos.

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

Requerimiento funcional 1	Nombre	R1 – Agregar una nueva cita
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Eliminar una cita
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Consultar todas las citas de un día organizadas por hora
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Avanzar hasta el siguiente día en la agenda que tenga una cita
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Retroceder hasta el anterior día en la agenda que tenga una cita
	Resumen	
	Entrada	
	Resultado	

Requerimiento funcional 6	Nombre	R6 – Consultar los datos de una cita
	Resumen	
	Entrada	
	Resultado	

Modelo conceptual. Construya el diagrama de clases, asegurándose de incluir las siguientes entidades: Agenda, Día y Cita.

Invariante de clase. Defina el invariante para cada una de las clases del diagrama anterior.

Clase Agenda	Clase Día	Clase Cita

Requerimientos no funcionales. Identifique los requerimientos no funcionales planteados en el enunciado.

Diagrama de clases del diseño. Transforme el diagrama de clases del análisis para que (1) el grupo de elementos de la clase **Día** se maneje como una estructura doblemente enlazada, ordenada ascendente por fecha, y (2) el grupo de elementos de la clase **Cita** se represente con una estructura sencillamente enlazada, ordenada de manera ascendente por hora.

Declaración de las clases. Declare en Java las clases (sólo atributos y asociaciones) del diseño anterior.

```
public class Agenda  
{  
}  
}
```

```
public class Dia  
{  
}  
}
```

```
public class Cita  
{  
}  
}
```

Desarrollo de métodos. Utilice los métodos que se describen a continuación para implementar los métodos que se plantean más adelante.

Clase	Método	Descripción
Cita	int darHora()	Retorna la hora de la cita.
Cita	int darDuracion()	Retorna la duración de la cita.
Cita	void cambiarSiguiente(Cita sig)	Cambia la cita siguiente por el valor recibido como parámetro.
Cita	Cita darSiguiente()	Retorna la cita siguiente del día.
Cita	void insertarDespues(Cita cita)	Inserta la cita que recibe como parámetro después de la actual.
Cita	void desconectarSiguiente()	Desconecta la cita que le sigue a la cita actual.

```
public class Dia
{
    public boolean estaHoraLibre( int hora, int duracion )
    {
```

Este método permite establecer si una hora está libre para una cita de una determinada duración, en el día representado por el objeto de la clase.

Utiliza un método auxiliar que permite calcular la hora a la cual va a terminar la cita.

```
}
```

```
private int calcularHoraFin( int hora, int duracion )
{
```

```
}
```

```
public class Dia
{
    public void agregarCita( Cita nuevaCita )
        throws CitaInvalidaExcepcion
    {
        }
    }
```

Este método agrega una nueva cita en la posición que le corresponde según la hora. Se lanza una excepción si esta cita no se puede poner en ese día porque se cruza con otra cita.

```
public class Dia
{
    public void eliminarCita(int horaCita)
        throws CitaInexistenteExpcion
    {

    }

    private Cita localizarAnterior( int hora )
    {
        }

    }
}
```

Este método elimina una cita del día dada su hora de inicio. Se lanza una excepción si no se encuentra una cita a la hora definida por el parámetro.

Utiliza un método auxiliar que localiza la cita anterior a la cita con una hora dada.

Desarrollo de métodos. Utilice los métodos que se describen a continuación para implementar los métodos que se plantean más adelante.

Clase	Método	Descripción
Dia	void cambiarSiguiente(Dia sig)	Cambia el día siguiente por el que recibe como parámetro.
Dia	Dia darSiguiente()	Retorna el siguiente día.
Dia	void cambiarAnterior(Dia ant)	Cambia el día anterior por el que recibe como parámetro.
Dia	Dia darAnterior()	Retorna el día anterior.
Dia	void insertarDespues(Dia dia)	Inserta el día que recibe como parámetro después del actual.
Dia	void insertarAntes(Dia dia)	Inserta el día que recibe como parámetro antes del actual
Dia	int compararPorFecha(Dia d2)	Retorna 0, 1 ó -1, siguiendo el estándar de los métodos de comparación.

```
public class Agenda
{
    public Dia buscarDia( Date fecha )
    {
```

Este método localiza un día en la agenda, dada una fecha. Si no lo encuentra retorna el valor null.

```
}
```

```
public class Agenda
{
    public Dia crearDia( Date fecha )
    {
        }
}
```

Este método crea un nuevo día en la agenda con la fecha indicada y lo deja en su lugar dentro de la lista. Suponga que en la agenda no hay inicialmente un día con la fecha indicada. El método retorna el objeto de la clase `Dia` que fue creado.

```
public class Agenda
{
    public void eliminarDiaSinCitas( Date fecha )
    {
        }
}
```

Este método elimina de la agenda un día para el cual no hay ninguna cita. Puede suponer que el día descrito por el parámetro existe y que no tiene ninguna cita presente.



6.2. Hoja de Trabajo Nº 2: Un Manejador de ADN

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

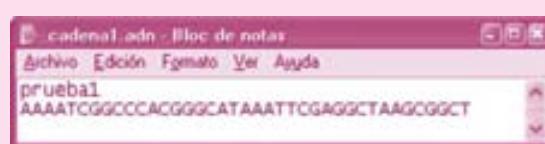
Una cadena de ADN es una secuencia muy larga de elementos sencillos llamados bases nitrogenadas, que pueden ser de cuatro tipos. Diferentes secciones de una cadena determinan características diferentes en un individuo: la forma en la que se presenta esa característica depende exclusivamente de los tipos de bases nitrogenadas que haya dentro de la sección correspondiente de la cadena. Los tipos de base nitrogenada son cuatro: adenina (A), timina (T), guanina (G) y citosina (C).

En los seres vivos las cadenas de ADN no se componen de un solo hilo sino de dos hebras entrelazadas que forman una hélice, y están unidas por sus bases nitrogenadas. Para que esta hélice pueda formarse se

deben respetar reglas que indican cuáles bases se pueden juntar con cuáles otras: la adenina solamente puede juntarse con la timina, mientras que la citosina solamente puede juntarse con la guanina. Dada una cadena de ADN, si ésta se puede doblar por la mitad y formar los emparejamientos correctamente, en el contexto de esta aplicación vamos a llamarla coherente.

El programa que se quiere construir debe permitir la manipulación y visualización de un grupo de cadenas de ADN.

Las cadenas iniciales se deberán cargar a partir de archivos de texto, cuyo formato se ilustra en el siguiente ejemplo:



Las cadenas que se construyan transformando las iniciales deberán poder ser guardadas en archivos de texto usando el mismo formato. Cada cadena tiene un nombre que es único, el cual viene definido dentro del archivo (en el ejemplo, la cadena se llama "prueba1").

Las operaciones que se pueden realizar sobre las cadenas son: (1) cargar una cadena de un archivo (dado el nombre del archivo), (2) guardar una cadena en un archivo (dados el nombre del archivo y el nombre de la cadena), (3) eliminar una cadena del programa (dado su nombre), (4) visualizar una o varias de las cadenas que se encuentran disponibles en el programa (dados sus nombres), (5) agregarle a una cadena las bases nitrogenadas de otra (dados los nombres de las dos cadenas), (6) corregir una mutación en una cadena dada (entra la secuencia de bases nitrogenadas correspondiente a la mutación y la secuencia por la cual se debe remplazar para corregirla), (7) crear una nueva cadena llamada "CadenaComún" con el fragmento común más largo a un par de cadenas (dados los nombres de las dos cadenas) y (8) eliminar un fragmento

de una cadena (dado el nombre de una cadena y una secuencia de bases nitrogenadas).

La interfaz de usuario que debe tener el programa es la siguiente:



En la parte izquierda de la interfaz de usuario se puede visualizar el grupo de cadenas de ADN que han sido seleccionadas en la parte derecha. En la parte de abajo aparecen los botones que ejecutan los distintos

requerimientos funcionales. Fíjese que como parte de la visualización de cada cadena aparece su nombre, si es o no coherente, el número de bases y la secuencia de bases que la componen, cada una con un color distinto.

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

Requerimiento funcional 1	Nombre	R1 – Cargar una cadena de un archivo
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Guardar una cadena en un archivo
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Eliminar una cadena
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Visualizar una o varias de las cadenas
	Resumen	
	Entrada	
	Resultado	

Requerimiento funcional 5	Nombre	R5 – Agregarle a una cadena las bases nitrogenadas de otra
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 6	Nombre	R6 – Corregir una mutación en una cadena dada
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 7	Nombre	R7 – Calcular el fragmento común más largo entre un par de cadenas
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 8	Nombre	R8 – Eliminar un fragmento de una cadena
	Resumen	
	Entrada	
	Resultado	

Modelo conceptual. Construya el diagrama de clases, asegurándose de incluir las siguientes entidades: ProcesadorAdn, Cadena y BaseNitrogenada.

Invariante de clase. Defina el invariante para cada una de las clases del diagrama anterior.

Clase ProcesadorAdn	Clase Cadena	Clase BaseNitrogenada

Requerimientos no funcionales. Identifique los requerimientos no funcionales planteados en el enunciado.

Diagrama de clases del diseño. Transforme el diagrama de clases del análisis para que (1) la clase **ProcesadorADN** maneje el grupo de cadenas, organizadas como una estructura sencillamente enlazada, ordenada ascendentemente por tamaño, y (2) la clase **Cadena** represente las bases nitrogenadas que la componen utilizando una estructura doblemente enlazada. Para poder hacer más eficientes las operaciones definidas sobre las cadenas, en esta clase se va a tener un atributo que haga referencia al primer elemento y otro atributo que haga referencia al último.

Declaración de las clases. Declare en Java las clases (sólo atributos y asociaciones) del diseño anterior.

```
public class ProcesadorAdn
{
}
```

```
public class Cadena
{
}
```

```
public class BaseNitrogenada
{
}
```

Desarrollo de métodos. Utilice los métodos que se describen a continuación para implementar los métodos que se plantean más adelante.

Clase	Método	Descripción
BaseNitrogenada	int darTipo()	Retorna el tipo de la base nitrogenada.
BaseNitrogenada	BaseNitrogenada darSiguiente()	Retorna la base siguiente.
BaseNitrogenada	BaseNitrogenada darAnterior()	Retorna la base anterior.
BaseNitrogenada	void cambiarAnterior(BaseNitrogenada b)	Cambia la base nitrogenada anterior.
BaseNitrogenada	void cambiarSiguiente(BaseNitrogenada b)	Cambia la base nitrogenada siguiente.
BaseNitrogenada	void insertarDespues(BaseNitrogenada b)	Inserta la base después de la actual.
BaseNitrogenada	boolean puedeEmparejar(BaseNitrogenada b)	Indica si esta base nitrogenada puede emparejarse con la base nitrogenada que llega como parámetro.

```
public class Cadena
{
    public void agregarBase( BaseNitrogenada nuevaBase )
    {
        }
}
```

Este método agrega una base al final de la cadena.

```
public class Cadena
{
    public int darLongitud( )
    {
        }
}
```

Este método retorna el número de bases nitrogenadas que tiene la cadena.

```
public class Cadena
{
    public void agregarBasesACadena( Cadena otraCadena )
    {
        }
}
```

Este método agrega a la cadena las bases nitrogenadas de la cadena que recibe como parámetro.

```
}
```

Este método se encarga de construir una nueva cadena en la cual todas las ocurrencias de la secuencia `cadenaMutada` son reemplazadas por la secuencia `reemplazo`.

Utiliza un método auxiliar que indica si a partir de la base indicada (`inicioCadena`) comienza una secuencia de bases nitrogenadas igual a la de la cadena mutada indicada.

```
public class Cadena
{
    public Cadena crearCadenaCorregida( BaseNitrogenada cadenaMutada,
                                         BaseNitrogenada reemplazo )
    {

    }

    private boolean empiezaSecuencia( BaseNitrogenada inicioCadena,
                                      BaseNitrogenada cadenaMutada )
    {
        }

    }
}
```

Este método elimina de la cadena el primer fragmento en el cual se encuentre la secuencia que llega como parámetro.

Utiliza el método auxiliar desarrollado en el punto anterior.

```
public class Cadena
{
    public void eliminarFragmento( BaseNitrogenada secuenciaBuscada )
    {
        }

    }
}
```

Este método indica si la cadena es coherente. Una cadena es coherente si se pueden establecer parejas entre las bases de tal forma que las adeninas siempre se junten con las timinas y las guaninas siempre se junten con las citosinas. Las parejas deben realizarse de la siguiente manera: primera base con la última base, segunda base con la penúltima base, tercera base con la antepenúltima base, etc.

```
public class Cadena
{
    public boolean esCoherente( )
    {
        }
}
```

Este método retorna la cadena común más larga entre la cadena sobre la que se invoca el método y la cadena que llega como parámetro.

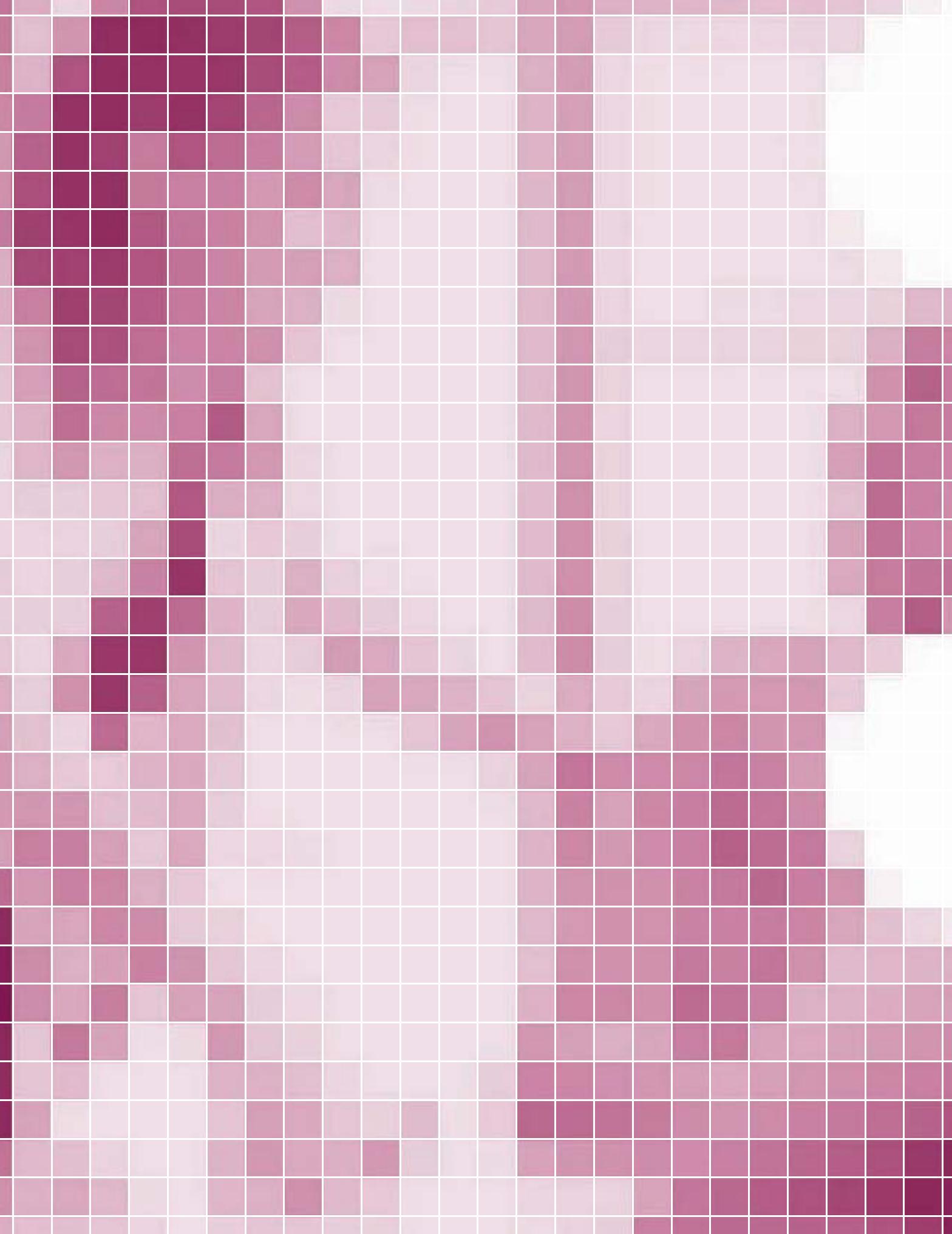
La cadena resultante debe tener el nombre "CadenaComún".

Utiliza un método auxiliar que se encarga de encontrar la secuencia común que hay al inicio de dos cadenas que empiezan por bases iguales.

```
public class Cadena
{
    public Cadena buscarSecuenciaComun(Cadena otraCadena )
    {
        }

    private Cadena buscarSecuenciaComun( BaseNitrogenada inicioCadena1,
                                         BaseNitrogenada inicioCadena2 )
    {
        }

}
```



Nivel 4

Mecanismos de Reutilización y Desacoplamiento

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Explicar la importancia de desacoplar las clases que hacen parte de un programa y utilizar interfaces para independizar los contratos funcionales de las implementaciones concretas. Esto con el fin de hacer más flexible y fácil de cambiar el programa que se construya.
- Utilizar las interfaces `Collection`, `List` e `Iterator` de Java que permiten la manipulación abstracta de estructuras contenedoras.
- Explicar la importancia de la herencia como mecanismo de reutilización, con el cual es posible construir nuevas clases a partir de clases ya existentes, las cuales han sido diseñadas con el propósito de facilitar la implementación de una familia de entidades que comparten elementos en común.
- Utilizar la herencia como mecanismo de construcción de aplicaciones en Java y entender el papel que juega la clase `Object` en dicho lenguaje.
- Entender el uso que le hemos dado a la herencia en niveles anteriores, para construir interfaces de usuario y tipos de excepciones.
- Construir interfaces de usuario que incluyan menús de opciones y gráficas simples en dos dimensiones.

2. Motivación

Es un hecho que los programas de computador que se construyen deben evolucionar, a medida que el problema y el contexto en el que éste ocurre cambian. Piense por ejemplo en el caso de la aerolínea del nivel anterior. ¿Qué pasa si en lugar de un solo tipo de avión debemos manejar varios? ¿Qué tan difícil es generar un nuevo reporte de los vuelos si las autoridades aeroportuarias así lo exigen? Estas modificaciones en el programa corresponden tanto a cambios en los requerimientos funcionales como en el mundo del problema, y difícilmente pueden predecirse en la etapa de análisis. Un programa de buena calidad es aquél que además de resolver el problema actual que se plantea, tiene una estructura que le permite evolucionar a costo razonable. No nos interesa construir un programa que tengamos que rehacer cada vez que algún aspecto del problema cambie o un nuevo requerimiento aparezca.

En este nivel introduciremos el concepto de **interfaz**, como un mecanismo que nos va a permitir construir programas en donde el acoplamiento entre las clases (nivel de dependencia entre ellas) sea bajo, de manera que podamos cambiar una de ellas con un bajo impacto sobre todas las demás. Esto nos va a facilitar la evolución de los programas, puesto que las clases van a poder disfrutar de un cierto nivel de aislamiento que les permitirá adaptarse a los cambios sin impactar las demás clases presentes o, por lo menos, con un impacto menor. Una interfaz la podemos ver como un contrato funcional expresado en términos de un conjunto de signaturas, el cual representa los compromisos que debe asumir cualquier clase que quiera jugar un papel particular dentro de un programa. Consideremos de nuevo el caso de la aerolínea.



No se debe confundir el término interfaz que utilizaremos en este nivel (un contrato funcional) con la interfaz de usuario de un programa (la parte de la aplicación encargada de la comunicación con el usuario).

¿Qué pasa si en lugar de la clase `Vuelo` definimos la lista de métodos que cualquier clase que quiera implementar esa parte del modelo del mundo deba incluir? En ese caso definiríamos la interfaz `IVuelo`, una de cuyas implementaciones posibles sería la clase `Vuelo` que ya hicimos.

Además de hacer evolucionar los programas a bajo costo, nos enfrentamos siempre al reto de hacer las implementaciones de la manera lo más eficiente posible (en tiempo e inversión). Para esto, lo ideal sería poder aprovechar otros desarrollos previos que nos ayuden en la construcción de nuestro programa. Esa es la idea del mecanismo de reutilización de clases denominado **herencia**, tema de este nivel. Imagínese lo elevados que serían los costos si cada vez que nos lanzamos a construir un nuevo programa debemos partir desde cero. ¿Qué tal tener que implementar cada vez las clases contenedoras? ¿Qué tal si en lugar de utilizar la clase `JFrame`, como hemos hecho hasta ahora, tuviéramos que implementarla? Es mucho más simple basarnos en la implementación existente en Java de una ventana y contentarnos con extenderla, precisando en el constructor el contenido que queremos que tenga, pero aprovechando su comportamiento básico y los métodos que ya fueron implementados (recuerde que usamos los métodos `add()`, `setTitle()` y `setLayout()` en la construcción de nuestra ventana como si nosotros los hubiéramos implementado). Para eso utilizamos la palabra `extends` en la declaración de nuestra clase, que es la manera en Java de expresar que una clase hereda (especializa o extiende) de otra clase. Este mecanismo lo hemos usado desde niveles anteriores, pero sólo hasta ahora lo vamos a estudiar a fondo y a ver cómo construir ese tipo de clases.

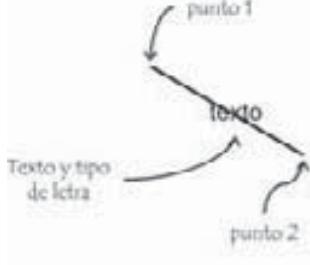
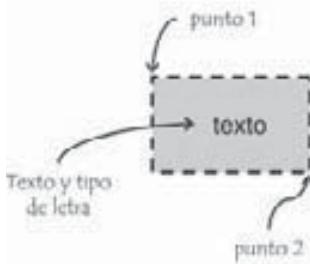
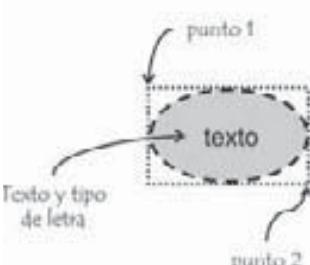
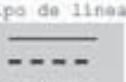
El último tema que vamos a tratar en este nivel tiene que ver con el manejo de dibujos en dos dimensiones (líneas, círculos, cuadrados, etc.), y con el manejo de menús de opciones. Con estos dos nuevos elementos cubrimos los programas que requieren de este tipo de elementos gráficos para interactuar con el usuario, como editores de planos de edificaciones, editores de dibujos, editores de lenguajes gráficos, etc.

3. Caso de Estudio N° 1: Un Editor de Dibujos

En este caso queremos construir un editor (que llamaremos Paint) que nos permita crear dibujos que contengan líneas, rectángulos y figuras ovaladas. Las líneas deben tener asociados un color, un punto inicial, un punto final, un texto, un tipo de letra (*font*), al igual que un ancho y un tipo de línea (punteada, continua, etc.).

Los rectángulos deben estar definidos por dos puntos, por un texto con su tipo, por un color de fondo y por el tipo de línea que debe dibujarse sobre el perímetro. Las figuras ovaladas están definidas por dos puntos, los cuales describen el rectángulo en el cual está incluida la figura, por un texto con su tipo, por un color de fondo y por el tipo de línea del borde del óvalo. En la figura 4.1 se resumen las características con las que se puede describir cada una de las figuras.

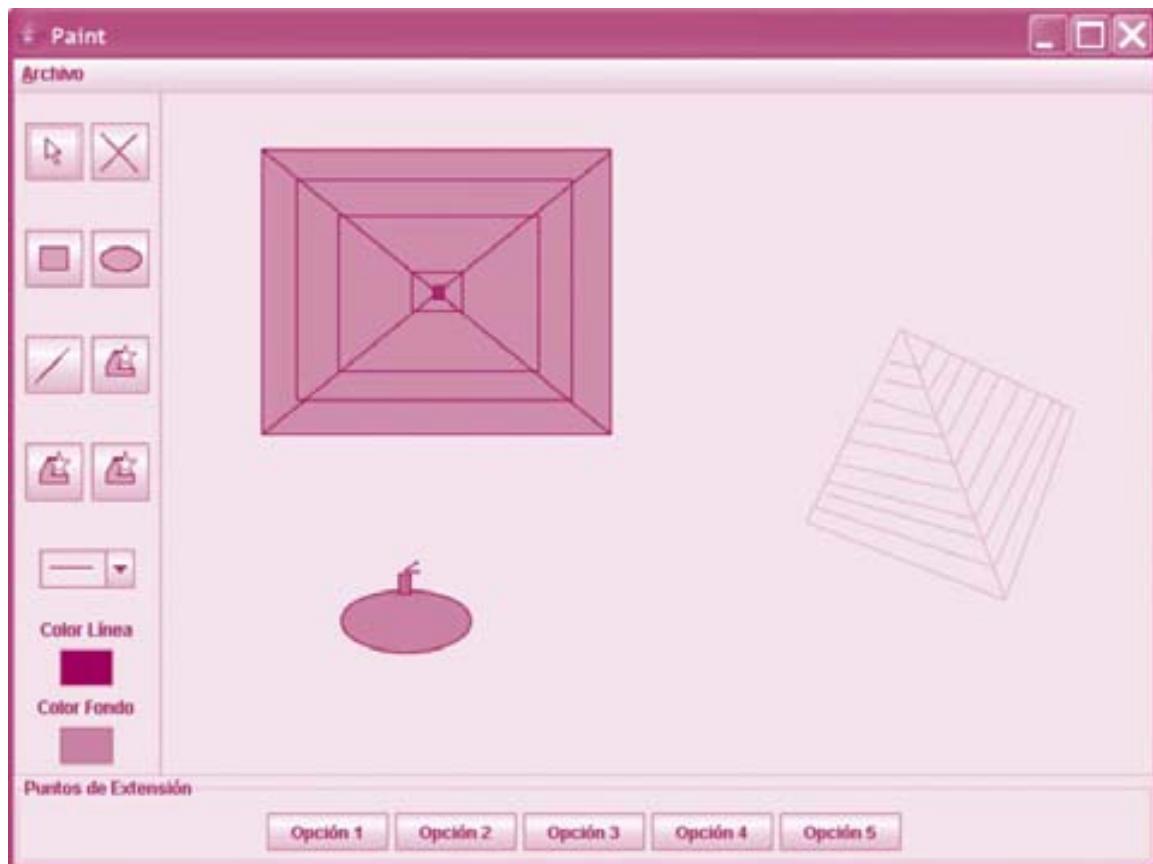
Fig. 4.1 – Características de las figuras del editor

 Características – Color de línea Tipo de línea  Ancho de línea 	<ul style="list-style-type: none"> ■ Cada punto está definido por dos valores de tipo entero (coordenada en X y coordenada en Y). ■ El color debe corresponder a cualquiera de los colores que maneja el lenguaje Java. ■ El texto debe aparecer en la mitad de la línea y debe tener asociado uno de los tipos de letra que maneja el computador. ■ El editor debe manejar tres tipos de línea, los cuales aparecen en el ejemplo. ■ El editor debe permitir tres anchos de línea distintos.
 Características – Color de fondo Color de línea Tipo de línea  Ancho de línea 	<ul style="list-style-type: none"> ■ Con los dos puntos se definen los dos vértices opuestos del rectángulo. ■ La línea que va sobre el perímetro de la figura tiene un color, un tipo y un ancho. ■ Debe ser posible definir el color de relleno del rectángulo. ■ Se debe poder definir un texto para incluir dentro de la figura, al cual se le puede asociar cualquier tipo de letra.
 Características – Color de fondo Color de línea Tipo de línea  Ancho de línea 	<ul style="list-style-type: none"> ■ Con los dos puntos se define el rectángulo dentro del cual se incluye la figura ovalada. <p>Esta figura debe manejar las mismas características de la figura anterior: color de fondo, color de línea, tipo de línea y ancho de línea.</p>

Las opciones que debe ofrecer el programa son: (1) agregar una nueva figura al dibujo sin ningún texto asociado, dando la información necesaria para crearla (los puntos, los colores, etc.); (2) seleccionar una de las figuras que hacen parte del dibujo, (3) eliminar la figura

seleccionada del dibujo, (4) cambiar el texto asociado con la figura seleccionada, (5) salvar el dibujo en un archivo y (6) cargar un dibujo de un archivo. En la figura 4.2 aparece la interfaz de usuario que debe tener el programa.

Fig. 4.2 – **Interfaz de usuario del editor de dibujos**



- Arriba a la izquierda aparece el punto de inicio del menú plegable **Archivo**, que permite abrir un archivo con un dibujo o salvar el dibujo que en ese momento se esté editando.
- Con los botones de la izquierda, que tienen la imagen de la respectiva figura, se agregan nuevos elementos al dibujo. Para hacerlo se debe seleccionar el respectivo botón, escoger el color de la línea y el color del fondo (si no es una línea). Luego, se deben indicar con el ratón los dos puntos en la zona de edición. Al dar el segundo punto, aparece la figura con las características pedidas.
- Si se siguen dando puntos, se siguen creando figuras del mismo tipo. Para terminar el modo inserción se debe seleccionar el botón con la flecha.

Para seleccionar una figura del editor se debe hacer clic con el ratón sobre ella (sin estar en modo de inserción). Si hay una imagen seleccionada, ésta debe mostrarse claramente al usuario. Para eliminar la figura seleccionada se utiliza el botón marcado con una X, el cual aparece en la parte superior izquierda de la ventana.

Es importante que cada vez que el usuario vaya a salir del programa sin haber salvado su trabajo, o cuando vaya a cargar un nuevo dibujo sin haber hecho persistir las modificaciones del actual, el editor le pregunte al usuario si desea salvar antes de continuar.

La persistencia se debe hacer en archivos secuenciales de texto, utilizando para esto cualquier formato decidido por el diseñador, pero permitiendo que los archivos se puedan visualizar y modificar desde un editor de texto. Esto puede facilitar en algunos casos la creación de dibujos desde un programa.

La principal restricción del editor es que debe estar construido de manera que sea fácilmente extensible, con nuevas figuras y nuevos requerimientos funcionales. La interfaz de usuario, por ejemplo, ya tiene listos los botones para tres nuevos tipos de imagen.

3.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none">  Proponer un diseño que sea fácilmente extensible, el cual permita la incorporación de nuevas figuras y requerimientos funcionales, a bajo costo. 	<ul style="list-style-type: none">  Estudiar el mecanismo de herencia, como una manera de construir programas que tienen una estructura que permite que los nuevos elementos sean creados por extensión de elementos ya existentes.  Estudiar las interfaces como un medio para obtener diseños en los cuales las clases tienen un bajo nivel de acoplamiento, lo cual facilita la evolución del programa.
<ul style="list-style-type: none">  Manejar en archivos secuenciales de texto la persistencia de los dibujos. 	<ul style="list-style-type: none">  Repasar el manejo de archivos secuenciales de texto y la manera de asignar responsabilidades de persistencia entre las clases.
<ul style="list-style-type: none">  Utilizar menús plegables de opciones, para facilitar la interacción con el usuario. 	<ul style="list-style-type: none">  Estudiar los componentes <code>JMenuBar</code>, <code>JMenu</code> y <code>JMenuItem</code> del <i>framework</i> gráfico de Java y la manera de incorporarlos al resto de la interfaz de usuario.
<ul style="list-style-type: none">  Dibujar en la zona de edición del programa figuras geométricas en dos dimensiones y tomar algunos eventos generados por el ratón sobre dicha zona. 	<ul style="list-style-type: none">  Estudiar la manera de dibujar figuras en un programa.  Estudiar las clases básicas de manejo geométrico en Java (<code>Line2D</code>, <code>Rectangle2D</code> y <code>Ellipse2D</code>).  Estudiar los eventos generados por el ratón que pueden ser escuchados e interpretados por un programa, los cuales hacen parte de la interfaz <code>MouseListener</code>.

3.2. Comprensión de los Requerimientos

Los seis requerimientos funcionales definidos en el enunciado están dirigidos a un solo actor que va-

mos a denominar "Usuario". En la siguiente tarea pedimos al lector que especifique cada uno de dichos requerimientos, haciendo explícitas las entradas que el usuario debe suministrar y el resultado obtenido.

Tarea 1



Objetivo: Entender el problema del caso de estudio del editor de dibujos.

(1) Lea detenidamente el enunciado del caso de estudio del editor de dibujos e (2) identifique y complete la documentación de los requerimientos funcionales.

Haga el diagrama de casos de uso:

Requerimiento funcional 1	Nombre	R1 – Agregar una nueva figura	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 2	Nombre	R2 – Seleccionar una figura	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 3	Nombre	R3 – Eliminar la figura seleccionada	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			

Requerimiento funcional 4	Nombre	R4 – Cambiar el texto de la figura seleccionada	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 5	Nombre	R5 – Salvar el dibujo en un archivo	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 6	Nombre	R6 – Cargar un dibujo de un archivo	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			

Hay tres requerimientos no funcionales que fueron expresados en el enunciado y que deben hacerse explícitos en los documentos producidos en la etapa de análisis. Éstos son:

Requerimiento no funcional 1	Tipo: Persistencia
	Descripción: <ul style="list-style-type: none"> • La información del modelo del mundo debe poder ser persistente. • Se deben usar archivos secuenciales de texto para almacenar la información, y estos archivos se deben poder visualizar y modificar con cualquier editor de texto.
Requerimiento no funcional 2	Tipo: Extensibilidad
	Descripción: <ul style="list-style-type: none"> • El programa debe estar diseñado de tal manera que su evolución sea sencilla y a bajo costo, permitiendo así integrar nuevos tipos de figuras y nuevos requerimientos funcionales.

Requerimiento no funcional 3

Tipo: Visualización e interacción

Descripción:

- Para la creación de las figuras, el usuario debe poder utilizar el ratón directamente sobre la zona de edición, para indicar sus coordenadas.
- El usuario debe poder seleccionar una figura del dibujo haciendo clic sobre ella.

3.3. Interfaces: Compromisos Funcionales

Al comenzar a estudiar el modelo del mundo del problema nos encontramos con dos elementos principales, relacionados tal como aparece en la figura 4.3: la entidad **Dibujo** y la entidad **Figura**. En el diagrama de clases se puede ver que hay una relación con cardinalidad indefinida, para indicar que un dibujo tiene asociado un grupo de figuras. Por ahora no vamos a entrar en los detalles de cada una de estas dos clases, sino que vamos a tratar de buscar una manera de desacoplarlas para que puedan evolucionar de la manera lo más independiente posible y satisfacer así uno de los requerimientos no funcionales del enunciado.

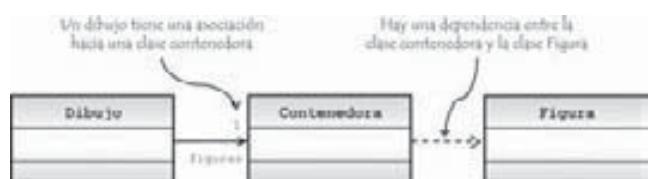
Fig. 4.3 – **Diagrama de análisis del editor de dibujos**



Para empezar debemos reconocer que existe una tercera entidad, que sólo aparece en la etapa de diseño, y que representa la contenedora de figuras. Con el fin de hacerla explícita, transformamos el diagrama de clases del análisis en un primer borrador de diseño, el cual se muestra en la figura 4.4. Por ahora vamos a

llamar Contenedora a esa entidad, pero eso cambiará más adelante, cuando hayamos terminado de refinar nuestro diseño.

Fig. 4.4 – **Primer borrador de diseño del editor de dibujos**



Vamos a concentrarnos ahora en la entidad **Figura**. El reto es tratar de encontrar una manera de implementar la clase **Dibujo** sin depender directamente de las clases **Línea**, **Rectángulo** y **Ovalo**, que van a representar las figuras que menciona el enunciado. Si logramos hacer esto, va a ser más fácil añadir nuevas figuras más tarde durante la evolución del programa. La pregunta que nos debemos hacer es, ¿existe algo que tengan en común todas las figuras que queremos incluir en el dibujo? La respuesta es simple: todas tienen en común las responsabilidades que deben asumir sus métodos, porque aunque cada figura los puede implementar de una manera distinta, todos deberían tener los mismos. La idea con una interfaz es agrupar bajo un nombre un conjunto de firmas de métodos para representar los compromisos funcionales de una o varias clases, de cuya implementación no nos queremos preocupar por ahora.

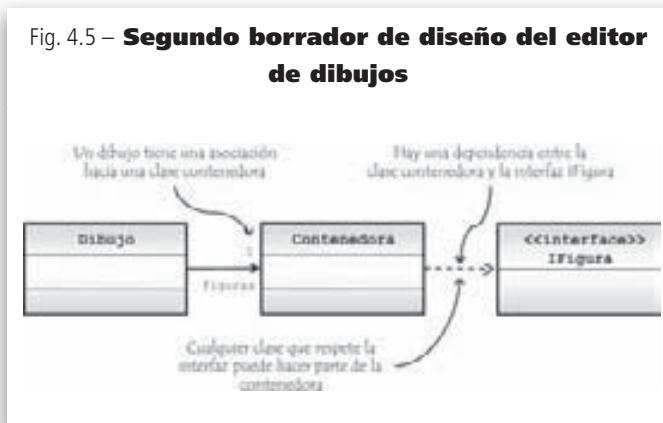
Considere la siguiente definición en Java de una interfaz, que representa la responsabilidad funcional de

cualquier figura que quiera hacer parte de un dibujo en nuestro editor:

<pre>package uniandes.cupi2.paint.mundo; public interface IFigura {</pre>	<ul style="list-style-type: none"> ■ Una interfaz en Java se declara de manera parecida a como se declara una clase, utilizando la palabra "interface". ■ Una interfaz se guarda en un archivo con el mismo nombre de la interfaz (IFigura.java), respetando también la estructura de paquetes.
<pre> public void dibujar(Graphics2D g, boolean seleccionada);</pre>	<ul style="list-style-type: none"> ■ Dibuja la figura sobre la superficie que se recibe como parámetro. Si la figura está seleccionada (lo indica el segundo parámetro), la debe pintar de manera diferente.
<pre> public boolean estaDentro(int x, int y);</pre>	<ul style="list-style-type: none"> ■ Sirve para saber si un punto está dentro de una figura o no. Cada figura establece qué quiere decir que un punto dado esté en su interior. Sirve para seleccionar una figura.
<pre> public String darTexto();</pre>	<ul style="list-style-type: none"> ■ Retorna el texto asociado con la figura.
<pre> public void cambiarTexto(String txt);</pre>	<ul style="list-style-type: none"> ■ Cambia el texto asociado con la figura por la cadena que se recibe como parámetro.
<pre> public Font darTipoLetra();</pre>	<ul style="list-style-type: none"> ■ Retorna el tipo de letra actual del texto. La clase Font es una clase de Java que se encuentra en el paquete java.awt.
<pre> public void cambiarTipoLetra(Font fuenteTexto);</pre>	<ul style="list-style-type: none"> ■ Cambia el tipo de letra actual del texto.
<pre> public String darTipoFigura();</pre>	<ul style="list-style-type: none"> ■ Retorna la cadena de caracteres que va a identificar este tipo de figuras en la persistencia.
<pre>}</pre>	<ul style="list-style-type: none"> ■ Guarda la figura en un archivo que recibe como parámetro.

Si aceptamos que esos ocho métodos constituyen el contrato funcional que debe respetar cualquier figura de nuestro editor, podemos escribir la clase `Dibujo` completamente en abstracto y más tarde construir las clases que implementan las figuras inicialmente pedidas en el enunciado. Introduciendo esta interfaz en el diseño, obtenemos el diagrama de clases de la figura 4.5, en donde se ilustra la sintaxis para representar interfaces en UML.

Fig. 4.5 – **Segundo borrador de diseño del editor de dibujos**



Es usual que los nombres de las interfaces comiencen por la letra “I”, en mayúsculas, para así hacer explícito que no se trata de una clase concreta sino de un contrato funcional.

1



Una interfaz está constituida por un conjunto de signaturas de métodos, a los cuales se les asocia un nombre para representar un contrato funcional. Una interfaz se declara en Java en un archivo aparte que tiene el mismo nombre de la interfaz, respetando la misma jerarquía de paquetes de las clases.

¿Y cómo diseñar los métodos que se deben incluir en una interfaz? ¿Cómo saber si hace falta alguno o si los parámetros de los métodos están completos? Ese tema lo abordaremos en un nivel posterior. Por ahora nos concentraremos en la manera de utilizar interfaces diseñadas por alguien más.

3.4. Referencias de Tipo Interfaz

Una vez definida una interfaz podemos declarar atributos, parámetros o variables de ese tipo y manipularlos como si fueran referencias a objetos normales, con la única restricción de que los únicos métodos que podemos invocar sobre ellos son los contenidos en la interfaz. A los atributos, parámetros y variables de un tipo definido por una interfaz, únicamente les podemos asignar objetos de una clase que implemente esa misma interfaz. Esto se ilustra en el ejemplo 1.

Ejemplo 1



Objetivo: Ilustrar el uso de referencias cuyo tipo es una interfaz.

En este ejemplo presentamos algunos métodos que podríamos implementar en la clase `Dibujo`, usando la declaración anterior de la interfaz `IFigura`. Estos métodos sólo se presentan como ilustración, ya que más adelante mostraremos una mejor implementación de la clase.

```

public class Dibujo
{
    private IFigura[] figuras;

    public void inicializacion( IFigura fig )
    {
        figura[ 0 ] = fig;
        figura[ 1 ] = new Linea( ... );
    }
}
  
```



Declaramos un arreglo de objetos cuyo tipo es la interfaz `IFigura`. En él podremos almacenar objetos de cualquiera de las clases que cumplan el contrato.



Este método ilustra algunas posibles inicializaciones para el arreglo de figuras. Mientras no tengamos los constructores de las clases, no podemos completar este método (por ahora usamos puntos suspensivos para indicar que no están

```

figura[ 2 ] = new Rectangulo( ... );
figura[ 3 ] = new Ovalo( ... );
}

```

```

public void dibujar( Graphics2D g )
{
    for( int i = 0; i < figuras.length; i++ )
    {
        figuras[ i ].dibujar( g, false );
    }
}

```

```

public boolean selecciono( int x, int y )
{
    for( int i = 0; i < figuras.length; i++ )
    {
        if( figuras[ i ].estaDentro( x, y ) )
            return true;
    }

    return false;
}

```

completas las llamadas). Hay que evitar este tipo de métodos en los cuales hay una dependencia hacia clases concretas, puesto que va a comprometer la evolución del programa. Aquí sólo lo usamos para ilustrar el uso de las interfaces.

 Fíjese que las referencias a `IFigura` se pueden pasar como parámetro y asignar.

 Este método permite dibujar en la pantalla todas las figuras presentes. La clase `Graphics2D` representa la zona de dibujo y será estudiada más adelante.

 Nos contentamos con pedirle a cada objeto del arreglo que invoque su método de dibujo. Cada uno utilizará su propia implementación para hacerlo.

 Fíjese que este método es completamente independiente de las figuras que maneje el editor y no requiere ningún cambio si aparece un nuevo tipo de figura.

 Este método recibe como parámetro las coordenadas de la ventana en donde el usuario hizo clic y retorna verdadero si seleccionó una de las figuras presentes, o falso en caso contrario.

 Este método le pregunta a cada una de las figuras que contiene el dibujo si las coordenadas del clic están en su interior.

 Cada figura utiliza su propia implementación del método `estaDentro()` cuando recibe la llamada.

 Este método también es completamente independiente del conjunto de figuras que soporte el editor.

3.5. Construcción de una Clase que Implementa una Interfaz

Cuando queramos construir una clase que implemente una interfaz, debemos hacer dos cosas. Por un lado, declarar en el encabezado de la clase que

vamos a respetar esa interfaz, y, por otro, incluir un método en la clase por cada método de la interfaz, usando la misma signatura. La sintaxis se ilustra en la siguiente tabla. El contenido exacto de esas clases (en el contexto del caso de estudio) es tema de una sección posterior:

```
public class Linea implements IFigura
{
    ...
}
```

Con la palabra "implements" informamos al compilador nuestra intención de respetar la interfaz IFigura.

```
public class Rectangulo implements IFigura
{
    ...
}
```

En la clase Linea debemos implementar los ocho métodos de la interfaz IFigura. Si no lo hacemos, el compilador genera un mensaje de error.

```
public class Ovalo implements IFigura
{
    ...
}
```

La clase Rectangulo, al igual que todas las clases con figuras del editor, deben cumplir con el contrato funcional definido en la interfaz IFigura.

No es suficiente con implementar los métodos de la interfaz, sino que es indispensable que en el encabezado se declare el compromiso de hacerlo.

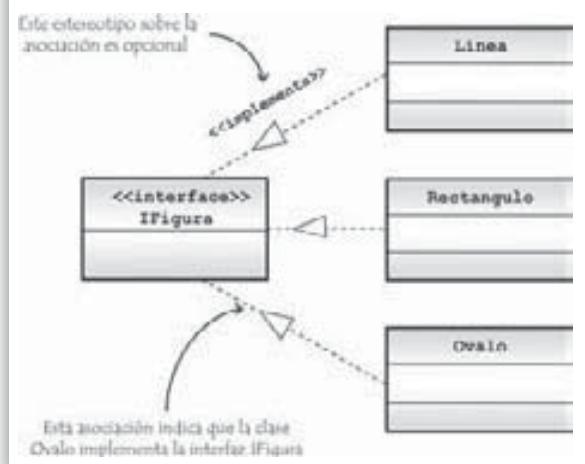


Si en una clase que dice que va a implementar una interfaz no se incluyen todos los métodos, se obtiene, por ejemplo, el siguiente error del compilador:

The type Linea must implement the inherited abstract method
IFigura.estaDentro(int, int)

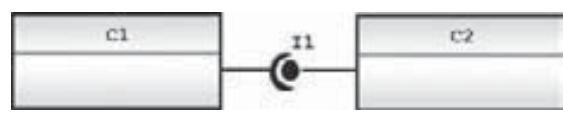
En UML las interfaces se muestran con el estereotipo <<interface>>, y se usa una asociación especial para indicar que una clase implementa una interfaz dada, tal como se muestra en la figura 4.6, para el caso del editor de figuras.

Fig. 4.6 – **Sintaxis para expresar que una clase implementa una interfaz**



Una sintaxis usada con frecuencia para representar la conexión entre dos clases a través de una interfaz es la que se muestra en la figura 4.7. Allí se hace explícito que la clase c1 tiene una asociación hacia cualquier implementación de la interfaz I1, y que la clase c2 satisface el contrato funcional exigido por dicha interfaz. Ésta es la sintaxis usada en los modelos de componentes de software, tema de un curso posterior, los cuales siempre se encuentran aislados por medio de una interfaz.

Fig. 4.7 – **Otra sintaxis para mostrar la conexión entre clases pasando por una interfaz**





Con la definición de la interfaz `IFigura` ya logramos un primer nivel de desacoplamiento entre las clases del programa, lo que nos va a permitir más adelante agregar nuevas figuras al editor con modificaciones mínimas.

necesidad de restringirse a una implementación concreta. La clase `ArrayList`, que hemos usado hasta este momento para manejar grupos de instancias, implementa estas dos interfaces, tal como se muestra en la figura 4.8.

Lo interesante es que Java dispone de otras clases que satisfacen los mismos contratos funcionales y que pueden remplazar en cualquier momento un `ArrayList`, a condición de escribir los métodos y las declaraciones en términos de las respectivas interfaces. En la figura 4.9 mostramos otras dos clases de Java (`Vector` y `LinkedList`) que implementan las mismas interfaces.

3.6. Interfaces para Contenedoras

Si avanzamos en la solución del caso de estudio, la siguiente etapa que nos encontramos es el diseño de la estructura contenedora de figuras. Por ahora contamos con tres grupos de opciones que ya hemos estudiado en niveles anteriores: arreglos, vectores o estructuras enlazadas. Cualquiera de esas opciones sería válida, pero ¿no sería mejor volver a aplicar la misma idea de las interfaces para obtener un nuevo punto de desacoplamiento? ¿Qué nos impide definir en términos de un contrato el funcionamiento de los grupos de objetos, y escribir nuestra clase `Dibujo` usando los métodos de dicha interfaz? Eso tendría la ventaja de que no dependeríamos de una implementación concreta de la contenedora, lo cual facilitaría la evolución del programa.

En Java ya existen dos interfaces llamadas `Collection` y `List`, que permiten a un programador manipular grupos de objetos sin

Fig. 4.8 – Relación entre la clase `ArrayList` y las interfaces `Collection` y `List`

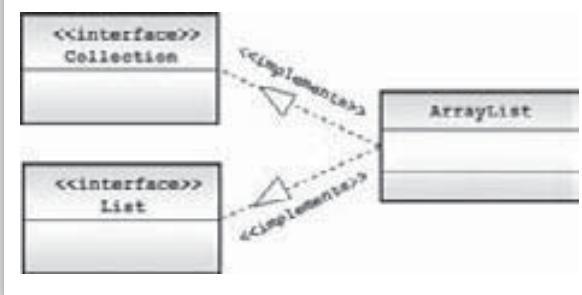
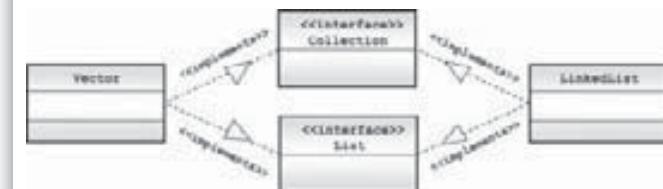


Fig. 4.9 – Contratos funcionales de las clases `Vector` y `LinkedList`



Esto quiere decir que si, por ejemplo, existe un método que espera como parámetro una referencia a un objeto que cumpla la interfaz `Collection`, podemos pasarle una instancia de la clase `ArrayList`, una instancia de la clase `Vector` o una instancia de la clase `LinkedList`, y dicho método funcionará correctamente.

Veamos entonces los principales métodos incluidos en estas dos interfaces. En la interfaz `Collection` se encuentran los métodos que permiten manejar grupos de elementos sin ningún orden o posición específica: básicamente podemos agregar un elemento, eliminarlo y preguntar si el elemento se encuentra en la colección.

```

public interface Collection
{
    public boolean add( Object o );

    public boolean addAll( Collection c );

    public void clear();

    public boolean contains( Object o );

    public boolean isEmpty();

    public Iterator iterator();

    public boolean remove( Object o );

    public int size();
}

```

- Agrega a la colección el objeto que recibe como parámetro. Retorna verdadero si la operación fue exitosa.
- Agrega a la colección todos los objetos de la colección que recibe como parámetro. Retorna verdadero si la operación fue exitosa.
- Elimina todos los elementos de la colección, dejándola vacía.
- Retorna verdadero si la colección tiene al menos un elemento “e” que cumple que e.equals(o) es verdadero.
- Retorna verdadero si la colección no tiene ningún elemento.
- Retorna un iterador sobre la colección. Esto es tema de la siguiente sección.
- Elimina la referencia que la colección tiene hacia un elemento “e”, que cumple que e.equals(o) es verdadero. El método retorna verdadero si dicho elemento fue localizado y eliminado.
- Retorna el número de elementos presentes en la colección.

La interfaz `List` contiene los métodos necesarios para manejar secuencias de valores, en los cuales cada elemento tiene una posición. Es más general que la interfaz `Collection` e incluye todos los métodos de esta primera interfaz, concretando en algunos casos su

comportamiento. Por ejemplo, para el método `add()`, que agrega un elemento a la colección, esta interfaz especifica que lo debe añadir como último elemento de la lista. Veamos los métodos de la interfaz `List` que no son compartidos con la interfaz `Collection`.

```

public interface List
{
    public void add( int i, Object o );

    public Object get( int i );

    public int indexOf( Object o );
}

```

- Inserta el objeto que recibe como parámetro en la posición “i” de la lista, desplazando los elementos ya presentes.
- Retorna la referencia al objeto que se encuentra en la posición “i” de la lista.
- Retorna el índice del primer elemento “e” de la lista, que cumple que e.equals(o) es verdadero. Si no encuentra ninguno retorna el valor -1.

```

public Object remove( int i );

}

public Object set( int i, Object o );
}

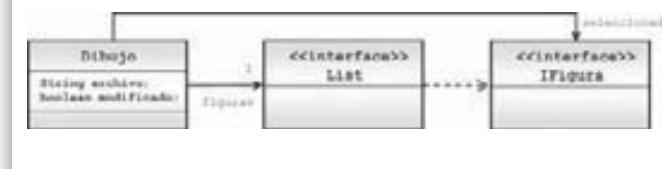
```

■ Elimina la referencia al objeto que se encuentra en la posición “i” de la lista y retorna la referencia que allí se encontraba.

■ Reemplaza el elemento que se encontraba en la posición “i” por el objeto que llega como parámetro. Retorna el elemento que antes se encontraba en esa posición.

Regresando a nuestro editor de dibujos, para mejorar el nivel de desacoplamiento vamos a utilizar en el diseño cualquier contenedora que cumpla con el contrato funcional definido por la interfaz `List`. Esto nos lleva al diagrama de clases que aparece en la figura 4.10. Allí incluimos también tres elementos importantes en el modelado de la entidad `Dibujo`: una asociación hacia la interfaz `IFigura` para señalar la figura seleccionada, un atributo de tipo lógico para indicar si el dibujo ha sido modificado y un atributo de tipo cadena de caracteres para almacenar el nombre del archivo del cual fue leído el dibujo. Ya estamos entonces listos para implementar la clase `Dibujo`, sin haber generado dependencias directas hacia otras clases del diseño.

Fig. 4.10 – **Diagrama de clases de diseño, para la clase Dibujo**



En el ejemplo 2 aparecen las declaraciones, el constructor y el invariante de la clase `Dibujo`. La implementación de los demás métodos la presentaremos en la siguiente sección.

Ejemplo 2



Objetivo: Presentar la declaración de una clase, definida en términos de interfaces.

En este ejemplo mostramos la declaración de la clase `Dibujo`, al igual que el constructor y el método que verifica el invariante.

```

package uniandes.cupi2.paint.mundo;

import java.util.List;

public class Dibujo
{
    // -----
    // Atributos
    // -----

    private List figuras;
    private IFigura seleccionada;
    private String archivo;
    private boolean modificado;
}

```

- La interfaz `List` (al igual que la interfaz `Collection`) se encuentran en el paquete `java.util`.
- El primer atributo es una lista con las figuras que hacen parte del dibujo, respetando el orden en el que fueron agregadas en el editor. Todas las figuras implementan la interfaz `IFigura`.
- El segundo atributo contiene la figura que está seleccionada actualmente en el dibujo (si la hay).
- El tercer atributo es el nombre del archivo donde se está guardando actualmente el dibujo.
- El último atributo indica si el dibujo ha sido modificado.

```

public Dibujo( )
{
    figuras = new ArrayList( );

    seleccionada = null;
    archivo = null;
    modificado = false;

    verificarInvariante( );
}

private void verificarInvariante( )
{
    assert ( figuras != null ) : "Lista nula";
    if( seleccionada != null )
    {
        for( int i = 0; i < figuras.size( ); i++ )
        {
            IFigura f = ( IFigura )figuras.get( i );
            if( seleccionada == f )
                return;
        }
        assert ( true ) : "Selección inválida ";
    }
}

```

En el constructor de la clase creamos una instancia de la clase `ArrayList` para manejar la lista de figuras. Si decidimos cambiar de estructura contenedora, basta con modificar una línea de toda la clase.

Luego, inicializamos los demás atributos de la clase y verificamos que se cumpla el invariante.

El invariante debe verificar que la lista se encuentre inicializada y que la eventual figura seleccionada haga parte de las figuras del dibujo.

Para la segunda condición hacemos un ciclo buscando la figura seleccionada y retornando tan pronto la encontramos. Si llega al final del ciclo quiere decir que la figura no hace parte de las que se encuentran incluidas en el dibujo.

Para el ciclo utilizamos los métodos `size()` y `get()` de la interfaz `List`.



Con este diseño obtenemos un segundo punto de desacoplamiento: la clase `Dibujo` ahora no depende ni de la implementación de la estructura contenedora, ni de las implementaciones de los distintos tipos de figuras que definamos. Eso nos permitirá hacer evolucionar el editor de manera mucho más sencilla, incorporando nuevas figuras y requerimientos funcionales. En caso de necesidad, podríamos incluso hacer nuestra propia implementación de la interfaz `List`, para cumplir, por ejemplo, con algún requerimiento no funcional como persistencia o seguridad.

3.7. Iteradores para Recorrer Secuencias

En algunos casos, y haciendo abstracción de la estructura exacta que estemos manejando, nuestro interés se

concentra en hacer un recorrido sobre los elementos contenidos en una estructura. Esto es, pasar una vez sobre cada uno de los objetos presentes. Para hacer esto existe una interfaz en Java, llamada `Iterator`, la cual cuenta con los siguientes métodos:

```

public interface Iterator
{
    public boolean hasNext( );

    public Object next( );
}

```

Retorna verdadero si en el iterador quedan todavía elementos por recorrer y falso en caso contrario.

Retorna el siguiente objeto contenido en el iterador.

Tanto la interfaz `Collection` como la interfaz `List` tienen un método que les permite retornar un iterador sobre la estructura contenedora. En los si-

guientes ejemplos mostramos la manera de utilizar los iteradores para hacer recorridos sobre la lista de figuras de nuestro editor.

Ejemplo 3



Objetivo: Mostrar la manera de utilizar un iterador para recorrer los elementos de una estructura contenedora.

En este ejemplo mostramos un método de la clase `Dibujo`, cuya solución implica un recorrido por la lista de figuras.

```
public class Dibujo
{
    private List figuras;
    private IFigura seleccionada;
    private String archivo;
    private boolean modificado;

    public void dibujar( Graphics2D g )
    {
        Iterator iter = figuras.iterator( );
        while( iter.hasNext( ) )
        {
            IFigura f = ( IFigura )iter.next( );
            f.dibujar( g, f == seleccionada );
        }
    }
}
```

■ En la lista, las figuras se van agregando al final de la estructura a medida que van llegando. Por esta razón se deben dibujar en ese orden, para que las últimas figuras den la impresión de estar sobre las primeras figuras.

■ El método que dibuja las figuras comienza pidiendo un iterador a la lista de figuras y asignándolo a la variable “iter”.

■ El ciclo se repite mientras haya elementos en el iterador que no hayan sido recorridos.

■ Con el método `next()` obtenemos el siguiente objeto del iterador e indicamos que debe implementar la interfaz `IFigura`.

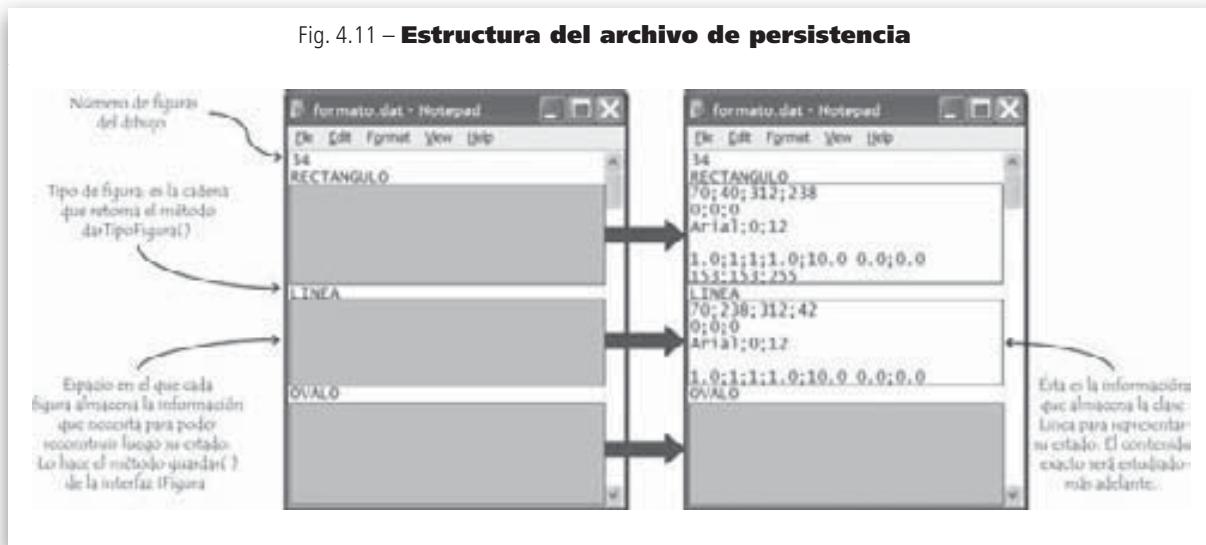
■ Finalmente llamamos el método de dibujo de la interfaz `IFigura`, cuyo primer parámetro es la superficie gráfica de dibujo y cuyo segundo parámetro es un valor lógico que indica si la figura está seleccionada.

■ Este método no depende de la estructura contenedora que utilicemos, ni de las figuras que se incluyan dentro del editor.

Antes de implementar el método que escribe el dibujo en memoria secundaria (el cual utiliza también un iterador), vamos a definir la estructura, contenido y convenciones que va a seguir este archivo. Sabemos que debe ser un archivo secuencial de texto, que pueda ser editado fácilmente. Aunque todavía no sabemos la información que debe almacenar cada una

de las figuras, sabemos que todas ellas deben implementar el método `guardar(PrintWriter)`. También tenemos en esa misma interfaz el método `darTipoFigura()`, que retorna una cadena de caracteres con el nombre de la figura. En la figura 4.11 aparece el esqueleto de la estructura del archivo que vamos a utilizar.

Fig. 4.11 – Estructura del archivo de persistencia



En el ejemplo 4 aparece la implementación del método que salva el contenido de un dibujo en un archivo, utilizando la estructura anterior.



Objetivo: Mostrar la manera de utilizar un iterador para recorrer los elementos de una estructura contenedora.

En este ejemplo mostramos el método de la clase `Dibujo`, que utiliza un iterador para salvar su estado en un archivo.

```
public class Dibujo
{
    private List figuras;
    private String archivo;
    private boolean modificado;

    public void salvar() throws IOException
    {
        PrintWriter out = new PrintWriter( archivo );
        out.println( figuras.size() );

        Iterator iter = figuras.iterator();

        while( iter.hasNext() )
        {
            IFigura f = ( IFigura )iter.next();

            out.println( f.darTipoFigura() );
            f.guardar( out );
        }
        out.close();

        modificado = false;
    }
}
```

- El método lanza la excepción `IOException` si hay un error en el disco duro en el momento de escribir el archivo.
- Lo primero que hace el método es abrir un flujo de escritura, asociado con el archivo en el cual se quiere salvar la información.
- Luego, escribe el número de figuras presentes en el dibujo, siguiendo la convención antes definida.
- Pide después un iterador sobre la lista de figuras y a cada una de ellas le pide su tipo para escribirlo en el archivo, y luego le delega la responsabilidad de escribir en el archivo la información que necesite para recuperar después su estado. Fíjese que el flujo pasa como parámetro a cada figura para que escriba allí su información.
- Finalmente, el método cierra el flujo de escritura e indica que el dibujo actual no ha sido modificado.

3.8. Implementación del Editor de Dibujos

En esta sección vamos a plantear como tarea la construcción de los distintos métodos que necesita la clase

Dibujo, dejando pendiente únicamente el método que lee un dibujo de un archivo, puesto que depende de los métodos constructores de las clases que implementan las figuras, los cuales no hemos definido todavía.

Tarea 2



Objetivo: Construir algunos métodos de la clase Dibujo, usando un diseño basado en interfaces.

Desarrolle los métodos que se plantean a continuación. Siga las indicaciones que se dan en cada uno de los casos.

```
public void agregarFigura( IFigura f )
{
```

Agrega una nueva figura al dibujo, al final de la lista de figuras.

```
}
```

```
public void reiniciar( )
{
```

Reinicia el dibujo, eliminando todas las figuras.

```
}
```

```
public String darNombreArchivo( )
{
```

Retorna el nombre del archivo donde se está guardando la información de este dibujo.

```
}
```

```
public void hacerClick( int x, int y )
{
```

Selecciona la última figura agregada al dibujo que se encuentra en el punto (x, y). Si no hay ninguna figura en esas coordenadas, no deja ninguna figura como seleccionada.

Este método debe hacer un recorrido de atrás hacia adelante de la lista de figuras.

```
}
```

```
public IFigura darSeleccionada( )
{
```

Retorna la figura que se encuentra seleccionada o null si no hay ninguna.

```
}
```

```
public void eliminarFigura( )
{
```

 Elimina la figura seleccionada. Si no hay ninguna figura seleccionada, este método no hace nada.

Utiliza el método remove(o) de la interfaz List.

```
public boolean haSidoModificado( )
{
```

 Indica si el dibujo ha sido modificado.

```
public void traerAdelante( )
{
```

 Los siguientes métodos no corresponden a requerimientos funcionales, sino a extensiones del editor.

 Deja la figura seleccionada como la última de la lista de figuras, de manera que se vea sobre las demás figuras del dibujo.

```
}
```

```
public int contarLineas( )
{
```

 Cuenta y retorna el número de líneas que hay en el dibujo. Utilice el método darTipoFigura() de la interfaz IFigura.

```
}
```

```
public void seleccionarSiguiente( )
{
```

 Selecciona la siguiente figura del dibujo, de acuerdo con el orden de inserción. Si no hay ninguna figura seleccionada, este método no hace nada.

```
}
```

```
public int contarTextos( )
{
```

 Cuenta y retorna el número de figuras del dibujo que tienen un texto asociado.

```
}
```

3.9. Otras Interfaces Usadas Anteriormente

Hay tres interfaces que hemos utilizado anteriormente

sin hacerlo explícito. En esta sección, antes de continuar con la solución de nuestro caso de estudio, vamos a hacer un breve recorrido por ellas, para mostrar sus métodos y su uso.

- La interfaz `ActionListener`:

Paquete: <code>java.awt.event</code> Métodos: <code>void actionPerformed(ActionEvent e)</code>	Uso: <ul style="list-style-type: none"> ■ Los paneles activos de la interfaz de usuario deben implementar esta interfaz, cuando contienen botones u otros componentes gráficos que generan este tipo de eventos. ■ El método <code>actionPerformed()</code> será invocado cada vez que un evento ocurra, recibiendo como parámetro un objeto de la clase <code>ActionEvent</code> con la información necesaria para que el panel pueda decidir la acción que debe seguir.
---	--

- La interfaz `ListSelectionListener`:

Paquete: <code>javax.swing.event</code> Métodos: <code>void valueChanged(ListSelectionEvent e)</code>	Uso: <ul style="list-style-type: none"> ■ Los paneles activos de la interfaz de usuario deben implementar esta interfaz cuando tienen un componente gráfico de la clase <code>JList</code> y quieren reaccionar a un cambio en la selección del usuario. ■ El método <code>valueChanged()</code> será invocado cada vez que cambie el elemento seleccionado por el usuario sobre la lista, recibiendo como parámetro un objeto de la clase <code>ListSelectionEvent</code> con la información necesaria para que el panel pueda reaccionar.
--	--

- La interfaz `Serializable`:

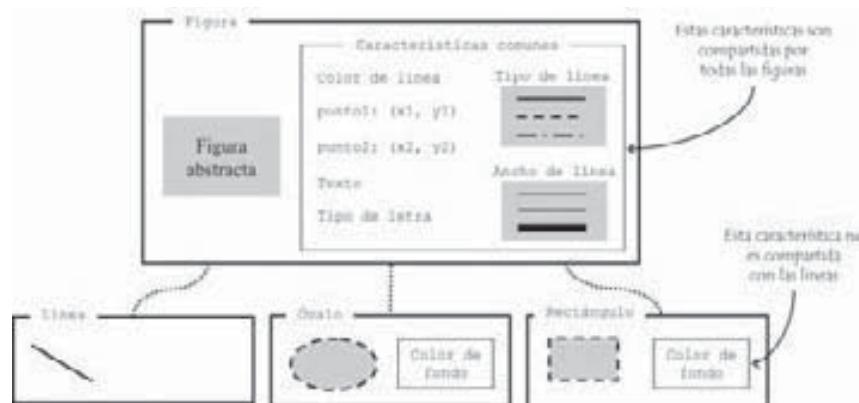
Paquete: <code>java.io</code> Métodos: <code>ninguno</code>	Uso: <ul style="list-style-type: none"> ■ Ésta es una interfaz muy particular de Java, puesto que no exige que se desarrolle ningún método en particular en la clase que la implementa. ■ La declaración la toma el compilador como una "autorización" para que las instancias de la clase puedan ser serializadas. ■ Esto se hace por seguridad, puesto que siempre hay un riesgo de manipulación del estado del objeto cuando éste se encuentra en un archivo, o ha sido enviado a otro computador.
--	---

3.10. La Herencia como Mecanismo de Reutilización

Vamos a concentrarnos ahora en la implementación de las distintas figuras que hacen parte del editor y a crear una estructura de base que nos permita crear otras nuevas a bajo costo. Lo primero que debemos resaltar es que todas las figuras, además de

tener que implementar la interfaz `IFigura`, tienen una estructura común, la cual hacemos explícita en la figura 4.12. Allí se puede apreciar que todas las figuras comparten ciertas características, y que nos podemos imaginar una especie de jerarquía o taxonomía de figuras, en donde se clasifican las figuras por los elementos que comparten.

Fig. 4.12 – Características comunes de las figuras



Intentemos implementar la clase `Figura` que se sugiere en la jerarquía anterior, de manera que incluya todos los elementos (atributos y métodos) que comparten las figuras del editor. El solo hecho de implementar (y probar) una sola vez todo lo que tienen en común las tres figuras de nuestro editor ya representa una ganancia

para nosotros. Aprovechamos también para identificar los elementos que no comparten, ya sea porque requieren más atributos (como es el caso del color del fondo para los óvalos y los rectángulos), o porque la implementación de los métodos debe ser diferente (el método de dibujar, por ejemplo, no puede ser compartido).



Objetivo: Implementar una parte de la clase `Figura`, de manera que contenga los elementos compartidos por las tres figuras de nuestro editor.

En este ejemplo mostramos y explicamos las declaraciones de los atributos comunes de las figuras, e implementamos algunos de los métodos que comparten. Hay métodos que no podemos desarrollar hasta haber estudiado las clases de Java que permiten dibujar figuras geométricas, de manera que los veremos en secciones posteriores.

```
public abstract class Figura implements IFigura
{
    // -----
    // Atributos
    // -----
}
```

En el encabezado debemos indicar que es sólo una implementación parcial y para eso usamos la palabra "abstract".

También indicamos que nos comprometemos con el contrato funcional descrito por `IFigura`.

<pre>private int x1; private int y1; private int x2; private int y2;</pre>	<p> Este grupo de atributos representa los dos puntos que definen la figura: (x1, y1) y (x2, y2). Debe ser claro que cada una de las figuras los puede interpretar de un modo diferente.</p>
<pre>private String texto; private Font tipoLetra;</pre>	<p> Estos dos atributos representan el texto asociado con la figura y el tipo de letra en el cual éste se debe presentar en el dibujo. La clase Font la provee Java para manejar esto (la estudiaremos más adelante en este nivel).</p>
<pre>private Color colorLinea;</pre>	<p> Este atributo representa el color de la línea. En el caso del rectángulo y el óvalo, es el color de la línea que marca el perímetro.</p>
<pre>private BasicStroke tipoLinea;</pre>	<p> Este atributo describe el ancho y el tipo de la línea. La clase BasicStroke hace parte de Java y la veremos en una sección posterior.</p>
<pre>public Figura(int x1f, int y1f, int x2f, int y2f, Color colorLineaF, BasicStroke tipoLineaF) { x1 = x1f; x2 = x2f; y1 = y1f; y2 = y2f; colorLinea = colorLineaF; tipoLinea = tipoLineaF; texto = ""; tipoLetra = new Font("Arial",Font.PLAIN, 12); verificarInvariantes(); }</pre>	<p> Éste es el primer constructor de la clase. Permite construir una figura si le dan un valor para cada uno de los atributos, con excepción del texto, que comienza en vacío por defecto y con un tipo de letra estándar ("Arial 12").</p> <p> Al finalizar el constructor, verificamos, como de costumbre, que se cumpla el invariante de la clase.</p>
<pre>public Figura(BufferedReader br) throws IOException, FormatoInvalidoException { ... }</pre>	<p> Éste es el segundo constructor de la clase. Permite construir una figura a partir de un flujo de entrada conectado a un archivo. El contenido exacto de este método lo veremos más adelante.</p>
<pre>public abstract void dibujar(Graphics2D g, boolean seleccionada); public abstract boolean estaDentro(int x, int y); public abstract String darTipoFigura();</pre>	<p> Estos tres métodos de la interfaz IFigura no se pueden implementar aquí, puesto que no tienen sentido a menos que se concrete la figura que se quiere representar. Por esta razón los métodos se declaran como abstractos y no se les define un cuerpo.</p> <p> Es obligatorio poner esta declaración, para asegurarle al compilador que sí vamos a cumplir el contrato funcional descrito por la interfaz, aunque no sea directamente en esta clase.</p>

```

public String darTexto( )
{
    return texto;
}

public void cambiarTexto( String txt )
{
    texto = txt;
}

public Font darTipoLetra( )
{
    return tipoLetra;
}

public void cambiarTipoLetra( Font fuenteTexto )
{
    tipoLetra = fuenteTexto;
}

private void verificarInvariantes( )
{
    assert colorLinea != null : "Atributo inválido";
    assert tipoLinea != null : "Atributo inválido";
    assert tipoLetra != null : "Atributo inválido";

    assert x1 >= 0 : "Coordenada x1 inválida";
    assert x2 >= 0 : "Coordenada x2 inválida";
    assert y1 >= 0 : "Coordenada y1 inválida";
    assert y2 >= 0 : "Coordenada y2 inválida";
}

```

Retorna el texto asociado con la figura.

Cambia el texto asociado con la figura.

Retorna el tipo de letra asociado con el texto de la figura.

Cambia el tipo de letra asociado con el texto de la figura.

Este método verifica que el invariante de la clase se cumpla.

Lo primero que valida es que el color de la línea, el tipo de línea y el tipo de letra tengan algún valor distinto de nulo (revisa que los respectivos objetos hayan sido creados).

Después verifica que las coordenadas de los dos puntos sean valores superiores o iguales a cero, puesto que vamos a trabajar únicamente sobre ese espacio de coordenadas.



Una **clase abstracta** es una implementación parcial, que sólo puede ser utilizada como base para construir otras clases de manera más eficiente. Esto quiere decir que no se pueden crear instancias de una clase abstracta, así tenga definido un método constructor. En Java se utiliza la palabra **abstract** para declarar este tipo de clases.



Un **método abstracto** es un método definido en una clase abstracta, para el cual no existe una implementación. Más adelante, las clases que se construyan con base en esta clase abstracta serán responsables de implementar estos métodos. En Java estos métodos no tienen cuerpo y se utiliza la palabra **abstract** como parte de su firma.

3.10.1. Superclases y Subclases

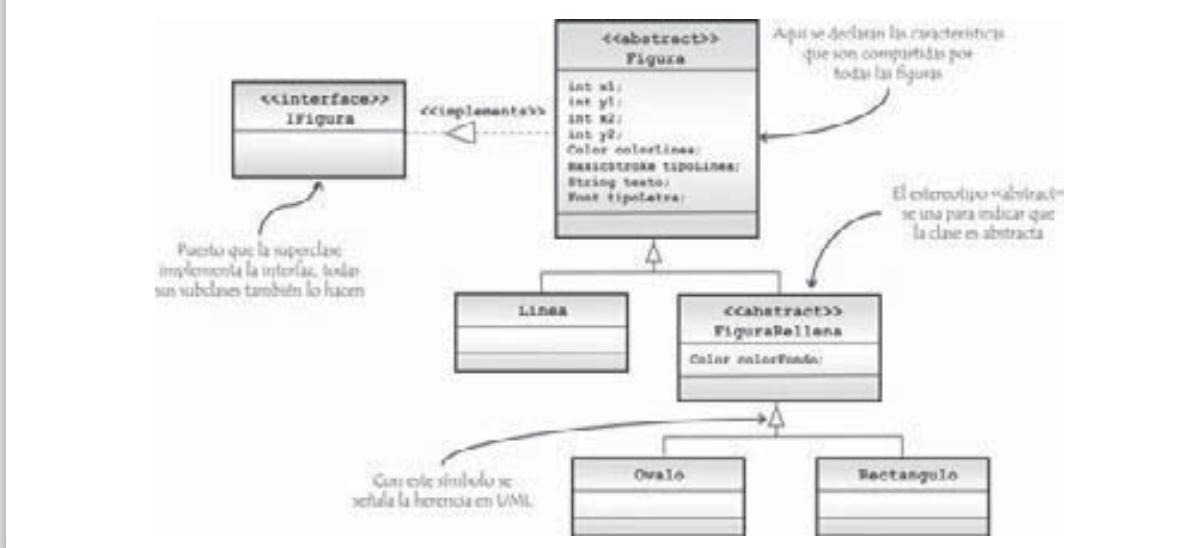
Ahora que ya tenemos una parte de la implementación de la clase **Figura**, ¿hay alguna manera de construir sobre ella las clases que necesitamos para nuestro editor? La respuesta es sí y el mecanismo que lo permite se denomina **herencia**.

La clase sobre la cual nos queremos basar para construir otra clase se denomina la **superclase** y la nueva clase se denomina la **subclase**. En Java se utiliza la sintaxis que se muestra en los siguientes fragmentos de código:

<pre>public class Linea extends Figura { ... }</pre>	<ul style="list-style-type: none"> Para indicar que la clase Linea hereda (especializa o extiende) de la clase Figura, utilizamos la palabra "extends".
<pre>public abstract class FiguraRellena extends Figura { // ----- // Atributos // ----- private Color colorFondo; ... }</pre>	<ul style="list-style-type: none"> El compilador localiza el archivo en el que está declarada la superclase y considera todas las declaraciones que hagamos en la subclase como extensiones de la primera.
<pre>public class Ovalo extends FiguraRellena { ... }</pre>	<ul style="list-style-type: none"> Definimos una segunda clase abstracta, que hereda de la clase Figura, con las características comunes que tienen las figuras ovaladas y los rectángulos.
<pre>public class Rectangulo extends FiguraRellena { ... }</pre>	<ul style="list-style-type: none"> Sigue siendo abstracta, porque hay métodos que no tienen una implementación. Declaramos como atributo la característica adicional que tiene este tipo de figuras.

En el diagrama de clases de UML se utiliza la sintaxis mostrada en la figura 4.13 para indicar que una clase hereda de otra. Allí se puede apreciar también que se utiliza el estereotipo <> para indicar que una clase es abstracta.

Fig. 4.13 – Representación de la herencia en el diagrama de clases de UML



Puesto que las subclases (`Línea`, `Ovalo` y `Rectángulo`) cumplen también los contratos funcionales (`IFigura`) a los que se comprometió la superclase (`Figura`), en nuestro editor podemos manipular las figuras como

referencias a esa interfaz sin ningún problema. Con este diseño vamos a poder incluir nuevas figuras en el editor con un esfuerzo mínimo, puesto que cada nueva figura va a **reutilizar** la estructura de base que ya le construimos.



Cuando una clase extiende de otra, hereda todos los atributos y métodos de la superclase. De esta forma, la subclase sólo tiene que incluir los nuevos atributos, los nuevos métodos, una implementación para los métodos abstractos de la superclase y los métodos de la superclase para los cuales quiera cambiar la implementación. Para esto último, basta con utilizar la misma firma del método heredado para que el compilador entienda que la nueva clase quiere modificar la implementación definida en la superclase.

Desde el punto de vista de la clase que utiliza una subclase, no hay ninguna diferencia entre los métodos que ésta hereda y los métodos que implementa. Hacia afue-

ra una subclase ofrece como suyos todos los métodos propios junto con los métodos heredados. Todo lo anterior se ilustra en el ejemplo 6.

Ejemplo 6



Objetivo: Ilustrar el mecanismo de herencia.

En este ejemplo ilustramos en un contexto simplificado el funcionamiento del mecanismo de herencia. Estudie las clases que se presentan a continuación y el comportamiento que se describe.

```
public abstract class C1
{
    private int atr1;

    public C1( )
    {
        atr1 = 5;
    }

    public abstract int m1( );

    public void cambiar( int valor )
    {
        atr1 = valor;
    }
}
```

■ La clase `C1` es abstracta. Tiene un único atributo (`atr1`) y un constructor sin parámetros que inicializa el atributo en el valor 5.

■ Por ser una clase abstracta, no es posible crear instancias de ella (a pesar de tener un método constructor implementado).

■ Esta clase únicamente puede ser utilizada para construir subclases a partir de ella.

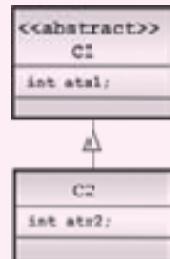
■ Cuenta también con dos métodos: uno abstracto (`m1`), sin parámetros, y otro implementado (`cambiar`), que permite modificar el valor del atributo.

```
public class C2 extends C1
{
    private int atr2;

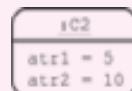
    public C2( )
    {
        atr2 = 10;
    }

    public int m1( )
    {
        return atr2;
    }
}
```

- La clase C2 es una subclase de la clase C1. No es una clase abstracta, porque implementa los métodos abstractos heredados. Es posible entonces crear instancias de esta clase.



- El siguiente es el diagrama de objetos de una instancia de C2, tan pronto ha sido creada. Fíjese que el objeto tiene dos atributos y que para su construcción debió ser llamado primero el constructor de la superclase (de manera implícita) y, luego, el constructor de la subclase:



- Note que al crearse la instancia, lo que se hace es reunir en un solo objeto los atributos de la superclase (atr1) y los de ella misma (atr2).

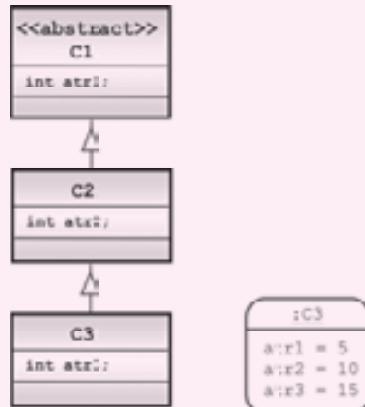
```
public class C3 extends C2
{
    private int atr3;

    public C3( )
    {
        atr3 = 15;
    }

    public void cambiar( int valor )
    {
        atr3 = valor * 3;
    }

    public String darCadena( )
    {
        return "valor=" + atr3;
    }
}
```

- La clase C3 hereda de la clase C2, luego tenemos los siguientes diagramas de clases y de objetos:



- Cada instancia de la clase C3 tiene tres atributos: dos heredados y uno propio.

```

public class C4
{
    private C2 obj2;
    private C3 obj3;

    public C4( )
    {
        obj2 = new C2( );
        obj3 = new C3( );

        obj2.cambiar( 20 );
        obj3.cambiar( 20 );

        String st = obj3.darCadena( );
    }
}

```

■ La clase C3 redefine el método cambiar(), puesto que tiene la misma firma del método con el mismo nombre de la clase C1. Esto quiere decir que las instancias de C3 contestarán a las invocaciones de este método de manera distinta a como lo van a hacer las instancias de C1 o de C2.

■ El método m1() es heredado de la clase C2. El método darCadena() es añadido por esta clase.

■ La clase C4 tiene un atributo de la clase C3 y un atributo de la clase C2. Puede invocar sobre ambas instancias los métodos m1() y cambiar(), pero con comportamientos distintos en algunos casos. Apenas ha sido creada, una instancia de C4 tendrá los siguientes valores:



■ Note que la instancia de C2 utiliza el método cambiar() implementado en la clase C1, mientras la instancia de C3 utiliza su propia implementación de este método.

■ Sólo sobre el objeto obj3 se puede invocar el método darCadena(), puesto que fue agregado por la clase C3.



En el momento de redefinir un método, se debe respetar exactamente la misma firma. Si se cambia el tipo de los parámetros, el compilador va a decidir que son dos métodos distintos y no redefine el de la superclase. Si la diferencia es sólo en el tipo de retorno, el compilador presenta un mensaje de error del siguiente estilo: The return type is incompatible with C1.cambiar(int)

Pasemos ahora a ver cómo se comportan las referencias a objetos cuando se maneja una jerarquía de herencia. Esto se ilustra con la siguiente secuencia de instruccio-

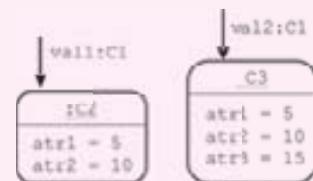
nes, cuyo significado se explica mediante diagramas de objetos. Vamos a utilizar las clases y la jerarquía de herencia definidas por el ejemplo anterior.

```
C1 val1 = null;
C1 val2 = null;
```

■ Es posible definir referencias a una clase aunque ésta sea abstracta. A estas referencias se les va a poder asignar un objeto de cualquiera de las subclases. Esta propiedad se denomina **polimorfismo**.

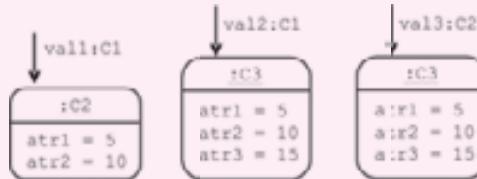
```
val1 = new C2( );
val2 = new C3( );
```

■ Extendemos la sintaxis gráfica utilizada hasta ahora, para hacer explícito el tipo de la referencia que apunta a un objeto:



```
C2 val3 = new C3();
```

- Las variables que hacen referencia a instancias de la clase C2 pueden señalar objetos de la clase C3, puesto que ésta es una de sus subclases:



```
val1.cambiar( 99 );
int aux = val2.m1();
```

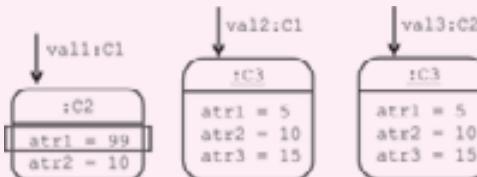
- A través de una referencia a una clase, sólo se pueden invocar los métodos que dicha clase tiene, aunque sean abstractos. Puesto que en la clase C1 se encuentran definidos los métodos `m1()` y `cambiar()`, es válido hacer estas dos llamadas.

- Durante la ejecución, para determinar el método exacto que debe invocar, Java utiliza la clase del objeto y no el tipo de la referencia a través de la cual se hace el llamado. Por esa razón se puede invocar el método `m1()`, que es abstracto en C1. Esta propiedad se denomina **asociación dinámica**.

- Para la primera llamada, va a utilizar el método `cambiar()` de la clase C2. Como allí no está redefinido, utiliza la implementación de la clase C1.

- Para la segunda invocación, va a utilizar el método `m1()` de la clase C3. Como en C3 no está redefinido, utiliza la implementación de este método que encuentra en C2.

- Siempre comienza a buscar el método sobre la clase real del objeto y no sobre la clase a la cual pertenece la referencia.



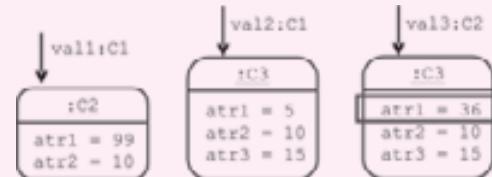
```
// Inválido
String s = val2.darCadena();
```

- Esta llamada es inválida. Aunque `val2` esté haciendo referencia a un objeto de la clase C3, que sí tiene el método `darCadena()` implementado, el compilador restringe las llamadas a los métodos de la clase a la cual pertenece la referencia. Como `val2` es una referencia de la clase C1 y el método `darCadena()` no está allí definido, no se acepta que se haga esta invocación.

```
val3.cambiar( 12 );
```



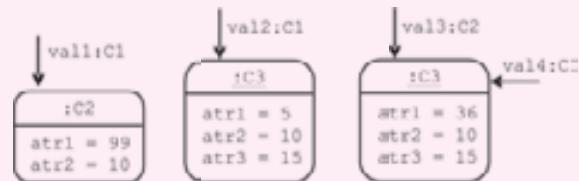
Esta llamada es válida. val3 es una referencia de la clase C2, pero se encuentra señalando a un objeto de la clase C3. Puesto que la clase C2 tiene ese método definido, la llamada se acepta, pero en ejecución se va a utilizar la implementación que de ese método tiene la clase C3, obteniendo el siguiente estado:



```
C1 val4 = val3;
```



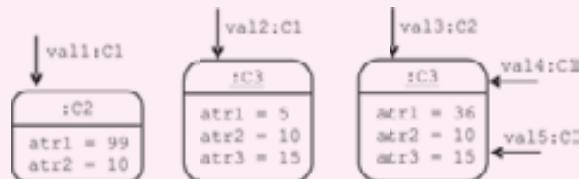
Es válido asignar a una variable de una superclase el valor que tiene una variable de una subclase. Al final, ambas referencias quedan apuntando al mismo objeto. En este caso asignamos a la variable val4 (clase C1) la referencia al objeto que está siendo señalado por la variable val3 (clase C2):



```
C3 val5 = ( C3 )val3;
String s = val5.darCadena( );
```



Para asignar una referencia a una variable de una subclase, es indispensable utilizar el operador de conversión (casting). En nuestro caso, esto es indispensable para poder invocar los métodos de la clase C3 que no están definidos en la clase C2.



Si al momento de aplicar el operador de conversión el computador se da cuenta de que el tipo del objeto no es compatible con el tipo de la referencia, lanza la siguiente excepción:

`java.lang.ClassCastException`



Una **subclase** es una versión especializada de una **superclase**, la cual hereda todos los atributos y métodos definidos en la superclase y añade los suyos propios.



El **polimorfismo** es una propiedad de los lenguajes orientados a objetos que permite a una variable tomar como valor un objeto de cualquiera de sus subclases. Es un mecanismo que facilita la reutilización de código. La **asociación dinámica** es otra propiedad muy importante de algunos lenguajes orientados a objetos (como Java), que garantiza que la decisión de cuál método invocar se toma siempre en ejecución (no durante la compilación), permitiendo así encontrar la implementación más adecuada para cada caso.

3.10.2. Elementos Privados y Elementos Protegidos

Antes de continuar desarrollando nuestro caso de estudio, vamos a detenernos un poco en el tipo de **visibilidad** que deben tener los atributos y métodos de una clase. Hasta ahora hemos definido por defecto que todos los atributos son privados y que todos los métodos

son públicos, pero por razones de eficiencia, en algunos casos podría ser conveniente que los métodos de una subclase tuvieran acceso directo a los atributos de la superclase, sin llegar al extremo de dejarlos públicos para todo el mundo. En esta sección presentamos un nuevo tipo de visibilidad llamada protegida (`protected`), en la cual logramos el punto intermedio mencionado anteriormente. Esto se ilustra en el ejemplo 7.



Ejemplo 7

Objetivo: Mostrar la visibilidad protegida para atributos y métodos de una superclase.

En este ejemplo mostramos, en un contexto simplificado, la manera como se manejan los atributos y métodos con visibilidad protegida.

```
public class C5
{
    private int atr1;
    protected int atr2;

    public C5( )
    {
        atr1 = 0;
        atr2 = 5;
    }

    public int darValor1( )
    {
        return atr1;
    }

    protected void servicio( )
    {
        atr2 = atr1 + 20;
    }
}
```

La clase C5 definida en este ejemplo tiene dos atributos. El primero tiene visibilidad privada, de manera que sólo puede ser consultado y modificado por los métodos de esta misma clase. El segundo tiene visibilidad protegida, lo cual indica que puede ser consultado y modificado tanto por los métodos de esta clase como por los métodos de cualquiera de sus subclases. Para las demás clases, este segundo atributo se considera privado.

Si cualquier clase, incluidas las subclases, quiere consultar el valor del atributo "atr1", debe hacer una llamada al método `darValor1()`, ya que este atributo está declarado como privado.

Puesto que el método `servicio()` es protegido, únicamente puede ser invocado desde las subclases de la clase C5. Este tipo de métodos se construyen así, de manera que faciliten el desarrollo de los métodos de las subclases, y no porque correspondan a una responsabilidad directa de la clase en la que están.

```
public class C6 extends C5
{
    public int m1( )
    {
        servicio();
        return atr2 + darValor1( );
    }
}
```

- La clase C6 es una subclase de C5 que no tiene atributos adicionales.
- En su único método invoca el método servicio() de la superclase, y retorna un valor utilizando directamente el atributo atr2 de la clase C5 y pasando por el método darValor1() para obtener el valor del otro atributo.
- Aquí no es posible hacer referencia de manera directa al atributo atr1 de la superclase.



Dependiendo de la visibilidad de un elemento (atributo o método) es posible restringir o permitir el acceso a ellos de la siguiente manera: (1) si es privado, el acceso está restringido a los métodos de la clase; (2) si es protegido, tienen acceso los métodos de la clase y los métodos de las subclases, o, (3) si es público, todos tienen acceso al elemento.

Teniendo en cuenta lo estudiado anteriormente, vamos a modificar la visibilidad de algunos de los atributos y métodos de las clases `Figura` y `FiguraRellena`,

tal como se presenta a continuación. Esto nos va a permitir el acceso directo desde las subclases a algunos atributos y al método que verifica el invariante.

```
public abstract class Figura implements IFigura
{
    // -----
    // Atributos
    // -----

    protected int x1;
    protected int y1;

    protected int x2;
    protected int y2;

    protected Color colorLinea;
    protected BasicStroke tipoLinea;

    private String texto;
    private Font tipoLetra;

    ...

    protected void verificarInvariante( )
    { ... }

    protected void dibujarTexto( Graphics2D g )
    { ... }
}
```

- Cambiamos la visibilidad de todos los atributos geométricos y gráficos de la clase, para facilitar la implementación en las subclases de los métodos de dibujo.
- Dejamos privados los atributos que manejan el texto asociado con la figura y aprovechamos los métodos de acceso y modificación que ya habíamos planteado anteriormente.
- Cambiamos la visibilidad del método que verifica el invariante, para que pueda ser llamado desde los métodos que verifican el invariante en las subclases.
- Agregamos un método de servicio (dibujarTexto), que será utilizado desde las subclases, el cual presenta en la pantalla el texto asociado con la figura en un punto intermedio entre los dos puntos que la definen. Es importante anotar que este método no debe ser público, puesto que no es una de las responsabilidades directas de la clase Figura (no resuelve completamente un requerimiento funcional, sino que puede ayudar a los métodos de las subclases resolviendo una parte del problema que se les plantea). Sólo pretendemos seguir simplificando el desarrollo de nuevas clases, construyendo métodos de apoyo. Esto no implica ninguna obligación de parte de los métodos de las subclases de utilizar estos métodos de servicio.

```

public abstract class FiguraRellena extends
Figura
{
    // -----
    // Atributos
    // -----

    protected Color colorFondo;

    ...

    protected void verificarInvariante( )
    { ... }
}

```

- Cambiamos la visibilidad del atributo que almacena el color de fondo de la figura rellena, con el mismo argumento que usamos en la clase Figura.
- Dejamos el método que calcula el invariante como protegido (usualmente lo definimos como privado), de manera que pueda ser invocado desde las subclases.



En una clase abstracta vamos a encontrar tres tipos de métodos: (1) los métodos abstractos, para los cuales no es imaginable una implementación; (2) los métodos finales, que ya son la solución que necesitan todas las subclases para asumir una cierta responsabilidad, y (3) los métodos de servicio (declarados como protegidos), que son un medio para facilitar el desarrollo de algunas partes de las subclases. En la etapa de diseño se deben identificar los métodos de estos tres grupos.

3.10.3. Acceso a Métodos de la Superclase

Un problema que se nos presenta algunas veces es que desde un método `m1()` de una subclase no podemos invocar el método `m1()` de la superclase que estamos redefiniendo. Suponga que en la clase `FiguraRellena` queremos redefinir el método que verifica el invariante, el cual ya fue implementa-

do en la superclase `Figura`. Lo queremos redefinir, porque ahora hay que incluir una verificación sobre el nuevo atributo que incluimos. Lo ideal sería poder invocar el método de la superclase que estamos redefiniendo y luego sí verificar la condición adicional. Para poder hacer esto Java nos provee una variable llamada `super`, que siempre hace referencia a la superclase. En el ejemplo 8 mostramos la manera como se usa.

Ejemplo 8



Objetivo: Mostrar la manera como se utiliza la variable `super`, para tener acceso a los métodos de la superclase.

En este ejemplo avanzamos en el desarrollo de la clase `Línea`, mostrando la manera de implementar los métodos constructores y el método que retorna el tipo de figura para la persistencia.

```

public class Linea extends Figura
{
    // -----
    // Constantes
    // -----

    public final static String TIPO = "LINEA";
}

```

- La constante que definimos en esta clase la vamos a usar para establecer la cadena de caracteres que va a identificar este tipo de figuras en el momento de persistir en un archivo.

```

// -----
// Constructores
// -----

public Linea( int x1f, int y1f, int x2f, int y2f,
              Color colorLineaF, BasicStroke tipoLineaF )
{
    super( x1f, y1f, x2f, y2f, colorLineaF, tipoLineaF );
}

public Linea( BufferedReader br ) throws IOException,
                                         FormatoInvalidoException
{
    super( br );
}

// -----
// Métodos
// -----

public String darTipoFigura( )
{
    return TIPO;
}
}

```

El método `darTipoFigura()` hace parte del contrato definido por la interfaz `IFigura`. En este caso, retorna la cadena “LINEA”.

En la superclase `Figura` tenemos dos constructores. El primero pide como parámetro un valor para cada uno de los atributos. El segundo recibe un flujo de entrada conectado a un archivo, por el cual recibe la misma información.

En esta clase también tendremos dos constructores. Como no hay información adicional que debamos manejar, nos contentamos con invocar el respectivo constructor de la superclase. Para eso utilizamos la variable “super”. Como no indicamos un método en particular, se invoca el constructor. En el primer caso, le pasamos la misma información que recibimos como parámetro. En el segundo caso pasamos el flujo de lectura que recibimos.

El método que verifica el invariante en la clase `Linea` es igual al que implementamos en la clase `Figura`, de manera que no tenemos que implementar nada aquí: sencillamente ese método se hereda.

Ejemplo 9



Objetivo: Mostrar la manera como se utiliza la variable `super`, para tener acceso a los métodos de la superclase.

En este ejemplo avanzamos en el desarrollo de la clase `FiguraRellena`, mostrando la manera de implementar los métodos constructores, el método que salva la información en un archivo y el método que verifica el invariante.

```

public abstract class FiguraRellena extends Figura
{
    // -----
    // Atributos
    // -----
    protected Color colorFondo;
}

```

Esta clase define un nuevo atributo (`colorFondo`), de manera que nos toca incluir esta nueva información en la verificación del invariante y en la persistencia, al igual que extender los métodos constructores.

```

// -----
// Constructores
// -----

public FiguraRellena( int x1f, int y1f, int x2f, int y2f,
                      Color colorLineaF, Color colorFondoF,
                      BasicStroke tipoLineaF )
{
    super( x1f, y1f, x2f, y2f, colorLineaF, tipoLineaF );
    colorFondo = colorFondoF;
}

public FiguraRellena( BufferedReader br ) throws
                      IOException, FormatoInvalidoException
{
    super( br );
    String lineaFondo = br.readLine();
    String[] strValoresFondo = lineaFondo.split( ";" );
    if( strValoresFondo.length != 3 )
        throw new FormatoInvalidoException( lineaFondo );
    try
    {
        int r2 = Integer.parseInt( strValoresFondo[ 0 ] );
        int g2 = Integer.parseInt( strValoresFondo[ 1 ] );
        int b2 = Integer.parseInt( strValoresFondo[ 2 ] );
        colorFondo = new Color( r2, g2, b2 );
    }
    catch( NumberFormatException nfe )
    {
        throw new FormatoInvalidoException( lineaFondo );
    }
}

// -----
// Métodos
// -----

public void guardar( PrintWriter out )
{
    super.guardar( out );
    out.println( colorFondo.getRed() + ";" +
                colorFondo.getGreen() + ";" +
                colorFondo.getBlue() );
}

protected void verificarInvariante()
{
    super.verificarInvariante();
    assert colorFondo != null : "Color inválido";
}

```

Para el primer constructor recibimos toda la información que necesita la figura, más el color de fondo. Invocamos el constructor de la superclase pasándole la información que recibimos y, luego, almacenamos el color de fondo en el atributo definido con este fin.

Para el segundo constructor, después de invocar el respectivo constructor de la superclase, leemos del archivo la línea con la información del color que salvamos (conocemos el formato, porque fue el que definimos en el método de guardar). Esa línea la partimos para recuperar los valores de los componentes rojo, verde y azul del color. Con estos valores numéricos reconstruimos el objeto con el color de fondo y lo guardamos en el atributo que declaramos. En caso de cualquier problema, lanzamos una excepción.

El método que almacena la información en un flujo de salida (guardar) invoca por medio de la variable "super" el mismo método que fue implementado en la superclase. Luego, agrega al archivo la información del color de fondo, de manera que después pueda recuperarlo. Para esto almacenamos el valor de los componentes rojo, verde y azul del color.

Finalmente tenemos el método que verifica el invariante de la clase. Debemos como primera medida invocar el respectivo método de la clase Figura (el cual está declarado como protegido). Eso lo hacemos utilizando la variable "super". Luego, validamos que el nuevo atributo tenga un valor distinto de null.



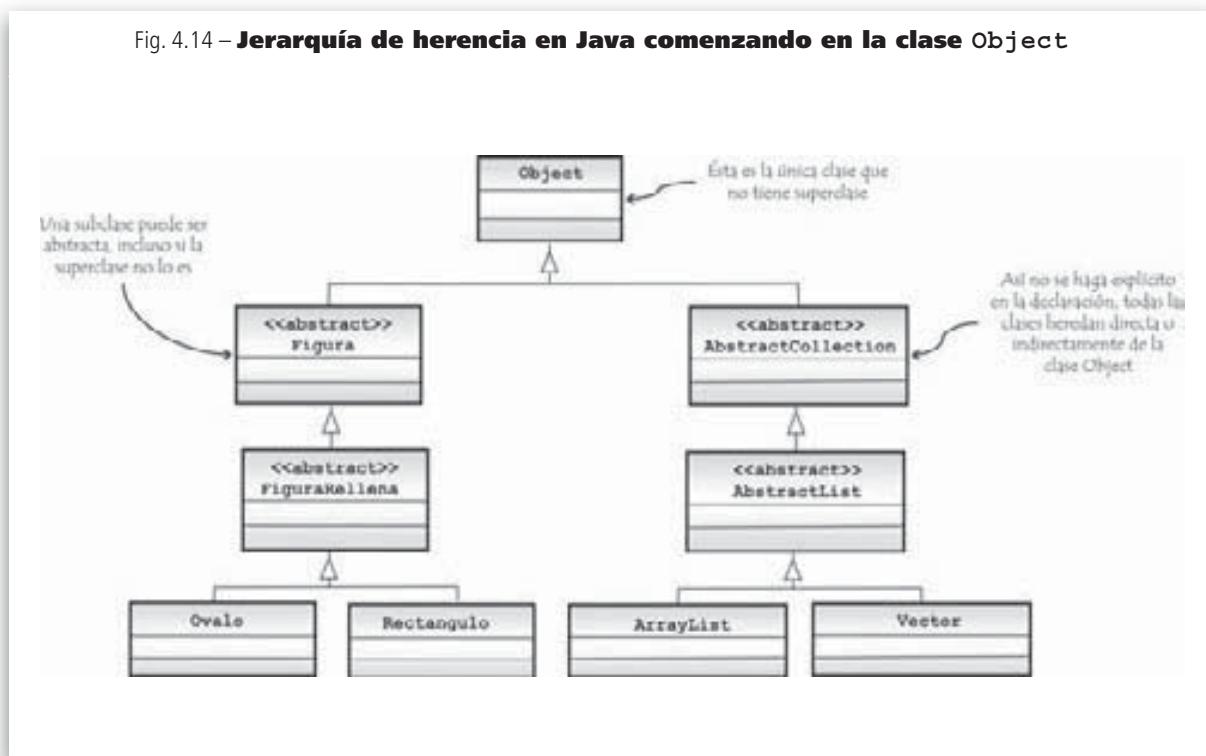
La variable `super` nos permite tener acceso a los métodos de la superclase cuando: (1) queremos invocar su método constructor o (2) queremos invocar el método que estamos redefiniendo. En ningún otro caso es necesario usarla.

3.10.4. La Clase Object

En Java existe una clase llamada `Object`, de la cual heredan todas las demás clases, así esto no lo hagamos explícito en la declaración con la palabra “`extends`”. Esta clase se puede ver como la raíz de toda la jerarquía

de herencia, tal como se sugiere en la figura 4.14. En esa imagen podemos apreciar también que las clases `Vector` y `ArrayList` comparten buena parte de sus implementaciones mediante el mecanismo de herencia y la construcción de clases abstractas. Una práctica muy común en el diseño orientado a objetos.

Fig. 4.14 – Jerarquía de herencia en Java comenzando en la clase Object



La clase `Object` no es abstracta (se pueden crear instancias de esta clase) y tiene tres métodos, en los cuales nos vamos a interesar en esta parte, los cuales ya hemos utilizado en niveles anteriores:

- `public boolean equals(Object o)`: Este método recibe como parámetro otro objeto y retorna verdadero si dicho objeto y el objeto que recibe el llamado son el mismo. En nuestras cla-
- ses, cada vez que queramos modificar el concepto de igualdad entre objetos, debemos redefinir este método. El hecho de que haga parte de la clase `Object` implica que todos los objetos de todas las clases se pueden comparar con este método.
- `public String toString()`: Este método retorna una representación como cadena de caracteres del objeto. Por defecto corresponde a una

cadena que contiene el nombre de la clase, seguido del carácter '@' y luego un valor numérico. Algunos componentes gráficos, como el `JList`, invocan este método sobre cada elemento que van a presentar, de manera que si este método no está redefinido, utilizan la implementación por defecto que viene con la clase `Object`. Siempre es válido invocar este método sobre cualquier objeto en Java.

- `protected Object clone()` : Este método protegido permite crear una copia de un objeto a partir de otro. Al igual que con los métodos anteriores, si queremos asociar un significado especial al concepto de clonación, debemos redefinir este método en nuestra clase.

En el ejemplo 10 presentamos la manera de redefinir estos métodos en una clase.

Ejemplo 10



Objetivo: Mostrar la manera de redefinir los métodos básicos de la clase `Object`.

En este ejemplo creamos una clase simple, en la cual redefinimos los tres métodos de la clase `Object` presentados anteriormente

```
public class Persona
{
    private String nombre;
    private int codigo;

    public Persona( String nom, int cod )
    {
        nombre = nom;
        codigo = cod;
    }

    public boolean equals( Object o )
    {
        Persona p = ( Persona )o;
        return codigo == p.codigo;
    }

    public String toString( )
    {
        return codigo + ": " + nombre;
    }

    public Object clone( )
    {
        return new Persona( nombre, codigo );
    }
}
```

- Definimos en este ejemplo la clase `Persona`, que por defecto va a heredar de la clase `Object`. Si quisieramos, podríamos hacerlo explícito en el encabezado, agregando "extends `Object`".
- La clase tiene dos atributos: el nombre de la persona y un código que suponemos único.
- Para redefinir el método `equals()` debemos respetar la signatura exacta del método que se encuentra definido en la clase `Object`. Por esa razón el parámetro que recibimos es de ese tipo. Lo primero que hacemos es la conversión del parámetro a una referencia de la clase `Persona`. Luego, puesto que estamos suponiendo que el código es único, si la persona que llega como parámetro tiene el mismo código, retornamos verdadero.
- Para redefinir el método `toString()` simplemente retornamos una cadena de caracteres con la información que consideramos importante en la clase. Si este objeto llega a un componente gráfico, ésta será la cadena de caracteres que se despliegue.
- Para redefinir el método de clonación, retornamos un objeto de la clase `Persona`, con la información de la persona actual. Podemos hacer este retorno sin necesidad de hacer la conversión explícita a la clase `Object`, aprovechando el polimorfismo que existe en Java. También habríamos podido utilizar el método implementado en la clase `Object`, contentándonos con llamar `super.clone()`.

```

Persona p1 = new Persona( "Paola", 2929 );

Persona p2 = ( Persona )p1.clone( );

System.out.println( p1 );
System.out.println( p2 );

if( p1.equals( p2 ) )
    System.out.println( "IGUALES" );
else
    System.out.println( "DISTINTOS" );

```

Supongamos que ejecutamos estas instrucciones desde algún método de otra clase. Con los métodos redefinidos, obtenemos la siguiente salida en consola:

2929: Paola
2929: Paola
IGUALES

Si las ejecutamos sin redefinir el método equals() obtenemos:

2929: Paola
2929: Paola
DISTINTOS

Si las ejecutamos sin redefinir el método toString() da:

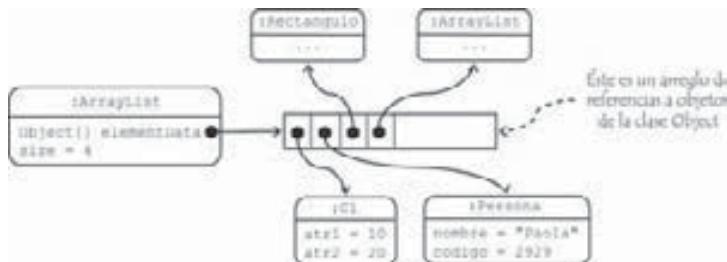
uniandes.cupi2.herencia.Persona@10b62c9
uniandes.cupi2.herencia.Persona@82ba41
IGUALES

Si las ejecutamos sin redefinir el método clone() no compila, porque la implementación por defecto de este método en la clase Object lanza la excepción CloneNotSupportedException, que no estamos manejando en el código. Si resolvemos ese problema, aparece en ejecución dicha excepción, puesto que la implementación por defecto exige que la clase en la que se use implemente la interfaz Cloneable. Si corregimos esto, el código funciona correctamente.

Pero, ¿cuál es realmente la ganancia de tener la clase Object en el lenguaje? Hay dos respuestas a esta pregunta. La primera es que sirve para garantizar que todos los objetos tengan un comportamiento común mínimo, que permita construir clases adaptables, que aprovechen el polimorfismo y la asociación dinámica. La segunda respuesta es que son la base para construir las estructuras polimorfas reutilizables (como por ejemplo las clases ArrayList y Vector). En dichas estructuras almacenamos elementos de la clase Object, lo que, por el

polimorfismo del lenguaje, nos permite guardar allí objetos de cualquier clase que queramos. Eso se ilustra en la figura 4.15. Allí se puede apreciar que dentro de un ArrayList hay en realidad un arreglo de referencias de la clase Object y un atributo de tipo entero que indica cuántos elementos están presentes. En cada casilla del arreglo puede haber una instancia de absolutamente cualquier clase. Esto explica por qué cuando utilizamos el método get() para recuperar un objeto de un ArrayList debemos utilizar el operador de conversión.

Fig. 4.15 – Una estructura contenedora polimorfa



En la siguiente tarea vamos a implementar una contenedora polimorfa simple.



Tarea 3

Objetivo: Implementar una contenedora polimorfa, definiendo una estructura que almacene elementos de la clase `Object`.

Siga las instrucciones que se plantean a continuación:

1. Cree en Eclipse un proyecto llamado `n10_contenedoras`. Vamos a construir las clases en el paquete `uniandes.cupi2.contenedoras`.

2. Declare una interfaz (archivo `IPila.java`) llamada `IPila`, que incluya las siguientes signatures de métodos:

```
public interface IPila
{
    public void agregar( Object elem );
    public Object darPrimero( );
    public void eliminar( );
    public int darLongitud( );
}
```

3. Construya una clase llamada `PilaArreglo`, que implemente la interfaz `IPila` antes descrita y un método que verifique el invariant de la clase. En el constructor debe definir una capacidad máxima de la pila de 20 elementos. Utilice los atributos que se plantean a continuación:

```
public class PilaArreglo implements IPila
{
    private Object[] elems;
    private int numElems;
    ...
}
```

4. Defina los escenarios de prueba y construya la clase `PilaArregloTest` que valida el correcto funcionamiento de la clase `PilaArreglo`. Ejecute las pruebas utilizando JUnit.

■ Una pila es una estructura en la cual los elementos se insertan por el tope y se retiran por el mismo punto. El método `agregar()` inserta un nuevo elemento en el tope de la pila. El método `darPrimero()` retorna el elemento que se encuentra en el tope. El método `eliminar()` suprime el elemento del tope de la pila. El método `darLongitud()` retorna el número de elementos en la pila.

■ En esta implementación la pila está representada por un arreglo de tipo `Object`, en el cual vamos a almacenar los elementos. El elemento del tope de la pila se encuentra en la posición `"numElems"` del arreglo. Si la pila está vacía, el atributo `"numElems"` tiene el valor -1.



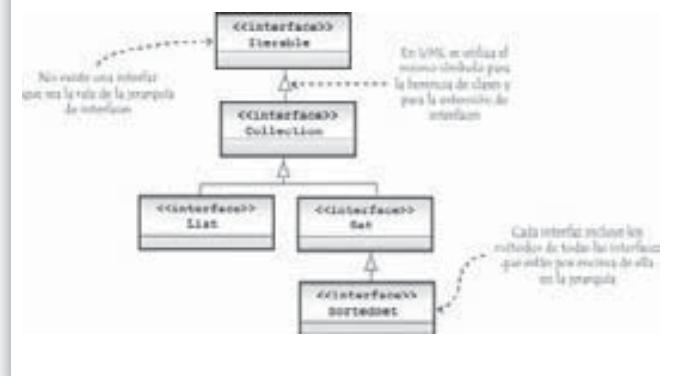
En este punto, el lector es capaz de construir una estructura contenedora polimorfa simple, que cumpla un contrato funcional definido por una interfaz. Podría eventualmente construir una contenedora que implementara la interfaz `List` que necesitamos en el editor de dibujos que estamos desarrollando.

3.10.5. Extensión de Interfaces

La herencia también se puede aplicar a las interfaces. En ese caso la herencia se interpreta como la suma de los contratos funcionales definidos por las interfaces de las cuales hereda. En este contexto es preferible utilizar el término **extensión**, puesto que no se aplican exactamente las mismas reglas que se utilizan en el mecanismo de herencia entre clases. En la figura 4.16 se puede apreciar la relación entre algunas interfaces definidas en Java, en donde no existe una interfaz que sea la raíz de toda la jerarquía.

En el caso de las interfaces la extensión puede ser múltiple, en el sentido de que una interfaz puede extender a la vez de varias interfaces. El mecanismo general de extensión se ilustra en el ejemplo 11.

Fig. 4.16 – **Ejemplo de extensión de interfaces en Java**



Ejemplo 11



Objetivo: Mostrar el mecanismo de extensión de interfaces en Java.

En este ejemplo se presentan algunas interfaces simples que utilizan el mecanismo de extensión para establecer su contrato funcional.

```

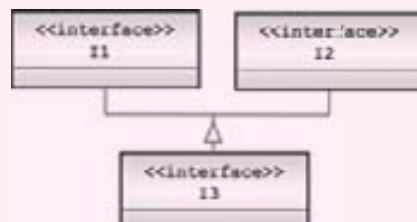
public interface I1
{
    public void m1( int a );
}

public interface I2
{
    public void m2();
}

public interface I3 extends I1, I2
{
    public void m3( int b );
}
  
```

■ Las interfaces I1 e I2 incluyen cada una la firma de un método. Cada interfaz se define en su propio archivo (en este caso I1.java e I2.java).

■ La interfaz I3 está construida por extensión de las dos interfaces anteriores. Cualquier clase que quiera implementar I3 debe incluir la definición de los tres métodos (m1, m2 y m3).



3.11. Uso del Mecanismo de Herencia en Niveles Anteriores

En los niveles anteriores construimos clases utilizando el mecanismo de herencia, sin hacerla explícita, en al menos tres casos: para hacer ventanas (heredando de

la clase `JFrame`), para hacer paneles (heredando de la clase `JPanel`) y para definir tipos de excepción (heredando de la clase `Exception`). En el ejemplo 12 volvemos a mirar esas tres situaciones sobre partes del caso de estudio, y aprovechamos para explicar la solución usando la terminología introducida en este nivel.

Ejemplo 12



Objetivo: Mostrar el uso que le hemos dado a la herencia en niveles anteriores y explicar las soluciones utilizando la terminología correcta.

En este ejemplo se presentan algunas clases del caso de estudio, que se deben construir utilizando el mecanismo de herencia.

```
public class FormatoInvalidoException extends Exception
{
    public FormatoInvalidoException( String linea )
    {
        super( "Formato inválido:" + linea );
    }
}
```

- Para crear un nuevo tipo de excepción debemos construir una subclase de la clase `Exception`. En nuestro caso definimos un constructor que recibe como parámetro la línea del archivo donde se encontró el problema. Llamamos al constructor de la superclase (usando la variable "super") con un mensaje explicando el error.

```
public class InterfazPaint extends JFrame
{
    private Dibujo dibujo;
    public InterfazPaint()
    {
        dibujo = new Dibujo();
        ...
        panelBotones = new PanelBotones( this );
        add( panelBotones, BorderLayout.WEST );
        setSize( 800, 600 );
        setTitle( "Paint" );
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setLocationRelativeTo( null );
        ...
    }
}
```

- Para crear la ventana de la interfaz de usuario debemos construir una subclase de la clase `JFrame`. Dicha clase ya tiene definidos unos atributos y métodos con el comportamiento básico de las ventanas.

- En el constructor invitamos los métodos heredados (`add`, `setSize`, `setTitle`, etc.) como si hubieran sido desarrollados en nuestra clase.
- Hay varios atributos que agregamos, en particular uno (`dibujo`) para manejar la referencia al modelo del mundo.
- Cuando creamos una instancia de esta clase, va a tener espacio para todos los atributos heredados y para los atributos aquí definidos.

```
public class PanelBotones extends JPanel
    implements ActionListener
{
    public PanelBotones( InterfazPaint ip )
    {
        ...
        setBorder( new TitledBorder( "" ) );
        ...
        botonColorFondo.addActionListener( this );
    }

    public void actionPerformed( ActionEvent evento )
    {
        ...
    }
}
```

- Para crear un panel dentro de la ventana del programa, debemos construir una subclase de la clase `JPanel`.
- Como es un panel activo (tiene botones a los cuales les debe atender sus eventos), debe implementar la interfaz `ActionListener`.
- Como parte de esta interfaz, implementa el método `actionPerformed()`.
- De nuevo, en esta clase utilizamos todos los métodos heredados como si fueran propios.

3.12. Los Componentes de Manejo de Menús

Para incluir menús de opciones en un programa debemos utilizar tres clases que nos provee Java en su paquete `javax.swing`: la clase `JMenuBar` (representa la barra de menú), la clase `JMenu` (representa un menú colgante) y la clase `JMenuItem` (representa una opción de un menú colgante). En la figura 4.17 mostramos la relación que existe entre el menú que queremos construir para el caso de estudio y las clases antes mencionadas. Allí podemos ver que necesitamos una instancia de la clase `JMenuBar`, la cual tiene asociado un objeto de la clase `JMenu` (que representa el menú colgante llamado **Archivo**), y éste tiene en su interior cinco objetos de la clase `JMenuItem`, cada uno representando una opción del programa.

Fig. 4.17 – Relación entre el menú del programa y las clases de Java que lo implementan



Los principales métodos de las clases que permiten implementar los menús son los siguientes:

Clase `JMenuBar`:

- `JMenuBar()`: Éste es el método constructor de la clase y permite crear una barra de menú sin ningún elemento.
- `add(menu)`: Con este método se agrega al final de la barra un nuevo menú colgante (instancia de la clase `JMenu`).

Clase `JMenu`:

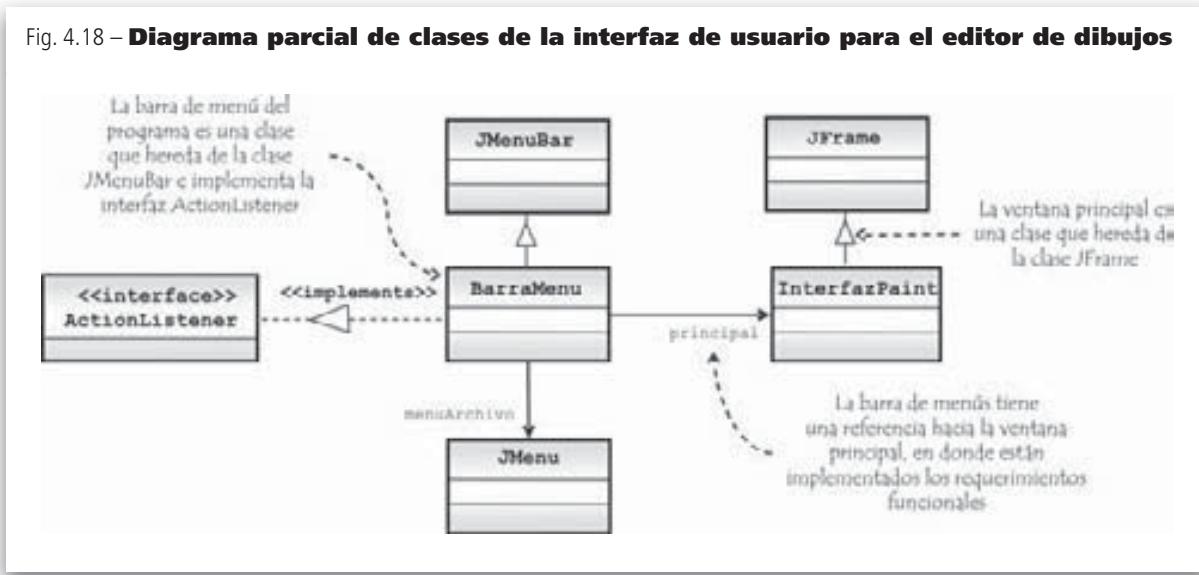
- `JMenu(título)`: Éste es el método constructor de la clase y permite crear un menú colgante cuyo título es el recibido como parámetro.
- `add(opción)`: Con este método se agrega al final del menú colgante una nueva opción (instancia de la clase `JMenuItem`).
- `addSeparator()`: Este método permite incluir dentro del menú colgante una línea de separación, como aquélla que se muestra en la figura 4.17 entre la opción **Salvar Como** y **Salir**. Estas líneas de separación son muy útiles para agrupar las opciones del menú por el tipo de funcionalidad que proveen.

Clase `JMenuItem`:

- `JMenuItem(nombre)`: Permite crear una opción con un nombre asociado (por ejemplo, **Salvar Como**).
- `setActionCommand(comando)`: Los eventos de las opciones de los menús van a ser manejados de manera similar a como se manejan los eventos de los botones. Con este método podemos asociar con la opción del menú una cadena de caracteres que va a permitir identificar la orden del usuario en el momento de atender el evento. Ésta es la cadena que va a retornar el método `getActionCommand()` cuando se invoca desde el método `actionPerformed()`.
- `addActionListener(responsable)`: Con este método definimos cuál es el componente que va a atender el evento que se produce cuando el usuario selecciona esta opción del menú. El parámetro "responsable" debe ser una instancia de una clase que implemente la interfaz `ActionListener`.

Utilizando las clases anteriores, obtenemos el diagrama con el diseño parcial que se muestra en la figura

4.18. Allí podemos ver que la barra de menús es la encargada de responder a los eventos generados por las opciones de sus menús (implementa la interfaz ActionListener), y que lo hace delegando esta responsabilidad en la ventana principal hacia la cual tiene una asociación.



En el ejemplo 13 mostramos la implementación de la parte de la interfaz de usuario que tiene que ver con el manejo del menú de opciones.

Ejemplo 13



Objetivo: Mostrar la manera de implementar un menú de opciones en un programa.

En este ejemplo mostramos la implementación de la clase BarraMenu encargada de implementar el menú de opciones en el editor de dibujos.

```

public class BarraMenu extends JMenuBar implements ActionListener
{
    // -----
    // Constantes
    // -----
    private static final String NUEVO = "Nuevo";
    private static final String ABRIR = "Abrir";
    private static final String SALVAR = "Salvar";
    private static final String SALVAR_COMO = "SalvarComo";
    private static final String SALIR = "Salir";
  
```

■ Esta clase debe heredar de la clase JMenuBar e implementar la interfaz ActionListener, para poder hacer el manejo de los eventos generados cuando el usuario selecciona una opción.

■ Declaramos cinco constantes de tipo cadena de caracteres, para identificar las opciones del menú.

```

// -----
// Atributos
// -----
private InterfazPaint principal;

private JMenu menuArchivo;
private JMenuItem itemNuevo;
private JMenuItem itemAbrir;
private JMenuItem itemSalvar;
private JMenuItem itemSalvarComo;
private JMenuItem itemSalir;

public BarraMenu( InterfazPaint ip )
{
    principal = ip;

    menuArchivo = new JMenu( "Archivo" );
    add( menuArchivo );

    itemNuevo = new JMenuItem( "Nuevo" );
    itemNuevo.setActionCommand( NUEVO );
    itemNuevo.addActionListener( this );
    menuArchivo.add( itemNuevo );

    itemAbrir = new JMenuItem( "Abrir" );
    itemAbrir.setActionCommand( ABRIR );
    itemAbrir.addActionListener( this );
    menuArchivo.add( itemAbrir );

    itemSalvar = new JMenuItem( "Salvar" );
    itemSalvar.setActionCommand( SALVAR );
    itemSalvar.addActionListener( this );
    menuArchivo.add( itemSalvar );

    itemSalvarComo = new JMenuItem( "Salvar Como" );
    itemSalvarComo.setActionCommand( SALVAR_COMO );
    itemSalvarComo.addActionListener( this );
    menuArchivo.add( itemSalvarComo );

    menuArchivo.addSeparator();

    itemSalir = new JMenuItem( "Salir" );
    itemSalir.setActionCommand( SALIR );
    itemSalir.addActionListener( this );
    menuArchivo.add( itemSalir );
}

```

Como atributos de la clase definimos: una referencia a la ventana principal, una instancia de la clase JMenu y cinco atributos, uno por cada una de las opciones.

El método constructor recibe como parámetro una referencia a la ventana principal, en donde se encuentran implementados los requerimientos funcionales del programa.

Lo primero que hacemos es almacenar dicha referencia en el respectivo atributo.

Luego, creamos el menú llamado Archivo y lo agregamos a la barra.

Después pasamos a crear cada una de las opciones del menú. Para esto seguimos cuatro pasos: (1) crear una instancia de la clase JMenuItem, (2) asignar un nombre al evento que se asocia con el objeto anterior, (3) indicar que el evento será atendido por la clase que estamos construyendo y (4) agregar la nueva opción al menú.

Utilizamos el método addSeparator() de la clase JMenu para agregar una línea de separación.

```

public void actionPerformed( ActionEvent evento )
{
    String comando = evento.getActionCommand( );

    if( NUEVO.equals( comando ) )
    {
        principal.reiniciar( );
    }
    else if( ABRIR.equals( comando ) )
    {
        principal.abrir( );
    }
    else if( SALVAR.equals( comando ) )
    {
        principal.salvar( );
    }
    else if( SALVAR_COMO.equals( comando ) )
    {
        principal.salvarComo( );
    }
    else if( SALIR.equals( comando ) )
    {
        principal.dispose( );
    }
}
}

```

Este método hace parte de la interfaz ActionListener y será invocado cada vez que el usuario seleccione una opción de algún menú de la barra.

El método recibe como parámetro una instancia de la clase ActionEvent, en la cual se incluye la información del evento generado por el usuario.

Lo primero que hacemos es tomar el nombre del evento. Aquí recuperamos la cadena de caracteres que asociamos con cada una de las opciones en el constructor.

Luego, establecemos cuál de las cinco opciones disponibles en el menú fue la que el usuario seleccionó e invocamos el método de la ventana principal que implementa dicho requerimiento funcional.

A continuación planteamos al lector dos tareas con extensiones a la barra de menús del programa.

Tarea 4



Objetivo: Extender el menú de opciones del editor de dibujos.

Modifique el programa siguiendo las etapas que se plantean a continuación.

- Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.

2. Agregue un nuevo menú llamado **Ayuda** con una opción **Acerca de...**, tal como aparece en la siguiente figura:



3. Cree una clase llamada **AcercaDe** que herede de la clase **JDialog** y que presente en la pantalla un diálogo modal con la información del autor del programa y la fecha de creación.

4. Asocie con el evento de la opción **Acerca de...** la presentación del diálogo anterior.

Tarea 5

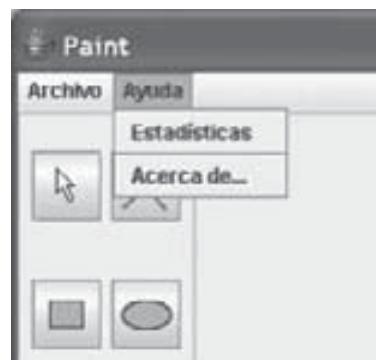


Objetivo: Extender el menú de opciones del editor de dibujos.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.

2. Agregue una nueva opción llamada **Estadísticas** al menú de **Ayuda** creado en la tarea anterior:



3. Cree una clase llamada **Estadisticas** que herede de la clase **JDialog** y que presente en la pantalla un diálogo modal con la siguiente información estadística: (1) número total de figuras en el editor, (2) tipo de la figura seleccionada (si hay alguna), (3) número de rectángulos en el dibujo, (4) número de líneas en el dibujo y (5) número de óvalos en el dibujo.

4. Asocie con el evento de la opción **Estadísticas** la presentación del diálogo anterior.

3.13. Manejo de Eventos del Ratón

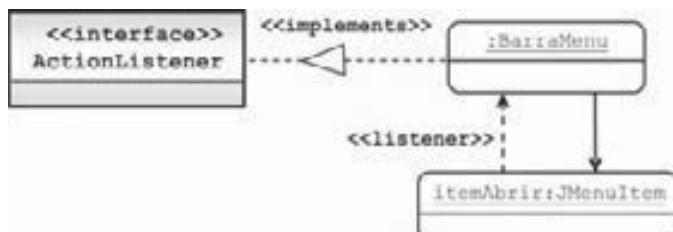
El siguiente problema al que nos enfrentamos es lograr tomar los eventos generados por el usuario cuando éste hace clic con el ratón sobre la zona de dibujo

del editor. El manejo de los eventos en Java siempre se hace utilizando dos conceptos (escuchador y evento) que se integran con la máquina virtual de Java, como se muestra en la figura 4.19. Allí se muestra el caso general del modelo de escucha de eventos y se ilustra con los eventos generados por un botón.

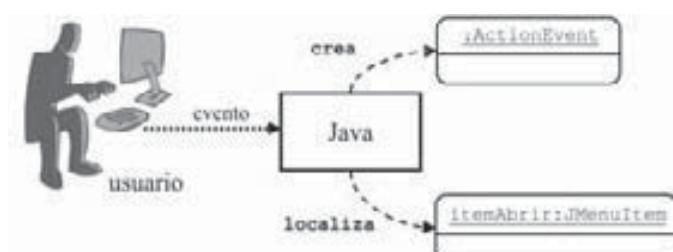
Fig. 4.19 – **Modelo de escucha de eventos en Java**



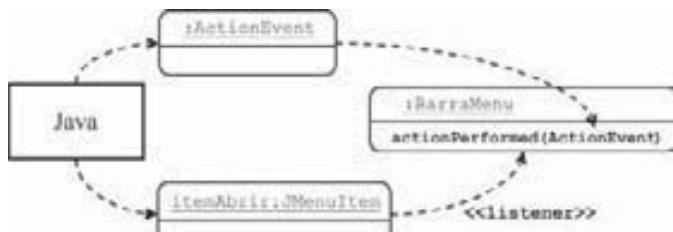
- Una clase implementa la interfaz de un tipo de escuchador (listener). En el caso de los botones, dicha interfaz se llama ActionListener.



- Las instancias de dicha clase van a poder escuchar ese tipo de eventos.



- Cuando el usuario realiza alguna acción sobre la interfaz, Java se encarga de crear un objeto que representa el evento, el cual incluye toda la información necesaria para interpretarlo.



- Al objeto sobre el cual se generó el evento (itemAbrir) se le pide que identifique su escuchador (:BarraMenu), y a dicho objeto Java le invoca el método que atiende ese tipo de eventos (actionPerformed), pasándole como parámetro el objeto que representa el evento producido (:ActionEvent).

Para reaccionar a los eventos generados por el ratón, el escuchador (*listener*) debe implementar la interfaz `MouseListener` (del paquete `java.awt.event`), y el evento que se produce es una instancia de la clase `MouseEvent` que se encuentra en ese mismo paquete. Los métodos con los que cuentan estos elementos se describen a continuación:

Interfaz `MouseListener`:

- `mouseClicked(Evento)`: Este método se invoca cuando se ha hecho clic con el ratón sobre un componente gráfico. Recibe como parámetro una instancia de la clase `MouseEvent`, con los detalles del evento.
- `mousePressed(Evento)`: Este método se invoca cuando un botón del ratón ha sido presionado mientras se encontraba sobre un componente gráfico. Recibe como parámetro una instancia de la clase `MouseEvent`, con los detalles del evento.
- `mouseReleased(Evento)`: Este método se invoca cuando un botón del ratón ha sido soltado mientras se encontraba sobre un componente gráfico. Recibe como parámetro una instancia de la clase `MouseEvent`, con los detalles del evento.
- `mouseEntered(Evento)`: Este método se invoca cuando el ratón ha entrado a la zona gráfica de un componente. Recibe como pará-

metro una instancia de la clase `MouseEvent`, con los detalles del evento.

- `mouseExited(Evento)`: Este método se invoca cuando el ratón ha salido de la zona gráfica de un componente. Recibe como parámetro una instancia de la clase `MouseEvent`, con los detalles del evento.

Clase `MouseEvent`:

- `getX()`: Retorna un valor entero con la coordenada horizontal del punto en el cual sucedió el evento, con respecto al sistema de referencia del componente.
- `getY()`: Retorna un valor entero con la coordenada vertical del punto en el cual sucedió el evento, con respecto al sistema de referencia del componente.
- `getButton()`: Retorna un valor entero que indica cuál de los botones del ratón (izquierdo o derecho) produjo el evento. Los valores posibles de retorno son las constantes `BUTTON1` o `BUTTON2`.
- `getClickCount()`: Retorna el número de veces que el usuario hizo clic con el ratón como parte del evento. Si este método retorna el valor 2, por ejemplo, se trata de un doble clic.

Un componente gráfico declara que quiere generar eventos del ratón utilizando el método `addMouseListener()` tal como se muestra en el ejemplo 14.

Ejemplo 14



Objetivo: Mostrar la manera de incluir en un programa el manejo de los eventos del ratón.

En este ejemplo presentamos una parte de la clase `PanelEditor`, la cual se encarga de hacer el manejo de los eventos del ratón generados por el usuario en el editor de dibujos.

```
public class PanelEditor extends JPanel implements MouseListener
{
    public PanelEditor( )
    {
        ...
    }
}
```



Esta clase va a ser su propio escuchador de eventos del ratón. Por esta razón implementa la interfaz `MouseListener`.

```

        addMouseListener( this );
    }

    public void mouseClicked( MouseEvent evento )
    {
        if( evento.getButton( ) == MouseEvent.BUTTON1 )
        {
            ...
        }
    }

    public void mousePressed( MouseEvent evento) { }

    public void mouseReleased( MouseEvent evento) { }

    public void mouseEntered( MouseEvent evento) { }

    public void mouseExited( MouseEvent evento) { }
}

```

■ Dentro del constructor se declara con el método addMouseListener(), que es la misma clase quien va a escuchar los eventos del ratón, puesto que pasamos como parámetro la variable "this".

■ Únicamente vamos a escuchar uno de los cinco eventos posibles: el evento de clic del ratón. Por esta razón implementamos el método mouseClicked().

■ Dentro de dicho método lo primero que hacemos es verificar que el evento se generó con el botón izquierdo del ratón. Luego vendría el código con la reacción del programa a dicho evento, dependiendo de las coordenadas y del estado en el que se encuentre.

■ Debemos implementar los cinco métodos de la interfaz, así sólo vayamos a manejar uno de los eventos.

A continuación proponemos al lector una serie de tareas que implican manejar los distintos eventos que pueden ser generados por el ratón.



Tarea 6

Objetivo: Extender el editor de dibujos manejando los eventos del ratón.

Modifique el programa siguiendo las etapas que se plantean a continuación.

- Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.

- Modifique el programa de manera que si el usuario (en modo de selección) hace clic sobre un punto del editor en donde no hay ninguna figura, presente una ventana de diálogo informándole el error.

**Tarea 7**

Objetivo: Extender el editor de dibujos manejando los eventos del ratón.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
2. Modifique el panel que contiene los botones de extensión para que haya dos zonas de texto que tengan asociadas las etiquetas "X" y "Y".
3. Modifique el programa para que, cada vez que el usuario haga clic en cualquier punto de la zona de dibujo, aparezcan en las zonas de texto agregadas en el punto anterior las coordenadas del evento.
4. Modifique el programa para que aparezca en la zona de extensión la etiqueta "DENTRO" si el ratón se encuentra dentro de la zona de dibujo, o "FUERA" en caso contrario.

**Tarea 8**

Objetivo: Extender el editor de dibujos manejando los eventos del ratón.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
2. Modifique el programa para que, cada vez que el usuario haga doble clic sobre una figura del editor, se abra una ventana con toda la información que se tiene sobre la figura.
3. Modifique el programa para que, si el usuario hace clic con el botón derecho del ratón sobre la figura seleccionada, ésta deje de estarlo (no queda ninguna figura seleccionada en el editor).

3.14. Dibujo Básico en Java

En esta sección vamos a estudiar los elementos gráficos necesarios para hacer dibujos simples en dos dimensiones, incluyendo el manejo de textos.

3.14.1. Modelo Gráfico de Java

Para comenzar, debemos entender tres aspectos básicos del funcionamiento del modelo gráfico de Java:

- Todo componente gráfico (en nuestro caso vamos a dibujar sobre un panel) tiene una superficie de dibujo, la cual nos provee los métodos necesarios

para hacerlo. El problema aquí se reduce a obtener dicha superficie a partir del componente en el que queremos dibujar y a invocar los métodos correspondientes. En Java la superficie de dibujo está implementada por las clases `Graphics` y `Graphics2D`, tema de la siguiente sección.

- Una de las responsabilidades de Java, en cuanto tiene que ver con el manejo de los componentes gráficos de la interfaz, es decidir en qué momento debe pintarlos en la pantalla. Si, por ejemplo, el usuario minimiza la ventana y luego la vuelve a maximizar, o si el usuario coloca una ventana de otro programa por delante y luego la cierra, es Java quien decide que debe volver a pintar sus

elementos o parte de ellos. Allí nosotros no participamos de manera directa. Nuestra participación en ese caso es indirecta, puesto que queremos que, cuando pida a los componentes gráficos que se muestren en la pantalla, invoque nuestros métodos de dibujo. Si no lo hacemos así, todo lo que pongamos en la superficie de dibujo de un panel

```
public class PanelEditor extends JPanel
{
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );

        // Aquí deben ir nuestras instrucciones
        // de dibujo
        ...
    }
}
```

va a desaparecer la primera vez que el panel se tenga que volver a dibujar. La solución es entonces redefinir el método llamado `paintComponent()` que tienen todos los componentes gráficos, para incluir allí nuestras instrucciones de dibujo, tal como se muestra en el siguiente fragmento de código:

- El método `paintComponent()` está definido en la superclase, de manera que Java lo puede invocar cada vez que se necesite volver a dibujar su contenido en la pantalla.
- Nosotros redefinimos este método en nuestra clase. Para eso invocamos como primera medida el método de la superclase encargado de dibujar el componente, y luego agregamos nuestras instrucciones de dibujo. En el caso del editor de dibujos, allí pediremos a cada figura que se dibuje sobre la superficie del panel.

- Si queremos que Java cambie lo que hemos dibujado sobre una superficie (por ejemplo, si el usuario pidió eliminar una figura), es nuestra responsabilidad invocar los métodos necesarios para que esto ocurra. Dentro del modelo gráfico de Java hay un método llamado `repaint()` (implementado por todos los componentes), que

se encarga de hacer todo el trabajo por nosotros, el cual invoca en algún momento el método `paintComponent()` que antes redefinimos. Eso quiere decir que si queremos proveer en el panel de dibujo un método de refresco, basta con utilizar el siguiente fragmento de código:

```
public class PanelEditor extends JPanel
{
    public void refrescar( )
    {
        repaint( );
    }
}
```

- El método `repaint()` está implementado en todos los componentes gráficos. Al invocarlo generamos una reacción en la cual el componente se vuelve a dibujar y a invocar su método `paintComponent()`.
- Puesto que ese método lo redefinimos e incluimos nuestras instrucciones de dibujo con el estado actual del editor, obtenemos como resultado que se actualiza nuestra visualización.

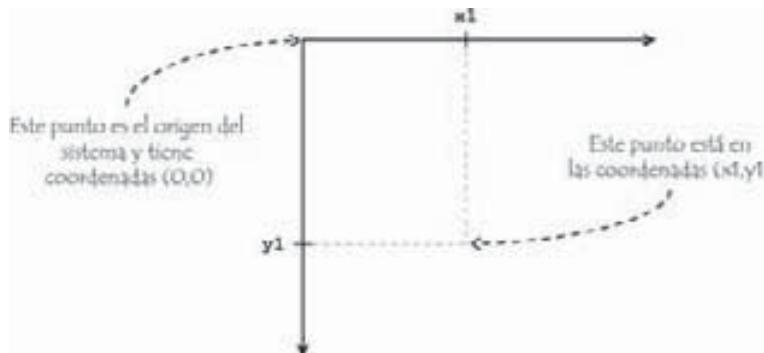


Para dibujar en un panel debemos redefinir su método `paintComponent()` e incluir allí las instrucciones que pintan sobre su superficie. La superficie de dibujo está implementada con las clases `Graphics` y `Graphics2D`. El método `repaint()` obliga al componente a recalcular su visualización.

3.14.2. Manejo de la Superficie de Dibujo

Todos los componentes gráficos en el *framework swing* de Java tienen una superficie de dibujo, que utiliza el sistema de coordenadas en píxeles que se muestra en la figura 4.20.

Fig. 4.20 – **Sistema de coordenadas en una superficie de dibujo**



La funcionalidad de dicha superficie se encuentra implementada por la clase `Graphics2D`, la cual hereda de la clase `Graphics`, que es una clase abstracta con algunos métodos básicos. La superficie de dibujo de un componente se recibe como parámetro del método

`paintComponent()` mencionado en la sección anterior, de manera que para iniciar el proceso de dibujo únicamente debemos tomar dicho parámetro y convertirlo en un elemento de la subclase `Graphics2D`, tal como se muestra en el siguiente fragmento de código:

```
public class PanelEditor extends JPanel
{
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        Graphics2D g2 = ( Graphics2D )g;
        // Aquí deben ir nuestras instrucciones
        // de dibujo sobre la superficie g2
        ...
    }
}
```

Comencemos estudiando dos métodos básicos de dibujo que hacen parte de la clase `Graphics2D`, de los cuales mostramos su uso en el ejemplo 15.

- `setColor(color)`: Este método permite definir el color con el que todas las operaciones de dibujo que siguen van a trabajar. Este valor se mantiene hasta que este método sea invo-

El primer paso para dibujar es tomar el objeto que representa la superficie de dibujo del componente gráfico, el cual llega como un parámetro de la clase `Graphics`.

Lo convertimos de una vez a una referencia de la clase `Graphics2D`, puesto que sabemos que el objeto que llega como parámetro pertenece a dicha subclase.

cado nuevamente con un valor distinto. Recibe como parámetro un objeto de la clase `Color`.

- `drawLine(x1,y1,x2,y2)`: Con este método dibujamos una línea en la superficie de dibujo (con el color definido por el método anterior) entre los puntos (x_1, y_1) y (x_2, y_2) .

Ejemplo 15

Objetivo: Mostrar la manera de utilizar los métodos básicos de dibujo.

En este ejemplo mostramos la implementación de un panel con una malla dibujada sobre él, como el que aparece a continuación. Si el tamaño de la ventana cambia, el panel debe completar el dibujo de la malla.



```
public class PanelMalla extends JPanel
{
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );

        Graphics2D g2 = ( Graphics2D )g;
        g2.setColor( Color.GRAY );

        int alto = getHeight();
        int ancho = getWidth();
        for( int i = 0; i < ancho; i +=25 )
        {
            g2.drawLine( i, 0, i, alto );
        }

        for( int j = 0; j < alto; j +=25 )
        {
            g2.drawLine( 0, j, ancho, j );
        }
    }
}
```

- Redefinimos el método de dibujo del componente y convertimos como primera medida el parámetro en una referencia de la clase Graphics2D.
- Con el método setColor() definimos que todo lo que dibujemos será de color gris (constante Color.GRAY).
- Con los métodos getHeight() y getWidth() de la clase JPanel establecemos el tamaño del mismo.
- Dibujamos líneas verticales cada 25 píxeles desde 0 hasta el ancho del panel. Cada línea va desde la posición 0 hasta el alto del panel.
- Luego dibujamos las líneas horizontales siguiendo un esquema equivalente al anterior.

Para dibujar cadenas de caracteres contamos con los siguientes métodos, para los cuales se ilustra su uso en el ejemplo 16.

- **setFont(tipo):** Este método recibe como parámetro un objeto de la clase `Font`, el cual define el tipo de letra que debe usarse en todas las operaciones de dibujo de cadenas de caracteres que se hagan a continuación. Este valor se mantiene hasta que este método sea invocado nuevamente con un valor distinto. Para crear un objeto de la clase `Font` se deben de-

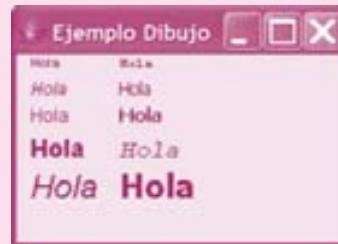
finir tres características: el nombre (por ejemplo "Arial", "Tahoma" o "Courier New"), el estilo (por ejemplo `Font.BOLD`, `Font.PLAIN` o `Font.ITALIC`) y el tamaño (por ejemplo 10, 12 ó 24).

- **drawString(cadena, x, y):** Este método permite escribir una cadena de caracteres sobre la superficie de dibujo, comenzando en el punto (x, y) definido en los parámetros. Utiliza para la operación el tipo de letra definido con el método anterior.

**Ejemplo 16**

Objetivo: Mostrar la manera de utilizar los métodos de dibujo de cadenas de caracteres.

En este ejemplo mostramos una parte de la implementación de un panel en el que se dibuja la cadena "Hola" con distintos tipos de letra.



```
public class PanelTipoLetra extends JPanel
{
    public void paintComponent( Graphics g )
    {
        super.paintComponent( g );
        Graphics2D g2 = ( Graphics2D )g;

        g2.setFont( new Font( "Arial", Font.BOLD, 10 ) );
        g2.drawString( "Hola", 10, 10 );
        g2.setFont( new Font("Courier New", Font.ITALIC,18));
        g2.drawString( "Hola", 75, 75 );
    }
}
```

■ Antes de escribir la cadena definimos el tipo de letra que queremos utilizar usando para esto el método `setFont()`.

■ Con el método `drawString()` escribimos la cadena que pasamos como parámetro, comenzando en las coordenadas dadas.

3.14.3. Manejo de Formas Geométricas

Además de las líneas y los textos, es posible dibujar distintos tipos de figuras geométricas, para las cuales existen clases especiales en Java, todas ellas implementando la interfaz `Shape`. Estas figuras se dibujan sobre un objeto de la clase `Graphics2D` utilizando los siguientes métodos:

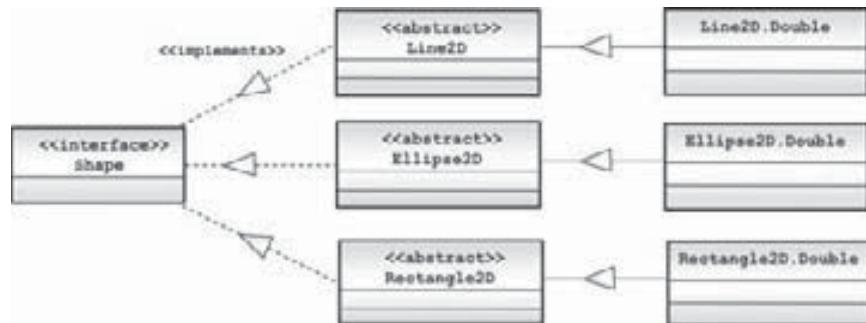
- `draw(figura)`: Este método recibe como parámetro un objeto de una clase que implementa la interfaz `Shape` y dibuja su borde en la superficie.
- `fill(figura)`: Este método recibe como parámetro un objeto de una clase que implementa la interfaz `Shape` y dibuja la figura rellena en la superficie.

La interfaz `Shape` se encuentra en el paquete `java.awt` y tiene dos métodos que nos interesan:

- `contains(x,y)`: Indica si el punto (x, y) se encuentra en el interior de la figura geométrica.
- `getPathIterator()`: Este método retorna la información geométrica necesaria para dibujar la figura. Su contenido exacto no nos interesa en este momento, puesto que es la superficie la que debe ser capaz de interpretar dicha información para poder hacer el dibujo.

Veamos ahora las clases que implementan las figuras geométricas que necesitamos para nuestro caso de estudio, las cuales se encuentran en el paquete `java.awt.geom`. En la figura 4.21 aparece una parte de la jerarquía de clases.

Fig. 4.21 – Jerarquía parcial de clases geométricas en Java



En la siguiente tabla se resume la manera de crear instancias de las clases que implementan las figuras geométricas y el resultado. Puesto que las clases Line2D, Ellipse2D y Rectangle2D son

clases abstractas, debemos crear instancias de las clases Line2D.Double, Ellipse2D.Double y Rectangle2D.Double:

```

int ancho = 100;
int alto = 200;

// Primer parámetro: coordenada X
// Segundo parámetro: coordenada Y
// Tercer parámetro: ancho de la figura en píxeles
// Cuarto parámetro: alto de la figura en píxeles

Rectangle2D.Double rect1 = new Rectangle2D.Double( 10, 10,
                                                 ancho,
                                                 alto );

// Con este método pedimos a la superficie que dibuje el
// borde de la figura
g2.draw( rect1 );
  
```



```

int ancho = 100;
int alto = 200;

// Primer parámetro: coordenada X
// Segundo parámetro: coordenada Y
// Tercer parámetro: ancho del rectángulo que lo contiene
// Cuarto parámetro: alto del rectángulo que lo contiene

Ellipse2D.Double oval1 = new Ellipse2D.Double( 10, 10,
                                              ancho,
                                              alto );

// Con este método pedimos a la superficie que dibuje la
// figura rellena
g2.fill( oval1 );
  
```



```
// Primer parámetro: coordenada X del origen
// Segundo parámetro: coordenada Y del origen
// Tercer parámetro: coordenada X del destino
// Cuarto parámetro: coordenada Y del destino

Line2D.Double lin1 = new Line2D.Double( 10, 10, 100, 100);

// Con este método pedimos a la superficie que dibuje la
// línea

g2.draw( lin1 );
```



En el CD que acompaña al libro puede encontrar una herramienta que le permite utilizar de manera interactiva los métodos de dibujo de Java.

Tarea 9



Objetivo: Estudiar la clase `BasicStroke`, que permite configurar la línea que bordea las figuras geométricas que dibujamos.

Consulte la documentación (Javadoc) de la clase `BasicStroke` y conteste las preguntas que se plantean a continuación:

1. ¿Qué interfaz implementa la clase?

2. ¿Qué hace el método `setStroke()` de la clase `Graphics2D`?

3. Para cada uno de los parámetros del constructor de la clase, explique su uso y los valores que puede tomar:

Parámetro	Tipo	Valores posibles	Descripción
width			
cap			
join			

miterlimit			
dash			
dash_phase			

Vamos ahora a recorrer las distintas clases y métodos del editor de dibujos, para estudiar la manera como se utilizaron las clases y métodos antes descritos para construir la solución.

Tarea 10


Objetivo: Recorrer el programa que resuelve el problema planteado en el caso de estudio.

Siga los pasos que se dan a continuación y conteste las preguntas que se plantean.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.

2. En la clase `Línea`. ¿Cómo está implementado el método de dibujo? ¿Qué métodos antes estudiados utiliza?

3. En la clase `Línea`. ¿Cómo está implementado el método que indica si un punto se encuentra en su interior? ¿Para qué se utiliza la constante `distanciaMinima`?

4. En la clase `Rectángulo`. ¿Cómo está implementado el método de dibujo? ¿Cómo hace este método para indicar el caso en el cual el rectángulo se encuentra seleccionado?

5. En la clase `Rectángulo`. ¿Cómo está implementado el método que indica si un punto se encuentra en su interior?

6. En la clase Ovalo. ¿Cómo está implementado el método de dibujo? ¿Cómo hace este método para indicar el caso en el cual el óvalo se encuentra seleccionado?		
7. Estudiando los métodos encargados de la persistencia en las clases Figura y FiguraRellena, indique el formato con el cual se almacena la información de cada una de las figuras.	Línea	
	Óvalo	
	Rectángulo	
8. ¿Quién utiliza y para qué el método dibujarTexto() de la clase Figura? ¿Por qué se puede decir que es un método de servicio?		

3.15. Evolución del Editor

En esta sección vamos a trabajar en extensiones del editor de dibujos, para validar así la facilidad de evolución del programa. Para cada tarea vamos a seguir cuatro etapas: (1) hacer el análisis de la modificación pedida (entender el problema que se plantea con el

cambio), (2) identificar el punto del diseño que se debe modificar y el impacto que tiene sobre el resto del programa, (3) diseñar la extensión, construyendo el diagrama de clases y asignando, entre todos los componentes, las responsabilidades que acaban de aparecer, e (4) implementar y probar lo diseñado en el punto anterior.

Tarea 11

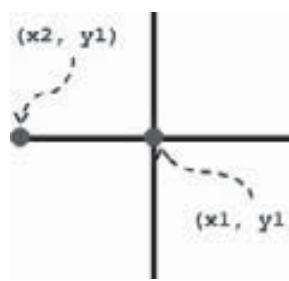


Objetivo: Probar la extensibilidad del diseño del programa, agregando una nueva figura al editor del caso de estudio.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.

2. Vamos a agregar al editor una nueva figura, que consiste en una cruz como la que aparece a continuación, y cuyas características se definen de la siguiente manera:



- La cruz está compuesta por dos líneas, una vertical y otra horizontal.
- Las líneas se cortan exactamente en la mitad (todos los lados de la cruz deben tener la misma medida).
- El valor y_2 no es utilizado, y se usa y_1 en los dos casos, para obtener siempre la cruz en una posición horizontal.
- Para crear la cruz el usuario debe primero seleccionar el botón con la figura y, luego, hacer clic en el centro de la cruz seguido de un clic sobre el punto en el extremo izquierdo de la línea horizontal.

3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta.

4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar la nueva figura.

Tarea 12

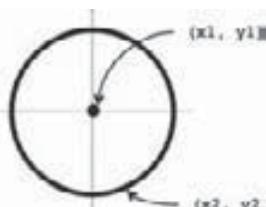


Objetivo: Probar la extensibilidad del diseño del programa, agregando una nueva figura al editor del caso de estudio.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.

2. Vamos a agregar al editor una nueva figura, que consiste en una circunferencia como la que aparece a continuación:



- Para definir una circunferencia el usuario debe seleccionar desde la interfaz de usuario el botón con la respectiva figura, hacer clic en el centro (x_1 , y_1) y luego hacer clic sobre cualquier punto que se encuentre sobre la circunferencia.
- Para seleccionar la figura, el usuario debe hacer clic sobre la circunferencia (no es una figura rellena).
- Se utilizan las mismas características de las demás figuras para manejar el ancho y el tipo de la línea.

3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta.

4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar la nueva figura.

Tarea 13



Objetivo: Probar la extensibilidad del diseño del programa, agregando un nuevo requerimiento funcional.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
 2. Asocie con el botón **Opción 1** un nuevo requerimiento funcional, que permita desplazar la figura seleccionada cinco píxeles a la derecha cada vez que se oprima.
 3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta.

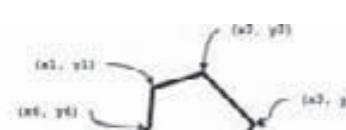
4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar el nuevo requerimiento.

Tarea 14



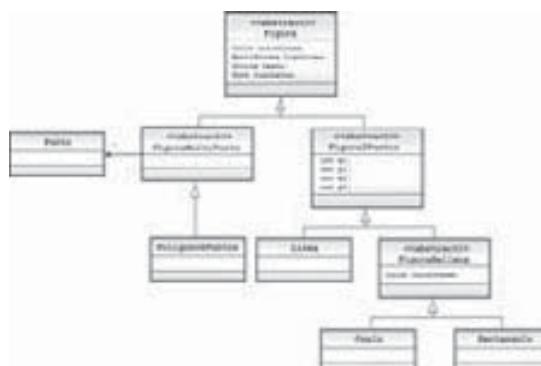
Objetivo: Probar la extensibilidad del diseño del programa, agregando una nueva familia de figuras al editor del caso de estudio.

Modifique el programa siguiendo las etapas que se plantean a continuación.

- Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
 - Vamos a agregar al editor una nueva figura, que consiste en un polígono de seis puntos, como el que aparece a continuación:

The diagram shows a polygon with six vertices labeled (x_1, y_1) through (x_6, y_6) . The vertices are connected by straight segments. Some segments have small wavy lines near them, indicating they are selected or highlighted.
 - Para definir un polígono el usuario debe seleccionar desde la interfaz de usuario el botón con la respectiva figura, y luego hacer seis veces clic señalando los vértices del polígono.
 - Para seleccionar la figura, el usuario debe hacer clic sobre el polígono (no es una figura rellena).
 - Se utilizan las mismas características de las demás figuras para manejar el ancho y el tipo de la línea.
 - Elimine de la clase `Figura` los dos puntos que definen la geometría de la misma (atributos `x1, y1, x2, y2`). Cree la clase abstracta `Figura2Puntos`, que hereda de la clase `Figura` y que tiene los cuatro atributos antes mencionados. Ajuste la herencia de las clases `Línea` y `FiguraRellena` para que ahora hereden de la clase `Figura2Puntos` y que todo siga funcionando correctamente.

4. Cree la clase **Punto**, que tiene como atributos una coordenada **x** y una coordenada **y**. Agréguele a esa clase un constructor, dos métodos para tener acceso al valor de cada coordenada (**darX()** y **darY()**) y dos métodos para modificar dichos valores (**cambiarX()** y **cambiarY()**).
5. Cree la clase **FiguraMultiPunto**, que hereda de la clase **Figura**, y que tiene una contenedora de objetos de la clase **Punto**. Agregue los métodos necesarios para que esta clase siga cumpliendo los mismos requerimientos del resto de las figuras (persistencia e invariante, por ejemplo).
6. Cree la figura **Polygono6Puntos**, que hereda de la clase **FiguraMultiPunto**. Haga todas las modificaciones necesarias en el programa para que funcione correctamente el editor con este nuevo tipo de figuras. El diagrama de clases que se debe obtener después de todas las modificaciones es el siguiente:

**Tarea 15**

Objetivo: Probar la extensibilidad del diseño del programa, agregando un nuevo requerimiento funcional.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo **n10_paint.zip**, el cual está disponible en el CD que acompaña al libro o en el sitio web.
2. Agregue un nuevo requerimiento funcional al editor para que, cuando el usuario haga clic de nuevo sobre la figura seleccionada, aparezca una ventana en la que se incluya toda la información de la figura, incluida su área.
3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta.
4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar el nuevo requerimiento.

Tarea 16

Objetivo: Probar la extensibilidad del diseño del programa, agregando un nuevo requerimiento funcional.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
 2. Asocie con el botón **Opción 2** un nuevo requerimiento funcional, que cambie a mayúsculas los textos de todas las figuras del dibujo.
 3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta.
-
4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar el nuevo requerimiento.

Tarea 17

Objetivo: Probar la extensibilidad del diseño del programa, agregando un nuevo requerimiento funcional.

Modifique el programa siguiendo las etapas que se plantean a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n10_paint.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
 2. Asocie con el botón **Opción 3** un nuevo requerimiento funcional, que permita cambiar el color de la línea que enmarca la figura. Para esto el programa debe presentar una ventana de selección de color.
 3. Explique las modificaciones que va a hacer en el programa y justifique claramente su respuesta.
-
4. Siga los cuatro pasos planteados al comienzo de la sección para hacer esta modificación. Recuerde que es importante hacer el menor esfuerzo posible por implementar el nuevo requerimiento.

4. Glosario de Términos

GLOSARIO



Complete las siguientes definiciones como una manera de resumir los conceptos aprendidos en el nivel. Asegúrese de entender bien el significado de cada uno de estos términos, puesto que son la base para poder continuar con los niveles que siguen en el libro.

Interfaz:

Herencia:

Superclase:

Subclase:

Menú:

Desacoplamiento:

Clase Object:

Interfaz List:

Interfaz Collection:

Interfaz Iterator:

Reutilización:

Polimorfismo y asociación dinámica:

Conversión de tipos (*casting*):

Atributos protegidos:

Redefinición de un método:

La variable super:

Clase abstracta:

Método abstracto:

Extensibilidad:

Interfaz MouseListener:

Interfaz Shape:

Método paintComponent:

5. Hojas de Trabajo



5.1. Hoja de Trabajo N° 1: Mapa de la Ciudad

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

Se desea construir un programa que permita la creación de un mapa sencillo de una ciudad. Dicho mapa está constituido por construcciones que comparten características comunes.

Todas las construcciones tienen unas dimensiones (alto y ancho, medidas en píxeles), un color de fondo, un punto (x, y), que corresponde a la esquina superior izquierda donde se va a ubicar

la construcción, y un texto que puede ser modificado y visualizado.

Las construcciones se encuentran divididas en edificaciones y carreteras. Una edificación puede ser una casa, un edificio, un hospital, una estación de policía o una estación de bomberos. Las carreteras pueden ser calles, carreras, glorietas y esquinas. A continuación se muestran esos elementos de la ciudad:

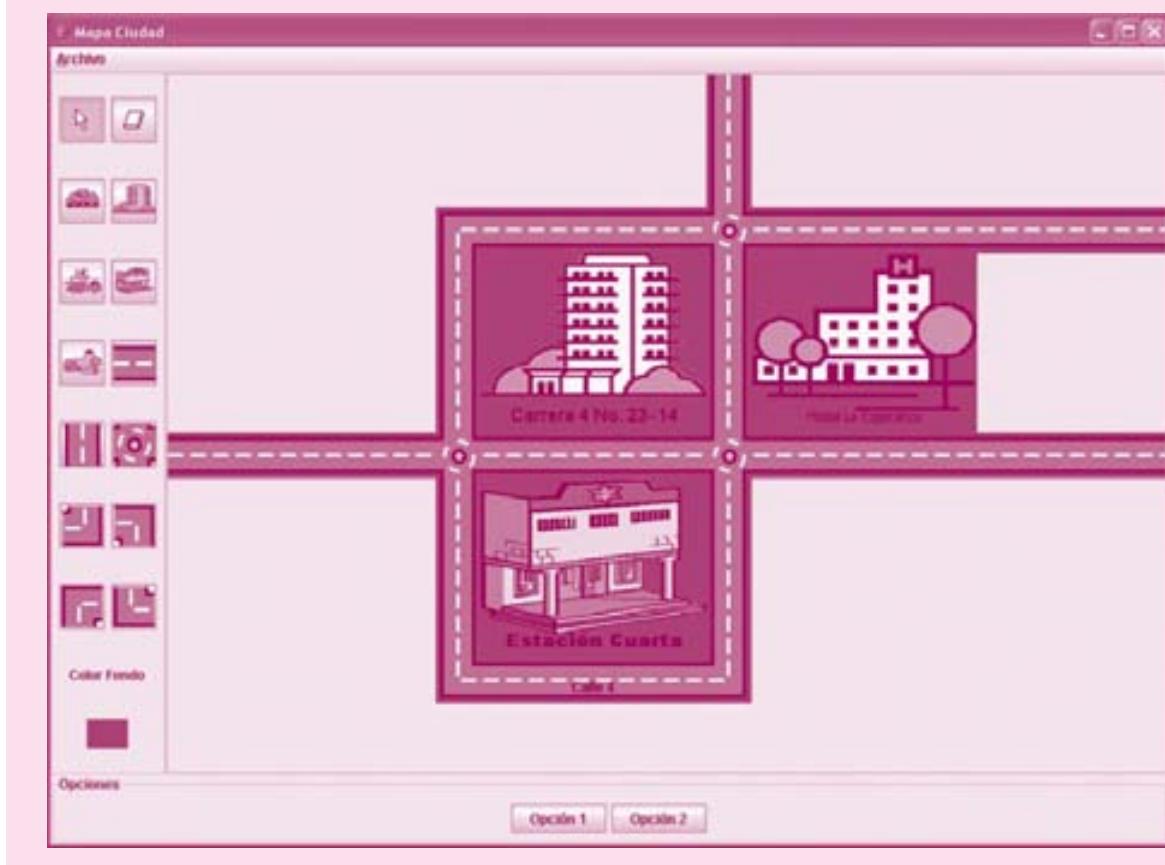
Nombre	Figura	Archivo	Alto	Ancho
Casa		Casa.gif	120	160
Edificio		Edificio.gif	160	200
Hospital		Hospital.gif	160	200
Estación de policía		Estación de policía.gif	160	200
Estación de bomberos		Estación de bomberos.gif	160	200
Calle		(no se debe dibujar)	40	40
Carretera		(no se debe dibujar)	40	40
Glorieta		(no se debe dibujar)	40	40
Esquina 1		(no se debe dibujar)	40	40

Esquina 2		(no se debe dibujar)	40	40
Esquina 3		(no se debe dibujar)	40	40
Esquina 4		(no se debe dibujar)	40	40

Para dibujar una construcción en el mapa se debe seleccionar el tipo de construcción entre las opciones disponibles y localizar el ratón en la posición del mapa donde ésta se quiere ubicar. No se debe permitir que dos construcciones se sobrepongan. Para tal fin, si la ubicación seleccionada está libre, se debe mostrar la silueta de la construcción. Para crear la construcción basta hacer clic sobre la zona del mapa seleccionada. Para borrar una construcción o modificar el texto asociado, el usuario debe seleccionarla antes, haciendo clic sobre ella.

El programa debe permitir (1) agregar una construcción al mapa, (2) borrar del mapa la construcción seleccionada, (3) cambiar el texto que describe la construcción seleccionada, personalizando sus atributos: estilo, tamaño y tipo de fuente; (4) guardar el mapa que se acaba de construir en un archivo y (5) cargar un mapa existente.

La interfaz de usuario que debe tener el programa es la siguiente:



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

Requerimiento funcional 1	Nombre	R1 – Agregar una construcción al mapa
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Borrar una construcción del mapa
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Cambiar el texto que describe una construcción
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Guardar el mapa en un archivo
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Cargar un mapa
	Resumen	
	Entrada	
	Resultado	

Modelo conceptual. Construya el diagrama inicial de clases con los cinco elementos que se describen a continuación y las asociaciones que los relacionan: (1) la clase `MapaCiudad`, que representa un mapa completo; (2) la interfaz `IConstruccion`, que define el contrato funcional de todas las construcciones (por ahora no le defina ningún método); (3) la clase abstracta `Construccion`, que incluye todas las características compartidas de las construcciones; (4) la subclase `Edificacion` (hereda de la clase `Construccion`), que incluye las características propias de este tipo de elementos, y (5) la subclase `Carretera` (hereda también de la clase `Construccion`).

Declaración de clases. Escriba en Java la declaración de cada una de las clases del diagrama anterior. El grupo de elementos de tipo `IConstruccion` que tiene la clase `MapaCiudad` debe ser definido utilizando la interfaz `List`.

```
public class MapaCiudad  
{
```

```
}
```

```
public abstract class Construccion  
{
```

```
}
```

Interfaces. Estudie el contrato funcional de la interfaz `IConstruccion` y llene la siguiente tabla con la descripción de los métodos. Para esto cree un proyecto en Eclipse a partir del archivo `n10_mapaCiudad.zip`, que se encuentra en el CD que acompaña al libro. Localice el archivo `IConstruccion.java` y analice su contenido. Asegúrese de entender la responsabilidad de cada método.

Implementación. A continuación se proponen algunas tareas que van a permitir al lector estudiar la solución de esta parte del programa.

- Si no lo ha hecho antes, cree un proyecto en Eclipse a partir del archivo `n10_mapaCiudad.zip`, el cual se encuentra disponible en el CD que acompaña al libro. Localice la clase `InterfazMapaCiudad` y ejecute desde allí el programa. Utilice las distintas opciones disponibles para crear una ciudad.
- Vamos a comenzar estudiando la clase `MapaCiudad`. Para esto edite el respectivo código desde Eclipse y haga el recorrido que se plantea a continuación por medio de preguntas:

Localice el método llamado
`pintarConstrucciones()`. ¿Cuál es su responsabilidad? ¿Qué recibe como parámetro?

Localice el método llamado
`eliminarConstrucción()`. ¿Cuál es su responsabilidad? ¿Qué recibe como parámetro? ¿Cómo localiza la construcción que debe eliminar?

Localice el método llamado
`buscarConstrucción()`. ¿Cuál es su responsabilidad? ¿Qué recibe como parámetro?

Localice el método llamado
`sobreponeConstrucción()`. ¿Cuál es su responsabilidad? ¿Qué recibe como parámetro? ¿Para qué se utiliza la constante `DIFERENCIA`? ¿Qué condiciones verifica?

¿Qué condiciones exige el invariante de la clase?

Edite uno de los archivos en los que se almacenan los mapas (directorio "data"). Estudie la estructura de este archivo y haga un resumen de su contenido.

¿Qué método de la clase es el encargado de leer un mapa de un archivo? ¿Qué responsabilidades asume y cuáles delega en otras clases? ¿Cómo sabe de qué clase debe construir la instancia?	
¿Qué método de la clase es el encargado de escribir un mapa en un archivo? ¿Qué método de la interfaz <code>IConstrucion</code> utiliza?	
3. Pasemos ahora a la clase <code>Construcion</code> :	
¿Cuántos constructores tiene? ¿En qué casos se utiliza cada uno de ellos?	
¿En qué coordenadas dibuja el texto el método <code>pintarTexto()</code> ? ¿Para qué se utiliza la clase <code>FontMetrics</code> ? ¿Quién invoca este método? ¿Por qué hace parte de esta clase?	
¿Por qué el método <code>pintar()</code> está declarado como abstracto?	
¿Por qué el método <code>darTipo()</code> está declarado como abstracto?	
¿Quién invoca los métodos <code>pintarSombreada()</code> y <code>pintarSeleccionada()</code> ? ¿En qué caso se invoca cada uno de ellos?	
¿Qué condiciones se verifican en el invariante de la clase?	
4. Edite la clase <code>Edificacion</code> :	
¿Qué atributos adicionales define esta clase? ¿Qué características modelan?	
¿Cuántos constructores tiene? ¿En qué casos se utiliza cada uno de ellos?	
¿Cómo se implementa el método <code>pintar()</code> en esta clase?	
¿Por qué se declara como abstracta esta clase?	

5. Estudie ahora la clase **Carretera**:

¿Qué atributos adicionales define esta clase? ¿Qué sentido tiene entonces definirla?

¿Cuántos constructores tiene? ¿En qué casos se utiliza cada uno de ellos?

¿Por qué no hay una implementación del método `pintar()` en esta clase?

Diagrama de clases. Dibuje el diagrama de clases con la jerarquía de herencia de la clase **Edificacion**.



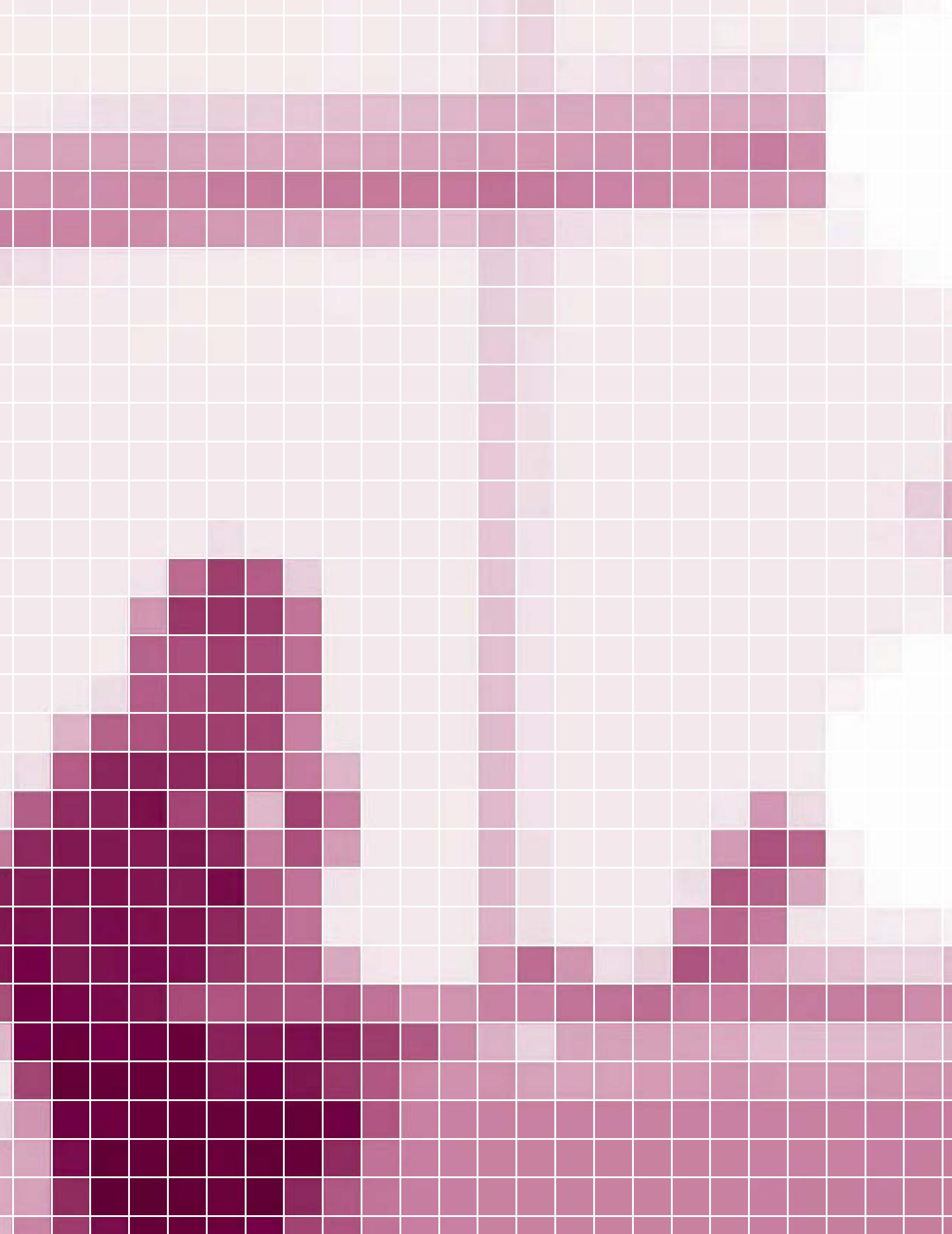
Diagrama de clases. Dibuje el diagrama de clases con la jerarquía de herencia de la clase **Carretera**.



6. Para cada una de las clases que se dan a continuación, describa los métodos que implementan y la forma en que lo hacen.

Calle	
Carrera	
Casa	
Edificio	
Esquina1	
Esquina2	

Esquina3		
Esquina4		
EstacionBomberos		
EstacionPolicia		
Glorieta		
Hospital		



Nivel 5

Estructuras y Algoritmos Recursivos

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

- Utilizar estructuras recursivas de datos para representar la información del modelo del mundo, cuando sea conveniente por razones de eficiencia o por el tipo de problema sobre el cual se trabaja.
- Escribir algoritmos recursivos para manipular estructuras de información recursivas y explicar las ventajas que, en este caso, estos algoritmos tienen sobre los algoritmos iterativos.
- Utilizar árboles binarios ordenados para representar grupos de objetos que mantienen entre ellos una relación de orden.
- Utilizar árboles n-arios, como una estructura que permite representar información de un mundo en el cual existe una relación jerárquica entre sus elementos.

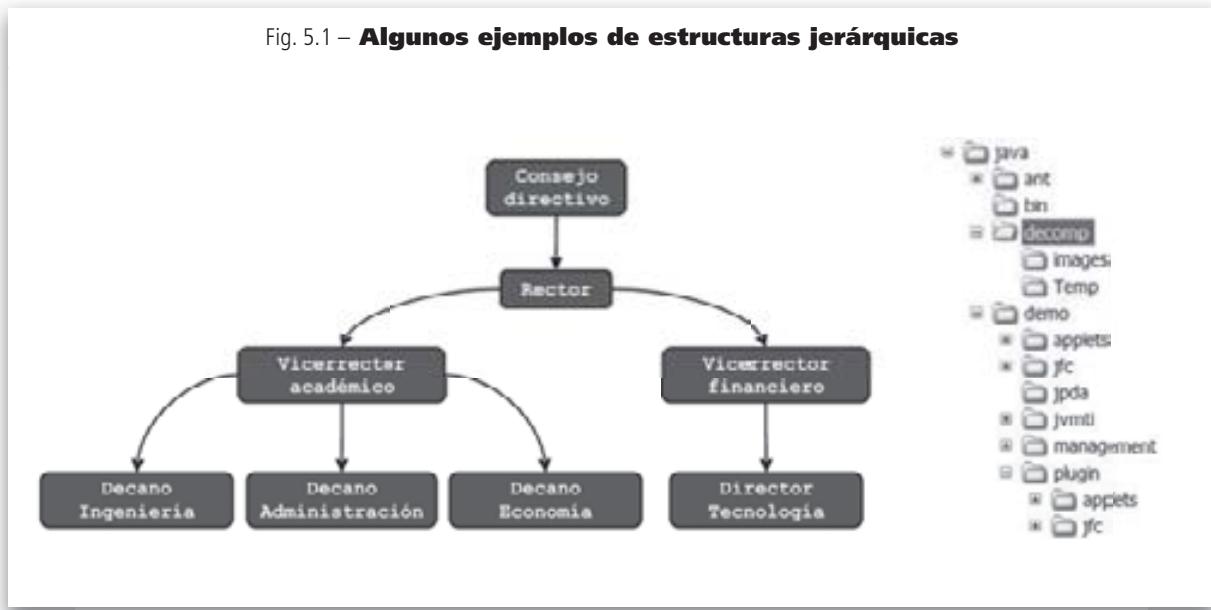
2. Motivación

La eficiencia de los algoritmos que podemos implementar para resolver un problema se encuentra fuertemente ligada con las estructuras de datos que utilizemos para representar el modelo del mundo. En el nivel 1, por ejemplo, estudiamos la búsqueda binaria, un algoritmo muy eficiente para buscar un valor en un arreglo que se encuentra ordenado. Si el arreglo no se encuentra ordenado o si utilizamos una lista enlazada para almacenar la información, no vamos a encontrar (porque no existe) ningún algoritmo así de eficiente para hacer las búsquedas. El problema con la búsqueda binaria es que los métodos de inserción y supresión de elementos son bastante inefficientes, puesto que deben desplazar muchos elementos para mantener siempre el orden dentro del arreglo. Lo ideal sería encontrar una estructura que nos permitiera hacer estas tres operaciones con tiempos de respuesta cortos. Para resolver el problema de manejar grupos ordenados de objetos, podemos contar

con una familia completa de estructuras recursivas de datos, con múltiples variantes e implementaciones, denominadas árboles. Es importante señalar que estas estructuras y la algorítmica que tienen asociada juegan un papel fundamental en la construcción de programas. De esta familia de estructuras recursivas nos vamos a interesar en este nivel en los árboles binarios ordenados, la estructura más simple que allí encontramos, la cual permite enlazar la información de tal manera que las operaciones de búsqueda, inserción y supresión se pueden implementar de manera eficiente.

El único uso de los árboles no es representar información ordenada. También es frecuente que se utilicen para almacenar elementos que tienen una relación de jerarquía entre ellos. Piense, por ejemplo, en el organigrama de una universidad, en el árbol genealógico de una familia o en el sistema de archivos de un computador. Algunas de estas estructuras se ilustran en la figura 5.1.

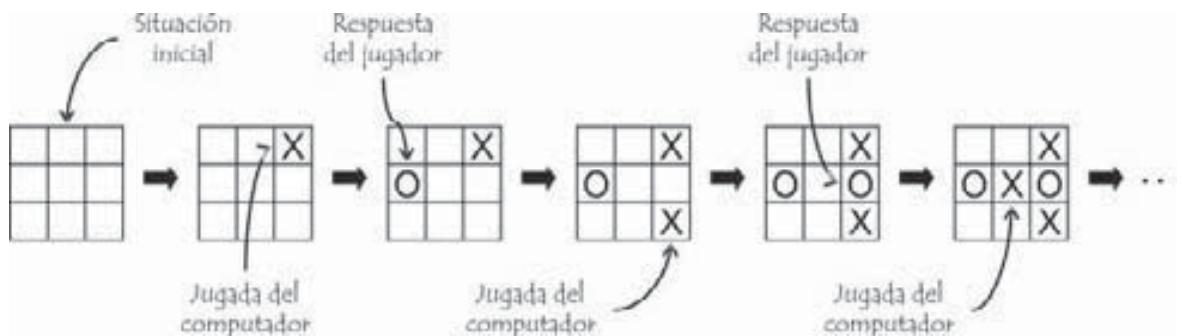
Fig. 5.1 – **Algunos ejemplos de estructuras jerárquicas**



En algunos casos, los árboles también se pueden utilizar para representar el espacio de soluciones posibles de un problema. Considere, por ejemplo, el juego de triqui,

en el cual un jugador se enfrenta al computador para intentar completar con su marca una fila, una columna o una diagonal, tal como se muestra en la figura 5.2.

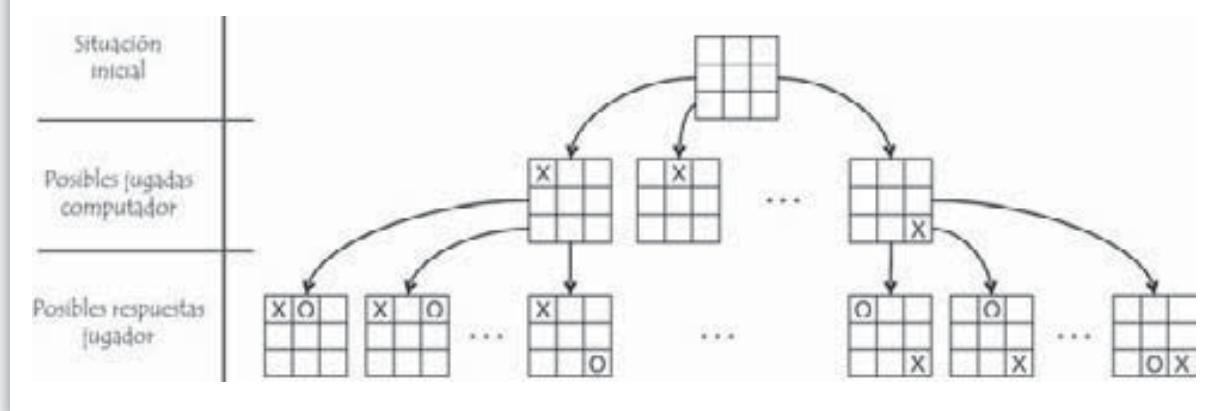
Fig. 5.2 – Secuencia de jugadas en una partida de triqui



Si tratamos de definir una buena estructura de datos para que el computador pueda seleccionar la mejor jugada en un momento dado del juego, nos podemos imaginar una estructura enlazada como la que se sugiere en la figura 5.3, en la cual cada estado posible del juego está relacionado con los estados a los cuales

puede llegar con una jugada. El computador debe hacer búsquedas sobre dicha estructura tratando de identificar la jugada que más le conviene hacer (la que lo lleva a un estado en donde haya más posibilidades de ganar). Esta estructura jerárquica se denomina un **árbol n-ario**, y será también estudiada en este nivel.

Fig. 5.3 – Fragmento del árbol n-ario para el juego de triqui



Los algoritmos para manipular árboles pueden resultar un poco complicados en algunos casos, por lo que resulta conveniente estudiar otro tipo de algoritmos que se adaptan mucho mejor a estas estructuras: los **algoritmos recursivos**. Debe ser claro que todo problema que se quiera resolver sobre cualquier tipo de árbol tiene tanto una solución con un algoritmo recursivo como una solución con un algoritmo iterativo (así

vamos a llamar a los algoritmos que hemos utilizado hasta ahora). Lo importante es decidir en cada caso particular la manera más conveniente de expresar la solución que se quiere construir. Típicamente, los algoritmos recursivos son más cortos y fáciles de escribir cuando se trata de árboles. En este nivel estudiaremos algunos aspectos metodológicos y algunos patrones simples para construir algoritmos recursivos.

3. Caso de Estudio Nº 1: Un Directorio de Contactos

El programa que queremos construir en este caso es un directorio en el cual el usuario pueda almacenar la información de sus contactos. Un contacto tiene cuatro datos: un nombre (que sin importar mayúsculas o minúsculas debe ser único), un teléfono, una dirección

postal y una dirección de correo electrónico. El directorio debe ofrecer al usuario las siguientes opciones: (1) agregar un nuevo contacto (sólo el nombre es obligatorio), (2) localizar un contacto dado su nombre, (3) desplegar la información de un contacto dado su nombre, (4) eliminar un contacto del directorio y (5) dar información estadística referente a las estructuras internas de información del directorio. En la figura 5.4 se muestra la interfaz de usuario que debe tener el programa.

Fig. 5.4 – **Interfaz de usuario del caso de estudio**



- El directorio puede estar en dos modos: "modo búsqueda" o "modo inserción", los cuales se seleccionan con el respectivo botón de la parte inferior de la ventana.
- En "modo inserción" sólo está activo el botón que permite agregar contactos al directorio. Para hacerlo se debe teclear la información de la persona y oprimir dicho botón.
- La lista de la parte izquierda de la ventana muestra los nombres de todos los contactos que el usuario tiene registrados en el directorio. Esta lista está ordenada ascendente siguiendo el orden alfabético.
- Cuando el programa está en "modo búsqueda" y el usuario hace clic sobre algún contacto de la lista, su información es desplegada en la ventana.
- En "modo búsqueda" también es posible localizar a una persona dando su nombre completo o eliminarla del directorio.
- Hay un botón de consulta de estadísticas, con el cual se despliega por pantalla información sobre el estado de las estructuras de datos que implementan el directorio. La información que se muestra depende del tipo de estructuras que se utilice.

Hay tres puntos adicionales que se deben tener en cuenta para el diseño del programa: (1) la información no debe ser persistente, (2) las estructuras de datos escogidas para la implementación deben garantizar eficien-

cia en las consultas, las inserciones y las supresiones de contactos, y (3) debe estar desacoplada la interfaz de usuario de las estructuras específicas que se diseñen, para facilitar así la evolución del programa.

3.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none"> ■ Definir un diseño en el cual la interfaz de usuario no dependa de las estructuras de datos que se seleccionen para almacenar los contactos. 	<ul style="list-style-type: none"> ■ Repasar la manera de utilizar interfaces para desacoplar las distintas partes de un programa.
<ul style="list-style-type: none"> ■ Definir unas estructuras de datos en las cuales los procesos de consulta y modificación se puedan hacer de manera muy eficiente. 	<ul style="list-style-type: none"> ■ Estudiar los árboles binarios ordenados, sus características y propiedades. ■ Estudiar la algorítmica de manipulación de árboles binarios. ■ Aprender a construir algoritmos recursivos, para que el desarrollo de los métodos que manejan los árboles resulte más sencillo.
<ul style="list-style-type: none"> ■ Transformar el diagrama de clases del análisis en un diseño que incluya árboles. 	<ul style="list-style-type: none"> ■ Repasar la manera de pasar del diagrama de clases del análisis al diagrama de clases del diseño.

3.2. Comprensión de los Requerimientos

Tarea 1									
	Objetivo: Entender el problema del caso de estudio del directorio de contactos. (1) Lea detenidamente el enunciado del caso de estudio del directorio de contactos e (2) identifique y complete la documentación de los requerimientos funcionales.								
Requerimiento funcional 1	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%;">Nombre</td><td>R1 – Agregar un contacto</td></tr> <tr> <td>Resumen</td><td></td></tr> <tr> <td>Entrada</td><td></td></tr> <tr> <td>Resultado</td><td></td></tr> </table>	Nombre	R1 – Agregar un contacto	Resumen		Entrada		Resultado	
Nombre	R1 – Agregar un contacto								
Resumen									
Entrada									
Resultado									

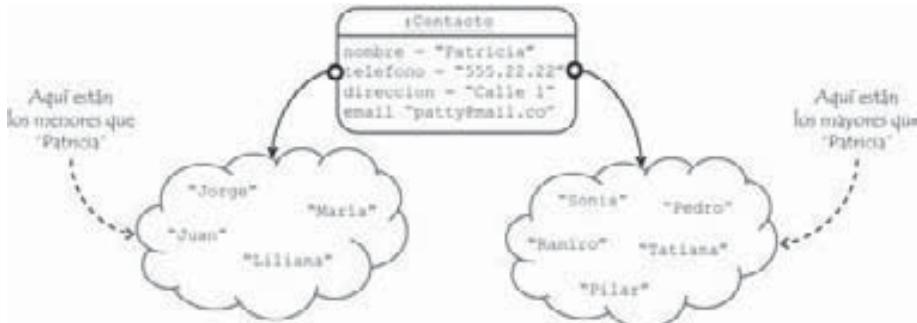
Requerimiento funcional 2	Nombre	R2 – Localizar un contacto
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Desplegar la información de un contacto
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Eliminar un contacto
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Dar información estadística
	Resumen	
	Entrada	
	Resultado	

3.3. Árboles Binarios Ordenados

Un árbol binario ordenado se implementa usualmente mediante una estructura enlazada, como la que se sugiere, para el caso del directorio de contactos, en la figura 5.5. En dicha estructura vamos a repartir en dos grupos los elementos que queremos almacenar (10 personas). Los que son menores que el elemento que

estamos mostrando ("Patricia") los vamos a situar en algún punto a partir del enlace izquierdo, mientras que los mayores alfabéticamente van a quedar situados en algún punto a partir del enlace derecho. De esta manera, cuando estemos intentando localizar a una persona en el árbol, basta con una sola comparación para encontrar el grupo en el que debemos continuar el proceso, descartando así un grupo grande de elementos.

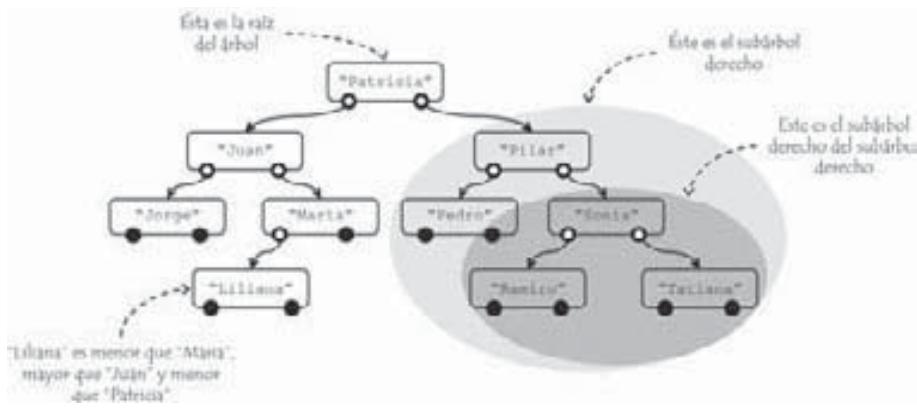
Fig. 5.5 – Estructura básica de un árbol binario ordenado



Luego, con cada uno de los grupos obtenidos podemos repetir el mismo proceso, separando los elementos que son menores a un elemento dado de los elementos que son mayores, y llegando así a una estructura de datos que podría ser la que se muestra en la figura 5.6. Si estamos buscando, por ejemplo, a "Liliana", avanzamos por el enlace izquierdo de "Patricia" ("Li-

liana" < "Patricia"), luego por el enlace derecho de "Juan" ("Liliana" > "Juan"), después por el enlace izquierdo de "María" ("Liliana" < "María"), para llegar finalmente al contacto que estamos tratando de localizar en el árbol. Si el árbol tiene sus elementos bien distribuidos, logramos la misma eficiencia de la búsqueda binaria.

Fig. 5.6 – Posible árbol binario ordenado de contactos



Ahora sí veamos las definiciones formales de la estructura. Un árbol binario ordenado está constituido por un elemento denominado la **raíz** y por dos estructuras que se le asocian, que son a su vez árboles binarios ordenados. Estas dos estructuras asociadas se denominan el **subárbol izquierdo** y el **subárbol**

derecho. Un árbol binario sin elementos (y por lo tanto sin subárboles asociados) se denomina **vacío**. En el árbol de la figura 5.6, los subárboles del contacto con nombre "Tatiana" son ambos vacíos. En el mismo árbol, el subárbol derecho del contacto "María" es vacío, pero no el izquierdo.



Una estructura de datos es recursiva si está constituida por elementos más pequeños, que tienen exactamente la misma estructura. En el caso de los árboles binarios, cada uno de ellos "incluye" en su interior dos árboles binarios, que a su vez tienen dos árboles binarios, cada uno de los cuales tiene dos árboles binarios...

La ventaja de esta estructura es que cada vez que avanzamos en el proceso de búsqueda de un elemento, descartamos un grupo grande de candidatos. En una lista, en cambio, en cada iteración que hacemos, descartamos únicamente uno de los elementos del grupo. Esto lo veremos en detalle un poco más adelante.

3.3.1. Algunas Definiciones

En esta sección presentamos algunas definiciones que se utilizan en el contexto de los árboles binarios. Éstas se ilustran a medida que se presentan, usando para esto el árbol de la figura 5.6.

- El **peso** de un árbol corresponde al número de elementos que tiene la estructura. En el ejemplo, el peso del árbol es 10 (hay 10 contactos en el directorio).

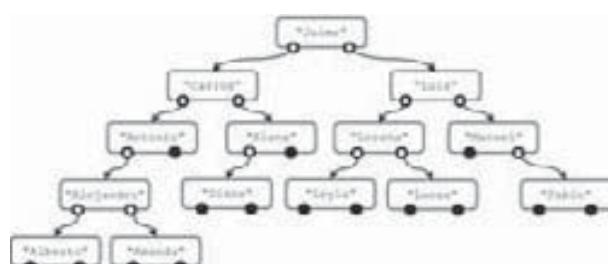
- El elemento que se encuentra en la raíz de un árbol se dice que está en el **nivel 1** del mismo. Las raíces de sus subárboles, denominados sus **hijos**, se encuentran en el **nivel 2**. En el ejemplo, "Patricia" está en el nivel 1, mientras "Juan" y "Pilar" están en el nivel 2. Como la definición es recursiva, encontramos a "Tatiana" en el cuarto y último nivel del árbol. También tenemos que "Jorge" es el hijo de "Juan" (se dice entonces que "Juan" es el padre de "Jorge") y que "Liliana" es el hijo de "María".
- La altura de un árbol es el número de niveles que éste tiene. En el ejemplo, el árbol tiene altura 4. Un árbol de altura alt puede tener a lo sumo un peso de $2^{alt} - 1$ elementos. Esto quiere decir que, en el árbol del ejemplo, podemos almacenar hasta 15 contactos antes de tener que aumentar un nivel.
- Una hoja es un árbol cuyos subárboles asociados son vacíos. El árbol del ejemplo tiene cinco hojas, que corresponden a los contactos llamados "Jorge", "Liliana", "Pedro", "Ramiro" y "Tatiana".
- Recorrer un árbol es el proceso de pasar una vez sobre cada uno de los elementos de la estructura. El recorrido en **inorden** en un árbol binario ordenado es aquél que pasa sobre cada uno de los elementos del árbol respetando la relación de orden que existe entre ellos. El recorrido en inorden del árbol del ejemplo es: "Jorge", "Juan", "Liliana", "María", "Patricia", "Pedro", "Pilar", "Ramiro", "Sonia", "Tatiana".

Tarea 2

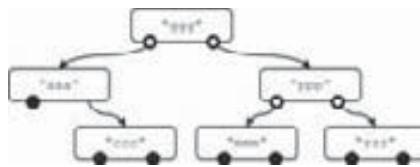


Objetivo: Aplicar las definiciones anteriores sobre árboles concretos, con el fin de reforzar las explicaciones dadas.

Para los árboles que se presentan, determine los valores que se piden.



Peso =	Altura =
Elementos nivel 1 =	Elementos nivel 2 =
Elementos nivel 3 =	Hojas =
Hijos de "Lorena" =	Padre de "Lucas" =
Recorrido en inorden =	



Peso =	Altura =
Elementos nivel 1 =	Elementos nivel 2 =
Elementos nivel 3 =	Hojas =
Hijos de "ggg" =	Padre de "ccc" =
Recorrido en inorden =	

3.3.2. Proceso de Búsqueda

El proceso de búsqueda en un árbol binario ordenado comienza en su raíz y se va moviendo por los enlaces izquierdo o derecho según si el elemento que estamos buscando es menor o mayor que aquél que se encuentra en la raíz. Este proceso es recursivo, en el sentido de

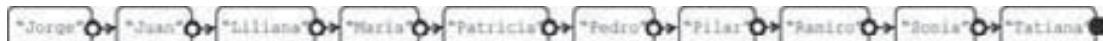
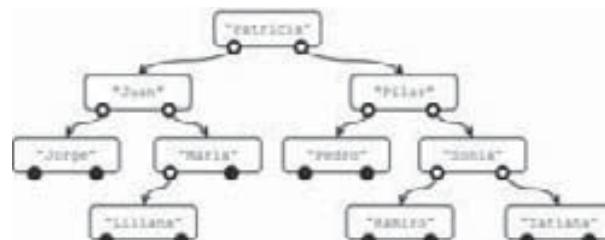
que se vuelve a aplicar idéntico cada vez que se llega al subárbol del siguiente nivel.

En la tarea 3 vamos a comparar la eficiencia de este proceso, contra la misma búsqueda cuando el directorio de contactos está representado con una lista enlazada ordenada ascendentemente.

Tarea 3

Objetivo: Comparar de manera informal la búsqueda de un elemento sobre un árbol binario ordenado y sobre una lista enlazada ordenada.

Considere las dos estructuras que se plantean a continuación para representar el directorio de contactos y haga los cálculos que se piden. Ambas estructuras contienen la misma información.



Contacto	Número de contactos que debe recorrer en el árbol antes de llegar a la respuesta	Número de contactos que debe recorrer en la lista antes de llegar a la respuesta
"Jorge"	3	1
"Juan"		
"Liliana"		
"María"		
"Patricia"		
"Pedro"		
"Pilar"		
"Sonia"		
"Tatiana"		
"Valentina" (no existe)		
Promedio		

¿Qué conclusiones puede sacar del ejercicio que acaba de hacer?



Si un árbol binario ordenado tiene una distribución adecuada (cada subárbol tiene aproximadamente la mitad de los elementos), el proceso de búsqueda en la estructura es tan eficiente como la búsqueda binaria sobre arreglos.

3.3.3. Proceso de Inserción

Para agregar un nuevo elemento a un árbol binario ordenado hay varias opciones. La más adecuada sería agregar el elemento en alguna posición de la estructura y después hacer los cambios necesarios en los otros elementos para que el árbol no pierda su equilibrio (dejar más o menos la misma cantidad de elementos en cada subárbol). Estos árboles se llaman árboles binarios balanceados, y serán el tema de un curso posterior. Por ahora vamos a insertar el elemento en el punto en el que resulte más sencillo hacerlo,

sabiendo que en algunos casos podemos perder el equilibrio del árbol y con esto disminuir la eficiencia en las búsquedas.

El proceso de inserción que vamos a utilizar consiste en localizar el punto del árbol en el que podemos agregar el nuevo elemento como una hoja. ¿Cómo podemos asegurar que ese punto siempre existe? La respuesta es simple. Podemos utilizar el algoritmo de búsqueda antes planteado e insertar el nuevo elemento en el punto en el que éste se da cuenta de que el elemento no existe. Este proceso se ilustra en el ejemplo 1.

Ejemplo 1



Objetivo: Ilustrar el proceso de inserción de elementos en un árbol binario ordenado.

En este ejemplo mostramos el proceso de inserción de elementos en un árbol binario ordenado, localizando el punto en el que se puede agregar el nuevo elemento como una hoja. Comenzamos con un árbol vacío y vamos insertando una secuencia de valores.

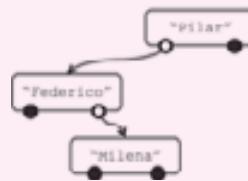
- El árbol inicialmente se encuentra vacío. Insertamos el contacto de nombre "Pilar". Este elemento queda como la raíz del árbol.



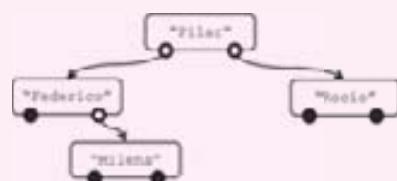
- Vamos a insertar el elemento "Federico". El proceso de búsqueda nos dice que debería estar en el subárbol izquierdo. Puesto que ese subárbol está vacío, podemos agregar el nuevo contacto en dicho punto como una nueva hoja.



- Ahora queremos insertar al árbol el contacto de nombre "Milena". El proceso de búsqueda nos lleva a que debería estar en el subárbol derecho de "Federico". Allí lo podemos agregar como una hoja, puesto que dicho árbol está vacío.

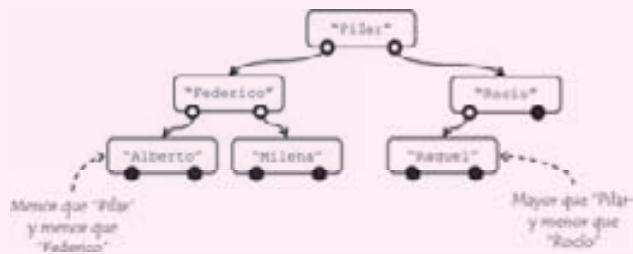


- El elemento de nombre "Rocío" se debe agregar como una hoja en el subárbol derecho de "Pilar".

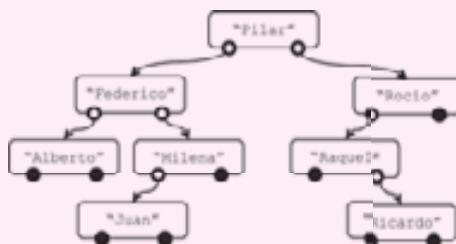




Ahora insertamos los elementos de nombre "Alberto" y "Raquel", quienes deben quedar en el subárbol izquierdo de "Federico" y "Rocío" respectivamente.



Finalmente insertamos los contactos de nombre "Ricardo" y "Juan", quienes deben quedar en el nivel 4 del árbol.



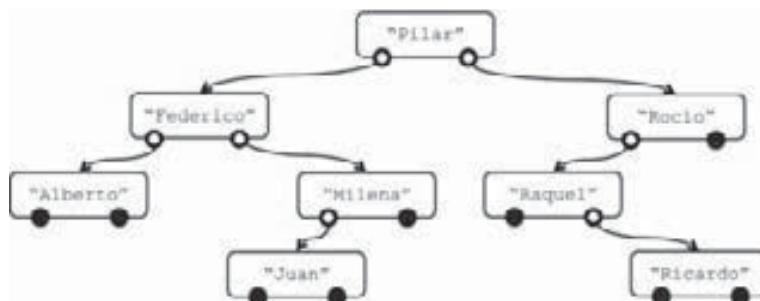
Tarea 4



Objetivo: Practicar el proceso de inserción en un árbol binario ordenado.

Muestre el árbol binario ordenado al cual se llega después de insertar la siguiente secuencia de elementos sobre el árbol al que se llegó en el ejemplo 1.

"Luisa" – "Penélope" – "Esteban" – "Hernando" – "Valentina" – "Anastasia"



3.3.4. Proceso de Supresión

Para eliminar un elemento existen también distintas estrategias, de las cuales vamos a escoger la más sencilla. Ésta considera tres casos:

- Si el elemento es una hoja, eliminarlo es trivial, puesto que al removerlo no afectamos el resto de la estructura del árbol.
- Si el elemento no es una hoja, pero sólo tiene uno

de los dos subárboles (el otro es vacío), dejamos como árbol el subárbol que no es vacío.

- Si el elemento tiene asociados dos subárboles no vacíos, buscamos el menor elemento del subárbol derecho, lo pasamos a la raíz (lo puede remplazar, puesto que se siguen satisfaciendo las condiciones de orden) y luego eliminamos del subárbol derecho el valor que acabamos de mover.

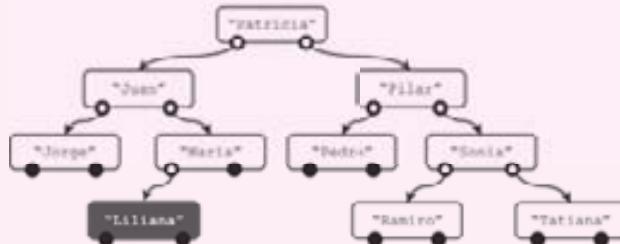
El proceso completo se ilustra en el ejemplo 2.

Ejemplo 2

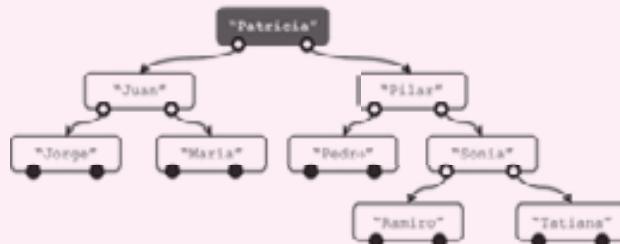
Objetivo: Ilustrar el proceso de supresión de elementos en un árbol binario ordenado.

En este ejemplo mostramos los tres casos que pueden aparecer en el momento de eliminar un elemento de un árbol binario ordenado.

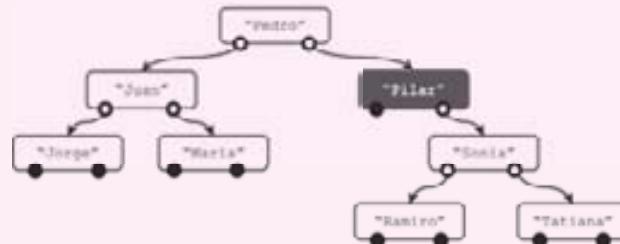
- Suponga que éste es el árbol inicial sobre el que vamos a eliminar elementos.



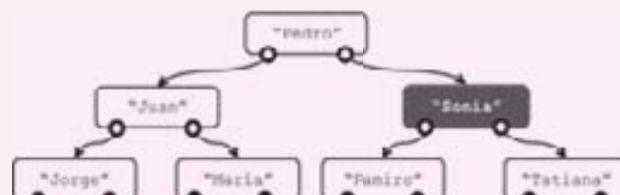
- Si eliminamos el elemento de nombre "Liliana", que es una hoja, estamos en el caso 1. La solución es simplemente quitarlo del árbol, puesto que no afecta al resto de la estructura.



- Si queremos eliminar el elemento de nombre "Patricia", llegamos al caso 3. La solución es buscar el menor del subárbol derecho ("Pedro"), pasarlo a la raíz del árbol y luego eliminar el elemento "Pedro" del subárbol derecho. El árbol sigue siendo ordenado, puesto que todos los elementos del subárbol derecho son mayores que "Pedro", y éste es a su vez mayor que todos los elementos del subárbol izquierdo.

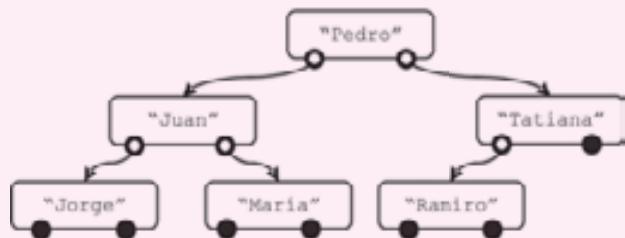


- Para eliminar el elemento de nombre "Pilar", aplicamos el caso 2, puesto que uno de sus subárboles es vacío. La solución consiste en subir un nivel el subárbol existente.





Finalmente, si queremos eliminar el elemento llamado "Sonia" (caso 3), buscamos el menor del subárbol derecho ("Tatiana"), hacemos el reemplazo por "Sonia" y luego eliminamos de la estructura el contacto de nombre "Tatiana".



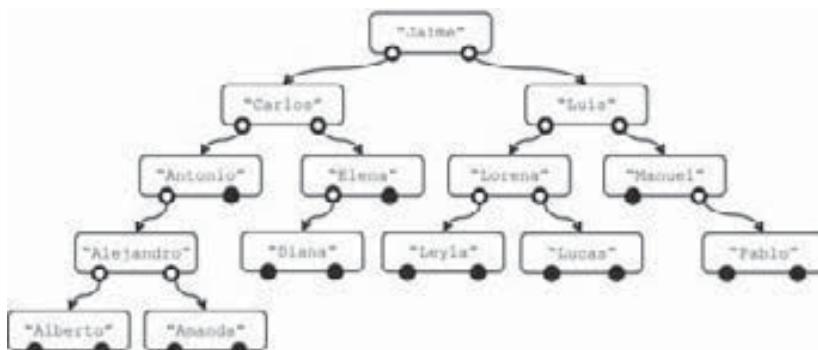
Tarea 5



Objetivo: Practicar el proceso de inserción en un árbol binario ordenado.

Muestre el árbol binario ordenado al cual se llega después de eliminar la siguiente secuencia de elementos, utilizando el algoritmo descrito en esta sección. Identifique en cada paso el caso al que pertenece.

"Jaime" – "Diana" – "Lorena" – "Antonio" – "Leyla" – "Alejandro".

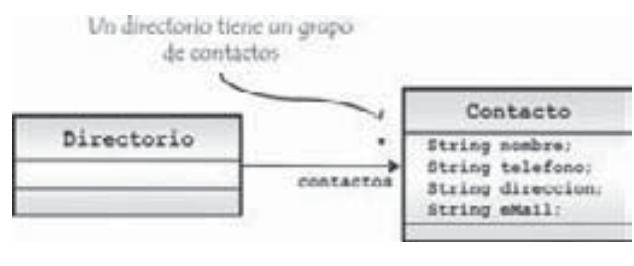


En el CD que acompaña al libro y en el sitio web puede encontrar un entrenador que permite animar los procesos de inserción y supresión de elementos en un árbol binario ordenado. En este punto se sugiere utilizarlo para reforzar las habilidades desarrolladas en las tareas anteriores.

3.4. Del Análisis al Diseño

Volvamos ahora a nuestro caso de estudio. Para comenzar, identifiquemos las entidades que hacen parte del mundo del problema y construyamos el diagrama de clases. En la figura 5.7 se puede ver que hay dos entidades: Directorio y Contacto, con una relación de cardinalidad múltiple entre ellas. Un contacto tiene cuatro atributos para representar su nombre, su teléfono, su dirección postal y su dirección electrónica.

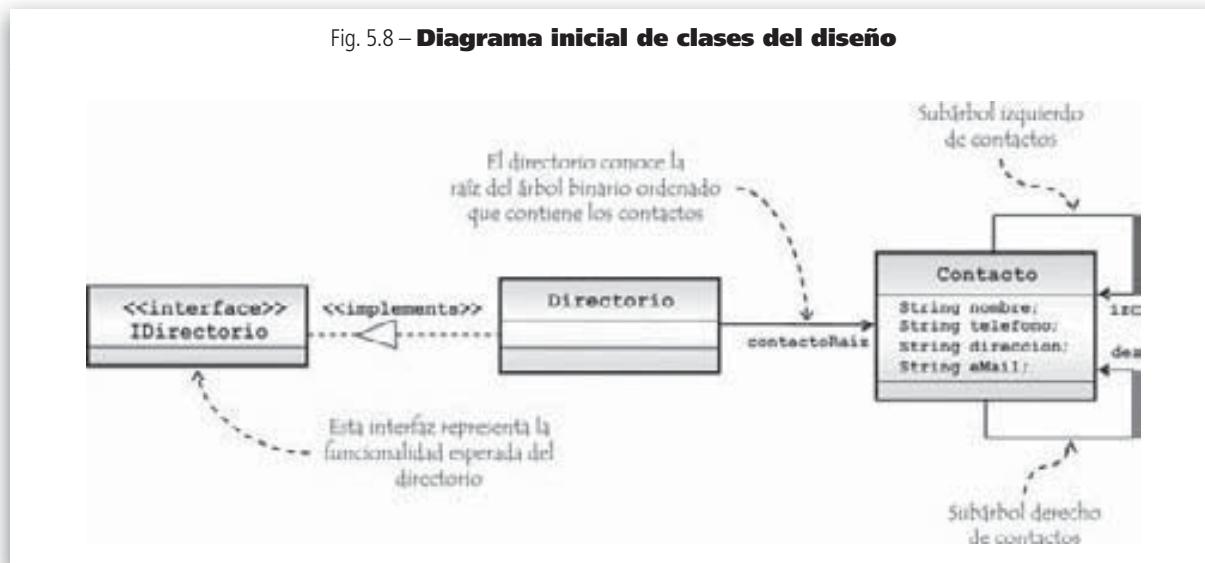
Fig. 5.7 – **Diagrama de clases del análisis**



Lo primero que vamos a hacer, para desacoplar las clases de la interfaz de usuario de las clases del modelo del mundo, es crear una interfaz `IDirectorio`, que represente el contrato funcional entre las dos partes del programa. Esto nos va a facilitar después

la evolución. También transformamos el diagrama de clases para que el grupo de contactos se maneje en un árbol binario ordenado, tal como se muestra en la figura 5.8. Esta decisión la tomamos para satisfacer el requerimiento no funcional de eficiencia.

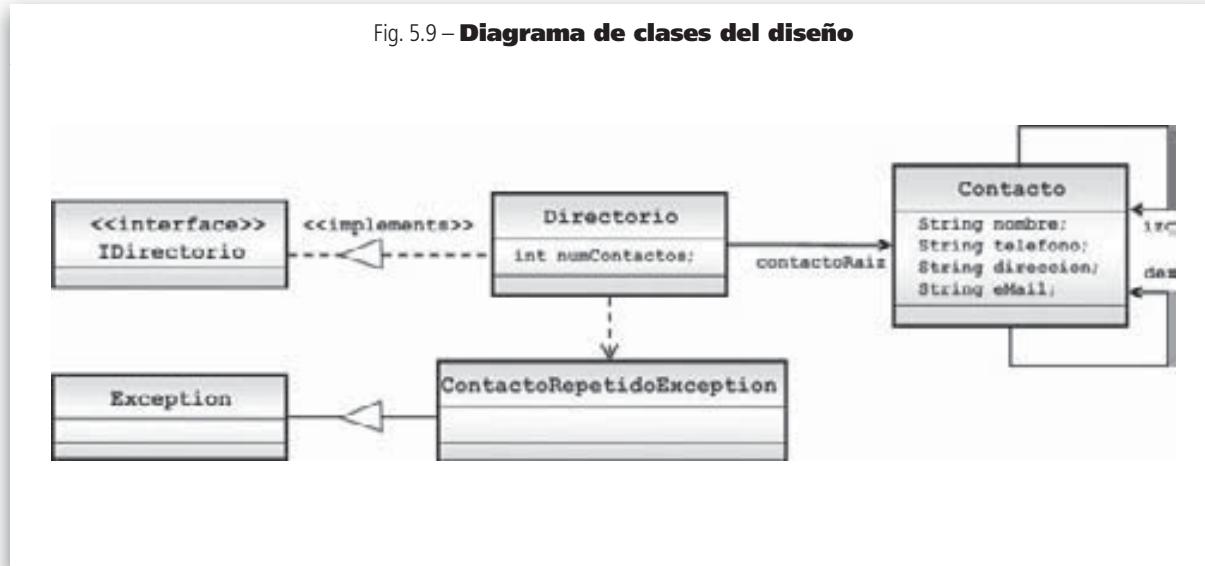
Fig. 5.8 – **Diagrama inicial de clases del diseño**



Finalmente agregamos, también por eficiencia, un atributo en la clase `Directorio` que indica el número de contactos presentes (eso nos evita tener que recorrer el árbol cada vez que alguien pida esta información) y definimos un

nuevo tipo de excepción, que va a ser lanzada si alguien intenta agregar un contacto con un nombre que ya existe en el directorio. En la figura 5.9 aparece el nuevo diagrama de clases del diseño.

Fig. 5.9 – **Diagrama de clases del diseño**



El invariante de la clase Directorio debe incluir las siguientes afirmaciones:

- No hay dos contactos con el mismo nombre en el directorio.
- contactoRaiz es la raíz de un árbol binario, ordenado por el nombre del contacto.
- numContactos es igual al peso del árbol binario de contactos.

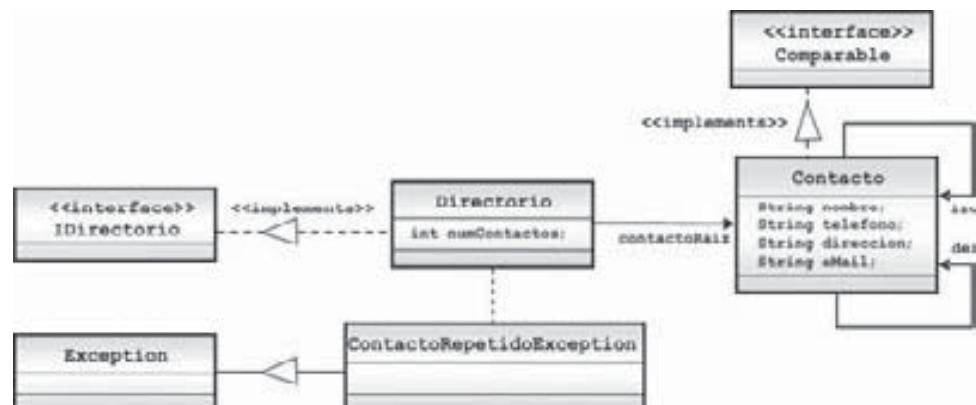
En Java existe una interfaz llamada Comparable, cuyo contrato funcional incluye el siguiente método:

- int compareTo (Object o): Compara el objeto sobre el cual se invoca el método con el

objeto que recibe como parámetro. Retorna un valor negativo si el objeto es menor que el parámetro, un valor positivo si el objeto es mayor que el parámetro y cero si el objeto y el parámetro son iguales.

Para poder independizar la implementación del árbol del concepto de orden que tengan entre sí los contactos, vamos a pedir que la clase Contacto implemente dicha interfaz y que las comparaciones en los métodos del árbol siempre se hagan utilizando el método que ésta incluye. Esto se hace de nuevo para facilitar la evolución del programa. Con esta última modificación llegamos al diagrama final de clases del diseño, el cual se muestra en la figura 5.10.

Fig. 5.10 – **Diagrama final de clases del diseño**



En el ejemplo 3 mostramos las declaraciones de estas clases y estudiaremos las responsabilidades que se deben incluir en la interfaz IDirectorio.

Ejemplo 3



Objetivo: Mostrar las declaraciones de las clases e interfaces que implementan el directorio de contactos.

En este ejemplo mostramos las declaraciones de las clases Directorio, Contacto y ContactoRepetidoException. También definimos los compromisos que debe asumir la interfaz IDirectorio.

```

public class Contacto implements Comparable
{
    // -----
  
```

La clase Contacto se compromete en su encabezado a implementar la interfaz Comparable.

<pre>// Atributos // ----- private String nombre; private String telefono; private String direccion; private String eMail; private Contacto izq; private Contacto der; }</pre>	<p>Declaramos un atributo por cada una de sus características: nombre, teléfono, dirección postal y dirección electrónica. Todos estos atributos se declaran de tipo cadena de caracteres.</p> <p>Finalmente definimos dos asociaciones hacia objetos de la misma clase Contacto. La asociación "izq" llevará al subárbol izquierdo y la asociación "der" nos permitirá llegar al subárbol derecho.</p>
<pre>public class ContactoRepetidoException extends Exception {}</pre>	<p>Esta clase hereda de la clase Exception, y no le agrega ningún nuevo atributo.</p>
<pre>public class Directorio implements IDirectorio { // ----- // Atributos // ----- private Contacto contactoRaiz; private int numContactos; }</pre>	<p>La clase Directorio se compromete con el contrato funcional establecido por la interfaz IDirectorio.</p> <p>El primer atributo de esta clase es la referencia a la raíz del árbol binario ordenado en donde están almacenados los contactos.</p> <p>El segundo atributo es el número de contactos presentes en el árbol.</p>
<pre>public interface IDirectorio { public void agregarContacto(String nombre, String telefono, String direccion, String email) throws ContactoRepetidoException; public Contacto buscarContacto(String nombre); public void eliminarContacto(String nombre); public Collection darListaContactos(); }</pre>	<p>El directorio debe asumir cuatro grandes responsabilidades que se deben expresar en esta interfaz. Sólo dejamos por fuera la parte de las estadísticas que se mencionan en los requerimientos funcionales, las cuales serán tratadas más adelante.</p> <p>Un directorio debe permitir que se le agreguen nuevos contactos, para los cuales recibe la información por parámetro; debe permitir localizar un contacto dado su nombre, debe tener un método para eliminar un contacto y, finalmente, debe tener la posibilidad de retornar una colección con los nombres de los contactos, ordenada alfabéticamente, para que puedan ser desplegados en la interfaz de usuario.</p> <p>El método agregarContacto() lanza la excepción ContactoRepetidoException si ya existe en el directorio una persona con el mismo nombre.</p> <p>El método buscarContacto() retorna null si no hay ningún contacto con ese nombre en el directorio.</p>

Hay un requerimiento funcional sobre el cual nos debemos detener un instante, puesto que habla de desplegar algunas estadísticas referentes al estado de las estructuras de datos con las que se implementa el directorio, pero no precisa cuáles. En nuestro caso, vamos a mostrar la siguiente información:

- Altura del árbol
- Peso del árbol (número de contactos)
- Número de hojas del árbol
- Menor elemento en el árbol
- Mayor elemento en el árbol

El problema es que si agregamos en la interfaz `IDirectorio` los métodos que nos dan esta

información, estaremos comprometiendo la evolución del programa. ¿Qué pasa si queremos implementar el directorio con un vector o con una lista enlazada? ¿Cómo podríamos, en ese caso, implementar un método que retorne la altura?

Una solución es crear una nueva clase llamada `DatoEstadistico`, con dos atributos: uno con el nombre de la característica que se calculó y otro con su valor. Esto nos va a permitir agregar un solo método en el contrato de `IDirectorio` que retorne un arreglo de estos objetos. De esa forma, la interfaz de usuario se puede encargar de presentar en la ventana cualquier información que allí retornemos, sin necesidad de restringirnos a un contenido específico. Esta solución se ilustra en el ejemplo 4.



Ejemplo 4

Objetivo: Mostrar la manera de retornar un conjunto de datos sin comprometerlos con un contenido preciso.

En este ejemplo mostramos la manera de retornar las estadísticas que se piden en el quinto requerimiento funcional del caso de estudio, sin necesidad de generar una dependencia con respecto a la estructura de datos que vamos a utilizar.

```
public class DatoEstadistico
{
    private String nombreDato;
    private Object valorDato;

    public DatoEstadistico( String nom, Object val )
    {
        nombreDato = nom;
        valorDato = val;
    }

    public String darNombre( )
    {
        return nombreDato;
    }

    public String darValor( )
    {
        return valorDato.toString( );
    }
}
```

■ En la declaración de la clase definimos dos atributos: uno con el nombre del dato y otro con su valor. Por generalidad definimos este segundo atributo de la clase `Object`.

■ El método `darValor()` invoca el método `toString()` sobre el objeto que representa el valor. Por la propiedad de asociación dinámica del lenguaje, cada valor contestará con su propia implementación.

■ Agregamos entonces un nuevo método a la interfaz `IDirectorio`, que retorna un arreglo de objetos con datos estadísticos. Este método puede retornar cualquier número de estos valores, y la interfaz de usuario debe ser capaz de desplegarlos de manera adecuada.

■ En la parte de abajo mostramos la implementación que tendría dicho método en la clase `Directorio`. Puesto que debemos retornar cinco valores estadísticos distintos, declaramos un arreglo de dicha dimensión.

```
public interface IDirectorio
{
    ...
    public DatoEstadistico[] darEstadisticas( );
}
```

Luego, para cada valor, creamos un objeto de la clase DatoEstadistico, el cual va a tener el nombre del valor que retorna ("Altura", por ejemplo) y el valor que tiene asociado en la estructura. Puesto que espera un objeto y no un tipo simple, creamos en algunos casos objetos de la clase Integer.

- Los métodos darAltura(), darMenor(), darMayor() y contarHojas() son métodos privados de la clase que serán implementados en la siguiente sección.
- Si queremos incluir cualquier otra información estadística bastaría con modificar este método, evitando tener que cambiar el contrato funcional y la interfaz de usuario.

```
public class Directorio implements IDirectorio
{
    ...

    public DatoEstadistico[] darEstadisticas( )
    {
        Contacto menor = darMenor( );
        Contacto mayor = darMayor( );

        DatoEstadistico[] estadisticas = new DatoEstadistico[5];
        estadisticas[0] = new DatoEstadistico( "Altura", new Integer( darAltura( ) ) );
        estadisticas[1] = new DatoEstadistico( "Peso", new Integer( numContactos ) );
        estadisticas[2] = new DatoEstadistico( "Menor", menor==null ? "" : menor.darNombre() );
        estadisticas[3] = new DatoEstadistico( "Mayor", mayor==null ? "" : mayor.darNombre() );
        estadisticas[4] = new DatoEstadistico( "Hojas", new Integer( contarHojas( ) ) );

        return estadisticas;
    }
}
```



En este punto hemos terminado ya el diseño del programa, satisfaciendo todos los requerimientos funcionales y no funcionales pedidos por el cliente. Pasamos ahora a implementar los métodos necesarios para que el programa funcione correctamente, y para esto debemos estudiar el tema de algoritmos recursivos.

3.5. Algoritmos Recursivos

En esta sección vamos a recorrer los distintos métodos que se deben implementar para el caso de estudio. Comenzaremos con los métodos para los cuales la solución iterativa y la solución recursiva tienen el mismo nivel de dificultad de desarrollo. Luego pa-

saremos a los problemas en los cuales la solución recursiva resulta mucho más sencilla. Finalmente, mostraremos algunos patrones de algoritmo que pueden guiar el proceso de construcción de soluciones recursivas para árboles binarios y algunas guías metodológicas que pueden ser muy útiles en algunos casos.

3.5.1. El Algoritmo de Búsqueda

Comencemos por el método de la clase `Directorio`, que va a permitir localizar un contacto en el árbol (dado

su nombre). Lo primero que debemos señalar es que todos los métodos de la clase `Directorio` se deben limitar a verificar si el árbol de contactos es vacío y, si no es así, delegar en la raíz del árbol la solución del problema.

```
public class Directorio implements IDirectorio
{
    private Contacto contactoRaiz;
    private int numContactos;

    public Contacto buscarContacto( String nombre )
    {
        return contactoRaiz == null ? null :
               contactoRaiz.buscar( nombre );
    }

}

public class Contacto implements Comparable
{
    private String nombre;
    ...
    private Contacto izq;
    private Contacto der;

    public Contacto buscar( String unNombre )
    {
        Contacto p = this;
        while( p != null )
        {
            int comp = p.nombre.compareToIgnoreCase( unNombre );

            if( comp == 0 )
                return p;

            else if( comp > 0 )
                p = p.izq;

            else
                p = p.der;
        }

        return null;
    }
}
```

 Si el árbol es vacío (`contactoRaiz` es `null`), el método retorna `null`, puesto que el contacto no existe.

 En caso contrario, delega en la clase `Contacto` la responsabilidad de localizarlo.

 El método utiliza una variable auxiliar “`p`” para desplazarse sobre el árbol, igual que lo haría sobre una lista enlazada. Esta variable se inicializa en el contacto raíz, usando la variable “`this`”.

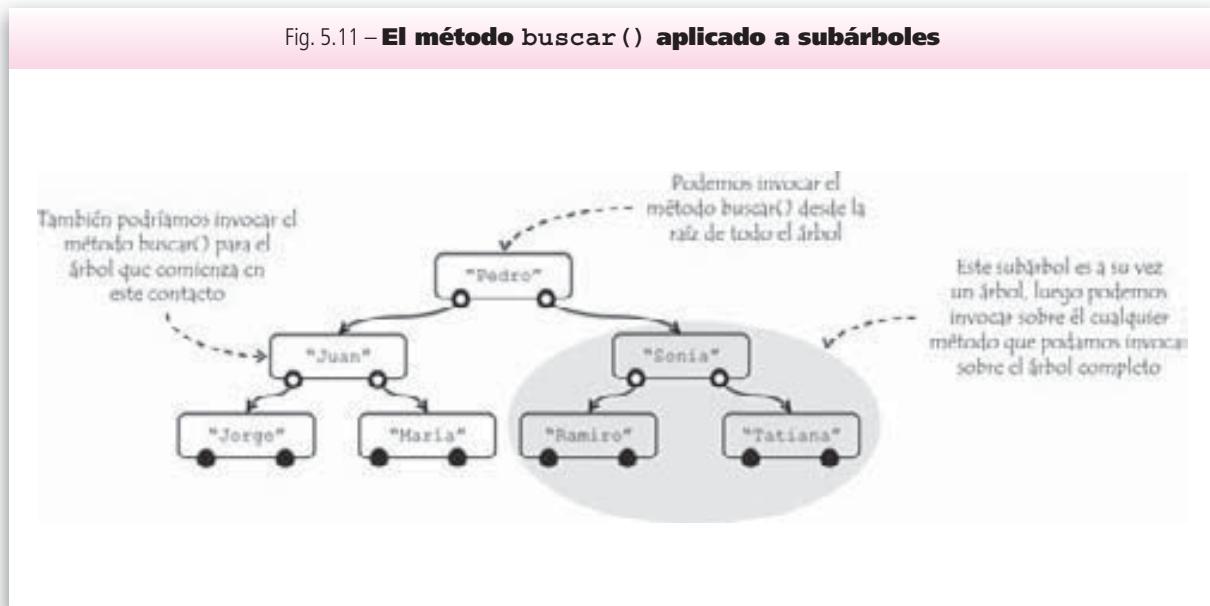
 La única diferencia con respecto a una búsqueda en una lista enlazada es que si el valor que está buscando no es el que se encuentra en la raíz, debe preguntar si es menor o mayor para mover la variable “`p`” hacia el subárbol derecho o el subárbol izquierdo (como si hubiera dos sucesores posibles).

 La comparación la hacemos utilizando el método `compareTo()` de la interfaz `Comparable`.

Consideremos ahora otro tipo de solución. Partamos del hecho de que el método `buscar()` es capaz de localizar un contacto en cualquier árbol binario ordenado. Esto quiere decir que este método es aplicable tanto

para el árbol completo, que contiene todo el directorio, como para cualquiera de los subárboles allí incluidos, tal como se sugiere en la figura 5.11, ya que un subárbol es a su vez un árbol binario ordenado.

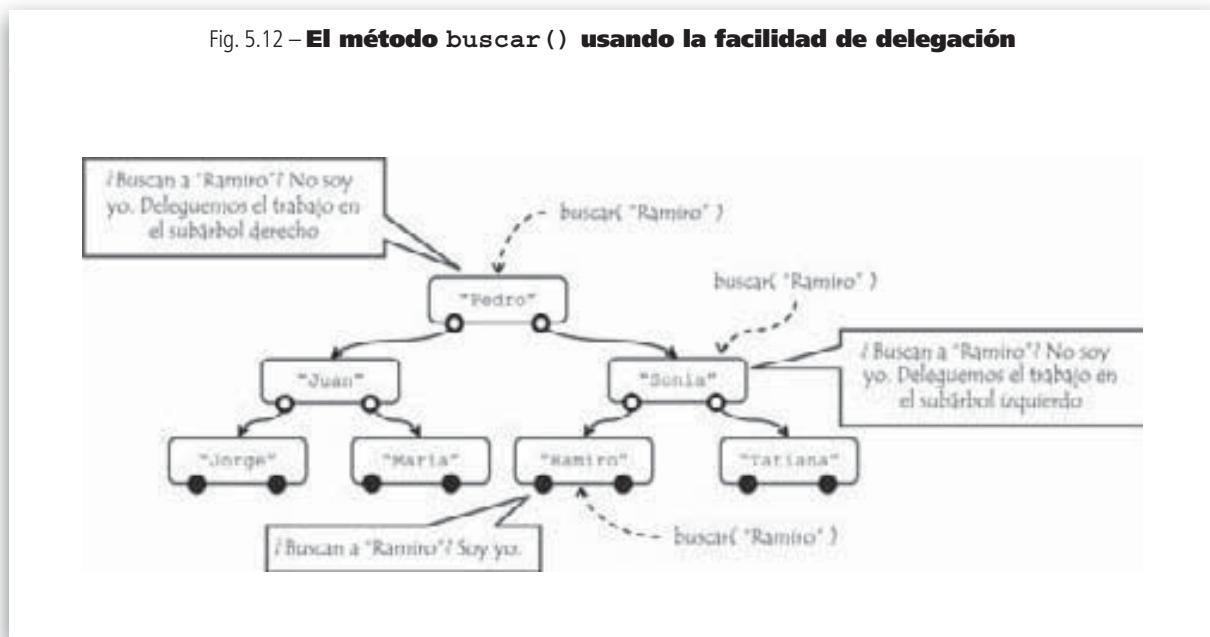
Fig. 5.11 – **El método buscar() aplicado a subárboles**



Si esto es cierto, ¿por qué no aprovechar esta facilidad y dejar que el árbol completo delegue en el subárbol respectivo la responsabilidad de localizar un contacto?

Esta idea se ilustra en la figura 5.12, para el caso en el que estemos tratando de localizar el contacto de nombre "Ramiro".

Fig. 5.12 – **El método buscar() usando la facilidad de delegación**



Si quisieramos implementar el método de búsqueda utilizando esta idea, obtendríamos el siguiente código:

```
public Contacto buscar( String unNombre )
{
    if( nombre.compareToIgnoreCase( unNombre ) == 0 )

        return this;

    else if( nombre.compareToIgnoreCase( unNombre ) > 0 )

        return ( izq == null ) ? null : izq.buscar( unNombre );

    else

        return ( der == null ) ? null : der.buscar( unNombre );
}
```

- Si el elemento que está en la raíz del árbol es el contacto que estamos buscando, se retorna él mismo.
- Si el nombre buscado es menor que el de la raíz, invitamos el método buscar() sobre el subárbol izquierdo.
- Si el nombre buscado es mayor que el de la raíz, invitamos el método buscar() sobre el subárbol derecho.
- Antes de hacer la llamada sobre el subárbol derecho o el subárbol izquierdo, verificamos que dicho árbol no esté vacío (o sea que la referencia al subárbol no sea nula).



Un método que se utiliza a sí mismo como parte de la solución de un problema se denomina un **método recursivo**. Estos métodos son muy naturales de escribir para manipular estructuras recursivas, como es el caso de los árboles.

En el caso del algoritmo de búsqueda, las dos soluciones son igualmente simples de desarrollar. Podemos utilizar cualquiera de las dos en nuestro directorio de contactos.

3.5.2. El Algoritmo de Inserción

Pasemos ahora al algoritmo de inserción, el cual nos permite agregar un contacto. El método de la clase Directorio que inicia el proceso es el siguiente:

```
public class Directorio implements IDirectorio
{
    public void agregarContacto( String nom, String tel,
                                 String dir, String email )
                                throws ContactoRepetidoException
    {
        Contacto c = new Contacto( nom, tel, dir, email );

        if( contactoRaiz == null )
            contactoRaiz = c;
        else
            contactoRaiz.insertar( c );

        numContactos++;
        verificarInvariante();
    }
}
```

- Este método recibe como parámetro los cuatro datos que describen un contacto y crea el objeto que lo va a representar.
- Puede lanzar la excepción que indica que existe un contacto con el mismo nombre.
- Si el árbol está vacío, dejamos el nuevo contacto en la raíz.
- Si no está vacío, le pedimos a la raíz actual que incluya en el árbol el contacto que le pasamos como parámetro.
- Al final aumentamos en 1 el número de contactos y verificamos que se cumpla el invariante.

Tarea 6

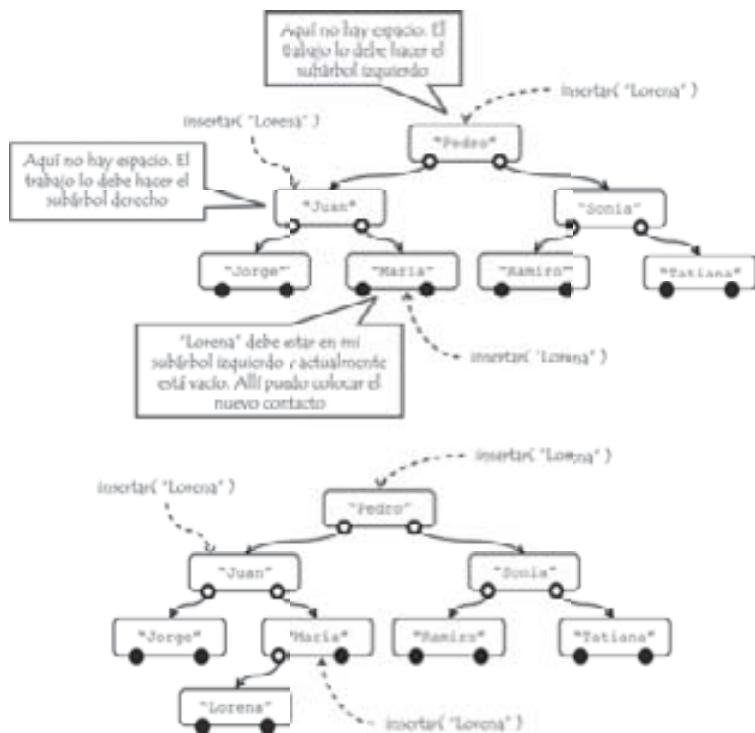
Objetivo: Hacer la implementación iterativa del método de la clase Contacto que agrega un nuevo elemento al árbol.

Implemente el método cuya firma aparece a continuación:

```
public class Contacto implements Comparable
{
    public void insertar( Contacto nuevo ) throws ContactoRepetidoException
    {
        }
}
```

Si queremos implementar la solución recursiva, podemos usar el planteamiento que se resume en la figura 5.13, para el caso en el que estemos tratando de agregar un contacto de nombre "Lorena".

Fig. 5.13 – **Proceso recursivo para insertar el contacto de nombre "Lorena"**



El código del método de la clase `Contacto` que implementa la solución recursiva es el siguiente:

```
public void insertar( Contacto nuevo )
    throws ContactoRepetidoException
{
    if( compareTo( nuevo ) == 0 )
        throw new ContactoRepetidoException( nuevo.nombre );

    if( compareTo( nuevo ) > 0 )
    {
        if( izq == null )
            izq = nuevo;
        else
            izq.insertar( nuevo );
    }
    else
    {
        if( der == null )
            der = nuevo;
        else
            der.insertar( nuevo );
    }
}
```

- El método recibe como parámetro el objeto de la clase `Contacto` que se quiere agregar.
- Si el nombre del nuevo contacto es igual al que se encuentra en la raíz, lanza una excepción.
- Si el nombre es menor que el de la raíz, consideramos dos casos: (1) si el subárbol izquierdo es vacío, ya encontramos el lugar en el que se debe agregar el nuevo elemento, así que simplemente lo enlazamos, o (2) si el subárbol izquierdo no es vacío, le delegamos a éste la responsabilidad de hacer la inserción.
- Si el nombre es mayor que el de la raíz, utilizamos la misma estrategia anterior, pero aplicada al subárbol derecho.

Fíjese que después de hacer el planteamiento recursivo de la solución, escribir el algoritmo respectivo es algo muy sencillo, puesto que sólo nos debemos concentrar en la manera de resolver el problema para la raíz actual e identificar, para el resto de casos, la manera de delegar la solución a uno (o ambos) de los subárboles.



Antes de hacer una llamada recursiva debemos verificar que el subárbol no sea vacío, puesto que si la referencia es nula, vamos a obtener la excepción:

`java.lang.NullPointerException`

Si estudiamos las soluciones recursivas que hemos planteado para los dos problemas anteriores, encontramos dos puntos en común. Éstos son:

- Hay uno o más casos en los cuales podemos dar la respuesta directamente, sin necesidad de delegar el problema, y considerando únicamente la información de la raíz del árbol. Estos casos se denominan

las **salidas de la recursividad** (o casos triviales), y siempre debe existir por lo menos una de ellas en una solución válida. Para la búsqueda de un contacto, el caso trivial se da cuando el elemento de la raíz tiene el nombre buscado. Allí podemos dar una respuesta directa. Para la inserción de contactos, hay dos casos triviales: (1) cuando el contacto que queremos agregar es menor que la raíz y el subárbol izquierdo es vacío y (2) cuando el contacto que queremos agregar es mayor que la raíz y el subárbol derecho es vacío. En ambas situaciones podemos enlazar el nuevo elemento sin necesidad de delegar el problema a uno de los subárboles.

- Hay uno o más casos en los cuales delegamos el trabajo a uno o ambos subárboles. Estos casos se denominan los **avances de la recursividad**, y en todo algoritmo recursivo debe existir por lo menos uno. Para la búsqueda del contacto, delegamos sobre el respectivo subárbol la responsabilidad de localizarlo. Para la inserción de un nuevo elemento, pasamos al subárbol adecuado la responsabilidad de agregarlo.

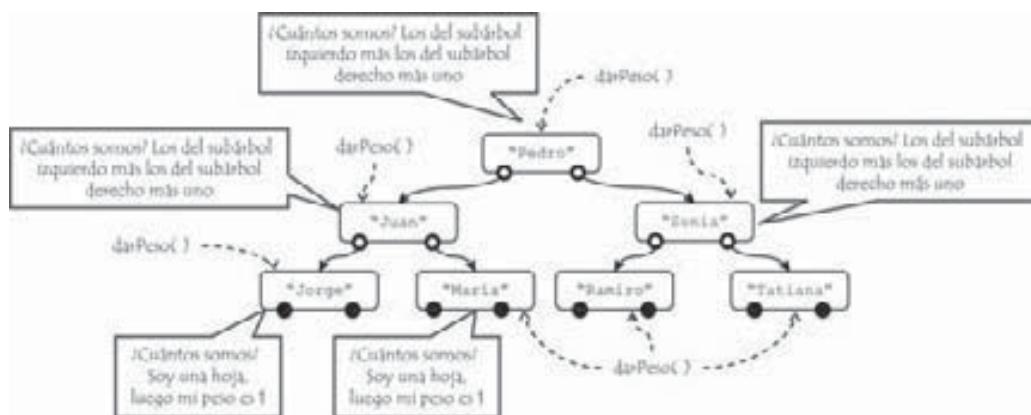


Los algoritmos recursivos funcionan porque en algún momento llegan a un caso trivial. Cuando diseñemos el planteamiento recursivo debemos asegurarnos de que esto suceda.

3.5.3. Algoritmos para Calcular Propiedades de un Árbol

Analicemos ahora la manera de construir otros algoritmos para el caso de estudio, de manera que podamos generalizar la forma de plantear una solución recursiva. Comencemos por el algoritmo que calcula el peso de un árbol. En la figura 5.14 ilustramos el planteamiento recursivo que vamos a utilizar.

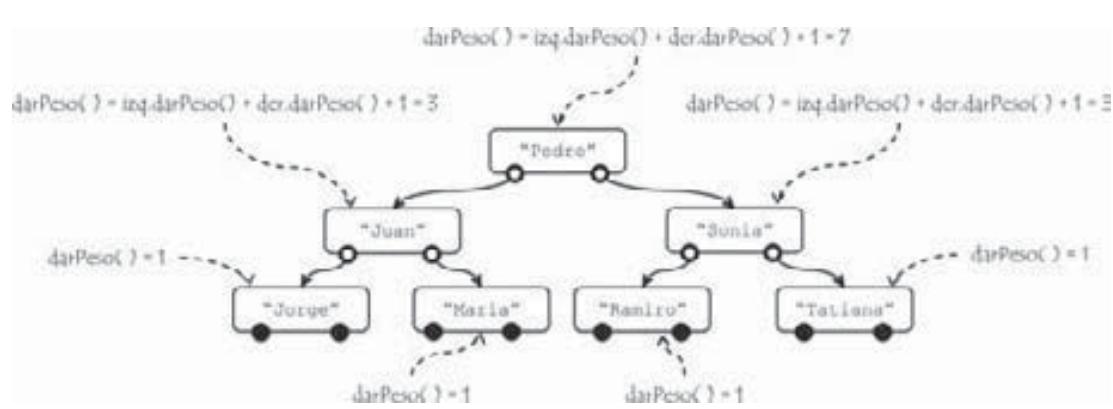
Fig. 5.14 – Planteamiento recursivo para calcular el peso de un árbol binario ordenado



A diferencia de los problemas anteriores, en este caso debemos hacer avanzar la recursividad sobre los dos subárboles a la vez y, luego, utilizar las respuestas que

nos den las dos llamadas para calcular así lo que nos están pidiendo. Eso se ilustra en la figura 5.15.

Fig. 5.15 – Planteamiento recursivo para calcular el peso de un árbol binario ordenado



El método que calcula el peso de un árbol es el siguiente:

```
public class Contacto implements Comparable
{
    public int darPeso( )
    {
        int p1 = ( izq == null ) ? 0 : izq.darPeso( );
        int p2 = ( der == null ) ? 0 : der.darPeso( );
        return 1 + p1 + p2;
    }
}
```

- El método refleja claramente el planteamiento recursivo que hicimos con anterioridad.
- La solución iterativa de este problema puede tener cerca de 20 líneas de código y necesita utilizar una estructura auxiliar de datos.
- En el CD puede consultar la solución iterativa de este método para comparar la complejidad de construir las dos versiones.

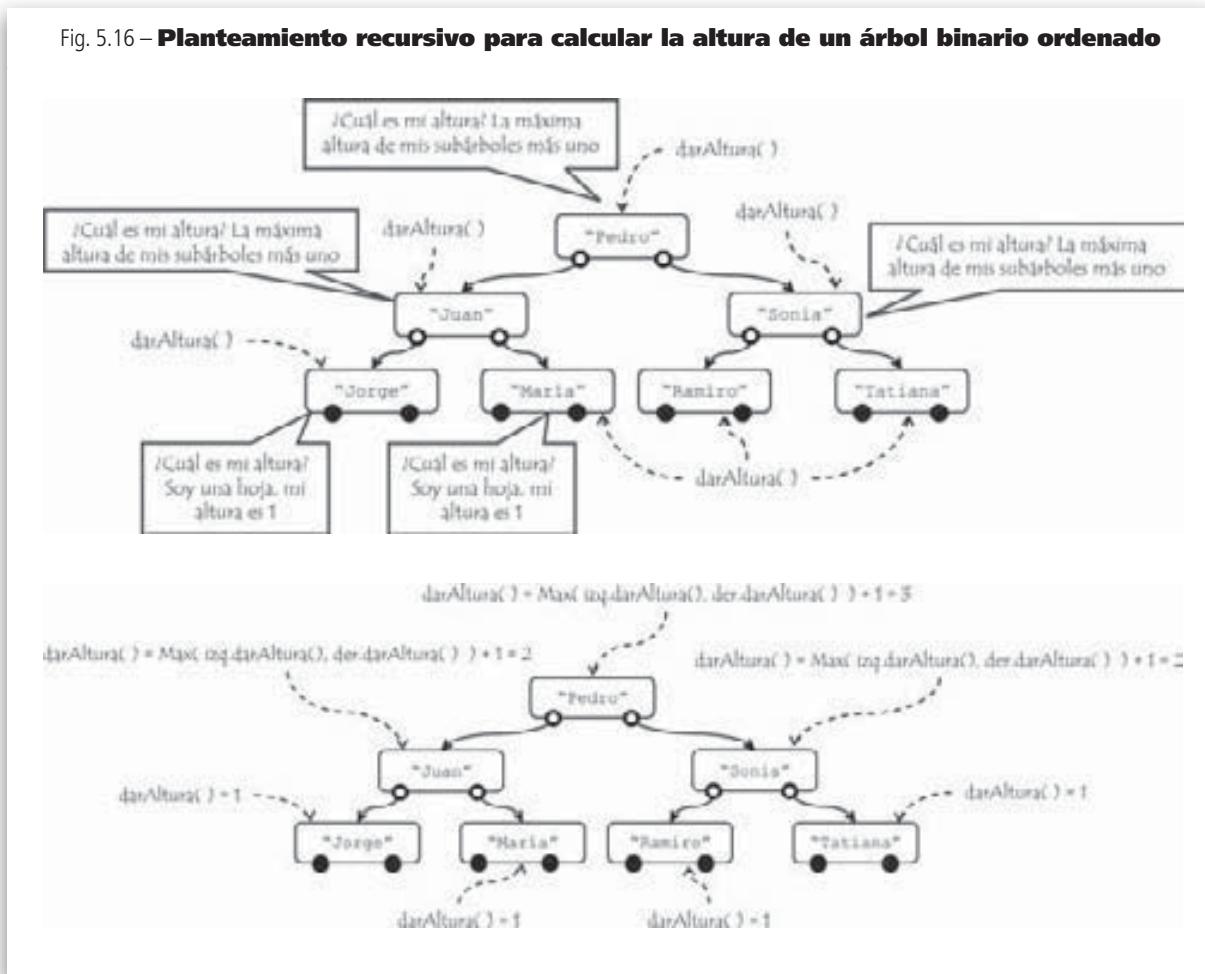
Ahora ya estamos listos para generalizar los puntos que debe incluir el planteamiento de una solución recursiva. Veámoslos en la siguiente tabla, aplicados al método que calcula el peso de un árbol:

Salidas de la recursividad o casos triviales	El árbol es una hoja. La respuesta al problema que se plantea es 1.
Avances de la recursividad	Plantear una parte del problema al subárbol izquierdo (calcular su propio peso). Plantear una parte del problema al subárbol derecho (calcular su propio peso).
Composición de las respuestas de los avances de la recursividad	Se deben sumar las respuestas de los dos avances de la recursividad y luego agregar el valor 1 (para incluir en la suma el elemento de la raíz).

Pasemos ahora al método que calcula la altura de un árbol binario ordenado. El planteamiento recursivo se resume en la siguiente tabla y se ilustra en la figura 5.16.

Salidas de la recursividad o casos triviales	El árbol es una hoja. La altura del árbol es 1.
Avances de la recursividad	Calcular la altura del subárbol izquierdo. Calcular la altura del subárbol derecho.
Composición de las respuestas de los avances de la recursividad	Escoger la mayor altura de los dos subárboles y sumarle 1 (el árbol completo tiene un nivel más que el que tienen los subárboles).

Fig. 5.16 – Planteamiento recursivo para calcular la altura de un árbol binario ordenado



El método que calcula la altura de un árbol es el siguiente:

```
public int darAltura( )
{
    if( esHoja( ) )
        return 1;
    else
    {
        int a1 = ( izq == null ) ? 0 : izq.darAltura( );
        int a2 = ( der == null ) ? 0 : der.darAltura( );
        return 1 + Math.max( a1, a2 );
    }
}
```

```
public boolean esHoja( )
{
    return izq == null && der == null;
}
```

■ Si el árbol es una hoja, su altura es 1.

■ Calculamos luego la altura de los dos subárboles, teniendo en cuenta que la altura de un árbol vacío es 0.

■ Calculamos el máximo entre los dos valores anteriores, utilizando para esto el método Math.max() disponible en Java y luego le sumamos el valor 1.

■ Este método se usa frecuentemente para establecer si un árbol es una hoja.

Veamos ahora los métodos que calculan el menor contacto y el mayor contacto del árbol, teniendo en cuenta el orden alfabético de los nombres. Para cada uno de estos métodos mostramos el planteamiento recursivo y el código en Java que lo implementa.

Salidas de la recursividad o casos triviales

- El árbol no tiene subárbol izquierdo, entonces el contacto de la raíz es el menor.

Avances de la recursividad

- Pedir al subárbol izquierdo el menor contacto.

```
public Contacto darMenor( )
{
    return ( izq == null ) ? this : izq.darMenor( );
}
```

- El menor contacto siempre lo vamos a localizar en el subárbol izquierdo. Terminamos cuando el árbol no tenga subárbol izquierdo.

Salidas de la recursividad o casos triviales

- El árbol no tiene subárbol derecho, entonces el contacto de la raíz es el mayor.

Avances de la recursividad

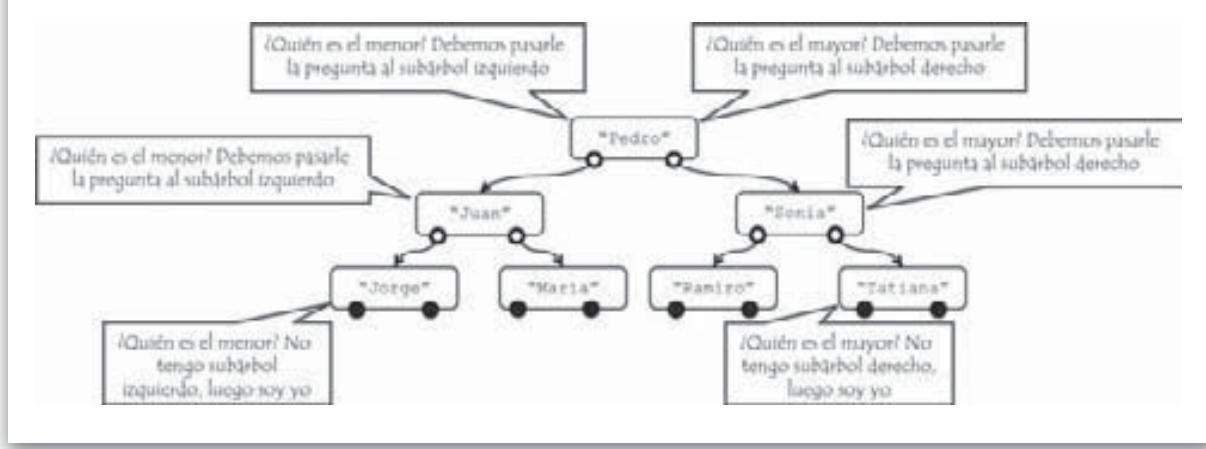
- Pedir al subárbol derecho el mayor contacto.

```
public Contacto darMayor( )
{
    return ( der == null ) ? this : der.darMayor( );
}
```

- El mayor contacto siempre lo vamos a localizar en el subárbol derecho. Terminamos cuando el árbol no tenga subárbol derecho.

La figura 5.17 ilustra el funcionamiento de los métodos anteriores.

Fig. 5.17 – **Planteamiento recursivo para los métodos que calculan el menor y el mayor elemento**



En la tarea 7 planteamos al lector la implementación de una serie de métodos recursivos de la clase `Contacto`.

Tarea 7

Objetivo: Generar habilidad en la construcción de métodos recursivos sobre árboles binarios ordenados.

Implemente los métodos de la clase `Contacto` que se describen a continuación. Haga el planteamiento recursivo antes de comenzar a escribir el código.

```
public int contarHojas( )  
{
```

■ Retorna el número de hojas que hay en el árbol.

```
}
```

■ No olvide verificar que los subárboles no son vacíos antes de hacer las llamadas recursivas.

```
public int contarOcurrencias( String nombreP )  
{
```

■ Calcula el número de contactos del árbol que tienen el nombre que llega como parámetro.

```
}
```

■ Este método lo vamos a utilizar más adelante para verificar que se cumpla el invariante del directorio.

```
public int contarMenores( String nombreP )  
{
```

■ Cuenta el número de contactos del árbol cuyo nombre es menor alfabéticamente a la cadena que se recibe como parámetro.

```
}
```

```
public boolean esOrdenado( )  
{
```

■ Indica si el árbol binario está bien ordenado. Este método lo vamos a utilizar más adelante para verificar el invariante del directorio.

```
}
```

■ Fíjese que es necesario utilizar los métodos `darMenor()` y `darMayor()` desarrollados anteriormente, puesto que no es suficiente con comparar la raíz del árbol con las raíces de sus subárboles.

```

public int contarElementosNivel( int nivel )
{
}

public boolean esCompleto( )
{
}

public String darNombreMasLargo( )
{
}

```

Retorna el número de elementos que tiene el árbol en el nivel que entra como parámetro. Recuerde que la raíz del árbol se encuentra en el nivel 1.

Un árbol binario es completo si todos sus elementos o son hojas o tienen dos subárboles no vacíos asociados.

Este método indica si el árbol de contactos es completo.

Retorna el nombre más largo que aparece en el árbol de contactos.

3.5.4. El Algoritmo de Supresión

Para el método que elimina un elemento del árbol de contactos, vamos a hacer el planteamiento recursivo antes de presentar el código. En la precon-

dición del método vamos a afirmar que existe en el árbol un contacto con el nombre que se quiere eliminar, aprovechando que en el programa que estamos desarrollando dicho contacto se selecciona de una lista.

Signatura del método	public Contacto eliminar(String unNombre)
Precondición	<ul style="list-style-type: none"> ■ El contacto que se quiere eliminar está presente en el árbol.
Salidas de la recursividad o casos triviales	<ul style="list-style-type: none"> ■ El árbol es una hoja. Por la precondición, tiene que ser el elemento que estamos buscando. El método debe retornar entonces el árbol vacío (null). ■ El elemento que queremos eliminar está en la raíz y el subárbol izquierdo es vacío. El método debe retornar el subárbol derecho. ■ El elemento que queremos eliminar está en la raíz y el subárbol derecho es vacío. El método debe retornar el subárbol izquierdo. ■ El elemento que queremos eliminar está en la raíz y los subárboles no son vacíos. El método debe localizar el menor contacto del subárbol derecho, eliminarlo de allí y luego ponerlo como la raíz del árbol. El método debe retornar la nueva raíz del árbol.
Avances de la recursividad	<ul style="list-style-type: none"> ■ El elemento que queremos eliminar es menor que la raíz. Pedimos al subárbol izquierdo que lo elimine, y al árbol que retorne la llamada recursiva lo dejamos como subárbol izquierdo. ■ El elemento que queremos eliminar es mayor que la raíz. Pedimos al subárbol derecho que lo elimine, y al árbol que retorne la llamada recursiva lo dejamos como subárbol derecho.

```

public class Contacto implements Comparable
{
    public Contacto eliminar( String unNombre )
    {
        if( esHoja( ) )
            return null;

        if( nombre.compareToIgnoreCase( unNombre ) == 0 )
        {
            if( izq == null )
                return der;

            if( der == null )
                return izq;

            Contacto sucesor = der.darMenor( );
            der = der.eliminar( sucesor.darNombre( ) );
            sucesor.izq = izq;
            sucesor.der = der;
            return sucesor;
        }

        else if( nombre.compareToIgnoreCase( unNombre ) > 0 )
            izq = izq.eliminar( unNombre );

        else
            der = der.eliminar( unNombre );

        return this;
    }
}

```

- Salida 1: es una hoja.
- Salida 2: no tiene subárbol izquierdo.
- Salida 3: no tiene subárbol derecho.
- Salida 4: tiene ambos subárboles.
- Avances de la recursividad

3.5.5. El Algoritmo de Recorrido en Inorden

En este punto, únicamente nos falta por desarrollar el método que retorna la colección de nombres de con-

tactos, ordenada alfabéticamente. Este método, cuya signatura aparece a continuación, lo va a utilizar la interfaz de usuario para desplegar la lista de personas que tenemos en el directorio.

```
public Collection darListaContactos();
```



Este método retorna una colección de cadenas de caracteres, con los nombres de todos los contactos en el directorio, ordenada alfabéticamente. Si el directorio está vacío, este método retorna null.

Para implementar este método vamos a utilizar una técnica un poco distinta, llamada **acumulación de parámetros**, que resulta más eficiente para este caso. Esta

técnica tiene muchas variantes posibles, y se basa en la idea de ir construyendo la respuesta en los parámetros, a medida que avanza el proceso recursivo.

```
public class Directorio implements IDirectorio
{
    public Collection darListaContactos()
    {
        if( contactoRaiz == null )
            return null;
        else
        {
            Collection resp = new ArrayList();
            contactoRaiz.inorden( resp );
            return resp;
        }
    }
}
```

■ Ésta es la implementación del método en la clase Directorio.

■ Este método considera dos casos. En el primero, si el directorio está vacío retorna directamente null.

■ En el segundo, si el directorio no está vacío, crea un objeto de la clase ArrayList (que implementa la interfaz Collection) y se lo pasa como parámetro en la invocación al método inorden() sobre la raíz del árbol. Éste lo debe modificar, dejando allí la secuencia de nombres. Finalmente el método retorna el vector modificado.

Ésta es la primera vez que vamos a modificar un objeto que recibimos como parámetro, de manera que el método que hizo la invocación reciba información en el

valor de los parámetros que pasó y no en el retorno del método. En el ejemplo 5 estudiamos esta idea y mostramos algunas restricciones que existen.



Esta manera de enviar información hacia el método que hace la invocación sólo funciona con parámetros que son objetos y no con tipos simples.

**Ejemplo 5**

Objetivo: Mostrar la manera de retornar información modificando el estado de los parámetros.

En este ejemplo mostramos la manera de enviar información hacia el método que hace la invocación de un método utilizando los parámetros de la llamada. Esta técnica se usa en muy pocos casos y casi siempre refleja un error de diseño del programa.

```
public class C2
{
    private int atr1;

    public void sumar()
    {
        atr1++;
    }
}
```

```
public class C1
{
    public void acumular( C2 obj )
    {
        obj.sumar();
    }

    public void destruir( C2 obj )
    {
        obj = null;
    }

    public C2 m1( )
    {
        C2 v2 = new C2();

        acumular( v2 );
        acumular( v2 );
        acumular( v2 );

        destruir( v2 );

        return v2;
    }
}
```

■ Esta clase tiene un único atributo de tipo entero, que va a servir para acumular información. Suponemos que el constructor de la clase inicializa dicho atributo en 0.

■ Con el método sumar() se incrementa el valor del atributo.

■ Esta clase tiene tres métodos: el primero modifica el parámetro que recibe, invocando el método sumar(). Esto hace que cambie el estado del objeto que se pase como parámetro.

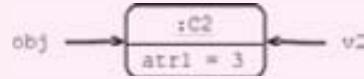
■ El segundo método asigna null a la referencia que recibe como parámetro.

■ El tercer método utiliza la técnica de acumulación de parámetros. Crea una instancia de la clase C2 y se la pasa como parámetro tres veces al método acumular(). Luego le pasa esa misma instancia al método destruir() y finalmente la retorna. ¿Qué valor tiene el retorno del método?

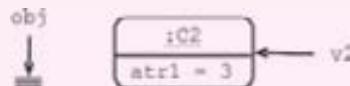
■ La respuesta es que retorna un objeto de la clase C2, cuyo atributo "atr1" tiene el valor 3. ¿Por qué? ¿Por qué no retorna null? La respuesta es que un método puede cambiar el estado del objeto que recibe como parámetro, pero no puede modificar la referencia que hay hacia él. Antes de la llamada del método destruir(), tenemos el siguiente diagrama de objetos:



■ En el momento de la llamada, obtenemos el siguiente diagrama, en el cual hay una nueva referencia al mismo objeto indicado por "v2", que corresponde al parámetro "obj":



■ Si invocamos cualquier método usando la referencia "obj", el estado del objeto cambia, pero si asignamos null a "obj", obtenemos el siguiente diagrama de objetos:



■ Esto hace que en el momento de retornar el objeto referenciado por "v2" sigamos teniendo el mismo objeto que pasamos como parámetro.

El método recursivo que hace el recorrido en inorden en la clase `Contacto` es el siguiente. En la figura 5.18 se ilustra su funcionamiento.

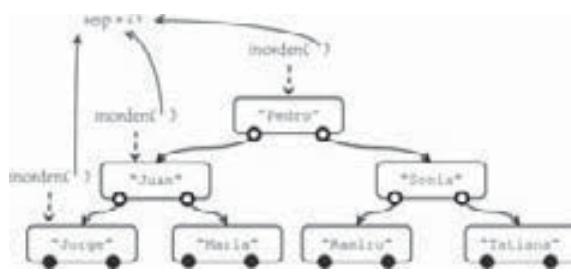
```
public void inorden( Collection acumulado )
{
    if( izq != null )
        izq.inorden( acumulado );

    acumulado.add( nombre );

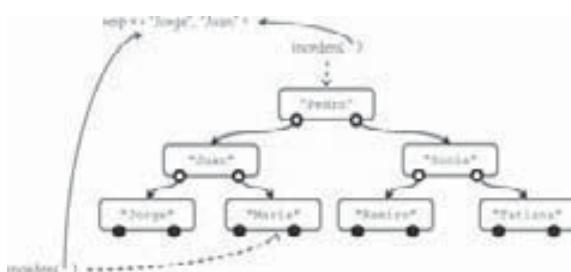
    if( der != null )
        der.inorden( acumulado );
}
```

- El método recibe como parámetro una colección en donde están almacenados los elementos que ya han sido recorridos. Se sabe que todos los elementos que allí aparecen son menores que el contacto que se encuentra en este punto del árbol.
- Si el subárbol izquierdo no es vacío, pedimos que se agreguen a la colección sus elementos en inorden.
- Luego agregamos el nombre del contacto que se encuentra en este punto del árbol.
- Finalmente, si el subárbol derecho no es vacío, pedimos que se agreguen a la colección sus elementos en inorden.

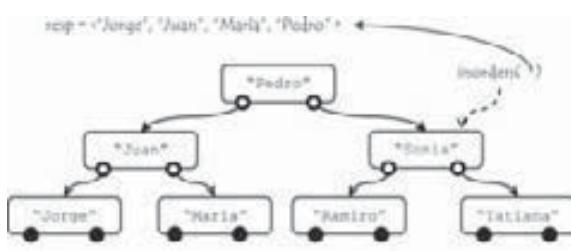
Fig. 5.18 – **Funcionamiento del recorrido en inorden de un árbol**



- Al hacer la llamada del método sobre la raíz del árbol, se le pasa una referencia al vector en donde se debe acumular el recorrido.
- Puesto que el subárbol izquierdo no es vacío, le pide que se recorra y deje la respuesta en la referencia que le pasa como parámetro.
- Éste a su vez repite lo mismo con el árbol que comienza con el contacto llamado "Jorge". Sus subárboles izquierdo y derecho son vacíos, de manera que se incluye en el recorrido y retorna.



- Al retornar, el contacto "Juan" se incluye en el recorrido y hace la invocación sobre su subárbol derecho.



- Los contactos "María" y "Pedro" se incluyen después en el recorrido, y se hace la llamada recursiva sobre el subárbol derecho ("Sonia"), pasándole como parámetro la referencia a la colección de elementos ya recorridos.
- El proceso continúa de la misma manera sobre el resto del árbol, hasta que termina el recorrido.



Las implementaciones completas (iterativas y recursivas) de los principales métodos de la clase `Contacto` pueden ser consultadas en el CD. Allí también se pueden ver las pruebas automáticas y la forma como la interfaz de usuario presenta la información estadística sin saber a priori su contenido ni su cantidad.

3.5.6. Verificación del Invariante

Tarea 8



Objetivo: Escribir el método que verifica el invariante de la clase `Directorio`.

Implemente los métodos de la clase `Directorio` que verifican que el invariante se cumpla. Debe verificar tres condiciones: (1) no hay dos contactos con el mismo nombre en el directorio, (2) el árbol de contactos es un árbol binario ordenado y (3) el atributo `numContactos` tiene un valor igual al peso del árbol. Utilice los métodos implementados en la tarea 7.

```
public class Directorio implements IDirectorio
{
```

```
    private void verificarInvariante( )
    {
        }
}
```

3.5.7. Patrones de Algoritmo para Árboles Binarios

Como vimos en las secciones anteriores, hay dos grandes tipos de algoritmos recursivos para manejar árboles binarios. A los primeros los llamaremos de **descenso recursivo**, y tienen el siguiente esqueleto de algoritmo.

```
if( condición1 )
{
    // Solución directa 1
}
else if( condición2 )
{
    // Solución directa 2
}
else
{
    // Avance de la recursividad sobre el subárbol izquierdo
    // Avance de la recursividad sobre el subárbol derecho
    // Composición de las respuestas de cada uno de los avances
}
```

El algoritmo tiene una o varias salidas de la recursividad, en las cuales es posible dar una solución directa al problema.

Se hace avanzar el proceso recursivo sobre uno o ambos subárboles y, luego, se compone las respuestas que se obtuvieron.

Los segundos corresponden a problemas que se resuelven utilizando el recorrido en inorder del árbol. Piense, por ejemplo, en el problema de encontrar el siguiente contacto en orden alfabetico dentro del árbol. Estos algoritmos los llamaremos de **avance ordenado**, y tienen el siguiente esqueleto de algoritmo.

```
if( izq != null )
{
    // Avance ordenado sobre el subárbol izquierdo
}

// Inclusión de la raíz en el avance ordenado

if( der != null )
{
    // Avance ordenado sobre el subárbol derecho
}
```

A diferencia del esqueleto anterior, en este caso los elementos del árbol se van procesando en orden ascendente de valor.

Es muy usual que este patrón de algoritmo vaya acompañado de la técnica de acumulación de parámetros.

Tarea 9



Objetivo: Desarrollar algunos métodos utilizando los patrones de algoritmo vistos anteriormente.

Implemente los métodos de la clase `Contacto` que se plantean a continuación. Identifique como primera medida el patrón de algoritmo que se debe usar y utilice el esqueleto correspondiente.

```
public boolean igualContenido( Contacto arbol2 )
{
}
```

Retorna verdadero si el árbol cuya raíz se recibe como parámetro tiene los mismos elementos del árbol que recibe la llamada, sin importar si están estructurados de manera distinta.

```
public boolean sonIdenticos( Contacto arbol2 )
{
```

■ Retorna verdadero si el árbol cuya raíz se recibe como parámetro es idéntico al árbol que recibe la llamada. Esto quiere decir que tiene la misma estructura y contenido.

```
}
```

```
public boolean estaIncluido( Contacto arbol2 )
{
```

■ Retorna verdadero si el árbol cuya raíz se recibe como parámetro es idéntico a uno de los subárboles del árbol que recibe la llamada.

■ Utilice el método anterior.

```
}
```

```
public int darPosicion( String nombreP,
{
```

) ■ Retorna la posición en orden alfabético del contacto que tiene el nombre que se recibe como parámetro. Si es el menor del directorio, por ejemplo, debe retornar el valor 1.

■ No utilice ninguna estructura auxiliar de datos para almacenar el recorrido en inorden.

■ Agregue los parámetros que crea convenientes para utilizar la técnica de acumulación de parámetros.

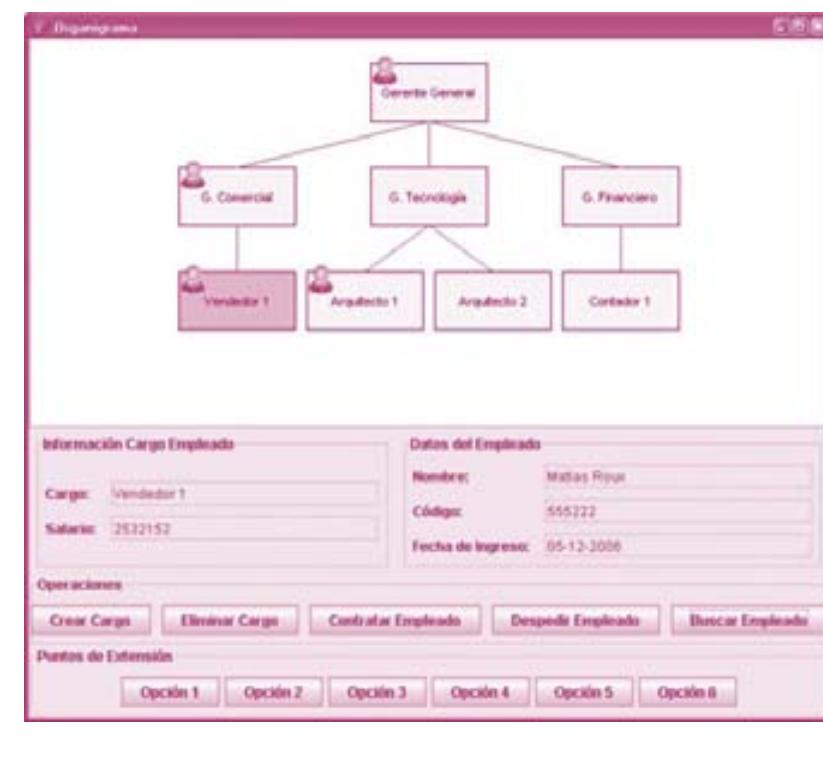
```
}
```

4. Caso de Estudio Nº 2: Organigrama de una Empresa

Una empresa necesita construir un programa para manejar su organigrama, en el cual sean claros los cargos que existen en la empresa y las personas que los ocupan. De cada cargo debemos saber (1) su nombre (que debe ser único), (2) el salario asignado al cargo, (3) de qué otro cargo depende (quién es su jefe) y (4) los cargos que dependen de él (quiénes son sus subalternos directos). Sólo hay un cargo en la empresa que no depende de nadie y éste es el punto de inicio del organigrama. Cada persona que trabaja en la empresa ocupa un cargo y tiene la siguiente información asociada: un código, un nombre y una fecha de ingreso. Un cargo que no tiene un empleado asignado se dice que está vacante.

Las opciones que debe ofrecer el programa son las siguientes: (1) Crear un nuevo cargo en la empresa. Para esto se debe dar el nombre del nuevo cargo, el salario asociado y el nombre del cargo del cual depende. (2) Eliminar un cargo de la empresa. En este caso el usuario debe dar el nombre del cargo que se quiere suprimir, el cual debe estar vacante y no tener cargos que lo tengan como jefe. (3) Contratar a una persona, dando un código que debe ser único, un nombre, una fecha de ingreso y el nombre del cargo vacante que entra a ocupar. (4) Despedir a un empleado, dando su código. En ese caso el cargo que ocupaba esta persona queda vacante. (5) Mostrar el organigrama de la empresa, en donde se puedan apreciar las relaciones entre los cargos. (6) Desplegar la información de un empleado dado su código. El programa debe tener la interfaz de usuario que se presenta en la figura 5.19.

Fig. 5.19 – Interfaz de usuario del caso de estudio



- En la parte superior de la ventana se puede apreciar el organigrama actual de la empresa. El ícono indica si el cargo se encuentra ocupado o vacante.
- Para crear o suprimir un cargo se utilizan los botones de la parte izquierda de la ventana. Si el cargo no se puede suprimir, se presenta un mensaje de error al usuario.
- Para contratar o despedir empleados, se debe suministrar la información que solicita el programa.
- Cuando se consulta la información de un empleado, ésta aparece en la parte inferior derecha de la ventana y el cargo que ocupa en el organigrama se despliega con otro color.

4.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
 Representar el organigrama de la empresa como un árbol n-ario	 Aprender a declarar y manipular una estructura jerárquica en la cual cada elemento tiene un número indefinido de subárboles asociados.
 Escribir los algoritmos que permiten actualizar el organigrama.	 Generalizar el concepto de algoritmo recursivo, para manejar árboles n-arios.

4.2. Comprensión de los Requerimientos

Tarea 10		
Objetivo: Entender el problema del caso de estudio del organigrama de una empresa.		
(1) Lea detenidamente el enunciado del caso de estudio para el manejo del organigrama de una empresa e (2) identifique y complete la documentación de los requerimientos funcionales.		
Requerimiento funcional 1	Nombre	R1 – Crear un nuevo cargo en la empresa
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Eliminar un cargo de la empresa
	Resumen	
	Entrada	
	Resultado	

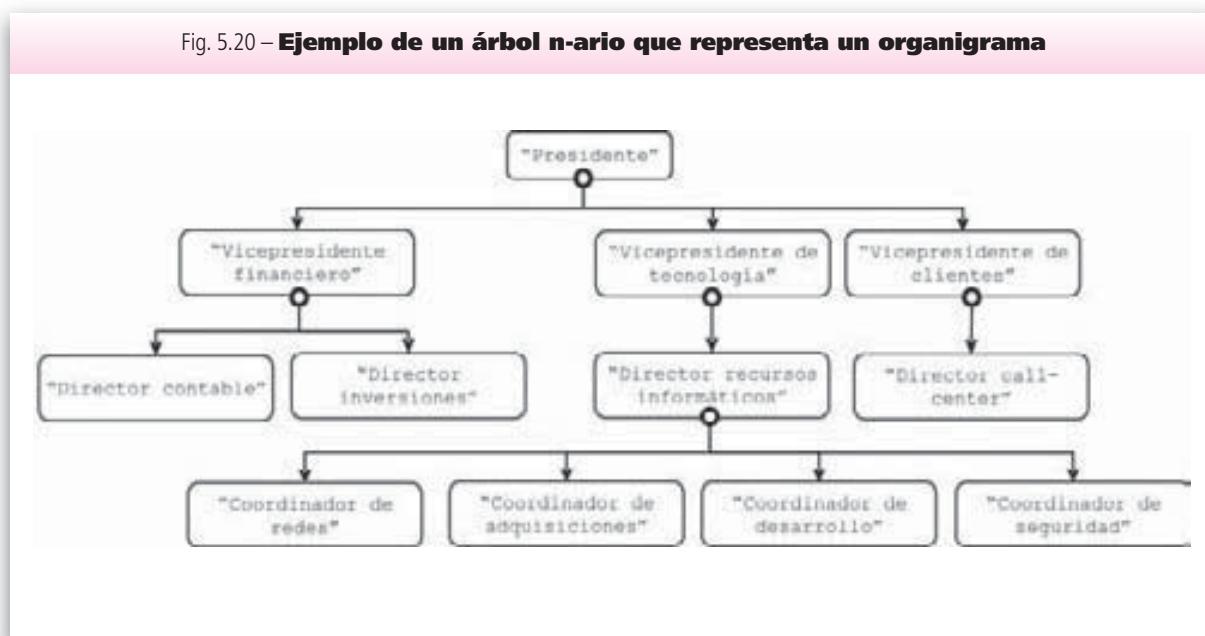
Requerimiento funcional 3	Nombre	R3 – Contratar a una persona
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Despedir a un empleado
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Mostrar el organigrama de la empresa
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 6	Nombre	R6 – Desplegar la información de un empleado
	Resumen	
	Entrada	
	Resultado	

4.3. Árboles n-arios

Existen dos diferencias fundamentales entre los árboles binarios ordenados que utilizamos en el caso de estudio anterior y los árboles n-arios que vamos a utilizar para representar el organigrama de una empresa. En el primer caso, cada elemento del árbol podía tener como máximo dos subárboles asociados. Ahora vamos a tener cualquier número de subárboles asociados, tenien-

do en cuenta que un cargo puede tener bajo su mando cualquier número de cargos subalternos. La segunda diferencia es que dentro del árbol que vamos a manejar no existe una noción de orden, ya que no lo estamos utilizando para manipular información ordenada de manera eficiente, sino que representa la estructura que tienen los elementos del mundo. Un ejemplo de un árbol n-ario que representa el organigrama de una empresa se muestra en la figura 5.20.

Fig. 5.20 – **Ejemplo de un árbol n-ario que representa un organigrama**



Un árbol n-ario es una estructura recursiva en la que cada elemento puede tener cualquier número de subárboles n-arios asociados. En este caso el orden de los subárboles no es importante,

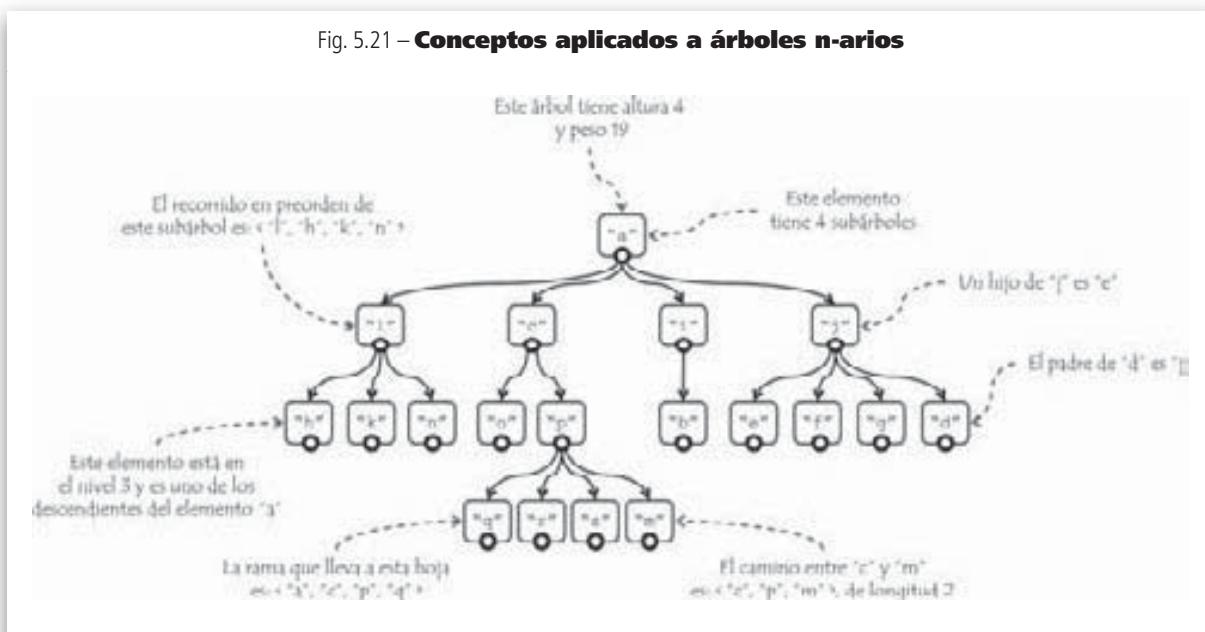
en el sentido de que no es necesario saber cuál es el primero o el último, sino simplemente saber que es un subárbol.

Las definiciones que presentamos anteriormente para árboles binarios se extienden sin problema para los árboles n-arios. Aquí tendremos también hojas, altura, peso, hijos, padre, raíz, subárbol y nivel. En la

figura 5.21 se muestra la manera como se adaptan estos conceptos a esta nueva estructura. Adicionales a estos conceptos, vamos a introducir otros que van a completar la terminología típica de árboles y los cuales también se ilustran en la figura 5.21:

- Para recorrer un árbol n-ario existe un algoritmo llamado **preorden**, en el cual se incluye primero la raíz del árbol y luego se recorren en preorden todos los subárboles que tiene asociados. El orden en el que se recorren los elementos del árbol no cumple ninguna propiedad particular, pero se garantiza que en el recorrido va a aparecer una vez cada uno de los elementos del árbol.

- Un **camino** entre dos elementos E1 y E2 de un árbol n-ario es una secuencia de elementos presentes en el árbol $\langle N_1, \dots, N_k \rangle$ que cumple las siguientes propiedades: (1) $N_1 = E_1$, (2) $N_k = E_2$, (3) N_i es el padre de N_{i+1} . Dicha secuencia no necesariamente existe entre todo par de elementos del árbol. En caso de que exista un camino en el árbol entre E1 y E2, se dice que E1 es un **ancestro** de E2 y que E2 es un **descendiente** de E1. La **longitud** del camino se define como el número de elementos de la secuencia menos 1.
- Una **rama** es un camino que lleva de la raíz a una hoja del árbol. Eso quiere decir que en un árbol existe el mismo número de ramas que de hojas. La altura de un árbol es la longitud de la rama más larga del árbol más 1.

Fig. 5.21 – **Conceptos aplicados a árboles n-arios**

A diferencia de los árboles binarios ordenados, para poder agregar o suprimir un elemento se debe indicar el punto del árbol en el cual se desea realizar la operación (como no hay un orden, no sabríamos dónde insertar un nuevo elemento, por ejemplo). En el caso del organigrama, como los nombres de los cargos son únicos, vamos a utilizarlos como punto de referencia.

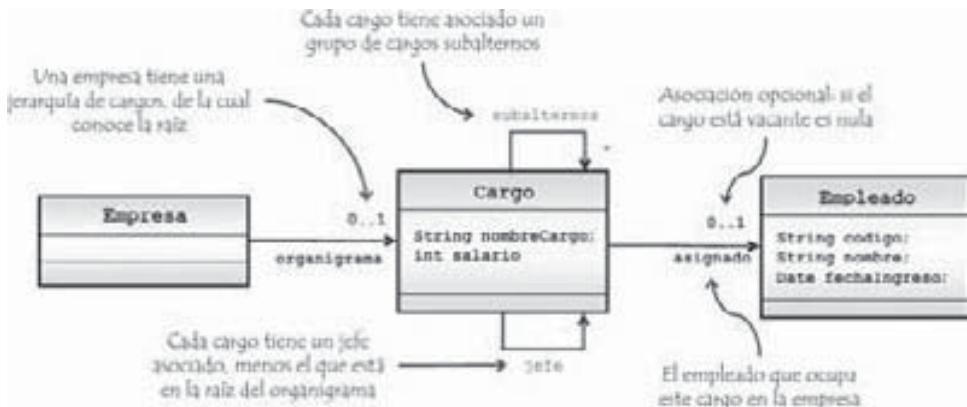


Existen también los árboles n-arios ordenados y balanceados, muy utilizados en el manejo de índices para búsquedas. En este grupo encontramos los árboles B y los árboles 2-3, tema de cursos posteriores.

Nuestras operaciones van a ser del estilo: agregue al organigrama el cargo "Coordinador" dependiendo del cargo "Director", o contrate a "Juan" para el cargo de "Coordinador". Con esa referencia debemos buscar por todo el árbol el elemento hasta localizarlo, y luego sí hacer la operación pedida.

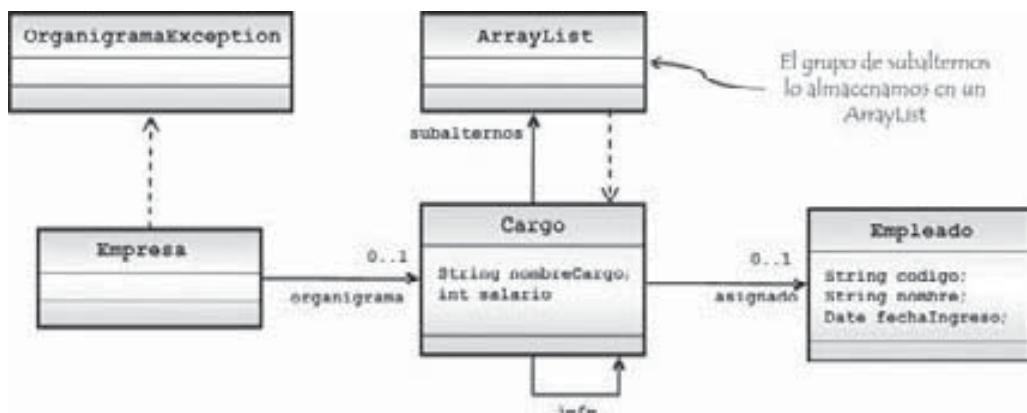
4.4. Del Análisis al Diseño

Después del proceso de análisis, se pueden identificar tres entidades en el modelo del mundo: la empresa, el cargo y el empleado. Estos tres conceptos se encuentran relacionados como aparece en la figura 5.22. Puesto que el mundo está organizado en una jerarquía, desde la misma etapa de análisis detectamos el árbol n-ario de cargos.

Fig. 5.22 – **Diagrama de clases del análisis**

En el diagrama de clases del diseño, únicamente tenemos que concretar el tipo de estructura contenedora en la que vamos a manejar los subalternos (un *ArrayList*) y las excepciones que van a lanzar los métodos de la

empresa (*organigramaException*). Con estas modificaciones, llegamos al diagrama que se muestra en la figura 5.23, sobre el cual vamos a hacer la implementación del caso de estudio.

Fig. 5.23 – **Diagrama de clases del diseño**

En el ejemplo 6 mostramos las declaraciones de estas clases, presentamos sus invariantes y establecemos las principales responsabilidades de la clase Empresa,

de manera que pueda satisfacer los requerimientos funcionales.

**Ejemplo 6**

Objetivo: Presentar las declaraciones de las clases para implementar el manejador de organigramas.

En este ejemplo mostramos las declaraciones de las clases `Empresa`, `Cargo`, `Empleado` y `OrganigramaException` y definimos las principales responsabilidades de la clase `Empresa`, a través de la cual se va a conectar la interfaz de usuario con el modelo del mundo.

```
public class Empleado implements Serializable
{
    // -----
    // Atributos
    // -----
    private String codigo;
    private String nombre;
    private Date fechaIngreso;
}
```

```
public class OrganigramaException extends Exception
{}
```

```
public class Cargo implements Serializable
{
    private String nombreCargo;
    private int salario;
    private Cargo jefe;

    private Empleado asignado;
    private ArrayList subalternos;

    public Cargo( String nCargo, int pago,
                 Cargo superior )
    {
        nombreCargo = nCargo;
        salario = pago;
        jefe = superior;

        asignado = null;
        subalternos = new ArrayList( );

        verificarInvariantes( );
    }
}
```

■ Esta clase implementa la interfaz `Serializable` para poder utilizar dicho esquema de persistencia en el programa.

■ El invariante de la clase incluye tres afirmaciones:

- `codigo != null && codigo != ""`
- `nombre != null && nombre != ""`
- `fechaIngreso != null`

■ La clase `OrganigramaException` no tiene atributos adicionales. En su constructor, invocamos simplemente el constructor de la superclase.

■ La clase `Cargo` define sus atributos de acuerdo con el diagrama de clases presentado anteriormente.

■ El invariante de la clase incluye tres afirmaciones:

- `nombreCargo != null && nombreCargo != ""`
- `salario > 0`
- `subalternos != null`

■ El constructor recibe como parámetro el nombre del cargo que se está creando, el salario asignado a este cargo por la empresa y una referencia al cargo del cual va a depender.

■ En el constructor, inicializamos el vector de subalternos en vacío y dejamos el cargo vacante (`asignado = null`).

■ Al final, verificamos que el objeto creado cumpla con el invariante de la clase.

■ Las responsabilidades de esta clase surcirán como consecuencia de la asignación de responsabilidades de la clase `Empresa`.

```

public class Empresa
{
    private Cargo organigrama;

    public Empresa( String archivo )
    {
        ...
    }

    public void crearCargo( String nCargo, int pago,
                           String nCargoJefe )
                           throws OrganigramaException
    { ... }

    public void eliminarCargo( String nCargo )
                           throws OrganigramaException
    { ... }

    public void contratarPersona( String idPersona,
                                 String nombre, Date ingreso,
                                 String nCargo )
                                 throws OrganigramaException
    { ... }

    public void despedirEmpleado( String idPersona )
                           throws OrganigramaException
    { ... }

    public Empleado buscarEmpleado(String idPersona)
    { ... }
}

```

- El único atributo de la clase Empresa es la raíz del árbol de cargos que representa el organigrama.
- El invariante de la clase incluye dos afirmaciones:
 - los nombres de los cargos son únicos.
 - los códigos de identidad de los empleados son únicos.
- En el constructor cargamos el estado del modelo del mundo del archivo cuyo nombre se recibe como parámetro.
- La clase debe asumir cinco grandes responsabilidades para poder satisfacer los requerimientos funcionales.
- Para crear un nuevo cargo, debe recibir el nombre del cargo, el salario asignado y el nombre del cargo del cual va a depender. Si algo falla en el proceso, lanza una excepción.
- Para eliminar un cargo se debe suministrar su nombre. Si hay algún problema, lanza una excepción.
- Para contratar a una persona se debe dar su identificación, su nombre, su fecha de ingreso y el nombre del cargo para el cual llega.
- Para despedir a un empleado basta con dar su código de identificación.
- Para localizar a un empleado se debe suministrar su código de identificación

4.5. Algorítmica de Árboles n-arios

La algorítmica para manejar árboles n-arios es muy parecida a aquella utilizada en los árboles binarios, con la diferencia de que en los avances de la recursividad debemos considerar todos los subárboles asociados con cada elemento, en lugar de sólo dos. Esto nos va a obligar a construir un ciclo para movernos sobre ellos. Comenzamos por la algorítmica de búsqueda, uno de los puntos centrales en el manejo de estas estructuras.

4.5.1. Los Algoritmos de Búsqueda

El primer método que vamos a desarrollar es el que permite localizar un cargo en el organigrama, dado su nombre. Al igual que con los árboles binarios, vamos a hacer la implementación en dos niveles: el primero, en la clase `Empresa`, en donde se considera el caso en el cual el organigrama está vacío. El segundo nivel (implementado en la clase `Cargo`) se resuelve de manera recursiva, avanzando el proceso por todos los subárboles hasta que alguno encuentre el elemento buscado.

```

public class Empresa
{
    public Cargo buscarCargo( String nCargo )
    {
        return organigrama == null ? null :
            organigrama.buscarCargo( nCargo );
    }
}

public class Cargo
{
    public Cargo buscarCargo( String nCargo )
    {
        if( nombreCargo.equalsIgnoreCase( nCargo ) )
        {
            return this;
        }
        else
        {
            for( int i = 0; i < subalternos.size( ); i++ )
            {
                Cargo hijo = ( Cargo )subalternos.get( i );
                Cargo temp = hijo.buscarCargo( nCargo );
                if( temp != null )
                    return temp;
            }
            return null;
        }
    }
}

```

El método considera dos casos. En el primero, si el organigrama está vacío, el método retorna null, puesto que no existe un cargo con el nombre pedido.

En el segundo caso, delega la responsabilidad de la búsqueda en el cargo que se encuentra en la raíz del árbol n-ario que contiene el organigrama.

El método tiene una salida de la recursividad, cuando el cargo que se encuentra en la raíz tiene el nombre que se está buscando. En ese caso el método retorna el elemento de la raíz (this).

En cualquier otro caso debemos delegar la responsabilidad de la búsqueda en los subárboles. Puesto que no existe una relación de orden entre los elementos, no sabemos en cuál de los subárboles debemos buscar. Por esa razón tenemos que intentar la llamada recursiva sobre todos ellos, terminando el proceso cuando alguno de los subárboles encuentre el cargo que estamos buscando.

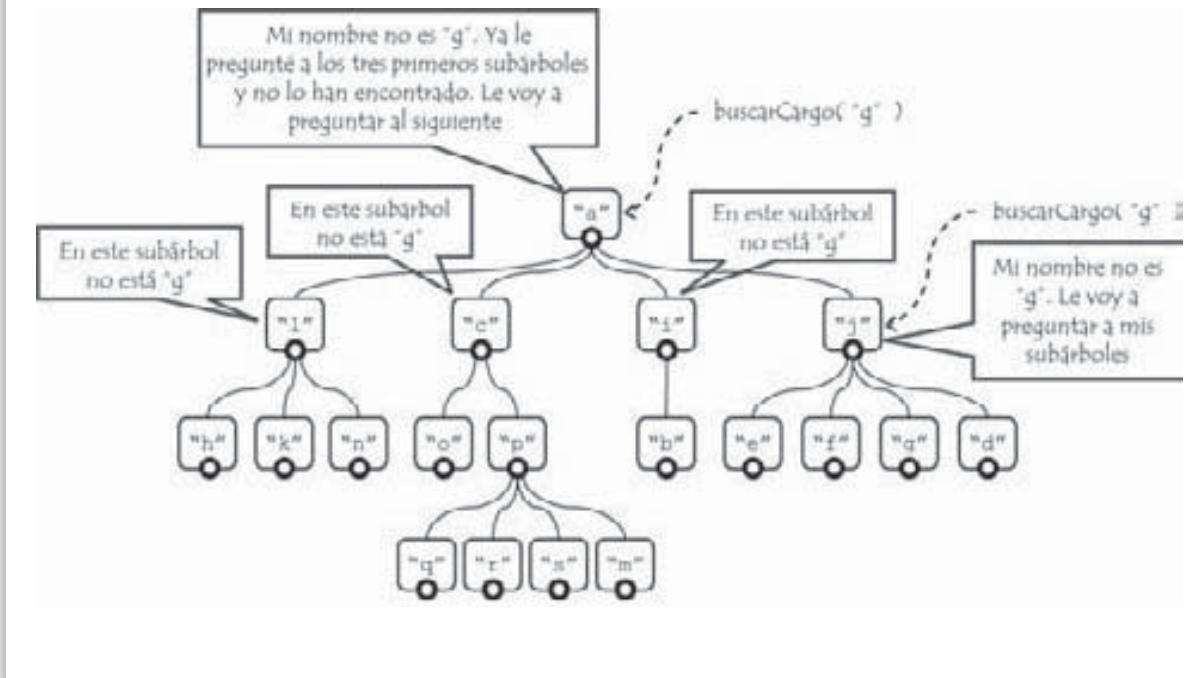
Para esto utilizamos un ciclo sobre todos los elementos del vector de subárboles, moviéndonos sobre éste con la variable "i". Luego, a cada uno de los subárboles le pedimos que busque en su interior el cargo con el nombre que recibimos como parámetro. Si la respuesta es distinta de null, el método termina porque ya encontramos la respuesta.

Al final del ciclo, si ningún subárbol ha encontrado la respuesta, el método retorna el valor null, para indicar que el cargo no aparece en todo el árbol.

El planteamiento recursivo del método anterior se ilustra en la figura 5.24. Allí hacemos explícito en el dibujo (suprimiendo la dirección de la flecha) que, en nuestro diseño, cada elemento del organigrama

conoce también la localización de su padre. Recuerde de que cada objeto de la clase `Cargo` tiene una asociación llamada "jefe" hacia el cargo del cual depende.

Fig. 5.24 – Planteamiento recursivo de la búsqueda de un elemento en un árbol n-ario



Pasamos ahora al método de búsqueda que nos permite localizar el padre de un elemento en un árbol n-ario. Este método va a ser muy útil para crear

o suprimir cargos, puesto que para cualquiera de esos dos procesos debemos situar primero el cargo del cual depende.

```
/*
 * Busca el cargo del que depende el cargo con el nombre dado
 * @param nCargo El nombre del cargo del que se desea el cargo jefe
 * @return El cargo del que depende el cargo cuyo nombre se da como
 * parámetro. Si el cargo no es encontrado se retorna null.
 */

public class Cargo
{
    public Cargo buscarJefe( String nCargo )
    {
        Cargo cargo = buscarCargo( nCargo );
        return ( cargo == null ) ? null : cargo.jefe;
    }
}
```

La localización en el organigrama del cargo del cual depende otro (cuyo nombre se recibe como parámetro) es muy simple. Basta con localizar el cargo en el árbol n-ario utilizando el método que desarrollamos antes, y luego navegar hacia el padre utilizando la asociación llamada "jefe".

El único cargo cuyo jefe va a tener el valor null es el que está en la raíz del árbol completo.

En la tarea 11 se propone al lector el desarrollo de algunos métodos de búsqueda sobre el organigrama, usando distintos criterios.

Tarea 11

Objetivo: Implementar distintos métodos de búsqueda sobre árboles n-arios.

Implemente los métodos de la clase `Cargo` que se plantean a continuación. Es importante que haga explícito el planteamiento recursivo antes de comenzar a escribir el código del método.

```
public class Cargo
{
    public Cargo buscarCargoEmpleado( String idEmpleado )
    {
```

```
}
```

```
public Cargo buscarPeorSalario( )
```

```
}
```



Este método localiza el cargo que desempeña en la empresa un empleado, dado su código de identificación. Si el empleado no trabaja en la empresa, el método retorna el valor null.



Este método localiza en el organigrama de la empresa el cargo con el peor salario.

```
public Cargo buscarMejorVacante( )
```

```
{
```

```
}
```

```
public Empleado buscarMasAntiguo( )
```

```
{
```

```
}
```

```
public Cargo buscarVacanteNivel( int nivel )
```

```
{
```

```
}
```

Este método localiza en el organigrama de la empresa el cargo vacante con el mejor salario.

Este método localiza al empleado más antiguo de la empresa.

Retorna cualquier cargo vacante que se encuentre en el nivel que se recibe como parámetro.

Si no encuentra ninguno vacante, o si el árbol no tiene ese número de niveles, el método retorna null.

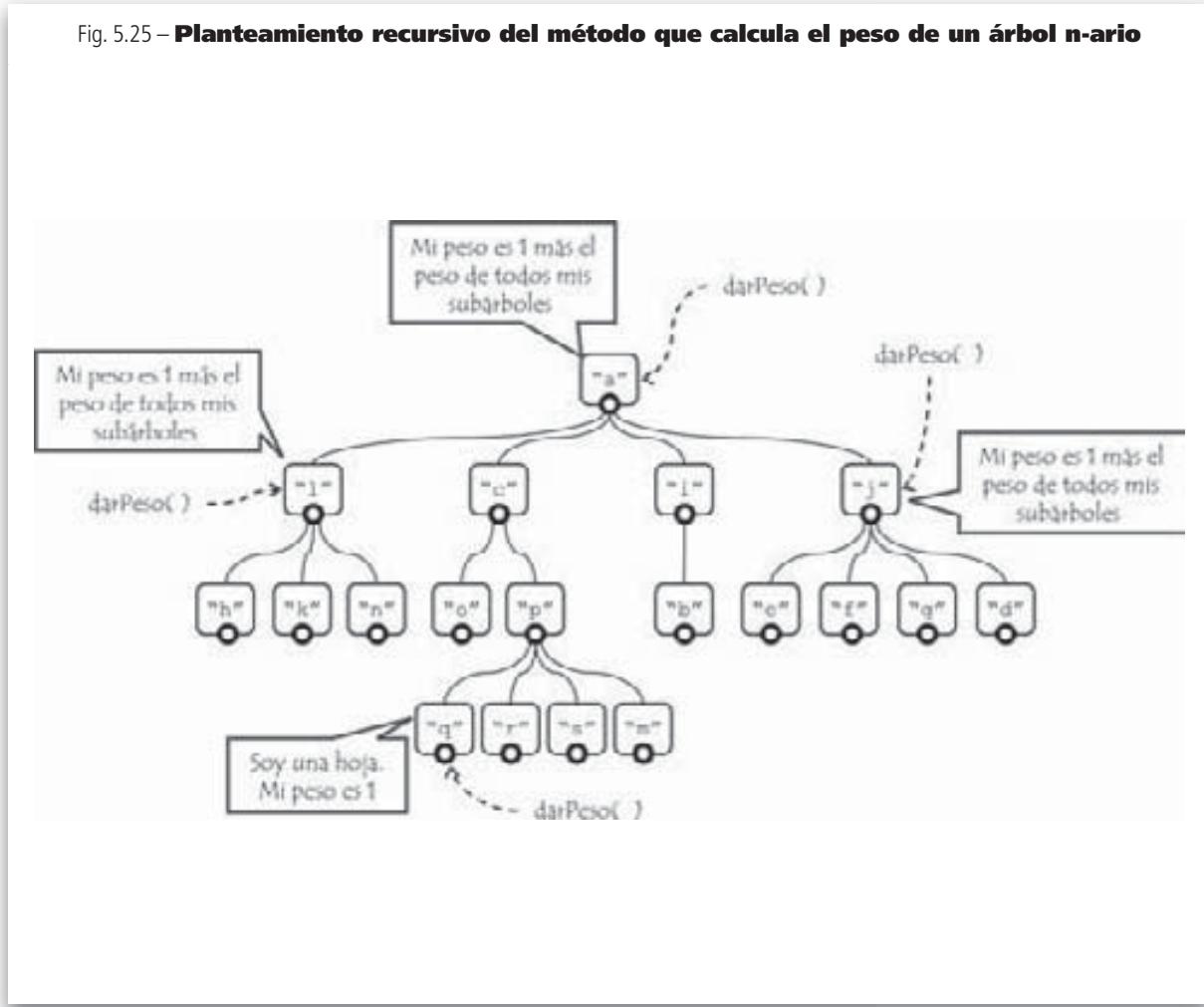
4.5.2. Algoritmos para Calcular Propiedades

Comenzamos esta sección con el algoritmo que calcula el peso de un árbol n-ario. La mayoría de métodos que

calculan propiedades sobre estos árboles van a seguir el mismo esquema de solución, que corresponde a una generalización del patrón de descenso recursivo. En la siguiente tabla se resume el planteamiento recursivo utilizado, y en la figura 5.25 se ilustra su funcionamiento.

Salidas de la recursividad o casos triviales	<ul style="list-style-type: none"> ■ El árbol es una hoja. El peso de una hoja es 1.
Avances de la recursividad	<ul style="list-style-type: none"> ■ Cada subárbol debe calcular y retornar su peso. ■ Se deben sumar todas las respuestas obtenidas, y a ese resultado sumarle 1.

Fig. 5.25 – **Planteamiento recursivo del método que calcula el peso de un árbol n-ario**



El código del método de la clase `Cargo` que calcula el peso del árbol es el siguiente:

```
public class Cargo
{
    public int darPeso( )
    {
        if( esHoja( ) )
            return 1;

        else
        {
            int pesoAcum = 1;

            for( int i = 0; i < subalternos.size( ); i++ )
            {
                Cargo hijo = ( Cargo )subalternos.get(i);

                pesoAcum += hijo. darPeso( );
            }
            return pesoAcum;
        }
    }

    public boolean esHoja( )
    {
        return subalternos.size( ) == 0;
    }
}
```

■ Si el árbol es una hoja (no tiene subárboles asociados), el peso del árbol es 1. Ésta es la salida de la recursividad.

■ En el caso general, utilizamos la variable “`pesoAcum`” para acumular las respuestas que van retornando los subárboles. Dicha variable la inicializamos en 1 para considerar en el peso el elemento actual.

■ El método finalmente retorna el valor que acumulamos en la variable “`pesoAcum`”.

■ Este método indica si un árbol es una hoja, verificando el tamaño del vector de subárboles.

A continuación planteamos al lector el desarrollo de una serie de métodos que calculan propiedades sobre el organigrama del caso de estudio.

Tarea 12



Objetivo: Implementar distintos métodos para calcular propiedades sobre árboles n-arios.

Implemente los métodos de la clase `Cargo` que se plantean a continuación. Es importante que haga explícito el planteamiento recursivo antes de comenzar a escribir el código del método.

```
public class Cargo
{
    public int darAltura( )
    {
    }
```

■ Este método calcula la altura del árbol n-ario que almacena el organigrama de la empresa.

```
public int contarHojas( )  
{
```

```
}
```

```
public int contarVacantes( )  
{
```

```
}
```

```
public int calcularNomina( )  
{
```

```
}
```

Este método cuenta el número de hojas que tiene el árbol, lo cual corresponde al número de cargos de los cuales no depende ningún otro.

Este método cuenta el número de cargos vacantes que hay en la empresa.

Este método calcula el valor total de la nómina de la empresa, sumando el salario de todos los cargos de la empresa que no se encuentran vacantes.

```
public int contarInjusticias( )  
{
```

```
}
```

```
public boolean dependeDe( String nCargo1, String nCargo2 )  
{
```

```
}
```



Este método calcula el número de cargos que tienen un salario superior al de su jefe directo.



Este método indica si un cargo (nCargo1) depende de otro (nCargo2), así sea de manera indirecta. Esto equivale a determinar si nCargo1 es descendiente de nCargo2 dentro del organigrama.

4.5.3. Los Algoritmos de Inserción y Supresión

El algoritmo para crear un nuevo cargo en la empresa se basa en el algoritmo de búsqueda antes planteado. Como de costumbre, divide las responsabilidades entre

las clases Empresa y Cargo, de manera que la primera clase trate el caso en el cual el cargo que llega es el primero en el organigrama y haga las validaciones necesarias, mientras la segunda clase se encarga de crear y agregar el nuevo cargo en un punto del organigrama que ya ha sido localizado.

```
public class Empresa
{
    public void crearCargo( String nCargo, int pago,
                           String nCargoJefe )
                           throws OrganigramaException
    {
        if( organigrama == null )
            organigrama = new Cargo( nCargo, pago, null );

        else
        {
            Cargo padre = buscarCargo( nCargoJefe );

            if( padre == null )
                throw new OrganigramaException("Cargo inválido");

            Cargo nodo = buscarCargo( nCargo );

            if( nodo != null )
                throw new OrganigramaException("Cargo repetido");

            padre.agregarCargo( nCargo, pago );
        }
    }
}

public class Cargo
{
    public void agregarCargo( String nCargo, int pago )
                            throws OrganigramaException
    {
        Cargo subalterno = new Cargo( nCargo, pago, this );
        subalternos.add( subalterno );
    }
}
```

Si la empresa no tiene ningún cargo creado, el primero que llega queda en la raíz.

Si ya existen cargos en el organigrama, lo primero que hacemos es localizar el cargo del cual va a depender el nuevo. Si no lo encuentra lanzamos una excepción.

Luego, verificamos que no exista en la empresa otro cargo con el mismo nombre. Si ya existe otro con ese nombre, lanzamos una excepción.

Finalmente delegamos al cargo (del cual va a depender el nuevo) la responsabilidad de agregarlo como cargo subalterno.

Este método crea el objeto que representa el nuevo cargo, utilizando el nombre y el salario recibidos como parámetro.

Luego agrega dicho objeto en el vector de cargos subalternos.

Para suprimir un cargo de la empresa seguimos un esquema equivalente al utilizado antes: la primera clase (Empresa) hace todas las validaciones y localiza el punto del árbol sobre el cual se debe hacer la modi-

ficación, y luego pasa a la segunda clase (Cargo) la responsabilidad de eliminar de la contenedora el cargo subalterno indicado.

```

public class Empresa
{
    public void eliminarCargo( String nCargo )
        throws OrganigramaException
    {
        if( organigrama == null )
            throw new OrganigramaException( "Cargo inválido" );

        if( organigrama.darNombreCargo( ).equalsIgnoreCase( nCargo ) )
        {
            if( organigrama.esHoja() && organigrama.estáVacante( ) )
                organigrama = null;
            else
                throw new OrganigramaException("Imposible eliminar" );
        }
        else
        {
            Cargo padre = organigrama.buscarJefe( nCargo );
            if( padre == null )
                throw new OrganigramaException( "Cargo inválido" );
            padre.eliminarCargo( nCargo );
        }
    }
}

```

```

public class Cargo
{
    public void eliminarCargo( String nCargo )
        throws OrganigramaException
    {
        for( int i = 0; i < subalternos.size( ); i++ )
        {
            Cargo hijo = ( Cargo )subalternos.get( i );
            if( hijo.darNombreCargo( ).equalsIgnoreCase( nCargo ) )
            {
                if( hijo.esHoja( ) && hijo.estáVacante( ) )
                    subalternos.remove( i );
                return;
            }
            else
                throw new OrganigramaException( "No eliminable" );
        }
    }
}

```

El método recibe como parámetro el nombre del cargo que se quiere eliminar.

La primera verificación que hacemos es que existan cargos en la empresa. Si no es así, lanzamos una excepción.

Debemos considerar aparte el caso en el cual el cargo que se quiere eliminar es el que se encuentra en la raíz del organigrama. Si se cumplen las condiciones exigidas para la supresión de un cargo (el cargo está vacante y no hay cargos que dependan de él), asignamos null al atributo "organigrama".

En el caso general, localizamos al padre del elemento que se quiere eliminar (usando el método buscarJefe) y delegamos en éste la responsabilidad de terminar el proceso. Si no puede localizar al padre, el método lanza una excepción.

El método recibe como parámetro el nombre del cargo que se quiere eliminar.

La precondición establece que dicho cargo existe y que es un subalterno directo del elemento sobre el cual se hace la invocación.

Recorremos el vector de subalternos buscando aquél que tenga el nombre que recibimos como parámetro. Al encontrarlo verificamos que cumpla las condiciones exigidas para que pueda ser suprimido de la empresa (el cargo está vacante y no hay cargos que dependan de él). Si no las cumple lanzamos una excepción. Si cumple las condiciones, lo eliminamos de la contenedora utilizando el método remove().

A continuación planteamos al lector el desarrollo de una serie de métodos que modifican la estructura del organigrama de la empresa.

Tarea 13


Objetivo: Implementar distintos métodos para modificar la estructura de un árbol n-ario.

Implemente los métodos de la clase `Cargo` que se plantean a continuación. Es importante que haga explícito el planteamiento recursivo antes de comenzar a escribir el código del método. Por simplicidad, vamos a suponer que como resultado del proceso no se afecta la raíz del organigrama completo de la empresa.

```
public class Cargo
{
    public Cargo recortePresupuestal( int salarioP )
    {
```

```
}
```

```
public int expansion(
{
```

```
}
```

Este método elimina todos los cargos de la empresa que no tienen subalternos asignados y que ganan un salario superior o igual al valor que llega como parámetro.

Algunos de los cargos que pierden sus subalternos se pueden convertir en hojas, los cuales no hay que considerar dentro del proceso de eliminación.

) Este método modifica el organigrama de la empresa, como respuesta a una política de expansión.

Esta política consiste en agregar un nuevo cargo de apoyo como subalterno de cada uno de los cargos de niveles 2 a 4.

Los cargos deben tener el nombre "asistente" y un valor consecutivo. Se puede suponer que dichos nombres no existen en el organigrama.

El método retorna el número de cargos creados.

Puede agregar los parámetros que estime convenientes en el encabezado del método.

4.5.4. Los Algoritmos de Modificación



Las implementaciones de los métodos que permiten contratar y despedir empleados de la empresa los puede consultar en el CD que acompaña al libro o en el sitio web.

En la siguiente tarea plantearemos algunos métodos que modifican el estado del organigrama sin modificar su estructura.

Tarea 14



Objetivo: Implementar distintos métodos para modificar el estado de un árbol n-ario, sin cambiar su estructura.

Implemente los métodos de la clase `Cargo` que se plantean a continuación. Es importante que haga explícito el planteamiento recursivo antes de comenzar a escribir el código del método.

```
public class Cargo
{
    public void aumentoSalarios1( )
```

Este método permite aumentar los salarios de la empresa utilizando la siguiente política: el aumento para los cargos de nivel 1 es del 1%, para los cargos de nivel 2 es del 2%, y así sucesivamente para el resto de cargos de la empresa.

Agregue en el encabezado del método los parámetros que estime convenientes.

```
}
```

```
public void aumentoSalarios2( )
```

Este método permite aumentar los salarios de la empresa utilizando la siguiente política: cada cargo recibe un 1% de aumento por cada subalterno que tenga.

```
}
```

```
public void ocuparVacantes( )
{
}

}
```

Este método busca los cargos vacantes de la empresa, y mueve a dicho cargo al empleado subalterno más antiguo que encuentre. Si no encuentra ninguno, no hace ninguna modificación.

4.5.5. El Algoritmo de Recorrido en Preorden

Para implementar el método que recorre los elementos del árbol y retorna una colección con sus nombres, vamos a utilizar la técnica de acumulación de parámetros, repartiendo las responsabilidades entre las clases `Empresa` y `Cargo`, tal como se muestra a continuación:

```
public class Empresa
{
    public Collection darListaEmpleados( )
    {
        Collection lista = new ArrayList( );
        if( organigrama != null )
        {
            organigrama.darListaCargos( lista );
        }
        return lista;
    }
}
```

```
public class Cargo
{
    public void darListaCargos( Collection lista )
    {
        lista.add( nombreCargo );
        for( int i = 0; i < subalternos.size( ); i++ )
        {
            Cargo hijo = ( Cargo )subalternos.get( i );
            hijo.darListaCargos( lista );
        }
    }
}
```

Este método es responsable de crear el vector que va a contener los nombres de todos los cargos que existen en la empresa.
Luego, si el organigrama está vacío, retorna este vector sin elementos.
En caso contrario, pide a la raíz del organigrama que inicie el recorrido en preorden de sus elementos, pasándole como parámetro la estructura en donde debe ir acumulando la respuesta a medida que avanza el proceso.

Al iniciar este método, en la colección "lista" que llega como parámetro ya está una parte del recorrido del árbol.
Lo primero que hacemos es agregar al final de dicha colección el nombre del cargo que se encuentra en la raíz.
Después, pedimos a cada uno de los subárboles que se recorra en preorden, acumulando el resultado en la misma estructura que todos están compartiendo.
Si el elemento es una hoja, el método no entra al ciclo, puesto que el número de subalternos es cero.

4.5.6. Patrones de Algoritmo para Árboles n-arios

Para resolver problemas sobre árboles n-arios existen dos grandes familias de algoritmos, que siguen dos patrones principales: el patrón de **descenso recursivo** y el patrón de **recorrido total**. Los esqueletos de estos dos patrones se muestran a continuación:

```

if( condición1 )
{
    // Solución directa 1
}

else if( condición2 )
{
    // Solución directa 2
}

else
{
    // Inicialización de las variables en las que se van a
    // acumular/componer las respuestas parciales

    for( int i = 0; i < subárboles.size( ) && !condicion; i++ )
    {
        Nodo hijo = ( Nodo )subárboles.get( i );

        // Avance de la recursividad sobre el subárbol "hijo"

        // Acumulación/composición de la respuesta
    }
}

// Inclusión de la raíz en el recorrido total

for( int i = 0; i < subárboles.size( ); i++ )
{
    Nodo hijo = ( Nodo )subárboles.get( i );

    // Recorrido total sobre el subárbol "hijo"
}

```

El algoritmo tiene una o varias salidas de la recursividad, en las cuales es posible dar una solución directa al problema.

Puesto que no se conoce a priori el número de subárboles presentes, es necesario avanzar con un ciclo que se desplaza sobre ellos con la variable "i".

En el esqueleto suponemos que el vector de subárboles se llama "subárboles" y que cada elemento del árbol es de la clase "Nodo".

Dentro del ciclo hacemos la llamada recursiva sobre cada uno de los subárboles y acumulamos o componemos la respuesta con las obtenidas anteriormente.

El ciclo puede terminar porque se hizo la llamada recursiva sobre todos los subárboles o porque se cumple una condición que refleja que el problema ya se ha resuelto. Esto último muchas veces se remplaza por la instrucción de retorno dentro del ciclo.

El objetivo de los algoritmos que siguen este patrón es recorrer cada uno de los elementos del árbol haciendo una operación cualquiera sobre cada uno de ellos.

Antes de entrar al ciclo, el método incluye el elemento de la raíz en el recorrido.

En el ciclo nos vamos desplazando con la variable "i" sobre cada uno de los subárboles.

4.5.7. Extensiones al Caso de Estudio

A continuación planteamos un grupo de tareas de desarrollo para extender el programa que maneja organigramas. Si no lo ha hecho, copie del CD o del sitio web el archivo `n11_organigrama.zip` y cree el respectivo proyecto en Eclipse.

Tarea 15



Objetivo: Extender el programa que maneja el organigrama de una empresa.

Modifique el proyecto del caso de estudio (`n11_organigrama`) para que incluya el requerimiento funcional que se plantea a continuación.

1. Implemente en las clases `Empresa` y `Cargo` los métodos necesarios para agregar al programa un nuevo requerimiento funcional, que permita hacer la fusión de dos cargos que dependen del mismo jefe. El método debe recibir el nombre de los dos cargos, verificar que sean hermanos, pasar todos los subalternos del segundo cargo al primero y finalmente eliminar el segundo cargo. En caso de cualquier problema el método debe lanzar una excepción.
2. Desarrolle las pruebas automáticas unitarias para los métodos que acaba de implementar.
3. Asocie con el botón **Opción 1** el nuevo requerimiento funcional.

Tarea 16



Objetivo: Extender el programa que maneja el organigrama de una empresa.

Modifique el proyecto del caso de estudio (`n11_organigrama`) para que incluya el requerimiento funcional que se plantea a continuación.

1. Implemente en las clases `Empresa` y `Cargo` los métodos necesarios para agregar al programa un nuevo requerimiento funcional, que permita contar el número de empleados que tienen menos de un año en la empresa.
2. Desarrolle las pruebas automáticas unitarias para los métodos que acaba de implementar.
3. Asocie con el botón **Opción 2** el nuevo requerimiento funcional.

Tarea 17



Objetivo: Extender el programa que maneja el organigrama de una empresa.

Modifique el proyecto del caso de estudio (`n11_organigrama`) para que incluya el requerimiento funcional que se plantea a continuación.

1. Implemente en las clases `Empresa` y `Cargo` los métodos necesarios para agregar al programa un nuevo requerimiento funcional, que permita encontrar el ancestro común más próximo de dos empleados. Para esto debe recibir los códigos de identificación de los dos empleados y retornar el objeto de la clase `Empleado` que cumple que el cargo que éste ocupa es un ancestro del cargo que ocupan los dos empleados. Se debe verificar además que no hay otro empleado de menor nivel que satisfaga la misma condición.
2. Desarrolle las pruebas automáticas unitarias para los métodos que acaba de implementar.
3. Asocie con el botón **Opción 3** el nuevo requerimiento funcional.

Tarea 18

Objetivo: Extender el programa que maneja el organigrama de una empresa.

Modifique el proyecto del caso de estudio (`n11_organigrama`) para que incluya el requerimiento funcional que se plantea a continuación.

1. Implemente en las clases `Empresa` y `Cargo` los métodos necesarios para agregar al programa un nuevo requerimiento funcional, que permita encontrar al empleado de menor nivel en la empresa. Si hay varios en el mínimo nivel, debe retornar aquél de menor salario.
2. Desarrolle las pruebas automáticas unitarias para los métodos que acaba de implementar.
3. Asocie con el botón **Opción 4** el nuevo requerimiento funcional.

Tarea 19

Objetivo: Extender el programa que maneja el organigrama de una empresa.

Modifique el proyecto del caso de estudio (`n11_organigrama`) para que incluya el requerimiento funcional que se plantea a continuación.

1. Implemente en las clases `Empresa` y `Cargo` los métodos necesarios para agregar al programa un nuevo requerimiento funcional, que permita hacer un recorte en la empresa, eliminando todos los cargos que se encuentran por debajo de un nivel dado.
2. Desarrolle las pruebas automáticas unitarias para los métodos que acaba de implementar.
3. Asocie con el botón **Opción 5** el nuevo requerimiento funcional.

Tarea 20

Objetivo: Extender el programa que maneja el organigrama de una empresa.

Modifique el proyecto del caso de estudio (`n11_organigrama`) para que incluya el requerimiento funcional que se plantea a continuación.

1. Implemente en las clases `Empresa` y `Cargo` los métodos necesarios para agregar al programa un nuevo requerimiento funcional, que permita despedir al empleado que ocupa el nivel 1 y, luego, remplazarlo por el subalterno más antiguo. Este procedimiento se debe repetir de manera recursiva, hasta que la vacante que se acaba de abrir llegue al nivel más bajo posible.
2. Desarrolle las pruebas automáticas unitarias para los métodos que acaba de implementar.
3. Asocie con el botón **Opción 6** el nuevo requerimiento funcional.

5. Glosario de Términos

GLOSARIO



Complete las siguientes definiciones como una manera de resumir los conceptos aprendidos en el nivel. Asegúrese de entender bien el significado de cada uno de estos términos, puesto que son la base para poder continuar con los niveles que siguen en el libro.

Árbol binario ordenado:	Hijo:
Estructura recursiva:	Árbol vacío:
Recorrido en inorden:	Interfaz Comparable:
Altura de un árbol:	Patrón de descenso recursivo:
Peso de un árbol:	Patrón de avance ordenado:
Raíz de un árbol:	Técnica de acumulación de parámetros:
Subárbol:	Árbol n-ario:
Nivel:	Recorrido en preorden:
Algoritmo recursivo:	Camino:
Salida de la recursividad:	Rama:
Avance de la recursividad:	Ancestro:
Hoja:	Descendiente:
Padre:	Patrón de recorrido total:

6. Hojas de Trabajo



6.1. Hoja de Trabajo N° 1: Juego 8-Puzzle

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

El juego 8-puzzle es una versión reducida del 15-puzzle, un juego popular en muchas partes del mundo, aunque no siempre se conoce bajo ese nombre. El juego está formado por una caja cuadrada con ocho piezas móviles, también cuadradas, numeradas entre 1 y 8. El objetivo del juego es ordenar las piezas de 1 a 8 realizando el desplazamiento de una pieza a la vez, utilizando el único espacio libre disponible. Las piezas no se pueden sacar de la caja, así que no es posible ordenarlas de cualquier forma.

Se quiere hacer una aplicación del juego 8-puzzle en la cual el usuario pueda jugar y además consultar al computador para que le sugiera una jugada. La interfaz de usuario del programa debe ser la siguiente:

usuario debe poder: (1) mover una pieza del 8-puzzle según las reglas del juego, (2) comenzar un nuevo juego y (3) pedir al computador una sugerencia para el siguiente movimiento. En modo inteligente el usuario debe poder: (4) regenerar el árbol de solución del juego y (5) visualizar la ejecución paso a paso de la solución del juego, en caso de que con el árbol actual se haya llegado a una.

Para realizar un movimiento el usuario debe hacer clic sobre una de las piezas y, si ésta se encuentra al lado del espacio disponible, se mueve a dicha posición. Para sugerir una jugada se utilizará un algoritmo sencillo: a partir del estado actual del juego se construye un árbol n-ario con todos los estados alcanzables, hasta una altura definida por el usuario.

Luego, sobre este árbol, se busca el nodo con el estado de terminación o, si no lo encuentra, localiza aquél que esté más cerca de la solución. La jugada sugerida por el computador es la raíz del árbol que lleva a ese nodo. Todo esto se basa en un valor que se le da a cada estado y que indica qué tan cerca está ese estado del final del juego.

Al final del juego las piezas deben quedar ordenadas ascendentemente de izquierda a derecha y de arriba a abajo, y el espacio libre debe quedar en la esquina inferior derecha.

El algoritmo presentado para sugerir jugadas tiene algunos problemas que deben ser tenidos en cuenta. Por una parte no es muy sencillo estimar qué tan cerca se está de la

solución en un momento dado: con la métrica utilizada es posible recorrer caminos que parecen estar llevando



El programa debe permitir al usuario jugar en dos modos: manual e inteligente. En el modo manual el

a la solución pero que realmente llegan a estados con valores cercanos pero que necesitan muchísimas jugadas adicionales para convertirse en solución. La solución a esto sería buscar mejores métricas o utilizar algoritmos más inteligentes que no se limiten solamente a buscar en el árbol usando fuerza bruta. Otro problema es que la generación de jugadas no tiene en cuenta que se puede

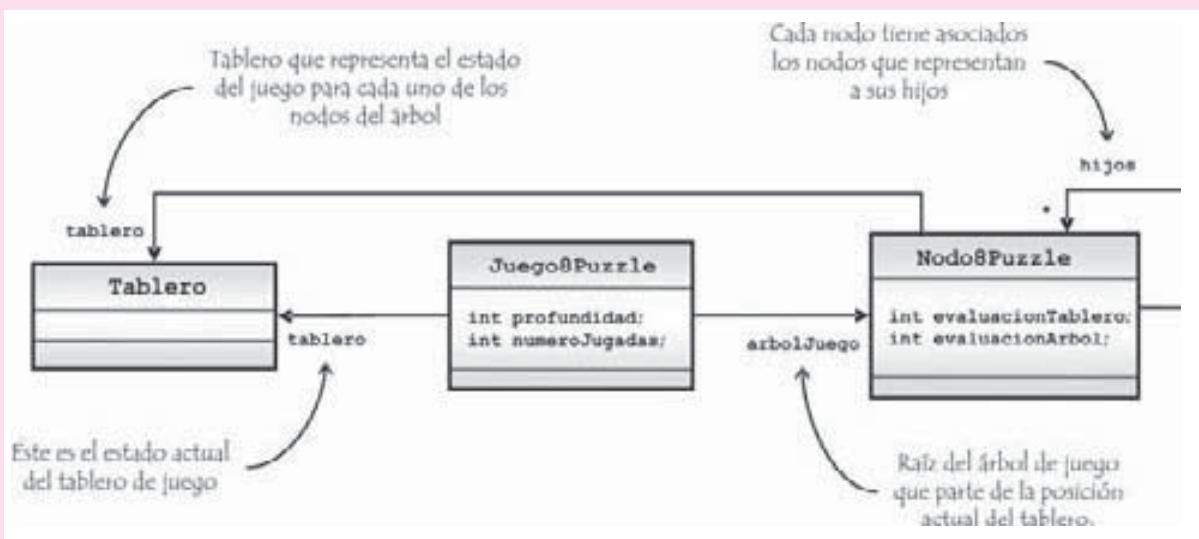
llegar al mismo estado siguiendo diferentes caminos, así que el árbol puede tener muchos más nodos que el número de estados posibles del juego. Para solucionar esto el árbol debería ser podado para asegurar que siempre se tome el camino más corto hasta un nodo determinado, pero esto requeriría algoritmos más complicados que los que queremos utilizar en esta hoja de trabajo.

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

Requerimiento funcional 1	Nombre	R1 – Mover una pieza
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Comenzar un nuevo juego
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Pedir una sugerencia para el siguiente movimiento
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Regenerar el árbol de solución del juego
	Resumen	
	Entrada	
	Resultado	

Requerimiento funcional 5	Nombre	R5 – Visualizar la ejecución paso a paso de la solución del juego
	Resumen	
	Entrada	
	Resultado	

Modelo conceptual. Estudie el diagrama de clases que se presenta a continuación. Asegúrese de entender el papel que juega cada entidad dentro del modelo, así como el significado de cada atributo y asociación.



Clase	Atributo	Descripción
Juego8Puzzle	profundidad	Representa el número máximo de niveles del árbol de juego. Con este atributo se controla el número de jugadas hacia adelante que va a calcular y considerar el programa dentro de sus sugerencias.
Juego8Puzzle	numeroJugadas	Indica el número de jugadas que se han hecho hasta el momento.
Nodo8Puzzle	evaluacionTablero	Es un valor que indica qué tan cerca de la posición final se encuentra un tablero (una estimación del número de jugadas que le faltan para ganar). Dependiendo de la manera como se calcule este valor, el algoritmo de búsqueda puede parecer más o menos inteligente.
Nodo8Puzzle	evaluacionArbol	Es un valor que indica la evaluación del mejor tablero que hace parte del árbol. Un árbol va a ser tan bueno como el mejor de los tableros a los que puede llegar.

Implementación. Vamos ahora a estudiar en detalle la clase Tablero. Estudie la siguiente declaración de la clase y desarrolle los métodos que se piden más adelante.

```
public class Tablero
{
    // -----
    // Constantes
    // -----

    /** Constante usada para identificar el hueco en el tablero */
    public final static int HUECO = -1;

    /** Movimiento del hueco hacia la izquierda */
    public final static int IZQUIERDA = 0;

    /** Movimiento del hueco hacia la derecha */
    public final static int DERECHA = 1;

    /* Movimiento del hueco hacia arriba */
    public final static int ARRIBA = 2;

    /** Movimiento del hueco hacia abajo */
    public final static int ABAJO = 3;

    /** Movimiento inválido en el tablero actual */
    public final static int INDEFINIDO = -1;

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

    /** La matriz que representa el estado actual del tablero */
    private int[][] tablero;

    /** Coordenada X del hueco en el tablero */
    private int huecoX;

    /** Coordenada Y del hueco en el tablero */
    private int huecoY;
}
```

Este método indica si el movimiento en la dirección especificada es válido. El parámetro dirección corresponde a una de las constantes antes descritas.

```
public boolean movimientoValido( int direccion )
{
}
```



El estado del tablero se va a representar con una matriz de 3 x 3, en cada una de cuyas casillas se tendrá el número de la ficha que allí se encuentra. Para el tablero que se presentó en el enunciado, la representación sería:

1	6	2
7	4	5
8	-1	3



Para marcar la posición que se encuentra vacía, utilizamos la constante HUECO.



Cualquier jugada que se haga sobre el tablero se puede representar a través de cuatro constantes que indican la dirección hacia la cual debe moverse la posición vacía: IZQUIERDA, DERECHA, ARRIBA y ABAJO. Esto va a simplificar el problema de indicar cuál es la jugada que hizo el usuario.

Este método determina si el tablero está en la posición final del juego.

```
public boolean verificarFinal( )
```

```
{
```

Este método hace en el tablero un movimiento en la dirección descrita en el parámetro.

```
public void mover( int direccion )
```

```
{
```

```
}
```

Este método estima qué tan lejos se encuentra el tablero de la posición final de juego. Para esto utiliza la distancia Maniatan, que consiste en sumar la distancia en una grilla de cada ficha a su posición final. Si la función vale 0, significa que éste es un estado del juego en el que ya se terminó.

Para el tablero:

1	6	2
7	4	5
8	-1	3

la distancia se calcula así:

- para el 1: 0
- para el 2: 1
- para el 3: 2
- para el 4: 1
- para el 5: 1
- para el 6: 2
- para el 7: 1
- para el 8: 1

```
}
```

Lo que en total nos da el valor 9, que indica que faltan por lo menos nueve jugadas para llegar al final del juego.

Implementación. Pasemos ahora a trabajar sobre la clase `Nodo8Puzzle`, que representa cada uno de los elementos del árbol de juego. Para esto vamos a comenzar estudiando algunos de los métodos que allí se implementan.

- Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n11_juego8Puzzle.zip`, que se encuentra en el CD que acompaña al libro. Localice la clase `InterfazJuegoPuzzle` y ejecute el programa. Juegue en modo manual y en modo inteligente, y estudie el comportamiento del programa.
- Edité en Eclipse la clase `Nodo8Puzzle` y responda las siguientes preguntas:

Estudie el método constructor de la clase. Es la primera vez que tenemos un método constructor que es recursivo. ¿Qué representa cada uno de los parámetros?
 ¿Cómo se construyen todos los tableros a los que se puede llegar en una jugada?
 ¿Cómo nos aseguramos de que esos tableros sean válidos?

¿Para qué se utiliza el método `darNumeroSoluciones()`? Explique su planteamiento recursivo.

¿Cómo funciona el método que sugiere una jugada? ¿Qué valor se almacena en el atributo `direccionJugada`?

¿Cómo funciona el método que aumenta el número de niveles del árbol? ¿Qué recibe como parámetro este método?

¿Cuál es la principal responsabilidad del método `avanzarArbol()`? ¿Por qué invoca el método que aumenta el número de niveles del árbol? ¿Qué recibe como parámetro el método?

¿Qué retorna el método `darSolucion()`? ¿Qué va en cada casilla del arreglo? ¿Cómo se calculan?

Implementación. Desarrolle los métodos de la clase Nodo8Puzzle que se plantean a continuación.

Este método cuenta el número de nodos que tienen exactamente cuatro hijos.

```
public int conCuatroHijos( )
{
}
```

Este método cuenta el número de nodos cuya evaluación es igual al valor que se recibe como parámetro.

```
public int contarNodosIgualEvaluacion( int eval )
{
}
```

Este método busca en el árbol el nodo que tenga la mejor evaluación y lo retorna. Si hay más de uno con la misma evaluación, retorna cualquiera de ellos.

```
public Nodo8Puzzle localizarElMejor( )
{
}
```

Este método calcula y retorna el número de nodos cuyo tablero tiene un hueco en el centro.

```
public int contarNodosHuecoCentro( )  
{  
}  
}
```

Implementación. Extienda el programa con los requerimientos funcionales que se explican a continuación.

1. Asocie con el botón **Opción 1** un nuevo requerimiento funcional, que permita mostrar en un diálogo la lista de movimientos de cada una de las soluciones que hay desde el estado en el que se encuentra el juego. Por ejemplo, si en el estado actual del juego hay dos soluciones posibles, el diálogo debería mostrar la siguiente información:
Posibles soluciones del juego:
Solución 1: ARRIBA, IZQUIERDA, ABAJO
Solución 2: ABAJO, ABAJO, DERECHA, DERECHA
2. Asocie con el botón **Opción 2** un nuevo requerimiento funcional, que permita seleccionar si el puzzle se va a ordenar en forma ascendente o descendente. El cambio de la forma de ordenamiento sólo debe permitirse al inicio del juego (es decir, cuando el número de jugadas sea cero). Sin importar la forma de ordenamiento seleccionada, todas las opciones existentes en la aplicación deben continuar funcionando de forma correcta. ¿Qué cambios habría que hacer al programa para que el impacto sea mínimo?
3. Asocie con el botón **Opción 3** un nuevo requerimiento funcional, que permita reiniciar el juego. Esta acción implica que el juego debe volver al estado en el que se encontraba cuando se inició.



6.2. Hoja de Trabajo N° 2: Juego Genético

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

El programa que se describe a continuación es un juego de estrategia para ser jugado por una sola persona basado en las leyes de Mendel. El objetivo es obtener una persona con ciertas características, a partir de un conjunto de personas (individuos) iniciales entre las cuales el jugador realiza entrecruzamientos.

El juego se realiza en varios niveles, y en cada uno hay un objetivo que se debe alcanzar. Dependiendo de qué tan rápido se alcance este objetivo, el jugador ganará una cierta cantidad de puntos. También se ganarán puntos por cada homocigoto recesivo (este concepto se estudia más adelante) que haya en el individuo ganador.

Además del juego propiamente dicho, en este programa es interesante ver la aplicación de las leyes de Mendel y cómo ciertas características pasan de una generación a las siguientes. Para tal fin será posible conocer la familia de cada uno de los individuos identificando algunas de sus características.

Características de los individuos

El juego se desarrolla alrededor de cinco características físicas definidas por cinco genes diferentes:

- forma de los ojos: pueden ser redondos (RECESIVO) o rasgados (DOMINANTE)
- color de piel: puede ser oscuro (DOMINANTE) o claro (RECESIVO)
- color del pelo: puede ser verde (RECESIVO) o morado (DOMINANTE)
- altura: puede ser alto (DOMINANTE) o bajo (RECESIVO)
- enfermedad: los individuos pueden estar afectados por una enfermedad o no.

De estas características, solamente las cuatro primeras se transmiten siguiendo las leyes de Mendel. El gen de la enfermedad tiene una forma de transmisión análoga a la del gen que causa la hemofilia y sigue las siguientes reglas:

- solamente los hombres pueden estar enfermos.
- solamente las mujeres en cuyo genotipo se encuentra la enfermedad pueden transmitirla.
- la enfermedad está asociada al alelo que proviene de la madre.

De esto se derivan varios puntos importantes:

- las mujeres nunca pueden padecer la enfermedad (no se puede manifestar en su fenotipo).
- los hombres enfermos nunca transmitirán esta enfermedad.
- una mujer con el alelo enfermo siempre transmitirá la enfermedad.
- todo hombre cuya madre sea portadora de la enfermedad presentará la enfermedad.

La interfaz de usuario del programa debe ser la que se presenta a continuación.



Para cada uno de ellos podemos visualizar y consultar sus características. El tercer individuo, por ejemplo, es un hombre, tiene los ojos rasgados, la piel oscura, el pelo morado, es alto y está enfermo. Cada individuo del juego tiene un identificador que es único.

Cuando se quieran ver los detalles de la familia de un individuo (botón **Ver Más Detalles**), debe aparecer una ventana similar a la siguiente. Allí se encuentra la siguiente información: (1) el identificador del individuo, (2) el número de generaciones en la familia, (3) el número de personas enfermas en la familia, (4) los hombres de la familia con sus características, (5) las mujeres de la familia con sus características, (6)



el número de homocigotos recesivos del individuo, (7) el número de antepasados y (8) la generación más vieja en la cual se presentó por primera vez la enfermedad.

Reglas del juego

Puede haber como máximo 20 individuos con los cuales se realice el entrecruzamiento. Si se necesita espacio para más individuos se deben eliminar algunos de los ya existentes. Al eliminar uno de los individuos no pasa nada.

Un nivel termina cuando se realiza un entrecruzamiento que da como resultado un individuo que tiene un aspecto exterior igual al del objetivo. Esto quiere decir que para todas las características el individuo obtenido y el objetivo tienen el mismo fenotipo.

Al finalizar un nivel se asignan 20 puntos por alcanzar el objetivo y un punto adicional por cada homocigoto recesivo en la familia del individuo igual al objetivo. Sin embargo, también se resta un punto por cada entrecruzamiento que se haya tenido que hacer para alcanzar el objetivo.

Comprepción del dominio del problema. Consulte en una enciclopedia o por Internet las leyes de Mendel y conteste las preguntas que se plantean a continuación, utilizando como contexto lo planteado en el enunciado.

¿Qué es un alelo? Dé un ejemplo dentro del contexto de nuestro caso de estudio

¿Qué es el genotipo? Dé un ejemplo dentro del contexto de nuestro caso de estudio.

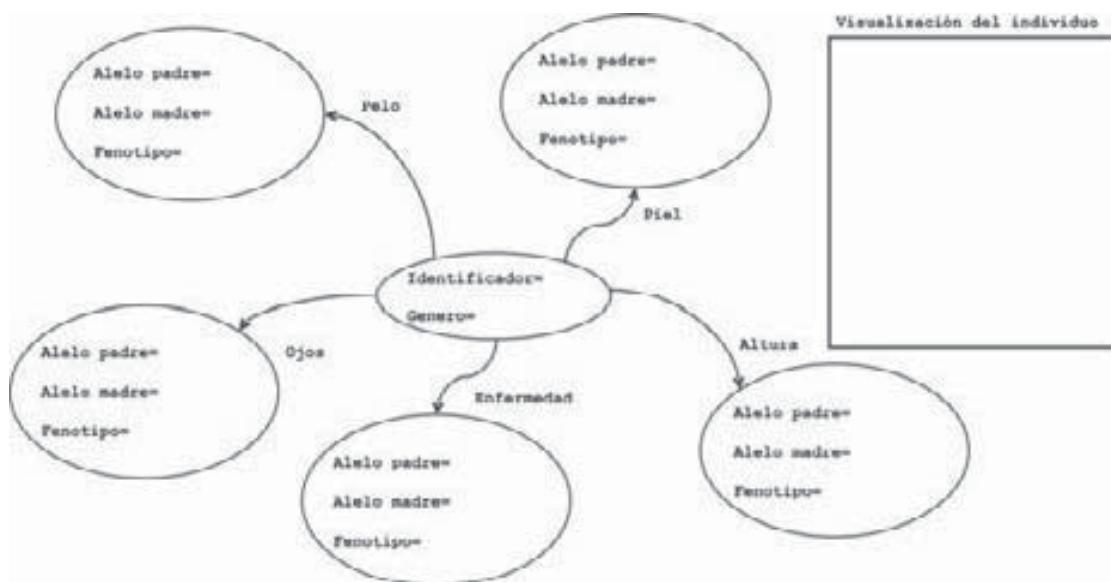
¿Qué es el fenotipo? Dé un ejemplo dentro del contexto de nuestro caso de estudio.

¿Qué es un genotipo homocigoto dominante?

¿Qué es un genotipo homocigoto recesivo?

¿Qué es un genotipo heterocigoto?

Dé un ejemplo completo de las características de un individuo completando el dibujo que se muestra. Asegúrese de entender cada uno de los componentes. Explique claramente la manera de calcular el fenotipo en cada una de las características.



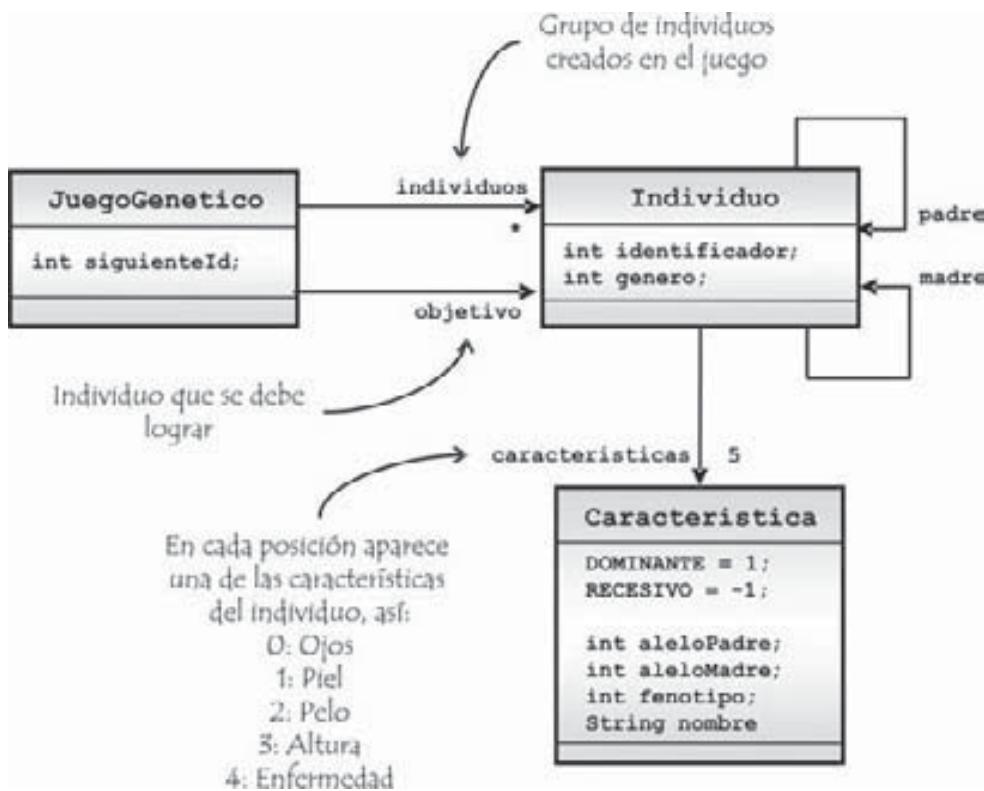
Explique el proceso aleatorio que define las características de un hijo, dada la información de sus padres.

¿Cuántos hijos distintos puede tener una pareja de individuos?

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

Requerimiento funcional 1	Nombre	R1 – Comenzar un nuevo juego
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Realizar el entrecruzamiento de dos individuos dados
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Eliminar del juego a un individuo dado
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Consultar las características de un individuo
	Resumen	
	Entrada	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Consultar la información de la familia de un individuo
	Resumen	
	Entrada	
	Resultado	

Modelo conceptual. Estudie el diagrama de clases que se presenta a continuación. Asegúrese de entender el papel que juega cada entidad dentro del modelo, así como el significado de cada atributo y asociación.



Clase	Atributo	Descripción
JuegoGenetico	siguienteId	Es un atributo de tipo entero que contiene el identificador que se le asignará al siguiente individuo que se cree.
Individuo	identificador	Es el identificador único de cada individuo.
Individuo	genero	Puede ser hombre o mujer.
Caracteristica	aleloPadre	Es un entero que puede tomar como valor una de las constantes DOMINANTE o RECESIVO. Es el alelo proporcionado por el padre para determinar la característica.
Caracteristica	aleloMadre	Es un entero que puede tomar como valor una de las constantes DOMINANTE o RECESIVO. Es el alelo proporcionado por la madre para determinar la característica.
Caracteristica	fenotipo	Es un entero que puede tomar como valor una de las constantes DOMINANTE o RECESIVO. Este atributo representa el fenotipo del individuo en esta característica.
String	nombre	Es el nombre de la característica (por ejemplo, "Ojos").

Implementación. Vamos a comenzar por estudiar la clase Caracteristica. Para esto cree el respectivo proyecto en Eclipse y edite el contenido de dicha clase.

¿Cuántos constructores tiene la clase? ¿En qué caso se utiliza cada uno de ellos?

¿Cuál es el objetivo del método llamado sortearAlelo()?

¿Qué responsabilidad tiene el método calcularFenotipo()? ¿Qué parámetros recibe?

¿Qué responsabilidad tiene el método calcularFenotipoEnfermedad()? ¿Qué parámetros recibe?

¿Cómo se calcula el genotipo de un individuo con respecto a una característica? ¿Qué método lo hace?

Implementación. Vamos a estudiar ahora la clase Individuo. Para esto, edite desde Eclipse su contenido.

¿Cuántos constructores tiene la clase? ¿En qué caso se utiliza cada uno de ellos?

¿Cuál es el objetivo del método generarCaracteristicasAleatorias()? ¿Qué retorna el método? ¿Cómo calcula el resultado?

¿Cuál es el objetivo del método heredarCaracteristicas()? ¿Qué retorna el método? ¿Cómo calcula el resultado?

¿Cuál es el objetivo del método generarCaracteristicaHijo()? ¿Qué retorna el método? ¿Cómo calcula el resultado?

<p>¿Cuál es el objetivo del método <code>generarCaracteristicaHijo()</code>? ¿Qué retorna el método? ¿Cómo calcula el resultado?</p>	
<p>Estudie el método llamado <code>darMujeresFamilia()</code>. ¿Qué patrón de algoritmo utiliza? ¿Para qué sirve el parámetro que recibe?</p>	
<p>Estudie el método llamado <code>darNumeroAntepasados()</code>. ¿Qué patrón de algoritmo utiliza?</p>	
<p>Estudie el método llamado <code>darGeneracionMasViejaEnferma()</code>. ¿Qué patrón de algoritmo utiliza?</p>	

Implementación. Desarrolle para la clase `Individuo` los métodos que se plantean a continuación.

Este método calcula la lista de ancestros que son altos. Utiliza la técnica de acumulación de parámetros.

```
public void darAncestrosAltos( List respuesta )
{
    }
}
```

Este método retorna el ancestro común más próximo, que comparten el individuo que recibe la invocación del método y el individuo que llega como parámetro. Si no comparten ninguno, el método retorna el valor null.

```
public Individuo darAncestroComun( Individuo ind )
```

```
{
```

```
}
```

Este método retorna el número de ancestros del individuo que tienen el pelo de color verde.

```
public void contarAncestrosVerdes( )
```

```
{
```

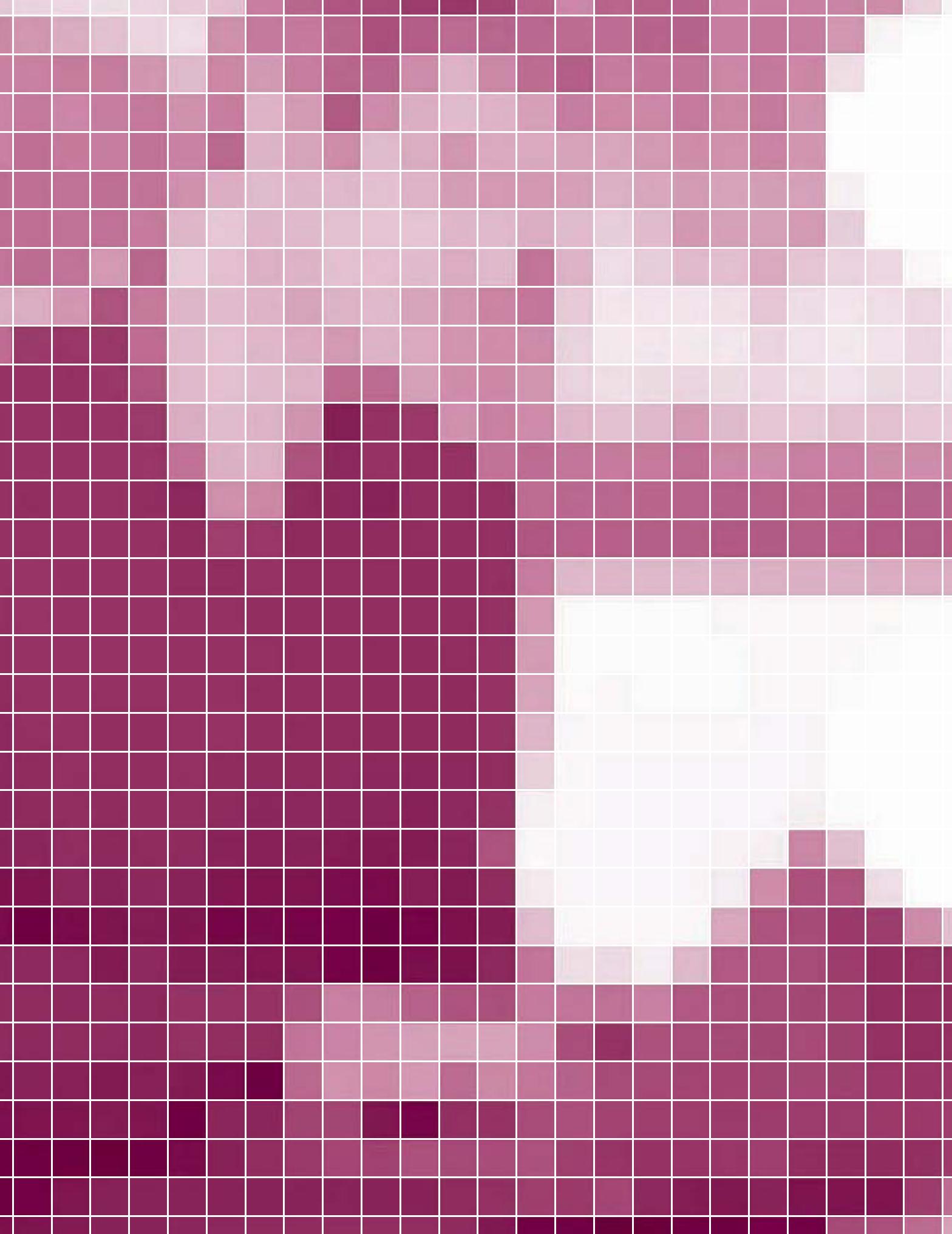
```
}
```

Este método retorna el número de personas de la familia que tienen por lo menos una característica en la cual son homocigotos recesivos.

```
public void darNumeroHomocigotosRecesivos( )  
{  
}  
}
```

Este método indica si son familiares el individuo que recibe la invocación y el individuo que se recibe como parámetro.

```
public boolean sonFamiliares( Individuo ind )  
{  
}  
}
```



Nivel 6

Bases de Datos y Distribución Básica

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

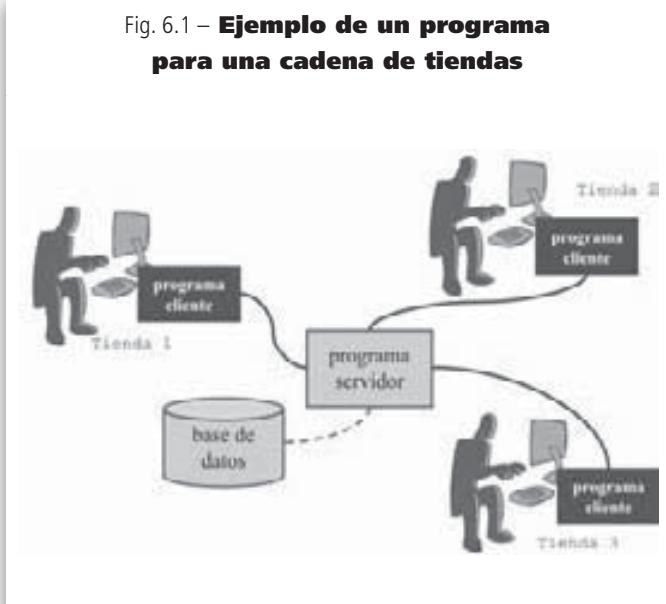
- Construir un programa simple que utilice un canal (*socket*) para comunicarse con otro programa que se ejecuta sobre un computador distinto, con la condición de que ambas máquinas se encuentren conectadas a una red.
- Desarrollar un programa que maneje concurrencia, de manera que sea posible que ejecute más de una parte del programa de manera simultánea, utilizando hilos de ejecución (*threads*).
- Construir un programa que almacene su información persistente en una base de datos elemental.
- Entender el papel que juega un programa servidor, al cual múltiples programas ejecutados por distintos usuarios se conectan para solicitar servicios.
- Integrar toda la teoría vista y las habilidades desarrolladas en los niveles anteriores, para resolver un problema un poco más complejo, que incluye algunos requerimientos no funcionales de persistencia, distribución y concurrencia.

2. Motivación

Hay dos requerimientos no funcionales que resultan críticos en el momento de diseñar un programa: la persistencia y la distribución. El primero se refiere a la manera de almacenar en algún medio secundario la información de un programa, mientras el segundo hace referencia al caso en el cual es necesario construir un programa como un conjunto de ellos, los cuales se deben poder ejecutar en computadores distintos por alguna condición del problema.

Imagine, por ejemplo, que queremos construir un programa para que una cadena de tiendas pueda cobrar a sus clientes por los productos que ha comprado, tal como se sugiere en la figura 6.1. El programa debe estar compuesto por dos partes: un programa llamado servidor, que almacena el inventario de la cadena de tiendas, y un programa llamado cliente que se ejecuta en el computador de cada una de las tiendas. Dichos programas se deben poder comunicar a través de una red para enviar información en ambos sentidos. El servidor es responsable de la persistencia de la información y de comunicarse simultáneamente con todas las tiendas con las cuales se encuentra conectado, para recibir de ellas alguna información

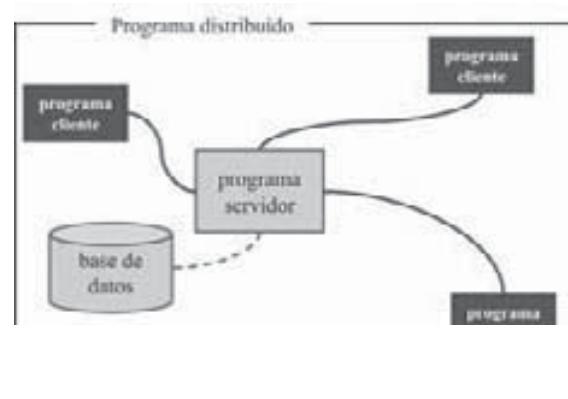
Fig. 6.1 – **Ejemplo de un programa para una cadena de tiendas**



(cantidad de un producto vendido) o enviar la información que una tienda pueda necesitar (el precio de un producto o una autorización de pago con tarjeta de crédito). El programa cliente, por su parte, se encarga de recibir la información del usuario y mantener la comunicación con el servidor.

Un programa distribuido se debe ver entonces como el conjunto de programas que hacen parte de él, con las diferentes ejecuciones y conexiones que se hagan entre ellos, tal como se muestra en la figura 6.2 para el ejemplo de la cadena de tiendas.

Fig. 6.2 – **Visión de un programa distribuido**



En la construcción de un programa distribuido debemos tener en cuenta tres aspectos muy importantes, los cuales constituyen el tema central de este nivel: el primer aspecto tiene que ver con la capacidad del servidor de atender simultáneamente a todos los clientes. Ese tema se denomina **conurrencia** y se puede ver como la capacidad de ejecutar al mismo tiempo múltiples partes de un programa. El servidor debe poder estar recibiendo de un cliente una información y al mismo tiempo estar respondiendo a otro una consulta que hizo. El segundo aspecto tiene que ver con la capacidad de un programa para comunicarse con otro que se puede encontrar en otro computador. Ese tema se denomina **distribución**.

En ese caso, un programa se debe poder conectar a otro, enviarle información y esperar una respuesta. Eso lo haremos en este nivel utilizando **canales de comunicación** (*sockets*). Por último tenemos el aspecto de la **persistencia**. Puesto que es importante que la información se almacene permanentemente en memoria secundaria y que sabemos que dicha información puede ser modificada y consultada por múltiples usuarios a la vez, es necesario utilizar una solución más sólida que los archivos que hemos usado hasta este momento. Una **base de datos** es una estructura en memoria secundaria construida pensando en los problemas de acceso múltiple y simultáneo a la información que almacena. En este nivel trabajaremos con los elementos básicos de una base de datos relacional, y utilizaremos un subconjunto del lenguaje SQL para manipularla.

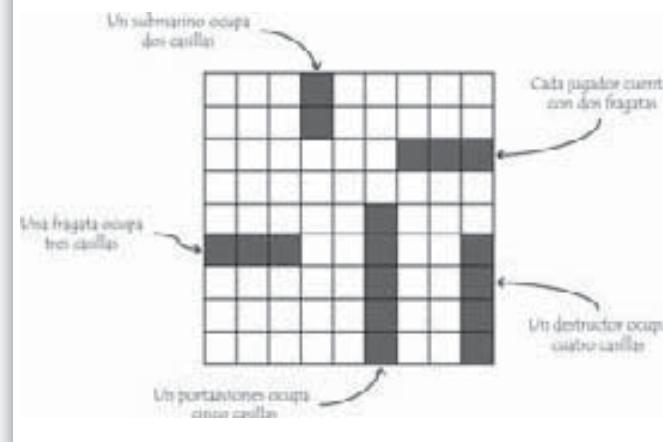
Este nivel no se puede considerar en ningún caso como un estudio completo de las problemáticas que trata. Cualquiera de los temas aquí presentados son abordados por libros y cursos avanzados, en los cuales se muestra el caso general dentro del cual se enmarca lo que vamos a ver más adelante. Nuestro objetivo es sólo dar una visión global de la estructura de los programas distribuidos que utilizan una base de datos como soporte, dando algunos elementos básicos con los cuales se puede construir una solución simple.

3. Caso de Estudio N° 1: Juego Batalla Naval

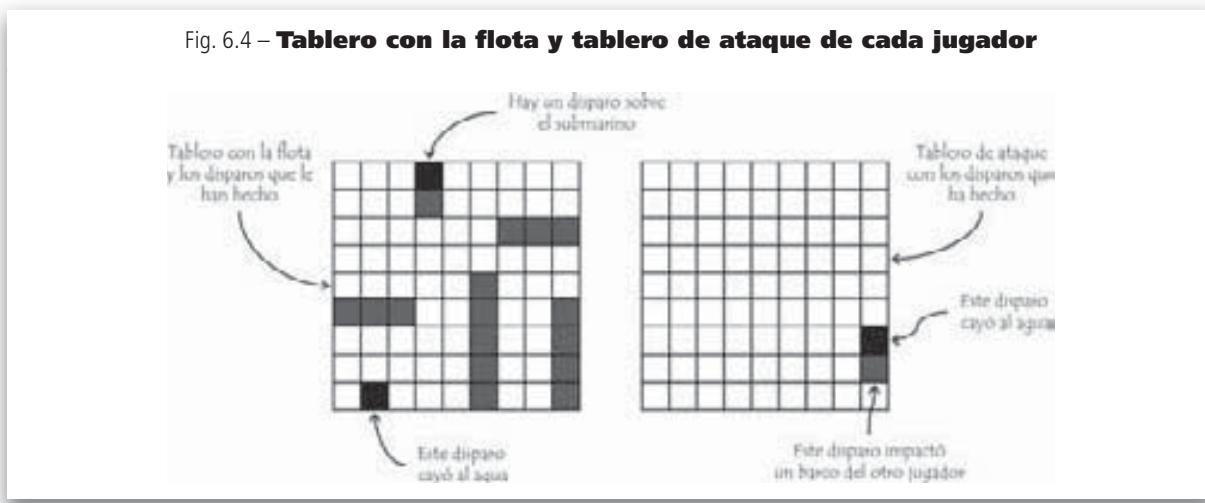
El objetivo de este caso de estudio es construir un programa distribuido que permita a un grupo de personas jugar por parejas Batalla Naval, permitiendo que cada una de ellas juegue desde un computador distinto. Como un servicio adicional, el programa debe almacenar información sobre los partidos perdidos y ganados de cada uno de los participantes.

El juego se lleva a cabo sobre un tablero de 9 x 9 posiciones, en el que cada jugador tiene situados cinco barcos (un portaaviones, un destructor, dos fragatas y un submarino), cada uno ocupando algunas de las casillas del tablero, dependiendo de su tamaño. En la figura 6.3 aparece un ejemplo con el posible tablero de un jugador.

Fig. 6.3 – **Tablero de juego para la batalla naval y forma de los barcos**

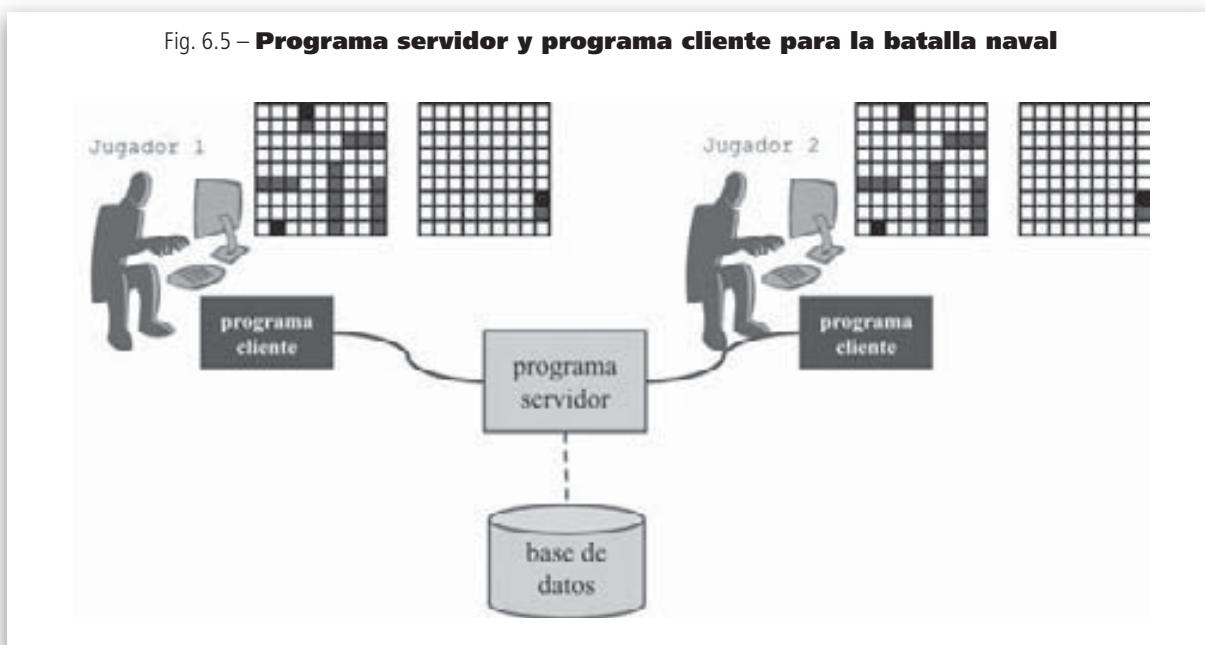


Al iniciar el juego el computador debe situar de manera aleatoria en el tablero la flota de barcos de cada uno de los jugadores que va a participar en la batalla (cada jugador conoce su tablero, pero no el del oponente). El objetivo del juego es hundir todos los barcos del otro jugador. Para esto, en cada turno, uno de los jugadores hace un disparo a una casilla del tablero del oponente. El disparo puede caer al agua o impactar alguno de los barcos contrarios. Para hundir un barco debe haber un impacto sobre cada una de las casillas que lo componen. Uno de los jugadores comienza el juego y se van turnando hasta que alguno de los dos haya logrado hundir la flota del otro jugador. En todo momento un jugador tiene un tablero con sus barcos y los disparos que le han hecho y otro tablero (llamado de ataque) con los barcos que ya ha hundido y los disparos que ha hecho, tal como se muestra en la figura 6.4.

Fig. 6.4 – **Tablero con la flota y tablero de ataque de cada jugador**

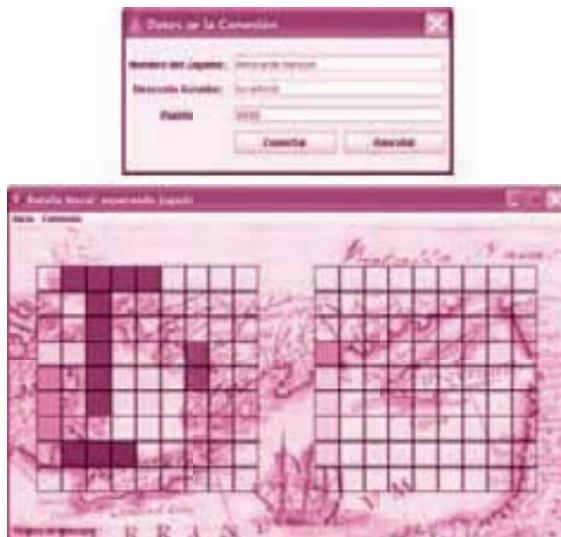
El programa distribuido debe constar de dos partes: un programa servidor, encargado de mantener la información estadística del juego y de permitir a los jugadores

encontrarse para una partida, y un programa cliente, a través del cual un usuario puede jugar Batalla Naval. En la figura 6.5 se muestran las dos partes del programa.

Fig. 6.5 – **Programa servidor y programa cliente para la batalla naval**

El programa cliente debe ofrecer las siguientes opciones al usuario: (1) Conectarse al servidor. Para esto el usuario debe suministrar un nombre, la dirección IP del servidor al cual se quiere conectar y el puerto por el cual dicho servidor se encuentra esperando las conexiones. (2) Disparar sobre una casilla del tablero

de ataque, en donde se encuentra la flota naval del contrincante. Como respuesta a esta acción, el programa debe informar si el disparo dio en algún blanco o cayó al agua. (3) Visualizar el estado de la partida. En la figura 6.6 aparece la interfaz de usuario para el programa cliente.

Fig. 6.6 – **Interfaz de usuario del programa cliente**

- Al iniciar una partida, el jugador debe suministrar un nombre, la dirección IP del servidor ("localhost" indica que el servidor está en el mismo computador del jugador) y el puerto por el cual se encuentra esperando conexiones el servidor.
- A la izquierda se encuentra el tablero del jugador y a la derecha el tablero de ataque.
- En el tablero del jugador aparecen marcadas las casillas en las que se encuentran cada uno de sus navíos. Con distintos colores se muestran los disparos del contrincante que han caído al agua y los disparos que han dado en alguno de los barcos.
- En el tablero de ataque se muestran los disparos que ha hecho el jugador, distinguiendo aquéllos que han dado en un navío enemigo de aquéllos que han caído al agua.

El programa servidor, por su lado, debe esperar a que los jugadores se vayan conectando y, por cada pareja que pueda armar, inicia un encuentro. Además, el servidor debe ofrecer las siguientes opciones sobre su interfaz de usuario (ver figura 6.7): (1) Mostrar el estado de todos los partidos que se encuentren en curso. De cada uno de ellos debe mostrar el nombre del jugador

y el número de disparos exitosos hasta ese momento. (2) Mostrar las estadísticas históricas del juego. Allí debe aparecer el nombre de cada jugador que haya participado en un encuentro, con el número de batallas que ha ganado, el número de batallas que ha perdido y el porcentaje de victorias que tiene en su historia de juegos.

Fig. 6.7 – **Interfaz de usuario del programa servidor**

- Para refrescar la información de los encuentros, se debe oprimir el botón Refrescar, que se encuentra en la parte superior.
- En la ventana de ejemplo se puede ver que en este momento sólo se está desarrollando un encuentro, entre el "Capitán Jack Sparrow" y "Barbanegra", y que el primero ha dado dos veces en un blanco, mientras el segundo sólo una.
- En la parte de abajo aparece la información histórica del juego, la cual se puede actualizar con el respectivo botón de refresco.
- Allí aparecen en el ejemplo tres jugadores, cada uno con el número de encuentros ganados, el número de encuentros perdidos y el porcentaje de victorias logradas.

Un programa cliente se puede encontrar en cuatro estados posibles: (1) desconectado del servidor, (2) conectado al servidor y esperando a que llegue un oponente, (3) jugando un encuentro y con el turno de disparar o (4) jugando un encuentro y esperando a que el contrincante haga su jugada.

La información histórica del juego debe ser persistente, de manera que cada vez que se ejecute de nuevo el servidor, los datos estadísticos deben aparecer en la ventana.

3.1. Objetivos de Aprendizaje

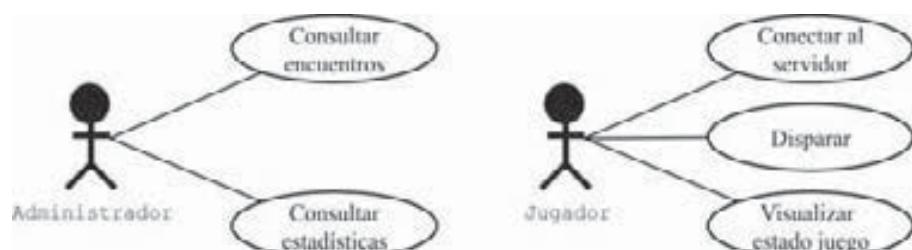
¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none"> ■ Construir el programa servidor de manera que soporte concurrencia. 	<ul style="list-style-type: none"> ■ Estudiar la manera en Java de manejar hilos de ejecución simultáneos (<i>threads</i>).
<ul style="list-style-type: none"> ■ Comunicar el programa servidor con las distintas ejecuciones del programa cliente. 	<ul style="list-style-type: none"> ■ Estudiar la forma de utilizar canales de comunicación (<i>sockets</i>) para comunicar programas que se ejecutan sobre máquinas distintas.
<ul style="list-style-type: none"> ■ Almacenar la información estadística de los resultados del juego. 	<ul style="list-style-type: none"> ■ Estudiar la estructura que tiene una base de datos. ■ Aprender a utilizar el lenguaje de consultas llamado SQL y la manera de invocarlo desde Java utilizando el framework JDBC.

3.2. Comprensión de los Requerimientos

Cada uno de los programas tiene sus propios requerimientos funcionales, los cuales se ilustran en el diagrama de casos de uso que aparece en la figura 6.8. En

dicho diagrama se puede apreciar que hay dos actores, uno asociado con cada parte del programa. El primero, llamado **Administrador**, representa al usuario que va a manipular la interfaz de usuario del programa servidor. El segundo, llamado **Jugador**, corresponde al usuario que va a utilizar el programa cliente.

Fig. 6.8 – Diagrama de casos de uso de la Batalla Naval



Tarea 1

Objetivo: Entender el problema del caso de estudio del directorio de contactos.

(1) Lea detenidamente el enunciado del caso de estudio del directorio de contactos e (2) identifique y complete la documentación de los requerimientos funcionales.

Requerimiento funcional 1	Nombre	R1 – Consultar los encuentros	Actor	Administrador
	Resumen			
	Entrada	Ninguna		
	Resultado			
Requerimiento funcional 2	Nombre	R2 – Consultar las estadísticas	Actor	Administrador
	Resumen			
	Entrada	Ninguna		
	Resultado			
Requerimiento funcional 3	Nombre	R3 – Conectar al servidor	Actor	Jugador
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 4	Nombre	R4 – Disparar	Actor	Jugador
	Resumen			
	Entrada	Ninguna		
	Resultado			
Requerimiento funcional 5	Nombre	R5 – Visualizar el estado del juego	Actor	Jugador
	Resumen			
	Entrada	Ninguna		
	Resultado			

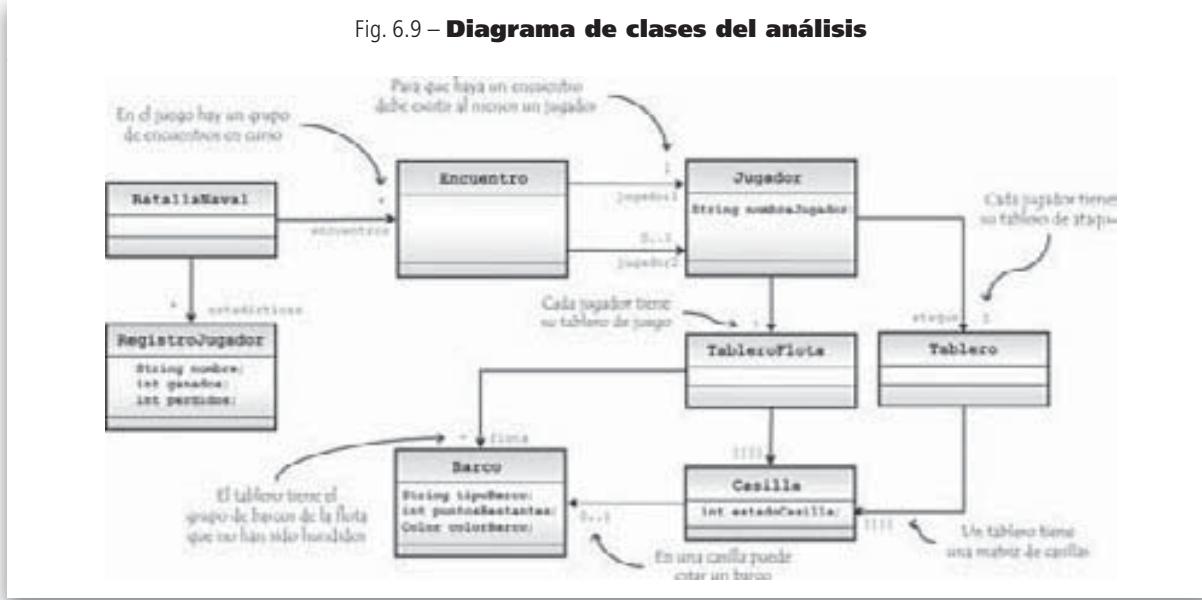
Requerimiento no funcional 1	Tipo: Persistencia Descripción:
Requerimiento no funcional 2	Tipo: Conurrencia Descripción:
Requerimiento no funcional 3	Tipo: Distribución Descripción:
Requerimiento no funcional 4	Tipo: Visualización e interacción Descripción:

3.3. Modelo del Mundo del Problema

En esta etapa debemos establecer el modelo del mundo del problema, ignorando todo lo que tenga que ver con los requerimientos no funcionales. Aquí, por ejemplo, no

va a aparecer que cierta información debe ser persistente o que el programa va a funcionar sobre varios computadores. En la figura 6.9 se puede apreciar el diagrama de clases del análisis, en el cual aparecen ocho conceptos con algunos de sus principales atributos y con las relaciones existentes entre ellos.

Fig. 6.9 – **Diagrama de clases del análisis**



En la siguiente tabla se hace un resumen de las clases y las asociaciones del diagrama.

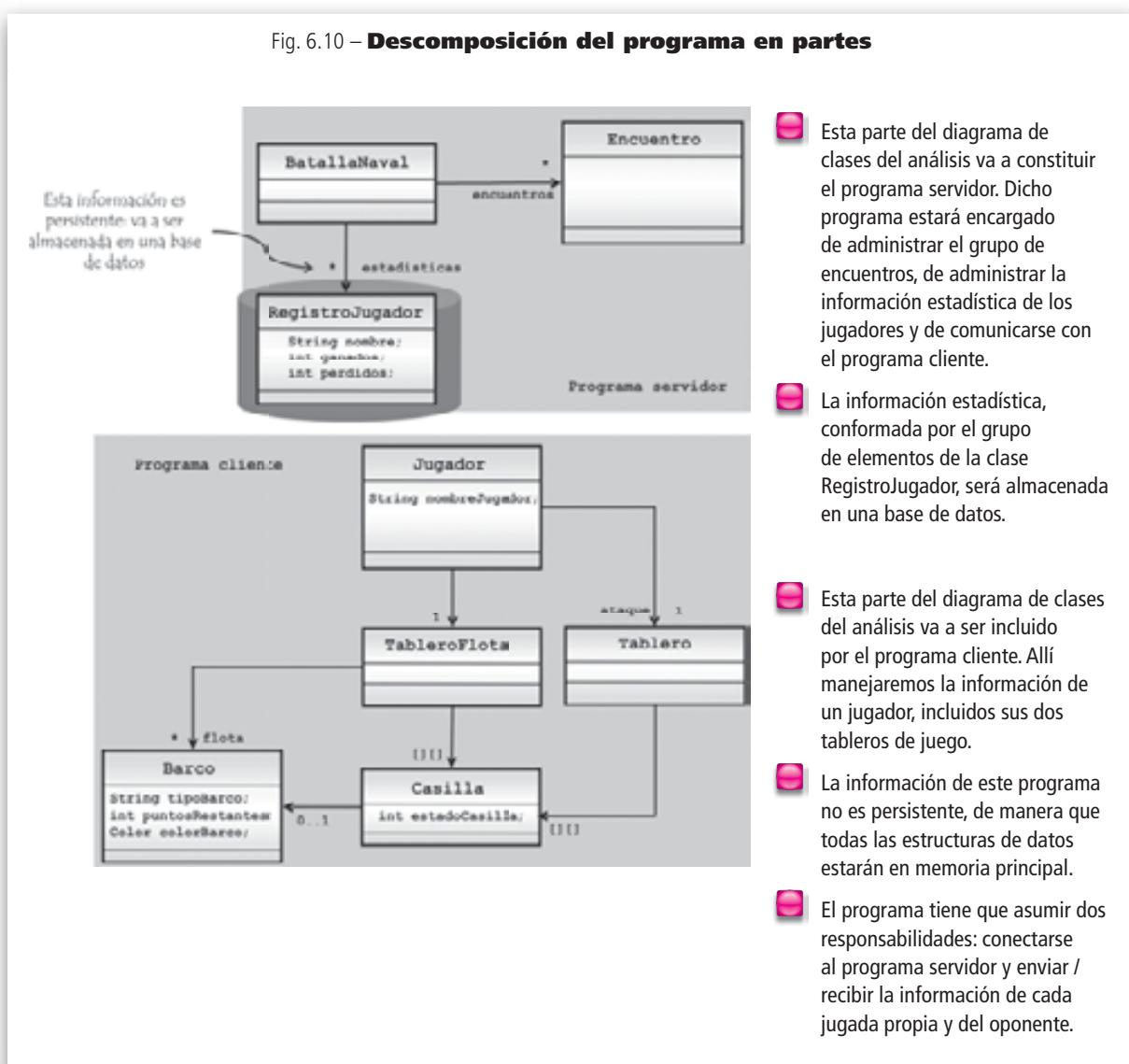
Clase	Descripción	Asociaciones
BatallaNaval	Esta clase representa el juego de Batalla Naval.	<ul style="list-style-type: none"> Tiene una relación hacia las estadísticas del juego (un grupo de objetos de la clase RegistroJugador). Tiene una referencia hacia el grupo de encuentros que se están desarrollando de manera simultánea entre distintos jugadores.
RegistroJugador	Esta clase maneja la información estadística de un jugador. Para esto almacena su nombre, el número de encuentros ganados y el número de encuentros perdidos.	
Encuentro	Esta clase representa un encuentro entre dos jugadores. Es posible que el encuentro no haya comenzado y que un jugador esté esperando la llegada de otro.	<ul style="list-style-type: none"> Tiene una asociación hacia el primer jugador. Tiene una asociación opcional hacia el segundo jugador.
Jugador	Esta clase representa a un jugador. Su único atributo es su nombre.	<ul style="list-style-type: none"> Tiene una asociación hacia el tablero de ataque. Tiene una asociación hacia el tablero en el que tiene localizada su flotilla de barcos.
Tablero	Esta clase representa el tablero de ataque del jugador. Debe almacenar los disparos que ha hecho al oponente.	<ul style="list-style-type: none"> Tiene una matriz de casillas, cada una de las cuales sabe si ya se hizo un disparo sobre ese punto.
TableroFlota	Esta clase representa el tablero en el cual el jugador tiene situada su flotilla de barcos. Debe almacenar los disparos que ha recibido del oponente.	<ul style="list-style-type: none"> Tiene una matriz de casillas. Tiene una asociación hacia el grupo de barcos de la flotilla que no han sido hundidos.
Casilla	Representa una casilla en un tablero. Su estado puede ser: VACÍA (no hay un barco), OCUPADA (hay un barco, pero no le han disparado), ATACADA (no hay un barco y ya le dispararon) o IMPACTADA (hay un barco y ya le dispararon).	<ul style="list-style-type: none"> Tiene una asociación hacia el barco que se encuentra en la casilla o nada si la casilla está vacía (o si hace parte del tablero de ataque).
Barco	Representa un barco de la flotilla. Almacena como información el tipo de barco, el número de puntos restantes (que equivale al tamaño del barco menos los impactos que ha recibido) y el color del barco.	

3.4. Primer Borrador de Arquitectura

La primera decisión que debemos tomar al iniciar el diseño es la manera en la que vamos a partir el programa, teniendo en cuenta que un requerimiento

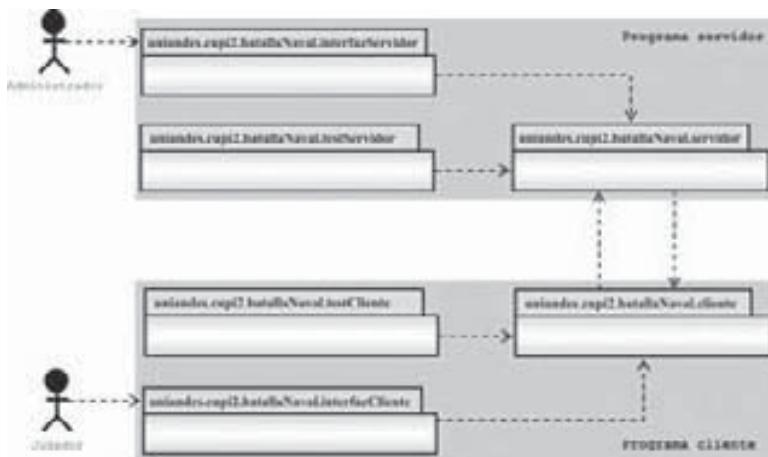
no funcional habla de distribución. En la figura 6.10 mostramos la manera de hacerlo y la asignación de responsabilidades que surge de dicha descomposición. También mostramos la forma como vamos a manejar la persistencia de la información estadística del juego.

Fig. 6.10 – Descomposición del programa en partes



Esto nos lleva a dividir nuestro programa en seis paquetes, tal como se muestra en la figura 6.11, para separar las pruebas automáticas y la interfaz de usuario de cada una de las partes. Allí se puede

apreciar claramente la relación que existe entre los actores y los paquetes incluidos en los programas, lo mismo que la relación de comunicación que hay entre el programa servidor y el programa cliente.

Fig. 6.11 – **Primer borrador de la arquitectura**

En las siguientes secciones comenzaremos a resolver los problemas que se presentan debido a los requerimientos no funcionales definidos en nuestro caso de estudio e iremos refinando poco a poco el diseño hasta construir los dos programas que hacen parte de la solución.

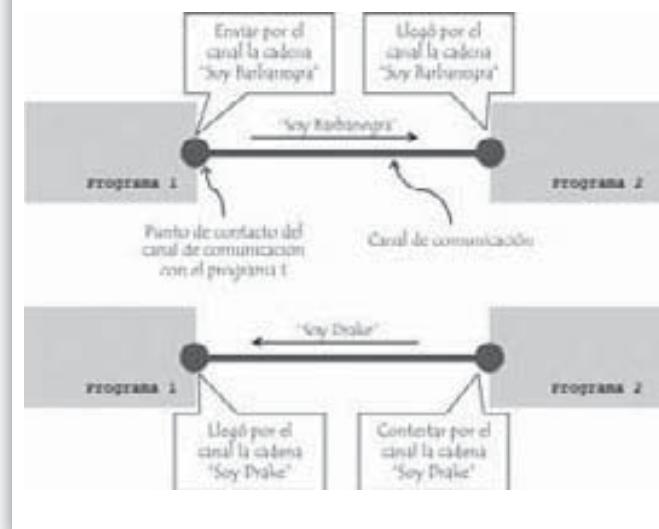
El tema de las pruebas automáticas para los requerimientos no funcionales aquí planteados está fuera del alcance de este libro. Si el lector está interesado en estudiar la manera como se implementan, le recomendamos revisar la solución contenida en el CD que acompaña al libro o en el sitio web.

3.5. Canales de Comunicación entre Programas

En esta sección nos vamos a concentrar en la manera de llevar información de un programa a otro, teniendo en cuenta que éstos pueden estar siendo ejecutados en computadores distintos. Para hacerlo vamos a utilizar lo que se denomina un canal de comunicación (*socket*), el cual representa el medio más simple de transporte de datos por una red. En la figura 6.12 ilustramos la idea de un canal de comunicación entre dos programas. Cada uno de ellos maneja uno de los extremos del canal y por allí puede enviar y recibir información. En nuestro caso, enviaremos cadenas de caracteres por

el canal, con la información de las jugadas, el resultado de cada disparo, etc.

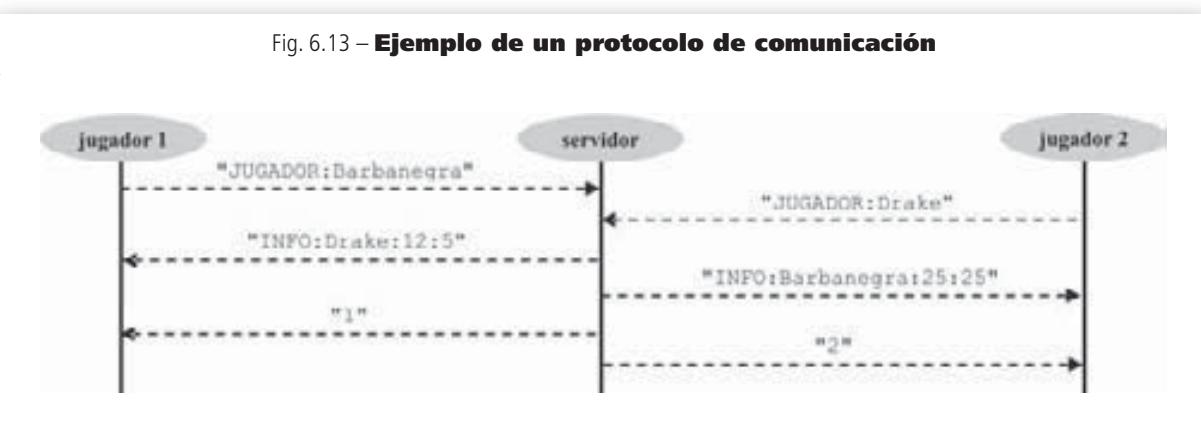
El problema de la comunicación se reduce entonces a tres aspectos: cómo crear un canal, cómo escribir en él y cómo leer la información proveniente de un canal. El resto corresponde a la manipulación y procesamiento de la información recibida, que es un problema ya estudiado en niveles anteriores.

Fig. 6.12 – **Canal de comunicación entre dos programas**

Un canal de comunicación sólo puede ser usado en un sentido en un momento dado de la ejecución. Esto implica que los programas deben estar previamente de acuerdo en quién lee, quién escribe y qué debe contener el mensaje transmitido. Esto es lo que se denomina un **protocolo**. El programa que está esperando algo por el canal bloquea su ejecución hasta que no llega el mensaje que el otro programa le debe enviar. En ese momento lo usual es que cambien de papel y que el programa que acaba de enviar el mensaje se bloquee a la espera de una respuesta. Un protocolo se expresa usualmente siguiendo la sintaxis

mostrada en la figura 6.13. Allí aparece una parte del protocolo que utilizaremos entre el programa servidor y el programa cliente del caso de estudio: (1) el jugador 1 se presenta, (2) el jugador 2 se presenta, (3) el servidor le informa al jugador 1 contra quién va a jugar, incluyendo el número de encuentros que este jugador ha tenido antes y el número de victorias; (4) el servidor le envía al jugador 2 la información de su contrincante, (5) el servidor le informa al jugador 1 que debe jugar de primero, (6) el servidor le informa al jugador 2 que va a jugar de segundo. El protocolo completo lo veremos más adelante.

Fig. 6.13 – **Ejemplo de un protocolo de comunicación**



En las secciones que siguen presentaremos las clases del lenguaje Java que implementan los canales de comunicación y el protocolo que utilizaremos como parte del desarrollo del juego de Batalla Naval.

Cada computador, por su parte, tiene un cierto número de puntos de conexión, denominados **puertos**, a los cuales se puede conectar un canal de comunicación. Los puertos están numerados entre 0 y 64K.

3.5.1. El Proceso de Conexión

En una red de computadores, cada uno de ellos tiene un nombre y una dirección únicos, lo cual permite referenciarlo desde otros puntos. Por ejemplo, el servidor del proyecto CUPi2 se llama cupi2.uniandes.edu.co y se encuentra en la dirección 157.253.201.12.



Para determinar la dirección de su computador, abra una ventana de comandos de Windows y utilice el comando ipconfig.



Para consultar los puertos de su computador que están siendo utilizados, abra una ventana de comandos de Windows y utilice el comando netstat -n. Obtendrá algo del siguiente estilo, en donde, para cada canal de comunicación abierto, se informa el puerto al cual está conectado, la dirección del computador con el cual está establecida la conexión y el puerto al cual está llegando el canal en la otra máquina.

Active Connections			
Proto	Local Address	Foreign Address	State
TCP	127.0.0.1:1197	127.0.0.1:1198	ESTABLISHED
TCP	127.0.0.1:1198	127.0.0.1:1197	ESTABLISHED
TCP	192.168.0.1:46:1039	216.155.193.166:5050	ESTABLISHED
TCP	192.168.0.1:46:1050	68.142.233.174:5061	ESTABLISHED

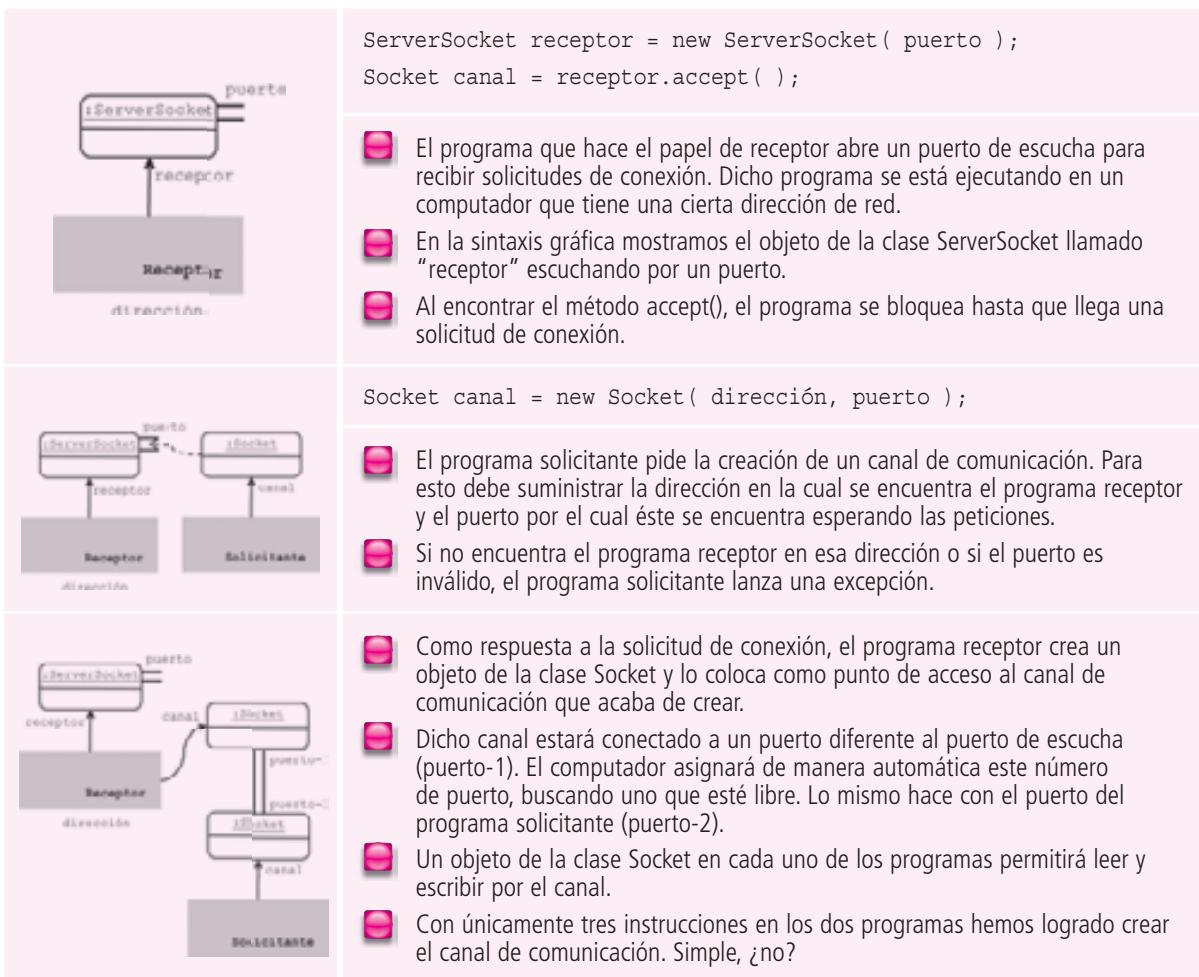


Algunos puertos del computador están reservados por el sistema operacional o tienen ya un uso predefinido. En el puerto 80, por ejemplo, se establecen las comunicaciones del protocolo HTTP.

En el puerto 25, se maneja el correo electrónico. Para evitar conflictos, es conveniente utilizar los puertos por encima del 1.000.

En el proceso de conexión entre dos programas hay dos participantes: el primero (llamado el **receptor**), que está esperando que alguien lo contacte para crear un canal de comunicación, y el segundo (llamado el **solicitante**), que contacta al primero con el fin de establecer una conexión.

En Java hay dos clases distintas para implementar el receptor y el solicitante. La primera clase se denomina `ServerSocket` y su función es esperar solicitudes de comunicación. La segunda clase se llama `Socket` y representa el canal de comunicación en sí mismo. El proceso de conexión sigue las etapas que se ilustran en la siguiente tabla, en la cual se utiliza una sintaxis gráfica y se asocian con ella las instrucciones necesarias en Java. Más adelante explicaremos en detalle los métodos disponibles en las clases `ServerSocket` y `Socket`.



La clase `ServerSocket` se encuentra en el paquete `java.net`, y cuenta con los siguientes métodos:

- `ServerSocket(puerto)`: Crea un objeto de la clase y abre un puerto de espera de solicitudes de conexión. Si se presenta un problema en el proceso, el método lanza la excepción `IOException`. El parámetro `puerto` es un valor entero que representa el número del puerto lógico del computador por el cual debe escuchar las solicitudes.
- `accept()`: Este método retorna un objeto de la clase `Socket`, como respuesta a una solicitud de conexión recibida. Si se presenta un problema en el proceso, el método lanza la excepción `IOException`. Este método bloquea la ejecución del programa hasta que llega una solicitud.
- `close()`: Con este método se cierra el puerto de espera de solicitudes de conexión. Si se presenta un problema en el proceso, el método lanza la excepción `IOException`.
- `getLocalPort()`: Este método retorna el puerto por el cual el objeto de la clase `ServerSocket` se encuentra escuchando.
- `close():` Con este método se cierra el canal de comunicación. Si se presenta un problema en el proceso, el método lanza la excepción `IOException`.
- `getPort():` Este método retorna un valor entero con el número del puerto al cual está asociado el canal de comunicación.
- `getLocalPort():` Este método retorna un valor entero con el número del puerto al cual está asociado el canal de comunicación en el otro computador.

La clase `Socket` también se encuentra en el paquete `java.net`, y algunos de sus principales métodos (de más de 40 que tiene) son los siguientes:

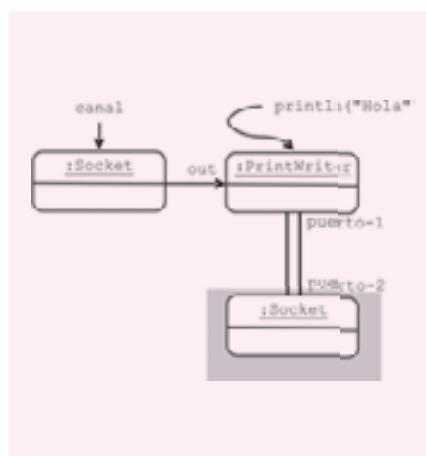
- `Socket(dirección, puerto)`: Permite crear un canal de comunicación hacia la máquina

que se encuentra en la dirección que llega como parámetro, solicitando la conexión por el puerto que se pasa también como parámetro. Puede lanzar la excepción `UnknownHostException` si no puede localizar al otro computador, la excepción `IOException` si ocurre un error en las comunicaciones mientras trata de crear el canal o la excepción `SecurityException` si el programa no tiene autorización para crear un canal. La dirección `localhost` se interpreta como la máquina en la cual se está ejecutando el programa.

- `close():` Con este método se cierra el canal de comunicación. Si se presenta un problema en el proceso, el método lanza la excepción `IOException`.
- `getPort():` Este método retorna un valor entero con el número del puerto al cual está asociado el canal de comunicación en el otro computador.

3.5.2. El Proceso de Escritura

Una vez establecido el canal de comunicación, veamos ahora la manera de escribir en él, lo cual es igual para cualquiera de los dos extremos del canal (ya no hay diferencia entre el receptor y el solicitante). Para esto vamos a utilizar un flujo de escritura idéntico al que utilizábamos para el caso de los archivos.

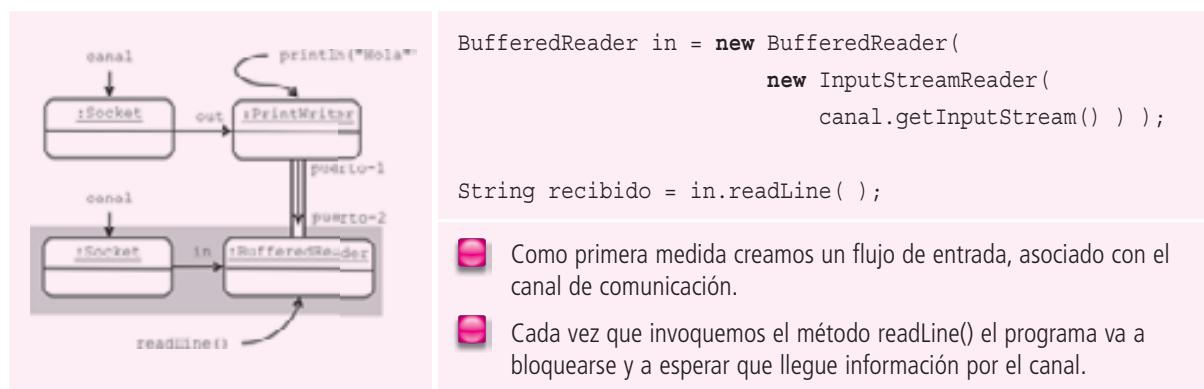


```
PrintWriter out = new PrintWriter(
    canal.getOutputStream( ), true );
out.println( "Hola" );
```

- Para crear el flujo de escritura, creamos un objeto de la clase `PrintWriter` y le pasamos como parámetro información sobre el canal de comunicación (suponemos que "canal" es una referencia al socket).
- Luego, usamos los métodos de escritura para flujos de salida que ya estudiamos en niveles anteriores ("println").
- Siempre que queramos enviar información por el canal, usamos el flujo de escritura ("out") que acabamos de crear.

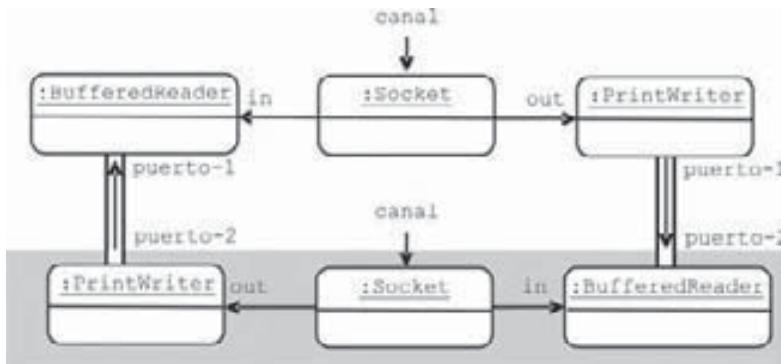
3.5.3. El Proceso de Lectura

Para leer la información que llega por un canal de comunicación, vamos a utilizar un flujo de entrada y a utilizar los mismos métodos que presentamos en el nivel que estudiamos el tema de archivos.



Si queremos leer y escribir de cada lado del canal, cada uno de los programas debe crear un flujo de lectura y un flujo de escritura, para lograr el estado mostrado en la figura 6.14.

Fig. 6.14 – **Estructura de comunicación en ambos sentidos en un canal**



En este punto ya sabemos crear un canal de comunicación entre dos programas, al igual que escribir y leer del mismo. Además, hemos visto que, desde el punto de vista de código, estos procesos son muy simples de implementar.

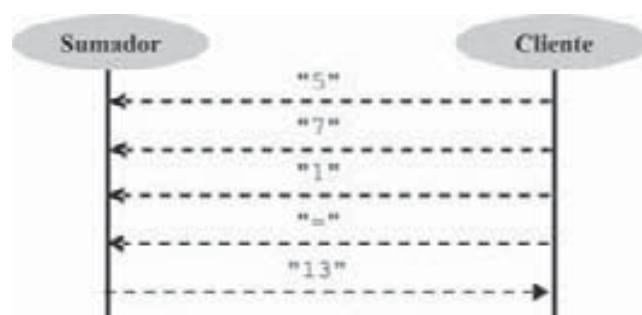


Para cerrar un canal de comunicación debemos utilizar el método `close()`.

Tarea 2

Objetivo: Crear dos programas simples que se comuniquen por un canal.

Desarrolle las dos clases que se plantean a continuación. La clase **Sumador** debe abrir un **ServerSocket** en el puerto 4444 para que el programa cliente que se ejecuta sobre el mismo computador (clase **Cliente**) se conecte. Luego, debe esperar a que le envíe cualquier cantidad de valores numéricos, uno por línea, tal como se plantea más adelante en el protocolo. Cuando el programa cliente envíe por el canal la cadena “=”, el programa servidor calcula la suma de todos los números que ha recibido, contesta dicho valor y termina su ejecución.



```

public class Sumador
{
    private ServerSocket server;
    private Socket canal;

    public Sumador( )
    {

    }

    public void atenderCliente( )
    {
        try
        {
            server = new ServerSocket(4444);
            while (true)
            {
                canal = server.accept();
                BufferedReader entrada = new BufferedReader(new InputStreamReader(canal.getInputStream()));
                String linea;
                int suma = 0;
                while ((linea = entrada.readLine()) != null)
                {
                    if (linea.equals("="))
                    {
                        break;
                    }
                    else
                    {
                        suma += Integer.parseInt(linea);
                    }
                }
                PrintWriter salida = new PrintWriter(canal.getOutputStream());
                salida.println(suma);
                salida.flush();
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

public class Cliente
{
    private Socket canal;

    public void conectar( )
    {
        try
        {
            canal = new Socket("localhost", 4444);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }

    public int pedirSuma( int valores[] )
    {
        BufferedReader entrada = new BufferedReader(new InputStreamReader(canal.getInputStream()));
        PrintWriter salida = new PrintWriter(canal.getOutputStream());
        String linea;
        for (int i = 0; i < valores.length; i++)
        {
            salida.println(valores[i]);
        }
        salida.println("=");
        salida.flush();
        linea = entrada.readLine();
        return Integer.parseInt(linea);
    }
}
  
```

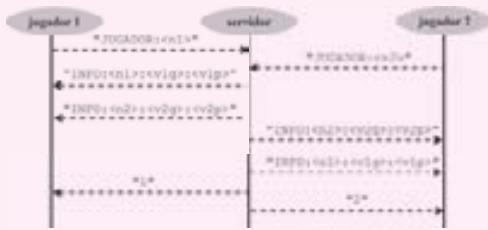
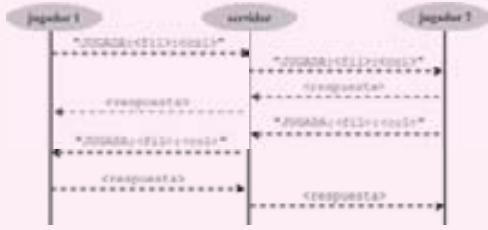
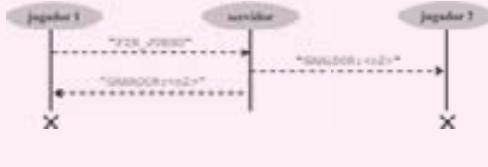
3.5.4. Los Protocolos de Comunicación

Definir los protocolos de comunicación resulta fundamental en el momento de diseñar la manera de interactuar de dos programas. En un protocolo debe hacerse explícito:

- Cómo comienza la interacción: quién lee y quién escribe en la primera interacción.

- Qué estructura tienen los mensajes que se envían.
- Cómo se interpreta cada uno de los mensajes.
- Cómo se manejan las situaciones de error.
- Cómo termina el protocolo.

En nuestro caso de estudio hay tres protocolos principales: uno de presentación (del cual ya vimos una parte anteriormente), un protocolo de juego y un protocolo de terminación. Veámoslos en detalle en la siguiente tabla:

 <pre> sequenceDiagram participant J1 as Jugador 1 participant J2 as Jugador 2 participant S as servidor J1->>S: "INICIAR JUEGO<1>" S->>J1: "JUGADOR<1>=<n1>" J1->>S: "INICIAR JUEGO<2>" S->>J2: "JUGADOR<2>=<n2>" </pre>	<ul style="list-style-type: none"> ■ <n1> es el nombre del primer jugador. ■ <n2> es el nombre del segundo jugador. ■ <vig> es el número de partidos ganados del jugador “i”. ■ <vip> es el número de partidos perdidos del jugador “i”. ■ Cuando el jugador recibe un “1” sabe que debe comenzar el encuentro. ■ Cuando el jugador recibe un “2” sabe que debe esperar el disparo del adversario.
 <pre> sequenceDiagram participant J1 as Jugador 1 participant J2 as Jugador 2 participant S as servidor J1->>S: "DISPARO<1>=<fil1><col1>" S->>J2: "DISPARO<2>=<fil2><col2>" J2->>S: "RESPUESTA<1>" S->>J1: "RESPUESTA<2>" </pre>	<ul style="list-style-type: none"> ■ <fil> y <col> son la fila y la columna de un disparo. ■ Durante el juego, el servidor sólo hace la intermediación de las comunicaciones. ■ <respuesta> puede ser: <ul style="list-style-type: none"> • “AGUA” • “IMPACTO:<tipo>:<hundido>” • “FIN_JUEGO” ■ Si es un impacto, en el mensaje va el tipo de barco que fue alcanzado y la palabra “true” o “false” para indicar si fue hundida la nave (por ejemplo, “IMPACTO:fragata:false”). ■ Si la respuesta es “FIN_JUEGO” se inicia el tercer protocolo.
 <pre> sequenceDiagram participant J1 as Jugador 1 participant J2 as Jugador 2 participant S as servidor J1->>S: "FIN_JUEGO<1>" S->>J1: "GANADOR<1>=<n1>" S->>J2: "GANADOR<1>=<n2>" </pre>	<ul style="list-style-type: none"> ■ Este protocolo comienza cuando uno de los jugadores (en este caso el primero) indica que con el disparo que acaba de recibir el juego ha terminado (y que el otro jugador fue el vencedor). ■ El servidor en ese caso informa a los dos participantes el nombre del ganador. Luego cierra los canales de comunicación que tenía con ambos.



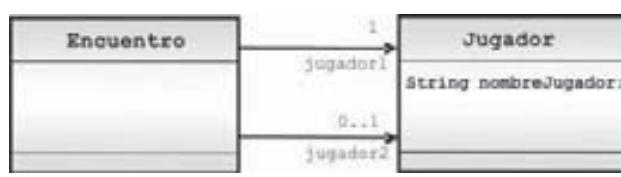
En el CD que acompaña al libro puede encontrar algunos entrenadores de manejo de *sockets*. En particular es conveniente mirar la herramienta que permite observar el funcionamiento del protocolo HTTP. Este protocolo permite a un navegador de Internet consultar un documento con formato HTML.

3.6. Primer Refinamiento de la Arquitectura

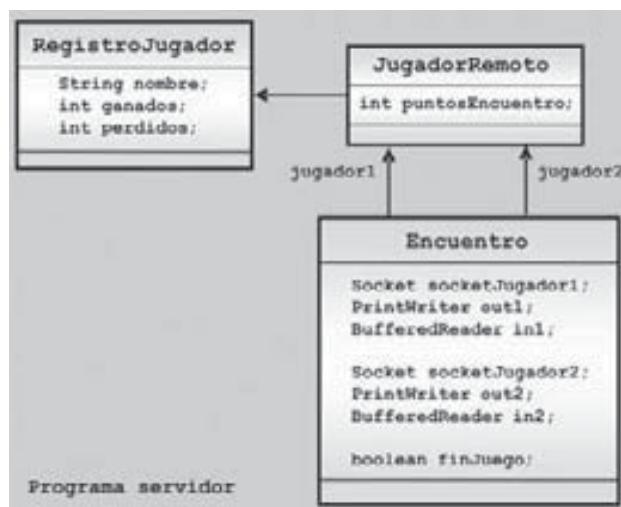
Después de haber estudiado la manera de comunicar dos programas, estamos listos para hacer el primer refinamiento de la arquitectura y ver las declaraciones de

las clases involucradas en la conexión entre el programa servidor y el programa cliente. En la figura 6.15 aparece la transformación de una parte del diagrama de clases del análisis para incluir los elementos necesarios para la comunicación.

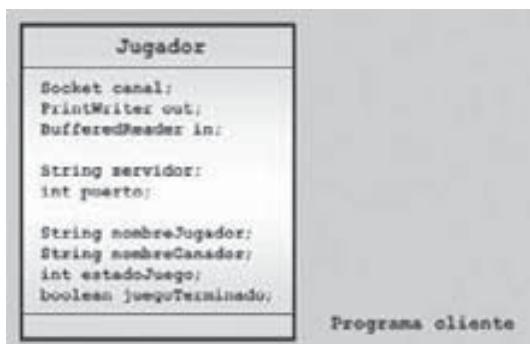
Fig. 6.15 – Primera transformación del diagrama de clases del análisis



- Ésta es una parte del diagrama de clases del análisis que vamos a considerar.
- Puesto que cada encuentro se debe encargar de manejar las comunicaciones con sus jugadores, nos concentraremos únicamente en este punto del diagrama.



- A la clase Encuentro le definimos como atributo los objetos que permiten materializar los canales de comunicación hacia los dos programas cliente.
- Los atributos "socketJugador1", "out1" e "in1" nos van a permitir hablar con el primer jugador.
- Los atributos "socketJugador2", "out2" e "in2" nos van a permitir hablar con el segundo jugador.
- Incluimos además un atributo que nos indica si el juego ya terminó.
- Creamos una clase llamada JugadorRemoto, que tiene la información de cada uno de los jugadores involucrados en el encuentro.
- La clase RegistroJugador ya había aparecido en la etapa de análisis. Creamos una asociación desde la clase JugadorRemoto hacia esta clase, para mantener la información de cada participante.



- Los primeros cinco atributos de la clase van a permitir la comunicación con el servidor: canal, flujo de salida, flujo de entrada, nombre del servidor y puerto en el cual el servidor se encuentra escuchando las conexiones.
- Luego vienen el nombre del jugador, el nombre del ganador (si ya hay alguno), el estado del juego (SIN_CONECTAR, ESPERANDO_LOCAL, ESPERANDO_OPONENTE o ESPERANDO_RESPUESTA) y un indicador de si el juego ya ha terminado.

Las declaraciones de las principales clases, así como sus invariantes, se muestran en el ejemplo 1.



Ejemplo 1

Objetivo: Mostrar las declaraciones de algunas de las clases del caso de estudio involucradas en la comunicación de información entre el programa servidor y el programa cliente.

En este ejemplo se muestran las declaraciones de las clases encargadas de la comunicación, así como los invariantes de las mismas.

```
public class Encuentro
{
    // -----
    // Constantes
    // -----

    public static final String JUGADOR = "JUGADOR";
    public static final String INFO_JUGADOR = "INFO";
    public static final String PRIMER_TURNO = "1";
    public static final String SEGUNDO_TURNO = "2";
    public static final String JUGADA = "JUGADA";

    public static final String AGUA = "AGUA";
    public static final String IMPACTO = "IMPACTO";
    public static final String FIN_JUEGO = "FIN JUEGO";
    public static final String GANADOR = "GANADOR";

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

    private Socket socketJugador1;
    private PrintWriter out1;
    private BufferedReader in1;

    private Socket socketJugador2;
    private PrintWriter out2;
    private BufferedReader in2;

    private JugadorRemoto jugador1;
    private JugadorRemoto jugador2;

    private boolean finJuego;
}
```

```
public class Jugador
{
    // -----
    // Constantes
    // -----

    public static final int SIN_CONECTAR = 0;
    public static final int ESPERANDO_LOCAL = 1;
    public static final int ESPERANDO_OPONENTE = 2;
    public static final int ESPERANDO_RESPUESTA = 3;
```

■ Definimos como constantes todas las cadenas que tienen que ver con el protocolo de comunicación. De esta forma, si el protocolo evoluciona, podemos manejarlo más fácilmente.

■ Un objeto de la clase Encuentro sólo se crea cuando ya hay dos participantes listos para enfrentarse. Por esta razón el invariante de la clase incluye las siguientes afirmaciones:

- !finJuego => socketJugador1 != null
- !finJuego => out1 != null
- !finJuego => in1 != null
- !finJuego => socketJugador2 != null
- !finJuego => out2 != null
- !finJuego => in2 != null
- jugador1 != null
- jugador2 != null

■ El símbolo " \Rightarrow " representa el operador "implica".

■ Cuando la variable "finJuego" es verdadera, se deben cerrar los canales de comunicación hacia los jugadores y registrar el resultado del partido en la base de datos.

■ En la clase Jugador se declaran cuatro constantes para representar los estados posibles en los que se puede encontrar el juego en un momento dado de la ejecución:

- SIN_CONECTAR: indica que el jugador no se ha conectado al servidor.
- ESPERANDO_LOCAL: indica que el programa se encuentra esperando a que el jugador decida su jugada.
- ESPERANDO_OPONENTE: indica que el programa está esperando que por el canal llegue la jugada del oponente.
- ESPERANDO_RESPUESTA: indica que el programa está esperando que por el canal llegue el resultado del último disparo que se le hizo al oponente.

■ Estas constantes reflejan los posibles estados por los que pasa el protocolo.

```

// -----
// Atributos
// -----
private Socket canal;
private PrintWriter out;
private BufferedReader in;

private String servidor;
private int puerto;

private String nombreJugador;
private String nombreGanador;

private int estadoJuego;
private boolean juegoTerminado;

private TableroFlota tableroFlota;
private Tablero tableroAtaque;
}

```

- El valor del atributo "estadoJuego" pertenece al conjunto { SIN_CONECTAR, ESPERANDO_LOCAL, ESPERANDO_OPONENTE, ESPERANDO RESPUESTA }
- Si estadoJuego = SIN_CONECTAR, entonces juegoTerminado = true
- Si estadoJuego != SIN_CONECTAR, entonces:
 - canal != null
 - out != null && in != null
 - tableroFlota != null
 - tableroAtaque != null
 - servidor != null && puerto > 0
 - nombreJugador != null

Teniendo en cuenta las declaraciones anteriores, ya podemos implementar el esqueleto de los métodos que materializan el protocolo de comunicación diseñado. Eso lo mostramos en el ejemplo 2 y la tarea 3.

Ejemplo 2



Objetivo: Mostrar el esqueleto de los métodos que implementan el protocolo de comunicación.

En este ejemplo mostramos los principales métodos de la clase `Encuentro` que se encargan de la comunicación.

```

public class Encuentro
{
    public Encuentro( Socket canal1, Socket canal2 ) throws IOException
    {
        out1 = new PrintWriter( canal1.getOutputStream( ), true );
        in1 = new BufferedReader(
            new InputStreamReader( canal1.getInputStream( ) ) );
        socketJugador1 = canal1;

        out2 = new PrintWriter( canal2.getOutputStream( ), true );
        in2 = new BufferedReader(
            new InputStreamReader( canal2.getInputStream( ) ) );
        socketJugador2 = canal2;

        finJuego = false;
        verificarInvariantes();
    }
}

```

■ El método constructor de la clase `Encuentro` recibe como parámetro dos canales de comunicación (uno con cada jugador) y se encarga de abrir los respectivos flujos de lectura y escritura para cada uno de ellos.

■ Inicializa luego el atributo `finJuego` en falso y verifica por último el invariante de la clase.

- ```

public void run()
{
 iniciarEncuentro();

 int atacante = 1;

 while(!finJuego)
 {
 procesarJugada(atacante);

 if(finJuego)
 terminarEncuentro();
 else
 atacante = (atacante == 1) ? 2 : 1;
 }
}

```
- Este es el método que controla todo el protocolo de comunicación.
- Por simplicidad ignoramos todas las excepciones que puede lanzar.
- Nos basamos en tres métodos: iniciarEncuentro(), procesarJugada() y terminarEncuentro().
- Después de cada jugada cambia la variable que indica el número del jugador que tiene el turno.
- 
- ```

private void iniciarEncuentro( )
{
    String info1 = in1.readLine( );

    RegistroJugador reg1 = consultarJugador( info1 );
    jugador1 = new JugadorRemoto( reg1 );

    String info2 = in2.readLine( );

    RegistroJugador reg2 = consultarJugador( info2 );
    jugador2 = new JugadorRemoto( reg2 );

    enviarInformacion( out1, jugador1.darRegistroJugador( ) );
    enviarInformacion( out1, jugador2.darRegistroJugador( ) );

    enviarInformacion( out2, jugador2.darRegistroJugador( ) );
    enviarInformacion( out2, jugador1.darRegistroJugador( ) );

    out1.println( PRIMER_TURNO );
    out2.println( SEGUNDO_TURNO );
}

```
- Este método es el encargado de iniciar un encuentro. Para esto lee por cada canal el nombre del jugador que se conecta y luego envía a los dos jugadores la información estadística de sus partidos anteriores (según establece el protocolo).
- El método enviarInformacion() se encarga de enviar por un flujo de escritura la información de uno de los jugadores (dependiendo de los parámetros)
- Algunos métodos aquí utilizados, de acceso a la información estadística de un jugador, serán estudiados más adelante. Por ahora, los nombres son suficientemente significativos como para poder imaginar su comportamiento.

```

private void procesarJugada( int atacante )
{
    PrintWriter atacanteOut = ( atacante == 1 ) ? out1 : out2;

    PrintWriter atacadoOut = ( atacante == 1 ) ? out2 : out1;

    BufferedReader atacanteIn = ( atacante == 1 ) ? in1 : in2;

    BufferedReader atacadoIn = ( atacante == 1 ) ? in2 : in1;

    String lineaAtaque = atacanteIn.readLine();

    atacadoOut.println( lineaAtaque );

    String lineaResultado = atacadoIn.readLine();

    if( lineaResultado.startsWith( IMPACTO ) )
    {
        // Aumenta los puntos del jugador atacante
    }
    else if( lineaResultado.startsWith( FIN_JUEGO ) )
    {
        // Aumenta los puntos del jugador atacante

        finJuego = true;
    }

    atacanteOut.println( lineaResultado );
}
}

```

Este método recibe como parámetro el número del jugador que acaba de hacer el disparo.

Con dicho valor seleccionamos los respectivos flujos de lectura y escritura en variables temporales. Así sabemos por cuál flujo debemos esperar la información de la jugada y por cuál flujo debemos informar el ataque.

Luego, por el flujo de lectura seleccionado (atacanteln), esperamos que llegue el disparo del jugador que tiene el turno. En ese punto, el programa se bloquea y no continúa hasta que llega por el canal dicha información.

Por el flujo de escritura seleccionado (atacadoOut) enviamos la jugada y esperamos una respuesta. Una vez que ésta llega analizamos su contenido, para decidir la acción que se debe tomar.

Finalmente enviamos al atacante el resultado de su jugada.

Si el juego terminó, cambiamos el valor del respectivo atributo, para que se inicie el protocolo de terminación de juego.

Tarea 3

Objetivo: Estudiar los métodos de la clase `Jugador` que se encargan de implementar el protocolo de comunicación.

Siga los pasos que se describen a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n12_batallaNaval.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
2. Localice y edite la clase `Jugador` que se encuentra en el paquete `uniandes.cupi2.batallaNaval.cliente`.
3. Estudie el código del constructor.
 - ¿Por qué la mayoría de atributos se inicializan en `null`?
 - ¿Qué dice el invariante al respecto?
4. Estudie el método que verifica el invariante de la clase. Mire la manera de verificar que las condiciones de corrección se cumplen cuando éstas dependen del valor de un atributo.
5. Estudie el método `conectar()`.
 - ¿En qué casos lanza una excepción?
 - ¿Qué valor se espera en el atributo llamado `puertoServ`?
6. Estudie el método `iniciarEncuentro()`.
 - ¿Qué método sitúa en el tablero los barcos del jugador?
 - ¿Qué es lo primero que hace el método como parte del protocolo de comunicación?
 - ¿Qué método de la clase `String` utiliza para partir la cadena de caracteres que llega después del servidor?
 - ¿Qué información viene en esta cadena?
 - ¿Qué hace el método si el servidor le indica que tiene el primer turno de juego?
 - ¿Para qué se utiliza la contenedora llamada `mensajesSinLeer`?
7. Estudie el método `esperarJugada()`.
 - ¿Qué dice la precondición del método?
 - ¿Qué está esperando que llegue por el canal?
 - ¿Cuántos casos posibles hay como consecuencia del disparo? ¿Qué contesta en cada caso?
 - ¿En qué valor termina el atributo `estadoJuego` y por qué razón?

8. Estudie el método `enviarJugada()`.

¿En qué punto del protocolo se encuentra el programa cuando se ejecuta este método?

¿Qué recibe como parámetro el método?

¿Cómo envía la jugada al oponente y cómo espera su respuesta?

¿En qué valor termina el atributo `estadoJuego` y por qué razón?

9. Estudie el método `terminarEncuentro()`.

¿En qué punto del protocolo se encuentra el programa cuando se ejecuta este método?

¿Qué método utiliza para cerrar el canal de comunicación y los flujos de lectura y escritura?

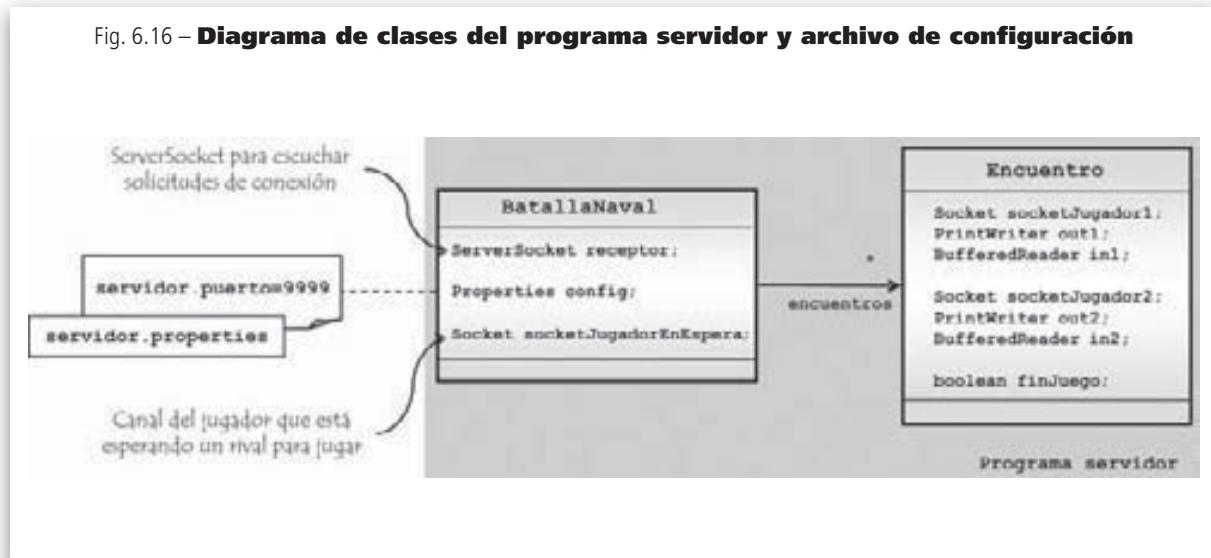
¿En qué valor termina el atributo `estadoJuego` y por qué razón?

¿Qué hace el método con el nombre del ganador del encuentro?

Sólo nos falta ahora la conexión inicial de los jugadores al servidor. Para esto debemos darle una mirada a la clase `BatallaNaval` en el programa servidor, la cual tiene la responsabilidad de manejar el grupo de encuentros simultáneos que se están desarrollando. Esta clase es la encargada de manejar el `ServerSocket`

que va a permitir a los jugadores obtener una conexión y, posteriormente, un rival para jugar un partido. La información sobre el número del puerto en el cual debe escuchar las solicitudes de conexión viene en un archivo de configuración, tal como se muestra en la figura 6.16.

Fig. 6.16 – **Diagrama de clases del programa servidor y archivo de configuración**



En el ejemplo 3 mostramos la declaración de la clase BatallaNaval y el método de la clase Jugador que inicia el proceso de conexión.

Ejemplo 3



Objetivo: Mostrar el proceso de conexión entre los jugadores y el programa servidor.

En este ejemplo presentamos la declaración de la clase BatallaNaval y los métodos que le permiten recibir las conexiones de los jugadores. También mostramos el método de la clase Jugador involucrado en este proceso.

```
public class BatallaNaval
{
    // -----
    // Atributos
    // -----


    private ServerSocket receptor;

    private Properties config;

    private Socket socketJugadorEnEspera;

    private Collection encuentros;

    public ServidorBatallaNaval( String archivo )
    {
        encuentros = new Vector( );
        FileInputStream fis = new FileInputStream( archivo );
        config = new Properties( );
        config.load( fis );
        fis.close( );
    }
}
```

■ Esta clase tiene el ServerSocket que le va a permitir a los jugadores solicitar una conexión. El número del puerto por el cual debe escuchar viene en un archivo de configuración cuyo nombre llega como parámetro al constructor.

■ Para manejar la configuración del programa servidor se utiliza un objeto de la clase Properties.

■ La clase tiene un atributo para mantener el canal de comunicación del último jugador que se conectó y para el cual todavía no tenemos contrincante.

■ El grupo de encuentros lo vamos a manejar en una contenedora que implementa la interfaz Collection. En el constructor podemos ver que vamos a utilizar un objeto de la clase Vector, la cual implementa dicho contrato funcional.

■ El constructor se encarga únicamente de crear una colección vacía de encuentros y de leer el archivo de configuración.

```

public void recibirConexiones( )
{
    String aux = config.getProperty( "servidor.puerto" );

    int puerto = Integer.parseInt( aux );

    receptor = new ServerSocket( puerto );

    while( true )
    {
        Socket socketNuevoCliente = socket.accept( );

        crearEncuentro( socketNuevoCliente );
    }
}

```

 El método `recibirConexiones()` es responsable de crear el `ServerSocket` en el puerto definido por el archivo de configuración. Usamos el método `getProperty()` para recuperar el valor de dicha propiedad y el método `parseInt()` para convertir la cadena de caracteres en un valor entero.

 Este método es llamado desde la interfaz de usuario cuando se quiere que el servidor comience a aceptar conexiones de jugadores.

 Dentro de un ciclo infinito (la única manera de detener el servidor es cerrando la ventana del programa) este método se queda a la espera de conexiones de parte de los jugadores. El método `accept()` bloquea la ejecución del programa hasta que se recibe una solicitud.

 Luego, pasa como parámetro al método `crearEncuentro()` el canal que se acaba de crear para comunicarse con el cliente.

```

private void crearEncuentro( Socket socketNuevoCliente )
{
    if( socketJugadorEnEspera == null )

        socketJugadorEnEspera = socketNuevoCliente;

    else

```

 Este método recibe el canal de un nuevo jugador y decide si ya puede crear un encuentro (si ya había otro jugador esperando) o si debe guardar este canal en

```

    {
        Socket jug1 = socketJugadorEnEspera;

        Socket jug2 = socketNuevoCliente;

        socketJugadorEnEspera = null;

        Encuentro nuevo = new Encuentro( jug1, jug2 );

        encuentros.add( nuevo );

        nuevo.run();
    }
}

```

```

public class Jugador
{
    public void conectar( String nom, String dirServ,
                          int puertoServ )
    {
        nombreJugador = nom;

        servidor = dirServ;

        puerto = puertoServ;

        canal = new Socket( dirServ, puertoServ );

        out = new PrintWriter( canal.getOutputStream( ), true );

        in = new BufferedReader(
            new InputStreamReader( canal.getInputStream( ) ) );

        iniciarEncuentro();
    }
}

```

el respectivo atributo (socketJugadorEnEspera) a la espera del segundo jugador.

Luego de creado el objeto que representa el nuevo encuentro, lo agrega a la contenedora e invoca el método run() que se presentó en el ejemplo 2, el cual se encarga de iniciar el protocolo de comunicación.

Más adelante replantearemos la manera de invocar el método run(), teniendo en cuenta el trabajo concurrente que debe soportar el servidor.

Este es el método de la clase Jugador que se encarga de abrir la conexión hacia el servidor y luego comenzar a ejecutar los protocolos definidos para el juego.

El método recibe como parámetro el nombre del jugador, la dirección del servidor y el puerto por el cual éste se encuentra escuchando peticiones de conexión. Estos tres valores los debe suministrar el usuario a través de la interfaz del programa.

El método iniciarEncuentro() fue estudiado como parte de la tarea 3.

Tarea 4

Objetivo: Ejecutar el programa que permite jugar Batalla Naval y estudiar algunos aspectos del código. Siga los pasos que se describen a continuación.

- Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n12_batallaNaval.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
- Localice el archivo de configuración del servidor, en el directorio “data”. Edítelo y seleccione el puerto en el cual quiere que el servidor escuche peticiones de conexión (por defecto está oyendo por el puerto 9999).
- Localice en el paquete `uniandes.cupi2.batallaNaval.interfazServidor` la clase `InterfazBatallaNaval` y ejecútela. Esto debe ejecutar el programa servidor. Debe aparecer una ventana con la información de los encuentros en curso y la información estadística de juegos anteriores. Oprima los botones de refresco, para asegurarse de que la información se encuentra actualizada.
- Abra una ventana de comandos de Windows y teclee la instrucción `netstat -a`. Debe aparecer el número del puerto en el cual está escuchando el servidor con la leyenda “LISTENING”, mostrando algo parecido a lo que aparece en la siguiente imagen:

Proto	Local Address	Foreign Address	State
TCP	quimera:9999	quimera:0	LISTENING
TCP	quimera.microsoft-ds	quimera:0	LISTENING
TCP	quimera:5051	quimera:0	LISTENING
TCP	quimera:5181	quimera:0	LISTENING
TCP	quimera:9292	quimera:0	LISTENING

- Localice en el paquete `uniandes.cupi2.batallaNaval.interfazCliente` la clase `InterfazJugador` y ejecútela. Esto debe ejecutar el programa cliente y hacer aparecer la ventana de juego. En el título de dicha ventana se puede ver el mensaje “Batalla Naval: desconectado...”. Todavía no aparecen los barcos ni ha comenzado un encuentro. Utilice la opción **Iniciar Juego** del menú **Inicio**. Defina allí su nombre como jugador y el puerto en el cual está escuchando el servidor.
- Ejecute de nuevo el comando `netstat -a` en la ventana de comandos de Windows y localice el puerto que le fue asignado al canal de comunicación. Debe aparecer algo como lo que se muestra a continuación. Allí se puede ver que el cliente quedó conectado en el puerto 1395.

Proto	Local Address	Foreign Address	State
TCP	quimera:9999	quimera:0	LISTENING
TCP	quimera.microsoft-ds	quimera:0	LISTENING
TCP	quimera:5051	quimera:0	LISTENING
TCP	quimera:5181	quimera:0	LISTENING
TCP	quimera:9292	quimera:0	LISTENING
TCP	quimera:1395	localhost:3939	ESTABLISHED
TCP	quimera:3939	localhost:1395	ESTABLISHED

- Ejecute otra vez el programa cliente, inicie un juego y determine el puerto que fue asignado a esta nueva conexión (usando el comando `netstat -a`). Ahora, en el título de cada ventana se puede ver quién tiene el turno (“jugando”) y quién va a ser atacado (“esperando jugada”). Los barcos ya fueron distribuidos de manera aleatoria y el juego puede comenzar. Oprima el botón **Refrescar** de la ventana del programa servidor, para poder ver los resultados parciales de este nuevo encuentro.

8. Si tiene acceso a otro computador conectado en red, cree en dicha máquina el proyecto a partir del archivo `n12_batallaNaval.zip`. Ejecute allí el programa cliente, identifique la dirección del servidor (usando el comando `ipconfig`) y comience un nuevo encuentro. Visualice en cada una de las máquinas los puertos asignados.
9. Utilice el archivo `build.bat` para crear el archivo `batallaNaval.jar`. Cree ahora un archivo de distribución (llámelo por ejemplo `batallaNaval.zip`) que contenga el archivo `run.bat` (que ejecuta el programa cliente) y el archivo `jar` que acaba de crear. No olvide respetar la estructura de archivos en los que se encuentran estos dos archivos. Pruebe en otro computador la ejecución del programa cliente, construido de esta forma. Así lo puede distribuir y permitir a otros jugar Batalla Naval, usando un servidor que alguien debe ejecutar en algún computador conectado a Internet.



Un canal es el medio que permite comunicar dos programas. Para hacerlo, cada uno de los participantes debe abrir un flujo de lectura (por el que le llega lo que el otro le escriba) y un flujo de escritura (por el que puede mandar información al otro programa). Un protocolo de comunicación define el orden y el contenido de los mensajes que los programas intercambian.

3.7. Concurrencia en un Programa

Hasta este punto hemos ignorado un problema, y es que, en el momento de crear un nuevo encuentro, el servidor no puede a la vez atender el protocolo de los jugadores y esperar nuevas conexiones. Lo que va a su-

ceder es que si llega un nuevo jugador, el servidor no le va a aceptar la conexión. Si dibujamos el flujo de control del programa, llegaríamos a lo que se muestra en la figura 6.17, en donde tratamos de mostrar el código que se ejecuta en el servidor, ignorando la estructura de los métodos y los detalles.

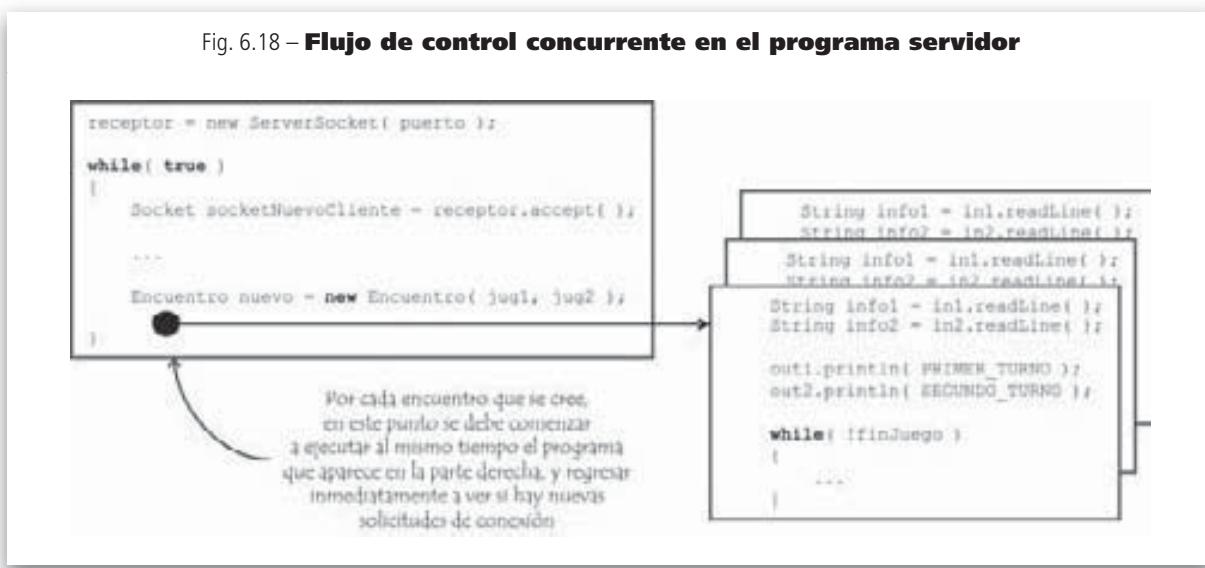
Fig. 6.17 – Flujo de control en el programa servidor



La única posibilidad que tenemos es decirle al programa servidor que trabaje en dos frentes simultáneos, como si fueran dos programas distintos, pero que puedan compartir el contexto de ejecución. Eso nos llevaría

a una estructura de ejecución como la que se sugiere en la figura 6.18, en donde cada encuentro está siendo atendido por una parte del programa servidor que se está ejecutando de manera concurrente.

Fig. 6.18 – Flujo de control concurrente en el programa servidor



Hacer esto es sorprendentemente sencillo en Java, puesto que el lenguaje fue diseñado para trabajar de esta manera. A cada una de las ejecuciones concurrentes la vamos a llamar un **hilo de ejecución** (*thread*). Hasta ahora nos hemos contentado con tener un solo hilo de ejecución en los programas, de manera que todas las instrucciones se ejecutan de manera secuencial, comenzando por el método `main()` (allí se crea el hilo que vamos a llamar principal).

Para hacer que la clase `Encuentro` se ejecute sobre un hilo distinto del principal, debemos seguir cinco pasos, los cuales se ilustran más adelante en el ejemplo 4. Éstos son:

- Hacer que la clase herede de la clase `Thread`.

- Implementar un método sin parámetros que se llame `run()`, que contenga las instrucciones que queremos que se ejecuten sobre el nuevo hilo.
- Declarar como atributos de la clase toda la información que el nuevo hilo va a necesitar para poder cumplir con su tarea. Esta información corresponde típicamente a atributos de la clase de la cual se va a desprender el nuevo hilo.
- Crear un constructor para la nueva clase, que reciba como parámetro la información necesaria para inicializar los atributos del punto anterior.
- Desde la clase que se va a desprender el nuevo hilo, crear una instancia de la nueva clase y llamar el método `start()`. Esto hace que se invoque el método `run()` de la nueva instancia.

Ejemplo 4



Objetivo: Modificar las clases `Encuentro` y `BatallaNaval`, para que cada nuevo encuentro se ejecute sobre un hilo distinto.

En este ejemplo mostramos la manera de ajustar las clases del programa servidor para que se pueda soportar el requerimiento no funcional de concurrencia.

```

public class Encuentro extends Thread
{
    // ...
    // Atributos
    // ...
    private Socket socketJugador1;
    private Socket socketJugador2;
}

```

Paso 1: Hacemos que la clase `Encuentro` herede de la clase `Thread`.

Paso 2: Implementamos el método `run()`. En la versión inicial ya le

```

public Encuentro( Socket canal1, Socket canal2 )
{
    socketJugador1 = canal1;
    socketJugador2 = canal2;
    ...
}

public void run()
{
    ...
    while( !finJuego )
    {
        ...
    }
}
}

```

habíamos puesto este nombre al método que se encargaba de controlar el protocolo de comunicación.

■ Paso 3: Declaramos los atributos de la clase para almacenar la información que necesitamos para la tarea. En este caso, con las referencias a los dos canales de comunicación involucrados, podemos resolver la tarea planteada. Por eso definimos dos atributos en la clase (socketJugador1, socketJugador2).

■ Paso 4: Creamos un constructor que reciba como parámetro la información necesaria para inicializar los atributos antes definidos.

```

public class BatallaNaval
{
    private void crearEncuentro( Socket socketNuevoCliente )
    {
        ...
        Encuentro nuevo = new Encuentro( jug1, jug2 );
        nuevo.start();
    }
}

```

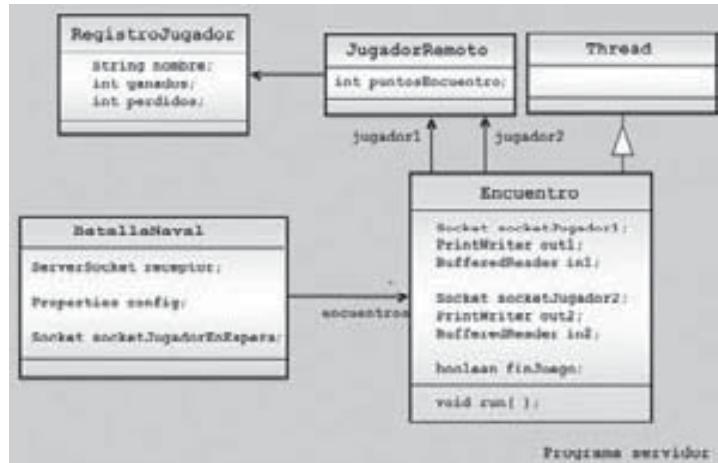
■ El último paso es modificar el programa desde el que vamos a desprender el nuevo hilo.

■ Para esto, después de crear una instancia de la clase Encuentro, invocamos simplemente el método start().

■ Esto es suficiente para que se invoque el método run() de dicha clase y que se inicie en un nuevo hilo de ejecución.

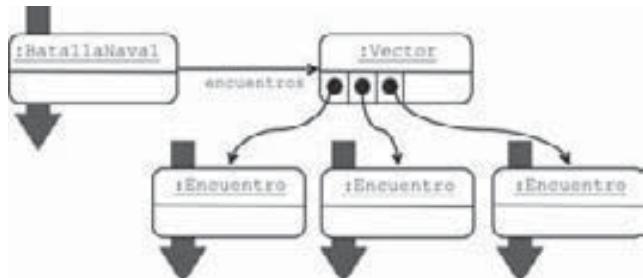
Esta modificación nos lleva a ajustar el diagrama de clases del programa servidor, obteniendo el diagrama que se presenta en la figura 6.19.

Fig. 6.19 – Nuevo diagrama de clases del programa servidor



Durante la ejecución, podemos tener el diagrama de objetos que aparece ilustrado en la figura 6.20. Allí vemos tres encuentros simultáneos soportados por el servidor. Cada uno de ellos se desarrolla sobre un hilo distinto de ejecución, diferente del hilo en el cual se ejecuta el objeto de la clase `BatallaNaval`, encargado de establecer las nuevas conexiones. Utilizamos en esta figura una extensión gráfica de la sintaxis de UML.

Fig. 6.20 – **Diagrama de objetos para ilustrar la estructura de hilos de ejecución**



Debe ser claro que cualquier método de cualquier otro objeto del programa que se invoque desde un hilo de ejecución, se va a ejecutar sobre este mismo hilo. Esto quiere decir que los hilos no pertenecen a las clases o a las instancias sino que en un hilo se monta toda la secuencia de llamadas de métodos e instrucciones, que comienzan en el método `run()` que lo inició. El hilo se destruye cuando se termina de ejecutar dicho método.

Puesto que en algunos casos hay objetos (o atributos de tipo simple) que pueden ser compartidos por varios hilos de ejecución, se debe tener cuidado con las modificaciones concurrentes que se puedan hacer sobre ellos. Piense, por ejemplo, en el caso en el cual dos instancias de la clase `Encuentro` invocan al mismo tiempo el mismo método sobre un objeto. Si eso resulta de algún modo inconveniente, es necesario definir dicho método como sincronizado. En ese caso, si dos hilos de ejecución llaman simultáneamente el método sobre el mismo objeto, la última llamada se queda esperando hasta que la primera haya terminado su ejecución. En caso de no ser definido como sincronizado, el mismo método se ejecuta de manera simultánea, una vez sobre cada hilo. Esto se ilustra en el ejemplo 5.

Ejemplo 5

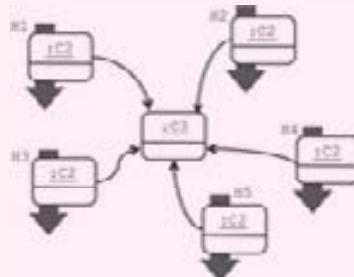


Objetivo: Mostrar la manera de definir un método como sincronizado y su comportamiento.
En este ejemplo presentamos la sintaxis con la que se declara y el comportamiento de un método sincronizado. Para esto utilizamos dos clases simples que no hacen parte del caso de estudio.

```

public class C1
{
    public void crearHilos( )
    {
        C3 obj3 = new C3( );
        for( int i = 0; i < 5; i++ )
            new C2( obj3 ).start( );
    }
}
  
```

La clase `C1` tiene un método llamado `crearHilos()` que crea e inicia cinco hilos distintos de ejecución (`H1` a `H5`). A todos les pasa como parámetro en su constructor una referencia a un objeto de la clase `C3`, que este método acaba de crear. Esto nos lleva a un diagrama de objetos como el siguiente:



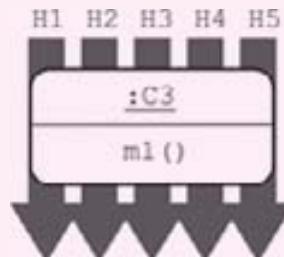
```
public class C2 extends Thread
{
    private C3 obj;

    public C2( C3 v )
    {
        obj = v;
    }

    public void run( )
    {
        obj.m1( );
        obj.m2( );
    }
}
```

La clase C2 hereda de la clase Thread, luego el método run() va a iniciar un nuevo hilo de ejecución. Allí hacemos una llamada a los métodos m1() y m2() del objeto de la clase C3 que recibió como parámetro. Después de estas dos llamadas el hilo se destruye.

El método m1() se va a ejecutar sobre el hilo de la clase C2 que hizo la llamada. Esto quiere decir que existe la posibilidad de que en los cinco hilos, de manera simultánea, se haga la llamada de este método, tal como se ilustra a continuación:



```
public class C3
{
    public void m1( )
    {
        ...
    }

    synchronized public void m2( )
    {
        ...
    }
}
```

El método m2(), en cambio, está declarado como sincronizado. Esto quiere decir que sólo un hilo a la vez entra a ejecutarlo, mientras los demás hacen fila para hacerlo después.

Si, por ejemplo, dentro del método m2() se quieren poder utilizar los atributos del objeto, con la seguridad de que nadie los va a modificar al mismo tiempo, se debe definir como sincronizado.

Declarar un método como sincronizado puede causar ineficiencia en el programa, de manera que la decisión de cuáles métodos declarar de esta manera se debe tomar con cuidado.

Tarea 5



Objetivo: Estudiar la concurrencia necesaria en el programa cliente, para evitar que la interfaz de usuario quede bloqueada mientras avanza el juego.

Siga los pasos que se describen a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n12_batallaNaval.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.

2. Localice y edite la clase `ThreadConectar` que está en el paquete `uniandes.cupi2.batallaNaval.cliente`.

¿Qué se hace en el hilo que se crea con esta clase?

¿Quién crea una instancia de esta clase e inicia la ejecución del hilo?

¿Qué información requiere el hilo para poderse ejecutar?

Dibuje el diagrama de clases que incluye esta clase.

3. Localice y edite la clase `ThreadEnviarJugada` en el mismo paquete anterior.

¿Qué se hace en el hilo que se crea con esta clase?

¿Quién crea una instancia de esta clase e inicia la ejecución del hilo?

¿Qué información requiere el hilo para poderse ejecutar?

Dibuje el diagrama de clases que incluye esta clase.

4. Localice y edite la clase `ThreadEsperarJugada` en el mismo paquete anterior.

¿Qué se hace en el hilo que se crea con esta clase?

¿Quién crea una instancia de esta clase e inicia la ejecución del hilo?

¿Qué información requiere el hilo para poderse ejecutar?

Dibuje el diagrama de clases que incluye esta clase.



El método `run()` de una clase que hereda de la clase `Thread` se puede ver como el equivalente al método `main()` de un programa completo, en el sentido de que define un punto de inicio de un nuevo hilo de ejecución.

La clase `Thread` cuenta, entre otros muchos, con los siguientes métodos:

- `Thread.currentThread()` : Este método retorna el hilo de ejecución sobre el cual se está ejecutando el método desde el cual se hace esta llamada.
- `getId()` : Retorna el identificador que Java le asignó a un hilo de ejecución. El hilo del programa principal tiene típicamente el identificador 1.

- `getName()` : Retorna el nombre que Java le asignó a un hilo de ejecución. El hilo del programa principal se llama típicamente “main”.
- `Thread.sleep(milisegundos)` : Este método permite “dormir” (dejar de ejecutar de manera temporal) el hilo sobre el cual se ejecuta este método, durante el número de milisegundos que recibe como parámetro. Durante el tiempo que se detiene la ejecución, el hilo no consume recursos del sistema. Este método puede ser usado para sincronización de tareas.

La presentación del tema de concurrencia que hicimos en este nivel no es muy profunda y sólo incluimos los elementos indispensables para hacer funcionar un programa distribuido sencillo, como el de la batalla naval. Para profundizar en el tema se recomienda consultar libros especializados.



En este punto del nivel ya hemos estudiado los problemas relacionados con la distribución y la concurrencia, y pasamos ahora a ver la manera de tratar el problema de la persistencia en nuestro caso de estudio. La conclusión que podemos sacar hasta ahora es que los dos primeros requerimientos no funcionales son fáciles de implementar en Java, y que el esfuerzo adicional que se debe hacer se centra sobre todo en el diseño del programa.

3.8. Introducción a las Bases de Datos

Una **base de datos** es una estructura en memoria secundaria que permite almacenar y manipular de manera concurrente grandes volúmenes de información. Lo que veremos en esta sección es una enorme simplificación del tema, de tal forma que sólo se puede considerar un primer paso en el uso de dichas estructuras. El hecho de que múltiples usuarios puedan estar consultando y modificando de manera simultánea la información contenida en una base de datos nos obliga, en todo momento, a mantener actualizado su contenido en memoria secundaria.

Para utilizar una base de datos se debe adquirir un producto que la maneje, puesto que los lenguajes

de programación no incluyen una implementación de este tipo de estructuras. En el mercado existen productos muy conocidos y difundidos como ORACLE, ACCESS, SQL Server, DB2, MySQL, etc. Nosotros vamos a utilizar un manejador de bases de datos libre, completamente implementado en Java, que se llama DERBY, desarrollado en el marco del proyecto APACHE y cuya información se puede encontrar en la dirección <http://db.apache.org/derby/>.

3.8.1. Estructura y Definiciones

Una base de datos es una estructura compuesta por **tablas**, cada una de las cuales tiene un nombre asociado. Una tabla, por su parte, está constituida por un conjunto de **registros**, cada uno de los cuales está dividido en

campos. Un campo es la unidad básica de consulta y manejo, y tiene asociado un nombre y un tipo. Un ejem-

pto de una base de datos se muestra en la figura 6.21, en la cual se almacena información de una universidad.

Fig. 6.21 – **Ejemplo de una base de datos para una universidad**

cursos			salones	
codigo	curso	estudiantes	nombre	capacidad
"ISIS-1204"	"Programación 1"	23	"AU-306"	35
"ISIS-1205"	"Programación 2"	25	"AU-102"	30
"FISI-1012"	"Física 1"	98	"AU-201"	30
"FISI-1013"	"Física 2"	87	"W-560"	20
"IIND-2101"	"Probabilidad"	45	"O-101"	90
"IIND-2102"	"Estadística"	52	"O102"	90

- Esta base de datos está compuesta por dos tablas. Una llamada "cursos" y la otra "salones".
- La primera tabla está compuesta por tres campos: código (una cadena de caracteres), curso (una cadena de caracteres) y estudiantes (un valor entero). Dicha tabla tiene en este ejemplo seis registros.
- La segunda tabla tiene dos campos: nombre (una cadena de caracteres) y capacidad (un valor entero). Esta tabla tiene en este ejemplo siete registros.
- Cada registro en la tabla "cursos" tiene tres valores, uno para cada campo. La tripleta ("FISI-1012", "Física 1", 98) es uno de los registros de dicha tabla.

En cada tabla se define uno de sus campos como su **llave primaria** de acceso, para el cual se debe garantizar que su valor es único (no puede haber dos registros en la tabla que tengan el mismo valor en dicho campo). En el ejemplo anterior, la llave primaria de la tabla llamada "cursos" debería ser el código del curso (no es imaginable que haya dos cursos con el mismo código). El hecho de seleccionar un campo como llave nos va a garantizar un rápido acceso a los registros cuando le suministremos dicho valor.

Las operaciones básicas en una base de datos son:

- Crear una nueva tabla, dando la descripción de sus campos (nombre y tipo de cada uno).
- Agregar un registro a una tabla, indicando la tabla y dando los valores para cada uno de sus campos.
- Modificar un registro, indicando la tabla, el registro y la modificación que se quiere hacer.

- Eliminar un registro, indicando la tabla y el registro que se quiere suprimir.
- Localizar los registros de una tabla que tengan una cierta característica.

3.8.2. El Lenguaje de Consulta SQL

SQL (*Structured Query Language*) es un lenguaje simple de instrucciones textuales que permite manipular una base de datos. Tiene la ventaja de ser un estándar, de tal forma que es comprendido por todos los manejadores de bases de datos, independiente del producto preciso que utilicemos.

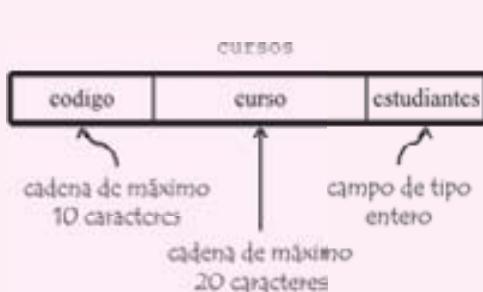
En esta sección sólo veremos un pequeño subconjunto de las instrucciones y tipos de datos que puede manejar SQL. La presentación la vamos a orientar a las operaciones básicas que necesitamos para nuestro caso de estudio y la haremos alrededor de ejemplos más que presentando la sintaxis general.



Las instrucciones del lenguaje SQL no hacen parte de los lenguajes de programación. Por ahora las debemos imaginar como instrucciones que le deben llegar por algún medio a la base de datos. Más adelante veremos cómo enviarlas desde Java hasta la base de datos que utilicemos.

- ¿Cómo agregar una tabla a una base de datos?

```
CREATE TABLE cursos
(
    codigo varchar(10),
    curso varchar(20),
    estudiantes int,
    PRIMARY KEY( codigo )
)
```



■ La instrucción CREATE TABLE crea una nueva tabla en la base de datos.

■ El tipo "varchar" se refiere a una cadena de caracteres, de la cual definimos entre paréntesis la longitud máxima.

■ El tipo "int" se utiliza para definir campos con un valor entero.

■ La instrucción PRIMARY KEY indica el nombre del campo con la llave primaria de la tabla.

- ¿Cómo agregar un registro a una tabla?

```
INSERT INTO cursos VALUES
(
    'ISIS-1204',
    'Programación 1',
    23
)
```

CURSOS		
codigo	curso	estudiantes
"ISIS-1204"	"Programación 1"	23

■ La instrucción INSERT INTO permite agregar un registro a una tabla.

Se debe dar un valor para cada uno de los campos, respetando el tipo al cual éste pertenece.

■ En SQL se utiliza una comilla simple para indicar que el valor es una cadena de caracteres. En el dibujo utilizamos comillas dobles dentro de la base de datos, para usar la misma sintaxis que utilizamos en Java para indicar un literal de la clase String.

```
INSERT INTO cursos VALUES
(
    'FISI-1012',
    'Física 1',
    98
)
```

CURSOS		
codigo	curso	estudiantes
"FISI-1012"	"Física 1"	98
"ISIS-1204"	"Programación 1"	23

- ¿Cómo eliminar un registro de una tabla?

```
DELETE FROM cursos WHERE
```

```
codigo = 'ISIS-1204'
```

CURSOS		
codigo	curso	estudiantes
"FISI-1012"	"Física 1"	98

■ La instrucción DELETE FROM permite eliminar un registro de una tabla.

■ Debemos suministrar la expresión que permite localizar el registro que queremos eliminar.

- ¿Cómo modificar un registro de una tabla?

```
UPDATE cursos
```

```
SET estudiantes = 50
```

```
WHERE codigo = 'FISI-1012'
```

CURSOS		
codigo	curso	estudiantes
"FISI-1012"	"Física 1"	50

■ La instrucción UPDATE permite modificar registros dentro de una tabla de la base de datos.

■ En la sección SET se establece el cambio que se quiere hacer. Allí se puede cambiar el valor de cualquier campo, utilizando para esto operadores aritméticos (+, -, *, /) o valores constantes.

■ Si hay varias modificaciones se separan por una coma (',').

■ SQL cuenta con un extenso conjunto de operadores y variantes que están fuera del alcance de este libro.

```
UPDATE cursos
```

```
SET estudiantes =
estudiantes+1
```

```
WHERE codigo = 'FISI-1012'
```

CURSOS		
codigo	curso	estudiantes
"FISI-1012"	"Física 1"	51

- ¿Cómo recuperar información de una base de datos?

<pre>SELECT estudiantes FROM cursos WHERE codigo = 'FISI-1012'</pre>	<div style="text-align: center; border: 1px solid black; padding: 5px;">51</div>	<ul style="list-style-type: none"> ■ La instrucción SELECT permite localizar un campo de un registro de una tabla, como aparece en este ejemplo. ■ La respuesta no es el valor sino el campo del registro, tal como se sugiere en la figura. ■ La manera de manipular la respuesta que nos da esta instrucción la veremos más adelante. 			
<pre>SELECT curso, estudiantes FROM cursos WHERE codigo = 'FISI-1012'</pre>	<div style="text-align: center; border: 1px solid black; padding: 5px;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">"Fisica 1"</td> <td style="padding: 2px;">51</td> </tr> </table> </div>	"Fisica 1"	51	<ul style="list-style-type: none"> ■ Esta misma instrucción permite localizar un conjunto de campos de un registro de una tabla. 	
"Fisica 1"	51				
<pre>SELECT * FROM cursos WHERE codigo = 'FISI-1012'</pre>	<div style="text-align: center; border: 1px solid black; padding: 5px;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">"FISI-1012"</td> <td style="padding: 2px;">"Fisica 1"</td> <td style="padding: 2px;">51</td> </tr> </table> </div>	"FISI-1012"	"Fisica 1"	51	<ul style="list-style-type: none"> ■ De la misma manera podemos recuperar todos los campos de un registro que cumple una cierta característica.
"FISI-1012"	"Fisica 1"	51			

Tarea 6

Objetivo: Definir la base de datos que necesitamos en el caso de estudio y construir algunas instrucciones en SQL para manipularla.

Escriba las instrucciones en SQL que se piden en cada paso.

Crear una tabla llamada **resultados**, que tenga tres campos: **nombre** (cadena de 32 caracteres como máximo), **ganados** (valor entero), **perdidos** (entero). La llave primaria es el campo **nombre**.

Insertar en la tabla anterior un registro para indicar que el jugador llamado "Barbanegra" lleva 10 partidos ganados y 5 perdidos.

Insertar en la tabla anterior un registro para indicar que el jugador llamado "Drake" lleva 0 partidos ganados y 25 perdidos.

Insertar en la tabla anterior un registro para indicar que el jugador llamado "Nelson" lleva 20 partidos ganados y 2 perdidos.	
Eliminar de la tabla de resultados al jugador con nombre "Drake".	
Aumentar en uno el número de partidos ganados del jugador "Barbanegra".	
Multiplicar por dos el número de partidos perdidos del jugador "Nelson".	



En el CD que acompaña al libro puede encontrar un entrenador del lenguaje SQL, con el cual puede crear de manera interactiva tablas, modificarlas y hacer consultas.

3.8.3. El Framework JDBC

Para poder utilizar fácilmente una base de datos desde Java, contamos con el *framework* JDBC (*Java Database*

Connectivity), el cual tiene un conjunto de clases por medio de las cuales nos podemos conectar con una base de datos y le podemos enviar instrucciones escritas en SQL. Esta estructura se ilustra en la figura 6.22.

Fig. 6.22 – Relación entre una base de datos y un programa en Java



El *driver* o controlador de la base de datos depende del producto específico que estemos utilizando. Por ejemplo, hay un controlador JDBC para DERBY y uno

distinto para ORACLE. Lo importante para nosotros es que de esta manera logramos desacoplar nuestro programa de un manejador de bases de datos

específico. Para trabajar en nuestro caso de estudio, debemos incluir el archivo `derby.jar` en el proyecto creado en Eclipse (en el directorio `lib`), puesto que allí se encuentra la implementación de la base de datos que vamos a utilizar y el controlador que vamos a necesitar.

Hay dos grandes etapas en el uso de JDBC: (1) crear la conexión a la base de datos y (2) utilizar la conexión para enviar por allí las instrucciones en SQL. Vamos a comenzar por la segunda etapa, que resulta

mucho más interesante desde el punto de vista de la programación.

En JDBC existe una clase llamada `Connection` (en el paquete `java.sql`), que representa el medio por el cual nos podemos comunicar con la base de datos. La etapa de conexión termina cuando hemos creado una instancia de dicha clase, conectada a la base de datos en la cual tenemos nuestra información. En la figura 6.23 aparece ilustrada la conexión a la base de datos para el caso del programa servidor del juego de Batalla Naval.

Fig. 6.23 – **Conexión a la base de datos para el programa servidor del caso de estudio**



Vamos a presentar ahora la manera de enviar instrucciones en SQL a la base de datos y de manipular la respuesta que obtenemos, respondiendo las tres preguntas siguientes:

- ¿Cómo crear una instrucción en SQL?

La clase `Connection` tiene un método llamado `createStatement()`, que retorna un objeto de la clase `Statement` que representa en JDBC una instrucción en SQL. Si hay algún problema en la creación de dicho objeto, se lanza la excepción `SQLException`.

```

try
{
    Statement st = conexion.createStatement( );
    ...
    s.close( );
}
catch( SQLException e )
{
    // Recuperación del error
}

```

 Con el método `createStatement()` obtenemos un objeto de la clase `Statement`, por medio del cual podemos enviar instrucciones en el lenguaje SQL a la base de datos.

 Cuando dejemos de utilizar dicho objeto, debemos usar el método `close()`.

- ¿Cómo ejecutar una instrucción SQL a través de una conexión a una base de datos?

Para enviar modificaciones a la base de datos utilizamos el método `executeUpdate()` de la clase `Statement`, el cual recibe como parámetro una cadena de caracteres con la instrucción en el lenguaje SQL. Esta instrucción lanza la excepción `SQLException` en caso de problemas. Este método se utiliza para las instrucciones INSERT, DELETE y UPDATE de SQL.

```
String sql = "UPDATE resultados SET perdidos=perdidos+1 WHERE nombre='Barbanegra'";  
Statement st = conexion.createStatement();  
st.executeUpdate( sql );
```

La clase `Statement` tiene un método llamado `execute()`, que se puede utilizar para ejecutar la instrucción de SQL de creación de tablas.

```
String sql = "CREATE TABLE resultados (nombre varchar(32), ganados int, perdidos int,  
PRIMARY KEY (nombre))";  
Statement st = conexion.createStatement();  
st.execute( sql );
```

Si queremos hacer una consulta a la base de datos, utilizamos el método `executeQuery()` de la clase `Statement`, el cual recibe como parámetro una cadena de caracteres con la instrucción en el lenguaje SQL. Este método retorna un objeto de la clase `ResultSet` con la respuesta a la consulta.

```
String sql = "SELECT * FROM resultados WHERE nombre='Barbanegra'";  
Statement st = conexion.createStatement();  
ResultSet resultado = st.executeQuery( sql );
```

- ¿Cómo manipular la respuesta obtenida de una consulta?

La clase `ResultSet` representa una respuesta a una consulta, la cual se encuentra constituida por una secuencia de elementos, cada uno de los cuales tiene uno o varios campos. Los objetos de esta clase se comportan de una manera semejante a un iterador, utilizando el método `next()` para avanzar sobre la secuencia. La clase cuenta con los métodos `getString()` y `getInt()`, que se utilizan para extraer la información de los campos que hacen parte del registro actual de la secuencia, tal como se muestra a continuación:

```

String sql = "SELECT * FROM resultados WHERE nombre='Barbanegra'";
Statement st = conexion.createStatement();
ResultSet resultado = st.executeQuery( sql );

if( resultado.next( ) )                                // Hay al menos un elemento en la
resuesta
{
    String nombre = resultado.getString( 1 );          // Toma el primer valor como una cadena
    int ganados = resultado.getInt( 2 );                // Toma el segundo valor como un entero
    int perdidos = resultado.getInt( 3 );               // Toma el tercer valor como un entero
}
resultado.close( );                                    // Cierra la respuesta
st.close( );                                         // Cierra la instrucción

```

Veamos un ejemplo detallado de las consultas y el manejo de las respuestas con la clase `ResultSet`:



Ésta es la base de datos sobre la cual vamos a trabajar. Tiene una única tabla llamada “cursos”, con información de una universidad.

CURSOS		
codigo	curso	estudiantes
“ISIS-1204”	“Programación 1”	23
“ISIS-1205”	“Programación 2”	25
“FISI-1012”	“Física 1”	98
“FISI-1013”	“Física 2”	87
“IIND-2101”	“Probabilidad”	45
“IIND-2102”	“Estadística”	52



Si hacemos la consulta:

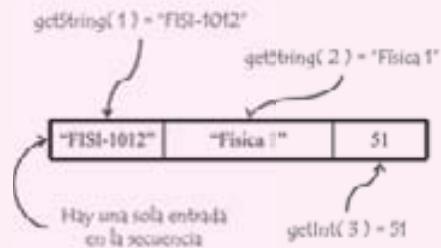
```
SELECT * FROM cursos WHERE codigo='FISI-1012'
```



Obtenemos el `ResultSet` que aparece en la figura, que sólo tiene una fila con la respuesta. Con el método `next()` obtenemos dicha entrada y con los métodos `getString()` y `getInt()` tenemos acceso a los valores de sus campos.



Si la respuesta es vacía, el método `next()` retorna falso.



Si hacemos la consulta:

```
SELECT curso FROM cursos WHERE estudiantes > 75
```

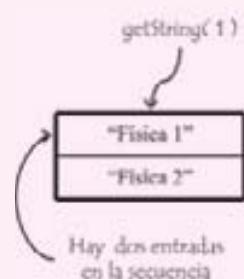


Obtenemos el `ResultSet` que aparece en la figura, el cual tiene dos entradas, cada una con un solo campo. Para recorrerlo podemos usar el siguiente código:

```

while( resultado.next( ) )
{
    String nombre = resultado.getString( 1 );
    ...
}

```



Pasemos ahora al problema de establecer la conexión con la base de datos. Lo primero que debemos advertir es que este proceso depende del producto específico que estemos utilizando. Aquí vamos a presentar el tema de conexión a una base de datos DERBY como la respuesta a algunas preguntas:

- ¿Dónde se guarda físicamente la información de una base de datos?

Una base de datos se representa internamente como un conjunto de directorios y archivos que el manejador almacena en algún punto de la memoria secundaria, con una estructura interna bastante compleja. DERBY localiza físicamente esta información en la ruta definida por la variable del sistema llamada `derby.system.home`.

Puesto que en nuestro caso de estudio nos interesa que dicha información se almacene en el subdirectorio `data` de nuestro proyecto `n12_BatallaNaval`, vamos a proceder de la siguiente manera: (1) agregar en el archivo de configuración del programa servidor (llamado `servidor.properties`) una nueva entrada con el valor `admin.db.path=../data` y (2) utilizar las siguientes instrucciones que permiten asignar un valor a una variable del sistema (`config` es el atributo de la clase `Properties` en donde se leyó el archivo de configuración):

```
File data = new File( config.getProperty( "admin.db.path" ) );
System.setProperty( "derby.system.home", data.getAbsolutePath( ) );
```

- ¿Cómo crear un controlador (*driver*) para la base de datos?

Aquí necesitamos algunas clases y métodos que no han sido estudiados anteriormente. Por esta razón lo más conveniente es utilizar las instrucciones tal como se muestran a continuación.

```
admin.db.driver=org.apache.derby.jdbc.EmbeddedDriver
```

 Esta propiedad se debe agregar al archivo de configuración del programa servidor.

```
String driver = config.getProperty( "admin.db.driver" );
Class.forName( driver ).newInstance( );
```

 Con la primera instrucción obtenemos el valor que la propiedad tiene en el archivo de configuración.

 Con la segunda instrucción se crea el controlador.

- ¿Cómo crear una conexión a la base de datos?

```
admin.db.url=jdbc:derby:batallaNaval;create=true
```

 Esta propiedad se debe agregar al archivo de configuración del programa servidor.

 En el directorio “`data/batallaNaval`” se va a almacenar la información de nuestra base de datos.

```
String url = config.getProperty( "admin.db.url" );
conexion = DriverManager.getConnection( url );
```

 Con la primera instrucción obtenemos el valor que la propiedad tiene en el archivo de configuración.

 Con la segunda instrucción se crea la conexión a nuestra base de datos.

- ¿Cómo desconectarse de la base de datos?

Antes de terminar la ejecución del programa servidor se debe desconectar de la base de datos.

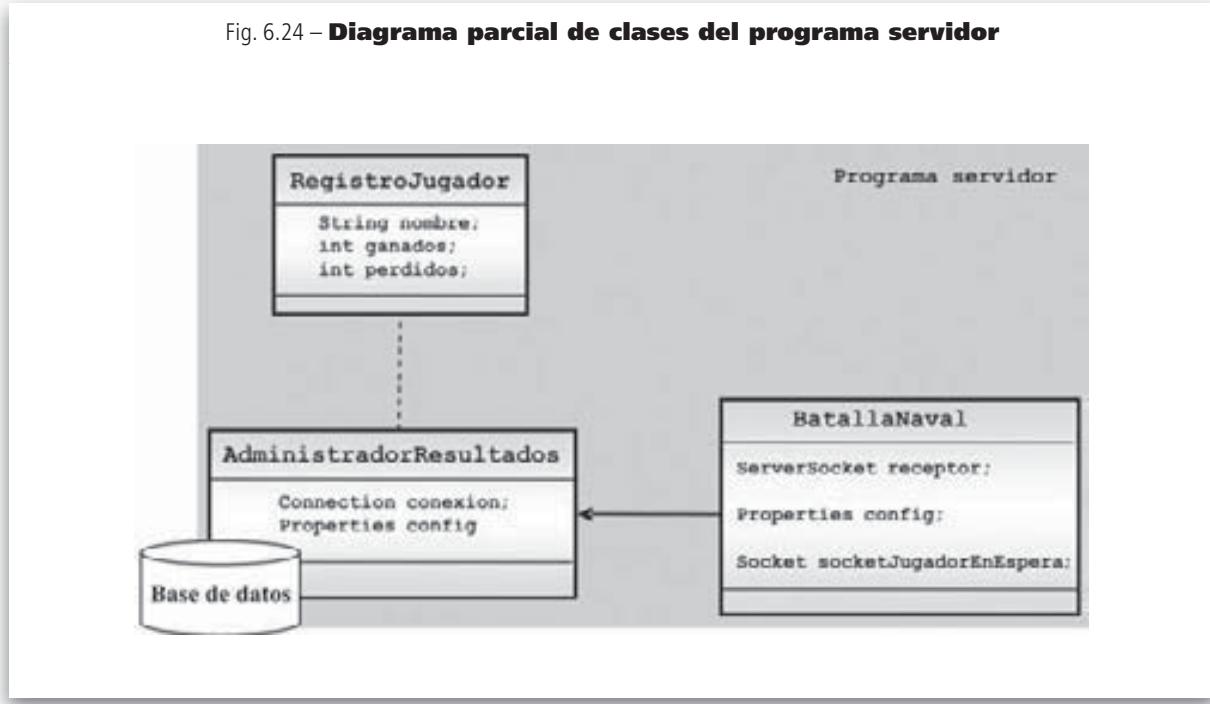
<pre>admin.db.shutdown=jdbc:derby:;shutdown=true</pre>	<ul style="list-style-type: none"> ■ Esta propiedad se debe agregar al archivo de configuración del programa servidor.
<pre>conexion.close(); String down = config.getProperty("admin.db.shutdown"); DriverManager.getConnection(down);</pre>	<ul style="list-style-type: none"> ■ Lo primero que debemos hacer es cerrar la conexión que tenemos con la base de datos. ■ Luego, damos a la clase DriverManager la instrucción de desconectarse de la base de datos.

3.8.4. Persistencia en el Caso de Estudio

Volvamos de nuevo a nuestro caso de estudio, y veamos el diagrama de clases del programa servidor que tiene que ver con el manejo de la persistencia. En la figura 6.24 aparece la versión definitiva que vamos a utilizar,

en donde se puede apreciar que agregamos una nueva clase llamada `AdministradorResultados`, a la que le delegamos toda la responsabilidad de manejar la información persistente. Esto puede facilitar la evolución del programa, puesto que podríamos cambiar la manera de guardar la información en memoria secundaria y sólo se vería afectada esta clase.

Fig. 6.24 – **Diagrama parcial de clases del programa servidor**



En la tarea 7 haremos un recorrido por el programa servidor, identificando los principales aspectos que tienen que ver con el manejo de la base de datos.

Tarea 7

Objetivo: Revisar el manejo que se hace de la base de datos en el caso de estudio.

Siga los pasos que se describen a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n12_batallaNaval.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
2. Localice y edite el archivo `servidor.properties`, que se encuentra en el directorio `data`. Mire los valores que se definieron para cada una de las propiedades.
3. En el directorio `data` localice el subdirectorio `batallaNaval`. Allí está nuestra base de datos. ¿Qué directorios y archivos encuentra allí dentro? ¿Es posible editar estos archivos?
4. Vaya al directorio `lib`. Allí debe aparecer el archivo `derby.jar`, el cual contiene la implementación del manejador de bases de datos que estamos utilizando. En Eclipse, haga clic con el botón derecho del ratón sobre el nombre del proyecto (`n12_batallaNaval`). Seleccione en el menú que aparece la opción `Properties`. En la ventana que aparece localice la sección llamada Java Build Path. Vaya ahora a la pestaña cuyo nombre es `Libraries`. ¿Qué archivos `jar` aparecen allí? En este punto se deben agregar los archivos `jar` cuyas clases queramos utilizar dentro de nuestro programa.
5. Edite la clase `AdministradorResultados` del programa servidor. Estudie los métodos `conectarBD()` y `desconectarBD()`. ¿Quién invoca estos métodos en el programa? ¿Cómo se asegura el programa que invoca siempre el método de desconexión antes de terminar su ejecución?
6. En la misma clase anterior, localice el método `consultarRegistroJugador()`. ¿Qué recibe este método como parámetro? ¿Qué retorna? ¿En qué caso crea un nuevo registro en la base de datos?
7. En la misma clase anterior, localice el método `registrarVictoria()`. ¿Qué recibe este método como parámetro? ¿Qué comando de SQL utiliza? ¿Qué método de JDBC llama?
8. En la misma clase anterior, localice el método `registrarDerrota()`. ¿Qué recibe este método como parámetro? ¿Qué comando de SQL utiliza? ¿Qué método de JDBC llama?
9. En la misma clase anterior, localice el método `consultarRegistrosJugadores()`. ¿Quién invoca este método en el programa? ¿Qué retorna? ¿Qué pasa en SQL cuando en el comando SELECT no aparece la parte WHERE? ¿Cuántas entradas hay en el ResultSet de la respuesta?

3.9. Extensiones al Caso de Estudio

A continuación proponemos un conjunto de tareas para extender el programa de la batalla naval.

Tarea 8

Objetivo: Extender el programa que permite jugar Batalla Naval.

Siga los pasos que se describen a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n12_batallaNaval.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
2. Agregue a la base de datos una nueva tabla llamada `conexiones`, para almacenar información histórica sobre las conexiones de cada jugador. Dicha tabla debe tener un campo con el nombre del jugador, un campo con la dirección desde la cual el jugador se conectó al servidor la última vez y un campo con el número total de conexiones.

3. Asocie con el botón **Opción 1** un nuevo requerimiento funcional, que recibe como entrada el nombre de un jugador y retorna su información histórica. Asegúrese de no bloquear la interfaz de usuario del programa servidor mientras atiende este nuevo requerimiento, creando un nuevo hilo de ejecución para atenderlo.

Tarea 9

Objetivo: Extender el programa que permite jugar Batalla Naval.

Siga los pasos que se describen a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n12_batallaNaval.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
2. Cambie el protocolo del juego de tal manera que si un jugador impacta un barco de su oponente, tenga derecho a volver a jugar de manera inmediata. Antes de comenzar a hacer las modificaciones, dibuje claramente el nuevo protocolo.

Tarea 10

Objetivo: Extender el programa que permite jugar Batalla Naval.

Siga los pasos que se describen a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n12_batallaNaval.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
2. Asocie con el botón **Opción 2** un nuevo requerimiento funcional que informa quién es el jugador más experimentado del juego. Para esto debe buscar en la base de datos el nombre de la persona que tenga el mayor número de encuentros, independientemente de si los ha ganado o perdido.

Tarea 11

Objetivo: Extender el programa que permite jugar Batalla Naval.

Siga los pasos que se describen a continuación.

1. Si no lo ha hecho antes, cree en Eclipse un proyecto a partir del archivo `n12_batallaNaval.zip`, el cual está disponible en el CD que acompaña al libro o en el sitio web.
2. Agregue a la base de datos una nueva tabla llamada `encuentros`, para almacenar información histórica sobre el último encuentro que hayan tenido dos jugadores. En el primer campo debe estar el identificador del encuentro, que es la concatenación de los nombres de los dos jugadores, ordenados alfabéticamente de izquierda a derecha (por ejemplo "Barbanegra-Drake"). En el segundo campo aparece el número total de disparos que se hicieron en el último juego que tuvieron. En el tercer campo se tiene el número de barcos hundidos por el primer jugador. En el cuarto campo, el número de barcos hundidos por el segundo jugador. En el quinto campo, el nombre del ganador del último encuentro.
3. Modifique el programa para que dentro del protocolo envíe esta información (adicional a la que ya está enviando) a los dos jugadores en el momento de comenzar el encuentro.

4. Glosario de Términos

GLOSARIO



Complete las siguientes definiciones como una manera de resumir los conceptos aprendidos en el nivel. Asegúrese de entender bien el significado de cada uno de estos términos, puesto que resumen los conceptos importantes de este nivel del libro.

Distribución:

Canal de comunicación:

Dirección de un computador:

Puerto:

Flujo de lectura:

Flujo de escritura:

Protocolo de comunicación:

Clase ServerSocket:

Clase Socket:

Comando ipconfig:

Comando netstat:

Concurrencia:

Hilo de ejecución:

Método sincronizado:

Clase Thread:

Base de datos:

Tabla:

Registro:

Campo:

Llave primaria:

SQL:

JDBC:

Clase ResultSet:

Driver o conector:

5. Hojas de Trabajo



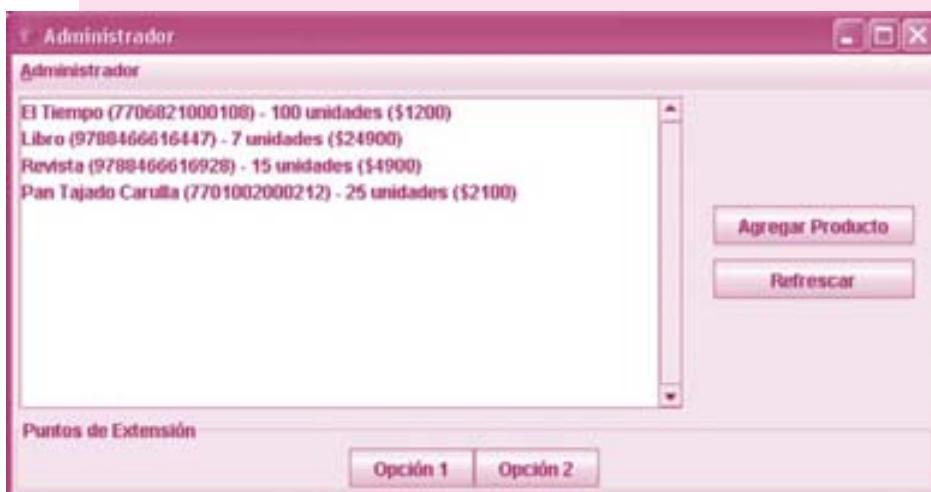
5.1. Hoja de Trabajo N° 1: Registro de Ventas de un Almacén

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

Se quiere construir un programa para manejar el inventario y las ventas de un almacén, en el cual existen múltiples puntos de pago, a los cuales se acercan los clientes para pagar sus productos. Este programa debe tener dos partes: un servidor, que mantiene el inventario del almacén, y los programas cliente, que se conectan al servidor y desde los cuales se informa el número de unidades que se venden de cada producto.

Como parte del inventario, se almacena el código del producto (que es único), su nombre, su precio y el número de unidades disponibles. El programa servidor debe ofrecer dos opciones al administrador del almacén: (1) agregar un nuevo producto y (2) consultar el inventario. La interfaz de usuario debe ser la que se presenta a continuación. Con el botón **Refrescar** se actualiza la lista de productos que se encuentra desplegada.

código. Como resultado de la operación, el programa debe mostrar el valor acumulado de la venta. La interfaz de usuario del programa cliente debe ser la siguiente:



El programa cliente sólo ofrece una opción al operador del punto de venta, que consiste en registrar la venta de una unidad de un producto, dado su

Cuando se hace una venta desde un punto de pago, el programa servidor contesta con información sobre el precio del producto vendido, de tal manera que el programa cliente pueda llevar el acumulado.

Tanto el programa servidor como el programa cliente deben tener un archivo de configuración en el que se tenga la información del puerto para establecer la conexión.

El programa servidor debe soportar concurrencia y almacenar la información del inventario en una base de datos.

Diagrama de casos de uso. Dibuje el diagrama de casos de uso del problema.

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

Requerimiento funcional 1	Nombre	R1 – Agregar un nuevo producto	Actor	Administrador
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 2	Nombre	R2 – Consultar el inventario	Actor	Administrador
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 3	Nombre	R3 – Registrar la venta de una unidad de un producto	Actor	Operador
	Resumen			
	Entrada			
	Resultado			

Requerimientos no funcionales. Describa los requerimientos no funcionales planteados en el enunciado.

Requerimiento no funcional 1	Tipo: Persistencia Descripción:
Requerimiento no funcional 2	Tipo: Conurrencia Descripción:
Requerimiento no funcional 3	Tipo: Distribución Descripción:

Modelo conceptual. Estudie el siguiente diagrama de clases, producto de la etapa de análisis. Allí no aparece todavía la división entre programa servidor y programa cliente.

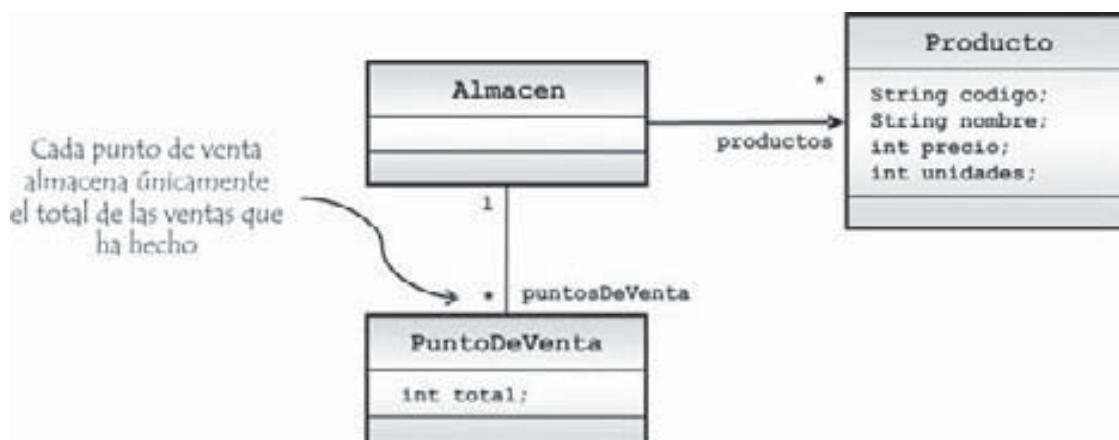


Diagrama de clases del programa cliente. Estudie el siguiente diagrama de clases, asegurándose de entender el significado de cada uno de los atributos.

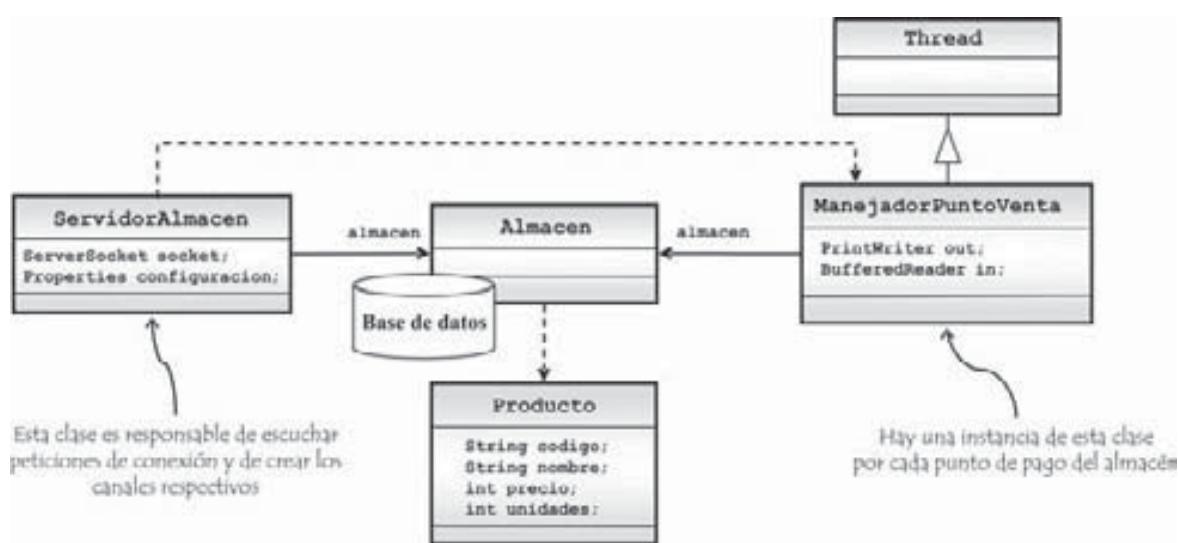
```

PuntoDeVenta
Socket socket;
Properties configuracion;
int total;
PrintWriter out;
BufferedReader in;

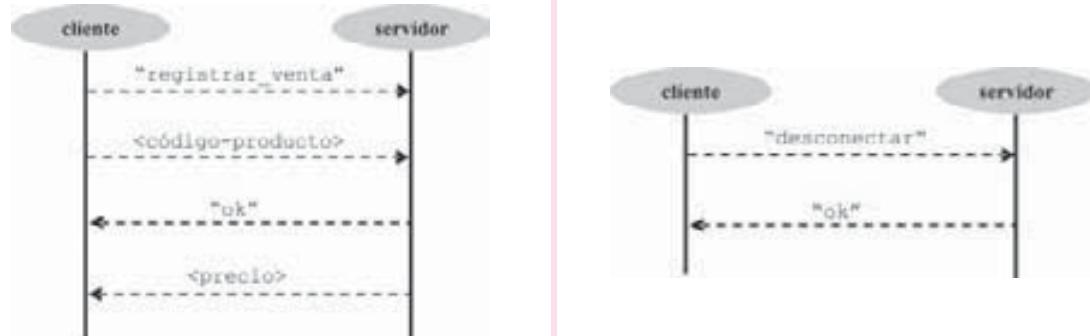
```

Clase	Atributo	Descripción
PuntoDeVenta	socket	Canal a través del cual el punto de venta se comunica con el servidor.
PuntoDeVenta	configuracion	Conjunto de propiedades que define la configuración del programa.
PuntoDeVenta	total	Valor total de las ventas hechas hasta el momento.
PuntoDeVenta	out	Flujo de salida hacia el canal de comunicación.
PuntoDeVenta	in	Flujo de entrada desde el canal de comunicación.

Diagrama de clases del programa servidor. Estudie el siguiente diagrama de clases, asegurándose de entender la responsabilidad de cada una de las entidades, al igual que el significado de cada atributo y asociación.



Protocolos de comunicación. El programa tiene dos protocolos muy simples, los cuales se muestran a continuación. El primero sirve para registrar una venta desde un punto de pago y el segundo para informar la desconexión de un programa cliente.



Persistencia. Para almacenar la información de los productos, vamos a utilizar una base de datos que sólo tiene una tabla, cuya estructura se presenta a continuación. Escriba en SQL las instrucciones que se piden.

producto			
codigo	nombre	precio	unidades
"9788466616447"	"Libro"	24900	7

Annotations: 'Llave primaria' points to the 'codigo' column. 'Cadena de 15 caracteres' points to the 'nombre' column. 'Valor entero' points to the 'precio' and 'unidades' columns. 'Cadena de 40 caracteres' points to the 'codigo' column.

Instrucción SQL para la creación de la tabla:

Instrucción SQL para agregar un producto dado:

Instrucción SQL para eliminar un producto, dado su código:

Implementación. Cree el respectivo proyecto en Eclipse. Localice y edite la clase `PuntoDeVenta`. Conteste las preguntas que se plantean a continuación.

¿Cuál es el objetivo de cada una de las constantes definidas en la clase?

Edite el archivo de configuración del programa cliente. ¿Qué información aparece allí?

¿Qué método de la clase se encarga de la conexión con el servidor? ¿En qué puerto busca la conexión?

¿Qué método de la clase es responsable de hacer la desconexión del servidor? Verifique que se cumpla el protocolo.

Estudie el método de la clase llamado `registrarVenta()` y mire la manera como implementa el protocolo.

Implementación. Localice y edite en Eclipse la clase `ServidorAlmacen`. Conteste las preguntas que se plantean a continuación.

Edite el archivo de configuración del programa servidor. ¿Qué información aparece allí?

Estudie el método de la clase llamado `recibirConexiones()`. ¿En qué puerto escucha las conexiones? ¿Qué hace para atender cada conexión? ¿Cómo garantiza que cada programa cliente que se conecta se atiende en un hilo de ejecución distinto?

Implementación. Localice y edite en Eclipse la clase `ManejadorPuntoVenta`. Conteste las preguntas que se plantean a continuación.

¿Cuál es el objetivo de cada una de las constantes definidas en la clase?

¿Qué parámetros recibe el constructor? ¿Quién se encarga de enviar dichos parámetros?

¿Qué hace el método `run()`? ¿Qué comandos es capaz de interpretar? ¿Qué hace en caso de error?

Estudie el método de la clase llamado `registrarVenta()`. Verifique que cumple el protocolo diseñado.

Implementación. Localice y edite en Eclipse la clase Almacen. Conteste las preguntas que se plantean a continuación.

Estudie el método de la clase llamado `agregarProducto()`. ¿Qué parámetros recibe? ¿Cómo verifica que no haya un producto en la base de datos con el mismo código? ¿Qué instrucción de SQL utiliza para agregar el nuevo producto? ¿Qué valor retorna el método `executeUpdate()`?

Estudie el método llamado `registrarVenta()`. ¿Qué instrucción de SQL utiliza para registrar la venta? ¿Cómo sabe si la modificación se pudo hacer en la base de datos? ¿Qué hace en caso de error?

¿Qué responsabilidad tiene el método `consultarProducto()`? ¿Qué instrucción de SQL utiliza para hacer la consulta? ¿Para qué sirve el método `next()` de la clase `ResultSet`?

Implementación. Haga las modificaciones del programa que se plantean a continuación.

1. Agregue una nueva tabla a la base de datos llamada "ventas". Dicha tabla debe tener tres campos: (1) una cadena de caracteres con la IP del punto de venta (ésta es la llave primaria de la tabla), (2) el código de un producto del almacén y (3) el número total de unidades de ese producto que se han vendido en dicho punto de venta. Haga todas las modificaciones necesarias al programa, para que el administrador de la tienda pueda consultar esta información.
2. Modifique el programa de manera que sea posible vender más de una unidad de cada producto desde el punto de pago. Para esto, modifique la interfaz de usuario del programa cliente y haga los ajustes necesarios en el protocolo de comunicación definido.
3. Modifique el servidor de manera que si, al agregar un producto dicho código ya está presente en la base de datos, en lugar de rechazar la operación, agregue el número de unidades que llegan. No importa que el nombre del producto o el precio no coincidan.

4. Agregue una nueva tabla a la base de datos llamada "promociones", que tenga tres campos: (1) código de la promoción (cadena de seis caracteres, con un valor único), (2) el porcentaje de descuento asociado con la promoción y (3) el código del producto al que se le aplica (un mismo producto puede tener muchas promociones distintas sobre él). Haga todas las modificaciones necesarias al programa servidor y al programa cliente, de manera que al hacer una compra el cliente pueda suministrar el código de la promoción que quiere aplicar sobre el producto que está comprando.
5. Asocie con el botón **Opción 1** un nuevo requerimiento funcional que aumenta en 1% el precio de todos los productos.
6. Asocie con el botón **Opción 2** un nuevo requerimiento funcional que elimina del inventario un producto, dado su código.



5.2. Hoja de Trabajo N° 2: Mensajería Instantánea

Enunciado. Analice la siguiente lectura y conteste las preguntas que se plantean más adelante, las cuales corresponden a los temas tratados en este nivel.

Se quiere construir un programa de mensajería instantánea para el intercambio de textos, entre parejas de usuarios que se encuentran en computadores distintos conectados a una red. Este sistema se parece a algunas aplicaciones existentes, como lo son MSN Messenger de Microsoft, Yahoo Messenger o AIM de AOL.

Para desarrollar este programa debemos dividir su construcción en dos partes. Una parte debe funcionar como el servidor, el cual deberá recibir las conexiones de los clientes, mantener el estado de la conexión de los usuarios (conectados o desconectados) y también deberá guardar la lista de contactos de cada usuario. El programa cliente será utilizado para conectarse al servidor, iniciar nuevas conversaciones, enviar mensajes y agregar nuevos amigos a la lista de contactos. Mientras esté utilizando el sistema, recibirá notificaciones cuando sus amigos se conecten o desconecten. El hecho de que un usuario tenga a otro

usuario dentro de su lista de amigos no implica que al revés también sea cierto.

El programa cliente debe ofrecer las siguientes opciones al usuario: (1) Conectarse al servidor. Para esto el usuario debe suministrar su nombre, el cual lo identifica en el sistema. La información del IP del servidor y el puerto al cual se debe conectar estarán definidos en un archivo de propiedades. (2) Agregar un amigo a la lista de contactos. Para poder realizar esto se debe suministrar el nombre del usuario que se quiere agregar. (3) Iniciar una conversación. El usuario escoge una sola persona con la que quiere hablar cuyo estado sea conectado, y una ventana para conversar se abre para ambos usuarios. No se puede tener una conversación entre tres personas. (4) Enviar un mensaje de texto dentro de una conversación.

La interfaz de usuario del programa cliente debe basarse en dos ventanas. La primera, con la lista de amigos, tal como se muestra a continuación:



La segunda es la ventana de la conversación, que debe tener la visualización siguiente, la cual se abre cuando un usuario utiliza el botón de

Abrir Conversación después de haber seleccionado a uno de sus amigos:



La interfaz de usuario del programa servidor debe permitir la consulta de todos los usuarios que en un

momento dado se encuentren conectados, tal como se muestra a continuación.



El programa servidor debe soportar concurrencia y almacenar la información del inventario en una base de datos.

Diagrama de casos de uso. Dibuje el diagrama de casos de uso del problema.

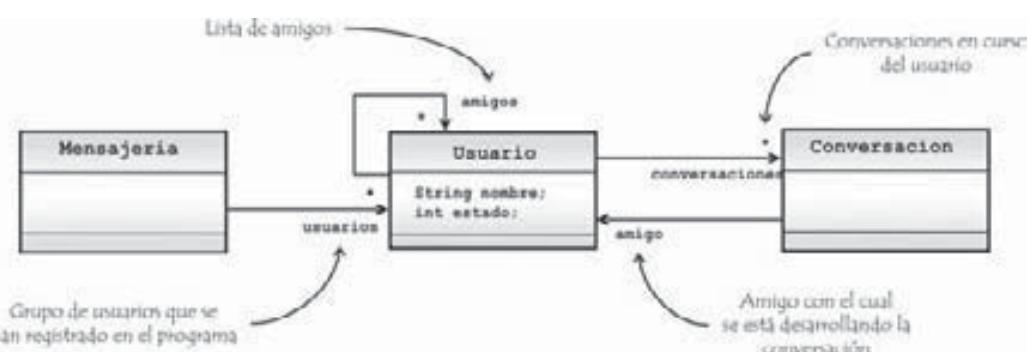
Requerimientos funcionales. Especifique los requerimientos funcionales descritos en el enunciado.

Requerimiento funcional 1	Nombre	R1 – Iniciar una sesión	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 2	Nombre	R2 – Agregar un amigo a la lista de contactos	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 3	Nombre	R3 – Iniciar una conversación con un amigo	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 4	Nombre	R4 – Enviar un mensaje de texto dentro de una conversación	Actor	Usuario
	Resumen			
	Entrada			
	Resultado			
Requerimiento funcional 5	Nombre	R5 – Consultar la lista de usuarios conectados al programa	Actor	Administrador
	Resumen			
	Entrada			
	Resultado			

Requerimientos no funcionales. Describa los requerimientos no funcionales planteados en el enunciado.

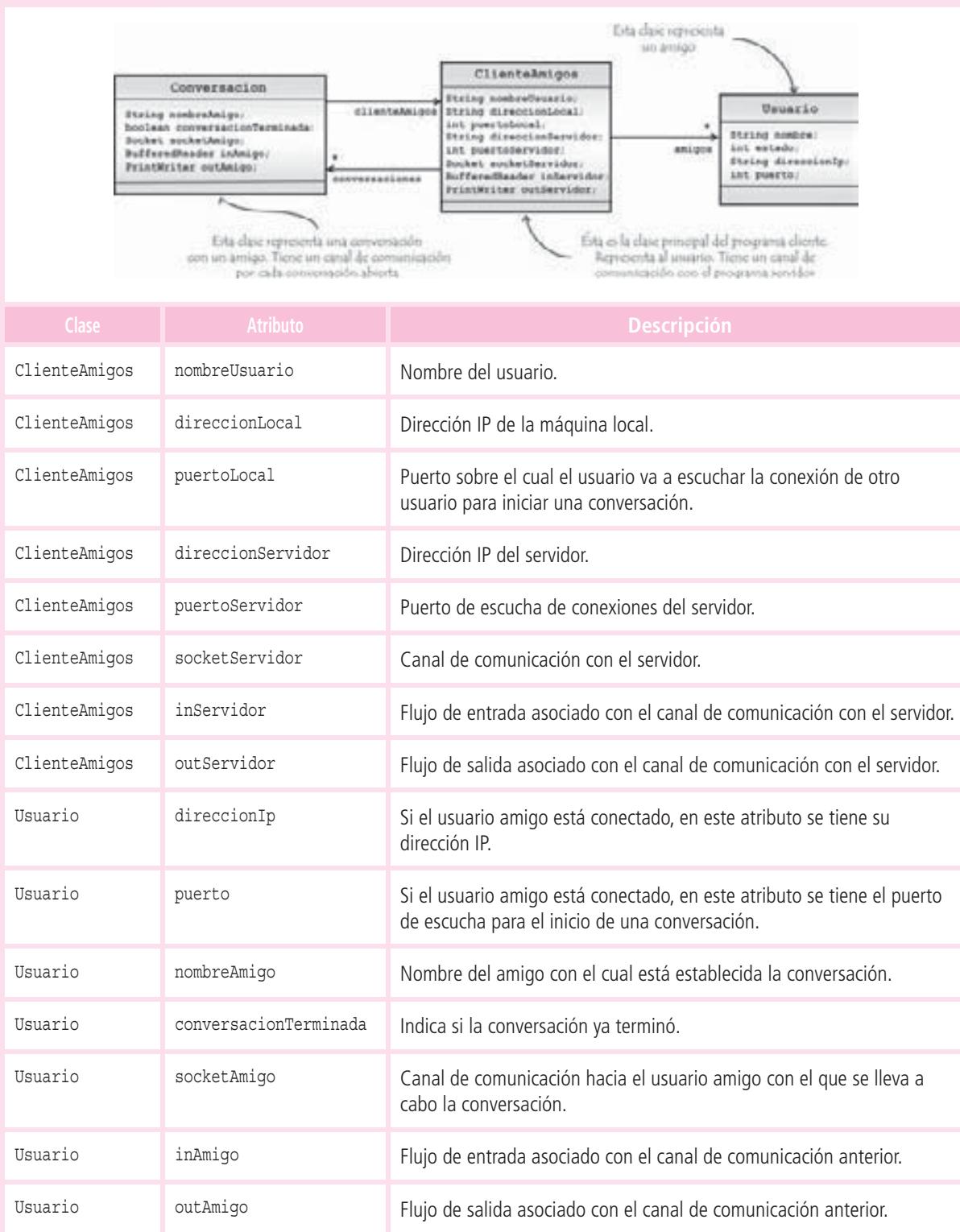
Requerimiento no funcional 1	Tipo: Persistencia Descripción:
Requerimiento no funcional 2	Tipo: Concurrencia Descripción:
Requerimiento no funcional 3	Tipo: Distribución Descripción:

Diagrama de clases del programa cliente. Estudie el siguiente diagrama de clases, asegurándose de entender el significado de cada uno de los atributos.



Clase	Atributo	Descripción
Usuario	nombre	Nombre bajo el cual el usuario se encuentra registrado en el programa.
Usuario	estado	Indica si el usuario se encuentra en ese momento conectado al sistema.

Diagrama de clases del programa cliente. Estudie el siguiente diagrama de clases, asegurándose de entender el significado de cada uno de los atributos.



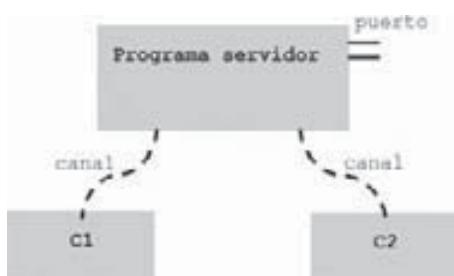
Arquitectura de comunicación. La siguiente secuencia de diagramas ilustra la manera como están conectados los programas cliente entre sí y con el servidor. Estudie cuidadosamente cada etapa, fijándose en los canales de comunicación que se utilizan. Sobre ellos se mostrarán más adelante los protocolos.



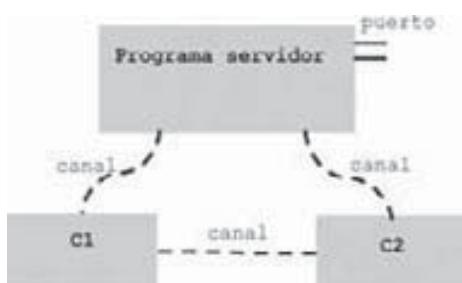
- El programa servidor inicia la ejecución. Se encuentra escuchando por un puerto las conexiones de los clientes.



- Un cliente C1 se conecta y se establece un canal de comunicación.

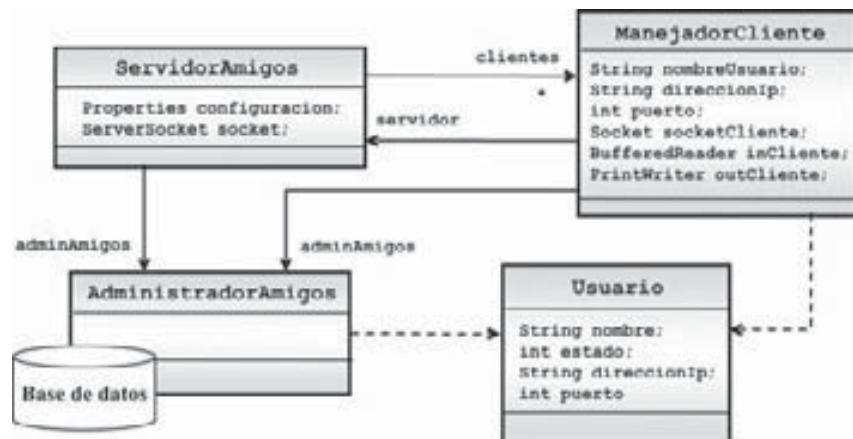


- Un segundo cliente C2 se conecta y se establece un canal de comunicación.



- El cliente C1 inicia una conversación con el cliente C2 y se crea un canal de comunicación entre ellos, por el cual van a enviarse los textos que se escriban.

Primer borrador del diagrama de clases del programa servidor. Estudie el siguiente diagrama, asegurándose de entender la responsabilidad de cada una de las entidades, al igual que el significado de cada atributo y asociación.

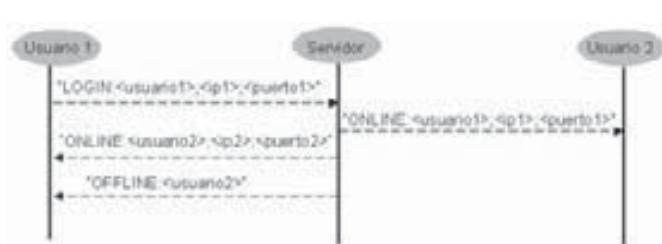


Clase	Atributo	Descripción
ServidorAmigos	configuración	Conjunto de propiedades que definen la configuración del programa.
ServidorAmigos	socket	Puerto de escucha para las conexiones de los clientes.
ManejadorCliente	nombreUsuario	Nombre del usuario al que está asociado este manejador.
ManejadorCliente	direccionIp	Dirección IP del usuario.
ManejadorCliente	puerto	Puerto de escucha del cliente para el inicio de una conversación.
ManejadorCliente	socketCliente	Canal de comunicación con el cliente.
ManejadorCliente	inCliente	Flujo de entrada asociado con el canal de comunicación anterior.
ManejadorCliente	outCliente	Flujo de salida asociado con el canal de comunicación anterior.
Usuario	direccionIp	Si el usuario está conectado, en este atributo se tiene su dirección IP.
Usuario	puerto	Si el usuario está conectado, en este atributo se tiene el puerto de escucha para el inicio de una conversación.

Diagrama de objetos. Construya un diagrama de objetos en el cual aparezcan el servidor y dos clientes que se encuentran en medio de una conversación. Utilice los diagramas de clases antes presentados.

Protocolos de comunicación. El programa tiene cinco protocolos, los cuales se muestran a continuación. Estudie cada uno de sus pasos.

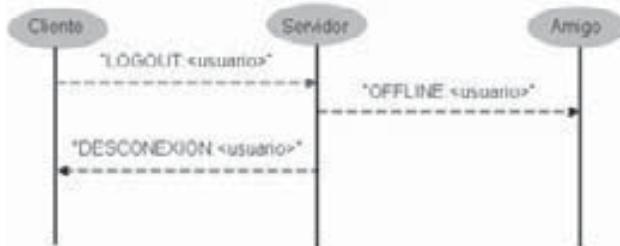
- Protocolo entre el servidor y los clientes conectados para indicar que ha llegado uno nuevo.
- En el primer mensaje que le envía el cliente al servidor le indica cuál es su nombre, la dirección IP de su máquina y el puerto en el que recibirá las conexiones de los otros clientes.
- El servidor envía un mensaje a las personas que conocen al usuario recién conectado, indicando que ahora está ONLINE. Luego, el servidor envía un mensaje al cliente recién conectado en el que indica el estado de cada uno de sus amigos: pueden ser mensajes ONLINE (para los que están conectados) y OFFLINE (para los que están desconectados).



- <usuario1> es el nombre del usuario que se está conectando.
- <ip1> es la dirección IP del usuario que se está conectando.
- <puerto1> es el puerto de escucha del usuario que se está conectando.
- <usuario2> es el nombre de un usuario conectado.
- <ip2> es la dirección IP de un usuario que ya está conectado.
- <puerto2> es el puerto de escucha de un usuario conectado.



Este protocolo se utiliza cuando un usuario identificado con el nombre <usuario> se quiere desconectar.



Cuando un cliente se va a desconectar, envía un mensaje de LOGOUT al servidor.



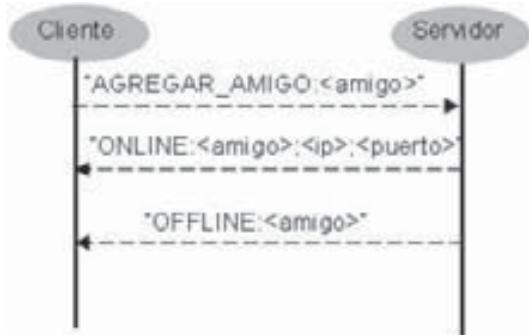
Este notifica su desconexión a todas las personas que lo conocen.



Luego, envía como respuesta al cliente un mensaje de DESCONEXION, que sirve como confirmación, para saber que todo el proceso se llevó a cabo sin problemas.



Este protocolo se utiliza cuando un cliente quiere agregar a un amigo a su lista.



El protocolo se inicia cuando se envía un mensaje AGREGAR_AMIGO al servidor.



Este recibe el mensaje, lo procesa y envía un mensaje en el cual informa sobre el estado actual del amigo (ONLINE u OFFLINE).



<amigo> representa el nombre del usuario que se quiere agregar.



<ip> es la dirección IP del amigo que se quiere agregar.



<puerto> es el puerto del amigo que se quiere agregar.



Este protocolo se utiliza cuando un cliente quiere iniciar una conversación con un amigo de su lista que está conectado.



Para iniciar una conversación un cliente envía un mensaje al servidor con el comando CONVERSACION y el nombre del <amigo> con el que quiere iniciar la conversación.



El servidor a su vez envía un mensaje al amigo, con el comando INICIO_CHARLA, y los datos del usuario al cual debe conectarse para establecer la conversación.



Una vez que los dos clientes se han conectado, se pueden enviar mensajes con el comando "MENSAJE:"



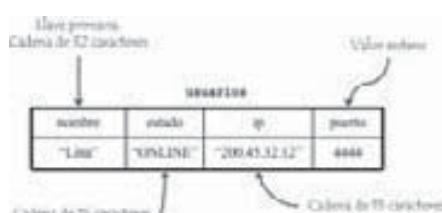
Este protocolo se utiliza cuando un cliente quiere terminar una conversación.



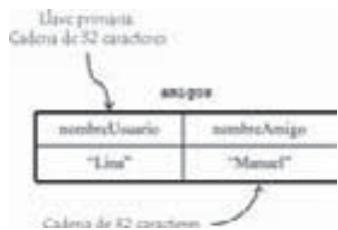
Cuando se quiere terminar la conversación, uno de los clientes debe enviar el mensaje TERMINAR.

El otro confirmará el fin de la conversación con un mensaje CONVERSACION_TERMINADA.

Protocolos de comunicación. El programa tiene cinco protocolos, los cuales se muestran a continuación. Estudie cada uno de sus pasos.



Instrucción SQL para la creación de la tabla:



Instrucción SQL para la creación de la tabla:

Instrucción SQL para agregar un usuario a la primera tabla:

Instrucción SQL para eliminar un usuario de la primera tabla:

Instrucción SQL para agregar un amigo de un usuario:

Instrucción SQL para eliminar a todos los amigos de un usuario:

Implementación. Cree el respectivo proyecto en Eclipse. Localice y edite la clase `ServidorAmigos`. Conteste las preguntas que se plantean a continuación.

Edite el archivo de configuración del programa servidor. ¿Qué información aparece allí?

En el método `recibirConexiones()`, ¿qué se hace cada vez que llega una nueva conexión? ¿Qué parámetros se pasan a la instancia que comienza el nuevo hilo de ejecución?

¿Qué responsabilidad tiene el método `enviarNotificacionAmigo()`? ¿Qué protocolo implementa?

¿Qué responsabilidad tiene el método `iniciarConversacion()`? ¿Qué protocolo implementa?

¿Qué responsabilidad tiene el método `desconectarCliente()`? ¿Qué protocolo implementa?

Implementación. Localice y edite la clase `AdministradorAmigos`. Conteste las preguntas que se plantean a continuación.

¿Qué método es el encargado de crear la base de datos si esta no existe? ¿Qué instrucción de SQL utiliza?

¿Qué responsabilidad tiene el método `crearUsuario()`? ¿Qué parámetros recibe? ¿Qué instrucción de SQL utiliza?

¿Qué responsabilidad tiene el método `agregarAmigo()`? ¿Qué parámetros recibe? ¿Qué instrucción de SQL utiliza?

¿Qué responsabilidad tiene el método `darAmigos()`? ¿Qué parámetros recibe? ¿Qué instrucción de SQL utiliza?

¿Qué responsabilidad tiene el método `darPersonasConocen()`? ¿Qué parámetros recibe? ¿Qué instrucción de SQL utiliza?

¿Qué responsabilidad tiene el método `existeUsuario()`? ¿Qué parámetros recibe? ¿Qué instrucción de SQL utiliza?

Implementación. Localice y edite la clase `ManejadorCliente`. Conteste las preguntas que se plantean a continuación.

1.

2.

3.

4.

5.

6.

Estudie el método de esta clase llamado `iniciarManejador()`. Explique los seis pasos que implementa el método. ¿Qué parte del protocolo implementa cada uno de ellos?

Implementación. Localice y edite la clase `ClienteAmigos`. Conteste las preguntas que se plantean a continuación.

¿Qué responsabilidad tiene el método `conectar()`? ¿Qué protocolo implementa? ¿Cómo hace para crear un nuevo hilo de ejecución que se encargue de recibir los mensajes del servidor?

1.

2.

3.

Estudie el método llamado `crearConversacionLocal()`. Explique los tres pasos que implementa el método. ¿Qué parte del protocolo implementa cada uno de ellos?

<p>Estudie el método llamado <code>conectarAConversacion()</code>. Explique los tres pasos que implementa el método. ¿Qué parte del protocolo implementa cada uno de ellos?</p>	<ol style="list-style-type: none">1.2.3.
---	--

Implementación. Localice y edite la clase `Conversacion`. Conteste las preguntas que se plantean a continuación.

<p>Estudie el primer constructor de la clase. ¿Qué parámetros recibe? ¿En qué casos se utiliza? ¿Para qué se crea una instancia de la clase <code>ServerSocket</code>? ¿Cómo se crea un nuevo hilo de ejecución para leer los mensajes enviados como parte de la conversación?</p>	
<p>Estudie el segundo constructor de la clase. ¿Qué parámetros recibe? ¿En qué casos se utiliza? ¿Cómo hace para crear el canal de comunicación?</p>	

Implementación. Localice y edite la clase Conversacion. Conteste las preguntas que se plantean a continuación.

1. Modifique el programa para que quede un registro (en una tabla de la base de datos) de la hora de conexión y desconexión de todos los usuarios. Esto va a permitir llevar un control de uso del programa. Agregue en la interfaz de usuario del administrador la posibilidad de consultar esta información.
2. Modifique el programa para que, cada vez que un usuario utilice una palabra de un conjunto definido por el administrador, se genere un nuevo registro en la base de datos, que indique el usuario, la palabra y la hora. Esto va a permitir llevar un control del vocabulario utilizado por los usuarios. Agregue en la interfaz de usuario del administrador la posibilidad de consultar esta información.
3. Modifique el programa para que quede un registro (en una tabla de la base de datos) de cada conversación que se realice. En dicho registro debe aparecer el usuario que la inició, el usuario que la aceptó, la hora de inicio, la hora de finalización y el número de mensajes enviado por cada uno. Agregue en la interfaz de usuario del administrador la posibilidad de consultar esta información.
4. Modifique el programa para que toda la comunicación entre los usuarios pase por el servidor. Esto implica cambiar la mayoría de los protocolos.

Anexo A

Tabla de Códigos UNICODE

1. Tabla de Códigos

La siguiente tabla muestra los principales caracteres UNICODE usados en Java, con su respectivo valor numérico.

33 !	51 3	69 E	87 W	105 i	123 {	175 -	193 Á	211 Ó	229 å	247 ÷
34 "	52 4	70 F	88 X	106 j	124	176 °	194 Â	212 Ô	230 æ	248 ø
35 #	53 5	71 G	89 Y	107 k	125 }	177 ±	195 Ã	213 Õ	231 ç	249 ù
36 \$	54 6	72 H	90 Z	108 l	126 ~	178 ²	196 Ä	214 Ö	232 è	250 ú
37 %	55 7	73 I	91 [109 m	161 í	179 ³	197 Å	215 ×	233 é	251 û
38 &	56 8	74 J	92 \	110 n	162 ¢	180 ´	198 Æ	216 Ø	234 ê	252 ü
39 '	57 9	75 K	93]	111 o	163 £	181 µ	199 Ç	217 Ù	235 ë	253 ý
40 (58 :	76 L	94 ^	112 p	164 ☒	182 ¶	200 È	218 Ú	236 ï	254 þ
41)	59 ;	77 M	95 _	113 q	165 ¥	183 ·	201 É	219 Û	237 í	255 ÿ
42 *	60 <	78 N	96 `	114 r	166 ¡	184 ,	202 Ê	220 Ü	238 î	338 €
43 +	61 =	79 O	97 a	115 s	167 §	185 ¹	203 Ë	221 Ý	239 ï	339 œ
44 ,	62 >	80 P	98 b	116 t	168 ¨	186 °	204 Ì	222 Þ	240 ð	352 Š
45 -	63 ?	81 Q	99 c	117 u	169 ®	187 »	205 Í	223 ß	241 ñ	353 š
46 .	64 @	82 R	100 d	118 v	170 ª	188 ¼	206 Î	224 à	242 ð	376 Ÿ
47 /	65 A	83 S	101 e	119 w	171 «	189 ½	207 Ï	225 á	243 ó	381 Ž
48 0	66 B	84 T	102 f	120 x	172 ¬	190 ¾	208 Đ	226 â	244 ô	382 ž
49 1	67 C	85 U	103 g	121 y	173 -	191 ¿	209 Ñ	227 ã	245 õ	402 f
50 2	68 D	86 V	104 h	122 z	174 ®	192 À	210 Ò	228 ä	246 ö	

