

Lógica y programación orientada a objetos: Un enfoque basado en problemas





Grupo de Investigación
en Ingeniería de Software
del Tecnológico de Antioquia

Proyecto Sismoo:
Sistema para el modelamiento por objetos

Investigadores principales:
RICARDO DE JESÚS BOTERO TABARES
CARLOS ARTURO CASTRO CASTRO
GABRIEL ENRIQUE TABORDA BLANDÓN

Coinvestigadores:
JUAN DAVID MAYA MONTOYA
MIGUEL ÁNGEL VALENCIA LÓPEZ

Ilustrador:
JAIME ROLDÁN ARIAS

ISBN: 978-958-8628-00-4

Contenido

PRESENTACIÓN.....	1
-------------------	---

Capítulo 1. Una didáctica para el aprendizaje de la programación

1.1. ALGUNAS CONSIDERACIONES PEDAGÓGICAS PARA EL APRENDIZAJE DE LA PROGRAMACIÓN.....	11
1.2. FASES PARA LA SOLUCIÓN DE UN PROBLEMA.....	14
1.3. TIPO DE PROBLEMAS A TRATAR	21
1.4. REFERENCIAS	24

Capítulo 2. Fundamentos de programación orientada a objetos

2.1. EL MUNDO DE LOS OBJETOS, EL MUNDO DE LAS COSAS	31
LOS CONCEPTOS DE OBJETO Y CLASE	32
PROBLEMA 1: HOLA MUNDO.....	37
CREACIÓN DE OBJETOS Y PASO DE MENSAJES.....	42
PROBLEMA 2: OBJETOS TIPO CÍRCULO.....	43
2.2. TIPOS DE DATOS PRIMITIVOS Y VARIABLES	47
2.3. OPERADORES Y EXPRESIONES	51

2.4.	CLASES DE USO COMÚN: UN MICRO MUNDO PARA EFECTOS DE REUTILIZACIÓN	58
	TIPOS DE DATOS ESTÁNDAR COMO OBJETOS	58
	LA CLASE ENTERO	
	LA CLASE REAL	
	LA CLASE CHARACTER	
	LA CLASE LOGICO	
	PAQUETES DE USO COMÚN.....	65
2.5.	CONFIGURACIÓN DE LAS CLASES DE USO COMÚN.....	67
	LA CLASE OBJETO	67
	LA CLASE TIPODE DATO	68
	LA CLASE FLUJO	69
	LA CLASE CADENA	70
	LA CLASE MAT	71
2.6.	PROBLEMAS RESUELTOS CON LAS CLASES DE USO COMÚN.....	73
	PROBLEMA 3: CLASES DE USO COMÚN	75
	PROBLEMA 4: CONVERSIÓN DE UNA CONSTANTE A DIFERENTES BASES NUMÉRICAS	79
2.7.	EJERCICIOS PROPUESTOS.....	82
2.8.	REFERENCIAS	84

Capítulo 3. Clases: tipos de datos abstractos

3.1.	ESTRUCTURA GENERAL DE UNA CLASE	91
3.2.	MÉTODOS	94
	DEFINICIÓN DE UN MÉTODO	95
	INVOCACIÓN DE UN MÉTODO	97
	MÉTODOS ESTÁTICOS	98
	PROBLEMA 5: OPERACIONES CON UN NÚMERO ENTERO.....	99
	PASO DE PARÁMETROS POR VALOR VS. PASO DE PARÁMETROS POR REFERENCIA	105
3.3.	SOBRECARGA DE MÉTODOS.....	106
	PROBLEMA 6: EMPLEADO CON MAYOR SALARIO.....	107
	PROBLEMA 7: INFORME SOBRECARGADO.....	112

3.4. ESTRUCTURAS DE CONTROL.....	117
LA ESTRUCTURA SECUENCIA	118
LA ESTRUCTURA SELECCIÓN	118
PROBLEMA 8: PRODUCTO MÁS CARO	121
PROBLEMA 9: ESTADÍSTICAS POR PROCEDENCIA	127
LA ESTRUCTURA ITERACIÓN	133
PROBLEMA 10: PRIMEROS CIENTO NATURALES.....	139
3.5. BANDERA O INTERRUPTOR.....	142
PROBLEMA 11: SUCESIÓN NUMÉRICA	142
3.6. MÉTODOS RECURSIVOS.....	145
PROBLEMA 12: FACTORIAL DE UN NÚMERO	146
PROBLEMA 13: CÁLCULO DE UN TÉRMINO DE FIBONACCI	149
3.7. EJERCICIOS PROPUESTOS.....	153
3.8. REFERENCIAS	156

Capítulo 4. Arreglos

4.1. OPERACIONES CON ARREGLOS	161
DECLARACIÓN DE UN ARREGLO	162
ASIGNACIÓN DE DATOS A UN ARREGLO	164
ACCESO A LOS ELEMENTOS DE UN ARREGLO	165
PROBLEMA 14: SUCESIÓN NUMÉRICA ALMACENADA EN UN VECTOR	165
4.2. LA CLASE VECTOR	168
PROBLEMA 15: UNIÓN DE DOS VECTORES.....	176
PROBLEMA 16: BÚSQUEDA BINARIA RECURSIVA	179
4.3. LA CLASE MATRIZ	182
PROBLEMA 17: PROCESO ELECTORAL	189
4.4. EJERCICIOS PROPUESTOS.....	196
4.5. REFERENCIAS	200

Capítulo 5. Relaciones entre clases

5.1. TIPOS DE RELACIÓN ENTRE CLASES	205
ASOCIACIÓN	206

DEPENDENCIA	209
GENERALIZACIÓN / ESPECIALIZACIÓN.....	209
AGREGACIÓN Y COMPOSICIÓN.....	211
REALIZACIÓN	212
PROBLEMA 18: SUMA DE DOS NÚMEROS.....	213
5.2. PAQUETES.....	219
PROBLEMA 19: VENTA DE PRODUCTOS.....	211
5.3. EJERCICIOS PROPUESTOS.....	227
5.4. REFERENCIAS	229

Capítulo 6. Mecanismos de herencia

6.1. HERENCIA	235
HERENCIA SIMPLE	236
HERENCIA MÚLTIPLE: INTERFACES.....	238
PROBLEMA 20: EMPLEADOS POR HORA Y A DESTAJO	240
6.2. POLIMORFISMO.....	246
6.3. EJERCICIOS PROPUESTOS.....	248
6.4. REFERENCIAS	251

Apéndices

A. ENTORNO INTEGRADO DE DESARROLLO SISMOO	252
B. ELEMENTOS SINTÁCTICOS DEL SEUDO LENGUAJE	260
C. GLOSARIO BÁSICO DE PROGRAMACIÓN ORIENTADA A OBJETOS.....	264

Índice analítico	271
------------------------	-----

Agradecimientos

Las sugerencias y aportes de estudiantes, egresados y profesores de la Facultad de Informática del Tecnológico de Antioquia fueron significativas durante la escritura de este libro. En particular, agradecemos a los estudiantes de Tecnología en Sistemas: Carlos Andrés García, Gloria Jazmín Hoyos, Steven Lotero, Emanuel Medina y Luis Gabriel Vanegas; a los Tecnólogos en Sistemas: Orlando Alarcón, Yeison Andrés Manco y Jader Rojas; a los ingenieros y profesores Eucario Parra, Gildardo Quintero, Darío Soto y Luis Emilio Velásquez; y al Comité para el Desarrollo de la Investigación del Tecnológico de Antioquia –CODEI–, en especial a la ex directora de Investigación y Posgrados, Amanda Toro, y a su actual director Jorge Ignacio Montoya.

Presentación

Lógica y programación orientada a objetos: Un enfoque basado en problemas, es el fundamento teórico y práctico para un primer curso de programación. Además de incursionar de manera directa en el aprendizaje de un lenguaje que soporta el paradigma orientado a objetos¹, incluye ejercicios de aplicación de los conceptos propios del paradigma, a saber: clase, objeto, encapsulación, paquete y herencia, que conllevan a otros como atributo, método, visibilidad, constructor, estado de un objeto, recolector de basura, ligadura estática y ligadura dinámica. De esta manera, el libro responde a propósitos de renovación pedagógica orientados al diseño de currículos “con base en la investigación, que promueven la calidad de los procesos educativos y la permanencia de los estudiantes en el sistema”, uno de los desafíos de la educación en Colombia propuesto en el Plan Nacional Decenal de Educación vigente [PNDE2006]. También responde a la estructura curricular del módulo *Desarrollar Pensamiento Analítico Sistémico I* del proyecto Alianza Futuro Digital Medellín, del plan de estudios 72 del programa Tecnología en Sistemas del Tecnológico de Antioquia - Institución Universitaria.

1 El paradigma orientado a objetos es un modelo o patrón para la construcción de software, entre otros existentes como los paradigmas imperativo, funcional y lógico.

Conviene señalar que este libro es el resultado del proyecto Sismoo – Sistema para el modelamiento por objetos –, adelantado en la línea de investigación “Ingeniería de software y sistemas de información”, inscrita en el grupo GIISTA – Grupo de Investigación en Ingeniería de Software del Tecnológico de Antioquia–, clasificado a 2009 en la categoría C de Colciencias. El proyecto Sismoo fue también una consecuencia de otro proyecto de la línea: MIPS00 – Método Integrado de Programación Secuencial y programación Orientada a Objetos para el análisis, diseño y elaboración de algoritmos–, que incluye un seudo lenguaje y una didáctica para el aprendizaje de la programación. Ahora bien, el proyecto Sismoo aporta un intérprete del seudo lenguaje propuesto por el MIPS00, sencillo y fácil de utilizar, que busca agilizar los procesos de enseñanza y aprendizaje de la programación de computadoras, e incluye un traductor al lenguaje de programación Java. Se proyecta seguir mejorando este objeto de aprendizaje.

El método para el aprendizaje y la enseñanza de la programación orientada a objetos del proyecto MIPS00 incluye elementos didácticos del *aprendizaje basado en problemas* y principios pedagógicos constructivistas. Su didáctica propone el seguimiento de cuatro fases para la solución de problemas:

- Definición de la tabla de requerimientos.
- Diseño del diagrama de clases.
- Definición de las responsabilidades de las clases.
- Desarrollo del seudo código (tiene una gran similitud con los lenguajes de programación Java, C# y Visual Basic.Net)

Como complemento de MIPS00, Sismoo es un proyecto de investigación aplicada cuyo producto de desarrollo es una herramienta de software diseñada para que el usuario pueda editar, compilar y ejecutar los problemas diseñados empleando el seudo lenguaje propuesto por MIPS00. Esta herramienta, denominada en el medio informático *Entorno Integrado de Desarrollo o IDE* (Integrated Development Environment), fue desarrollada en lenguaje Java y su diagrama estructural se asemeja al de cualquier compilador (ver Figura 1):

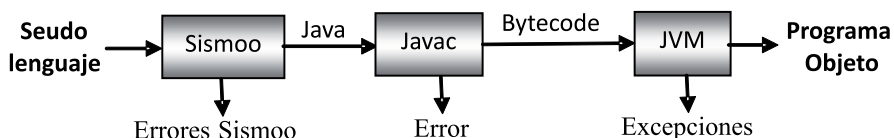


Figura 1. IDE traductor SISMOO

Conocidos los objetivos y antecedentes de este libro, examinemos brevemente las características de sus lectores y capitulación.

Como todo texto académico, este libro presupone dos lectores principales: estudiantes y profesores, en particular, de tecnología e ingeniería de sistemas y áreas afines; también admite otros lectores con conocimientos básicos en programación estructurada o imperativa.

Los profesores encontrarán en el capítulo 1 (*Una didáctica para el aprendizaje de la programación*) y en el apéndice B (*Elementos sintácticos del pseudo lenguaje*), claves para el diseño de problemas propios de la programación; sin embargo se recomienda a los estudiantes la lectura de los apartados 1.2 y 1.3., donde pueden enterarse de las fases para la solución de los problemas y los tipos de problemas, que encontrará en este libro. Los capítulos restantes desarrollan teóricamente el paradigma orientado a objetos² y presentan la didáctica para el aprendizaje de los fundamentos de programación.

Continuando con la descripción de los capítulos, en el número 2, (*Fundamentos de Programación Orientada a Objetos*), se exponen los fundamentos de la programación orientada a objetos, por ello su lectura es primordial para el estudiante, quien adquiere las bases para abordar la lectura de los próximos capítulos. El capítulo 3, (*Clases: tipos de datos abstractos*), profundiza lo concerniente al manejo de métodos e introduce el empleo de las sentencias de control y los interruptores; aquí el estudio de la recursión es importante, en tanto puede retomarse para un curso de estructuras de datos. El capítulo 4, (*Arreglos*), es una incursión a las clases contenedoras lineales Vector y Matriz (clases que almacenan conjuntos de objetos); es también un abre bocas a un módulo

² Varios profesores nos referimos a él como “paradigma objetual”

o curso relacionado con estructuras de datos donde se puede profundizar mucho más. En los capítulos 5 (*Relaciones entre clases*) y 6 (*Mecanismos de herencia*) se tratan las relaciones fundamentales entre clases, como las asociaciones y dependencias, y los paquetes.

Una vez revisado el contenido del libro, quien tenga conocimientos del paradigma objetual, sea estudiante o profesor, y desee interactuar de entrada con un micro mundo de clases reutilizables (una visión bastante reducida de algo como la API de Java o los espacios de nombres de .Net Framework), puede comenzar con la lectura de los capítulos 2 y 3, y pasar a los capítulos finales, 5 y 6.

Se recomienda a un lector casual empezar a estudiar las historias que preceden cada capítulo, cuentos ilustrados y representados con un diagrama de clases, en consonancia con un artículo publicado por Zapata [Zapata1998].

Desde ahora es conveniente la lectura de los apéndices A (*Entorno Integrado de Desarrollo Sismoo*), donde se exponen los elementos necesarios para interactuar con el entorno integrado de desarrollo Sismoo; B (*Elementos sintácticos del seudo lenguaje*) que presenta un listado de palabras reservadas para el seudo lenguaje objetual, un resumen de las estructuras de control y los elementos estructurales que permiten definir una clase, un método, un paquete y una interfaz; y C (*Glosario básico de programación orientada a objetos*), que presenta un listado de términos con su significado para consulta rápida en el proceso de aprendizaje de la programación.

A la vez, durante la lectura de este libro, puede ser consultado su índice analítico, el cual contiene términos de relevancia que a nivel conceptual, deben ser claros para plantear soluciones concretas a los problemas sugeridos; en el libro, estos conceptos se escribieron en *cursiva* para facilitar su identificación en las páginas que señala el índice. El CD anexo incluye el intérprete Sismoo, además el JDK6.7 con uno de sus entornos (NetBeans 6.1) y el Microsoft .Net Framework 2.0 para el trabajo con el entorno SharpDevelop 2.2.1.

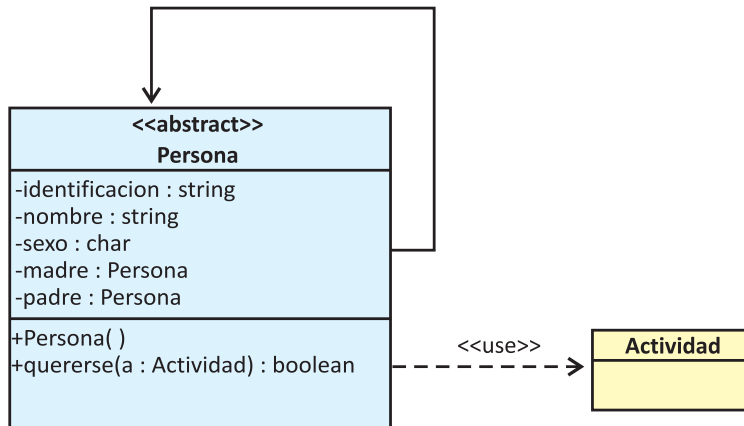
En la siguiente página el lector encontrará una frase célebre como pretexto a la familiarización con los diagramas de clase. Esperamos que este libro sea un disfrute para los estudiantes de programación y para sus profesores.

Los autores

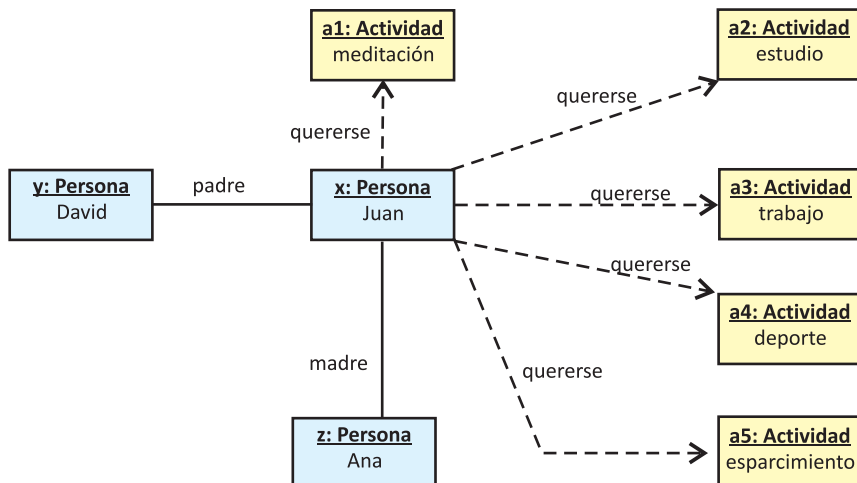
Quererse a sí mismo es el principio de un romance para toda la vida

Oscar Wilde

Diagrama de clases propuesto:



Un diagrama de objetos:



CAPÍTULO 1

Una didáctica para el aprendizaje de la programación

- 1.1. Algunas consideraciones pedagógicas para el aprendizaje de la programación
 - 1.2. Fases para la solución de un problema
 - 1.3. Tipo de problemas a tratar
 - 1.4. Referencias
-

TARARÍ Y TARARÁ

De: Alicia a través del espejo
Lewis Carroll

- Ahora está soñando -señaló Tarará- ¿y a que no sabes lo que está soñando?
- ¡Vaya uno a saber -replicó Alicia- ¡Eso no podría adivinarlo nadie!
- ¡Anda! ¡Pues si te está soñando a ti! -exclamó Tarará batiendo palmas en aplauso de su triunfo-. Y si dejara de soñar contigo, ¿qué crees que te pasaría?
- Pues que seguiría aquí tan tranquila, por supuesto -respondió Alicia.
- ¡Ya! ¡Eso es lo que tú quisieras! -replicó Tarará con gran suficiencia-. ¡No estarías en ninguna parte! ¡Como que tú no eres más que un algo con lo que está soñando!
- Si este Rey aquí se nos despertara -añadió Tarará- tú te apagarías... ¡zas! ¡Como una vela!



Diagrama de clases:

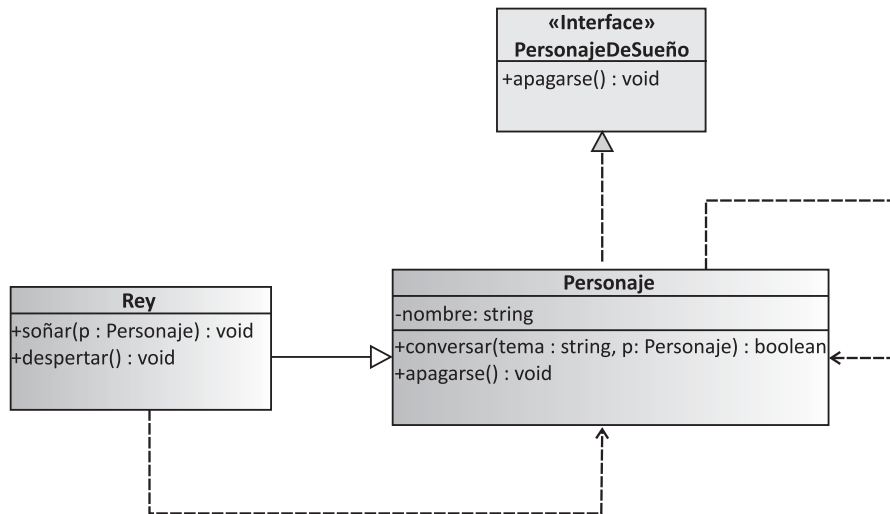
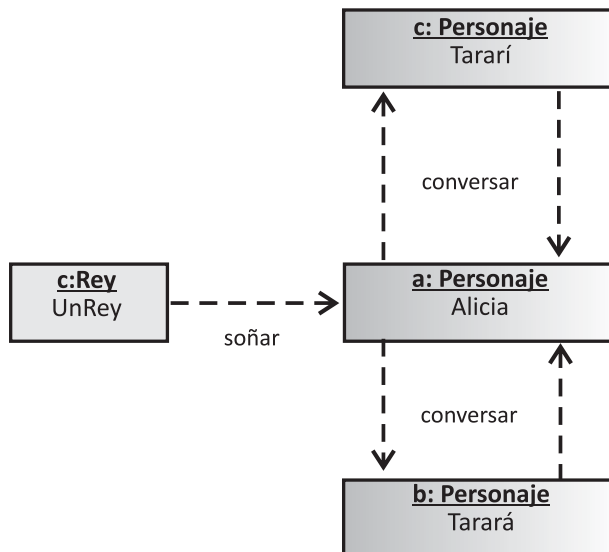


Diagrama de objetos:



Objetivo de aprendizaje

- Definir una didáctica para la enseñanza y aprendizaje de la programación orientada a objetos.

Para alcanzar este objetivo, consideramos que la didáctica de un profesor para enseñar la programación de computadores debe sustentarse en metodologías inspiradas en la ingeniería de sistemas, las tecnologías dominantes de la informática, la aproximación a problemas reales, el rigor abstracto y lógico y la puesta en común entre las necesidades expresadas por el sector informático donde se van a desenvolver sus estudiantes como profesionales.

Es por lo anterior que este capítulo tiene por objetivo establecer algunos lineamientos didácticos para el aprendizaje de la programación, a la vez que define el tipo de problemas a tratar y las etapas para su solución.

1.1. ALGUNAS CONSIDERACIONES PEDAGÓGICAS PARA EL APRENDIZAJE DE LA PROGRAMACIÓN

Aprender a programar computadores desde un enfoque basado en la solución de problemas, involucra estrategias de enseñanza-aprendizaje constructivistas. Los aprendizajes por descubrimiento, colaborativo, significativo y basado en problemas o situado, son aplicados por los estudiantes de programación en mayor o menor grado durante los diferentes estadios del proceso cognitivo. El profesor debe ser consciente de la pedagogía latente en el proceso de formación para propiciar ambientes donde estos aprendizajes se relacionen, porque de alguna manera todos ellos se complementan.

En el *aprendizaje por descubrimiento* [Bruner1966] la nueva información o conocimiento se adquiere a través de los propios esfuerzos del estudiante (tiempo de trabajo independiente), con los aportes del aprendizaje por exposición o instrucción

impartido por el docente (horas de trabajo presencial). Los contenidos desarrollados deben ser percibidos por el aprehendiente como un conjunto de problemas a tratar y relaciones conceptuales a establecer. En el *aprendizaje colaborativo* [Zañartu2005], el estudiante puede también ayudarse de los conocimientos de sus compañeros o de comunidades externas, por medio de encuentros presenciales extra clase o con actividades propiciadas por las tecnologías de la información y comunicación. En el *aprendizaje significativo* [Ausubel1983], el aprendizaje del alumno depende de la estructura cognitiva previa que se relaciona con la nueva información, de tal manera que el nuevo conocimiento adquiere significado y puede enlazarse en nuevos procesos cognitivos.

En el *aprendizaje basado en problemas* [González2006] confluyen algunas características de los aprendizajes anteriores: el estudiante descubre por sí mismo, interactúa con los demás, adquiere conocimientos con significado en la medida que estos se relacionan con los ya establecidos y sientan la base para otros venideros. En el aprendizaje basado en problemas un grupo de alumnos se reúne, con la facilitación de un tutor, para analizar y resolver un problema seleccionado o diseñado especialmente para el logro de ciertos objetivos de aprendizaje.

Las estrategias de aprendizaje expuestas se posibilitan y consolidan en el aula³ con las siguientes actividades:

- Exposiciones del profesor para explicar los aspectos teóricos más relevantes, a la vez que soluciona problemas concretos en clase.
- Solución de talleres por el estudiante, quien se hace cargo de su propio proceso de aprendizaje al retroalimentar los conceptos vistos en clase y solucionar dudas con la colaboración de otros aprendices y el apoyo del tutor. En este punto adquieren especial interés las actividades desarrolladas por el aprendiz en su tiempo de trabajo independiente, en concreto las relacionadas con la transcripción y prueba del pseudo código con IDE Sismoo, la codificación de las soluciones en lenguajes comerciales orientados a objetos, la relación con otras instituciones o Grupos de Investigación interesados en el desarrollo de software y la propuesta de nuevos problemas que conlleven a la creación de objetos de aprendizaje innovadores.

3 Se entiende por *aula* cualquier entorno propicio para el aprendizaje como un salón de clase, una biblioteca, un espacio natural, un teatro, un ambiente virtual, entre otros.

- Desarrollo de un proyecto integrador por parte del estudiante para aplicar todos o un gran porcentaje de los conceptos aprendidos.

De esta forma, los siguientes criterios pueden servir de base para la incursión didáctica apropiada en los cursos de fundamentos de programación:

- La clase es constructiva y tiene el liderazgo del docente para incentivar el aprendizaje del estudiante.
- Desde las primeras incursiones en la clase, las herramientas de programación apoyan el desarrollo de la lógica, respetando los momentos necesarios para la consolidación conceptual y las prácticas en el computador por parte del estudiante.
- La autoevaluación del estudiante y su descubrimiento autónomo, son factores importantes en el proceso de aprendizaje. En este sentido, el mayor nivel se alcanza cuando el mismo estudiante descubre los puntos críticos de sus algoritmos y programas.
- El desarrollo a partir de problemas y casos hacen de la clase un espacio para aprendizajes contextualizados y significativos, en la medida que se proporcione a los estudiantes información acerca de dominios económicos, administrativos, financieros y logísticos, relacionados con la cotidianidad de los entornos productivos de las empresas.
- El paradigma de programación orientado a objetos debe concebirse como un proceso. Esto lleva a considerar que la abstracción de objetos y *comportamientos* debe hacer parte de la modelación desde los inicios del proceso de aprendizaje, esto es, debe incluirse en los cursos de lógica de programación y estructuras de datos. Así se garantiza su debido tratamiento en otras asignaturas relacionadas con lenguajes de programación, bases de datos e ingeniería de software.
- La actitud recursiva del estudiante es otro factor de éxito en la formulación de su proyecto, así como en la interacción con las plataformas de desarrollo de software durante la ejecución del mismo.
- Una idea transversal debe marcar el norte de los cursos de programación: estructurar los contenidos por problemas, en módulos que se comunican y que permiten concebir sus soluciones como la integración de partes. Esto debe aportarle a los estudiantes las bases para resolver problemas cuando estén por fuera de la clase.

- Otro de los principios didácticos que todo docente, no sólo de programación, debe tener en cuenta a la hora de diseñar sus clases, es que el aprendizaje de los contenidos de una asignatura implica a sus estudiantes conflictos cognitivos que varían según sus habilidades. Por esta razón, los docentes deben motivar a sus estudiantes con actividades que impliquen un aprendizaje que trascienda lo individual como el *aprendizaje colaborativo*; y que se dirija hacia el apoyo e interrelación con sus compañeros: el aprendizaje basado en la solución de problemas.

Resumiendo, un enfoque constructivista como el que aquí se propone, se apoya de variadas estrategias de enseñanza-aprendizaje: retoma del *aprendizaje por descubrimiento*, los esfuerzos que debe hacer el estudiante para reconocer una nueva información; y del *aprendizaje colaborativo*, el que un estudiante es capaz de cuestionar y compartir sus conocimientos con los colegas de su grupo o con comunidades externas contactadas a través de la Web. A su vez, del *aprendizaje significativo* valora la estructura cognitiva previa del estudiante para vincular lo ya conocido con la nueva información; y por último, reconoce del *aprendizaje basado en problemas* (que conlleva características de los aprendizajes ya citados), lo siguiente: la adquisición de conocimientos como el desarrollo de habilidades y actitudes hacen parte del entorno de la clase; un espacio donde un grupo pequeño de alumnos se reúne con la facilitación de un tutor a analizar y resolver un problema seleccionado o diseñado especialmente para el logro de ciertos objetivos de aprendizaje propuestos en el curso.

En la próxima sección se explican con detalle las fases necesarias para la solución de un problema en el curso de programación orientada a objetos que en este libro se presenta.

1.2. FASES PARA LA SOLUCIÓN DE UN PROBLEMA

La solución de un problema conlleva una didáctica de cuatro fases:

- a) Construcción de la tabla de requerimientos.
- b) *Abstracción de clases* (diagramas conceptuales – si se requiere- y de clases

según formalismo de *UML*⁴).

- c) Descripción de las responsabilidades de las clases, formalizadas con los *contratos* de cada método.
- d) Escritura de *seudo código orientado a objetos*.

Veamos cada fase con más detalle, aplicadas al siguiente problema: Visualizar la suma de dos números enteros ingresados por el usuario.

a) Construcción de la tabla de requerimientos

La *construcción de la tabla de requerimientos* [Villalobos2006] forma parte del análisis del problema. Los requerimientos hacen referencia a las necesidades de los usuarios, es decir, identifican los aspectos que los usuarios del programa desean resolver mediante software. Estos requerimientos se denominan *funcionales* al sostener una relación directa con la funcionalidad del sistema.

La tabla de requerimientos está compuesta por cuatro columnas:

- **Identificación del requerimiento:** es un código que identifica al requerimiento, generalmente compuesto por una letra y un dígito. Identificadores comunes para los requerimientos son R1, R2, R3, etc.
- **Descripción:** consiste en una descripción concisa y clara, en lenguaje natural, del requerimiento.
- **Entradas:** son los insumos o datos necesarios para que el requerimiento se pueda suplir con éxito.
- **Resultados o salidas:** constituyen el cumplimiento del requerimiento, es decir, son los resultados que dan solución a un requerimiento funcional definido por el usuario.

Apliquemos en el Ejemplo 1.1., la tabla de requerimientos para la suma de dos números enteros.

4 UML (Unified Modeling Language) es un lenguaje para construir planos de software.

Ejemplo 1.1. Tabla de requerimientos

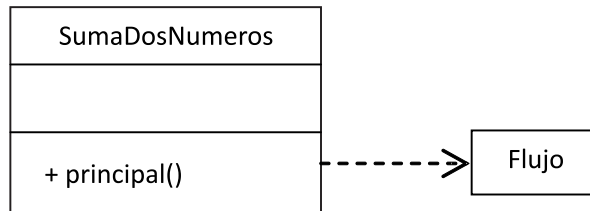
Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Capturar dos números enteros	Dos números enteros ingresados por el usuario	Dos números enteros almacenados en memoria (variables n1 y n2)
R2	Operar dos números enteros	Las variables n1 y n2	La suma de los números n1 y n2

Observaciones a tener en cuenta para el diseño de tablas de requerimientos:

- Es común que la salida de un requerimiento se convierta en la entrada de otro; por ejemplo, la salida del requerimiento R1 (las variables n1 y n2) es la entrada del requerimiento R2.
- Los requerimientos funcionales son *casos de uso* que describen de una manera detallada el comportamiento del sistema con los distintos actores que interactúan en él. Un caso de uso es la descripción de una secuencia de acciones que un sistema ejecuta y que produce un resultado observable de interés para un actor particular [Booch1999]. Un actor es un usuario del sistema.

b) Abstracción de clases

La *abstracción de clases* también forma parte del análisis del problema y es el primer momento para diseñar la solución. Consiste en una representación gráfica del problema, plano de software, donde se dibujan abstracciones de la realidad relacionadas con el mundo del problema, modelables con software. El plano construido se puede presentar en dos fases, que comprenden un *diagrama conceptual* y un *diagrama de clases*. Es de aclarar que el primero de ellos es opcional en tanto no se requiere cuando los problemas a tratar son pequeños o con cierto margen de trivialidad: mostrar un mensaje, comparar dos cadenas, hallar una sumatoria, entre otros.

Ejemplo 1.2: Diagrama de clases para la suma de dos números.

En este diagrama la solución se presenta en dos clases: **SumaDosNumeros** que contiene un método denominado *principal()*, y **Flujo**, clase de uso común que se estudiará en el apartado 2.5 (*Configuración de las clases de uso común*).

El rectángulo que representa la clase se divide en tres partes: en la primera se coloca su nombre: **SumaDosNumeros**; en la segunda se ubican los *atributos* de la clase (no existen para este caso), y en la tercera las *operaciones* o *métodos*: el único método presentado en el ejemplo 1.2 es *principal()*. Si el lector requiere una mayor información sobre la estructura de una clase, puede encontrarla en la sesión 2.1 (*El mundo de los objetos, el mundo de las cosas*).

c) Análisis de responsabilidades de las clases

El análisis de *responsabilidades de las clases* conlleva la descripción de los métodos de cada clase mediante *contratos* que incluyen los requerimientos asociados, la *precondición* o estado del objeto antes de ejecutar el método, la *postcondición* que aclara el estado del objeto luego de ejecutar el método, y el *modelo verbal* que consiste en una descripción en lenguaje natural de la solución planteada, algo similar al denominado *algoritmo cualitativo*.

La identificación de responsabilidades forma parte de la *documentación* de la solución o del futuro sistema basado en software.

Ejemplo 1.3. Contrato de la clase SumaDosNumeros

Nombre del método	Requerimientos asociados	Precondición	Postcondición	Modelo verbal
principal ()	R1 R2	Se desconocen los números a sumar	Se ha calculado y visualizado la suma de dos números enteros	1. Declarar variables 2. Capturar los dos números enteros 3. Sumar los números 4. Visualizar el resultado de la suma

d) Escritura del pseudo código orientado a objetos

El *seudo código orientado a objetos (OO)* especifica la solución del problema, porque da cuenta del cómo obtener una solución. Se escribe de una manera similar al proceso de codificación en un lenguaje de programación como Java o C#.

Esta fase conlleva la aplicación de pruebas manuales para cada uno de los métodos (correspondientes a las denominadas “pruebas de escritorio”), o de manera automática con el traductor Sismoo.

Ejemplo 1.4. Pseudo código OO para la suma de dos números

```

1.  clase SumaDosNumeros
2.      publico estatico vacio principal ( )
3.          entero a, b, c
4.          Flujo.imprimir ("Ingrese el primer número entero:")
5.          a = Flujo.leerEntero ( )
6.          Flujo.imprimir ("Ingrese el segundo número entero:")
7.          b = Flujo.leerEntero ( )
8.          c = a + b
9.          Flujo.imprimir (a + " + " + b + " = " + c)
10.     fin_metodo
11. fin_clase

```

Observaciones:

- Para facilitar las explicaciones, en este ejemplo se han numerado las diferentes instrucciones del algoritmo objetual.
- Las *palabras clave* (o *reservadas*) del pseudocódigo orientado a objetos se escribirán en este libro siempre sin acentos y en **negrita**, con el objetivo diferenciarlas de los identificadores usados por el analista para designar clases, variables, constantes, atributos, métodos y objetos. Las palabras reservadas del ejemplo son **clase**, **publico**, **estatico**, **vacio**, **principal**, **entero**, **Flujo**, **fin_metodo** y **fin_clase**. Para una mayor información acerca de las palabras reservadas del pseudo lenguaje orientado a objetos utilizado en este libro, se recomienda revisar el apéndice B.
- Las variables *a*, *b* y *c* son declaradas enteras, como se indica en la instrucción 3. Una buena práctica de programación es inicializar las variables cuando se declaran, para evitar algunos errores en tiempo de ejecución. En lenguajes como Java, C# y Visual Basic.Net se pueden presentar este tipo de situaciones con algunas instrucciones; sin embargo, para evitar contratiempos en la ejecución, toda declaración de variables en estos lenguajes implica su inicialización con valores preestablecidos. Por ejemplo, las variables de tipo numérico son puestas a cero, las cadenas en la cadena vacía y los objetos son inicializados en la constante nula.

En la siguiente tabla se observan los valores con los cuales se deben comenzar los distintos tipos de variables.

Tabla 1.1. Valores de inicialización por omisión para los tipos estándar de datos

Tipo	Valor con el que se debe inicializar	Valor de inicialización por omisión (con descripción)
Númerica (entero, real)	Cualquier número (depende del problema)	0 (cero)
carácter	Cualquier carácter	' ' (espacio)
cadena	Cualquier cadena	" " (cadena vacía)
lógico	falso o cierto	cierto (verdadero)
Objeto	nulo	nulo

Observaciones:

- Para la entrada de datos enteros se utiliza la clase **Flujo** y el *método estático* `leerEntero()`, como en la instrucciones 5 y 7 del seudo código 00 para la suma de dos números enteros:

`a = Flujo.leerEntero ()`

`b = Flujo.leerEntero ()`

La *clase* **Flujo** se encuentra definida en el *paquete* o *espacio de nombres* importado por defecto, denominado **sistema**. Una vista parcial de este paquete se ilustra en la figura 1.2.

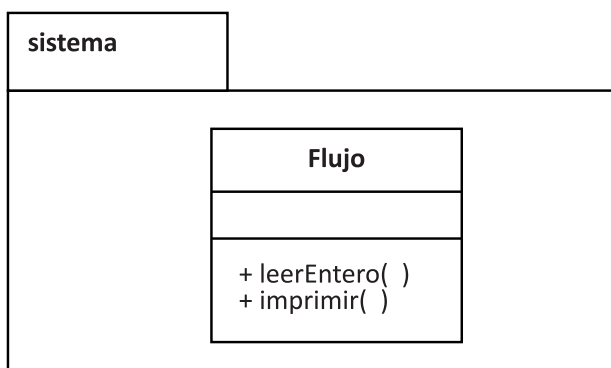


Figura 1.2. Diagrama de clases UML

- Para imprimir constantes y variables en el dispositivo de salida estándar (el monitor o pantalla), se ha utilizado el método **Flujo.imprimir()**, como en las instrucciones 4, 6 y 9.
- Las instrucciones 4 y 6, dadas por **Flujo.imprimir** (“Ingrese el primer número entero:”) y **Flujo.imprimir** (“Ingrese el segundo número entero:”), imprimen cada una su respectiva constante de cadena.
- La instrucción 9 tiene un matiz especial:

Flujo.imprimir (a + “ + ” + b + “ = ” + c) imprime una combinación de

variables y de constantes tipo carácter. Por ejemplo, si las variables *a* y *b* contienen los valores 7 y 10 respectivamente, la instrucción visualizará la expresión:

$$7 + 10 = 17$$

En la instrucción 9 puede notarse la diferencia entre el símbolo cruz (+) no encerrado entre comillas y el símbolo cruz que sí las tiene: en el primer caso se trata del operador de concatenación, mientras que en el segundo se trata de la constante “+” que se visualiza tal cual aparece en la expresión. De igual manera, en la instrucción 8 ($c = a + b$) el símbolo cruz denota al operador aritmético de adición. Se dice entonces que el operador cruz (+) se encuentra *sobrecargado*, porque dependiendo del lugar donde se utilice denota suma de números o concatenación de valores.

- Se puede omitir el uso de la variable *c* haciendo visible el resultado de evaluar una expresión aritmética de la siguiente manera:

Flujo.imprimir (a + “ + ” + b + “ = ” + (a + b))

- Otras alternativas de solución para la suma de dos números, se presentan en el problema 18 del apartado 5.1 (*Tipos de relación entre clases*).

1.3. TIPO DE PROBLEMAS A TRATAR

En la enseñanza y el aprendizaje de la programación es necesario plantear problemas reales o ficticios, claros en su descripción y susceptibles de solucionar con un programa de computadora.⁵

⁵ Es de aclarar que los problemas no determinísticos, denominados también NP completos, imposibles hasta el momento de solucionar con un programa de computadora, no serán tratados en este libro.

A continuación, se presentan los cuatro tipos de problemas, ordenados según su volumen y complejidad, y en los que será posible la aplicación de clases de uso común.

a) Problemas donde se reutilizan las clases de uso común

Aunque los cuatro tipos de problemas elegidos reutilizan las clases de uso común, en esta primera categoría se hace énfasis en ellos. Se caracterizan porque presentan soluciones donde es necesario utilizar componentes con una funcionalidad preestablecida, probada y verificada, esto es, donde el funcionamiento interior de los métodos pasa a un segundo plano. De esta manera, lo que interesa es reutilizar la interfaz pública de un pequeño conjunto de clases requeridas en todas o en una gran parte de las aplicaciones, lo cual permite simular las acciones realmente ejecutadas por los programadores cuando trabajan con plataformas de desarrollo comerciales como J2SE o .NET. Por simple que sea el problema a resolver, los programadores siempre reutilizan algún componente de software predefinido.

Las *clases de uso común* que se emplean para las explicaciones de este texto se encuentran almacenadas en el paquete *sistema*; algunas de ellas son **Flujo** (controla la entrada y salida de datos estándar), **Cadena** (maneja textos o cadenas de caracteres), **Entero** (maneja cantidades enteras), **Mat** (contiene funciones matemáticas), entre otras. Para mayor información remitirse a la sesión 2.4. *Clases de uso común: un micro mundo para efectos de reutilización*.

b) Problemas con una clase del ámbito de la solución (Proyecto) y métodos analizadores estáticos (opcionales)

En este tipo de problemas se usan clases vacías, es decir, clases que carecen de atributos, contienen uno o varios métodos analizadores (uno de ellos de carácter obligatorio denominado *principal*), donde la solución se plantea sin la creación de objetos.

En realidad nunca se tendrán clases sin atributos. Lo que se persigue con este tipo de problemas es aprender a manejar las sentencias de control y los subprogramas (denominados en adelante *métodos*), conceptos propios de la programación

imperativa. De esta manera el método de aprendizaje aplicado es mixto, es decir, se fundamenta en la programación orientada a objetos y maneja conceptos de la programación procedimental como una fase complementaria a la modelación por objetos.

Los métodos *analizadores* utilizados son de naturaleza estática, y conducen a la solución del problema con la estrategia de diseño “*Divide y vencerás*”: división de un problema en otros más pequeños y sencillos de solucionar.

El uso de la *clase Flujo* es inamovible, porque en la mayoría de las situaciones problemáticas se emitirán mensajes al usuario a través del método `imprimir()` o se ingresarán datos por medio de los métodos `leerEntero()` y similares.

c) Problemas con varias clases asociadas entre sí

En este tipo de problemas es posible trabajar con propiedad la teoría de la orientación a objetos, en tanto se aplican los conceptos de encapsulación de datos, métodos constructores, métodos modificadores, métodos analizadores, creación de objetos y paso de mensajes.

Aquí cobra relevancia el concepto de *asociación* entre dos *clases*, las cuales se producen a partir del ámbito del problema, es decir, relacionadas directamente con las percepciones del usuario. *Abstracciones* como Empleado, Producto, Viaje y TransacciónBancaria, se consolidan en *objetos* dentro de la clase del ámbito de la solución por excelencia, denominada en general *Proyecto* (puede adquirir otra nominación según la preferencia del analista). Por último, es necesario resaltar que en este tipo de problemas las clases asociadas con una funcionalidad común se pueden agrupar en un *paquete*.

d) Problemas cuya solución involucra mecanismos de herencia

En estos problemas se tratan los conceptos de clase abstracta, clase base (o superclase), clase derivada (o subclase), clase final, interfaces (para la herencia múltiple) y polimorfismo.

Observación:

Para finalizar la clasificación de los problemas a tratar en este libro, es necesario reconocer que en ellos siempre estará presente la clase *Proyecto*, nombre que se puede reemplazar, como ya se había dicho, por otro como *Prueba*, *Solución*, o algún identificador nemotécnico para el caso que se trata: *Universidad*, *Almacén*, *Producto*, etc. Esta clase puede carecer de atributos (algo absurdo en la realidad, pero que en términos de software se puede manejar), pero siempre tendrá un método imprescindible denominado *principal* (*()*), punto de entrada y salida de la solución planteada. Además, vale reiterar que en los cuatro tipos de problemas se podrán reutilizar las clases de uso común, haciendo los diagramas de clase más expandidos, es decir, que incluyan dos o más clases.

1.4. REFERENCIAS

[Ausubel1983]: Ausubel-Novak-Hanesian. Psicología Educativa: Un punto de vista cognoscitivo. 2° Ed. TRILLAS, México, 1983.

[Booch1999]: Booch-Rumbaugh-Jacobson. El Lenguaje Unificado de Modelado. Addison Wesley Iberoamericana, Madrid, 1999.

[Bruner1966]: Bruner, Jerome. Toward a Theory of Instruction. Cambridge, MA: Harvard University Press, 1966

[Carrol1983]: Carroll, Lewis. Alicia a través del espejo. El libro de Bolsillo, Alianza Editorial, Madrid, 1983.

[González2006]: González Ocampo, Juan Antonio. El aprendizaje basado en problemas como forma de innovación pedagógica. U. de Manizales, Facultad de Ingeniería, Grupo de investigación en innovación curricular. Disponible en:
<http://www.umanizales.edu.co/programs/ingenieria/abpinnovacionpedagogica.pdf>. 2006

[PNDE2006]: Plan Nacional Decenal de Educación 2006-2016. (Colombia). Disponible en:
<http://www.plandecenal.edu.co>

[Villalobos2006]: Villalobos J., Casallas R. Fundamentos de programación. Aprendizaje activo basado en casos. Pearson - Prentice Hall. México, 2006

[Zañartu2000]: Zañartu, Luz María. Aprendizaje colaborativo: Una nueva forma de diálogo interpersonal y en red. 2000 Disponible en: <http://contextoeducativo.com.ar/2003/4/nota-02.htm>.

[Zapata1998]: Zapata, Juan Diego. “El Cuento” y su papel en la enseñanza de la orientación por objetos. IV Congreso RIBIE, Brasilia 1998. Disponible en: <http://www.c5.cl/ieinvestiga/actas/ribie98/146.html>.

CAPÍTULO 2

Fundamentos de programación orientada a objetos

- 2.1. El mundo de los objetos, el mundo de las cosas
 - 2.1.1. Los conceptos de objeto y clase
Problema 1. Hola mundo
 - 2.1.2. Creación de objetos y paso de mensajes
Problema 2. Objetos tipo Círculo
 - 2.2. Tipos de datos primitivos y variables
 - 2.3. Operadores y expresiones
 - 2.4. Clases de uso común: un micro mundo para efectos de reutilización
 - 2.4.1. Tipos de datos estándar como objetos
 - 2.4.2. Paquetes de uso común
 - 2.5. Configuración de las clases de uso común
 - La clase **Objeto**
 - La clase **TipoDeDato**
 - La clase **Flujo**
 - La clase **Cadena**
 - La clase **Mat**
 - 2.6. Problemas resueltos con las clases de uso común
 - Problema 3. Clases de uso común
 - Problema 4. Conversión de una constante a diferentes bases numéricas
 - 2.7. Ejercicios propuestos
 - 2.8. Referencias
-

LA SOSPECHA

Lie Zi

Un hombre perdió su hacha; y sospechó del hijo de su vecino. Observó la manera de caminar del muchacho — exactamente como un ladrón. Observó la expresión del joven — idéntica a la de un ladrón. Observó su forma de hablar — igual a la de un ladrón. En fin, todos sus gestos y acciones lo denunciaban culpable de hurto.

Pero más tarde, encontró su hacha en un valle. Y después, cuando volvió a ver al hijo de su vecino, todos los gestos y acciones del muchacho le parecían muy diferentes a los de un ladrón.

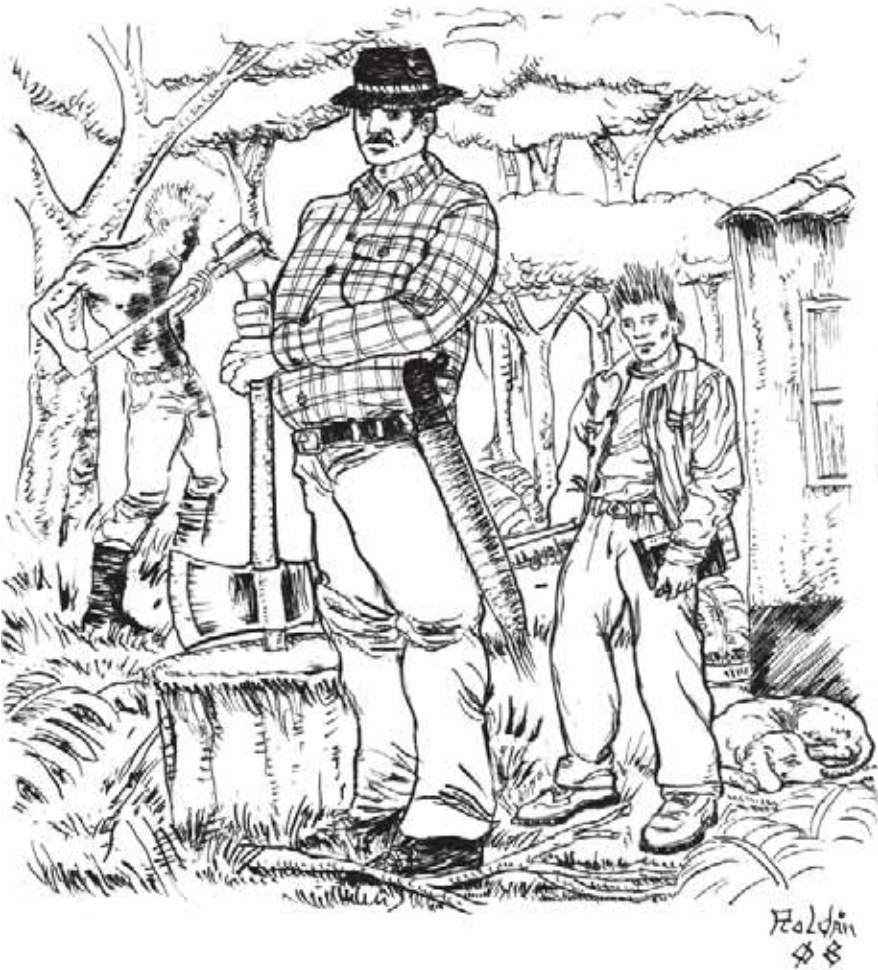
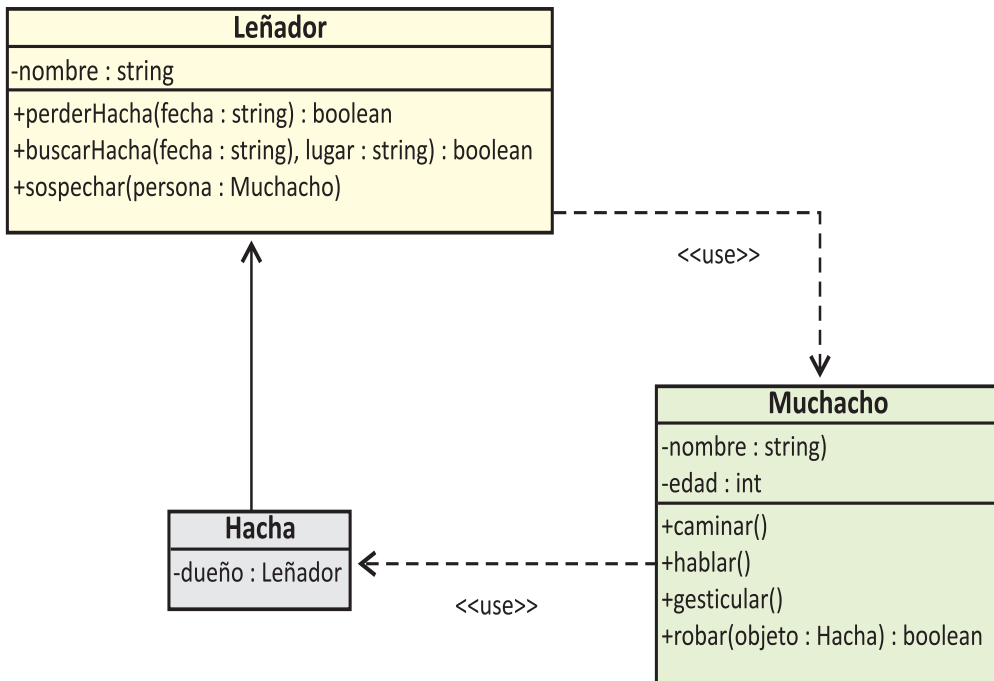


Diagrama de clases:



Objetivos de aprendizaje

- Comprender el concepto de objeto como elemento esencial para la solución de problemas.
- Ilustrar el uso de clases incluidas en paquetes de uso común o reutilización de un componente predefinido.
- Reconocer el concepto de método como estrategia para la solución de un problema, con la técnica de diseño “Divide y vencerás”.
- Efectuar procesos de entrada y salida de datos.

2.1. EL MUNDO DE LOS OBJETOS, EL MUNDO DE LAS COSAS

Según el diccionario de la Real Academia de la Lengua Española, la primera acepción de la palabra Cosa es “Todo lo que tiene entidad, ya sea corporal o espiritual, natural o artificial, real o abstracta” [RAE2001]. En el mismo diccionario, el vocablo Objeto se define como “Todo lo que puede ser materia de conocimiento o sensibilidad de parte del sujeto, incluso este mismo”. Es claro que el mundo en que vivimos está lleno de cosas y objetos.

En el lenguaje de la programación, los gestores del Unified Modeling Language -UML indican que “los términos ‘instancia’ y ‘objeto’ son en gran parte sinónimos y, por ello, la mayoría de las veces pueden intercambiarse”. En términos del paradigma de programación orientado a objetos, un *objeto* es un elemento o instancia de una *clase*, la cual es definida por Booch (1999) como “una descripción de un conjunto de objetos que comparten los mismos atributos, operaciones, relaciones y semántica”. [Booch 1999].

Para aclarar los conceptos de objeto y clase, a continuación se presentarán sus características y comportamientos.

LOS CONCEPTOS DE OBJETO Y CLASE

Un *objeto* es un elemento de una *clase*. Así, un reloj de pulso, el reloj de pared de una morada específica, el reloj Big Ben de Londres, el reloj interno de un computador, todos, pertenecen a una clase llamada Reloj. Asimismo, los caballos Bucéfalo de Alejandro Magno, Strategos de Aníbal, Genitor de Julio César, Rocinante de Don Quijote, Palomo de Bolívar y el corcel que más se recuerda en la vida, son todos objetos de la clase Caballo. La consignación bancaria que realizó Juan Valdés hace dos semanas en el banco principal de su pueblo natal y el retiro por cajero electrónico que su tío acaba de realizar, son ambos objetos pertenecientes a la clase TransaccionBancaria.

Un objeto tiene las mismas *características* (atributos, variables) y *comportamientos* (métodos, operaciones, algoritmos) de todos los elementos de la clase a la cual pertenece. Así, todos los relojes tienen características de forma, marca, precio y tipo, y comportamientos similares como sonar, mostrar hora, despertar y parar. Todos los caballos tienen características de raza, color, peso, edad y alzada, y comportamientos parecidos como correr, relinchar, pastar y dormir. Toda transacción bancaria tiene características de tipo (retiro, consulta, consignación), valor, cuenta asociada, fecha y hora, así como las operaciones de iniciar, finalizar y cancelar.

Los objetos se crean (nacen), interactúan entre sí (se envían mensajes) y mueren (dejan de ser objetos). En un determinado instante, las *características* de un objeto adquieren valores particulares que evidencian su estado. Dos transacciones bancarias pueden tener los estados (“retiro”, 200000, 110107, octubre 17 de 2008, 11:45:23) y (“consignación”, 550000, 235742, noviembre 10 de 2009, 15:00:01).

Las clases y objetos también admiten varias representaciones. Una de las más aceptadas en el mercado del software es la establecida por UML, donde los rectángulos preponderan por su sencillez y claridad, como se observa en las figuras 2.1 y 2.2.

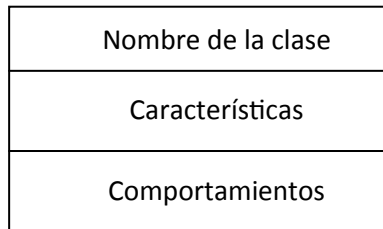


Figura 2.1. Representación de una clase

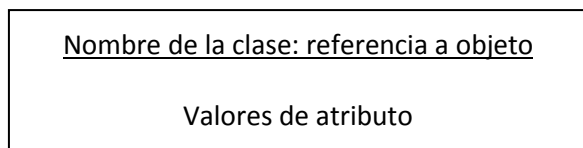


Figura 2.2. Representación de un objeto

Ejemplo 2.1. La figura 2.3 ilustra a la clase Caballo y dos objetos con algunas de sus características.

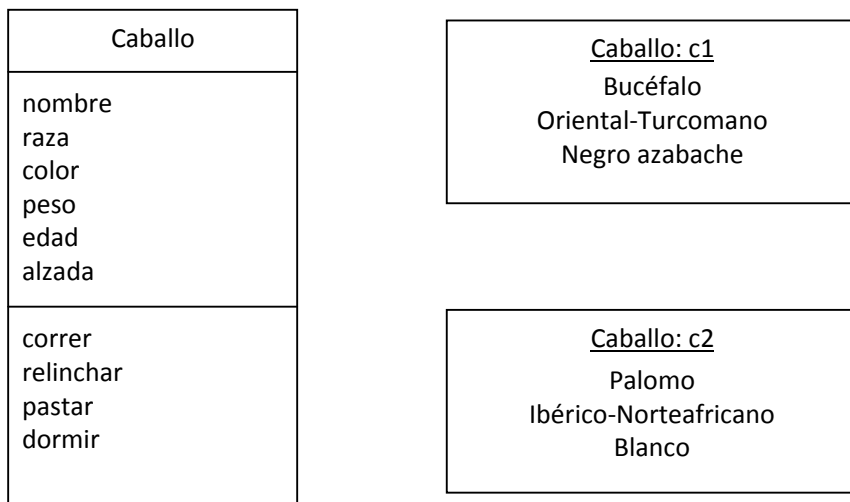


Figura 2.3. La clase Caballo con los objetos c1 y c2

Los objetos por lo general ocultan sus *características* al mundo exterior y muestran los *comportamientos* para poder comunicarse. Para representar lo anterior, con la ayuda del UML se antepone un signo menos (-) a los atributos (significa que son de *visibilidad privada*) y un signo más (+) a los comportamientos (significa que son de *visibilidad pública*), tal como se observa en la figura 2.4. Para diferenciar un atributo de un comportamiento, a este último se le pospone un par de paréntesis () para denotar acción. Por último, puede observarse que a los *atributos* se les asocia un *tipo de dato*, resaltado en letra negrita.

Caballo
-raza: cadena -color: cadena -peso: real -edad: entero -alzada: real
+correr () +relinchar () +pastar () +dormir ()

Figura 2.4. La clase Caballo con atributos privados y comportamientos públicos

Un mecanismo interesante para la comprensión inicial y la ambientación al trabajo con objetos, según [Zapata1998], es la expresión literaria del cuento, narración breve de ficción donde los personajes (sustantivos propios) son los objetos, los sustantivos comunes son las clases, los adjetivos son los *atributos* y los verbos los comportamientos.

Para comprender lo anterior, el lector puede retomar las historias que preceden a los capítulos de este libro. En los diagramas que las acompañan, sus objetos se crean mediante un *método constructor*, que tiene una identidad manifiesta en su *estado*, el cual se controla a través de métodos denominados *modificadores* y tiene unos comportamientos manifiestos a través de los métodos *analizadores*. En el lenguaje de programación, el *estado* de un objeto está dado por los valores actuales de sus

atributos, y se puede cambiar con los métodos `asignarX()` y `obtenerX()`, donde X corresponde a cualquier atributo.

Para una mayor claridad, retomemos el ejemplo 2.1., representado en la figura 2.5. En ella puede observarse que se presenta la clase `Caballo` con los métodos ya mencionados: un constructor (tiene el mismo nombre de la clase), doce métodos modificadores (`asignar` y `obtener` para cada atributo) y los métodos analizadores ya expuestos (`correr`, `relinchar`, `pastar`, `dormir`).

Caballo
- nombre: cadena - raza: cadena - color: cadena - peso: real - edad: entero - alzada: real
+ Caballo() + asignarNombre (cadena p) + asignarRaza (cadena p) + asignarColor (cadena p) + asignarPeso (real p) + asignarEdad (entero p) + asignarAlzada (real p) + obtenerNombre (): cadena + obtenerRaza (): cadena + obtenerColor (): cadena + obtenerPeso (): real + obtenerEdad (): entero + obtenerAlzada (): real + correr () + relinchar () + pastar () + dormir ()

Figura 2.5. La clase `Caballo` con métodos constructores, modificadores y analizadores

El método `Caballo()` permite crear caballos (podríamos afirmar que este método hace que “nazcan” caballos). Todos los métodos *asignar* requieren de un parámetro del mismo tipo del atributo que modifican; los métodos *obtener* siempre retornan un valor del mismo tipo del atributo que recuperan. Para ilustrar todo esto véase el ejemplo 2.2.

Ejemplo 2.2. Crear un caballo llamado Trueno de raza Bávaro, color castaño, peso 350 kg, edad 10 años y alzada 1.60 metros. Visualizar su estado, es decir, mostrar los valores de todos sus atributos.

```

1.  clase Caballeriza
2.      publico estatico vacio principal ( )
3.          Caballo c = nuevo Caballo ( )
4.          c.asignarNombre ("Trueno")
5.          c.asignarRaza ("Bávaro")
6.          c.asignarColor ("Castaño")
7.          c.asignarPeso (350)
8.          c.asignarEdad (10)
9.          c.asignarAlzada (1.60)
10.         Flujo.imprimir ("Los datos del caballo son: ")
11.         Flujo.imprimir ("Nombre: " + c.obtenerNombre ( ))
12.         Flujo.imprimir ("Raza: " + c.obtenerRaza ( ))
13.         Flujo.imprimir ("Color: " + c.obtenerColor ( ))
14.         Flujo.imprimir ("Peso: " + c.obtenerPeso ( ))
15.         Flujo.imprimir ("Edad: " + c.obtenerEdad ( ))
16.         Flujo.imprimir ("Alzada: " + c.obtenerAlzada ( ))
17.     fin_método
18. fin_clase

```

Observaciones:

- La instrucción 3 crea un nuevo caballo sirviéndose del operador **nuevo** y el constructor sin *argumentos* `Caballo()`. El operador **nuevo** permite reservar memoria para el objeto *c*.
- De la instrucción 4 a la 9 se establece un estado para el objeto *c*, asignándole valores a cada uno de sus atributos. Debe tenerse en cuenta que sería incorrecta una instrucción como *c.nombre* = "Trueno" porque el tributo *nombre* tiene visibilidad privada.

Si se hubiera definido en la clase *Caballo* otro constructor sobrecargado como:
 Caballo (cadena n, cadena r, cadena c, real p, entero e, real a), las instrucciones
 3 a la 9 se podrían resumir en una sola, así:

Caballo c = nuevo Caballo("Trueno", "Bávaro", "Castaño", 350, 10, 1.60)

- Las instrucciones 11 a 16 obtienen el estado del objeto *c* y lo visualizan en pantalla por medio del método *imprimir* de la clase *Flujo*. Debe notarse un error cuando se escribe una instrucción como *Flujo.imprimir ("Alzada:" + c.alzada)*, pues el atributo *alzada* es de acceso privado.

A continuación se presentan algunos problemas para afianzar los anteriores conceptos y procedimientos.

PROBLEMA 1: HOLA MUNDO

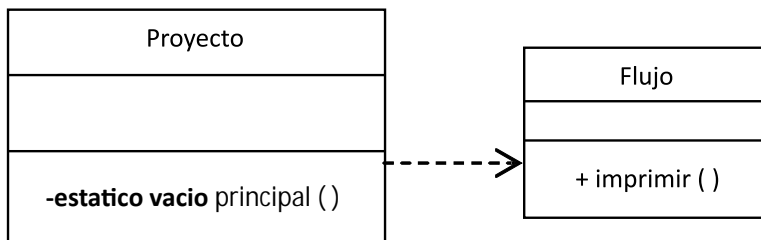
Visualizar el mensaje "Hola mundo".

Este clásico problema se soluciona de forma "inmediata" con las cuatro fases implícitas en la didáctica propuesta en el capítulo 1: Tabla de requerimientos, Diagrama de clases, Contratos de clase y Seudo código orientado a objetos. Cabe anotar que mientras más complejo sea el problema, la aplicación de cada fase se hace más pertinente. Dada la trivialidad del problema, las tres primeras fases se pueden omitir, pudiendo exponer de manera directa el seudo código. Sin embargo, aquí se exponen todas las fases en la búsqueda de logros escuetamente didácticos.

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R	Visualizar un mensaje	Ninguna	El mensaje expuesto al usuario

b) Diagrama de clases



El *diagrama de clases* expuesto es el más simple que se pueda concebir en UML. El rectángulo dividido en tres áreas representa, en el área superior, la clase, que contiene su nombre (*Proyecto o Flujo*); el área del medio corresponde a la declaración de atributos de clase (en este caso se encuentra vacía para ambas clases), y el área inferior contiene la *firma (cabecera o prototipo)* de los métodos.

La clase **Flujo** se ha expuesto de manera parcial. Su estructura completa se encuentra en el apartado 2.5. (*Configuración de las clases de uso común*).

En la firma “+ **estático vacío** principal ()”, el signo cruz (+) indica que el método tiene visibilidad pública; la palabra reservada **estático** indica que el método se puede invocar haciendo uso del nombre de la clase, sin la intermediación de un objeto; y la palabra reservada **vacío** especifica que el método no retorna valor.

Dado que el método *principal()* siempre estará presente en cualquier aplicación porque constituye su punto de entrada y salida, bastará con especificar la visibilidad y nombre, y se asumirá siempre por omisión que el método es estático y no retorna valor. Por tanto, en un diagrama de clases la firma “+ *principal()*” equivale a “+ **estático vacío principal()**”, y en el seudocódigo el encabezado de método “**público principal()**” equivale a “**público estático vacío principal()**”. En varias partes del libro se utiliza la forma abreviada.

c) Contrato de la clase Proyecto

Nombre del método	Requerimiento asociado	Precondición	Postcondición
principal ()	R	No se ha visualizado el mensaje	El mensaje "Hola mundo" se ha emitido

Nota: la columna "Modelo verbal" no aplica para este caso.

d) Seudo código orientado a objetos (OO)

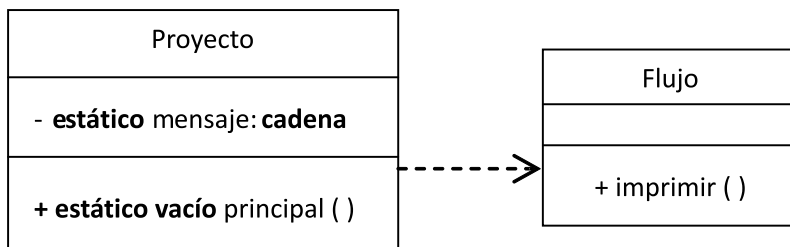
```

clase Proyecto
  publico estatico vacio principal ( )
    Flujo.imprimir ("Hola mundo")
  fin_método
fin_clase

```

Observaciones:

- El método estático *imprimir()* de la clase de uso común *Flujo* permite mostrar información al usuario en el dispositivo estándar de salida (el monitor). Las clases de uso común y los métodos estáticos se estudian en las secciones 2.4 y 3.2, respectivamente.
- El diseño del tipo abstracto de dato Proyecto, aquí presentado, no es el más apropiado, debido a que se expone dicha clase carente de atributos. Otra posible alternativa de solución al problema se presenta modificando la estructura de la clase Proyecto de la siguiente manera:



En este caso se define el atributo estático *mensaje*, con el objetivo de almacenar allí el mensaje “Hola mundo”.

Los *atributos estáticos* se comportan como variables globales para cualquier método declarado como estático dentro de la clase; esto significa que todo método estático puede referenciar directamente a una variable estática, como se observa en el siguiente pseudocódigo.

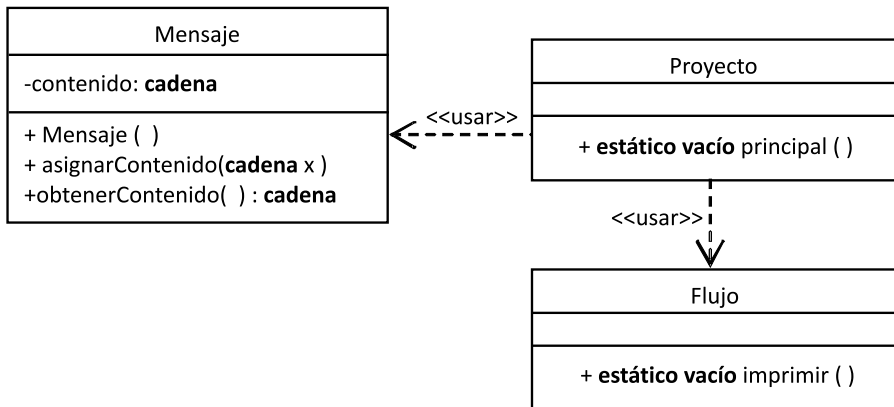
```

clase Proyecto
  privado estatico cadena mensaje

  publico estatico vacío principal ( )
    mensaje = “Hola mundo”
    Flujo.imprimir (mensaje)
  fin_metodo
fin_clase
  
```

El método *principal()* se tiene que declarar como estático porque siempre es llamado antes de que exista cualquier objeto. Los métodos y atributos estáticos se pueden acceder sin la intermediación de un objeto de la clase donde fueron definidos; estos conceptos se profundizarán en el ítem 3.2 (*Métodos*).

- Una alternativa adicional donde se utiliza un objeto para imprimir el mensaje se presenta a continuación.



La clase *Mensaje* consta de un atributo de tipo *cadena* y tres métodos: el constructor (tiene el mismo nombre de la clase) y los métodos modificadores *asignarContenido()* y *obtenerContenido()*.

```

clase Mensaje
  privado cadena contenido

  publico Mensaje ( )
  fin_metodo
  //-----
  publico asignarContenido(cadena x)
    contenido = x
  fin_metodo
  //-----
  publico cadena obtenerContenido( )
    retornar contenido
  fin_metodo
fin_clase
//*****
  
```

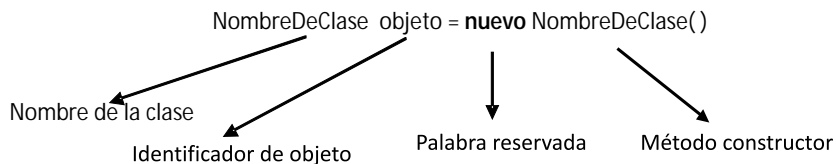
```

clase Proyecto
  publico estatico vacío principal ( )
    Mensaje saludo = nuevo Mensaje( )
    saludo.asignarContenido("Hola mundo")
    Flujo.imprimir(saludo.obtenerContenido( ))
  fin_metodo
fin_clase
  
```

Se pueden notar en el diagrama de clases las relaciones de *dependencia* representadas con líneas dirigidas punteadas, las cuales indican que la *clase Proyecto* “usa” o utiliza un objeto de tipo *Mensaje* y un método de la clase *Flujo*.

CREACIÓN DE OBJETOS Y PASO DE MENSAJES

Un objeto se crea mediante el *método constructor*, fácil de identificar porque tiene el mismo nombre de la clase a la cual pertenece. El formato general para la creación de un objeto es:



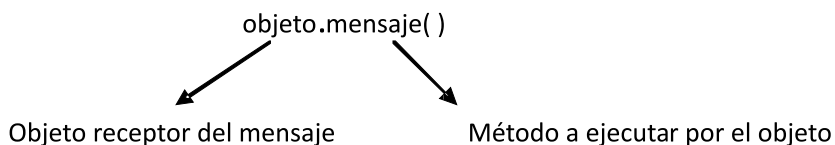
Algunos ejemplos:

Entero num = nuevo Entero()

Real x = nuevo Real()

Producto p = nuevo Producto()

Los objetos se comunican entre sí a través del *paso de mensajes*. Cuando un objeto recibe un mensaje se obliga a la ejecución de un método. En general se tiene:



En particular, se enviarán tres mensajes, uno a cada objeto creado en los ejemplos anteriores:

```

num.asignarValor(17)
max = x.obtenerMaxValor()
p.asignarPrecio(23890.5)

```

PROBLEMA 2. OBJETOS TIPO CÍRCULO

La clase Círculo se encuentra definida de la siguiente manera:

Círculo
<ul style="list-style-type: none"> - <i>x</i>: entero - <i>y</i>: entero - <i>radio</i>: real - <i>colorContorno</i>: caracter - <i>colorRelleno</i>: caracter
<ul style="list-style-type: none"> + Círculo () + Círculo (entero <i>x1</i>, entero <i>y1</i>, real <i>r1</i>) + asignarX (entero <i>p</i>) + asignarY (entero <i>p</i>) + asignarRadio (real <i>p</i>) + asignarColorC (caracter <i>p</i>) + asignarColorR (caracter <i>p</i>) + obtenerX (): entero + obtenerY (): entero + obtenerRadio (): real + obtenerColorC (): cadena + obtenerColorR (): cadena + dibujar () + área (): real + perímetro (): real

Los atributos *x* e *y* corresponden a las coordenadas del centro del círculo; el atributo *radio* a la longitud del radio; los atributos de tipo carácter representan los colores del contorno (negro, amarillo, azul o rojo) e interior del círculo (negro, amarillo, azul, rojo o sin relleno).

Respecto a los métodos se debe observar lo siguiente:

- El constructor sin argumentos crea un círculo con centro (0, 0), radio igual a cero, color de contorno negro y sin relleno.
- El constructor sobrecargado con los argumentos $x1$, $y1$ y $r1$ crea un círculo con centro $(x1, y1)$ y radio $r1$; también con un color de perímetro negro y sin relleno.
- Los métodos `asignarX(entero p)`, `asignarY(entero p)` y `asignarRadio(real p)`, permiten modificar las coordenadas del centro y la longitud del radio.
- Los métodos `asignarColorC(caracter p)` y `asignarColorR(caracter p)` establecen el color del contorno y el relleno, respectivamente. En ambos casos el parámetro p tiene la siguiente interpretación:

$$p = \begin{cases} \text{'n'} \rightarrow \text{negro} \\ \text{'a'} \rightarrow \text{amarillo} \\ \text{'b'} \rightarrow \text{azul} \\ \text{'r'} \rightarrow \text{rojo} \\ \text{'s'} \rightarrow \text{sin relleno} \end{cases}$$

- Todos los métodos *obtener* devuelven el valor respectivo de su atributo; así, `obtenerX()` retorna el valor de la abscisa del centro y `obtenerY()` el valor de la ordenada; sin embargo, los métodos `obtenerColorC()` y `obtenerColorR()` retornan la cadena correspondiente al color del contorno o del relleno, en lugar del carácter asignado al color.
- El método `dibujar()`, dibuja un círculo con centro, radio y colores de contorno y relleno asignados al momento.
- El método `área()` calcula y retorna el área del círculo.
- El método `perímetro()` calcula y retorna el perímetro del círculo.

Se requiere escribir un pseudocódigo orientado a objetos que permita:

1. Dibujar el círculo `c1` con centro (7, 8) y radio 9.2 cm, contorno negro y sin relleno.
2. Dibujar los círculos `c2` (3, 4, 8.0) y `c3` (-9, 3, 4.7). El círculo `c2` debe tener contorno rojo y relleno amarillo. El círculo `c3` irá todo pintado de azul.
3. Imprimir el área y el perímetro de los tres círculos.

Solución:

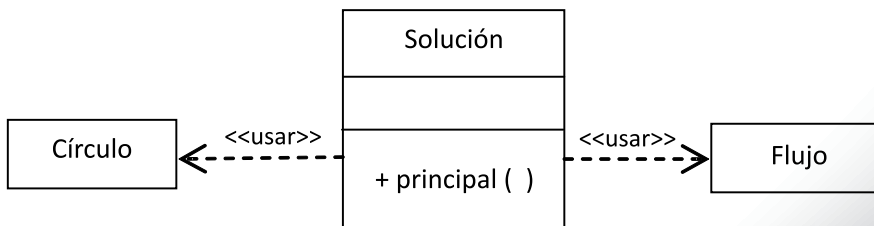
a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Dibujar el círculo c1 con centro (7, 8) y radio 9.2 cm, perímetro negro y sin relleno.	Ninguna	Primer círculo (c1) dibujado.
R2	Dibujar los círculos c2 (3, 4, 8.0) y c3 (-9, 3, 4.7), con los colores de contorno y relleno especificados.	Ninguna	Segundo y tercer círculo dibujados (c2 y c3).
R3	Hallar el área y el perímetro de los tres círculos.	Ninguna	El área y perímetro de los círculos c1, c2 y c3.

Observación:

La tabla de requerimientos indica que R1, R2 y R3 carecen de entradas. No sería contraproducente especificar que se tienen entradas no nulas, en este caso constantes. Por ejemplo, para el requerimiento R1 la entrada sería “Tres constantes enteras, dos para el centro (7 y 8) y una para el radio (9.2)”. Sin embargo, se ha decidido escribir “Ninguna” teniendo en cuenta que toda entrada implica un ingreso de datos por el usuario o una lectura desde un dispositivo externo de memoria, como un disco magnético.

b) Diagrama de clases



c) Seudocódigo

```

clase Solución
  publico estatico vacio principal ()
    // Requerimiento R1
    Círculo c1 = nuevo Círculo ()
    c1.asignarX (7)
    c1.asignarY (8)
    c1.asignarRadio (9.2)
    c1.dibujar ()
    // Requerimiento R2
    Círculo c2 = nuevo Círculo (3, 4, 8.0)
    c2.asignarColorC('r')
    c2.asignarColorR('a')
    c2.dibujar ()
    Círculo c3 = nuevo Círculo (-9, 3, 4.7)
    c3.asignarColorC('b')
    c3.asignarColorR('b')
    c3.dibujar ()
    // Requerimiento R3
    Flujo.imprimir ("Área y perímetro de los tres círculos:")
    Flujo.imprimir ("Círculo 1:")
    Flujo.imprimir ("Área = " + c1.área() + "cm2")
    Flujo.imprimir ("Perímetro = " + c1.perímetro() + "cm")
    Flujo.imprimir ("Círculo 2:")
    Flujo.imprimir ("Área = " + c2.área() + "cm2")
    Flujo.imprimir ("Perímetro = " + c2.perímetro() + "cm")
    Flujo.imprimir ("Círculo 3:")
    Flujo.imprimir ("Área = " + c3.área() + "cm2")
    Flujo.imprimir ("Perímetro = " + c3.perímetro() + "cm")
  fin_metodo
fin_clase

```

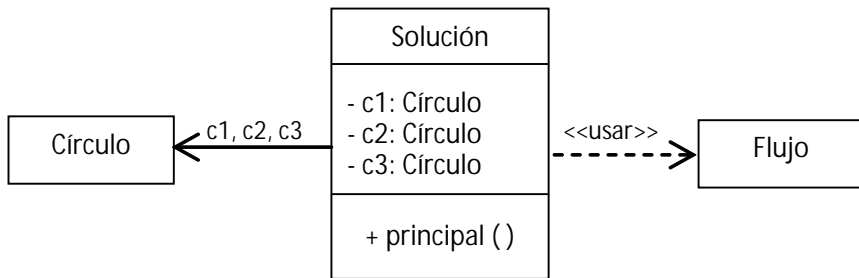
Observaciones:

- El círculo *c1* fue creado con el constructor sin argumentos y los círculos *c2* y *c3* con el constructor sobrecargado.
- Al imprimir el área y perímetro de un círculo, también se puede visualizar su estado, es decir, la información referente a su centro y radio. Para el caso del círculo *c1* se tendría:

```

Flujo.imprimir ("Círculo 1:")
Flujo.imprimir ("Centro: (" + c1.obtenerX() + ", " + c1.obtenerY() + ")")
Flujo.imprimir ("Radio: " + c1.obtenerRadio() + "cm")
Flujo.imprimir ("Área = " + c1.area() + "cm²")
Flujo.imprimir ("Perímetro = " + c1.perímetro() + "cm")
    
```

- Un diseño orientado a objetos más adecuado, que se estudia en el capítulo 5 (Relaciones entre clases), es el siguiente:



Se deja propuesto el desarrollo del pseudocódigo concerniente.

2.2. TIPOS DE DATOS PRIMITIVOS Y VARIABLES

Un *dato* es la representación formal de hechos, conceptos o instrucciones, adecuada para su comunicación, interpretación y procesamiento por seres humanos o medios automáticos. Un *tipo de datos* es la especificación de un dominio (rango de valores) y de un conjunto válido de operaciones a los que normalmente los traductores⁶ asocian un esquema de representación interna propio.

⁶ Un traductor es un lenguaje que decodifica las instrucciones escritas en un lenguaje de programación para que un computador las pueda interpretar.

Cuando un tipo de datos está definido para un lenguaje o pseudo lenguaje, se habla de *tipos de datos primitivos o estándar*, utilizados para la construcción de expresiones o nuevos tipos de datos. El pseudo lenguaje definido en el proyecto Método Integrado de Programación Secuencial y Programación Orientada a Objetos -MIPSOO [GIISTA2006], utiliza cinco tipos de datos:

- **entero:** números enteros con signo almacenados en 32 bits (4 bytes), comprendidos entre -2147483648 a 2147483647 (-2^{31} a $2^{31}-1$)

Ejemplos:

-56789, 403, 0, 2344521567

- **real:** denota un tipo simple que almacena valores de punto flotante de 32 bits, en un rango aproximado de $3.4 * 10^{-38}$ a $3.4 * 10^{38}$.

Ejemplos:

3.1416, -88.7, 49795.1, 17, -1.0

Las cantidades reales anteriores, separadas por comas, incluyen punto decimal. Teniendo en cuenta que los números enteros son un subconjunto de los números reales, la cifra 17 dada en los ejemplos se podría haber escrito de igual manera como 17.0.

- **caracter:** se utiliza para declarar un *carácter Unicode*. Todo carácter va encerrado entre comillas simples.

Ejemplos:

'@', ',', 'b', 'B', '4', ' ', ' + '

Se puede notar que los caracteres pueden ser numéricos como '4', alfabéticos como 'B' y especiales como '@'. La barra espaciadora (carácter blanco) se representa como dos comillas simples vacías (' ').

- **logico:** se utiliza para declarar variables que almacenan el *valor de verdad cierto* o *falso*, únicos valores posibles para una variable de este tipo, propios del algebra de Boole.

- **cadena:** este tipo representa una cadena de caracteres Unicode. Toda cadena se escribe entre comillas dobles.

Ejemplos:

“XYZ123”, “Programar con OO es fascinante”, “Ser o no ser”, “123”

Los tipos de datos se utilizan para declarar variables locales, definir atributos de clases, declarar parámetros formales y especificar valores de retorno de los métodos que se comportan como una función.

Una *variable* se considera como un compartimiento de memoria con una dirección y nombre únicos, cuyo contenido puede variar en tiempo de ejecución del programa. Para efectos de procesamiento de una variable, interesa conocer su nombre y no su dirección. Un nombre de variable es un *identificador* que debe cumplir las siguientes características:

- Debe comenzar con una letra.
- No puede contener espacios ni caracteres especiales, a excepción del guión bajo o underline.
- No puede coincidir con una palabra reservada del pseudo lenguaje o con un tipo de datos estándar. El apéndice B expone el listado de palabras reservadas.
- No existe límite para la longitud de un identificador, aunque en general se aconseja que no exceda los 15 caracteres para facilitar su escritura y manejo. A continuación se dan ejemplos de identificadores válidos y observaciones sobre su creación.

Tabla 2.1. Identificadores válidos

Algunos identificadores inválidos se presentan en la siguiente tabla:

IDENTIFICADOR	OBSERVACIÓN
suma contadorDePlantas díasConFrío árbol mesa	Los cinco identificadores contienen sólo letras

cedula cédula	Ambos identificadores son distintos: se diferencian por la vocal <i>e</i> tildada. Es necesario aclarar que se evitarán las tildes en los identificadores para facilitar el proceso de codificación a un lenguaje de programación
promedioAnual promedio_anual tiempoEsperado tiempo_esperado deporteExtremo deporte_extremo	Los seis identificadores están formados por dos palabras; si no se utiliza el guión bajo, la letra inicial de cada nueva palabra, a excepción de la primera, va en mayúsculas para efectos de claridad en su lectura. Si las palabras van separadas por guión bajo, se puede escribir todo en minúsculas
PI VALOR_MIN CONST_DE_AVOGADRO	Por estar escritos en mayúsculas, deben hacer referencia a constantes figurativas
Animal Producto CuentaBancaria	Al comenzar con letra mayúscula deber hacer referencia a nombres de clases
mes7, cont123, a1b2_c	Contienen combinaciones de letras, dígitos y guiones

Tabla 2.2. Identificadores inválidos

IDENTIFICADOR	OBSERVACIÓN
suma total	Contiene un espacio
3cedula	Comienza con un dígito
si Clase fin_si segun	Coinciden con palabras reservadas del pseudo lenguaje
entero caracter	Coinciden con tipos de datos estándar o primitivos
Entero Mat Real Cadena	Coinciden con clases de uso común
calle#31 valorEn\$, pago*mes	Contienen caracteres especiales no permitidos

Para *declarar variables* se sigue la siguiente sintaxis general:
tipoDeDato lista_de_identificadores_separados_por_comas

veamos:

entero x

real y

entero edad, contador

logico sw, sexo, estado

Si un identificador se declara como *parámetro* o al interior de un método se considera *campo variable local* o simplemente *variable local*; si se declara por fuera de todo método se convierte en un *atributo* de clase. Por ejemplo, en la figura 2.6 *codigo* y *precio* son atributos, mientras que *c* y *p* son variables locales pasadas como parámetros.

Producto
<ul style="list-style-type: none"> - codigo: cadena - precio: real
<ul style="list-style-type: none"> + asignarCodigo(cadena c) + asignarPrecio(real p)

Figura 2.6. Diferencia entre atributos y variable

2.3. OPERADORES Y EXPRESIONES

Se utilizarán en este libro los *operadores binarios* del lenguaje C, especificados en la tabla 2.3.

Tabla 2.3. Operadores binarios del lenguaje C

TIPO DE OPERADOR	OPERADORES	DESCRIPCIÓN
Aritméticos	+ - * / % -	Suma Resta Producto División Módulo Menos unitario
Lógicos	&& !	Conjunción Disyunción Negación
Relacionales	< <= > >= == !=	Menor que Menor o igual que Mayor que Mayor o igual que Igual a Diferente de
De asignación	=	Asignación
De concatenación	+	Concatenar con

Una *expresión*, es una combinación válida de *operadores* y *operandos* que devuelven un valor único. En la expresión $a + b - c$, las variables a , b y c son *operandos* y los símbolos $+$ y $-$ son *operadores*.

Una expresión puede ser aritmética, lógica, relacional, de asignación o de concatenación, como se indica en la tabla contigua:

Tabla 2.4. Tipos de expresiones

TIPO DE EXPRESIÓN	EXPRESIONES	DESCRIPCIÓN
Aritmética	$a + b$ $5 * 7$ $\text{valor} - 17 \% 9$ $\text{Mat.seno}(30) - \text{Mat.coseno}(45) * 2$	<ul style="list-style-type: none"> • Suma de dos variables • Producto de dos constantes • Expresión compuesta: resta a la variable <i>valor</i> el resultado de $17 \% 9$ • Expresión que incluye a la clase de uso común Mat para calcular el seno de un ángulo y el coseno de otro

Lógica	$p \ \&\& \ q \ \ r$	Expresión lógica compuesta. Las variables p , q y r son de tipo lógico
Relacional	$a < b$ $\text{salario} \geq 2500000$ $\text{nombre} == \text{"Ana"}$	<ul style="list-style-type: none"> • Compara dos variables • Compara una variable con una constante numérica • Compara una variable con una constante de cadena
De asignación	$x = 9$ $\text{nombre} = \text{"Darío"}$	<ul style="list-style-type: none"> • Asigna a una variable, una constante numérica entera • Asigna a una variable, una constante de tipo cadena
De concatenación	$\text{"Total ="} + \text{total}$	Concatena un texto (o constante de tipo cadena) con una variable de tipo cadena.

Observaciones:

- Una expresión lógica puede combinar sub-expresiones aritméticas, lógicas y relacionales, como en el siguiente caso:
 $(7 \% 5 < 9) \ \&\& \ (\text{edad} > 20) \ || \ (\text{indicativo} == \text{cierto})$
- En los operadores aritméticos el módulo (%) sólo opera con enteros y el resultado es otro entero.

La operación $33 \% 7$ da como resultado el residuo de la división entera, es decir $33 \% 7 == 5$ porque

$$\begin{array}{r} 33 \mid 7 \\ 5 \mid 4 \end{array}$$

- El menos unitario devuelve el mismo tipo que el operando. En el resto de los operadores aritméticos, si algún operador es real, el resultado será real.
- Para los operadores lógicos los operandos deben ser del mismo tipo.
- En la asignación sólo se pueden operar datos del mismo tipo en los miembros izquierdo (que siempre es una variable) y derecho (que puede ser otra variable,

una constante o una expresión). Para la asignación entre diferentes tipos, se puede aplicar la *conversión forzada de tipos* (casting), como en el siguiente caso:

entero a, b

real c

$c = (\text{real}) a + b$

- La *prioridad de un operador* hace referencia al orden en el cual se ejecuta dentro de una expresión. La prioridad se puede alterar con el uso del agrupador paréntesis, en cuyo caso se evalúa primero su contenido. Obsérvese la tabla 2.5, organizada de arriba hacia abajo según la precedencia del operador, donde el agrupador paréntesis posee la mayor prioridad; y los operadores de relación y asignación, la menor.

Tabla 2.5. Prioridad de los operadores

Prioridad de los operadores	
Paréntesis	()
Operadores unitarios	-, +
Operaciones de la <i>clase Mat</i>	Mat.elevar(), Mat.aleatorio(), etc.
Operadores multiplicativos	*, /, %, &&
Operadores auditivos	+, -,
Operadores de relación	=, !=, <, <=, >, >=
Operador de asignación	=

En el ejemplo 2.3. se evaluarán tres expresiones donde se utilizan los mismos operandos y operadores, pero se cambia el orden de prioridad con el uso de paréntesis. Los círculos debajo de las llaves indican el orden en el cual se efectúan las operaciones. Notar que los tres resultados son distintos.

Ejemplo 2.3: evaluación de tres especímenes aritméticas

$$1) \quad 5 - 8 * 7 / 6 + \underbrace{\text{Mat.elevar}(2, 5) * 3 \% 2}_{\textcircled{1} 32}$$

$$5 - \underbrace{8 * 7 / 6}_{\textcircled{2} 56} + 32 * 3 \% 2$$

$$5 - \underbrace{56 / 6}_{\textcircled{3} 9.33} + 32 * 3 \% 2$$

$$5 - 9.33 + \underbrace{32 * 3 \% 2}_{\textcircled{4} 96}$$

$$5 - 9.33 + \underbrace{96 \% 2}_{\textcircled{5} 0}$$

$$\underbrace{5 - 9.33}_{\textcircled{6} -4.33} + 0$$

$$\underbrace{-4.33 + 0}_{\textcircled{7} -4.33}$$

$$\textcircled{7} \text{ -4.33 } \rightarrow \text{ Respuesta}$$

$$2) (5 - 8 * \underbrace{(7 / 6)}) + \text{Mat.elevar}(2, 5) * 3 \% 2$$

① 1.16

$$(5 - \underbrace{8 * 1.16}) + \text{Mat.elevar}(2, 5) * 3 \% 2$$

② 9.28

$$\underbrace{(5 - 9.28)} + \text{Mat.elevar}(2, 5) * 3 \% 2$$

③ -4.28

$$-4.28 + \underbrace{\text{Mat.elevar}(2, 5) * 3 \% 2}$$

④ 32


$$-4.28 + \underbrace{32 * 3 \% 2}$$

⑤ 96

$$-4.28 + \underbrace{96 \% 2}$$

⑥ 0

$$\underbrace{-4.28 + 0}$$

⑦ -4.28  Respuesta

$$3) \underbrace{(5 - 8)}_{\textcircled{1} -3} * 7 / (6 + (\text{Mat.elevar}(2, 5) * 3) \% 2)$$

$$-3 * 7 / (6 + \underbrace{(\text{Mat.elevar}(2, 5) * 3)}_{\textcircled{2} 32} \% 2)$$

$$-3 * 7 / (6 + \underbrace{(32 * 3)}_{\textcircled{3} 96} \% 2)$$

$$-3 * 7 / (6 + \underbrace{96 \% 2}_{\textcircled{4} 0})$$

$$-3 * 7 / \underbrace{(6 + 0)}_{\textcircled{5} 6}$$

$$\underbrace{-3 * 7}_{\textcircled{6} -21} / 6$$

$$\underbrace{-21 / 6}$$

$$\textcircled{7} \text{ -3.5 } \rightarrow \text{ Respuesta }$$

2.4. CLASES DE USO COMÚN: UN MICRO MUNDO PARA EFECTOS DE REUTILIZACIÓN

Las *clases de uso común*, como su nombre lo indica, son de uso generalizado para una gran variedad de problemas y establecen un marco de trabajo similar a un pequeño mundo reutilizable.

TIPOS DE DATOS ESTÁNDAR COMO OBJETOS

Los tipos de datos primitivos *entero*, *real*, *caracter*, *logico* y *cadena* representan valores simples, no objetos complejos. En ciertos casos, el uso de tipos de datos primitivos puede aumentar el rendimiento por cuestiones de almacenamiento interno. Sin embargo, en ocasiones la eficiencia así lograda dificulta ciertos procesos, resultando más conveniente el tratamiento de los tipos de datos en base a objetos.

En lo que sigue se analizarán las clases de uso común. Debe aclararse que el nombre de las clases correspondientes a cada tipo de datos coincide con el nombre del dato primitivo; sólo cambia la primera letra del nombre del tipo a mayúscula: *Entero*, *Real*, *Caracter*, *Logico* y *Cadena*.

LA CLASE ENTERO

Entero
<ul style="list-style-type: none"> - valor: entero - const estatico entero MIN_VALOR = -2.147.483.648 - const estatico entero MAX_VALOR = 2.147.483.647
<ul style="list-style-type: none"> + Entero () + asignarValor (entero x) + obtenerValor (): entero + obtenerMinValor(): entero + obtenerMaxValor(): entero + estatico obtenerMinValor(): entero + estatico obtenerMaxValor(): entero + esIgual (Entero x): logico + convBin (entero x): Cadena + convBin (): Cadena + estatico convBin (entero x): Cadena + convOctal (entero x): Cadena + convOctal (): Cadena + estatico convOctal (entero x): Cadena + convHex (entero x): Cadena + convHex (): Cadena + estatico convHex (entero x): Cadena + convertirANum (cadena x): entero + estatico convertirANum (cadena x): entero

Miembros dato de la clase Entero:

valor: cantidad que representa el valor entero, comprendido entre -2.147.483.648 y 2.147.483.647 (almacenamiento de un entero en 4 bytes).

MIN_VALOR: constante estática entera igual a -2.147.483.648

MAX_VALOR: constante estática entera igual a 2.147.483.647

Tabla 2.6. Métodos de la clase Entero

MÉTODO	DESCRIPCIÓN
Entero()	Constructor por defecto. Inicializa el miembro dato <i>valor</i> en 0
asignarValor (entero x)	Asigna el entero <i>x</i> al miembro dato <i>valor</i>
obtenerValor (): entero	Devuelve el valor actual del miembro dato <i>valor</i>
obtenerMinValor(): entero	Devuelve el mínimo valor entero
obtenerMaxValor(): entero	Devuelve el máximo valor entero
estatico obtenerMinValor(): entero	Devuelve el mínimo valor entero. Método de clase
estatico obtenerMaxValor(): entero	Devuelve el máximo valor entero. Método de clase
esIgual (Entero x): logico	Devuelve cierto si el objeto Entero que lo llama es equivalente a <i>x</i> . En caso contrario devuelve falso
convBin (entero x): Cadena	Convierte en binario el entero <i>x</i> . Retorna el resultado como un objeto de tipo Cadena
convBin (): Cadena	Retorna una cadena igual al equivalente binario del miembro dato <i>valor</i>
estatico convBin (entero x): Cadena	Retorna una cadena igual al equivalente binario del parámetro <i>x</i> . Método de clase
convOctal (entero x): Cadena	Convierte a base octal el entero <i>x</i> . Retorna el resultado como un objeto de tipo Cadena
convOctal (entero x): Cadena	Retorna una cadena igual al equivalente octal del miembro dato <i>valor</i>
estatico convOctal (entero x): Cadena	Retorna una cadena igual al equivalente octal del parámetro <i>x</i> . Método de clase
convHex (entero x): Cadena	Convierte a base hexadecimal el entero <i>x</i> . Retorna el resultado como un objeto de tipo Cadena
convHex (): Cadena	Retorna una cadena igual al equivalente hexadecimal del miembro dato <i>valor</i>
estatico convHex (entero x): Cadena	Retorna una cadena igual al equivalente hexadecimal del parámetro <i>x</i> . Método de clase
convertirANum(cadena x): entero	Convierte una cadena a número entero
estatico convertirANum (cadena x): entero	Convierte una cadena a número entero Método de clase

LA CLASE REAL

Real
- valor: real - const estatico real MIN_VALOR = $3.4 * 10^{-38}$ - const estatico real MAX_VALOR = $3.4 * 10^{38}$
+ Real () + asignarValor (real x) + obtenerValor (): real + obtenerMinValor(): real + obtenerMaxValor(): real + estatico obtenerMinValor(): real + estatico obtenerMaxValor(): real + esIgual (Real x): lógico + convBin (real x): Cadena + convBin (): Cadena + estatico convBin (real x): Cadena + convOctal (real x): Cadena + convOctal (): Cadena + estatico convOctal (real x): Cadena + convHex (real x): Cadena + convHex (): Cadena + estatico convHex (real x): Cadena + convertirANum (cadena x): real + estatico convertirANum (cadena x): real

Miembros dato de la clase Real:

valor: cantidad que representa el valor real, comprendido en el rango de $3.4 * 10^{-38}$ a $3.4 * 10^{38}$.

MIN_VALOR: constante estática real igual a $3.4 * 10^{-38}$

MAX_VALOR: constante estática real igual a $3.4 * 10^{38}$

Tabla 2.7. Métodos de la clase Real

MÉTODO	DESCRIPCIÓN
Real ()	Constructor por defecto. Inicializa el miembro dato <i>valor</i> en 0.
asignarValor (real x)	Asigna el real <i>x</i> al miembro dato <i>valor</i> .
obtenerValor (): real	Devuelve el valor actual del miembro dato <i>valor</i> .
obtenerMinValor (): real	Devuelve el mínimo valor real.
obtenerMaxValor (): real	Devuelve el máximo valor real.
estatico obtenerMinValor (): real	Devuelve el mínimo valor real. Método de clase.
estatico obtenerMaxValor (): real	Devuelve el máximo valor real. Método de clase.
esIgual (Real x): lógico	Devuelve cierto si el miembro dato <i>valor</i> es equivalente a <i>x</i> . En caso contrario devuelve falso.
convBin (real x): Cadena	Convierte en binario el real <i>x</i> . Retorna el resultado como un objeto de tipo Cadena.
convBin (): Cadena	Retorna una cadena igual al equivalente binario del miembro dato <i>valor</i> .
estático convBin (real x): Cadena	Retorna una cadena igual al equivalente binario del parámetro <i>x</i> . Método de clase.
convOctal (real x): Cadena	Convierte a base octal el real <i>x</i> . Retorna el resultado como un objeto de tipo Cadena.
convOctal (): Cadena	Retorna una cadena igual al equivalente octal del miembro dato <i>valor</i> .
estático convOctal (real x): Cadena	Retorna una cadena igual al equivalente octal del parámetro <i>x</i> . Método de clase.
convHex (real x): Cadena	Convierte a base hexadecimal el real <i>x</i> . Retorna el resultado como un objeto de tipo Cadena.
convHex (): Cadena	Retorna una cadena igual al equivalente hexadecimal del miembro dato <i>valor</i> .
estático convHex (real x): Cadena	Retorna una cadena igual al equivalente hexadecimal del parámetro <i>x</i> . Método de clase.
convertirANum(Cadena x): real	Convierte una cadena a número real.
estático convertirANum (cadena x): real	Convierte una cadena a número real. Método de clase.

LA CLASE CHARACTER

Character
- valor: character - const estatico character MIN_VALOR = nulo - const estatico character MAX_VALOR = ' '
+ Carácter () + asignarValor (Character x) + obtenerValor (): Character + esDígito (Character x): logico + esDígito (): logico + esLetra (Character x): lógico + esLetra (): logico + esEspacio (Character x): logico + esEspacio (): logico + aMinúscula (Character x): character + aMinúscula (): character + aMayúscula (Character x): character + aMayúscula (): character + estatico convertirACarácter (cadena x): character

El miembro dato *valor* es una cantidad que representa un carácter, comprendido entre MIN_VALOR (nulo, carácter ASCII 0) y MAX_VALOR (blanco, carácter ASCII 255).

Tabla 2.8. Métodos de la clase Carácter

MÉTODO	DESCRIPCIÓN
Character ()	Constructor por defecto. Inicializa el miembro dato <i>valor</i> con un espacio en blanco.
asignarValor (Character x)	Modifica el estado del objeto, al asignar el dato <i>x</i> al miembro dato <i>valor</i> .
obtenerValor (): Character	Retorna el valor actual del atributo <i>valor</i> .
esDígito (Character x): logico	Devuelve cierto si el <i>argumento x</i> corresponde a un dígito. De lo contrario, falso .
esDígito (): logico	Devuelve cierto si el miembro dato <i>valor</i> corresponde a un dígito. De lo contrario, falso .

esLetra (caracter x): logico	Devuelve cierto si el <i>argumento x</i> corresponde a un Caracter alfabético. De lo contrario, falso.
esLetra (): logico	Devuelve cierto si el miembro dato <i>valor</i> corresponde a un carácter alfabético. De lo contrario, falso.
esEspacio (caracter x): logico	Devuelve cierto si el argumento <i>x</i> corresponde a un espacio en blanco. De lo contrario devuelve falso.
esEspacio (): logico	Devuelve cierto si el miembro dato <i>valor</i> corresponde a un espacio en blanco. De lo contrario devuelve falso.
aMinúscula (Caracter x): caracter	Devuelve la letra minúscula correspondiente al carácter <i>x</i> .
aMinúscula (): caracter	Devuelve la letra minúscula correspondiente al miembro dato <i>valor</i> .
aMayúscula (caracter x): caracter	Devuelve la letra mayúscula correspondiente al carácter <i>x</i> .
aMayúscula (): caracter	Devuelve la letra mayúscula correspondiente al miembro dato <i>valor</i> .
convertirACarácter (cadena x): caracter	Convierte una cadena en carácter. Si la cadena consta de varios caracteres, se retorna el primer carácter de la cadena.

LA CLASE LOGICO

Logico
- valor: logico
+ Logico () + asignarValor (logico x) + obtenerValor (): logico + iguales (logico x): logico

La clase cadena será expuesta como una clase de uso común.

Nota: Se sugiere al docente y a los estudiantes desarrollar algunos de los métodos de las clases base de uso común antes especificadas, a manera de ejercicios para el repaso y aplicación de los conceptos estudiados.

PAQUETES DE USO COMÚN

La solución de problemas en diferentes contextos, permite identificar que un grupo de clases se utiliza en todas o en la mayoría de las soluciones planteadas. Estas clases se denominan de uso común y pueden empaquetarse para efectos de reutilización. Esta característica también las agrupa en *paquetes sistema y contenedor*. El primero de ellos (*sistema*) se importa por defecto e incluye las clases que encapsulan los tipos de datos primitivos (**Entero**, **Real**, **Caracter**, **Logico**, **Cadena**). Mientras que la clase **Mat**, cuya utilidad radica en las funciones matemáticas de uso frecuente, contiene la clase **Flujo** para controlar la entrada y salida estándar, la clase **Cadena** para el tratamiento de cadenas de caracteres y la clase **Excepcion** para el control de los errores en tiempo de ejecución. El segundo paquete (*contenedor*) incluye tipos abstractos de datos contenedores denominados *estructuras de datos internas* (residen en memoria RAM), como las clases **Arreglo**, **Vector**, **Matriz**, **Pila**, **Cola**, **Listaligada**, **Árbol**, **ListaOrdenada**, **ABB** (árbol binario de búsqueda) y **Grafo**; la única *estructura de datos externa* de este paquete (reside en memoria externa como el disco duro de la computadora o una memoria USB) es la clase **Archivo**. Los siguientes diagramas representan los *paquetes de uso común*:

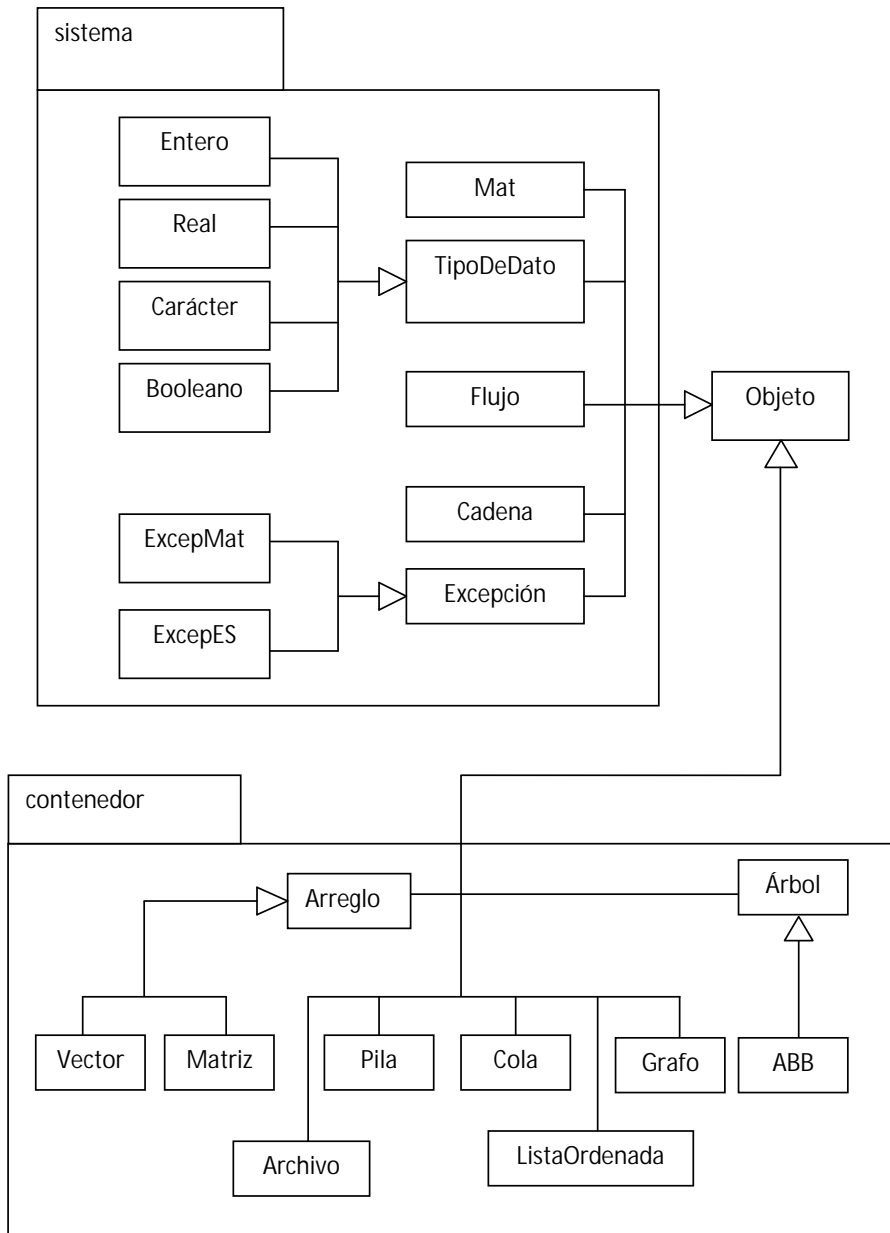


Figura 2.7. Paquetes de uso común

Observación: todas las clases contenidas en los paquetes de uso común se desligan de la *clase Objeto*, siendo ésta la clase superior de la jerarquía.

2.5. CONFIGURACIÓN DE LAS CLASES DE USO COMÚN

Cada clase de uso común se configura con los atributos y operaciones que le caracterizan, lo que permite solucionar ciertos problemas de manera rápida con simples mecanismos de reutilización.

A continuación se presenta la configuración de las clases Objeto, TipoDeDato, Flujo, Cadena y Mat, con la correspondiente descripción de sus atributos.

LA CLASE OBJETO

La *clase Objeto* es la raíz de todo el árbol de la jerarquía de clases uso común, y proporciona cierto número de métodos de utilidad general que pueden utilizar todos los objetos.

Objeto
+ comparar (Objeto x): entero + aCadena (): cadena + aEntero(): entero + aReal(): real + aCaracter(): caracter + obtenerClase ()

Métodos de la clase:

Tabla 2.8. Métodos de la clase Objeto

MÉTODO	DESCRIPCIÓN
comparar(Objeto x) : entero	<p>Compara objetos entre sí. Esta comparación no es la misma que proporciona el operador <code>==</code>, que solamente se cerciora si dos referencias a objetos apuntan al mismo objeto</p> <p>El método <i>comparar()</i> se usa para saber si dos objetos separados son del mismo tipo y contienen los mismos datos. El método devuelve un valor entero que puede ser:</p> <ul style="list-style-type: none"> • igual a cero, si los objetos son iguales • menor que cero, si el objeto actual es menor que el objeto <i>x</i> • mayor que cero, si el objeto actual es mayor que el objeto <i>x</i>
aCadena() : cadena	Convierte un objeto en una cadena. Aquí, el método <i>aCadena()</i> extrae el entero contenido en un objeto <i>Entero</i> y lo retorna como una cadena
aEntero() : entero	Convierte un objeto en un número entero
aReal() : real	Convierte un objeto en un número real
aCaracter() : caracter	Convierte un objeto en un carácter
obtenerClase()	Devuelve la clase de un objeto. Este método se utiliza para determinar la clase de un objeto, es decir, devuelve un objeto de tipo <i>Clase</i> , que contiene información importante sobre el objeto que crea la clase. Una vez determinada la clase del objeto, se pueden utilizar sus métodos para obtener información acerca del objeto

LA CLASE TIPODEDATO

TipoDeDato es una clase abstracta utilizada para definir, mediante los mecanismos de herencia, las clases asociadas a los tipos de datos estándar.

TipoDeDato <<abstracta>>
+ asignarValor() + obtenerValor()

Por polimorfismo, las clases **Entero**, **Real**, **Caracter** y **Logico** deben implementar los métodos `asignarValor ()` y `obtenerValor ()`. Si una clase es abstracta, sus métodos también lo son.

LA CLASE FLUJO

La *clase Flujo* ofrece los servicios de ingreso de datos desde el dispositivo estándar de entrada (teclado) y visualización de información en el dispositivo estándar de salida (monitor).

Flujo
+ estatico leerCadena(): cadena + estatico leerEntero(): entero + estatico leerReal(): real + estatico leerCaracter(): Caracter + estatico leerObjeto(): Objeto + estatico imprimir(<lista_de_argumentos>)

Tabla 2.9. Métodos de la clase Flujo

MÉTODO	DESCRIPCIÓN
<code>leerCadena() : cadena</code>	Lee una cadena de caracteres desde el teclado.
<code>leerEntero() : entero</code>	Lee un número entero.
<code>leerReal() : real</code>	Lee un número real.
<code>leerCaracter() : caracter</code>	Lee un carácter.
<code>leerObjeto() : objeto</code>	Lee un objeto. Este método puede reemplazar a cualquiera de los anteriores.
<code>imprimir()</code>	Visualiza en pantalla el contenido de los argumentos. La lista de argumentos puede ser de cualquier tipo primitivo, objetos, variables o constantes numéricas y de cadena, separadas por el símbolo de concatenación cruz (+).

Algunos ejemplos de instrucciones de lectura y escritura donde se utiliza la clase **Flujo**, se presentan en el siguiente segmento de seudo código:

```
entero cod
real precio
cadena nombre

Flujo.imprimir ("Ingrese el código, nombre y precio del artículo:")
cod = Flujo.leerEntero()
precio = Flujo.leerReal()
nombre = Flujo.leerCadena()
Flujo.imprimir ("codigo: " + cod + "Nombre: " + nombre + "Precio: $" + precio)
```

Obsérvese el uso de los métodos estáticos de la clase **Flujo**: **imprimir()**, **leerEntero()**, **leerReal()** y **leerCadena()**.

LA CLASE CADENA

La *clase* **Cadena** contiene las operaciones básicas que permiten administrar cadenas de caracteres en un algoritmo.

Cadena
<ul style="list-style-type: none"> - cad []: caracter - longitud: entero - const estatico entero MAX_LONG = 256
<ul style="list-style-type: none"> + Cadena () + asignarCad (Cadena x) + obtenerCad (): Cadena + car (entero x): caracter + num (carácter x): entero + valor (Cadena x): entero + valor (Cadena x): real + convCad (entero x): Cadena + convCad (real x): Cadena + estatico comparar (Cadena c1, Cadena c2): entero

La clase **Cadena** contiene tres miembros dato:

cad: arreglo unidimensional de tipo **caracter**.

longitud: cantidad de caracteres de la cadena (equivale al número de elementos o tamaño del vector *cad*).

MAX_LONG: es una constante entera y estática que determina la longitud máxima de la cadena, dada por 256 caracteres.

Descripción de los métodos:

Tabla 2.10. Métodos de la clase Cadena

MÉTODO	DESCRIPCIÓN
Cadena ()	Constructor que inicializa el atributo <i>cad</i> con la cadena vacía.
asignarCad (Cadena x)	Asigna el objeto <i>x</i> al objeto mensajero.
obtenerCad(): Cadena	Retorna el contenido del miembro privado <i>cad</i> .
car (entero x): caracter	Devuelve el código ASCII correspondiente al entero <i>x</i> .
num (caracter x): entero	Devuelve el número entero asociado a un carácter ASCII.
valor (Cadena x): entero	Convierte una cadena en un valor numérico entero.
convCad (entero x): Cadena	Convierte un entero en una cadena.
comparar(Cadena c1, Cadena c2): entero	Compara la cadena <i>c1</i> con la cadena <i>c2</i> . Devuelve 0 si las cadenas son iguales, -1 si $c1 < c2$ y 1 si $c1 > c2$.

LA CLASE MAT

Esta clase, carente de atributos, contiene una serie de métodos estáticos (funciones matemáticas) susceptibles de invocar desde cualquier algoritmo, sin la necesidad de pasar un mensaje a un objeto determinado.

Mat
<ul style="list-style-type: none"> + estatico abs (real x): real + estatico abs (entero x): entero + estatico parteEntera (real x): entero + estatico truncar (real x): entero + estatico redondear (real x): entero + estatico aleatorio (entero x): entero + estatico aleatorio (entero x): real + estatico exp (entero x): real + estatico ln (real x): real + estatico ln (entero x): real + estatico log (real x): real + estatico log (entero x): real + estatico elevar (entero b, entero p): entero + estatico elevar (entero b, real p): real + estatico elevar (real b, real p): real + estatico elevar (real b, entero p): real + estatico seno (real x): real + estatico seno (entero x): real + estatico coseno (real x): real + estatico coseno (entero x): real + estatico arcTan (real x): real + estatico arcTan (entero x): real + estatico tipo_base (<i>expresión</i>): <i>expresión</i>

Algunas de las operaciones de la clase `Mat` están sobrecargadas; `abs(real)` y `abs(entero)` hallan el valor absoluto de un número real y el valor absoluto de un número entero, respectivamente. Todos sus métodos son estáticos, por tanto se consideran métodos de clase, es decir, siempre serán invocados de la siguiente manera:

`Mat.nombreMétodo()`, como en `Mat.elevar(4, 3)` o en `Mat.redondear(3.7)`

En lo que sigue se explican las funciones de cada método estático. Debe aclararse que se omiten las operaciones sobrecargadas:

Tabla 2.11. Métodos de la clase Mat

MÉTODO	DESCRIPCIÓN
abs (real x): real	Valor absoluto del número x. Devuelve un número real.
parteEntera (real x): entero	Valor entero de x (se redondea al entero menor).
truncar (real x): entero	Valor entero de x (se truncan los decimales).
redondear (real x): entero	Redondea el número x.
aleatorio (entero x): entero	Número aleatorio entre 0 y $x - 1$. Si x es igual a 1, se genera un número aleatorio entre 0.0 y 0.9 (se ejecuta la función sobrecargada).
exp (entero x): real	Exponencial de un número (e^x).
ln (real x): real	Logaritmo neperiano de x (base e).
log (real x): real	Logaritmo decimal de x (base 10).
eleva (entero b, entero p): real	Eleva el entero b a la potencia p.
seno (real x): real	Seno del ángulo x.
coseno (real x): real	Coseno del ángulo x.
arcTan (real x): real	Arco tangente del ángulo x
tipo_base(expresión)	Convierte la expresión en el tipo base especificado.

2.6. PROBLEMAS RESUELTOS CON LAS CLASES DE USO COMÚN

Las *clases de uso común* permiten solucionar un vasto conjunto de problemas que van desde el procesamiento de cadenas, manejo de excepciones y cálculos matemáticos, hasta casos que requieran del uso de tipos de datos estándar como clases. En las páginas que siguen, el lector encontrará dos problemas resueltos por medio de estas clases, cada uno ahorra trabajo al reutilizar componentes de software.

PROBLEMA 3. CLASES DE USO COMÚN

Leer dos números enteros n_1 y n_2 . Visualizar:

- El equivalente octal de n_1 .
- El equivalente hexadecimal de n_2 .
- El resultado de n_1 elevado a la potencia n_2 .

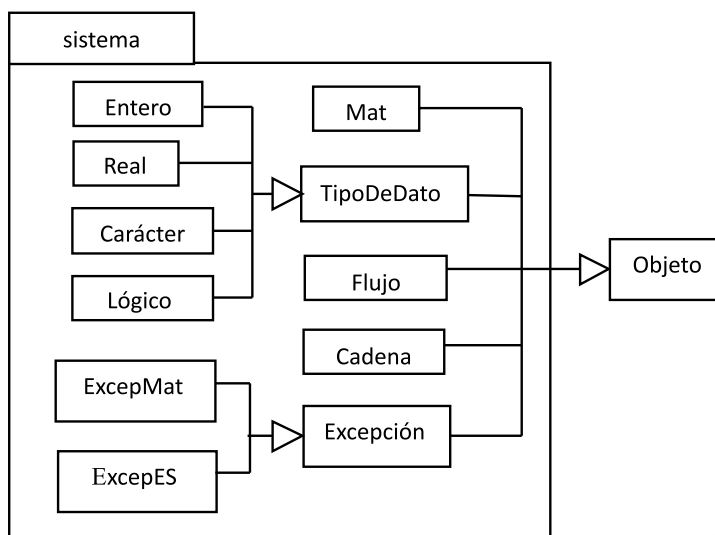
Solución:

a) Tabla de requerimientos funcionales

Id. del requerimiento	Descripción	Entrada	Salida (Resultado)
R1	Ingresar dos números enteros	Dos números enteros digitados por El usuario.	Los números ingresados están almacenados en las variables n1 y n2.
R2	Hallar el equivalente octal de n1	El número n1	El equivalente octal de n1
R3	Hallar el equivalente hexadecimal de n2	El número n2	El equivalente hexadecimal de n2
R4	Encontrar el resultado de una potencia.	Los números n1 y n2	El resultado de elevar n1 a la n2ava potencia

Para suplir todos los requerimientos especificados, se requiere de algunas clases de uso común almacenadas en el paquete *sistema*. En concreto, se reutilizarán las clases *Entero*, *Mat*, y *Flujo*.

Recordemos el paquete de clases de uso común *sistema*:



Observaciones:

- Este se importa por defecto, es decir, es innecesario escribir la línea:
importar sistema
- Los *métodos estáticos* a reutilizar son **Flujo.leerEntero()** y **Mat.elevar()** y los no estáticos son **Entero.convOctal()** y **Entero.convHex()**. La estructura completa de estas clases se presentó en los apartados 2.4. (*Clases de uso común: un único mundo para efectos de reutilización*) y 2.5. (*Configuración de las clases de uso común*). La estructura parcial de las clases **Entero**, **Mat** y **Flujo**, según los métodos reutilizados en este problema, se presenta en las figuras 2,8 a 2,10.

Entero
- valor: entero
+ Entero () + asignarValor (entero x) + obtenerValor (): entero + convOctal (): Cadena + convHex (): Cadena

Figura 2.8: Estructura parcial de la clase Entero

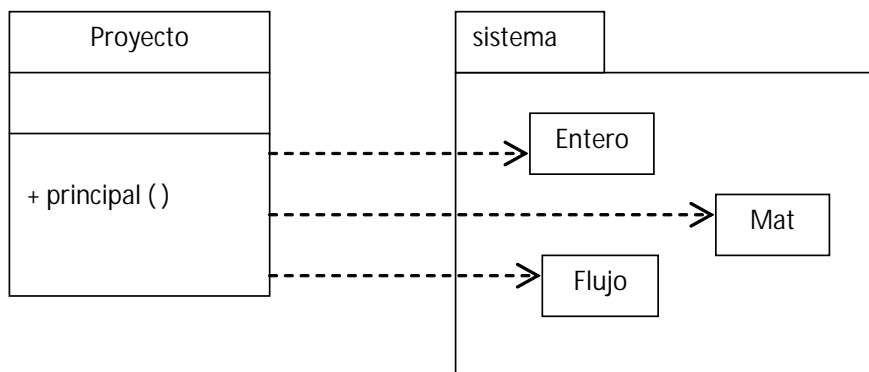
Mat
+ estatico elevar (entero b, entero p): entero

Figura 2.9: Vista de la clase Mat

Flujo
+ estatico leerEntero (): entero + estatico imprimir (<lista_de_argumentos>)

Figura 2.10: Vista de la clase Flujo

b) Diagrama de clases:



c) Responsabilidades de las clases

Las responsabilidades de las clases **Entero**, **Flujo** y **Mat** se presentan mediante contratos parciales, donde sólo se describen los métodos utilizados en la solución del problema. Aquí se omite, de manera deliberada, la descripción de los demás métodos, útiles en la solución de problemas adicionales. En las tablas que siguen, además de los métodos y de los requerimientos asociados con los contratos de las clases, se definen sus precondiciones y poscondiciones.

Contrato parcial de la *clase Entero*:

Método	Requerimiento asociado	Precondición	Postcondición
Entero()	R1	No existe un objeto de la clase Entero .	Existe un objeto de la clase Entero listo para ser procesado.
asignarValor()	R1	El atributo <i>valor</i> de la clase Entero es igual a cero.	El atributo <i>valor</i> de la clase Entero contiene una cantidad especificada por el usuario.

obtenerValor()	R2, R3 y R4	El usuario desconoce el contenido de la variable privada <i>valor</i> .	El usuario conoce el contenido de la variable privada <i>valor</i> .
convOctal()	R2	El usuario desconoce el equivalente octal del entero ingresado.	El usuario conoce el equivalente octal del entero ingresado.
convHex()	R3	El usuario desconoce el equivalente hexadecimal del entero ingresado.	El usuario conoce el equivalente hexadecimal del entero ingresado.

Contrato parcial de la *clase Flujo*:

Método	Requerimiento asociado	Precondición	Postcondición
imprimir()	R1, R2, R3 y R4	No se ha visualizado en la salida estándar un mensaje o resultado.	Se ha visualizado un mensaje o resultado en la salida estándar.
leerEntero()	R1	No se ha capturado desde el teclado, los datos ingresados por el usuario.	La entrada del usuario ha sido almacenada en memoria.

Contrato parcial de la *clase Mat*:

Método	Requerimiento asociado	Precondición	Postcondición
eleva()	R4	No se ha calculado la operación de potenciación.	La operación de potenciación se ha efectuado.

Contrato de la clase Proyecto:

Método	Requerimiento asociado	Precondición	Postcondición
principal	R1 R2 R3 R4	No hay entradas de datos y no se ha efectuado proceso alguno.	Se tienen dos objetos de tipo Entero y se conoce el equivalente octal del atributo <i>valor</i> del primer objeto, el equivalente hexadecimal del atributo <i>valor</i> del segundo objeto y el resultado de elevar el miembro dato del primer objeto al miembro dato del segundo objeto.

d) Seudo código

Clase Proyecto

publico principal()

Entero n1 = nuevo Entero()

Entero n2 = nuevo Entero()

Flujo.imprimir("Ingrese el primer número entero:")

n1.asignarValor(Flujo.leerEntero())

Flujo.imprimir ("Ingrese el segundo número entero:")

n2.asignarValor(Flujo.leerEntero())

Flujo. imprimir ("El equivalente octal de " + n1.obtenerValor() +
" es " + n1.convOctal())

Flujo.imprimir ("El equivalente hexadecimal de " + n2.obtenerValor() +
" es " + n2.convHex())

Flujo.imprimir (n1.obtenerValor() + " ^ " + n2.obtenerValor() + " = " +
Mat.elevar(n1, n2))

fin_método

fin_clase

PROBLEMA 4. CONVERSIÓN DE UNA CONSTANTE A DIFERENTES BASES NUMÉRICAS

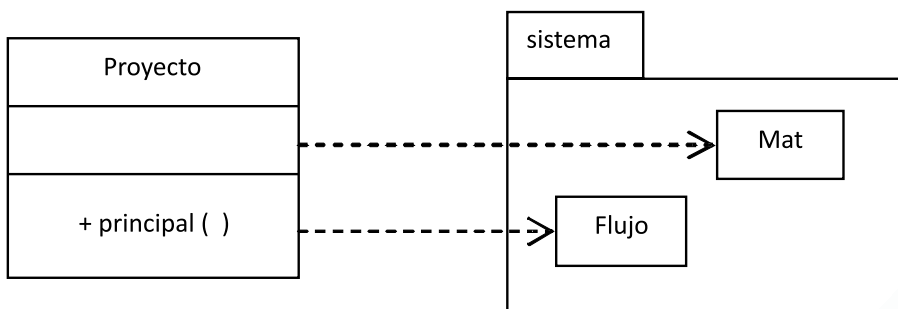
Convertir el número arábigo 74 a base binaria, octal y hexadecimal.

Solución:

a) Requerimientos funcionales

Identificación del requerimiento	Descripción	Entradas	Resultado (Salidas)
R1	Convertir el número 74 a base binaria.	Ninguna	Equivalente binario de 74
R2	Convertir el número 74 a base octal.	Ninguna	Equivalente octal de 74
R3	Convertir el número 74 a base hexadecimal.	Ninguna	Equivalente hexadecimal de 74

b) Diagrama de clases



c) Responsabilidades de las clases

Contrato parcial de la clase Flujo:

Método	Requerimiento asociado	Precondición	Postcondición
imprimir ()	R1 R2 R3	No se han visualizado en pantalla los equivalentes en binario, octal y hexadecimal del número 74	Se han visualizado en pantalla los equivalentes en binario, octal y hexadecimal del número 74

Contrato parcial de la clase Mat:

Método	Requerimiento asociado	Precondición	Postcondición
convBinario ()	R1	Se desconoce el equivalente binario del número decimal 74	Se ha calculado y se conoce el equivalente binario del número 74.
convOctal ()	R2	Se desconoce el equivalente octal del número decimal 74	Se ha calculado y se conoce el equivalente octal de 74
convHex()	R3	Se desconoce el equivalente hexadecimal del número 74	Se ha calculado y se conoce el equivalente hexadecimal de 74.

d) Seudo código

```
clase Proyecto
  publico principal()
    Flujo. imprimir ("El equivalente binario de 74 es " + Entero.
                    convBinario(74 ))
    Flujo. imprimir ("El equivalente octal de 74 es " + Entero.
                    convOctal(74 ))
    Flujo. imprimir ("El equivalente hexadecimal de 74 es " +
                    Entero.convHex(74 ))
  fin_método
fin_clase
```

Observaciones:

- Los contratos de las *clases de uso común* se encuentran establecidos por omisión; los presentados aquí corresponden a un problema particular.
- Los métodos *convBinario()*, *convOctal()* y *convHex()* de la clase **Entero**, están sobrecargados. En este caso, se han utilizado las versiones estáticas de estos métodos, mientras que en el problema 3 se utilizaron versiones no estáticas de los métodos *convOctal()* y *convHex()*.

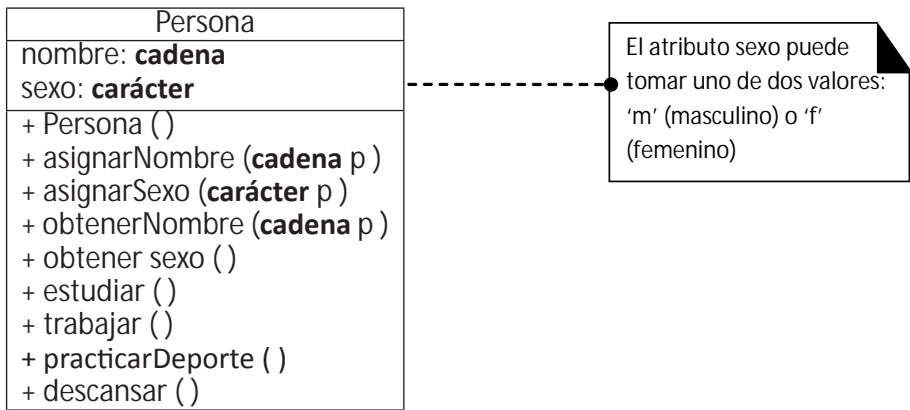
2.7. EJERCICIOS PROPUESTOS

1. La clase Archivo se encuentra definida así:

Archivo
nombre: cadena tipo: cadena ubicación: cadena tamaño: real fechaCreación: cadena fechaUltimaModificación: cadena
+ Archivo () + Archivo (archivo p) + asignarNombre (cadena p) + asignarTipo (cadena p) + asignarUbicación (cadena p) + asignarTamaño (real p) + asignarFechaCreación (cadena p) + asignarFechaUModif (cadena p) + obtenerNombre (): cadena + obtenerTipo (): cadena + obtenerUbicación (): cadena + obtenerTamaño (): real + obtenerFechaCreación (): cadena + obtenerFechaUModif (): cadena + crear() + editar() + guardar() + cerrar() + abrir() + renombrar () + enviarA () + copiar () + cortar () + eliminar () + crearAccesoDirecto ()

Plantee problemas con la clase *Archivo* y soluciónelos a partir de la interfaz pública de la clase.

2. La clase *Persona* se encuentra definida así:



Crear cuatro personas: tres hombres llamados Arcadio Buendía, Juan Valdés y Pedro Páramo y una mujer llamada Flora Flores. Hacer que Arcadio estudie, Juan trabaje, Pedro practique deporte, estudie y trabaje, y Flora practique deporte. A su vez, mostrar los mensajes respectivos cuando cada persona desempeñe una actividad; puede ser de la siguiente manera: cuando Arcadio realice su actividad se debe imprimir el mensaje “Arcadio está estudiando”. Al final del proceso todas las personas deben descansar.

3. Convertir a base binaria, octal y hexadecimal:
 - 3.1 El número entero 457.
 - 3.2 El número real 98.3
 - 3.3 Un número entero ingresado por el usuario.
4. Crear un objeto de la clase *Entero*, asignarle un valor establecido por el usuario. Visualizar dicho valor y su raíz cuadrada.
5. Imprimir los máximos y mínimos valores para los tipos de datos *real* y *entero*.
6. Leer un carácter y especificar si corresponde a un dígito o a una letra.
7. Leer dos cadenas de caracteres e indicar si son iguales o diferentes.

8. Leer un número entero. Imprimir su seno, tangente y cotangente.
9. Ingresar dos números, el primero entero y el segundo real. Imprimir: el primero elevado al segundo y el resultado de redondear el segundo.
10. Generar dos números aleatorios de tipo real. Imprimir los valores generados, así como ambos valores truncados y redondeados.
11. Desarrollar el seudo código de los siguientes métodos, asociados a *clases de uso común*:
 - sistema.Entero.asignarValor (entero x)
 - sistema.Entero.esIgual (entero x): **booleano**
 - sistema.Entero.convOctal (): **cadena**
 - sistema.Entero.convHex (): **cadena**
 - sistema.Carácter.esEspacio (): **booleano**
 - sistema.Cadena.comparar (cadena c1, cadena c2): **entero**
 - Todos los métodos de la clase sistema.Mat
12. Teniendo en cuenta el paquete de uso común *sistema*, escriba métodos Proyecto.principal() que permitan:
 - Leer un número entero y convertirlo a binario, octal y hexadecimal. Mostrar las tres equivalencias.
 - Leer una frase y retornarla convertida a mayúsculas.

2.8. REFERENCIAS

[Booch1999]: Booch-Rumbaugh-Jacobson (1999). El Lenguaje Unificado de Modelado. Addison Wesley Iberoamericana, Madrid.

[RAE2001]: Real Academia Española (2001). Diccionario de la Lengua Española. Vigésima segunda edición. Editorial Espasa Calpe. S.A.

[Zapata1998]: Zapata, Juan Diego. (1998). “El Cuento” y su papel en la enseñanza de la orientación por objetos. IV Congreso RIBIE, Brasilia. <http://www.c5.cl/ieinvestiga/actas/ribie98/146.html>. Consultado en agosto de 2008.

[Zi2007]: Zi, Lie. La Sospecha (2007). http://www.juecesyfiscales.org/cuentos_cortos.htm. Consultado en noviembre de 2007.

[GIISTA2006]: GIISTA (Grupo de Investigación en Ingeniería de Software del Tecnológico de Antioquia). Fundamento de Programación con orientación a objetos. Begón Ltda, Medellín, 2006.

CAPÍTULO 3

Clases: tipos de datos abstractos

- 3.1. Estructura general de una clase
 - 3.2. Métodos
 - Definición de un método
 - Invocación de un método
 - Métodos estáticos
 - Problema 5: Operaciones con un número entero
 - Paso de parámetros por valor vs. Paso de parámetros por referencia
 - 3.3. Sobrecarga de métodos
 - Problema 6: Empleado con mayor salario
 - Problema 7: Informe sobrecargado
 - 3.4. Estructuras de control
 - La estructura secuencia
 - La estructura selección
 - Problema 8: Producto más caro
 - Problema 9: Estadísticas por procedencia
 - La estructura iteración
 - Problema 10: Primeros cien naturales
 - 3.5. Bandera o interruptor
 - Problema 11: Sucesión numérica
 - 3.6. Métodos recursivos
 - Problema 12: Factorial de un número
 - Problema 13: Cálculo de un término de Fibonacci
 - 3.7. Ejercicios propuestos
 - 3.8. Referencias
-

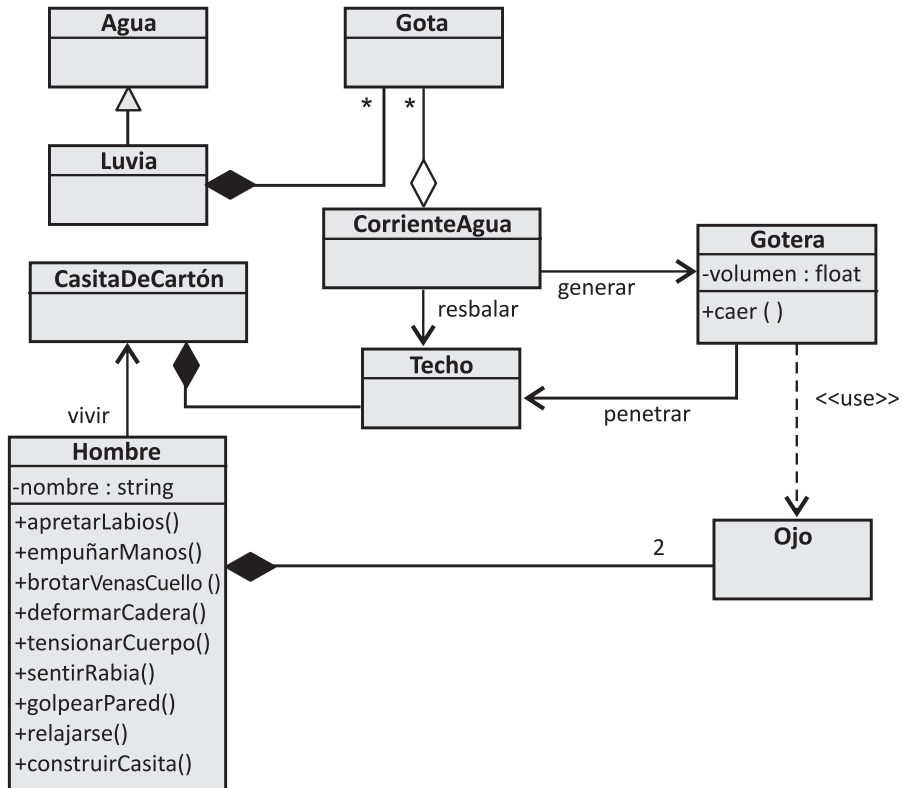
RABIA

Diego Aristizábal, 2007

Esa noche, por culpa de una gotera que le caía en un ojo, el hombre se despertó con los labios apretados, con las manos empuñadas, con las venas brotadas en el cuello, con la cadera deformada, con la voz iracunda insultándose por dentro, con el cuerpo tensionado y con tanta rabia que pensó que golpearía por primera vez una pared, pero no lo hizo. Si lo hacía, tendría que relajarse al instante para construir de nuevo su casita de cartón.



Diagrama de clases:



O

bjetivos de aprendizaje

- Familiarizarse con las estructuras de control selectivas y repetitivas.
- Plantear soluciones con métodos estáticos o métodos de clase.
- Construir la tabla de requerimientos, el diagrama de clases, las responsabilidades de las clases (expresadas con los contratos de los métodos) y el seudo código, para una serie de problemas de carácter convencional.
- Resolver problemas utilizando métodos recursivos.

3.1. ESTRUCTURA GENERAL DE UNA CLASE

Como se ha notado en los cuatro problemas tratados hasta el momento, una *clase* se puede definir en seudo código o de forma gráfica utilizando los formalismos de UML. La representación en seudo código para una *clase* incluye algunos elementos excluyentes entre sí (la visibilidad de la clase, por ejemplo) y otros opcionales (visibilidad, herencia e interfaz), como se observa a continuación:

```
[abstracto | final] [público | privado | protegido] clase nomClase [heredaDe
                                nomClaseBase] [implementa nomInterfaz]
    // Cuerpo de la clase
fin_clase
```

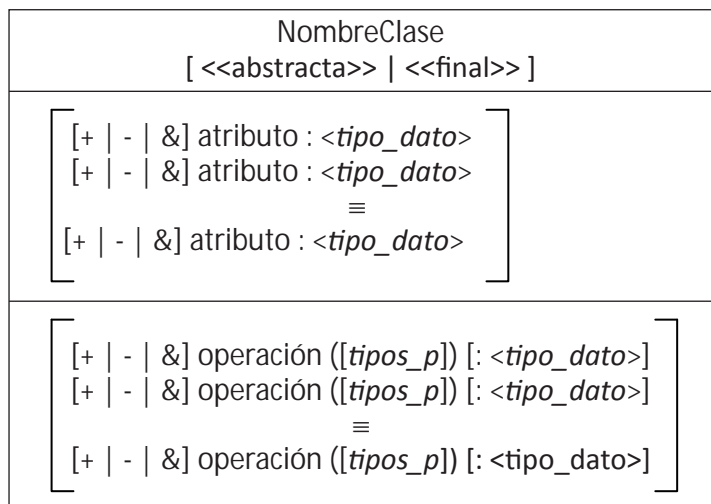
El cuerpo de la clase incluye dos secciones: una dedicada a los *atributos* y otra dedicada a las *operaciones*. Ambas partes son opcionales, lo que permite, en términos de software, definir una clase sin atributos o sin operaciones. Esta característica no concuerda con la realidad, pues tanto atributos como operaciones son inherentes a la naturaleza de las clases.

El *cuerpo de una clase* tiene la siguiente configuración:

```
[público | privado | protegido]:
    atributo : <tipo_dato>
    atributo : <tipo_dato>
    ≡
    atributo : <tipo_dato>

[público | privado | protegido]:
    operación ([tipos_p]) [: <tipo_dato>]
    operación ([tipos_p]) [: <tipo_dato>]
    ≡
    operación ([tipos_p]) [: <tipo_dato>]
```

De igual manera y para efectos de un curso introductorio de fundamentos de programación, la estructura parcial de una clase en UML es la siguiente:



Observaciones:

- Para ambos casos -representación en pseudo código y en UML-, lo que se encierra entre corchetes es opcional. El carácter barra vertical (|) significa que se debe escoger una opción entre las disponibles, por tanto, una clase se declara pública, privada o protegida, mientras que una *clase abstracta* nunca puede ser *final*.

- El orden en la declaración de atributos y operaciones no interesa. Sin embargo, según la sintaxis de UML, se deben especificar primero los atributos y luego las operaciones.
- Cada elemento sintáctico propone estos significados:

público, privado, protegido: hace referencia al *especificador de acceso, alcance o visibilidad* de la clase, atributo u operación.

La visibilidad según UML se representa con los signos + (público), - (privado) y & (protegido).

Si no se indica la visibilidad, se asume pública.

abstracta: denota a una *clase abstracta*, aquella desde la cual no se pueden instanciar objetos y que permite definir otras clases que heredan de ella.

final: indica que la clase finaliza la jerarquía, es decir, no se pueden definir clases derivadas a partir de ella.

clase: palabra reservada que permite definir una nueva clase.

- **NombreClase:** es el nombre de un *tipo abstracto de datos*. Por convención el nombre de una clase inicia con una letra mayúscula y las restantes con minúsculas. Para el caso de un nombre de clase compuesto por varias palabras, se puede optar por el uso de guión como separador de palabras o concatenar todo el identificador comenzando toda nueva palabra con una letra mayúscula. Ejemplos:
Empleado, EmpleadoOficial, Empleado_oficial
- **atributo:** nombre válido de un atributo de clase. Siempre comienza con una letra, puede incluir dígitos, letras y carácter de subrayado, no admite espacios y nunca puede coincidir con una palabra reservada.
- **<tipo_dato>:** corresponde a un tipo de datos estándar o a un tipo abstracto de datos.

Se tendrán en cuenta cinco tipos de datos estándar: **entero**, **real**, **carácter**, **cadena** y **lógico**. Todo atributo siempre debe tener un tipo asociado. Sin embargo, en una operación se puede omitir el tipo si ésta no retorna valor.

- **operación()**: corresponde a una función o procedimiento que realiza algún tipo de operación con los atributos de la clase. Las *operaciones* son los algoritmos que se deben desarrollar para darle funcionalidad a la clase.
- **tipos_p**: lista de tipos de datos correspondiente a los *parámetros*. Aquí se debe observar que no interesa el nombre del *argumento*, pero sí el tipo. Los nombres de los argumentos cobran relevancia al momento de definir la operación.
- Si las operaciones de una clase se definieran en línea, estilo lenguaje Java, la expresión [+ | - | &] operación ([argumentos]) : [<tipo_dato>], se conoce como *declaración del método* o *definición de prototipo*.

3.2. MÉTODOS

Un *método* es un pequeño programa diseñado para efectuar determinada tarea en la solución de un problema. En el problema 2, se identificaron los métodos *asignarX()*, *obtenerY()*, *dibujar()*, y *principal()*, este último de carácter inamovible y obligatorio para toda solución planteada, pues constituye el punto de entrada y salida de la aplicación.

El término *subprograma* es propio del paradigma de programación imperativo o estructurado, y es comúnmente utilizado cuando hay independencia entre los datos y los algoritmos que los tratan. En programación orientada a objetos, el término subprograma es reemplazado por *método*, donde los datos y los algoritmos se integran en un componente denominado *objeto*.

Los subprogramas, y por tanto los métodos, se clasifican en procedimientos y funciones. Un *procedimiento* puede o no tener parámetros y puede retornar cero, uno o

más valores al método que lo invocó. Una *función* tiene cero, uno o más parámetros y siempre retorna un valor.

DEFINICIÓN DE UN MÉTODO

La definición de un método incluye la especificación de *visibilidad*, *tipo de retorno*, *estereotipado*, *nombre del método*, *parámetros*, *cuerpo* y *finalización*.

```
[público | privado | protegido] [<tipo_dato>] [estático]
    nombre_método ([lista_parámetros])
        // Instrucciones o cuerpo del método
    [retornar(valor)]
fin_método
```

Observaciones:

- Lo que se encuentra encerrado entre corchetes es opcional. El carácter barra vertical o pipe (|) indica que se debe seleccionar entre una de las opciones; así, una clase es pública, privada o protegida.
- Las palabras reservadas **público**, **privado** y **protegido** hacen referencia al *especificador de acceso* o *visibilidad* de la operación. Si no se indica la visibilidad se asume que es pública.
- *<tipo_dato>* corresponde al *tipo de retorno* (tipo del resultado devuelto por el método), el cual puede corresponder con un tipo de datos estándar o un tipo abstracto de datos.

Se tendrán en cuenta los cinco tipos de datos estándar: **entero**, **real**, **carácter**, **cadena** y **lógico**. Si no se especifica el tipo de retorno, se asume vacío.

- La palabra *estático* indica que el método así lo es.
- *nombre_método* es cualquier identificador válido para el nombre de un método, con las siguientes restricciones:

- No puede coincidir con una palabra reservada (ver el listado de las mismas en el apéndice B, ítem B1).
- Debe comenzar con una letra o con el carácter de subrayado. No se admiten comienzos con dígitos o caracteres especiales.
- No puede contener espacios.
- La cantidad de caracteres del identificador es irrelevante, aunque se aconseja el uso de identificadores con una longitud que no exceda los 15 caracteres.
- El nombre del método debe conservar la mnemotecnia adecuada, es decir, recordar en términos generales la tarea que realiza.

Algunos identificadores inválidos son:

3rValor (comienza con un dígito), para (coincide con una palabra reservada del seudo lenguaje), #TotalPctos (comienza con un carácter especial no permitido), cuenta corriente (incluye un espacio).

Por el contrario, los siguientes identificadores son correctos:

valorFinal, PI, direccionOficina, promedio_anual, mes7.

- Una operación que emule a una función debe incluir la sentencia **retornar**. Si la operación señala a un procedimiento, no se incluye dicha sentencia.
- *lista_parámetros* es una lista de variables u objetos separados por comas, con sus tipos respectivos. Los *parámetros* permiten la transferencia de datos entre un objeto perteneciente a la clase y el mundo exterior; forman parte de la interfaz pública de la clase. Una de las condiciones para aquella transferencia de datos es que en el momento del *paso de un mensaje* al objeto exista una relación uno a uno con los *argumentos* en cuanto a orden, número y tipo.
- El *cuerpo del método* constituye su desarrollo algorítmico; puede incluir declaración de variables, constantes, objetos, registros, tipos enumerados y sentencias de control para toma de decisiones e iteración de instrucciones.
- La instrucción **retornar** (*valor*) es opcional. Si el método tiene algún tipo de retorno, la variable local *valor* debe ser del mismo tipo que *<tipo_dato>*, es decir, el tipo de retorno del método.

- **fin_método** es una palabra reservada que indica la *finalización del método*; la palabra *método* se puede escribir sin tilde.

INVOCACIÓN DE UN MÉTODO

La *invocación de un método* implica adaptarse a la siguiente sintaxis:

```
[variable = ] nombre_objeto.nombre_método([lista_parámetros_actuales])
```

Observaciones:

- Si el método retorna un valor, este se debe asignar a una variable.
- Se debe tener en cuenta que la lista de parámetros actuales debe coincidir con la lista de *argumentos* dada en la declaración de prototipo de la clase.

Ejemplo:

Suponga que la función *factorial()* y el procedimiento *mostrarVector()* están definidos como métodos no estáticos de la clase *Arreglo* como se indica a continuación:

```
entero factorial (entero n)
    entero f = 1, c
    para (c = n, c > 1, c = c - 1)
        f = f * c
    fin_para
    retornar f
fin_método
// -----
vacío mostrarVector(real vec[ ])
    entero i = 0
    mientras (i < vec.longitud)
        Flujo.imprimir(vec[i])
        i = i + 1
    fin_mientras
fin_método
// -----
```

Posibles invocaciones para cada uno de estos métodos serían:

- Invocación dentro de otro método de la clase *Arreglo*:

```
// Factorial de 7 y de un entero ingresado por el usuario
entero x7 = factorial(7)
entero n = Flujo.leerEntero( )
Flujo.imprimir("El factorial de 7" + " es " + x7)
Flujo.imprimir("El factorial de " + n + " es " + factorial(n))

//...

real valores[ ] = {1.2, 5.7, -2.3, 4.0, 17.1, 2.71}
mostrarVector(valores)
```

- Invocación dentro de un método que no pertenece a la clase *Arreglo*:

```
//Creación de un objeto de la clase Arreglos
Arreglo x = nuevo Arreglo ( )
Flujo.imprimir("El factorial de 7 es " + x.factorial(7))
real valores[ ] = {1.2, 5.7, -2.3, 4.0, 17.1, 2.71}
Flujo.imprimir("Las " + valores.longitud + " constantes reales son:")
x.mostrarVector(valores)
```

MÉTODOS ESTÁTICOS

Un *método estático*, denominado también *método de clase*, no requiere de una instancia de objeto para ejecutar la acción que desarrolla, puesto que tiene la semántica de una función global.

La sintaxis general para invocar a un método estático es la siguiente:

NombreDeClase.métodoEstático([parámetros])

Ejemplos: los métodos *imprimir()* y *leerReal()* son estáticos porque se invocan directamente desde la clase **Flujo**; de igual manera, el método *seno()* es estático para la clase **Mat**.

```
Flujo.imprimir ("Buena noche")
estatura = Flujo.leerReal()
x = Mat.seno(30)
```

PROBLEMA 5. OPERACIONES CON UN NÚMERO ENTERO

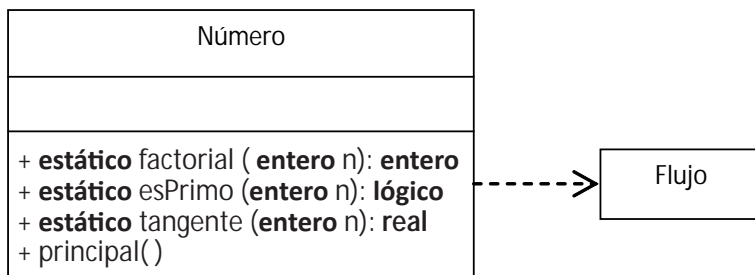
Ingresar un número entero positivo o cero. Imprimir su factorial, determinar si corresponde a un número primo y hallar la tangente del ángulo que representa.

Solución:

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Ingresar un número entero mayor o igual a cero	Un número entero mayor o igual a cero, digitado por el usuario	Número entero almacenado en una variable de memoria (<i>num</i>)
R2	Conocer el factorial del número del número ingresado	El número <i>num</i>	El factorial de <i>num</i>
R3	Determinar si el número ingresado corresponde o no a un número primo	El número <i>num</i>	Un mensaje que indica si el número <i>num</i> es o no primo
R4	Encontrar la tangente del ángulo dado por <i>num</i>	El número <i>num</i>	La tangente de <i>num</i>

b) Diagrama de clases



Vale aclarar que los métodos *factorial()*, *esPrimo()* y *tangente()* son de naturaleza estática. De esta manera., un *método estático* puede ser invocado desde cualquier otro método estático sin la necesidad de comprometer objeto alguno. Estos tres métodos serán llamados por el método *principal()*, también de naturaleza estática aunque no se haga explícito con la palabra reservada, dando solución al problema planteado.

El orden de los métodos es irrelevante, siempre y cuando se defina el método *principal()* que, como se ha indicado, es de carácter obligatorio en toda aplicación. En este texto, si un problema conlleva el desarrollo de varios métodos, el *principal()* se definirá siempre de último.

c) Contrato de la clase *Número*

Nombre del método	Req. aso_ ciado	Precon_ dición	Postcondición	Modelo verbal
factorial (num)	R2	Se desconoce el factorial de un número entero <i>num</i> dado como parámetro del método	Se conoce el factorial del número entero <i>num</i>	1. Declarar e iniciar una variable local (<i>facto</i>) de tipo entero en 1 2. Declarar un contador de tipo entero 3. En un ciclo para generar el valor definitivo de la variable <i>facto</i> $facto = num * (num-1) * (num-2) * \dots * 2 * 1$ 4. Retornar el valor de la variable <i>facto</i>

esPrimo (num)	R3	No se sabe si el número entero <i>num</i> corresponde a un número primo	Se sabe con certeza si el número <i>num</i> es primo o no lo es	<ol style="list-style-type: none"> 1. Declarar e inicializar una variable (<i>sw</i>) de tipo lógico en cierto (supuesto: el número <i>num</i> es primo) 2. Declarar un contador (<i>cont</i>) de tipo entero e iniciarlo en 2 3. En un ciclo mientras: Verificar si la operación $num \% cont == 0$; si es verdadero cambiar la variable <i>sw</i> a falso 4. Retornar la variable <i>sw</i>
tangente(num)	R4	Se desconoce la tangente del ángulo especificado por el número <i>num</i>	Se sabe la tangente del número <i>num</i>	<ol style="list-style-type: none"> 1. Declarar una variable local (<i>tang</i>) de tipo real y asignarle el valor $\text{seno}(\text{num}) / \text{coseno}(\text{num})$. 2. Retornar el valor de la variable <i>tang</i>
principal()	R1 R2 R3 R4	No existe algún dato pendiente por procesar	Se conoce el número entero a procesar, su factorial y tangente y se ha identificado si corresponde a un número primo	<ol style="list-style-type: none"> 1. Declarar una variable entera (<i>n</i>) 2. Solicitar al usuario un valor para <i>n</i> 3. Invocar los métodos <i>factorial(n)</i>, <i>esPrimo(n)</i> y <i>tangente(n)</i>

Observaciones:

En el contrato de la clase Número se ha incluido una nueva columna, correspondiente al *modelo verbal*. Este modelo es una descripción en lenguaje natural, lo más simple posible, de la secuencia lógica de instrucciones que debe ejecutar el método, algo similar a lo que se ha denominado *algoritmo cualitativo* dentro del argot conceptual de la programación estructurada. Debe anotarse que las dos denominaciones para esta columna de naturaleza opcional, son válidas, y que cuando aparece, es porque el método exige cierto nivel de abstracción o desarrollo lógico. Para un aprendiz de programación, la columna “modelo verbal” adquiere gran relevancia porque en ella se plantea la solución de una parte del problema con la fluidez que ofrece la lengua vernácula.

d) Seudo código orientado a objetos (OO)

```

clase Numero
  publico estatico entero factorial(entero num)
    entero facto = 1, cont
    para (cont = num, cont > 1, cont = cont - 1)
      facto = facto * cont
    fin_para
    retornar facto
  fin_metodo
  //-----
  publico estatico logico esPrimo(entero num)
    logico sw = cierto //Supuesto: num es primo
    si (num > 2)
      entero cont = 2
      mientras (cont <= num/2 && sw == cierto)
        si (num % cont == 0)
          sw = falso
        sino
          cont = cont + 1
        fin_si
      fin_mientras
    fin_si
    retornar sw
  fin_metodo
  //-----
  publico estatico real tangente(entero num)
    real tang
    tang = Mat.seno(num) / Mat.coseno(num)
    retornar tang
  fin_metodo
  //-----
  principal ()
1.  entero n
2.  // R1
3.  Flujo.imprimir("Ingrese un número entero mayor o igual a cero:")
4.  repetir
5.    n = Flujo.leerEntero()
6.  hasta (n < 0)
7.  // R2
8.  entero f = factorial(n)
9.  Flujo.imprimir(n + "!" = " + f)
10. // R3
11. logico p = esPrimo(n)
12. si (p == cierto)
13.   Flujo.imprimir("El número " + n + " es primo")
14. sino
15.   Flujo.imprimir("El número " + n + " no es primo")
16. fin_si
17. // R4
18. real t = tangente(n)
19. Flujo.imprimir("tangente( " + n + ") = " + t)
  fin_metodo
fin_clase

```

Observaciones:

- En el método *principal()* estarán sin excepción involucrados todos los requerimientos del problema, porque es allí precisamente donde se lleva a cabo la solución del mismo, sea que se invoquen o no otros métodos, o se instancien o no objetos.
- Los métodos *factorial()*, *esPrimo()* y *tangente()* son estáticos y esto se indica explícitamente en el seudo código. Recordar que el método principal también es estático por omisión.
- En aras de claridad, cada método es separado por una línea de comentario. Los comentarios se preceden por dos barras inclinadas (slash), estilo C# o Java.
- Las *palabras reservadas* o *palabras claves* que llevan tilde, como **público**, **estático**, **lógico** y **fin_método**, no han sido tildadas en el seudo código expuesto. Este detalle es indiferente, así que el analista puede decidir entre tildar o no las palabras reservadas.
- El cuerpo del método *tangente()* que incluye una declaración y dos instrucciones ejecutables, se puede reemplazar por el siguiente *atajo* (instrucción que incluye dos o más declaraciones o instrucciones):
 - `retornar (Mat.seno(num) / Mat.coseno(num))`
- En computación, un *atajo* es una instrucción que incluye a varias otras, utilizada por programadores avanzados para ahorrar escritura de código. Ejemplos de otros atajos podrían ser:

```
entero multiplicacion (entero a, entero b)
    entero c
    c = a * b
    retornar c
fin_ metodo
```

Atajo:

```
entero multiplicacion (entero a, entero b)
    retornar (a*b)
fin_ metodo
```

```

real valor1
entero valor2
valor1 = Flujo.leerReal( )
valor2 = Mat.redondear(valor1)

```

Atajo:

```

entero valor2
valor2 = Mat.redondear(Flujo.leerReal( ))

```

Este atajo ha permitido inutilizar la variable `valor1` con el consecuente ahorro de memoria.

- Las instrucciones del método *principal()* se han numerado para facilitar las siguientes explicaciones:
 - La sesión de declaración de variables del método principal está esparcida a lo largo del algoritmo. Por ejemplo, en las líneas 1, 8, 11 y 18 se han declarado variables *n*, *f*, *p* y *t*, respectivamente. Estas cuatro declaraciones se pueden escribir antes de la primera instrucción ejecutable del método, es decir, en la línea 1.
 - El método *principal()* suple todos los requerimientos del usuario, tal como se indicó en el contrato de la clase *Número*. Las líneas de comentario 2, 7, 10 y 17 así lo enfatizan.
 - La lectura de la variable *n* ha sido validada a través de un *ciclo repetir / hasta*, para garantizar un valor mayor o igual a cero, como se observa en las líneas 4 a 6. Los ciclos o estructuras de control iterativas serán profundizadas en la sesión 3.4. (Estructuras de control).
- Dado que los tres métodos invocados en el *principal()* retornan un valor por su calidad de funciones, se pueden llamar directamente desde el método *imprimir()* de la clase *Flujo*, ahorrándonos así el uso de las variables locales *f*, *p* y *t*. Una nueva versión del método *principal()*, que incluye esta observación, es la siguiente:


```

principal ( )
  entero n
  // R1
  Flujo.imprimir("Ingrese un número entero mayor o igual a cero:")
  repetir
    n = Flujo.leerEntero( )
  hasta (n < 0)
  // R2
  Flujo.imprimir(n + " ! = " + factorial(n))
  // R3
  si (esPrimo(n))
    Flujo.imprimir("El número " + n + " es primo")
  sino
    Flujo.imprimir("El número " + n + " no es primo")
  fin_si
  // R4
  Flujo.imprimir("tangente( " + n + " ) = " + tangente(n))
fin_metodo

```

Nota:

La decisión "**si** (esPrimo(n))" compara implícitamente el valor retornado con el valor de verdad cierto, por tanto equivale a "**si** (esPrimo(n) == cierto)"

PASO DE PARÁMETROS POR VALOR VS. PASO DE PARÁMETROS POR REFERENCIA

Aunque un método puede carecer de *parámetros*, estos constituyen un medio de comunicación con el medio externo. Los parámetros pueden ser de entrada (↓), de salida (↑) o de entrada y salida (↕).

Un *parámetro es de entrada* cuando su valor proviene del método que realiza la llamada; en este caso, el contenido del parámetro no lo puede modificar el método que lo recibe. *De salida* cuando su contenido es producido por el método y será devuelto al método que realiza la llamada. Y por último, es de *entrada y salida* cuando ingresa con un valor determinado adquirido desde el método que lo llama, el cual es posteriormente modificado dentro del método que lo recibe. En cuanto a los tipos de pasos de los

parámetros, puede decirse que si es de entrada se da *paso de parámetros por valor*; si es salida o de entrada y salida se *presenta paso de parámetros por referencia*, denominado también *paso de parámetros por dirección*.

Para concluir, en la lógica de lenguajes de programación como C++, C# o Java todo paso de parámetros entre métodos se da por valor; en lenguajes como C o Visual Basic.NET el paso de parámetros puede suceder por valor o por referencia.

3.3. SOBRECARGA DE MÉTODOS

Cuando se sobrecarga un método, éste se redefine con el objetivo de añadirle una nueva funcionalidad. Esto implica una de las siguientes acciones: agregar nuevos parámetros al método, conservar el mismo número de parámetros cambiando el tipo de los mismos, o intercambiar el orden de los parámetros originales; en todos los casos, el nombre del método permanece inalterable.

Cualquier método se puede sobrecargar, menos el método *principal()*; así, es común la sobrecarga de constructores y métodos analizadores, y el que los métodos sobrecargados también difieran en el tipo de valor retornado. Asimismo puede suceder que una clase sobrecargue un método una o más veces, lo cual llevaría al programador a la siguiente incógnita: ¿cuál método ejecutar cuando se llame una clase? Esto se hace comparando el número y tipos de los *argumentos* especificados en la llamada, con los parámetros especificados en las distintas definiciones del método.

Los problemas 6 y 7 desarrollan métodos constructores y analizadores sobrecargados y presentan paso de parámetros por valor y por referencia. Así, en el problema 6 se sobrecarga el método constructor Empleado(), se pasan parámetros por valor en los métodos asignarNombre() y asignarSalario() y por referencia en el método nombreMayorSalario(); en el Problema 7, se sobrecarga el método analizador informe().

PROBLEMA 6. EMPLEADO CON MAYOR SALARIO

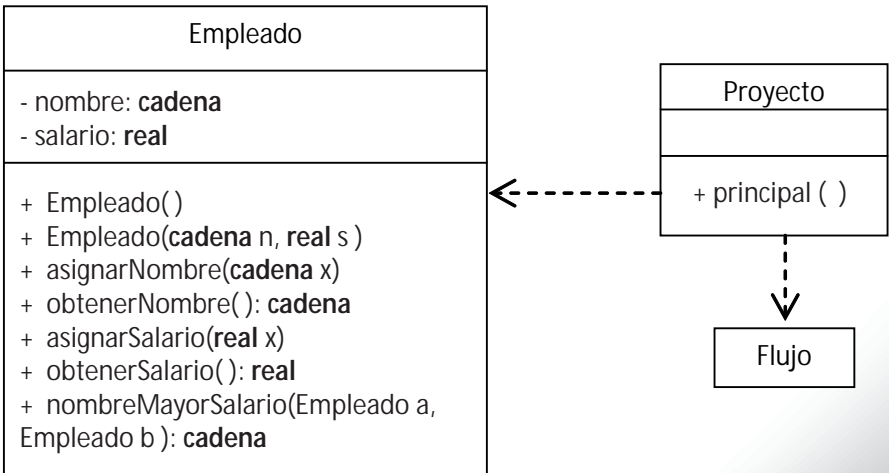
Visualizar el nombre del empleado que devenga el mayor salario y verificar la solución planteada con dos pares de empleados: el primer par de datos ingresados por el usuario y el segundo par dado por Adán Picasso, 4500000 y Eva Modigliani, 2000000.

Solución:

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	De dos empleados, visualizar el nombre de aquel que devenga el mayor salario	Nombre y salario de dos empleados especificados por el usuario	El nombre del empleado que devenga el mayor salario
R2	Para dos empleados específicos (Adán Picasso y Eva Modigliani), visualizar el nombre del que devenga el mayor salario	Los datos (nombre y salario) de un par de empleados.	El nombre del empleado que devenga el mayor salario.

b) Diagrama de clases



c) Responsabilidades de las clases

Contrato de la clase Empleado

Método	R. a.*	Precondición	Postcondición	Modelo verbal
Empleado()	R1	El empleado no existe en memoria	El empleado existe con valores para nombre y salario asignados por defecto	Se inicializa de manera automática el nombre con la cadena vacía y el salario en cero
Empleado (cadena n, real s)	R2	El empleado no existe en la memoria	El empleado existe con los valores establecidos por los parámetros de entrada	<ol style="list-style-type: none"> 1. Asignar al nombre del empleado el valor del parámetro n 2. Asignar al salario del empleado el valor del parámetro
nombre MayorSalario(Empleado a, Empleado b): cadena	R1 R2	Se desconoce el nombre del empleado que devenga el mayor salario	Se conoce el nombre del empleado que devenga el mayor salario	<ol style="list-style-type: none"> 1 Comparar el salario de los dos empleados 2 Devolver el nombre del empleado con el mayor salario

asignarNombre (cadena x)	R1	El empleado tiene un nombre igual a la cadena vacía	El empleado tiene un nombre dado por el parámetro x	Asignar al atributo <i>nombre</i> el valor de x
obtenerNombre(): cadena	R1 R2	El usuario desconoce el nombre del empleado	El usuario conoce el nombre del empleado	Devolver el valor del atributo <i>nombre</i>
asignarSalario (real x)	R1	El empleado tiene un salario igual a cero	El empleado tiene un salario dado por el parámetro x	Asignar al atributo <i>salario</i> el valor de x
obtenerSalario(): real	R1 R2	El usuario desconoce el salario del empleado	El usuario conoce el salario del empleado	Devolver el valor del atributo <i>salario</i>


* R. a.: Requerimiento asociado

d) Seudo código

```

clase Empleado
  privado:
    cadena nombre
    real salario
  publico:
    Empleado( )
    fin_metodo
    //-----
    Empleado(cadena n, real s)
        nombre = n
        salario = s
    fin_metodo
    //-----
    asignarNombre(cadena x)
        nombre = x
    fin_metodo
    //-----
    cadena obtenerNombre( )
        retornar nombre
    fin_metodo
    //-----
    asignarSalario(real x)
        salario = x
    fin_metodo
    //-----
    real obtenerSalario( )
        retornar salario
    fin_metodo
    //-----
    estatico cadena nombreMayorSalario( Empleado a, Empleado b)
        cadena nom
        si (a.obtenerSalario( ) > b.obtenerSalario())
            nom = a.obtenerNombre()
        sino
            si (b.obtenerSalario( ) > a.obtenerSalario())
                nom = b.obtenerNombre()
            sino
                nom = " "
            fin_si
        fin-si
        retornar nom
    fin_metodo
fin_clase
//*****

```





```

clase Proyecto
  publico principal ( )
1    cadena nom
2    Empleado e1 = nuevo Empleado()
3    Empleado e2 = nuevo Empleado()
4
5    Flujo.imprimir("Ingrese nombre y salario de dos empleados:")
6    e1.asignarNombre(Flujo.leerCadena( ))
7    e1.asignarSalario(Flujo.leerReal( ))
8    e2.asignarNombre(Flujo.leerCadena( ))
9    e2.asignarSalario(Flujo.leerReal( ))
10   nom = Empleado. nombreMayorSalario(e1, e2)
11   si (Cadena.comparar(nom, " "))
12       Flujo.imprimir("Ambos empleados devengan igual salario")
13   sino
14       Flujo.imprimir("Quien más devenga entre " +
15           e1.obtenerNombre() + " y " + e2.obtenerNombre() +
16           " es " + nom)
17   fin_si
18   Empleado e3 = nuevo Empleado("Adán Picasso", 4500000)
19   Empleado e4 = nuevo Empleado("Eva Modigliani", 2000000)
20   nom = Empleado. nombreMayorSalario(e3, e4)
21   si (Cadena.comparar(nom, " "))
22       Flujo.imprimir("Ambos empleados devengan igual salario")
23   sino
24       Flujo.imprimir("Quien más devenga entre " +
25           e3.obtenerNombre() + " y " + e4.obtenerNombre() +
26           " es " + nom)
27   fin_si
28   fin_metodo
29   fin_clase

```

Observaciones:

- El método *nombreMayorSalario()* es estático y retorna el nombre del empleado que devenga el mayor salario. Dado el caso que ambos empleados obtengan un igual salario, el método retorna una cadena vacía.

- El cuerpo del método *principal()* se ha enumerado para facilitar ciertas explicaciones. Las líneas 4, 16 y 19 están vacías: cuestión de estilo en la presentación del seudo lenguaje. Las instrucciones 14 y 24 conllevan tres líneas de seudo código.
- El método constructor fue sobrecargado una vez, agregando dos parámetros: uno de tipo *cadena* para el nombre y otro de tipo *real* para el salario. Esto permite crear objetos de dos formas distintas, tal como se observa en las líneas 2, 3, 17 y 18.

PROBLEMA 7: INFORME SOBRECARGADO

Determinado usuario, del cual se conoce su nombre, desea obtener los valores correspondientes al seno y coseno de dos ángulos establecidos por él mismo, uno de tipo entero y otro de tipo real. Se pide visualizar dos informes: el primero debe incluir un saludo previo y el nombre del usuario. El saludo depende del tiempo del día (mañana, tarde o noche) y puede ser uno de tres: “Buen día”, “Buena tarde” o “Buena noche”. El segundo informe debe mostrar los valores de las funciones solicitadas y el mayor entre los dos ángulos ingresados.

Véanse un par de salidas posibles:

- *Buen día Sr(a) Ava Gardner.*
Para 30 y 52.8 grados, tenemos:

$$\text{Seno}(30) = 0.50 \qquad \text{Coseno}(30) = 0.87$$

$$\text{Seno}(52.8) = 0.80 \qquad \text{Coseno}(52.8) = 0.60$$
El ángulo mayor es 52.8.
- “Buena tarde Sr(a) Juan D’Arienzo.
Para 45 y 7.5 grados, tenemos:

$$\text{Seno}(45) = 0.71 \qquad \text{Coseno}(45) = 0.71$$

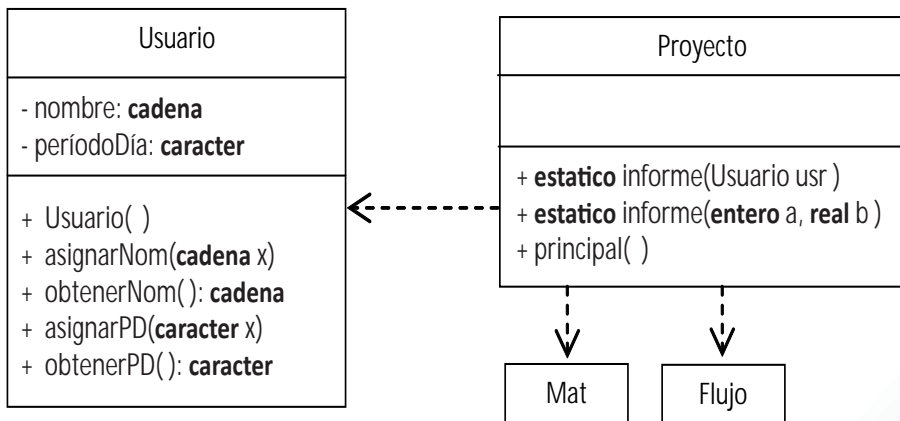
$$\text{Seno}(7.5) = 0.13 \qquad \text{Coseno}(7.5) = 0.99$$
El ángulo mayor es 45.

Solución:

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Conocer el nombre del usuario y el tiempo del día	Los valores digitados por el usuario	Nombre de usuario y período del día establecidos
R2	Visualizar el primer informe, que incluye un saludo y el nombre del usuario	Nombre del usuario y período del día ('m': Mañana, 't': Tarde, 'n': Noche)	Primer informe emitido
R3	Visualizar el segundo informe, que incluye seno y coseno de los dos ángulos y el mayor de los ángulos	Los valores de dos ángulos	Segundo informe emitido

b) Diagrama de clases



Observaciones:

- El método *informe()* está sobrecargado una vez: la definición inicial del método acepta un objeto tipo *Usuario* como parámetro, con el objetivo de elaborar el informe inicial. El método sobrecargado acepta como parámetros los valores entero y real de los dos ángulos, en búsqueda de elaborar el informe final.
- Para el cálculo de las funciones trigonométricas seno y coseno se utilizarán los métodos estáticos *Mat.sen()* y *Mat.cos()*, ambos sobrecargados para recibir argumentos enteros y reales, como se observa en la sesión 2.5 (Configuración de las clases de uso común).

c) Responsabilidades de las clases

Contrato de la clase *Usuario*

Método	R. a.*	Precondición	Postcondición
<i>Usuario()</i>	R1	Ninguna	Existe un objeto tipo <i>Usuario</i> en memoria
<i>asignarNom(cadena x)</i>	R1	Un objeto tipo <i>Usuario</i>	El atributo <i>nombre</i> del objeto tipo <i>Usuario</i> ha sido asignado
<i>obtenerNom(): cadena</i>	R1	Se desconoce el nombre del usuario	El nombre del usuario se ha obtenido
<i>asignarPD(caracter x)</i>	R1	Un objeto tipo <i>Usuario</i>	El atributo <i>periodoDia</i> del objeto tipo <i>Usuario</i> ha sido asignado
<i>obtenerPD(): caracter</i>	R1	Se desconoce el período del día en el cual el usuario requiere los informes	El período del día se ha obtenido

* Requerimiento asociado

Contrato de la clase Proyecto

Método	R. a.	Precondición	Postcondición
estatico informe(Usuario u)	R2	Conocer los datos de un usuario	Se ha visualizado el primer informe
estatico informe(entero a, real b)	R3	Conocer los dos ángulos	Se ha visualizado el segundo informe
principal()	R1 R2 R3	Conocer los datos de un usuario y los valores de los ángulos	Se han visualizado los dos informes

Observación:

La columna “*Modelo verbal*” se ha omitido en los contratos de las dos clases, porque los métodos son realmente simples. Esta columna se expone cuando los métodos presentan algún grado de complejidad o es muy extenso el proceso lógico de los mismos. Queda al libre elección del analista incluir o no esta columna dentro del contrato de las clases.

d) Seudo código

Clase Usuario

privado:

cadena nombre

carácter períodoDía

publico:

Usuario()

fin_metodo

//-----

asignarNom(**cadena** x)

 nombre = x

fin_metodo

//-----



```

↓
obtenerNom(): cadena
    retornar nombre
fin_metodo
//-----
asignarPD(caracter x)
    periodoDía = x
fin_metodo
//-----
obtenerPD(): carácter
    retornar periodoDía
fin_metodo
//-----
fin_clase
//*****
Clase Proyecto
publico:
    estatico informe(Usuario usr )
        según (usr.obtenerPD())
            caso 'm':
                Flujo.imprimir("Buen día")
                saltar
            caso 't':
                Flujo.imprimir("Buena tarde")
                saltar
            caso 'n':
                Flujo.imprimir("Buena noche")
        fin_según
        Flujo.imprimir(" Sr(a) " + usr.obtenerNom ())
    fin_metodo
    //-----
    estatico informe(entero a, real b )
        Flujo.imprimir("Para " + a + " y " + b + " grados, tenemos:")
        Flujo.imprimir("Seno(" + a + ") =" + Mat.sen(a))
        Flujo.imprimir("Coseno(" + a + ") =" + Mat.cos(a))
        Flujo.imprimir("Seno(" + b + ") =" + Mat.sen(b))
        Flujo.imprimir("Coseno(" + b + ") =" + Mat.cos(b))
        Flujo.imprimir("El ángulo mayor es ")
        si (a > b)
            Flujo.imprimir(a)
        sino
            Flujo.imprimir(b)
        fin_si
    fin_metodo
    //-----
↓

```



```

principal( )
    Usuario u = nuevo Usuario( )
    entero a1
    real a2
Flujo.imprimir("Ingrese nombre del usuario: ")
    u.asignarNom(Flujo.leerCadena( ))
Flujo.imprimir("Ingrese período del día ('m': mañana, 't': tarde, 'n': noche): ")
    u.asignarPD(Flujo.leerCaracter( ))
    informe(u)
Flujo.imprimir("Ingrese el valor de los dos ángulos: ")
    a1 = Flujo.leerEntero( )
    a2 = Flujo.leerReal( )
    informe(a1, a2)
fin_metodo
//-----
fin_clase

```

3.4. ESTRUCTURAS DE CONTROL

Las *estructuras de control*, como su nombre lo indica, permiten controlar el flujo de ejecución de un método; tienen una entrada y una salida y se clasifican en tres tipos:

- **Secuencia:** ejecución sucesiva de una o más instrucciones.
- **Selección:** ejecuta uno de dos posibles conjuntos de instrucciones, dependiendo de una condición (expresión lógica) o del valor de una variable.
- **Iteración:** repetición de una o varias instrucciones a la vez que cumple una condición.

LA ESTRUCTURA SECUENCIA

En la *estructura secuencia*, las instrucciones se aparecen una a continuación de la otra, en secuencia lineal, sin cambios de ruta. Una sintaxis general de la estructura secuencia está dada por:

```
instrucción_1  
instrucción_2  
:  
:  
instrucción_N
```

Ejemplo:

```
x = Flujo.leerEntero()  
r = Mat.seno(x) - 7.5  
Flujo.imprimir(r)
```

LA ESTRUCTURA SELECCIÓN

La *estructura selección* admite dos variantes: la *decisión* y el *selector múltiple*. La primera, identificada con la *sentencia si*, evalúa una condición y, dependiendo de su valor de verdad, ejecuta un bloque de instrucciones u otro. La segunda, identificada con la *sentencia según*, evalúa una variable denominada “selector” y ejecuta uno de tres o más casos.

Sintaxis de la instrucción de decisión:

```
si (e)  
    instrucciones_1  
[sino  
    instrucciones_2]  
fin_si
```

Donde,

e: expresión lógica.

instrucciones_1: bloque de instrucciones a ejecutar si la expresión *e* es verdadera.

instrucciones_2: bloque de instrucciones a ejecutar si *expresión* es falsa.

La cláusula **sino** es opcional, por eso va encerrada entre corchetes. Una *cláusula* es una palabra reservada que forma parte de una instrucción; en este caso, la instrucción de decisión incluye la sentencia **si** y dos cláusulas: **sino** y **fin_si**.

Ejemplos de estructuras de decisión:

- Una instrucción por bloque

```
si (x % 2 == 0)
    Flujo.imprimir(x + "es par")
sino
    Flujo.imprimir(x + "es impar")
fin_si
```

- Sin cláusula **sino**

```
si (num > 0)
    valor = num * 2
fin_si
```

- Varias instrucciones por bloque

```
si (x % 2 == 0)
    Flujo.imprimir(x + "es par")
    sw = cierto
sino
    Flujo.imprimir(x + "es impar")
    sw = falso
    cont = cont + 1
fin_si
```

– Condición compuesta

```
si (a > b && a > c)
    Flujo.imprimir("El mayor es " + a)
fin_si
```

– *Decisión anidada*: bloque decisivo dentro de otro

```
si (nombre.comparar("John Neper"))
    indicativo = cierto
    c1 = c1 + 1
sino
    si (nombre.comparar("Blaise Pascal"))
        indicativo = falso
        c2 = c2 + 1
    fin_si
fin_si
```

No existe un límite para el número de anidamientos.

El *selector múltiple* tiene la siguiente sintaxis:

```
según (vs)
    caso c1:
        instrucciones_1
    caso c2:
        instrucciones_2
    caso cN:
        instrucciones_N
    [sino
        instrucciones_x]
fin_según
```

Donde,

vs: variable selectora. Debe ser de tipo entero, carácter, tipo enumerado o subrango.

ci : constante i , de igual tipo al de la variable selectora vs . $i = 1, 2, \dots, N$
 $instrucciones_i$: lista de instrucciones a ejecutar en caso de existir coincidencia entre vs y ci , $i = 1, 2, \dots, N$. Si no se presenta coincidencia, se ejecutan las $instrucciones_x$, en caso de existir.

Ejemplo:

```
según (opción)
    caso 1:
        c1 = c1 + 1
        adicionar( )
        romper
    caso c2:
        c2 = c2 + 1
        eliminar( )
        romper
    caso c3:
        c3 = c3 + 1
        ordenar( )
        romper
    sino
        Flujo.imprimir("Opción inválida")
fin_según
```

PROBLEMA 8. PRODUCTO MÁS CARO

Dados los nombres y precios de dos productos, imprimir el nombre del producto más caro. Si ambos tienen el mismo precio mostrar el mensaje "Tienen precios iguales".

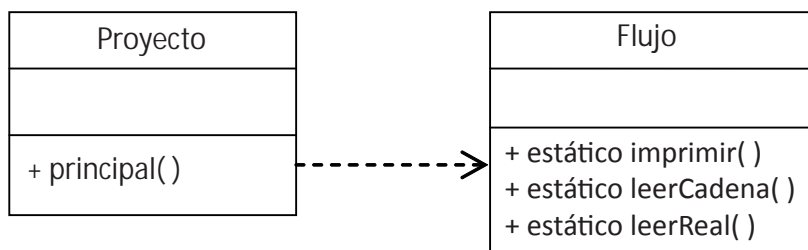
Solución:

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Conocer los nombres y precios de dos productos	Datos aportados por el usuario	Los nombres y precios de los dos productos se encuentran almacenados en variables de memoria
R2	Identificar y visualizar el nombre del producto más caro	El nombre y precio de dos productos	El nombre del producto más caro o el mensaje "Tienen precios iguales"

b) Diagrama de clases

El siguiente diagrama de clases dista de un enfoque orientado a objetos acertado. Se centra en el trabajo con la sentencia de control de decisión. En las observaciones presentadas después del pseudocódigo, se expone una solución alternativa más sólida en cuanto a la abstracción de objetos se refiere, porque se instancian dos objetos de clase Producto, y se les envían los mensajes respectivos para la solución del problema.



c) Contrato de la clase *Proyecto*

Nombre del método	Requerimiento asociado	Precondición	Postcondición	Modelo verbal
principal()	R1 R2	Se desconocen los datos de los productos	Se sabe el nombre del producto más caro o se ha identificado que ambos productos tienen igual precio	<ol style="list-style-type: none"> 1. Declarar dos variables de tipo cadena y dos variables de tipo real 2. Ingresar los nombres y precios de los dos productos 3. Comparar los precios de los productos para identificar el nombre del producto más caro o visualizar el mensaje que indic precios iguales

d) Seudo código orientado a objetos (OO)

```

clase Proyecto
  público principal ( )
    cadena nom1, nom2
    real precio1, precio2

    Flujo.imprimir ("Ingrese nombre y precio de dos productos:")
    nom1 = Flujo.leerCadena()
    precio1 = Flujo.leerReal()
    nom2 = Flujo.leerCadena()
    precio2 = Flujo.leerReal()

    si (precio1 > precio2)
      Flujo.imprimir ("El producto más caro es " + nom1)
    sino
      si (precio2 > precio1)
        Flujo.imprimir ("El producto más caro es " + nom2)
      sino
        Flujo.imprimir ("Tienen precios iguales")
      fin_si
    fin_si
  fin_método
fin_clase

```

Observaciones:

- Las variables *nom1* y *nom2* son de tipo cadena y las variables *precio1* y *precio2* son de tipo real.
- La entrada y salida de datos estándar se realiza a través de la clase **Flujo**.
- Para facilitar la lectura de datos, se debe mostrar al usuario el mensaje respectivo. Por ejemplo, antes de leer los nombres y precios de los productos se imprimió el mensaje “Ingrese nombre y precio de dos productos:”, teniendo así, para abreviar, una instrucción de salida por cuatro de entrada. No obstante, se puede preceder cada instrucción de entrada por una de salida, de la siguiente manera:

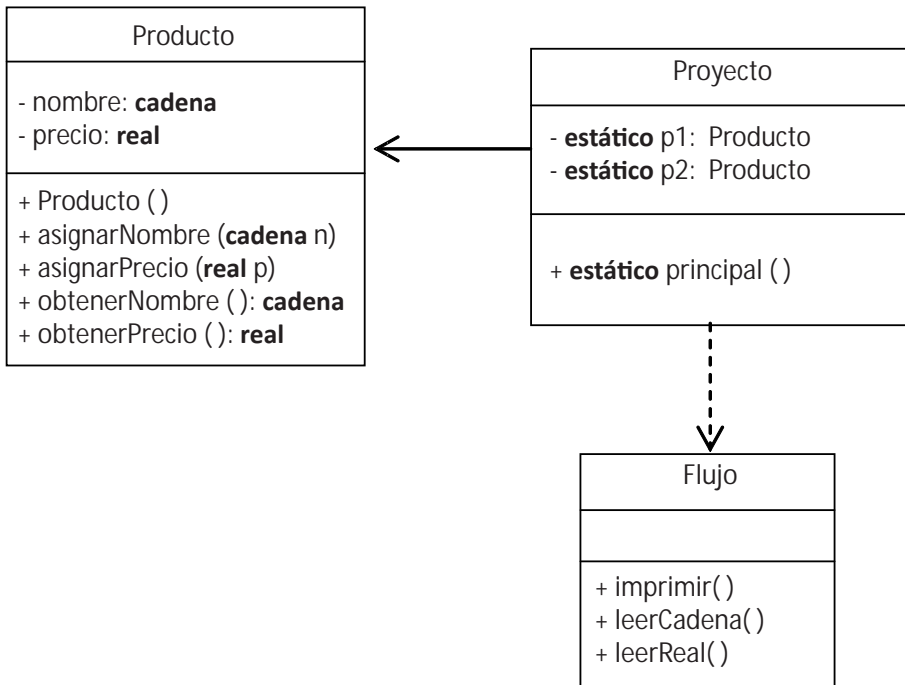
```
Flujo.imprimir (“Ingrese nombre del primer producto:”)
nom1 = Flujo.leerCadena( )
Flujo.imprimir (“Ingrese precio del primer producto:”)
precio1 = Flujo.leerReal( )
Flujo.imprimir (“Ingrese nombre del segundo producto:”)
nom2 = Flujo.leerCadena( )
Flujo.imprimir (“Ingrese precio del segundo producto:”)
precio2 = Flujo.leerReal( )
```

En general, en el seudo código se prefiere la forma abreviada. De esta manera, la comunicación con el usuario se realiza a través de interfaces de texto, donde los datos de entrada se solicitan en un solo mensaje. La comunicación con el usuario a través de interfaces gráficas se pospone para la etapa de codificación en un lenguaje de programación, lo cual está excede los intereses del presente texto.

- La *estructura de control selectiva* (o sentencia de *decisión si*) permite ejecutar un grupo de instrucciones u otro, dependiendo del valor de verdad de una expresión lógica o condición. En el pseudocódigo expuesto se presentan dos estructuras de control selectivas, una de ellas anidada, es decir, una sentencia de decisión dentro de otra.
- El método *imprimir* de la clase **Flujo** se encuentra sobrecargado, es decir, admite parámetros enteros, reales, caracteres y cadenas. Cuando se pretende visualizar

información variada, ésta va separada por el *operador de concatenación* cruz (+) que también se encuentra sobrecargado (dependiendo del contexto, significa sumar o concatenar).

- Como se indicó antes, se puede presentar un diseño de clases más acertado en términos de la orientación por objetos.



Los atributos `p1` y `p2` son estáticos porque se crean e instancian dentro del método `principal ()`, que también es de naturaleza estática.

El *diagrama conceptual* correspondiente se presenta en la figura 3.1, donde los atributos estáticos `p1` y `p2` se han reemplazado por la *asociación* (flecha de línea continua) que parte de la *clase Proyecto* y llega a la *clase Producto*. Observar que entre las *clases Proyecto* y *Flujo* se presenta una *relación de uso* porque dentro del método `principal ()` se utilizan los métodos estáticos `imprimir ()`, `leerCadena ()` y `leerReal ()` de dicha clase. Ver Figura 3.1.

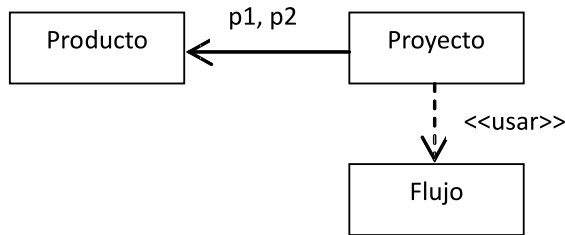


Figura 3.1. Asociación entre clases Proyecto y Producto

El pseudocódigo correspondiente a la clase Proyecto es el siguiente:

```

clase Proyecto
  privado estático Producto p1, p2
  público estático principal ()
    // Crear dos objetos tipo Producto
    p1 = nuevo Producto ()
    p2 = nuevo Producto ()

    // Asignar estado a los objetos
    Flujo.imprimir ("Ingrese el nombre y precio del primer producto:")
    p1.asignarNombre (Flujo.leerCadena())
    p1.asignarprecio (Flujo.leerReal())
    Flujo.imprimir ("Ingrese el nombre y precio del segundo producto:")
    p2.asignarNombre (Flujo.leerCadena())
    p2.asignarprecio (Flujo.leerReal())

    // Analizar el estado de los objetos
    si (p1.obtenerPrecio () > p2.obtenerPrecio ())
      Flujo.imprimir ("El producto más caro es " + p1.obtenerNombre ())
    sino
      si (p2.obtenerPrecio () > p1.obtenerPrecio ())
        Flujo.imprimir ("El producto más caro es " + p2.obtenerNombre ())
      sino
        Flujo.imprimir ("Tienen precios iguales")
      fin_si
    fin_si
  fin_método
fin_clase
  
```

Ahora se tratará un problema que involucra las sentencias de control ciclo y selector múltiple, además de aportar algunas observaciones acerca de la definición de la tabla de requerimientos para un problema específico.

PROBLEMA 9. ESTADÍSTICAS POR PROCEDENCIA

De un número determinado de personas se conoce la estatura, procedencia y edad. La estatura y la procedencia se manejan de acuerdo a las siguientes convenciones:

Estatura = 1 (alta), 2 (baja) o 3 (Mediana)

Procedencia = 'L' (Americana), 'E' (Europea), 'A' (Asiática) u 'O' (Otra).

Determinar:

- *El número de americanos altos, europeos bajos y asiáticos medianos.*
- *La edad promedio de los individuos de otras procedencias.*
- *La cantidad de americanos bajos mayores de edad.*

Problema adaptado de [Oviedo2004]

Solución:

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Conocer la cantidad de personas a procesar	Un número entero ingresado por el usuario	La cantidad de personas a procesar almacenada en una variable (n)
R2	Conocer las estaturas, procedencias y edades de todas las personas	<ul style="list-style-type: none"> - Un número entero (n), igual a la cantidad de personas a procesar. - Por cada persona: su estatura, procedencia y edad 	Se conocen los datos de todas las personas

R3	Encontrar el número de americanos altos	Los datos de todas las personas	El número de americanos altos
R4	Encontrar el número de europeos bajos	Los datos de todas las personas	El número de europeos bajos
R5	Encontrar el número de asiáticos medianos	Los datos de todas las personas	El número de asiáticos medianos
R6	Hallar la edad promedio de los individuos de otras procedencias	Los datos de todas las personas	La edad promedio de los individuos de otras procedencias
R7	Hallar la cantidad de americanos bajos mayores de edad	Los datos de todas las personas	La cantidad de americanos bajos mayores de edad

Nota importante:

La *identificación de requerimientos* es de trascendental importancia en el desarrollo de software, ya que en esta etapa se analiza el problema y se identifican las necesidades (requerimientos) de los usuarios. Si los requerimientos están mal especificados, vendrán futuros inconvenientes cuando se analicen los resultados del programa.

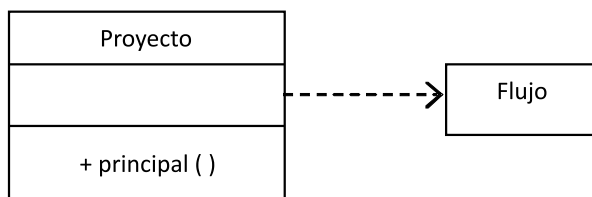
Aunque la definición de requerimientos de un problema debe ser objetiva, su identificación es subjetiva. Por esto, los diferentes analistas de un problema deben llegar a estructurar, en términos generales, los mismos requerimientos, aunque su descripción en lenguaje natural tenga ligeras diferencias. Incluso, el número de filas de la tabla de requerimientos puede diferir de un analista a otro, porque cualquiera de ambos puede describir un requerimiento como la unión de dos o más de ellos propuestos por el otro analista.

Por lo anterior, la siguiente tabla con cinco requerimientos también es válida, aunque en este ejemplo seguiremos trabajando con los siete requerimientos ya definidos.

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Conocer la cantidad de personas a procesar	Un número ingresado por el usuario	Se conoce la cantidad de personas a procesar
R2	Conocer las estaturas, procedencias y edades de todas las personas.	<ul style="list-style-type: none"> - Un número entero (n), igual a la cantidad de personas a procesar. - Por cada persona: su estatura, procedencia y edad 	Se conocen los datos de todas las personas
R3	Encontrar el número de americanos altos, europeos bajos y asiáticos medianos	Los datos de todas las personas	La cantidad de americanos altos, europeos bajos y asiáticos medianos
R4	Hallar la edad promedio de los individuos de otras procedencias	Los datos de todas las personas	La edad promedio de los individuos de otras procedencias
R5	Hallar la cantidad de americanos bajos mayores de edad	Los datos de todas las personas	La cantidad de americanos bajos mayores de edad

b) Diagrama de clases

El diagrama de clases se enfoca al uso exclusivo del selector múltiple, por lo que todo el seudocódigo de incluye en el método principal. Como ejercicio propuesto se invita a rediseñar la solución con una clase adicional, por ejemplo denominada *Persona*.



b) Contrato de la clase Proyecto

Nombre del método	Requerimiento asociado	Precondición	Postcondición	Modelo verbal
principal ()	R1 R2 R3 R4 R5 R6 R7	Se desconocen los datos de las personas	Se conocen todos los datos estadísticos solicitados	<ol style="list-style-type: none"> 1. Declarar e inicializar contadores y acumuladores 2. Ingresar la cantidad de personas a procesar 3. Por cada persona <ol style="list-style-type: none"> 3.1. Ingresar edad, estatura y procedencia. 3.2. Según los datos ingresados, actualizar contadores y acumuladores. 4. Mostrar los resultados

c) Seudo código orientado a objetos (OO)

Antes de proceder con el pseudocódigo es conveniente documentarlo, es decir, definir el significado de cada variable utilizada. La *documentación* se dará en la siguiente tabla, donde los identificadores de las variables se describen según su orden de aparición en la sección de declaración de variables del método *principal()*.

Documentación:

Identificador	Tipo	Significado
n	entero	Cantidad de personas a procesar.
cont	entero	Contador de personas. Es la variable controladora del ciclo para.
estat	entero	Estatura de una persona.
cla	entero	Cantidad (o contador) de latinos altos.
ceb	entero	Cantidad de europeos bajos.
cam	entero	Cantidad de asiáticos medianos
co	entero	Cantidad otros (o cantidad de individuos de otras procedencias).
proc	carácter	Procedencia de una persona.
edad	real	Edad de una persona.
epo	real	Edad promedio individuos de otras procedencias.
aeo	real	Acumulador de edad para individuos de otras procedencias.

Seudocódigo:

```

clase Proyecto
  público principal ()
    entero n, cont, estat, cla = 0, ceb = 0, cam = 0, co = 0
    caracter proc
    real edad, epo, aeo = 0

    Flujo.imprimir ("Ingrese la cantidad de personas a procesar:")
    n = Flujo.leerEntero()
    Para (cont = 1, cont <= n, cont = cont + 1)
      Flujo.imprimir ("Ingrese estatura (1 = Alta, 2 = Baja, 3 = Mediana): ")
      estat = Flujo.leerEntero()
      Flujo.imprimir ("Ingrese procedencia (L = Latino, E = Europeo,
        A = Asiático, O = Otra): ")
      proc = Flujo.leerCaracter()
      Flujo.imprimir ("Ingrese edad: ")
      edad = Flujo.leerEntero()
      según (proc)
        caso 'L':
          si (estat == 1)
            cla = cla + 1
          fin_si
        romper
        caso 'E':
          si (estat == 2)
            ceb = ceb + 1
          fin_si
        romper
        caso 'A':
          si (estat == 3)
            cam = cam + 1
          fin_si
        romper
        caso 'O':
          co = co + 1
          aeo = aeo + edad
      fin_según
    fin_para

    Flujo.imprimir ("Número de latinos altos: " + cla)
    Flujo.imprimir ("Número de europeos bajos: " + ceb)
    Flujo.imprimir ("Número de asiáticos medianos: " + cam)
    epo = aeo / co
    Flujo.imprimir ("Edad promedio individuos de otras procedencias: " + epo)
  fin_metodo
fin_clase

```

Observación:

Como antes se anotó, el diagrama de clases y en consecuencia el pseudocódigo expuesto, hacen hincapié en el uso del *selector múltiple* o sentencia **según**.

LA ESTRUCTURA ITERACIÓN

La *estructura iteración*, denominada también estructura *repetición*, *bucle* o *ciclo*, permite repetir un bloque de instrucciones, un determinado número de veces. Esta estructura de control admite tres variantes fundamentales: ciclo **mientras**, ciclo **para** y ciclo **repetir / hasta**.

- a) El *ciclo mientras* repite un bloque de instrucciones, denominado *cuerpo del ciclo*, mientras que una condición sea cierta; cuando la condición sea falsa, el ciclo **mientras** dejará de ejecutarse.

Sintaxis del ciclo **mientras**:

mientras (<i>e</i>) <i>instrucciones o cuerpo del ciclo</i> fin_mientras
--

Donde,

e: expresión lógica.

instrucciones: instrucciones a ejecutar si la expresión *e* es verdadera; al menos una de ellas debe hacer variar la expresión *e*, con el objetivo de evitar un ciclo infinito.

Ejemplos:

- Ciclo mientras simple:

```
cont = 1
mientras (cont < 10)
    Flujo.imprimir (cont)
    cont = cont + 1
fin_mientras
```

- Ciclo mientras con una condición compuesta y selector múltiple interno:

```
entero opcion = Flujo.leerEntero( )
lógico sw = cierto
entero cont = 1, c1 = 0, c2 = 0, c3 = 0

mientras (opcion != 4 && sw == cierto)
    según (opcion)
        caso 1:
            c1 = c1 + 1
            adicionar( )
            romper
        caso c2:
            c2 = c2 + 1
            eliminar( )
            Flujo.imprimir (cont)
            cont = cont + 1

        si (cont == 15)
            sw = falso
        fin_si
            romper
        caso c3:
            c3 = c3 + 1
            ordenar( )

    fin_según

    Flujo.imprimir("Ingrese opción: ")
    opcion = Flujo.leerEntero( )
fin_mientras
```

Puede observarse que la estructura secuencia del caso *c2* del selector múltiple incluye una instrucción de decisión.

– Ciclo **mientras** anidado:

```
entero a, b
a = 1
```

```
mientras (a <= 12)
```

```
  b = 1
```

```
  mientras (b <= 10)
```

```
    Flujo.imprimir(a + " * " + b + " = " + a * b)
```

```
    b = b + 1
```

```
  fin_mientras
```

```
  a = a + 1
```

```
fin_mientras
```

No existe un límite para la cantidad de ciclos anidados; el número de anidamientos depende de la situación a resolver.

b) El *ciclo para* es una variación del ciclo **mientras**, donde el bloque de instrucciones se repite una cantidad conocida de veces.

Sintaxis del ciclo **para**:

<pre>para (c = vi, e, i) instrucciones fin_para</pre>

Donde,

c: variable contadora del ciclo.

vi: valor inicial para *c*.

e: expresión lógica relacionada con *c*.

i: incremento para *c*.

Ejemplo:

```
entero c
para (c = 0, c < 100, c = c + 1)
    Flujo.imprimir(c)
fin_para
```

Observar la relación entre los ciclos **para** y **mientras**, ya que el anterior ciclo se puede reescribir de la siguiente manera:

```
entero c
c = 0
mientras (c < 100)
    Flujo.imprimir(c)
    c = c + 1
fin_mientras
```

- c) El *ciclo repetir / hasta* también es una variante del ciclo **mientras**, donde la condición del ciclo es evaluada al final del mismo. Si de entrada la condición es falsa, el cuerpo de un ciclo **mientras** puede no ejecutarse, mientras que el cuerpo del ciclo **repetir / hasta** siempre se ejecutará mínimo una vez.

Sintaxis del ciclo **repetir / hasta**:

<pre>repetir instrucciones hasta (e)</pre>
--

Donde,

e: expresión lógica.

instrucciones: instrucciones a ejecutar si la *e* es verdadera; al menos una de ellas hace variar la *e*.

Ejemplo:

```
repetir
    valor = Flujo.leerReal( )
hasta (valor < 0)
```

Es interesante observar la relación que existe entre los tres ciclos; por ejemplo, los siguientes tres segmentos de código son equivalentes:

- Segmento de código con ciclo **mientras**:


```
entero n = 5, suma = 0
mientras (n < 1200)
    suma = suma + n
    n = n + Mat.aleatorio(10)
fin_mientras
```
- Segmento de código con *ciclo para*:


```
entero n, suma = 0
para (n = 5, n < 1200, n = n + Mat.aleatorio(10) )
    suma = suma + n
fin_mientras
```
- Segmento de código con ciclo **repetir / hasta**:


```
entero n = 5, suma = 0
repetir
    suma = suma + n
    n = n + Mat.aleatorio(10)
hasta (n < 1200)
```

La equivalencia de estos tres ciclos se rompe si, por ejemplo, inicializamos la variable *n* en un valor mayor a 1200, cantidad que controla la ejecución de los tres bucles. Así, al hacer *n* = 1300, la variable *suma* en los dos primeros ciclos queda en 0 (cero), pero en el ciclo **repetir / hasta** queda valiendo 1300 (el cuerpo de este ciclo se ejecuta, en este caso, una vez).

En general, el *ciclo mientras* puede realizar la tarea de los otros dos, por esta razón es el más utilizado. Sin embargo, debemos tener en cuenta las siguientes recomendaciones para utilizar un determinado ciclo:

- Si se desconoce el número de iteraciones del ciclo, o sea cuando estas dependen de una entrada del usuario o de una función interna generada por el programa como un número aleatorio, se debe utilizar el bucle **mientras**.

Ejemplo:

```
cadena respuesta = Flujo.leerCadena( )
mientras (Cadena.comparar(respuesta, "si") == 0)
    // instrucciones
    respuesta = Flujo.leerCadena( )
fin_mientras
```

- Si la cantidad de iteraciones del ciclo es conocida, dada por una constante numérica o por un valor entero ingresado por el usuario, se recomienda el uso del ciclo **para**. Ejemplos:

- entero c
para (c = 1, c < 100, c = c + 2)
 // cuerpo del ciclo
fin_para
- n = Flujo.leerEntero()
para (cont = 1, cont < n, cont = cont + 1)
 // cuerpo del ciclo
fin_para

- Cuando se requiere de la validación de entradas de datos o que las instrucciones del cuerpo del ciclo se ejecuten por lo menos una vez, se recomienda el uso del *ciclo repetir / hasta*. Ejemplo:

```

Flujo.imprimir ("Ingrese un entero positivo:")
entero num
repetir
    num = Flujo.leerEntero( )
hasta (num < 0)

```

El apéndice B, ítem B2, presenta una tabla con un resumen de las estructuras de control.

Para llevar a la práctica la estructura de control ciclo con sus tres posibilidades (mientras, para y hacer / mientras), se tratará en la siguiente sesión un problema sencillo que permite establecer diferencias entre los tipos de ciclos.

PROBLEMA 10. PRIMEROS CIENTO NATURALES

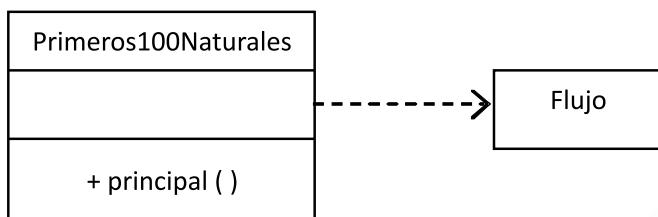
Mostrar los números enteros del 1 al 100.

Solución:

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R	Mostrar los números enteros del 1 al 100	Ninguna	Los primeros 100 números naturales se han mostrado en pantalla

b) Diagrama de clases



c) Contrato de la clase *Primeros100Naturales*

Nombre del método	Requerimiento asociado	Precondición	Postcondición	Modelo verbal
principal()	R	No se han visualizado en pantalla los números solicitados	Los números del 1 al 100 han sido impresos en pantalla	<ol style="list-style-type: none"> 1. Declarar un contador e inicializarlo en 1. 2. Mientras el contador sea menor o igual a 100: <ol style="list-style-type: none"> 2.1. Mostrar el contador 2.2. Incrementar el contador en una unidad

d) Seudo código orientado a objetos (OO)

```

clase Primeros100Naturales
  público principal ( )
    entero numero = 1

    mientras (numero <= 100)
      Flujo.imprimir (numero)
      numero = numero + 1
    fin_mientras
  fin_método
fin_clase

```

Observaciones:

- La clase se ha denominado *Primeros100Naturales*, en lugar de *Proyecto* como nombre genérico que aplica a cualquier solución.
- El *ciclo mientras* es una *sentencia de control* que permite la ejecución reiterada de un conjunto de instrucciones en tanto que una expresión lógica sea verdadera.

Este problema se puede resolver con los ciclos **para** y **repetir/hasta**, obteniendo el mismo efecto final, aunque un determinado ciclo aplica más que otro dependiendo de la circunstancia.

— Solución con ciclo **para**:

```
clase Primeros100Naturales
  público principal ()
    entero numero

    para (numero = 1, numero <= 100, numero = numero + 1)
      Flujo.imprimir (numero)
    fin_para
  fin_método
fin_clase
```

Notar que la inicialización y el incremento de la variable *numero*, así como la condición que garantiza el fin de las iteraciones, van todas ubicadas en la cabecera del ciclo y encerradas entre paréntesis.

— Solución con ciclo **repetir / hasta**:

```
clase Primeros100Naturales
  público principal ()
    entero numero = 1

    repetir
      Flujo.imprimir (numero)
      numero = numero + 1
    hasta (numero < 100)
  fin_método
fin_clase
```

Observar que el bucle **repetir / hasta** carece de fin, porque la palabra reservada **repetir** determina el inicio y la palabra **hasta** determina el fin de la estructura.

- La finalización de un método, un ciclo o una clase se especifica con las cláusulas `fin_metodo`, `fin_mientras`, `fin_para` y `fin_clase`, a excepción del ciclo `repetir / hasta que`, como se acaba de indicar, carece de cláusula de finalización.

3.5. BANDERA O INTERRUPTOR

Una variable cuyo contenido adquiere uno de dos valores, se denomina *bandera*, *interruptor* o *switch*. Aunque una bandera puede ser de tipo entero, carácter o lógico, su tipo natural es este último, porque precisamente las variables lógicas admiten tan sólo los valores de verdad: cierto o falso.

Los interruptores sirven para controlar procesos alternos o para determinar si una condición en particular se cumple o no.

PROBLEMA 11. SUCESIÓN NUMÉRICA

Generar y visualizar los primeros n términos de la sucesión:

3, -1, 4, 0, 5, 1, 6, 2, 7, ...

Solución

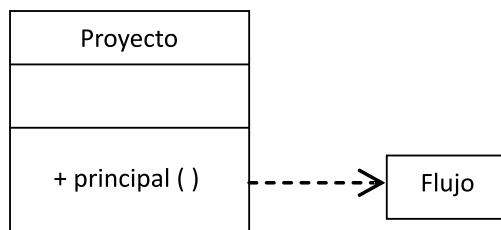
Pre análisis

En este problema se presenta un proceso alterno porque para generar cada término de la sucesión, que comienza en 3, se le suma de manera alternada los números -4 y 5. ¿Cómo controlar si se suma un valor o el otro? Una manera sencilla es por medio de una bandera que se inicializa en cierto para indicar que sumo -4 al término actual y que cambia a falso para indicar que sumo 5. La bandera cambiará de valor de forma reiterada y alternada dentro de un ciclo que contiene una sentencia de decisión: si el interruptor vale cierto, sumo -4 al término actual y cambio el switch a falso; de lo contrario sumo 5 y cambio la bandera a cierto.

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Saber la cantidad de términos de la serie	Un número entero (n)	Se conoce la cantidad de términos de la sucesión
R2	Generar la sucesión	n	Sucesión generada e impresa

b) Diagrama de clases



c) Contrato de la clase Proyecto

Método	Requerimiento asociado	Precondición	Postcondición
principal ()	R1 R2	Se desconoce la cantidad de términos a generar en la sucesión	La sucesión se ha mostrado al usuario

Nota: por su extensión la columna **Modelo verbal**, fue unida a la tabla y se presenta a continuación.

1. Iniciar término *ter* en 3 y bandera *band* en cierto.
2. Ingresar el valor de n .
3. En un ciclo controlado por n :

- Mostrar el término
- ¿La bandera es verdadera?
- Sí: Incrementar el término en -4
Poner bandera a falso.
- No: Incrementar el término en 5
Poner bandera a verdadero

d) Seudo código

```
clase Proyecto
  publico principal ()
    lógico band = cierto
    entero ter = 3, n, cont

    Flujo.imprimir("Ingrese la cantidad de términos de la sucesión: ")
    repetir
      n = Flujo.leerEntero()
    hasta (n <= 0)
    para (cont = 1, cont <= n, cont = cont + 1)
      Flujo.imprimir(cont)
      si (band)
        ter = ter - 4
        band = falso
      sino
        ter = ter + 5
        band = cierto
      fin_si
    fin_para
  fin_metodo
fin_clase
```

Este mismo ejercicio se resolverá con un vector en la sesión 4.1., en el problema 14 de este libro.

3.6. MÉTODOS RECURSIVOS

Un método es *recursivo* o *recursión directa* cuando se invoca a sí mismo. Cuando dos o más métodos se atraen, hablamos de *recursión indirecta*. Los problemas que a continuación se presentan son de recursión directa, y pueden graficarse mediante un pseudocódigo, veamos:

```
[visibilidad][<tipo_dato>] nombreMetodo(pf)
si (x)
    //Ejecución del caso base
sino
    nombreMetodo(pa) // Ejecución del caso recursivo
fin_si
retorne
```

Donde:

pf : Conjunto de *parámetros formales*.

x : Expresión que evalúa la llegada al caso base.

pa : Conjunto o lista de *parámetros actuales*.

Debe recordarse que la lista de parámetros actuales debe coincidir en número, orden y tipo con la lista de parámetros formales.

En general, todo algoritmo recursivo debe evaluar el caso base; si este no se presenta, se realiza la llamada recursiva con un parámetro que permita el acercamiento paulatino al caso base (decremento o incremento de una variable, cambio del estado de un objeto, etc.). Cuando se llega al caso base finaliza el proceso recursivo y se llega al resultado final por evacuación de las pilas de direcciones, parámetros y variables locales.

El seguimiento de la recursión implica el uso de la estructura de datos *Pila* para guardar las direcciones de retorno, valores de los parámetros y variables locales. Una pila, que se gestiona según la metodología FIFO (Primero en Entrar Primero en Salir, por sus siglas en inglés), forma parte del paquete de uso común **Contenedor expuesto**

brevemente en la sesión 2.4. En lo que sigue, se tratarán solo las estructuras de datos lineales y estáticas, vale decir, los arreglos; las *estructuras internas* restantes como las pilas, colas, listas, árboles y grafos quedan por fuera del alcance de este libro. El estudio de los algoritmos (métodos) recursivos se hará por medio de algunos problemas representativos.

PROBLEMA 12: FACTORIAL DE UN NÚMERO

Calcular el factorial de un número entero $n \geq 0$.

Solución:

Pre análisis:

Una definición iterativa (no recursiva) del factorial del número entero $n \geq 0$ es:

$$n! = \begin{cases} n * (n - 1) * (n - 2) * \dots * 2 * 1, & \text{si } n > 0 \\ 1, & \text{si } n == 0 \end{cases}$$

Ejemplos: $5! = 5 * 4 * 3 * 2 * 1 = 120$

$$0! = 1$$

La definición recursiva del factorial de un número – y de cualquier otro caso que implique al diseño recursivo – implica el planteamiento de la denominada *ecuación de recurrencia*, la cual constará siempre de dos casos: uno no recursivo llamado “caso base” y otro denominado “caso recursivo”, que al ejecutarse reiteradamente, acercará al caso base posibilitando la solución del problema.

La ecuación de recurrencia para el factorial de un número es la siguiente:

$$n! = \begin{cases} 1, & \text{si } n == 0 \text{ (caso base)} \\ n * (n - 1)!, & \text{si } n > 0 \text{ (caso recursivo)} \end{cases}$$

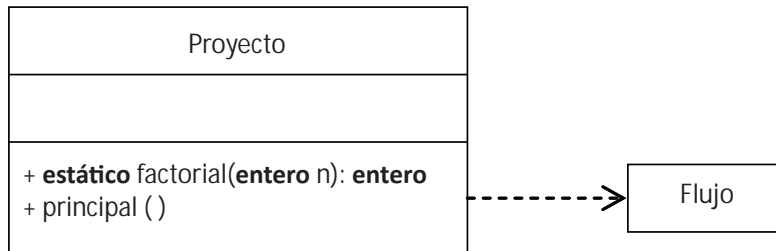
En términos algorítmicos, la representación matemática para el factorial de un valor entero n , dada por $n!$, se transforma en $\text{factorial}(n)$, y la ecuación de recurrencia se reescribe así:

$$\text{factorial}(n) = \begin{cases} 1, & \text{si } n == 0 \\ n * \text{factorial}(n-1), & \text{si } n > 0 \end{cases}$$

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultado (Salidas)
R1	Conocer la entrada del usuario	Un número entero	Número entero almacenado en memoria y listo para procesar
R2	Hallar el factorial de un número entero positivo o cero	Un número entero	El factorial del número ingresado.

b) Diagrama de clases



c) Responsabilidades de las clases

Contratos de la clase Proyecto:

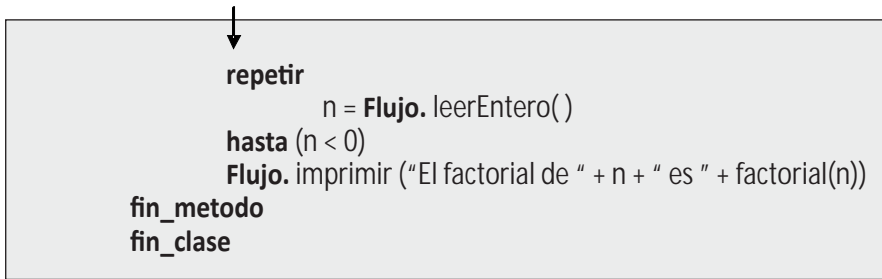
Método	Requerimiento asociado	Precondición	Postcondición	Modelo verbal
factorial(entero x)	R2	Se desconoce el factorial del número entero x	Se conoce el factorial del número entero x	Remitirse a la ecuación de recurrencia
principal()	R1 R2	Se desconoce el número entero al cual se le hallará el factorial	Se conoce el número entero ingresado por el usuario y su respectivo factorial	1. Ingresar el número entero. 2. Hallar el factorial del número (de manera recursiva)

d) Seudo código

```

Clase Proyecto
    entero estático factorial (entero x)
        entero f // variable local

        si (x == 0)
            f = 1 // Caso base
        sino
            f = x * factorial (x - 1) // Caso recursivo
        fin_si
        retornar(f)
    fin_metodo
    //-----
    principal()
        entero n
        Flujo. imprimir ("Ingrese un entero >= 0:")
  
```



Observaciones:

- El método *factorial()* tiene un *parámetro formal* de entrada x , tal como lo indica la flecha. Dicho parámetro es del mismo tipo que el *argumento* (*parámetro actual*, o de *llamada*) n del método *principal()*.
- La llamada recursiva es limitada por el caso base; las reiteradas llamadas recursivas simulan un ciclo transparente al programador.

PROBLEMA 13: CÁLCULO DE UN TÉRMINO DE FIBONACCI

Calcular y mostrar el n -ésimo término de la serie de Fibonacci, dada por 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Solución:

Pre análisis

La serie a generar fue ideada por el matemático italiano Leonardo Fibonacci a partir del estudio reproductivo de dos conejos (primer par de elementos de la serie).

Se observa que todo término de la serie se genera a partir de la suma de los dos anteriores, exceptuando los dos primeros, considerados semillas de la serie. La ecuación de recurrencia para generar el n -ésimo término de la serie está dada por:

$$\text{fibonacci}(n) = \begin{cases} 1, & \text{si } n == 1 \text{ ó } n == 2 \\ \text{fibonacci}(n-1) + \text{fibonacci}(n-2), & \text{si } n > 2 \end{cases}$$

Ejemplos:

$\text{fibonacci}(1) = 1$ (caso base 1)

$\text{fibonacci}(2) = 1$ (caso base 2)

$\text{fibonacci}(3) = \text{fibonacci}(2) + \text{fibonacci}(1) = 2$

$\text{fibonacci}(4) = \text{fibonacci}(3) + \text{fibonacci}(2) = 3$

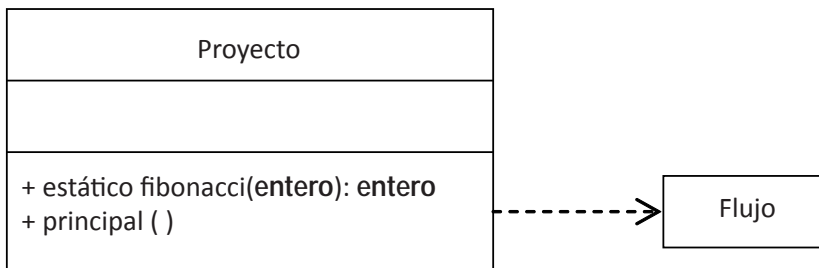
...

$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2) = x$

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultado (Salidas)
R1	Conocer el valor de n (número del término de Fibonacci a generar)	Un número entero	Número entero almacenado en memoria y listo para procesar
R2	Hallar el n_ésimo término de la serie de Fibonacci	El número entero n.	El n_ésimo término de la serie de Fibonacci

b) Diagrama de clases



c) Responsabilidades de las clases

Contrato de la clase Proyecto:

Método	Requerimiento asociado	Precondición	Postcondición	Modelo verbal
Fibonacci (entero n)	R2	Se desconoce el n-ésimo término de la serie de Fibonacci.	Se conoce el n-ésimo término de la serie de Fibonacci.	Remitirse a la ecuación de recurrencia
principal ()	R1 R2	Se desconoce el número del término a generar de la serie de Fibonacci.	Se conoce el número entero correspondiente al n-ésimo término de la serie de Fibonacci.	1. Ingresar el número entero n. 2. Hallar el n-ésimo término de la serie de Fibonacci.

d) Seudo código

↓

```

Clase Proyecto
    entero fibonacci (entero n)
        entero ter //ter: término de Fibonacci a encontrar
        entero ter1, ter2

        si (n == 1 || n == 2)
            ter = 1
        sino
            ter1 = fibonacci(n - 1)
            ter2 = fibonacci(n - 2)
            ter = ter1 + ter2

        fin_si
        retornar(ter)
    fin_metodo
//-----
principal()
    entero n
    Flujo.imprimir ("Ingrese el número del término de la serie de
                    Fibonacci que desea encontrar:")

    repetir
        n = Flujo.leerEntero()
    hasta (n < 0)
    Flujo.imprimir ("El " + n + "ésimo término de la serie de
                    Fibonacci es " + fibonacci(n))
    fin_metodo
fin_clase
  
```

Observaciones:

- El método *fibonacci()* tiene un *parámetro formal* *n*, cuyo nombre coincide con el *argumento* o *parámetro de llamada* utilizado en el método *principal()*. Los argumentos y los parámetros pueden tener igual nombre, pero esto no significa que se trate del mismo espacio de memoria; en ambos casos se trata más bien de variables locales, una pasada como parámetro al método *fibonacci()*, y otra definida dentro del método *principal()*. En general, es preferible utilizar nombres distintos entre los argumentos y parámetros correspondientes, aunque esto queda a decisión personal del programador.
- El método *fibonacci()* presenta dos llamadas recursivas:
 $ter1 = fibonacci(n - 1)$
 $ter2 = fibonacci(n - 2)$

Los productos de cada llamada se almacenan en variables locales, cuyos contenidos son sumados posteriormente para así obtener el término deseado.

Las variables locales *ter1* y *ter2* se pueden omitir si se utiliza la instrucción:

$$ter = fibonacci(n - 1) + fibonacci(n - 2)$$

Incluso, el método *fibonacci()* admite una nueva versión con una variable local infaltable — el parámetro *n*—, como se ilustra a continuación:

```
entero fibonacci (entero n)
    si (n == 1 || n == 2)
        retornar(1)
    sino
        retornar( fibonacci(n - 1) + fibonacci(n - 2))
    fin_si
fin_metodo
```


3.7. EJERCICIOS PROPUESTOS

Métodos

1. Escriba métodos que permitan:
 - Hallar el dato mayor entre tres elementos.
 - Visualizar la media, mediana y desviación típica para un conjunto de datos.
 - Imprimir el valor total de la serie:

$$(1 - x) + (3 + x)^2 + (5 - x)^4 + (7 + x)^8 + \dots$$
 El método recibe como parámetros el valor de x y la cantidad de términos a generar.

Sentencias de control

2. Solucionar el problema nueve (Estadísticas por procedencia), utilizando otra clase además de la que contiene el método *principal()*. El enunciado de dicho problema es el siguiente:

Para un número determinado de personas se conoce su estatura, procedencia y edad. La estatura y la procedencia se manejan de acuerdo a las siguientes convenciones:

Estatura = 1 (alta), 2 (baja) o 3 (Mediana)

Procedencia = 'L' (Americana), 'E' (Europea), 'A' (Asiática) u 'O' (Otra).

Determinar:

- El número de americanos altos, europeos bajos y asiáticos medianos.
 - La edad promedio de los individuos de otras procedencias.
 - La cantidad de americanos bajos mayores de edad.
3. Hallar los datos mayor y menor entre una muestra de n números enteros ingresados por el usuario.
Presentar dos estilos de solución: una con aplicación de sentencias de control según lo explicado en este capítulo, otra con aplicación de las clases de uso común, según el capítulo 2.

Recursión

4. Sean b y p números enteros mayores o iguales a cero. Calcular b^p : el número b elevado a la potencia p .
5. El máximo común divisor de los enteros x y y es el mayor entero que divide tanto a x como a y . La expresión $x \% y$ produce el residuo de x cuando se divide entre y . Defina el máximo común divisor (mcd) para n enteros x y y mediante:

$$\text{mcd}(x, y) = y \text{ si } (y \leq x \ \&\& \ x \% y == 0)$$

$$\text{mcd}(x, y) = \text{mcd}(y, x) \text{ si } (x < y)$$

$$\text{mcd}(x, y) = \text{mcd}(y, x \% y) \text{ en otro caso.}$$

Ejemplos.: $\text{mcd}(8, 12) = 4$, $\text{mcd}(9, 18) = 9$, $\text{mcd}(16, 25) = 1$, $\text{mcd}(6, 15) = 3$.

6. Suponga que $\text{com}(n, k)$ representa la cantidad de diferentes comités de k personas que pueden formarse, dadas n personas entre las cuales elegir. Por ejemplo $\text{com}(4, 3) = 4$, porque dadas cuatro personas A, B, C y D hay cuatro comités de tres personas posibles: ABC, ABD, ACD y BCD.

Para n valores, compruebe la identidad

$$\text{com}(n, k) = \text{com}(n-1, k) + \text{com}(n-1, k-1), \text{ donde } n, k \geq 1.$$

7. Los coeficientes binomiales pueden definirse por la siguiente relación de recurrencia, que es el fundamento del triángulo de Pascal:

$$c(n, 0) = 1 \text{ y } c(n, n) = 1 \text{ para } n \geq 0$$

$$c(n, k) = c(n-1, k) + c(n-1, k-1) \text{ para } n > k > 0.$$

Ejemplo: triángulo de Pascal de altura 5.

```

      1
    1  1
  1  2  1
1  3  3  1
1  4  6  4  1
    
```

Defina $c(n, k)$ para dos enteros cualquiera $n \geq 0$ y $k \geq 0$, donde $n \geq k$.

8. Multiplicar n pares de números naturales por medio de la recurrencia

$$a * b = \begin{cases} a & \text{si } b = 1 \\ a * (b - 1) + a & \text{si } b > 1 \end{cases}$$

donde a y b son enteros positivos.

9. La función de Ackerman se define para los enteros no negativos m y n de la siguiente manera:

$$a(m, n) = n + 1 \text{ si } m = 0.$$

$$a(m, n) = a(m - 1, 1) \text{ si } m \neq 0, n = 0.$$

$$a(m, n) = a(m - 1, a(m, n - 1)) \text{ si } m \neq 0, n \neq 0.$$

Compruebe que $a(2, 2) = 7$.

10. El siguiente es un algoritmo recursivo para invertir una palabra:

```

si (la palabra tiene una sola letra)
    Esta no se invierte, solamente se escribe.
sino
    Quitar la primera letra de la palabra.
    Invertir las letras restantes
    Agregar la letra quitada.
fin_si
    
```

Por ejemplo, si la palabra a invertir es ROMA, el resultado será AMOR; si la palabra es ANILINA (cadena capicúa), el resultado será ANILINA.

Invertir n palabras, donde la variable n debe ser especificada por el usuario.

3.8. REFERENCIAS

[Aristizábal2007]: Aristizábal, Diego Alejandro (2007). Un Cuento para tu Ciudad en Cien Palabras. Metro de Medellín, Tercer puesto.

[Bobadila2003]: Bobadilla, Jesús. Java a través de ejemplos. Alfaomega-Rama, Madrid, 2003.

[Eckel2002]: Eckel, Bruce. Piensa en Java. Segunda edición, Pearson Educación, Madrid, 2002.

[Oviedo2004]: Oviedo, Efraín. Lógica de Programación. Segunda Edición. Ecoe Ediciones, Bogotá, 2004.

CAPÍTULO 4

Arreglos

- 4.1. Operaciones con arreglos
 - Declaración de un arreglo
 - Asignación de datos a un arreglo
 - Acceso a los elementos de un arreglo
 - Problema 14: Sucesión numérica almacenada en un vector
 - 4.2. La clase Vector
 - Problema 15: Unión de dos vectores
 - Problema 16: Búsqueda binaria recursiva
 - 4.3. La clase Matriz
 - Problema 17: Proceso electoral
 - 4.4. Ejercicios propuestos
 - 4.5. Referencias
-

ESE INVOLVIDABLE CAFECITO

Juan Carlos Vásquez

Un día en el Instituto nos invitaron -a los que quisiéramos acudir-, a pintar una pobre construcción que hacía de colegio y que era el centro de un poblado de chozas, cuyo nombre no puedo acordarme, en una zona muy marginal, muy pobre y muy apartada de nuestras urbanizaciones, aunque, no muy distante.

Voluntariamente, acudió todo el curso, acompañado de nuestros hermanos guías, los promotores de la iniciativa solidaria.

Fue un sábado muy temprano, cuando montados en nuestras dos cafeteras de autobuses todos tan contentos, armados con nuestras respectivas brochas, para pintar de alegría y de esperanza, los rostros de aquella desconocida gente.

Cuando llegamos, vimos como unas veinte chozas alrededor de una pobre construcción de cemento que hacía de colegio y, escuchamos la soledad escondida, excluida, perdida.

Nos pusimos manos a la obra: unos arriba, otros abajo; unos dentro, otros fuera. Como éramos como ochenta pintores de brocha grande, la obra duró tan solo unas tres o cuatro horas.

Pero, antes de terminar, nos llamaron para que descansáramos, y salimos para fuera y vimos una humilde señora que nos invitaba a tomar café. La señora, con toda la amabilidad, dulzura, y agradecimiento, nos fue sirviendo en unas tacitas de lata que íbamos pasando a otros después de consumirlo.

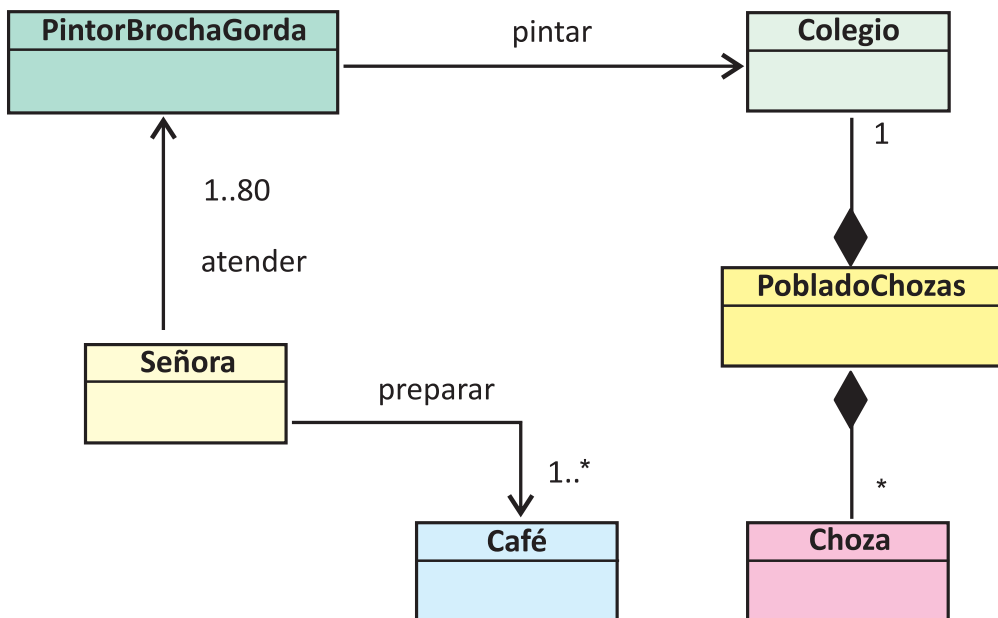
Nunca olvidaré ese olor y ese sabor de café, pues quedó grabado en mi memoria olfativa y gustativa para siempre. Nunca me han brindado un café tan rico como el que nos ofrecieron en ese día solidario.

Fue un café dado con todo el amor del mundo. Me supo a humanidad, me supo a gloria.

Fue mi mejor café, el café más rico del mundo.



Diagrama de clases:



Objetivos de aprendizaje

- Resolver problemas mediante la utilización de las estructuras de datos contenedoras lineales Vector y Matriz.
- Manejar arreglos unidimensionales y bidimensionales de tipos estándar y tipos abstractos de datos.

Un *arreglo* es una *estructura de datos lineal y estática*, con nombre único, compuesta por elementos del mismo tipo; se considera lineal porque para cada elemento del arreglo, siempre existirán un predecesor o elemento anterior y un sucesor o elemento siguiente (exceptuando los elementos primero y último del arreglo), dando la sensación abstracta de linealidad en la disposición de los elementos que componen la estructura; se considera estática porque la memoria de computadora (RAM – Random Access Memory) que ocupará la estructura se debe definir en tiempo de compilación y no en tiempo de ejecución, como ocurre con las estructuras dinámicas como las listas ligadas.⁷ Esto significa que el programador debe conocer la máxima cantidad de elementos del arreglo para poder dimensionarlo antes de proceder con la ejecución del programa.

4.1. OPERACIONES CON ARREGLOS

Los arreglos se clasifican de acuerdo a su dimensión así:

- Unidimensionales o vectores.
- Bidimensionales o matrices.

⁷ Las estructuras de datos dinámicas como las listas enlazadas, así como la implementación dinámica de otras estructuras contenedoras de datos como las pilas, colas, árboles y grafos, forman parte de un segundo curso de lógica de programación. Para profundizar estas estructuras remitirse a las referencias [Cairó 1993], [Weiss1995] y [Flórez 2005].

- N-dimensionales, donde $N > 2$

Los arreglos pueden contener muchos elementos del mismo tipo, a modo de paquete de datos que se puede referenciar a través de un solo nombre; por esto se dice que un arreglo es una *estructura de datos contenedora*. Debe tenerse en cuenta que todo arreglo se debe declarar y dimensionar antes de comenzar a utilizarlo.

DECLARACIÓN DE UN ARREGLO

En general, la *declaración de un arreglo* tiene la siguiente sintaxis:

$\underbrace{\text{<tipo_dato> nomArreglo[] [[] \dots []]} = \text{nuevo}}_{\text{Declaración}}$	$\underbrace{\text{<tipo_dato> [d1] [[d2] \dots [dn]]}}_{\text{Dimensionamiento}}$
--	--

Donde:

El miembro izquierdo de la asignación corresponde a la declaración del arreglo, y se explicita el tipo de datos del arreglo, su nombre y cantidad de dimensiones (cantidad de pares de corchetes vacíos); la parte derecha hace referencia al *dimensionamiento del arreglo*, es decir, al espacio de memoria que utilizará el arreglo.

<tipo_dato>: Es el tipo de datos del arreglo. Puede ser un tipo estándar (entero, real, caracter, logico, cadena), un tipo abstracto de datos (Producto, Cuenta, Planta, etc.) o un tipo estándar clasificado (Entero, Real, Caracter, Logico, Cadena).

nomArreglo: Es el nombre dado al arreglo, que debe cumplir todas las características para la construcción de identificadores.

[] [[] \dots []]: Son pares de corchetes, donde cada par indica una nueva dimensión del arreglo; del segundo al último son opcionales.

nuevo: Es una palabra reservada del pseudo lenguaje con la cual se solicita memoria para el arreglo. Como se verá, con esta misma palabra se podrán crear objetos.

[d1] [[d2]... [dn]]: Son números enteros que determinan la máxima capacidad de cada dimensión del arreglo. No hay un límite para la cantidad de dimensiones, simplemente se debe tener claro que $n > 0$.

Si se quiere definir un *vector* se especifica sólo la dimensión d1, para una *matriz* se especifican las dimensiones d1 y d2, para un arreglo de tridimensional las dimensiones d1, d2 y d3, y así sucesivamente.

Observar que las dimensiones d2 a dn son opcionales, por esto se encierran entre un par de corchetes adicionales.

Ejemplos:

entero valores[] = **nuevo** entero[100]

real numeros[] [] = **nuevo** real[4] [5]

cadena nombres[] [] [] = **nuevo** cadena[2] [3] [5]

En el primer ejemplo se ha declarado el vector *valores* de 100 enteros, en el segundo la matriz *numeros* de 20 reales (en cuatro filas y 5 columnas) y en el tercero el arreglo tridimensional *nombres* con 30 cadenas (2 filas, 3 columnas y 5 planos).

La declaración y el *dimensionamiento de un arreglo* se pueden independizar en dos instrucciones, de la siguiente manera:

<tipo_dato> nomArreglo[] []... [] // Declaración

nomArreglo = **nuevo** <tipo_dato> [d1] [[d2]... [dn]] // Dimensionamiento

Reescribiendo los ejemplos anteriores, se tiene:

a) **entero** valores[]
valores = **nuevo** entero[100]

b) `real numeros[] []`

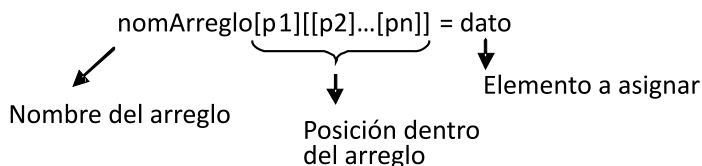
`números = nuevo real[4] [5]`

c) `cadena nombres[] [] []`

`nombres = nuevo cadena[2] [3] [5]`

ASIGNACIÓN DE DATOS A UN ARREGLO

La asignación de datos a un arreglo debe seguir la sintaxis:



Donde:

`p1, p2, ..., pn`: son números enteros válidos para alguna posición del arreglo.

`dato`: un dato del mismo tipo del arreglo.

Algunos ejemplos de asignación de datos en los arreglos ya citados:

`valores [0] = 7`

`valores [24] = 238`

`numeros[1] [4] = 2,718`

`numeros[3] [0] = 9,81`

`nombres[0] [0] [1] = "guayasamín"`

`nombres[1] [2] [3] = "las acacias"`

También se pueden asignar datos definidos por el usuario a todas las posiciones del vector, mediante un ciclo y una instrucción de lectura. Así, el vector *valores* se puede asignar – este proceso también se denomina llenado del vector- de la siguiente manera:

```

entero i
para (i = 0, i < 100, i = i + 1)
    Flujo.imprimir("Ingrese un número entero: ")
    valores[i] = Flujo.leerEntero()
fin_para

```

ACCESO A LOS ELEMENTOS DE UN ARREGLO

El *acceso a los elementos de un arreglo* se hace escribiendo el nombre del mismo y entre corchetes la posición que ocupa el elemento referenciado. La posición está dada por un número entero que se puede especificar a manera de constante o variable. Algunos ejemplos:

- a) valores[i] = valores[i + 1]
- b) entero f, c


```

      para (f = 0, f < 4, f = f + 1)
          para (c = 0, c < 5, c = c + 1)
              Flujo.imprimir(numeros[f][c])
          fin_para
      fin_para
      
```
- c) nombres[i][j][k] = Cadena.concatenar("valor ", "agregado")

PROBLEMA 14: SUCESIÓN NUMÉRICA ALMACENADA EN UN VECTOR

Generar los primeros n términos de la sucesión:

3, -1, 4, 0, 5, 1, 6, 2, 7, ...

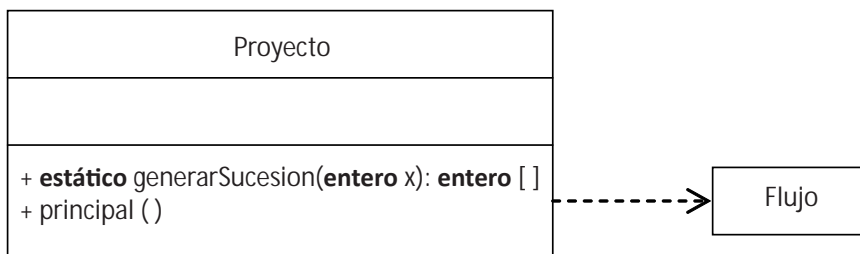
Solución:

Pre análisis

Aunque este problema es idéntico al número 11, ahora se soluciona con el uso de un vector donde se almacenarán los elementos de la sucesión. La tabla de requerimientos

no cambia con respecto a la ya planteada; sin embargo, el diagrama de clases, responsabilidades y pseudo código se reestructuran como sigue.

a) Diagrama de clases



El método estático *generarSucesión()* recibe un parámetro *x* equivalente al número de términos de la sucesión, y retorna un vector de enteros con la sucesión almacenada.

En el método *principal()* se lee el número *n* (cantidad de términos de la sucesión) y se invoca al método *generarSucesion()* con la variable *n* como argumento. Es conveniente recordar que el método *principal()* no retorna valor y es de carácter estático.

b) Responsabilidades de las clases

Contratos de la clase Proyecto

Método	Reque- rimiento asociado	Precondición	Postcondición
generarSucesion()	R2	Conocer el valor de <i>n</i>	La sucesión se ha generado y se encuentra almacenada en un vector
principal ()	R1 R2	Se desconoce la cantidad de términos de la sucesión	La sucesión se ha mostrado al usuario

c) Seudo código

```

clase Proyecto
  público:
    estatico generarSucesion(entero x): entero [ ]
      entero vec[ ] = nuevo entero[x]
      lógico band = cierto
      entero ter = 3, cont
      para (cont = 0, cont < x, cont = cont + 1)
        vec[cont] = nuevo entero( )
        vec[cont] = ter
        si (band)
          ter = ter - 4
          band = falso
        sino
          ter = ter + 5
          band = cierto
        fin_si
      fin_para
      retornar vec
    fin_metodo
  //-----
  publico principal ( )
    entero n
    Flujo.imprimir("Ingrese la cantidad de términos de la sucesión: ")
    repetir
      n = Flujo.leerEntero( )
    hasta (n <= 0)

    entero v[ ] = nuevo entero[n]
    v = generarSucesion(n)
    v.mostrar( )
  fin_metodo
fin_clase

```

Observaciones:

- El vector *v* declarado después del ciclo *repetir* / *hasta* del método *principal()*, es un arreglo local unidimensional que recibirá el vector devuelto por el método *generarSucesion()*.
- La sucesión es mostrada al usuario a través del método *mostrar()* de la clase *Vector*, definida a continuación.

4.2. LA CLASE VECTOR

Un *vector* es un arreglo unidimensional que puede almacenar elementos de cualquier tipo primitivo de datos o referencias a objetos. La figura 4.1. representa los vectores *valores*, *letras* y *frutas* de tipo *entero*, *caracter* y *cadena*, respectivamente.

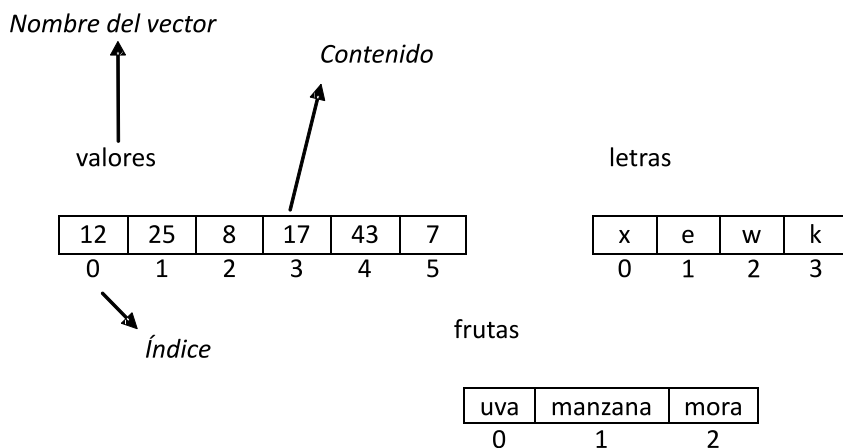


Figura 4.1. Tres vectores de diferente tipo

El vector *valores* de tipo *entero* contiene seis números, el vector *letras* de tipo *caracter* contiene cuatro caracteres alfabéticos, y *frutas* de tipo *cadena* posee tres nombres de frutas. Cada elemento tiene una posición absoluta, dada por un número entero denominado *índice*. Así, el contenido del vector *valores* en su posición 3 es igual a 17, y se representa por: $\text{valores}[3] = 17$. De manera similar se debe notar que:

$\text{valores}[0] = 12$

$\text{valores}[1] = 25$

$\text{valores}[5] = 7$

$\text{valores}[6]$ no existe.

$\text{letras}[0] = \text{'x'}$

$\text{letras}[1] = \text{'e'}$

$\text{letras}[2] = \text{'w'}$

$\text{letras}[-1]$ no existe.

$\text{frutas}[0] = \text{"uva"}$

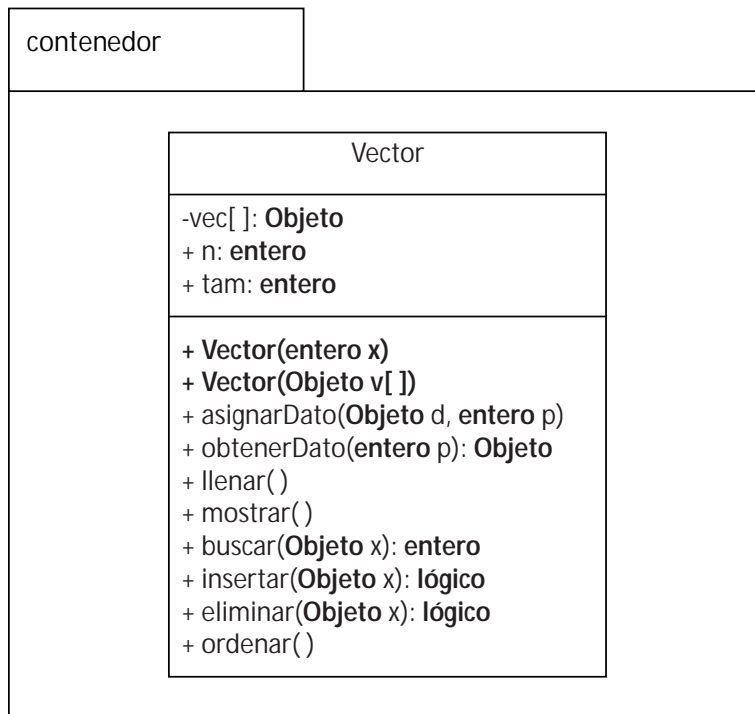
$\text{frutas}[1] = \text{"manzana"}$

frutas[2] = "mora"
frutas [3] no existe

Observaciones:

- Los nombres de los vectores se ciñen a las reglas de cualquier identificador. Comienzan con letras, no admiten espacios, no pueden coincidir con palabras reservadas del seudo lenguaje, admiten el guión bajo en cualquier posición. Así, otros nombres válidos para vectores serían:
teléfonos_empresa, tel, vec, v5, v
- Recordar que las cantidades no incluyen comillas, pero los caracteres una comilla y las cadenas dos. Cabe entonces diferenciar entre el carácter '8', el número 8 y la cadena de un carácter "8"; y entre el número 438 y la cadena "438".
- Cuando un índice sobrepasa los límites del vector, sea por debajo o por encima, ocurre una excepción de índice fuera de rango. Las *excepciones* son errores que se deben manejar en tiempo de ejecución para evitar una finalización anormal del programa.

La *clase Vector* se define de la siguiente manera:



Donde:

- La clase **Vector** forma parte del paquete de uso común **contenedor**, presentado en la sesión 2.4 (*Paquetes de uso común*). Este paquete contiene otras clases (**Matriz**, **ListaOrdenada**, **Pila**, **Cola**, **ABB**, **Grafo** y **Archivo**) conocidas como *estructuras de datos internas* debido a que su objetivo es optimizar el manejo de grandes cantidades de datos en la memoria interna de la computadora. En este libro nos ocuparemos de las clases **Vector** y **Matriz**.
- La *clase* **Vector** posee tres atributos: un arreglo unidimensional de visibilidad privada denominado *vec*, que guarda datos de tipo **Objeto**; una variable *n* de visibilidad pública que almacena el número de datos que guarda el vector; y una variable *tam*, también de visibilidad pública, que contiene la máxima cantidad de elementos que puede almacenar el vector. Tener en cuenta el rango del atributo *n*: $0 \leq n \leq \text{tam}$.

Los atributos *ny tam* son públicos porque usualmente se utilizan en las diferentes operaciones que pueden hacerse sobre un vector: recorrido, búsqueda de un elemento, ordenamiento, entre otras.

- Esta clase maneja dos constructores. El primero, *Vector(entero x)*, tiene como parámetro el número de elementos que va a guardar el vector. Al tamaño *tam* se le suma 10 con el objetivo de conservar espacio de memoria adicional para futuras adiciones de elementos; este valor es subjetivo y puede variar de acuerdo a la conveniencia del caso. Este constructor inicializa en **nulo** todos los elementos del arreglo, como se detalla a continuación:

Vector (entero x)

entero i

n = x

tam = x + 10

vec = nuevo Objeto[x]

para (i = 0, i < x, i = i + 1)

vec[i] = nuevo Objeto()

vec[i] = nulo

fin_para

fin_metodo

Para crear un vector v de tamaño cien se escribe:

```
Vector v = nuevo Vector(100)
```

El segundo constructor, `Vector(Objeto v[])`, tiene como parámetro un arreglo unidimensional de tipo `Objeto`, e inicializa el atributo `vec` con los valores que guardan los elementos del parámetro v por medio de una única y simple operación de asignación, como se observa a continuación:

```
Vector(Objeto v[ ])
vec = v
n = v.n
tam = v.tam
fin_metodo
```

Para crear un vector $v1$ que guarde los datos del arreglo $v2$ se escribe:

```
real v2[ ] = {1, 2, 3, 4}
Vector v1 = nuevo Vector(v1)
```

- Los métodos restantes se describen así:
 - `asignarDato(Objeto d, entero p)` → Asigna el dato d a la posición p del vector vec . Este método permite modificar el estado del vector porque el contenido de $vec[p]$ cambia por d .
 - `obtenerDato(entero p): Objeto` → Retorna el contenido de $vec[p]$, es decir, permite conocer el estado del vector en su posición p .
 - `llenar()` → Llena el vector con n datos de tipo `Objeto`.
 - `mostrar()` → Imprime el contenido del vector vec .
 - `buscar(Objeto x): entero` → Busca el dato x en el vector vec y retorna la posición (índice del vector) donde fue hallado. Si x no existe retorna un valor igual a -1.

- **insertar(Objeto x): lógico** → Inserta el dato x al final del vector vec , es decir, en la posición $n + 1$. Si la inserción tuvo éxito se retorna **cierto**; si no hay espacio disponible porque $tam == n$, se retorna **falso**.
- **eliminar(Objeto x): lógico** → Elimina el dato x en el vector vec . Retorna **cierto** si x fue eliminado o **falso** en caso contrario.
- **ordenar()** → Ordena el vector vec de menor a mayor.

El pseudocódigo para la *clase Vector* es el siguiente:

```

clase Vector
  privado Objeto vec[ ]
  publico entero n
  publico entero tam

  publico:
    Vector(entero x)
      entero i
      n = x
      tam = x
      vec = nuevo Objeto[x]
      para (i = 0, i < x, i = i + 1)
        vec[i] = nulo
      fin_para
    fin_metodo
    //-----
    Vector(Objeto v[ ])
      vec = v
      n = v.n
      tam = v.tam
    fin_metodo
    //-----
    asignarDato (Objeto d, entero p)
      vec[p] = d
    fin_metodo
    //-----
    Objeto obtenerDato (entero p)
      retornar vec[p]
    fin_metodo
    //-----

```



```

↓
llenar()
    entero i
    para (i = 0, i < n, i = i + 1)
        vec[i] = Flujo.leerObjeto()
    fin_para
fin_metodo
//-----
cadena mostrar()
    cadena texto = " "
    entero i
    para (i = 0, i < n, i = i + 1)
        texto = texto + vec[i].aCadena() + " "
    fin_para
    retornar texto
fin_metodo
//-----
entero buscar(Objeto x)
    entero pos = 0 // Contador para recorrer el vector
    lógico hallado = falso // Supuesto: x no existe
    // Mientras no encuentre a x y existan
    // elementos por comparar
    mientras (hallado == falso && pos < n)
        si (vec[pos].comparar(x) == 0) // ¿Lo encontró?
            hallado = cierto
        sino
            pos = pos + 1 // Avance a la siguiente posición
    fin_si
fin_mientras
si (hallado == falso) // ¿No encontró a x en vec?
    pos = -1
fin_si
retornar pos
fin_metodo
//-----
lógico insertar(Objeto x)
    lógico indicativo = falso // No se ha insertado
    entero p = buscar(x) // Verifica si el dato a
    insertar existe en vec
    si (p == -1) // ¿No encontró el dato x?
    si (n < tam) // ¿Hay espacio disponible en el
        vector?
        vec[n] = x
        n = n + 1
        indicativo = cierto // Inserción realizada
    Flujo.imprimir("Inserción realizada")

```



```

↓
    sino
        Flujo.imprimir("No hay espacio para la inserción")
    fin_si
sino
    Flujo.imprimir(x.aCadena() + " ya existe")
fin_si
retornar indicativo
fin_metodo
//-----
entero eliminar(Objeto x)
    entero p = buscar(x) // Verifica si el dato a eliminar existe en vec
    si (p != -1) ¿Encontró a x en la posición p?
        // Desde p, desplazar los elementos de vec una posición hacia atrás
        para (i = p, i < n - 1, i = i + 1)
            vec[i] = vec[i + 1]
        fin_para
        n = n - 1 // Borrado lógico de la última posición
        Flujo.imprimir (x.aCadena() + " eliminado")
    sino
        Flujo.imprimir(x.aCadena() + " no existe")
    fin_si
    retornar p
fin_metodo
//-----
ordenar()
    entero i, j
    Objeto aux
    para (i = 0, i < n - 1, i = i + 1)
        para (j = i + 1, j < n, j = j + 1)
            si (vec[i].comparar(vec[j]) > 0)
                aux = vec[i]
                vec[i] = vec[j]
                vec[j] = aux
            fin_si
        fin_para
    fin_para
fin_metodo
//-----
fin_clase

```

Observaciones:

- El método *mostrar()* utiliza la variable local *texto* de tipo *cadena* para crear una representación textual del vector a imprimir. Cada elemento del vector *vec* es convertido a texto mediante el método *aCadena()* de la clase *Objeto*, con el fin de concatenarlo a la variable *texto* a través el operador cruz (+).

A manera de ejemplo, para mostrar el vector *v* en pantalla, basta con escribir las instrucciones:

```
real v1[] = {1.7, 2.8, 3.9}
Vector v = nuevo Vector(v1)
v.mostrar()
```

- El método *buscar()* devuelve la posición -número entero- donde fue hallado el dato *x*. En el proceso de búsqueda del elemento, se utiliza el método *comparar()* de la clase *Objeto*.
- Los métodos *insertar()* y *eliminar()* verifican si el dato *x* existe o no dentro del vector *vec*, porque este contiene claves de búsqueda -no puede contener elementos repetidos-. En los ejercicios propuestos del capítulo se plantea un problema sobre un vector que puede contener elementos duplicados.
- El método *insertar()* inserta el dato *x* al final del vector *vec*, siempre y cuando exista espacio disponible.
- La eliminación de un elemento del vector implica un borrado lógico en lugar de un borrado físico. En la figura 4.2. se observa el proceso que se lleva a cabo cuando se desea eliminar el dato *x*, ubicado en la posición *pos* del vector *vec*, que en este caso almacena números enteros.

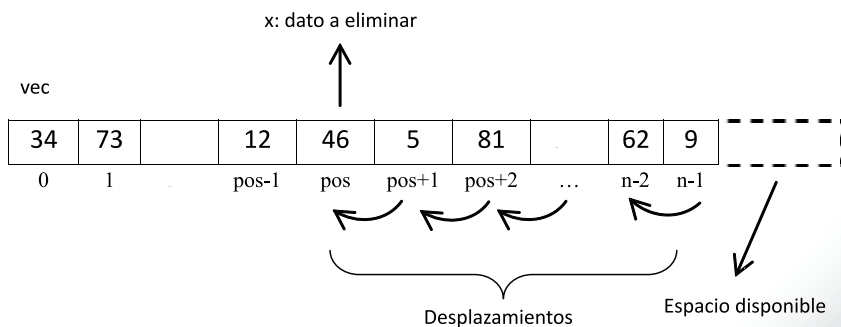


Figura 4.2. Eliminación de un dato en un vector

- El ordenamiento ascendente del vector se lleva a cabo a través del método de la burbuja, que compara el contenido de la primera posición con las demás que le siguen, el contenido de la segunda con todas las que le preceden, y así sucesivamente hasta llegar a comparar el contenido de la penúltima posición con el de la última. Cuando la comparación es verdadera, es decir, cuando se cumple $vec[i].comparar(vec[j]) > 0$ (esto significa que el contenido del vector en la posición i es mayor que el contenido del vector en la posición j), se debe realizar el intercambio de contenido de las dos posiciones, lo que se logra con la variable auxiliar *aux*.

El método de la burbuja tiene varias versiones; aquí se ha presentado una de las más conocidas.

Si se analiza la eficiencia del método *ordenar()* por medio del cálculo de su *orden de magnitud*⁸, se llega a la conclusión que este tipo de ordenamiento es aplicable cuando el volumen de datos a tratar es pequeño, de lo contrario se hace ineficiente. Cuando la cantidad de datos es grande se pueden utilizar otros métodos como el ordenamiento rápido (quick sort), por inserción, selección o montículo (heap sort), que tienen un mejor comportamiento en el tiempo de ejecución. En este sentido, el método *ordenar()* se puede sobrecargar siguiendo los lineamientos lógicos de otros métodos de ordenamiento.

PROBLEMA 15: UNIÓN DE DOS VECTORES

Se tienen dos vectores $v1$ y $v2$ de n y m elementos, respectivamente. Se pide: crear otro vector $v3$ con los elementos correspondientes a la unión de $v1$ y $v2$, ordenar el vector $v3$, e imprimir los tres vectores.

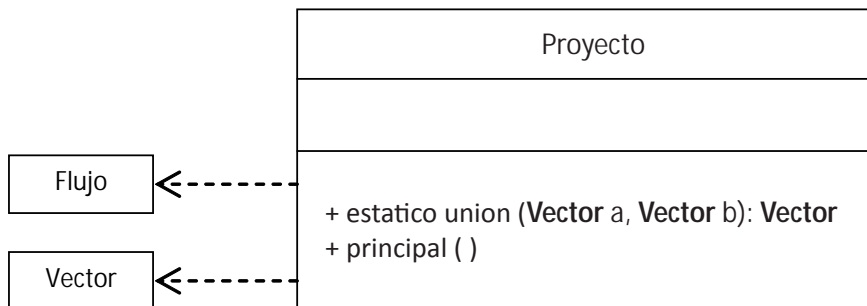
8 El orden de magnitud, conocido también como notación O Grande, es una medida para la eficiencia de un algoritmo. El método de la burbuja tiene orden de magnitud cuadrática, expresado como $O(n^2)$. Para mayor información sobre este tema remitirse a [Brassard 1997] y [Aho 1988].

Solución:

a) Tabla de requerimientos

Id. del req.	Descripción	Entradas	Resultados
R1	Conocer el tamaño de los vectores	Dos números enteros m y n	Se conoce el tamaño de los dos arreglos
R2	Conocer los datos del primer vector	m	El vector v1 ha sido creado
R3	Conocer los datos del segundo vector	n	El vector v2 ha sido creado
R4	Unir los vectores v1 y v2	m y n	El vector unión v3 ha sido creado
R5	Ordenar el vector unión	v3.n	El vector v3 está ordenado
R6	Visualizar los tres vectores	v3	Los vectores v1, v2 y v3 han sido impresos

b) Tabla de requerimientos



c) Contrato de la clase Proyecto

Método	Reque- rimiento asociado	Precondición	Postcondición
Vector union (Vector a, Vector b)	R4	No se han unido los vectores <i>a</i> y <i>b</i>	La unión de los vectores <i>a</i> y <i>b</i> se encuentra almacenada en un tercer vector
principal ()	R1, R2, R3, R4, R5 y R6	No existen vectores <i>a</i> procesar	Se conoce la unión de los vectores <i>v1</i> y <i>v2</i>

d) Seudo código

Clase Proyecto**publico:****estatico Vector union(Vector a, Vector b)****Vector c = nuevo Vector (a.n + b.n)****entero i, p**// Copiar el contenido del vector *a* en el vector *c***para** (i = 0, i < a.n, i = i + 1)

c.insertar(a.obtenerDato(i))

fin_para// Copiar en el vector *c* los elementos de *b* que no están en *a***para** (i = 0, i < b.n, i = i + 1)p = a.buscar(b.obtenerDato(i)) // buscar *b.vec[i]* en *a***si** (p != -1) // ¿ Lo encontró?

c.insertar (b.obtenerDato(i))

fin_si**fin_para****retornar** c**fin_metodo**

//-----

principal()**entero** m, n**Flujo.imprimir** ("Creación de dos vectores de tamaños m y n")**Flujo.imprimir** ("Ingrese el valor de m:")

```

↓
repetir
    m = Flujo.leerEntero( )
hasta (m < 0)
Flujo.imprimir ("Ingrese el valor de n:")
repetir
    n = Flujo.leerEntero( )
hasta (n < 0)

// Reserva de memoria para los dos vectores
Vector v1= nuevo Vector(m)
Vector v2 = nuevo Vector(n)
//Llenar los vectores v1 y v2
v1.llenar( )
v2.llenar( )
// Unir v1 y v2
Vector v3 = nuevo Vector(m + n)
v3 = union(v1, v2)
// Ordenar v3
v3.ordenar( )
// Visualizar el contenido de los tres vectores
v1.mostrar( )
v2.mostrar( )
v3.mostrar( )

fin_metodo
fin_clase

```

PROBLEMA 16: BÚSQUEDA BINARIA RECURSIVA

Escribir un método que busque un elemento dado como parámetro, dentro de un vector que se encuentra ordenado.

Solución:

Pre análisis

La *búsqueda binaria* se aplica a conjuntos de datos ordenados, almacenados en estructuras de datos lineales como vectores o listas enlazadas. Si los datos no se encuentran ordenados, se aplican otros métodos de búsqueda como la secuencial o la hash (por transformación de claves).

En este caso, nos ocuparemos de la búsqueda binaria en un vector, método que formaría parte de la clase *VectorOrdenado*. Un objeto perteneciente a esta clase es un arreglo unidimensional compuesto por elementos del mismo tipo, los cuales se hallan ordenados de forma ascendente (por omisión siempre se asume este tipo de ordenamiento).

Todos los métodos aplicados a la clase *VectorOrdenado* deben conservar el estado de ordenamiento de los datos. Una posible definición para esta clase sería la siguiente.

VectorOrdenado
- vec[]: Objeto + tam: entero + n: entero
+ VectorOrdenado(entero x) + llenarVector(entero x) + mostrarVector() + busquedaBinaria(Objeto x, entero b, entero a): entero + busquedaBinaria(Objeto x): entero + eliminarDato(Objeto x): logico + insertarDato(Objeto x): logico

Los métodos *llenarVector()*, *mostrarVector()*, *eliminarDato()* e *insertarDato()* quedan propuestos como ejercicios para el lector, y no necesariamente se deben tratar como métodos recursivos. Nos interesa por el momento el desarrollo del método recursivo *busquedaBinaria(Objeto x, entero b, entero a):entero*, y el método sobrecargado *busquedaBinaria(Objeto x):entero*, que es de naturaleza no recursiva.

Documentación del método *busquedaBinaria()* recursiva.

Según el diagrama de clases, el método forma parte de la clase *VectorOrdenado*, que a su vez contiene los miembros privados *vec*, *tam* y *n*, donde:


- vec* : nombre del vector ordenado.
- tam*: tamaño del vector, equivalente a la máxima cantidad de elementos que puede almacenar.
- n*: cantidad de elementos reales en el vector. Siempre se cumplirá que $n \leq tam$.

El método *busquedaBinaria(Objeto x, entero b, entero a):entero* retorna la posición *p* donde el dato *x* fue encontrado o retorna un valor igual a -1 si *x* no existe. $0 \leq p < n$

Los parámetros x , b y a adquieren el siguiente significado:

- x : dato a buscar en el vector ordenado.
- b : valor más bajo para un índice del arreglo. Su valor inicial es 0 (cero).
- a : valor más alto para un índice del arreglo. Su valor inicial es el atributo n .

Definición del método:



```

entero busquedaBinaria(Objeto x, entero b, entero a)
    entero p, m

    si (b > a)
        p = -1
    sino
        m = (a + b) / 2
        si (x < vec[m])
            p = busquedaBinaria(x, b, m - 1)
        sino
            si (x > vec[m])
                p = busquedaBinaria(x, m + 1, a)
            sino
                p = m
        fin_si
    fin_si
    fin_si
    retornar(p)
fin_metodo
  
```

Observaciones:

- Los tres parámetros del método son pasados por valor. La primera vez que se ejecute el método los valores de b y a serán 0 y $n-1$, respectivamente. (Recordar que el valor de n es igual a la cantidad de elementos presentes en el vector). El valor del parámetro x siempre será el mismo para las distintas llamadas recursivas.
- El método *busquedaBinaria()* recursiva busca el dato x dividiendo sucesivamente por mitades el vector *vec*. Si el dato x es menor que el dato ubicado en la posición media del vector (dada por m), se busca por la mitad izquierda del

vector mediante la llamada recursiva *busquedaBinaria* ($x, b, m - 1$); si el dato x es mayor que el dato ubicado en la posición media del vector, se busca por la mitad derecha del vector mediante la llamada recursiva *busquedaBinaria*($x, m + 1, a$); finalmente, si no se cumplen los dos casos anteriores se concluye que el dato fue hallado y se asigna a la posición p el valor de m . La finalización de la recursividad se presenta cuando el valor de b supera al valor de a (primer bloque de decisión del método).

4.3. LA CLASE MATRIZ

Una *Matriz* es un arreglo de dos dimensiones compuesto por elementos del mismo tipo, cada uno de ellos ubicado en una posición específica tanto por un número para la fila como para la columna. Toda matriz -y de hecho cualquier arreglo- tiene un nombre único que sirve para diferenciarlo de los demás en el marco de una aplicación.

La matriz *tabla* de m filas y n columnas es llamada “matriz de orden $m * n$ ” y se denota por *tabla*($m * n$); su presentación gráfica está dada por:

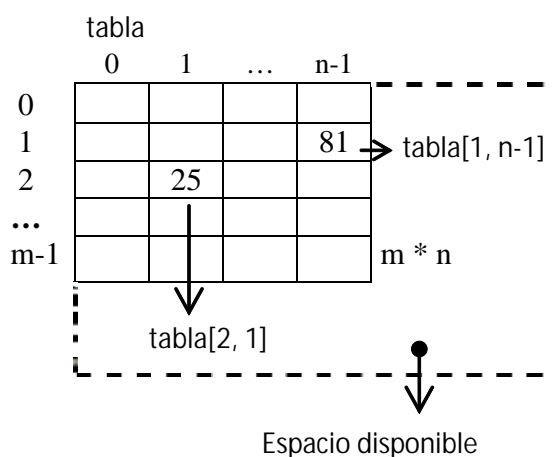
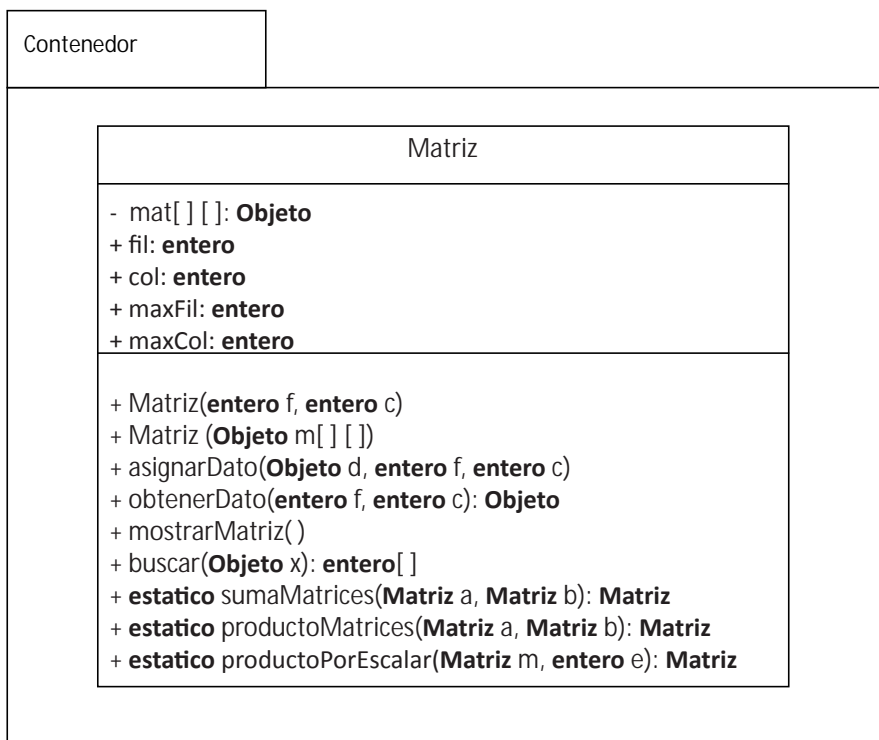


Figura 4.3. Representación de una matriz

Para referenciar los elementos de una matriz se escribe el nombre del arreglo y entre corchetes dos expresiones enteras separadas por comas; la primera correspondiente al número de la fila y la segunda al número de columna. Así, el primer elemento de toda matriz ocupa la posición $[0, 0]$. Así en la figura 4.3. puede observarse que $tabla[2, 1] = 25$ y $tabla[1, n-1] = 81$. Debe señalarse que, como en los vectores, debe existir un espacio disponible para realizar operaciones de inserción de nuevas filas o columnas.

La *clase* **Matriz** se puede definir de la siguiente manera:



Donde:

- La clase **Matriz** maneja cinco atributos: un arreglo bidimensional de objetos con visibilidad privada, denominado *mat*, y cuatro atributos públicos de tipo entero para la cantidad de filas, la cantidad de columnas, el número máximo de filas y de columnas, respectivamente.

Cabe anotar que $0 \leq \text{fil} \leq \text{maxFil}$ y $0 \leq \text{col} \leq \text{maxCol}$.

- Esta clase maneja dos constructores. Veamos:
Al primero se le pasa el número de filas y de columnas e inicializa todos sus miembros en cero.

```
Matriz (entero f, entero c)
    fil = f
    col = c
    maxFil = f + 10
    maxCol = c + 10
    mat = nuevo Objeto[f][c]
    entero i, j
    para (i = 0, i < f, i = i + 1)
        para (j = 0, j < c, j = j + 1)
            mat[i][j] = 0
        fin_para
    fin_para
fin_metodo
```

Para crear una matriz *m* de dimensión de orden 5 * 6, se escribe:

```
Matriz m = nuevo Matriz(5, 6)
```

Al segundo constructor se le pasa un arreglo bidimensional de objetos, para inicializar el miembro dato *mat* con los valores que guardan los elementos de dicho arreglo mediante una operación de asignación.

```
Matriz(Objeto m[ ][ ])
    mat = m
    fil = m.fil
    col = m.col
    maxFil = m.maxFil
    maxCol = m.maxCol
fin_metodo
```


Lo anterior, para crear la matriz *tabla*(4 * 3) con el siguiente contenido:

Tabla

2	1	0
4	3	9
6	5	10
8	7	11

A partir de la tabla se crea un arreglo bidimensional – para este caso denominado *cuadro*- y se le pasa al constructor de la clase **Matriz**, como se observa a continuación:

```
entero cuadro[ ][ ] = { {2, 1, 0}, {4, 3, 9}, {6, 5, 10}, {8, 7, 11} }
Matriz tabla = nuevo Matriz(cuadro);
```

- El método *asignarDato(Objeto d, entero f, entero c)*, asigna el objeto *d* a *mat[f, c]*.
- El método *obtenerDato(entero f, entero c): Objeto*, devuelve el dato almacenado en *mat[f, c]*.
- Los métodos restantes se describen así:
 - *mostrarMatriz()* → Muestra el contenido de la matriz *mat*.
 - *buscar(Objeto x): entero[]* → Busca el dato *x* en la matriz *mat*; y retorna un vector de tipo entero con la posición donde fue hallado el dato *x*. La primera posición del vector retornado contendrá el número de la fila y la segunda el número de la columna. Pero, si el dato *x* no existe, el vector contendrá dos valores iguales a -1.
 - *estatico sumaMatrices(Matriz a, Matriz b): Matriz* → Suma dos matrices del mismo orden y retorna otra matriz con el resultado de la suma.
 - *estatico productoMatrices(Matriz a, Matriz b): Matriz* → Multiplica dos matrices compatibles respecto al producto. Para que dos matrices se puedan multiplicar, el número de columnas de la primera debe coincidir con el número de columnas de la segunda. Por tanto, si las matrices a multiplicar son *a(m * n)* y *b(n * p)*, el resultado será una matriz de orden *m * p*.
 - *estatico productoPorEscalar(Matriz m, entero e): Matriz* → Realiza el

producto de una matriz por un escalar de tipo real, dando como resultado otra matriz cuyos elementos están todos multiplicados por dicho escalar.

El pseudocódigo para la *clase Matriz* es el siguiente:

```

Clase Matriz
  privado Objeto mat[ ][ ]
  publico entero fil, col, maxFil, maxCol
  publico:
    Matriz(entero f, entero c)
      fil = f
      col = c
      maxFil = f + 10
      maxCol = c + 10
      mat = nuevo Objeto[f][c]
      entero i, j
      para (i = 0, i < f, i = i + 1)
        para (j = 0, j < c, j = j + 1)
          mat[i][j] = 0
        fin_para
      fin_para
    fin_metodo
  //-----
  Matriz (Objeto m[ ][ ])
    mat = m
    fil = m.fil
    col = m.col
    maxFil = m.maxFil
    maxCol = m.maxCol
  fin_metodo
  //-----

```





```

asignarDato(Objeto d, entero f, entero c)
    mat[f][c] = d
fin_metodo
//-----

Objeto obtenerDato(entero f, entero c)
    retornar mat[f][c]
fin_metodo
//-----

mostrarMatriz()
    cadena texto = " "
    entero i, j
    para ( i = 0, i < fil, i = i + 1)
        para ( j = 0, j < col, j = j + 1)
            texto = texto + mat[i][j].aCadena() + " "
        fin_para
        // salto de línea
    fin_para
    Flujo.imprimir(texto)
fin_metodo
//-----

entero [ ] buscar(Objeto x)
    entero pos[ ] = {-1, -1}
    entero i = 0, j
    logico hallado = falso // Supuesto: x no existe

    mientras ( i < fil && hallado == falso)
        j = 0
        mientras ( j < col && hallado == falso)
            si (mat[i][j].comparar(x) == 0) // ¿Se encontró x?
                pos[0] = i
                pos[1] = j
                hallado = cierto
            fin_si
            j = j + 1
        fin_mientras
        i = i + 1
    fin_mientras
    retornar pos
fin_metodo
//-----

```





```

Matriz sumaMatrices(Matriz a)
    Matriz b = nuevo Matriz (a.fil, a.col)
    entero i, j
    para ( i = 0, i < a.fil, i = i + 1)
        para (j = 0, j < b.col, j = j + 1)
            b.mat[i][j] = a.mat[i][j] + mat[i][j]
        fin_ para
    fin_ para
    retornar b
fin_metodo
//-----

```

```

Matriz productoMatrices(Matriz a)
    Matriz b = nulo
    b = nuevo Matriz (fil, a.col)
    entero i, j, k

    para (i = 0, i < fil, i = i + 1)
        para (j = 0, j < b.col, j = j + 1)
            b.mat[i][j] = 0
            para ( k = 0, k < col, k = k + 1)
                b.mat[i][j] = b.mat[i][j] + mat[i][k] * a.mat[k][j]
            fin_ para
        fin_ para
    fin_ para
    retornar b
fin_metodo
//-----

```

```

Matriz productoPorEscalar(entero e)
    Matriz r = nuevo Matriz (fil, col)
    entero i, j

    para (i = 0, i < fil, i = i + 1)
        para (j = 0, j < col, j = j + 1)
            r.mat[i][j] = mat[i][j] * e
        fin_ para
    fin_ para
    retornar r
fin_metodo
//-----
fin_clase

```

Observaciones:

- El método *mostrarMatriz()* utiliza la variable local *texto* de tipo *cadena* para crear una representación textual de la matriz a visualizar. Cada elemento de la matriz *mat* es convertido a texto mediante el método *aCadena()*, con el objetivo de concatenarlo a la variable *texto* a través el operador cruz (+).

La matriz es visualizada por filas, para lo cual se debe incluir un salto de línea al finalizar cada fila.

Por ejemplo, para mostrar la matriz *m* en pantalla se podría escribir:

```
caracter tabla[ ] [ ] = { {'a', 'b', 'c'}, {'1', '2', '3'}, {'@', '*', '&'}
Matriz m = nuevo Matriz(tabla)
tabla.mostrarMatriz( )
```

- En el producto de dos matrices, los elementos c_{ij} se obtienen multiplicando los elementos a_{ik} de la fila *i* por los elementos b_{kj} de la columna *j*, sumando los resultados *k* veces, de acuerdo a la siguiente fórmula:

$$c[i][j] = \sum_{K=0}^{a.col-1} a[i][k] * b[k][j]$$

PROBLEMA 17: PROCESO ELECTORAL

En las elecciones para alcalde de una ciudad se han presentado cuatro candidatos (A, B, C y D). La ciudad está dividida en cinco zonas de votación. El reporte de votos se recibe en orden según la zona: primero la zona 1, segundo la 2, y así sucesivamente hasta la zona 5. Calcular el total de votos obtenido por cada candidato, con su porcentaje correspondiente. Escriba un mensaje declarando ganador a un candidato, si éste obtuvo más del 50% de la votación.

Solución:

Pre análisis

Este problema plantea un proceso de elecciones simple donde el analista debe decidir sobre las estructuras de datos a utilizar.

El análisis del enunciado induce al trabajo con una matriz y un vector. En la matriz se pueden almacenar los votos obtenidos por cada candidato en las diferentes zonas, de manera que las filas correspondan a las zonas y las columnas a los candidatos, obteniendo así una matriz de orden 5×4 . En el vector se almacenará el total de votos por candidato, este se calcula a partir de la suma de la columna correspondiente, la cual forma un vector de tamaño igual a 4. Esta representación se ilustra en la figura 4.4., donde *votos* representa la matriz de votos y *vxc* el vector de votos por candidato.

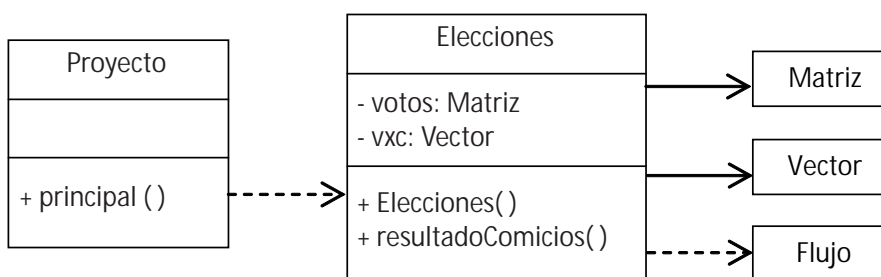
votos		A	B	C	D	Candidatos
		0	1	2	3	
1	0	15	34	70	32	
2	1	24	80	55	43	
3	2	70	81	34	23	
4	3	45	15	44	10	
5	4	23	7	89	60	
Zonas		vxc				
		177	217	292	168	
		0	1	2	3	

Figura 4.4. Matriz de votos y vector de votos por candidato

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Para cada zona, conocer la cantidad de votos por candidato	Veinte números enteros ingresados por el usuario	Los votos obtenidos por cada candidato en todas las zonas, es decir, la matriz inicializada con valores consistentes (<i>votos</i>)
R2	Conocer la cantidad de votos por candidato con su porcentaje correspondiente, así como identificar si hubo un ganador.	La matriz <i>votos</i> .	Los resultados de las elecciones para alcalde

b) Diagrama de clases



En este diagrama de clases se puede observar la diferencia entre una relación de *dependencia* (flecha con línea punteada) y una de *asociación* (flecha con línea continua). Entre las clases *Elecciones* y *Matriz* existe una *asociación* a través del atributo *votos*; de manera similar, entre las clases *Elecciones* y *Vector* existe una *asociación* a través del atributo *vxc*.

Entre las clases *Proyecto* y *Elecciones* y entre *Elecciones* y *Flujo* se presentan relaciones de dependencia (en este caso, de uso), porque el método *principal()* instancia un

objeto de la clase *Elecciones* y esta última utiliza la clase *Flujo* en operaciones de entrada y salida de datos.

c) Responsabilidades de las clases

Contratos de la clase *Elecciones*

Método	Req. asociado	Precondición	Postcondición
<i>Elecciones()</i>	R1	Se desconocen los votos obtenidos por los candidatos en cada una de las zonas	Los votos obtenidos por los candidatos en cada una de las zonas se conocen y están almacenados en la matriz <i>votos</i>
<i>resultadoComicios()</i>	R2	Se desconoce el resultado de los comicios.	Se conoce la cantidad de votos por cada candidato con su porcentaje respectivo y se sabe si existe o no un ganador

Contrato de la clase *Proyecto*

Método	Reque_rimiento asociado	Precondición	Postcondición
<i>principal()</i>	R1 R2	No se ha realizado el proceso de votación	Las elecciones se han realizado, se conocen los votos por candidato y si existe o no un ganador

d) Seudo código

```

clase Elecciones
  privado:
    Matriz votos
    Vector vxc
  publico:
    Elecciones()
    entero f, c
    Flujo.imprimir("Elecciones para alcalde")
    para (f = 0, f < 5, f = f + 1)
      Flujo.imprimir("Zona N° " + (f + 1))
      para (c = 0, c < 4, c = c + 1)
        segun (c)
          caso 0:
            Flujo.imprimir("Candidato A")
            saltar
          caso 1:
            Flujo.imprimir("Candidato B")
            saltar
          caso 2:
            Flujo.imprimir("Candidato C")
            saltar
          caso 3:
            Flujo.imprimir("Candidato D")
        fin_segun
      votos.asignarDato(Flujo.leerEntero( ), f, c)
    fin_para
  fin_para
fin_metodo
//-----
resultadoComicios( )
1. entero f, c
2. para (c = 0, c < 4, c = c + 1)
3.   vxc.asignarDato(0, c)
4.   para (f = 0, f < 5, f = f + 1)
5.     vxc.asignarDato(vxc.obtenerDato(c) +
                       votos.obtenerDato(f, c), c)
6.   fin_para
7. fin_para
8. entero totalV = vxc.obtenerDato(0) + vxc.obtenerDato(1) +
                  vxc.obtenerDato(2) + vxc.obtenerDato(3)
9. real pa, pb, pc, pd //Porcentajes de votos por candidato

```





```

10. pa = (vcx.obtenerDato(0) / totalV) * 100
11. pb = (vcx.obtenerDato(1) / totalV) * 100
12. pc = (vcx.obtenerDato(2) / totalV) * 100
13. pd = (vcx.obtenerDato(3) / totalV) * 100
14. Flujo.imprimir("Total votos candidato A: " +
    vxc.obtenerDato(0) + ", con un porcentaje de: " + pa + "%")
15. Flujo.imprimir("Total votos candidato B: " +
    vxc.obtenerDato(1) + ", con un porcentaje de: " + pb + "%")
16. Flujo.imprimir("Total votos candidato C: " +
    vxc.obtenerDato(2) + ", con un porcentaje de: " + pc + "%")
17. Flujo.imprimir("Total votos candidato D: " +
    vxc.obtenerDato(3) + ", con un porcentaje de: " + pd + "%")
18. si (pa>50) // vxc.obtenerDato(0) > totalV / 2
    Flujo.imprimir("Candidato ganador. " + A)
    sino
19.    si (pb>50)
        Flujo.imprimir("Candidato ganador. " + B)
    sino
20.    si (pc>50)
        Flujo.imprimir("Candidato ganador. " + C)
    sino
21.    si (pd>50)
        Flujo.imprimir("Candidato ganador. " + A)
    sino
        Flujo.imprimir("No hay ganador")
22.        fin_si
23.    fin_si
24. fin_si
25. fin_si
    fin_metodo
//-----
fin_clase // Elecciones

//*****
class Proyecto
    publico principal()
        Elecciones e = nuevo Elecciones()
        e.resultadoComicios()
    fin_metodo
fin_clase

```

Observaciones:

- El constructor sin argumentos *Elecciones()* es el encargado de llenar la matriz *votos*, miembro privado de la clase *Elecciones*. Este constructor recorre la matriz de votos por filas mediante dos ciclos **para** unirlos. El ciclo externo imprime un mensaje relativo a la zona; el ciclo interno utiliza un selector múltiple para informar acerca del candidato en proceso.

La instrucción *votos.asignarDato(Flujo.leerEntero(), f, c)* ofrece un particular interés en tanto asigna al miembro privado *mat* de la matriz *votos* en su posición $[f, c]$, y fija el dato digitado por el usuario, obtenido a través de la ejecución de la sentencia *Flujo.leerEntero()*.

- El cuerpo del método *resultadoComicios()* se encuentra numerado para facilitar las siguientes explicaciones:
 - La instrucción 3 asigna 0 (cero) a la posición *c* del vector *vxc*; la instrucción 5, que consume dos líneas de pseudo código, asigna al vector *vxc* en la posición *c*, el contenido de su propia posición más el contenido de la matriz *votos* en la posición $[f, c]$; la instrucción 8 asigna a la variable *totalV* la suma de los votos obtenidos por todos los candidatos.
 - Las instrucciones 10 a 13 asignan a las variables *pa*, *pb*, *pc* y *pd* los porcentajes de votación obtenidos por los candidatos A, B, C y D, respectivamente; las instrucciones 14 a 17 imprimen la cantidad de votos por candidato y su respectivo porcentaje, mientras que las instrucciones 18 a 25 contienen una serie de sentencias **si** anidadas, que permiten conocer quién fue el candidato ganador de las elecciones o si no hubo ganador.

4.4. EJERCICIOS PROPUESTOS

1. En el problema 16 se presentó la *búsqueda binaria* recursiva sobre un vector ordenado. Sobrecargar el método de la búsqueda binaria de tal manera que se elimine la recursividad (búsqueda binaria no recursiva) y desarrollar el pseudocódigo de los demás métodos de la clase VectorOrdenado, definida así:

VectorOrdenado
- vec[]: Objeto + tam: entero + n: entero
+ VectorOrdenado(entero x) + llenarVector(entero x) + mostrarVector() + busquedaBinaria(Objeto x, entero b, entero a): entero + busquedaBinaria(Objeto x): entero + eliminarDato(Objeto x): logico + insertarDato(Objeto x): logico

El constructor asigna x al tamaño del vector, pone n en cero e inicializa el vector *vec* con nulos. El método llenarVector(**entero** x) asigna el valor de x al atributo n y llena el vector con x elementos. Los métodos eliminarDato() e insertar dato() retornan **cierto** o **falso**, dependiendo del éxito o fracaso de la operación. Recordar que todas las operaciones deben conservar el ordenamiento del vector.

2. Escriba un método que acepte como parámetros dos vectores ordenados y que retorne un tercer vector ordenado que contenga los elementos de los vectores iniciales.
3. Se tiene una matriz cuadrada de orden n . Formar un vector con los elementos de la matriz triangular inferior, ordenar el vector creado e imprimirlo.
4. Redefina la clase *Vector* considerando que el vector puede contener elementos repetidos. Realice los supuestos semánticos necesarios. Por ejemplo, para eliminar

un dato del vector debe aclarar si quita la primera ocurrencia, la última o todas las ocurrencias que se presentan.

5. En un arreglo de enteros, presente algoritmos recursivos para calcular:
 - a) El elemento máximo del arreglo.
 - b) El elemento mínimo del arreglo.
 - c) La suma de los elementos del arreglo.
 - d) El producto de los elementos del arreglo.
 - e) El promedio de los elementos del arreglo.
6. Llenar dos conjuntos. Hallar los conjuntos unión, intersección, diferencia y producto cartesiano entre el primero y el segundo, respectivamente. Recuerde que un conjunto no admite elementos repetidos.
7. Una compañía distribuye N productos distintos. Para ello almacena en un arreglo toda la información relacionada con su mercancía: clave, descripción, existencia, mínimo a mantener de existencia y precio unitario.

Escriba métodos que permitan suplir los siguientes requerimientos:

- a). Venta de un producto: se deben actualizar los campos que correspondan, y comprobar que la nueva existencia no esté por debajo del mínimo (Datos: clave, cantidad vendida).
 - b). Reabastecimiento de un producto: se deben actualizar los datos que correspondan (Datos: clave, cantidad comprada).
 - c). Actualizar el precio de un producto aclarando si el precio aumenta o disminuye (Datos: clave, porcentaje).
 - d). Informar sobre un producto: se deben proporcionar todos los datos relacionados con un producto (Dato. Clave).
 - e). Listado de productos ordenado por precio unitario. El listado incluye clave y precio.
8. El dueño de una cadena de tiendas de artículos deportivos desea controlar sus ventas por medio de una computadora. Los datos de entrada son :

- El número de la tienda (1 a 10)
- Un numero que indica el deporte relacionado con el artículo (1 a 5)
- Costo del artículo.

Al final del día, visualizar el siguiente informe:

- Las ventas totales en el día para cada tienda
 - Las ventas totales de artículos relacionados con cada uno de los deportes.
 - Las ventas totales de todas las tiendas.
9. El departamento de policía de una ciudad ha acumulado información referente a las infracciones de los límites de velocidad durante un determinado periodo de tiempo. El departamento ha dividido la ciudad en cuatro cuadrantes y desea realizar una estadística de las infracciones a los límites de velocidad en cada uno de ellos. Para cada infracción se ha preparado una tarjeta que contiene la siguiente información:
- Número de registro del vehículo.
 - Cuadrante en el que se produjo la infracción.
 - Límite de velocidad en milla por hora (mph).
 - Velocidad registrada.
- a) Elabore métodos para obtener dos informes; el primero debe contener una lista con las multas de velocidad recolectadas, donde la multa se calcula como la suma del costo de la corte (\$20,000) mas \$ 1,250 por cada mph que exceda la velocidad límite. Prepare una tabla con los siguientes resultados:

INFRACCIONES A LOS LÍMITES DE VELOCIDAD

Registro del Vehículo-Velocidad Registrada (MPH)-Velocidad Límite-Multa

- b) El segundo informe debe proporcionar un análisis de las infracciones por cuadrante. Para cada uno de los 4 mencionados, debe darse el número de infracciones y la multa promedio.

10. Crear una matriz de orden $m * n$. Imprimir los elementos ubicados en:

- a) La matriz triangular inferior
- b) La matriz triangular superior
- c) Las diagonales principal y secundaria (letra X)

d) El primer y cuarto cuadrante.

1	2
3	4

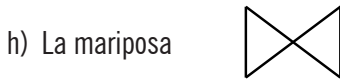
e) El segundo y tercer cuadrante.

1	2
3	4

- f) El reloj de arena



- g) El ajedrez (recuerde que no necesariamente es $8 * 8$ sino $m * n$)



- i) La Letra Z

- j) Las montañas



11. Multiplicar dos matrices $A(m, n)$ y $B(n, p)$, ambas de contenido entero. Validar que el número de columnas A sea igual a la cantidad de filas de b, para garantizar el producto.

12. Leer un número en arábigo y mostrarlo en letras.

Ejemplo: 1795 se lee como MIL SETECIENTOS NOVENTA Y CINCO

El programa debe manejar números en un rango de 0 a 99999999

13. Se tienen dos matrices del mismo tipo, $A(m, n)$ y $B(f, c)$. Crear un vector $C(m * n + f * c)$, que contenga los elementos de las matrices A y B recorridas por filas.

4.5 REFERENCIAS

[Aho1998]: Aho-Hopcroft-Ullman. Estructuras de Datos y Algoritmos. Addison-Wesley Iberoamericana, Wilmington, Delaware, E.U.A., 1998.

[Brassard1997]: Brassard-Bratley. Fundamentos de Algoritmia. Prentice Hall, Madrid, 1997.

[Cairó1993]: Cairó-Guardati. Estructuras de Datos. McGraw-Hill, México, 1993.

[Flórez2005]: Flórez Rueda, Roberto. Algoritmos, estructuras de datos y programación orientada a objetos. Ecoe Ediciones, Bogotá, 2005.

[Vásquez2007]: Vázquez, Juan Carlos. Ese inolvidable cafecito. La Coruña, España, <http://www.servicioskoinonia.org/cuentoscortos/articulo.php?num=031>. Consultado en noviembre de 2007.

[Weiss1995]: Weiss, Mark Allen. Estructuras de Datos y Algoritmos. Addison-Wesley Iberoamericana, Wilmington, Delaware, E.U.A., 1995.

CAPÍTULO 5

Relaciones entre clases

5.1. Tipos de relación entre clases

Asociación

Dependencia

Generalización / Especialización

Agregación y composición

Realización

Problema 18: Suma de dos números

5.2. Paquetes

Problema 19: Venta de productos

5.3. Ejercicios propuestos

5.4 Referencias

SUEÑO DE LA MARIPOSA

Chuang Tzu

Chuang Tzu soñó que era una mariposa. Al despertar ignoraba si era Tzu que había soñado que era una mariposa o si era una mariposa y estaba soñando que era Tzu.



Diagrama de clases:

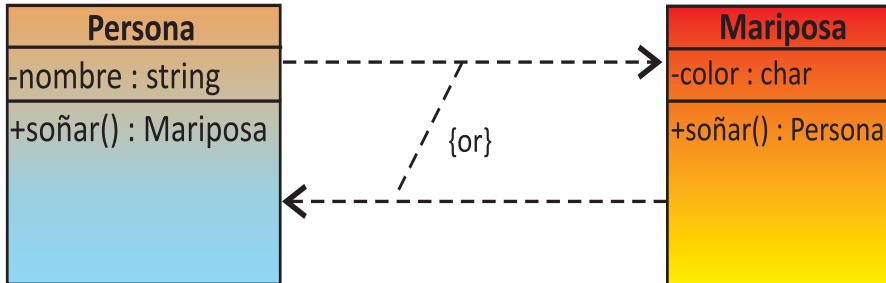
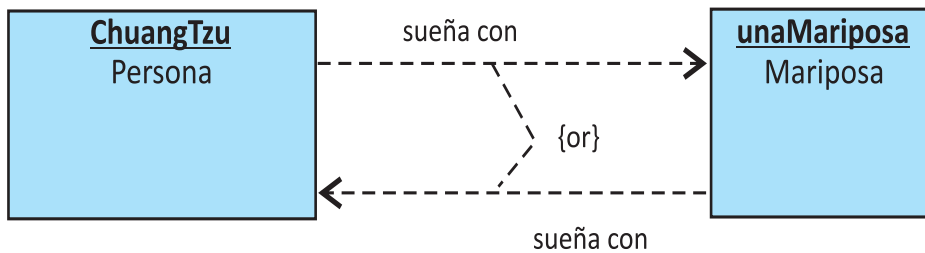


Diagrama de objetos:



Objetivos de aprendizaje

- Establecer relaciones lógicas entre las clases identificadas para la solución de un problema.
- Identificar los diferentes tipos de relación en un diagrama de clases.
- Crear objetos en un proyecto y reconocer la interrelación entre ellos.
- Crear clases a partir de otras ya existentes.
- Empaquetar clases con una funcionalidad común para efectos de reutilización.

5.1. TIPOS DE RELACIÓN ENTRE CLASES

En el Unified Modeling Language (UML) los elementos se unen entre sí a la vez que establecen una forma de interacción entre dichos elementos, incluyendo *casos de uso*, clases, estados, objetos, paquetes y otras representaciones propias del lenguaje de modelado. En este libro nos ocuparemos de las relaciones entre clases.

En un diagrama de clases, las relaciones que unen las clases entre sí tienen dos funciones: vincularlas entre sí y establecer las formas de su interacción. Se clasifican en cinco tipos:

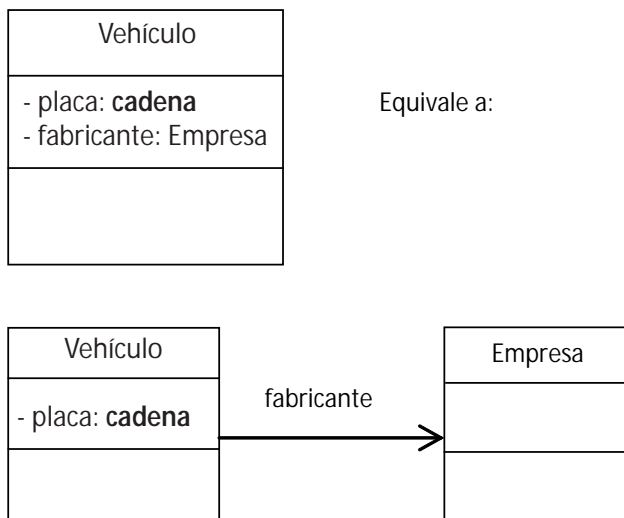
- Asociación: conexión entre clases.
- Dependencia: relación de uso.
- Generalización/Especialización: relación de *herencia* o superior-subordinado.
- Agregación y composición: relaciones todo-parte.
- Realización: relación semántica entre clases.

Veámoslas en detalle:

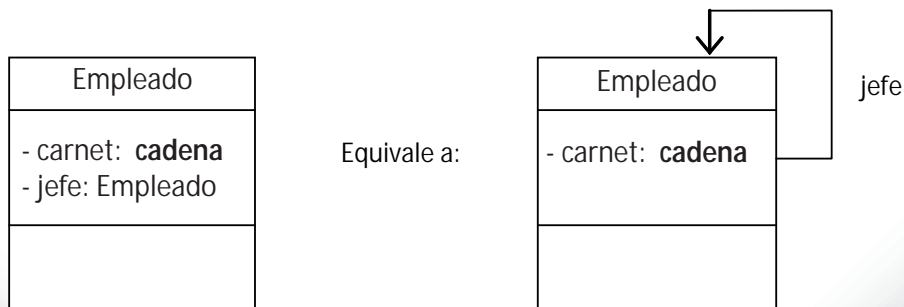
ASOCIACIÓN

Una *asociación* es una relación estructural que describe un conjunto de enlaces, los cuales son conexiones entre objetos. Se representa con una línea continua que puede incluir puntas de flecha que indican la navegabilidad (dirección o sentido) de la asociación. Si no existen flechas, la asociación se considera bidireccional. Algunos casos son:

– Asociación entre Vehículo y Empresa



– Asociación recursiva



La *multiplicidad o cardinalidad de una asociación* hace referencia a la cantidad de objetos inmiscuidos en la asociación. La multiplicidad puede ser de uno a uno (1-1), uno a muchos (1...*), cero o uno (0, 1) u otra específica que depende de la naturaleza de la asociación, como 2..24, ó 1..*. Un resumen de la multiplicidad puede observarse en la siguiente tabla.

Tabla 5.1. Multiplicidades de una asociación

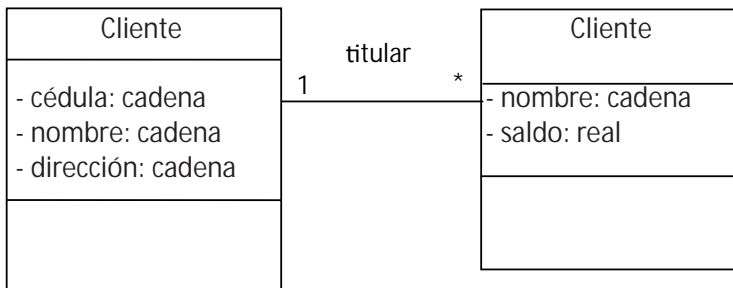
Multiplicidad	Significado
1	Uno y sólo uno
0,1	Cero o uno
0.. *	Cero o muchos
*	Cero o muchos
1..*	Uno o muchos
n..m	Desde n hasta m, donde n y m son números enteros
m	Exactamente m, donde m es un número entero

Observaciones a la multiplicidad:

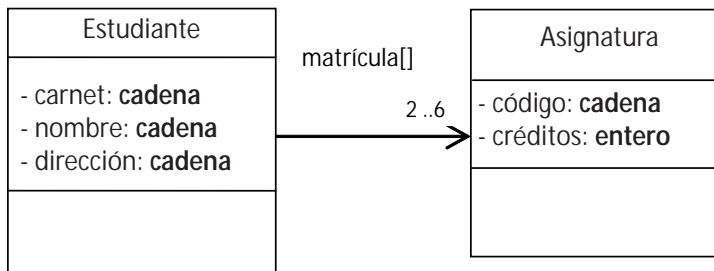
- Para indicar la multiplicidad de una asociación, se deben especificar las multiplicidades mínima y máxima.
- Cada asociación tiene dos multiplicidades, una a cada extremo de la línea. Si se omite información sobre la multiplicidad, se asume igual a 1 (uno).
- Cuando la multiplicidad mínima es cero, la asociación es opcional.
- Multiplicidades específicas de m..n podrían ser 1..10, 2..24 o 5..20.

Ejemplos:

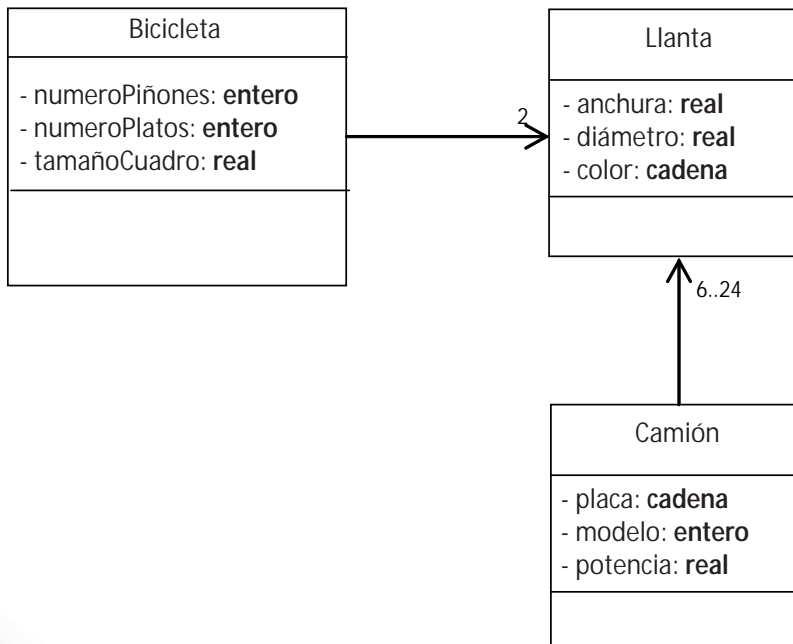
- Un cliente puede ser titular de varias cuentas, pero una cuenta de ellas tiene el titular único



- Un estudiante puede matricular desde dos a seis asignaturas:

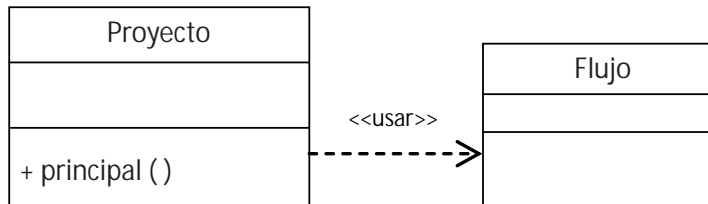


- Una bicicleta tiene exactamente 2 llantas y un camión de 6 a 24 llantas.



DEPENDENCIA

La *dependencia* es una relación semántica entre dos elementos, en la cual un cambio a un elemento (el elemento independiente) puede afectar la semántica del otro elemento (el dependiente). Esta relación se representa con un segmento de flecha punteado, donde el segmento inicia en la clase dependiente y finaliza con la flecha que señala al elemento independiente. Por ejemplo, en el siguiente ejemplo donde la clase *Proyecto* depende de la clase *Flujo*.



GENERALIZACIÓN / ESPECIALIZACIÓN

En una relación de *generalización/especialización* los objetos del elemento especializado (el hijo) pueden sustituir a los objetos del elemento general (el padre); de esta forma el hijo comparte la estructura y el comportamiento del padre. Una relación de especialización / generalización se conoce también como *herencia* y será tratada en el capítulo 6.

La relación de herencia se representa con una flecha de punta triangular hueca que señala hacia la clase padre, denominada también *clase base* o *superclase*. La clase especializada recibe el nombre de *clase derivada* o *subclase*. Por ejemplo, la clase *Rinoceronte* es una subclase de la clase *Animal*, como se observa en la figura 5.1.

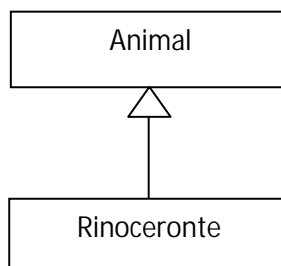


Figura 5.1. Rinoceronte es subclase de Animal

Una *clase abstracta* no puede ser instanciada pues posee métodos abstractos (aún no han sido definidos, es decir, sin implementación). La única forma de utilizarla es a través de la definición de subclases, que implementan los métodos abstractos declarados. Una clase es *final* si no se pueden definir más subclases a partir de ella. Por ejemplo, la clase *Persona* es una clase abstracta, y las clases *Eprimaria*, *ESecundaria* y *EUniversitario* son finales, según lo muestra la figura 5.2, donde se observa el uso de estereotipos en un diagrama de clases.

Un *estereotipo* es una estructura flexible que se puede utilizar de varios modos. Así, un estereotipo o clisé es una extensión del vocabulario de UML que permite crear nuevos bloques de construcción desde otros ya distintos pero específicos a un problema concreto [Booch1999]; se representa como un nombre entre dos pares de paréntesis angulares. Se puede utilizar el estereotipo sobre el nombre de una clase para indicar algo respecto al papel de la clase [Schmuller2000].

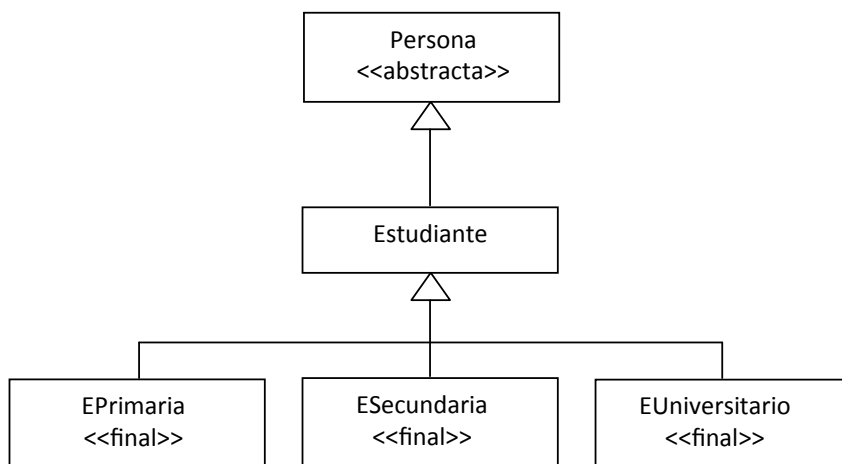


Figura 5.2. estereotipos <<abstracta>> y <<final>>

En el capítulo 6, los estereotipos y las clases finales se volverán a tratar con el estudio de los mecanismos de herencia.

AGREGACIÓN Y COMPOSICIÓN

Cuando se requiere estructurar objetos que son instancias de clases ya definidas por el analista, se presentan dos posibilidades: *composición* y *agregación*.

La *composición* es un tipo de relación estática, en donde el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye. En este tipo de relación el objeto base se construye a partir del objeto incluido, presentándose así una relación “todo/parte”.

La composición se representa con un rombo relleno que señala hacia el “todo”, como se ilustra en la figura 5.3:

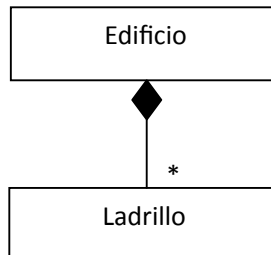


Figura 5.3. Composición

Nota: en la figura 5.3., (*) significa cero o muchos

La *agregación* es un tipo de relación dinámica, donde el tiempo de vida del objeto incluido es independiente del que lo incluye. En este tipo de relación, el objeto base se ayuda del incluido para su funcionamiento. La agregación se representa con un rombo sin relleno que señala hacia el “todo”.

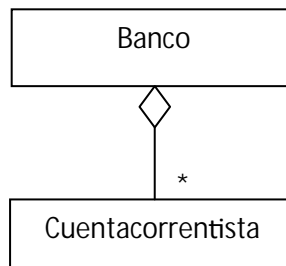


Figura 5.4. Agregación

Se pueden presentar relaciones de *composición* y *agregación* de manera simultánea, como en el siguiente caso:

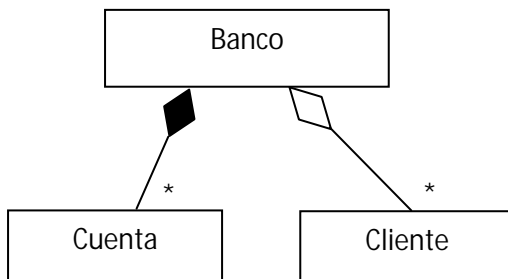
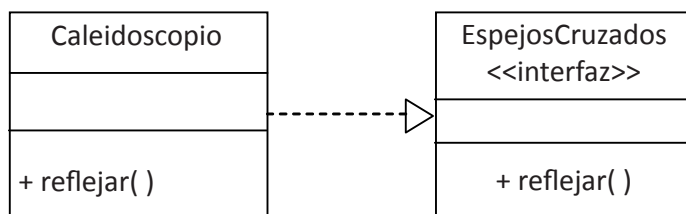


Figura 5.5. Combinación de composición y agregación

REALIZACIÓN

La *realización* es un vínculo semántico entre clases, en donde una clase específica tiene un contrato que otra clase garantiza que cumplirá (clasificación). Se pueden encontrar relaciones de realización entre interfaces y las clases o componentes que las realizan.

Semánticamente, la realización es una mezcla entre dependencia y *generalización*. La realización se representa como la generalización, con la punta de la flecha señalando hacia la interfaz. Por ejemplo, la clase *Caleidoscopio* realiza interfaz *EspejosCruzados*.



A continuación, en el problema 18, pueden observarse dos relaciones de dependencia, debido a que el funcionamiento del método principal() depende de las clases **Flujo** y **Suma2**.

PROBLEMA 18: SUMA DE DOS NÚMEROS

Sumar dos números enteros. Visualizar el resultado.

Solución:

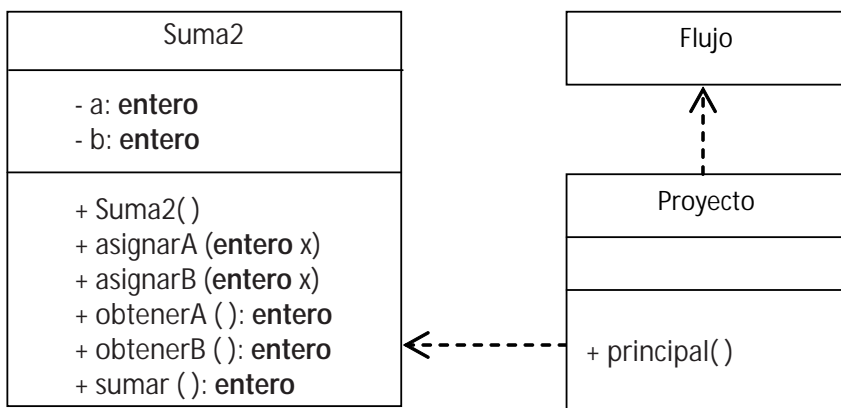
Aunque este problema es de extrema sencillez, puede resolverse dentro del método *principal()* con el uso de dos variables locales, a saber: una operación de suma y la clase **Flujo**. Aquí se plantea otra alternativa que busca aplicar una relación de uso (dependencia) entre dos clases, veamos:

a) Tabla de requerimientos

La tabla de requerimientos es idéntica a la ya presentada en el Capítulo 1 de este libro.

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Capturar dos números enteros	Dos números ingresados por el usuario	Dos números enteros almacenados en memoria (variables n1 y n2)
R2	Operar dos números enteros	Las variables n1 y n2	La suma de los números n1 y n2

b) Diagrama de clases



La clase *Suma2* consta de dos atributos (los números a sumar), un constructor, cuatro métodos modificadores y un método analizador.

c) Responsabilidades de las clases

Contratos de la clase *Suma2*

Nombre del método	Requerimientos asociados	Precondición	Postcondición	Modelo verbal
Suma2()	R1	No existen los números a sumar	Los números a sumar existen en un objeto de tipo Sumar2	No aplica
asignarA (entero <i>x</i>)	R1	$a == 0$	El atributo <i>a</i> ha cambiado su estado	Asignar el parámetro <i>x</i> al atributo <i>a</i> (modificar el estado del atributo)
asignarB (entero <i>x</i>)	R1	$b == 0$	El atributo <i>b</i> ha cambiado su estado	Asignar el parámetro <i>x</i> al atributo <i>b</i>
obtenerA(): entero	R2	Se desconoce el valor del atributo <i>a</i> (encapsulado)	Se conoce el valor del atributo <i>a</i>	Recuperar u obtener el estado actual del atributo <i>a</i>
obtenerB(): entero	R2	Se desconoce el valor del atributo <i>b</i> (encapsulado)	Se conoce el valor del atributo <i>b</i>	Recuperar u obtener el estado actual del atributo <i>b</i>
sumar(): entero	R2	No se han sumado los números	Se conoce la suma de los atributos <i>a</i> y <i>b</i>	1. Sumar los valores actuales de los atributos <i>a</i> y <i>b</i> . 2. Retornar la suma

Contrato de la clase Proyecto

Nombre del método	Requerimientos asociados	Precondición	Postcondición	Modelo verbal
principal ()	R1 y R2	Se desconocen los números a sumar	Se ha calculado y visualizado la suma de dos números enteros	<ol style="list-style-type: none"> 1. Instanciar un objeto de la clase Suma2 2. Asignar valores a los atributos a y b del objeto creado 3. Enviar el mensaje sumar() al objeto creado para calcular la suma 4. Visualizar el resultado de la suma

d) Seudo código:

```

clase Suma2
  privado:
    entero a
    entero b
  público:
    //----- métodos para modificar el estado del objeto -----
    asignarA (entero x)
      a = x
    fin_método

    asignarB (entero x)
      b = x
    fin_método
    //----- métodos obtener el estado del objeto -----
    entero obtenerA ( )
      retornar a
    fin_método

    entero obtenerB ( )
      retornar b
    fin_método
    //----- constructores -----
    Suma2 ( )
    fin_método

```





```

Suma2 (entero a1, entero b1)
    asignarA (a1)
    asignarB (b1)

fin_método
//----- función -----
entero sumar ( )
    retornar a + b
fin_método
fin_clase

```

En la clase *Proyecto* se crea un objeto de la clase *Suma2*, tal como se especifica en el contrato de dicha clase:

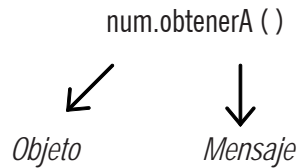
```

clase Proyecto
    estático principal ( )
        Suma2 num = nuevo Suma2 ( )
        num.asignarA(7)
        num.asignarB(8)
        Flujo.imprimir (num.obtenerA ( ) + " + " + num.obtenerB ( ) +
                        " = " + num.sumar ( ))
    fin_método
fin_clase

```

Observaciones:

- La definición dada para la clase *Suma2* (métodos asignar, obtener, constructores y función), le permite al analista-programador una serie de posibilidades para plantear la solución del problema. Ya se ha presentado una primera alternativa: usar el constructor por defecto para crear el objeto *num*, asignarle dos valores a sus miembros dato (7 y 8, valores arbitrarios escogidos a criterio del analista) y mostrar el resultado con tres mensajes al objeto *num*: *obtenerA()*, *obtenerB()* y *sumar()*.



Veamos otras alternativas para sumar dos números enteros, redefiniendo el método `principal()`, único método que no admite sobrecarga.

Alternativa 1: uso del constructor con argumentos.

estático `principal()`

Suma2 num = nuevo Suma2(15, 24)

Flujo.imprimir (num.obtenerA() + " + " + num.obtenerB() + " = " +
num.sumar())

fin_método

Alternativa 2: aquí se prescinde del método `sumar()`, es decir, se imprime el resultado de una expresión aritmética.

estático `principal()`

Suma2 num = nuevo Suma2(73, -55)

Flujo.imprimir (num.obtenerA() + " + ", num.obtenerB() + " = " +
num.obtenerA() + num.obtenerA())

fin_método

Alternativa 3: con entrada de datos especificada por el usuario.

estático `principal()`

Suma2 num = nuevo Suma2()

Flujo.imprimir("Ingrese dos números enteros:")

num.asignarA(Flujo.leerEntero())

num.asignarB (Flujo.leerEntero())

Flujo.imprimir (num.obtenerA() + " + " + num.obtenerB() + " = " +
num.sumar())

fin_método

- El desempeño de los métodos `asignarA ()`, `asignarB ()` y `sumar ()` se puede mejorar con el uso de la clase de uso común `Entero` o con el *manejo de excepciones*.

En el primer caso, se utilizan las constantes figurativas `MAX_VALOR` y `MIN_VALOR` que representan el máximo y mínimo valor del tipo de dato `entero`.

```

asignarA (entero a1)
    si (a1 > Entero.MAX_VALOR || a1 < Entero.MIN_VALOR)
        a = 0
    sino
        a = a1
    fin_si
fin_método

```

En el segundo caso, se puede aplicar el manejo de excepciones para controlar la instrucción de asignación:

```

asignarA (entero a1)
    probar
        a = a1
    fin_probar
    capturar (Excepción e)
        Flujo.imprimir ("Error" + e.mensaje ( ))
        a = 0
    fin_capturar
fin_método

```

De igual forma, se puede reescribir el método `asignarB ()`.

El método `sumar()` puede tomar el siguiente cuerpo:

```
entero sumar ( )
    si ((a + b) > Entero.MAX_VALOR)
        Flujo.imprimir ("La suma ha sobrepasado al máximo
                        entero")
    retornar 0
sino
    retornar a + b
fin_método
```

5.2. PAQUETES

Los *paquetes* ofrecen un mecanismo general para la organización de las clases que tienen responsabilidades relacionadas, y permiten agrupar modelos o subsistemas con funcionalidades comunes. En general un paquete se representa como:

Un *paquete* puede contener otros paquetes sin límite de anidamiento, pero cada elemento pertenece a (o está definido en) un sólo paquete. Una clase de un paquete puede aparecer en otro paquete por medio de la importación a través de una relación de *dependencia* entre paquetes como en la siguiente figura:

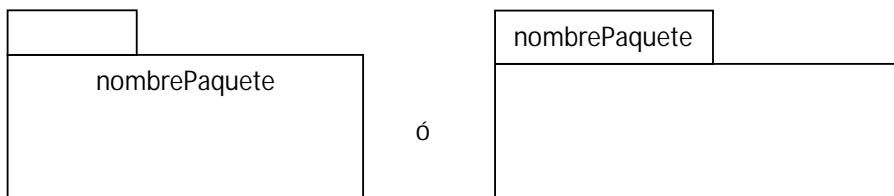


Figura 5.6. Representación de un paquete

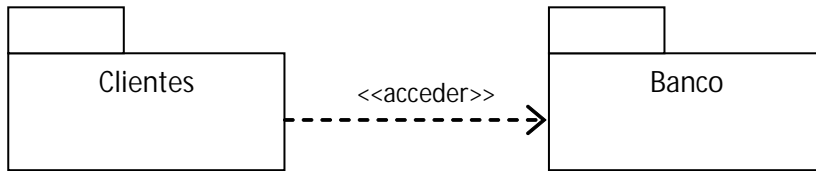


Figura 5.7. Dependencia entre paquetes

Además, una clase puede establecer relaciones de dependencia con otras clases del mismo paquete. (Ver Figura 5.8)

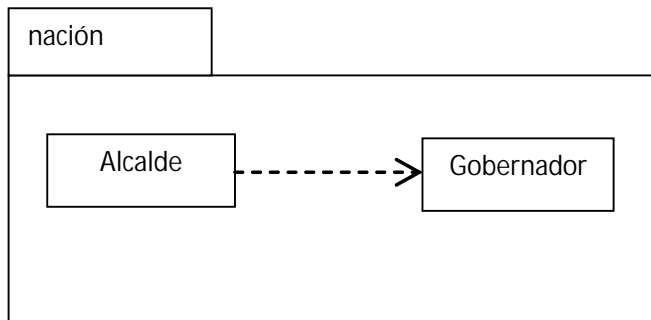


Figura 5.8. Relación de dependencia entre clases del mismo paquete

En la figura 5.9 las clases *clase1* y *clase2* forman parte del *espacio de nombres paquete*. La clase *Prueba* no forma parte del espacio de nombres e incluye al método *principal* que instancia por lo menos un objeto de las otras clases del paquete.

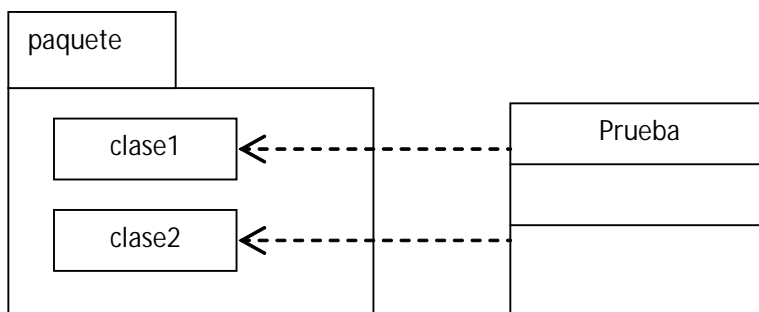


Figura 5.9. Dependencia entre una clase externa y otras del mismo paquete

Otro modelo posible consiste en un conjunto de clases con asociaciones de *composición* y *dependencia*, como ilustra el diagrama de clases de la figura 5.10, donde *Clase_B* está formada por uno o más objetos *Clase_A* y la *clase Proyecto* depende de la clase *Clase_B* y/o de la clase *Clase_A*.

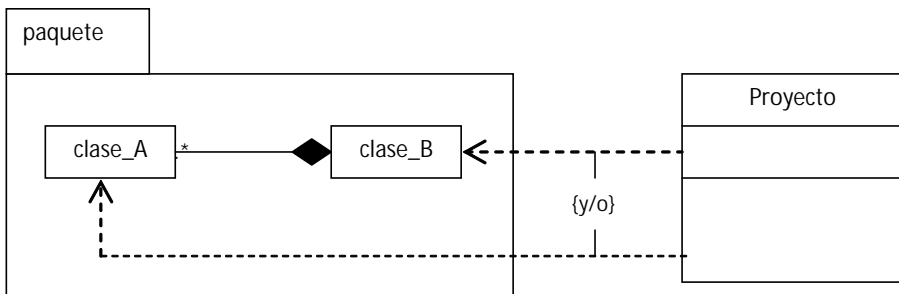


Figura 5.10. Composición, dependencia, paquete y restricción

Como se ha expuesto, la *clase Proyecto* contiene el método principal (). La *composición* establece una relación todo-parte entre clases; en la figura 5.10, *Clase_B* (el todo) está formada por una o más partes de objetos *Clase_A* (la parte).

A continuación, en el problema 19 se presenta un diagrama de clases con relaciones de dependencia y asociación, donde las clases *Producto* y *Venta* se agrupan en el paquete denominado *almacén*.

PROBLEMA 19: VENTA DE PRODUCTOS

Al producirse una venta en un determinado almacén, se toman los detalles del producto (su código y precio) y la cantidad vendida. Se pide: simular el proceso de dos ventas, imprimir el valor de cada una de ellas y el monto total por las dos transacciones.

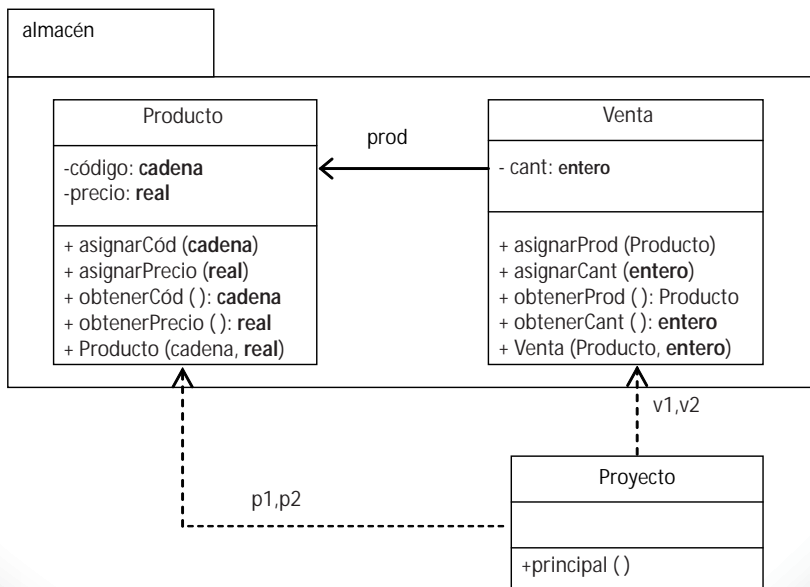
Nota: suponga que cada venta involucra un solo producto

Solución:

a) Tabla de requerimientos

Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Conocer los datos de las dos ventas: <ul style="list-style-type: none"> • Código del producto • Precio del producto • Cantidad vendida 	Los códigos y precios de dos productos aportados por el usuario	Los códigos y precios de los dos productos se encuentran almacenados en dos ventas. (objetos v1 y v2)
R2	Hallar el valor de cada venta.	Los objetos (o ventas) v1 y v2.	El valor de cada venta ()
R3	Hallar el valor total de las dos ventas	Las ventas v1 y v2	El valor total por las dos ventas

b) Diagrama de clase



c) Seudo código

```

paquete almacén
clase Producto
    privado:
        cadena código
        real precio
    público:
        asignarCód (cadena c)
            código = c
        fin_método
        //-----
        asignarPrecio (real p)
            precio = p
        fin_método
        //-----
        cadena obtenerCód ( )
            retornar código
        fin_método
        //-----
        real obtenerPrecio ( )
            retornar precio
        fin_método
        //-----
        Producto (cadena cod, real costo)
            asignarCód (cod)
            asignarPrecio (costo)
        fin_método
fin_clase
//-----
clase Venta
    privado:
        Producto prod
        entero cant
    público:
        asignarProducto(Producto p)
            prod = p
        fin_método
        //-----

```



```

↓
AsignarCant(entero c)
    cant = c
fin_método
//-----
Producto obtenerProducto()
    retornar prod
fin_método
//-----
entero obtenerCant()
    retornar cant
fin_método
//-----
Venta(Producto p, entero c )
    asignarProducto (p)
    asignarCant (c)
fin_método
fin_clase
fin_paquete

```

En otro archivo de código fuente se reutilizan las clases del paquete *almacén*:

```

importar almacén
clase Proyecto
    estático principal()
        cadena codP1, codP2
        real precP1, precP2
        entero cantVP1, cantVP2
        // Primera venta
        codP1 = Flujo.leerCadena ()
        precP1 = Flujo.leerReal ()
        cantVP1 = Flujo.leerEntero ()
        Producto p1 = nuevo Producto (codP1, precP1 )
        Venta v1 = nuevo Venta (p1, cantVP1 )
        // Segunda venta
        codP2 = Flujo.leerCadena ()
        precP2 = Flujo.leerReal ()
        cantVP2 = Flujo.leerEntero ()
        Producto p2 = nuevo Producto (codP2, precP2 )
        Venta v2 = nuevo Venta (p2, cantVP2 )
↓

```


↓

```

real totV1, totV2, total
totV1 = p1.obtenerPrecio () * v1.obtenerCant ()
totV2 = p2.obtenerPrecio () * v2.obtenerCant ()
total = totV1 + totV2
Flujo.imprimir (totV1, totV2, total)
    fin_método
fin_clase

```

Observaciones:

- Los constructores de las clases se han omitido porque están disponibles por omisión.
- La clase **Flujo** pertenece al paquete *sistema* (*espacio de nombres* importado por defecto). La lectura de cualquier tipo de dato incluye el manejo de *excepciones* (errores en el proceso de entrada de datos), como se observa en el siguiente pseudo código:

```

Flujo :: leerCadena ()
    Cadena c
    probar
        c = leer ()
    fin_probar

    capturar (Excepción)
        c = " "
    fin_capturar
    retornar c
fin_método

```

El carácter `::` es el *operador de resolución de ámbito* propio de C++ y no forma parte del pseudo lenguaje Sismoo; se usa aquí para indicar que “el método `leerCadena ()` pertenece a la clase **Flujo**”.

El método *leer()* posibilita la entrada de datos desde el dispositivo estándar (teclado).

Las palabras reservadas **probar** y **capturar** equivalen a las sentencias **try** y **catch** propias de algunos lenguajes de programación orientados a objetos.

De forma análoga se definen los demás métodos:

```
Flujo :: leerCarácter ( )
    Carácter c
    probar
        c = Carácter.convertirACarácter (leer ( ))
    fin_probar
    capturar (Excepción)
        c = Carácter.MIN_VALOR
    fin_capturar
    retornar c
fin_método
// -----
```

```
Flujo:: leerReal ( )
    Real r
    probar
        r = Real.convertirANum (leer ( ))
    fin_probar
    capturar (Excepción)
        r = Real.MIN_VALOR
    fin_capturar
    retornar r
fin_método
// -----
```

```
Flujo :: leerEntero( )
    Entero e
    probar
        e = Entero.convertirANum (leer ( ))
    fin_probar
```

```

    capturar (Excepción)
        e = entero. MIN_VALOR
    fin_capturar
    retornar e
fin_método

```

En los métodos para lectura de datos estándar se ha tratado con objetos de las clases *Cadena*, *Carácter*, *Real* y *Entero*, que encapsulan a sus respectivos tipos de datos.

El método *leer()* captura una cadena de la entrada estándar — teclado—, es decir, almacena un flujo de bits ingresado desde el dispositivo de entrada de consola hacia la memoria interna de la computadora.

5.3. EJERCICIOS PROPUESTOS

1. Escriba una aplicación que calcule el total de entradas vendidas para un concierto. Hay tres tipos de asiento: A, B y C. El programa acepta el número de entradas vendidas. El total de ventas se calcula de la siguiente forma:

$$\begin{aligned}
 \text{ventasTotales} = & \text{numeroAsientos_A} * \text{precioAsiento_A} + \\
 & \text{numeroAsientos_B} * \text{precioAsiento_B} + \\
 & \text{numeroAsientos_C} * \text{precioAsiento_C}
 \end{aligned}$$

Definir y usar la clase *TipoAsiento*: una instancia de la clase *TipoAsiento* conoce el precio para un tipo de asiento dado A, B o C.

2. Resolver un sistema de ecuaciones con tres incógnitas.
3. Un negocio de deportes ofrece un 10% de descuento en la compra de balones de fútbol y un 5% por paquete de tres pelotas de tenis. Se requiere escribir una

solución objetual que le permita a un empleado del negocio ingresar los precios originales de los balones y los paquetes de pelotas. La solución deberá usar esta entrada de datos para calcular el precio rebajado; y la salida deberá mostrar tanto el precio original como el precio con descuento.

4. Suponga que trabaja en un videoclub. El encargado quiere que le escriba un programa que calcule el recargo que tienen que pagar los clientes cuando se retrasan en la devolución de películas de acuerdo con las siguientes normas:
 - El alquiler de los videos cuesta \$1000 al día, pagados por el usuario en el almacén. El periodo de alquiler es de un día y el recargo por retraso es de \$100 al día, valor que debe abonarse al ser devuelta la película.
 - Cuando el cliente entregue la película, un empleado introducirá los siguientes datos: nombre del cliente, título de la película y número de días de retraso (que pueden ser cero). Se pide, entonces, que programa muestre la siguiente información: el nombre del cliente, el título de la película y el recargo por retraso.
5. Crear una clase para los vehículos de carga con los siguientes atributos, constructores y métodos:
 - Atributos privados *carga* y *cargaMáxima*.
 - Constructor público que aporta un valor inicial al atributo *cargaMáxima*.
 - Método público: *obtenerCarga()* que retorna el valor del atributo *carga*.
 - Método público: *obtenerCargaMaxima()* que retorna el valor del atributo *cargaMáxima*.
 - Método publico *agregarCaja()*, que toma el peso de una caja en kilos, y verifica que al agregar la caja al vehículo de este no sobrepase la carga máxima. Si una caja sobrepasa la carga máxima, es rechazada y se retorna el valor **falso**; en caso contrario, se agrega el peso de la caja a la carga del vehículo, y se retorna el valor **cierto**. Todos los datos están expresados en kilos.

Después de definir la clase, hacer las siguientes tareas: crear una cantidad desconocida de vehículos de carga, y visualizar el listado de vehículos ingresados, la carga promedio, la mínima y la máxima.
6. Plantear soluciones alternativas al problema 19. Sea creativo, usted es un analista potenciado y en potencia.

5.4 REFERENCIAS

[Booch1999]: Booch-Jacobson-Rumbaugh. El lenguaje Unificado de Modelado. Addison-Wesley Iberoamericana, Madrid, 1999.

[Schmuller2000]: Schmuller, Joseph. UML en 24 horas. Pearson Educación, México, 2000.

[Tzu2006]: Tzu, Chuang. Sueño de la mariposa. <http://www.ciudadseva.com/textos/cuentos/mini/suenyo.htm>. Consultado en junio de 2006.

CAPÍTULO 6

Mecanismos de herencia

- 6.1. Herencia
 - Herencia simple
 - Herencia múltiple: interfaces
 - Problema 20: Empleados por hora y a destajo
 - 6.2. Polimorfismo
 - 6.3. Ejercicios propuestos
 - 6.4. Referencias
-

FLORES DE LAS TINIEBLAS

Villers de L'Isle-Adam

A Léon Dierx
"Buenas gentes que pasáis, irogad por los difuntos!"
(Inscripción al borde del camino)

¡Oh, los bellos atardece-
res! Ante los brillantes
café de los bulevares, en
las terrazas de las horcha-
terías de moda,iqué de
mujeres con trajes multi-
colores, qué de elegantes
"callejeras" dándose tono!

Y he aquí las pequeñas
vendedoras de flores,
quienes circulan con sus
frágiles canastillas.

Las bellas desocupadas
aceptan esas flores pe-
recederas, sobrecojidas,
misteriosas...

-¿Misteriosas?

-¡Sí, si las hay!

Existe -sabadlo, sonrien-
tes lectoras-, existe en el
mismo París cierta agencia
que se entiende con varios

conductores en los entierros de lujo, incluso con enterradores, para despojar a los difuntos de la mañana, no dejando que se marchiten inútilmente en las sepulturas todos esos espléndidos ramos de flores, esas coronas, esas rosas que, por centenares, la piedad filial o conyugal coloca diariamente en los catafalcos.

Estas flores casi siempre quedan olvidadas después de las tenebrosas ceremonias. No se piensa más en ello; se tiene prisa por volver. ¡Se concibe!...

Es entonces cuando nuestros amables enterradores se muestran más alegres. ¡No olvidan las flores estos señores! No están en las nubes, son gente práctica. Las quitan a brazadas, en silencio. Arrojarlas apresuradamente por encima del muro, sobre un carretón propicio, es para ellos cosa de un instante.

Dos o tres de los más avisados y espabilados transportan la preciosa carga a unos floristas amigos, quienes, gracias a sus manos de hada, distribuyen de mil maneras, en ramitos de corpiño, de mano, en rosas aisladas inclusive, esos melancólicos despojos.

Llegan luego las pequeñas floristas nocturnas, cada una con su cestita. Pronto circulan incesantemente, a las primeras luces de los reverberos, por los bulevares, por las terrazas brillantes, por los mil y un sitios de placer.

Y jóvenes aburridos y deseosos de hacerse bienquitos por las elegantes, hacia las cuales sienten alguna inclinación, compran esas flores a elevados precios y las ofrecen a sus damas.

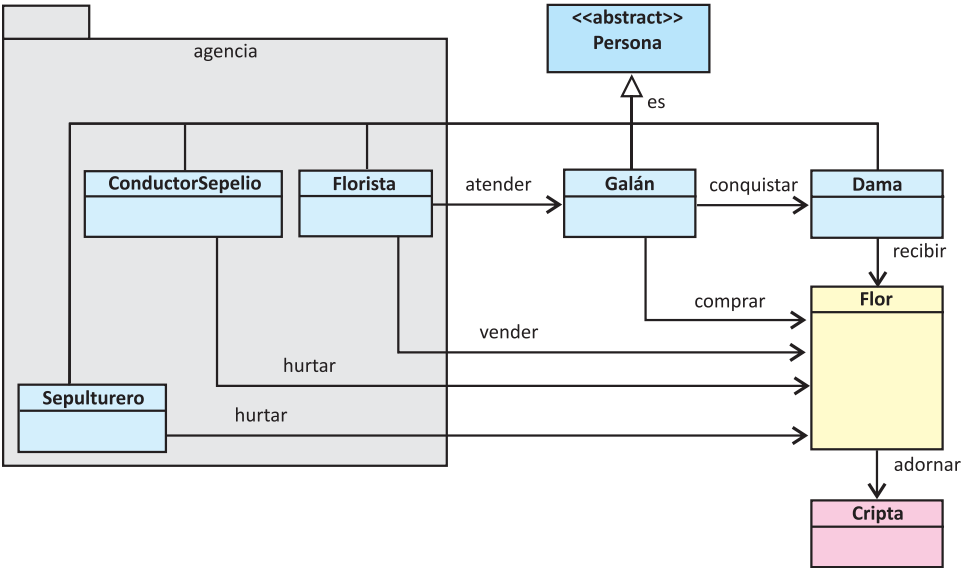
Éstas, todas con rostros empolvados, las aceptan con una sonrisa indiferente y las conservan en la mano, o bien las colocan en sus corpiños.

Y los reflejos del gas empalidecen los rostros.

De suerte que estas criaturas-espectros, adornadas así, con flores de la Muerte, llevan, sin saberlo, el emblema de amor que ellas dieron y del amor que reciben.



Diagrama de clases:



Objetivos de aprendizaje

- Identificar jerarquías de clase, y dentro de ellas los conceptos de clase base y clase derivada.
- Diferenciar entre clases abstracta, convencional y final.
- Implementar mecanismos de herencia simple y herencia múltiple, esta última mediante interfaces.
- Comprender el concepto de asociación entre clases, roles y cardinalidad (o multiplicidad).
- Aprender a empaquetar clases para casos específicos, en búsqueda de su reutilización para la solución de otros problemas.

6.1. HERENCIA

La *herencia* es una propiedad que permite crear objetos a partir de otros ya existentes. Con ella fácilmente pueden crearse clases derivadas (clase específica) a partir de una *clase base* (clase general); y deja reutilizar a la *clase derivada* los atributos y métodos de la clase base. La herencia conlleva varias ventajas:

- Permite a los programadores ahorrar líneas de código y tiempo en el desarrollo de aplicaciones.
- Los objetos pueden ser contruidos a partir de otros similares, heredando código y datos de la clase base.
- Las clases que heredan propiedades de otra clase pueden servir como clase base de otras, formando así una jerarquía de clases.

La herencia se clasifica en simple y múltiple

HERENCIA SIMPLE

La *herencia simple* le permite a un objeto extender las características de otro objeto y de ningún otro, es decir, solo puede heredar o tomar atributos de un solo padre o de una sola clase.

En el UML, la herencia simple se representa con una flecha de punta triangular. En la figura 6.1., el segmento de línea de la flecha inicia en la clase subordinada y la punta señala hacia la *clase base*.

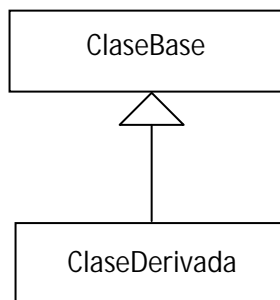


Figura 6.1. Herencia simple

Por ejemplo, en geometría, un rectángulo es un cuadrilátero, así como los cuadrados, los paralelogramos y los trapezoides. Por ende, se puede decir que la clase Rectángulo hereda de la clase Cuadrilátero. En este contexto, el Cuadrilátero es una *clase base* y Rectángulo es una *clase derivada*. [Deitel2007]

A partir de una clase base -o superclase- se pueden definir varias clases derivadas, como se observa en la figura 6.2, equivalente en semántica a la figura 6.3.

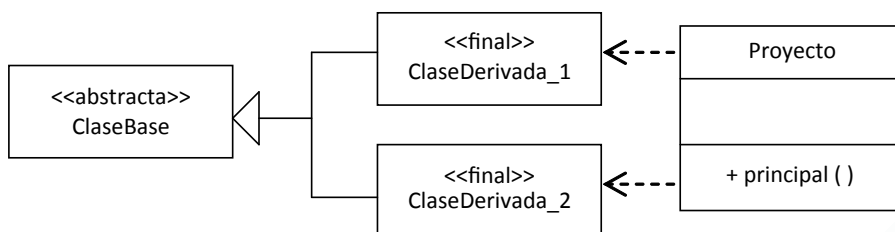


Figura 6.2. Dos clases derivadas de una misma clase base

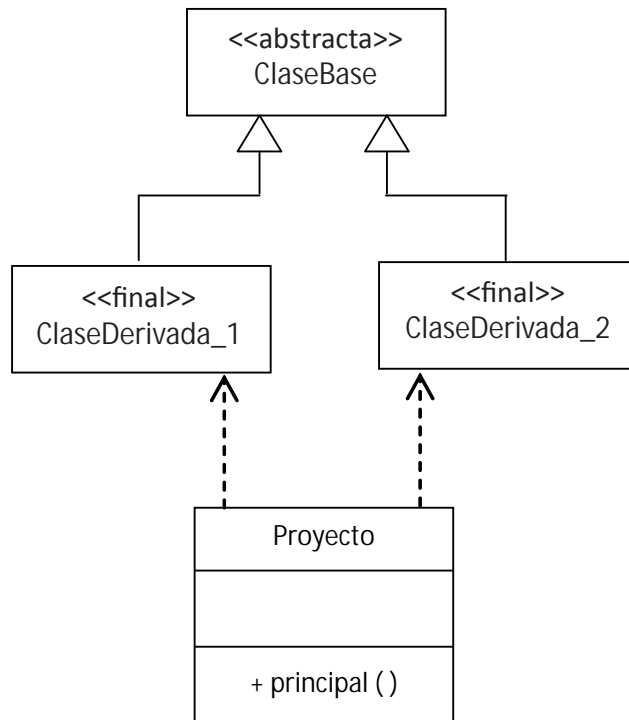


Figura 6.3. La clase *ClaseDerivada* realiza la interfaz *InterfazX*

La superclase *ClaseBase* es abstracta (obsérvese el uso del *estereotipo* `<<abstracta>>` propio de UML). No se pueden definir objetos a partir de una *clase abstracta*, es decir, una clase abstracta no puede ser instanciada. Las clases abstractas se usan como patrón para definir clases instanciables o concretas por medio de mecanismos de herencia.

Por su parte, una *clase final* (`<<final>>`) nunca puede ser abstracta. Cuando una clase es definida como final se establece, por razones de diseño o seguridad, que no se desea heredar de ella.

A su vez, las subclases `ClaseDerivada_1` y `ClaseDerivada_2` heredan el estado y comportamiento (atributos y métodos) de la superclase `ClaseBase`.

HERENCIA MÚLTIPLE: INTERFACES

Con la ayuda de la *herencia múltiple* un objeto puede extender las características de uno o más objetos, es decir, puede tener varios padres. Respecto a esta herencia, son varias las discrepancias entre los diseñadores de lenguajes de programación: algunos de ellos han preferido no admitir la herencia múltiple por las posibles coincidencias en nombres de métodos o datos miembros. Por ejemplo C++ y Python admiten herencia múltiple, mientras que Java, Ada y C# sólo permiten herencia simple, pero manejan la herencia múltiple por medio de *interfaces*, como se ilustra en la figura 6.4.

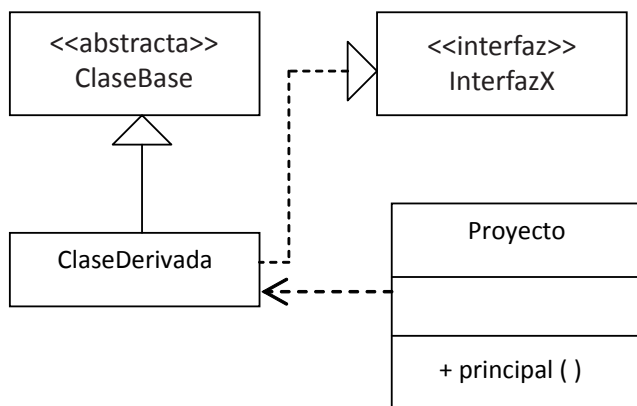


Figura 6.4. La clase *ClaseDerivada* realiza la interfaz *InterfazX*

La clase *ClaseDerivada* hereda de *ClaseBase*, realiza la interfaz *InterfazX* y es usada por la clase *Proyecto*.

Una clase puede realizar (o implementar) varias interfaces, como en el siguiente diagrama de clases de la figura 6.5, donde se presentn los *estereotipos* «abstracto», «final» e «interfaz».

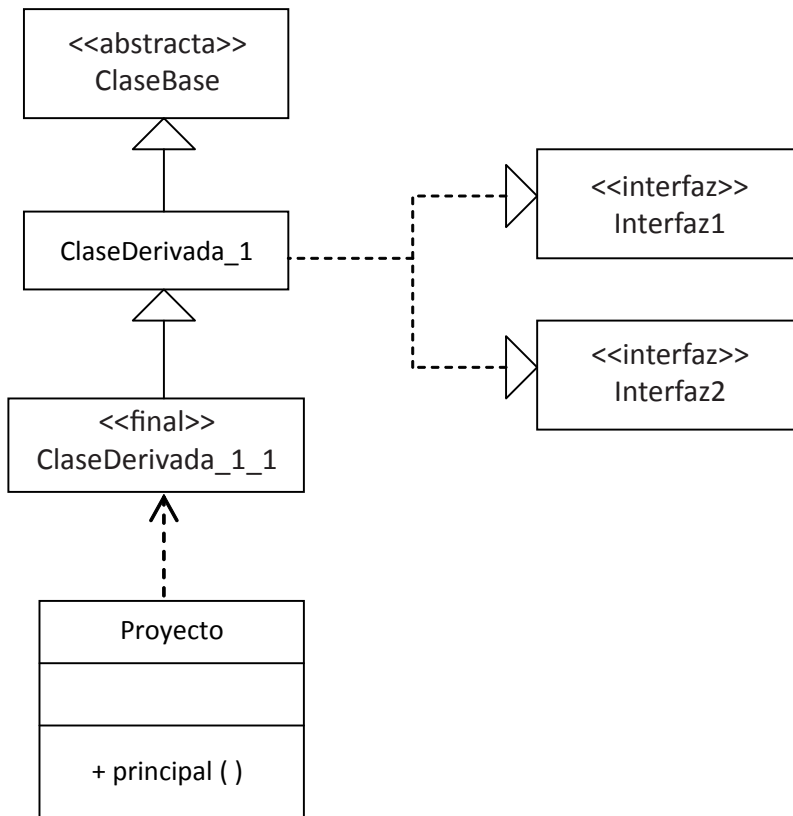


Figura 6.5. La clase *ClaseDerivada_1* realiza dos interfaces

Una interfaz permite al creador establecer la forma de una clase: nombres de métodos, listas de parámetros y tipos de retorno, pero no cuerpos de métodos. En algunos lenguajes de programación una interfaz puede contener campos (caso Java); en otros una interfaz no admite el uso de campos (caso C#).

Una interfaz proporciona sólo la forma, pero no la implementación, y se puede considerar como una clase que contiene un conjunto de operaciones que otras clases deben implementar o realizar.

PROBLEMA 20. EMPLEADOS POR HORA Y A DESTAJO

Una pequeña empresa hace dos tipos de empleados: los que trabajan por horas o por cantidad de trabajo realizado. Del primer tipo se conoce su identificación, número de horas trabajadas y costo de la hora; del segundo grupo se conoce su identificación y número de unidades producidas. Para éstos últimos el pago por unidad producida tiene un valor constante de \$2500.

Se pide: ingresar la información de m empleados por hora y n a destajo, donde m y n son variables enteras. Visualizar un informe con la identificación y salario devengado por cada empleado. Finalmente, mostrar el salario promedio de los empleados asalariados y a destajo.

Problema adaptado de [Deitel2007]

Solución:

a) Tabla de requerimientos

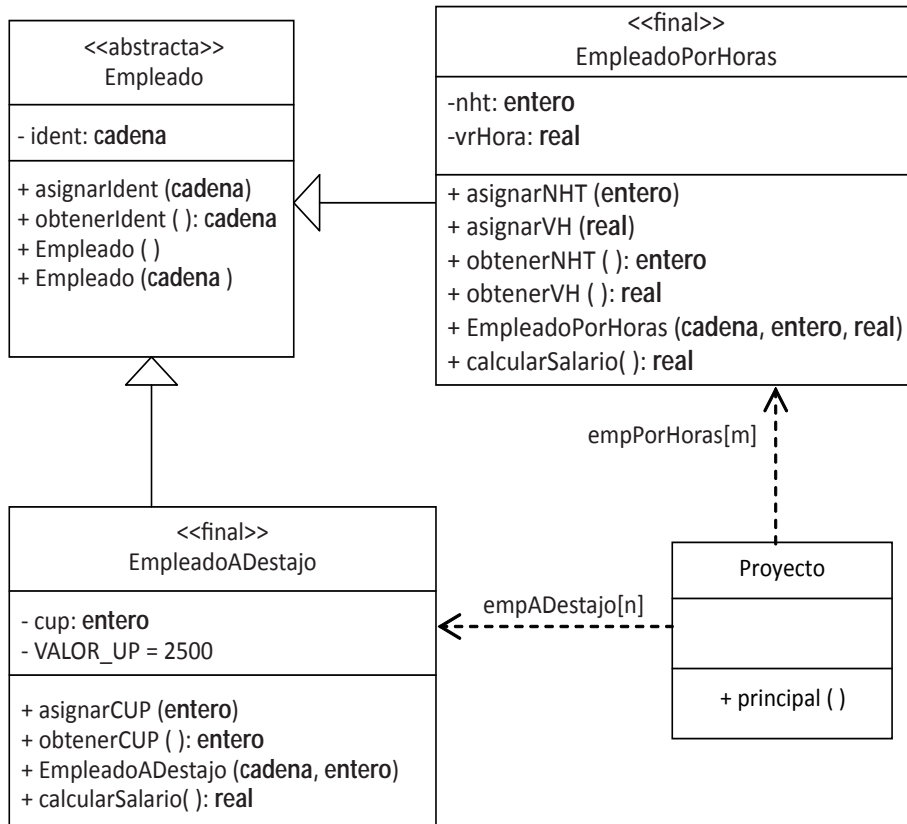
Identificación del requerimiento	Descripción	Entradas	Resultados (Salidas)
R1	Conocer la cantidad de empleados por hora y a destajo	Dos números enteros ingresados por el usuario	Las variables enteras m y n han sido inicializadas
R2	Para una cantidad conocida de empleados por hora, dada por la variable m , conocer su identificación, número de horas trabajadas y valor de la hora	El número m . Para cada empleado por hora: $ident$, nht y $vrHora$	Un vector de tamaño m ($empPorHoras$) con los datos de los empleados por hora

R3	Para una cantidad conocida de empleados a destajo, dada por la variable n , conocer su identificación y cantidad de unidades producidas	El número n . Para cada empleado a destajo: <i>ident</i> y <i>cup</i>	Un vector de tamaño n (<i>empADestajo</i>) con los datos de los empleados a destajo
R4	Para todos los empleados (por horas y a destajo): conocer su identificación y salario devengado	Los datos de todos los empleados.	Por cada empleado: identificación y salario.
R5	Conocer el salario promedio de los empleados asalariados y a destajo.	Los salarios de todos los empleados	Salario promedio de los empleados asalariados y salario promedio de los empleados a destajo

Documentación:

Identificador	Tipo	Significado
m	entero	Cantidad de empleados por hora.
n	entero	Cantidad de empleados a destajo.
ident	cadena	Identificación de un empleado (por horas o a destajo).
nht	entero	Número de horas trabajadas.
vrHora	real	Valor de la hora.
cup	entero	Cantidad de unidades producidas.
empPorHoras	EmpleadoPorHoras	Vector de empleados por hora.
empADestajo	EmpleadoADestajo	Vector de empleados a destajo.

b) Diagrama de clases



c) Seudo código

```

abstracta clase Empleado
  privado:
    cadena ident
  público:
    asignarIdent (cadena i)
      ident = i
    fin_método
    //-----
    cadena obtenerIdent ()
      retornar ident
    fin_método
    //-----
    Empleado ()
    fin_método
    //-----
    Empleado (cadena i)
      asignarIdent (i)
    fin_método
    //-----
fin_clase // Empleado
//*****
final clase EmpleadoPorHoras heredaDe Empleado
  privado:
    entero nht
    real vrHora
  público:
    asignarNHT (entero ht)
      nht = ht
    fin_método
    //-----
    asignarVH (real vh)
      vrHora = vh
    fin_método
    //-----
    entero obtenerNHT ()
      retornar nht
    fin_método
    //-----
    real obtenerVH ()
      retornar vrHora
    fin_método
    //-----

```



```

↓
EmpleadoPorHoras( ) //Constructor por defecto: se puede omitir
    fin_método
    //-----
    EmpleadoPorHoras(cadena id, entero nh, real vh)
        super(id) // invoca al constructor de la superclase
        asignarNHT(ht)
        asignarVH(vh)
    fin_método
    //-----
    real calcularSalario()
        retornar nht * vrHora
    fin_método
fin_clase // EmpleadoPorHoras
//*****
final clase EmpleadoADestajo heredaDe Empleado
    privado:
        entero cup
        const entero VALOR_UP = 2500
    público:
        asignarCUP(entero up)
            cup = up
        fin_método
        //-----
        entero obtenerCUP()
            retornar cup
        fin_método
        //-----
        EmpleadoADestajo()
        fin_método
        //-----
        EmpleadoADestajo(cadena id, entero up)
            super(id)
            asignarCUP(up)
        fin_método
        //-----
        real calcularSalario()
            retornar VALOR_UP * cup
        fin_método
fin_clase // EmpleadoADestajo
//-----
↓

```

↓
clase Proyecto

```

    estático principal()
        entero m, n
        m = Flujo.leerEntero()

        n = Flujo.leerEntero()
        EmpleadoPorHoras empPorHoras[ ] = nuevo EmpleadoPorHoras[m]
        EmpleadoADestajo empADestajo[ ] = nuevo EmpleadoADestajo[n]
        cadena ident
        entero numHT, numUP, c
        real vrH
        // ingreso de empleados por hora
        para (c = 0, c < m, c = c + 1)
            ident = Flujo.leerCadena()
            numHT = Flujo.leerEntero()
            vrH = Flujo.leerReal()
            empPorHoras[c] = nuevo EmpleadoPorHoras(ident, numHT,
                                                         vrH)

        fin_para
        // ingreso de empleados a destajo
        para (c = 0, c < n, c = c + 1)
            ident = Flujo.leerCadena()
            numUP = Flujo.leerEntero()
            empADestajo[c] = nuevo EmpleadoADestajo(ident, numUP)

        fin_para
        // informe de salarios: nómina
        real salario, suma1 = 0, suma2 = 0
        entero c
        para (c = 0, c < m, c = c + 1)
            salario = empPorHoras[c].calcularSalario()
            suma1 = suma1 + salario
            Flujo.imprimir (empPorHoras[c].obtenerIdent(), salario)

        fin_para
        para (c = 0, c < n, c = c + 1)
            salario = empADestajo[c].calcularSalario()
            suma2 = suma2 + salario
            Flujo.imprimir (empADestajo[c].obtenerIdent(), salario)

        fin_para
        real prom1 = suma1 / m
        real prom2 = suma2 / n
        Flujo.imprimir (prom1, prom2)

    fin_método // principal()
fin_clase // Proyecto

```

Observaciones:

- En el método *Proyecto.principal()* se han omitido los mensajes respectivos que deben preceder las lecturas de las variables *m* y *n*, y los mensajes al usuario para la lectura de datos dentro de los ciclos *para*, esto por simple cuestión de ahorro escritural de pseudocódigo.
- La instrucción:
EmpleadoPorHoras empPorHoras[] = nuevo EmpleadoPorHoras[m],
reserva memoria para almacenar *m* empleados en el vector de objetos *empPorHoras*. Observar que *empADestajo* es otro vector de objetos.
- Al interior de los ciclos *para* que posibilitan la entrada de datos (o llenado de los dos vectores), se requiere de una instrucción que almacene en una posición determinada del vector los datos del nuevo empleado; para esto se requiere del operador *nuevo* (que reserva memoria) y del constructor sobrecargado, como en la instrucción:

empPorHoras[c] = nuevo EmpleadoPorHoras(ident, numHT, vrH)

6.2. POLIMORFISMO

El *polimorfismo* es uno de los conceptos esenciales de la programación orientada a objetos. Si la herencia está relacionada con las clases y su jerarquía, el polimorfismo lo está con los métodos.

En general, hay tres tipos de polimorfismo: polimorfismo de sobrecarga, polimorfismo paramétrico (también llamado polimorfismo de plantillas) y polimorfismo de inclusión (también llamado redefinición o subtipado).

El *polimorfismo paramétrico* es la capacidad para definir varios métodos utilizando el mismo nombre dentro de una misma clase, pero con parámetros diferentes (figura 6.6). El polimorfismo paramétrico selecciona automáticamente el método correcto

a aplicar en función del tipo de datos pasado en el parámetro, y es equivalente al concepto de sobrecarga de un método.

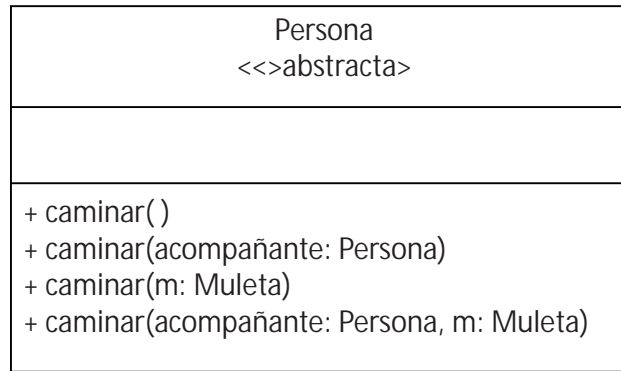


Figura 6.6. Polimorfismo paramétrico

El *polimorfismo de subtipado* permite ignorar detalles de las clases especializadas de una familia de objetos, enmascarándolos con una interfaz común (siendo esta la clase básica). Imagine un juego de ajedrez con los objetos rey, reina, alfil, caballo, torre y peón, cada uno heredando el objeto pieza (ver figura 6.7).

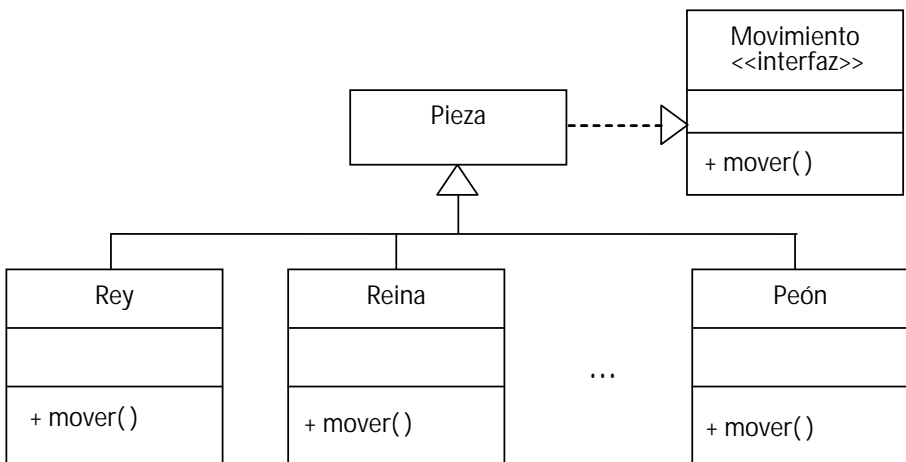


Figura 6.7. Polimorfismo de subtipado

El método *mover()* podría, usando polimorfismo de subtipado, hacer el movimiento correspondiente de acuerdo a la clase de objeto que se llame. Esto permite al programa realizar el movimiento de pieza sin tener que verse conectado con cada tipo de pieza en particular.

6.3. EJERCICIOS PROPUESTOS

1. Con la ayuda de ejemplos de la vida cotidiana, explique los siguientes términos: objeto, clase, herencia simple, herencia múltiple, atributo, método, mensaje, sobrecarga de métodos, polimorfismo y encapsulación de datos.
2. Un centro hospitalario posee N ambulancias distribuidas a lo largo y ancho de una ciudad. De cada ambulancia se conoce su localización. En caso de presentarse una emergencia (por ejemplo un accidente de tránsito o una atención domiciliaria), se toman los datos de su localización.

El problema consiste en hallar la ambulancia más cercana al punto de la emergencia, considerando que ésta puede estar ocupada en otra situación, lo que solicita hallar la siguiente ambulancia más cercana. Se debe enviar un mensaje de SOS al hospital, en caso de que todas las ambulancias estén ocupadas.

3. Defina el paquete **tiempo** con las clases **Fecha** y **Hora**.

La clase **Fecha** debe incluir los atributos enteros **día** (entre 1 y 30), **mes** (entre 1 y 12) y **año** (mayor o igual a 2006), cuyos valores predeterminados están dados para el 1 de enero de 2006; sobrecargue el constructor con tres parámetros enteros para **día**, **mes** y **año**.

La clase **Hora** se debe definir con los atributos **hor** (entre 0 y 23), **min** (entre 0 y 59) y **seg** (entre 0 y 59), cuyos valores predeterminados están dados para las 06:30:00; sobrecargue el constructor con tres parámetros enteros para **hor**, **min** y **seg**.

Ambas clases deben constar de las operaciones *asignar* y *obtener* respectivas; las funciones *asignar* deben verificar los rangos de cada atributo.

Incluir además las operaciones:

- **mostrarFecha** (), que despliega la fecha actual.
- **mostrarHora** (), que despliega la hora actual.
- **incrementarFecha** (entero d, entero m, entero a), que incrementa la fecha actual en d días, m meses y a años. Este último método debe finalizar con una fecha consistente.
- **incrementarHora** (entero h, entero m, entero s), que incrementa la hora actual en h horas, m minutos y s segundos. Este último método debe finalizar con una hora consistente. Sobrecargar este método con un objeto tipo **Hora**, así: **incrementarHora** (Hora h).

Fuera del paquete **tiempo** definir la clase **TransacciónBancaria**, con los atributos **numeroCuenta** (tipo **cadena**), **tipoTransaccion** (tipo **entero**), **saldo** (tipo **real**), **fecha** (tipo **Fecha**) y **hora** (tipo **Hora**). Realizar dos transacciones bancarias y visualizar todos sus datos.

Incrementar tanto la fecha de la primera transacción en 3 días, 5 meses y 4 años, como la hora de la segunda transacción con la hora de la primera.

4. El gobierno colombiano desea reforestar los bosques del país según el número de hectáreas que mida cada bosque. Si la superficie del terreno excede a 1 millón de metros cuadrados, entonces decidirá sembrar de la siguiente manera:

Porcentaje de la superficie del bosque	Tipo de árbol
70%	Pino
20%	Oyamel
10%	Cedro

Si la superficie del terreno es menor o igual a un millón de metros cuadrados, entonces la siembra se realizará así:

Porcentaje de la superficie del bosque	Tipo de árbol
50%	Pino
30%	Oyamel
20%	Cedro

De cada bosque se conoce su nombre, extensión en hectáreas y departamento donde se ubica; de cada árbol se conoce su nombre (vulgar) y valor de trasplante de un ejemplar.

El gobierno desea saber el número de pinos, oyameles y cedros que tendrá que sembrar cada bosque, si se conocen los siguientes datos: en 10 metros cuadrados caben 8 pinos, en 15 metros cuadrados caben 15 oyameles y en 18 metros cuadrados caben 10 cedros; se requiere una hectárea equivale a 10 mil metros cuadrados. Además se requiere responder los siguientes interrogantes: ¿Cuál es el Departamento de Colombia con más bosques reforestados? ¿Cuál fue el precio por cada reforestación realizada?

5. Escriba una jerarquía de herencia para las clases **Cuadrilátero**, **Trapezoide**, **Paralelogramo**, **Rectángulo** y **Cuadrado**. Utilice **Cuadrilátero** como superclase de la jerarquía, y haga la jerarquía tan profunda (esto es, con tantos niveles) como pueda. Los datos privados de **Cuadrilátero** deberán incluir los pares de coordenadas (x, y) para las cuatro esquinas del cuadrilátero. Escriba un método **principal()** que ejemplarice y exhiba objetos de cada una de estas clases con su área respectiva.
6. Diseñar una jerarquía de clases: **Círculo**, **Cilindro** y **CilindroHueco**. En esencia se puede decir que un objeto cilindro es un círculo con una altura y un cilindro hueco es un cilindro con un espacio hueco dentro de él. Escribir el método **Proyecto.principal()** que permita crear objetos **Círculo**, **Cilindro** y **CilindroHueco** y calcule la longitud de la circunferencia y las áreas del círculo, del cilindro y del cilindro hueco, y los volúmenes del cilindro y del cilindro hueco.

Fórmulas:

Círculo	Longitud: $2.\pi.r$ Área: $\pi.r^2$
Cilindro	Área: $2.\pi.r.h + 2.\pi.r^2$ Volumen: $\pi.r^2.h$
Cilindro hueco	Longitud: $2.\pi.(r^2 - r_{\text{interno}}^2) + 2.\pi.r.h + 2.\pi.r_{\text{interno}}.h$ Volumen: $\pi.(r^2 - r_{\text{interno}}^2).h$

Nota:

r = radio del cilindro y radio externo cilindro hueco.

r_{interno} = radio interno del cilindro hueco.

6.4. REFERENCIAS

[de L'Isle-Adam2003]: de L'Isle-Adam, Villers. Cuentos Crueles (Serie Grandes Clásicos Universales). Espasa, Madrid, 2003.

[Deitel2007]: Deitel&Deitel. Como programar en C#. Segunda edición, Pearson Educación, México, 2007.

[Eckel2002]: Eckel, Bruce. Piensa un Java. Segunda edición. Pearson Educación, Madrid, 2002.

Apéndice A

Entorno Integrado de Desarrollo Sismoo

El entorno integrado de desarrollo (IDE) Sismoo presenta el ambiente de trabajo para el usuario expuesto en la fig. A.1, donde se observa una ventana con un menú principal, un barra de iconos o herramientas, un panel lateral izquierdo para exploración de archivos, un panel lateral derecho que hace las veces de editor, donde se pueden desplegar varios archivos a la vez y un panel inferior para la emisión de mensajes al usuario sobre el estado del sistema y el proceso de compilación de las aplicaciones.

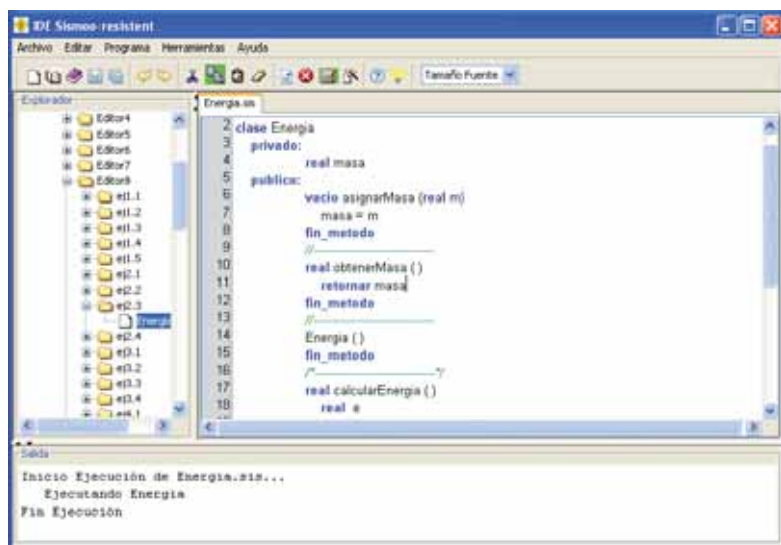


Figura A.1. Entorno integrado de desarrollo Sismoo

El entorno permite editar, compilar y ejecutar programas escritos en el seudolenguaje propuesto en MIPS00, de una manera tan simple que permite agilizar los procesos de aprendizaje de la programación. El trabajo con el editor Sismoo puede constituir un paso previo para interactuar con otros editores comerciales durante el desarrollo de software a gran escala.

Los resultados que emiten los programas se presentan en ventanas independientes a la ventana principal, como se observa en la figura A.2. Sismoo cuenta con la opción de traducción a lenguaje Java, que se considera una buena herramienta de introducción al lenguaje de programación y se proyecta la traducción a otro lenguaje como Visual Basic.Net o C#.

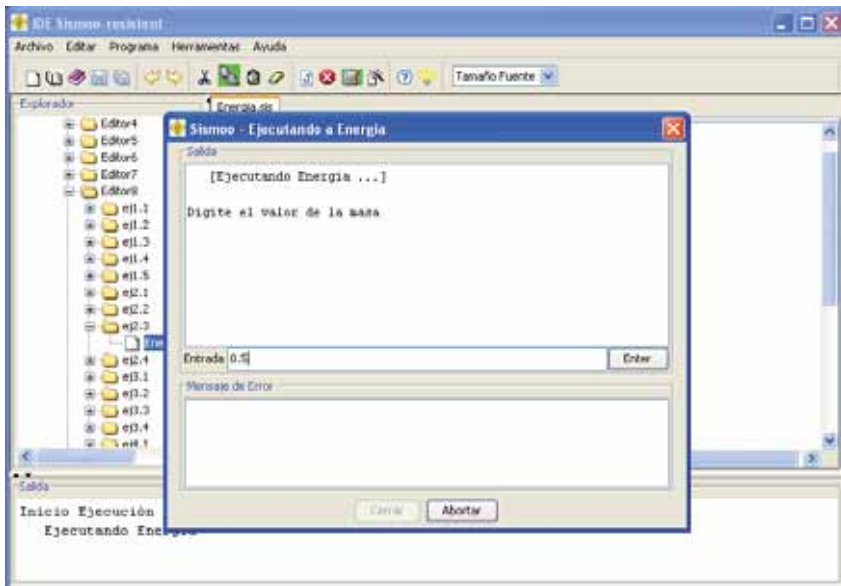


Figura A.2. Resultados de un programa

A continuación se da una breve descripción de la parte funcional del IDE Sismoo, orientada a los lectores que no estén muy familiarizados con este tipo de herramienta y posteriormente se presentan algunos aspectos referentes al diseño, dirigido a lectores avanzados para que reconozca su estructura interna.

Funcionalidad de Sismoo

Al ingresar a Sismoo se despliega la siguiente pantalla de presentación, tipo splash:



Figura A.3. Pantalla de presentación de Sismoo

En ella se ofrece al usuario la posibilidad de trabajar pseudocódigo MIPS00, o código en lenguajes Java o C#; esta dos últimas opciones están desactivadas, porque en el momento de la impresión de este libro no estaban implementadas. Futuras versiones del programa incluirán estas opciones; la versión actual corresponde a la 1.0.

El ambiente de trabajo del IDE Sismoo posee cinco opciones en el menú principal: Archivo, Editar, Programa, Herramientas y Ayuda, las cuales se muestran en detalle a continuación. Presenta además una barra de iconos que permite, de izquierda a derecha, ejecutar las siguientes tareas: Nuevo, Abrir, Cerrar, Guardar, Guardar todo, Deshacer, Rehacer, Cortar, Copiar, Pegar, Borrar todo el texto, Traducir a Java, Detener Ejecución, Compilar, Ejecutar, Ayuda, Acerca de . . . , y Tamaño de Fuente.

Una breve descripción de cada opción del menú principal se presenta a continuación.

- **Menú Archivo:** Permite la manipulación de archivos o ficheros. Por definición, todo archivo almacenado posee la extensión *sís*. El menú Archivo cuenta con ítems que permiten crear un nuevo archivo, abrirlo, cerrarlo, guardarlo, cambiarle de nombre y guardar todos los archivos abiertos.

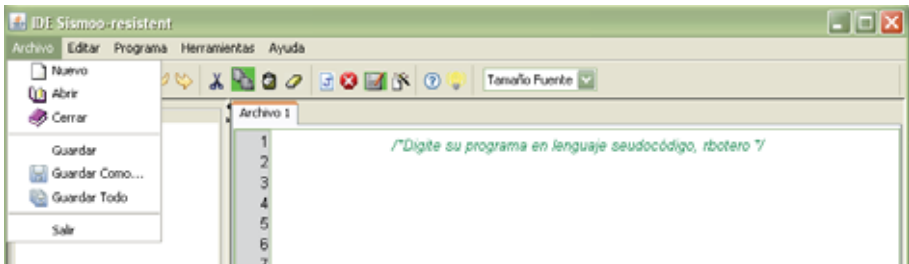


Figura A.4. Menú Archivo

- **Menú Editar:** Su funcionalidad es similar a la de cualquier editor de texto y consta de las opciones: deshacer y rehacer la última acción realizada en el editor, cortar y copiar el texto seleccionado en el editor, pegar lo almacenado en memoria o clipboard, seleccionar todo lo escrito en el editor y modificar el tamaño de la fuente.

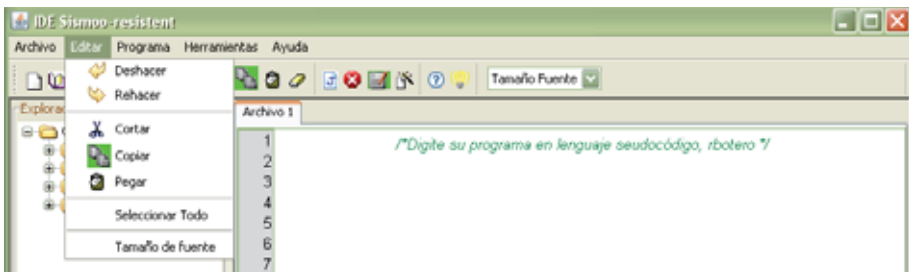


Figura A.5. Menú Editar

- **Menú Programa:** Permite compilar y ejecutar un programa escrito enseudolenguaje orientado a objetos; además también permite detener la ejecución del programa actual y traducir a Java el código escrito en el editor.

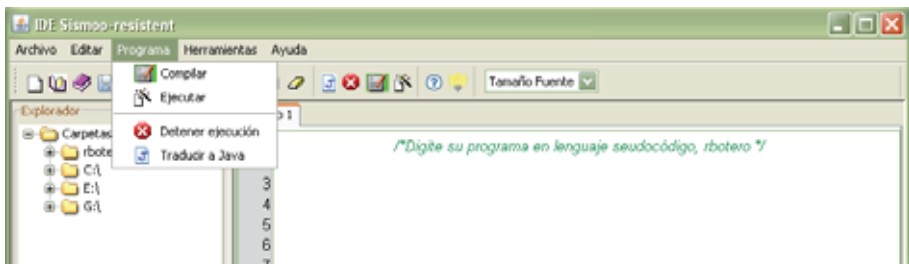


Figura A.6. Menú Programa

- **Menú Herramientas:** Sismoo requiere para su funcionamiento del Kit de Desarrollo de Java (JDK), por lo tanto debe estar instalado previamente en el computador. Esta opción permite configurar el traductor de tal manera que se puedan compilar y ejecutar los programas de la manera correcta.

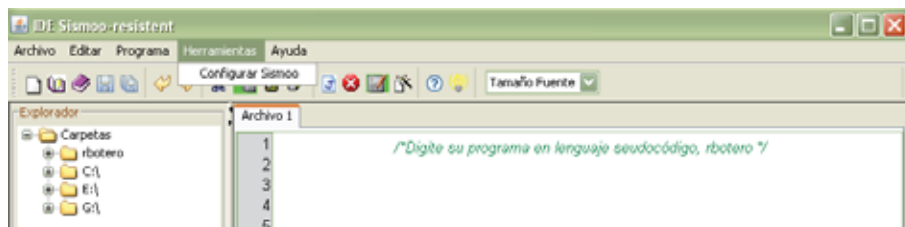


Figura A.7. Menú Herramientas

Al seleccionarla aparece la siguiente ventana:

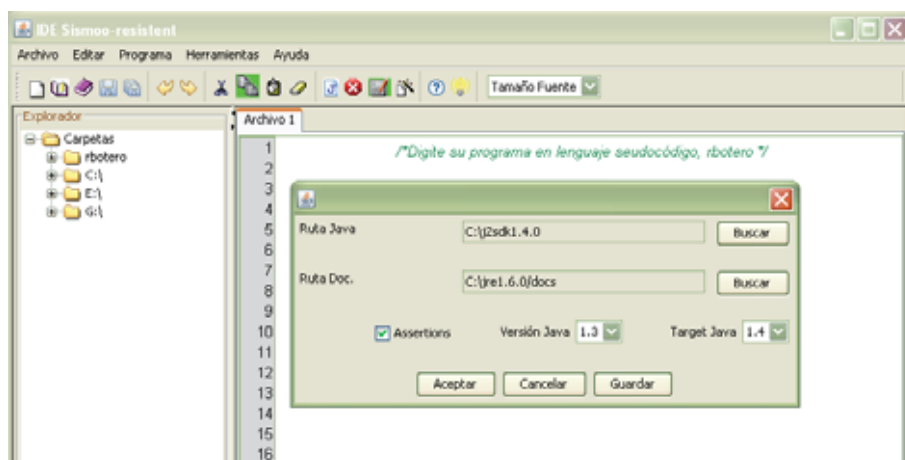


Figura A.8. Configuración de Sismoo

- **Menú Ayuda:** Permite obtener ayuda en línea acerca del programa SISMOO Resistent e informa acerca de los derechos de autor del programa.

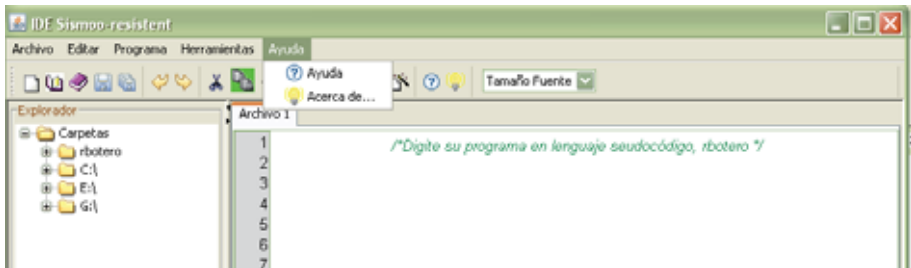


Figura A.9. Menú Ayuda

Aspectos de diseño

Algunos aspectos relacionados con el diseño y construcción de IDE traductor Sismoo se presentan a continuación, en forma grafica, para simplificar su descripción.

En la representación simbólica de SISMOO de la figura A.10, se ilustran los lenguajes que intervienen en el proceso.

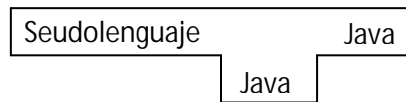


Figura A.10. Diagrama simbólica de Sismoo

Del diagrama esquemático presentado en la figura A.11, se deduce el proceso de ejecución de un programa desarrollado enseudolenguaje y la relación de Sismoo con el lenguaje de programación Java; se observan las labores de compilación y traducción a bytecode (que la realiza la aplicación Javac) y el trabajo de la máquina virtual de Java (JVM) en el manejo de excepciones y generación del programa objeto.

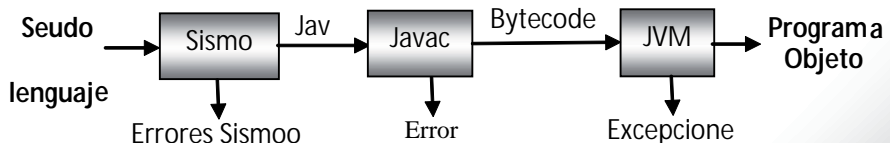


Figura A.11. Diagrama esquemático de Sismoo

Para finalizar, en la figura A.12 se ilustran los diagramas de clases conceptuales en notación UML y en la tabla A.1 una breve descripción de cada una de las clases que hacen parte de aplicativo Sismoo.

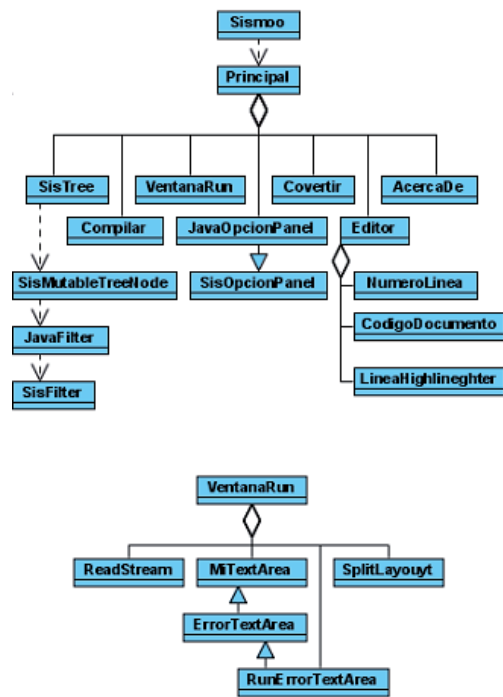


Figura A.12. Diagrama conceptual de clases Sismoo

Tabla A.1 Descripción de clases para Sismoo

Nombre	Deriva de	Breve Descripción
SISMOO	JFrame	Ventana de inicio
Principal	JFrame	Es la clase central que contiene el ambiente de trabajo de SISMOO
SisTree	JTree	Implementa el explorador
SisMutable TreeNode	DefaultMutable TreeNode	Para manejar los nodos o carpetas en el explorador
JavaFilter	FileFilter	Filtro general para los archivos

SisFilter	FileFilter	Filtra los archivos tipo SISMOO, con extensión punto sis (.sis)
Editor	JPanel	Panel principal para el editor de texto
SismoText Panel	JPanel	Panel para las carpetas de cada archivo
Numero Linea	JComponent	Maneja la numeración de las líneas de código
Codigo Documento	DefaultStyled Document	Reconoce los elementos que poseen formato especial
lineaHighlighter	DefaultHighlighter	Para resaltar el texto seleccionado
MenuPopup	JPopupMenu	Menú contextual del área de edición
VentanaRun	JDialog	Despliega la venta de ejecución
MiJTextArea	JScrollPane	Maneja el área de texto de salida en la ventana de ejecución
ErrorTextArea	MiJTextArea	Captura las excepciones o errores en tiempo de ejecución
RunErrorTextArea	ErrorTextArea	Maneja el área de texto de errores en tiempo de ejecución
ReadStream	Runnable	Controla la caja de texto de lectura en tiempo de ejecución
SplitLayout	LayoutManager	Para el diseño de los panel en la ventana de ejecución
SisActivo Panel	JPanel	Panel para la ventana de configuración
JavaOpcion Panel	SisActivoPanel	Ventana de configuración de Kit de Desarrollo de Java
Convertir	Object	Realiza la traducción a Java
Compilar	Thread	Ejecuta la compilación en Java
AcercaDe	JFrame	Ventana de Ayudas

Apéndice B

Elementos sintácticos del seudo lenguaje

B.1. Listado de palabras reservadas para el seudo lenguaje orientado a objetos

abstracta cadena Cadena capturar caracter Caracter cierto clase const defecto entero Entero estatico este Excepcion ExcepES ExcepMat falso	fin_capturar fin_clase fin_metodo fin_mientras fin_interfaz fin_para fin_probar fin_según fin_si final Flujo hasta heredaDe implementa importar interfaz logico Logico Mat mientras nuevo nulo	Objeto paquete para principal privado probar protegido publico real Real repetir retornar romper según si sino Sismoo super TipoDeDato vacio
---	---	---

B.2. Estructuras de control

ESTRUCTURA DE CONTROL	SINTAXIS GENERAL	OBSERVACIONES
Secuencia	<i>instrucción_1</i> <i>instrucción_2</i> : : <i>instrucción_N</i>	El conjunto de instrucciones conforma una secuencia o lista. Una instrucción puede ser una asignación, una decisión, un selector o un ciclo
Decisión	si (<i>e</i>) <i>instrucciones_1</i> [sino <i>instrucciones_2</i>] fin_si	<i>e</i> : expresión lógica <i>instrucciones_1</i> : bloque de instrucciones a ejecutar si la expresión <i>e</i> es verdadera <i>instrucciones_2</i> : bloque de instrucciones a ejecutar si <i>expresión</i> es falsa
Selector múltiple	según (<i>vs</i>) caso <i>c1</i> : <i>instrucciones_1</i> caso <i>c2</i> : <i>instrucciones_2</i> caso <i>cN</i> : <i>instrucciones_N</i> [sino <i>instrucciones_x</i>] fin_según	<i>vs</i> : variable selectora. Debe ser de tipo entero, carácter, tipo enumerado o subrango <i>ci</i> : constante <i>i</i> , de igual tipo al de la variable selectora <i>vs</i> . $i = 1, 2, \dots, N$ <i>instrucciones_i</i> : lista de instrucciones a ejecutar en caso de existir coincidencia entre <i>vs</i> y <i>ci</i> , $i = 1, 2, \dots, N$. Si no se presenta coincidencia, se ejecutan las <i>instrucciones_x</i>
Ciclos	mientras (<i>e</i>) <i>instrucciones</i> fin_mientras	<i>e</i> : expresión lógica <i>instrucciones</i> : instrucciones a ejecutar si la expresión <i>e</i> es verdadera; al menos una de ellas debe hacer variar la <i>e</i>
	para (<i>c = vi, e, i</i>) <i>instrucciones</i> fin_para	<i>c</i> : variable contadora del ciclo. <i>vi</i> : valor inicial para <i>c</i> <i>e</i> : expresión lógica relacionada con <i>c</i> . <i>i</i> : incremento para <i>c</i>
	repetir <i>instrucciones</i> hasta (<i>e</i>)	<i>e</i> : expresión lógica <i>instrucciones</i> : instrucciones a ejecutar si la <i>e</i> es verdadera; al menos una de ellas hace variar la <i>e</i>

B.3. Elementos estructurales del pseudo lenguaje orientado a objetos

En la especificación de la forma general de los elementos estructurales del pseudo lenguaje orientado a objetos, se utilizan las siguientes símbolos:

Corchetes: []	Todo lo encerrado entre ellos se considera opcional Si no se especifica la visibilidad (público , privado), por omisión se asume privada
Barra vertical:	Indica posibilidad, es decir, se debe escoger una de las opciones separadas por la barra vertical
Doble barra inclinada: //	Indica comentario

ESTRUCTURA DE UNA CLASE

```
[abstracto | final] [publico | privado | protegido] clase nomClase [heredaDe
nomClaseBase] [implementa nomInterfaz]
    // Cuerpo de la clase
fin_clase
```

Si no se indica la visibilidad de la clase, se asume privada.

ESTRUCTURA DE UN MÉTODO

```
[estatico][<tipo_devuelto>] nomMétodo([argumentos] )
    [ publico:
// Miembros públicos]
    [ privado:
// Miembros privados]
fin_metodo
```

<tipo_devuelto> puede ser un tipo primitivo de datos, o cualquier tipo abstracto de datos definido por el programador.

ESTRUCTURA DE UN PAQUETE

```
paquete nomPaquete
    // Cuerpo del paquete
fin_paquete
```

ESTRUCTURA DE UNA INTERFAZ

```
interfaz nomInterfaz
    // Cuerpo de la interfaz
fin_interfaz
```

ESTRUCTURA GENERAL DE UNA SOLUCIÓN ORIENTADA A OBJETOS

```
[sentencias importar ]
[definición de clases del ámbito de la solución]
clase Proyecto
    estatico principal( )
        // Cuerpo del método principal
    fin_metodo
                                fin_clase
```

Apéndice C

Glosario básico para la orientación a objetos

Abstracción. Tomar del espacio del problema aquellas características que son importantes para utilizarlas en el espacio de la solución. Una buena abstracción es aquella que enfatiza sobre detalles significativos al lector y al usuario y suprime aquellos irrelevantes o que causan distracción.

Existen dos tipos básicos: el primer tipo es la **abstracción de entidades**, que representa una entidad ya sea del dominio del problema o del dominio de la solución. El segundo tipo es la **abstracción de acciones** o **abstracción de comportamiento**, que proporciona un conjunto especializado de operaciones que pueden desempeñar un objeto en el ámbito del espacio del problema a solucionar.

Agregación. Forma de asociación que especifica una relación todo-parte entre el agregado (el todo) y las partes que lo componen.

Atributo. Miembro dato de una clase. Puede ser de un tipo primitivo o de un tipo abstracto de datos. Ver *objeto*.

Bit. Contracción del término inglés Binary digit. Condición binaria (encendido | apagado) almacenada en un medio (generalmente magnético).

Byte. Conjunto de 8 bits.

Clase. Es la abstracción de un grupo de objetos que comparten las mismas características y el mismo comportamiento.

Clase abstracta. Clase desde la cual no se pueden instanciar objetos, pero sí otras clases. El comportamiento de esta clase también es abstracto, por tanto debe ser implementado por una clase derivada (sea clase común o clase final).

Clase base. Es padre de otra clase en una relación de *generalización*, es decir, en la especificación del elemento más general. Una clase base puede ser abstracta o común pero nunca final. Tiene como sinónimo la palabra Superclase.

Clase común. Es una clase que tiene clases ascendientes y descendientes. A partir de ella se pueden instanciar objetos u otras clases.

Clase derivada. Es la hija de otra clase en una relación de *generalización*, es decir, en la relación más específica. Esta clase hereda la estructura, relaciones y comportamiento de su superclase; además puede hacer sus propias adiciones. Sinónimo: Subclase.

Clase final. Es una clase que no admite instanciar a otras a partir de ella, es decir, no admite clases descendientes. Desde una clase final sólo se pueden instanciar objetos.

Composición. Forma de asociación de agregación con fuerte sentido de posesión y tiempo de vida coincidente de las partes con el todo. Una pieza puede pertenecer a solamente una composición. Las partes con multiplicidad no fija se pueden crear después del elemento compuesto, pero una vez creadas, viven y mueren con él (es decir, comparten tiempo de vida). Tales piezas pueden quitarse explícitamente antes de la muerte del elemento compuesto. La composición puede ser recurrente.

Constructor. Método u operación con el mismo nombre de la clase, usado para crear un objeto. Un constructor establece el estado inicial de un objeto y puede ser sobrecargado varias veces.

Dato. Conjunto de bytes que operan “algo” con significado.

Destructor. Método con el mismo nombre de la clase (en C++), y usado para destruir un objeto. Un destructor da un estado final a los atributos, es el último método a invocar antes de la finalización del programa. En lenguajes como Java y C# no existen destructores, porque su acción la realiza el recolector de basura (“garbage collector”) y el método *finalize()*.

Encapsulación. Protege los atributos que conforman el objeto, al definirles una visibilidad privada. Para alterar el estado de un objeto en su atributo *x*, se requiere enviar un mensaje *asignarX (dato)*, denominado método *setX (dato)* en ciertos lenguajes de programación.

Enlace dinámico. Al correr el programa se puede cambiar el tipo de dato del objeto creado.

Enlace estático. Una vez declarado un objeto, se asocia con su tipo abstracto de dato, y no se puede modificar con posterioridad. En C++ la ligadura por defecto es estática; para implementar la ligadura dinámica es necesario declarar las funciones como virtuales.

Estado de un objeto. Identidad de un objeto en determinado momento del tiempo. La identidad la dan los valores específicos que toman los atributos del objeto. Un constructor establece el estado inicial de un objeto.

Excepción. Es una condición anormal que ocurre cuando se ejecuta un programa. Una excepción ocurre cuando al flujo normal de una solución le surge un imprevisto. En este caso se debe tomar una decisión adecuada que permita controlar el error.

Extensibilidad. Propiedad de crear nuevas clases con elementos (atributos y propiedades) de otras clases ya definidas.

Firma. Denota el tipo de visibilidad, el nombre, el tipo de retorno, el tipo y número de parámetros con los cuales se va a especificar una clase, un método o un atributo. La firma de un método se conoce como *prototipo*.

Función. Operación con o sin parámetros (por lo general uno), que retorna un valor único al punto de la llamada (*paso de un mensaje* al objeto).

Generalización/Especialización. Relación taxonómica entre un elemento más general (generalización) y un elemento más específico (especialización), el cual es

completamente consistente con el elemento más general y contiene información adicional. Una relación de generalización se establece entre elementos del mismo tipo, como clases, paquetes y otros tipos de elementos.

Herencia. Proceso que permite a unos elementos más específicos incorporar la estructura y el comportamiento definidos por otros elementos más generales. La herencia puede crear una jerarquía entre clases, lo cual a su vez permite realizar nuevas clases con base en una (herencia simple) o más clases ya existentes (herencia múltiple). La nueva clase adquiere todas las características y el comportamiento de las que hereda, a la vez que puede agregar otras. Esto permite reutilizar código y generar nuevos programas por extensión.

La herencia múltiple puede conducir a un conflicto cuando los padres tienen métodos iguales. Para evitar este inconveniente, Java y C# - entre otros lenguajes- manejan la herencia múltiple creando una segunda clase padre llamada *Interfaz*, reemplazando el termino *herencia* por el de *implementa*.

Instancia. Valor específico de una clase, es decir, un objeto en particular. Un objeto se instancia cuando se ejecuta su constructor o cuando se le envía un *mensaje* que cambie su estado -con un método asignarX (dato).

Interfaz. Es una clase sin atributos, con los comportamientos especificados a nivel de prototipo o firma. Por definición una interfaz es abstracta, por tanto sus comportamientos deben ser implementados por otras clases (que implementan la interfaz). Las interfaces son una manera de tratar la herencia múltiple.

Jerarquía. Es una disposición que establece relaciones lógicas entre clases u objetos. Un objeto puede estar formado por una cantidad de objetos distintos (*agregación*), o puede heredar las características y comportamientos de otro (*generalización*).

Mensaje. Ejecución de un método.

Método. También se denomina *función miembro* (como en C++) u *operación*

(UML). Es una operación definida sobre un objeto, es un algo que el objeto sabe (o conoce) y por tanto define su comportamiento. Todo objeto debe tener al menos un atributo (X) y tres métodos: un constructor, un modificador de atributo para controlar el estado del objeto (asignarX ()) y un recuperador de la información privada del objeto (obtenerX ()).

Método de clase. Ver *Método estático*.

Método estático. Es un método que no requiere de una instancia u objeto para ejecutar la acción que desarrolla, debido a que tiene la semántica de una función global. Sinónimo: Método de clase.

Método de instancia. Método que requiere de una instancia (objeto) para ejecutar su acción mediante el envío de un *mensaje*.

Modularidad. Consiste en dividir un sistema en módulos con el fin de reducir la complejidad. Cada módulo se puede compilar de manera independiente, tiene la posibilidad de conectarse con otros módulos y realiza una tarea bien específica y delimitada. Cuando se agrupan clases con dirección o comportamiento común en un sólo paquete, se presenta la modularidad basada en paquetes.

Objeto. Instancia de una clase. Es un tipo de dato abstracto. Tiene *características* llamadas atributos y comportamientos denominados métodos. Un objeto es una abstracción que puede representar algo tangible o no tangible asociado al dominio del problema, y que refleja la información sobre un sistema.

Persistencia. Ya creado un objeto, este consume un espacio de memoria y un tiempo de vida. La función de la persistencia es permitirle que esté disponible durante la ejecución de una aplicación. Así, cuando un objeto deja de ser persistente, se hace obsoleto y ya no tiene importancia para la aplicación, es decir, se convierte en “basura” y debe liberarse de la memoria. Este proceso de limpieza se conoce como “recolección de basura”; en algunos lenguajes (Java) es responsabilidad de un hilo de ejecución (thread), por lo tanto siempre se está liberando memoria, de tal forma que se use la

realmente necesaria. En otros lenguajes (C++), la recolección de basura debe ser explícita, lo que implica trabajo extra para el programador.

Polimorfismo. Se presenta cuando un método funciona de varias maneras, según el objeto que lo invoque. Hay tres formas de implementar polimorfismo: por sobre escritura (a través de la ligadura dinámica), por sobrecarga de métodos (polimorfismo por parámetros) y polimorfismo por reemplazo (una clase derivada y su clase base implementan de manera distinta la misma operación).

Procedimiento. Operación que puede tener cero, uno o más parámetros de salida.

Prototipo. Ver *firma*.

Reusabilidad. Utilizar los métodos públicos de clases existentes (su interfaz pública) y adecuarlos a nuestra necesidad, incorporándolos al espacio del dominio del problema o al espacio del dominio de la solución.

Sobrecarga. La sobrecarga se aplica al poner el mismo nombre a varios métodos, pero cada uno recibe parámetros diferentes. De esta manera, consiste en tomar una firma existente y pasarle parámetros distintos; la especificación del parámetro definirá cuál método será el que se enlace en tiempo de ejecución.

Subclase. Ver *Clase derivada*.

Superclase. Ver *Clase base*.

Tipificación. Conocida también como verificación de tipos y se refiere al uso de datos abstractos. Existen dos tipos de tipificación: fuerte y débil. La fuerte detecta los errores desde la compilación y la débil únicamente marca errores en la ejecución. Java hace ambos tipos de verificaciones.

Tipo de Dato, TD. Forma de representación de un dato. Existen seis tipos de dato primitivos (nativos, intrínsecos o estándar): entero, real, lógico, vacío, carácter y cadena.

Tipo de Dato Abstracto, TDA. Son objetos, estructuras de datos (extendidas) a las que se les aplica una característica (son creados, por ejemplo “Entero”).

Visibilidad. Es el tipo de permiso que se establece para una clase, sus atributos y operaciones. La visibilidad puede ser pública, privada, protegida o amigable. La visibilidad pública implica que los métodos pertenecientes o no a la clase pueden acceder a lo declarado público. La visibilidad privada implica que sólo los métodos de la clase pueden usar lo privado definido en ella; la visibilidad protegida implica que sólo la misma clase o su descendencia tiene el derecho de usar lo declarado protegido. Por último, la visibilidad amigable se hace implícita para aquellas clases que están almacenadas en el mismo lugar físico; por ejemplo, en el mismo directorio, así que pueden compartir información entre sí.

Índice Analítico

A

Abstracción de clases, 14, 16, 23

Agregación, 205, 211, 215, 264

Alcance, 93 (V. *también* Visibilidad)

Algoritmo cualitativo, 17, 101

Aprendizaje basado en problemas, 2, 12, 14

Aprendizaje colaborativo, 12, 14

Aprendizaje por descubrimiento, 11, 14

Aprendizaje significativo, 12, 14

Argumento, 106, 149

Arreglo, 161

 asignación de datos a un, 164

 declaración de un, 162

 dimensionamiento de un, 163

 acceso a los elementos de un, 165

Asociación, 23, 125, 205

 cardinalidad de una, 207 (V. *también* Multiplicidad)

Atajo, 103

Atributo, 17, 34, 91

 estático, 40

B

Bandera, 142

Bucle, 133 (V. *también* Estructura iteración)

Búsqueda binaria, 179, 196

C

Campo variable local, 51 (V. *también* Variable local)

Características, 34

Casos de uso, 16

Ciclo, 133, 142 (V. *también* Estructura de control iteración)

mientras, 133, 138, 140

para, 135, 138

repetir / hasta, 104, 136, 141

Clase, 32, 264

 abstracta, 92, 210, 238, 264

 base, 209, 236, 238, 265 (V. *también* Superclase)

 Cadena, 70

 Carácter, 63

 cuerpo de una, 92

 derivada, 209, 236, 265 (V. *también* Subclase)

 Entero, 59, 75

 Flujo, 17, 20, 65, 75

 final, 237, 265

 Lógico, 64

 Mat 54, 71

 Matriz, 182, 190

 Objeto, 67

 Proyecto, 23, 39, 130, 215, 220

 Real, 61

 TipoDeDato, 68

 Vector, 168

 VectorOrdenado, 196

Clases de uso común, 22, 58, 73, 81

Cláusula, 119

Comportamientos, 13, 34 (V. *también* Método)

Composición, 211, 221, 265

Constructor, 34, 42, 265 (V. *también* Método constructor)
 sobrecargado, 44
Contratos, 17
Conversión forzada de tipos, 54

D

Dato, 47, 265
Decisión anidada, 120
Dependencia, 42, 209, 221
Diagrama conceptual, 16, 125
Diagrama de clases, 16, 38
Divide y vencerás, 23
Documentación, 17, 131

E

Ecuación de recurrencia, 146
Espacio de nombres, 20, 225 (V. *también* Paquete)
Especialización, 209
Especificador de acceso, 93 (V. *también* Visibilidad)
Estereotipado, 95
Estereotipo, 210, 238
Estructura de control, 117, 261
 decisión, 118, 124
 iteración, 133
 secuencia, 118, 261
 selección, 118
 selector múltiple, 118, 120, 133, 261
Estructura de datos, 161
 externa, 65
 interna, 65
Excepción, 61, 169, 266
Expresión, 52

F

Función, 95

Firma, 38 (V. *también* Prototipo)

G

Generalización, 209, 267

H

Herencia, 209, 212, 237, 264

simple, 236

múltiple, 238 (V. *también* Interfaz)

I

Identificación de requerimientos, 128

Identificador, 49

Índice, 168

Interfaz, 238

Interruptor , 142 (V. *también* Bandera)

Iteración, 133 (V. *también* Estructura de control iteración)

M

Matriz, 182

Mensaje, 267

paso de un, 42

Método, 17, 94

analizador, 23, 34

asignarX(), 35

constructor, 34, 41

cuerpo de un, 96

declaración de un, 94

definición de un, 95

estático, 20, 71, 98

- finalización, de un 95, 97
- invocación de un, 97
- modificador, 34
- nombre del, 95
- obtenerX(), 35
- principal, 24, 38, 94, 125
- recursivo, 145

Modelo verbal, 17, 115

Multiplicidad, 207 (V. *también* Asociación)

O

Objeto, 31, 94

- estado de un, 32

Operaciones, 17, 91

Operador, 52

- de concatenación, 125
- de resolución de ámbito, 225
- nuevo, 42
- prioridad de un, 54

Operando, 52

Orden de magnitud, 176

P

Palabras clave, 19, 103

Paquete, 23 (V. *también* Espacio de nombres)

- contenedor, 65
- sistema, 20, 65, 74

Paquetes de uso común, 65

Parámetro, 94, 95, 105

- actual, 145
- de entrada, 105
- de entrada y salida, 105
- de salida, 105
- formal, 145, 149

Paso de parámetros, 105

- por dirección, 106
 - por referencia, 106
 - por valor, 106
- Polimorfismo, 246
 - de subtipado, 247
 - paramétrico, 247
- Postcondición, 17
- Precondición, 17
- Procedimiento, 94
- Prototipo, 38
 - definición de, 94 (V. *también* Firma)

R

- Realización, 212
- Recursión
 - directa, 145
 - indirecta, 145
- Relación
 - de agregación, 205 (V. *también* Agregación)
 - de composición, 211
 - de dependencia, 209 (V. *también* Dependencia)
 - de uso, 125
- Responsabilidades de las clases, 17

S

- Selector múltiple, 118 (V. *también* Estructura de control selector múltiple)
- Sentencia
 - retornar, 96
 - según, 120
 - si, 118
- Seudo código orientado a objetos, 18
- Sobrecarga
 - de operadores, 21
 - de métodos, 106

Subclase, 209 (V. *también* Clase derivada)

Subprograma, 94

Superclase, 209 (V. *también* Clase base)

Switche, 142 (V. *también* Bandera)

T

Tabla de requerimientos, 15

Tipo abstracto de datos, 93

Tipo de datos, 47, 269

 cadena, 70

 carácter, 63

 entero, 59

 lógico, 64

 primitivo, 48

 real, 61

Tipo de retorno, 95

V

Variable, 49

 declaración de, 51

 local, 51

Vector, 168

Visibilidad, 93, 95

 privada, 34

 pública, 34

Este libro se procesó con
Microsoft Office Word 2007.

Los diagramas de clase de los cuentos,
epígrafes de algunos capítulos, se realizaron
con una copia de evaluación
de VP-UML EE versión 6.3.

El software Sismoo fue desarrollado en
la edición Java2SE 1.6
Medellín-Colombia, octubre de 2009.