

## PROBLEMAS, SOLUCIONES Y PROGRAMAS

01



# 1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

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

## 2. Motivación

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

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

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

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

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

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

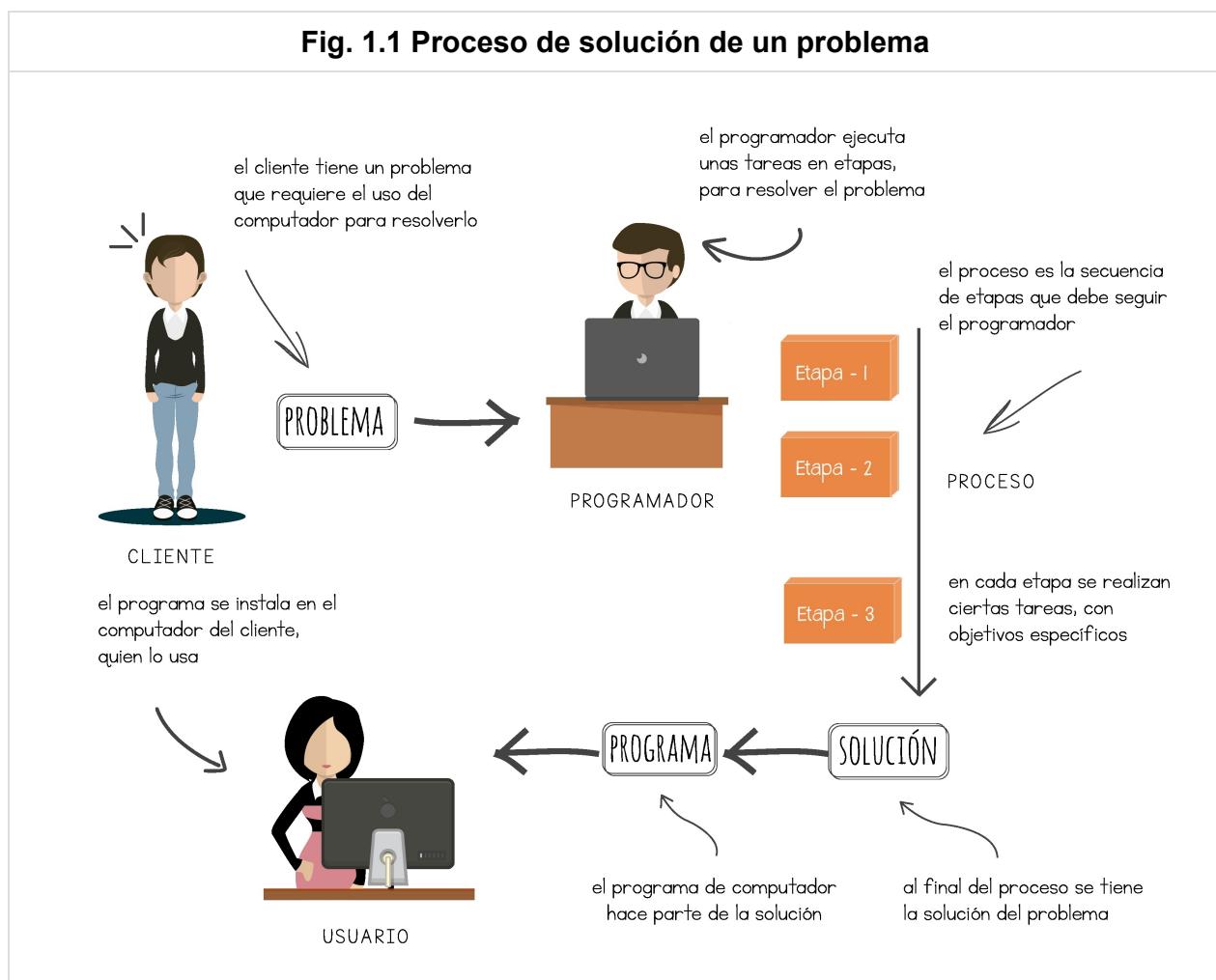
Bienvenidos, entonces, al mundo de la construcción de programas. Un mundo en **constante evolución**, en donde hay innumerables áreas de aplicación y posibilidades profesionales.

### 3. Problemas y Soluciones

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

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

**Fig. 1.1 Proceso de solución de un problema**



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

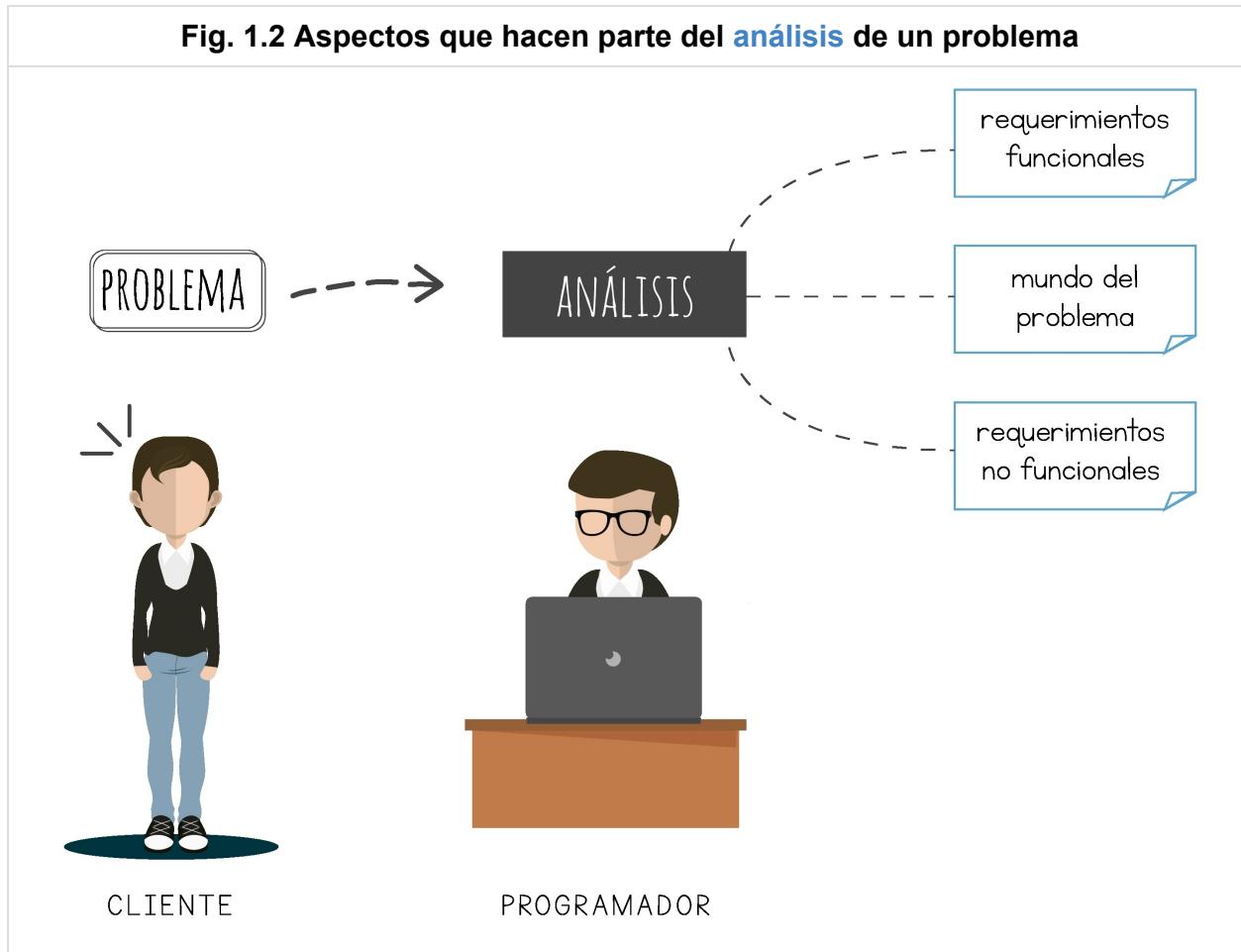
## 3.1. Especificación de un Problema

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

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

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

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



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

## Ejemplo 1

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

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

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

## Tarea 1:

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

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

<b>Cliente</b>	
Usuario	
Requerimiento funcional	
Mundo del problema	
Requerimiento no funcional	

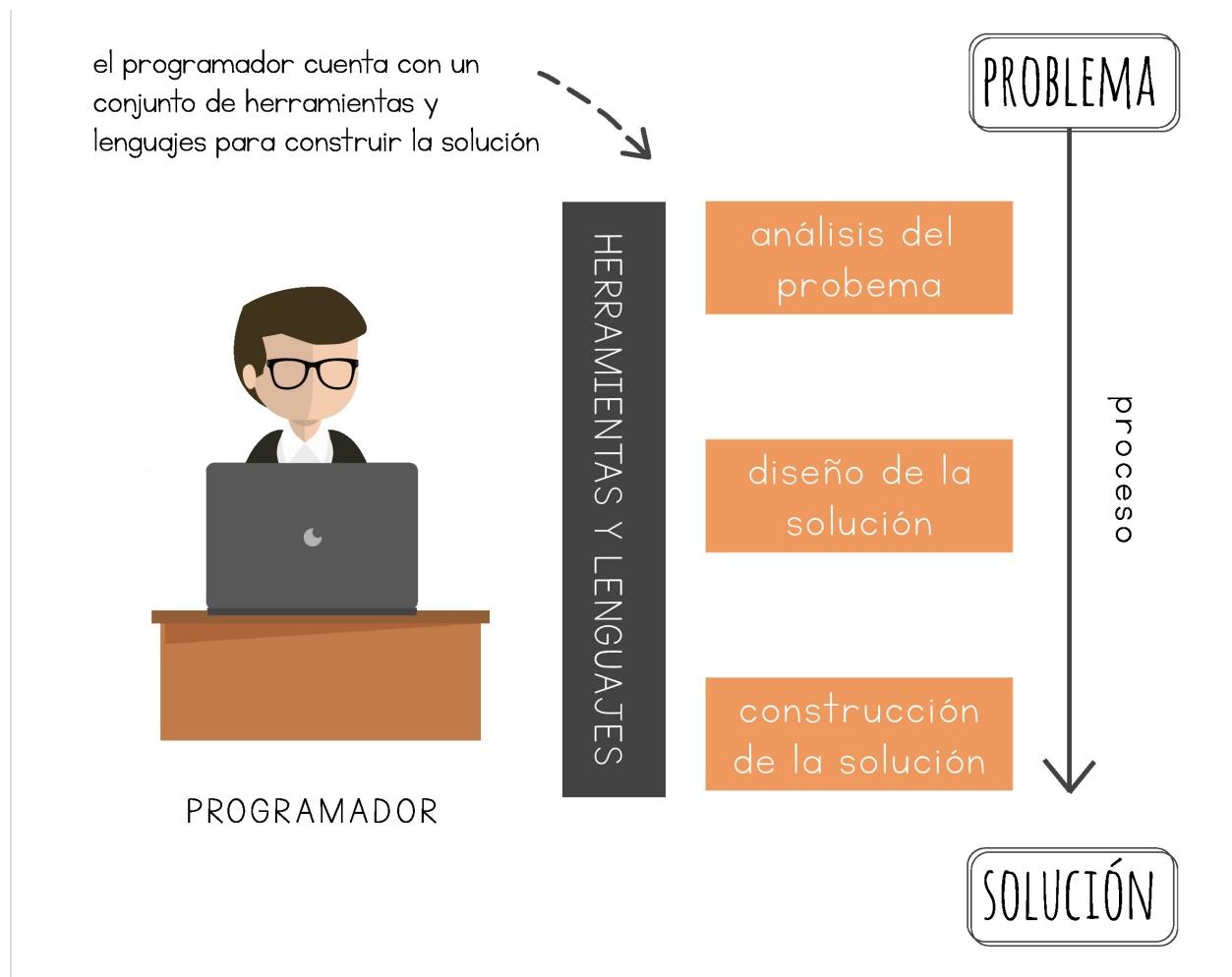
Analizar un problema significa entenderlo e identificar los tres aspectos en los cuales siempre se puede descomponer: los requerimientos funcionales, el mundo del problema y los requerimientos no funcionales. Esta división es válida para problemas de cualquier tamaño.

## 3.2. El Proceso y las Herramientas

Entender y especificar el problema que se quiere resolver es sólo la primera etapa dentro del

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

**Fig. 1.3 Principales etapas del proceso de solución de problemas**



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

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

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

[especificación](#) es parte integral de la solución.

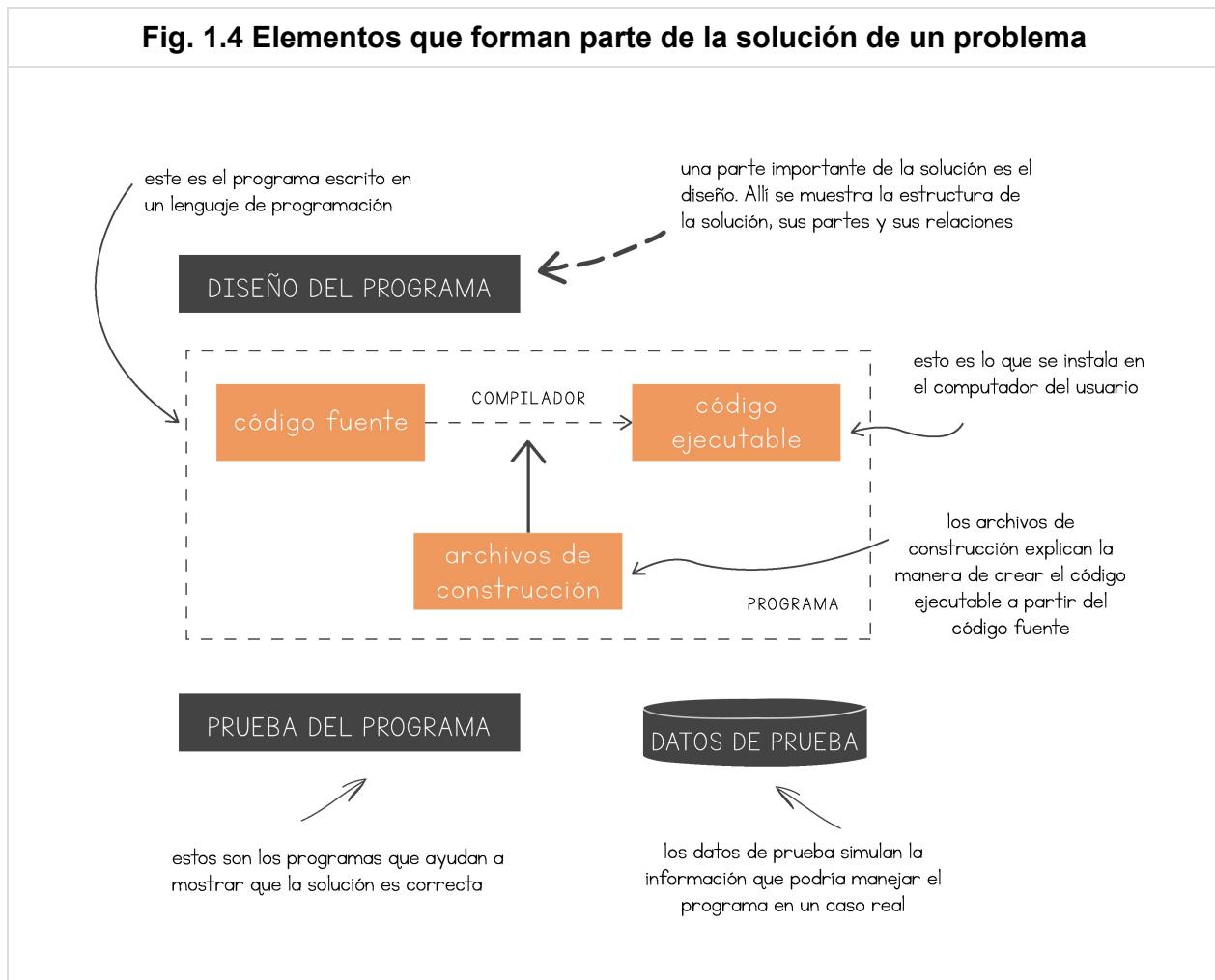
- **Construcción de la solución:** tiene como objetivo **implementar** el programa a partir del [diseño](#) y probar su correcto funcionamiento.

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

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

### 3.3. La Solución a un Problema

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

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

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

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

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

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

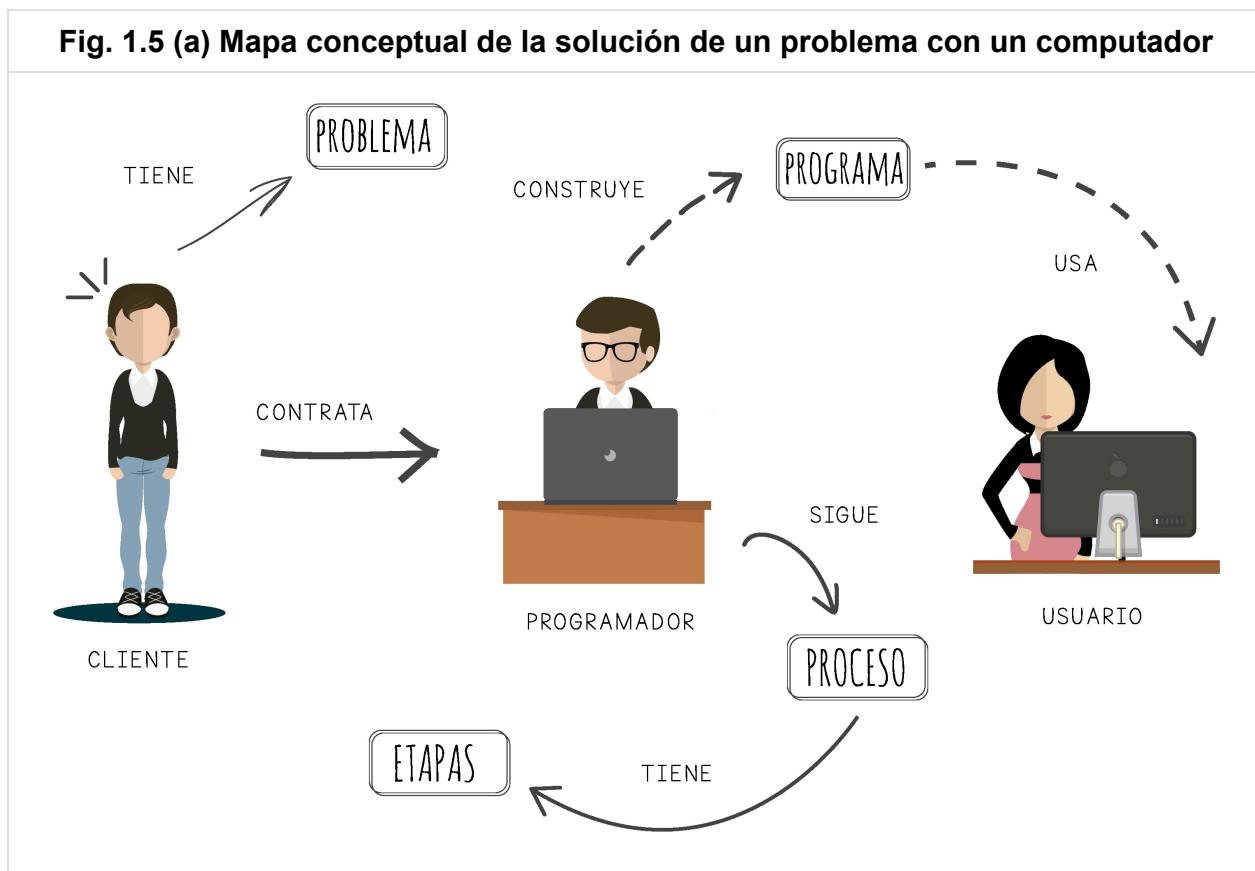
capaz de validar que para esos datos predefinidos (y que simulan datos reales), el programa funciona bien.

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

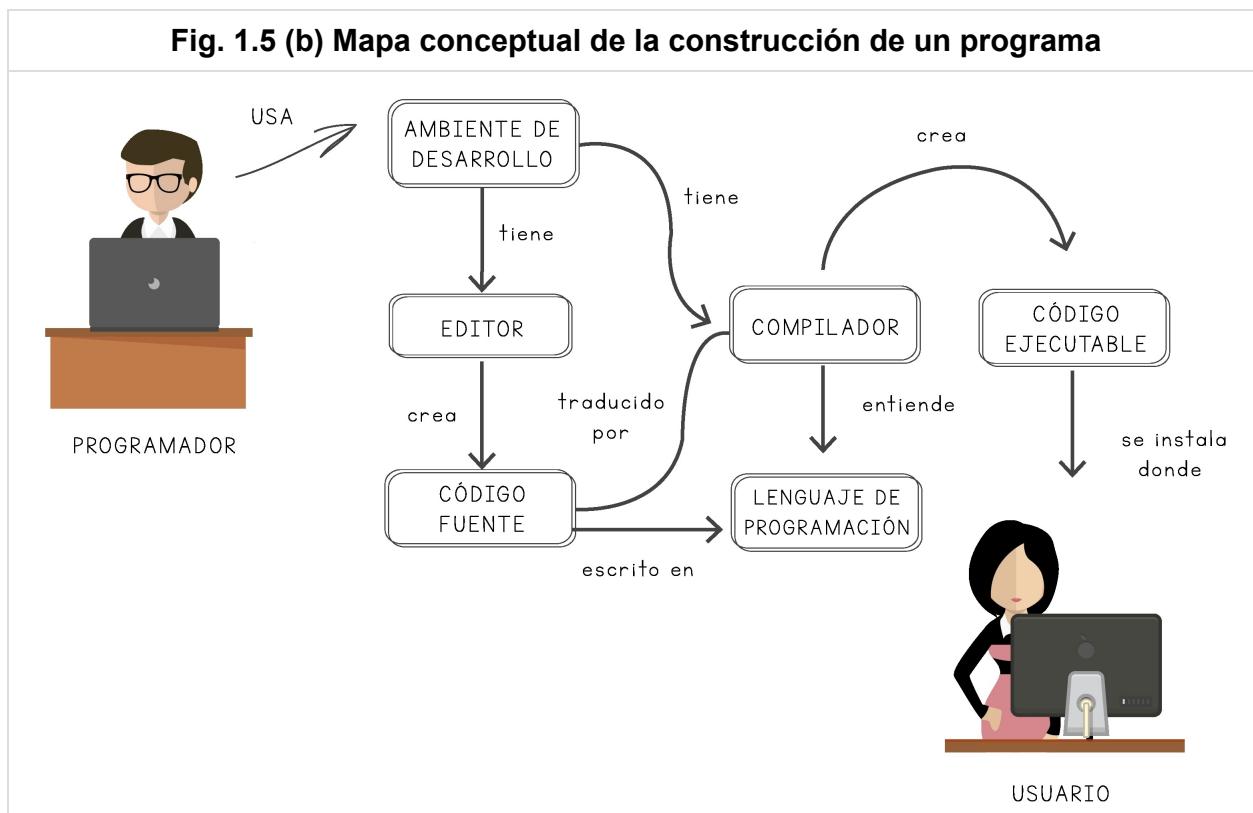
Si por alguna razón el problema del cliente evoluciona (por ejemplo, si el cliente pide un nuevo [requerimiento funcional](#)), cualquier programador debe poder leer y entender el [diseño](#), añadirle la modificación pedida, ajustar el programa y extender las pruebas para verificar la nueva extensión.

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

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



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



## 4. Casos de Estudio

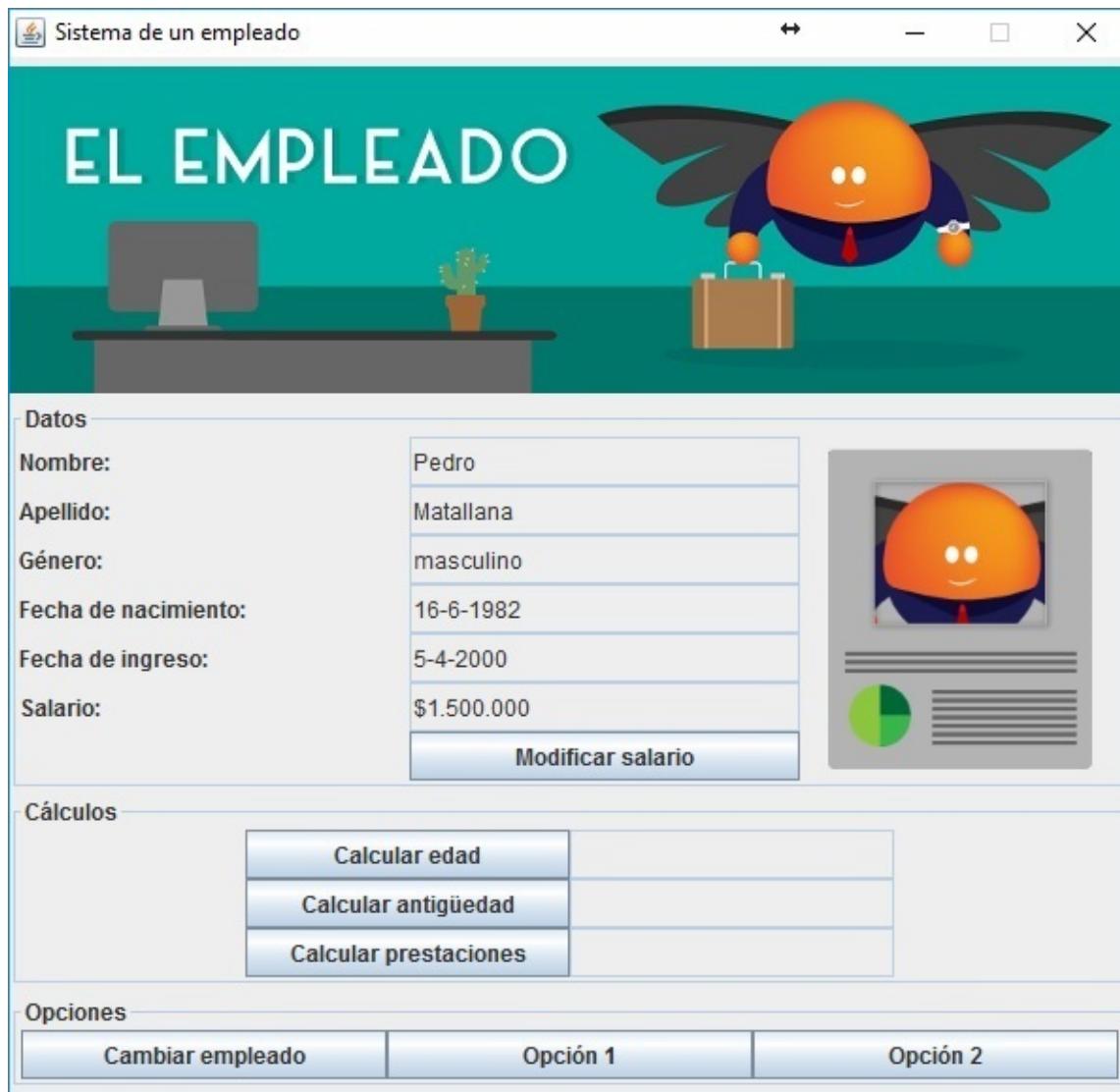
Los tres casos de estudio que se presentan a continuación serán utilizados en el resto del capítulo para ilustrar los conceptos que se vayan introduciendo. Puede encontrar estos casos de estudio en la [página web](#). Se recomienda leerlos detenidamente antes de continuar y tratar de imaginar el funcionamiento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

### 4.1 Caso de Estudio N° 1: Un Empleado

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

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

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



## 4.2. Caso de Estudio Nº 2: Un Simulador Bancario

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

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

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

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

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

<b>Datos cliente</b>			
Nombre:	Sergio López	Cédula:	50.152.468
<b>Información Bancaria</b>			
<b>Cuenta de ahorros</b>			
Saldo ahorros:	\$ 0,00 [0.6%]	<b>Consignar</b>	<b>Retirar</b>
<b>Cuenta corriente</b>			
Saldo corriente:	\$ 0,00	<b>Consignar</b>	<b>Retirar</b>
<b>CDT</b>			
Saldo CDT:	\$ 0,00 [0.0%]	<b>Abrir</b>	<b>Cerrar</b>
Mes:	1	<b>Avanzar mes</b>	
Total: \$ 0,00			
<b>Opciones</b>			
<b>Opción 1</b>		<b>Opción 2</b>	

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

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

### 4.3. Caso de Estudio Nº 3: Un Triángulo

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

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



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

área y la altura del triángulo (en píxeles).

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

# 5. Comprensión y Especificación del Problema

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

Es imposible resolver un problema que no se entiende.

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

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

## 5.1. Requerimientos Funcionales

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

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

### Ejemplo 2

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

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

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

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

Para determinar si algo es o no un [requerimiento funcional](#), es conveniente hacerse tres preguntas:

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

restricciones.

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

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

## Ejemplo 3

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

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

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

## Tarea 2

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

## Requerimiento Funcional 1

	<b>Nombre</b>	
	<b>Resumen</b>	
	<b>Entradas</b>	
	<b>Resultado</b>	

## Requerimiento Funcional 2

	<b>Nombre</b>	
	<b>Resumen</b>	
	<b>Entradas</b>	
	<b>Resultado</b>	

## Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

## Tarea 3

**Objetivo:** Crear habilidad en la identificación y [especificación](#) de requerimientos funcionales.

Para el [caso de estudio 3](#), un programa para manejar un triángulo, identifique y especifique tres requerimientos funcionales.

### Requerimiento Funcional 1

	<b>Nombre</b>	
	<b>Resumen</b>	
	<b>Entradas</b>	
	<b>Resultado</b>	

## Requerimiento Funcional 2

	<b>Nombre</b>	
	<b>Resumen</b>	
	<b>Entradas</b>	
	<b>Resultado</b>	

## Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

## 5.2. El Modelo del Mundo del Problema

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

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

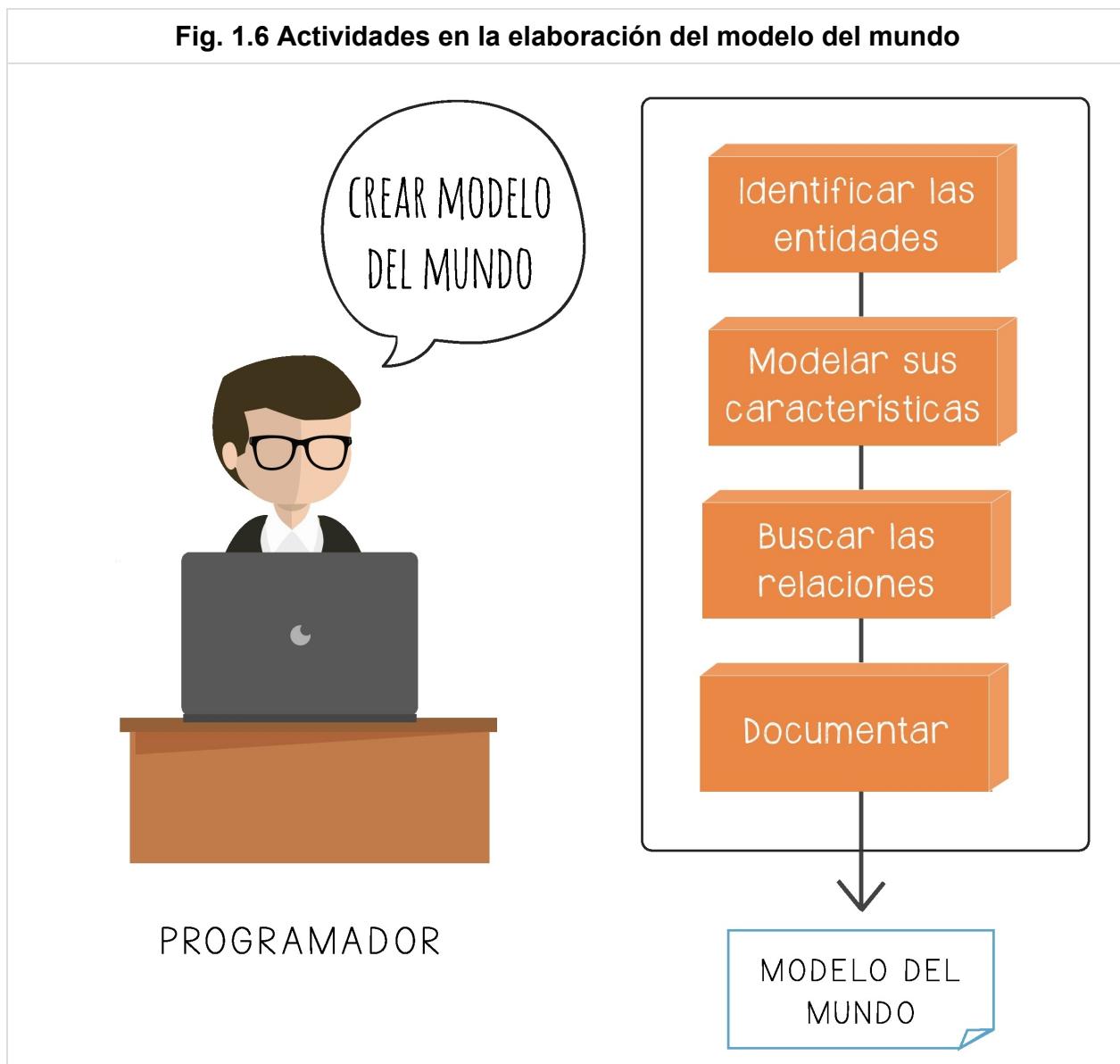
Como salida de esta actividad, se debe producir lo que se denomina un **modelo del mundo** del problema, en el cual hayamos identificado todos los elementos del mundo que participan en el problema, con sus principales características y relaciones. Este modelo será utilizado en la etapa de [diseño](#)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

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

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

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

**Fig. 1.6 Actividades en la elaboración del modelo del mundo**



### 5.2.1. Identificar las Entidades

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

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

Una convención es una regla que no es obligatoria en el [lenguaje de programación](#), pero que suelen respetar los programadores que utilizan el lenguaje. Por ejemplo, por convención, los nombres de las clases comienzan por mayúsculas.

Seguir las convenciones hace que sea más fácil entender los programas escritos por otras personas. También ayuda a construir programas más "elegantes".

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

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

## Ejemplo 4

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

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

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

## Tarea 4

**Objetivo:** Identificar las entidades del mundo para el [caso de estudio 3](#): un programa que maneje un triángulo.

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

	Nombre	Descripción
Entidad		
Entidad		
Entidad		

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

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



## 5.2.2. Modelar las Características

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

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

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

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

## Ejemplo 5

**Objetivo:** Mostrar la manera de identificar y modelar los atributos de una [clase](#).

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

**Clase:** Empleado

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



### Clase: Fecha

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

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

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



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

## Tarea 5

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

**Clase:** SimuladorBancario

Atributo	Valores Posibles

Diagrama UML:



**Clase:** CuentaCorriente

Atributo	Valores Posibles

Diagrama UML:



**Clase:** CuentaAhorros

Atributo	Valores Posibles

Diagrama UML:



**Clase:** CDT

Atributo	Valores Posibles

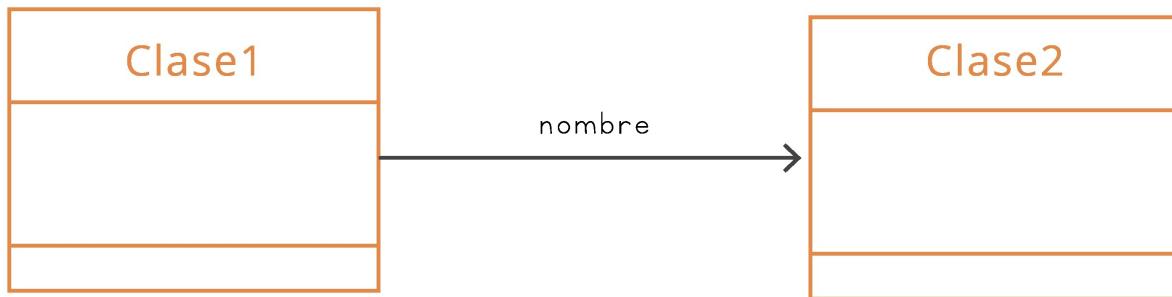
Diagrama UML:



### 5.2.3. Las Relaciones entre las Entidades

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

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



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

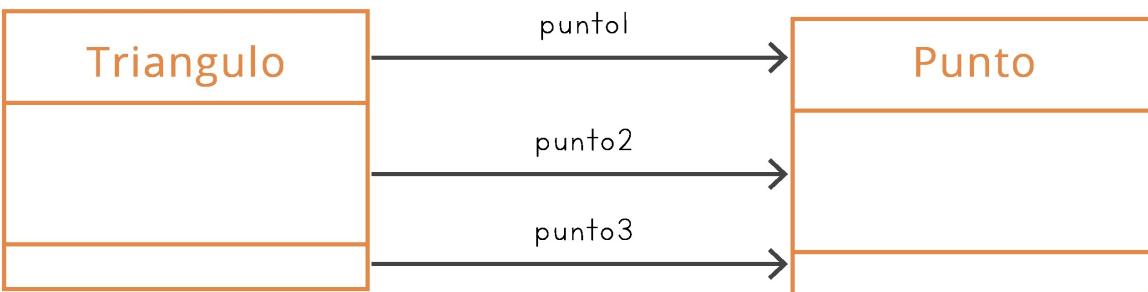
El nombre de la **asociación** sigue las mismas convenciones del nombre de los atributos y debe reflejar la manera en que una **clase** utiliza a la otra como parte de sus características.

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

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



### CASO DE ESTUDIO DEL EMPLEADO



### CASO DE ESTUDIO DEL TRIÁNGULO

#### Caso de Estudio del Empleado:

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

#### Caso de Estudio del Triángulo:

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

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

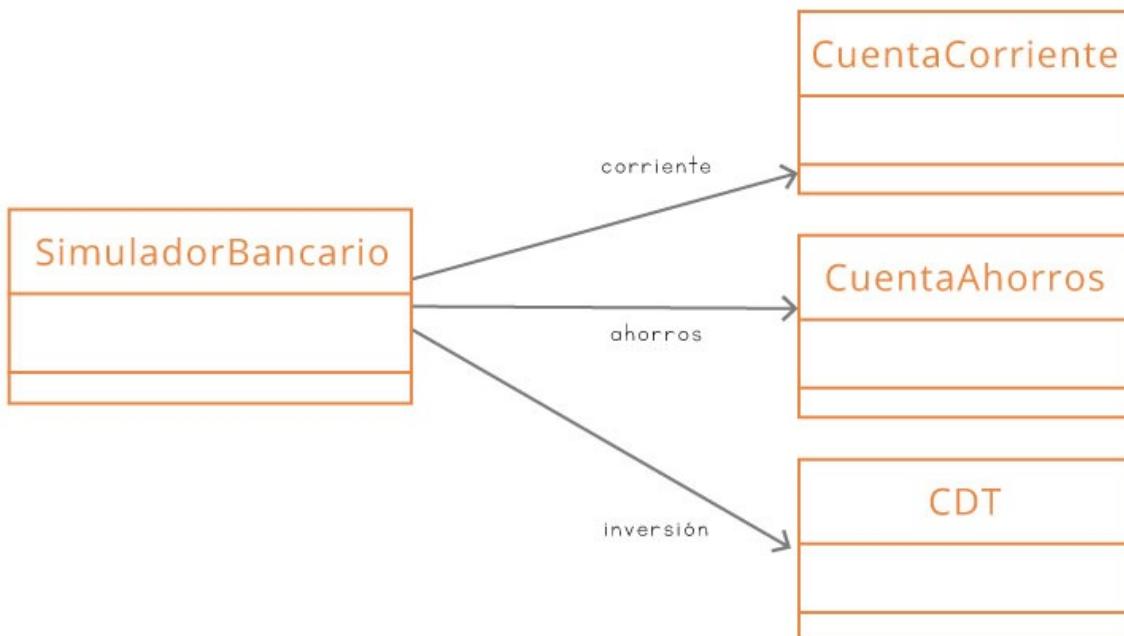
Volveremos a abordar el tema de las relaciones entre entidades en los niveles posteriores, así que por ahora sólo es importante poder identificar las relaciones para casos muy simples. En el ejemplo 6 se muestran y explican las relaciones que existen entre las entidades del caso del simulador bancario.

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

## Ejemplo 6

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

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



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

"corriente" y refleja el hecho de que una cuenta bancaria tiene una cuenta corriente como parte de ella.

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

## 5.3. Los Requerimientos no Funcionales

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

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

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

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

# 6. Elementos de un Programa

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

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

## 6.1. Algoritmos e Instrucciones

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

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

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

### Tarea 6

**Objetivo:** Reflexionar sobre el nivel de precisión que debe ser usado en un [algoritmo](#) para evitar ambigüedades.

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

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

**Utilice este espacio para anotar sus conclusiones:**



## Tarea 7

**Objetivo:** Entender la complejidad que tiene la tarea de escribir un [algoritmo](#).

Esta tarea es para ser desarrollada en parejas:

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

Dibujo:	Algoritmo
	

**Haga una síntesis de los resultados obtenidos:**

---

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

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

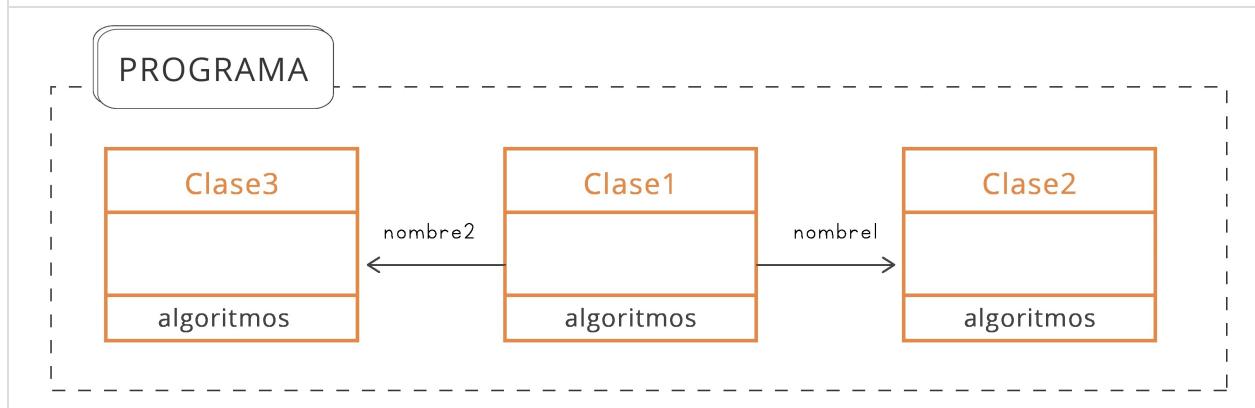
Un [programa de computador](#) está compuesto por un conjunto de algoritmos, escritos en un [lenguaje de programación](#). Dichos algoritmos están estructurados de tal forma que, en conjunto, son capaces de resolver el problema.

## 6.2. Clases y Objetos

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

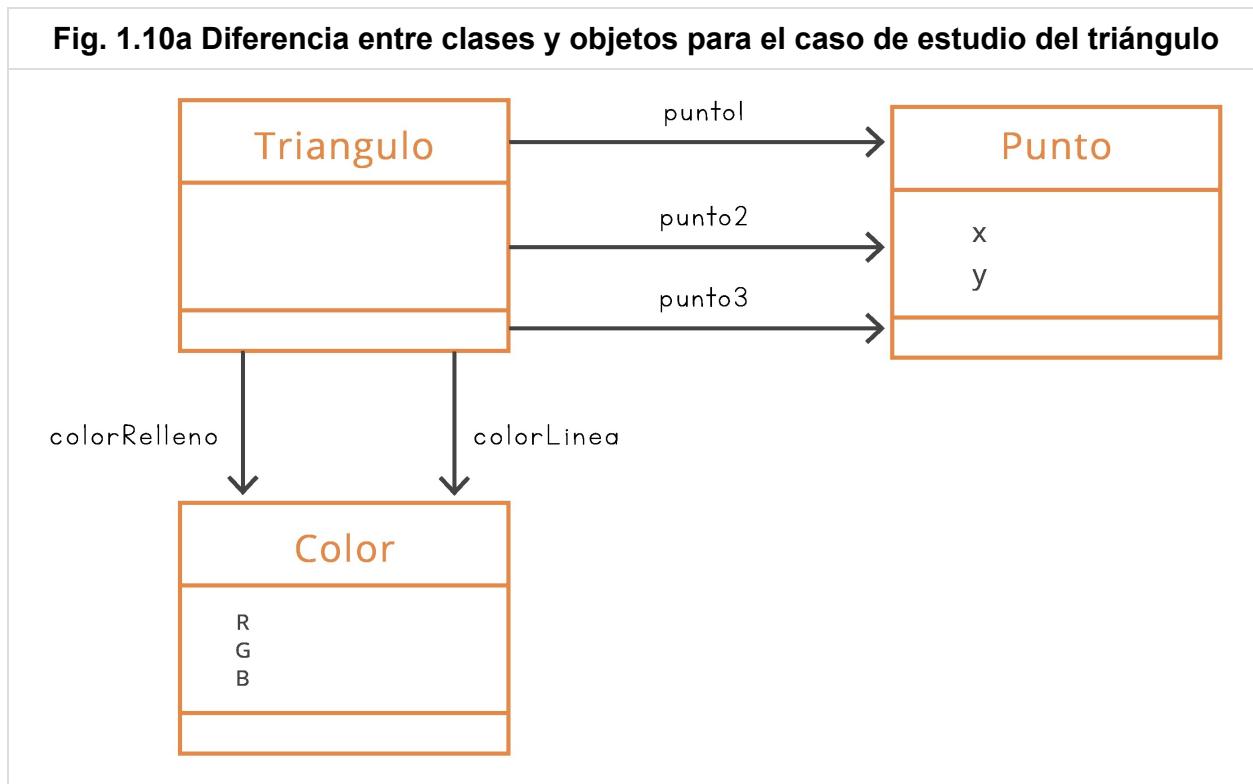
imagine que los algoritmos están dentro de las clases, y que son estas últimas las que establecen la manera en que los algoritmos colaboran para resolver el problema global (ver [figura 1.9](#)). Esta visión la iremos refinando a medida que avancemos en el libro, pero por ahora es suficiente para comenzar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

**Fig. 1.9 Visión intuitiva de la estructura de un programa**

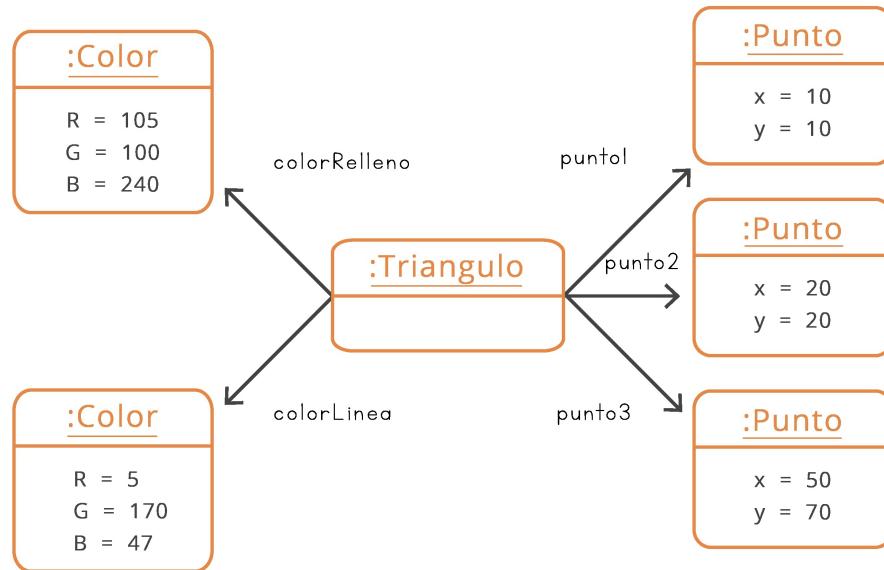


Hasta ahora es claro que en un programa hay una [clase](#) por cada entidad del mundo del problema. Pero, ¿qué pasa si hay varias "instancias" (es decir, varios ejemplares) de alguna de esas entidades? Piense por ejemplo que en vez de crear un programa para manejar un empleado, como en el primer caso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

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

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

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

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

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

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

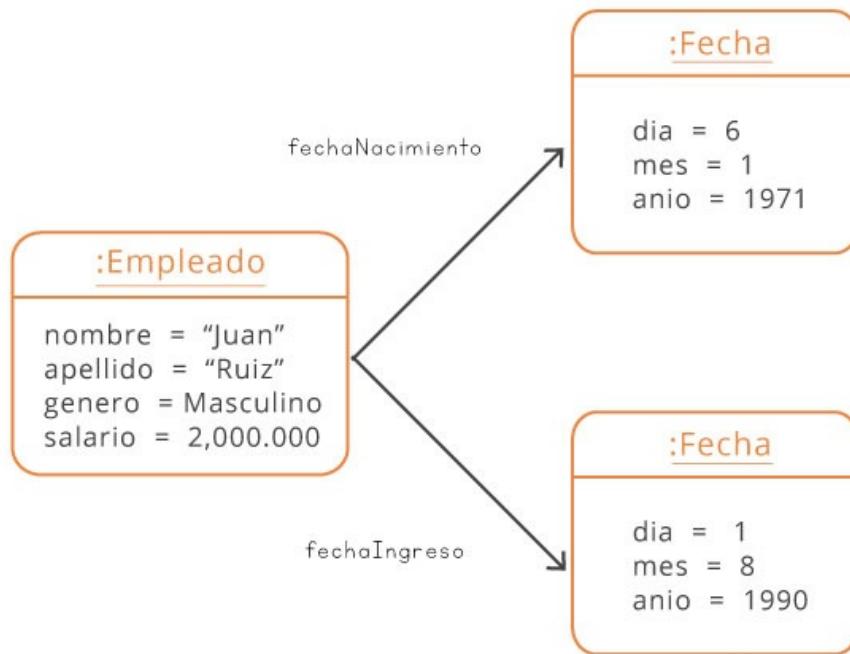
## Ejemplo 7

**Objetivo:** Ilustrar utilizando una extensión del [caso de estudio 1](#) la diferencia entre los conceptos de [clase](#) y [objeto](#).

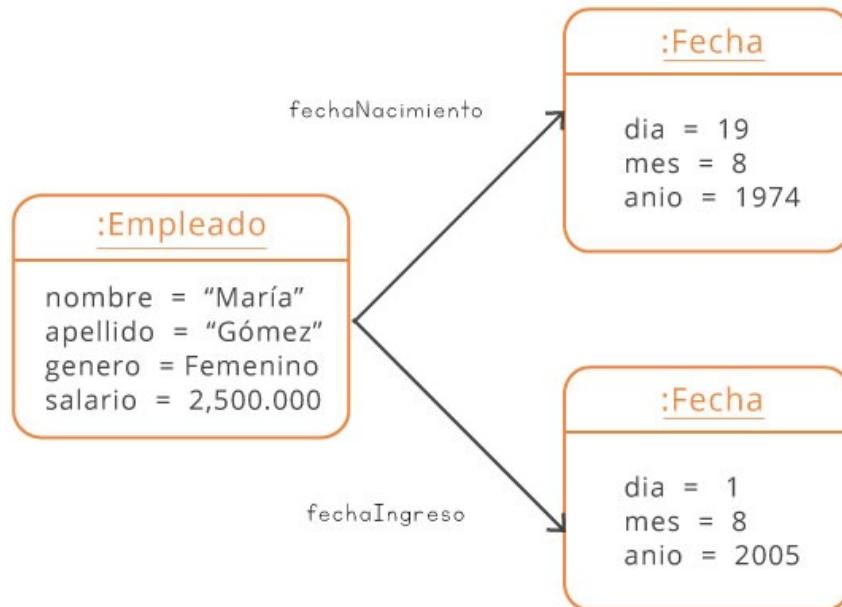
La extensión consiste en suponer que el programa debe manejar todos los empleados de una empresa, en lugar de uno solo de ellos.



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



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



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

## 6.3. Java como Lenguaje de Programación

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

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

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

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

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

## Ejemplo 8

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

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

**Archivo:** Empleado.java

**Clase:** Empleado

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

**Archivo:** Fecha.java

**Clase:** Fecha

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

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

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

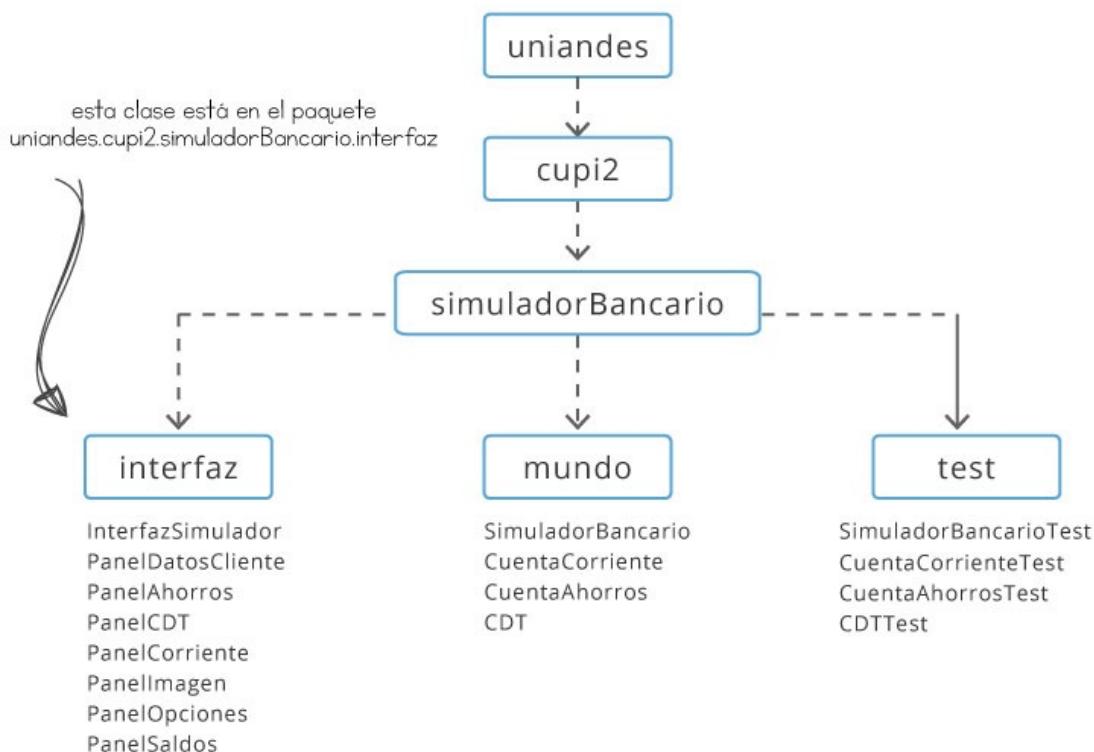
El programa del simulador bancario, por ejemplo, consta de 16 clases distribuidas de la siguiente manera:

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

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

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

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



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

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

```
package uniandes.cupi2.empleado;

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



}
```

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

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

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

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

## 6.4. Tipos de Datos

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

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

```
package uniandes.cupi2.empleado;

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

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

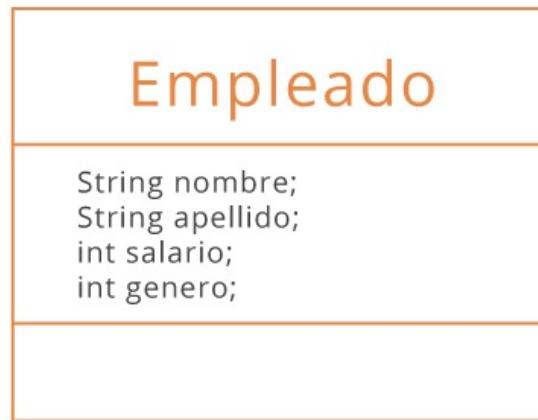
Para modelar el **atributo** "genero", debemos utilizar alguno de los tipos de datos con los que cuenta el lenguaje. Lo mejor en este caso es utilizar un **atributo** de tipo entero y usar la convención de que si dicho **atributo** tiene el valor 1 se está representando un empleado con género masculino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

```
public class Empleado
{
    ...
    /**
     * 1 = masculino, 2 = femenino
     */
    private int genero;
    ...
}
```

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

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

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

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

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

```
package uniandes.cupi2.empleado;

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

    private Fecha fechaNacimiento;
    private Fecha fechaIngreso;

}
```

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

En la [figura 1.13](#) aparece el diagrama de clases completo del caso de estudio del empleado.

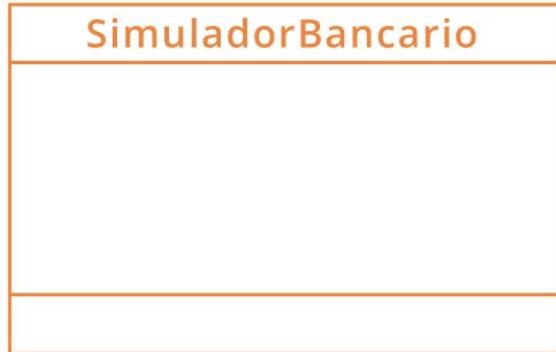
**Fig. 1.13 Representación de la clase Empleado en UML**



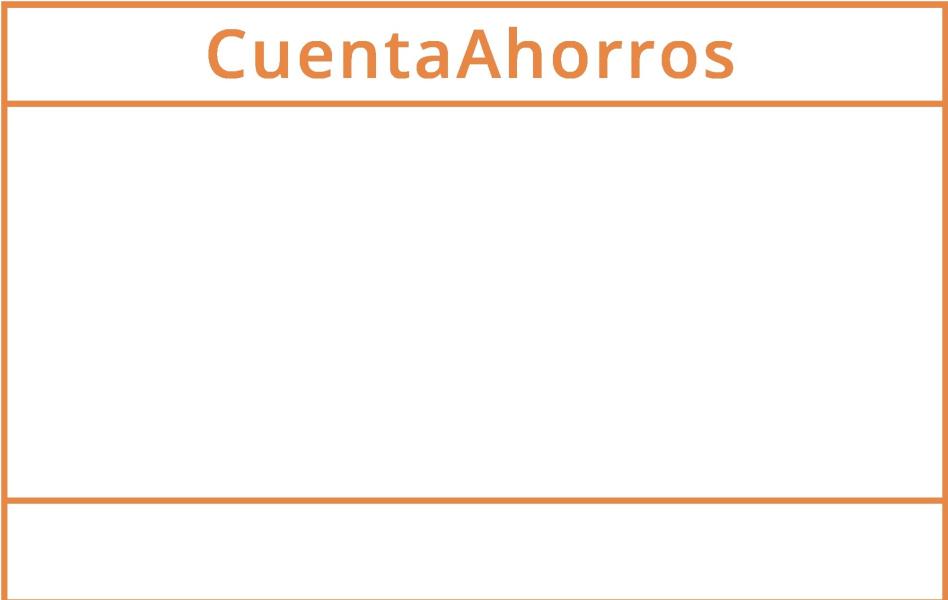
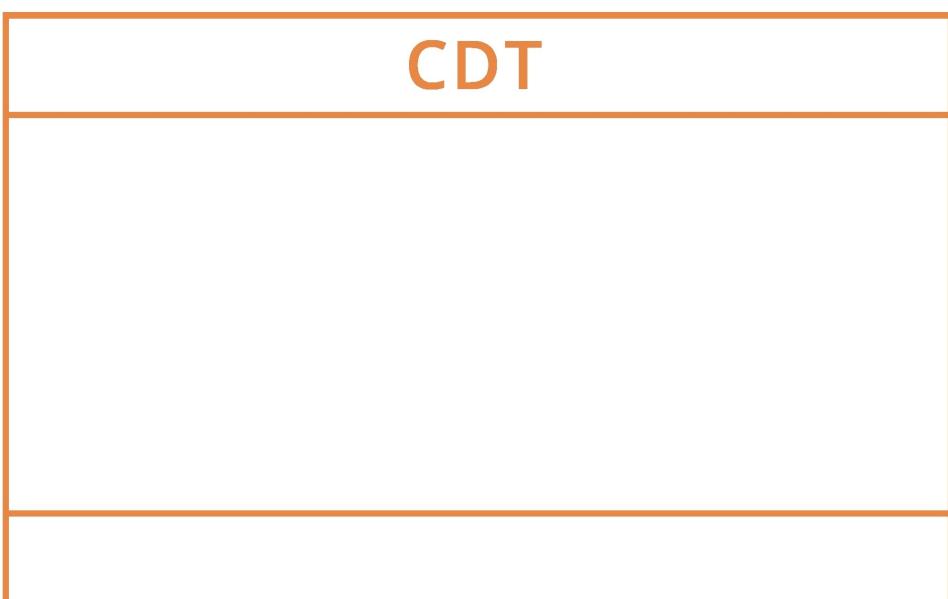
## Tarea 8

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

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

Declaración en Java	Descripción de la clase en UML
	 <p>SimuladorBancario</p> <pre>classDiagram SimuladorBancario</pre>

Declaración en Java	Descripción de la clase en UML
	 <p>CuentaCorriente</p> <pre>classDiagram CuentaCorriente</pre>

Declaración en Java	Descripción de la <a href="#">clase</a> en UML
	 <p>The diagram shows a UML class named "CuentaAhorros". It consists of three horizontal compartments. The top compartment contains the class name "CuentaAhorros". The middle compartment is empty. The bottom compartment is also empty.</p>
	 <p>The diagram shows a UML class named "CDT". It consists of three horizontal compartments. The top compartment contains the class name "CDT". The middle compartment is empty. The bottom compartment is also empty.</p>

## 6.5. Métodos

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

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

Un **método** está compuesto por cuatro elementos:

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

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

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

## Ejemplo 9

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

Usamos para esto el caso de estudio del empleado, con tres métodos sin cuerpo, suponiendo que cada uno debe resolver el problema que ahí mismo se describe. La declaración que aquí se muestra hace parte de la declaración de la **clase** (los métodos van después de la declaración de los atributos).

Se deja un cuarto **método** al final como tarea para el lector; en este caso, a partir de la descripción, debe determinar los parámetros, el retorno y la **signatura** del **método**.

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

**Nombre:** cambiarSalario

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

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

**Descripción:** cambia el salario del empleado, asignándole el valor que se entrega como [parámetro](#).

---

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

**Nombre:** darSalario

**Parámetros:** ninguno, puesto que con la información que ya tienen los objetos de la [clase](#) Empleado es posible resolver el problema.

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

**Descripción:** retorna el salario actual del empleado.

---

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

**Nombre:** calcularPrestaciones

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

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

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

---

**Nombre:** aumentarSalario

**Parámetros:**



**Retorno:**

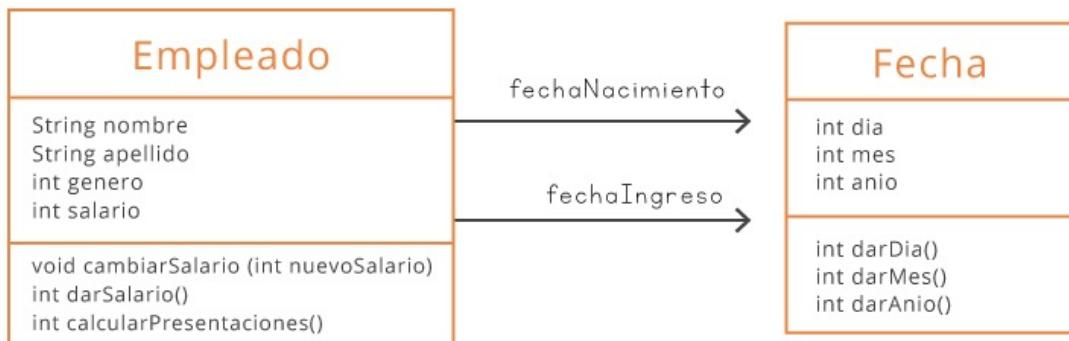


**Descripción:** aumenta el salario del empleado en un porcentaje que corresponde a la inflación anual del país.

---

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

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



## Tarea 9

**Objetivo:** Escribir y entender en Java la [signatura](#) de algunos métodos del caso de estudio del simulador bancario.

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

```

public void consignarValor( double pValor )
{
}

```

	<b>Nombre:</b>	
	<b>Parámetros:</b>	
	<b>Retorno:</b>	
	<b>Descripción:</b>	

<b>Nombre:</b>	<b>darSaldo</b>
Parámetros:	ninguno.
Retorno:	valor de tipo real.
Descripción:	retorna el saldo de la cuenta de ahorros.

**Signatura del Método:**



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

**Signatura del Método:**

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

**Signatura del Método:**



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

#### Signatura del Método:



## 6.6. La Instrucción de Retorno

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

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

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

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

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

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

## 6.7. La Instrucción de Asignación

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

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

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

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

```

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

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

```

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

## 6.8. La Instrucción de Llamada de un Método

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



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

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

```
public class Fecha
{
    ...
    public int darAnio( )
    {
        return anio;
    }
}
```

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

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

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

```

public class Empleado
{
    ...

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

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

```

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

## Ejemplo 10

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

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

```

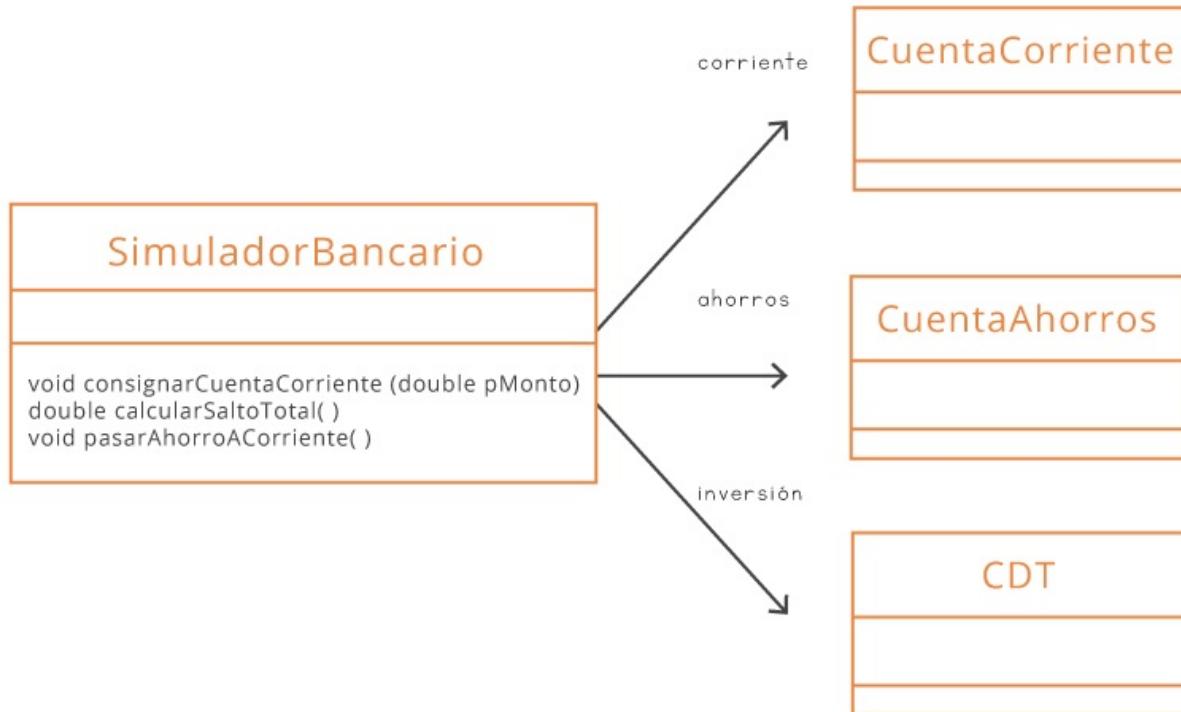
package uniandes.cupi2.simuladorBancario.mundo;

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

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

```

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



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

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

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

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

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

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

Si hay necesidad de convertir un valor real en un valor entero, se puede usar el **operador de conversión** `(int)`. Dicho **operador** se utiliza de la siguiente manera:

```
int respuesta = ( int )( 1000 / 33 );
```

En ese caso, el computador primero evalúa la **expresión** y luego elimina las cifras decimales.

## Tarea 10

**Objetivo:** Escribir el cuerpo de algunos métodos simples.

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

```
public void ahorrar( double pMonto )
{
}
```

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

```
public void retirarAhorro( double pMonto )
{
}
```

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

```
public double darSaldoCorriente( )
{
}
```

Retorna el saldo que hay en la cuenta corriente. No olvide que éste es un [método](#) de la [clase](#) CuentaBancaria.

```
public void retirarTodo( )
{
}
```

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

```
public void duplicarAhorro( )
{
}
```

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

```
public void avanzarMesSimulacion( )
{
}
```

Avanza un mes la simulación de la cuenta bancaria.

Dentro de un **método**:

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

## 6.9. Llamada de Métodos con Parámetros

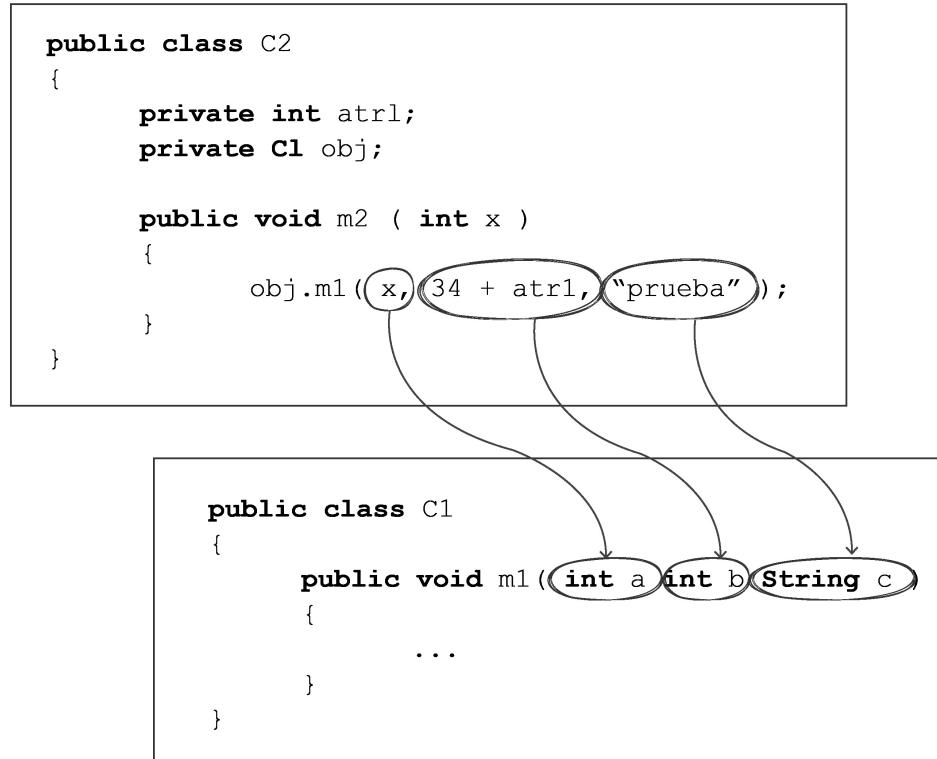
Este tema se profundizará en los capítulos posteriores. Por ahora sólo queremos dar una idea global del proceso de llamada de un **método** con parámetros. Para eso vamos a contestar siete preguntas:

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

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

- ¿Cómo se hace la relación entre esos valores y los parámetros? Los valores se deben pasar teniendo en cuenta el orden en el que se declararon los parámetros. Eso se ilustra en la [figura 1.15](#).
- ¿Qué sucede si se pasan más (o menos) valores que parámetros? El **compilador** informa que hay un error en la llamada. Lo mismo sucede si los tipos de datos de los valores no coinciden con los tipos de datos de los parámetros.

**Fig. 1.15 Llamada de un **método** con parámetros**



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

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

## 6.10. Creación de Objetos

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

- ¿Quién crea los objetos del modelo del mundo? Típicamente, el proceso lo inicia la **interfaz de usuario**, creando una instancia de la **clase** más importante del modelo. Lo que sigue, depende del **diseño** que se haya hecho del programa.
- ¿Cómo se guarda un **objeto** que acaba de ser creado? Más que guardar un **objeto** se debe hablar de referenciar. Una referencia a un **objeto** se puede guardar en cualquier **atributo** o **variable** del mismo tipo.

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

**Fig. 1.16 Creación de un objeto usando la instrucción new**

```
Punto p= new Punto ( );
```



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

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

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

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

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

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

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

El **método** constructor anterior le asigna un valor por defecto a cada uno de los atributos del **objeto**, evitando así tener valores indefinidos. El hecho de incluir este **método** constructor en la declaración de la **clase** hace que éste siempre se invoque como parte de la respuesta del computador a la instrucción `new`. En la figura 1.17 se ilustra la creación de un **objeto** de una **clase** que tiene un **método** constructor.

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

```
Punto p= new Punto ( );
```



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

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

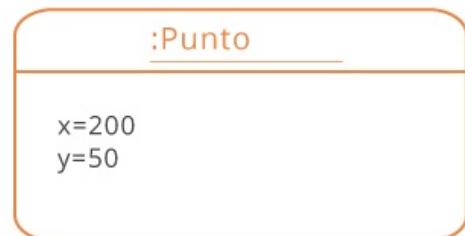
```
public Punto( double pX, double pY )  
{  
    x = pX;  
    y = pY;  
}
```

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

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

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

Punto p= **new** Punto (200,50);



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

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

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

## Tarea 11

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

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

```
public Color( )
{
}
```

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

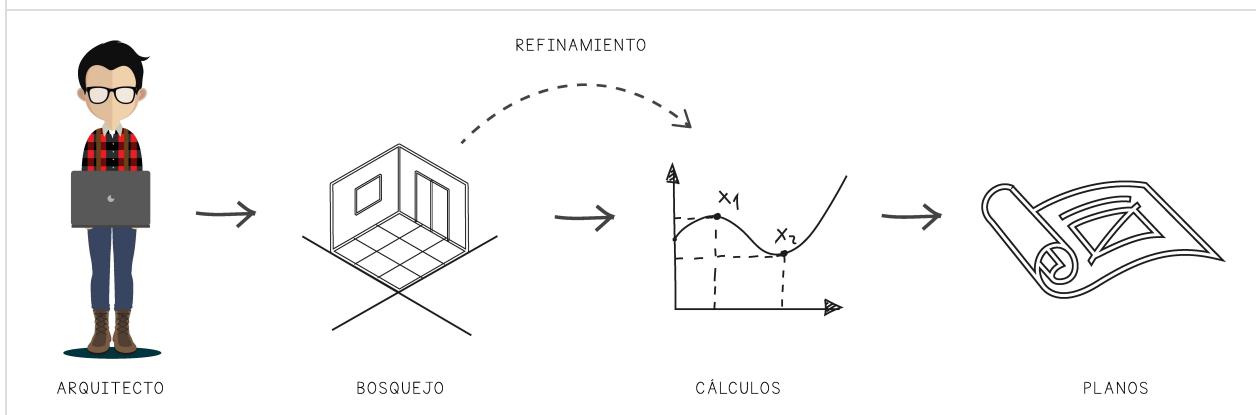
```
public Triangulo( )
{
    // Inicializa los puntos
    punto1 = new Punto( 200, 50 );
    punto2 = new Punto( 300, 200 );
    punto3 = new Punto( 100, 200 );
}
```

## 7. Diseño de la Solución

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

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

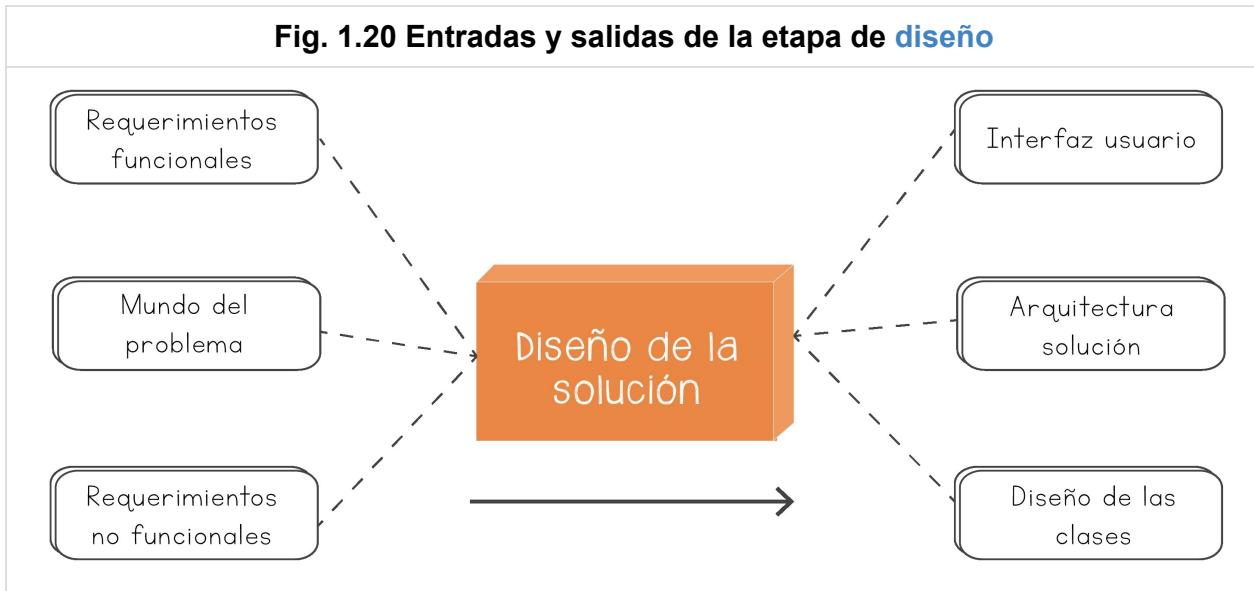
**Fig. 1.19 El diseño es una actividad iterativa hasta encontrar una solución**



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

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

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

**Fig. 1.20 Entradas y salidas de la etapa de diseño**

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

## 7.1. La Interfaz de Usuario

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

## 7.2. La Arquitectura de la Solución

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

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

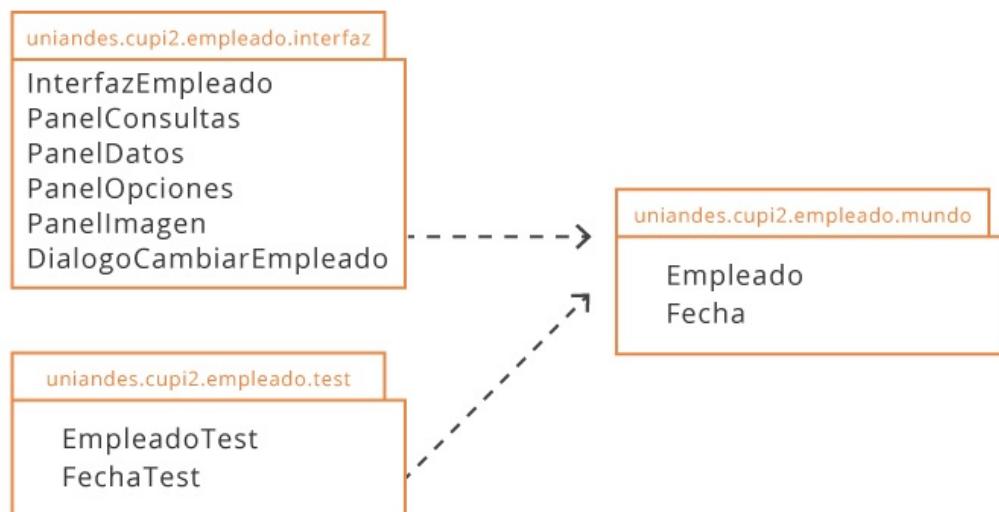
el caso de los problemas que tratamos en este libro, dado que son pequeños y su complejidad es baja, nos vamos a contentar con identificar los paquetes y las clases que van en cada uno de ellos. Luego, nos dedicaremos a trabajar en las clases de cada paquete, para finalmente armar la solución completa.

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

1. Las clases que implementan la [interfaz de usuario](#).
2. Las clases que implementan el modelo del mundo.
3. Las clases que implementan las pruebas.

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

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



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

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

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

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

## 7.3. El Diseño de las Clases

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

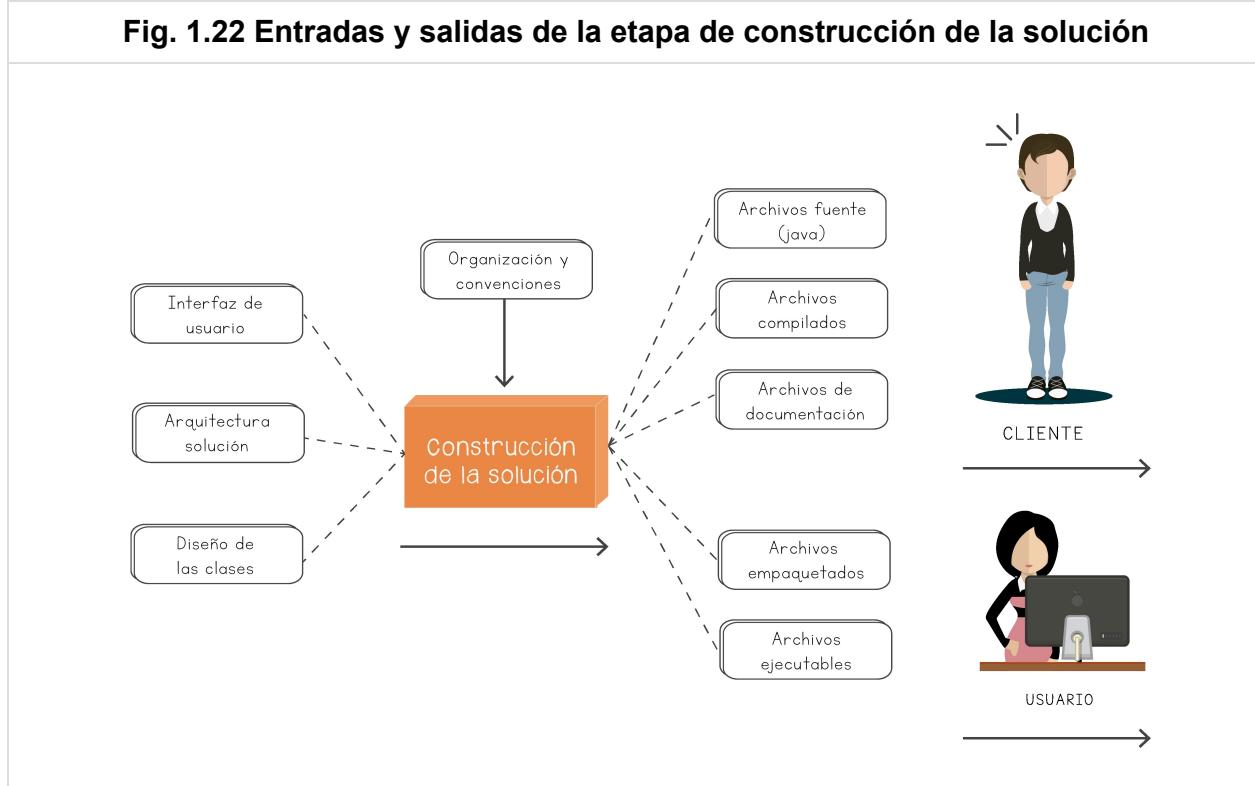
# 8. Construcción de la Solución

## 8.1. Visión Global

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

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

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



## 8.2. Tipos de Archivos

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

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

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

### 8.3. Organización de los Elementos de Trabajo

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

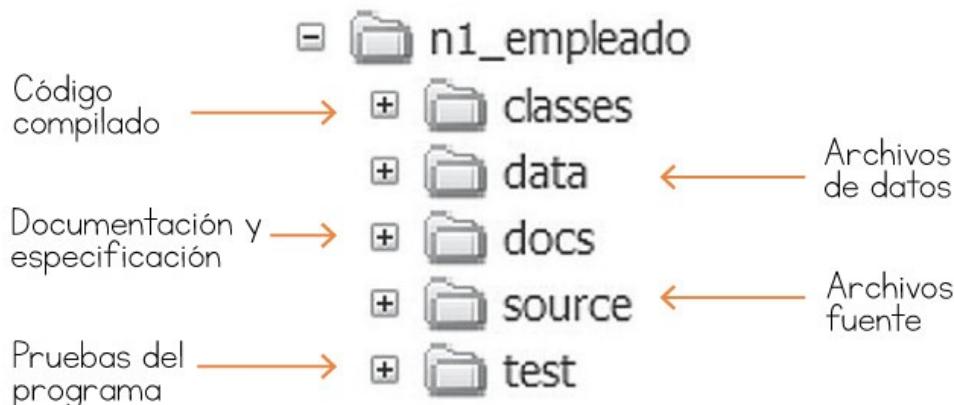
Para la construcción de un programa se sigue la misma idea: se define una organización (siempre debemos saber dónde buscar un elemento de la solución) y un conjunto de convenciones (por ejemplo, el [archivo](#) en el que están los requerimientos funcionales siempre se va a llamar de la misma manera).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

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

### 8.3.1 Proyectos y Directorios

Un proyecto de desarrollo va siempre en un directorio, cuyo nombre indica su contenido. En nuestro caso el nombre del directorio comienza por el nivel, seguido del nombre del caso de estudio (por ejemplo, n1\_empleado).

Dentro del directorio principal, se encuentran siete directorios, con el contenido que se muestra en la [figura 1.23](#).

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

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

## Tarea 12

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

Siga los pasos que se enuncian a continuación:

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

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

### 8.3.2. El Directorio source

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

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



## Tarea 13

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

Siga los pasos que se dan a continuación.

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

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

### 8.3.3. El Directorio `classes`

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

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

### 8.3.4. El Directorio `test`

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

### 8.3.5. El Directorio `docs`

En este directorio hay dos subdirectorios:

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

### 8.3.6. El Directorio `lib`

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

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

### 8.3.7 El Directorio data

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

## 8.4. Eclipse: Un Ambiente de Desarrollo

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

### Tarea 14

**Objetivo:** Estudiar tres funcionalidades básicas del [ambiente de desarrollo](#):

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

Siga los pasos que se enuncian a continuación.

#### 1. ¿Cómo abrir en Eclipse el programa n1\_empleado? Puede hacerlo de dos formas:

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

- Descomprima el [archivo](#) .zip que contiene el proyecto (por ejemplo en C:/temp/).
- Cree un proyecto Java en Eclipse (menú *File/New/Java Project*), con la ruta del directorio (C:/temp/n1\_empleado) y el nombre del proyecto (n1\_empleado).
- Puede aceptar la creación ahora (botón "*Finish*"), o navegar a la siguiente [ventana](#) ("*Next*") para ver las propiedades del proyecto.

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

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

- Seleccione la carpeta del proyecto (C:/temp/n1\_empleado) y finalice.

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

- Utilice la vista llamada navegador. Si la vista no está disponible, búscala en el menú *Window>Show View/ Navigator*.
- Revise la estructura de directorios del proyecto n1\_empleado y recuerde el contenido de cada uno de ellos (puede ocurrir que algunos directorios no contengan archivos en el proyecto que está explorando).

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

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

## 4. ¿Cómo editar una [clase](#) Java?

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

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

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

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

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

## 9. Hojas de Trabajo

### 9.1 Hoja de Trabajo N° 1: Una Encuesta

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

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

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

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

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

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

La [interfaz de usuario](#) de este programa es la que se muestra a continuación:

Encuesta del curso

# LA ENCUESTA



Agregar opinión a encuesta

## Bienvenido(a) a nuestra encuesta!

Por favor, seleccione su rango de edad:

0-17 años >>



Opciones

Opción 1      Opción 2

Encuesta del curso

# LA ENCUESTA

Agregar opinión a encuesta

## Bienvenido(a) a nuestra encuesta!

Su estado civil es:

Casado(a)

<< >>

Opciones

Opción 1 Opción 2

Encuesta del curso

# LA ENCUESTA



Agregar opinión a encuesta

## Bienvenido(a) a nuestra encuesta!

<< Califique de 0 a 10 el curso: 4 Enviar



0 5 10



Opciones

Opción 1	Opción 2
----------	----------

Encuesta del curso

# LA ENCUESTA



Agregar opinión a encuesta

## Bienvenido(a) a nuestra encuesta!

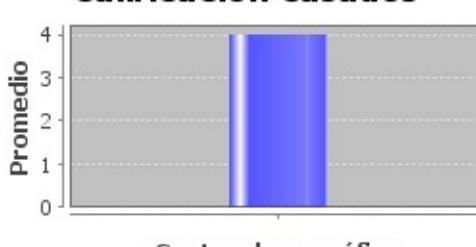
Consulta por sector demográfico

Rango de edad: 0-17 años    Estado civil: Casado(a)    Consultar

Estadísticas generales

Número total de opiniones: 1  
Promedio total encuesta: 4.00

**Calificación casados**



Sector demográfico	Promedio
0-17 años	0.00
18-54 años	4.00
55 o más	0.00

**Calificación solteros**



Sector demográfico	Promedio
0-17 años	0.00
18-54 años	0.00
55 o más	0.00

Responder nuevamente

Opciones

Opción 1    Opción 2

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

## Requerimiento Funcional 1

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 2

Nombre	
Resumen	
Entradas	
Resultado	

## Requerimiento Funcional 3

Nombre	
Resumen	
Entradas	
Resultado	

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

Entidad	Descripción

**Características de las entidades.** Identifique las características de cada una de las entidades y escriba la [clase](#) en UML con el [tipo de datos](#) adecuado.

## Entidad 1

Atributo	Valores Posibles

**Diagrama UML**

## Entidad 2

Atributo	Valores Posibles

**Diagrama UML**



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



**Métodos de las entidades.** Lea las siguientes descripciones de métodos y escriba su [implementación](#) en el lenguaje Java.

## Método 1

Clase	RangoEdad
Nombre	darNumeroCasados
Parámetros	Ninguno.
Retorno	El número de personas casadas que respondieron la encuesta, en el rango de edad de la <a href="#">clase</a> .
Descripción	Retorna el número de personas casadas que respondieron la encuesta, en el rango de edad de la <a href="#">clase</a> .

### Implementación en Java



## Método 2

Clase	RangoEdad
Nombre	darTotalOpinionCasados
Parámetros	Ninguno.
Retorno	La suma de todas las opiniones de los encuestados casados en el rango de edad de la <a href="#">clase</a> .
Descripción	Retorna la suma de todas las opiniones de los encuestados casados en el rango de edad de la <a href="#">clase</a> .

### Implementación en Java



## Método 3

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

### Implementación en Java



## Método 4

<b>Clase</b>	<b>RangoEdad</b>
Nombre	agregarOpinionCasado
Parámetros	Opinión del encuestado.
Retorno	Ninguno.
Descripción	Añade la opinión de una persona casada en el rango de edad que representa la <a href="#">clase</a> .

### **Implementación en Java**



## Método 5

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

### Implementación en Java



## Método 6

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

### Implementación en Java



## Método 7

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

### Implementación en Java



## Método 8

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

**Implementación en Java****Método 9**

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

**Implementación en Java**

## 9.2 Hoja de Trabajo N° 2: Una Alcancía

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

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

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

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

La [interfaz de usuario](#) de este programa es la que se muestra a continuación:



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

## Requerimiento Funcional 1

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

## Requerimiento Funcional 2

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

## Requerimiento Funcional 3

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

## Requerimiento Funcional 4

Nombre	R4 – Romper la alcancía.
Resumen	
Entradas	
Resultado	

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

Entidad	Descripción

**Características de las entidades.** Identifique las características de cada una de las entidades y escriba la [clase](#) en UML con el [tipo de datos](#) adecuado.

## Entidad 1

Atributo	Valores Posibles

**Diagrama UML**



**Métodos de las entidades.** Complete las siguientes descripciones de métodos y escriba su [implementación](#) en el lenguaje Java.

**Método 1**

Clase	Alcancia
Nombre	AgregarMoneda50
Parámetros	
Retorno	
Descripción	

#### Implementación en Java

A large, empty rectangular box with a thick orange border, intended for writing the Java implementation code for the method.

## Método 2

Clase	Alcancia
Nombre	AgregarMoneda500
Parámetros	
Retorno	
Descripción	

#### Implementación en Java



## Método 3

Clase	Alcancia
Nombre	darTotalDinero
Parámetros	
Retorno	
Descripción	

#### Implementación en Java

## Método 4

Clase	Alcancia
Nombre	darNúmeroMonedas100
Parámetros	
Retorno	
Descripción	

#### Implementación en Java

## Método 5

Clase	Alcancia
Nombre	romperAlcancia
Parámetros	
Retorno	
Descripción	

**Implementación en Java**

A large, empty rectangular box with a thick orange border, intended for writing the Java implementation code for the method.