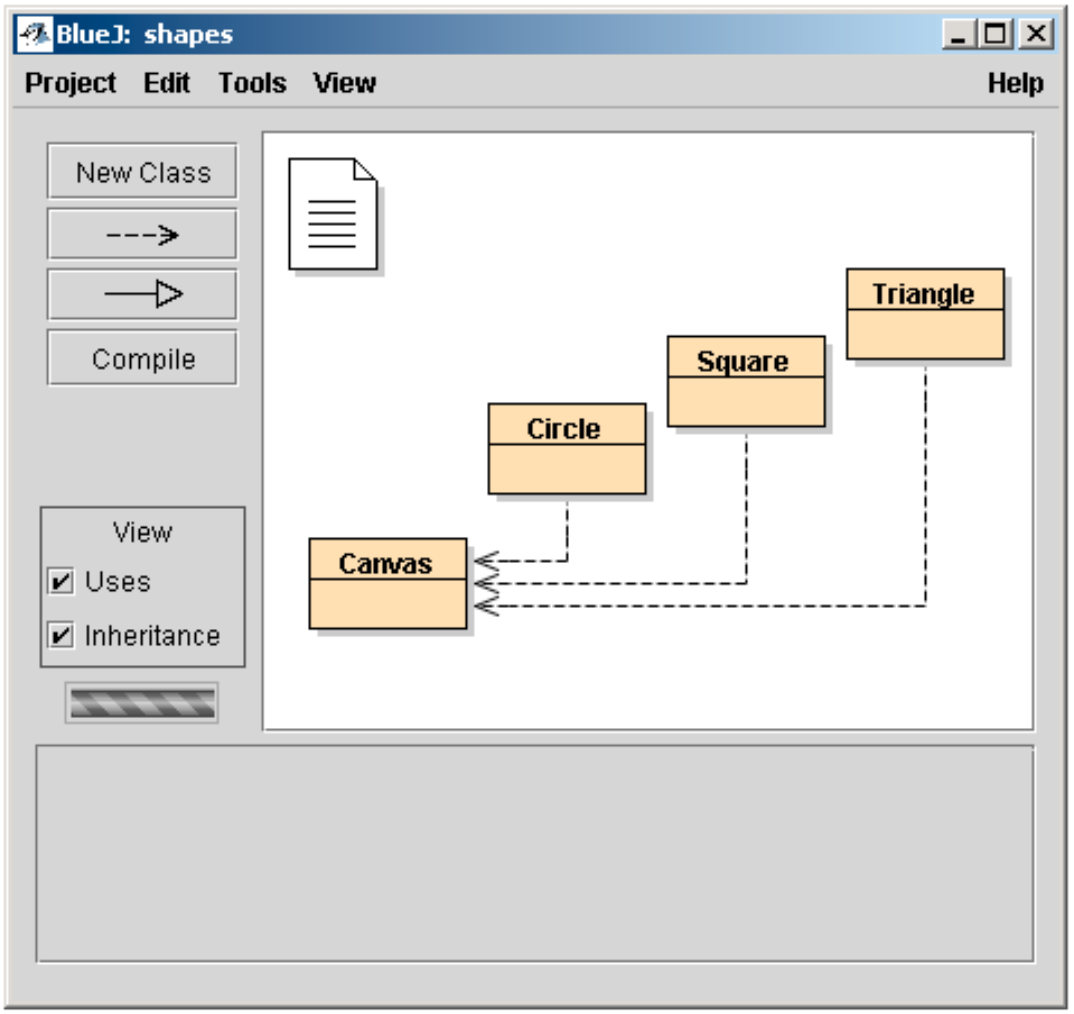
Centro Asociado Palma de Mallorca

Introducción Práctica de Programación Java

Antonio Rivero Cuesta

Sesión X

Entorno de Desarrollo BlueJ


BlueJ: shapes
Project
Edit
Tools
View
Help
New Class
Compile
View
Uses
Inheritance
Triangle
Square
Circle
Canvas

(Figura 1.1)

En esta ventana aparece un diagrama en el que cada uno de los rectángulos coloreados representa una clase en nuestro proyecto.

En este proyecto tenemos las clases de nombre Círculo, Cuadrado, Triangulo y Canvas.

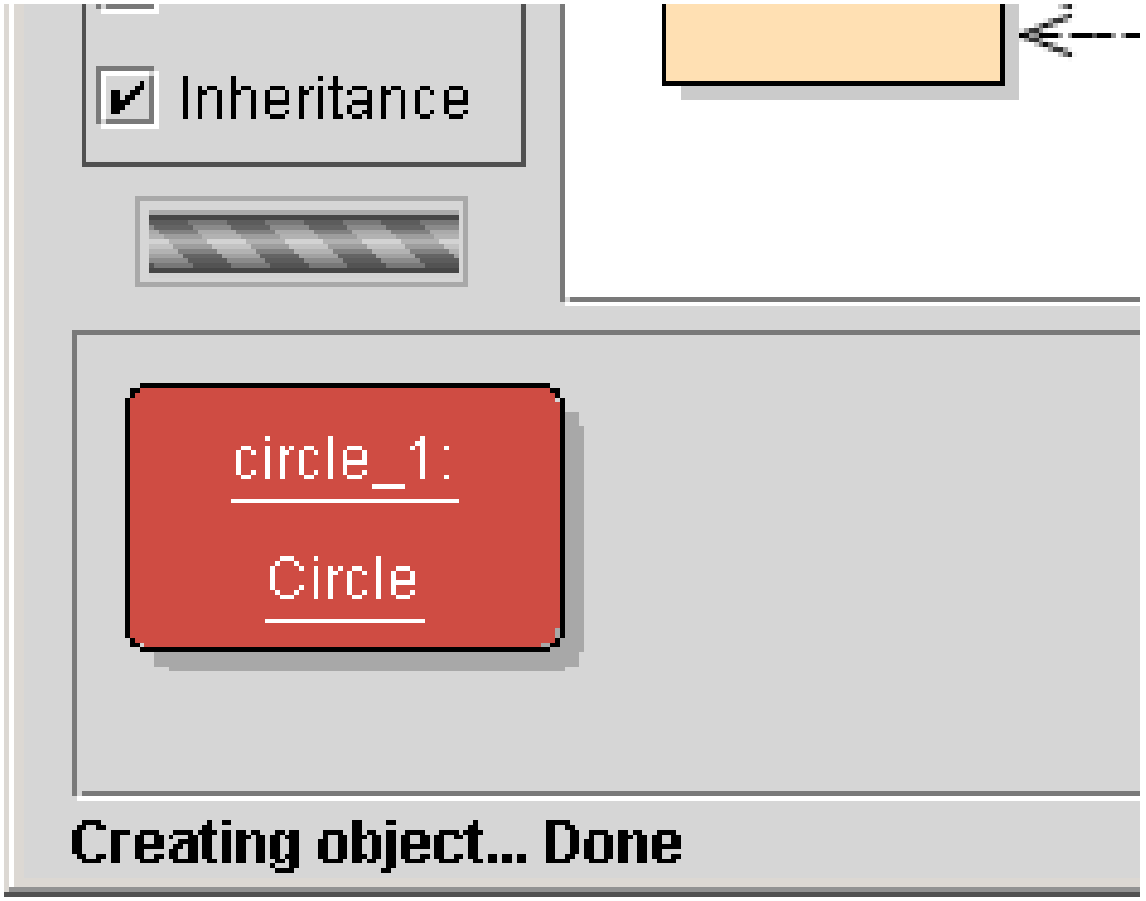
Haga c1ic con el botón derecho del ratón sobre la clase Círculo y seleccione el elemento

`new Circulo ( )`

del menú contextual.

El sistema solicita el «nombre de la instancia», haga c1ic en Ok ya que, por ahora, el nombre por defecto es suficientemente.

Se verá un rectángulo rojo ubicado en la parte inferior de la ventana, etiquetado con el nombre «circulo1»



(Figura 1.2)

Acabamos de crear el primer objeto

El icono rectangular «Círculo» de la Figura 1.1

Representa la clase Círculo mientras que circulo1 es un objeto creado a partir de esta clase.

La zona de la parte inferior de la ventana en la que se muestran los objetos se denomina ***banco de objetos***.

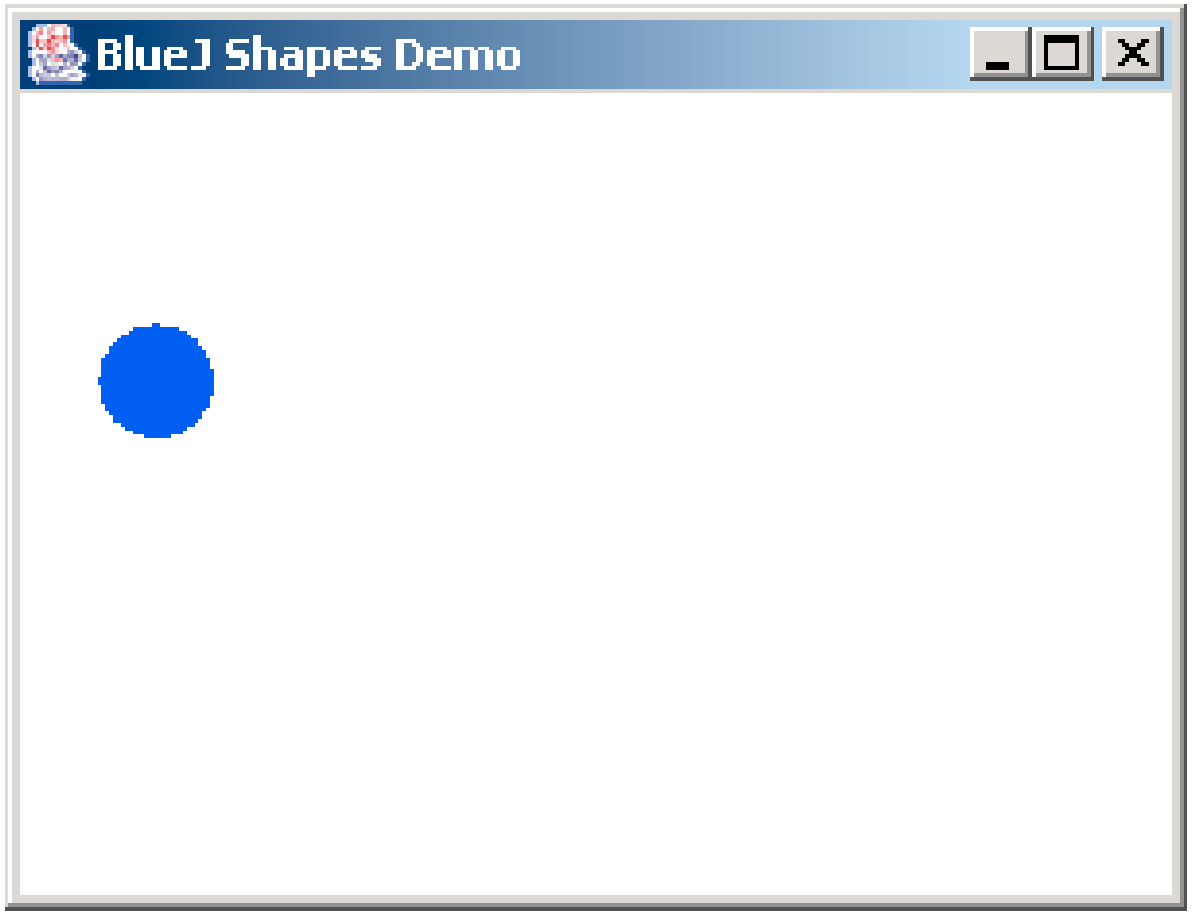
Los nombres de las clases comienzan con una letra mayúscula (como Círculo) y los nombres de los objetos con letras minúsculas (circulo1). Esto ayuda a distinguir de qué elemento estamos hablando.

Invocar métodos

Hacemos un c1ic derecho sobre un objeto círculo y se verá un menú contextual que contiene varias operaciones.

De este menú, seleccionar **`volverVisible`**;

Esta operación dibujará una representación de este círculo en una ventana independiente.



(Figura 1.3)

Observamos que hay varias operaciones en el menú contextual del círculo.

Invocamos un par de veces las operaciones **`moverDerecha`** y **`moverAbajo`** para desplazar al círculo más cerca del centro de la pantalla.

También podría probar **`volverlnvisible`** y **`volverVisible`** para ocultar y mostrar el círculo.

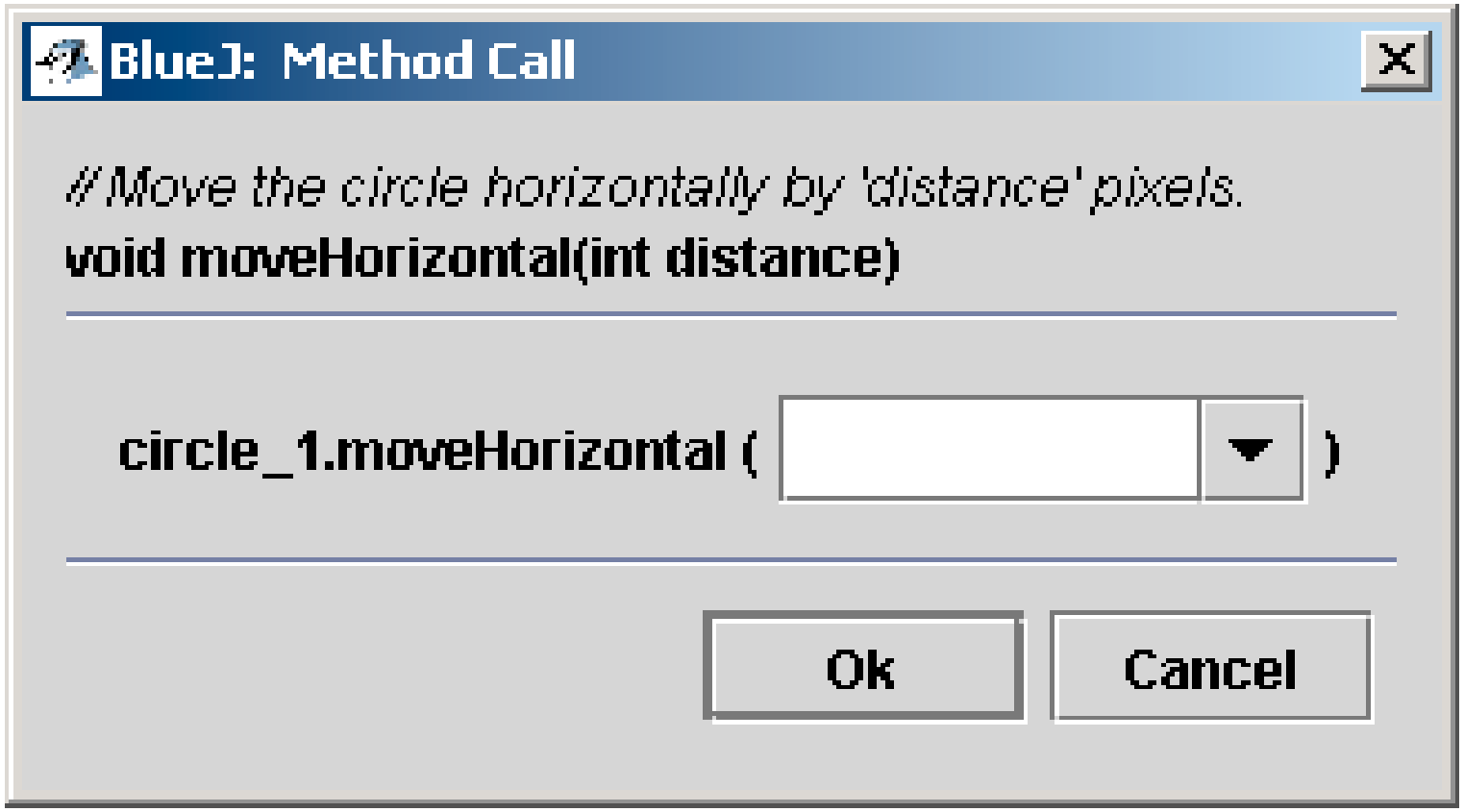
Parámetros

Ahora invocamos el método **`moverHorizontal`**.

Aparecerá una caja de diálogo que le solicita que ingrese algunos datos (Figura 1.4).

Ingrese el número 50 y haga c1ic en Ok.

Verá que el círculo se mueve 50 píxeles hacia la derecha.


BlueJ: Method Call
// Move the circle horizontally by 'distance' pixels.
void moveHorizontal(int distance)
circle_1.moveHorizontal (
)
Ok
Cancel

(Figura 1.4)

El método **`moverHorizontal`** que se acaba de nombrar, está escrito de modo tal que requiere información adicional para ejecutarse.

En este caso, la información requerida es la distancia y esto hace que el método **`moverHorizontal`** sea más flexible que los métodos **`moverDerecha`** o **`moverlzquierda`**.

Los últimos métodos mueven siempre al círculo una distancia determinada mientras que **`moverHorizontal`** permite especificar cuánto se quiere mover.

Los ***valores adicionales*** que requieren algunos métodos se denominan ***parámetros***.

Un método indica el tipo de parámetros que requiere.

Cuando invoca al método **`moverHorizontal`** tal como muestra la Figura l.4, la caja de diálogo muestra en su parte superior la línea

`void moverHorizontal (int distancia)`

Esta línea se denomina ***signatura del método***.

La signatura proporciona algo de información sobre el método en cuestión.

La parte comprendida entre paréntesis:

`(int distancia)`

Es la información sobre el parámetro requerido.

Para cada parámetro se define un ***tipo*** y un ***nombre***.

La signatura anterior establece que el método requiere un parámetro de tipo **`int`** y de nombre distancia.

El nombre ofrece alguna pista sobre el significado del dato esperado.

Tipos de dato

Un tipo especifica la naturaleza del dato que debe pasarse a un parámetro.

El tipo **`int`** significa números enteros (en inglés, «*integer numbers*», de aquí su abreviatura «**`int`**»).

Debemos de conocer la sintaxis de Java para poder programar.

Estado

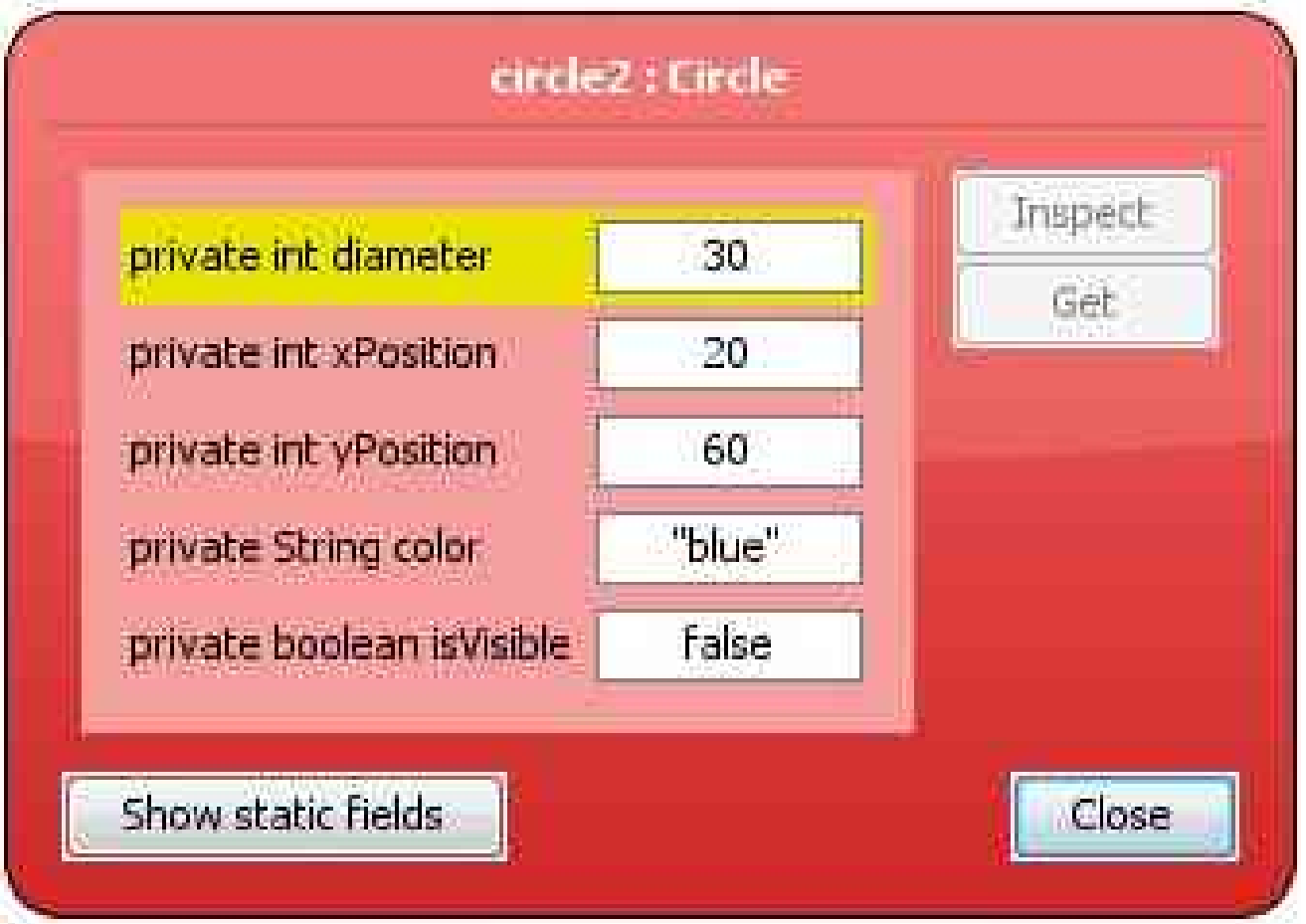
El concepto ***estado*** hace referencia al conjunto de valores de todos los atributos que definen un objeto tales como:

- las posiciones *x* e *y*
- el color
- el diámetro
- el estado de visibilidad para un círculo

Este es otro ejemplo de terminología común que usaremos de aquí en adelante.

En BlueJ, el estado de un objeto se puede inspeccionar seleccionando la función **`Inspect`** del menú contextual del objeto.

Cuando se inspecciona un objeto, se despliega una ventana similar a la que se muestra en la Figura 1.5 denominada ***Inspector del Objeto*** *(Object Inspector).*


circle2 : Circle
private int diameter
30
private int xPosition
20
private int yPosition
60
private String color
"blue"
private boolean isVisible
false
Inspect
Get
Show static fields
Close

(Figura 1.5)

Algunos métodos, cuando son llamados, cambian el estado de un objeto.

Por ejemplo, **`moverlzquierda`** modifica el atributo **`posicionX`**.

Java se refiere a los atributos de los objetos como ***campos*** *(fields)*.

Código fuente

Cada clase tiene algún ***código fuente*** asociado.

El código fuente es un texto que define los detalles de la clase.

En BlueJ, se puede visualizar el código fuente de una clase seleccionando la función *Open Editor* del menú contextual de la clase o haciendo doble c1ic en el icono de la clase.

Compilación y depuración

El compilador traduce el código Java a código máquina.

Podemos escribir en Java, ejecutar el compilador, que genera el código máquina, y la computadora puede entonces leer el código máquina.

Como resultado, cada vez que cambiamos el código debemos ejecutar el compilador antes de poder usar nuevamente la clase para crear un objeto.

Los ***errores de sintaxis*** son errores en la estructura del código propiamente dicho; son fáciles de solucionar porque el compilador los señala y muestra algún mensaje de error.

Los programadores más experimentados se concentran más en los ***errores de lógica***.

Un error de lógica ocurre cuando el programa compila y se ejecuta sin errores obvios pero da resultados incorrectos.

Los problemas de lógica son mucho más severos y difíciles de encontrar que los errores de sintaxis.

La escritura de programas lógicamente correctos es muy dificultosa para cualquier problema que no sea trivial y la prueba de que un programa es correcto, en general, no puede ser automática; en realidad, es tan difícil que es bien conocido el hecho de que la mayoría del software que se vende comercialmente contiene un número significativo de fallos.

En este apartado veremos varias actividades que están relacionadas con mejorar la exactitud de un programa que incluyen la ***prueba***, la ***depuración***.

La ***prueba*** es la actividad cuyo objetivo es determinar si una pieza de código, un método, una clase o un programa, produce el comportamiento pretendido.

La ***depuración*** viene a continuación de la prueba.

Si las pruebas demostraron que se presentó un error, usamos técnicas de depuración para encontrar exactamente dónde está ese error y corregirlo.

La ***prueba*** y la ***depuración*** son habilidades cruciales en el desarrollo de software.

Frecuentemente necesitará controlar sus programas para ver si tienen errores y luego, cuando ocurran, localizarlos en el código.

En las secciones que siguen analizaremos las siguientes técnicas de prueba y depuración:

- Pruebas de unidad en BlueJ
- Pruebas automatizadas
- Seguimiento manual
- Sentencias de impresión
- Depuradores

Pruebas de unidad en BlueJ

El término ***prueba de unidad*** se refiere a la prueba de partes individuales de una aplicación en contraposición con el término ***prueba de aplicación*** que es la prueba de una aplicación en su totalidad.

Las unidades que se prueban pueden ser de tamaños diversos.

Pueden ser un grupo de clases, una sola clase o simplemente un método.

Debemos observar que la prueba de unidad puede escribirse mucho antes de que una aplicación esté completa.

Puede probarse cualquier método, una vez que esté escrito y compilado.

La experimentación y prueba temprana conlleva varios beneficios.

En primer lugar, nos dan una experiencia valiosa con un sistema que hace posible localizar problemas tempranamente para corregirlos, a un costo mucho menor que si se hubieran encontrado en una etapa más avanzada del desarrollo.

En segundo término, podemos comenzar por construir una serie de casos de prueba y resultados que pueden usarse una y otra vez a medida que el sistema crece.

En una aplicación, cuando tenemos que decidir qué parte probar, distinguimos los casos de:

- *Pruebas positivas*.
- *Pruebas negativas*.

Una ***prueba positiva*** es la prueba de la funcionalidad que esperamos que realmente funcione.

Una ***prueba negativa*** es la prueba de aquellos casos que esperamos que fallen

Vamos a hacer una serie de pruebas con el proyecto

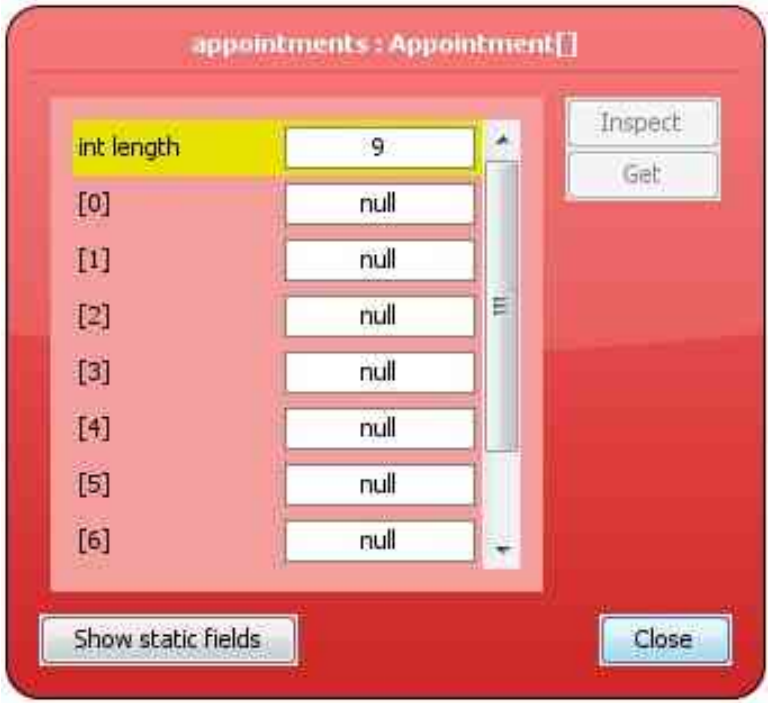
diary-prototype.

El código de este proyecto lo tenéis en el cd que viene con el libro.

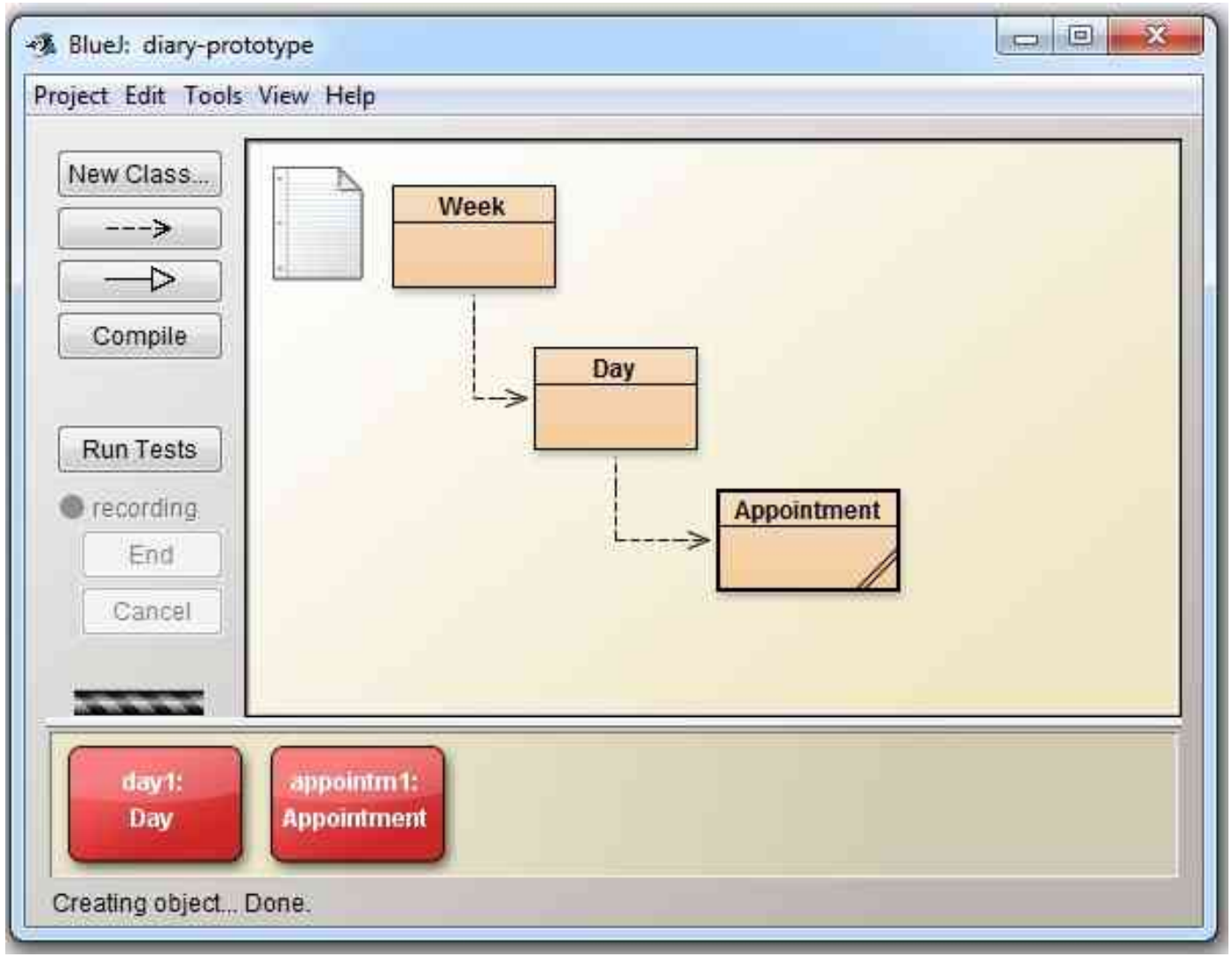
Creamos un objeto Día

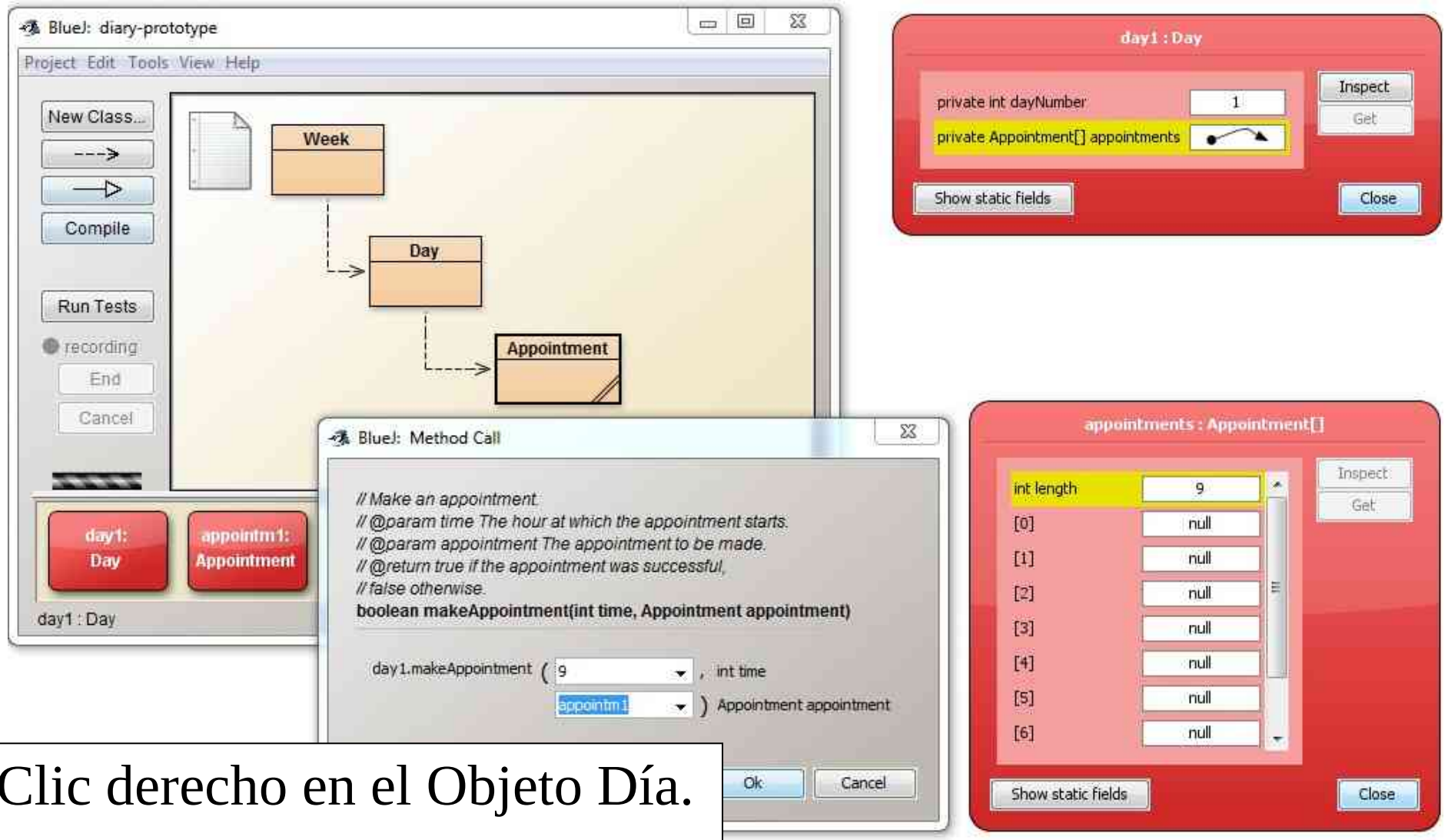
Hacemos clic derecho y abrimos el Inspector

Hacemos doble clic en el array del inspector



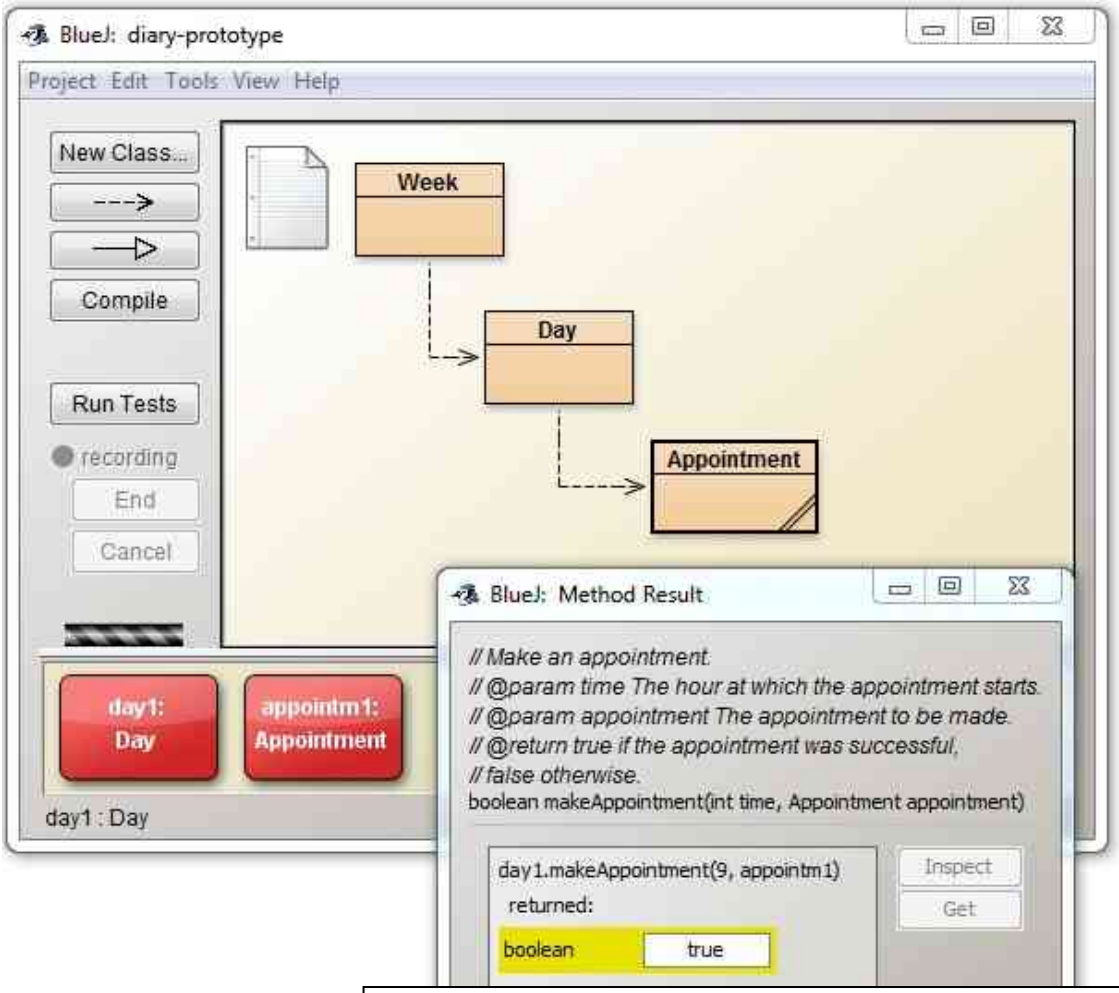
Creamos un objeto Cita


BlueJ: diary-prototype
Project Edit Tools View Help
New Class...
Compile
Run Tests
recording
End
Cancel
Week
Day
Appointment
day1:
Day
appointm1:
Appointment
Creating object... Done.



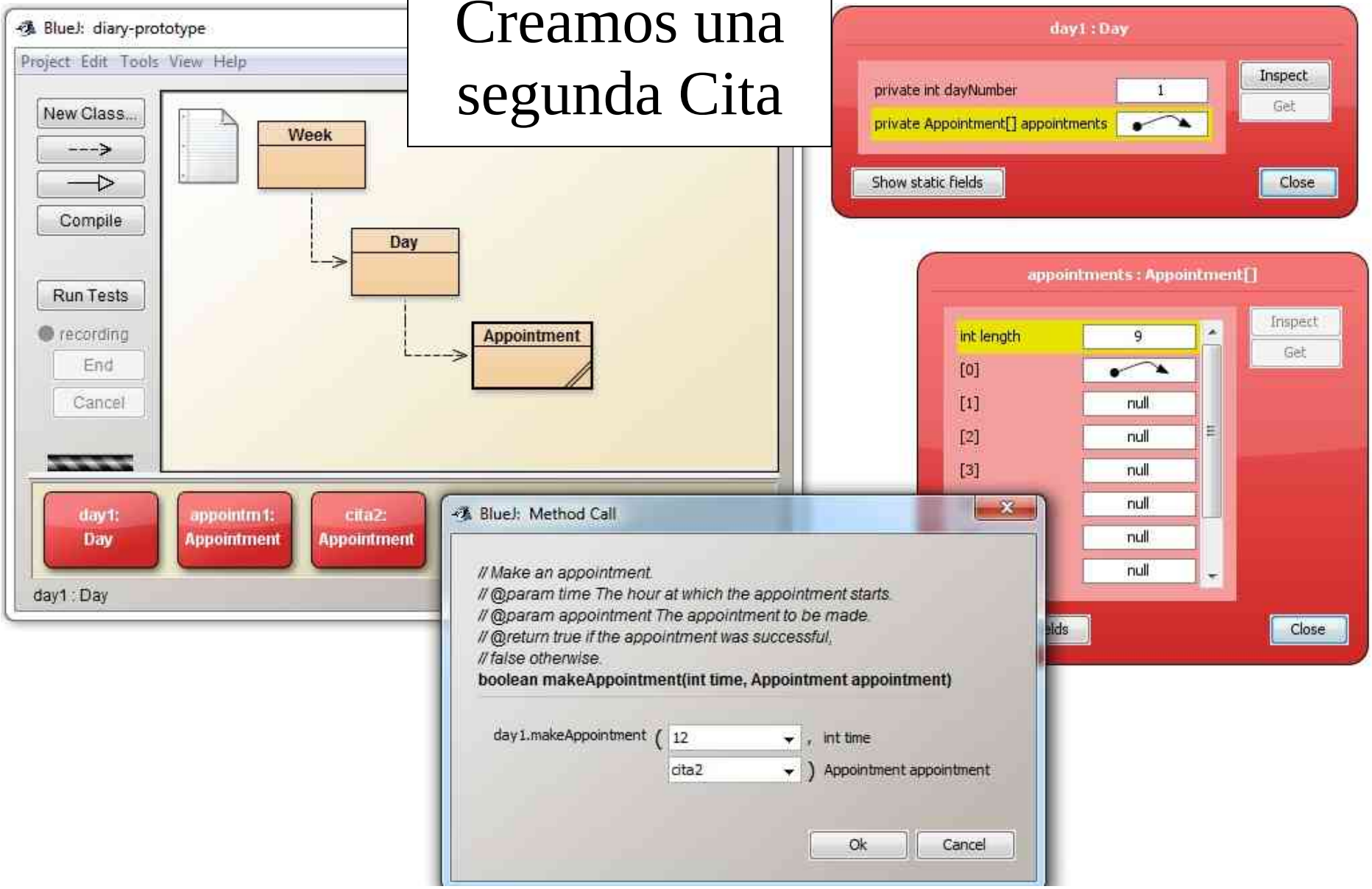
Clic derecho en el Objeto Día.

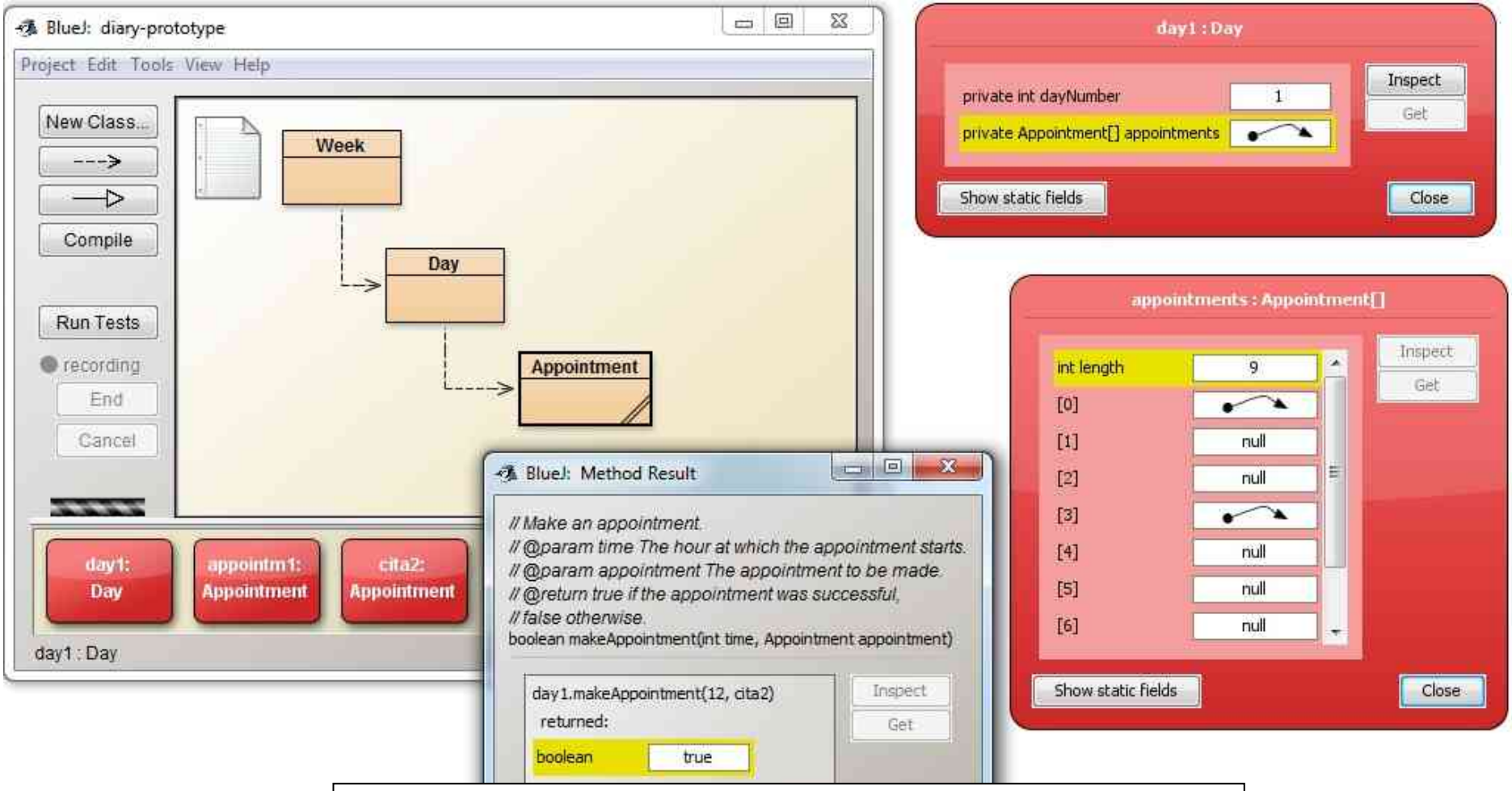
Invocamos el método crear cita



Como la cita es correcta, obtenemos ***true*** y vemos como cambia la primera posición del vector

Creamos una segunda Cita





Como la cita es correcta, obtenemos ***true*** y vemos como cambia la cuarta posición del vector

Pruebas automatizadas

Existen técnicas disponibles que nos permiten automatizar las pruebas repetitivas y así eliminar el trabajo pesado asociado que traen aparejadas.

Veremos las pruebas en el contexto de una ***prueba de regresión***.

Las ***pruebas de regresión*** consisten en ejecutar nuevamente las pruebas pasadas previamente para asegurarse de que la nueva versión aún las pasa.

Probablemente, estas pruebas son mucho más realizables cuando se las puede automatizar de alguna manera.

Una de las formas más fáciles de automatizar las pruebas de regresión es escribir un programa que actúa como un ***equipo de pruebas*** o una ***batería de pruebas***.

El proyecto ***diary-testing*** proporciona una ilustración de la manera en que podemos comenzar a construir un equipo de prueba para aquellas pruebas.

Las ***pruebas de regresión automáticas*** podrían ser más efectivas si pudiéramos construir pruebas que se autocontrolen y que requieran la intervención humana sólo cuando el resultado de una o más de ellas indiquen un posible problema.

El proyecto ***diary-testing-junit-v1*** representa un paso significativo en esta dirección.

Depuradores

Algunas veces resulta beneficioso usar herramientas adicionales que nos ayudan a comprender más profundamente cómo se ejecuta un programa.

Una de estas herramientas es el ***depurador*** (***debugger***).

El depurador es una herramienta de software que ayuda a examinar cómo se ejecuta una aplicación.

Es un programa que permite que los programadores ejecuten una aplicación paso a paso.

Ofrece funciones para detener y comenzar la ejecución de un programa en un punto seleccionado del código y para examinar los valores de las variables.

Fijamos puntos de interrupción en las sentencias en donde queremos comenzar nuestro seguimiento y luego usamos las funciones ***Step*** y ***Step Into*** para llevarlo a cabo.

Una de las ventajas es que el depurador automáticamente tiene el cuidado de mantener el trazo del estado de cada objeto y al hacer esto, es más rápido y produce menos errores que cuando hacemos lo mismo manualmente.

Una desventaja de los depuradores es que no mantienen registro permanente de los cambios de estado por lo que resulta difícil volver atrás y controlar el estado en que estaba unas cuantas sentencias antes.

Un depurador ofrece información sobre la ***secuencia de llamadas*** en cada momento.

La secuencia de llamadas muestra:

- el nombre del método que contiene la sentencia actual
- el nombre del método desde donde fue llamado
- el nombre del método que fue llamado, etc.

La secuencia de llamadas contiene un registro de todos los métodos activos y aún no terminados, de manera similar a la que hemos hecho manualmente durante nuestro seguimiento escribiendo marcas próximas a las sentencias de invocación de métodos.

En BlueJ, la secuencia de llamadas se muestra en la parte izquierda de la ventana del depurador.

Cada nombre de método en dicha secuencia puede ser seleccionado para inspeccionar los valores actuales de las variables locales de dicho método.

Ejemplo del sistema de correo electrónico

La idea de este proyecto es simular las acciones de usuarios que se envían correos electrónicos entre ellos.

Un usuario utiliza un cliente de correo para enviar mensajes a un servidor que se encarga de despacharlos al cliente de correo de otro usuario.

Vamos a hacer una prueba con el depurador:

Creamos un objeto **`ServidorDeCorreo`**. Pág 80.

En el momento de crear un cliente necesitaremos aportar la instancia de **`ServidorDeCorreo`** como un parámetro, utilizaremos el que hemos creado al principio.

Luego creamos un objeto **`ClienteDeCorreo`** para cada uno de los usuarios.

Por ejemplo: «Sofia» y «Juan», también podemos nombrar las instancias como «sofia» y «juan» para que las pueda distinguir en el banco de objetos.

Luego, enviamos un mensaje para Juan mediante el método **`enviarMensaje`** de Sofia.

Establecemos un ***punto de interrupción***.

Un punto de interrupción es una bandera que se adjunta a la línea de código en la que se detendrá la ejecución de un método cuando encuentre dicho punto.

En BlueJ, este punto de interrupción se representa mediante una pequeña señal de parada (icono de «stop») (Figura 3.5).

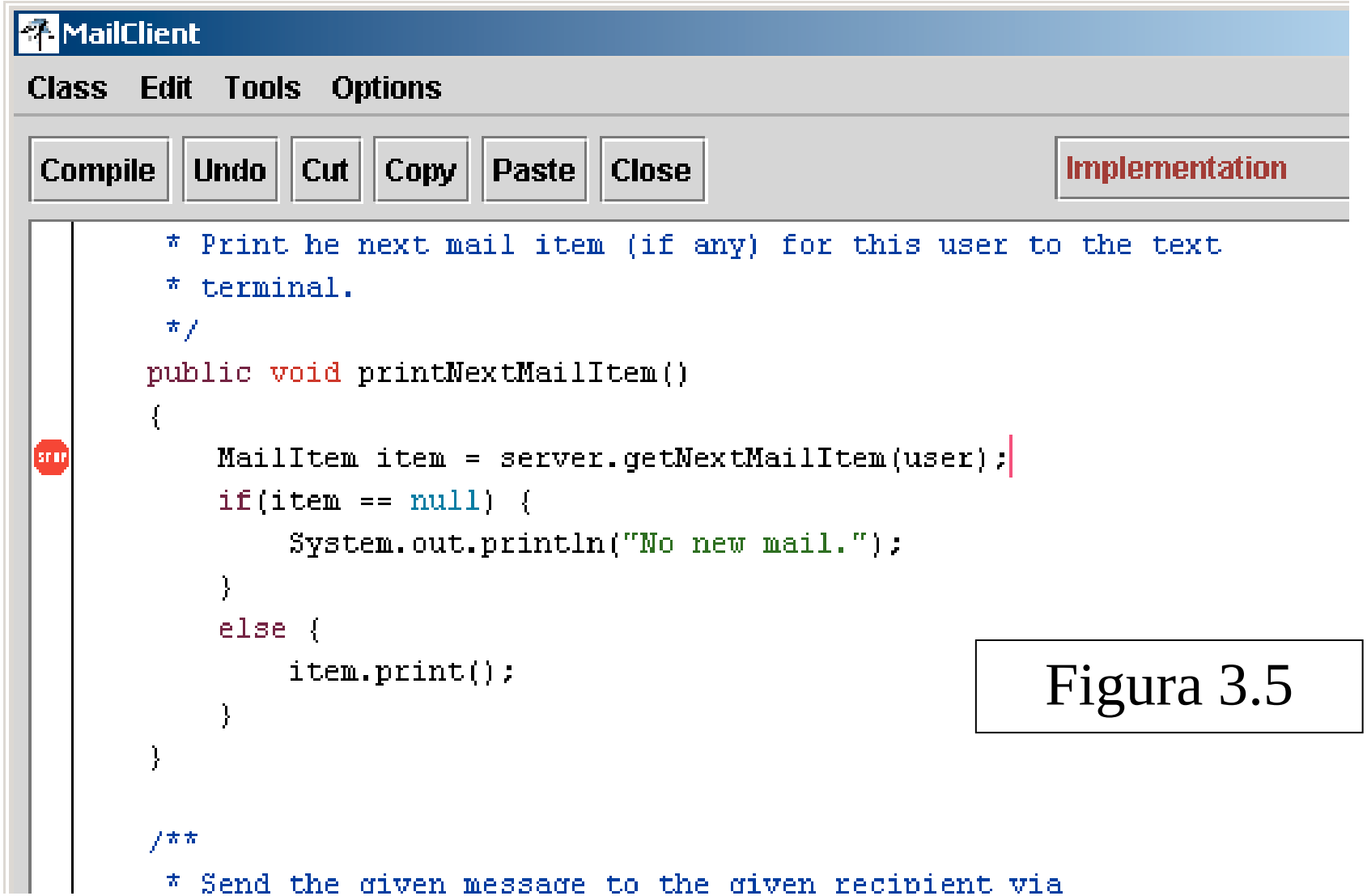

MailClient
Class Edit Tools Options
Compile Undo Cut Copy Paste Close
Implementation
* Print he next mail item (if any) for this user to the text
* terminal.
*/
public void printNextMailItem()
{
MailItem item = server.getNextMailItem(user);
if(item == null) {
System.out.println("No new mail.");
}
else {
item.print();
}
}
/**
* Send the given message to the given recipient via

Figura 3.5

Ponemos un punto de interrupción abriendo el editor de BlueJ.

Seleccionamos la línea apropiada y seleccionando ***Set Breakpoint*** del menú ***Tools***.

También podemos, hacer clic en la zona situada a la izquierda de las líneas de código en la que aparece el símbolo de parada para agregar o quitar puntos de interrupción.

La clase debe estar compilada para poder ubicar puntos de interrupción, además al compilar se eliminan los puntos establecidos.

Una vez establecido un punto de interrupción, se invoca el método **`imprimirMensajeSiguiente`** desde el cliente de correo de Juan.

Se abren la ventana del editor de la clase **`ClienteDeCorreo`** y la ventana del depurador (Figura 3.6).

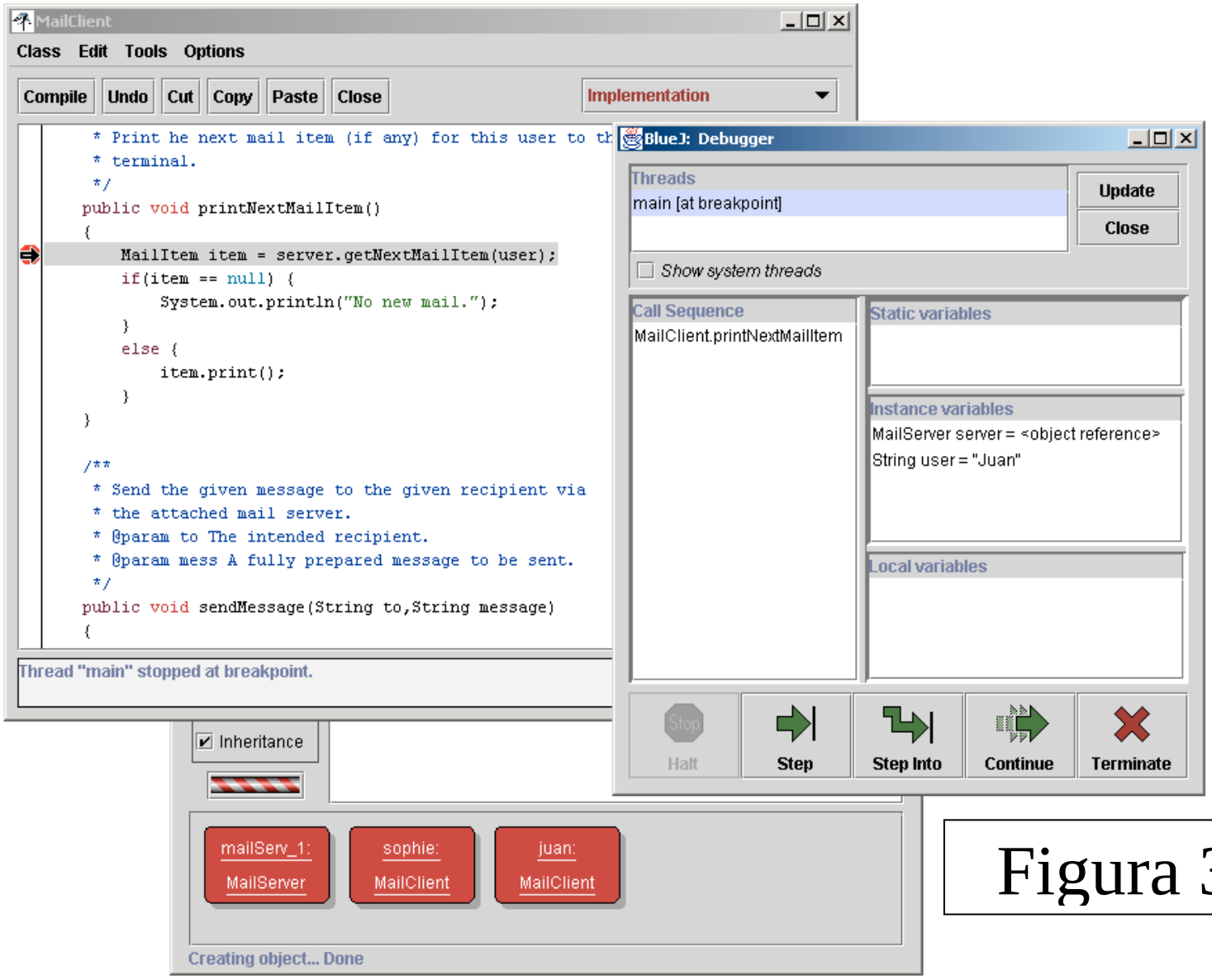

MailClient
Class Edit Tools Options
Compile Undo Cut Copy Paste Close
Implementation
* Print he next mail item (if any) for this user to th
* terminal.
*/
public void printNextMailItem()
{
MailItem item = server.getNextMailItem(user);
if(item == null) {
System.out.println("No new mail.");
}
else {
item.print();
}
}
/**
* Send the given message to the given recipient via
* the attached mail server.
* @param to The intended recipient.
* @param mess A fully prepared message to be sent.
*/
public void sendMessage(String to,String message)
{
Thread "main" stopped at breakpoint.
BlueJ: Debugger
Threads
main [at breakpoint]
Update
Close
Show system threads
Call Sequence
MailClient.printNextMailItem
Static variables
Instance variables
MailServer server = <object reference>
String user = "Juan"
Local variables
Halt
Step
Step Into
Continue
Terminate
Inheritance
mailServ_1: MailServer
sophie: MailClient
juan: MailClient
Creating object... Done

Figura 3.6

La ventana del depurador tiene tres áreas para mostrar las variables:

- ***static variables*** (variables estáticas),
- ***instance variables*** (variables de instancia)
- ***local variables*** (variables locales).

Este objeto tiene dos variables de instancia o campos:

- Servidor
- Usuario

También podemos ver sus valores actuales.

La variable ***usuario*** almacena la cadena «Sofía»

La variable ***servidor*** almacena una referencia a otro objeto.

La referencia a un objeto la hemos representado anteriormente, mediante una flecha en los diagramas de objetos.

Aún no hay ninguna variable local y se debe a que la ejecución del código se detuvo justo antes de la línea en la que se encuentra el punto de interrupción.

Dado que la línea con el punto de interrupción contiene la declaración de la única variable local y que esta línea aún no se ha ejecutado, no existen variables locales en este momento.

El depurador no sólo nos permite interrumpir la ejecución del programa e inspeccionar las variables sino que también podemos recorrer lentamente el código.

Cuando la ejecución se detiene en un punto de interrupción, al hacer clic sobre el botón ***Step*** se ejecuta una sola línea de código y luego se detiene nuevamente.

El resultado de ejecutar la primera línea del método **`imprimirMensajeSiguiente`** se muestra en la Figura 3.7.

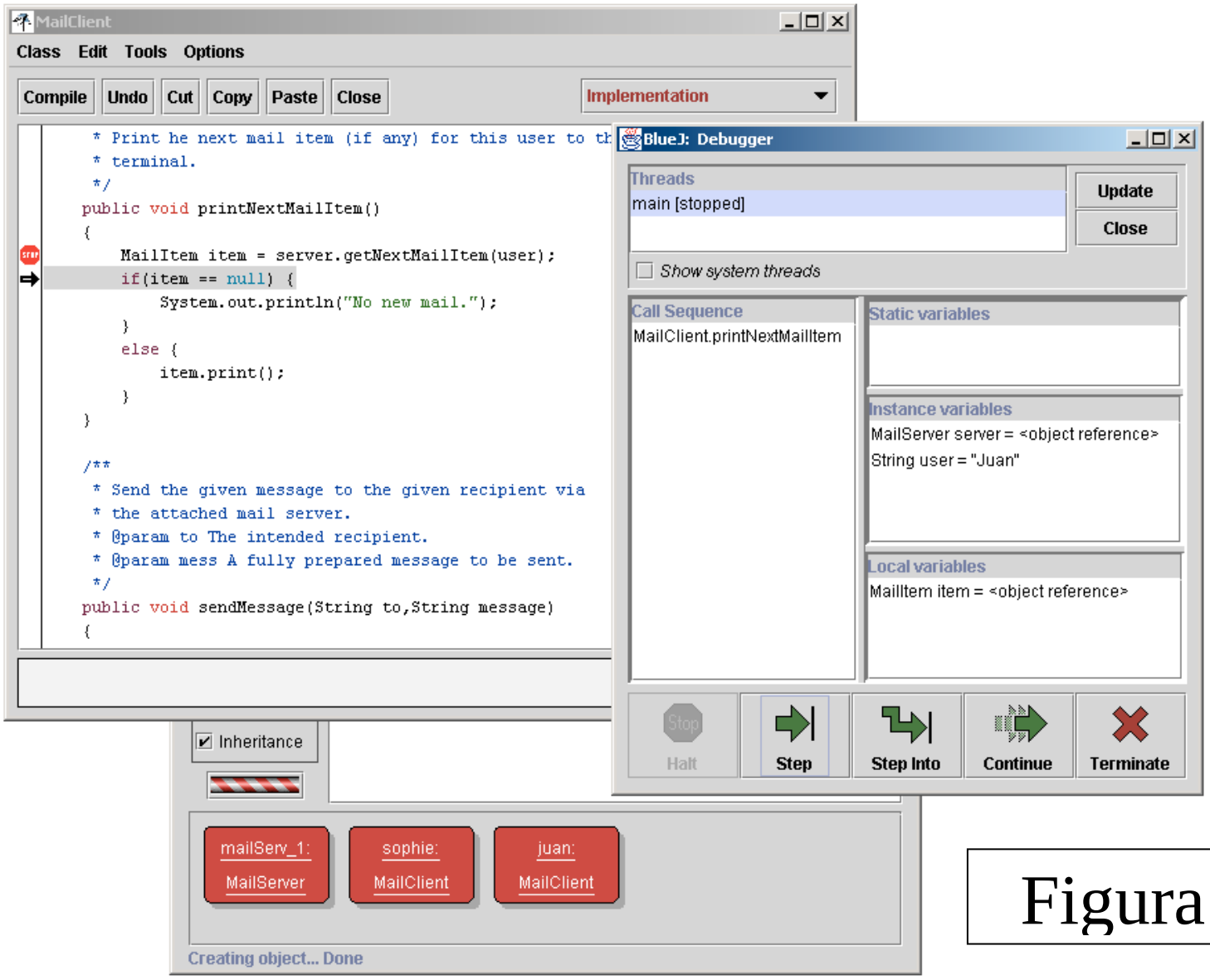

MailClient
Class Edit Tools Options
Compile Undo Cut Copy Paste Close
Implementation
* Print he next mail item (if any) for this user to th
* terminal.
*/
public void printNextMailItem()
{
MailItem item = server.getNextMailItem(user);
if(item == null) {
System.out.println("No new mail.");
}
else {
item.print();
}
}
/**
* Send the given message to the given recipient via
* the attached mail server.
* @param to The intended recipient.
* @param mess A fully prepared message to be sent.
*/
public void sendMessage(String to,String message)
{
BlueJ: Debugger
Threads
main [stopped]
Update
Close
Show system threads
Call Sequence
MailClient.printNextMailItem
Static variables
Instance variables
MailServer server = <object reference>
String user = "Juan"
Local variables
MailItem item = <object reference>
Stop
Halt
Step
Step Into
Continue
Terminate
Inheritance
mailServ_1:
MailServer
sophie:
MailClient
juan:
MailClient
Creating object... Done

Figura 3.7

Podemos ver que la ejecución se desplazó una sola línea.

Aparece una pequeña flecha negra cerca de la línea de código que indica la posición actual.

La lista de variables locales en la ventana del depurador indica que se ha creado una variable local y que se ha asignado un objeto a ella.

Ahora podemos usar el botón ***Step*** reiteradamente hasta el final del método, lo que nos permite visualizar la ruta que toma la ejecución.

Este recorrido paso a paso es especialmente interesante cuando hay sentencias condicionales.

Podemos ver claramente cómo se ejecuta una de las ramas de la sentencia condicional y visualizar si se satisfacen nuestras expectativas.