

Programación Orientada a Objetos

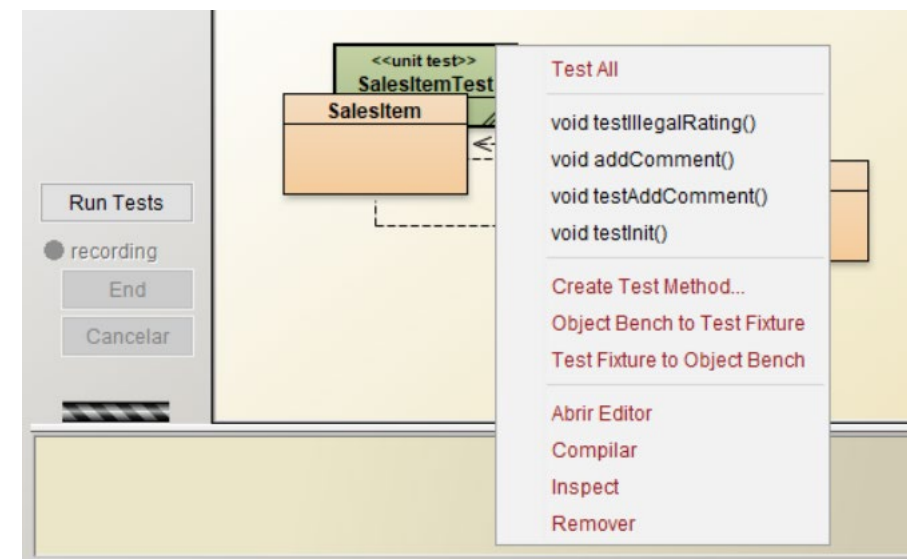
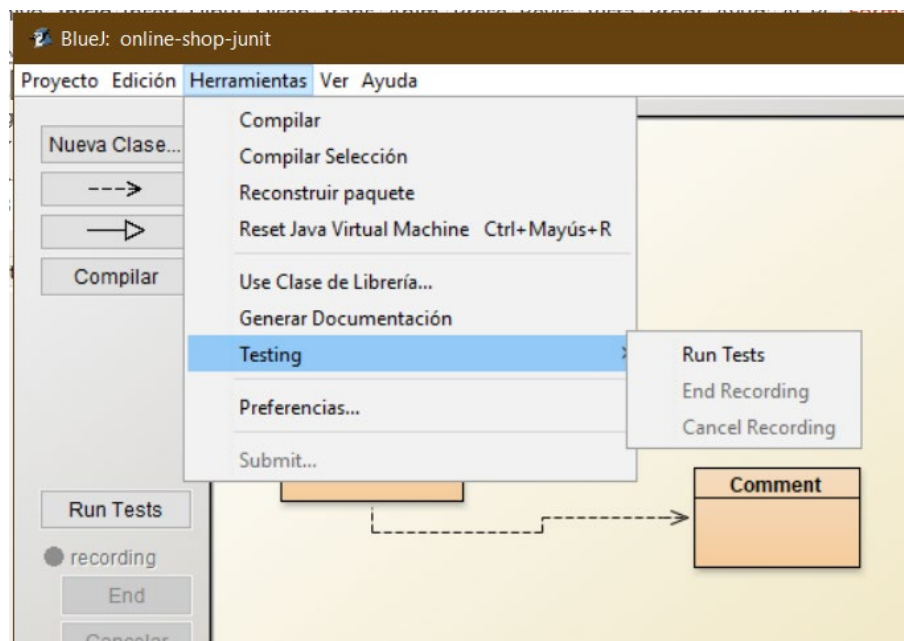
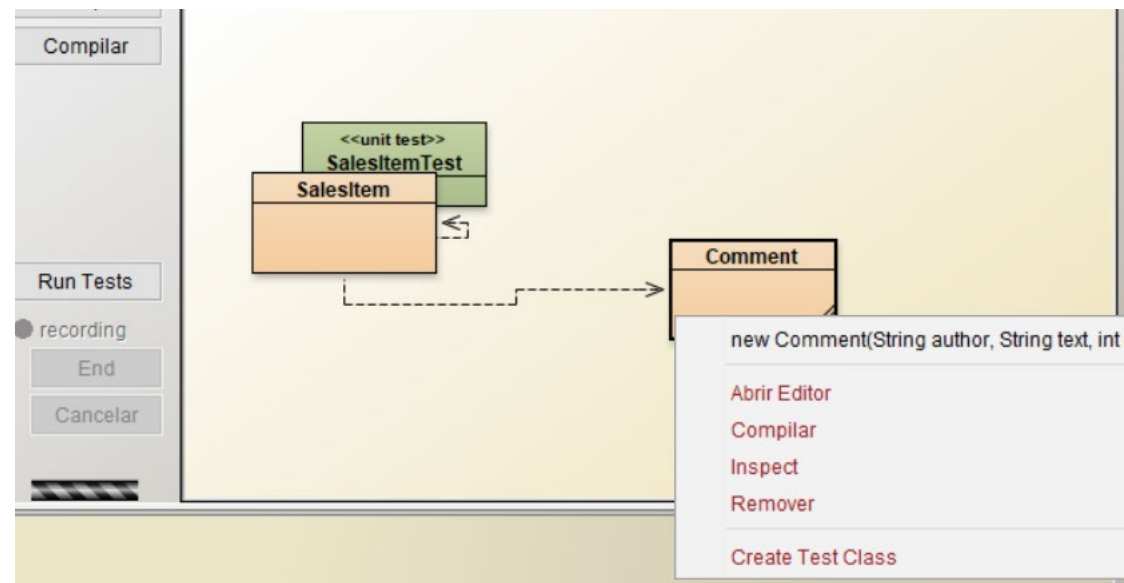
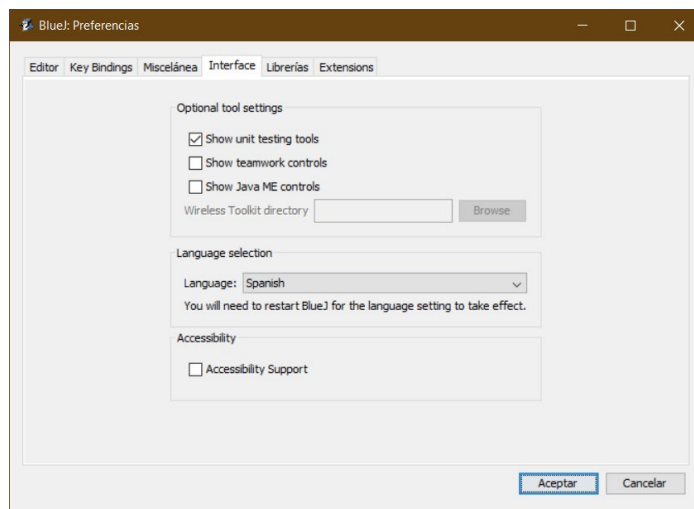
Capítulo 9: Objetos con un buen comportamiento
Apéndice G: Herramientas Junit de prueba de unidades

PRUEBAS: Conceptos importantes

- **Prueba:** determinar si una parte del código produce el comportamiento pretendido
- **Depuración:** intento de apuntar con precisión para localizar un error en el código y corregirlo
- **Prueba de unidad:** Prueba de una parte individual de una aplicación: un método, una clase, un grupo de clases... Puede hacerse antes de que la aplicación esté completa.
 - **Elementos clave:**
 - Analizar el comportamiento de las estructuras de datos cuando están vacías y cuando están llenas
 - Siempre analizar los límites
 - Comprobar que no se sobrescriban las posiciones de los arrays
- **Prueba de aplicación:** Prueba de la aplicación en su totalidad
- **Prueba automatizada:**
 - **Pruebas de regresión** cada vez que se hace una modificación: las que ya se han pasado se siguen pasando
 - Para facilitarlos se puede escribir una batería de pruebas
- **Prueba positiva:** Prueba sobre los casos que se espera que funcionen bien (lo que sería el uso normal)
- **Prueba negativa:** Prueba sobre los casos que se espera que fallen (casos que no se deberían producir)

PRUEBAS: Conceptos importantes

- **JUnit, www.junit.org** JUnit es un popular marco de trabajo (*framework*) para implementar en Java pruebas de unidad organizadas y pruebas de regresión.
- **Aserción:** expresión que establece una condición que se espera que resulte verdadera
(<https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html>)
- **Fixture:** banco de objetos con un estado definido que sirve como base para las pruebas de unidades.
 - Para crearlo, se crean los objetos que se quiera que formen parte del banco de objetos, se elige la opción “Object Bench to Test Fixture”
 - Los objetos desaparecen del entorno y se añaden a la clase de prueba. Para modificarlos, se selecciona “Test Fixture to Object Bench” y vuelven a aparecer



```

/**
 * Create a new sales item.
 */
public SalesItem(String name, int price)
{
    this.name = name;
    this.price = price;
    comments = new ArrayList<>();
}

/**
 * Return the name of this item.
 */
public String getName()
{
    return name;
}

/**
 * Return the price of this item.
 */
public int getPrice()
{
    return price;
}

/**
 * Return the number of customer comments for this item.
 */
public int getNumberOfComments()
{
    return comments.size();
}

/**
 * Add a comment to the comment list of this sales item. Return true if successful;
 * false if the comment was rejected.
 *
 * The comment will be rejected if the same author has already left a comment, or
 * if the rating is invalid. Valid ratings are numbers between 1 and 5 (inclusive).
 */
public boolean addComment(String author, String text, int rating)
{
    if (ratingInvalid(rating)) { // reject invalid ratings
        return false;
    }

    if (findCommentByAuthor(author) != null) { // reject mutiple comments by same author
        return false;
    }

    comments.add(new Comment(author, text, rating));
    return true;
}

```

SalesItemTest - online-shop-junit

Clase Editar Herramientas Opciones

Compilar Deshacer Cortar Copiar Pegar Encontrar... Cerrar Implementación

```

/**
 * Test that a comment can be added, and that the comment count is correct afterwards.
 */
@Test
public void testAddComment()
{
    SalesItem salesItem = new SalesItem("Brain surgery for Dummies", 21998);
    assertEquals(true, salesItem.addComment("James Duckling", "This book is great. I perform brain surgery every week now"));
    assertEquals(1, salesItem.getNumberOfComments());
}

/**
 * Test that a comment using an illegal rating value is rejected.
 */
@Test
public void testIllegalRating()
{
    SalesItem salesItem = new SalesItem("Java For Complete Idiots, Vol 2", 19900);
    assertEquals(false, salesItem.addComment("Joshua Black", "Not worth the money. The font is too small.", -5));
}

/**
 * Test that a sales item is correctly initialised (name and price).
 */
@Test
public void testInit()
{
    SalesItem salesItem = new SalesItem("test name", 1000);
    assertEquals("test name", salesItem.getName());
    assertEquals(1000, salesItem.getPrice());
}

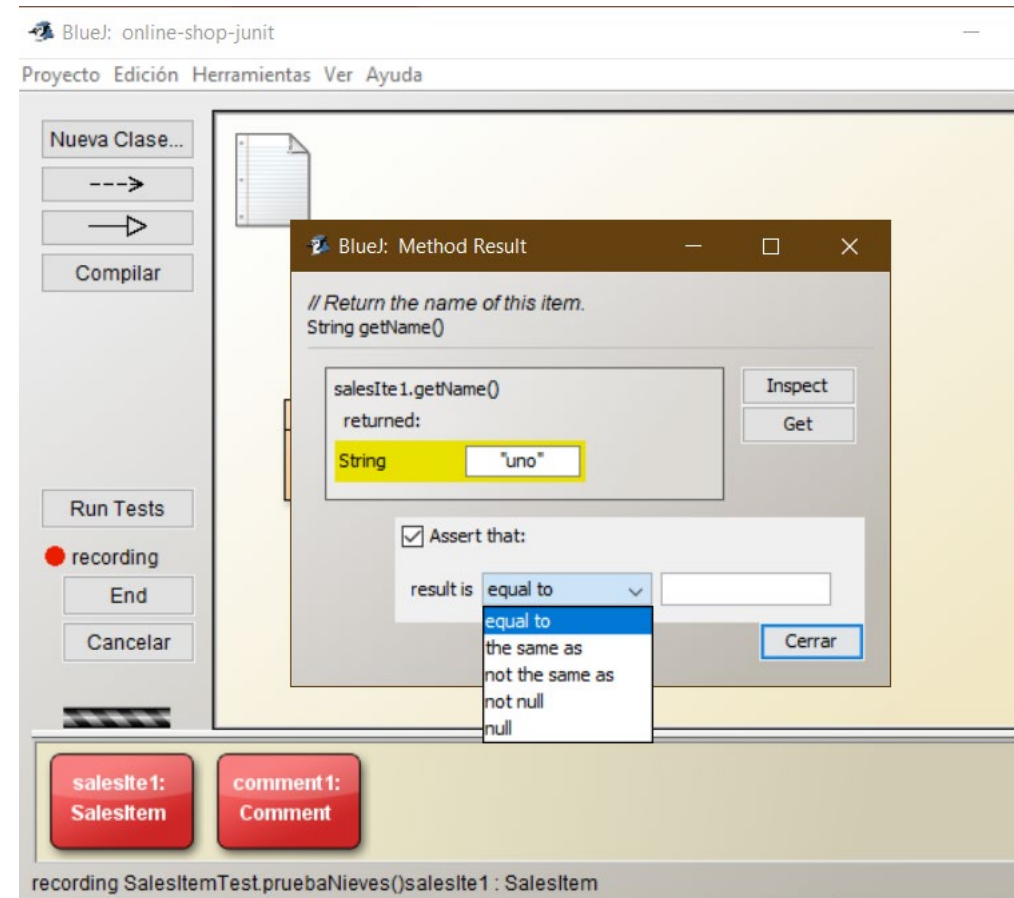
@Test
public void addComment()
{
    SalesItem salesItem = new SalesItem("Brain Surgery for Dummies.", 9899);
    assertEquals(true, salesItem.addComment("Fred", "Great - I perform brain surgery every week now!", 4));
}

```

guardado

Grabar una prueba

- Desde la clase de prueba -> botón derecho -> Create test Method -> damos nombre al método de prueba -> empieza a grabar -> se llama a un método y se selecciona la salida esperada
- (Se repite las veces que se quiere)
- End para finalizar



Consideraciones

- Se ejecutará un método anotado con `@Before` antes de cada ejecución de los métodos `@Test`.
- Análogo un `@After` método anotado es ejecutado después de cada `@Test` método. Esto se puede usar para establecer repetidamente una configuración de prueba y limpiar después de cada prueba.
- Así las pruebas son independientes y el código de preparación no se copia dentro del método `@Test`.
- Los métodos anotados con `@Before` o `@After` deben ser `public void` y con cero argumentos.

MODULARIZACIÓN E INTERFACES

- Interfaz: aquellas partes de una clase que se conocen y que se utilizan en otras clases.
- Las signatures de los métodos proporcionan suficiente información de una clase sobre cómo interactuar con otra sin necesidad de saber cómo están implementados sus métodos.
- Tratamos de separar las interfaces de las clases de los detalles de implementación.

COMENTARIOS Y ESTILO

- Clases:
 - Comentario en la parte superior indicando el propósito de la misma, el autor y el número de versión.
 - Identación que permita distinguir los niveles de los bloques anidados y las estructuras de control.
 - Nombres significativos para variables y métodos.
- Métodos de interfaz:
 - Comentario que indica su propósito, sus parámetros y su tipo de retorno.

SEGUIMIENTO DEL CÓDIGO PARA LOCALIZAR ERRORES

- Seguimiento manual o prueba de escritorio es la actividad en la que trabajamos sobre un segmento de código línea por línea mientras se observan los cambios de estado y otros comportamientos de la aplicación.
 - Una forma de hacer esto en papel y lápiz es construyendo una tabla de los campos del objeto y sus valores
 - Se puede agregar una nueva línea para llevar el registro de los valores que surgen durante la ejecución, después de cada llamada a método
- Sentencias de impresión:
 - qué métodos se han invocado;
 - los valores de los parámetros;
 - el orden en que se han invocado los métodos;
 - los valores de las variables locales y de los campos en lugares estratégicos.

Apéndice F del libro: Utilización del depurador