

INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS

APRENDIZAJE ACTIVO BASADO EN CASOS



Nivel 1

Búsqueda, Ordenamiento y Pruebas Automáticas

1. Objetivos Pedagógicos

Al final de este nivel el lector será capaz de:

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

2. Motivación

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

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

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

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

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

3. Caso de Estudio #1: Un Traductor de Idiomas

En este primer caso de estudio del nivel, queremos construir un programa de computador que permita traducir palabras de español a otros idiomas (inglés, francés e italiano) y de los otros idiomas entre ellos, haciendo la suposición de que las traducciones son únicas. Para esto el programa debe contar con tres diccionarios, que pueden tener conjuntos de palabras distintas. Esto quiere decir que no necesariamente las mismas palabras que tienen traducción al inglés, tienen traducción al francés y viceversa. Lo anterior se ilustra en la figura 1, en la cual aparecen tres diccionarios: español-inglés, español-francés y español-italiano.

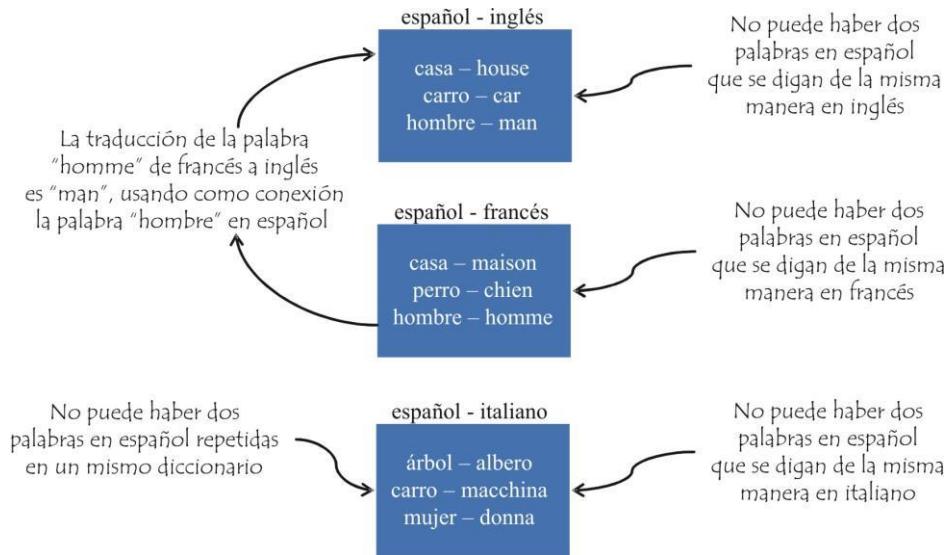
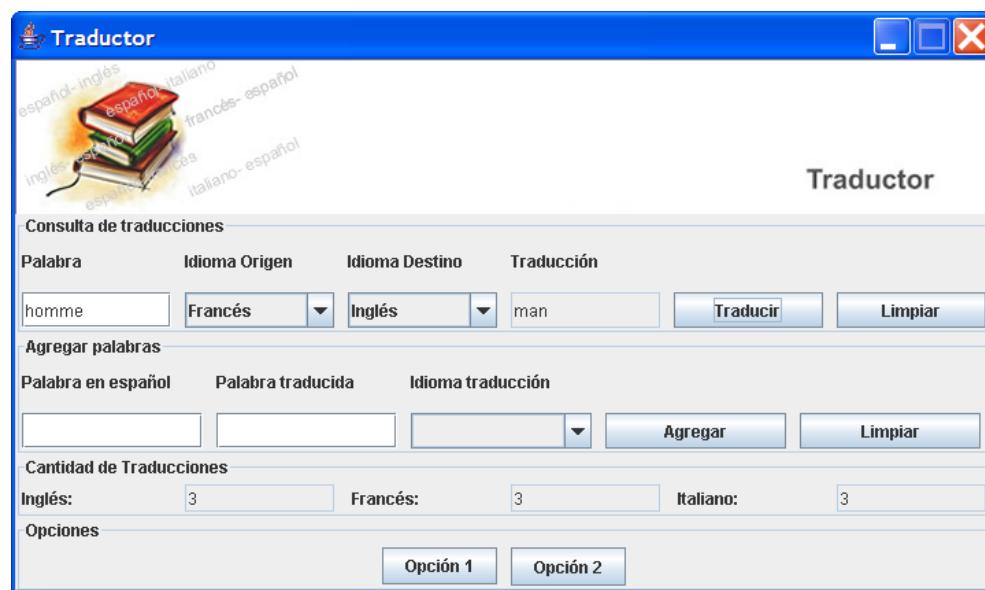


Fig. 1 – Ilustración del proceso de traducción utilizado por el programa.

Para que el programa funcione correctamente hay dos características que deben respetar los diccionarios. La primera es que no hay dos palabras en español repetidas en ningún diccionario (cada palabra tiene una sola traducción en el otro idioma). La segunda es que no hay dos palabras en español con la misma traducción en ningún diccionario (cada palabra tiene una traducción que es única). Con estas dos características, podemos utilizar los tres diccionarios para

hacer traducciones entre los cuatro idiomas: si es de español a otro idioma, sencillamente utilizamos el respectivo diccionario. Si es de un idioma distinto a español, a cualquier otro idioma, lo que hacemos es encontrar en el diccionario que corresponde la palabra en español de la cual la palabra que buscamos es una traducción, y luego consultamos la traducción de dicha palabra en el otro diccionario. Por ejemplo, si el usuario quiere traducir la palabra “homme” de francés a inglés, primero usamos el diccionario español-francés para determinar que “homme” es la traducción en francés de la palabra “hombre”. Luego, usamos el diccionario español-inglés para buscar la traducción de “hombre” y encontramos la palabra “man”.

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



- La ventana está dividida en tres zonas: la primera para las consultas, la segunda para agregar palabras y la tercera con información sobre el número de palabras en cada diccionario.
- Para consultar una palabra el usuario debe teclearla en la primera zona de texto, luego debe seleccionar el “Idioma origen” y el “Idioma destino” y finalmente oprimir el botón “Traducir”.
- Para agregar una palabra a un diccionario, en la segunda zona debe teclear la palabra y su traducción, y luego seleccionar el “Idioma traducción” y oprimir el botón “Agregar”.

Fig. 2 – Interfaz de usuario del traductor de idiomas.

3.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none"> • Construir un programa siguiendo un proceso disciplinado de desarrollo. 	<ul style="list-style-type: none"> • Revisar y seguir el proceso de desarrollo de programas estudiado en cursos anteriores.
<ul style="list-style-type: none"> • Utilizar los mecanismos disponibles para ayudar a construir un programa correcto. 	<ul style="list-style-type: none"> • Aprender a construir el invariante de una clase y a implementar los métodos que van a permitir verificar que éste se cumpla en todo momento. • Aprender a desarrollar las pruebas unitarias automáticas para verificar que los métodos de una clase cumplen con su contrato (en un contexto controlado) y aprender a utilizarlas de manera que apoyen el proceso de desarrollo de programas.

3.2. Comprensión de los Requerimientos



El primer paso cuando se va a resolver un problema es analizarlo, lo cual significa entenderlo e identificar los tres aspectos en los cuales siempre se puede descomponer: los requerimientos funcionales, el mundo del problema y los requerimientos no funcionales.

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

TAREA #1	<u>Objetivo:</u> Entender el problema del caso de estudio del traductor.	
	(1) Lea detenidamente el enunciado del caso de estudio del traductor, (2) identifique y complete la documentación de los tres requerimientos funcionales que allí aparecen.	
Requerimiento funcional 1	Nombre	R1 – Agregar una palabra en español a un diccionario.
	Resumen	
	Entradas	(1) (2) (3)
	Resultado	
Requerimiento funcional 2	Nombre	R2 – Consultar la traducción de una palabra.
	Resumen	
	Entradas	(1) (2) (3)
	Resultado	
Requerimiento funcional 3	Nombre	R3 – Consultar el número de palabras en cada diccionario.
	Resumen	
	Entradas	
	Resultado	

3.3. Comprensión del Mundo del Problema



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

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

En la figura 3 aparece el diagrama de clases del traductor. Allí aparecen dos entidades (**Traductor** y **Traducción**) con tres asociaciones entre ellas. Cada una de estas asociaciones representa un diccionario, que está constituido por un grupo de traducciones. Cada traducción, por su parte, es una pareja de la forma palabra-traducción. Para representar los idiomas, agregamos en el modelado cuatro constantes, una por idioma.

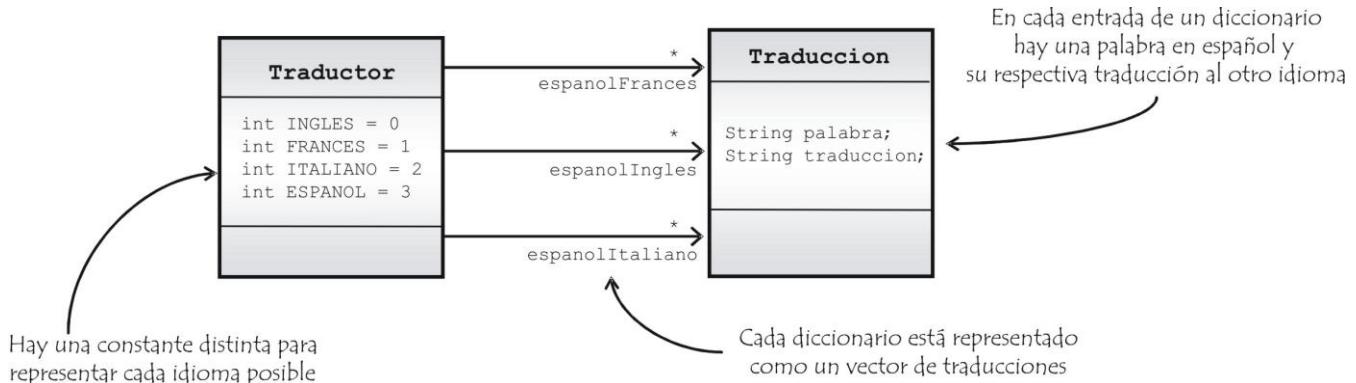


Fig. 3 – Diagrama de clases del modelo del mundo del problema.

En algunos casos es conveniente ilustrar el diagrama de clases de un modelo mediante un diagrama de objetos, que no es otra cosa que un ejemplo de cómo podría materializarse el diagrama de clases en un escenario particular. En la figura 4 aparece un diagrama de objetos posible para el traductor. Allí aparecen por ejemplo dos palabras en el diccionario español-francés (“hombre” y “casa”) con sus respectivas traducciones (“homme” y maison”). También aparecen dos palabras en el diccionario español-inglés (“hombre” y “carro”) con las palabras del idioma inglés con las que se traducen (“man” y “car”). Finalmente, en el diccionario español-italiano se encuentran las parejas “carro-macchina” y “mujer-donna”. Estudie detenidamente el diagrama que se muestra en la figura y vea la manera como se representan los vectores y los objetos contenidos en cada uno de ellos.

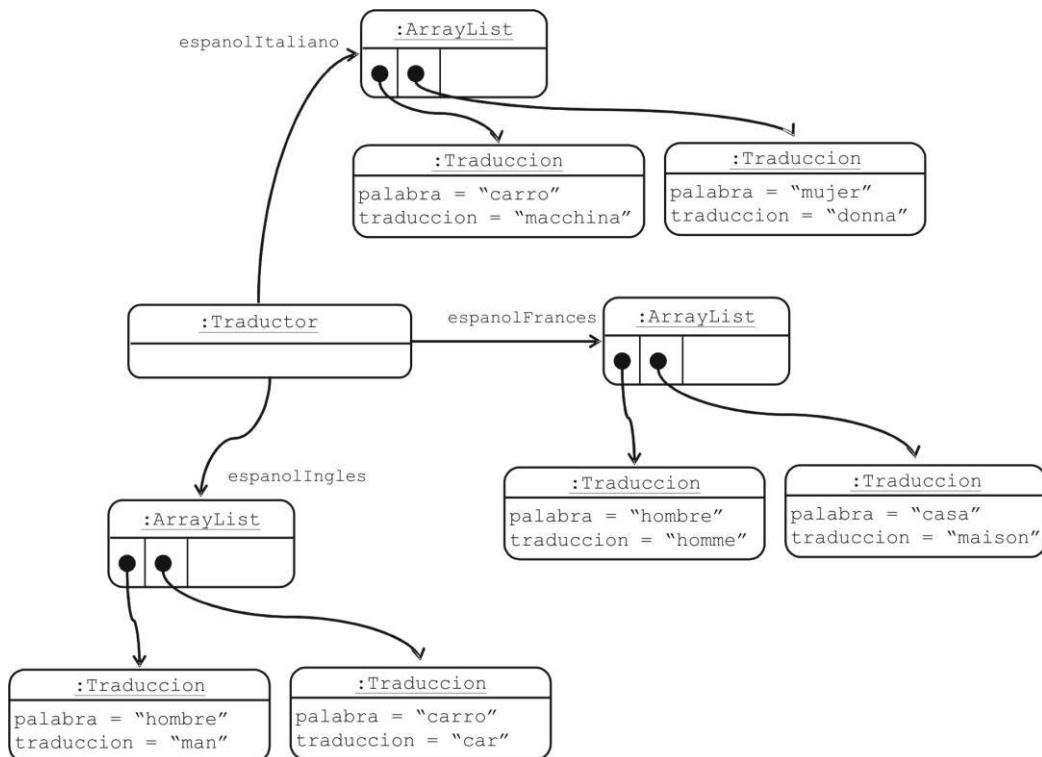


Fig. 4 – Diagrama de objetos para ilustrar un estado posible del modelo del mundo.

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



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

Utilizando los diagramas presentados en las figuras 3 y 4, escriba en Java las declaraciones de las clases. Suponga que éstas van en el paquete `uniandes.cupi2.traductor.mundo`.

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

```
public class Traductor
{
```

```
}
```

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

```
public class Traduccion
{
```

```
}
```

3.4. Invariantes de Clase y Noción de Corrección del Modelo

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

- ¿Cuál es el problema que se quiere resolver?

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

Aquí aparecen por lo menos tres problemas: (1) Falta de documentación de las características del modelo del mundo. El objetivo que nos habíamos planteado de que cualquiera que leyera los documentos producidos en la etapa de análisis entendiera el problema completo no se está cumpliendo. ¡Eso hay que corregirlo! (2) Duplicación de la información en los contratos de los métodos. Si la clase `Traductor` tiene diez métodos, en la precondición de los contratos de todos ellos debe ir la información con las propiedades de los diccionarios. Por todos lados debemos estar repitiendo que las traducciones son únicas, por ejemplo. Eso no es grave, pero en problemas grandes eso representa una pérdida de tiempo. (3) Poca claridad en la asignación de la responsabilidad de verificar si las propiedades definidas para una clase se cumplen. ¿Debe haber un método de la clase `Traductor` que verifique que las traducciones sean únicas? ¿Siempre hay que llamarlo antes de invocar cualquier otro método de la clase para asegurarnos que se cumple la precondición del contrato? La respuesta a esta última pregunta es “claro que no” (por razones de eficiencia), pero entonces, ¿cómo hacer para asegurarnos que siempre se cumplan las propiedades que definimos para una clase? Fíjese que si todos los métodos de la clase se comprometen a dejar sus objetos en un estado que cumpla esas propiedades, no habría necesidad de verificarlo antes de llamar los otros métodos.

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

- ¿Qué es el invariante?

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

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

EJEMPLO #1		Objetivo: Identificar el invariante de las clases del caso de estudio.
		En este ejemplo se muestran y se explican las aserciones que hacen parte del invariante de las clases Traductor y Traducción.
	Invariante	Explicación
Clase Traducción	<ul style="list-style-type: none"> palabra != null !palabra.equals("") traducción != null !traducción.equals("") 	<ul style="list-style-type: none"> → En una traducción la palabra no puede ser nula ni vacía. → En una traducción la palabra traducida no puede ser nula ni vacía. → Si esas cuatro condiciones se cumplen, consideramos que una traducción es válida, y como tal puede hacer parte de un diccionario. → Estas cuatro condiciones no hay necesidad de incluirlas en las precondiciones de los métodos de la clase, sino que se suponen siempre ciertas. → Más adelante veremos quién es responsable de garantizar que estas condiciones siempre se cumplan.
Clase Traductor	<ul style="list-style-type: none"> españolIngles != null españolFrances != null españolItaliano != null En el vector españolIngles no hay palabras repetidas En el vector españolFrances no hay palabras repetidas En el vector españolItaliano no hay palabras repetidas En el vector españolIngles no hay traducciones repetidas En el vector españolFrances no hay traducciones repetidas En el vector españolItaliano no hay traducciones repetidas 	<ul style="list-style-type: none"> → El invariante de esta clase consta de nueve aserciones. → Las tres primeras son restricciones sobre los valores que pueden tener los tres vectores que representan los diccionarios del traductor. Aquí afirmamos que están convenientemente inicializados. Fíjese que esto va a definir hasta dónde va la responsabilidad de los métodos constructores. → Las siguientes aserciones indican que en un diccionario no hay palabras repetidas. En un invariante se puede utilizar lenguaje natural (español) mientras lo que se diga sea suficientemente preciso. También se puede utilizar notación matemática. Por ejemplo, la cuarta aserción se podría haber escrito así: $\text{españolIngles}_i.\text{palabra} \neq \text{españolIngles}_k.\text{palabra}$, para todo $i \neq k$ → Las últimas tres aserciones dicen que las traducciones son únicas en cada uno de los diccionarios. → Todos los métodos de la clase tienen derecho a suponer que todo lo que se afirma en el invariante es cierto cuando el método va a comenzar a ejecutarse.

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



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

- ¿Cómo calcular el invariante?

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

Para definir el invariante se debe revisar inicialmente el enunciado y sacar de allí toda la información de restricciones o relaciones que puedan aparecer en los elementos del mundo. Luego, se incluyen las aserciones correspondientes al modelado de las características. Si por ejemplo, un atributo es un entero que sólo puede tomar como valor algunas constantes que definimos especialmente con ese propósito, se debe incluir esta información en el invariante. Por último, se revisan las responsabilidades de construcción y se decide quién es responsable de inicializar los atributos. Si es el constructor, debe ir al invariante. Si todos los métodos deben verificar si ya están inicializados los atributos y hacerlo en caso de que no sea así, en el invariante se debe simplemente omitir cualquier referencia a la inicialización.

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

- ¿Cómo documentar el invariante de una clase?

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

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

```

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

```

- ¿Cómo verificar el invariante durante la ejecución?

Antes de presentar la manera de escribir los métodos que van a permitir verificar si el invariante se cumple, vamos a presentar una nueva instrucción del lenguaje Java, disponible desde la versión 1.4, que tiene algunas características que la hacen muy interesante para escribir este tipo de métodos.

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

<code>assert expresión;</code>	<p>→ La expresión debe ser de tipo lógico. Si al evaluar la expresión da verdadero, el programa continúa normalmente. Si la evaluación de la expresión da falso, el programa lanza un tipo especial de excepción llamado <code>AssertionError</code> que hace que el programa termine si nadie lo atrapa.</p> <p>→ Por ejemplo: <code>assert palabra != null;</code></p>
<code>assert expresión1 : expresión2;</code>	<p>→ Este caso es una variante del anterior, en el cual la expresión2 debe ser de tipo cadena de caracteres. La única diferencia es que al lanzar la excepción con el error, se le asocia el resultado de evaluar la expresión2.</p> <p>→ Por ejemplo: <code>assert palabra != null : "palabra inválida";</code></p>

Utilizar esta instrucción tiene varias ventajas, entre las cuales está que Java permite activar y desactivar la verificación de las aserciones de manera muy sencilla, de tal forma que cuando esté desactivada la verificación (por ejemplo cuando se instala el programa en el computador del usuario después de estar seguros de que está correcto) el computador no pierde tiempo haciendo verificaciones que en ese momento ya son inútiles. Más adelante veremos como activar y desactivar esta opción desde Eclipse y desde el programa .bat que lo ejecuta.

Existen muchas formas posibles de escribir los métodos para verificar el invariante de una clase. En todos los casos de estudio del libro vamos a utilizar una manera particular de hacerlo, lo que nos va a permitir guiar metodológicamente el proceso de escribirlo y facilitar la localización de los métodos que lo hacen.

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

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

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

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

**EJEMPLO #2**

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

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

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

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

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

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

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

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

```
public class Traductor
{
    ...
    // -----
    // Invariante
    // -----
    private void verificarInvariante( )
    {
        assert espanolIngles != null :
            "Diccionario español-inglés sin inicializar";
        assert espanolFrances != null :
            "Diccionario español-francés sin inicializar";
        assert espanolItaliano != null :
            "Diccionario español-italiano sin inicializar";

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

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

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

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



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

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

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

```
}
```

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

```
}
```

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

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

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

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

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

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



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

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

<code>-ea</code>	→ Las aserciones de todas las clases del programa serán ejecutadas.
<code>-ea:uniandes.cupi2.traductor.mundo.Traducion</code>	→ Se activan únicamente las aserciones de la clase Traducion. Las instrucciones assert de las demás clases del proyecto no son ejecutadas.
<code>-ea:uniandes.cupi2.traductor.mundo...</code>	→ Con esta opción, en la llamada de un programa, se ejecutan las instrucciones assert de todas las clases del paquete uniandes.cupi2.traductor.mundo (en nuestro caso las clases Traductor y Traducion).
<code>-ea -da:uniandes.cupi2.traductor.mundo.Traductor</code>	→ Las aserciones de todas las clases del programa serán ejecutadas, con excepción de la clase Traductor. La opción <code>-da</code> (<code>-disableassertions</code>) desactiva la ejecución de la instrucción assert en una clase o paquete particular.

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

3.5. Asignación de Responsabilidades e Implementación de Métodos



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

EJEMPLO #3 	<p><u>Objetivo:</u> Entender la asignación de responsabilidades y los contratos de los métodos en el caso de estudio.</p> <p>En este ejemplo podemos ver los contratos de los métodos y la explicación de la manera como se asignó cada responsabilidad a cada uno de ellos. En algunos casos aparece el código que implementa el método, para ilustrar los puntos en los cuales se debe verificar el invariante de las clases.</p> <p>Para estudiar el código completo de las clases, debe consultar el CD que acompaña el libro.</p>
<pre>public class Traducion { // ----- // Atributos // ----- private String palabra; private String traducion; /** * Crea la traducción de una palabra. * post: Se creó la traducción de la palabra especificada. * @param pal es la palabra. pal != null y pal != "" * @param trad es la traducción. trad != null, trad != "" */ public Traducion(String pal, String trad) { palabra = pal; traducion = trad; // Verifica el invariante verificarInvariant(); } }</pre>	<p>→ Comenzamos por la clase <code>Traducion</code> que es la más simple. Sólo tiene dos atributos: uno con la palabra y otro con su traducción en algún otro idioma.</p>
<pre> /** * Retorna la palabra base de la traducción. * @return La palabra base de la traducción */ public String darPalabra() { return palabra; } }</pre>	<p>→ La primera responsabilidad de la clase es crear instancias correctas de la misma.</p> <p>→ En la precondición exigimos que tanto la palabra como la traducción sean válidas. No estamos obligados entonces a hacer ninguna verificación.</p> <p>→ Al final validamos que el invariante de la clase se cumpla.</p>
	<p>→ La técnica del experto nos pide que tengamos un método para retornar la información que manejamos y que es necesaria en otras clases.</p>

<pre> /** * Retorna la traducción de la palabra. * @return La traducción de la palabra */ public String darTraduccion() { return traduccion; } } </pre>	<ul style="list-style-type: none"> → Por la técnica del experto tenemos también este método. → No hay métodos de modificación en esta clase, porque no hay ningún requerimiento funcional en el caso de estudio que lo requiera.
<pre> public class Traductor { // ----- // Constantes // ----- public final static int INGLES = 0; public final static int FRANCES = 1; public final static int ITALIANO = 2; public final static int ESPANOL = 3; // ----- // Atributos // ----- private ArrayList espanolIngles; private ArrayList espanolFrances; private ArrayList espanolItaliano; } </pre>	<ul style="list-style-type: none"> → Ahora pasamos a la clase Traductor, la cual declara cuatro constantes para representar los posibles idiomas que maneja. → Como atributos tiene los tres diccionarios, que son vectores (ArrayList) que contienen objetos de la clase Traduccion.
<pre> /** * Crea el traductor con los diccionarios vacíos. * post: Se creó el traductor con los diccionarios vacíos. */ public Traductor() { espanolIngles = new ArrayList(); espanolFrances = new ArrayList(); espanolItaliano = new ArrayList(); // Verifica el invariante verificarInvariante(); } } </pre>	<ul style="list-style-type: none"> → El constructor debe crear instancias válidas de la clase (deben cumplir el invariante a la salida del constructor), que satisfagan también la postcondición. → Por eso el constructor debe asumir la responsabilidad de crear los vectores que representan los diccionarios (el invariante dice que no pueden ser nulos y la postcondición dice que deben estar vacíos).
<ul style="list-style-type: none"> → Usando la técnica de descomposición de requerimientos, encontramos que la clase Traductor debe asumir tres grandes responsabilidades. La primera, permitir el ingreso de nuevas palabras. La segunda, buscar la traducción de una palabra. La tercera, calcular el número de palabras que hay en cada diccionario. A continuación se muestran los contratos de estos tres métodos. → Decidimos no lanzar excepciones desde los métodos, sino retornar valores lógicos que indican si la operación se llevó a cabo sin problemas. → Para la implementación de estos métodos es conveniente utilizar la técnica de dividir y conquistar (estudiada en cursos anteriores) y crear un conjunto de métodos privados que resuelvan partes del problema. 	
<pre> /** * Agrega al diccionario una traducción de una palabra en español a un idioma dado. * post: La traducción fue adicionada al diccionario del idioma especificado. * @param pal es la palabra. pal != null y pal != "" * @param trad es la traducción. trad != null y trad != "" * @param idDestino es el idioma destino. idDestino pertenece a {FRANCES, INGLES, ITALIANO} * @return true si la palabra pudo ser adicionada al diccionario o false en caso contrario. */ public boolean agregarTraduccion(String pal, String trad, int idDestino) { } } </pre>	

```

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

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

```

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

En la figura 5 aparece una parte del diagrama de clases de la interfaz de usuario. Allí se muestra la relación entre la interfaz y el modelo del mundo (asociación hacia la clase `Traductor`), además de cinco paneles: tres activos (con botones para el usuario) y dos pasivos (que solo despliegan información). También se puede apreciar la firma de los tres métodos de la ventana principal, mediante los cuales se van a ejecutar los requerimientos funcionales del programa.

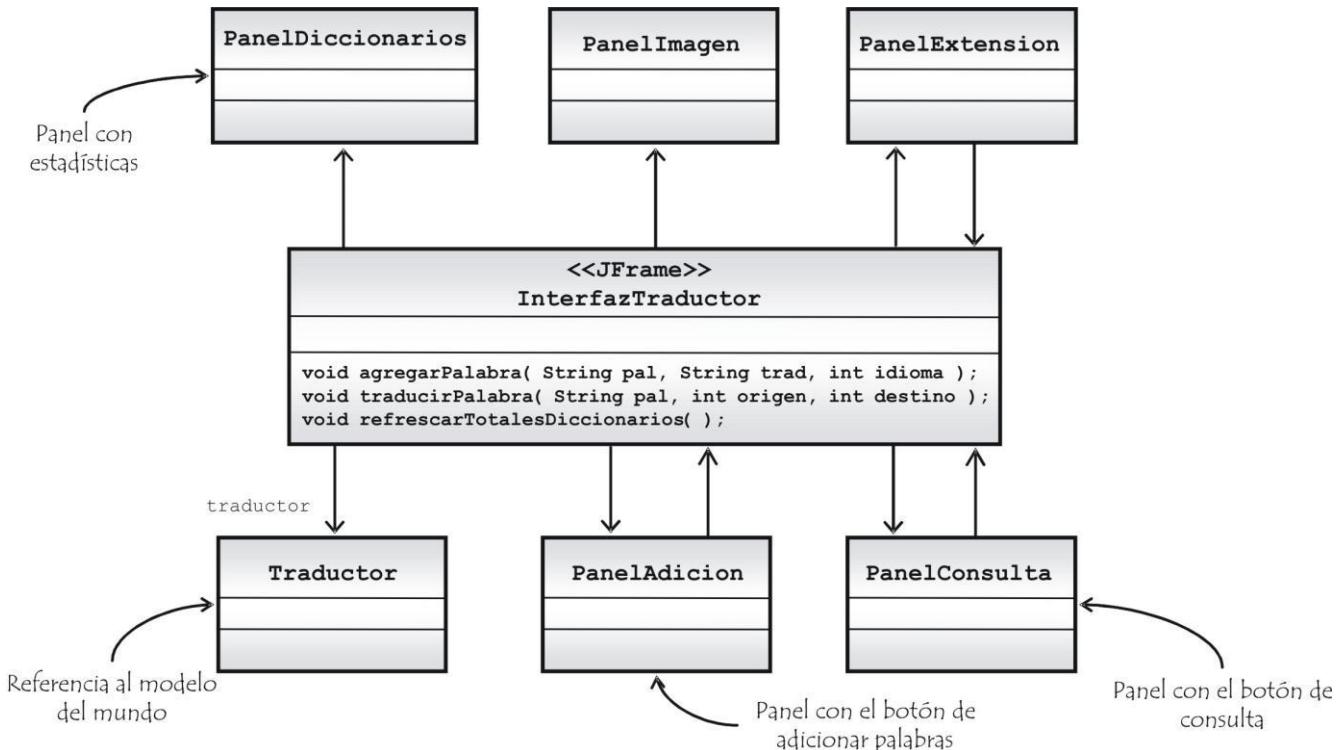


Fig. 5 – Parte del diagrama de clases de la interfaz de usuario.

3.6. Pruebas Unitarias Automáticas

Después de haber dividido las responsabilidades entre todas las clases y de haber definido los contratos de los métodos comienza el proceso de implementación y pruebas, en el cual debemos escribir el cuerpo de los métodos y verificar que cumplan efectivamente su contrato. La manera de probar el código que vayamos desarrollando es el tema de esta sección. En ella comenzaremos con una visión abstracta del problema y terminaremos mostrando cómo utilizar la herramienta `JUnit` para construir las pruebas unitarias automáticas de un programa escrito en Java.

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

3.6.1. Introducción y Motivación

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

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

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

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

3.6.2. Visión Conceptual de las Pruebas

Para facilitar su evolución y desarrollo, vamos a implementar una clase de prueba por cada una de las clases del modelo del mundo que estemos interesados en probar, la cual tendrá siempre una asociación hacia la clase objeto de la verificación. En la figura 6 damos una idea global de la arquitectura que va a tener el programa y la manera como se conectan la interfaz de usuario y las clases de prueba al modelo del mundo. Allí se puede apreciar que cada clase de prueba (`TraductorTest`) tiene una relación directa con la clase que quiere probar (`Traductor`), sin pasar por las clases que implementan la interfaz de usuario (`InterfazTraductor`).

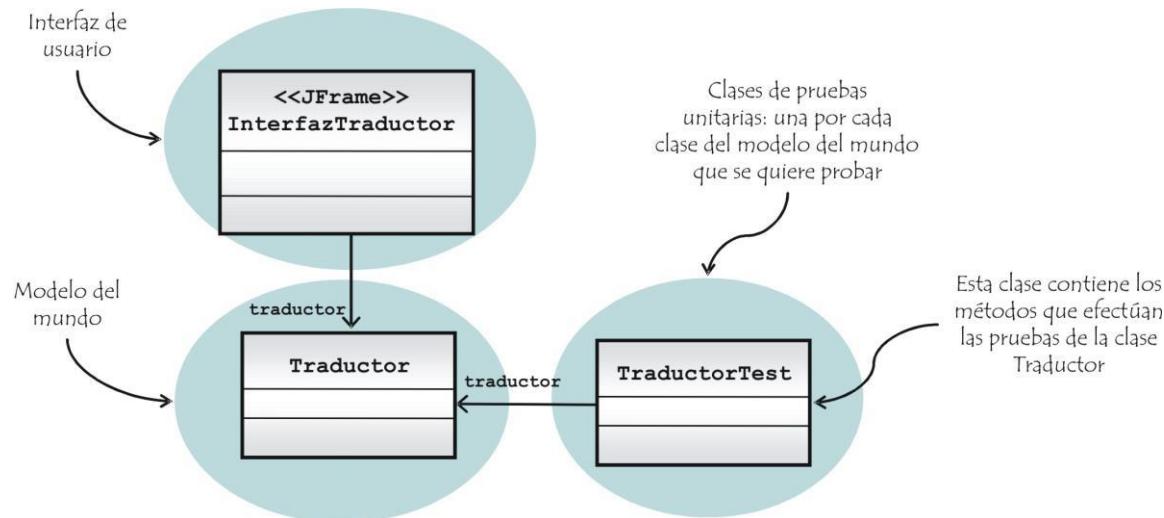
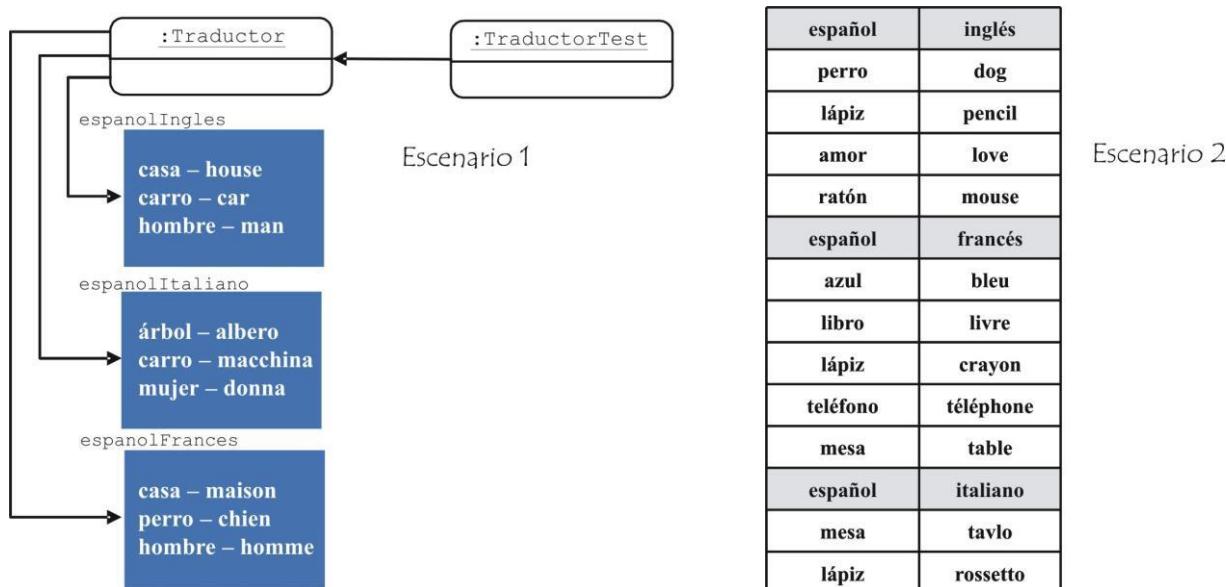


Fig. 6 – Las clases de prueba en la arquitectura de un programa.

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

Hay dos componentes fundamentales que se deben tener en cuenta en el momento de construir una clase de prueba: los escenarios y los casos de prueba. Un **escenario** es un objeto de la clase que se quiere probar, el cual tiene un estado conocido por nosotros (sabemos el valor de cada uno de los atributos y conocemos el estado de los objetos con los cuales está asociado). Un escenario se puede representar con un diagrama de objetos, tal como se mostró en la figura 4 para el caso del traductor, aunque más adelante introduciremos una sintaxis un poco más simple. La primera responsabilidad de una clase de prueba es saber construir distintos escenarios sobre los cuales podamos probar el correcto funcionamiento de los métodos. Vamos a escribir un método por cada escenario que queramos construir. Un escenario siempre debe cumplir con el invariante de la clase.

En la figura 7 mostramos dos escenarios distintos para las pruebas de la clase `Traductor`. Cualquier sintaxis es válida para expresar un escenario, en la medida en que contenga suficiente información para establecer el estado del objeto. Es importante construir los escenarios de manera que representen todas las situaciones posibles en las que se pueda encontrar el modelo del mundo. En el caso del traductor, por ejemplo, sería conveniente definir un escenario en el cual los diccionarios estén vacíos.

Fig. 7 – Dos escenarios posibles para las pruebas de la clase `Traductor`.

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

EJEMPLO #4		Objetivo: Ilustrar la idea de un caso de prueba para un método.			
Clase	Método	Escenario	Valores de entrada		Resultado
Traductor	agregarTraducion()	Fig. 7 Escenario1	pal = azul trad = bleu idDestino = FRANCES		Verdadero. La palabra y la traducción se agregaron al diccionario español-francés.
Traductor	agregarTraducion()	Fig. 7 Escenario1	pal = casa trad = maison idDestino = FRANCES		Falso. No se ha modificado el diccionario español-francés.
Traductor	agregarTraducion()	Fig. 7 Escenario1	pal = coche trad = car idDestino = INGLES		Falso. No se ha modificado el diccionario español-inglés.

Es usual que los casos de prueba se planteen de manera incremental y con un propósito compartido y definido, más que como un conjunto independiente de verificaciones al azar. En el ejemplo 5 mostramos la información adicional que debe hacer parte de la documentación de un caso de prueba.

EJEMPLO #5		Objetivo: Ilustrar la manera de documentar un caso de prueba.							
Prueba No. 1	En este ejemplo se documentan algunos casos de prueba para el método agregarTraducion() de la clase Traductor.								
Prueba No. 1	Objetivo: Probar que el método es capaz de agregar palabras y traducciones válidas a cualquiera de los diccionarios.								
Traductor	agregarTraducion()	Fig. 7 Escenario1	pal = azul trad = bleu idDestino = FRANCES		Verdadero. La palabra y la traducción se agregaron al diccionario español-francés.				
Traductor	agregarTraducion()	-	pal = azul trad = blue idDestino = INGLES		Verdadero. La palabra y la traducción se agregaron al diccionario español-inglés.				
Traductor	agregarTraducion()	-	pal = mesa trad = tavlo idDestino = ITALIANO		Verdadero. La palabra y la traducción se agregaron al diccionario español-italiano.				
Prueba No. 2	Objetivo: Probar que el método no permite agregar palabras repetidas en español a sus diccionarios.								
Traductor	agregarTraducion()	Fig. 7 Escenario1	pal = casa trad = maison idDestino = FRANCES		Falso. No se ha modificado el diccionario español-francés, porque la palabra “casa” ya existe en español en ese diccionario.				
Traductor	agregarTraducion()	-	pal = casa trad = house idDestino = INGLES		Falso. No se ha modificado el diccionario español-inglés, porque la palabra “casa” ya existe en español en ese diccionario.				

Prueba No. 3	Objetivo: Probar que el método no permite agregar traducciones repetidas a sus diccionarios.			
Traductor	agregarTraducion()	Fig. 7 Escenario1	pal = coche trad = car idDestino = INGLES	Falso. No se ha modificado el diccionario español-inglés, porque la traducción “car” ya está asignada a otra palabra en el diccionario.

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

TAREA #4 	<u>Objetivo:</u> Definir las pruebas para el método <code>traducir()</code> de la clase <code>Traductor</code> . (1) Lea detenidamente el contrato del método <code>traducir()</code> de la clase <code>Traductor</code> e identifique los valores de entrada y el resultado esperado, (2) para las tres pruebas cuyos objetivos se plantean más adelante, proponga un conjunto de casos de prueba, usando el segundo escenario definido en la figura 7.			
Prueba No. 1	Objetivo: Probar que el método es capaz de encontrar correctamente la traducción de una palabra del español a cualquiera de los otros idiomas.			
Clase	Método	Escenario	Valores de entrada	Resultado
Traductor	traducir()	Fig. 7 Escenario2		
Traductor	traducir()	-		
Traductor	traducir()	-		
Prueba No. 2	Objetivo: Probar que el método es capaz de encontrar correctamente la traducción de una palabra de un idioma distinto a español a cualquiera de los otros idiomas.			
Traductor	traducir()	Fig. 7 Escenario2		
Traductor	traducir()	-		
Traductor	traducir()	-		

Prueba No. 3	Objetivo: Probar que el método no encuentra la traducción para palabras que no están en el diccionario.			
Traductor	traducir()	Fig. 7 Escenario2		
Traductor	traducir()	-		
Traductor	traducir()	-		



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

Por último, necesitamos alguien que tome cada clase de prueba que nosotros señalemos, ejecute las verificaciones que ella contiene y genere un reporte con los resultados. Dicha clase la llamaremos el **ejecutor de pruebas** y, a menos que utilicemos alguna herramienta que ya tenga uno (como es el caso de JUnit), debemos desarrollarlo. La idea del ejecutor de pruebas se ilustra en la figura 8. Allí aparecen las clases de prueba para el traductor.

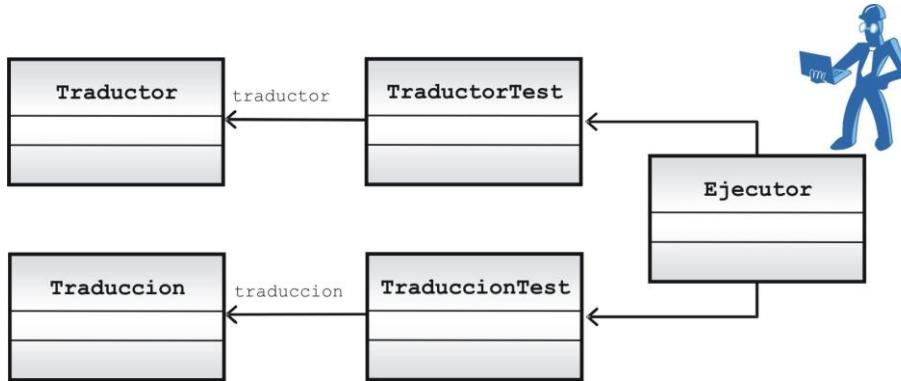


Fig. 8 – El ejecutor de pruebas en el caso de estudio.

3.6.3. JUnit: Un Framework de Construcción de Pruebas

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

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

EJEMPLO #6

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

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

```
package uniandes.cupi2.traductor.test;

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

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

    private Traductor traductor;

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

- Se debe importar la clase TestCase de JUnit, lo mismo que la clase del modelo del mundo que queremos probar. Se debe importar porque esta clase está localizada en un paquete distinto.
- La clase la llamamos TraductorTest para que sea evidente la clase del mundo que pretende probar.
- En el encabezado incluimos la declaración “extends TestCase” para indicar que es una clase de prueba que va a utilizar los métodos definidos en la clase TestCase que hace parte de JUnit.
- Se declara una asociación hacia la clase que se quiere probar. Dicha asociación será utilizada para referenciar en todo momento el escenario de prueba.
- En este ejemplo sólo aparece un método de prueba. Es de tipo void y su nombre comienza por el prefijo “test”. El nombre del método debe dar una idea de la prueba que se quiere hacer con él.

Los métodos para construir escenarios los vamos a definir como privados y van a comenzar por el prefijo “setupEscenario” y un número. Estos métodos deben encargarse de crear una instancia del modelo del mundo e invocar los métodos necesarios para llevarlo al estado que se definió en su diseño. Esto se muestra en el ejemplo 7.

EJEMPLO #7

Objetivo: Mostrar el método de creación de un escenario.

En este ejemplo se presenta el método de la clase TraductorTest encargado de crear el primer escenario definido en la figura 7.

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

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

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

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

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

    ...
}
```

- El objetivo del método es crear el escenario 1 definido en la figura 7, y dejarlo en el atributo “traductor”.

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

- `assertTrue(mensaje, condicion)`: Este método evalúa la condición. Si es verdadera, continúa normalmente la ejecución. Si es falsa, lanza la excepción `AssertionFailedError` asociándole el mensaje que llega como parámetro. Dicho mensaje será utilizado por el ejecutor de pruebas para generar el reporte del error. El mensaje es opcional en este método y en todos los que siguen.
- `assertFalse(mensaje, condicion)`: Funciona de manera similar al anterior, pero lanza la excepción si la condición no es falsa.
- `assertNull(mensaje, objeto)`: Este método lanza la excepción `AssertionFailedError` si el objeto que llega como parámetro no es `null`.
- `assertNotNull(mensaje, objeto)`: Este método lanza la excepción `AssertionFailedError` si el objeto que llega como parámetro es `null`.
- `assertEquals(mensaje, esperado, actual)`: Este método verifica si `esperado == actual`. Si eso es cierto, continúa normalmente la ejecución. Si es falso, lanza la excepción `AssertionFailedError`. Los parámetros `esperado` y `actual` deben ser de tipo `int`, `long` o `char`.
- `assertEquals(mensaje, esperado, actual, delta)`: Este método se utiliza para comparar valores reales, en los cuales no es posible verificar igualdad total, debido a las aproximaciones que hacen los computadores. En ese caso verifica si `esperado == actual`, pero les da un `delta` de error (si no son iguales, la diferencia entre los dos no puede superar este valor).
- `fail(mensaje)`: Este método siempre lanza la excepción `AssertionFailedError`. Sólo se debe situar en puntos de las pruebas en donde es un error que el programa llegue.

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

EJEMPLO #8



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

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

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

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

- Vamos a trabajar sobre un tercer escenario (diccionarios vacíos), puesto que sería un error partir de un escenario que necesita el método que vamos a probar para poderse construir.
- El método va a probar el caso en el cual sí es posible agregar palabras a cada uno de los diccionarios.

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

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

```

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

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

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

// Realiza la búsqueda de traducciones de español a inglés
Traduccion traduccion;

traduccion = traductor.traducir( "blanco", Traductor.ESPAÑOL, Traductor.INGLES );
assertEquals( "white", traduccion.darTraduccion() );

traduccion = traductor.traducir( "perro", Traductor.ESPAÑOL, Traductor.INGLES );
assertEquals( "dog", traduccion.darTraduccion() );

traduccion = traductor.traducir( "casa", Traductor.ESPAÑOL, Traductor.INGLES );
assertEquals( "house", traduccion.darTraduccion() );

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

assertEquals( "bleu", traduccion.darTraduccion() );

traduccion = traductor.traducir( "libro", Traductor.ESPAÑOL, Traductor.FRANCES );
assertEquals( "livre", traduccion.darTraduccion() );

traduccion = traductor.traducir( "casa", Traductor.ESPAÑOL, Traductor.FRANCES );
assertEquals( "maison", traduccion.darTraduccion() );

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

assertEquals( "tavlo", traduccion.darTraduccion() );

traduccion = traductor.traducir( "hoja", Traductor.ESPAÑOL, Traductor.ITALIANO );
assertEquals( "foglia", traduccion.darTraduccion() );

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

}

```

En el método anterior pudimos observar algo que va a ser recurrente en la construcción de las pruebas: los métodos se necesitan entre sí para probarse. La única manera de saber si la palabra y su traducción se agregaron efectivamente en un diccionario es buscando allí la palabra y verificando que el programa le retorna la traducción que le acabamos de dar. Esto nos va a suceder sobre todo cuando vayamos a probar los métodos modificadores.

**TAREA #5**

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

Utilice el escenario 1 (figura 7), definido anteriormente, para probar que el método de traducción funciona de manera efectiva. Utilice el diseño de la prueba que hizo en la tarea 4. Separe el código en secciones, en donde sea claro el objetivo de cada parte del método.

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

 TAREA #6	<p>Objetivo: Ejecutar las pruebas del caso de estudio y localizar en los respectivos directorios los elementos necesarios para que funcione.</p> <p>Siga las instrucciones que se dan a continuación, con el fin de ejecutar las pruebas del traductor.</p> <ol style="list-style-type: none"> 1. Copie del CD el ejemplo <code>n7_traductor.zip</code> en algún lugar del disco duro y descomprímalo. 2. Localice en el subdirectorio “bin” el archivo “build.bat” y ejecútelo. 3. Haga lo mismo con el archivo “buildTest.bat” que se encuentra en ese mismo directorio. Con este archivo compilamos las clases de prueba y creamos el archivo “traductorTest.jar”. Vaya al directorio “test\lib” y verifique que allí se encuentra este archivo, junto con el archivo “junit.jar”. En este último archivo se encuentran todas las clases de JUnit. 4. Edite en el directorio “bin” el archivo “runTest.bat” y revise la manera en que se invoca el ejecutor de JUnit. Fíjese la manera como le decimos al ejecutor la clase que contiene las pruebas. 5. Ejecute el archivo “runTest.bat” y mire en detalle el reporte que genera JUnit, luego de haber ejecutado las pruebas. ¿Cuántos métodos de prueba ejecutó? ¿Cuántos fueron exitosos? 6. Lance Eclipse y cree el proyecto <code>n7_traductor</code>. Localice el código fuente de las pruebas, en el directorio “test\source”. Edite el archivo <code>TraductorTest</code> y estudie la manera en que se crean los escenarios. ¿Cuántos escenarios crea esa clase? ¿En cuál escenario están todos los diccionarios vacíos? 7. En la clase <code>TraductorTest</code> localice los métodos de prueba. ¿Cuántos son? ¿Qué objetivo tiene cada uno de ellos? 8. Vaya a la vista del explorador de paquetes (<i>Package Explorer</i>). Localice allí el archivo <code>TraductorTest</code>. Vamos a utilizar la facilidad que nos da Eclipse para ejecutar las pruebas desde su interior. Con el botón derecho del ratón sobre el archivo, seleccione la opción “Run As JUnit Test”. ¿Cómo despliega Eclipse el reporte de las pruebas? Modifique algo en la clase de prueba, de manera que aparezca que una de las pruebas falla. ¿Qué información nos da JUnit para localizar el punto que generó el error? ¿Qué otra información nos suministra JUnit en el reporte? 9. En la vista del explorador de paquetes, seleccione el proyecto <code>n7_traductor</code> y con el botón derecho escoja la opción “Properties”. Allí aparece una ventana con las propiedades del proyecto. Busque en la parte izquierda la entrada llamada “Java Build Path” y haga clic sobre ella. Seleccione la pestaña llamada “Source” y verifique que aparezca el directorio “test\source” allí incluido. De esa manera le decimos a Eclipse que en ese directorio también hay código fuente. Seleccione ahora la pestaña “Libraries”. Allí aparece que el archivo <code>junit.jar</code> se encuentra en el directorio “test\lib”. De esa forma le contamos a Eclipse en dónde debe buscar otras clases ya compiladas. 10. Cierre la ventana de propiedades del proyecto y ejecute el programa. Verifique que todo funciona según la especificación. Imagine cómo sería un guión de pruebas para que un usuario pudiera verificar la correcta implementación de todos los requerimientos funcionales. 11. Vaya al CD que acompaña el libro y localice el menú de recursos. Busque el Javadoc de JUnit. Mire la documentación de las clases <code>TestCase</code> y <code>Assert</code>. Estudie los otros métodos disponibles en JUnit para la construcción de pruebas.
--	--

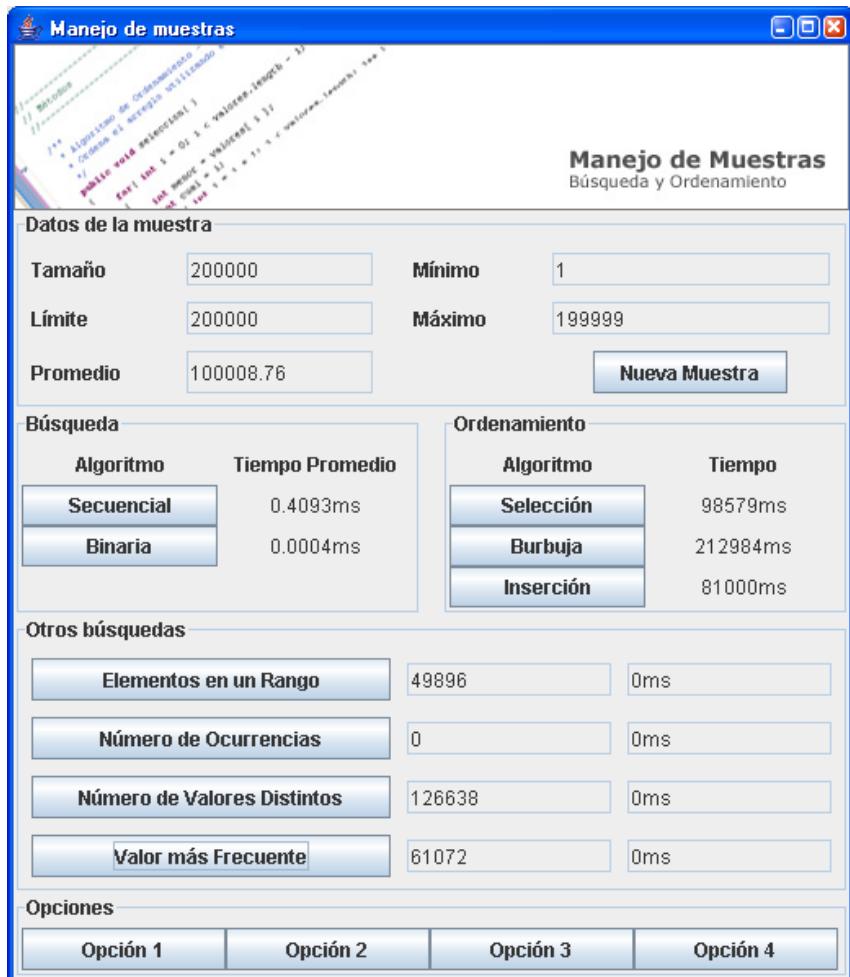
4. Caso de Estudio #2: Un Manejador de Muestras

En este segundo caso del nivel vamos a desarrollar un programa que nos permita manejar una muestra de datos. Una muestra es una secuencia de valores enteros que se encuentran en un rango dado (por simplicidad vamos a suponer que el límite inferior del rango es siempre uno). Piense por ejemplo en los resultados de una encuesta, en la cual las personas califican el desempeño del presidente con un valor entre 1 y 10. O piense también en los números de la

tarjeta de identidad de todos los niños nacidos en el país entre 1999 y 2001. Las operaciones que nos interesan sobre una muestra tienen que ver con ordenamientos y búsqueda de información, y con el tiempo que gastan los distintos algoritmos disponibles para cumplir dichas tareas.

El programa que vamos a desarrollar debe ofrecer las siguientes opciones: (1) crear una nueva muestra de manera aleatoria, de un tamaño y con un límite superior definidos por el usuario (el usuario podría por ejemplo pedir una muestra de 200.000 elementos, en el rango 1 a 200.000), (2) ordenar la muestra utilizando uno de los siguientes algoritmos: inserción, selección o intercambio (burbuja), y mostrar el tiempo que gasta el computador en ejecutar dicha tarea, (3) mostrar la eficiencia de los algoritmos de búsqueda secuencial y binaria. Para esto, el programa debe calcular el tiempo que gasta en buscar cada uno de los elementos del rango de la muestra y dividirlo por el número de elementos que buscó. Por ejemplo, si la muestra está en el rango 1 a 200.000, el programa debe buscar cada uno de esos valores en la muestra, calcular el tiempo total del proceso y dividir por 200.000, (4) calcular el número de elementos que se encuentran en un rango dentro de la muestra ordenada (por ejemplo, determinar cuántos valores de la muestra están entre 50.000 y 100.000), (5) calcular el número de veces que aparece un valor dado en la muestra ordenada (por ejemplo, cuántas veces aparece en la muestra el valor 30.000), (6) calcular el número de valores distintos en la muestra ordenada, (7) encontrar el valor que más veces aparece en la muestra ordenada y (8) dar información estadística de la muestra no ordenada, incluyendo el mayor valor, el menor valor y el promedio.

La interfaz de usuario del programa se muestra en la figura 9, en donde aparecen los resultados de la ejecución para una muestra de 200.000 datos, con valores en un rango de 1 a 200.000. Allí podemos apreciar que hay diferencias considerables de tiempo al ordenar la muestra utilizando los distintos algoritmos disponibles en el programa. También se puede ver la diferencia de tiempo que existe entre una búsqueda secuencial y una búsqueda binaria. Casi 1000 veces más veloz la segunda que la primera.



- Para iniciar el programa debemos crear una muestra, usando el botón “Nueva Muestra”. Allí definimos el tamaño y el límite superior. El programa contesta creando la muestra de manera aleatoria y presentando los primeros resultados estadísticos (mínimo, máximo y promedio).
- Inicialmente todos los botones que sólo funcionan sobre una muestra ordenada se encuentran desactivados. Es el caso, por ejemplo, de la búsqueda binaria.
- Luego, se debe ordenar la muestra utilizando cualquiera de los algoritmos disponibles en el programa. Después de esto se activan las opciones que no se encontraban disponibles.
- En este momento se pueden utilizar todas las opciones del programa: ejecutar las búsquedas, determinar cuántos elementos de la muestra se encuentran en un rango dado, cuántas veces aparece un valor dado, cuántos valores distintos hay en la muestra o cuál es el valor que más veces se presenta en la muestra. Para todos ellos se presenta el respectivo resultado, acompañado del tiempo que le tomó al computador calcularlo.
- Es posible ordenar varias veces la muestra, usando los distintos algoritmos.

Fig. 9 – Interfaz de usuario del manejador de muestras.

4.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
• Implementar los algoritmos de ordenamiento por selección, inserción e intercambio para tipos simples de datos.	• Aprender cómo funcionan esos tres algoritmos de ordenamiento.
• Implementar los algoritmos de búsqueda que se piden en el enunciado.	• Aprender en qué consiste la búsqueda binaria. Para las demás búsquedas ya tenemos los conocimientos y habilidades necesarios.
• Generar valores aleatorios en un rango.	• Estudiar el método <code>Math.random()</code> de Java y adaptarlo para que genere valores en un rango dado.
• Calcular el tiempo de ejecución de un método.	• Estudiar los métodos que ofrece Java para manejo de tiempo y la manera en que se pueden utilizar para medir el tiempo de ejecución de una parte de un programa.
• Construir un programa que funcione correctamente.	• Practicar la utilización de invariantes y la construcción de pruebas unitarias.

4.2. Comprensión de los Requerimientos

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

	<p><u>Objetivo:</u> Entender el problema del caso de estudio del manejador de muestras.</p> <p>(1) Lea detenidamente el enunciado del caso de estudio del manejador de muestras, (2) identifique y complete la documentación de los ocho requerimientos funcionales que allí aparecen.</p>								
Requerimiento funcional 1	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">Nombre</td><td style="padding: 5px;">R1 – Crear una nueva muestra.</td></tr> <tr> <td style="padding: 5px;">Resumen</td><td style="padding: 5px;"></td></tr> <tr> <td style="padding: 5px;">Entradas</td><td style="padding: 5px;"></td></tr> <tr> <td style="padding: 5px;">Resultado</td><td style="padding: 5px;"></td></tr> </table>	Nombre	R1 – Crear una nueva muestra.	Resumen		Entradas		Resultado	
Nombre	R1 – Crear una nueva muestra.								
Resumen									
Entradas									
Resultado									
Requerimiento funcional 2	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">Nombre</td><td style="padding: 5px;">R2 – Ordenar la muestra.</td></tr> <tr> <td style="padding: 5px;">Resumen</td><td style="padding: 5px;"></td></tr> <tr> <td style="padding: 5px;">Entradas</td><td style="padding: 5px;"></td></tr> <tr> <td style="padding: 5px;">Resultado</td><td style="padding: 5px;"></td></tr> </table>	Nombre	R2 – Ordenar la muestra.	Resumen		Entradas		Resultado	
Nombre	R2 – Ordenar la muestra.								
Resumen									
Entradas									
Resultado									

Requerimiento funcional 3	Nombre	R3 – Calcular el tiempo de ejecución de los algoritmos de búsqueda.
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 4	Nombre	R4 – Calcular el número de elementos en un rango.
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Calcular el número de veces que aparece un valor.
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 6	Nombre	R6 – Calcular el número de valores distintos.
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 7	Nombre	R7 – Encontrar el valor que más veces aparece.
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 8	Nombre	R8 – Dar información estadística de la muestra.
	Resumen	
	Entradas	
	Resultado	

4.3. Arquitectura de la Solución

En esta sección vamos a presentar los tres diagramas que definen la arquitectura de la solución propuesta: el diagrama de clases del modelo del mundo, el diagrama de clases de la interfaz de usuario y el diagrama de clases de las pruebas unitarias. En el modelo del mundo hay dos clases. La primera, llamada `Muestra`, representa una secuencia no ordenada de valores en un rango. Tiene en su interior tres atributos, tal como se muestra en la figura 10: un arreglo con los valores de la muestra (`valores`), el cual tiene reservado el espacio definido en el constructor, pero que sólo tiene un número dado de elementos presentes (`tamano`), todos estos valores en el rango `1..limiteSuperior`. En la figura 11 aparece un posible diagrama de objetos, en donde se puede observar una instancia de esta clase.

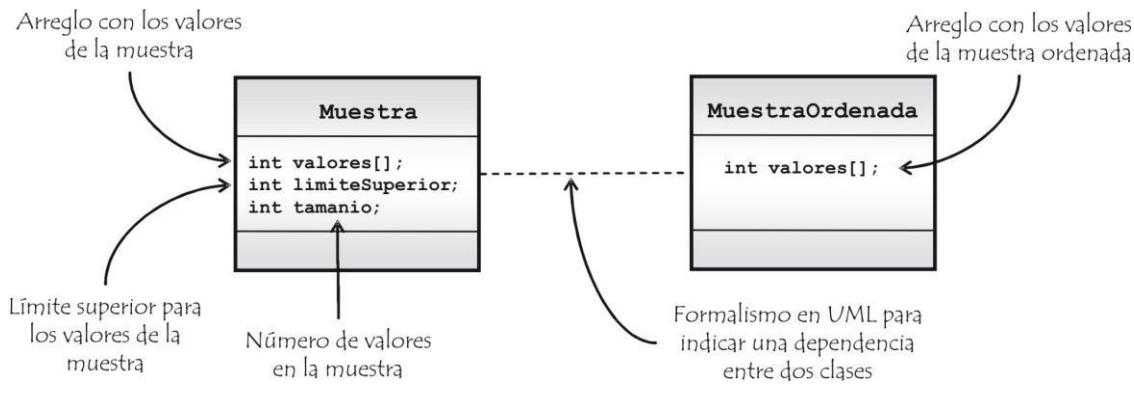


Fig. 10 – Diagrama de clases del modelo del mundo.

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

En el diagrama de clases se puede ver una asociación marcada con una línea punteada, que indica que las dos clases tienen una dependencia, aunque no exista una asociación directa entre ellas. En este caso la dependencia consiste en que vamos a tener un método de la clase `Muestra` que es capaz de crear instancias de la clase `MuestraOrdenada`. En la figura 11 se ilustra esto con un diagrama de objetos.

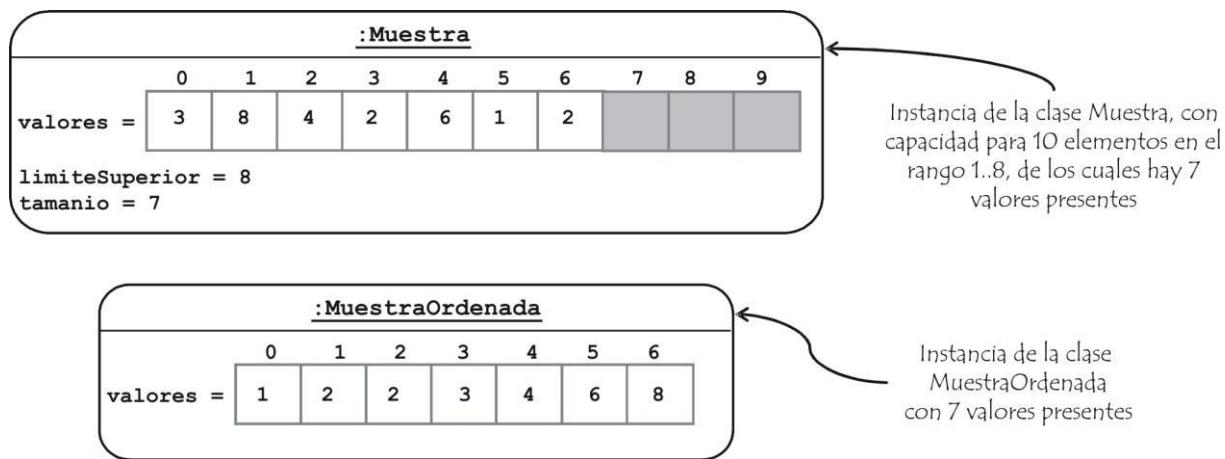


Fig. 11 – Diagrama de objetos como un ejemplo de un estado posible del modelo del mundo.

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



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

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

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

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

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

```

public class Muestra
{
    private void verificarInvariante( )
    {

    }
}

public class MuestraOrdenada
{
    private void verificarInvariante( )
    {

    }
}

```

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

Clase Muestra:

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

Clase MuestraOrdenada:

<code>MuestraOrdenada(int[] vals)</code>	→ Crea una muestra ordenada a partir de la información que recibe en un arreglo de valores enteros. La precondición dice que dicho arreglo no es nulo y que viene ordenado ascendenteamente,
<code>int darTamanio()</code>	→ Retorna el número de elementos de la muestra ordenada (que en este caso es igual al tamaño del arreglo de valores).
<code>int contarOcurrencias(int valor)</code>	→ Calcula y retorna el número de veces que aparece un valor dado en la muestra ordenada.
<code>int contarValoresDistintos()</code>	→ Calcula y retorna el número de valores distintos que hay en la muestra ordenada.
<code>int contarElementosEnRango(int inf, int sup)</code>	→ Calcula y retorna el número de elementos de la muestra ordenada que están en el rango inf...sup, incluyendo los dos límites.
<code>int darValorMasFrecuente()</code>	→ Calcula y retorna el valor que más veces aparece en la muestra ordenada.

```
boolean buscarBinario( int valor )
```

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

En la figura 12 aparece una parte del diagrama de clases de la interfaz de usuario, en la que se puede apreciar su relación con el modelo del mundo. En este diagrama se puede ver que existe una asociación opcional (0..1) hacia la clase *Muestra* y otra asociación también opcional hacia la clase *MuestraOrdenada*. Tan pronto el usuario genera una muestra, se crea la primera instancia. En el momento de utilizar cualquiera de los métodos de ordenamiento, se crea la instancia de la segunda clase. Existe también un atributo de tipo lógico (*botonesHabilitados*) en la clase de la ventana principal, que indica si se pueden ofrecer al usuario las opciones que trabajan sobre una muestra ordenada, cuyo valor se vuelve verdadero después de haber ordenado la muestra.

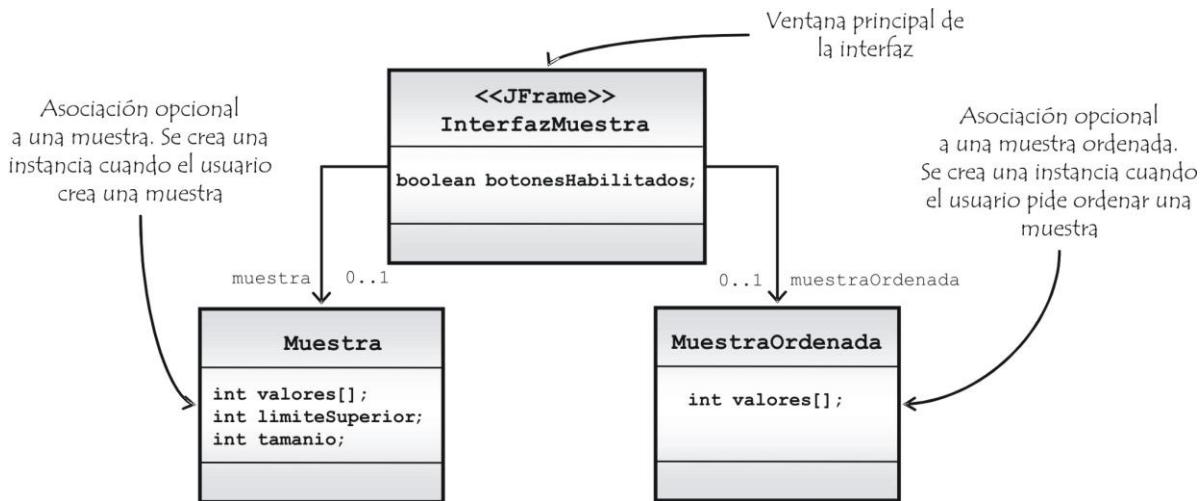


Fig. 12 – Diagrama de clases de la interfaz de usuario.

Para las pruebas unitarias construiremos dos clases (*MuestraTest* y *MuestraOrdenadaTest*), relacionadas con el modelo del mundo como aparece en la figura 13. Los escenarios y los casos de prueba son tema de una sección posterior. Por ahora nos contentamos con definir la parte estructural de la solución, e identificar las responsabilidades de cada uno de los componentes de ella.

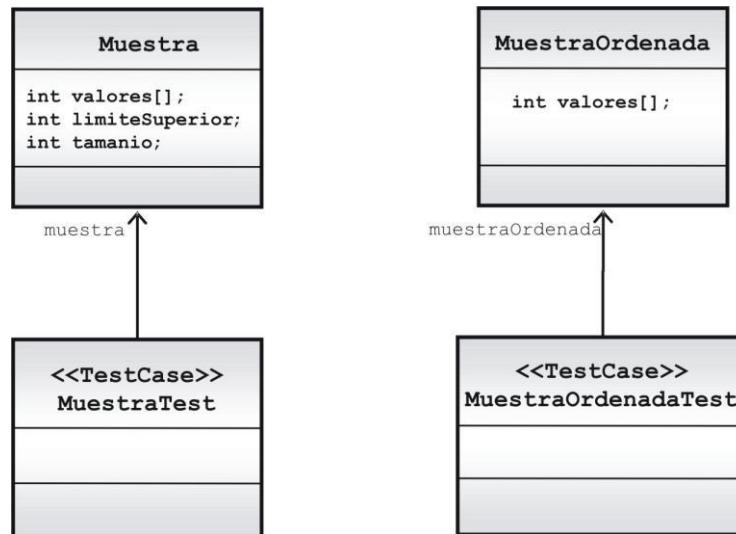


Fig. 13 – Diagrama de clases de las pruebas.

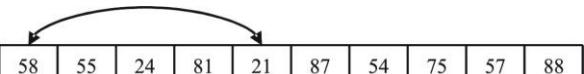
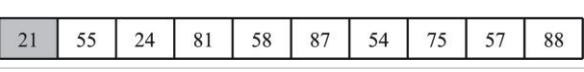
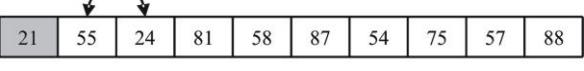
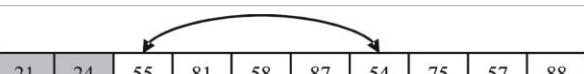
4.4. Algoritmos de Ordenamiento en Memoria Principal

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

Todos los algoritmos de ordenamiento que presentamos en esta sección manejan un orden ascendente y trabajan en una dirección determinada (izquierda a derecha o derecha a izquierda). Los cambios que se deben hacer para ordenar un arreglo de valores descendente o para trabajar en la dirección contraria son mínimos, por lo que no serán tratados aquí de manera aparte.

4.4.1. Ordenamiento por Selección

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

EJEMPLO #9	<p><u>Objetivo:</u> Mostrar el funcionamiento del algoritmo de ordenamiento por selección.</p> <p>En este ejemplo se muestra cada una de las etapas por las que pasa el algoritmo de ordenamiento por selección para ordenar ascendente un arreglo de valores de tipo simple.</p>																																																																																																				
	<table border="1" style="width: 100%; border-collapse: collapse;"> <tbody> <tr> <td>58</td><td>55</td><td>24</td><td>81</td><td>21</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> <tr> <td>21</td><td>55</td><td>24</td><td>81</td><td>58</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>55</td><td>81</td><td>58</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>54</td><td>81</td><td>58</td><td>87</td><td>55</td><td>75</td><td>57</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>54</td><td>55</td><td>58</td><td>87</td><td>81</td><td>75</td><td>57</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>54</td><td>55</td><td>57</td><td>87</td><td>81</td><td>75</td><td>58</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>54</td><td>55</td><td>57</td><td>58</td><td>81</td><td>75</td><td>87</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>54</td><td>55</td><td>57</td><td>58</td><td>75</td><td>81</td><td>87</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>54</td><td>55</td><td>57</td><td>58</td><td>75</td><td>81</td><td>87</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>54</td><td>55</td><td>57</td><td>58</td><td>75</td><td>81</td><td>87</td><td>88</td></tr> </tbody> </table>	58	55	24	81	21	87	54	75	57	88	21	55	24	81	58	87	54	75	57	88	21	24	55	81	58	87	54	75	57	88	21	24	54	81	58	87	55	75	57	88	21	24	54	55	58	87	81	75	57	88	21	24	54	55	57	87	81	75	58	88	21	24	54	55	57	58	81	75	87	88	21	24	54	55	57	58	75	81	87	88	21	24	54	55	57	58	75	81	87	88	21	24	54	55	57	58	75	81	87	88
58	55	24	81	21	87	54	75	57	88																																																																																												
21	55	24	81	58	87	54	75	57	88																																																																																												
21	24	55	81	58	87	54	75	57	88																																																																																												
21	24	54	81	58	87	55	75	57	88																																																																																												
21	24	54	55	58	87	81	75	57	88																																																																																												
21	24	54	55	57	87	81	75	58	88																																																																																												
21	24	54	55	57	58	81	75	87	88																																																																																												
21	24	54	55	57	58	75	81	87	88																																																																																												
21	24	54	55	57	58	75	81	87	88																																																																																												
21	24	54	55	57	58	75	81	87	88																																																																																												
1	 <table border="1" style="width: 100%; border-collapse: collapse;"> <tbody> <tr> <td>58</td><td>55</td><td>24</td><td>81</td><td>21</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> <tr> <td>21</td><td>55</td><td>24</td><td>81</td><td>58</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> </tbody> </table>  <table border="1" style="width: 100%; border-collapse: collapse;"> <tbody> <tr> <td>21</td><td>55</td><td>24</td><td>81</td><td>58</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>55</td><td>81</td><td>58</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> </tbody> </table>	58	55	24	81	21	87	54	75	57	88	21	55	24	81	58	87	54	75	57	88	21	55	24	81	58	87	54	75	57	88	21	24	55	81	58	87	54	75	57	88																																																												
58	55	24	81	21	87	54	75	57	88																																																																																												
21	55	24	81	58	87	54	75	57	88																																																																																												
21	55	24	81	58	87	54	75	57	88																																																																																												
21	24	55	81	58	87	54	75	57	88																																																																																												
2	 <table border="1" style="width: 100%; border-collapse: collapse;"> <tbody> <tr> <td>21</td><td>24</td><td>55</td><td>81</td><td>58</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>55</td><td>81</td><td>58</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> </tbody> </table>	21	24	55	81	58	87	54	75	57	88	21	24	55	81	58	87	54	75	57	88																																																																																
21	24	55	81	58	87	54	75	57	88																																																																																												
21	24	55	81	58	87	54	75	57	88																																																																																												
3	 <table border="1" style="width: 100%; border-collapse: collapse;"> <tbody> <tr> <td>21</td><td>24</td><td>55</td><td>81</td><td>58</td><td>87</td><td>54</td><td>75</td><td>57</td><td>88</td></tr> <tr> <td>21</td><td>24</td><td>54</td><td>81</td><td>58</td><td>87</td><td>55</td><td>75</td><td>57</td><td>88</td></tr> </tbody> </table>	21	24	55	81	58	87	54	75	57	88	21	24	54	81	58	87	55	75	57	88																																																																																
21	24	55	81	58	87	54	75	57	88																																																																																												
21	24	54	81	58	87	55	75	57	88																																																																																												

4	 	<p>→ Cuarta iteración: Incluimos en la parte ordenada el elemento 55, intercambiándolo por el 81. Con eso completamos ya cuatro elementos ordenados ascendenteamente.</p>
5	 	<p>→ Quinta iteración: Repetimos el mismo proceso anterior para intercambiar los elementos 57 y 58. Fíjese que en todo momento los valores de la parte ordenada son menores que todos los que se encuentran en la parte no ordenada.</p>
6	 	<p>→ Sexta iteración: Ya hay cinco elementos ordenados. Seleccionamos el menor de la parte no ordenada (58) y lo intercambiamos con el primero de la parte no ordenada (87).</p>
7	 	<p>→ Séptima iteración: Intercambiamos los elementos 75 y 81 para avanzar en el proceso de ordenamiento.</p>
8	 	<p>→ Octava iteración: Intercambiamos los elementos 87 y 81 para completar 8 elementos ordenados en el arreglo.</p>
9	 	<p>→ Novena iteración: En este caso no hay que hacer ningún intercambio, puesto que el menor de la parte no ordenada (87) ya se encuentra en la primera posición.</p> <p>→ Cuando hayamos ordenado todos los elementos menos uno terminamos el proceso, puesto que el restante (88) es necesariamente mayor que todos los que ya fueron incluidos en la parte ordenada.</p>

A continuación se muestra el código del método de la clase `Muestra` encargado de crear una instancia de la clase `MuestraOrdenada` usando la técnica de ordenamiento por selección:

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

- Inicialmente crea una copia de los valores de la muestra, para ordenarlos, usando el método `darCopiaValores()`.
- El ciclo externo del método lleva el índice “i” señalando el punto en el cual comienza la parte sin ordenar del arreglo. Comienza en 0 y termina cuando llega a la penúltima posición.
- El objetivo del ciclo interior es buscar el menor valor de la parte sin ordenar. Dicha parte comienza en la posición “i” y va hasta el final del arreglo. Al final del ciclo, deja en la variable “menor” el menor valor encontrado y en la variable “cual” su posición dentro del arreglo.
- Despues de haber localizado el menor elemento, lo intercambia con el que se encuentra en la casilla “i” del arreglo. Para esto utiliza una variable temporal “temp”.
- Cuando finalmente ha terminado de ordenar el arreglo, crea una instancia de la clase `MuestraOrdenada`.



TAREA #9

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

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



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

4.4.2. Ordenamiento por Intercambio (Burbuja)

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



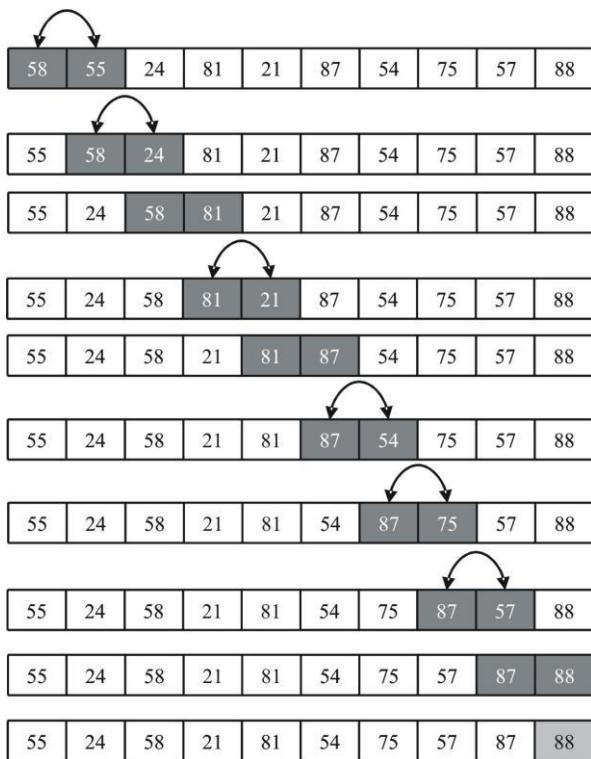
EJEMPLO #10

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

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

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

1



- En este ejemplo, tenemos inicialmente un arreglo no ordenado de 10 posiciones (primera fila de la figura).
- En la primera iteración recorremos el arreglo de izquierda a derecha intercambiando todo par de valores consecutivos que no se encuentren ordenados. Al final de la primera iteración, el mayor de los elementos (88) debe quedar en la última posición del arreglo, mientras todos los demás valores han avanzado un poco hacia su posición final.
- El valor 87, por ejemplo, alcanzó a avanzar 3 posiciones hacia su posición definitiva.
- Este mismo proceso se repite para la parte del arreglo que todavía no está ordenada (en blanco en la figura).
- El proceso termina cuando sólo queda un elemento en la zona no ordenada del arreglo (al comienzo), lo cual requiere nueve iteraciones en nuestro ejemplo (número de elementos menos uno).

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



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

A continuación se muestra el código del método de la clase `Muestra` encargado de crear una instancia de la clase `MuestraOrdenada` usando la técnica de ordenamiento por intercambio (burbuja):

```
public MuestraOrdenada ordenarBurbuja( )
{
    int[] arreglo = darCopiaValores( );

    for( int i = tamanio; i > 0; i-- )
    {
        for( int j = 0; j < i - 1; j++ )
        {

            if( arreglo[ j ] > arreglo[ j + 1 ] )
            {
                int temp = arreglo[ j ];
                arreglo[ j ] = arreglo[ j + 1 ];
                arreglo[ j + 1 ] = temp;
            }
        }
    }

    return new MuestraOrdenada( arreglo );
}
```

- Inicialmente crea una copia de los valores de la muestra, para ordenarlos, usando el método `darCopiaValores()`.
 - El ciclo externo va controlando con la variable “*i*” el punto hasta el cual hay que llevar el proceso de intercambio. Inicialmente va hasta el final de la secuencia, y va disminuyendo hasta que sólo queda un elemento (termina cuando *i*=0).
 - En el ciclo interno se lleva a cabo el proceso de intercambio con la variable “*j*”, comenzando desde la posición 0 hasta llegar al punto anterior al límite marcado por la variable “*i*”. Allí compara (y si es necesario intercambia) los valores de las posiciones “*j*” y “*j+1*”.
 - Cuando finalmente ha terminado de ordenar el arreglo, crea una instancia de la clase `MuestraOrdenada`.



TAREA #10

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

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

4.4.3. Ordenamiento por Inserción

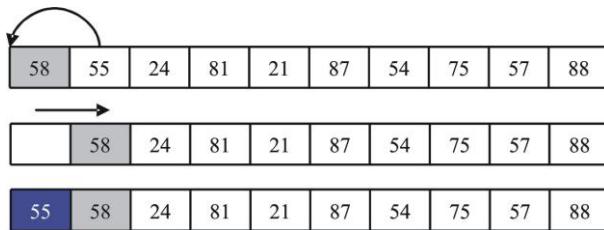
Uno de los métodos más naturales para ordenar una secuencia de valores consiste en separar la secuencia en dos grupos: una parte con los valores ordenados (inicialmente con un solo elemento) y otra con los valores por ordenar (inicialmente todo el resto). Luego, vamos pasando uno a uno los valores a la parte ordenada, asegurándonos que se vayan colocando ascendentemente, tal como se ilustra en el ejemplo 11.

EJEMPLO #11

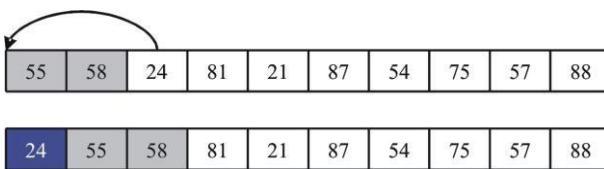
Objetivo: Mostrar el funcionamiento del algoritmo de ordenamiento por inserción.

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

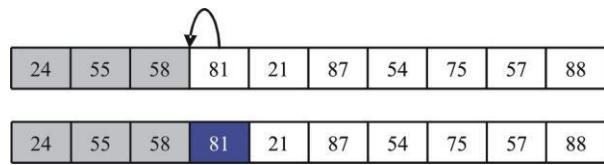
58	55	24	81	21	87	54	75	57	88
55	58	24	81	21	87	54	75	57	88
24	55	58	81	21	87	54	75	57	88
24	55	58	81	21	87	54	75	57	88
21	24	55	58	81	87	54	75	57	88
21	24	55	58	81	87	54	75	57	88
21	24	54	55	58	81	87	75	57	88
21	24	54	55	58	75	81	87	57	88
21	24	54	55	57	58	75	81	87	88
21	24	54	55	57	58	75	81	87	88



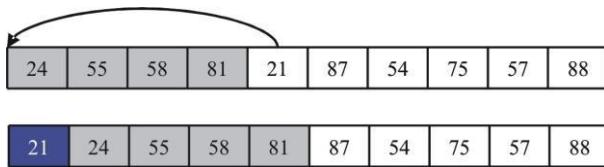
1



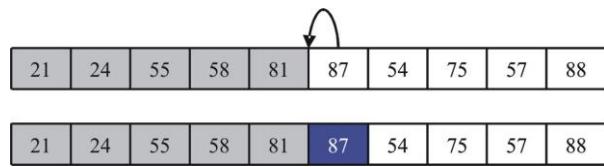
2



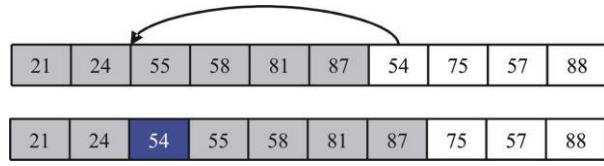
3



4



5



6

- En este ejemplo, tenemos inicialmente un arreglo no ordenado de 10 posiciones (primera fila de la figura).
- Al inicio, podemos suponer que en la parte ordenada del arreglo hay un elemento (58). No está todavía en su posición final, pero siempre es cierto que una secuencia de un solo elemento está ordenada.
- Luego tomamos uno a uno los elementos de la parte no ordenada y los vamos insertando ascendenteamente en la parte ordenada. Comenzamos con el 55 y vamos avanzando hasta insertar el valor 88.
- En la figura, en cada iteración aparece marcado el elemento que acaba de ser insertado.

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

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

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

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

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

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

7	 	<p>→ Séptima iteración: Vamos a insertar ahora el valor 75, para lo cual movemos hacia la derecha los valores 81 y 87.</p>
8	 	<p>→ Octava iteración: Para insertar el valor 57 movemos hacia la derecha los valores 58, 75, 81 y 87.</p>
9	 	<p>→ Novena iteración: La última iteración no implica ningún desplazamiento, puesto que el 88 es el mayor valor de todo el arreglo.</p>

A continuación se muestra el código del método de la clase `Muestra` encargado de crear una instancia de la clase `MuestraOrdenada` usando la técnica de ordenamiento por inserción:

<pre>public MuestraOrdenada ordenarInsercion() { int[] arreglo = darCopiaValores(); for(int i = 1; i < tamanio; i++) { for(int j = i; j > 0 && arreglo[j - 1] > arreglo[j]; j--) { int temp = arreglo[j]; arreglo[j] = arreglo[j - 1]; arreglo[j - 1] = temp; } } return new MuestraOrdenada(arreglo); }</pre>	<p>→ El ciclo externo tiene la responsabilidad de señalar con la variable “i” la posición del nuevo elemento que se va a insertar (hasta “i-1” está la parte ordenada).</p> <p>→ El ciclo interno va desplazando hacia abajo el elemento que se encontraba inicialmente en la posición “i”, hasta que encuentra la posición adecuada.</p>
---	---



TAREA #11

Objetivo: Utilizar la técnica de ordenamiento por inserción, para ordenar ascendente una secuencia de valores.

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



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

4.4.4. ¿Y Cuándo Ordenar?

Hay dos razones principales para ordenar información. La primera, para facilitar la interacción con el usuario. Si tenemos, por ejemplo, una lista de códigos de productos para que el usuario seleccione uno de ellos, es mucho más fácil para él encontrar lo que está buscando si le presentamos esta lista de manera ordenada. En ese caso no tenemos mayores opciones y debemos utilizar nuestros algoritmos de ordenamiento. La segunda razón es para hacer más eficientes los programas. Un método que busca información sobre una estructura que está ordenada gasta mucho menos tiempo que si los valores allí presentes no tienen ningún orden. El problema que se nos presenta aquí es que ordenar la información puede tomar un tiempo considerable, luego la pregunta que nos debemos hacer es, ¿cuándo es conveniente ordenar la información? La respuesta afortunadamente es simple y depende del número de búsquedas que se vayan a hacer. Si es una sola búsqueda, definitivamente no es un buen negocio ordenar antes la información, pero si vamos a hacer miles de ellas sobre el mismo conjunto de datos, la ganancia en tiempo amerita hacer un proceso previo de ordenamiento.

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

4.5. Algoritmos de Búsqueda en Memoria Principal

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

4.5.1. Búsqueda de un Elemento

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

TAREA #13



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

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

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

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

    }
}
```

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

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

EJEMPLO #12		<u>Objetivo:</u> Mostrar el funcionamiento del algoritmo de búsqueda binaria.																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																
		En este ejemplo se muestra cada una de las etapas por las que pasa el algoritmo de búsqueda binaria para localizar un elemento en un arreglo ordenado de valores.																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																
		<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 10px;">0</td><td style="text-align: center; width: 10px;">1</td><td style="text-align: center; width: 10px;">2</td><td style="text-align: center; width: 10px;">3</td><td style="text-align: center; width: 10px;">4</td><td style="text-align: center; width: 10px;">5</td><td style="text-align: center; width: 10px;">6</td><td style="text-align: center; width: 10px;">7</td><td style="text-align: center; width: 10px;">8</td><td style="text-align: center; width: 10px;">9</td><td style="text-align: center; width: 10px;">10</td><td style="text-align: center; width: 10px;">11</td><td colspan="2"></td></tr> <tr> <td style="text-align: center;">21</td><td style="text-align: center;">24</td><td style="text-align: center;">54</td><td style="text-align: center;">55</td><td style="text-align: center;">57</td><td style="text-align: center;">58</td><td style="text-align: center;">75</td><td style="text-align: center;">81</td><td style="text-align: center;">87</td><td style="text-align: center;">88</td><td style="text-align: center;">90</td><td style="text-align: center;">95</td><td colspan="2" rowspan="2"></td></tr> </table>														0	1	2	3	4	5	6	7	8	9	10	11			21	24	54	55	57	58	75	81	87	88	90	95																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																									
0	1	2	3	4	5	6	7	8	9	10	11																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																							
21	24	54	55	57	58	75	81	87	88	90	95																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																							
		<p>→ Este es el arreglo ordenado sobre el que vamos a hacer las búsquedas. Tiene 12 valores.</p>																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																
1		<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 10px;">0</td><td style="text-align: center; width: 10px;">1</td><td style="text-align: center; width: 10px;">2</td><td style="text-align: center; width: 10px;">3</td><td style="text-align: center; width: 10px;">4</td><td style="text-align: center; width: 10px;">5</td><td style="text-align: center; width: 10px;">6</td><td style="text-align: center; width: 10px;">7</td><td style="text-align: center; width: 10px;">8</td><td style="text-align: center; width: 10px;">9</td><td style="text-align: center; width: 10px;">10</td><td style="text-align: center; width: 10px;">11</td><td colspan="2"></td></tr> <tr> <td style="text-align: center;">21</td><td style="text-align: center;">24</td><td style="text-align: center;">54</td><td style="text-align: center;">55</td><td style="text-align: center;">57</td><td style="text-align: center;">58</td><td style="text-align: center;">75</td><td style="text-align: center;">81</td><td style="text-align: center;">87</td><td style="text-align: center;">88</td><td style="text-align: center;">90</td><td style="text-align: center;">95</td><td colspan="2"></td></tr> </table> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 10px;"></td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;">↓</td><td colspan="2"></td></tr> <tr> <td style="text-align: center;">inicio</td><td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align: center;"> </td><td colspan="2"></td></tr> <tr> <td style="text-align:</tr></table>	0	1	2	3	4	5	6	7	8	9	10	11			21	24	54	55	57	58	75	81	87	88	90	95																															↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓			inicio																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																									
0	1	2	3	4	5	6	7	8	9	10	11																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																							
21	24	54	55	57	58	75	81	87	88	90	95																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																							
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																							
inicio																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																		

<p>4</p>	<ul style="list-style-type: none"> → Puesto que el único elemento que queda en la zona de búsqueda es el 95 y es diferente del valor buscado, podemos afirmar en este punto que el 92 no está en el arreglo. → Fíjese que en sólo 4 iteraciones nos pudimos dar cuenta de que un valor no existía en un arreglo de 12 posiciones. ¿Nada mal, no?
----------	--

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

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

- Vamos a utilizar 3 variables enteras (inicio, medio y fin) para indicar en cada iteración la posición donde comienza la zona de búsqueda, la posición media y la posición en la que termina.
- Inicialmente “inicio” comienza en 0 y “fin” en la última posición del arreglo.
- La posición media se calcula sumando las posiciones de inicio y fin, y dividiendo el resultado por 2.
- Luego, hay tres casos: (1) si el valor es igual al de la mitad, ya lo encontramos, (2) si el valor es menor que el de la mitad, reposicionamos la marca final de la zona de búsqueda, (3) si el valor es mayor, movemos el inicio de la zona de búsqueda.
- Este proceso se repite mientras no hayamos encontrado el elemento y mientras exista algún elemento en el interior de la zona de búsqueda.

<p>TAREA #14</p>	<p><u>Objetivo:</u> Medir el tiempo de ejecución de los algoritmos de búsqueda e intentar identificar diferencias entre ellos.</p> <p>Localice en el CD el ejemplo <code>n7_muestra</code>, cópielo al disco duro y ejecútelo. Siga luego las instrucciones que aparecen a continuación.</p>
-------------------------	--

Algoritmo	5000	10000	20000	40000	60000	80000	100000	200000	→ Llene la tabla de tiempos de ejecución de los dos algoritmos de búsqueda, para muestras de distintos tamaños. Utilice una muestra entre 1 y 200.000.
Secuencial									
Binaria									

¿Qué se puede concluir de la tabla anterior?

4.5.2. Búsquedas en Estructuras Ordenadas

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

TAREA #15 	<p><u>Objetivo:</u> Escribir los métodos que se necesitan en el caso de estudio, para calcular algún valor sobre la muestra ordenada.</p> <p>Escriba los métodos que se piden a continuación en la clase <code>MuestraOrdenada</code>. Asegúrese de utilizar en su algoritmo el hecho de que los valores de la muestra se encuentran ordenados, para así tratar de encontrar una solución lo más eficiente posible.</p>
<pre>public int contarOcurrencias(int valor) { } } }</pre>	<p>→ Cuenta el número de veces que aparece un valor en la muestra ordenada.</p>
<pre>public int contarValoresDistintos() { } }</pre>	<p>→ Cuenta el número de valores distintos que hay en la muestra ordenada.</p>
<pre>public int contarElementosEnRango(int inf, int sup) { } }</pre>	<p>→ Cuenta el número de elementos que hay en un rango de valores (incluidos los extremos).</p>

```
public int darValorMasFrecuente( )
{
    }

}
```

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

4.6. Generación de Datos y Medición de Tiempos

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

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

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

```
public void generarValores( )
{
    for( int i = 0; i < valores.length; i++ )
    {
        double aleatorio = Math.random() * limiteSuperior;
        valores[ i ] = ( int )aleatorio + 1;
    }
    tamano = valores.length;
}
```

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

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

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

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

- `System.currentTimeMillis()`: Retorna un valor de tipo `long` con la fecha actual, expresada como el número de milisegundos que han transcurrido desde el 1 de enero de 1970 a las 0 horas.
- `System.nanoTime()`: Retorna un valor de tipo `long` que mide el tiempo en nanosegundos (la milmillonésima parte de un segundo), con respecto a un sistema no especificado de tiempo. Este método sólo está disponible desde la versión 5 de Java.

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

```
public long medir( )
{
    long t1 = System.currentTimeMillis( );
    muestra.metodo( );
    long t2 = System.currentTimeMillis( );
    return t2 - t1;
}
```

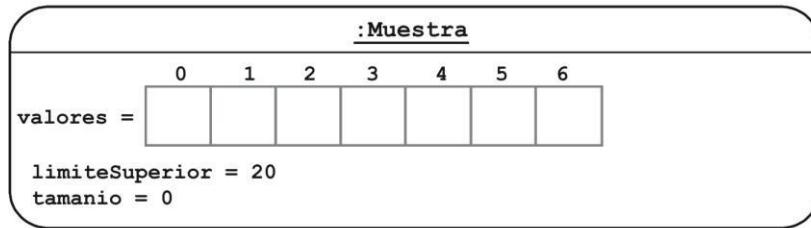
- Suponga que estamos interesados en medir el tiempo que toma la ejecución de un método de la clase `Muestra`.
- Primero calculamos el tiempo antes de comenzar la ejecución y almacenamos este valor en la variable “t1”.
- Luego ejecutamos el método y volvemos a medir el tiempo cuando éste termine. Este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

4.7. Pruebas Unitarias Automáticas

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

Vamos a trabajar sobre dos escenarios, los cuales se muestran en la figura 14. El primero de ellos se encuentra vacío y el segundo ya está lleno (llegó a su capacidad que es de 10 elementos). Sobre el primero haremos las pruebas del método que agrega valores, mientras que sobre el segundo probaremos los métodos de ordenamiento.

Escenario 1:



Escenario 2:

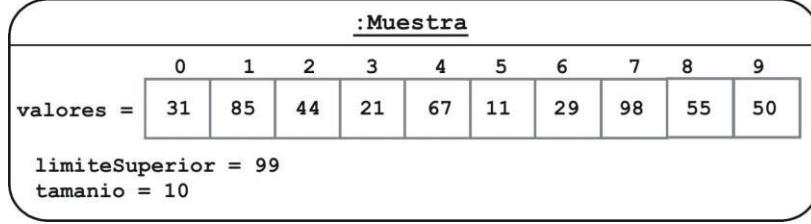


Fig. 14 – Escenarios de prueba para la clase `Muestra`.

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

TAREA #16

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

Escriba la declaración completa de la clase `MuestraTest`, siguiendo el diagrama de clases que se presenta en la figura 13. Luego, implemente los métodos necesarios para construir los dos escenarios de la figura 14. No olvide que en la sección 4.3 están definidos los métodos de la clase `Muestra` que puede utilizar con este fin.

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

EJEMPLO #13

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

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

Prueba No. 1

Objetivo: Probar que el método `agregarDatos()` es capaz de incluir nuevos valores en la muestra. Puesto que la precondición del método exige que exista espacio en la muestra y que el valor se encuentre en el rango válido, no es mucho lo que se debe probar.

Escenario: 1

Valores de entrada:

12, 5, 7, 1, 10, 1, 19

Resultado:

La muestra debe quedar así: [12, 5, 7, 1, 10, 1, 19]

El tamaño de la muestra debe ser 7

Prueba No. 2	Objetivo: Probar que el método <code>generarValores()</code> llena de manera correcta la muestra con valores aleatorios.	
Escenario: 1	Valores de entrada: Ninguno	Resultado: El tamaño de la muestra debe ser 7 No hay necesidad de verificar nada más, puesto que el invariante se encarga de ello.
Prueba No. 3	Objetivo: Probar que el método <code>ordenarInsercion()</code> ordena de manera correcta la muestra. Aquí no hay manera de verificar que el método haya aplicado la técnica de ordenamiento por inserción, sino nos tenemos que contentar con comprobar que al final haya creado una muestra ordenada correcta, con los mismos valores de la muestra original.	
Escenario: 2	Valores de entrada: Ninguno	Resultado: La muestra ordenada es: [11, 21, 29, 31, 44, 50, 55, 67, 85, 98]
Prueba No. 4	Objetivo: Probar que el método <code>buscarSecuencial()</code> es capaz de localizar los elementos presentes en la muestra.	
Escenario: 2	Valores de entrada: 31 (el primero)	Resultado: Verdadero
Escenario: 2	Valores de entrada: 50 (el último)	Resultado: Verdadero
Escenario: 2	Valores de entrada: 11 (en la mitad)	Resultado: Verdadero
Escenario: 2	Valores de entrada: 40 (inexistente)	Resultado: Falso

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

TAREA #17 	<u>Objetivo:</u> Escribir los métodos que ejecutan las pruebas definidas en el ejemplo 12. Implemente en la clase <code>MuestraTest</code> los cuatro métodos que llevan a cabo las pruebas definidas en el ejemplo 13. No olvide invocar el método que crea el escenario respectivo.
<pre>public class MuestraTest extends TestCase { private Muestra muestra; public void testAgregarDatos() { </pre>	

```
public void testGenerarValores( )
{
}

public void testOrdenarInsercion( )
{
}

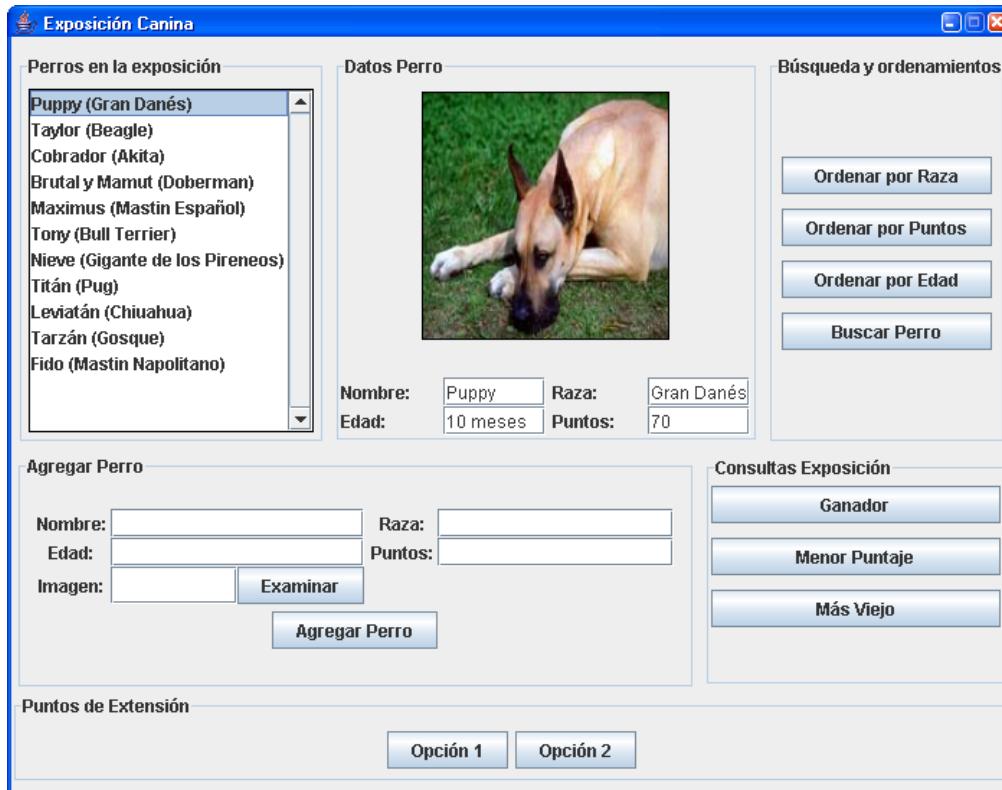
public void testBuscarSecuencial( )
{
}

}
```

5. Caso de Estudio #3: Una Exposición Canina

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

En la figura 15 aparece la interfaz de usuario que se espera tenga el programa. Este debe ofrecer los siguientes servicios al usuario: (1) mostrar la lista de los perros registrados en la exposición, ordenada por raza, puntos o edad, (2) mostrar la información de un perro específico, (3) registrar un nuevo perro, (4) localizar un perro por su nombre, (5) buscar el perro ganador de la exposición (el que tiene un mayor puntaje asignado), (6) buscar el perro con menor puntaje y (7) buscar el perro más viejo de todos (con mayor edad).



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

Fig. 15 – Interfaz de usuario para el caso de estudio de la exposición canina.

El programa va a manejar un esquema muy simple de persistencia, en el cual el estado inicial del programa debe cargarse desde un archivo de propiedades llamado “perros.txt”, localizado en el directorio “data”, y el cual tiene el formato que se ilustra en el siguiente cuadro:

```
total.perros = 2

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

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

- La primera propiedad del archivo (“total.perros”) define el número de perros presentes en el archivo.
- Para cada uno de los perros tenemos 5 datos: nombre, raza, imagen, puntos y edad.
- Cada uno de los datos de un perro se encuentra asociado con una propiedad.

5.1. Objetivos de Aprendizaje

¿Qué tenemos que hacer en el caso de estudio?	¿Qué tenemos que aprender o reforzar?
<ul style="list-style-type: none"> • Presentar en la interfaz de usuario la lista de los perros de la exposición. 	<ul style="list-style-type: none"> • Estudiar el componente <code>JList</code> de Java, aprender a integrarlo a una interfaz de usuario y a tomar los eventos que sobre los elementos de la lista genere el usuario (clic sobre un perro).
<ul style="list-style-type: none"> • Ordenar por distintos conceptos un grupo de longitud variable de objetos. 	<ul style="list-style-type: none"> • Adaptar lo visto en el caso anterior para hacer los ordenamientos pedidos, por distintos conceptos y teniendo en cuenta que vamos a manejar objetos en lugar de tipos simples.

5.2. Comprensión de los Requerimientos

 <p>TAREA #18</p>	<p><u>Objetivo:</u> Entender el problema del caso de estudio de la exposición canina.</p> <p>(1) Lea detenidamente el enunciado del caso de estudio de la exposición canina, (2) identifique y complete la documentación de los requerimientos funcionales que allí aparecen.</p>
Requerimiento funcional 1	Nombre R1 – Mostrar la lista de perros de la exposición.
	Resumen
	Entradas
	Resultado
Requerimiento funcional 2	Nombre R2 – Mostrar la información de un perro.
	Resumen
	Entradas
	Resultado
Requerimiento funcional 3	Nombre R3 – Registrar un nuevo perro.
	Resumen
	Entradas
	Resultado

Requerimiento funcional 4	Nombre	R4 – Localizar un perro por su nombre.
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 5	Nombre	R5 – Buscar el perro ganador de la exposición.
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 6	Nombre	R6 – Buscar el perro con menor puntaje.
	Resumen	
	Entradas	
	Resultado	
Requerimiento funcional 7	Nombre	R7 – Buscar el perro más viejo.
	Resumen	
	Entradas	
	Resultado	

5.3. Arquitectura de la Solución

El mundo del problema de este caso es muy simple, como se muestra en la figura 16. Sólo hay dos entidades: la primera que representa un perro (clase `Perro`), con todas sus características, y la segunda, la exposición (clase `ExposicionPerros`), que tiene un vector de perros como su única asociación.

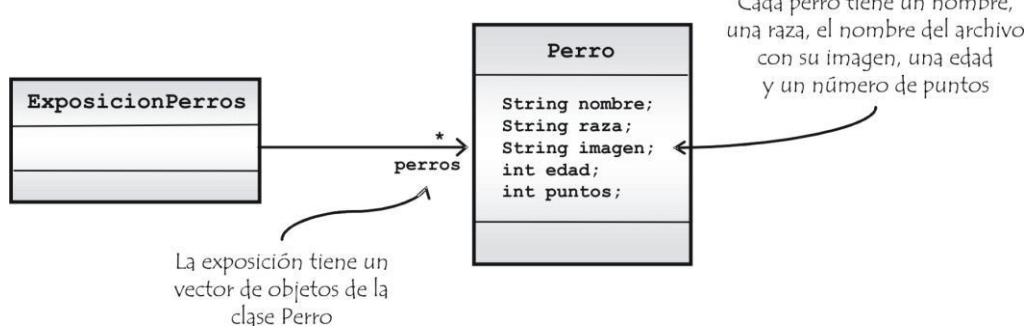


Fig. 16 – Diagrama de clases del modelo del mundo.

Los invariantes de estas clases son:

- Clase **ExposicionPerros**:
El vector de perros está inicializado (`perros != null`)
No hay dos perros en el vector que tengan el mismo nombre.
- Clase **Perro**:
El nombre del perro es una cadena no nula de caracteres (`nombre != null`)
La raza del perro es una cadena no nula de caracteres (`raza != null`)
La imagen del perro (nombre del archivo) está dada por una cadena no nula de caracteres (`imagen != null`)
La edad del perro es un valor positivo (`edad > 0`)
El perro tiene un número no negativo de puntos (`puntos >= 0`)

Para las pruebas tendremos dos clases, como se muestra en la figura 17: una para probar la clase **Perro** y otra para probar la clase **ExposicionPerros**. El tema de las pruebas no será abordado en este caso de estudio, por lo que se recomienda revisar el contenido del CD para estudiar allí los escenarios utilizados y los casos de prueba definidos.

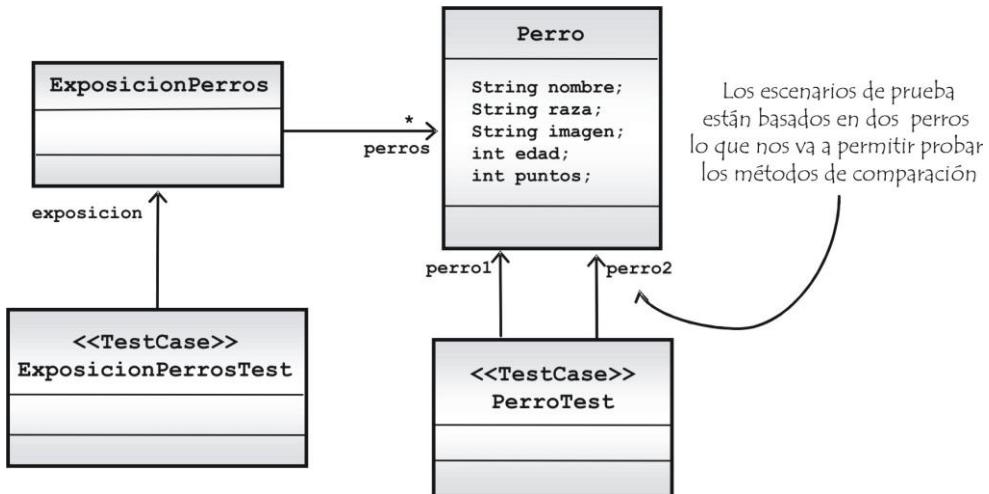


Fig. 17 – Diagrama de clases de las pruebas.

5.4. Comparación de Objetos por Múltiples Criterios

Tanto los algoritmos de ordenamiento como los algoritmos de búsqueda se basan en la capacidad que ellos deben tener de comparar dos elementos y decidir si son iguales, mayores o menores. Sin eso no podrían trabajar. En el caso de las muestras, en los cuales teníamos valores de tipo simple (enteros), utilizamos los operadores relacionales (`==`, `<`, `>`, `<=`, `>=`, `!=`) que provee Java para tal fin. ¿Cómo hacer ahora que queremos utilizar los mismos algoritmos, pero aplicados a objetos? ¿Cómo saber si un perro es mayor que otro cuando se quieren ordenar por raza?

Comencemos tratando el problema de la igualdad de objetos. Aquí tenemos dos niveles posibles: el primero, se refiere al caso en el cual dos objetos son físicamente el mismo. Si eso es lo que queremos establecer, podemos utilizar el operador `==` de Java. Al decir `obj1 == obj2` estamos preguntando si las variables `obj1` y `obj2` están haciendo referencia al mismo objeto en memoria. El segundo caso posible es cuándo queremos saber si dos objetos son iguales bajo algún concepto (por ejemplo, si dos perros tienen la misma raza). En ese caso, es necesario escribir un método llamado `equals()`, que indique si dos objetos son iguales sin ser necesariamente el mismo. Así es, por ejemplo, como comparamos dos cadenas de caracteres, ya que la clase `String` define este método. Al decir `cad1.equals(cad2)` estamos tratando de establecer si el “contenido” de `cad1` es igual al “contenido” de `cad2`, aunque sean objetos distintos.

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

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

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

Nosotros vamos a utilizar el mismo enfoque, generalizando la idea de tener un método de comparación por cada criterio de orden que pueda tener un objeto. En el caso de la exposición canina, los perros se deben saber comparar por nombre, por raza, por edad y por puntos. Para cada uno de esos criterios, la clase debe proveer el respectivo método de comparación. Es así como en la clase `Perro` vamos a encontrar los siguientes métodos:

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

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

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

EJEMPLO #14  <p><u>Objetivo:</u> Escribir los métodos de comparación de una clase usando distintos criterios.</p> <p>En este ejemplo implementamos los cuatro métodos de comparación que necesita la clase <code>Perro</code>, cuya especificación se dio anteriormente.</p>	<pre>public int compararPorNombre(Perro p) { return nombre.compareToIgnoreCase(p.nombre); }</pre>	<ul style="list-style-type: none"> → Este método compara dos perros usando como criterio su nombre. → La solución consiste en usar los nombres de los dos perros y delegar esta responsabilidad al método <code>compareToIgnoreCase()</code> de la clase <code>String</code>. → Puesto que el parámetro “p” pertenece a la misma clase en la cual estamos definiendo el método (clase <code>Perro</code>), tenemos derecho a hacer referencia de manera directa a sus atributos. En este caso, el atributo “nombre” contiene el nombre del perro sobre el cual se hace la invocación del método y el atributo “p.nombre” hace referencia al nombre del perro que llega como parámetro.
<pre>public int compararPorRaza(Perro p) { return raza.compareToIgnoreCase(p.raza); }</pre>		<ul style="list-style-type: none"> → Este método compara dos perros usando como criterio su raza. → Utilizamos el mismo tipo de solución del método anterior, pero usando el atributo que contiene la raza de cada uno de los perros.
<pre>public int compararPorPuntos(Perro p) { if(puntos == p.puntos) return 0; else if(puntos > p.puntos) return 1; else return -1; }</pre>		<ul style="list-style-type: none"> → Este método compara dos perros usando como criterio el número de puntos conseguidos en la exposición. → Si son iguales retorna 0, si el perro tiene más puntos que el parámetro p retorna 1, en cualquier otro caso retorna -1.
<pre>public int compararPorEdad(Perro p) { if(edad == p.edad) return 0; else if(edad > p.edad) return 1; else return -1; }</pre>		<ul style="list-style-type: none"> → Este método compara dos perros usando como criterio la edad de los perros. → Si tienen la misma edad retorna 0, si el perro es más viejo que el parámetro p retorna 1, en cualquier otro caso retorna -1.

5.5. Métodos de Ordenamiento y Búsqueda de Objetos

En esta sección nos vamos a dedicar a adaptar los métodos de ordenamiento y búsqueda que vimos en el caso anterior, para que puedan trabajar sobre objetos y puedan manejar distintos criterios de ordenamiento. En el ejemplo 15 mostraremos la adaptación de la técnica de ordenamiento por inserción, aplicada al problema de ordenar los perros por puntaje. Luego plantearemos en términos de tareas al lector la adaptación de los demás algoritmos de ordenamiento y el algoritmo de búsqueda binaria.

EJEMPLO #15

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

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

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

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

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

- Al tratar de adaptar el algoritmo de ordenamiento por inserción que escribimos antes, vamos a encontrar tres dificultades
- La primera es que la sintaxis para recuperar un elemento es más pesada, por lo que es conveniente agregar variables temporales (“`porInsertar`” y “`actual`”).
- La segunda es que no podemos asignar directamente los valores, luego debemos pasar por el método `set()` de la clase `ArrayList`.
- La tercera es que no podemos comparar los objetos con el operador relacional `==`, lo que nos obliga a utilizar el método `compararPorPuntos()`.
- De resto, la traducción es directa.

TAREA #19

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

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

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

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

```
}
```

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

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

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

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

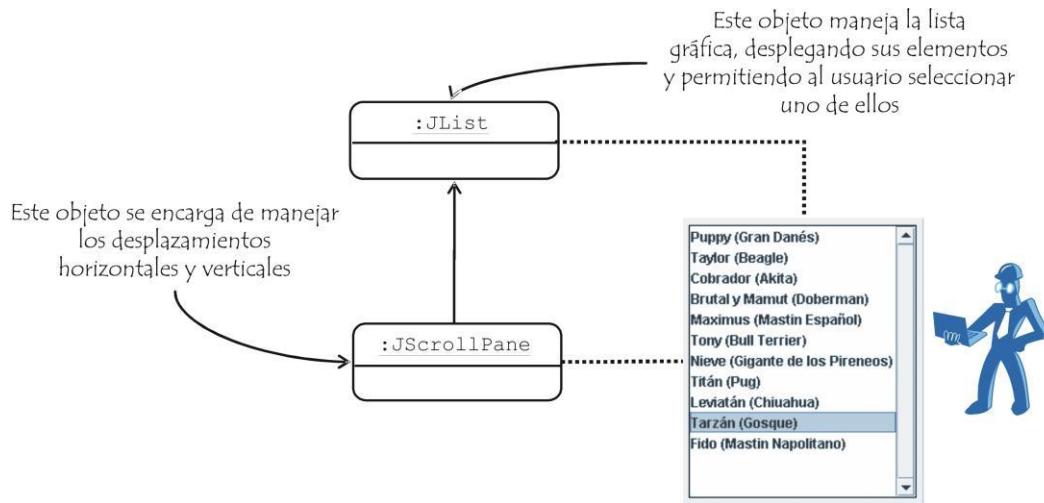
5.6. Manejo de Grupos de Valores en la Interfaz de Usuario

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

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

- ¿Cómo crear y configurar una lista gráfica?

<pre><code>public class PanelListaPerros extends JPanel { // ----- // Atributos // ----- private JList listaPerros; private JScrollPane scroll;</code></pre>	<ul style="list-style-type: none"> → Para ilustrar la declaración, creación y configuración de la lista gráfica, mostraremos el código del panel que lo contiene. → Al igual que con los demás componentes gráficos, debemos declarar un atributo por cada uno que queramos incluir en la interfaz. → Declaramos una lista gráfica (<code>listaPerros</code>) y un componente de desplazamiento (<code>scroll</code>).
<pre><code>public PanelListaPerros() { ... listaPerros = new JList(); scroll = new JScrollPane(listaPerros); listaPerros.setSelectionMode(ListSelectionModel.SINGLE_SELECTION); scroll.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER); scroll.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS); ... add(scroll, BorderLayout.CENTER); }</code></pre>	<ul style="list-style-type: none"> → En el constructor del panel creamos inicialmente cada una de las instancias. → Al crear la instancia del componente de desplazamiento pasamos como parámetro la lista gráfica que debe contener. En la figura 18 se ilustra la relación entre las instancias de estas dos clases. → Después de creadas las instancias pasamos a configurarlas. Con el método <code>setSelectionMode()</code> y el parámetro <code>SINGLE_SELECTION</code> indicamos que es una lista gráfica de selección simple (sólo se puede escoger un elemento). → Con los métodos <code>setHorizontalScrollBarPolicy()</code> y <code>setVerticalScrollBarPolicy()</code> indicamos que no queremos una barra de desplazamiento horizontal, pero si una barra de desplazamiento vertical. → Sólo agregamos al panel el componente de desplazamiento, puesto que la lista gráfica ya está incluida dentro de éste. → Existen muchos otros métodos de configuración en estas dos clases. Invitamos al lector a visitar la respectiva documentación y a ensayar su efecto sobre el programa del caso de estudio.

Fig. 18 – Relación entre la clase `JList` y la clase `JScrollPane`.

- ¿Cómo asignarle datos a una lista gráfica?

```
public void refrescarLista( ArrayList nuevaLista )
{
    listaPerros.setListData( nuevaLista.toArray( ) );
    listaPerros.setSelectedIndex( 0 );
}
```

- Vamos a crear un método en la clase del panel, que va a utilizar la información que recibe como parámetro (un vector), para construir la lista de elementos que debe visualizar. Este método se va a llamar desde la ventana principal cada vez que un requerimiento cambie la lista de perros.
- El método `setListData()` de la clase `JList`, recibe como parámetro un arreglo de objetos, y a partir de esta información despliega cada uno de ellos en la interfaz. Utilizamos el método `toArray()` que convierte un `ArrayList` en un arreglo.
- Con el método `setSelectedIndex()` le decimos a la lista que el primer elemento que ella contenga debe aparecer como seleccionado.

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

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

    private String nombre;
    private String raza;

    ...

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

- La respuesta a la última pregunta es que esta responsabilidad la tiene que asumir la clase de los objetos que van a ser incluidos en la lista gráfica. En nuestro caso, la clase `Perro`.
- La clase `JList` se va a contentar con invocar el método `toString()` sobre cada uno de los objetos que va a presentar en su interior. Si la clase no tiene definido dicho método, se utiliza un método por defecto que tiene Java.
- En nuestro caso, el método `toString()` de la clase `Perro` retorna el nombre del perro, concatenado con la raza del perro entre paréntesis.

- ¿Cómo tomar el elemento que el usuario seleccionó?

```

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

    private InterfazExposicionCanina principal;
    private JList listaPerros;
    ...

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

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

```

- Debemos manejar el panel como un panel activo: esto implica una asociación hacia la ventana principal inicializada en el constructor.
- Para que un panel pueda manejar los eventos de una lista gráfica, debe incluir en su declaración la cláusula “`implements ListSelectionListener`”, e implementar el método `valueChanged()`, con la firma exacta que aparece en el ejemplo.
- Dicho método será invocado cada vez que el usuario cambie la selección en la lista (ya sea por un clic sobre un elemento o por un desplazamiento con las flechas).
- Con el método `addListSelectionListener()` definimos que es el propio panel quien va a responder a los eventos de la lista.
- Allí utilizamos el método `getSelectedValue()` para tomar el objeto que fue seleccionado por el usuario (no la cadena de caracteres que aparece desplegada, sino el objeto incluido en la lista).
- También podemos usar los métodos `isSelectionEmpty()` para saber si hay algún valor seleccionado, `clearSelection()` para no dejar ningún elemento seleccionado o `getSelectedIndex()` para saber el índice dentro de la lista del elemento seleccionado.

6. Glosario de Términos

GLOSARIO	Complete las siguientes definiciones como una manera de resumir los conceptos aprendidos en el nivel. Asegúrese de entender bien el significado de cada uno de estos términos, puesto que son la base del trabajo de los niveles que siguen en el libro.
Requerimiento funcional	
Modelo del mundo	
Diagrama de clases	

Diagrama de objetos	
Invariante de clase	
Instrucción assert	
Pruebas unitarias automáticas	
Escenario de prueba	
Caso de prueba	
Framework JUnit	
Ejecutor de pruebas	
Clase TestCase	
Ordenamiento por inserción	
Ordenamiento por intercambio	
Ordenamiento por selección	
Búsqueda secuencial	
Búsqueda binaria	
Método Math.random()	
Clase JList	
Clase JScrollPane	

7. Hojas de Trabajo



7.1. Hoja de Trabajo #1: Bolsa de Empleo

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

Una empresa del sector financiero quiere construir un programa para manejar su bolsa de empleo. Se espera que dicho programa le dé apoyo al proceso de selección de personal, administrando las hojas de vida de los aspirantes a los diferentes cargos en los cuales la empresa tiene vacantes. Para la bolsa de empleo es importante poder clasificar los mejores aspirantes de acuerdo a unos criterios previamente definidos (años de experiencia, edad y profesión), haciendo la suposición de que no hay dos aspirantes con el mismo nombre. El programa sólo contempla aspirantes de cuatro profesiones posibles: “administrador”, “ingeniero industrial”, “contador” y “economista”.

La aplicación debe permitir (1) agregar nuevas hojas de vida de aspirantes, (2) mostrar la lista de aspirantes, (3) mostrar la información detallada de un aspirante, (4) buscar por nombre del aspirante, (5) permitir ordenar la lista de aspirantes por los diferentes criterios: años de experiencia, edad y profesión, (6) localizar el aspirante con mayor experiencia, (7) localizar el aspirante más joven, (8) contratar un aspirante (eliminarlo de la lista de aspirantes de la bolsa) y (9) eliminar aquellos aspirantes cuya experiencia sea menor a una cantidad de años especificada.

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

Aspirantes Registrados en la Bolsa	
Pilar Duque - Ingeniero Industrial	
Carlos Méndez - Ingeniero Industrial	
Andres Pérez - Administrador	
Marcela Otero - Administrador	
Jorge Campos - Ingeniero Industrial	
Ana Rojas - Economista	
Juan Rodríguez - Ingeniero Industrial	

Datos del Aspirante	
Nombre:	Pilar Duque
Edad:	36 años
Profesión:	Ingeniero Industrial
Experiencia:	10 año(s)
Teléfono:	2145500

Agregar Aspirante	
Nombre:	<input type="text"/>
Edad:	<input type="text"/>
Profesión:	<input type="text"/> Administrador
Años experiencia:	<input type="text"/>
Imagen:	<input type="file"/> Examinar
Teléfono:	<input type="text"/>
<input type="button" value="Agregar hoja de vida"/>	

Búsqueda y Ordenamiento	
<input type="button" value="Ordenar por experiencia"/>	
<input type="button" value="Ordenar por Edad"/>	
<input type="button" value="Ordenar por profesión"/>	
<input type="button" value="Buscar Aspirante"/>	

Consultas	
<input type="button" value="Más Joven"/>	
<input type="button" value="Mayor Edad"/>	
<input type="button" value="Mayor Experiencia"/>	
<input type="button" value="Contratar"/>	
<input type="button" value="Eliminar por Experiencia"/>	

Puntos de Extensión

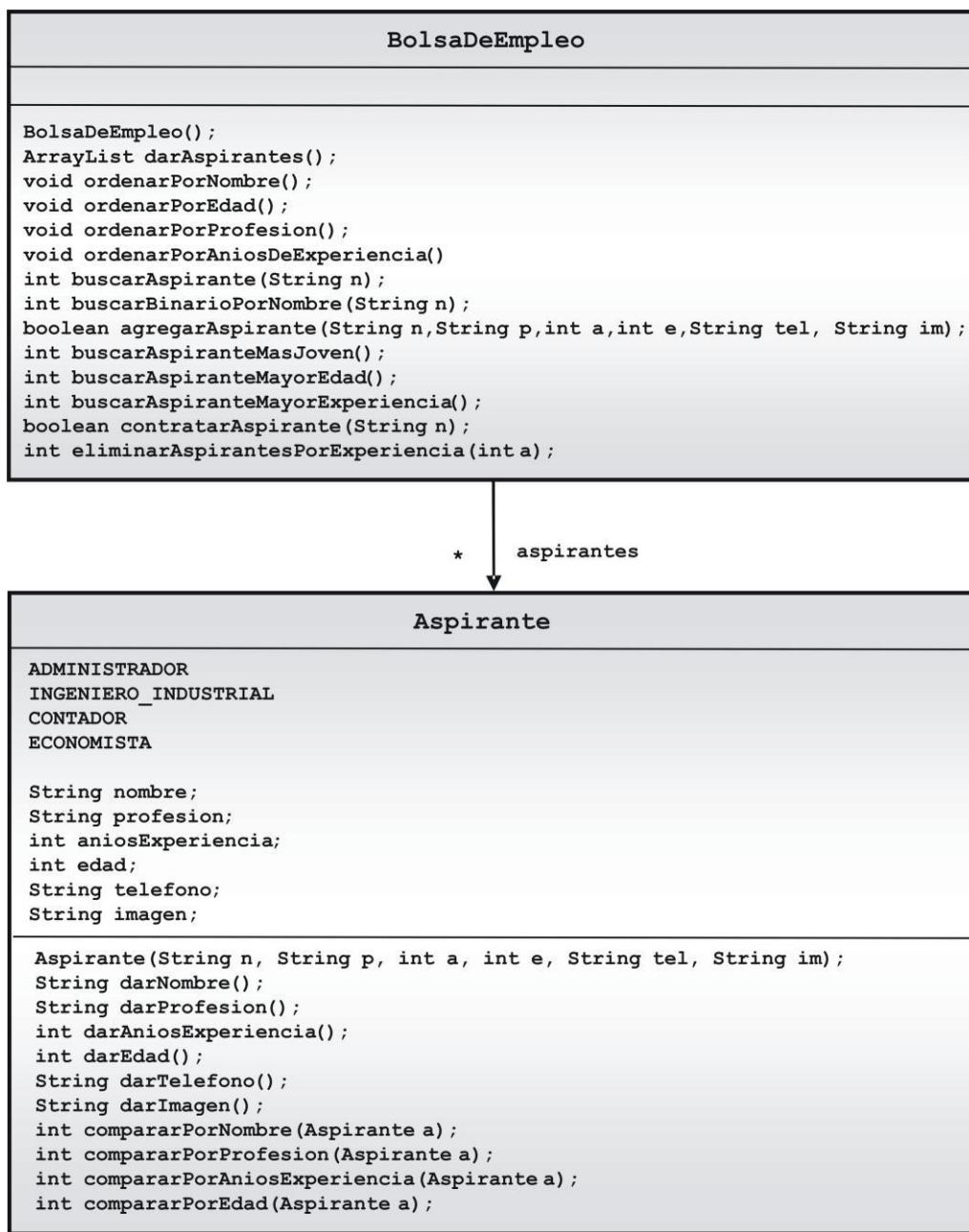
Opción 1 Opción 2

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

Requerimientos funcionales: Especifique los nueve requerimientos funcionales descritos en el enunciado	
Nombre:	R1 – Agregar una hoja de vida.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R2 – Mostrar la lista de aspirantes de la bolsa de empleo.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R3 – Consultar los datos de un aspirante.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R4 – Localizar un aspirante.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R5 – Ordenar la lista de aspirantes por distintos conceptos.
Resumen:	
Entradas:	
Resultados:	

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

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



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

Clase	Atributo	A/D	Restricción o relación

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

	<pre> public class BolsaDeEmpleo { private void verificarInvariant() { } private int contarVecesAparece(String nombre) { } } } </pre>
<p>Este método verifica el invariante de la clase.</p> <p>Se apoya en un segundo método que cuenta el número de veces que aparece un nombre dado dentro de la bolsa de empleo.</p>	<pre> public class BolsaDeEmpleo { public boolean agregarAspirante(String nombre, String profesion, int experiencia, int edad, String telefono, String imagen) { } } } </pre> <p>Este método agrega un aspirante a la bolsa de empleo, retornando verdadero si se pudo agregar o falso en caso contrario (porque ya existe alguien con ese mismo nombre). Utilice el constructor de la clase <code>Aspirante</code> que aparece definido en el diagrama de clases. Utilice también el método auxiliar del punto anterior.</p>

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

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

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

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



7.2. Hoja de Trabajo #2: Venta de Vehículos

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

Se quiere construir un programa para un negocio de venta de vehículos usados, en el cual se ofrecen a los clientes automóviles, buses, motos y camiones. Además del tipo de cada vehículo, se conoce la marca, el nombre del modelo, el año de producción, el número de ejes, la cilindrada, el valor y una foto del mismo. Por condiciones del negocio, se estableció que no hay dos vehículos que correspondan al mismo modelo y año.

La aplicación debe permitir al usuario las siguientes operaciones: (1) obtener la lista de todos los vehículos que están a la venta, (2) obtener la información detallada de un vehículo dado, (3) agregar un nuevo vehículo a la venta, (4) ordenar la lista de vehículos por modelo, por marca o por año, (5) hacer una búsqueda usando el modelo y el año del vehículo, (6) comprar un vehículo (eliminarlo de la lista de vehículos que están a la venta), (7) disminuir en un 10% el precio de los vehículos que tienen un valor mayor a una cantidad dada, (8) localizar el vehículo más antiguo, (9) localizar el vehículo más potente (el de más cilindrada), y (10) localizar el vehículo más barato (el de menor precio).

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

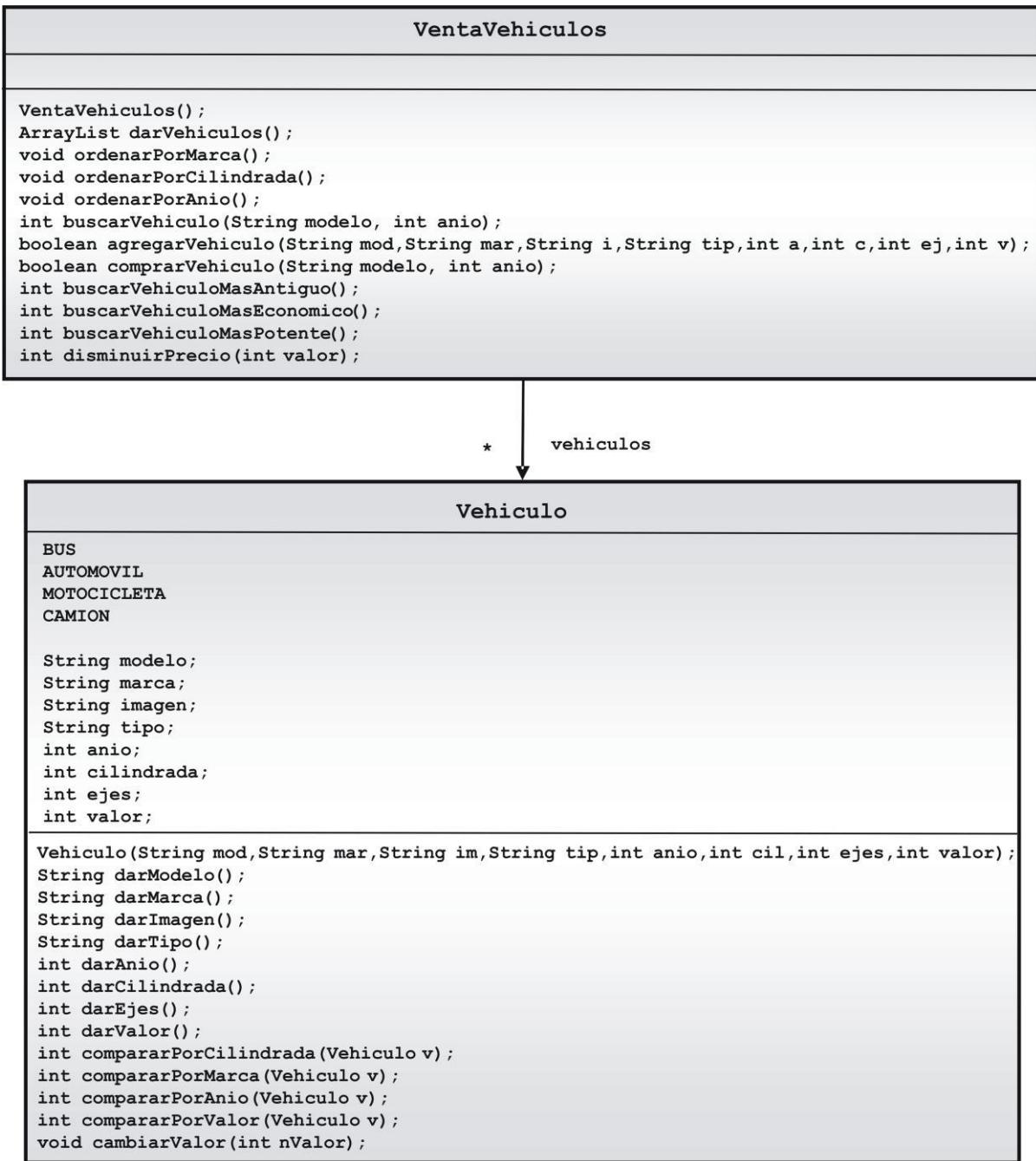
Venta de Vehículos

En la parte superior izquierda aparece la lista de cada uno de los vehículos disponibles. Con los botones que aparecen un poco más abajo, el usuario puede ordenar esta lista por marca, por cilindrada o por antigüedad. También puede buscar un vehículo, dando su modelo y su año. En la parte derecha, aparecen los datos del vehículo que se encuentra seleccionado en la lista. Allí aparece toda la información que se tiene del vehículo. Con los botones del panel llamado “Consultas y Operaciones”, el usuario puede agregar un nuevo vehículo, localizar el vehículo más económico, el más antiguo o el más potente, al igual que disminuir el precio de todos los vehículos que superan un cierto valor. Allí mismo se encuentra un botón que permite a un usuario comprar el vehículo cuya información se está mostrando, haciendo que éste salga de la lista de vehículos disponibles.

Requerimientos funcionales: Especifique los diez requerimientos funcionales descritos en el enunciado	
Nombre:	R1 – Obtener la lista de todos los vehículos que están a la venta.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R2 – Obtener la información detallada de un vehículo dado.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R3 – Agregar un nuevo vehículo a la venta.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R4 – Ordenar la lista de vehículos por modelo, por marca o por año.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R5 – Hacer una búsqueda usando el modelo y el año del vehículo.
Resumen:	
Entradas:	
Resultados:	

Nombre:	R6 – Comprar un vehículo dado.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R7 – Disminuir en un 10% el precio de los vehículos que tienen un valor mayor a una cantidad dada.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R8 – Localizar el vehículo más antiguo.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R9 – Localizar el vehículo más potente.
Resumen:	
Entradas:	
Resultados:	
Nombre:	R10 – Localizar el vehículo más barato.
Resumen:	
Entradas:	
Resultados:	

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



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

Clase	Atributo	A/D	Restricción o relación

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

<p>Este método verifica el invariante de la clase.</p>	<pre>public class Vehiculo { private void verificarInvarianto() { } }</pre>
<p>Retorna 0 si los dos vehículos tienen la misma cilindrada, 1 si el vehículo tiene mayor cilindrada que el que viene por parámetro y -1 si el vehículo tiene menor cilindrada que el que viene por parámetro.</p>	<pre>public class Vehiculo { public int compararPorCilindrada(Vehiculo v) { } }</pre>
<p>Retorna 0 si los dos vehículos tienen el mismo modelo, 1 si el vehículo tiene un modelo mayor que el que viene por parámetro y -1 si el vehículo tiene un modelo menor que el que viene por parámetro.</p>	<pre>public class Vehiculo { public int compararPorModelo(Vehiculo v) { } }</pre>

	<pre> public class VentaVehiculos { private void verificarInvariantes() { private boolean buscarVehiculosModeloYAnioRepetido() { return false; } } } </pre> <p>Este método verifica el invariante de la clase.</p> <p>Se apoya en un segundo método que retorna verdadero si hay dos vehículos con los mismos modelo y año, y falso en caso contrario.</p>
	<pre> public class VentaVehiculos { public int buscarVehiculo(String modelo, int anio) { return -1; } } </pre> <p>Este método busca un vehículo según su modelo y año, y retorna la posición del vector en la que se encuentra. Si no encuentra ningún vehículo con ese modelo y año retorna -1.</p>
	<pre> public class VentaVehiculos { public boolean agregarVehiculo(String modelo, String marca, String imagen, String tipo, int anio, int cilindrada, int ejes, int valor) { return false; } } </pre> <p>Este método agrega un nuevo vehículo a la venta. Retorna falso en caso de que ya exista un vehículo con el mismo modelo y año. Utiliza el método desarrollado en el punto anterior.</p>

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

    }
}
```

Este método ordena los vehículos de manera ascendente teniendo en cuenta la cilindrada.

Utiliza para hacerlo el algoritmo de inserción.

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

    }
}
```

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

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

Escenario:

Clase Vehiculo:

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

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