

**Grado en Ingeniería de Computadores – Curso 2021-22**

**Ingeniería del Software: Laboratorio**

**PECL2 Pruebas y métricas de mantenimiento**

**Enunciado:**

Se trata de un trabajo individual, no se puede hacer en grupo. Consiste en realizar tres ejercicios independientes.

Contenido

[Ejercicio 1. Pruebas unitarias y de integración 2](#_Toc103080662)

[PARTE A. PRUEBAS UNITARIAS 2](#_Toc103080663)

[Clase SecuenciaADN: 3](#_Toc103080664)

[PARTE B. PRUEBAS DE INTEGRACIÓN 9](#_Toc103080665)

[Ejercicio 2. Pruebas unitarias con objetos simulados (mocks) 14](#_Toc103080666)

[Caso 1 del método solicitarPrestamoPersonal: 18](#_Toc103080667)

[Caso 2 del método solicitarPrestamoPersonal: 19](#_Toc103080668)

[Caso 3 del método solicitarPrestamoPersonal: 20](#_Toc103080669)

[Caso 4 del método solicitarPrestamoPersonal: 21](#_Toc103080670)

[Caso 5 del método solicitarPrestamoPersonal: 22](#_Toc103080671)

[Caso 6 del método solicitarPrestamoPersonal: 23](#_Toc103080672)

[Caso 7 del método solicitarPrestamoPersonal: 24](#_Toc103080673)

[Caso 8 del método solicitarPrestamoPersonal: 25](#_Toc103080674)

[Ejercicio 3. Métricas de mantenimiento del software: 26](#_Toc103080675)

[Forma de entrega: 26](#_Toc103080676)

# Ejercicio 1. Pruebas unitarias y de integración

## PARTE A. PRUEBAS UNITARIAS

1. Calcular la **complejidad ciclomática** de los métodos de la clase *SecuenciaADN* (ver código fuente en Anexo), dibujando previamente un grafo para cada método.

### Clase SecuenciaADN:

* Método esValida:

Diagrama

Descripción generada automáticamente

Cálculo: 9 aristas - 7 nodos + 2 = 4.

Hay que preparar 4 casos de prueba:

* + - Uno para cuando el tamaño de la variable secuencia sea 0, retornando False.
      * testEsValida\_Sec\_0
    - Uno para cuando el tamaño de la secuencia sea >= 1 y tenga caracteres válidos, retornando True.
      * testEsValida\_Validos
    - Uno para cuando el tamaño de la secuencia sea >= 1 y tenga algún carácter inválido, retornando False.
      * testEsValida\_Invalidos
    - Uno para cuando el tamaño de la secuencia sea >= 1 y no entre en el bucle for, retornando True. ESTE CAMINO ES IMPOSIBLE EN LA PRÁCTICA, PERO HAY QUE CONTARLO.
      * testEsValida\_No\_For
* Método contar:

Diagrama

Descripción generada automáticamente

Cálculo: 6 aristas - 5 nodos + 2 = 3.

Hay que preparar 3 casos de prueba:

* Uno para cuando no se cumple la condición for, y se retornaría con aux = 0.
  + testContar\_No\_For
* Uno para cuando se cumple el for, pero no se cumplen los if, retornando aux = 0.
  + testContar\_No\_Encontrado
* Uno para cuando se cumple el for y algún if, retornando [ aux ! = 0].
  + testContar\_Encontrado

1. Diseñar casos de prueba para cada uno de los métodos de la clase *SecuenciaADN* aplicando la técnica de **caminos básicos**, con tantos casos de prueba como indique la complejidad ciclomática.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **No** | **Nombre/identificador** | **Descripción** | **Pre-condiciones** | **Entradas** | **Pasos** | **Resultados Esperados** | **Resultados de error** |
| 1 | testEsValida\_Sec\_0 | Se prueba que el resultado sea False al tener tamaño 0 secuencia. | Instanciar Secuencia con String() [Tamaño 0] | - | Llamar SecuenciaADN.esValida() | False | True |
| 2 | testEsValida\_Validos | Se prueba que el resultado sea True al meter una secuencia válida. | Instanciar Secuencia con String("") [ENTRADA] | ACGTTTTTTTTAGCCCCCA | Llamar SecuenciaADN.esValida() | True | False |
| 3 | testEsValida\_Invalidos | Se prueba que el resultado sea False al meter un carácter inválido. | Instanciar Secuencia con String("") [ENTRADA] | ACGTTTTTTTZTAGCCCCA | Llamar SecuenciaADN.esValida() | False | True |
| 4 | testEsValida\_No\_For | Se prueba que siempre se entra en el for. | Instanciar Secuencia con String("") [ENTRADA] | A | Llamar SecuenciaADN.esValida() | True | False |
| 5 | testContar\_No\_For | Se prueba la secuencia con tamaño 0. | Instanciar Secuencia con String() [Tamaño 0], nucleotido = 'A' | - | Llamar SecuenciaADN.esValida() | 0 | !=0 |
| 6 | testContar\_No\_Encontrado | Se prueba el conteo con un carácter no presente en la secuencia. | Instanciar Secuencia con String() [ENTRADA], nucleotido = 'A' | CGTTTTTTTZTGCCCC | Llamar SecuenciaADN.esValida() | 0 | !=0 |
| 7 | testContar\_Encontrado | Se prueba el conteo con un carácter presente en la secuencia. | Instanciar Secuencia con String() [ENTRADA], nucleotido = 'A' | CGTTTTATTTZTGCCACCA | Llamar SecuenciaADN.esValida() | 3 | !=3 |

Los casos de prueba:

* 1,2,3,4 – Método esValida
* 5,6,7 – Método contar

1. Crear un proyecto nuevo y copiar el código de la clase *SecuenciaADN*. Programar con JUnit 5 las **pruebas unitarias** necesarias para cubrir todos los caminos básicos, y ejecutar todos los casos de prueba.

Proyecto adjuntado en la entrega nombrado como cadenaADN.zip

* Método testEsValida\_Sec\_0

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

* Método testEsValida\_Validos

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

* Método testEsValida\_Invalidos

Texto

Descripción generada automáticamente

* Método testEsValida\_No\_For

Texto

Descripción generada automáticamente

* Método testContar\_No\_For

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

* Método testContar\_No\_Encontrado

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

* Método testContar\_Encontrado

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Ejecución de las pruebas:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

## PARTE B. PRUEBAS DE INTEGRACIÓN

1. Calcular la **complejidad ciclomática** del método *analizarADN* de la clase Laboratorio (ver código fuente en Anexo), dibujando previamente un grafo.

* Método analizarADN:

Diagrama

Descripción generada automáticamente

Cálculo: 6 aristas - 6 nodos + 2 = 2.

Hay que preparar 2 casos de prueba:

* Uno para cuando se cumple la condición if, y se retornaría True.
  + testAnalizarADN\_valido
* Uno para cuando no se cumple el if, retornando False.
  + testAnalizarADN\_no\_valido

1. Diseñar casos de prueba para el método *analizarADN* aplicando la técnica de **caminos básicos**, con tantos casos de prueba como indique la complejidad ciclomática.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **No** | **Nombre/identificador** | **Descripción** | **Pre-condiciones** | **Entradas** | **Pasos** | **Resultados Esperados** | **Resultados de error** |
| 1 | testAnalizarADN\_valido | Se prueba que el resultado sea True al meter una secuencia válida. | Instanciar Secuencia con adn | ACGTTTTTTTTAGCCCCCA | Llamar SecuenciaADN.esValida() | True | False |
| 2 | testAnalizarADN\_no\_valido | Se prueba que el resultado sea False al meter una secuencia inválida. | Instanciar Secuencia con adn | ACGTTTTTTTTAZCCCCCA | Llamar SecuenciaADN.esValida() | False | True |

Los casos de prueba:

* 1,2 – Método analizarADN

1. Crear un proyecto nuevo y copiar el código de la clase *Laboratorio*. Programar con JUnit 5 las **pruebas de integración** de la clase *Laboratorio* con la clase *SecuenciaADN*, necesarias para cubrir todos los caminos básicos, y ejecutar todos los casos de prueba.

Proyecto adjuntado en la entrega nombrado como laboratorio.zip

* Método testAnalizarADN\_valido:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

* Método testAnalizarADN\_no\_valido:

Interfaz de usuario gráfica, Texto

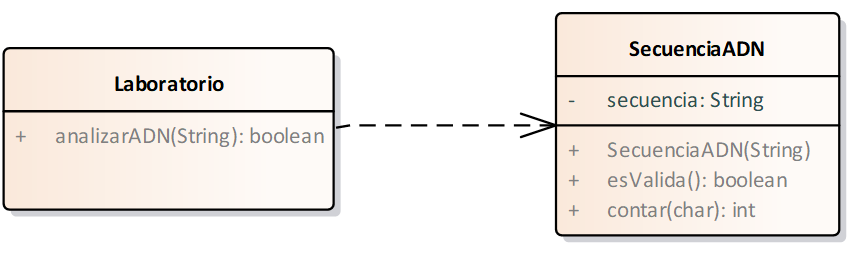
Descripción generada automáticamente

Ejecución de las pruebas:

Interfaz de usuario gráfica, Texto, Aplicación

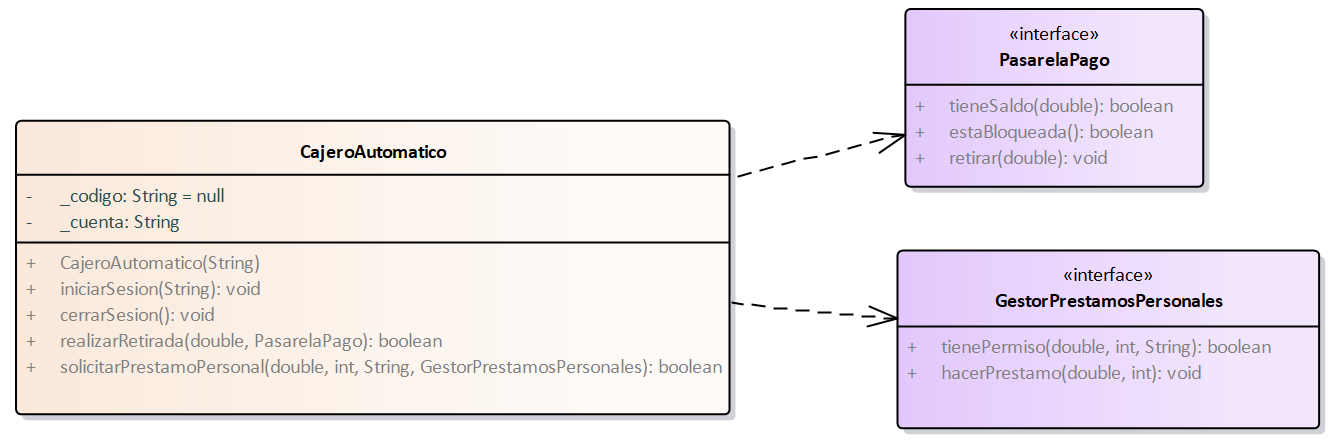
Descripción generada automáticamente

En el siguiente diagrama se muestran las clases del ejercicio.



# Ejercicio 2. Pruebas unitarias con objetos simulados (mocks)

Se debe partir del código finalizado de la práctica P6 de la asignatura, y añadir una nueva clase denominada *GestorPrestamosPersonales*, que será utilizada por *CajeroAutomático*. Esta nueva clase aún no está desarrollada, por lo que se especificará su comportamiento mediante una interfaz.



El gestor de préstamos permitirá desde el cajero solicitar un préstamo personal (especificando la cantidad, el plazo de amortización y la fecha de solicitud) con las siguientes reglas:

1. No se puede pedir préstamos de más de 3000 EUR o menos de 1000 EUR.
2. No se pueden pedir préstamos a más de 12 meses ni a menos de 6 meses.
3. No se puede conceder más de un préstamo dentro del mismo año (por eso se proporciona la fecha de solicitud al gestor de préstamos para que lo compruebe).

Se pide lo siguiente:

1. Codificar la interfaz *GestorPrestamosPersonales* con dos métodos:
   * *tienePermiso*, que recibe como parámetros la cantidad pedida, el plazo de amortización en meses y la fecha de solicitud (por simplicidad se puede utilizar un String como fecha). El tipo de resultado es boolean para indicar si se tiene o no permiso para obtener el préstamo.
   * *hacerPrestamo*, que recibe como parámetros la cantidad pedida y el plazo, y realiza el préstamo solicitado.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

1. Codificar el método *solicitarPrestamoPersonal* en la clase *CajeroAutomático*, que utilice los métodos *tienePermiso* y *hacerPrestamo* de la interfaz *GestorPrestamosPersonales*. Este método tiene como parámetros:
   * La cantidad solicitada de préstamo (double)
   * El plazo de amortización (int)
   * La fecha de solicitud (String)
   * El gestor de préstamos personales (de la clase GestorPrestamosPersonales) que se encargará de gestionar esa solicitud de préstamo

Texto

Descripción generada automáticamente con confianza media

Se ha usado el método Objects.isNull(Object) para comprobar que el gestor no sea null.

1. Diseñar los casos de prueba para el método *solicitarPrestamoPersonal* del cajero, considerando la técnica de clases de equivalencia. Realizar una tabla con los casos válidos y no válidos.

Método solicitarPrestamoPersonal:

|  |  |  |
| --- | --- | --- |
| **Parámetros** | **Clases válidas** | **Clases no válidas** |
| cantidadPrestamo | (1) 1000 <= cantidadPrestamo <= 3000 | (2) cantidadPrestamo < 1000  (3) cantidadPrestamo > 3000 |
| plazoPagoMeses | (4) 6 <= plazoPagoMeses <= 12 | (5) plazoPagoMeses < 6  (6) plazoPagoMeses > 12 |
| fecha | (7) "fecha" válida y prestamo no pedido | (8) "fecha" no válida  (9) "fecha" prestamo pedido anteriormente |
| gestor | (10) Objeto "gestor" instanciado de la clase GestorPrestamosPersonales | (11) Objeto "gestor" con valor null |

**Casos válidos:**

* (1) (4) (7) (10): cantidadPrestamo = 1500, plazoPagoMeses = 11, fecha = “15/07/2021”, gestor != null

**Casos no válidos:**

* (2) (4) (7) (10): cantidadPrestamo = 900, plazoPagoMeses = 11, fecha = “15/07/2021”, gestor != null
* (3) (4) (7) (10): cantidadPrestamo = 3300, plazoPagoMeses = 11, fecha = “15/07/2021”, gestor != null
* (1) (5) (7) (10): cantidadPrestamo = 1500, plazoPagoMeses = 4, fecha = “15/07/2021”, gestor != null
* (1) (6) (7) (10): cantidadPrestamo = 1500, plazoPagoMeses = 13, fecha = “15/07/2021”, gestor != null
* (1) (4) (8) (10): cantidadPrestamo = 1500, plazoPagoMeses = 11, fecha = “hola”, gestor != null
* (1) (4) (9) (10): cantidadPrestamo = 1500, plazoPagoMeses = 11, fecha = “18/09/2021”, gestor != null
* (1) (4) (7) (11): cantidadPrestamo = 1500, plazoPagoMeses = 11, fecha = “15/07/2021”, gestor == null

Se han optado por estas clases de equivalencia, debido a que hay que tener en cuenta que los parámetros cuenten con las siguientes condiciones impuestas:

- cantidadPrestamo debe tener un valor double o fraccionario entre 1000 y 3000.

- plazoPagoMeses debe tener un valor int o entero entre 6 y 12.

- fecha debe tener un valor String con un formato de fecha válido dd/mm/yyyy y que en ese mismo año no se haya solicitado un prestamo por el mismo usuario.

- gestor debe estar instanciado como objeto de la clase GestorPrestamosPersonales y por lo tanto no puede ser null.

1. Implementar, utilizando JUnit e EasyMock, los casos de pruebas establecidos en el apartado anterior, utilizando objetos simulados (mock) para *GestorPrestamosPersonales*.

### Caso 1 del método solicitarPrestamoPersonal:

(1) (4) (7) (10): cantidadPrestamo = 1500, plazoPagoMeses = 11, fecha = “15/07/2021”, gestor != null

**Texto

Descripción generada automáticamente con confianza media**

### Caso 2 del método solicitarPrestamoPersonal:

(2) (4) (7) (10): cantidadPrestamo = 900, plazoPagoMeses = 11, fecha = “15/07/2021”, gestor != null

Texto

Descripción generada automáticamente con confianza media**Nota: No se debe programar ningún expect en el gestorMock, debido a que la comprobación de la cantidadPrestamo se realiza en el método CajeroAutomatico.solicitarPrestamoPersonal(…)**

### Caso 3 del método solicitarPrestamoPersonal:

(3) (4) (7) (10): cantidadPrestamo = 3300, plazoPagoMeses = 11, fecha = “15/07/2021”, gestor != null

**Imagen que contiene Tabla

Descripción generada automáticamente**

**Nota: No se debe programar ningún expect en el gestorMock, debido a que la comprobación de la cantidadPrestamo se realiza en el método CajeroAutomatico.solicitarPrestamoPersonal(…)**

### Caso 4 del método solicitarPrestamoPersonal:

1. (5) (7) (10): cantidadPrestamo = 1500, plazoPagoMeses = 4, fecha = “15/07/2021”, gestor != null

**Nota: No se debe programar ningún expect en el gestorMock, debido a que la comprobación de la cantidadPrestamo se realiza en el método CajeroAutomatico.solicitarPrestamoPersonal(…)**

**Imagen que contiene Texto

Descripción generada automáticamente**

### Caso 5 del método solicitarPrestamoPersonal:

1. (6) (7) (10): cantidadPrestamo = 1500, plazoPagoMeses = 13, fecha = “15/07/2021”, gestor != null

**Nota: No se debe programar ningún expect en el gestorMock, debido a que la comprobación de la cantidadPrestamo se realiza en el método CajeroAutomatico.solicitarPrestamoPersonal(…)**

**Tabla

Descripción generada automáticamente**

### Caso 6 del método solicitarPrestamoPersonal:

(1) (4) (8) (10): cantidadPrestamo = 1500, plazoPagoMeses = 11, fecha = “hola”, gestor != null

**Escala de tiempo

Descripción generada automáticamente**

### Caso 7 del método solicitarPrestamoPersonal:

(1) (4) (9) (10): cantidadPrestamo = 1500, plazoPagoMeses = 11, fecha = “18/09/2021”, gestor != null

**NOTA: Se supone que el usuario ha pedido un prestamo en ese mismo año antes de ejecutar este caso de prueba, por eso debe salir false.**

**Texto, Aplicación

Descripción generada automáticamente**

### Caso 8 del método solicitarPrestamoPersonal:

1. (4) (7) (11): cantidadPrestamo = 1500, plazoPagoMeses = 11, fecha = “15/07/2021”, gestor == null

**NOTA: Se supone que se ha pasado la variable “gestorMock” con valor Null, por eso debería retornar un false.**

**Imagen que contiene Escala de tiempo

Descripción generada automáticamente**

Ejecución de las pruebas:

Tabla

Descripción generada automáticamente

# Ejercicio 3. Métricas de mantenimiento del software:

1. Seleccionar un programa en Java, que tenga más de 10 clases, de un repositorio de código abierto (SourceForge.net, GoogleCode, etc.), e identificar las clases que pueden ser más problemáticas en mantenimiento utilizando como herramientas de análisis **JavaNCSS**, **CKJM** y **Dependency Finder**.
2. **Razonar las decisiones** basándose en el tipo de métricas utilizadas y su significado. Citar referencias de posibles autores que proponen rangos de valores recomendados para cada métrica. Se recomienda consultar el documento “[Medición en la Orientación a Objetos](http://www.cc.uah.es/drg/b/RodHarRama00.pdf)” como punto de partida.

Nota: No se puede usar un proyecto ya usado por otros compañeros ni el utilizado en la práctica de la asignatura (JChecs). Para ello, se debe anotar el nombre del proyecto en el Foro “PECL2 – Lista de proyectos de código abierto”, creado en Blackboard, siguiendo las instrucciones que allí se recogen.

# Forma de entrega:

El trabajo debe enviarse a través del Campus virtual, desde la sección Trabajos > PECL2.

Debe enviarse un único archivo comprimido (zip o rar) con el siguiente contenido:

* *PECL2(ApellidoNombredelAlumno).docx*. Con el informe del trabajo, incluyendo copias de pantalla, tablas, fragmentos de código y tantos elementos explicativos como considere necesario. Debe tener índice y tres secciones:
  + Ejercicio 1: Con los 6 apartados indicados en el enunciado del ejercicio. Es necesario razonar los casos de prueba diseñados, e incluir las evidencias necesarias (capturas de pantallas del código Java de las pruebas y de su ejecución mostrando que se han superado todas), para que el profesor pueda comprobar que es correcto sin tener que abrir los proyectos en Netbeans.
  + Ejercicio 2: Con los 4 apartados indicados en el enunciado del ejercicio. Incluyendo las clases, métodos y casos de prueba implementados. Deben razonarse los casos de prueba diseñados así como las clases de equivalencia utilizadas. Deben incluirse en el documento todas las evidencias necesarias (capturas de pantallas del código Java de las pruebas y de su ejecución), para que el profesor pueda comprobar que es correcto sin tener que abrir los proyectos en Netbeans.
  + Ejercicio 3: Con los 2 apartados indicados en el enunciado del ejercicio. Indicando la URL desde la que se ha descargado el programa analizado. Incluyendo capturas de pantalla de las tres herramientas utilizadas, y razonando las conclusiones más relevantes sobre los valores obtenidos y su repercusión en el mantenimiento de la aplicación. Hay que indicar, al menos, qué clase(s) tiene(n) el valor más alto en métricas relevantes, y aquellas que están fuera de rangos recomendados por expertos, y citar la fuente de los expertos.
* *Proyectos de NetBeans*. Tres archivos comprimidos (zip o rar), uno con cada proyecto Java realizado en NetBeans: cadenaADN.zip, laboratorio.zip, cajero.zip.
* *Archivo Excel con métricas*. Que incluya los valores de las métricas del ejercicio 3, con un formato similar al ejemplo de la práctica 7.
* *Archivo comprimido con el código fuente del programa analizado en el ejercicio 3.*

**Criterios de valoración**

Se valorará la legibilidad, claridad, concisión y pertinencia respecto a lo solicitado así como la precisión de los documentos entregados y de las explicaciones correspondientes. El peso de cada ejercicio en la calificación es el siguiente: Ejercicio 1: 40%, Ejercicio 2: 30%, Ejercicio 3: 30%.

**Anexo. Código fuente para el Ejercicio 1**

|  |
| --- |
| package secuenciaADN;  /\*\*  \* Representa una secuencia de ADN o secuencia de nucleótidos,  \* que consiste en una sucesión de letras A, C, G y T, que simbolizan las cuatro  \* subunidades de nucleótidos: Adenina, Citosina, Guanina y Timina.  \* Las secuencias se presentan pegadas unas a las otras, sin espacios.  \* Por ejemplo, la secuencia AAAGTCTGAC.  \*\*/  public class SecuenciaADN {  private String secuencia;  /\*  \* Crea la secuencia convirtiendo a mayúsculas todas las letras de la secuencia  \* que se recibe como parámetro, que no debe ser nulo.  \*/  public SecuenciaADN(String sec) {  assert (sec != null);  secuencia = sec.toUpperCase();  }  /\*  \* Comprueba si una secuencia el válida, es decir, si sólo contiene  \* los caracteres A, C, G y T.  \*/  public boolean esValida() {  if (secuencia.lenght()==0) {  return false  }  else {  for (int i = 0; i < secuencia.length(); i++) {  if ("ACGT".indexOf(secuencia.charAt(i)) == -1) {  return false;  }  }  return true;  }  }  /\*  \* Cuenta las veces que aparece un nucleótido válido en la secuencia.  \* Los nucleótidos válidos son: A, C. G y T.  \*/  public int contar(char nucleotido) {  int aux = 0;  for (int i = 0; i < secuencia.length(); i++) {  if (secuencia.charAt(i) == nucleotido) {  aux++;  }  }  return aux;  }  } |

|  |
| --- |
| package laboratorio;  import secuenciaADN.SecuenciaADN;  public class Laboratorio {  public boolean analizarADN(String adn) {  SecuenciaADN sec = new SecuenciaADN(adn);    if (sec.esValida()) {  return true;  } else {  return false;  }  }  } |