

Módulo Profesional 05: Entornos de desarrollo

UF2 Actividad B

CICLO FORMATIVO DE GRADO SUPERIOR EN
DESARROLLO DE APLICACIONES MULTIPLATAFORMA
MODALIDAD ONLINE

ALUMNA:

MARÍA LAURA RONDINA IOBBI

Descripción de la actividad:

En la siguiente actividad se valorarán los conceptos más importantes de la actividad, Por lo que deberás de dar respuesta a los siguientes ejercicios teórico/práctico en el mismo documento para entregar por el campus. Poner nombre al documento y entregar en formato Word/pdf.

1. Define el concepto de refactorización.[0,5 puntos]

La **refactorización** es una técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo (lo que se conoce informalmente por “*limpiar el código*”). Se realiza a menudo como parte del proceso de desarrollo del software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Los tests aseguran que la refactorización no cambia el comportamiento del código, es decir, que el programa continúa haciendo exactamente lo mismo que antes de la misma.

Es entonces una parte del mantenimiento del código que no arregla errores ni añade funcionalidad. Su objetivo es mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro. Añadir nuevo comportamiento a un programa puede ser difícil con la estructura dada del programa, así que un desarrollador puede refactorizarlo primero para facilitar esta tarea y luego añadir el nuevo comportamiento.

La refactorización debe ser realizada como un paso separado, para poder comprobar con mayor facilidad que no se han introducido errores al llevarla a cabo. Al final de la refactorización, cualquier cambio en el comportamiento es un *bug* y puede ser arreglado de manera separada a la depuración de la nueva funcionalidad.

Un ejemplo de una refactorización trivial es cambiar el nombre de una variable para que sea más significativo, de una sola letra 't' a 'tiempo'. Una refactorización más compleja es transformar el trozo dentro de un bloque en una subrutina. Una refactorización todavía más compleja es remplazar una sentencia condicional if por polimorfismo.

2. ¿La refactorización incluye cambios en el comportamiento del software? ¿Si/No y por qué? [0,5 puntos]

La refactorización NO supone cambios en el comportamiento del software, ya que se trata únicamente de una técnica de limpieza de código que no altera el código fuente de cara a los resultados de ejecución. Por lo tanto, la refactorización no arregla errores ni incorpora funcionalidades al programa. Altera únicamente la estructura interna del código, pero sin cambiar su comportamiento externo.

Si durante una refactorización se ha cambiado el comportamiento del software o web, es que se ha generado un error o bug, en discordancia con el propósito de la refactorización, que es “limpiar” sin alterar el resultado. De hecho, los objetivos de la refactorización son:

- Mejorar la facilidad de comprensión del código.
- Cambiar su estructura y diseño (para facilitar así nuevos desarrollos, la pronta localización y resolución de errores, la adición de alguna funcionalidad al software...)
- Eliminar código muerto o inutilizado.
- Facilitar el mantenimiento del programa en el futuro.
- Mejorar la Calidad y Eficiencia, evitando duplicación de código, simplificando el diseño, y minimizando el número de clases y de métodos.

3. ¿Cuáles son las limitaciones que existen para aplicar la refactorización y cuáles son los patrones de refactorización más usados? [0,5 puntos]

La refactorización se manifiesta en el marco de la realidad de cada empresa, es decir, con una serie de limitaciones, como **plazos, presupuestos y recursos finitos**, por lo que probablemente nunca podrán refactorizar tanto como fuera deseable. El equilibrio entre lo necesario, por un lado, y el tiempo, los costes y los recursos por el otro, hará que la refactorización sea la mejor opción en ese momento concreto. Algunas de las **limitaciones** que sufre la refactorización son las siguientes:

- Los proyectos suelen estar condicionados por el aspecto económico.
- La refactorización no puede facturarse a un cliente inmediatamente.
- El programador que “limpia” el código puede no ser el mismo que el creador, lo que aumenta el riesgo de introducir nuevos errores.
- Las modificaciones del código han de poder probarse sin esfuerzo, por lo que hay que tener preparadas unas pruebas funcionales o unitarias.
- Si se trata de un código antiguo en el que han trabajado muchos programadores, algunos de ellos noveles, podemos encontrarnos con un auténtico lío cuya mejora puede costar cara.
- A veces no podemos cambiar la interfaz de usuario y se complica la refactorización.

Los patrones de refactorización más usados pueden clasificarse en **tres grandes grupos**:

- **Patrones de creación.** Ayudan a la creación y configuración de objetos. Se subdividen en:

- **Simple Factory (Fábrica simple).** Clase utilizada para la creación de instancias de objetos.
- **Factory Method (Método de Fábrica).** Define una interfaz para crear objetos y permite a las subclases decidir la clase a instanciar.
- **Abstract Factory (Fábrica Abstracta).** Interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases.
- **Builder (Constructor).** La construcción de un objeto se separa de su representación, por lo que pueden crearse diferentes representaciones con el mismo proceso de construcción.

- **Patrones de estructura.** Asisten en la generación de grupos de objetos en estructuras mayores y se adecúan a la forma de combinar clases y objetos en el montaje de dichas estructuras. Se subdividen en:

- **Adapter (Adaptador).** Hace de intermediario entre dos clases que posean interfaces incompatibles y, por lo tanto, de difícil comunicación.
- **Bridge (Puente).** Descompone un componente en dos jerarquías de clase: una abstracción funcional (algo que hace) y una implementación (la cosa en sí).
- **Decorator (Decorador).** Agrega o limita competencias a un objeto

- **Patrones de comportamiento.** Conforman el flujo y tipo de comunicación entre los objetos de un programa. Se subdividen en:

- **Chain of Responsibility (Cadena de responsabilidad).** La cadena de mensajes puede ser respondida por más de un receptor.
- **Command (Orden).** Representa una petición con un objeto.
- **Mediator (Mediador).** Se introduce un nuevo objeto que administra los mensajes entre objetos.

4. Indicar la diferencia entre un analizador de código estático y uno dinámico. [0,5 puntos]

Un **análisis de código** consiste en comprobar el código fuente con la intención de realizar posteriormente trabajos de mantenimiento (refactorización). Algunas de las habilidades que los analizadores de código externos tienen están implementadas en los propios depuradores de los entornos de desarrollo. Se justifica el analizador de código externo para aspectos peculiares de depuración.

Podemos clasificar los análisis de código en **dos tipos**:

- **Análisis dinámico.** Se ejecuta el programa para detectar defectos de ejecución:
 - **Valgrind Tool Suite (C / C++)**. Por cada instrucción que ejecuta el programa, Valgrind añade una serie de instrucciones adicionales para analizar el comportamiento del programa.
 - **IBM Rational AppScan**. Chequea las vulnerabilidades de seguridad.
- **Análisis estático.** Se examina el código sin ejecutar el programa:
 - **FxCop .NET Framework 2.0**. Ofrece información acerca del diseño, localización, rendimiento y mejoras de la seguridad.
 - **PMD Rule Designer**. Chequea código fuente Java encontrando errores, variables no usadas y código duplicado.

Como el **análisis estático** no necesita de ejecución, permite detectar errores en una fase muy temprana de la escritura. Así se ahorra mucho tiempo en fases posteriores del desarrollo. El problema más grave que ofrece en cambio, es que puede arrojar positivos que no lo son y cuya falsedad sólo se verá durante la ejecución del código. Por otro lado, el **análisis dinámico** es más lento y necesita un proceso completo de testeo. Sin embargo, permite ver muchos errores que quedan ocultos en un análisis estático.

5. ¿Para qué sirve el control de versiones? ¿Cuál es la estructura de las herramientas de control de versiones? [0,5 puntos]

Una correcta gestión del código fuente, de los documentos, de los gráficos y del resto de ficheros que conforman un proyecto informático necesita de un software de control de versiones que provea de una base de datos que incluya la trazabilidad de las revisiones hechas por todos los programadores. Un **sistema de control de versiones** es una herramienta que registra todos los cambios hechos en uno o más proyectos, guardando así versiones del producto en todas sus fases del desarrollo. Las versiones son como fotografías que registran su estado en ese momento del tiempo y se van guardando a medida que se hacen modificaciones al código fuente.

Para un desarrollador esta herramienta es muy valiosa porque permite viajar atrás en el tiempo (hacer rollback) si los cambios aplicados no resultaron de la manera que se esperaba, pudiendo restaurar en cualquier momento una versión previa. Es como un respaldo permanente. Hoy en día son usados no solo por desarrolladores independientes sino también por *startups* y grandes corporaciones.

La nomenclatura más utilizada para las diferentes versiones es la unión de números (y, en ocasiones, también letras), que indican el nivel que definen. Algunos de los sistemas de control de versiones más famosos son **Subversion** (también conocido como Svn), **Git** y **Mercurial**.

En cuanto a la **Estructura de las herramientas de control de versiones**, el sistema gestiona una base de datos en forma de árbol en el que se van apilando los diferentes cambios que se realizan en el software. Algunos elementos de esta configuración son:

- **Tronco.** Es el camino de cambio principal.
- **Cabeza.** Es la última versión del tronco.
- **Ramas.** Son los caminos secundarios.
- **Delta.** Son los cambios de una versión respecto a otra.

6. Define que es un repositorio. [0,5 puntos]

El **repositorio** es el lugar en donde se almacena una copia completa de todos los ficheros y directorios que están bajo la gestión del software de control de versiones. De este modo, se monitoriza y gestiona los cambios en un sistema de archivos. Al operar al nivel del sistema de archivos, el sistema de control de versiones (VCS) monitorizará las acciones de adición, eliminación y modificación aplicadas a archivos y directorios. En el alcance de los archivos individuales de códigos fuente, un VCS monitorizará las adiciones, eliminaciones y modificaciones de las líneas de texto que contiene ese archivo. Asimismo, ofrecerá herramientas de colaboración para compartir e integrar dichos cambios en otros usuarios del VCS.

Los proyectos diferentes suelen estar ubicados en distintos repositorios. En cualquier caso, éstos no deberán atenderse entre ellos. Un mismo repositorio controla la concurrencia de varios programadores e intenta acceder al mismo elemento y a la misma versión de código mediante avisos y posibilidad de lectura sin grabación de código.

Los repositorios favorecen la unidad en el sentido de que todos los documentos están almacenados en una misma base de datos, lo que hace más fácil su recuperación), la preservación a largo plazo de la información, y una mayor visibilidad y comunicación entre sus usuarios.

Caso práctico 1: Aplicar refactorización a un programa.[2,5 puntos]

Utilizando el IDE de Eclipse o NetBeans, proponer y realizar un programa en JAVA y posteriormente modificarlo aplicando la refactorización.

Indicar qué herramienta de refactorización has utilizado.

Para demostrar el antes y después de la refactorización, documentar cada paso, te puedes ayudar con capturas de pantalla o cualquier otro material en el que sea visible la refactorización.

El programa propuesto a continuación solicita de manera secuencial los datos de cinco personas (nombre, apellido e idioma), y los almacena en un fichero de texto. Finalmente lee el fichero anterior para mostrar por consola los datos del número de persona que el usuario indique.

PROGRAMA INICIAL:

CLASE PRINCIPAL:

```
package Refactorizacion;

public class Pers {

    private String nom;
    private String ap;
    private String idi;

    Pers(String nom, String ape, String idiom) {

        this.nom = nom;
        ap = ape;
        idi = idiom;

    }

    public String getNombre() {

        return nom;

    }

    public String getApellido() {

        return ap;

    }

    public String getIdioma() {
        return idi;
    }

    public String toString() {

        return nom + "," + ap + "," + idi + ".";

    }

}
```

CLASE QUE CONTIENE EL MAIN:

```
package Refactorizacion;

import java.io.BufferedReader;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

import activ1.Persona;
```

```

public class Pers2 {

    public static void main(String[] args) {

        Scanner entrada = new Scanner(System.in);

        Pers[] personas = new Pers[5];

        for (int i = 0; i < 5; i++) {

            System.out.println("Introduzca el nombre");

            String nombre = entrada.nextLine();

            System.out.println("Introduzca el apellido");

            String apellido = entrada.nextLine();

            System.out.println("Introduzca el idioma");

            String idioma = entrada.nextLine();

            Pers persona = new Pers(nombre, apellido, idioma);

            personas[i] = persona;

        }

        File archivoPersonas = new File("filePersonas.txt");

        try (PrintWriter pw = new PrintWriter(archivoPersonas);) {
            for (int i = 0; i < personas.length; i++) {
                pw.println(personas[i]);
            }
        } catch (FileNotFoundException e) {
            System.out.println("Fichero No Localizado");
        }

        try (FileReader fr = new FileReader(archivoPersonas);
            BufferedReader br = new BufferedReader(fr);) {
            String linea = br.readLine();
            int i = 0;

            while (linea != null) {

                String[] datosPersona = linea.split(",");
                String nombre = datosPersona[0];

                String apellido = datosPersona[1];

                String idioma = datosPersona[2];

                Pers persona = new Pers(nombre, apellido, idioma);
                personas[i] = persona;

                linea = br.readLine();
                i++;

            }
        }
    }
}

```

```

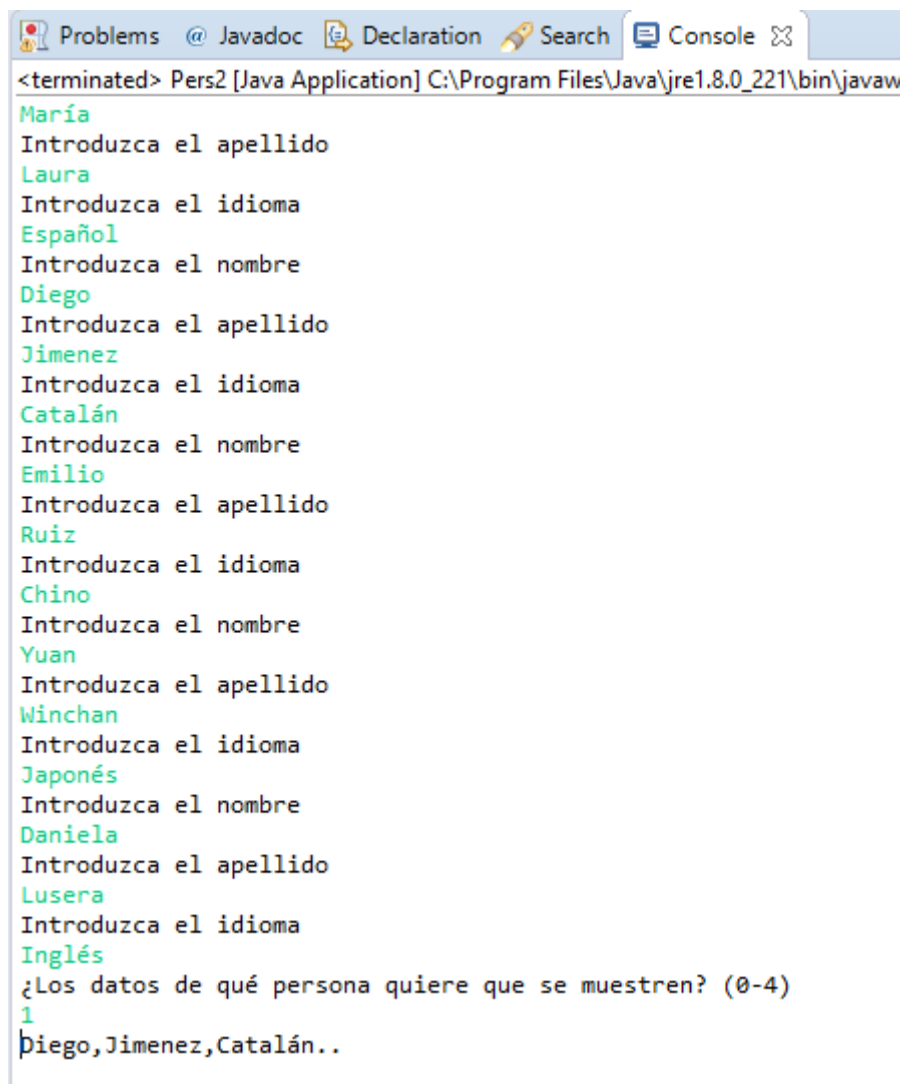
        System.out.println("¿Los datos de qué persona quiere que se
muestran? (0-4)");

        int indicePersona = entrada.nextInt();
        System.out.println(personas[indicePersona]);

    } catch (IOException e) {
        System.out.println("Fichero No Localizado");
    }
    entrada.close();
}
}

```

RESULTADO EN CONSOLA:

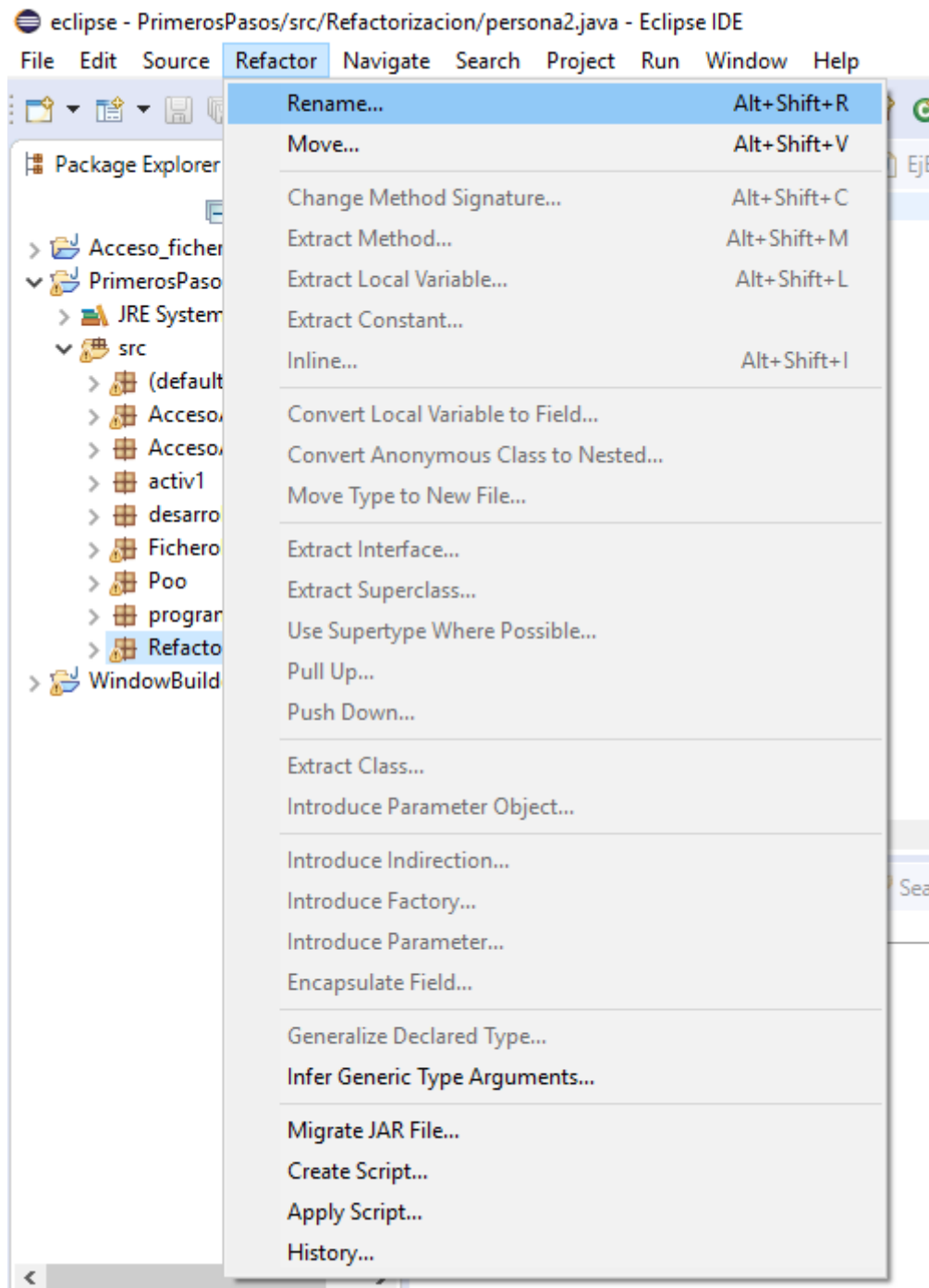


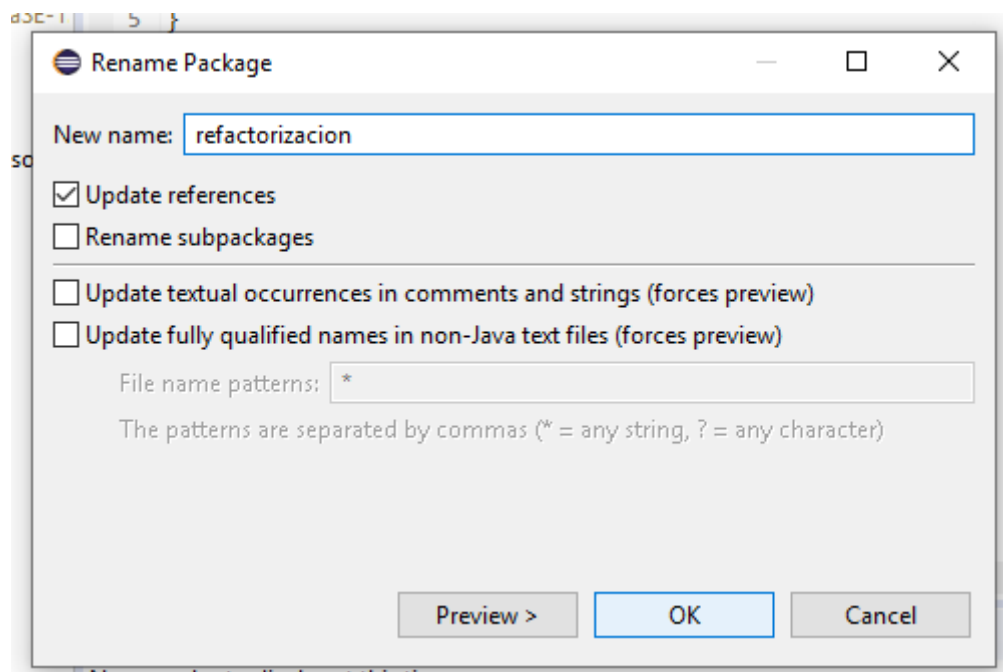
```

<terminated> Pers2 [Java Application] C:\Program Files\Java\jre1.8.0_221\bin\javaw
María
Introduzca el apellido
Laura
Introduzca el idioma
Español
Introduzca el nombre
Diego
Introduzca el apellido
Jimenez
Introduzca el idioma
Catalán
Introduzca el nombre
Emilio
Introduzca el apellido
Ruiz
Introduzca el idioma
Chino
Introduzca el nombre
Yuan
Introduzca el apellido
Winchan
Introduzca el idioma
Japonés
Introduzca el nombre
Daniela
Introduzca el apellido
Lusera
Introduzca el idioma
Inglés
¿Los datos de qué persona quiere que se muestren? (0-4)
1
Diego,Jimenez,Catalán..

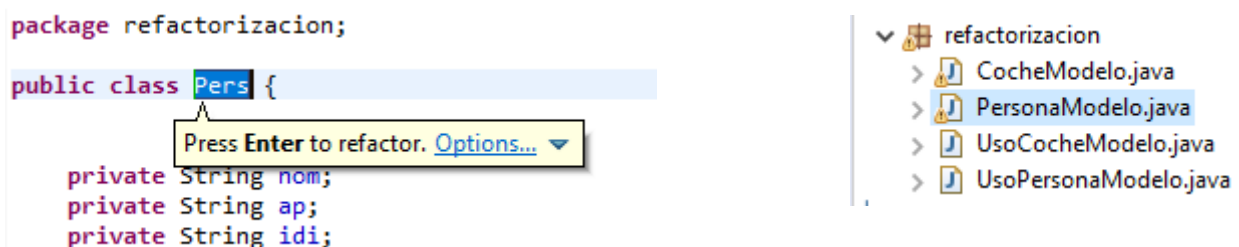
```


En el programa creado en Eclipse, nos damos cuenta que el nombre del paquete está escrito con mayúscula inicial y vamos a corregirlo, ya que por convención, los nombres de los paquetes generalmente comienzan con una letra minúscula. Para ello se usa la herramienta **Rename**: es la opción empleada para cambiar el identificador a cualquier elemento (nombre de variable, clase, método, paquete, directorio, etc). Cuando lo aplicamos, se cambian todas las veces que aparece dicho identificador.





Además comprobamos que los nombres dados a las dos clases dentro del paquete (Pers y Pers2) no representan correctamente la funcionalidad de las clases, por lo que los modificamos para que la estructura visible en la Ventana de Proyecto sea más intuitiva: usamos en su lugar PersonaModelo (la clase “planilla”) y UsoPersonaModelo (la clase contenedora del main).



Tampoco son representativos los nombres de los atributos, por lo que los renombramos usando la misma herramienta.

```
package refactorizacion;

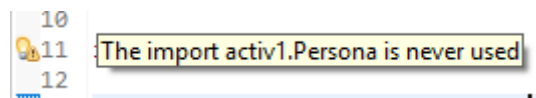
public class PersonaModelo {

    private String nombre;
    private String apellido;
    private String idioma;

    PersonaModelo(String nombre, String apellido, String idioma) {

        this.nombre = nombre;
        this.apellido = apellido;
        this.idioma = idioma;
    }
}
```

Se localiza la importación de una librería o función nunca utilizada, por lo que se procede a borrar esa línea de código.



También detectamos **código muerto** en lo relativo a la creación de métodos getters individuales en la clase principal para obtener los atributos, ya que el método toString creado en la misma clase los contiene a todos y es el que se invoca en la “clase main” durante la ejecución del programa. Por lo tanto se procede a eliminar el código inutilizado, es decir, a eliminar el siguiente fragmento de código:

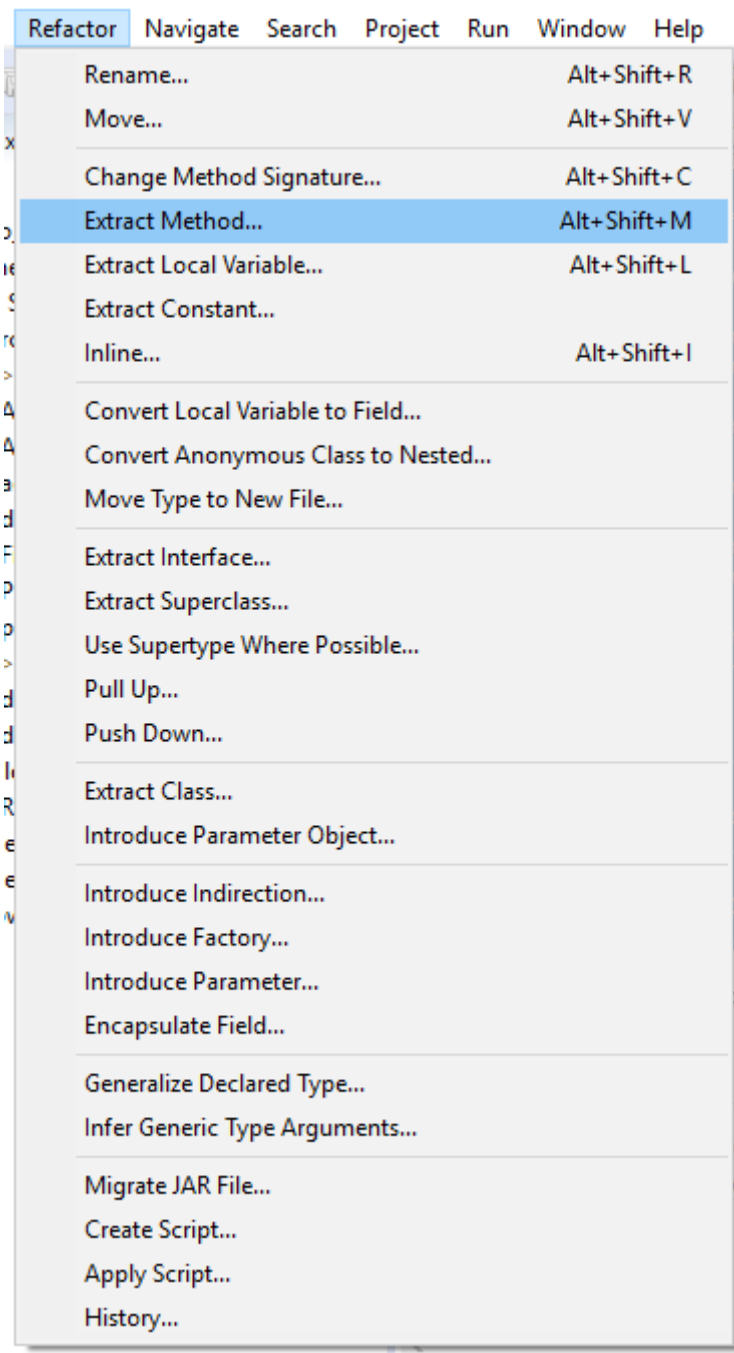
```
public String getNombre() {  
    return nombre;  
}  
  
public String getApellido() {  
    return apellido;  
}  
  
public String getIdioma() {  
    return idioma;  
}
```

Por último, se procede a extraer un método para cada uno de los bloques try-catch que contiene el programa, con ánimo de diferenciarlos (uno engloba la funcionalidad de escritura del fichero: “guardarPersonas”, y el otro la de lectura del fichero: “lecturaPersonas”).

La herramienta **Extract Method** convierte un bloque de código en un método, a partir de un bloque cerrado por llaves { }. Eclipse ajusta los parámetros y el retorno del método. Es muy útil cuando detectamos *bad smells* en métodos muy largos, o en bloques de código que se repiten.

Para extraer un método, se señala el bloque de código de interés y se sigue la ruta:

Refactor- → Extract Method



El código finalmente tiene el siguiente aspecto después de la refactorización, que no ha implicado cambios en el resultado del software:

CLASE PRINCIPAL:

```
package refactorizacion;

public class PersonaModelo {

    private String nombre;
    private String apellido;
    private String idioma;

    PersonaModelo(String nombre, String apellido, String idioma) {
```

```

        this.nombre = nombre;
        this.apellido = apellido;
        this.idioma = idioma;
    }

    public String toString() {

        return nombre + "," + apellido + "," + idioma + ".";
    }
}

```

CLASE QUE CONTIENE EL MAIN:

```

package refactorizacion;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class UsoPersonaModelo {

    public static void main(String[] args) {

        Scanner entrada = new Scanner(System.in);

        PersonaModelo[] personas = new PersonaModelo[5];

        for (int i = 0; i < 5; i++) {

            System.out.println("Introduzca el nombre");

            String nombre = entrada.nextLine();

            System.out.println("Introduzca el apellido");

            String apellido = entrada.nextLine();

            System.out.println("Introduzca el idioma");

            String idioma = entrada.nextLine();

            PersonaModelo persona = new PersonaModelo(nombre, apellido,
            idioma);
            personas[i] = persona;
        }
    }
}

```

```

File archivoPersonas = new File("filePersonas.txt");

guardarPersonas(personas, archivoPersonas);

LecturaPersonas(entrada, personas, archivoPersonas);
entrada.close();

}

private static void lecturaPersonas(Scanner entrada, PersonaModelo[] personas,
File archivoPersonas) {
    try (FileReader fr = new FileReader(archivoPersonas);
        BufferedReader br = new BufferedReader(fr);)
    {
        String linea = br.readLine();
        int i = 0;

        while (linea != null) {

            String[] datosPersona = linea.split(",");
            String nombre = datosPersona[0];

            String apellido = datosPersona[1];

            String idioma = datosPersona[2];

            PersonaModelo persona = new PersonaModelo(nombre, apellido,
idioma);

            personas[i] = persona;

            linea = br.readLine();
            i++;

        }

        System.out.println("¿Los datos de qué persona quiere que se
muestran? (0-4)");

        int indicePersona = entrada.nextInt();
        System.out.println(personas[indicePersona]);

    } catch (IOException e) {
        System.out.println("Fichero No Localizado");
    }
}

private static void guardarPersonas(PersonaModelo[] personas, File
archivoPersonas) {
    try (PrintWriter pw = new PrintWriter(archivoPersonas);) {
        for (int i = 0; i < personas.length; i++) {
            pw.println(personas[i]);
        }

    } catch (FileNotFoundException e) {
        System.out.println("Fichero No Localizado");
    }
}
}

```

Eclipse también nos permite ver un histórico de la refactorización que se ha hecho en un proyecto, abriendo el menú **Refactor** → **History**.

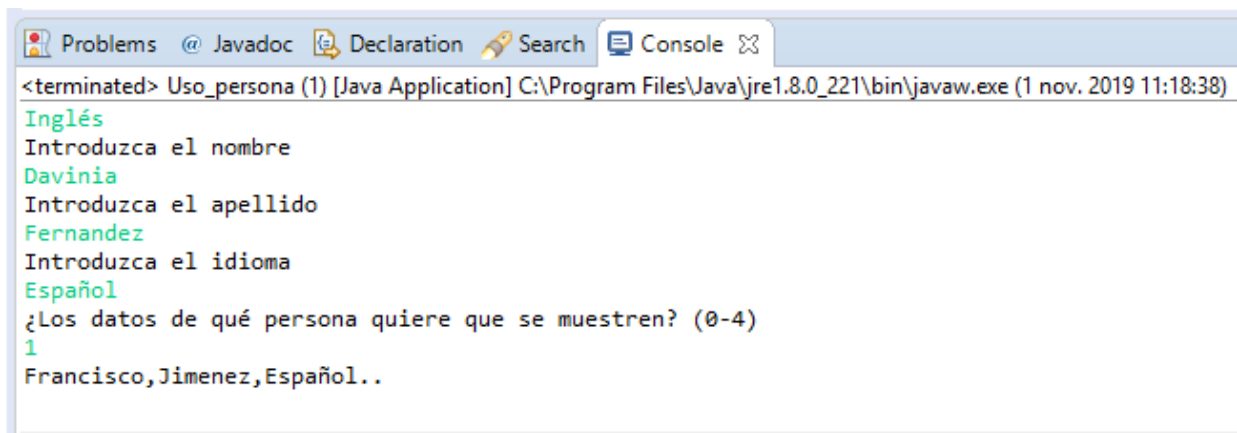
Caso práctico 2: Análisis de código (estático/dinámico) [2,5 puntos]

Al programa generado del caso práctico 1, analizarlo con uno de los analizadores de código, ya sea estático o dinámico, que están disponibles en los IDE de NetBeans o Eclipse.

Indicar qué analizador de código has utilizado.

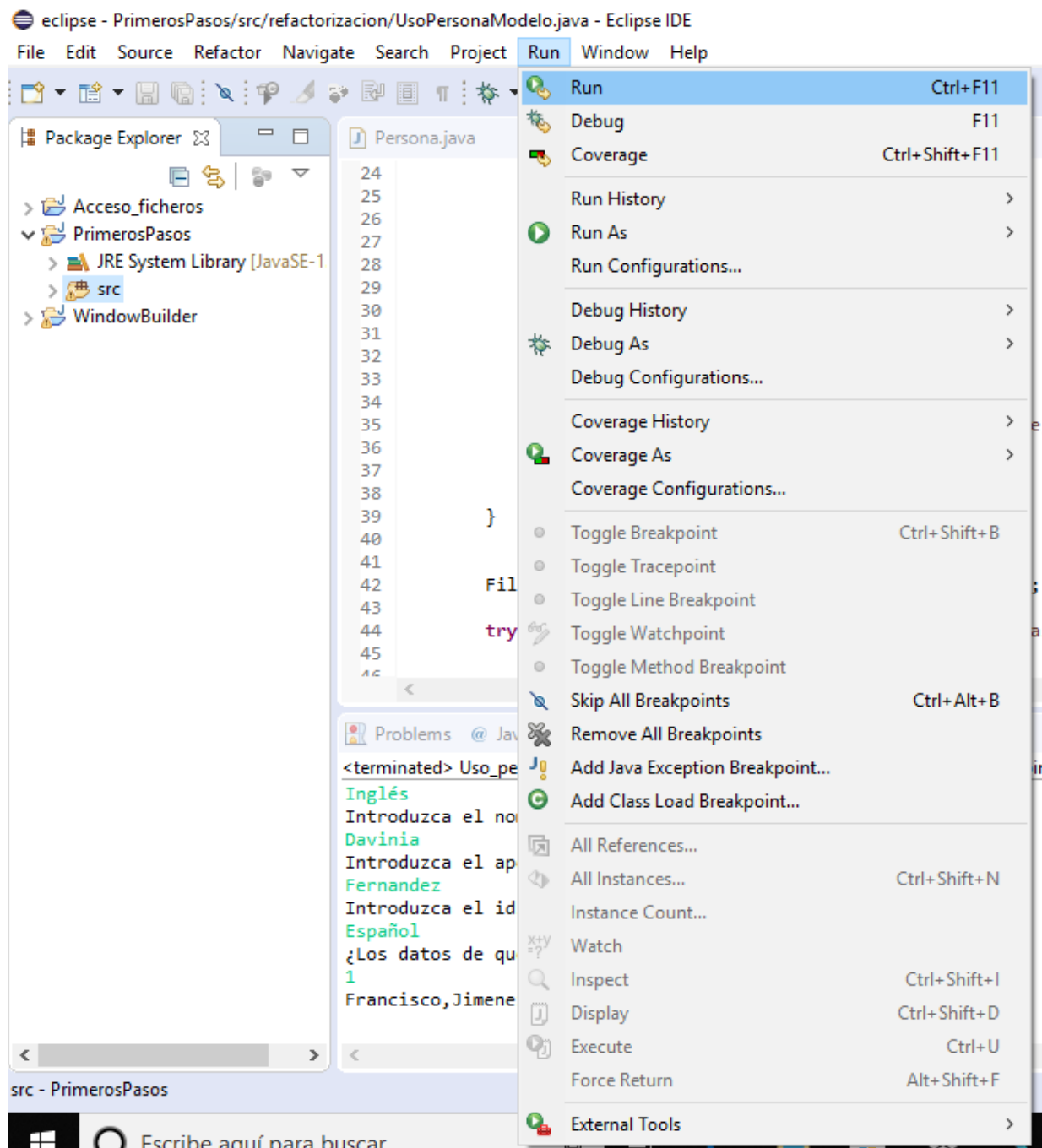
Documentar cada paso, te puedes ayudar con captures de pantalla. Se deberá de visualizar el uso del analizador de código.

A continuación se ejecuta el programa para detectar errores de ejecución, lo cuál se conoce como **Análisis Dinámico**. Comprobamos que el programa nos solicita adecuadamente los datos de cinco personas (nombre, apellido e idioma), instanciando un objeto persona a cada vuelta dada por el bucle for . Con las 5 personas, construye un array de tipo PersonaModelo. Almacena los datos de las cinco personas en un fichero de texto y finalmente ejecuta la lectura de los datos del nº de persona solicitado por el usuario, mostrando dichos datos por consola. Por lo tanto, verificamos que usando el análisis dinámico el programa transcurre sin errores al interactuar con el usuario y devuelve por consola los datos solicitados, sin producirse errores durante su ejecución.

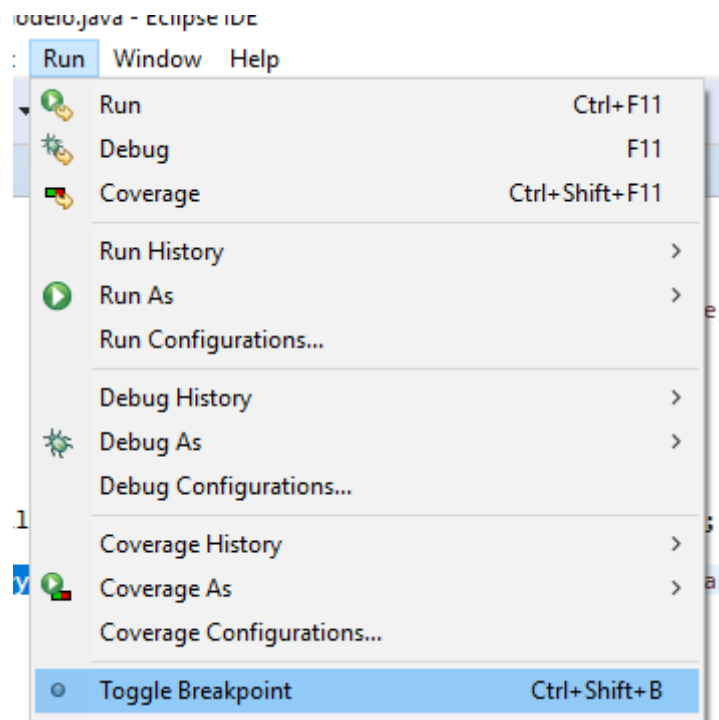


```
<terminated> Uso_persona (1) [Java Application] C:\Program Files\Java\jre1.8.0_221\bin\javaw.exe (1 nov. 2019 11:18:38)
Inglés
Introduzca el nombre
Davinia
Introduzca el apellido
Fernandez
Introduzca el idioma
Español
¿Los datos de qué persona quiere que se muestren? (0-4)
1
Francisco,Jimenez,Español..
```

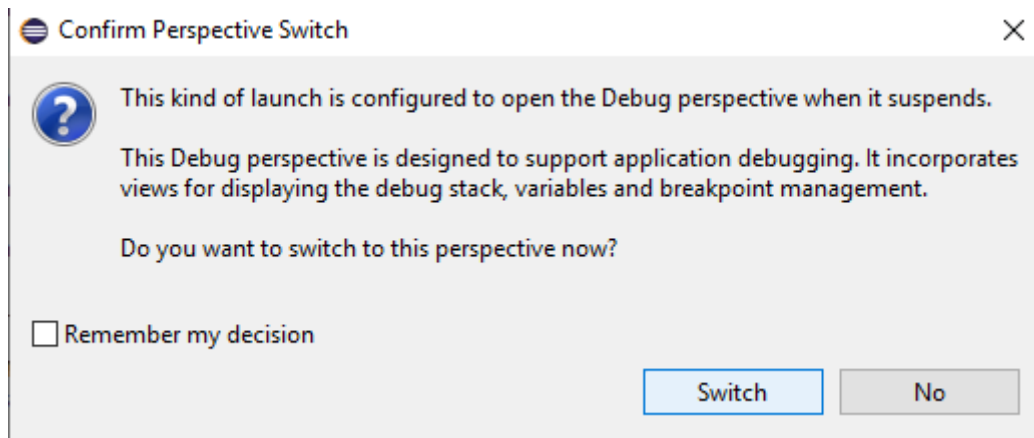
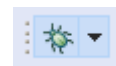
Este análisis dinámico también podría realizarse usando un **debugger**, que tiene por objeto probar y depurar (eliminar) los errores que pudieran existir. Permite establecer puntos de ruptura para que el programa se detenga temporalmente en su ejecución al llegar al punto señalado, y que una vez que analice los datos, continúe la ejecución. En Eclipse se integra la funcionalidad en la pestaña Run:



A modo de ejemplo, se colocan tres puntos de ruptura usando la herramienta **Run → Toggle Breakpoint**



Pasamos al entorno de **Debug Operaciones Test** con el icono:

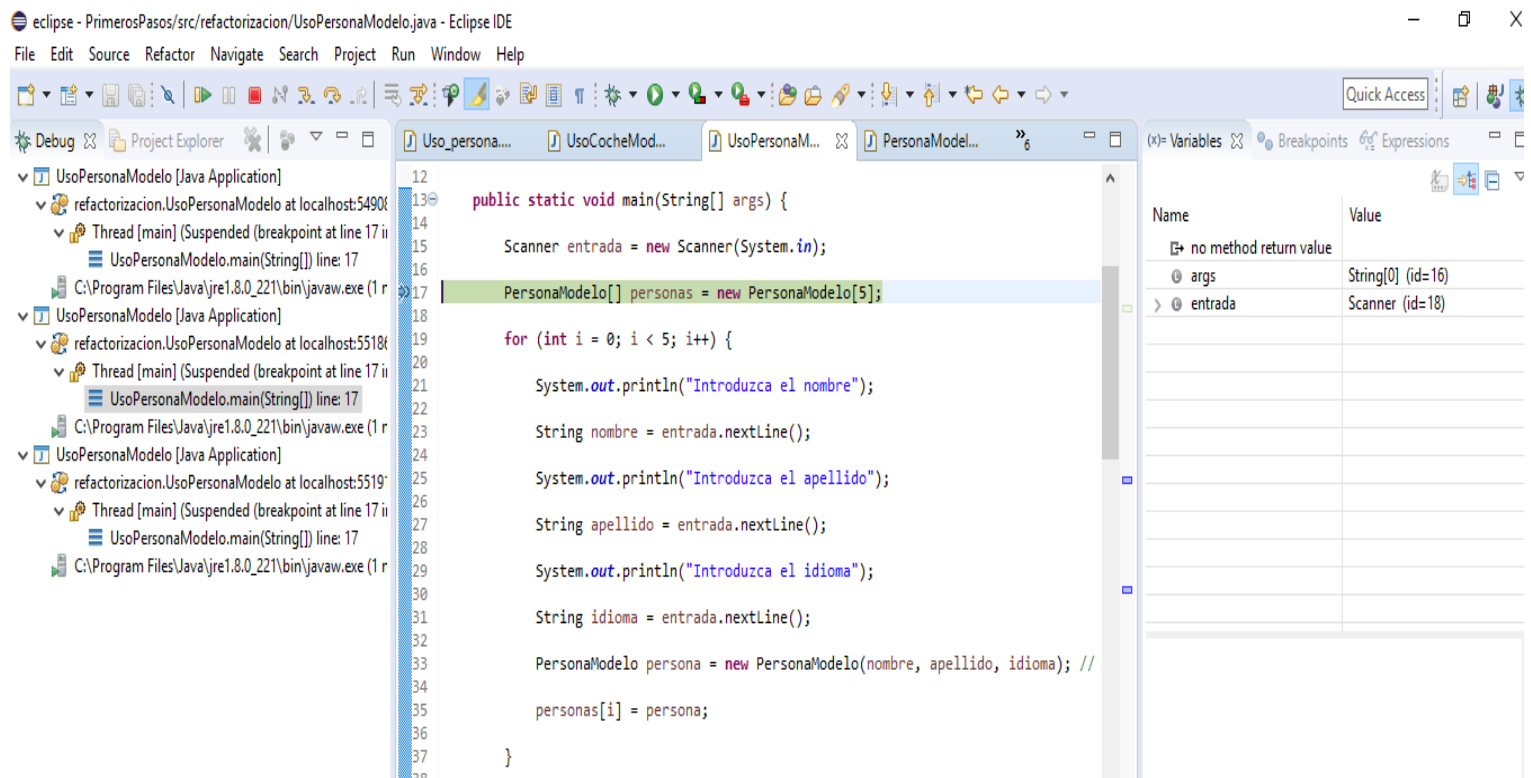


Procedemos a la ejecución: **Run → Debug as → Java Application**

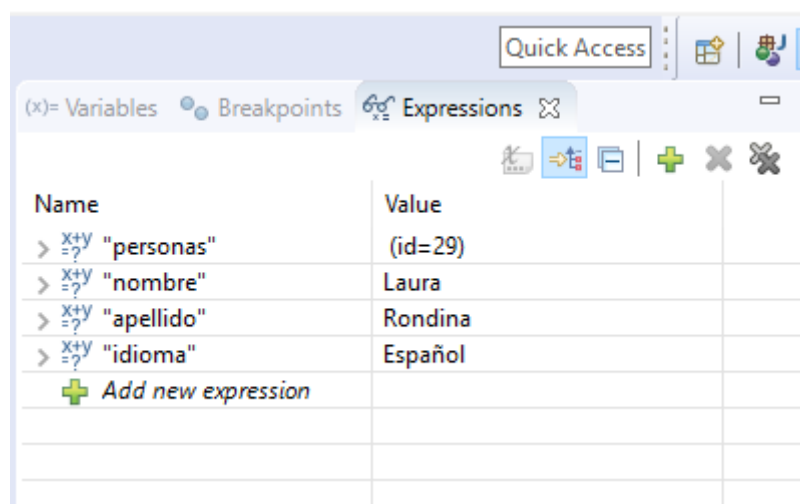
Usamos las siguientes funciones localizadas en la barra de herramientas del entorno:

1. **Resume(F8)**; continúa con la ejecución (hasta el próximo breakpoint).
2. **Suspend**; podemos detener la ejecución aunque no alcancemos un breakpoint (muy útil cuando entramos en un ciclo infinito).
3. **Stop**; detiene la depuración.

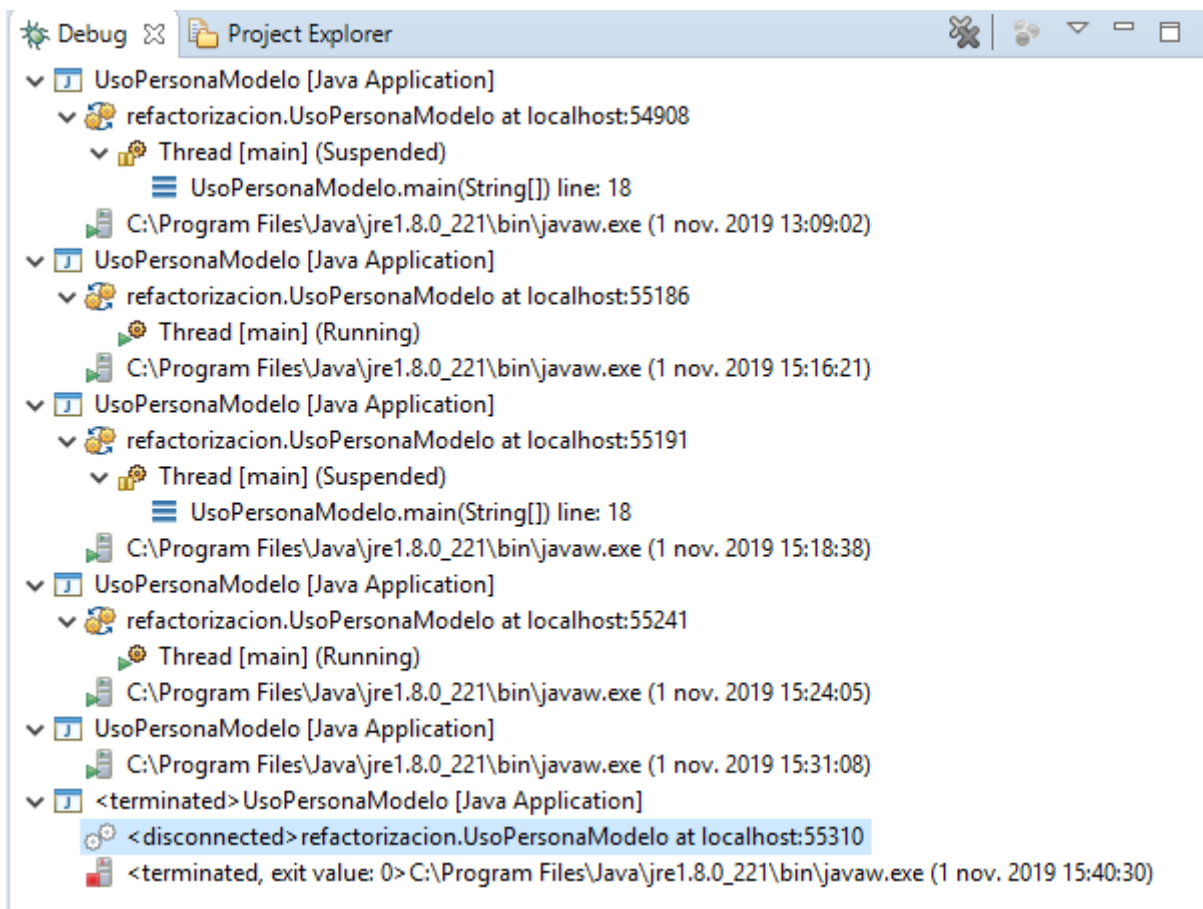
4. **Step Into (F5)**; se detiene en la primer línea del código del método que estamos ejecutando. Si no hay método, hace lo mismo que Step Over.
5. **Step Over (F6)**; pasa a la siguiente línea que vemos en la vista de código.
6. **Step Return (F7)**; vuelve a la línea siguiente del método que llamó al método que se está depurando actualmente. O lo que es lo mismo, sube un nivel en la pila de ejecución, que vemos en la vista Debug.



También podemos ir comprobando el valor de las variables o constantes de interés a medida que ejecutamos el Run en modo Debug as. Cuando tengamos marcada en verde la línea, señalar la variable en cuestión, clic en botón derecho del ratón, y selección de la opción **Watch**. En el panel derecho se constatará el valor que toma en cuanto avancemos la línea, lo cuál nos permite detectar si hubo algún error durante la toma de valor.



En este caso la ejecución del programa finaliza sin incidencias.



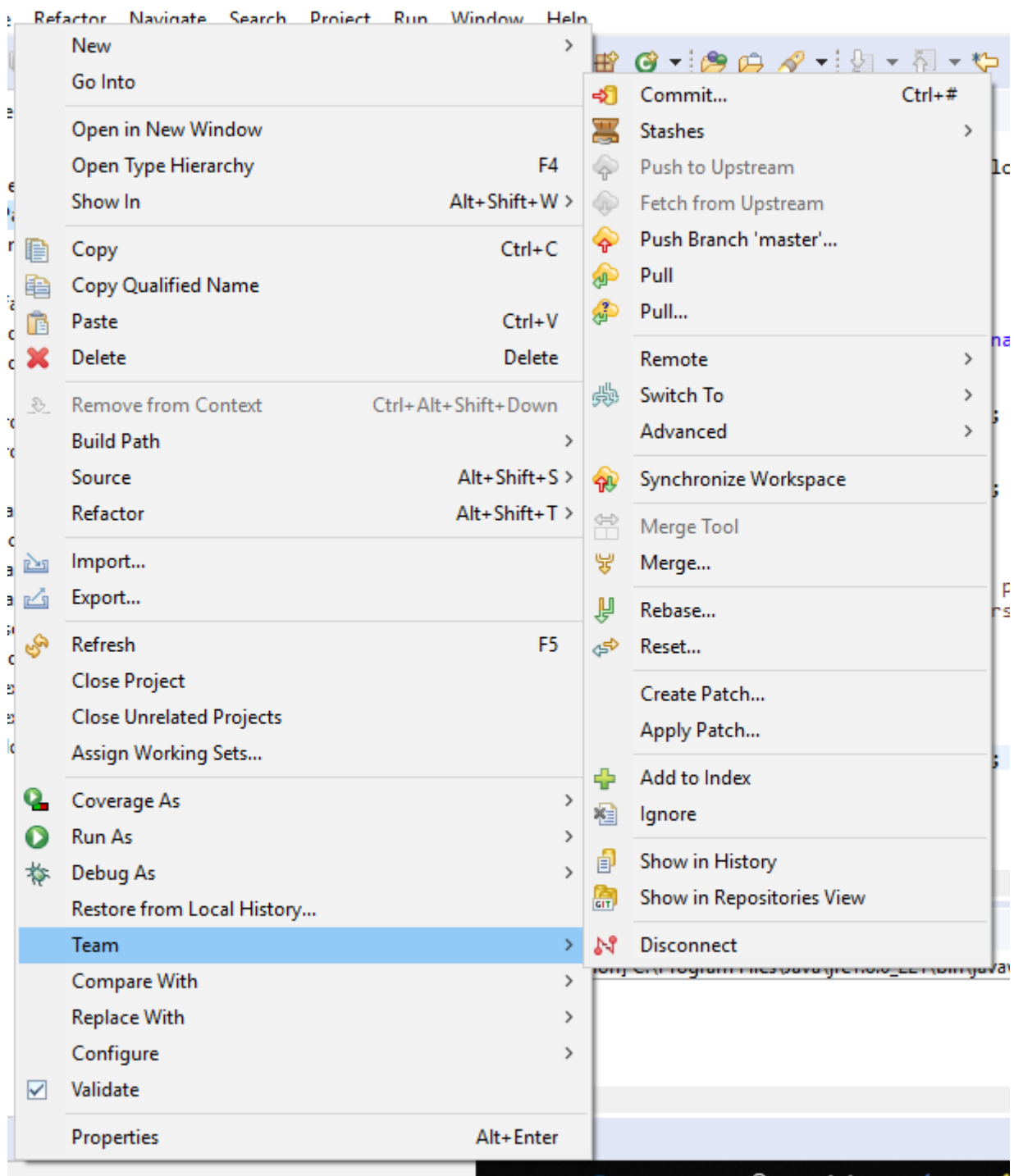
Caso práctico 3: Control de versiones. [2,0 puntos]

Genera un control de versiones al programa del caso práctico 1, al menos que aparezcan dos versiones. Utilizar uno de los clientes de control de versiones que viene integrado dentro del IDE NetBeans o Eclipse. Indicar qué cliente de control de versiones has utilizado.

Documentar cada paso, te puedes ayudar con capturas de pantalla. Se deberá de visualizar el uso del control de versiones.

En este caso se ha seguido el proceso para subir un proyecto Java con eclipse a **GitHub**. Esta herramienta nos permitirá tanto compartir nuestro código, como tenerlo disponible en la nube para posteriormente poder descargarlo a cualquier lugar. El plugin de Eclipse eGit se puede descargar desde la página oficial, o bien agregar al eclipse a través de Help > Eclipse Marketplatce. Aunque lo más probable es que la distribución de eclipse ya venga con un plugin de Git incorporado, el cuál ha sido mi caso.

Para acceder a todas las opciones de **GIT** desde Eclipse debemos pulsar el botón derecho del ratón sobre el proyecto o un archivo del mismo y acceder al elemento **“TEAM”**, y aquí encontraremos todas las opciones para trabajar con **GIT**.



Necesitamos también crear una cuenta y un repositorio en la web oficial de GitHub:

<https://github.com/>

La mía es laura1985 y el nombre de mi repositorio es ProbandoGitHub. Obtenemos así la ruta que necesitaremos en pasos siguientes.

Inicialmente nuestro repositorio está vacío, pero podremos ir compartiendo nuestros contenidos.

Owner



laura1985 ▾

/

Repository name *

ProbandoGitHub



Great repository names are short and memorable. Need inspiration? How about **fictional-funicular**?

Description (optional)



Public

Anyone can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ Initialize this repository with a README

This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▾

Add a license: **None** ▾



Create repository

laura1985 / ProbandoGitHub

Unwatch ▾ 1

★ Star 0

🍴 Fork 0

Code

Issues 0

Pull requests 0

Projects 0

Wiki

Security

Insights

Settings

Quick setup — if you've done this kind of thing before

Set up in Desktop or **HTTPS** **SSH** `https://github.com/laura1985/ProbandoGitHub.git`



Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# ProbandoGitHub" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/laura1985/ProbandoGitHub.git
git push -u origin master
```



...or push an existing repository from the command line

```
git remote add origin https://github.com/laura1985/ProbandoGitHub.git
git push -u origin master
```

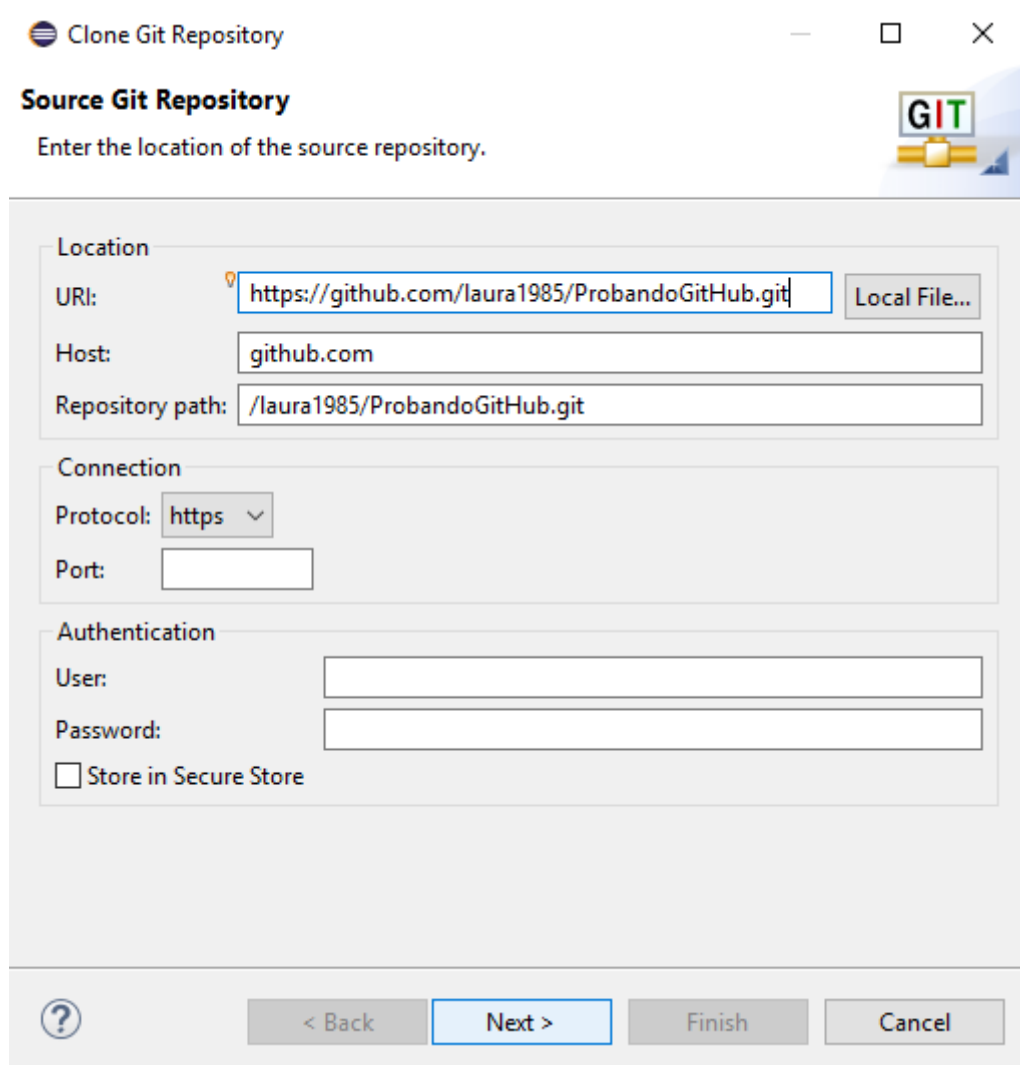


Ya en eclipse, el primer paso es pulsar botón derecho sobre el proyecto e ir a la opción:

Team → **Share** (compartir).

Realizada esta operación nos aparece una nueva ventana donde elegimos qué carpeta será nuestro repositorio.

Cuando creamos un repositorio en GitHub, existe un repositorio *remoto*. Se puede clonar tu repositorio para crear una copia *local* en tu computadora y sincronizarla entre las dos ubicaciones (icono Clone a Git Repository and add the clone to this view).



The screenshot shows the 'Clone Git Repository' dialog box in Eclipse IDE. The window title is 'Clone Git Repository'. Below the title bar, there's a section titled 'Source Git Repository' with a subtitle 'Enter the location of the source repository.' and a small Git logo. The dialog is divided into three main sections: 'Location', 'Connection', and 'Authentication'. In the 'Location' section, the 'URI:' field contains 'https://github.com/laura1985/ProbandoGitHub.git', the 'Host:' field contains 'github.com', and the 'Repository path:' field contains '/laura1985/ProbandoGitHub.git'. There is a 'Local File...' button next to the URI field. In the 'Connection' section, the 'Protocol:' dropdown is set to 'https' and the 'Port:' field is empty. In the 'Authentication' section, there are fields for 'User:' and 'Password:', and a checkbox labeled 'Store in Secure Store' which is currently unchecked. At the bottom of the dialog, there are four buttons: a help button (question mark icon), '< Back', 'Next >', and 'Cancel'. The 'Next >' button is highlighted with a blue border.

Clone Git Repository

Source Git Repository
Enter the location of the source repository.

Location

URI: Local File...

Host:

Repository path:

Connection

Protocol: v

Port:

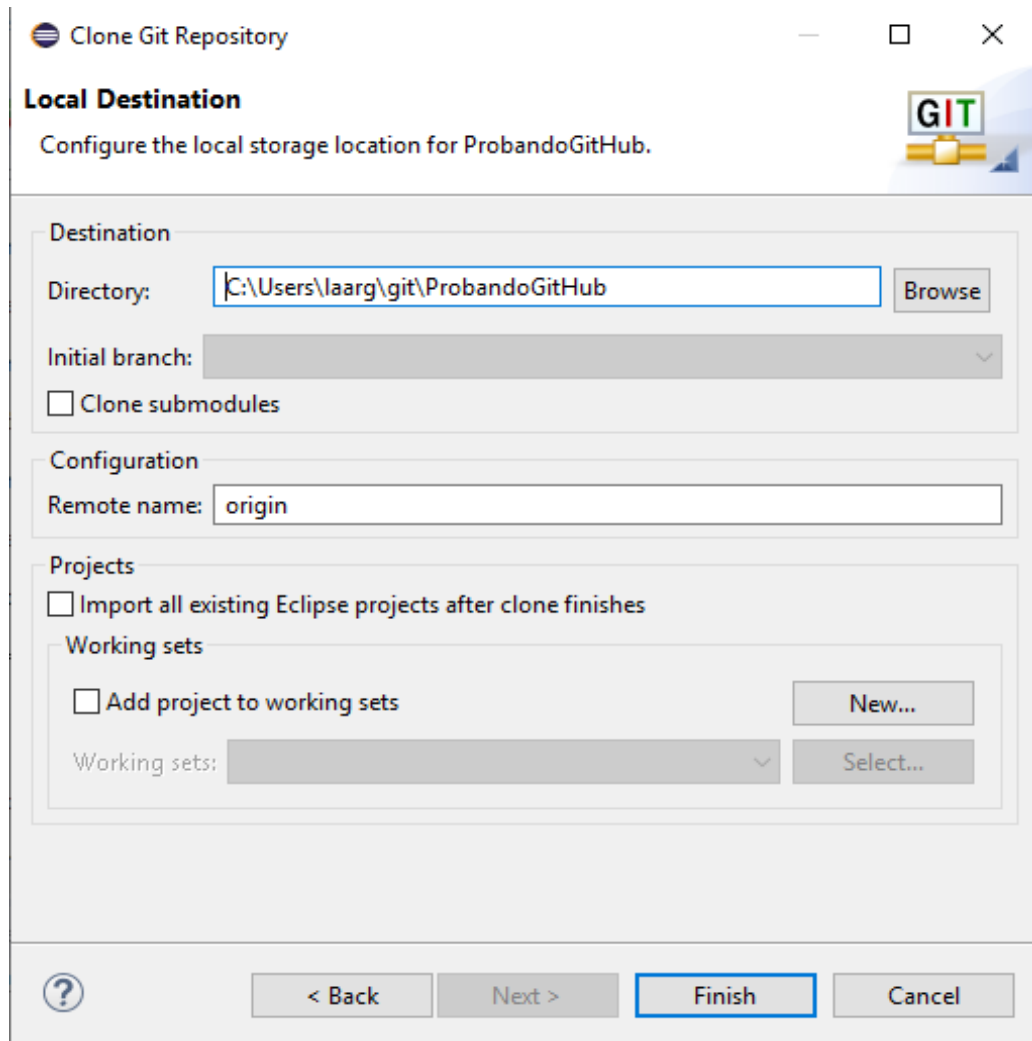
Authentication

User:

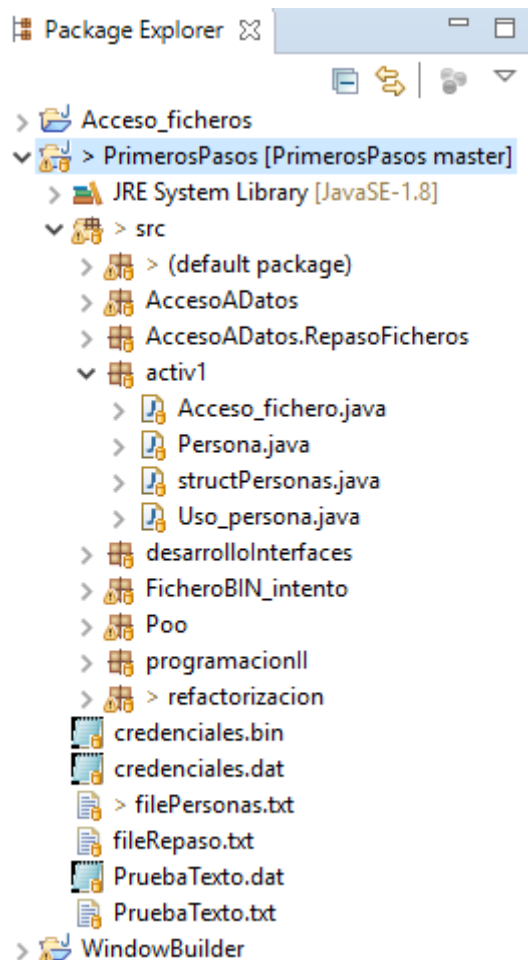
Password:

☐ Store in Secure Store

? < Back Next > Finish Cancel

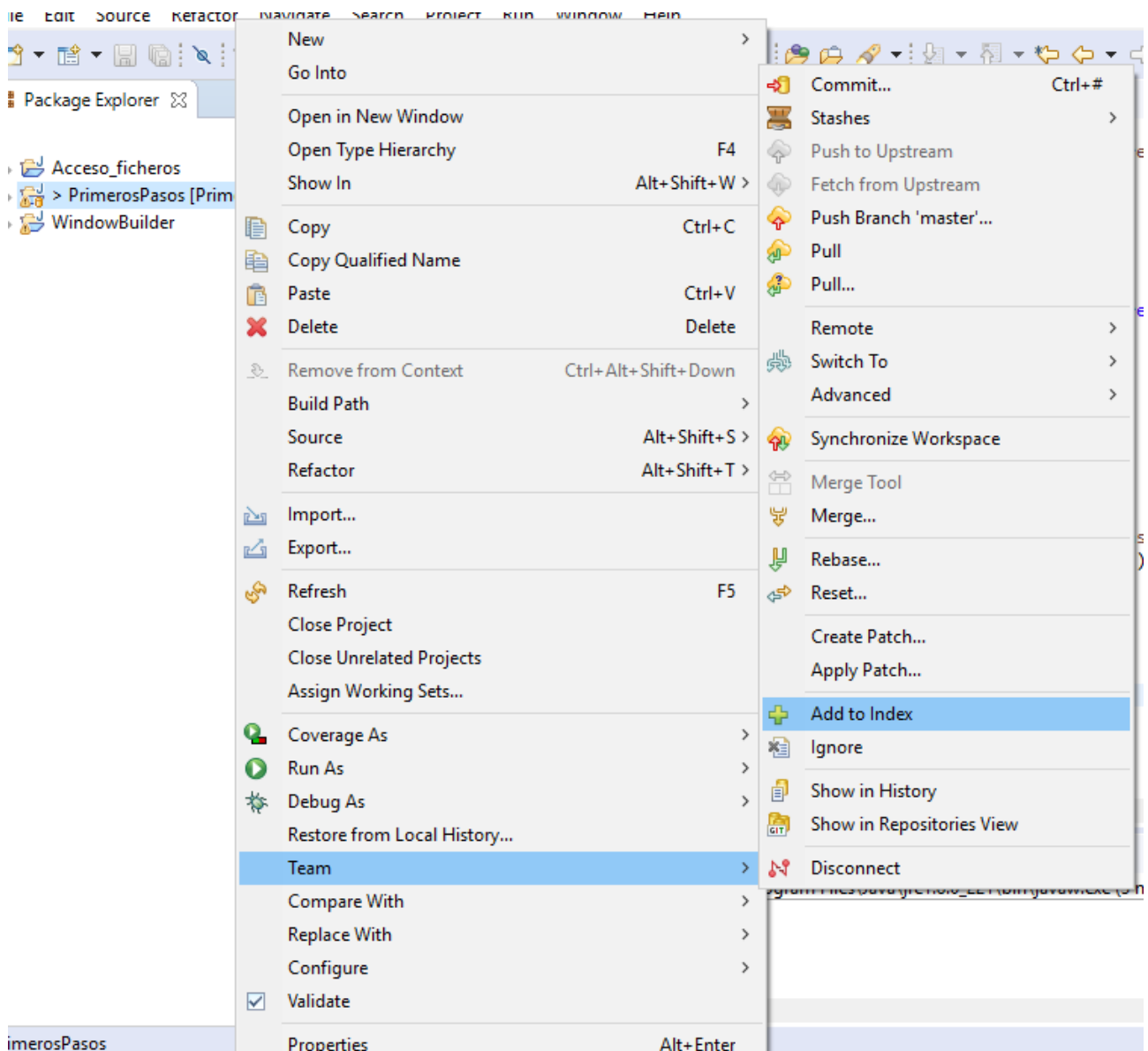


Pulsamos finalizar en las ventanas que nos aparecen y los iconos del proyecto de eclipse cambiarán.



Ya tenemos a nuestra disposición todos los ficheros para añadirlos al repositorio. Sobre el proyecto, Clic en botón derecho y :

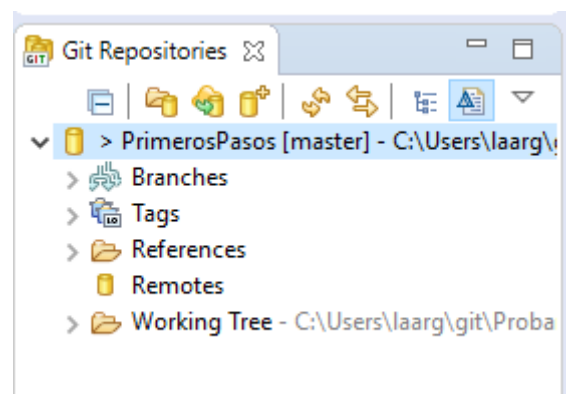
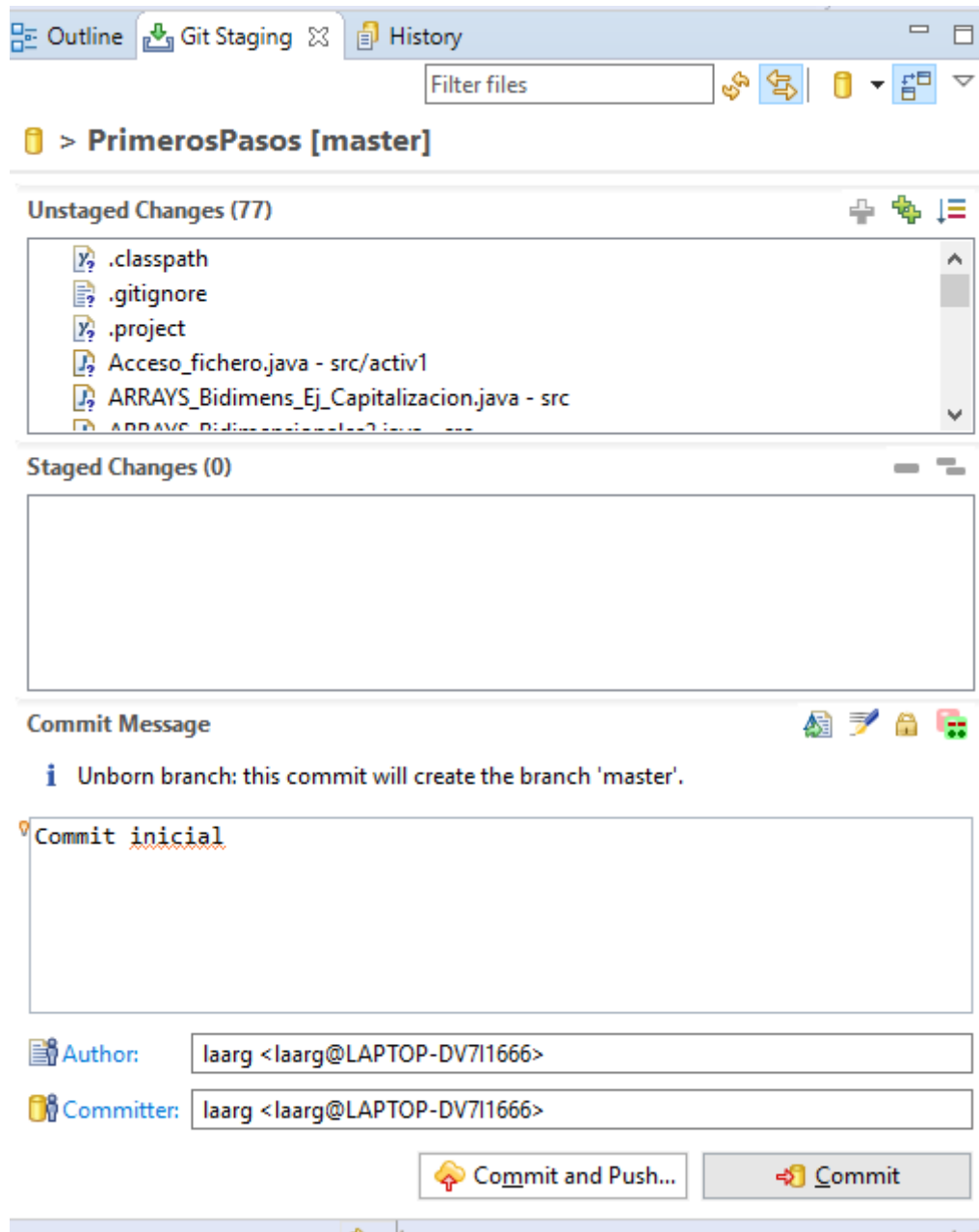
Team → Add to Index



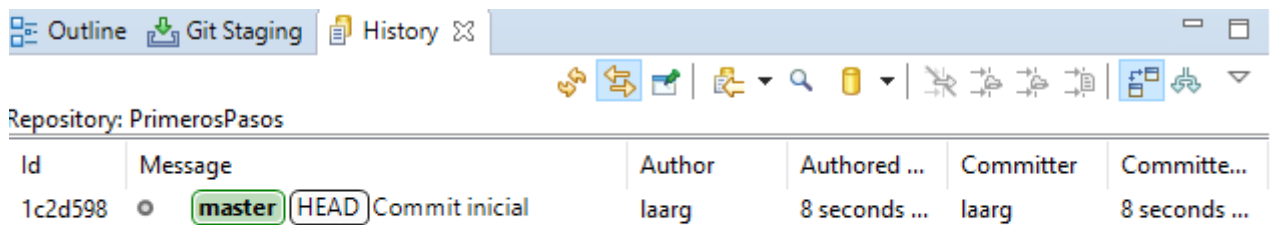
Acabamos de añadir todos los ficheros del proyecto para que se agreguen a GIT. Es momento de lanzar el comando de commit y añadir los ficheros al repositorio. Clic derecho sobre proyecto y:

Team → Commit...

Introducimos el **mensaje** que nos solicita el comando de commit (en mi caso, “commit inicial”). Pulsamos sobre el botón **Commit**, y ya tenemos salvados los ficheros en el repositorio local. Acabamos de construir nuestro primer repositorio con GIT local.

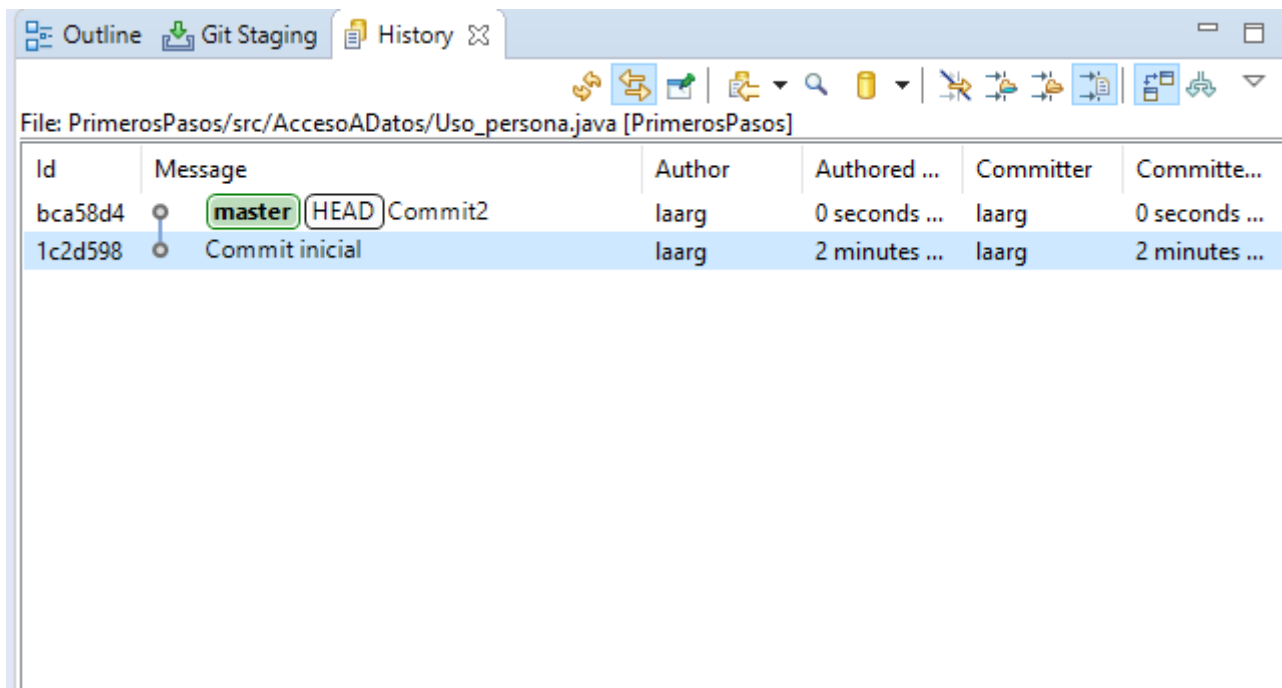


Se queda registrada la primera versión del programa:



Id	Message	Author	Authored ...	Committer	Committe...
1c2d598	Commit inicial	laarg	8 seconds ...	laarg	8 seconds ...

Realizamos algunos cambios, y registramos la segunda versión con el mensaje “Commit2”:



Id	Message	Author	Authored ...	Committer	Committe...
bca58d4	Commit2	laarg	0 seconds ...	laarg	0 seconds ...
1c2d598	Commit inicial	laarg	2 minutes ...	laarg	2 minutes ...

*** **Nota:** SI CERRAMOS EL PANEL DE GIT REPOSITORIES en nuestro IDE de Eclipse, podemos proceder a su apertura cuando queramos utilizarlo a través de la siguiente ruta:

