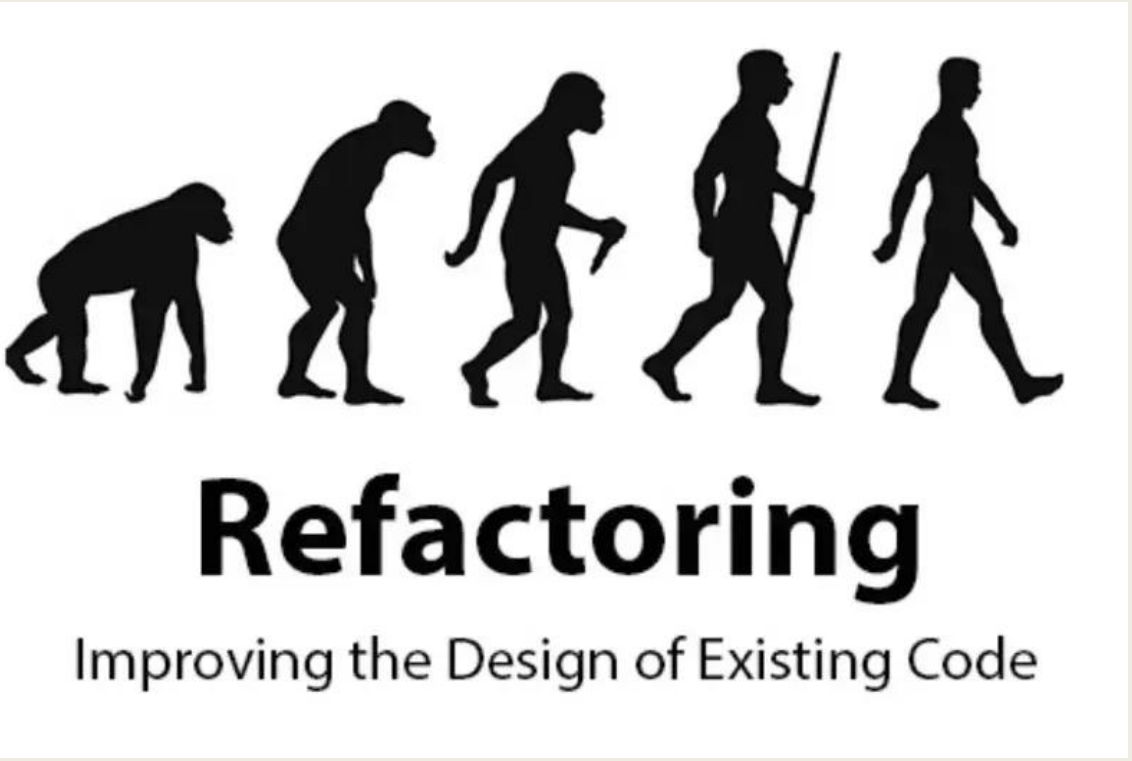
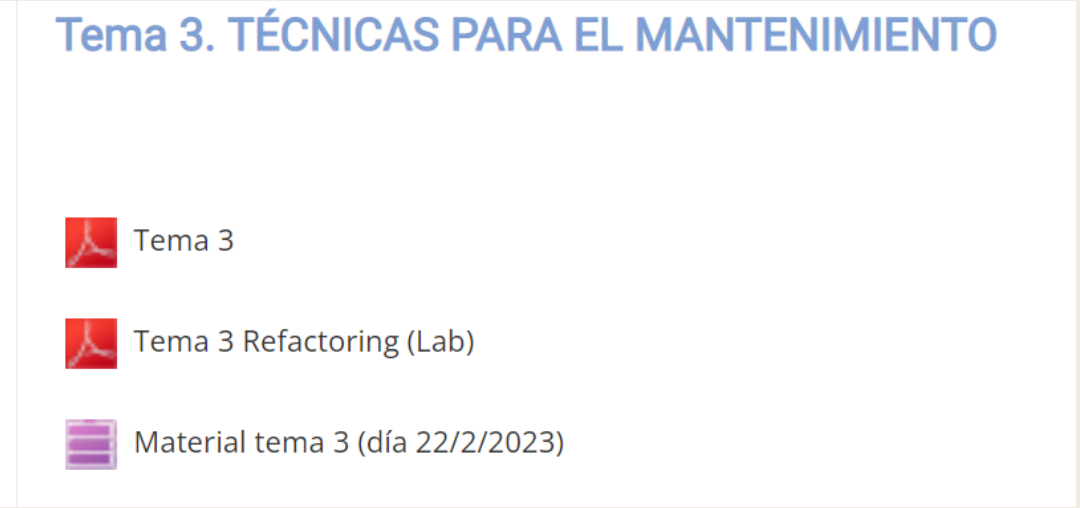


PRACTICAS DÍA 2 REFACTORING EN ECLIPSE

3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.



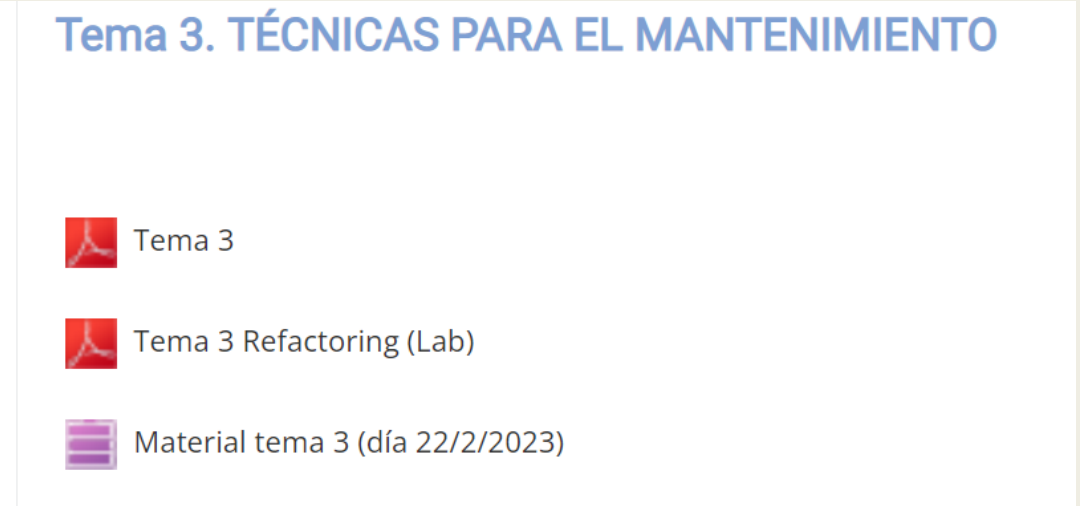
En esta sesión vamos a practicar varios casos de refactorización y cómo son realizados en el entorno eclipse, como vimos en la sesión anterior.

Objetivos

Los objetivos de esta sesión son los siguientes:

- Conocer qué es la refactorización.
- Estudiar algunos procesos de refactorización
- Aplicar los procesos de refactorización a un proyecto de cierta envergadura, dentro del entorno eclipse.

3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.



Paso 1: Crear un proyecto y cargar las clases.

Descargue el material, descomprímalo, e importe el directorio descargado a Eclipse.

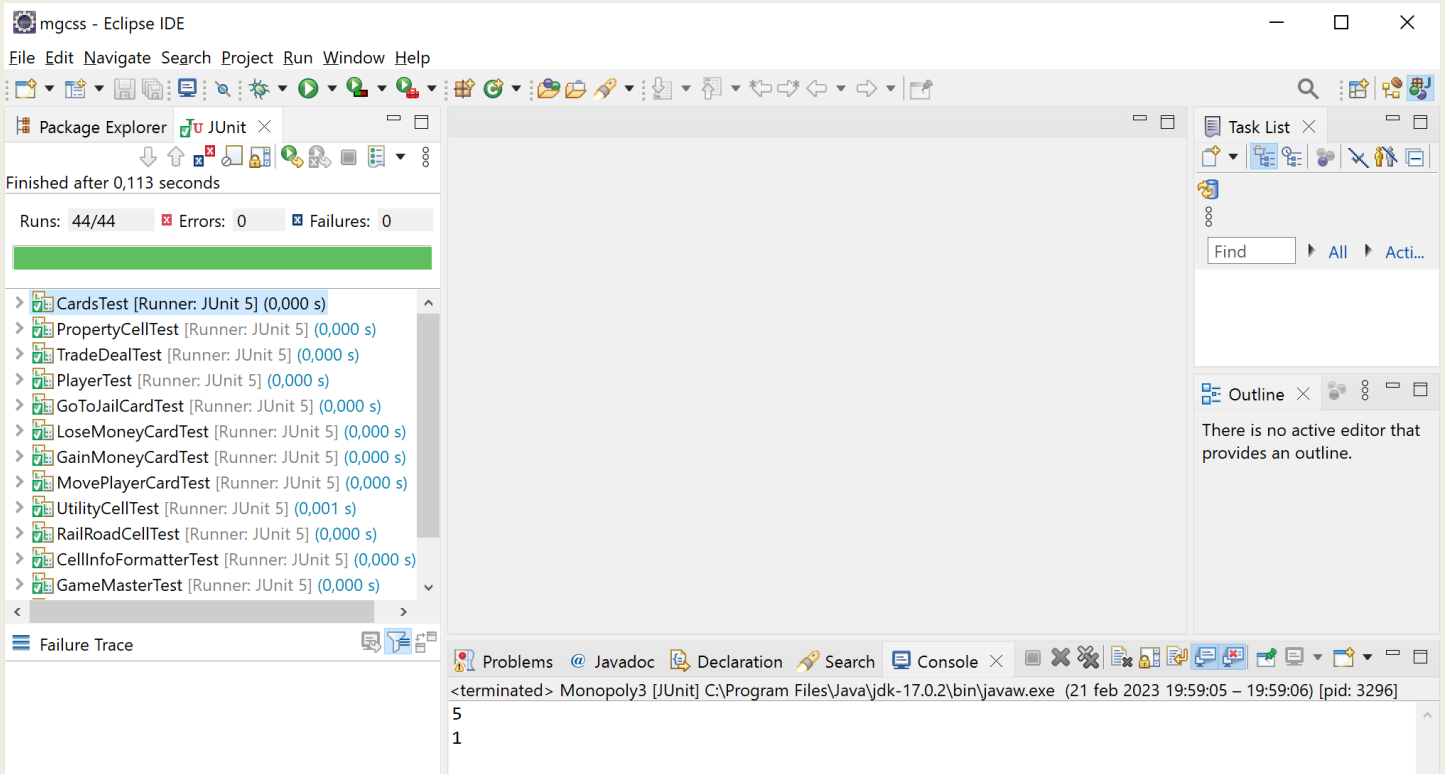
Una vez cargadas las clases, debemos tener todas las clases bajo 2 paquetes. Sin embargo, algunas clases podrían estar marcadas por el indicador rojo de error. Los indicadores deben desaparecer después de añadir las clases de Junit 5 al *Build-path* del proyecto.

Para ello, pulse con el botón derecho en la raíz del proyecto y seleccione *Properties*, A continuación seleccionar la opción *Java Build Path*, y en la parte derecha seleccionar *Libraries*. Finalmente pulsar en el botón "*Add Library*" y seleccionar la librería Junit 5.

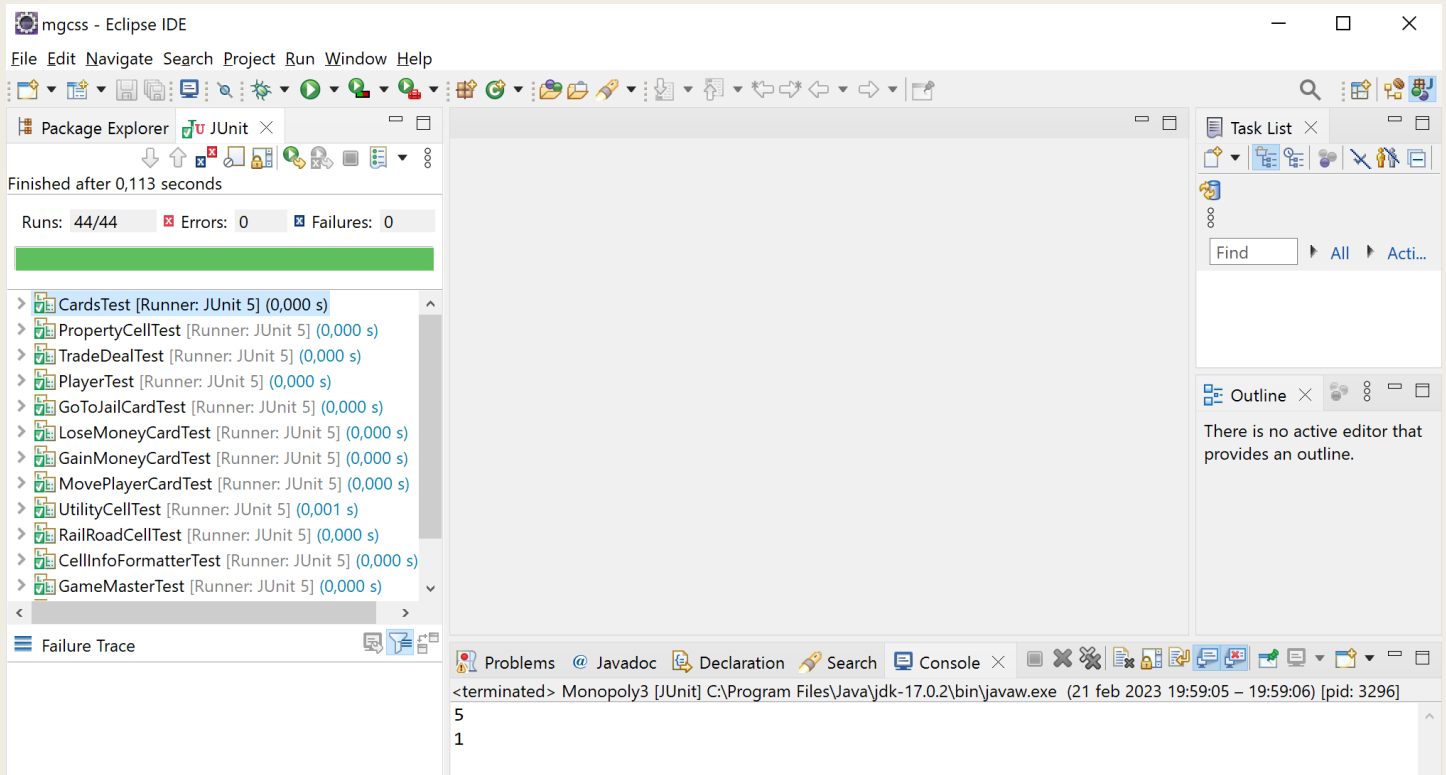
Paso 2: Probar que funciona.

Vamos a ver que la aplicación funciona correctamente, y que supera satisfactoriamente todos los casos de prueba. Nos posicionamos en la raíz del proyecto y pulsando el botón derecho, seleccionamos la opción ***Run As >> JUnit Test***. Si no hay ningún contratiempo, en la pestaña Junit comprobareis que todos los casos de prueba se han ejecutado correctamente.

Ejercicio: Vamos a modificar un caso de prueba para que falle cambiando algún dato. Por ejemplo, en la clase CardTest.java en el método testCardType() modificar la última aserción para que se compruebe con TYPE_CC en vez de TYPE_CHANCE. Ejecutar de nuevo los casos de prueba y ver que tipo de fallo nos da JUnit. A continuación ejecutamos el fichero Main.java para ver que la aplicación se ejecuta correctamente.

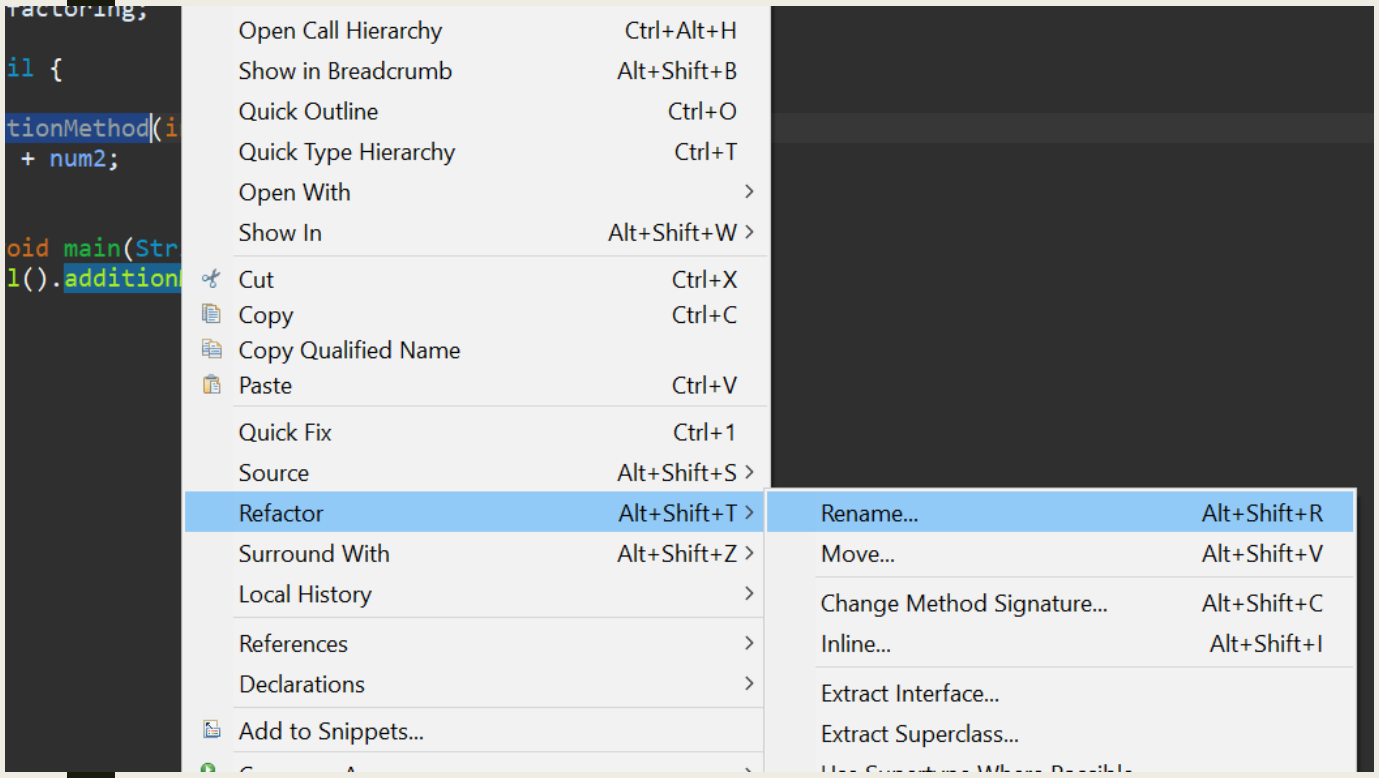


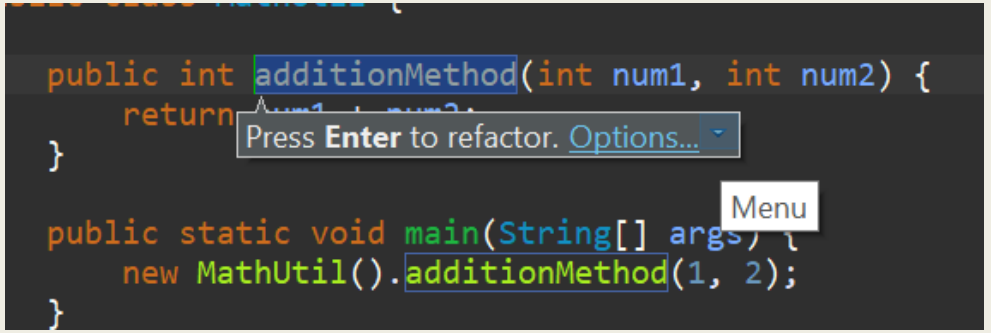
Iniciamos las refactorizaciones...

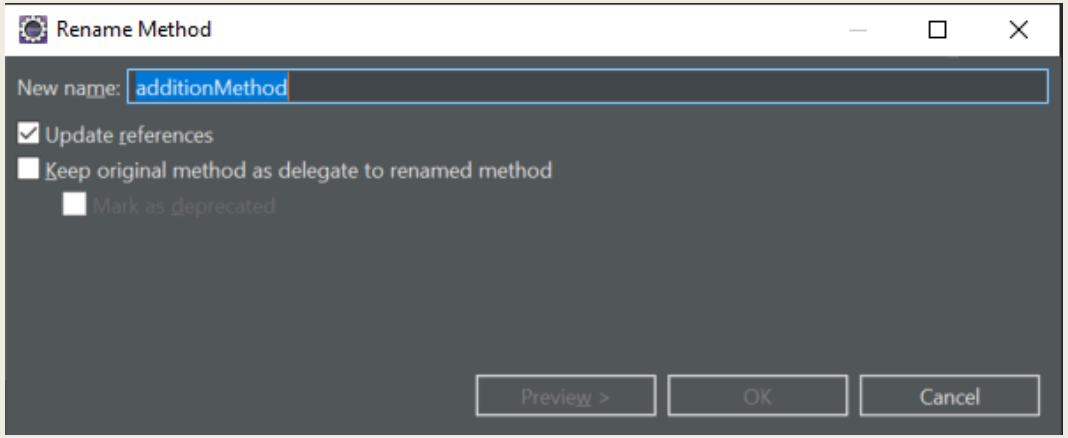


En la sesión anterior...

Renombrar: Facilita la comprensión de la utilidad de la variable o el método. Existe un método avanzado para modificar las referencias en otras clases.





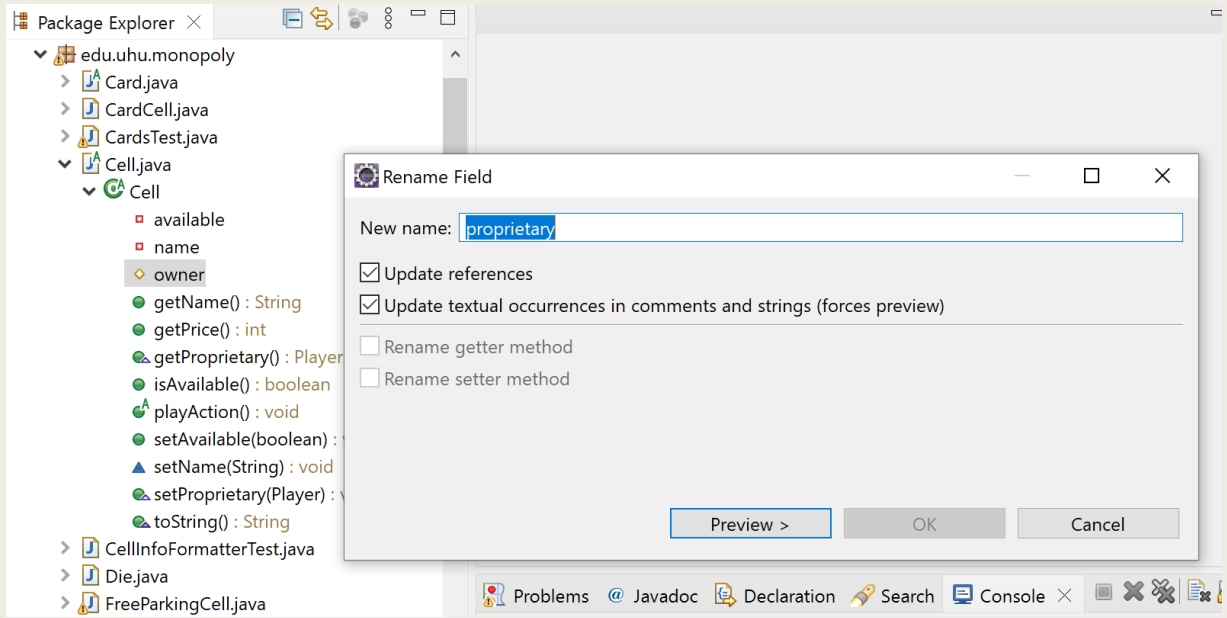


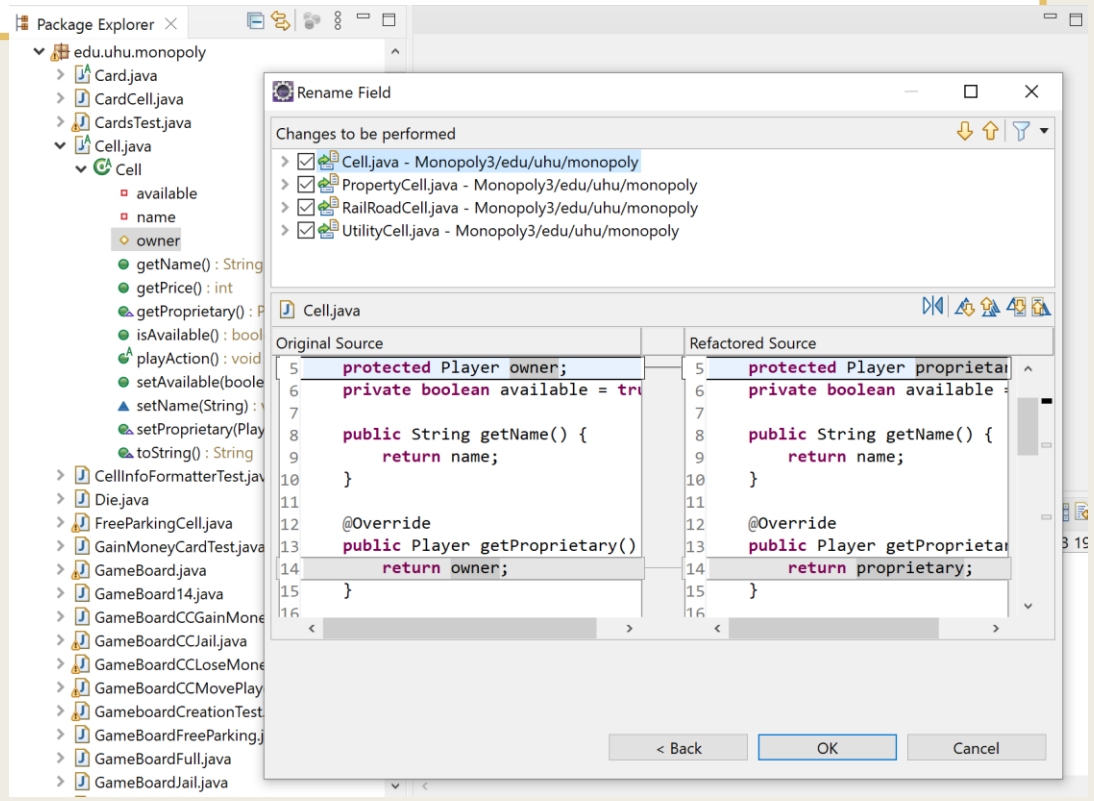
Refactorización 1: Renaming a Class Field.

La clase Cell es una clase abstracta con muchas subclases. Se puede ver la jerarquía de clases posicionándonos en una clase y pulsando el botón derecho seleccionando la opción "*Open Type Hierarchy*". También puedes observar que la clase Cell tiene un atributo *owner*. En este ejercicio vamos a modificar el nombre del atributo *owner* por *proprietary*.

En la pantalla que aparece a continuación selecciona todas las opciones del menú (para cambiar la referencias, comentarios y los métodos getters y setters). Antes de realizar el cambio, pincha en la opción *Preview* para ver los cambios que se van a realizar.

Finalmente pulsa en la botón OK. Hace realmente esta operación todos los cambios? Utiliza la operación Search>>Java para buscar todas las referencias al campo *owner* para ver que ha sido modificado. A continuación busca todas las referencias al campo *proprietary* para ver donde ha sido modificado.



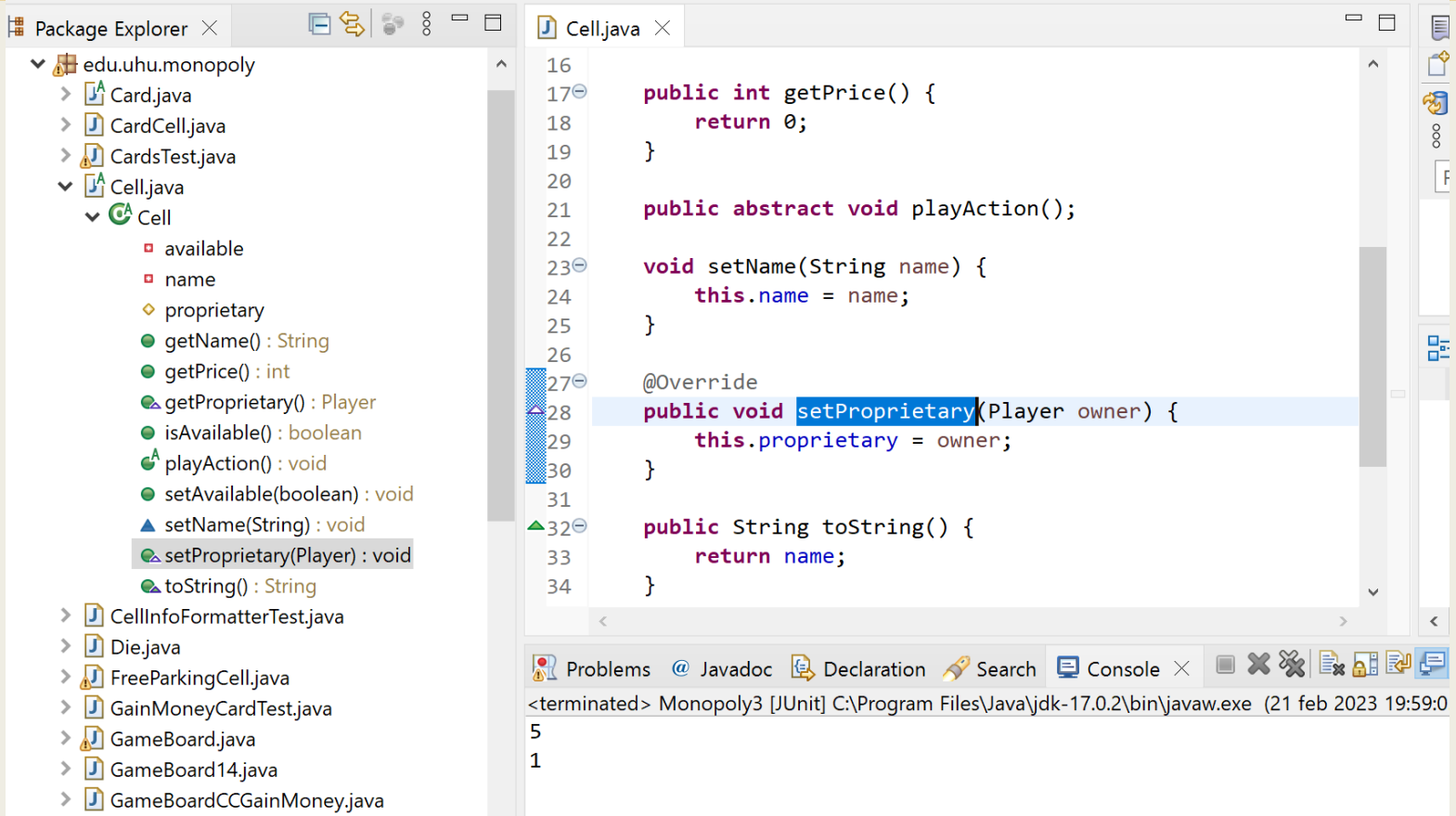


Refactorización 1: Renaming a Class Field.

Sin embargo, el parámetro del método *Cell.setProprietary* todavía sigue teniendo el identificador owner. La refactorización realizada únicamente modifica el nombre del atributo definido en Cell; puede haber otras variables (parámetros, variables locales) con ese mismo nombre.

Si hubiésemos realizado la operación global *Search-and-replace* hubiéramos renombrado todos los strings, sin embargo, eclipse conoce la estructura de tu programa java y únicamente realiza los cambios oportunos.

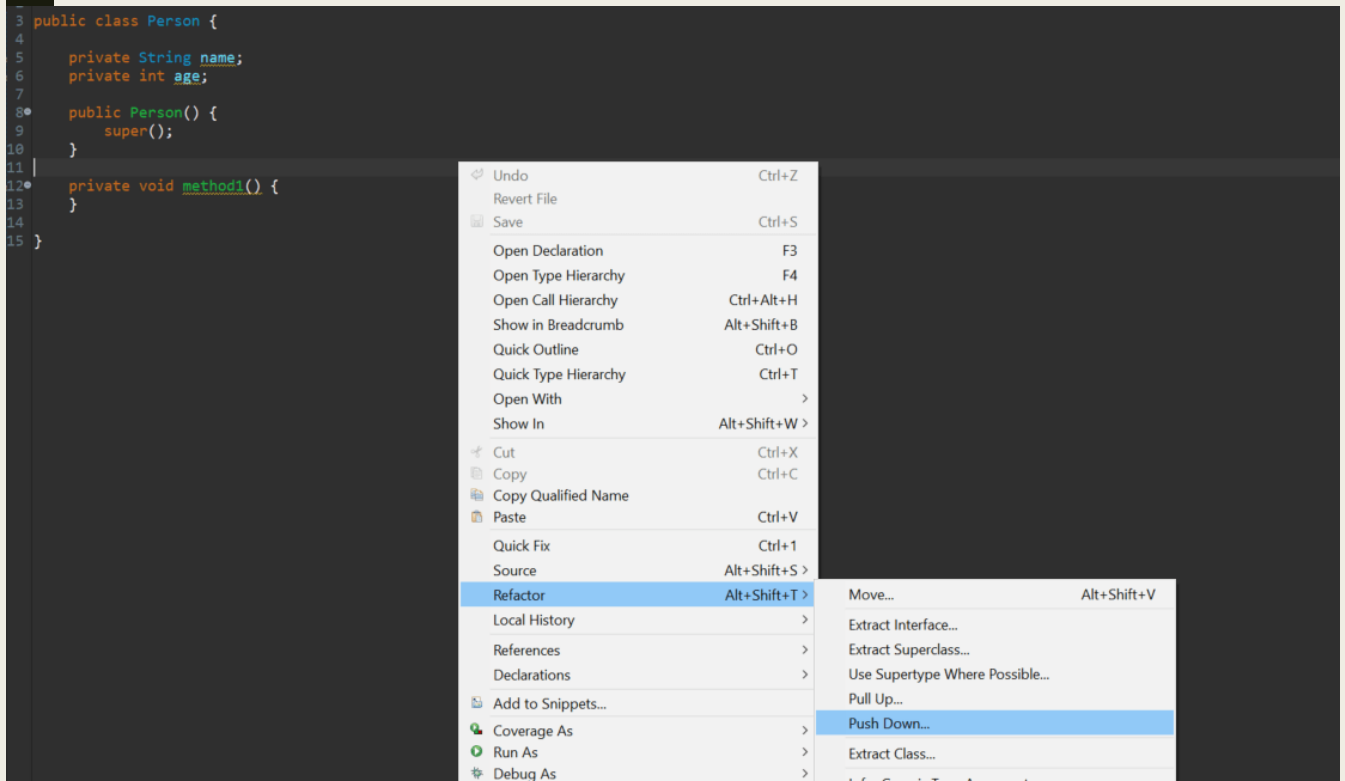
Pon el proyecto bajo revisión en Github y publica una versión después de cada Refactorización.

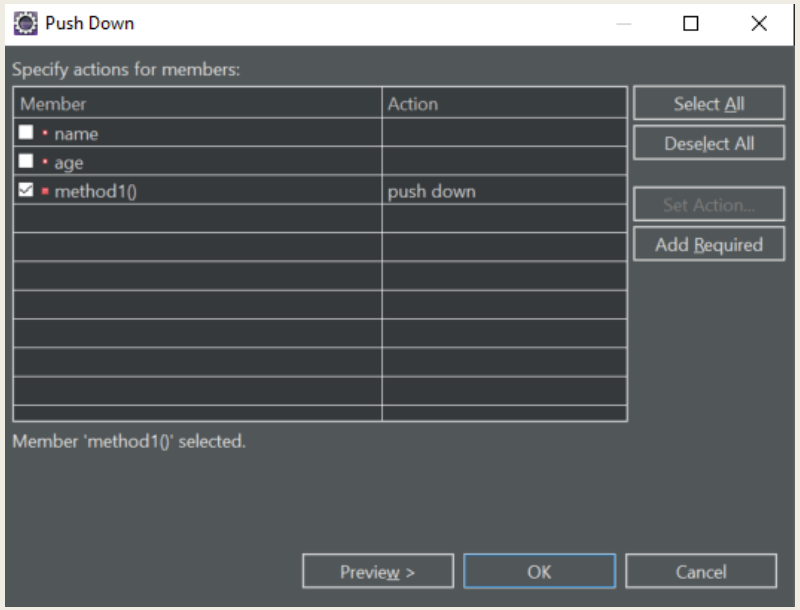


3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Push Down and Pull Up: Si tenemos una relación padre-hijo (como nuestro ejemplo anterior de Empleado y Persona) entre nuestras clases, y queremos mover ciertos métodos o variables entre ellas, podemos usar las opciones push/pull proporcionadas por Eclipse.

Como sugiere el nombre, la opción Push Down mueve métodos y campos de una clase principal a todas las clases secundarias, mientras que Pull Up mueve métodos y campos de una clase secundaria en particular a la principal, lo que hace que ese método esté disponible para todas las clases secundarias.

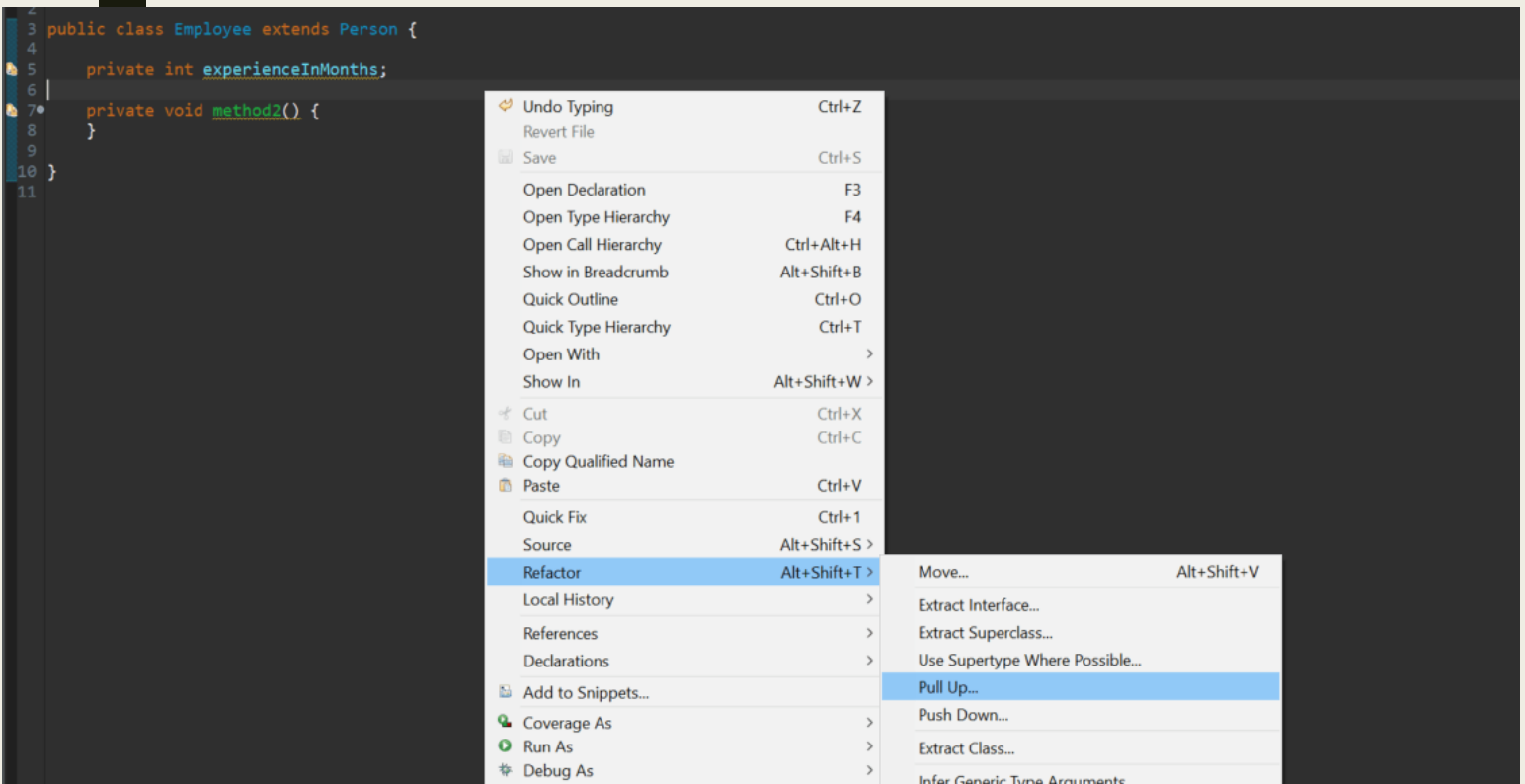


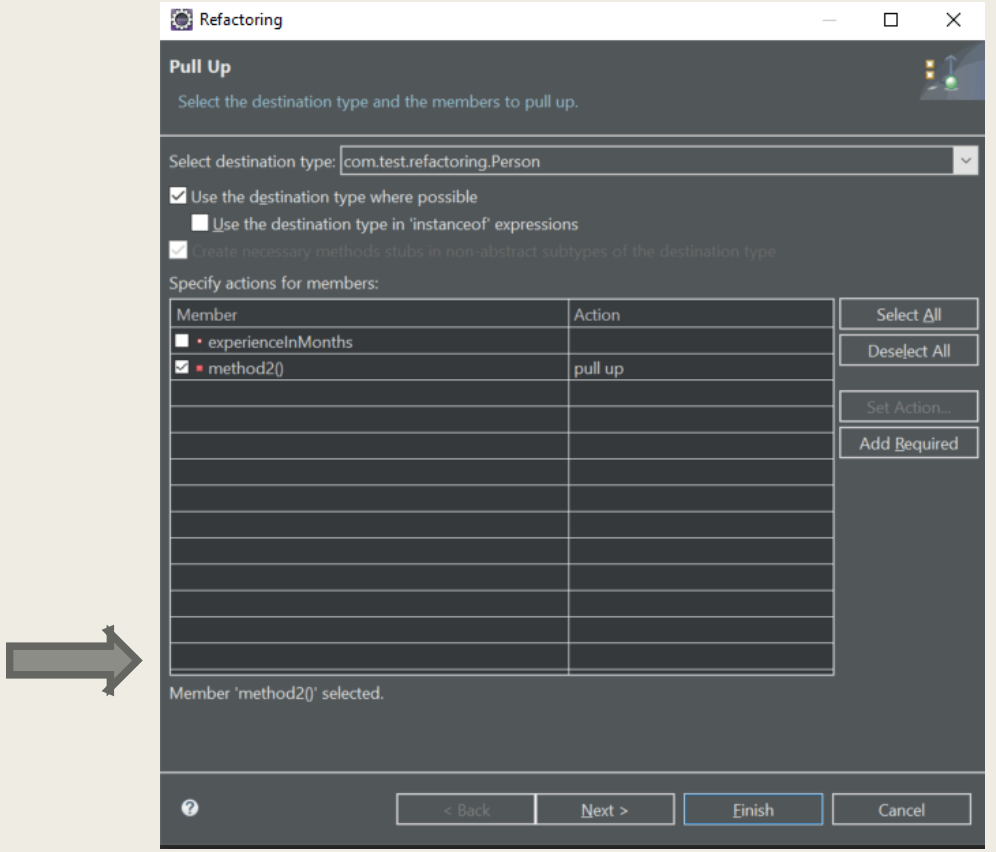


3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Push Down and Pull Up: Si tenemos una relación padre-hijo (como nuestro ejemplo anterior de Empleado y Persona) entre nuestras clases, y queremos mover ciertos métodos o variables entre ellas, podemos usar las opciones push/pull proporcionadas por Eclipse.

De manera similar, para mover métodos de una clase secundaria a una clase principal, debemos hacer clic con el botón derecho en cualquier parte de la clase y elegir Refactor > Pull Up:



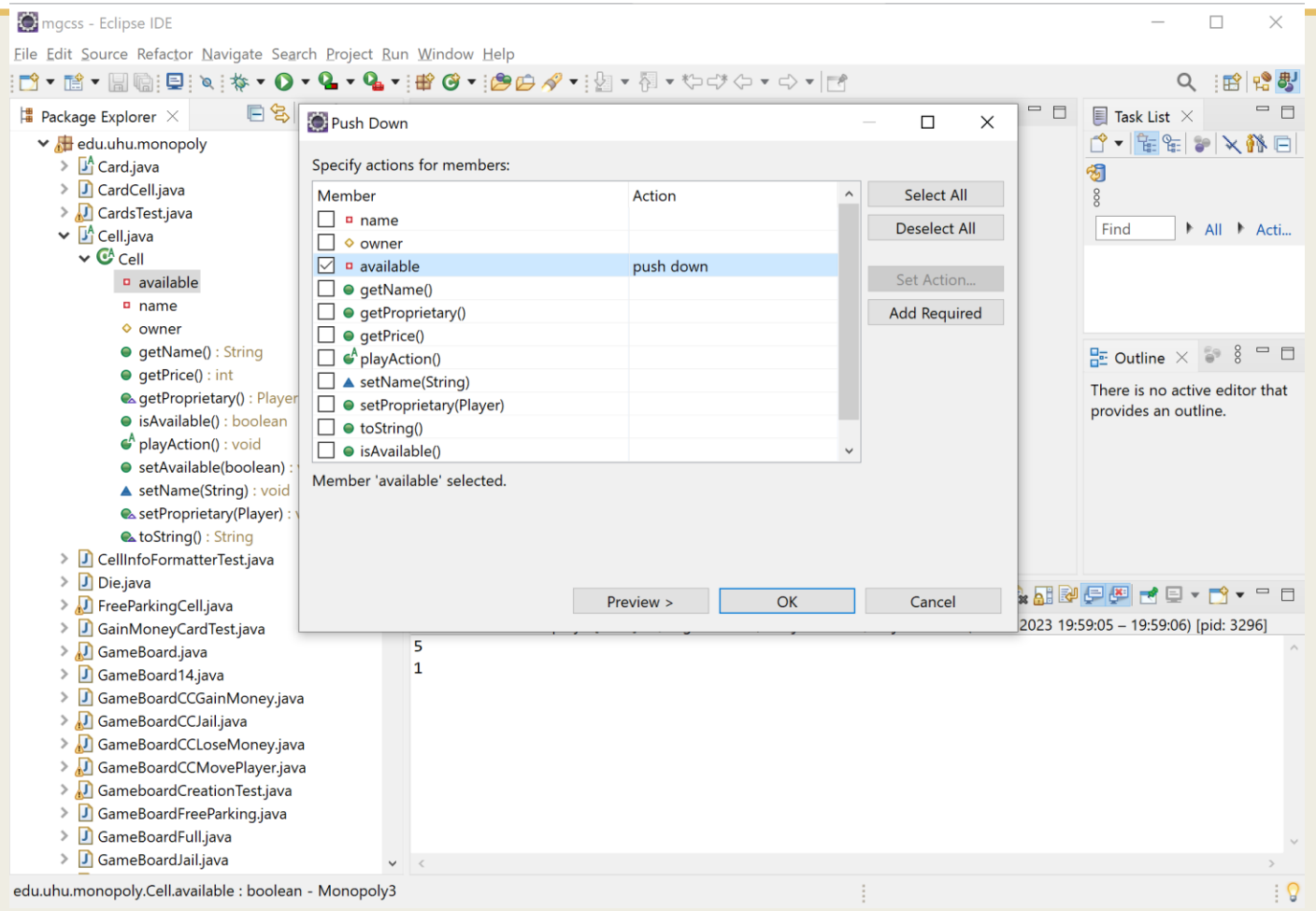


Refactorización 2: : Cambiar la jerarquía de clases: PushDown y PullUp.

Siguiendo en la clase Cell, podemos observar que tenemos un atributo *available*. Vamos a utilizar la opción *PushDown* para mover este atributo desde la superclase a todas sus subclases.

Ojo!!, pensar si los métodos *getter* y *setter* asociados a este atributo también tienen que ser reubicados. Hay que reflexionar detenidamente antes de realizar la refactorización.

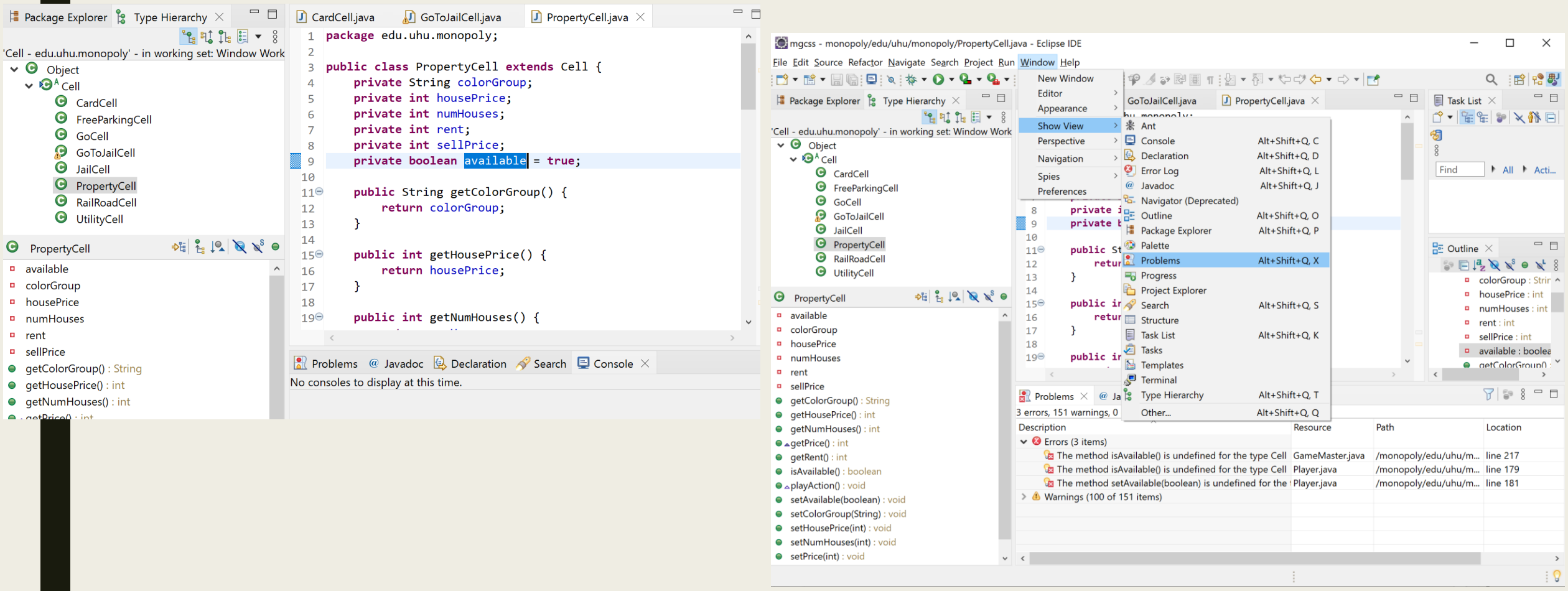
De nuevo, utilizamos la opción *Preview* para ver como quedarán las clases después de realizar los cambios.



Refactorización 2: : Cambiar la jerarquía de clases: PushDown y PullUp.

Después de realizar la refactorización, comprobar en algunas de las subclases (ver la jerarquía utilizando la opción del menú Navigate>>Open Type Hierarchy) que efectivamente tanto el atributo como sus métodos (getter y setter) han sido reubicados.

Sin embargo, el mover bajar este atributo, nos ha ocasionado varios errores en el código. Si seleccionamos la opción del menú Window>>Show view>>Problems, en la parte inferior de la siguiente pantalla podemos ver los errores existentes.

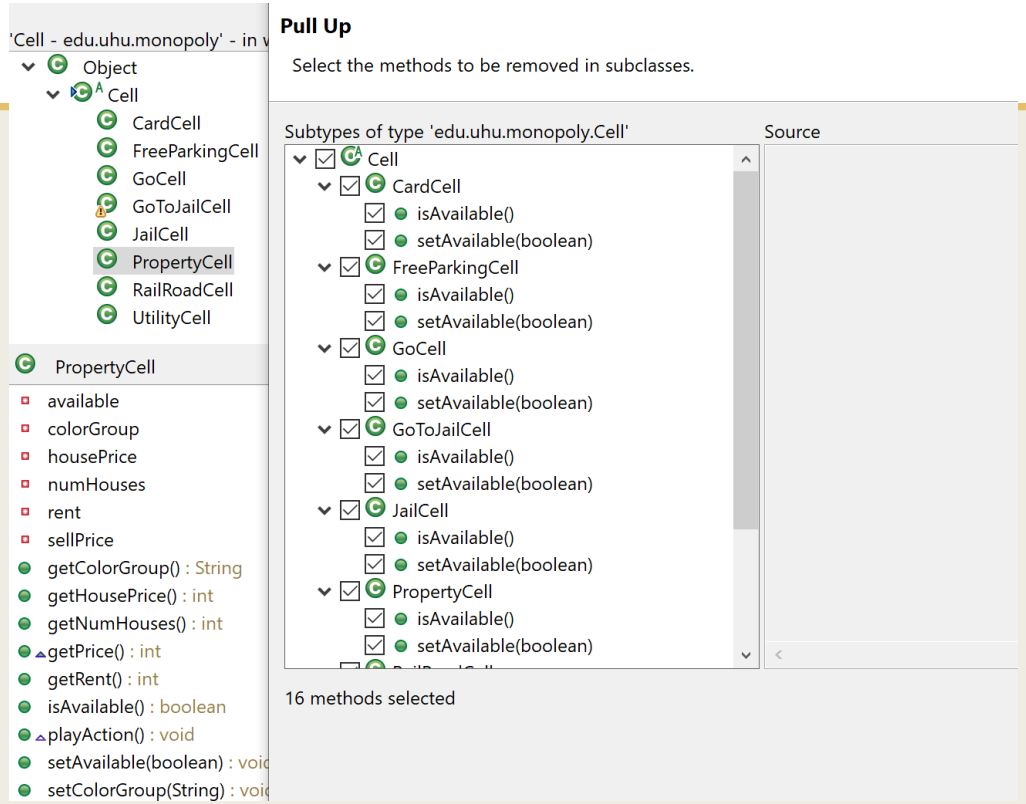


Refactorización 2: : Cambiar la jerarquía de clases: PushDown y PullUp.

Básicamente estos errores nos están indicando, que estamos haciendo referencia a los métodos *isAvailable()* y *setAvailable()* desde un objeto de tipo Cell, y que no están definidos en esta clase.

Vamos a comprobar el potencial de eclipse realizando la refactorización inversa, es decir, Pull Up. Elige una de las subclases de Cell, por ejemplo CardCell. Selecciona el atributo available y con el botón derecho selecciona la opción Refactor>>Pull Up. A continuación selecciona los métodos asociados que también quieres subir a la superclase(es decir, Cell).

Como puedes observar eclipse reconoce el problema de que hay más clases con método idéntico, y nos permite en todas las subclases de Cell subir (Pull Up) de manera simultanea, aquellos métodos y atributos con idéntico nombre del que inicialmente habíamos seleccionado. Para seleccionar todos los métodos, pulsar el selector de Cell 2 veces.



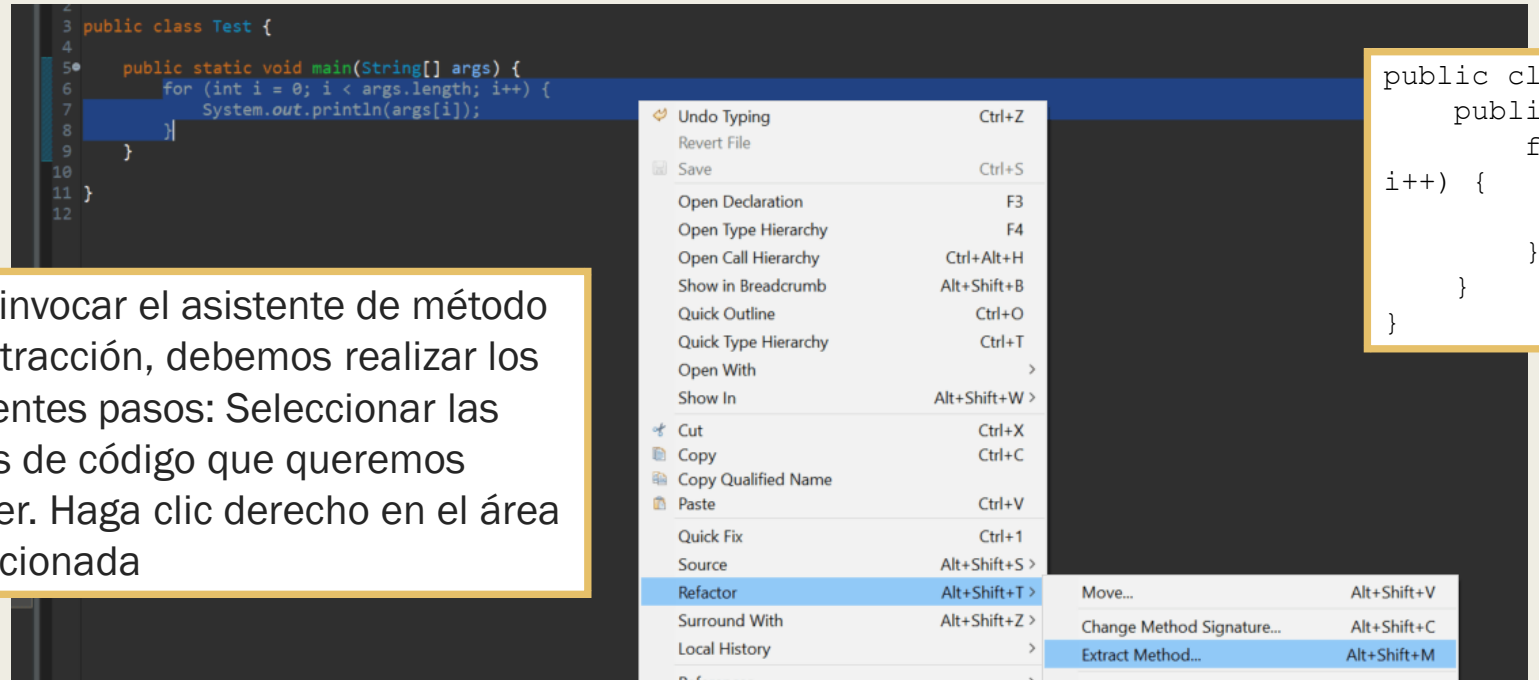
Recuerda publicar una versión después de cada Refactorización.

Realizar este tipo de tareas sin herramientas automáticas puede llegar a ser farragosa, larga y proclive a errores.

3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Extracción: Extraer código significa tomar un fragmento de código y moverlo/aislar para poder reusarlo. Por ejemplo, podemos extraer código en una clase, superclase o interfaz diferente. Incluso podríamos extraer código a una variable o método en la misma clase. Eclipse proporciona una variedad de formas de lograr extracciones, que demostraremos en las siguientes secciones.

Extraer un método: A veces, es posible que deseemos extraer una determinada pieza de código dentro de nuestro método a un método diferente para mantener nuestro código limpio y fácil de mantener. Digamos, por ejemplo, que tenemos un bucle for incrustado en nuestro método:

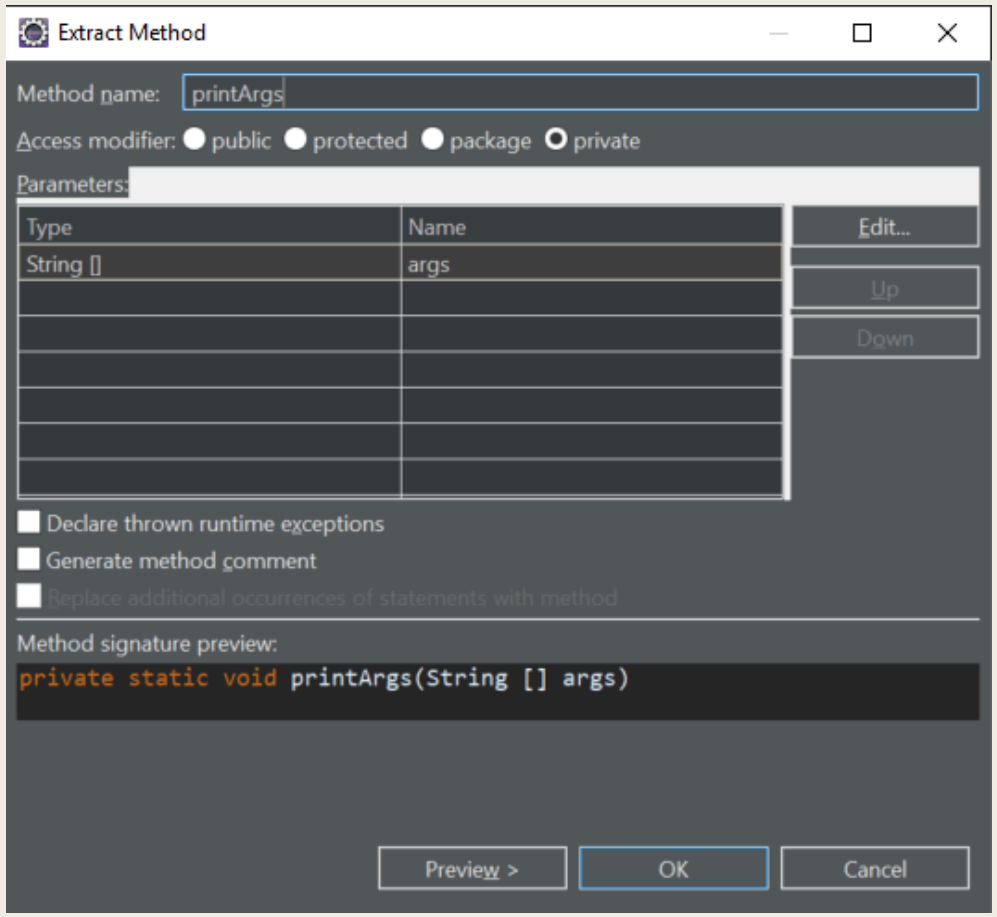


Para invocar el asistente de método de extracción, debemos realizar los siguientes pasos: Seleccionar las líneas de código que queremos extraer. Haga clic derecho en el área seleccionada

```
public class Test {
    public static void main(String[] args) {
        for (int i = 0; i < args.length;
i++) {
            System.out.println(args[i]);
        }
    }
}
```

3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Extraer un método: A veces, es posible que deseemos extraer una determinada pieza de código dentro de nuestro método a un método diferente para mantener nuestro código limpio y fácil de mantener. Digamos, por ejemplo, que tenemos un bucle for incrustado en nuestro método:




```
public class Test {
    public static void main(String[] args) {
        for (int i = 0; i < args.length;
i++) {
            System.out.println(args[i]);
        }
    }
}
```

Esto refactorizará nuestro código a:

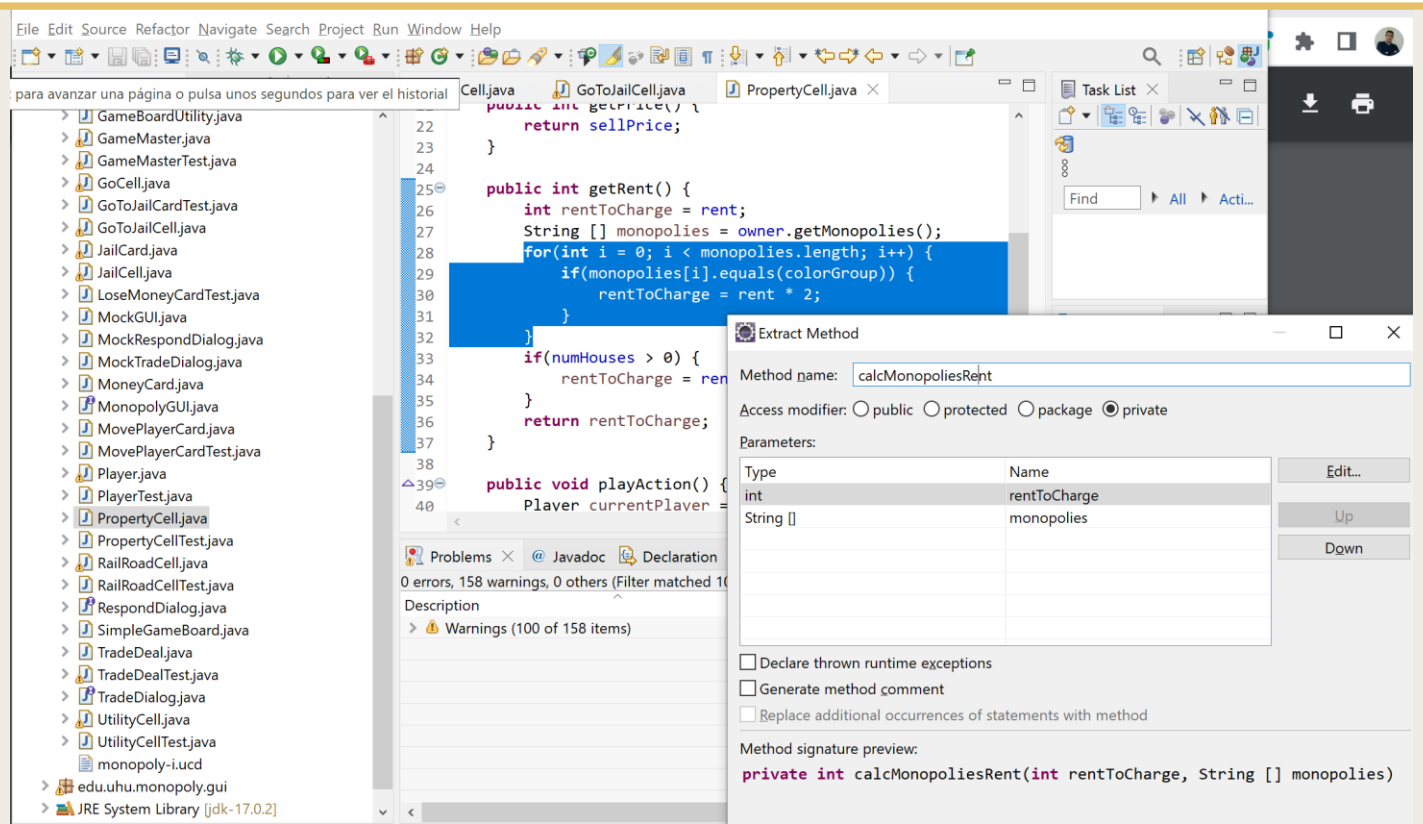
```
public class Test {

    public static void main(String[] args) {
        printArgs(args);
    }

    private static void printArgs(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }
}
```

Refactorización 3: Extracting a Method from Code

Una de las refactorizaciones más utilizadas es coger un fragmento de código y transformarlo en un método, de manera que pueda ser invocado desde varios lugares.

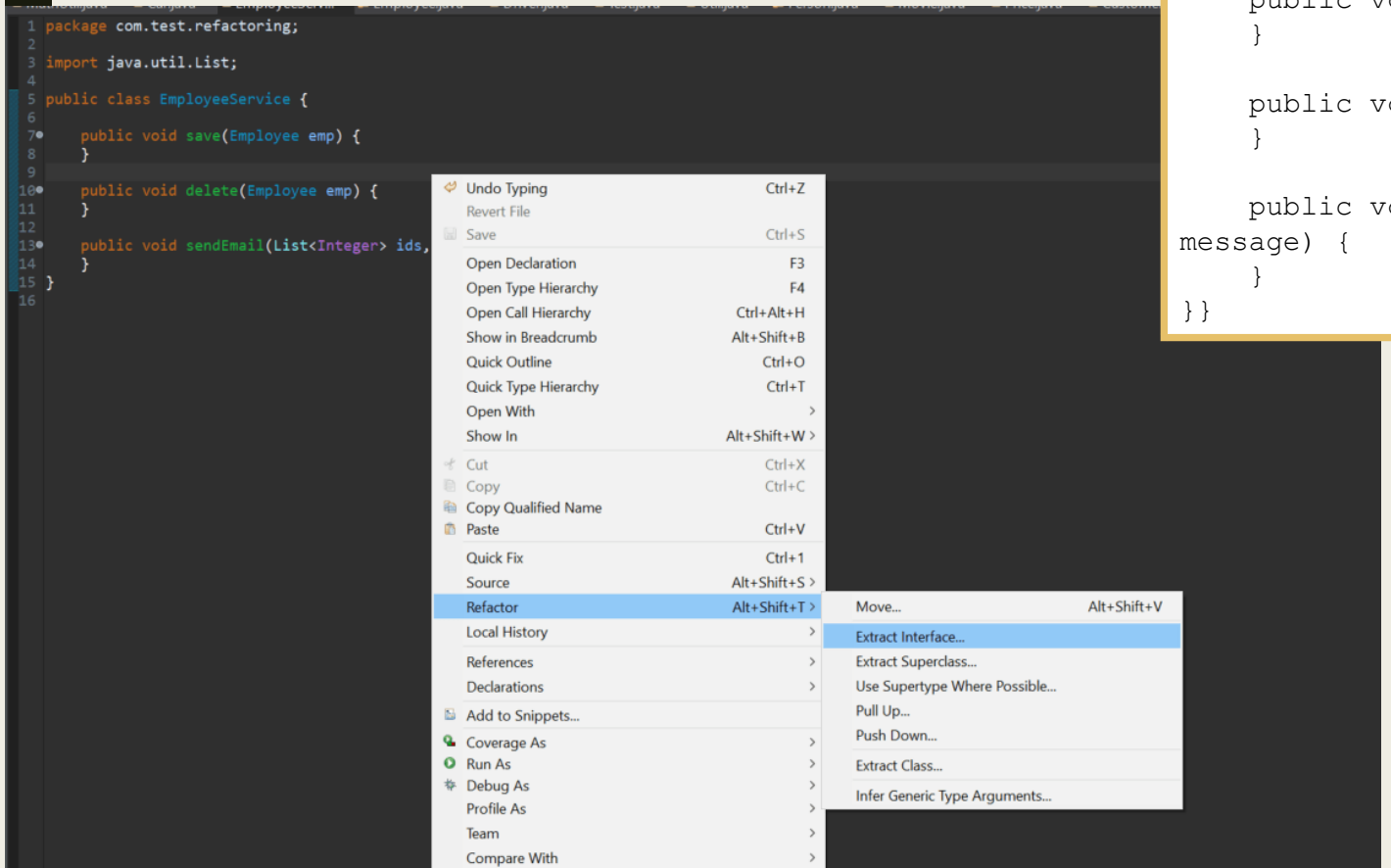
En la clase PropertyCell tenemos el método getRent(). Vamos a coger el primer bucle for y lo vamos agrupar en un nuevo método calcMonopoliesRent().

Selecciona todo el bucle y a continuación con el botón derecho selecciona la opción Refactor>>Extract Method. En la pantalla que aparece a continuación añade el nombre del método y pulsa el botón Preview para estudiar cómo se va a realizar la refactorización. Reflexiona si realmente has entendido cuáles son los parámetros de entrada y de salida del método generado, y porqué en la pantalla de refactorización aparecen esos parámetros.



3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Extraer una interface: Imagina que tenemos la siguiente clase:

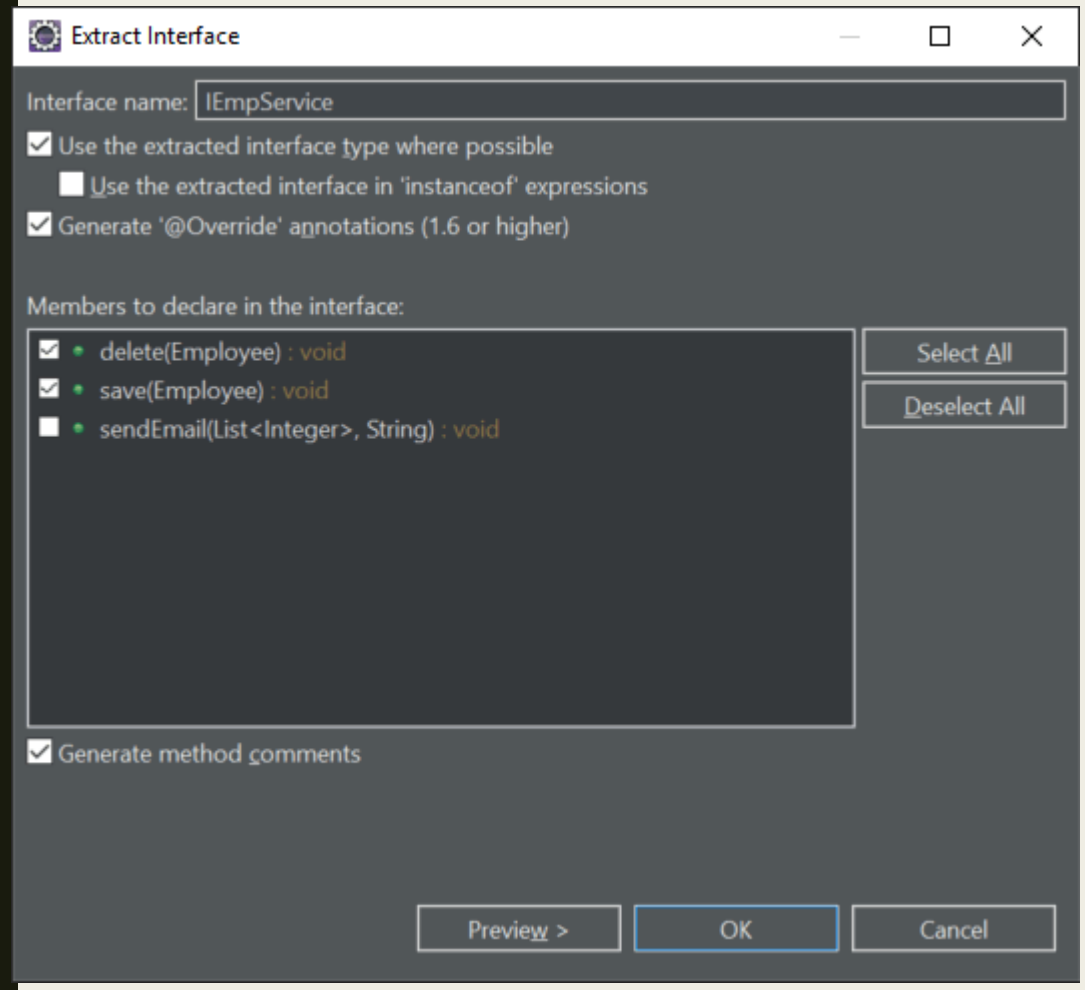



```
public class EmployeeService {

    public void save(Employee emp) {
    }

    public void delete(Employee emp) {
    }

    public void sendEmail(List<Integer> ids, String
message) {
    }
}}
```

3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Extraer una interface: Imagina que tenemos la siguiente clase:

Podemos ingresar el nombre de la interfaz y decidir qué miembros declarar en la interfaz.




```
public class EmployeeService {

    public void save(Employee emp) {
    }

    public void delete(Employee emp) {
    }

    public void sendEmail(List<Integer> ids, String
message) {
    }
}}
```

3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Extraer una interface: Imagina que tenemos la siguiente clase:

Podemos ingresar el nombre de la interfaz y decidir qué miembros declarar en la interfaz.

```
public class EmployeeService {

    public void save(Employee emp) {
    }

    public void delete(Employee emp) {
    }

    public void sendEmail(List<Integer> ids, String
message) {
    }
}}
```

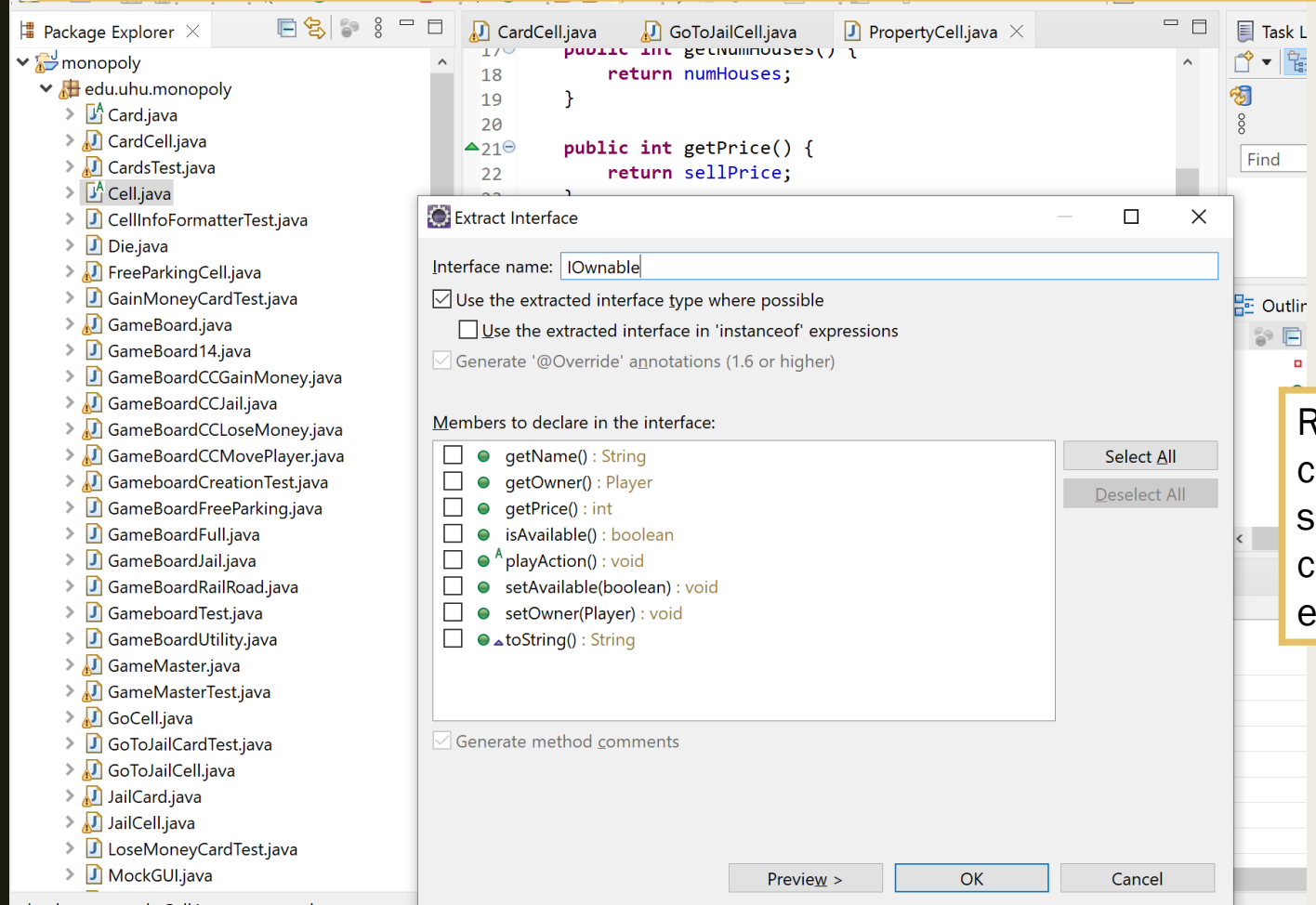
Como resultado de esta refactorización, tendremos una interfaz IEmpService y nuestra clase EmployeeService también cambiará:

```
public class EmployeeService implements IEmpService {

    @Override
    public void save(Employee emp) {
    }

    @Override
    public void delete(Employee emp) {
    }

    public void sendEmail(List<Integer> ids, String
message) {
    }
}
```

Refactorización 4: Extracting an Interface

La clase Abstracta Cell tiene un campo *owner* porque los jugadores pueden ser propietarios de parcelas en el tablero del Monopoly. Qué sucede si los jugadores pudiesen ser propietarios de otras cosas? Supongamos que esto tuviese sentido y decidiésemos hacer la noción de propiedad una interfaz.

Elige la clase Cell y selecciona la operación de refactorización *Extract Interface*. Asígnale el nombre de *IOwnable* a la nueva interfaz. Qué métodos de la clase Cell deberían moverse a esta interfaz? **Responde en la publicación de la versión.**

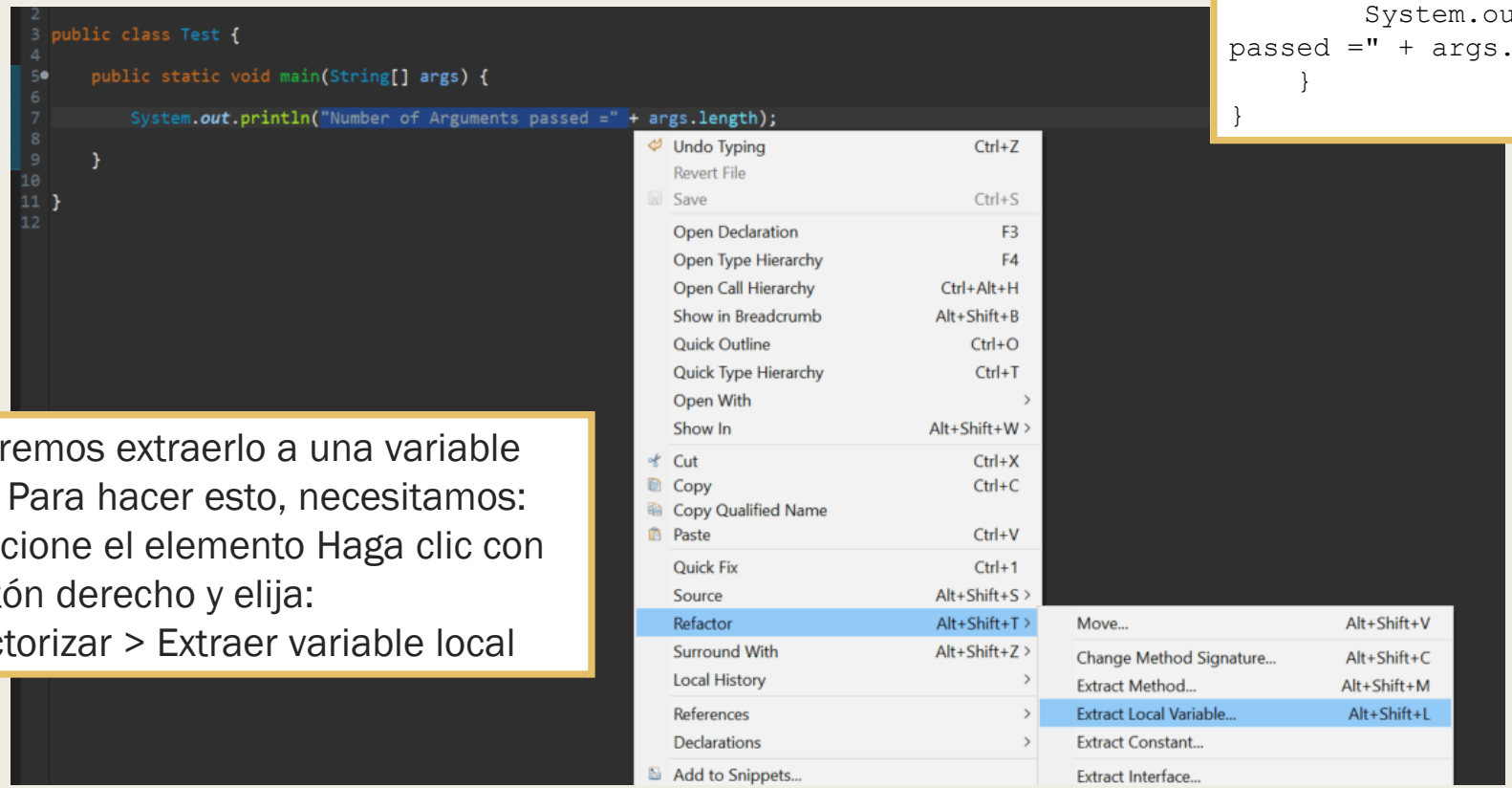


Realiza la refactorización y reflexiona acerca de cómo ha cambiado el código. Qué ficheros han sido modificados? Qué ficheros nuevos se han creado? La opción Preview te puede ser útil para entender la refactorización.

3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Extraer variables locales: Podemos extraer ciertos elementos como variables locales para que nuestro código sea más legible. Esto es útil cuando tenemos un literal de tipo String:

```
public class Test {

    public static void main(String[] args) {
        System.out.println("Number of Arguments
passed =" + args.length);
    }
}
```



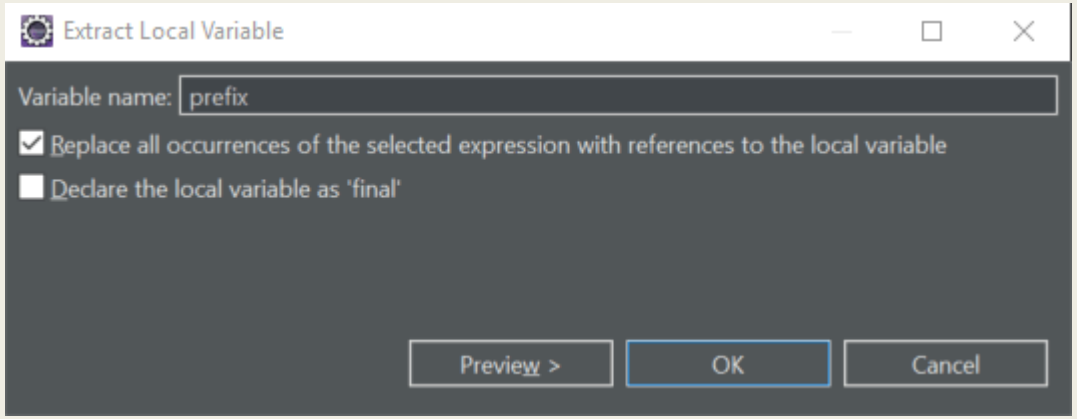
y queremos extraerlo a una variable local. Para hacer esto, necesitamos: Seleccione el elemento Haga clic con el botón derecho y elija: Refactorizar > Extraer variable local

3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Extraer variables locales: Podemos extraer ciertos elementos como variables locales para que nuestro código sea más legible. Esto es útil cuando tenemos un literal de tipo String:

y queremos extraerlo a una variable local. Para hacer esto, necesitamos: Seleccione el elemento Haga clic con el botón derecho y elija: Refactorizar > Extraer variable local

```
public class Test {

    public static void main(String[] args) {
        System.out.println("Number of Arguments
passed =" + args.length);
    }
}
```




```
public class Test {

    public static void main(String[] args) {
        final String prefix = "Number of Arguments passed =";
        System.out.println(prefix + args.length);
    }
}
```

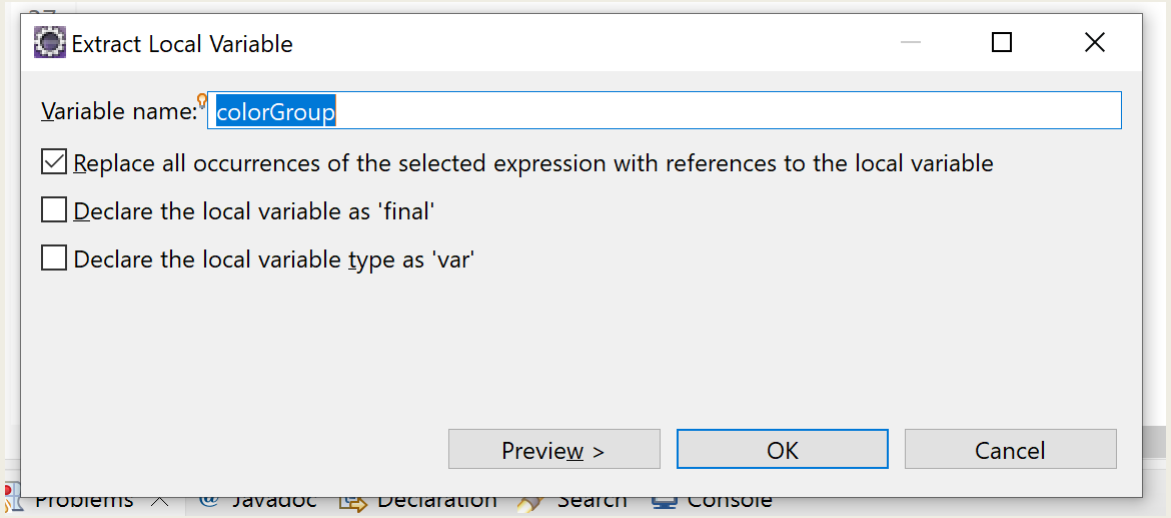
Refactorización 5: Creating a Local Variable from Repeated Code.

La refactorización Extraer Variable Local permite tomar una expresión que podría repetirse en el código y crear una variable local para esa expresión. Veamos un ejemplo.

Localiza el método GameBoard.addCell(PropertyCell). Como puedes observar la expresión cell.getColorGroupse utiliza en dos instrucciones.

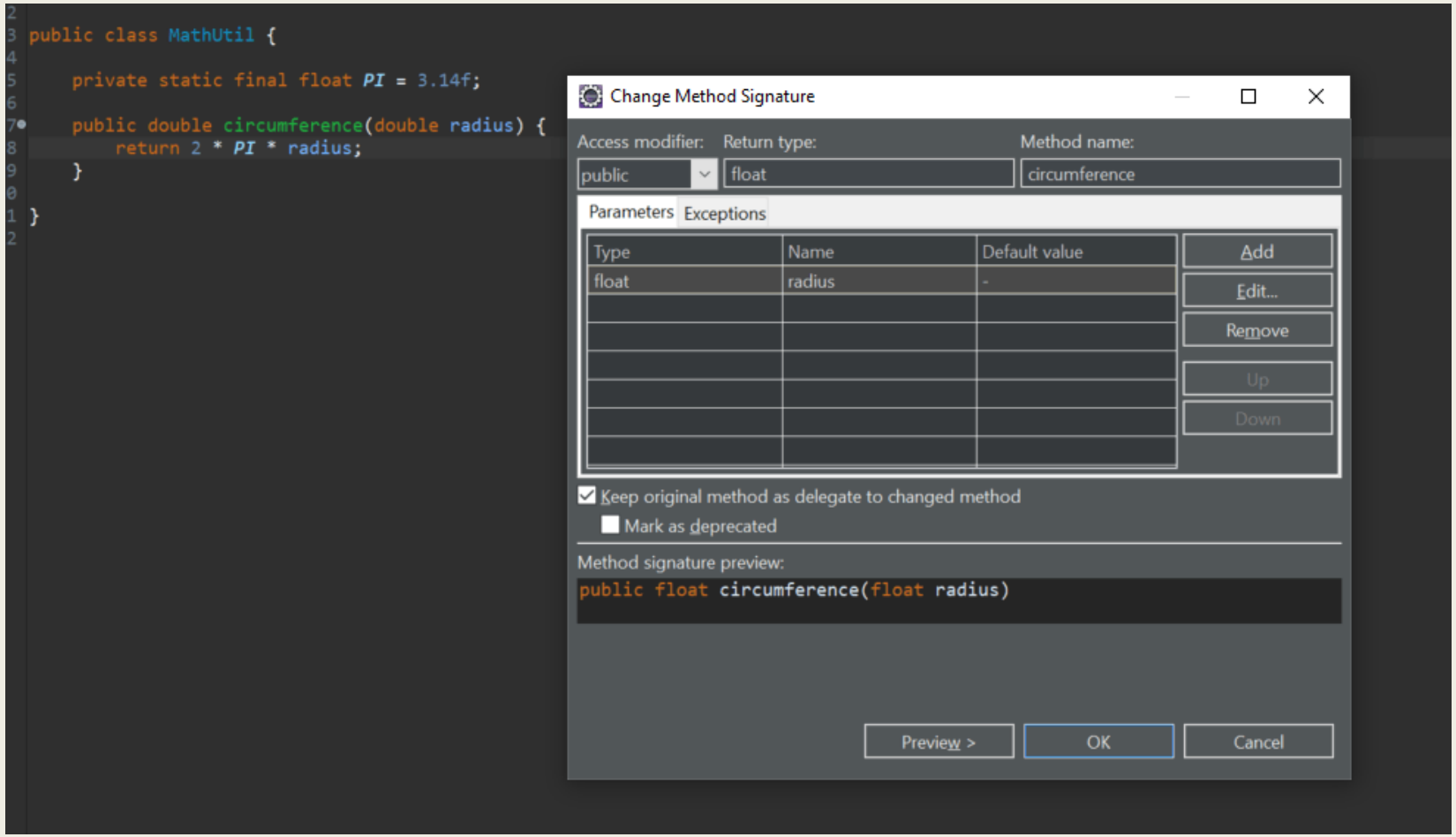
```
32  public void addCell(PropertyCell cell) {
33      int propertyNumber = getPropertyNumberForColor(cell.getColorGroup());
34      colorGroups.put(cell.getColorGroup(), new Integer(propertyNumber + 1));
35      cells.add(cell);
36  }
```

Marca la expresión a refactorizar y con el botón derecho selecciona la opción Refactor>>Extract Local Variable. Como puedes observar eclipse te propone nombres para la variable local. **Recuerda publicar una versión después de cada Refactorización.**



3.3 RECONSTRUCCIÓN (REFACTORIZACIÓN) DE PROGRAMAS.

Cambiar la declaración de un método.



Refactorización 6: Changing a Method's Signature.

La última refactorización que vamos a estudiar en este laboratorio es la más complicada de utilizar: Cambiar la Signatura de un Método. Aunque la semántica de la operación es clara – cambiar los parámetros, visibilidad o el tipo del resultado, no es tan obvio gestionar los efectos de estos cambios en el propio método o el en código que invoca a este método.

Veamos un ejemplo utilizando el método GameBoard.getPropertyNumberForColor(String).

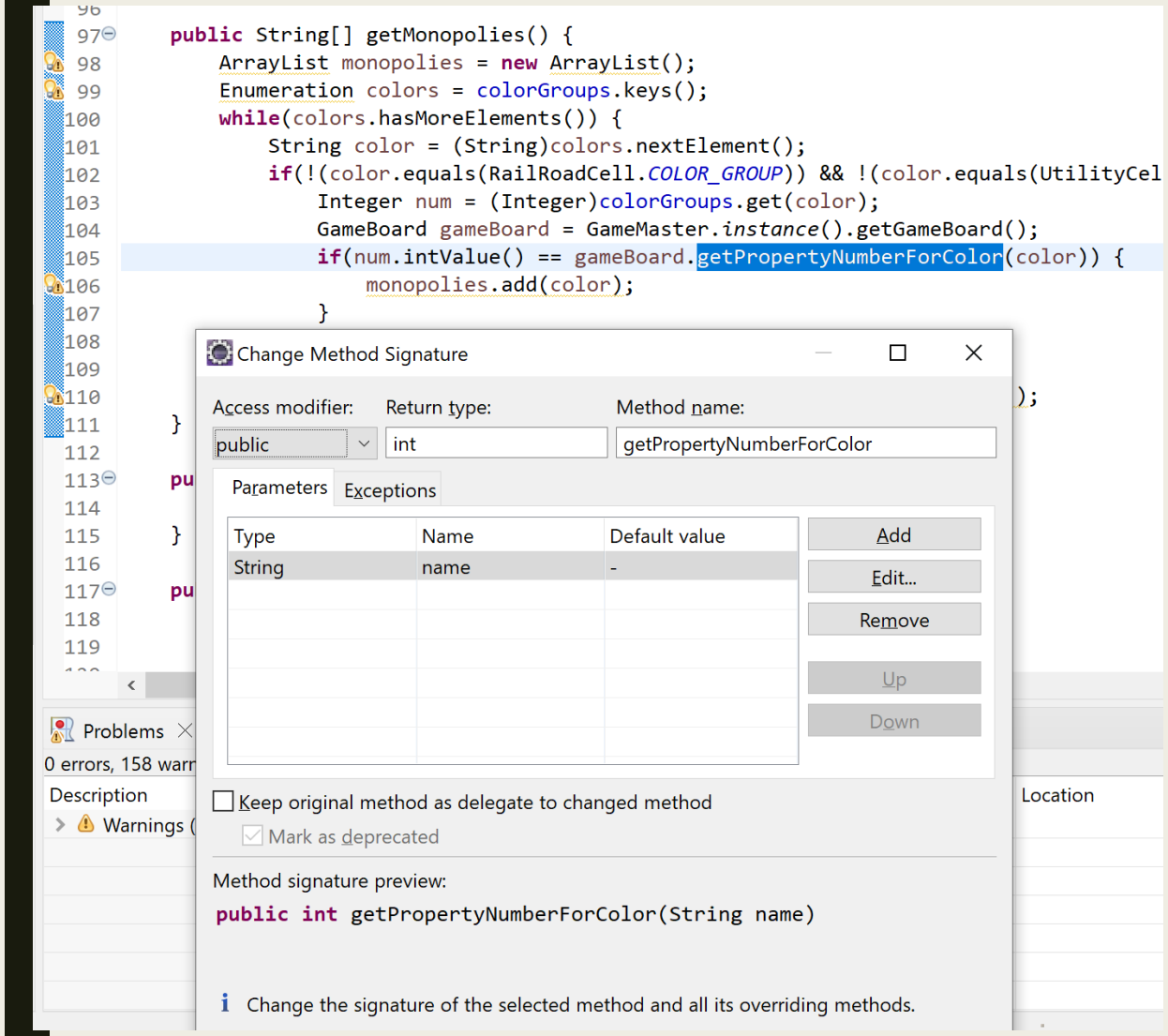
```
78      public int getPropertyNumberForColor(String name) {
79          Integer number = (Integer)colorGroups.get(name);
80          if(number != null) {
81              return number.intValue();
82          }
83          return 0;
84      }
```

El método getPropertyNumberForColor() en la clase de arriba es invocado por el método getMonopolies en la clase Player tal y como se muestra a continuación:

```
97      public String[] getMonopolies() {
98          ArrayList monopolies = new ArrayList();
99          Enumeration colors = colorGroups.keys();
100         while(colors.hasMoreElements()) {
101             String color = (String)colors.nextElement();
102             if(!(color.equals(RailRoadCell.COLOR_GROUP)) && !(color.equals(UtilityCell.
103                 Integer num = (Integer)colorGroups.get(color);
104                 GameBoard gameBoard = GameMaster.instance().getGameBoard();
105                 if(num.intValue() == gameBoard.getPropertyNumberForColor(color)) {
106                     monopolies.add(color);
107                 }
108             }
109         }
110         return (String[])monopolies.toArray(new String[monopolies.size()]);
111     }
```

Refactorización 6: Changing a Method's Signature.

Marca el método getPropertyNumberForColor, con el botón derecho selecciona la opción Refactor > Change Method Signature y aparecerá la siguiente ventana:



Los cambios que podemos realizar son los siguientes:

1. **Cambiar la visibilidad del método**. Si cambiamos la visibilidad del método getPropertyNumberForColor a protected o private, estaremos eliminando la posibilidad de que el método getMonopolies() en la clase Player acceda al método. Eclipse no informa de este posible error durante el proceso de refactorización. Por tanto es tarea del programador seleccionar la opción apropiada.
2. **Cambiar el tipo del resultado**. Vamos a modificar el tipo del resultado de int a float. Esta modificación no va a producir ningún error en el método getPropertyNumberForColor, ya que el int devuelto por el código del método se transforma automáticamente a float. Sin embargo puede tener consecuencias en los métodos que invocan al método refactorizado.

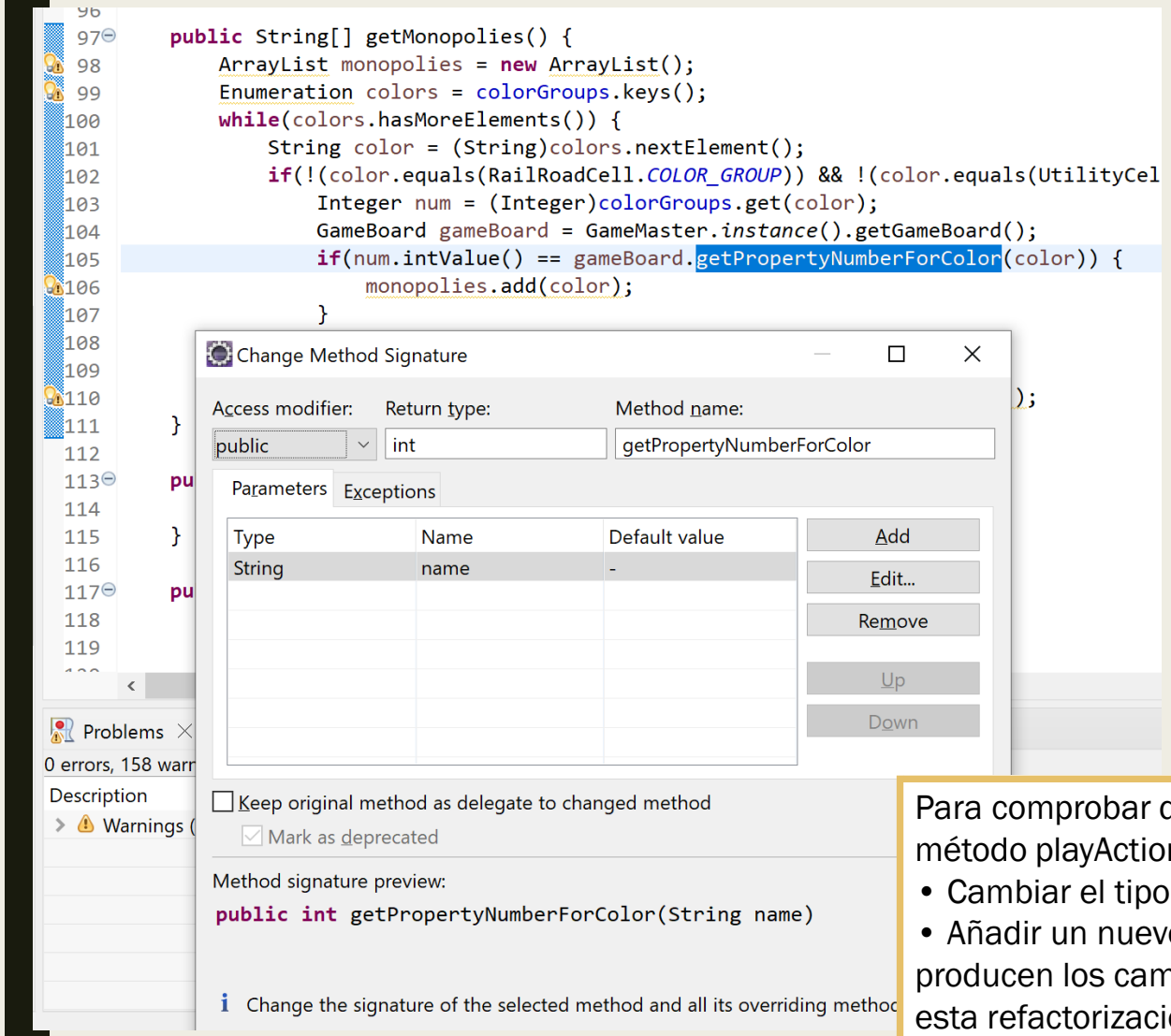
La figura nos está indicando que en el método addCell(), el tipo float devuelto por el método refactorizado getPropertyNumberForColor no puede asignarse a una variable de tipo int (propertyNumber). El problema se puede solucionar haciendo un cast del valor devuelto a int.

```
int propertyNumber=
(int)getPropertyNumberForColor(cell.
```

O bien modificando el tipo de la variable propertyNumber a float.

Refactorización 6: Changing a Method's Signature.

Marca el método getPropertyNumberForColor, con el botón derecho selecciona la opción Refactor > Change Method Signature y aparecerá la siguiente ventana:



3. **Cambiar el tipo de los parámetros**. Vamos a cambiar el tipo del parámetro del método getPropertyNumberForColor de String a int. En este caso podemos aplicar la mismas reflexiones que hemos realizado en el punto anterior. Esta refactorización producirá una alerta durante el proceso de previsualización. En este caso podemos ver cómo existen 2 llamadas a este método que no cumplen la nueva especificación. En el caso de que la refactorización genere errores, estos deberán ir resolviéndose de manera particularizada.
4. **Añadir un nuevo parámetro**. Imaginemos que queremos añadir un nuevo parámetro booleano visibility al método getPropertyNumberForColor. Marcamos el método y seleccionamos la opción Refactor > Change Method Signature.

Es importante tener en cuenta la tercera columna “default Value”. El valor que pongamos en este campo será el valor que tomara el nuevo parámetro en los métodos que invocan al método refactorizado

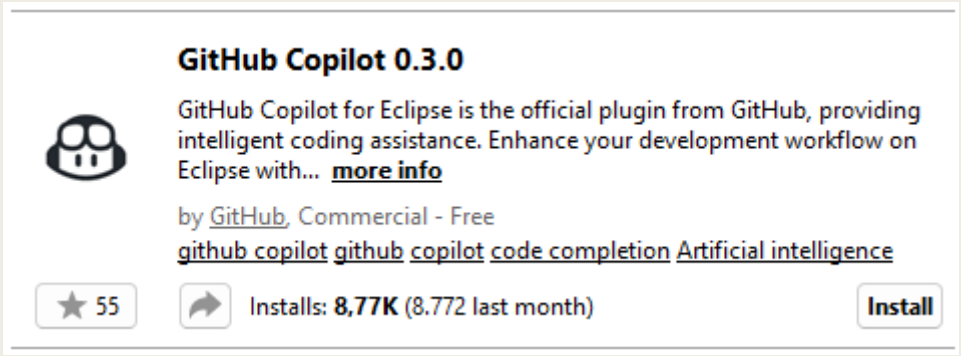
Para comprobar que has entendido esta refactorización, en la clase Cell, selecciona el método playAction() y utiliza la refactorización Change Method Signature para:

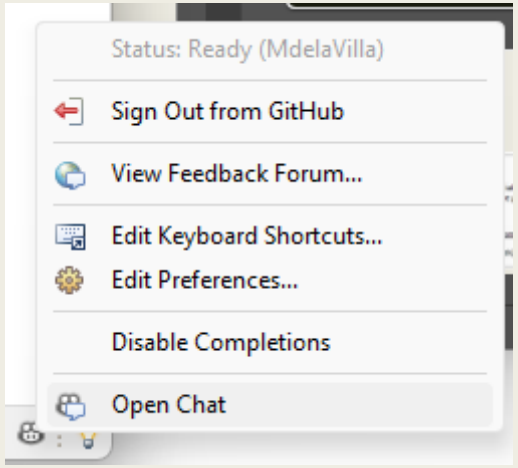
- Cambiar el tipo del resultado de void a boolean
- Añadir un nuevo parámetro msg de tipo String. Utiliza Preview para ver dónde y cómo se producen los cambios. ¿Por qué cambian cosas además de en la clase Cell? ¿Cómo afecta esta refactorización a la definición de otras clases además de Cell?

Responde al publicar la versión de la Refactorización

Bonus track

- Instalación de la extensión GitHub Copilot en tu entorno





- Getting code suggestions in your IDE with GitHub Copilot

```
public String getLabel() {
    return "Go to " + destination;
}
```