Introducción

En este laboratorio se presentan varios casos de refactorización y cómo son soportados en el entorno eclipse.

Objetivos

Los objetivos de este laboratorio 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.

¿Qué es Refactorizar?

Refactorizar es el proceso de modificar el código de un desarrollo para mejorar su estructura interna sin alterar la funcionalidad que ofrece el desarrollo externamente.
Para comenzar a Refactorizar es imprescindible que exista un desarrollo y que ese desarrollo tenga asociado **código de prueba** que nos permita saber en cualquier momento, pasando las pruebas automáticas, si el desarrollo sigue cumpliendo los requisitos que implementaba.
Si no existen estas pruebas será realmente complicado llevar a cabo refactorizaciones, dado que no podremos conocer si nuestras modificaciones han hecho que el desarrollo deje de funcionar.

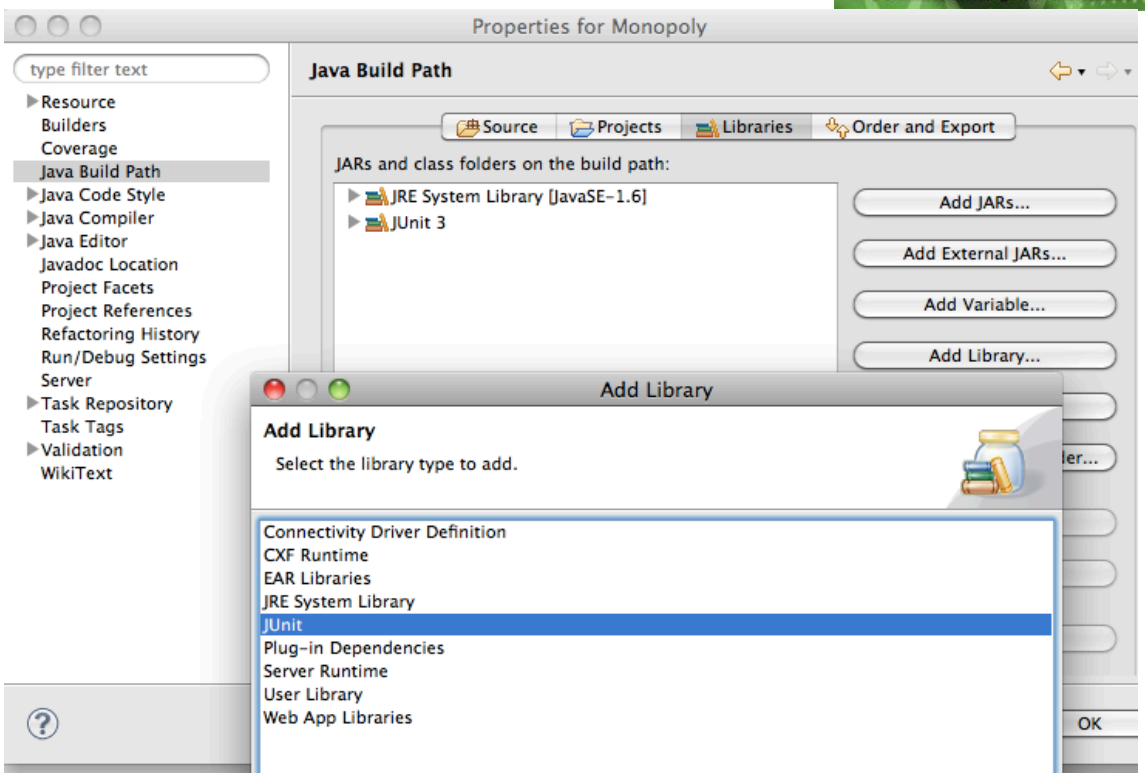
Para realizar las refactorizaciones en este laboratorio, vamos a utilizar la aplicación del Monopoly desarrollada en la Universidad del Estado de North Carolina. El objetivo de este laboratorio no consiste en estudiar cómo esta desarrollada esta aplicación, sino que nuestro objetivo consiste en realizar refactorizaciones sobre parte de su código.

Paso 1: Crear un proyecto y cargar las clases

Obtener una copia de los ficheros java de la aplicación de esta dirección (http://www.cs.virginia.edu/~horton/cs494/s05/slides/Monopoly3.zip).

A continuación, crear un proyecto en Eclipse e importar los ficheros previamente descargados.

Una vez cargadas las clases, debemos tener todas las clases bajo 2 paquetes. Sin embargo, algunas clases estarán marcadas por el indicador rojo de error. Los indicadores deben desaparecer después de añadir las clases de JUnit al Build-path del proyecto. Para ello, clicar con el botón derecho en la raíz del proyecto y seleccionar la opción Properties, donde os aparecerá la siguiente pantalla:



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.

Paso 2: Probar que funciona

Vamos a ver que la aplicación funciona correctamente, y que supera satisfactoriamente todos los casos de prueba.

Primero comprobaremos que se ejecutan correctamente 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 caso de prueba de 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. Antes de continuar, dejar el caso de prueba como al principio.

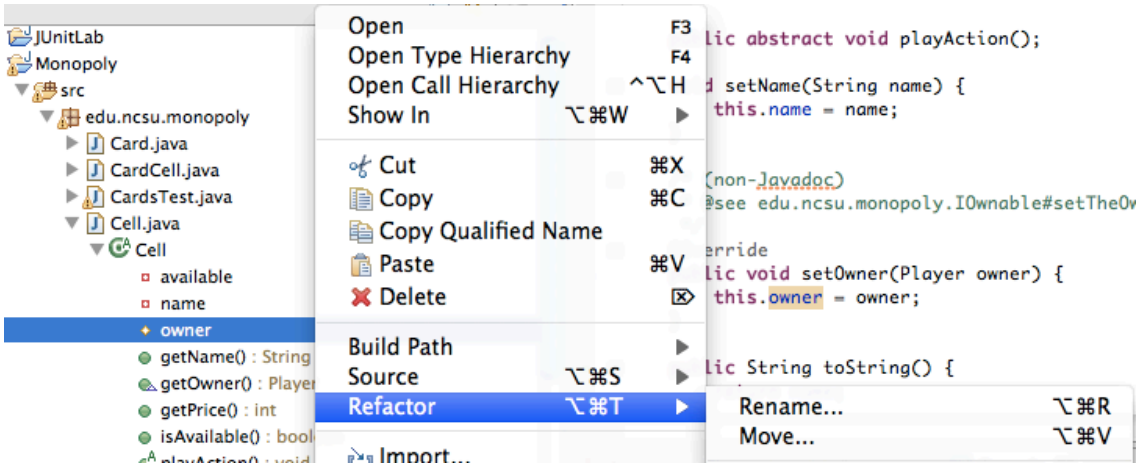
A continuación ejecutamos el fichero Main.java ubicado en paquete edu.ncsu.monopoly.gui para ver que la aplicación se ejecuta correctamente. No es el objetivo de este laboratorio ver cómo funciona la aplicación, por tanto, con ver que se ejecuta es suficiente.

A continuación vamos a realizar algunas refactorizaciones.

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. Seleccionamos el atributo y con el botón derecho seleccionamos la opción Refactor>>Rename tal y como aparece en la siguiente pantalla:



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.

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.

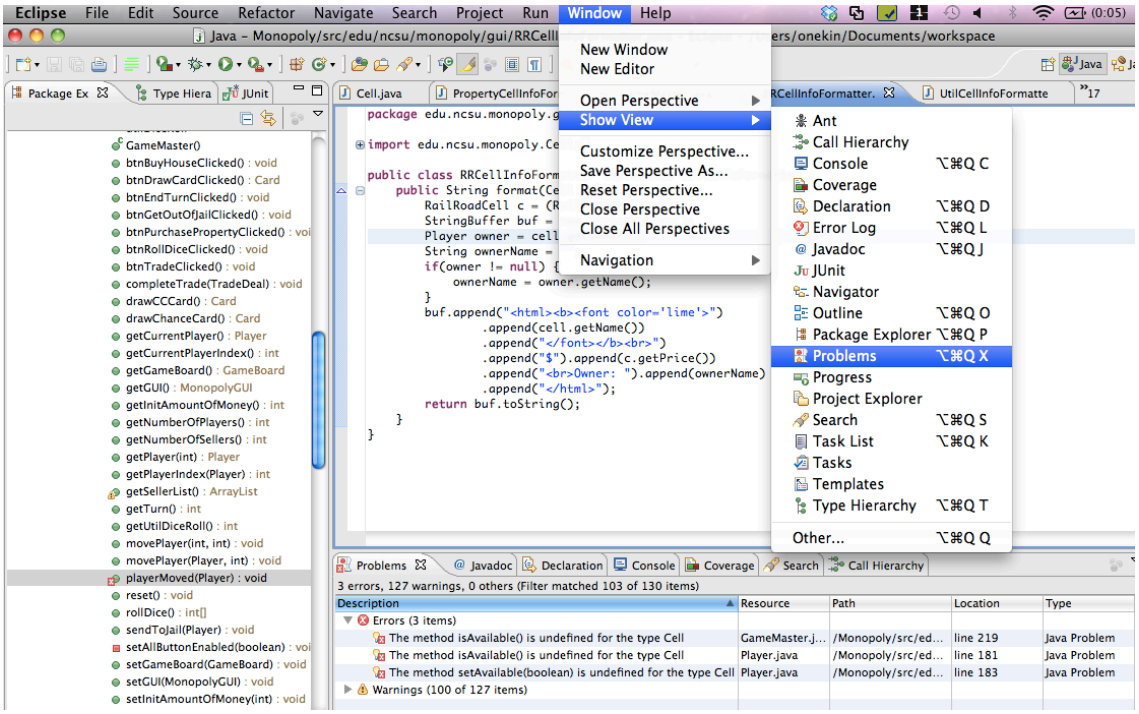
Para finalizar, ten en cuenta que siempre tienes la opción Undo disponible en el menú Refactor, para deshacer la refactorización realizada (puedes hacer undo varias veces para deshacer varias refactorizaciones).

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.

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.

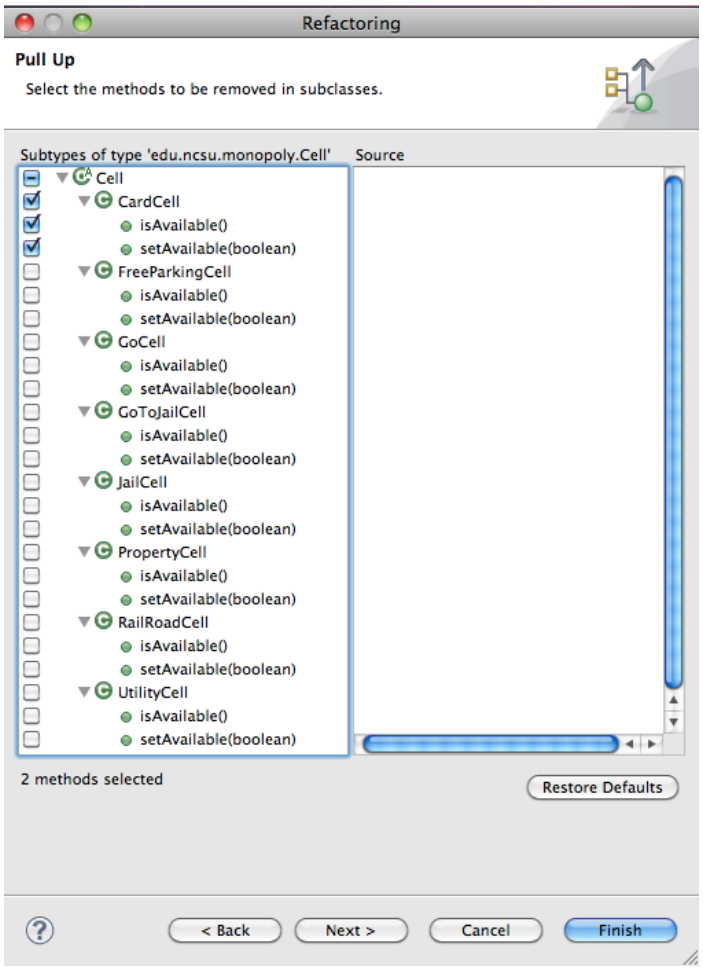
Sin embargo, el mover bajar este atributo, nos ha ocasionado varios errores en el código. Ver los indicadores rojos de error en las clases GameMaster.java y Player.java Si seleccionamos la opción del menú Window>>Show view>>Problems, en la parte inferior de la siguiente pantalla podemos ver los errores existentes:



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.

Por lo tanto, bajar este método no ha sido una buena idea (aunque nos haya servido para ver cómo se realiza) y vamos a dejar el atributo y sus métodos donde estaban (es decir, en la superclase). Podemos resolver este error eligiendo la opción Refactor -> Undo. Pero antes de realizar esto, 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). En este punto te podrías cuestionar si hay que realizar este proceso para cada una de las subclases en las cuales habías bajado el atributo available (es decir, FreeParkingCell, GoCell, GoToJailCell etc.…). Pulsa el botón Next y en la siguiente figura tienes la respuesta:



Como puedes observar eclipse reconoce el problema, 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 (una vez para deseleccionar todos, y la segunda para seleccionar todos). Finalmente pulsar el botón Finish, y comprobar que los errores han desaparecido, tanto los sintácticos, como los de las pruebas unitarias.

En resumen, los cambios realizados en una refactorización pueden afectar a varias clases en la jerarquía. En este ejercicio se ha mostrado cómo Eclipse facilita el proceso de refactorización de un manera sencilla. Realizar este tipo de tareas sin herramientas automáticas puede llegar a ser farragosa, larga y proclive a errores.

Refactorización 3: 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?

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.

Refactorización 4: 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. También puede ser utilizado para agrupar código similar en diferentes subclases y subirlo a la superclase.

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. Espera, no realices aún la refactorización.

Cancela la refactorización y selecciona el bucle for y la instrucción precedente en donde se declara e inicializa la variable monopolies. A continuación realiza el proceso de extraer el método de nuevo. Porqué ha cambiado la signatura del método?

Realiza una de las dos refactorizaciónes. Examina el código y comprueba que entiendes perfectamente el código generado.

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.

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

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.

Reflexión: Se puede realizar esta refactorización en todos los casos en los que se

repita una función en el código? Sigue siendo correcto[1] el programa?.

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. Si los cambios realizados pueden causar problemas en el método refactorizado – porque deja variables sin definir o error de tipos – las operaciones de refactorización nos lo notificarán. En este caso, tienes la opción de realizar la refactorización y posteriormente corregir los problemas o cancelarla. Si la rafactorización causa problemas en otros métodos, estos son ignorados y deberás corregirlos después de la refactorización. Veamos un ejemplo utilizando el método GameBoard.getPropertyNumberForColor(String).

```
public class GameBoard {
  public int getPropertyNumberForColor(String name) {
        Integer number = (Integer)colorGroups.get(name);
        if(number != null) {
          return number.intValue();
        }
        return 0;
  }
}
```

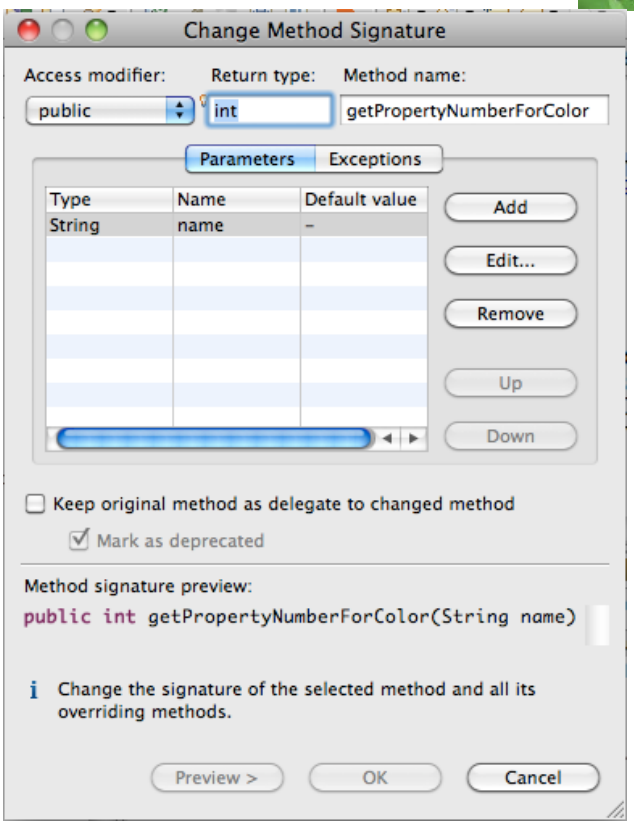
El método getPropertyNumberForColor() en la clase de arriba es invocado por el método getMonopolies en la clase Player tal y como se muestra a continuación:

```
public class Player {
  public String[] getMonopolies() {
     ……

  if(num.intValue() == gameBoard.getPropertyNumberForColor(color)) {
        monopolies.add(color);
  }
}
```

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

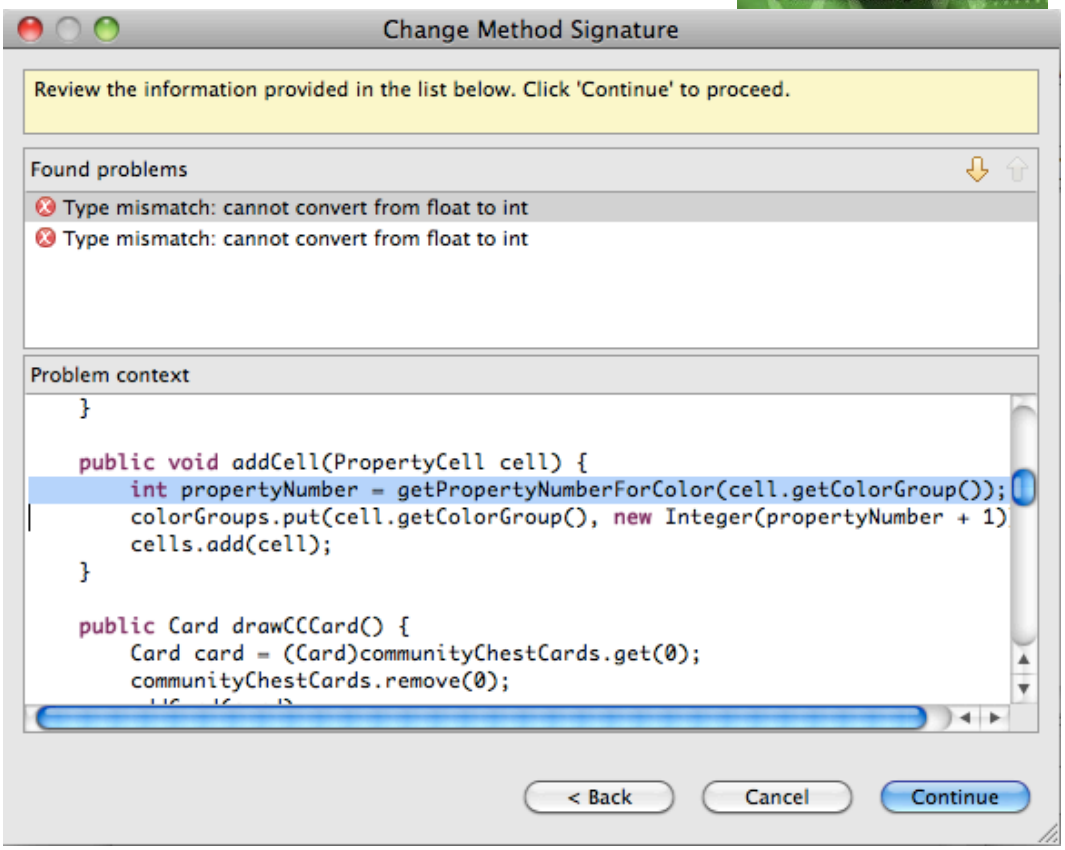
[1] Un programa es correcto si para cada entrada produce la salida esperada.



Los cambios que podemos realizar son los siguientes:

1. Cambiar la visibilidad del método. En este ejemplo, 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 tal y como se muestra en la siguiente figura.

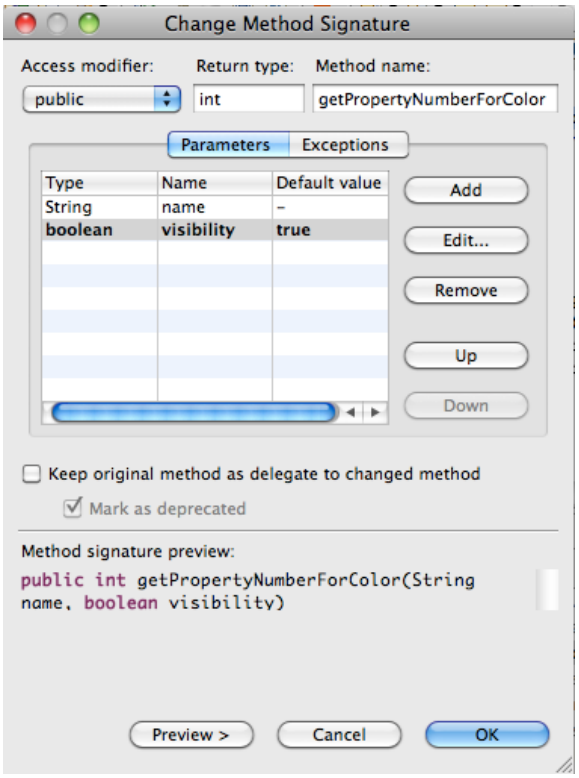


La figura nos está indicando que en el método addCell(), el tipo float devuelto por el método refactorizado getPropertyNumberForColor no puede asignarse a una variable de tipo int (propertyNumber). El problema se puede solucionar haciendo un cast del valor devuelto a int.

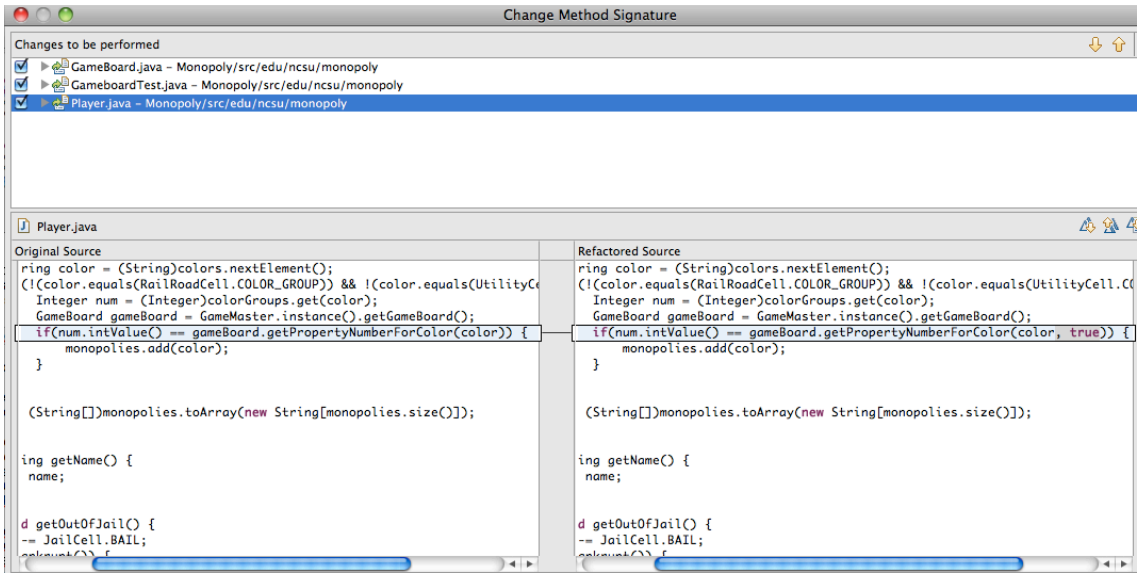
```
int propertyNumber= (int)getPropertyNumberForColor(cell.
```

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

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.** A continuación añadimos el nuevo parámetro tal y como aparece en la siguiente figura:



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. En la siguiente figura tenéis un ejemplo.



En la parte izquierda aparece el la invocación al método original. En la parte derecha, como el método tiene un nuevo parámetro, el valor con el que se le invocará será “true” tal y como se ha especificado previamente.

En este apartado hemos visto como se puede realizar una modificación en la signatura de un método. Esta refactorización aun no siendo problemática, sí que requiere de una planificación meditada para ser utilizada de forma satisfactoria.

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?

Resumen

Estos son los aspectos más relevantes a tener en cuenta en este laboratorio:

- La refactorización conlleva cambios estructurales en el código fuente.

- Aunque estos cambios sean simples (p.ej. renombrar un campo) podría conllevar a errores si no disponemos de herramientas que ayuden al proceso de refactorización. La refactorización sin herramientas no es útil.

- Eclipse soporta de manera eficiente la refactorización, ya que conoce las relaciones entre los métodos y clases de un proyecto y por lo tanto, permite realizar cambios estructurales que afectan a varias clases de un proyecto.

- La refactorización sin casos de prueba es una actividad arriesgada. Antes de realizar cualquier refactorización debemos definir los casos de prueba. La ejecución de los casos de prueba nos servirán para comprobar que la aplicación sigue comportándose correctamente.

Referencias:

[1] **Refactorización: camino hacia la calidad**
http://programacion.net/articulo/refactorizacion_camino_hacia_la_calidad_221

[2] **Refactoring for everyone**
http://www.ibm.com/developerworks/opensource/library/os-ecref/

[3] **Refactoring en eclipse**
http://www.slideshare.net/srcid/eclipse-refactoring