Refactorización en

Benito Cid Míllara
Diego Blanco Estévez
Tomás Guerra Cámara

Samuel Rodríguez Cid
Ismael González Suárez

Refactorización en eclipse

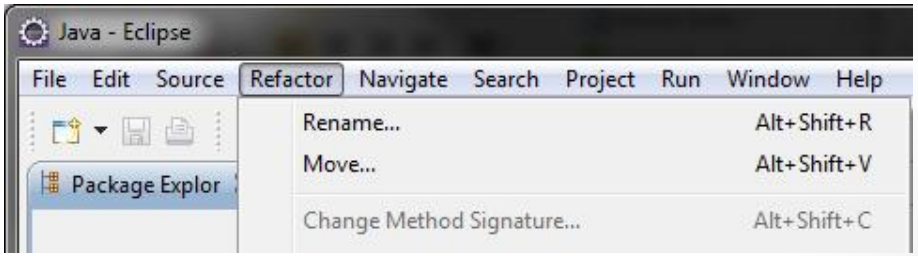
- Refactorización: técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo.
- Limpiar el código → mejorar la consistencia interna y la claridad.
- Mantenimiento del código
  - No arregla errores
  - No añade funcionalidad
- Se pueden realizar fácilmente cambios en el código.
- Probar trozos de código.
- Código limpio y altamente modularizado.

Opciones de refactorización

- Rename
- Move
- Extract Local Variable
- Extract Constant
- Convert Local Variable to Field
- Convert Anonymous Class to Nested
- Member Type to Top Level
- Extract Interface
- Extract Superclass
- Extract Method
- Inline
- Change Method Signature
- Infer Generic Type Arguments
- Migrate JAR File
- Refactoring Scripts

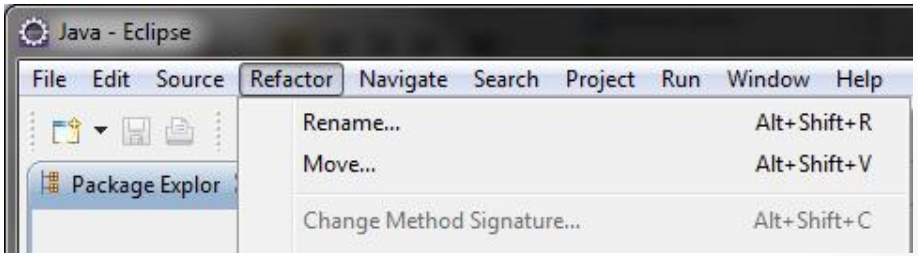
1. Rename

- Opción más utilizada.
- Cambia el nombre de variables, clases, métodos, paquetes, directorios y casi cualquier identificador Java.
- Tras la refactorización, se modifican las referencias a ese identificador.
- Refactor > Rename...
- Alt + Shift + R



2. Move

- Mover una clase de un paquete a otro.
  - Se mueve el archivo *.java* a la carpeta.
  - Se cambian todas las referencias.
- Arrastrar y soltar una clase a un nuevo paquete.
  - Refactorización automática.
- Refactor > Move...
- Alt + Shift + V



3. Extract Local Variable

- Asignar expresión a variable local.
- Tras la refactorización, cualquier referencia a la expresión en el ámbito local se sustituye por la variable.
- La misma expresión en otro método no se modifica.

```
public class ExtractLocalVariable {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

```
public class ExtractLocalVariable {

    public static void main(String[] args) {
        String string = "Hello World!";
        System.out.println(string);
    }
}
```

4. Extract Constant

- Convierte un número o cadena literal en una constante.
- Tras la refactorización, todos los usos del literal se sustituyen por esa constante.
- Objetivo: Modificar el valor del literal en un único lugar.

```
public class ExtractConstant {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

```
public class ExtractConstant {

    private static final String HELLO_WORLD = "Hello World!";

    public static void main(String[] args) {
        System.out.println(HELLO_WORLD);
    }
}
```

5. Convert Local Variable to Field

- Convierte una variable local en un atributo privado de la clase.
- Tras la refactorización, todos los usos de la variable local se sustituyen por ese atributo.

```
public class ConvertLocalVariable {

    public static void main(String[] args) {
        String msg = "Hello World!";
        System.out.println(msg);
    }
}
```

```
public class ConvertLocalVariable {

    private static String msg;

    public static void main(String[] args) {
        msg = "Hello World!";
        System.out.println(msg);
    }
}
```

6. Convert Anonymous Class to Nested

- Clase anónima
  - Clase sin nombre de la que sólo se crea un único objeto.
  - No se pueden definir constructores.
- Se utilizan con frecuencia para definir clases y objetos que gestionen los eventos de los distintos componentes de la interfaz de usuario.
- Maneras de definir una clase anónima:
  - Palabra new seguida de la definición de la clase anónima, entre llaves {...}.
  - Palabra new seguida del nombre de la clase de la que hereda (sin extends) y la definición de la clase entre llaves {...}.
  - Palabra new seguida del nombre de la interfaz (sin implements) y la definición de la clase anónima entre llaves {...}.

6. Convert Anonymous Class to Nested

- Convierte una clase anónima en una clase anidada de la clase contenedora.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;

public class ConvertAnonymousClass {
    public void createPool() {
        Object threadPool =  Executors.newFixedThreadPool(1, new ThreadFactory() {
            @Override
            public Thread newThread(Runnable r) {
                Thread t = new Thread(r);
                t.setName("Worker thread");
                t.setPriority(Thread.MIN_PRIORITY);
                t.setDaemon(true);
                return t;
            }
        });
    }
}
```

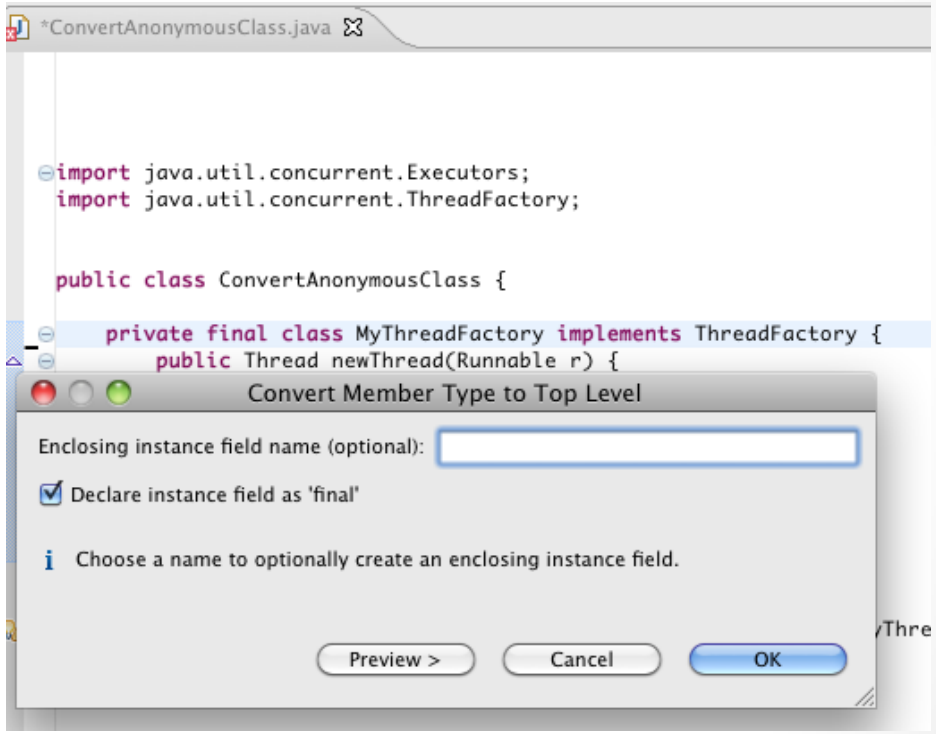
6. Convert Anonymous Class to Nested

- Asignar nombre de la clase, los modificadores de acceso (*public, private, protected*) y el tipo (*static, final*).
- Refactorización → código más limpio

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;

public class ConvertAnonymousClass {

    private final class MyThreadFactory implements ThreadFactory {
        @Override
        public Thread newThread(Runnable r) {
            Thread t = new Thread(r);
            t.setName("Worker thread");
            t.setPriority(Thread.MIN_PRIORITY);
            t.setDaemon(true);
            return t;
        }
    }

    public void createPool() {
        threadPool = Executors.newFixedThreadPool(1, new MyThreadFactory());
    }
}
```

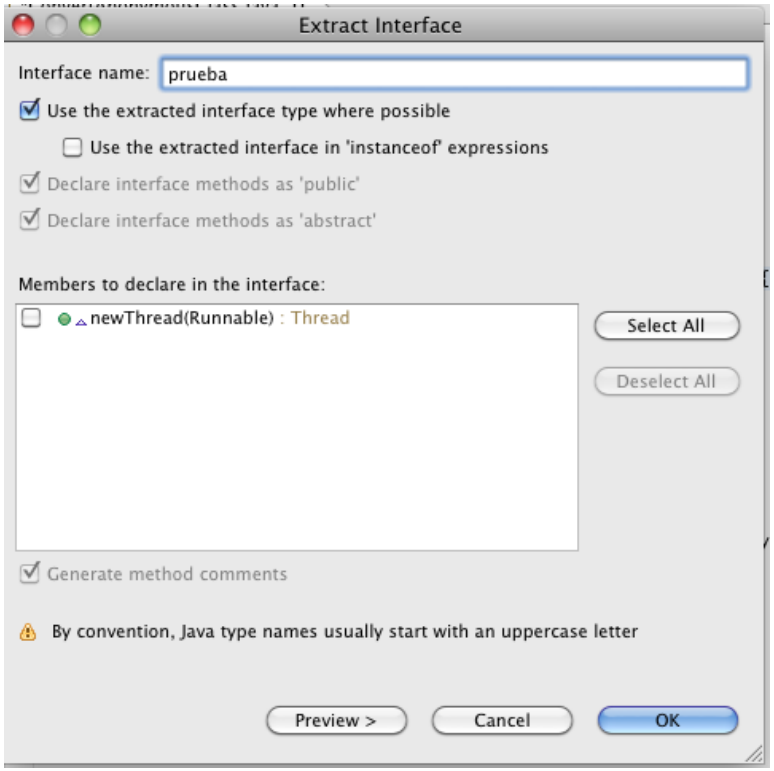
7. Member Type to Top Level

- Convierte una clase anidada en una clase de nivel superior con su propio archivo de java.
- Si la clase es estática, la refactorización es inmediata.
- Si no es estática nos pide un nombre para declarar el nombre de la clase que mantendrá la referencia con la clase inicial.



8. Extract Interface

- Se crea una interfaz con los métodos de una clase.
- Nos permite escoger que métodos de la clase, formarán parte de la interfaz.



9. Extract Superclass

- Extrae una superclase en lugar de una interfaz.
- Si la clase ya utilizaba una superclase, la recién creada pasará a ser su superclase.
- Se pueden seleccionar los métodos que formaran parte de la superclase.
- Diferencia: En la superclase, los métodos están actualmente allí, así que si hay referencias a campos de clase original, habrá fallos de compilación.

10. Extract Method

- Nos permite seleccionar un bloque de código y convertirlo en un método.
- Eclipse ajustará automáticamente los parámetros y el retorno de la función.
- Aplicaciones
  - Un método es muy extenso y lo queremos subdividir en diferentes métodos.
  - Un trozo de código se utiliza en diferentes métodos.

10. Extract Method

- Seleccionamos el bloque de código que queremos refactorizar.

```
@Override
public Object get(Object key) {
    TimedKey timedKey = new TimedKey(System.currentTimeMillis(), key);
    Object object = map.get(timedKey);
    if (object != null) {
        /**
          * if this was removed after the 'get' call by the worker thread
          * put it back in
          */
        map.put(timedKey, object);
        return object;
    }
    return null;
}
```

- Seleccionamos la opción “Extract Method” (alt+shift+M).
- Configuramos nuestro nuevo método indicando:
  - Nombre
  - Visibilidad (Public , Private, Protected)
  - Tipo de retorno

10. Extract Method

- Resultado:

```
@Override
public Object get(Object key) {
    TimedKey timedKey = new TimedKey(System.currentTimeMillis(), key);
    Object object = map.get(timedKey);
    return putIfNotNull(timedKey, object);
}

private Object putIfNotNull(TimedKey timedKey, Object object) {
    if (object != null) {
        /**
          * if this was removed after the 'get' call by the worker thread
          * put it back in
          */
        map.put(timedKey, object);
        return object;
    }
    return null;
}
```

11. Inline

- Nos permite ajustar una referencia a una variable o método con la línea en la que se utiliza y conseguir así una única línea de código.
- Cuando se utiliza, se sustituye la referencia a la variable o método con el valor asignado a la variable o la aplicación del método, respectivamente.
- Esto puede ser útil para limpiar nuestro código en diversas situaciones:
  - Cuando un método es llamado una sola vez por otro método, y tiene más sentido como un bloque de código.
  - Cuando una expresión se ve más limpia en una sola línea.

11. Inline

- Código a refactorizar con inline:

```
public Object put(Object key, Object value) {
    TimedKey timedKey = new TimedKey(System.currentTimeMillis(), key);
    return map.put(timedKey, value);
}
```

- Posicionamos el cursor en la referencia al método o variable.
- Seleccionamos la opción “Inline” (alt+shift+i).

- Resultado:

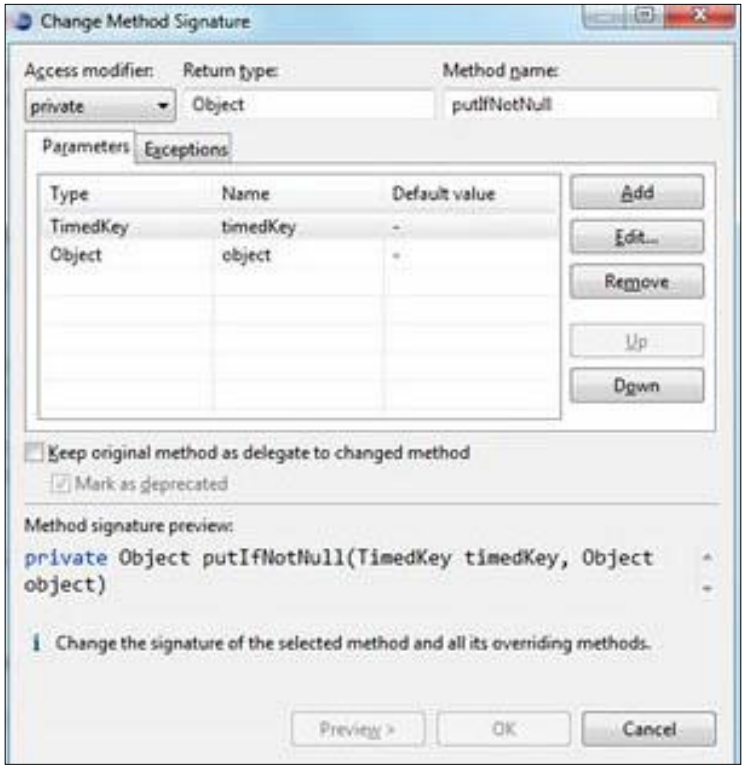
```
@Override
public Object put(Object key, Object value) {
    return map.put(new TimedKey(System.currentTimeMillis(), key), value);
}
```

12. Change Method Signature

- Esta función nos permitirá cambiar la firma de un método. La firma de un método es única y está compuesta por:
  - Nombre
  - Parámetros
- De forma automática se actualizarán todas las dependencias y llamadas a este en nuestro proyecto.
- Nota: Si añadimos parámetros o cambiamos el tipo de retorno a nuestro método lógicamente nos dará errores de compilación en las llamadas a este, por lo que debemos modificarlos manualmente.

12. Change Method Signature

- Seleccionamos la opción “Change method signature”.
- Nos presentará una ventana donde podremos cambiar la firma del método.



13. Infer Generic Type Arguments

- La refactorización de Eclipse permite inferir los tipos correctos de los argumentos para clases.

- Esta utilidad es especialmente utilizada para convertir código de versiones anteriores a Java 5 a esta versión y superiores.

- Podemos llamar a esta funcionalidad desde el explorer de Eclipse. Haciendo boton derecho en el proyecto, paquete o clase y seleccionando **Refactor > Infer Generic Type Arguments**. Veamos un ejemplo:

13. Infer Generic Type Arguments

- El código del siguiente ejemplo muestra la función ConcurrentHashMap que puede recibir tipos de argumentos genéricos, aunque no especifique el tipo de los parámetros.

```
private final ConcurrentHashMap map = new ConcurrentHashMap();
```

- Tras la refactorización, Eclipse determita el tipo correcto de los parámetros y produce lo siguiente:

```
private final ConcurrentHashMap<TimedKey, Object> map =
    new ConcurrentHashMap<TimedKey, Object>();
```

14. Migrate JAR File

- La migración de un fichero JAR permite actualizarlo directamente dentro de la ruta de un proyecto de versión superior. El proceso antes sería el siguiente:

1. En las propiedades del proyecto, se elimina el anterior JAR de la ruta del proyecto.
2. Posteriormente, se elimina de la carpeta manualmente.
3. Se copia el nuevo JAR con el nombre a través del cual es referenciado en los scripts codificados.
4. Se añade el nuevo JAR a la ruta del proyecto manualmente.

14. Migrate JAR File

- Con la refactorización, podemos hacerlo todo de manera más sencilla:
  1. Se selecciona **Refactor > Migrate Jars**.
  2. En el diálogo que se abre, se selecciona la localización del nuevo archivo JAR.
  3. En la barra de exploración se selecciona el JAR desde el proyecto que se quiere actualizar a una nueva versión. Seleccionando **Replace Jar file contents but preserve existing filename**, el nuevo JAR adquiere el nombre del archivo, sin romper las referencias hacia ese nombre.

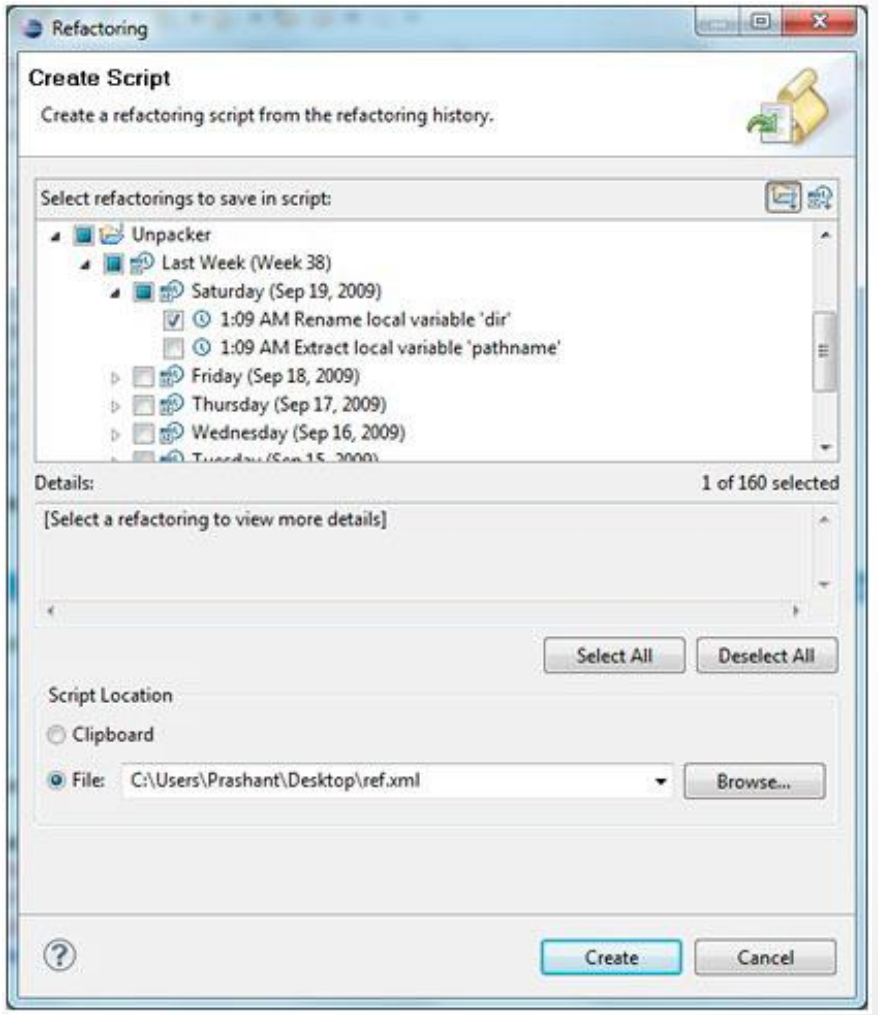
- En cualquier caso, al pulsar **Finish**, el JAR anterior se elimina y el nuevo es copiado a la ruta del proyecto pasando a ser utilizado como principal por el proyecto.

15. Refactoring scripts

- Permite exportar y compartir acciones de refactorización.
- La refactorización de scripts es muy útil cuando se va a distribuir una versión de una aplicación que puede producir errores en quienes ejecuten versiones anteriores.
- Entregando scripts con refactorización y sus librerías, los usuarios que corren versiones anteriores pueden hacer uso de los scripts en sus proyectos con las nuevas versiones de las librerías.

15. Refactoring scripts

- Para crear un script de refactorización:
  - **Refactor > Create Script**.
  - Se mostrará el listado de refactorizaciones.
  - Se seleccionan las refactorizaciones, se introduce una ruta, y se pulsa **Create** para generarlo.



15. Refactoring scripts

- Para aplicar un script de refactorización existente, se selecciona **Refactor > Apply Script**.
- En el cuadro de diálogo se selecciona la localización. Se pulsa **Next** para ver las refactorizaciones que se van a aplicar y con **Finish** se aplican.
- Ejemplo:
  - En nuestra primera versión del JAR tenemos la clase com.A, y en la segunda, es renombrada a com.B. Quienes usen la versión 1 y actualicen alguna librería, tendrán errores en el código.
  - Distribuyendo un script de refactorización que renombre las clases junto con el JAR, permitirá que se pueda pasar de la versión 1 a la 2 fácilmente.