

1. Refactorización

1.1 Introducción

Nesta parte da unidade didáctica que nos ocupa preténdense os seguintes obxectivos:

- Identificar a refactorización, os patróns de refactorización máis usuais, as vantaxes e limitacións da refactorización e a asociación coas probas.
- Realizar as refactorizacións de código que permita o contorno de desenvolvemento libre.

1.2 A refactorización

Introdución

A enxeñaría do software foi cambiando e creando novas técnicas que melloran a calidade do software e, sobre todo, que o fan máis adaptable e reutilizable. Un exemplo é a técnica de programación orientada a obxectos.

Os expertos viron que determinados problemas de deseño de código se repiten e que sería aconsellable formalizar e documentar a descrición dos mesmos e a súa solución para aforrar esforzos, de tal maneira que esta documentación permita a outros desenvolvedores aplicar a solución baseada na experiencia.

A POO combinada co uso de patróns de deseño, antipatróns de deseño e refactorización consegue crear código máis **adaptable** e **reutilizable**.

Un **patrón de deseño** documenta coñecemento relativo a solucións exitosas.

Refactorizar é "modificar a estrutura interna do software co obxecto de que sexa máis fácil de entender e modificar no futuro, de tal maneira que o comportamento observable do software ao executarse non se vexa afectado" (Fowler).

Un **antipatrón** de deseño documenta coñecemento relativo a solucións que darán problemas e documenta a refactorización necesaria para evitalas.

Patrón de deseño

Un patrón de deseño é unha forma estandarizada de solución de deseño dun problema recorrente atopado no deseño de software orientado a obxectos, xa probada con éxito no pasado e ben documentada.

É unha descrición sobre como resolver un problema de deseño, que pode ser utilizada en diversas situacións pero que deberá ser adaptada ao contexto do problema actual.

É unha ferramenta para o desenvolvedor pero, por si soa, non garante nada, xa que non é un esquema ríxido a seguir, necesita ao desenvolvedor e á súa creatividade.

Non é un deseño terminado que poida pasarse directamente a código.

Non é un principio abstracto, teoría ou idea non probados.

Non é unha solución que só funcionou unha vez.

Hai patróns de deseño en moitas áreas entre as que se atopa a POO.

Obxectivos

- Evitar a repetición de procura de solucións a problemas que outros deseñadores xa tiveron e solucionaron.
- Crear catálogos de patróns.
- Crear un vocabulario común entre deseñadores.
- Estandarizar o deseño pero non impor un modelo nin eliminar a creatividade.
- Facilitar a transmisión de coñecementos entre xeracións de deseñadores.

Clasificación

Existen máis patróns que os patróns de deseño. Unha posible clasificación dos patróns en xeral pode ser:

- **Patróns de arquitectura:** centrados na arquitectura do sistema. Prové de conxuntos predefinidos de subsistemas e como se relacionan.
- **Patróns de deseño:** centrados no deseño.
- **Patróns de codificación ou dialectos** (idiomas) que axudan a implementar aspectos particulares do deseño nunha linguaxe de programación específica.
- **Antipatróns** de deseño que de forma parecida aos patróns, reúnen solucións que se sabe que producen efectos negativos. Fornecen refactorizacións para transformar as solucións negativas en positivas.
- **Patróns de interacción** centrados no deseño de interfaces web.

Catálogo de patróns de deseño

O principal catálogo de patróns de deseño está no libro **Design Patterns** escrito pola Gang of Four (GoF) composto por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, referenciado como Gamma et al.1995 no que se recollían 23 patróns de deseño comúns agrupados en tres grupos:

- **patróns creacionais** para o proceso de instanciación de obxectos.
- **patróns estruturais** para combinar clases e obxectos e formar estruturas máis grandes.
- **patróns de comportamento** para organizar o fluxo de control dentro do sistema.

O grupo GoF describe un patrón utilizando un modelo con:

- Nome, normalmente en inglés.
- Nomes alternativos.
- Clasificación do patrón: creacional, estrutural ou de comportamento.
- Descrición do problema que pretende resolver o patrón.

- Motivos polos que se debe de aplicar o patrón.
- Estrutura e descrición das clases e entidades que participan no patrón así como as relacións entre elas.
- Vantaxes e inconvenientes da aplicación do patrón.
- Implementación. Indica como aplicar o patrón.
- Código fonte de exemplo.
- Usos coñecidos en sistemas reais.
- Relación con outros patróns.

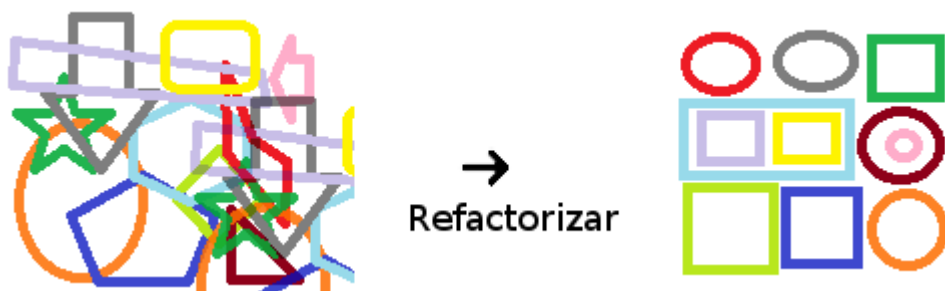
Refactorización

Cando un programador empeza un novo traballo de programación, producirá seguramente un código limpo e ben organizado seguindo os patróns de deseño, pero, a medida que ese código se vaia mantendo e, aínda que funcione correctamente, o código vaise leando cada vez máis e pode resultar necesario reorganizalo.

O mantemento do código pódese complicar se:

- Intervenien diferentes programadores.
- Hai que incorporar novas funcionalidades. Aínda se complica máis se, para incorporar as novas funcionalidades, se incorporan anacos adaptados do código doutros programas.

Canto máis antigo e máis grande é o código, máis evidente é a necesidade de reorganizar.

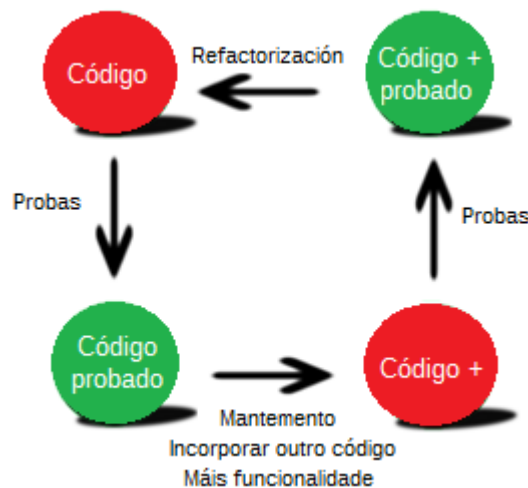


A refactorización é a transformación controlada de código fonte que asegura a reorganización do código para mellorar o seu mantemento e/ou facelo máis comprensible ou para saír dalgún antipatrón.

- Non altera o seu comportamento externo pero mellora a súa estrutura interna.
- Permite tomar deseños defectuosos, con código que non segue os patróns de deseño, por exemplo con duplicidade innecesaria, e adaptalo para conseguir un bo e ben organizado.

Refactorización e probas

A referencia clásica para a refactorización é o libro **Refactoring** de Martin Fowler de 1999. Os desenvolvedores alternan a inserción de novas funcionalidades e probas con refactorización.



O proceso de alternar consiste en:

- Probar automaticamente o programa coa nova funcionalidade mediante un lote de casos de proba que avalen o seu correcto funcionamento.
- Analizar as refactorizacións a realizar.
- Mentres haxa refactorizacións pendentes débense de realizar unha a unha e por cada refactorización:
 - Recoméndase realizar os cambios de forma progresiva, un a un e paso a paso.
 - Repetir a proba automática obtendo os mesmos resultados que antes de refactorizar.
 - Se é necesario, engadir novas probas.

É fundamental que os cambios na refactorización se realicen un a un e paso a paso. Por exemplo, se un método é moi grande e se quere dividir en métodos máis pequenos, a refactorización manual consistiría en:

- Crear os novos métodos baleiros. Executar probas. Pode ser que haxa problemas coa definición dos novos métodos.
- Escribir o código dos novos métodos. O método orixinal segue existindo. Ninguén chama aos novos métodos. Executar probas.
- Posiblemente habería que crear novas probas unitarias para os novos métodos. Executar probas cos novos métodos.
- Substituír no método orixinal, os anacos de código correspondentes polas chamadas aos novos métodos un a un se é posible. Executar probas.

As probas unitarias permiten comprobar que o código funciona correctamente e, tamén, teñen as vantaxes (Massol e Husted, 2003):

- Preveñen probas de regresión, é dicir, probas para detectar un erro que aparece despois dunha modificación no código.
- Limitan as probas de depuración (debug) de código.
- Achegan confianza á hora de modificar o código xa que se poden executar novamente e verificar se a aplicación dun cambio foi correcta.
- Facilitan a refactorización.
- É a primeira proba á que se somete o código fonte. Se o código non é fácil de probar de forma unitaria é un sinal que indica que algo non vai ben e que quizais se necesita reorganizalo.

As probas unitarias presentan as vantaxes anteriores, pero tamén teñen un custo asociado xa que deben de ser mantidas a medida que se modifica o código fonte. Isto significa que se se elimina funcionalidade do sistema deben eliminarse os casos de proba da devandita funcionalidade, se se agrega funcionalidade deberanse agregar novos casos e, se se modifica algunha funcionalidade existente, as probas unitarias deberán ser actualizadas de acordo ao cambio realizado.

As vantaxes aumentan se se realizan as probas de forma automática xa que entón dispónse dun sistema automático de proba utilizable en calquera momento. Estas probas son imprescindibles para a refactorización xa que, sen elas, sería unha operación de alto risco.

[Vantaxes da refactorización](#)

Tanto Fowler como Piattini e García consideran que a refactorización aumenta a produtividade xa que:

- Facilita a comprensión do código fonte, principalmente para os desenvolvedores que non estiveron involucrados desde o comezo do desenvolvemento, xa que fai que sexa máis fácil de ler, que exprese de forma máis clara cales son as súas funcións, e que sexa o máis autodocumentable posible.
- Reduce os erros xa que a comprensión do código permite tamén detectalos con máis facilidade.
- Permite programar máis rápido xa que permite mellorar os deseños de base.
- Facilita os cambios.

Limitacións da refactorización

Tanto Fowler como Piattini e García consideran que as áreas conflitivas para a refactorización son as bases de datos, e os cambios de interfaces.

- É moi custoso aplicar cambios que estean ligados con bases de datos, xa que se isto supón cambios no esquema da base de datos, habería que aplicar os cambios e logo migrar os datos.
- O cambio nunha interface non ten demasiada complexidade se se ten acceso ao código fonte de todos os clientes da interface a refactorizar e ademais pódese modificar. Si é problemático cando esta se converte no que (Fowler et al, 1999) chama interface publicada (published interface) e non se pode modificar o código fonte dos clientes da mesma.

Malos cheiros no código

Tanto Fowler como Piattini e García describen os **bad smells** (malos cheiros) como os síntomas que indican que se debe de refactorizar. Algúns deles son:

- **Código duplicado.** É a principal razón para refactorizar. Se se detecta o mesmo código en máis dun lugar, débese buscar a forma de extraelo e unificalo.
- **Método longo.** Os métodos curtos son máis fáciles de reutilizar.
- **Clase grande.** Se unha clase ten demasiados métodos públicos, pode ser conveniente dividila.
- **Lista de parámetros extensa.** Os métodos con moitos parámetros son difíciles de comprender.
- **Cambio diverxente** (Divergent change). Preséntase cando ao facer o cambio A nunha clase, hai que modificar uns métodos e ao facer o cambio B hai que modificar outros. Deberíase entón de separar a clase orixinal en varias, de modo que un cambio afecte a unha soa das clases novas. Este caso é o oposto do seguinte.
- **Cirurxía de escopeta** (Shotgun surgery). O contrario do anterior. Este síntoma preséntase cando un cambio nun determinado lugar, obriga a realizar outros en diversos lugares polo que se debería de reunir nunha clase o que está distribuído en varias e poida estar afectado polo cambio.
- **Envexa de funcionalidade** (Feature envy). Prodúcese cando un método utiliza máis elementos doutra clase que da propia. Adóitase resolver o problema pasando o método á clase da que se queren usar os compoñentes.
- **Clase de datos** (Data class). Prodúcese en clases que só teñen atributos e métodos públicos de acceso a eles ("get" e "set"). Este tipo de clases deberían cuestionarse dado que non adoitan ter comportamento algún.

- **Legado rexeitado** (Refused bequest). Prodúcese cando hai subclases que usan poucas características das súas superclases. Se as subclases non requiren todo o que lles provén por herdanza das súas superclases, adoita indicar que non é correcta a xerarquía de clases.

A detección de malos cheiros no código é difícil. Ás veces, mesmo pode parecer que contradí algúns patróns de deseño. Por exemplo, o mal cheiro de "envexa de datos" pode parecer contradictorio co patrón "visitante" que, cando se aplica, permite crear unha clase externa para actuar nos datos doutra clase.

Cando non refactorizar

- Cando o código non funciona, refactorizar non é a solución xa que non arranxa erros. Pode axudar a facer o código máis comprensible e, por tanto, pode axudar a atopar o problema pero non arranxa o problema. Refactorizar un código que non funciona pode engadir novos problemas ao código xa que non se dispón de probas fiables que garantan que o código segue tendo a mesma funcionalidade despois da refactorización.
- Cando se necesitan novas funcionalidades, xa que refactorizar non engade funcionalidade, pero si será máis fácil engadir novas funcionalidades con éxito se o código está refactorizado.
- Cando está próxima a data de entrega do software.

Patróns de refactorización

Exemplos:

- **Clases:** extraer clase, extraer subclase, extraer superclase, mover, copiar,
- **Campos:** ascender, descender,
- **Métodos:** Engadir parámetros, eliminar parámetros, extraer método, cambiar modificador de acceso, ascender, descender, renomear, substituír condicional con polimorfismo, substituír código de erro con Exception,
- **Interfaces:** extraer,

Exemplo de refactorización manual para substituír os condicionais por polimorfismo

Paso 1: Separar as condicións nun único método e subir ese método ao máis alto posible na xerarquía.

```
public class Metro {

    private int estado;
    static final private int PARADO = 0;
    static final private int EN_MARCHA = 1;
    static final private int PARANDO = 2;
    static final private int ARRANCANDO = 3;

    public void cambiaEstado() {
        if (estado == PARADO) {
            estado = ARRANCANDO;
        } else if (estado == EN_MARCHA) {
            estado = PARANDO;
        } else if (estado == PARANDO) {
            estado = PARADO;
        } else if (estado == ARRANCANDO) {
            estado = EN_MARCHA;
        } else {
            throw new RuntimeException("Estado desconocido");
        }
    }
}
```

Paso2:

- Crear unha clase por cada posible estado.
- Substituír == por *instanceof*()
- Substituír as variables estáticas enteiras por referencias a instancias desas clases.

```
public interface Estado {

}

public class Parado implements Estado{

}

public class Arrancando implements Estado{

}

public class Parando implements Estado{

}

public class Metro {

    private Estado estado;
    static final private Estado PARADO = new Parado();
    static final private Estado EN_MARCHA = new EnMarcha();
    static final private Estado PARANDO = new Parando();
    static final private Estado ARRANCANDO = new Arrancando();

    public void cambiaEstado() {
        if (estado instanceof Parado) {
            estado = ARRANCANDO;
        } else if (estado instanceof EnMarcha) {
            estado = PARANDO;
        } else if (estado instanceof Parando) {
            estado = PARADO;
        } else if (estado instanceof Arrancando) {
            estado = EN_MARCHA;
        } else {
            throw new RuntimeException("Estado desconocido");
        }
    }
}
```


Paso3:

- Crear o método *siguiente()* dentro de cada clase para que devuelva o siguiente estado.
- Eliminar os if.
- Eliminar throw.

```
public interface Estado {
    public Estado siguiente();
}

public class Parado implements Estado{
    public Estado siguiente(){
        return new Arrancando();
    }
}

public class Arrancando implements Estado{
    public Estado siguiente(){
        return new EnMarcha();
    }
}

public class Parando implements Estado{
    public Estado siguiente(){
        return new Parado();
    }
}

public class Parado implements Estado{
    public Estado siguiente(){
        return new Arrancando();
    }
}

public class Metro {

    private Estado estado;

    public void cambiaEstado() {
        estado = estado.siguiente();
    }

    public Estado conseguirEstado() {
        return estado;
    }

    public void iniciarEstado() {
        estado = new Parado();
    }
}
```

Refactorizar código java con NetBeans

Introdución

NetBeans permite reestructurar código Java sen cambiar o comportamento do programa. A refactorización é unha operación que conserva a funcionalidade do código orixinal e permite:

- Facer o código máis fácil de ler e comprender.
- Facer máis rápida a actualización.
- Facer máis fácil o engadido de novas funcionalidades.
- Borrar repeticións innecesarias de código.
- Permitir usar o código para necesidades máis xerais.
- Mellorar o rendemento do código.

NetBeans axuda á reestruturación de código Java mostrando as partes do software que se verán afectadas, permitindo incluír ou excluír cambios e realizando os cambios seleccionados. Por exemplo, se a reestruturación consiste no cambio de nome dunha clase, NetBeans encontrará a aparición dese nome no código e ofrecerá a posibilidade de cambialo; se consiste en cambiar a estrutura do código, actualizará o resto do código para reflectir ese cambio de estrutura.

Opción Refactor

As posibles reestruturacións que se poden facer están accesibles dende a opción **Refactor** do menú principal ou elixindo un elemento do código, facendo clic dereito co rato e elixindo Refactor. Esas reestruturacións son:

Rename...	Ctrl+R
Move...	Ctrl+M
Copy...	Alt+C
Safely Delete...	Alt+Delete
Inline...	Alt+Shift+N
Change Method Parameters...	
Pull Up...	
Push Down...	
Extract Interface...	
Extract Superclass...	
Use Supertype Where Possible...	
Introduce	>
Move Inner to Outer Level...	
Convert Anonymous to Member...	Ctrl+Alt+Shift+A
Encapsulate Fields...	
Replace Constructor with Factory...	
Replace Constructor with Builder...	
Invert Boolean...	
Inspect and Transform...	

Introdución breve aos casos de reestruturación

Reestruturación	Descrición
Cambiar de nome	Cambia o nome dunha clase, interface, variable ou método por algo máis significativo e actualiza todo o código fonte do proxecto para reflectir este cambio
Introducir variable, constante, campo, parámetro ou método	O programador selecciona un fragmento de código, xérase unha declaración baseada no código seleccionado e substitúe o bloque de código por unha chamada a esa declaración. A declaración pode ser a creación dunha variable, constante, campo, parámetro ou método
Cambiar parámetros dun método	Engade, elimina, modifica ou cambia a orde dos parámetros dun método ou cambia o modificador de acceso (<i>public</i> , <i>private</i> , <i>protected</i>)
Encapsular campos	Xera métodos get e set para un campo e opcionalmente actualiza todas as referencias a ese campo utilizando os métodos get e set
Ascender	Move métodos e campos a unha superclase da que herdará a actual clase
Descender	Move clases internas, métodos e campos a unha subclase da clase actual
Mover clase	Move unha clase a outro paquete ou dentro doutra clase. Ademais todo o código fonte do proxecto é actualizado para referenciar a clase no novo paquete
Copiar clase	Copia unha clase ao mesmo ou diferente paquete
Mover de nivel interior a exterior	Move unha clase membro un nivel cara arriba na xerarquía de clases
Converter anónimo en membro	Converte unha clase anónima en clase membro que contén nome e construtor. A clase anónima é substituída cunha chamada á clase membro
Extraer interface	Crea unha nova interface formada a partir do método público non estático seleccionado en unha clase ou interface. De extraerse dunha clase, esta implementará a nova interface creada. De extraerse dunha interface, esta estenderá a interface creada.
Extraer superclase	Informa ao programador dos métodos e campos que se poden mover a unha superclase. O programador selecciona os que quere mover e NetBeans: <ul style="list-style-type: none">▪ crea unha nova clase abstracta que conterá eses campos e métodos▪ cambia a clase actual para estender a nova clase, e▪ move os métodos e campos seleccionados á nova clase

Desfacer e refacer cambios

A icona **Undo** (desfacer) activarase despois de facer unha refactorización permitindo desfacer todos os cambios en todos os arquivos afectados pola reestruturación.

A icona **Redo** (refacer) activarase despois de desfacer unha reestruturación e servirá para volver facer a refactorización.

As iconas que se utilizar para desfacer e refacer unha factorización son as seguintes:

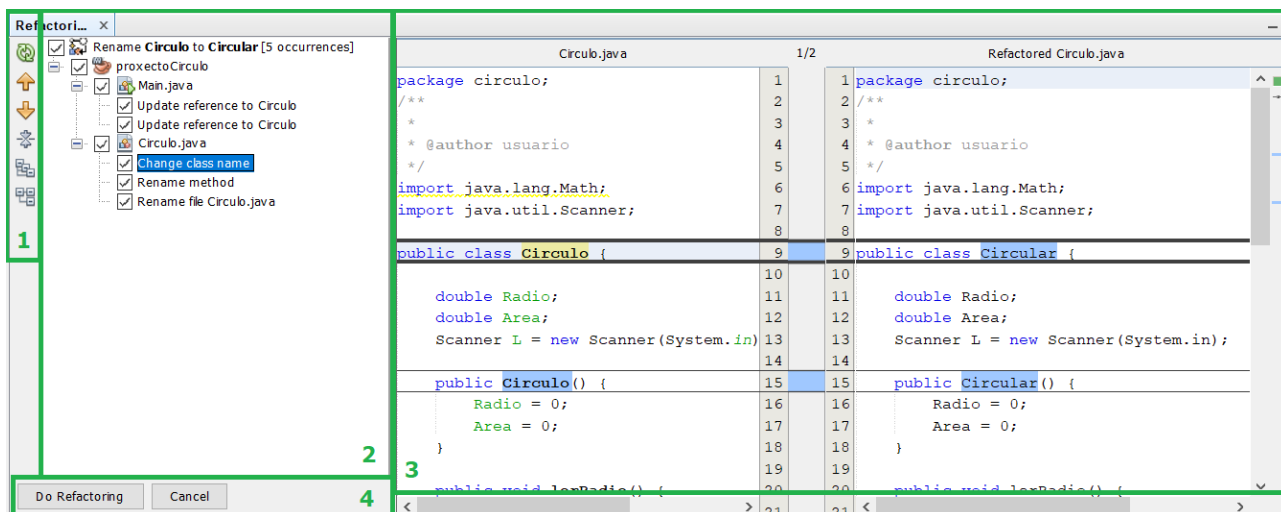


Ventá Refactoring

Esta ventá aparece cando no proceso de reestruturación, se elixe ter unha vista previa dos cambios antes de levar a cabo realmente a reestruturación.

Por exemplo para cambiar o nome dunha clase ou interface, hai que ir á ventá de proxecto ou á ventá de edición, facer clic co botón dereito sobre a clase ou interface, escoller Refactor > Rename..., teclear o novo nome da clase e premer no botón de Preview.

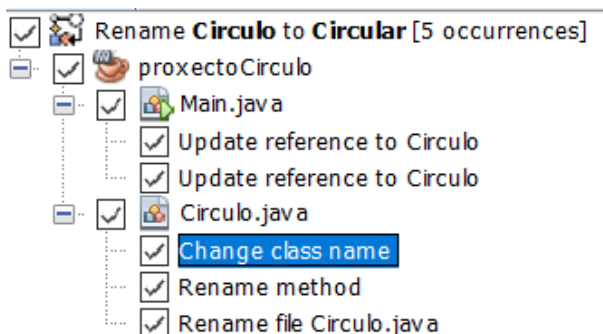
A ventá Refactoring está dividida en varias zonas:



- **Zona 1.** Menú de iconas para actuar sobre a vista previa da reestruturación que permite:

Icona	Descrición
	Actualizar as zonas despois de modificar as condicións de reestruturación.
	Expandir totalmente ou contraer totalmente os nodos da árbore da zona 2
	Mostrar a vista lóxica
	Mostrar a vista física
	Moverse polas ocorrencias

- **Zona 2.** Mostra o número de ocorrencias da reestruturación no proxecto e a árbore de ocorrencias, resaltando a ocorrencia considerada actual que é a que se está vendo detallada na zona 3. Pódese:
 - marcar ou desmarcar unha ocorrencia para incluíla ou excluila da reestruturación,
 - expandir ou contraer cada nodo da árbore ou
 - facer dobre clic riba dun nodo para que sexa a ocorrencia actual e se mostre na zona 3.



- **Zona 3.** Mostra os detalles da ocorrencia resaltada na zona 2 para permitir comparar entre o código fonte inicial (na parte esquerda) e o que resultaría despois da reestruturación (na parte dereita).

A vista dos dous arquivos ten:

- Numeradas as liñas de código.
- Os cambios resaltados.
- Todas as liñas de código afectadas polos cambios aparecen remarcadas entre liñas paralelas finas que conectan os dous arquivos.
- A ocorrencia actual remarcada entre liñas paralelas grosas que conectan os dous arquivos.

Circulo.java	1/2	Refactored Circulo.java
package circulo;	1	1 package circulo;
/**	2	2 /**
*	3	3 *
* @author usuario	4	4 * @author usuario
*/	5	5 */
import java.lang.Math;	6	6 import java.lang.Math;
import java.util.Scanner;	7	7 import java.util.Scanner;
	8	8
public class Circulo {	9	9 public class Circular {
	10	10
double Radio;	11	double Radio;
double Area;	12	double Area;
Scanner L = new Scanner(System.in);	13	Scanner L = new Scanner(System.in);
	14	14
public Circulo() {	15	public Circular() {
Radio = 0;	16	Radio = 0;
Area = 0;	17	Area = 0;
}	18	}

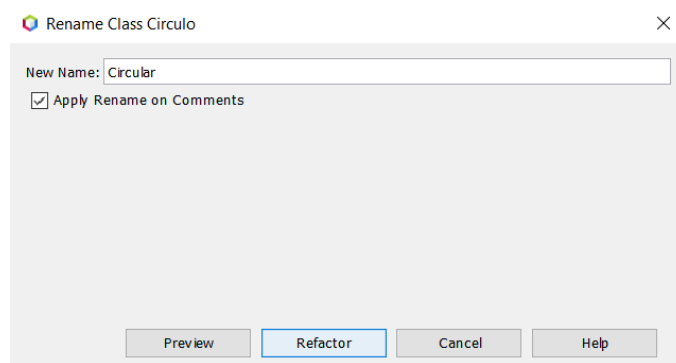
- **Zona 4.** Contén os botóns que permiten levar a cabo a reestruturación ou cancelar a mesma.

Casos de reestruturación

Cambiar de nome

Para poder renomear unha clase, interface, variable ou método, débese ir á ventá de proxecto ou á ventá de edición, facer clic co botón dereito sobre a clase ou interface e escoller Refactor > Rename... Aparece unha caixa que permite:

- Teclear o nome novo
- Opcionalmente cambiar o nome nos comentarios
- Ter unha vista previa dos cambios
- Realizar a refactorización
- Cancelar a operación de refactorización
- Recorrer á axuda en liña de NetBeans



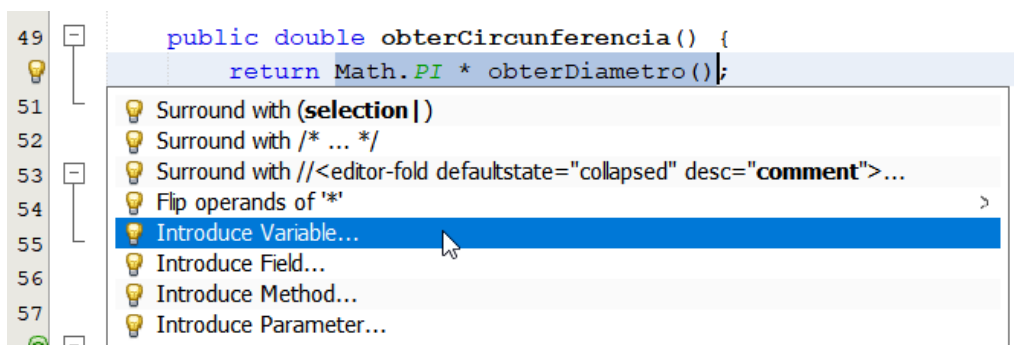
Cambiará todas as aparicións ou referencias á clase Circulo por Circular no proxecto e cambiará o nome de Circulo.java por Circular.java.

Surround with... Introduce...

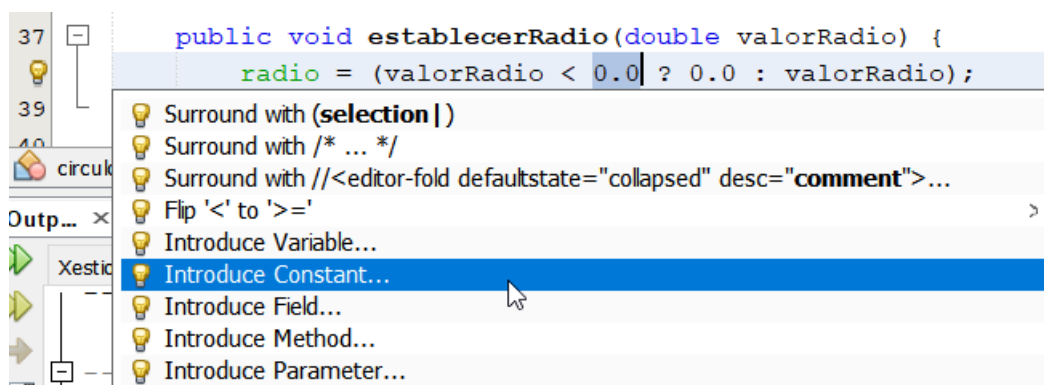
NetBeans pode facer suxestións sobre a conveniencia de encerrar un código seleccionado entre estruturas de control, sobre a posibilidade de introducir unha nova variable, campo ou método co código seleccionado ou comentalo.

Para iso, na ventá de edición de código fonte, débese seleccionar a expresión ou grupo de instrucións e premer **Alt-Enter**. Dependendo do código seleccionado, NetBeans suxerirá encerrar o código entre as estruturas de control e comentarios que suxire, ou introducir variable, constante, campo, parámetro ou método.

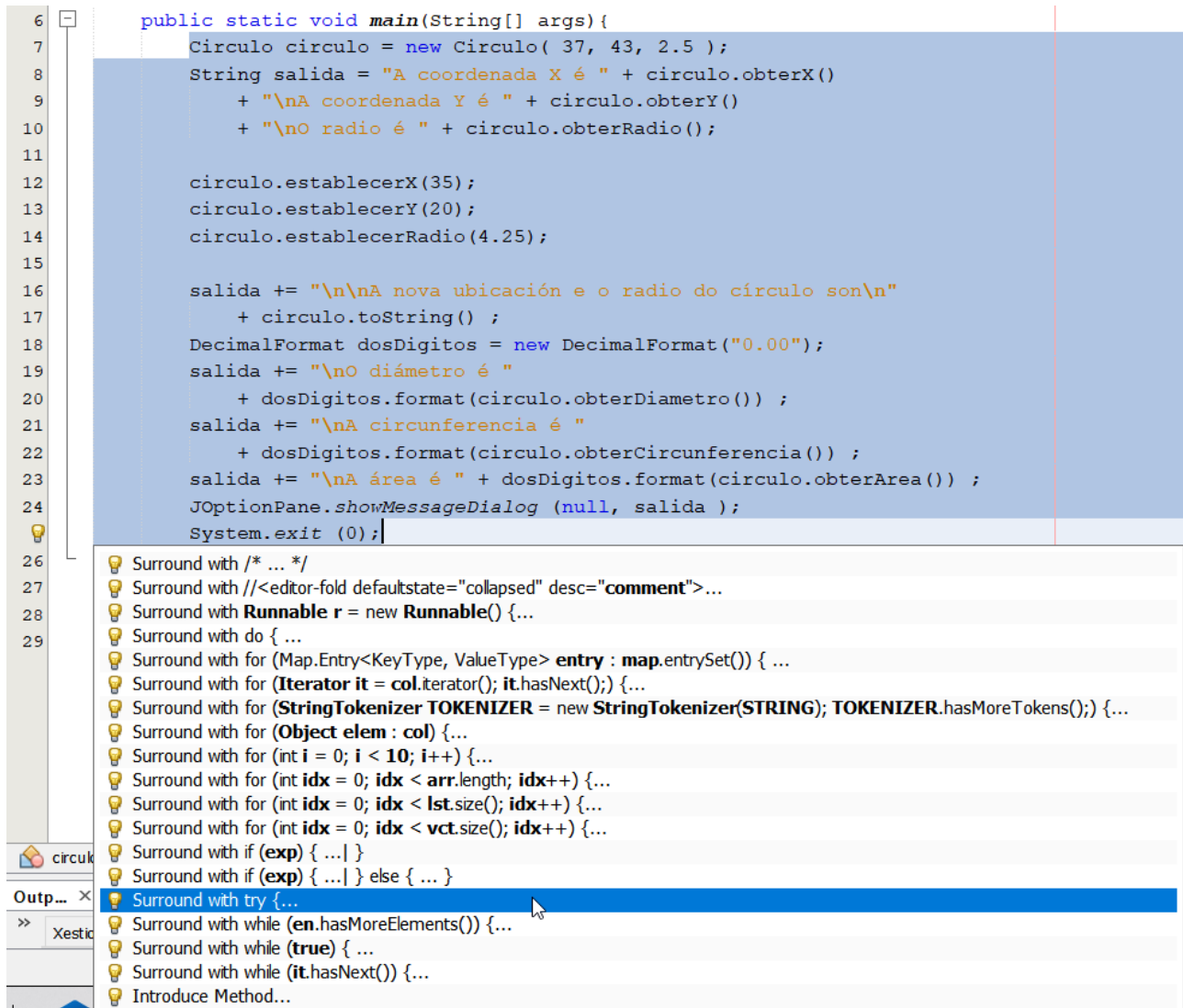
Por exemplo para a seguinte expresión seleccionada, NetBeans suxire comentala ou crear con ela unha nova variable, campo ou método e non suxire constante porque non ten sentido.



Por exemplo para a seguinte expresión seleccionada, NetBeans suxire ademais crear con ela unha constante.



Por exemplo para a seguinte expresión seleccionada, NetBeans suxire encerrala entre estruturas de control, try, comentarios ou introducir método:



Introducir variable, constante, campo, parámetro ou método

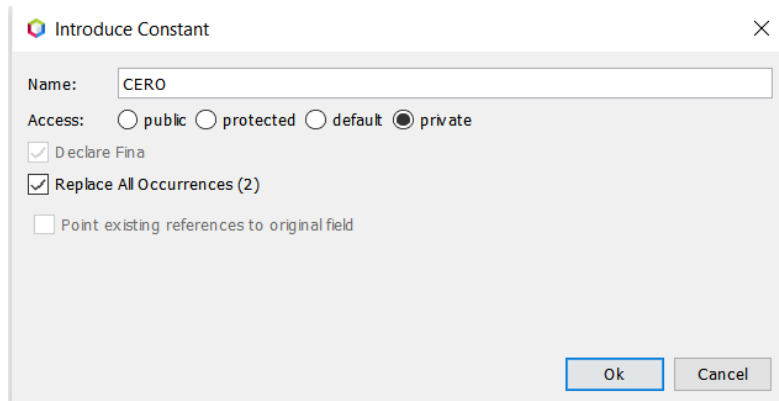
Pódese cambiar un fragmento de código seleccionado para crear con el unha variable, unha constante ou un método. Isto faise cando se quere dividir o código en anacos máis pequenos ou máis significativos e así facilitar as modificacións futuras, a reusabilidade do código e mellorar a comprensión do código.

Pode introducirse variable, constante, campo ou método como se viu no apartado anterior, ou seleccionando a expresión no código fonte e elixindo no menú principal Refactor > Introduce variable... ou constante ou campo ou parámetro ou método. Con calquera das dúas formas aparecerá unha nova ventá na que se detallará a reestruturación como se indica de contado.

Introducir constante

Para introducir constante hai que teclear o nome da constante (NetBeans propón un nome en maiúsculas), o tipo de acceso, non deixa cambiar a declaración final e permite tamén substituír todas as coincidencias ou só a que está seleccionada.

Considerando seleccionado o valor 0.0 do método establecerRadio():



Cambiaranse as dúas coincidencias polo nome da constante.

```
37 public void establecerRadio(double valorRadio) {  
38     radio = (valorRadio < CERO ? CERO : valorRadio);  
39 }
```

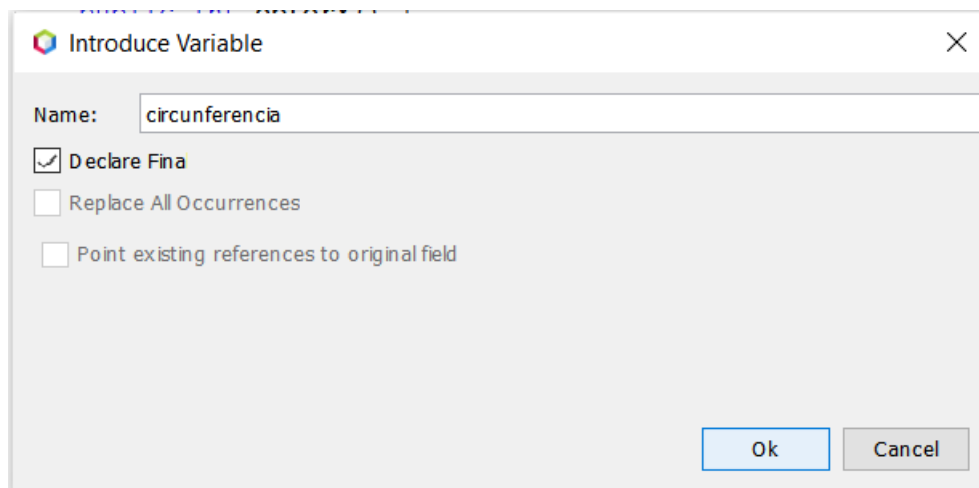
Aparecerá a declaración da constante na clase.

```
40 private static final double CERO = 0.0;
```

Introducir variable

Para introducir variable hai que teclear o nome da nova variable (NetBeans suxire un nome en minúsculas), a declaración final se é que se desexa e se aparecese esa expresión en mais dun sitio poderíase decidir se a substitución se realizaría en todas as coincidencias ou só na selección actual.

Considerando seleccionada a expresión *Math.PI * obterDiametro()* do módulo *obterCircunferencia()*:



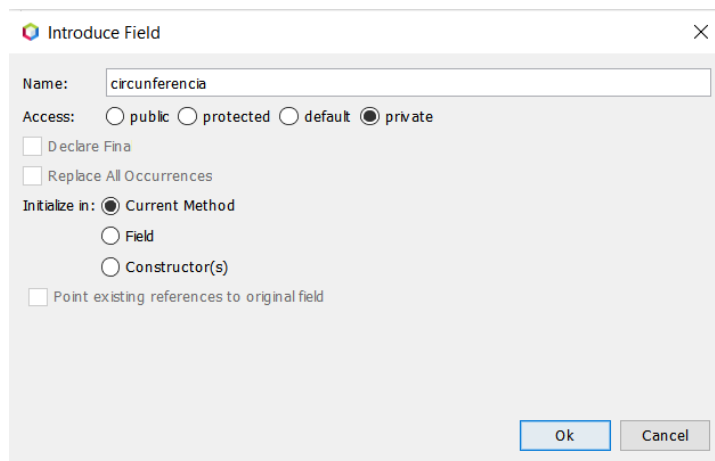
Aparecerá substituída a expresión por esa variable dentro do método actual:

```
49  public double obterCircunferencia() {  
51      final double circunferencia = Math.PI * obterDiametro();  
52      return circunferencia;  
}
```

Introducir campo

Para introducir campo hai que indicar as características do campo (nome, tipo de acceso, se a declaración é Final, e onde se inicializará).

Considerando seleccionada a expresión *Math.PI * obterDiametro()* do módulo *obterCircunferencia()*:



The 'Introduce Field' dialog box is shown. It has a title bar with a close button. The 'Name' field contains 'circunferencia'. The 'Access' section has radio buttons for 'public', 'protected', 'default', and 'private', with 'private' selected. There are checkboxes for 'Declare Final' and 'Replace All Occurrences', both of which are unchecked. The 'Initialize in' section has radio buttons for 'Current Method', 'Field', and 'Constructor(s)', with 'Current Method' selected. There is also a checkbox for 'Point existing references to original field' which is unchecked. At the bottom right are 'Ok' and 'Cancel' buttons.

Aparecerá ese novo campo declarado na clase

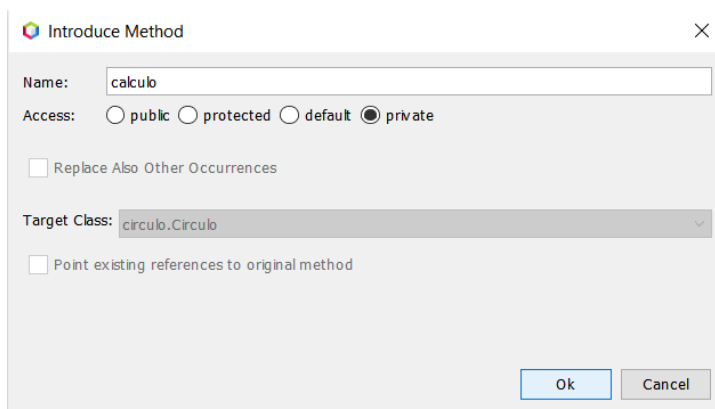
```
53  private double circunferencia;
```

Aparecerá o campo inicializado no método actual

```
49  public double obterCircunferencia() {  
51      circunferencia = Math.PI * obterDiametro();  
52      return circunferencia;  
}
```

Introducir método

De elixir método, haberá que indicar o nome do método e o tipo de acceso. Considerando seleccionada a expresión *Math.PI * obterDiametro()* do módulo *obterCircunferencia()*:



The 'Introduce Method' dialog box is shown. It has a title bar with a close button. The 'Name' field contains 'calculo'. The 'Access' section has radio buttons for 'public', 'protected', 'default', and 'private', with 'private' selected. There is a checkbox for 'Replace Also Other Occurrences' which is unchecked. The 'Target Class' dropdown menu is set to 'circulo.Circulo'. There is a checkbox for 'Point existing references to original method' which is unchecked. At the bottom right are 'Ok' and 'Cancel' buttons.

Crearase ese método e a referencia a el dentro do método actual.

```

49 public double obterCircunferencia() {
50     return calculo();
51 }
52
53 private double calculo() {
54     return Math.PI * obterDiametro();
55 }
56

```

Se aparece unha mensaxe de erro ao introducir un método, pode ser debido a que o código seleccionado non pode formar un método como por exemplo:

- A selección non pode ter máis dun parámetro de saída.
- A selección non pode ter instrucións break nin continue se a seguinte operación a facer non está dentro da selección.
- A selección non pode conter unha sentencia return que non sexa a última da selección.
- A selección non pode ter un return condicional.

Cambiar parámetros dun método

Cambiar parámetros nun método permite alterar a firma do método engadindo parámetros, cambiando a orde dos parámetros, cambiando o tipo de acceso do método e permitindo que eses cambios se propaguen en todo o código.

Para proceder a estes cambios, hai que estar na ventá de edición do código fonte, sobre o método a cambiar, facer clic co botón dereito do rato e elixir Refactor → Change Method Parameters... Ábrese entón a ventá de **Change Method Parameters** na que ademais de poder usar os botóns de calquera ventá de reestruturación (para ter unha vista previa dos efectos dos cambios, para facer efectiva a reestruturación, cancelala ou ter axuda en liña), pódense agregar parámetros, eliminalos, cambiar a orde dos parámetros colocándoos antes ou despois, e cambiar o modificador de acceso do método.

Change Method Parameters

Parameters:

Type	Name	Default Value
double	valorRadio	

Buttons: Add, Remove, Move Up, Move Down

Access: <do not change> Return Type: void Name: establecerRadio

☐ Update Existing Javadoc of This Method

Code Generation:

☒ Update Existing Method

☐ Create New Method and Delegate from Existing Method

Method Signature Preview

```
public void establecerRadio(double valorRadio)
```

Buttons: Preview, Refactor, Cancel, Help

- Engadir parámetros

Na ventá **Change Method Parameters** prémese o botón **Add** e na táboa de parámetros tecléase o nome, tipo e valor predeterminado do parámetro para colocar nas chamadas ao método. Para editar o nome, tipo ou valor predeterminado é necesario facer dobre clic na cela correspondente.

- Cambiar orde de parámetros

Na ventá **Change Method Parameters**, selecciónase o parámetro que se quere mover e prémese nos botóns **Move Up** ou **Move Down** segundo se queira.

- Cambiar tipo de acceso

Na ventá **Change Method Parameters**, selecciónase o modificador de acceso que se necesite.

Como exemplo engadiranse dous parámetros no método *trasladarCentro()* para que se traslade o centro do círculo segundo os valores dos parámetros.

```

66 public void trasladarCentro(){
67     x+=5;
68     y+=5;
69 }

```

Que é utilizado en Main.java

```

25 circulo.trasladarCentro();
26 salida="\nA nova ubicación e o radio do círculo son\n"+circulo.toString();
27 JOptionPane.showMessageDialog (null, salida);

```

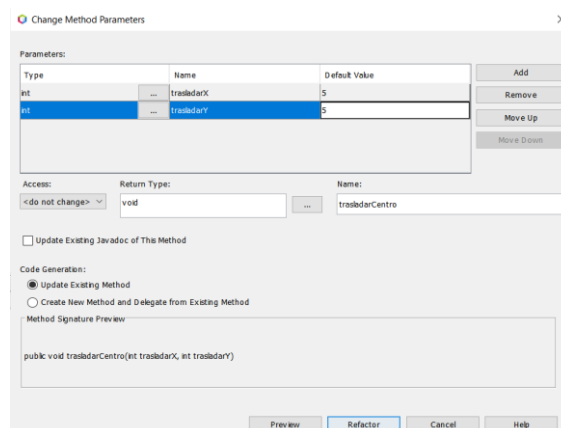
E en CirculoTest.java:

```

113 @Test
114 public void testTrasladarCentro() {
115     System.out.println("Trasladar centro");
116     Circulo instance=new Circulo();
117     int resultx=instance.obterX();
118     int resulty=instance.obterY();
119     instance.trasladarCentro();
120     int resultnx=instance.obterX();
121     int resultny=instance.obterY();
122     assertEquals(resultx+5,resultnx);
123     assertEquals(resulty+5,resultny);
124 }

```

Engadiranse os parámetros *trasladarY* e *trasladarX* de tipo int e con valor 5 por defecto.



Quedando os códigos modificados como:

```
66 public void trasladarCentro(int trasladarX, int trasladarY){
67     x+=5;
68     y+=5;
69 }

25 circulo.trasladarCentro(5, 5);
26 salida="\nA nova ubicación e o radio do círculo son\n"+circulo.toString();
27 JOptionPane.showMessageDialog (null, salida);

113 @Test
114 public void testTrasladarCentro() {
115     System.out.println("Trasladar centro");
116     Circulo instance=new Circulo();
117     int resultx=instance.obterX();
118     int resulty=instance.obterY();
119     instance.trasladarCentro(5, 5);
120     int resultnx=instance.obterX();
121     int resultny=instance.obterY();
122     assertEquals(resultx+5, resultnx);
123     assertEquals(resulty+5, resultny);
124 }
```

A reestruturación efectuada non cambia o código do método, polo que si despois de reestruturar se necesítase cambiar as liñas:

```
x+=5;
y+=5;
```

por:

```
x+=trasladarX;
y+=trasladarY;
```

deberíase de facer manualmente.

Encapsular campos

Encapsular un campo consiste en facer que o campo teña acceso **private** e só sexa accesible utilizando un par de métodos de tipo *get* e *set* que serán públicos.

A reestruturación para encapsular campos en NetBeans é moito máis flexible e permite xerar de forma xeral métodos de tipo *get* e *set* para acceder a un campo. Este proceso subdivídese en:

- Xerar os métodos de acceso.
- Axustar os modificadores de acceso para os campos.
- Substituír as referencias a ese campo no código por chamadas aos métodos de acceso.

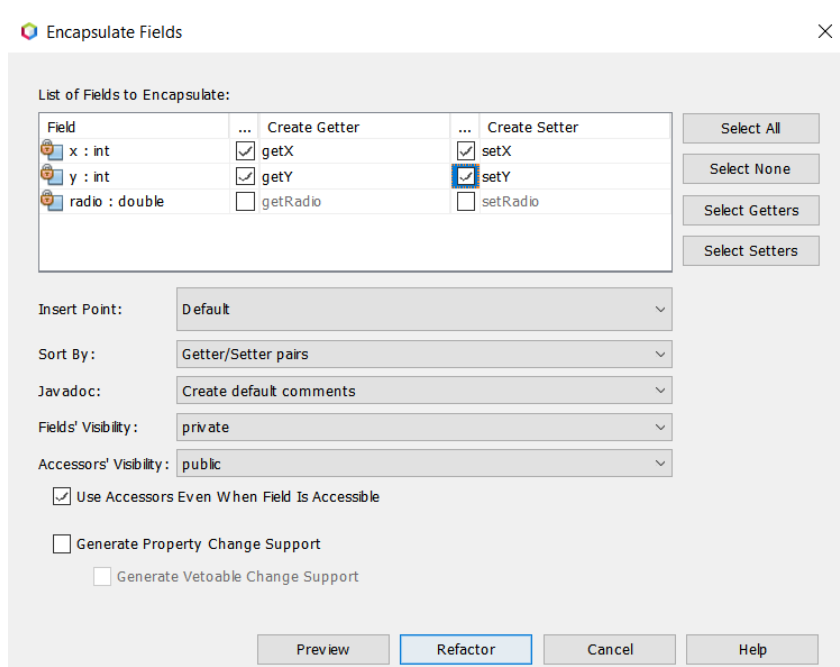
Esta reestruturación non eliminará métodos que xa existan para acceder aos campos dende dentro da clase.

Para encapsular un campo, débese facer clic co botón dereito do rato sobre o campo ou unha referencia ao campo e elixir Refactor > Encapsulate Fields... e ábrese a caixa de diálogo **Encapsulate Fields**.

Aparecen a lista de campos que se poden encapsular e aparece marcado o campo seleccionado co que se accedeu á reestruturación. Pódese:

- Marcar os campos que se queren encapsular de forma individual utilizando os campos checkbox que están ao carón dos campos na lista de campos, ou utilizar os botóns que están á dereita e que permitirán marcar todo, desmarcar todo, marcar só os métodos de tipo get ou só os de tipo set.
- Indicar en que punto do código se deben de inserir os métodos novos: como primeiro método, como último método ou despois de algún dos métodos existentes na clase (aparece a lista de todos).
- Indicar como se deben de colocar os diferentes métodos tipo get e set dentro da localización anterior: intercalando get e set de cada campo, por nome de método ou primeiro todos os get e despois os set.
- Indicar como se quere que sexa a documentación dos métodos.
- Indicar a visibilidade dos campos que van a estar encapsulados.
- Indicar a visibilidade dos métodos tipo get e set.
- Desmarcar o checkbox **Use Accessor Even When Field Is Accessible** se se queren utilizar os novos métodos de acceso aínda que o campo xa sexa accesible.

Como exemplo, encapsularanse os campos x, y da clase Circulo, creando os métodos get e set, poñendo todos os métodos get xuntos e despois os set xuntos e colocando os métodos como os primeiros da clase.



Que dará como resultado:

```
public void trasladarCentro() {
    setX(getX() + 5);
    setY(getY() + 5);
}

/**
 * @return the x
 */
public int getX() {
    return x;
}

/**
 * @param x the x to set
 */
public void setX(int x) {
    this.x = x;
}

/**
 * @return the y
 */
public int getY() {
    return y;
}

/**
 * @param y the y to set
 */
public void setY(int y) {
    this.y = y;
}
```

e inclúe as chamadas a estes métodos nos métodos que fan referencia aos campos x e y:

```
61  @Override
62  public String toString() {
63      return "Centro = [" + getX() + ", " + getY() + "]; Radio = " + radio;
64  }
65
66  public void trasladarCentro() {
67      setX(getX() + 5);
68      setY(getY() + 5);
69  }
```

pero que non elimina, senón que modifica aqueles métodos que fan o mesmo que os métodos tipo *get* e *set*. Antes da reestruturación eran:

```
21  public void establecerX(int valorX) {
22      x = valorX;
23  }
24
25  public int obterX() {
26      return x;
27  }
```

e agora son:

```
21  public void establecerX(int valorX) {
22      setX(valorX);
23  }
24
25  public int obterX() {
26      return getX();
27  }
```

que non teñen ningunha utilidade nova e deberían de ser borrados.

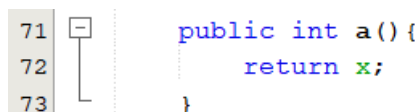
Borrar de forma segura

Débese utilizar esta opción sempre que se necesite borrar un elemento do código xa que vai permitir coñecer se existen referencias a el dentro do proxecto e cales son, antes de facer o borrado real.

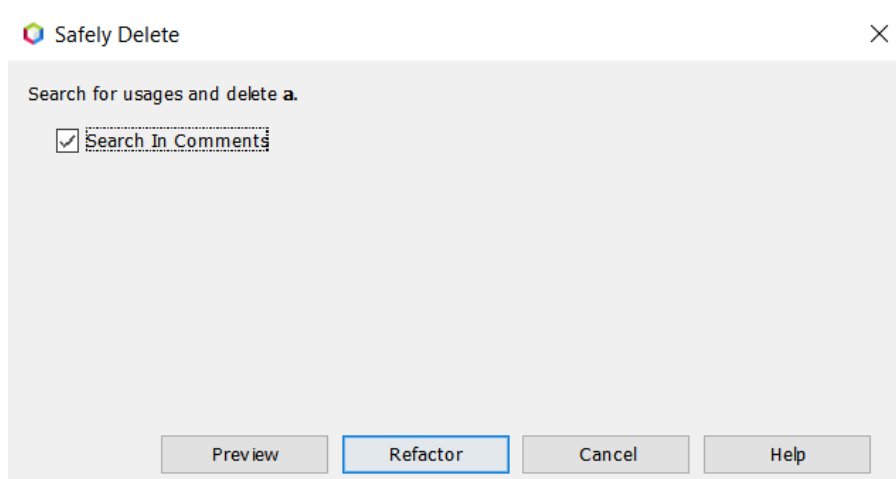
Pasos a seguir para un borrado seguro:

- Colocar o cursor no elemento de código que se quere borrar e elixir Refactor > Safely Delete. Ábrese a caixa de diálogo **Safely Delete**.
- Asegurarse de que o elemento que aparece na caixa é o que queremos borrar e realizar unha vista previa do borrado.
 - Se o elemento non está referenciado en ningún outro sitio do código, aparecerá a ventá **Refactoring** e pódese proceder a boralo.
 - Se o elemento está referenciado en algures, aparece a seguinte ventá de **Safely Delete** co aviso de que o elemento está referenciado en algún sitio e que polo tanto non se pode borrar directamente. Nesta ventá pódese:
 - cancelar a operación de borrado,
 - levala a cabo aínda que queden referencias ao elemento que se pretende borrar (non sería un borrado seguro) e
 - premer no botón **Show Usage...** para ter información detallada das referencias. Na ventá **Usage** aparece unha vista en forma de árbore cos arquivos e elementos de clases que referencian ao elemento que se pretende borrar e unha serie de iconas coma as vistas na ventá **Refactoring**. Pódense utilizar as iconas da esquerda desa ventá e pódese facer dobre clic sobre unha das ocorrencias para editar o anaco de código no que está a referencia podendo modificala para que deixe de referencialo e premer no botón **Rerun Safely Delete** para volver a iniciar o proceso de borrado ata que o elemento que se pretende borrar non estea referenciado en ningún sitio.

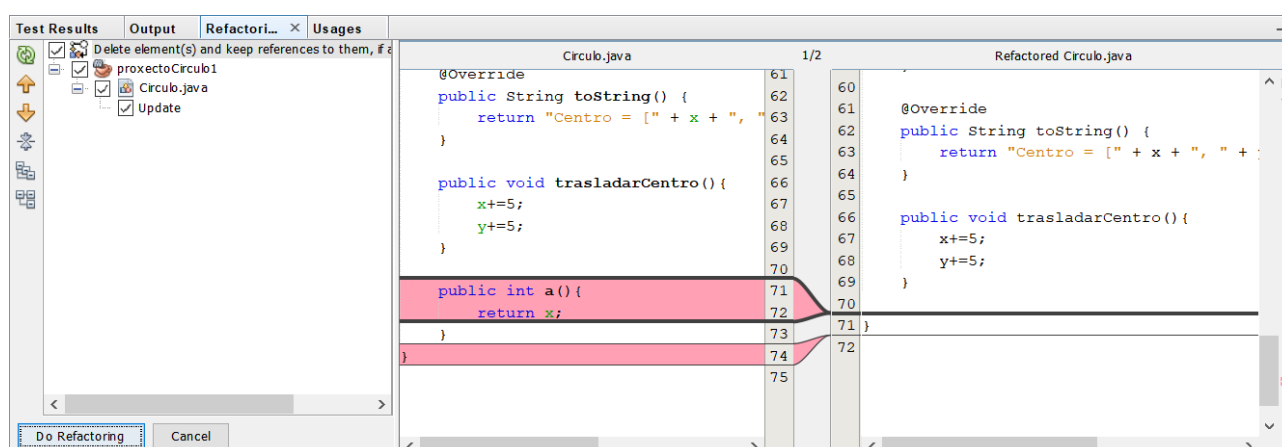
Por exemplo, borrar de forma segura o método `a` ao que nunca se chama:



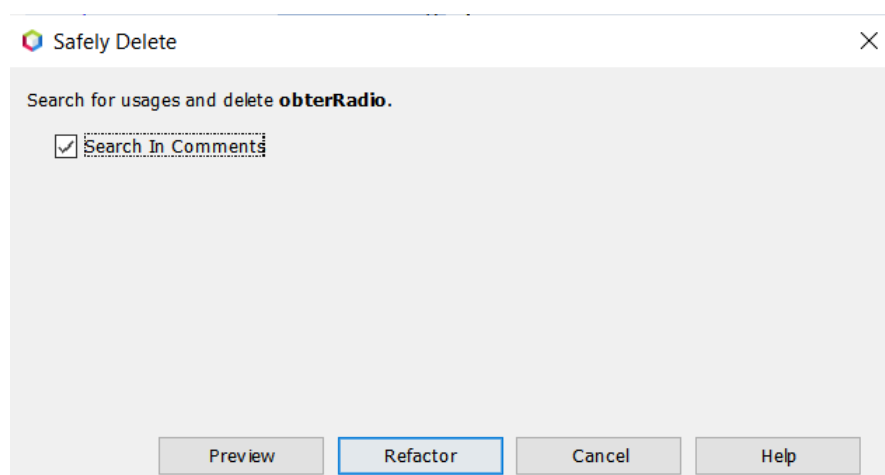
The image shows a snippet of code in a text editor. On the left, there is a vertical line with numbers 71, 72, and 73. To the right of these numbers is a small square icon with a minus sign inside. Further to the right is the code itself: `public int a() {` on line 71, `return x;` on line 72, and `}` on line 73.



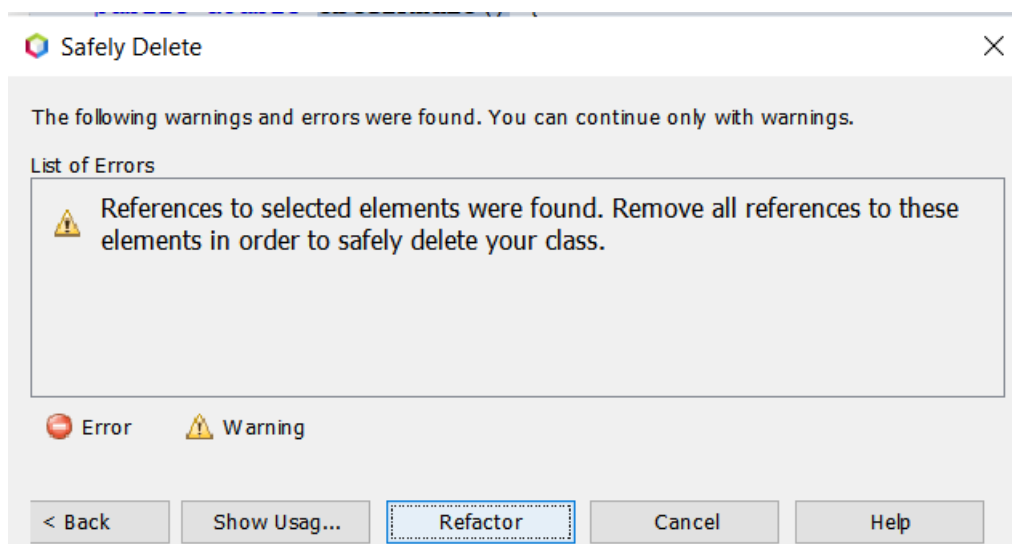
Ao premer en **Preview**, aparece a ventá de **Refactoring** xa que non hai ningún aviso de referencias ao método e vese marcado en roxo o método a borrar. Pódese realizarse a reestruturación premendo en **Refactoring**.



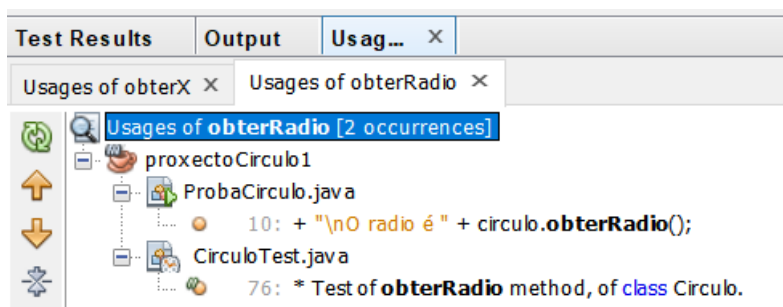
Se o método a borrar estivera referenciado noutros sitios do código como no exemplo seguinte:



Ao premer en **Preview** aparecería a seguinte caixa:



Na que se poderían facer entre outras operacións, a cancelación do borrado ou premer en **Show Usage...** para ver os detalles das referencias na ventá **Usage**:



Na ventá **Usage** indícase que hai 2 referencias: 1 no arquivo ProbaCirculo.java e 1 no arquivo CirculoTest.java. Movéndose por esas referencias coas iconas do menú da esquerda ou directamente facendo dobre clic riba dunha referencia, pódese ver e editar a liña de código fonte no que está a referencia, podendo modificala para que deixe de existir esa referencia. Despois de modificar a referencia, deberíase de premer en **Rerun Safely Delete** para volver comezar o proceso de eliminación segura ata que non existan referencias ao método obterRadio e apareza a ventá de **Refactoring**, momento no que xa se podería eliminar o método con toda seguridade.

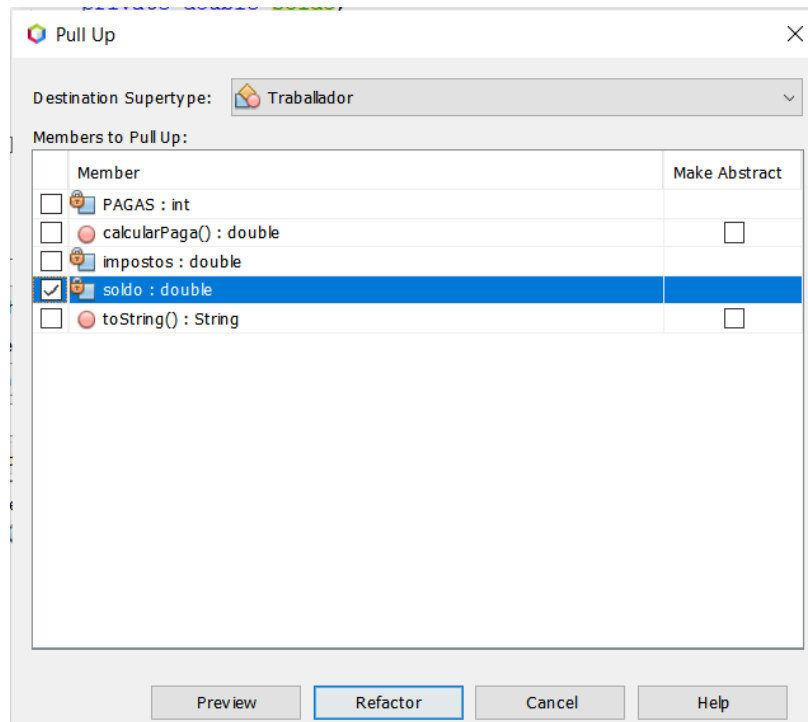
Hai que recordar que se se comete algún erro neste proceso de borrado seguro ou despois de realizar o borrado, pódese desfacer a operación con **Undo**, se non se fixo ningunha outra modificación no código.

[Mover membros dunha clase a unha superclase](#)

Pódense mover métodos e campos a unha superclase. Os pasos a seguir son:

- No código fonte ou na ventá de proxectos, hai que seleccionar a clase que contén os membros que se queren mover e elixir Refactor > Pull Up...

- Aparece a caixa de diálogo *Ascender* cunha lista de membros da clase e interfaces que a clase implementa.



- Selecciónase a clase que se quere mover.
- Selecciónanse os membros que se queren mover. Se a clase actual implementa interfaces, hai checkboxes para esas interfaces que, de marcarse, se moverían as súas implementacións á superclase.
- De querer crear un método abstracto, hai que seleccionar o checkbox correspondente para ese método e entón será declarado na superclase como un método abstracto e sobreescrito na clase actual. O método terá un acceso protected.

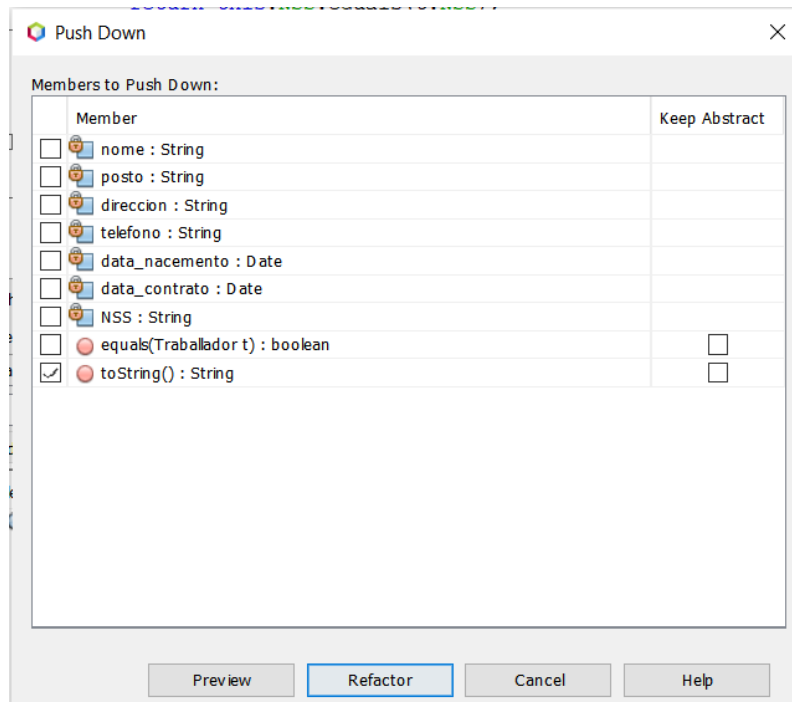
Se a clase da que se están subido membros, ten subclases e non se desexa que todos os seus elementos sexan ascendidos, deberase ter unha vista previa da reestruturación e desmarcar os checkboxes correspondentes.

[Mover membros dunha clase a unha subclase](#)

Pódense mover clases internas, métodos e campos a todas as subclases da clase actual. Os pasos a seguir son:

- No código fonte ou na ventá de proxectos, hai que seleccionar os elementos que se queren mover e elixir Refactor > Push Down.
- Aparece a caixa de diálogo **Push Down** cunha lista de membros da clase co mesmo aspecto que a caixa de diálogo **Pull Up**. Hai que marcar o checkbox dos que se queren mover.

- De existir métodos abstractos que se desexan manter definido na clase actual e ter implementados na subclase, haberá que marcar o checkbox **Keep Abstract** correspondentes. O checkbox da columna esquerda debe estar marcado para que a definición da clase sexa copiada á subclase.

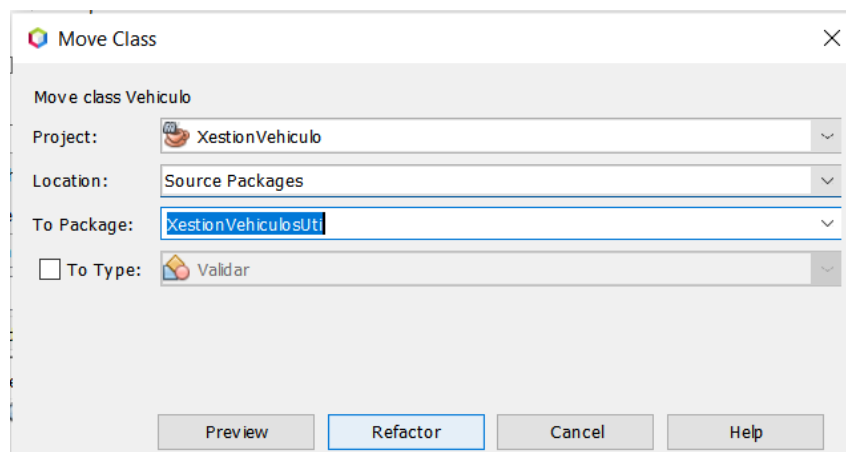


Se a clase da que se están baixando membros, ten subclases e non se desexa que todos os seus elementos sexan baixados, deberase ter unha vista previa da reestruturación e desmarcar os chekboxes correspondentes.

[Mover unha clase a outro Java Package](#)

Pasos a seguir para mover unha clase a outro paquete e cambiar o código que fai referencia a esa clase:

- No código fonte ou na ventá de proxectos, sobre a clase a mover hai que facer clic co botón dereito do rato e elixir *Refactor > Move...*
- Aparece a caixa de diálogo *Move Class*.



Esta caixa tamén se visualiza despois de cortar e pegar arquivos Java na ventá de proxectos ou na ventá de arquivos ou despois de arrastrar e soltar arquivos nas mesmas ventás. Ten os campos:

Project: co nome do proxecto que contén as clases a mover.

Location: a parte do proxecto que contén as clases a mover. Normalmente *Source Packages*.

To Package: nome do paquete ao que se queren mover as clases (recoméndase esta opción) ou nome completo do paquete como por exemplo: com.myCom.myPkg.

Se a clase que se está movendo, ten subclases e non se desexa que todos os seus elementos sexan movidos, deberase ter unha vista previa da reestruturación e desmarcar os checkboxes correspondentes.

Non se recomenda mover unha clase a outro paquete sen utilizar a reestruturación aínda que é posible realizando os pasos seguintes:

- Mover manualmente a clase a outro paquete dende a ventá de proxectos facendo cortar e pegar ou arrastrar e soltar.
- Aparece a caixa de diálogo **Move Class** e selecciónase o checkbox **Move without refactoring**.

Copiar clase

Pasos a seguir para copiar unha clase dentro do mesmo paquete ou noutro paquete e cambiar o código que referencie a esa clase:

- No código fonte ou na ventá de proxectos, sobre a clase, hai que facer clic co botón dereito do rato e elixir Refactor > Copy.
- Aparece a caixa de diálogo **Copy class** que é similar en visualización e compoñentes á caixa **Move class**. Nesta caixa selecciónase o paquete ao que se quere copiar (recoméndase esta opción) ou tecléase o nome completo como por exemplo com.myCom.myPkg.

Se a clase que se está copiando, ten subclases e non se desexa que todos os seus elementos sexan copiados, deberase ter unha vista previa da reestruturación e desmarcar os checkboxes correspondentes.

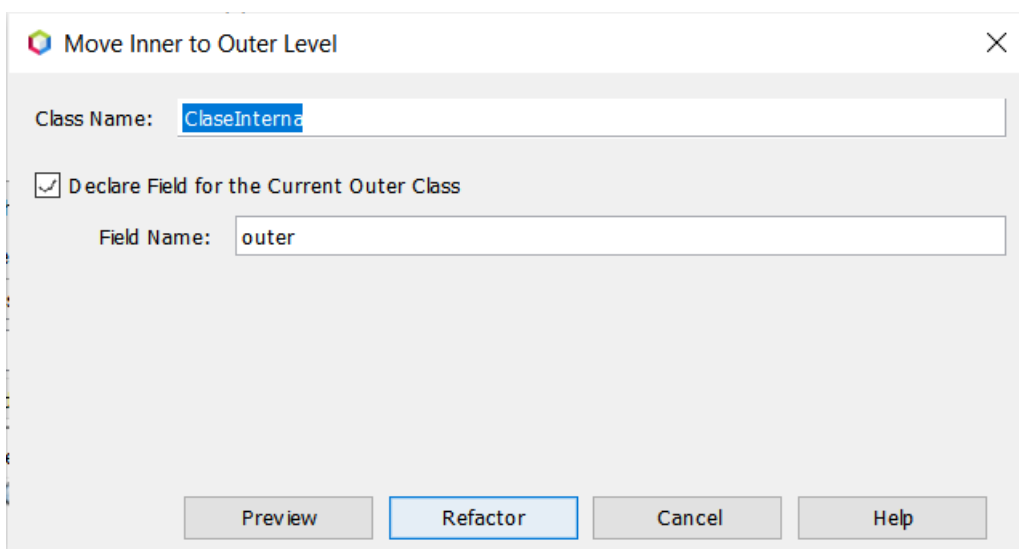
Non se recomenda copiar a clase sen utilizar a reestruturación aínda que é posible:

- Copiar manualmente a clase a outro paquete ou ao mesmo dende a ventá de proxectos facendo cortar e pegar ou arrastrar e soltar.
- Aparece a caixa de diálogo **Copy class** e selecciónase o checkbox **Copy without refactoring**.

Mover unha clase de nivel interior a exterior

Pódese mover unha clase interna a un nivel superior na xerarquía de clases. Por exemplo, se a clase seleccionada está aniñada dentro dunha clase superior, créase a clase seleccionada nese nivel superior. Se a clase seleccionada está aniñada nunha clase interna, a clase seleccionada móvese ao nivel da clase interna. Pasos a seguir:

- No código fonte e sobre a clase interna que se quere mover, hai que facer clic co botón dereito do rato e elixir Refactor → **Move inner to outer level**.
- Aparece a caixa de diálogo **Move Inner to Outer Level**. Nesta caixa:
 - pódese cambiar o nome da nova clase
 - opcionalmente pódese crear un campo na nova clase que fará referencia a un obxecto da clase envolvente orixinal e darlle nome a ese campo.



Converter anónimo en membro

Pódese converter unha clase anónima en clase interna que contén nome e construtor, creando unha nova clase interna e reemplazando a clase interna anónima por unha chamada á nova clase interna creada. Para iso, colócase o cursor na clase anónima, prémese **Alt+Enter** e elíxese **Convert Anonymous to Member...**

Extraer interface

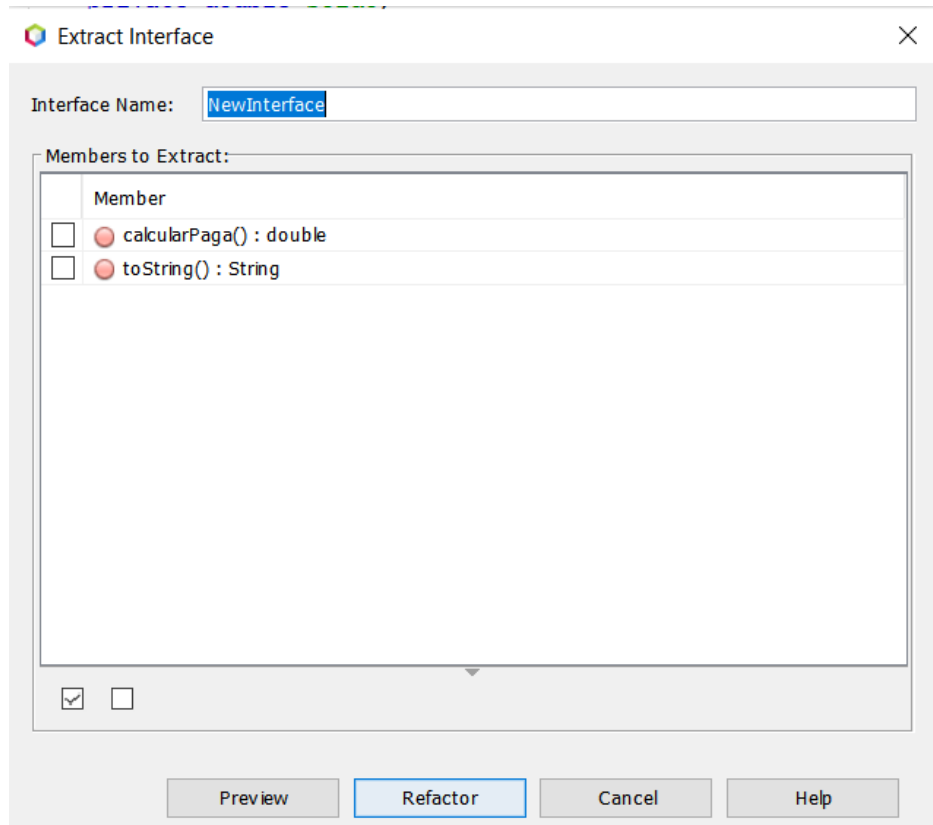
Permite seleccionar os métodos públicos non estáticos dunha clase ou interface, que irán a parar a unha nova interface. Unha interface non restrinxe como son implementados os seus métodos, as interfaces poden ser utilizadas en clases que teñen funcións diferentes. Crear interfaces pode aumentar a reutilización do código.

Cando se extrae unha interface, o IDE fai o seguinte:

- Crea unha interface nova cos métodos seleccionados no mesmo paquete da clase actual ou interface.
- Actualiza, aplica ou estende a clase actual ou interface para incluír a interface nova.

Para extraer unha interface:

- Abrir a clase ou a interface que contén os métodos que se queren mover a unha interface.
- Clic na opción de menú Refactor → **Extract Interface**.
- Aparece a caixa de diálogo de **Extract Interface**.



- Escribir o nome para a nova interface no campo de texto.
- Escoller os membros que se queiran extraer á interface nova.

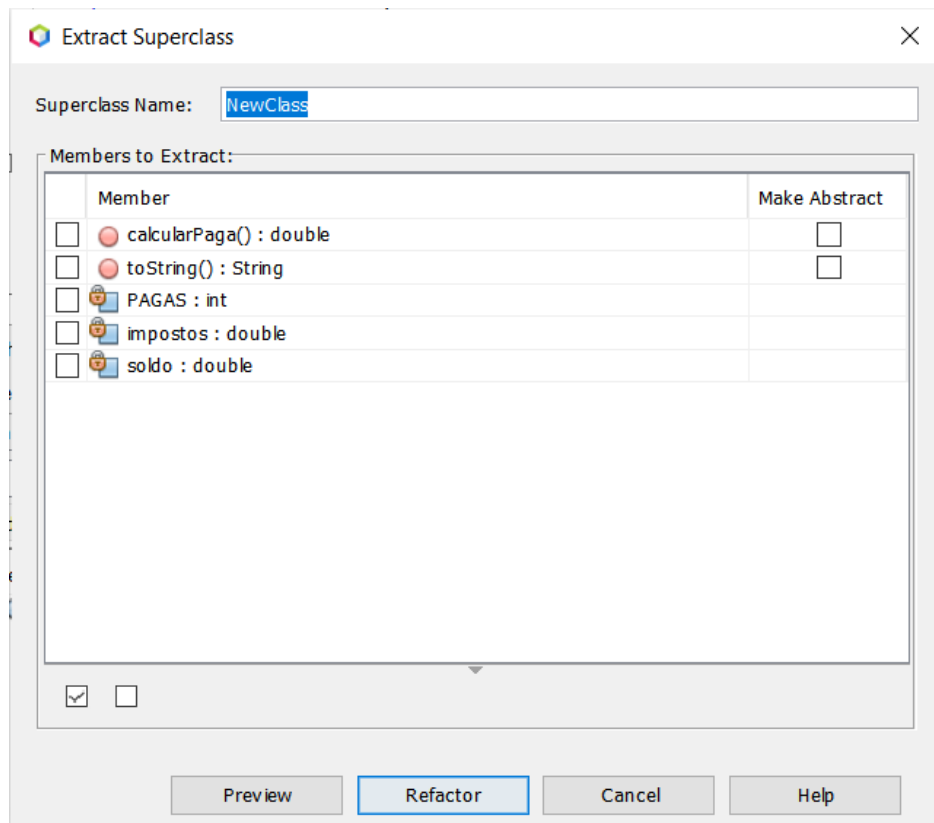
Extraer superclase

Cando se extrae unha superclase, o IDE fai o seguinte:

- Crea unha clase nova cos métodos e campos seleccionados na clase seleccionada. Tamén pode aplicar interfaces a clase nova que son aplicadas na clase seleccionada.
- Se a clase seleccionada estende unha clase, a clase nova tamén estende a mesma clase.
- A clase seleccionada é modificada de modo que estende a nova superclase.
- Move os campos públicos ou protexidos seleccionados á nova superclase.
- Fai unha previsualización de como quedarán as clase despois de extraer a superclase.

Para extraer unha superclase:

- Abrir a clase que contén os métodos ou campos que se queren mover á nova superclase.
- Clic na opción de menú *Refactor* → **Extract superclass**.
- Ábrese o cadro de diálogo de **Extract superclass** cos métodos e campos que poden extraerse.



- Escribir o nome para a nova superclase na caixa de texto.
- Seleccionar os membros que se queren extraer á nova superclase.
- (Opcional) Pódese facer un método abstracto, seleccionar checkbox **Make Abstract** para o método. Se se selecciona este checkbox, o método será declarado na superclase como un método abstracto e sobreescrito na clase actual.

Despois de refactorizar

Despois de realizar unha reestruturación, sobre todo se implica varias operacións complicadas, débese de limpar e construír de novo o proxecto. Para iso, co cursor sobre o nome do proxecto na ventá de proxectos, débese de facer clic co botón dereito do rato e elixir **Clean and Build**.