CONTORNOS DE DESENVOLVEMENTO

Tema 4: Optimización e Documentación

C.S. Desenvolvemento de Aplicacións Multimedia
Módulo: Contornos de Desenvolvemento -

XUNTA DE GALICIA
CONSELLERÍA DE EDUCACIÓN
E ORDENACIÓN UNIVERSITARIA

Índice

1. Refactorización 3

1.1 Introdución 3
1.2 Patrón de deseño 3
1.3 Catálogo de patróns de deseño 4
1.4 Refactorización 5
1.5 Reestruturar código Java con NetBeans 9

2. Control de versións 35

2.1 Definición 35
2.2 Características 35
2.3 Operacións básicas 36
2.4 Clasificación 38

3. Git 41

3.1 Descrición 41
3.2 Instalación Git en Windows 43
3.3 Creando o repositorio 44
3.4 Operacións básicas 45
3.5 Traballando con ramas 51
3.6 Resolución de erros e conflitos 53
3.7 Repositorios remotos 55
3.8 Outros comandos 66
3.9 Control de versións en NetBeans con Git 68

5. Documentar clases 75

5.1 Agregar documentación Javadoc a NetBeans 75
5.2 Comentarios Javadoc no código fonte 76
5.3 Xerar documentación Javadoc 83
5.4 Etiquetas e anotacións 91
5.5 Exemplos 94

Traballo derivado por: Patricia González Pardo
con Licenza CC: BY-NC-SA (*) a partir dos documentos orixinais:

© Xunta de Galicia. Consellería de Cultura, Educación e Ordenación Universitaria.
Autores: María del Carmen Fernández Lameiro, Máximo Fernández López Andrés del Río Rodríguez.
Licenza CC: BY-NC-SA (*)

Excepto capítulo 4: Git: Autor: Fernando Rodríguez Diéguez con Licenza Creative Commons BY-NC-SA (*)

(*) Licenza CC: BY-NC-SA: Creative Commons Recoñecemento - Non Comercial - Compartir Igual. Para ver unha copia desta licenza, visitar a ligazón http://creativecommons.org/licenses/by-nc-sa/3.0/es/

1. Refactorización

1.1 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 reusable. Un exemplo é a técnica de programación orientada a obxectos.

Os expertos viron que determinados problemas de deseño e código repítense 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.

1.2 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 só 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 fixeron e solucionaron.
- Crear catálogos de patróns.
- Crear un vocabulario común entre deseñadores.
- Estandarizar o deseño pero non impor un modelo nin eliminar 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 específico.
- Antipatróns de deseño que con forma parecida aos patróns, reúnen solucións que se sabe que producen efectos negativos. Provén de refactorizaciones para transformar as solucións negativas en positivas.
- Patróns de interacción centrados no deseño de interfaces web.

1.3 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 Vlisides, 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

Na seguinte web tamén se poden consultar patróns de deseño en diferentes liguaxes.

https://refactoring.guru/es

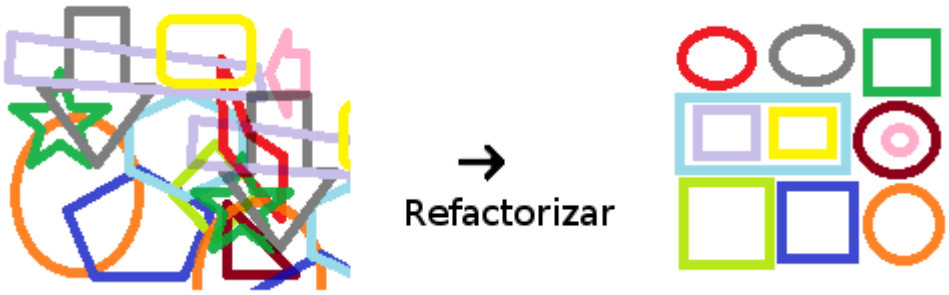
1.4 Refactorización

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:

- Interveñen diferentes programadores.
- Incorpóranse novas funcionalidades. Aínda se complica máis se para incorporar novas funcionalidades, incorpóranse anacos adaptados do código doutros programas.

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



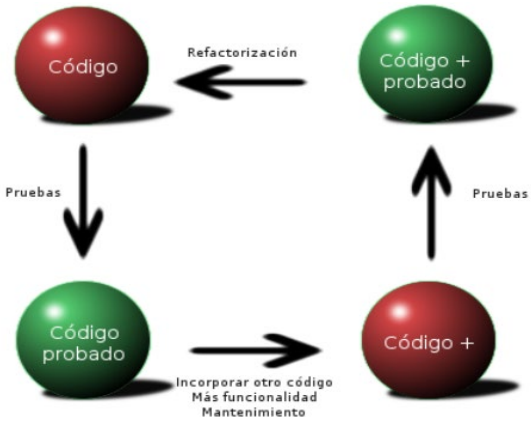
A refactorización é a transformación controlada de código fonte que asegura a reorganización do código para mellorar o seu mantemento e/ou facelo máis comprensible ou para saír dalgún antipatrón.

- Non altera o seu comportamento externo pero mellora o seu estruturación 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.

1. Recoméndase realizar os cambios de forma progresiva, un a un e paso a paso.
2. Repetir a proba automática obtendo os mesmos resultados que antes de refactorizar.
3. 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.
- 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 uno a un se é posible. Executar probas.

As probas unitarias permiten comprobar que o código funciona correctamente, pero 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 as probas se realizan de forma automática xa que entón disponse 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

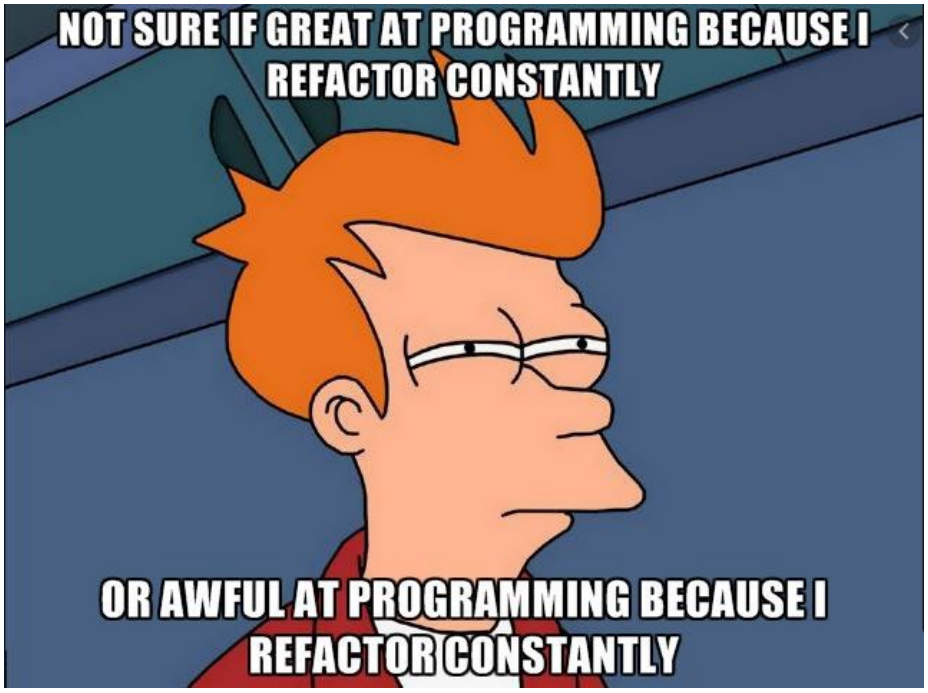
Tanto Fowler como Piattini e García consideran que a refactorización aumenta a produtividade xa que:

- 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 mais clara cales son as súas funcións, e que sexa o mais 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 convértese 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 refactorizan. 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 o 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 cuxos compoñentes son máis requiridos para usar.
- 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 as súas superclases lles provén por herdanza, 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 funcionalidade" pode parecer contraditorio co patrón "visitante" cuxa aplicación 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 garanten que o código segue tendo a mesma funcionalidade despois da refactorización.
- Cando se necesita 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, ...

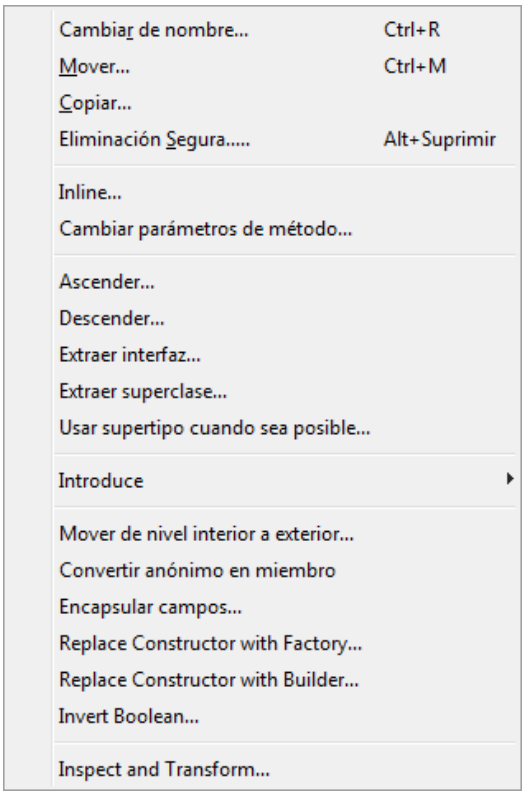
1.5 Reestruturar código Java con NetBeans

Introdución

NetBeans permite reestruturar código Java sen cambiar o comportamento do programa. A reestruturación 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 ditos 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 *Reestruturar*

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

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 ou	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

método	declaración. A declaración pode ser a creación dunha variable, constante, campo ou método
Cambiar parámetros dun método	Engade, elimina, modifica ou cambia a orde dos parámetros dun método ou cambia o modificador de acceso (*public, private, protected)*
Encapsular campos	Xera métodos get e set para un campo e opcionalmente actualiza tódalas referencias a ese campo utilizando os métodos get e set
Ascender	Move métodos e campos a unha superclase da que herdaría 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: Crea unha nova clase abstracta que conterá ditos 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 opción *Undo* activarase despois de facer unha reestruturación e aparecerá como *Undo [nome da reestruturación]* permitindo desfacer tódolos cambios en tódolos arquivos afectados pola reestruturación. A opción *Redo* activarase despois de desfacer unha reestruturación e aparecerá como *Redo[nome da reestruturación]* servindo para volver a facer a reestruturación.

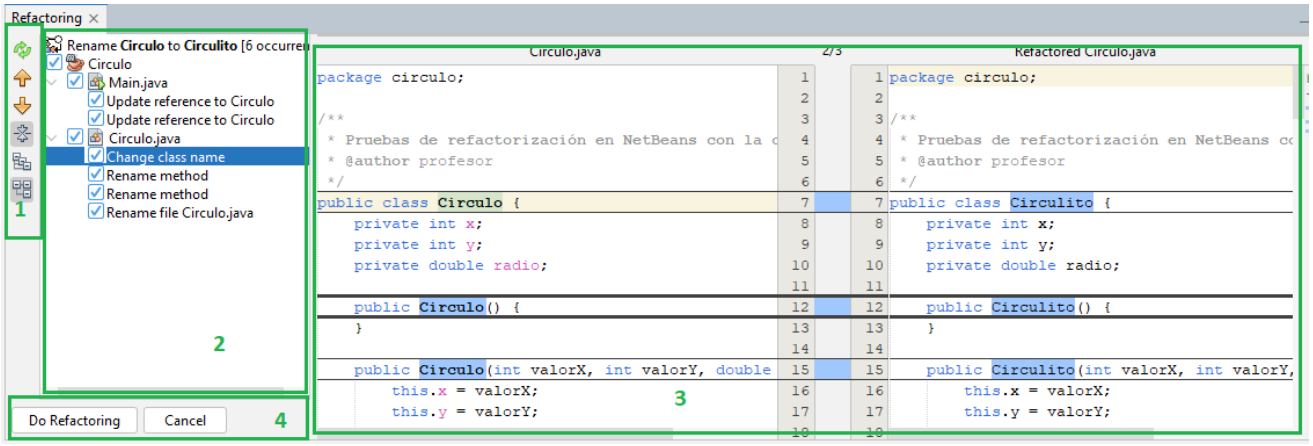
Estas opcións non terán efecto se algún dos arquivos afectados foi modificado dende que a reestruturación tivo lugar ou cando a reestruturación se pode desfacer simplemente coas iconas:

Ventá Reestruturando

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 Reestruturar→*Cambiar nombre*, teclear o novo nome da clase e premer no botón de Preview.

A ventá Refactoring está dividida en varias zonas:

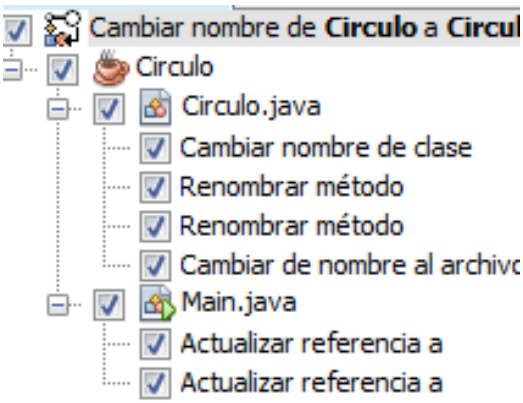


- Zona 1, menú de iconas para actuar sobre a vista previa da reestruturación que permite:

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:

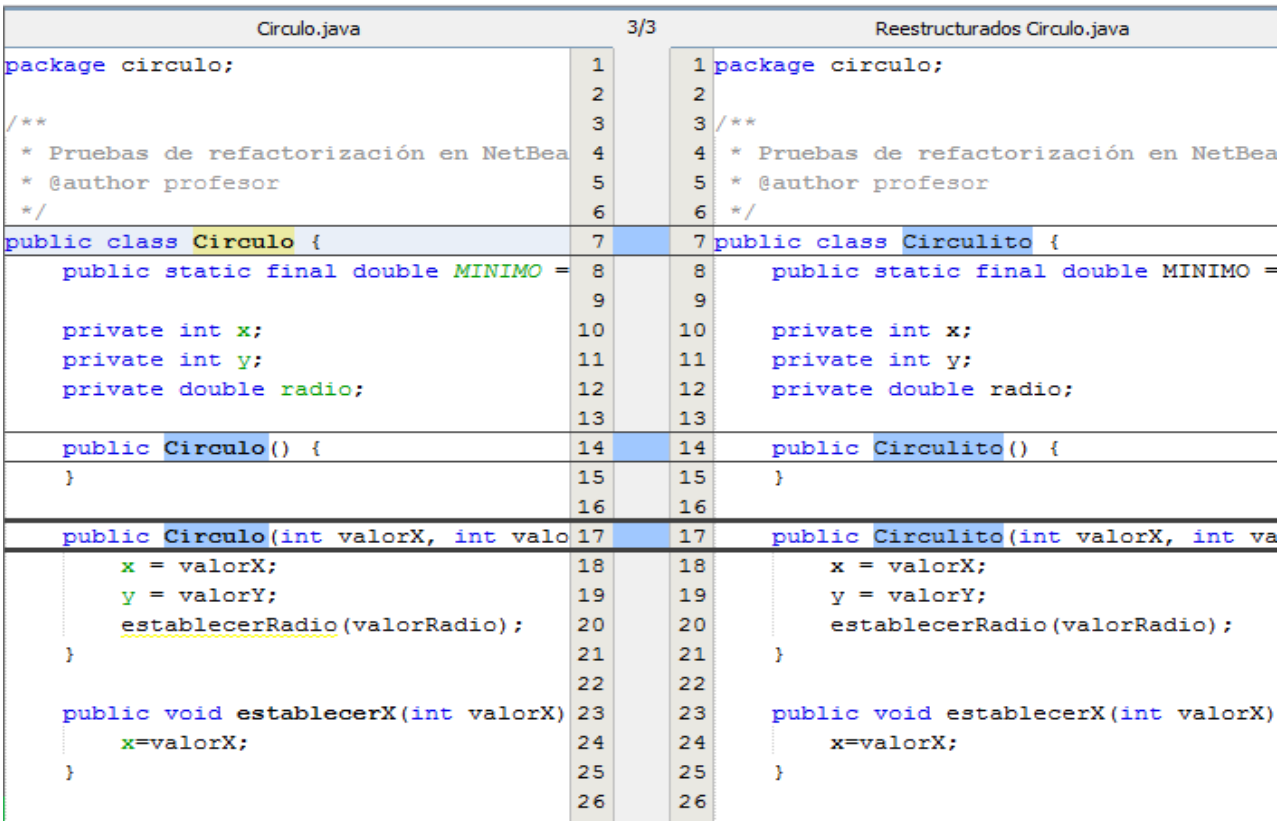
  - ✓ marcar ou desmarcar unha ocorrencia para incluíla ou excluíla da reestruturación,
  - ✓ expandir ou contraer cada nodo da árbore ou
  - ✓ facer dobre clic riba dun nodo para que sexa a ocorrencia actual e se mostre na zona 3.



- 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:

1. Numeradas as liñas de código.
2. Os cambios resaltados.
3. Tódalas liñas de código afectadas polos cambios aparecen remarcadas entre liñas paralelas finas que conectan os dous arquivos.
4. A ocorrencia actual remarcada entre liñas paralelas grosas que conectan os dous arquivos.



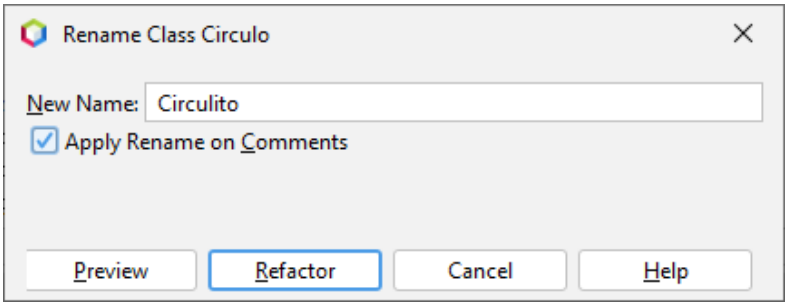
- Zona 4: Contén os botóns que permiten levar a cabo a reestruturación ou cancelar a mesma.

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 Reestruturar→*Cambiar nome*... Aparece unha caixa que permite:

1. Teclear o nome novo
2. Opcionalmente cambiar o nome nos comentarios
3. Ter unha vista previa dos cambios
4. Realizar a reestruturación
5. Cancelar a operación de reestruturación
6. Recorrer á axuda en liña de NetBeans



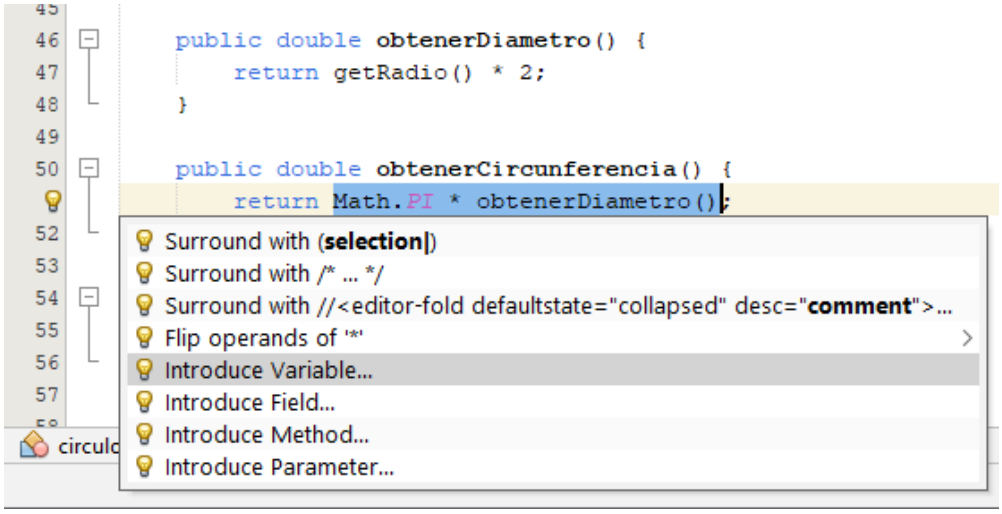
Cambiará tódalas aparicións ou referencias á clase Circulo por Circulito no proxecto e cambiará o nome de Circulo.java por Circulito.java.

Encerrar con... Introducir...

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 có 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 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.

```
36
37    public void establecerRadio(double valorRadio) {
38
        setRadio(valorRadio < 0.0 ? 0.0 : valorRadio);
40
41
42
43
44
45
46
47
```

- Surround with (**selection**)
- Surround with /* ... */
- Surround with //<editor-fold defaultstate="collapsed" desc="**comment**">...
- Introduce Variable...
- Introduce Constant...
- Introduce Field...
- Introduce Method...
- Introduce Parameter...

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

```
9   public class Main {
10      public static void main(String[] args) {
11          Circulo circulo = new Circulo(valorX: 37,valorY: 43,valorRadio: 2.5);
12          String salida =
13                  "La coordenada X es "+circulo.obtenerX()+
14                  "\nLa coordenada Y es "+circulo.obtenerY()+
15                  "\nEl radio es "+circulo.obtenerRadio();
16          circulo.establecerX(valorX: 35);
17          circulo.establecerY(valorY: 20);
18          circulo.establecerRadio(valorRadio: 4.2);
19          DecimalFormat dosDigitos = new DecimalFormat(pattern:"0.00");
20          salida+="\nEl diámetro es "+dosDigitos.format(number: circulo.obtenerDiametro());
21          salida+="\nLa circunferencia es "+dosDigitos.format(number: circulo.obtenerCircunferencia());
22          salida+="\nEl área es "+dosDigitos.format(number: circulo.obtenerArea());
            System.out.println(x: salida);
24
25
26
27
```

- Surround with /* ... */
- Surround with //<editor-fold defaultstate="collapsed" desc="**comment**">...
- Surround with **Runnable r** = new **Runnable**() {...
- Surround with do { ...
- Surround with for (Map.Entry<KeyType, ValueType> **entry** : **map**.entrySet()) { ...
- Surround with for (**Iterator it** = **col**.iterator(); **it**.hasNext();) {...
- Surround with for (**StringTokenizer TOKENIZER** = new **StringTokenizer**(**STRING**); **TOKENIZER**.hasMoreTokens();) {...
- Surround with for (**Object elem** : **col**) {...
- Surround with for (int **i** = 0; **i** < **10**; **i**++) {...
- Surround with for (int **idx** = 0; **idx** < **arr**.length; **idx**++) {...
- Surround with for (int **idx** = 0; **idx** < **lst**.size(); **idx**++) {...
- Surround with for (int **idx** = 0; **idx** < **vct**.size(); **idx**++) {...
- Surround with if (**exp**) { ...| }
- Surround with if (**exp**) { ...| } else { ... }
- Surround with switch (**exp**) { ...| }
- Surround with synchronized (**this**) {...
- Surround with try { ...| } catch(Exception e) { ... }
- Surround with try { ...| } catch(Exception e) { ... } finally { ... }
- Surround with try { ...| } finally { ... }
- Surround with while (**en**.hasMoreElements()) {...
- Surround with while (**true**) { ...
- Surround with while (**it**.hasNext()) {...
- Introduce Method...

circu

= " + ge

Introducir variable, constante, campo 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

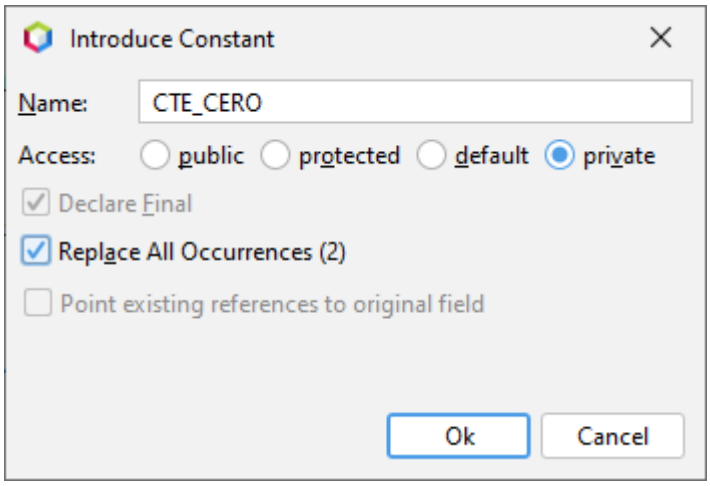
máis significativos e así facilitar as modificacións futuras, a reusabilidade do código e mellorar a comprensión do código.

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 *Reestruturar->Introducir variable...* ou *constante* ou *campo* 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 tódalas coincidencias ou só a que está seleccionada.

Considerando seleccionado o valor 0.0 do método establecerRadio():



Cambiaranse as dúas coincidencias polo nome da constante e aparece a declaración da constante na clase.

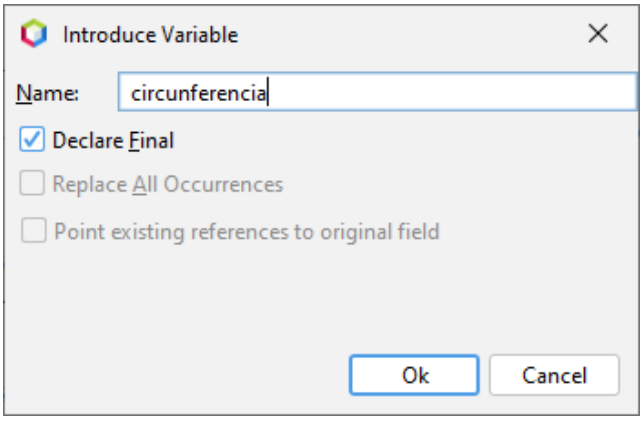
```
public void establecerRadio(double valorRadio) {

    setRadio(valorRadio < CTE_CERO ? CTE_CERO : valorRadio);
}
private static final double CTE_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 tódalas coincidencias ou só na selección actual.

Considerando seleccionada a expresión *Math.PI * obtenerDiametro()* do módulo *obtenerCircunferencia()*:



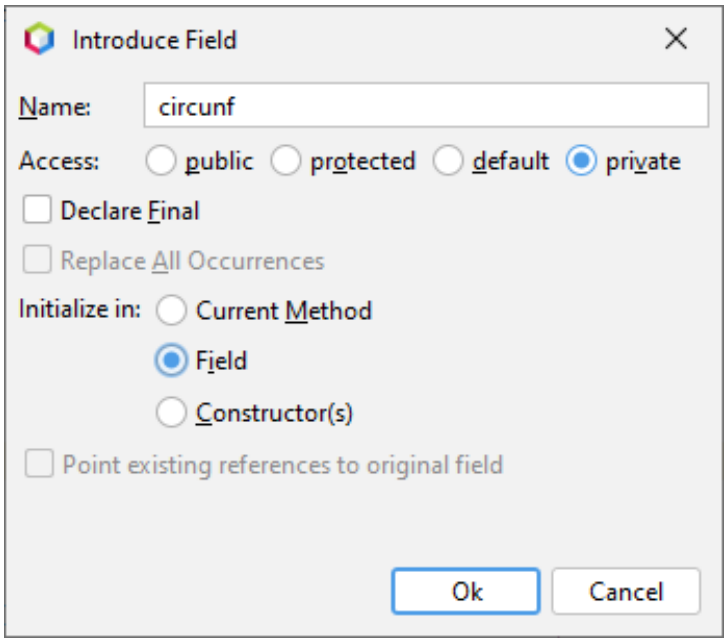
Aparecerá substituída a expresión por esa variable dentro do método actual:

```
53      public double obtenerCircunferencia() {
54          circunf = Math.PI * obtenerDiametro();
55          final double circunferencia = circunf;
56          return circunferencia;
57      }
58      private double circunf;
```

Introducir propiedade (Field)

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 * obtenerDiametro()* do método *obtenerCircunferencia()*:

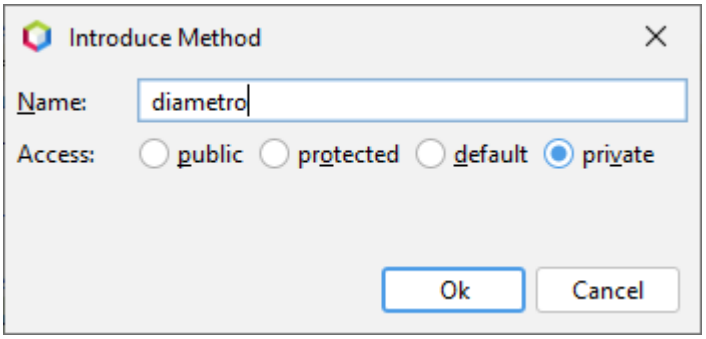


Aparecerá ese novo campo declarado na clase

```
public double obtenerCircunferencia() {
    return circunf;
}
private double circunf = Math.PI * obtenerDiametro();
```

Introducir método

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



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

```
public double obtenerCircunferencia() {
    circunf = diametro();
    final double circunferencia = circunf;
    return circunferencia;
}

private double diametro() {
    return Math.PI * obtenerDiametro();
}
```

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:

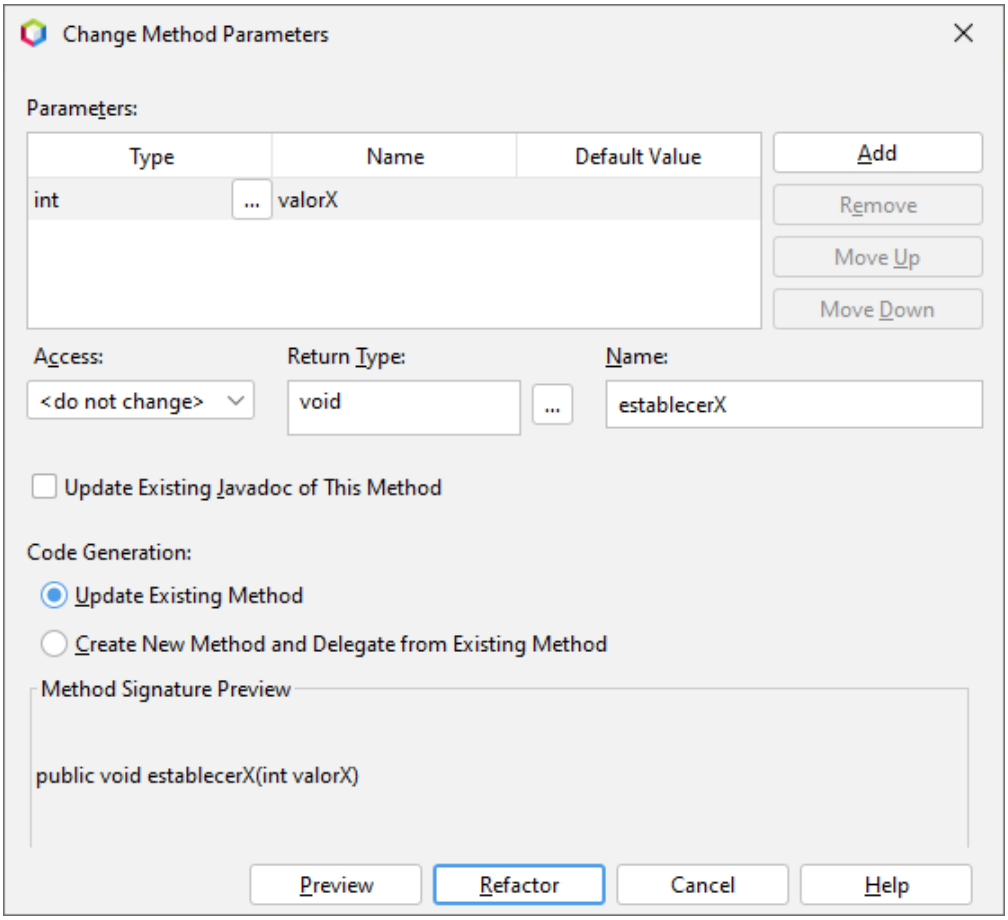
- A selección non pode ter máis dun parámetro de saída.
- A selección non pode ter instrucións break nin continue se a seguinte operación a facer non está dentro da selección.
- A selección non pode conter unha sentencia return que non sexa a última da selección.
- A selección non pode ter un return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

de método.... Ábrese entón a ventá de *Cambiar parámetros de método* 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.



- Engadir parámetros

  Na ventá Cambiar parámetros de método prémese o botón de Agregar 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 Cambiar parámetros de método, selecciónase o parámetro que se quere mover e prémese nos botóns Subir ou Desplazar hacia abajo segundo se queira.

- Cambiar tipo de acceso

  Na ventá Cambiar parámetros de método, 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.

```
63      public void trasladarCentro(){
64          x=x + 5;
65          y=y + 5;
66      }
```

Que é utilizado en Main.java, engadindo as seguintes liñas:

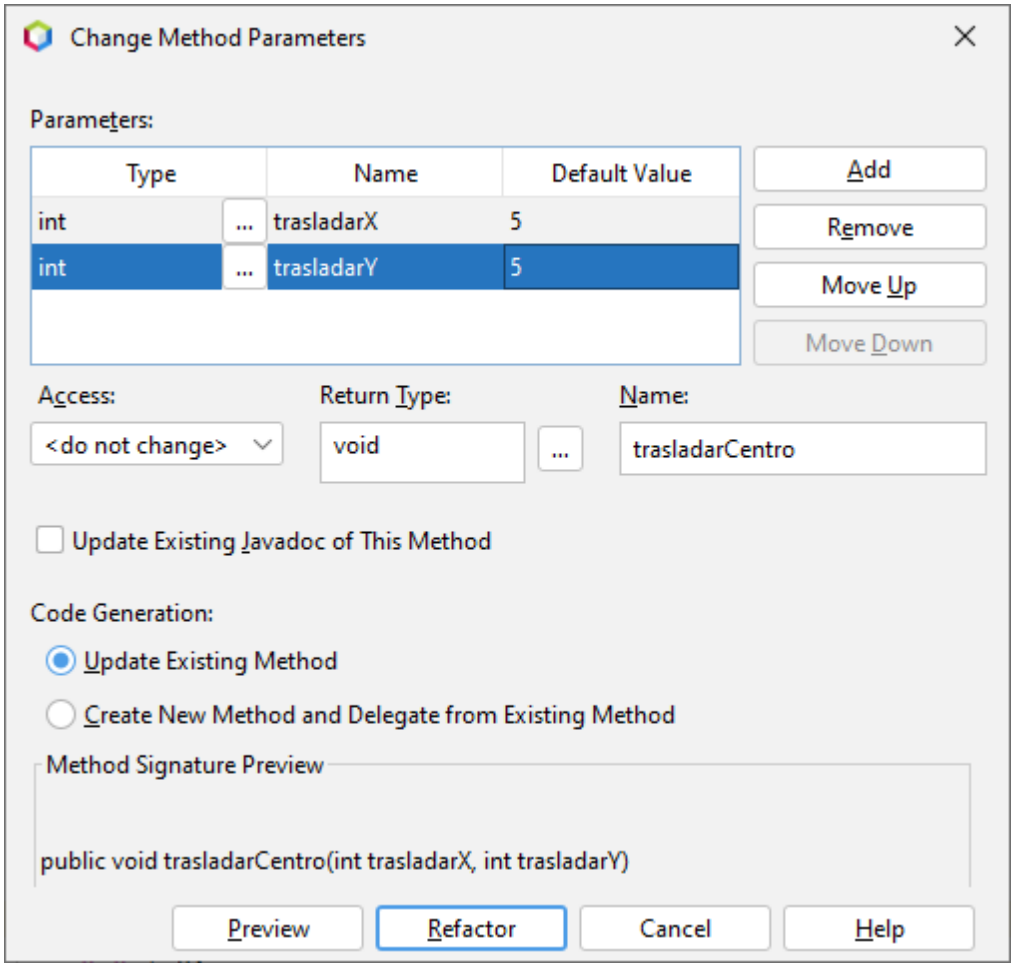
Para

```
System.out.println(x: salida);
circulo.trasladarCentro();
salida = "\n\nLa nueva ubicación y el radio de círculo son \n " + circulo.toString();
System.out.println(x: salida);
```

E en CirculoTest.java:

```
159     @Test
        public void testTrasladarCentro() {
161         System.out.println(x: "trasladarCentro");
162         Circulo instance = new Circulo();
163         int resultx = instance.obtenerX();
164         int resulty = instance.obtenerY();
165         instance.trasladarCentro();
166         int resultnx = instance.obtenerX();
167         int resultny = instance.obtenerY();
168         assertEquals(resultx + 5, actual: resultnx);
169         assertEquals(resulty + 5, actual: resultny);
170
171     }
```

Para iso, teño que engadirlle parámetros ó método



Engadiranse os parámetros *trasladarY* e *trasladarX* de tipo int e con valor 5 por defecto, quedando os códigos modificados como:

```
        System.out.println(x: salida);
        circulo.trasladarCentro(trasladarX: 5, trasladarY: 5);
        salida = "\n\nLa nueva ubicación y el radio de círculo son \n " + circulo.toString();
        System.out.println(x: salida);
    }
```

```
    */
    @Test
    public void testTrasladarCentro() {
        System.out.println(x: "trasladarCentro");
        Circulo instance = new Circulo();
        int resultx = instance.obtenerX();
        int resulty = instance.obtenerY();
        instance.trasladarCentro(trasladarX: 5, trasladarY: 5);
        int resultnx = instance.obtenerX();
        int resultny = instance.obtenerY();
        assertEquals(resultx + 5, actual: resultnx);
        assertEquals(resulty + 5, actual: resultny);

    }
```

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

```
X+=5;
y+=5;
```

por:

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

deberíase de facer manualmente.

Encapsular campos

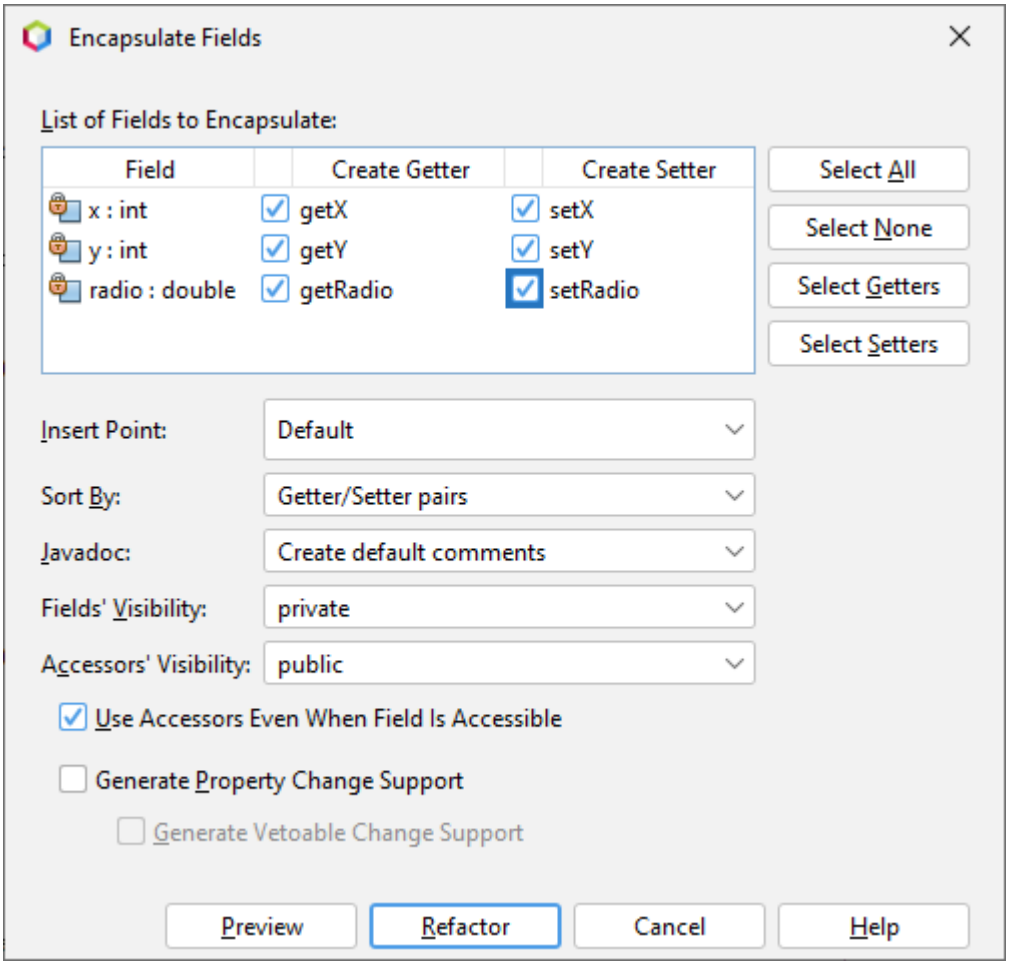
Encapsular un campo consiste en facer que o campo teña acceso *private* e só sexa accesible utilizando un par de métodos de tipo *get* e *set* que serán públicos.

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.

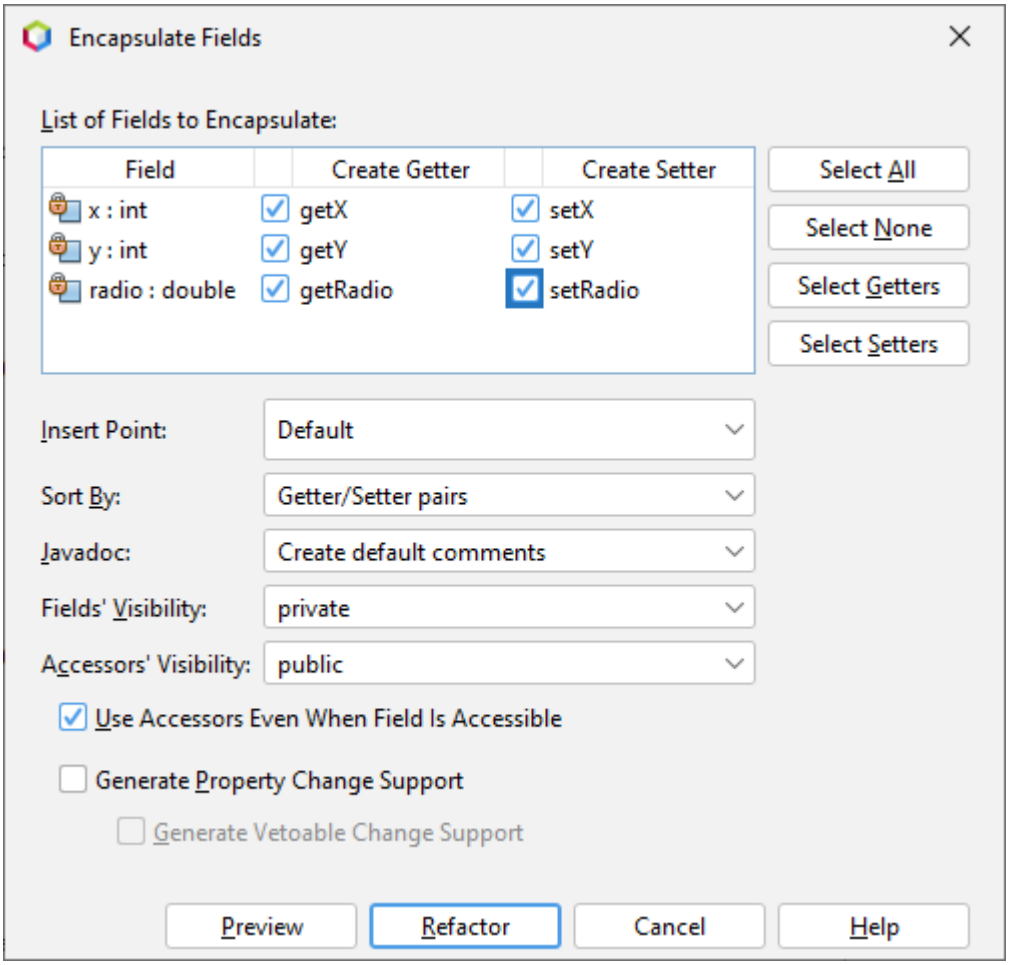
Para encapsular un campo, débese facer clic co botón dereito do rato sobre o campo ou unha referencia ao campo e elixir *Reestructurar→Encapsular campos...* e ábrese a caixa de diálogo *Encapsulsar campos* .



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 st.
- 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 tódolos 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 Usar métodos de acceso ainda que o campo sexa accesible, se non se quere utilizar os novos métodos de acceso se é que o campo xa está accesible.

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



Que dará como resultado:

```java
*/
public class Circulo {

    private int x;
    private int y;
    private double radio;

    public Circulo() {
    }

    public Circulo(int valorX, int valorY, double valorRadio) {
        x = valorX;
        y = valorY;
        establecerRadio(valorRadio);
    }
    /**
     * @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:

```java
@Override
public String toString() {
    return "Centro = [" + getX() + "," + getY() + "]; Radio = " + getRadio()
}

public void trasladarCentro(int trasladarX, int trasladarY) {
    setX(getX() + trasladarX);
    setY(getY() + trasladarY);

}
```

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

```
public void establecerX(int valorX) {
    setX(x: valorX);
}

public int obtenerX() {
    return getX();
}

public void establecerY(int valorY) {
    setY(y: valorY);
}

public int obtenerY() {
    return getY();
}
```

e agora son:

```
public void establecerX(int valorX) {
    setX(x: valorX);
}

public int obtenerX() {
    return getX();
}

public void establecerY(int valorY) {
    setY(y: valorY);
}

public int obtenerY() {
    return getY();
}
```

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.

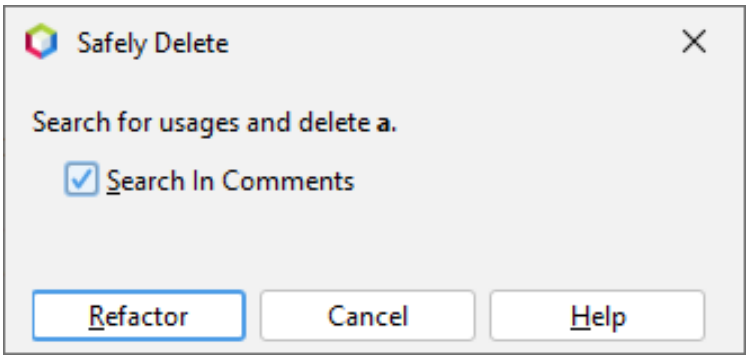
Pasos a seguir para un borrado seguro:

- Colocar o cursor no elemento do código que se quere borrar e elixir Refactor->Safety Delete. Ábrese a caixa de diálogo Safety Delete.
- Asegurarse de que o elemento que aparece na caixa é o que queremos borrar e realizar unha vista previa do borrado.

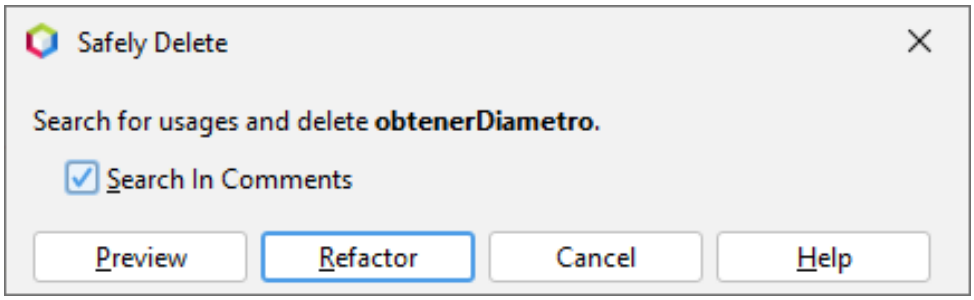
- Se o elemento non está referenciado en ningún outro sitio do código, aparecerá a ventá Reestruturando e pódese proceder a borralo.
- Se o elemento está referenciado en algures, aparece a seguinte ventá de Eliminación segura co aviso de que o elemento está referenciado en algún sitio e que polo tanto 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 *Mostrar uso...* para ter información detallada das referencias. Na ventá *Usos* 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 *Reestruturando*. 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 Executar *de novo Eliminación segura* 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:

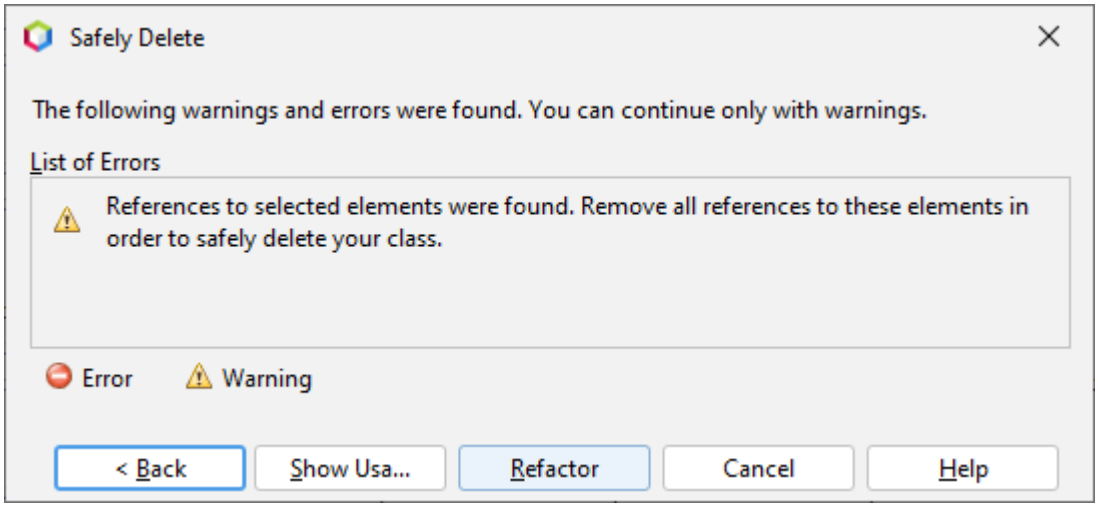
```
public int a(){
    return x;
}
```



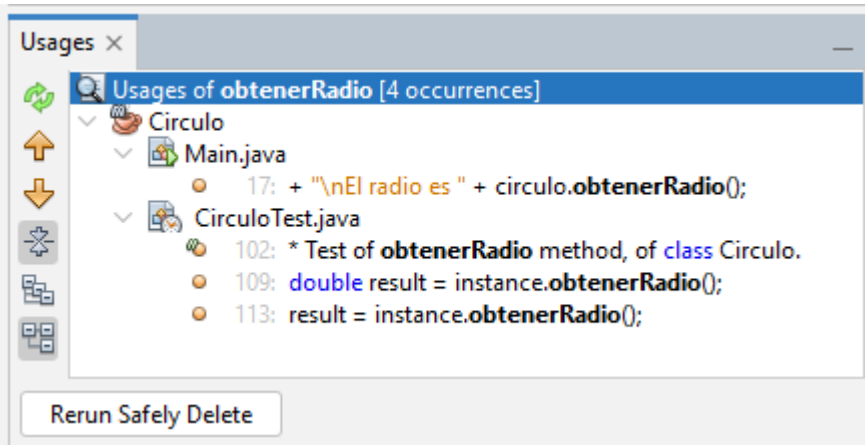
Se o método a borrar estivera referenciado noutros sitios do código como no exemplo seguinte:



Ao premer en *Vista Previa* aparecería a seguinte caixa:



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



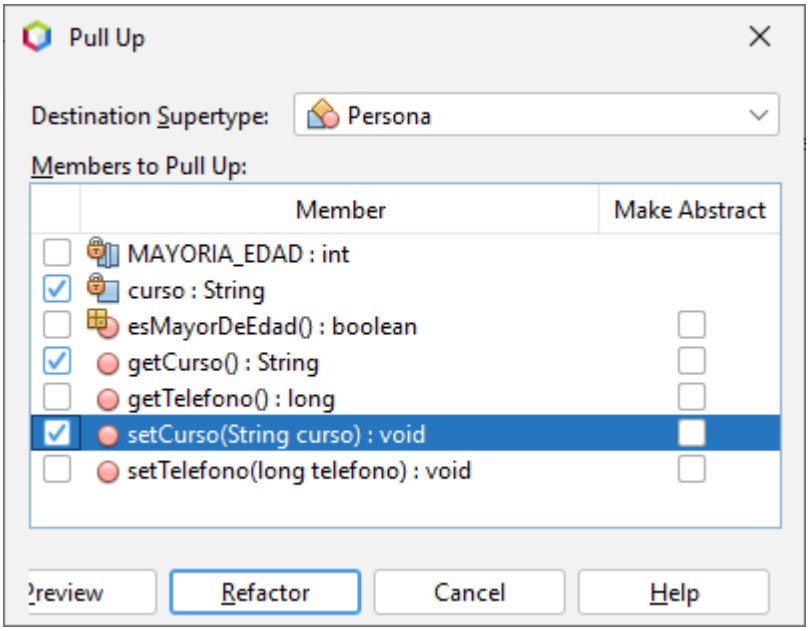
Na ventá de *Usos* indícase que hai 4 referencias: 1 no arquivo *Main.java* e 3 no arquivo *CirculoTest.java*. Movéndose por esas referencias cas iconas do menú da esquerda ou directamente facendo dobre clic riba dunha referencia, pódese ver e editar a liña de código fonte no que está a referencia, podendo modificala para que deixe de existir esa referencia. Despois de modificar a referencia, deberíase de premer en *Ejecutar de nuevo Eliminación segura* para volver a empezar o proceso de eliminación segura ata que non existan referencias ao método *obtenerRadio* e apareza a ventá de *Reestruturando* en cuxo momento 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 en *Edición →Deshacer* se é que 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 Reestruturar →Ascender(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 moveríanse as 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 sobrescrito 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 chekboxes correspondentes.

Mover membros dunha clase a unha subclase

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

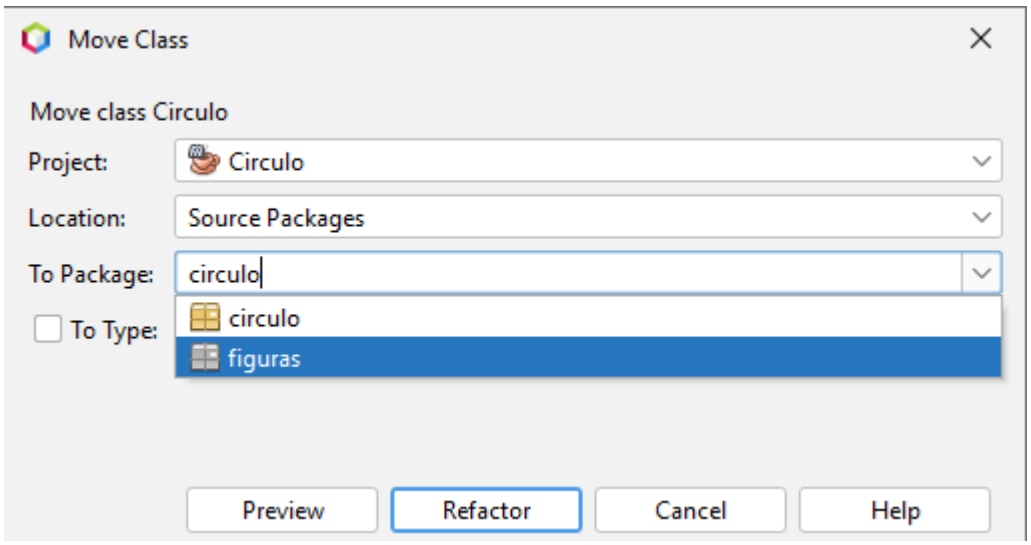
- No código fonte ou na ventá de proxectos, hai que seleccionar os elementos que se queren mover e elixir Reestructurar→Descender (Push down).
- Aparece a caixa de diálogo Descender cunha lista de membros da clase co mesmo aspecto cá caixa de diálogo Ascender. Hai que marcar o checkbox dos que se queren mover.
- De existir métodos abstractos que se desexan manter definido na clase actual e ter implementados na subclase, haberá que marcar o checkbox Mantener abstracto 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 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 Reestructurar->Mover.
- Aparece a caixa de diálogo Mover clase.



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:

- *Proyecto*: co nome do proxecto que contén as clases a mover.
- *Localización*: a parte do proxecto que contén as clases a mover. Normalmente *Paquetes de* fuentes.
- *Al paquete*: nome do paquete ao que se queren mover as clases (recoméndase esta opción). Neste caso creamos anteriormente o novo paquete que se chama figuras e por iso podémolo seleccionar como destino.

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.

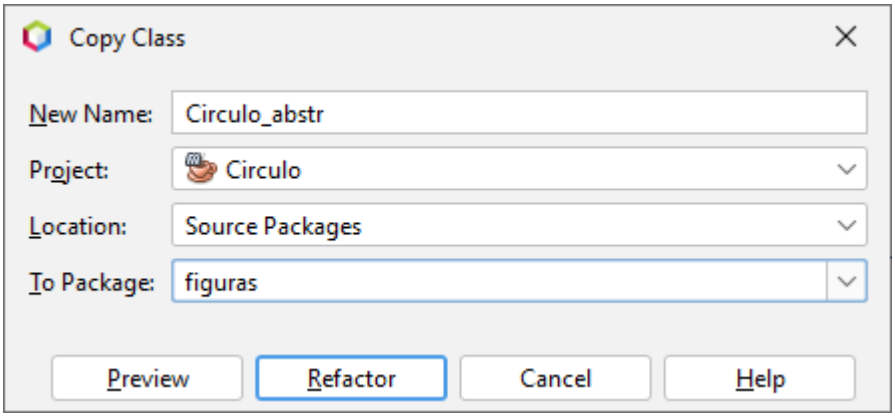
Non se recomenda mover unha clase a outro paquete sen utilizar a reestruturación aínda que é posible realizando os pasos seguintes:

- 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 Mover clase e selecciónase o checkbox Mover sin reestructurar.

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 Reestructurar→Copiar.
- Aparece a caixa de diálogo Copiar clase que é similar en visualización e compoñentes á caixa Mover clase. Nesta caixa selecciónase o paquete ao que se quere copiar (recoméndase esta opción) ou tecléase o nome completo.



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 chekboxes 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 Copiar clase e selecciónase o checkbox Copiar sin reestruturar.

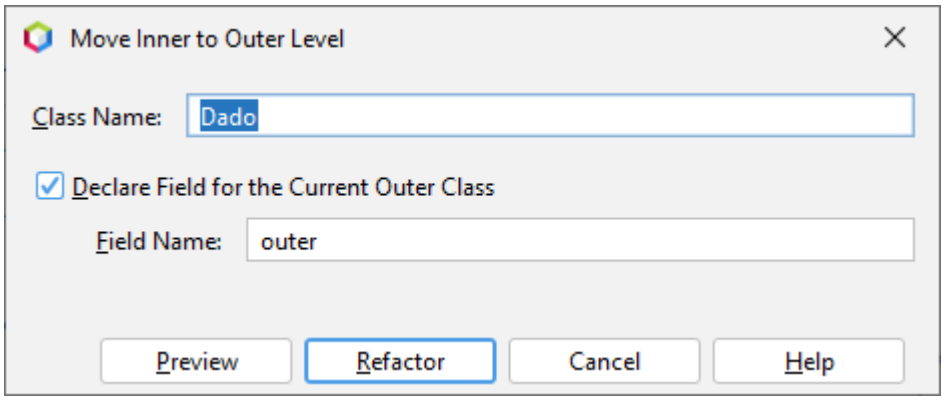
Mover unha clase de nivel interior a exterior

Pódese mover unha clase interna a un nivel superior na xerarquía de clases. Por exemplo, si 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 Reestruturar →Mover de nivel interior a exterior.
- Aparece a caixa de diálogo Mover de nivel interior a exterior.

Nesta caixa:

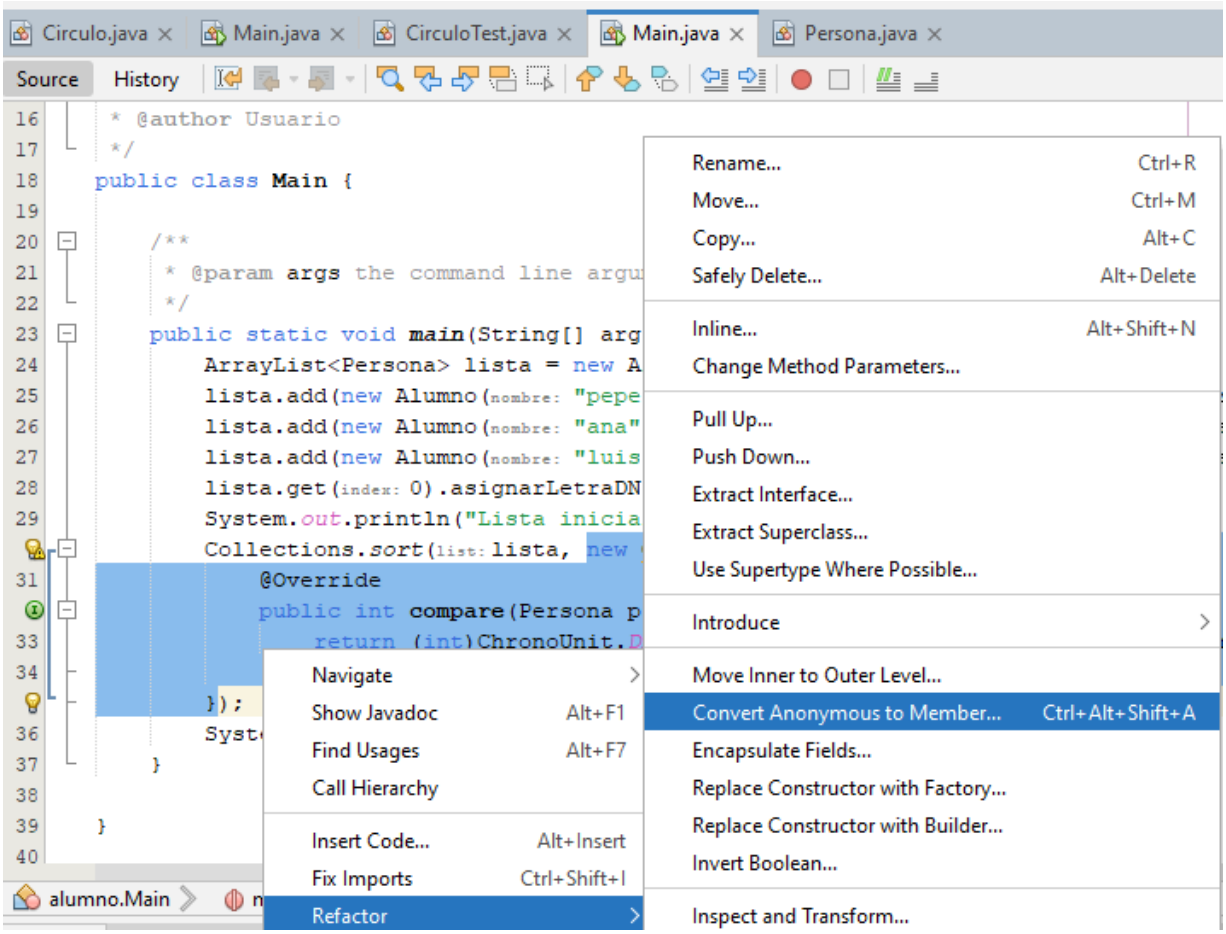
- Pódese cambiar o nome da nova clase
- Opcionalmente pódese crear un campo na nova clase que fará referencia a un obxecto da clase envolvente orixinal e darlle nome a ese campo.



Converter anónimo en membro

Pódese converter unha clase anónima en clase interna que contén nome e construtor, creando unha nova clase interna e substituíndo 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 Converter *anónimo en membro.*



Obtendo:

```
public static void main(String[] args) {
    ArrayList<Persona> lista = new ArrayList<>();
    lista.add(new Alumno(nombre: "pepe", dni:"12345678", fechaNacimiento:LocalDate.p
    lista.add(new Alumno(nombre: "ana",  dni:"87654321X", fechaNacimiento:LocalDate.
    lista.add(new Alumno(nombre: "luis", dni:"00000000T", fechaNacimiento:LocalDate.
    lista.get(index: 0).asignarLetraDNI();
    System.out.println("Lista inicial:\n"+lista);
    Collections.sort(list: lista, new ComparatorImpl());
    System.out.println("Lista ordenada por edad:\n"+lista);
}

private static class ComparatorImpl implements Comparator<Persona> {

    public ComparatorImpl() {
    }

    @Override
    public int compare(Persona p1, Persona p2) {
        return (int)ChronoUnit.DAYS.between(temporalInclusive: p2.getFechaNacimie
    }
}
```

Outra refactorización posible e recomendable, es substituír a clase anónima por unha función lambda. Ofrécenos esa opción como suxestión:

```
This anonymous inner class creation can be turned into a lambda expression.
----
(Alt-Enter shows hints)
        Collections.sort(lista, new Comparator<Persona>() {
            @Override
            public int compare(Persona p1, Persona p2) {
                return (int)ChronoUnit.DAYS.between(p2.getFechaNacimiento(), p1.getFechaNacimiento());
            }
        });
```

e premendo no aviso de número de liña.

```
        Collections.sort(list: lista, new Comparator<Persona>() {
Use lambda expression
            public int compare(Persona p1, Persona p2) {
                return (int)ChronoUnit.DAYS.between(temporal1Inclusive: p2.getFechaNacimiento(), temporal2
            }
        });
```

Obtemos a versión coa expresión lambda:

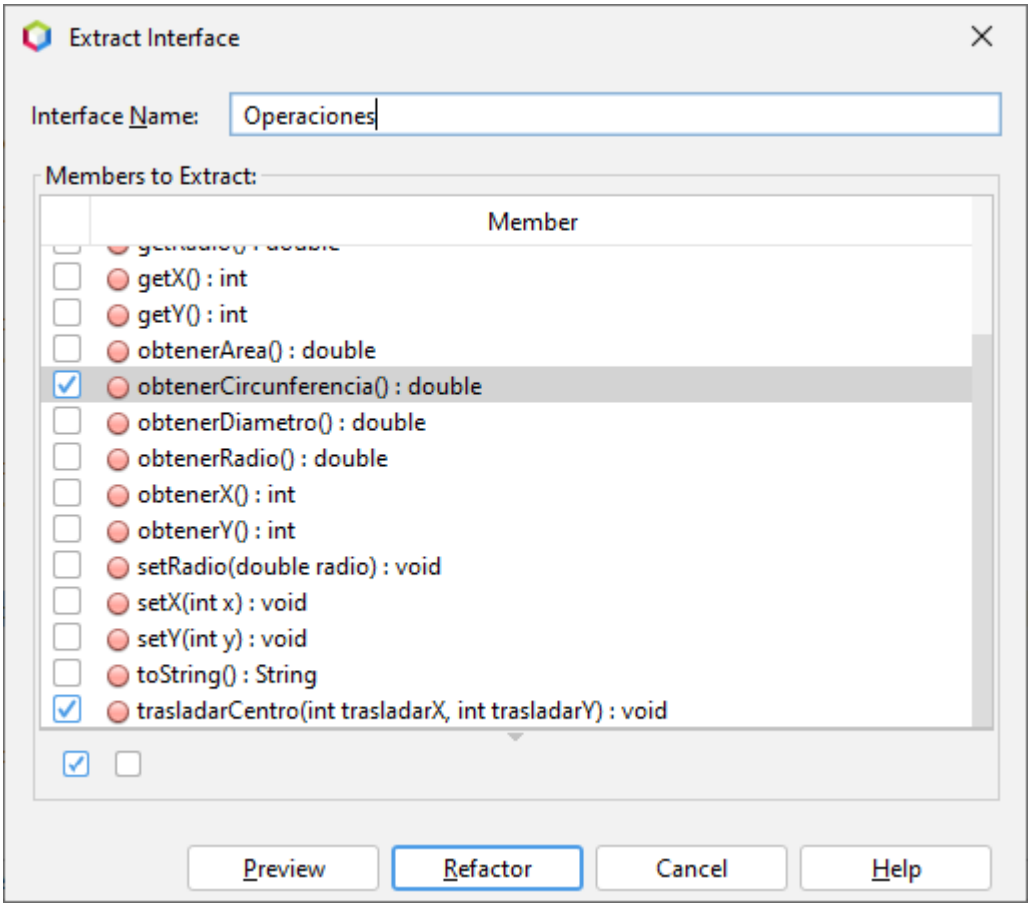
```
        System.out.println("Lista inicial:\n"+lista);
        Collections.sort(list: lista, (Persona p1, Persona p2) -> (int)ChronoUnit.DAYS.between(temporal1Inclusive: p2.getFechaNac
        System.out.println("Lista ordenada por edad:\n"+lista);
    }
```

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 cá 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.

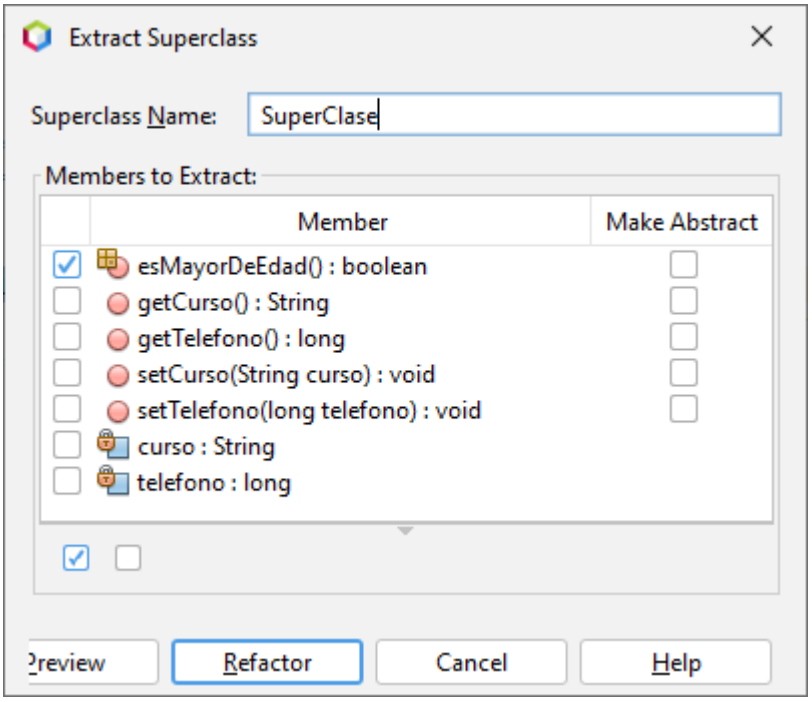
Extraer superclase

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 Reestruturar →Extraer superclase.
- Ábrese o cadro de diálogo de Extraer superclase 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 Converter en abstracto para o método. Se seleccionas este checkbox, o método será declarado na superclase como un método abstracto e sobrescrito na clase actual.

Despois de reestruturar

Despois de realizar algunha 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 *Limpiar y generar*.

> ✍ **Tarefa 4.1. Practicar a reestruturación de código Java en NetBeans. Debes indicar en texto a entrada de menú ou menú contextual empregada, unha captura de pantalla das opcións da acción e outra da situación despois da reestruturación.**

Entréganse 3 proxectos: Circulo.zip, Alumno.zip e MTB.zip que se poden importar mediante a opción File > Import project > From zip

Realizar as seguintes reestruturacións sobre o proxecto Circulo que dispón dunha clase de probas unitarias JUnit. Recorda que despois de realizar a reestruturación de cada exercicio, débese comprobar coas probas unitarias que a clase segue pasando as probas.

- Renomear a clase Circulo por Circulito.
- Renomear o método ObtenerArea por ObtenerAreaCirculo.
- Renomear os campo x e y por coordenadaX e coordenadaY.
- Introducir constante LIMITERADIO de tipo double co valor 0.0

- Cambiar parámetros do método trasladarCentro para que teña dous parámetros trasladarX e trasladarY de tipo int. Facer os cambios necesarios para que o código do método permita engadir á coordenada x o valor de trasladarX e o engadir á coordenada y o valor de trasladarY.
- Encapsular os tres atributos da clase: coordenadaX coordenadaY, radio.
- Eliminar de forma segura os métodos obtenerX, obtenerY, obtenerRadio, establecerX, establecerY e establecerRadio que agora son innecesarios facendo os cambios necesarios no código da clase, da clase Main e das probas para que sexan substituídos polos correspondentes métodos tipo get e set creados.
- Ao rematar comproba que as probas seguen funcionando (achega captura de pantalla co resultado)

Realizar as seguintes reestruturacións sobre o proxecto Alumno, modificando se é o caso, os tests implementados.

- Move o atributo 'telefono' da clase Alumno a súa superclase Persona.
- Move o método 'esMayorDeEdad' da clase Alumno a súa superclase Persona.
- Na clase 'Persona' substituir o número 18 por unha constante estática.
- Nos métodos comprobarDNI e asignarLetraDNI repítese o código:

```
char letraCorrecta = ' ';
        try {
             num = Integer.parseInt(numero);
        } catch (NumberFormatException numberFormatException) { numeroValido
= false;}
        String letras = "TRWAGMYFPDXBNJZSQVHLCKE";
        if (numeroValido)
             letraCorrecta= letras.charAt(num%23);
```

  Crea un método privado con ese código que sexa invocado desde os outro dous métodos.

- Na clase Main, converte a clase anónima que está como segundo parámetro do método *Collections.sort* nunha clase membro.
- Ao rematar, comproba que o proxecto e os tests seguen funcionando correctamente.

Realizar as seguintes reestruturacións sobre o proxecto MTB.

Neste proxecto non hai tests JUnit.

- Move o atributo marcha, o método getMarcha e o método setMarcha dende a superclase Bicicleta á subclase MTB ca opción de menú Reestructurar/Descender probablemente quede algún problema que solucionar e non se faga todo automaticamente soluciona e explica dito problema.
- Extrae un interface para os métodos getAltoAsiento e setAltoAsiento, usando a opción de menú Reestruturar/Extraer Interface.
- Extrae unha superclase co campo velocidad e os métodos getVelocidad, acelerar e frenar usando a opción de menú Reestruturar/Extraer Superclase.

2. Control de versións

2.1 Definición

Chámase control de versións á xestión dos diversos cambios que se realizan sobre os elementos dalgún produto ou unha configuración do mesmo. Unha versión, revisión ou edición dun produto, é o estado estable no que se atopa devandito produto nun momento dado do seu desenvolvemento ou modificación.

Os programas de control de versións permiten traballar de forma conxunta a un grupo de persoas no desenvolvemento de proxectos normalmente a través de Internet.

O control de versións leva a cabo principalmente na industria informática para controlar ases distintas versións do código fonte pero tamén se aplica noutros ámbitos como documentación, imaxes, sitios web, e en xeral en calquera proxecto colaborativo que requira traballar con equipos de persoas de forma concorrente.

O control de versións de código está integrado no proceso de desenvolvemento de software de moitas empresas sobre todo se teñen máis dun programador traballando no mesmo proxecto.

2.2 Características

Os programas de control de versións realizan funcións indispensables durante a vida dun proxecto entre as que destacan:

- Permitir o control dos usuarios que traballarán en paralelo no proxecto:
  - Establecer os usuarios que terán acceso.
  - Asignarlles o tipo de acceso.
- Con relación aos ficheiros:
  - Permitir o almacenamento dos ficheiros.
  - Permitir realizar cambios sobre os ficheiros almacenados: modificar parcialmente un arquivo, borralo, cambiarlle o nome, movelo,...
  - Dispor dun histórico detallado (cambios, data, motivo, usuario,...) das accións realizadas no tempo.
- Con relación ás versións:
  - Etiquetar os arquivos nun punto determinado do desenvolvemento do proxecto para sinalar unha versión estable e así poder identificala posteriormente mediante esa etiqueta.
  - Dispor dun histórico detallado das versións (usuario responsable, data,...).
  - Permitir a recuperación de todos ou algún dos arquivos dunha versión.
  - Comparar versións tendo unha vista ou informe dos cambios entre elas.

- Con relación ao proxecto, permitir a creación de ramas ou branchs, é dicir, bifurcar o proxecto en dous ou máis liñas que poden evolucionar paralelamente por separado. As ramas poden utilizarse para ensaiar novas características de forma independente sen perturbar a liña principal do desenvolvemento. Se as novas características son estables a rama de novo desenvolvemento pode ser fusionada coa rama principal ou tronco.

Exemplos:

Un proxecto no que a versión 1.0 considérase estable e péchase e no que se crea unha rama para a versión 1.1.

- Na rama da versión 1.0 pódese seguir traballando na solución de novos erros que aparezan e seguir creando novas versións: 1.01 e posteriores.
- Na rama da versión 1.1 pódense traballar de forma independente pero coa posibilidade de utilizar as modificacións que permitiron emendar erros na versión 1.0.

Proxecto para elaborar o manual dunha aplicación. Un departamento dunha empresa pide adaptar algunha das partes do manual á súa forma especial de traballar.

- Poderíase abrir unha rama no tronco principal (manual orixinal) para desenvolver o manual específico de forma independente do manual orixinal. Se aparece un cambio que debese de aplicarse ao tronco e a todas as ramas, por exemplo, un erro ortográfico no tronco, o cambio poderíase aplicar a todos eles.

2.3 Operacións básicas

Os sistemas de control de versións teñen un repositorio local para cada proxecto, é dicir, un lugar no que se almacenan todos os arquivos do proxecto. O repositorio ten que crearse e administrarse para que conteña o proxecto (novo ou importado doutro proxecto), teña a estrutura de directorios necesaria, e sexa compartido por un grupo de usuarios autorizados. Adicionalmente podemos ter un repositorio remoto (GitHub, GitLab)

Crear o repositorio local (init)

Esta é a primeira operación e permite crear un proxecto non versionado existente no computador do usuario por primeira vez. Traballa a nivel de cartafoles polo que o que faremos é situarnos nun cartafol e crear o repositorio.

Engadir arquivos a repositorio local (commit)

Esta operación permite actualizar o contido dun repositorio local cos cambios realizados no código. Tamén se coñece como publicar ou chek-in. En Git veremos que esta operación divídese en dous: add e commit (ou ben commit –a)

Un usuario pode modificar algún dos arquivos da súa copia local ou crear arquivos novos. Unha vez que considere finalizados os cambios debe enviar os cambios realizados no seu directorio de traballo ao repositorio local.

Os cambios deben de ir acompañados de comentarios que os xustifiquen para que todos os usuarios estean informados.

Este paso, nun contorno multiusuario, pode producir conflitos. Por exemplo:

- Os usuarios X e Y traballan sobre o arquivo A.
- O usuario X publica cambios entre as liñas n1 e n2 ao arquivo A.
- O usuario Y non descarga do repositorio o arquivo A tras a publicación do usuario X.
- O usuario Y realiza cambios entre as liñas n1 e n2 e tenta posteriormente publicar eses cambios. Neste momento o sistema detecta que a copia local sobre a que traballou o usuario Y cambiou e é incapaz de realizar os cambios automaticamente. O usuario Y debe resolver o conflito combinando os cambios, ou elixindo un deles para descartar o outro.

Para favorecer o traballo en equipo, recoméndase que as tarefas asignadas para o desenvolvemento resólvanse a curto prazo.

Actualizar directorio de traballo desde o repositorio (check-out)

Esta operación permite actualizar o directorio de traballo cos cambios (arquivos modificados, directorio novos, arquivos novos, directorios que quedan baleiros no directorio,...) realizados no repositorio local. Tamén se coñece como sincronizar.

A medida que os usuarios van facendo cambios nos arquivos e vanos publicando no repositorio, os directorios de traballo dos outros usuarios quedan desactualizados.

Crear ramas (branch)

Esta operación permite bifurcar o proxecto en ramas que levarán unha evolución paralela do código de tal maneira que en calquera momento poida realizarse unha fusión de cambios entre elas.

Exemplos:

- Ao pór en produción un software, péchase unha versión do mesmo, e pódese crear unha rama evolutiva na que o proxecto seguirá evolucionando e outra correctiva na que se resolverán os erros que poidan xurdir desa versión.
- Crear unha rama para realizar unha modificación sobre a versión que podería facer inestable o tronco.

A xestión das versións en desenvolvemento complícase a medida que se crean ramas.

Deben de considerarse as ramas como de vida limitada, ben polo peche dunha versión ou por fusión de cambios coa rama da que se fixo a ramificación ou por utilizala para crear un proxecto novo.

Descargar repositorio remoto clone/pull

Esta operación permite que un usuario poida crear un directorio de traballo no seu disco duro local cunha copia dunha versión dun repositorio, normalmente a última. Esta operación faise a primeira vez que se descarga o proxecto do repositorio.

merge

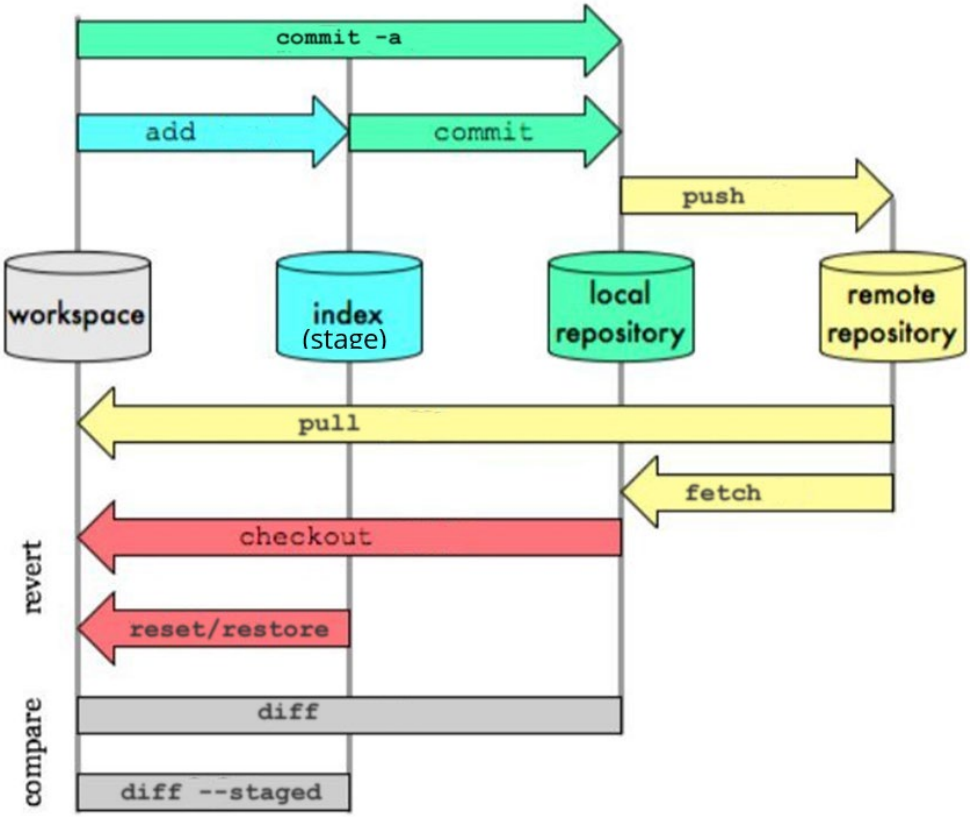
Esta operación permite aplicar todos os cambios realizados entre dúas versións nunha rama a outra rama calquera do repositorio.

Exemplo: Supondo que se ten unha rama correctiva e unha evolutiva e que se realizan cambios na rama correctiva para emendar un erro, poderanse fusionar os cambios realizados na rama correctiva cos cambios da rama evolutiva para que así a rama evolutiva non conteña ese erro.

Funcionamento diff

Antes de enviar os cambios ao repositorio, pode resultar interesante ver os cambios que se realizaron con relación ao repositorio. Esta operación coñécese como diff.

A seguinte figura mostra un resumo dos movementos do código entre o directorio de traballo, o repositorio local e o repositorio remoto. Os comandos os veremos na seguinte sección, na ferramenta *git*.



2.4 Clasificación

Programas de control de versións baseados no:

Modelo cliente-servidor

Un Sistema de control de versións centralizado (CVCS) almacena e xestiona os cambios de arquivos nun repositorio central ao que acceden varios usuarios en todo o mundo. A

arquitectura básica dun CVCS implica un servidor central que contén a copia mestra dos arquivos, e varios usuarios poden acceder e modificar eses arquivos.

Nun CVCS, os desenvolvedores poden retirar e rexistrar arquivos no repositorio central, e o sistema realiza un seguimento dos cambios. CVCS a miúdo conéctase a un servidor central para xestionar e realizar un seguimento dos cambios.

Modelo distribuído

No Sistema de control de versións distribuído (DVCS -Distributed Version Control System-), cada usuario ten unha copia completa do código base do proxecto no seu sistema local, incluído o historial completo de cambios realizados nos arquivos, no que pode traballar e modificar sen necesidade de estar conectado a un servidor central.

Os diferentes usuarios de DVCS traballan na mesma base de código simultaneamente, o que lles permite traballar independentemente da súa rama e fusionar os seus cambios sen sobrescribir os cambios dos demais, evitando así conflitos.

Programas software libre

Modelo cliente-servidor

- SVN (Subversion http://subversion.apache.org): Apache Subversion (comunmente coñecido como SVN) foi desenvolto por Apache Software Foundation. SVN é un sistema de control de versións centralizado que permite aos desenvolvedores xestionar os cambios nos seus proxectos de software ao longo do tempo. Distribúese baixo a licenza Apache, unha licenza de código aberto que permite aos usuarios utilizar, modificar e distribuír o software libremente.

  Subversion axuda aos desenvolvedores a xestionar o seu código fonte, documentación e outros activos dixitais. Permite que varios desenvolvedores colaboren na mesma base de código simultaneamente e proporciona un repositorio centralizado para almacenar todas as versións do código. Isto facilita a xestión de cambios, a colaboración con outros e o mantemento dun historial claro de todas as modificacións realizadas no código.

- Perforce (https://www.perforce.com) Perforce ten unha arquitectura centralizada onde un único servidor almacena todos os arquivos e as súas revisións. Os desenvolvedores extraen arquivos do servidor para realizar cambios e logo vólvenos a rexistrar cando terminan. Isto permite aos desenvolvedores traballar nos mesmos arquivos simultaneamente sen interferir cos cambios dos demais.

- TFVC (https://learn.microsoft.com/en-us/azure/devops/repos/tfvc/?view=azure-devops)- Team Foundation Version Control-. É un sistema de control de versións centralizado desenvolvido por Microsoft. En TFVC, todos os arquivos almacénanse nun servidor central e os membros do equipo normalmente teñen só unha versión de cada arquivo nas súas máquinas de desenvolvemento locais. Os datos históricos, incluídas as versións anteriores dos arquivos e os cambios realizados neses arquivos, mantéñense unicamente no servidor.

  FVC tamén admite ramificacións e fusións, o que permite aos desenvolvedores traballar en copias separadas do código sen interferir cos cambios dos demais. Pode crear ramas no servidor e utilizalas para illar o traballo en novas funcións, corrección de erros ou outras tarefas de desenvolvemento. Unha vez que complete o traballo, pode fusionar os cambios no código base principal.

Modelo distribuido:

- Git (http://git-scm.com): Git é un sistema de control de versións distribuído (DVCS) de código aberto e amplamente utilizado. Linus Torvalds creouno inicialmente en 2005 para xestionar o desenvolvemento do kernel de Linux. Co tempo, converteuse nun dos sistemas de control de versións máis populares da industria do software.

  Git proporciona ferramentas para realizar cambios, bifurcar e fusionar código. As ramas de Git son fáciles de fusionar mentres se colabora con outros desarrolladores, ademais de crear e administrar repositorios do seu código fonte. Co seu modelo distribuído, os desarrolladores poden traballar de forma independente no código e fusionar os seus cambios máis adiante.

- Bitbucket (https://bitbucket.org/product): Bitbucket Server é un servidor Git e unha interface web que permite aos equipos colaborar en código e administrar repositorios mentres controla o acceso ao código. Está construído con Java e Apache Maven e ofrece moitas das mesmas operacións básicas de Git que outros sistemas de control de versións baseados en web, como revisar e fusionar cambios de código.

- Mercurial (https://www.mercurial-scm.org): Mercurial foi desenvolto como un sistema de control de versións distribuído e está escrito en Python con obxectivos similares a Git. A natureza distribuída mercurial permite unha maior flexibilidade e solidez no desenvolvemento colaborativo. Cada desenvolvedro ten unha copia completa do seu repositorio de código, incluído o historial completo de cambios. As capacidades avanzadas de ramificación e fusión facilitan aos desarrolladores a xestión de cambios e o traballo en múltiples versións de proxectos simultaneamente. A interface web integrada tamén simplifica o proceso de acceder e compartir datos do proxecto.

- AWS CodeCommit (https://aws.amazon.com/es/codecommit). AWS CodeCommit é un servizo de aloxamento de repositorios Git totalmente administrado que permite aos equipos colaborar en código de forma segura e escalable. Ao aproveitar AWS CodeCommit, os desenvolvedores poden almacenar e facer unha versión o seu código nun repositorio centralizado totalmente incorporado ao resto do ecosistema de AWS.

  Ademais das súas integracións nativas cos servizos de AWS, AWS CodeCommit admite unha ampla gama de complementos e ferramentas de terceiros a través das súass APIs abertas. Isto permite aos desenvolvedores personalizar os seus fluxos de traballo para satisfacer as súas necesidades específicas e ao mesmo tempo aproveitar a seguridade e escalabilidade da plataforma AWS.

3. Git

3.1 Descrición

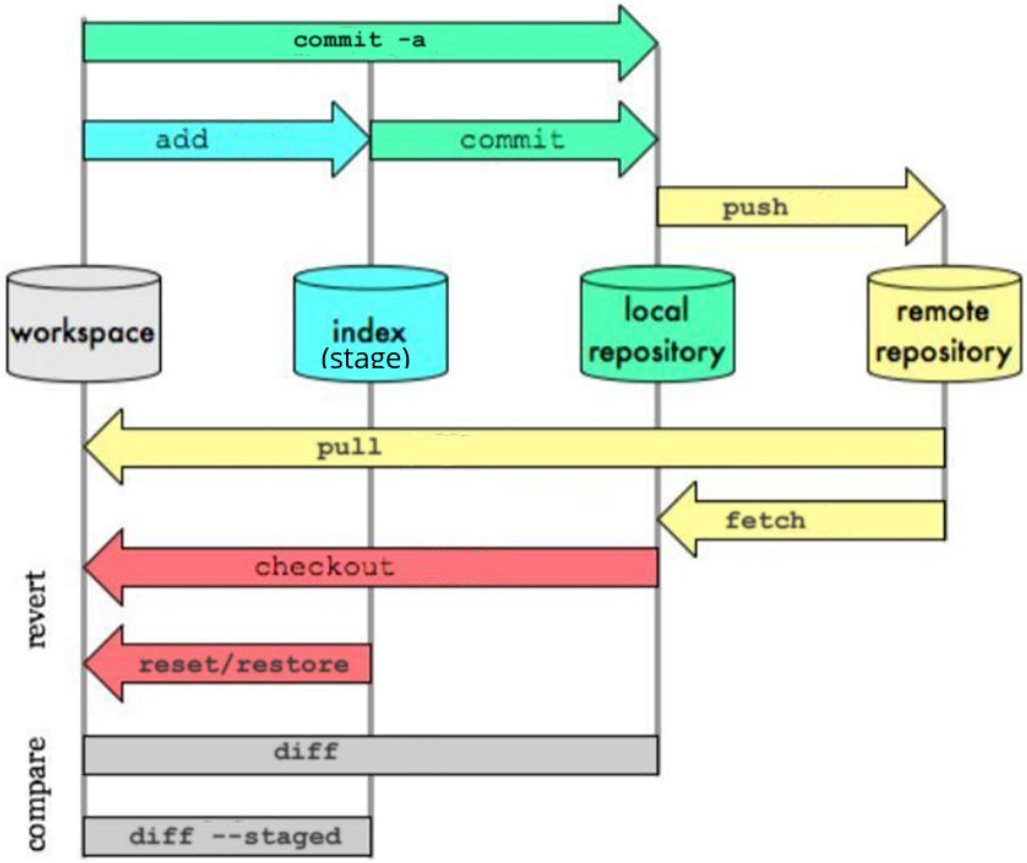
É un software de control de versións deseñado por Linus Torvalds, pensando na eficiencia e na confiabilidade do mantemento de versións de aplicacións cando estas teñen un gran número de arquivos de código fonte (por exemplo, o núcleo de Linux).

O deseño de Git baseouse en BitKeeper e en Monotone. Orixinalmente era un motor de sistema de control de versións de baixo nivel sobre o que outros podían realizar tarefas, aínda que evolucionou a un sistema de control de versións completo.

En Git un arquivo pode estar en tres estados:

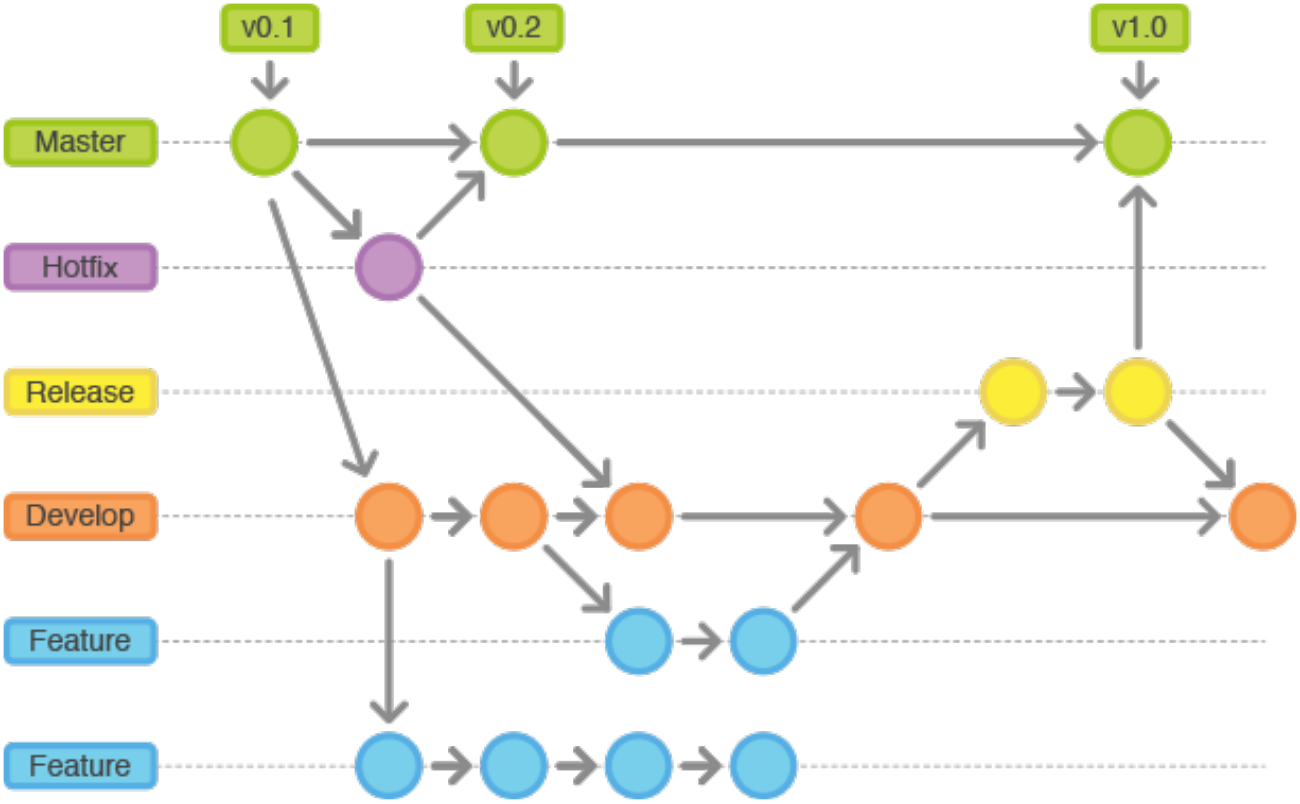
- **Confirmado** (commited): os seus datos xa están actualizados de forma segura no repositorio local.
- **Modificado** (untracked): na túa copia local (directorio de traballo) hai modificacións pero *git* aínda non está informado destes cambios. Como destes arquivos *git* non está ao tanto, non haberá posibilidade de recuperalo en caso de problemas.
- **Preparado** (staged): en un punto intermedio nos anteriores, o arquivo está modificado e listo para ser actualizado no repositorio no momento que desexemos, cando teñamos todos os arquivos da nova versión listos para actualizar no repositorio. Esta área chámase stage ou index.

Estas tres situacións van a determinar os comandos para mover os arquivos dunha situación ou área a outra.



Boas prácticas: "Git Flow", "GitHub Flow", etc...

Existen diferentes modelos de desenvolvemento de versións. Un típico e o "Git Flow" que se basea nas liñas que se ven na seguinte figura:



Cada desenvolvedor ou equipo de desenvolvemento pode facer uso de Git da forma que lle pareza máis adecuada. Sen embargo unha boa práctica é a seguinte:

Débense utilizar 4 tipos de ramas: Master, Development, Features, y Hotfix.

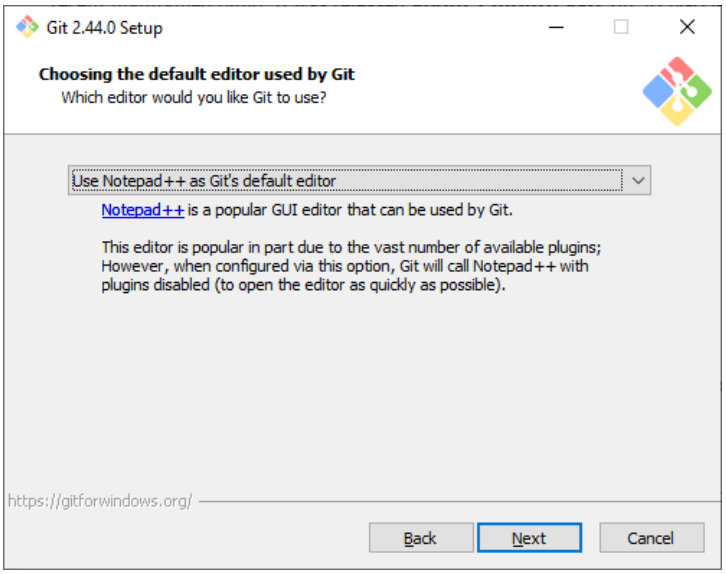
- *Master*: É a rama principal. Contén o repositorio que se atopa publicado en producción, polo que debe estar sempre estable.
- *Development*: É unha rama sacada de '*master*'. É a rama de integración, todas as novas funcionalidades débense integrar nesta rama. Logo que se realice a integración e se corrixan los erros (no caso de haber algún), é dicir, que a rama se atope estable, pódese facer un *merge* de '*development*' sobre a rama '*master*'.
- *Features*: Cada nova funcionalidade debese realizar nunha rama nova, específica para esa funcionalidade. Estas débense sacar de *development*. Unha vez que a funcionalidade estea feita, faise un *merge* da rama sobre *development*, onde integrarase coas demais funcionalidades.
- Ramas de Releases: Son ramas creadas para preparar unha nova versión para o seu lanzamento. Estas ramas créanse a partir de Develop e empréganse para facer probas finais e para preparar os cambios para a súa liberación en produción.
- *Hotfix*: Son bugs que xorden en produción, polo que se deben amañar e publicar de forma urxente. É por iso, que son ramas sacadas de *master*. Unha vez corrixido o erro, debese facer un *merge* da rama sobre *master*. Ao final, para que non quede des actualizada, debese realizar o merge de *master* sobre *development*.

GitHub Flow cambia lixeiramente o modo de traballo. Non hai rama “Development” senón que as novas *features* parten directamente da rama *master.* O *deploy* tamén é diferente por que faise a nivel *feature* , non a nivel *master.*

3.2 Instalación Git en Windows

Descargaremos a ferramenta oficial *Git for Windows* desde https://git-scm.com/download/win. Intégrase no explorador de arquivos.

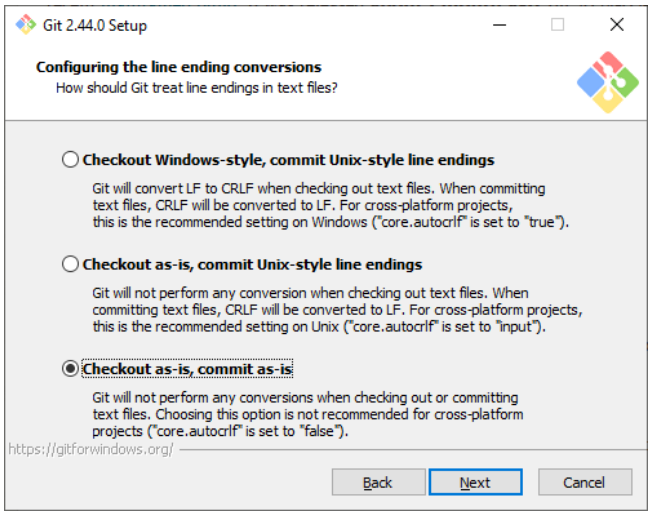
Na instalación deixamos todas as ventás sen cambiar nada agás na que seleccionamos o editor por defecto, marcando Notepad++.



Se non marcamos esta opción, logo, no uso de git podemos asociar Notepad++ á aplicación co comando:

```
git config --global core.editor "'C:/Program Files
(x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"
```

E finalmente na que indica o tratamento de fin de liña (estilo Windows ou Linux) marcamos a terceira opción: checkout as-is, commit as-is xa que traballaremos soamente en un sistema operativo.



Tamén empregaremos unha ferramenta visual chamada *Tkdiff* para ver as diferenzas entre dous arquivos de texto. É un programa sinxelo pero moi visual para ver con cores as diferenzas nas liñas de código (a ferramenta que ven por defecto en gif é *vimdiff* , máis complexa).

Descargamos o .exe de Tkdiff desde https://sourceforge.net/projects/tkdiff/files/tkdiff/4.1.4/ e instalamos sen modificar ningún parámetro. Una vez instalado, copiamos el archivo tkdiff.exe de c:\Program Files (x86)\tkdiff a c:\ProgramFiles\git\usr\bin.

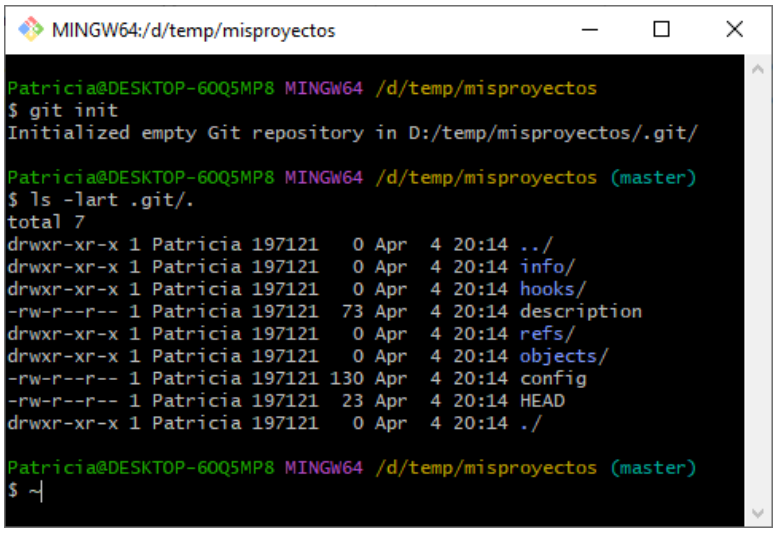
Existen outras opcións para non traballar desde a consola como son Atlassian SourceTree ou GitKraken.

3.3 Creando o repositorio

O primeiro paso será crear un novo cartafol con nome, por exemplo, de *misproyectos,* e con botón dereito: `Open Git Bash Here`

Na consola que aparece, creamos un novo cartafol para o noso primeiro proxecto: `mkdir proyecto1` . Na consola dispoñemos de comandos estilo linux como `pwd` , `cd`, `ls, rm -rf.`

Para crear o repositorio git, executamos: `git init` creándose un cartafol oculto .git coa información necesaria para a xestión da ferramenta. Podes comprobalo con `ls -lart .git/`.



É preciso tamén indicarlle dous parámetros, o email e o nome do noso usuario:

```
git config --global user.name "Patricia Glez"
git config --global user.email "patricia@gmail.com"
```

Esta configuración almacénase en ./gitconfig

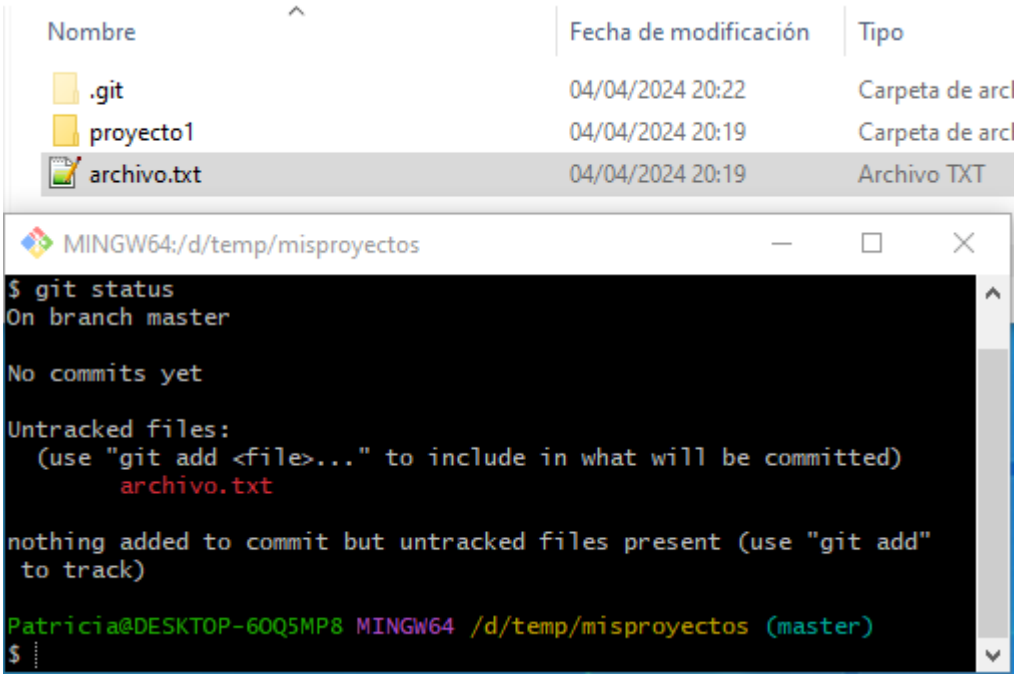
3.4 Operacións básicas

Ollo: Revisar o gráfico do principio deste tema co esquema de operacións git entre working directory / staged (ou index) / repositorio local / repositorio remoto.

Engadir arquivos

Se creamos agora un arquivo nese cartafol (por exemplo o código da nosa aplicación) por agora está fora de git, non ten constancia, non está rexistrado. Podemos comprobalo con **git status**:

```
git status
```



Os arquivos non rexistrados amósanse en cor vermella. O primeiro paso será introducilo no sistema, é dicir, que git o rexistre como un arquivo sobre o que xestionar versións. Para iso executamos o comando **git add**:

```
git add archivo.txt        → engade o arquivo
git add carpeta            → engade toda a carpeta
```

Para engadir máis arquivos nunha soa vez, podemos usar o *punto* para indicar todos os arquivos do directorio actual: `git add .` ou con `-A` ou `--all`. tamén empregar máscaras: `git add *.txt`

Agora repetimos *git status* para comprobar a situación. Comprobamos que está nesa primeira situación chamada **staged**, e dicir, listo para ser enviado ao repositorio.

```
MINGW64:/d/temp/misproyectos

Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ git add archivo.txt

Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   archivo.txt
```

Estes arquivos aparecen en verde, preparados para aprobar.

Agora se facemos **git commit**, o arquivo xa quedaría rexistrado, nunha "foto" ou revisión coa que poderemos traballar ao longo do tempo. Non hai que indicar o nome do arquivo, faise commit de todos os arquivos que estean na situación de *staged* (os que aparecen en verde). Executamos co parámetro –m para indicar cun texto o obxecto desta revisión ou versión

```
git commit -m  "primer commit"
git commit -a -m  "segundo commit"
```

Fai commit de todos os arquivos modificados ou borrados pero non arquivos novos **(aos novos é preciso facerlle add unha primeira vez)**

Se agora facemos *git status* veremos que non hai nada pendente e con: `git log` vemos o historial de cambios.

```
MINGW64:/d/temp/misproyectos

Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ git commit -m "primer commit"
[master (root-commit) a77ae80] primer commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 archivo.txt

Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ git commit -a -m "segundo commit"
On branch master
nothing to commit, working tree clean

Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ git status
On branch master
nothing to commit, working tree clean

Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ git log --oneline
a77ae80 (HEAD -> master) primer commit

Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ |
```

Son interesantes estas opcións de *git log:* `git log --oneline --decorate --graph --all`

Diferenzas entre arquivos

Se modificamos un arquivo do que xa fixemos un *commit*, git detectará a diferenza entre os dous (o da versión do *working directory* e que está na área de *commit*). Para seguir con exemplo, modificamos o arquivo *archivo.txt* inserindo unha liña.

```
Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   archivo.txt

no changes added to commit (use "git add" and/or "git commit -a")

Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ |
```

Para ver as diferenzas **git diff** ou **git difftool**:

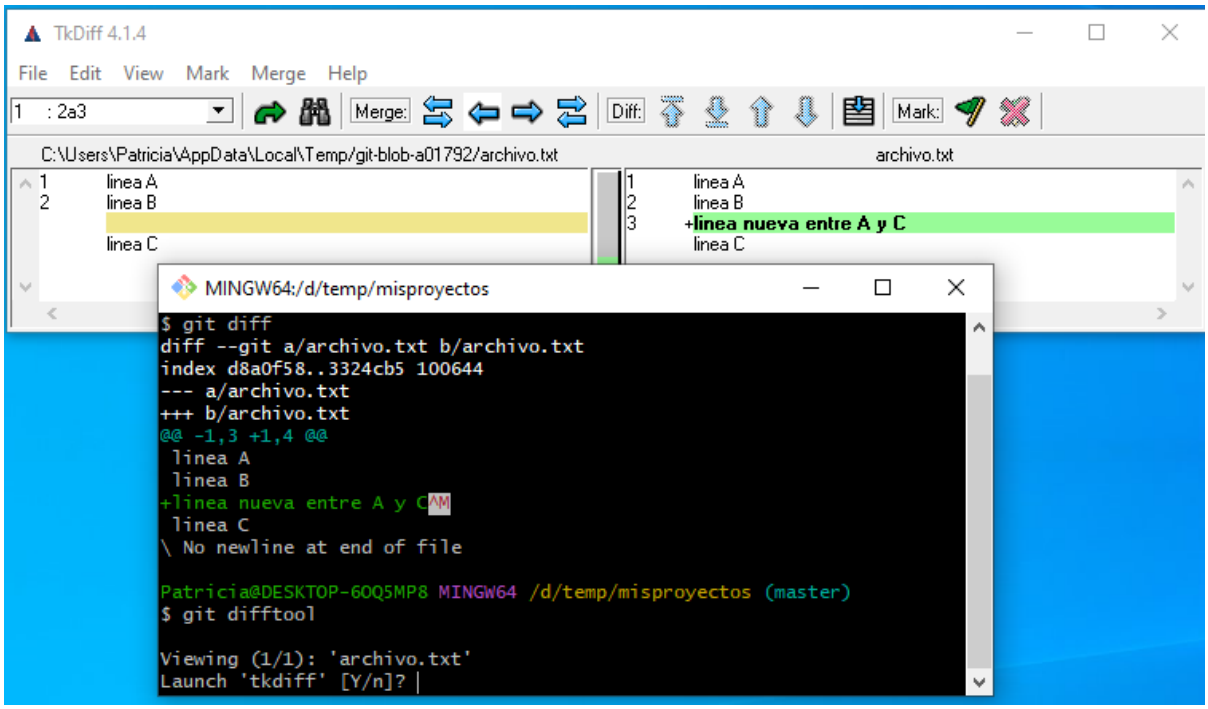
`git diff`

```
Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ git diff
diff --git a/archivo.txt b/archivo.txt
index d8a0f58..3324cb5 100644
--- a/archivo.txt
+++ b/archivo.txt
@@ -1,3 +1,4 @@
 linea A
 linea B
+linea nueva entre A y C^M
 linea C
\ No newline at end of file

Patricia@DESKTOP-6OQ5MP8 MINGW64 /d/temp/misproyectos (master)
$ |
```

E coa ferramenta *tkdiff* que instalamos:

`git difftool`



Borrar e Renomear arquivos

`git diff` e `git difftool` comparan diferenzas de arquivos entre os que están no working directory e no repositorio (*commit*), pero non nos que xa están no *staging*, é dicir non ten en conta sobre os que xa fixemos *add*. Para comparar os *staged* cos que xa están na área de commit engadimos o parámetro `--staged` o `--cached`

Farémolo sempre con **git rm** e **git mv** respectivamente para que *git* rexistre o cambio. Despois de calquera das dúas operacións deberemos facer *commit,* e dicir, confirmar o cambios no HEAD, que apunta á última versión. **Se facemos borrados o renomeados por fora de git estarán *untracked,* e dicir, non rexistrados en git.**

Desfacendo cambios

checkout

Agora imos tratar casos no sentido oposto, se con *add* e *commit* pasabamos arquivos do *working directory* a *stage* e *commit* respectivamente, agora veremos o contrario, como pasar cambios xa aprobados ao noso directorio de traballo, sobrescribindo a versión que teñamos no directorio de traballo.

HEAD é a última situación da rama, a situación actual.

Para pasar da sección *commit* ao directorio de traballo: **git checkout**. Isto é útil para desfacer cambios que teñamos no working directory.

`git checkout -- arquivo` (ou arquivos empregando máscaras ou *punto*.)
`git checkout HEAD -- .` (recupera todo na súa última versión, moi frecuente)
`git checkout idRevision` (Volve provisionalmente a un commit anterior, para botar un vistazo. A cabeza queda en estado desacoplado. Voltamos con git `checkout master`)

reset

Para pasar da sección *staging* ao directorio de traballo: **git reset.** Isto é útil cando cremos que xa temos os cambios listos para pasar ao repositorio pero queremos modificar algo de novo.

A opción **--hard** de *reset* elimina commits previos. É perigoso porque perderemos o histórico desde commit do noso historial (se por exemplo fixemos push a un servidor remoto previamente, pode crear confusión que haxa uns commits no remoto que no existen no local). Por exemplo:

```
git reset --hard HEAD~1
git reset --hard idRevision
```

A virgulilla despois do HEAD indica o número de revisión, se HEAD é a última, HEAD~1 é a penúltima (no teclado a virgulilla escrébese con Alt + 1 2 6 *no teclado numérico ou con* AltGr + 4 *)*

```
git reset <arquivo(s)>
```

Saca o ficheiro ou ficheiros da zona de preparación e volta ao directorio de traballo. Co que o arquivo está coas modificacións que fixéramos na zona de preparación, pero git vai detectar que está modificado.

restore

Outra ferramenta para desfacer cambios é **restore**, que reverte cambios dos que aínda non fixemos commit, cambios non aprobados tanto no working directory como en staged, son revertidos desde a copia que hai na zona de commit. É similar ao checkout e reset, pero usarémolo en caso de arquivos borrados do working directory/staged que queremos recuperar.

`git restore fich` (se está no working directory)
`git restore --staged fich` e logo: `git restore fich` (se está staged)

Diferentes situacións dos arquivos á hora de borrar /restaurar

Borrados por fora de git (por exemplo borrado dende o explorador de arquivos)

*** Se creo un arquivo e o borro (por fóra de git)**

Con un git status non vemos nada. Aí git non sabe nada del e polo tanto non é capaz de recuperalo con restore.

Supoñemos que nunca se fixo commit sobre ese arquivo.

*** Se creo un arquivo, lle fago add e o borro (por fóra de git)**

Con un git status vemos o delete en vermello (un borrado non rexistrado para facer commit dese borrado)

Se facemos un restore, git o coñece (tívoo en area de stage un tempo) así que si é capaz de recuperaro.

*** Se creo un arquivo, lle fago add+commit e o borro (por fóra de git)**

Igual que no caso anterior. Cun git status vemos o delete en vermello (un borrado non rexistrado para facer commit dese borrado)

Si facemos un restore, git o coñece (teno no HEAD) así que si é capaz de recuperalo.

Borrados con "git rm"

A diferenza fundamental co borrado por fóra, é que pasa ese borrado á área de stage. É dicir, é un borrado rexistrado para facerlle un commit a ese borrado e por tanto borrar o arquivo que estivera en HEAD (porque lle fixemos un commit anteriormente).

*** Se creo un arquivo e o borro con git rm**

aí git non sabe nada del, non nos deixa facer git rm (e por tanto non o borra)

*** Se creo un arquivo, lle fago add e o borro con git rm**

Non deixa facer git rm directamente, porque git rm borra o working directory e este xa está en stage/index. Ten medo a que borremos algo que teñamos listo para facer commit.

Habería que facer **git rm -f** (f de force) e nese caso o elimina totalmente ou ben **git rm --cached**, isto elimina o paso add. En calquera caso, volveríamos ó caso inicial, non é capaz de recuperalo, porque o arquivo non está rexistrado.

revert

Para rematar este apartado, un último comando: **revert** que permite reverter a situacións anteriores indicando as modificacións que queremos eliminar. Exemplo:

```
git revert HEAD --no-edit                //reverter o último commit
git revert 899001f --no-edit             //reverter o commit 899001f
git revert HEAD...HEAD~2 --no-edit       //reverter os 3 últimos commits
```

A principal diferenza con *reset -- hard* é que non desaparecen os commits que queren desfacer, se non que se engade un novo commit que desfai eses cambios. O resultado nos arquivos é o mesmo pero seguimos mantendo o historial previo, cun novo commit que desfai os cambios.

Tarefa 4.5. Operacións básicas en Git

Instala git en Windows e tkdiff e, a continuación, realiza as seguintes operacións sobre un arquivo html e css comprobando nun navegador o resultado dos cambios que vas facendo.

a) Crea un cartafol (e repositorio git) chamado *myweb*

b) Incorpora a ese cartafol un arquivo *index.html:*

```
<html>
```

```
    <head>
        <link rel="stylesheet" type="text/css" href="style.css">
    </head>
    <body>
        <h1>Hola mundo</h1>
    </body>
</html>
```

e un arquivo *style.css*:

```
body { background-color: pink;}
h1 {color: blue;}
```

Comproba o estado de git.

c) Pasa os arquivos a *staged*, comproba o estado, pásaos a commit e volve a comprobar o estado e tamén o log. Supón que aquí terminaches a túa web, e xa está enviada ao cliente.
d) Agora pídenche uns cambios: modifica lixeiramente o arquivo *index.html* e executa tkdiff para ver os cambios. Agora te arrepintes dos cambios e aínda non fixeches nin add ni commit deses cambios. Como recuperas o *index.html* na versión que estaba no teu repositorio (na zona de commit)?
e) Fai de novo algúns cambios no *index.html*, e esta vez, si leva eses cambios ao repositorio e consulta o log.
f) Ao teu cliente non lle gustan os cambios así que che pide que voltes a web á situación anterior (toda a web, non o fagas vendo que arquivos modificaches).
g) Borra o arquivo *style.css* (borrándoo por fóra de git, non con: *git rm*) En que estado queda o repositorio? Volve a situación anterior.
h) Borra o arquivo *style.css* (borrándoo con: *git rm*) En que estado queda o repositorio? Volve a situación anterior.
i) Modifica o arquivo *style.css* En que estado queda o repositorio? Volve a situación anterior.
j) Agora queremos probar a opción –a de *git commit*, para facer commit sen o add previo (recorda so funciona para arquivos modificados, non novos, non funcionaría no punto 'c')
k) Mostra o log con todos os cambios feitos no exercicio.

3.5 Traballando con ramas

Unha rama (branch) permite traballar nun mesmo repositorio con versións diferentes dun mesmo conxunto de arquivos. Normalmente unha rama xurde a partir dunha situación (por exemplo da rama MASTER) e a partir dese momento ten "vida propia" e evoluciona de forma independente.

Exemplos típicos de ramas son as correspondentes á fase de desenvolvemento de novas funcionalidades, ou a parches de debemos desenvolver de forma rápida separada doutras liñas de traballo no código, como xa vimos na sección: *Boas prácticas: "Git Flow", "GitHub Flow", etc...*

Poderemos "movernos" entre as distintas ramas sen movernos de directorio. Ao cambiar de rama, git cambia os contidos do working directory, pero mantendo todas as ramas actualizadas. Veremos como logo podemos fusionar distintas "ramas" resolvendo os conflitos que se produzan.

O comando para crear unha nova rama é **branch***:*

`git branch novarama`	(copia da rama actual)
`git branch novarama MASTER`	(copia de MASTER ou rama principal)
`git branch novarama ramaorixe`	(copia de *ramaorixe*)

Para cambiar dunha rama a outra, facemos **checkout**: `git checkout rama`

Agora podemos traballar na nova rama, e todos os cambios soamente afectan a esa rama, permanecendo intactas o resto de ramas.

Se queremos crear unha rama e movernos a ela, en vez de facer o dous comandos anteriores podemos facelo con **checkout –b** e a crea co contido da rama na que esteamos nese momento.

```
git checkout -b novarama
```

Para ver o conxunto de ramas que temos no proxecto: **branch** sen parámetros.

`git branch` (con opción `-a` mostra as ocultas e con `-v` informa do último commit)

Para facer a fusión de rama, situámonos na rama destino é **merge**: `git merge ramaOrixe`

Este é un dos pasos máis importantes e críticos xa que se fusionarán os cambios da rama indicada coa rama actual, quedando un código único. A fusión faise liña a liña e pode haber conflitos se dúas ramas distintas fixeron cambios sobre as mesmas liñas, como veremos no seguinte apartado.

fast forward é un termo que se aplica ao *merge* cando á rama orixe é tal cal a última situación en todos os aspectos. É dicir, non necesitamos ningunha liña da rama actual, collendo a rama orixe dos cambios temos a situación final que buscamos. Neste caso, git é intelixente, e en vez de facer un merge "normal" xuntando liñas dunha e outra rama, o que fai é que o HEAD apunte á última situación da rama orixe en vez de facer os cambios e commit sobre a rama actual.

Para que se dea esta situación, a rama actual non debeu ter ningún cambio desde que creamos a rama orixe ata o momento do merge.

Existe outra forma de facer a fusión de ramas, co comando **rebase**. O resultado final é o mesmo que con *merge* pero a diferencia e que neste caso non se fai a fusión da situación "final" de cada rama, senón que se van aplicando todos os commits que tivera esa rama ao longo da súa historia sobre a outra rama. Esta opción é máis propensa a ter conflitos.

Exemplo:

Para rematar co tema das ramas, quedaríanos por ver como eliminar unha rama (por exemplo unha vez fusionada con outro contido xa non a precisamos). O facemos co comando: **branch –d**

```
git branch -d rama
```

Tarefa 4.6. Xestión de ramas

A tarefa consiste en realizar as seguintes operacións.

a) Crea un repositorio similar ao da tarefa 4.5 e fai commit dos dous arquivos *(podes empregar o anterior borrando o cartafol .git)*

b) Crea unha rama chamada "novaWeb" e cámbiate a ela.

Previamente hai que ter feito algún commit xa que a rama crease desde a zona de commit.

c) Borra nesa nova rama *style.css*, crea *style2.css* e modifica *index.html* con novos textos en empregando o novo arquivo css e mostra o estado

```
<html>
  <head>
        <link rel="stylesheet" type="text/css" href="style2.css">
  </head>
  <body>
        <h1>Nova Web</h1>
  </body>
</html>
```

e un arquivo *style2.css*:

```
body { background-color: black;}
h1 {color: white;}
```

d) Asegúrate que estás na rama *novaWeb* e fai commit dos cambios.

e) Mostra o contido da web desta nova rama no navegador. Cambia a master e mostra o contido da web no navegador.

f) Mostra o log coas opcións: oneline, all y graph (verás o asterisco mostra o último commit de cada rama).

g) Fusiona a rama creada coa rama principal e mostra no navegador a rama principal.gi

h) Mostra o estado e o log.

i) Elimina a rama creada.

3.6 Resolución de erros e conflitos

Ver diferenzas

Xa vimos que o comando *diff* é unha boa ferramenta para comprobar as diferencias entre arquivos que están nas distintas áreas do repositorio (working directory, staged, commit) pero tamén serve para comprobar as diferencias dun arquivo en diferentes momentos (en diferentes commits). Por exemplo para comparar a situación actual coa de fai dous commits:

```
git diff HEAD HEAD~2
```

Conflitos nas modificacións das ramas

Como vimos na sección anterior, 'merge' xunta ramas (a indicadas sobre a rama na que nos atopamos) pero podemos atoparnos con conflitos, e dicir, que en distintas ramas fan cambios sobre o mesmo código ao mesmo tempo.

Situación 1:

1. Creamos desde MASTER unha rama na que modificamos un arquivo nunha liña
2. Creamos desde MASTER outra rama na que modificamos o mesmo arquivo pero noutra liña.
3. Facemos *merge* da rama do punto 1 en MASTER.
4. Facemos *merge* da rama do punto 2 en MASTER

Que ocorre?

Neste caso non habería problema. Git fai a fusión de ramas liña a liña.

Situación 2:

1. Creamos desde MASTER unha rama na que modificamos un arquivo nunha líña
2. Creamos desde MASTER outra rama na que modificamos o mesmo arquivo na mesma liña.
3. Facemos *merge* da rama do punto 1 en MASTER.
4. Facemos *merge* da rama do punto 2 en MASTER

Que ocorre?

- Neste caso o paso 3 non da problemas porque hai ainda non hay conflito.
- No paso 4 atópase que o MASTER actual (despois do paso3) non é o mesmo do que partiu, e isto é un conflito que git non sabe resolver.

Solucións:

- A solución é modificar o arquivo coas liñas que consideremos correctas (as da primeira rama, as da segunda ou o que queiramos) e facer commit a ese arquivo. Podemos ver as diferencias con **git diff *rama1 rama2*** **(ou git difftool)**
- Se hai moitos conflitos e non é sinxelo resolvelo modificando os arquivos involucrados podemos desfacer o último merge: `git merge --abort`

Tarefa 4.7. Conflitos entre ramas

a) Crea un repositorio igual ao da tarefa 4.5 cos arquivos index.html e style.css. *(podes empregar o anterior borrando o cartafol .git)*

b) Crea unha rama chamada cambios01 a partir da rama principal. Nesa nova rama, crea o arquivo style2.css similar ao da tarefa 4.6, elimina o style.css e fai que index.html use a nova folla de estilos.

c) Crea unha rama chamada cambios02 a partir da rama principal (ollo! Non a partir de cambios01) e cambia o contido da etiqueta <h1>.

d) Mostra o log coas opcións *all* e *oneline*

e) Mostra no navegador o contido da web nos tres casos, é dicir, cambiándote de MASTER a cambios01e a cambios02

f) Fusiona MASTER e cambios01.

g) Fusiona MASTER e cambios02. ¿Algún conflito? Explica por que. En caso afirmativo comproba as liñas en conflito e amaña a situación.

h) Borra as ramas que xa non precisas.

Tarefa 4.8. Máis conflitos entre ramas

a) Crea un repositorio igual ao da tarefa 4.5 (con index.html e style.css)

b) Crea unha rama chamada cambios01 a partir da rama principal. Nesa nova rama, crea o arquivo style2.css similar ao da tarefa 4.6, elimina o style.css e fai que index.html use a nova folla de estilos. Modifica o contido da etiqueta <h1> . Fai add/commit dos cambios.

c) Crea unha rama chamada cambios02 a partir da rama principal (ollo! Non a partir de user01) e cambia o contido da etiqueta <h1>.

d) Mostra o log coas opcións *all* e *oneline*

e) Mostra no navegador o contido da web nos tres casos, é dicir, cambiándote de MASTER a cambios01 e a cambios02.

f) Fusiona MASTER e cambios01.

g) Fusiona MASTER e cambios02. ¿Algún conflito? Explica por que. En caso afirmativo comproba as liñas en conflito e amaña a situación.

h) Borra as ramas que xa non precisas.

3.7 Repositorios remotos

Os repositorios remotos son versións do teu proxecto que están hospedados en internet o na intranet da túa empresa. Podes ter varios, dalgúns deles serás o propietario, doutros un colaborador e de moitos deles so terás permisos de lectura.

A forma de traballar con eles, a nivel xeral, consiste en traer os seus datos a local (pull), modificalos e subir os cambios (push).

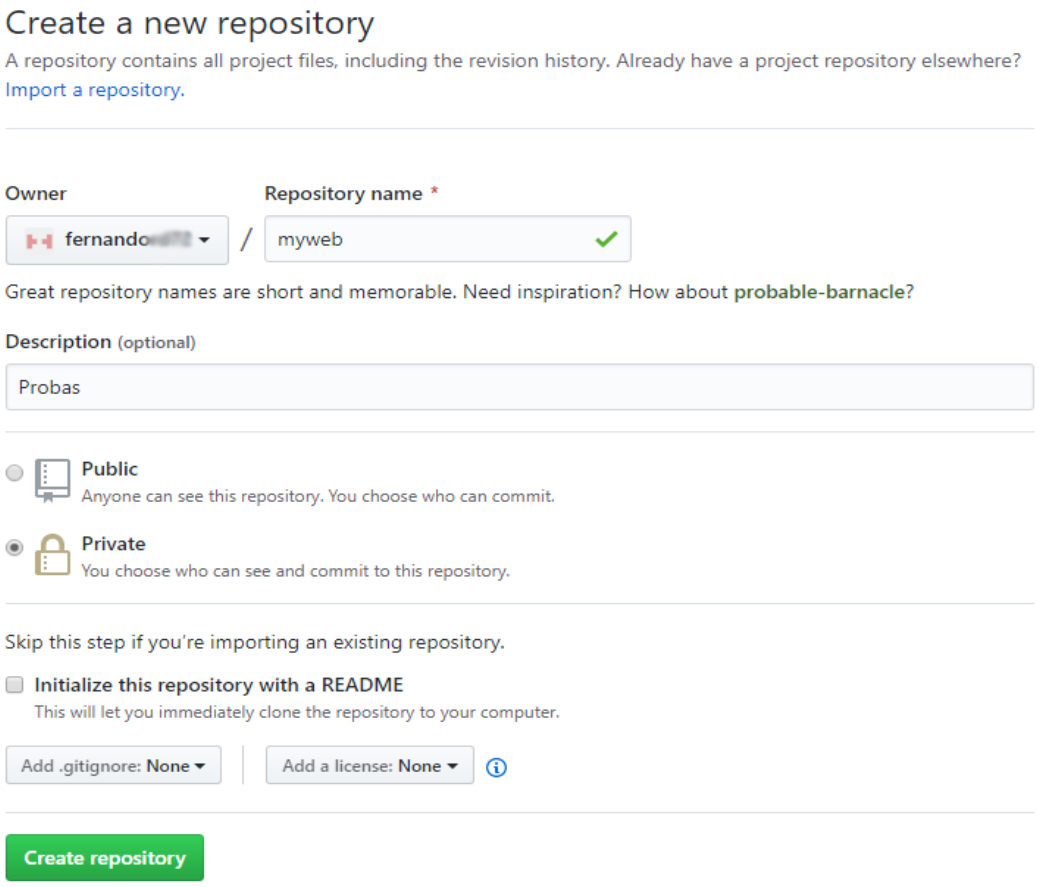
Dispoñemos de dúas ferramentas moi coñecidas para xestionar as nosas versións de código en **remoto** e compartilas con outros usuarios ou co público en xeral: GitHub e GitLab. Teñen ambas unhas características comúns.

- Teñen interface web e como plataforma social para compartir coñecemento e traballo.
- Podemos ter un número ilimitado de repositorios públicos (vista, non commit) pero tamén privados.
- Hai contas para empresa (de pago).
- Permite ramas e comunicarnos cos usuarios (pull request).

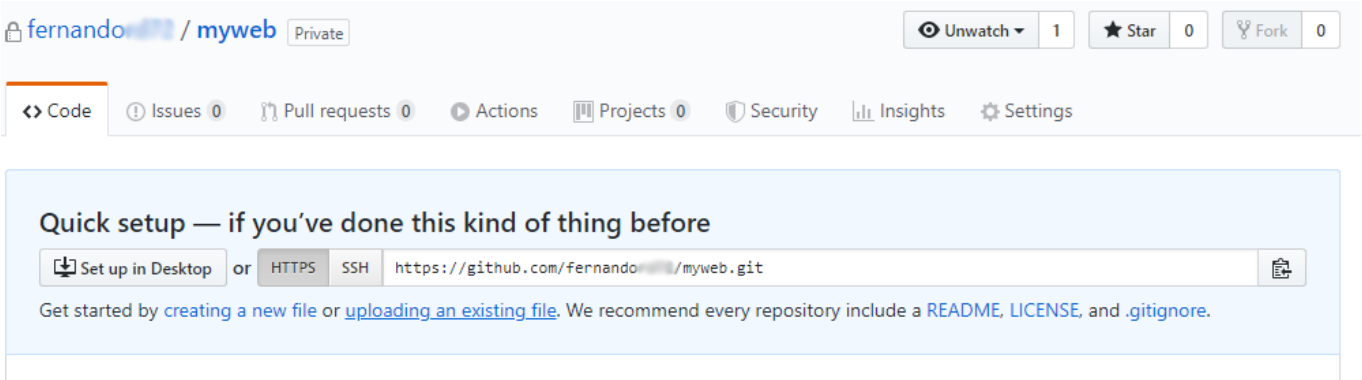
A maior diferenza é que GitLab podémolo instalar no noso servidor e o primeiro sempre hai que úsalo a través da web github.com. Por outra banda, nas versións web gratuítas, GitLab ofrece mellores condicións de espazo e número de colaboradores por proxecto.

Conexión remota

Para conectarnos cun repositorio, primeiro crearemos o noso repositorio local con *git init* .Logo crearemos o repositorio baleiro en GitHub /GitLab:



Obtendo a URL de conexión:



Despois o vinculamos con remoto con **remote add origin url**:

```
git remote add origin https://github.com/fernandoxxxx/myweb.git
```

"*origin* " non é unha palabra reservada, pódese poñer calquera nome, pero é un estándar chamarlle así ao nos repositorio principal. De feito, se temos varios repositorios propios, terán distintos nomes. E claro, os repositorios dos que so somos colaboradores tamén lle asignaremos outros nomes.

Para ver os remotos que temos configurados empregamos simplemente **remote**, coa opción **–v** para ter máis información.

```
git remote -v
```

A partires dese momento, nos comandos traballaremos con este nome, e non coa url real.

Para inspeccionar a información en profundidade: **remote show origin** ou remote show *nomRepo*

"Subir" información ao repositorio remoto

A operación contraria, sería "subir" ao repositorio remoto os cambios feitos en local (os cambios fariámolos co proceso xa visto: modificar o arquivo + add + commit). O comando subilo ao repositorio remoto é **push**:

```
git push origin nomeRama
```
solicitando as credenciais en GitHub

origin é o nome co que nos referimos á conexión remota, e no nome da rama poñeremos a rama que desexamos subir, en moitas ocasións será MASTER

```
git push origin master
```

"Baixar" información do repositorio remoto

O paso seguinte será "ver" a estrutura do repositorio remoto, as distintas "ramas". Veremos máis adiante que as ramas son as distintas versión do programa (a de produción, as que están desenvolvendo novas funcionalidades, outra pode estar amañando algún bug, etc). Sempre haberá unha rama principal, que é a que está en produción e chámase *master.* Descargaremos con **fetch origin** ou ben **fetch** *nomeRepoRemoto:*

```
git fetch origin
```

Unha vez feito isto, temos descargados os arquivos pero nunha rama local oculta (chámase origin/master) Agora temos que pasar o descargado á nosa rama master no repositorio local. Iso facémolo con **merge origin/master**:

```
git merge origin/master
```

Como estas dúas operacións fanse habitualmente xuntas, temos un comando que fai as dúas seguidas. É o comando **pull** e é máis empregando que os dous anteriores.

```
git pull origin master
```

Tamén temos un comando git que fai dunha sola vez o `git init` e o `pull`, e chámase **clone**.

```
git clone https://github.com/nomerepositorio  .
```

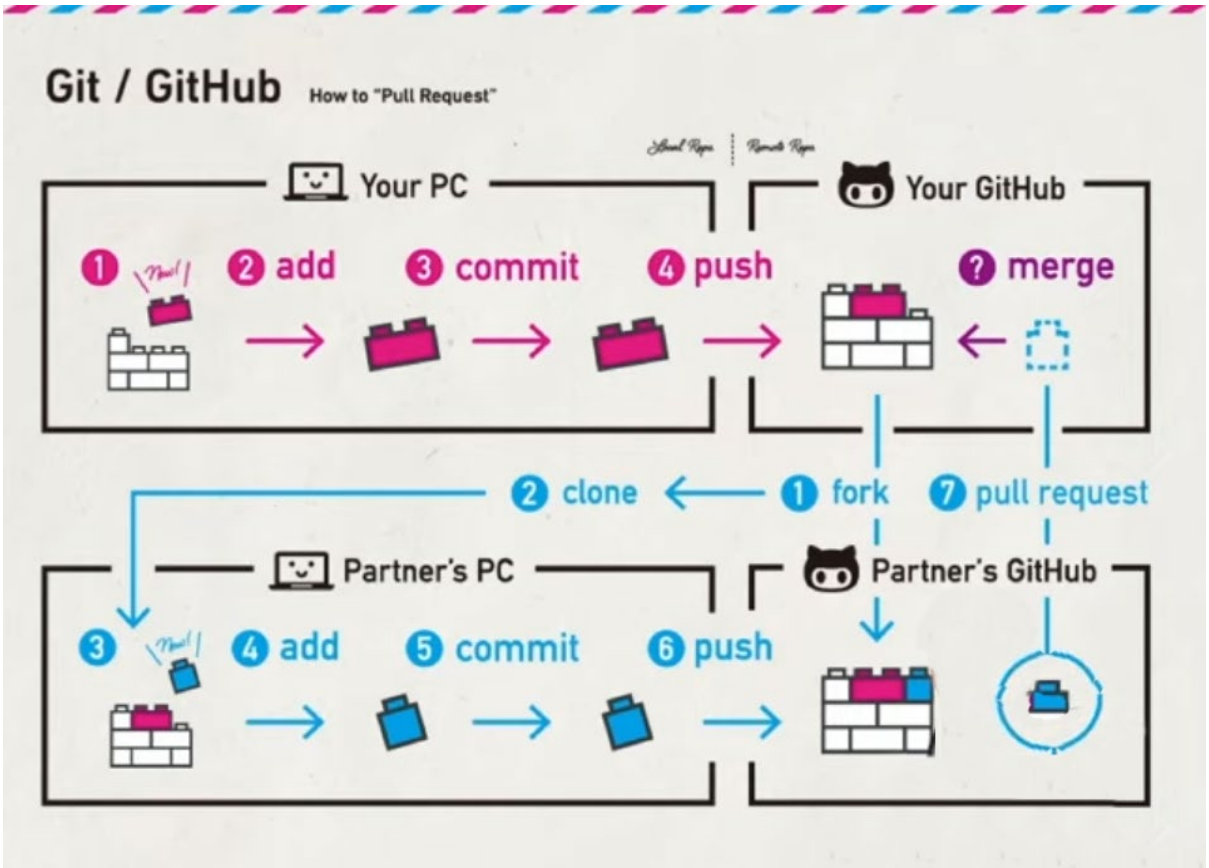
o punto indica que o copie no cartafol actual

Temos tamén unha opción no checkout para traballar con ramas remotas: **checkout -t origin/*nomeRama***: Se existe unha rama remota de nome "nomeRama", ao executar este comando crease unha rama local co mesmo nome para facer un seguimento da rama remota co mesmo nome.

pull request

Son as solicitudes que facemos para modificar proxecto en GitHub que non son nosos, por exemplo para facer contribucións en proxectos de software libre. O propietario é o que vai facer os cambios polo que o noso traballo será obter una copia (*fork*) do proxecto, traelo a local (clone), modificalo (*commit*), subilo á nosa conta (*push*) e facer a petición de integración (*pull request*)

O propietario será o encargado de facer a fusión (*merge*).



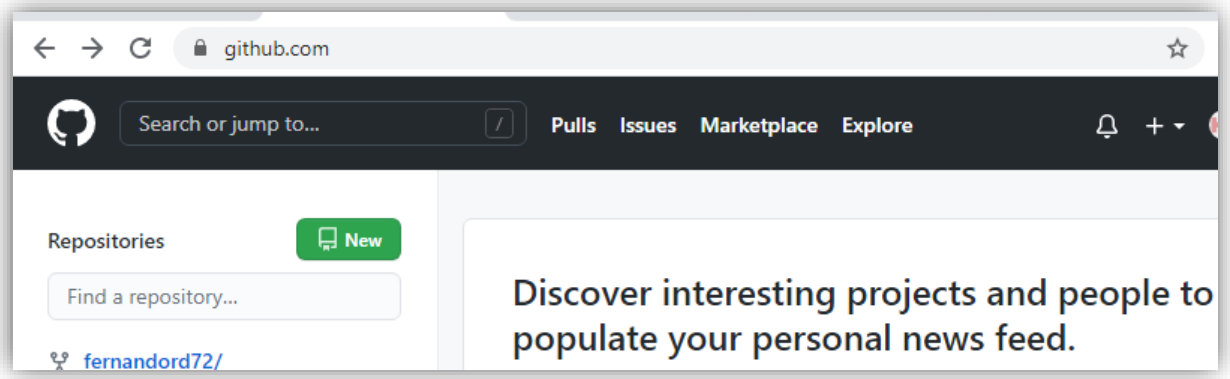
O proceso vai ter lugar entón con dous usuarios diferentes, un que é o dono repositorio orixinal (neste caso empregaremos *fernandord72*) e outro usuario (neste caso *wirtzPruebas*) que é o que vai facer unha copia do repositorio orixinal, modificalo e solicitar ao usuario inicial que integre os cambios propostos.

Imos entón a dividir o proceso en tres fases:

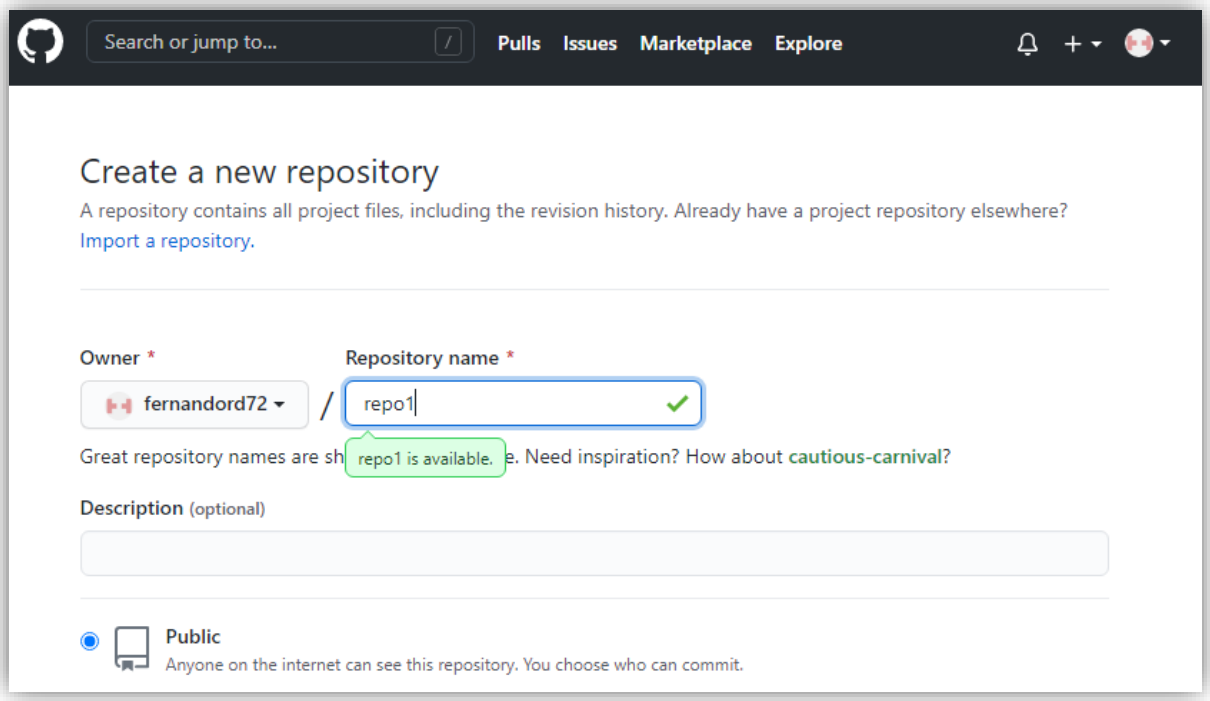
1. O usuario *fernandord72* crear un repositorio en Github.
2. O usuario *wirtzPruebas* fai unha copia (fork) do repositorio previo, modificando esa copia e facendo a *pull request* ao usuario inicial.
3. De novo *fernandord72*, aceptará os cambios (merge) propostos por *wirtzPruebas*.

Pull Request : Crear o repositorio en GitHub

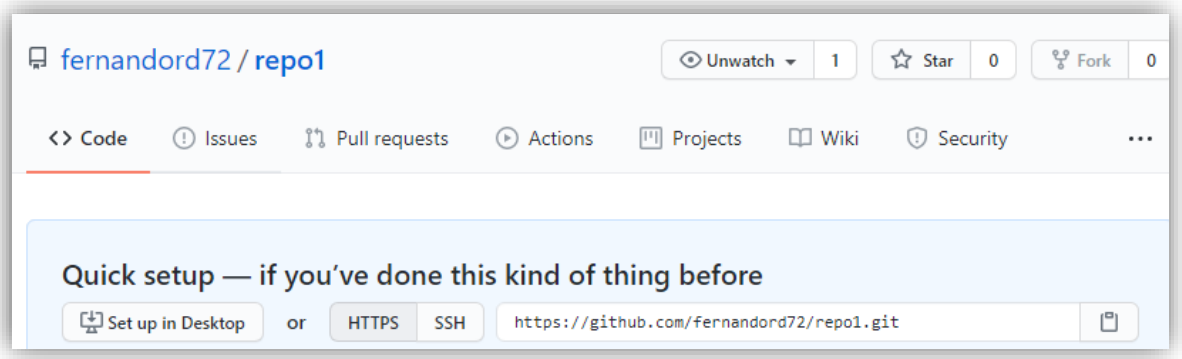
Partimos dun repositorio público en GitHub, na conta *fernandord72*. Podemos crealo desde a propia ferramenta premendo no botón [New]



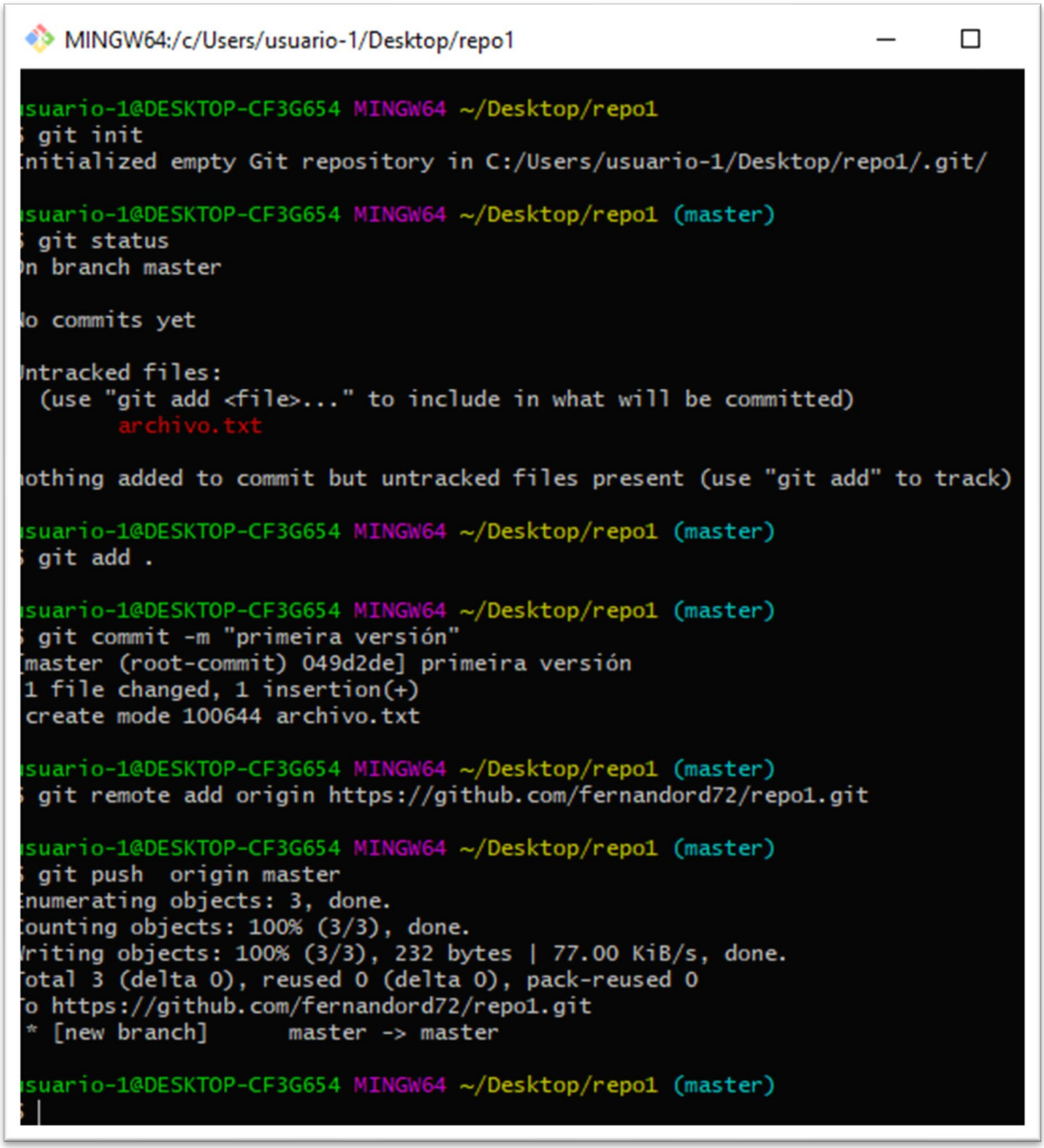
Obtendo:



Amósanos a URL de conexión:



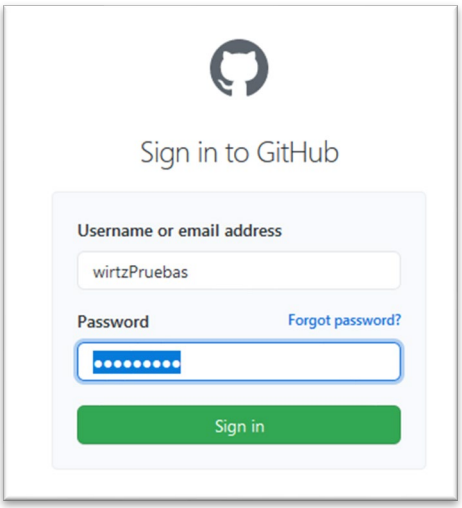
Desde o noso ordenador podemos subir un arquivo a ese repositorio co proceso habitual: add, commit, push...



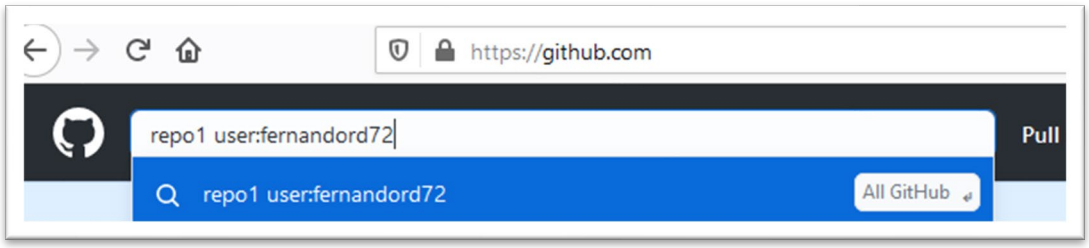
Agora, outro usuario podería facer unha "pull request" ou solicitude de modificación desde repositorio, todo desde a propia web de GitHub.

Facendo a Pull Request

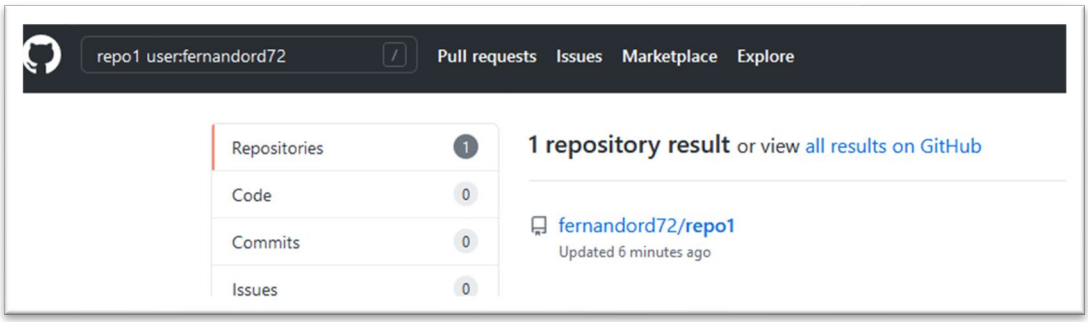
Primeiro outro usuario (*wirtzPruebas)* fai login:



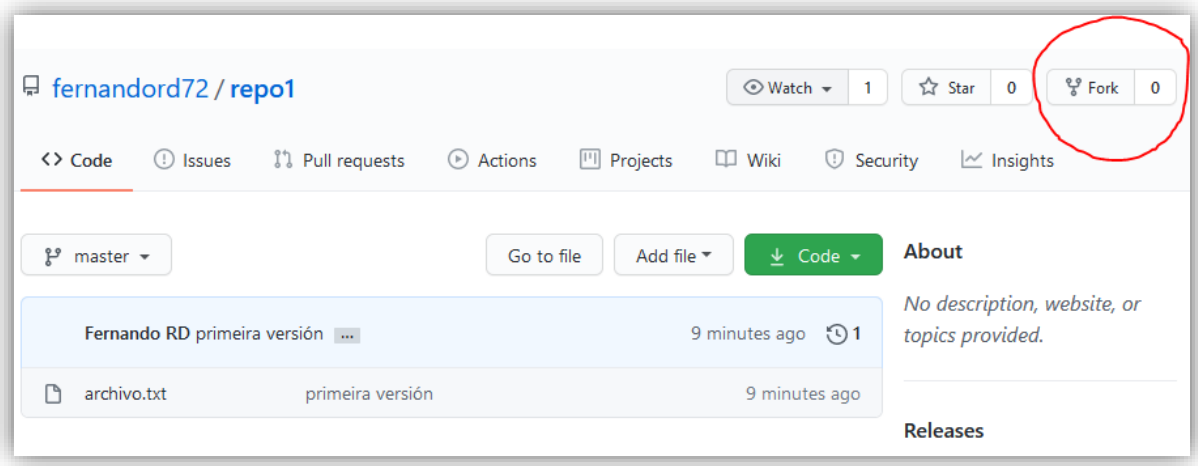
Busca o repositorio ao que quere facer a "pull request" *(repo1 do usuario fernandord72)*:



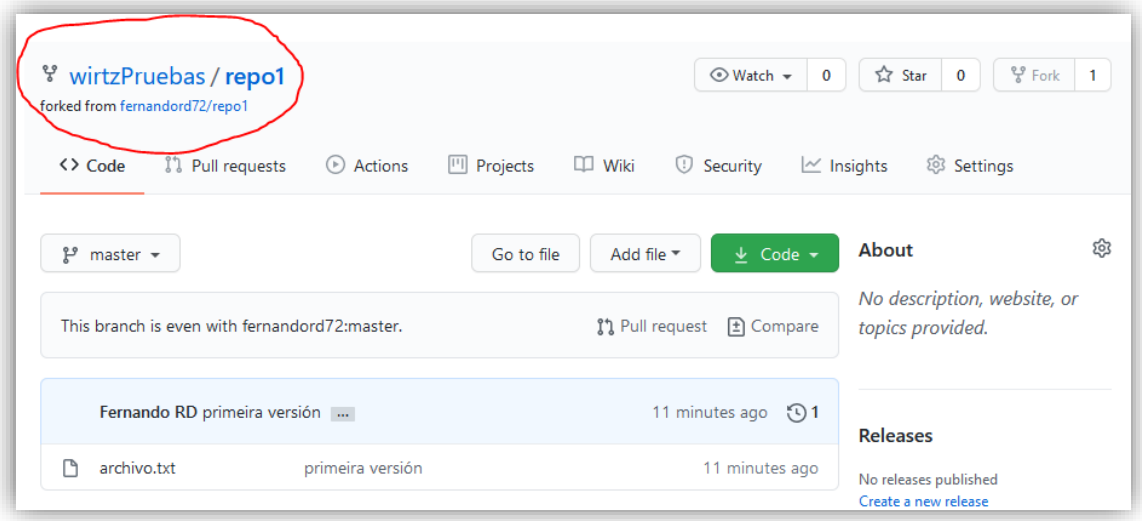
Como é publico non ten problema para atopalo.



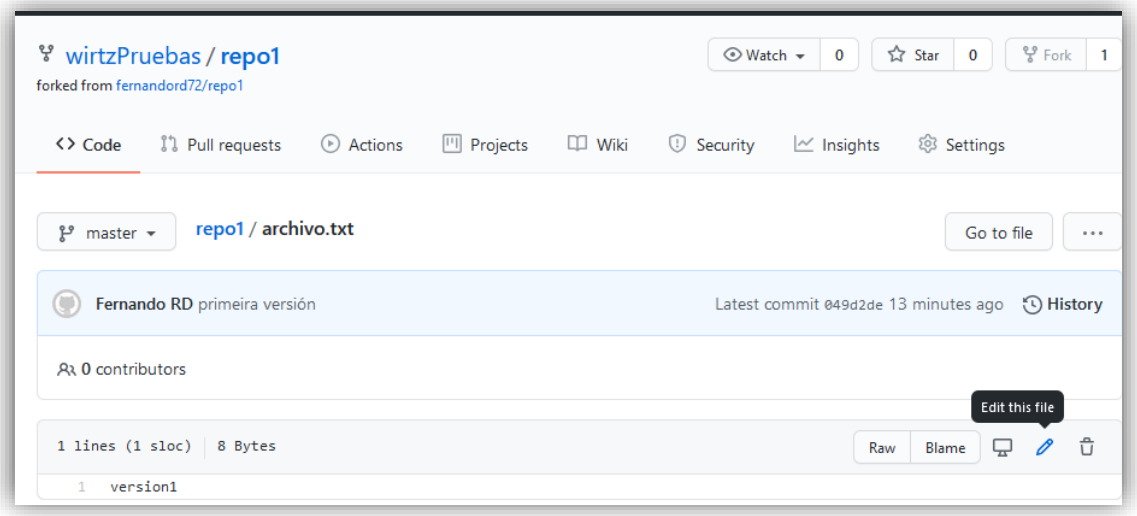
Selecciónao e preme no botón *[Fork]* para crear unha copia do repositorio orixinal na súa conta e poder modifícalo como queiramos sen afectar ao outro repositorio.



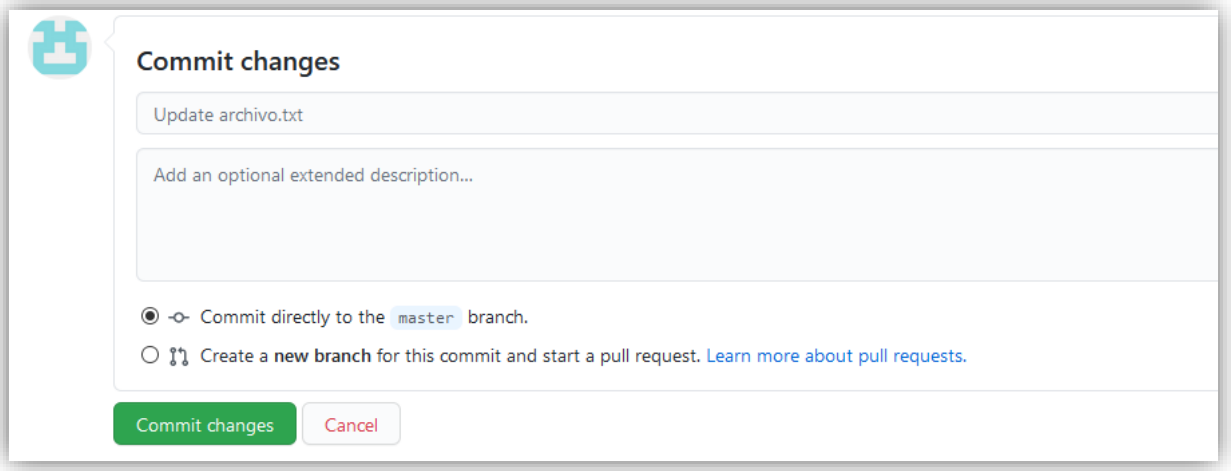
Obtendo o novo repositorio:



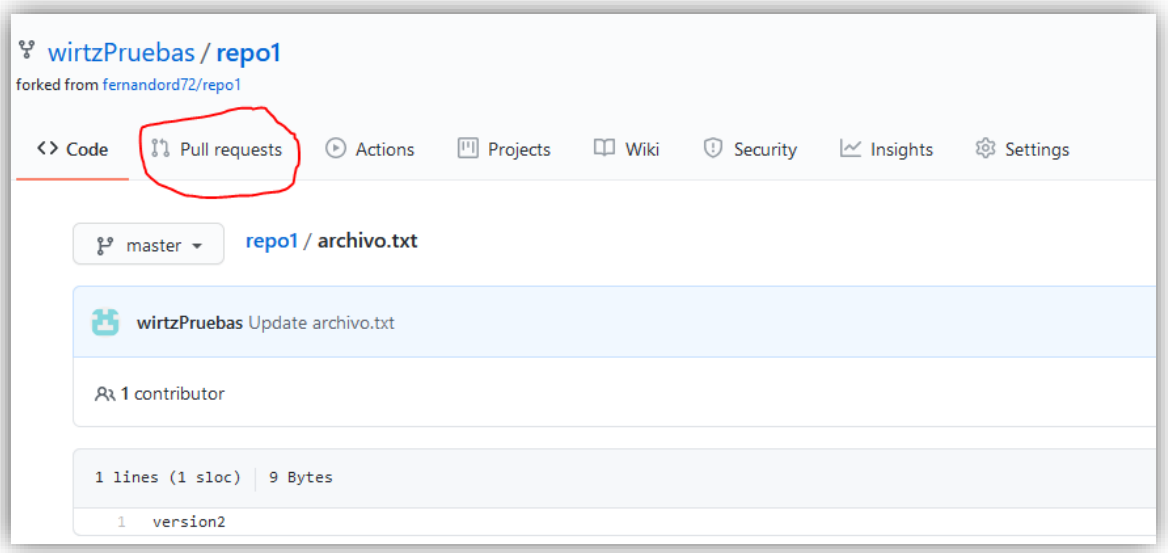
Premendo no arquivo e logo no botón de editar, podemos modifícalo desde aí mesmo, para facer os cambios de xeito rápido Poderíamos descárgalo a local, modifícalo e subir os cambios como fixemos no repositorio inicial.



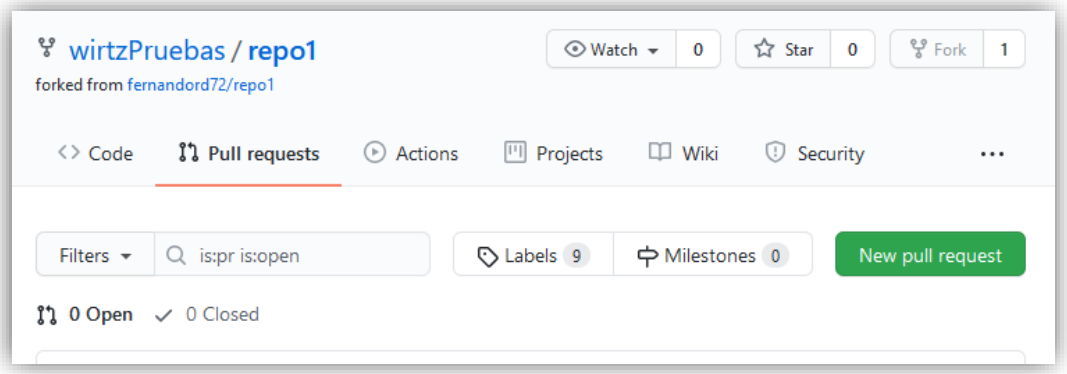
Modificamos e gardamos os cambios premendo *[Commit changes]* na parte baixa da pantalla.



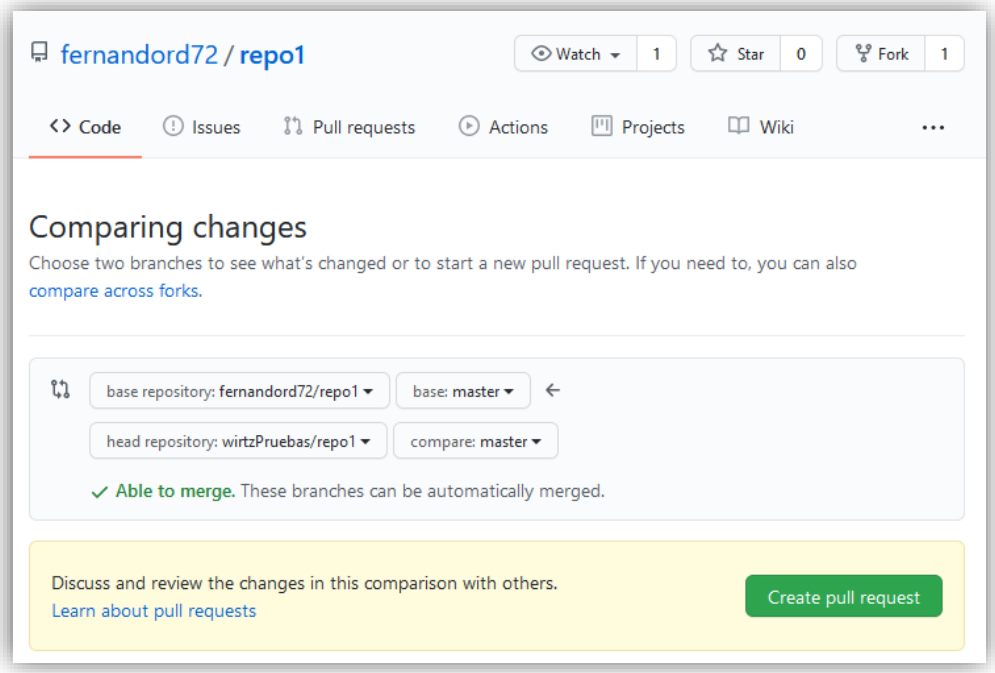
Agora xa se pode facer a “Pull Request” premendo no botón con ese texto:



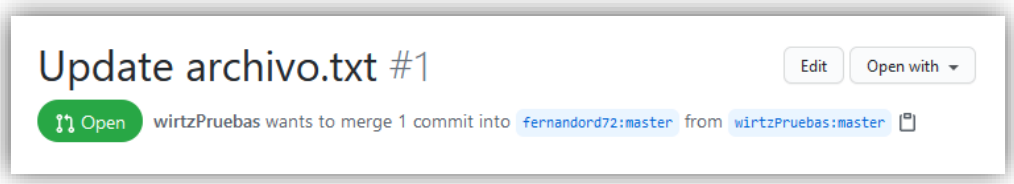
Chegando a xanela:



Prememos no botón verde *[New Pull Request]*, chegando a un resumo de como será a integración:



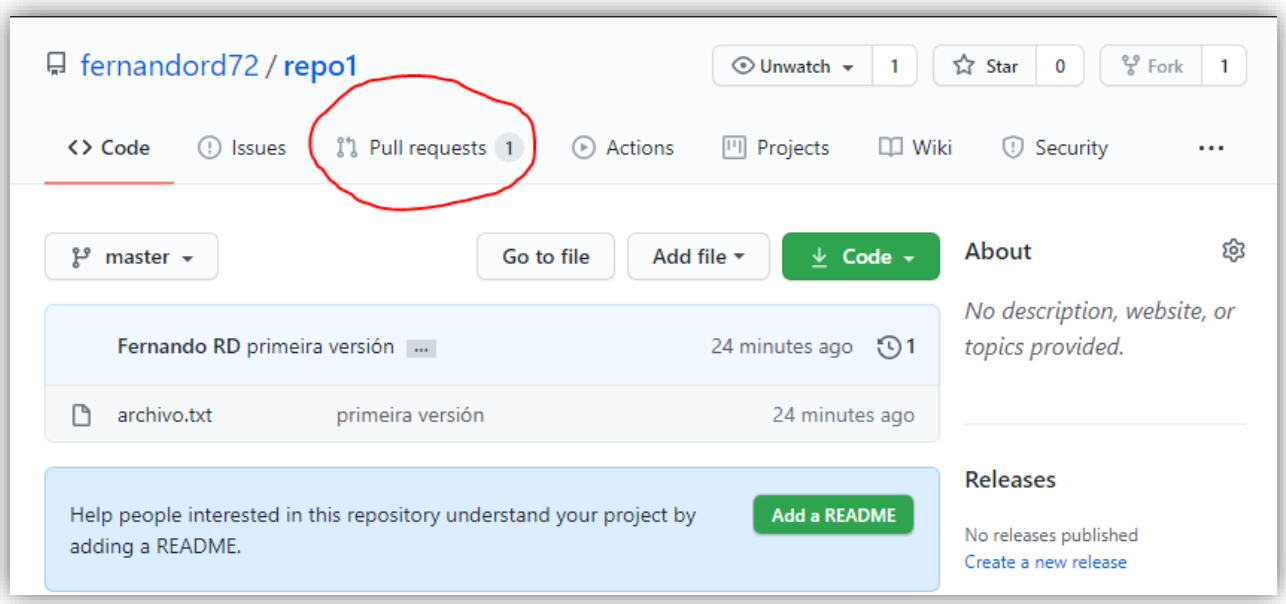
E confirmamos premendo *[Create pull request]*



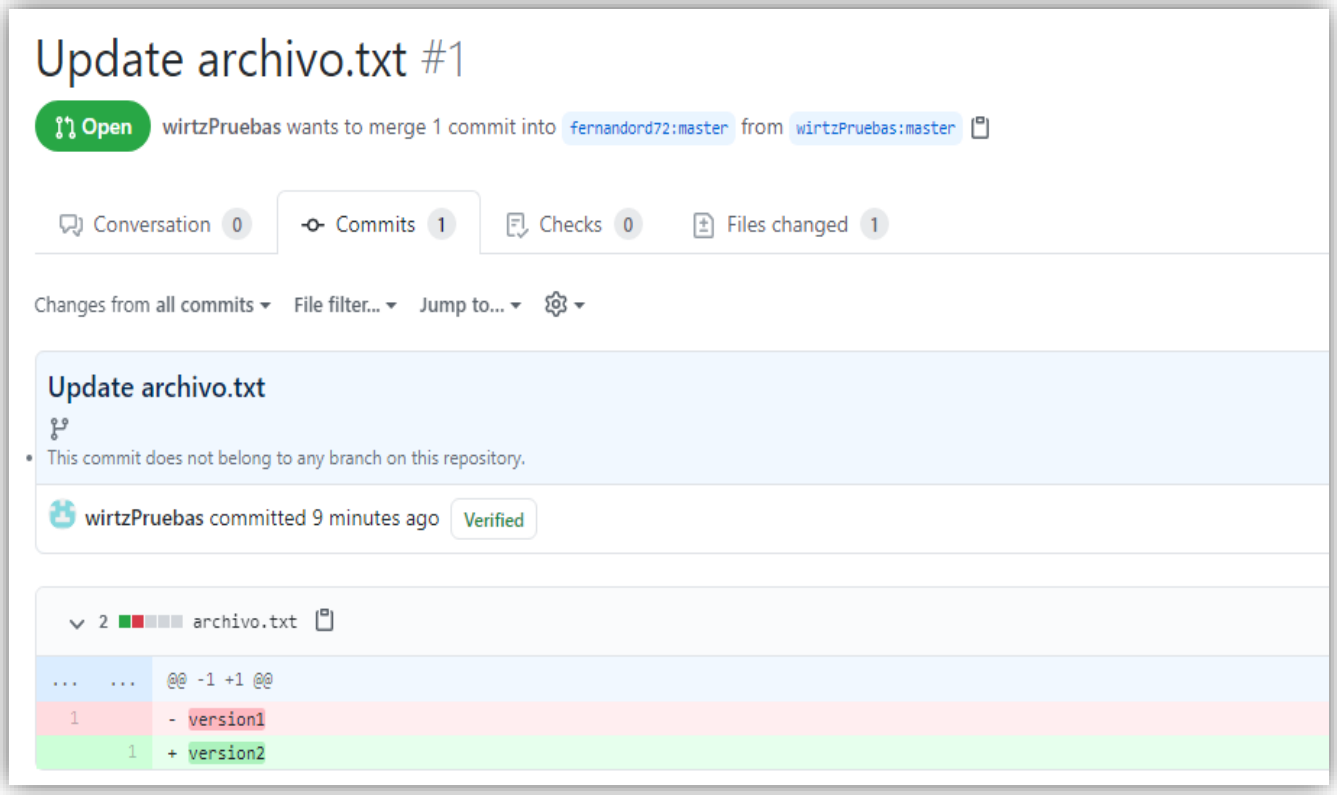
Aquí termina o traballo do usuario que fai a pull request *(wirtzPruebas)*.

Aceptar a Pull Request (merge)

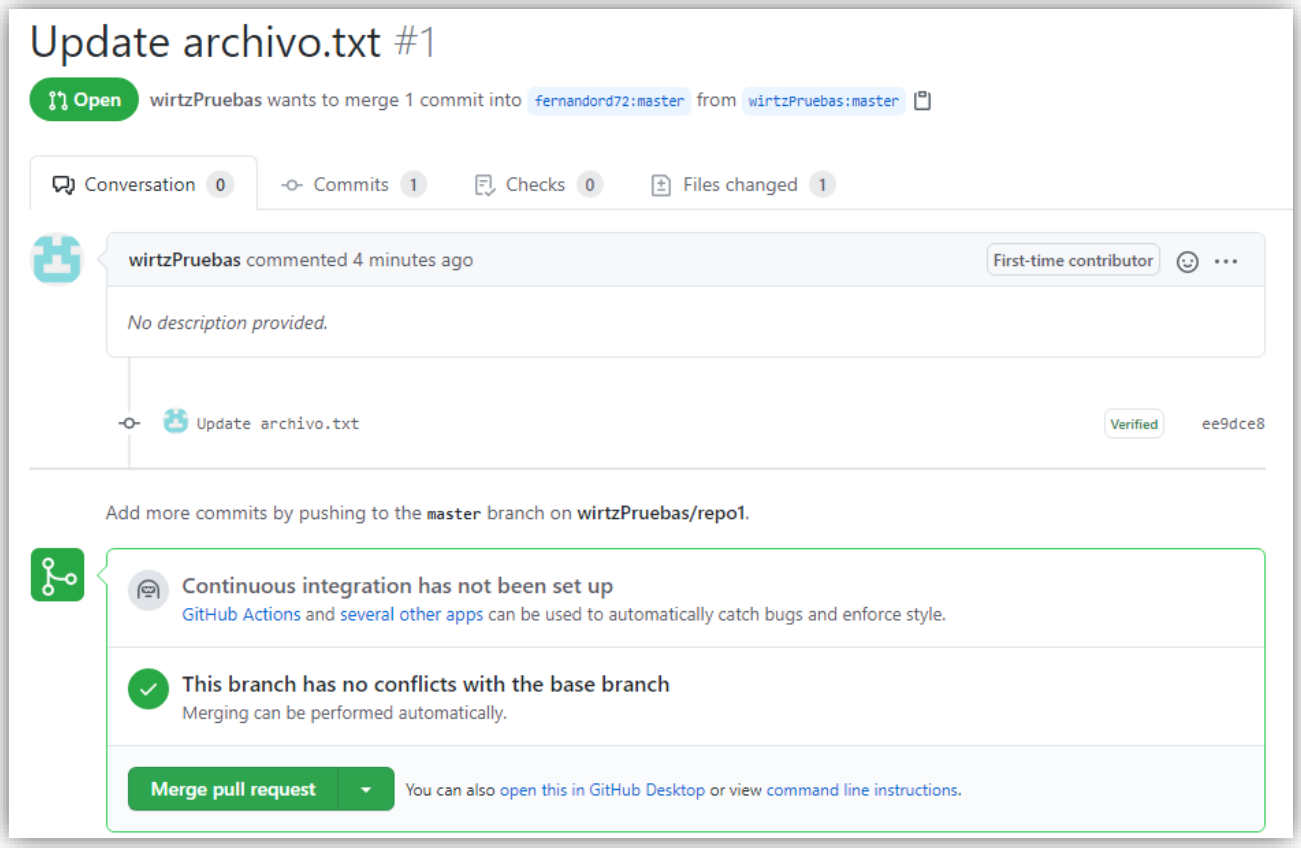
Cando faga login o usuario dono do repositorio *(fernandord72)* xa verá que ten unha pull request (tamén terá recibido un correo electrónico notificando a pull request):



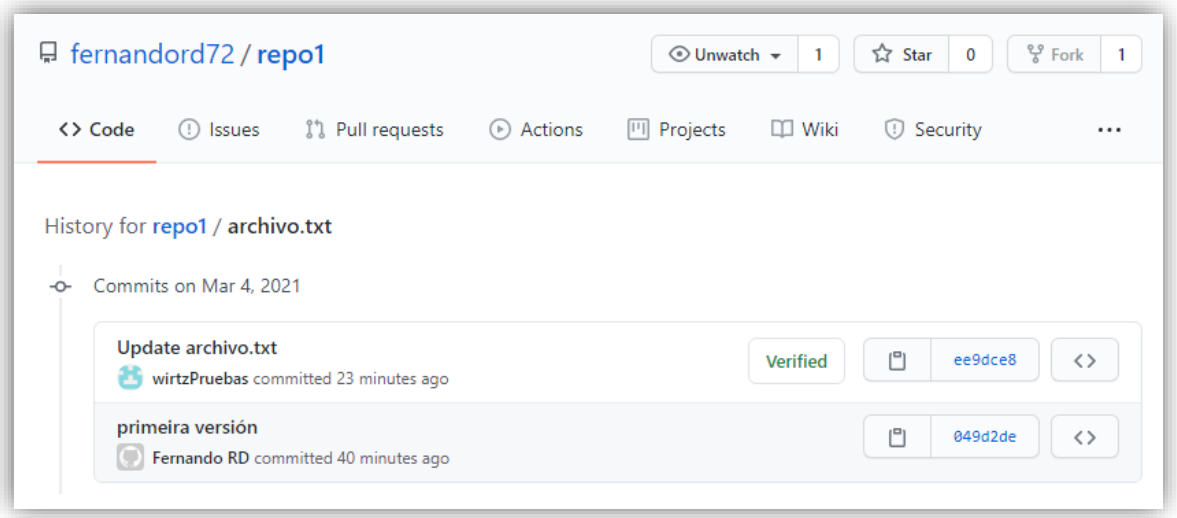
Podemos premer no arquivo para ver os cambios antes de aceptalos:



Finalmente, voltando a páxina anterior, hai que premer *[Merge pull request]* para integrar os cambios doutro usuario no neste repositorio:



Quedando os cambios integrados como un novo commit:

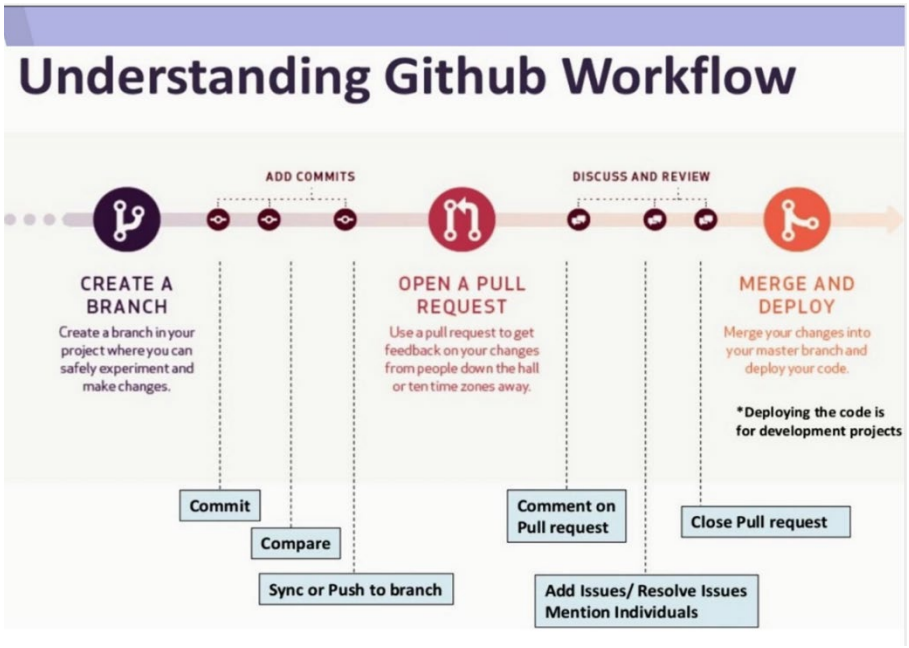


O usuario que fixo a pull request recibirá tamén un email comunicando o *merge.*

Tarefa 4.9. Operacións con GitHub

Crea unha conta en GitHub e despois:

a) Crea en local un repositorio como o da tarefa 4.5
b) Crea en GitHub un repositorio baleiro.
c) "Sube" o repositorio a GitHub
d) Elimina o repositorio local e descárgao con *clone.*
e) Crea unha rama en local con cambios como na tarefa 4.6
f) "Sube" a nova rama
g) En Github, non en local: Compara a nova rama e a MASTER e fai un merge.
h) Desde a consola git local, descarga a rama Master (pull) a local
i) Fai unha pull request" no repositorio de GitHub dalgún compañeiro (engadindo algún párrafo <p> ao arquivo *index.html*.
j) Comproba e acepta os "pull request" que che fagan os teus compañeiros.



3.8 Outros comandos

blame

Cando traballamos en grupo pode ser interesante saber que fixo algún cambio determinado, por exemplo para acordar coordinar os novos cambios (ou se é erróneo, para botarlle a culpa...). Para iso temos o comando **blame**.

```
git blame nomearquivo
```

podemos indicar quen modificou un arquivo pero nunhas liñas concretas.

```
git blame -L 12,14 src/Main.java
```

Cambiar o nome do repositorio remoto

Para cambiar o nome que lle dimos a un repositorio ao mapealo con "add origin" empregaríamos **remote rename** nomeRepo e para eliminalo (por exemplo se non colaboras máis nese proxecto) **remote rm** nomeRepo

Tags

Unha etiqueta ou **tag** é un nome que lle podemos asignar a unha determinada rama nun determinado momento (isto é, a un commit determinado). Poderíamos usalo para nomear versións do noso software, o para marcar "logros".

```
git tag -a v1 -m 'Release v0.1'
```

Logo será sinxelo voltar e esas versións (moi útil en proxectos *agile*, por exemplo para etiquetar o final de cada sprint semanal). Farémolo como nos cambiamos dunha rama a outra, co comando checkout.

```
git checkout v1
```

stash

Se estamos traballando no noso código e temos que cambiar a outra rama por calquera motivo pero non queremos facer un *commit* do noso traballo porque se atopa nun estado parcial podemos facer *stashing* como comando **stash**, isto é, "conxelar" os nosos cambios desde o último *commit* e almacenalo de forma temporal, quedando o directorio de traballo limpo, sen ningún commit pendente.

```
git stash
```

Agora poderíamos pasar á nova tarefa, e cando desexásemos retomar o traballo anterior:

```
git stash list
```

mostraría a lista de "stashes" e seleccionaríamos o que queiramos:

```
git stash apply stash@{x}
```

Sendo **x** o número mostrado na lista.

rebase

Este comando, que xa vimos para o *merge* de ramas, tamén permite "refundir" varios commits do historial nun so. Imaxinemos que nun proxecto grande fixemos o ano pasado 100 commits e

este ano outros 100. Pódenos interesar, por limpeza, xuntar os commits do ano pasado, por que xa non nos interesa telos por separado, nun so.

```
git rebase --interactive --root
```

A opción *interactive* permítenos seleccionar "a man" o rango de commits mediante unha pantalla e a opción *root* quere dicir que mostre desde o comezo dos tempos.

Esta perda de historial (perda de commits), pode ser perigoso, xa que nunca poderemos voltar á situación dos *commit* eliminados.

3.9 Control de versións en NetBeans con Git

NetBeans dispón dun complemento para poder utilizarse como cliente de Git. Os pasos a seguir para que NetBeans funcione como cliente de Git son:

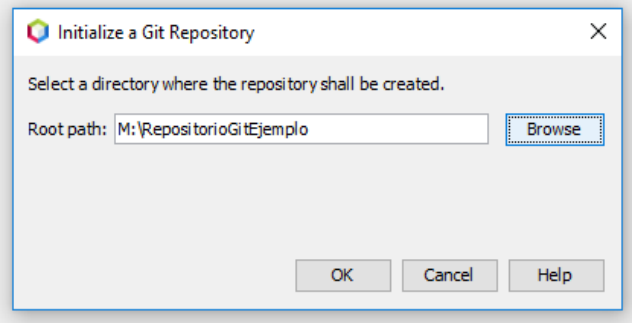
Inicialización do repositorio

- Pódese crear un repositorio dende un proxecto existente. Botón dereito sobre o proxecto > Versioning > Initialize Git Repository...
- Creación dun repositorio sen proxecto existente para realizar importación. Menú superior *Team > Git > Initialice Repository...*

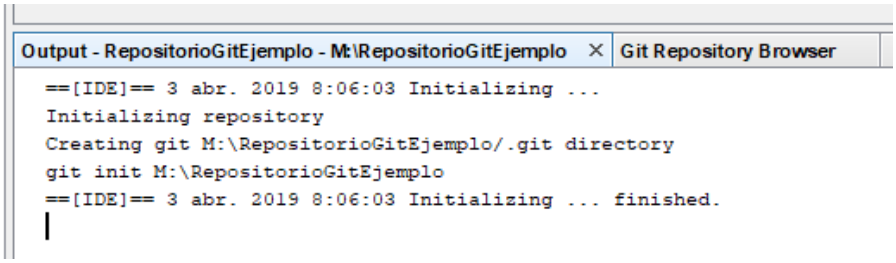
A partir deste momento, os cartafoles do proxecto poderán mostrar insignias de cores e os arquivos tamén o seu nome con distintas cores. Estas cores cambiarán segundo os estado do arquivo cando fagamos modificacións.

Selección da ubicación do repositorio.

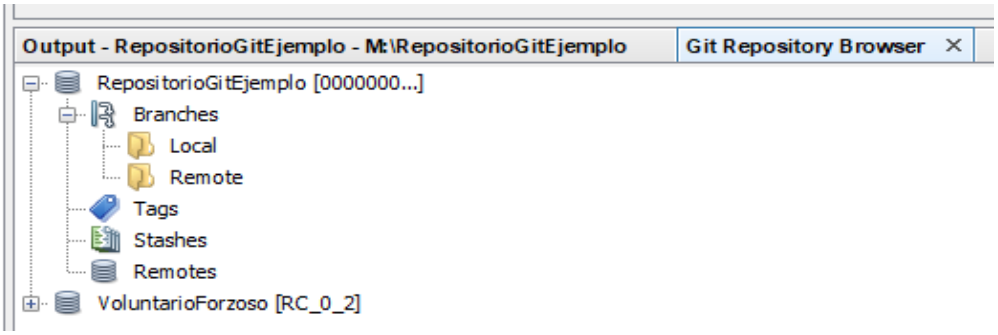
Soamente pode haber un repositorio dentro dunha carpeta.



Na saída de NetBeans podemos ver o comando utilizado para a creación do repositorio en local, así como a ubicación.



Na saída de repositorio podemos ver a estrutura básica que crea o proxecto.

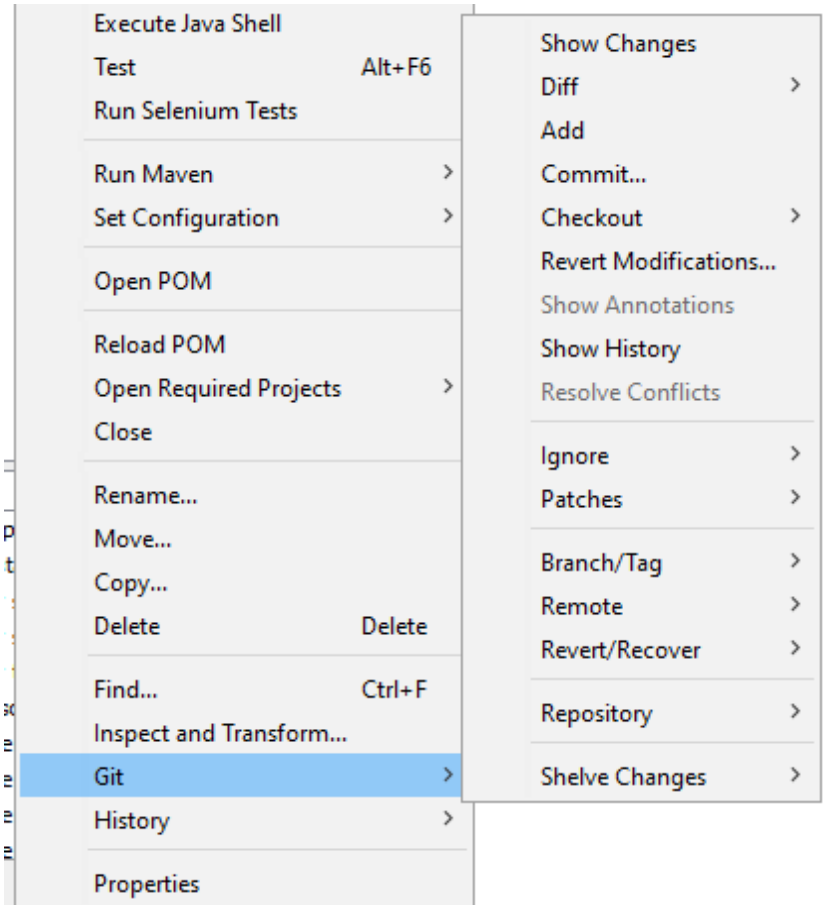


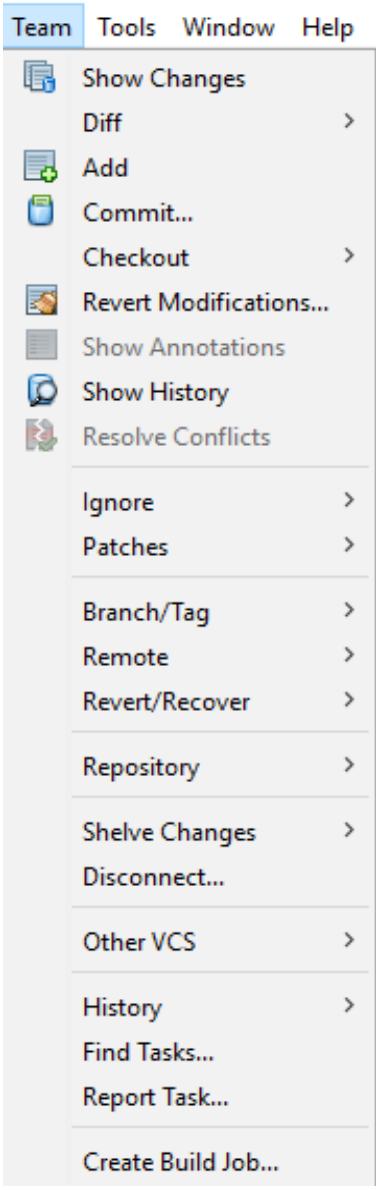
Configuración inicial

A configuración podemos vela e configurala no menú: *Team* > *Repository* > *Open Global Configuration.* Podemos ver o usuario e o email co que serán identificados os cambios realizados en todos os repositorios.

Accións dispoñibles

A forma de traballar con Netbeans e similar á vista previamente por interface de comandos pero accedendo a estas operacións mediante menús. Podemos acceder de dúas formas: desde o menú superior *Team* ou botón dereito sobre o Proxecto > *Git*

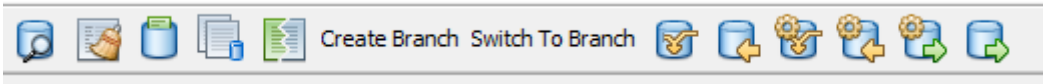




Podemos ver opcións coas que traballamos con comandos git, como por exemplo: *add*, *commit*, *checkout*, *branch*, etc.

Na opción *Remote* dispoñemos dos comando para traballar con repositorios remotos como: *push, fetch, pull,* etc.

No menú *View* podemos activar a barra de ferramentas de Git, para acceder dunha forma rápida aos comandos habituais.



Traballando en local

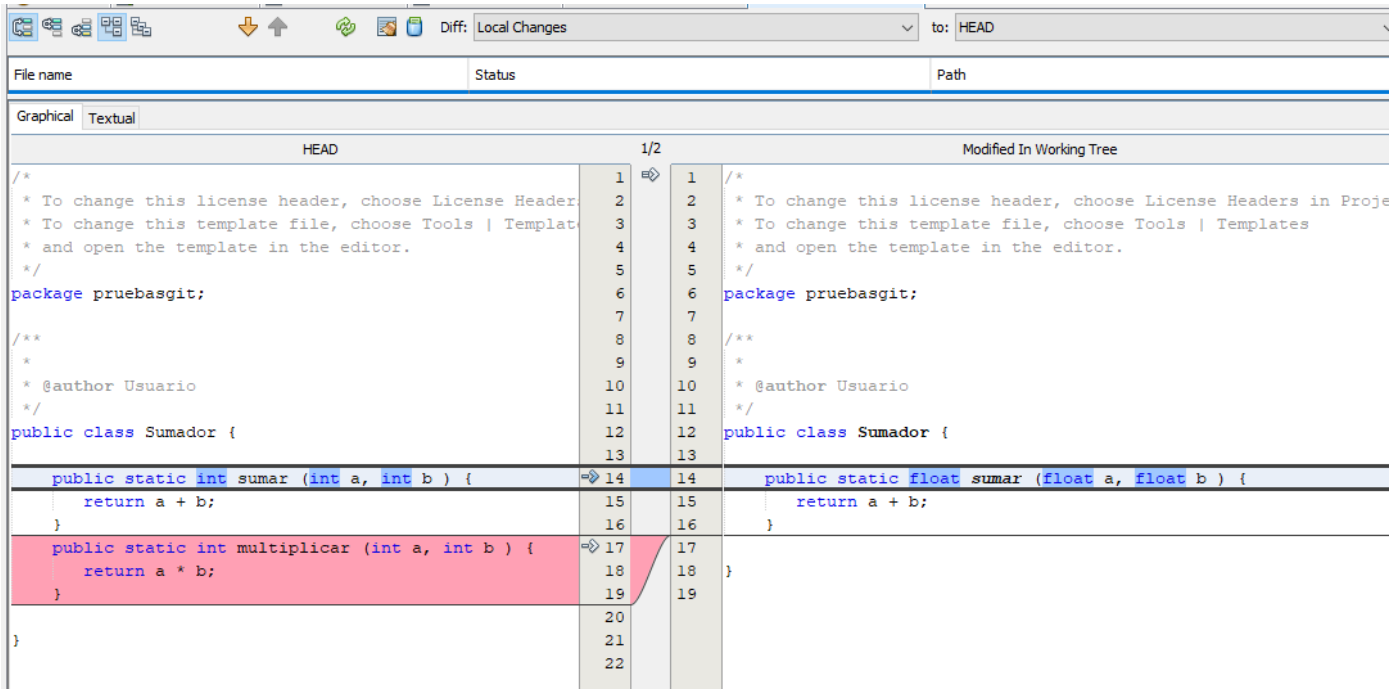
Cando traballemos en local, os novos arquivos mostraranse co nome en cor verde e os modificados en cor azul. Os que non se modifiquen, é dicir, os que están igual desde o último commit aparecerán en negro.

> Importante: Traballaremos igual que fixemos desde a liña de comandos, podendo facer commit, diff, checkout, etc. polo que non imos repetir aquí a funcionalidade de cada comando, simplemente algunha particularidade.

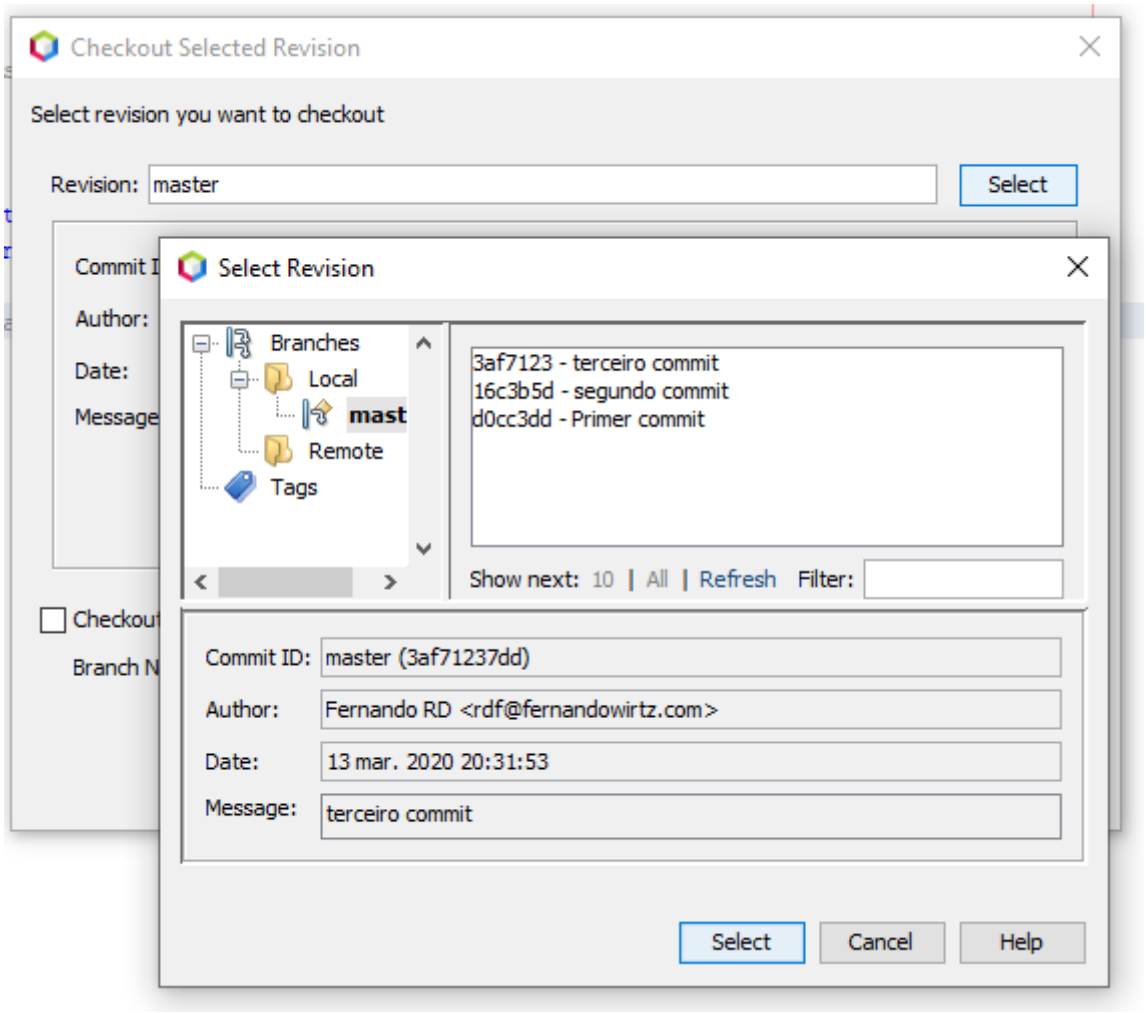
Á hora de facer un commit podemos facelo desde a zona *stage* (é dicir, dos arquivos que fixemos previamente *Add*) ou directamente de todos os arquivos modificados, fixeramos ou non o *Add.* Este comportamento controlámolo desde a ventá de *commit*, coas iconas:

Coa opción de menú *Diff* podemos comprobar as diferenzas entre as distintas versión dun arquivo. Hai que recordar que *HEAD* representa a última versión da que fixemos *commit,* é dicir, a versión "en produción", e *Working Tree* representa a versión na que estamos traballando.

En vermello mostra o texto eliminado e en azul o modificado ou engadido.



Para desfacer os cambios desenvolvidos no *working directory* empregaremos a opción de menú *checkout* como vimos desde liña de comandos, permitindo ben recuperar a versión da zona commit (HEAD) coa opción de menú *Checkout files,* ben a algunha versión de *commits* intermedios coa opción de menú *checkout revisión.*



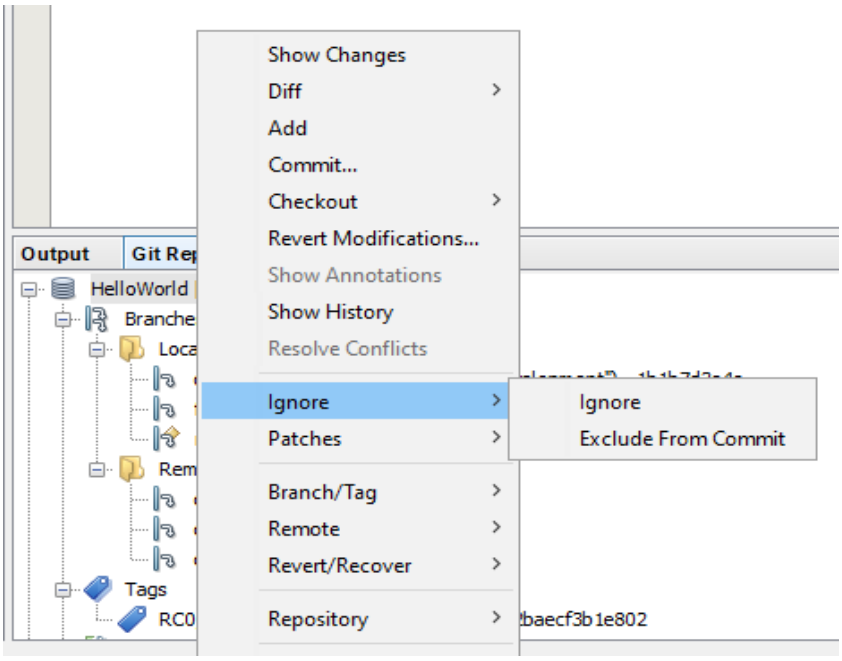
Mediante a opción de menú *Show History* podemos ver a información completa de todas as situacións polas que pasou cada arquivo.

Xestión de ramas

O concepto de rama é o mesmo que o que xa vimos na consola de Git. Desde o menú *git* > *branch* temos a opción de crear ramas *(create branch)* como de movernos a elas *(switch branch)*

Ignorar arquivos para no repositorio

Os arquivos que son propios do entorno de desenvolvemento, ou os arquivos que conteñen datos de configuración non se deben sincronizar (como moito unha plantilla dos posibles cambios con valores de proba).



Traballar cun servidor remoto.

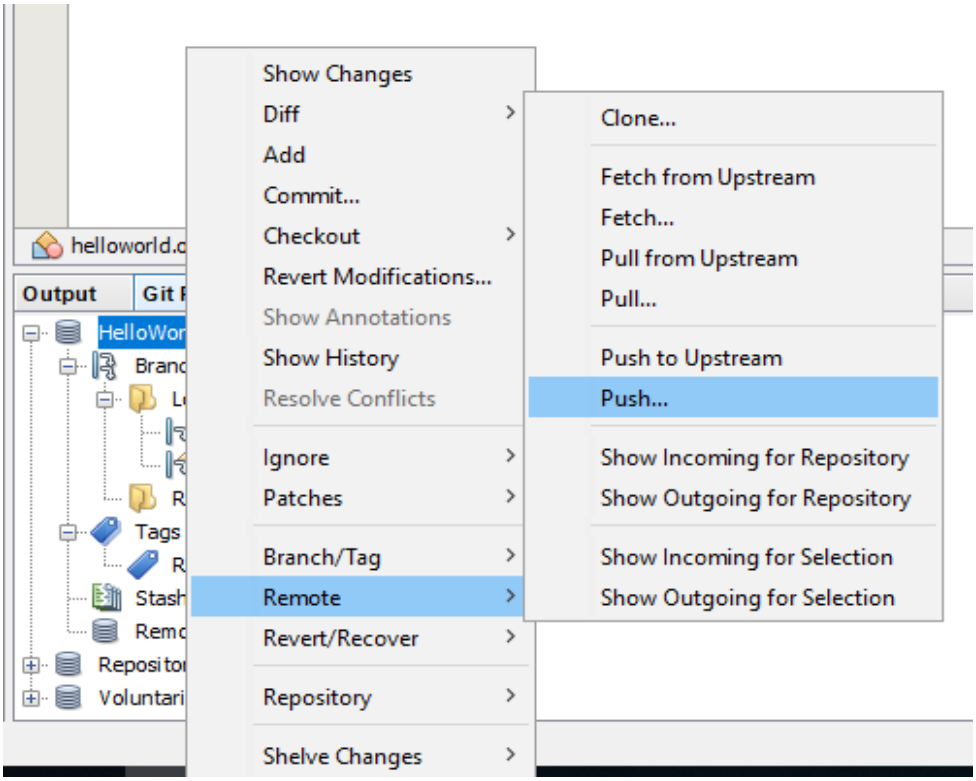
Ata agora, todos os cambios realizáronse en local, polo que é momento de aplicalos no servidor centralizado para que o resto do equipo poida velos.

Se xa temos un repositorio en GitHub:

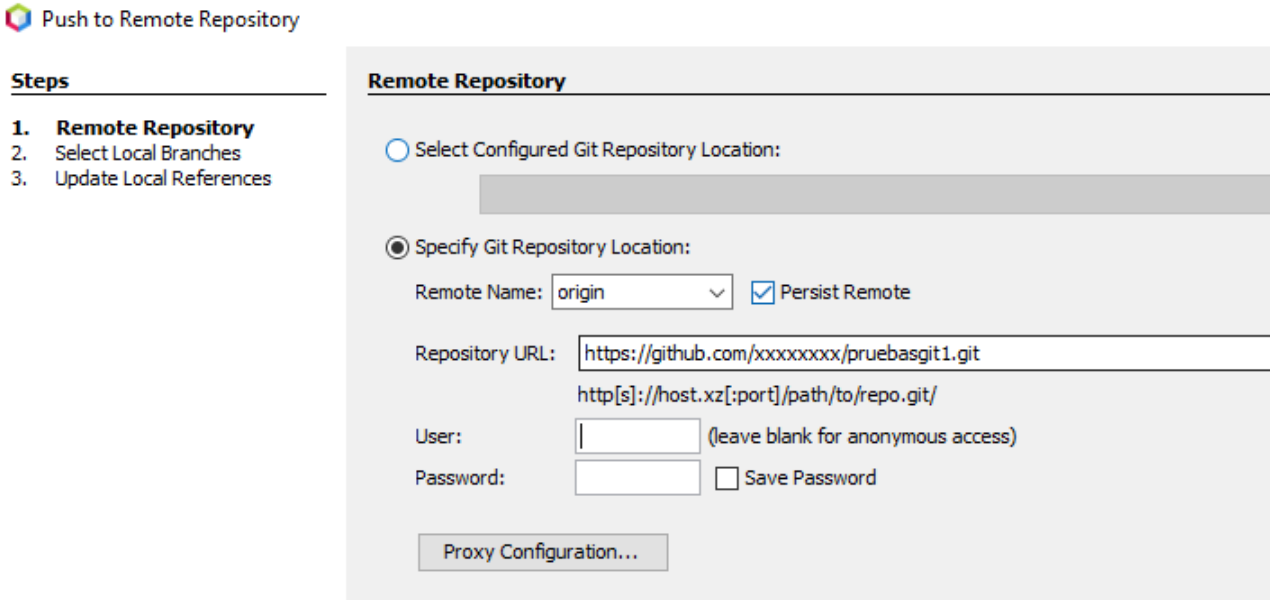
- Na túa conta en GitHub, vai ao teu repositorio e pulsa *[Clone]* para obter URL do repositorio.
- E logo, xa en NetBeans: *Team* > *Git* > *Clone*. Aí enchemos a URL do noso repositorio remoto, usuario, contrasinal e cartafol destino se é preciso.
- No derradeiro paso deste proceso ofrécenos crear un proxecto a partires do repositorio remoto, contestamos que si.
- Agora traballaremos co proxecto local como comentamos nos apartados anteriores e cando queiramos subir os nosos commits a GitHub, premeremos con botón dereito sobre o proxecto > *Git*> *Remote* > *Push*. Hai que ter en conta, que os commits que se fagan en local, mentres non se realice o push, non quedan reflexados no servidor externo.

Se queremos copia o noso proxecto local a un repositorio en GitHub baleiro:

- Creamos en GitHub un repositorio, sen contido. Ao crealo obteremos unha URL.
- No noso proxecto en Netbeans: Git > Remote > Push:



- Agora debemos indicar a URL do noso repositorio en GitHub e as nosas credenciais:



- E finalmente indicamos as ramas as sincronizar, xeralmente todas ou a *master.*
- Tamén, se un usuario realiza cambios no repositorio remoto, podemos "baixar" eses cambios ao noso repositorio local con botón dereito sobre o proxecto > *Git* > *Remote* > *Pull.*

Tarefa 4.10. Git con Netbeans

a) Crear un proxecto con dous arquivos: unha clase coma a seguinte:

```
public class Sumador {
  public static float sumar (float a, float b ) {
     return a + b;
}}
```

e un programa que invoca algún ao seu métodos

```
public class NewMain {
    public static void main(String[] args) {
        System.out.println(Sumador.sumar (15f,4f));
 }}
```

Fai que o proxecto teña control de versións con git. Que cor teñen os nomes dos arquivos?

b) Fai *commit* do proxecto. Que cor teñen agora os nomes dos arquivos?

c) Fai *push* do proxecto a un novo repositorio na túa conta en GitHub.

d) Crea en local unha rama chamada "dev" na que, elimina a clase *Sumador* do punto anterior e crea unha nova chamada *Restador:*

```
public class Restador {
  public static float restar (float a, float b ){
         return a - b;
}}
```

Na clase *NewMain* que contén o main() chama ao novo método en vez de ao anterior:

```
System.out.println(Restador.restar (15f,4f));
```

Fai commit e push da nova rama.

e) Proba en Netbeans a facer git > Branch > Swith to... para cambiar dunha rama a outra e verás como na ventá de proxectos ves o arquivo *Sumador.java* ou *Restador.java* segundo a rama na que esteas.

f) Vai á túa conta de GitHub e fai dúas capturas de pantalla, unha na que se vexa a rama *master* e outra na que se vea a rama *dev.*

g) En Netbeans, comproba as diferenzas entre a clase NewMain da versión *master* coa da versión *dev.*

h) Crea unha rama chamada "hotfix" a partires da *master* na que cambias a liña:
`System.out.println(Sumador.sumar (15f,4f));` por:
`System.out.println(Sumador.sumar (16f, 4f));`

i) Fai commit da rama *hotfix* e aplica os cambios sobre a rama *master.* É posible *fast forward?*

j) Aplica os cambios da rama *dev* sobre master. Se hai conflitos, resólveos.

k) Fai push de todas as ramas ao repositorio remoto.

l) Mostra un histórico cos cambios feitos.

5. Documentar clases

A ferramenta javadoc é un xerador de documentación que extrae información sobre as clases, clases internas, métodos, interfaces, e campos baseándose nos comentarios Javadoc e no código fonte, e que gardará nun grupo de arquivos HTML que poderá ser consultado facilmente.

Este proceso de documentación pode facerse directamente con javadoc na liña de comandos ou o que é máis cómodo, pode utilizarse un IDE como NetBeans que mediante un contorno gráfico integrado con outras tarefas de programación permite documentar baseándose na ferramenta estándar javadoc, é dicir, o que chamaremos crear documentación Javadoc.

Baseándose nun proxecto Java de NetBeans farase a creación da documentación Javadoc en dúas etapas:

- Inserir comentarios Javadoc no código fonte.
- Xerar documentación Javadoc para un proxecto. Esta documentación é unha especificación da API (Application Programming Interface) do proxecto que contribuirá a unha mellor comprensión do proxecto por parte do programador. Un exemplo desta documentación é a https://docs.oracle.com/en/java/javase/21/docs/api/index.html

Despois de ter a documentación e de que NetBeans coñeza onde está gardada esa documentación, poderase consultar directamente dende o arquivo fonte. Un paso previo para que NetBeans coñeza onde está gardada a documentación de Java SE é agregala a NetBeans.

Se á documentación Javadoc xerada (ou especificación API) se lle engaden exemplos, definicións de termos comúns de programación, a descrición de erros e as súas solucións, teríase unha guía de programación do proxecto. Un exemplo desta guía é o paquete de documentación de JDK. Cabe destacar que na descrición de erros da guía de programación distínguese normalmente entre:

- Erros de especificación API presentes na declaración de métodos ou nos comentarios que afectan á sintaxe ou a semántica.
- Erros de código que se poden producir na implementación. Normalmente distribúense separados nun informe de erros. Sen embargo, pódense incluír en comentarios Javadoc utilizando a etiqueta *@bug*.

5.1 Agregar documentación Javadoc a NetBeans

Algunhas operacións relacionadas coa documentación Javadoc en NetBeans requiren que se agregue a documentación Javadoc a NetBeans. No caso de non estiver, faise en dous pasos:

- Descargar o arquivo de documentación Java SE.
- Agregar o arquivo descargado a NetBeans.

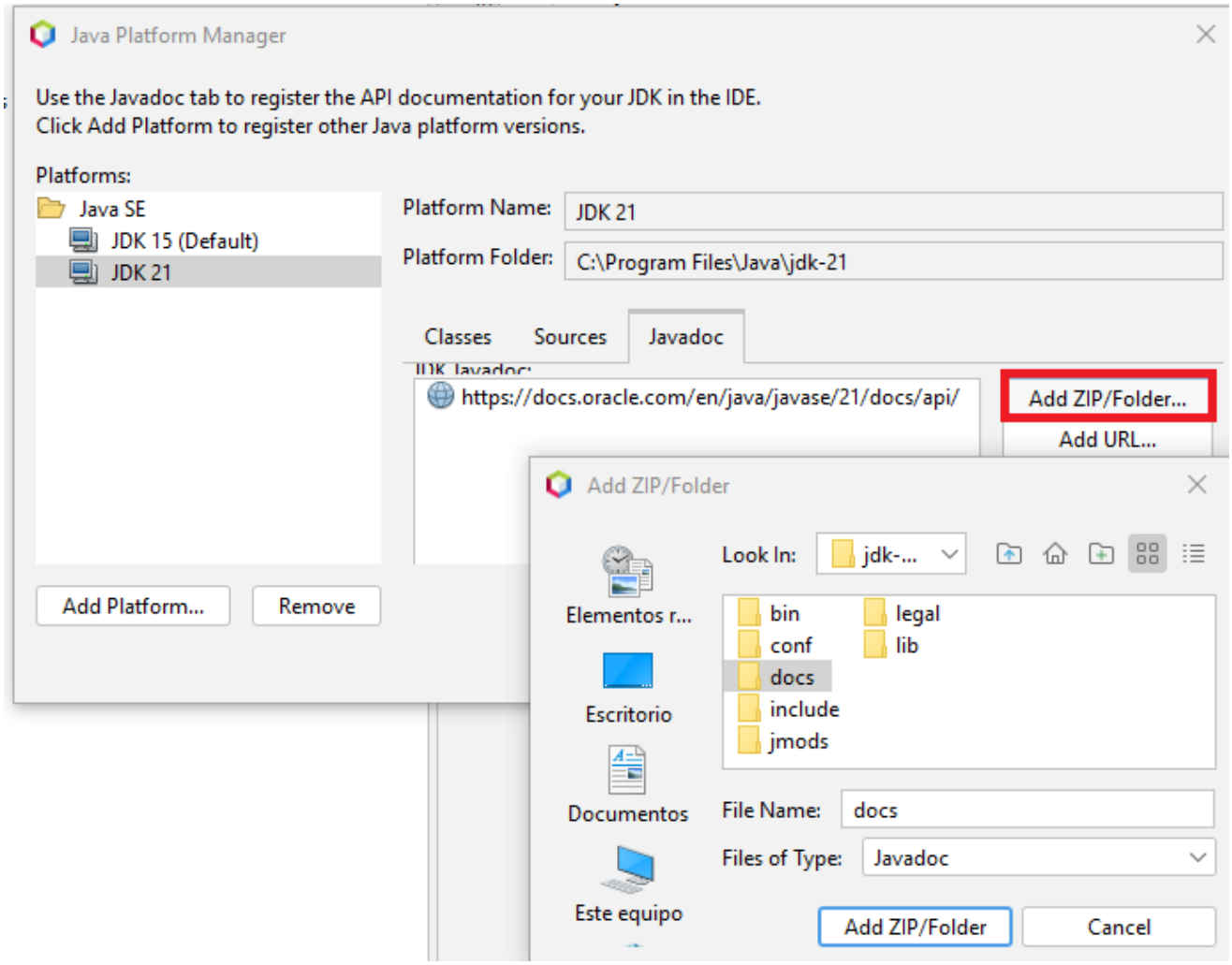
Descargar Java SE Documentation

Descárgase Java SE Development Kit 21 Documentation (se non está incluida co JDK), da páxina web de Oracle https://www.oracle.com/java/technologies/javase-jdk21-doc-downloads.html e premendo en *Document Download.* O arquivo *.zip* pódese gardar en calquera localización, por exemplo nun novo cartafol *"docs"* en *C:\Program Files\Java\jdk-21\docs*.

Agregar Java SE Documentation a NetBeans

No menú principal, elíxese a opción Tools->Java Platforms. Aparece a ventá Administrador de Java Platform na que se debe de:

- Paso 1. Seleccionar a plataforma a administrar.
- Paso 2. Elixir a pestana Javadoc,
- Paso 3. Premer no botón de Agregar arquivo zip.
- Paso 4. Aparecerá unha nova ventá na que poderá elixir o arquivo que se descargou no paso anterior e que aparecerá reflectido no apartado Javadoc de la plataforma:
- Paso 5. Premer no botón de Close.



5.2 Comentarios Javadoc no código fonte

Os comentarios Javadoc serán aqueles comentarios Java (empezan por /** e finalizan por */) que serán utilizados por Javadoc para xerar a documentación dun proxecto.

Forma

Os comentarios Javadoc empezan por /** e finalizan con */ e poden incluír texto, código HTML e etiquetas Javadoc. A forma clásica de colocación destes comentarios e adaptada pola maioría dos programadores Java é a seguinte:

- Tódalas liñas de comentario teñen a mesma sangría que o código que documenta.
- A primeira liña do comentario só contén /**.
- As seguintes liñas empezan por espazo en branco, * e espazo en branco seguidos do texto do comentario.
- A última liña de comentario só contén espazo en branco e */.

```
/**
* Texto con alguna descripción (posibilidad de usar HTML)
* Etiquetas javadoc (comienzan por @)
*/
```

Etiquetas

As etiquetas Javadoc

https://docs.oracle.com/en/java/javase/21/docs/specs/javadoc/doc-comment-spec.html son expresións que empezan por @ e que se colocan dentro dos comentarios Javadoc. A primeira versión de Javadoc é Javadoc 1.0 e posteriormente foron xurdindo novas versión con novas etiquetas. As etiquetas máis usadas son as seguintes e colócanse normalmente seguindo a orde:

Etiqueta	Descrición
@author nome_autor	Só para clases e interfaces. Utilízase para indicar o nome ou nomes de programadores. Os nomes pódense separar por comas.
@version versión	Só para clases e interfaces. Utilízase para describir a versión.
@param nome_parametro descrición	Por convención deberíase de poñer sempre. Utilízase para describir os parámetros de cada método ou construtor.
@return descrición	Para describir o dato de retorno de cada método (non construtor). Non se usa con métodos void e por convención debería de poñerse sempre nos outros casos.
@throws nome_clase descrición	Utilízase para describir unha excepción lanzada por un método. Outra etiqueta sinónima é: @exception nome_clase_excepcion descrición
@see paquete.clase#membro texto	Utilízase para que no documento HTML xerado apareza un enlace na sección "See Also" (ou un texto con enlace) á documentación doutro paquete, clase, método ou campo. NetBeans guiará na descrición da referencia mediante axuda en liña. O membro pode ser un método ou un campo.
@since texto	Utilízase para indicar a versión do produto no que foi engadido o elemento.

Existen tamén as etiquetas en liña que se encerran entre { e } como por exemplo:

Etiqueta	Descrición
{@code texto}	Aparece na documentación HTML como <code>texto</code>
{@docRoot}	Representa a localización do directorio raíz do sitio HTML xerado. Esta etiqueta utilízase para incluír un arquivo, como unha páxina de copyright ou un logo.
{@inheritDoc}	Permite copiar documentación da clase máis próxima da que herda ou da interface implementada máis próxima ao sitio da etiqueta.

Etiqueta	Descrición
{@link paquete.clase#membro texto }	É igual que @see, pero xera o enlace en liña en lugar de colocalo na sección "See Also".
{@linkplain paquete.clase#membro texto}	Igual cá anterior pero xera o enlace en texto plano en lugar de dentro da etiqueta <code>. Neste caso, <b> non se interpretaría como etiqueta HTML.
{@literal texto}	Permite ver o texto de forma literal sen ser interpretado como texto HTML.
{@value [paquete.clase#constante]}	Mostra o valor dunha constante. Pódese utilizar sen nome da constante cando está dentro do comentario da constante.

```
/**
 * Clase Empleado
 *
 * Contiene informacion de cada empleado
 *
 * @author Patricia
 * @version 1.0
 */
public class Empleado {
    /**
     * Nombre del empleado
     */
    private String nombre;
    /**
     * Apellido del empleado
     */
    private String apellido;
    /**
     * Edad del empleado
     */
    private int edad;
    /**
     * Salario del empleado
     */
    private double salario;

    /**
     * Suma un plus al salario del empleado si el empleado tiene mas de 40 años
     * @param sueldoPlus cantidad a sumar al salario
     * @return <ul>
     *          <li>true: se suma el plus al sueldo</li>
     *          <li>false: no se suma el plus al sueldo</li>
     *          </ul>
     */
    public boolean plus (double sueldoPlus){
        boolean aumento=false;
        if (edad>40 && compruebaNombre()){
            salario+=sueldoPlus;
            aumento=true;
        }
        return aumento;
    }

    /**
     * Comprueba que el nombre no este vacio
     * @return <ul>
     *  <li>true: el nombre es una cadena vacia</li>
     *  <li>false: el nombre no es una cadena vacia</li>
     *  </ul>
     */
    private boolean compruebaNombre(){
        if(nombre.equals("")){
            return false;
        }
```

```
        return true;
    }


    /**
     * Constructor por defecto
     */
    public Empleado(){
        this ("", "", 0, 0);
    }

    /**
     * Constructor con 4 parametros
     * @param nombre nombre del empleado
     * @param apellido nombre del empleado
     * @param edad edad del empleado
     * @param salario salario del empleado
     */
    public Empleado(String nombre, String apellido, int edad, double salario){
        this.nombre=nombre;
        this.apellido=apellido;
        this.edad=edad;
        this.salario=salario;
    }
}
```

Estilo

Normalmente, por convención, os comentarios seguen unhas regras de estilo:

- Usar a etiqueta HTML <code> para as palabras claves e os nome, é dicir, nomes de paquetes, clases, métodos, interfaces, campos, exemplos, palabras clave de Java.
- Restrinxir, dentro do posible, o uso de {@ link URL} xa que dificultan a lectura da documentación.
- Omitir parénteses cando se utiliza a forma xeral de métodos ou construtores e só utilizalos cando se quere facer referencia a unha específica forma.
- Utilizar breves descricións sobre todo no resumo inicial e nas descricións dos parámetros.
- Utilizar a terceira persoa dos tempos verbais. Por exemplo na descrición dun método: Obtén o valor da área do círculo
- Empezar as descricións cun tempo verbal. Por exemplo na descrición dun método: Obtén o valor da área do círculo en lugar de Este método permite obter o valor da área do círculo.
- Nas descricións de clase, interface ou campo omitir o suxeito. Por exemplo: Etiqueta de botón en lugar de Este campo é unha etiqueta de botón.
- Usar este en lugar de o artigo o para referirse a un obxecto da presente clase. Por exemplo: Obtén o conxunto de ferramentas para este compoñente en lugar de Obtén o conxunto de ferramentas para o compoñente.
- Engadir na descrición algo máis co nome. Os mellores nomes son os que se documentan a si mesmos, pero na descrición debe haber máis información que a simple repetición da información que dá o nome. Por exemplo para o método public void establecerX(int valorX) é mellor poñer:

```
/**
 * Establece el valor de la coordenada x.
 * Se admite cualquier valor entero
 * @param valorX valor de la coordenada x
```

```
*/
   x=valorX;
```

que poñer establece X que só repetiría o nome do método.

- Utilizar a palabra campo para referirse a variables de clase e utilizar as palabras campo de fecha, campo numérico ou campo de texto para referirse aos obxectos correspondentes da clase TextField.
- Non utilizar abreviaturas non estándar na descrición. Por exemplo utilizar por exemplo en lugar de p.e.

Localización

Os comentarios Javadoc colócanse xusto antes da definición do elemento que comentan; campos, métodos, interfaces e clases.

Por exemplo, os comentarios de clase xusto antes da definición da clase.

```
1   package circulo;
2
3   /**
4    * Clase <b>Circulo</b> para pruebas en NetBeans
5    * @author profesor
6    * @version 1.0
7    */
8   public class Circulo {
```

Por exemplo, os comentarios de campos xusto antes da declaración de cada campo.

```
9       /**
10       * coordenada x
11       */
12      private int x;
13      /**
14       * coordenada y
15       */
16      private int y;
```

```
22      /**
23       * Constructor para la clase Circulo que asigna los valores de las
24       * coordenadas x, y y el valor del radio
25       * @param valorX valor de la coordenada x
26       * @param valorY valor de la coordenada y
27       * @param valorRadio valor del radio
28       */
29      public Circulo(int valorX, int valorY, double valorRadio) {
30          establecerX(valorX);
31          establecerY(valorY);
32          establecerRadio(valorRadio);
33      }
```

Por exemplo, os comentarios de método e interfaces xusto antes da declaración da firma.

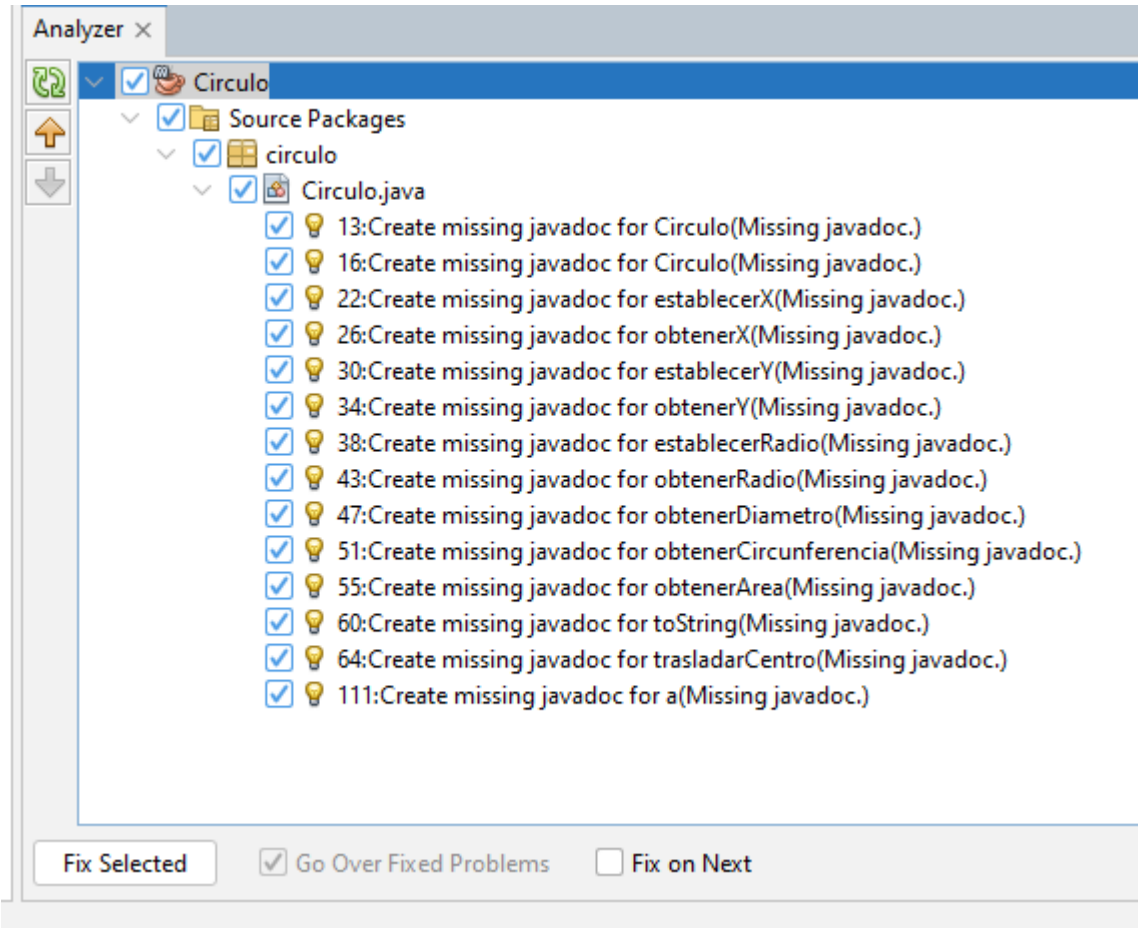
Analizar Javadoc estaticamente

NetBeans permite analizar estaticamente o código fonte e emitir informe sobre como mellorar a documentación Javadoc. A análise pódese facer de dúas maneiras:

- Na ventá de proxecto, seleccionar o arquivo a analizar, facer clic dereito e elixir *Generate Javadoc*
- No menú principal e co cursor na ventá de edición do código fonte a analizar, elixir *Tools→ Analyze Javadoc*.

En calquera dos dous casos anteriores, ábrese unha pestana Analyzer co informe dos problemas de documentación encontrados e na que se pode ver:

- Unha barra de ferramentas á esquerda que permiten actualizar o informe sobre a análise, ir ó anterior problema ou ir ó seguinte problema.
- A lista de problemas encontrados coa liña na que se encontrou o problema e unha breve descrición do mesmo. Os problemas poden ser seleccionados para ser reparados.
- Unha parte inferior na que se poden reparar os problemas marcados, que permita volver a examinar problemas reparados e na que se pode seleccionar a reparación do seguinte problema.



De marcar algún dos problemas e premer en Reparar seleccionado, aparece inserido no código a estrutura básica do comentario Javadoc, incluíndo etiquetas Javadoc cando sexa posible. Por exemplo pode aparecer @param se é un método con parámetros, ou @return cando é un método diferente de void ou non aparecer ningún código e só aparecer a estrutura de comentario cando é un método void que non ten parámetros. Despois de reparalo, verase na ventá Analyzer desmarcado e coa descrición tachada.

```
64
65      /**
66       * Método que permite establecer el valor del radio
67       *
68       */
         public void establecerRadio(double valorRadio) {
70
71          radio = (valorRadio < 0.0 ? 0.0 : valorRadio);
72      }
73
```

Find: overr

circulo.Circulo > establecerRadio

Output - Circulo (javadoc) | Analyzer

- Circulo
  - Source Packages
    - circulo
      - Circulo.java
        - 68:Add @param valorRadio tag(Missing @param tag for valorRadio)
      - Main.java
        - 11:Create missing javadoc for Main(Missing javadoc.)

A operación de Reparar seleccionado realiza a primeira parte da reparación do problema pero é o programador quen deberá de revisar e completar eses comentarios e para iso conta coa axuda en liña de NetBeans que permite visualizar a lista de posibles códigos Javadoc axeitados cando se teclea @ dentro da estrutura de comentario Javadoc.

Despois de solucionar tódolos problemas de documentación Javadoc que detecta NetBeans, e despois de actualizar a ventá Analyzer, esta debe aparecer baleira.

```
61      public int obtenerY() {
62          return y;
63      }
64
65      /**
66       * Método que permite establecer el valor del radio
67       *
68       * @param valorRadio Valor del radio
69       */
70      public void establecerRadio(double valorRadio) {
```

Find: overr

circulo.Circulo

Output - Circulo (javadoc) | Analyzer

5.3 Xerar documentación Javadoc

NetBeans permite crear automaticamente documentación Javadoc para un proxecto, é dicir, xera un conxunto de páxinas HTML que describan as clases, interfaces, construtores, métodos e campos dun proxecto, a partir do código fonte e dos comentarios Javadoc embebidos no código.

Os pasos a seguir son:

- Seleccionar o proxecto na ventá de proxectos.
- Xerar Javadoc. Pódese facer de calquera das dúas maneiras seguintes:
  - Run→Generate Javadoc(Circulo) do menú principal ou
  - Facer clic dereito e elixir Generate Javadoc.
- A documentación Javadoc xerada verase en primeira instancia no navegador externo pero tamén se poderá ver no IDE dende os arquivos fonte e utilizando a ventá Javadoc, ou no IDE utilizando o índice de busca de Javadoc.

Javadoc ante os seguintes casos particulares:

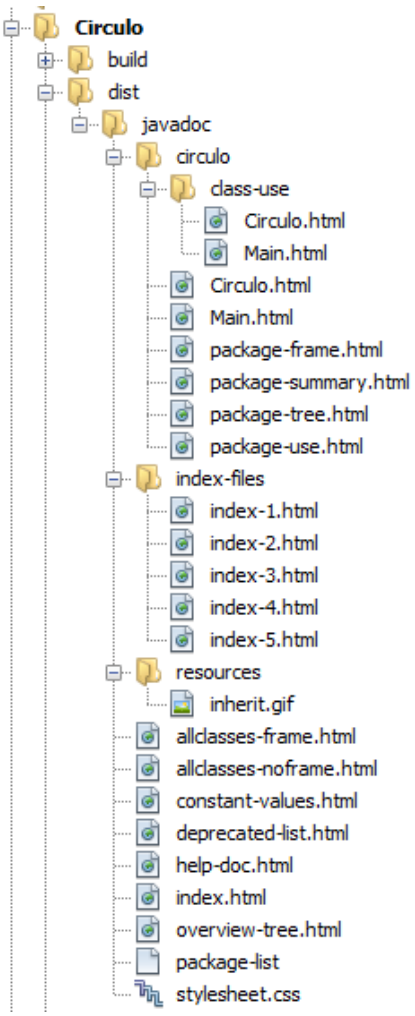
- un método nunha clase sobrescribe un método dunha superclase,
- un método nunha interface sobrescribe un método nunha superinterface ou
- un método nunha clase implementa un método dunha interface,

que resolve:

- por defecto xerando un Overrides na documentación para o método e cun enlace ao método que sobrescribe nos dous primeiros casos,
- xerando un Specified by na documentación cun enlace ao método que se implementa, no terceiro caso.
- en calquera dos tres casos, o método pode ter comentarios Javadoc escritos polo programador, que tamén aparecerán na documentación xerada.

Localización

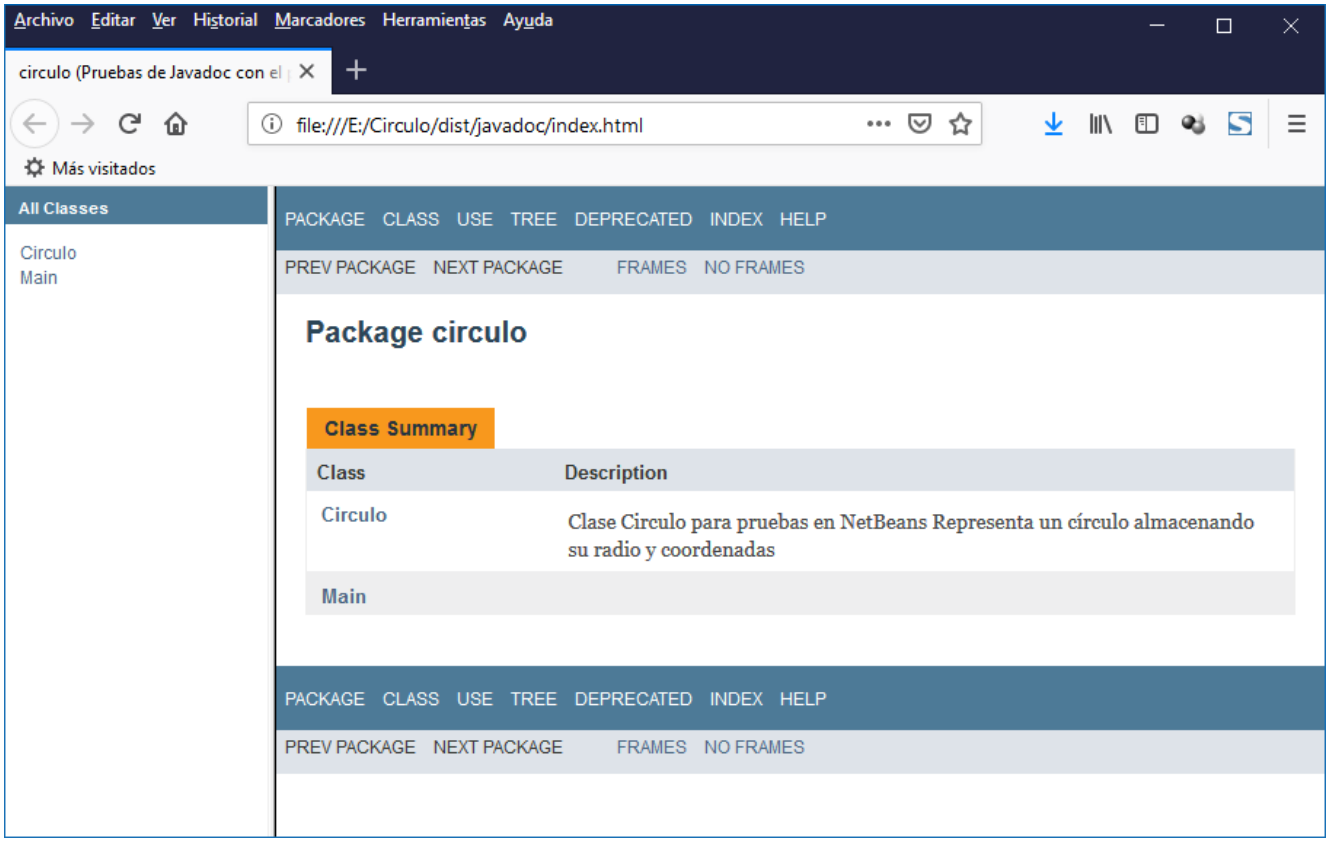
NetBeans xera as páxinas web necesarias, almacénaas dentro do proxecto na carpeta *dist/Javadoc* e lanza index.html no navegador designado.



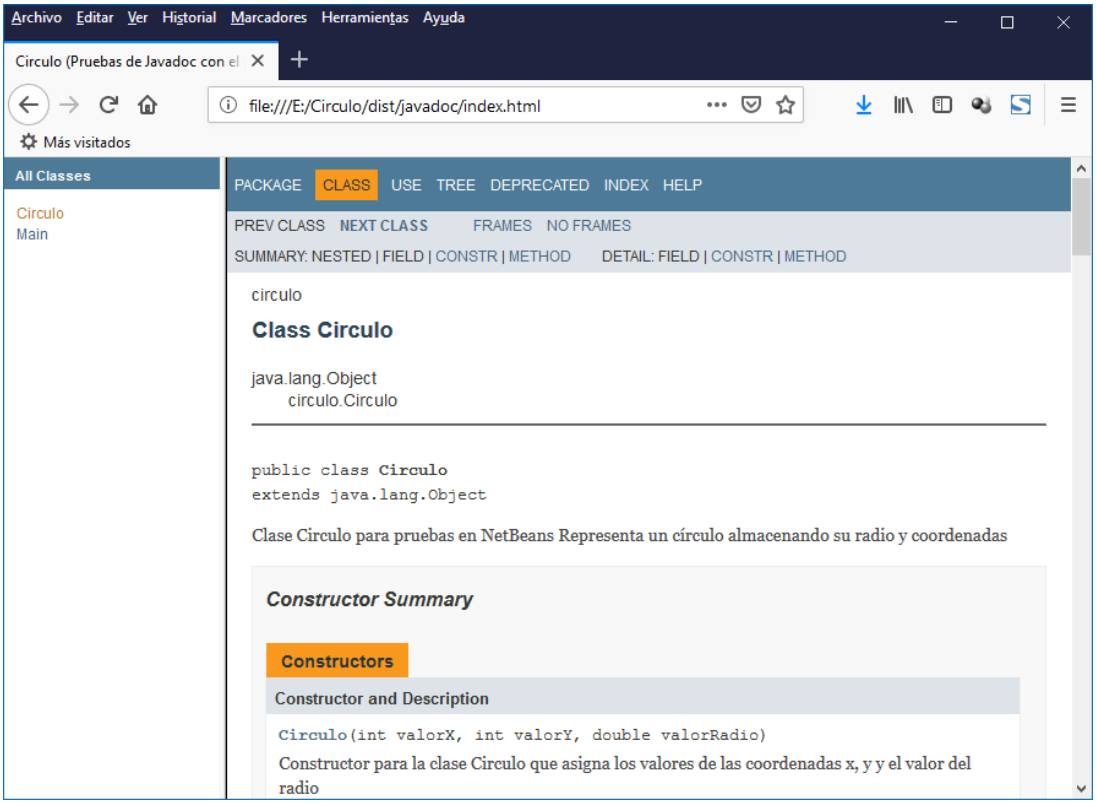
Forma

As páxinas web xeradas teñen a mesma forma cás da especificacións API de Java. Son páxinas HTML 4.01 Frameset que por defecto son vistas utilizando frames aínda que isto pode cambiarse. De utilizar frames, as páxinas estarán divididas en tres ou dous frames dependendo de se hai máis dun paquete. Os tres frames son: frame de paquetes, frame de clases (debaixo do de paquetes), frame de detalle (á dereita dos dous frames anteriores). Por defecto o frame de detalle aparece cunha barra de navegación na parte superior. O menú Help describe a forma das páxinas (en inglés).

No exemplo actual só hai un paquete, polo que as páxinas xeradas por Javadoc non terán o frame de paquetes. Na páxina principal descríbese o paquete único circulo. Os comentarios a carón de cada clase extráense dos comentarios Javadoc de cada clase. Observe que que a palabra Circulo aparece en bold ou grosa tal e como se colocou no comentario Javadoc correspondente e que non aparecen referencias ao autor nin a versión nin aparecen os campos de tipo private.



De seleccionar a clase Circulo no frame de clases, aparecería no frame de detalle información sobre esa clase en varios apartados: descrición xeral, resumo de métodos e campos e detalle de cada un deles.



Nos detalles do método construtor aparece:

Constructor Detail

Circulo

```
public Circulo(int valorX,
               int valorY,
               double valorRadio)
```

Constructor para la clase Circulo que asigna los valores de las coordenadas x, y y el valor del radio

Parameters:

`valorX` - Coordenada x entero

`valorY` - Coordenada y entero

`valorRadio` - Radio en decimal

que se extrae do código:

```
/**
 * Constructor para la clase Circulo que asigna los valores de las
 * coordenadas x, y y el valor del radio
 *
 * @param valorX Coordenada x entero
 * @param valorY Coordenada y entero
 * @param valorRadio Radio en decimal
 */
```

Na clase Main pode aparecer un enlace á documentación da clase Circulo no apartado See Also

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

circulo

Class Main

java.lang.Object
circulo.Main

```
public class Main
extends java.lang.Object
```

Clase Main para hacer pruebas en NetBeans con la clase Circulo

Version:

1.0

Author:

Profesor

See Also:

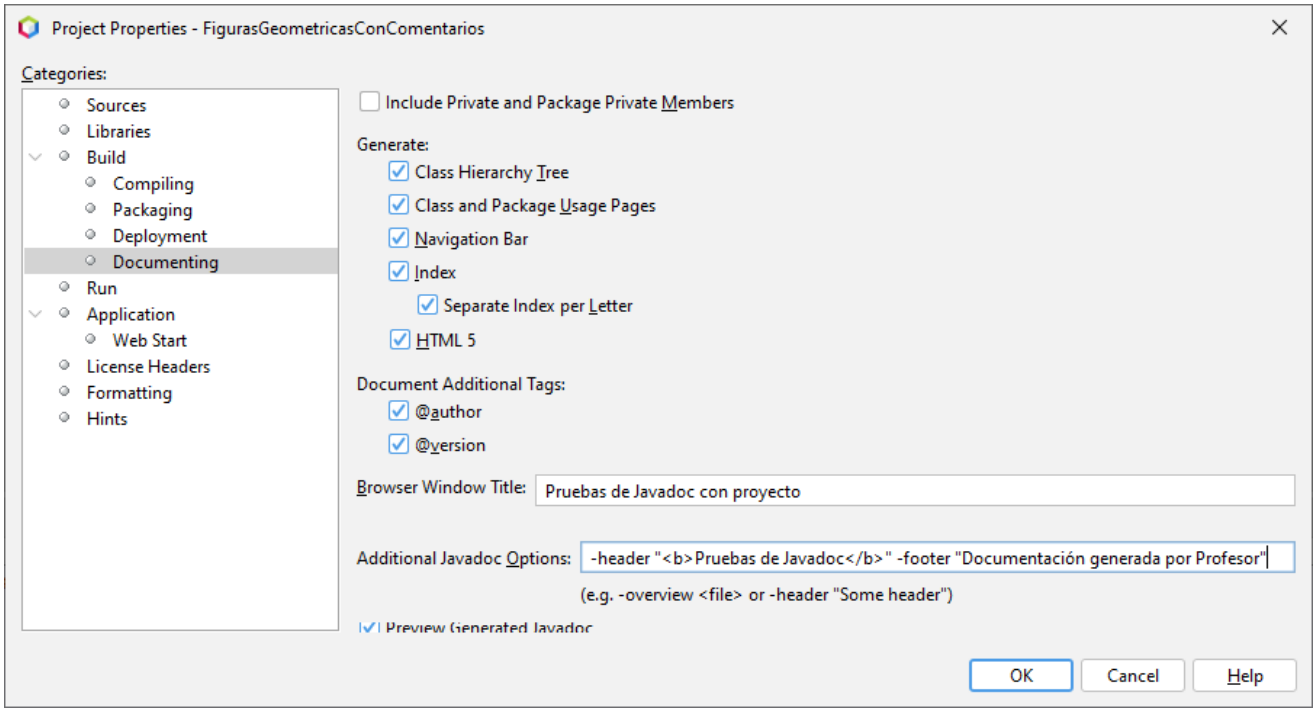
Circulo

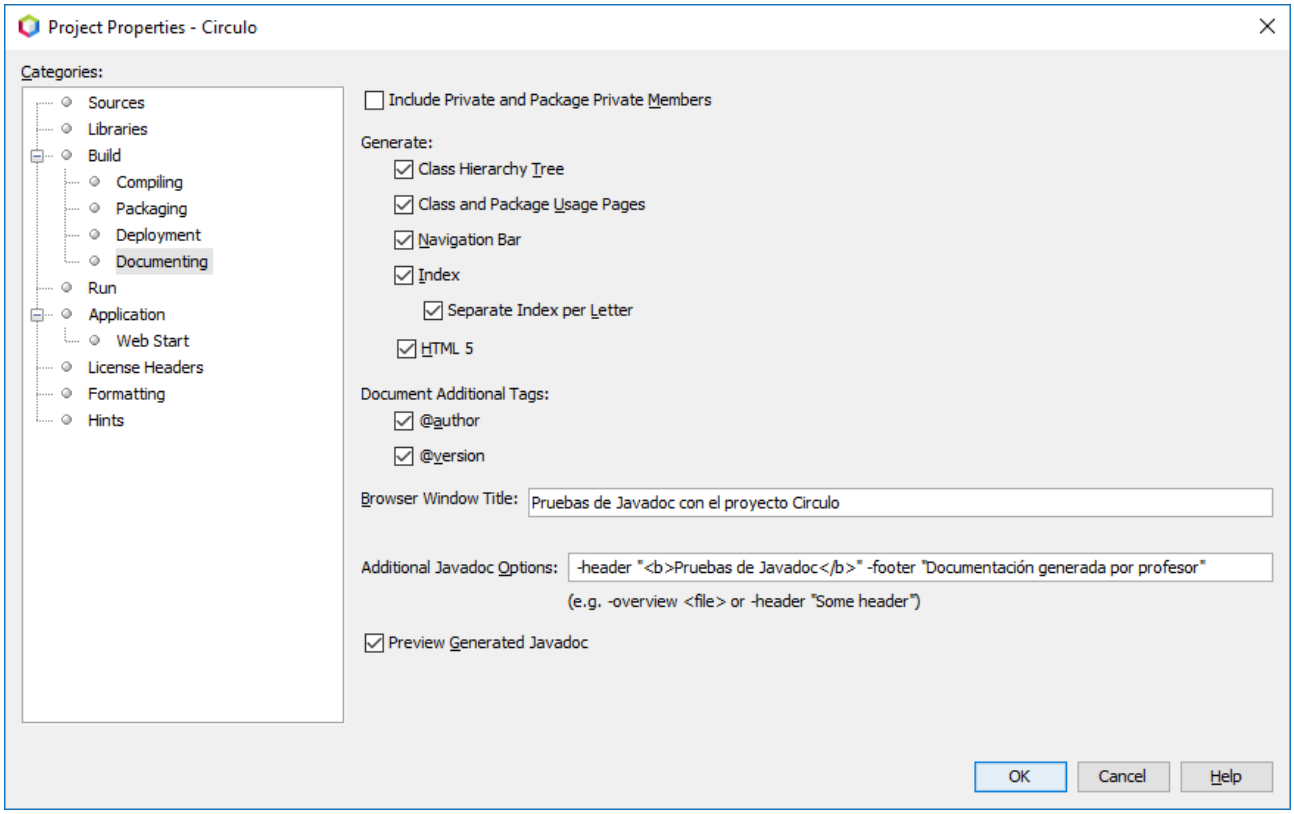
que se extrae do código

```
1   package circulo;
2
3   import java.text.DecimalFormat;
4
5   /**
6    * Clase Main para hacer pruebas en NetBeans con la clase Circulo
7    * @author profesor
8    * @version 1.0
9    * @see circulo.Circulo
10   */
11  public class Main {
12      public static void main(String[] args) {
```

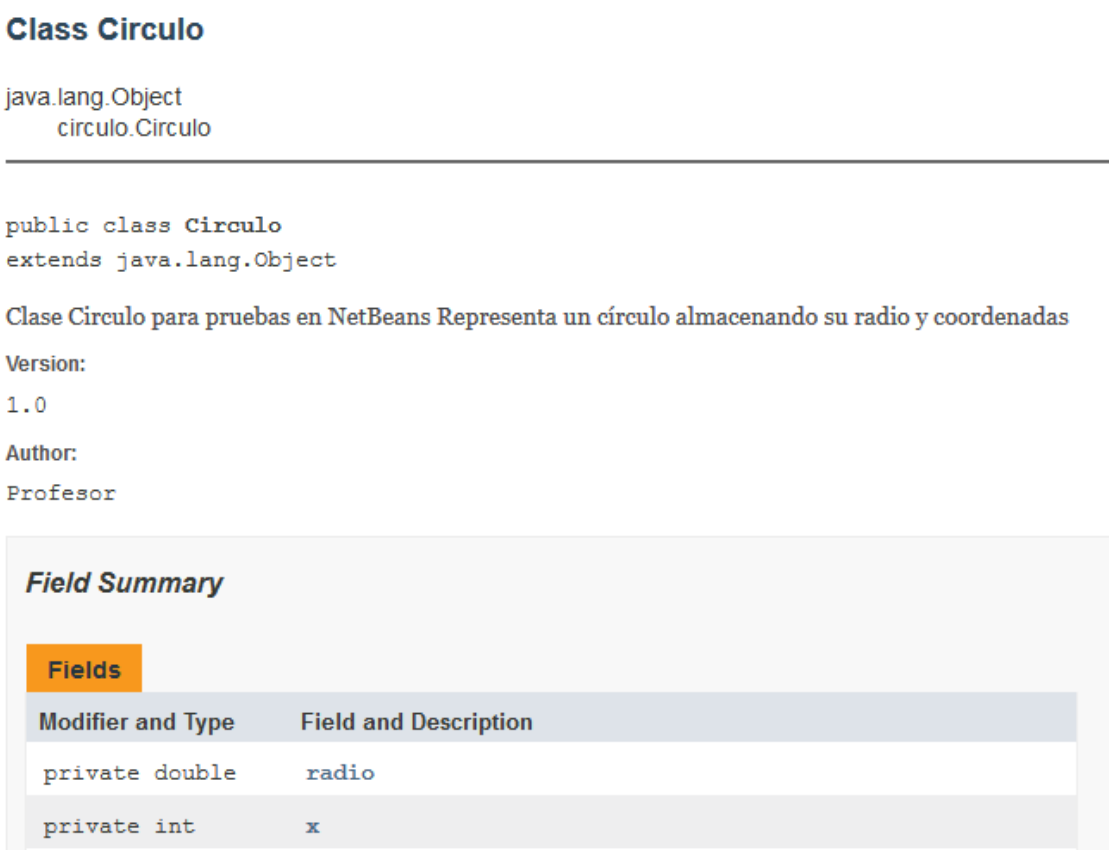
Configurar a xeración de documentación Javadoc para un proxecto

Para cambiar a documentación Javadoc xerada para un proxecto, débese seleccionar o proxecto na ventá de proxectos, facer clic dereito e elixir Propiedades. Aparece a caixa de diálogo das propiedades do proxecto na que se selecciona Categories--> Build--> Documenting e pódense modificar os valores por defecto.





Pódese marcar incluír membros privados y de paquete para que nos arquivos xerados aparezan os campos que son privados e pódense marcar os tags @author e @version, para que nos arquivos xerados apareza a documentación relativa a esas etiquetas Javadoc. Neste caso aparecerá incluída a seguinte documentación:

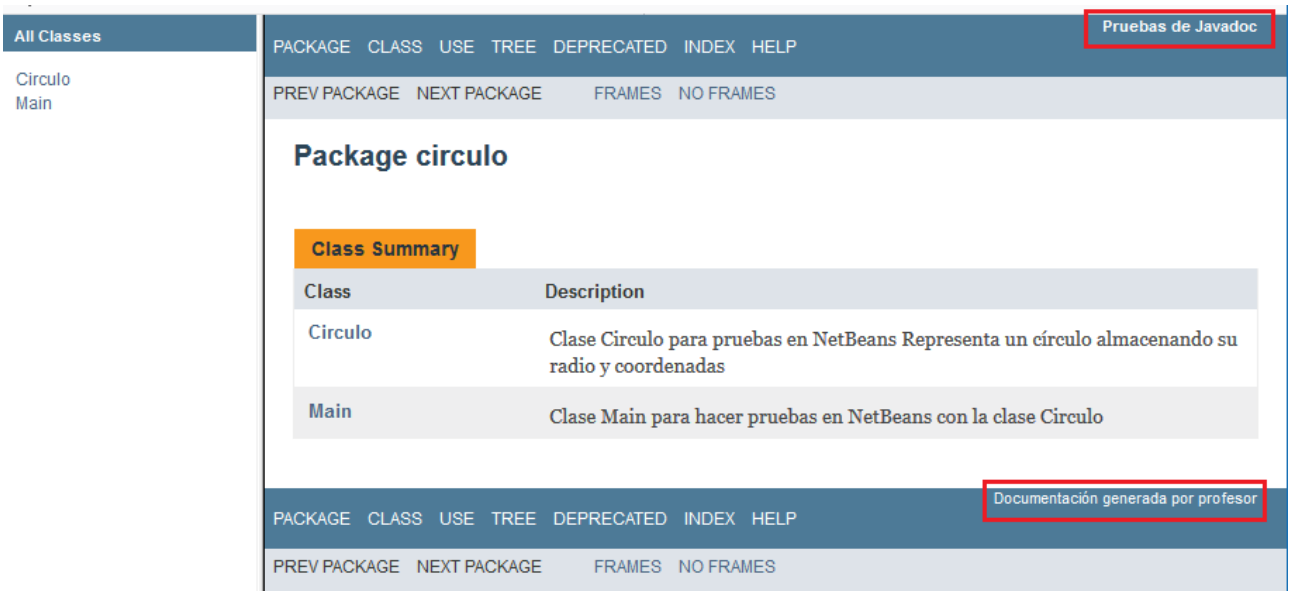


Pódese poñer un título para que apareza como contido da etiqueta <title> das páxinas HTML. En Opcións para Javadoc adicionais poderíanse poñer opcións de Javadoc. Ver opcións completas en

https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html#CHDFDACB

Por exemplo para que á dereita da barra de navegación apareza o texto Pruebas de Javadoc en letra grosa e apareza un texto no pé da páxina deberíase de escribir:

```
-header "<b>Pruebas de Javadoc</b>"
-footer "Documentación generada por profesor"
```

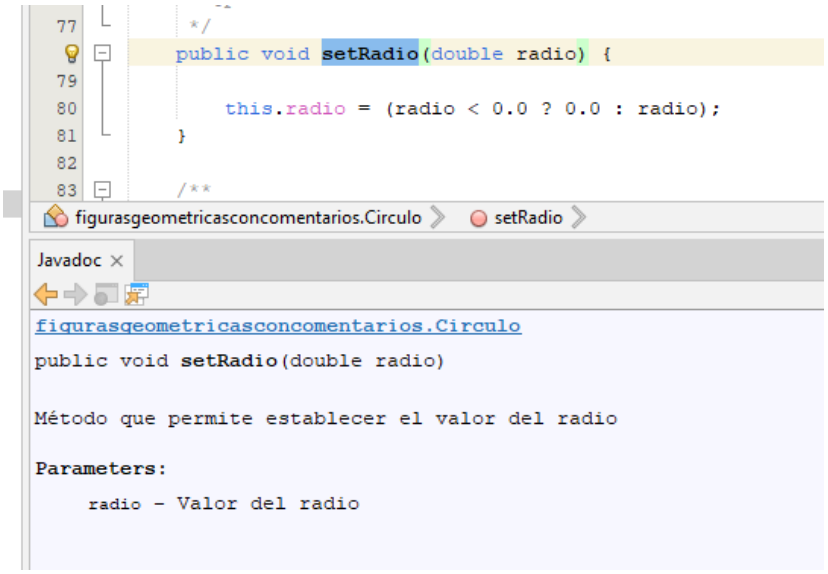


Ver documentación Javadoc dende o arquivo fonte

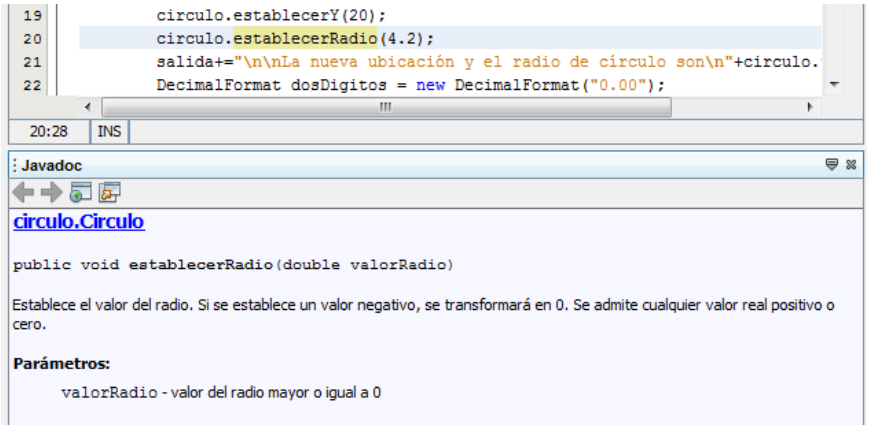
Para ver a documentación Javadoc dun elemento dende o código fonte, pódese facer de dúas maneiras.

- Vendo o resultado no navegador externo:

  Colocar o cursor sobre o elemento do código fonte do que se quere ver a documentación e premer Alt+F1 ou tamén facer clic dereito e elixir Mostra Javadoc. Se non funciona este forma é porque se necesita agregar Javadoc a NetBeans.



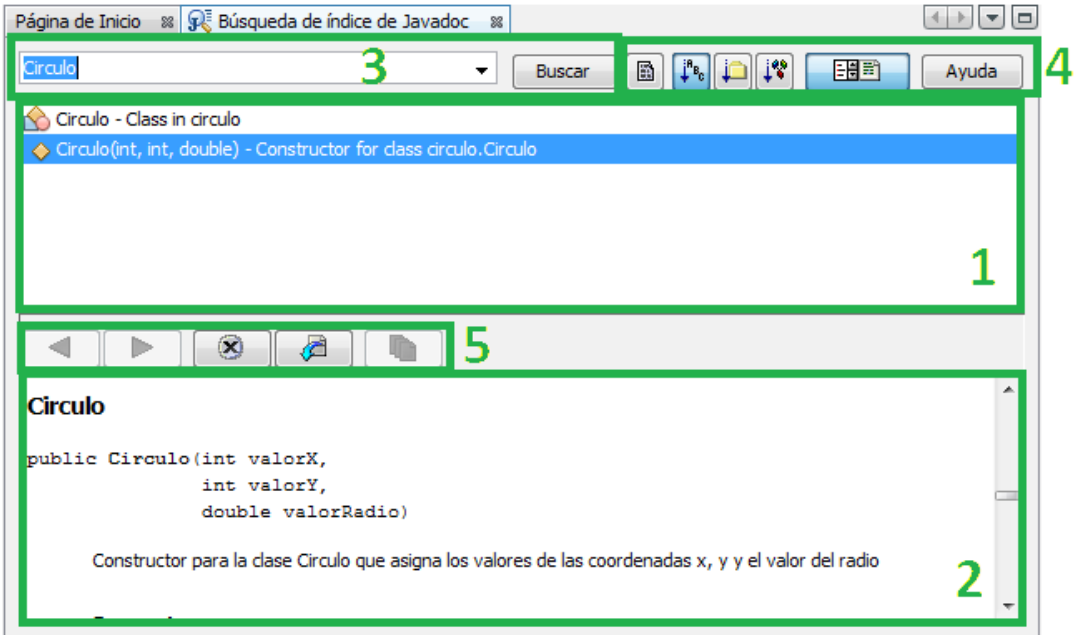
- Vendo o resultado na ventá de Javadoc (Windows→IDE Tools→Javadoc Documentation): Colocar o cursor sobre o elemento do código fonte do que se quere ver a documentación e aparecerá na ventá de Javadoc a documentación do elemento.



Ver documentación Javadoc dende o índice de busca de Javadoc

NetBeans permite facer buscas rápidas e fáciles na documentación Javadoc xerada sen saír de NetBeans. Para iso necesítase elixir Help→Javadoc index search de índice de Javadoc (Mayusculas-F1) no menú principal. Aparece a pestana Búsqueda de índice de Javadoc.

Se previamente o cursor estivese sobre un elemento do código fonte do que se quere ver a documentación, a pestana Búsqueda de índice de Javadoc aparecería cos resultados da busca dese elemento.



Na que se distinguen as seguintes zonas:

1. Zona de aparición dos resultados da busca. Pulsando dobre clic sobre un dos resultados, aparece a documentación no navegador.
2. Visor HTML. Anaco de páxina HTML de documentación relativa ó resultado resaltado na zona 1.

3. Permite teclear un texto ou seleccionar entre as buscas anteriores e premer en Buscar para facer unha nova busca.
4. Utilizar a barra de ferramentas para:
    - Mostrar orixe do resultado resaltado na zona 1 no código fonte. Ábrese a pestana có código fonte.
    - Ordenar os resultados alfabeticamente.
    - Ordenar os resultados alfabeticamente polo nome do paquete.
    - Ordenar os resultados alfabeticamente por entidade (clase, interface, método, construtor ou campo)
    - Activar ou desactivar o visor HTML.
    - Axuda.
5. Barra de ferramentas para moverse dentro do visor HTML:
    - Atrás. Retorna á páxina previa.
    - Vai á páxina visitada antes de premer en Atrás.
    - Para a carga da páxina.
    - Recargar a páxina actual.
    - Visualiza o historial das páxinas.

5.4 Etiquetas e anotacións

As etiquetas Javadoc non se deben de confundir coas anotacións Java tamén chamados metadatos Java. Semellanzas:

- colócanse xusto antes da clase, método ou campo á que afectan,
- empezan por @,
- poden ter os mesmos nomes aínda que nas anotación empezarán por maiúsculas e nas etiquetas en minúsculas,
- poden utilizarse para describir as mesmas situacións,
- no código fonte aparecerá o nome do elemento desaprobado tachado.

Diferencias:

- as etiquetas Javadoc van dentro de comentarios Javadoc e as anotación xusto antes do elemento seguidos dun espazo ou unha nova liña e fóra de comentarios,
- as etiquetas Javadoc serven para xerar documentación e as anotacións dan información aos analizadores e compiladores,
- as anotacións Java aportan información a Javadoc para xerar documentación,
- poden complementarse

Exemplo @deprecated

```java
/**
 * Obtiene el valor del diámetro
 * @return el valor del diámetro
 * @deprecated A partir de la vesión 1.1.
 * Se recomienda substituir por la operación radio*2
 */
public double obtenerDiametro() {
    return radio * 2;
}

/**
 * Obtiene el valor de la circunferencia
 * @return el valor de la circunferencia
 */
public double obtenerCircunferencia() {
    return Math.PI * obtenerDiametro();
}
```

No código fonte aparece tachado o método e as referencias ao método:

Na documentación HTML xerada, o método obtenerDiametro() sae como elemento obsoleto, en itálica e precedido por un warning en letra grosa.

obtenerDiametro

```
public double obtenerDiametro()
```

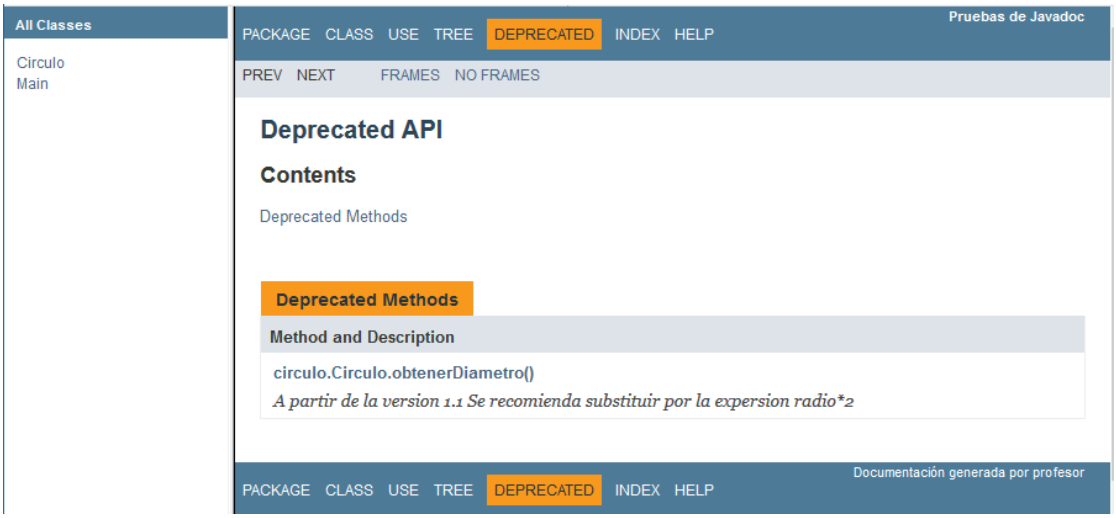
Deprecated. *A partir de la version 1.1 Se recomienda substituir por la expersion radio*2*

Calcula el diámetro del círculo

Returns:

Diámetro en decimal

Na documentación HTML dos elementos obsoletos aparece o método obtenerDiametro().



Exemplo @Deprecated

A anotación @Deprecated provoca que os compiladores (polo menos os de Sun) emitan unha mensaxe warning cando se utiliza o método indicando que o método está declarado como obsoleto e ademais xera na documentación un elemento desaprobado.

No código fonte:

```
/**
 * Calcula el diámetro del círculo
 * @return Diámetro en decimal

 */
@Deprecated
public double obtenerDiametro() {
    return radio * 2;
}

/**
 * Calcula el perímetro del círculo
 *
 * @return Perímetro en decimal
 */
public double obtenerCircunferencia() {
    return Math.PI * obtenerDiametro();
}
```

Na documentación da clase Circulo:

All Classes

Circulo
Main

obtenerDiametro

```
@Deprecated
public double obtenerDiametro()
```

Deprecated.
Calcula el diámetro del círculo

Returns:

Diámetro en decimal

obtenerCircunferencia

```
public double obtenerCircunferencia()
```

Calcula el perímetro del círculo

Returns:

Perímetro en decimal

Na documentación dos elementos obsoletos:

All Classes

Circulo
Main

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV NEXT FRAMES NO FRAMES

Deprecated API

Contents

Deprecated Methods

Deprecated Methods

Method and Description
circulo.Circulo.obtenerDiametro()

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

5.5 Exemplos

@see

Exemplo de colocación de enlace á documentación da clase Circulo na clase Main.java do proxecto Circulo utilizando @see circulo.Circulo

```
package circulo;

import java.text.DecimalFormat;

/**
 * Clase Main para hacer pruebas en NetBeans con la clase Circulo
 * @author profesor
 * @version 1.0
 * @see circulo.Circulo
 */
public class Main {
```

All Classes

Circulo
Main

Pruebas de Javadoc

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

circulo

Class Main

java.lang.Object
circulo.Main

```
public class Main
extends java.lang.Object
```

Clase Main para hacer pruebas en NetBeans con la clase Circulo

Version:

1.0

Author:

Profesor

See Also:

Circulo

{@link}

Exemplo de colocación de enlace á documentación da clase Circulo na clase Main.java do proxecto Circulo utilizando {@link circulo.Circulo circulo}:

```
package circulo;

import java.text.DecimalFormat;

/**
 * Clase Main para hacer pruebas en NetBeans con la
 * clase {@link circulo.Circulo circulo}
 * @author Profesor
 * @version 1.0
 * @see circulo.Circulo
 */
```

@docRoot

Exemplo de colocación dun logo na cabeceira da documentación:

- Na carpeta raíz do proxecto circulo, Circulo\dist\javadoc, crear a carpeta imaxes co logo lagarto.jpg.
- Nas propiedades do proxecto circulo, na categoría documentando, colocar nas opcións adicionais para Javadoc:

  -header "<img src="{@docRoot}/imaxes/lagarto.jpg" />"

{@value}

Exemplo de visualización do valor da constante PI de java.lang.Math:

```java
/**
 * Calcula el perímetro del círculo
 * y utiliza el valor de PI={@value java.lang.Math#PI}
 * @return Perímetro en decimal
 */
public double obtenerCircunferencia() {
    return Math.PI * obtenerDiametro();
}
```

obtenerCircunferencia

```
public double obtenerCircunferencia()
```

Calcula el perímetro del círculo y utiliza el valor de PI=3.141592653589793

Returns:

Perímetro en decimal

✍ Tarefa 4.11. Practicar coa documentación de código Java en NetBeans.

Documenta dúas aplicacións completas que teñas feita ata que "Analizar Javadoc" non dea ningún aviso e por tanto consideramos que está completamente documentado.

Usa as etiquetas @autor e @version nas clases

Mostra o resultado xerado por Javadoc en formato HTML