5 Herramientas para el control y documentación de software

5.1 Refactoring

Al desarrollar una aplicación es necesario tener muy presentes algunos aspectos del código de programación que se irá implementando. Hay pequeños aspectos que permitirán que este código sea considerado más óptimo o que facilitará su mantenimiento. Por ejemplo, uno de estos aspectos será la utilización de constantes. Si hay un valor que se utilizará varias veces a lo largo de un determinado programa, es mejor utilizar una constante que contenga ese valor, de esta manera, si el valor cambia sólo se tendrá que modificar la definición de la constante y no será necesario irlo buscando por todo el código desarrollado ni recordar cuántas veces se ha utilizado.

```
Class CalculCostos {
    Public static double CostTreballadors (double NreTreballadors)
    {
  Return 1200 * NreTreballadors;
    }
}
```

En el código anterior se muestra un ejemplo de cómo sería la codificación de una clase que tiene como función el cálculo de los costes laborales totales de una empresa. Se puede ver que el coste por trabajador no se encuentra en ninguna variable ni en ninguna constante, sino que el método CostTrabajadores devuelve el valor que ha recibido por parámetro (NreTrabajadores) por un número que considera el salario bruto por trabajador (en este caso supuesto 1200 euros). Probablemente, este importe se utilizará en más clases a lo largo del código de programación desarrollado o, al menos, más veces en la misma clase.

```
Class CalculCostos {
    final double SALARI_BRUT= 1200;
    Public static double CostTreballadors (double NreTreballadors)
    {
      Return SALARI_BRUT* NreTreballadors;
    }
}
```

El término ***refactoring*** hace referencia a los cambios efectuados en el código de programación desarrollado, sin implicar ningún cambio en los resultados de su ejecución. Es decir, se transforma el código fuente manteniendo intacto su comportamiento, aplicando los cambios sólo en la forma de programar o en la estructura del código fuente, buscando su optimización.

En la figura 5.1 puede verse un ejemplo de lo que se quiere expresar. ¿Cuál de las dos ¿casas facilita más la vida de sus inquilinos? Si tuviera que desarrollarse una aplicación informática, en cuál de las dos casas debería parecerse más, a la de la izquierda o en la de la derecha?

FIGURA 2.1. Disseny d'una casa

Posiblemente ambas casas cumplen las especificaciones iniciales, edificio en el que se pueda vivir, pero la primera casa parece ser más confortable y más óptima.

El término refacción proviene del término refactorizar (refactoring). Este término se convierte de su similitud con el concepto de factorización de los números o de los polinomios. Es lo mismo tener 12 que tener 3×4, pero, en el segundo caso, el término 12 está dividido en sus factores (aún podría factorizarse más y llegar al 3×2×2). Uno otro ejemplo: el polinomio de segundo grado $x^2 - 1$ es el mismo que el resultado del producto (x + 1)(x - 1), pero en el segundo caso se ha dividido en factores. A la hora de simplificar o realizar operaciones, será mucho más sencillo el trabajo con el segundo caso (con los términos ya factorizados) que con el primero. Con la factorización aparecen unos términos, unos valores, que inicialmente se encuentran ocultos, aunque forman parte del concepto inicial.

En el caso de la programación ocurre una situación muy similar. Si bien el código que se desarrolla no está factorizado, es decir, no se ven a simple vista los factores internos, porque son estructuras que aparentemente se encuentran escondidas, cuando se lleva a término una refacción del código fuente sí que se pueden ver.

5.1.1 Ventajas y limitaciones de la refacción

La utilización de la refacción puede aportar algunas ventajas a los desarrolladores de software, pero hay que tener en cuenta que tiene limitaciones que es necesario conocer antes de tomar la decisión de utilizarla.

Ventajas de la refacción

Hay muchas **ventajas** en la utilización de la refacción, aunque también existen inconvenientes y algunas limitaciones. Por qué los programadores dedican tiempo ¿a la refacción del código fuente? Una de las respuestas a esta pregunta es el aumento de la calidad del código fuente. Un código fuente sobre el que se han utilizado técnicas de refacción se mantendrá en un estado mejor que un código fuente sobre el que no se hayan aplicado. A medida que un código fuente original se ha ido modificando, ampliando o manteniendo, habrá sufrido modificaciones en la estructura básica sobre la que se va diseñar, y es cada vez más difícil efectuar evolutivos y aumenta la probabilidad generar errores.

Algunas de las ventajas o razones para utilizar la técnica de refacción del código fuente son:

- Prevención de la aparición de problemas habituales a partir de los cambios provocados por los mantenimientos de las aplicaciones.
- Ayuda a aumentar la simplicidad del diseño.
- Mayor entendimiento de las estructuras de programación.
- Detección más sencilla de errores.
- Permite agilizar la programación.
- Genera satisfacción en los desarrolladores.

A continuación, se desarrollan algunos de estos **puntos fuertes** de la utilización de la refacción:

• **Detección y prevención más sencilla de errores**. La refacción mejora la robustez del código fuente desarrollado, haciendo que sea más sencillo encontrar errores en el código o encontrar partes del código que sean más propensas a tener o provocar errores en el conjunto del software. A partir de la utilización de casos de prueba adecuados, se podrá mejorar mucho el código fuente.

• **Prevención de problemas por culpa de los mantenimientos del software**. Con el tiempo, suelen surgir problemas a medida que se va aplicando un mantenimiento evolutivo o un mantenimiento correctivo de las aplicaciones informáticas. Algunos ejemplos de estos problemas pueden ser que el código fuente se vuelva más complejo de entender de lo que sería necesario o que haya duplicidad de código, debido a que, muchas veces, son personas distintas las que han desarrollado el código de las que están llevando a cabo su mantenimiento.

• **Comprensión del código fuente y simplicidad del diseño**. Volviendo a la situación en qué un equipo de programación puede estar compuesto por un número determinado de personas diferentes o que el departamento de mantenimiento de una empresa es diferente al equipo de desarrollo de nuevos proyectos, es muy importante que el código fuente sea muy fácil de entender y que el diseño de la solución haya sido creado con una simplicidad considerable. Es necesario que el diseño se lleve a cabo teniendo en cuenta que se hará una posterior refacción, es decir, teniendo presentes posibles necesidades futuras de la aplicación que se está creando. Esta tarea no es nada sencilla, pero con un buen y exhaustivo análisis, por medio de muchas conversaciones con los usuarios finales, se podrán llegar a vislumbrar estas necesidades futuras.

• **Programación más rápida**. Precisamente, si el código se comprende de una forma rápida y sencilla, la evolución de la programación será mucho más rápida y eficaz. El diseño llevado a cabo en la fase anterior también será decisivo en que la programación sea más ágil.

Limitaciones de la refacción

En cambio, se pueden encontrar varias razones por no considerar adecuada su utilización, ya sea por sus **limitaciones** o por las posibles problemáticas que pueden surgir:

- Personal poco preparado para utilizar las técnicas de refacción.
- Exceso de obsesión por conseguir el mejor código fuente.
- Excesiva dedicación de tiempo a la refacción, provocando efectos negativos.
- Repercusiones en el resto del software y del equipo de desarrolladores
- cuando uno aplica técnicas de refacción.
- Posibles problemas de comunicación provocados por el punto anterior.
- Limitaciones debidas a las bases de datos, interfaces gráficas...

Algunos de estos **puntos débiles** relacionados con la utilización de la refacción quedan desarrollados a continuación:

• **Dedicación de tiempo**. Una actitud obsesiva con la refacción podrá llevar a un efecto contrario al que se busca: dedicar mucho más tiempo de lo que debería la creación de código y aumentar la complejidad del diseño y de la programación innecesariamente.

• **Afectar o generar problemas en el resto del equipo de programación**. Una refacción de un programador puede generar problemas a otros componentes del equipo de trabajo, en función de dónde se haya llevado a cabo esta refacción. Si sólo afecta a una clase o a algunos de sus métodos, la refacción será imperceptible al resto de sus compañeros. Pero cuando afecta a varias clases, podrá alterar otro código fuente que haya sido desarrollado o se está desarrollando por parte de otros compañeros. Este problema sólo se puede solucionar con una buena comunicación entre los componentes del equipo de trabajo o con una refacción sincronizada desde los responsables del proyecto, manteniendo informados a los afectados.

• **Limitaciones debidas a las bases de datos**. La refacción tiene algunas limitaciones o áreas conflictivas, como las bases de datos o las interfaces gráficas. En el caso de las bases de datos, es un problema el hecho de que programas que se desarrollan actualmente estén tan ligados a sus estructuras. En el caso de existir modificaciones relacionadas con la refacción en el diseño de la base de datos vinculada a una aplicación, debería llevarse a término muchas acciones que complicarían esta actuación: habría que ir a la base de datos, efectuar los cambios estructurales adecuados, hacer una migración de los datos hacia el nuevo sistema y adaptar de nuevo todo lo de la aplicación relacionado con los datos (formularios, informes...).

• **Interfaces gráficas**. Una segunda limitación se encuentra con las interfaces gráficas de usuario. Las nuevas técnicas de programación han facilitado la independencia entre los distintos módulos que componen las aplicaciones. De este modo, se podrá modificar el contenido del código fuente sin tener que hacer modificaciones en el resto de módulos, como por ejemplo, en las interfaces. El problema con la refacción vinculado con las interfaces gráficas radica en que si esta interfaz ha sido publicada en muchos usuarios clientes o si no se tiene accesibilidad al código que la genera, será prácticamente imposible actuar sobre ella.

La **refacción** se considera un aspecto muy importante para el desarrollo de aplicaciones mediante programación extrema.

5.1.2 Patrones de refacción más usuales

Los patrones, en un contexto de programación, ofrecen una solución durante el proceso de desarrollo de software a un tipo de problema o de necesidad estándar que puede darse en distintos contextos. El patrón dará una resolución a este problema, que ya ha sido aceptada como buena solución, y que ya ha sido bautizada con un nombre.

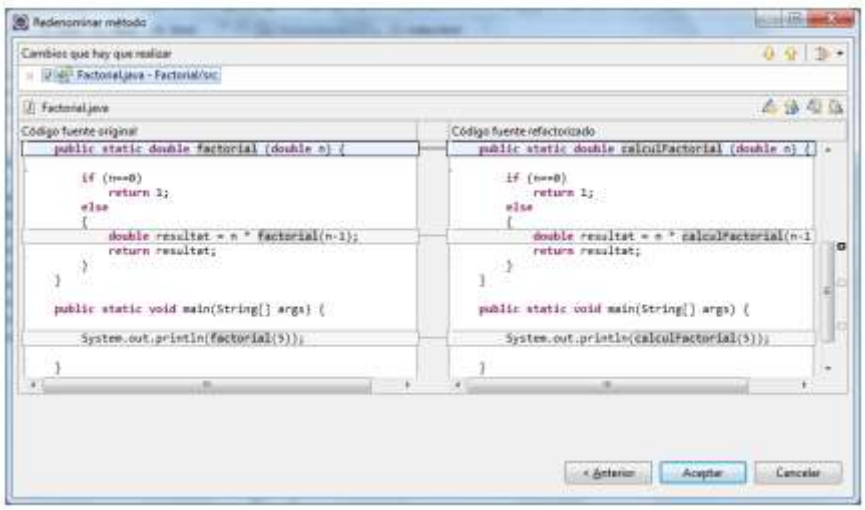
Por otra parte, ya se ha definido la refacción como adaptaciones del código fuente sin que esto provoque cambios en las operaciones del software. Si se unen estos dos conceptos pueden encontrarse algunos patrones existentes.

Los patrones más habituales son los siguientes:

- Renombrar
- Mover
- Extraer una variable local
- Extraer una constante
- Convertir una variable local en un campo
- Extraer una interfaz
- Extraer el método

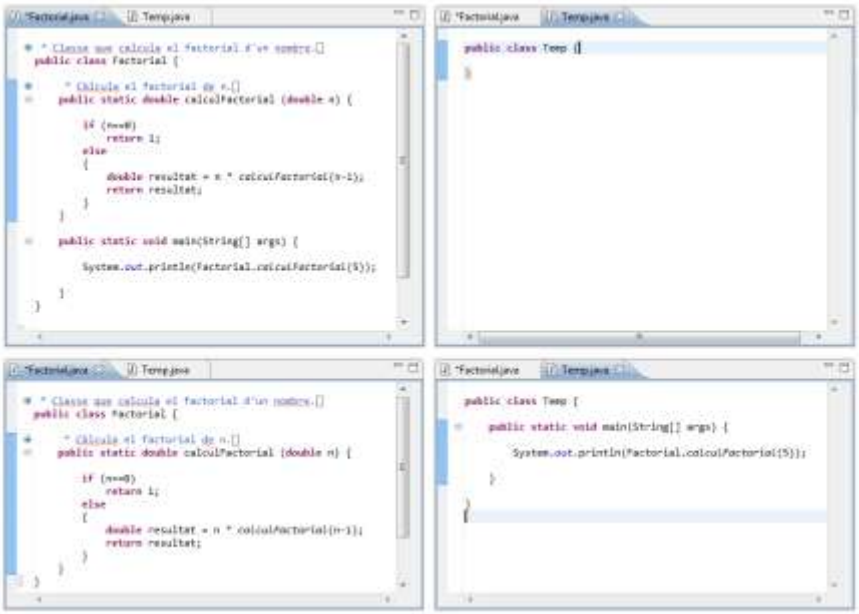
Renombrar

Este patrón cambia el nombre de variables, clases, métodos, paquetes... teniendo en cuenta sus referencias.



Mover

Este patrón mueve un método de una clase a otra; mueve una clase de un paquete a otro... teniendo en cuenta sus referencias.



Extraer variable local

Dada una expresión, ese patrón le asigna una variable local; cualquiera referencia a la expresión en el ámbito local será sustituida por la variable.

```
1  public static void main(String[] args) {
2      int nre = 3;
3      System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
4      nre = 5;
5      System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
           );
6    }
```

El codi refactoritzat és el que es mostra a continuació:

```
1  public static void main(String[] args) {
2      int nre = 3;
3      String text = "El factorial de ";
4      System.out.println(text + nre + " és " + calculFactorial(nre));
5      nre = 5;
6      System.out.println(text + nre + " és " + calculFactorial(nre));
7    }
```

Extraer una constante

Dada una cadena de caracteres o un valor numérico, este patrón le convierte en una constante, y cualquier referencia será sustituida por la constante.

```
public static void main(String[] args) {
    int nre = 3;
    System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
        );
    nre = 5;
    System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
        );
  }
```

El codi refactoritzat és el que es mostra a continuació:

```
private static final String TEXT = "El factorial de ";

  public static void main(String[] args) {
    int nre = 3;
    System.out.println(TEXT + nre + " és " + calculFactorial(nre));
    nre = 5;
    System.out.println(TEXT + nre + " és " + calculFactorial(nre));
  }
```

Convertir una variable local en un campo

Dada una variable local, ese patrón la convierte en atributo de la clase; cualquier referencia será sustituida por el nuevo atributo.

```
public class Factorial {
  public static double calculFactorial (double n) {
    if (n==0)
      return 1;
    else
    {
      double resultat = n * calculFactorial(n-1);
      return resultat;
    }
  }
  public static void main(String[] args) {
    int nre = 3;
    System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
        );
    nre = 5;
    System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
        );
  }
}
```

El codi refactoritzat és el que es mostra a continuació:

```
public class Factorial {
  private static int nre;
  public static double calculFactorial (double n) {
    if (n==0)
      return 1;
    else
    {
      double resultat = n * calculFactorial(n-1);
      return resultat;
    }
  }
  public static void main(String[] args) {
    nre = 3;
    System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
        );
    nre = 5;
    System.out.println("El factorial de " + nre + " és " + calculFactorial(nre)
        );
  }
}
```

Extraer interfaz

Este patrón crea una interfaz con los métodos de la clase.

```
public class Factorial {
  public double calculFactorial (double n) {
    if (n==0)
      return 1;
    else
    {
      double resultat = n * calculFactorial(n-1);
      return resultat;
    }
  }
}
```

El codi refactoritzat és el que es mostra tot seguit:

```
public interface InterficieFactorial {
  public abstract double calculFactorial(double n);
}
```

```
public class Factorial implements InterficieFactorial {
  /* (non-Javadoc)
   * @see InterficieFactorial#calculFactorial(double)
   */
  @Override
  public double calculFactorial (double n) {
    if (n==0)
      return 1;
    else
    {
      double resultat = n * calculFactorial(n-1);
      return resultat;
    }
  }
}
```

Extraer método

Este patrón convierte un trozo de código en un método.

```
public static void main(String[] args) {
      int nre = 3;
      int comptador = 1;
      double resultat = 1;
      while (comptador<=nre){
        resultat = resultat * comptador;
        comptador++;
      }
      System.out.println("Factorial de " + nre + ": " + calculFactorial(nre));
  }
```

El codi refactoritzat és el que es mostra tot seguit:

```
private static void calcularFactorial(int nre) {
      int comptador = 1;
      double resultat = 1;
      while (comptador<=nre){
        resultat = resultat * comptador;
        comptador++;
      }
  }
  public static void main(String[] args) {
      int nre = 5;
      System.out.println("Factorial de " + nre + ": " + calculFactorial(nre));
  }
```

5.2 Control de versiones

Un **sistema de control de versiones** es una herramienta de ayuda al desarrollo de software que irá almacenando, según los parámetros indicados, la situación del código fuente en momentos determinados. Es como una herramienta que va haciendo fotos de forma regular, cada cierto tiempo, sobre el estado del código.

En un entorno donde sólo trabajará un programador, bastará con guardar la información del código cada cierto tiempo o cada vez que él guarde la información, junto con los datos principales, como el número de la versión o la fecha y la hora en el que se ha almacenado. En un entorno multiusuario, donde muchos programadores diferentes pueden manipular los archivos y donde, incluso, se podrá ofrecer la posibilidad de modificar a la vez un mismo documento, en estos casos es necesario almacenar mucha más información, como qué usuario ha implementado los cambios, las referencias de la máquina desde donde se han realizado los cambios, si se está produciendo alguno tipos de conflicto.

Esta funcionalidad no sólo será importante en los casos de desarrollo software, sino también en muchos otros ámbitos. Por poner unos ejemplos, se puede encontrar la importancia del control de versiones en el caso de trabajar en la creación de un libro, colaborando varios autores en su redacción. Si queda grabado en qué momento ha realizado cada cambio cualquier colaborador diferente, y, además, queda registrada una copia de cada modificación hecha, esto permitirá al resto de colaboradores tener una información muy importante para no repetir contenidos ni duplicar trabajos. Un ejemplo real para lo que se acaba de explicar se puede encontrar en la redacción de contenidos de la wikipedia, donde muchos redactores crean contenidos mediante una herramienta que permite el trabajo en equipo y el control de los contenidos creados y colgados. Otro caso se puede encontrar en una gestoría o un bufete de abogados que deben redactar un contrato o un documento en colaboración con uno o varios clientes. En este caso, no será necesario utilizar una herramienta como una wiki, quizás será suficiente utilizar un buen editor de textos que permita la funcionalidad de gestión de cambios, donde cada vez que un usuario distinto tenga que hacer un cambio, quede indicado en el documento con un color diferente en función del usuario que lo haya creado.

En el desarrollo de software, los sistemas de control de versiones son herramientas que pueden facilitar mucho el trabajo de los programadores y aumentar la productividad de forma considerable, siempre que estas herramientas sean utilizadas de forma correcta. Algunas funcionalidades de los sistemas de control de versiones pueden ser:

• **Comparar cambios en los distintos archivos a lo largo del tiempo**, pudiendo ver quien ha modificado por última vez un determinado archivo o pedazo de código fuente.

• Reducción **de problemas de coordinación que puede haber entre los distintos programadores**. Con los sistemas de control de versiones podrán compartir su trabajo, ofreciendo las últimas versiones del código o de los documentos, y trabajar, incluso, de forma simultánea sin miedo a encontrarse con conflictos en el resultado de esta colaboración.

• **Posibilidad de acceder a versiones anteriores de los documentos o código fuente**. De forma programada se podrá automatizar la generación de copias de seguridad o, incluso, almacenar todo cambio efectuado. En el caso de haberse equivocado de forma puntual, o durante un período largo de tiempo, el programador podrá tener acceso a versiones anteriores del código o deshacer, paso a paso, todo lo desarrollado durante los últimos días.

• **Ver qué programador ha sido el último en modificar un determinado pedazo de código** que puede estar causando un problema.

• **Acceso al historial de cambios sobre todos los archivos a medida que avanza el proyecto**. También puede servir para el jefe de proyectos o para cualquier otra parte interesada (stakeholder), con permisos para acceder a este historial, para ver la evolución del proyecto.

• **Devolver un archivo determinado o todo el proyecto entero a uno o varios estados anteriores**.

Los sistemas de control de versiones ofrecen, además, algunas funcionalidades para poder gestionar un proyecto informático y para poder realizar su seguimiento. Entre estas funcionalidades se pueden encontrar:

• **Control histórico detallado de cada archivo**. Permite almacenar toda la información de lo que ha sucedido en un archivo, tales como todos los cambios que se han efectuado, por quien, el motivo de los cambios, almacenar todas las versiones que ha habido desde su creación...

• **Control de usuarios con permisos para acceder a los archivos**. Cada usuario tendrá un tipo de acceso determinado a los archivos para poder consultarlos o modificarlos o, incluso, borrarlos o crear otros nuevos. Este control debe de ser gestionado por la herramienta de control de versiones, almacenando todos los usuarios posibles y los permisos que tienen asignados.

• **Creación de ramas de un mismo proyecto**: en el desarrollo de un proyecto hay momentos en los que es necesario ramificarlo, es decir, a partir de un determinado momento, de un determinado punto, es necesario crear dos ramas del proyecto que se podrán seguir desarrollando por separado. Este caso se puede dar en el momento de finalizar una primera versión de una aplicación que se entrega a los clientes, pero que es necesario seguir evolucionando.

• **Fusionar dos versiones de un mismo archivo**: permitiendo fusionarlas, cogiendo, de cada parte del archivo, el código que más interese a los desarrolladores. Esta funcionalidad deberá validarse manualmente por parte de una persona.

5.2.1 Componentes de un sistema de control de versiones

Un sistema de control de versiones se compondrá de varios elementos o componentes que utilizan una terminología algo específica. Hay que tener en cuenta que no todos los sistemas de control de versiones utilizan los mismos términos para referirse a los mismos conceptos. A continuación, encontrará una lista con los términos más comunes:

• Repositorio (**repository** o depot): conjunto de datos almacenados, también referido a versiones o copias de seguridad. Es el sitio donde estos datos quedan almacenadas. Se podrá referir a muchas versiones de un único proyecto o de varios proyectos.

• Módulo (module): se refiere a una carpeta o directorio específico del repositorio. Un módulo podrá hacer referencia a todo el proyecto entero o sólo a una parte del proyecto, es decir, a un conjunto de archivos.

• Tronco (trunk o **master**): estado principal del proyecto. Es el estado del proyecto destinado, al terminar su desarrollo, a ser pasado a producción.

• Rama (**branch**): es una bifurcación del tronco o rama maestra de la aplicación que contiene una versión independiente de la aplicación ya la que pueden aplicarse cambios sin que afecten ni al tronco ni a otras ramas. Estos cambios, en un futuro, pueden incorporarse al tronco.

• Versión o Revisión (**version** o revision): es el estado del proyecto o de una de las sus ramas en un momento determinado. Se crea una versión cada vez que se añaden cambios a un repositorio.

• Etiqueta (**tag**, label o baseline): información que se añadirá a una versión. A menudo indica alguna característica específica que le hace especial. Ésta información será textual y, en muchas ocasiones, se generará de forma manual. Por ejemplo, se puede etiquetar la primera versión de un software (1.0) o una versión en la que se ha solucionado un error importante.

• (head o tip): hace referencia a la versión más reciente de una determinada rama o del tronco. El tronco y cada rama tienen su propia cabeza, sin embargo, para referirse a la cabeza del tronco, a veces se utiliza el término HEAD, en mayúsculas.

• Clonar (**clone**): consiste en crear un nuevo repositorio, que es una copia idéntica de otro, puesto que contiene las mismas revisiones.

• Bifurcación (**fork**): creación de un nuevo repositorio a partir de otro. Éste nuevo repositorio, al contrario que en el caso de la clonación, no está ligado al repositorio original y se trata como un repositorio distinto.

• **Pull**: es la acción que copia los cambios de un repositorio (habitualmente remoto) en el repositorio local. Esta acción puede provocar conflictos.

• **Push** o fetch: son acciones utilizadas para añadir los cambios del repositorio local en otro repositorio (habitualmente remoto). Esta acción puede provocar conflictos.

• Cambio (change o diff): representa una modificación concreta de un documento bajo el control de versiones.

• Sincronización (update o sync): es la acción de combinar los cambios realizados en el repositorio con la copia de trabajo local.

• Conflicto (conflict): se produce cuando se intentan añadir cambios a un archivo que ha sido modificado previamente por otro usuario. Antes de poder combinar los cambios con el repositorio deberá resolverse el conflicto.

• Fusionar (**merge** o integration): es la acción que se produce cuando se quieren combinar los cambios de un repositorio local con un remoto y detectar cambios en el mismo archivo en ambos repositorios y se produce un conflicto. Para resolver este conflicto deben fusionarse los cambios antes de poder actualizar los cambios repositorios. Esta fusión puede consistir en descartar los cambios de uno de los dos repositorios o editar el código para incluir los cambios del archivo en ambas bandas. Cabe destacar que es posible que un mismo archivo presente cambios en muchos puntos diferentes que se tendrán que resolver para poder dar la fusión por finalizada.

• Bloqueo (lock): algunos sistemas de control de versiones en lugar de utilizar el sistema de fusiones lo que hacen es bloquear los archivos en uso, por lo que sólo puede haber un solo usuario modificando un archivo en un momento dado.

• Directorio de trabajo (working directory): directorio en el que el programador trabajará a partir de una copia que habrá hecho del repositorio en su ordenador local.

• Copia de trabajo (working copy): hace referencia a la copia local de los archivos que se han copiado del repositorio, que es sobre la que se hacen los cambios (es en decir, se trabaja, de ahí el nombre) antes de añadir estos cambios al repositorio. Se almacena en el directorio de trabajo.

• Volver a la versión anterior (revert): descarta todos los cambios producidos en la copia de trabajo desde la última subida al repositorio local.

5.2.2 Clasificación de los sistemas de control de versiones

Se puede generalizar la estructura de las herramientas de control de versiones teniendo en cuenta la clasificación de los sistemas de control de versiones, que pueden ser locales, centralizados o distribuidos.

Sistemas locales

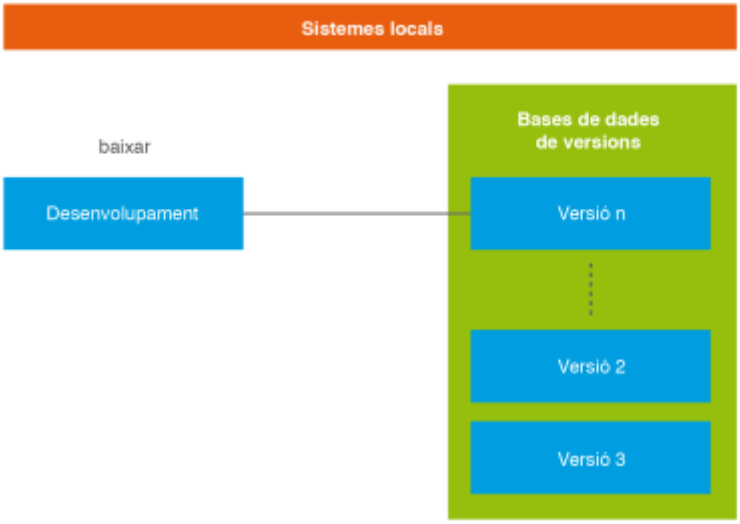
Los sistemas de control de versiones locales son sistemas que permiten llevar a cabo las acciones necesarias de forma local. Si no se utiliza ningún sistema de control de versiones concreto, un programador que trabaje en su propio ordenador podrá ir haciendo copias de seguridad, de vez en cuando, de los archivos o carpetas con las que trabaje. Este sistema implica dos características específicas:

• El mismo programador será la persona que tendrá que acordarse de ir haciendo las copias de seguridad cada cierto tiempo, lo que él haya establecido.

• El propio programador deberá decidir dónde llevará a cabo estas copias de seguridad, muy probablemente en local, en otra ubicación del disco duro interno, o en un dispositivo de almacenamiento externo.

Este sistema tiene un alto riesgo porque es un sistema altamente dependiente del programador, pero es un sistema extremadamente sencillo de planificar y ejecutar. Lo habitual, en estos casos, es crear copias de seguridad (que se pueden considerar versiones del proyecto); los archivos se almacenan en carpetas que suelen tener un nombre representativo, como por ejemplo: la fecha y la hora en la que se efectúa la copia. Pero, tal y como se ha comentado anteriormente, se trata de un sistema propenso a errores por diversos motivos, tales como que se produzcan olvidos en la realización de la copia de seguridad, o confusiones que lleven al programador a continuar el desarrollo en una de las copias del código, e ir avanzando con el proyecto en diferentes ubicaciones distintos días.

Para hacer frente a estos problemas, aparecieron los primeros repositorios de versiones que contenían una pequeña base de datos donde se podían grabar todos los cambios efectuados, sobre qué archivos, quién los había hecho, cuándo... Efectuando las copias de seguridad de forma automatizada.

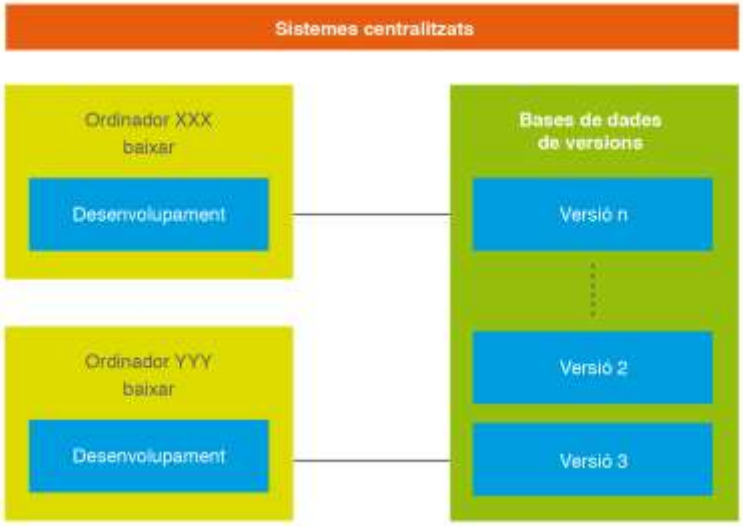


Sistemas centralizados

Un sistema de control de versiones que permita desarrollar un proyecto informático con más de un ordenador es el sistema de control de versiones centralizado que se contrapone a otro sistema de control de versiones, también para más de uno ordenador, cómo es el distribuido.

En los sistemas centralizados habrá más de un programador desarrollando un proyecto en más de un ordenador. Desde ambos ordenadores se podrá acceder a los mismos archivos de trabajo y sobre las mismas versiones almacenadas. Ésta es una situación más cercana a las situaciones actuales reales en el desarrollo de proyectos informáticos.

El sistema de control de versiones centralizado es un sistema donde las copias de seguridad o versiones almacenadas se encuentran de forma centralizada a uno servidor que será accesible desde cualquier ordenador que trabaje en el proyecto.



Pero este sistema tendrá también sus inconvenientes, a veces muy difíciles que salvar. ¿Qué ocurre si falla el servidor central? Si es un fallo temporal no se podrá acceder durante un pequeño período tiempo. Si es un fallo del sistema, será necesario tener preparado un sistema de recuperación o un sistema alternativo (una copia).

Durante este tiempo, los distintos desarrolladores no podrán trabajar de forma colaborativa ni acceder a las versiones almacenadas ni almacenar de nuevas. Ésta es una desventaja difícil de contrarrestar.

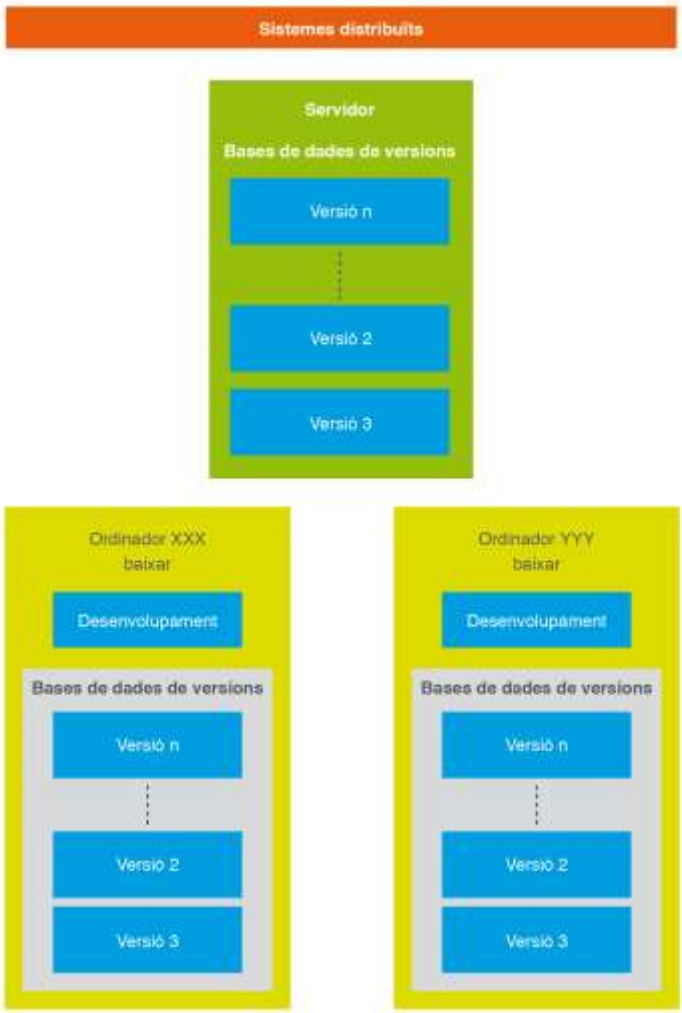
Debido a que el repositorio se encuentra centralizado en un único ordenador y una única ubicación, la creación de una rama se llevará a cabo en la misma ubicación que el resto del repositorio, utilizando una carpeta nueva para crear la duplicidad. Los puntos débiles

con el trabajo de las ramas serán los mismos que se han expuesto en general por en los sistemas de control de versiones centralizados.

Sistemas distribuidos

Los sistemas de control de versiones distribuidos ofrecen una solución al mismo desventaja ofrecida por los sistemas de control de versiones centralizados. La solución que ofrecen los sistemas distribuidos es

disponer cada ordenador de trabajo, así como el servidor, de una copia de las versiones almacenadas. Esta duplicidad de las versiones ofrece una disponibilidad que disminuye muchísimo las posibilidades de no tener accesibilidad a los archivos y las sus versiones.



¿Cuál es la forma de trabajar de este sistema? Cada vez que un ordenador cliente accede al servidor para tener acceso a una versión anterior, no sólo copian ese archivo, sino que hacen una descarga completa de los archivos almacenados.

Si el servidor tiene un fallo del sistema, el resto de ordenadores clientes podrán continuar trabajando y cualquiera de los ordenadores clientes podrá efectuar una copia entera de todo el sistema de versiones hacia el servidor para restaurarlo. La idea es que el servidor es el que gestiona el sistema de versiones, pero en caso de necesidad puede acceder a las copias locales que se encuentran en los ordenadores clientes.

5.2.3 Operaciones básicas de un sistema de control de versiones

Entre las operaciones más habituales de un sistema de control de versiones (tanto en los sistemas centralizados como en los distribuidos) se pueden diferenciar dos tipos: aquellas que permiten la entrada de datos en el repositorio y aquellas que permiten obtener datos del repositorio.

Entre las operaciones de introducción de datos en el repositorio se pueden encontrar:

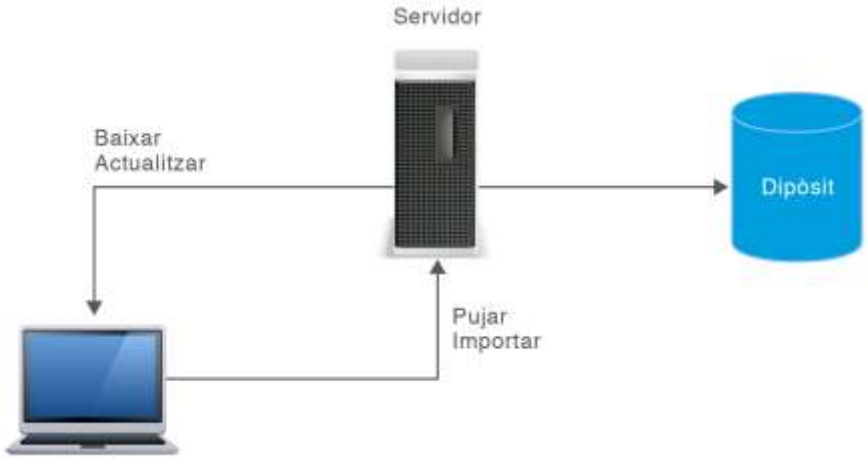
• **Importación de datos**: esta operación permite realizar la primera copia de seguridad o versionado de los archivos con los que se trabajará en local en el repositorio.

• Subir (**commit** o check in): esta operación permite enviar al repositorio los datos correspondientes a los cambios que se han producido en el servidor local. No se hará una copia entera de toda la información, sino que sólo se trabajará con los archivos que se hayan modificado en el último espacio de tiempo. Cabe destacar que no los envía al servidor; los cambios quedan almacenados en el repositorio local, que debe sincronizarse.

Entre las operaciones de exportación de datos del repositorio se pueden encontrar:

• Bajar (**check-out**): con esta operación se podrá tener acceso y descargar en el área de trabajo local una versión desde un repositorio local, un repositorio remoto o una rama diferente.

• Actualización (**update**): esta operación permite realizar una copia de seguridad de todos los datos del repositorio en el ordenador cliente con el que trabajará el programador. Será una operación que se podrá efectuar de forma manual, cuando el programador lo estime oportuno, o de forma automática, como en los sistemas distribuidos, donde cada vez que un cliente accede al repositorio se realiza una copia completa en local.



Actualmente, la opción más popular es una mezcla entre ambos sistemas: utilizar un servidor central y utilizar a los clientes un sistema distribuido. Conviene elegir como servidor central una plataforma con un alto nivel de seguridad.

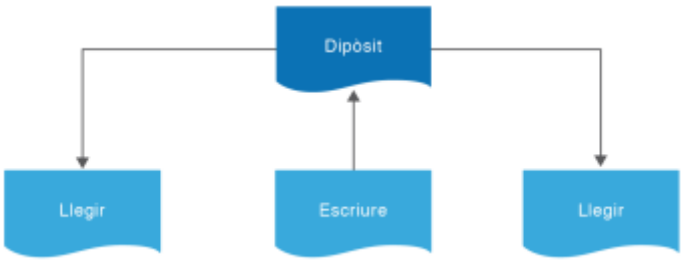
5.3 Depósitos de las herramientas de control de versiones

Un repositorio es un almacén de datos donde se guardará todo lo relacionado con una aplicación informática o unos datos determinados de un determinado proyecto.

El uso de repositorios no es una técnica exclusiva de las herramientas de control de versiones. Las aplicaciones y proyectos informáticos suelen utilizar depósitos que contienen información. Por ejemplo, un repositorio con información referente a una base de datos contendrá todo lo referente a cómo está implementada esta base de datos: cuáles tablas tendrá, qué campos, cómo serán estos campos, sus características, sus valores límites...

En el ámbito del control de versiones, tener este depósito supone poder contar con un almacén central de datos que guarda toda la información en forma de archivos y directorios, y que permite llevar a cabo una gestión de esta información.

Toda herramienta de control de versiones tiene un repositorio que es utilizado como un almacén central de datos. La información almacenada será toda la referente al proyecto informático que se estará desarrollando. Los clientes se conectarán a este repositorio para leer o escribir esta información, accediendo a información de otros clientes o haciendo pública su propia información. En la figura inferior se muestra un ejemplo conceptual de un repositorio centralizado donde acceden varios clientes por escribir o leer archivos.



Un repositorio es la parte principal de un sistema de control de versiones. Son sistemas diseñados para grabar, guardar y gestionar todos los cambios y informaciones sobre todos los datos de un proyecto a lo largo del tiempo.

Gracias a la información registrada en el repositorio se podrá:

- Consultar la última versión de los archivos que se hayan almacenado.
- Acceder a la versión de un determinado día y compararla con la actual.
- Consultar quién ha modificado un determinado pedazo de código y cuándo fue modificado.