Ingeniería del Software

1. Introducción a Git, Markdown y Eclipse

Javier Barbero Gómez

Ingeniería del Software
2º Curso Grado en Ingeniería Informática
Escuela Politécnica Superior de Córdoba
Universidad de Córdoba
jbarbero@uco.es

16 de septiembre de 2019

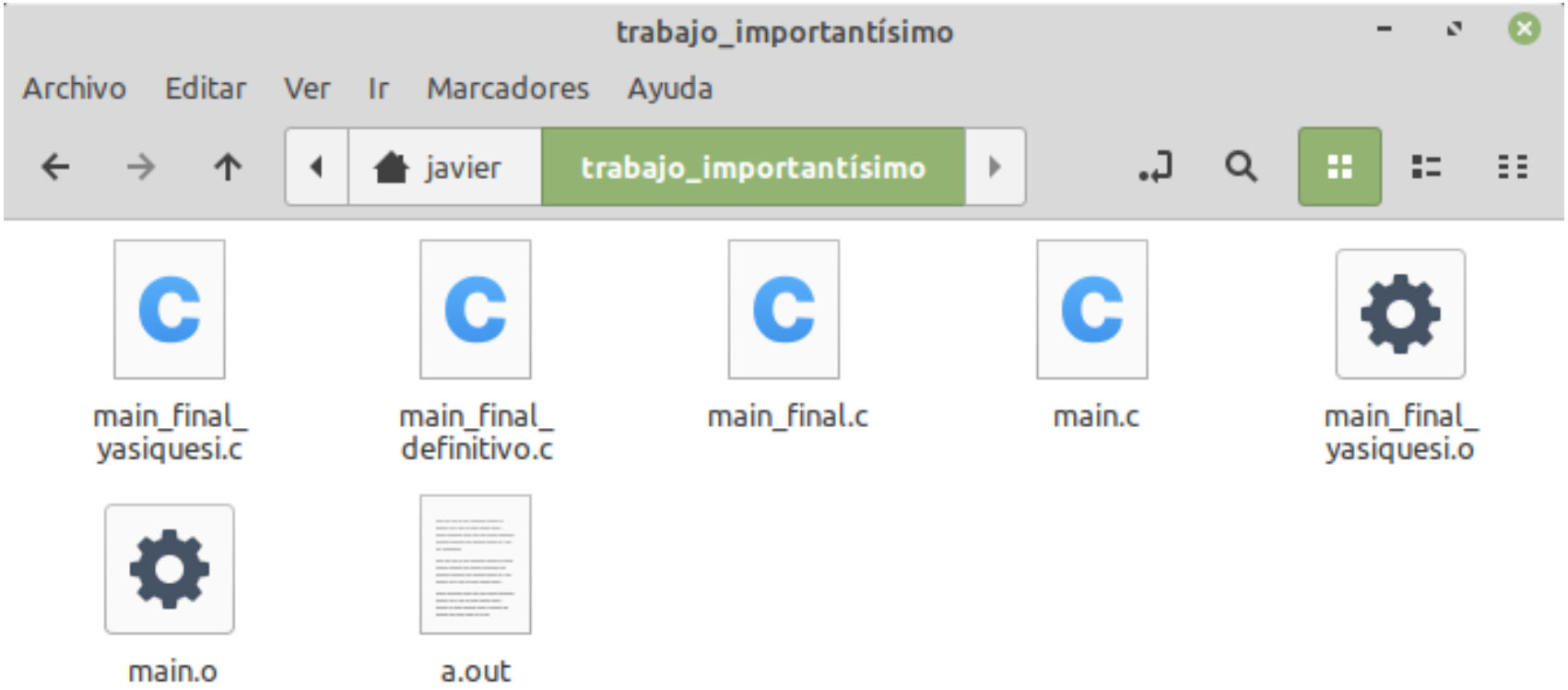
Contenidos

Evaluación

- Las entregas en Moodle se realizarán por medio del representante o líder de cada grupo.
- Se debe entregar en Moodle la dirección del repositorio de Github.
- Esta primera práctica *no* tendrá entrega, aunque su contenido es *esencial* para la realización de las siguientes.

Contenidos

CVS: Motivación I



CVS: Motivación II

- Necesidad de mantener todas las versiones del código fuente.
- Problemas en organizaciones para mantener el código actualizado.
- Coherencia de versiones.
- Conocimiento del cambio que ha provocado que el sistema no funcione.
- Fallos en el disco duro que suponen riesgo de información desactualizada.
- Satisfacer el compromiso de entrega.

Git y GitHub

Git

Sistema para el control distribuido de versiones de código. Fundamentalmente permite:

- Dar seguimiento a los cambios realizados sobre un archivo.
- Almacenar una copia de los cambios.

GitHub, GitLab, Bitbucket...

Sitio web donde podemos alojar un repositorio Git.

Ventajas

Git

- Historial y documentación de cambios.
- Habilidad de deshacer/rehacer cambios.
- Múltiples versiones de código.
- Habilidad de resolver conflictos entre versiones de distintos programadores.
- Copias independientes.

GitHub, GitLab, Bitbucket...

- Copia remota.
- Herramientas de colaboración (*issue tracking*, aportaciones de terceros...)

Instalación

- Para instalar Git: `https://git-scm.com`
- En este curso se utilizará Git a través de líneas de comandos.
- Para Eclipse existen *plugins* integrados:
  `https://www.eclipse.org/egit`

Configuración básica

Nombre del administrador:

```
git config --global user.name "Javier Barbero Gomez"
```

Correo electrónico:

```
git config --global user.email jbarbero@uco.es
```

Editor de texto:

```
git config --global core.editor "gedit"
```

Color de la interfaz:

```
git config --global color.ui true
```

Listado de la configuración:

```
git config --list
```

Conceptos básicos: objetivo de Git

Gestionar un **proyecto**, o **conjunto de ficheros**, y sus **cambios en el tiempo**. Git almacena todo esto en una estructura denominada *repositorio*.

Conceptos básicos: repositorio

Componentes de un **repositorio**:

- Conjunto de *commits*
- Conjunto de *heads*

Toda esta información se almacena en un directorio denominado `.git` en la raíz del proyecto

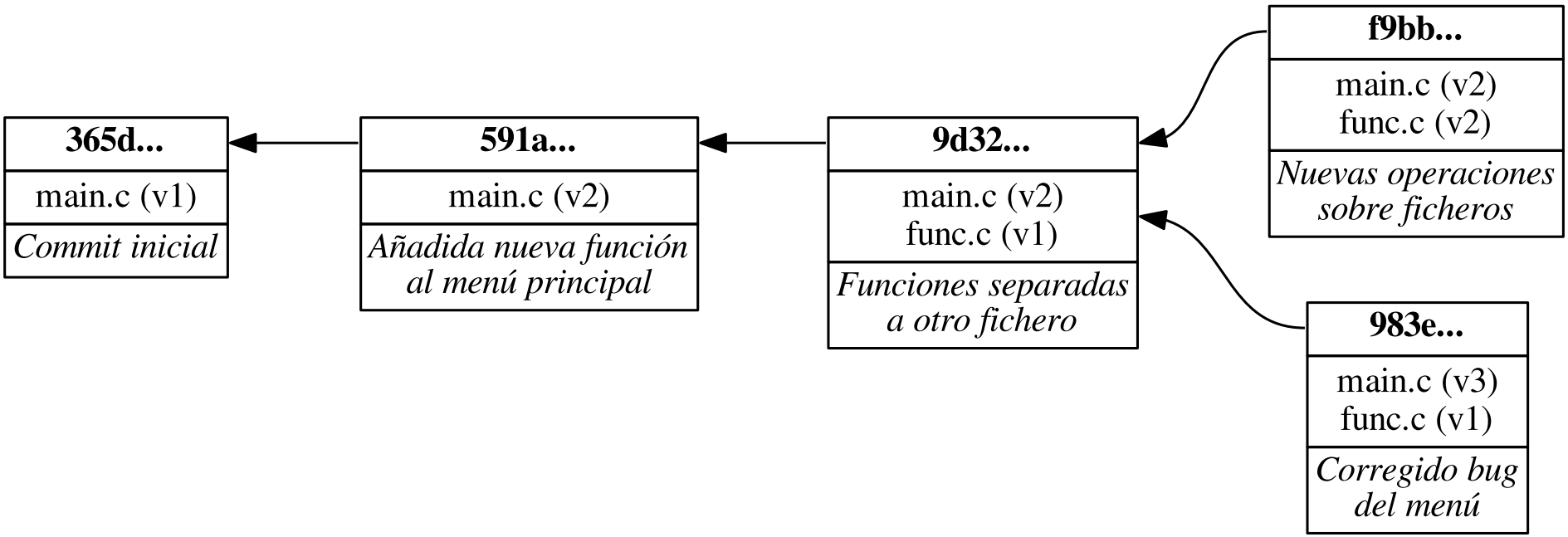
Conceptos básicos: *commit*

Un objeto **commit** se compone de:

- Conjunto de *ficheros* y su estado en un punto del tiempo
- Una *referencia al commit padre* (aquel que fue editado para obtener este commit)
- Un *hash* del contenido del commit, que lo identifica de manera única
- Una *descripción*

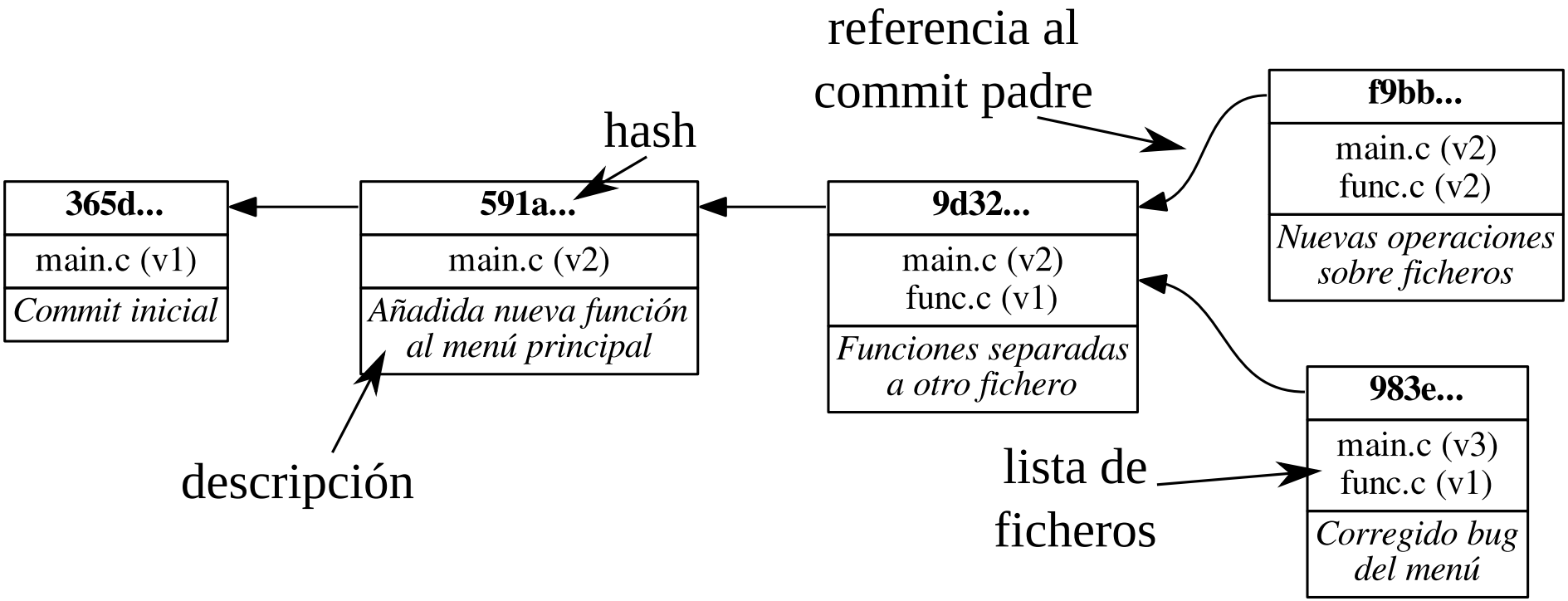
Conceptos básicos: *commit*

Los commits de un repositorio Git forma un *grafo acíclico dirigido* o *árbol*



Conceptos básicos: *commit*

Los commits de un repositorio Git forma un *grafo acíclico dirigido* o *árbol*



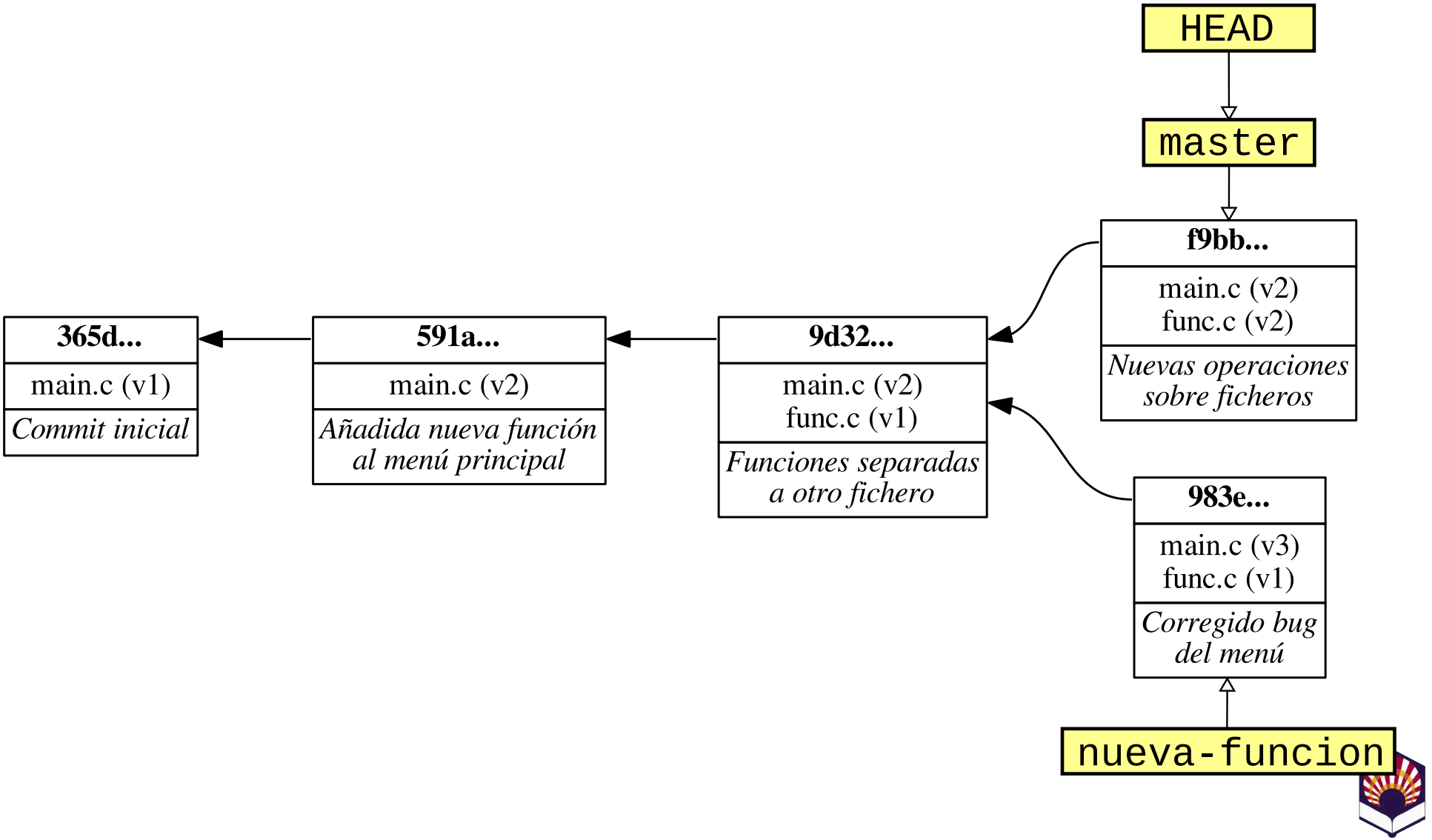
Conceptos básicos: *head*

Un **head** no es más que una *referencia a un objeto commit*:

- Tiene un *nombre descriptivo*
- Por defecto se crea un *head* denominado `master`
- Puede haber cualquier número de *heads* en un repositorio
- En todo momento hay un *head* seleccionado como *actual*. Se refiere a este como `HEAD` (en mayúscula)

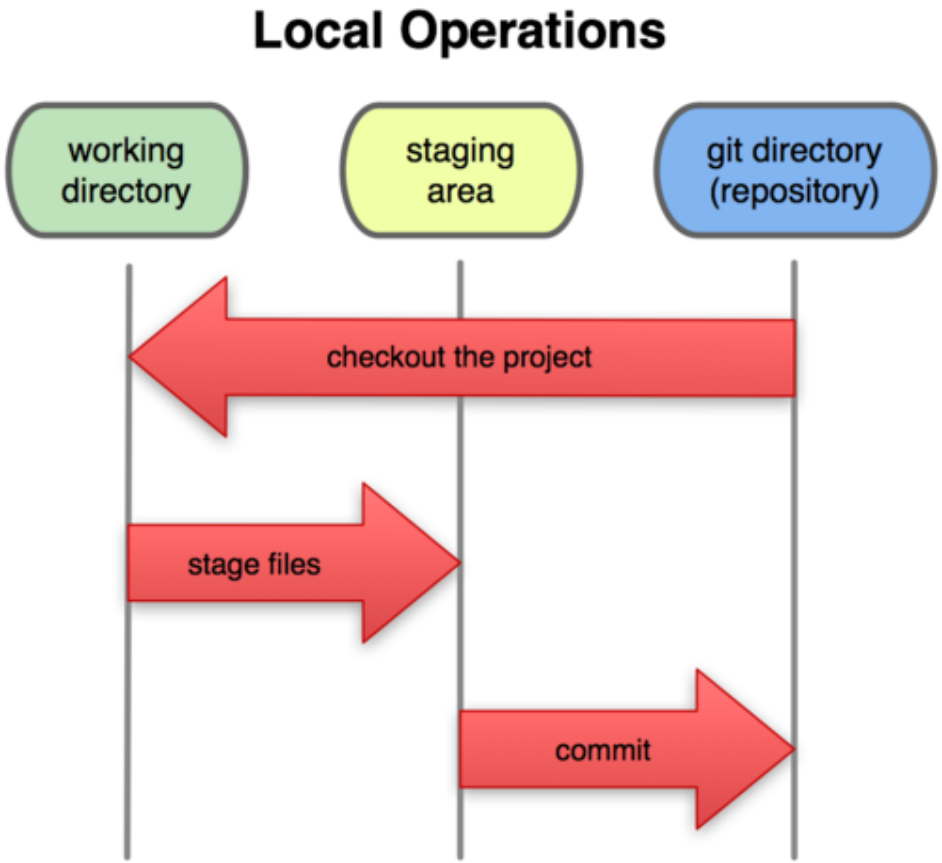
A menudo se refiere a un *head* como **branch** (rama), generalmente para referirse a un *commit* y a toda su ascendencia (la lista de *commits* desde éste hasta el inicial). Son términos *equivalentes*.

Conceptos básicos: *head*



Los tres estados de Git

- *Working directory*: Lo que hay en el directorio raíz, lo que «ve el editor».
- *Repositorio*: Los datos al completo del repositorio (todo el historial de cambios), dentro del directorio `.git`.
- *Staging area*: Especie de «limbo» donde agregamos los cambios para el siguiente *commit*.



Comandos básicos I

Iniciar repositorio en un directorio:

```
git init
```

Ver el estado del *directorio de trabajo* y el área de *staging*:

```
git status
```

Agregar cambios al área de *staging* (ojo, recursivo):

```
git add <fichero>
```

Validar cambios del área de *staging*, crear un nuevo objeto *commit*:

```
git commit
```

Visualizar el historial de *commits*:

```
git log
```

Comandos básicos II

Trabajar sobre un *commit* distinto:

```
git checkout <hash o head>
```

Crear un nuevo *head* que apunte al *commit* actual:

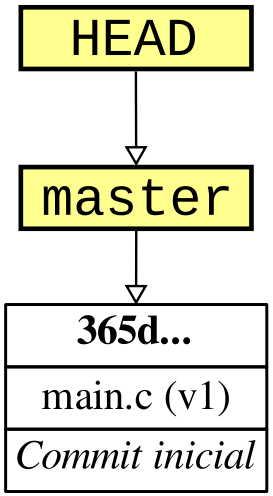
```
git checkout -b <nombre>
```

Visualizar todo el árbol de *commits*:

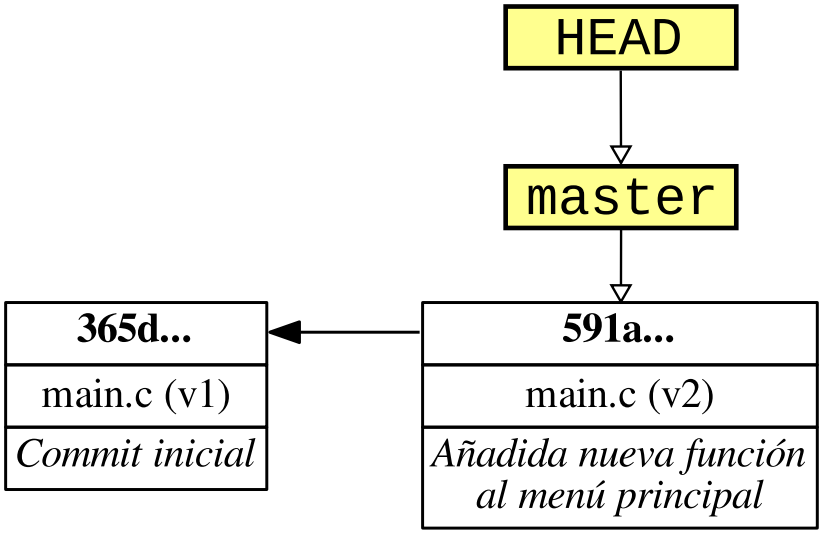
```
git log --graph --all
gitk --all # Necesita instalar la herramienta "gitk"
```

Ejemplo práctico

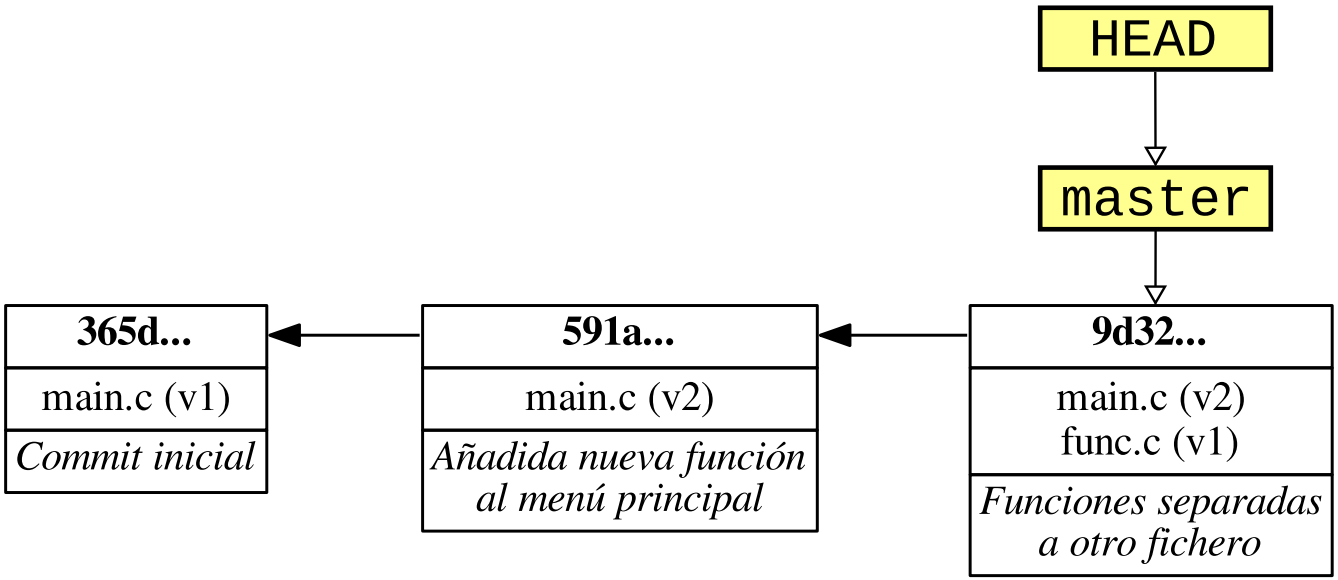
Ejemplo práctico



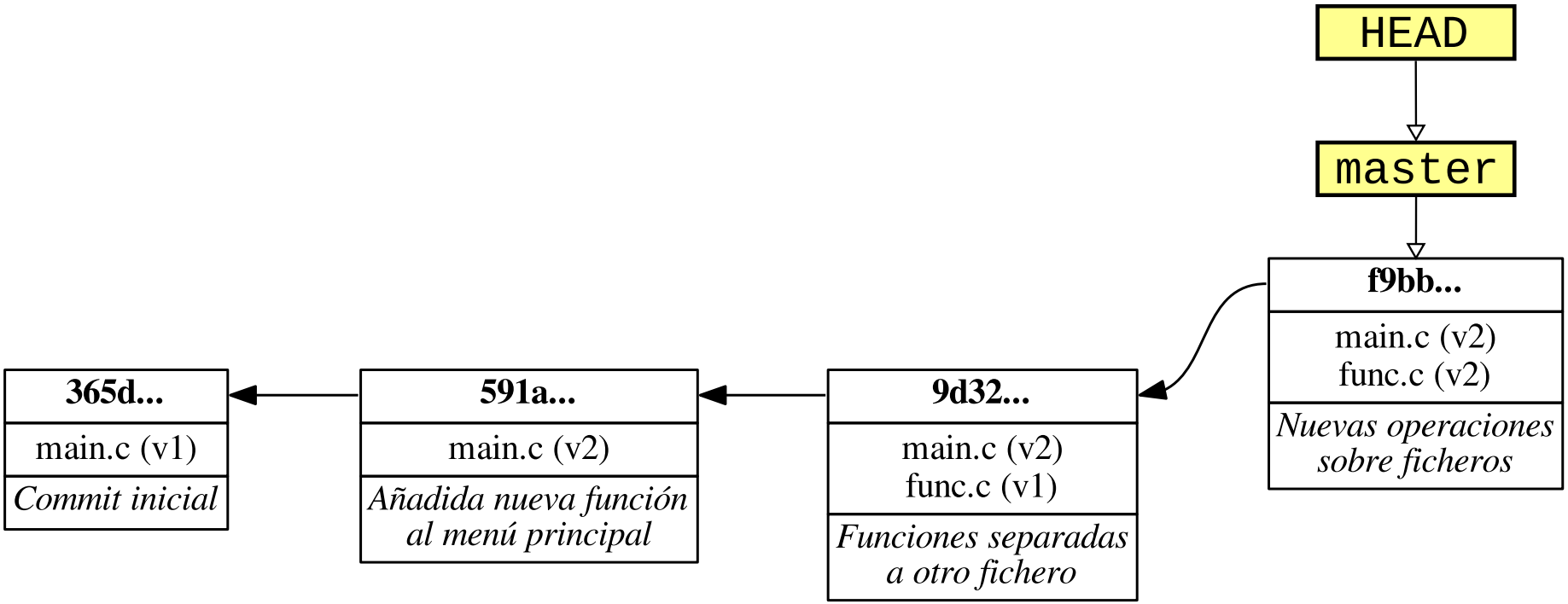
Ejemplo práctico



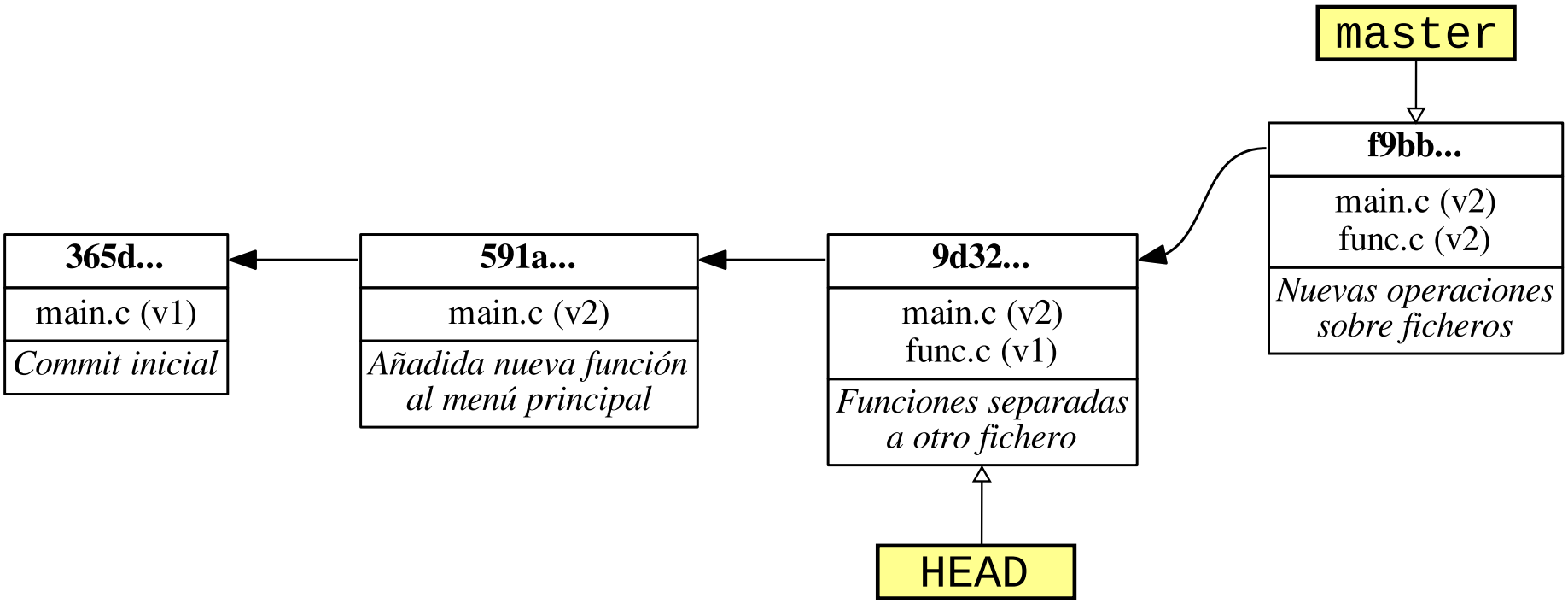
Ejemplo práctico



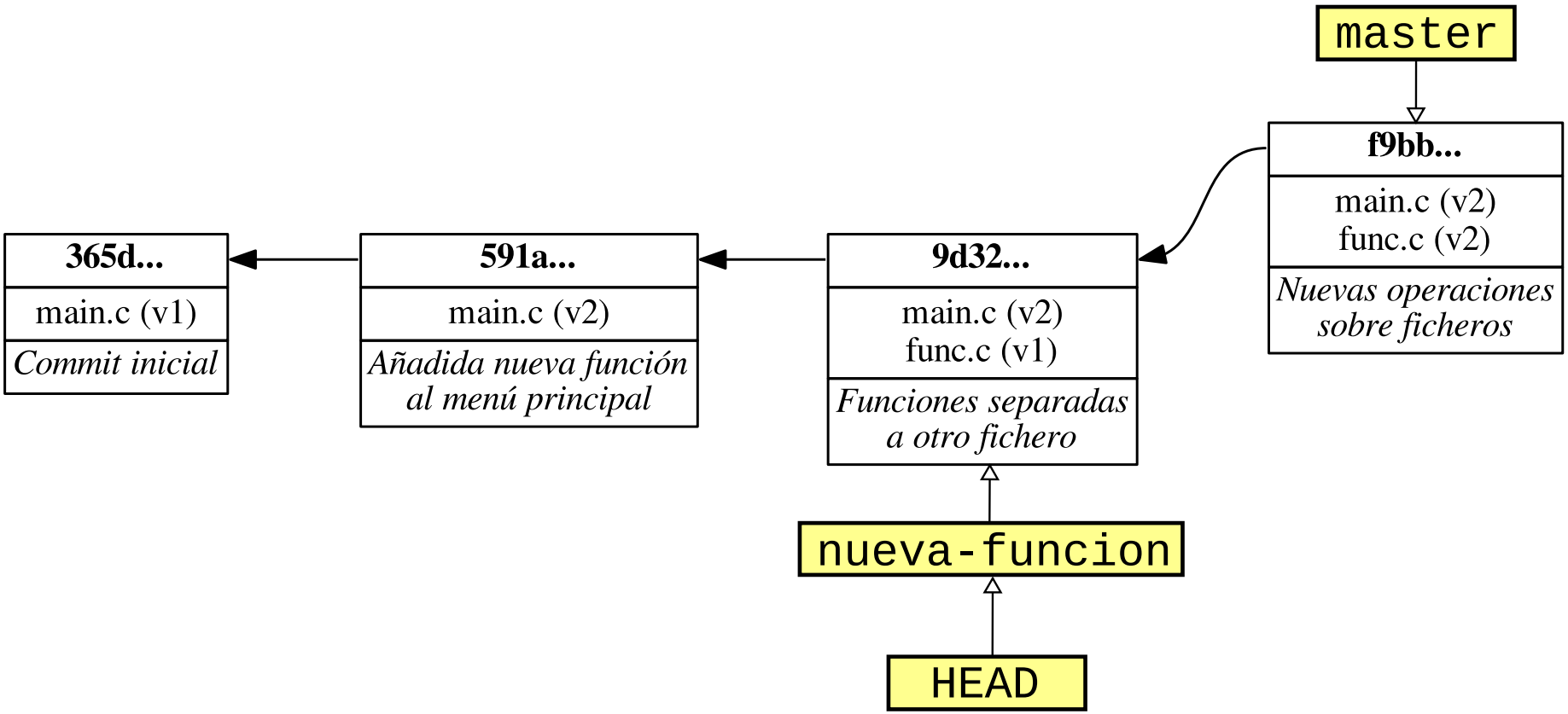
Ejemplo práctico



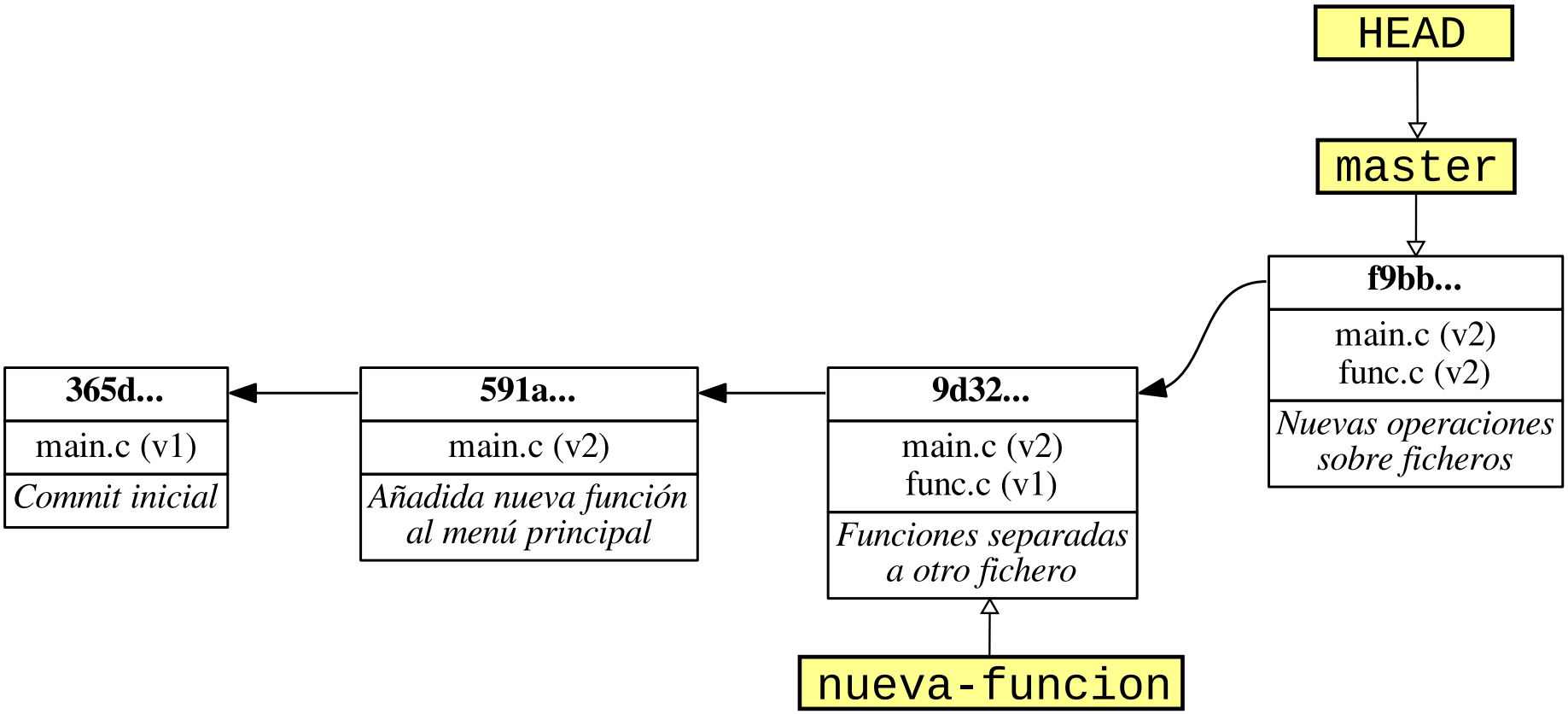
Ejemplo práctico



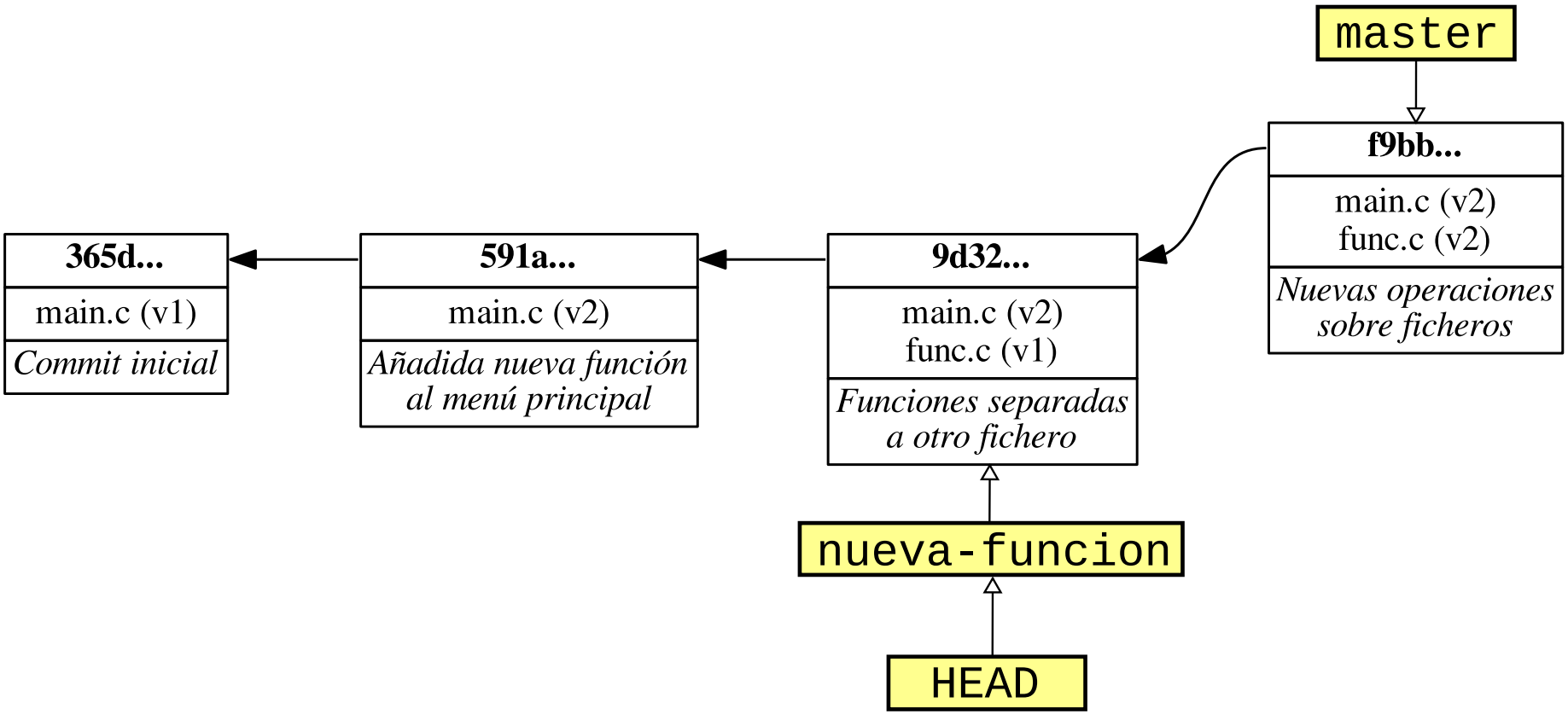
Ejemplo práctico



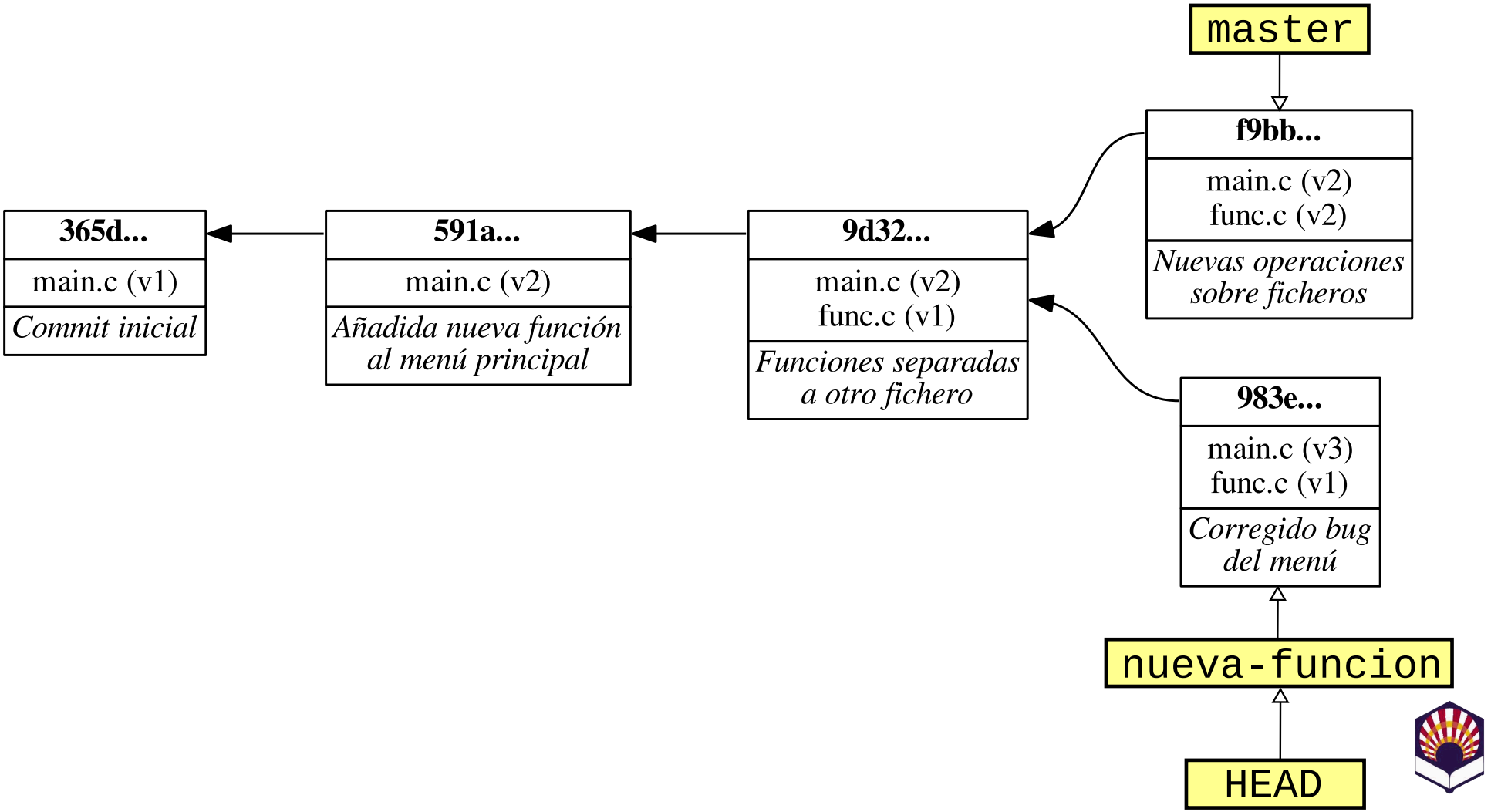
Ejemplo práctico



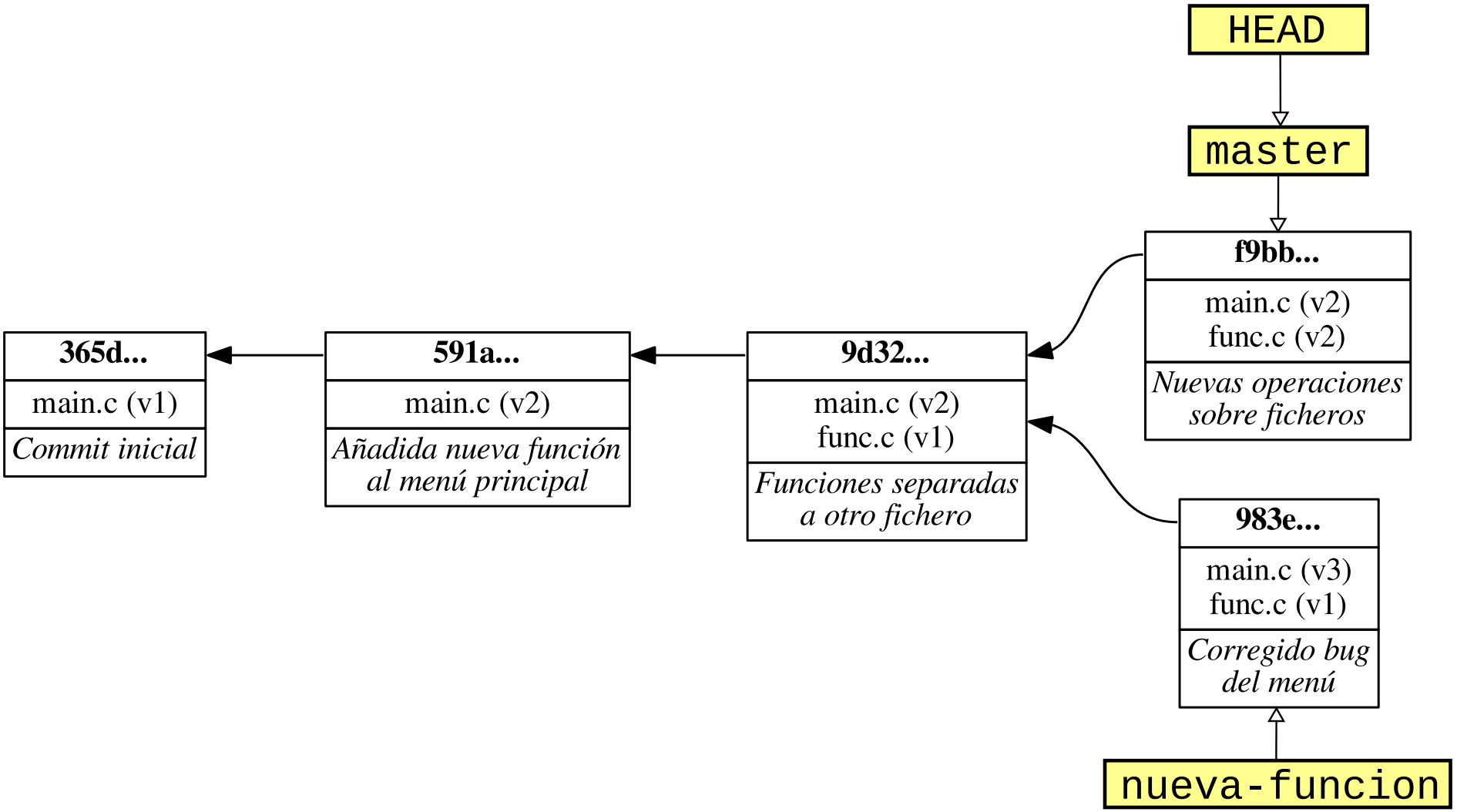
Ejemplo práctico



Ejemplo práctico



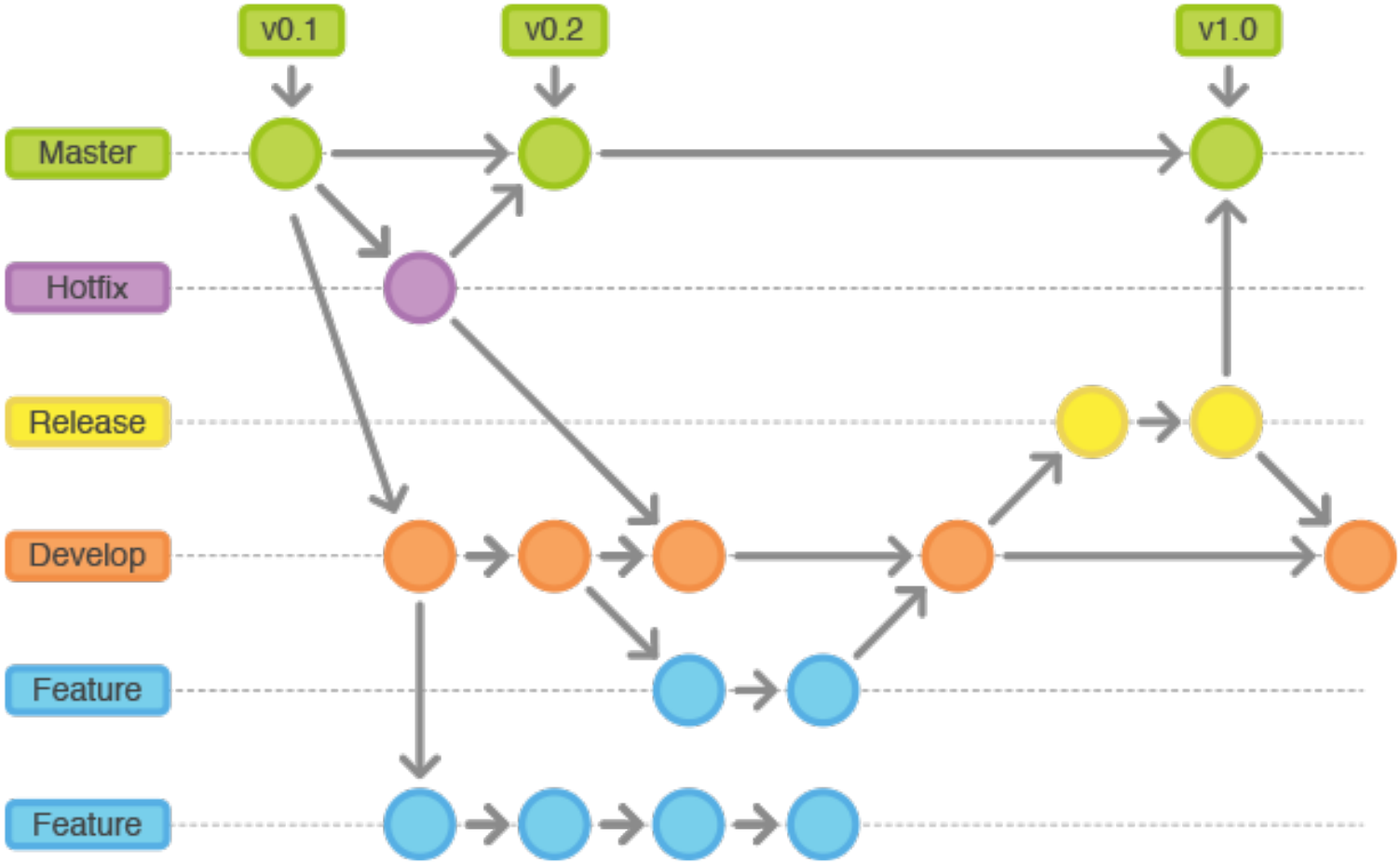
Ejemplo práctico



Por qué usar distintas ramas

- Un patrón común de uso en Git es mantener una rama principal (habitualmente *master*) y crear nuevas ramas para implementar nueva funcionalidad.
- Idealmente, según este patrón, la rama principal siempre está en un estado «publicable», y el resto podrían contener trabajo «a medio hacer», funcionalidad inestable, etc. Este es un buen patrón de uso para *colaborar con otros desarrolladores* (o incluso para organizar tu propio trabajo).
- Si cada desarrollador trabaja en su propia rama, puede realizar *commits* sin interferir en el trabajo ajeno.
- En resumen, **asegúrate de trabajar en tu propia rama**, de esta forma no es necesario tener tanto cuidado al agregar nuevos *commits*.

Uniendo el trabajo de distintas ramas: *merge* I



Uniendo el trabajo de distintas ramas: *merge* II

Nos posicionamos en la rama donde queremos añadir los cambios y los unimos con la rama que contiene los cambios. Ejemplo:

```
git checkout master
git merge nueva-funcion # Se unirán los cambios de
 ↪  nueva-funcion a master

# En lugar de "merge" podría hacerse este comando
 ↪  equivalente
git pull . nueva-funcion
```

Uniendo el trabajo de distintas ramas: *merge* III

Git trata de simplificar en la medida de lo posible esta unión. El proceso que lleva a cabo es el siguiente:

1. Identificar el «ancestro común» de ambas ramas
2. Intentar tratar con los «casos fáciles»
   - Si el ancestro común es la rama con los cambios, no se hace nada
   - Si el ancestro común es la rama actual, hacer un *fast-forward*
3. En caso contrario, comprobar las diferencias entre ambas ramas
4. Intentar unir ambos cambios
5. Si no hay conflictos, se crea un nuevo *commit* con dos padres: la rama actual y la de los cambios. Este será el nuevo *commit* actual
6. Si hay conflictos, se añaden marcadores a los ficheros y se informa al usuario para que los corrija, NO se crea ningún *commit*

Merge: *fast-forward*

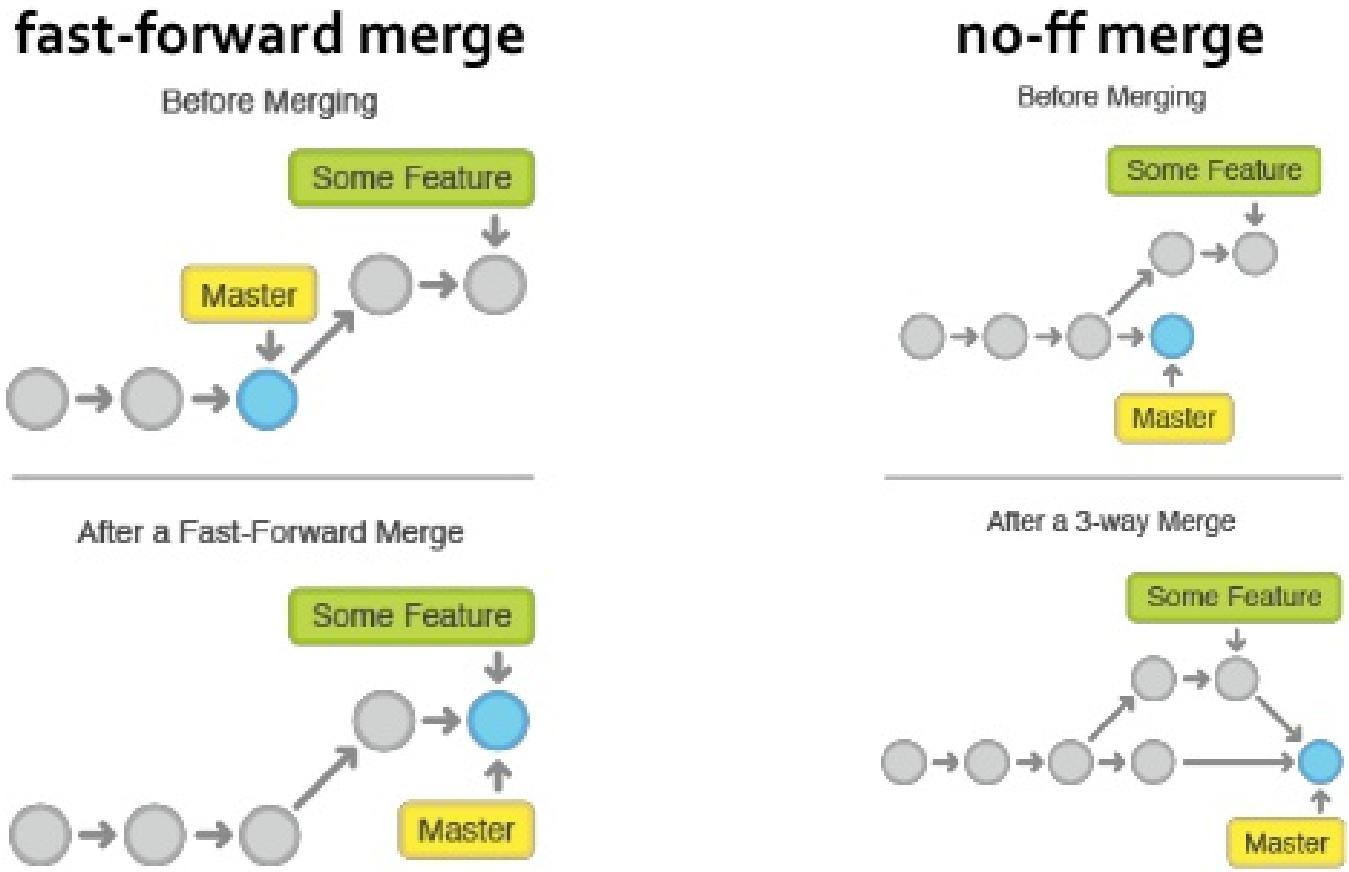


Figura: Diferencia entre *merge* con y sin FF (fuente: Atlassian)

Merge: Conflictos

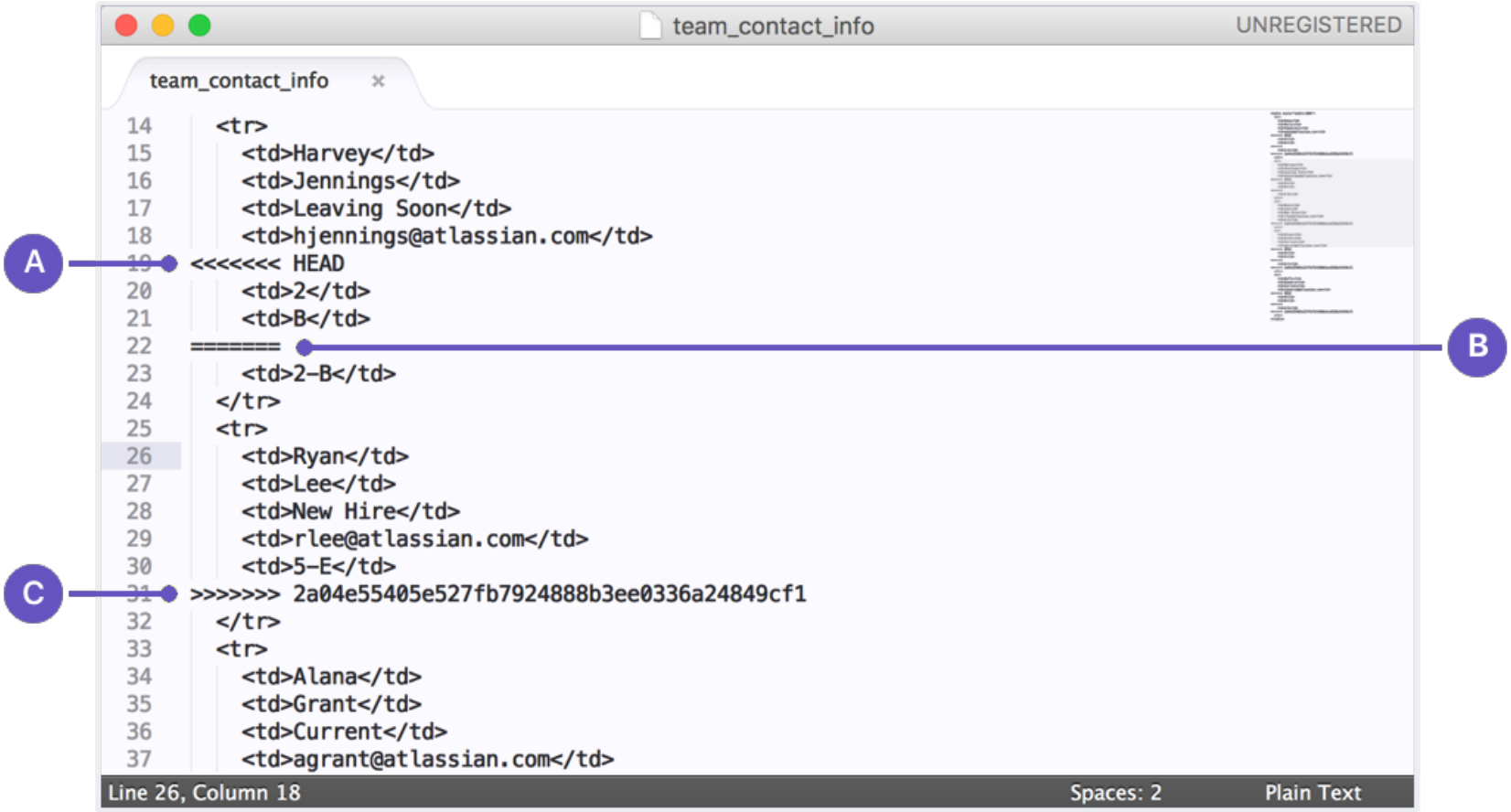


Figura: Marcas indicadas por Git para resolver conflictos en una operación de *merge* (fuente: Atlassian)

Merge: Casos de uso

- Añadir cambios realizados en otra rama (nueva funcionalidad, arreglos de *bugs*...) a la rama principal
- Actualizar una rama secundaria con los últimos cambios de otra rama principal, para estar al día y evitar conflictos en el futuro

Colaborando con otros desarrolladores

En Git, todos los datos del repositorio se almacenan de forma local, no es posible modificar el repositorio de otro desarrollador.

Modelo distribuido

Git utiliza un *modelo distribuido*, asume que no existe un repositorio central.
Si bien es posible utilizar un repositorio en concreto como central, es importante comprender el modelo distribuido.

Obtener una copia desde una fuente remota

Comando:

```
git clone <ruta/url>
```

Esto realiza lo siguiente:

1. Crea un directorio e inicializa un repositorio en él
2. Copia todos los *commits* de la fuente a este repositorio
3. Añade una *referencia remota* llamada `origin` (nombre por defecto)
4. Añade un «*head* remoto», que corresponde con un *head* en el origen (p. ej.: `origin/master` sería la rama `master` en la fuente `origin`)
5. Crea un *head* local asociado a este *head* remoto

Obtener una copia desde una fuente remota: ejemplo

Ejecutar lo siguiente:

```
git clone https://github.com/ayrna/orca.git
git branch -vv -a
# -vv Muestra más información sobre cada rama
# -a Muestra tanto ramas locales como remotas
```

```
* master                          400d6c7 [origin/master] Fixed link
  remotes/origin/HEAD             -> origin/master
  remotes/origin/develop          ca25ae1 Merge pull request #65 from ayrna/master
  remotes/origin/fix_links        e6928ef Synced MD with notebooks
  remotes/origin/master           400d6c7 Fixed link
  remotes/origin/orca-extra-methods b43c379 Merge pull request #68 from ayrna/master
```

Obtener nuevos cambios desde una fuente remota

```
git fetch # por defecto, origin
git fetch <fuente>
```

Esto obtiene los posible cambios de la fuente remota, pero *no interfiere en los datos locales*.
Es decir, actualiza los *heads* que comienzan por «`origin/`», no los demás

```
git pull origin master
```

Esto actualiza la rama local `master` con la remota `origin/master`

Una forma resumida para realizar los dos comandos anteriores:

```
git pull
```

Enviar cambios a una fuente remota

```
git push # sincroniza todas las ramas con tracking
git push <fuente> <rama> # más selectivo
```

El repositorio receptor añade los nuevos *commits* a la rama correspondiente y la actualiza. El repositorio local también actualiza su información remota.

Ejemplo práctico, trabajo remoto

+

Contenidos

Lenguaje Markdown

- Markdown es un lenguaje de etiquetado ligero que simplifica la elaboración de documentos.
- Se ideó pensando en una herramienta para escribir páginas web en un texto simple fácil de leer.
- Actualmente, se utiliza para documentar software ya que al ser texto plano puede entrar dentro de cualquier sistema de control de versiones e incluye muchas extensiones para colorear código fuente en distintos lenguajes.

Sintaxis I

Formato	Sintaxis
Negrita	`**Texto en negrita**`
Cursiva	`*Texto en cursiva*`
Lista con viñetas	`1. Primera línea` `2. Segunda línea`
Lista anidada	`* Primer nivel` `*  Segundo nivel`

Encabezados (hasta 6 niveles)	`# Encabezado primer nivel`
	`## Encabezado segundo nivel`
	`### Encabezado nivel tres`

Citas en bloque	`> Las citas en bloque deben comenzar y terminar con una línea en blanco.`

Sintaxis II

Formato	Sintaxis
Código en línea	‘Esto es codigo en linea‘
Bloques de código	‘‘‘ Ejemplo de bloque ‘‘‘
Imágenes	![Texto alternativo](url_imagen)
Vínculos	[Texto del vínculo](url_enlace)
Imágenes con vínculos	[![Texto alternativo](url_imagen)](url_enlace)
Línea horizontal	--- (Salto de línea antes y después)

Contenidos

Eclipse

Eclipse es un entorno integrado de desarrollo (IDE).

- Se diseñó inicialmente como IDE para Java, sin embargo ahora soporta otros lenguajes como C++.
- Ayuda a escribir código más rápido y libre de algunos errores sintácticos, y ayuda a mantener un estilo de programación homogéneo.
- Facilita la depuración de código.
- Hay una amplia documentación.

Instalación

- Utilizaremos eclipse para C++.
  Eclipse para C++
- Para eclipse existen *plugins* integrados con git.
  https://www.eclipse.org/egit
- En las aulas, disponible bajo el comando `eclipse`
  - Será necesario instalar el *plugin* para C++ en nuestro home:
    `http://www.eclipse.org/downloads/download.php?file=/tools/cdt/releases/8.6/cdt-8.6.0.zip`

Uso básico

- Instalar plugins: Help > Install new software...
- Crear un nuevo proyecto: File > New > C/C++ Project
- Crear un fichero: File > New > Source file/Header file
- Autocompletado de código: Ctrl + Espacio (Edit > Content assist)
- Autoformato: Ctrl + Mayús + F (Source > Format)
- Resolver dependencias (auto-*include*): Ctrl + Mayús + O (Source > Organize includes)
- Cambiar entre Debug/Release: Project > Build configurations > Set active
- Compilar: Ctrl + B (Project > Build all)
- Ejecutar: Ctrl + F11 (Run > Run)

Debugging

- Comenzar debugging: F11 («Run → Debug»)
- Añadir breakpoint: Doble clic junto al número de línea o Ctrl + Mayús + B
- Continuar ejecución (hasta el siguiente breakpoint): F8
- Siguiente línea («Step into»): F5
- Siguiente línea, no entrar en función («Step over»): F6

Contenidos

Recursos

Recursos Git:

- Understanding Git Conceptually
- Configurar Git
- Usar login SSH en Git
- Guía sencilla de Git.
- Pro Git book.
- Git cheat sheet

Recursos Markdown:

- Markdown Cheatsheet.
- Guía en castellano extendida.

Recursos Eclipse:

- Eclipse para C++

Ingeniería del Software

1. Introducción a Git, Markdown y Eclipse

Javier Barbero Gómez

Ingeniería del Software
2º Curso Grado en Ingeniería Informática
Escuela Politécnica Superior de Córdoba
Universidad de Córdoba
jbarbero@uco.es

16 de septiembre de 2019