# Branching Patterns


## Integrantes:

-Raúl Alonso García (UO270656)

-Enrique Vera Cueto (UO246711)

-Pablo Rodríguez Rodríguez (UO271246)


# Introducción

Los sistemas actuales de control de versiones permiten crear ramas del código del base para que cada miembro del equipo trabaje por separado, estos patrones de ramificación ayudan a hacer esto de forma efectiva.

# Patrones básicos

Estos patrones establecen las bases sobre las que funcionen el resto, y que muestran las estrategias fundamentales.

**Source Branching**. Este patrón indica que los desarrolladores deben crear una copia del código base y trabajar sobre ella recordando todos los cambios que se hagan sobre ella mediante commits.

**Mainline** .Se debe tener una mainline de donde copiar el código en integrar el trabajo de todos. La mainline o línea principal es el estado actual del producto accesible para todos los desarrolladores.

**Healhty Branch**.Se deben tener las ramas limpias, especialmente la principal, haciendo comprobaciones de que no hay errores graves cada vez que se hacen cambios.

# Patrones de Integración

En algún punto del desarrollo es necesario integrar el trabajo de todos los participantes, estos patrones buscan las mejores estrategias para hacerlo.

**Mainline Integration**. Los desarrolladores integran su trabajo con el de la línea principal trayendo está a su entorno (pull), uniendo su trabajo (merge) y subiéndola de nuevo al espacio común (push). Una línea principal da una visión más clara del proyecto y simplifica mucho la integración.

**Integration Frequency**. Con este concepto al que hacen referenciaa los siguientes patrones se hace referencia a la frecuencia con la que los commits se integran en la línea principal. Con una baja frecuencia habrá que integrar menos veces, pero estas integraciones serán mucho más grandes y difíciles, y podrán presentar errores graves con más probabilidad, si se hacen integraciones más frecuentes estas serán más fáciles y con menor probabilidad de errores, además que estos se encontrarán antes

**Feature Branching**. Con este patrón cada tarea o característica del programa se desarrollará en su propia rama, y se integrará en la rama principal cuando se haya terminado, con la gran ventaja que no habrá tareas a medio hacer en la línea principal. Habitualmente esto implica una frecuencia de integración normalmente baja salvo excepciones cuando las tareas son muy pequeñas.

**Continuos Integration**. Este patrón es considerado el opuesto al anterior, aunque en ocasiones se pueden usar juntos, los desarrolladores deben integrar su trabajo en la rama principal tan pronto como tengan un commit en buen estado que puedan compartir. Con este patrón es común tener tareas a medio hacer en la línea principal por lo que se deberán tomar precauciones haciendo tests para evitar que estas causen problemas graves, también se debe evitar el acceso a ellos para que no se usen si el proyecto se va a mandar a producción.

**Pre-Integration review**.Todo commit que vaya a ser integrado en la rama principal debe ser revisado antes de poder aceptarse.

# Patrones de Producción

Estos patrones están pensados para mantener el producto en una versión always-releasable, para así poder lanzar el código directamente desde la rama a producción.

El primer patrón es el llamado **RELEASE BRANCH** este patrón consiste en crear una copia de la rama principal, esta nueva rama será la que contendrá una versión del producto lista para lanzamiento.

Es recomendable usarlo cuando al equipo le cuesta mantener la rama principal en un estado saludable.

El siguiente patrón es el llamado **MATURITY BRANCH**, consiste en crear ramas que guarden diferentes versiones del código que cumplan con un nivel de madurez.

Si juntamos los dos patrones anteriores el patrón se llamaría **Long Lived Release Branch,** convertimos la rama release en una rama maturity.

El siguiente patrón es el llamado **ENVIROMENT BRANCH**. Este es un patrón sencillo, consiste en crear una nueva rama para cada entorno y así almacenar la configuración necesaria para cada entorno.

El cuándo usarlo está claro, cuando nuestra aplicación trabaja con diferentes entornos.

El siguiente patrón es el llamado **HOTFIX BRANCH**. Este consiste en crear una rama nueva para solucionar un error rápidamente. Muy útil cuando el bug es muy grave y es prioritario arreglarlo lo antes posible.

El siguiente patrón se llama **RELEASE TRAIN**. Con este patrón lo que hacemos es crear una rama por cada fecha de lanzamiento.

Una variación de este patrón sería que en vez que la nueva rama del siguiente ciclo se cree cunado la anterior ya terminó, se creen todas a la vez.

El último patrón de producción es el **RELEASE-READY MAINLINE**. Este consiste en mantener la rama principal lo suficientemente estable y saludable para que siempre pueda ser puesta en producción.

# Otros patrones

La **Experimental Branch** recopila el trabajo experimental en una code base que no se espera que este unida con el producto final. Rama donde los desarrolladores quieren probar algunas ideas, pero no esperan que estos cambios se vayan a integrar en la línea principal. Lo más posible es que el código de esta rama no vaya a ser utilizado en la mainline. Se hace de forma “poco limpia” y si al final se quiere añadir a la mainline se aplicaría en esta la idea del experimento. En git, una vez acabado el trabajo con esta rama, añadimos un tag y la borramos (el nombre empieza por “exp”).

Una **Future Branch** utilizada para cambios demasiado invasivos para los que las estrategias utilizadas normalmente no son útiles. Es un patrón poco común, pero aparece a veces cuando las personas usan la Integración Continua cuando realizamos un cambio muy intrusivo en la base de código y las técnicas que se han ido usando usualmente no se integran bien. La gran diferencia entre Future Branch y Feature Branch es que solo hay un Future Branch.

La **Colaboration Branch** utiliza una rama creada para que un desarrollador comparta el trabajo con otros miembros del equipo sin una integración formal. Cuando un equipo usa mainline, la mayor parte de la colaboración se produce en la rama mainline. Solo cuando se produce la Mainline Integration el resto de los miembros ven lo que está haciendo uno de los desarrolladores. Estas ramas de colaboración se vuelven cada vez más útiles cuando disminuye la frecuencia de integración. Es improbable que los equipos que utilizan Continuous Integration necesiten una de estas ramas. Si varias personas trabajan juntas en un experimento harán que esta rama se convierta en una Collaboration Branch.

El **Team integration Branch** permite que los miembros del equipo se integren entre sin integrarse con todos los miembros del proyecto que utilizan la línea principal (sub-equipos). El equipo trata esta rama como una línea principal dentro de la línea principal del proyecto. Un factor más importante para las ramas de integración de equipos es la diferencia en la frecuencia de integración deseada. Si el proyecto en general espera que los equipos tengan ramas de características de un par de semanas de duración, pero el sub-equipo prefiere la Continuous Integration, entonces el equipo puede configurar una Team Integration Branch, hacer Integración Continua con eso e integrar la función que están trabajando con mainline una vez que esté hecho.

# Branching Policies

**Git-Flow** usa Mainline, (llamado "desarrollar") en un único repositorio de origen. Utiliza Feature Branching para coordinar a múltiples desarrolladores. Se anima a los desarrolladores a utilizar sus repositorios personales como rama de colaboración para coordinarse con otros desarrolladores que trabajan en trabajos similares. No dice nada

sobre la longitud de las ramas de características, por lo tanto, ni la frecuencia de integración esperada.

**GitHub Flow.** La diferencia esencial entre los dos es un tipo diferente de producto, lo que significa un contexto diferente y, por lo tanto, patrones diferentes. Git-Flow asumió un producto con varias versiones en producción. GitHub Flow asume una única versión en producción con integración de alta frecuencia en Release-Ready Mainline. Release Branch no es necesario. Llama a su línea principal "master".

Una alternativa de política de ramificación a Git-flow y GitHub Flow es **Trunk-Based Development**. Enfoca en hacer todo el trabajo en Mainline y así evitar cualquier tipo de ramas de larga duración. Los equipos más pequeños se comprometen directamente con la línea principal mediante la Mainline Integration, los equipos más grandes pueden usar la ramificación de funciones de corta duración.