

Universidad de Sevilla

Escuela Técnica Superior de Ingeniería Informática

# Metodología de Gestión de la Configuración




Grado en Ingeniería Informática – Ingeniería del Software

Proceso de Software y Gestión 2

Curso 2022 – 2023


Fecha	Versión
15/02/2023	1.0

Grupo de prácticas: G4-43		
Autores por orden alfabético	Rol	Descripción del rol
Alcobendas Santos, Jose Javier - 53986326J	Developer	Miembro del equipo de desarrollo
Campos Garrido, Juan Jesús - 47547107N	Scrum Master	Encargado de facilitar la labor de Scrum
Nunes Ruiz, Javier - 29517615J	Developer	Miembro del equipo de desarrollo
Pizarro López, Eduardo - 77933507P	Developer	Miembro del equipo de desarrollo
Reyes Alés, David - 29504757N	Developer	Miembro del equipo de desarrollo

	Proceso de Software y Gestión 2 Documentación del Sprint 2
	<b>Control de Versiones</b>


### Control de Versiones

Fecha	Versión	Descripción
15/02/2023	0.1	Creación del documento
15/02/2023	0.7	Metodología definida
22/02/2023	1.0	Versión final corregida

	<div>Proceso de Software y Gestión 2</div>
---	--

## Índice de contenido

1. Introducción	2
2. Objetivo	2
3. Contenido	2
3.1 Estándar de código	2
3.2 Política de mensajes de commit	3
3.3 Estructura de repositorios y ramas por defecto	3
3.4 Estrategia de ramificación	4
3.4.1 Cómo desarrollar ramas de características	4
3.4.2 Cómo preparar lanzamientos	4
3.4.3 Cómo corregir errores en producción	4
3.5 Políticas de versionado	5
3.6 Definición de hecho	5
4. Conclusiones	5
5. Referencias	5

	<div>Proceso de Software y Gestión 2</div>
---	--

## 1. Introducción

Tras haber trabajado en varias tareas sin seguir ningún patrón, es momento de ponernos todos de acuerdo para seguir la misma metodología a la hora de realizar tareas y de eso mismo, nos encargaremos en este documento.

## 2. Objetivo


El objetivo de este documento es poder establecer unos estándares que tendremos que respetar a la hora de realizar código, de crear commits, gestionar conflictos o solucionar errores. Seguir estos estándares nos ayudará a poder entender más fácilmente el trabajo de los demás integrantes ya que lo realizarán de la misma forma que nosotros mismos.

## 3. Contenido

### 3.1 Estándares de código

En cuanto a los estándares de código una de las cosas más importantes a definir sería el nombre de variables, métodos y tipos y en cuanto a esto hemos tomado la decisión de utilizar nombres en inglés ya que nos suena mejor por ejemplo como nombre de un método `getOwners` que `obtenPropietarios`. Además con respecto al nombrado, cuando se trate de una clase utilizaremos el siguiente formato con la primera letra en mayúscula y el resto en minúscula, aunque si se trata de un nombre de más de una palabra, la primera letra de la siguiente palabra deber ser en mayúscula también ej: `TwoWords`. Las variables y los métodos se nombrarán en minúscula y si son de más de una palabra, la primera letra de las siguientes palabras será en mayúscula, es decir, siguiendo la convención camel case. ej: `getOwners` o `numeroAleatorio`.

Otro de los estándares que queremos definir es la longitud de los métodos, si un método sobrepasa las 20 líneas, debemos plantearnos su refactorización.

	<div>Proceso de Software y Gestión 2</div>
---	--

### 3.2 Política de mensaje de commit

Los mensajes de commit seguirán el mismo formato que se dé en el product backlog, con el nombre categorizado y sí se quiere una explicación sobre lo realizado en el apartado de descripción en Github. La línea del asunto del commit estará limitada a 50 caracteres y no deberá acabar en punto.

La estructura a seguir será <tipo> [objetivo]

donde tipo puede ser:

- fix: para parchear un error en el código base.
- feat: al introducir una nueva característica.
- build: cambios en el sistema de build, tareas de despliegue o instalación.
- docs: cambios en la documentación.
- refactor: refactorización del código como cambios de nombre de variables o métodos.
- test: añade tests.

Ejemplo de commit: feat: Tarea A2.8.d


Título: feat: Tarea A2.8.d

Descripción: Implementación del borrado de dueños de mascotas, veterinarios y mascotas para el administrador.

### 3.3 Estructura de repositorios y ramas por defecto

Las ramas tendrán de nombre tipo/nº de issue-nombre

donde tipo será uno de los tipos definidos en el apartado anterior, el nº de issue correspondiente y el nombre según el nombre dado en la tarea de GitHub Project. Las ramas creadas no deben borrarse. Se creará una rama develop donde subiremos el código realizado en las ramas de las tareas de ese Sprint y antes de acabar el Sprint todo el contenido estará volcado en la rama master, que es la rama con un estado listo para producción.

	Proceso de Software y Gestión 2
---	---------------------------------

### **3.4 Estrategia de ramificación**

#### **3.4.1 Cómo desarrollar ramas de características**

Para desarrollar características, cada uno creará una rama feature, rama correspondiente a la característica que vaya a desarrollar y realizará el código necesario. Cuando termine hará commit y realizará una pull request para así mezclar el código con la rama develop. Otro integrante del grupo revisará el código comprobando los estándares, además de comprobar que tiene sentido para así aceptar esa pull request y que se mezclen los cambios.


Si llegado a un punto, alguien no sabe cómo seguir con una tarea, puede mostrarle los cambios realizados hasta el momento a un compañero para revisar el código y así orientarlo y darle consejos.

#### **3.4.2 Cómo preparar lanzamientos**

Para preparar lanzamientos, incorporaremos todos los cambios a la rama master, y entre todos revisaremos el código en busca de bugs, mientras probamos de forma informal la aplicación. Para ello, volcaremos los cambios de la rama develop a la rama release para preparar el nuevo lanzamiento en producción, donde se realizarán cambios de última hora, menores correcciones de errores y metadatos para el lanzamiento. Se probará informalmente la aplicación y se desplegará una nueva versión en AppEngine. Una vez finalizado el proceso volcaremos los cambios en la rama master y añadiremos la correspondiente etiqueta, probando nuevamente de forma informal la aplicación, para asegurarnos lo máximo posible de no pasar por alto ningún error.

#### **3.4.3 Cómo corregir errores en producción**

Si se detectan errores en producción se crearán tareas para corregirlos. De cara a corregir estos errores, se crearán nuevas ramas a partir de master que es la que tiene el código de la aplicación en producción y se trabajará para solucionar los bugs sobre estas ramas, las cuales se denominan hotfix. Una vez estén arreglados, desplegaremos una nueva versión en AppEngine, volcaremos los cambios a la rama master y añadiremos la correspondiente etiqueta.

	<div>Proceso de Software y Gestión 2</div>
---	--

### 3.5 Estrategia de versionado

Para el versionado utilizaremos la estrategia del versionado semántico visto en clase, utilizando las variables X.Y.Z, donde X representará los lanzamientos, Y las nuevas características y Z las correcciones de bugs.

### 3.6 Definición de hecho

Una tarea se considerará terminada cuando el responsable de realizarla, haya terminado y otra persona que supervise el resultado de la tarea, considere que está terminada.

## 4. Conclusiones

En conclusión, utilizaremos la estrategia de ramificación GitFlow, junto a la estrategia de versionado semántico y la revisión por pares. Además todos intentaremos producir código siguiendo los mismos estándares.

## 5. Referencias

- Product Backlog de la asignatura Proceso Software y Gestión II, Universidad de Sevilla
- Presentación S2.3 - Methodologies and Best Practices, asignatura Proceso Software y Gestión II, Universidad de Sevilla
- Presentación S2.4 - Applying a Branching Strategy, asignatura Proceso Software y Gestión II, Universidad de Sevilla