Informe de la metodología de gestión de la configuración

decide-single-morancos01
Evolución y Gestión de la Configuración

Control de versiones

Versión	Fecha	Autores	Cambios
v1.0.0	16/11/2023	Todos los miembros	Borrador del documento
v2.0.0	18/12/2023	Todos los miembros	Documento final

Índice

1. Normas de codificación

El objetivo principal de este apartado es poner de manifiesto las diferentes normas de codificación que se van a seguir a lo largo de todo el desarrollo del proyecto, así como en sucesivas versiones.

Para ello, en este apartado se tratarán los siguientes puntos:

- Denominación
- Buenas prácticas

a. Denominación

Los identificadores usan solamente letras y dígitos ASCII y, en un pequeño número de casos, guiones bajos.

Nombres de clases:

Los nombres de las clases se escriben en **UpperCamelCase** y suelen ser sustantivos o frases nominales.

Las clases de prueba tendrán el nombre de Test o TestCase.

Nombres de métodos:

Para el caso de los métodos, estos se escribirán en **lowerCamelCase**, pudiendo ser verbos o frases verbales. Los guiones bajos pueden aparecer en el caso de los nombres de los métodos de prueba.

Nombres de constantes:

En cuanto a las constantes, estas se escribirán en **UPPER_SNAKE_CASE**, es decir, con todas las letras mayúsculas y las palabras separadas por guiones bajos.

Otros nombres y nomenclaturas:

Por otro lado, tanto los campos que no son constantes como los parámetros y las variables se nombrarán utilizando de nuevo la estructura **lower_snake_case.**

2.Política de mensaje de commits

Para la correcta realización de los commits por parte del equipo de desarrollo se darán unas pautas y reglas que indicarán la forma correcta de escribir los mensajes de estos.

Las reglas a cumplir son las siguientes:

- La descripción no podrá superar los 50 caracteres.
- El cuerpo deberá ser de máximo 72 caracteres.
- No concluir la descripción con un punto final pero sí el cuerpo.
- Usar el imperativo en la descripción.
- Se escribirá la descripción completa en minúsculas y el cuerpo empieza por mayúscula.
- En el cuerpo se explicará el qué y el por qué de las modificaciones realizadas en ese commit.
- El tipo será obligatorio indicarlo.

Adicional a estas reglas se indica la estructura que deberán tener estos mensajes:

<tipo>(alcance opcional): <descripción>

[cuerpo opcional]

Se utilizarán estos tipos en los commits:

fix: se indica como tipo cuando se corrigen bugs en el código fuente

feat: se especifica al añadir una nueva funcionalidad

test: se especifica al añadir test de una funcionalidad.

3. Estructura del repositorio y rama por defecto

El objetivo de este apartado y el siguiente es mostrar la estructura del repositorio, cómo vamos a aplicar el sistema organizacional en el que se ha organizado el trabajo por persona.

El repositorio de código que usaremos durante todo el proyecto será **Github** que nos va a permitir administrar el proyecto de forma eficiente, ya que permite el visionado de las diferentes versiones del código y nos aporta distintas herramientas de utilidad: revisión de código, estadísticas, tableros para la aplicación de scrum etc…

Cuando se crea un repositorio en GitHub, se crea una sola rama. Esta primera rama es la rama por defecto que GitHub mostrará cuando alguien acceda al repositorio. En nuestro caso **la rama por defecto será la rama main**. A parte de esta rama se pueden crear muchas más.

En este apartado hablaremos de las ramas principales:

En el proyecto se usarán **dos ramas principales**: Main y Develop:

- **Main**: Los commit realizados en esta rama deben estar listos para subir a producción. Se deberá etiquetar cada confirmación de esta rama con un número de versión.

  Nomenclatura: main

- **Develop**: Servirá como integración para las funciones. Se deberá crear al inicio del proyecto una rama develop (a partir de la rama main) que contendrá todo el historial completo del proyecto.

  Nomenclatura: develop

4. Estrategia de ramificación

En este apartado se hablará de las ramas de soporte:

- Ramas personales
- Bugfix

y de las revisiones por pares.

a. Ramas personales

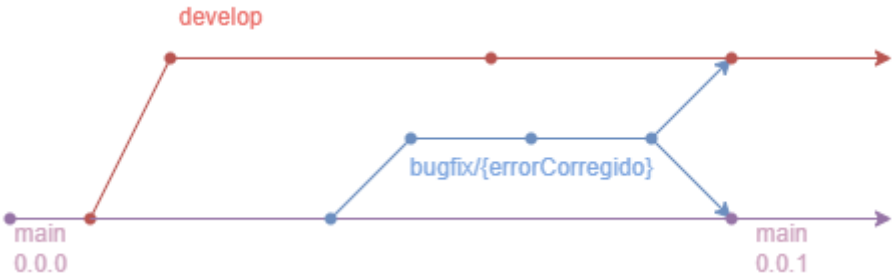
Para implementar funcionalidades se debe abrir una rama personal desde la última versión de develop. Cuando la nueva función esté terminada, se fusionará en develop pero no deben interactuar directamente con main.

Nomenclatura: uvus/

Ejemplo: manvazmar3

b. BugFix

Esta rama se usará para corregir errores en el código en producción. Esta rama no se planifica, se crea si se detecta algún bug en producción, por tanto se crean a partir de la rama main y los cambios deben fusionarse (merge) con main y develop.



Nomenclatura: bugfix/{errorCorregido}

NOTA: Tras un bug corregido se aumentará el parche de la rama main, así como el de la TAG.

c. Revisiones

Las revisiones de las pull-request se realizarán mediante un miembro distinto al que haya subido al pull-request.

- Todos los miembros del equipo deben revisar tareas de otros compañeros.
- Para iniciar una revisión habrá que crear un **PULL-REQUEST**
- Al revisar código, hay tres acciones disponibles:
  - Aceptado
  - Solicitar cambio
  - Comentar

5.Política de versiones

En este apartado se discutirá sobre las políticas de versionados que emplearemos a lo largo de la asignatura.

Hemos decidido adoptar la base del versionado semántico para construir nuestra metodología.

De acuerdo a la especificación se cumplirán estas normas:

- Un número de versión normal debe tener la forma siguiente X.Y.Z donde la X, Y y Z son números enteros no negativos y no deben ir acompañados de un cero por delante. Good 1.11.0 Bad 01.2.1 -0.1.5.

- Cuando un paquete con su número de versión se haya publicado, los contenidos de su versión no pueden ser modificados bajo ningún concepto. Los cambios introducidos se deben reflejar como una nueva versión del paquete.

- La versión de parche Z (x.y.Z) sólamente se puede modificar en el caso de que se introduzcan correcciones de errores que no rompan la funcionalidad implementada. Se considera un bug aquellos errores internos que introducen comportamientos inesperados en la aplicación.

- La versión menor Y (x.Y.z) debe incrementarse si se introduce nueva funcionalidad compatible en la aplicación, es decir, que no rompa la funcionalidad anterior. Se debe incrementar el número de versión menor si alguna funcionalidad se marca como en desuso. Se debe incrementar el número si se introducen mejoras o cambios sustanciales en la aplicación. La versión de parche de reiniciarse a 0 cuando se incremente el número de versión menor.

- La versión mayor X (X.y.z) debe incrementarse si se introducen cambios que rompen la compatibilidad hacia atrás de la aplicación. Cuando se incremente este número la versión menor y la de parche debe reiniciarse a 0.

- El orden de las versiones debe calcularse mirando el valor numérico correspondiente a la versión mayor, menor y de parche en ese orden. La versión mayor tiene más peso que la menor y de parche, y la menor tiene más peso que la versión de parche. Por ejemplo 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1.