Informe de la metodología de gestión de la configuración

psg2-2223-g7-72

Proceso Software y Gestión II,
Universidad de Sevilla.

Pedro González Marcos
Rocío López Moyano
José Manuel Moreno Guerrero
Victoria de Carmen Ruíz Delgado
Manuel Vázquez Martín
David Zarandieta Ortiz

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:

- Formateo
- Denominación
- Buenas prácticas

a. Formateo

Empleo de llaves:

Las llaves {} se utilizarán en las declaraciones **if, else, for**, incluso cuando el cuerpo esté vacío o contenga una sola declaración.

Para las llaves hay que tener en cuenta lo siguiente:

- Ningún salto de línea antes de la llave de apertura.
- Salto de línea después de la llave de apertura
- Salto de línea antes de la llave de cierre.
- Salto de línea después de la llave de cierre solo si esa llave termina una declaración o el cuerpo de un método, constructor o clase con nombre.

Indentación de bloques y estructuras:

Para la indentación de los bloques, en lugar de espacios se utilizará la tabulación para agilizar el proceso y el formateo de estas estructuras.

Número de declaraciones:

Una declaración por línea.

```
int a;
int b;
```

En lugar de:

```
int a,b;
```

Excepción: Se podrán utilizar múltiples declaraciones de variables en el encabezado de un **for** bucle.

Anotaciones:

Las anotaciones aparecerán situadas en la cabecera, independientemente si se trata de una clase o un método.

```
@Service
public class OwnerService {

  private OwnerRepository ownerRepository;

  @Autowired
  private UserService userService;

  @Autowired
  private AuthoritiesService authoritiesService;

  @Autowired
  public OwnerService(OwnerRepository ownerRepository) {
    this.ownerRepository = ownerRepository;
  }
}
```

Comentarios:

Los comentarios se posicionarán al mismo nivel que el código circundante utilizando el tabulador. Podrán utilizarse tanto comentarios del estilo **/*...*/** como el estilo **//.** Para los comentarios de varias líneas, **/*...*/,** las siguientes líneas deberán comenzar alineadas con la línea anterior. En cualquier caso, evitar el uso de comentarios en la medida de lo posible, minimizando así la cantidad de estos elementos a lo largo del código.

b. Denominación

Los identificadores usan solamente letras y dígitos ASCII y, en un pequeño número de casos, guiones bajos. El nombre del archivo de origen consiste en el nombre que distingue entre mayúsculas y minúsculas de la clase de nivel superior que contiene, más la .java extensión.

Nombres de paquetes:

Los nombres de los paquetes usan solo letras minúsculas y dígitos (sin guiones bajos). Las palabras consecutivas simplemente se concatenan juntas.

Nombres de clases:

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

Las clases de prueba tendrán un nombre que terminen con Test, por ejemplo: IntegrationTest. Si el test está relacionado con una sola clase entonces recibirá el nombre de esa clase seguido de Test.

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 **lowerCamelCase.**

c. Buenas prácticas

En este apartado se recomiendan una serie de buenas prácticas a seguir para hacer un código más legible y eficaz. De manera que se tienen prácticas tales como:

- Escribir la menor cantidad de líneas posible, estableciéndose así un límite de ochenta caracteres por línea.
- Segmentar bloques de código para facilitar su legibilidad.

- Utilizar la tabulación y el espaciado para marcar el comienzo y el final de las estructuras de control, especificando claramente el código entre ellos.
- **NO** utilizar funciones excesivamente largas.
- Utilizar el principio **DRY (Don't Repeat Yourself)**, evitando así tareas repetitivas.
- Evitar el anidamiento profundo, ya que demasiados niveles de anidamiento dificultan la legibilidad del código.
- Evitar las largas colas, es decir, evitar bloques de líneas horizontalmente cortos y verticalmente largos.
- Utilizar nombres que indiquen el propósito de la tarea que realiza cada método, evitando nombres excesivamente cortos así como nombres que no estén relacionados con el desempeño que realiza el código.

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:

- En caso de que se incluya cuerpo o pie de página, ambos obligatoriamente se escribirá con un salto de línea tras lo que lo preceda.
- El asunto 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 empezará 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]

[pie(s) de página opcional]

Se utilizarán estos tipos en los commits:

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

feat: se especifica al añadir una nueva funcionalidad

Otros tipos son también: **build: chore: ci: docs: style: refactor: perf: test:**

Se añadirá **!** después del tipo si el commit tiene algún cambio de última hora o importante.

En este último caso se podrá indicar en el pie de página escribiendo **"BREAKING CHANGE:"** y seguido de una descripción.

También pueden incluir mensajes como **"Fixes #issue"**, **"Refs #issue"**.

Algunos ejemplos de mensajes de commit serían:

Mensaje de commit con descripción y breaking change en su pie de página

feat: añadida la posibilidad de comprar productos para mascotas

BREAKING CHANGE: la clave 'compra' del archivo config ahora asigna las compras al cliente correspondiente

Mensaje de commit con alcance y !

feat(api) ! : añadido que se envíe un correo al cliente que solicite una visita

Mensaje de commit con cuerpo y pie de página

fix: solucionado el solapamiento de solicitudes

Se introducirá una ID de solicitud y una referencia de la última solicitud para solucionar los tiempos de espera en las solicitudes.

Fixes: #45
Refs: #123

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 de ramificación basado en git-flow a nuestro proyecto y la nomenclatura a seguir para cada una de las ramas .

El repositorio de código que usaremos durante todo el proyecto será **Git Hub** 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, basada en Git Flow y revisiones por pares

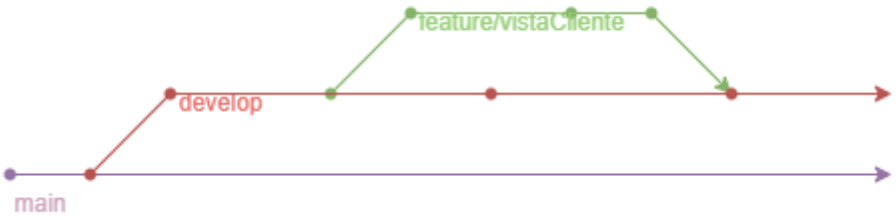
En este apartado se hablará de las ramas de soporte:

- Feature
- Release
- Bugfix

y de las revisiones por pares.

a. Features

Para implementar nuevas funcionalidades se debe abrir una nueva rama feature desde la última versión de develop. Cuando la nueva función esté terminada, se fusionará en develop pero no deben interactuar directamente con main.Desde la rama develop se hará merge a la rama feature, será necesario realizar un **PULL-REQUEST** antes del merge. Además se usará la opción **--no-ff** del merge para no eliminar la rama feature.



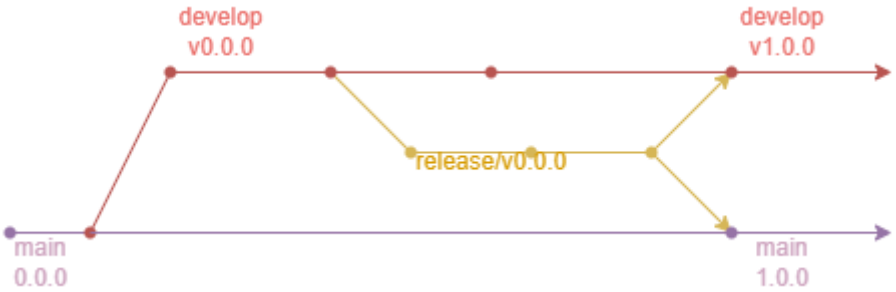
Nomenclatura: feature/{numeroDeIssue}-{nuevaFuncionalidad}

Ejemplo: feature/vistaCliente

Comando para hacer merge (antes hay que posicionarse en la rama develop): git merge --no-ff feature/{nuevaFuncionalidad}

b. Releases

La rama release se utilizará para preparar el siguiente código listo para producción, en ella se harán los últimos ajustes (archivos de configuraciones, archivos de librerias, pequeñas correcciones) antes de incorporar el código a la rama main. Además los cambios también se incorporarán a la rama develop. Al incorporarlo a main se hace una etiqueta con un versionado



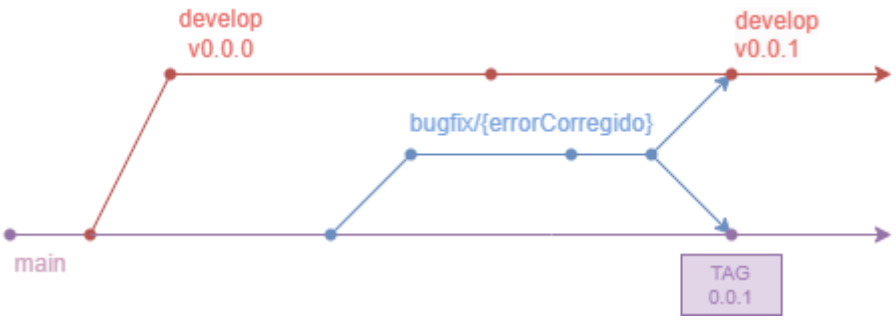
. **Nomenclatura**: release/v{versiónDeRelease}

NOTA: La versión de release se corresponde con la versión de la rama develop en el momento que se bifurcó la rama

Ejemplo: release/v0.0.0 (Esta rama parte de la rama develop v0.0.0. Una vez se realiza el merge con la rama develop, la rama develop pasa a ser v1.0.0 La rama main también aumenta su versión al hacer merge con ella y además se crea una TAG)

c. 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 develop y la rama main, así como el de la TAG

d. Revisiones por pares

Las revisiones por pares mejoran considerablemente la calidad del código, por tanto será implementada a lo largo de todo el proyecto.

- Todo el código que se fusione a otra rama debe ser revisado y aprobado por al menos otro miembro del grupo. Además también usaremos esta técnica para la revisión de tareas colocadas en la columna "In review" en el tablero del Sprint Backlog.
- 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

En los siguientes enlaces se aporta información de como realizar PULL-REQUEST y sus acciones disponibles.

Información sobre cómo realizar el PULL-REQUEST.

Información sobre cómo realizar la revisión.

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.

- Como convención adicional a cada lanzamiento se incluirá un alias que identificará a la versión publicada. Se utilizarán nombres de mascota para nombrar a las releases.

Anexo I: Nuestro equipo

Pedro González Marcos

Rocío López Moyano

José Manuel Moreno Guerrero

Victoria de Carmen Ruíz Delgado

Manuel Vázquez Martín

David Zarandieta Ortiz

Anexo II: Bibliografía

- A. (s. f.). *Estándar de codificación Java - XWiki*. https://amap.cantabria.es/amap/bin/view/AMAP/CodificacionJava
- Acerca de las bifurcaciones. (s. f.). GitHub Docs. https://docs.github.com/es/pull-requests/collaborating-with-pull-requests/working-with-forks/about-forks
- Acerca de las solicitudes de incorporación de cambios. (s. f.). GitHub Docs. https://docs.github.com/es/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests
- Bose, S. (2021, 2 febrero). *Coding Standards and Best Practices To Follow*. BrowserStack. https://www.browserstack.com/guide/coding-standards-best-practices
- Cambiar la rama predeterminada. (s. f.). GitHub Docs. https://docs.github.com/es/repositories/configuring-branches-and-merges-in-your-repository/managing-branches-in-your-repository/changing-the-default-branch
- *Google Java Style Guide*. (s. f.). https://google.github.io/styleguide/javaguide.html