

Universidad de Sevilla

Escuela Técnica Superior de Ingeniería Informática

Sprint 2 – Reporte Técnico

Metodología de Gestión de la Configuración




Grado en Ingeniería Informática – Ingeniería del Software

Proceso de Software y Gestión II

Curso 2022 – 2023

Fecha	Versión
02/03/2023	V1.1

Grupo de prácticas: G3 - 31		
Autores por orden alfabético	Correo	Rol
Campano Galán, Alejandro	Alecamgal1@alum.us.es	Scrum Master
Cuenca Pérez, Pablo	Pabcueper@alum.us.es	Desarrollador
Gómez Romero, Guillermo	Guiqomrom@alum.us.es	Desarrollador
López-Benjumea Novella, Alberto Miguel	alblopnov@alum.us.es	Desarrollador
Naredo Bernardos, Ignacio	ignnarber@alum.us.es	Desarrollador
Ortiz Blanco, Manuel	manortbla@alum.us.es	Desarrollador

 <p>UNIVERSIDAD D SEVILLA</p> <p>Proceso Software y Gestión II</p>	<p>Proceso de Software y Gestión II</p> <p>Reporte Técnico - Metodología de Gestión de la Configuración</p> <p>Control de Versiones</p>
---	--

Control de Versiones

Fecha	Versión	Descripción
22/02/2023	V1.0	Desarrollo del documento completo.
02/02/2023	V1.1	Revisión del documento y versión final

Índice de contenido

1. Introducción.....	2
2. Contenido.....	2
2.1. Estándares de código	2
2.2. Políticas de mensajes de commit.....	3
2.3. Estructura del repositorio.....	4
2.4. Estrategia de ramas.....	4
2.4.1. Desarrollar ramas “feature”	5
2.4.2. Preparar “releases”	6
2.4.3. Arreglar bugs en producción	6
2.5. Políticas de versionado.....	6
3. Conclusiones	7
4. Referencias.....	8

1. Introducción

En este reporte técnico, comentaremos las metodologías seguidas en nuestro proyecto para cumplir con la estrategia que acordamos durante el desarrollo del proceso software. Es estrictamente necesario cumplir todas las buenas prácticas definidas a continuación y así lograr un producto final acorde al trabajo que se nos solicita.

2. Contenido

A continuación, iremos describiendo las distintas metodologías seguidas para cada uno de los apartados.

2.1. Estándares de código

En primer lugar, en cuanto al código implementado, tratamos de cumplir unos estándares generales que siguen el lenguaje Java con el objetivo de conseguir mejorar la legibilidad de código, comprensión, mantenibilidad y colaboración.

En cuanto al nombrado en Java, hemos seguido las siguientes pautas:

- Hemos tratado de programar siempre en Inglés.
- Los paquetes siempre van en minúscula
- Las distintas clases las llamaremos con la primera letra de cada palabra en mayúscula (CamelCase) e intentaremos que estos nombres estén compuestos de palabras completas y descriptivas.
- Los distintos métodos deberán ser verbos en infinitivo. Estos se escribirán en minúscula la primera letra del nombre y la primera letra de cada palabra interna en mayúscula (lowerCamelCase)
- Para las variables, trataremos de seguir el mismo tratamiento de nombramiento que para los métodos (lowerCamelCase)
- Para las constantes, las escribiremos en mayúsculas y separadas por “_”, estas siempre las declararemos como *public static final*.

Por otro lado, trataremos de realizar buenas prácticas como pueden ser escribir una sola sentencia por línea, tratar de acceder/modificar a las propiedades de una clase mediante get/set, evitar duplicar nombres de variables (a no ser que se declaren varias veces, pero sean exactamente lo mismo) y tratar de que el código esté correctamente formateado, ayudándonos del IDE.

Finalmente, cabe destacar que trataremos de reducir la deuda técnica todo lo que nos sea posible. Para ello, trataremos de evitar soluciones ineficientes que, pese a que funcionen, nos lleve a futuros quebraderos de cabeza mediante distintas refactorizaciones del código. Con el objetivo de cumplir esto, realizaremos distintas pruebas, trabajaremos todos juntos e intentaremos no retrasar la refactorización.

2.2. Políticas de mensajes de commit.

A la hora de realizar un commit, hemos tomado una serie de pautas para describir correctamente el contenido de los archivos modificados.

En primer lugar, hemos decidido escribir el título que no debe sobrepasar los 50 caracteres y que siempre comienza con un identificador de nuestra tarea. Por ejemplo, “A2.8.a” que describiría la funcionalidad del Pet Hotel como marca en el Product Backlog. A continuación, en mayúsculas, describiríamos la acción que hemos realizado en imperativo y, sin terminar en punto, finalizamos el título con la expresión “Fixes #N” donde la N marca el número de issue al que estamos dando solución, si es que estamos cerrando un issue.

En segundo lugar, describiremos el cuerpo del mensaje que lo separaremos del título con una línea en blanco y, comenzaremos a describir muy brevemente qué hemos hecho y cómo, sin pasar de los 72 caracteres.

Entonces, un mensaje de ejemplo nos quedaría tal que así, tomando el issue N12 que nos marca la tarea realizada:

A2.8.d DELETE SUPPORT DONE Fixes #12

Delete support to Vets, Owners, Pets and Visits done with views.

2.3. Estructura del repositorio

Para nuestro repositorio, hemos usado Git, el DVCS (Sistema de control de versiones distribuido) más popular de la industria. Más concretamente, hemos utilizado GitHub, ya que nos permite realizar todas las buenas prácticas definidas en este documento con facilidad gracias a su interfaz gráfica de usuario con la que podremos hacer uso de distintas herramientas como pueden ser:

- La creación de issues para nuestras distintas actividades del Product Backlog
- Realizar las reviews del trabajo hecho entre varios compañeros con los *Pull Requests*.
- Distintos diagramas de información del repositorio.
- Realizar nuestro Scrum Board, mediante el apartado de Project, donde iremos reflejando todas nuestras actividades en las columnas *To do*, *In progress*, *In review*, *Done* con el objetivo de mantener la comunicación constante entre nosotros.

A la hora de programar, gracias a este software, podemos tener cada uno de nosotros una copia local de nuestro repositorio, en la que podremos trabajar cada uno en nuestras tareas. Así, logramos mantener líneas independientes de desarrollo sin ningún tipo de conflicto, independiente de la red y teniendo un versionado local.

2.4. Estrategia de ramas

Nuestra estrategia se basará en la metodología de GitFlow junto con el Peer Review, con esto tendremos 2 ramas principales en las que no realizaremos ningún tipo de commit que son las ramas *develop* y *main* y 3 tipos de ramas distintas que son las ramas de *feature*, *release* y *hotfix* las que iremos desarrollando posteriormente.

En primer lugar, nos encontramos con nuestra rama *main*. Esta rama, la tendremos en estado de producción y estará lista para producir nuestras versiones de la aplicación.

Por el otro lado, tenemos la rama *develop* que es la rama en la que iremos introduciendo los nuevos cambios que vamos desarrollando.


Para introducir los cambios en estas dos ramas, realizamos un *merge*, para introducir los cambios que se encontrarán en las ramas que se explicarán a continuación. Para ello, cuando terminemos de realizar nuevos cambios y queramos introducirlos en nuestra rama *develop* o cuando queramos producir una nueva versión en la rama *main*, debemos hacer uso de esta herramienta de *merge*.

A la hora de hacer un *merge*, siempre trataremos de hacer una *Pull Request*, solicitando los cambios y haciendo uso de la metodología de Peer Review. Es decir, otra persona que no haya sido la que ha realizado el código, deberá revisar el nuevo código que se va a introducir, comprobando que todo funciona correctamente y que se cumple con todo lo marcado en este documento. Para asegurarnos de esto, directamente hemos configurado GitHub y, así, no nos dejará cerrar la pull request sin, al menos, 1 review positiva de otro compañero.

A continuación, hablaremos de las distintas ramas que crearemos e iremos desarrollando. Estas son las ramas *feature*, *release* y *hotfix*. Cabe recordar que, siempre que comencemos a trabajar con cualquiera de estas ramas, debemos tomar tiempo de lo que tardamos en desarrollarla con la herramienta clockify y, en nuestro Scrum Board con la pestaña de *projects* en GitHub, iremos cambiando la tarea a las columnas *In Progress* o *In Review*.

2.4.1. Desarrollar ramas “feature”

A la hora de desarrollar una nueva característica de la aplicación, tenemos que crear una nueva rama *feature* a partir de la rama **develop**. En nuestro caso, llamaremos a la rama *feature/N-descripción*. Siendo N el número de *issue* y descripción, unas pocas palabras contando la funcionalidad. Una vez creada, realizaremos la tarea y comenzaremos con el proceso de merge, mencionado anteriormente, para así, unir la rama correctamente a la rama *develop*. Cabe destacar que, cuando terminemos la funcionalidad, la rama nos quedará inutilizada, pero es importante no borrarla por indicaciones de nuestro cliente.

 UNIVERSIDAD DE SEVILLA Proceso Software y Gestión II	Proceso de Software y Gestión II Reporte Técnico - Metodología de Gestión de la Configuración
--	--

2.4.2. Preparar “releases”

Las ramas de *releases*, las crearemos cuando queramos preparar una versión para su lanzamiento a producción. En esta rama, trataremos de corregir todos los problemas que hayan podido surgir a la hora de hacer merge con todas las *features*, corregir otros problemas y, preparar los metadatos necesarios para poder unir la rama con nuestra rama *main*. Diremos que en esta rama se hace la puesta a punto de la aplicación para, posteriormente, pasar nuestra nueva versión a producción.

Estas ramas las crearemos con el nombre *release/x.y.z*, en la que *x.y.z* son 3 números que marcarán la versión que estamos lanzando. Posteriormente, se define la política de versionado que hemos seguido.

2.4.3. Arreglar bugs en producción

Las ramas *hotfixs*, serán utilizadas para arreglar distintos errores en una versión que ya está en producción. Para ello, cuando detectemos un error, crearemos una rama con el nombre de *hotfix/x.y.z*, siendo *x.y.z* la versión marcada por la política de versionado seguida y arreglaremos los errores en esta rama. Una vez arreglados, uniremos esta rama a la rama *main*, lanzando una nueva versión a producción de nuestra aplicación.

2.5. Políticas de versionado.

Las distintas versiones de nuestra aplicación, las indicaremos con el versionado semántico. Este versionado se basa en la estructura *x.y.z* donde cada una de estas letras representa a un número.

En primer lugar, la letra *x*, representa a la versión principal que la cambiaremos cuando se realizan cambios mayores que pueden romper la compatibilidad del proyecto con versiones anteriores.

En segundo lugar, la letra *y*, se cambia cuando se producen pequeños cambios con nuevas funcionalidades pero que no rompen la compatibilidad de nuestra aplicación.

Finalmente, la letra z, indican los distintos parches que realizaremos arreglando fallos sin romper la compatibilidad de la aplicación.

Entonces, comenzamos el desarrollo con la versión 0.0.0 y la vamos elevando, la primera versión pública que desplegaremos será la versión 1.0.0 y, continuaremos aumentando versiones con las distintas implementaciones futuras.

Este versionado lo reflejaremos en nuestro proyecto con las herramientas que nos da Git que son los tags, con las que guardamos el estado de la rama que queramos en el momento que desplegamos la versión.

3. Conclusiones

En conclusión, debemos de ser bastante estrictos en seguir toda esta metodología de configuración descrita en este documento para asegurarnos una versión estable y correcta de nuestra aplicación. Así, lograremos nuestros objetivos de acorde al trabajo solicitado.

4. Referencias

Estándar de codificación Java -

<https://amap.cantabria.es/amap/bin/view/AMAP/CodificacionJava>