Informe académico final

Obligatorio

Ingeniería de Software Ágil 2

Juan Martín Rodríguez 230371

Francisco Aguilar 230143

German Cotelo 212350

Gestión del proyecto

Primera entrega

Lo primero que realizamos fue un análisis de deuda técnica, para esto se “dividió” la deuda técnica en tres categorías, “Análisis de código”, “Bugs” y “Missing Requirements”. Junto con el análisis de deuda técnica realizamos tres guías denominadas “Definición del proceso de ingeniería en el contexto de Kanban”, “Explicación del tablero y su vínculo con el proceso de ingeniería” y finalmente “Creación y posterior mantenimiento del repositorio: elementos que contiene y cómo los van a versionar”. El registro y trazabilidad de estos ítems se llevó a cabo utilizando GitHub Issues.
Se creó también la primera versión del tablero Kanban, esta primera versión contaba únicamente con tres columnas, “To Do”, “Doing” y “Done”, en estas columnas ubicamos las tareas que estábamos realizando así como su estado actual, moviéndolas de izquierda a derecha, como exige la metodología Kanban.
La categoría “Análisis de código” hace referencia a las malas prácticas llevadas a cabo por el equipo de desarrollo previo, estos issues se clasificaban respecto a su seriedad. Dentro de esta categoría algunos ejemplos encontrados fueron: Métodos que recibían 6 parámetros, Endpoints que no retornaban códigos de error correctos. Mientras que en la categoría “Bugs” colocamos todos los errores que descubrimos en la aplicación utilizando testing exploratorio, los bugs los clasificamos respecto a su prioridad y severidad. Algunos de los errores que encontramos fueron: El tracker agregaba pedidos trackeados cuando ingresabas un código inexistente, así como también validaciones que no se realizaban y tiraba un error 500. Y finalmente en la categoría “Missing Requirements” colocamos solamente uno que encontramos que fue el botón de login sin implementar por lo que había que dirigirse a la página “/login” manualmente.
Para esta instancia se registró el esfuerzo horas-persona que se requirió encontrar estos defectos así como también para elaborar los documentos, estas métricas fueron analizadas en profundidad en entregas posteriores.
De la misma manera se realizó una retrospectiva, utilizando el método “DAKI”, junto con el “Scrum Master”, que no fue tal, ya que estábamos utilizando Kanban, por lo que lo denominamos “Kanban Master”.

Segunda entrega

Continuando con la segunda entrega nuestro equipo procedió a seleccionar los tres bugs de mayor prioridad para realizar su corrección utilizando TDD, así como también implementar la primera versión del pipeline.
Al contar con tests únicamente en el back-end se realizó TDD únicamente en el back-end, los bugs seleccionados por nuestro equipo fueron “Visibilidad de algunos recursos para usuarios anónimos”, “El campo cantidad para crear una droga permite valores erróneos” y “Trackear un código de trackeo inexistente arroja un error”. Luego de corregir estas funcionalidades utilizando TDD procedimos a crear nuestra primera versión del deployment pipeline, donde se verifica que el back-end .NET cree una build correctamente y que todos los tests unitarios corran correctamente.

Durante el transcurso de esta segunda entrega se siguen tomando métricas de esfuerzo y registrando, para luego ser analizadas.
Actualizamos los documentos presentados en la entrega anterior y creamos uno nuevo denominado “Configuración del pipeline y su vínculo con el tablero”.
Se realizó de igual manera una videollamada con nuestro PO donde se verifica que los bugs han sido debidamente arreglados.
Para finalizar realizamos una retrospectiva con nuestro Kanban Master siguiendo el formato DAKI, donde discutimos los altos y bajos del equipo.

Tercera entrega

Para la tercera entrega se replanteo la definición del proceso de ingeniería de la segunda entrega para lograr un mejor manejo de las necesidades de la tercera entrega. De esta manera, se pasó de una proceso de ingeniería orientado a TDD a uno orientado a BDD, logrando así una metodología que permita generar escenarios de prueba automatizados para cada una de las user stories definidas por el equipo.

Con esto en mente el equipo creó un tablero KANBAN donde definió dos flujos, uno basado en BDD para las User Story que el equipo previamente definió a partir de los requerimientos a desarrollar y otro más sencillo para otro tipo de tareas.

A continuación el equipo implementó las User Story siguiendo la metodología especificada. Se crearon los escenarios de prueba Specflow y se desarrolló el código front-end y back-end. Además se agregaron los casos de prueba automatizados al pipeline.

El equipo tuvo una instancia de review con el product owner donde se evaluó las US implementadas según los criterios de aceptación explicitados previo al desarrollo. En esta se noto que una de las US no estaba completa y se terminó en la entrega siguiente.

Finalmente el equipo tuvo una retrospectiva mediante la plataforma metro-retro utilizando la metodología DAKI en donde el equipo sacó importantes conclusiones para mejorar en la siguiente entrega.

Cuarta entrega

Para la cuarta entrega se hizo una recapitulación de las anteriores de manera de definir un proceso de ingeniería unificado (y tablero KANBAN correspondiente) que permita cumplir con tres flujos diferentes. Uno basado en TDD para el arreglo de bugs, en este es importante notar que se condensó el flujo de TDD completo (Red, Green, Refactor) en una sola columna Development(TDD) ya que este es un proceso cíclico. Un segundo flujo, basado en BDD, es para las US. En este caso es importante aclarar que se eliminó la columna To merge ya que, como parte del desarrollo trunk based, los merges se hacían constantemente. Se unificó una columna integration testing la cual hace referencia al testing de integración que se hizo tanto para bugs como para US. Finalmente, el tercer flujo es el correspondiente a las tareas y es el mismo que el de las entregas anteriores.

A continuación se realizaron las pruebas automáticas de integración con la herramienta Selenium.

Finalmente se realizó una review con el PO owner donde se analizaron todos los requerimientos de las entregas anteriores. También se realizó una retrospective mediante la plataforma metro-retro utilizando la metodología DAKI en donde el equipo sacó importantes conclusiones.

En esta cuarta entrega también se hizo un análisis de las métricas que el equipo recabó durante todo el proyecto, estas están analizadas en detalle en el documento respectivo de la entrega 4.

El equipo tuvo un Lead time promedio, para bugs de 2 y para User Storys de 8.8. Un Cycle Time promedio para bugs de 1 y para User Sorys de 3,6. Esto nos da un Flow efficiency (medido como Cycle time / Lead time) de 0,5 para bugs y 0,6 para US. Por encima de la marca de 40%, generalmente considerado bueno en la industria. Además se registró un throughput promedio de 2.6

Reflexiones sobre los objetivos de la rúbrica y lecciones aprendidas

Aplicar un marco de gestión ágil

Lo primero que se tuvo que hacer fue organizar el proyecto aplicando un marco de gestión ágil. Esto quiere decir que se tuvieron que definir los roles de cada integrante, las ceremonias, los artefactos y cómo se iba a proceder en cada entrega.
Al final se respetó durante todo el proyecto una estructura similar, empezando cada entrega con una reunión en la que se definían las tareas, se asignaban a un miembro o dos del equipo y se asignaban roles y responsabilidades. Si se necesitaba se hacía otra reunión para reorganizar prioridades. Y siempre se hacía una retrospectiva al final de cada entrega.
La ingeniería de software ágil suele tener dailies, en nuestro caso no hicimos una reunión diaria, ya que no era fácil coordinar entre todos un horario. Sin embargo siempre se mantuvo la conversación a través de un chat del equipo lo que ayudó mucho a la hora de organizarse.
Sin embargo, esta ausencia de una reunión diaria puede que haya invitado al descuido que llevó al equipo a no entregar todos los requerimientos en la tercera entrega, ya que con la reunión puede que los tiempos de trabajo diario se respetaran más. En el futuro se debería por lo menos probar esta idea sin olvidar que no siempre es posible juntarse, pero que el esfuerzo puede valer la pena, sobre todo en una entrega más ‘pesada’ como la tercera.

Analizar la deuda técnica

La deuda técnica es el costo del retrabajo adicional causado por la elección de la solución más rápida en lugar de la más efectiva. Durante la primera entrega se tuvo que analizar

este costo y ver qué partes del proyecto que se nos fue entregado debían ser mejoradas, arregladas o terminadas.
Para hacer este análisis se le dejó a cada integrante del equipo diferentes partes del código, ya que este era mucho, y para la página, cada integrante probó lo que pudo y una vez que encontraba un error creaba el issue si aún no había sido creado.
Esto funcionó como solución rápida, pero la realidad es que este mismo método de investigar todo por separado y luego comparar debería haber sido aplicado para el código también. Capaz que hasta combinar ambas ideas y separar las áreas a analizar y luego turnarse el área que se mira, de esta forma nos aseguramos que lo que uno pueda haber pasado por alto, otro lo note y lo reporte.
Para la separación de código, el código del frontend lo analizó el desarrollador con más experiencia en el lenguaje, lo cuál fue una buena idea, pero capaz que podría haber sido mejor que lo vieran todos juntos o de a pares. De esta forma los desarrolladores menos familiarizados con el lenguaje podrían aprender y estar más listos para las siguientes entregas. Además muchas veces, al revisar código con alguien al que se lo estamos explicando, podemos notar detalles que hubiéramos pasado por alto solos.

Implementar un repositorio y procedimientos de versionado

Una de las partes más importantes de todo este proyecto fue el manejo del repositorio en GitHub y la documentación que se guardó en el mismo.
Dentro del repositorio se guardaron los documentos versionados, separados en carpetas por entrega.
Esto ayuda a futuro a entender bien cómo se manejó cada parte del proyecto y cuáles fueron las necesidades de cada una de las entregas. Si se pensara volver a trabajar sobre el proyecto en el futuro esta información podría ser útil. Y también es documentación útil para futuros proyectos, ya que todos los aprendizajes y mejoras que se hicieron en estos documentos y sus versiones, muy probablemente se puedan aplicar en otros proyectos.
También, en GitHub, donde se guardó el repositorio, se guardó el tablero y las tareas con las que se trabajó en las mismas. Hay un tablero por entrega, lo que permite a futuro ver bien que se trabajó en cada parte y facilita el análisis de las métricas. Tuvimos un problema al principio con el lenguaje de los nombres de los tickets en el tablero pero después de la primera entrega esto se solucionó.
El procedimiento de versionado se debería haber definido mejor desde el principio ya que se nota al mirar los nombres de los documentos, que no se mantuvo un estándar durante el transcurso del proyecto. Sin embargo, este menor error no evita que se entienda cual es la versión de cada documento ya que se mantuvo una buena definición para los nombres de las carpetas luego de la primera entrega.

Crear un pipeline con eventos y acciones

En GitHub existen acciones que se pueden aplicar para el pipeline, que ejecutan pruebas sobre el código y aseguran la calidad del mismo. No se implementaron muchas y en algunos momentos se ignoraron sus advertencias ya que no estaban correctamente aplicadas. Se pusieron chequeos de las pruebas cuando se quiere hacer un merge a main y se sacó la posibilidad de hacer un commit a main. Estas medidas aseguran cierto nivel de

calidad y aseguran el flujo de las ramas a main para no hacer deploy de nada que no haya sido revisado.
Está claro que a futuro se debería trabajar mejor con estas herramientas ya que traerían más seguridad, tranquilidad y velocidad de feedback al pipeline y al proceso de ingeniería. Es más, esto es lo que se concluyó en una de las retrospectivas.

Integrar prácticas de QA en el pipeline y gestionar el feedback

Pruebas y revisión con el product owner, github actions
Para poder asegurar la calidad se pusieron varias instancias de pruebas en todos los tableros de las entregas en las que se hicieron cambios de código. Tanto pruebas automáticas (dentro de las cuales tanto se ejecutaron como se crearon donde fuera apropiado), como pruebas manuales. También se utilizaron un poco las opciones de GitHub actions para mantener QA. Lamentablemente estos no se usaron mucho y una de las cosas que reflexionó el equipo durante la última retrospectiva fue que sería una buena idea a futuro familiarizarse mejor con esta herramienta ya que puede aportar mucho valor.
Lo que sí se pudo aplicar bien fue que se hiciera el merge regularmente a main para mantener a la rama principal siempre con los últimos cambios y poder recibir feedback más rápido. De todas formas este flujo hubiera sido más valioso implementando correctamente las acciones de GitHub ya que no hubiera permitido confirmar con más seguridad que los cambios que se estaban subiendo funcionaban correctamente.
Otra forma de feedback que se utilizó fué la reunión con el product owner. Estas eran más una simulación que una verdadera reunión pero fueron útiles ya que nos permitieron mostrar nuestro trabajo y nos hicieron asegurarnos de que funcionara todo correctamente.En la última entrega no se pensaba hacer una muestra con el product owner, ya que no se iban a agregar funcionalidades al producto, sin embargo se tuvo que terminar una de las funciones después de la tercera entrega, lo que nos obligó a hacerla y mostrar el arreglo.
Como era una reunión simulada se pudo hacer de forma rápida y desordenada, en una situación real sería mejor tener preparado de antemano el material que se va a mostrar, a diferencia de como se hizo para la tercera entrega.

Generar escenarios de testing desde la perspectiva del usuario

Los escenarios de testing desde la perspectiva del usuario suelen ser pruebas con el frontend y durante el transcurso de este proyecto se probó el frontend en todas las entregas. Durante la primera fue para probar las funcionalidades y ver cuales funcionaban correctamente. Durante la segunda fue para validar que los errores que se habían corregido estaban bien corregidos. Y durante la tercera fue donde se generaron los criterios de aceptación, que en este caso eran pruebas desde la perspectiva del usuario, que permitían validar que las funcionalidades nuevas estuvieran bien implementadas.
Estas pruebas además se automatizaron en la cuarta entrega con la ayuda de Selenium. Lo que permitió que en el futuro se puedan validar fácilmente y rápidamente estas funcionalidades. Además, en el futuro se deberían agregar casos de prueba nuevos por cada función nueva que se implemente, de esta forma en el futuro todo se va a poder validar más rápido.

Algo que ocurrió que se podría mejorar es intentar separar mejor estas pruebas automáticas, ya que para poder mantenerlas independientes unas de otras, sin depender de los datos de base, se combinaron muchos aspectos del frontend.
Además se podría definir mejor qué tipos de pruebas manuales se deberían hacer para validar todo el frontend, o por lo menos las partes que no se prueban de forma automática.

Automatizar el testing funcional o de caja negra

Continuando con las pruebas, ahora vamos a hablar de cómo se generaron las pruebas automáticas del backend.
Para este tipo de pruebas ya existían muchas que probaban el funcionamiento del backend. Pero para las funciones nuevas o las funciones que se arreglaron, no existían pruebas que aseguraran el funcionamiento correcto de nuestro código. Es por esto que en las entregas en las que se agregó funcionalidad al backend, se crearon pruebas. Durante la segunda entrega se utilizó TDD, lo que hizo que se crearan pruebas unitarias donde fuera necesario. Y durante la tercera entrega se aplicó BDD, es por esto que se crearon pruebas con SpecFlow en el backend, ya que eso crea pruebas más similares a como se definen los criterios de aceptación.
Lamentablemente estas pruebas no se pudieron implementar correctamente, y por pobre manejo del tiempo se abandonaron, lo cual es una lastima ya que como pruebas no solo son similares a los criterios de aceptación, sino que además son muy simples de comprender y fáciles de modificar y debuggear. En el futuro se debería planificar mejor para no dejar incompleta una de las partes más importantes del BDD, el testing automático.

Conclusiones

A partir de lo evidenciado en este informe se desprenden varias conclusiones sobre las metodología DevOps utilizada y los aprendizajes del equipo.
DevOps busca hacer más rápido el proceso de desarrollo, feedback y aprendizaje. En este proyecto esto se vió aplicado a través de varias entregas, casi periódicas, con diferentes objetivos y formas de encararlas. Esto forzó al equipo a estar constantemente cambiando y buscando la mejor solución frente a los diferentes obstáculos. Intentando, como busca DevOps, entregar rápidamente valor, recibiendo feedback de nuestro trabajo lo más rápido posible y aprendiendo a medida que avanzamos y trabajamos como mejorar los procesos que estamos utilizando y cómo adaptarnos a nuevos.
Estas mismas entregas permitieron analizar más seguido lo que estaba funcionando bien y mal, lo que podía mejorar y lo que se podía sacar, lo cúal permitió una mejora a medida que se avanzaba en el proyecto. Esto permitió que cada entrega siguiente fuera mejor que la anterior, mejorando el resultado final. Si se hubiera hecho una sola entrega final nos hubiéramos percatado de estos detalles después de terminar el proyecto.
Sin embargo, no todo el proceso fue efectivo. Durante la tercera entrega hubo dificultades de horarios entre los integrantes del equipo y se retrasó mucho el trabajo planificado. Esto fue lo que causó que parte de esta entrega no se pudiera terminar a tiempo. Como se quería entregar valor periódicamente y no había mucho espacio de ajuste, un período en el que no se pudo juntar el equipo causó problemas. Este también puede haber sido un fallo

de planificación pero eso no cambia que al trabajar con espacios de tiempo más cortos, los imprevistos pueden tener mayor riesgo.
Dicho esto, concluimos que es parte del proceso analizar estos imprevistos e intentar de evitar que vuelvan a pasar, algo que DevOps puede analizar y resolver rápidamente, ya que busca el feedback y la solución de problemas constante.
Un aspecto de DevOps que no se pudo aprovechar mucho fue el de feedback con clientes y product owners ya que estos no existían realmente. De todas formas, la instancia simulada permitió ver algunas de sus ventajas. De la misma forma que mencionamos antes, recibir feedback a medida que se avanza en el proceso, en lugar de al final, termina aportando más valor en total.
Como una de las conclusiones fundamentales, las ventajas de DevOps fueron muy claramente evidenciadas. Tanto el valor que aporta al producto y al product owner, a través de la entrega periódica y constante, como el valor que aporta al equipo y al proceso de ingeniería a través del feedback, la reflexión y el análisis, sobre todo en un proyecto de mantenimiento como el que se estaba haciendo durante este obligatorio.
Con esto en mente, de ser necesario seguir con el mantenimiento y agregando valor al software de la entrega, el equipo sería capaz de enfrentar el desafío con una perspectiva de DevOps, utilizando lo aprendido a través de la práctica de la metodología y la retroalimentación en base a métricas que se obtuvieron sobre el final del proyecto.

Guía de instalación para desarrollo y despliegue en producción.

A continuación detallaremos los pasos a seguir para poder correr el proyecto denominado Pharma Go, está guía está dedicada únicamente para usuarios de **Windows**, dado que Microsoft SQL Server se encuentra solo en esa plataforma.
Para poder ejecutar este proyecto en un entorno de producción será necesario tener las siguientes herramientas instaladas previamente en nuestras máquinas:

a. Node, utilizamos la versión 18.17.1
b. .NET, utilizamos el SDK versión 6.0
c. Microsoft SQL Server, versión 2018

1. **Base de datos**
   El DBMS utilizado para esta entrega es Microsoft SQL Server para poder correr la base de datos los pasos a seguir son:
   1. Crear la base de datos, para esto ejecutar el archivo “PharmaGoDb.sql”
   2. Cargar los datos de prueba, para eso restaurar con el archivo “PharmaGoDb.bak”
   3. Modificar el connection string, en el archivo /Backend/PharmaGo.WebApi/appsettings.json poner las credenciales de la base de datos PharmaGoDb

2. **Backend**
   Para poder correr el backend, es necesario en la ruta /Backend/PharmaGo.WebApi/ correr el siguiente comando:
   1. dotnet run

3. **Frontend**
   Los pasos para correr el frontend de nuestro proyecto son los siguientes:
   Dentro de la carpeta /Frontend/ correr los siguientes comandos:
   1. npm install, para instalar las dependencias de nodejs.
   2. ng serve, de esta manera corremos esta instancia de node.

De esta manera si seguimos todos los pasos en la ruta http://localhost:4200/ tendremos a nuestra página web funcionando correctamente.