Implementación del proceso automatizado para la generación de métricas: Code Health

Juan Guillermo Espitaleta Benítez

Informe final Modalidad Práctica Empresarial

Asesor

Miriam Cecilia Delgado Cadavid, Ingeniera de Sistemas

Universidad de Antioquia

Facultad de Ingeniería

Ingeniería de Sistemas

Medellín

2022

TABLA DE CONTENIDO

LISTA DE FIGURAS

RESUMEN

Sofka Technologies es una empresa innovadora originaria de la ciudad de Medellín que hace parte de la industria tecnológica. En el área de "Aseguramiento de la Calidad" tenía como responsabilidad con uno de sus clientes principales la generación de métricas de "salud del código" y políticas establecidas para los proyectos de automatización de pruebas y seguimiento de actividades dentro de los repositorios de control de versiones; esta labor era bastante dispendiosa ya que se realizaba manualmente con estrategias de "code review"; debido a esto se decidió desarrollar e implementar un proceso automatizado de generación de métricas e informes, con el cual se pudiera optimizar el cumplimiento de esta responsabilidad, permitiendo verificar que se estuviera dando cumplimiento y seguridad a los acuerdos que se tenían con el cliente anteriormente mencionado.

Para la implementación de "Code Health" se siguió el marco de trabajo Scrum como metodología, el cual permite desarrollar software de forma agil y posibilita crear un producto de forma incremental a través de periodos de tiempo llamados Sprints. Se muestra el resultado y el análisis del desarrollo profundizando en los ejes temáticos que hace referencia a la arquitectura de la aplicación, la interacción entre los componentes que hace parte de la automatización, la declaración de los escenarios que describen el comportamiento del proceso automático (los pasos, las secuencias y las acciones que se ejecutan), los recursos usados de donde se obtiene y almacena la información. Se plantean las conclusiones que permiten validar el alcance inicial del proyecto y entender de forma global el resultado final de la implementación del proceso automatizado.

Palabras clave – Automatización de pruebas, metodologías ágiles, Scrum, repositorios, salud del código, aseguramiento de la calidad, GitLab, control de versiones.

ABSTRACT

Sofka Technologies is an innovative company from the city of Medellin that is part of the technology industry. In the "Quality Assurance" area has the responsible with one of his main clients for the generation of "code health" metrics and established policies for test automation projects and monitoring of activities within the version control repositories.; This work was quite time-consuming since it was done manually with “code review” strategies; Due to this, it was decided to develop and implement an automated process for generating metrics and reports, with which the fulfillment of this responsibility could be optimized, allowing to verify that the agreements with the aforementioned client were being fulfilled and secured.

For the implementation of "Code Health" the Scrum framework was followed as a methodology, which allows developing software in an agile way and makes it possible to create a product incrementally through periods of time called Sprints. The result and the analysis of the development are shown, delving into the thematic axes that refer to the architecture of the application, the interaction between the components that are part of the automation, the declaration of the scenarios that describe the behavior of the automatic process (the steps, sequences and actions that are executed), the resources used from where the information is obtained and stored. The conclusions that allow to validate the initial scope of the project and to understand in a global way the final result of the implementation of the automated process are proposed.

.

Keywords – Test automation, agile methodologies, Scrum, repositories, code health, quality assurance, Git, GitLab, version control.

I. INTRODUCCIÓN

Sofka viene de la unión de Software + Kaizen (Filosofía de mejora continua, compromiso y disciplina).

Sofka Technologies es una compañía que desde sus inicios viene desarrollando el talento tanto técnico como humano, para estar a la vanguardia de la industria tecnológica generando soluciones de alto impacto para los clientes con los que siempre se trabaja en equipo.

Entre los servicios que presta Sofka Technologies está el de Agile Testing (pruebas de software que sigue los principios del desarrollo ágil de software) responsabilidad del área de QA (Quality Assurance en español Aseguramiento de la Calidad) de la compañía, donde dependiendo del cliente con el que se esté trabajando se definen políticas para el manejo de los proyectos y salud del código; con uno de sus clientes se tenía como responsabilidad la generación de métricas de salud del código de los proyectos de automatización de pruebas y seguimiento de actividades dentro de los repositorios, lo cual era un proceso y una tarea bastante dispendiosa, ya que se realizaba manualmente con estrategias de “code review”; con Code Health esta labor se pudo automatizar implementando un proceso a la medida que respondiera a la necesidad con este cliente inicialmente y en un futuro con otros clientes de la compañía.

II. OBJETIVOS

A. Objetivo general

Implementar un proceso automático para generación de métricas que apalanque las mediciones de salud de código de los proyectos activos de automatización de pruebas del área de calidad de la compañía.

B. Objetivos específicos

- Diseñar un proceso automático, configurable y escalable basado en el framework de automatización de pruebas propio de Sofka con el fin de que el código sea mantenible y fácil de modificar.
- Generar informes sobre las métricas, hallazgos y actividad reciente de los repositorios de automatización de pruebas activos para mejorar el seguimiento sobre los proyectos.
- Documentar la implementación del proceso de automatización para la generación de métricas.

III. MARCO TEÓRICO

Gracias al Manifiesto Ágil redactado en el año 2001[1] con dos de sus cuatro pilares ("Software funcionando sobre documentación extensiva" y "respuesta ante el cambio sobre seguir un plan") y a la evolución de modelos de ciclo de vida de desarrollo de software como lo fue el modelo en V en el año 2005[2] donde para cada actividad de desarrollo había una actividad de pruebas, la calidad se volvió responsabilidad de todos; por lo tanto el aseguramiento de la calidad y las pruebas empezaron a cumplir un papel fundamental para el éxito de proyectos y todo lo relacionado al ciclo de vida del desarrollo de software, y aunque como se mencionó anteriormente la calidad es responsabilidad de todos, hay roles que se enfocan en mayor medida de esta parte. A continuación se mencionan diferentes conceptos y definiciones que ayudan a mejorar la comprensión y entendimiento de todo lo relacionado al aseguramiento de la calidad y las pruebas en la industria del software, y cual es el papel que juega Code Health.

Para empezar hay que mencionar por qué es importante probar de forma rigurosa los componentes y sistemas, y su documentación asociada, ya que puede ayudar a reducir el riesgo de que se produzcan fallos durante la operación. Al detectar defectos, y posteriormente corregirlos, se contribuye a la calidad de lo que se probó. Además, la prueba del software también puede ser necesaria para cumplir con requisitos contractuales o legales o estándares específicos de la industria según la organización International Software Testing Qualifications Board (2018) de aquí en adelante ISTQB[3].

Las pruebas y el aseguramiento de la calidad se realizan dentro del ciclo de vida del desarrollo de software; este ciclo de vida describe "los tipos de actividad que se realizan en cada etapa de un proyecto de desarrollo de software, y cómo las actividades se relacionan entre sí de forma lógica y cronológica"[3]. Hay diferentes modelos de ciclo de vida de desarrollo de software, los cuales requieren diferentes enfoques de prueba.
Según ISTQB[3] para que las pruebas sean adecuadas en los diferentes modelos de ciclo de vida de desarrollo de software se tienen las siguiente características:

- Para cada actividad de desarrollo, hay una actividad de prueba asociada.
- Cada nivel de prueba tiene objetivos específicos para ese nivel.
- El análisis y diseño de la prueba para un nivel de prueba dado comienza durante la actividad de desarrollo correspondiente.

Como se mencionó anteriormente cada nivel de prueba tiene objetivos específicos, por lo cual es importante diferenciar cuales son estos niveles; según la organización internacional de certificación de la calidad del software[3] afirma que hay 4 niveles que son los siguientes:

- Prueba de componente, comúnmente conocidas como pruebas unitarias, en las cuales se verifica el funcionamiento correcto de métodos, clases, módulos, en general

unidades individuales de software; este nivel de prueba es responsabilidad de los desarrolladores.

- Prueba de integración, nivel de prueba que implica verificar el funcionamiento correcto de varios módulos o componentes de una aplicación como un grupo; en este nivel de prueba pueden participar desarrolladores como probadores (testers).

- Prueba de sistema, nivel de prueba que verifica a nivel general el funcionamiento correcto de las diferentes interfaces de los subsistemas que componen el objeto de prueba; los tester tienen su mayor participación en este nivel.

- Prueba de aceptación, nivel de prueba que busca validar que el sistema está completo y que funciona como se esperaba.

Por otra parte, los modelos de ciclo de vida de desarrollo de software se clasifican en modelos de desarrollo secuencial y modelos de desarrollo iterativos e incrementales.
Independientemente del modelo de ciclo de vida que se esté ejecutando dentro de cualquier proyecto de desarrollo de software se pueden realizar dos tipos de pruebas: dinámicas o estáticas.
Las pruebas dinámicas según Gomez[4] son aquellas que se realizan mientras el código está en ejecución. Tienen como objetivo asegurar que el software se comporte de acuerdo a los requerimientos y especificaciones mediante la ejecución de pruebas funcionales y no funcionales. Estas pruebas se enfocan en la detección y confirmación de la corrección de defectos en el software y generalmente se realizan en una etapa más tardía que las pruebas estáticas, por lo cual, los defectos encontrados pueden ser más costosos.

Además Gomez[4] afirma que *las pruebas estáticas* a diferencia de las pruebas dinámicas, no requieren de la ejecución de software para ser realizadas y el objetivo de estas es la revisión de productos de trabajo como documentos, requerimientos, especificaciones, casos de prueba, planes de prueba, código, entre otros. Estas pruebas se enfocan en la prevención de defectos y en la detección temprana de los mismos, ya que se pueden realizar en cualquier etapa del ciclo de vida del software según la información que se tenga disponible y el costo de corregirlos puede ser menor.[4]

Code Health se puede decir que clasifica dentro de lo que es una prueba estática, ya que su principal objetivo es la revisión de código como objeto de prueba o producto de trabajo, y aunque al ser un proceso automatizado derivado de la automatización de pruebas, no ejecuta el código que va a ser inspeccionado. Code Health no busca encontrar o prevenir defectos funcionales dentro del código, sino, más bien verificar que se esté dando cumplimiento y seguridad a los acuerdos que se tienen en el proyecto con el cliente.

Tal como se mencionó anteriormente Code Health es un proceso automatizado derivado de la automatización de pruebas, ¿Pero qué es la automatización de pruebas? Hamilton[5] un tester experto define que es una técnica de prueba de software que se realiza utilizando

herramientas de software o frameworks de pruebas automáticas especiales para ejecutar un conjunto de casos de prueba. Por el contrario, la prueba manual la realiza un ser humano sentado frente a una computadora y ejecutando cuidadosamente los pasos de la prueba. También Hamilton afirma que “El software de prueba de automatización también puede ingresar datos de prueba en el sistema bajo prueba, comparar los resultados esperados y reales y generar informes de prueba detallados” Por lo tanto las pruebas automatizadas siguen el mismo procedimiento que se realiza en las pruebas manuales. Entonces ¿cuáles son los beneficios de las pruebas automatizadas? Thomas[5] indica que con ellas se aumenta la cobertura de las pruebas, se aumenta la eficacia y la velocidad con la que se realizan las ejecuciones; además comenta que estas pruebas se realizan sin supervisión humana y en comparación de las pruebas manuales no son subjetivas y al no ser supervisadas no tienden a volverse aburridas y por lo tanto, pueden ser menos propensas a errores.

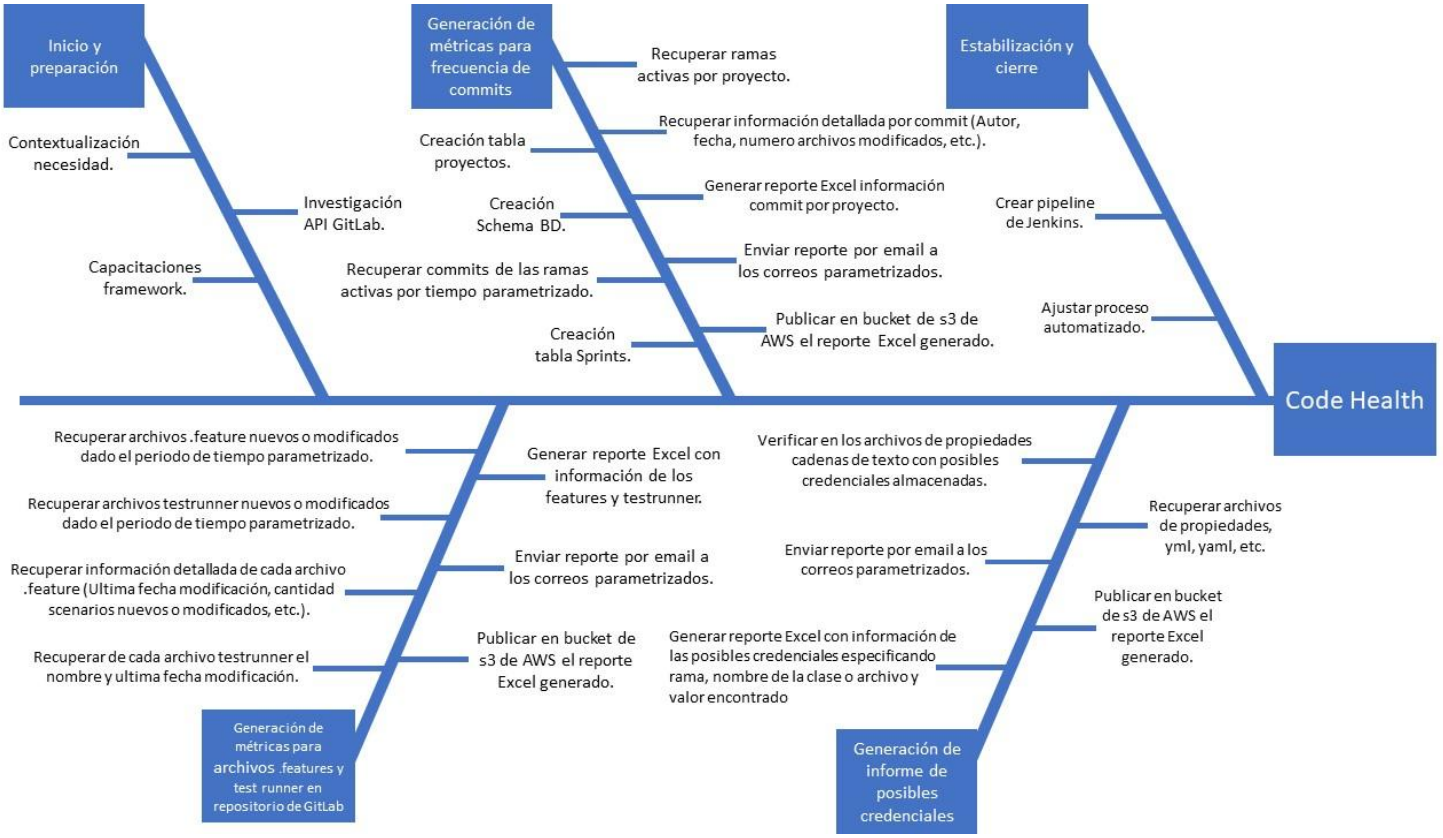
IV. METODOLOGÍA

La metodología de trabajo de Sofka Technologies es basada en el manifiesto ágil, lo que permite según el contexto adaptarse y decidir cuál metodología o marco de trabajo es el más adecuado para cualquier tipo de proyecto o actividad; este proyecto se realizó dentro del marco de trabajo ágil Scrum, el cual fue altamente recomendado debido a la gran experiencia que tiene la compañía haciendo uso de este framework, por su fácil uso y la flexibilidad que este permite, ya que el proyecto fue susceptible a cambios y por tanto la importancia de utilizar un marco de trabajo que permita la adaptación al cambio; Scrum permite tener iteraciones (Sprints) con duración de dos semanas e incrementos periódicos sobre el objeto de trabajo; se hizo uso de los principales roles que son:

- *Product Owner:* Persona encargada de tener una visión clara de lo que se va a construir y transmitir esa visión al equipo de desarrollo.
- *Scrum Master:* Líder al servicio del equipo Scrum, ayudando a mantenerse enfocado de los objetivos del proyecto y facilitando su correcto desarrollo.
- *Development Team:* Conjunto de personas con conocimientos técnicos que desarrollan el producto del proyecto.

A continuación se presenta un diagrama fishbone (**Figura 1**) con las actividades relacionadas a cada historia de usuario épica y partes del proyecto.

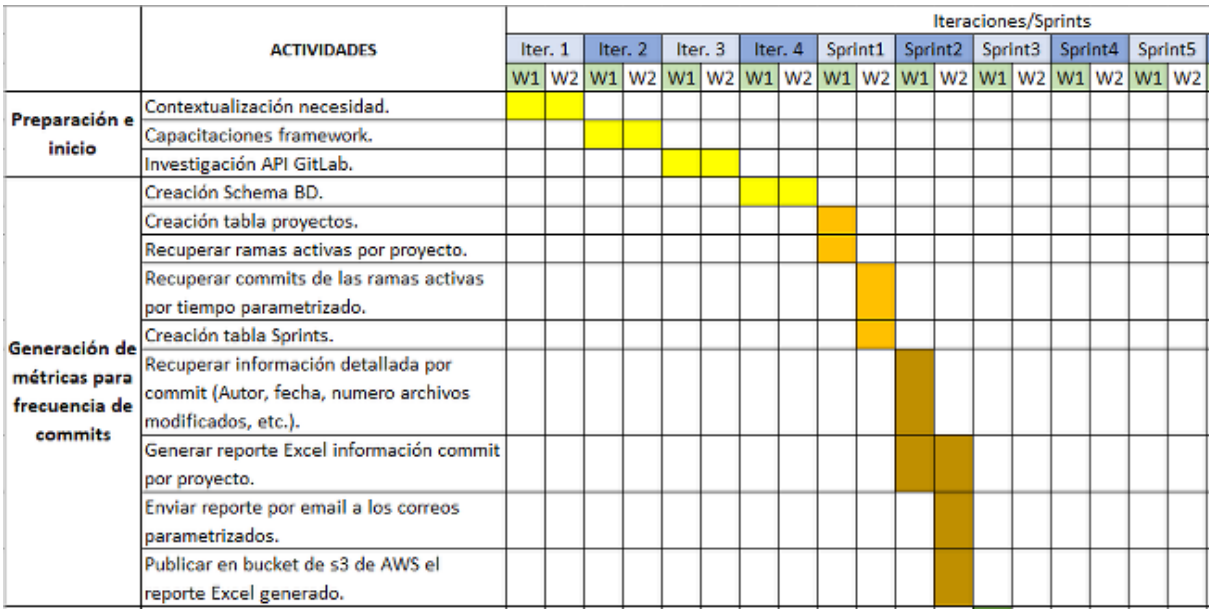
Figura 1.
Diagrama fishbone

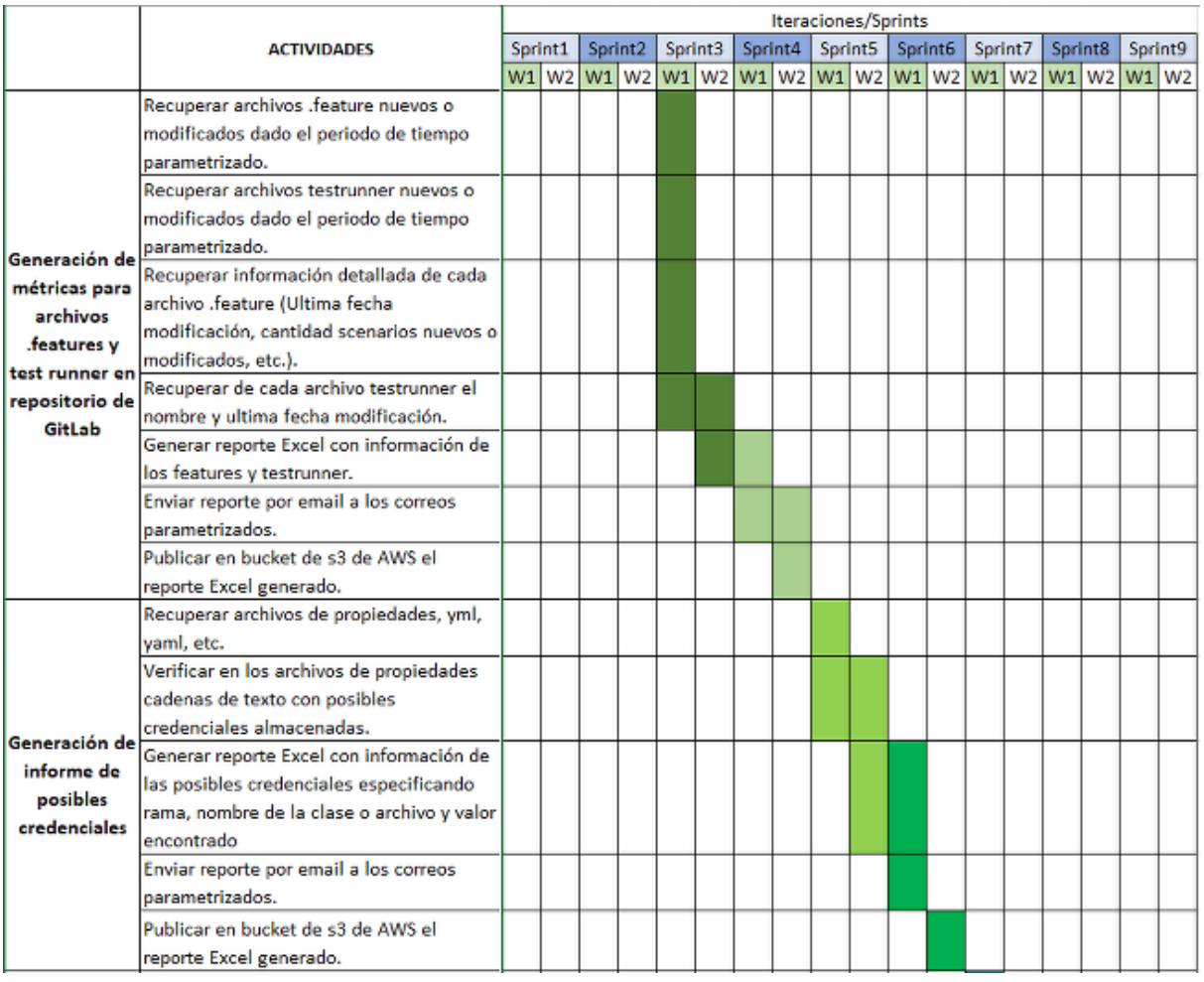


Nota. Fuente: Elaboración propia

Finalmente se organizan dichas actividades dentro de un diagrama de Gantt (**Figura 2**) donde se muestran las actividades desarrolladas dentro de cada iteración.

Figura 2.
Diagrama de Gantt





	ACTIVIDADES	Iteraciones/Sprints															
		Sprint1		Sprint2		Sprint3		Sprint4		Sprint5		Sprint6		Sprint7		Sprint8		Sprint9	
		W1	W2	W1	W2	W1	W2	W1	W2	W1	W2	W1	W2	W1	W2	W1	W2	W1	W2
Estabilización y cierre	Crear pipeline de Jenkins.																		
	Ajustar proceso automatizado.																		
	Elaborar informe final.																		

Nota. Fuente: Elaboración propia

V. RESULTADOS Y ANÁLISIS

El alcance definido de Code Health fue implementar un proceso automático de generación de métricas como mínimo producto viable, haciendo uso de un framework de automatización de pruebas y otras tecnologías dentro de un marco de trabajo iterativo e incremental como lo es Scrum, teniendo como base las siguientes tres historias de usuario épicas listadas:

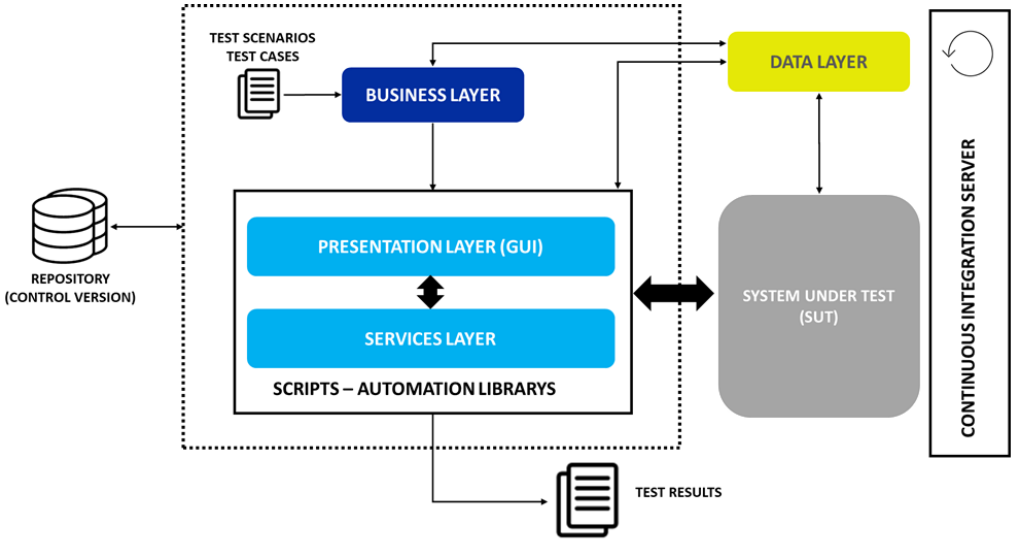
- HU01. Generación de métricas para frecuencia de commits.
- HU02. Generación de métricas para archivos .features y test runner en repositorio de gitlab.
- HU03. Generación de informe de posibles credenciales.

Dicho esto, se presentan diferentes ejes temáticos que explican cómo funciona y fue desarrollado Code Health a nivel técnico sin profundizar mucho sobre la codificación y que permiten validar el cumplimiento del alcance inicial propuesto.

A. ARQUITECTURA, PAQUETES E INTERACCIONES

Code Health fue desarrollado basándose en el framework de automatización de pruebas de Sofka Technologies, el cual es un conjunto de librerías implementadas en java y que cumplen con una arquitectura previamente definida para el área de Aseguramiento de Calidad (**Figura 3**).

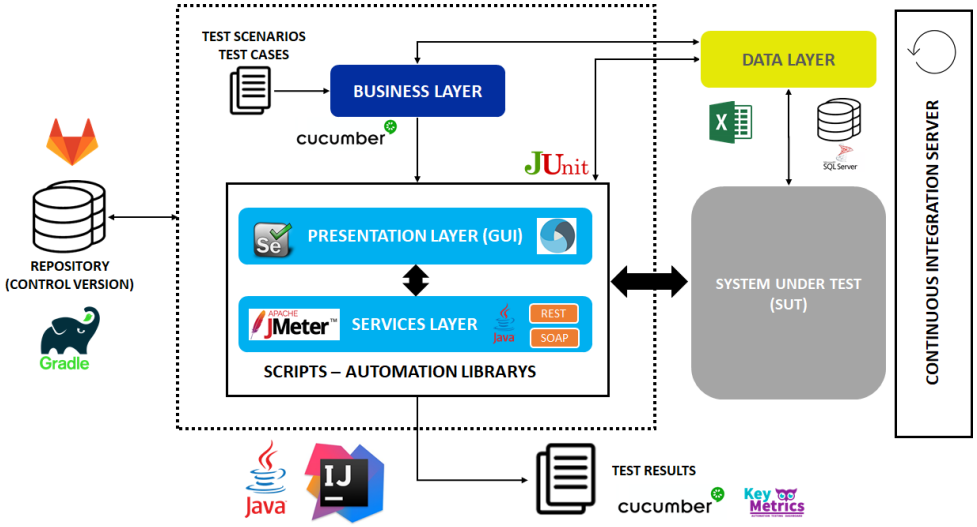
Figura 3.
Arquitectura del Framework



Nota. Fuente: Elaboración propia

Dicho framework contiene métodos, utilidades y herramientas para realizar automatización en FRONTEND y BACKEND, lo cual optimiza de forma significativa la construcción de autómatas de pruebas(**Figura 4**).

Figura 4.
Herramientas utilizadas generalmente



Nota. Fuente: Elaboración propia

Para el caso de Code Health no se utilizaron métodos, utilidades y herramientas para automatizar FRONTEND, se utilizaron algunos para BACKEND, ya que toda la información que se deseaba obtener es proveniente de los repositorios de control de versionamiento de GitLab haciendo uso de su API REST de la cual se hablará más adelante; las principales herramientas utilizadas en Code Health son:

- **Cucumber,** herramienta que utiliza una sintaxis llamada Gherkin la cual tiene como principal característica ser un lenguaje de dominio específico [6] que ayuda a separar el lenguaje lógico/técnico y convertir las instrucciones en un lenguaje comprensible para casi cualquier persona, ya que se puede plasmar como descripciones de comportamiento o conducta, haciendo uso de la estrategia de desarrollo BDD (Behavior Driven Development, en español desarrollo dirigido por comportamiento).

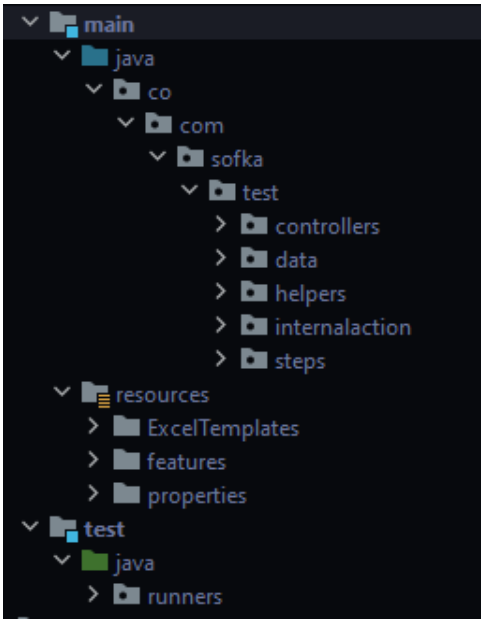
- **Lombok** es una librería de Java que permite con el uso de anotaciones la implementación o creación en tiempo de compilación métodos get, set, constructores, entre otros; lo que permite tener un código visualmente más limpio, para el caso de Code Health se utilizó principalmente para la creación de los DTO (Data Transfer Object).

- **JUnit** es un framework usado para la creación de pruebas unitarias y de integración, para el desarrollo de Code Health se utilizó junto con Cucumber principalmente para la creación del runner que se encarga de ejecutar o disparar el proceso automático y

además con el uso de anotaciones para la creación del setup donde se crea la ruta donde se almacenan las evidencias de la ejecución dependiendo del sistema operativo.

A nivel de carpetas y archivos Code Health tiene una estructura de paquetes bastante comprensible y organizada (**Figura 5**), dividida en tres partes, una donde se almacenan todo lo relacionado a las clases desarrolladas en Java, una segunda donde se guardan todos los recursos (plantillas de Excel, features y propiedades) y un último paquete que contiene el runner para ejecutar el proceso automatizado.

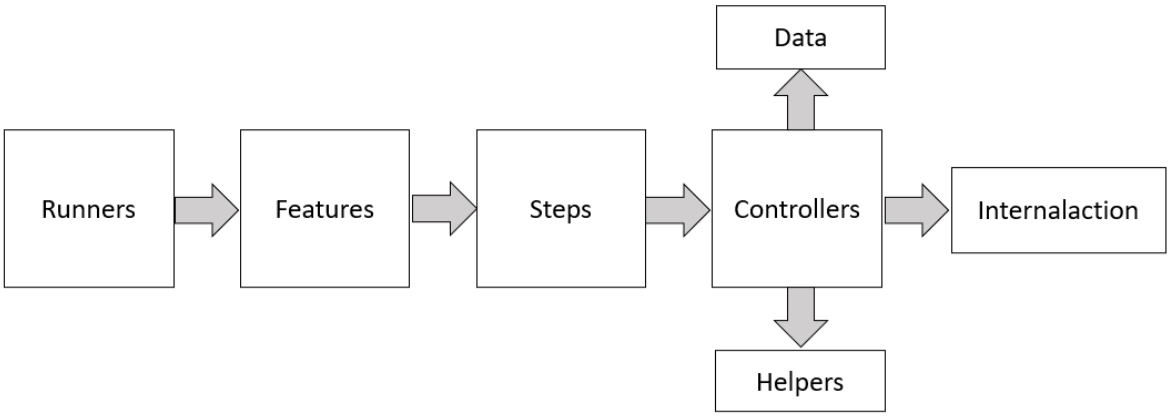
Figura 5.
Estructura y paquetes



Nota. Fuente: Elaboración propia

La interacción entre los componentes o los paquetes inicia desde los runners (también llamados testrunners en automatización de pruebas) pasando por los features, después a los steps (comúnmente conocidos como step definitions), luego hacia los controllers que se encargan de orquestar las acciones a realizar, ya sea de extracción de información de los repositorios de GitLab, realizar operaciones en la base de datos, procesamiento de datos o generar los informes de Excel de la salud del código con la información y métricas obtenidas(**Figura 6**).

Figura 6.
Flujo ejecución de componentes



Nota. Fuente: Elaboración propia

El paquete "controllers" almacena tres controladores (**Figura 7**) que son los encargados de realizar las acciones principales y llamados a clases utilitarias para la consulta, manejo de datos y la generación de los informes; dichos controladores son:

Figura 7.
Paquete controlles

Nota. Fuente: Elaboración propia

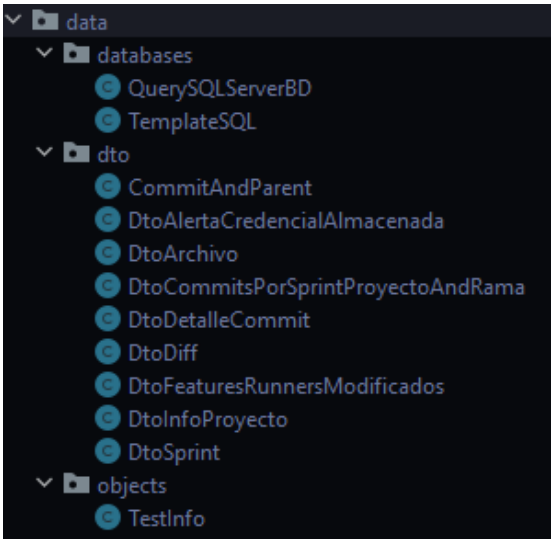
- ControllerData: Controlador encargado de realizar las consultas e inserciones en la base de datos, el procesamiento de la información como por ejemplo verificar si los datos obtenidos cumplen con las características de un credencial almacenada, procesar la lista de correos de los destinatarios e interesados para el envío de los reportes, decodificar contenido encriptado, etc.

- ControllerFiles: Controlador encargado de la creación de las carpetas e informes, selección de las plantillas (archivos de Excel), creación de las copias a partir de las plantillas y el llenado de las mismas.

- ControllerGitLab: Controlador encargado de obtener la información acerca de cada proyecto a través del consumo de la API REST de GitLab, como por ejemplo: verificar si es posible el acceso a los repositorios, información básica de los proyectos, cantidad de commits realizados por rango de fechas, el detalle de cada commit, las ramas activas, los nombres de los archivos modificados, los escenarios de pruebas modificados, etc.

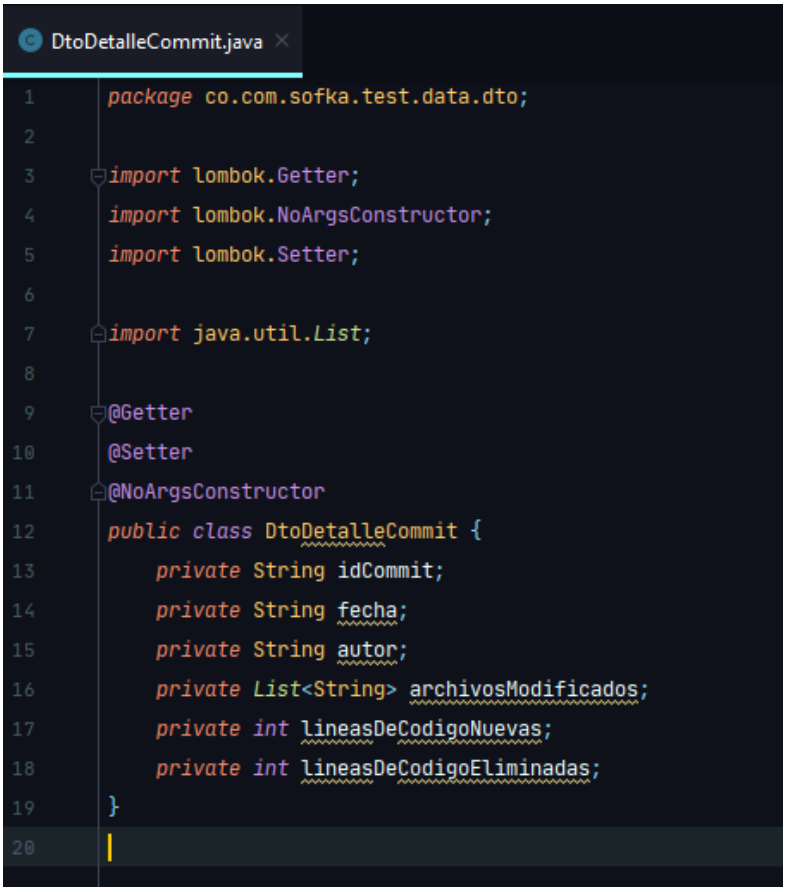
El paquete “data” (**Figura 8**) almacena todo lo relacionado a las clases encargadas de las operaciones en la base de datos, además contiene los DTOs (Data Transfer Object) para el mapeo de los datos de cada respuesta obtenida desde la API REST de GitLab implementando las anotaciones de Lombok como se evidencia en la imagen de ejemplo(**Figura 9**), además el manejo de los Logs de ejecución realizada por el “TestInfo” dentro del paquete “objects”.

Figura 8.
Paquete data



Nota. Fuente: Elaboración propia

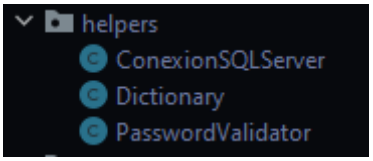
Figura 9.
DTO con Lombok



Nota. Fuente: Elaboración propia

El paquete "helpers" contiene tres clases utilitarias (**Figura 10**) que cumplen diferentes funciones, las cuales son:

Figura 10.
Paquete helpers



Nota. Fuente: Elaboración propia

- ConexionSQLServer: Clase encarga del manejo de la conexiones a la base de datos, clase hija de "ConnectionDB" (propia de Framework), es decir que hereda sus atributos y métodos, de los cuales destaca la implementación de la interfaz "AutoCloseable" que permite cerrar las conexiones de manera automática.

- PasswordValidator: Clase vital en la generación del tercer informe referente a las posibles credenciales almacenadas; como tal esta clase aplica ingeniería inversa a una cadena de texto dada, que hace parte del contenido de un archivo de

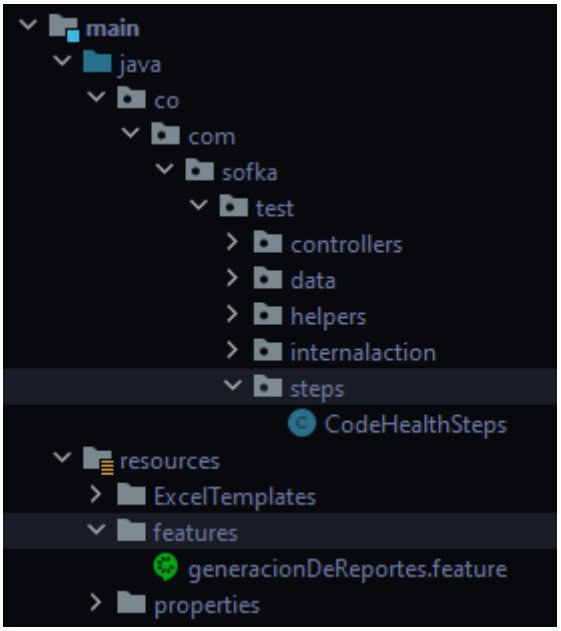
configuración, buscando coincidencias y ciertas características que se utilizan para la creación de una credencial o una contraseña segura, tales como: la longitud, letras en minúscula, números y caracteres especiales. Esta función la realiza haciendo uso de expresiones regulares para encontrar dichas coincidencias.

- Dictionary: Clase configurable la cual almacena constantes referentes a extensiones y nombres de archivos de configuración a obtener para buscar las posibles credenciales almacenadas; además contiene la lista de posibles excepciones ya identificadas para que no sean tenidas en cuenta como posibles credenciales, esta lista se puede complementar si se encuentran nuevas excepciones.

El paquete "internalaction" contiene como tal una clase utilitaria que se utiliza para consumo de las APIs y servicios web retornando las respuestas (body response).

Los paquetes "steps" y "features" (**Figura 11**) aunque están separados contienen dos componentes que trabajan de forma articulada, gracias a la herramienta Cucumber que se había mencionado anteriormente; el archivo "generacionDeReportes.features" ejecuta cada instrucción descrita en lenguaje no técnico (sintaxis Gherkin) y busca dicha expresión regular dentro de la clase "CodeHealthSteps" (**Figura 13**), permitiendo seguir una secuencia de pasos y de esta forma ejecutar los escenarios que describen cómo se generan cada uno de los tres reportes.

Figura 11.
Paquetes steps y features



Nota. Fuente: Elaboración propia

Actualmente hay cuatro escenarios implementados en el archivo de features (**Figura 12**) los cuales tienen los siguientes propósitos:

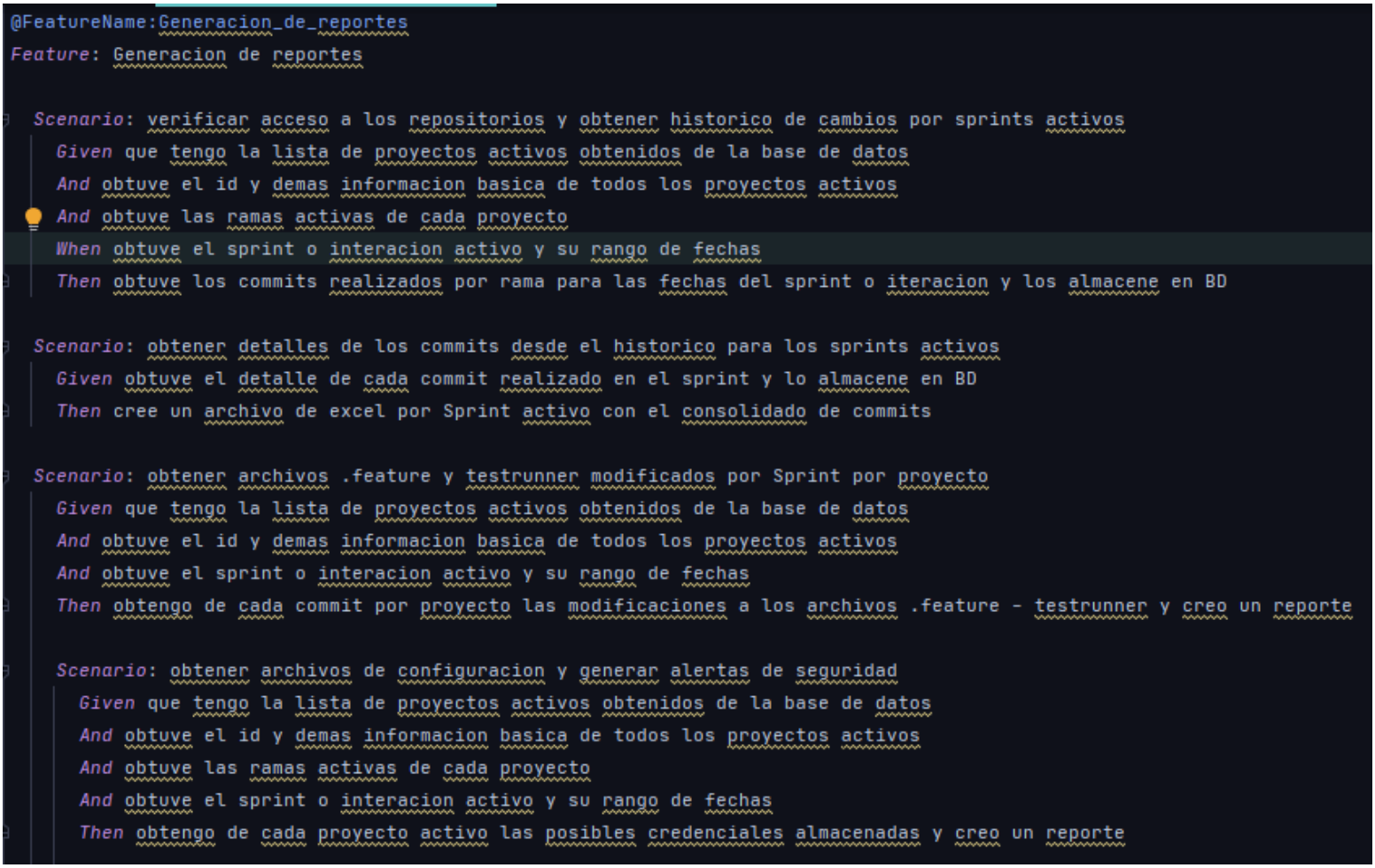
- `Scenario: verificar acceso a los repositorios y obtener histórico de cambios por sprints activos`
  Este primer escenario consulta en la base de datos los proyecto y los sprints activos, posteriormente desde GitLab se verifica si se tiene acceso al repositorio de cada uno de los proyectos activos, si se cumple esta condición procede a obtener el historial de cambios por sprint, es decir los commits realizados para los rangos de fechas de cada sprint activo por proyecto; luego almacena un registro en base de datos por cada commit encontrado, asociándolo al id del proyecto en GitLab y al sprint.

- `Scenario: obtener detalles de los commits desde el histórico para los sprints activos`
  Este escenario se encarga de consultar en la base de datos el consolidado de cada uno de los commits realizados por proyecto y por sprint, creando el primer reporte, cumpliendo con la primera historia de usuario épica listada en el alcance inicial "HU01. Generación de métricas para frecuencia de commits."

- `Scenario: obtener archivos .feature y testrunner modificados por Sprint por proyecto`
  El tercer escenario, como bien lo indica su descripción, obtiene los archivos .features y testrunner que hayan sido modificados, estos se obtienen dentro del detalle de cada commit proveniente del histórico de cambios por proyecto y por sprint, generando con esta información el segundo informe, cumpliendo con la segunda historia de usuario épica "HU02. Generación de métricas para archivos .features y test runner en repositorio de gitlab."

- `Scenario: obtener archivos de configuración y generar alertas de seguridad`
  El cuarto y último escenario, es el encargado de generar el tercer informe referente a posibles credenciales almacenadas en archivos de configuración, cumpliendo con la tercera historia de usuario épica "HU03. Generación de informe de posibles credenciales.". Este escenario obtiene los archivos de configuración de cada proyecto por rama, luego se obtiene los contenidos de cada archivo de configuración y se procede a separar el texto por líneas o cadenas de texto individuales, estas líneas son procesadas una a una por la clase utilitaria "PasswordValidator" la cual indica cuando se encuentra una posible credencial, estas posibles credenciales se consolidan y se insertan dentro del tercer informe.

Figura 12.
Scenarios



Nota. Fuente: Elaboración propia

Figura 13.
Ejemplo Steps

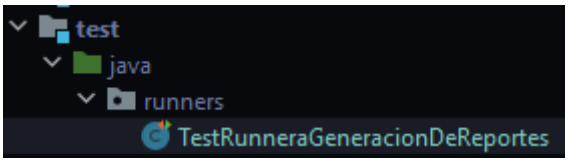
CodeHealthSteps.java

```
106
107     @Given("que tengo la lista de proyectos activos obtenidos de la base de datos")
108     public void queTengoLaListaDeProyectosActivosObtenidosDeLaBaseDeDatos() {
109         proyectosActivosList = controllerData.getListadoProyectoActivos();
110     }
111
112     @And("obtuve el id y demas informacion basica de todos los proyectos activos")
113     public void obtuveElIdYDemasInformacionBasicaDeTodosLosProyectosActivos() {
114         proyectosActivosList = controllerGitLab.getIdAndInformacionProyectosActivos(proyectosActivosList);
115     }
116
117     @And("obtuve las ramas activas de cada proyecto")
118     public void obtuveLasRamasActivasDeCadaProyecto() {
119         proyectosActivosList = controllerGitLab.getRamasParaCadaProyecto(proyectosActivosList);
120     }
121
122     @And("obtuve el sprint o interacion activo y su rango de fechas")
123     public void obtuveElSprintOInteracionActivoYSuRangoDeFechas() { sprintList = controllerData.getSprintsActivos(); }
```

Nota. Fuente: Elaboración propia

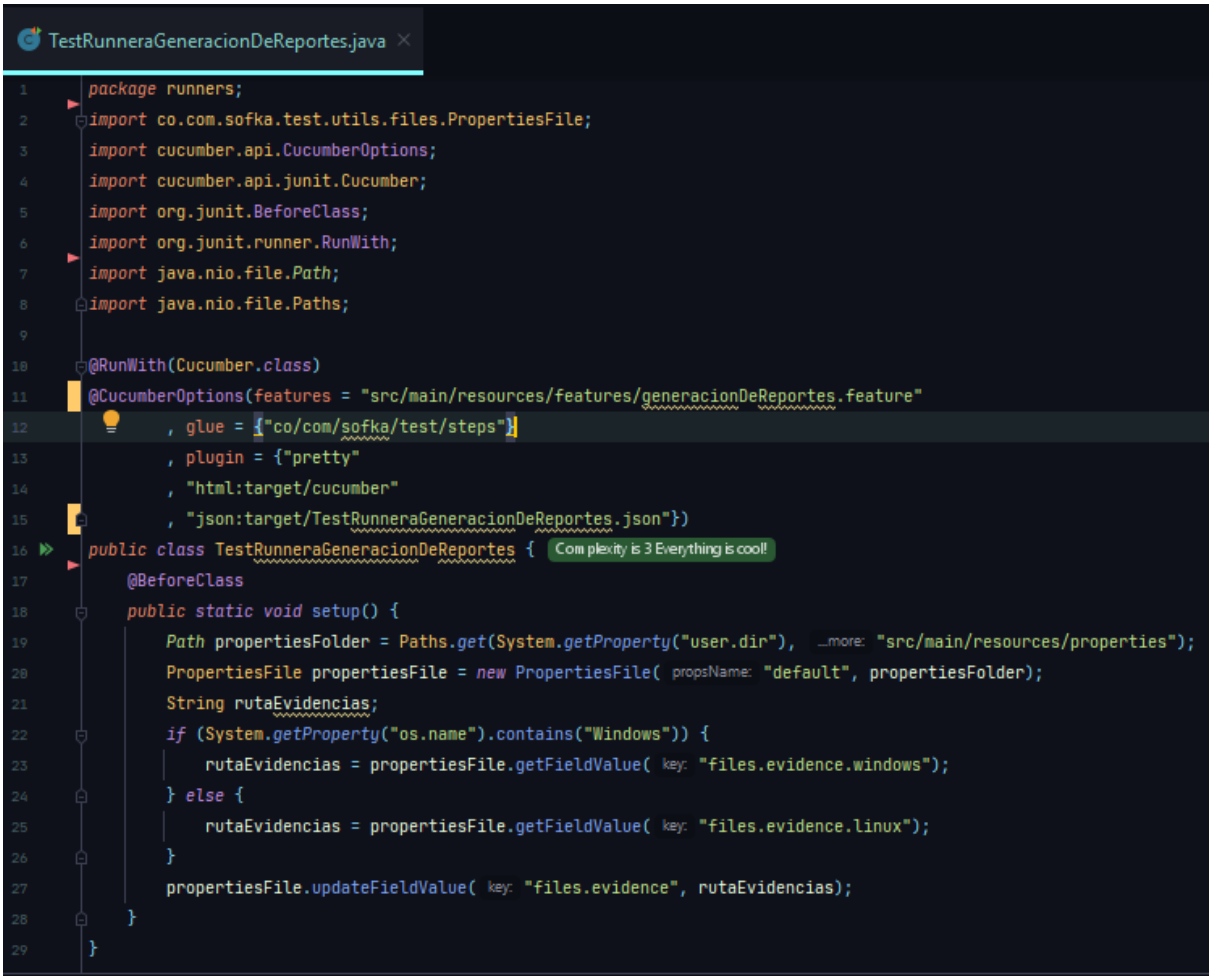
Finalmente el paquete "runners" contiene la clase encargada de ejecutar el proceso automatizado para la generación de métricas y creación de informes(**Figura 14**), la cual implementa anotaciones de Cucumber y JUnit para la ejecución y la creación de evidencias de la misma (**Figura 15**).

Figura 14.
Paquete runners



Nota. Fuente: Elaboración propia

Figura 15.
TestRunneraGeneracionDeReportes

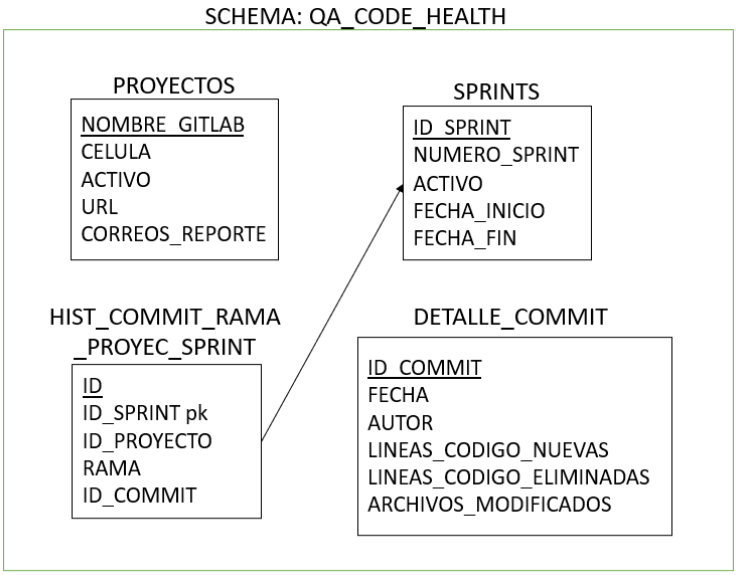


Nota. Fuente: Elaboración propia

B. BASE DE DATOS

El motor de base de datos sobre el cual se creó el Schema para Code Health fue SQL Server, en el cual se crearon 4 tablas, de las cuales solo dos tienen relación (**Figura 16**), esto debido a la forma en la que se consulta e inserta la información ya que requiere que se realice en cierto orden que se detalla a continuación:

Figura 16.
Tablas base de datos



Nota. Fuente: Elaboración propia

- Primero, se consulta la tabla de PROYECTOS, filtrando solo los proyectos activos, con estos datos se realiza un consumo sobre la API REST de GitLab, obteniendo principalmente el ID_PROYECTO de cada repositorio consultado. Los proyectos se insertan y modifican actualmente de forma manual por el usuario encargado de programar las ejecuciones del proceso automatizado.

- Segundo, se consulta la tabla SPRINTS, obteniendo solo los sprints o iteraciones activas, estos registros son insertados y modificados actualmente de forma manual por el usuario encargado de programar las ejecuciones de proceso automatizado.

- Tercero, teniendo el rango de fechas del sprint o iteración y el ID_PROYECTO se obtiene a través de la API REST de GitLab el histórico de los commits por proyecto, sprint y rama; posteriormente se inserta en la base de datos dicho histórico en la tabla HIST_COMMIT_RAMA_PROYEC_SPRINT commit por commit.

- Cuarto, teniendo ya en la base de datos el consolidado de commits para el sprint o iteración se procede a consumir nuevamente la API REST de GitLab para obtener el

detalle de cada commit por ID_COMMIT, posteriormente insertando en la tabla DETALLE_COMMIT lo relacionado a dicho a este.

Por tales motivos, la tabla PROYECTOS no tiene ningún campo que se pueda relacionar a otra tabla, además por la definición de las historias de usuario, la idea de Code Health era que el proceso obtuviera la información básica del proyecto a través de la API REST de GitLab, principalmente el id del proyecto en este paso inicial. Los registros de la tabla DETALLE_COMMIT son los últimos que se crean, por lo tanto no es posible crear una relación (constraint o clave foránea) entre la tabla HIST_COMMIT_RAMA_PROYEC_SPRINT y la tabla DETALLE_COMMIT, ya que al crear el registro del histórico el detalle aun no existe.

C. GITLAB API REST

GitLab es un servidor web de control de versiones y desarrollo de software que permite trabajar de forma colaborativa junto con otros servicios adicionales; GitLab además tiene entre sus servicios un conjunto de APIs que como bien lo menciona su sitio de documentación [7] se pueden usar para automatizar sus diferentes funcionalidades.

En Code Health se hizo uso de la API REST, la cual con uso de los diferentes endpoints, instrucciones y claves de valor se pudo obtener la información sobre los proyectos y los repositorios; para obtener los datos requeridos se usaron siete endpoints que están formados inicialmente por la URL del servidor de GitLab, seguido por las instrucciones que apuntan a cada uno de los recursos; para realizar los consumos solo se necesita una cabecera (header) que es un token de autorización que se genera para un usuario que tenga acceso a los repositorios que se desea consultar, este token se configuró de tal forma que tuviera fecha de vencimiento, además todos los endpoint se consultan con el método HTTP GET. A continuación se describen cada uno de los endpoints usados, las palabras resaltadas son tuplas clave-valor reemplazables para realizar los consumos.

- **BUSCAR PROYECTO**
  *URL_SERVER/api/v4/search?scope=projects&search=**name_project_gitlab***

  Descripción: Con este endpoint se busca y se obtiene la información básica del proyecto, se verifica si se tiene acceso al mismo, además se obtiene el **project_id** necesario para realizar los demás consumos.

- **RAMAS PROYECTO**
  *URL_SERVER/api/v4/projects/**project_id** /repository/branches*

  Descripción: Al consultar este endpoint con el id del proyecto (**project_id**) se obtienen los nombres de las ramas (**branch_name**) activas, que posteriormente se utilizaran para obtener los históricos por cada una de estas.

- **HISTÓRICO DE COMMITS**
  *URL_SERVER/api/v4/projects/**project_id**/repository/commits?ref_name=**branch_name**&since=**fecha_inicial**&until=**fecha_final**&all=false&per_page=500&page=1*

  Descripción: Este endpoint retorna todos los commits (**commit_id**) almacenados por proyecto, rama y rango de fechas (sprint o iteración).

- ***OBTENER COMMIT***
  *URL_SERVER/api/v4/projects/**project_id**/repository/commits/**commit_id***

  Descripción: El consumo de este endpoint entrega la información del commit buscado (autor, número de líneas modificadas, etc), también se menciona el commit padre (**parent_ids**) que es el commit anterior al actualmente buscado, necesario para encontrar las modificaciones realizadas en los archivos y las clases.

- **OBTENER DETALLE COMMIT**
  *URL_SERVER/api/v4/projects/**project_id**/repository/compare?from=**parent_ids**&to=**commit_id***

  Descripción: Con este endpoint se obtiene el detalle de las modificaciones realizadas en los archivos del proyecto entre dos commits dados, en este caso entre el padre y el hijo. Específicamente se almacenan los cambios realizados para los escenarios de pruebas plasmados en los archivos .feature.

- **OBTENER ÁRBOL DE ARCHIVOS**
  *URL_SERVER/api/v4/projects/**project_id**/repository/tree?recursive=true&per_page=100&page=**page_number***

  Descripción: La información entregada por este endpoint es un fragmento del árbol de archivos por página (**page_number**) del repositorio asociado al id del proyecto (**project_id**); dentro del árbol de archivos se buscan los archivos de configuración por sus nombres o extensiones y luego se obtienen las rutas de estos archivos dentro del proyecto (**path**) para luego extraer sus contenidos.

- **OBTENER CONTENIDO DE UN ARCHIVO**
  *URL_SERVER/api/v4/projects/**project_id**/repository/files/**path**?ref=**branch_name***

  Descripción: Para obtener el contenido de un archivo de un proyecto dado (**project_id**) se consulta este endpoint con la ruta completa del archivo (**path**) incluyendo su extensión y la rama (**branch_name**) de donde se desea obtener. Luego se revisa su contenido para encontrar posibles credenciales almacenadas.

D.

E. EJECUCIÓN CICLOMÁTICA POR SPRINT Y PROYECTOS

Code Health implementa una estructura de control comúnmente conocida como ciclos, lo que permite generar tantos reportes como se deseen, ya sea que se necesiten métricas para uno o más sprint, y por otro lado uno o más proyectos (**Figura 17**); de esta manera, no hay límites a nivel de cantidad de sprint y proyectos a los cuales se les quiera generar métricas, lo que sí se puede afirmar es que entre más sprints y proyectos hayan activos el tiempo de ejecución será mayor, por lo tanto se recomienda que regularmente se ejecute el proceso automatizado para los sprints que acaban de finalizar. Es importante comentar que para cada sprint ejecutado se crea una carpeta con el nombre de dicho sprint, y allí se almacenan los reportes generados relacionados a este.

Figura 17.
Ciclos por Sprints y Proyectos

Nota. Fuente: Elaboración propia

F. ESQUEMA DE INFORMES

Como se ha mencionado anteriormente Code Health genera tres informes, los cuales tienen información básica de cada proyecto, consolidados y métricas obtenidas dependiendo del informe, a continuación se presentan ejemplos sobre cada uno de ellos y una breve descripción de cada reporte, por temas de confidencialidad se oculta cierta información de las imágenes.

- **Consolidado de commits por Sprint**
  Este informe (**Figura 18**) contiene una recopilación de los commits realizados para un sprint o iteración, por hoja presenta información relevante sobre un proyecto inspeccionado; cada fila contiene el id del commit, la rama donde se realizó, la fecha de creación, su autor, el número de líneas de código nuevas o modificadas, las líneas eliminadas y finalmente la lista de archivos modificados separados por comas; cabe aclarar que lo referente a líneas de código nuevas y eliminadas no son un producto del proceso automatizado, son datos obtenidos que no requieren ser procesados.

Figura 18.
Consolidado commits

sofka

Consolidado de commits por Sprint

NOMBRE PROYECTO	SPRINT	CELULA	URL
	Sprint 72		

ID COMMIT	RAMA	FECHA CREACIÓN	AUTOR	LINEAS DE CODIGO NUEVAS	LINEAS DE CODIGO ELIMINADAS	ARCHIVOS MODIFICADOS
feac545067aba237c77648454e2d2eb4b505b717	develop	2021-10-05T08:37:03.000-05:00	JESPITAL	1	0	
c76f067bf62a5fa8ac4569d1067ec383cde04f2e	develop	2021-10-05T07:56:36.000-05:00	Juan	1	0	
e38c501af34517d0fa4ede233530d6585d39e871	develop	2021-10-04T19:34:29.000-05:00	JESPITAL	141	40	[build.gradle, src/main/resourc
ff253209a8a880f9366be3413dfde31731de1f1f	develop	2021-10-04T19:33:25.000-05:00	Juan	0	0	[]
61bc7c6c3f3152d3bc568753a343a591784a5d57	develop	2021-10-04T19:33:05.000-05:00	Juan	141	40	[build.gradle, src/main/resourc
f4735cc3c0ad1665f7623bbda5d5e63c6f142928	develop	2021-10-04T18:35:36.000-05:00	JESPITAL	815	39	[build.gradle, src/main/resourc
61f7ac0a4da26e3f4c00913f51bb856b4e24cb85	develop	2021-10-04T18:15:31.000-05:00	Juan	801	23	[build.gradle, src/main/resourc
88d959f37c759b4aa4b87943d73ad1b486844e56	develop	2021-10-01T16:28:57.000-05:00	Juan	14	16	
b5a7603894c703397b6d31223cdc46fe7fbb187d	develop	2021-10-01T15:15:12.000-05:00	JESPITAL	91	33	
7c4161ac14ccc4dc20ee94a3e8403c9c21e2eca9	develop	2021-10-01T15:14:07.000-05:00	Juan	91	33	
feac545067aba237c77648454e2d2eb4b505b717	master	2021-10-05T08:37:03.000-05:00	JESPITAL	1	0	
c76f067bf62a5fa8ac4569d1067ec383cde04f2e	master	2021-10-05T07:56:36.000-05:00	Juan	1	0	
e38c501af34517d0fa4ede233530d6585d39e871	master	2021-10-04T19:34:29.000-05:00	JESPITAL	141	40	[build.gradle, src/main/resourc
ff253209a8a880f9366be3413dfde31731de1f1f	master	2021-10-04T19:33:25.000-05:00	Juan	0	0	[]
61bc7c6c3f3152d3bc568753a343a591784a5d57	master	2021-10-04T19:33:05.000-05:00	Juan	141	40	[build.gradle, src/main/resourc
f4735cc3c0ad1665f7623bbda5d5e63c6f142928	master	2021-10-04T18:35:36.000-05:00	JESPITAL	815	39	[build.gradle, src/main/resourc
61f7ac0a4da26e3f4c00913f51bb856b4e24cb85	master	2021-10-04T18:15:31.000-05:00	Juan	801	23	[build.gradle, src/main/resourc
88d959f37c759b4aa4b87943d73ad1b486844e56	master	2021-10-01T16:28:57.000-05:00	Juan	14	16	
b5a7603894c703397b6d31223cdc46fe7fbb187d	master	2021-10-01T15:15:12.000-05:00	JESPITAL	91	33	
7c4161ac14ccc4dc20ee94a3e8403c9c21e2eca9	master	2021-10-01T15:14:07.000-05:00	Juan	91	33	

template

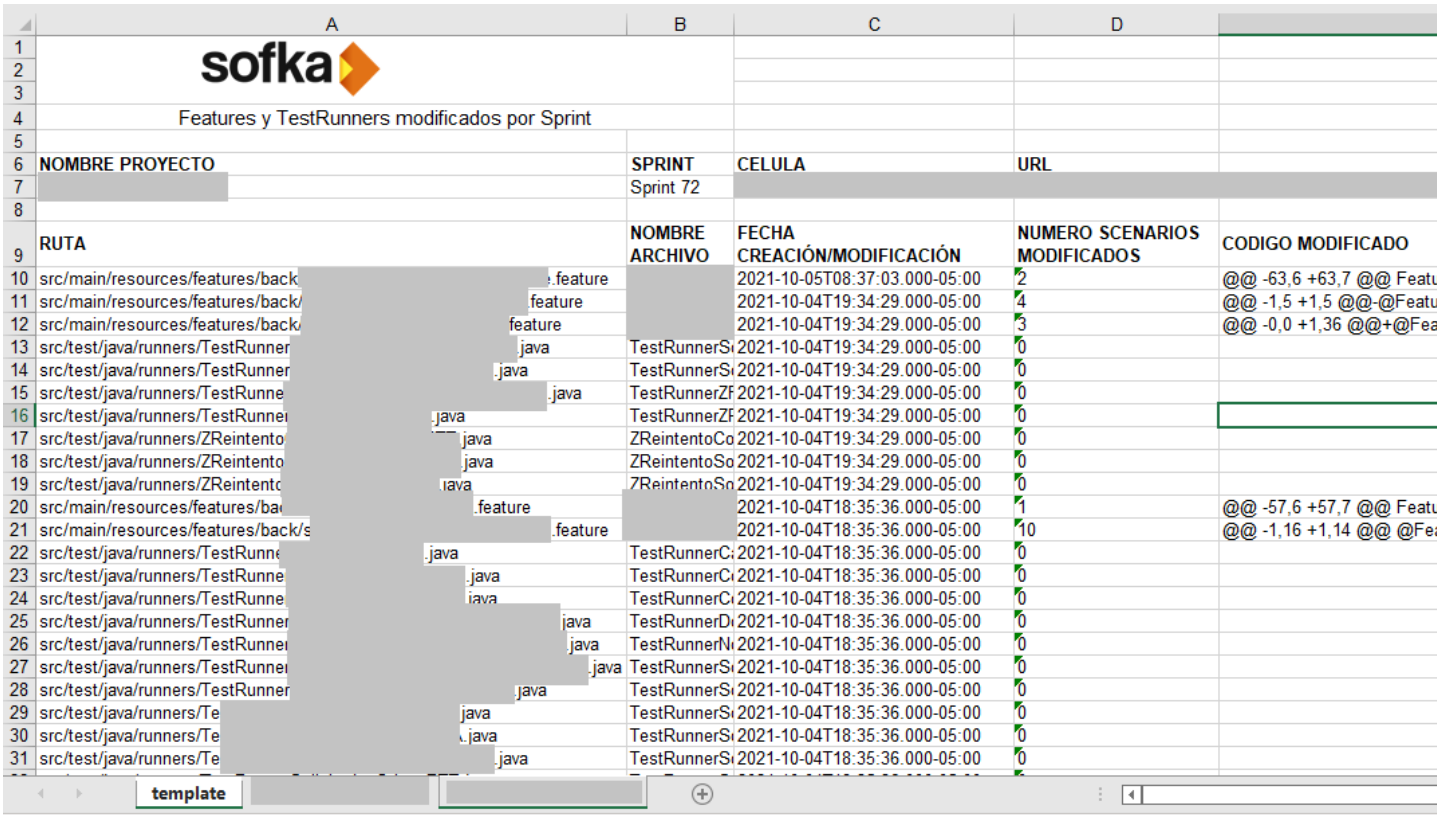
Nota. Fuente: Elaboración propia

- **Features y TestRunners modificados por Sprint**
  Este informe contiene el listado de archivos .feature y testrunners modificados en el sprint (**Figura 19**), cada fila muestra la ruta completa del archivo, el nombre del archivo, la fecha de creación o modificación, el número de escenarios de pruebas modificados y finalmente el segmento de código que se modificó (solo para los

features), en dicho segmento de código el símbolo “+” representa la línea o líneas adicionadas, el símbolo “-” representa la línea o líneas eliminadas (Figura 20. Feature modificado).

Figura 19.

Features y TestRunners



sofka

Features y TestRunners modificados por Sprint

NOMBRE PROYECTO	SPRINT	CELULA	URL
	Sprint 72		

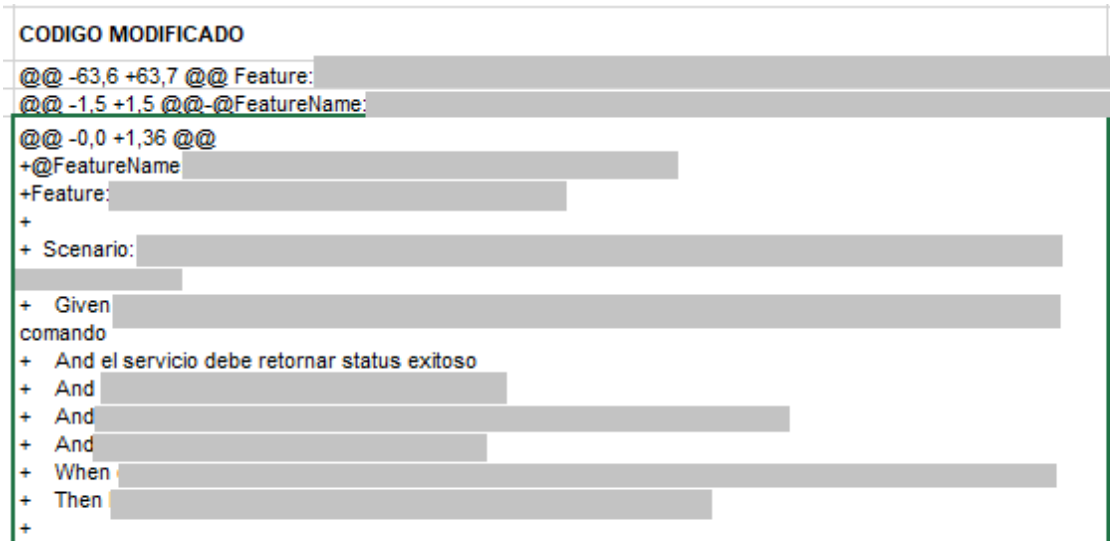
RUTA	NOMBRE ARCHIVO	FECHA CREACIÓN/MODIFICACIÓN	NUMERO SCENARIOS MODIFICADOS	CODIGO MODIFICADO
src/main/resources/features/back .feature		2021-10-05T08:37:03.000-05:00	2	@@ -63,6 +63,7 @@ Featu
src/main/resources/features/back/ feature		2021-10-04T19:34:29.000-05:00	4	@@ -1,5 +1,5 @@-@Featu
src/main/resources/features/back/ feature		2021-10-04T19:34:29.000-05:00	3	@@ -0,0 +1,36 @@+@Fea
src/test/java/runners/TestRunner .java	TestRunnerS	2021-10-04T19:34:29.000-05:00	0	
src/test/java/runners/TestRunner .java	TestRunnerS	2021-10-04T19:34:29.000-05:00	0	
src/test/java/runners/TestRunne .java	TestRunnerZF	2021-10-04T19:34:29.000-05:00	0	
src/test/java/runners/TestRunne .java	TestRunnerZF	2021-10-04T19:34:29.000-05:00	0	
src/test/java/runners/ZReintento java	ZReintentoCo	2021-10-04T19:34:29.000-05:00	0	
src/test/java/runners/ZReintento java	ZReintentoSo	2021-10-04T19:34:29.000-05:00	0	
src/test/java/runners/ZReintento java	ZReintentoSo	2021-10-04T19:34:29.000-05:00	0	
src/main/resources/features/ba .feature		2021-10-04T18:35:36.000-05:00	1	@@ -57,6 +57,7 @@ Featu
src/main/resources/features/back/s .feature		2021-10-04T18:35:36.000-05:00	10	@@ -1,16 +1,14 @@ @Fea
src/test/java/runners/TestRunne .java	TestRunnerC	2021-10-04T18:35:36.000-05:00	0	
src/test/java/runners/TestRunne .java	TestRunnerC	2021-10-04T18:35:36.000-05:00	0	
src/test/java/runners/TestRunne java	TestRunnerC	2021-10-04T18:35:36.000-05:00	0	
src/test/java/runners/TestRunner java	TestRunnerD	2021-10-04T18:35:36.000-05:00	0	
src/test/java/runners/TestRunner .java	TestRunnerN	2021-10-04T18:35:36.000-05:00	0	
src/test/java/runners/TestRunner .java	TestRunnerS	2021-10-04T18:35:36.000-05:00	0	
src/test/java/runners/TestRunner .java	TestRunnerS	2021-10-04T18:35:36.000-05:00	0	
src/test/java/runners/Te java	TestRunnerS	2021-10-04T18:35:36.000-05:00	0	
src/test/java/runners/Te .java	TestRunnerS	2021-10-04T18:35:36.000-05:00	0	
src/test/java/runners/Te .java	TestRunnerS	2021-10-04T18:35:36.000-05:00	0	

template

Nota. Fuente: Elaboración propia

Figura 20.

Feature modificado



CODIGO MODIFICADO
@@ -63,6 +63,7 @@ Feature:
@@ -1,5 +1,5 @@-@FeatureName
@@ -0,0 +1,36 @@ +@FeatureName +Feature: + + Scenario: + Given comando + And el servicio debe retornar status exitoso + And + And + And + When + Then +

Nota. Fuente: elaboración propia

- **Alerta posibles credenciales almacenadas**
  Este informe contiene el consolidado de posibles credenciales almacenadas en el código (**Figura 21**), específicamente en archivos de configuración, cada hoja contiene el resultado de la inspección por proyecto; por cada fila se tiene la ruta y nombre del archivo, la rama y la alerta (posible credencial); este reporte es muy importante ya que permite verificar el cumplimiento de las políticas de seguridad y evitar brechas. Cabe aclarar que las alertas presentadas pueden no ser credenciales, pero cumplen con las condiciones de una credencial segura, debido al proceso de ingeniería inversa que se le realiza con la clase utilitaria "PasswordValidator".

Figura 21.
Alerta posibles credenciales almacenadas

sofka

Alerta posibles credenciales almacenadas

NOMBRE PROYECTO	SPRINT	CELULA	URL
	Sprint 72		

RUTA	RAMA	NOMBRE ARCHIVO	ALERTA
build.gradle	develop	build.gradle	
build.gradle	develop	build.gradle	
build.gradle	develop	build.gradle	
src/main/resources/properties/configuration.properties	develop	configuration.properties	SqlServer.host=
src/main/resources/properties/configuration.properties	develop	configuration.properties	SqlServer.port=
src/main/resources/properties/configuration.properties	develop	configuration.properties	SqlServer.databaseName=
src/main/resources/properties/configuration.properties	develop	configuration.properties	SqlServer.password=
src/main/resources/properties/configuration.properties	develop	configuration.properties	
src/main/resources/properties/configuration.properties	develop	configuration.properties	
src/main/resources/properties/configuration.properties	develop	configuration.properties	
src/main/resources/properties/configuration.properties	develop	configuration.properties	Splunk.ip =
src/main/resources/properties/configuration.properties	develop	configuration.properties	Splunk.search.index=
build.gradle	master	build.gradle	
build.gradle	master	build.gradle	
build.gradle	master	build.gradle	
src/main/resources/properties/configuration.properties	master	configuration.properties	SqlServer.host=
src/main/resources/properties/configuration.properties	master	configuration.properties	SqlServer.port=
src/main/resources/properties/configuration.properties	master	configuration.properties	SqlServer.databaseName=
src/main/resources/properties/configuration.properties	master	configuration.properties	SqlServer.password=
src/main/resources/properties/configuration.properties	master	configuration.properties	
src/main/resources/properties/configuration.properties	master	configuration.properties	
src/main/resources/properties/configuration.properties	master	configuration.properties	
src/main/resources/properties/configuration.properties	master	configuration.properties	Splunk.ip =

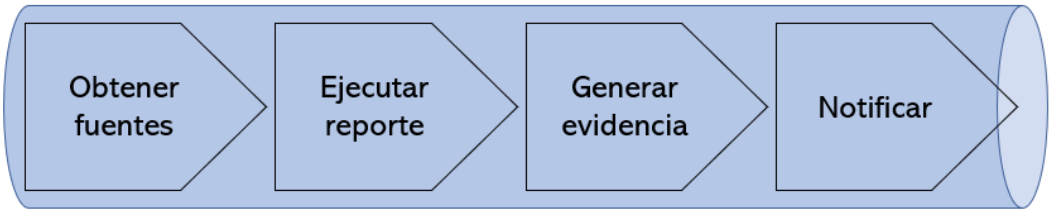
template

Nota. Fuente: Elaboración propia

G. EJECUCIÓN DEL PROCESO AUTOMÁTICO: CODE HEALT

Code Health se puede ejecutar de dos formas, localmente o dentro de un pipeline de Jenkins. El pipeline fue implementado con cuatro stages modificables (**Figura 22**) descritos brevemente a continuación:

Figura 22.
Flujo pipeline



Nota. Fuente: Elaboración propia

- Primero se obtienen las fuentes, es decir, se descarga el proyecto del repositorio, para esto se tiene que dejar dentro del stage credenciales con las cuales se pueda descargar el proyecto y la url del repositorio.

- Segundo, se ejecuta el proceso automatizado de generación de métricas que corre desde runner "TestRunneraGeneracionDeReportes".

- Tercero, se generan las evidencias, específicamente se publican los informes en un bucket de Amazon S3 (Simple Storage Service)[8], que como tal es un servicio de almacenamiento de objetos que permite resguardar la información de forma segura, tener controles de acceso, entre otros.

- Cuarto, se procede a enviar por correo electrónico los informes a las personas interesadas.

VI. CONCLUSIONES

- El proceso automático de generación de métricas de salud del codigo Code Health fue desarrollado basándose en el framework de automatización de pruebas de Sofka Technologies, lo cual facilita su mantenimiento y modificación, ya que su arquitectura es similar a la propuesta por el área de Aseguramiento de la Calidad para el desarrollo de proyectos de pruebas automatizadas.

- Code Health fue desarrollado de tal forma en que puede generar reportes para uno o más proyectos activos y uno o más sprints activos que se hayan almacenado en base de datos, debido a que su proceso de generación de métricas se realiza de forma ciclomática, permitiendo no tener un límite para su ejecución; sólo con crear los registros en base de datos es suficiente para que se puedan realizar ejecuciones, por lo tanto se puede mencionar que en cierto sentido es escalable y configurable.

- Code Health produce tres tipos de informes, uno sobre la frecuencia de commits relacionados por proyecto y sprint, un segundo reporte que plasma los archivos .feature y testrunners modificados por proyecto y sprint, y por último un reporte de alertas de posibles credenciales almacenadas por proyecto. Estos reportes son de gran utilidad para mejorar el seguimiento sobre los proyectos.

- Code Health permite nutrir las listas referentes a nombres de archivos de configuración y extensiones en los cuales se buscan las alertas por posibles credenciales almacenadas; además de las excepciones que ayudan a identificar qué cadenas de texto predefinidos no son credenciales. Estas propiedades hacen parte de la configuración del proceso automatizado.

VII. REFERENCIAS BIBLIOGRÁFICAS

[1] Beck, K., et al. (2001). The Agile Manifesto. Agile Alliance [Online]. Available: http://agilemanifesto.org/

[2] TRY QA. What is V-model- advantages, disadvantages and when to use it? [Online]. Available: http://tryqa.com/

[3] International Software Testing Qualifications Board. (2018). Programa de Estudio Foundation Level 2018 [Archivo PDF]. Available: https://brightest.org/es/recursos/

[4] C. Gomez, (2021, julio, 7). Pruebas dinámicas vs Pruebas estáticas [Online]. Available: https://www.diariodeqa.com

[5] T. Hamilton, (2021, agosto, 29). Automation Testing Tutorial: What is Automated Testing? [Online]. Available: https://www.guru99.com/

[6] writing features - gherkin language [Online]. Available: https://docs.behat.org/

[7] API Docs [Online]. Available: https://docs.gitlab.com/

[8] Amazon S3 [Online]. Available: https://aws.amazon.com/es/