UNIVERSIDAD DE COSTA RICA
SISTEMA DE ESTUDIOS DE POSGRADO

EVALUACIÓN DE LA EFECTIVIDAD DEL PROCESO DE REMOCIÓN DE DEFECTOS EN UNA ORGANIZACIÓN ÁGIL QUE UTILIZA SCRUM

Trabajo Final de Investigación Aplicada sometido a la consideración de la Comisión del Programa de Estudios de Posgrado en Computación e Informática para optar al grado y título de Maestría Profesional en Computación e Informática

JASON BOLAÑOS BARRANTES

Ciudad Universitaria Rodrigo Facio, Costa Rica

2021

Dedicatoria

Le dedico este trabajo a mi familia, que siempre me ha apoyado para seguir adelante y, sin importar las circunstancias, me alienta y da motivación para no renunciar ante las adversidades que presenta la vida. Ellos han sido el pilar fundamental en todos mis esfuerzos para ser un mejor profesional y una mejor persona cada día.

A mis padres, a mi esposa, a mis hermanas.

Agradecimientos

Agradezco al profesor Marcelo Jenkins Coronas, quien me ha guiado en el desarrollo de este trabajo. Además de su apoyo, me ha brindado importantes recomendaciones y críticas constructivas que han motivado la realización de esta investigación.

A su vez quiero agradecer a los profesores Alexandra Martínez Porras y Christian Quesada López por haberme brindado su apoyo como miembros del comité, a través de sus comentarios y recomendaciones.

Le agradezco a la empresa que me contrató como desarrollador con sede en Costa Rica, por darme la oportunidad de trabajar en una organización norteamericana y así obtener la información necesaria para el desarrollo de este documento.

Por último, le agradezco a mis compañeros de trabajo que han contribuido en la realización de esta investigación, al reportar su progreso en las herramientas que han generado la información requerida. También por colaborar validando la veracidad de los resultados expuestos en este trabajo.

"Este trabajo final de investigación aplicada fue aceptado por la Comisión del Programa de Estudios de Posgrado en Computación e Informática de la Universidad de Costa Rica, como requisito parcial para optar al grado y título en Maestría Profesional en Computación e Informática."

ALLAN FRANCISCO BERROCAL ROJAS (FIRMA)
PERSONA FISICA, CPF-02-0553-0149.
Fecha declarada: 07/12/2021 08:13:33 PM
Esta representación visual no es fuente
de confianza. Valide siempre la firma.

M.Sc Allan Francisco Berrocal Rojas
Representante del Decano del Sistema de Estudios de Posgrado

MARCELO JENKINS CORONAS (FIRMA)
Firmado digitalmente por MARCELO JENKINS CORONAS (FIRMA)
Fecha: 2021.12.08 08:04:21 -06'00'

Dr. Marcelo Jenkins Coronas
Profesor Guía

ALEXANDRA MARIA MARTINEZ PORRAS (FIRMA)
Digitally signed by ALEXANDRA MARIA MARTINEZ PORRAS (FIRMA)
Date: 2021.12.08 10:56:52 -06'00'

Dra. Alexandra Martínez Porras
Lectora

CRISTIAN QUESADA LOPEZ (FIRMA)
Firmado digitalmente por CRISTIAN QUESADA LOPEZ (FIRMA)
Fecha: 2021.12.08 07:15:35 -06'00'

Dr. Christian Quesada López
Lector

GABRIELA MARIN RAVENTOS (FIRMA)
Firmado digitalmente por GABRIELA MARIN RAVENTOS (FIRMA)
Fecha: 2021.12.08 11:04:31 -06'00'

Dra. Gabriela Marín Raventós
Directora Programa de Posgrado en Computación e Informática

JASON BOLAÑOS BARRANTES (FIRMA)
PERSONA FISICA, CPF-01-1445-0979.
Fecha declarada: 07/12/2021 01:25:28 PM
Esta representación visual no es fuente
de confianza. Valide siempre la firma.

Jason Bolaños Barrantes
Sustentante

Índice

01.Dedicatoria ... ii
02.Agradecimientos ... iii
03. Hoja de Aprobación ... iv
04.Indice ... v
05.Resumen ... vii
06.Lista de Tablas ... viii
07.Lista de Figuras ... ix
1.Introducción
1.1.Problema ... 1
1.2.Objetivos ... 2
1.3.Preguntas de investigación ... 2
1.4. Descripción de la organización bajo estudio ... 3
1.5.Justificación ... 4
1.6. Estructura del documento ... 5
2.Marco teórico
2.1.Métrica *Defect Removal Efficiency* ... 6
2.2.SCRUM ... 7
2.3 Control estadístico de procesos ... 10
2.3.1 Gráficos de control ... 11
2.3.1.1Gráfico de control XmR ... 12
2.3.2 Capacidad del proceso ... 13
2.4.Pruebas de regresión ... 14
2.5.JIRA ... 15
2.6.GitHub ... 16
2.6.1 Sistema de control de versiones ... 16
2.6.1.1 Sistema distribuido de control de versiones ... 17
2.6.2 Git ... 18
2.6.3 Pull request ... 19
3.Trabajo relacionado
3.1.Medodologías secuenciales ... 20
3.2.Metodologías Ágiles ... 22
3.3.Contribuciones ... 24
3.4.Diferencia con trabajo realizado anteriormente ... 23
4. Metodología
4.1 Diseño del caso de estudio ... 27
4.1.1 Proceso de conteo ... 27
4.2 Recolección de los datos ... 30
4.3 Análisis de los datos ... 32
4.4 Reporte de resultados ... 32

4.5 Amenazas a la validez 33
5. Resultados
5.1 Descripción de resultados 34
5.2. Análisis de estabilidad y capacidad del Proceso de Remoción de Defectos 36
5.2.1. Remoción de puntos inestables y Actualización de Gráfico 37
5.3. Revisión de causas asignables y propuesta de mejora 38
5.3.1 Causas asignables del *sprint* 106 38
5.3.2 Causas asignables con respecto al bajo porcentaje del DRE 40
5.3.2.1. Mala implementación de código 42
5.3.2.2. Corrección Defectuosa 48
5.3.2.2. Implementación Incompleta 51
5.3.2.4. Problemas Posteriores a Actualizacion 52
5.3.2.5. Definición de requerimientos incompleta 52
5.3.3 Plan de mejora ante las causas asignables identificadas 54
6. Conclusiones 59
6.1. Trabajo Futuro 61
I. Bibliografía 63
II. Anexos 65
Anexo 1 Defectos encontrados dentro de las categorias *Regression-Bug* y *Bug* 65
Anexo 2 Defectos encontrados dentro de las categoria *Story-Bug* 73

Resumen

El proceso de remoción de defectos en proyectos de desarrollo de software se puede analizar mediante la métrica de calidad llamada “eficiencia de remoción de defectos”, o DRE (por sus siglas en inglés: *Defect Removal Efficiency*). El uso de esta métrica se ha reportado mayoritariamente en contextos que siguen modelos de desarrollo tradicionales como cascada. Más recientemente, algunos investigadores se han visto motivados a aplicar dicha métrica en contextos que usan metodologías ágiles. Esta investigación busca complementar el estudio realizado por Castro y Jenkins [6], quienes propusieron calcular el DRE con base en un conteo acumulado a lo largo de los *sprints*, bajo un contexto ágil. El objetivo consiste en evaluar la efectividad en remover defectos de una organización que utiliza la metodología ágil SCRUM.

Para desarrollar esta investigación, se siguió la metodología de caso de estudio (*case study*), la cual dentro de su diseño incluyó el diseño del proceso de conteo, la delimitación de los datos a recolectar, y el tipo de análisis a realizar. También se consideraron las amenazas a la validez del estudio y la forma en que se podrían mitigar.

Una vez realizado el proceso de conteo y calculada la métrica, se analizó si el proceso de remoción de defectos era estable y capaz (mediante el gráfico de control XmR). Los resultados indicaron que el proceso no era estable ya que al aplicar las cuatro pruebas de estabilidad planteadas por Floriac [8], se encontró un punto inestable, que se encontraba fuera de los limites de desviación. Por otro lado, se determinó que el proceso no era capaz, según el valor de Caper Jones obtenido. Por esta razón, se elaboró un plan de mejora como una propuesta a la organización bajo estudio.

Con base en los resultados obtenidos, se concluye que el proceso actual de remoción de defectos de la organización puede mejorarse, ya que al haber encontrado el *sprint* de origen, se determina que algunos *sprints* tuvieron mayor impacto en la calidad del producto. Encontrar el *sprint* de origen no fue fácil, pero se logró gracias a la ayuda de clasificaciones como el tipo de origen y la resolución del defecto.

Finalmente se ofrecen algunas alternativas de futuras líneas de investigación. En primer lugar, se puede tratar de automatizar el proceso de análisis para encontrar el *sprint* de origen, pues en este estudio se hizo de forma manual, pero se podrían usar herramientas de procesamiento de lenguaje natural en conjunto con la herramienta de versionamiento de código *GitHub* y la herramienta *JIRA* para automatizar dicha tarea. En segundo lugar, se propone darle seguimiento a la organización bajo estudio, analizando nuevos datos después de aplicar las sugerencias indicadas en el plan de mejora. Finalmente, se propone usar otras métricas adicionales como la efectividad de los casos de prueba, para analizar su posible correlación con los resultados de la métrica de DRE.

Lista de Tablas

Tabla 1. Defectos encontrados y cálculo del DRE por sprint. 35
Tabla 2: Cantidad de defectos por sprint según el tipo de origen. 41

Lista de Figuras

Figura 1. Ejemplo de un gráfico de control y sus componentes. 11
Figura 2. Ejemplo de histograma para determinar si el proceso es capaz. 13
Figura 3. Ejemplo de un workflow en JIRA (Tomado de Atalassian). 16
Figura 4. Visualización de un pull request en GitHub. 19
Figura 5. Resumen de metodología usada en la investigación. 26
Figura 6. Resumen del proceso de conteo. Elaboración propia. 30
Figura 7. Gráfico de control XmR del DRE por sprint. 36
Figura 8. Gráfico de control XmR del porcentaje de DRE por sprint removiendo puntos inestables. 38
Figura 9. Plantilla para un reporte de revisión formal (tomado de D. Galin, Software Quality Assurance, From the theory to the implementation). 56

UNIVERSIDAD DE COSTA RICA

SEP Sistema de Estudios de Posgrado

Autorización para digitalización y comunicación pública de Trabajos Finales de Graduación del Sistema de Estudios de Posgrado en el Repositorio Institucional de la Universidad de Costa Rica.

Yo, Jason Bolaños Barrantes, con cédula de identidad 1-1445-0979, en mi condición de autor del TFG titulado EVALUACIÓN DE LA EFECTIVIDAD DEL PROCESO DE REMOCIÓN DE DEFECTOS EN UNA ORGANIZACIÓN ÁGIL QUE UTILIZA SCRUM.

Autorizo a la Universidad de Costa Rica para digitalizar y hacer divulgación pública de forma gratuita de dicho TFG a través del Repositorio Institucional u otro medio electrónico, para ser puesto a disposición del público según lo que establezca el Sistema de Estudios de Posgrado. **SI [X] NO [ ]***

***En caso de la negativa favor indicar el tiempo de restricción: _____ año (s).**

Este Trabajo Final de Graduación será publicado en formato PDF, o en el formato que en el momento se establezca, de tal forma que el acceso al mismo sea libre, con el fin de permitir la consulta e impresión, pero no su modificación.

Manifiesto que mi Trabajo Final de Graduación fue debidamente subido al sistema digital Kerwá y su contenido corresponde al documento original que sirvió para la obtención de mi título, y que su información no infringe ni violenta ningún derecho a terceros. El TFG además cuenta con el visto bueno de mi Director (a) de Tesis o Tutor (a) y cumplió con lo establecido en la revisión del Formato por parte del Sistema de Estudios de Posgrado.

JASON BOLAÑOS BARRANTES (FIRMA)
PERSONA FISICA, CPF-01-1445-0979.
Fecha declarada: 07/12/2021 01:26:16 PM
Esta representación visual no es fuente
de confianza. Valide siempre la firma.

FIRMA ESTUDIANTE

Nota: El presente documento constituye una declaración jurada, cuyos alcances aseguran a la Universidad, que su contenido sea tomado como cierto. Su importancia radica en que permite abreviar procedimientos administrativos, y al mismo tiempo genera una responsabilidad legal para que quien declare contrario a la verdad de lo que manifiesta, puede como consecuencia, enfrentar un proceso penal por delito de perjurio, tipificado en el artículo 318 de nuestro Código Penal. Lo anterior implica que el estudiante se vea forzado a realizar su mayor esfuerzo para que no sólo incluya información veraz en la Licencia de Publicación, sino que también realice diligentemente la gestión de subir el documento correcto en la plataforma digital Kerwá.

1. Introducción

1.1 Problema

Es bien sabido que la calidad de un producto de software es importante para lograr la satisfacción del cliente. La calidad del software se puede entender como la conformidad con respecto a los requerimientos [1]. En algunas ocasiones, cuando se libera el producto al cliente, pueden encontrarse no-conformidades o defectos. Estas pueden deberse a que el producto no se ajusta a lo requerido, o a que tiene un comportamiento no esperado a causa de la implementación de nuevas funcionalidades que afectan las existentes.

El proceso de remoción de defectos en proyectos de desarrollo de software se puede analizar mediante la métrica de calidad conocida como DRE (por sus siglas en inglés: *Defect Removal Efficiency*) o "eficiencia de remoción de defectos", la cual fue desarrollada por IBM en los años 70, con el fin de evaluar la efectividad de las inspecciones de software en comparación con las pruebas de software [2].

La aplicación de la métrica DRE se ha llevado a cabo principalmente en contextos donde se emplean modelos de desarrollo de software tradicionales, como cascada. En estudios previos [3] [4], se ha realizado un análisis de la capacidad para remover defectos en cada una de las fases que componen el proceso de desarrollo de software bajo la metodología cascada.

Son pocos los trabajos de investigación que utilizan la métrica DRE en proyectos que siguen metodologías agiles. Un ejemplo de estos trabajos es el estudio de Kumaresh et al. [5], quienes calcularon la métrica DRE para un proyecto de mantenimiento de software. Otro ejemplo es el estudio de Castro et al. [6], donde se implementó la misma métrica, pero para un proyecto de desarrollo de software. Ambos trabajos fueron analizados con el fin de identificar cómo recolectaban sus datos y calculaban la métrica en el contexto de metodologías ágiles, y así definir nuestro esquema de recolección de datos para el presente estudio.

En este trabajo de investigación se diseñó e implementó un proceso de recolección y cálculo de la métrica DRE para un proyecto de una organización que utiliza la metodología de desarrollo SCRUM, con el fin de evaluar la efectividad de su proceso de remoción de defectos. También se determina la estabilidad y capacidad del proceso de remoción de defectos.

1.2 Objetivos

1.2.1 Objetivo general

Evaluar la efectividad del proceso de remoción de defectos en un proyecto de desarrollo de software en una organización ágil que utiliza SCRUM.

1.2.2 Objetivos específicos

1. Diseñar un proceso de recolección y cálculo de la métrica de Efectividad de Remoción de Defectos en el contexto de un proyecto de la organización bajo estudio.
2. Implementar el proceso de recolección y cálculo de la métrica de Efectividad de Remoción de Defectos en un proyecto de la organización bajo estudio.
3. Evaluar la estabilidad y capacidad del proceso de remoción de defectos en un proyecto de la organización bajo estudio.

1.3 Preguntas de investigación

Para alcanzar los objetivos específicos de esta investigación, se plantearon las siguientes preguntas de investigación:

RQ1. ¿Cómo se recolectan los defectos durante el ciclo de desarrollo del software y después de la liberación del producto en un proyecto ágil?

RQ2. ¿Cuál es el valor de la métrica de Efectividad de Remoción de Defectos por cada iteración y cuántas iteraciones deben ser contadas para la evaluación de la estabilidad del proceso?

RQ3. ¿Cómo se corrobora que el proceso es estadísticamente estable y capaz?

1.4 Descripción de la organización bajo estudio

La organización bajo estudio inició con el desarrollo de su solución desde el año 2015. Su modelo de negocios se basa en la telemedicina, la organización tiene a su disposición una serie de especialistas en distintas ramas de la medicina, encargados de atender consultas electrónicas (o *eConsults*) provenientes de sus clientes los cuales van desde clínicas en condados hasta hospitales. Dicha interacción se lleva a cabo a través de la solución desarrollada. Con el pasar del tiempo algunos de sus clientes han solicitado personalizaciones que han sido integradas en el software como pate del trabajo del equipo de desarrollo.

Sus Oficinas principales se encuentran en San Diego, California la cual se encuentra conformada por 40 personas en donde se encuentran los directivos, departamento de ventas, soporte, inteligencia de negocios, líderes de producto y algunos desarrolladores. Por otro lado, sub contrata los servicios de desarrollo de una empresa de *outsourcing* norteamericana con sede en Costa Rica, añadiendo a 10 personas a su fuerza laboral.

Actualmente el equipo de desarrollo se encuentra conformado por un líder de entrega y tres lideres técnicos que también fungen como desarrolladores. Cada líder técnico tiene a su cargo dos desarrolladores y un ingeniero de calidad, sumando un total de 13 miembros. Algunos de los desarrolladores han trabajado en la organización desde el inicio del proyecto, por lo que poseen de 5 a 6 años de experiencia, mientras que otros se integraron hace 2 años.

El equipo se rige bajo los principios de SCRUM. Los desarrolladores definen notas de implementación con el fin de realizar una estimación de las historias de usuario. Cuando se completa el código fuente de la solución, se hace una revisión de

pares. Para la corrección de defectos, el desarrollador da una estimación de puntos de historia aproximada que se basa en su criterio. Esto se debe a que la complejidad del defecto puede cambiar según lo que dicho desarrollador identifique como problema y lo que implique resolverlo. Al finalizar con la corrección, se debe realizar una revisión de pares del código fuente y se debe proporcionar una explicación de la causa raíz que originó el defecto.

Actualmente la organización bajo estudio cuenta con algunas métricas para medir su desempeño con respecto a la entrega del producto. Por ejemplo, se determina la cantidad de puntos de historia (*story points*) que se entrega por mes, así como el número de entregas (*releases*) que se realiza por mes. La primera métrica determina el esfuerzo mensual, mientras que la segunda indica la frecuencia con la que el equipo de desarrollo envía nuevas funcionalidades a producción.

1.5 Justificación

Este trabajo de investigación extiende el estudio realizado por Castro et al. [6], en el cual se adaptó la métrica DRE para ser utilizada bajo la metodología SCRUM. En dicho estudio, el cálculo de la métrica DRE se realizó mediante un conteo de defectos basado en el *sprint* donde se reportaron los defectos, y no se evaluó la capacidad ni la estabilidad del proceso de remoción de defectos. La presente investigación, en cambio, sí implementa el control estadístico de procesos con la métrica DRE, para evaluar si el proceso de remoción de defectos es estable y capaz. Adicionalmente, esta investigación propone modificar el cálculo de la métrica DRE para considerar el *sprint* donde se originaron los defectos en lugar del *sprint* donde se reportaron.

En el contexto del curso PF-3880 Métricas de Software del programa de Maestría en Computación e Informática de la Universidad de Costa Rica, el autor de esta investigación realizó un estudio piloto donde usó la métrica de DRE para evaluar la calidad del proceso de desarrollo de la organización bajo estudio, e incluso se implementaron técnicas de control estadístico de procesos para dicha métrica. Sin

embargo, uno de los problemas detectados en ese estudio piloto fue que el proceso de conteo de defectos no era adecuado, puesto que solo tomaba en cuenta los datos de defectos obtenidos de la herramienta Jira, sin clasificarlos con base en su origen, por lo que la información analizada podía potencialmente presentar sesgos. De ahí nace la idea de la presente investigación, que busca definir un proceso de conteo de defectos consistente y sistemático, mediante el cual se pueda saber cuándo un defecto debe ser contado o no para efectos de la métrica DRE, evitando así dicho sesgo.

Esta investigación beneficia a la organización bajo estudio al brindarle una forma de calcular la métrica DRE adaptada a la metodología SCRUM que usa la organización, y adicionalmente, al evaluar la estabilidad y capacidad de su proceso de remoción de defectos. Mediante los gráficos de control estadístico, la organización puede determinar si su proceso es estadísticamente estable, y realizar predicciones para futuros desarrollos. Aún cuando el proceso no se encuentra bajo control estadístico, la organización se beneficiará de un análisis de causas asignables, y una propuesta de mejora para el proceso de remoción de defectos. Este trabajo puede ser utilizado como base para otros estudios que involucren el cálculo de la métrica DRE bajo metodologías ágiles, y esperamos que fomente el uso de esta métrica en otras organizaciones.

1.6 Estructura del documento

A continuación se resumen la estructura conceptual del presente documento. El capítulo 2 describe el marco teórico de la investigación. El capítulo 3 resume los principales trabajos relacionados con esta investigación, indicando los aspectos en común y las diferencias. El capítulo 4 detalla la metodología seguida para el desarrollo de esta investigación. El capitulo 5 presenta los resultados obtenidos y su correspondiente análisis. El capitulo 6 muestra las conclusiones de la investigación así como ideas de trabajos futuros que pueden desarrollarse a partir de los hallazgos de esta investigación.

2. Marco teórico

Esta sección define, en primera instancia, la métrica seleccionada así como su origen y utilizad en el desarrollo de productos de software. A continuación, se detalla el concepto de SCRUM, el cual representa la metodología de desarrollo empleada por la organización bajo estudio. Luego, se explican los conceptos relacionados a los gráficos de control y cómo estos fungen como herramientas para la comprobación de la estabilidad y la capacidad de un proceso. Finalmente, se describen las herramientas usadas por la organización de estudio para el desarrollo de la solución de software y, por ende, utilizadas en esta investigación como datos de entrada.

2.1 Métrica de eficiencia de remoción de defectos

La eficiencia de remoción de defectos (DRE) es una métrica de calidad que fue desarrollada por IBM en la década de 1970 para evaluar la efectividad de las inspecciones de software en comparación con las pruebas de software. DRE es considerada como una de las métricas de software más importantes, debido a que es un indicador de la salud del proyecto, indicando qué tan cercano está el equipo de desarrollo de cumplir los requerimientos del cliente [7].

Caper Jones [7] propone la siguiente fórmula para expresar la métrica DRE:

$$DRE = \frac{Defectos\ removidos\ en\ desarrollo}{Defectos\ removidos\ en\ desarrollo + defectos\ encontrados\ por\ el\ cliente} * 100\ \%$$

Según Jones [2], si el valor de esta métrica está por debajo del 85%, el proyecto estará retrasado o estará por encima del presupuesto y el cliente no estará feliz. Por otro lado, si el valor está por encima del 95%, se obtendrán buenos resultados y un cliente satisfecho.

Cabe aclarar que esta métrica requiere que se lleve un conteo acumulado de los defectos encontrados durante el desarrollo. Una vez liberado el producto, en un periodo de 90 días, se debe obtener el acumulado de defectos reportados por el cliente. Con base en ambos acumulados, se puede obtener el valor de la métrica DRE.

DRE es una de las métricas más relevantes en el desarrollo de software debido a que es el eje de los procesos de mejora. Cualquier metodología que no mida y busque mejorar el valor de DRE es ineficiente [2].

Es importante aclarar que algunos autores se refieren a esta métrica como *Defect Removal Effectiveness,* pero en este trabajo de investigación se prefirió denotarla como *Defect Removal Efficiency.*

2.2 SCRUM

SCRUM es un marco de trabajo liviano en el cual las personas, los equipos y las organizaciones, pueden generar productos de valor a través de soluciones adaptativas para problemas complejos [8]. Dentro de este marco de trabajo se pueden emplear varios procesos, métodos y técnicas. SCRUM visibiliza la eficacia de las prácticas de gestión y trabajo, de manera que se puedan hacer mejoras.

El marco de trabajo define la conformación de los equipos SCRUM, sus roles, eventos, artefactos y reglas asociadas. Cada componente sirve a un propósito especifico y es esencial para el éxito de SCUM.

SCRUM se sustenta en el empirismo y en el pensamiento *lean*. El empirismo afirma que el conocimiento procede de la experiencia y de la toma de decisiones basada en observación. Por su parte, el pensamiento *lean* reduce el desperdicio y se enfoca en lo esencial. SCRUM usa un enfoque iterativo e incremental para optimizar la predictibilidad y controlar el riesgo.

Los tres pilares empíricos de SCRUM son:

1. **Transparencia:** Los aspectos significativos del proceso deben ser visibles para aquellos que son responsables del resultado final. Se requiere que dichos aspectos sean definidos por un estándar común de tal forma que los observadores compartan un entendimiento común.
2. **Inspección:** Se debe inspeccionar frecuentemente los artefactos y el progreso hacia un objetivo para detectar variaciones indeseadas.

3. **Adaptación:** Si se determina que uno o mas aspectos de un proceso se desvían de los limites aceptables y que el resultado será inaceptable, el proceso debe ajustarse con el fin de evitar desviaciones mayores.

El equipo SCRUM se compone de un *scrum master*, un dueño de producto, y los desarrolladores. El *scrum master* es el responsable de asegurar que el equipo SCRUM trabaje de acuerdo a los principios, prácticas y reglas de SCRUM. El dueño del producto es el responsable de maximizar el valor del producto y el trabajo del equipo. El equipo de desarrollo se compone de los desarrolladores, quienes entregan un incremento de producto "terminado" que potencialmente se pueda poner en producción al final de cada *sprint*.

En SCRUM existen eventos predefinidos con el fin de crear regularidad y minimizar la necesidad de reuniones no definidas. SCRUM prescribe los siguientes eventos:

1. ***Sprint*:** Es el corazón de SCRUM, es un bloque de tiempo de un mes o menos durante el cual se crea un incremento de producto "terminado" utilizable o parcialmente desplegable. Es más conveniente si la duración de los *sprints* es consistente a lo largo del esfuerzo de desarrollo. Cada nuevo *sprint* comenzará inmediatamente después de la finalización del *sprint* anterior.
2. **Planificación del *sprint*:** tiene como máxima duración 8 horas para un *sprint* de un mes, y responde a las preguntas:
    a. ¿Qué puede entregarse en el incremento resultante del *sprint* que comienza?
    b. ¿Cómo se conseguirá hacer el trabajo necesario para entregar el incremento?
3. **SCRUM diario:** es una reunión con un bloque de tiempo de 15 minutos para que el equipo de desarrollo sincronice sus actividades y cree un plan para las siguientes 24 horas. Esto se lleva a cabo inspeccionando el trabajo avanzado desde el último SCRUM diario, y haciendo una proyección acerca del trabajo que podría completarse antes del siguiente.

4. **Revisión del *sprint*:** Al final del *sprint* se lleva a cabo una revisión para inspeccionar el incremento y adaptar la lista de producto, si fuese necesario. El equipo SCRUM y los interesados discuten acerca de lo que se hizo durante el *sprint*. Basándose en esto y en cualquier cambio a la lista de producto, los asistentes colaboran para determinar las siguientes cosas que podrían hacerse para optimizar el valor. Se restringe a un tiempo máximo de 4 horas para un *sprint* de un mes.
5. **Retrospectiva del *sprint***: tiene lugar después de la Revisión del *sprint* y antes de la siguiente Planificación de *sprint*. Se trata de una reunión restringida con un bloque de tiempo de 3 horas para *sprint* de un mes. El propósito de este evento es evaluar cómo estuvo el último *sprint* en cuanto a personas, relaciones, procesos y herramientas. Además, identifica y ordena los elementos más importantes que salieron bien y las posibles mejoras al proceso, creando un plan para implementar mejoras a la forma coo el equipo SCRUM desempeña su trabajo.

Los artefactos de SCRUM representan trabajo o valor en diversas formas que son útiles para proporcionar transparencia y oportunidades para la inspección y adaptación. Están diseñados para maximizar la transparencia de la información clave, necesaria para asegurar que todos tengan el mismo entendimiento del artefacto. Dentro de los artefactos de SCRUM podemos definir los siguientes:

1. **Lista de producto:** es una lista ordenada de todo lo que podría ser necesario en el producto y es la única fuente de requisitos para cualquier cambio a realizarse en el producto. Enumera todas las características, funcionalidades, requisitos, mejoras y correcciones que constituyen cambios a realizarse sobre el producto para entregas futuras.
2. **Lista de pendientes del *sprint*:** es el conjunto de elementos de la 'Lista de producto' seleccionado para el *sprint*. Es una predicción hecha por el equipo de desarrollo sobre la funcionalidad que formará parte del

próximo incremento, y del trabajo necesario para entregar dicha funcionalidad.

3. **Incremento:** es la suma de todos los elementos de la 'Lista de producto' completados durante un *sprint* y el valor de los incrementos de todos los *sprints* anteriores.

2.3 Control estadístico de procesos

El control estadístico de procesos se lleva a cabo con el fin de verificar si un proceso es estadísticamente estable. Cuando un proceso es estable, las fuentes de variabilidad se deben a causas comunes. Para determinar esto estadísticamente, se utilizan los gráficos de control [9].

Un proceso es inestable si alguna de las siguientes condiciones se cumple (en cuyo caso las pruebas de estabilidad sobre el gráfico de control fallan):

1. Uno de los puntos del gráfico de control se encuentra fuera de los 3 limites de desviación sigma.
2. Hay al menos 2 de 3 puntos consecutivos que se encuentran alejados a más de 2 unidades sigma del límite central, en el mismo lado del gráfico (arriba o abajo).
3. Hay al menos 4 de 5 puntos consecutivos que se encuentran alejados a más de una unidad sigma del límite central, en el mismo lado del gráfico (arriba o abajo).
4. Hay al menos 8 puntos consecutivos del mismo lado distantes de la línea central.

La falla de alguna de las cuatro pruebas de estabilidad se puede atribuir a la presencia de causas asignables, las cuales se deben contraatacar con el fin de alcanzar estabilidad estadística. Un proceso estable es un proceso predecible estadísticamente.

2.3.1 Gráficos de control

El gráfico de control [9] es una herramienta para determinar si un proceso está bajo control estadístico. Inicialmente el gráfico utiliza una serie de observaciones que se recolectan a lo largo del proceso. Estas observaciones pueden ser valores individuales en secuencia, o una serie de valores clasificados por subgrupos. A partir de los valores graficados, se obtienen los valores para delimitar la línea central y los límites a cada extremo.

En un gráfico de control, la línea central generalmente representa el promedio de las observaciones, pero también puede representar otros valores como la mediana o el rango medio. En el caso de los límites, se derivan de una o varias observaciones que representan la variabilidad del proceso mediante diferentes fórmulas, dependiendo del tipo de variable que se grafica.

Los rangos entre la línea central y los límites pueden ser subdivididos en pequeños rangos, expresados en términos de la desviación estándar de la muestra (llamada sigma), los que representan una estimación de la desviación estándar de las observaciones. En la figura 1 se muestra un gráfico de control y sus componentes.

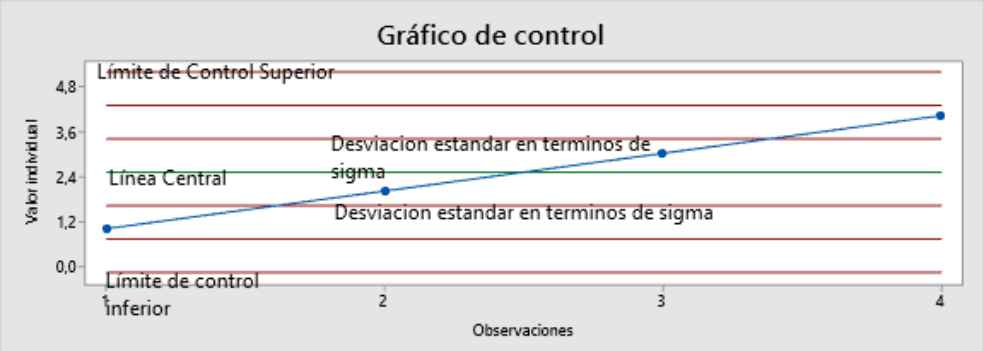

Figura 1. Ejemplo de un gráfico de control y sus componentes.

Los gráficos de control se han usado de forma exitosa en la industria desde la década de 1920 [9]. Entre las razones por las cuales los gráficos de control son populares están:

1. Son una técnica que mejora la productividad.

2. Son eficientes en la prevención de defectos.
3. Evitan ajustes innecesarios en los procesos.
4. Proporcionan información de diagnóstico.
5. Proporcionan información sobre la capacidad del proceso.

2.3.1.1 Gráfico de control XmR

El gráfico de control XmR [9] es un tipo de gráfico que contiene como elementos mediciones que se hacen en un rango de tiempo, y las mediciones son subconjuntos de un elemento (n=1). En este caso, se genera un gráfico que posee todos los elementos individuales (X) y otro gráfico que representa el rango movible (mR).

El cálculo de los límites para estos elementos se hace mediante la variación entre elementos a corto plazo cuando son adyacentes, de esta forma se observa la variación natural del proceso. Para ello se utilizan los dos gráficos: X y mR. Por ejemplo, los puntos de historias de usuario (*story points*) entregados por mes representan 12 observaciones realizadas en el lapso de un año; estos puntos se grafican de forma secuencial para el gráfico X. Posteriormente se obtienen los valores del gráfico mR restando al valor x+1 el valor x de forma secuencial, obteniendo 11 valores que representan la variación entre 2 meses.

Los gráficos XmR suelen usarse cuando es relevante medir la tendencia del desempeño del proceso, en lugar de la varianza entre las mediciones individuales. Este gráfico evalúa el promedio y los cambios que ocurren entre dos puntos, y muestra la tendencia de los datos en el tiempo.

2.3.2 Capacidad del proceso

Se debe reconocer que con el pasar del tiempo los valores medidos de las características de un proceso y producto varían con el tiempo [9]. Cuando un proceso es estable se puede afirmar que los resultados obtenidos obtendrán una media predecible y los valores estarán dentro de rangos predecibles cercanos a dicha media [9].

Una vez que se ha comprobado que un proceso es estable mediante el uso de un gráfico de control, se puede utilizar herramientas como los histogramas para determinar si un proceso es capaz. Esto quiere decir que dicho proceso se encuentra dentro de los parámetros aceptados por el cliente.

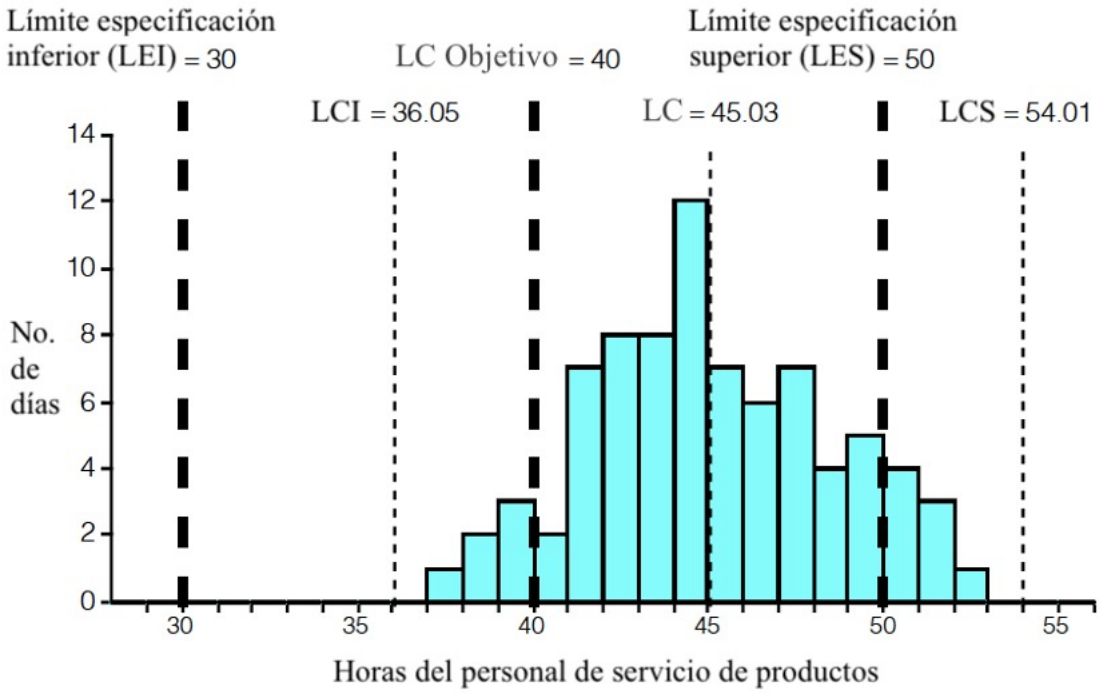

Figura 2. Ejemplo de histograma para determinar si el proceso es capaz.

La variación en el histograma presentado se puede deber a causas fortuitas, es por eso que el limite central y los limites inferior y superior representan como el proceso está definido, como se ejecuta actualmente y lo que es capaz de hacer (conocido como la Voz del Proceso) [9].

El proceso deja de ser capaz cuando los límites del proceso natural caen fuera de los límites esperados por el cliente. En la figura 2 se puede observar que las líneas punteadas representan los limites aceptados por el cliente y su valor medio. Por lo que se deben tomar acciones correctivas de tal forma que la variabilidad de las medidas o su valor medio se desplace (o que ocurran ambos) y así se acerquen a los valores especificados. Una alternativa que puede ser aceptada en algunas situaciones es relajar las mediadas (o cambiar los valores esperados por el cliente) de tal forma que el proceso se adapte a las especificaciones.

Cabe aclarar que cuando un proceso es estable y se ajusta a los requerimientos del cliente este se puede considerar capaz, por lo que la capacidad de un proceso depende tanto de la estabilidad como de la capacidad [9]. Aunque el proceso sea estable existe la posibilidad de que se requiriera mejoras en su capacidad con el fin de satisfacer las presiones competitivas (o necesidades del cliente)[9].

2.4 Pruebas de regresión

Las pruebas de regresión se refieren a un proceso de pruebas que es aplicado después de que un programa se modifica. Involucra hacer pruebas al programa modificado con algunos casos de prueba, con el objetivo de reestablecer la confianza de que el programa funciona con respecto a una especificación acordada [10]. En este trabajo de investigación, se contaron los defectos encontrados por las pruebas de regresión como hallazgos del equipo de desarrollo, debido a que es un esfuerzo extra que se hace con el fin de entregar un producto con menos defectos.

2.5 JIRA

Jira es un conjunto de soluciones de administración de trabajo ágil que potencia la colaboración, y va desde la definición del producto hasta el resultado que se entrega al cliente final [11]. Jira utiliza el concepto de proyecto como una colección de historias de usuario, defectos, y tareas, entre otros. Típicamente representa el trabajo desarrollado para un producto, proyecto o servicio. Los proyectos dentro de JIRA se pueden clasificar en iteraciones basadas en SCRUM, iteraciones basadas en Kanban, y seguimiento de defectos [12].

Dentro de cada proyecto, una organización debe registrar bloques de trabajo. Cada bloque de trabajo -ya sea una historia de usuario, un defecto, o una tarea- se considera un *issue*. Para relacionar cada *issue,* JIRA utiliza series de caracteres alfanuméricos que ofrecen una descripción del trabajo que se está realizando a través de toda la herramienta [13]. Por ejemplo, para el proyecto de la organización bajo estudio, se decidió utilizar la palabra "piloto" o *PILOT* para denotar cada *issue.* En ese caso, cualquier historia de usuario, defecto, o tarea, será denotado con la palabra *PILOT* seguido de un consecutivo numérico.

Es importante que cada *issue* tenga el nivel de detalle suficiente para asegurar que cualquier persona que esté trabajando en él pueda resolverlo lo más pronto posible. La herramienta JIRA permite agregar imágenes, diagramas, archivos, referencias a otros *issues,* y otros detalles para tener una imagen claras del trabajo que se debe hacer [14]. En la organización bajo estudio, se dejan comentarios como parte del seguimiento a un *issue,* donde el equipo de trabajo deja detalles a considerar en cada una de las fases de desarrollo.

JIRA maneja el concepto de flujo de trabajo o *workflow*, que es un conjunto de estados y transiciones por los que un *issue* se mueve en su ciclo de vida y típicamente representa un proceso dentro de la organización. Los *workflows* pueden estar asociados a un proyecto en particular [15]. La figura 3 muestra un ejemplo de un *workflow* genérico en JIRA.

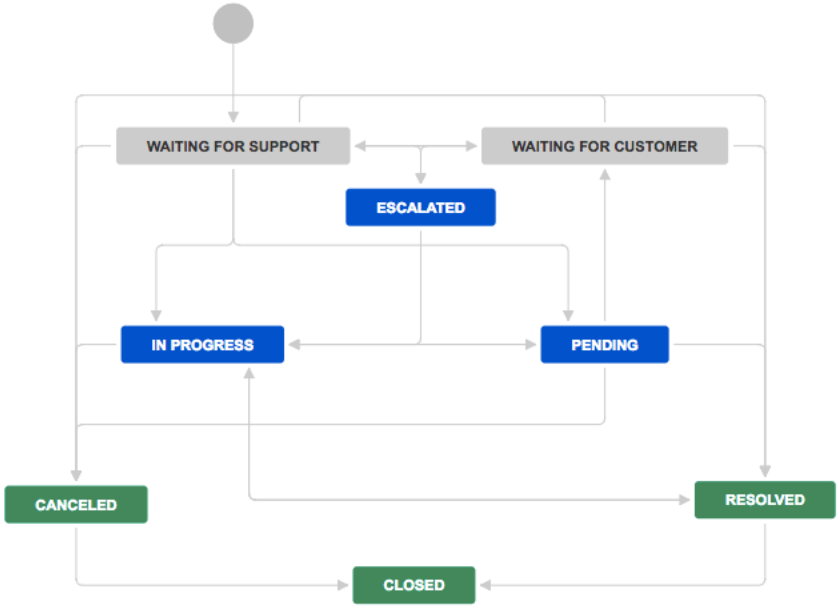

Figura 3. Ejemplo de un *workflow* en JIRA (Tomado de Atalassian).

2.6 GitHub

GitHub es un sitio web y un servicio en la nube que ayuda a los desarrolladores a almacenar y administrar su código, llevando un registro y control de cualquier cambio sobre el código [16]. GitHub se basa en el concepto de un sistema de control de versiones, y en el uso de la herramienta Git, que se definirán a continuación.

2.6.1 Sistema de control de versiones

El sistema de control de versiones es un sistema que maneja el desarrollo de un objeto en evolución, en otras palabras, es un sistema que registra cada cambio hecho por los desarrolladores de software [17].

En el proceso de desarrollo de software es normal que los desarrolladores de software continuamente realicen cambios al código y otros archivos, para agregar, mejorar, o remover una funcionalidad. El sistema de control de versiones ayuda a manejar y organizar cada uno de los cambios de código en sistemas complejos, para así llegar a una versión final del código, simplificando el proceso de desarrollo.

El sistema de control de versiones se subdivide en 2 categorías:

1. Sistema centralizado de control de versiones: es un sistema donde solo hay un repositorio central, de manera que cada usuario que necesite acceso debe estar conectado a este nodo central.
2. Sistema distribuido de control de versiones: consiste en una arquitectura donde cada usuario tiene un repositorio completo, el cual se denomina un "repositorio local". Esta arquitectura permite que cada usuario pueda trabajar completamente desconectado, pero una vez que necesite compartir su código debe conectarse a la red.

Para efectos de este estudio, nos enfocamos en los sistemas distribuidos.

2.6.1.1 Sistema distribuido de control de versiones

En la actualidad se prefieren los sistemas distribuidos de control de versiones, ya que utilizar un "repositorio local" evita la dependencia de un nodo central que es propenso a fallos. En caso de fallo en el repositorio en línea, los desarrolladores pueden seguir trabajando localmente sin ser bloqueados.

Este tipo de sistemas están diseñados para mantener un historial de cambios en cada máquina de forma local. También pueden sincronizar los cambios locales realizados por un usuario en el servidor cuando sea necesario. En caso de existir una falla en el servidor, cada usuario con una copia local es capaz de restaurar el sistema simplemente actualizando el servidor con su copia local.

Los sistemas distribuidos de control de versiones utilizan el concepto de ramificación o *branching,* el cual consiste en permitirle al usuario crear una divergencia de la línea principal de desarrollo, y así

realizar su trabajo sin necesidad de verse involucrado en cambios sobre la línea principal [18].

2.6.2 Git

Git es un sistema distribuido de control de versiones diseñado por la comunidad de desarrolladores de Linux. Desde el 2005 este sistema ha evolucionado y madurado con el fin de ser un sistema fácil de usar, manteniendo sus cualidades iniciales [18].

En comparación a otros sistemas, *Git* hace un manejo diferente de los archivos y sus cambios. Toma una fotografía de cómo lucen los archivos en un momento específico y almacena una referencia. Si no hay cambios a un archivo desde la última fotografía, no se genera una nueva.

Git también se asegura de mantener la integridad de los cambios, es decir, genera una firma basada en los cambios que se realizaron y la asocia a la versión que se genera. De esta manera, no hay posibilidad de que los archivos se puedan corromper sin que *Git* se dé cuenta.

Los archivos dentro de *Git* pueden tener solamente 3 estados:

1. Modificado: significa que los datos han sido cambiados, pero no se han guardado sus cambios en una versión.
2. Escenificado (*staged*): significa que el usuario ha marcado un archivo como modificado y que próximamente será almacenado en una nueva versión.
3. Guardado (*committed*): los cambios del archivo fueron guardados de manera exitosa en una nueva versión.

2.6.3 *Pull Request*

GitHub proporciona una funcionalidad llamada *pull request*, la cual le permite a un desarrollador dar a conocer a otros miembros del equipo los cambios de un *branch* que se han subido al servidor. Una vez disponibles los cambios, se puede hacer una revisión de pares para generar discusión sobre los potenciales cambios y decidir si estos son apropiados para ser incorporados a la línea principal [19].

A diferencia de un *commit*, los *pull requests* muestran todos los cambios que existían entre la línea principal y el *branch* que se quiere combinar. La figura 4 muestra un ejemplo de cómo luce un *pull request*: las líneas de código con fondo verde representan líneas agregadas y las líneas con fondo rojo representan líneas de código removidas.

Figura 4. Visualización de un *pull request* en GitHub.

3. Trabajo relacionado

Esta investigación se basó en estudios que han implementado la métrica DRE para solucionar uno o varios problemas que presentan organizaciones de software. Estos estudios se presentarán según la metodología de desarrollo en la cual se implementaron, comenzando con las metodologías secuenciales y continuando con las metodologías ágiles.

3.1 DRE aplicado a metodologías de desarrollo secuenciales

Leung [3] llevó a cabo una investigación sobre los defectos encontrados en la industria en la cual describe el proceso de remoción de defectos para cada una de las fases del desarrollo del software y se aplica esta métrica en cada una de ellas, con el objetivo de encontrar qué tan efectivo es el proceso en cada fase. Adicionalmente, el autor propone un modelo que consiste en aplicar más de una fase de remoción de defectos en el ciclo de desarrollo, y argumenta que se puede mejorar el proceso de remoción agregando más fases o mejorando de forma equitativa las fases existentes. Además, afirma que el valor de DRE en cada fase no puede exceder el valor total (ya que cada valor por fase compone el valor total del DRE).

La presente investigación difiere del estudio de Leung [3] en que además de implementar la métrica DRE en un contexto ágil, aplicamos el control estadístico sobre el proceso de remoción de defectos para analizar su estabilidad y capacidad.

El estudio de Andrade et al. [20] realizó un análisis de la factibilidad de contratar empresas que se enfocan en crear pruebas de software. Luego implementaron el ciclo Planear, Hacer, Revisar, Actuar (PDCA, por sus siglas en inglés) con el fin de asignar las tareas correspondientes a las empresas externas, de donde extrajeron 12 lecciones aprendidas. Aplicando dichas lecciones aprendidas, encontraron una disminución en el número de fallas y el porcentaje de DRE mejoró. Si bien dicho estudio realiza una aplicación de la métrica DRE, lo hace con el objetivo de evaluar la factibilidad de emplear compañías externas para la creación de pruebas de software, no con el fin que persigue nuestra investigación.

Kumaresh et al. [4] propusieron un modelo de remoción de defectos en el cual se calcula la métrica DRE para cada una de las fases del desarrollo, tomando en cuenta la cantidad de defectos introducidos, la cantidad de defectos corregidos, y la cantidad de defectos que fueron parcialmente corregidos o no se encontraron. Con el fin de mejorar los resultados de dicha métrica, se propone una estrategia para mejorar el proceso, en la que se designa a un equipo encargado de hacer una revisión adicional en cada una de las fases, con el fin de minimizar la cantidad de defectos. Finalmente, los autores argumentan que dicho mecanismo ayuda a mejorar la productividad y a reducir los costos, debido a que una detección temprana ayuda a bajar costos de corrección. A pesar de que la implementación del equipo especial conlleva un costo, este se equilibra con la reducción de defectos y la eficiencia del proceso.

El estudio [20] busca demostrar la factibilidad del uso del ciclo PDCA, mientras que en el estudio [4] solo busca demostrar la factibilidad de llevar a cabo una fase de revisión adicional para minimizar el numero de defectos, en ambos casos se utiliza la métrica de DRE para verificar la mejora obtenida. En contraposición, la presente investigación busca realizar la implementación de la métrica de DRE en complemento con el uso de gráficos de control para determinar la estabilidad y capacidad del proceso de remoción de defectos, en un proyecto de software bajo la metodología de desarrollo SCRUM.

Kuruba [21] propone que la métrica DRE no siempre es un reflejo de la efectividad del equipo para remover defectos. Por ejemplo, el DRE puede estar en un 98% pero el cliente no se encuentra satisfecho con el producto entregado. Esto puede suceder cuando el equipo de desarrollo está encontrando defectos que no son tan críticos en comparación a los que encuentra el cliente. Como no se está considerando la satisfacción del cliente al calcular la efectividad de remoción de defectos críticos, el autor propone un nuevo alcance de la métrica, que le otorgue un peso a cada tipo de defecto. Para los pesos se usa un Proceso de Jerarquía Analítica, que considera factores como los puntos de función y esfuerzo expresado en persona-mes, generando un factor de productividad. El estudio concluye que aplicando este peso en conjunto con el valor del DRE por fase, se logra una mejor retroalimentación puesto que los

valores se correlacionan con la satisfacción del cliente. Esto, a su vez, hace que el equipo se concentre en encontrar defectos que sean significativos para el cliente y ofrece mejoras al proceso de prueba. Sin embargo, las desventajas son la dificultad de computar los datos, y el hecho de que la clasificación de los defectos influye en el proceso de inspección.

La presente investigación se asemeja al trabajo de Kuruba [21] ya que ofrece una variación al uso de la métrica de DRE; sin embargo, difiere en que nosotros buscamos analizar la estabilidad y capacidad del proceso, en lugar de buscar la satisfacción del cliente.

3.2 DRE aplicado a metodologías de desarrollo ágiles

El estudio de Yamato et al. [5] implementó la métrica de DRE en un proceso de mantenimiento de un proyecto que usaba una metodología ágil. Como todos los problemas que reportaba el cliente llegaban al departamento de mantenimiento, hicieron una clasificación de los defectos según la fase de origen. Tomando como base el conteo de defectos propuesto en un trabajo anterior, los autores lograron hacer el cálculo de la métrica, para la cual obtuvieron un valor de 98%.

Tanto el trabajo de Yamato et al. [5] como nuestra investigación comparten la necesidad de conocer el valor del DRE para evaluar la calidad del proceso de remoción de defectos, y ambos se dan en el contexto de una metodología ágil. No obstante, Yamato et al. [5] obtiene el valor de la métrica con un conteo de defectos acumulado en un año, mientras que en esta investigación se realiza un cálculo por cada iteración ágil y además verifica si el proceso se encuentra bajo control estadístico.

Jie Ba y Shujian Wu [22] diseñaron un modelo de predicción dinámico de defectos basado en la dinámica de sistemas. Ellos explican que en un sistema básico de detección de defectos la entrada al sistema corresponde a los defectos inyectados mientras que las salidas corresponden a los defectos que fueron detectados y a los defectos que se escaparon. Su objetivo consiste en realizar una comparación con otros

modelos de predicción como redes bayesianas y el modelo de predicción *COQUALMO*. En su modelo los datos de entrada corresponden a valores empíricos del valor de la métrica de DRE, la métrica de Rango de defectos insertados, la densidad de defectos de un pre-release y finalmente los valores de esfuerzo, tamaño (expresado en líneas de código) y el numero de defectos de 16 proyectos con el fin de obtener la predicción de los defectos insertados y removidos por cada una de las fases del desarrollo. En comparación con la presente investigación este estudio solo utiliza la métrica de DRE como parte del algoritmo para un modelo de predicción. Ya que al proporcionar el valor del DRE como parte de la entrada se infiere que se utiliza de forma inversa para obtener los datos esperados para realizar dicha comparación.

El trabajo de Castro et al. [6], en el cual se basa la presente investigación, aplica la métrica DRE en el contexto de la metodología ágil SCRUM. Los autores indican que dicha métrica solo se ha usado bajo metodologías tradicionales y por ello quieren analizar la factibilidad de usarla bajo una metodología ágil. Luego del proceso de recolección y análisis de datos, los autores encuentran que los valores de DRE no se ajustan al valor de 85% recomendado por Jones [2], debido a que el análisis se hace por iteración individual. Por esta razón, deciden realizar una modificación al cálculo de la métrica DRE para usar el valor acumulado de defectos. Con esta variación, logran obtener resultados que se apegan más al valor esperado de DRE.

3.3 Estudio base

Como se mencionó anteriormente, el presente trabajo se basa en el estudio de Castro y Jenkins [6], el cual vamos a detallar a continuación para luego señalar la contribución de nuestra investigación.

Los datos recolectados en el estudio original de Castro y Jenkins [6] fueron:

1. ***Sprint* donde se encontró el defecto:** periodo de tiempo en el que el defecto fue encontrado, ya sea por el cliente o por el equipo de desarrollo.

2. **Creador del defecto:** la persona que reportó el defecto, es decir, si fue el cliente o un miembro del equipo de desarrollo. Este dato se utiliza con el fin de conocer si el defecto fue encontrado antes o después de la liberación (*release*).
3. **Fuente del defecto:** una de las cuatro fuentes esperadas de un defecto:
    a. Un problema de código
    b. Un problema de diseño
    c. Un error en los requerimientos
    d. Una mala implementación o una corrección defectuosa

Castro y Jenkins [6] realizaron una modificación al cálculo de la métrica DRE, mostrando un valor acumulado por cada *sprint*. Con este cambio obtuvieron un valor más representativo, que se acercaba al valor esperado según Caper Jones [7].

La primera extensión que hace esta investigación con respecto al estudio original, es que recolecta un dato adicional: el *sprint* de origen. Para ello se utilizó la lista de defectos que fueron reportados después de un *release* en el año 2019, donde se agregó un *pull request* que fue elaborado para la solución del problema. Con base en los cambios realizados, se pudo hacer un análisis de juicio experto (basado en el conocimiento del código fuente) que permitió identificar en cuál *sprint* se introdujo el defecto.

La segunda extensión consistió en aplicar control estadístico al proceso de remoción de defectos, utilizando un gráfico de control XmR con secuencias de datos de una sola variable en un periodo de tiempo (en este caso, cada *sprint*).

3.4 Contribuciones

Este trabajo de investigación contribuye de las siguientes maneras a la organización bajo estudio:

1. Le da a conocer cuál es el valor de su eficiencia para remover defectos mediante el uso de la métrica de DRE.
2. Si el proceso de remoción de defectos está bajo control estadístico, aun así, se podrá encontrar casos en los cuales se deban realizar mejoras al proceso y su vez permitirá realizar predicciones a futuro.

3. Permite generar un proceso que sea genérico de tal forma que pueda ser aplicado a otros proyectos dentro de la organización bajo estudio.

Por otro lado, los aportes de este trabajo a la comunicad científica son:

1. Se lleva a cabo la implementación de una métrica que se utiliza en contextos secuenciales en una metodología ágil.
2. Proporciona resultados que podrán ser utilizados para la comprobación de la veracidad del uso de la métrica DRE.
3. Aporta una serie de mejoras que pueden ser usadas como base para otras organizaciones.

4. Metodología

En la figura 5 se muestran los pasos que seguimos para el desarrollo de este trabajo de investigación. Esto se considera un protocolo de caso de estudio en donde se toman las decisiones de diseño y donde se definen los procedimientos que se llevan a cabo. Este se actualiza constantemente con el objetivo de fungir como una guía para la recolección de los datos, también le da forma a la fase de planeación donde se decide que fuentes usar y que preguntas realizar. Por otra parte, otros investigadores pueden retroalimentar el proceso definido. Finalmente lleva el registro de todos los datos recolectados, el análisis y los cambios que se debieron realizar con el fin de reportarlos más adelante [23].

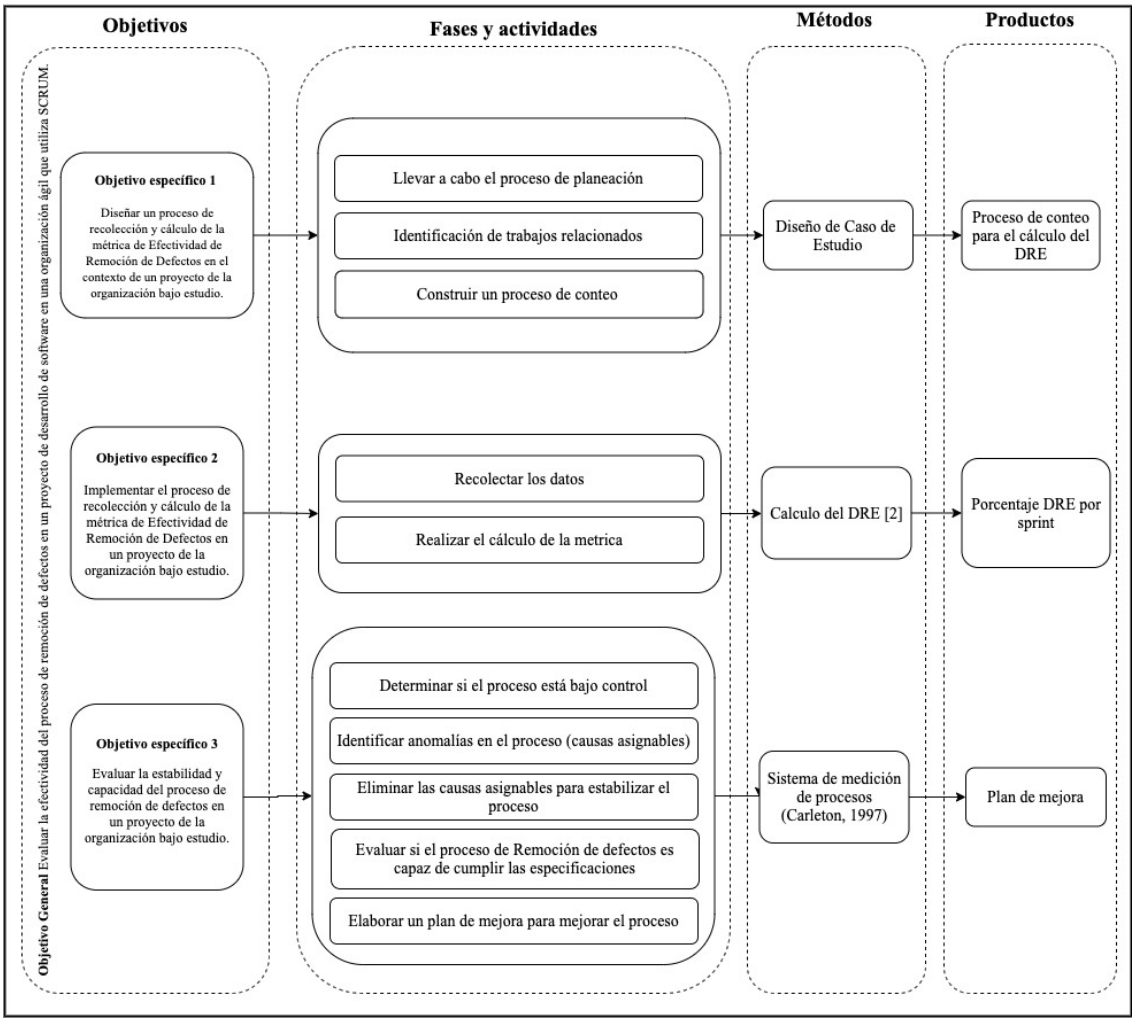

Figura 5. Resumen de metodología usada en la investigación.

4.1 Diseño del estudio

El objetivo del estudio es evaluar qué tan efectiva es la remoción de defectos, para lo cual se define un proceso que permita recolectar y calcular la métrica de DRE en uno de los proyectos de software dentro de la organización bajo estudio.

Con el fin de recolectar la información necesaria para el cálculo de la métrica DRE, se utilizó la herramienta JIRA, en la que se registran las tareas que se trabajan en cada *sprint*. Para contabilizar los defectos, se filtraron las tareas que fueron clasificadas como defectos. Adicionalmente, los defectos encontrados por el cliente pudieron ser identificados en su mayoría mediante la aplicación de una etiqueta que los categoriza como defectos en producción. Como esta etiqueta es registrada por los administradores del proyecto al reportar el defecto, es posible que no siempre sea agregada (por olvido) o que se agregue a un defecto porque este afecta directamente al cliente, sin embargo, podría ser una afectación fuera del código fuente del proyecto (afectación por una mala configuración o un problema a un servicio externo a la organización).

4.1.1 Proceso de conteo

Dentro de la organización se maneja la siguiente clasificación de defectos:

1. *Story-bug*: representa todos aquellos defectos que hayan sido capturados por los ingenieros de calidad durante el desarrollo, en la fase de aseguramiento de calidad.
2. *Regression-bug*: representa todos aquellos defectos que son reportados por el equipo de desarrollo (desarrolladores, ingenieros de calidad, administradores del proyecto) durante el proceso de pruebas de regresión que se lleva a cabo antes o después del lanzamiento de código.
3. *Bug*: representa cualquier defecto que sea encontrado por el cliente después del lanzamiento de una nueva versión de código.

Para esta investigación los *Story-bugs* y los *Regression-Bugs* encontrados antes del lanzamiento (es decir, los que fueron originados en el mismo *sprint* en el cual fueron encontrados) se consideraron defectos encontrados durante la fase de desarrollo, mientras que los *Bugs* y los *Regression-bugs* encontrados después del lanzamiento del código, se consideraron como hallazgos del cliente.

Cabe aclarar que algunos defectos pueden tener soluciones que no requieren un cambio de código. Por ejemplo, en algunas ocasiones una mala configuración de servidor puede generar un defecto no relacionado a código. También un defecto puede no ser reproducible debido a eventos aleatorios o que hubo una mala interpretación de la funcionalidad. Por lo tanto, cualquier tipo de defecto reportado que sea una copia de otro, o cualquier defecto que el equipo de desarrollo haya considerado como inválido o no reproducible no se contó para efectos de la investigación.

Como se mencionó previamente, Castro et al. [6] modificaron el uso de la métrica con el fin de observar el comportamiento de los defectos de una marera acumulada a través de los *sprints de un proyecto*. En nuestro trabajo identificamos además el origen de los defectos encontrados posteriormente a la fase de lanzamiento del software.

Se puede decir que un defecto se origina cuando la verificación o la validación de un producto falla [24]. La verificación consiste en determinar que se obtenga el producto deseado al final de una fase de desarrollo, y la validación corresponde al aseguramiento de que el producto final cumpla con los requerimientos del cliente. Esto implica que el origen de un defecto solo puede ser determinado mediante juicio experto debido a que se debe identificar cuál es la causa de que la validación del producto final no se cumpliera, y con esto determinar dónde fue introducido el defecto. Esta investigación estableció una serie de pasos a seguir, con el fin de determinar dicho origen:

1. **Determinar cuál extracto de código genera el defecto:** en cada uno de los defectos se agrega un *pull request* al fragmento del código que cambió.
2. **Determinar qué código existía anteriormente que aseguraba la validación del producto**: Para ello se utiliza la herramienta *GitHub*. Se puede revisar el historial con el fin de identificar qué cambios se realizaron para causar el defecto.
3. **Asociar dichos cambios a un *sprint*:** la herramienta *GitHub* permite identificar mediante descripciones o versiones de código qué historia de usuario introdujo el defecto.

Por otra parte, para algunos de los defectos seleccionados para el proceso de conteo, se detectó que se había introducido en algún *sprint* anterior que no estaba incluido en el rango de tiempo a analizar. Es por esto que dichos defectos, a pesar de tener una causa raíz, no fueron incluidos como parte del conteo debido a que esto implicaría extender el lapso de tiempo a considerar.

En la figura 6 se resume el proceso de conteo que se llevó a cabo para el conteo de defectos para la aplicación de la métrica DRE.

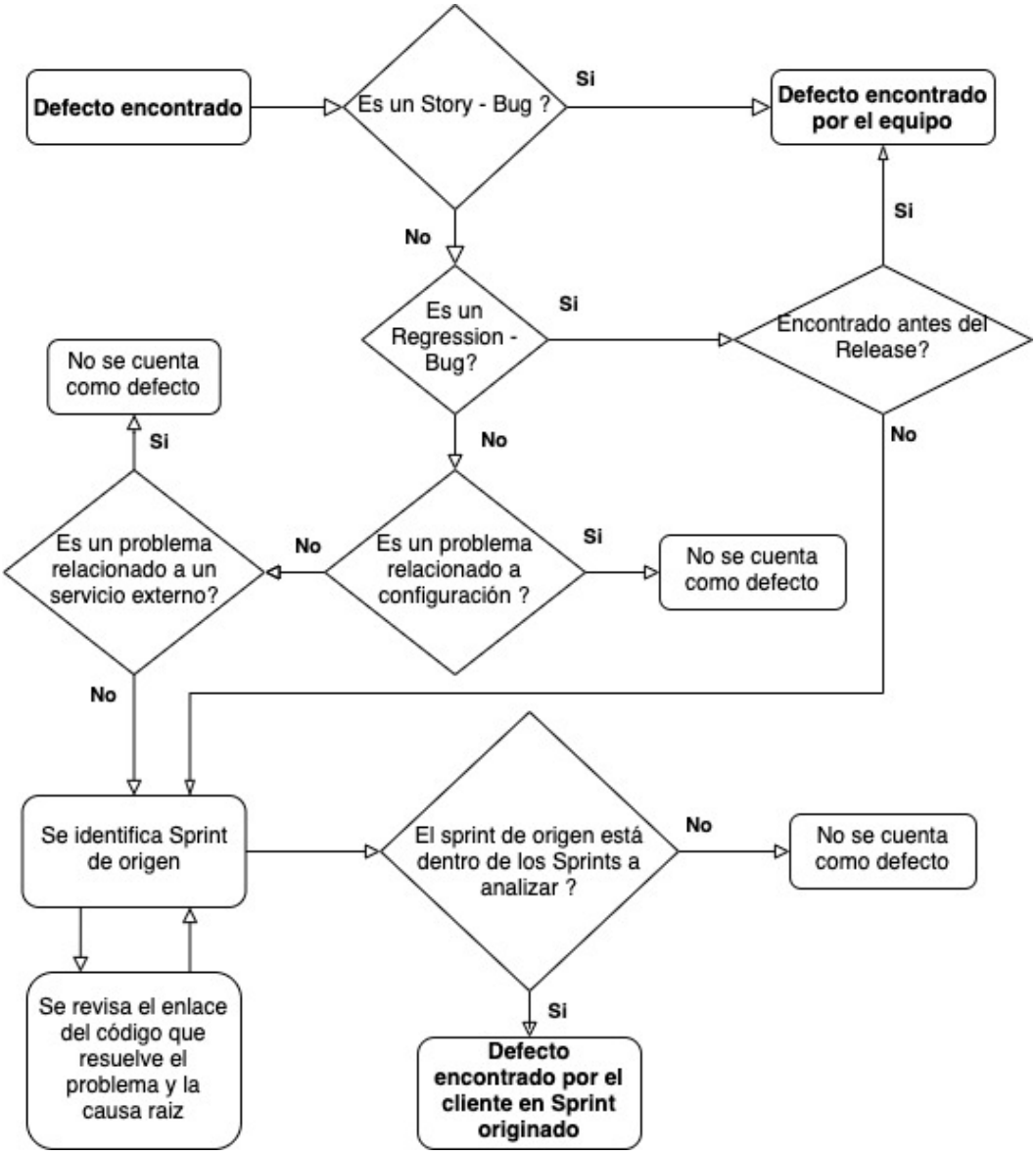

Figura 6. Resumen del proceso de conteo. Elaboración propia.

4.2 Recolección de los datos

La organización bajo estudio eligió estudiar el proyecto “PILOT” debido a que es el más antiguo, ofreciendo más información que puede ser estudiada. Para efectos de esta investigación, se analizó la información recolectada durante el año 2019.

Cada defecto fue clasificado según los siguientes atributos, con el fin de calcular la métrica:

a. Número de *sprint* en el cual se introdujo el defecto.

b. Determinar si fue removido antes del *release* o encontrado por el cliente (*post-release*) donde todo aquel defecto encontrado antes del *release* se considera un defecto de regresión.
c. Tipo de origen (a qué se le atribuye la causa del defecto)
    i. **Corrección Defectuosa:** Esta categoría aplica para defectos previamente reportados. Ocurre cuando la solución del defecto causa otro problema.
    ii. **Mala implementación de código**: La solución que se implementó para una historia de usuario no es la indicada y es propensa a generar un defecto.
    iii. **Problemas posteriores a una actualización**: a causa de una actualización en las librerías del proyecto se genera un defecto.
    iv. **Implementación incompleta:** Cuando se implementó la solución al problema no se tomó en cuenta todo lo que se solicitaba en los requerimientos.
    v. **Definición de requerimientos incompleta**: durante el proceso de planificación de la historia de usuario no se tomó en cuenta información necesaria para la elaboración de la solución.
d. En qué *sprint* se originó el error.

Actualmente, en el proyecto se proporciona una causa raíz definida por el desarrollador que corrige el defecto, por lo que esta información puede ser utilizada para clasificar el defecto según los cuatro atributos anteriormente mencionados.

Finalmente, para obtener la información de los defectos a analizar se debe realizar una consulta a la herramienta *JIRA* para obtener una lista de defectos por categoría que es contada manualmente, y posteriormente cada dato es guardado en una hoja de cálculo. A su vez, se debe usar la herramienta *GitHub* con el fin de realizar el análisis del código que se introdujo para solucionar el defecto.

4.3 Análisis de los datos

Una vez obtenidos los datos por *sprint* en el lapso anteriormente propuesto, se debe realizar el cálculo de la métrica para cada uno de los *sprints* y construir el gráfico de control XmR. Se escogió este gráfico debido a que utiliza subconjuntos de un elemento para mostrar el valor del DRE y permite hacer un análisis en el tiempo [9].

Seguidamente, se aplica una serie de pruebas de estabilidad que permiten determinar si el proceso se encuentra bajo control estadístico o no. En caso de no estarlo, se puede decir que existen causas asignables que deben ser eliminadas del proceso en iteraciones futuras. Si por el contrario el proceso se encuentra bajo el control estadístico, entonces se debe determinar si el valor del DRE se encuentra bajo el valor mínimo de un 85% definido por Jones [2].

Si se identifican puntos de inestabilidad se debe realizar una investigación de dichas observaciones. Si se determina que son causas que perjudican el trabajo del equipo se deben arreglar con el fin de que no vuelvan a ocurrir [9].

4.4 Reporte de resultados

Una vez realizado el análisis, se concluyó mediante los gráficos de control si el proceso se encontraba bajo control estadístico o no. En caso de que no fuese estable o en caso de que el proceso no fuese capaz (llegar al menos al 85% definido por Jones [2]), se debía generar un listado de causas asignables como una base para la elaboración de una propuesta de mejora para ser presentada a la organización y dejar a su criterio si es conveniente la aplicación de dichas mejoras al proceso.

Los Anexos 1 y 2 de este trabajo muestran los datos obtenidos en esta investigación. También se muestran los gráficos de control utilizados para comprobar la estabilidad del proceso, a su vez, como se encontró una inestabilidad que se debe a causas asignables, se muestra un gráfico final en donde se demuestra que el proceso es estable luego de remover dichas causas.

4.5 Amenazas a la validez

Amenaza interna: El equipo de desarrollo podría no registrar de forma apropiada los defectos debido a que no se tiene claro cómo hacerlo. Para mitigarlo se agregó una definición dentro del proceso de conteo que explica bajo qué condiciones específicas se debe categorizar un defecto por su tipo.

Amenaza al constructo: La métrica de DRE no se diseñó para una metodología ágil. El proceso se ha enfocado en metodologías tradicionales como cascada. Con el fin de mitigarla se imitó la forma en que se calcula la métrica utilizando los *sprints* en lugar de las fases de desarrollo de la metodología cascada. Cabe resaltar que los defectos en SCRUM pueden originarse en *sprints* anteriores con respecto al *sprint* en los que son encontrados. Esto se logró mitigar con la introducción del dato del *sprint* de origen dando como resultado un cálculo más exacto del valor del DRE por *sprint.*

Amenaza externa: generalmente el dato del *sprint* de origen no es recolectado en proyectos de software que utilizan la metodología SCRUM por lo que dicha investigación no podría ser replicada. Para mitigarlo se da una explicación del método que se llevó a cabo para la recolección de dicho dato. Además, se le aclara al lector que se requiere de dos datos fundamentales para identificar en donde se originó el defecto: Un *pull request* que introdujo un cambió para solucionar dicho defecto y una breve explicación en donde se explique que causó el defecto.

Fiabilidad: En otros proyectos de software desarrollados bajo SCRUM los defectos pueden ser clasificados de distintas formas por lo que la identificación de causas asignables puede ser confusa. Se debe tomar en cuenta que la clasificación ayuda a encontrar un problema en común y así identificar una solución para todos los defectos. Por lo que cada investigador puede agrupar sus datos de la manera que sea mas conveniente.

5. Resultados

5.1 Descripción de resultados

Al realizar la consulta en la herramienta *JIRA* se encontraron un total de 109 defectos en las categorías de *Regression-Bug* y *Bug,* y un total de 55 defectos dentro de la categoría de *Story-Bug,* los que están descritos en los Anexos 1 y 2 de este trabajo. Al analizar las primeras categorías se encontraron 12 defectos que no eran válidos, la razón de esto es que 6 de ellos se convirtieron en mejoras, 2 de ellos se debieron a un problema de configuración en los servidores, 2 estaban relacionados a problemas en un servicio externo a la organización, y por último 2 se debieron a una configuración errónea del lado del cliente.

En la tabla 1 se introduce la columna de defectos por *sprint* la cual indica el total de defectos que se originaron en el *sprint* a base del proceso de recolección de datos. También se agrega la columna defectos encontrados de regresión que representa todo defecto originado y detectado dentro del mismo *sprint* (antes del *release*). La columna Total de Defectos encontrados después del *release* se compone de la resta de los defectos originados en el *sprint* y los defectos de regresión (originados y encontrados en el mismo *sprint*). La columna de los defectos encontrados por el equipo se identifica como los defectos encontrados por el equipo en la fase de pruebas y la columna de Total de Defectos antes del *release* representa la suma de los defectos de regresión (originados y encontrados en el mismo *sprint*) y los defectos encontrados por el equipo. Finalmente, el cálculo del DRE se realiza usando las columnas Total de defectos encontrados antes del *release* y defectos encontrados después del *release*.

En la tabla 1 se puede observar que los tres últimos *sprints* tienen un valor no definido para el DRE debido a que ni el cliente ni el equipo de desarrollo fueron capaces de encontrar defectos. Estos *sprints* no fueron incluidos en el gráfico de control XmR ya que el decrecimiento se debe a que ninguna de las dos partes encontró defectos.

Cabe aclarar que de los 109 defectos clasificados como tipo *Regression-Bug* y *Bug,* 41 de ellos no pudieron ser incluidos dentro de la tabla 1, esto se debe a que su origen fue encontrado en *sprints* anteriores y como el cálculo del DRE utiliza los defectos originados dentro de un *sprint* estos valores no tienen importancia para dicho calculo en los *sprints* que

se quieren analizar. Aun así, se podrían usar para estudios posteriores que se realicen sobre *sprints* pasados.

Tabla 1. Defectos encontrados y cálculo del DRE por *sprint.*

Sprint	Defectos por sprint	Defectos encontrados de *regresión*	Total, de defectos encontrados **después** del *release*	Defectos encontrados por equipo	Total, de defectos encontrados **antes** del *release*	**DRE**
Sprint 105	1	0	1	1	1	**50%**
Sprint 106	2	0	2	0	0	**0%**
Sprint 107	2	0	2	2	2	**50%**
Sprint 108	2	0	2	1	1	**33%**
Sprint 109	5	2	3	4	6	**67%**
Sprint 110	3	2	1	3	5	**71%**
Sprint 111	1	1	0	0	1	**100%**
Sprint 112	2	0	2	3	3	**60%**
Sprint 113	4	1	3	1	2	**40%**
Sprint 114	4	2	2	5	7	**78%**
Sprint 115	2	1	1	4	5	**83%**
Sprint 116	4	2	2	6	8	**80%**
Sprint 117	4	1	3	3	4	**57%**
Sprint 118	5	1	4	2	3	**42%**
Sprint 119	3	1	2	2	3	**60%**
Sprint 120	2	0	2	3	3	**60%**
Sprint 121	0	0	0	1	1	**100%**
Sprint 122	2	1	1	3	4	**80%**
Sprint 123	1	1	0	1	2	**100%**
Sprint 124	2	1	1	3	4	**80%**
Sprint 125	0	0	0	2	2	**100%**
Sprint 126	1	0	1	5	5	**83%**
Sprint 127	3	2	1	0	2	**67%**
Sprint 128*	0	0	0	0	0	**N/A**
Sprint 129*	0	0	0	0	0	**N/A**
Sprint 130*	0	0	0	0	0	**N/A**

***** Estos *sprints* no reportaron defectos de ningún tipo por lo que no se puede calcular el DRE dado que el denominador es igual a cero

5.2 Análisis de estabilidad y capacidad del proceso de remoción de defectos

Como siguiente paso elaboramos el gráfico de control XmR con el fin de comprobar la estabilidad del proceso. Tal y como se mencionó en la sección 5.1, los últimos tres *sprints* (Sprint 128, Sprint 129 y Sprint 130) no se incluyeron en el gráfico.

En la figura 7 se muestra el gráfico de control XmR generado a partir de los datos de la tabla 1. En primer lugar, se muestra el gráfico X, el cual representa cada uno de los valores individuales del DRE a lo largo del tiempo (en la parte inferior se muestra el número de *sprint* al que cada punto se relaciona). Luego se muestra el grafico mR, el cual representa el rango de variación entre los puntos de forma consecutiva.

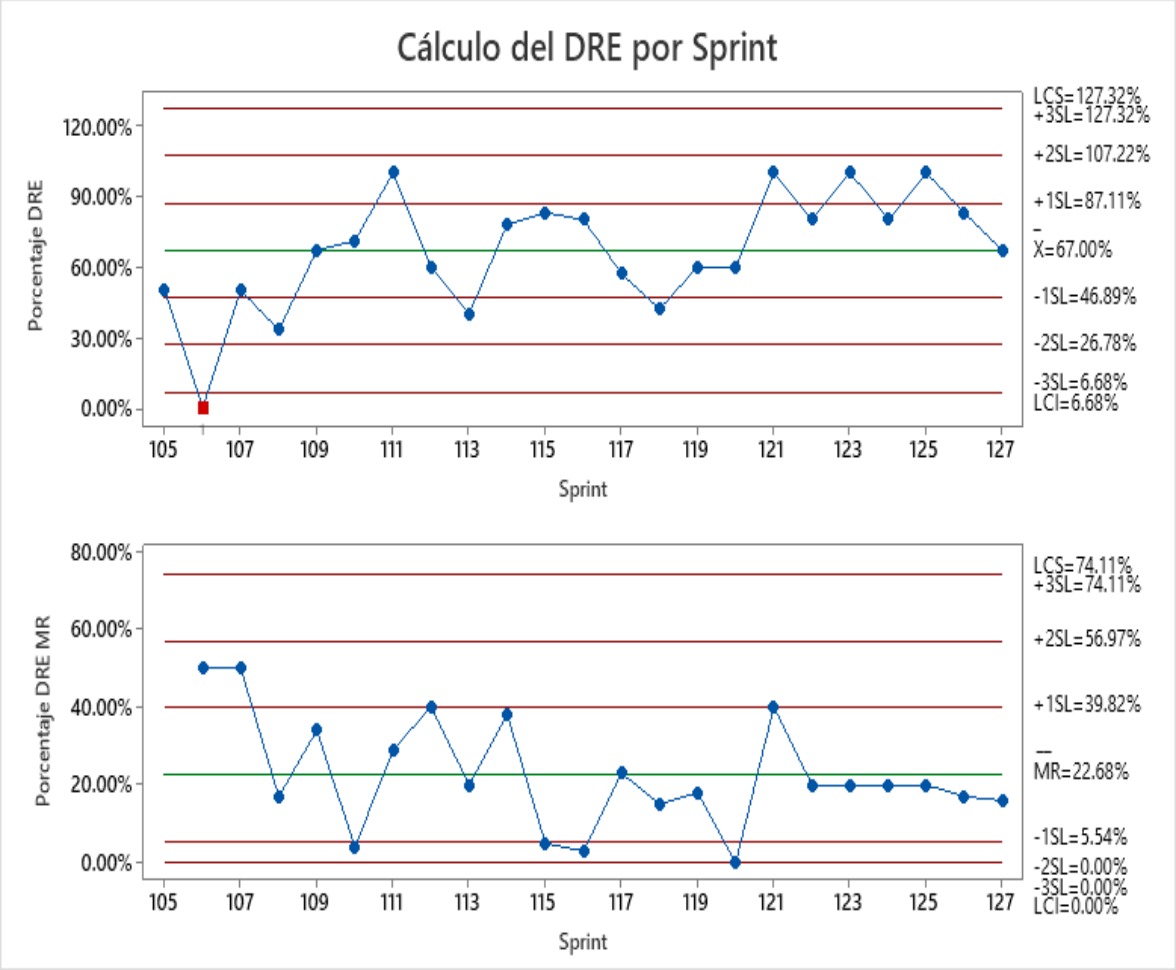

Figura 7. Gráfico de control XmR del DRE por *sprint.*

En la figura 7 se puede observar que se presenta un punto de inestabilidad para el *sprint* 106, por lo que se debe analizar sus causas asignables.

Por otra parte, el valor promedio del gráfico de control X es de 67,00%, lo que significa que el proceso de remoción de defectos de este proyecto es de 67%. Caper Jones [2] afirma que un valor por debajo del 85% representa una insatisfacción para el cliente, por lo tanto, se puede afirmar que el proceso de remoción de defectos de esta organización en el proyecto PILOT no es ni estable (por el *sprint* 106) ni capaz (67% contra 85% esperado).

5.2.1 Remoción de puntos inestables y actualización de gráfico

Durante el proceso de estabilización se removió solamente el *sprint* 106 que es el único valor que causa la inestabilidad. Con este punto eliminado de la muestra, se puede observar que el proceso es estable en el gráfico de control de la figura 8.

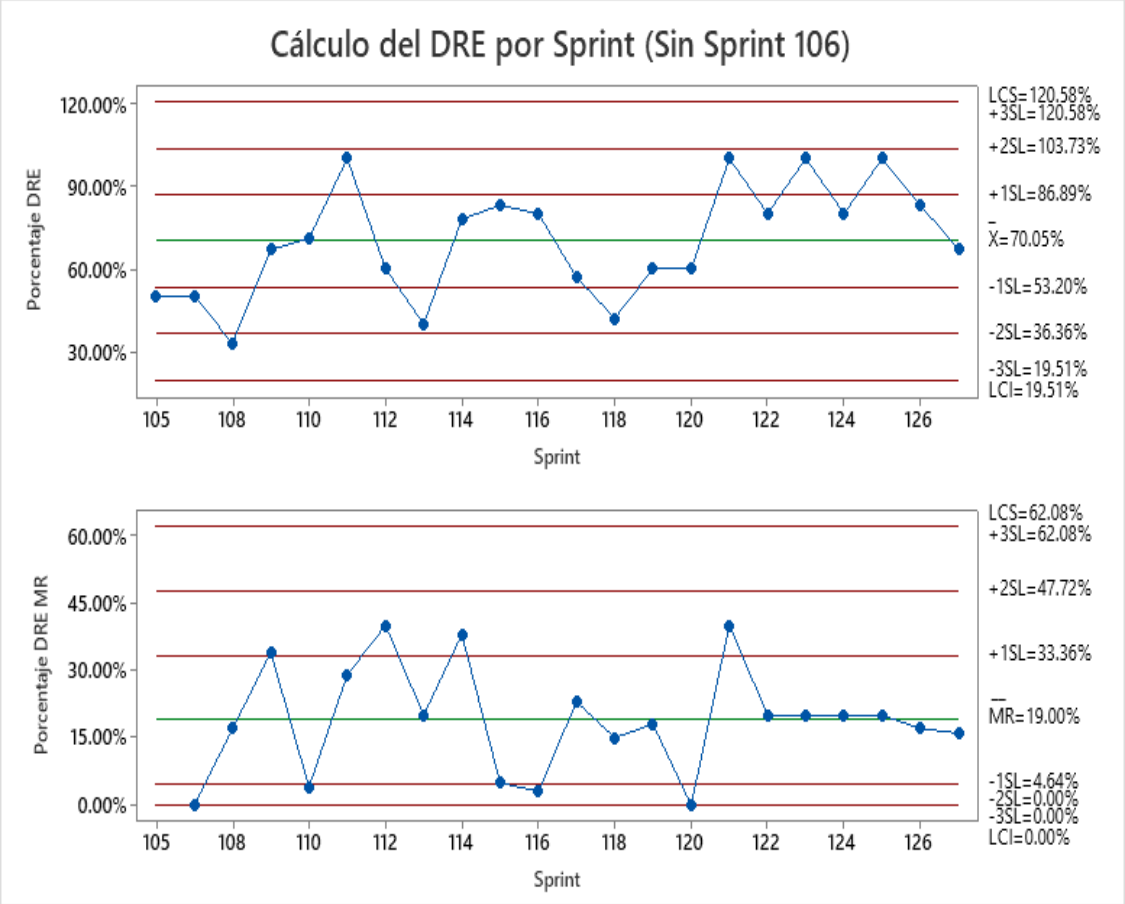

Figura 8. Gráfico de control XmR del porcentaje de DRE por sprint removiendo puntos inestables.

Como se observa, el proceso es estable para ambos gráficos, inclusive el valor promedio del gráfico se incrementó a 70.05%, por lo que se debe buscar las causas asignables d el *sprint* 106. Además, ya que el DRE es bajo (67%), se debe proponer una serie de medidas que mejoren este valor y así asegurar que el proceso sea capaz para la organización bajo estudio.

5.3 Revisión de causas asignables y propuesta de mejora

5.3.1 Causas asignables del *sprint* 106

Como se observa en la Tabla 1, el DRE del *sprint* 106 es 0% ya que el equipo de desarrollo no fue capaz de encontrar defectos, pero el cliente halló dos defectos. Estos defectos corresponden por su identificador a *PILOT-9727 y PILOT-9486.*

El primer defecto reportado en el *sprint* 114 generaba una afectación en un enlace web en una notificación: cuando se intentaba usar el enlace no redirigía al usuario a la página esperada. Una vez solucionado el defecto se encontró que solo ocurría en el cliente de correo de *MacOS* ya que en el enlace no se especificaba el protocolo *https* por lo que dicho cliente intentaba arreglar el enlace a su manera dejándolo inservible.

Al revisar el código que lo arreglaba se pudo identificar que ese cambio lo causó otro defecto con el identificador *PILOT-9279* que intentaba reparar un error relacionado con el mismo enlace, sin embargo, parece que no se tomó en cuenta el hecho de agregar el protocolo *https* al enlace. Al consultar la evidencia que demostraba que el arreglo se había aplicado se observa que se utiliza el cliente de correo en versión web en lugar de un gestor de correo del sistema operativo. Por lo que la causa probable a este defecto es la falta de pruebas en múltiples ambientes usando diferentes herramientas de gestión de correo.

El otro defecto introducido en el *sprint* 106 es el *PILOT-9486,* que fue reportado en el *sprint* 113. El problema encontrado consistía en que la aplicación *PILOT* debía enviar un archivo PDF a un servicio externo, pero a causa de una excepción en el código no prevenida el archivo no se enviaba. Al revisar la causa raíz del defecto documentada por el desarrollador se encuentra que está relacionado directamente a la historia de usuario *PILOT-9240*. Este desarrollador afirma que se introdujo una función que ocultaba cierta información en un *log* por reglas de negocio, sin embargo, hay un flujo en el cual dicha información no existe y al intentar accederla se genera una excepción.

Al revisar la evidencia se observa que solo se hizo una prueba con el escenario trivial (donde la información siempre existe), por lo que la causa probable en este escenario sería la falta de pruebas en escenarios no convencionales o que no sigan el camino esperado.

En conclusión, las dos causas asignables que causan la inestabilidad del proceso son:

- Falta de pruebas en aplicaciones que el usuario final usará en su ambiente de trabajo.
- Falta de pruebas en escenarios alternativos relacionados con la historia de usuario en la que se trabaja.

5.3.2 Causas asignables con respecto al bajo porcentaje del DRE

Una vez identificadas las causas asignables del *sprint* 106 y eliminado este punto de la muestra, se generó un segundo gráfico de control XmR que demuestra que el proceso es estable. Sin embargo, el porcentaje promedio de DRE señala que el proceso de remoción de defectos no es capaz pues es menor al 85% deseado. Este valor debe mantenerse al menos en un 85% [2] con el fin de lograr la satisfacción del cliente, por lo que el valor obtenido de 67% debe ser mejorado.

En la tabla 2 se detalla el conteo de defectos por *sprint* de acuerdo con su tipo de origen. Se puede observar que 20 defectos se debieron a una mala implementación de código, 8 de ellos se originaron por un defecto que no se arregló de forma adecuada, 3 de ellos se debieron a que no se completaron los requerimientos solicitados, y, por último, un defecto fue originado por una actualización y el otros 6 fueron originados por una definición de requerimientos incompleta. En la siguiente sección se detalla el origen de los defectos por sprint.

Tabla 2: Cantidad de defectos por *sprint* según el tipo de origen.

Sprint	**Mala implementación de código**	**Corrección Defectuosa**	**Implementación Incompleta**	**Problemas posteriores a actualización**	**Definición de requerimientos incompleta**
Sprint 105	1	0	0	0	0
Sprint 106	1	1	0	0	0
Sprint 107	2	0	0	0	0
Sprint 108	0	0	2	0	0
Sprint 109	3	0	0	0	0
Sprint 110	2	1	0	0	0
Sprint 112	1	1	0	0	0
Sprint 113	0	1	0	1	1
Sprint 114	2	0	0	0	0
Sprint 115	0	0	1	0	0
Sprint 116	2	0	0	0	0
Sprint 117	0	3	0	0	0
Sprint 118	3	1	0	0	0
Sprint 119	0	0	0	0	2
Sprint 120	0	0	0	0	2
Sprint 122	1	0	0	0	0
Sprint 124	1	0	0	0	0
Sprint 126	1	0	0	0	0
Sprint 127	0	0	0	0	1
Total	20	8	3	1	6

5.3.2.1. Mala implementación de código

En el *sprint* 105 se observa que la historia de usuario *PILOT-9211* introduce el defecto que fue corregido como parte del defecto *PILOT-9568*. Esta historia de usuario tenía como objetivo filtrar una serie de datos que se obtenían de un servicio externo para ser mostrados en una interfaz. Sin embargo, en el transcurso del desarrollo de la historia de usuario no se contaba con datos reales, por lo que se aceptó la historia utilizando datos de una estructura por defecto definida con base en la documentación de un servicio de terceros. Una vez que el cliente evaluó la funcionalidad se determinó que había una mala implementación debido a que no se utilizaba el valor adecuado para el filtrado de resultados. Para este *sprint* la causa asignable es la falta de pruebas con datos reales una vez implementado el servicio.

Para el *sprint* 106 previamente determinamos que es un punto de inestabilidad en el primer grafico de control XmR, denotando que los defectos *PILOT-9727 y PILOT-9486* fueron los responsables y generando como causas asignables la falta de pruebas de escenarios alternativos y la falta de pruebas con respecto a herramientas usadas por el usuario final.

En el *sprint* 107 se detectó que la funcionalidad implementada en la historia de usuario *PILOT-9284* generó una afectación que fue resuelta como parte del defecto *PILOT-9385*. Se encontró que la interfaz de usuario estaba bloqueada y solo se mostraba un *spinner* de forma permanente, lo que ocurría debido a un error del código *JavaScript* en el navegador. Un dato importante es que esto solo ocurría en el navegador Internet Explorer versión 11, y al identificar la causa se encontró que ciertos métodos del lenguaje no son soportados por este navegador, pero si por otros navegadores. Debido a esto, la causa raíz de este defecto se describe como la falta de pruebas en otros navegadores, un escenario similar a uno de los defectos reportados en el *sprint* 106.

Además, en el *sprint* 107 se implementó un nuevo tipo de autenticación que utilizaba el mecanismo de *SSO* (*Single Sign On por sus siglas en inglés*) mediante el cual el usuario solo debe proporcionar sus credenciales una vez para tener acceso a múltiples sitios. El requerimiento tenía algo de incertidumbre debido a que no se conocía las capacidades del método de autenticación, sin embargo, cuando se desarrolló este requerimiento se hizo en

base a configuraciones específicas del sitio, por ejemplo: el uso del subdominio web especifico de un cliente para validar la autenticación. Cuando se revisó esta funcionalidad no hubo ningún problema en principio ya que solo existía una organización usando este mecanismo. Pero cuando se reportó el defecto *PILOT-9921* ya existían varias organizaciones usándolo, por lo que la causa de este defecto está relacionada en el hecho de construir funcionalidades con datos pre existentes en lugar de elaborar soluciones genéricas.

En el *sprint* 109 se identificó que la causa de 3 defectos se era una mala implementación. El primero corresponde a *PILOT-10292,* el segundo a *PILOT-9500* y el tercero a *PILOT-9635.*

El primer defecto fue originado por la implementación de la historia de usuario *PILOT-9367* la que consistía en la adición de una serie de comentarios en una sección de la interfaz de usuario. Lo que ocurría es que el almacenaje de dichos comentarios se hacía en un valor *JSON* dentro de la base de datos mientras que otros se guardaban como registros en la base de datos. Al combinar estos 2 tipos de datos se hizo de manera irregular por lo que al realizar la construcción de los datos para la interfaz se generaban errores. Cuando se verificó solo se utilizó un tipo de registro de datos (Registros *JSON*) por lo que no se observó la afectación. Por lo tanto, la causa probable de este defecto erradica en una definición irregular de datos y una falta de pruebas con escenarios alternativos.

El segundo defecto fue ocasionado por la implementación de la historia de usuario *PILOT-9401* en la que se había introducido una nueva entrada para un formulario en donde su visibilidad se controlaba por una configuración asociada a la organización a la que pertenece el usuario final. Al implementar la funcionalidad claramente se introdujo el valor de configuración, pero solo para una de las organizaciones en la aplicación, por lo que una vez que se cambió de usuario y por ende de organización, dicho valor no se encontraba disponible. Es por esto que al ingresar al formulario se generaba un error de *JavaScript.* Por lo que la causa asignable a este defecto se le atribuye al hecho de no probar con otras organizaciones que no tuviesen el valor de configuración esperado.

El tercer defecto fue ocasionado por la historia de usuario *PILOT-9393* relacionada con la mejora de la validación de los archivos que se suben a la aplicación mediante un

filtrado de extensiones. Al implementar esta solución no se tomó en cuenta que algunas extensiones se pueden escribir en letras mayúsculas, por lo que al subir un archivo con este tipo de extensión la aplicación lo identificaba como invalido cuando en realidad no lo era, consultando la evidencia se observa que solo se adjuntan archivos con extensiones en letras minúsculas. Por lo que la causa se debe a la falta de pruebas con respecto a escenarios alternativos.

Para el *sprint* 110 se encontró que la historia de usuario *PILOT-9458* fue la responsable del defecto reportado en el defecto *PILOT-9534*. En la funcionalidad a implementar se pretendía completar un valor por defecto en un formulario basado en el rol del usuario. Sin embargo, esta definición no se hizo de manera adecuada debido a que una vez que se almacenaban los datos del lado del servidor y se volvían a cargar para continuar con un proceso de edición el valor se sobre escribía por la forma en que se ejecutaba el código. Al revisar la evidencia de esta funcionalidad se observó que solo se probó el escenario donde se introducen los datos por primera vez, dejando de lado cualquier edición posterior, por lo que la causa a este defecto se debió nuevamente a la falta de pruebas en flujos alternativos.

En el *sprint* 112 en la historia de usuario *PILOT-9512* fue la responsable de ocasionar el defecto *PILOT-9778*. La funcionalidad buscaba la implementación de un listado de archivos adjuntos a un registro por lo que se desarrolló dicho componente visual, fue evaluado y aceptado para ser entregado al cliente. Sin embargo, por no probar el escenario donde se abre uno de los archivos al seleccionarlo de una lista no se detectó que la interfaz de usuario se bloqueaba y solo se mostraba un *spinner* en pantalla. Al corregirse el defecto se identificó una mala implementación en los eventos de código *JavaScript* generaba este efecto solo cuando se abría un archivo, por lo que la causa a este defecto se atribuye a la falta de pruebas con escenarios alternativos, en este caso el hecho de no abrir el archivo.

Dentro del *sprint* 114 se originaron 2 defectos de los cuales el defecto *PILOT-9849* está directamente relacionado con una afectación a un servicio externo a la aplicación *PILOT,* pero desarrollado dentro de la organización bajo estudio el cual generaba una afectación para la aplicación en cuestión. La historia de usuario asociada al servicio *SP* con el identificador *SP-155* busca implementar una sincronización de usuarios de la aplicación *PILOT*, sin

embargo, al ser un servicio externo esta debe almacenar los identificadores de la aplicación de origen por cada registro que sincronice con el fin de lograr una interacción adecuada entre ambas aplicaciones. Al corroborar la funcionalidad solo se aseguró que el proceso sincronización de datos se ejecutara de forma exitosa, mas no se verificó la integridad de los datos. Por lo que al interactuar la aplicación *PILOT* con el servicio *SP* se originaba un error debido a que los identificadores no coincidían. Se puede decir que la causa asignable a este defecto está relacionada a la falta de pruebas de código al no verificar de qué forma se replicaban los datos.

Por otra parte, el defecto *PILOT-9751* cuyo origen también se encuentra en el *sprint* 114 fue causado por la historia de usuario *PILOT-9591*. La idea era cambiar el texto que se mostraba en notificaciones basadas en correo electrónico, sin embargo, un detalle importante que no se consideró era el hecho de que estas notificaciones se enviaban de forma asíncrona, por lo que implica que un subproceso de la aplicación era el encargado de realizar dicha tarea usando como punto común de la información entre la aplicación y el subproceso una base de datos en memoria. Al modificar el código no se tomó en cuenta que los datos debían también ser cambiados en la base de datos en memoria, por lo que al ejecutarse el nuevo código esto generó un conflicto con la información almacenada por el código anterior. La causa en este caso se describe como la falta de actualización de datos de sub procesos al realizar un cambio en su ejecución.

Dentro del *sprint* 116 se originaron los defectos *PILOT-9883* y *PILOT-10165,* este último fue ocasionado por un cambio en el servicio externo *SP* (*SP-264*) al igual que en el *sprint* anterior. En la historia se requería que se aplicara cambio en el estado de un usuario dentro de *SP* basado en el estado del mismo usuario en la aplicación *PILOT* ya que cuando los usuarios ingresan por primera vez lo hacen a través de esta aplicación y no a través del servicio *SP.* Al hacer esto se introdujo un cambio dentro de la aplicación *PILOT* para hacer uso de recursos del servicio *SP*. Este cambio no consideró que ya se hacían llamados similares en otras partes de la aplicación y al cambiar el orden en el código estas devolvían un error de *usuario no autenticado* (401 de acuerdo con el protocolo *http*). Al revisar la evidencia se encontró que solo se verificó la funcionalidad involucrada a la historia y no se consideró revisar otras funcionalidades relacionadas.

Para el defecto *PILOT-9883* se identificó que fue originada la historia de usuario *PILOT-9781*. Esta funcionalidad buscaba filtrar las opciones de un selector en un formulario basado en la configuración de una organización. Para ello se utilizó una consulta *SQL* que filtraba los datos necesarios, sin embargo, esta consulta tenía un error ya que traía una serie de datos duplicados que solamente fueron percibidos por un error en el código *JavaScript.* Al consultar la evidencia se observa que la funcionalidad fue revisada de forma general sin observar la consola del navegador, hay que aclarar que este error de *JavaScript* no bloqueaba el funcionamiento de la interfaz de usuario, por lo que no tuvo problema en ser aceptado. La causa probable en este caso es la falta de revisión en la consola del navegador en búsqueda de errores.

En el *sprint* 118 como parte de la historia de usuario *PILOT-9894* se agregó una funcionalidad que requería la refactorización de un código existente con el fin de aplicarlo para un cliente con necesidades específicas, pero el problema se origina cuando solo se prueba la funcionalidad original.

Al revisar la causa raíz se identifica que lo que antes solía ser la propiedad de un objeto en código *JavaScript* ahora es un método por lo que, al no usar los paréntesis para invocar al método, este no devolvía el valor esperado. Al igual que en casos anteriores solo se prueba la funcionalidad del nuevo requerimiento ignorando que cambio en componentes existentes también se pueden ver afectados. Por otra parte, al cambiar la definición de un método se vio afectada la capacidad de crear nuevos registros debido a que la estructura de datos suministrada no era válida. Este defecto fue capturado en el defecto *PILOT-10215.*

En el *sprint* 118 también se introdujo una nueva funcionalidad que consistía en la creación de un *API* (*Application Programming Interface por sus siglas en inglés*) para que clientes externos pudieran interactuar con la funcionalidad de la aplicación. Uno de los requerimientos que fue desarrollado en la historia de usuario *PILOT-9967* consistía en proporcionar un mecanismo de autenticación en el cual cada usuario utilizaba un identificador propio definido por el cliente externo. El problema con esto es que el nombre del identificador dependía de la configuración de la organización a la que pertenece el usuario por lo que al cambiarlo de organización o agregarle otras organizaciones el valor del nombre del identificador se perdía. Cuando se revisó la evidencia de esta historia se encontró que se

hizo una prueba usando solo una organización por usuario, no se hizo un cambio de la misma o se agregó múltiples organizaciones, por lo que la causa en este escenario se debe a la falta de pruebas ante escenarios alternativos.

denle el *sprint* 122 se introdujo la historia de usuario *PILOT-10230* el cual consistía en generar un formulario para que el usuario final compartiera la satisfacción con el servicio brindado. El problema radica en que el nombre de la persona que prestaba el servicio no se mostraba correctamente, inclusive el nombre era el de otra persona. Al identificar la causa se encontró que el uso de un identificador erróneo traía los datos de otro usuario. En la evidencia se observa que el valor es incorrecto por lo que se puede decir que la vista del formulario no fue verificada de forma minuciosa.

En el *sprint* 124 como parte de la historia de usuario *PILOT-10439* se agregó una funcionalidad que consistía en mostrar la pre-visualización de un archivo en un nuevo componente. Cabe aclarar que ya existía una funcionalidad que mostraba dicha pre-visualización, pero en una ventana aparte. Además, se debe considerar que esta nueva funcionalidad no estaba disponible para todos los usuarios; era controlado por la configuración de la organización del usuario. Nuevamente al revisar este requerimiento solo se verifica la nueva funcionalidad ignorando que la funcionalidad original seguía estable, por lo que no se detecta que esta estaba fallando como se reportó en el defecto *PILOT-10562*.

Finalmente, en el *sprint* 126 en la historia de usuario *PILOT-10482* se pide que se haga un cambio en los mensajes mostrados al usuario. Sin embargo, uno de estos mensajes se construía utilizando una propiedad errónea ocasionando que el mensaje no se mostrara. Como ya se ha mencionado en otros *sprints,* para esta historia solo se verifica uno de los escenarios.

En resumen, se encontraron las siguientes causas asignables como parte de una mala implementación en el código:

- Falta de pruebas con datos reales generados por el usuario final.
- Falta de pruebas usando herramientas que utiliza el usuario final (navegadores, clientes de correo).
- Falta de pruebas ante inconsistencia en los datos retornados por el servidor.

- Falta de pruebas de código ante la replicación de información entre servicios.
- No verificar que subprocesos que dependen de información almacenada en memoria se vean afectados por cambios en el código.
- Falta de pruebas sobre refactorizaciones de código que afecten funcionalidades existentes.
- La revisión de pares del código (o *code review* en inglés) no evita que errores de sintaxis o de flujo sean prevenidos.
- Falta de verificación de todos los escenarios posibles dentro de una misma historia de usuario.
- Uso de datos estáticos para el desarrollo de una funcionalidad que se pretende que sea genérica.

5.3.2.2. Corrección defectuosa

En el *sprint* 106 ya se había encontrado *PILOT-9727* como parte de la inestabilidad en el primer gráfico de control en donde se determinó que ocurría como parte de un arreglo al enlace en una notificación de correo, pero al no enviarse el protocolo *http* el cliente de correo de MacOS interpretaba el enlace de otra forma por lo que la causa probable se había asociado a la falta de pruebas con herramientas del usuario final.

En el *sprint* 110 se soluciona el defecto *PILOT-9516* el cual reportaba que se perdía el valor un registro categorizado por un nombre dentro de un formulario, esto se debía a que este registro se guardaba como una relación y al no enviarse el identificador que establecía dicha relación nada impedía que se generara duplicaciones. Sin embargo, este defecto no solventaba situaciones en las cuales se proporcionaban valores para otras categorías, por lo que aún se seguían reportando valores duplicados. Por lo que se debió reportar el defecto *PILOT-9549*. Cuando se consultó la evidencia la prueba se realizó solamente usando una de las categorías.

En el *sprint* 112 se realizó la implementación de la historia de usuario *PILOT-9539* en el cual se aprovechó y se realizó una mejora a la mantenibilidad del código *JavaScript,* sin embargo, el desarrollador no tomó en cuenta que la sección en donde hizo la mejora iba

a ocasionar un bloqueo de la interfaz de usuario debido a que el código que mejoró tenía una sintaxis que no era compatible con el navegador Internet Explorer 11. En el proyecto existe una herramienta que convierte este tipo de sintaxis a código funcional por casi todos los navegadores, sin embargo, la sección cambiada está escrita en código *JavaScript* básico por lo que no se debe aplicar el proceso de conversión. Una causa probable en este escenario se describe como la falta de documentación sobre la estructura del proyecto, esto quiere decir que no se explica que partes deben ser mantenidas y cuáles no.

En el *sprint* 113 se encontró un escenario similar a la historia anterior, pero en este caso era una mejora a la mantenibilidad del lado del código del servidor. En la historia de usuario *PILOT-9623* el desarrollador a cargo introdujo una mejora basada en el lenguaje de programación *PHP,* en sus versiones más recientes se introduce el concepto de *typehint* el cual consiste en indicar el tipo de dato del parámetro de un método. El problema con esto es cuando ese parámetro puede recibir varios tipos de valores, el desarrollador no tenía claro esto y cuando se encontró el defecto *PILOT-9663* se identificó que ese parámetro podía recibir valores de tipo *string* y a su vez valores nulos. Por lo que la causa probable en este escenario es similar a la anterior, pero también se puede afirmar que no se tenía claro el concepto que se quería aplicar para la mejora de la mantenibilidad.

Dentro del *sprint* 117 una misma historia de usuario (*PILOT-9719*) ocasionó 2 defectos (*PILOT-10045* y *PILOT-10607* respectivamente). Esta historia consistía en mejorar el tiempo de respuesta del servidor cuando se le hacia una solicitud ya que en ese momento duraba alrededor de 3 segundos y al finalizar la historia se mejoró el tiempo a menos de un segundo. Este tipo de mejoras implica re diseñar consultas *SQL* en donde se eliminan datos que no son requeridos por la interfaz. Pero esto ocasionó que se eliminaran 2 datos importantes que se creyeron innecesarios (cada defecto reporta uno de estos datos perdidos). Cuando se revisó la evidencia de esta historia solo se corroboró la mejora en el tiempo, mas no se revisó la integridad del interfaz una vez realizado el cambio. En este caso existen 2 causas probables, la primera se le adjudica a la falta de revisión de la integridad de la información mostrada en la interfaz y la segunda se le adjudica a la falta de pruebas de integración por parte de los desarrolladores sobre la información requerida que se debía retornar.

Además de estos 2 defectos se determinó que el defecto *PILOT-10302* fue originado en el *sprint* 117 como parte del arreglo introducido por el defecto *PILOT-9860*. El origen de este defecto fue clasificado como una definición de requerimientos incompleta, ya que se pedía borrar un archivo adjunto solo si este no estaba relacionado con otros registros sin embargo nunca se tomó en cuenta que los archivos adjuntos podían ser eliminados dentro del formulario del mismo registro asociado. Al solucionarse este defecto se verificó que podía ser eliminado de registros existentes. Sin embargo, cuando el archivo se eliminaba del formulario y el registro no había sido guardado aun esto generaba un error en el servidor impidiendo nuevamente que el archivo adjunto pudiese ser eliminado. Por lo que la causa probable a este defecto se debe a la falta de pruebas ante escenarios alternativos, en este caso eliminar el archivo adjunto cuando el registro relacionado aún no existe.

Para el *sprint* 118, con la introducción de la refactorización de la historia de usuario *PILOT-9894* se hizo un cambio con el fin de seguir los estándares de mantenibilidad del proyecto en los que se indica que se debe usar tres iguales (===) en lugar de dos iguales (==) para el lenguaje *JavaScript*. Es importante aclarar que su función no es la misma ya que el primero compara los tipos de datos para una mayor exactitud (razón por la que se decide usar este operador), mientras que el segundo solo verifica si el valor es el mismo ignorando el tipo de datos. Según parece, el desarrollador a cargo de la refactorización no tenía clara esta diferencia y no se percató que comparaba un valor de tipo numérico (tipo *int*) con otro valor de tipo *string* por lo que no se hacía un filtrado correcto y esto ocasionaba que no se mostrara ciertas opciones requeridas en un formulario. Entonces, la causa de este defecto es compartida con el defecto *PILOT-9663* en la cual no se tenía claro el uso del operador de igualdad en *JavaScript.*

En resumen, el tipo de origen Corrección Defectuosa identifica las siguientes causas asignables:

- Falta de verificación de todos los escenarios posibles dentro de una misma historia de usuario (También reportada para los defectos de tipo Mala implementación).
- Falta de claridad al aplicar mejoras en la mantenibilidad (desconocimiento en los operadores y la sintaxis de los lenguajes de programación usados).

- Falta de pruebas de integración por parte de los desarrolladores ante mejoras en el rendimiento en las respuestas del servidor.

5.3.2.3. Implementación incompleta

En el *sprint* 108 se originaron 2 defectos el primero (*PILOT-9462*) fue originado como parte de la historia de usuario *PILOT-9322* la cual consiste en calcular una fecha basada en 2 configuraciones de la organización: un lapso de horas y la zona horaria. Al revisar el código implementado se observa que se aplicó la primera configuración mas no la segunda. Cuando se revisó la evidencia se encontró que solo se revisó el cálculo basado en el lapso de tiempo, no hay evidencia de que se considerara la zona horaria de la organización. La causa en este caso fue la falta de atención a los requerimientos por parte del desarrollador y del ingeniero de calidad.

El segundo defecto del *sprint* 108 fue causado por la historia de usuario *PILOT-9321*. En esta historia se pedía actualizar una serie de registros en la base de datos usando subprocesos donde los cambios provenían de un archivo en formato *CSV* (*Comma Separated Value por sus siglas en inglés*) que era suministrado por el usuario final. Cuando se probó esta funcionalidad se hizo de forma apropiada sin embargo se probó en un ambiente en donde el procesamiento de datos se hace dentro de un mismo servidor. Es importante aclarar que la aplicación utiliza una arquitectura de ambientes distribuidos en el ambiente de producción con el fin de asegurar la disponibilidad de la aplicación. Cuando se volvió a probar la funcionalidad (ya en un ambiente real) se encontró que uno de los servidores (el que ejecuta los subprocesos específicamente) no era capaz de encontrar el archivo adjunto ya que este se encontraba en otro servidor (el de aplicación). La causa de este defecto radica en la falta de pruebas ante el ambiente distribuido que maneja actualmente la aplicación en producción.

Para el *sprint* 115 se encontró que la historia de usuario *PILOT-9659* realizaba una asignación de una relación basado en el estado de un registro. En la actualidad dicha asignación se aplica cuando el registro se encuentra en un subconjunto de estados, pero cuando la historia se implementó inicialmente se ignoró uno de estos estados, por lo que cuando se reportó el defecto *PILOT-10574* se había encontrado un bloqueo en la

funcionalidad cuando el registro se encontraba en el estado ignorado. Al revisar la evidencia se aprecia que solo se probó con uno de los estados, por lo que la causa en este caso fue la falta de pruebas ante escenarios alternativos.

En resumen, para el tipo de origen de Implementación incompleta se encontraron las siguientes causas asignables:

- Falta de verificación de requerimientos en ambientes distribuidos.
- Falta de verificación de todos los escenarios posibles dentro de una misma historia de usuario.

5.3.2.4. Problemas posteriores a la actualización

En el *sprint* 113 se realizó una actualización al *framework* de la aplicación del lado del servidor, todo el esfuerzo de este trabajo fue reportado bajo la historia de usuario *PILOT-9390*, Al reportar el defecto asociado a esta historia (*PILOT-9738*) se descubrió que en la actualización se había renombrado una clase asociada a la ejecución de sub procesos. Ya que la ejecución de sub procesos depende de una base de datos localizada en memoria esta no es cambiada una vez que se cambia el código, por lo que el nombre anterior de la clase persistía dentro de ella. Por lo tanto, este defecto presenta como causas asignables:

- Fallo en la documentación del *framework* ya que no aclaró que esta clase había cambiado de nombre y se debía prevenir.
- No verificar que subprocesos que dependen de información almacenada en memoria se vean afectados por cambios en el código.

5.3.2.5. Definición de requerimientos incompleta

En el *sprint* 113 se definió la historia de usuario *PILOT-9523*, esta historia ha estado relacionada a otros defectos mencionados en las secciones anteriores. La idea era permitir la eliminación de un archivo adjunto relacionado a un registro, sin embargo, se debía validar que el archivo no pudiese ser eliminado si existía una asociación con otros registros que están asociados al primer registro. El problema con esto es que este archivo podía ser eliminado a

través de los formularios de los registros secundarios y ya que existía una validación el servidor retornaba un error al intentar eliminarlos. Al consultar los requerimientos de esta historia se observa que nunca se contempló el flujo de la eliminación en el lado de los registros secundarios. Si se introdujo la validación se debió contemplar ese escenario como parte de los requerimientos, no esperar a que fuese encontrado como un defecto. Por lo que la causa asignable a este defecto fue el hecho de no revisar la afectación de otras partes de la aplicación al definir una regla de negocio.

Dentro del *sprint 119* se originaron 2 defectos (*PILOT-10245* y *PILOT-10383* respectivamente), El primero se debió a una confusión con la organización asociada a un usuario para el proceso de autenticación para el *API* descrita en la historia de usuario *PILOT-9967*. Esta confusión fue generada debido a que en otro defecto reportado con el identificador *PILOT-10075* ya que este especificaba de forma directa el valor que se debía utilizar en aquel entonces. Por lo que se puede decir que la causa asignable se debe a la falta de comunicación en el equipo ya que los valores a utilizar cambian entre tiquetes.

El segundo defecto reportado bajo el identificador *PILOT-10383* especifica de forma directa que no se incluyó la funcionalidad que se reportó como defectuosa, por lo que al igual que el defecto anterior, la causa probable corresponde a una falta de comunicación del equipo ya que no se tiene claridad que funcionalidades fueron implementadas o no para una historia de usuario.

Para el *sprint* 120 se origina el defecto *PILOT-10221* el reporte indica que una entrada de un formulario no respeta la configuración de la organización a la que se encuentra asociada. Al solucionar el problema se identifica que se consulta la configuración, pero en la organización equivocada. Sin embargo, al revisar la historia de usuario relacionada (*PILOT-10055*) se determina que nunca se especificó de cual organización se debía obtener el valor de configuración, simplemente se especifica que el valor de configuración debe ser especificado. Por lo que la causa asignable a este defecto corresponde a una falta en la definición de detalle en el requerimiento.

También dentro del *sprint* 120 se origina el defecto *PILOT-10685*, según la descripción de este defecto se solicita hacer un cambio en la documentación del *API*

desarrollada en la historia de usuario *PILOT-9967* ya que el cliente usa dicha documentación para la generación de su código (el que se consumiría dicho *API*). Ya que este cambio fue reportado por el cliente parecería que se reportó de forma errónea ya que esto debería ser considerado una mejora, sin embargo, se debe tomar en cuenta que la documentación fue diseñada bajo un estándar conocido, por lo que realmente la causa asignable a este defecto se atribuye a una falta de investigación sobre cómo se debía elaborar dicha documentación con respecto al uso general que le pueden dar usuarios finales.

Finalmente, como parte del *sprint* 127 se origina el defecto *PILOT-10679* en donde se especifica que el valor de una fecha es incorrecto de acuerdo con el contexto de la información mostrada al usuario. Al revisar el defecto y corregirlo se identifica que el atributo utilizado para obtener esa fecha no era el indicado. Cuando se revisa un defecto asociado (*PILOT-10605*) se observa que el atributo a utilizar nunca se especifica de hecho si se consulta el requerimiento directamente (*PILOT-10552*) no hay información relacionada al valor que se debe utilizar. Al igual que el defecto *PILOT-10221* se le atribuye como causa a la falta en la definición de detalle en el requerimiento.

En resumen, los defectos originados por una definición de requerimientos incompleta se deben a las siguientes causas asignables:

- Falta de revisión de secciones relacionadas cuando se define una regla de negocio.
- No hay una comunicación apropiada dentro del equipo de desarrollo al definir un requerimiento o al reportar un tiquete.
- Investigación inadecuada de estándares que pueden ser utilizados para elaborar documentación usada por el usuario final.
- Falta de definición en los detalles de una historia de usuario.

5.3.3 Plan de mejora ante las causas asignables identificadas

La principal causa asignable encontrada dentro de los resultados obtenidos es la falta de pruebas. A continuación, se presenta un listado de factores relacionados con esta causa:

- **Uso de datos inadecuados:** por ejemplo, usar datos estáticos en una funcionalidad genérica o datos irreales que no representan los valores proporcionados por el usuario final.
- **La no utilización de herramientas que usará el usuario final**: por lo general por reglas de negocio del cliente final o por resistencia al cambio el usuario no suele usar las herramientas más actualizadas como si lo hace los desarrolladores.
- **Información inconsistente:** Al realizar mejoras o cambio en los requerimientos muchas veces la información no suele ser retornada o actualizada en la forma esperada.
- **Cobertura inadecuada:** los casos de prueba diseñados para una funcionalidad suelen ser insuficientes para asegurar la correcta funcionalidad de la funcionalidad en todas sus aristas (diferentes escenarios o funcionalidades relacionadas).

Como una solución a este tipo de problemas se propone el uso de procesos de revisiones técnicas formales que, a diferencia de otros métodos de revisión, suelen ser requeridas para la aprobación del diseño del producto [25]. Se debería llevar a cabo una sesión donde un equipo realice la discusión y la aprobación de cada uno de los casos de prueba relacionados a una historia de usuario o defecto. Esta discusión es dirigida por un líder que tenga experiencia con proyectos similares al proyecto de estudio, a su vez es conveniente que tenga una buena relación con todos los miembros del equipo.

Al iniciar el proceso de una revisión técnica formal, el líder debe elegir a los miembros del equipo que revisará los casos de prueba y a su vez distribuir la revisión de forma equitativa, y debe calendarizar las sesiones de revisión. Por otra parte, el equipo encargado de revisar los casos de prueba debe hacer una revisión de los requerimientos antes de las sesiones de discusión y definir preguntas que generen incertidumbre en lo planteado.

Al llevar a cabo las sesiones de revisión se presenta el diseño de los casos de prueba y cualquier pregunta relacionada a ellos con el fin de decidir si son aprobados o no. Finalmente, se genera un reporte resumen en donde se detalla las conclusiones obtenidas, incluyendo la lista de los ítems que se aprobaron y los ítems que se debieron corregir para lograr su aprobación. En la figura 9 se muestra un ejemplo de una plantilla de un reporte de una revisión formal que se puede utilizar como guía para reportar estos datos.

Design Review Report

DR date: _____ The report was prepared by: _____

Project name: _____

The review document: _____ Version: _____

The review team: _____

1 Summary of the discussions

#	Discussion subject	Number of action items

2 The action items

#	Action items to be performed	Responsible employee	Completion date	Approval of completion	
				Date	Signature

3 Decision regarding the design product

☐ Full approval

☐ Partial approval. Approval granted for continuation to the next phase of the following parts:

☐ Denial of approval

Comments:

The report was approved by:

Name of participant	Date	Signature	Name of participant	Date	Signature

Approval of sucessful completion of all action items

Comments:

Name: Signature: Date:

Figura 9. Plantilla para un reporte de revisión formal (tomado de D. Galin, *Software Quality Assurance, From the theory to the implementation*).

A su vez, se propone utilizar una métrica conocida como la Efectividad de los Casos de Prueba o (*TCE por sus siglas en inglés*) [25], expresada por la siguiente fórmula:

$$\boldsymbol{TCE = \frac{N_{tc}}{N_{tot}} * 100\%}$$

En este caso $\boldsymbol{N_{tc}}$ representa el número de defectos encontrados por un caso de prueba especificado y $\boldsymbol{N_{tot}}$ representa el número de defectos que este caso de prueba no pudo

encontrar. De esta manera se puede mejorar el proceso de definición de casos de prueba y retroalimentar las secciones de revisión formal sugeridas anteriormente.

Otra de las principales causas asignables es la falta de documentación del proyecto. Los siguientes escenarios evidencian que se requiere una mejora en la documentación:

- **Estándares en la mejora de la mantenibilidad**: Algunos de los defectos encontrados se debieron a que no se conocía que partes del proyecto pueden ser o no mejorados para incrementar el índice de mantenibilidad del proyecto.
- **Funcionalidad de módulos existentes:** Algunos de los defectos demuestra que se desconoce cómo funcionan algunas secciones de la aplicación por lo que se define requerimientos incompletos o hay un mal entendimiento al definir una regla de negocio.
- **Requerimientos incompletos:** Para ciertos defectos encontrados se identificó que no había claridad en los requerimientos debido a que no se especificaron las necesidades de un cliente en específico o que los administradores del proyecto y los desarrolladores no establecieron una comunicación clara.

Una solución a este problema es la generación de documentación controlada [24] con una estructura clara de cómo indexar los documentos para su fácil acceso. Además de ello, se debe definir un estándar sobre cómo documentar cada proceso del ciclo de vida del software. Actualmente la organización cuenta con herramientas que recopilan documentos sobre tareas realizadas por los desarrolladores con el fin de ayudar a los otros miembros del equipo, sin embargo, estos documentos son elaborados de manera independiente y por lo general no hay un acuerdo mutuo entre todos los miembros del equipo. Por lo que se recomienda que se definan categorías claras de documentos y que cada una de ellas contenga una guía que le facilite al creador seguir una serie de pasos fácilmente identificable. Por otra parte, la herramienta *JIRA* suele ser la encargada de recopilar la información que describe los requerimientos en su mayoría a través de descripciones breves y criterios de aceptación, sin embargo, no existe una explicación detallada sobre el valor de negocio que representa su trabajo para el usuario final. Debido a esto, se recomienda que estos requerimientos sean traducidos a documentos formales que describan el objetivo de negocio y a su vez las

implicaciones que puede traer sus funcionalidades respectivas a futuro para próximas implementaciones.

Finalmente, la falta de revisiones de código (o *code reviews* en inglés) son una de las causas asignables para algunos de los defectos encontrados por el cliente. Se determinó que errores de sintaxis o uso de operadores incorrectos generaban este tipo de fallas. Por lo general, los desarrolladores utilizan la herramienta de control de versiones *GitHub* para hacer las revisiones. Cabe destacar que esta herramienta solo muestra qué código se agrega o se remueve de un archivo, por lo que la decisión de aprobar o rechazar una revisión se basa en lo que sea mostrado en esta herramienta. Algunos editores de código suelen resaltar errores sintácticos por lo que se aconseja que el código implementado sea revisado en una herramienta de edición de código que sea capaz de resaltar dichos errores.

6. Conclusiones

Este trabajo aplicó la métrica de Efectividad de Remoción de Defectos en el contexto de una organización que utiliza *SCRUM* como metodología de desarrollo. Se analizaron los defectos del año 2019, en donde se descubrió que cerca de la mitad correspondía a defectos introducidos en *sprints* que se ejecutaron en años anteriores. Esto es señal de que los defectos se propagan en gran cantidad a través de varios *sprints*.

Como este estudio se basa en la información de cuál *sprint* origina el defecto, es de suma importancia extraer este dato correctamente. En el proceso de conteo definido en este estudio se conocía la diferencia entre los defectos reportados por el equipo de desarrollo y los defectos reportados por el cliente gracias a la herramienta *JIRA*. Otro factor importante que contribuyó en la realización de este trabajo fue el registro de dos tipos de datos por parte del desarrollador: (1) el *pull request*, que muestra los cambios en el código para corregir un defecto, y (2) la causa raíz que explica por qué ocurrió el defecto. El registro apropiado de los datos de cada defecto es de gran utilidad para determinar su origen mediante juicio experto. Con esto se puede determinar se puede identificar en qué *sprint* se introdujo cada defecto.

Es importante aclarar que, por cada historia de usuario, tarea, o defecto registrado se debe especificar en qué *sprint* se reporta y en cuál se finaliza la tarea, o al menos llevar un control de versiones fielmente asociado a los *sprints*, ya que no serviría de nada tener el código asociado o una explicación de por qué ocurrió el defecto si no se sabe a qué *sprint* pertenece. Una buena práctica encontrada en la organización bajo estudio es que cada *commit* que se hace a la herramienta *GitHub* contiene el identificador del tiquete que se trabajó.

Aplicar el proceso de conteo sobre la información recolectada de la herramienta *JIRA* fue una tarea de cuidado ya que determinar el origen del *sprint* puede se engañoso de acuerdo con la información que se tiene. Una de las amenazas a la validez en este estudio consiste en que los desarrolladores podían no reportar de forma apropiada la resolución del defecto ya que de acuerdo con los requerimientos estos suelen convertirse en mejoras o suelen estar asociados a otros factores. Por lo tanto, se decidió introducir el tipo de origen por cada

defecto, lo que contribuyó de manera considerable a realizar un filtro de los datos para justificar por qué un defecto se encontraba en dicha categoría. Estos tipos introdujeron nuevas clasificaciones para los defectos recolectados: (1) si el problema se debía a un problema de configuración de servidor, (2) si el problema era del lado del cliente, (3) si la afectación la causaba un servicio externo, (4) si el defecto se convirtió en una mejora.

Un dato de gran importancia en este estudio fue saber el *sprint* en que se reportó el defecto ya que este valor ayudó a identificar Regression-Bugs. Por regla general se aplicó que si el *sprint* de origen era el mismo que el *sprint* en el cual se había reportado, se podría afirmar que el defecto era un Regression-Bug.

Al realizar el cálculo del valor del DRE por *sprint,* se pudo generar el gráfico de control XmR para responder a las preguntas ¿Es estable el proceso? y ¿Es el proceso capaz?

La primera pregunta se puede responder con las pruebas de estabilidad que se realiza sobre el gráfico de control. Cualquiera de los puntos que incumpla con alguna de las cuatro pruebas de estabilidad sobre los datos proporcionados representan una inestabilidad en el proceso, por lo que se debe identificar las causas asignables de esa variación. En la organización bajo estudio se determinó que si el cliente encontró dos defectos y el equipo de desarrollo no encontró ninguno, entonces el valor DRE debía ser 0%, por lo que se requería identificar qué factores habían ocasionado esto.

La segunda pregunta de investigación se puede responder utilizando el mismo gráfico de control, ya que para hacer las pruebas se realiza el cálculo de al menos 3 límites que representan grados de desviación sobre el valor promedio. El valor promedio (línea central del gráfico X) representa el DRE promedio del proyecto para el periodo de tiempo analizado (un año en nuestro caso). En la organización bajo estudio se identificó que para el año 2019 el porcentaje era del 70,05%, si se compara este valor con el DRE esperado por Caper Jones [2] se puede afirmar que el proceso no es capaz ya que un valor menor de 85% conlleva una insatisfacción para el cliente.

6.1 Trabajo Futuro

A continuación, presentamos algunas posibles líneas de trabajo futuro, basadas en los resultados de esta investigación.

En primer lugar, como parte del trabajo realizado se hizo la recolección del dato de *sprint* de origen para cada defecto, para lo que se debió analizar el código fuente que reparaba el defecto y la causa raíz documentada por el desarrollador para definir el *sprint* de origen basado en el juicio experto del investigador. Si se utiliza herramientas tales como procesamiento de lenguaje natural, se podría automatizar de cierta manera esta tarea. Herramientas como *JIRA o GitHub* permiten exportar la información relacionada a los tiquetes a través de archivos separados por coma (o *CSV* por sus siglas en inglés), o utilizando su interfaz de programación de aplicaciones (o *API* por sus siglas en inglés).

También, durante la recolección de los datos se utilizó la información del año 2019 exclusivamente. Para un trabajo a futuro se podría aplicar la métrica DRE desde el comienzo del proyecto utilizando un lapso similar de tiempo con el fin de corroborar si existe un comportamiento distinto.

Por otra parte, si la organización decide implementar las propuestas de mejora sugeridas en este estudio, sería interesante evaluar la efectividad de las mejoras implementadas en el proceso midiendo el cambio en el DRE de la organización.

Dentro del plan de mejora se propone el uso de una nueva métrica llamada Efectividad de los Casos de Prueba, la que evalúa la efectividad de un caso de prueba en específico para encontrar defectos para una historia de usuario. Un trabajo a futuro podría estudiar si existe una correlación entre esta métrica con los casos de pruebas diseñados para las historias de usuario de un *sprint*. Por ejemplo, si un *sprint* reporta un alto número de defectos con respecto a uno o varios casos de prueba en específico esto puede significar que dichos casos no fueron bien diseñados y se debe analizar que defectos no podrían ser cubiertos.

Dentro del análisis de control estadístico se determinó que uno de los puntos inestables se identificó porque se encontraba por debajo del limite de control inferior, de esto se infiere que el porcentaje de DRE corresponde al 0%. En este caso pudieron existir

otras causas además de las ya encontradas, entre ellas se puede deber a un incumplimiento de los principios de *SCRUM*, ya que podría existir una sobrecarga de trabajo que no permite encontrar los defectos a tiempo. Por esta razón, se propone estudiar si existe una correlación entre la métrica DRE y el número de *story-points* completados por sprint, para así determinar si el completar muchas tareas afecta de forma negativa al porcentaje de DRE de un *sprint*.

Por otra parte, el valor de DRE puede no ser un reflejo de la satisfacción de la organización bajo estudio. Kuruba [21] indica que el porcentaje de DRE puede ser el mejor, pero si el cliente no se encuentra satisfecho esto no tendrá valor para la organización, por lo que se propone introducir un valor que le otorgue peso a los defectos y así identificar si el porcentaje de DRE resulta ser inapropiado en relación con defectos que son de prioridad para la organización.

Bibliografía

1. S. Kan, Metrics and Models in Software Quality Engineering, 2nd ed: Addison Wesley, 2002.

2. C. Jones, "Software Quality Metrics: Three Harmful Metrics and Two Helpful Metrics", Namcook Analytics LLC, 2012.

3. H. K. N. Leung, "Improving defect removal effectiveness for software development," Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, Florence, Italy, 1998, pp. 157-164, doi: 10.1109/CSMR.1998.665789.

4. S. Kumaresh and R. Baskaran, "Experimental design on defect analysis in software process improvement," International Conference on Recent Advances in Computing and Software Systems, Chennai, 2012, pp. 293-298, doi: 10.1109/RACSS.2012.6212683.

5. Y. Yamato, S. Katsuragi, S. Nagao, N. Miura, "Software Maintenance Evaluation of Agile Software Development Method Based on OpenStack," IEICE Transactions on Information and Systems, 2015.

6. D. Castro, M. Jenkins, "Aplicación de la métrica de efectividad de remoción de defectos en una metodología ágil," Universidad de Costa Rica, 2013.

7. C. Jones, A Guide to selecting software measures and metrics, CRC Press, 2017

8. K. Schewaber, J Sutherland, "The Scrum Guide", 2020. Disponible: Scrum Guides, https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf#zoom=100

9. A. Carleton, W. Florac, P. Robert, Practical Software Measurement: Measuring for Process Management and Improvement: Carnegie Menllon University, 1997.

10. H. K. N. Leung and L. White, "Insights into regression testing (software testing)," Proceedings. Conference on Software Maintenance, 1989, pp. 60-69.

11. Atlassian, “Jira overview: Products, projects and hosting”, *Atlassian*. Disponible: https://www.atlassian.com/software/jira/guides/getting-started/overview.

12. Atlassian, “What is a Jira Software project?”, *Atlassian*. Disponible: https://support.atlassian.com/jira-software-cloud/docs/what-is-a-jira-software-project/.

13. Atlassian, “What is an issue?”, *Atlassian*. Disponible: https://support.atlassian.com/jira-software-cloud/docs/what-is-an-issue/.

14. Atlassian, “Add files, images, and other content to describe an issue”, *Atlassian*. Disponible: https://support.atlassian.com/jira-software-cloud/docs/add-files-images-and-other-content-to-describe-an-issue/.

15. Atlassian, “Work with issue workflows”, *Atlassian*. Disponible: https://support.atlassian.com/jira-cloud-administration/docs/work-with-issue-workflows/.

16. GitHub, “¿Qué es GitHub? Una GUÍA para Principiantes SOBRE GITHUB,” *Kinsta*, 2020. Disponible: https://kinsta.com/es/base-de-conocimiento/que-es-github/.

17. N. Nurlisa, A. Ngah, A. Deraman, “Version Control System: A Review”, Procedia Computer Science, 2018.

18. S. Chacon, B. Straub, “Pro Git”, Apress, 2021.

19. GitHub, “About Pull Requests”, Github, Disponible: https://docs.github.com/es/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests.

20. R. Andrade, I. Santos, V. Lell, K. Oliveira, A. Rocha, "Software Testing Process in a Test Factory," Ad hoc Activities to an Organizational Standard, 2017.

21. M. Kuruba, “Defect removal effectiveness and defect removal efficiency – a refined approach”, Japan SEPG, 2005.

22. J. Ba and S. Wu, "SdDirM: A dynamic defect prediction model," Proceedings of 2012 IEEE/ASME 8th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications, 2012, pp. 252-256, doi: 10.1109/MESA.2012.6275570.

23. Runeson, P., Höst, M. “Guidelines for conducting and reporting case study research in software engineering”. *Empir Software Eng* 2009, 14**,** 131.

24. Y. Chernak, "Validating and improving test-case effectiveness," in IEEE Software, vol. 18, no. 1, pp. 81-86, Jan.-Feb. 2001, doi: 10.1109/52.903172.

25. D. Galin, Software Quality Assurance, From the theory to the implementation: Pearson Education Limited, 2004.

Anexos

Anexo 1 Defectos encontrados dentro de las categorías *Regression-Bug* y *Bug*

Defecto	Reportado en *Sprint*	Fecha de Creación	*Sprint* de origen	Tipo de origen	Tiquete Relacionado	Problema en la configuración del servidor	Problema del lado del cliente	Relacionado a un servicio externo	Defecto de Regresión	Convertido en una mejora
PILOT-10699	Sprint 130	18/12/19	Sprint 122	Mala implementación del código	PILOT-10230	No	No	No	No	No
PILOT-10685	Sprint 129	10/12/19	Sprint 120	Implementación incompleta	PILOT-10128	No	No	No	No	No
PILOT-10683	Sprint 129	9/12/19	Sprint 101 (Anterior)	Mala implementación del código	PILOT-8533	No	No	No	No	No
PILOT-10679	Sprint 129	6/12/19	Sprint 127	Definición de requerimientos incompleta	PILOT-10605	No	No	No	No	No
PILOT-10621	Sprint 128	19/11/19	Sprint 103 (Anterior)	Implementación incompleta	PILOT-9158	No	No	No	No	No
PILOT-10616	Sprint 128	15/11/19	Sprint 126	Mala implementación del código	PILOT-10482	No	No	No	No	No
PILOT-10607	Sprint 127	8/11/19	Sprint 117	Corrección Defectuosa	PILOT-9719	No	No	No	No	No
PILOT-10605	Sprint 127	6/11/19	Sprint 127	Implementación incompleta	PILOT-10552	No	No	No	Si	No
PILOT-10600	Sprint 127	5/11/19	Sprint 96 (Anterior)	Implementación incompleta	PILOT-8617	No	No	No	No	No
PILOT-10598	Sprint 127	4/11/19				No	**No**	Si	No	No
PILOT-10574	Sprint 126	28/10/19	Sprint 115	Implementación incompleta	PILOT-9660	No	No	No	No	No
PILOT-10567	Sprint 126	23/10/19				No	**No**	Si	No	No
PILOT-10566	Sprint 132	23/10/19	Sprint 97 (Anterior)	Problemas posteriores a actualización	PILOT-7858	No	No	No	No	No

PILOT-10562	Sprint 126	18/10/19	Sprint 124	Mala implementación del código	PILOT-10439	No	No	No	No	No
PILOT-10561	Sprint 126	18/10/19	Sprint 127	Implementación incompleta	PILOT-10386	No	No	No	Si	No
PILOT-10506	Sprint 125	7/10/19	Sprint 26 (Anterior)	Mala implementación del código	PILOT-1598	No	No	No	No	No
PILOT-10492	Sprint 125	2/10/19	Sprint 75 (Anterior)	Mala implementación del código	PLOT-6410	No	No	No	No	No
PILOT-10462	Sprint 124	23/9/19	Sprint 45 (Anterior)	Mala implementación del código	PILOT-3019	No	No	No	No	No
PILOT-10436	Sprint 124	19/9/19	Sprint 124	Corrección Defectuosa	PILOT-10412	No	No	No	Si	No
PILOT-10384	Sprint 123	10/9/19	Sprint 123	Corrección Defectuosa	PILOT-10326	No	No	No	Si	No
PILOT-10383	Sprint 123	10/9/19	Sprint 119	Definición de requerimientos incompleta	PILOT-10012	No	No	No	No	No
PILOT-10326	Sprint 122	29/8/19	Sprint 39 (Anterior)	Mala implementación del código	PILOT-2692	No	No	No	No	No
PILOT-10302	Sprint 122	26/8/19	Sprint 117	Corrección Defectuosa	PILOT-9860	No	No	No	No	No
PILOT-10292	Sprint 122	22/8/19	Sprint 109	Mala implementación del código	PILOT-9367	No	No	No	No	No
PILOT-10285	Sprint 122	21/8/19	Sprint 97 (Anterior)	Corrección Defectuosa	PILOT-8648	No	No	No	No	No
PILOT-10245	Sprint 121	13/8/19	Sprint 119	Definición de requerimientos incompleta	PILOT-10075	No	No	No	No	No
PILOT-10243	Sprint 121	13/8/19	Sprint 118	Mala implementación del código	PILOT-9894	No	No	No	No	No

PILOT-10221	Sprint 121	9/8/19	Sprint 120	Definición de requerimientos incompleta	PILOT-10055	No	No	No	No	No
PILOT-10218	Sprint 122	9/8/19	Sprint 57 (Anterior)	Mala implementación del código	PILOT-3804	No	No	No	No	No
PILOT-10216	Sprint 121	8/8/19	Sprint 118	Corrección Defectuosa	PILOT: 9894	No	No	No	No	No
PILOT-10215	Sprint 121	8/8/19	Sprint 118	Corrección Defectuosa	PILOT-9960	No	No	No	No	No
PILOT-10209	Sprint 122	7/8/19	Sprint 122	Implementación incompleta	PILOT-10263	No	No	No	Si	No
PILOT-10176	Sprint 120	5/8/19	Sprint 79 (Anterior)	Corrección Defectuosa	PILOT-7043	No	No	No	No	No
PILOT-10170	Sprint 121	5/8/19	Sprint 79 (Anterior)	Problemas posteriores a actualización	PILOT-6566	No	No	No	No	No
PILOT-10165	Sprint 122	2/8/19	Sprint 116	Mala implementación del código	SP-264	No	No	Si	No	No
PILOT-10115	Sprint 119	24/7/19	Sprint 119	Mala implementación del código	PILOT-10012	No	No	No	Si	No
PILOT-10105	Sprint 119	23/7/19	Sprint 118	Mala implementación del código	PILOT-9967	No	No	No	No	No
PILOT-10045	Sprint 119	15/7/19	Sprint 117	Corrección Defectuosa	PILOT-9719	No	No	No	No	No
PILOT-10020	Sprint 118	8/7/19	Sprint 118	Implementación incompleta	PILOT-9968	No	No	No	Si	No
PILOT-10011	Sprint 122	4/7/19	Sprint 99 (Anterior)	Implementación incompleta	PILOT-8712	No	No	No	No	No
PILOT-9984	Sprint 121	28/6/19	Sprint 62 (Anterior)	Mala implementación del código	PILOT-4901	No	No	No	No	No
PILOT-9982	Sprint 118	27/6/19	Sprint 26	Implementación incompleta	PILOT-1528	No	No	No	No	No

			(Anterior)							
PILOT-9981	Sprint 121	27/6/19	Sprint 41 (Anterior)	Implementación incompleta	PILOT-2703	No	No	No	No	No
PILOT-9955	Sprint 117	25/6/19	Sprint 90 (Anterior)	Mala implementación del código	PILOT-8132	No	No	No	No	No
PILOT-9945	Sprint 117	21/6/19	Sprint 117	Corrección Defectuosa	PILOT-9719	No	No	No	Si	No
PILOT-9936	Sprint 120	19/6/19	Sprint 55 (Anterior)	Mala implementación del código	PILOT-4001	No	No	No	No	No
PILOT-9921	Sprint 117	12/6/19	Sprint 107	Definición de requerimientos incompleta	PILOT-9095	No	No	No	No	No
PILOT-9895	Sprint 117	7/6/19	Sprint 96 (Anterior)	Mala implementación del código	PILOT-8617	No	No	No	No	No
PILOT-9890	Sprint 117	5/6/19				No	No	No	No	Si
PILOT-9883	Sprint 117	4/6/19	Sprint 116	Mala implementación del código	PILOT-9781	No	No	No	No	No
PILOT-9882	Sprint 116	4/6/19				No	No	No	No	Si
PILOT-9873	Sprint 116	3/6/19	Sprint 116	Mala implementación del código	PILOT-9780	No	No	No	Si	No
PILOT-9860	Sprint 116	29/5/19	Sprint 113	Definición de requerimientos incompleta	PILOT-9523	No	No	No	No	No
PILOT-9851	Sprint 118	28/5/19				No	No	No	No	Si
PILOT-9849	Sprint 116	27/5/19	Sprint 114	Mala implementación del código	SP-155	No	No	Si	No	No
PILOT-9816	Sprint 116	22/5/19	Sprint 116	Corrección Defectuosa	PILOT-8885	No	No	No	Si	No
PILOT-9805	Sprint 115	21/5/19	Sprint 115	Mala implementación del código	PILOT-9661	No	No	No	Si	No

PILOT-9802	Sprint 115	20/5/19	Sprint 97 (Anterior)	Problemas posteriores a actualización	PILOT-7858	No	No	No	No	No
PILOT-9778	Sprint 116	15/5/19	Sprint 112	Mala implementación del código	PILOT-9512	No	No	No	No	No
PILOT-9753	Sprint 115	13/5/19	Sprint 97 (Anterior)	Problemas posteriores a actualización	PILOT-7858	No	No	No	No	No
PILOT-9751	Sprint 115	10/5/19	Sprint 114	Mala implementación del código	PILOT-9591	No	No	No	No	No
PILOT-9750	Sprint 115	10/5/19				No	No	No	No	Si
PILOT-9738	Sprint 115	9/5/19	Sprint 113	Problemas posteriores a actualización	PILOT-9390	No	No	No	No	No
PILOT-9727	Sprint 114	7/5/19	Sprint 106	Corrección Defectuosa	PILOT-9279	No	No	No	No	No
PILOT-9725	Sprint 114	6/5/19	Sprint 101 (Anterior)	Mala implementación del código	PILOT-8995	No	No	No	No	No
PILOT-9696	Sprint 116	2/5/19	Sprint 82 (Anterior)	Mala implementación del código	PILOT-7298	No	No	No	No	No
PILOT-9683	Sprint 114	30/4/19	Sprint 100 (Anterior)	Mala implementación del código	PILOT-8188	No	No	No	No	No
PILOT-9682	Sprint 114	30/4/19				No	Si	No	No	No
PILOT-9664	Sprint 114	25/4/19	Sprint 114	Implementación incompleta	PILOT-9576	No	No	No	Si	No
PILOT-9663	Sprint 114	25/4/19	Sprint 113	Corrección Defectuosa	PILOT-9623	No	No	No	No	No

PILOT-9662	Sprint 114	24/4/19	Sprint 55 (Anterior)	Corrección Defectuosa	PILOT-3563	No	No	No	No	No
PILOT-9652	Sprint 114	24/4/19	Sprint 114	Mala implementación del código	AUTH-39	No	No	Si	Si	No
PILOT-9647	Sprint 114	24/4/19	Sprint 101 (Anterior)	Mala implementación del código	PILOT-8995	No	No	No	No	No
PILOT-9635	Sprint 113	18/4/19	Sprint 109	Mala implementación del código	PILOT-9393	No	No	No	No	No
PILOT-9634	Sprint 113	17/4/19	Sprint 113	Implementación incompleta	PILOT-9620	No	No	No	Si	No
PILOT-9625	Sprint 113	10/4/19	Sprint 112	Corrección Defectuosa	PILOT-9539	No	No	No	No	No
PILOT-9609	Sprint 112	5/4/19	Sprint 100 (Anterior)	Implementación incompleta	PILOT-8574	No	No	No	No	No
PILOT-9576	Sprint 113	28/3/19	Sprint 65 (Anterior)	Implementación incompleta	PILOT-4398	No	No	No	No	No
PILOT-9569	Sprint 113	27/3/19	Sprint 74 (Anterior)	Corrección Defectuosa	PILOT-6267	No	No	No	No	No
PILOT-9568	Sprint 112	27/3/19	Sprint 105	Mala implementación del código	PILOT-9211	No	No	No	No	No
PILOT-9550	Sprint 111	20/3/19	Sprint 111	Mala implementación del código	PILOT-8388	No	No	No	Si	No
PILOT-9549	Sprint 112	19/3/19	Sprint 110	Corrección Defectuosa	PILOT-9516	No	No	No	No	No
PILOT-9536	Sprint 111	13/3/19	Sprint 111	Mala implementación del código	PILOT-9418	No	No	No	Si	No
PILOT-9534	Sprint 111	12/3/19	Sprint 110	Mala implementación del código	PILOT-9458	No	No	No	No	No
PILOT-9532	Sprint 110	12/3/19	Sprint 110	Mala implementación del código	PILOT-9480	No	No	No	Si	No

PILOT-9531	Sprint 111	11/3/19	Sprint 56 (Anterior)	Mala implementación del código	PILOT-4535	No	No	No	No	No
PILOT-9516	Sprint 110	6/3/19	Sprint 75 (Anterior)	Mala implementación del código	PILOT-6498	No	No	No	No	No
I	Sprint 113	5/3/19				No	Si	No	No	No
PILOT-9503	Sprint 111	4/3/19	Sprint 76 (Anterior)	Mala implementación del código	PILOT-6688	No	No	No	No	No
PILOT-9500	Sprint 110	4/3/19	Sprint 109	Mala implementación del código	PILOT-9401	No	No	No	No	No
PILOT-9487	Sprint 110	27/2/19				Si	No	No	No	No
PILOT-9486	Sprint 113	27/2/19	Sprint 106	Mala implementación del código	PILOT-9240	No	No	No	No	No
PILOT-9478	Sprint 109	26/2/19	Sprint 104 (Anterior)	Mala implementación del código	PILOT-9213	No	No	No	No	No
PILOT-9477	Sprint 109	26/2/19	Sprint 109	Mala implementación del código	PILOT-9405	No	No	No	Si	No
PILOT-9471	Sprint 113	25/2/19	Sprint 48 (Anterior)	Corrección Defectuosa	PILOT-3572	No	No	No	No	No
PILOT-9468	Sprint 109	21/2/19	Sprint 109	Mala implementación del código	PILOT-9095	No	No	No	Si	No
PILOT-9467	Sprint 109	21/2/19				No	No	No	No	Si
PILOT-9462	Sprint 110	20/2/19	Sprint 108	Implementación incompleta	PILOT-9322	No	No	No	No	No
PILOT-9433	Sprint 109	13/2/19	Sprint 108	Implementación incompleta	PILOT-9321	No	No	No	No	No

PILOT-9432	Sprint 115	13/2/19	Sprint 104 (Anterior)	Mala implementación del código	PILOT-9213	No	No	No	No	No
PILOT-9389	Sprint 109	4/2/19				No	No	No	No	Si
PILOT-9385	Sprint 108	31/1/19	Sprint 107	Mala implementación del código	PILOT-9284	No	No	No	No	No
PILOT-9381	Sprint 108	30/1/19	Sprint 104 (Anterior)	Mala implementación del código	PILOT-6640	No	No	No	No	No
PILOT-9374	Sprint 107	25/1/19	Sprint 104 (Anterior)	Implementación incompleta	PILOT-9055	No	No	No	No	No
PILOT-9364	Sprint 111	24/1/19	Sprint 72 (Anterior)	Mala implementación del código	PILOT-6204	No	No	No	No	No
PILOT-9356	Sprint 112	23/1/19				Si	No	No	No	No
PILOT-9329	Sprint 107	15/1/19	Sprint 26 (Anterior)	implementación incompleta	PILOT-1524	No	No	No	No	No
PILOT-9306	Sprint 106	3/1/19	Sprint 100 (Anterior)	Implementación incompleta	PILOT-8894	No	No	No	Si	No
PILOT-9302	Sprint 105	2/1/19	Sprint 96 (Anterior)	Mala implementación del código	No Disponible	No	No	No	No	No

Anexo 2 Defectos encontrados dentro de la categoría *Story-Bug*

Defecto	***Sprint***	**Fecha de creación**
PILOT-9301	Sprint 105	2/1/19
PILOT-9343	Sprint 107	17/1/19
PILOT-9363	Sprint 107	24/1/19
PILOT-9397	Sprint 108	5/2/19
PILOT-9450	Sprint 109	15/2/19
PILOT-9452	Sprint 109	15/2/19
PILOT-9455	Sprint 109	18/2/19
PILOT-9464	Sprint 109	20/2/19
PILOT-9501	Sprint 110	4/3/19
PILOT-9518	Sprint 110	7/3/19
PILOT-9529	Sprint 110	11/3/19
PILOT-9600	Sprint 112	3/4/19
PILOT-9610	Sprint 112	5/4/19
PILOT-9613	Sprint 112	8/4/19
PILOT-9624	Sprint 113	10/4/19
PILOT-9675	Sprint 114	29/4/19
PILOT-9674	Sprint 114	29/4/19
PILOT-9676	Sprint 114	29/4/19
PILOT-9684	Sprint 114	30/4/19
PILOT-9692	Sprint 114	1/5/19
PILOT-9801	Sprint 115	20/5/19
PILOT-9804	Sprint 115	20/5/19
PILOT-9824	Sprint 115	23/5/19
PILOT-9844	Sprint 115	24/5/19
PILOT-9852	Sprint 116	28/5/19
PILOT-9853	Sprint 116	28/5/19
PILOT-9854	Sprint 116	28/5/19
PILOT-9855	Sprint 116	28/5/19
PILOT-9861	Sprint 116	29/5/19
PILOT-9862	Sprint 116	29/5/19
PILOT-9935	Sprint 117	19/6/19
PILOT-9942	Sprint 117	21/6/19
PILOT-9946	Sprint 117	24/6/19
PILOT-10010	Sprint 118	4/7/19
PILOT-10019	Sprint 118	5/7/19

PILOT-10075	Sprint 119	18/7/19
PILOT-10106	Sprint 119	23/7/19
PILOT-10120	Sprint 120	26/7/19
PILOT-10160	Sprint 120	1/8/19
PILOT-10183	Sprint 120	6/8/19
PILOT-10286	Sprint 122	22/8/19
PILOT-10289	Sprint 121	22/8/19
PILOT-10364	Sprint 122	4/9/19
PILOT-10391	Sprint 122	11/9/19
PILOT-10444	Sprint 123	20/9/19
PILOT-10525	Sprint 124	10/10/19
PILOT-10526	Sprint 124	10/10/19
PILOT-10536	Sprint 124	15/10/19
PILOT-10557	Sprint 125	17/10/19
PILOT-10580	Sprint 126	30/10/19
PILOT-10581	Sprint 126	31/10/19
PILOT-10582	Sprint 126	31/10/19
PILOT-10583	Sprint 126	31/10/19
PILOT-10609	Sprint 125	11/11/19
PILOT-10659	Sprint 126	3/12/19