Universidad de Medellín

Modelo para el tratamiento de la deuda técnica orientado a la evolución de los componentes para que las aplicaciones sean sostenibles a largo plazo

Tesis para optar el Grado Académico de Maestría en Ingeniería de Software

Presentada por:

Daniel Arafat Torres Ruiz

Director:

Jesús Andrés Hincapié Londoño

Medellín – Colombia

2019

Contenido

1.

Capítulo I

Resumen

La Deuda Técnica es un fenómeno habitual del proceso de construcción de software, la cual es reconocida como un conjunto de malas prácticas y decisiones incorrectas en la fase de construcción de software, aceptada generalmente como efecto colateral en proyectos de software donde puede existir presiones de cronograma y con entregas continuas de valor a corto plazo. Ciertos aspectos de este fenómeno son conocidos de forma general, lo que sigue sin conocerse en profundidad, es cómo la deuda técnica se manifiesta y afecta específicamente los procesos de software, y cómo las técnicas de desarrollo de software empleadas acomodan o mitigan la presencia de esta deuda (Holvitie et al., 2018).

La deuda técnica ha ganado visibilidad en los últimos años debido al interés en los proyectos de software que usan marcos de desarrollo ágil, los cuales detallan la responsabilidad que tienen los equipos de desarrollo en producir código de calidad, y que en su afán de entregar software funcionando en el menor tiempo posible, renuncian a actividades relacionadas con la producción de código de calidad, como son, el uso de buenas prácticas de codificación, definiciones de diseños y arquitecturas incompletas, poca cobertura de pruebas y documentación, entre otros, que a largo plazo acumulan una gran cantidad de deuda técnica que se vuelve perjudicial para el éxito de los proyectos. Parte de este estudio es comprender los antecedentes, dimensiones, atributos y consecuencias de la deuda técnica, para así poder proponer un modelo de referencia para el tratamiento de la deuda técnica, y cómo puede éste ser adaptado como parte de un proceso de madurez similar a los existentes en desarrollo de software general.

El entendimiento de las causas para lograr que las aplicaciones puedan ser sostenibles a largo plazo, será el principal enfoque para la selección de las principales técnicas para el tratamiento de la deuda técnica definidas en la literatura, las tecnologías y herramientas que serán útiles para registrar y hacer las mediciones de deuda técnica, especificar qué actividades y ejemplos claros de cómo deberían ser implementadas según el modelo por parte de los equipos de desarrollo, contribuyendo a reducir y gestionar la deuda técnica en las aplicaciones. El modelo propuesto debe proporcionar un enfoque útil para comprender y gestionar la deuda técnica con fines prácticos, que involucre futuras líneas de investigación.

Palabras Claves: Deuda Técnica, Desarrollo Ágil, Refactorización, Diseño de Software, Código Limpio, Calidad de Código

1.1 Abstract

Technical Debt is a common phenomenon of the software construction process, the quality is recognized as a set of bad practices and incorrect decisions in the software construction phase, normally accepted as a side effect in software projects where there may be schedule difficulties and with continuous deliveries of short-term value. Certain aspects of this phenomenon are generally known, what remains unknown in depth, is how technical debt manifests and specifically affects software processes, and how the software development techniques used accommodate or mitigate the presence of this debt (Holvitie et al., 2018).

Technical debt has gained visibility in recent years due to interest in software projects that use agile development frameworks, which detail the responsibility that development teams have in producing quality code, and that in their eagerness to deliver software functioning in the shortest possible time, they give up activities related to the production of quality code, such as the use of good coding practices, definitions of incomplete designs and architectures, little evidence and documentation coverage, among others, that over term accumulate a large amount of technical debt that becomes detrimental to the success of the projects. Part of this study is to understand the background, dimensions, attributes and consequences of technical debt, in order to propose a reference model for the treatment of technical debt, and how it can be adapted as part of a maturity process similar to those existing in general software development.

Understanding the causes to ensure that applications can be sustainable in the long term will be the main approach for the selection of the main techniques for the treatment of technical debt defined in the literature, the technologies and tools that will be useful for recording and make the measurements of technical debt, specify what activities and clear examples of how they should be implemented according to the model by the development teams, contributing to reduce and manage the technical debt in the applications. The proposed model should provide a useful approach to understand and manage technical debt for practical purposes, which involves future lines of research.

Keywords: Technical Debt, Agile Development, Refactoring, Software Design, Clean Code, Code Quality

1.2 Agradecimientos

Agradezco a mi familia por todo su apoyo durante estos años de estudio y por los fines de semana sacrificados.

A mi grupo de estudio y docentes de la Maestría en Ingeniería de Software, por el tiempo compartido y los conocimientos aportados.

A mi Director de Tesis, Prof. Jesús Andrés Hincapié Londoño, por su tiempo, consejos y revisiones para que este trabajo llegue a buen puerto.

Dedicado a mi esposa Edelsy Carreño y mi hija Ariana Lucía Torres Carreño

2.

Capítulo II

Introducción

La metáfora de la Deuda Técnica (DT) o "*Technical Debt*" en inglés, describe un fenómeno en el cual los problemas de calidad técnica en un sistema de software pueden conducir a problemas futuros si no se resuelven inmediatamente. Es un medio para comunicar los desafíos que surgen de las consecuencias del software mal desarrollado para los interesados no técnicos y que puede expresarse usando la siguiente analogía: "Enviar el código de primera vez es como endeudarse. Una pequeña deuda acelera el desarrollo siempre y cuando se devuelva rápidamente con una reescritura. Los objetos hacen que el costo de esta transacción sea tolerable. El peligro ocurre cuando la deuda no se paga. Cada minuto gastado en un código no muy correcto cuenta como intereses en esa deuda" (Cunningham, 1992).

Este trabajo de investigación surge de la necesidad de gestionar problemas relacionados con la calidad del código en los proyectos de software y cómo estos problemas afectan la mantenibilidad y tiempo de vida de las aplicaciones. El principal enfoque del modelo propuesto es la definición de actividades y tareas para el tratamiento de la DT, que apoye distintos frentes en la mejora de la calidad de código y ayude al equipo de desarrollo en mejorar sus competencias.

La hipótesis principal de investigación ha sido formulada de la siguiente manera: "La implementación de un modelo para el tratamiento de la deuda técnica orientado a la evolución de los componentes, puede reducir defectos de calidad en el código de éstos, haciéndolos sostenibles a largo plazo a diferencia de la implementación de los métodos actuales de tratamiento de DT en proyectos de desarrollo".

2.1 Justificación

En la industria del desarrollo de software y la academia están constantemente examinando nuevas alternativas que ayuden a desarrollar software de mayor calidad y excelencia técnica, definiendo así nuevos paradigmas y prácticas que sumen alternativas para llevar a cabo y de mejor manera los nuevos retos del mercado. La deuda técnica es un término que se ha utilizado para describir el costo creciente de cambiar o mantener un sistema debido a los atajos que se tomaron durante su desarrollo (Alves, Ribeiro, Caires, Mendes, & Spínola, 2014). La ausencia de una definición clara y un modelo de deuda técnica agrava el desafío de su identificación y gestión adecuada. Si bien la deuda técnica se discute fácilmente en la

industria, existe una brecha de conocimiento evidente entre la academia y la industria, acerca de la naturaleza de la deuda técnica (Tom, Aurum, & Vidgen, 2013).

El impacto esperado de este trabajo de investigación es el desarrollo de un modelo que permita la gestión de la deuda técnica en los proyectos de software, que valore principalmente los atributos de calidad de código, diseño y arquitectura, garantizando la implementación de técnicas de programación e ingeniería de software, de manera que los desarrolladores utilicen el modelo para debatir y dirigir el modo actual en cómo construyen software, descubriendo los efectos del modelo propuesto para el tratamiento de la deuda técnica en las aplicaciones. El escenario ideal como resultado de la implementación del modelo, serían proyectos de software con diseños óptimos para el problema a solucionar, con buena cobertura de pruebas y código escrito de una manera limpia y legible, con uso de buenas prácticas de codificación y el aprendizaje de conceptos claves de paradigmas de programación como lo es el orientado a objetos para la gestión de la deuda técnica. Los beneficiarios serán los desarrolladores de aplicaciones, que obtendrán un conocimiento para poder dar solución a las malas prácticas de codificación, que a su vez genera altos niveles de DT y reducir de una manera óptima los indicadores de DT, logrando así que su conocimiento también sea compartido con otros profesionales.

2.2 Problema

Desde los inicios del desarrollo de software han existido problemas con la evolución y mantenibilidad de las aplicaciones, de este modo la noción de deuda técnica siempre ha existido, pero no poseía un nombre preciso y era difícil de hablar y aceptar por parte del personal técnico y personal comercial. Comprender la deuda técnica tanto desde la perspectiva teórica como práctica es importante para avanzar en el estado del arte de este concepto (Kruchten, 2016). A medida que las empresas de desarrollo de software adquieren más experiencia en la ejecución de proyectos de desarrollo ágil, se ha consolidado la importancia de brindar las formaciones, herramientas y conocimientos para construir código de mayor calidad por parte de los equipos de desarrollo, teniendo así muchas más posibilidades de éxito en el producto software y logrando el cumplimiento de las metas de todos los interesados (cliente, usuarios, fabricas software, entre otros).

Inicialmente la deuda técnica era exclusiva a temas de malas prácticas de código, pero en la actualidad ha evolucionado para incorporar diferentes aspectos del desarrollo de software que van desde los problemas arquitectónicos, de diseño hasta la documentación, así como las pruebas, las personas y las cuestiones de despliegue (Behutiye, Rodríguez, Oivo, & Tosun, 2017). En la actualidad, los proyectos de software (principalmente los que utilizan una metodología de desarrollo ágil) dedican espacios para revisar el uso de buenas prácticas de codificación y la gestión de deuda técnica de sus proyectos, pero muchas veces estas son ignoradas o “perdonadas” por los involucrados en el desarrollo, debido a temas

de priorización o negligencia. Una gestión de la deuda técnica ineficiente conlleva a males del software que son recurrentes, como los retrocesos en la entrega de productos de valor, errores en la aplicación, mal rendimiento o lentitud de las aplicaciones, aumento de los costes, etc. Otro problema que cada vez se consolida más en el área de conocimiento del Mantenimiento de Software y de la Ingeniería de Software en general, es que las aplicaciones sean capaces de cumplir un tiempo de uso acorde al costo y expectativa del cliente, que pueda respaldar todo lo relacionado al retorno de la inversión y recaudación de utilidades económicas, así como garantizar un producto software capaz de soportar futuras modificaciones, implementaciones de nuevas bibliotecas o librerías, tener un aceptable grado de escalabilidad y otras características que se pueden lograr a través de una gestión de los distintos tipos de deuda técnica y demostrando buenos resultados a los actores participantes del desarrollo, en construcción y mantenimiento de las aplicaciones. Las problemáticas más importantes reconocidas en la literatura se relacionan con los siguientes cuestionamientos:

1) ¿Cuál es la importancia del concepto de Deuda Técnica en el desarrollo de software ágil?

El interés de desarrollar esta problemática de investigación radica en establecer el área de investigación en proyectos y equipos de software con marcos de desarrollo agiles, que quieran realizar mediciones de la cual es su deuda técnica, y de la importancia significativa que adquiere su gestión a medida que los sistemas crecen y evolucionan, de cómo su arquitectura puede degradarse, aumentando los costos de mantenimiento y reduciendo la productividad del desarrollador, provocando demoras en los tiempos de entrega, aumento de los costos, dificultad para realizar futuras adaptaciones con otros sistemas, entre otras repercusiones (MacCormack & Sturtevant, 2016). Por consiguiente, tener una buena gestión de la DT en los equipos de trabajo ayuda a que las probabilidades de éxito de los proyectos sean mayores y que no se vean afectados los tiempos de entrega y costos del desarrollo.

2) ¿Qué tipo de decisiones son tomadas en el desarrollo para que haya efectos negativos causados por la Deuda Técnica?

Las malas decisiones que se toman en el ciclo de desarrollo del software afectan las futuras actividades y a menudo el software disminuye en calidad a lo largo del tiempo, en gran parte debido a las decisiones que tienen que ver con desarrollar una solución suficientemente buena (es decir, comprometer la calidad del software para atender las demandas del mercado a corto plazo) o una solución de calidad que lleva más tiempo implementar (Guo, Seaman, & da Silva, 2016).

Aquellas decisiones aíslan las revisiones y pruebas para asegurar un código de calidad, o evitar las buenas prácticas de desarrollo en la búsqueda de desarrollar en menor tiempo, se refieren a aspectos técnicos del sistema que involucran a personal técnico y no técnico, quienes deben considerar los beneficios y consecuencias que puede causar para el negocio

este tipo de decisiones en un futuro, sin embargo, la visualización de tales consecuencias en el momento de la decisión es difícil (Guo et al., 2016). Además, los desarrolladores, líderes de proyectos y personal de negocio, suelen tener diferentes perspectivas sobre las inversiones en calidad de software, cuando el rendimiento a corto plazo no es visible. Es importante resaltar que aun realizando sacrificios en aspectos de calidad de código y teniendo altos índices de deuda técnica en los proyectos, se pueden obtener beneficios con respecto a entregas rápidas de software y costos (Guo et al., 2016).

3) ¿Cuáles son las causas comunes para que proyectos de software alcancen indicadores significativos de una alta Deuda Técnica?

Comprender los motivos de la toma de decisiones en los equipos de trabajo que incurren en una mala gestión de la deuda técnica y que al mismo tiempo es la causa indicadores negativos de calidad de código, es fundamental para el ejercicio de investigación, debido que estas causas comunes nos servirán de apoyo para fundamentar la propuesta modelo. Se ha encontrado en la literatura que una de las causas que más se presenta es la inmadurez en el software, que normalmente se debe a decisiones estratégicas para desarrollar software en un corto plazo, atajando otras tareas importantes en el desarrollo (por ejemplo, buenos diseños y pruebas) y obtener de esta manera una ventaja comercial (Guo et al., 2016). Otra causa común nombrada es el aumento de la deuda técnica por deficiencias en los desarrolladores de software, que normalmente se debe a temas de formación y el uso incorrecto de técnicas de codificación. El conjunto de malas prácticas de desarrollo y el crecimiento en negativo sobre las estadísticas de calidad, nos introduce en la noción de deuda técnica (principalmente de código fuente) y con el tiempo va generando unos intereses que se traducen en errores y mayores costos en el mantenimiento de las aplicaciones, conduciendo a que la inversión en software sea mayor al valor de retorno. A lo largo de este trabajo encontraremos decisiones o actividad causantes de una alta deuda técnica en el producto software y cómo se pueden controlar.

Según (Guo et al., 2016), una metáfora de la deuda técnica conocida, es que los beneficios a corto plazo frente al costo a largo plazo, es decir, ejecutando tareas de desarrollo más cortas y sacrificando la calidad, y que a futuro pueden generar altos costos debido a mayores dificultad en tareas de mantenimiento para que el software sea sostenible a largo plazo; esta apreciación es muy valiosa para el problema de investigación en lo referente a las actividades desarrollo, de manera que con el tiempo se van generando unos intereses que se traducen en errores y mayores costos de mantenimiento de software, conduciendo a que la inversión en software sea mayor al valor de retorno.

2.3 Objetivos de la Tesis de Investigación

La meta principal de esta Tesis está relacionada con la implementación de un modelo que ayude a tratar la deuda técnica de código fuente en las aplicaciones. Los objetivos específicos están formulados como una serie de pasos a seguir para poder delimitar la

formulación, análisis y solución del problema, logrando posteriormente el objetivo general planteado.

2.3.1 Objetivo General

Proponer un modelo para el tratamiento de la deuda técnica en el desarrollo de software ágil que ayude a disminuir la deuda en el código fuente de los componentes de la aplicación para que sean sostenibles a largo plazo.

2.3.2 Objetivos Específicos

- ✓ **OE1.** Realizar un análisis del entorno y de las variables más importantes acerca de la deuda técnica de código fuente, a partir de una evaluación de las propuestas en la literatura.
- ✓ **OE2.** Analizar la influencia de las metodologías de desarrollo ágil en el uso de buenas prácticas de codificación y gestión de la deuda técnica.
- ✓ **OE3.** Elegir las técnicas que sean referentes en la producción de código de calidad que gobiernen la gestión y disminución de la deuda técnica para las distintas fases que estructuran el modelo.
- ✓ **OE4.** Adaptar principios de programación en enfoques desarrollo ágil de software para definir arquetipos para el tratamiento de la deuda técnica de código fuente.
- ✓ **OE5.** Estructurar los componentes y elementos del modelo para el tratamiento de la deuda técnica orientado a la evolución de los componentes de software, a partir de las técnicas seleccionadas y de los aspectos de la influencia del desarrollo ágil.
- ✓ **OE6.** Identificar en el proceso de construcción de software qué actividades contribuyen a mejorar la eficiencia y disminución de la deuda técnica en los equipos de desarrollo, basado en una técnica de priorización, costos y beneficios.
- ✓ **OE7.** Validar si el modelo para el tratamiento de la deuda técnica es apto para que las aplicaciones sean sostenibles a largo plazo, con su aplicación en un caso de estudio.

2.3.3 Comprensión Objetivos Específicos

Objetivos Específicos	Capítulos Relacionados	Entregables
OE1 y OE2	Capítulo III. Marco Teórico	Introducción de las principales definiciones relacionadas al concepto de deuda técnica.
OE1 y OE2	Capítulo IV. Trabajos Relacionados	Compendio de investigaciones previas para el estudio del estado del arte en gestión de la deuda técnica. Se hace un análisis de algunos trabajos que han servido de apoyo para la realización de la tesis de investigación.

OE3 y OE5	Capítulo V. Evaluación de técnicas para la producción de código de calidad y tratamiento de la deuda técnica	Recopilación de las prácticas de código limpio para el desarrollo de código de alta calidad técnica para llevar a cabo un proceso eficiente de disminución de deuda técnica.
OE4 y OE6	Capítulo VI. Propuesta de Solución	Definición de las fases de gestión de DT del modelo propuesto. Definición de las actividades, valores de entrada y salida, roles participantes y diagramas de las fases del modelo.
OE7	Capítulo VII. Validación	Presentación de la validación del experimento de investigación, la aplicación de la propuesta modelo y los resultados arrojados.

2.4 Metodología Empleada

La Metodología de Investigación utilizada es la Ciencia basada en el Diseño en Sistemas de Información (Hevner et al., 2004), la cual es un paradigma de resolución de problemas en investigaciones de informática y ciencias de la computación, buscando entender el problema de investigación, realizar un diseño para la propuesta del modelo y ejecutar la validación. El método de investigación experimental aplicado a la Ingeniería de Software, llamado formalmente "Ingeniería de Software Experimental", funcionara como apoyo a la promoción de la aplicación del método científico en el reconocimiento de los experimentos y tratamientos aplicados a los sujetos de investigación.

Las investigaciones, y en particular los diseños experimentales, intentan establecer básicamente relaciones causa-efecto. Más específicamente, cuando se desea estudiar cómo una variable independiente "causa" modifica una variable dependiente "efecto" (Ramón, 2010). Para la Tesis de Maestría, la aplicación de la metodología de investigación tiene un ciclo de vida iterativo e incremental, buscando contribuir al conocimiento de la ingeniería de software en el marco de las siguientes tres fases:

Fase 1 "Relevancia". Análisis de entorno: En esta fase se tendrá un análisis de las hipótesis y del problema de investigación que justifique la necesidad de investigar acerca de la deuda técnica y su impacto en los proyectos de software.

Fase 2 "Rigor". Análisis de la base de conocimiento: En esta fase se realizará una revisión del estado del arte del tema de investigación, que permita determinar qué técnicas, modelos y casos de estudio han tenido una relevancia importante en el problema de la deuda técnica y su impacto en los proyectos de software.

Fase 3. Diseño y validación del modelo: En esta fase se definirá la propuesta del modelo y un diseño que plasme los componentes y elementos, objetivos y pasos para enfrentar los problemas de la deuda técnica. Del mismo modo se realizará una comprobación de la propuesta de modelo en un ambiente controlado de desarrollo y pruebas de un proyecto de software, que debe cumplir con los requisitos de deuda técnica que exigirá el modelo para su experimentación.

2.5 Organización del Documento

El documento está organizado para representar el desarrollo del trabajo de investigación, el cual se estructura en los siguientes cinco capítulos:

El *Capítulo 1* comprende el Resumen, Abstract y Agradecimientos de la tesis de investigación.

El *Capítulo 2* es una introducción sobre la temática tratada en la tesis de maestría y se contextualiza la problemática encontrada en la literatura. Posteriormente se describen las motivaciones para investigar sobre la deuda técnica y su impacto en el desarrollo de software, el Objetivo General y los Objetivos Específicos consiguiendo especificar el alcance de la investigación. Este capítulo también abarca cuál es la metodología empleada para el desarrollo del trabajo, así como un adelanto sobre aproximaciones a la solución.

El *Capítulo 3* trata sobre el Marco Teórico, se realiza una presentación de los principales conceptos en el área de la Deuda Técnica.

El *Capítulo 4* hace una presentación de los Trabajos Relacionados con esta investigación, donde se encuentra categorizados los artículos referentes a los distintos tipos de DT, además se analizan aquellos trabajos de autores con las propuestas más significativas desde el año 2012 hasta 2018.

El *Capítulo 5* trata sobre la evaluación y compendio de técnicas de para la producción de código de calidad, según el paradigma de programación en el que se encuentra enmarcado el tratamiento de la DT de este trabajo.

El *Capítulo 6* está dedicado a presentar la propuesta de modelo para la solución de la tesis de maestría, expresando los pasos llevados a cabo para la formulación y elaboración del contenido del modelo propuesto, haciendo énfasis principalmente en la gestión de los tipos de Deuda Técnica como son: Código fuente, Diseño, Arquitectura y Pruebas.

En el *Capítulo 7* se presenta el experimento llevado a cabo para la validación del modelo. Se explica el tipo de investigación y el diseño experimental, se recogen los datos arrojados por los grupos experimentales, se realiza en análisis y diseño de la información, logrando mostrar la validez de los experimentos y que conclusiones arrojaron.

En el *Capítulo 8* se describen las conclusiones obtenidas a partir del desarrollo completo de la tesis de maestría, mencionando las aportaciones derivadas del desarrollo y aplicación del Modelo para el tratamiento de la deuda técnica orientado a la evolución de los componentes para que las aplicaciones sean sostenibles a largo plazo, la respuesta dada a los interrogantes de investigación planteados, la discusión de resultados y terminar con las futuras líneas de investigación a desarrollar a partir de esta tesis.

3.

Capitulo III

Marco Teórico

En este capítulo se introducen los conceptos básicos para entender el fenómeno de la deuda técnica en la construcción de software y cómo genera una serie de problemas relacionados con la calidad de código fuente, que termina afectando la evolución de las aplicaciones con el paso del tiempo. El aumento continuo de la DT durante el desarrollo del sistema suele perjudicar los cambios en el diseño y arquitectura de las mismas, y estas no van en concordancia con su idea original. De esta manera, se cae en un deterioro de la calidad a partir de la pérdida de mantenibilidad, flexibilidad, legibilidad, fácil de probar, adaptación al cambio, entre otros atributos. Entender la DT, su impacto, costo y esfuerzo para emplear una gestión correcta de su tratamiento, puede ayudar a mejorar la calidad sin alterar el comportamiento observable del sistema. Todos estos conceptos son utilizados durante el desarrollo de este trabajo para definir y validar un modelo para el tratamiento de la DT guiado por escenarios de calidad código, refactorización y código desagradable (*code smell*).

3.1 Deuda Técnica

El concepto de Deuda Técnica, fue introducido por primera vez por Ward Cunningham en la conferencia de OOPSLA 1992, como una forma de describir las consecuencias de la mala arquitectura del software y la mala codificación. La DT es un medio para comunicar los desafíos que surgen de las consecuencias del software mal desarrollado para los interesados no técnicos, usando la siguiente analogía: *"Enviar el código de primera vez es como endeudarse. Una pequeña deuda acelera el desarrollo siempre y cuando se devuelva rápidamente con una reescritura. Los objetos hacen que el costo de esta transacción sea tolerable. El peligro ocurre cuando la deuda no se paga. Cada minuto gastado en un código no muy correcto cuenta como intereses en esa deuda"* (Cunningham, 1992). Recientemente, se redefinió la deuda técnica como: *"Construcciones de diseño o implementación que son convenientes a corto plazo, pero configuran un contexto técnico que puede hacer que un cambio futuro sea más costoso o imposible. La deuda técnica es un pasivo contingente cuyo impacto está limitado a cualidades internas del sistema, principalmente mantenibilidad y evolución"* (Kruchten, 2016).

3.2 Calidad de Código

La calidad de nuestro código puede ser vista desde distintos frentes. Uno de ellos tiene que ver con que el código esté lo más limpio de fallas posibles, y otro tiene que ver con su estructura y mantenibilidad. En particular, la mantenibilidad ha sido una gran problemática durante todos los decenios de la crisis del desarrollo de software desde mitad del siglo XX. En la Tabla 1 encontramos algunas definiciones de código limpio realizadas por reconocidos expertos en la industria del software:

Tabla 1. Definiciones de código limpio.

Autor	Definición
Bjarne Stroustrup	Elegante, eficiente, lógica directa, mínimas dependencias y fácil de mantener; el manejo de error completo de acuerdo a una estrategia, el rendimiento cerca a lo óptimo.
Grady Booch	El código limpio es simple y directo, se lee como prosa bien escrita que no obscurece las intenciones del diseñador, sino que está lleno de abstracciones claras.
Dave Thomas	El código limpio puede ser leído y mejorado por un desarrollador distinto de su autor original. Tiene pruebas unitarias y de aceptación. Tiene nombres con significado. Proporciona una forma de hacer las cosas en lugar de muchas alternativas.
Michael Feathers	El código limpio parece estar hecho por alguien a quien le importa.
Ron Jeffries	Pasa todas las pruebas. No tiene duplicidades. Expresa las ideas de diseño del sistema. Minimiza el número de entidades como clases, métodos y similares.
Ward Cunningham	Sabes que estás trabajando con código limpio cuando cada rutina que lees, resulta ser como lo que esperabas encontrarte. Cuando parece que el lenguaje fue hecho para el problema que resuelve el código.

Es por eso que mantener el código limpio no solo es un tema de costos, sino que también es un tema de supervivencia profesional, y en la industria del desarrollo de aplicaciones han sido definidos una serie de definiciones, prácticas y consejos para lograr mayor eficiencia en la construcción de software:

Costo de pertenencia del desorden: Escribir código basado en el diseño tiene beneficios, y es que nuestro código se adhiere a las arquitecturas. A pesar de esto es muy posible que a medida que pase el tiempo la base de código de un producto crezca, el código empiece a desordenarse y a entorpecer la labor de los mismos desarrolladores que lo hicieron, o de nuevos desarrolladores involucrados en el proyecto. El código comienza a volverse costoso de mantener, pues es cada vez más difícil de desarrollar sobre él, lo que genera problemas en la productividad del equipo. Lo que al comienzo era un proyecto con desarrolladores rápidos, con los años se convierte en un proyecto con desarrolladores lentos.

Esto genera también incertidumbre en los gerentes de proyectos, que intentarán incrementar la productividad adicionando nuevas personas en los equipos, que desafortunadamente no conocen cómo hacer código que se adhiera adecuadamente al diseño. Finalmente, los esfuerzos y la presión sobre el equipo, harán que el equipo intente correr lo que más pueda; pero el efecto será como estar en un pantano: en vez de que la productividad salga a flote, lo que va a pasar es que la productividad se va a hundir más.

El Arte del Código Limpio: Así como reconocer una pintura bien hecha de otra mal hecha no significa que se es capaz de pintar, reconocer código bien hecho del mal hecho no significa que se sepa programar. Se requiere de un uso disciplinado de muchas técnicas y principios para poder crear código limpio.

La regla del *Boy Scout*: No es suficiente con escribir bien el código, también hay que mantenerlo limpio a medida que pasa el tiempo, pues el código se va degradando. El desarrollador debe tener un rol muy activo previniendo dicha degradación. Los *Boy Scout* de América tienen una regla simple que puede ser aplicada a la construcción del código: "Deje el campamento más limpio de lo que lo encontró". Mantener el código limpio es tan fácil como tener conciencia de ello a medida que se trabaja, y la mejor alternativa es mantener el código altamente legible: mejorar el nombre de las variables, fragmentar funciones grandes, eliminar duplicaciones, los condicionales anidados, etc. La legibilidad del código incrementa la mantenibilidad, ya que no se puede escribir código si no se puede leer código circundante. El profesionalismo está en que el código mejore cada vez más, y no lo contrario.

Refactorización: La refactorización surge como un intento de mejorar la producción de software reusable y que través de pequeños cambios, el código va evolucionando de manera iterativa. El software posee características que se manifiestan en forma externa, tales como robustez, extensibilidad, comportamiento, reusabilidad, etc., y otras que son propias de la estructura interna del mismo, definidas como características internas, como son los conceptos de comprensibilidad, legibilidad, redundancia, etc., (Mendez, 2010), de modo que la refactorización enfrenta principalmente las características externas, pero realizando un trabajo integral puede ayudar a corregir aspectos internos como la velocidad o desempeño.

TDD. *Test Driven Development* o desarrollo dirigido por pruebas, es una práctica que pone el acento en que todo el software construido debe tener pruebas automatizadas que pongan a prueba su correcto comportamiento. Esto previene la introducción de defectos o regresiones al momento de realizar modificaciones. Además, la práctica de TDD tiende a producir porcentajes de cobertura mayores, una característica que impacta positivamente en el mantenimiento (Sterling, 2010).

Sesiones de revisión. Son reuniones periódicas entre los miembros del equipo, para obtener retroalimentación y discutir opciones de diseño. Esto habilita al equipo a tomar decisiones que tiendan a clarificar la estructura del sistema (Sterling, 2010).

Programación Par. Dos programadores se sientan frente a una estación de trabajo. Uno de ellos escribe el código fuente y el otro colabora entregando opinión y retroalimentación de inmediato sobre lo que se está haciendo, mejorando así las decisiones de diseño (Sterling, 2010). Esta práctica tiende a elevar la calidad del código y es parte esencial de la metodología XP (Extreme Programming) (Beck & Andres, 2004).

Principios de Diseño: Son prácticas de construcción de software que ayudan a que las aplicaciones sean más robustas, mantenibles y modificables. En la programación orientada a objetos existen cuatro acrónimos conocidos como DRY, KISS, YAGNI y NIH; estos síndromes describen algunas prácticas de programación como:

- **DRY – *Don't Repeat Yourself* (No te repitas)**, se refiere a evitar el copiado y pegado de código fuente que realiza lógica semejante. Esto genera duplicaciones de código e impacta negativamente en la mantenibilidad del sistema. Busca que cada pieza de conocimiento debe tener una única, no ambigua, representación autorizada dentro de un sistema. En otras palabras, se debe tratar de mantener el comportamiento de una funcionalidad del sistema en usa sola pieza de código, evitando tener código similar en diferentes partes del sistema.
- **KISS – *Keep It Simple, Stupid!* (¡mantenlo simple, estúpido!)**, se refiere a quedarse con opciones y utilización de técnicas que produzcan diseños simples de entender. Los sistemas más eficaces son los que mantienen la simplicidad, evitando la complejidad innecesaria. El objetivo es que el diseño del software sea lo más simple posible. Tanto la hora de escribir código como de depurarlo, aún más cuando se enfrentan a código complejo.
- **YAGNI – *You Aren't Gonna Need It* (no vas a necesitarlo)**, se refiere a no producir diseños que traten de capturar posibles requerimientos futuros, pero sin la certeza de que serán necesarios. Indica que un programador no debe agregar funcionalidades extras hasta que no sea necesario. Las bases de la construcción deben ser hechos y no suposiciones. Así se evita sobre-diseñar sistemas, tendencia que retrasa y entorpece el alcance inicial de las tareas que estén desarrollando.
- **NIH – *Not Invented Here* (no inventado aquí)**, se refiere a la tendencia de rechazar soluciones de terceros que cumplen cabalmente con una funcionalidad propuesta, reconocidas en el mercado y razonables en cuanto a funcionalidad o precio, por parte de desarrolladores o empresas de software, y a optar por soluciones desarrolladas internamente en la empresa. Algo así como el síndrome del "vamos a reinventar la rueda". Este síndrome tiene diferentes intensidades, que van desde un sentirse moderadamente reacio a aceptar nuevas ideas hasta el extremo de tener

un rechazo irracional a cualquier software externo, simplemente por el hecho de que “No fue inventado aquí”.

3.3 Procesos de Refactorización

En esta sección se tendrá una visión general de la investigación existente en materia de refactorización de software en la actualidad. Refactorizar es el proceso en el cual se aplican cambios en un sistema de software de tal forma que no altere el comportamiento externo del código, mejorando su estructura interna (Fowler & Beck, 1999). Es una inversión en una base de código. Su objetivo es hacer que el código sea más fácil de entender, más fácil de mantener y más fácil de extender.

Se han identificado algunos formalismos, procesos, métodos y herramientas que se ocupan de la refactorización de una manera más consistente, genérica, escalable y flexible. Existen diferentes criterios para las actividades del proceso de refactorización, como técnicas específicas usadas para soportar esas actividades, tipos de artefactos que se pueden refactorizar, asuntos importantes a tener en cuenta al desarrollar herramientas de soporte a la refactorización y el efecto de la refactorización en el proceso del software (Mens & Tourwé, 2004). Generalmente se aplican dos tácticas fundamentales para ejecutar refactorizaciones; la primera es llamada *floss refactoring*, que es cambiar el código en pequeños pasos y en la cual el desarrollador siempre sabe lo que necesita ser refactorizado, dado que este solo esta refactorizando lo que está entorpeciendo su progreso; la segunda es llamada *root canal refactoring*, que trata de dedicar un periodo de tiempo de desarrollo exclusivamente a la refactorización del sistema. A partir de un análisis estadístico de un grupo de proyectos de software, se pudo determinar que aproximadamente un 88% de las refactorizaciones son de tipo *root canal*, mientras que el restante 12% es de *tipo floss* (Zhang et al., 2012).

En la práctica, las herramientas no se utilizan tanto como podría ser ya que a veces no se alinean con el *floss refactoring*, la táctica de refactorización más utilizada por los desarrolladores. Algunas herramientas se han diseñado de una manera que las hace más apropiadas para la refactorización “*root canal*” que para refactorización “*floss*”, es por esta razón que las herramientas de refactorización no se utilizan tanto como se podría esperar. Según Kim, Zimmermann & Nagappan. 2014, la definición de refactorización en la práctica no se limita a una definición rigurosa de las transformaciones de código y que los desarrolladores perciben que la refactorización implica costos y riesgos considerables, los cuales fueron encuestados para un caso de estudio (ver Tabla 2 y 3), en el que evaluaron las consideraciones de los riesgos y beneficios de la refactorización en un rango de 0 a 100.

Tabla 2. Riesgos asociados a la refactorización

Riesgo	Valor
Desorden	1.64%
Errores de regresión y compilación	75.41%
Conflictos por Merge	11.15%
Tiempo tomado de otras tareas	18.69%
Costos de testing	23.61%
Dificultad en la revisión de código	6.89%

Tabla 3. Beneficios asociados a la refactorización

Beneficio	Valor
Mejora de mantenibilidad	29.9%
Mejora de legibilidad	43.3%
Mejora en el desempeño	12.03%
Mejora de modularidad	18.56%
Menor cantidad de errores	26.8%
Reducción en el tamaño del código	4.12%
Reducción en la duplicación de código	6.19%
Mejora en la testeabilidad	12.03%
Extensibilidad. Facilidad para agregar nuevas funciones	27.15%

3.4. Refactorización guiada por código desagradable (*code smell*)

Actualmente existe un amplio abanico de herramientas automáticas y métricas de código utilizados para realizar diagnósticos de calidad de un sistema y detectar oportunidades de refactorización. Es recomendable en organizaciones dedicadas al desarrollo de software, que la toma de decisiones con respecto a la mejora del software con el uso de la refactorización, sea una combinación de criterios, métricas de código y evaluaciones subjetivas realizadas por desarrolladores.

Olores y Heurísticas: Durante muchos años hemos conocido herramientas que validan la calidad del código. Dichas herramientas fundamentan su trabajo en un conjunto de reglas que se han venido estableciendo dentro de la comunidad de desarrollo. Algunas reglas pueden ser aplicadas a cualquier lenguaje, otras reglas están asociadas a la plataforma en la que corren algunos lenguajes, y otras son sobre algunos aspectos de diseño de la solución. Las reglas pueden ser agrupadas y puestas a disposición de un servidor de IC que constantemente esté validándolas a medida que se suban actualizaciones de software.

Integración continua. Es una práctica para la detección de fallos en el menor tiempo posible, que consiste en revisar cada modificación del código fuente que se construye en un proyecto, definiendo una serie de pasos que verifiquen que el código cumpla con ciertos criterios de calidad anteriormente definidos. Los pasos establecidos son principalmente de compilación y pruebas unitarias de código, pero también se pueden ejecutar otro tipo de pasos como pruebas de seguridad, integración, despliegues y entre otras actividades en forma automática. Esto provee una retroalimentación temprana, que detecta en cuestión de minutos si se han introducido defectos (Sterling, 2010).

3.5 Desarrollo ágil de software

Se refiere a métodos de Ingeniería del Software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan con el tiempo según la necesidad del proyecto (Lasa et al., 2017). Según el Manifiesto Ágil, el desarrollo ágil se caracteriza por (Beck et al., 2001):

- Se reconoce a las personas como primeros implicados en el éxito de un proyecto.

- Aceptar requisitos cambiantes, incluso en etapas avanzadas.
- Entregas parciales del producto. Conversación cara a cara.
- Los responsables de negocio y los desarrolladores deben trabajar juntos diariamente a lo largo del proyecto.
- Construir proyectos con profesionales motivados.
- Se satisface al cliente a través de la entrega temprana y continua de software con valor.
- Software que funciona es la principal medida de progreso.
- Los procesos ágiles promueven el desarrollo sostenible.
- La atención continua a la excelencia técnica y los buenos diseños mejoran la agilidad.
- La simplicidad es esencial.
- Las mejores arquitecturas, requisitos y diseños surgen de equipos que se auto-organizan.
- A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo.
- Es mejor crear un ambiente de colaboración y confianza entre el cliente y los desarrolladores que estar refinando el contrato.

3.6 Ingeniería de Software Basada en Componentes

La Ingeniería de Software Basada en Componentes (ISBC) promueve el desarrollo de sistemas de software a través de la construcción de componentes de software existentes, el desarrollo de componentes como entidades reutilizables y la evolución del sistema mediante la personalización y reemplazo de componentes (Szyperski, 2000). Las motivaciones detrás de ISBC han sido de naturaleza comercial y técnica, es decir, mayor eficiencia y efectividad, menores costos y, por otro lado, menor tiempo de comercialización, por un lado, y mejor calidad en cuanto a menos errores, mejor desempeño, facilidad de mantenimiento, portabilidad, etc. Según Szyperski. (1998) “Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio”. Al desarrollar un componente, este debe ser capaz de reutilizarse y ser funcional en contextos distintos, incluso en aquellos para los no ha sido debidamente diseñado.

3.6.1 Marcos de Trabajo

Los Marcos de Trabajo o *Frameworks* en lo referente a desarrollo de componentes de software, son ejemplos de un diseño reutilizable de todo o parte de un sistema, representado por un conjunto de clases de clases abstractas y la forma en la cual interactúan. En un *Framework* se encapsulan distintos patrones de arquitectura de software, debido a que encapsulan en un alto nivel la estructura de un sistema, apropiada separación de sus partes, las interacciones entre ellas y la definición de restricciones. Las principales ventajas que ofrecen son la reducción del coste de los procesos de desarrollo de aplicaciones de software para dominios específicos, y la mejora de la calidad del producto

final (Fayad & Schmidt, 1997). Entre las dificultades que encontramos en los marcos de trabajo son los de una correcta o actualizada documentación, hasta qué grado es capaz dar solución a los problemas para los que fue diseñado, su escalabilidad, actualización, especialización y complejidad de adaptación o uso que puede requerir representar en otro sistema.

3.6.2 Desarrollo de aplicaciones basadas en componentes

La importancia de los marcos de trabajo a la hora de desarrollar aplicaciones basadas en componentes es muy notoria, debido a que aportan distintas propiedades y ventajas en nuestro sistema, tales como fiabilidad, eficiencia, escalabilidad, calidad de servicio, entre otros (propiedades no funcionales), de esta forma se ha encontrado un equilibrio muy importante entre los componentes y los marcos de trabajo, liberando las propiedades funcionales de las aplicaciones sobre los primeros, y las no funcionales sobre los segundos. El desarrollo de componentes no puede hacerse de forma arbitraria, sino de acuerdo a los modelos existentes, incorporando aquellos mecanismos que permitan a dichos componentes interoperar con otros.

Los Entornos de Desarrollo Integrados (IDE, "*Integrated Development Environment*, en inglés"), es una aplicación visual que sirve para la construcción de aplicaciones a partir de componentes. Por lo general todas ellas cuentan con los siguientes elementos:

- Distintos tipos de vista o paletas, para mostrar como los componentes disponibles para el desarrollo. Contenedores en donde se colocan los componentes y se interconectan entre sí.
- Editores específicos para configurar y especializar los componentes. Visores (browsers) para localizar componentes de acuerdo a ciertos criterios de búsqueda. Directorios de componentes, acceso a editores, interpretes, compiladores y depuradores para desarrollar nuevos componentes.
- Herramientas de control y gestión de proyectos, esenciales para grandes proyectos software. En el mercado encontramos una gran variedad de IDE's, que soportan más de un lenguaje programación y van evolucionando a añadiendo nuevas características, ejemplo de ellos son:
    - ❖ Eclipse, NetBeans, IntelliJ IDEA y JDeveloper de Oracle para lenguajes como Java, Scala, PHP, Groovy, JavaScript, entre otros.
    - ❖ KDevelop, Anjunta, MS Visual Studio, Visual C++ para lenguajes como Phyton, C++, Visual Basic, C Sharp, entre otros.

En conclusión, la ISBC es de gran interés para la propuesta modelo para el tratamiento de la deuda técnica desarrollada en esta investigación, debido a que este enfoque de desarrollo sintetiza unos conceptos claros para desarrollar software de manera evolutiva y donde la gestión de la DT puede ser llevada a cabo en un proceso sistemático, escalable, eficiente y menores costos para la vida útil de los aplicativos.

4.

Capítulo IV

Trabajos Relacionados

Este capítulo describe los estudios más significativos relacionados con la Gestión de la Deuda Técnica, el estado del arte y las contribuciones realizadas en las áreas de DT. Con el objetivo de presentar los distintos enfoques analizados, los mismos se explicarán en la función de los aportes que realizarán en las temáticas que más interesan al desarrollo de este trabajo final. Algunos de estos trabajos brindan indicios de problemas que resultan útiles para guiar y enfocar el proceso de remediación de DT sobre los sistemas, basándose exclusivamente en la identificación de oportunidades de refactorización y mejora de indicadores de código desagradable (*code smell*). Este capítulo se organiza en cinco secciones que permiten dar a conocer los temas más importantes para esta investigación en materia de DT como: Gestión de la deuda técnica, Tipos y subtipos de deuda técnica, Remediación de la deuda técnica, Análisis de propuestas símiles a modelos y Marcos de Trabajo para la remediación de la DT y Conclusión de los trabajos relacionados.

4.1 Gestión de la Deuda Técnica (GDT)

El segundo objetivo específico del presente trabajo es mostrar las actividades que componen la Gestión de la Deuda Técnica o "*Technical Debt Management"* en inglés, y los principales aportes encontrados en la literatura, como su importancia para los proyectos de software. En todo proyecto de software la DT es inevitable, a pesar de que se pueda reducir con el uso de buenas prácticas. El primer paso a seguir para una estrategia de Gestión de la Deuda Técnica (GDT) es la Identificación de lo que es DT y de lo que no es DT, para luego proceder a realizar la gestión de la DT identificada. Este primer paso desde la perspectiva de los costos se considera difícil y que consume mucho tiempo, siendo una gran sobrecarga inicial para la administración de DT (Li, Avgeriou, & Liang, 2015).

La DT se puede encontrar en muchos artefactos diferentes producidos durante el proceso de desarrollo de software. Como consecuencia, se deben emplear una variedad de estrategias si el objetivo es encontrar todos los tipos de DT que puedan tener un impacto negativo. La GDT se compone de un conjunto de actividades que evitan que se incurra en la adquisición de nueva deuda o que se trabaje en la deuda existente para mantenerlos por debajo de un nivel razonable. Las principales actividades para la GDT son las siguientes (Li et al., 2015):

- **Identificación (AI):** Detecta la DT causada por decisiones técnicas intencionales o no intencionales en un sistema de software a través de métodos específicos, como la revisión de código estático. Responde el primer interrogante fundamental sobre DT: ¿Dónde se ubica la DT?
- **Medición (AME):** cuantifica el beneficio y el costo de la DT conocida en un sistema de software a través de técnicas de estimación, o estima el nivel de DT total en un sistema. Responde el segundo interrogante fundamental sobre DT: ¿Cuánto cuesta la DT?
- **Remediación (AR):** También conocida como Amortización. Durante esta actividad es que la DT es solucionada o mitigada. Responde el tercer y último interrogante fundamental sobre DT: ¿Cómo pagar la DT?
- **Priorización (API):** Los rangos de priorización de DT se identifican de acuerdo con ciertas reglas predefinidas para ayudar a decidir qué ítems de DT se deben reembolsar primero y qué ítems de DT se pueden tolerar hasta versiones posteriores.
- **Prevención (APE):** Tiene como objetivo evitar que se incurra en DT.
- **Monitoreo (AMO):** Observa los cambios en el costo y los beneficios de la DT no resuelta a lo largo del tiempo.
- **Documentación (AD):** Proporciona una forma de representar y codificar la DT de manera uniforme abordando las inquietudes de actores particulares.
- **Comunicación (AC):** Hace que la DT identificada sea visible para las partes interesadas, de modo que pueda ser discutido y gestionado.

4.1.2 Análisis de propuestas y contribuciones en GDT

Realizando un análisis de los estudios más significativos por año de publicación, **Guo et al. (2011)**, propone un enfoque de GDT en un caso de estudio, investigando los motivos en los retrasos en el compromiso de las tareas y el contexto que rodea estos motivos. Estiman el esfuerzo requerido para completar las tareas de DT y miden el tiempo tomado en completarlas. Finalmente determinan los beneficios que se obtuvieron con la medición y gestión explícita de la DT.

Seaman et al. (2012), primero organiza la DT en la fase de identificación, para luego determinar la necesidad de técnicas y enfoques de decisión para apoyar la evaluación de las compensaciones entre las mejoras propuestas, el mantenimiento correctivo y el pago de la DT. Busca diferenciarse del enfoque de costo-beneficio e inspirarse en la metáfora de la deuda tomada del mundo de las finanzas, examinando diferentes enfoques de decisión de inversión para priorizar las tareas de mantenimiento del software.

Tom et al. (2013), aplica una técnica exploratoria de casos de estudio, que involucra una revisión sistemática de literatura, complementada por entrevistas con profesionales de software y académicos, buscando definir y validar rigurosamente el concepto de deuda

técnica, entender sus impactos, tanto en la comunidad profesional como en la academia. El estudio encontró una falta de estudios primarios que examinaran el fenómeno de la deuda técnica. Además, había una ausencia de una definición exhaustiva, un modelo conceptual o un marco teórico sustantivo de la deuda técnica en la literatura académica.

Griffith et al. (2014), realiza un estudio de técnicas de estimación y de su idoneidad, buscando la relación entre los modelos de calidad externos y la DT. Cada método seleccionado de estimación de deuda técnica utiliza un modelo de calidad subyacente diferente para el que no se disponía de comparación directa. Finalmente utilizan el modelo de calidad de un tercero, llamado QMOOD, para evaluar la precisión de las técnicas de estimación de la DT.

Li et al. (2015), realizan un mapeo sistemático de literatura sobre la Gestión de la Deuda Técnica (GDT), con el fin de tener una comprensión clara y completa sobre el estado de la técnica en GDT. Se han utilizado, propuesto y desarrollado diferentes métodos y herramientas para GDT, pero no está claro cómo estos métodos y herramientas se relacionan con las actividades de GDT. Su trabajo se enfoca en resolver ambigüedades en torno a la inevitable exageración de DT, diferenciar entre qué se puede clasificar como DT y qué no en el desarrollo de software, el compromiso de qué atributos de calidad del sistema se considera DT, y cuáles son los límites de la metáfora DT.

Guo et al. (2016), desarrollan un marco general que propone administrar la DT de manera explícita: identificando la DT, midiendo su costo, es decir, capital e interés, y tomando la decisión de pagar la TD a través del análisis de costo-beneficio. Desarrollaron un caso de estudio al aplicar el enfoque simple pero explícito de la gestión de la DT a un proyecto de software, recopilaron y analizaron los datos sobre diversos costos incurridos, de cómo los elementos de la DT identificados influyeron en las decisiones sobre si invertir o mantener la DT.

Kosti et al. (2017), investiga si las evaluaciones monetizadas del capital de DT pueden ser subsumidas por puntajes métricos. Determinan la fiabilidad de los enfoques de evaluación existentes, en el sentido de que se verifica su acuerdo, proporcionan un pequeño conjunto de métricas que son más fáciles de calcular que SQALE, y en la práctica, generalmente se calculan como parte del proceso de evaluación de la calidad de muchas industrias.

Yan et al. (2018), la motivación de esta investigación es identificar la DT admitida intencionalmente mediante comentarios *"Self-admitted Technical Debt (SATD)"*, en inglés. Comparado con la DT identificada por métricas de código desagradable, SATD es más confiable ya que es admitido por los desarrolladores mediante comentarios. Los enfoques existentes en SATD se basan en el análisis de la deuda a nivel de archivos, los autores de esta investigación también incursionan en el análisis de los comentarios realizados en las herramientas de control de cambios de código como propuesta diferenciadora. Ellos

desarrollaron un modelo para determinar la DT a nivel de cambio antes del lanzamiento de un producto y así poder sumar esta actividad en la evaluación continua de GDT.

En la Tabla 4 se agrupan los trabajos más relevantes encontrados en la literatura sobre GDT desde el año 2011 hasta el 2018, además de su relación según las actividades que se ejecutan en estos:

Tabla 4. Estudios relacionados con las actividades para la GDT

Id	Artículo	AI	AME	AR	API	APE	AMO	AD	Autor
GDT1	An exploration of technical debt	X							(Tom et al., 2013)
GDT2	Tracking technical debt - An exploratory case study	X							(Guo et al., 2011)
GDT3	Using technical debt data in decision making: Potential decision approaches		X		X				(Seaman et al., 2012)
GDT4	Managing Technical Debt in Enterprise Software Packages	X	X					X	(Ramasubbu & Kemerer, 2014)
GDT5	Evaluating Technical Debt in Cloud-Based Architectures Using Real Options	X	X		X				(Alzaghoul & Bahsoon, 2014)
GDT6	The Correspondence Between Software Quality Models and Technical Debt Estimation Approaches	X			X				(I. Griffith et al., 2014)
GDT7	Searching for build debt: Experiences managing technical debt at Google		X	X					(Morgenthaler et al., 2012)
GDT8	A portfolio approach to technical debt management		X		X				(Guo & Seaman, 2011)
GDT9	Systematic Elaboration of Compliance Requirements Using Compliance Debt and Portfolio Theory		X		X				(Ojameruaye & Bahsoon, 2014)
GDT10	Estimating the Principal of an Application's Technical Debt		X				X		(Curtis, Sappidi, & Szynkarski, 2012)
GDT11	Estimating the size, cost, and types of Technical Debt, in Managing Technical Debt (MTD)		X						(Curtis, Sappidi, & Szynkarski, 2012b)
GDT12	Domain-specific tailoring of code smells: an empirical study			X				X	(Guo et al., 2010)
GDT13	Understanding the impact of technical debt on the capacity and velocity of teams and organizations				X	X			(Power, 2013)
GDT14	Estimating the Principal of an Applications Technical Debt		X	X					(Curtis, Sappidi, et al., 2012b)
GDT15	Costs and obstacles encountered in technical debt management - A case study		X		X			X	(Guo et al., 2016)
GDT16	Technical Debt Principal Assessment through Structural Metrics		X	X					(Kosti et al., 2017)

GDT17	Automating Change-level Self-admitted Technical Debt Determination	X				X	X		(Yan et al., 2018)

4.1.3 Estrategias para la GDT

Numerosos autores detallan el uso de estrategias ya probadas para la ejecución de las actividades de GDT, principalmente las de medición, identificación y priorización. **Alves et al. (2016),** a través de un proceso de revisión sistemática, define cinco estrategias de gestión que son utilizadas como guías de trabajo y apoyan la realización de las tareas para las actividades de GDT. En la Tabla 5 se hace una relación entre los estudios más relevantes para la GDT y las estrategias en las que se apoyan. Enfoque de cartera y Análisis de costo-beneficio fueron las más citadas:

Tabla 5. Artículos para la GDT y su relación con las estrategias

Artículo	Estrategia
GDT1	Marcado de dependencias y problemas de código
GDT2	Análisis de costo-beneficio
GDT3	Análisis de costo-beneficio
	Proceso de Jerarquía Analítica
GDT4	Análisis de costo-beneficio
GDT5	Análisis de costo-beneficio
	Enfoque de Cartera
	Proceso de Jerarquía Analítica
GDT6	Análisis de costo-beneficio
	Enfoque de Cartera
GDT7	Análisis de costo-beneficio
	Cálculo de capital e interés
GDT8	Enfoque de Cartera
GDT9	Proceso de Jerarquía Analítica
GDT10	Cálculo de capital e interés
GDT13	Enfoque de Cartera
GDT14	Cálculo de capital e interés
GDT15	Marcado de dependencias y problemas de código
GDT16	Análisis de costo-beneficio
GDT17	Marcado de dependencias y problemas de código

- **Análisis de costo-beneficio:** Evalúa si el interés esperado es lo suficientemente alto como para justificar el pago de la deuda. La tasa de interés se compone de dos partes: la probabilidad de interés y su valor. La primera parte se refiere a la probabilidad de que la deuda, si no se paga, resultará en un costo adicional para el proyecto. La segunda parte es una cantidad estimada de trabajo adicional que se requerirá si este artículo no se paga (Seaman et al., 2012).
- **Enfoque de cartera:** El concepto central de esta estrategia es construir la lista de elementos de DT. Esta lista contiene elementos de deuda identificados por el

proyecto. La información de registro se informa en una tabla que contiene la ubicación de la deuda, el momento en que se identifica, la persona responsable, la razón por la cual se considera DT, una estimación del capital, estimaciones del monto de interés esperado, desviación estándar de interés (ISD), y estimaciones de las correlaciones de este ítem con otros ítems DT, entre otros atributos definido por cada proyecto. Durante la planificación de cada incremento de software, se realiza un análisis de lo que se debe pagar y lo que se puede posponer (Guo & Seaman, 2011).

- **Proceso de Jerarquía Analítica (PJA):** PJA (AHP en inglés) proporciona un método para estructurar un problema, comparar alternativas con respecto a criterios específicos y determinar una clasificación general para cada alternativa. Cuando se aplica a DT, las alternativas de decisión serían los diversos elementos identificados sobre la DT, y el resultado de la estrategia sería una clasificación priorizada de estos elementos, indicando cuál debería pagarse primero (Seaman et al., 2012).
- **Cálculo de Capital e Interés:** Establecer un proceso definido para estimar el capital e interés de un ítem de DT identificada y asociar la DT con diferentes atributos de calidad que sea capaz de impulsar a los gerentes a tomar mejores decisiones (Curtis, Sappidi, et al., 2012b).
    - Capital: es el esfuerzo que se requiere para abordar la diferencia entre el nivel actual y el nivel óptimo de calidad de tiempo de diseño, en un artefacto de software inmaduro o en el sistema de software completo (Areti Ampatzoglou, Ampatzoglou, Chatzigeorgiou, & Avgeriou, 2015). Según Yli-Huumo, Maglyas, & Smolander. (2016), el término se usa para describir el costo de remediar las violaciones planificadas del sistema de software con respecto a la Deuda Técnica, en otras palabras, *el costo de refactorización*. Se puede calcular como una combinación del número de infracciones en una aplicación, las horas para corregir cada infracción y el costo de la mano de obra (Curtis, Sappidi, et al., 2012b).
    - Interés: es el esfuerzo adicional que se necesita gastar en el mantenimiento del software y mientras vaya transcurriendo el tiempo, este interés va ir creciendo si no se toman medidas para realizar los pagos.
- **Marcado de dependencias y problemas de código:** Es una estrategia utilizada para gestionar problemas y dependencias en el código fuente del proyecto. El objetivo es insertar etiquetas en la fuente del proyecto de una manera que sea fácil para el equipo de desarrollo para visualizar dónde se inserta DT y así decidir cuándo pagarlo, en función del esfuerzo involucrado y la disponibilidad del tiempo del proyecto (Holvitie & Leppänen, 2013).

4.1.3 Conclusión GDT

La DT impacta directamente sobre los atributos internos del sistema, por lo que su visibilidad y conocimiento es clave para que los equipos de trabajo se cuestionen en aspectos de cómo están construyendo sus aplicaciones, los efectos positivos o negativos en el tratamiento de la DT y la comunicación de la deuda a todos los interesados para aumentar la confianza y tomar los correctivos. El análisis de las publicaciones sobre la GDT, determina que todas las etapas del proceso de desarrollo de software son permeables a la DT, y que esta debe ser lo más simple posible, de modo que no debe afectar significativamente la capacidad de producción de software y que el propio proceso de GDT no acumule más DT quitando otras actividades para la calidad de software.

4.2 Tipos de Deuda Técnica (TDT)

No todo lo identificado como deuda técnica se refiere a las mismas deficiencias en la calidad del software, sino que distintas clases de problemas pueden tipificarse diferentemente. El concepto de DT se divide principalmente en trece tipos asociados con las áreas de estudio (aunque existen otras más). El TDT de código fuente es el de principal interés para el desarrollo de esta investigación y la posterior implementación del modelo. Otros tipos de DT de gran valor para el desarrollo del modelo son las de Arquitectura y Diseño.

4.2.1 Revisión de literatura en TDT

La investigación sobre DT está altamente concentrada en algunos tipos de deuda (diseño, arquitectura, código y defectos), mientras que los otros tipos avanzan con los años en la publicación de nuevas investigaciones. Las principales actividades de identificación de DT priorizan lo concerniente a temas relacionados con el código fuente, diseño y arquitectura, sin embargo, debe asumirse que es un riesgo olvidar los otros TDT no relacionada con el código, debido a que puede tener impactos negativos significativos en el proyecto. Por lo tanto, se sugiere evitar la tentación de limitar el enfoque a la deuda relacionada con el código. En la tabla 6 se encuentra la clasificación de los TDT, en las tablas 7 y 8 se registran los estudios más relevantes del análisis de literatura.

Tabla 6. Tipos de Deuda Técnica. Basado en Alves et al., 2014.

TDT	Nombre DT	Definición
TDT1	**Deuda Técnica de Arquitectura *(Architecture Debt)***	**Problemas encontrados en la arquitectura, por ejemplo, la violación de modularidad, que afecta requisitos arquitectónicos como rendimiento, robustez, entre otros. Este tipo de deuda no puede pagarse con intervenciones simples en el código e implica actividades de desarrollo más amplias.**
TDT2	Deuda Técnica de Compilación *(Build Debt)*	Problemas que dificultan la tarea de compilación, que consume mucho tiempo y procesamiento inútilmente. Se puede deber a definiciones innecesarias para la compilación y dependencias mal definidas.
TDT3	**Deuda Técnica de Código *(Code Debt)***	**Problemas encontrados en el código fuente que pueden afectar la legibilidad del código y dificulta su mantenimiento. Se identifica**

		examinando el código fuente considerando los problemas relacionados con las prácticas de mala codificación.
TDT4	Deuda Técnica de Defectos o Bugs *(Defect Debt)*	El software puede tener defectos conocidos y desconocidos en el código fuente. Los bugs consisten en defectos conocidos e identificados por actividades de prueba o por el usuario y que han sido informados, pero debido a otras prioridades, son diferidos para un momento posterior. Acumular una cantidad significativa de deuda técnica de defectos dificulta su posterior reparación.
TDT5	**Deuda Técnica de Diseño *(Design Debt)***	**Se puede descubrir al analizar el código fuente e identificar el uso de prácticas que violaron los principios de un buen diseño orientado a objetos, por ejemplo, clases muy grandes o estrechamente vinculadas.**
TDT6	Deuda Técnica de Documentación *(Documentation Debt)*	Problemas encontrados en la documentación, porque es inadecuada o incompleta de cualquier tipo.
TDT7	Deuda Técnica de Infraestructura *(Infrastructure Debt)*	Problemas de infraestructura que retrasan u obstaculizan algunas actividades de desarrollo. Puede referirse también a aspectos que dificultan las pruebas o el despliegue del software.
TDT8	Deuda Técnica de Personas *(People Debt)*	Problemas de personas que pueden retrasar u obstaculizar algunas actividades de desarrollo. Un ejemplo de este tipo de deuda es la experiencia concentrada en muy pocas personas, como efecto de poca capacitación.
TDT9	Deuda Técnica de Proceso *(Process Debt)*	Se refiere a procesos ineficientes y su diseño ya no es apropiado.
TDT10	Deuda Técnica de Requisitos *(Requirement Debt)*	La deuda de los requisitos se refiere a los compromisos contraídos con respecto a los requisitos que el equipo de desarrollo necesita implementar, por ejemplo, requisitos que solo se implementan parcialmente, requisitos que se implementan, pero de una manera que no satisface completamente todos los requisitos no funcionales (seguridad, rendimiento, entre otros).
TDT11	Deuda Técnica de Servicio Web *(Service Debt)*	La necesidad de sustitución de un servicio web podría estar impulsada por objetivos comerciales o técnicos. La sustitución puede introducir una deuda técnica, que debe gestionarse y transformarse de la responsabilidad al valor agregado. La deuda técnica incluye características que están relacionadas con la selección, composición y operación del servicio.
TDT12	Deuda Técnica de Automatización de Pruebas *(Test Automation Debt)*	Es la deuda de la automatización de las pruebas de funcionalidad desarrollada previamente para apoyar la integración continua y los ciclos de desarrollo agiles.
TDT13	Deuda Técnica de Pruebas *(Test Debt)*	Problemas encontrados en las actividades de prueba afectan la calidad. Ejemplos de este tipo de deuda son las pruebas planificadas que no se ejecutaron o las deficiencias conocidas en el conjunto de pruebas, por ejemplo, cobertura de código baja.

En la tabla 7 se describen los trabajos que tratan los dos tipos de deuda técnica más relevantes para la propuesta de modelo de esta tesis, la DT de Código y de Arquitectura.

Tabla 7. Artículos relacionados con los tipos de DT de Arquitectura y Código. Basado en Alves et al., 2014.

Artículo	Autores	Año	Tipo DT
Technical Debt: From Metaphor to Theory and Practice	P. Kruchten, R. L. Nord & I. Ozkaya	2012	TDT1, TDT10
A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component.	A. Martini, E. Sikander & N. Madlani	2017	TDT1
Analyzing the concept of technical debt in the context of agile software development: A systematic literature review	W. Nema Behutiye, P. Rodriguez., M. Oivo, A. Tosun	2016	TDT1
Technical debt and system architecture: The impact of coupling on defect-related activity	A. MacCormack, D. Sturtevant	2016	TDT1, TDT3
How do software development teams manage technical debt? - An empirical study	J. Yli-Huumo, A. Maglyas, K. Smolander	2016	TDT1, TDT3, TDT4, TDT5
Managing Architectural Technical Debt: A unified model and systematic literature review	T. Besker, A. Martini, J. Bosh	2016	TDT1, TDT3
Towards an Ontology of Terms on Technical Debt	N. Alves, L. Ribeiro, V. Caires, T. Mendes, R. Spínola	2014	TDT1, TDT3
Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study	A. Martini, J. Bosch, M. Chaudron	2015	TDT1
Measuring and Monitoring Technical Debt	C. Seaman, Y. Guo	2011	TDT3, TDT5, TDT6, TDT13
An exploration of technical debt	E. Tom, A. Aurum, R. Vidgen	2013	TDT1, TDT3, TDT5, TDT6, TDT8, TDT13
A case study on effectively identifying technical debt	N. Zazworka, R. O. Spínola, A. Vetró, F. Shull, C. Seaman	2013	TDT3
Monitoring code quality and development activity by Software maps	R. O. Spínola, N. Zazworka, J. Bohnet, J. Dụllne	2011	TDT3

En la tabla 8 se describen los trabajos que hablan los otros once tipos de deuda técnica, más otros temas de interés para la investigación.

Tabla 8. Otros Artículos que sirven de referencia para la investigación. Basado en Alves et al., 2014.

Artículo	Autores	Año	Tipo DT
Costs and obstacles encountered in technical debt management - A case study	Y. Guo, C. Seaman, F. Da Silva	2016	-
Identification and management of technical debt: A systematic mapping study	N. Alves, T. Mendes, M. de Mendonça, R. Spínola, F. Shull, C. Seaman	2016	-
A systematic mapping study on technical debt and its management	L. Zengyang, A. Paris, P. Liang	2015	-
Managing technical debt in software-reliant systems	N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, N. Zazworka	2010	TDT2
Searching for build debt: Experiences managing technical debt at Google	J. Morgenthaler, M. Gridnev, R. Sauciuc, S. Bhansali	2012	TDT2
Defining the decision factors for managing defects: A technical debt perspective	W. Snipes, B. Robinson, Y. Guo, C. Seaman	2012	TDT4
Organizing the Technical Debt Landscape	C. Izurieta, A. Vetró, N. Zazworka, Y. Cai, C. Seaman, F. Shull	2012	TDT5

A case study on effectively identifying technical debt	N. Zazworka, R. O. Spínola, A. Vetró, F. Shull, C. Seaman	2013	TDT5
Managing Technical Debt	C. Seaman, R. O. Spínola	2013	TDT7, TDT8
Managing technical debt: An industrial case study	Z. Codabux, B. Williams	2013	TDT9
CloudMTD: Using real options to manage technical debt in cloud-based service selection	E. Alzaghoul, R. Bahsoon	2013	TDT11

4.2.2 Subtipos de Deuda Técnica (SDT)

Para cada uno de los TDT existen un gran número se problemas que son calificados como malos olores o producción de código de baja calidad. Para esta sección se nombran como subtipos de deuda técnica a los problemas más comunes referenciados en la literatura. Los SDT han sido discutidos en un importante número de trabajos, recogiendo en la Tabla 9 problemas explicados en alguno de los trabajos relacionados. Los TDT que se relacionan con los problemas de código son los Código, Diseño, Arquitectura, Pruebas y Pruebas Automatizadas.

Tabla 9. Problemas más comunes encontrados en los TDT.

ID	Problema	TDT	# de Artículos
BACOD	Código de Baja Calidad	Código	5
DUCOD	Código Duplicado	Código	7
DCCOD	Demasiada complejidad en el código	Código	6
VRCOD	Violaciones de reglas de codificación	Código	11
HCDIS	código desagradable (*code smell*)	Diseño	10
CMDIS	Complejidad en clases y métodos	Diseño	3
SUDIS	Suciedad (Grime)	Diseño	4
ESDIS	Especificación de Diseño incompleta	Diseño	1
HAARQ	Arquitecturas desagradables (smell architecture)	Arquitectura	1
ANARQ	Antipatrones Arquitectónicos	Arquitectura	2
CDARQ	Complejas Dependencias Arquitectónicas	Arquitectura	1
VPARQ	Violaciones de buenas prácticas Arquitectónicas	Arquitectura	1
PCARQ	Problemas de cumplimiento arquitectónico	Arquitectura	3
CEARQ	Problemas de calidad de la estructura a nivel del sistema	Arquitectura	9
BCPRU	Baja cobertura de pruebas	Pruebas	5
IUPRU	Pruebas unitarias insuficientes	Pruebas	3
DRPRU	Defectos residuales no encontrados en las pruebas	Pruebas	3
COPRU	Pruebas muy costosas	Pruebas	1
BCPRUA	Baja cobertura de pruebas automatizadas	Pruebas Automatizadas	5
IAPRU	Pruebas automatizadas insuficientes	Pruebas Automatizadas	3
EPPRU	Errores de estimación en el plan de esfuerzo de la prueba	Pruebas Automatizadas	1

4.2.3 Revisión de literatura en SDT

En esta sección se agrupan los trabajos relevantes a los SDT de principal relevancia para el desarrollo de la propuesta de modelo de esta tesis de investigación, y como se relacionan con los problemas más comunes para SDT definidos en la Tabla 9. Los estudios que se enfocan en las DT de Código (Tabla 10), DT de Diseño (Tabla 11) y DT de Arquitectura (Tabla 12), realizan un análisis en profundidad de sus orígenes, esto se debe principalmente a que

realizan una apuesta a ser reconocidas y consideradas por interesados del negocio e implementadas por el equipo de desarrollo.

En la Tabla 10 se abarcan los principales artículos en DT de Código y los SDT con los que está relacionado: Código de Baja Calidad (BACOD), Código Duplicado (DUCOD), Demasiada complejidad en el código (DCCOD), Violaciones de reglas de codificación (VRCOD).

Tabla 10. Artículos relacionados con los SDT de Código.

Artículo	BACOD	DUCOD	DCCOD	VRCOD	Autor
Technical debt as a meaningful metaphor for code quality	X				(Gat, 2012)
A portfolio approach to technical debt management	X				(Guo & Seaman, 2011)
Technical debt: from metaphor to theory and practice	X			X	(Kruchten, Nord & Ozkaya, 2012)
Technical debt: towards a crisper definition	X			X	(Kruchten, Nord, Ozkaya & Falessi 2013)
Technical debt: showing the way for better transfer of empirical results	X				(Shull, Falessi, Seaman & Layman, 2013)
The SQALE quality and analysis models for assessing the quality of Ada source code		X		X	(Coq & Rosen, 2011)
A Threshold Based Approach to Technical Debt		X	X	X	(Eisenberg, 2012)
Revolution in software: using technical debt techniques to govern the software development process		X	X	X	(Gat, 2010)
Technical debt assessment: a case of simultaneous improvement at three levels		X	X		(Gat, 2010)
Release duration and enterprise agility		X	X		(Greening, 2013)
Assessing technical debt by identifying design flaws in software systems		X	X		(Marinescu, 2012)
Visualizing and managing technical debt in agile development: an experience report		X	X		(dos Santos, Varella, Dantas, & Borges, 2013)
Estimating the Principal of an Application's Technical Debt		X	X	X	(Curtis, Sappidi, & Szynkarski, 2012b)
Estimating the size, cost, and types of Technical Debt, in Managing Technical Debt (MTD)				X	(Curtis, Szynkarski, Worth, & York, 2012)
Organizing the Technical Debt Landscape				X	(Izurieta et al., 2012)
Designing and implementing a measurement program for Scrum teams: what do agile developers really need and want?				X	(Ktata & Lévesque, 2010)
Comparing four approaches for technical debt identification				X	(Zazworka et al., 2014)

En la Tabla 11 se abarcan los principales artículos en DT de Diseño y los SDT con los que está relacionado: Hediondez del código (HCDIS), Complejidad en clases y métodos (CMDIS), Suciedad (SUDIS), Especificación de Diseño incompleta (ESDIS).

Tabla 11. Artículos relacionados con los subtipos de deuda técnica de Diseño.

Artículo	HCDIS	CMDIS	SUDIS	ESDIS	Autor
Technical debt as a meaningful metaphor for code quality	X				(Gat, 2012)
Managing technical debt: An industrial case study	X				(Codabux & Williams, 2013)
Investigating the impact of code smells debt on quality code evaluation	X				(Fontana, Ferme, & Spinelli, 2012)
Design pattern decay: an extended taxonomy and empirical study of grime and its impact on design pattern evolution	X		X		(Griffith & Izurieta, 2013)
Design Pattern Decay: The Case for Class Grime	X				(I. D. Griffith, 2015)
Domain-specific tailoring of code smells: an empirical study	X				(Guo et al., 2010)
Organizing the Technical Debt Landscape			X		(Izurieta et al., 2012)
Technical debt: from metaphor to theory and practice	X				(Kruchten, Nord, & Ozkaya, 2012)
Technical debt aggregation in ecosystems	X				(Guo, Seaman, Zazworka, & Shull, 2010)
Prioritizing design debt investment opportunities	X				(Zazworka, Seaman, & Shull, 2011)
Comparing four approaches for technical debt identification	X		X		(Zazworka et al., 2014)
A Threshold Based Approach to Technical Debt		X			(Eisenberg, 2012)
Designing and implementing a measurement program for Scrum teams: what do agile developers really need and want?					(Ktata & Lévesque, 2010)
Managing Technical Debt with the SQALE Method		X			(Ktata & Lévesque, 2010)
Organizing the Technical Debt Landscape			X		(Izurieta et al., 2012)
Practical considerations, challenges, and requirements of tool-support for managing technical debt				X	(Falessi, Shaw, Shull, Mullen, & Keymind, 2013)

En la Tabla 12 se abarcan los principales artículos en DT de Arquitectura y los SDT con los que está relacionado: Arquitecturas desagradabes (HAARQ), Antipatrones Arquitectónicos (ANARQ), Complejas Dependencias Arquitectónicas (CDARQ), Violaciones de buenas prácticas arquitectónicas (VPARQ), Problemas de cumplimiento arquitectónico (PCARQ), Problemas de calidad de la estructura a nivel del sistema (CEARQ).

Tabla 12. Artículos relacionados con los subtipos de Deuda Técnica de Arquitectura.

Artículo	HAARQ	ANARQ	CDARQ	VPARQ	PCAQR	CEARQ	Autor
Mapping architectural decay instances to dependency models	X						(Mo, Garcia, Cai, & Medvidovic, 2013)
Assessing technical debt by identifying design flaws in software systems		X				X	(Marinescu, 2012)
Design pattern decay: an extended taxonomy and empirical study of grime and its		X					(Griffith & Izurieta, 2013)

impact on design pattern evolution							
Visualising architectural dependencies			X		X	X	(Brondum & Zhu, 2012)
Estimating the Principal of an Application's Technical Debt				X			(Curtis, Sappidi, et al., 2012a)
Organizing the Technical Debt Landscape					X		(Izurieta et al., 2012)
Managing Technical Debt with the SQALE Method					X	X	(Ktata & Lévesque, 2010)
In search of a metric for managing architectural technical debt					X		(Nord, Ozkaya, Kruchten, & Gonzalez-Rojas, 2012)
On the limits of the technical debt metaphor some guidance on going beyond					X	X	(Schmid, 2013)
Comparing four approaches for technical debt identification					X		(Zazworka et al., 2014)
A Threshold Based Approach to Technical Debt						X	(Eisenberg, 2012)
Estimating the size, cost, and types of Technical Debt, in Managing Technical Debt (MTD)						X	(Curtis, Szynkarski, et al., 2012)
A Threshold Based Approach to Technical Debt						X	(Eisenberg, 2012)
Technical debt: from metaphor to theory and practice						X	(Kruchten, Nord, & Ozkaya, 2012)
Technical debt: towards a crisper definition						X	(Kruchten, Nord, Ozkaya & Falessi 2013)

4.3 Remediación de la DT

La remediación o disminución de la DT es un proceso que se alimenta del trabajo realizado en las actividades de gestión de la DT, principalmente, las que tienen que ver con la identificación, estimación y priorización de DT. Una vez que se ha establecido el porcentaje de tiempo y esfuerzo que se va a destinar para la remediación de DT en un ciclo de desarrollo (llámese *Sprint, Release*, Iniciativa, etc.), los interesados del negocio deben decidir que ítems de DT van a ser solucionados, teniendo como guía una combinación de características de la DT que es definida en conjunto con el equipo de desarrollo:

1. DT que requiere menor esfuerzo.
2. DT que requiere de solución inmediata para apoyar decisiones estratégicas.
3. DT relacionada con los componentes de software más utilizados o importantes.
4. DT que apoya el desarrollo de nuevas características.
5. DT que apoya actividades de soporte o mantenimiento de software.

Los ítems de DT representan la DT identificada dentro de un sistema y el costo de remediación de estas deudas. El tipo de deuda es un par que combina el tipo de decisión (estratégica, táctica, incremental o inadvertida) con el tipo de artefacto (código,

arquitectura, diseño, o pruebas). Los ítems de DT se caracterizan por tener las siguientes propiedades específicas: tipo de ítem, tipo de deuda, principal, interés o ahorro total. Pueden ser divididos en dos niveles diferenciados por la complejidad, capacidad y compromiso de trabajo:

1. DT potencial: secciones del sistema que tienen un impacto negativo en la calidad del sistema.
2. DT efectiva: DT potencial que puede afectar a secciones del sistema que se ven afectadas por elementos de evolución actuales o futuros.

Las entidades de programa están contenidas en espacios de nombres y también actúan como contención para otras entidades de programa. Cada entidad de programa tiene un identificador único (dentro del alcance de su contenedor) así como un tipo específico: Clase, Atributo de Método, Declaración, Prueba de unidad, Documentación, Diseño, Secuencia de comandos de compilación o Artefacto de soporte.

Las tareas representan el trabajo que se completará durante la implementación de su elemento de trabajo principal. Por ejemplo, un ítem de DT que representa una nueva característica, necesitaría ser diseñado, implementado y probado. Por otro lado, otro ítem de DT solo necesitaría ser refactorizado y probado. Cada tarea asociada con un elemento de trabajo tiene un conjunto de dependencias con otras tareas dentro de ese elemento de trabajo. Las tareas también tienen un valor de esfuerzo estimado asociado. Este valor de esfuerzo estimado desempeña un papel en la determinación del tiempo de ejecución de la tarea, junto con la productividad del desarrollador para el tipo de tarea (para el desarrollador asignado a esta tarea). El tiempo de ejecución de la tarea puede obtenerse dividiendo el esfuerzo estimado requerido para una tarea por la productividad del desarrollador asignado para el tipo de tarea. Por lo general, la deuda no se pagará con el propósito de pagar deuda, sino con el propósito de hacer una inversión lucrativa en el software. Como ejemplo, uno no quiere pagar deuda de diseño en un sistema heredado que se garantiza que no se modificará en el futuro.

4.4 Análisis de propuestas similares a Modelos y Marcos de Trabajo para la Remediación de la DT

En el análisis de los trabajos que comparten características con esta propuesta de investigación de maestría para el desarrollo de un modelo para el tratamiento de la deuda técnica, se encuentran afinidades en aquellos que abarcan temáticas como el desarrollo de Marcos de Trabajo, adaptación de las metodologías Agiles y DT o estas relacionados con el concepto de Integración Continua.

Hamdan & Alramouni (2015), combina dos áreas de estudio, la primera es la calidad del software y la segunda, prácticas ágiles de integración continua. Esta investigación presenta un marco que identifica las características de calidad del software del proceso de desarrollo al aplicar las prácticas de integración continua en el desarrollo de proyectos de software.

Los principios de integración continua se identifican y se aplican a un caso de estudio para analizar su impacto en los factores de calidad del proceso de desarrollo de software. Entre los beneficios identificados en el trabajo, se detectó que la compañía constructora de software al cambiar sus prácticas de desarrollo hacia una integración continua, mostró una mejora significativa en la calidad general del software.

Muchos de los riesgos involucrados en el proceso de integración del software se mitigaron como resultado de la integración continua de las partes del software desarrollado una vez que está listo.

Melin (2016), implementa un entorno de inspección continua de código utilizando herramientas de revisión de código estático, con la meta de verificar la calidad del código y encontrar defectos, así como compartir experiencias y conocimientos entre los empleados. El autor se plantea como objetivo investigar cómo se puede usar la retroalimentación del entorno de inspección continua para mejorar la arquitectura y el diseño de la base del código, además de brindar soporte durante las revisiones del código. Realiza la evaluación de la herramienta SonarQube para determinar el rendimiento, gravedad y legitimidad de los problemas, revisar las reglas de inspección y las métricas calculadas. Finalmente, el autor confía en afirmar que la introducción de un entorno de inspección de código continuo es rentable, aunque se requiere una configuración razonable para poder obtener este volumen de negocios.

Ramasubbu & Kemerer (2017), presentan el desarrollo y pruebas de un caso de estudio en tres compañías de software de distintos tamaños, anexando un marco de proceso normativo para la gestión de la deuda técnica, incorporando los procesos de gestión de calidad de software en compañías y analizando los pasos del proceso sistemático del marco de trabajo de la GDT en la producción de software comercial. El marco integra los procesos requeridos para la GDT con los procesos de gestión de calidad de software existentes prescritos por el cuerpo de conocimiento de gestión de proyectos (PMBOK), y contribuye al desarrollo adicional de extensiones específicas de software para el PMBOK. Organizan los diferentes procesos para la GDT en tres grandes pasos: 1) Hacer visible la DT. 2) Realizar análisis costo-beneficio. 3) Controlar la deuda técnica. Para promulgar cada uno de los tres pasos anteriores, el marco considera entradas, herramientas y técnicas específicas, y salidas. Esto es similar a la organización de las diversas prácticas de gestión de proyectos para cada área de conocimiento cubierta por el PMBOK.

Martini, Sikander & Madlani (2018), presentan una técnica novedosa para identificar y estimar la Deuda Técnica de Arquitectura (DTA), que pueda cuantificar la conveniencia de pagar el DTA de manera sistemática y que determine si refactorizar la arquitectura puede ser conveniente una vez que existan beneficios claros y cuantificados. Proponen un framework holístico y semiautomático, compuesto por un sistema de medición y estimación para la identificación de DTA. Destacan que las decisiones de refactorización sobre la modularización del sistema son difíciles de tomar, especialmente para las partes

interesadas no técnicas. Los autores atribuyen las siguientes contribuciones de este estudio al campo de la ingeniería de software: 1) Diseño y evaluación de un sistema de medición para identificar DTA. 2) Diseño y evaluación de una fórmula para el cálculo del esfuerzo que puede usarse para estimar el valor comercial de realizar la modularización de un componente con respecto a los beneficios de reembolsar DTA. 3) Combinar el sistema de medición y la estimación del esfuerzo en un marco semiautomático para ayudar la difícil y costosa elección de priorizar el reembolso de ATD mediante la refactorización de un componente para lograr mejor modularidad del sistema. 4) Proporcionar evidencia cuantificada (aunque limitada a este caso específico) de que la modularización de un componente (pago de DTA) puede llevar a una ganancia continua para la organización (reducción del interés de ATD) y a un sistema más sostenible en el largo plazo.

Ampatzoglou et al. (2018), diseñaron un marco de trabajo denominado **FITTED** (*A Framework for Managing Interest in Technical Debt To develop a theory on managing*), que se puede utilizar para la gestión a largo plazo de la DT. FITTED considera que el interés acumulado durante la evolución del software puede potencialmente superar la cantidad requerida para pagar el capital de DT. Desarrollan una teoría sobre la gestión del interés de la DT, tomando prestados fundamentos de las teorías de interés económico sobre el logro del equilibrio. Particularmente adopta la idea del punto de ruptura en el mercado monetario, donde la oferta monetaria es igual a la demanda monetaria. Por lo tanto, es crítico para los líderes de proyectos poder estimar un punto en el tiempo donde el interés acumulado será igual al capital de TD.

Este punto en el tiempo se denomina "punto de ruptura", en el sentido de que cualquier beneficio derivado de la decisión de asumir una DT se neutraliza después de ese punto, es decir, el costo es mayor que el beneficio. Mientras el interés acumulado sea menor que el capital, el beneficio derivado de la decisión inicial de ahorrar esfuerzo es mayor que el costo generado por el esfuerzo adicional para mantener el software. Los resultados del estudio, el interés (según lo calculado por el enfoque propuesto) parece ser más sensible que el capital (según lo calculado por SonarQube) en el sentido de que exhibe más fluctuaciones en el transcurso de la evolución del proyecto.

Para resumir los conceptos vistos en este capítulo, se ha diseñado un diagrama en la **Figura 1.** para abordar las siguientes temáticas:

a) Flujo de actividades y estrategias referentes a la GDT (Gestión de la Deuda Técnica),
b) Los TDT (Tipos de Deuda Técnica) que se vinculan a la actividad de Remediación y,
c) Como se listan los principales SDT (Subtipos de Deuda Técnica) para TDT de Código, Arquitectura, Diseño y Pruebas.

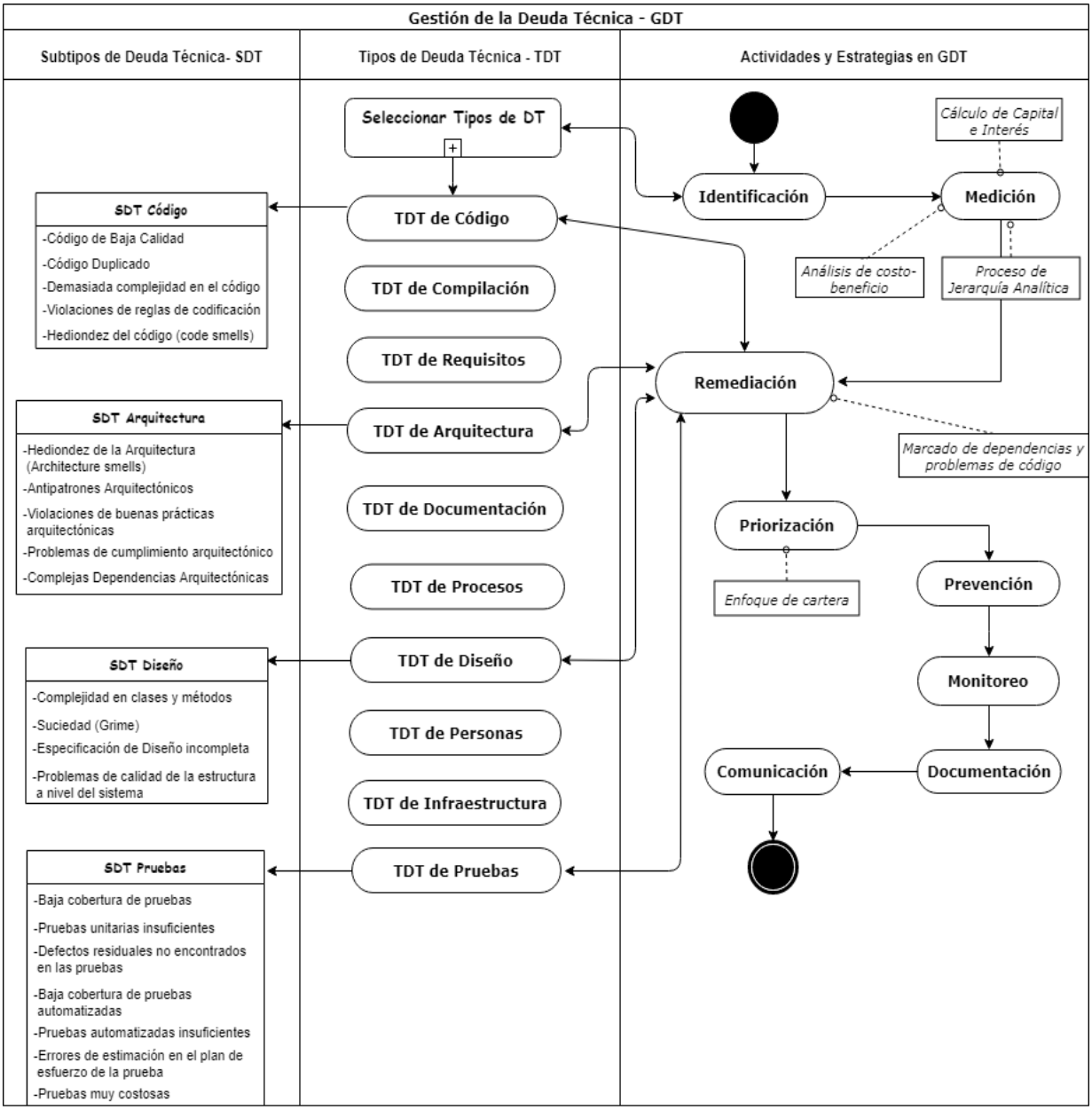


Figura 1. Diagrama conceptual de actividades y en GDT, TDT y SDT.

4.5 Conclusión

En general, la revisión de los trabajos relacionados se centró en aquellos que realizan propuestas para la identificación y gestión de la DT, explicando los beneficios de invertir esfuerzos en disminuir la DT en el código y en explicar las principales actividades para el control de la DT. Los estudios exponen los tipos de DT más comunes y tratados en la literatura y las diferencias que existen entre ellos, donde encontramos que los más estudiados son los tipos de DT de Código, Diseño, Arquitectura y Pruebas, planteando distintas alternativas para distinguir las métricas de calidad y realizar las actividades de estimación y remediación de la DT.

La necesidad de la actual tesis de maestría de plantear una propuesta diferente y complementaria a las existentes radica en lo siguiente:

1. Los trabajos analizados no se consideran ideales como marcos de trabajo o modelos lo suficientemente interesantes para llevar a cabo una gestión de la deuda técnica que pueda abarcar distintas tácticas y estrategias para la construcción de código de calidad, tales como son la refactorización, mejora del código desagradable, utilización de prácticas de desarrollo como de TDD y programación par, mejora continua de la legibilidad del código, correcto nombramiento de métodos y variables, clases más cortas, disminuir acoplamiento, entre otros, como la manera más eficiente para contrarrestar la DT en los proyectos de software.
2. Un aspecto importante encontrado en la revisión de literatura radica en explicar cómo llevar a cabo el proceso de gestión disponiendo de actividades de registro de la DT y en cómo vender a los interesados la necesidad de llevar a cabo una GDT eficiente, que pueda conectarse con los objetivos de valor y contribuir a la mejora de calidad del producto. Pero no existe un concepto claro en cómo realizar una compenetración en entornos agiles, donde la GDT no sea visto como un proceso robusto y que le reste espacio al desarrollo de funcionalidades de software. Por tal razón, un aspecto importante del desarrollo del modelo que se propone, es entender cómo integrar conceptos claves de DT en entornos de trabajo de desarrollo ágil, vinculando conceptos de Extreme Programming como Inspección Continua, Test First Design, estándares de codificación, así como el uso de potentes herramientas para el análisis de código y conceptos de DevOps, que ofrezca un mayor abanico de soluciones a los equipos de trabajo.
3. Gracias a la revisión de trabajos relacionados que proponen algún marco de trabajo para la DT y de propuestas parecidas a la nuestra (explicadas con mayor detalle en el sexto capítulo), esta investigación es diferente al poder comparar estudios similares y determinar que les hizo falta en su momento, en términos conceptuales, competitivos y la viabilidad de su implementación, para poder plantear una propuesta nueva, que no posea las mismas carencias y este más a la vanguardia del conocimiento del área tratada que otros estudios. El modelo propuesto en esta tesis de investigación, debe ser simple de implementar, optimizado para los recursos que se disponen en los equipos de trabajo, según el tamaño del proyecto y el contexto regional en el que desarrolla en este trabajo.

5.

Capítulo V

Evaluación de técnicas para la producción de código de calidad y tratamiento de la deuda técnica

El tercer objetivo específico del presente trabajo es elegir las técnicas que sean referentes en la producción de código de calidad que gobiernen la gestión y disminución de la deuda técnica para los distintos niveles definidos en el modelo. Se utiliza el método DAR de CMMI-DEV, el cual tiene como propósito analizar las posibles decisiones utilizando un proceso de evaluación formal que evalúa alternativas identificadas contra los criterios establecidos. En este capítulo también se inicia el trabajo del quinto objetivo específico, el cual busca estructurar los componentes y elementos del modelo para el tratamiento de la deuda técnica orientado a la evolución de los componentes de software, a partir de las técnicas seleccionadas y de los aspectos de la influencia del desarrollo ágil.

5.1 *Decision Analysis and Resolution* (DAR) (CMMI-DEV)

El propósito del DAR es analizar las posibles decisiones utilizando un proceso de evaluación formal que evalúa las alternativas identificadas contra los criterios establecidos. El DAR CMMI-DEV se divide en 6 Prácticas Fundamentales, de las cuales se ejecutarán 4 para el desarrollo de los objetivos específicos de investigación y se realizará un resumen de los resultados obtenidos:

5.1.1 Establecer directrices para el análisis de decisiones: Establezca y mantenga pautas para determinar qué asuntos están sujetos a un proceso de evaluación formal. No todas las decisiones son lo suficientemente significativas como para requerir un proceso formal de evaluación. La elección entre lo trivial y lo realmente importante no está clara sin una guía explícita. Si una decisión es significativa o no depende del proyecto y las circunstancias y está determinada por las pautas establecidas.

5.1.2 Seleccionar métodos de evaluación: Los métodos para evaluar soluciones alternativas contra los criterios establecidos pueden variar desde simulaciones hasta el uso de modelos probabilísticos y la teoría de la decisión. Estos métodos deben ser cuidadosamente seleccionados. El nivel de detalle de un método debe ser acorde con los costos, el cronograma, el desempeño y los impactos del riesgo.

Resultados de las prácticas 5.1.1 y 5.1.2:

La justificación para utilizar DAR se debe a la necesidad de realizar un juicio formal para la selección de las técnicas de programación que serán una parte importante para la construcción del modelo propuesto, parte de la novedad y pertinencia científica de la tesis de investigación. La definición de los criterios de evaluación fue el resultado de aplicar razonamientos sobre los aspectos más importantes de las técnicas de programación y calidad de software a evaluar. Los métodos de evaluación seleccionados para validar los criterios de evaluación son los siguientes:

- Estudios de ingeniería
- Pruebas
- Encuestas
- Sentencia proporcionada por un experto

5.1.3 Establecer criterios de evaluación: Establecer y mantener criterios para evaluar alternativas y la clasificación relativa de estos criterios. Los criterios de evaluación proporcionan la base para evaluar soluciones alternativas. Los criterios se clasifican de modo que los criterios mejor calificados ejerzan la mayor influencia en la evaluación. Esta área de proceso es referenciada por muchas otras áreas de proceso en el modelo y muchos contextos en los que se puede usar un proceso de evaluación formal. Por lo tanto, en algunas situaciones, puede encontrar que los criterios ya se han definido como parte de otro proceso.

Resultado de la práctica 5.1.3:

Fueron escogidos 10 criterios de evaluación (ver Tabla 13), con propiedades como la identificación, nombre y peso, además de un método de validez con su respectiva escala de calificación:

Validez. Se determinó un método para dar validez a los criterios y poder realizar la medición de los mismos:

- Bibliografía: En cuatro criterios de evaluación se utilizará la validez por referencias encontradas en la literatura, donde realizan alguna mención de la técnica y para qué sirve.
- Encuesta: En tres criterios de evaluación se utilizará el método de encuesta a los desarrolladores de software para realizar análisis de los datos encuestados.
- Puntuación: En tres criterios de evaluación se utilizará la puntuación por criterio de experto. Las personas escogidas para realizar esta evaluación deben contar con experiencia en desarrollo de varios años y que estén implicados en temas de calidad de software.

Escala de calificación para la Validez. Se establecerá una escala de calificación a los tres tipos de validez expuestos. Para la Bibliografía se tendrá una escala numérica de 1, 2 y 3, donde 1 es el valor más bajo y 3 el más alto. La Encuesta y la Puntuación tendrán escala de 1 a 5, donde 1 es el valor más bajo y 5 el más alto.

Peso. Los criterios de evaluación contarán con una métrica de 1, 2 y 3 para indicar su importancia en relación con la calificación final de las técnicas de programación. El valor de 1 indica el de menor importancia y 3 el de mayor**.**

Tabla 13. Criterios de evaluación.

ID	Nombre	Validez	Peso
C1	La técnica es enunciada en alguna metodología de desarrollo ágil	Bibliografía	3
C2	La técnica es enunciada en algún patrón de diseño	Bibliografía	2
C3	Relación directa entre la técnica y los subtipos de DT	Bibliografía	2
C4	La técnica está relacionada con alguna práctica de desarrollo colaborativo	Bibliografía	1
C5	Importancia que representa la técnica (Desarrolladores)	Encuesta	3
C6	Frecuencia con la que utiliza la técnica (Desarrolladores)	Encuesta	2
C7	Conocimiento de la técnica (Desarrolladores)	Encuesta	3
C8	Complejidad que presenta la implementación de la técnica	Puntuación	2
C9	Impacto de la técnica en el pago de la DT	Puntuación	3
C10	Calificación de la técnica dada por criterio de experto	Puntuación	3

5.1.4 Identificar soluciones alternativas: Identificar soluciones alternativas para abordar problemas. Puede surgir una gama más amplia de alternativas al solicitar la participación de tantas partes interesadas como sea posible. Las aportaciones de las partes interesadas con diversas habilidades y antecedentes pueden ayudar a los equipos a identificar y abordar suposiciones, limitaciones y sesgos. Las sesiones de lluvia de ideas pueden estimular alternativas innovadoras a través de una interacción y retroalimentación rápidas. La generación y consideración de múltiples alternativas al principio de un proceso de análisis y resolución de decisiones aumenta la probabilidad de que se tome una decisión aceptable y se entiendan las consecuencias de la decisión.

Resultado de la práctica 5.1.4: Para esta práctica se realizó una búsqueda bibliográfica sobre las técnicas de programación que sirvió para definir los criterios de evaluación, adquirir nuevos conocimientos y tener mayor claridad en los conceptos (ver Tablas 14 y 15).

Tabla 14. Bibliografía para la selección de las técnicas de programación

Publicación	Autor	Año
Code Complete (Second Edition)	Steve McConnell	2004
Implementation Patterns	Kent Beck	2008
Agile Software Development – Principles, Patterns, and Practices	Robert C. Martin	2003
Clean Code	Robert C. Martin	2009

Tabla 15. Artículos de referencia sobre prácticas de programación y repago de la DT

Nombre	Autor	Año
Managing Technical Debt: An Industrial Case Study	Z. Codabux, B. Williams	2013
A systematic mapping study on technical debt and its management	L. Zengyang, A. Paris, P. Liang	2015
How Do Developers Fix Issues and Pay Back Technical Debt in the Apache Ecosystem?	G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, A. Ampatzoglou	2017
Managing Architectural Technical Debt: A unified model and systematic literature review	G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, A. Ampatzoglou	2017
A Strategy Based on Multiple Decision Criteria to Support Technical Debt Management	L. Ferreira, M.G. De Mendonça, N. Souza, R. Oliveira	2017
Technical Debt and Agile Software Development Practices and Processes An Industry Practitioner Survey	J. Holvitiea, S. Licorishc, R. Spínola, S. Hyrynsalmi, S. MacDonellc, T. Mendesg, J. Buchanh, V. Leppänen	2017

5.2 Técnicas para la producción de código de calidad en el tratamiento de DT

Cada vez cada vez cobra mayor importancia la definición del uso de buenas prácticas de codificación y estilos de codificación, que puedan ser adoptados por el equipo de desarrollo como recomendaciones y normas que cumplan con la mayoría de criterios de calidad en un ambiente controlado por el equipo de desarrollo.

Después de una revisión de literatura de las principales prácticas de código limpio, se realizó una selección de las técnicas que mejor se adaptan a un contexto de tratamiento de DT y según los criterios de evaluación establecidos.

En la Tabla 16.1 encontramos un listado de 77 técnicas de refactorización de código, que se caracterizan por ser definir pequeños cambios en el código para lograr una serie de atributos de calidad que valen como apoyo a disminuir los indicadores de código desagradable y el número de horas de DT de un proyecto.

Tabla 16.1. Selección de Técnicas de Refactorización.

	Técnicas de Refactoring	Tipo de Refactoring	Publicación
1	Increase cohesion	Refactoring	Clean Code
2	Decrease coupling	Refactoring	Clean Code
3	Separate concerns	Refactoring	Clean Code
4	Modularize system concerns	Refactoring	Clean Code
5	Shrink our functions and classes	Refactoring	Clean Code
6	Choose better names: TooLong, TooShort, JustRigth, NamedConventions	Refactoring	Clean Code
7	Eliminate duplication	Refactoring	Clean Code
8	Ensure expressiveness	Refactoring	Clean Code

9	Minimize the number of classes and methods	Refactoring	Clean Code
10	Save the code you start with	Refactoring Safely	Code Complete
11	Keep refactorings small	Refactoring Safely	Code Complete
12	Do refactorings one at a time	Refactoring Safely	Code Complete
13	Make a list of steps you intend to take	Refactoring Safely	Code Complete
14	Make a parking lot	Refactoring Safely	Code Complete
15	Make frequent checkpoints	Refactoring Safely	Code Complete
16	Use your compiler warnings	Refactoring Safely	Code Complete
17	Retest	Refactoring Safely	Code Complete
18	Add test cases	Refactoring Safely	Code Complete
19	Review the changes	Refactoring Safely	Code Complete
20	Adjust your approach depending on the risk level of the refactoring	Refactoring Safely	Code Complete
21	Refactor when you add a routine	Refactoring Strategies	Code Complete
22	Refactor when you add a class	Refactoring Strategies	Code Complete
23	Refactor when you fix a defect	Refactoring Strategies	Code Complete
24	Target error-prone modules	Refactoring Strategies	Code Complete
25	Target high-complexity modules	Refactoring Strategies	Code Complete
26	In a maintenance environment, improve the parts you touch	Refactoring Strategies	Code Complete
27	Define an interface between clean code and ugly code, and then move code across the interface	Refactoring Strategies	Code Complete
28	Replace a magic number with a named constant	Data-Level Refactorings	Code Complete
29	Rename a variable with a clearer or more informative name	Data-Level Refactorings	Code Complete
30	Move an expression inline	Data-Level Refactorings	Code Complete
31	Replace an expression with a routine	Data-Level Refactorings	Code Complete
32	Introduce an intermediate variable	Data-Level Refactorings	Code Complete
33	Convert a multiuse variable to multiple single-use variables	Data-Level Refactorings	Code Complete
34	Use a local variable for local purposes rather than a parameter	Data-Level Refactorings	Code Complete
35	Convert a data primitive to a class	Data-Level Refactorings	Code Complete
36	Convert a set of type codes to a class or an enumeration	Data-Level Refactorings	Code Complete
37	Convert a set of type codes to a class with subclasses	Data-Level Refactorings	Code Complete

38	Change an array to an object	Data-Level Refactorings	Code Complete
39	Encapsulate a collection	Data-Level Refactorings	Code Complete
40	Replace a traditional record with a data class	Data-Level Refactorings	Code Complete
41	Move a complex boolean expression into a well-named boolean function	Statement-Level Refactorings	Code Complete
42	Consolidate fragments that are duplicated within different parts of a conditional	Statement-Level Refactorings	Code Complete
43	Use break or return instead of a loop control variable	Statement-Level Refactorings	Code Complete
44	Return as soon as you know the answer instead of assigning a return value within nested if-then-else statements	Statement-Level Refactorings	Code Complete
45	Replace conditionals (especially repeated case statements) with polymorphism	Statement-Level Refactorings	Code Complete
46	Create and use null objects instead of testing for null values	Statement-Level Refactorings	Code Complete
47	Routine-Level Refactorings	Routine-Level Refactorings	Code Complete
48	Move a routine's code inline	Routine-Level Refactorings	Code Complete
49	Convert a long routine to a class	Routine-Level Refactorings	Code Complete
50	Add a parameter	Routine-Level Refactorings	Code Complete
51	Substitute a simple algorithm for a complex algorithm	Routine-Level Refactorings	Code Complete
52	Remove a parameter	Routine-Level Refactorings	Code Complete
53	Separate query operations from modification operations	Routine-Level Refactorings	Code Complete
54	Combine similar routines by parameterizing them	Routine-Level Refactorings	Code Complete
55	Separate routines whose behavior depends on parameters passed in	Routine-Level Refactorings	Code Complete
56	Pass a whole object rather than specific fields	Routine-Level Refactorings	Code Complete
57	Pass specific fields rather than a whole object	Routine-Level Refactorings	Code Complete
58	Encapsulate downcasting	Routine-Level Refactorings	Code Complete
59	Change value objects to reference objects	Class Implementation Refactorings	Code Complete

60	Change reference objects to value objects	Class Implementation Refactorings	Code Complete
61	Replace virtual routines with data initialization	Class Implementation Refactorings	Code Complete
62	Change member routine or data placement	Class Implementation Refactorings	Code Complete
63	Extract specialized code into a subclass	Class Implementation Refactorings	Code Complete
64	Combine similar code into a superclass	Class Implementation Refactorings	Code Complete
65	Move a routine to another class	Class Interface Refactorings	Code Complete
66	Convert one class to two	Class Interface Refactorings	Code Complete
67	Introduce a foreign routine	Class Interface Refactorings	Code Complete
68	Introduce an extension class	Class Interface Refactorings	Code Complete
69	Encapsulate an exposed member variable	Class Interface Refactorings	Code Complete
70	Remove Set() routines for fields that cannot be changed	Class Interface Refactorings	Code Complete
71	Hide routines that are not intended to be used outside the class	Class Interface Refactorings	Code Complete
72	Encapsulate unused routines	Class Interface Refactorings	Code Complete
73	Create a definitive reference source for data you can't control	System-Level Refactorings	Code Complete
74	Change unidirectional class association to bidirectional class association	System-Level Refactorings	Code Complete
75	Change bidirectional class association to unidirectional class association	System-Level Refactorings	Code Complete
76	Provide a factory method rather than a simple constructor	System-Level Refactorings	Code Complete
77	Replace error codes with exceptions or vice versa	System-Level Refactorings	Code Complete

En la Tabla 16.2 encontramos las técnicas de Diseño y Programación, en total son 43 técnicas que se caracterizan por ser conceptos del paradigma de POO, definiciones de comportamiento y consejos para estilos de programación más robustos.

Tabla 16.2. Selección de Técnicas y Patrones de Diseño.

	Técnicas de Diseño y Programación	Categoría	Publicación
77	Find Real-World Objects	Design Heuristics	Code Complete
78	Form Consistent Abstractions	Design Heuristics	Code Complete
79	Encapsulate Implementation Details	Design Heuristics	Code Complete
80	Identify Areas Likely to Change	Design Heuristics	Code Complete
81	Look for Common Design Patterns	Design Heuristics	Code Complete
82	Aim for Strong Cohesion	Design Heuristics	Code Complete
83	Build Hierarchies	Design Heuristics	Code Complete
84	Formalize Class Contracts	Design Heuristics	Code Complete
85	Design for Test	Design Heuristics	Code Complete
86	Avoid Failure	Design Heuristics	Code Complete
87	Choose Binding Time Consciously	Design Heuristics	Code Complete
88	Make Central Points of Control	Design Heuristics	Code Complete
89	Consider Using Brute Force	Design Heuristics	Code Complete
90	Draw a Diagram	Design Heuristics	Code Complete
91	Keep Your Design Modular	Design Heuristics	Code Complete
92	Use naming conventions	Declarations Easy	Code Complete
93	Check variable names	Declarations Easy	Code Complete
94	Ideally, declare and define each variable close to where it's first used	Declarations Easy	Code Complete
95	Use final or const when possible	Declarations Easy	Code Complete
96	Pay special attention to counters and accumulators	Declarations Easy	Code Complete
97	Initialize a class's member data in its constructor	Declarations Easy	Code Complete
98	Check the need for reinitialization	Declarations Easy	Code Complete
99	Initialize named constants once; initialize variables with executable code	Declarations Easy	Code Complete
100	Assign two people to every part of the project	Encouraging Good Coding	Code Complete
101	Review every line of code	Encouraging Good Coding	Code Complete
102	Require code sign-offs	Encouraging Good Coding	Code Complete
103	Route good code examples for review	Encouraging Good Coding	Code Complete

104	Emphasize that code listings are public assets	Encouraging Good Coding	Code Complete
105	Reward good code	Encouraging Good Coding	Code Complete
106	One easy standard	Encouraging Good Coding	Code Complete
107	Requirements developer experience and capability	Encouraging Good Coding	Code Complete
108	Programmer experience and capability	Encouraging Good Coding	Code Complete
109	Team motivation	Encouraging Good Coding	Code Complete
110	Management quality	Encouraging Good Coding	Code Complete
111	Amount of code reused	Encouraging Good Coding	Code Complete
112	Personnel turnover	Encouraging Good Coding	Code Complete
113	Requirements volatility	Encouraging Good Coding	Code Complete
114	Quality of relationship with customer	Encouraging Good Coding	Code Complete
115	User participation in requirements	Encouraging Good Coding	Code Complete
116	Customer experience with the type of application	Encouraging Good Coding	Code Complete
117	Extent to which programmers participate in requirements development	Encouraging Good Coding	Code Complete
118	Classified security environment for computer, programs, and data	Encouraging Good Coding	Code Complete
119	Amount of documentation	Encouraging Good Coding	Code Complete
120	Project objectives (schedule vs. quality vs. usability vs. the many other possible objectives)	Encouraging Good Coding	Code Complete

En la Tabla 16.3 encontramos un listado de 15 técnicas denominadas “Técnicas Avanzadas de Programación” debido a que realiza la combinación de muchos conceptos y patrones de software de distinta complejidad para la construcción de software robustos.

Tabla 16.3. Selección de Técnicas de Avanzadas de Desarrollo de Software.

	Técnicas Avanzadas de Programación	**Concepto**	**Publicación**
121	The Single Responsibility Principle	SOLID	Principles, Patterns, and Practices
122	The Open-Closed Principle	SOLID	Principles, Patterns, and Practices
123	The Liskov Substitution Principle	SOLID	Principles, Patterns, and Practices
124	The Dependency Inversion Principle	SOLID	Principles, Patterns, and Practices
125	The Interface Segregation Principle	SOLID	Principles, Patterns, and Practices
126	Test-first development	Practices	Code Complete
127	TDD - Test-Driven Development	Practices	Test-Driven Development By Example
128	DRY - Don’t Repeat Yourself	Practices	The Pragmatic Programmer
129	KISS - Keep It Simple, Stupid!	Practices	General
130	YAGNI - You Aren’t Gonna Need It	Practices	General
131	Pair Programming	Collaborative Construction	Code Complete
132	Formal Inspections	Collaborative Construction	Code Complete
133	Walk-Throughs	Collaborative Construction	Code Complete
134	Code Reading	Collaborative Construction	Code Complete
135	Dog-and-Pony Shows	Collaborative Construction	Code Complete

6.

Capítulo VI

Propuesta de Solución

El desarrollo del sexto capítulo del presente estudio es el más importante en términos de ejecución de los objetivos específicos, debido que en él se desarrollarán tres de los objetivos. El cuarto objetivo específico, “Adaptar principios de programación en enfoques desarrollo ágil de software para definir arquetipos para el tratamiento de la deuda técnica de código fuente”, se tratará en la fase de Remediación.

El modelo propuesto de gestión y disminución de la DT debe pertenecer a un marco de proceso de mejora continua, puesto que no es posible concebirla como un proyecto puntual para solucionar problemas de desarrollo de software, sino como parte del proceso iterativo de desarrollo. En el capítulo anterior hubo un acercamiento a la definición de la propuesta de modelo con la ejecución del quinto objetivo. Con la definición de la propuesta de modelo se culmina la ejecución del quinto objetivo específico, debido a que finaliza la estructuración los componentes y elementos del modelo para el tratamiento de la DT, así como la descripción de las actividades que componen su gestión diaria.

El sexto objetivo específico del presente trabajo es “Identificar, en el proceso de construcción de software, qué actividades contribuyen a mejorar la eficiencia y disminución de la deuda técnica en los equipos de desarrollo, basado en una técnica de priorización, costos y beneficios”, hace parte de las ocho fases de GDT que componen la propuesta de modelo.

6.1 Motivación

La propuesta de modelo para el tratamiento de la deuda técnica está motivada en la evolución de aquellos componentes que hacen parte de la capa de Backend de la aplicación, implementando una gestión de DT para garantizar una buena calidad técnica del código, ayudando de este como a la sostenibilidad y vida útil de las aplicaciones. Otro hecho de importancia es poder establecer en el equipo de desarrollo e interesados una cultura de trabajo donde sean capaces de reconocer, medir y remediar la DT, acorde a los objetivos estratégicos del proyecto de software, de manera que puede ser gestionada, controlada y que haga parte de un proceso de construcción iterativo, que contribuya a la disminución de fallos y aumento de la calidad en las aplicaciones.

Una de las características más importantes en la definición del modelo y su validación, es que cuente con una definición concisa y de fácil comprensión para los integrantes del equipo de trabajo. En general, este se divide en fases por las cuales se deberá ir avanzando, cumpliendo con diferentes tareas que permitan lograr los objetivos planteados. A lo largo de esta investigación se ha descrito el fenómeno de la DT y su gestión, tanto en términos generales encontrados en la literatura, como en términos específicos del desarrollo individual de esta investigación, basado en la experiencia de problemas identificados en sistemas de información desarrollados en lenguajes como Java o C#.

6.2 Introducción

La DT que se genera en forma inadvertida debe ser minimizada, una forma de lograrlo es capacitando al equipo. La que se genera en forma intencional es la que apalanca los objetivos de negocio, por lo que puede (y debe) gestionarse como cualquier otro recurso. En principio, es necesario el análisis del código del sistema para detectar errores y síntomas de problemas asociados al desarrollo y las malas prácticas de código. Con este diagnóstico, se pasa a una estrategia de priorización y realización de cambios en el código, con el objetivo de solucionar la DT detectada. Finalmente se realiza una evaluación de los cambios desarrollados y su impacto en la calidad del sistema. En la Figura 2 se define la relación entre las fases del modelo para el tratamiento de la DT, dividido en un ciclo de vida de 7 estados de implementación.

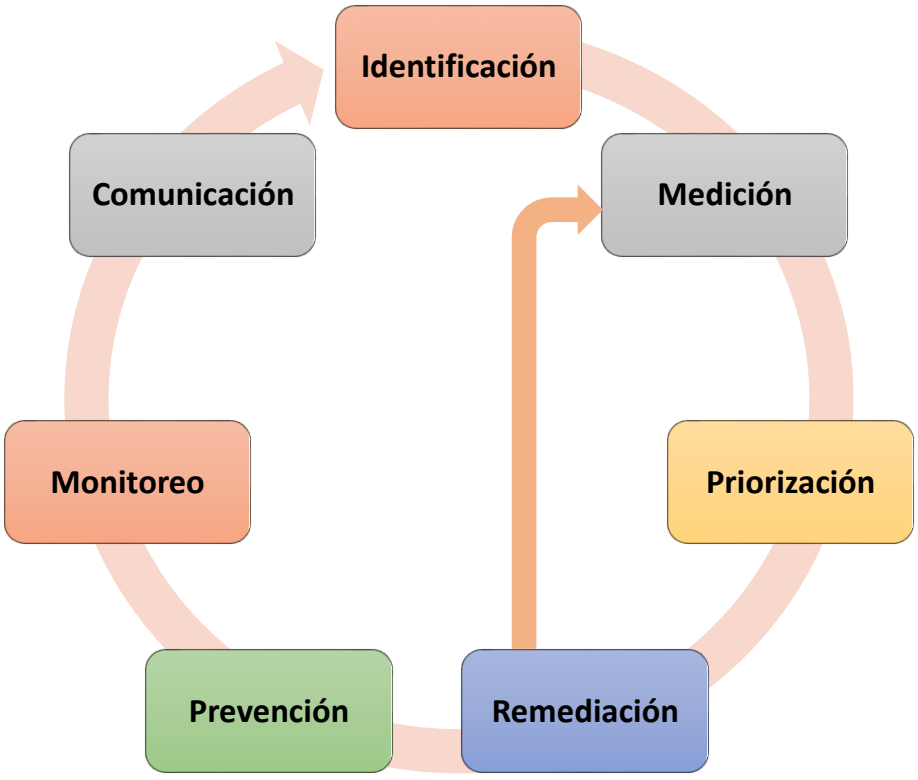


Figura 2. Ciclo de vida de las 7 fases del modelo propuesto.

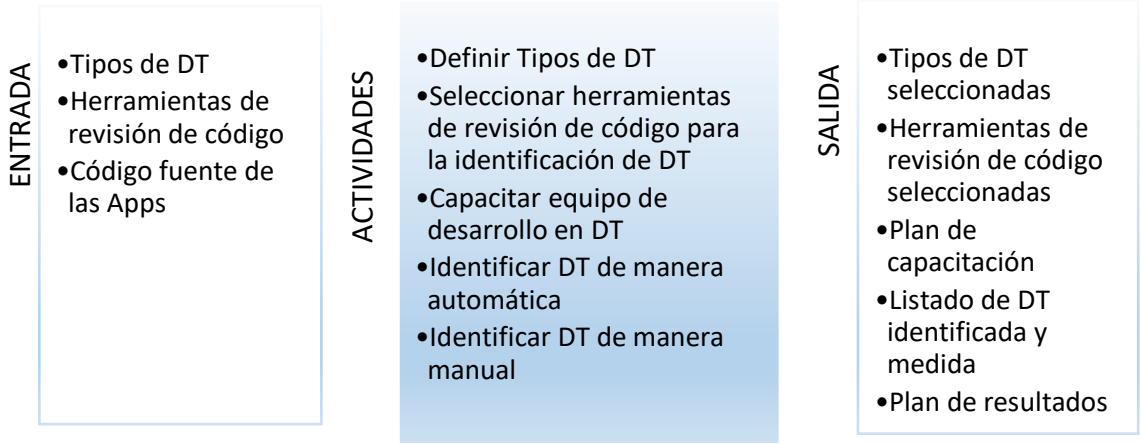
6.3 Definición de las Sietes Fases del Modelo

La división de estructura del ciclo de vida del producto de la investigación llamado “Modelo para el tratamiento de la deuda técnica orientado a la evolución de los componentes para que las aplicaciones sean sostenibles a largo plazo”, son un conjunto de siete fases que están definidas para un proceso de desarrollo de software centrado en los principios de calidad, productividad y mejora continua de los componentes de software. La calidad del código se obtiene conociendo las principales deficiencias que tiene el código fuente de nuestro software y cuáles son las mejores prácticas y técnicas para solucionarlos. La productividad del equipo de desarrollo debe ir aumentando a medida que se vaya remediando la DT y se adopte una cultura de GDT, donde las tareas de mantenimiento y corrección de errores generados por mala programación sean mínimos. La mejora continua radica en desarrollar actividades de prevención y monitoreo sobre la DT, propiciando las buenas prácticas de codificación, revisiones de código, cobertura de pruebas y la brecha tecnológica. Una explicación rápida de las fases del modelo radica en definir unos parámetros de entrada, necesarios para el cumplimiento de las actividades en las que se descompone cada fase. La ejecución de las actividades y tareas del proceso, dará como resultado unos parámetros de salida que son necesarios para determinar las acciones de entre todas las fases del modelo.

6.3.1 Fase de Identificación

Esta es la primera fase del ciclo de vida de gestión y tratamiento de la DT, en términos de acercamientos al concepto de DT, los tipos de problemas que resuelve y la ventaja estratégica como aporte de conocimiento para las personas que participan en ella. Se debe determinar qué tipos de DT se van a trabajar y cuáles de estas son difíciles de identificar por en la revisión de código, de modo que necesiten un trabajo adicional de algún integrante del equipo para ser identificada. Para el control de las reglas de buenas prácticas de codificación se debe realizar una selección de las herramientas de revisión de código estático para el proyecto de software, efectuando las configuraciones específicas de identificación de DT. Algunas de las actividades de esta fase requieren de gran esfuerzo y solo se ejecutan una única vez, quedando así otras actividades que serán ejecutadas en todo el ciclo de vida de la GDT. Si la organización ya cuenta con un proceso de identificación de DT o integración continua ya establecido, se simplifica considerablemente el tiempo de ejecución de esta fase en lo que se refiere a la selección de las herramientas y solo se concentra en las configuraciones especificas del proyecto.

6.3.1.1 Actividades vinculadas a la Fase de Identificación



6.3.1.2 Tareas y Roles vinculados a la Fase de Identificación

Tabla 17. Descripción de tareas y actividades de la fase de identificación.

Tareas	Roles Participantes	Actividad
1. Conocer los tipos de deuda técnica 2. Definir tipos de deuda técnica por importancia estratégica 3. Seleccionar tipos de deuda técnica a disminuir en el proyecto	• Líder Técnico • Líder de Proyecto	**Definir Tipos de DT** En la literatura se describen al menos 13 tipos de DT, de las cuales la de Arquitectura, Diseño, Código y Pruebas son las más comunes e importantes.
4. Conocer las herramientas para la identificación de DT según los tipos seleccionadas 5. Conocer las herramientas que mejor se adaptan a tu infraestructura 6. Seleccionar las herramientas para la identificación de DT 7. Configurar las herramientas y aplicaciones para la identificación de DT	• Líder Técnico • Desarrollador	**Seleccionar herramientas de revisión de código para la identificación de DT** En el mercado podemos encontrar herramientas para ambientes de desarrollo JavaEE tales como Jenkins, Sonar, etc.
8. Definir plan de capacitación 9. Capacitar sobre los tipos de DT seleccionadas 10. Capacitar sobre las herramientas para la detección de DT 11. Explicar las métricas de DT 12. Capacitar en el uso de técnicas de buena calidad de desarrollo 13. Definir plan de capacitación	• Líder Técnico • Desarrollador	**Capacitar a equipo de desarrollo en DT** Capacitaciones en principios básicos de DT, enfocados a temas de identificación y monitoreo de DT
14. Ejecutar revisión de código estático para la identificación de DT 15. Ir a plan de relacionamiento de DT	• Líder Técnico • Desarrollador	**Identificar DT de manera automática** Identificar deuda técnica con revisión de código estático
16. Revisión de deficiencias de código relacionados con DT 17. Definir tiempos para la inspección manual de código 18. Ir a plan de relacionamiento de DT	• Líder Técnico • Desarrollador	**Identificar DT de manera manual** Identificar deuda técnica con revisiones de criterio de experto

6.3.1.3 Diagrama Fase de Identificación

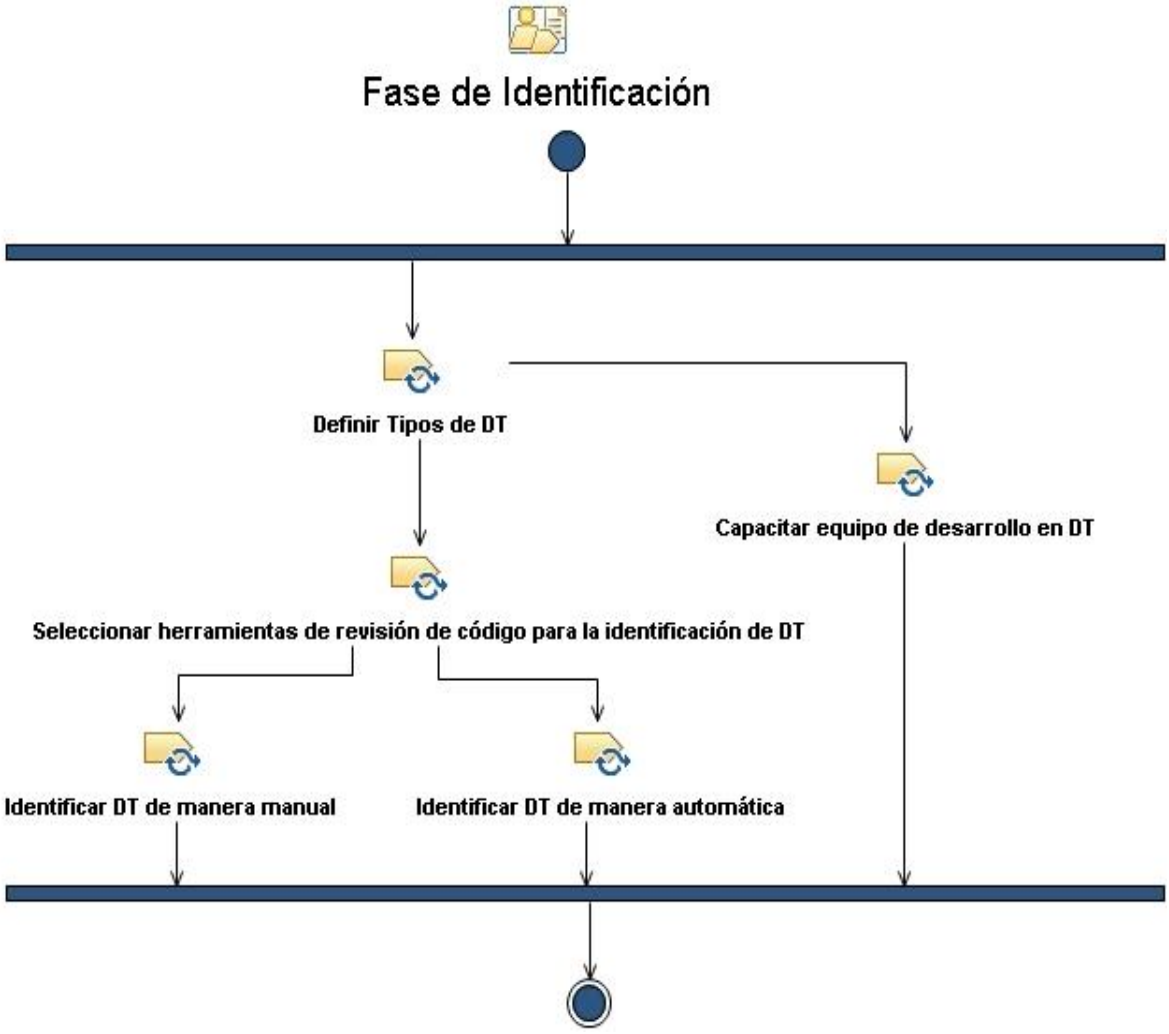


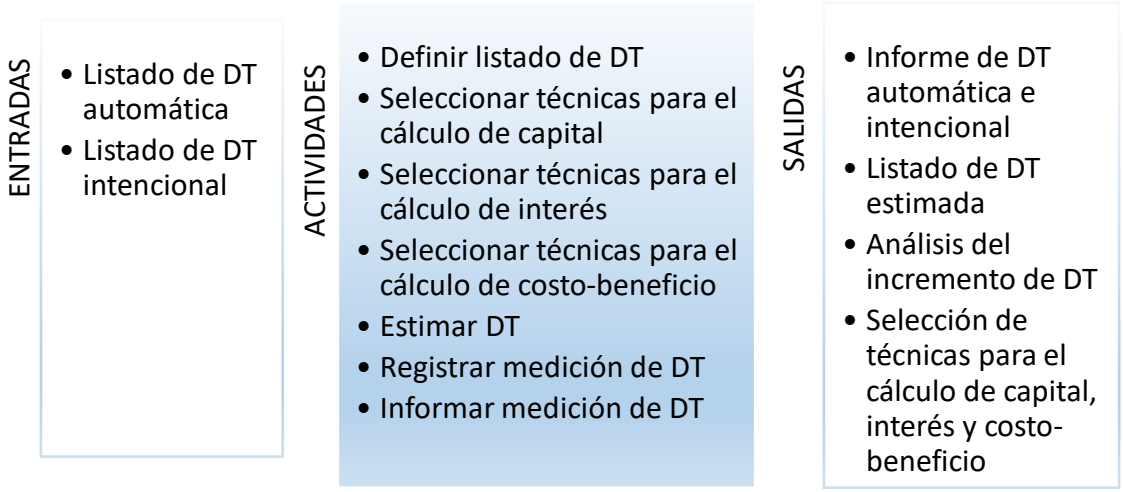
Figura 3. Diagrama de actividades de Identificación con entradas y salidas.

6.3.2 Fase de Medición

En esta fase se calculan los costos y beneficios por el tratamiento de la DT, mediante técnicas de estimación. El Costo se puede obtener estimando el Capital en términos de esfuerzo o bancarios. Las herramientas de análisis de código pueden calcular el Capital en aspectos que son automatizables (violaciones de buenas prácticas, código duplicado, etc.). Las mediciones obtenidas de las herramientas solo representan una porción de la DT, otras mediciones a realizar son temas asociados a malas decisiones efectuadas a nivel de arquitectura, por lo que se recomienda revisar la DT detectada por las herramientas para identificar si aplica una medición adicional por criterio de experto (Sharma & Samarthyam, 2015).

Otro aspecto relacionado a los costos es el Interés (esto es, el esfuerzo extra requerido por no haber pagado el Capital de la DT), que es de difícil estimación. Además de la calidad interna del sistema, el esfuerzo extra podría llegar a estar influido por muchos otros factores que disminuyen la productividad del equipo, por ejemplo, cantidad de soporte y tareas de mantenimiento del sistema, accesos a base de datos para corregir información mal almacenada, cantidad injustificada de interrupciones, tales como envió de emails o llamadas telefónicas que haya que contestar. El interés es relevante sobre los cambios en nuevos requerimientos que afecten componentes con importante DT. Los beneficios en una medición más subjetiva que la del Capital: la tarea es dar un sentido positivo a la DT encontrada, entonces se debe tomar como un supuesto que esta DT ha sido generada en forma intencional, es decir que representa una estrategia para alcanzar un objetivo particular y justifica el hecho de haber aumentado la DT en el sistema.

6.3.2.1 Actividades vinculadas a la Fase de Medición



6.3.2.2 Tareas y Roles vinculados a la Fase de Medición

Tabla 18. Descripción de tareas y actividades de la fase de medición.

Tareas	Roles Participantes	Actividad
1. Recopilar información de la revisión de código 2. Seleccionar archivos que presentan DT según inventario de identificación 3. Seleccionar la DT que se relaciona con los módulos y desarrollos estratégicos	• Líder Técnico • Líder Monitoreo-Prevención • Desarrollador	**Definir listado de DT** Se toma un listado del análisis realizado en la identificación de DT
4. Seleccionar métodos para la estimación de capital 5. Aplicar métodos para la estimación de capital 6. Estimar capital en DT de código 7. Registrar y documentar estimación de capital	• Líder Monitoreo-Prevención	**Seleccionar técnicas para el cálculo de capital** El cálculo de capital es aquel que estima el costo de remediar la DT sobre alguna infracción
8. Seleccionar métodos para la estimación de interés 9. Aplicar métodos para la estimación de interés 10. Estimar interés en DT de código 11. Registrar y documentar estimación de interés	• Líder Monitoreo-Prevención	**Seleccionar técnicas para el cálculo de interés** El cálculo de interés es una estimación del costo que va transcurriendo en el tiempo sobre la DT que no es intervenida
12. Seleccionar métodos para la estimación según costo-beneficio 13. Aplicar métodos para la estimación según costo-beneficio 14. Estimar costo-beneficio en DT de código 15. Registrar y documentar estimación según costo-beneficio	• Líder Monitoreo-Prevención	**Seleccionar técnicas para el cálculo de costo-beneficio** El costo-beneficio tiene en cuenta las oportunidades que se pueden obtener sobre el pago de algunos ítems de DT
16. Utilizar criterio de experto para la estimación de subtipos de DT 17. Registrar y documentar la estimación para subtipos de DT 18. Definir escala de medición 19. Definir el modo en cómo se calculará la velocidad 20. Calcular el reembolso de DT por iteración	• Líder Monitoreo-Prevención • Desarrollador	**Estimar DT** Es dar un valor según las estrategias de estimación seleccionadas. La estimación se toma desde la revisión de código estático y de las valoraciones de criterio de experto
21. Registrar medición de DT en backlog 22. Gestión del backlog para la medición de DT	• Desarrollador • Líder Proyecto	**Registrar medición de DT**
23. Informar la estimación de DT a los integrantes del proyectos	• Líder Proyecto	**Informar medición de DT**

6.3.2.3 Diagrama Fase de Medición

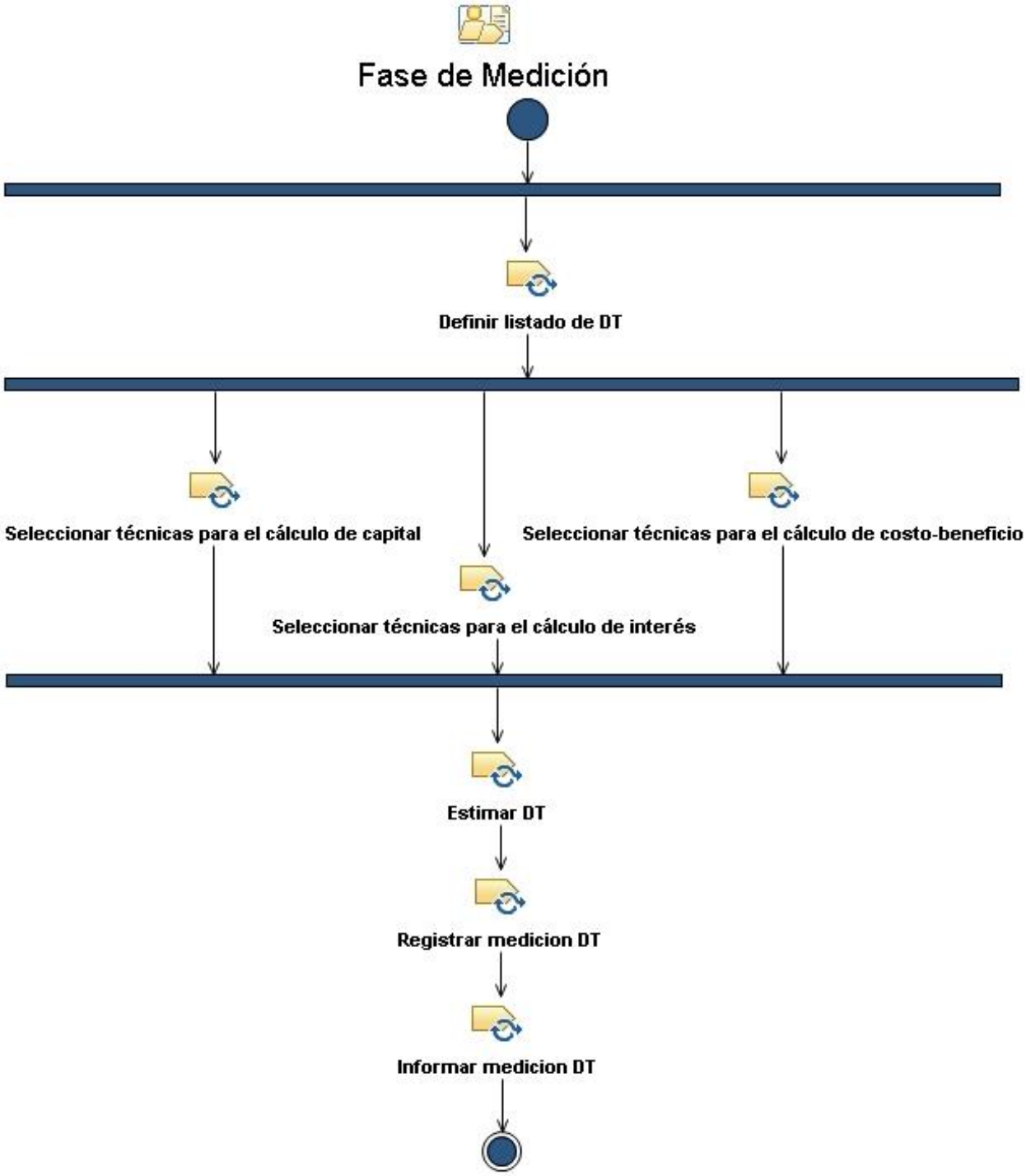
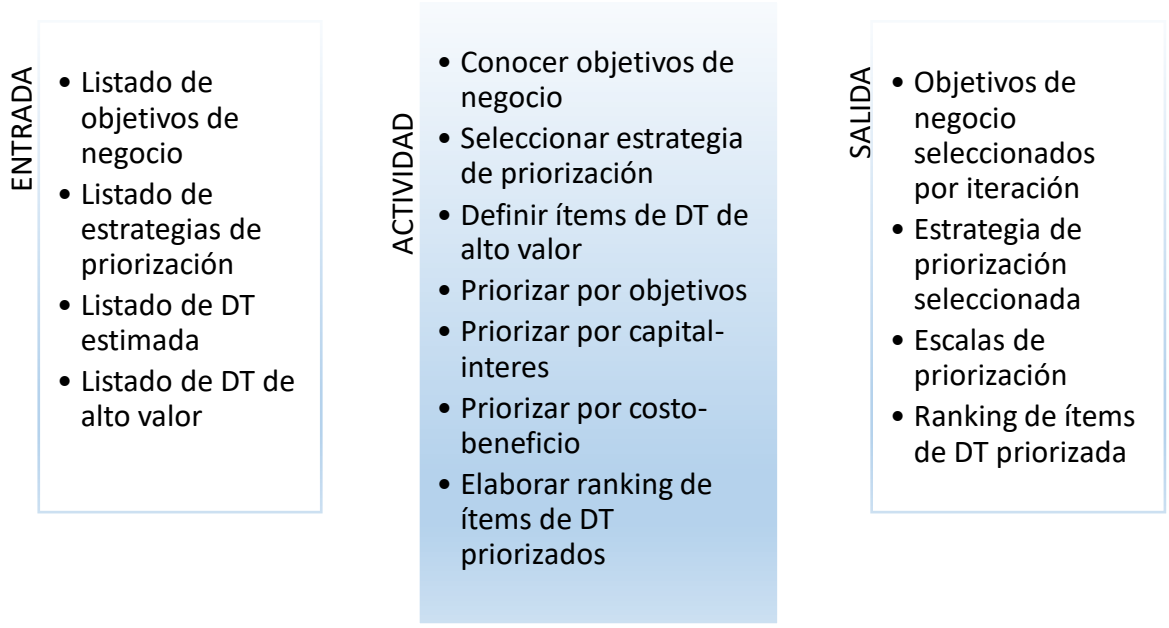


Figura 4. Diagrama de actividades de Medición con entradas y salidas.

6.3.3 Fase de Priorización

En esta fase se construye un ranking de ítems de DT para establecer el orden en el cual deben ser solucionados en el corto tiempo y cuáles pueden ser tolerados para futuras entregas. Una vez identificados y estimados el ranking de ítems de DT, es necesario priorizarlos por algún criterio para poder concentrar el foco en cumplir con los objetivos de negocio. Las herramientas de automatización priorizan algunos fragmentos de código en diferentes niveles de severidad (crítica, mayor, menor, informativa), esto puede ser utilizado como estrategia de pago, donde primero se soluciona las DT de mayor severidad y al mismo tiempo se consideren las de menor esfuerzo posible. Otra priorización posible es asignar una escala cualitativa de una funcionalidad o artefacto de software, dependiendo de la comparación entre el capital de la DT y el costo estimado de desarrollar nuevamente la funcionalidad.

6.3.3.1 Actividades vinculadas a la Fase de Priorización



6.3.3.2 Tareas y Roles vinculados a la Fase de Priorización

Tabla 19. Descripción de tareas y actividades de la fase de priorización.

Tareas	Roles Participantes	Actividad
1. Establecer los objetivos de negocio para una iteración 2. Identificar qué DT puede ser relacionada con el desarrollo de los objetivos de negocio	• Analista de Negocio • Dueño de Producto • Líder de Proyecto	**Conocer objetivos de negocio** Conocer de antemano los objetivos de negocio será de gran valor para mejorar la priorización
3. Especificar estrategia de priorización según la severidad de la DT según por el análisis automático de código (ejemplo: crítica, mayor, menor o informativa) 4. Especificar estrategia de priorización relacionada con el cumplimiento de objetivos de negocio	• Analista de Negocio • Líder de Proyecto	**Seleccionar estrategia de priorización** Definir cuáles son los aspectos más importantes a la hora de priorizar el pago de DT
5. Crear listado de ítems de DT que se desean revisar o solucionar para mínimo una iteración	• Analista de Negocio	**Definir ítems de DT de alto valor**

6. Discutir importancia del ítem de DT y asignarle un peso o valor	• Desarrollador	Antes de iniciar una iteración de desarrollo de producto se puede realizar la validación de que ítems de DT son imprescindibles
7. Organizar un listado de los ítems de DT de alto valor según su peso o valor por objetivo de negocio	• Desarrollador • Líder de Proyecto	**Priorizar por objetivos**
8. Organizar un listado de los ítems de DT de alto valor según su peso o valor por capital-interés	• Desarrollador • Líder de Proyecto	**Priorizar por capital-interés**
9. Organizar un listado de los ítems de DT de alto valor según su peso o valor por costo-beneficio	• Desarrollador • Líder Proyecto	**Priorizar por costo-beneficio**
10. Elaborar un listado único en el backlog que reúna los que ítems de DT por objetivos de negocio, capital-interés y costo-beneficio para mínimo una iteración.	• Desarrollador • Líder Proyecto	**Elaborar ranking de ítems de DT priorizados**

6.3.3.3 Diagrama Fase de Priorización

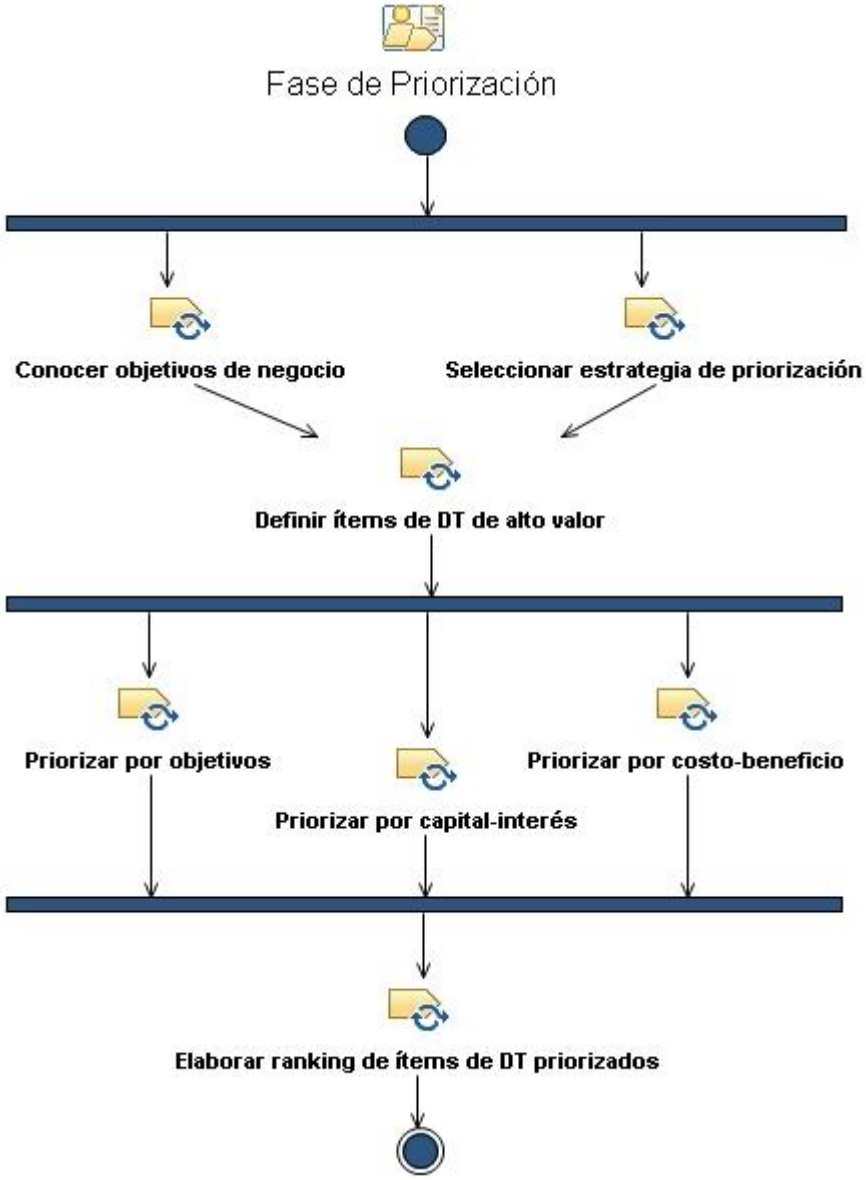
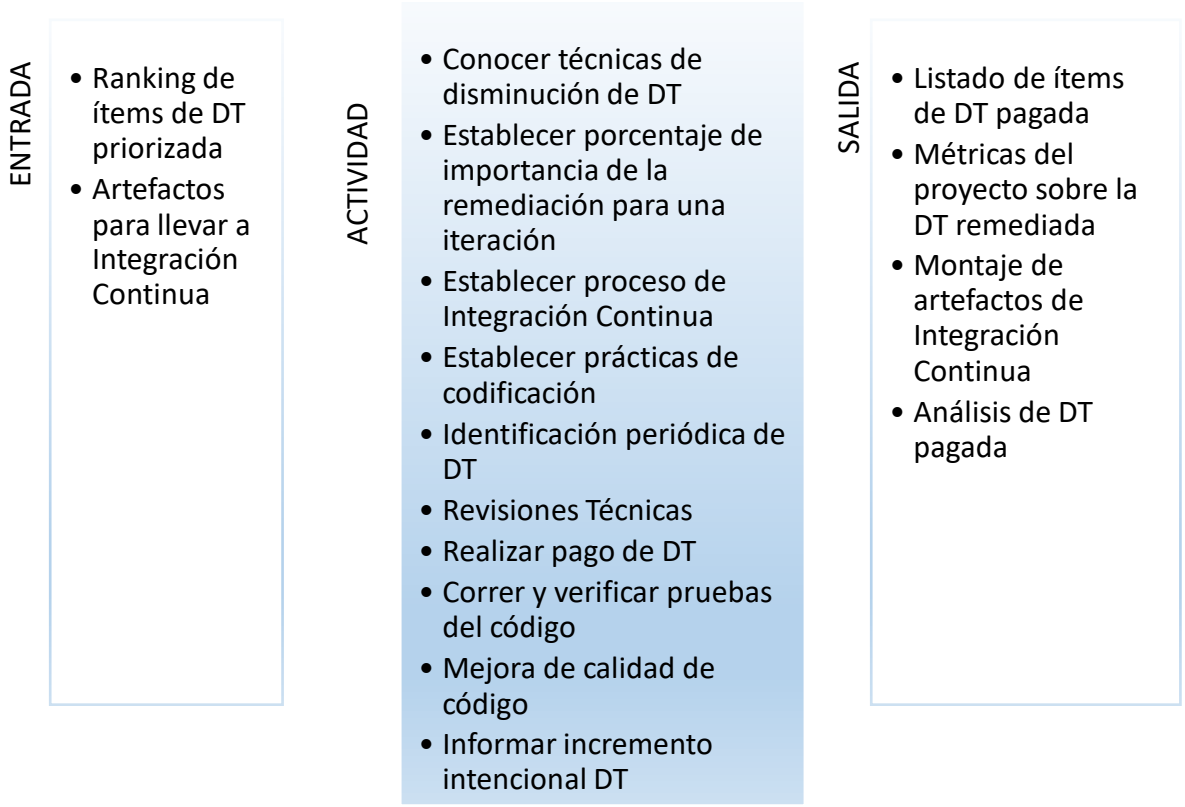


Figura 5. Diagrama de actividades de Priorización con entradas y salidas.

6.3.4 Fase de Remediación

En esta fase es donde la DT es solucionada o mitigada. Se debe realizar el pago de la DT estimada y en casos excepcionales se pueden realizar remediaciones no priorizadas que al mismo tiempo estén relacionadas con la remediación actual y que sean fáciles de pagar. Implica modificar los artefactos que poseen DT utilizando buenas prácticas de desarrollo de software, por ejemplo: Refactorización, TDD, Limpieza de código, entre otras. No es necesario solucionar todos los ítems de DT encontrados. Debe notarse que la remediación implica contar con la conformidad del cliente, y en la mayoría de los casos no generar nueva DT, o si ocurre ésta debería tener menor severidad que la que se está pagando.

6.3.4.1 Actividades vinculadas a la Fase de Remediación



6.3.4.2 Tareas y Roles vinculados a la Fase de Remediación

Tabla 20. Descripción de tareas y actividades de la fase de remediación.

Tareas	Roles Participantes	Actividad
1. Identificar y seleccionar las técnicas de disminución de DT apropiadas según el paradigma de programación	• Desarrollador • Líder Técnico	**Conocer técnicas de disminución de DT**
2. Realizar capacitaciones y ejercicios de las técnicas de disminución de DT seleccionadas		

3. Definir el nivel de importancia de la remediación de DT para la construcción de nuevas funcionalidades	• Líder Técnico • Líder de Proyecto	**Establecer porcentaje de importancia de la remediación para una iteración**
4. Establecer porcentaje mínimo de tiempo de desarrollo para realizar el pago de DT		
5. Definir artefactos de software relacionados con la DT y demás que sean relevantes para la integración continua	• Líder Técnico • Desarrollador	**Establecer proceso de Integración Continua**
6. Seleccionar herramientas para la integración continua		
7. Definir flujo de integración continua		
8. Integración al flujo de integración continua		
9. Búsqueda de buenas prácticas de codificación óptimas para la remediación de DT	• Líder Técnico • Desarrollador	**Establecer prácticas de codificación**
10. Escoger buenas prácticas para la remediación de DT		
11. Ejecutar periódicamente la revisión de código estático para la identificación de DT	• Desarrollador	**Identificación periódica de DT**
12. Reunirse con arquitectos o líderes técnicos para resolver inquietudes y dar retroalimentación en el pago de la DT	• Líder Técnico • Desarrollador	**Revisiones Técnicas**
13. Reunión con equipo de desarrollo para discutir sobre una solución a un problema específico		
14. Discusión y revisiones de código compartido entre los desarrolladores		
15. Pagar deuda técnica priorizada	• Desarrollador	**Realizar pago de DT**
16. Implementar técnicas y buenas prácticas de codificación para disminuir la DT		
17. Implementar aspectos de calidad interna, tales como refactorización, automatización de pruebas, DevOps, seguridad y rendimiento		
18. Visualizar las métricas de DT después de realizado el pago		
19. Verificar y correr pruebas automatizadas del código remediado o nuevo construido	• Desarrollador	**Correr y verificar pruebas del código**
20. Reunión periódica para encontrar otros aspectos distintos de la DT en el código que contribuyan a la mejora de calidad del código	• Líder Técnico • Desarrollador	**Mejora de calidad de código**
21. Recopilar de la base de código el incremento intencional y no intencional de DT	• Desarrollador • Líder de Proyecto	**Informar incremento intencional de DT**

6.3.4.3 Diagrama Fase de Remediación

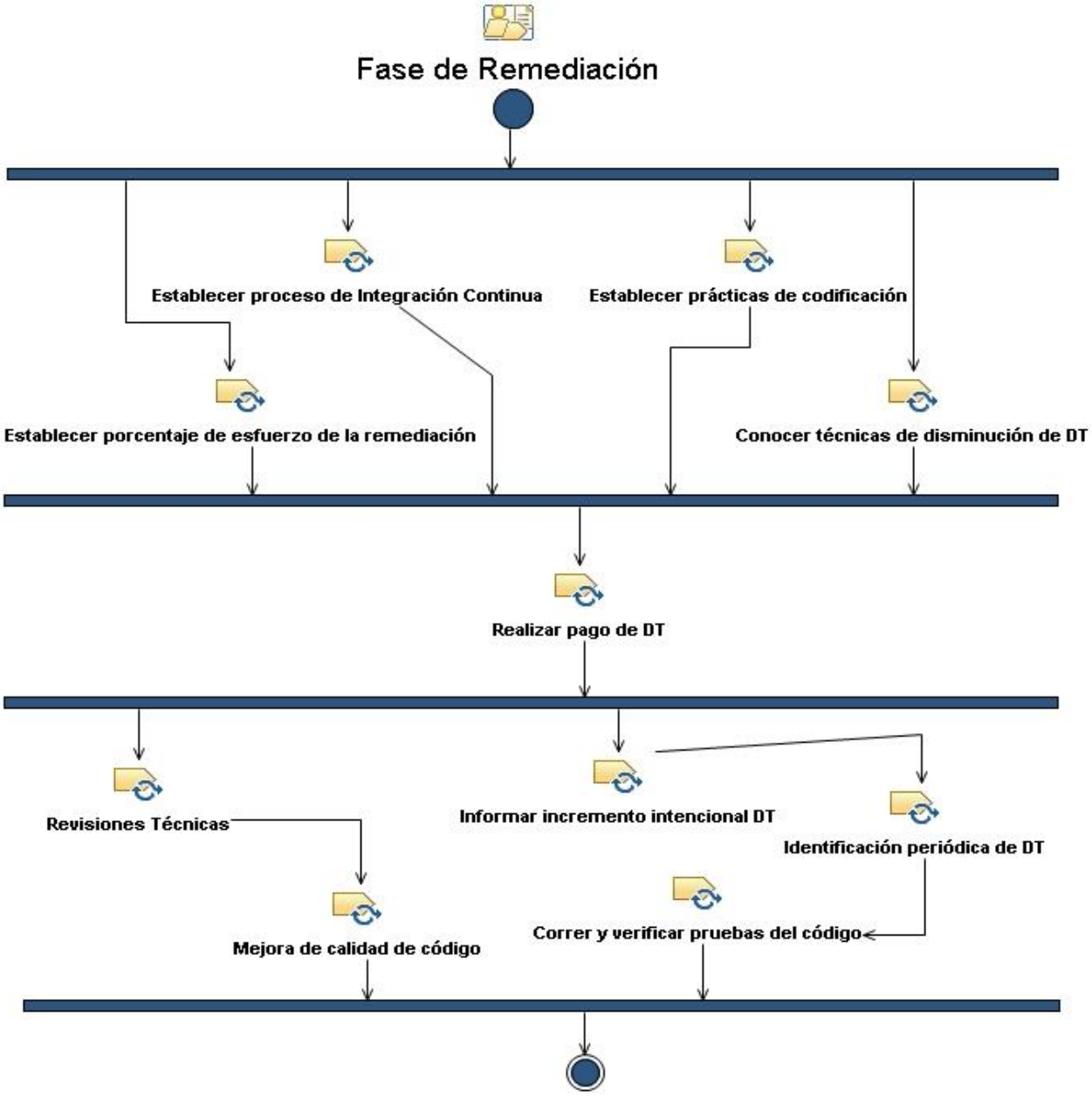


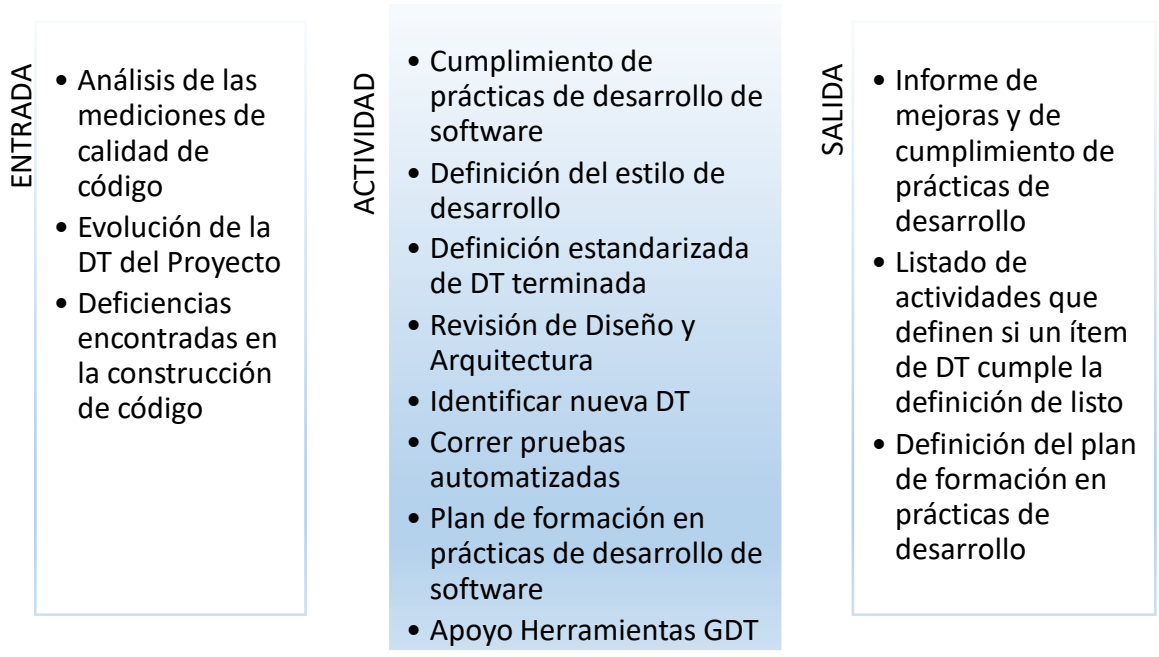
Figura 6. Diagrama de actividades de Remediación con entradas y salidas.

6.3.5 Fase de Prevención

Esta actividad consiste, principalmente, en evitar la generación de nueva DT, además de ir reduciendo la DT identificada con la ejecución de tareas de remediación, entre otras. La prevención es una actividad que resulta más efectiva cuando el equipo de trabajo está capacitado y es disciplinado en cuanto a la aplicación de buenas prácticas de desarrollo de software. Un aspecto importante a tener en cuenta en la prevención es la aparición de DT de forma inadvertida y no intencional, la cual debe ser corregida de la manera más rápida posible y aprender de la causa que originó la deuda para no cometerla de nuevo. Solo es aceptable la aparición de nueva DT si ésta representa algún beneficio estratégico. Algunas de las estrategias de prevención típicamente usadas son (Ambler, 2015):

1) Arquitectura liviana por adelantado.
2) La DT es considerada cuando se diseña.
3) Equipo incluye un Líder Técnico o de Arquitectura.
4) Equipo trabaja con Arquitectos Empresariales.
5) Miembros del equipo capacitados en DT.
6) Modelado detallado de arquitectura por adelantado.

6.3.5.1 Actividades vinculadas a la Fase de Prevención



6.3.5.2 Tareas y Roles vinculados a la Fase de Prevención

Tabla 21. Descripción de tareas y actividades de la fase de prevención.

Tareas	Roles Participantes	Actividad
1. Prevención de deuda técnica utilizando estándares de codificación y prácticas de revisión de código 2. Construir un plan de motivación y estrategias de incentivos	• Desarrollador • Líder Técnico	**Cumplimiento de prácticas de desarrollo de software** No se puede asegurar una buena GDT sin la adecuada definición de un plan de prevención que ataje el crecimiento de DT
3. Definir un estilo común de desarrollo para el proyecto	• Líder Técnico • Líder de Proyecto	**Definición del estilo de desarrollo**
4. Definición de criterios estandarizados por el proyecto sobre la calidad del código entregado en la remediación de DT, de manera que ayude a mitigar la reescritura del código.	• Líder de Proyecto • Desarrollador	**Definición estandarizada de DT terminada** Para establecer si las actividades se remediación de DT si son terminadas correctamente, se debe definir el cumplimiento de unos criterios de calidad de código
5. Reunirse con arquitectos o líderes técnicos para resolver inquietudes y dar retroalimentación de diseño y arquitectura relacionada con la DT	• Líder Técnico • Desarrollador	**Revisión de Diseño y Arquitectura**
6. Ejecutar periódicamente el revisión de código estático para la identificación de DT	• Desarrollador	**Identificar nueva DT**
7. Correr pruebas automatizadas del código como medida de prevención 8. Corregir pruebas automatizadas si se encuentra alguna anomalía	• Líder Técnico • Desarrollador	**Correr pruebas automatizadas** Por cada desarrollo e integración del código fuente al repositorio es importante lanzar las distintas pruebas automáticas
9. Definir un plan de capacitaciones concertados con equipo de desarrollo e interesados de negocio 10. Visualizar las métricas de DT después de realizado el pago	• Líder Monitoreo-Prevención	**Plan de formación en prácticas de desarrollo de software** Continuar capacitaciones sobre buenas prácticas de desarrollo de software
11. Recoger información de la remediación de la DT y seguir definiendo estrategia de gestión de la DT	• Líder Monitoreo-Prevención • Líder de Proyecto	**Apoyo Herramientas GDT**

6.3.5.3 Diagrama Fase de Prevención

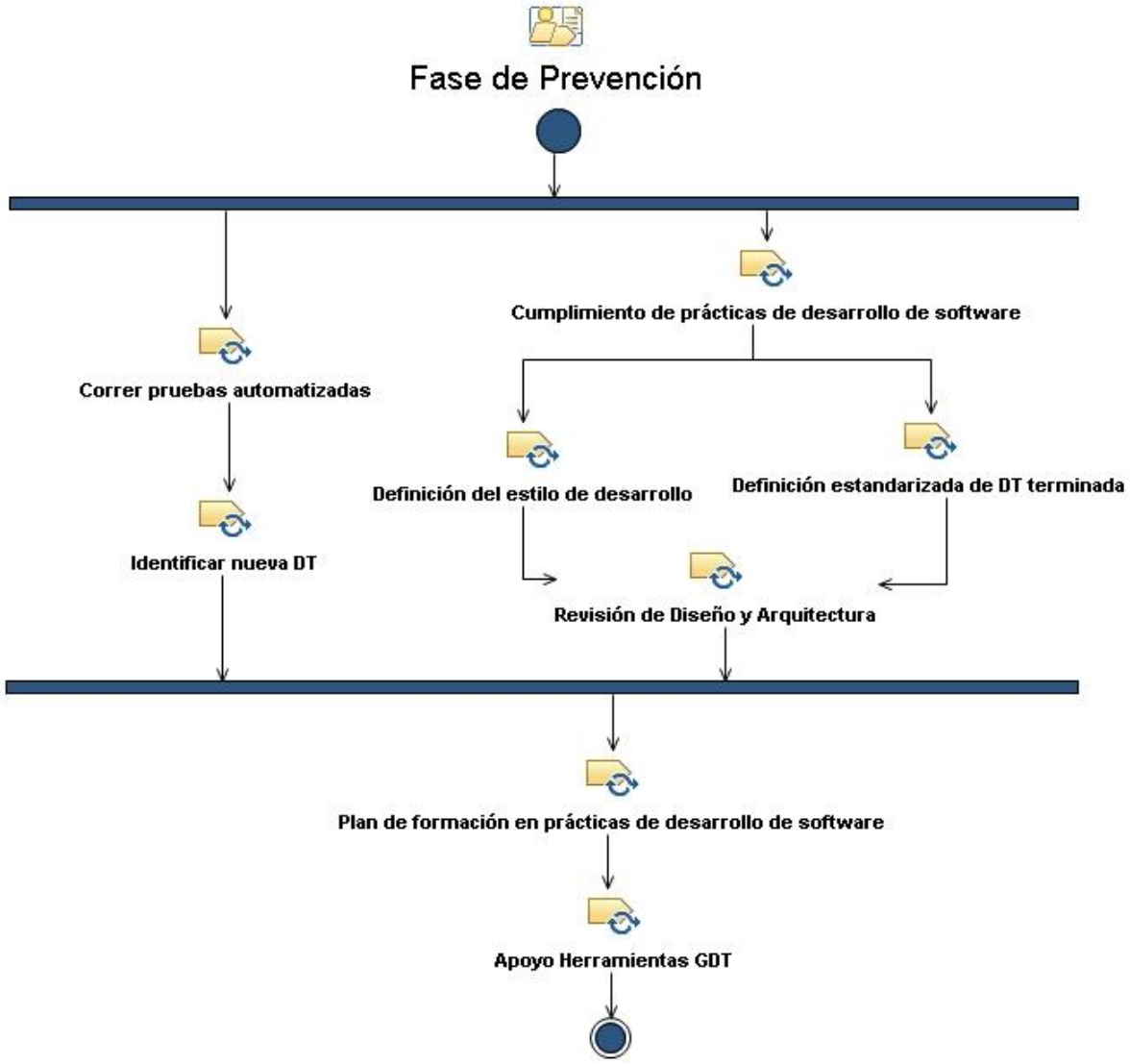
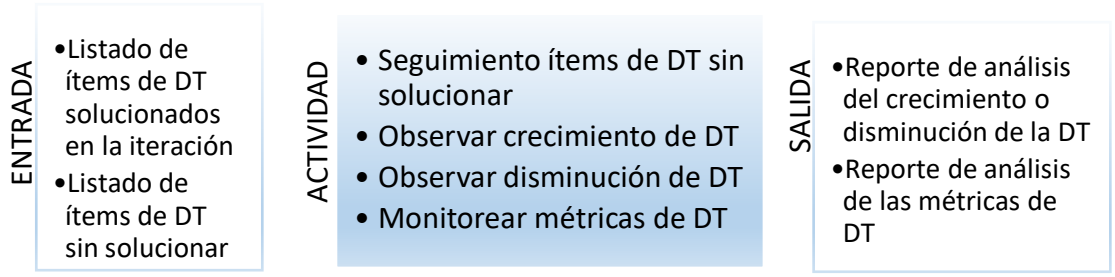


Figura 7. Diagrama de actividades de Prevención con entradas y salidas.

6.3.6 Fase de Monitoreo

El monitoreo de la DT consiste en efectuar tareas de observación sobre los ítems de DT que han sido solucionados, para determinar cómo sus progresos han contribuido con los objetivos técnicos y de negocio. Comprobar cómo la implementación de las otras fases del modelo contribuye a una mejora de las métricas en DT, ayuda a entender cómo se han realizado las estimaciones que herramientas o personas han establecido. También es importante realizar observaciones de lo que se considera aumento de la DT en el periodo actual, para determinar luego su causa y efecto. Los ítems de DT que no han sido solucionados deben evaluarse para determinar su evolución en términos de costo y beneficio. Los artefactos que no han sido pagados o donde la DT se ha remediado parcialmente, son objeto de monitoreo.

6.3.6.1 Actividades vinculadas a la Fase de Monitoreo



6.3.6.2 Tareas y Roles vinculados a la Fase de Monitoreo

Tabla 22. Descripción de tareas y actividades de la fase de monitoreo.

Tareas	Roles Participantes	Actividad
1. Analizar los ítems de DT para futuras iteraciones, y determinar si la solución de estos tienen beneficios a corto plazo	• Líder Monitoreo-Prevención	**Seguimiento a ítems de DT sin solucionar** Esta actividad se debe combinar con tareas de las fases de medición y priorización, para la toma de decisiones de GDT
2. Identificar el crecimiento de la deuda técnica, en dónde y él porqué	• Desarrollador • Líder de Proyecto	**Observar crecimiento de DT**
3. Activar tareas de medición y priorización		
4. Identificar las características del trabajo realizado para disminuir la DT	• Líder de Proyecto	**Observar disminución de DT**
5. Revisión constante de las medidas (o mediciones) de DT en una iteración, para no permitir un crecimiento constante de DT	• Líder Monitoreo-Prevención • Desarrollador • Líder de Proyecto	**Monitorear métricas de DT** Los principales actores en actividades de GDT deben estar monitoreando constantemente la métricas de DT y tomar las acciones necesarias cuando no se están cumpliendo los objetivos propuestos
6. Documentar cómo evolucionan las mediciones de DT para la toma de acciones y decisiones		

6.3.6.3 Diagrama Fase de Monitoreo

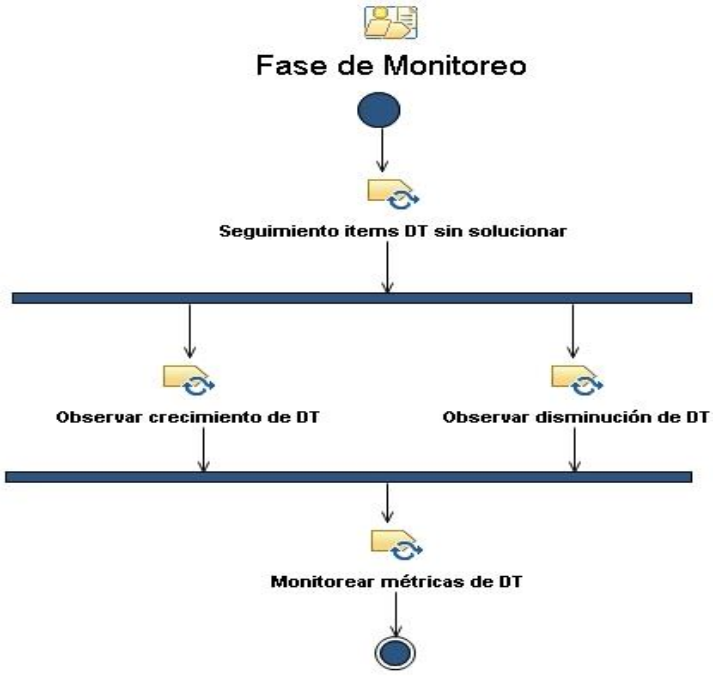
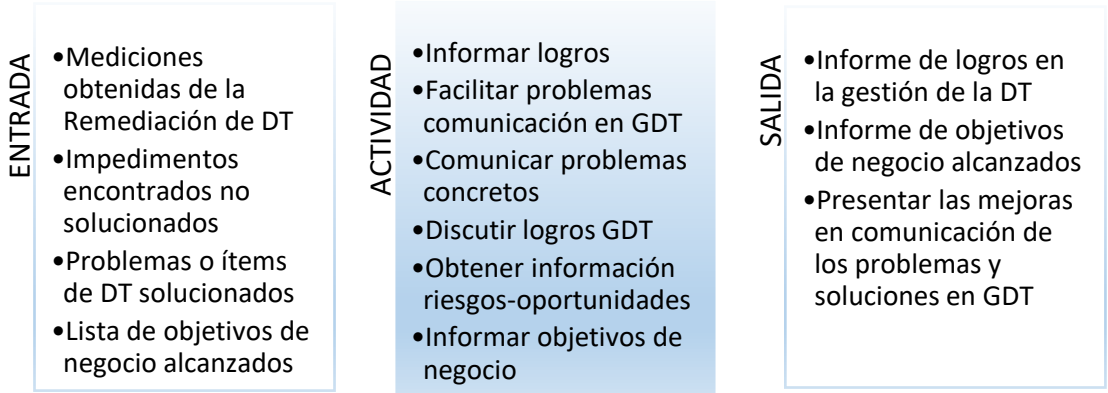


Figura 8. Diagrama de actividades de Monitoreo con entradas y salidas.

6.3.7 Fase de Comunicación

La gestión de la deuda técnica permite dar transparencia al proceso de desarrollo de software, debido a que otorga visibilidad y transparencia de los resultados alcanzados hacia todos los interesados (niveles técnicos, gerenciales, clientes, usuarios, etc.), para que las partes puedan ser debidamente informadas y los problemas puedan ser discutidos. Se debe contar siempre con evidencias y problemas concretos, acompañados de su nivel de severidad, urgencia, costos y beneficios, para así determinar los riesgos y la toma de decisiones con la mayor información posible, ya sea decidir pagar la deuda o continuar con los requerimientos del negocio.

6.3.7.1 Actividades vinculadas a la Fase de Comunicación



6.3.7.2 Tareas y Roles vinculados a la Fase de Comunicación

Tabla 23. Descripción de tareas y actividades de la fase de comunicación.

Tareas	Roles Participantes	Actividad
1. Presentar y comunicar a los interesados el trabajo realizado en la disminución de DT	• Líder de Proyecto	**Informar logros**
2. Comunicar con evidencias los problemas concretos	• Desarrollador • Líder de Proyecto	**Facilitar problemas de comunicación** Se deben encontrar espacios en los cuales los desarrolladores expongan las diferencias de criterios, dificultades técnicas o desmotivación en el proceso de GDT
3. Definir el nivel de severidad, urgencia, costos y beneficios para poder evaluar el riesgo		
4. Establecer espacios de comunicación y niveles de confianza entre actores como Desarrolladores, Arquitectos, Cliente, Gerencia, entre otros.		
5. Comunicar los impedimentos o demoras en la gestión y remediación de la DT	• Desarrollador	**Comunicar problemas concretos**
6. Resumen de la gestión de la deuda técnica realizada por el equipo de desarrollo en un periodo concreto	• Líder de Proyecto	**Discutir logros GDT** Dar exposición a los logros obtenidos en la GDT es una buena forma de legitimar la apuesta y el trabajo invertido en el tratamiento de la deuda técnica en los proyectos
7. Enumerar las prioridades identificadas en la gestión y remediación de DT		
8. Informar nuevas evidencias y características a tener en cuenta para la priorización a los interesados		

9. Discutir la importancia que la gestión y remediación de la DT ha tenido en el proyecto	• Desarrollador • Líder de Proyecto	**Obtener información riesgos-oportunidades**
10. Realizar informe de resultados por objetivos de negocios alcanzados debido al pago de deuda técnica	• Líder de Proyecto	**Informar objetivos de negocio**

6.3.7.3 Diagrama Fase de Comunicación

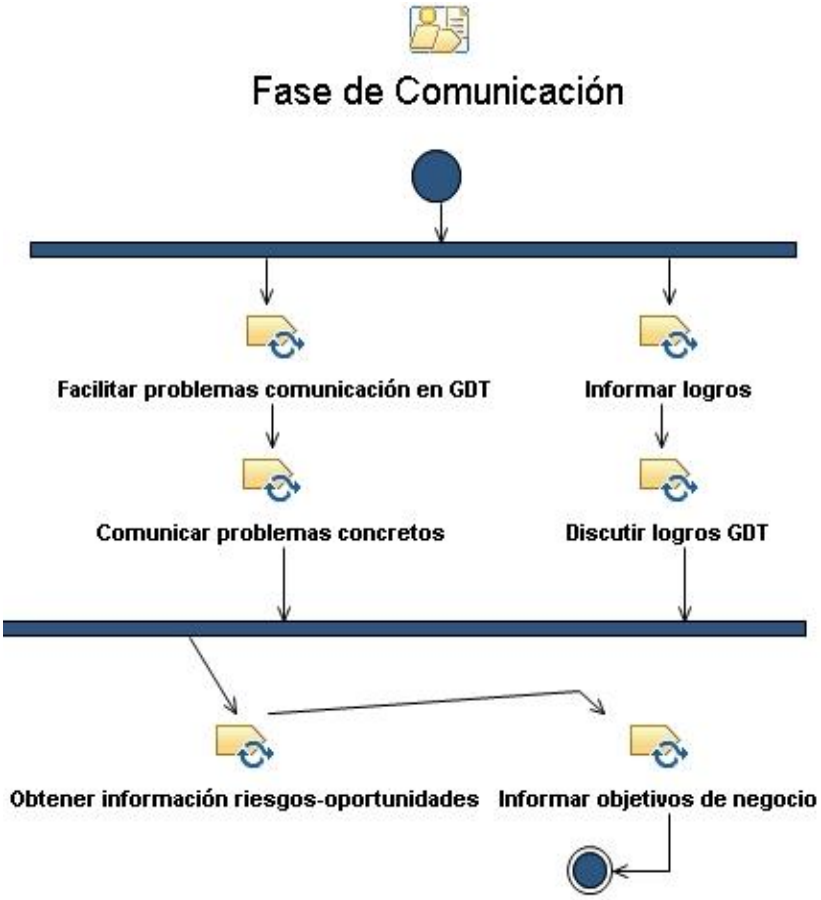


Figura 9. Diagrama de actividades de Comunicación con entradas y salidas.

6.4 Plantillas

Como apoyo a las fases del modelo se desarrollaron un conjunto de plantillas para apoyar la GDT del equipo de desarrollo. En la Tabla 24 se describe una plantilla para la documentación del ítem de DT que puede ser llevado a una herramienta de control de incidentes, tales como Jira, Mantis, Wikis u otro tipo de documentos.

Tabla 24. Plantilla para la documentación de la DT

Ítem	Descripción
Id	Número de identificación de la DT
Nombre	Puede darse una descripción o nombre al ítem de DT identificado, para dar un mayor contexto
Efecto causado	Descripción del comportamiento o anomalía ocasionada por el ítem de DT
HU Asociada	Identificación de la historia de usuario en caso de que esté reportado en algún Backlog
Reportado por	Persona de identifica el ítem de DT, fallo, demora excesiva, entre otros
Impacto	Se puede agregar una explicación de los beneficios y el impacto de solucionar la DT
Conoce quién puede apoyarlo en la solución	Persona o Rol que podrían apoyar sobre la funcionalidad afectada
Solución / Recomendación	Observaciones a tener en cuenta para la solución o recomendaciones de la DT
Prioridad	Nivel de importancia que podría tener el ítem de DT
Fuente (email, imagen, video, etc.)	Documentación que apoye a contextualizar o solucionar la DT
Estimación	Consideración del tiempo que puede tomar solucionar la DT
Responsable	Persona o Rol responsable de remediar la DT
Localización Código	Documentación sobre el proyecto, paquetes, componentes, clases etc., que indique donde se localiza la DT
Localización UI	Si el ítem de DT afecta a alguna UI (User Interface) o se encuentra localizada en alguna pantalla, puede documentarlo aquí
Progreso	Indicar estado actual o si ya existe algún progreso con la remediación de la DT
Otras Consideraciones	Observaciones o documentación adicional

En las tablas 25 y 26 se definen las plantillas para Backlog Técnico Personal (gestionada únicamente por el desarrollador) y Backlog Técnico del Proyecto (gestionada por todo el equipo de desarrollo), respectivamente.

Tabla 25. Plantilla para la documentación del Backlog Técnico Personal

Nombre Columna	Descripción Columna	Obligatorio
Tema	Tema a estudiar	S
Documentación	Enlaces, Libros o Referencias de Documentación	S
Importante	Indicaciones o avisos acerca del tema	N
Prioridad	Importancia del tema	N
¿Motivación Propia?	Tema auto-impuesto por el Desarrollador, dado por una necesidad o gusto personal	N
Estado	Estado en que se encuentra el estudio del tema	N
Otras notas	Espacio destinado para observaciones adicionales o toma de notas	N

Tabla 26. Plantilla para la documentación del Backlog Técnico del Proyecto

Nombre Columna	Descripción Columna	Obligatorio
Identificador Tarea	Identificador único de la tarea	S
Descripción Tarea	Descripción del problema a solucionar o temática a investigar.	S
Prioridad	Importancia de la tarea	S
Estado	Estado en que se encuentra la tarea	S
Observaciones	Observaciones adicionales, Documentación, etc.	N
Responsable	Asignación de persona(s) responsable(s)	N
Estimación	Valor por criterio de experto en tiempo o esfuerzo para terminar dicha tarea	N
Fecha de Entrega	Fecha en la que se necesita una solución o respuesta sobre el ítem de backlog	N

6.5 Conclusión

Con el desarrollo de la propuesta de investigación nombrada “Modelo para el tratamiento de la deuda técnica orientado a la evolución de los componentes para que las aplicaciones sean sostenibles a largo plazo”, se ha logrado engranar un flujo de trabajo de diferentes métodos técnicos de gestión de la deuda marcado con filosofía de desarrollo ágil, tomando como ejemplo, el alto conocimiento de los objetivos técnicos y del negocio, entregas oportunas (o en breve) de alto valor y contribuir con desarrollos de buena calidad técnica que ayude al crecimiento del equipo de desarrollo y a las organizaciones que pertenecen. Cabe señalar que la definición del modelo, a diferencia de otras propuestas de investigación, busca de un modo eficiente ser un apoyo a la reducción de la deuda técnica de código que avale al mismo tiempo el desarrollo continuo de características de producto, en la se pueda verificar la inversión de la DT a futuro convendría a garantizar aplicaciones con un soporte amigable, fácil de mantener y mayor tiempo de vida útil.

7.

Capitulo VII

Validación

Este capítulo corresponde a la ejecución del séptimo objetivo específico del presente estudio, que es la fase de demostración de la propuesta y que valida si el modelo para el tratamiento de la deuda técnica es apto para que las aplicaciones sean sostenibles a largo plazo, con su aplicación en un caso de estudio. El cumplimiento del último objetivo específico intenta dar respuesta a la problemática de investigación planteada, definiendo un diseño del experimento de investigación, definición de la variable independiente, el proceso de recolección de datos, análisis de la validación de las variables dependientes y finalmente se realiza la presentación de los resultados obtenidos.

7.1 Plan de evaluación y validación

La clasificación de variables e indicadores del proceso de validación, se precisa con el siguiente alcance: ***Analizar*** el modelo para el tratamiento de la deuda técnica. ***Con el propósito*** de evaluar su impacto en la calidad técnica en las aplicaciones de software. ***Con respecto*** a la calidad, esfuerzo y satisfacción de aplicar el modelo. ***Desde el punto de vista*** del investigador. ***En el contexto*** de un equipo de desarrollo de software de un ambiente empresarial.

7.1.1 Variable Independiente

Implementación de la propuesta de modelo para el tratamiento de la deuda técnica.

Descripción. La manipulación de la variable independiente, se puede simplificar en la ejecución de cada una fases y actividades definidas en el modelo para el tratamiento de la DT por parte de un grupo propuesto, y que es controlada por parte del investigador para determinar su influencia sobre los fenómenos observados (aumento o disminución de la DT).

7.1.2 Variable Dependiente

Disminución de la deuda técnica de tipo código sobre una aplicación software con tratamiento de deuda técnica

Descripción. La variable dependiente representa todos aquellos cambios que pueden ser observados, analizados y medidos por el investigador y que es influenciado por los distintos

procedimientos y pruebas que se apliquen del modelo de DT propuesto (efecto de la variable independiente). Los efectos causados sobre la variable dependiente serán registrados en la definición de métricas utilizadas para la validación del experimento.

7.1.3 Amenazas a la integridad

Para mitigar las amenazas a la integridad del experimento y mantener un control apropiado de las pruebas, se sugieren las siguientes medidas:

	Amenaza	Medida
1.	**Pocos o nulos conocimientos del tema de investigación:** Los participantes tienen muy pocos conocimientos, y se les puede dificultar demasiado iniciar las actividades a medir.	Realizar una introducción y capacitación de los conceptos claves para que puedan iniciar las actividades de modo autónomo.
2.	**Demasiado conocimiento del área de investigación:** Los participantes poseen demasiados conocimientos de los tópicos de investigación. Esto podría afectar los resultados.	Asegurarse que entre los participantes no hayan personas con mucho conocimiento del tema de investigación.
3.	**Inestabilidad del ambiente experimental:** Las condiciones del ambiente o entorno del experimento no sean iguales para todos los grupos participantes.	Lograr que las condiciones ambientales sean las mismas para todos los grupos.
4.	**Selección:** Que los grupos del experimento no sean equivalentes.	Lograr que los grupos sean equivalentes.
5.	**Deserción:** Que los participantes abandonen el experimento.	Validar la estabilidad de los participantes en el periodo de experimento y reclutar los suficientes para todos los grupos.

7.1.4 Recolección de datos e Instrumentos de medición

El proceso de recolección de datos para verificar la propuesta de investigación, se alimenta de la información obtenida a través de la ejecución de las actividades definidas en las fases del modelo y el uso de herramientas de cálculo o almacenamiento estadístico del código fuente (localización de fuente de información). La mayoría de los datos recolectados se obtendrán en actividades de Remediación de DT y de los reportes del análisis en el código fuente, donde posteriormente se verificará la evolución de la DT (aumento o disminución).

Para especificar un poco más en el instrumento de prueba o medición, este cumplirá con un enfoque cuantitativo de análisis de datos, debido a que las métricas utilizadas en el

experimento son determinadas del Análisis de Contenido, Pruebas de Rendimiento y Estadísticas de Calidad de Código Fuente, de manera que este instrumento debe cumplir tres requisitos para la seguridad de los datos recolectados: **Confiabilidad.** Se refiere al grado en que su aplicación repetida al mismo individuo u objeto produce resultados iguales (consistentes y coherentes). **Validez.** Se refiere al grado en que un instrumento realmente mide la variable que pretende medir evaluando varios tipos de evidencia. **Objetividad.** Aunque es un término aplicado más que todo a ciencias sociales, se refiere al grado en que se puedo permear un instrumento a la influencia de los sesgos y tendencia de los investigadores que lo administran, califican e interpretan.

7.1.5 Métricas utilizadas

Se realiza la selección de cinco métricas a utilizar, que simboliza los datos más relevantes para validar el modelo propuesto (en la sección de **Análisis de Resultados** encontrara los resultados de cada una de las métricas utilizadas):

1. **Nombre:** Valor de la deuda técnica identificada por iteración
   Descripción: El indicador tomado es el valor calculado en la revisión de código estático como la cantidad de DT identificada en una iteración
   Cálculo: Automático
   Unidad de medida: Hora-persona
2. **Nombre:** Remediación de deuda técnica por iteración
   Descripción: El indicador tomado es la cantidad de DT que ha sido pagada o remediada en una iteración
   Cálculo: Automático
   Unidad de medida: Hora-persona
3. **Nombre:** Cantidad total de deuda técnica del proyecto
   Descripción: Es el tiempo total de DT estimado que posee actualmente el proyecto
   Cálculo: Automático
   Unidad de medida: Días-persona (1 día equivale a 24 Horas-persona)
4. **Nombre:** Cubrimiento de código fuente por iteración
   Descripción: El indicador tomado es el porcentaje estimado del cubrimiento por código de pruebas unitarias sobre el código fuente en una iteración
   Cálculo: Manual (se toma de la diferencia del porcentaje total de código entre la iteración anterior y la iteración siguiente)
   Unidad de medida: Porcentaje
5. **Nombre:** Número total de pruebas y code smells
   Descripción: Cantidad total de pruebas unitarias sobre el código fuente y las deficiencias código (code smells) que posee actualmente el proyecto.
   Cálculo: Automático
   Unidad de medida: Cantidad

7.2 Diseño del Experimento

La preparación del equipo de desarrollo para la ejecución del experimento se realizará a través de sesiones de capacitación en las actividades concernientes a la propuesta modelo y objetivos del experimento. Los integrantes del equipo de desarrollo deben ejecutar estas actividades, que serán monitoreadas por un encargado o líder de equipo. Es de vital importancia tener evidencia de que se están llevando a cabo las actividades del experimento para lograr una recolección de datos significativa. En caso de que se evidencie la no ejecución de estas actividades, se debe tener explicación del porqué y revisar si se pueden tomar medidas alternativas para estos casos. En resumen, las actividades concernientes al experimento se dividen en tres nuevos grupos, de la siguiente manera:

1) **Fase de Identificación y Fase de Medición:** Instalación de las herramientas de identificación de DT. Para que sirven y cómo utilizarlas. En algunas ocasiones se deben organizar sesiones de revisiones manuales de código por parte de un integrante del equipo, debido a que existen ciertos indicadores de DT que no pueden ser evaluados por herramientas automatizadas. Las mediciones obtenidas en estas dos primeras fases servirán sobre todo como insumo para la fase de remediación.

2) **Fase de Priorización y Fase de Remediación**: Definición de las estrategias para la priorización de DT. La DT creada en una misma iteración debe ser pagada lo más pronto posible. El pago de la DT o remediación es la fase más importante con respecto a la obtención de métricas y resultados de la implementación del modelo.

3) **Fase de Prevención, Fase de Monitoreo y Fase de Comunicación:** Las actividades de prevención y monitoreo son transversales y son ejecutadas en los ciclos de desarrollo de las fases posteriores. Las actividades de comunicación son opcionales, pueden ser utilizadas como estrategia de comunicación entre los integrantes del equipo de desarrollo. Estas tres fases no arrojan mediciones contundentes para el experimento.

Uno de los aspectos importantes del experimento y éxito del modelo es que se trata de potenciar las habilidades técnicas, para esto, se abrirán espacios de capacitación de personas individuales sobre las falencias encontradas en el proyecto. Un punto importante para conocer si una actividad para la disminución de la DT ha sido completada, es la **Definición de criterio de Hecho**. El criterio de Hecho consiste de aquellas características que debe cumplir un requerimiento, funcionalidad o historia de usuario, definidas ya sea en consenso por el equipo de desarrollo o por alguna área de la compañía encargada del seguimiento y mejora de calidad de código.

7.2.1 Sujetos

Los grupos de trabajo para el experimento deben ser equivalentes en tamaño, conocimiento y experiencia, como mínimo dos para establecer comparaciones equivalentes. Para la muestra se seleccionarán dos equipos de trabajo de proyectos de desarrollo de software, de aproximadamente diez personas cada uno, y divididos de la siguiente manera:

- **Grupo Propuesto:** Equipo de Desarrollo aplicando el modelo para el tratamiento de la DT. Registrar los cambios que se detectan sobre el grupo experimental y comprobar cuál de estos son positivos para seguir implementándolos y cuales son negativos para mejorarlos o retirarlos. Ir aplicando las mediciones, sin que el grupo se sientan controlado u observado.
- **Grupo Base o Control:** Equipo de Desarrollo sin aplicar el Modelo para el tratamiento de la DT. Este grupo implementará un método basado en lo empírico, que adquiere su conocimiento en DT de lo encontrado en la literatura, manuales, portales, recomendaciones generales y otros documentos. Asegurar que este grupo tiene las mismas características que el experimental.

7.3 Herramientas

Las herramientas para el cálculo de DT cumplen un rol fundamental para el desarrollo de su gestión, puesto que permiten automatizar gran parte del análisis efectuado sobre el código fuente, ahorrando en tiempo, costo, diagnósticos rápidos y eficientes de las reglas de buen código, retroalimentación instantánea sobre la evolución de los atributos internos del sistema. Existe gran variedad de opciones disponibles en el mercado, con diversos propósitos y siendo clasificadas por el tipo de lenguaje de programación o tecnología que se utiliza en el proyecto software.

En lo referente a las herramientas usadas para la gestión de la deuda técnicas del actual experimento, cabe resaltar que estas fueron seleccionadas para entornos de aplicaciones desarrolladas en lenguaje de programación Java.

7.3.1 SonarQube

La herramienta SonarQube es un software libre que actúa como un servidor (servicio web) que integra otras herramientas especializadas en análisis estático de código fuente para obtener métricas que pueden ayudar a mejorar la calidad del código de un software. SonarQube es capaz de analizar código fuente de los lenguajes de programación más populares y tiene una arquitectura extensible a la cual se puede aumentar su funcionalidad por medio de plugins. Normalmente SonarQube hace parte de un flujo controlado de Integración Continua, apoyándose en un servidor de automatización de compilación y

pruebas del código como Jenkins, y nos proporciona una visión integral de la DT de un software, actuando de tablero de comando publicando distintos indicadores como:

- Porcentaje de cobertura. Muestra que porcentaje del código fuente cubierto con pruebas unitarias.
- Porcentaje de código duplicado. El porcentaje de código fuente duplicado dentro del mismo proyecto.
- Defectos y vulnerabilidades. La cantidad de errores y vulnerabilidades de seguridad detectados.
- Cambios en últimos 30 días. Permite visualizar diferencias en los indicadores con una ventana de 30 días.
- Línea de tiempo. Muestra la evolución de cada indicador con un gráfico para una interpretación.

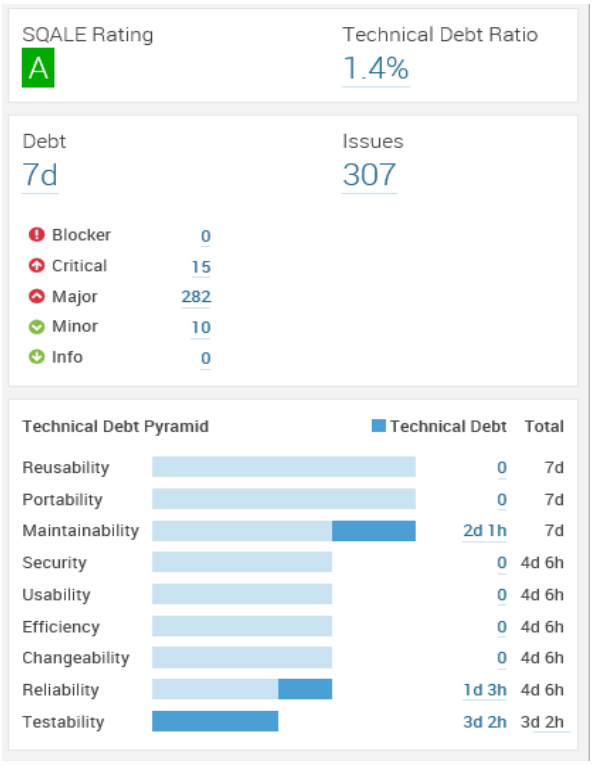


Figura 10. Tablero métricas SQALE

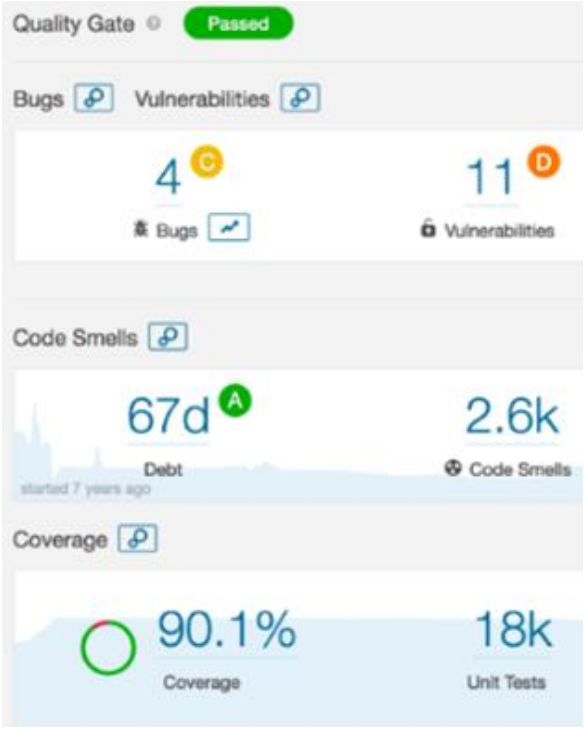


Figura 11. Tablero con las Métricas principales del proyecto

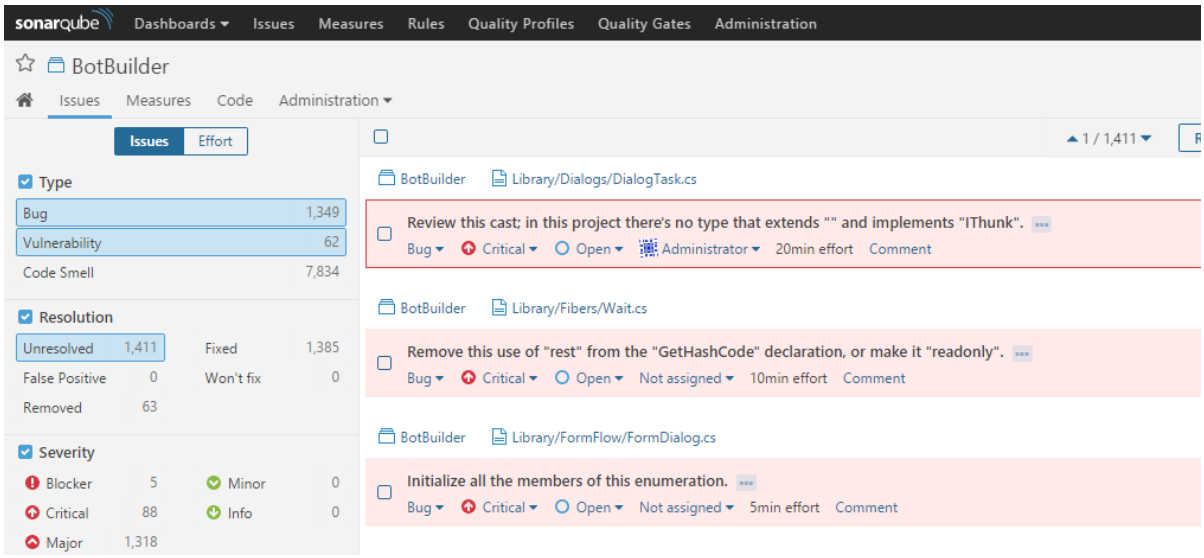


Figura 12. Tablero de problemas concretos encontrados en las clases.

7.3.2 Jenkins

Se trata de una herramienta muy poderosa con la que podremos realizar pruebas de una aplicación para así detectar fallos cuanto antes. Es un servidor de automatización *open source* escrito en Java, encargado de la automatización de procesos de desarrollo de software e integración continua, facilitando ciertos aspectos de la entrega continua. Se entiende por integración la compilación y ejecución de pruebas de todo un proyecto. El proceso se lleva a cabo normalmente cada cierto tiempo y su función es la descarga de las fuentes desde el control de versiones, su posterior compilación, la ejecución de pruebas y generar informes. Admite herramientas de control de versiones como CVS, Subversion, Git, Mercurial, Perforce y Clearcase y puede ejecutar proyectos basados en Apache Ant y Apache Maven, así como secuencias de comandos de consola y programas por lotes de Windows.

7.3.3 Herramientas de extensibilidad (Plugins)

Los plugins de extensibilidad proveen a las herramientas de servicios web tercerizar ciertos aspectos del análisis de código, manteniéndose en constante actualización y mejora debido al trabajo que desarrollan de manera independiente.

- **JaCoCo.** Permite conocer el porcentaje de código fuente que tiene pruebas unitarias asociadas, calculando la métrica de cubrimiento de código. Con esta métrica podemos determinar qué tan propenso es el sistema a introducir defectos a medida que va evolucionando.
- **PMD *(Programming Mistake Detector).*** En un analizador de código que detecta defectos y *code smells* para varios lenguajes de programación. Opera sobre el código

fuente del sistema, chequeando si existen violaciones a reglas predefinidas de buenas prácticas, o defectos conocidos.

- **Findbugs.** Es otro analizador de código configurable con reglas, pero a diferencia de PMD que realiza su análisis sobre el código fuente, Findbugs opera sobre el bytecode, esto es, el código ya compilado, por lo que suele complementar eficazmente la detección de defectos, clasificándolos en cuatro rangos: (1) muy grave (2) grave, (3) preocupante y (4) cuidado. Esta es una pista para el desarrollador sobre su posible impacto o gravedad.

7.3.4 Herramientas integradas al entorno de desarrollo (IDE)

Las últimas versiones de los entornos de desarrollo nos ofrecen distintas herramientas para la gestión y revisiones tempranas del código:

- **JUnit.** En un framework que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera, desarrollando un esquema de pruebas unitarias, donde un método recibe un valor de entrada y se espera que retorne un valor de salida.
- **Gradle.** Es un sistema de automatización de construcción de código abierto que construye sobre los conceptos de Apache Ant y Apache Maven e introduce un lenguaje especifico del dominio (DSL) basado en Groovy para declarar la configuración de proyecto. Permite gestionar las dependencias del sistema con los componentes de software que se utilizan para su construcción, ejecución y testing. Fue diseñado para construcciones multiproyecto, las cuales pueden crecer para ser bastante grandes, y da apoyo a construcciones incrementales determinando inteligentemente qué partes del árbol de construcción están actualizadas, de modo que cualquier tarea dependiente a aquellas partes no necesitarán ser reejecutadas. Su ejecución puede dispararse a voluntad del desarrollador (manual) o en el contexto de un servidor de integración continua (automática).

7.4 Análisis de Resultados

En la Tabla 27 tenemos el resultado de la métrica 1. “Valor de la deuda técnica identificada por iteración”, que es el resultado de identificar la nueva de DT generada en una iteración. El propósito del equipo de desarrollo ya sea implementando el modelo o no, es el de mantener este valor en 0, debido a que la creación de nueva DT va generando capital e interés en la deuda, perjudicando las variables de calidad del código. El análisis de datos arroja un resultado positivo sobre el Grupo Propuesto tanto en la identificación del tiempo (en horas) de nueva DT es menor (el Grupo Base lo duplica) y del porcentaje de mejora es superior en el Grupo Propuesto.

Tabla 27. Resultados de la métrica 1. Valor de la deuda técnica identificada por iteración

1. Valor de la deuda técnica identificada por iteración		
Fase de Desarrollo	**Grupo Base**	**Grupo Propuesto**
	# horas de nueva DT	
Iteración 0	-	-
Iteración 1	14	6
Iteración 2	10	4
Iteración 3	11	3
Promedio	11,7	4,3
Porcentaje de Mejora	16,7%	27,8%

En la Tabla 28 tenemos el resultado de la métrica 2. “Remediación de deuda técnica por iteración”, la cual es el resultado del trabajo del equipo en separar los ítems de DT que se priorizaron para ser remediados en la iteración. Existen múltiples estrategias de pago de DT, muchas de estas pueden bajar notablemente el valor de la DT calculada. Entre estas estrategias se realizan sesiones de pago masivo de DT, alguna de ellas las podemos categorizar de la siguiente manera:

1. Eliminar métodos, parámetros, variables, referencias e importaciones que no se usan: Es una estrategia simple para bajar el valor de la DT, que consiste en eliminar el código sin uso o que no se está siendo referenciando en ninguna parte y, por lo tanto, no genera error ni dependencia en ninguna parte.
2. Eliminar código duplicado: También es una estrategia simple pero no más que la anterior y tiene muchas formas de ser llevado a cabo. En el caso del experimento se evidenciaron estrategias para refactorizar y unificar código, reparar o crear nuevo código para eliminar el duplicado.
3. Otras estrategias de pago masivo de DT puede ser refactorización de métodos extensos, complejidad cliclomatica, desagregar clases de código extenso o clases dios; realizar sesiones de pago masivo como el Deudatón, que consiste en negociar

con el dueño de producto para realizar pago de DT en un tiempo prolongado (1/2 día, 1 día o 2 días como mucho) y verificar luego la mejora alcanzada.

El análisis de datos arroja un resultado positivo del Grupo Propuesto en la remediación de DT con respecto al tiempo (en horas) y el porcentaje de mejora sobre el Grupo Base.

Tabla 28. Resultados de la métrica 2. Remediación de deuda técnica por iteración

2. Remediación de deuda técnica por iteración		
Fase de Desarrollo	**Grupo Base**	**Grupo Propuesto**
	# de horas remediadas	
Iteración 0	-	-
Iteración 1	0	43,2
Iteración 2	14,4	26,4
Iteración 3	21,6	38,4
Promedio	12,0	36,0
Porcentaje de Mejora	5,0%	11,1%

En la Tabla 29 tenemos el resultado de la métrica 3. “Cantidad total de deuda técnica del proyecto”, que es la medición del total de la DT del proyecto. Más que tener en cuenta que proyecto tiene la mayor DT, lo que se valora es el porcentaje de mejora que se alcanza con la implementación del modelo.

Tabla 29. Resultados de la métrica 3. Cantidad total de deuda técnica del proyecto

3. Cantidad total de deuda técnica del proyecto		
Fase de Desarrollo	**Grupo Base**	**Grupo Propuesto**
	Total en días/horas	
Iteración 0	20	12,5
Iteración 1	20	10,7
Iteración 2	19,4	9,6
Iteración 3	18,5	8
Promedio	19,3	9,4
Porcentaje de Mejora	7,5%	36,0%

En la Tabla 30 tenemos el resultado de la métrica 4. “Cobertura de código fuente por iteración”, esta métrica mide el porcentaje de cobertura de nuestro código fuente para la actual iteración. Aquí se valora más el esfuerzo y tenacidad del desarrollador por implementar el suficiente número de pruebas que garantice que todas las entradas y salidas de una función se comporten del modo esperado. Para construir mejores pruebas y con resultados de alto valor, nos podemos apoyar en la documentación y los ejemplos en la literatura.

Tabla 30. Resultados de la métrica 4. Cobertura de código fuente por iteración

4. Cubrimiento de código fuente por iteración		
Fase de Desarrollo	**Grupo Base**	**Grupo Propuesto**
	Porcentaje de cubrimiento	
Iteración 0	0%	0%
Iteración 1	0%	4,3%
Iteración 2	3%	6,8%
Iteración 3	3%	8,6%
Porcentaje promedio	3%	6,6%

En la Tabla 31 tenemos el resultado de la métrica 5. "Número total de pruebas y code smells". El desarrollo guiado por pruebas (TDD) u otras metodologías, pueden ser utilizadas para que el proceso de pago de DT y cubrimiento de código sea más eficiente.

Tabla 31. Resultados de la métrica 5. Número total de pruebas y code smells

5. Número total de pruebas y code smells				
Fase de Desarrollo	**Grupo Base**		**Grupo Propuesto**	
	Cantidad de pruebas unitarias	**# de code smell**	**Cantidad de pruebas unitarias**	**# de code smell**
Iteración 0	151	85	278	60
Iteración 1	151	82	291	52
Iteración 2	157	74	304	41
Iteración 3	165	63	321	30
Porcentaje de mejora	9%	18%	15%	50%

7.5 Implementación

La etapa de implementación del modelo inicia con una planificación del equipo de trabajo para revisar los parámetros iniciales que se tenía conocimiento del código fuente de la aplicación y de cuáles eran las actividades base que el equipo debía tener en cuenta y trabajar para lograr una validación significativa del modelo. Se inicia con una pre-iteración llamada "Iteración 0", en la que se busca realizar un análisis general del sistema, entendiendo las características de los ítems de DT de código fuente con los que se cuenta, el porcentaje de tiempo que se le va a dedicar a la remediación de DT, que para el caso actual radica en un porcentaje entre el 15% o 30% del tiempo total de trabajo dedicado al pago de DT.

En la Figura 13 se visualizan las evidencias del primer reporte de métricas de DT para el grupo propuesto:

Etapa.	Iteración 0
Bugs y Vulnerabilities.	No se encuentran errores de compilación y ni vulnerabilidades en el código **(valor 0)**
Code Smells.	La DT inicia con un valor de **12 días y 5 horas**, y una cantidad de code smells de **60**
Coverage.	El porcentaje de cobertura del proyecto es 35.1%, con una cantidad de 278 pruebas unitarias

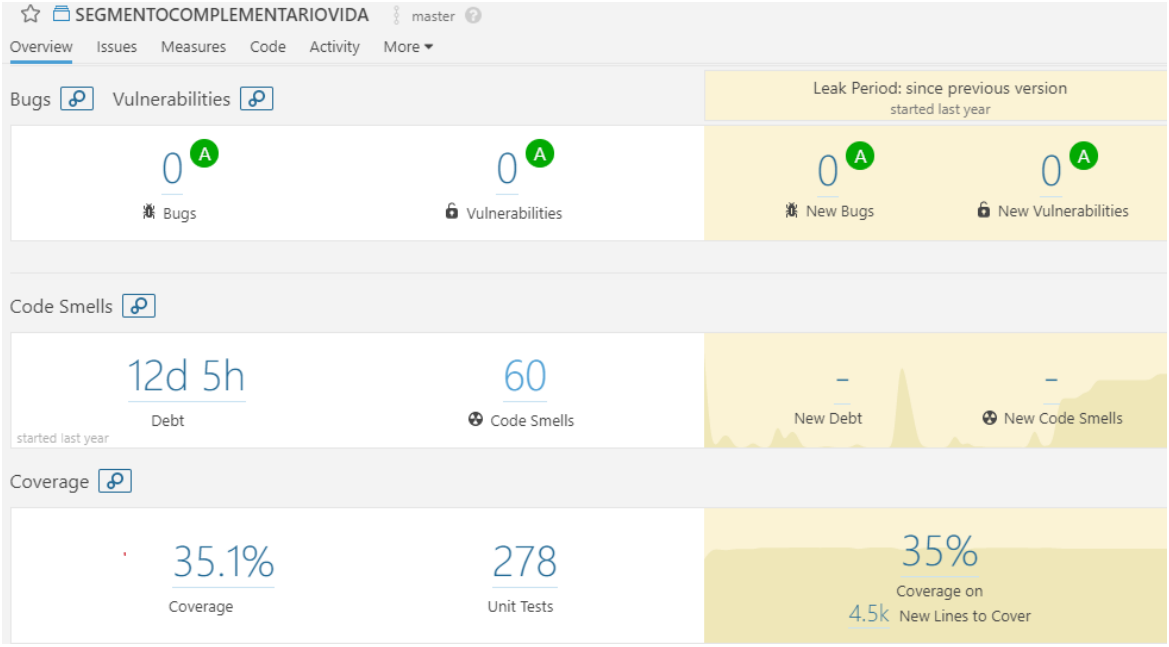


Figura 13. Tablero página principal de SonarQube de la DT en la Iteración 0

Otro aspecto importante que se tuvo en cuenta fue la negociación con el equipo en buscar de potenciar sus capacidades para desarrollar código de mayor calidad, que desde el mismo momento de su construcción cuente con criterios de entrega que incurran en la remediación y control de la DT, tales como la legibilidad, modificabilidad, sin duplicidad, bajo nivel de código desagradable, cobertura de pruebas unitarias, entre otros. Se debe aclarar que la mayoría de desarrolladores ya cuentan con experiencia previa considerable en el desarrollo del sistema, por lo que les resulta más fácil realizar los cambios en el código.

Para el inicio de la Iteración 1, se revisa el listado priorizado de ítems de DT y el código desagradable más común en el proyecto, seleccionando cuál es el más fácil de remediar y al mismo tiempo ayudar a disminuir las mediciones de tiempo de remoción de DT actuales. Se desarrollan capacitaciones sobre GDT y buenas prácticas de desarrollo. Se define como estrategia de medición y priorización para esta iteración qué ítems de DT son de bajo esfuerzo y de alto impacto, debido a que estos agregan más valor de negocio. Los ítems de alto esfuerzo con alto o bajo impacto no serán tenidos en cuenta.

En la Figura 14 se visualizan las evidencias del segundo reporte de métricas de DT para el grupo propuesto:

Etapa.	Iteración 1
Bugs y Vulnerabilities.	No se encuentran errores de compilación y ni vulnerabilidades en el código **(valor 0)**
Code Smells.	La DT disminuyo 1 día y 22 horas, posee un valor de **10 días y 7 horas**, y una cantidad de code smells de **52**
Coverage.	El porcentaje de cobertura del proyecto subió 4.3%, posee un valor de 39.4%, con una cantidad de 291 pruebas unitarias

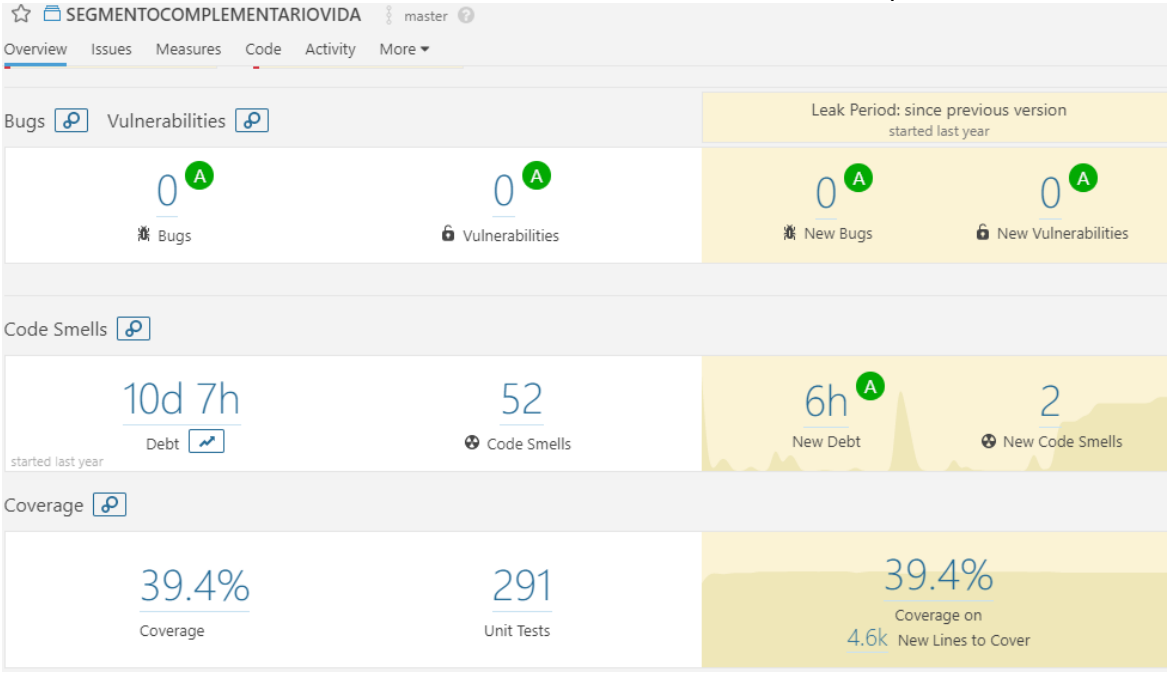


Figura 14. Tablero página principal de SonarQube de la DT en la Iteración 1

En la Iteración 2 se programaron distintas reuniones con el líder técnico, para verificar el plan de trabajo. Se tomaron decisiones con respecto a implementaciones encontradas de mal diseño o arquitectura del proyecto. Entre estas decisiones se enumeraron los siguientes consejos para que los desarrolladores implementaran de una mejor manera:

1. No usar la sentencia de impresión de log o System.out.println
2. Documentación excesiva de código puede desordenarlo; se aconseja que con un buen nombramiento de métodos o variables debe ser suficiente para un código ordenado.
3. Utilizar Inyección de Dependencias en lugar de instanciar el mismo objeto para una clase.
4. Realizar pequeños cambios sin alterar la lógica de existente, desarrollando las respectivas pruebas que garanticen un mínimo de afectaciones.
5. Realizar pruebas funcionales (incluyendo la capa de presentación), pidiendo incluso ayuda al cliente o analista de pruebas de la aplicación para garantizar que la funcionalidad sigue siendo la misma.

6. Recordar que el pago de la DT no debe ser vista como resolución de defectos del sistema, en casos que se quiera remediar la DT, al mismo tiempo que resolver un defecto identificado, se deben tomar mayor tiempo y medidas especiales para garantizar un buen funcionamiento.

En la Figura 15 se visualizan las evidencias del tercer reporte de métricas de DT para el grupo propuesto:

Etapa.	Iteración 2
Bugs y Vulnerabilities.	No se encuentran errores de compilación y ni vulnerabilidades en el código **(valor 0)**
Code Smells.	La DT disminuyo 1 día y 1 hora, posee un valor de **9 días y 5 horas**, y una cantidad de code smells de **41**
Coverage.	El porcentaje de cobertura del proyecto subió 6.8%, posee un valor de 46.2%, con una cantidad de 304 pruebas unitarias

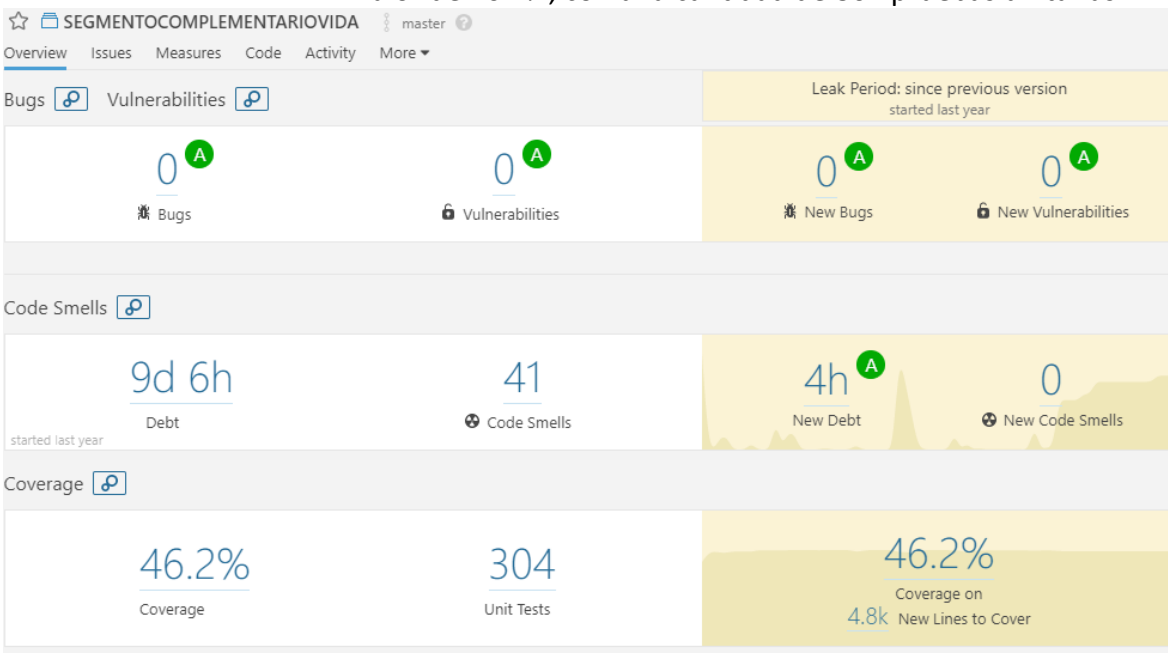


Figura 15. Tablero página principal de SonarQube de la DT en la Iteración 2

En la Iteración 3 se realizaron actividades relacionadas con el proceso de refactorización, verificando el código fuente construido en las dos iteraciones anteriores y mejorándolo en aquellos puntos donde aún existía nueva DT identificada. En la medida de lo posible se debe subir al repositorio el código con algún tipo de revisión (simple o profunda), ya sea que se utilice como estrategia herramientas de análisis de código y revisiones par con los compañeros de equipo, lideres técnicos o no tenga definida ninguna.

En la Figura 16 se visualizan las evidencias del tercer reporte de métricas de DT para el grupo propuesto, y el resultado total de la suma de la implementación del modelo en las tres iteraciones estudiadas:

Etapa.	Iteración 3
Bugs y Vulnerabilities.	No se encuentran errores de compilación y ni vulnerabilidades en el código **(valor 0)**
Code Smells.	La DT disminuyo 1 día y 5 horas, posee un valor de **8 días**, y una cantidad de code smells de **30**
Coverage.	El porcentaje de cobertura del proyecto subió 8.6%, posee un valor de 54.8%, con una cantidad de 321 pruebas unitarias

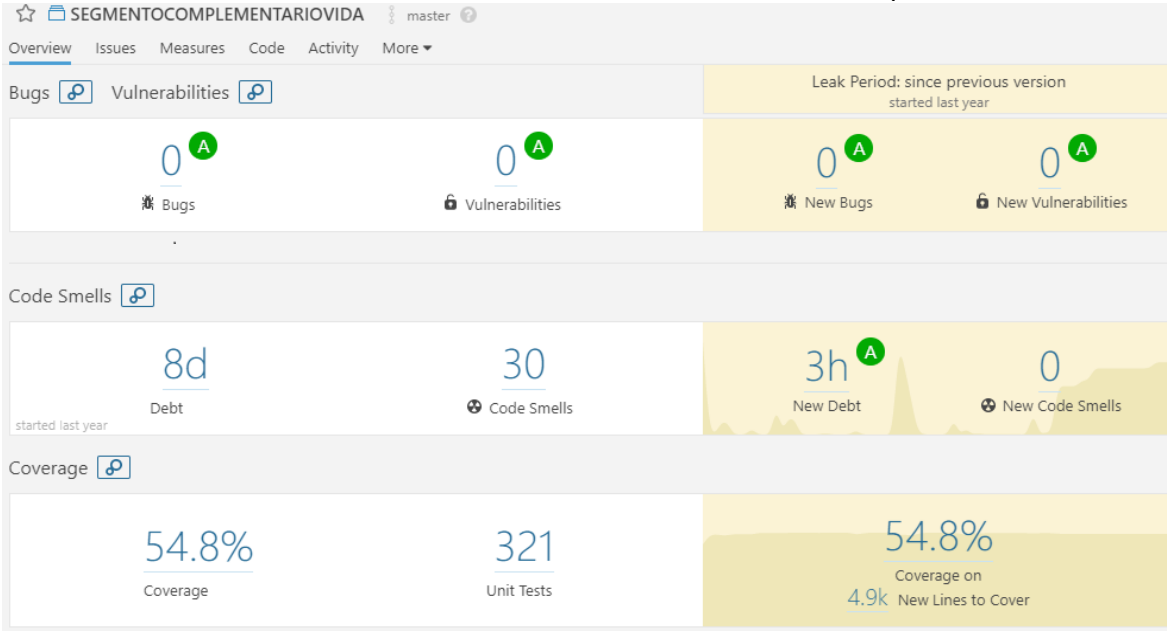


Figura 16. Tablero página principal de SonarQube de la DT en la Iteración 3

8.

Capítulo VIII

Conclusión

La Deuda Técnica es un tema que ha tomado gran interés y avances significativos para su gestión en la ingeniería de software, muchas veces debido al enfoque excesivo en la entrega continua de funcionalidades para el cliente que llevan a una situación donde la calidad del código fuente construido carece de la mayoría de criterios de calidad aceptados por los estándares actuales.

La principal contribución del proyecto de investigación estuvo en la definición de las siete fases del modelo para la Gestión de la Deuda Técnica, que, en conjunto, describen un ciclo de vida iterativo y de fácil adaptación, enfocado a las buenas prácticas de desarrollo, orientado al conocimiento de los objetivos de negocio, trabajo en equipo, la sostenibilidad y mantenibilidad de las aplicaciones. Es un aporte para aquellos desarrolladores, líderes técnicos y líderes de proyecto que deseen conocer de primera mano cómo se encuentra la salud del código de su proyecto, de realizar seguimientos, retroalimentaciones, toma de decisiones para la mejora técnica, escalabilidad de las aplicaciones y mantener un proceso de mejora continua de las aplicaciones y las personas.

Otro aporte interesante es el análisis de los distintos enfoques y propuestas de investigación identificados en la revisión de literatura, la descripción de los diferentes tipos de deuda técnica que se están trabajando los autores y cuál es su aporte en la industria, que trabajos y autores son referentes en todo el tema de deuda técnica, principalmente en la de código, diseño, arquitectura y pruebas.

Si bien cuidar y mantener un código con atributos de buena calidad, controlado a través de herramientas de monitoreo, es solo una parte importante de muchos otros atributos de éxito de un proyecto; es prenda de garantía al cliente a quien se le hace entrega de productos de software mejor construidos; puede ser fundamental en el crecimiento, análisis de mejores soluciones y adquisición de conocimiento de los desarrolladores para futuras aplicaciones. Dados los resultados obtenidos en la implementación del modelo propuesto, éste puede servir como base para futuras aplicaciones y extensiones para mejorar la calidad de otros sistemas y el proceso de desarrollo en general.

8.1 Producto de la Validación

Resumen análisis implementación. La implementación del modelo se desarrolló sobre dos proyectos de software en etapa de producción, con equipos de desarrollo divididos en dos Grupos (Base y Propuesto), en un caso de estudio que contó con la participación de doce desarrollares, que a su vez se conformaban por distintos perfiles de experiencia, desde desarrolladores en el nivel Junior, hasta el nivel Senior. Esta experiencia permite recoger importantes conclusiones sobre la utilización del modelo para otros proyectos de gestión de la DT:

1. La estructura del modelo y los conceptos en los que se basa son lógicos y fáciles de entender para los profesionales.
2. El modelo tiene una buena relación con el área de experiencia de los profesionales y contiene referencias a temas de buenas prácticas de codificación que utilizan los desarrolladores en su trabajo diario.
3. Los participantes en el proceso de estimación de la DT deben tener una experiencia considerable con los sistemas que van a medir, de lo contrario los resultados pueden ser mucho menos confiables.
4. Siguiendo un proceso definido, claro y estructurado de la GDT, permitió tener una mejor perspectiva sobre el estudio del sistema, la detección de problemas, ponderarlos, proponer soluciones e implementarlas de manera efectiva y planificada.
5. En proyectos con menor grado de formalidad en su metodología de desarrollo y sin llevar a cabo un proceso definido de GDT, se observa que, si es posible mantener una buena salud del código, pero de modo más lento y menos confiable, sin embargo, se pueden lograr buenos resultados mientras se mantenga una cultura de preocupación por la calidad técnica del código.
6. El modelo desarrollado, actualmente, solo cubre la parte que tiene que ver con la DT de código fuente. Se puede hacer una mejor discusión de viabilidad cuando se recopilarán datos de otras capas.

Resumen resultados métricas. Para la validación y obtención de las métricas del experimento de investigación, se tuvieron en cuenta seis indicadores referentes al fenómeno de la DT, las cuales representan en conjunto el estado de calidad del código de la aplicación, su impacto e importancia están descritas en la Tabla 32:

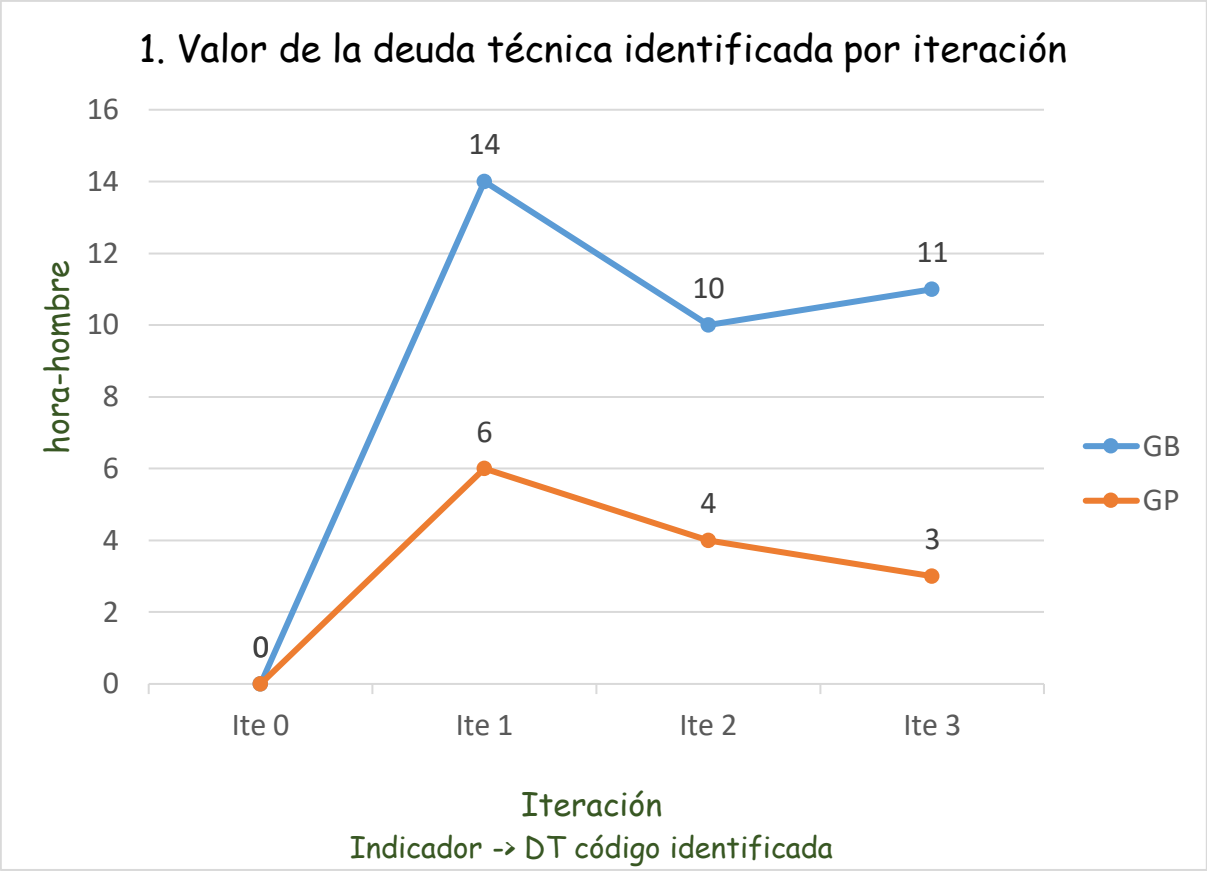
Tabla 32. Listado de indiciadores calculados con su unidad de medida y valor obtenido

Valor obtenido	Unidad de medida	Indicador calculado
Deuda de código	Hora-Persona	DT código identificada por iteración
		DT código pagada por iteración
		DT código existente en la aplicación por iteración
Cubrimiento de pruebas	Porcentaje	Incremento del cubrimiento de código por pruebas unitarias por iteración

Número de pruebas	Cantidad	Cantidad de pruebas unitarias identificadas en la aplicación por iteración
Número de code smells	Cantidad	Cantidad de deficiencias de código identificadas en la aplicación por iteración

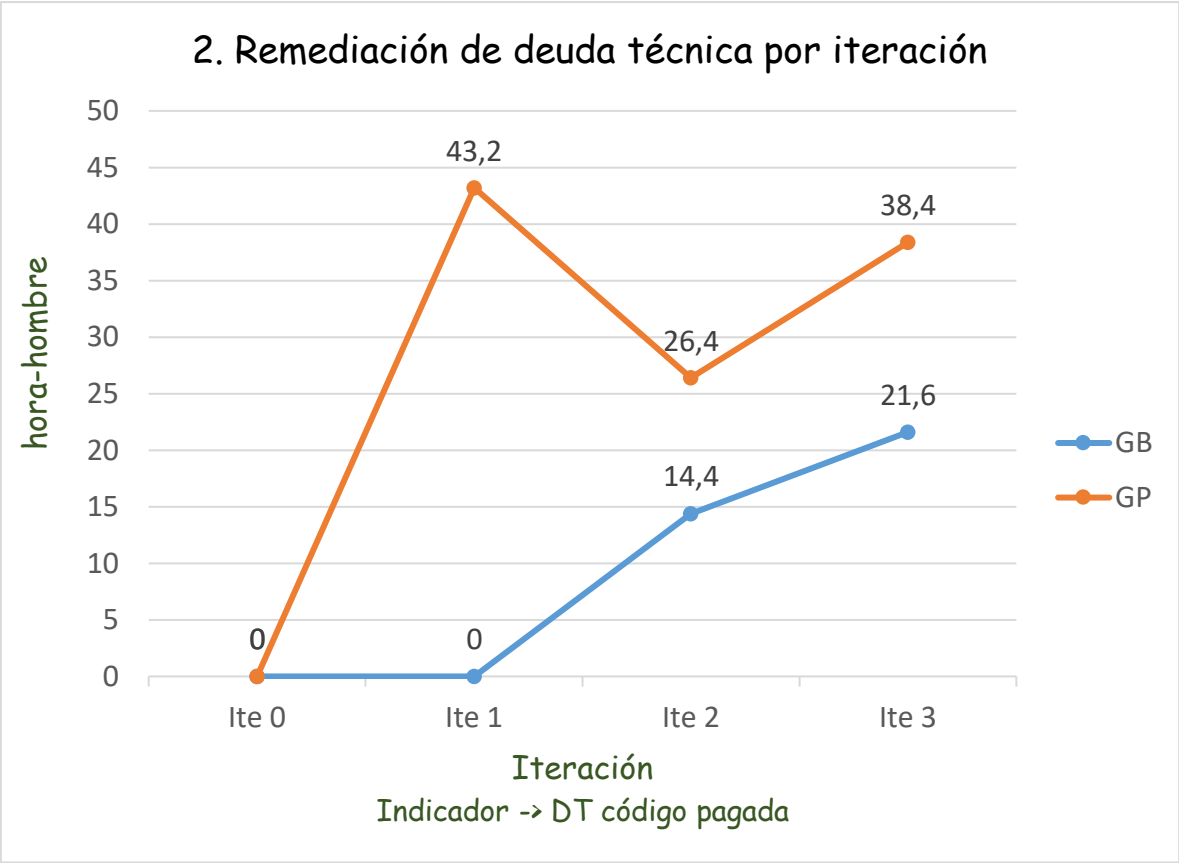
En la Figura 17 se detallan los resultados de la Métrica 1. *"Valor de la deuda técnica identificada por iteración"* sobre el Grupo Base (GB) y Grupo Propuesto (GP)

Figura 17. Gráfica de la métrica #1 sobre los dos grupos observados



En la Figura 18 se detallan los resultados de la Métrica 2. *"Remediación de deuda técnica por iteración"* sobre el Grupo Base (GB) y Grupo Propuesto (GP)

Figura 18. Gráfica de la métrica #2 sobre los dos grupos observados



En la Figura 19 y 20 se detallan los resultados de la Métrica 3. *"Cantidad total de deuda técnica del proyecto"* sobre el Grupo Base (GB) y Grupo Propuesto (GP)

Figura 19. Gráfica de la métrica #3 sobre el Grupo Base

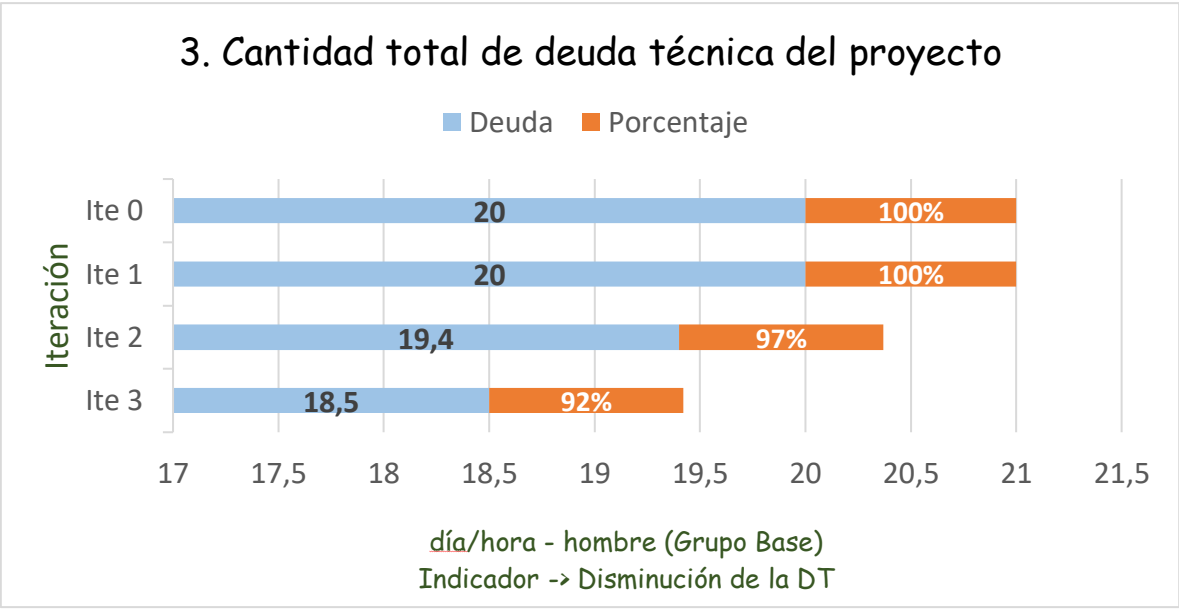
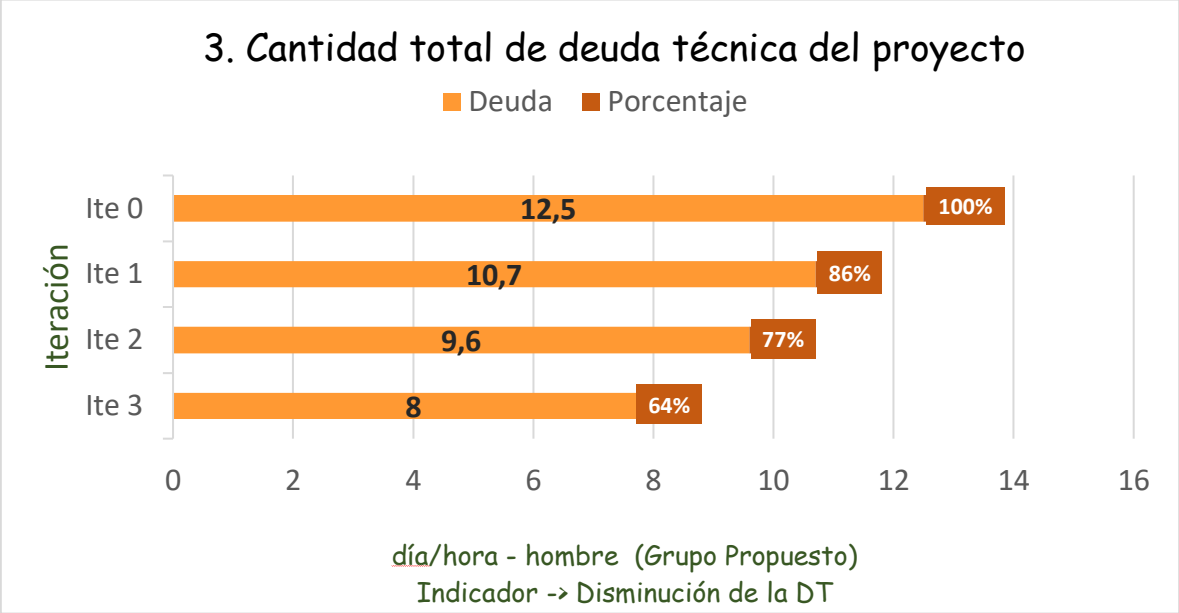
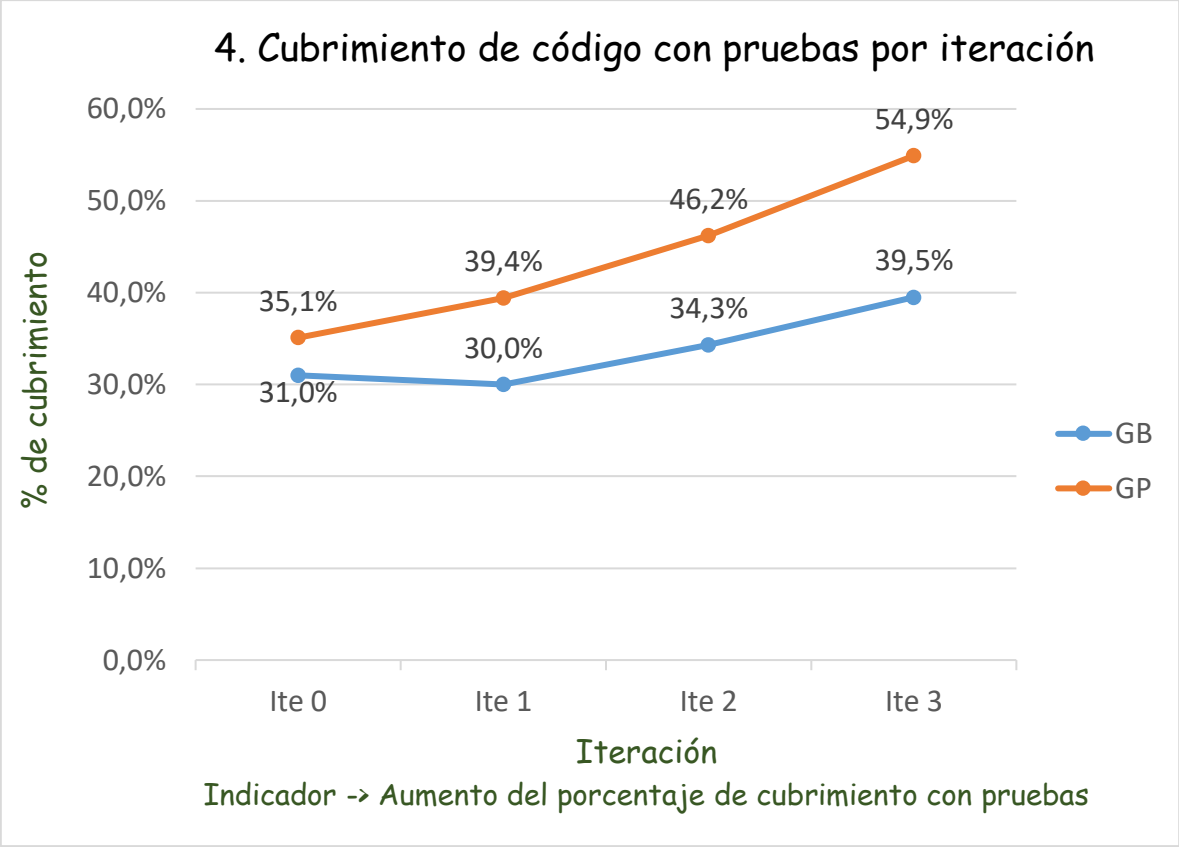


Figura 20. Gráfica de la métrica #3 sobre el Grupo Propuesto



En la Figura 21 se detallan los resultados de la Métrica 4. *"Cubrimiento de código con pruebas por iteración"* sobre el Grupo Base (GB) y Grupo Propuesto (GP)

Figura 21. Gráfica de la métrica #4 sobre los dos grupos observados



En la Figura 22 y 23 se detallan los resultados de la Métrica 5. *"Cubrimiento de código con pruebas por iteración"* con respecto a los code smells

Figura 24. Gráfica de code smells sobre la métrica #5 en el Grupo Base

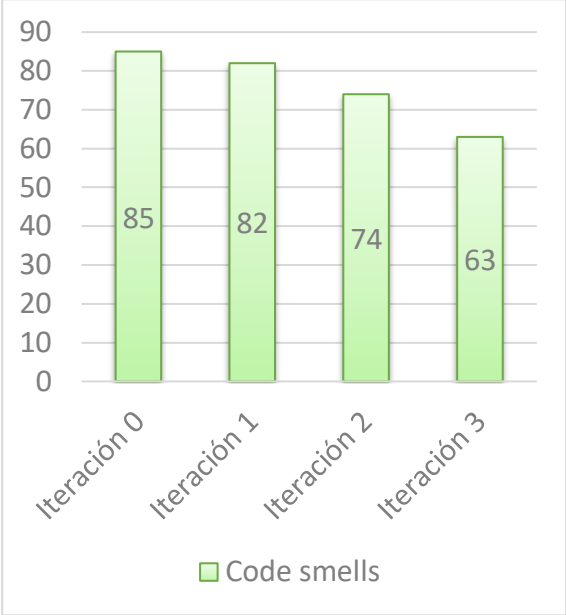
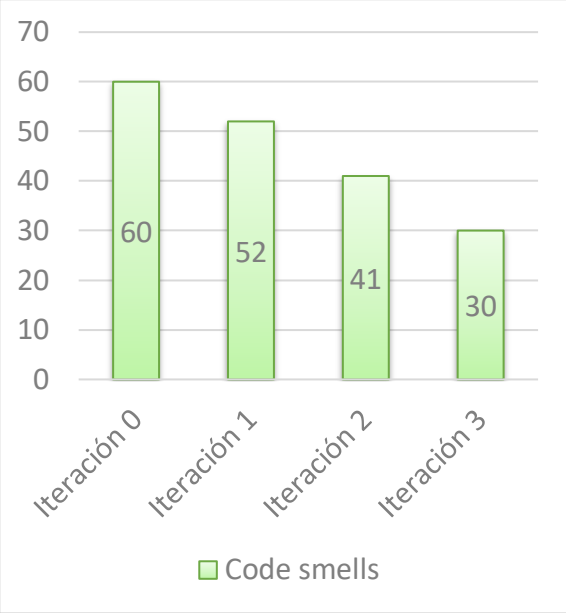


Figura 25. Gráfica de code smells sobre la métrica #5 en el Grupo Propuesto



En la Figura 24 y 25 se detallan los resultados de la Métrica 5. *"Cubrimiento de código con pruebas por iteración"* con respecto a la cantidad de pruebas unitarias

Figura 24. Gráfica de cantidad de pruebas unitarias de la métrica #5 en el Grupo Base

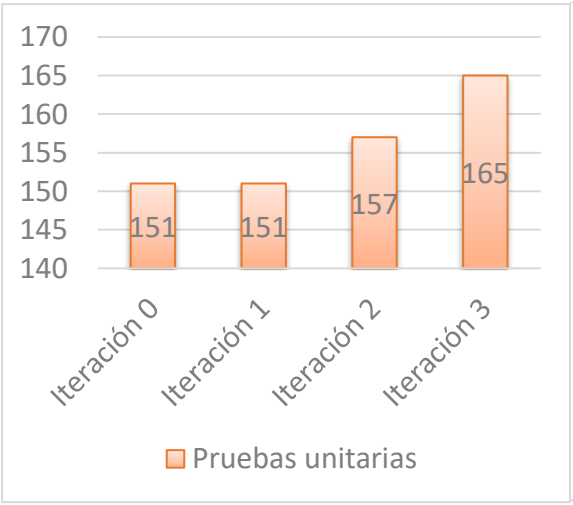
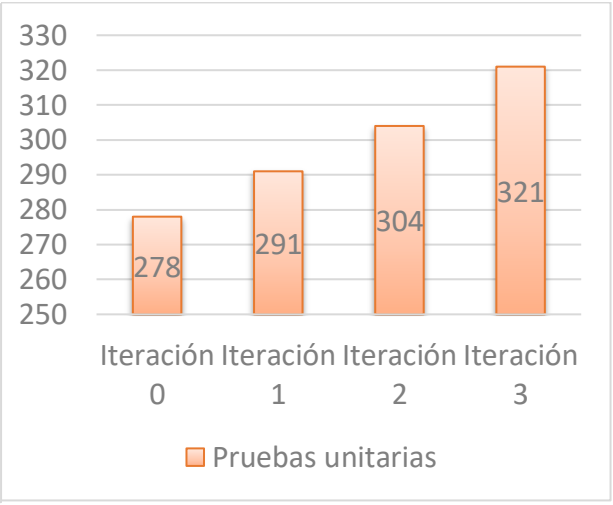


Figura 25. Gráfica de cantidad de pruebas unitarias de la métrica #5 en el Grupo Propuesto



En conclusión, la implementación del modelo se beneficia significativamente en los casos donde los proyectos de software mantienen una documentación actualizada, requerimientos claros, atributos de calidad definidos, que mantiene un ritmo de desarrollo o mantenimiento de funcionalidades constante y contar con desarrolladores comprometidos por mantener un buen diseño, arquitectura y cobertura de pruebas, garantizará en mayor medida el éxito de una GDT más eficiente, ya que estos insumos facilitan el análisis, la detección de problemas y la implementación de mejores soluciones.

8.2 Limitaciones

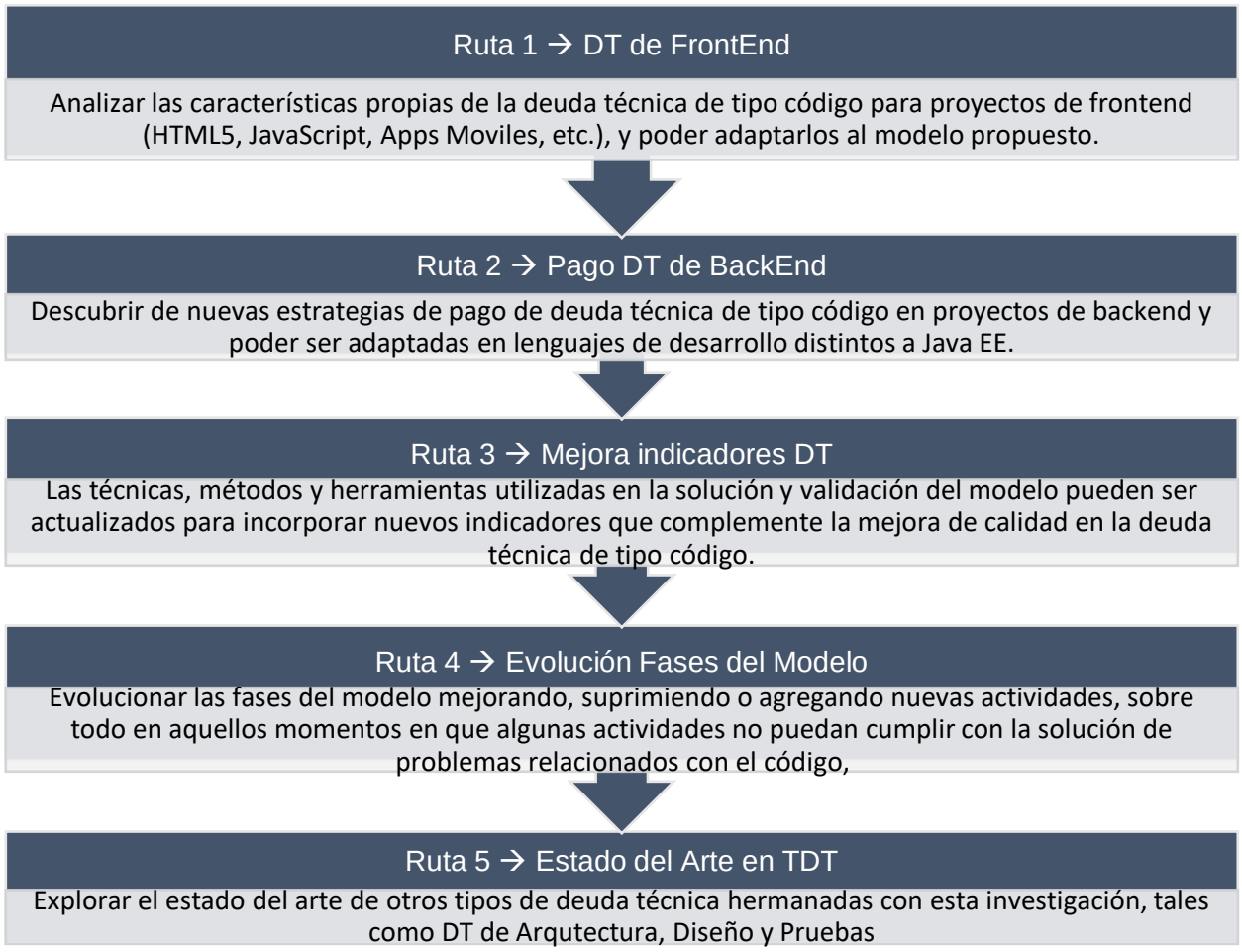
A partir de la implementación del modelo propuesto, se detectan algunas limitaciones que es bueno tener en cuenta en otros casos de aplicación.

Escenarios de implementación del modelo:

- La revisión del estado del arte de la DT y definición del modelo para esta investigación, está en enmarcado en un paradigma de programación específico de Programación Orientada a Objetos (POO). Aunque ciertas actividades pueden ser concebidas de manera genérica, los ejemplo y documentación de apoyo para los procesos de capacitación en DT, están orientados igualmente a equipos con conocimiento en el paradigma de POO.
- Conocimiento previo a escenarios de tratamiento de la DT: En muchos casos los escenarios donde se requiere realizar un proceso de remediación de la DT, solo cuentan con descripciones básicas en la documentación o con simples sugerencias dadas por una herramienta de revisión de código estático, por lo que un desarrollador sin conocimiento en el sistema o de buenas prácticas de codificación, puede verse con varias dificultades y baja productividad al inicio, mientras nivela su situación.
- Desconocimiento de la DT estimada: Se sugiere que en la fase de medición se reúnan todos los integrantes del equipo que tengan conocimiento de lo se quiere estimar, debido a que, si se delega a integrantes sin experiencia previa en el análisis del problema, puede llegar extenderse considerablemente el pago de la DT.
- Garantizar 100% de cobertura de pruebas en algunos aspectos del código: Se evidencia que en un código que no cuenta inicialmente con cobertura, acoplado y altamente complejo, se hace difícil su total desacoplamiento y garantizar que esté totalmente cubierto de pruebas es demasiado costoso en tiempo y dinero. Por tal razón, en estos casos se debe tener en cuenta sólo cubrir el código más crítico en lo referente a la lógica de negocio.

8.3 Trabajo Futuro

Ningún ciclo de desarrollo de un producto software es inmune a problemas de deuda técnica, debido que este fenómeno ha escalado a otros aspectos separados al código fuente, como son los Procesos, Requisitos, Documentación, Usabilidad, Pruebas, entre otros. Cuando se desea extender un proceso de gestión de deuda técnica con características adicionales al código o su arquitectura, se recomiendas que sean lo más simple posible, por diversas razones como, no afectar significativamente la capacidad de producción del equipo de desarrollo o TI, cuyo objetivo principal debe ser siempre es la generación de valor mediante la provisión de funcionalidades de software, y por otro lado, no es deseable que el propio proceso de gestión de deuda técnica acumule DT (dejando actividades sin realizar). Cada experiencia de gestión de deuda técnica tiene sus particularidades, por lo que se requiere de más implementaciones en distintos equipos de trabajo para realizar validaciones con mayor variedad, a fin de definir un proceso consistente que garantice una mejora de calidad tanto a nivel de código, por tal motivo, se realiza un resumen de Roadmap (hoja de ruta) como continuación para futuras investigaciones:



Referencias

Alves, N. S. R., Mendes, T. S., De Mendonça, M. G., Spinola, R. O., Shull, F., & Seaman, C. (2016). Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, *70*, 100–121. https://doi.org/10.1016/j.infsof.2015.10.008

Alves, N. S. R., Ribeiro, L. F., Caires, V., Mendes, T. S., & Spínola, R. O. (2014). Towards an ontology of terms on technical debt. *Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014*, 1–7. https://doi.org/10.1109/MTD.2014.9

Alzaghoul, E., & Bahsoon, R. (2014). Evaluating technical debt in cloud-based architectures using real options. *Proceedings of the Australian Software Engineering Conference, ASWEC*, 1–10. https://doi.org/10.1109/ASWEC.2014.27

Ampatzoglou, A., Michailidis, A., Sarikyriakidis, C., Ampatzoglou, A., Chatzigeorgiou, A., & Avgeriou, P. (2018). A framework for managing interest in technical debt: An industrial validation. *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1145/3194164.3194175

Ampatzoglou, Areti, Ampatzoglou, A., Chatzigeorgiou, A., & Avgeriou, P. (2015). The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology*, *64*, 52–73. https://doi.org/10.1016/j.infsof.2015.04.001

Behutiye, W. N., Rodríguez, P., Oivo, M., & Tosun, A. (2017). Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, *82*, 139–158. https://doi.org/10.1016/j.infsof.2016.10.004

Coq, T., & Rosen, J. P. (2011). The SQALE quality and analysis models for assessing the quality of Ada source code. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *6652 LNCS*, 61–74. https://doi.org/10.1007/978-3-642-21338-0_5

Curtis, B., Sappidi, J., & Szynkarski, A. (2012a). Estimating the principal of an application's technical debt. *IEEE Software*, *29*(6), 34–42. https://doi.org/10.1109/MS.2012.156

Curtis, B., Sappidi, J., & Szynkarski, A. (2012b). presentatie - Estimating the Principal of Technical Debt. *IEEE Software*, *29*(6), 34–42. https://doi.org/10.1109/MS.2012.156

Curtis, B., Szynkarski, A., Worth, F., & York, N. (2012). Estimating the Size, Cost, and Types of Technical Debt, 812. https://doi.org/10.1109/MTD.2012.6226000

dos Santos, P. S. M., Varella, A., Dantas, C. R., & Borges, D. B. (2013). Visualizing and managing technical debt in agile development: An experience report. *Lecture Notes*

in Business Information Processing, *149*, 121–134. https://doi.org/10.1007/978-3-642-38314-4_9

Eisenberg, R. J. (2012). A threshold based approach to technical debt. *ACM SIGSOFT Software Engineering Notes*, *37*(2), 1. https://doi.org/10.1145/2108144.2108151

Falessi, D., Shaw, M. A., Shull, F., Mullen, K., & Keymind, M. S. (2013). Practical considerations, challenges, and requirements of tool-support for managing technical debt. *2013 4th International Workshop on Managing Technical Debt, MTD 2013 - Proceedings*, (Ml), 16–19. https://doi.org/10.1109/MTD.2013.6608673

Fayad, M. E. y Schmidt, D. C. (1997). Object-Oriented Application Frameworks. Communications of the ACM, 40(10):32–38.

Fontana, F. A., Ferme, V., & Spinelli, S. (2012). Investigating the impact of code smells debt on quality code evaluation. *2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings*, 15–22. https://doi.org/10.1109/MTD.2012.6225993

Greening, D. R. (2013). Release duration and Enterprise agility. *Proceedings of the Annual Hawaii International Conference on System Sciences*, 4835–4841. https://doi.org/10.1109/HICSS.2013.463

Griffith, I. D. (2015). Design Pattern Decay – a Study of Design Pattern Grime and Its Impact on Quality and Technical Debt.

Griffith, I., Reimanis, D., Izurieta, C., Codabux, Z., Deo, A., & Williams, B. (2014). The correspondence between software quality models and technical debt estimation approaches. *Proceedings - 2014 6th IEEE International Workshop on Managing Technical Debt, MTD 2014*, 19–26. https://doi.org/10.1109/MTD.2014.13

Guo, Y., & Seaman, C. (2011). A portfolio approach to technical debt management. *Proceeding of the 2nd Working on Managing Technical Debt - MTD '11*, 31. https://doi.org/10.1145/1985362.1985370

Guo, Y., Seaman, C., & da Silva, F. Q. B. (2016). Costs and obstacles encountered in technical debt management – A case study. *Journal of Systems and Software*, *120*, 156–169. https://doi.org/http://dx.doi.org/10.1016/j.jss.2016.07.008

Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F. Q. B., … Siebra, C. (2011). Tracking technical debt — An exploratory case study. *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 528–531. https://doi.org/10.1109/ICSM.2011.6080824

Hamdan, S., & Alramouni, S. (2015). A Quality Framework for Software Continuous Integration. *Procedia Manufacturing*, *3*(Ahfe 2015), 2019–2025. https://doi.org/10.1016/j.promfg.2015.07.249

Holvitie, J., & Leppänen, V. (2013). DebtFlag: Technical debt management with a development environment integrated tool. *2013 4th International Workshop on*

Managing Technical Debt, MTD 2013 - Proceedings, 20–27. https://doi.org/10.1109/MTD.2013.6608674

Holvitie, J., Licorish, S. A., Spínola, R. O., Hyrynsalmi, S., MacDonell, S. G., Mendes, T. S., … Leppänen, V. (2018). Technical debt and agile software development practices and processes: An industry practitioner survey. *Information and Software Technology*, *96*, 141–160. https://doi.org/10.1016/j.infsof.2017.11.015

Izurieta, C., Vetrò, A., Zazworka, N., Cai, Y., Seaman, C., & Shull, F. (2012). Organizing the technical debt landscape. *2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings*, 23–26. https://doi.org/10.1109/MTD.2012.6225995

Kim, M., Zimmermann, T., & Nagappan, N. (2014). An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, *40*(7), 633–649. https://doi.org/10.1109/TSE.2014.2318734

Kosti, M. V., Ampatzoglou, A., Chatzigeorgiou, A., Pallas, G., Stamelos, I., & Angelis, L. (2017). Technical debt principal assessment through structural metrics. *Proceedings - 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017*, *1*, 329–333. https://doi.org/10.1109/SEAA.2017.59

Ktata, O., & Lévesque, G. (2010). Designing and implementing a measurement program for Scrum teams. *Proceedings of the Third C* Conference on Computer Science and Software Engineering - C3S2E '10*, 101–107. https://doi.org/10.1145/1822327.1822341

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, *101*, 193–220. https://doi.org/10.1016/j.jss.2014.12.027

MacCormack, A., & Sturtevant, D. J. (2016). Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software*, *120*, 170–182. https://doi.org/10.1016/j.jss.2016.06.007

Marinescu, R. (2012). Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development*, *56*(5), 9:1-9:13. https://doi.org/10.1147/JRD.2012.2204512

Martini, A., Sikander, E., & Madlani, N. (2018). A semi-automated framework for the identification and estimation of Architectural Technical Debt: A comparative case-study on the modularization of a software component. *Information and Software Technology*, *93*, 264–279. https://doi.org/10.1016/j.infsof.2017.08.005

Melin, T. (2016). Implementation and evaluation of a continuous code inspection platform.

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, *30*(2), 126–139. https://doi.org/10.1109/TSE.2004.1265817

Mo, R., Garcia, J., Cai, Y., & Medvidovic, N. (2013). Mapping architectural decay instances to dependency models. *2013 4th International Workshop on Managing Technical Debt (MTD)*, 39–46. https://doi.org/10.1109/MTD.2013.6608677

Nord, R. L., Ozkaya, I., Kruchten, P., & Gonzalez-Rojas, M. (2012). In search of a metric for managing architectural technical debt. *Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012*, 91–100. https://doi.org/10.1109/WICSA-ECSA.212.17

Ramasubbu, N., & Kemerer, C. F. (2014). Managing technical debt in enterprise software packages. *IEEE Transactions on Software Engineering*, *40*(8), 758–772. https://doi.org/10.1109/TSE.2014.2327027

Ramasubbu, N., & Kemerer, C. F. (2017). Integrating Technical Debt Management and Software Quality Management Processes: A Normative Framework and Field Tests. *IEEE Transactions on Software Engineering*, 1–17. https://doi.org/10.1109/TSE.2017.2774832

Schmid, K. (2013). On the limits of the technical debt metaphor some guidance on going beyond. *2013 4th International Workshop on Managing Technical Debt, MTD 2013 - Proceedings*, 63–66. https://doi.org/10.1109/MTD.2013.6608681

Seaman, C., Guo, Y., Zazworka, N., Shull, F., Izurieta, C., Cai, Y., & Vetrò, A. (2012). Using technical debt data in decision making: Potential decision approaches. *2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings*, 45–48. https://doi.org/10.1109/MTD.2012.6225999

Szyperski, C. (2000). Component versus Objects. ObjectiveView, 1(5):8–16.

Szyperski, C. (1998). Component Software. Beyond Object-Oriented Programming. Addison-Wesley

Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, *86*(6), 1498–1516. https://doi.org/10.1016/j.jss.2012.12.052

Yan, M., Xia, X., Shihab, E., Lo, D., Yin, J., & Yang, X. (2018). Automating Change-level Self-admitted Technical Debt Determination. *IEEE Transactions on Software Engineering*, *5589*(c), 1–18. https://doi.org/10.1109/TSE.2018.2831232

Yli-Huumo, J., Maglyas, A., & Smolander, K. (2016). How do software development teams manage technical debt? – An empirical study. *Journal of Systems and Software*, *120*, 195–218. https://doi.org/10.1016/j.jss.2016.05.018

Zazworka, N., Vetro', A., Izurieta, C., Wong, S., Cai, Y., Seaman, C., & Shull, F. (2014). Comparing four approaches for technical debt identification. *Software Quality Journal*, *22*(3), 403–426. https://doi.org/10.1007/s11219-013-9200-8

Zhang, Y., Huang, G., Liu, X., Zhang, W., Mei, H., & Yang, S. (2012). Refactoring android Java code for on-demand computation offloading. *ACM SIGPLAN Notices*, *47*(10), 233. https://doi.org/10.1145/2398857.2384634