UNIVERSIDAD DE COSTA RICA

SISTEMA DE ESTUDIOS DE POSGRADO

EVALUACIÓN DE UNA HERRAMIENTA DE DETECCIÓN PARA LA IDENTIFICACIÓN DE BAD SMELLS ARQUITECTÓNICOS EN LOS SISTEMAS WEB DE LA OFICINA DE BECAS Y ATENCIÓN SOCIOECONÓMICA

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

JOSÉ DAVID FALLAS DELGADO

Ciudad Universitaria Rodrigo Facio, Costa Rica

2024

DEDICATORIA

A mi esposa, mi hijo y mi hija por ser la inspiración para cada decisión de vida que he tomado y por apoyarme y tenerme paciencia durante todo el proceso.

AGRADECIMIENTO

A mis compañeros de Posgrado que fueron un gran apoyo en todos los cursos y trabajos de investigación.

Al Dr. Gustavo López Herrera por guiarme y aconsejarme durante la propuesta y ejecución de este trabajo de investigación. A la Dr. Alexia Pacheco Hernández y al M.Sc. Julio Guzmán Benavides por el tiempo dedicado y la colaboración brindada.

Al personal del Área de Informática de la Oficina de Becas y Atención Socioeconómica y al del Área de Desarrollo de Software del Centro de Informática, ambos de la Universidad de Costa Rica, por participar de la investigación y compartir su conocimiento y experiencia.

Este Trabajo Final de Investigación Aplicada fue aceptado por la Comisión del Programa de Estudios de Posgrado en Computación e Informática de la Universidad de Costa Rica, como requisito parcial para optar por el grado y título de Maestría Profesional en Computación e Informática.

ADRIAN LARA PETITDEMANGE (FIRMA)
PERSONA FISICA, CPF-01-1225-0679.
Fecha declarada: 14/10/2024 08:58:51 PM
Esta es una representación gráfica únicamente, verifique la validez de la firma.

Dr. Adrián Lara Petitdemange
Representante de la Decana
Sistema de Estudios de Posgrado

GUSTAVO LOPEZ HERRERA (FIRMA)
PERSONA FISICA, CPF-01-1415-0174.
Fecha declarada: 15/10/2024 02:45:37 PM
Esta es una representación gráfica únicamente, verifique la validez de la firma.

Dr. Gustavo López Herrera
Profesor Guía

ALEXIA PACHECO HERNANDEZ (FIRMA)
PERSONA FISICA, CPF-02-0432-0730.
Fecha declarada: 15/10/2024 03:38:39 PM
Esta es una representación gráfica únicamente, verifique la validez de la firma.

Dra. Alexia Pacheco Hernández
Lectora

JULIO CESAR GUZMAN BENAVIDES (FIRMA)
PERSONA FISICA, CPF-01-1381-0186.
Fecha declarada: 12/10/2024 06:40:12 AM
Lugar: Reutlingen, Alemania

M.Sc. Julio Guzmán Benavides
Lector

ALEXANDRA MARIA MARTINEZ PORRAS (FIRMA)
PERSONA FISICA, CPF-01-1026-0736.
Fecha declarada: 15/10/2024 02:51:28 PM
Esta es una representación gráfica únicamente, verifique la validez de la firma.

Dra. Alexandra Martínez Porras
Representante del Director
Programa de Posgrado en Computación e Informática

JOSE DAVID FALLAS DELGADO (FIRMA)
Firmado digitalmente por JOSE DAVID FALLAS DELGADO (FIRMA)
Fecha: 2024.10.15 20:59:20 -06'00'

José David Fallas Delgado
Sustentante

TABLA DE CONTENIDO

RESUMEN

El diseño de la arquitectura de un sistema define la organización e interacción de sus componentes, así como las pautas y reglas que los desarrolladores deben seguir para garantizar productos de alta calidad. Aunque se pueden realizar ajustes durante la implementación, es fundamental que la definición inicial incluya reglas claras para minimizar la probabilidad de errores durante las etapas de codificación y pruebas.

En algunas ocasiones la mala planificación, la mala ejecución de las buenas prácticas, la falta de conocimiento o incluso descuidos, pueden ocasionar que las soluciones desarrolladas tengan problemas y presenten una calidad no deseada. A la acción de acumular este tipo de problemas se le conoce como deuda técnica y puede provocar que el costo de mantenimiento de los sistemas sea tan alto que requiera de invertir mucho esfuerzo, tiempo y dinero.

Los indicios de que un sistema tiene problemas de codificación son conocidos como *bad smells*. Estos se pueden presentar tanto a nivel de la definición como de la implementación de la arquitectura del sistema. En esta investigación se recopilaron un total de 17 *bad smells* que afectan el tipo de arquitectura definido para los sistemas web de la OBAS. A partir de estos, se establecieron un conjunto de 21 reglas que permiten identificar cuándo hay presencia de *bad smells* en la implementación de la arquitectura.

El objetivo principal de esta investigación aplicada se centró en la construcción y evaluación de una herramienta de detección de *bad smells* a nivel de la implementación de la arquitectura de los sistemas web de la OBAS. El diagnóstico inicial del contexto de la OBAS reveló que existen indicios de que los sistemas están acumulando un grado de deuda técnica que lleva a los desarrolladores a tener que retrabajar en funcionalidades, lo que aumenta el esfuerzo y, a su vez, afecta de manera negativa en los tiempos de entrega. El desarrollo de la herramienta de detección automática de *bad smells* se hizo basado en la plataforma de compilación de .Net (Roslyn Analyzer). Luego de realizar pruebas en el contexto de la OBAS, se ratificó que la herramienta mejora el tiempo de búsqueda de posibles errores en el código. Además, se demostró que al incorporar la herramienta al IDE de desarrollo VisualStudio se pueden detectar errores en tiempo real conforme se va ingresando el código nuevo, lo que brinda la opción a los desarrolladores de crear soluciones de mejor calidad.

El aporte de los expertos en el desarrollo de sistemas facilitó la construcción de una herramienta que se adaptó bien al contexto de la OBAS y las personas involucradas en el desarrollo de aplicaciones web de la OBAS quedaron dispuestas a incorporar la herramienta para analizar el código de los sistemas que están en construcción y mantenimiento.

LISTA DE TABLAS

LISTA DE FIGURAS

CAPÍTULO 1. INTRODUCCIÓN

El proceso de desarrollo de los sistemas de información consta de varias etapas fundamentales [1] y es durante las etapas de análisis, diseño que se define la organización e interacción de sus componentes, así como las pautas y reglas que los desarrolladores deben seguir. En algunas ocasiones la mala planificación, la mala ejecución de las buenas prácticas, la falta de conocimiento o incluso descuidos, pueden ocasionar que las soluciones desarrolladas tengan problemas y presenten una calidad no deseada; al acto de acumular este tipo de problemas es lo que se conoce como deuda técnica [2].

Este concepto fue introducido por Cunningham en 1992 [3] para referirse a decisiones que se toman durante la creación de un sistema que, si bien permiten avanzar más rápido en el desarrollo, hacen que la calidad de este sea menor. El concepto es explicado con una metáfora que relaciona los costos resultantes de avanzar en la programación de un sistema dejando elementos de baja calidad, con los intereses que se pagan en una entidad bancaria al solicitar un crédito.

En la etapa de diseño de un sistema, se selecciona la arquitectura que este tendrá y se establecen los fundamentos sobre la forma en que se distribuyen los componentes y cómo interactúan entre sí [4]. Los desarrolladores de software deben ajustarse a los parámetros establecidos en la arquitectura definida para mantener la calidad de las aplicaciones, aunque es común que la calidad se vea afectada debido a la adopción de ciertas decisiones de diseño [5]. Al ignorar parámetros y recomendaciones del diseño, ya sea por poco conocimiento en la arquitectura o por poco tiempo para la implementación, los desarrolladores pueden introducir soluciones de baja calidad [6], lo que puede llevar a crear una deuda técnica que crezca con el tiempo. Esto suele ocurrir por aplicar un enfoque apresurado y descuidado en el que se ignoran principios y conceptos establecidos, en lugar de utilizar un enfoque sistemático y disciplinado [7].

Existen síntomas de que un sistema ha tenido una mala implementación o sufrido de problemas de diseño, a estos síntomas se les conoce como *smells* o *bad smells* [8]. Los *smells* pueden presentarse a niveles bajos, en el código, o a niveles superiores, como en el diseño o

implementación de la arquitectura, a estos últimos se les conoce como *bad smells* arquitectónicos [9].

En el desarrollo de sistemas, es común encontrar elementos de baja calidad o que no cumplen con las reglas de la arquitectura seleccionada, ya sea por decisiones conscientes tomadas por los desarrolladores o por decisiones inconscientes a la hora de implementar los elementos de la arquitectura. La falta de conciencia sobre la deuda técnica generada con estas decisiones puede afectar la calidad del sistema entregado. Detectar las causas requiere de un esfuerzo considerable para los equipos de desarrollo. Sin embargo, hacer uso de herramientas de detección automática puede facilitar el trabajo y disminuir los tiempos para corregir los problemas [10].

En la Oficina de Becas y Atención Socioeconómica (OBAS) de la Universidad de Costa Rica, no existe actualmente un método para detectar la existencia de *bad smells* como síntomas de deuda técnica por lo que no es posible determinar cuáles elementos se pueden corregir para mejorar la calidad de los sistemas desarrollados.

En este trabajo se propone el uso y evaluación de una herramienta de detección de *bad smells* arquitectónicos para determinar la existencia de problemas de diseño o implementación de la arquitectura de los sistemas web de la OBAS, desarrollados con arquitecturas en capas enfocadas a servicios y con uso del estilo MVC para la capa de presentación de datos.

Con base en esto, se plantea la siguiente **pregunta de investigación**:

> ¿Cuál herramienta de detección de *bad smells* arquitectónicos es pertinente aplicar a los sistemas web desarrollados en la Oficina de Becas y Atención Socioeconómica?

1.1. Contexto de la investigación

La Oficina de Becas y Atención Socioeconómica (OBAS), forma parte de la Vicerrectoría de Vida Estudiantil (ViVE) de la Universidad de Costa Rica (UCR). Es una unidad administrativa que gestiona tanto la asignación, el control y seguimiento de las becas como de los beneficios complementarios que cada categoría de beca brinda a las personas estudiantes. Propicia el ingreso de estudiantes de limitados recursos económicos con potencial para realizar estudios superiores [11] y tiene un impacto directo en más de veintidós mil estudiantes becados por año.

Para realizar estas funciones, la OBAS debe capturar información de la situación social y económica de la población estudiantil y sus grupos familiares.

Gran parte de esta información es ingresada e incorporada en sistemas de información que permiten analizar y tomar decisiones con respecto a las becas y beneficios complementarios a los cuáles puede acceder una persona estudiante.

Para facilitar el acceso a los formularios -de solicitudes de beca y beneficios complementarios- y a los sistemas de procesamiento y consulta de datos para la asignación de becas y confección de designaciones, la OBAS cuenta con sistemas web e interfaces de comunicación con otros sistemas institucionales como el Sistema de Acciones de Personal (también conocido como SIRH por las siglas Sistemas de Información de Recursos Humanos) y el Sistema de Administración Financiera (SIAF).

La OBAS cuenta con dos sistemas web en funcionamiento, y uno más que está en etapa de desarrollo, que han sido construidos utilizando el lenguaje de programación Visual Basic .NET (VB.NET) y siguiendo una arquitectura por capas y Orientada a Servicios (SOA, por las siglas del inglés *Service Oriented Architecture)*. Además, se ha utilizado el estilo WebForms para la capa de presentación de datos.

Como parte de las iniciativas del Centro de Informática (CI) de la UCR para mantener uniformidad en los desarrollos de sistemas, los desarrollos futuros deben realizarse con el lenguaje de programación C# y seguir una arquitectura por capas y SOA, agregando funcionalidad del estilo de arquitectura MVC a la interfaz gráfica. Actualmente, se está aplicando esta iniciativa con las etapas de análisis y diseño del nuevo Sistemas de Aplicaciones Estudiantiles que incluirá aplicaciones para las diferentes oficinas de la ViVE.

En la **Figura 1** se pueden observar las capas que utilizan los desarrollos actuales y que será utilizada en desarrollos futuros. La diferencia se verá marcada principalmente por la incorporación de MVC en la interfaz gráfica.

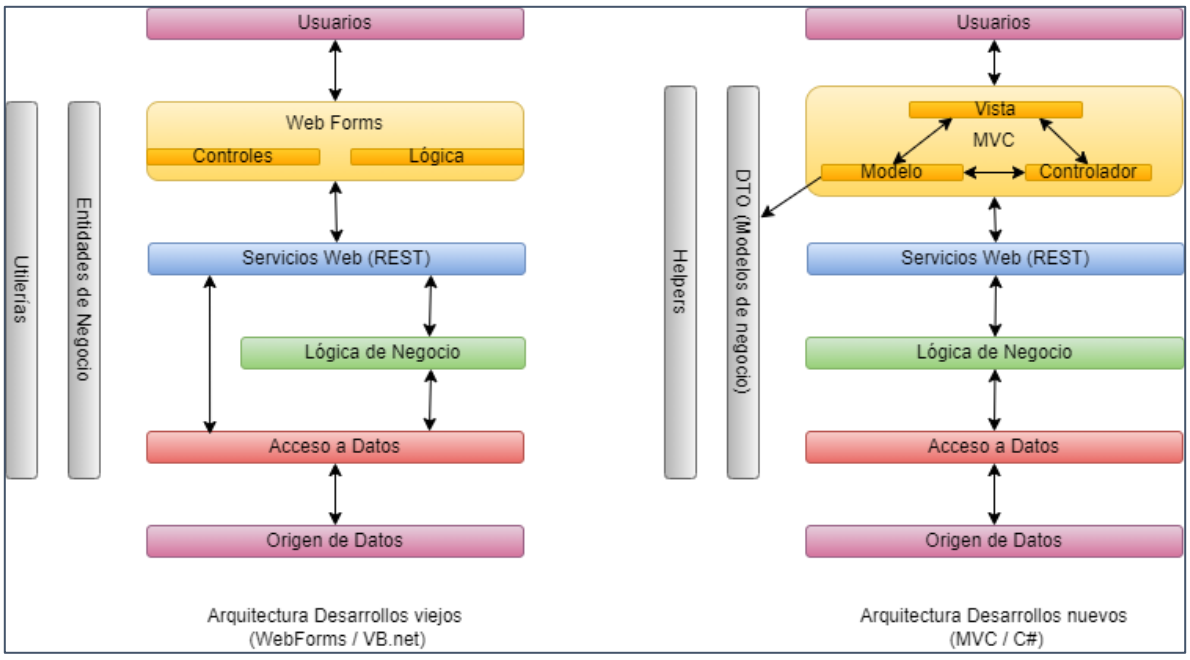

Figura 1 - Arquitectura por capas utilizada en OBAS (elaboración propia)

Los sistemas web de la OBAS son esenciales para la captación y procesamiento de los datos necesarios para la asignación de becas y beneficios complementarios, por lo que se requiere que estos sistemas sean sencillos de mantener y que las mejoras estén disponibles lo más pronto posible. Existen estudios que evidencian la percepción de deuda técnica en la industria de Costa Rica [2], donde se indica que retrasos en la entrega, necesidad de volver a trabajar sobre una porción de código y contar con una baja mantenibilidad del sistema, son algunas de las principales consecuencias. Es por esta razón que para la OBAS es importante contar con una herramienta que permita detectar indicios de problemas en el desarrollo de los sistemas web.

1.2. Objetivos

Para responder a la pregunta de investigación se establece el siguiente objetivo general de investigación: Evaluar el uso de una herramienta de detección de *BadSmells* arquitectónicos en los sistemas web de la Oficina de Becas y Atención Socioeconómica.

Los objetivos específicos de la presente investigación incluyen:

1. Diagnosticar la existencia de deuda técnica en la Oficina de Becas y Atención Socioeconómica.
2. Comparar herramientas de detección de *Bad smells* arquitectónicos para los lenguajes de programación C# y Visual Basic .NET.
3. Valorar la aplicabilidad y pertinencia de una herramienta de detección de *Bad smells* arquitectónicos en los sistemas web de la Oficina de Becas y Atención Socioeconómica.

Para efectos de la conclusión de estos objetivos, se toma en cuenta que el concepto de aplicabilidad se refiere a la capacidad de la herramienta de detección para ser utilizada en un contexto específico [12], mientras que el concepto de pertinencia se refiere a que es apropiado y significativo utilizar este tipo de herramientas en el contexto evaluado [13].

1.3. Estructura

El resto del presente documento se estructura de la siguiente manera: En el Capítulo 2 se brinda información relevante a la investigación, explicando conceptos básicos relacionados con la arquitectura de sistemas y los *bad smells* como indicios de deuda técnica. En el Capítulo 3 se presentan estudios previos relacionados con el tema de investigación. El Capítulo 4 detalla la metodología utilizada para responder a la pregunta de investigación. Se detallan los resultados de la investigación en el Capítulo 5 y se presentan las conclusiones y trabajo futuro en el Capítulo 6. Finalmente se listan las referencias bibliográficas que sustentan la investigación y se añaden anexos que complementan la documentación.

CAPÍTULO 2. MARCO CONCEPTUAL

A continuación, se explican los conceptos más importantes para comprender los objetivos de la propuesta de investigación.

2.1 Arquitectura de sistemas

La **arquitectura de un sistema** es el conjunto de estructuras necesarias para razonar sobre el sistema y comprende los elementos, las relaciones entre ellos y los principios operativos [14]. Esto incluye la forma en que el sistema está organizado en subsistemas y componentes, la ubicación de la funcionalidad dentro de estos y cómo interactúa cada componente en el ambiente de ejecución [4]. La **arquitectura orientada a servicios** (*Service-Oriented Architecture* SOA), por ejemplo, permite que los servicios interactúen entre sistemas y dominios encapsulando cada componente y las conexiones de datos necesarias para llevar a cabo las operaciones de una manera desacoplada, lo cual ayuda a reducir la dependencia entre componentes [15].

Por otra parte, contar con **estilos de arquitectura** ayuda a los desarrolladores a definir de mejor manera las características básicas y el comportamiento de la aplicación. Uno de los estilos más comunes es el de **arquitectura en capas** (también conocido como arquitectura de *n* niveles) [16]. En este estilo, los componentes se organizan en capas horizontales, donde cada capa desempeña una función específica. No existe un número específico de capas, sin embargo, la mayoría de las arquitecturas en capas constan de cuatro capas estándar: (1) presentación, (2) lógica de negocio, (3) persistencia y (4) base de datos.

Otro estilo ampliamente utilizado es el **modelo-vista-controlador** (MVC), que consta de tres componentes principales: (1) el modelo, formado por una o más clases que trabajan juntas para realizar las funcionalidades del sistema; (2) la vista, formada por una o más clases que trabajan juntas para implementar la representación visual del modelo; y (3) el controlador, formado por una o más clases que trabajan juntas para coordinar la mensajería entre modelo y vista [17].

2.2 Deuda técnica y *bad smells*

La **Deuda técnica** es un concepto introducido por Ward Cunnigham en 1992 [3] para referirse a código de baja calidad que es entregado y que, si no es reparado de manera oportuna, será más costoso conforme se avance en el desarrollo. Actualmente el concepto ha evolucionado y es utilizado para describir decisiones técnicas que pueden beneficiar a una compañía a corto plazo (como la entrega más rápida de soluciones para el cliente), pero que puede producir mayores costos y problemas en la calidad de las aplicaciones a largo plazo [2].

Un ***bad smell*** es un síntoma de un posible problema en el diseño o la codificación de un sistema [8]. El término se popularizó luego de la publicación del libro *Refactoring* de Fowler en 1999 [18], en el que el autor define 22 *bad smells* de código y brinda recomendaciones para realizar la refactorización del código y poder corregirlos.

Cuando los *bad smells* se encuentran en un nivel superior de diseño, es decir, en la arquitectura del sistema, son llamados ***bad smells* arquitectónicos** [9]. Estos ocurren debido a diferentes causas, entre las cuales están: aplicar un concepto de diseño arquitectónico en un contexto inapropiado, mezclar responsabilidades entre componentes o aplicar abstracciones de diseño en un nivel incorrecto [7]. Esto suele suceder debido al poco entendimiento o poco cuidado de parte de los desarrolladores por implementar bien la arquitectura; o a tiempos cortos para la entrega del producto, que inducen a los desarrolladores a violar la arquitectura definida con tal de cumplir con las fechas límites [9].

El estilo MVC ha sido ampliamente adoptado por la industria para el desarrollo de aplicaciones web [19]; sin embargo, algunas de las reglas definidas en este estilo no son respetadas por los desarrolladores, lo que da como resultado decisiones de diseño deficientes de diseño o implementación, a esto se le conoce como ***bad smells* MVC** [7].

Detectar *bad smells*, tanto en código como en arquitectura, requiere de mucho esfuerzo para los equipos de desarrollo y el resultado se ve afectado por la percepción de cada desarrollador. Hacer uso de estrategias formales y bien definidas puede colaborar en la detección, eliminando los sesgos relacionados con las personas [10]. Una de las estrategias existentes, es el **análisis estático de código,** que consiste en el análisis de alguna versión de código fuente sin que este

sea ejecutado [7]. El análisis puede ser ejecutado de manera automática, a través de una herramienta de detección, o de manera manual, a través de la revisión del código por parte de una persona.

CAPÍTULO 3 - ANTECEDENTES

En esta siguiente sección se listan trabajos que se han enfocado en caracterizar, catalogar e identificar *bad smells* de código y de arquitectura.

En 2016, Aniche et al. [19] publican un conjunto de *bad smells* específicos del estilo de arquitectura MVC. Para llegar a este catálogo, los autores entrevistan y encuestan a 53 desarrolladores MVC sobre malas prácticas que se deben evitar cuando se desarrollan aplicaciones web que utilizan este estilo de arquitectura. Luego de la recopilación de datos, definen un catálogo de 6 *bad smells* MVC enfocados en los componentes del Modelo y del Controlador. Posteriormente, analizan sistemas desarrollados con Spring MVC, un *framework* para el lenguaje de programación Java. Adicionalmente, encuestan a 21 desarrolladores para verificar su percepción sobre los *bad smells* definidos. Concluyen su trabajo indicando que los *bad smells* definidos pueden tener un impacto negativo, ya que aumentan las probabilidades de que el sistema presente defectos y requiera cambios, afectando la mantenibilidad de los sistemas.

En 2018, Velasco et al. [7] realizan una caracterización de *bad smells* relevantes al estilo de arquitectura MVC con el objetivo de llenar un vacío que encuentran con respecto al enfoque para detectar y corregir de manera automática los *bad smells* arquitectónicos de este estilo. En búsqueda de lograr su objetivo, realizan una recopilación de información de diferentes fuentes para caracterizar el estilo de arquitectura, las restricciones que en este se aplican y los *bad smells* que indican la existencia de posibles infracciones. Luego, para determinar la utilidad de herramientas de detección automática de *bad smells* MVC, extienden las capacidades de un analizador de código estático llamado *PHP_Sniffer* y definen estándares de código para analizar cinco sistemas desarrollados con Yii, que es un *framework* de desarrollo Web para el lenguaje PHP. Finalmente, comparan los resultados obtenidos con su herramienta contra los resultados de una búsqueda manual realizada por 15 desarrolladores. Concluyen su trabajo afirmando que las herramientas de detección *de bad smells* arquitectónicos son ciertamente útiles para evaluar las partes del código que irrespetan las restricciones impuestas por el estilo arquitectónico utilizado y proponen como trabajo futuro la inclusión de nuevos *bad smells* y ampliar el soporte para más lenguajes de programación en su herramienta de detección automática.

En 2019, Sabir et al. [6] realizan una revisión sistemática de literatura sobre la detección de *bad smells* y su evolución en sistemas orientados a objetos y orientados a servicios. En esta revisión, los autores detectan una serie de *bad smells* de código clasificados como *smells* arquitectónicos. Además, identifican dos técnicas de detección de *smells* que son muy utilizadas en la literatura: (1) análisis estático de código y (2) análisis dinámico de código basado en adaptación dinámica de parámetros. Discuten sobre la dificultad y el tiempo que consume realizar una clasificación y detección manual de *bad smells*, por lo que recomiendan un abordaje experto, tanto en la academia como la industria, para proveer un catálogo de categorías de *smells* y sus enfoques de detección. Finalmente, proponen para trabajos futuros la integración de técnicas de detección automática de *smells* a través de los IDE (entornos de desarrollo integrado) de software.

En 2021, Mumtaz, Singh y Blincoe [5] efectúan un mapeo sistemático sobre detección de *bad smells*. En este trabajo listan técnicas y herramientas para la detección de *bad smells* arquitectónicos para identificar sus limitaciones. Hallan que la técnica que se aplica con mayor frecuencia consiste en el enfoque basado en reglas predefinidas, haciendo uso de métricas y parámetros. Con respecto a los estilos de arquitectura, observan que aproximadamente el 60% de las técnicas basadas en reglas están enfocadas al estilo de arquitectura orientada a servicios; mientras que sólo una se enfocó en el estilo de arquitectura modelo-vista-controlador. En el tema de herramientas existentes y los lenguajes de programación que soportan, evidencian que Java es el lenguaje con mayor soporte; mientras que lenguajes como .Net, Python y PHP reciben poca atención. Concluyen el estudio mencionando que hay una necesidad de cuantificar y medir *bad smells* arquitectónicos enfocándose en identificar apropiadamente las métricas y parámetros para su detección; además de la necesidad de involucrar a desarrolladores en las evaluaciones de técnicas y herramientas para mejorar la aplicabilidad de estas en escenarios del mundo real.

En los trabajos mencionados, se identifica que tanto el estilo de arquitectura modelo vista controlador (MVC) como el lenguaje de programación .NET han sido poco abordados. En el presente trabajo se aborda este vacío a través de la recopilación de varios catálogos de *bad smells* que tienen impacto en estilos de arquitectura MVC. Además, se realiza la evaluación de una herramienta de detección de estos en el contexto de la OBAS, adaptando los *bad smells* a la arquitectura utilizada y a los lenguajes de programación VB.net y C#. Para esta evaluación se

contó con la participación de desarrolladores expertos en este lenguaje de programación siguiendo las recomendaciones propuestas por Sabir et al. [6] y Mumtaz, Singh y Blincoe [5] acerca de la importancia de realizar un abordaje con expertos tanto del académico como del ámbito laboral.

CAPÍTULO 4 - METODOLOGÍA

En esta sección se detalla la metodología utilizada durante esta investigación, que se basó en el modelo de capas de diseño de investigación propuestas por Saunders y Tosey [20] que proporcionan una estructura integral y sistemática para diseñar investigaciones. Este modelo incluye las siguientes capas: (1) Filosofía de la investigación: Se refiere a las creencias y suposiciones sobre cómo se debe desarrollar el conocimiento. (2) Elección de la metodología: Describe el enfoque para analizar los datos. (3) Estrategia de investigación: Incluye las técnicas y procedimientos específicos utilizados para recopilar y analizar datos. (4) Horizonte temporal: Define el marco temporal, ya sea transversal o longitudinal. (5) Técnicas y procedimientos: Detalla los métodos específicos de recolección y análisis de datos.

Se hizo uso del paradigma de investigación de tres ciclos para las ciencias de diseño propuesto por Hevner [21], que se puede observar en la **Figura 2**. Para esto se define: (1) el ciclo de relevancia: donde se analizó el contexto de los sistemas web de la OBAS y se establecieron los requerimientos técnicos; (2) el ciclo de diseño: en el que se desarrolló y evaluó una versión piloto de la herramienta para la detección de *bad smells*, identificando mejoras y aplicando correcciones para proceder con la implementación de la versión final; y (3) el ciclo de rigor: donde se aprovechó la base de conocimiento existente sobre *bad smells* y la experticia de los desarrolladores para la evaluación final de la herramienta desarrollada.

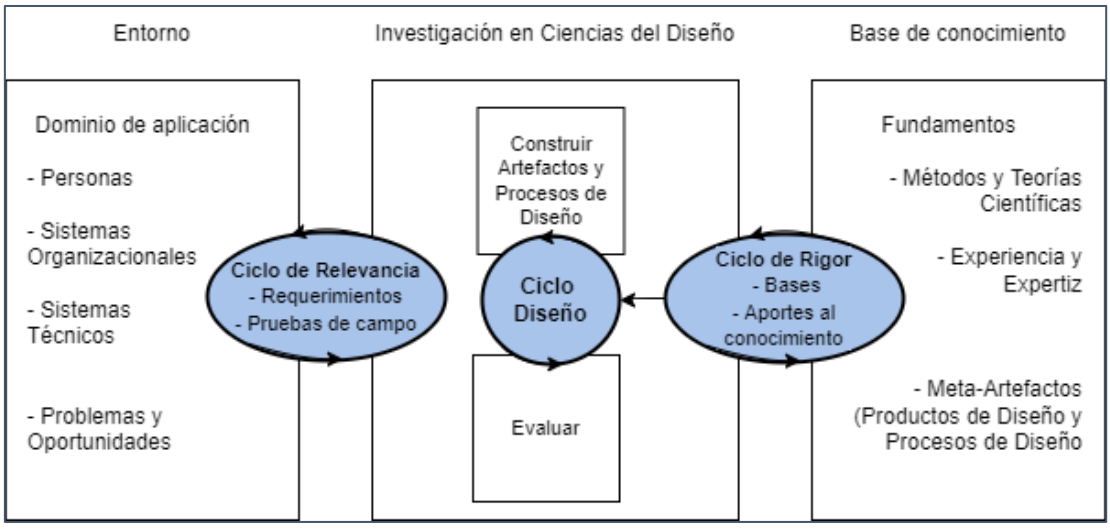

Figura 2 - Ciclos de investigación de ciencias del diseño (Traducido de Hevner, 2007)

Se aplicó la metodología mixta de investigación de acuerdo con el diseño convergente paralelo, que propone la recolección y análisis simultáneo de datos cualitativos y cuantitativos, integrándolos durante la interpretación de la información [22]. Por un lado, se analizaron aspectos cualitativos como precio, tipo de licencia, alcance de tipos arquitectura y lenguajes soportados por las herramientas que permiten detectar *bad smells* arquitectónicos; así como la percepción de deuda técnica existente en la OBAS y las opiniones sobre la aplicabilidad y pertinencia de una herramienta de detección automática de *bad smells*. Por otra parte, se analizaron aspectos cuantitativos relacionados con la cantidad de *bad smells* arquitectónicos que pueden detectar las herramientas existentes en el mercado y se analizó la estadística relacionada con la evaluación de la herramienta de detección de *bad smells*.

Además, se aplicó un caso de estudio donde se utilizó la herramienta desarrollada a los sistemas web desarrollados en la OBAS y se emplearon entrevistas semi estructuradas para captar percepciones de los desarrolladores de software de dicha oficina sobre el uso de dicha herramienta. El estudio fue realizado de manera transversal, es decir, en un momento específico de tiempo.

En la **Figura 3** se presenta un resumen de las actividades que se efectuaron para lograr los objetivos de la investigación.

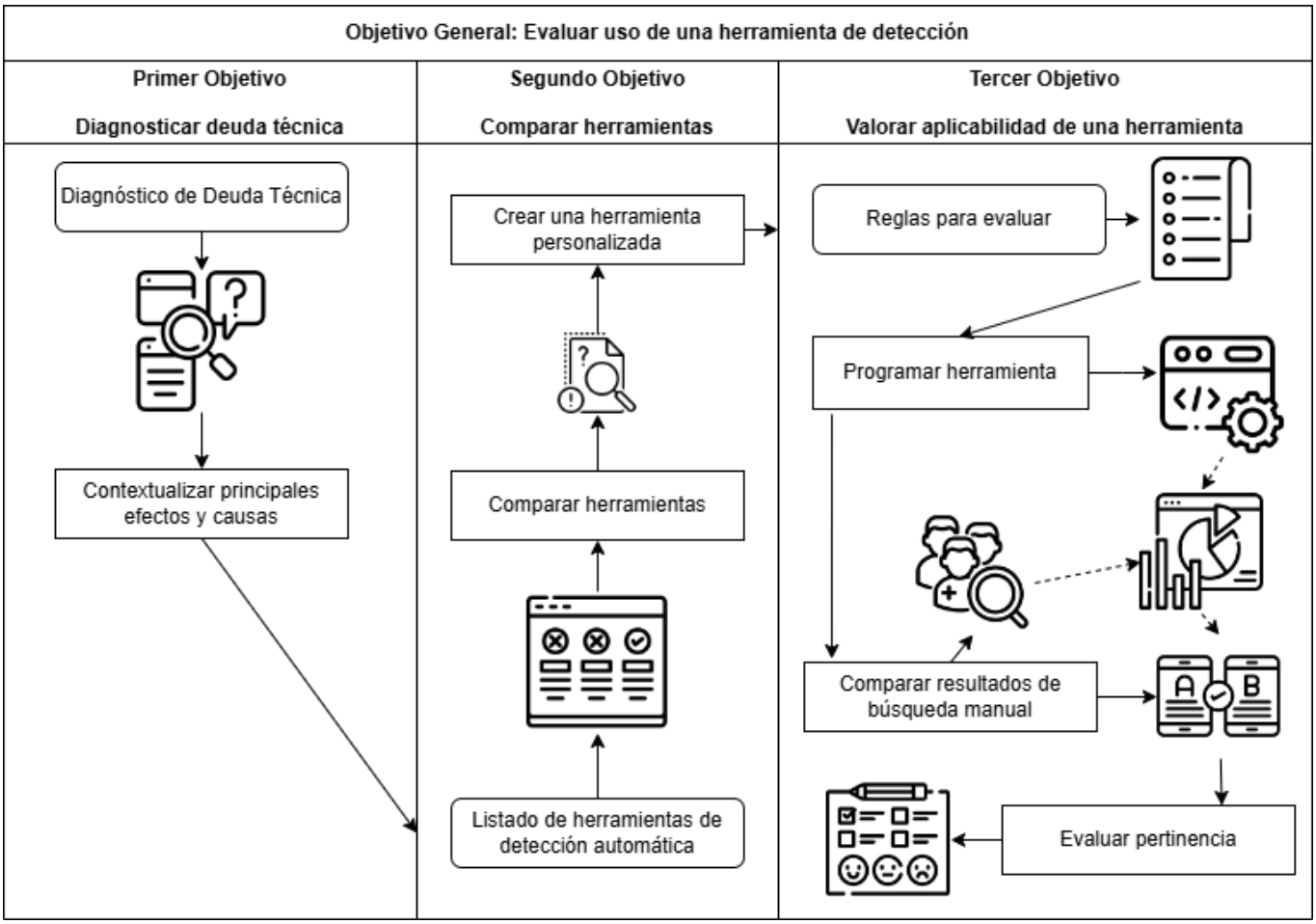

Figura 3 - Diagrama de metodología (elaboración propia)

Para cumplir con el primer objetivo específico: “Diagnosticar la existencia de deuda técnica en la Oficina de Becas y Atención Socioeconómica”, se aplicó un instrumento de diagnóstico de deuda técnica [23], tanto para determinar la existencia de deuda técnica en los sistemas web de la organización, como para identificar sus posibles causas y efectos. Este instrumento fue aplicado a un grupo de 4 desarrolladores de la OBAS y un grupo de 2 desarrolladores del CI que están involucrados en el proceso de desarrollo de sistemas web de la OBAS. Con los resultados del instrumento aplicado, se determinó que hay indicios de deuda técnica en el contexto aplicado y que los principales efectos percibidos por el grupo de desarrolladores son

retrasos en tiempo de entrega, necesidad de retrabajo y aumento en el esfuerzo de desarrollo de aplicaciones.

Posteriormente, para alcanzar el siguiente objetivo específico, “Comparar herramientas de detección de *BadSmells* arquitectónicos para los lenguajes de programación C# y Visual Basic .NET”, se realizó una revisión de literatura para identificar y listar las herramientas propuestas en la literatura para la detección de *bad smells* arquitectónicos en el lenguaje de programación utilizado por la organización en estudio. Una vez identificadas las herramientas existentes, se realizó una comparación entre ellas, incluyendo sus principales características y los *bad smells* arquitectónicos que pueden identificar.

Se determinó que las herramientas existentes mantienen un enfoque de detección de errores de código fuente que no incluyen aspectos para la detección de problemas de diseño o implementación de la arquitectura de los sistemas web desarrollados en la OBAS, por lo que se tomó la decisión de codificar una herramienta con la capacidad de detectar la infracción de reglas definidas a través de Analizadores Roslyn. Para este proceso, se construyó primero una herramienta capaz de detectar un set de prueba con 5 *bad smells* con sus reglas correspondientes de detección y se presentó a uno de los desarrolladores expertos de la OBAS para obtener retroalimentación sobre el uso del prototipo. Luego de determinar aspectos a mejorar, tales como las descripciones de cada error y el ajuste en las reglas que validan las capas de la arquitectura, se procedieron a aplicar las correcciones y continuar con la codificación de la herramienta para obtener el producto final con capacidad de detectar el set completo de 17 *bad smells*.

Por último, para cumplir con el tercer objetivo específico de esta investigación, “Valorar la aplicabilidad y pertinencia de una herramienta de detección de *bad Smells* arquitectónicos en los sistemas web de la Oficina de Becas y Atención Socioeconómica”, se entablaron reuniones con los desarrolladores involucrados en la construcción de aplicaciones web en la OBAS con el objetivo de presentar los conceptos de deuda técnica junto con las reglas que se estaban evaluando para detectar la presencia de *bad smells* arquitectónicos.

Después de la exposición de las reglas, se acordó con los desarrolladores un periodo de 15 minutos para que pudieran realizar una búsqueda manual de *bad smells* en uno de los sistemas

con el estilo de arquitectura MVC que estaba en etapa de desarrollo y que era familiar para los participantes. Este tiempo se fijó en 15 minutos con base en los comentarios de los desarrolladores que indicaron que normalmente no realizan ningún tipo de revisión de código por el poco tiempo que tienen para avanzar con sus tareas diarias.

Posterior a esto, se les presentó la herramienta de detección automática desarrollada y se demostró como se ejecutaba un análisis en el mismo sistema en el que se buscaron *bad smells* de manera automática. Se recopilaron los resultados de ambas búsquedas y se analizaron junto con los participantes para comparar tanto la cantidad de *bad smells* detectados como el tiempo y esfuerzo invertido en detectarlos.

Finalmente, se aplicó un instrumento (ver **Anexo 2**) construido para entrevistar a cada participante de manera individual y así valorar su percepción sobre aspectos de eficacia, eficiencia y uso de la herramienta para determinar su aplicabilidad y pertinencia en los sistemas de la OBAS.

En el siguiente capítulo, de profundiza en los pasos que se ejecutaron para cada actividad descrita en este capítulo y se detallan los resultados que llevaron a cumplir los objetivos propuestos y así responder la pregunta de investigación.

CAPÍTULO 5 - RESULTADOS

5.1 Diagnóstico de Deuda Técnica

Se inició la investigación indagando sobre la posible existencia de deuda técnica en la Oficina de Becas y Atención Socioeconómica; para esto se empleó un instrumento especializado en la identificación de efectos y causas de deuda técnica en el contexto aplicado [23]. El instrumento consta de una serie de preguntas semiabiertas y cerradas que se ajustan de manera dinámica para guiar a los participantes en la identificación de las causas que pueden estar relacionadas con los efectos que perciben. Cuenta con una base de datos que se originó a raíz de un estudio exploratorio que identificó las causas y efectos comunes de deuda técnica en Costa Rica [2].

El instrumento fue aplicado a un grupo de 6 programadores que trabajan de manera directa en los sistemas web de la OBAS con el objetivo de obtener la percepción de estos con respecto a los efectos, posibles causas y acciones que se toman con relación a aspectos de deuda técnica. A cada uno de ellos se le brindó una explicación de los principales conceptos de deuda técnica, se le explicó el objetivo de la investigación y se le indicó que para contestar el instrumento debían elegir las opciones que le hicieran sentir más identificado, aclarando que no hay respuestas correctas o incorrectas. Finalmente, se le envió el enlace para que pudieran contestar las preguntas de manera individual.

5.1.1. Resultado del diagnóstico

En la **Figura 4** se presentan los resultados de la percepción sobre los principales efectos que la deuda técnica puede estar generando en los proyectos de desarrollo. La totalidad de los participantes indicaron que existe la necesidad de realizar constantemente un retrabajo en el código y que además hay retrasos en los tiempos de entrega estimados para cada tarea asignada. La mitad de los participantes indican que perciben que existe un aumento en el esfuerzo a la hora desarrollar mientras que la tercera parte exponen la necesidad de refactorizar código. Otros efectos indicados por los participantes incluyen el bajo rendimiento en las soluciones brindadas y una baja mantenibilidad del código, además de la generación de estrés para las partes interesadas.

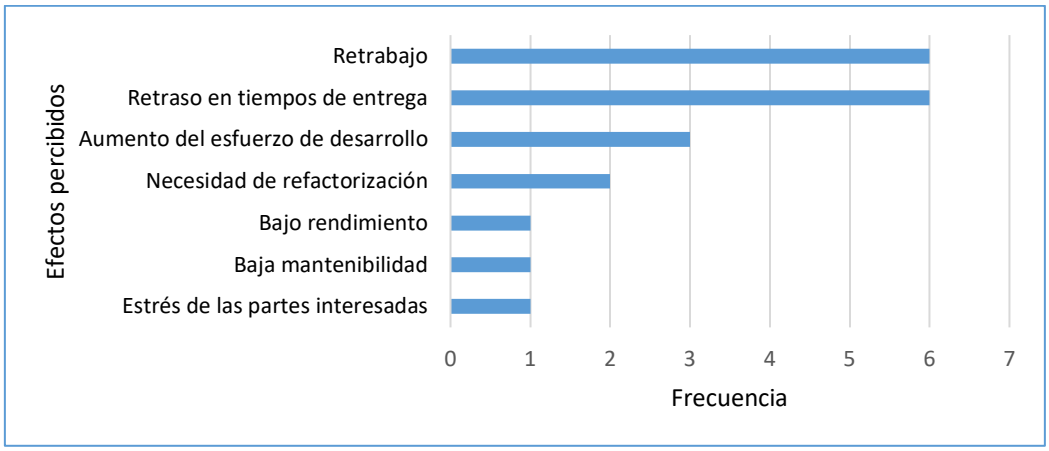

Figura 4 – Gráfico efectos de deuda técnica en sistemas web de la OBAS

En la **Figura 5** se pueden apreciar los resultados de la percepción de las causas que provocan deuda técnica en los sistemas web de la OBAS, según las respuestas de los participantes. Se identifican 16 causas y de acuerdo con la frecuencia en que fueron seleccionadas por los participantes, se denota un porcentaje muy alto de causas relacionadas con la planificación y gestión. En la categoría de conocimiento, se puede observar que hay un porcentaje alto relacionado con la falta de conocimiento técnico y la falta de experiencia. Cabe recalcar que estas son causas que pueden llevar a la incorporación de *bad smells* de código y a una mala implementación de la arquitectura [6] [9].

Otros aspectos que influyen de manera directa son los relacionados con la metodología aplicada y la presencia de problemas durante el desarrollo de los sistemas; mientras que aspectos relacionados con el personal, la organización y factores externos tienen muy poca afectación en la presencia de deuda técnica en la OBAS.

Con los resultados arrojados por el instrumento aplicado, se pudo determinar que hay indicios de la existencia de deuda técnica en los sistemas web de la OBAS. Teniendo como principales causas aspectos de planificación y falta de conocimiento.

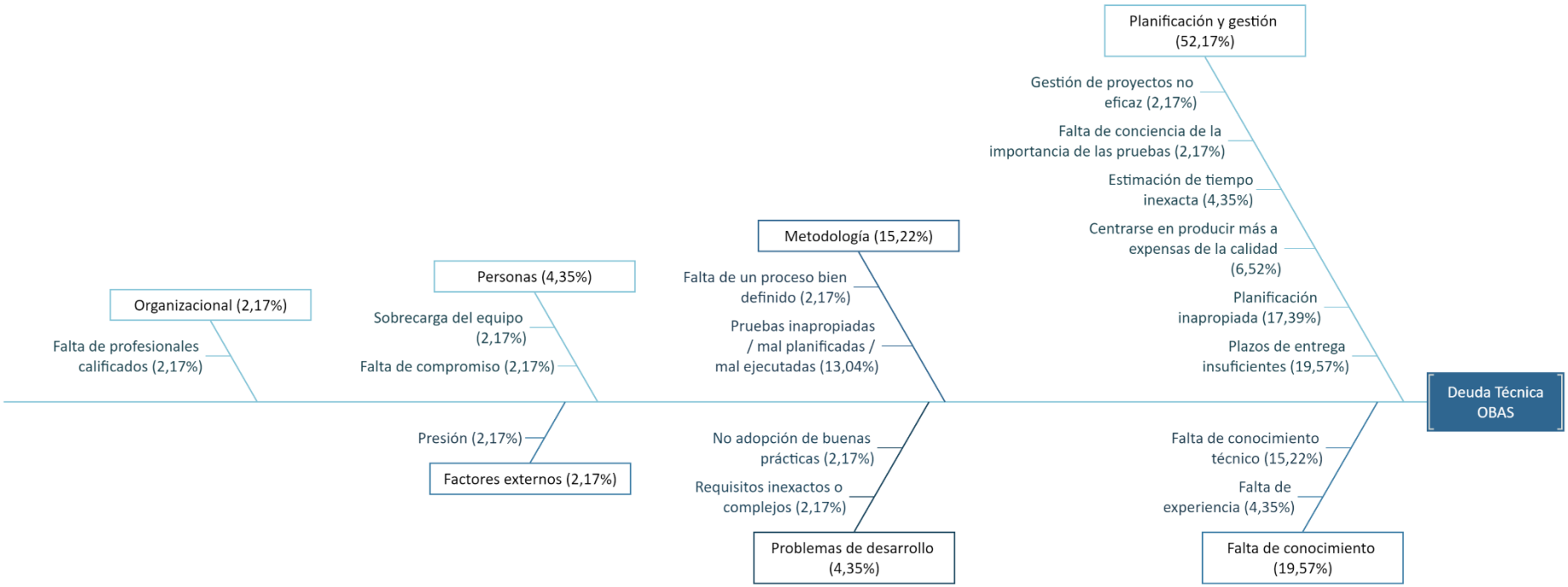

Figura 5 - Causas de deuda técnica en sistemas web de la OBAS

5.2 Identificación de herramientas de detección

El siguiente paso en la presente investigación fue la búsqueda de una herramienta que permitiese detectar *bad smells* en los lenguajes C# y VB.NET, y que además permitiese detectar *smells* relacionados con el estilo de arquitectura utilizado por la OBAS. Para ello, se utilizaron las bases de datos ScienceDirect, IEEE, ACM Digital Library y SpringerLink; y aplicando la siguiente cadena de búsqueda:

> *"architecture" AND "smell*" AND ("detection" or "identification" or "estimation") AND ("tool" or "platform") AND (".net" AND "C#")*

De los resultados obtenidos, se tomaron en cuenta aquellos que listan herramientas de detección de *bad smells* con soporte para código en lenguajes C# y VisualBasic.NET, o que trabajan sobre ensamblados de .NET. Se recopiló una lista de herramientas para buscar la documentación oficial en cada uno de los sitios web y verificar que la capacidad de detectar *bad smells* arquitectónicos. Finalmente se determinó la existencia de las herramientas listadas en la **Tabla 1**.

5.2.1 Resultado de la comparación

Luego de detectar y listar las herramientas propuestas en la literatura, se realizó una revisión de la documentación en el sitio oficial de cada herramienta. Es importante indicar que no existe documentación explícita, para ninguna de las herramientas, que indique la capacidad de detectar *bad smells* enfocados en el estilo de arquitectura MVC; por lo que fue necesario buscar alguna herramienta con capacidad para detectar errores de acuerdo con la validación de reglas personalizadas y que además sea sencillo de integrar en los ambientes de desarrollo según lo recomiendan Sabir et al [6].

Herramienta	*Bad Smell* Arquitectónicos detectados	Lenguajes Soportados	Precio anual
Structure 101 Se compone de un conjunto de herramientas para visualizar y administrar la arquitectura de sistemas complejos de software. **[24]**	- Cyclic Dependency - *Architecture Violation* (a través de diagramas)	C/C++, Java, **.Net**, ActionScript, InterSystems Cache Objects, Pascal, PHP, Python, SQL y UML	$349 a $2395
NDepend Es una herramienta de análisis estático con enfoque en desarrollo de aplicaciones .Net. Ofrece un gran número de características, entre las que destacan la visualización de gráficos interactivos y el uso de métricas para calcular la calidad del código **[25]**. Permite el uso de reglas personalizadas a través de Code Query LINQ **[26]**.	- Mutually dependent namespaces and types - Cyclic dependency - Many small library Assemblies - Relational cohesion - Large Classes/Methods - Long Parameter List - Feature Envy - *Architecture violation* (Requiere reglas personalizadas)	**.Net**	$492 a $1232

Herramienta	*Bad Smell* Arquitectónicos detectados	Lenguajes Soportados	Precio anual
ReSharper Es una herramienta/extensión que se integra como complemento a Visual Studio. Ofrece un amplio conjunto de características que incluyen un asistente de codificación inteligente, reporte y corrección de errores sobre la marcha y refactorización de código **[27]**.	- Cyclic Dependency - Large Classes/Methods - Long Parameter List - Feature Envy - *Architecture violation* (Requiere reglas personalizadas)	C#, **VB.NET**, XAML, **ASP.NET**, TypeScript, JavaScript, CSS	$349 a $779
SonarQube Es una herramienta especializada en la revisión y medición automática de código, se integra en el flujo de trabajo existente y detecta problemas en el código **[28]**. Se basa en el uso de reglas y métricas para calcular el tiempo requerido para corregir los problemas **[29] [30]**.	- Large Classes/Methods - Long Parameter List - Feature Envy - *Architecture violation* (Requiere reglas personalizadas)	C#, CSS, Java, JavaScript, PHP, Python, Ruby, **VB.NET**, HTML, XML, entre otros.	$0 a $20.000

Tabla 1 - Lista de herramientas de detección de *bad smells*

Para la detección de errores en el código De Kourt [31] propone el uso de analizadores automáticos de código que complementen la administración de deuda técnica. Además, explica el funcionamiento de la plataforma de compilación de .NET *(.NET Compiler Platform)* que permite la evaluación de reglas personalizadas, también conocidas como analizadores Roslyn [32]. Estos analizadores pueden ser codificados como proyectos independientes capaces de analizar código de proyectos específicos y pueden ser integrados en el IDE (por sus siglas en inglés para Entorno de Desarrollo Integrado) de cada desarrollador para realizar detecciones de código tanto de manera manual como de manera automática.

5.3 Codificación de la herramienta personalizada de detección de *bad smells*

Teniendo en cuenta la limitación de las herramientas actuales y la capacidad de los analizadores Roslyn para generar reglas personalizadas, se tomó la decisión de crear una herramienta nueva para identificar *bad smells* arquitectónicos en los sistemas web de la OBAS a través de la codificación de reglas aplicables en el contexto y estilo de arquitectura existente.

5.3.1 Definición de reglas a evaluar

El primer paso en la construcción de la herramienta fue definir las reglas que deben ser evaluadas a través de los analizadores personalizados. Para esto, se inició con la recopilación de catálogos propuestos en la literatura enfocados en *bad smells* arquitectónicos. De los catálogos encontrados en [7], [19], [28], [29] y [33], se realizó una selección de 17 *bad smells* que pueden ser aplicados al estilo de arquitectura utilizado en la OBAS. En la **Tabla 2** se listan y clasifican estos de acuerdo con el ámbito en que serán evaluados.

Luego, para cada uno de los *bad smells* listado, se definió una serie de reglas para ser evaluadas con la herramienta de detección automática. Cuando la regla se rompe, se notifica al desarrollador marcando la clase o la porción de código de código que tiene un error y a través de la descripción del error se le brinda información detallada del tipo de regla que incumple. En el **Anexo 1** se listan las 21 reglas definidas para la detección de los *bad smells*.

Id	Nombre	Descripción	Tipo	Componente
1	Modelo es Vista	Ocurre cuando el Modelo realiza responsabilidades relacionadas a presentación de datos en la interfaz del usuario.	MVC	Modelo
2	Modelo es Controlador	Ocurre cuando el Modelo hace uso directo de los métodos HttpPost y HttpGet.	MVC	Modelo
3	Vista es Modelo	Ocurre cuando la Vista tiene responsabilidades relacionadas a la consulta y persistencia de datos.	MVC	Vista
4	Vista es Controlador	Ocurre cuando la Vista accede de forma directa a información sobre las solicitudes del usuario.	MVC	Vista
5	Controlador es Vista	Ocurre cuando el Controlador realiza responsabilidades relacionadas a presentación o recepción de datos en la interfaz del usuario.	MVC	Controlador
6	Controlador es Modelo	Ocurre cuando el Controlador tiene responsabilidades relacionadas a la consulta y persistencia de datos.	MVC	Controlador
7	Controlador Cerebral (*Brain Controller*)	Controladores tienen demasiado control de flujo.	MVC	Controlador
8	Controlador Promiscuo (*Promiscuos Controller*)	Controladores que ofrecen demasiadas acciones.	MVC	Controlador
9	Servicio Entrometido (*Meddling Service*)	Servicios que consultan directamente la base de datos.	Capas / Servicio	Capa de Negocio
10	Repositorio Cerebral (*Brain Repository*)	Lógica compleja en el repositorio.	Capas / Repositorio	Capa de Acceso a Datos
11	Repositorio Grande (*Fat Repository*)	Un repositorio que gestiona demasiadas entidades.	Capas / Repositorio	Capa de Acceso a Datos

Id	Nombre	Descripción	Tipo	Componente
12	Repositorio Laborioso (Laborious Repository Method)	Un método de repositorio que tiene múltiples acciones de base de datos.	Capas / Repositorio	Capa de Acceso a Datos
13	Abstracción sin desacople (Abstraction without Decoupling)	Cuando un cliente utiliza un servicio representado como un tipo abstracto, pero también una implementación concreta de este servicio.	General	General
14	Interface ambigua (*Ambiguous interface*)	Cuando una interface ofrece un único punto de entrada general a un componente e internamente despacha a otros servicios o métodos más específicos de acuerdo a los parámetros generales que recibe.	General	General
15	Violación de la definición de la arquitectura (*Architecture Violation*)	Cuando la definición de la arquitectura es diferente de la implementación actual.	General	General
16	Clase Dios (God Class / Component)	Cuando una clase o componente es excesivamente grande.	General	General
17	Dependencia Cíclica (*Cyclic Dependency*)	Cuando dos o más componentes de la arquitectura dependen entre sí directa o indirectamente.	General	General

Tabla 2 - Lista de *Bad Smells* Arquitectónicos

En el caso del *bad smell* "17-Dependencia Cíclica (Cyclic Dependency)" se encontró que existe una regla predeterminada en la plataforma de compilación de .NET, que está habilitada por defecto en el entorno de desarrollo de VisualStudio, que es capaz de detectar la presencia de herencia cíclica entre clases según lo define Microsoft en el Error ID: **BC30257** [34].

Luego de definir las reglas que se deben evaluar, se investigó sobre la forma en que funcionan los analizadores Roslyn para poder implementar el código que evalúa el complimiento de cada regla. A continuación, se presenta el funcionamiento básico de un analizador y se presenta un ejemplo del código desarrollado para la herramienta de detección de *bad smells*.

5.3.2 Analizadores Roslyn

Un analizador Roslyn es una herramienta que se encarga de analizar código fuente en búsqueda de problemas de estilo, calidad, mantenibilidad y diseño. El análisis ocurre en tiempo real durante el momento de codificación de aplicaciones o puede ser ejecutado manualmente para re inspeccionar el código existente. Está basado en la plataforma de compilador de .NET y es capaz de analizar código desarrollado con los lenguajes VB.NET o C# [32].

Una parte fundamental de un analizador Roslyn es el **diagnóstico**, que se encarga de ejecutar el código que valida las reglas definidas y reportar alguna infracción en el código analizado. Para esto es necesario definir un **descriptor** para cada regla (que contiene los detalles como el título, la descripción y la gravedad), registrar la regla en la plataforma de compilación y codificar los métodos que permiten analizar la sintaxis o la semántica del código fuente.

Los niveles de diagnóstico son los que indican la gravedad del problema encontrado por el analizador y la forma en que se muestran al usuario. Los niveles existentes son: error, advertencia, información y oculto [32].

5.3.3 Herramienta de diagnóstico para sistemas web de la OBAS

Haciendo uso del IDE de VisualStudio se crearon dos proyectos de tipo Analizador Roslyn llamados ObasAnalyzerVB enfocado en la arquitectura actual de los sistemas web de la OBAS que hace uso de SOA y WebForms en el lenguaje VB.NET; y "ObasAnalyzerCSharp" enfocado en la nueva arquitectura que emplea SOA y MVC en lenguaje C#. En estos proyectos que se incluyeron las clases de diagnóstico encargadas de validar las reglas que permiten detectar la presencia de los *bad smells* listados en la **Tabla 2**.

A continuación, se presenta y detalla un ejemplo de una de las clases de diagnóstico que forman parte del proyecto:

5.3.3.1 Ejemplo de validación del *bad smell* Controlador Cerebral

Definición de la clase

Para la detección del *Bad Smell* "Controlador Cerebral (Brain Controller)" se creó la regla "RA07001 - El Controlador no puede sobrepasar el límite definido para el control de flujo" y para codificar el Analizador Roslyn correspondiente se creó un archivo llamado "ControladorCerebralAnalyzer.cs". Este archivo contiene una clase pública de nombre "ControladorCerebralAnalyzer" que hereda de la clase "DiagnosticAnalyzer" y tiene un atributo que especifica el lenguaje que puede analizar.

En este ejemplo se especifica que puede evaluar código de tipo CSharp (C#).

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
public class ControladorCerebralAnalyzer : DiagnosticAnalyzer

…

end class
```

Definición y registro de descriptores

Dentro de la clase se especifican los descriptores de todas las reglas que serán evaluadas en el analizador. Posteriormente se crea un arreglo con los diagnósticos soportados y se registra en el contexto cada tipo de sintaxis que se debe evaluar.

En este ejemplo se crea la categoría “OBAS-SmellArquitectura-MVC”. En el descriptor “Regla001ControladorCerebral” se establece el identificador “RA07001”, el título “RA07-001: Control de flujo muy grande”, el mensaje “El Controlador sobre pasa el límite definido para el control de flujo”, la severidad “Error” y la opción para que esté habilitado por defecto.

```
internal const string Category = "OBAS-SmellArquitectura-MVC";

internal static readonly DiagnosticDescriptor Regla001ControladorCerebral = new
DiagnosticDescriptor(
        "RA07001",                                                    //Identificador
        "RA07-001: Control de flujo muy grande",                      //Título
        "El Controlador sobre pasa el límite definido para el control de flujo.",
//Mensaje
        Category,                                                     //Categoria
        DiagnosticSeverity.Error,                                     //Severidad
        true); //Está habilitado por defecto

public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics {
get                                                                   {
        return ImmutableArray.Create(Regla001ControladorCerebral); } }


public        override        void        Initialize(AnalysisContext        context)
    {
        // Registra el tipo de sintaxis que se analiza para cada regla y el método que
```

Finalmente se registra la sintaxis “SyntaxKind.ClassDeclaration” para que el analizador pueda analizar todas las clases que estén presentes en el contexto de análisis y se define el método que será encargado de ejecutar el análisis.

Definición de métodos de análisis

Los métodos encargados del análisis reciben como parámetro la sintaxis registrada en el contexto y proceden a evaluar la lógica definida para determinar si existe una violación de la regla que está siendo analizada.

En este ejemplo, se recibe el parámetro de tipo “SyntaxKind.ClassDeclaration” para analizar clases completas. En el primer paso se obtiene la definición de la clase, se extrae el nombre y se identifica si es una clase cuyo nombre contiene la cadena de texto que identifica la nomenclatura especificada para un Controlador dentro de la definición de constantes.

Si es un controlador, se revisan todos los métodos que son parte de la clase y que no corresponden a constructores o sobrecargas de otros métodos. Posteriormente se cuenta la cantidad de veces que aparecen controladores de flujo (If, Switch, For, ForEach, While, Do, Catch y Expresiones condicionales). Si el conteo es superior al límite definido en las constantes, se crea un diagnóstico con los datos del descriptor y se reporta la ubicación de la clase que contiene el código.

```
private void AnalyzeRegla001ControladorCerebral(SyntaxNodeAnalysisContext context)
    {
        // Obtiene la declaración de la clase
        var classDeclaration = (ClassDeclarationSyntax)context.Node;

        // Obtiene nombre de la clase
        var nombreClase = classDeclaration.Identifier.ValueText;

        // Revisa si el nombre de la clase contenedora contiene la cadena "controller"
        if (nombreClase.ToLower().Contains(Constantes.nomenclaturaControlador))
        {
            // Revisa la complejidad de cada método
            int complejidad = 0;
…
```

```
…
            foreach (var member in classDeclaration.Members)
            {
              MethodDeclarationSyntax metodo;

              // Revisa métodos públicos que no son constructores o sobrecargas
              if (member is MethodDeclarationSyntax)
              {
                metodo = (MethodDeclarationSyntax)member;

                if (metodo.Modifiers.Any(SyntaxKind.PublicKeyword) &&
                  !metodo.Modifiers.Any(SyntaxKind.OverrideKeyword) &&
                  metodo.Identifier.ValueText != classDeclaration.Identifier.ValueText)
                {
                  // Se cuentan los controladores de flujo
                  complejidad += metodo.DescendantNodes().Count(n => n is
IfStatementSyntax || n is SwitchStatementSyntax || n is ForStatementSyntax || n is
ForEachStatementSyntax || n is WhileStatementSyntax || n is DoStatementSyntax || n is
CatchClauseSyntax || n is ConditionalExpressionSyntax);
                }
              }
            }

            // Genera reporte si la complejidad calculada es mayor a lo definido en las
            constantes
            if (complejidad > Constantes.limiteControlFlujo)
            {
              var diagnostic = Diagnostic.Create(Regla001ControladorCerebral,
              classDeclaration.Identifier.GetLocation(),
              classDeclaration.Identifier.ValueText);

              context.ReportDiagnostic(diagnostic);
            }
          }
        }
```

El resultado del análisis

En la **Figura 6** se puede observar un ejemplo de cómo se notificaría al usuario en caso de detectar que se incumple con las reglas definidas y revisadas en el analizador.

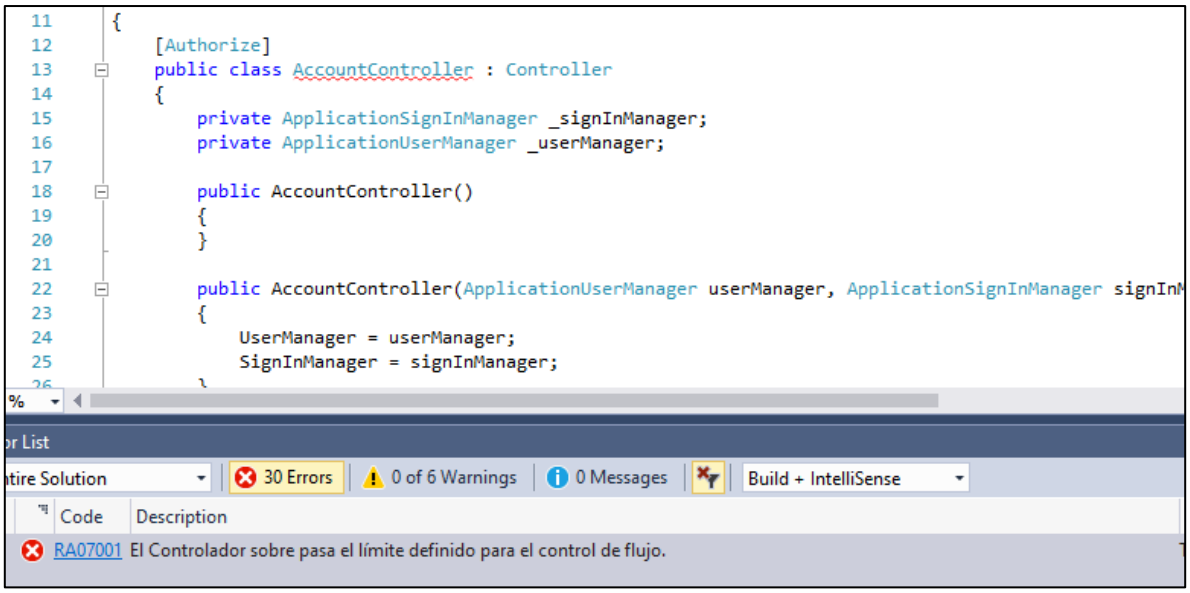

Figura 6 - Ejemplo de resultado de análisis

5.4 Aplicabilidad y pertinencia de la herramienta

Para finalizar la investigación, se evaluó que la herramienta desarrollada pueda ser utilizada en el contexto de los sistemas web desarrollados en la OBAS. Tomando en cuenta que el concepto de aplicabilidad se refiere a algo que puede o debe ser usado [12] y que la pertinencia es la cualidad de algo de corresponderse totalmente con su contexto [13]; se enfocó la evaluación de la herramienta en obtener la percepción de los desarrolladores involucrados en sistemas web de la OBAS luego de utilizar la herramienta para detectar *bad smells,* consultándoles acerca de su eficacia, eficiencia y el posible uso a futuro.

Se entablaron reuniones con los desarrolladores involucrados en la construcción de aplicaciones web en la OBAS con el objetivo de presentar los conceptos de deuda técnica y de *bad smells* junto con las reglas que se estaban evaluando para detectar la presencia de *bad smells*

arquitectónicos. Se detalló el listado de 17 *bad smells* de la **Tabla 2** junto con el detalle de las 21 reglas definidas en el **Anexo 1** y se establecieron límites con base en juicio experto para cada una de las reglas.

Después de la explicación de las reglas, se acordó con los desarrolladores un periodo de 15 minutos para que pudieran realizar una búsqueda manual de errores en uno de los sistemas con el estilo de arquitectura MVC que estaba en etapa de desarrollo y que era familiar para los participantes. Este tiempo se fijó en 15 minutos con base en los comentarios de los desarrolladores que indicaron que normalmente no realizan ningún tipo de revisión de código por el poco tiempo que tienen para avanzar con sus tareas diarias.

Posterior a esto, se les presentó la herramienta de detección automática desarrollada y se demostró como se ejecutaba un análisis en el mismo sistema en el que se buscaron *bad smells* de manera automática. Se recopilaron los resultados de ambas búsquedas y se analizaron junto con los participantes para comparar tanto la cantidad de bad smells detectados como el tiempo y esfuerzo invertido en detectarlos.

5.4.1 Resultados de búsqueda manual

Para las pruebas de búsqueda manual se le solicitó a cada desarrollador que intentaran encontrar la mayor cantidad de *bad smells* en el código fuente del sistema web llamado Servicios Beca y en el código fuente de las bases de la nueva arquitectura MVC que se encuentra en desarrollo al momento del estudio.

De manera general, los desarrolladores optaron por abrir cada una de las clases existentes en las diferentes capas de los sistemas para recolectar datos de tamaño de clase, tamaño y cantidad de lógica empleada en los métodos (en búsqueda de probables casos de Clase Dios, Repositorio Grande, Controlador Cerebral). Además, realizaron búsqueda de creación de objetos de una capa superior en una clase de una capa inferior (en búsqueda de probables casos de Violación de Arquitectura). En la **Tabla 3** se detalla la incidencia en la búsqueda de *bad smells,* detallando la cantidad de veces en que cada desarrollador encontró un error y especificando cuándo buscó un *bad smell* pero no encontró ningún error. En la última columna se indica la cantidad de veces

que estaba presente el *bad smell* en los proyectos analizados para poder comparar la cantidad de aciertos.

Bad Smell buscado	**Desarrollador**						**Bad Smell Presente**
	I	II	III	IV	V	VI	
07 Controlador Cerebral	-	-	Lo buscó, pero no lo encontró	-	-	-	Había 0 errores
11 Repositorio Grande	Lo encontró 2 veces	-	-	-	-	Lo buscó, pero no lo encontró	Había 8 errores
15 Violación de Arquitectura	Lo buscó, pero no lo encontró	Lo buscó, pero no lo encontró	-	-	Lo buscó, pero no lo encontró	Lo buscó, pero no lo encontró	Había 0 errores
16 Clase Dios	Lo encontró 3 veces	Lo encontró 5 veces	Lo encontró 5 veces	Lo encontró 6 veces	Lo encontró 4 veces	Lo encontró 4 veces	Había 23 errores

Tabla 3 - *Bad smells* detectados manualmente

Se puede observar que tanto el *bad smell* llamado "Controlador Cerebral" como el llamado "Violación de arquitectura" fueron buscados por los desarrolladores y no se encontraron, lo cual está bien ya que no había clases que incumplieran con las reglas definidas para cada uno. Sin embargo, esta búsqueda consumió tiempo y esfuerzo que en un contexto real pudo ser utilizado en otras tareas.

En el caso del *bad smell* llamado "Repositorio Grande" fue encontrado en dos oportunidades por un desarrollador y buscado por otro que no logró encontrar ninguna incidencia. Por otra parte, el *bad smell* llamado "Case Dios" fue el que más se buscó y tuvo mayor cantidad de errores detectados. No obstante, en ambos casos había presentes muchos más errores que no fueron encontrados por los desarrolladores. Lo que implica que el código hubiera seguido presentando infracciones a las reglas definidas a pesar del tiempo y esfuerzo invertido.

5.4.2 Resultados de búsqueda automática

Finalizado el tiempo de la búsqueda manual, se introdujo el concepto de analizadores estáticos de código y se presentó a los participantes la herramienta de detección automática desarrollada. Se ejecutó un análisis de código con las reglas programadas que finalizó en promedio en 40 segundos y encontró 44 errores en el código fuente, el detalle de la incidencia de *bad smells* según la detección automática se detalla en la **Tabla 4**.

Bad Smell	Cantidad de detecciones
05 – Controlador es Vista	1
11 – Repositorio Grande	8
12 – Repositorio Laborioso	10
14 – Interface Ambigua	2
16 – Clase Dios	23
Total	44

Tabla 4 - *Bad smells* detectados automáticamente

Se puede observar que los *bad smells* llamados "Controlador es Vista", "Repositorio Laborioso" e "Interface ambigua" estaban presentes en el código fuente y no fueron ni siquiera buscados por los desarrolladores en la búsqueda manual. Esto demuestra que realizar búsquedas manuales en código fuente es algo que, además de consumir tiempo y esfuerzo, puede estar limitado por la interpretación de las reglas que hace cada persona.

En última instancia, se presentó la funcionalidad de detección en tiempo de codificación que provee la herramienta. Es decir, se demostró cómo la herramienta es capaz de detectar errores en el código conforme el desarrollador va ingresando código nuevo, ahorrando la necesidad de estar revisando de manera manual cada elemento programado.

5.4.3 Resultados evaluación de aplicabilidad y pertinencia

Finalizada la presentación de la herramienta, se les solicitó a los desarrolladores que llenaran la encuesta de evaluación de aplicabilidad y pertinencia (ver **Anexo 2**) para poder obtener la percepción sobre el uso de esta. Se recopilaron y analizaron las respuestas (ver **Anexo 3**) para determinar si la herramienta sería útil en el contexto probado.

En la primera parte de la encuesta se consultó a los participantes por los años de experiencia en desarrollo de sistemas, de los 6 participantes hay 4 que tienen más de 10 años de experiencia, 1 que indica tener entre 5 y 10 años y 1 con menos de 5 años de experiencia. Se puede observar en la **Figura 7** que el grupo de desarrolladores involucrados en los sistemas web de la OBAS cuenta con amplia experiencia, por lo que se espera que su nivel de experticia sea un factor positivo a la hora de ubicar y corregir errores en código y en la implementación de la arquitectura.

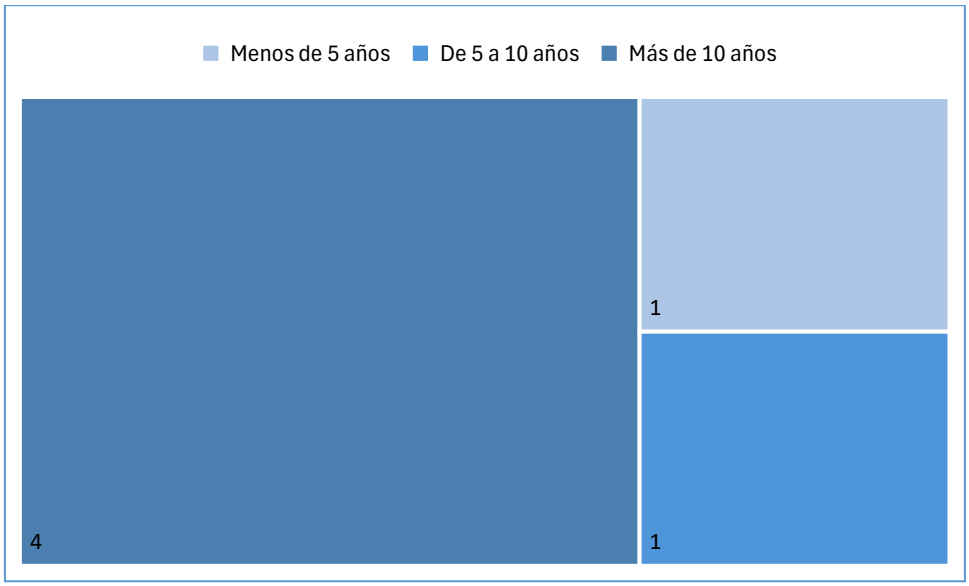

Figura 7 - Gráfico de años de experiencia de desarrolladores

5.4.3.1 Pertinencia

Para determinar la pertinencia de la herramienta desarrollada, se consultó al grupo de participantes por la frecuencia con que se revisa el código en búsqueda de posibles errores en la implementación de la arquitectura y sólo 2 de estos indican revisar el código de manera periódica con una frecuencia trimestral o semestral. El resto de los participantes aceptaron no realizar ningún tipo de revisión; como se observa en la **Figura 8** esto representa un porcentaje muy alto. Además, se les consultó si tenían experiencia en el uso de herramientas de análisis de

código y sólo 1 participante responde haber utilizado el compilador de Visual Studio para ejecutar este tipo de análisis.

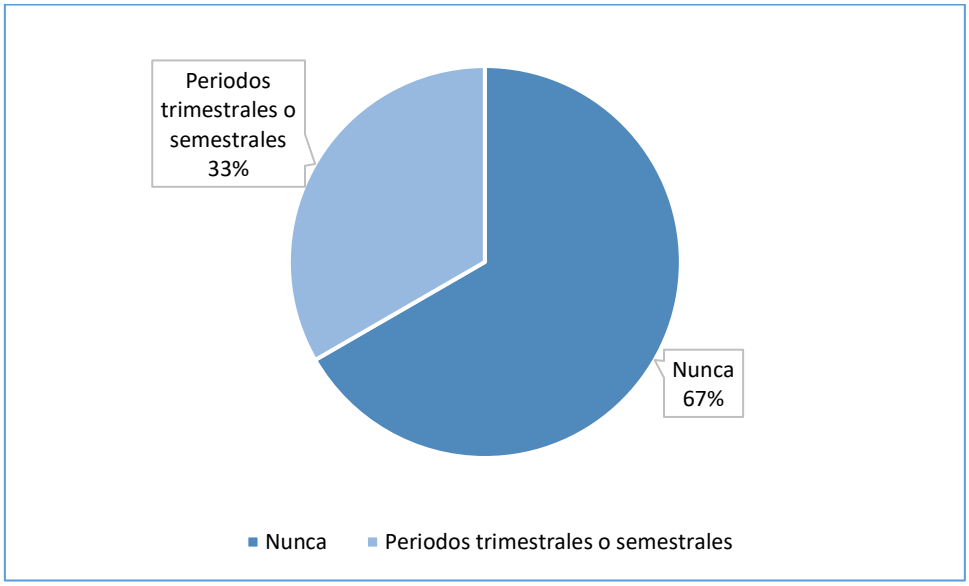

Figura 8 - Gráfico de frecuencia de revisión de posibles errores

Se puede entender de estos resultados que no existe un procedimiento formal para la revisión de código y que los errores se comprueban únicamente con base en las reglas predeterminadas de la plataforma de compilación de .Net. Esto evidencia la pertinencia de una herramienta que permita analizar el código para validar que la arquitectura esté siendo implementada de manera adecuada de acuerdo con su definición.

5.4.3.2 Aplicabilidad

Para validar el funcionamiento de la herramienta presentada se solicitó llenar, con escalas de Likert [35], una serie de preguntas relacionadas con los siguientes aspectos: (1) Facilidad de uso: referente a qué tan sencillo es realizar un análisis de código. (2) Satisfacción del rendimiento: referente al tiempo que tardó en terminar el análisis de código. (3) Efectividad en la detección: referente a la cantidad y calidad de detecciones. (4) Rapidez de ejecución de la

herramienta: referente al tiempo que tardaron en abrir y ejecutar la herramienta. En la **Figura 9** se puede observar que los desarrolladores brindan una valoración positiva en todos los aspectos evaluados. De los 6 participantes, 2 indican que la herramienta es fácil de usar y 3 que es muy fácil. Además, 5 participantes valoran que la herramienta tiene un tiempo de ejecución muy rápido para la detección de posibles errores. En cuanto a los aspectos de rendimiento y efectividad en la detección, se puede notar que hay menor aceptación ya que 2 participantes indican estar algo satisfechos con el rendimiento y 2 de ellos consideran que la herramienta es algo efectiva. Sin embargo, la valoración sigue siendo positiva ya que no hay aspectos en los que se haya indicado estar en una posición de desacuerdo o de insatisfacción.

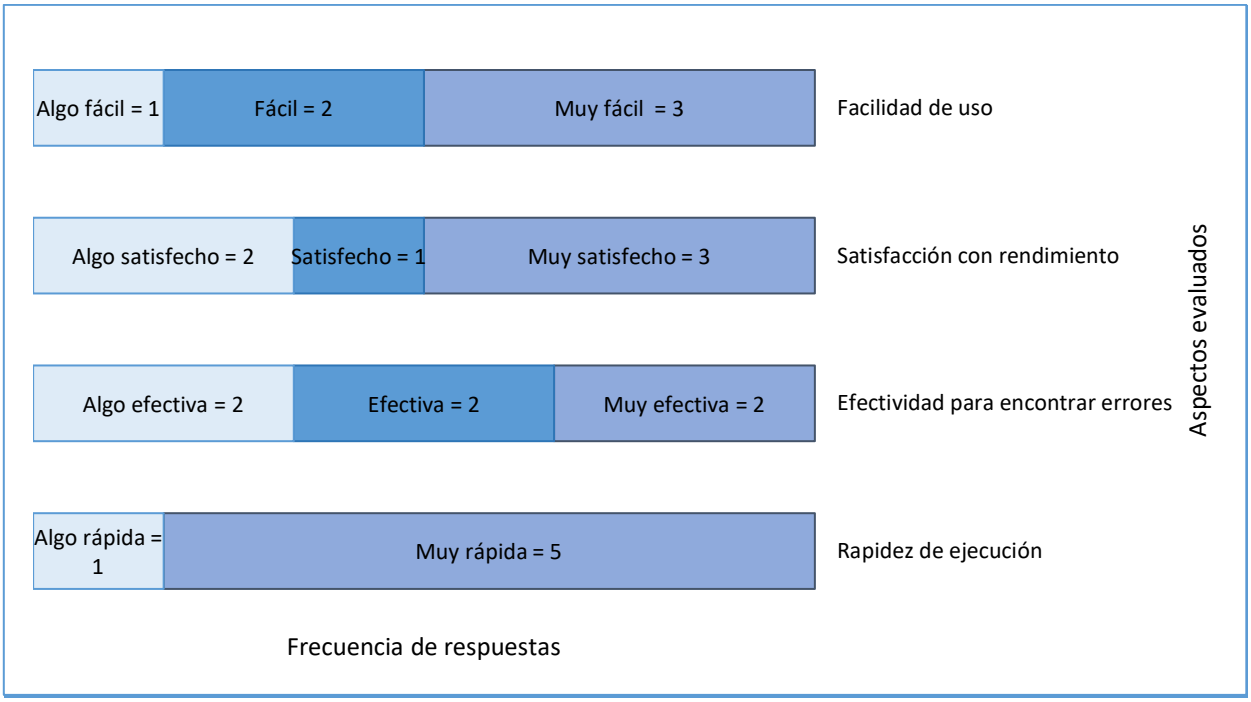

Figura 9 - Gráfico valoración de funcionamiento de herramienta

Además, se incluyó una pregunta para saber si los desarrolladores estarían o no dispuestos a utilizar la herramienta en evaluación y todos contestaron que sí la utilizarían. Con estos resultados se puede determinar que la herramienta sí es aplicable en el contexto de los sistemas web de la OBAS.

5.4.3.3 Ventajas y desventajas

En la encuesta también se solicitó a los participantes indicar cuáles ventajas o beneficios y desventajas o inconvenientes observan, y se les brindó la posibilidad de agregar sugerencias para mejorar la herramienta desarrollada.

Como resultados de la encuesta, se obtuvo que la principal ventaja detectada (seleccionada por 4 de los participantes) es que la herramienta ayudaría a reducir la necesidad de retrabajo; seguido de una mejora en la calidad del código (seleccionada por 3 participantes) y finalmente una mejora de rendimiento en las aplicaciones desarrolladas (seleccionada por 2 participantes). En cuanto a las desventajas, cabe destacar que 3 de los participantes marcaron que no encuentran desventajas mientras que 2 participantes tienen preocupaciones con aumento en los tiempos de entrega y 2 de ellos indican que se pueden presentar falsos positivos en la detección de posibles errores.

Con respecto a las sugerencias y recomendaciones para mejorar la herramienta, se obtuvo retroalimentación importante que apunta a mejorar la definición de las reglas y límites evaluados de manera que los desarrolladores que utilicen la herramienta conozcan previamente los estándares de codificación de la OBAS. Además, se solicita que a futuro se implementen más controles y que los límites puedan ser configurables mediante parámetros o dependencias inyectables de forma que cada proyecto pueda contar con reglas y límites que se puedan ajustar al contexto específico de cada sistema.

CAPÍTULO 6 - CONCLUSIONES

En esta investigación, se diagnosticó la existencia de deuda técnica en los sistemas web de la OBAS, identificando tanto sus causas como sus efectos, los cuales generan principalmente retrasos en los tiempos de entrega. Se explicaron los conceptos de deuda técnica y de *bad smells* al equipo de desarrollo, logrando recopilar un conjunto de 17 de *bad smells* arquitectónicos junto con 21 reglas para detectar su presencia en la implementación de la arquitectura modelo-vista-controlador en los sistemas web de la OBAS. A través de analizadores Roslyn, se construyó una herramienta capaz de analizar automáticamente el código fuente para validar estas 21 reglas. Los resultados del uso de la herramienta desarrollada indicaron una mejora significativa en la detección de *bad smells* arquitectónicos, optimizando así los procesos de desarrollo en la OBAS. Finalmente, la herramienta desarrollada fue evaluada por el equipo de desarrolladores de la OBAS, quienes confirmaron la aplicabilidad y pertinencia de esta herramienta, cumpliendo así el objetivo general de esta investigación: Evaluar el uso de una herramienta de detección de *BadSmells* arquitectónicos en los sistemas web de la Oficina de Becas y Atención Socioeconómica.

A continuación, se amplían detalles sobre las principales conclusiones de la presente investigación.

En la OBAS se desarrollan e implementan sistemas web para captar información importante para la gestión de becas y beneficios complementarios de una gran cantidad de estudiantes. Sin embargo, hasta el momento no se había utilizado ningún tipo de herramienta para el control y gestión de deuda técnica. Por ello fue necesario entablar reuniones para explicar a los desarrolladores el concepto de deuda técnica y dejar claro las posibles consecuencias de no controlarla de manera efectiva. Además, fue necesario presentar el concepto de *bad smell* como un indicador o síntoma de que los sistemas tienen problemas de diseño o implementación. Se aplicó un diagnóstico de deuda técnica en la OBAS y los resultados demostraron que los sistemas que se desarrollan en esta unidad están acumulando un grado de deuda técnica que lleva a los desarrolladores a tener que retrabajar en funcionalidades, lo que aumenta el esfuerzo y afecta negativamente los tiempos de entrega.

Con base en esto, se recomienda que futuras investigaciones relacionadas con el diagnóstico y manejo de deuda técnica tomen en cuenta estrategias efectivas para la identificación y gestión temprana de la deuda técnica, creando conciencia sobre las consecuencias a largo plazo de no realizar un control adecuado.

Por otra parte, quedó demostrada la importancia de establecer una definición clara y concisa de la arquitectura de un sistema, incluyendo la documentación de reglas y pautas de manera exhaustiva para reducir la probabilidad de cometer errores durante las etapas de codificación y pruebas de los sistemas. Sumado a esto, se evidencia la importancia de tener mecanismos de análisis, diagnóstico y control de la deuda técnica para evitar situaciones que afecten la eficiencia operativa, la calidad de los sistemas desarrollados y la entrega oportuna de las soluciones tecnológicas, entre otros posibles problemas

En esta investigación se logró recopilar una serie de *bad smells* que afectan el tipo de arquitectura definido para los sistemas web de la OBAS y un conjunto de reglas que permiten identificar cuando está presente cada uno de estos. Este listado podría servir de base para mejorar la identificación de posibles errores en la implementación de la arquitectura MVC y con esto incentivar que se aborde más el tema.

Asimismo, se lograron identificar varias herramientas de análisis estático con capacidad de identificar *bad smells* en el código fuente; sin embargo, ninguna de las herramientas compatibles con los lenguajes de programación VisualBasic.net y C# documenta la posibilidad de detectar *bad smells* a nivel de arquitectura MVC. Debido a esto se investigó sobre herramientas con capacidad de crear reglas personalizadas y realizar detecciones en el código fuente de los sistemas.

Se da respuesta a la pregunta de investigación "¿Cuál herramienta de detección de *bad smells* arquitectónicos es pertinente aplicar a los sistemas web desarrollados en la Oficina de Becas y Atención Socioeconómica?" concluyendo que la plataforma de *Roslyn Analyzer* ofrece las funcionalidades deseadas ya que permite crear reglas personalizadas para detectar *bad smells* en la implementación de la arquitectura y además se integra de manera sencilla al entorno de

desarrollo, apoyando al proceso de construcción de sistemas de calidad permitiendo mejorar la calidad del código y asegurar que se respetan las pautas establecidas en la arquitectura definida.

Se utilizó esta plataforma de compilación de .Net (*Roslyn Analyzer*) para desarrollar una herramienta de detección automática de los *bad smells* arquitectónicos relacionados con la arquitectura MVC y se hizo uso de las funcionalidades de análisis de código y reporte de errores de la plataforma. Luego de realizar pruebas en el contexto de la OBAS, se ratificó que el tiempo de búsqueda manual de errores es mucho mayor que el tiempo de ejecución de la búsqueda automática. Además, se demostró que al incorporar la herramienta al IDE de desarrollo VisualStudio se pueden detectar errores en tiempo real conforme se va ingresando el código nuevo, lo que brinda la opción a los desarrolladores de crear código de mejor calidad reduciendo el esfuerzo para la detección de posibles errores.

Durante el proceso de construcción de la herramienta de detección fue fundamental el aporte de los funcionarios expertos en el desarrollo de sistemas para lograr la correcta adaptación de la herramienta al contexto de la OBAS. Esto también permitió a los desarrolladores entender el funcionamiento de la plataforma de compilación y las capacidades que tiene para mejorar código ingresado.

Gracias a esto, la evaluación de pertinencia y aplicabilidad de la herramienta dio resultados positivos y las personas involucradas en el desarrollo de aplicaciones web de la OBAS quedaron dispuestas a incorporar la herramienta para analizar el código de los sistemas ya desarrollados y en proceso de construcción.

Basado en esta experiencia, se recomienda que futuras investigaciones enfocadas en la aplicación de herramientas de detección de errores inicien profundizando en la estandarización y documentación de las reglas y pautas específicas para la implementación de la arquitectura de sistemas elegida. Además, para asegurar el éxito de la aplicación de este tipo de herramientas, se recomienda la colaboración continua con los equipos de desarrollo de software al momento de establecer los criterios de incumplimiento de las reglas.

En contextos similares al de la OBAS, donde no se utiliza ninguna herramienta de análisis de código, implementar una herramienta de detección de *bad smells* arquitectónicos no sólo ayuda

a mejorar la calidad de software colaborando con la prevención y atención de la deuda técnica, sino que sirve como punto clave para mejorar el nivel de madurez las áreas de desarrollo de software ya que esto representa aplicar una tecnología novedosa, pionera y con posibilidad de crecimiento en cuanto a las reglas que puede detectar.

Se recomienda que para futuras aplicaciones de la herramienta desarrollada en la OBAS (así como cualquier otro desarrollador que la quiera utilizar) se establezcan pautas y reglas formales pertinentes a la arquitectura definida. Por ejemplo, para definir si una clase se comporta con una "Clase Dios" es necesario que sea definido el límite para la cantidad de líneas de código que puede tener una clase. Esto se puede realizar para cada conjunto de *bad smell*-regla presentada en esta investigación.

6.1 Limitaciones

La arquitectura definida para los nuevos desarrollos de sistemas web de la OBAS fue presentada en el año 2022 y se brindó una capacitación al equipo de desarrollo en el año 2023. Sin embargo, al momento de ejecutar las pruebas de la herramienta de detección de *bad smells* no se contó con sistemas en producción que utilizaran la arquitectura nueva.

Por este motivo se realizaron pruebas en sistemas que se encuentran aún en etapa de desarrollo y sistemas que han sido desarrollados con la arquitectura SOA-WebForms, en específico en las capas implementadas con SOA debido a su parecido con la arquitectura SOA-MVC. Para lograr estas pruebas, fue necesario crear una herramienta enfocada en detección de posibles errores en lenguaje VB.NET y otra enfocada en lenguaje C#.

A pesar de estas limitaciones para probar la herramienta en sistemas web en producción, se obtuvieron resultados que indican que el uso de la herramienta de detección de *bad smells* a nivel de arquitectura puede ayudar a prevenir futuros problemas en el desarrollo y puesta a producción de los sistemas que actualmente se están construyendo.

Adicionalmente, se identificaron otras amenazas a la validez de la investigación. En primer lugar, el desarrollador experto que proporcionó retroalimentación inicial sobre los primeros 5

bad smells también participó en la evaluación final, lo que pudo representar un sesgo en la objetividad de la evaluación, ya que su familiaridad con la herramienta pudo influir en su juicio.

Segundo, a pesar de que el equipo completo de desarrolladores de la OBAS participó en las evaluaciones, se puede considerar que es un grupo pequeño y con características muy similares. En un contexto con equipos de desarrollo más grande y con características más variadas se pudo haber obtenido opiniones y evaluaciones con mayor diversidad.

Finalmente, los resultados de la evaluación con los desarrolladores podrían haber sido afectados por varios factores: (1) la falta de experiencia en el uso de herramientas de este tipo, lo que podría haber limitado su capacidad de explotar todo el potencial de la herramienta; (2) el hecho de que los desarrolladores analizaron código con el que ya estaban familiarizados, lo que puede haber llevado a que omitieran revisar ciertos aspectos al asumir que estaban bien implementados; y (3) el tiempo limitado que se destinó para realizar la búsqueda manual de *bad smells* pudo afectar la profundidad de su análisis y con ello los resultados de la comparación entre la búsqueda manual y la búsqueda automática.

6.2 Trabajo futuro

En términos generales, se evidenció que hay pocos estudios que se enfocan en *bad smells* a nivel de arquitectura MVC, por lo que es necesario que se amplie la investigación relacionada a este tipo de arquitecturas.

La herramienta desarrollada puede ser utilizada como base para agregar más reglas y mejorar la parametrización de los límites definidos con el objetivo de mejorar la capacidad de adaptación a diferentes contextos. En este proceso se debe incluir la participación tanto de expertos académicos como expertos en el ámbito laboral para facilitar la creación de reglas que tengan buen fundamento y sean bien aceptadas.

Se sugiere implementar y evaluar la herramienta desarrollada en otros contextos que utilicen arquitecturas de sistemas web similares a la de la OBAS. Esto permitiría obtener una variedad de opiniones y recopilar insumos valiosos que contribuyan a mejorar y optimizar la herramienta.

Finalmente, se recomienda a las unidades o departamentos de organizaciones y empresas que puedan investigar y aplicar herramientas para diagnosticar y controlar la deuda técnica que se genera y acumula durante los procesos de desarrollo de sistemas y aplicaciones con el objetivo de mejorar sus procesos y asegurar la entrega productos de calidad en tiempos oportunos.

REFERENCIAS

[1] A. Hernández Trasobares, «Los sistemas de información: Evolución y Desarrollo,» *Proyecto social: Revista de relaciones laborales,* nº 10-11, pp. 149-165, 2003.

[2] M. I. Murillo, A. Pacheco, G. López, G. Marín y J. Guzmán, «Common Causes and Effects of Technical Debt in Costa Rica: InsighTD Survey Replication,» *2021 XLVII Latin American Computing Conference (CLEI),* pp. 1-9, 2021.

[3] W. Cunningham, «The WyCash portfolio management system,» *SIGPLAN OOPS Mess.,* vol. 4, nº 2, p. 29.30, 04 1992.

[4] J. Garcia, D. Popescu, G. Edwards y N. Medvidovic, «Toward a Catalogue of Architectural Bad Smells,» de *Lecture Notes in Computer Science*, vol. 5581, Berlin, Springer, 2009, pp. 146-162.

[5] H. Mumtaz, P. Singh y K. Blincoe, «A systematic mapping study on architectural smells detection,» *The Journal of Systems & Software,* vol. 173, 2021.

[6] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc y N. Moha, «A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems,» *Softw Pract Exper,* vol. 49, pp. 3-39, 2019.

[7] P. Velasco Elizondo, L. Castañeada Calvillo, A. García Fernández y S. Vazquez Reyes, «Caracterización y Detección Automática de Bad Smells MVC,» *Revista Ibérica de Sistemas de Tecnologías de Información,* nº 26, pp. 54-67, 2018.

[8] E. Fernandes, O. Johnatan, G. Vale, T. Paiva y E. Figueiredo, «A Review-based Comparative Study of Bad Smell,» de *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, New York, Association for Computing Machinery, 2016, pp. 18-30.

[9] M. Lippert y S. Roock, Refactoring in Large Software Projects: Performing Complex Restructurings Successfully, John Wiley & Sons, 2006.

[10] B. Sousa, P. Souza, E. Fernandes, K. Ferreira y M. Bigonha, «FindSmells: Flexible Composition of Bad Smell Detection Strategies,» *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC),* pp. 360-363, 2017.

[11] Oficina de Becas y Atención Socioeconómica, «Oficina de Becas y Atención Socioeconómica,» 2022. [En línea]. Available: becas.ucr.ac.cr. [Último acceso: 20 04 2021].

[12] J. Pérez Porto y A. Gardey, «Aplicabilidad - Qué es, definición y concepto,» 20 05 2022. [En línea]. Available: https://definicion.de/aplicabilidad/. [Último acceso: 12 2023].

[13] Equipo editorial, Etecé, «Pertinencia,» 2 02 2022. [En línea]. Available: https://concepto.de/pertinencia/. [Último acceso: 12 2023].

[14] P. Clements, F. Bachmann, L. Bass, D. Garlan, I. James, R. Little, P. Merson, R. Nord y J. Stafford, Documenting Software Architectures: Views and Beyond, Segunda ed., Boston: Pearson Education, Inc., 2011.

[15] L. Haorongbam, R. Nagpal y R. Sehgal, «Service Oriented Architecture(SOA): A Literature Review on the Maintainability, Approaches and Design Process,» *12th International Conference on Cloud Computing, Data Science & Engineering (Confluence),* pp. 647-652, 2022.

[16] M. Richards, Software Architecture Patterns, Primera ed., 0'Reilly Media Inc., 2015.

[17] R. Miller, C# for Artists: The Art, Philosophy, and Science of Object-oriented Programming, Pulp Free Press, 2008.

[18] M. Fowler, Refactoring - Improving the Design of Existing Code, Addison-Wesley, 1999.

[19] M. Aniche, G. Bavota, C. Treude, A. Van Deursen y M. A. Gerosa, «A Validated Set of Smells in Model-View-Controller Architectures,» *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME),* pp. 233-243, 2016.

[20] M. Saunders y P. Tosey, «The Layers of Research Design,» *RAPPORT,* 2013.

[21] A. Hevner, «A Three Cycle View of Design Science Research,» *Scandinavian Journal of Information Systems,* vol. 19, pp. 87-92, 2007.

[22] P. Arévalo Chávez, J. Cruz Cárdenas, C. Guevara Maldonado, A. Palacio Fierro, S. Bonilla Bedoya, A. Estrella Bastidas, J. Guadalupe Lanas, M. Zapata Rodríguez, J. Jadán Guerrero, H. Arias Flores y C. Ramos Galarza, Actualización en metodología de la investigación científica, Quito, Ecuador: Universidad Tecnológica Indoamérica, 2020.

[23] M. I. Murillo Quintana, *Instrumento para el diagnóstico de la gestión de la deuda técnica en organizaciones que desarrollan software,* 2024.

[24] «Structure101 - Software Architecture Development Environment (ADE),» [En línea]. Available: https://structure101.com/. [Último acceso: 06 2023].

[25] «NDepend,» [En línea]. Available: https://www.ndepend.com/. [Último acceso: 06 2023].

[26] N. Kumar, «Software Architecture Validation Methods, Tools Support and Case Studies,» de *Emerging Research in Computing, Information, Communication and Applications*, 2015, p. 335–345.

[27] «ReSharper,» [En línea]. Available: https://www.jetbrains.com/resharper/. [Último acceso: 03 2023].

[28] U. Azadi, F. Arcelli Fontana y D. Taibi, «Architectural Smells Detected by Tools: a Catalogue Proposal,» *2019 IEEE/ACM International Conference on Technical Debt (TechDebt),* pp. 88-97, 2019.

[29] J. Garcia, D. Popescu, G. Edwards y N. Medvidovic, «Identifying Architectural Bad Smells,» de *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 255-258.

[30] «SonarQube,» [En línea]. Available: https://www.sonarsource.com/products/sonarqube/. [Último acceso: 03 2023].

[31] W. de Kort, «Managing Technical Debt,» de *DevOps on the Microsoft Stack*, 2016, pp. 137-160.

[32] Microsoft Learn, «Overview of source code analysis,» [En línea]. Available: https://learn.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview?view=vs-2022. [Último acceso: 12 2023].

[33] A. Martini, F. A. Montana, A. Biaggi y R. Roveda, «Identifying and Prioritizing Architectural Debt Through Architectural Smells: A Case Study in a Large Software Company,» de *ECSA 2018: Software Architecture*, 2018, pp. 320-335.

[34] Microsoft Learn, «Class '<classname>' cannot inherit from itself: <message>,» 15 09 2021. [En línea]. Available: https://learn.microsoft.com/en-us/dotnet/visual-basic/misc/bc30257. [Último acceso: 12 2023].

[35] R. Likert, «A technique for the measurement of attitudes,» *Archives of Psychology,* 1932.

ANEXOS

Anexo 1. Reglas para detectar BadSmells

Bad Smell	Nombre	Reglas	Descripción de la regla
1	Modelo es Vista		
		RA01001	No se permite el uso de etiquetas HTML en archivos
2	Modelo es Controlador		
		RA02001	No se permite el uso de etiquetas [HttpPost] y [HttpGet]
3	Vista es Modelo		
		RA03001	No se permite la presencia de comandos comunes de SQL
		RA03002	No se permite la presencia de referencias a librerías de acceso a datos
4	Vista es Controlador		
		RA04001	No se permite el uso de etiquetas [HttpPost] y [HttpGet]
5	Controlador es Vista		
		RA05001	No se permite el uso de etiquetas HTML en archivos
6	Controlador es Modelo		
		RA06001	No se permite la presencia de comandos comunes de SQL
		RA06002	No se permite la presencia de referencias a librerías de acceso a datos
7	Controlador Cerebral (Brain Controller)		
		RA07001	El Controlador no puede sobrepasar el límite definido para el control de flujo
8	Controlador Promiscuo (Promiscuos Controller)		

Bad Smell	Nombre	Reglas	Descripción de la regla
		RA08001	El Controlador no puede sobrepasar el límite definido para la cantidad de acciones y rutas que ofrece
9	Servicio Entrometido (Meddling Service)		
		RA09001	No se permite la presencia de comandos comunes de SQL.
		RA09002	No se permite la presencia de referencias a librerías de acceso a datos
10	Repositorio Cerebral (Brain Repository)		
		RA10001	El Repositorio no puede contener comandos SQL con lógica que sobrepase el límite definido
11	Repositorio Grande (Fat Repository)		
		RA11001	El Repositorio no puede superar el límite establecido de entidades que gestiona.
12	Repositorio Laborioso (Laborious Repository Method)		
		RA12001	El Repositorio no puede tener métodos que sobrepasen la cantidad máxima permitida para ejecución de acciones sobre la base de datos.
13	Abstracción sin desacople (Abstraction without Decoupling)		
		RA13001	Cuando una interface es utilizada de manera abstracta e invocada como un tipo concreto en el mismo método

Bad Smell	Nombre	Reglas	Descripción de la regla
14	Interface ambigua (Ambiguous interface)		
		RA14001	La interface no puede sobrepasar el límite establecido para la cantidad de métodos que define.
15	Violación de la definición de la arquitectura (Architecture Violation)		
		RA15001	La capa de servicios no puede referenciar objetos de la capa de acceso a datos
		RA15002	La capa de lógica de negocios no puede referenciar objetos de la capa de servicios
		RA15003	La capa de acceso a datos no puede referenciar objetos de la capa de lógica de negocios o la capa de servicios
16	Clase Dios (God Class / Component)		
		RA16001	La clase sobrepasa el límite de líneas de código establecido.

Anexo 2. Encuesta de aplicabilidad y pertinencia

Valoración de herramienta de detección de *Bad Smells* Arquitectónicos

Un *bad smell* es un síntoma de un posible problema en el diseño o codificación de un sistema, cuando estos se encuentran en un nivel superior de diseño, es decir, en la arquitectura del sistema, son llamados *bad smells* arquitectónicos. La herramienta que usted utilizó tiene la función de detectar la presencia de estos síntomas en la arquitectura de los sistemas web desarrollados en la OBAS.

La presente encuesta tiene como objetivo valorar aspectos de eficacia, eficiencia y uso de la herramienta utilizada para determinar su aplicabilidad y pertinencia en los sistemas de la oficina.

* Indica que la pregunta es obligatoria

1. Nombre *

_____

2. Años de experiencia en desarrollo de sistemas *

- ◯ Menos de 5 años
- ◯ De 5 a 10 años
- ◯ Más de 10 años

3. ¿Con qué frecuencia revisa el código ya desarrollado en búsqueda de posibles errores de aplicación de la arquitectura definida? *

- ◯ Diariamente
- ◯ Semanalmente
- ◯ Mensualmente
- ◯ Periodos trimestrales o semestrales
- ◯ Anualmente
- ◯ Nunca

4. ¿Ha utilizado alguna herramienta para análisis de código? *

- ◯ No
- ◯ Sí

4.1. Si indicó que sí en la pregunta anterior, por favor indique cuáles herramientas ha utilizado.

_____

_____

_____

_____

5. ¿Qué tan fácil o difícil le resultó usar la herramienta que se está evaluando? *

Muy fácil	1	2	3	4	5	6	7	Muy difícil
	◯	◯	◯	◯	◯	◯	◯	

6. ¿Qué tan satisfecho o insatisfecho está con el rendimiento de la herramienta que se está evaluando? *

Muy satisfecho	1	2	3	4	5	6	7	Muy insatisfecho
	⬭	⬭	⬭	⬭	⬭	⬭	⬭	

7. ¿Qué tan efectiva o inefectiva cree que es la herramienta se está evaluando para encontrar los errores en el código? *

Muy efectiva	1	2	3	4	5	6	7	Muy inefectiva
	⬭	⬭	⬭	⬭	⬭	⬭	⬭	

8. ¿Qué tan rápido o lento es el tiempo de ejecución de la herramienta que se está evaluando? *

Muy rápido	1	2	3	4	5	6	7	Muy lento
	⬭	⬭	⬭	⬭	⬭	⬭	⬭	

9. ¿Qué ventajas o beneficios cree que tiene el uso de la herramienta que se está evaluando? *

- [ ] Ninguna
- [ ] Ahorro de tiempo
- [ ] Mejora de la calidad
- [ ] Mejora de rendimiento de aplicaciones
- [ ] Reducir la necesidad de retrabajo
- [ ] Otro: _____

10. ¿Qué desventajas o inconvenientes cree que tiene el uso de la herramienta que se está evaluando? *

- [ ] Ninguna
- [ ] Aumento de retrabajo
- [ ] Aumento en los tiempos de entrega
- [ ] Complejidad
- [ ] Falsos negativos
- [ ] Falsos positivos
- [ ] Otro: _____

11. ¿Qué sugerencias o recomendaciones tienes para mejorar la herramienta que se está evaluando?

_____

_____

12. En términos generales, ¿utilizaría la herramienta que se está evaluando? *

- Sí
- No

Anexo 3. Respuestas de encuesta de aplicabilidad y pertinencia

(se omite la respuesta de la pregunta #1 “Nombre” para preservar la privacidad de las personas que responden)

	Participante #					
Pregunta #	**1**	**2**	**3**	**4**	**5**	**6**
2. Años de experiencia en desarrollo de sistemas	Más de 10 años	Más de 10 años	Más de 10 años	Menos de 5 años	De 5 a 10 años	Más de 10 años
3. ¿Con qué frecuencia revisa el código ya desarrollado en búsqueda de posibles errores de aplicación de la arquitectura definida?	Nunca	Nunca	Periodos trimestrales o semestrales	Periodos trimestrales o semestrales	Nunca	Nunca
4. ¿Ha utilizado alguna herramienta para análisis de código?	No	No	No	No	Sí	No
4.1 Si indicó que sí en la pregunta anterior, por favor indique cuáles herramientas ha utilizado.					El compilador de Visual Studio	
5. ¿Qué tan fácil o difícil le resultó usar la herramienta que se está evaluando?	2	1	3	2	1	1

	Participante #					
Pregunta #	**1**	**2**	**3**	**4**	**5**	**6**
6. ¿Qué tan satisfecho o insatisfecho está con el rendimiento de la herramienta que se está evaluando?	3	3	1	1	1	2
¿Qué tan efectiva o inefectiva cree que es la herramienta se está evaluando para encontrar los errores en el código?	2	3	1	1	2	3
¿Qué tan rápido o lento es el tiempo de ejecución de la herramienta que se está evaluando?	1	3	1	1	1	1
¿Qué ventajas o beneficios cree que tiene el uso de la herramienta que se está evaluando?	Mejora de la calidad, Reducir la necesidad de retrabajo	Mejora de la calidad	Mejora de rendimiento de aplicaciones, Reducir la necesidad de retrabajo	Mejora de la calidad, Mejora de rendimiento de aplicaciones	Reducir la necesidad de retrabajo	Reducir la necesidad de retrabajo
¿Qué desventajas o inconvenientes cree que tiene el uso de la herramienta que se está evaluando?	Ninguna	Falsos positivos	Ninguna	Ninguna	Aumento en los tiempos de entrega	Falsos positivos, Aumento en los tiempos de entrega

	Participante #					
Pregunta #	**1**	**2**	**3**	**4**	**5**	**6**
¿Qué sugerencias o recomendaciones tienes para mejorar la herramienta que se está evaluando?	A futuro podría incluir otro tipo de validaciones en el código.	Parametrizar/inyectar configuración para hacerla más flexible		Contar con métricas de rendimiento de programación para su programación previa en la herramienta	Establecer reglas más reales de acuerdo a las necesidades de cada proyecto	Definir claramente las reglas y límites que se van a aplicar en la detección de errores
En términos generales, ¿utilizaría la herramienta que se está evaluando?	Sí	Sí	Sí	Sí	Sí	Sí