Universidad Continental

FACULTAD DE INGENIERÍA

Escuela Académico Profesional de Ingeniería de Sistemas e Informática

Tesis

Herramienta GXC en el desarrollo de sistemas con el lenguaje C# caso: Procesos Prud del Sistema HISMINSA

Juan Moises Rojas Torres

Para optar el Título Profesional de
Ingeniero de Sistemas e Informática

Huancayo, 2021

AGRADECIMIENTO

Nuestro agradecimiento a la Universidad Continental, Escuela Académico Profesional de Ingeniería de Sistemas e Informática, por habernos dado la oportunidad de alcanzar este título profesional. Del mismo modo, quisiéramos reconocer especialmente la labor de nuestro asesor, el Ing. Pedro Yuri Márquez Solís, quien siempre nos brindó su conocimiento, experiencia y tiempo en el desarrollo de esta tesis.

También, quisiéramos expresar nuestro profundo agradecimiento al Mag. Hugo David Calderón Vilca por brindarnos su rigurosa asesoría y especial dedicación al trabajo de investigación que se realizó antes de desarrollar la presente tesis.

DEDICATORIA

A mis padres, quienes siempre fueron un pilar importante para que pueda realizar la presente investigación.

A mis hijos y esposa, pues, cuando lean esta tesis, verán todo el esfuerzo que realicé para lograr un objetivo más, y para que nunca se rindan en sus sueños y siempre alcancen sus metas. Nunca dejen de aprender; todos los días se aprende una lección nueva.

A mis hermanas y sobrinos, quienes siempre fueron una motivación extra.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

La presente tesis titulada “HERRAMIENTA GXC EN EL DESARROLLO DE SISTEMAS CON EL LENGUAJE C#. CASO: PROCESOS CRUD DEL SISTEMA HISMINSA” aborda el problema referente a la siguiente interrogante: ¿cómo mejorar el desarrollo de sistemas con el lenguaje C#? Para ello, se construyó la herramienta GXC, que se aplicó a los procesos CRUD del sistema HISMINSA. Para este desarrollo, se adaptó la metodología *Discipline Agile Delivery*. Esta herramienta GXC genera código para el lenguaje *CSharp*, orientado a los procesos CRUD realizados con la tecnología *Entity Framework*, y se puede conectar con el gestor de base de datos SQL *Server* para obtener información de esquemas, tablas, columnas y otros. Las pruebas compararon la programación manual con la programación que utiliza la herramienta GXC. Para estas evaluaciones, se usaron 100 tablas, lo que dio como resultado que el desarrollo de *software* con la herramienta GXC es mejor que el desarrollo de *software* con la programación manual. Se aplico la prueba estadística de Kolmogorov-Smirnov y considerando que las muestras no son normales y relacionadas, se aplicaron pruebas estadísticas de Wilcoxon con un valor de significancia menor a 0.05, pruebas que demuestran que la programación con GXC ocasiona menos errores y el tiempo de programación es menor, ya que la media de errores de programación manual es 10,43 errores; sin embargo, la programación con GXC no generó ningún error. Finalmente, el tiempo de programación manual obtuvo una media de 7,49866663 minutos, mientras que la media del tiempo de programación con GXC resultó en 0,00030852 minutos.

PALABRAS CLAVE

Herramienta GXC, Generación de código, *CSharp*, Herramientas CASE, HISMINSA, Eficiencia, Tiempo de programación.

ABSTRACT

This thesis entitled “GXC TOOL IN THE DEVELOPMENT OF SYSTEMS WITH THE C# LANGUAGE. CASE: HISMINSA SYSTEM CRUD PROCESSES” addresses the problem referring to the following question: how to improve the development of systems with the C# language? For this, the GXC tool was built, which was applied to the CRUD processes of the HISMINSA system. For this development, the Discipline Agile Delivery methodology was adapted. This GXC tool generates code for the CSharp language, oriented to CRUD processes carried out with the Entity Framework technology, and can connect to the SQL Server database manager to obtain information on schemas, tables, columns, and others. The tests compared manual programming with programming using the GXC tool. For these evaluations, 100 tables were used, which resulted in software development with GXC tool being better than software development with manual programming. The Kolmogorov-Smirnov statistical test was applied and considering that the samples are not normal and related, Wilcoxon statistical tests were applied with a significance value of less than 0.05, tests that show that programming with GXC causes fewer errors and the programming is lower, since the average number of manual programming errors is 10.43 errors; however, programming with GXC did not generate any errors. Finally, the manual programming time obtained an average of 7.49866663 minutes, while the average programming time with GXC was 0.00030852 minutes.

KEYWORDS

GXC Tool, Code Generation, CSharp, CASE Tools, HISMINSA, Efficiency, Programming time.

INTRODUCCIÓN

Durante el desarrollo de sistemas, las métricas de fiabilidad, eficiencia y mantenibilidad de código buscan ser mejoradas por parte de las empresas. En (1) , se considera que un *software* debe ser fiable, mientras que, en (2), se señala que se debe lograr la eficiencia en el desarrollo de *software*. Luego, en (3), se agrega que la mantenibilidad es una de las más importantes métricas para lograr que un producto sea fácil de modificar, corregir, adaptar y mejorar. Por ello, el objetivo de la presente tesis busca mejorar el desarrollo de sistemas con el lenguaje C# mediante la generación automática de código para los procesos CRUD. Para ello, se realizaron pruebas usando como caso de uso los procesos CRUD del sistema HISMINSA. Los resultados de estas pruebas llevadas a cabo mostraron que la media de programación manual es 7,49866663 minutos, mientras que, con GXC, es 0,00030852 minutos.

De esta forma, en principio, sobre la estructura de este trabajo de investigación, el primer capítulo fundamentará la problemática referente al desarrollo de *software*. Asimismo, se definirán los objetivos, los cuales se centrarán en mejorar la fiabilidad, eficiencia y mantenibilidad de código. También, se precisará y sustentará la justificación de la presente tesis.

Luego, en el segundo capítulo, se mostrarán y resumirán los antecedentes internacionales y nacionales que servirán de marco teórico a la presente investigación. Además, se expondrán las bases teóricas de nuestro análisis.

Por otro lado, el tercer capítulo contiene con detalle la metodología que se aplicará, que es una adaptación de *Discipline Agile Delivery*. A su vez, se mostrarán los artefactos esperados.

Después, el análisis y diseño de la solución se encontrarán en el capítulo cuarto. En él, se podrá ver la designación de roles, el alcance general, así como el *product backlog* de los requerimientos. También, en este capítulo, se podrán encontrar la planificación de los *sprints*, el diseño de interfaces, la validación de diseños y el diagrama de navegación.

Por último, la construcción de la herramienta, pruebas y resultados los podremos encontrar en el quinto capítulo, además de la arquitectura de la herramienta, conexiones, proceso de generación de código fuente y los datos de pruebas de forma detallada. Finalmente, en este capítulo, se expondrán los resultados obtenidos.

El autor

CAPÍTULO I

PLANTEAMIENTO DEL PROBLEMA

1.1 Planteamiento y formulación del problema

1.1.1 Planteamiento del problema

Existen diferentes formas de desarrollar *software*. En las etapas de análisis y diseño del proyecto, se utilizan muchas metodologías. De esta manera, existen herramientas para ayudar al desarrollo de *software* y la mayoría de estas utilizan modelos preestablecidos. Al desarrollar *software*, se generan restricciones como el tiempo de construcción, la reducción de costos, la calidad del *software*, la fiabilidad del código y la mantenibilidad de un programa.

En ese sentido, muchas empresas buscan mejorar la fiabilidad del *software* que ofrecen. Por ejemplo, una de las empresas más destacadas en tecnología lo es Microsoft (1), donde la mayoría de desarrolladores son conscientes de que un *software* debe ser fiable. Sobre esta cualidad, el IEEE, en vocabulario de ingeniería de sistemas y *software* (4), define fiabilidad, primero, como la cualidad de un sistema o componente para desempeñar sus funciones requeridas bajo condiciones determinadas durante un período fijo; y, segundo, la define como la capacidad del producto de *software* para mantener un nivel determinado de rendimiento al ser utilizado en condiciones concretas. Por lo

anterior, con respecto al desarrollo de *software*, debemos tener presente entonces, como un problema a enfrentar, la fiabilidad del código.

Sobre este objetivo, la presente investigación analiza el problema de mejorar el desarrollo de sistemas mediante la implementación de código fiable, eficiente y mantenible. Para ello, se propone como caso el uso de la base de datos del sistema HISMINSA, que tiene un total de 151 tablas, y el comparar métricas de fiabilidad, eficiencia y mantenibilidad entre código obtenido manualmente y generado a través de un sistema generador (GXC). Para ello, se consideraron 100 tablas.

Standish Group en (5) muestra el análisis de la resolución de proyectos de *software* por industria entre el año fiscal 2011 al 2015, donde los proyectos bancarios y financieros tienen un 15% de fallo, los proyectos de Gobierno tienen 24% de fallo, los proyectos de cuidado de la salud tienen 18% de fallo, los proyectos de fabricación tienen 19% de fallo, los proyectos de venta minorista tienen 16% de fallo, los proyectos de servicio tienen 19% de fallo, los proyectos de Telecom tienen 23% de fallo y otros proyectos de Gobierno tienen 23% de fallo. Asimismo, la resolución de proyectos de *software* por áreas del mundo indica que en Norteamérica existe un 18% de fallo; en Europa, un 19% de fallo; en Asia, un 20% de fallo; y, en el resto del mundo, un 21% de fallo. Ante esto, debemos indicar que aún existen errores en el desarrollo de *software*, por lo cual es importante que se realicen investigaciones orientadas a la resolución de este tipo de errores. La presente investigación es un aporte para la resolución de errores en los proyectos de *software*.

El error humano en el desarrollo de *software* siempre estará presente, porque los seres humanos estamos constantemente expuestos a descuidos, yerros, ya sea por cansancio, distracción, falta de conocimiento u otros motivos. En la programación, los errores pueden significar retraso en el tiempo de entrega de *software*, la aparición de *software* poco fiable o la presencia de *software* difícil de mantener. Para ilustrar esto, en la investigación (6), se analizaron los errores que cometen los estudiantes al momento de desarrollar *software* de forma ágil. Esta investigación recopiló información del 2013 al 2016. El 2013, se formaron 9 equipos entre 49 estudiantes; el 2014, se formaron 9 equipos entre 54 estudiantes; el 2015, se formaron 8 equipos entre 46 estudiantes; y el 2016, se formaron 9 equipos entre 54 estudiantes. De los equipos formados, se recopilaron errores cometidos de forma de trabajo local y remoto, así como los

errores que son de dependencia, sintaxis, falta de coincidencia de tipo, semántica, anotación y entorno. En la Tabla 1, podemos ver la cantidad de errores por la clasificación.

Tabla 1. Estadísticas de errores de compilación para cada categoría y compilaciones exitosas

Clasificación	Local (Errores)				Remota (Errores)			
	2013	2014	2015	2016	2013	2014	2015	2016
C1: Dependencia	34	14	18	34	86	118	64	18
C2: Sintaxis	12	6	12	7	0	10	0	0
C3: Falta de coincidencia de tipo	2	3	1	16	28	27	18	2
C4: Semántica	6	16	2	10	19	5	10	2
C5: Anotación	6	22	11	15	16	59	2	8
C6: Entorno	280	81	253	249	0	0	0	0
Total # de C1 a C5	60	61	44	82	149	219	94	30
Total # de construcciones exitosas	2395	1506	1670	2655	2309	1610	1764	2065

Fuente: (6)

Diferentes estudios consultados a la fecha de realización de la presente tesis hacen referencia a problemas relacionados con el desarrollo de *software*. En la Tabla 2, podemos ver que los problemas y retos en este desarrollo siguen presentes. Las páginas de las que recopilamos la información son binaryfolks.com, devetry.com, velneo.es, ibeta.com y northware.mx. los años de la publicación van desde el año 2016 hasta el año 2021. Así mismo podemos ver que detallan entre tres a siete problemas respectivamente.

Tabla 2. Publicaciones sobre problemas de desarrollo de *software*

Página *web*	binaryfolks.com	devetry.com	velneo.es	ibeta.com	northware.mx
Año de Publicación	-	2021	2019	2019	2016
Título de la publicación	5 problemas más comunes de desarrollo de *software* personalizado	Los problemas más comunes en el desarrollo de *software*	4 problemas más comunes en un departamento de programación	3 problemas comunes con el proceso de desarrollo de *software*	7 errores comunes en proyectos de desarrollo de *software*
Problema 1	Requisitos de *software* poco claros y en constante cambio	Errores, código roto y deuda técnica	Los silos crean cuellos de botella en el seno de los equipos	Comunicación inadecuada entre equipos	Mala estimación de tiempos
Problema 2	Comunicación inadecuada	Optimización prematura	Los silos **hacen** que el proyecto se vuelva más vulnerable	Programación deficiente del proceso de desarrollo de *software*	Insuficiente administración de los riesgos
Problema 3	Confidencialidad de la información	Atascado con nuevas tecnologías complicadas	Los silos reducen la cohesión del equipo de desarrollo	Falta de pruebas de *software*	Escatimar en el control de calidad
Problema 4	Demasiados errores y un producto final defectuoso	Demasiada (o no suficiente) abstracción	Los silos promueven la desconfianza y las dicotomías en la empresa y en los equipos		Diseño inadecuado
Problema 5	Costos ocultos	Pasar por alto las pequeñas cosas			Confiar demasiado en tecnologías-herramientas no exploradas previamente
Problema 6					Motivación débil
Problema 7					Añadir más personal a un proyecto atrasado

1.1.2 Formulación del problema

1.1.2.1 Problema general

- ¿Cómo mejorar el desarrollo de sistemas con el lenguaje C#? Caso: procesos CRUD del sistema HISMINSA

1.1.2.2 Problemas específicos

- ¿Cómo mejorar la fiabilidad del desarrollo de sistemas con el lenguaje C#? Caso: procesos CRUD del sistema HISMINSA
- ¿Cómo incrementar la eficiencia del desarrollo de sistemas con el lenguaje C#? Caso: procesos CRUD del sistema HISMINSA
- ¿De qué manera mejorar la mantenibilidad del desarrollo de sistemas con el lenguaje C#? Caso: procesos CRUD del sistema HISMINSA

1.2 Objetivos

1.2.1 Objetivo general

- Mejorar el desarrollo de sistemas con el lenguaje C#. Caso: procesos CRUD del sistema HISMINSA

1.2.2 Objetivos específicos

- Mejorar la fiabilidad del desarrollo de sistemas con el lenguaje C#. Caso: procesos CRUD del sistema HISMINSA
- Incrementar la eficiencia del desarrollo de sistemas con el lenguaje C#. Caso: procesos CRUD del sistema HISMINSA
- Mejorar la mantenibilidad del desarrollo de sistemas con el lenguaje C#. Caso: procesos CRUD del sistema HISMINSA

1.3 Justificación e importancia

1.3.1 Justificación práctica

Los ingenieros y analistas de sistemas, desarrolladores de *software*, etc., se ocupan de digitalizar procesos para las organizaciones ya sea con sistemas de información, aplicaciones informáticas u otra solución informática. Muchas de estas soluciones terminan en grandes desarrollos de sistemas o en una aplicación informática.

En este contexto, esta tesis se justifica en el ámbito práctico, ya que se aplica teoría de la ingeniería de *software*, estructura de datos y algorítmica para generar código que sea fiable, eficiente y mantenible.

A pesar de que en la actualidad existen herramientas CASE que se ocupan de ayudar en las diferentes fases de desarrollo de *software*, así como también existen diferentes generadores de código, estas herramientas CASE no son muy utilizadas, porque muchas no están modeladas de acuerdo a las necesidades de los programadores de *software*. Esto se debe a que traen modelos predefinidos y estos no se pueden configurar para poder generar el código. Asimismo, el dispositivo GXC es configurable; esto quiere decir que los

programadores pueden configurar la herramienta al gusto de cada uno. De esta manera, no se obliga a ningún programador a utilizar el mismo modelo que utiliza otro de estos profesionales.

1.3.2 Importancia

La importancia de la presente tesis radica en que mediante el desarrollo de la herramienta GXC, se mejora el desarrollo de sistemas con el lenguaje C# en base al patrón de diseño MVC dirigido a ADO.NET Entity Framework. Se establece modelos y en base a estos se genera código para la capa de datos y capa lógica de negocios.

La finalidad de la herramienta GXC es aumentar la eficiencia, reducir los recursos económicos y reducir los errores en el desarrollo de *software*.

Finalmente, la herramienta GXC genera código fuente para los procesos CRUD con la tecnología *ADO.NET Entity Framework*, destinada al lenguaje CSharp. Así mismo, la herramienta GXC debe convertirse en una herramienta útil para los desarrolladores de *software*, reduciendo el fallo y, también, mejorando el tiempo en su desarrollo. Para lograr lo anterior, evaluamos las formas de generación de código y realizar así un óptimo desempeño de esta última.

CAPÍTULO II

MARCO TEÓRICO

2.1 Antecedentes del problema

2.1.1 Tesis internacionales

Como referencia de nuestro estudio, podemos ver a *DotNetGenerator*, que es una herramienta desarrollada en la investigación (7). Dentro del problema, se refiere a que se requiere un generador de código para ASP.NET con MVC, y, así, automatizar la construcción de componentes comunes en el desarrollo de una aplicación *web*. Por esta razón, se propone el desarrollo del generador de código *DotNetGenerator* con el objetivo de generar el código fuente de una aplicación *web* ASP.NET MVC a partir de un modelo ISML (*Information Systems Modeling Language*). Sobre este propósito, se utilizó la metodología DAD (*Disciplined Agile Delivery*) para el desarrollo del proyecto. Además, se realizó una prueba de concepto con la fabricación de un modelo ISML para una aplicación de gestión de tienda de mascotas llamada *GestionPetStoreISML*.

Como otra de las referencias de nuestra tesis, la investigación (8) presenta un enfoque combinado para generar un código completo a partir de los modelos estructurales y de comportamiento. Para ello, se usa el enfoque de arriba hacia abajo aplicando refinamientos recursivos de elementos de actividad en un

diagrama de actividad. Además, se usa el enfoque de abajo hacia arriba desarrollando una herramienta para la creación de ejecutores de actividad y el mecanismo de mapeo del nivel más bajo de actividades en los ejecutores. En su contexto, un modelo que no incluye ejecutores es un modelo PIM. Con respecto a la generación de código, se considera que el modelado es solo el cambio de enfoque de lenguajes orientados a la implementación a lenguajes orientados a gráficos (modelos). Por lo tanto, para generar código, los lenguajes de modelado necesitan abstraer operaciones aritméticas básicas, operaciones lógicas básicas y manipulaciones de cadenas, como ejecutores de actividades básicas.

En tercer lugar, Junno Tantra Pratama Wibowo, Bayu Hendradjaya y Yani Widyani, en su artículo publicado en la *2015 International Conference on Data and Software Engineering* (9), proponen como producto final un generador de prueba de unidad Lua (LUTG), integrado a uno de los IDE de Lua más populares, ZeroBrane *Studio*, como *plugin* para conectar sin problemas el proceso de codificación y prueba. El generador puede generar código de prueba de unidad, guardar datos de casos de prueba en formato de archivo Lua y XML, y generar los datos de prueba automáticamente, usando una técnica basada en búsqueda y un algoritmo genético. De esta forma, se lograrán criterios de prueba de cobertura de rama completa. Por ello, resaltamos el aporte de este estudio para nuestra tesis, pues la investigación referida consiste en la forma de guardar datos en formato XML con la finalidad de guardar las opciones de generación de código y exportar la base de datos.

Por otro lado, la investigación (10) tuvo como objetivo establecer una integración perfecta de generación de código basada en la arquitectura de los procesos individuales de los desarrolladores en la base de convenciones estrictas de código. En una de sus conclusiones, se indica que hay dos condiciones previas principales para utilizar el enfoque presentado. Primero, se debe describir una arquitectura de referencia en un estilo basado en patrones para lograr que la extracción de patrones AIM sea lo suficientemente fácil. Esto incluye la existencia de pautas sobre el nivel de código de manera óptima y fragmentos de código de muestra que indican las mejores prácticas. En segundo lugar, la arquitectura de referencia debe prescribir suficientes convenciones de nomenclatura, así como la pila de tecnología, para permitir que la generación incremental entreteja automáticamente el código generado en el código existente. De esta manera, resaltamos los aportes de este estudio

por la forma en que plantean los patrones. Con este planteamiento, nosotros podremos generar nuestros propios patrones y, generar, así, correctamente el código.

En la investigación (11), se proporciona un análisis exhaustivo de una herramienta CASE y la evaluación se realiza con una comparación sólida con otros modelos basados en UML. Como conclusión, este estudio demuestra claramente que la automatización del desarrollo basado en modelos es más eficiente que el desarrollo simple. Así mismo, se indica que el modelado UML se ha convertido en un ejemplar estándar en el mundo de la ingeniería y, para el éxito del proyecto de la investigación mencionada, ese enfoque fue esencial. Esto se debe a que la automatización del proceso les ayudó mucho a escribir código libre de errores en menos tiempo y esfuerzo. De igual manera, en este estudio, se manifiesta que la participación humana se minimiza. Por ello, se eliminan casi todos los errores causados por diferentes factores, como poco conocimiento, bajo interés y fatiga. También, se indica que la eficiencia y la precisión son dos causas principales que empujan a los lectores a utilizar el enfoque de su investigación. Por estas razones, la evaluación de este trabajo sobre herramientas CASE nos es de utilidad en vista de que un desarrollo de *software* basado en modelos es más eficiente que un desarrollo simple.

Del mismo modo, en la *3rd International Conference on Mechatronics and Intelligent Robotics* (ICMIR-2019), en la investigación (12), se propone el modelo métrico que incluye seis características métricas. Además, se considera que existen muchas investigaciones orientadas a la métrica de código fuente estático y que existen pocas acerca de generación automática de código. En ese sentido, en este estudio, se utilizan tres herramientas automáticas de generación de código para medir el código fuente experimental. Dentro de sus conclusiones, se señala que los resultados de medición del análisis pueden mostrar que el modelo métrico de generación automática de código es razonable, lo que puede ser útil para medir este tipo de generación automática. Dentro de los procedimientos para obtener resultados, se puede decir que, para verificar su modelo métrico, utilizan *CodeGeneration*, *java-codetool* y *aiXcoder* y generar, así, los objetos experimentales. Asimismo, se utiliza generación de código basada en plantillas y los programadores predefinen algunos elementos de forma experimental. También, utilizan generación de código basada en el aprendizaje automático. Luego, el código fuente se mide automáticamente. La calidad del código fuente generado por

CodeGeneration y *java-codetool* es aproximadamente la misma. Las herramientas basadas en plantillas generan menos líneas de código que las basadas en aprendizaje automático. Asimismo, las herramientas de aprendizaje automático emplean un tiempo y espacio mayor que las herramientas basadas en plantilla, esto con respecto a la cantidad. Al obtener como resultado que la generación automática de código es efectiva, la investigación referida contribuye a sostener lo afirmado por la presente tesis en relación con la eficiencia.

Por otro lado, Han, J., y otros en (13) proponen el diseño e implementación del sistema de generación automática de código de arquitectura B / S. Sobre esto, concluyen que, debido a la gran similitud entre módulos en la arquitectura B/S, a menudo se realiza un trabajo repetitivo de copiar y pegar en el desarrollo. Este estudio presenta las tecnologías relacionadas y utilizadas en su sistema, y se elabora la función de cada módulo de su sistema. Asimismo, se define un formato XML para describir entidades de información, se implementan módulos generales y módulos de funciones del sistema. En una de sus conclusiones, se manifiesta que su sistema de generación automática de código puede mejorar la eficiencia del desarrollo. El proceso que siguen es el modelo de plantilla que lee la información del atributo y reemplaza las etiquetas de la variable. Luego, se genera el código objetivo final. Con la práctica, se ha demostrado que el usuario puede generar rápidamente aplicaciones de arquitectura B/S mediante el archivo de descripción de entidad que proporciona el usuario utilizando el sistema de generación automática de código. De esta forma, valoraremos la definición de un formato XML que se realiza en el estudio referido, en vista de que, para la presente tesis, se definen 2 archivos XML: uno, donde se almacenan los tipos de datos; y otro, para el tipo de código que se va a generar.

Con respecto a la siguiente propuesta, Marcos Antonio Possatto y Daniel Lucrédio en (14) proponen un mecanismo para detectar y propagar semiautomáticamente los cambios del código de referencia a las plantillas, manteniéndolos sincronizados con menos esfuerzo. En una de sus conclusiones, manifiestan que todavía hay un margen de mejora. A su vez, sus resultados indican que la automatización se puede utilizar para reducir el esfuerzo y el costo en el mantenimiento y la evolución de una generación de código basada en plantillas de infraestructura. Como resultados, observaron que el mecanismo desarrollado puede conducir a una reducción del 50% en el esfuerzo necesario para realizar la plantilla de mantenimiento/evolución, en

comparación con un enfoque manual. Así mismo, estos investigadores observaron que ese efecto depende de la naturaleza de la tarea de evolución/mantenimiento, ya que, para una de las tareas, no había una ventaja observable en el uso del mecanismo (14). Finalmente, los resultados obtenidos en la investigación referida resultan un importante aporte para los resultados obtenidos en la presente tesis, en vista de que también tenemos un índice de mantenimiento adecuado con el código generado.

Luego, en la investigación (15), se propone un método automático de generación de código de página basado en la plantilla de Excel y la tecnología POI, dentro de la cual tiene dos ventajas. De esta forma, se concluye en que los desarrolladores de *software* no necesitan prestar atención a los detalles de la lógica empresarial. Para páginas de contenido complejas, los usuarios pueden diseñar / importar directamente un documento relativamente excelente y describir los contenidos de Excel basados en diferentes marcos *web* (como *easyUI*, *BootStrap*, etc.). En su aplicación práctica, este estudio demuestra que su método realiza la generación rápida y automática de interfaces. Con esto, se logra la reutilización de código, se mejora la eficiencia de desarrollo y se reducen costos de desarrollo. La importación de un documento se tomó como referencia para nuestra tesis en vista de que se puede exportar e importar una base de datos.

Sobre el siguiente aporte, Jean Pierre Alfonso Hoyos y Felipe Restrepo Calle en la investigación (16) proponen un enfoque para generar automáticamente un prototipo de aplicación *web* que ejecute procesos comerciales utilizando una especificación de lenguaje natural restringida. En esta, concluyen que tienen un método de creación rápida de prototipos para aplicaciones *web*, que ejecutan procesos comerciales utilizando una especificación de lenguaje natural restringido. Como resultado, obtuvieron un código fuente listo para ser parte del producto final, lo que reduce los problemas asociados con las etapas de obtención de requisitos y diseño. Ante esto, entonces, utilizamos de guía el método de creación rápida de prototipos que utilizan estos investigadores para poder definir el método más eficiente y, así, generar rápidamente el código.

La siguiente propuesta, la investigación (17), es la de desarrollar ISML-MDE, que viene a ser un entorno basado en modelos que comprenden tres componentes principales. Estos son los siguientes: un lenguaje de modelado para aplicaciones empresariales (ISML), un marco de generación de código

(ISML-GEN) y *LionWizard*. Luego del análisis respectivo, en este estudio se concluyó que presentaron la experiencia práctica de la creación de ISML-MDE en un entorno basado en modelos que fueron utilizados para abstraer y automatizar el desarrollo de aplicaciones empresariales. Como resultado, se obtuvo que el nivel de abstracción en ISML es suficientemente alto para ocultar los detalles de implementación, pero bajo para especificar la mayoría de una aplicación empresarial. Asimismo, los ingenieros aprenden gradualmente el idioma y lo adoptan fácilmente debido a las facilidades de modelado parcial proporcionadas por la palabra clave nativa. Además, la generación y transformación de código modular automatizan la mayor parte de la implementación de la aplicación y, también, contribuyen al olvido de los detalles de la plataforma de destino. Sobre lo revisado, verificamos positivamente los modelos de esta investigación para elaborar nuestra propia forma de generación de código.

Finalmente, la investigación (18) propone *JDriver*, que es un marco de generación automática de clases de controladores para herramientas de *fuzzing* basadas en AFL, que puede generar código de controlador para los archivos de procesamiento de métodos, así como métodos ordinarios que no procesan archivos. En conclusión, su método de generación automática de clases de controladores emplea análisis de dependencia para analizar a qué campos accede el método y luego construir secuencias de métodos para mutar los valores de los campos a los que se accede. El objetivo es que las secuencias de métodos generadas puedan mutar el estado de las instancias de clase. Para obtener los resultados, se evaluó a *JDriver* en *commons-Imaging*, una biblioteca de imágenes ampliamente utilizada. Proporcionada por la organización Apache, *JDriver* ha generado con éxito 149 métodos auxiliares que se pueden usar para crear instancias destinadas a 110 clases. Además, se crean 99 clases de controladores para cubrir 422 métodos. En esta perspectiva, la forma de generación de código de los controladores de esta investigación nos sirvió para realizar la generación de los *controller* de la presente tesis.

2.1.2 Tesis nacionales

De acuerdo a (19), se realizó una investigación cuyo objetivo fue determinar la influencia en la mantenibilidad de código de los sistemas de información empresariales mediante la aplicación de un *software* generador de código de

funcionalidades tipo CRUD, lo que logró obtener una diferencia en facturación electrónica pretest y postest, una complejidad dicromática de 10.25, acoplamiento entre objetos 17.25, densidad de comentarios de código 5.25 y volumen del programa 1103.48. Las diferencias en el sistema de repartos dan como resultado entre el pretest y postest una complejidad dicromática de 12.75, acoplamiento entre objetos 23.3, densidad de comentarios de código 3.94 y volumen del programa 1558.15. Sobre estos resultados del estudio en mención, se destaca como aporte que un generador de código tipo CRUD ocasiona influencia positiva sobre la mantenibilidad y la dimensión.

Finalmente, sobre la investigación (20), esta propuso un *framework* para automatizar las tareas repetitivas de acceso a datos. Para ello, se usaron estándares internacionales de escritura de código y se utilizó el patrón Facade para minimizar la complejidad de un sistema. A partir de este estudio, se concluyó que se mejora la eficiencia en el desarrollo de *software* aplicando patrones MVC y *Unit of Works*. También, se mejora esta eficiencia automatizando la codificación de procedimientos almacenados, automatizando la codificación de clases y métodos, y automatizando la propagación de cambios.

2.2 Bases teóricas

2.2.1 Desarrollo de *software*

El desarrollo de *software* es un área profesional que construye *software* para organizaciones cuyos propósitos son específicos o para ser un producto de *software*, esto a través de diferentes métodos de desarrollo de *software*. Este desarrollo es apoyado por la ingeniería de *software* mediante técnicas para la especificación, el diseño y la evolución del programa (21).

2.2.1.1 Ingeniería de *software*

Es una disciplina de la ingeniería que se interesa y se enfoca en la producción de *software* desde la etapa de especificaciones hasta el mantenimiento. Esta disciplina se refiere a que los ingenieros hacen funcionar, aplican teorías, métodos y herramientas, todo de manera selectiva, y brindan solución a problemas aun cuando no hay teorías ni métodos aplicables. La ingeniería de *software* también incluye actividades

como la administración de proyectos, métodos, teorías para apoyar la producción de *software* y desarrollo de herramientas (21).

2.2.1.2 Metodologías de desarrollo de *software*

Las metodologías ayudan a definir quién debe hacerlo, cuándo debe hacerlo y cómo debe hacerlo. Por lo tanto, resultan un necesario marco de trabajo para planificar, estructurar y controlar el proceso de desarrollo en *software* (22). En la actualidad, existen diferentes metodologías de desarrollo de *software*. Entre estas, podemos ver SCRUM, *Extreme Programming* XP, *Discipline Agile Delivery*, etc.

En la Tabla 3, podemos ver un comparativo de las fases de desarrollo de *software* de las metodologías *Extreme Programing* XP, SCRUM y *Discipline Agile Delivery*.

2.2.1.3 Procedimientos CRUD

Decir CRUD para el desarrollo de *software* es hacer referencia a crear (C), leer (R), Actualizar (U) y eliminar (R). Todo lo anterior es con el fin de gestionar la información dentro de un sistema diseñado para que se pueda administrar información (19).

La mayoría de aplicaciones de *software* interactúan con bases de datos y administran información, por lo cual requieren de los procedimientos CRUD para llevar a cabo dichos procedimientos (23).

Crear es el proceso de agregar un nuevo registro a la base de datos. Por ejemplo, un caso que ilustra lo anterior es crear un nuevo registro de un paciente.

Leer es la forma de recuperar los registros de la base de datos. Un ejemplo de ello es mostrar los datos personales de un paciente.

Actualizar es la acción de modificar un registro previamente creado. Por ejemplo, es modificar el apellido de un paciente que se registró con algún error o se cambió de apellido mediante acciones legales.

Eliminar es la acción de remover un registro que se creó por algún motivo erróneo o por orden de algún superior.

Tabla 3. Cuadro comparativo de fases de metodologías ágiles

Fases del desarrollo de software	Metodologías ágiles		
	Extreme Programing XP	SCRUM	Discipline Agile Delivery
Inicio		Crea la visión del proyecto, se identifica a los *ScrumMaster* y a los *stakeholders*, se forma equipos SCRUM, desarrolla épicas, crea *backlogs* o listas de requerimiento priorizando el producto y se planifica el lanzamiento.	Se forma el equipo y se alinea con la dirección de la organización, se explora el alcance, se identifica la estrategia de la arquitectura, se planifica el lanzamiento, se desarrolla la estrategia de pruebas, se desarrolla visión común, se asegura el financiamiento.
Planificación	Identifica las historias de usuario, prioriza y descompone miniversiones. La planificación se va revisando. Aproximadamente cada dos semanas de iteración, se debe obtener software útil, funcional, listo para probar y lanzar	En esta fase, se crean, estiman y comprometen las historias de usuario. También se identifican y estiman tareas. Se crea el *sprint backlog* o iteración de tareas.	
Diseño	Se intenta trabajar con código sencillo, haciendo lo mínimo imprescindible para que funcione. De ello se obtiene el prototipo. Se crean tarjetas CRC (Clase-Responsabilidad-Colaboración) para el diseño del *software* orientado a objetos.		
Construcción	La programación se realiza en parejas frente al mismo ordenador. En ocasiones, se cambian las parejas. De esta manera, se puede realizar un *software* universal.	Se crean entregables, se realiza *daily stand-up* y se refinancia el *backlog* priorizado del producto.	Prueba la arquitectura temprano, se abordan necesidades cambiantes de interesados, se produce la solución potencialmente consumible, se mejora la calidad y se acelera la entrega del valor.
Pruebas	Se realizan pruebas automáticas de manera constante. Este testeo automatizado y continuo es clave. El cliente puede hacer y proponer nuevas pruebas e ir validándolas.	En esta fase se demuestra y valida el *sprint.* También se realiza la retrospectiva del *sprint*.	
Despliegue	Se logra cuando se ha probado todas las historias de usuario o miniversiones con éxito.	Se debe de enviar entregables y enviar retrospectiva del proyecto.	Se debe asegurar presteza para producción e implementar la solución.

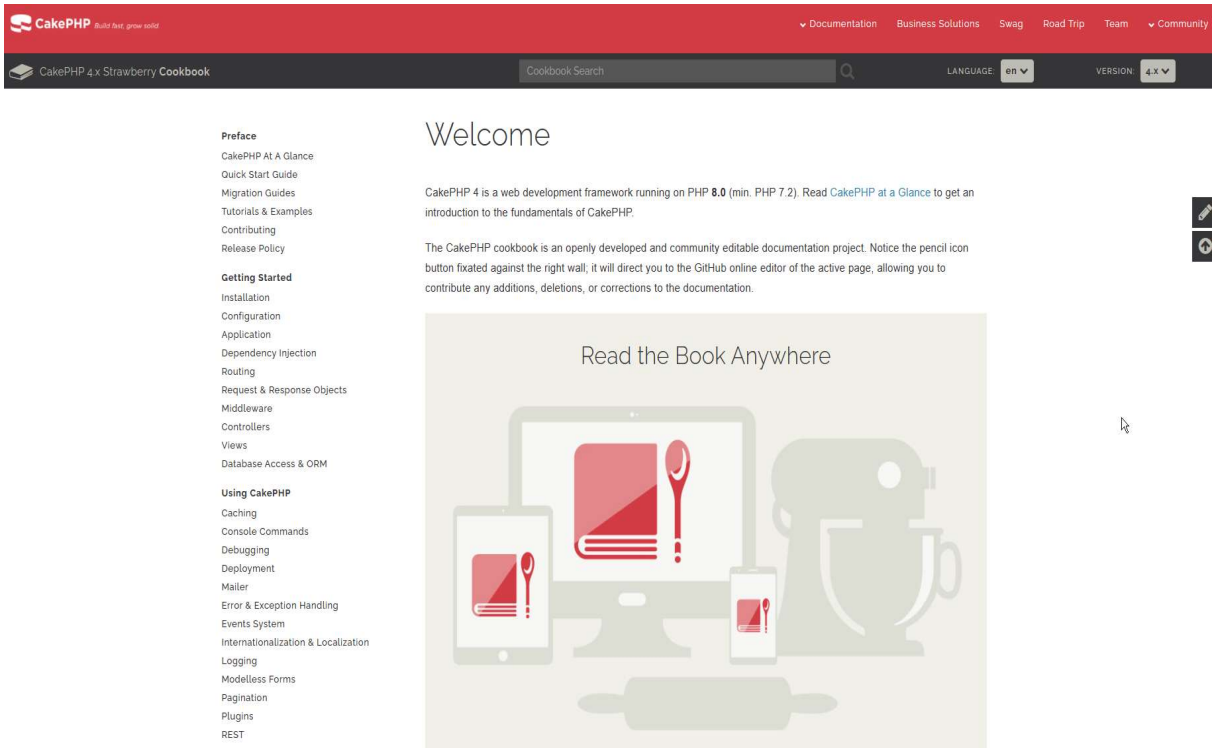
2.2.2 Generación de código fuente

La generación de código es una práctica frecuente en el desarrollo de *software.* Al utilizar los *framework,* una parte utiliza plantillas. Por lo tanto, se está generando código. También es generación de código los fragmentos de código que genera la herramienta GITHUB *copilot*.

Existen diversos generadores de código como *CakePHP*, *Symfony*, *ScriptCase*, CRUD *Admin Generator*, *Genexus*, etc.

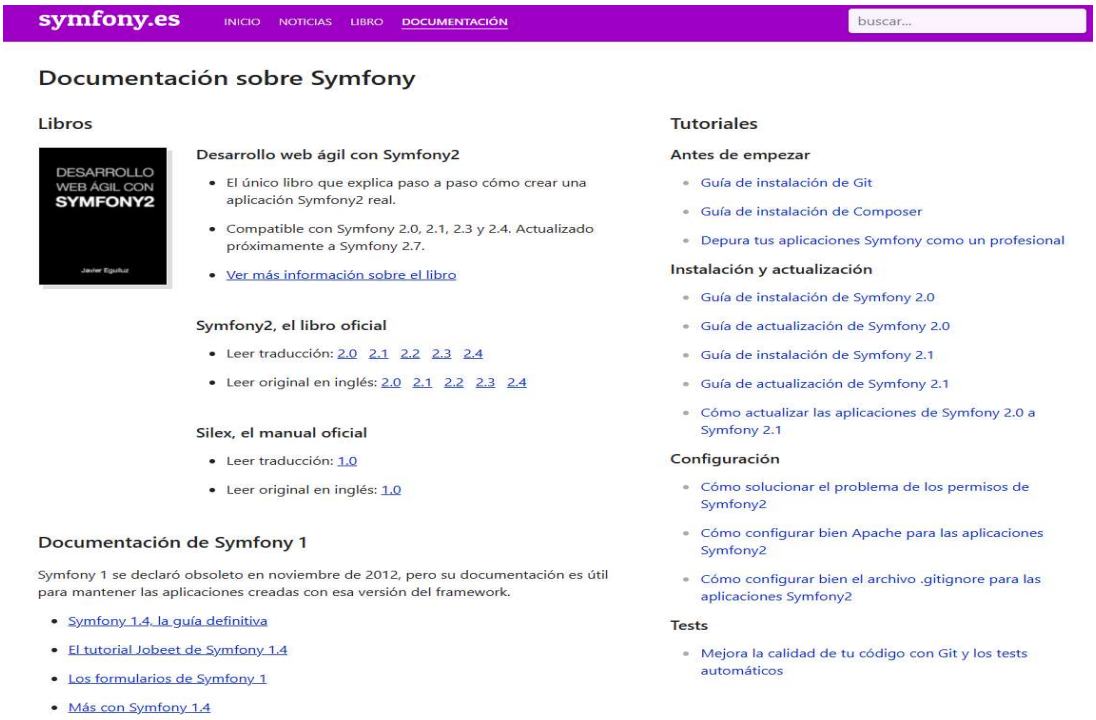
En la Figura 1, se muestra una captura de pantalla de *book* de la página *CakePHP*, que es un generador de código para el lenguaje de programación PHP.

Figura 1: Captura de pantalla de *book* de la página *CakePHP*



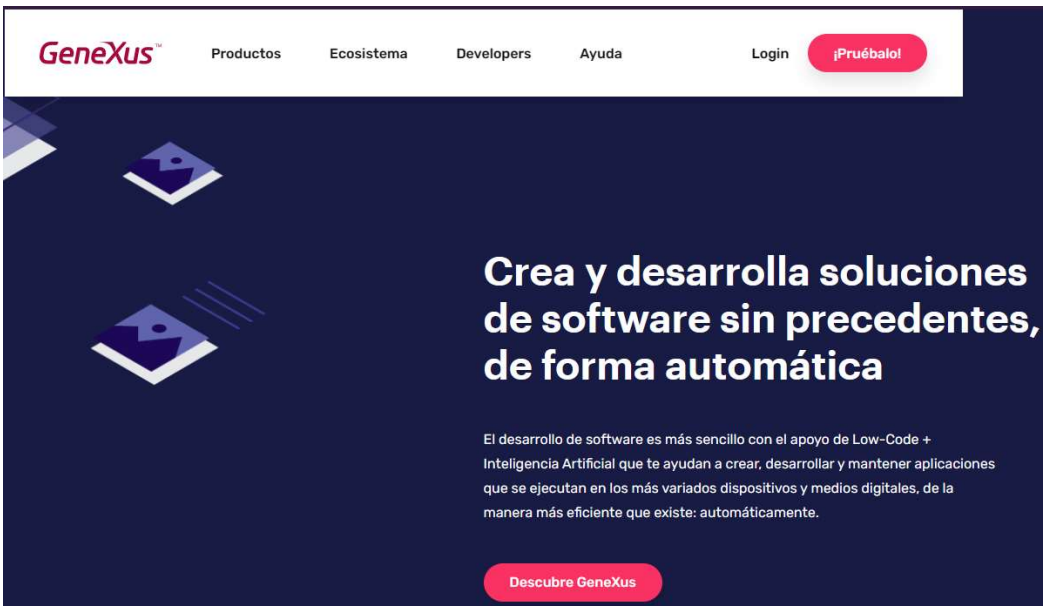
Una captura de pantalla de la documentación oficial del generador de código *Symfony* se aprecia en la Figura 2.

Figura 2: Captura de pantalla de documentación de *Symfony*



Genexus incluye un generador Genexus .NET. Este es un generador inteligente para Microsoft .NET *Framework* y genera código nativo *CSharp*. A su vez, maneja las funciones de transformación de datos, y permite el desarrollo de aplicaciones *web* y aplicaciones GUI. En la Figura 3, podemos ver la captura de pantalla de la página *web* de Genexus.

Figura 3: Captura de pantalla de la portada de página *web* de Genexus



Las herramientas CASE (*Computer Aided Software Engineering*, traducido como “Ingeniería de *Software* Asistida por Computadora”) son útiles para diferentes procesos del desarrollo de *software*. También, generan código fuente para mejorar la eficiencia y eficacia de los procesos, reducir los recursos necesarios y reducir los defectos en el desarrollo de *software*. En ese sentido, las herramientas CASE contribuyen con la economía en el desarrollo de *software*.

2.2.2.1 Framework

Un *framework* es una estructura tecnológica, guía para el desarrollo y la organización de *software*, que se basa en el uso de plantillas (24). También, podemos afirmar que es la abstracción de cierto código, y que puede ser sobrescrita de forma selectiva. De esta manera, un *framework* es una solución incompleta, pero concreta para un problema recurrente bien conocido (25). Sobre esto, Pree en (26) hace referencia a puntos calientes de *frameworks* e indica que constan de bloques de construcción listos para usar y bloques semiacabados. En el caso de los puntos calientes, estos son aquellos aspectos de dominio que deben mantenerse flexibles y estos puntos representan la dificultad de un buen diseño orientado a objetos. Flexible, se refiere a que los programadores pueden modificar o personalizar esa parte del código o en ese punto de código.

En la actualidad, tenemos diferentes *frameworks* para agilizar el desarrollo de *software*: Laravel para PHP, *Ruby on Rails* para Ruby, *AngularJS JavaScript*, *Django* para *Python*, ASP.NET para *CSharp*, ASP.NET MVC para *CSharp*, etc. En la Tabla 4, podemos ver un *ranking* de *Frameworks* que se obtuvo desde *hotframeworks*.

Tabla 4. *Ranking* de *Frameworks*

Framework	Puntaje *Github*	Puntaje *Stack Overflow*	Puntaje general
ASP.NET MVC	-	95	95
Ruby on Rails	87	99	93
AngularJS	90	97	93
Django	89	97	93
Laravel	90	93	91
ASP.NET	80	100	90

Fuente: (27)

Entity Framework (28) es un conjunto de tecnologías de ADO.NET para el desarrollo de sistemas orientados a datos. *Entity Framework* permite trabajar con los datos en forma de objetos y propiedades específicas de dominio. De esta manera, se puede trabajar a un nivel más alto de abstracción y las líneas de código son menores que las tradicionales. Es importante mencionar que los datos de *Entity Framework* se pueden consultar desde LINQ en vista de que es compatible.

2.2.2.2 Herramientas CASE

Las herramientas CASE brindan soporte a todas las actividades del proceso de ingeniería de *software*. Existen diferentes tipos de estas herramientas: para planificar proyectos, construir prototipos, para control de calidad, modelado y otros.

Por otro lado, existen diferentes investigaciones que estudian las herramientas CASE y ven cómo estas mejoran los procesos de desarrollo (29).

Los procesos de desarrollo de *software* exigen adaptarse rápidamente a los cambios y a utilizar técnicas y tecnologías. Por ello, se utilizan las herramientas CASE que asistan a los involucrados en proyectos de *software* (30).

En (31), observamos que podemos clasificar las herramientas CASE de acuerdo a la cobertura y a la funcionalidad. De acuerdo a su cobertura, tenemos la siguiente clasificación: herramientas integradas, que abarcan todas las fases del ciclo de vida, tienen un repositorio y aportan técnicas estructuradas; herramientas de alto nivel, que están orientadas a la automatización y soporte en el análisis y diseño; herramientas de bajo nivel, que abarcan las últimas fases de desarrollo como son construcción e implantación; juego de herramientas, que automatizan dentro del ciclo de vida, y que, dentro de este juego, se pueden encontrar herramientas para reingeniería y mantenimiento. Luego, por la clasificación de acuerdo a su funcionalidad, tenemos las siguientes: herramientas de planificación de sistemas de gestión, que sirven para modelar requisitos de información, brindan un metamodelo y proporcionan ayuda para nuevas estrategias cuando no se satisfacen necesidades de la organización; herramientas de análisis y diseño, que sirven para crear un modelo, evaluar y validar el

sistema que se pretende construir, así como dar un grado de confianza con respecto al análisis y prevenir errores; herramientas de programación; herramientas de integración y prueba; herramientas de gestión de prototipos, que son utilizados para la evaluación de especificaciones y para un mejor entendimiento de requisitos; herramientas de mantenimiento (aquí entran las herramientas de ingeniería inversa, reestructuración y análisis de código, así como la reingeniería); herramientas de gestión de proyectos, que brindan soporte global en la planificación de proyectos, seguimiento de requerimientos, gestión y medida; y herramientas de soporte, que son las herramientas para la documentación, el *software* de sistemas, el control de calidad y la bases de datos.

Un análisis de herramientas CASE se puede encontrar en (32), donde, para el diseño de la evaluación, se utiliza escala Likert de 3 puntos; donde 3 es fuertemente abordado, 2 es parcialmente abordado y 1 no es abordado. En la evaluación cualitativa, se abordan 5 aspectos que son el diseño de GUI, usabilidad de herramientas, funciones que promueven el uso, soporte para integración de fuentes de datos heterogéneos, y soporte y documentación. Para la evaluación cuantitativa, se utilizan métricas internas definidas por la norma ISO/IEC 9126, que son calidad en uso y mantenibilidad.

En la Tabla 5, podemos ver la evaluación cualitativa con aspectos y factores mencionados anteriormente y definidos por (32).

Tabla 5. Evaluación cualitativa

Aspecto	Factores
Diseño de GUI	Factor 1. GUI con principios de simplicidad y previsibilidad Factor 2. Interacción del usuario basada en las mejores prácticas
Usabilidad de herramientas	Factor 1. Facilidad de uso Factor 2. Facilidad de aprendizaje Factor 3. Conocimientos y habilidades técnicas
Funciones que promueven el uso	Factor 1. Uso de principios de modularidad Factor 2. Funciones con propósitos bien definidos
Soporte para integración de fuentes de datos heterogéneas	Factor 1. Interoperabilidad con herramientas de software externas Factor 2. Seguridad de los datos
Soporte y documentación	Factor 1. Documentación Factor 2. Roles activos de los usuarios Factor 3. Ayuda en línea Factor 4. Documentación para desarrolladores en línea

En la Tabla 6, podemos ver la evaluación cuantitativa con respecto a calidad en uso y mantenibilidad que plantea (32).

Tabla 6. Evaluación cuantitativa

Métrica	Factor
Calidad en uso	Productividad (Tiempo de la tarea) Aplicación básica (frecuencia de la ayuda)
Mantenibilidad	Posibilidad de cambiar (complejidad de la modificación) Analizabilidad (completitud de la documentación en línea)

2.2.3 Programación orientada a objetos

2.2.3.1 Programación orientada a objetos

De acuerdo a (33), la programación orientada a objetos es la programación estructurada y modular. Las instrucciones y datos se encapsulan en entidades llamadas clases y, luego, se crean objetos. En ese sentido, los lenguajes modernos permiten la programación orientada a objetos y es un estilo que predomina en la industria de programación.

2.2.3.2 Objeto

Este una persona, animal o cosa. En (33), indican que los objetos tienen una característica en particular que la distingue de los demás, la cual puede ser representada dentro de una clase. Así, cada objeto puede evidenciar tener diferentes campos. Por ejemplo, si referimos a un cuadrado, este tiene un ancho, alto, color, etcétera, mientras que un círculo tiene un centro, radio, diámetro, etcétera.

2.2.3.3 Clase

La clase es la categoría que agrupa objetos similares. En ese sentido, las clases se crean a partir de la agrupación de objetos de la misma categoría. Por ejemplo, si referimos sobre un objeto círculo, cuadrado o triángulo, estaríamos refiriendo a una categoría de figuras geométricas. En este caso, mi clase se llamará “figuras geométricas”. Otro ejemplo sería si hablamos de los objetos camioneta, automóvil u ómnibus. En ese contexto, estaríamos hablando de la categoría “vehículo” y la clase sería “vehículo” (33).

2.2.3.4 Método

Los métodos son básicamente las operaciones que se pueden realizar con un objeto. Por ejemplo, si tengo un triángulo, algunos métodos serían poder pintar el triángulo, girar, cambiar el tamaño, mover, etc. Estos métodos son llamados o invocados.

2.2.3.5 Procedimientos almacenados

Un procedimiento o subrutina es un subprograma que se encarga de ejecutar un proceso o procesos específicos. Los procedimientos se llaman o se invocan por su nombre. Por ejemplo, el procedimiento ActualizarColor nos indica que se va a actualizar el color de un objeto.

2.2.3.6 Tipos de dato

El tipo de dato especifica cómo se va a tratar al dato y cómo se le va a pasar el parámetro. Por ejemplo, un mes del año se puede tratar como *int* (número) o como *string* (texto).

2.2.3.7 Base de datos

Base de datos es un conjunto de datos almacenados que están organizados mediante una estructura de datos. Estas bases están diseñadas para almacenar la información de una empresa según los requerimientos de los usuarios (34).

Existen bases de diferentes tipos orientadas a objetos, objeto-relacionales, activas, multimedias, científicas, estadísticas, espaciales, temporales, multidimensionales, semiestructuradas, deductivas, difusas, con restricciones, distribuidas, federadas, móviles, multi-bd, Grid, paralelas, no-sql, etcétera. (35).

2.2.4 Métricas de calidad de código

Una métrica de calidad de código es una medida de alguna porción de código dentro de un *software*. No siempre es sencillo aplicar métricas, porque no solo es determinar, sino que también tenemos que interpretarlas. Las métricas se utilizan para estimar costos, calcular la productividad, determinar tamaño y calidad de producto (36). Para medir la calidad de código, McCall y Boëhm

proponen un enfoque científico (37) . Un ejemplo de tiempo medio entre fallos (MTBF) es el siguiente:

MTBF = T-tot / N

Donde:

T-tot = El periodo de tiempo total

N = El número de fallas en T-tot

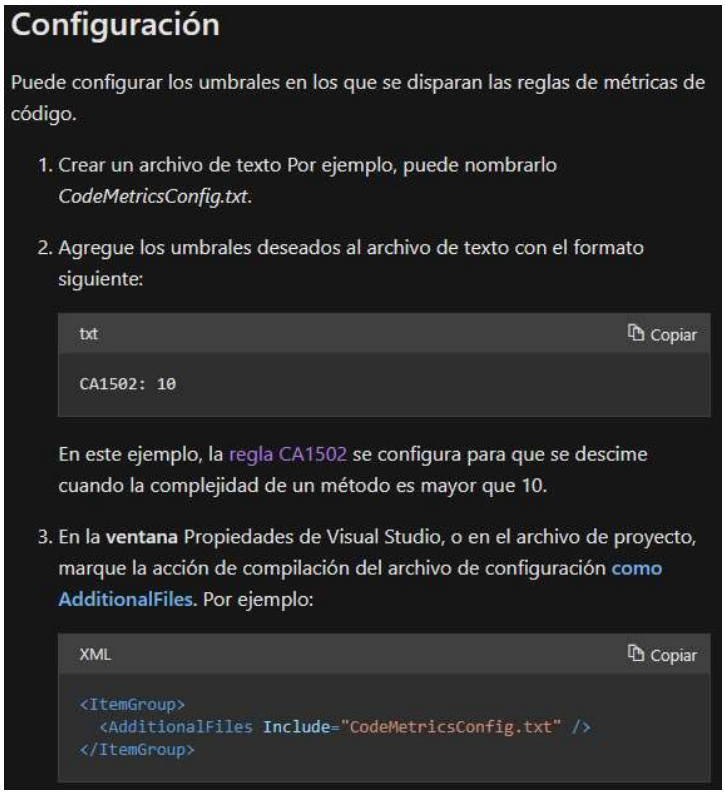
McCall fue el primero en presentar el modelo McCall en 1977. Este modelo tiene tres ejes desde los cuales el usuario puede evaluar la calidad de un producto. Este se basa en once factores de calidad organizados en torno a los ejes y cada factor se distribuye en criterios de calidad. Dentro de los factores, podemos ver, en el eje operación de producto, la fiabilidad y eficiencia. Asimismo, dentro de la revisión del producto, podemos ver la mantenibilidad.

Por otro lado, el modelo ISO 9126 fue propuesto como estándar internacional para la medición de la calidad en el *software*. Sobre este lineamiento, se clasifica la calidad de *software* en características y subcaracterísticas. Según lo observado, es una variación del modelo McCall. En este modelo, se puede ver la fiabilidad, eficiencia y mantenibilidad como factores de la calidad.

Luego, en (45), podemos ver que, para generar los datos de las métricas de código en *Visual Studio*, podemos hacerlo de tres maneras:

- Habilitar analizadores de calidad de código de .NET y habilitar las cuatro reglas de métricas de código. En la Figura 4, podemos ver un ejemplo que Microsoft nos proporciona.

Figura 4: Configuración de analizadores de calidad de código en *Visual Studio*



Fuente: (45)

- Se requiere elegir en *Visual Studio* en la barra de menú “analizar”. Luego, se debe elegir “calcular métricas de código" y, después, elegir la opción “para el proyecto o solución”.
- Se puede realizar desde la línea de comandos para proyectos *CSharp*, instalando el paquete *NuGet Microsoft.CodeAnalysis.Metrics*. Luego, debemos ejecutar el comando *msbuild /t:Metrics*. En la Figura 5, podemos ver la salida luego de ejecutar el comando *msbuild /t:Metrics*.

Figura 5: Salida de métricas de código

XML Copiar

```xml
<?xml version="1.0" encoding="utf-8"?>
<CodeMetricsReport Version="1.0">
  <Targets>
    <Target Name="ConsoleApp20.csproj">
      <Assembly Name="ConsoleApp20, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
        <Metrics>
          <Metric Name="MaintainabilityIndex" Value="100" />
          <Metric Name="CyclomaticComplexity" Value="1" />
          <Metric Name="ClassCoupling" Value="1" />
          <Metric Name="DepthOfInheritance" Value="1" />
          <Metric Name="SourceLines" Value="11" />
          <Metric Name="ExecutableLines" Value="1" />
        </Metrics>
        <Namespaces>
          <Namespace Name="ConsoleApp20">
            <Metrics>
              <Metric Name="MaintainabilityIndex" Value="100" />
              <Metric Name="CyclomaticComplexity" Value="1" />
              <Metric Name="ClassCoupling" Value="1" />
              <Metric Name="DepthOfInheritance" Value="1" />
              <Metric Name="SourceLines" Value="11" />
              <Metric Name="ExecutableLines" Value="1" />
            </Metrics>
            <Types>
              <NamedType Name="Program">
                <Metrics>
                  <Metric Name="MaintainabilityIndex" Value="100" />
                  <Metric Name="CyclomaticComplexity" Value="1" />
                  <Metric Name="ClassCoupling" Value="1" />
                  <Metric Name="DepthOfInheritance" Value="1" />
                  <Metric Name="SourceLines" Value="7" />
                  <Metric Name="ExecutableLines" Value="1" />
                </Metrics>
                <Members>
                  <Method Name="void Program.Main(string[] args)" File="C:\source\repos\ConsoleApp20\ConsoleApp20\Progra
                    <Metrics>
                      <Metric Name="MaintainabilityIndex" Value="100" />
                      <Metric Name="CyclomaticComplexity" Value="1" />
                      <Metric Name="ClassCoupling" Value="1" />
                      <Metric Name="SourceLines" Value="4" />
                      <Metric Name="ExecutableLines" Value="1" />
                    </Metrics>
                  </Method>
                </Members>
              </NamedType>
            </Types>
          </Namespace>
        </Namespaces>
      </Assembly>
    </Target>
  </Targets>
</CodeMetricsReport>
```

Fuente: (45)

2.2.4.1 Eficiencia

La eficiencia en la ejecución del *software* se relaciona con la minimización del tiempo de procesamiento. En cuanto al almacenamiento, la eficiencia se relaciona con la minimización del tamaño de almacenamiento, según el modelo de McCall (39). Según el modelo ISO 9126, la eficiencia tiene como subcaracterística al comportamiento en el tiempo que emplea y el comportamiento con respecto a recursos. Se calcula con el tiempo que demora un código para ejecutarse y los recursos que utiliza para llevar a

cabo dicha tarea. En la actualidad, tanto IDEs como *Visual Studio* 2019 tienen la opción de evaluar dicha métrica.

Lograr la eficiencia de un código en el desarrollo de *software* es un problema al que se enfrentan diariamente los desarrolladores. Cumplir con el propósito de lograr la eficiencia (2), sin desperdiciar recursos y reutilizar la mayor parte de estos recursos, midiendo respecto al tiempo y almacenamiento, es todo un reto. Sobre esto, el factor de la eficiencia es considerado por McCall y Boëhm dentro de las características de la calidad del *software* en (4).

En (40), para evaluar la eficiencia de un producto de *software*, se desarrolló un conjunto de métricas y se realizaron 3 tipos de pruebas: el porcentaje de tiempo de respuesta, el porcentaje de concurrencia y el porcentaje de errores HTTP. Las fórmulas que utilizaron son las siguientes:

Para calcular la Evaluación Final (EF):

$$\mathbf{EF = (\%TR + \%C + \%EHTTP) / 3}$$

Donde:

%TR = Por Ciento de Tiempo de Respuesta
%C = Por Ciento de Concurrencia
%EHTTP = Por Ciento de Errores HTTP

Para calcular el Por Ciento de Tiempo de Respuesta (%TR):

$$\mathbf{\%TR = (TRR * 100) / TRE}$$

Donde:

TRR = Tiempo de Respuesta Real
TRE = Tiempo de Respuesta Esperado

Para calcular el Por Ciento de Concurrencia (%C):

$$\mathbf{\%C = (CR * 100) / CE}$$

Donde:

CR = Concurrencia Real
CE = Concurrencia Esperada

Para calcular Por Ciento de Errores HTTP (%HTTP)

%HTTP = (EHTTP * 100) / CURL

Donde:

EHTTP = Errores HTTP generados

CURL = Cantidad de URL Visitadas

Del mismo modo, en el documento (41), dentro de la medición de atributos externos del producto, en medidas de usabilidad para aplicaciones *web*, se definen como eficiencia a la capacidad de ayudar a los usuarios finales a lograr objetivos en el menor tiempo. Los indicadores son estos:

- El tiempo necesario para completar exitosamente una transacción
- Número de transacciones completadas con éxito en un periodo de tiempo dado

En (41), también se indica que se calcula la media aritmética entre rangos asignados a cada indicador para la medición final de eficiencia. En la Tabla 7, se muestran los rangos asignados.

Tabla 7. Rangos asignados para cada indicador

Rango	Tiempo	N° de transacciones completadas (TC)
0	tiempo <= 3 minutos	TC > 20
1	3 minutos < tiempo <= 5 minutos	12 < TC <= 20
2	5 minutos < tiempo <= 10 minutos	6 < TC <= 12
3	tiempo > 10 minutos	TC <= 6

Fuente: (41)

2.2.4.2 Fiabilidad

La fiabilidad, según (39), tiene como métricas a la complejidad, la consistencia, la exactitud, la modularidad, la simplicidad y la tolerancia a fallos. Según el modelo de McCall, la pregunta que se debe realizar desde la perspectiva de los usuarios es la siguiente: ¿el código escrito lo hace de forma exacta siempre? Las subcaracterísticas, según el modelo ISO 9126, son la madurez, la recuperabilidad y la tolerancia a fallos.

La fiabilidad, según (42), se evalúa midiendo la frecuencia y la gravedad de los fallos, en tiempo medio de fallos, la capacidad de recuperación y la capacidad de predicción.

$$\mathbf{TMEF = TMDF + TMDR}$$

Donde:

TMEF = Tiempo Medio Entre Fallos

TMDF = Tiempo Medio De Fallo

TMDR = Tiempo Medio De Reparación

Para obtener la disponibilidad, se indica que se utiliza:

$$\mathbf{Disponibilidad = TMDF / (TMDF + TMDR) * 100\%}$$

Donde:

TMDF = Tiempo Medio De Fallo

TMDR = Tiempo Medio De Reparación

Para tener una buena fiabilidad de código, podemos tomar la complejidad ciclomática que, según (42), mide la complejidad de la estructura de código y se crea al calcular el número de rutas de código diferente en el flujo del programa. Las fórmulas a las que hace referencia y se pueden utilizar para calcular la complejidad son estas:

$$\mathbf{M = E - N + 2P}$$

Donde:

M = Complejidad

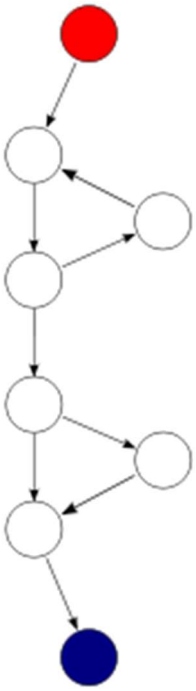
E = Número de aristas

N = Número de nodos

P = Número de componentes conectados

Para entender mejor la fórmula, podemos ver la Figura 6, donde se muestra el control de flujo de un programa simple.

Figura 6: Gráfico de control-flujo de un programa simple



Fuente: (43)

Una fórmula alternativa es la siguiente:

$$M = E - N + P$$

Donde:

M = Complejidad
E = Número de aristas
N = Número de nodos
P = Número de componentes conectados

En la investigación (6), se realizó el análisis de errores de construcción de *software*. La recopilación se hizo entre el 2013 al 2016. En el 2013, tuvieron 49 estudiantes, con los cuales se formaron 9 equipos; el 2014, tuvieron 54 estudiantes, con los cuales se formaron 9 equipos; el 2015, tuvieron 46 estudiantes, con los cuales se formaron 8 equipos; y el 2016, tuvieron 54 estudiantes, con los cuales se formaron 9 equipos.

En la Figura 7, podemos ver las estadísticas de errores de compilación que presenta la investigación (6).

Figura 7: Estadísticas de errores de compilación

Table 2: Statistics of Build Errors for Each Category and Success Builds

	LOCAL				REMOTE			
Classification	2013	2014	2015	2016	2013	2014	2015	2016
C1: Dependency	34	14	18	34	86	118	64	18
C2: Syntax	12	6	12	7	0	10	0	0
C3: TypeMismatch	2	3	1	16	28	27	18	2
C4: Semantic	6	16	2	10	19	5	10	2
C5: Annotation	6	22	11	15	16	59	2	8
C6: Environment	280	81	253	249	0	0	0	0
Total # of C1 to C5	60	61	44	82	149	219	94	30
Total # of Success Builds	2395	1506	1670	2655	2309	1610	1764	2065

Fuente: (6)

2.2.4.3 Mantenibilidad

McCall hace referencia a la concisión, consistencia, modularidad, y autodescripción o autodocumentación para el factor de mantenibilidad (39). Por otra parte, la facilidad de análisis, cambio, pruebas y estabilidad son subcaracterísticas del modelo ISO 9126.

La documentación del IDE *Visual Studio* 2019 en (44) indica que se utilizan los intervalos de 0 a 100 para calcular el adecuado índice de mantenimiento de código, donde 0-9 es color rojo mantenimiento pobre; 10-19 es de color amarillo mantenimiento moderado; y de 20-100 es de color verde mantenimiento bueno. Estos utilizan la fórmula siguiente:

IC = MAX(0,(171 - 5.2 * ln(VH) - 0.23 * (CC) - 16.2 * ln(LC))*100 / 171).

Donde:

IC = Índice de mantenibilidad
VH = Volumen de Halstead
CC = Complejidad ciclomática
LC = Líneas de código

En (37), se afirma que la métrica de mantenibilidad es una de las más importantes, porque comprende la facilidad con la que el producto de *software* desarrollado se modificará, corregirá, adaptará y mejorará en su rendimiento (3). Esto puede verse como un reto y convertirse en un

problema en la mayoría de casos. Ante esto, las empresas desean que su *software* cumpla con los requerimientos de usuario y estándares de calidad. Esto produce que la complejidad aumente y que los sistemas sean difíciles de mantener (37).

2.3 Definición de términos básicos

CASE: Es un sistema de ingeniería de *software* asistida por computadora. Este es un programa que está creado con la finalidad de facilitar la creación de algún componente entregable de otro *software*.

Interfaz: Hace referencia a la interfaz gráfica del usuario –conocida también como GUI– que utiliza imágenes y objetos gráficos para que se muestre la información.

SQL *Server*: Es un potente sistema gestor de base de datos de Microsoft. Está diseñado para administrar y recuperar información. Tiene características como soporte de transacciones, escalabilidad, estabilidad y seguridad, soporte de procedimientos almacenados y otras.

***CSharp*:** Es un lenguaje de programación desarrollado y estandarizado por Microsoft. Es multiparadigma y es parte de la plataforma .NET, a la vez de estar aprobado como un estándar de ECMA e ISO.

***Script*:** Es una secuencia de comandos escritos en orden lógico. Los *scripts* SQL se utilizan para crear estructura de base de datos, operaciones para agregar, actualizar o eliminar registros. También, se utiliza *script* para cambiar la estructura de la base de datos.

GXC: Es el nombre de la herramienta propuesta. Proviene de la expresión "generador extremo de código" y se encarga de la generación de código fuente para *Entity Framework* en la presente tesis.

Métrica: Hace referencia a la forma de realizar la medición de los datos. Asimismo, son aquellos datos que son expresados en números para poder analizar la calidad de un código. Por ejemplo, está el índice de mantenibilidad, acoplamiento entre clases, complejidad ciclomática, etcétera.

Módulos: Es una porción de código de un programa. De las diferentes tareas que debe realizar un programa, se puede encargar de una de las tareas o de varias de ellas.

Eficiencia: Es la minimización del tiempo de procesamiento, reducción de recursos físicos del computador, que utiliza un programa para llevar a cabo todas las tareas para las que está creada.

Fiabilidad: Tiene como criterios a la tolerancia a errores, precisión, modularidad, simplicidad y exactitud de un código. Siempre y cuando un código cumpla con estos criterios, podemos decir que el código es fiable.

Mantenibilidad: Cuando hablamos de mantenibilidad, nos referimos a la facilidad para realizar cambios, concisión, consistencia, modularidad y autodescripción en el código escrito, esto para indicar que el código tiene facilidad de análisis. Se pueden realizar cambios y pruebas de manera simple.

Complejidad ciclomática: Esta es una métrica que mide la complejidad de la estructura de código y se crea al calcular el número de rutas de código en el flujo del programa.

CAPÍTULO III

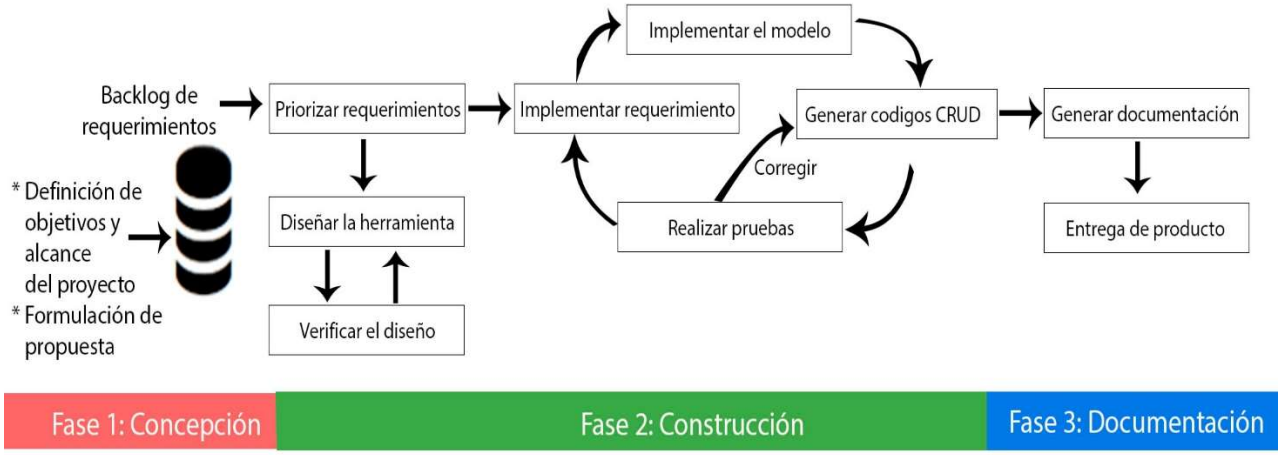
METODOLOGÍA

3.1 Metodología aplicada para el desarrollo de la solución

La metodología escogida para la investigación es una adaptación de *Discipline Agile Delivery* (DAD) (46). Esta fue elegida, ya que se adapta a equipos pequeños. Además, es ágil y flexible. Asimismo, la metodología DAD es compatible con el modelo *Backlog* de SCRUM, ya que facilita la concepción de un producto mínimo viable en un periodo corto de tiempo y en cada iteración de su fase de construcción.

En ese sentido, la Figura 8 muestra el ciclo de vida del proyecto con la adaptación que realizamos a la metodología DAD. Al adaptar la metodología DAD al proyecto, se reemplazó la fase de transición por la fase de documentación. A su vez, se incluyó un ciclo de iteración para validar la herramienta GXC y otro para implementar cada requerimiento de diseño.

Figura 8: Ciclo de vida del proyecto adaptado a la metodología DAD



En la Tabla 8, especificamos las 3 fases del proyecto con la adaptación respectiva a la metodología DAD. De esta forma, podemos ver las etapas y el listado de artefactos esperados como resultado de cada etapa.

Tabla 8. Especificación de fase, etapas y artefactos esperados

Fase	Etapa	Artefactos esperados
Concepción	Estudio de formas de realizar los CRUD en el lenguaje C#.	Código fuente de formas de generación de los CRUD con el lenguaje C#, que pueda ser utilizada como referencia para el diseño de la herramienta.
	Designación de roles al equipo de trabajo	Equipo de trabajo con roles definidos en la construcción de la herramienta.
	Definición y especificación de requerimientos	Documento de especificación de requerimientos de la herramienta GXC
Construcción	Diseño de la herramienta GXC	Documento de diseño de la herramienta GXC
	Implementación de la herramienta GXC	Instalador de la herramienta GXC
	Pruebas y refinamiento	Documento con reporte de pruebas y validación de la implementación
Documentación	Construcción de manuales y documentación	Manual de instalación y de usuario
		Repositorio en GITHUB
		Documento de memoria del trabajo de grado

CAPÍTULO IV

ANÁLISIS Y DISEÑO DE LA SOLUCIÓN

Para el análisis y diseño de la herramienta GXC, consideraremos la parte de la fase 1 del ciclo de vida del proyecto adaptado a la metodología DAD, que viene a ser la fase de concepción.

4.1 Designación de roles

En la presente tesis, el autor es quien desempeña todos los roles en el desarrollo. Luego, se tiene como colaboradores eventuales a dos personas que son parte del equipo. En la Tabla 9, se puede ver la designación de roles de los integrantes que participarán en la investigación.

Tabla 9. Designación de roles

Id de cargo	Cargo / Rol	Persona encargada	Código
01	Dueño del producto	Juan Moises Rojas Torres	JMRT
02	Equipo de trabajo	Edwin Alberto Ramírez Zamata	EARZ
		Ricardo Adolfo Dávila Valencia	RADV
		Juan Moises Rojas Torres	JMRT

4.2 Alcance general

4.2.1 Alcance del producto:

Dentro de los alcances del producto, se establece lo siguiente:

- La herramienta permite la conexión a base de dato SQL *Server* para cargar la información de las bases de datos.
- La herramienta muestra la información de la base de datos y permite agregar esquemas, tablas y columnas.
- La herramienta permite generar código para los *view models* y *controllers* de los procesos CRUD.

4.3 *Backlog* de requerimientos

Se realizó un *Backlog* con los 11 requerimientos considerados por el equipo. Esto lo podemos ver en la Tabla 10.

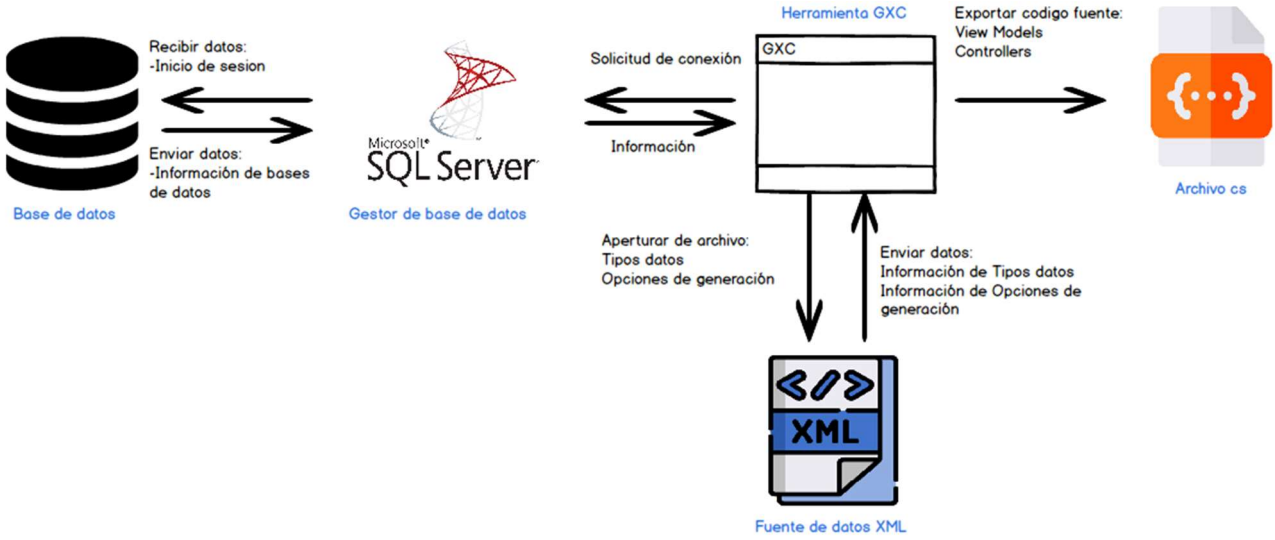
Tabla 10. *Backlog* de requerimientos

Identificador (ID)	Requerimiento
2021-0001	Realizar conexión con el gestor de base de datos SQL Server
2021-0002	Mostrar las bases de datos del servidor conectado
2021-0003	Recuperar información de las tablas de la base de datos para agregar
2021-0004	Agregar tablas para generar información
2021-0005	Exportar la estructura de la base de datos cargada
2021-0006	Cargar información de tablas para generar código fuente
2021-0007	Cargar equivalentes de tipo de dato para SQL *Server* y para *CSharp*
2021-0008	Mostrar las opciones de generación de código fuente
2021-0009	Generar código fuente según opción seleccionada
2021-0010	Exportar código fuente generado
2021-0011	Seleccionar varias tablas para generar código fuente

4.4 Arquitectura de la solución

La arquitectura de la solución del proyecto se muestra en la Figura 9, en la cual podemos ver que la herramienta interactúa con el gestor de base de datos Microsoft SQL *Server*, y así recuperar la información de esquemas, bases de datos, tablas, columnas, tipos de datos, claves primarias y claves foráneas. También, interactúa con archivos .XML generados en el desarrollo de la solución con la finalidad de recuperar información de tipos de dato y opciones de generación de código. El usuario cuenta con la opción de exportar el código fuente generado en un archivo .CS.

Figura 9: Arquitectura de solución del proyecto



En la Tabla 11, se muestra la utilidad de las DLL principales utilizadas en la construcción de la herramienta GXC.

Tabla 11. DLL principales en GXC

Namespace	*Assembly*	Utilidad para GXC
System.IO	*System.IO.FileSystem*.dll	Crear archivos *viewmodel, viewmodel list* y *controller*
System.Data	*System.Data*.dll	Abre y cierra la conexión a la base de datos
System.Data.SqlClient	*System.Data.SqlClient*.dll	Recuperar la información de las bases de datos
System.Xml	*System.Xml*.dll	Leer y crear los archivos XML
System.Linq	*System.Linq*.dll	Ordenar datos, mostrar información de registros

4.5 Planificación de *sprints*

En la Tabla 12, por cada requerimiento funcional, se muestra la descripción de prioridades, dimensión, *sprint* y responsable. Está ordenado de acuerdo al número de *sprint*.

Tabla 12. Descripción de prioridades

Identificador (ID)	Estado	Dimensión / Esfuerzo	Sprint	Prioridad	Responsable
2021-0001	Planificado	2 días	1	5	EARZ
2021-0002	Planificado	3 días	1	4	JMRT
2021-0003	Planificado	5 días	2	5	JMRT
2021-0004	Planificado	4 días	2	3	JMRT
2021-0005	Planificado	3 días	2	2	RADV
2021-0006	Planificado	3 días	3	4	JMRT
2021-0007	Planificado	3 días	3	2	JMRT
2021-0008	Planificado	3 días	3	2	RADV
2021-0009	Planificado	10 días	4	5	JMRT
2021-0010	Planificado	3 días	5	4	RADV
2021-0011	Planificado	3 días	5	3	EARZ

En la Tabla 13, mostramos la descripción de cada valor de las prioridades.

Tabla 13. Descripción de valores de prioridad

Valor	Descripción
1	Muy baja
2	Baja
3	Media
4	Alta
5	Urgente

4.6 Diseño de interfaces

Para el diseño de interfaces, elaboramos los *mockups* de la herramienta propuesta, los cuales mostramos a continuación:

Podemos ver en la Figura 10 Formulario Base de Datos, donde cada uno de los elementos se ha identificado con números, los cuales son los siguientes:

Número 1: se debe realizar la conexión al gestor de base de datos SQL *Server* (servidor).
Número 2: se debe poder elegir el tipo de autenticación. Esta elección puede ser mediante el modo de autenticación de Windows o con el modo de autenticación de Windows y SQL *Server*.

Número 3: se debe elegir la base de datos con la cual se va a trabajar para obtener los datos.

Número 4: si el modo de autenticación es Windows y SQL *Server*, se debe registrar el usuario.

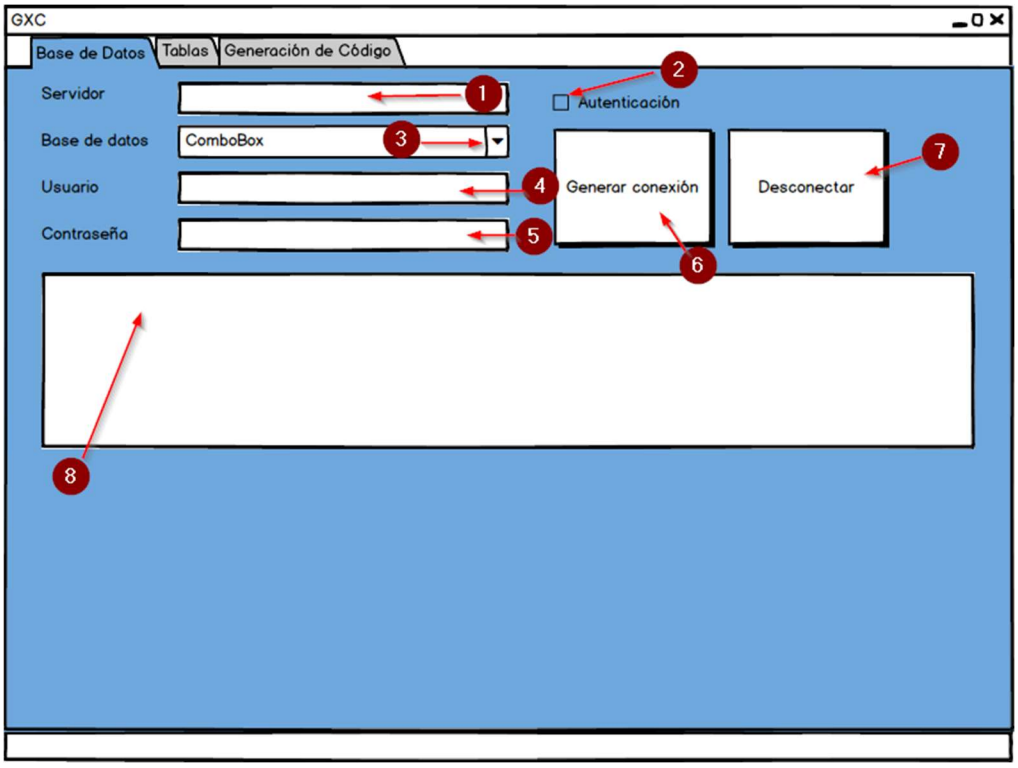
Número 5: si el modo de autenticación es Windows y SQL *Server*, se debe registrar la contraseña.

Número 6: el botón "generar conexión" debe permitir conectar y generar la línea de código de conexión al servidor.

Número 7: el botón "desconectar" debe realizar lo que su nombre indica, desconectar la herramienta del servidor.

Número 8: se debe mostrar la cadena de conexión.

Figura 10: Formulario Base de Datos



El formulario tablas debe requerir que exista una conexión en curso para poder cargar la información de la base de datos. En este formulario, se han numerado los elementos para facilitar la descripción. Estos son los siguientes:

Número 1: se debe poder agregar esquema.

Número 2: se debe agregar el nombre de la tabla.

Número 3: se debe dar clic en el botón "agregar tabla".

Número 4: se debe agregar el nombre de la columna.

Número 5: se debe seleccionar el tipo de dato de la columna.

Número 6: se debe agregar el tamaño de la columna si es necesario para el tipo de dato.

Número 7: se debe elegir si la columna es nula.

Número 8: se debe elegir si la columna es identidad.

Número 9: si la columna es identidad se pone el número de incremento.

Número 10: se debe elegir si la columna es parte de algún tipo de llave.

Número 11: se elige la tabla de referencia si es clave foránea.

Número 12: se agrega la columna.

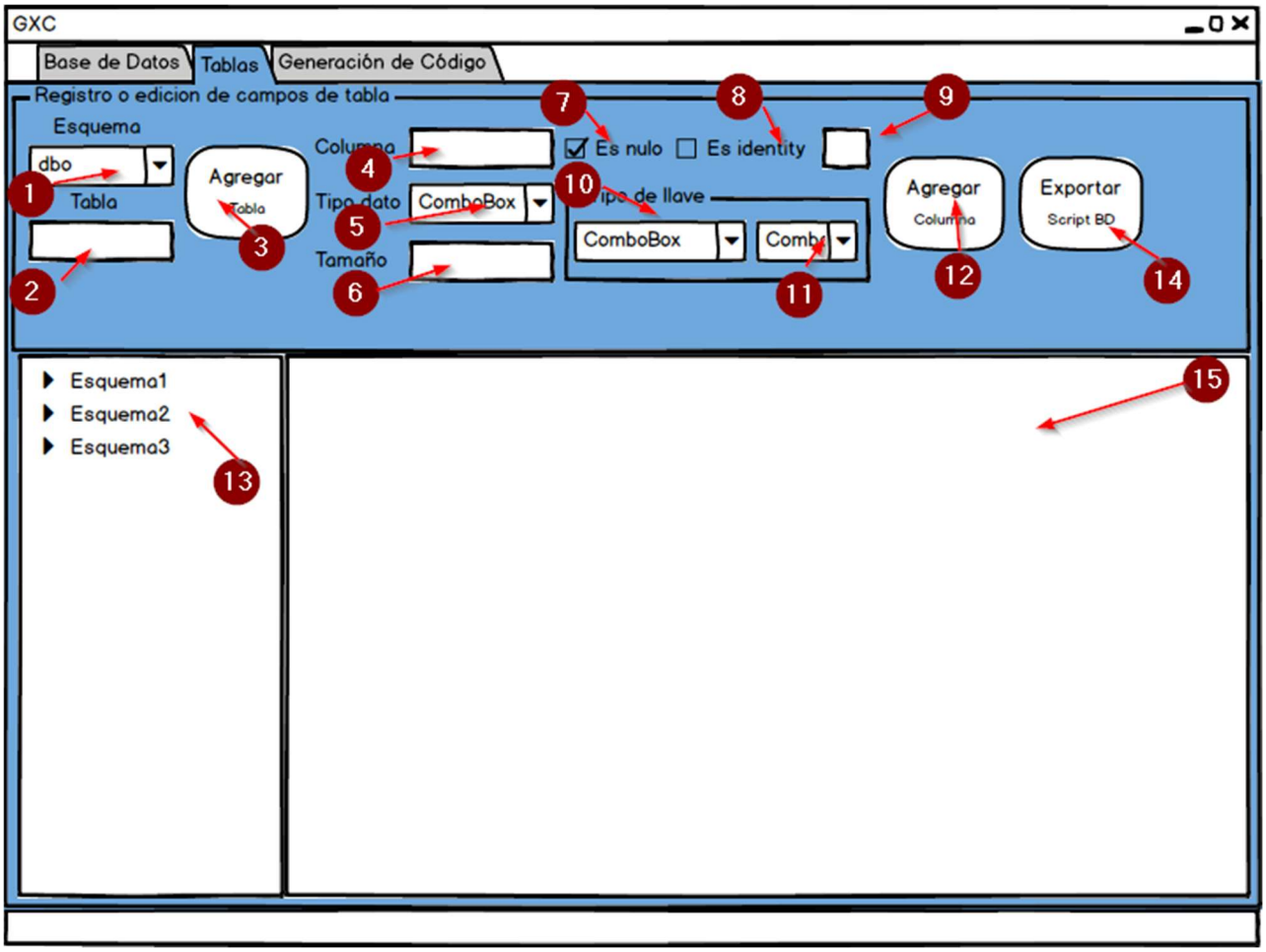
Número 13: se debe mostrar la información de la base de datos en caso exista alguna conexión abierta.

Número 14: también se debe poder exportar el *script* de la base de datos.

Número 15: se puede ver el código de exportación de la base de datos.

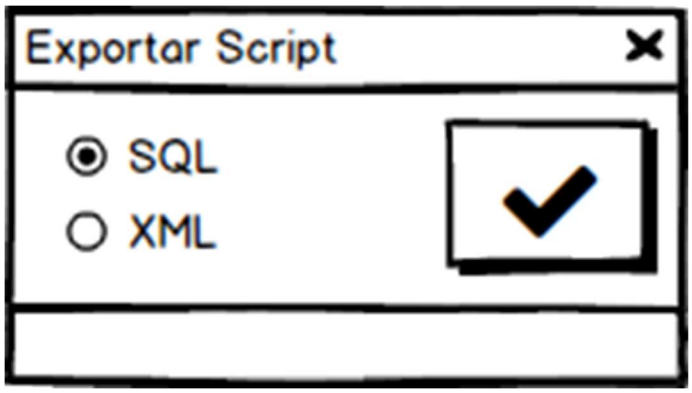
Lo mencionado anteriormente lo podemos ver en la Figura 11.

Figura 11: Formulario Tablas



El formulario Exportar *Scrip*t, que se muestra en la Figura 12, interactúa con el formulario Tablas y solo confirma el tipo de *script* con el que quiere realizar la exportación de la base de datos. Dicho código corresponde a la capa de datos.

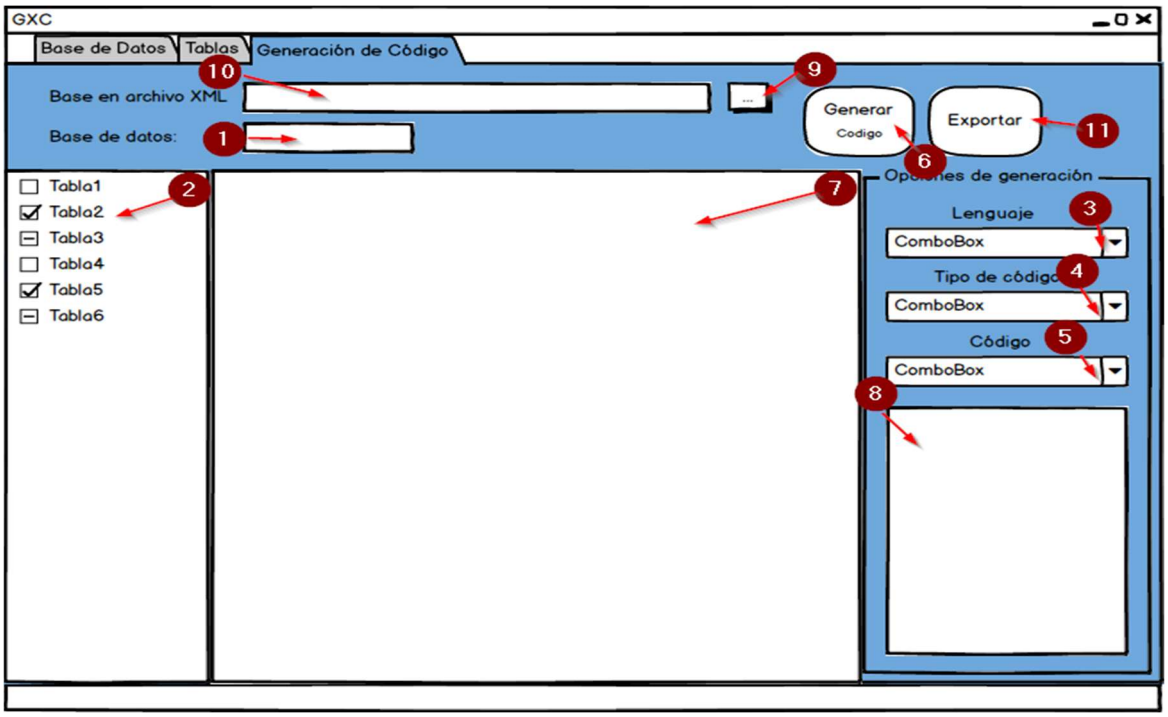
Figura 12: Formulario Exportar *Script*



Para la generación de código fuente, tenemos el formulario de generación de código, el que se muestra en la Figura 13. Podemos ver la siguiente enumeración:

Número 1: se debe poder ver el nombre de la base datos para la cual se va a generar el código.
Número 2: se debe poder ver el nombre de las tablas para elegir la generación de código.
Número 3: se debe poder elegir el lenguaje para el cual se desea generar.
Número 4: se elige el tipo de código para el cual se desea generar.
Número 5: se debe elegir qué código se va a generar.
Número 6: se debe dar clic en el botón “generar” para que se pueda generar el código.
Número 7: es el *textboxt* donde se muestra el código generado.
Número 8: se debe poder ver el tiempo de generación de código.
Número 9: se debe poder cargar información del formato XML exportado en el formulario Tablas.
Número 10: se debe visualizar la ruta del archivo XML.
Número 11: con el botón “exportar”, se debe poder obtener los archivos de *controller* y *view model*.

Figura 13: Formulario Generación de Código



4.7 Validación de diseños

Para validar los *mockups*, se realizó una validación como se muestra en la Tabla 1414, donde se comparó la utilización y el seguimiento de las interfaces diseñadas con las historias de usuario.

Tabla 14. Validación de interface con los requerimientos

Id de requerimiento	Requerimiento	Secuencia de interfaces
2021-0001	Realizar conexión con el gestor de base de datos SQL *Server*	I1
2021-0002	Mostrar las bases de datos del servidor conectado	I1-> lista_base_datos (*ComboBox*)
2021-0003	Recuperar información de las tablas de la base de datos para agregar	I1->I2->lista_Información (*TreeView*)
2021-0004	Agregar tablas para generar información	I2
2021-0005	Exportar estructura de la base de datos cargada	I2->I3
2021-0006	Cargar información de tablas para generar código fuente	I1->I2->I4
2021-0007	Cargar equivalentes de tipo de dato para SQL *Server* y para *CSharp*	I4
2021-0008	Mostrar las opciones de generación de código fuente	I4
2021-0009	Generar código fuente según opción seleccionada	I4
2021-0010	Exportar código fuente generado	I4->Archivo cs
2021-0011	Seleccionar varias tablas para generar código fuente	I4

La descripción de los valores usados en la secuencia de interfaces en la Tabla 12 la podemos ver en la Tabla 15.

Tabla 15. Descripción de interfaces

Valor	Descripción
I1	Formulario Base de Datos
I2	Formulario Tablas
I3	Formulario Exportar Script
I4	Formulario Generación de Código

4.8 Diagrama de navegación para generar código fuente

En la Figura 14, se muestra el diagrama de navegación general de la herramienta. En esta, podemos ver cómo interactúa cada formulario.

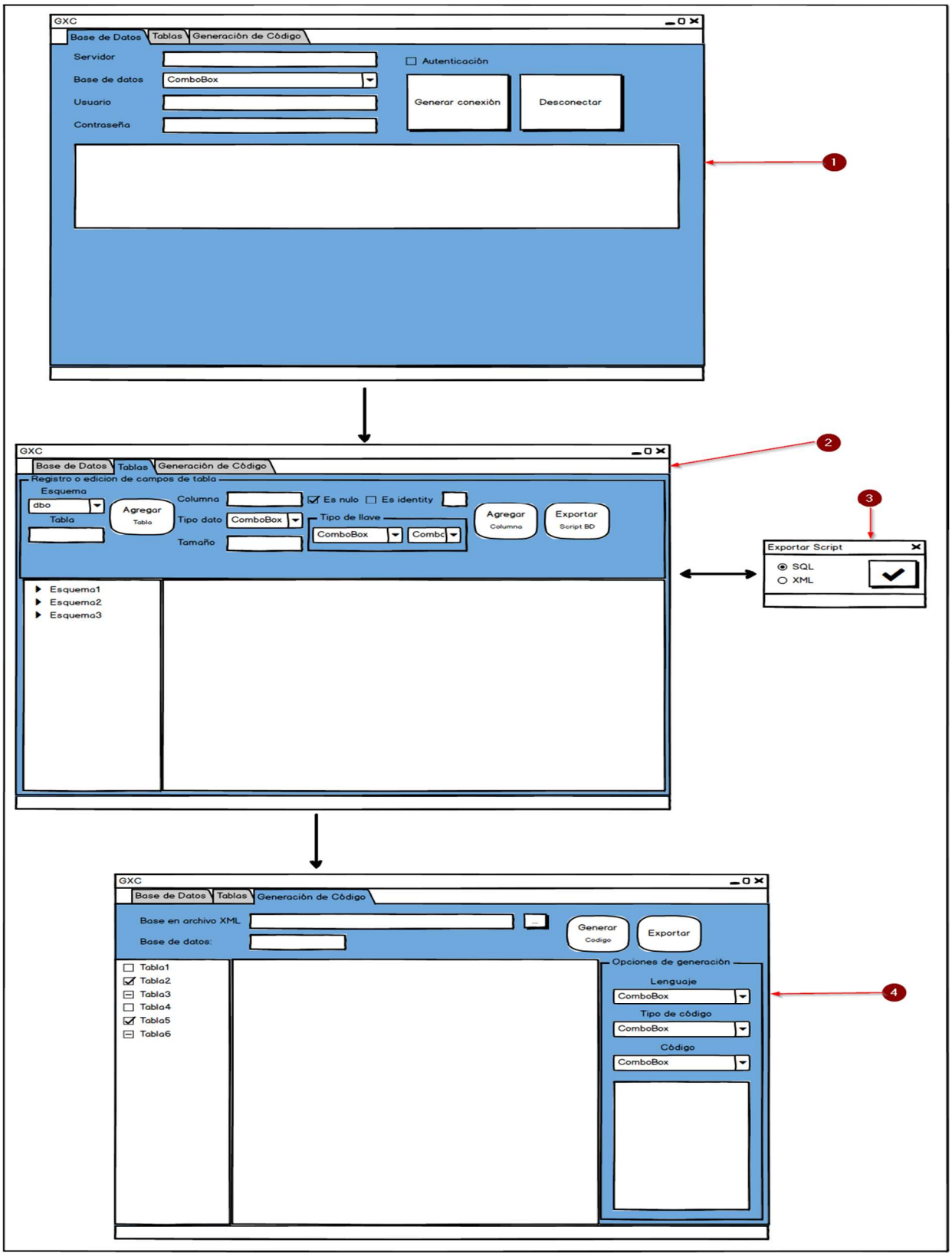
Número 1: podemos ver la pestaña “base de datos”, que es el primer formulario que se ve dentro del diagrama de navegación general. De este formulario, se debe pasar a la pestaña “tablas”.

Número 2: el formulario Tablas obtiene información de la base de datos cuando existe una conexión abierta previamente desde el formulario Base de datos.

Número 3: el formulario Exportar puede o no exportar la información de la base de datos en formato SQL o XML. Esto depende del usuario cuando este decide utilizar la opción.

Número 4: En el formulario Generación de Código, se podrán seleccionar las tablas y generar el código fuente que se necesite.

Figura 14: Diagrama de navegación general



CAPÍTULO V

CONSTRUCCIÓN

5.1 Construcción

5.1.1 Arquitectura de la herramienta y DLL

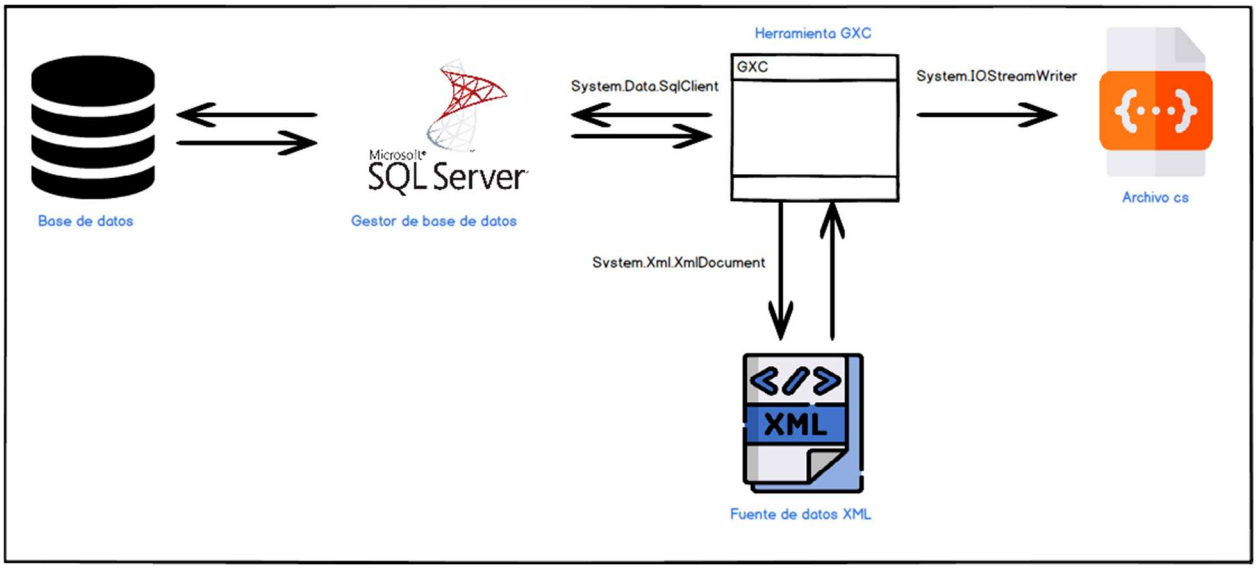
Durante la construcción de la herramienta, se utilizaron diferentes referencias mediante las DLL. Para la conexión y recuperación de información de las bases de datos, se utilizó la DLL *System.Data.SqlClient*. De esta manera, pudimos recuperar la información de los esquemas, tablas, columnas, tipos de datos, campos *identity*, tipos de llaves y otros datos necesarios para poder generar el código fuente. La representación la podemos ver en la Figura 15. La DLL *System.Data* se encarga de abrir y cerrar la conexión con la base de datos.

Luego, para recuperar la información del archivo Opciones_De_Generacion.xml y del archivo TiposDato.xml, se utilizó la DLL *System.Xml.XmlDocument*, lo que muestra, de esta manera, información para las opciones lenguaje, tipo de código y código para generar el código fuente. También, se utiliza para recuperar los tipos de dato SQL *Server* y su equivalente en *CSharp*.

En el proceso de exportación de código fuente generado, fue utilizada la DLL *System.IO.StreamWriter*. De esta manera, se pudo exportar el código de base de datos, el código de *ViewModels* y el de *Controller*.

Sobre lo afirmado, se realizó una representación gráfica de la arquitectura de la herramienta GXC que se construyó para realizar la generación de código.

Figura 15: Arquitectura de herramienta



5.1.2 Conexión a base de datos

La conexión a la base de datos se realiza para extraer la información de las bases de datos existentes. En este caso, nosotros trabajamos con la base de datos BDHIS_WEB. Para ello, utilizamos una conexión nativa mediante la DLL *System.Data.SqlClient*.

Asimismo, se tiene la variable oConexion de tipo *SqlConnection*, que es la responsable de abrir y cerrar la conexión con la base de datos máster del gestor de base de datos SQL *Server*. Esta variable se muestra en la Figura 16.

Figura 16: Fragmento de código para realizar la conexión

```
protected SqlConnection oConexion;
2 referencias
private void Conexion_Master()
{
    //Establecemos la conexion por Autenticación de windows
    oConexion = new SqlConnection("Data Source=" + txtServidor.Text + ";Initial Catalog=master;Integrated Security=True");
}
1 referencia
private void Conexion_BD()
{
    //Establecemos la conexion por Autenticación de windows
    oConexion = new SqlConnection(txtCadenaConexion.Text);
}
```

5.1.3 Conexión a archivo XML opciones de generación de código

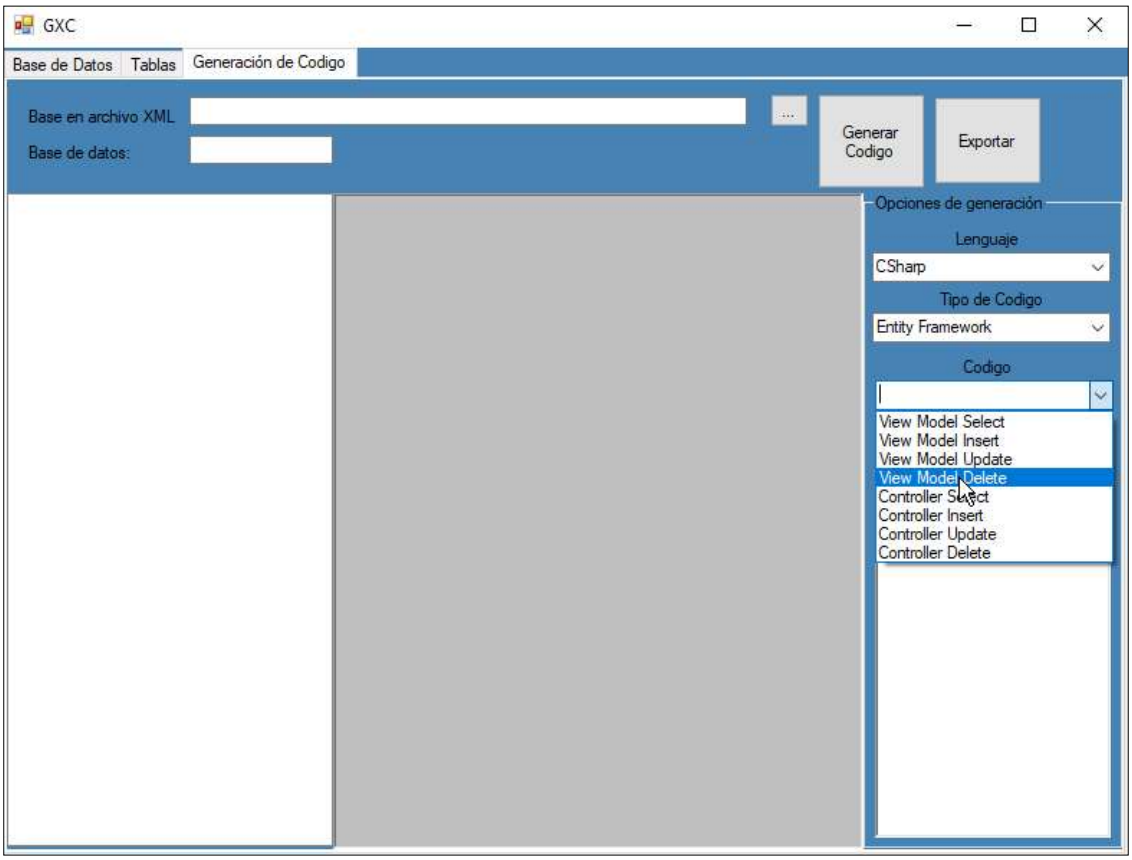
Para recuperar las opciones de generación de código que se muestran en el formulario de generación de código, tal como se ve en la Figura 18, utilizamos como fuente de datos el archivo XML Opciones_De_Generacion.xml, que tiene las etiquetas <datos>, <lenguaje>, <tipocodigo> y <codigo>. La estructura del archivo XML permite recuperar la opción de lenguaje, tipo de código y código, tal como se muestra en la Figura 17.

Figura 17: Porción de XML Opciones_De_Generacion.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<datos>
  <lenguaje name="CSharp">
  <tipocodigo lenguaje="CSharp" name="Entity Framework">
      <codigo tipocodigo="Entity Framework">View Model Select</codigo>
      <codigo tipocodigo="Entity Framework">View Model Insert</codigo>
      <codigo tipocodigo="Entity Framework">View Model Update</codigo>
      <codigo tipocodigo="Entity Framework">View Model Delete</codigo>
      <codigo tipocodigo="Entity Framework">Controller Select</codigo>
      <codigo tipocodigo="Entity Framework">Controller Insert</codigo>
      <codigo tipocodigo="Entity Framework">Controller Update</codigo>
      <codigo tipocodigo="Entity Framework">Controller Delete</codigo>
    </tipocodigo>
```

El primer paso para generar el código fuente en el formulario Generación de Código se realiza al elegir el tipo de código que se quiere generar. Como opciones, tenemos *View Model Select*, *View Model Insert*, *View Model Update*, *View Model Delete*, *Controller Select*, *Controller Insert*, *Controller Update* y *Controller Delete*. Lo mencionado anteriormente lo podemos ver en la Figura 18.

Figura 18: Formulario Generación de Código con opciones de generación



5.1.4 Conexión a archivo Tipos de dato

Para tener los equivalentes entre tipos de dato que se manejan en *CSharp* y SQL *Server*, se tuvo que construir un archivo XML de nombre TiposDato.xml. De este, se extraen los datos para generar el código correctamente. En las Figuras 19 y 20, vemos fragmentos del archivo XML, tanto para SQL *Server* como para *CSharp*. Ambos archivos se relacionan por medio del Id. De esta manera, podemos conseguir los equivalentes correctos.

Figura 19: Fragmento de código del archivo TiposDato.xml - SQL *Server*

```
<?xml version="1.0" encoding="utf-8" ?>
<tiposdato>
    <tipo id="1" nametype="SQLServer">bigint</tipo>
    <tipo id="2" nametype="SQLServer">binary</tipo>
    <tipo id="3" nametype="SQLServer">bit</tipo>
    <tipo id="4" nametype="SQLServer">char</tipo>
    <tipo id="5" nametype="SQLServer">date</tipo>
    <tipo id="6" nametype="SQLServer">datetime</tipo>
    <tipo id="7" nametype="SQLServer">datetime2</tipo>
    <tipo id="8" nametype="SQLServer">datetimeoffset</tipo>
```

Figura 20: Fragmento de código del archivo TiposDato.xml - *CSharp*

```
<tipo id="1" nametype=".Net Framework">long</tipo>
<tipo id="2" nametype=".Net Framework">byte[]</tipo>
<tipo id="3" nametype=".Net Framework">boolean</tipo>
<tipo id="4" nametype=".Net Framework">string</tipo>
<tipo id="5" nametype=".Net Framework">DateTime</tipo>
<tipo id="6" nametype=".Net Framework">DateTime</tipo>
<tipo id="7" nametype=".Net Framework">DateTime</tipo>
<tipo id="8" nametype=".Net Framework">DateTimeOffset</tipo>
```

En la Tabla 16, mostramos las equivalencias entre el tipo de dato que se utiliza en *SqlServer* y *.Net Framework*. Ambos se identifican por el Id.

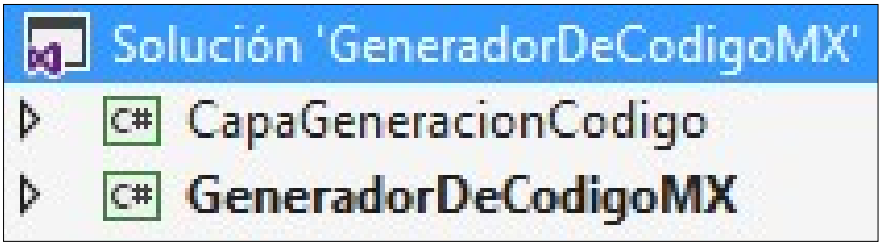
Tabla 16. Equivalencias de tipo de dato *SqlServer* y *.Net Framework*

Id	SqlServer	.Net Framework	Id	SqlServer	.Net Framework
1	bigint	long	17	numeric	decimal
2	binary	byte[]	18	nvarchar	string
3	bit	boolean	19	real	single
4	char	string	20	rowversion	byte[]
5	date	DateTime	21	smalldatetime	DateTime
6	datetime	DateTime	22	smallint	int
7	datetime2	DateTime	23	smallmoney	decimal
8	datetimeoffset	DateTimeOffset	24	sql_variant	Object
9	decimal	decimal	25	text	string
10	varbinary(max)	byte[]	26	time	TimeSpan
11	float	double	27	timestamp	byte[]
12	image	byte[]	28	tinyint	byte
13	int	int	29	uniqueidentifier	guid
14	money	decimal	30	varbinary	byte[]
15	nchar	string	31	varchar	string
16	ntext	string	32	xml	xml

5.1.5 Generación de código fuente

Para poder realizar este proceso, dividimos el proyecto en dos capas, las cuales se especifican en dos archivos: CapaGeneracionCodigo y GeneradorDeCodigoMX, como podemos ver en la Figura 21.

Figura 21: Capas de la solución



Dividir en dos capas nos brinda como resultado que la capa generación de CapaGeneracionCodigo se encarga de devolver el código fuente generado mediante un *string*, esto a partir del tipo de código que se eligió para la generación, como muestra en las opciones la Figura 18.

La capa CapaGeneracionCodigo es la encargada de devolver el texto con el código generado. En la Figura 22, podemos ver como ejemplo el método *ViewModelList que corresponde a la capa de negocio*, que recibe como parámetro de entrada los siguientes elementos:

- BD, que es el nombre de la base de datos
- Tabla, que es el nombre de la tabla seleccionada para generar el código
- Columnas, que son las columnas o campos de la tabla seleccionada.

Figura 22: Código para generar el *View Model List* - CapaGeneracionCodigo

```
public string ViewModelList(string BD, string Tabla, string Columnas)
{
    string Apertura = "";
    string Cierre = "";
    Apertura =
    "using System;\n"+
    "namespace " +BD+"EntityFramework.Models.ViewModels\n"+
    "{\n"+
    "    public class List"+Tabla+"ViewModel\n"+
    "    {";
    Cierre = "    }\n"+
    "}";
    return Apertura + "\n" + Columnas + "\n" + Cierre;
}
```

La capa GeneradorDeCodigoMX se encarga del proceso fuerte al momento de generar el código fuente, en vista de que esta recupera el nombre de la base de datos para enviar al método elegido. Para continuar con el ejemplo de la Figura 22, el método Generar_*ViewModel_List*_Una_Tabla llama al método *ViewModelList* para devolver el *script* generado. Dicho código será parte de la capa de negocio. Este método recibe como parámetros de entrada los siguientes elementos:

- Tabla (nombre de la tabla)
- Exportar, que viene a ser *true* si queremos exportar los archivos, o *false* si solo queremos generar *script*.

En ese sentido, este método recupera la información de la tabla (columnas, campo de identidad, claves foráneas y si la columna puede ser un valor nulo), realiza un recorrido *while* para ir recuperando la estructura de cada columna y, posteriormente, hace el llamado al método *ViewModelList* para devolver el *script* generado. En la Figura 23, podemos ver el proceso de generación del método Generar_ViewModel_List_Una_Tabla.

Figura 23: Generación de *View Model List* de una tabla - GeneradorDeCodigoMX

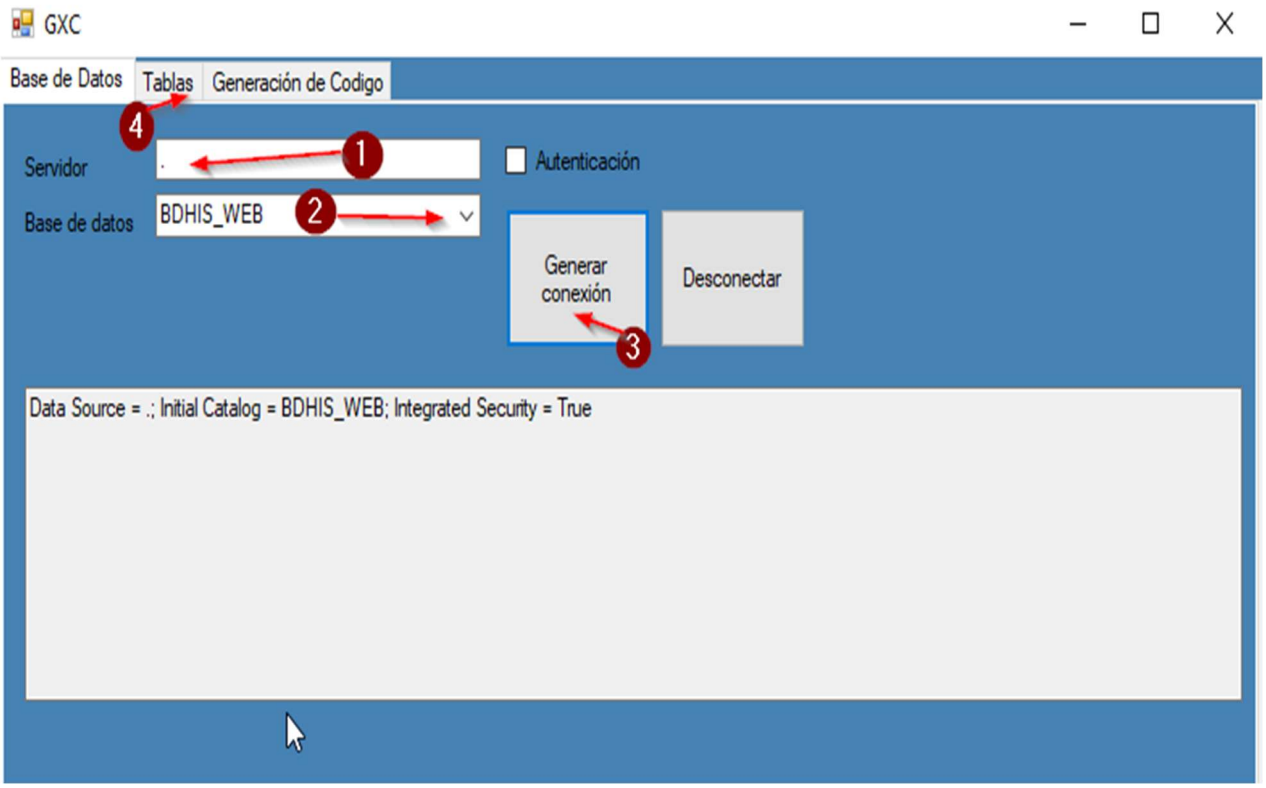
```
private string Generar_ViewModel_List_Una_Tabla(string Tabla,bool Exportar)
{
    string Script = "";
    oCommand = new SqlCommand("sp_Datos_Para_Listar_VM", oConexion);
    oCommand.CommandType = CommandType.StoredProcedure;
    oCommand.Parameters.AddWithValue("@Tabla", Tabla);
    oConexion.Open();
    SqlDataReader oDataReader = oCommand.ExecuteReader();
    string BD = txtBaseDeDatos.Text.Trim();
    string Nombre_Tabla = "";
    string Columna = "";
    string Columnas = "";
    string TipoDato_Actual = "";
    string TipoDato_Actual_CSharp;
    while (oDataReader.Read() == true)
    {
        Nombre_Tabla= ConvertirPrimeraLetraMayuscula(oDataReader["Table_Name"].ToString());
        //Cargamos los tipos de dato
        TipoDato_Actual = oDataReader["Column_Data_Type"].ToString();
        //Buscamos el tipo de dato para c#
        TipoDato_Actual_CSharp = BuscarTipoDatoCSharp("SQLServer", TipoDato_Actual);
        //Verificamos si permite nulos
        if (oDataReader["Is_Nullable"].ToString() == "1" && TipoDato_Actual_CSharp!="string")
        {
            TipoDato_Actual_CSharp = TipoDato_Actual_CSharp + "?";
        }
        //Recuperamos la columna
        Columna= ConvertirPrimeraLetraMayuscula(oDataReader["Column_Name"].ToString());
        //Juntamos la columna para el view model
        if (Columnas=="")
        {
            Columnas = "        public " + TipoDato_Actual_CSharp + " " + Columna + " { get; set; }";
        }
        else
        {
            Columnas= Columnas+"\n"+ "        public " + TipoDato_Actual_CSharp + " " + Columna + " { get; set; }";
        }
    }
    oConexion.Close();
    Script=oEntityFramework.ViewModelList(BD, Nombre_Tabla, Columnas);
    //Verificamos si se va a exportar
    if(Exportar==true)
    {
        oExportar.ViewModelList(Nombre_Tabla,Ruta_A_Exportar,Script);
    }
    return Script;
}
```

A continuación, mostraremos gráficamente el proceso de generación de los CRUD, tanto para exportar y para solo generar el *script*.

Con el fin de generar código CRUD, primero debemos ubicarnos en el formulario Base de Datos, que es donde se inicia el proceso. En la Figura 24, se enumeran los pasos a seguir. El ejemplo se realiza con el modo de autenticación de Windows.

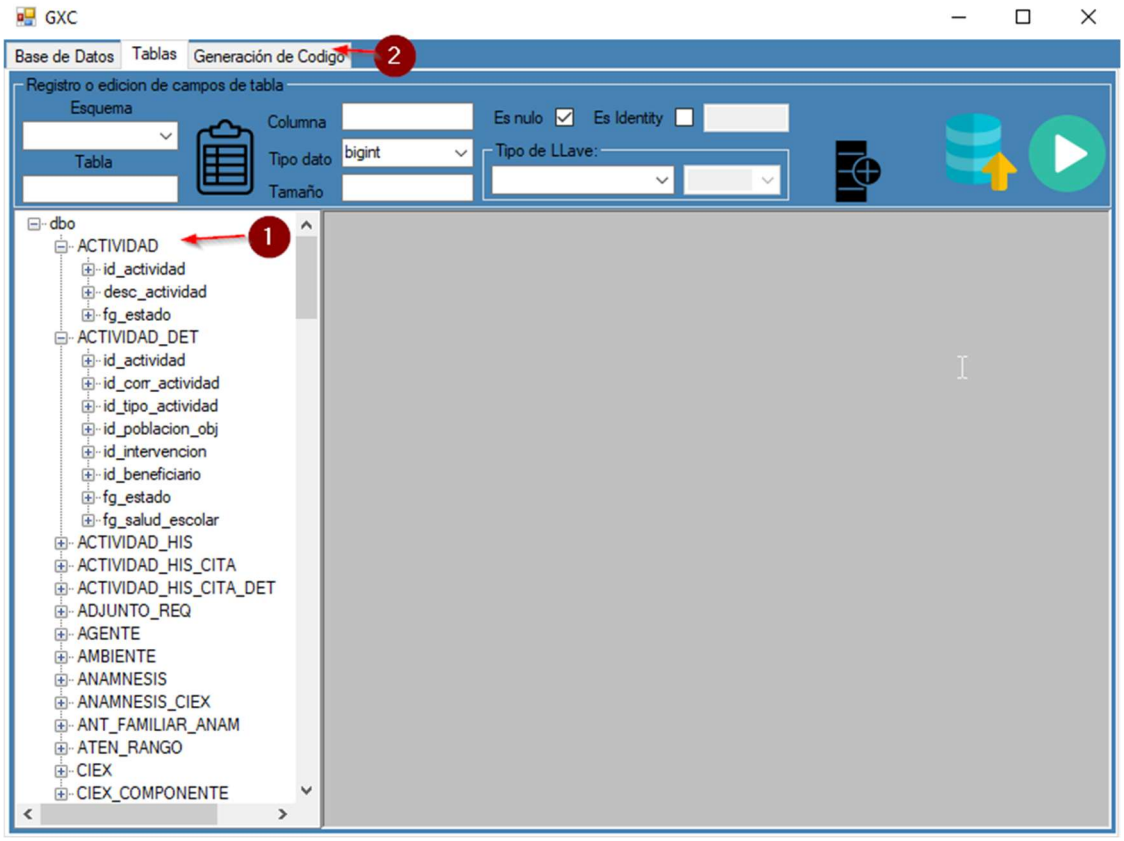
En los siguientes enunciados, describiremos el proceso. En el elemento número 1, se especifica el servidor al que nos vamos a conectar. En el número 2, se elige la base de datos a la cual nos vamos a conectar. Asimismo, el número 3 indica que debemos dar clic en el botón “Generar conexión”; y, por último, en el número 4, se dará clic en “Tablas” para pasar al formulario siguiente y, así, continuar el proceso de generación.

Figura 24: Primer paso para iniciar el proceso de generación de código - Base de Datos



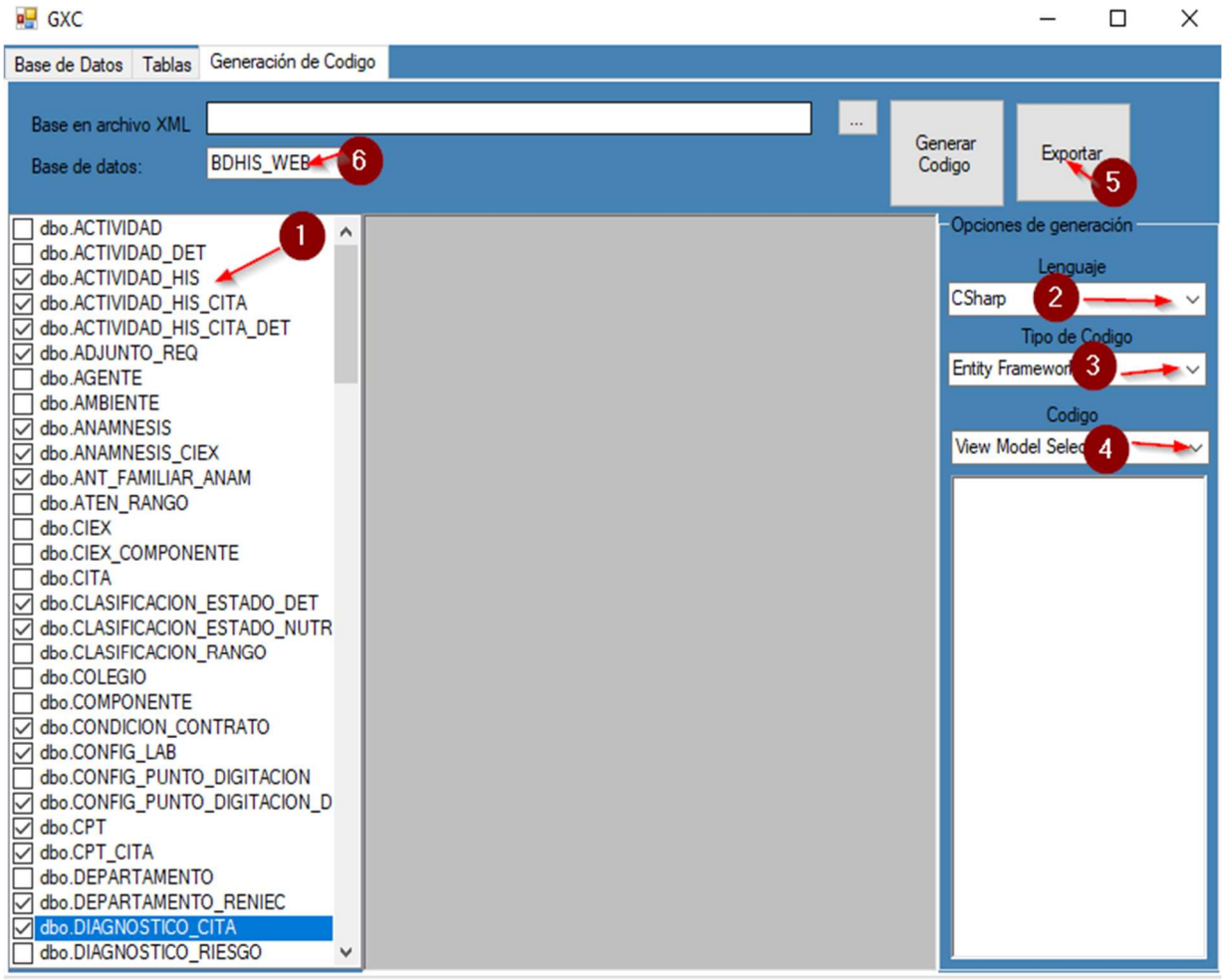
Para continuar con la generación de código, una vez que dimos clic en “Tablas”, este nos va a mostrar el formulario “Tablas”, como se puede apreciar en la Figura 25. En este paso, debemos verificar que encuentre la información de la base de datos en el *TreeView* (esquemas, tablas, columnas, tipo de dato, condición nula, tipo de clave), tal como muestra el número 1, y, posteriormente, debemos dar clic en “Generación de Código”, como lo muestra el número 2, para continuar con la generación de código.

Figura 25: Segundo paso para generar código - Tablas



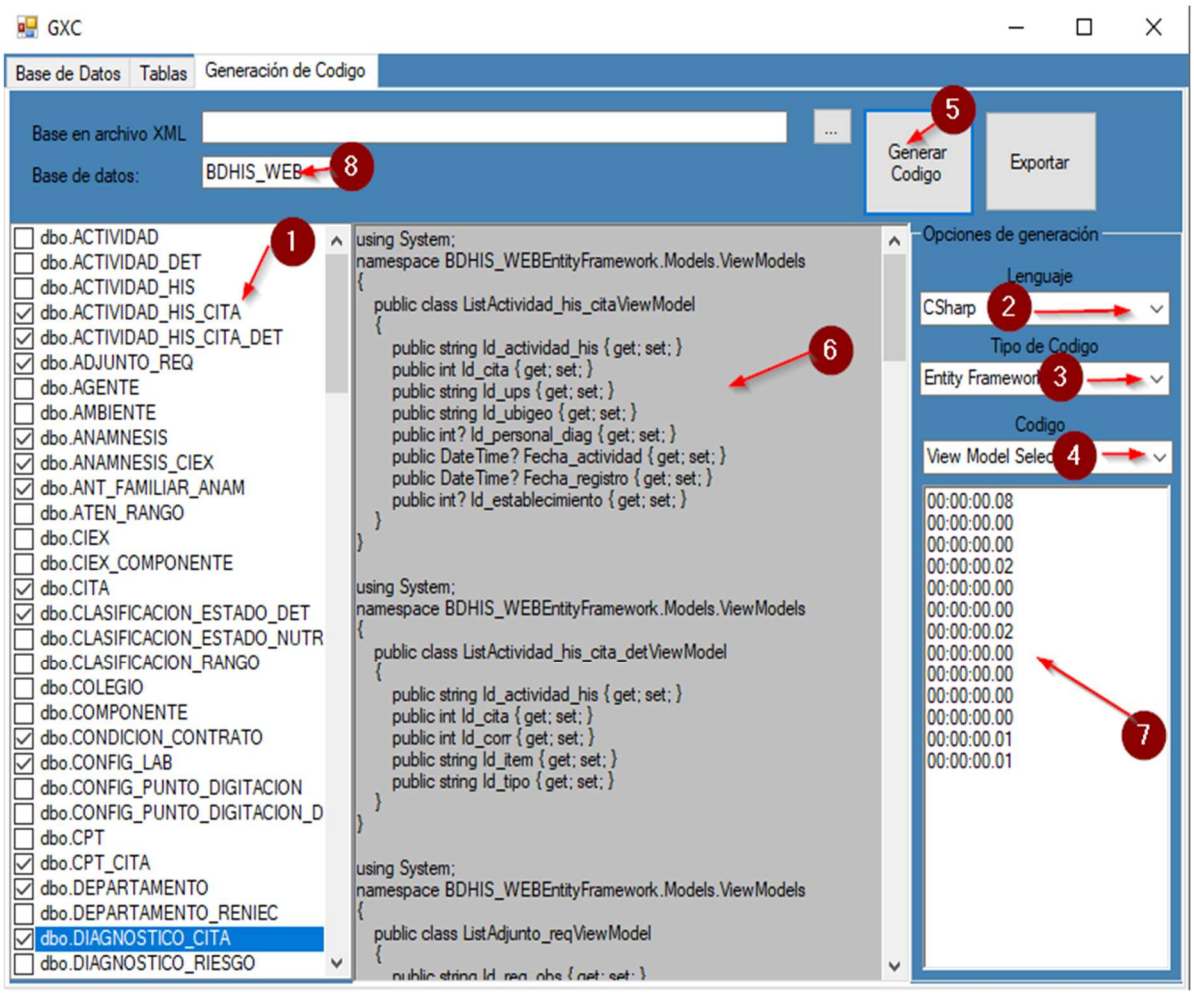
Como tercer y último paso, en el formulario Generación de Código, vamos a elegir cómo queremos generar el código. Para el ejemplo de exportar, podemos ver la Figura 26. En el elemento número 1, se deben elegir las tablas para las cuales deseamos generar código; en el número 2, elegimos el lenguaje; en el número 3, elegimos el tipo de código; en el número 4, elegimos el código que deseamos exportar; y, en el número 5, se debe dar clic en el botón “Exportar” para poder obtener los archivos .cs. Un ejemplo de los archivos exportados, según el código elegido, lo podemos ver en la Figura 27 y en la Figura 28. Para ver las opciones de código a exportar, podemos ver la Figura 18. Finalmente, en el número 6, podemos ver el nombre de la base de datos con la que se está generando el código.

Figura 26: Tercer paso ejemplo de exportar



Para el formulario Generación de Código, en el ejemplo de la Figura 27, solo generar código. En el número 1, se deben elegir las tablas para las cuales deseamos generar código; en el número 2, elegimos el lenguaje; en el número 3, elegimos el tipo de código; en el número 4, elegimos el código que deseamos exportar; en el número 5, se debe dar clic en el botón “Generar Código”; en el número 6, podemos ver el código generado; en el número 7, podemos ver el tiempo en horas, minutos, segundo y milisegundo que le tomó a GXC generar el código; y, en el número 8, podemos ver el nombre de la base de dato con la que se está trabajando.

Figura 27: Tercer paso ejemplo solo generar código

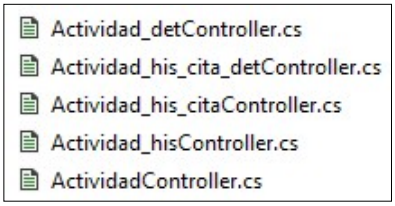


5.1.6 Creación de archivo .CS

La creación consiste en generar todas las opciones necesarias para tener los *View model* y el *controller*. Este proceso es necesario para que los archivos .cs estén listos para anexar a un proyecto que está ya creado en ASP.NET y utiliza *Entity Framework*. En la Figura 23, podemos ver que, para poder generar el código, se requiere un parámetro que es exportar. Asimismo, para el caso de la creación del archivo CS, tiene que ser *true*. En las Figuras 28 y 29, podemos ver los archivos que fueron exportados.

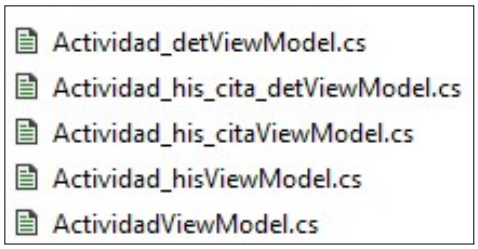
Para el caso de la Figura 28, podemos ver que los archivos .cs de los *controller* tienen como nombre predefinido “la tabla” y seguido de *Controller*.

Figura 28: Archivos *Controller* CS exportados



En la Figura 29, podemos ver los archivos que exporta la opción *ViewModel*. Estos tienen como nombre predefinido “la tabla” y seguido de *ViewModel*.

Figura 29: Archivos *View Model* CS exportados



Al exportar los *controller* (Figura 28) y *view model* (Figura 29), se llevan a cabo todos los procesos CRUD para *Entity Framework*. En la Figura 18, podemos ver cómo podemos seleccionar el código que se va a exportar; luego, se da clic en el botón “Exportar”. Por ejemplo, se llama al método Generar_*ViewModel*_*List*_Una_Tabla y, luego, esta llama al método *ViewModelList* para que devuelva el *script* al *ViewModel List*. De igual manera, se realiza el proceso para generar para el resto de métodos de *ViewModel* y *Controller*.

5.1.7 Estimación de costos

Sobre esto, estimamos los costos durante la implementación del proyecto. Estos se presentan en las tablas 17, 18 y 19.

Del mismo modo, los gastos realizados para los costos de servicio como Internet y fluido eléctrico, así como los gastos de útiles de escritorio, los podemos ver en la Tabla 17. En total, se realizó un gasto de S/. 374.00.

Tabla 17. Costo de servicio y útiles de escritorio

Concepto	Costo por mes	Cantidad	Unidad de medida	Costo total
Internet	S/ 85.00	2	Mes	S/ 170.00
Luz	S/ 70.00	2	Mes	S/ 140.00
Impresiones y fotocopias	S/ 30.00	1	Gasto único	S/ 30.00
Papel Bond A4	S/ 14.00	1	Paquete	S/ 14.00
Lapiceros	S/ 6.00	3	Unidad	S/ 18.00
Lápiz	S/ 1.00	1	Unidad	S/ 1.00
Borrador	S/ 1.00	1	Unidad	S/ 1.00
Total				**S/ 374.00**

En la Tabla 18, se muestran los costos por los servicios del personal necesario para llevar a cabo el desarrollo de la herramienta.

Tabla 18. Costos de servicio por personal

Perfil	Cantidad	Descripción de funciones	Pago mensual	Tiempo	Pago total
Técnico analista de sistemas	1	Programación y documentación	S/ 1,500.00	2 meses	S/ 3,000.00
Desarrollador	1	Encargado del proyecto	S/ 3,000.00	2 meses	S/ 6,000.00
Técnico analista de sistemas	1	Programación y documentación	S/ 1,500.00	2 meses	S/ 3,000.00
Total					**S/ 12,000.00**

El detalle de costos del *software* que se utilizó para el desarrollo lo podemos ver en la Tabla 19. Así mismo, debemos indicar que, como la herramienta se realizó con fines educativos, se utilizó en todos los casos la versión libre *Community* y *Express*. Por ello, no se realizó ningún gasto en licencias de *software*.

Tabla 19. Costos de *software* utilizado para el desarrollo

Software	Descripción	Versión	Costo de licencia
Visual Studio Code	Editor de código fuente	1.59.1	S/ 0.00
Visual Studio 2019	IDE para el desarrollo de *software*	*Community* 2019	S/ 0.00
SQL *Server*	Sistema gestor de base de datos de Microsoft	SQL *Express* 2014	S/ 0.00
Total			S/ 0.00

5.2 Pruebas y resultados

5.2.1 Pruebas

Para realizar las pruebas y obtener los resultados. Utilizamos la base de datos del Sistema Nacional de Salud HISMINSA, cuyo nombre es BDHIS_WEB, la cual tiene en total 151 tablas. Inicialmente se pensó en un experimento censal con las tablas de la base de datos. El experimento para obtener datos de la programación manual fue muy tedioso, pero ya se habían avanzado 100 tablas. Decidimos utilizar la siguiente fórmula para determinar el tamaño de la muestra:

$$n_0 = \frac{z^2 pq}{e^2} \qquad n = \frac{n_0}{1 + \frac{n_0}{N}}$$

Donde:

$n_0 = Muestra$ **sin ajustar**

$z = Limite\ de\ confianza$

$p = Proporción\ de\ aciertos$

$p = Proporción\ de\ desaciertos$

$e = Error\ máximo\ permisible$

$n = Muestra\ ajustada$

Las tablas fueron elegidas de forma aleatoria, para tener una mejor exactitud se trabajó con 100 tablas. Asimismo, debemos indicar que realizamos los CRUD de forma manual para, luego, comparar estos resultados con el código CRUD generado automáticamente usando la herramienta GXC.

El HISMINSA tiene diferentes módulos entre ellos están el módulo de punto de digitación, salud escolar, oficina de seguro, carga y descarga de datos, actividades complementarias, admisión y cubo por estrategias. Las 100 tablas elegidas de forma aleatoria pertenecen a los diferentes módulos.

De esta manera, se empleó un total de 12 horas, 29 minutos y 52 segundos para programar los CRUD de forma manual de las 100 tablas. Participaron todos los integrantes del equipo de desarrollo en este proceso. En ese sentido, los detalles los podemos ver en la Tabla 20.

Tabla 20. Datos generales de programación de CRUD manual

Responsable	Cantidad de tablas (Unidad)	Cantidad de columnas (Unidad)	Tiempo en minutos HH:MM:SS.MS
EARZ	21	129	03:43:16.00
JMRT	58	287	04:32:13.00
RADV	21	98	04:14:23.00
Totales	**100**	**514**	**12:29:52.00**

Al momento de realizar la programación, se utilizaron plantillas para agilizar el proceso. También, se obtuvieron los datos de las tablas en forma digital (Excel). Una vez finalizado el proceso de codificación, anexamos los *ViewModel* y los *Controllers* de todas las tablas a un proyecto. Al momento de compilar el proyecto, se tuvo un total de 1017 errores en todas las tablas.

En cambio, la herramienta GXC solo se demoró 18 segundos y 51 milésimas de segundo en generar el código para las 100 tablas. En la Tabla 21, podemos ver los detalles; también, hemos de indicar que, al momento de anexar a un proyecto, no se produjo ningún error.

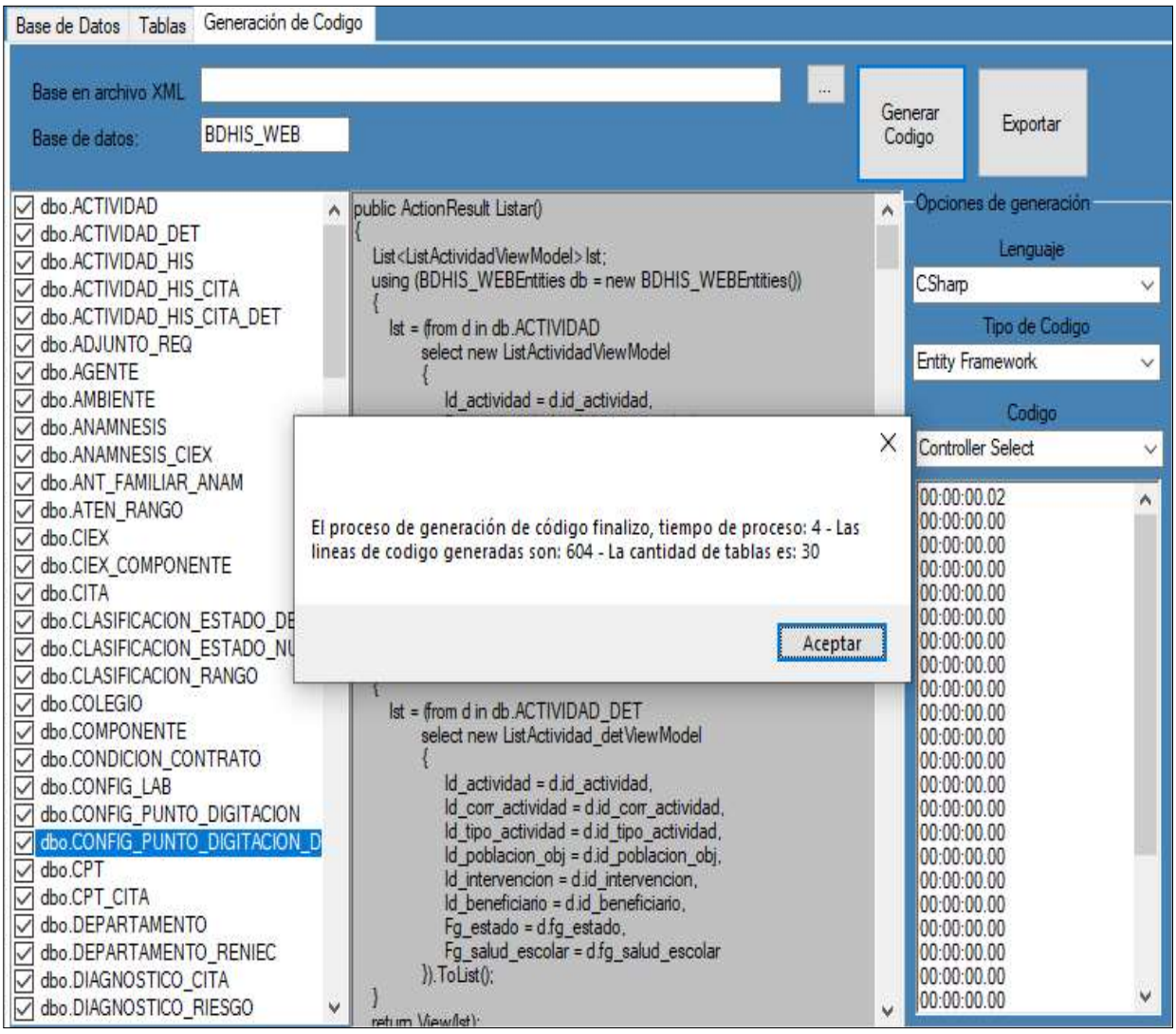
Tabla 21. Datos generales de generación de código fuente CRUD

Responsable	Cantidad de tablas (Unidad)	Cantidad de columnas (Unidad)	Tiempo en minutos HH:MM:SS.MS
GXC	100	514	00:00:18.51
Total	100	514	00:00:18.51

La herramienta GXC tiene incluido en el código un *Timer* que se encarga de medir el tiempo que demora la herramienta al generar código fuente y, también, al exportar. A su vez, se encarga de contar las líneas de código generadas y muestra la cantidad de tablas que se seleccionó para realizar la generación.

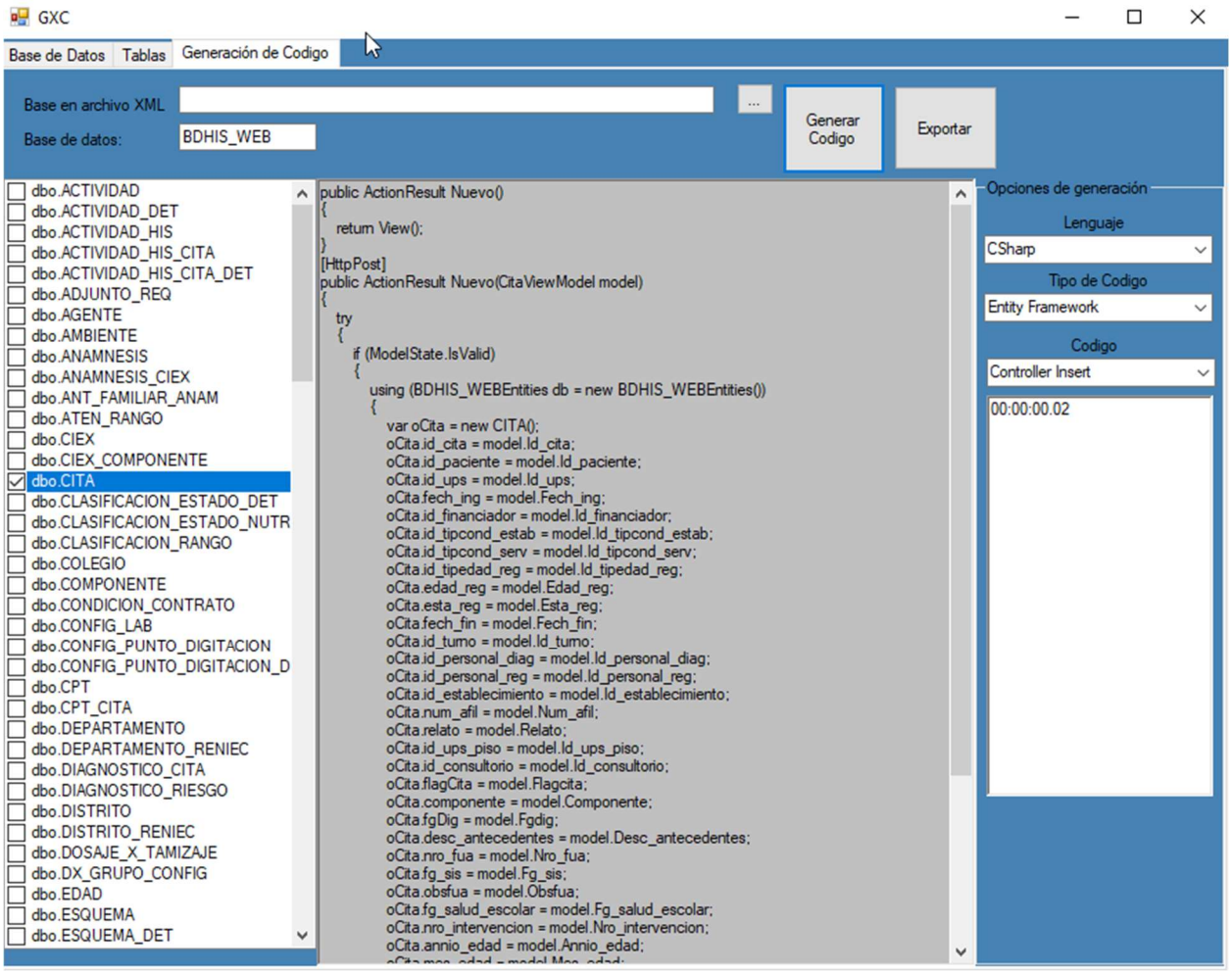
En la Figura 30, podemos ver un ejemplo de generación de código CRUD para 30 tablas que, en el *TextBoxt* de *script*, nos muestra el código generado; y un *RichTextBoxt* donde podemos ver los tiempos por cada tabla en horas, minutos, segundos y milisegundos. Por último, vemos un *MensageBox*, que nos da la información del tiempo, líneas de código y cantidad de tablas generadas.

Figura 30: Ejemplo de generación de código CRUD de 30 tablas



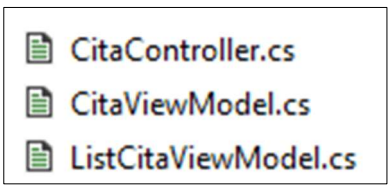
También, mostraremos un ejemplo de generación de código para la tabla CITA, de la base de datos BDHIS_WEB. En la Figura 31, podemos ver el código del *Controller Insert* generado para la tabla CITA, que nos servirá para registrar las citas de los pacientes.

Figura 31: Código generado para la tabla CITA



Para el ejemplo de la tabla CITA, se exportó el *View Model*, *View Model List* y el *Controller*. Esto lo podemos ver en la Figura 32.

Figura 32: Archivos exportados para la tabla CITA



5.2.1.1 Fiabilidad

Para realizar las pruebas de fiabilidad, se contabilizó la cantidad de errores que se cometieron al realizar la programación manual para las 100 tablas y los errores obtenidos con la herramienta GXC para las mismas tablas. Estos errores se muestran en la Tabla 22.

Tabla 22. Datos de errores de programación

Tabla	Errores con programación manual (Unidad)	Errores con el Generador GXC (Unidad)	Tabla (Unidad)	Errores con programación manual (Unidad)	Errores con el Generador GXC (Unidad)
1	8	0	51	8	0
2	8	0	52	8	0
3	8	0	53	8	0
4	18	0	54	10	0
5	8	0	55	12	0
6	10	0	56	10	0
7	8	0	57	8	0
8	8	0	58	8	0
9	8	0	59	8	0
10	14	0	60	8	0
11	28	0	61	8	0
12	24	0	62	8	0
13	26	0	63	8	0
14	8	0	64	8	0
15	12	0	65	18	0
16	8	0	66	8	0
17	8	0	67	8	0
18	8	0	68	8	0
19	8	0	69	8	0
20	8	0	70	14	0
21	10	0	71	14	0
22	8	0	72	70	0
23	9	0	73	8	0
24	8	0	74	8	0
25	15	0	75	8	0
26	10	0	76	8	0
27	16	0	77	16	0
28	8	0	78	8	0
29	10	0	79	13	0
30	20	0	80	8	0
31	8	0	81	8	0
32	8	0	82	8	0
33	8	0	83	12	0
34	8	0	84	14	0
35	8	0	85	12	0
36	8	0	86	8	0
37	8	0	87	8	0
38	8	0	88	8	0
39	8	0	89	10	0
40	8	0	90	8	0
41	8	0	91	8	0
42	8	0	92	8	0
43	8	0	93	8	0
44	16	0	94	8	0
45	10	0	95	8	0
46	8	0	96	10	0
47	8	0	97	8	0
48	8	0	98	8	0
49	8	0	99	8	0
50	8	0	100	8	0

Para extraer los datos de complejidad ciclomática, usamos tanto la solución de programación manual como la solución con la herramienta GXC. En la Figura 33, podemos ver cómo pudimos obtener las métricas. Para este objetivo, se siguieron los siguientes pasos:

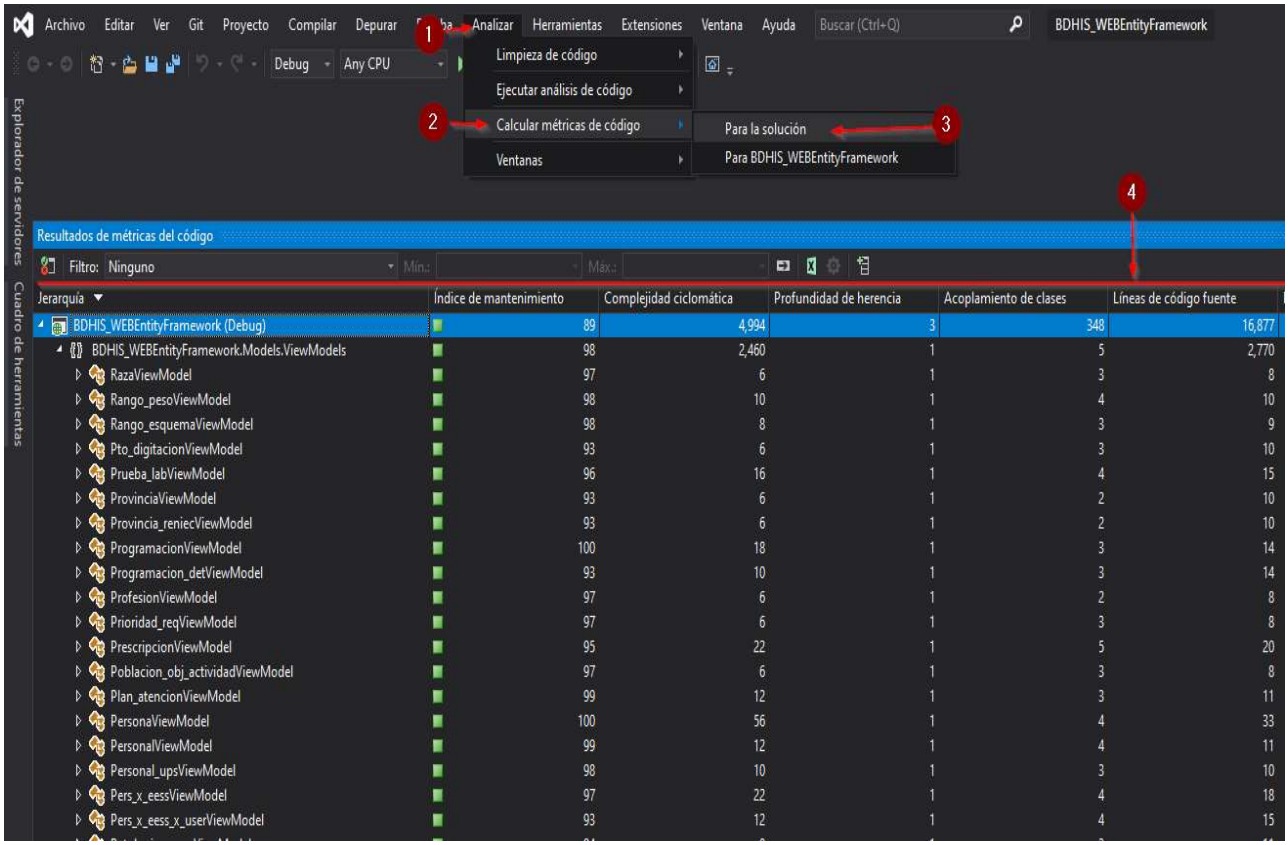
Paso 1: se da clic en la barra “Analizar”.

Paso 2: damos clic en “Calcular métricas de código”.

Paso 3: se da clic en “Para la solución”.

Paso 4: podemos ver las métricas que generó *Visual Studio*.

Figura 33: Calcular métricas de código con *Visual Studio* 2019



Se registraron los resultados de complejidad ciclomática de las mismas 100 tablas obtenidas al compilar el código en el IDE *Visual Studio* 2019. Estos resultados se muestran en la Tabla 23.

Tabla 23. Datos de complejidad ciclomática

Tabla	Complejidad ciclomática manual (Unidad)	Complejidad ciclomática GXC (Unidad)	Tabla	Complejidad ciclomática manual (Unidad)	Complejidad ciclomática GXC (Unidad)
1	20	20	51	16	16
2	32	32	52	20	20
3	20	20	53	16	16
4	40	40	54	16	16
5	28	28	55	36	36
6	24	24	56	24	24
7	16	16	57	19	19
8	16	16	58	20	20
9	152	152	59	20	20
10	16	16	60	16	16
11	22	22	61	16	16
12	24	24	62	16	16
13	68	68	63	28	28
14	16	16	64	28	28
15	120	120	65	44	44
16	40	40	66	12	12
17	40	40	67	20	20
18	40	40	68	16	16
19	20	20	69	12	12
20	20	20	70	64	64
21	20	20	71	80	80
22	12	12	72	68	68
23	36	36	73	24	24
24	20	20	74	16	16
25	36	36	75	16	16
26	24	24	76	20	20
27	16	16	77	24	24
28	16	16	78	20	20
29	32	32	79	28	28
30	20	20	80	20	20
31	16	16	81	20	20
32	24	24	82	44	44
33	12	12	83	20	20
34	20	20	84	100	100
35	12	12	85	24	24
36	16	16	86	16	16
37	80	80	87	28	28
38	28	28	88	20	20
39	20	20	89	44	44
40	16	16	90	20	20
41	28	28	91	20	20
42	16	16	92	40	40
43	16	16	93	20	20
44	20	20	94	16	16
45	28	28	95	20	20
46	28	28	96	32	32
47	72	72	97	20	20
48	32	32	98	24	24
49	20	20	99	28	28
50	24	24	100	20	20

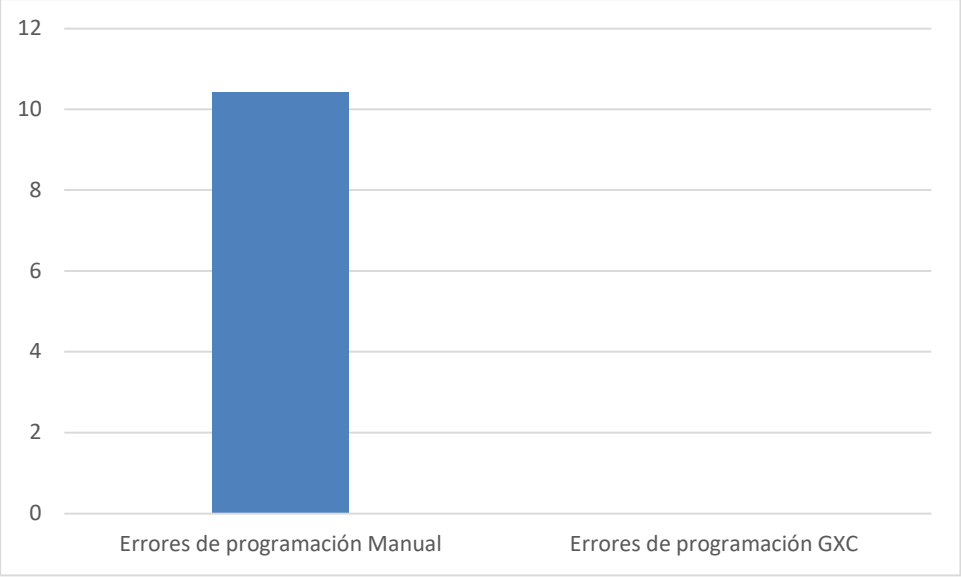
En la Tabla 24, se aprecian los valores de la media tanto de la programación manual como de los valores al utilizar la herramienta GXC. Asimismo, se aprecia que la media de los errores de programación manual son 10.43, un valor mínimo de errores de 8, un valor máximo de errores de 70, una desviación estándar de 7.169 y una varianza de 51.399; mientras, con la herramienta GXC, no se obtiene ningún error. Por lo tanto, todos los datos son 0. *A priori*, se observa que la herramienta GXC genera menos errores de programación.

Tabla 24. Estadísticos descriptivos de errores de programación

Dato evaluado	N	Mínimo	Máximo	Media	Desviación estándar	Varianza
Errores de programación manual	100	8	70	10.43	7.169	51.399
Errores de programación GXC	100	0	0	0.00	0.000	0.000
N válido (por lista)	100					

Luego, en la Figura 34, podemos ver la diferencia en la media de errores de la programación manual y la programación con la herramienta GXC.

Figura 34: Media de errores de programación manual y GXC



En la Tabla 25, se aprecia que la media de la complejidad ciclomática manual es 28.85, un valor mínimo de complejidad ciclomática de 8, un valor máximo de complejidad ciclomática de 152, una desviación estándar de 22.245 y una varianza de 494.836. Mientras, con la herramienta GXC, se

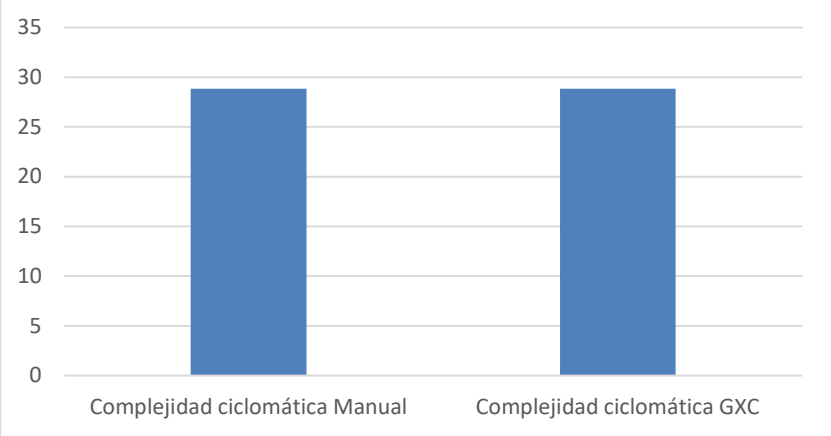
obtienen los mismos valores. Esto se debe a que la forma de generación de código fue escrita por los mismos programadores. *A priori*, se observa que la forma manual de programación y la programación con la herramienta GXC generan una complejidad ciclomática igual.

Tabla 25. Estadísticos descriptivos de complejidad ciclomática

Dato evaluado	N	Mínimo	Máximo	Media	Desviación estándar	Varianza
Complejidad ciclomática manual	100	12	152	28.85	22.245	494.836
Complejidad ciclomática GXC	100	12	152	28.85	22.245	494.836
N válido (por lista)	100					

Después, en la Figura 35, podemos ver que no existe diferencia entre la media de complejidad ciclomática de la programación manual y la programación con la herramienta GXC.

Figura 35: Media de complejidad ciclomática de programación manual y GXC



5.2.1.2 Eficiencia:

Para realizar las pruebas de eficiencia, se trabajó con el tiempo de programación manual y con el tiempo de programación con GXC (tiempo en mm,ss). Sobre esto anterior, los datos de ambos tiempos de programación los podemos ver en la Tabla 26.

Tabla 26. Datos de tiempo de programación

Tabla	Tiempo de programación manual (Minutos)	Tiempo de programación GXC (Minutos)	Tabla	Tiempo de programación manual (Minutos)	Tiempo de programación GXC (Minutos)
1	15,533333	0,000367	51	6,750000	0,000267
2	18,300000	0,000283	52	2,983333	0,000317
3	17,650000	0,000300	53	10,633333	0,000333
4	17,966667	0,000333	54	3,100000	0,000317
5	21,383333	0,000283	55	7,133333	0,000350
6	6,000000	0,000317	56	3,416667	0,000317
7	17,866667	0,000333	57	3,300000	0,000300
8	4,000000	0,000283	58	6,483333	0,000283
9	25,583333	0,000400	59	9,316667	0,000283
10	16,016667	0,000300	60	3,850000	0,000317
11	9,716667	0,000300	61	6,533333	0,000333
12	15,633333	0,000267	62	2,733333	0,000300
13	26,633333	0,000350	63	10,683333	0,000283
14	4,233333	0,000283	64	3,533333	0,000267
15	13,650000	0,000350	65	8,800000	0,000300
16	21,100000	0,000283	66	3,650000	0,000317
17	8,283333	0,000267	67	3,533333	0,000350
18	15,283333	0,000333	68	7,800000	0,000350
19	8,700000	0,000317	69	2,933333	0,000333
20	4,566667	0,000317	70	15,000000	0,000300
21	6,066667	0,000350	71	7,633333	0,000350
22	5,116667	0,000317	72	13,766667	0,000350
23	14,933333	0,000317	73	6,483333	0,000333
24	6,583333	0,000317	74	3,766667	0,000317
25	20,233333	0,000317	75	3,816667	0,000283
26	8,500000	0,000250	76	6,816667	0,000317
27	10,483333	0,000300	77	6,016667	0,000300
28	4,950000	0,000300	78	3,050000	0,000283
29	4,416667	0,000300	79	6,733333	0,000300
30	4,816667	0,000333	80	3,250000	0,000300
31	3,550000	0,000283	81	6,200000	0,000333
32	3,966667	0,000300	82	4,250000	0,000300
33	2,450000	0,000283	83	5,866667	0,000300
34	4,350000	0,000350	84	10,233333	0,000333
35	1,983333	0,000267	85	3,700000	0,000317
36	3,133333	0,000317	86	2,616667	0,000300
37	8,250000	0,000367	87	3,583333	0,000300
38	15,383333	0,000250	88	2,616667	0,000300
39	8,966667	0,000300	89	3,966667	0,000317
40	6,283333	0,000250	90	3,266667	0,000283

Tabla	Tiempo de programación manual (Minutos)	Tiempo de programación GXC (Minutos)	Tabla	Tiempo de programación manual (Minutos)	Tiempo de programación GXC (Minutos)
41	8,683333	0,000300	91	2,883333	0,000300
42	9,083333	0,000283	92	3,166667	0,000300
43	4,016667	0,000300	93	2,533333	0,000300
44	3,050000	0,000300	94	2,500000	0,000300
45	7,583333	0,000333	95	3,116667	0,000300
46	3,483333	0,000317	96	2,916667	0,000317
47	10,650000	0,000317	97	2,766667	0,000317
48	4,083333	0,000317	98	2,216667	0,000317
49	5,183333	0,000317	99	2,733333	0,000317
50	3,500000	0,000283	100	3,400000	0,000250

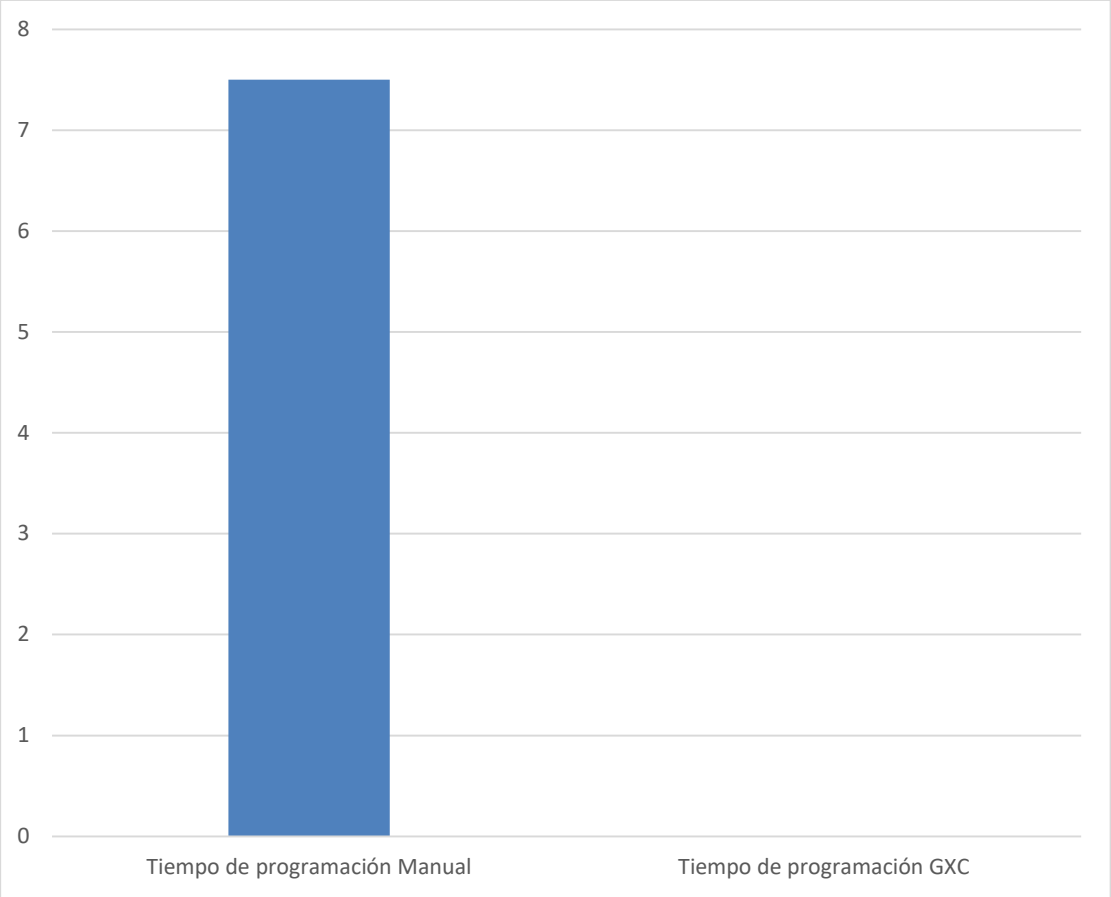
En la Tabla 27, se aprecia que la media del tiempo de programación manual es 7.49866663 minutos, un valor mínimo de tiempo manual de 1.983333 minutos, un valor máximo de tiempo manual de 26.633333 minutos, una desviación estándar de 5,598953515 y una varianza de 31,348. Mientras, con la herramienta GXC, se aprecia que la media del tiempo de programación es 0.00030852 minutos, un valor mínimo de tiempo de programación de 0.000250 minutos, un valor máximo de tiempo de programación de 0.000400 minutos, una desviación estándar de 0.000027192 y una varianza de 0.000. *A priori*, se observa que el tiempo de la programación con la herramienta GXC es mejor que la programación manual.

Tabla 27. Estadísticos descriptivos de tiempo de programación

Dato evaluado	N	Mínimo	Máximo	Media	Desviación estándar	Varianza
Tiempo de programación manual	100	1.983333	26.633333	7.49866663	5.598953515	31.348
Tiempo de programación GXC	100	0.000250	0.000400	0.00030852	0.000027192	0.000
N válido (por lista)	100					

En la Figura 36, podemos ver la diferencia entre la media del tiempo de la programación manual y la programación con la herramienta GXC.

Figura 36: Media de tiempo de programación manual y GXC



5.2.1.3 Mantenibilidad

Para realizar las pruebas de mantenibilidad, se trabajó con los resultados obtenidos al compilar el código con el IDE *Visual Studio* 2019, con respecto a la métrica de índice de mantenimiento. Los datos detallados los vemos en la Tabla 28.

Tabla 28. Datos de índice de mantenimiento

Tabla	Índice de mantenimiento manual (Unidad)	Índice de mantenimiento GXC (Unidad)	Tabla	Índice de mantenimiento manual (Unidad)	Índice de mantenimiento GXC (Unidad)
1	262	262	51	263	263
2	258	258	52	262	262
3	262	262	53	263	263
4	263	263	54	263	263
5	255	255	55	260	260
6	263	263	56	257	257
7	263	263	57	290	290
8	263	263	58	258	258
9	248	248	59	262	262
10	254	254	60	261	261
11	258	258	61	263	263
12	263	263	62	263	263
13	256	256	63	262	262
14	267	267	64	258	258
15	251	251	65	260	260
16	261	261	66	260	260
17	257	257	67	264	264
18	257	257	68	265	265
19	262	262	69	267	267
20	265	265	70	261	261
21	262	262	71	250	250
22	267	267	72	252	252
23	260	260	73	254	254
24	258	258	74	261	261
25	260	260	75	263	263
26	263	263	76	264	264
27	263	263	77	263	263
28	263	263	78	262	262
29	262	262	79	263	263
30	262	262	80	260	260
31	263	263	81	258	258
32	257	257	82	265	265
33	267	267	83	260	260
34	262	262	84	265	265
35	267	267	85	252	252
36	267	267	86	265	265
37	255	255	87	265	265
38	263	263	88	260	260
39	262	262	89	260	260
40	263	263	90	257	257
41	261	261	91	262	262
42	263	263	92	265	265
43	263	263	93	254	254
44	262	262	94	261	261
45	261	261	95	260	260
46	263	263	96	264	264
47	256	256	97	256	256
48	262	262	98	263	263
49	262	262	99	263	263
50	263	263	100	260	260

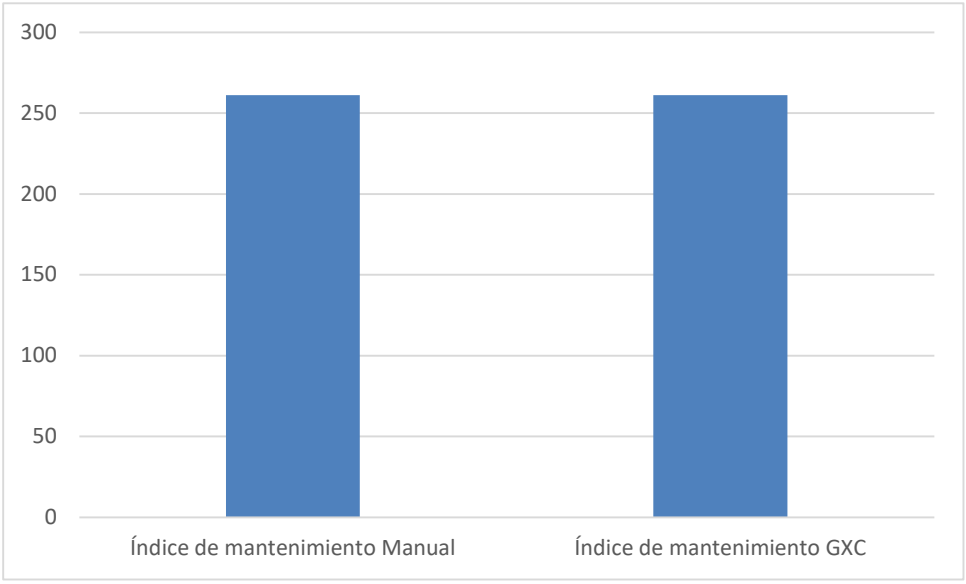
En la Tabla 29, se aprecia que la media del índice de mantenimiento de programación manual es 261.19, un valor mínimo del índice de mantenimiento de 248, un valor máximo del índice de mantenimiento de programación manual de 290, una desviación estándar de 4.819 y una varianza de 23.226. Mientras, con la herramienta GXC, se obtienen los mismos valores. Esto se debe a que la forma de generación de código fue escrita por los mismos programadores. *A priori*, se observa que la forma manual de programación y la programación con la herramienta GXC generan índice de mantenimiento igual.

Tabla 29. Estadísticos descriptivos de índice de mantenimiento

Dato evaluado	N	Mínimo	Máximo	Media	Desviación estándar	Varianza
Índice de mantenimiento Manual	100	248	290	261.19	4.819	23.226
Índice de mantenimiento GXC	100	248	290	261.19	4.819	23.226
N válido (por lista)	100					

En la Figura 37, podemos ver que no existe diferencia entre la media del índice de mantenimiento de la programación manual y la programación con la herramienta GXC.

Figura 37: Media de índice de mantenimiento de programación manual y GXC



5.2.2 Resultados

Para obtener los resultados de todas las pruebas que realizamos, trabajamos con el *software* IBM SPSS V24.

Del mismo modo, para obtener los resultados de la herramienta GXC, se realizó una comparación entre la programación manual y la programación con la herramienta GXC. De acuerdo a los objetivos, se aplicaron los instrumentos listados en la Tabla 30.

Tabla 30. Tabla de objetivos e instrumentos

Objetivo	Tipo de instrumento
Mejorar la fiabilidad del desarrollo de sistemas con el lenguaje C#. Caso: procesos CRUD del sistema HISMINSA	Registros manuales vs. Registros de la herramienta GXC
Incrementar la eficiencia del desarrollo de sistemas con el lenguaje C#. Caso: procesos CRUD del sistema HISMINSA	Registros manuales vs. Registros de la herramienta GXC
Mejorar la mantenibilidad del desarrollo de sistemas con el lenguaje C#. Caso: procesos CRUD del sistema HISMINSA	Registros manuales vs. Registros de la herramienta GXC

5.2.2.1 Resultados de mejorar la fiabilidad del desarrollo de sistemas con el lenguaje C#

Para la medición de la fiabilidad, se creó una solución nueva a la cual se le agregó el modelo de datos con la conexión a la base de datos del sistema HISMINSA; anexamos el código que se escribió de forma manual de las 100 tablas. Posteriormente, se compiló la solución y se contaron de forma manual los errores. Para recolectar los errores de la programación con la herramienta GXC, se creó una solución nueva a la cual se agregó el modelo de datos con la conexión a la base de datos del sistema HISMINSA; anexamos el código generado de las 100 tablas; posteriormente, se compiló la solución y los errores se contaron de forma manual.

Para recuperar la información de la complejidad ciclomática –lo hicimos para la programación manual y para la programación con la herramienta GXC– se utilizó la solución que creamos, y, con la facilidad que brinda *Visual Studio*, elegimos en la barra de menú “analizar”, luego “calcular métricas de código” y después elegimos la opción “para la solución”.

Por otro lado, realizamos la prueba de normalidad para la variable errores de programación manual. En el análisis, se encuentra que P es igual a 0.000. Además, se puede apreciar que es menor a 0.05. Esto se puede observar en la Tabla 31.

Tabla 31. Prueba de Kolmogorov-Smirnov para la variable errores de programación manual

		Errores de programación manual
N		100
Parámetros normales[a,b]	Media	10,43
	Desviación estándar	7,169
Máximas diferencias extremas	Absoluta	,367
	Positivo	,333
	Negativo	-,367
Estadístico de prueba		,367
Sig. asintótica (bilateral)		,000[c]

Una prueba similar se realizó para la variable de la programación con la herramienta GXC, donde P es igual a 0.000 y, también, es menor a 0.05. En la Tabla 32, podemos apreciar los resultados.

Tabla 32. Prueba de Kolmogorov-Smirnov para la variable errores de programación GXC

		Errores de programación GXC
N		100
Parámetros normales[a,b]	Media	,00
	Desviación estándar	,000c

En vista de que los resultados de la variable errores de programación manual y errores de programación GXC dieron como resultados de que los datos no son normales y además relacionados, se aplicó la prueba de Wilcoxon, tal y como se aprecia en la Tabla 33. Como el valor de p es igual a 0.000, es decir, menor a 0.05, se comprueba que la programación con la herramienta GXC es muy efectiva para reducir los errores de programación para el caso de los procesos CRUD, que se realizó con 100 tablas.

Tabla 33. Prueba de Wilcoxon para errores de programación manual y GXC

Rangos				
		N	Rango promedio	Suma de rangos
Errores de programación GXC - Errores de programación manual	Rangos negativos	100[a]	50,50	5050,00
	Rangos positivos	0[b]	,00	,00
	Empates	0[c]		
	Total	100		
a. Errores de programación GXC < Errores de programación manual b. Errores de programación GXC > Errores de programación manual c. Errores de programación GXC = Errores de programación manual				

Estadísticos de prueba[a]	
	Errores de programación GXC - Errores de programación manual
Z	-9,074[b]
Sig. asintótica (bilateral)	,000
a. Prueba de rangos con signo de Wilcoxon b. Se basa en rangos positivos.	

La prueba de normalidad de la variable complejidad ciclomática manual indica que la distribución de esta variable no es normal en vista de que el valor de p es igual a 0.000 y es menor a 0.05. Estos resultados se muestran en la Tabla 34.

Tabla 34. Prueba de Kolmogorov-Smirnov para la variable complejidad ciclomática manual

		Complejidad ciclomática manual
N		100
Parámetros normales[a,b]	Media	28,85
	Desviación estándar	22,245
Máximas diferencias extremas	Absoluta	,275
	Positivo	,275
	Negativo	-,232
Estadístico de prueba		,275
Sig. asintótica (bilateral)		,000c

La prueba de la variable complejidad ciclomática GXC indica que la distribución no es normal; esto se debe a que el valor de p es igual a 0.000 y es menor a 0.05. Esto se muestra en la Tabla 35.

Tabla 35. Prueba de Kolmogorov-Smirnov para la variable complejidad ciclomática manual

		Complejidad ciclomática GXC
N		100
Parámetros normales[a,b]	Media	28,85
	Desviación estándar	22,245
Máximas diferencias extremas	Absoluta	,275
	Positivo	,275
	Negativo	-,232
Estadístico de prueba		,275
Sig. asintótica (bilateral)		,000c
a. La distribución de prueba es normal. b. Se calcula a partir de datos. c. Corrección de significación de Lilliefors		

En vista de que los datos de complejidad ciclomática de la programación manual y la programación con GXC no son normales y relacionadas, se realizó la prueba de Wilcoxon. Esta prueba dio como resultado que la complejidad ciclomática de la programación manual y la programación con GXC es igual. Esta prueba la podemos ver en la Tabla 36.

Tabla 36. Prueba de Wilcoxon para Complejidad Ciclomática de programación manual y GXC

Rangos				
		N	Rango promedio	Suma de rangos
Complejidad ciclomática GXC - Complejidad ciclomática manual	Rangos negativos	0[a]	,00	,00
	Rangos positivos	0[b]	,00	,00
	Empates	100[c]		
	Total	100		
a. Complejidad ciclomática GXC < Complejidad ciclomática manual b. Complejidad ciclomática GXC > Complejidad ciclomática manual c. Complejidad ciclomática GXC = Complejidad ciclomática manual				

Estadísticos de prueba[a]	
	Complejidad ciclomática GXC - Complejidad ciclomática manual
Z	,000[b]
Sig. asintótica (bilateral)	1,000
a. Prueba de rangos con signo de Wilcoxon b. La suma de rangos negativos es igual a la suma de rangos positivos.	

5.2.2.2 Resultados de incrementar la eficiencia del desarrollo de sistemas con el lenguaje C#

Para realizar las pruebas y obtener resultados con el objetivo de incrementar la eficiencia del desarrollo de sistemas con el lenguaje C#, se trabajó con el tiempo de programación manual y el tiempo de programación con la herramienta GXC.

Para recuperar los datos del tiempo de programación manual, se realizó de forma manual con las grabaciones de pantalla durante el proceso de programación. Además, para recuperar el tiempo de la programación con la herramienta GXC, se recuperó el tiempo que brinda la herramienta GXC al momento realizar el proceso de generación de código.

Asimismo, realizamos la prueba de normalidad para la variable tiempo de programación manual. En el análisis, se encuentra que P es igual a 0.000 y se puede apreciar que es menor a 0.05. Esto se puede observar en la Tabla 37.

Tabla 37. Prueba de Kolmogorov-Smirnov para la variable tiempo de programación manual

		Tiempo de programación manual
N		100
Parámetros normales[a,b]	Media	7,49866663
	Desviación estándar	5,598953515
Máximas diferencias extremas	Absoluta	,178
	Positivo	,178
	Negativo	-,164
Estadístico de prueba		,178
Sig. asintótica (bilateral)		,000[c]

a. La distribución de prueba es normal.
b. Se calcula a partir de datos.
c. Corrección de significación de Lilliefors.

La prueba de normalidad de la variable tiempo de programación GXC indica que la distribución de esta variable no es normal. Esto se debe a que el valor de p es igual a 0.000 y es menor a 0.05. Estos resultados se muestran en la Tabla 38.

Tabla 38. Prueba de Kolmogorov-Smirnov para la variable tiempo de programación GXC

		Tiempo de programación GXC
N		100
Parámetros normales[a,b]	Media	,00030852
	Desviación estándar	,000027192
Máximas diferencias extremas	Absoluta	,148
	Positivo	,148
	Negativo	-,137
Estadístico de prueba		,148
Sig. asintótica (bilateral)		,000[c]
a. La distribución de prueba es normal. b. Se calcula a partir de datos. c. Corrección de significación de Lilliefors		

En vista de que los datos de las variables del tiempo de la programación manual y tiempo de la programación con GXC no son normales, se realizó la prueba de Wilcoxon. Esta prueba dio como resultado que la programación con la herramienta GXC es más efectiva que la programación manual. Los resultados de esta prueba pueden ser vistos en la Tabla 39.

Tabla 39. Prueba de Wilcoxon para Tiempo de programación manual y GXC

Rangos				
		N	Rango promedio	Suma de rangos
Tiempo de programación GXC - Tiempo de programación manual	Rangos negativos	100[a]	50,50	5050,00
	Rangos positivos	0[b]	,00	,00
	Empates	0[c]		
	Total	100		
a. Tiempo de programación GXC < Tiempo de programación manual b. Tiempo de programación GXC > Tiempo de programación manual c. Tiempo de programación GXC = Tiempo de programación manual				

Estadísticos de prueba[a]	
	Tiempo de programación GXC - Tiempo de programación manual
Z	-8,682[b]
Sig. asintótica (bilateral)	,000
a. Prueba de rangos con signo de Wilcoxon b. Se basa en rangos positivos.	

5.2.2.3 Resultados de mantenibilidad del desarrollo de sistemas con el lenguaje C#

Para realizar las pruebas y obtener resultados con el objetivo de mejorar la mantenibilidad del desarrollo de sistemas con el lenguaje C#, se trabajó con el índice de mantenibilidad de programación manual y el índice de mantenibilidad de programación con la herramienta GXC.

Para recuperar la información del índice de mantenibilidad, comparamos la programación manual y la programación con la herramienta GXC. Para ello, se utilizó la solución que creamos y, con la facilidad que brinda *Visual Studio*, elegimos en la barra de menú “analizar”, luego “calcular métricas de código” y, después, elegimos la opción “para la solución”.

Por otro lado, realizamos la prueba de normalidad para la variable índice de mantenimiento de programación manual. En el análisis, se encuentra que P es igual a 0.000, además de que se puede apreciar que es menor a 0.05. Esto se puede observar en la Tabla 40.

Tabla 40. Prueba de Kolmogorov-Smirnov para el índice de mantenimiento de programación manual

		Índice de mantenimiento manual
N		100
Parámetros normales[a,b]	Media	261,19
	Desviación estándar	4,819
Máximas diferencias extremas	Absoluta	,184
	Positivo	,184
	Negativo	-,162
Estadístico de prueba		,184
Sig. asintótica (bilateral)		,000c
a. La distribución de prueba es normal. b. Se calcula a partir de datos. c. Corrección de significación de Lilliefors		

La prueba de normalidad de la variable índice de mantenimiento GXC indica que la distribución de esta variable no es normal en vista de que el valor de p es igual a 0.000 y es menor a 0.05. Estos resultados se muestran en la Tabla 41.

Tabla 41. Prueba de Kolmogorov-Smirnov para una muestra para el índice de mantenimiento de programación GXC

		Índice de mantenimiento GXC
N		100
Parámetros normales[a,b]	Media	261,19
	Desviación estándar	4,819
Máximas diferencias extremas	Absoluta	,184
	Positivo	,184
	Negativo	-,162
Estadístico de prueba		,184
Sig. asintótica (bilateral)		,000[c]
a. La distribución de prueba es normal. b. Se calcula a partir de datos. c. Corrección de significación de Lilliefors		

En vista de que los datos de las variables del índice de mantenimiento de la programación manual y el índice de mantenimiento de la programación con GXC no son normales, se realizó la prueba de Wilcoxon. Esta prueba dio como resultado que los **índices de mantenimiento son iguales** en ambas formas de programación. De esta manera, los resultados de esta prueba podemos verlos en la Tabla 42.

Tabla 42. Prueba de Wilcoxon para índice de mantenimiento de programación manual y GXC

Rangos				
		N	Rango promedio	Suma de rangos
Índice de mantenimiento GXC - Índice de mantenimiento manual	Rangos negativos	0[a]	,00	,00
	Rangos positivos	0[b]	,00	,00
	Empates	100[c]		
	Total	100		
a. Índice de mantenimiento GXC < Índice de mantenimiento manual b. Índice de mantenimiento GXC > Índice de mantenimiento manual c. Índice de mantenimiento GXC = Índice de mantenimiento manual				

Estadísticos de prueba[a]	
	Índice de mantenimiento GXC - Índice de mantenimiento manual
Z	,000[b]
Sig. asintótica (bilateral)	1,000
a. Prueba de rangos con signo de Wilcoxon b. La suma de rangos negativos es igual a la suma de rangos positivos.	

CONCLUSIONES

En primer lugar, se logró el objetivo de mejorar la fiabilidad del desarrollo de sistemas con el lenguaje C# en vista de que, en una programación manual, se tiene la media de 10,43 errores y, con la programación con la herramienta GXC, no se obtuvo ningún error. Ambas formas de programación tienen una media de complejidad ciclomática de 28,85, considerando que ambos tienen el *controller*, *view model* y un *view model list*.

Luego, se logró incrementar la eficiencia del desarrollo de sistemas, porque, con una programación manual, tenemos un tiempo medio de 7,498667. Mientras, con la programación con GXC, tenemos un tiempo medio 0,000309.

Del mismo modo, se logró tener una mantenibilidad adecuada. Con el código generado, la media es de 261,190000 y, con la programación manual, es el mismo valor. Por ello, es un valor adecuado, porque es la suma del índice de mantenimiento del *controller*, *view model* y un *view model list*.

Finalmente, con base en las conclusiones anteriores, podemos indicar que se ha cumplido con todos los objetivos que se plantearon para la presente tesis. Esto se debe a que se logra la mejora significativa de la eficiencia en la programación con la herramienta GXC y se tienen adecuados niveles de las métricas de complejidad ciclomática e índice de mantenibilidad, esto con respecto a los objetivos de fiabilidad, eficiencia y mantenibilidad.

TRABAJOS FUTUROS

Primero, se debería ver la posibilidad de agregar formas de generación de código para otros lenguajes como *Python*, rescatando la lógica para generar código de la presente tesis. De esta manera, se mejorará la fiabilidad del desarrollo de sistemas con el lenguaje *Python*.

Segundo, la eficiencia en el desarrollo de sistemas siempre será un tema que debe ser analizado y estudiado, por lo que se debe considerar lo indicado en el párrafo anterior para lograr mejoras significativas.

Por otro lado, lograr la mantenibilidad de un código debe ser importante para todos los equipos de desarrollo. Por ello, el tema debe ser estudiado. Sin embargo, la generación de código es una buena opción para lograr este objetivo.

Por último, se deberían analizar los temas de investigación relacionados a la generación de código para otros lenguajes de programación y que puedan ser integrados a la herramienta GXC. También, se debería ampliar la conexión con otros gestores de base de datos con la finalidad de recuperar información de las bases de datos y poder realizar la generación de código fuente. Esto sería importante para continuar aportando a la mejora del proceso de desarrollo de *software*.

REFERENCIAS BIBLIOGRÁFICAS

1. **MICROSOFT.** *Mejora de la fiabilidad para los componentes de actualización de Windows 10.* [En línea] [Citado el: 10 de 07 de 2021.] Disponible en: https://support.microsoft.com/es-es/topic/mejora-de-la-fiabilidad-para-los-componentes-de-actualizaci%C3%B3n-de-windows-10-ecef08d2-e544-372d-d2ec-e0d3792734f3.

2. *Relationship between Factors of Quality Models and the System Development Life Cycle.* **Habib, B. & Raza, R. A.** 10, 2013, Vol. 81.

3. *Analizando la Mantenibilidad de Software Desarrollado Durante la Formación Universitaria.* **Perez-Gonzales, H. & Martinez, F. E. & Nava, S. & Nuñez, V. & Vázquez, M. & Flores, J. A.** 231-236, s.l. : Revista Latinoamericana de Ingenieria de Software, 2015.

4. **IEEE.** *Systems and software engineering -- Vocabulary.* [En línea] 15 de 12 de 2010. [Citado el: 10 de 07 de 2021.] Disponible en: https://ieeexplore.ieee.org/document/5733835. ISBN electrónico:978-0-7381-6205-8.

5. **STANDISH GROUP.** *CHAOS REPORT 2015.* [En línea] 2015. [Citado el: 27 de 09 de 2021.] Disponible en: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf.

6. **Makihara, E., et al.** *A Multi-Year Analysis of Students' Build Errors in Agile Software Development Educational Projects.* [En línea] 2018. [Citado el: 24 de 09 de 2021.] Disponible en: https://doi.org/10.1145/3183440.3195064. ISBN: 9781450356633.

7. **Sánchez, J. A.** DotNetGenerator: Generador de Código para Arquitectura Microsoft .NET a partir de modelos ISML. Bógota : s.n., 2018.

8. *Recursive modeling for completed code generation.* **Sulistyo, S. y Prinz, A.** s.l. : ACM, 2009.

9. *Unit test code generator for lua programming language.* **Wibowo, J. T. P., Hendradjaya, B. y Widyani, Y.** Bandung : IEEE, 2015, IEEE, pág. 5.

10. *Application of Architecture Implementation Patterns by Incremental Code Generation.* **Brunnlieb, M y Poetzsch-Heffter, A.** Leerdam : ACM, 2016. DOI: http://dx.doi.org/10.1145/3022636.3022647.

11. *Evaluation of FED-CASE - A Tool to Convert.* **Sadaf, S., Athar, A. y Azam, F.** Islamabad : IEEE, 2016. DOI 10.1109/IS3C.2016.57.

12. *The Metric for Automatic Code Generation.* **Li, Z., y otros.** s.l. : ACM, 2020, Vol. III. doi:10.1016/j.procs.2020.02.099.

13. *Design and Implementation of B/S Architecture Code Automatic Generation System.* **Han, J., y otros.** Shijiazhuang : IEEE, 2019.

14. *Automatically propagating changes from reference implementations to code generation templates.* **Possatto, M. A. y Lucrédio, D.** Rod. Washington Luís : ELSEVIER, 2015.

15. *An Automatic Page Code Generation Method Based On Excel Template And Poi Technology.* **She, X. y Zheng, Y.** Changehun 130012 : IEEE, 2020. DOI 10.1109/ICITBS49701.2020.00123.

16. *Automatic Source Code Generation for Web-based Process-oriented Information Systems.* **Alfonso, J. y Restrepo, F.** Bogotá : ACM, 2017, Vol. XII. ISBN: 978-989-758-250-9.

17. *ISML-MDE: A Practical Experience of Implementing a Model Driven Environment in a Software Development Organization.* **Franky, M. C., y otros.** s.l. : Emerald Insight, 2016.

18. *JDriver: Automatic Driver Class Generation for AFL-Based Java Fuzzing Tools.* **Huang, Z. y Wang, Y.** China : Symmetry, 2018. doi:10.3390/sym10100460.

19. **Becerra, J. C.** Generador de código de funcionalidades tipo CRUD en la mantenibilidad de software aplicado a sistemas de información empresariales. Trujillo : s.n., 2019.

20. **Palma, C. W. & Velasquez, C. E.** Framework para aplicaciones con base de datos relacional orientado a desarrolladores de software. Lima : s.n., 2019.

21. **Sommerville, I.** *Ingeniería de software.* México : Person Educación, 2011. ISBN: 978-607-32-0603-7.

22. **Sistemas, División de.** Metodología de desarrollo de software. Chimbote : s.n., 2017. 1.

23. *CRUD - PG.* **Sotologo, A. & Gutierrez, M. & Piñeiro, B.** 1, La Habana : Revista Cubana de Ciencias Informáticas, 2011, Vol. V. Revista Cubana de Ciencias Informáticas.

24. **González, Rocío.** Crehana. [En línea] 4 de Febrero de 2021. [Citado el: 17 de 08 de 2021.] https://www.crehana.com/pe/blog/desarrollo-web/que-es-un-framework/.

25. **Gutierrez, D.** codecompiling. [En línea] Abril de 2010. [Citado el: 18 de Agosto de 2021.] http://www.codecompiling.net/files/slides/IS_clase_10_frameworks_componentes.pdf.

26. *Meta Patterns - A Means For Capturing the Essentials of Reusable Object-Oriented Design.* **Pree, W.** Linz : s.n., 1994, Vol. VIII.

27. **HOTFRAMEWORKS.** *Encuentra tu nuevo framework web favorito.* [En línea] [Citado el: 28 de 09 de 2021.] Disponible en: https://hotframeworks.com/.

28. **MICROSOFT.** [En línea] 2018. [Citado el: 24 de 09 de 2021.] https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview.

29. *UAICase: Integración de un Entorno Académico con una Herramienta CASE en una Plataforma Virtual Colaborativa.* **Battaglia, N. & Neil, C. & De Vincenzi, M. & Martinez, R.** 123-131, s.l. : TE&ET, 2016, Vol. XI. ISBN: 978-987-3977-30-5.

30. *Aplicación de herramientas CASE a la enseñanza de ingeniería de software: gestión de la configuración de software y testing funcional.* **Daniele, M. & Uva, M. & Martelloto P. & Picco G.** Río Cuarto : TE&ET, 2010, Vol. V.

31. **Gallegos, L. C.** *Análisis del impacto del uso de herramientas CASE en el desarrollo de software dentro de las pequeñas y medianas empresas de Querétaro.* Queretaro : s.n., 2011.

32. *An analysis of tools for automatic software development and automatic code generation.* **Rosales, M. V. Y. & Alor, H. G. & Garcia, A. J. L. & Zatarain, C. R. & Barrón, E. M. L.** 77, Juárez : Revista Facultad de Ingeniería, Universidad de Antioquia, 2015. DOI: 10.17533/udea.redin.n77a10.

33. **Vozmediano, A. M.** *Java para novatos: Cómo aprender programación orientada a objetos con Java sin desesperarse en el intento.* s.l. : Safe creative, 2017.

34. **Marqués, M.** *Bases de datos.* s.l. : Universitat Jaume I, 2010. ISBN: 978-84-693-0146-3.

35. **Aramburu, M. & Sanz, I.** *Bases de datos avanzadas.* s.l. : Universitat Jaume I, 2013. ISBN: 978-84-695-6769-2.

36. **Reis de Micaelo Simões, G.** Automação da análise de qualidade de código. 2014.

37. **Fitzpatrick, R.** *Software quality: definitions and strategic issues.* [En línea] 1996. [Citado el: 06 de 08 de 2021.] Disponible en: https://arrow.tudublin.ie/cgi/viewcontent.cgi?article=1000&context=scschcomrep.

38. **MICROSOFT.** *Cómo: Generar datos de métricas de código.* [En línea] 02 de 11 de 2018. [Citado el: 25 de 09 de 2021.] Disponible en: https://docs.microsoft.com/es-es/visualstudio/code-quality/how-to-generate-code-metrics-data?view=vs-2019#net-code-quality-analyzers-code-metrics-rules.

39. **Pilalonga, A.** *Modelo de Calidad Mc Call.* [En línea] 27 de 09 de 2017. [Citado el: 19 de 08 de 2021.] Disponible en: https://sites.google.com/site/moduloevaluacionred/modelo-mc-call.

40. **Echeverría, P. D. y Abella, P. A.** *Testing como Práctica para Evaluar la Eficiencia en Aplicaciones Web.* [En línea] 2014. [Citado el: 24 de 09 de 2021.] Disponible en: https://core.ac.uk/download/pdf/234157038.pdf.

41. **Moreno, G. M. N.** Departamento de Informática y Automática. *Medoción del software.* [En línea] [Citado el: 24 de 09 de 2021.] Disponible en: http://avellano.usal.es/~mmoreno/APITema2.pdf.

42. **Gonzáles, D. H.** *Las métricas de software y su uso en la region.* [En línea] 7 de 05 de 2001. [Citado el: 16 de 08 de 2021.] Disponible en: http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/gonzalez_d_h/.

43. **MICROSOFT.** *Valores de métricas de código.* [En línea] 02 de 11 de 2018. [Citado el: 19 de 09 de 2021.] Disponible en: https://docs.microsoft.com/es-es/visualstudio/code-quality/code-metrics-values?view=vs-2019.

44. **WIKIPEDIA.** *Complejidad ciclomática.* [En línea] 2020. [Citado el: 28 de 09 de 2021.] Disponible en: https://es.wikipedia.org/wiki/Complejidad_ciclom%C3%A1tica#/media/Archivo:Control_flow_graph_of_function_with_loop_and_an_if_statement_without_loop_back.svg.

45. **MICROSOFT.** *Métricas de código: rango y significado del índice de mantenimiento.* [En línea] 08 de 01 de 2021. [Citado el: 19 de 09 de 2021.] Disponible en:

https://docs.microsoft.com/es-es/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2019.

46. **Ambler, S. W. & Lines, M.** *Disciplined agile delivery: A practitioner's guide to agile software delivery in the enterprise.* 2012.

47. **Kendall, Julie E. y Kendall, Kenneth E.** *Análisis y Diseño de Sistemas.* s.l. : PEARSON EDUCATION, 2011.

48. **Pipinellis, Achilleas.** *GitHub Essentials.* s.l. : Packt Publishing, 2015.

49. **Palacio, Juan.** *SCRUM Manager: Gestión de Proyectos.* s.l. : Safe Creative, 2008.

50. **Menzinsky, Alexander, López, Gertrudis y Palacio, Juan .** *Scrum Manager: Guía de Formación.* s.l. : Lubaris Info 4 Media SL, 2016.

51. *Software Quality: Concepts and Practice, CASE Tools and IDEs - Impact on Software Quality.* **D., Galin.** 2018.