Universidad Nacional
Federico Villarreal

Vicerrectorado de
INVESTIGACIÓN

ESCUELA UNIVERSITARIA DE POSTGRADO

"METODOLOGÍA ÁGIL ICONIX EN LA CALIDAD DEL PRODUCTO SOFTWARE, LIMA, 2017"

TESIS PARA OPTAR EL GRADO ACADÉMICO DE:
DOCTOR EN INGENIERÍA DE SISTEMAS

AUTOR:
PORRAS FLORES, EFRAÍN ELÍAS

ASESOR:
DR. MAYHUASCA GUERRA, JORGE VICTOR

JURADO:
DR. GOMEZ LORA, JHON WALTER
DR. RODRIGUEZ RODRIGUEZ, CIRO
DR. SOTO SOTO, LUIS

LIMA – PERÚ

2019

TÍTULO DE LA TESIS

METODOLOGÍA ÁGIL ICONIX EN LA CALIDAD DEL PRODUCTO SOFTWARE, LIMA, 2017.

AUTOR

PORRAS FLORES, EFRAÍN ELÍAS.

DEDICATORIA

A Dios por darme la vida e inteligencia

A Mis Padres Salvador y Victoria por su lucha diaria y sus ejemplos

A Helena que la recuerdo eternamente

A mi Familia

A Vicky por su paciencia y comprensión

AGRADECIMIENTO

A la Universidad Nacional Federico Villareal y Docentes por Mi formación Doctoral

A Mi alma Mater la Universidad Nacional de Ingeniería

A Mis Maestros de todos los tiempos

A todas las personas que hicieron posible la culminación de este estudio

ÍNDICE

INDICE DE TABLAS

INDICE DE FIGURAS

RESUMEN

El Perú es una nación inmersa en la globalización económica y social, para ser un país competitivo necesita que la industria nacional de software desarrolle sistemas de información con alto grado de calidad del producto software, según el INEI en Lima existe 1,600,000 Pymes y Mypes, estas empresas privadas y otras públicas requieren software de calidad para su gestión.

Estudiamos adaptar la metodología ágil Iconix en las fases de análisis de requisitos, diseño e implementación, mediante técnicas probadas de ingeniería de software con la finalidad de mejorar la calidad del producto software, el estudio se realiza en la Ciudad de Lima; el tipo de investigación es prospectivo, longitudinal, analítico y nivel aplicativo.

Se aplicó el diseño cuasi-experimental, interviniendo las tareas de la metodología ágil Iconix, estas tareas finales intervenidas con técnicas formales de ingeniería de software, persiguen el objetivo de mejorar la calidad del producto software, luego se realiza una encuesta a expertos en desarrollo ágil de software, antes y después de la intervención a la metodología, los datos recolectados son procesados mediante las pruebas: T- Student, la homocedasticidad y análisis de varianza, a fin de contrastar las hipótesis de investigación formuladas.

Se concluye que la intervención a las fases de: análisis de requisitos mediante la inclusión de ingeniería de requisitos, diseño mediante la inclusión de patrones de diseño, implementación mediante la inclusión de técnicas de programación orientada a objetos con los principios Solid mejoran significativamente la calidad del producto software. En resumen, la intervención a la metodología ágil Iconix con técnicas de ingeniería de software mejoran significativamente la calidad del producto software.

Palabras clave: Metodología ágil Iconix, Calidad de productos software, Ingeniería de requisitos, Patrones de diseño, Principios Solid.

ABSTRACT

Peru is a nation immersed in economic and social globalization, to be a competitive country it needs the national software industry to develop information systems with a high degree of software product quality, according to INEI in Lima there are 1,600,000 Pymes and Mypes. Private and other public companies require quality software for their management.

We study adapting the agile Iconix methodology in the phases of requirements analysis, design and implementation, using proven software engineering techniques in order to improve the quality of the software product, the study is carried out in the City of Lima; The type of research is prospective, longitudinal, analytical and application level.

The quasi-experimental design was applied, intervening the tasks of the agile Iconix methodology, these final tasks intervened with formal techniques of software engineering, pursue the objective of improving the quality of the software product, then a survey is carried out to experts in agile development of software, before and after the intervention to the methodology, the collected data are processed through the tests: T-Student, homocedasticity and analysis of variance, in order to contrast the research hypotheses formulated.

It is concluded that the intervention to the phases of: requirements analysis through the inclusion of requirements engineering, design through the inclusion of design patterns, implementation through the inclusion of object oriented programming techniques with the Solid principles significantly improve the quality of the software product. In summary, the intervention to the agile Iconix methodology with software engineering techniques significantly improves the quality of the software product.

Key words: Agile Iconix methodology, Software product quality, Requirements engineering, Design patterns, Solid principles

INTRODUCCION

La metodología ágil Iconix tiene las fases de: análisis de requisitos, con tareas y productos software: definir los requisitos funcionales y no funcionales, identificar objetos del mundo real y construir el modelo de dominio, realizar prototipos de interfaz gráfica de usuario, identificar los casos de uso y describir el primer borrador de casos de uso (Rosenberg y Scott, 2001). En la fase de diseño, se realiza: diseño preliminar, descripción de los casos de uso completa, correcta, detallada y explícita, identificar los estereotipos de objetos, diagrama de secuencia y clases (Rosenberg y Scott, 1999). Implementación, definir la arquitectura técnica, codificar desde el diagrama de secuencia, sincronizar el diseño y código fuente, centrarse en las pruebas unitarias y de integración (Rosenberg y Stephens, 2007).

El modelo de calidad para un producto software, está compuesto por la calidad externa e interna según la NTP ISO/IEC 9126-1:2004 (2004), que presenta características y sub características. Según Scalone (2006) el modelo de calidad está compuesto por la calidad externa e interna de los productos software, existen modelos y estándares que poseen un conjunto de características, sub características, atributos y métricas, el equipo de desarrollo debe evaluar la calidad del producto software durante el ciclo de vida del software.

Tome la decisión de investigar en la línea de metodologías agiles para desarrollo de software y calidad del producto software, porque las instituciones que forman Ingenieros para el desarrollo del Perú, necesitan nuevas formas de hacer software con agilidad que tiene beneficios como: disminuir el tiempo y los costos, y mejorar la calidad del producto software, muy necesario para miles de empresas.

Los objetivos específicos planteados son: desarrollar la fase de análisis de requisitos mediante la inclusión de ingeniería de requisitos a fin de mejorar la calidad del producto software, desarrollar la fase de diseño mediante la inclusión de patrones de diseño a fin de mejorar la calidad del producto software, desarrollar la fase de implementación mediante la inclusión de técnicas de programación orientada a objetos con la finalidad de mejorar la calidad del producto software

.

I. PLANTEAMIENTO DEL PROBLEMA

1.1. DESCRIPCION DEL PROBLEMA

La industria nacional e instituciones formadoras de ingenieros de sistemas, estudian procesos agiles para desarrollar software alineados a la calidad de productos software limitadamente. Siendo el Perú una nación inmersa en la globalización económica y social, debe buscar ser un país competitivo, para esto se necesita que la industria de software garantice sistemas de información con alto grado de calidad de productos software, y que apoye la gestión de instituciones públicas y privadas; el logro de estas premisas será posible si estudiamos nuevas técnicas probadas de ingeniería de software como: ingeniería de requisitos, patrones de diseño y técnicas de programación orientada a objetos, como es el presente caso, intervenir la metodología ágil Iconix.

La industria nacional de software para gestión de negocios, nace el año 1974 con la creación de la carrera profesional de Ingeniería de Sistemas en la Universidad Nacional de Ingeniería, paralelamente se instala la empresa norteamericana IBM del Perú, empresa que comienza a brindar servicios de desarrollo y mantenimiento de software para las empresas financieras, públicas y privadas, en la década del 80, los ingenieros de sistemas comienzan a aplicar métodos de desarrollo de software estructurado, paradigma para desarrollo de software que ya no se usa; durante la década del 90, aparece avances para desarrollar software como: desarrollo de software ágil, análisis y diseño orientado a objetos y la notación UML propuesto por Grady, Booch y Rumbaugh, estos paradigmas han contribuido al desarrollo de software, estos aportes carecen del alineamiento con la calidad de productos software.

Estudiaremos la adaptación de la metodología ágil Iconix, porque en Lima se concentran 1,600,000 Pymes y Mypes, según el INEI del año 2013, estas empresas requieren software para su gestión, es necesario plantear nuevas formas para desarrollar y mantener software con metodologías agiles que garantice la aplicación de técnicas probadas de ingeniería de software, en las actuales condiciones, muy temprano en la fase de análisis de requisitos, un error costará diez veces más de lo invertido para su corrección cuando se identifica las fallas al momento del uso, que cuando se detecta esta durante la fase de análisis y de la misma forma al cometer errores en las fases de diseño, implementación y pruebas.

El estudio aportará en el ámbito social, porque la intervención a la metodología ágil Iconix con técnicas formales de ingeniería de software, debe garantizar un grado de calidad de productos software, que beneficie al micro, pequeño o mediano empresario que adquiere un software de calidad. En el aspecto teórico y práctico, es una contribución a la academia durante la formación de técnicos y profesionales que desarrollan y mantienen software.

1.2. FORMULACION DEL PROBLEMA

1.2.1. Problema Principal

¿De qué manera adaptar la metodología ágil Iconix para mejorar la calidad del producto software, Lima, 2017?.

1.2.2. Problemas Secundarios

a. ¿Cómo modificar la fase de análisis de requisitos para mejorar la calidad del productos software?.
b. ¿Cómo variar la fase de diseño para mejorar la calidad del producto software?.
c. ¿Cómo transformar la fase de implementación para mejorar la calidad del producto software?.

1.3. ANTECEDENTES DEL PROBLEMA

1.3.1. Antecedentes Internacionales

Nema, Rodríguez, Oivo y Tosun (2016), en el estudio "Análisis del concepto de deuda técnica en el contexto del desarrollo de software ágil: una revisión sistemática de la literatura", tiene el objetivo de analizar y sintetizar el estado del arte de la deuda técnica (TD); se encontró áreas de investigación relacionadas a TD en desarrollo de software ágil (ASD), estas áreas son: la gestión de TD en ASD, arquitectura en ASD y su relación con TD, las causas más populares de incurrir TD en ASD fueron: enfoque en la entrega rápida y problemas de arquitectura y diseño, las consecuencias más significativas son: productividad reducida, degradación del software, mayor costo de mantenimiento, estrategias para manejar TD en el contexto de ASD, refactorización; concluyendo que los resultados del estudio proporcionan una síntesis estructurada de TD y su análisis en el contexto de ASD.

Yanga, Lianga y Avgerioub (2015), en la investigación "Un estudio de mapeo sistemático sobre la combinación de arquitectura de software y desarrollo ágil", el estudio analiza la combinación de arquitectura, métodos ágiles y explora: las actividades y enfoques de la arquitectura, métodos y prácticas ágiles, costos, beneficios, desafíos, factores, herramientas y lecciones aprendidas sobre la combinación; revisaron publicaciones entre febrero de 2001 y enero de 2014. Los resultados del análisis de **c**incuenta y cuatro estudios, presentan los aspectos más destacados: (a) Existe diferencia significativa en la proporción de diversas actividades de arquitectura, métodos ágiles y prácticas ágiles empleadas en la combinación. (b) Ninguno de los enfoques de arquitectura ha sido ampliamente utilizado en la combinación. (c) Hay una falta de descripción y análisis respecto a los costos y las historias fallidas de la combinación. (d) Se identificaron veinte desafíos, veintinueve factores y veinticinco lecciones aprendidas. Las conclusiones ayudan a la comunidad de ingeniería de software a reflexionar sobre los últimos trece años de investigación y práctica de la combinación arquitectura y desarrollo ágil.

Silveira y Silva (2015), en el estudio "Adaptación de métodos ágiles: una revisión sistemática de la literatura", indican que la industria de desarrollo de software ha adoptado métodos ágiles en lugar de métodos tradicionales, porque son más flexibles y pueden traer beneficios para administrar cambios de requerimientos, mejorar la productividad y la alineación con el cliente; el objetivo del estudio fue: evaluar, sintetizar y presentar características sobre la adaptación de métodos ágiles, y los criterios utilizados para la selección de prácticas ágiles, han revisado 783 estudios publicados entre 2002 y 2014, se analizó 56 artículos como descriptores de enfoques de adaptación de métodos ágiles. La adaptación de métodos ágiles puede considerarse madura, ya que aproximadamente 2/3 de los artículos utilizan métodos de investigación empírica, por la necesidad que las organizaciones establezcan y adopten modelos para la adaptación de métodos ágiles; según el entorno interno utilizaron variables como el tipo de proyecto, la comunicación, la cultura y el soporte de gestión, de acuerdo a los objetivos se utilizaron variables como los objetivos comerciales, el grado de innovación y la complejidad.

Rodríguez, Musat y Yagüe (2010), en el estudio "Adopción de metodologías ágiles: un estudio comparativo entre España y Europa", el objetivo fue analizar el estado de la adopción de metodologías ágiles en la industria de software española comparado con la

europea, se aplicó cuestionarios para evaluar el uso de diferentes metodologías y prácticas ágiles, las estrategias empleadas en el proceso de adopción, los factores que motivan su uso, los beneficios que se reportan, limitaciones y retos que implican su aplicación. En España, el estudio se realizó en las organizaciones que participaron en el Agile Open Spain 2009, en Europa, la encuesta se realizó en organizaciones del proyecto Flexi, pioneras en la adopción de metodologías ágiles a nivel mundial. Los resultados muestran: la aplicación de metodologías ágiles en la industria española es un enfoque joven, 44% no tenía ninguna experiencia y 22% tenía menos de un año de experiencia. En Flexi, 60% tenía experiencia de 1 a 5 años y 20% más de 5 años, en España se adopta las metodologías ágiles a nivel local y personal, en europea presenta un nivel de madurez mayor, adoptan metodologías ágiles en proyectos de equipos distribuidos. Se concluye que las empresas europeas centraran sus esfuerzos en aplicación a equipos de mayor tamaño y, las empresas españolas están eliminando las barreras existentes para la aplicación de las metodologías ágiles.

Holvite, et al. (2017), en el estudio "Deuda técnica y desarrollo de software ágil prácticas y procesos: Una encuesta a los profesionales de la industria", se estudia cómo la deuda técnica se manifiesta y afecta los procesos de software, y cómo las técnicas de desarrollo de software empleadas mitigan la presencia de esta deuda; el objetivo fue buscar los conocimientos y las experiencias de los profesionales para clasificar los efectos del uso de métodos ágiles en la gestión de la deuda técnica, se realizó una encuesta multinacional para abordar los objetivos, recibiendo 184 respuestas de profesionales en Brasil, Finlandia y Nueva Zelanda. Los resultados indican que: (a) Los profesionales son conscientes de la deuda técnica, aunque pocos conocen el concepto. (b) La deuda técnica comúnmente reside en sistemas heredados, los casos concretos de deuda técnica son difíciles de conceptualizar, haciendo la gestión problemática. (c) Las prácticas y procesos ágiles consultados ayudan a reducir la deuda técnica, en particular, las técnicas que verifican y mantienen la estructura y la claridad de los productos software implementado, ejemplo, estándares de codificación y refactorización, afectan positivamente la gestión técnica de la deuda. Se concluye que, a pesar de los efectos positivos de algunas prácticas ágiles en la gestión de la deuda técnica, los intereses de las partes siguen siendo una preocupación.

Britto (2014), en la tesis "Adaptación de un proceso de desarrollo de software basado en buenas prácticas", trata el desarrollo de software mediante las mejores prácticas obtenidas con las metodologías agiles: scrum, programación extrema e Iconix, aplicando el modelo de

mejora y evaluación CMMI-DEV 1.3, se ha evaluado el proceso actual y comparando con los procesos ágiles, seleccionando las de mayor posibilidad de mejora, luego ha diseñado el nuevo proceso aplicando estas prácticas seleccionadas. El nuevo proceso fue sometido a una prueba piloto y evaluado respecto a CMMI, obteniendo una matriz de comparación frente a CMMI para evaluar el proceso de desarrollo propuesto, se observa una mejora en el cumplimiento de buenas prácticas frente a CMMI del 25% con respecto al proceso anterior.

Figueroa, Solis y Cabrera (2007), en el estudio de comparación entre las metodologías tradicional y ágil, informan que para desarrollar software que presenten calidad de productos, debe elegirse el mejor proceso para un equipo y un proyecto, mediante los enfoques de procesos tradicionales y procesos ágiles. La tabla 1, muestra la diferencia entre procesos tradicionales y agiles, según las tablas de 2 a 4 observamos; las diferencias, ventajas, desventajas y como seleccionar un proceso para un determinado proyecto de software.

Tabla 1. Diferencias entre procesos tradicionales y ágiles

Procesos Tradicionales	**Procesos Agiles**
Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo	Basadas en heurísticas provenientes de prácticas de producción de código
Cierta resistencia a los cambios	Especialmente preparados para cambios durante el proyecto
Impuestas externamente	Impuestas internamente, por el equipo.
Proceso mucho más controlado, con numerosas políticas y normas	Proceso menos controlado, con pocos principios
El cliente interactúa con el equipo de desarrollo mediante reuniones	El cliente es parte del equipo de desarrollo
Más artefactos	Pocos artefactos
Más roles	Pocos roles
Grupos grandes y posiblemente distribuidos	Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio
La arquitectura del software es esencial y se expresa mediante modelos	Menos énfasis en la arquitectura del software
Existe un contrato prefijado	No existe contrato tradicional o al menos es bastante flexible

Tabla 2. Diferencias por etapas y enfoque metodológico

Modelos Rigurosos	Etapa	Modelos Agiles
Planificación predictiva y "aislada"	Análisis de requerimientos	Planificación adaptativa: Entregas frecuentes + colaboración del cliente
	Planificación	
Diseño flexible y Extensible + modelos + Documentación exhaustiva	Diseño	Diseño Simple: Documentación Mínima + Focalizado en la comunicación
Desarrollo individual con Roles y responsabilidades estrictas	Codificación	Transferencia de conocimiento: Programación en pares + conocimiento colectivo
Actividades de control: Orientado a los hitos + Gestión de miniproyectos	Pruebas	Liderazgo-Colaboración: Empoderamiento + Autoorganización
	Puesta en Producción	

Tabla 3. Modelo de procesos versus características del proyecto

Modelo de Proceso	Tamaño del Proceso	Tamaño del Equipo	Complejidad del Problema
RUP	Medio / Extenso	Medio / Extenso	Medio / Alto
ICONIX	Pequeño / Medio	Pequeño / Medio	Pequeño / Medio
XP	Pequeño / Medio	Pequeño	Medio / Alto
SCRUM	Pequeño / Medio	Pequeño	Medio / Alto

Tabla 4. Modelos de proceso versus ayudas para el modelo

Modelo de Proceso	Curva de Aprendizaje	Herramienta de Integración	Soporte Externo
RUP	Lenta	Alto soporte	Alto soporte
ICONIX	Rápida	Algún soporte disponible	Algún soporte disponible
XP	Rápida	No mencionado	Algún soporte disponible
SCRUM	Rápida	No mencionado	Algún soporte disponible

Bona (2002), en el estudio "Avaliação de processos de software: Um estudo de caso em XP e Iconix", considera que Iconix es un proceso de desarrollo ágil, usa la orientación a

objetos, tiene las fases: análisis de requisitos, diseño preliminar, diseño e implementación, su esencia se resume como "definición de modelos de objetos a partir de los casos de uso". Está dirigido por casos de uso, presenta proceso iterativo e incremental, mantiene enfoque direccionado a la rastreabilidad de los requisitos. Iconix, tiene las ventajas siguientes:

a. ¿Quiénes son los usuarios y que hacen?. Son actores del sistema, usan los casos de uso;
b. ¿Qué son en el "mundo real"?. El dominio de problema;
c. ¿Qué objetos y asociaciones existen entre ellos?. Diagramas de clases de alto nivel;
d. ¿Qué objetos son necesarios para cada caso de uso?. Diagramas de robustez;
e. ¿Cómo los objetos están colaborando e interactuando en cada caso de uso?. Diagramas de secuencia;
f. ¿Cómo será construido el sistema a nivel práctico?. Diagramas de clases de bajo nivel.

Tarhan y Yilmaz (2013), en el estudio "Análisis sistemático y comparación del rendimiento de desarrollo y la calidad de productos software del proceso incremental y el proceso ágil", los resultados muestran que el proceso ágil tiene mejor desempeño que el incremental en términos de productividad (79%), densidad de defectos (57%), relación de esfuerzo de resolución de defectos (26%), efectividad de ejecución de pruebas V & V (21 %) y la capacidad de predicción de esfuerzo (4%). Los resultados sobre rendimiento de desarrollo y la calidad de productos software logrado con el proceso ágil fue superior al aplicar el proceso incremental en los proyectos comparados, en conclusión: la medición, análisis y comparación permitieron la revisión integral de los dos procesos de desarrollo, y dieron como resultado la comprensión de sus fortalezas y debilidades, los resultados de la comparación son una evidencia objetiva para usar los procesos agiles en la empresa.

Kupiainena, Mäntyläa y Itkonena (2015), en el estudio "Uso de métricas en el desarrollo de software ágil y eficiente: Una revisión sistemática de la literatura", tienen por objetivo ampliar el conocimiento, las razones y los efectos del uso de métricas en el desarrollo industrial de software ágil, enfocado en las métricas que usan los equipos ágiles, en lugar de las que usan los investigadores de ingeniería de software; los resultados indican que las razones y los efectos del uso de la métrica se centran en las siguientes áreas: planificación de actividades, seguimiento del progreso, medición de calidad de productos software, mantenimiento del

proceso de software y motivación de las personas; concluyen que el uso de métricas en el desarrollo de software ágil es similar al desarrollo de software tradicional, donde los proyectos y actividades cortas necesitan ser planificadas y seguidas, la calidad tiene que ser medida y, los problemas del proceso deben ser identificados y corregidos.

Domínguez, Escalona, Mejías, Ross y Staples (2012), en el estudio "Evaluación de calidad para metodologías de ingeniería web basadas en modelos", indican que hay muchas metodologías en la ingeniería web basada en modelos, las metodologías de ingeniería web basada en modelos evolucionan constantemente y la calidad es un factor muy importante para identificar una metodología, que define procesos, técnicas y artefactos para desarrollar aplicaciones web, por esto al analizar una metodología, no solo es evaluar la calidad, sino también averiguar cómo mejorarla, en el contexto de los productos de software, algunos estándares ampliamente propuestos, como ISO/IEC 9126 o ISO / IEC 25000, sugieren un modelo para calidad de productos software; los resultados recomiendan un conjunto de características de calidad y sub características, basadas en estos estándares para evaluar la calidad de las metodologías de ingeniería web basada en modelos y, define una forma ágil de relacionar estas sub características de calidad con los atributos de los productos, concluyen que la aplicación de estas características de calidad y sub características podría promover la eficiencia en las metodologías de desarrollo web basado en modelos.

Hansen, Jonasson y Neukirchen (2011), en la investigación "Un estudio empírico de la arquitecturas de software y su efecto sobre la calidad del producto", afirman que la arquitectura de software se refiere a la estructura de los sistemas de software y generalmente se acepta que influya en la calidad de productos software; existe poca investigación empírica sobre la relación entre la arquitectura del software y la calidad de productos software, el estudio considera a 1141 proyectos de código abierto en java, han calculado tres métricas de arquitectura de software que son: medición de clases por paquete, distancia normalizada y el exceso del grado de acoplamiento, analizan que la medida de estas métricas están relacionadas con las métricas del producto como: ratio de defectos, velocidad de descarga, métodos por clase y complejidad del método, concluyen que hay una serie de relaciones significativas entre las métricas de productos y las métricas de arquitectura, particularmente, la cantidad de defectos abiertos depende significativamente de todas las medidas de arquitectura.

Orantes (2006), en el estudio "Calidad de software usando metodologías ágiles para el

desarrollo de software", indica que existe propuestas metodológicas con incidencia en el proceso de desarrollo y propuestas tradicionales en el control del proceso, estableciendo formas rigurosas de actividades, de producto software, herramientas a usar y notaciones; las propuestas han demostrado ser efectivas y necesarias en varios proyectos, pero han presentado problemas en otros, se sugieren mejoras, como adicionar a los procesos de desarrollo más actividades, artefactos y restricciones, como consecuencia, se tendría un proceso de desarrollo complejo limitando el desarrollo con agilidad. En búsqueda de soluciones, se encuentran metodologías centradas en otros aspectos, como el factor humano o producto software, siendo esta la filosofía de las metodologías agiles, dando mayor importancia a la colaboración con el cliente y al desarrollo incremental del software con iteraciones cortas, mostrando su efectividad en proyectos con requisitos cambiantes y reducción drásticamente de tiempos de desarrollo, pero manteniendo alta calidad de productos software.

Freitas, Da Mota, Neto, O'Leary y Santana (2014) en la investigación "Usando un enfoque de métodos múltiples para comprender las pruebas en línea de producto de software (SPL) y la agilidad", el objetivo es comprender el desarrollo con agilidad y SPL y mejorar la generalización de la evidencia identificada, el enfoque considera tres métodos que son: estudio de mapeo, estudio de caso y opinión de expertos, los resultados muestran 23 hallazgos que brindan evidencia sobre cómo combinar agilidad y SPL; en conclusión, el enfoque requiere mucho tiempo y un alto grado de esfuerzo para planificar, diseñar y realizar, pero ayuda a aumentar la comprensión sobre SPL.

Da Mota, Do Carmo, Machado, McGregor, Santana y Romero (2010), en la investigación "Un estudio de mapeo sistemático de pruebas en líneas de productos de software", durante el desarrollo de software, las pruebas son importantes para identificar defectos y asegurar que los productos software funcionen según lo especificado, siendo práctica común y valida, los objetivo del estudio se enfocan en: investigar las prácticas de pruebas, identificar las brechas entre las técnicas requeridas y los enfoques existentes, se evaluaron 120 estudios de 1993 a 2009, los resultados son: varios aspectos relacionados con las pruebas en línea han sido cubiertos para desarrollo de un software, no se pueden aplicar directamente SPL por problemas específicos. Los aspectos específicos sobre SPL no están cubiertos por los enfoques de SPL existentes y, cuando se cubren estos aspectos, la literatura ofrece resúmenes breves; se concluye que se debe realizar investigación adicional, empírica y práctica.

Inayat, Salim, Marczak, Daneva y Shamshirband (2014), en la investigación "Un framework

para estudiar la colaboración basada por los requisitos entre equipos ágiles: resultados de dos casos de estudio"; el estudio indica que se requiere la colaboración de los interesados para la elicitación de requisitos durante el desarrollo de software con agilidad, la ingeniería de requisitos y los métodos ágiles comparten una base colaboración entre interesados, se estudió dos casos para examinar las redes de comunicación y conocimiento entre los equipos ágiles, los resultados del framework ayudan a determinar: los miembros principales del equipo, tendencias de colaboración, tendencia a la agrupación, reciprocidad de comunicación, concientización de los equipos y, rendimiento de interacción de los equipos ágiles.

Daneva, Van der Veen, Amrit, Ghaisas, Sikkel y Kum (2012), en el estudio "Priorización de requisitos ágiles en proyectos de sistemas tercerizados a gran escala: un estudio empírico", nos informan que la aplicación de prácticas ágiles para la priorización de requisitos en proyectos distribuidos y tercerizados es una tendencia reciente; se utilizó entrevistas para la recopilación de datos, los hallazgos fueron: (a) Comprender las dependencias de los requisitos es importante para el despliegue exitoso de enfoques ágiles. (b) El criterio de priorización es más importante en el establecimiento de grandes proyectos ágiles tercerizados. (c) Se ha desarrollado un nuevo producto software valioso para el desarrollo ágil de software denominada "historias de entrega", las historias de usuarios se complementan con las implicaciones técnicas, estimación de esfuerzo y riesgo asociado; las historias de entrega juegan un papel fundamental en la priorización de los requisitos. (d) El uso de las prácticas de priorización ágil depende del tipo de acuerdo en la tercerización del proyecto.

Belfo (2012), en el artículo "Personas, dimensión organizacional y tecnológica para especificación de requerimientos de software", indican que la especificación de requisitos inapropiada es una razón del fracaso en desarrollo de software, este fracaso es porque la especificación de los requisitos tiende a sobrevalorar lo tecnológico de los requisitos, los buenos requisitos se garantizan por la combinación correcta de tres dimensiones: personas, organización y tecnología, mediante estas tres dimensiones se ha llegado a los resultados: (a) La comunicación y colaboración continua es el método más usado para involucrar a las partes interesadas en el proceso desarrollo con ASD. (b) Respecto al usuario, se debe hacer ASD más centrado en este, como método útil de desarrollo de software ágil. (c) Existen problemas sobre la documentación de los requisitos para identificar los productos, para mejorar la colaboración entre las partes interesadas, y el equipo desarrollador ágil, se necesita la creación de directrices apropiadas para la gestión de requisitos dentro de ASD.

Baruah (2015), en la conferencia “Gestión de requerimientos en un entorno de software ágil”, concluye que la gestión de requisitos para cualquier metodología de desarrollo de software se da durante el ciclo de vida del sistema, en la industria del software los requisitos cambiantes del lado del cliente dificultan que los desarrolladores produzcan un software de calidad, no existe un enfoque adecuado para gestionar los requisitos que cambian con frecuencia durante el ciclo para cumplir con los requisitos del cliente. Las metodologías ágiles de desarrollo de software, admiten cambios en los requisitos, pero se debe garantizar la necesidad del cliente escuchándolo en todas las fases del desarrollo.

Kumar (2015), en el estudio “Investigando el cálculo de la recompensa de penalización de los usuarios de software y su impacto sobre la priorización de requisitos”, uno de los objetivos del estudio fue evaluar la eficacia relativa de 5 métodos de priorización de requisitos en la satisfacción del usuario sobre conjuntos de requisitos prioritarios; según uno de los resultados del experimento se propone el método Kano, para priorizar los requisitos con satisfacción del usuario; en conclusión el estudio demuestra empíricamente las complejidades de priorizar los requisitos de software y exige una nueva generación de métodos para abordarlos, comprender y resolver estas complejidades que permitirá a los proveedores de software, seleccionar de manera cuidadosa solo aquellos requisitos para la implementación del producto software que tienen un máximo impacto en la satisfacción del usuario.

Azadegan, Papamichail y Sampaio (2015), en la investigación “Aplicación de diseño de proceso colaborativo en la obtención de requisitos del usuario: Un caso de estudio”, la ingeniería de requisitos es un proceso técnico, social y cognitivo complejo, que produce requisitos para un sistema de software intensivo (Maiden y Hare, 1998), la obtención de requisitos es una actividad importante en el proceso de ingeniería de requisitos ya que implica descubrir y formalizar los requisitos del sistema (Sun, Zeng y Liu, 2011). En las dos últimas décadas, la obtención de requisitos, se ha trasladado a un entorno de colaboración de empresas extendidas, debido a la globalización de los mercados, el avance de las tecnologías, la segregación de las demandas de los clientes y la competencia interna y externa. La obtención de requisitos es altamente colaborativa e involucra a muchas partes interesadas: los responsables de la toma de decisiones de la organización que decidirán el origen del sistema y negociarán los requisitos funcionales y no funcionales, junto con el precio a pagar, el usuario que interactúa con él, el dominio de representante expertos, de

ventas y marketing y los desarrolladores que lo construyen (Nuseibeh y Easterbrook, 2000; Katasonov y Sakkinen, 2005). Las partes interesadas tiene diferentes necesidades, junto con su propia experiencia, prejuicios y puntos de vista que deben ser satisfechos en la entrega del sistema futuro (Robertson, 2001). Se conoce que hasta el 80% del costo del producto está determinado por las decisiones tomadas en colaboración con los interesados, durante las primeras etapas del ciclo de vida de los requisitos; la gestión efectiva del diseño del producto, la capacidad de los interesados y lo que esperan las partes interesadas de los requisitos, juega un papel importante en todo el ciclo de vida del producto.

Guerrero, Suárez y Gutiérrez (2013), en la investigación sobre "Patrones de diseño en el contexto de procesos de desarrollo de aplicaciones orientadas a la web", analizan patrones de diseño definidos por The Gang of Four (GoF), los resultados indican que de 23 patrones existentes, 8 patrones de diseño GoF son usados en desarrollo de software, los expertos usan 10 patrones de diseño GoF, las tablas del 5 a 9 muestran los resultados del estudio.

Tabla 5. Patrones de diseño creacionales identificados en proyectos de desarrollo

Patrón	**Cantidad de procesos de desarrollo que lo utilizaron**	**Porcentaje sobre el total de procesos de desarrollo (36)**
Abstract factory	0	0
Builder	15	42
Factory method	18	50
Prototype	0	0
Singlenton	24	67

Tabla 6. Patrones de diseño estructurales identificados en proyectos de desarrollo

Patrón	**Cantidad de procesos de desarrollo que lo utilizaron**	**Porcentaje sobre el total de procesos de desarrollo (36)**
Adapter	0	0
Bridge	0	0
Composite	0	0
Decorator	14	39
Facade	13	36
Flyweight	0	0
Proxy	0	0

Tabla 7. Patrones de diseño de comportamiento identificados en proyectos de desarrollo

Patrón	Cantidad de procesos de desarrollo que lo utilizaron	Porcentaje sobre el total de procesos de desarrollo (36)
Chain of reponsability	0	0
Command	0	0
Interpreter	0	0
Iterator	20	39
Mediator	13	36
Memento	0	0
Observer	0	0
State	0	0
Strategy	15	0
Template method	17	0
Visitor	0	0

Tabla 8. Patrones GoF empleados en proyectos software para aplicaciones web

Categoría GoF	Patrones
Creacionales	1. Builder 2. Factory method 3. Singleton
Estructurales	1. Decorator 2. Facade
De comportamiento	1. Iterator 2. Strategy 3. Template method

Tabla 9. Patrones GoF empleados por expertos

Categoría GoF	Patrones
Creacionales	1. Abstract Factory 2. Builder 3. Factory method 4. Singleton
Estructurales	1. Decorator 2. Facade

De comportamiento	1. Iterator 2. Observer 3. Strategy 4. Template method

En conclusión, no se usa los 23 patrones de diseño para el desarrollo de aplicaciones web, los expertos afirman que “Usar un patrón de diseño requiere de años de experiencia, a veces se puede leer y leer sobre un patrón y no entenderlo y otras veces aplicarlo mal”.

Jiménez, Tello y Ríos (2014), en el estudio “Lenguajes de patrones de arquitectura de software: Una aproximación al estado del arte”, el propósito es mostrar el estado del arte en un área de la arquitectura de software llamada "Lenguajes de patrones", desde sus orígenes, los avances actuales y sus aplicaciones en la construcción de arquitecturas de software en diferentes dominios de aplicación. Se concluye que la evolución de la ingeniería y arquitectura de software ha llegado a niveles superiores, que permite resolver diversos problemas de arquitectura con mejor calidad y en menor tiempo, en trabajos futuros es importante profundizar en metodologías y procesos que han seguido expertos del área para resolver sus problemas, esto abre las puertas a encontrar métodos eficientes y de referencia al momento de iniciar un diseño de lenguaje de patrones.

Palacios, García, Oliva y Granollers (2015), en la investigación “Exploración de patrones de interacción para su uso en la web semántica”, afirman que en los últimos años se han publicado múltiples propuestas de conjuntos de patrones de interacción aplicables al diseño y desarrollo de interfaces web, paralelamente, la web semántica (WS) mejora los sitios web con contenidos semánticos. Mediante estudios preliminares, se ha definido una lista de tareas elementales de usuario final para la WS, proponen un conjunto de tareas que incluye: navegar, anotar, mezclar, mapear, compartir, comunicar y tramitar; finalmente han analizado cómo los patrones de interacción pueden mejorar la funcionalidad de las tareas de usuario final en el contexto de la WS.

Pereira (2013), en la tesis “Principios de diseño SOLID aplicados para la mejora de código fuente en sistemas orientados a objetos”, afirma que los aspectos que definen la calidad de productos software, pasa por hacer el proyecto y desarrollar el software, compuesto por productos de software, el código fuente producido, existen diversas formas de creación de

mejores códigos, entre los cuales están los "Principios SOLID de diseño", se estudia el impacto que estos principios pueden generar en el proceso de desarrollo de un sistema orientado a objetos, mediante las métricas de la norma ISO/IEC 9126 para la calidad de productos software, se concluye que la mejora de calidad de productos software requiere mucha disciplina y conocimiento; los principios SOLID traen una percepción diferente de los problemas que afectan el código fuente de sistemas desarrollados y la calidad.

1.3.2. Antecedentes Nacionales

Huanca (2015), en la tesis “Revisión sistemática de la calidad del software en prácticas ágiles”, afirma que el desarrollo de software ágil representa un alejamiento importante de los enfoques tradicionales con planificación detallada, la revisión sistemática de la literatura hasta el año 2014, presenta resultados de estudios empíricos sobre la evaluación de la calidad en prácticas ágiles para el estándar ISO/IEC 25010, se analizó cinco grupos: programación en pares, desarrollos guiados por pruebas, extreme programming, scrum y otras prácticas ágiles. Los resultados sugieren que las prácticas ágiles pueden ayudar a mejorar la calidad de productos software si son aplicadas correctamente.

Samamé (2013), en la tesis titulada “Aplicación de una metodología ágil en el desarrollo de un sistema de información”, resume que las aplicaciones informáticas están en nuestras vidas directa o indirectamente y somos consumidores o desarrolladores de ellas; en la actualidad se presentan diversas metodologías para un proyecto de software como las clásicas y las ágiles, para el caso se usó la metodología ágil programación extrema, se concluye que: se aplicó herramientas como Java y XML, se obtiene un sistema con portabilidad, la filosofía de programar y probar es un gran aporte de la metodología, sintetizar o refactorizar el código genera un código más limpio y fácil de mantener.

Cano (2015), en la tesis denominada “Revisión sistemática de comparación de modelos de procesos software”, se desarrolla la revisión de literatura a fin de proponer un esquema de comparación que aporte a la crisis del software a nivel mundial caracterizada por la baja calidad de productos software, revisión que involucra el análisis de diversos modelos de proceso y producto como: comparación basados en implementaciones, comparación según un esquema sistemático y manual, esquema basado en una representación gráfica denominada composition trees. De acuerdo a los modelos analizados se ha determinado que los modelos pueden ser mejorados y en algunos casos combinados para obtener mejores prestaciones sobre

la calidad del proceso y producto software.

Salvador (2013), en la tesis “Una revisión sistemática de usabilidad en metodologías ágiles”, afirma que en los últimos años, se han aplicado técnicas de evaluación de usabilidad en el desarrollo de software, en metodologías ágiles éstas técnicas se están considerando, porque se han propuesto mejorar la calidad de productos software; la estrategia de búsqueda abarco 307 artículos, se seleccionó 32. Los resultados indican que las técnicas de usabilidad utilizadas con mayor frecuencia son: prototipo rápido (40%), indagación individual (37%), pruebas formales de usabilidad (25%) y evaluaciones heurísticas (18%), estos resultados han permitido conocer el estado actual de las técnicas de evaluación de usabilidad en metodologías ágiles.

1.4. JUSTIFICACIÓN E IMPORTANCIA DE LA INVESTIGACION

1.4.1. Justificación de la Investigación

El estudio está enmarcado en las líneas de investigación, metodología ágil para desarrollar software y calidad del producto software, que presenta: tareas, técnicas, productos software y responsables de desarrollo del software, los procesos adaptados permitirán desarrollar software orientado a micro, pequeñas y medianas empresas que requieren software con calidad de productos. Por otra parte, se debe identificar y seleccionar las características y sub características de calidad de productos software y alinearlos al desarrollado con la metodología ágil Iconix, para que sea evaluado mediante el estándar de calidad con la NTP ISO/IEC 9126. Teóricamente se justifica porque se adapta las tareas actuales de la metodología ágil Iconix, para sustituirlos mediante técnicas probadas de ingeniería de software alineados a la calidad de productos software; la comunidad académica y las empresas proveedoras de software, necesitan disponer de procesos agiles formales y probados por expertos en desarrollo de software con agilidad.

La industria nacional del software es incipiente, presentando escasa oferta de desarrollo de software nacional orientados a micro, pequeñas y medianas empresas, se requiere formular una metodología para desarrollar software que incorpore técnicas probadas de ingeniería de software, que mejore la calidad del producto software, y sirva como guía para el desarrollo de proyectos pequeños y medianos de software.

Se plantea formular un proceso adaptado considerando la intervención a la metodología ágil Iconix, para desarrollar software urgente y sostenible, aplicable a proyectos de

software pequeño y mediano, validado mediante la evaluación de calidad del producto software durante el desarrollo, mediante los atributos de los productos software y la norma NTP ISO/IEC 9126 y sus extensiones. Es decir intervenir a los procesos del análisis de requisitos mediante la ingeniería de requisitos, variar la fase de diseño aplicando patrones de diseño y, transformar la fase de implementación con técnicas de la programación orientada a objetos, con el objetivo de mejorar la calidad del producto software.

La intervención propuesta a la metodología ágil Iconix, puede ser usado para producir y mantener software con grado de calidad de productos aceptable, que los software producidos en el Perú mediante este proceso adaptado, sean adquiridos porque tienen un grado de calidad aceptable, que la inversión en software para la gestión de estos negocios tenga una rentabilidad para la empresa, por el contrario que no sea un gasto.

1.4.2. Importancia de la Investigación

Se podrá disponer de la metodología ágil Iconix adaptado para desarrollar software, siguiendo los principios y valores del manifiesto ágil, la solución al problema se enfocará mediante la aplicación de técnicas probadas y aceptadas de ingeniería de software como son: ingeniería de requisitos, patrones de diseño y técnicas para programación orientada a objetos, ésta solución proveerá un método que apoye la producción de software para micro, pequeñas y medianas empresas, con procesos alineado a la evaluación del grado de calidad del producto software, usando la norma técnica peruana NTP-ISO/IEC 9126 y sus extensiones. La industria peruana de software tendrá la oportunidad de producir software de calidad con la metodología adaptada, las empresas que brindan servicios de desarrollo y mantenimiento tendrán el sello de garantía de calidad del producto software, la comunidad académica podrá contar con una guía de aprendizaje, donde se observa la integración del desarrollo de software con la evaluación de calidad del producto software.

1.5. ALCANCES Y LIMITACIONES DE LA INVESTIGACIÓN

1.5.1. Alcances

El estudio se realiza en el contexto de las metodologías agiles para desarrollo de software, los principios y valores de agilidad, aseguramiento de calidad del producto software, aplicación de las NTP ISO/IEC 9126 y sus extensiones, ingeniería de requisitos, patrones de diseño de software, técnicas y principios de programación orientada a objetos

para producir código limpio. La metodología ágil Iconix propuesta será formulada para desarrollar y mantener software con equipos de hasta 20 profesionales.

1.5.2. Limitaciones

En el Perú existen escasos estudios sobre la adaptación de procesos para desarrollo de software con agilidad, orientado a proyectos pequeños y medianos, aplicando metodologías ágiles y que durante el desarrollo sea evaluado la calidad de los productos software, razón por la que se referencia mayoritariamente bibliografía Internacional. El límite del estudio es de contexto para metodologías agiles del desarrollo de software, el financiamiento de la investigación es personal, existe limitaciones para levantar información sobre la intervención durante la encuesta a los expertos de metodologías agiles que son muy escasos en el Perú y se ha tenido que recurrir a expertos extranjeros.

1.6. OBJETIVOS DE LA INVESTIGACIÓN

1.6.1. Objetivo General

Incluir técnicas de ingeniería de software a la metodología ágil Iconix mediante técnicas e instrumentos con la finalidad de mejorar la calidad del producto software, Iconix, Lima, 2017.

1.6.2. Objetivos Específicos

a. Desarrollar la fase de análisis de requisitos mediante la inclusión de ingeniería de requisitos a fin de mejorar la calidad del producto software.
b. Desarrollar la fase de diseño mediante la inclusión de patrones de diseño a fin de mejorar la calidad del producto software.
c. Desarrollar la fase de implementación mediante la inclusión de técnicas de programación orientada a objetos con la finalidad de mejorar la calidad del producto software.

1.7. HIPÓTESIS DE LA INVESTIGACIÓN

1.7.1. Hipótesis General

Si adaptamos la metodología ágil Iconix con la inclusión de técnicas de ingeniería de software, entonces se mejora la calidad del producto software, Lima, 2017.

1.7.2. Hipótesis Específicas

a. Si modificamos la fase de análisis de requisitos mediante la inclusión de ingeniería de requisitos, entonces se mejora la calidad del producto software.

b. Si variamos la fase de diseño mediante la inclusión de patrones de diseño, entonces se mejora la calidad del producto software.

c. Si transformamos la fase de implementación mediante la inclusión de técnicas de programación orientada a objetos, entonces se mejora la calidad del producto software.

II. MARCO TEÓRICO

2.1. TEORÍAS GENERALES

El Banco Mundial (2018), establece el objetivo 17 de desarrollo sostenible: "Fortalecer los medios de implementación y revitalizar la Alianza Mundial para el Desarrollo Sostenible", siendo el objetivo específico 17.8 "Poner en pleno funcionamiento, a más tardar en 2017, el banco de tecnología y el mecanismo de apoyo a la creación de capacidad en materia de ciencia, tecnología e innovación para los países menos adelantados y aumentar la utilización de tecnologías instrumentales, en particular la tecnología de la información y las comunicaciones". En este contexto, se propone el estudio como una forma mejorada para desarrollar software con la metodología ágil Iconix, orientada a aplicaciones web y móviles que se ejecutan por internet.

Soete, Schneegans, Eröcal, Angathevar y Rasiah (2015), en el informe UNESCO sobre la ciencia hacia el 2030, "Cada vez más países se enfrentan a una serie de dilemas comunes, tales como la dificultad de encontrar un equilibrio entre la participación local e internacional en investigación, o entre la ciencia básica y la aplicada, la generación de nuevos conocimientos y de conocimientos comercializables, o la oposición entre ciencia para el bien común y ciencia para impulsar el comercio". La afirmación nos induce a pensar que que la ciencia debe estar orientada al desarrollo de las naciones y la población mundial, particularmente la ciencia aplicada a la generación de nuevas tecnologías de información para mejorar la calidad de vida de los habitantes con respeto por la naturaleza.

La UNESCO a. (2017) en la asamblea general del año 2015, en el examen decenal de la Cumbre Mundial sobre la Sociedad de la Información (CMSI), reconoce mediante la resolución 70/125, que había que aprovechar el potencial de las tecnologías de informacion y comunicaciones (TIC), a fin de cumplir la agenda 2030 para el desarrollo sostenible, observando que las TIC pueden acelerar el progreso en relación con los 17 Objetivos de Desarrollo Sostenible (ODS). La contribución de la UNESCO en la CMSI a lo largo de estos 10 años, reconoce que pasar de las "sociedades de la información" a las "sociedades del conocimiento", la información no solo se crea y difunde, sino que, se pone al servicio del desarrollo humano.

En febrero de 2001 con la participan de 17 expertos de la industria del software, celebrada

en Utah-EEUU, surge el manifiesto para “Desarrollo Ágil de Software”, y se crea “Agile Alliance”, organización sin fines de lucro, dedicada a promover los conceptos para el desarrollo ágil de software. El Manifiesto resume la filosofía “ágil”, que considera valores y principios (Agile Alliance, 2001), como se describe a continuación.

Valores para Desarrollo Ágil de Software

Según Agile Alliance (2001), los valores son:

a. Al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas. Las personas son el principal factor de éxito de un proyecto software. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte automáticamente. Es mejor crear el equipo y que éste configure su propio entorno de desarrollo en base a sus necesidades.
b. Desarrollar software que funciona más que conseguir una buena documentación. La regla a seguir es “no producir documentos a menos que sean necesarios de forma inmediata para tomar un decisión importante”. Estos documentos deben ser cortos y centrarse en lo fundamental.
c. La colaboración con el cliente más que la negociación de un contrato. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.
d. Responder a los cambios más que seguir estrictamente un plan. La habilidad de responder a los cambios que puedan surgir a los largo del proyecto (en los requisitos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

Principios para Desarrollo Ágil de Software

Según Agile Alliance (2001), los principios son:

i. La prioridad es satisfacer al cliente mediante entregas tempranas y continuas de software que le aporte un valor.
ii. Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.

iii. Entregar frecuentemente software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.
iv. Las personas del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.
v. Construir el proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.
vi. El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.
vii. El software que funciona es la medida principal de progreso.
viii. Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz permanente.
ix. La atención continua a la calidad técnica y al buen diseño mejora la agilidad
x. La simplicidad es esencial.
xi. Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.
xii. En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más efectivo, y según esto ajusta su comportamiento.

"El software se compone de programas, datos y documentos. Cada uno de estos elementos compone una configuración que se crea como parte del proceso de la ingeniería del software." (Pressman, 2002, p. 10).

"La ingeniería del software es una disciplina de la ingeniería que comprende todos los aspectos de la producción de software desde las etapas iniciales de la especificación del sistema, hasta el mantenimiento de éste después de que se utiliza. En esta definición, existen dos frases clave: disciplina de la ingeniería y todos los aspectos de producción de software." (Sommerville, 2005, p. 6).

El ciclo de vida del software, es una sucesión de fases por las que atraviesa un producto software a lo largo de su desarrollo y existencia. Cada fase está compuesta por un conjunto de actividades que son ejecutadas. De acuerdo a las clasificaciones de Schwartz (1975), Pressman (2002) y Sommerville (2005), podemos aproximar las siguientes fases: especificación, diseño, implementación, validación, mantenimiento y evolución. Existen

subproductos que son generados en cada fase, ejemplo, en la fase de especificación, se ha desarrollado y entregado uno o más documentos que detallan los requisitos del sistema, estos subproductos son denominados entregables, artefactos o productos software.

El desarrollo ágil de software inicia con una corriente para enfrentar los problemas que se presentan durante la aplicación de las metodologías de desarrollo de software tradicionales, en contraposición, las metodologías ágiles enfrentan lo impredecible apoyándose en el equipo y su creatividad en lugar de los procesos (Cockburn y Highsmith, 2001). Se caracterizan por: presentar ciclos iterativos de desarrollos cortos, inducidos por las particularidades del producto, períodos de abstracción y observación, desarrollo colaborativo, consenso rápido para la retroalimentación, e integración continua de los cambios del código fuente en el desarrollo de software (Nerur, et. al, 2005).

El modelo de calidad para mantenibilidad definidas en la ISO/IEC 25010, presenta cinco subcaracterísticas que son: analizabilidad, modularidad, capacidad de ser modificado, capacidad de ser reutilizado, capacidad de ser probado. Según Suarez y Garzas (2014) proponen centrar el modelo de calidad para mantenibilidad, de acuerdo a la ISO/IEC 25010, por las siguientes razones: (a) El mantenimiento supone una fase del ciclo de vida de desarrollo más costosa, llegando a alcanzar el 60%. (b) La mantenibilidad es una de las características más demandadas hoy en día por los clientes de software, que piden que el producto software que se desarrolle pueda ser después mantenido por ellos mismos o incluso por un tercero. (c) Las tareas de mantenimiento sobre productos con poca mantenibilidad tienen más probabilidad de introducir nuevos errores en el producto.

2.2. MARCO CONCEPTUAL

2.2.1. Metodología Ágil Iconix

Rosenberg y Scott (1999), Rosenberg y Scott (2001), Rosenberg et al. (2005) y Rosenberg y Stephens (2007), consideran que Iconix es una metodología ágil, con principios de desarrollo incremental e iterativo, que presenta las fases de: análisis de requisitos, diseño preliminar, diseño detallado, implementación y pruebas, generan un conjunto de productos software en cada fase modelado con UML, presenta una parte dinámica y otra estática, el modelo estático se incrementa y es refinado por el modelo dinámico.

En la figura 1, observamos los principios de la concepción del Iconix, donde: OMT de Rumbaugh aporta su modelo de objetos de dominio del problema, Objetory de Jacobson el modelo de dominio de solución dirigida por el usuario y Booch con sus modelos a nivel de diseño detallado. La metodología consiste en producir un conjunto de productos software, con parte dinámica y estática del sistema, desarrollados incrementalmente y en paralelo, el modelo estático se incrementa y es refinado por el modelo dinámico.

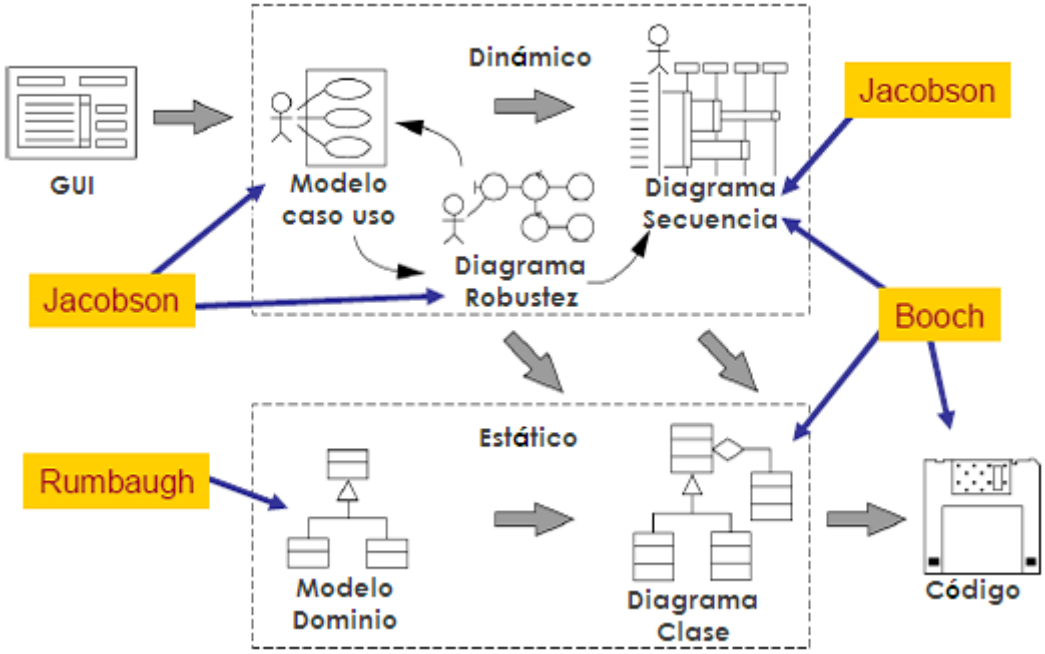


Figura 1. Esquema de la metodología agil Iconix (Rosenberg y Scott, 2001).

Características fundamentales de la metodología Iconix:

a. El proceso es conducido por casos de uso;
b. Relativamente pequeño y simple, tal como la programación extrema (XP), pero sin eliminar el análisis y diseño que XP no contempla;
c. Uso simplificado del UML (OMG®, 2001) mediante: modelo de dominio, diagrama de casos de uso, diagrama de robustez, diagrama de secuencia y diagrama de clases;
d. Alto grado de trazabilidad, para seguir la relación entre los productos software producido y los requisitos.

Rosenberg y Scott (1999), Rosenberg y Scott (2001), Rosenberg et al. (2005) y Rosenberg y Stephens (2007), presentan las fases: análisis de requisitos, diseño preliminar, diseño detallado, implementación, pruebas y revisiones, que se detallan líneas abajo.

Rosenberg y Scott (1999), Rosenberg y Scott (2001), Rosenberg et al. (2005) y Rosenberg y Stephens (2007), opinan que: Iconix está considerado como proceso puro, práctico y simple, con el componente de análisis y representación del problema sólido y eficaz, Iconix es un proceso no tan burocrático como el proceso unificado de rational (RUP), no genera tanta documentación, es un proceso simple como XP, no deja de hacer el análisis y diseño, se destaca como un poderoso proceso de análisis de software, este proceso hace uso del lenguaje de modelado (UML) con característica de "rastreabilidad de los requisitos".

2.2.1.1. Análisis de Requisitos

En la fase de análisis de requisitos de la metodología ágil Iconix, se realizan tareas y se generan productos software (Rosenberg y Stephens, 2007) y (Rosenberg y Scott, 2001). Como: definir lo que el sistema debe hacer mediante los requisitos funcionales y no funcionales, identificar los objetos del mundo real y construir el modelo de dominio, realizar prototipos de interfaz gráfica de usuario, identificar los casos de uso del sistema y su diagrama, presentar los casos de uso mediante paquetes, asignar los requisitos funcionales a los casos de uso y describir el primer borrador de casos de uso.

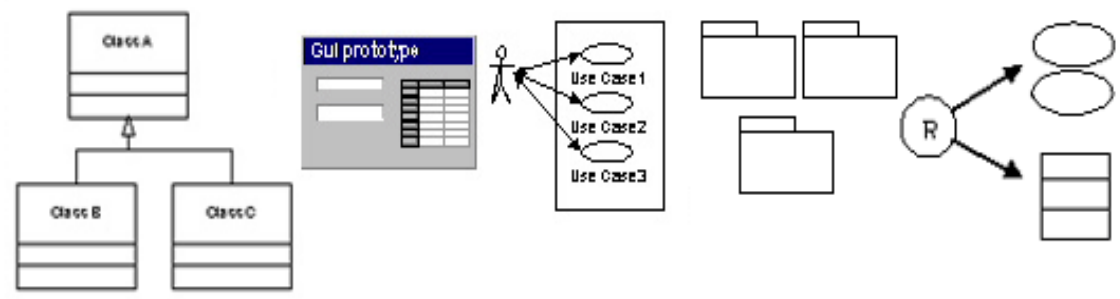


Figura 2. Tareas del análisis de requisitos (Rosenberg y Scott, 2001).

a. Definir lo que el sistema debe ser capaz de hacer identificando los requisitos funcionales y no funcionales;
b. Identificar los objetos del "mundo real" y sus relaciones de agregación y generalización, presentando en el diagrama de clase de alto nivel, modelo de dominio;
c. Realizar un prototipo rápido de interfase hombre - máquina (GUI), para que el cliente pueda comprender mejor el sistema en desarrollo;
d. Identificar los casos de uso del sistema, mostrando los actores involucrados,

presentando en el diagrama de casos de uso;

e. Organizar los casos de uso lógicamente en grupos, presentando en los diagramas de paquetes;
f. Asignar los requisitos funcionales a los casos de uso y objetos de dominio;
g. Escribir el primer borrador de casos de uso.

Modelado de Dominio

Rosenberg y Scott (1999), Rosenberg y Stephens (2007), definen el modelado de dominio como, la tarea de descubrir objetos (clases), que representan cosas y conceptos del "mundo real". El modelo de dominio para un proyecto define el alcance y la base para construir los casos de uso. El modelo de dominio sirve como un glosario de términos, que es utilizado al inicio para escribir los casos de uso. El objetivo del modelado de dominio es realizar un primer levantamiento de las clases (objetos) que forman parte del problema.

De acuerdo a Rosenberg y Stephens (2007), el enfoque de Iconix asume que el modelo de dominio inicial es incorrecto y proporciona un mejoramiento incremental a medida que se analizan los casos de uso.

Rosenberg y Scott (1999) definen una clase como "una descripción de un conjunto de objetos con propiedades similares, comportamiento común, relaciones comunes y semántica común".

Rosenberg y Stephens (2007), alcanzan diez consejos para el modelado de dominio, presentado en orden a su importancia, siendo:

1. Enfocarse en los objetos del mundo real (dominio del problema);
2. Usar las relaciones de generalización (es-un) y agregación (tiene-un) para mostrar las relaciones de los objetos entre sí;
3. Limitar sus esfuerzos para el modelado de dominio inicial a dos horas;
4. Organizar sus clases con abstracciones clave en el dominio del problema;
5. No confundir el modelo de dominio con el modelo de datos;
6. No confundir un objeto con una tabla de la base de datos;
7. Usar el modelo de dominio como un glosario del proyecto;
8. Hacer su modelo de dominio inicial antes de escribir sus casos de uso, para evitar ambigüedad de nombres;

9. No espere que su diagrama de clases final coincida con su modelo de dominio, pero recuerde que debe existir cierto parecido entre ellos;
10. No ponga clases interface y control en su modelo de dominio.

Los errores más frecuentes del modelado de dominio, destacados por Rosenberg y Scott (1999), son:

1. Asignar multiplicidad a las asociaciones de clases al inicio del modelado;
2. Analizar los verbos y sustantivos exhaustivamente;
3. Asignar operaciones a las clases sin explorar los casos de uso y diagramas de secuencia;
4. Debatir sobre usar agregación o composición para cada asociación;
5. Presumir una estrategia de implementación específica en esta actividad;
6. Usar nombres difíciles de entender para las clases;
7. Hacer directamente la implementación;
8. Crear un esquema de "uno-a-uno" entre las clases del modelo de dominio y las tablas de la base de datos;
9. Utilizar patrones de diseño de forma prematura.

Modelado de Casos de Uso

El modelo de casos de uso es el concepto central del desarrollo, porque guía toda la metodología ágil Iconix. Así, Rosenberg y Scott (1999) muestran elementos clave, donde; los casos de uso son desarrollados a partir del modelo de dominio, el análisis de robustez identifica un conjunto de objetos que satisfacen cada caso de uso, el diagrama de secuencia traza el flujo de los mensajes entre los objetos conforme se especificó en los casos de uso, los requisitos de usuario son asignados a los casos de uso y clases, los casos de uso se usan para las pruebas durante la implementación.

Un caso de uso describe la interacción entre el usuario y el sistema para alcanzar un objetivo (Rosenberg y Stephens, 2007). Un caso de uso describe y valida lo que el sistema debe hacer, sirve para el control entre el usuario, cliente y desarrolladores.

"El caso de uso es un documento narrativo que describe la secuencia de eventos de un actor (agente externo) que utiliza un sistema para completar un proceso (Larman, 2000; ápud Jacobson, 1992). Los casos de uso son historias o casos de utilización de un sistema,

no son exactamente los requerimientos ni las especificaciones funcionales, sino que, ejemplifican e incluyen tácitamente los requerimientos en las historias que narran". (Larman, 2000).

Es importante indicar que UML (OMG®, 2001) no define un estándar para describir casos de uso. Esta actividad debe ser definida por el flujo de trabajo de la metodología seleccionada.

Los actores representan el rol de una entidad externa al sistema, como: un usuario, el hardware, otro sistema que interactúa con el sistema modelado. Los actores no forman parte del sistema, pero identifican sus límites. Fowler y Scott (2000) afirman que, cuando hablamos de actores, es importante pensar en los roles y no en las personas o en cargos. Un único actor puede desempeñar varios casos de uso, y para un caso de uso puede haber muchos actores.

Un paquete en UML (OMG®, 2001), es un elemento organizacional. Permite agregar diferentes elementos de un sistema en grupos, de forma que semánticamente o estructuralmente tenga sentido. Entonces, un grupo de casos de uso relacionados será definido como paquete de casos de uso.

Rosenberg y Stephens (2007), presentan diez consejos para el modelado de casos de uso, los ítems de 1 a 6 describen el uso del sistema y de 7 a 10 para describir los casos de uso en el contexto del modelo de dominio, como sigue:

1. Siga la regla de dos párrafos (curso básico y curso alterno);
2. Organizar sus casos de uso con actores y diagramas de caso de uso;
3. Escriba sus casos de uso en voz activa (las oraciones activas dejan en claro quién hace qué y, escriba desde la perspectiva del usuario);
4. Escriba sus casos de uso utilizando un flujo de evento/respuesta (cómo el usuario utiliza el sistema y que responde el sistema), describiendo ambos lados del dialogo usuario/sistema;
5. Use prototipos GUI e historia de eventos del usuario;
6. Recordar que sus casos de uso son especificaciones del comportamiento del sistema en tiempo de ejecución;
7. Escriba sus casos de uso en el contexto del modelo de dominio (objetos);

8. Escriba sus casos de uso utilizando una estructura tipo “sustantivo-verbo-sustantivo”;
9. Haga referencia a los objetos de dominio por su nombre;
10. Haga referencia a las clases interfaz por su nombre.

Diez errores que deben evitarse cuando escribe los casos de uso, destacados por Rosenberg y Scott (1999), son:

1. Escribir requisitos funcionales en lugar del escenario de uso;
2. Describir atributos o métodos en lugar de uso;
3. Escribir los casos de uso de forma muy simplificada;
4. Desvincular la descripción de la interface del usuario;
5. Evitar nombres explícitos para los objetos interfaz;
6. Escribir en voz pasiva (ejemplo, el verbo “ser/estar" seguido de un verbo en tiempo pasado, es señal clara de una oración pasiva);
7. Describir solo interacciones de usuario, ignorando respuestas del sistema;
8. Omitir la descripción del curso alterno;
9. Describir algo que no es parte del caso de uso, como; precondiciones o post-condiciones;
10. Perder tiempo decidiendo si usar includes o extends.

No generalizar casos de uso, porque es algo como "generalizar el manual del usuario", seguro dirá ¿Qué cosa? (Rosenberg y Stephens, 2007).

Tabla 10. Relaciones comunes de casos de uso

Relación	Descripción	Solución
A <include> B	A medio camino del caso de uso A, se llama al caso de uso B. Cuando B termina, A se lleva a cabo desde que B se detiene. ▸ Es como decir "A tiene un B”.	Bala de plata
A <extend> B	Todos los pasos del caso de uso A se realizan durante la ejecución del caso de uso B, en el punto de extensión que se especifica en B. ▸ En la mayoría de los casos <extends> es	Estaca en el corazón

	<includes> con una flecha hacia atrás. (Ambos son subtipos de <invokes>).	
A <precedes> B	El caso de uso A debe llevarse a cabo en su totalidad antes de comenzar el caso de uso B	Agua bendita
A <invokes> B	El caso de uso B sucede durante la vida útil del caso de uso A.	Una buena taza de té y un huevo de pascua.

Fuente: (Rosenberg y Stephens, 2007)

Tabla 11. Casos de uso versus algoritmos

Casos de Uso	**Algoritmo**
Diálogo entre usuario y sistema	Computación "atómica"
Secuencia de evento/respuesta	Serie de pasos
Cursos básico/alterno	Un paso de un caso de uso
Múltiples objetos participantes	Operación en una clase
El usuario y sistema	Todo el sistema

Fuente: (Rosenberg y Stephens, 2007)

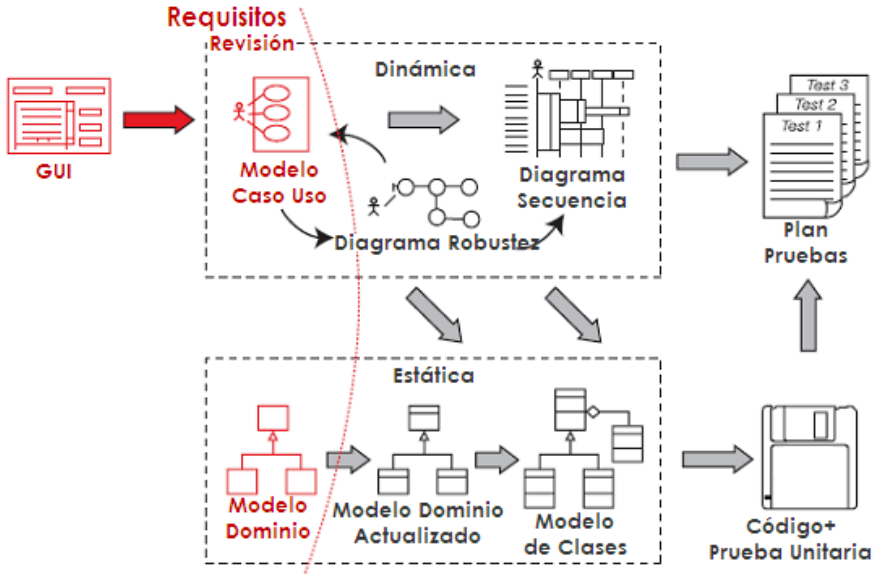


Figura 3. Esquema para revisión de análisis de requisitos (Rosenberg y Stephens, 2007).

Diez consejos para la revisión de requisitos según Rosenberg y Stephens (2007), que mostramos a continuación:

1. Asegurar que el modelo de dominio contiene al menos 80% de objetos más importantes del dominio de problema (objetos del mundo real);
2. Asegurar que el modelo de dominio muestra relaciones de generalización y agregación entre los objetos de dominio;
3. Asegurar que los casos de uso describen el curso básico y curso alterno, en voz activa;
4. Asegúrese que la voz pasiva y los requisitos funcionales, no están en la descripción de los caso de uso;
5. Asegúrese que ha organizado los casos de uso en paquetes y cada paquete tiene un diagrama de caso de uso;
6. Asegúrese que sus casos de uso están escritos en el contexto del modelo de dominio;
7. Los casos de uso deben estar escritos en contexto de interfaz de usuario;
8. Asegúrese que ha completado la descripción del caso de uso con alguna historia o prototipo GUI;
9. Revisar los casos de uso, modelo de dominio y prototipos GUI con el personal indicado para esta actividad;
10. Revisar los casos de uso mediante los "ocho pasos sencillos para un mejor caso de uso" como se indica:
    a. Eliminar todo lo que este fuera del ámbito de aplicación;
    b. Cambiar la descripción de voz pasiva a voz activa;
    c. Compruebe que la descripción no es demasiado abstracta;
    d. Presentar con precisión la GUI relacionada;
    e. Nombre los objetos de dominio que participan;
    f. Asegúrese que tiene todos los cursos alternos;
    g. Asocie cada requisito a los casos de uso;
    h. Describir lo que el usuario intenta hacer para cada caso de uso.

2.2.1.2. Diseño

En la fase de diseño de la metodología ágil Iconix, se realiza el diseño preliminar, introducido por Ivar Jacobson en 1991; el diseño se realiza cuando se concluyó el diseño preliminar y la descripción de los casos de uso es completa, correcta, detallada y explícita, se identifica los objetos: interfaz, entidad y control (Rosenberg y Scott, 1999), el diseño está representado por el diagrama de secuencia, que permite asignar comportamiento al

modelo de clases con operaciones, atributos y navegabilidad de las clases, asimismo, muestra la colaboración dinámica entre objetos del sistema; interfaz, controlador y entidad (Rosenberg y Stephens, 2007).

Análisis de Robustez

El concepto de análisis de robustez fue introducido por Ivar Jacobson para el mundo de la orientación a objetos en 1991. El análisis de robustez es, analizar los casos de uso e identificar un primer conjunto de objetos para cada caso de uso (Rosenberg y Scott, 1999), observamos el esquema en la figura 6, los objetos son clasificados en tres estereotipos:

a. Objeto interfaz.- capa de presentación que los actores usan para interactuar con el sistema (pantallas o páginas Web), objeto en el caso de uso que debe tener un nombre (sustantivo);
b. Objeto entidad.- son objetos del modelo de dominio, objeto en el caso de uso que debe tener un nombre (sustantivo);
c. Objeto control (controlador).- funcionan como "pegamento" entre los objetos interfaz y los objetos entidad, éste objeto en el caso de uso debe ser un verbo. Un controlador en un diagrama de robustez no siempre es una clase control real, puede ser contenedor de una función de software y, son convertidos en métodos de los objetos entidad o interfaz.

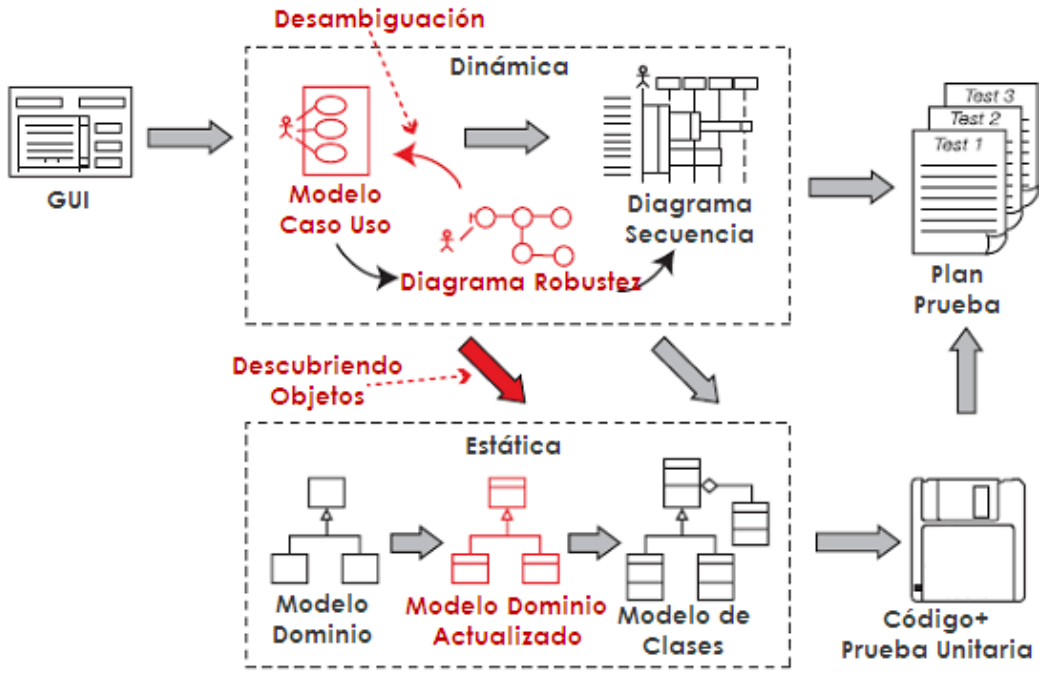


Figura 4. Esquema del análisis de robustez (Rosenberg y Stephens, 2007).

El diagrama de robustez es una “imagen” de un caso de uso, el diagrama de robustez y la descripción del caso de uso deben coincidir con precisión. Si el análisis (casos de uso) es el "Qué" y diseño el "Cómo", entonces el análisis de robustez es un diseño preliminar. Por lo tanto, es parte del análisis y del diseño, ver la figura 4.

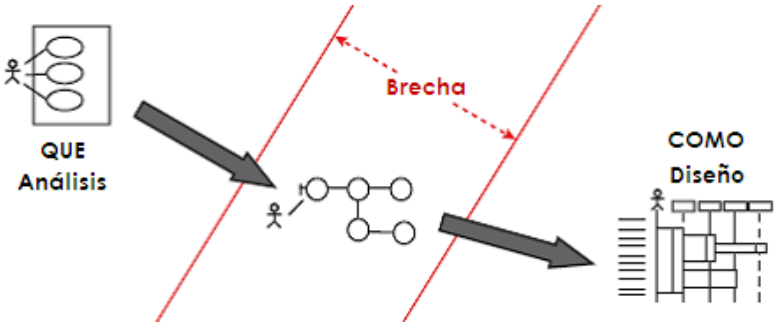


Figura 5. Análisis de robustez entre análisis y diseño (Rosenberg y Stephens, 2007).

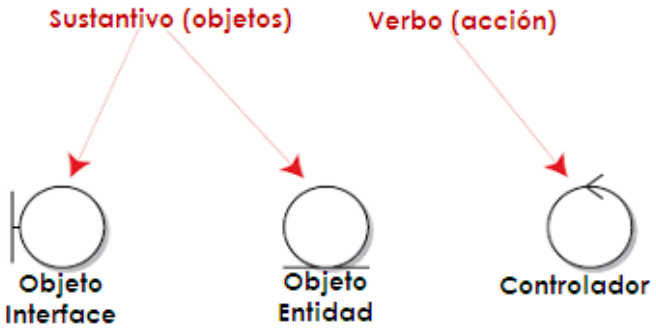


Figura 6. Símbolos del diagrama de robustez (Rosenberg y Stephens, 2007).

Las reglas que ayudan a hacer cumplir el formato nombre-verbo-nombre en la descripción del caso de uso, para construir los diagramas de robustez, son:

a. Los nombres (objetos entidad e interfaz) pueden comunicarse con los verbos (y viceversa);
b. Los nombres no pueden comunicarse con otros nombres;
c. Los verbos (controladores) pueden comunicarse con otros verbos.

En el diagrama de robustez, se describe los elementos del GUI usando los objetos interfaz, las funciones de software usando los controladores y, los objetos de dominio usando las entidades.

Diez normas para construir el diagrama de robustez según Rosenberg y Stephens (2007),

indicamos a continuación:

1. Pegar la descripción del caso de uso en la interface para construir el diagrama de robustez;
2. Usar las clases entidad del modelo de dominio, para actualizar el modelo de dominio al descubrir clases durante el análisis de robustez y construir el diagrama de robustez;
3. Rescribir los casos de uso mientras hace el análisis de robustez (técnica de desambiguación);
4. Crear un objeto interfaz para cada pantalla y nombrar esas pantallas sin ambigüedad;
5. Recordar que los controladores a veces no son objetos control real, normalmente son funciones de software lógicas.
6. No preocuparse sobre la dirección de las flechas en un diagrama de robustez.
7. Relacionar un caso de uso a un diagrama de robustez, solo si este es invocado de otro caso de uso;
8. El diagrama de robustez representa el diseño conceptual (diseño preliminar), el diseño real es mostrado en el diagrama de secuencia;
9. Las clases interfaz y entidad de un diagrama de robustez, se convertirán en objetos en un diagrama de secuencia, mientras que los controladores se convertirán en mensajes;
10. Un diagrama de robustez es una "imagen" de un caso de uso, cuyo fin es refinar la descripción del caso de uso y modelo de dominio (actualizado).

Los diez beneficios del análisis de robustez, presentados por Rosenberg y Scott (1999), son:

1. Obliga la escritura de los casos de uso de forma correcta y concreta;
2. Fuerza la escritura de los casos de caso en voz activa;
3. Provee mecanismos de verificación de los casos de uso;
4. Ayuda a definir reglas de sintaxis del tipo “el actor interactúa solo con objetos interfaz”, para los casos de uso;
5. Construir un diagrama de robustez es más fácil y más rápido, que un diagrama de secuencia;
6. Permite esbozar un framework para GUI-lógica-datos para sistemas

cliente/servidor;

7. Permite relacionar lo que hará el sistema (casos de uso) y cómo funcionará el sistema (diagrama de secuencia);
8. Llena el vacío entre el análisis (caso de uso) y diseño (diagrama de secuencia);
9. Permite identificar el reúso de estructuras definidas en la realización de los casos de uso;
10. Facilita el uso del patrón, modelo - vista - control.

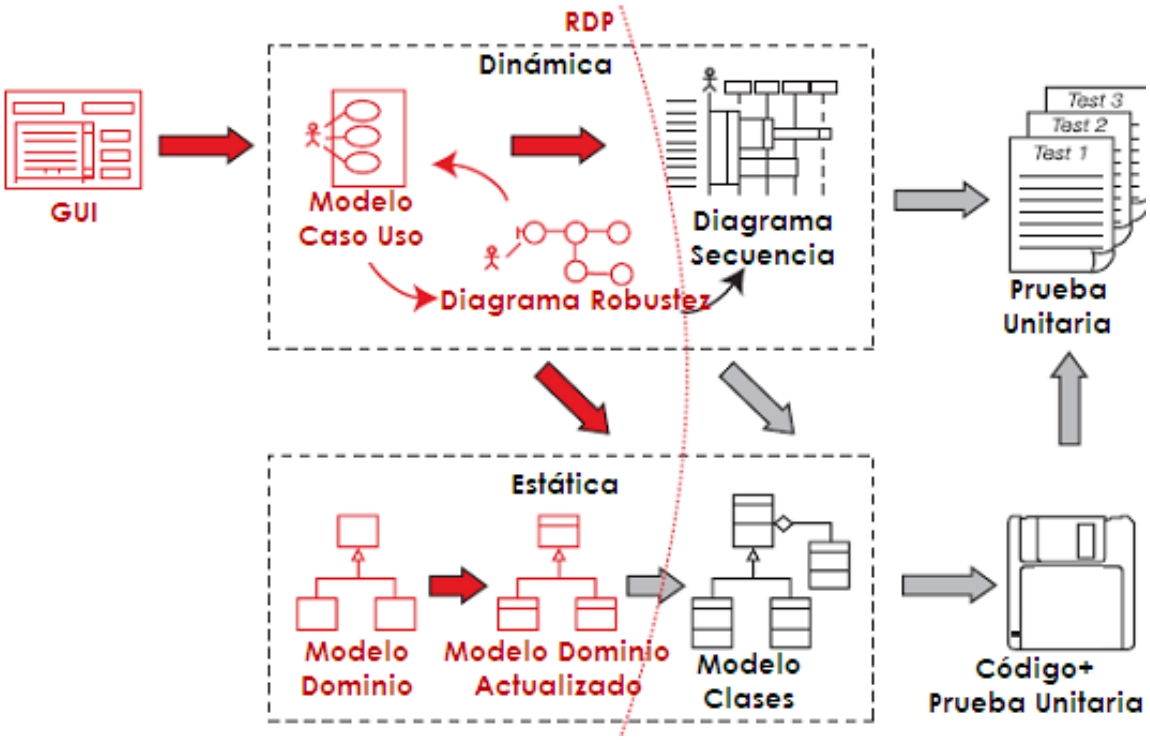


Figura 7. Esquema para revisión de diseño preliminar (Rosenberg y Stephens, 2007).

Revisión de Diseño Preliminar

Según Rosenberg y Stephens (2007), antes de pasar a la revisión del diseño preliminar, hay que actualizar el diagrama de clases continuamente, mientras se trabaja con los casos de uso y el análisis de robustez.

Para asegurarse que los diagramas de robustez, el modelo de dominio y, la descripción de casos de uso coincidan entre sí. Esta revisión es el "puente" entre el diseño preliminar y el diseño detallado, se hace para cada paquete de casos de uso. Las personas involucradas en esta actividad son, el mismo grupo de la revisión de requisitos.

Las reglas que permiten hacer una buena revisión del diseño preliminar, son principios en resumen, como una lista de consejos (Rosenberg y Scott, 2001).

1. Para cada caso de uso asegúrese que su descripción coincide con el diagrama de robustez, utilizar la prueba del resaltador;
2. Asegúrese que todas las clases entidad de todos los diagramas de robustez aparecen en el modelo de dominio actualizado;
3. Asegúrese que puede trazar flujos de datos entre clases entidad y clases interfase;
4. No olvide escribir los cursos alternos y su comportamiento, para cada caso de uso;
5. Asegúrese que la descripción de cada caso de uso, cubre ambos lados del diálogo entre usuario y sistema;
6. Use correctamente las reglas de sintaxis de comunicación entre objetos para construir el diagrama de robustez;
7. Asegúrese que esta revisión incluye a personal no-técnico (clientes, equipo de marketing, etc.) y técnico (arquitectos de sistema, programadores, etc.);
8. Asegúrese que los casos de uso están en el contexto del modelo de objetos y en el contexto de la interfaz gráfica;
9. Asegúrese que el diagrama de robustez y su descripción del caso de uso, no están mostrando el nivel de detalle del diagrama de secuencia;
10. Siga los "seis pasos sencillos" para el mejor diseño preliminar, pasos ampliados en Rosenberg y Scott (2001), aplicar a los diagramas de robustez y descripción de los casos de uso, para cada diagrama de robustez, hacer lo siguiente:
    a. El diagrama debe coincidir con la descripción del caso de uso;
    b. El diagrama debe cumplir las reglas del análisis de robustez;
    c. Comprobar que el diagrama sigue el flujo lógico del caso de uso;
    d. El diagrama debe tener todos los cursos alternos para el caso de uso;
    e. Tener cuidado con los "diseños de patrones " en el diagrama;
    f. Verificar que el diagrama no es un diseño detallado.

Diagrama de Secuencia

Finalizado el análisis de robustez y la revisión de diseño preliminar, podemos comenzar el diseño detallado (asignación de comportamiento). La descripción de los casos de uso debe

ser completa, correcta, detallada y explícita (casos de uso para crear un diseño detallado), tener descubiertas casi todas las clases de dominio y, contar con la arquitectura técnica.

El comportamiento de un caso de uso se detalla mediante el diagrama de secuencia, éste muestra la colaboración dinámica entre objetos del sistema, también observamos la secuencia de mensajes enviados entre los objetos (interacción entre los objetos). El diagrama de secuencia, ayuda a asignar operaciones a los objetos interfaz y entidad, porque los controladores (verbos) del diagrama de robustez, se convierten en mensajes (operaciones) entre los objetos interfaz y entidad (nombres). En conclusión, actualizamos el modelo estático con las operaciones y los atributos identificados en las fases anteriores.

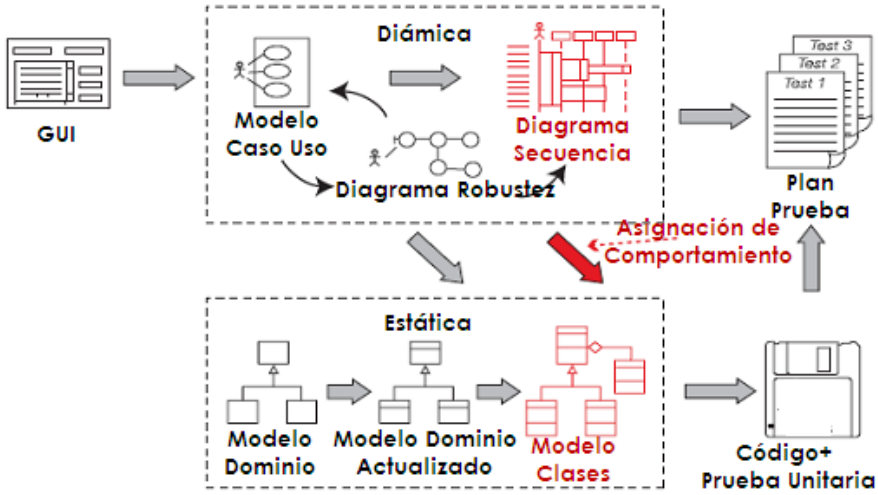


Figura 8. Desarrollo del diagrama de secuencia (Rosenberg y Stephens, 2007).

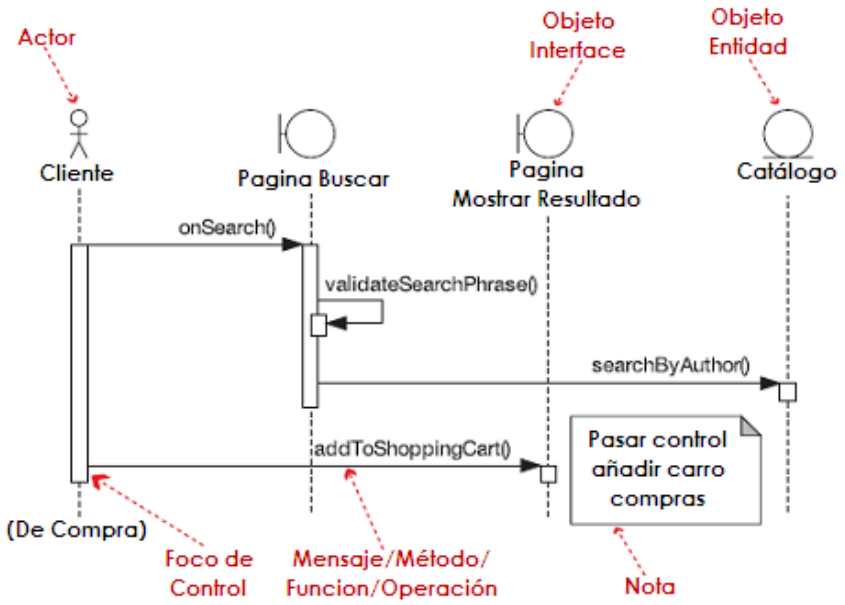


Figura 9. Elementos del diagrama de secuencia (Rosenberg y Stephens, 2007).

Según Rosenberg y Scott (1999) pasos construir el diagrama de secuencia, son:

1. Copiar texto del caso de uso al margen izquierdo;
2. Objetos del diagrama de robustez, presentar el nombre del objeto, opcionalmente el nombre de la clase (objeto::clase);
3. Mensajes, representados como flechas entre los objetos;
4. Métodos, son la implementación de las operaciones.

Decidir que mensajes y clases quedan, es la esencia del modelado de secuencia, no es tarea fácil y exige mucho esfuerzo y experiencia (Rosenberg y Scott, 1999). Iconix presenta algunas sugerencias para esta tarea:

a. Comenzar convirtiendo los controladores del diagrama de robustez, como mensajes que representan el comportamiento deseado;
b. Usar el diagrama de robustez como una lista de chequeo, para estar seguros que todo el comportamiento necesario del sistema está en el diagrama de secuencia;
c. Responder a la pregunta "qué objetos son responsables de que funciones?";
d. Diseñar mensajes entre objetos es equivalente a atribuir operaciones para las clases.

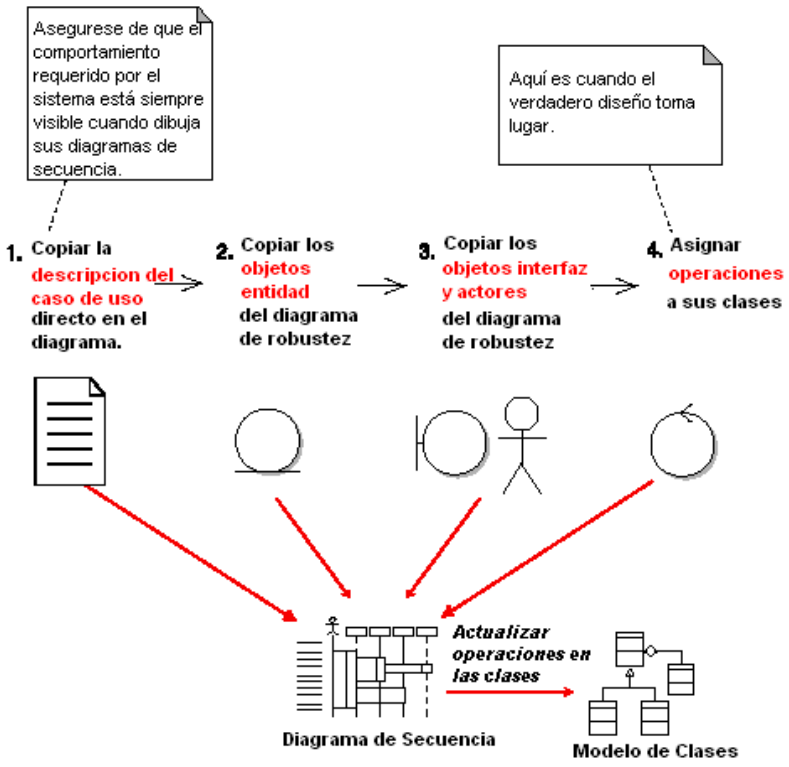


Figura 10. Cuatro pasos para construir el diagrama de secuencia.

Las diez reglas más importantes para construir el diagrama de secuencia, sugeridos por Rosenberg y Stephens (2007), son:

1. Comprender porque se está dibujando un diagrama de secuencia;
2. Dibujar un diagrama de secuencia para cada caso de uso, incorpore todo el curso básico y los cursos alternos;
3. Comenzar el diagrama de secuencia con las clases interfaz, entidad, actores y, la descripción de los casos de uso del análisis de robustez;
4. Usar el diagrama de secuencia para mostrar cómo el comportamiento del caso de uso (controlador diagrama de robustez) se realiza por los objetos;
5. Asegúrese que la descripción de los casos de uso tienen relación con los mensajes del diagrama de secuencia;
6. No perder mucho tiempo preocupándose por la línea de vida del objeto;
7. Asignar operaciones a las clases mientras dibuja los mensajes;
8. Revisar el diagrama de clases mientras asigna sus operaciones, para estar seguro que las operaciones están en las clases que le corresponden;
9. Hacer refactoring al diseño en el diagrama de secuencia antes de codificar;
10. Depurar el modelo estático antes de la revisión crítica de diseño, para; solucionar los problemas del diseño del mundo real, identificar los patrones par el diseño.

Revisión Crítica de Diseño

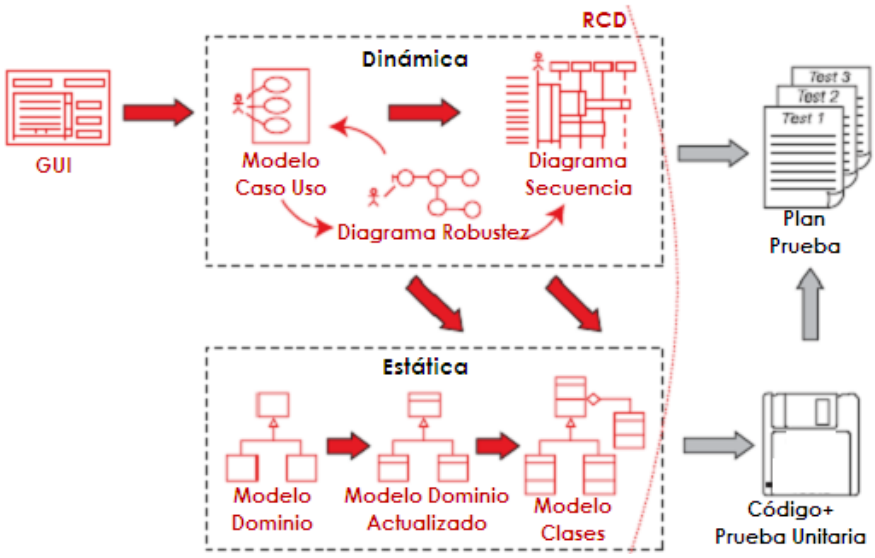


Figura 11. Esquema para revisión crítica de diseño (Rosenberg y Stephens, 2007).

Para Rosenberg y Stephens (2007), existen diez elementos clave para la revisión crítica de diseño, que son:

1. Verificar que los diagramas de secuencia coincidan con la descripción de los casos de uso;
2. Verificar (sí, otra vez) que cada diagrama de secuencia representa los cursos básicos y alternos de acción;
3. Verificar que las operaciones se asignaron correctamente a las clases;
4. Revisar las clases del diagrama de clase y ver que todas tienen atributos y operaciones apropiadas.
5. Si el diseño usa patrones u otras construcciones detalladas de aplicación, verificar que estos detalles se reflejan en el diagrama de secuencia;
6. Relacione los requisitos funcionales, no funcionales a los casos de uso y a las clases, para asegurarse que fueron considerados todos;
7. Verificar que los programadores "analizaron" el diseño y que pueden construirlo para que funcione como se ha previsto;
8. Verificar que todos los atributos son correctos, sus valores de retorno y la lista de parámetros en sus operaciones son completas y correctas;
9. Generar la cabecera de código de las clases e inspecciónalo;
10. Revisa el plan de prueba para esta versión.

2.2.1.3. Implementación

Rosenberg y Stephens (2007), en la fase de implementación, para comenzar la codificación desde el diseño debemos; definir la arquitectura técnica, tener los diagramas de secuencia para cada caso de uso y, haber revisado el diseño. Luego codificar directamente desde el diagrama de secuencia, implementar los cursos básico y alterno, mantener sincronizado el diseño y código fuente, centrarse en las pruebas unitarias.

Presentamos los principios más importantes para la implementación, recomendados por Rosenberg y Stephens (2007), son:

1. Conducir la codificación directamente desde el diseño;
2. Si la codificación revela que el diseño está mal, cámbialo y revisa el proceso;
3. Revisar periódicamente el código;
4. Cuestionar siempre las opciones de diseño del framework;

5. No deje que el framework se haga cargo de la capa del negocio;
6. Si el código comienza a salirse de control, para y revisa el diseño;
7. Mantener sincronizados el diseño y el código;
8. Centrarse en las pruebas unitarias, mientras implementa el código;
9. No comentar excesivamente el código, hace que el código sea menos mantenible y difícil de leer;
10. Recordar implementar los cursos alternos y los cursos básicos.

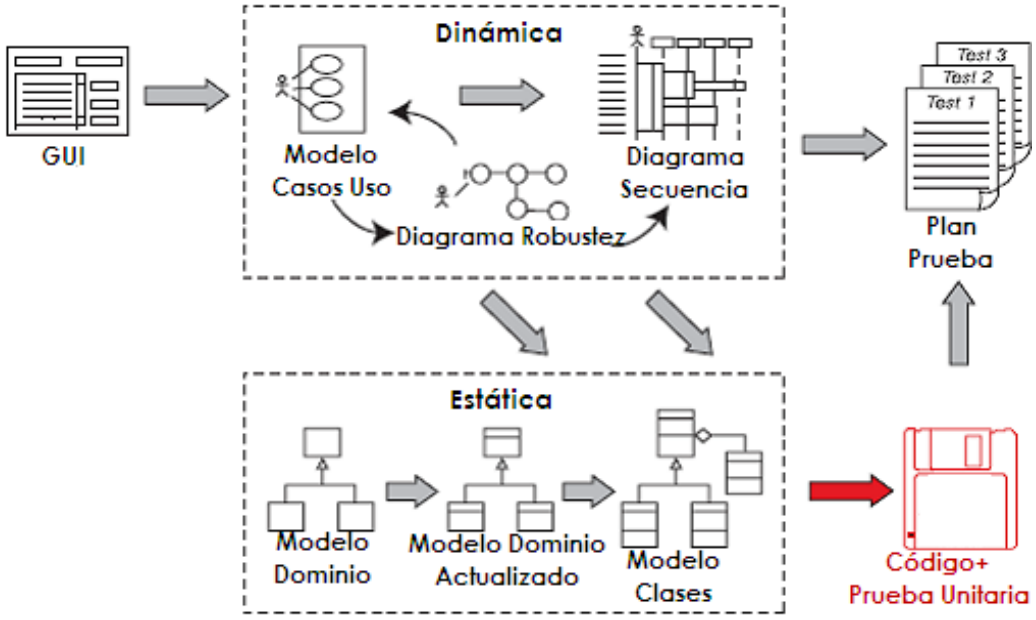


Figura 12. Esquema para implementación (Rosenberg y Stephens, 2007).

Pruebas

Las prueba basadas en diseño proveen un método para producir casos de prueba desde los casos de uso y, verificar que los escenarios se implementaron correctamente. Las pruebas deben estar estrechamente ligadas a los requisitos, para verificar que cada requisito ha sido implementado correctamente. Comenzar las pruebas identificando los casos de prueba desde los diagramas de robustez, luego escribir el código para pruebas unitarias durante la implementación.

Tabla 12. Las pruebas y su aplicación

Prueba	¿Cuándo Aplicar?
Prueba unitaria.- Prueba para componentes individuales de software (controladores,	La prueba unitaria se ejecuta en cada integración del software durante el

operación). El resultado se usa para simular entradas y salidas de un componente de forma que opere en modo autónomo.	desarrollo, incluso en corrección de errores cuando el software es derivado a pruebas de sistema.
Prueba de aceptación.- Prueba conducida por el cliente para determinar si el sistema reúne los requisitos especificados en el contrato.	La prueba de aceptación opera en línea con la prueba del sistema, para probar que se entrega el sistema especificado.

Fuente: (Sociedad Computacional Británica, 2007).

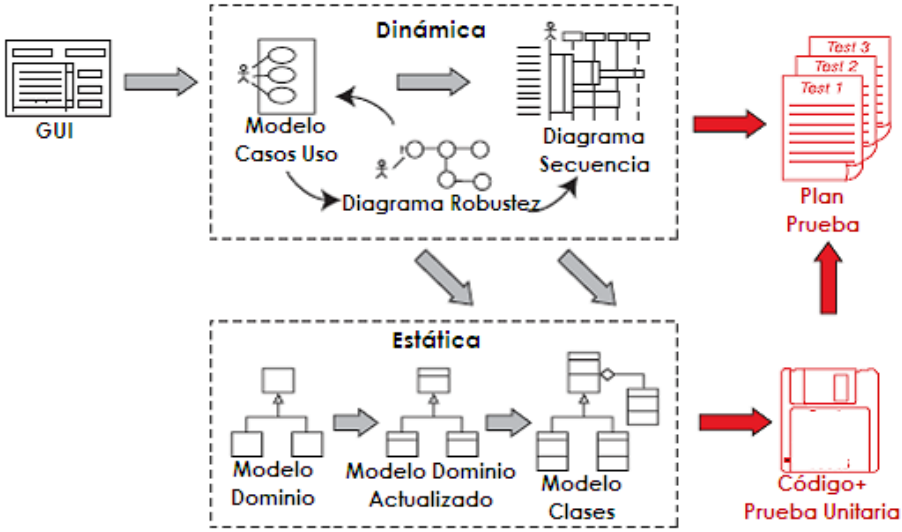


Figura 13. Esquema para pruebas basadas en diseño (Rosenberg y Stephens, 2007).

Las recomendaciones más importantes para las pruebas, según Rosenberg y Stephens (2007), son:

1. "Pensar en las pruebas", si encuentras y corriges el error durante la prueba, los usuarios no lo encontrarán en el producto final.
2. Prepara cada prueba por adelantado y, aplica cada prueba en el momento correcto;
3. Al realizar pruebas unitarias, crea una prueba para cada controlador de los diagramas de robustez. Además, crea una prueba unitaria para cada operación de las clases de diseño. A veces, son lo mismo, un controlador es implementado como una operación simple en una clase, y otras veces un controlador se realiza mediante varias operaciones;

4. Para sistemas de tiempo real, usa los diagramas de estado como base para los casos de prueba;
5. Verifica a nivel de requisitos, cada requisito debe ser explicado;
6. Construye una matriz para asistir la verificación de requisitos;
7. Realiza pruebas de aceptación para cada escenario de los casos de uso;
8. Diseñar pruebas para cubrir una ruta completa de escenarios desde un curso básico;
9. Usa una herramienta, como JUnit, para almacenar y organizar las pruebas unitarias;
10. Realiza mantenimiento a tus pruebas unitarias de manera regular.

2.2.2. Calidad del Producto Software

Según Witold (2014) la calidad de un producto tiene diferentes definiciones. El "grado en que un conjunto de características inherentes cumple con los requisitos" (ISO 9000, 2005); "valor para el cliente" o "niveles de defectos" (Highsmith, 2002), según Kitchenham y Pfleeger (1996) la calidad del software, presenta cinco visiones: (a) La visión trascendental; considera la calidad como algo que puede reconocerse pero es difícil de definir. (b) La visión del usuario; percibe la calidad como idoneidad para un propósito, al evaluar la calidad de un producto, uno debe hacerse la pregunta clave ¿El producto software satisface las necesidades y expectativas del usuario?. (c) La visión de desarrollo; aquí se entiende por calidad la conformidad con la especificación, el nivel de calidad de un producto software está determinado por la medida en que el producto cumple con sus especificaciones. (d) La visión del producto software; la calidad se considera vinculada a las características inherentes del producto, es decir, las cualidades internas, determinan sus cualidades externas. (e) La visión basada en el valor; la calidad, depende de la cantidad que un cliente esté dispuesto a pagar por ella (Naik y Tripathy, 2008, p.5).

El modelo de calidad esta compuesto por la calidad externa e interna según la NTP ISO/IEC 9126-1:2004 (2004), que presnta características y sub características como se muestra en la figura 14.

Scalone (2006), opina que el modelo de calidad esta compuesto por la calidad externa e interna a nivel de productos software, existen distintos modelos y estándares que poseen un conjunto de características, sub características, atributos y métricas. El equipo de desarrollo

debe evaluar la calidad de software durante el ciclo de vida del software.

Durante el desarrollo del software, la calidad comprende los requisitos, especificaciones y diseño del sistema. La calidad de concordancia es un aspecto centrado en la implementación (Pressman, 2002), establece una relación más "intuitiva":

> *Satisfacción del usuario = producto satisfactorio + buena calidad + entrega dentro de presupuesto y del tiempo establecidos.*

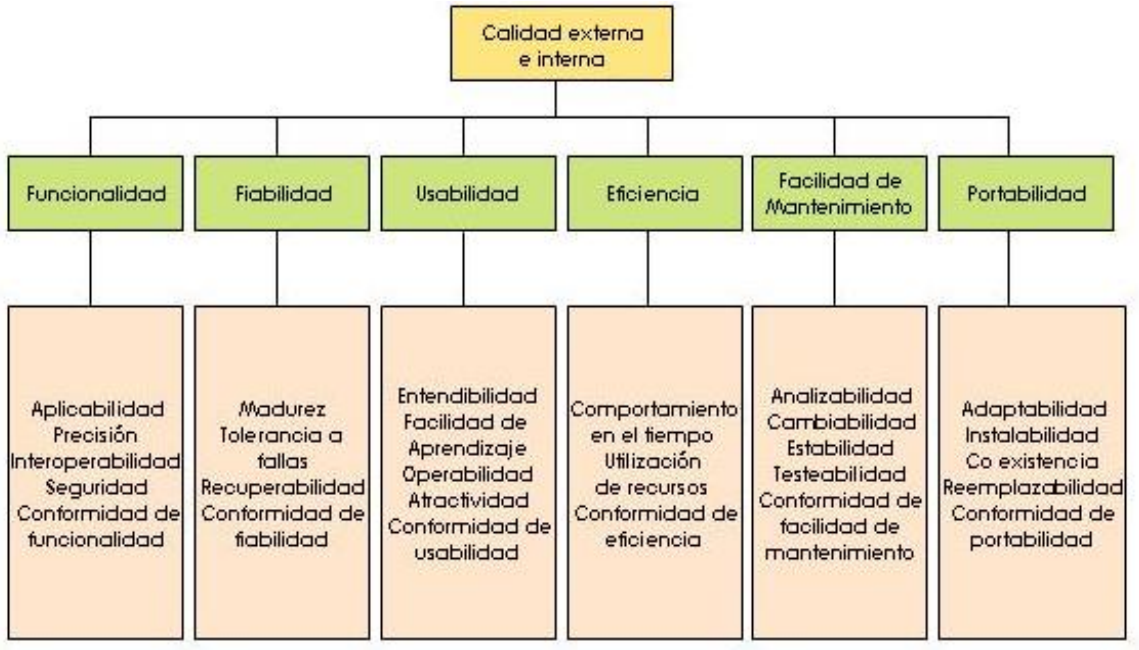


Figura 14. Modelo de calidad (NTP ISO/IEC 9126-1:2004, 2004).

Los "objetivos físicos" de la ingeniería de calidad del software, son: (a) La calidad interna, opera con código (desarrollado o en desarrollo) y artefactos relacionados, como documentos fuente, arquitectura, planes y resultados de pruebas unitarias. (b) La calidad externa, se mide ejecutando ejecutables en el llamado entorno técnico (operación y mantenimiento), artefactos relacionados como arquitectura, documentación de operación, manuales, planes y resultados de pruebas del sistema, y servicios como capacitación de operación y mantenimiento. (c) La calidad en uso, opera en el producto listo en su entorno comercial (tanto emulado como real) y artefactos relacionados, como todos los manuales de usuario y servicios (aplicación y capacitación de uso) (Witold, 2014, p.39).

La calidad en el ciclo de vida del software, durante el análisis es especificada por los requisitos de los usuarios, en las fases de diseño e implementación, la calidad externa se

traduce en un diseño técnico, como se muestra en la figura 15.

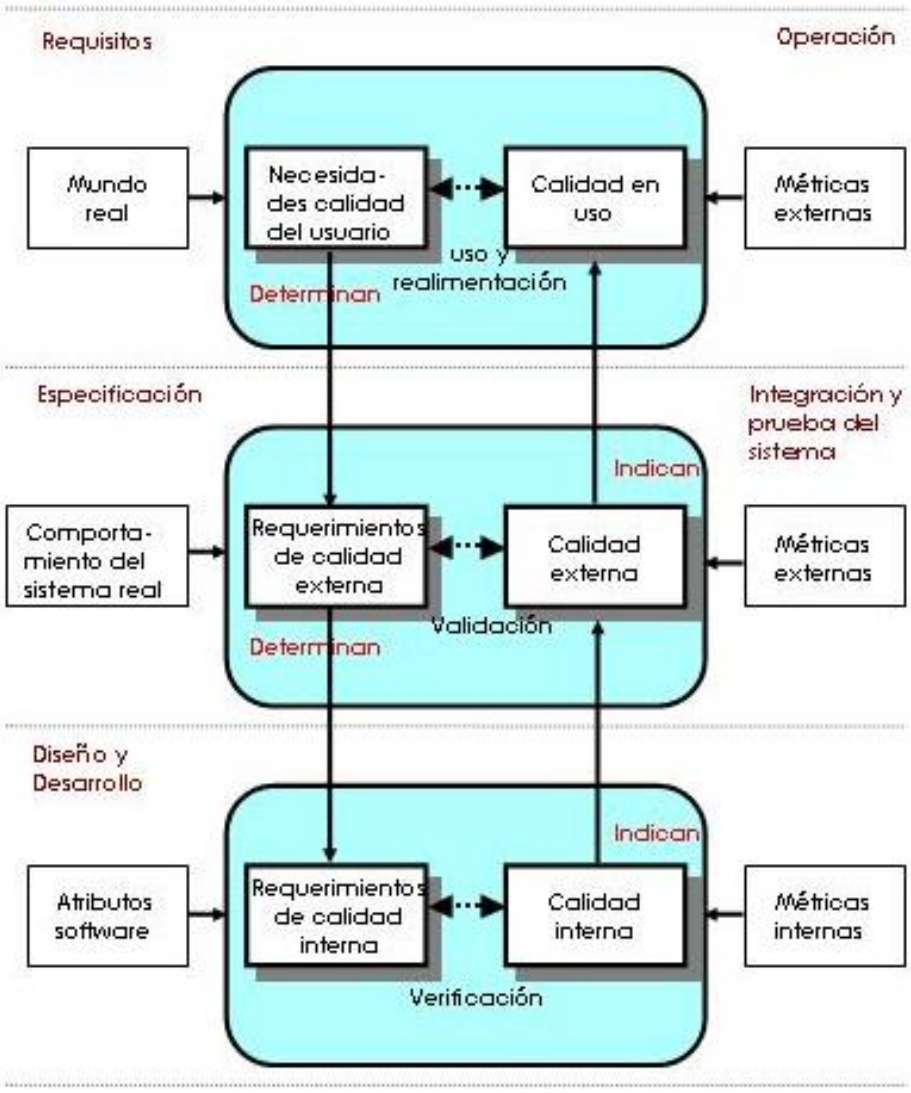


Figura 15. La calidad en el ciclo de vida del software (NTP ISO/IEC 14598-1, 2004).

De acuerdo a las fases para un modelo genérico del proceso de desarrollo de software, tenemos: (a) la calidad interna, es crucial en la fase de implementación. (b) la calidad externa, tiene el papel más importante en el diseño y más tarde, desde la fase de pruebas del sistema hasta que el sistema se retira del servicio. (c) la calidad en uso, se debe observar en la fase de definición de requisitos y luego nuevamente desde la fase de pruebas del sistema hasta que el sistema se retire del servicio (Witold, 2014, p.39).

2.2.2.1. Calidad Externa

Es la totalidad de las características del producto software desde una perspectiva externa. Es la calidad cuando el software es ejecutado, la cual es típicamente medida y

evaluada mientras se prueba en un ambiente simulado con datos simulados y usando métricas externas. Durante las pruebas, muchas fallas serán descubiertas y eliminadas. Sin embargo, algunas fallas todavía pueden permanecer después de las pruebas. Como es difícil corregir la arquitectura de software u otros aspectos fundamentales del diseño del software, el diseño fundamental permanece sin cambios a través de las pruebas (NTP-ISO/IEC 9126-1, 2004, p.9). Según la NTP ISO/IEC 9126-2 (2004) los requerimientos de calidad externa, incluyen requerimientos derivados de las necesidades de calidad de usuarios y, calidad de requerimientos de uso. Estos requerimientos son usados como los objetivos para la validación en varias etapas de desarrollo.

2.2.2.2. Calidad Interna

Es la totalidad de las características del producto software desde una perspectiva interna. La calidad interna es medida y evaluada en base a los requerimientos de calidad interna. Los detalles de la calidad del producto software pueden ser mejorados durante la implementación, revisión y prueba del código software pero la naturaleza fundamental de la calidad del producto software representada por la calidad interna permanece sin cambios a menos que sea rediseñado (NTP-ISO/IEC 9126-1, 2004, p.9). Según la NTP-ISO/IEC 9126-3 (2005), la calidad interna puede incluir modelos estáticos y dinámicos, otros documentos y código fuente.

2.2.2.3. Métricas

La métrica es "una medida cuantitativa del grado en que un sistema, componente o proceso posee un atributo dado" (IEEE Std 610.12-1990).

Para Rodríguez y Harrison (1997), el conjunto de métricas debe indicar que aspectos de la calidad mediremos y para quién está dirigido, el conjunto de métricas a utilizar debe ser de un modelo de calidad. Las métricas deben estar validadas teóricamente o experimentalmente. La validación experimental, se realiza mediante métodos empíricos como estadísticos para evaluar la utilidad e importancia de las métricas.

La valoración de las características de calidad del producto software, se realiza mediante las características y sub características del modelo de calidad interna y externa para los productos software según la norma NTP ISO/IEC 9126-1:2004.

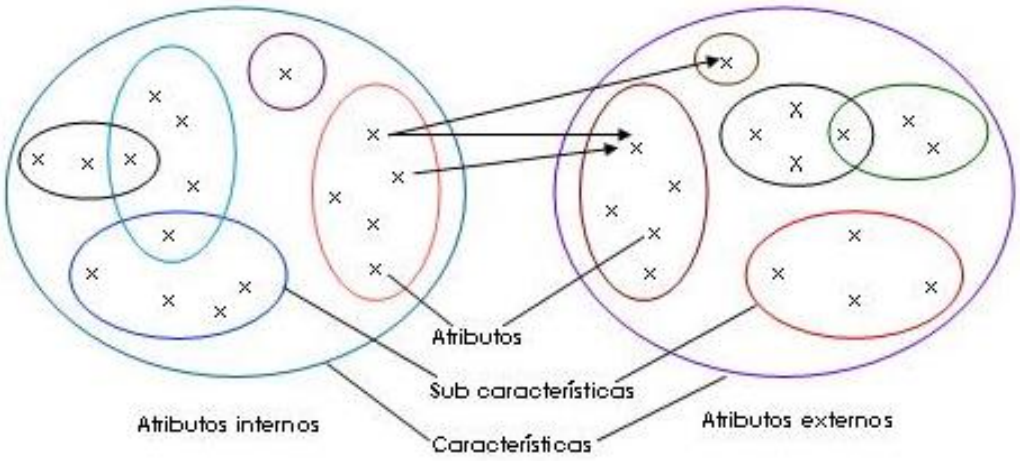


Figura 16. Características, sub características y atributos NTP ISO/IEC 14598-1:2005.

Según la NTP ISO/IEC 9126-2 (2004), los requerimientos de calidad externos para las características de calidad deben definirse como requerimientos de calidad usando métricas externas, deben transformarse en requerimientos de calidad internos y, usarse como criterios para evaluar un producto; las métricas externas, usan medidas derivadas del comportamiento del software, mediante la prueba, operación y observación del software ejecutable. Las métricas externas proporcionan a los; usuarios, evaluadores, probadores, desarrolladores, el beneficio que puedan evaluar la calidad del producto software durante las pruebas u operación.

Según la NTP-ISO/IEC 9126-3 (2005), los requerimientos específicos de calidad interna deben ser descritos cuantitativamente usando métricas internas. Las métricas internas se aplican a un producto software no ejecutable (una especificación o código fuente) durante el diseño y la codificación. Cuando se desarrolla un producto software, los productos intermedios deben ser evaluados usando métricas internas que permitan medir sus propiedades. El objetivo principal de las métricas internas es asegurar que se logre la calidad externa y la calidad de uso requerida. Las métricas internas proporcionan una forma de evaluar la calidad del producto software, antes que el producto software sea puesto en ejecución. Las métricas internas miden atributos internos o indican los atributos externos a través del análisis de las propiedades estáticas de los productos software.

2.2.3. Metodología Ágil Iconix con Intervención

La intervención se realiza mediante los principios y valores para desarrollo de software ágil, la ingeniería de requisitos para procesos agiles, patrones de diseño, y los

principios de implementación (SOLID); responsabilidad única, abierto cerrado, sustitución de Liskov, segregación de interfaz e inversion de dependencia, con la finalidad de mejorar la calidad del producto software durante el desarrollo.

2.2.3.1. Fase de Análisis de Requisitos con Intervención

La ingeniería de requisitos (RE) describe las tareas que definen los requisitos de un software nuevo o rediseñado, teniendo en cuenta los requisitos en conflicto de los interesados. Durante la primera fase del proceso de ingeniería de requisitos, que se conoce como la fase de obtención de requisitos, los interesados analizan y acuerdan los requisitos del software (Nuseibeh y Easterbrook, 2000).

La ingeniería de requisitos es un proceso técnico, social y cognitivo complejo, que produce requisitos para un sistema de software (Maiden y Hare, 1998). La obtención de requisitos es una actividad importante en el proceso de ingeniería de requisitos, porque implica descubrir y formalizar los requisitos del software (Sun, Zeng y Liu, 2011).

La ingeniería de requisitos es parte de un proceso social centrado en el ser humano, llamado ingeniería de software; los requisitos se tratan como una unidad de trabajo para la cual los interesados colaboran, es decir, se comunican entre sí, comparten conocimiento e intercambian información, para lograr colaboración impulsada por los requisitos durante su desarrollo y gestión de los productos software posteriores, es decir, código, diseño, etc., se define como colaboración basada en requisitos (Damian, Kwan y Marczak, 2010).

Se ha revisado artículos de conferencias sobre agilidad y revisiones sistemáticas de literatura del 2010 y 2011 de temas ágiles en ingeniería de software (Jalali y Wohlin, 2010), desarrollo de software distribuido (Prikladnicki y Audy, 2010) y proyectos grandes (Barlow et al., 2011). En estos artículos, las empresas tratan la ingeniería de requisitos como parte de ingeniera de software ágil, las grandes organizaciones de proyectos de software consideran los requisitos desde una perspectiva ágil; otras organizaciones rediseñan sus prácticas de especificación de requisitos para "volverlas más ágiles" (Grewal y Maurer, 2007; Kendall et al., 2010; Christou et al., 2010; Gary et al., 2011) la mayoría trata la priorización de los requisitos durante una iteración, existiendo poca evidencia empírica y el contraste con la teoría sobre la priorización de requisitos ágiles a gran escala.

Los interesados en la funcionalidad del software tienen diferentes necesidades, expectativas, su propia experiencia, prejuicios y puntos de vista que deben ser satisfechos por la presentación y entrega del sistema futuro (Robertson, 2001). A pesar de tener puntos de vista alternativos sobre los requisitos, las partes interesadas colaboradoras buscan llegar a un consenso y acordar la lista final de requisitos del sistema (Robertson y Robertson. (2006).

Desde la perspectiva del usuario, un requisito se puede definir como "una condición o capacidad que necesita un usuario para resolver un problema o alcanzar un objetivo". Desde el lado del sistema, se puede definir como "una condición o capacidad que debe cumplir o poseer un componente del sistema o sistema para satisfacer un contrato, norma, especificación u otro documento formalmente impuesto" (IEEE, 1998).

Wnuk, Gorschek, y Zahda (2012), han realizado un estudio sobre la obsolescencia de requisitos, clasificaron los requisitos según la propuesta de Aurum y Wohlin (2005) y SWEBOOK por tipos de requisitos: funcionales, de calidad y en categorías; según SWEBOOK se incluye requisitos relacionados al diseño y arquitectura. Los tipos de requisitos que probablemente se vuelvan obsoletos en mayor grado, son: los requisitos ambiguos, inconsistentes e incorrectos; es decir con certeza que los requisitos se vuelven obsoletos si presentan insuficiencias en: corrección, coherencia y especificación ambigua; y los requisitos que probablemente no se volverán obsoletos, son: los requisitos relacionados a estándares y las leyes. En la categoría de requisitos cambiantes, se clasifico a los requisitos funcionales originados por: clientes, usuarios finales y tipos de desarrolladores. Los requisitos de los expertos de dominio quienes participan en la priorización de todos los requisitos, se considera menos propensos a quedar obsoletos que los requisitos de los: clientes (Wnuk, Regnell y Karlsson, 2009), usuarios finales y desarrolladores.

Según Daneva, Van der Veen, Amrit, Ghaisas, Sikkel y Kum (2012), en la ingeniería de requisitos ágil se ha identificado cuatro características del proyecto que influyen en la toma de decisiones sobre priorización de requisitos al momento de una iteración, siendo: (i) La necesidad de aceptar el cambio, (ii) Las limitaciones del proyecto, (iii) Alcance y (iv) La cantidad de personas del proyecto; (i) y (ii) tienen un impacto en la priorización de requisitos ágil para los proyectos grandes y pequeños, y las características (iii) y (iv) son

exclusivas de proyectos grandes.

En proyectos de desarrollo ágil incremental se tiene como objetivo liberar la funcionalidad del software en pocas semanas (Larman, 2004). Según Wiegers y Beatty (2013), se tendrán esfuerzos de desarrollo de requisitos frecuentes, pero pequeños, estos proyectos comienzan al inicio recogiendo las necesidades del usuario en historias que describen los principales objetivos que el usuario quiere lograr con el sistema. En este enfoque, es necesario aprender sobre las historias para estimar su esfuerzo de desarrollo y priorizarlas. Priorizar las necesidades de los usuarios permite asignar a los incrementos de desarrollo específicos, llamados iteraciones. Esos requisitos asignados se pueden explorar con más detalle de la forma justo a tiempo para cada ciclo de desarrollo.

Wiegers y Beatty (2013), opinan que las declaraciones que describen, qué tan bien el sistema hace algo, son los atributos de calidad. Revise las palabras que describen las características deseables del sistema: rápido, fácil, fácil de usar, confiable y seguro. Se tiene que trabajar con los usuarios para comprender exactamente lo que quieren decir con estos términos ambiguos y subjetivos para escribir los objetivos de calidad claros y verificables.

En la tabla 13, se muestra una revisión sistemática de obtención de requisitos, el cual ha sido interpretado considerando únicamente la ingeniería de requisitos agiles (Carrizo y Rojas, 2016).

Tabla 13. Prácticas de obtención de requisitos en desarrollo ágil de software

Nº	**Título**	**Autor**	**Práctica**	**Foco**
1	Agile Requirements Engineering Practices, An Empirical Study.	Cao, L. & Ramesh, B.	Las organizaciones pretenden transferir eficazmente las ideas del cliente al equipo de desarrollo, usando técnicas como historias de usuario para definir los requisitos de alto nivel, mediante la comunicación cara a cara sobre las especificaciones	Personas

			escritas.	
2	An Agile Requirements Elicitation Approach based on NFRs and Business Process Models for Micro-Businesses	Macasaet, R., Chung, L., Garrido, J., Noguera, M. & Rodríguez, M.	Para obtener y definir requisitos adecuados, es importante que el propietario de la microempresa enumere los objetivos del negocio y diagrame los procesos de negocios, y junto con el desarrollador analicen estos diagramas y encuentren patrones similares de negocios existentes que sean de utilidad.	Negocio
3	An approach to requirements elicitation and analysis using goal	Chand, M., Reddy, K., Rao, A. & Kumar, J.	Para comprender mejor los sistemas más complejos se debe obtener múltiples vistas de los distintos stakeholders. Utilizar palabras claves en la identificación de requisitos de entrada, del sistema y de salida.	Técnicas
4	Generating User Stories in Groups	Read, A., Gallagher, E., Nguyen, C. & de Vreede, G.	Permitir que los stakeholders generen historias en grupos para facilitar el flujo de información valiosa y no olviden información como sucede en historias de usuarios. Además permite que un problema pueda ser visto desde varias perspectivas y el conocimiento de un stakeholder puede ser ampliado por otro.	Técnicas
5	Using Business Rules in EXtreme Requirements	Leonardi, M. & do Prado Leite, J.	Para mejorar la comunicación con los stakeholders, es necesario desarrollar un vocabulario y una semántica que unifique el lenguaje. Esto facilita	Técnicas

			la comprensión del problema, el proceso de construcción de escenarios, reglas de negocio y ayuda en su descripción.	
6	Using Human Factors Standards to Support User Experience and Agile Design	Maguire, M.	Los usuarios, deben ser conducidos de una etapa a otra, hasta alcanzar su objetivo, mejorando su experiencia, y permitir a los desarrolladores definir los requisitos no funcionales.	Técnicas
7	Development of Agile Security Framework Using a Hybrid Technique for Requirements Elicitation	Singhal, A.	La utilización en conjunto de historias de intrusión y diagramas de árbol de ataque, durante la educción de requisitos de seguridad, permite agilizar el proceso de generar estrategias para evitar amenazas y comprender cómo se efectúan ataques al sistema.	Técnicas

Seleccionar el ThinkLet (TL) correcto es una tarea crítica, realizada por el ingeniero de colaboración, con referencia a la meta del cliente, los riesgos, las habilidades de los profesionales y los resultados asignados (Kolfschoten y Rouwette, 2006). El mapa de elección del TL se usa para encontrar la combinación adecuada de TL que persigue el objetivo de cada tarea según el patrón de colaboración, mostrado en la tabla 14.

Tabla 14. Mapa de elección del ThinkLet

Thinklet Name	**FreeBrainstorm**	**OnePage**	**Comparative Brainstorm**
Starting Point	Combinación excelente	Combinación excelente	Combinación excelente
FreeBrainstorm	Combinación imposible	Combinación imposible	Combinación excelente

OnePage	Combinación imposible	Combinación imposible	Combinación problemática

Fuente: (Kolfschoten y Vreede, 2007).

A continuación se describe todas las actividades de 1 a 6, en la figura 00, se observa el flujo de los procesos, según (Kolfschoten y Rouwette, 2006).

Actividad 1.- Identificar los requisitos de usuario del sistema

Se realiza la tarea generar y la técnica "FreeBrainstorm". El taller comienza con la selección de los usuarios; los usuarios aplican "FreeBrainstorm", que es la lluvia de ideas de forma libre, y generan los requisitos de usuarios del sistema.

Actividad 2.- Categorizar los requisitos del usuario

Se ejecuta la tarea categorizar y la técnica "Popcorn-sort". Los participantes para categorizar los requisitos utilizan la técnica Popcorn-Sort, luego de la técnica divergente FreeBrainstorm, que consiste en buscar convergencias de los grupos de usuarios para categorizar los requisitos de usuarios del sistema.

Actividad 3.- Ubicar los requisitos del usuario dentro de cada categoría

Se ejecuta la tarea generar y la técnica "Leaf-hooper". La técnica de esta actividad es apropiada para con la participación de equipo, ubicar los requisitos del usuario dentro de cada categoría, mediante la lluvia de ideas, procedimiento que se realiza iterando hasta finalizar con todos los requisitos.

Actividad 4.- Asegurar la categorización correcta

Se ejecuta la tarea evaluar y la técnica "Bucket-Walk". Se utiliza para validar los resultados de Popcorn-Sort o Leaf-Hopper; se invita a los participantes a revisar los contenidos de cada categoría para asegurarse que son correctos, si algún participante cuestiona el contenido de una categoría, el grupo inicia una discusión moderada para tomar decisiones posteriores. Esta técnica es utilizada para eliminar la redundancia y consolidar las características para cada grupo de interés, en esta etapa las partes interesadas colaboradoras tienen la oportunidad de plantear cualquier cuestión relacionada con las interdependencias entre los requisitos, incluyendo aspectos tales como conflictivos y recursivos, dependencias entre requerimientos de la lista y tomen un acuerdo de final.

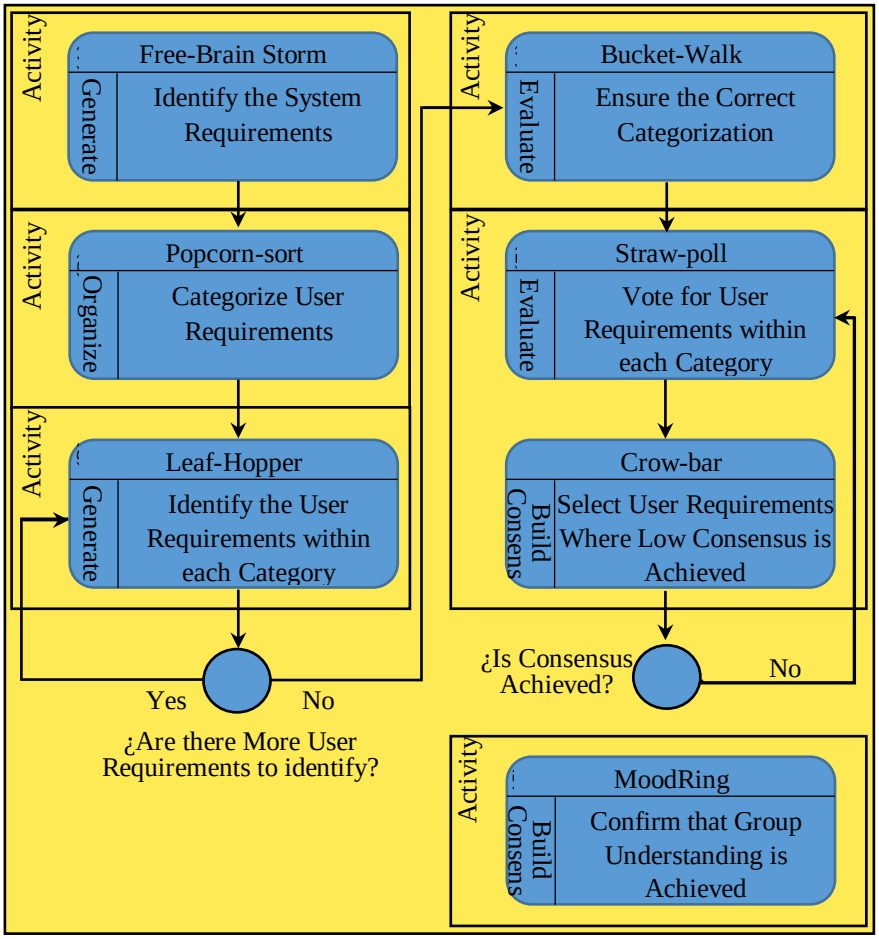


Figura 17. Proceso de colaboración para obtener requisitos de usuario (Kolfschoten y Rouwette, 2006).

Actividad 5.- Votar por los requisitos de usuario dentro de cada categoría

Se ejecuta la tarea evaluar y la técnica "Straw-poll". Los participantes evalúan los requisitos dentro de cada categoría según su prioridad, luego hacen una sesión de votación, la técnica "Straw-poll" ayuda a medir el consenso y a revelar los patrones de acuerdo o desacuerdo en el grupo. Usando el método elegido de votación (una escala de 1 a 5), los participantes deciden cuál de los requisitos son más importantes (tienen prioridad) que otros. También se ejecuta la tarea construir consenso y la técnica "Crow-Bar", técnica iterativa que permite estudiar y examinar los supuestos y compartir información, provocando una discusión enfocada sobre temas que el grupo tiene bajo consenso, la actividad se repite con los participantes hasta que se alcanza un nivel aceptable de consenso.

Actividad 6.- Confirmar que la comprensión grupal tiene consenso

Se ejecuta la tarea construir consenso y la técnica "MoodRing", técnica utilizada para

rastrear patrones de consenso en una problema único en tiempo real, e indicar cuándo es el momento de parar de hablar y tomar una decisión, esta técnica garantiza el acuerdo y consenso entre los participantes sobre el resultado general de la sesión

La ingeniería de requisitos ágil como una técnica de ingeniería de software, tiene muchas prácticas y técnicas para realizar el desarrollo (obtener, analizar, especificar) de requisitos agiles de software nuevo y en mantenimiento, los interesados (clientes, usuarios finales, expertos de dominio, desarrolladores) ejecutan las tareas: descubrir, comunicar, colaborar, compartir conocimientos, analizar, priorizar, formalizar y generar consenso sobre los requisitos para desarrollar los productos software de una iteración, con entregas pequeñas en el contexto de agilidad. Los estudios sobre clasificación de requisitos indican: tipos que se vuelvan obsoletos (ambiguos, inconsistentes e incorrectos), requisitos que no se volverán obsoletos fácilmente (relacionados a estándares y leyes), requisitos cambiantes originados (clientes, usuarios finales y desarrolladores), requisitos menos propensos a quedar obsoletos (expertos de dominio). La priorización de requisitos agiles es importante porque se requiere para desarrollar la siguiente iteración, las características de calidad deben trabajarse con los interesados, los stakeholders deben participar en la identificación y priorización de requisitos. En base a los planteamientos anteriores, realizamos la intervención para: obtener, analizar y especificar requisitos agiles visto en la tabla 15.

Tabla 15. Desarrollo de la fase de análisis de requisitos del software con intervención

Tarea	**Técnica**	**Producto software**	**Responsable**
Obtener los requisitos de software	a. Lluvia de ideas b. Focus groups c. Análisis documental	Requisitos funcionales y no funcionales	a. Cliente b. Usuario final c. Experto de dominio d. Desarrollador
a. Clasificar los requisitos de software b. Categorizar los requisitos de software	a. Sesiones de trabajo (Workshops) b. Votación de stakeholders	a. Paquetes de requisitos funcionales y no funcionales clasificados b. Requisitos	a. Cliente b. Usuario final c. Experto de dominio d. Desarrollador

c. Evaluar la clasificación de requisitos de software d. Priorizar los requisitos de software		funcionales y no funcionales priorizados	
Crear casos de prueba	Escribir al menos un caso de prueba por cada requisito	Casos de prueba	Usuario final

A continuación definimos la matriz para el análisis de requisitos de la metodología agil Iconix; donde proponemos obtener, clasificar, categorizar, evaluar y priorizar los requisitos según la tabla 15, para crear el modelo de dominio inicial retiramos la técnica de usar agregación y generalización por ser muy pronto esta abstracción, la tarea "Descubrir casos de uso" para la lista de casos de uso varia en función a que participe los stakeholders quienes según las evidencias practicas crean requisitos y consecuentemente casos de uso mas estables, se retira la tarea "Relacionar requisitos funcionales con los casos de uso", porque durante la creación de los casos de uso se realizo esta tarea, la matriz para la fase de análisis de requisitos con intervención esta en el anexo 6.

2.2.3.2. Fase de Diseño con Intervención

La intervención a la metodología ágil Iconix en la fase de diseño, se realiza mediante los patrones de diseño, donde los patrones de diseño son técnicas probadas de diseño de software, que no brindan código, brindan soluciones generales para resolver problemas de: (a) Diseño y arquitectura. (b) Diseño de sistemas reutilizables. (c) Técnicas para uso de objetos, herencia, polimorfismo. (d) Parametrizar un sistema mediante algoritmos, comportamiento, estados y creación de objetos; como se expone a continuación.

Patrones de diseño

Alexander et. al. (1977), afirma que "cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces".

Diseñar para el cambio

Gamma, Helm, Jhonson y Vlissides (1994), opinan que para maximizar la reutilización; hay que anticiparse a los nuevos requisitos y cambios en los requisitos existentes, diseñar sistemas que puedan evolucionar, hacer frente a los cambios durante las fases de desarrollo. Las causas más comunes de rediseño y los patrones de diseño involucrados son:

a. **Crear un objeto especificando su clase explícitamente.** Nombrar una clase al crear un objeto que relaciona a una implementación concreta en lugar de una interfaz; esto puede complicar cambios futuros, para evitarlo, debemos crear los objetos indirectamente, usar los patrones de diseño: Abstract Factory, Factory Method.

b. **Dependencia de plataformas hardware o software.** Las interfaces externas de los sistemas operativos y las interfaces de programación de aplicaciones (API) varían para las diferentes plataformas hardware y software, el software que depende de una plataforma concreta será difícil de portar a otras plataformas, incluso puede resultar difícil mantenerlo actualizado en su plataforma nativa. Entonces, es importante diseñar nuestros sistemas de manera que se limiten sus dependencias de plataforma, usar el patrón de diseño: Abstract factory.

c. **Dependencias de las representaciones o implementaciones de objetos.** Los clientes de un objeto que saben cómo se representa, se almacena, se localiza o se implementa, quizá deban ser modificados cuando cambie dicho objeto, ocultar esta información a los clientes previene los cambios en cascada; usar el patrón de diseño: Abstract Factory.

d. **Dependencias algorítmicas.** Los algoritmos se amplían, optimizan o sustituyen por otros durante el desarrollo y posterior reutilización, los objetos que dependen de un algoritmo tendrán que cambiar cuando este cambie. Entonces, aquellos algoritmos que probablemente cambien deberían estar aislados; usar los patrones de diseño: Builder, Iterator, Strategy, Template Method.

e. **Fuerte acoplamiento.** Las clases que están fuertemente acopladas son difíciles de usar por separado, porque dependen unas de otras, el fuerte acoplamiento lleva a sistemas monolíticos, donde no se puede cambiar o quitar una clase sin entender y cambiar por otras clases. El sistema se convierte en algo difícil de aprender, portar y mantener. El bajo

acoplamiento aumenta la probabilidad que una clase pueda ser reutilizada ella sola y que un sistema pueda aprenderse, portarse, modificarse y extenderse más fácilmente. Usar de técnicas como el acoplamiento abstracto y la estructuración en capas para promover sistemas escasamente acoplados; usar los patrones de diseño: Abstract Factory, Facade, Observer.

f. **Añadir funcionalidad mediante la herencia.** Derivar un objeto de otra clase no es fácil, cada clase nueva tiene un costo de implementación, definir una subclase también requiere un profundo conocimiento de la clase padre. La composición de objetos en general proporciona alternativas flexibles a la herencia para combinar comportamiento, podemos añadir funcionalidad nueva a una aplicación componiendo los objetos existentes, en lugar de definir subclases nuevas de otras clases existentes; usar los patrones de diseño: Abstract Factory, Facade, Observer.

g. **Incapacidad para modificar las clases adecuadamente.** A veces hay que modificar una clase adecuadamente, quizá necesitamos el código fuente y no tenemos, o cualquier cambio requiere modificar muchas subclases existentes; usar los patrones de diseño: Adapter, Decorator.

Tabla 16. Patrones de diseño propuestos para desarrollo de software con agilidad

		Propósito		
		De Creación	**Estructurales**	**De Comportamiento**
Ámbito	**Clase**	Ninguno	Ninguno	Ninguno
	Objeto	Abstract Factory Singleton	Adapter Decorator Facade	Iterator Observer State Strategy

Fuente: Elaboración propia

Freeman, Freeman, Sierra y Bates (2009), definen un patrón de diseño como “es una solución a un problema en un contexto”, y consideran los siguientes principios de diseño:

a. Identifique los aspectos de su aplicación que varían y sepárelos de lo que permanece igual.

b. Programa una interfaz, no a una implementación.

c. Favorecer la composición sobre la herencia.

d. Esfuércese por diseños débilmente acoplados entre los objetos que interactúan.
e. Las clases deben estar abiertas para la extensión, pero se cierran para su modificación.
f. Depende de las abstracciones. No dependas de clases concretas.

A. Patrón Abstract Factory

"Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas" (Gamma, et. al., 1994, p87); el patrón permite a un cliente utilizar una interfaz abstracta para crear un conjunto de objetos relacionados sin conocerlos, de esta forma el cliente esta desacoplado de las características específicas de los demás objetos. A continuación desarrollamos los usos, estructura, participantes y código C++.

Usos del patrón Abstract Factory

a. Un sistema debe ser independiente de cómo: se crean, componen y representan sus productos.
b. Un sistema debe ser configurado con una familia de productos de entre varias.
c. Una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esta restricción.
d. Quiere proporcionar una biblioteca de clases de productos, y solo quiere revelar sus interfaces, no sus implementaciones.

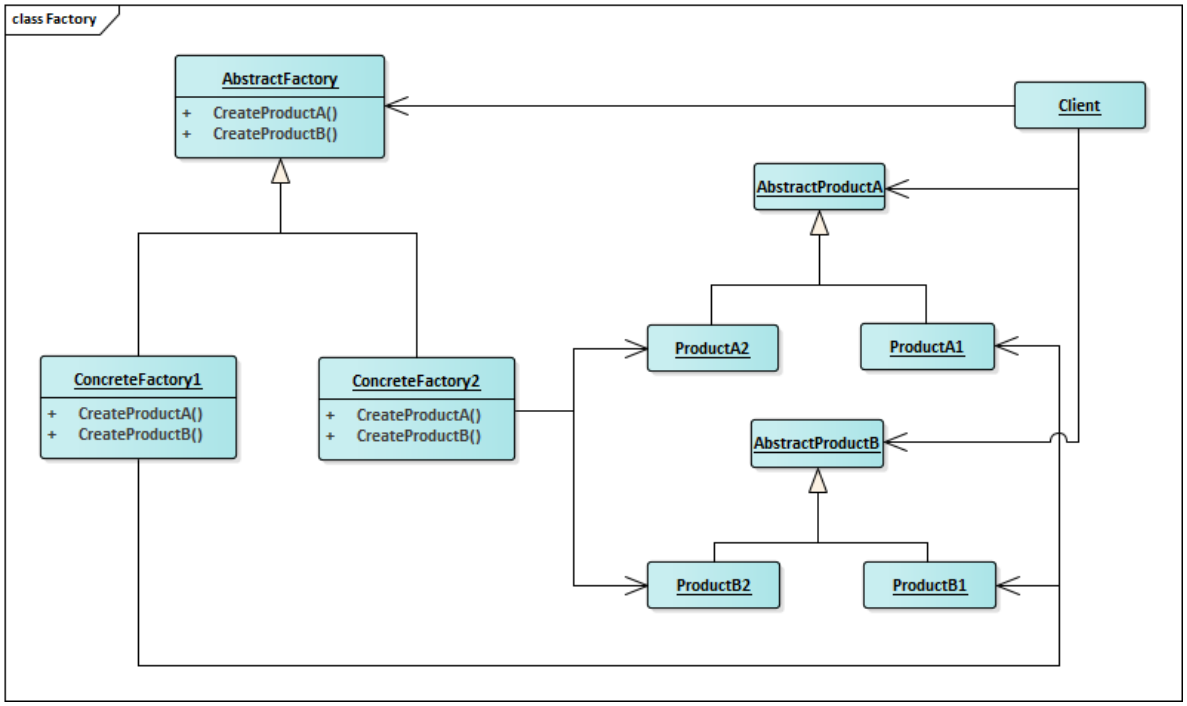


Figura 18. Estructura del patrón abstract factory (Gamma, et. al., 1994, p88).

Participantes

AbstractFactory. Declara una interfaz para operaciones que crea objetos producto abstracto.

ConcreteFactory. Implementa las operaciones para crear objetos producto concreto.

AbstractProduct. Declara una interfaz para un tipo de objeto producto.

ConcreteProduct. Define un objeto producto para ser creado por AbstractFactory. Implementa la interfaz AbstractFactory.

Cliente. Usa solo interfaz declaradas por las clases AbstractFactory y AbstractProduct.

Código del patrón Abstract Factory en C++

```
Class MazFactory{
Public:
    mazeFactory();
    Virtual Maze* MakeMaze() const{
        Return new Maze;
    };
    Virtual Wall* MakeWall() const{
        Return new Wall;
    }
    Virtual Room* MakeRoom(n) const{
        Return new Room(n);
    }
    Virtual Door* MakeDoor (Room* r1, Room* r2) const{
```

```
        Return new Door(r1, r2);
    }
}

Maze* mazeGame::CreateMaze ( MazeFactory factory){
    Maze* aMaze = Factory.MakeMaze();
    Room r1 = facroty.MakeMaze();
    Room r2 = facroty.MakeMaze(1);
    Room r3 = facroty.MakeMaze(2);
    Door* aDoor = factory.MakeDoor(r1,r2);

    aMaze ->AddRoom(r1);
    aMaze ->AddRoom(r2);

    R1 ->setSide(North, facrory.makewall());
    R1 ->setSide(East, Adoor);
    R1 ->setSide(South, facrory.makewall());
    R1 ->setSide(West, facrory.makewall());
    R1 ->setSide(North, facrory.makewall());
    R1 ->setSide(East, facrory.makewall());
    R1 ->setSide(Southt, facrory.makewall());
    R1 ->setSide(West,aDoor);

    Return aMaze;
}
```

B. Patrón Singleton

"Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella" (Gamma, et. al., 1994, p127). Esto, permite el acceso global a dicha instancia mediante un método de clase y el acceso global se refiere a que siempre que se necesite una instancia del objeto, simplemente consulte la clase y le devolverá la instancia única

Gamma, et. al., (1994), afirman que es importante que algunas clases tengan exactamente una instancia. ¿Cómo podemos asegurar que una clase tenga una única instancia y que esta sea fácilmente accesible?, haciendo que la propia clase sea responsable de su única instancia, la clase garantiza que no se puede crear ninguna otra instancia, interceptando las peticiones para crear nuevos objetos; en eso consiste el patrón Síngleton, ahora se explica los usos, estructura y código C++.

.

Usos del patrón Singleton

a. Debe haber exactamente una instancia de una clase, y esta debe ser

accesible desde un punto conocido.

b. La única instancia debería ser extensible mediante herencia, se debe usar la instancia extendida sin modificar su código.

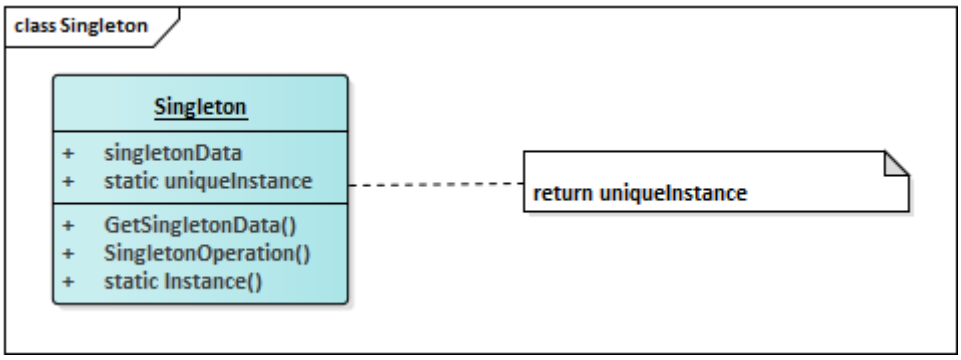


Figura 19. Estructura del patrón singleton (Gamma, et. al., 1994, p. 127).

Participantes

Singleton. Define una operación Instance que permite que los clientes accedan a esta única instancia. Puede ser responsable de crear su propia instancia única.

Código del patrón Singleton en C++

```
class Singleton {
public :
      Static Singleton * Instance();
protected:
      Singleton();
private:
   Static Singleton * _instance;
};
```

C. Patrón Adapter

"Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles" (Gamma, et. al., 1994, p139). El patrón permite desacoplar al cliente de la interfaz implementada, si esperamos que la interfaz cambie con el tiempo, el adaptador encapsula ese cambio para que el cliente no tenga que modificar cada vez que ocurra el cambio, ahora desarrollamos los usos, participantes, estructura y código C++.

Usos del patrón Adapter

a. Cuando se quiere usar una clase existente y su interfaz no concuerda con la

que necesita.

b. Se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas.

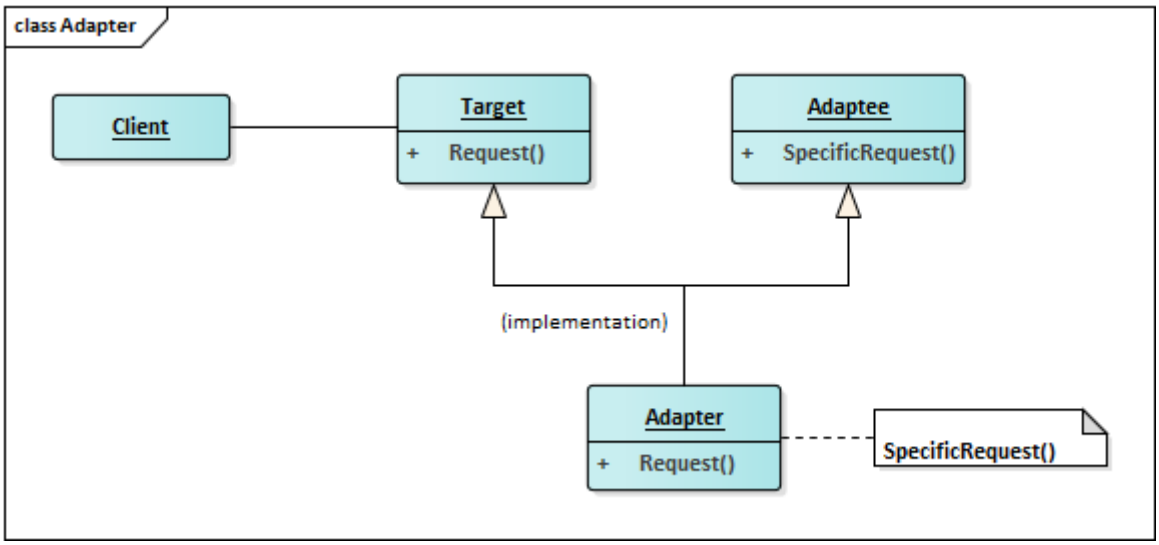


Figura 20. Estructura del patrón adapter (Gamma, et. al., 1994, p. 141).

Participantes

Target. Define la interfaz de dominio específica que usa el cliente.

Client. Colabora con objetos que se ajustan a la interfaz Target.

Adaptee. Define una interfaz existente que necesita ser adaptada.

Adpater. Adapta la interfaz de Adaptee a la interfaz Target.

Código del patrón Adapter en C++

```
Class Shape {
Public:
    Shape();
    Virtual void BoundingBox{
        Point& bottonLeft, Point& topRight
    } cont;
    Virtual manipulator* CreateManipulator() const;
};

Class textView{
Public:
    TextView();
    Void GetOrigin(Coords x, Coords y ) const;
    Void GetExtent(Coords widht, Coords height ) const;
    Virtual bool isEmpty() const;
}
```

```
Class TextShape : public Shape, private TextView{
Public:
      TextShape();
      Virtual void BoundingBox{
            Point bottonleft, point topright
      } const;
      Virtual bool isEmpty() const;
      Virtual Manipulator* CreateManipulator() const;
}

Void TextShape::Bounding Box{
      Point bottonLeft, point topright
}const {
      Coord botton, left, width, height;
      GetOrigin (botton, left);
      GetExtent(width, height);
      bottonLeft = Point(bottom, left);
      topRight = Point (botton + height, left + widht);
}

Bool textShape::IsEmpty ()const{
      Return TextView::Is Empty;
}
Manipulator* TextShape :: CreateManipulator () const{
      Return new TextManipulator(this);
}
```

D. Patrón Decorator

"Añade dinámicamente responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad" (Gamma, et. al., 1994, p.175). En otros términos, cuando se quiere agregar funcionalidad adicional a un objeto individual en lugar de toda una clase, ahora desarrollamos los usos, participantes, estructura y código C++.

Uso del patrón Decorator

a. Se quiere añadir objetos individuales de forma dinámica y transparente, sin afectar a otros objetos.
b. Las responsabilidades pueden ser retiradas.
c. La extensión mediante la herencia no es viable.

Participantes

Component. Define la interfaz para objetos a los que se puede adicionar responsabilidades

dinámicamente.

ConcreteComponent. Define un objeto al que se puede adicionar responsabilidades.

Decorator. Mantiene una referencia a un objeto Component y define una interfaz que se ajusta a su interfaz Component.

ConcreteDecorator. Adiciona responsabilidades al componente.

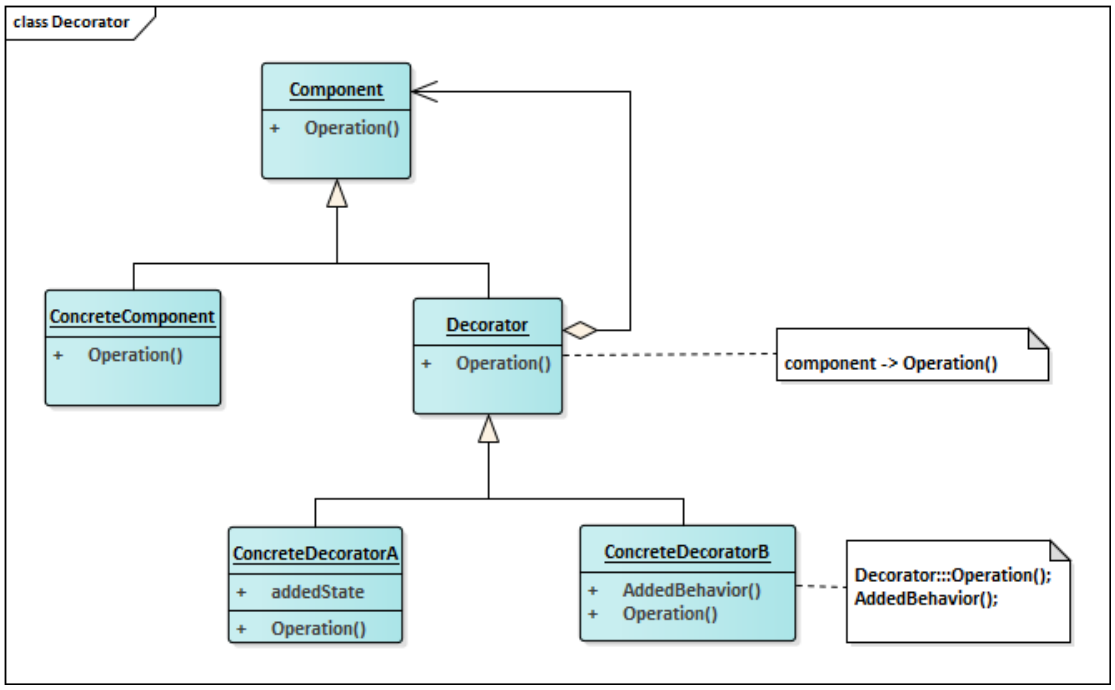


Figura 21. Estructura del patrón decorator (Gamma, et. al., 1994, p. 177).

Código del patrón en Decorator C++

```
class Virtualcomponent {
Public:
      VisualComponent();
      Virtual void Draw();
      Virtual void Resize();
};

Class Decorator : public VisualComponent {
Public:
      Decorator (VisualComponent *);
      Virtual void Draw();
      Virtual void Resize();

Private:
      VisualComponent* _component;
}

Void Decorator :: Draw(){
```

```
    _component -> Draw();
}
Void Decorator :: Resize(){
    _component -> Resize();
}

Class BorderDecorator : public Decorator {

Public:
    BorderDecorator (VisualComponent*, int borderWidth);
    Virtual void Draw();
Private:
    Void DrawBorder(int);
Private:
    Int _width;
};
Void BorderDecorator ::Draw (){
    Decorator::Draw();
    DrawBorder(_width);
}
```

E. Patrón Facade

"Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar" (Gamma, et. al., 1994, p187). Estructurar un sistema en un subsistema ayuda a reducir la complejidad al momento de implementar clases que podrían minimizarse, a continuación el desarrollo de usos, participantes, estructura y código C++.

Usos del patrón Facade

a. Se quiere proporcionar una interfaz simple para un subsistema complejo, los subsistemas se vuelven más complejos a medida que evolucionan.
b. Se tenemos muchas dependencias entre los clientes y las clases que implementan la abstracción. Facade ayuda a desacoplar el subsistema de sus clientes y de otros subsistemas, promoviendo portabilidad y la independencia de subsistemas.
c. Se quiera dividir las capas en subsistemas. Si tenemos mucha dependencia, se puede simplificar las dependencias entre las clases comunicándose entre si únicamente mediante Facade.

Participantes

Facade (Compiler). Conoce que clases del subsistema son responsables ante una petición.

Delega las peticiones de los clientes a los objetos apropiados del subsistema.

Subsystem clases (Scanner, Parser, ProgramNode, etc.). Implementan la funcionalidad del subsistema. Realizan las tareas encargadas por el objeto Facade. No conocen a Facade, es decir no tienen referencia a ella.

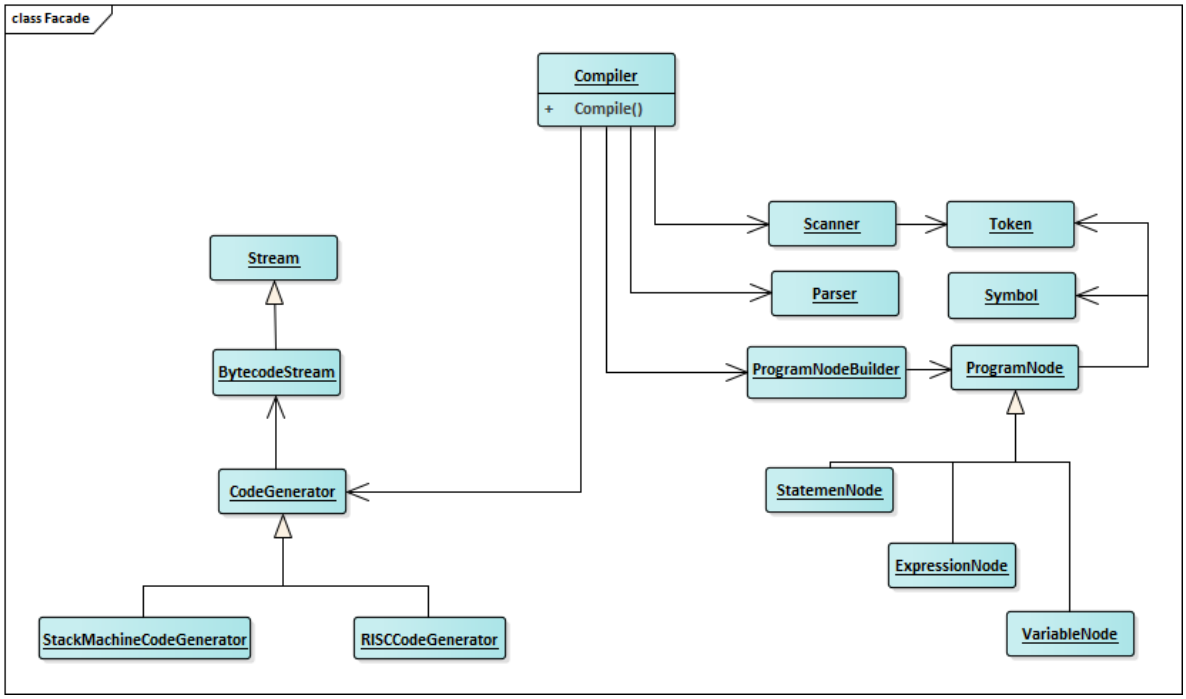


Figura 22. Estructura del patrón facade (Gamma, et. al., 1994).

Código del patrón Facade en C++

```
Class Scanner{
Public:
      Scanner (Isteam);
      Virtual Scanner();
      Virtual Token & Scan();
Private:
      Istream & _inputStream;
};

Class Parser{
Public:
      Parser();
      Virtual Parser();
      Virtual void parse(Scanner, programNodeBuilder);
}

Class ProgramNodeBuilder{
Public:
      ProgramNodeBuilder();
```

```
    Virtual ProgramNode* NewVariable{
        Const char* variableName
    } const;
    Virtual ProgramNode* NewAssignment{
        ProgramNode* variable, ProgramNode* expression
    } const;
    Virtual ProgramNode* NreReturnStatement{
        ProgramNode* Value
    } const;
    Virtual ProgramNode* NewCondition{
        ProgramNode* condition,
        ProgramNode* truePart, ProgramNode* falsePart
    } const;
}

Class ProgramNode{
Public:
    Virtual void GetSourcePosition(int line, int index);
    Virtual void Add(ProgramNode*);
    Virtual void Remove(ProgramNode*);
    Virtual void Traverse(CodeGenerator*);

Protected:
    ProgramNode();
}

Class CodeGenerator {
Public:
    Virtual void Visit (StatementNode*);
    Virtual void Visit (expressionNode*);
Protected:
    CodeGenerator(BytecodeStream*);

Protected:
    BytecodeStream & _ouput;
};
```

F. Patrón Iterator

"Proporciona un modo de acceder secuencialmente a los elementos de un objeto, agregando sin exponer su representación interna" (Gamma, et. al., 1994, p257). Un objeto agregado debería poder accederse a sus elementos sin exponer su estructura interna, por otro lado, el patrón define una interfaz para acceder a los elementos de una lista (objetos), explicamos el desarrollo de usos, participantes, estructura y código C++.

Usos del patrón Iterator

a. Acceder al contenido de un objeto agregado sin exponer su representación

interna.

b. Permitir varios recorridos sobre objetos agregados.

c. Proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas, iteración polimórfica.

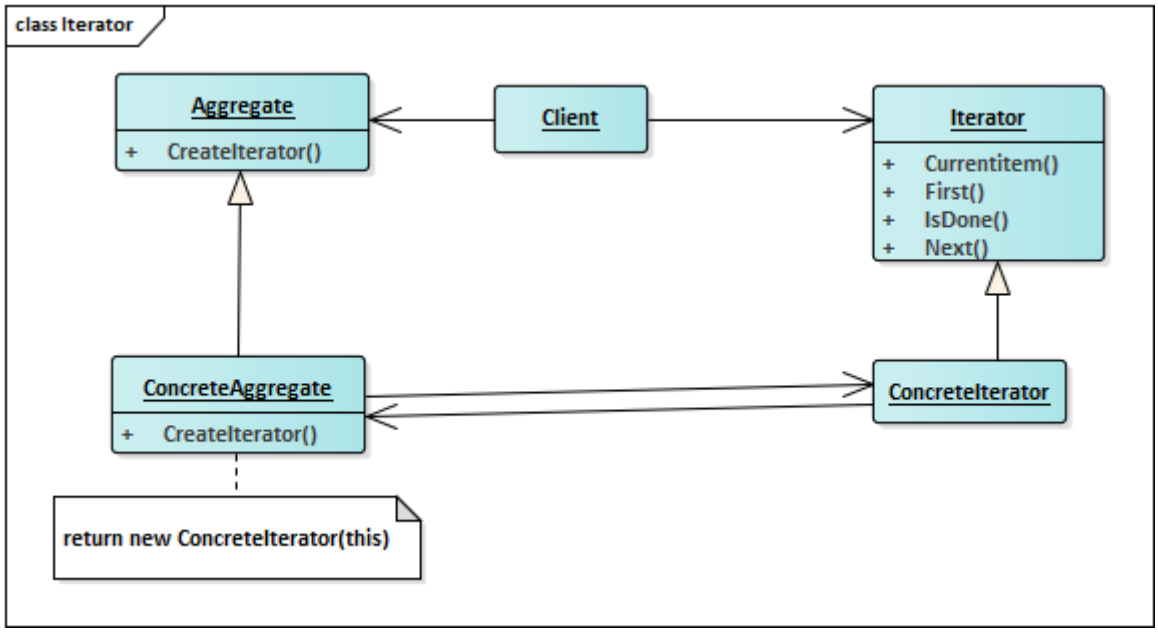


Figura 23. Estructura del patrón iterator (Gamma, et. al., 1994, p. 259).

Participantes

Iterator. Define una interfaz para acceso y atravesado de los elementos.

ConcreteIterator. Implementa la interfaz Iterator. Mantiene la posición actual en el recorrido del Aggregate.

Aggregate. Define la interfaz para crear un objeto Iterator.

ConcreteAggregate. Implementa la interfaz de creación de Iterator para retornar una instancia propia de ConcreteIetrator.

Código del patrón Iterator en C++

```
Template <class Item>
Class List{
Public:
    List (long size = DEFAULT_LIST_CAPACITY);
    Item & Get (long index) const;
};

Template <class Item>
Class Iterator{
```

```
Public:
    Virtual void First() = 0;
    Virtual void Next() = 0;
    Virtual bool IsDone() const = 0;
    Virtual Item CurrentItem() const= 0;
Protected:
    Iterator();
};

Template <class Item>
Class ListIterator: public Iterator<Item>{
Public:
    ListIterator(const List<Item>* aList);
    Virtual void First() = 0;
    Virtual void Next() = 0;
    Virtual bool IsDone() const = 0;
    Virtual Item CurrentItem() const= 0;
Protected:
    Const List<Item>* _list;
    Long _current;
};

Template <class Item>
ListIterator<Item>: ListIterator{
    Const List<Item>* alist;
} : _list(alist), _current(0){
};

Template <class Item>
Void ListIterator<Item>: First(){
    _current= 0;
};

Template <class Item>
Void ListIterator<Item>: Next(){
    _current++;
};

Template <class Item>
Bool ListIterator<Item>:IsDone() const{
    Return _current >= _list->count();
};
```

G. Patrón Observer

"Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualicen automáticamente todos los objetos que dependen de él" (Gamma, et. al., 1994, p293). Para todos los objetos que son heredados por el patrón Observer, las clases que definen los datos de las aplicaciones y las

representaciones pueden reutilizarse de forma independiente, también pueden trabajar juntas, desarrollamos los usos, participantes, estructura y código C++.

Usos del patrón Observer

a. Una abstracción tiene dos aspectos y uno depende del otro. Encapsular estos aspectos en objetos separados permite modificarlos y reutilizarlos de forma independiente.
b. Un cambio en un objeto requiere cambiar otros objetos.
c. Un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quienes deben ser notificados.

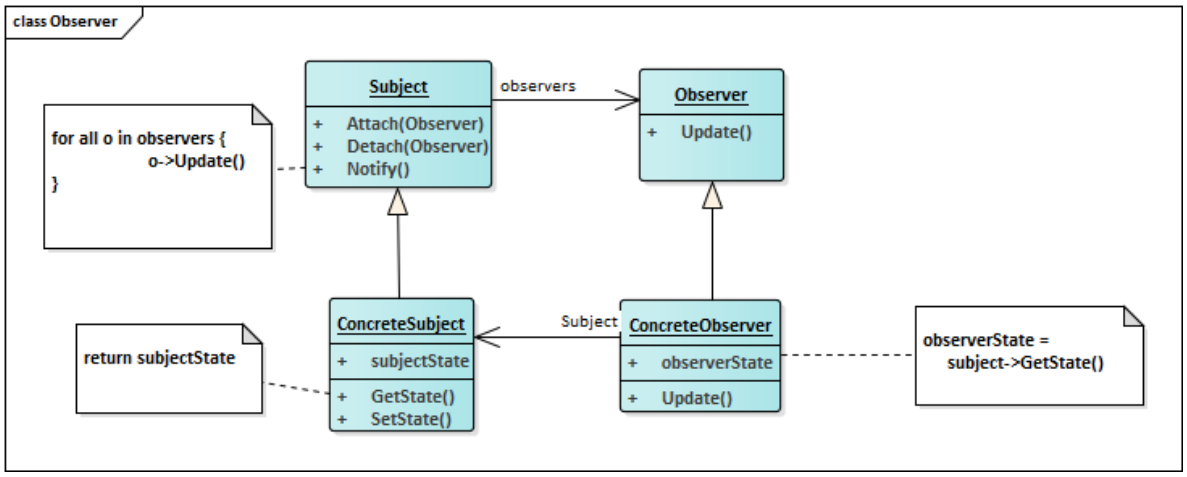


Figura 24. Estructura del patrón observer (Gamma, et. al., 1994, p. 294).

Participantes

Subject. Conoce a sus observadores. Un Subject puede ser observado por cualquier número de objetos Observer.

Observer. Define una interfaz para actualizar los objetos que deben ser notificados por cambios en Subject.

ConcreteSubject. Almacena el estado de interés para los objetos ConcreteObserver. Envia una notificación a sus observadores cuando cambia su estado.

ConcreteObserver. Mantiene una referencia para un objeto ConcreteSubject. Guarda un estado que debería ser consistente con Subject. Implementa la interfaz de actualización de Observer para mantener su estado consistente con Subject.

Código del patrón Observer en C++

```
class Subject;
class observer{
Public:
      Virtual Observer();
      Virtual void Update(Subject * theChangeSubject) = 0;
protected:
      Observer();
};

Class Subject {
      Public:
      Virtual Subject();
      Virtual void Attach(Observer*);
      Virtual void Detach(Observer*)
      Virtual void Notify();

Protected:
      Subject();

Private:
      List<Observer*> *_observers;
};

Void Subject :: Attach (Observer * o){
      _observers -> Append(o);
}
Void Subject :: Detach(Observer * o){
      _observers -> Remove(o);
}
Void Subject :: Notify(Observer * o){
      List<Observer*>  i(_observers);
}
Void Subject :: Notify (){
      ListIterator<Observer*>  i(_observers);

      For (i.First(); !i.IsDone(); i.Next()){
            i.CurrentItem() -> Update (this);
      }
}
```

H. Patrón State

"Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase" (Gamma, et. al., 1994, p305). La idea básica del patrón es introducir una clase abstracta que representa los estados del objeto, declarando una interfaz común para todas las clases que representan diferentes estados, ahora se explica los usos, participantes, estructura y código C++.

Usos del patrón State

a. El comportamiento de un objeto depende de su estado y debe de cambiar en tiempo de ejecución (runtime) dependiendo de su estado.

b. Las operaciones largas tienen sentencias adicionales con múltiples ramas que dependen del estado del objeto.

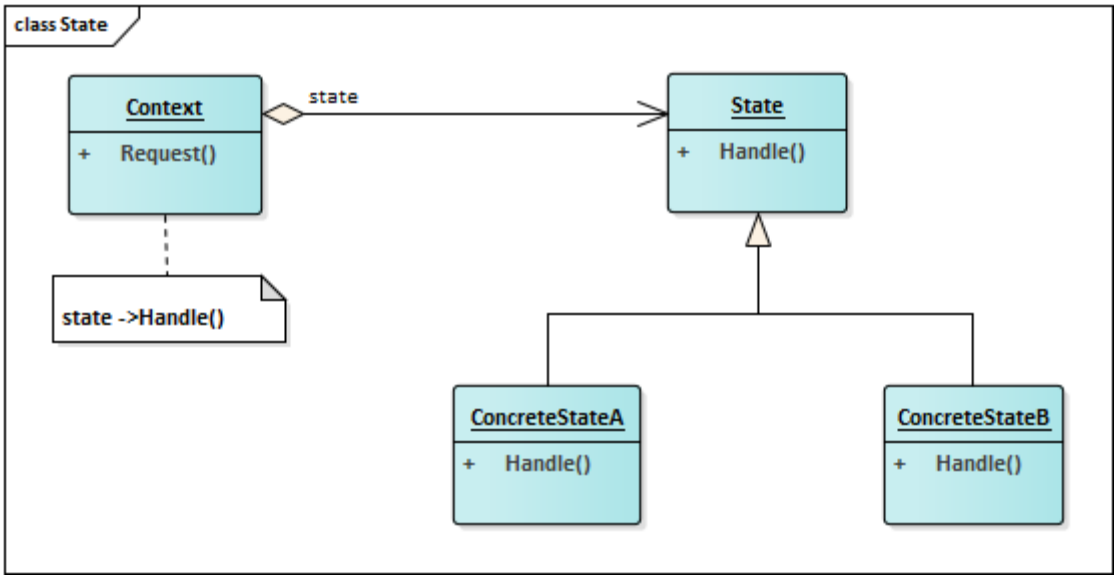


Figura 25. Estructura del patrón state (Gamma, et. al., 1994, p. 306).

Participantes

Context (TCPConnection). Define la interfaz de interés para los clientes. Mantiene una instancia de una subclase ConcreteState que define el estado actual.

State (TCPState). Define una interfaz para encapsular el comportamiento asociado con un determinado estado de Context.

ConcreteState (TCPEstablished, TCPListen, TCPClosed). Cada subclase implementa un comportamiento asociado con un estado de Context.

Código del patrón State en C++

```
class TCPOctetStream;
Class TCPState;

Class TCPConnection {
Public:
    TCPConnection();

    Void ActivateOpen();
    Void PassiveOpen();
    Void Close();
```

```
        Void Send();
        Void Acknowledge();
        Void synchronize();
        Void ProcessOctet (TCPOctetStream*);

Private :
        Friend class TCPState;
        Void ChangeState(TCPState*);

Private:
        TCPState* _state;
}

Class TCPState {
Public:
        Virtual void Transmit (TCPConnection*, TCPOctetStream*);
        Virtual void ActiveOpen (TCPConnection*);
        Virtual void PassiveOpen (TCPConnection*);
        Virtual void Clase (TCPConnection*);
        Virtual void Synchronize (TCPConnection*);
        Virtual void Acknowledge (TCPConnection*);
        Virtual void Send (TCPConnection*);
Protected:
        void ChangeState(TCPConnection*, TCPOctetStream*);
}

TCPConnection::TCPConnection (){
        _state = TCPClosed::Instance();
}

Void TCPConnection::ChangeState (TCPState* s){
        _state = s;
}

Void TCPConnection::activeOpen(){
        _state ->ActiveOpen(this);
}

Void TCPConnection::pasiveOpen(){
        _state ->pasiveOpen(this);
        }

Void TCPConnection::Close(){
        _state ->Close(this);
        }

Void TCPConnection::Acknowledge(){
        _state ->Acknowledge(this);
        }
Void TCPConnection::Synchronize(){
```

```
_state ->Synchronize(this);
}
```

I. Patrón Strategy

"Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan" (Gamma, et. al., 1994, p315). Se cambiara el comportamiento en runtime, siempre que el objeto implemente la interfaz de comportamiento correcto, explicamos usos, participantes, estructura y código C++.

Usos del patrón Strategy

a. Si muchas clases relacionadas difieren solo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre los muchos posibles.
b. Cuando se necesitan distintas variantes de un algoritmo.
c. Si un algoritmo usa datos que los clientes no deberían conocer.
d. Si una clase define muchos comportamientos, y estos se representan como múltiples sentencias condicionales en sus operaciones. En lugar de tener muchos condicionales, podemos mover las ramas de estos a su propia clase.

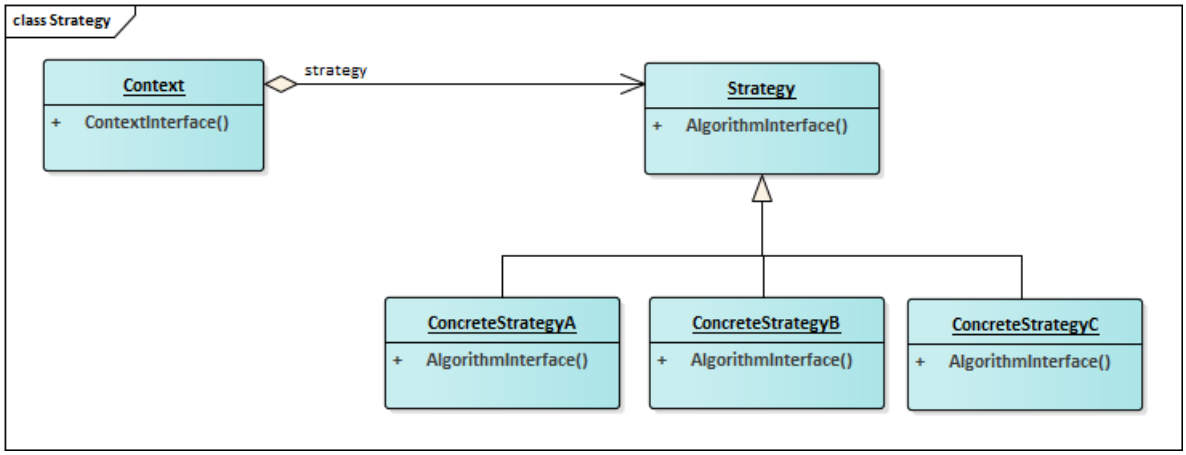


Figura 26. Estructura del patrón strategy (Gamma, et. al., 1994, p. 315).

Participantes

Strategy (Compositor). Declara una interfaz común a todos los algoritmos permitidos. El Context usa esta interfaz para llamar al algoritmo definido por un ConcreteStrategy.

ConcreteStrategy (SimpleCompositor, TexTCompositor, ArrayCompositor). Implementa el

algoritmo usando la interfaz Strategy. Context (Composition). Es configurado con un objeto ConcretStrategy. Mantiene una referencia al objeto Strategy. Puede definir una interfaz que permita a Strategy acceder a sus datos.

Código del patrón Strategy en C++

```
Void Composition::Repair(){
    Switch (_breakingStrategy){
    Case SimpleStrategy:
        ComposeWithSimpleCompositor();
        Breack;
    Case TextStrategy:
        ComposeWithTextCompositor();
        Breack;
    }
}

Void Composition::Repair(){
    _compositor -> Compose();
}

Template <class aStrategy>
Class Context{
    Void Operation() { theStrategy.DoAlgorithm();}
Private:
    aStrategy theStrategy;
}

Class MyStrategy {
Public:
    Void DoAlgorithm();
};

Context <myStrategy> aContext;
```

De acuerdo con la tabla 16, desarrollamos la intervención a la fase de diseño de la metodología agil Iconix, para diseñar el diagrama de secuencia que es el producto software; no se considera el ámbito de aplicación para clases porque el modelado de dominio y diagrama de clases de Iconix aseguran un modelado correcto porque se aplica semántica en su desarrollo. Entonces, consideramos el ámbito de intervención de la fase de diseño con patrones para objetos cuyos propósitos son: de creación, estructurales y de comportamiento, presentamos patrones para el ámbito de objetos que se listan a continuación, esto no quiere decir que debemos usarlos a todos en la fase de diseño, se aplicaran algunos, según los objetivos del software; a continuación mostramos en la tabla

17; las tareas, técnicas, producto software y responsables del diseño.

Tabla 17. Desarrollo de la fase de diseño del software con intervención

Tarea	Técnica	Producto software	Responsable
Crear familias de objetos relacionados	a. Declarar una interfaz para operaciones que crea objetos producto abstracto b. Implementar las operaciones para crear objetos producto concreto c. Declarar una interfaz para un tipo de objeto producto d. Definir un objeto producto para ser creado por AbstractFactory	Patrón abstract factory	a. Arquitecto de software b. Programador
Crear clase con instancia única	a. Definir una operación Instance b. Crear objetos únicos incluido su propia instancia	Patrón Singleton	a. Arquitecto de software b. Programador
Crear clase reutilizable que coopere con clases no relacionadas	a. Definir la interfaz de dominio específica b. Definir una interfaz existente que necesita ser adaptada c. Adaptar la interfaz	Patrón Adapter	a. Arquitecto de software b. Programador
Añadir dinámicamente responsabilidades a un objeto	a. Definir la interfaz de objetos para adicionar responsabilidades b. Definir un objeto para adicionar responsabilidades c. Mantener una referencia a un objeto component d. Definir una interfaz que se ajusta a su interfaz component e. Adicionar responsabilidades al	Patrón Decorator	a. Arquitecto de software b. Programador

	objeto concreto decorator		
Crear una interfaz única para un conjunto de interfaces de subsistemas	a. Definir la clase compiler responsable ante una petición b. Delegar peticiones de clientes a objetos del subsistema c. Definir las clases subsistemas d. Implementar la funcionalidad del subsistema e. Realizar las tareas encargadas por compiler	Patrón Facade	a. Arquitecto de software b. Programador
Crear acceso secuencial a los elementos de un objeto	a. Definir una interfaz de acceso a los elementos b. Implementar la interfaz c. Mantener la posición actual del recorrido d. Definir la interfaz para crear un objeto Iterator e. Implementar la interfaz de creación de Iterator	Patrón Iterator	a. Arquitecto de software b. Programador
Definir dependencia de uno a muchos entre objetos	a. Crear objeto subject visto por observers b. Definir observer que actualiza los objetos por cambios en subject c. Almacenar estado de interés de objetos ConcreteObserver d. Mantener referencia para un objeto ConcreteSubject e. Implementar la interfaz de actualización de Observer	Patrón Observer	a. Arquitecto de software b. Programador
Modificar comportamiento de un objeto	a. Definir la interfaz de interés para los clientes b. Mantener una instancia de una	Patrón State	a. Arquitecto de software b. Programador

cuando cambia su estado interno	subclase c. Definir una interfaz que encapsula el comportamiento de Context		
Definir una familia de algoritmos encapsulados e intercambiables	a. Declarar una interfaz (Strategy) común para todos los algoritmos b. Implementar (ConcreteStrategy) el algoritmo usando la interfaz Strategy c. Configurar (Context) con un objeto ConcretStrategy	Patrón Strategy	a. Arquitecto de software b. Programador

Fuente: Elaboración propia.

La aplicación práctica para un caso de cada patrón de diseño se muestra en el anexo 7.

2.2.3.3. Fase de Implementación con Intervención

Ble (2013) afirma que los cinco principios Solid, uno por cada letra, tratan el diseño orientado a objetos en términos de gestión de dependencias. Las dependencias entre unas clases (objetos) y otras son las que hacen al código más frágil o más robusto y reutilizable, produciendo código de calidad. El problema con el modelado tradicional orientado a objetos, es que no interviene a profundidad en la gestión de dependencias entre clases (objetos) sino se enfoca en la conceptualización de las clases.

Principio de Responsabilidad Única (SRP)

Este principio define que todas las clases de una aplicación deben tener una sola responsabilidad encapsulada totalmente y que esta funcionalidad debe definirse en la clase. Entonces, cada clase debe tener una única razón para modificarse, guardando el control de su complejidad, mejorar su lectura y el trabajo en equipo (Delechamp y Laugie, 2016, p. 96).

Una clase debe tener una, y solo una, razón para cambiar. La idea detrás de este principio es diseñar una clase que tenga una responsabilidad o varios métodos con funcionalidad única. De acuerdo con este principio, un método no debe hacer más que una tarea a la vez.

Cada función debe ser designada como una tarea única (Mainkar, 2017, p. 81).

El principio de responsabilidad única se basa en el principio de cohesión, donde cada módulo o clase debe ser responsable de la funcionalidad de una sola parte proporcionada por el software, y esa responsabilidad debe estar completamente encapsulada por la clase (Chandel, 2018, p. 7).

Pijierro (2017) cuando se incumple el principio SRP, tenemos los siguientes problemas de diseño: (a) Clases que involucran varias capas de arquitectura, como mezclar interfaces graficas con lógica de negocio o a esta última con persistencia. (b) Número alto de métodos públicos, una clase con muchos métodos públicos da una idea de muchas operaciones que realiza o de muchos comportamientos. (c) Número de imports, si una clase necesita muchas clases externas puede significar que tiene muchas responsabilidades y está intentando hacer muchas cosas para las cuales necesita a otras clases. (d) Una nueva funcionalidad afecta siempre a una clase, si introducimos una nueva característica una clase es afectada, entonces esa clase hace muchas cosas relacionadas a muchas características.

"No debería haber más de una razón para cambiar una clase" Martin (2000). Lo que quiere decir este principio es que una clase (objeto) debe tener únicamente una responsabilidad totalmente encapsulada en la clase; principio que indica una alta cohesión en la clase y bajo acoplamiento entre las clases, y código de complejidad mínima para la clase que genera. Lo contrario es que modelamos una clase con más de una responsabilidad, entonces habrá más de una razón para que cambie la clase, si ocurre esto, tendremos que dividir la clase y cada clase se encargue de su responsabilidad, al cumplirse el principio, estamos desarrollando productos software de calidad.

Principio Abierto Cerrado (OCP)

Gómez (2014), indica que en este principio las clases están abiertas a la extensión, pero cerradas a la modificación, significando que, ante peticiones de cambio en nuestro código, hay que añadir funcionalidad sin modificar la existente. Lo más común para aplicar el principio OCP es usar interfaces o clases abstractas de las que dependen implementaciones concretas, así puede cambiar la implementación de la clase concreta mantenido la interfaz intacta.

Delechamp y Laugie (2016) indican que el principio abierto/cerrado enuncia que un programa debe extenderse, pero estar cerrado a la modificación, aplicar el principio ayuda a que los sistemas presenten mejor mantenibilidad y soporten los cambios de requisitos.

Meyer (1997) afirma que debemos tener la capacidad de extender el comportamiento de las clases sin necesidad de modificar su código, esto permite seguir adicionando funcionalidad con la seguridad que no cambie el código existente, nuevas funcionalidades implican adicionar nuevas clases y métodos, pero no se debería modificar el código existente.

García y Pardo (1998) afirman que con el principio abierto/cerrado ante la necesidad de un cambio de requisitos, el diseño de las entidades que existen debe permanecer inalterado y para adicionar comportamiento de las entidades se debe escribir código nuevo, pero nunca cambiar el código existente. Las entidades software deben cumplir dos propiedades primarias: (a) Están abiertas para su extensión; esto significa que el comportamiento de estas entidades software puede ser extendido. (b) Están cerradas para su modificación; significando que el código fuente de la entidad software es inalterable, no puede cambiar el código fuente existente.

Martin y Martin (2006) indican que para aplicar OCP, se deben crear abstracciones con clases bases abstractas e ilimitados comportamientos representados por todas las clases que pueden ser derivadas de ellas; podemos afirmar que un módulo estaría cerrado para la modificación si depende de una abstracción fija que no cambia, pero su comportamiento puede ser extendido creando nuevas clases derivadas de su abstracción. Los patrones de diseño Strategy y Template, satisfacen el principio abierto/cerrado porque representan una separación muy clara entre la funcionalidad y los detalles de implementación de esa funcionalidad.

“Entidades de software (clases, módulos, paquetes, librerías, funciones, etc.) deberían estar abiertas a la extensión pero cerradas a la modificación” (Martin, 2000). Interpretando este principio, se debe cambiar el comportamiento de una clase mediante la implementación de herencia y composición, aplicando este principio se logra acoplamiento menos rígido, mejora la lectura del código, se disminuye el riesgo de malograr una funcionalidad al modificar el código existente.

Principio de Sustitución de Liskov (LSP)

Este principio define una relación tipo/subtipo basada en el comportamiento, un tipo se considera subtipo de otro cuando cualquier instancia del primer tipo puede aparecer en cualquier lugar donde se espera una instancia del segundo (Liskov, 1994).

La importancia del principio LSP es evidente si pensamos en las consecuencias de violarla. Si tenemos una función f con argumento un tipo base B, ahora considera que el tipo base B tiene un subtipo derivado D, que cuando se pasa a f se comporta de forma incorrecta, en este caso podemos asegurar que D no puede reemplazar a B (Akbary, 2001)

La violación del principio LSP pone de manifiesto la importancia del mismo. Supongamos una función que utiliza un puntero a una clase base, pero que no cumple el principio de Liskov, entonces, esta función debe conocer de forma explícita todas las clases derivadas de dicha clase base. Por tanto, esta función violaría el principio abierto/cerrado porque tendría que ser modificada cada vez que se tiene que crear una nueva clase derivada de la clase base (Booch, 1996).

En los lenguajes de programación orientado a objetos conseguir el polimorfismo es mediante la herencia de clases, en esta característica se basa el principio de sustitución de Liskov. El principio indica los fundamentos básicos de diseño que debe seguir la herencia para un caso particular, o como crear jerarquías de herencia entre clases, el hecho que una clase herede de otra, no asegura que se cumple el principio LSP, si no se respetan sus "contratos de diseño" (Martin, 1996).

Jurado (2010) opina que el principio LSP está estrechamente relacionado a la extensibilidad de las clases cuando se realiza mediante herencia o subtipos. Si una función no cumple el LSP, entonces, rompe el OCP, porque para ser capaz de funcionar con subtipo (clase hija) se necesita saber de la clase padre (tipo) y por tanto modificarla.

Delechamp y Laugie (2016) afirma que en el LSP, todo el código que manipule una clase padre (tipo) debe poder usar una de sus clases derivadas sin conocer su uso; la clase base (padre) define un contrato que toda subclase (hija) debe obligatoriamente respetar. El principio reciproco es que las clases derivadas deben poder sustituirse por su clase base (p. 98).

Schmiedehausen (2018) afirma que el LSP es una guía para crear jerarquías de herencia para las clases. Aplicando LSP, el cliente puede usar cualquier clase o subclase sin comprometer el comportamiento esperado, al usar LSP los clientes no tienen conocimiento de los cambios en las jerarquías de clase. Cuando no hay cambios en la interfaz, no debería haber ninguna razón para cambiar el código existente (p. 26)

Leiva (2016) El principio de Liskov nos ayuda a utilizar la herencia de forma correcta y a modelar mejor al extender clases.

“Las funciones que utilicen punteros o referencias a clases base deben ser capaces de usar objetos de clases derivadas de estas sin saberlo” (Martin, 2000). Interpretado, las subclases deben tener comportamiento según contrato cuando son usadas en lugar de sus clases base, las abstracciones ayudan a tener código fácil de reutilizar y jerarquía de clases fácil de entender, una violación de LSP es una violación latente de OCP.

Principio de Segregación de Interfaces (ISP)

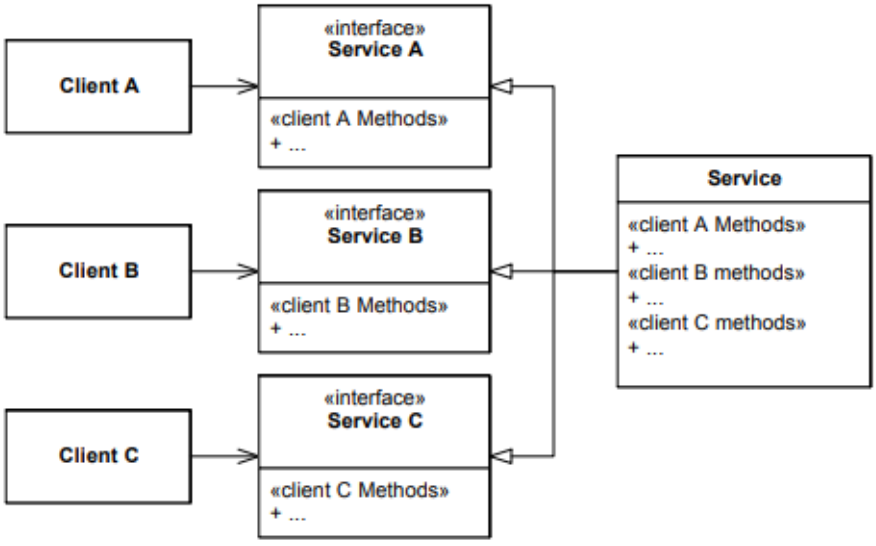


Figura 27. Segregación de interfaces

Este principio es bastante simple. Si existe una clase con varios clientes, en lugar de cargar la clase con todos los métodos que los clientes necesitan, crear interfaces específicas para cada cliente. En la figura 27, los métodos que necesita cada cliente se colocan en interfaces específicas de ese cliente, esas interfaces se heredan de forma múltiple por la clase de servicio y se implementan ahí, si la interfaz para Client A necesita cambiar, Client B y

Client C no se verán afectados, no tendrán que ser recompilados o redistribuidos (Martin, 2000).

Martin (2002) opina sobre el cambio de interfaces lo siguiente. Cuando a las aplicaciones orientadas a objetos se hacen mantenimiento, las interfaces con las clases y componentes existentes cambian, estos cambios pueden tener un gran impacto y obligar a la recopilación y redistribución de una gran parte del diseño. Este impacto se puede disminuir agregando nuevas interfaces a los objetos existentes, en lugar de hacer cambios en la interfaz existente (p. 15).

Detectando la violación del ISP. Si al implementar una interfaz, uno o varios métodos no tienen sentido, los dejas vacíos o haces excepciones, es muy probable que estas violando ISP; si la interfaz forma parte del código, divide en varias interfaces que definan comportamientos más específicos. Lo importante es usar todos los métodos definidos por esas interfaces (Leiva, 2016).

"Los clientes no deberían ser forzados a depender de interfaces que no utilizan" (Martin, 2000). Trellini (2012) opina que varias interfaces especificas a los clientes son mejores que una interfaz de propósito general; resuelve el problema de tener interfaces no cohesivas, ésta interfaz es dividida en grupos de métodos cohesivos, donde cada grupo sirve a un tipo de cliente específico.

Principio de Inversión de Dependencias (DIP)

Martin (2002) afirma que el objetivo del DIP es hacer que el código no dependa de los detalles de implementación, ósea permite organizar el código para que sea flexible ante cualquier modificación. La inversión de dependencias es la estrategia para depender de interfaces o funciones y clases abstractas, en lugar de funciones y clases concretas.

Las dependencias que una clase tiene no deben ser asignadas por ella misma sino por un agente externo, al tener el código desacoplado nos permite cambiar las dependencias en tiempo de ejecución en base a cualquier factor que consideremos, para ello necesitamos un inyector o contenedor que sería el encargado de inyectar las dependencias correctas en el momento necesario. La inyección de dependencia permite centralizar la creación de los objetos y agrupar la responsabilidad de crearlos en las acciones (Álvarez, 2012).

El DIP está basado en reducir las dependencias entre los módulos del código. Este principio será de gran ayuda para entender cómo acoplar correctamente sistemas. Si los detalles de la implementación dependen de altos niveles de abstracciones, ayudará a conseguir un sistema bien acoplado, también influirá en la encapsulación y cohesión del sistema (Karam, 2017).

"A. Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones, B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones" (Martin, 2002). El código es robusto, flexible y reutilizable cuando depende de abstracciones, entonces, utilizar interfaces y clases abstractas, no clases concretas, exponer por constructores o parámetros las dependencias que una clase pueda tener; el objetivo del DIP es atacar el alto acoplamiento, mejorar la cohesión y encapsular adecuadamente los métodos de las clases concretas mediante el código donde cada dependencia debe apuntar a una abstracción, este principio no debe usarse indistintamente en todas las clases, porque haría el código más complejo y poco mantenible.

Tabla 18. Desarrollo de la fase de implementación del software con intervención

Tarea	**Técnica**	**Producto software**	**Responsable**
Aplicar SRP	a. Identificar clases con responsabilidad única b. Identificar clases con más de una responsabilidad c. Crear clases con responsabilidad única d. Generar código para las clases con responsabilidad única	Código para clases con responsabilidad única	Analista Programador
Aplicar OCP	a. Crear clases abstractas cuando se necesita b. Identificar clases que presentan herencia y	Código para clases abierto y cerrado	Analista Programador

	composición c. Generar código para clases con herencia y composición d. No modificar código para adicionar funcionalidad e. Aplicar patrones strategy y template si es necesario f. Escribir código nuevo para adicionar funcionalidad		
Aplicar LSP	a. Identificar clases abstractas (tipo) b. Identificar clases concretas (subtipo) c. Modelar herencia entre clases d. Definir contratos de diseño de la clase base (tipo) e. Escribir código para implementar la herencia	Código con principios Liskov	Analista Programador
Aplicar LSP	a. Identificar clases con varios clientes b. Crear interfaces específicas para cada cliente c. Agregar nuevas interfaces para nuevos requisitos d. Aplicar la separación por delegación con el patrón adapter cuando es necesario	Código con principio de segregación de interfaces	Analista Programador

	e. Aplicar la separación por herencia múltiple cuando es necesario f. Escribir código para implementar los métodos específicos		
Aplicar DIP	a. Identificar clases concretas para crear clases abstractas b. Analizar las dependencias entre módulos de código c. Analizar dependencias para inyección por constructor d. Analizar dependencias para inyección por interfaces e. Analizar dependencias para inyección por método f. Generar código con inyección de dependencias	Código con principio de inversión de dependencias	Analista Programador

Fuente: Elaboración propia.

2.3. MARCO LEGAL

Según la R. M. N° 041-2017-PCM, se aprueba la "NTP ISO/IEC 12207:2016, Tecnología de la información: Procesos del ciclo de vida del software", norma técnica de cumplimiento obligatorio de las todas las Instituciones públicas, respecto a los cinco procesos principales del ciclo de vida del software, específicamente el tercer proceso de desarrollo, se lista las actividades y tareas que son: análisis de requerimientos, diseño, codificación (implementación) integración, pruebas e instalación y aceptación relacionados con los productos software. La lista de actividades para este proceso son: Implementación del proceso, análisis de los requerimientos del sistema, diseño de la arquitectura del

sistema, análisis de los requerimientos software, diseño de la arquitectura del software, diseño detallado del software, codificación y pruebas del software, integración del software, pruebas de calificación del software, integración del sistema, pruebas de calificación del sistema, instalación del software, apoyo a la aceptación del software. El proceso de apoyo, tiene el sub proceso, aseguramiento de la calidad que define las actividades para asegurar que los productos software y los procesos son conformes a sus requerimeintos especificados (PCM, 2017).

La Presidencia del Consejo de Ministros (PCM) emite el D.S. N° 004-2103-PCM, aprueba la Política nacional de modernización de gestión pública al 2021, estableciendo los lineamientos para las entidades públicas en general, lineamiento 7 del gobierno electrónico, que indica. Promover el gobierno electrónico a través del uso intensivo de las tecnologías de información y comunicación (TIC) como soporte a los procesos de planificación, producción y gestión de las entidades públicas permitiendo a su vez consolidar propuestas de gobierno abierto, mediante: (a) Facilitar el acceso de los ciudadanos a servicios públicos en línea, organizados en forma sencilla, cercana y consistente. (b) Dar a los ciudadanos acceso a información permanentemente actualizada sobre la entidad. (c) Integrar, en lo posible, los sistemas de comunicación de la entidad a las plataformas nacionales de gobierno electrónico, en concordancia con el Plan Nacional de Gobierno Electrónico (PCM, 2013).

Mediante el D. S. N° 066-2011-PCM, se aprueba el "Plan de desarrollo de la sociedad de la información en el Perú, la agenda digital peruana 2.0", que en el objetivo 4 dice "Impulsar la investigación científica, el desarrollo tecnológico y la innovación con base en las prioridades nacionales", y objetivo 5 "Incrementar la productividad y competitividad a través de la innovación en la producción de bienes y servicios, con el desarrollo y aplicación de las TIC" (PCM, 2011).

El D. S. N° 054-2011-PCM, aprueba el Plan Bicentenario: El Perú hacia el 2021, el plan menciona el avance de las tecnologías de las comunicaciones y los nuevos inventos que abren un panorama promisorio para la humanidad. El World Economic Forum 2010 presenta el índice de conectividad, y propone en uno de sus pilares la preparación individual, siendo la escasa productividad laboral promedio, que es diez veces inferior al

valor medio de la región, de las economías desarrolladas según la OIT, obedece al limitado desarrollo de la ciencia, la tecnología y la innovación productiva. El indicador más representativo de este atraso en el Perú es el escaso número de patentes otorgadas a sus residentes, apenas quince frente a más de cien en países como Argentina y México (PCM, 2011).

Ley N° 30224, del Sistema Nacional para la Calidad (SNC) y el Instituto Nacional de Calidad (INACAL), cuyo fin es promover y asegurar el cumplimiento de la Política Nacional para la Calidad, con miras al desarrollo y la competitividad de las actividades económicas y la protección del consumidor, siendo uno de sus objetivos: orientar y articular las actividades de normalización, acreditación, metrología y evaluación de la conformidad, acorde con normas, estándares y códigos internacionales reconocidos mundialmente por convenios y tratados de los que el Perú es parte, particularmente en calidad del producto software se tiene las normas (Poder Ejecutivo, 2014).

2.4. MARCO FILOSÓFICO

La UNESCO b. (2017) ha trabajado con los Estados Miembros y otros interesados, cuatro campos distintos entre la política y la práctica de Internet, estos campos son: acceso a la información y al conocimiento, libertad de expresión, privacidad y normas de comportamientos éticos en línea. Estos cuatro campos son las piedras angulares necesarias para construir una Internet global libre y confiable que posibilite sociedades del conocimiento inclusivas. La metáfora de "piedra angular" se refiere al elemento arquitectónico que se coloca en el centro de un arco para que las otras piedras permanezcan en su lugar. La metáfora se usa para transmitir la importancia de esas cuatro dimensiones para construir el Internet global. Finalmente, la ética considera si las normas, reglas y procedimientos que rigen el comportamiento en línea, como el diseño de Internet y los medios digitales afines, están basados en principios éticos, sujetos a los derechos humanos y dirigidos a la protección de la dignidad y seguridad de individuos en el ciberespacio, mejorando la accesibilidad, apertura e inclusión en Internet.

ACM/IEEE (1999), crea y difunde el código de ética y práctica profesional de ingeniería de software, donde los ingenieros de software deben comprometerse a realizar del análisis, la especificación, el diseño, el desarrollo, la prueba y el mantenimiento del software, una profesión benéfica y respetada. Según el compromiso con la salud, seguridad y bienestar

del usuario, los ingenieros de software tienen que adherirse a los ocho principios siguientes: (a) Público: los ingenieros de software deben actuar consecuentemente con el interés del usuario. (b) Cliente y empleador: los ingenieros de software tienen que comportarse de tal forma que fomente el mejor interés para su cliente y empleador, en coherencia con el interés del usuario. (c) Producto: los ingenieros de software deben garantizar que sus productos y modificaciones relacionadas satisfagan los estándares profesionales más altos posibles. (d) Juicio: los ingenieros de software tienen que mantener integridad e independencia en su juicio profesional. (e) Gestión: los administradores y líderes en la ingeniería de software deben suscribir y promover un enfoque ético a la gestión del desarrollo y el mantenimiento del software. (f) Profesión: los ingenieros de software tienen que fomentar la integridad y la reputación de la profesión consecuente con el interés público. (g) Colegas: los ingenieros de software deben ser justos con sus colegas y apoyarlos. (h) Uno mismo: los ingenieros de software tienen que intervenir en el aprendizaje para toda la vida, en cuanto a la práctica de su profesión, y promover un enfoque ético.

Leiva y Villalobos (2015) opinan que las metodologías ágiles mejoran la flexibilidad del desarrollo y la productividad, brindando métodos que se adaptan a los cambios y que se aprenden de la experiencia, en el caso de entornos móviles existen: procesos iterativos e incrementales, desarrollo conducido por pruebas, procesos adaptativos, tratar con el cliente continuamente, desarrolladores altamente calificados, asegurar la calidad, revisiones continuas del proceso y priorización de los requerimientos. En conclusión, la filosofía de los métodos ágiles proveen: mayor valor al individuo, colaboración con el cliente, desarrollo incremental con iteraciones muy pequeñas; el enfoque ágil está mostrando su efectividad en proyectos con requisitos muy cambiantes y cuando se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad.

III. MÉTODO

3.1. TIPO Y NIVEL DE INVESTIGACIÓN

3.1.1. Tipo de Investigación

Es un estudio prospectivo con intervención del investigador, porque se interviene la variable independiente metodología ágil Iconix, con el objetivo de mejorar el grado de calidad de productos software desarrollado con Iconix.

Es un estudio longitudinal porque al intervenir la variable independiente metodología ágil Iconix sin y con intervención, medimos la calidad del producto software que es la variable dependiente.

Es un estudio analítico, porque tenemos la variable independiente metodología ágil Iconix (manipulado), y la variable dependiente calidad de producto software (no manipulado), además, se plantea hipótesis de investigación.

3.1.2. Nivel de Investigación

Es un estudio de nivel aplicativo, porque al intervenir la variable independiente metodología ágil Iconix, esperamos mejorar el grado de calidad del producto software. Entonces, los desarrolladores de software, podrán utilizar las tablas de la metodologia intervenida, con el objetivo de mejorar el grado de calidad del producto software.

3.2. DISEÑO DE LA INVESTIGACIÓN

Se interviene las tareas que generan productos software mediante la metodología ágil Iconix, estas tareas finales se interviene mediante técnicas formales de ingeniería de software, con el objetivo de mejorar el grado de calidad del producto software; se realizará una encuesta antes y otra después de la intervención, a expertos en desarrollo ágil de software, luego se contrasta las hipótesis con técnicas de estadística inferencial, para concluir si las intervenciones mejoran o no, el grado de calidad del producto software. Por tanto, el diseño es cuasi experimental.

3.3. VARIABLES

Las variables de investigación son; la metodología ágil Iconix, intervenida con

técnicas formales de ingenieria de software, a fin de mejorar la calidad del producto software.

Tabla 19. Variables de investigación

Variable independiente	Metodologia agil Iconix (X)
Variable dependiente	Calidad del producto software (Y)

3.3.1. Variable independiente: Metodologia agil Iconix (X)

Iconix es una metodología ágil, que aplica el desarrollo incremental e iterativo como principios de agilidad, tiene las fases de: análisis de requisitos, diseño, implementación y pruebas; se genera productos software en cada fase y usa la notación UML, presenta una parte dinámica y otra estática, el modelo estático se incrementa y es refinado por el modelo dinámico.

3.3.2. Variable dependiente: Calidad del producto software (Y)

Calidad de un producto software es el "grado en que un conjunto de características inherentes cumple con los requisitos" siendo las visiones. (a) La visión del usuario; percibe la calidad como idoneidad para un propósito, al evaluar la calidad de un producto, uno debe hacerse la pregunta ¿El producto software satisface las necesidades y expectativas del usuario?. (b) La visión de desarrollo; aquí se entiende por calidad la conformidad con la especificación; el nivel de calidad de un producto software está determinado por la medida en que el producto cumple con sus especificaciones. (c) La visión del producto software; la calidad se considera vinculada a las características inherentes al producto, es decir, las cualidades internas, determinan sus cualidades externas.

3.4. POBLACIÓN Y MUESTRA

3.4.1. POBLACIÓN

Esta compuesta por todos los productos software de análisis de requisitos, diseño e implementación de la metodología ágil Iconix, con y sin intervención.

3.4.2. MUESTRA

La muestra se ha tomado por juicio de expertos, donde los productos software son

los siguientes: requisitos funcionales y no funcionales, lista de casos de uso, descripción de casos de uso, interfaz gráfica de usuario, arquitectura técnica (diagrama de despliegue y de componentes), diagrama de clases, técnicas de codificación, código fuente, pruebas unitarias, reporte de pruebas unitarias, reporte de pruebas de integración, casos de pruebas, procedimiento de instalación, a los cuales se evalua la calidad del producto software desarrollado mediante la metodología ágil Iconix, con y sin intervención.

3.5. OPERACIONALIZACIÓN DE LAS VARIABLES

Tabla 20. Operacionalización de variables

Variable	Dimensión	Indicador
Metodología ágil Iconix	Análisis de requisitos	Productos software de análisis de requisitos
	Diseño	Productos software de diseño
	Implementación	Productos software de implementación
Calidad del producto software	Calidad interna	Métricas internas
	Calidad externa	Métricas externas

Fuente: Elaboración propia.

3.6. TÉCNICAS E INSTRUMENTOS DE INVESTIGACIÓN

3.6.1. Técnicas

Encuesta.- Se usará la técnica de encuesta, para recolectar información de los expertos en procesos agiles para desarrollar software, con la finalidad de evaluar, cuál de los procesos Iconix, con o sin intervención, presenta mayor grado de calidad del producto software.

Análisis documental.- Se usará esta técnica, para levantar información sobre métricas según las Normas NTP ISO/IEC 9126 y sus extensiones, para evaluar el valor de calidad de un atributo de los productos software.

Análisis Estadístico.- Las mediciones que se obtendrán de los atributos de calidad del producto software, según la escala de Likert, será mediante las técnicas de diferencia de medias relacionadas con la prueba T – Student, prueba de homocedasticidad de Levine y,

análisis de varianza, que nos permitirá definir cuál de los procesos Iconix, con o sin intervención, genera mayor grado de calidad del producto software.

3.6.2. Instrumentos

Los instrumentos para evaluar la calidad del producto software, están de acuerdo a la metodología ágil Iconix con o sin intervención, y el modelo de calidad adaptado por Indecopi, Norma Técnica Peruana ISO/IEC 9126 y sus extensiones. Los instrumentos están desarrollados para recolectar datos mediante un cuestionario formulado con la escala de Likert cualitativa ordinal de cinco opciones, aplicado a diez expertos en desarrollo de software según los valores y principios de agilidad.

3.6.3. Confiabilidad del Instrumento

Los datos generados con el instrumento de recolección de datos utilizado para evaluación de la "Metodología ágil iconix con y sin intervención", se somete a la prueba Alfa de Cronbach que determina la consistencia interna de la escala de Likert, que contextualiza un conjunto de variables politómicas que estarán correlacionadas entre sí, dado que miden los aspectos de la metodología de interés y los articula formando un sistema de medición estable que a partir de sus aspectos específicos permite contextualizar la propiedad aditiva de los ítems que articula la "Metodología agil Iconix con y sin intervención". La confiabilidad se muestra en el anexo 4.

3.6.4. Validación del Instrumento

Los instrumentos para la toma de datos, se valoran a través de la validez de contenido y la validez de criterio, en el primer caso se consulta el juicio de expertos para determinar si el instrumento diseñado para evaluar la metodología Iconix con y sin intervención, que abarca los aspectos de la norma NTP ISO/IEC 9126 y sus extensiones, permite ver funcionalidad, confiabilidad, usabilidad, eficiencia, mantenibilidad y portatilidad, para determinar la calidad de los productos software de la metodología en cuestión. Asimismo, para el segundo caso se aplica la validez de criterio mediante la correlación de la métrica interna y la métrica externa, dado que, la primera métrica proporciona información de los aspectos técnico y funcionalidades de la metodología Iconix con y sin intervención, proporcionada por los profesionales que desarrollan las fases de Iconix y la otra métrica proporciona información de los aspectos técnicos y funcionalidad desde el punto de vista del usuario. La validación se muestra en el anexo 3.

3.7. PROCEDIMIENTOS

3.7.1. Estrategia de Prueba de Hipótesis

Para lo cual, formularemos las hipótesis nulas (Ho), donde las diversas proporciones asociadas a las características y sub características en evaluación son iguales o diferentes y, la hipótesis alterna (Ha) donde por lo menos una de ellas es diferente, o no mejora la calidad del producto software. Luego se hace el análisis estadístico que está determinado por la diferencia de medias relacionadas a través de la prueba T – Student para determinar ¿Cuál de los tratamientos puestos a prueba es mejor? y las pruebas homocedasticidad con la prueba de Levine, asi como, el análisis de varianza de las puntuaciones para cada fase de desarrollo de la Metodología Ágil Iconix con y sin intervención.

3.7.2. Técnicas de Procesamiento de Datos

Los datos registrados usando los instrumentos para la toma de datos se muestran en el anexo 2, en una primera etapa son analizados para determinar la calidad de la respuesta esperada y de registro. Asimismo, se procede a codificar la data para ser trasladada a una matriz de datos del software estadístico SPSS, y en una segunda etapa, se selecciona del menú del software las técnicas estadísticas para calcular los estadígrafos descriptivos y, realizar las pruebas de inferencia estadística que suministren información para contrastar las hipótesis planteadas en el estudio.

3.7.3. Diseño Estadístico

El diseño estadístico está basado en la metodología cuasi experimental, con pre test y post test, con un solo grupo, cuyas unidades experimentales están definidas previamente por los productos software desarrollados con la Metodología Ágil Iconix con y sin intervención.

Tabla 21. Diseño estadístico cuasi experimental

Esquema: O_1 X O_2	
O = Observación	Medición o métrica, entre ambas medidas se aplica el tratamiento cuya eficacia se está investigando

O_1 = Primera medida (variable dependiente)	Calidad del producto software determinada con la Metodología Ágil Iconix sin intervención.
O_2 = Segunda medida (variable dependiente)	Calidad del producto software determinada con la Metodología Ágil Iconix con intervención
X= Tratamiento o intervención (variable independiente)	Aplicación de la Metodología Ágil Iconix con intervención

Fuente: Elaboración propia.

El análisis estadístico está determinado por la diferencia de medias relacionadas a través de la prueba T – Student para. ¿Cuál de los tratamientos puestos a prueba es mejor? y el análisis de varianza de las puntuaciones para cada fase de desarrollo de la Metodología Ágil Iconix con y sin intervención.

3.7.4. Técnicas de Análisis e Interpretación de la Información

La información obtenida usando los instrumentos para la toma de datos que se muestran en el anexo 2, es sistematizada aplicando la combinación del método inductivo y método deductivo, dado que se tiene que articular las tareas de la metodología ágil Iconix con y sin intervención para determinar la calidad del producto software. Asimismo, se aplica el método analítico y sintético para determinar las bondades de la intervención al: análisis de requisitos, diseño e implementación, mediante la calidad interna y calidad externa, medido por atributos del producto software para la metodología ágil iconix con y sin intervención, estableciendo los aspectos convergentes y divergentes. En forma transversal, se aplica las técnicas de análisis estadístico para administrar la información suficiente que permita probar las hipótesis de estudio y vincularlos a la realidad o dinámica durante el desarrollo de software.

IV. RESULTADOS

4.1. HIPÓTESIS ESTADÍSTICAS

Los siguientes párrafos que se describen a continuación son los argumentos para contrastar las hipótesis de estudio formuladas y en el que se aplica el método inductivo para realizar las inferencias lógicas a fin de sintetizar la información. Los cálculos que se presentan para contrastar las hipótesis de estudio están de acuerdo a las hipótesis estadísticas que se formulan a continuación.

4.1.1. Primera Hipótesis Específica

Ho: Si modificamos la fase de análisis de requisitos mediante la inclusión de ingeniería de requisitos, entonces no se mejora la calidad del producto software.

Ha: Si modificamos la fase de análisis de requisitos mediante la inclusión de ingeniería de requisitos, entonces se mejora la calidad del producto software.

4.1.2. Segunda Hipótesis Específica

Ho: Si variamos la fase de diseño mediante la inclusión de patrones de diseño, entonces no se mejora la calidad del producto software.

Ha: Si variamos la fase de diseño mediante la inclusión de patrones de diseño, entonces se mejora la calidad del producto software.

4.1.3. Tercera Hipótesis Específica

Ho: Si transformamos la fase de implementación mediante la inclusión de técnicas de programación orientada a objetos, entonces no se mejora la calidad del producto software.

Ha: Si transformamos la fase de implementación mediante la inclusión de técnicas de programación orientada a objetos, entonces se mejora la calidad del producto software.

4.1.4. Hipótesis General

Ho: Si adaptamos la metodología ágil Iconix con la inclusión de técnicas de ingeniería de software, entonces no se mejora la calidad del producto software, Lima, 2017.

Ha: Si adaptamos la metodología ágil Iconix con la inclusión de técnicas de

ingeniería de software, entonces se mejora la calidad del producto software, Lima, 2017.

4.2. ANÁLISIS E INTERPRETACIÓN DE DATOS

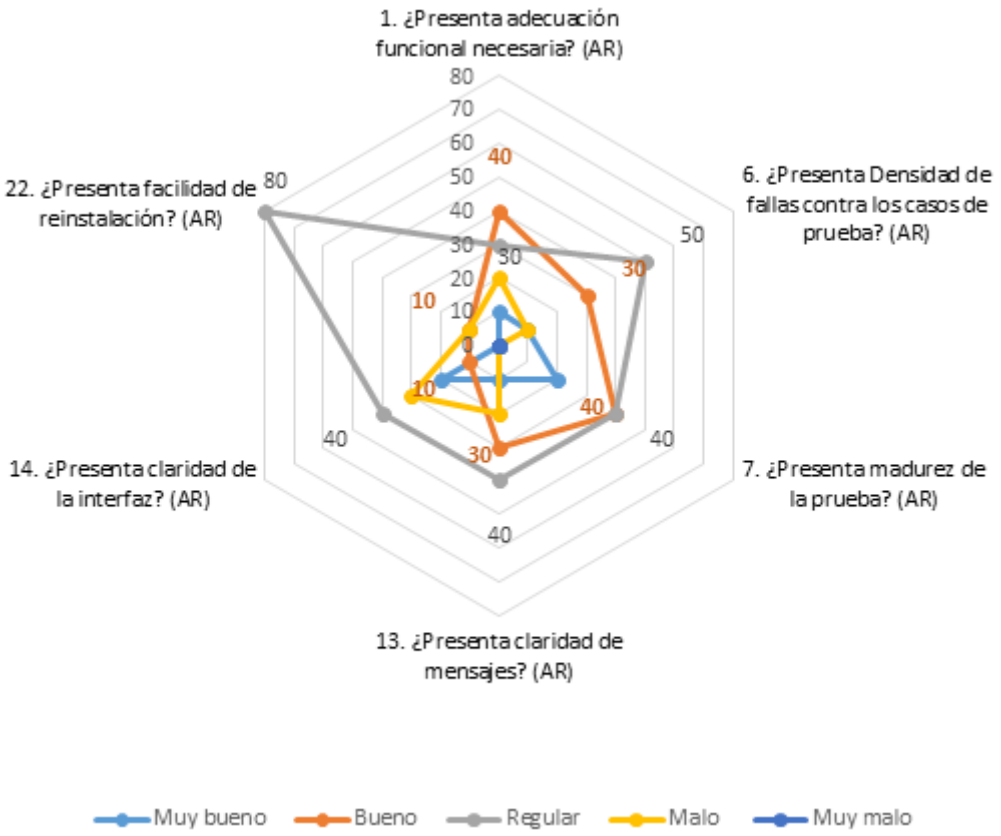
En el presente estudio se establece la forma de adaptar la metodología Ágil Iconix para mejorar la calidad del producto software, con la aplicación del diseño cuasi experimental, con pre test y post test, con un solo grupo, cuyas unidades experimentales están definidas previamente por los productos software desarrollados con la Metodología Ágil Iconix con y sin intervención; la colección de las unidades de análisis forman la muestra intencional que está compuesta por los productos software que están identificados por las tareas, artefactos, técnicas y responsables del proceso Ágil Iconix con y sin intervención, según las fases definidas para: análisis de requisitos, diseño e implementación.

En los párrafos que se describen a continuación se presentan los argumentos para la contratación de las hipótesis de estudio, a través del análisis exploratorio de los datos y de las pruebas de hipótesis correspondientes que permiten establecer la relación de causa – efecto que existe entre la metodología ágil Iconix con la calidad del producto software. En el acápite siguiente describimos las fases de la Metodología Ágil Iconix sin intervención.

Tabla 22. Distribución porcentual del Pre Test de la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix sin intervención

ANALISIS DE REQUISITOS	Muy bueno	Bueno	Regular	Malo	Muy malo
1. ¿Presenta adecuación funcional necesaria? (AR)	10	40	30	20	0
6. ¿Presenta Densidad de fallas contra los casos de prueba? (AR)	10	30	50	10	0
7. ¿Presenta madurez de la prueba? (AR)	20	40	40	0	0
13. ¿Presenta claridad de mensajes? (AR)	10	30	40	20	0
14. ¿Presenta claridad de la interfaz? (AR)	20	10	40	30	0
22. ¿Presenta facilidad de reinstalación? (AR)	0	10	80	10	0

Figura 28. Distribución porcentual del Pre Test de la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix sin intervención.



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Comentario

En la tabla 22 y figura 28, se observa que la distribución de las puntuaciones asignadas por los Ingenieros de Software que evaluaron la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix sin intervención, es más frecuente para la modalidad de la escala "regular", que resalta que un 30% de las puntuaciones califican a la *Presentación adecuada funcional necesaria* como regular para los requisitos funcionales y no funcionales, para la descripción de casos de uso, para la interfaz gráfica de usuario (GUI) y para el código fuente; asimismo se tiene que, el 50% de las puntuaciones califican a la *Presencia densidad de fallas contra los casos de prueba* de regular que abarca los aspectos de reporte de pruebas unitarias y reporte de pruebas de integración; un 40% de las puntuaciones califican a la *Presencia de madurez de la prueba* como regular para los aspectos de reporte de pruebas unitarias y reporte de pruebas de integración; análogamente, el 40% de las puntuaciones califican como regular la *Presencia*

de claridad de mensajes que evalúa los productos: descripción de casos de uso, el código fuente y la interfaz gráfica de usuario (GUI); otro 40% de las puntuaciones califican como regular la *Presencia de claridad de la interfaz* que también mide a los productos software que abarca la descripción de casos de uso, el código fuente y la interfaz gráfica de usuario (GUI) y el 80% de las puntuaciones califican como regular la *Presentación de facilidad del procedimiento de reinstalación*.

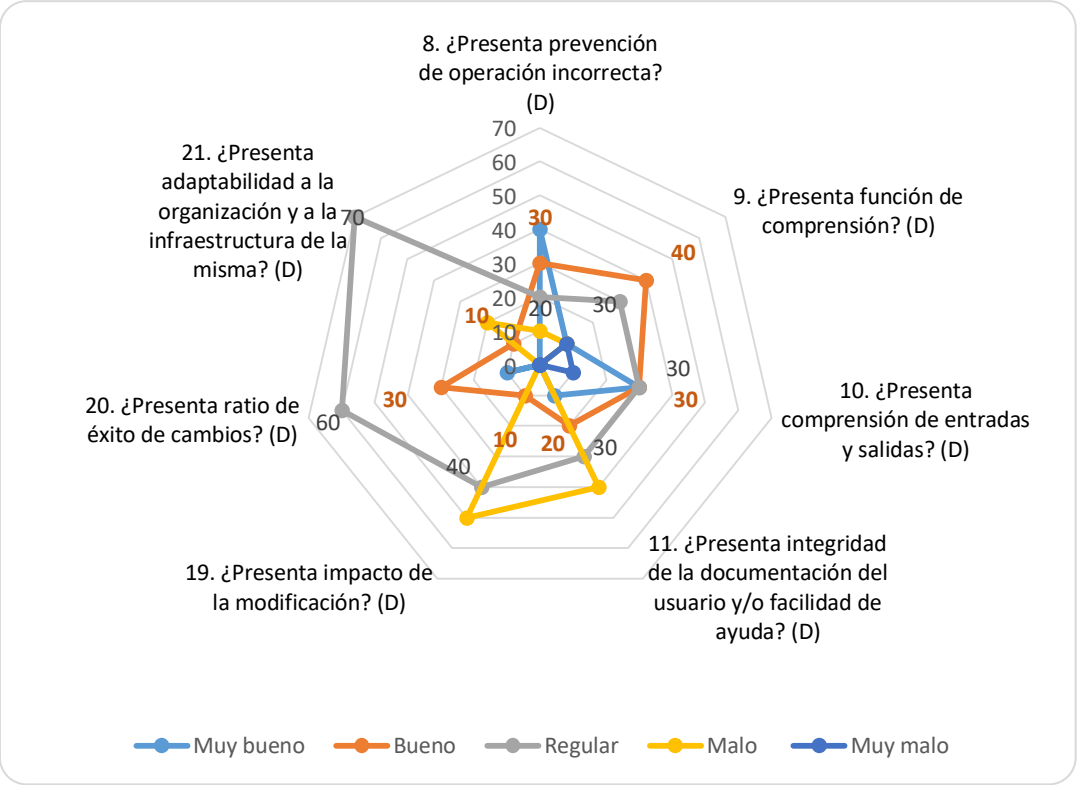
El otro grupo que sigue en importancia, definido por las puntuaciones asociadas a la escala valorativa de bueno para la evaluación de la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix sin intervención, cuya distribución de porcentajes son: Un 40% de las puntuaciones califican a la *Presentación adecuada funcional necesaria* como buena para los requisitos funcionales y no funcionales, para la descripción de casos de uso, para la interfaz gráfica de usuario (GUI) y para el código fuente; asimismo se tiene que, el 30% de las puntuaciones califican a la *Presencia Densidad de fallas contra los casos de prueba* con atributo de bueno que abarca los aspectos de reporte de pruebas unitarias y reporte de pruebas de integración; un 40% de las puntuaciones califican a la *Presencia de madurez de la prueba* como buena para los aspectos de reporte de pruebas unitarias y reporte de pruebas de integración; análogamente, el 30% de las puntuaciones califican como buena la *Presencia de claridad de mensajes* que evalúa los productos: descripción de casos de uso, el código fuente y la interfaz gráfica de usuario (GUI); otro 10% de las puntuaciones califican como bueno la *Presencia de claridad de la interfaz* que también mide a los productos software que abarca la descripción de casos de uso, el código fuente y la interfaz gráfica de usuario (GUI) y el 10% de las puntuaciones califican como bueno la *Presentación de facilidad del procedimiento de reinstalación*.

El análisis conjunto de las trayectorias poligonales observadas en la figura 28, determinan que la modalidad de la escala de medición de regular es la más importante en el análisis de requisito, luego sigue en importancia la escala de bueno y las demás modalidades de la escala de medición son de menor importancia, pero permite visualizar que los Ingenieros evaluadores según su criterio, prescriben que la calidad de los productos software en función del análisis de requisitos de la Metodología Ágil Iconix sin intervención es regular tanto para la métrica interna y para la métrica externa que son dos puntos de referencia vistos desde la fabricación y desde la perspectiva del usuario.

Tabla 23. Distribución porcentual del Pre Test de la Calidad del Producto Software en función del Diseño de la Metodología Ágil Iconix sin intervención

DISEÑO	Muy bueno	Bueno	Regular	Malo	Muy malo
8. ¿Presenta prevención de operación incorrecta? (D)	40	30	20	10	0
9. ¿Presenta función de comprensión? (D)	10	40	30	10	10
10. ¿Presenta comprensión de entradas y salidas? (D)	30	30	30	0	10
11. ¿Presenta integridad de la documentación del usuario y/o facilidad de ayuda? (D)	10	20	30	40	0
19. ¿Presenta impacto de la modificación? (D)	0	10	40	50	0
20. ¿Presenta ratio de éxito de cambios? (D)	10	30	60	0	0
21. ¿Presenta adaptabilidad a la organización y a la infraestructura de la misma? (D)	0	10	70	20	0

Figura 29. Distribución porcentual del Pre Test de la Calidad del Producto Software en función del Diseño de la Metodología Ágil Iconix sin intervención



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Comentario

En la tabla 23 y figura 29, se observa que la distribución de las puntuaciones asignadas por los Ingenieros de Software que evaluaron la Calidad del Producto Software en función del Diseño de la Metodología Ágil Iconix sin intervención, es más frecuente para una mixtura de las modalidades de la escala que resalta las bondades y deficiencias del diseño de los productos software que se describen a continuación: En forma acumulada el 70% de los Ingenieros de software entrevistados califican a la *Presentación de prevención de operación incorrecta*, como buena o muy buena, en su tolerancia a fallas para los productos: requisitos funcionales y no funcionales; así como, los reportes de pruebas unitarias; por otra parte se tiene que, el 50% de los entrevistados califican a *Presenta función de comprensión* como bueno o muy bueno, para la entendibilidad de los productos: requisitos funcionales y no funcionales, descripción de casos de uso y interfaz gráfica de usuario (GUI); también se tiene, al 60% de los ingenieros entrevistados que califican a *Presenta comprensión de entradas y salidas* como bueno o muy bueno, en su entendibilidad de los productos software: requisitos funcionales y no funcionales, descripción de casos de uso e interfaz gráfica de usuario (GUI); además, el 40% de los

entrevistados califica a *Presenta integridad de la documentación del usuario y/o facilidad de ayuda* como malo en su facilidad de aprendizaje de los productos software: requisitos funcionales y no funcionales, lista de casos de uso y descripción de casos de uso; el 50% de los ingenieros entrevistados califican a *Presenta impacto de la modificación* como malo, en lo que corresponde a la estabilidad de los producto software: Diagrama de clases y Código fuente; análogamente se tiene que, el 60% de los entrevistados califican a *Presenta ratio de éxito de cambios* como regular en su estabilidad de los productos software: diagrama de clases y código fuente y el 70% de ingenieros califica a *Presenta adaptabilidad a la organización y a la infraestructura de la misma* como regular en su adaptabilidad del producto: arquitectura técnica (diagrama de despliegue y de componentes).

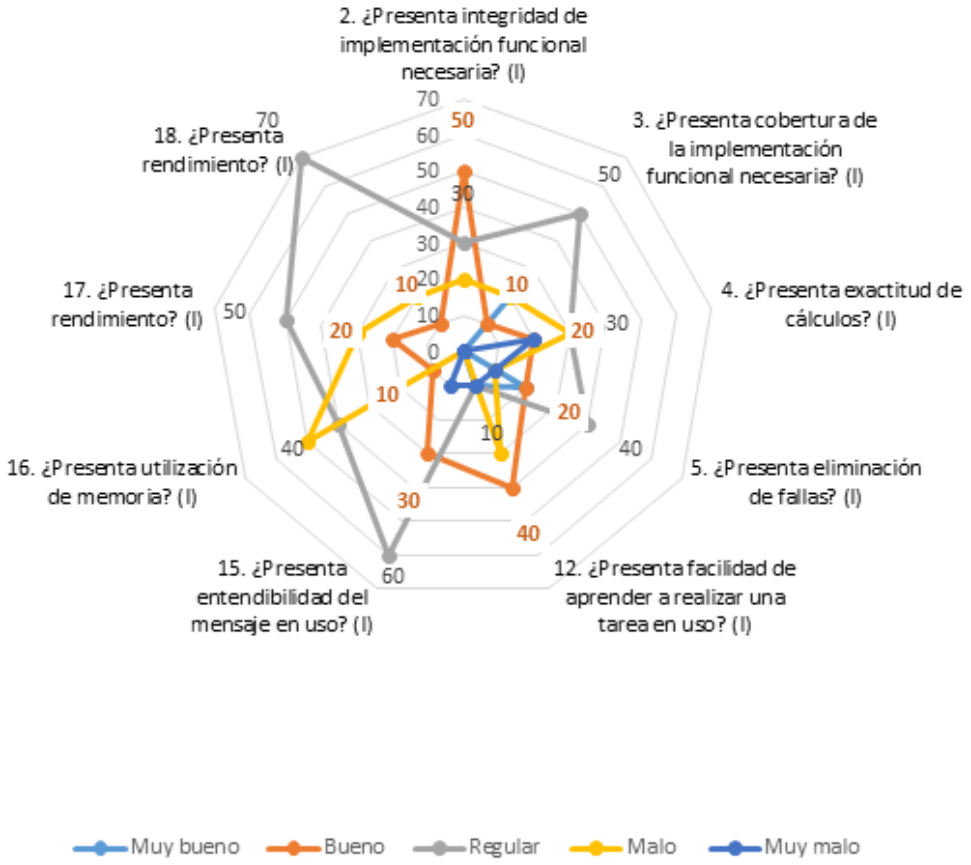
El análisis de los criterios descritos en el párrafo anterior permite determinar que en la fase de diseño de la Metodología Ágil Iconix sin intervención, tiene algunas bondades como: *Presenta prevención de operación incorrecta* en la tolerancia a fallas; *Presenta función de comprensión* en su entendibilidad; *Presenta comprensión de entradas y salidas* en su entendibilidad; *Presenta ratio de éxito de cambios* en lo correspondiente a estabilidad; *Presenta adaptabilidad a la organización y a la infraestructura de la misma* en el aspecto de adaptabilidad, asimismo, se tiene la debilidad en el diseño, en el aspecto de *Presenta impacto de la modificación* en lo correspondiente a la estabilidad.

Tabla 24. Distribución porcentual del Pre Test de la Calidad del Producto Software en función de la Implementación de la Metodología Ágil Iconix sin intervención

IMPLEMENTACION	**Muy bueno**	**Bueno**	**Regular**	**Malo**	**Muy malo**
2. ¿Presenta integridad de implementación funcional necesaria? (I)	0	50	30	20	0
3. ¿Presenta cobertura de la implementación funcional necesaria? (I)	20	10	50	20	0
4. ¿Presenta exactitud de cálculos? (I)	0	20	30	30	20
5. ¿Presenta eliminación de fallas? (I)	20	20	40	10	10
12. ¿Presenta facilidad de aprender a realizar una tarea en uso? (I)	10	40	10	30	10
15. ¿Presenta entendibilidad del mensaje en	0	30	60	0	10

uso? (I)					
16. ¿Presenta utilización de memoria? (I)	0	10	40	50	0
17. ¿Presenta rendimiento? (I)	0	20	50	30	0
18. ¿Presenta rendimiento? (I)	0	10	70	20	0

Figura 30. Distribución porcentual del Pre Test de la Calidad del Producto Software en función de la Implementación de la Metodología Ágil Iconix sin intervención



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Comentario

En la tabla 24 y figura 30, se tiene que la distribución de los porcentajes de la fase de Implementación de la Metodología Ágil Iconix sin intervención, presenta una mixtura de la escala de medición para determinar los aspectos favorables de los productos software, los que se describen a continuación: el 80% de los ingenieros entrevistados califican a *Presenta integridad de implementación funcional necesaria* como regular o bueno, en su aplicabilidad de los productos software: requisitos funcionales y no funcionales,

descripción de casos de uso, interfaz gráfica de usuario (GUI) y código fuente; el 80% de los entrevistados califica a *Presenta cobertura de la implementación funcional necesaria* con atributos que van de regular a muy bueno, en su aplicabilidad de los productos software: requisitos funcionales y no funcionales, descripción de casos de uso, interfaz gráfica de usuario (GUI) y código fuente; el 50% de los ingenieros entrevistados califica a *Presenta exactitud de cálculos* de regular a bueno, en su funcionalidad de los productos software: código fuente y pruebas unitarias; el 80% de los entrevistados califican a *Presenta eliminación de fallas* con atributos de regular a muy bueno, en lo correspondiente a madurez de los productos software: reporte de pruebas unitarias y reporte de pruebas de integración; asimismo se observa que, el 60% de los ingenieros entrevistados califica a *Presenta facilidad de aprender a realizar una tarea en uso* con un atributo que varía de regular a muy bueno, en su facilidad de aprendizaje en los producto software: Requisitos funcionales y no funcionales, Lista de casos de uso y descripción de casos de uso; el 90% de los entrevistados califica a *Presenta entendibilidad del mensaje en uso* como regular o bueno, a los productos software: descripción de casos de uso, código fuente y interfaz gráfica de usuario (GUI); análogamente, el 50% de los ingenieros califica a *Presenta utilización de memoria* como regular o bueno, en lo correspondiente a utilización de recursos de los productos software: código fuente y técnicas de codificación; el 70% de los ingenieros entrevistados califican a *Presenta rendimiento* como regular o bueno, en su comportamiento en el tiempo de los productos: casos de pruebas y reporte de pruebas de integración; y el 80% de los entrevistados califica a *Presenta rendimiento* como regular o buena, en el aspecto de confiabilidad del código fuente.

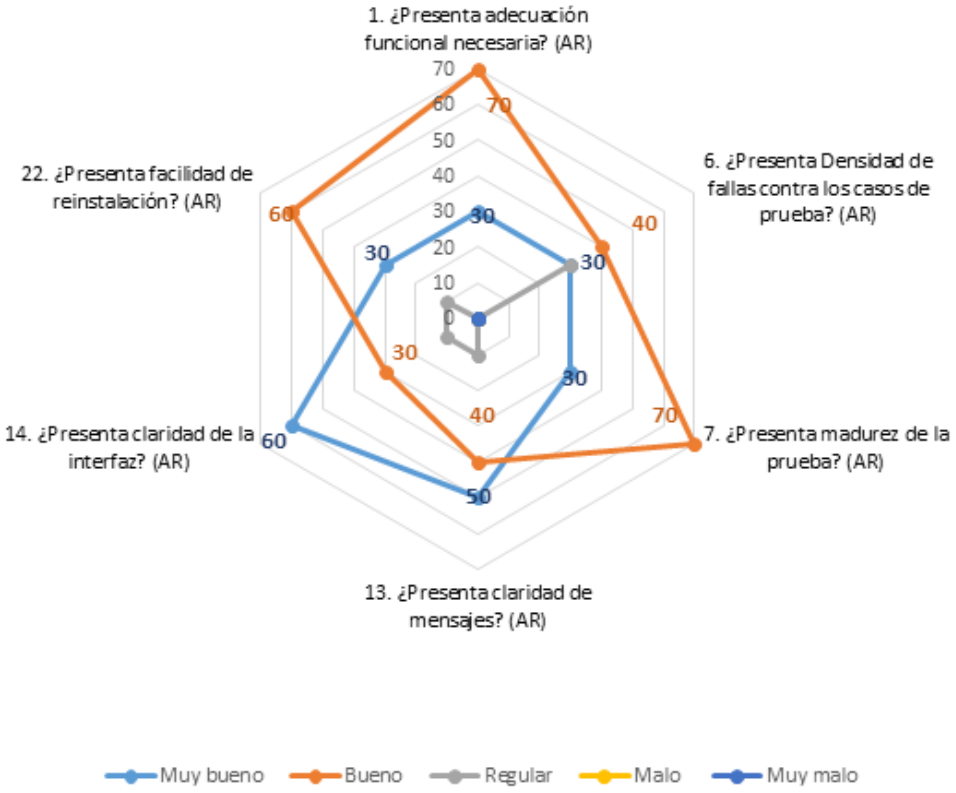
El análisis de los resultados de la fase de implementación de la Metodología Ágil Iconix sin intervención, presenta algunas bondades en los productos software evaluados cuya valoración más frecuente es la de regular o buena con menor incidencia. A continuación, se describen los aspectos descriptivos de la Metodología Ágil Iconix con intervención.

Tabla 25. Distribución porcentual del Pos Test de la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix con intervención

ANALISIS DE REQUISITOS	Muy bueno	Bueno	Regular	Malo	Muy malo
1. ¿Presenta adecuación funcional necesaria? (AR)	30	70	0	0	0

6. ¿Presenta Densidad de fallas contra los casos de prueba? (AR)	30	40	30	0	0
7. ¿Presenta madurez de la prueba? (AR)	30	70	0	0	0
13. ¿Presenta claridad de mensajes? (AR)	50	40	10	0	0
14. ¿Presenta claridad de la interfaz? (AR)	60	30	10	0	0
22. ¿Presenta facilidad de reinstalación? (AR)	30	60	10	0	0

Figura 31. Distribución porcentual del Pos Test de la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix con intervención



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Comentario

En la tabla 25 y figura 31, se observa que la fase de análisis de requisito de la Metodología Ágil Iconix con intervención, tiene mayor incidencia en los calificativos de bueno y acumulando los porcentajes, la tendencia de los calificativos se concentran más para la escala de bueno o muy bueno, los detalles se presentan a continuación: El 100% de los ingenieros de software entrevistados califican a *Presenta adecuación funcional necesaria* como bueno o muy bueno, en su aplicabilidad de los productos software: requisitos

funcionales y no funcionales, descripción de casos de uso, interfaz gráfica de usuario (GUI) y código fuente; el 70% de los entrevistados evalúa a *Presenta Densidad de fallas contra los casos de prueba* como bueno o muy bueno, en su madurez en los productos software: reporte de pruebas unitarias y reporte de pruebas de integración; el 100% de los entrevistados evalúan a *Presenta madurez de la prueba* como bueno o muy bueno, en su madurez para los productos de software: reporte de pruebas unitarias y reporte de pruebas de integración; el 90% de los entrevistados evalúa a *Presenta claridad de mensajes* como bueno o muy bueno, en su operabilidad para los productos software: descripción de casos de uso, código fuente y interfaz gráfica de usuario (GUI); otro tanto de 90% de los entrevistados evalúa a *Presenta claridad de la interfaz* como bueno o muy bueno, en su operabilidad de los productos software: descripción de casos de uso, código fuente y interfaz gráfica de usuario (GUI); y otro 90% de los entrevistados evalúa a *Presenta facilidad de reinstalación* como bueno o muy bueno, en su adaptabilidad de los procedimiento de instalación.

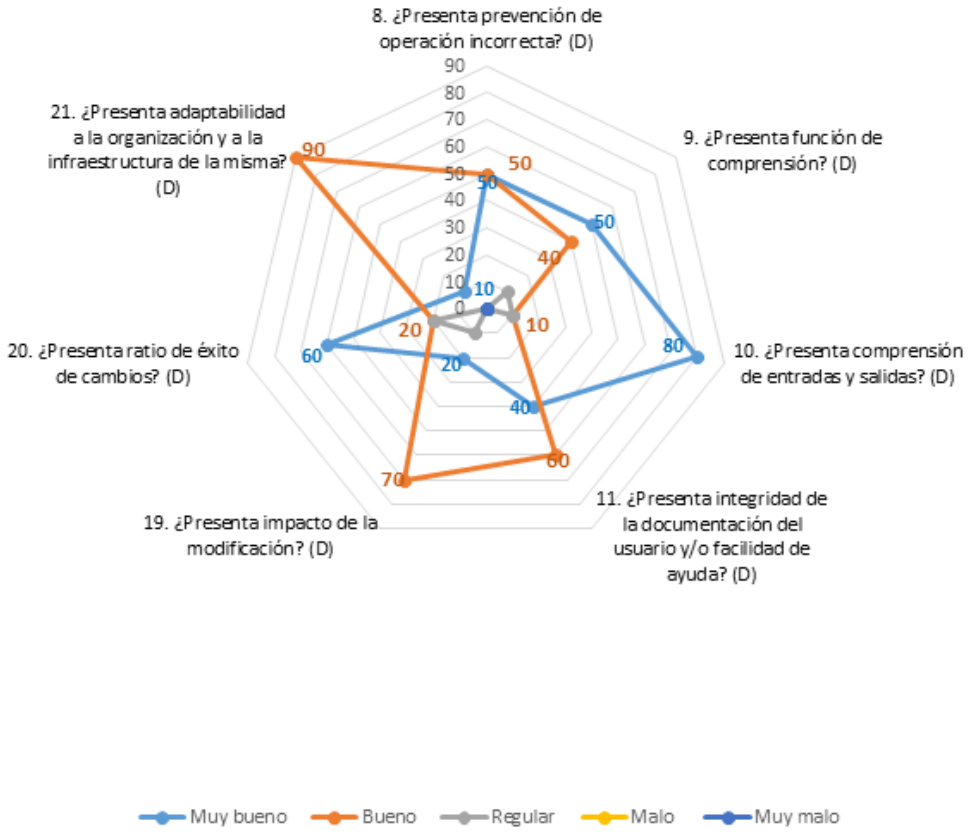
Los resultados descritos con respecto a la calidad del producto de software en función del análisis de requisitos determinan que las bondades de esta fase son seis aspectos buenos: *Presenta adecuación funcional necesaria, en los productos* requisitos funcionales y no funcionales, descripción de casos de uso, interfaz gráfica de usuario (GUI) y código fuente; *Presenta Densidad de fallas contra los casos de prueba:* reporte de pruebas unitarias y reporte de pruebas de integración; *Presenta madurez de la prueba, en los productos* reporte de pruebas unitarias y reporte de pruebas de integración; *Presenta claridad de mensajes e interfaz con sus productos software, cuyos productos son:* descripción de casos de uso, código fuente y Interfaz gráfica de usuario (GUI), *y Presenta facilidad de reinstalación.*

Tabla 26. Distribución porcentual del Pos Test de la Calidad del Producto Software en función del Diseño de la Metodología Ágil Iconix con intervención.

DISEÑO	**Muy bueno**	**Bueno**	**Regular**	**Malo**	**Muy malo**
8. ¿Presenta prevención de operación incorrecta? (D)	50	50	0	0	0
9. ¿Presenta función de comprensión? (D)	50	40	10	0	0
10. ¿Presenta comprensión de entradas y salidas? (D)	80	10	10	0	0

11. ¿Presenta integridad de la documentación del usuario y/o facilidad de ayuda? (D)	40	60	0	0	0
19. ¿Presenta impacto de la modificación? (D)	20	70	10	0	0
20. ¿Presenta ratio de éxito de cambios? (D)	60	20	20	0	0
21. ¿Presenta adaptabilidad a la organización y a la infraestructura de la misma? (D)	10	90	0	0	0

Figura 32. Distribución porcentual del Pos Test de la Calidad del Producto Software en función del Diseño de la Metodología Ágil Iconix con intervención



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Comentario

En la tabla 26 y figura 32, se tiene que la fase de diseño de la Metodología Ágil Iconix con intervención, tiene mayor incidencia en los calificativos de bueno y muy bueno, acumulando los porcentajes, la tendencia de los calificativos se concentran en la escala de bueno o muy bueno, los detalles se describen a continuación: El 100% de los ingenieros entrevistados evalúan a *Presenta prevención de operación incorrecta* con el atributo de

bueno o muy bueno, en su tolerancia a fallas de los productos de software: requisitos funcionales y no funcionales y reporte de pruebas unitarias; el 90% de los entrevistados evalúa a *Presenta función de comprensión* como bueno o muy bueno, en su entendibilidad de los productos software: requisitos funcionales y no funcionales, descripción de casos de uso e interfaz gráfica de usuario (GUI); el 90% de los entrevistados evalúan a *Presenta comprensión de entradas y salidas* como bueno o muy bueno, en su entendibilidad de los productos software: requisitos funcionales y no funcionales, descripción de casos de uso. y interfaz gráfica de usuario (GUI); el 100% de los ingenieros especialistas evalúan a *Presenta integridad de la documentación del usuario y/o facilidad de ayuda* como bueno o muy bueno, en su factibilidad de aprendizaje de los productos software: requisitos funcionales y no funcionales, lista de casos de uso y descripción de casos de uso; análogamente se tiene al 90% de los entrevistados que evalúan a *Presenta impacto de la modificación* como bueno o muy bueno, en su estabilidad de los productos software: diagrama de clases y código fuente; el 80% de los ingenieros evalúa a *Presenta ratio de éxito de cambios* como bueno o muy bueno, en su estabilidad de los productos software: diagrama de clases y código fuente; y el 100% de los ingenieros especialistas evalúan a *Presenta adaptabilidad a la organización y a la infraestructura de la misma* como bueno o muy bueno, en su adaptabilidad en el producto de arquitectura técnica (diagrama de despliegue y de componentes).

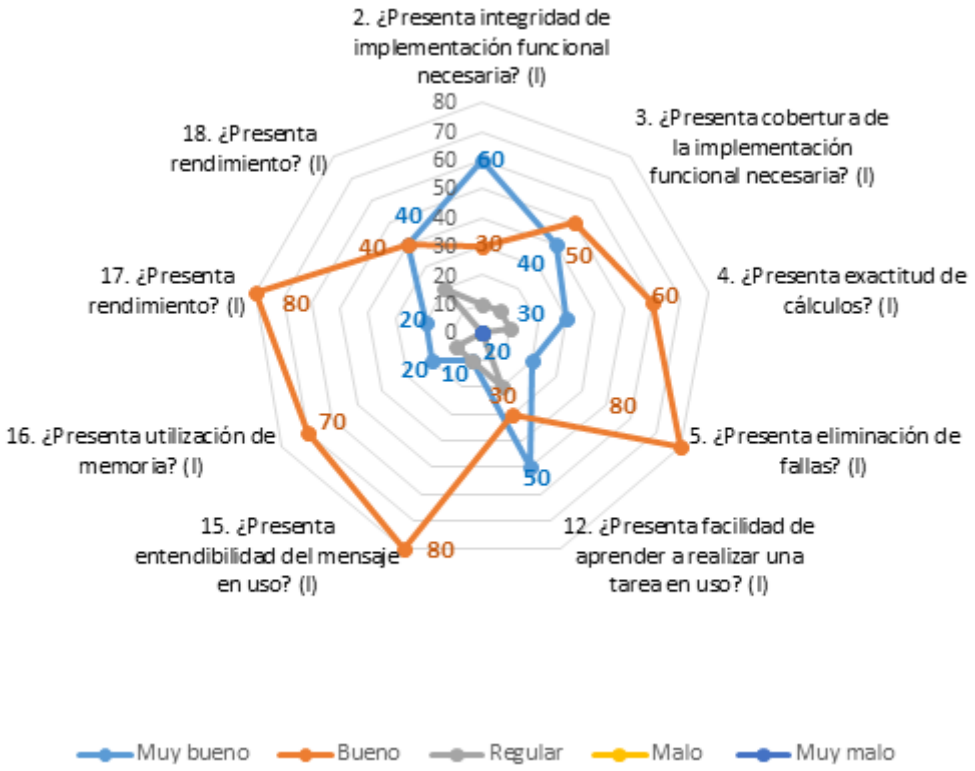
Los resultados descritos con respecto a la calidad del producto de software en función del diseño de la Metodología Ágil Iconix con intervención, determinan que las bondades de esta fase son siete aspectos buenos o muy buenos, en los siguientes aspectos: *Presenta prevención de operación incorrecta*, en su tolerancia a fallas de los productos de software: requisitos funcionales y no funcionales y reporte de pruebas unitarias; *Presenta función de comprensión*, en su entendibilidad de los productos software: requisitos funcionales y no funcionales, descripción de casos de uso e interfaz gráfica de usuario (GUI); *Presenta comprensión de entradas y salidas*, en su entendibilidad de los productos software: requisitos funcionales y no funcionales, descripción de casos de uso e interfaz gráfica de usuario (GUI); *Presenta integridad de la documentación del usuario y/o facilidad de ayuda*, en su factibilidad de aprendizaje de los productos software: requisitos funcionales y no funcionales, lista de casos de uso y descripción de casos de uso; *Presenta impacto de la modificación*, en su estabilidad de los productos software: diagrama de clases y código fuente; *Presenta ratio de éxito de cambios*, en su estabilidad de los productos software:

diagrama de clases y código fuente; y *Presenta adaptabilidad a la organización y a la infraestructura de la misma*, en su adaptabilidad en el producto de arquitectura técnica (diagrama de despliegue y de componentes).

Tabla 27. Distribución porcentual del Pos Test de la Calidad del Producto Software en función de la Implementación de la Metodología Ágil Iconix con intervención

IMPLEMENTACION	**Muy bueno**	**Bueno**	**Regular**	**Malo**	**Muy malo**
2. ¿Presenta integridad de implementación funcional necesaria? (I)	60	30	10	0	0
3. ¿Presenta cobertura de la implementación funcional necesaria? (I)	40	50	10	0	0
4. ¿Presenta exactitud de cálculos? (I)	30	60	10	0	0
5. ¿Presenta eliminación de fallas? (I)	20	80	0	0	0
12. ¿Presenta facilidad de aprender a realizar una tarea en uso? (I)	50	30	20	0	0
15. ¿Presenta entendibilidad del mensaje en uso? (I)	10	80	10	0	0
16. ¿Presenta utilización de memoria? (I)	20	70	10	0	0
17. ¿Presenta rendimiento? (I)	20	80	0	0	0
18. ¿Presenta rendimiento? (I)	40	40	20	0	0

Figura 33. Distribución porcentual del Pos Test de la Calidad del Producto Software en función de la Implementación de la Metodología Ágil Iconix con intervención.



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Comentario

En la tabla 27 y figura 33, se tiene que la fase de implementación de la Metodología Ágil Iconix con intervención, tiene mayor incidencia en los calificativos de bueno y acumulando los porcentajes, la tendencia de los calificativos también se concentran en la escala de bueno o muy bueno, los detalles se describen a continuación: El 90% de los ingenieros especialistas evalúan a *Presenta integridad de implementación funcional necesaria* como bueno o muy bueno, en su aplicabilidad para los productos software: requisitos funcionales y no funcionales, descripción de casos de uso, interfaz gráfica de usuario (GUI) y código fuente; otro 90% de los especialistas entrevistados evalúan a *Presenta cobertura de la implementación funcional necesaria* como bueno o muy bueno, en su aplicabilidad de los productos software: requisitos funcionales y no funcionales, descripción de casos de uso, interfaz gráfica de usuario (GUI) y código fuente; otro 90% de los ingenieros consultados evalúa a *Presenta exactitud de cálculos* como bueno o muy bueno, en su precisión de los productos software: código fuente y pruebas unitarias; el

100% de los ingenieros especialistas evalúan a *Presenta eliminación de fallas* como bueno o muy bueno, en su madurez de los productos: reporte de pruebas unitarias y reporte de pruebas de integración; análogamente se tiene que, el 80% de los ingenieros entrevistados evalúa a *Presenta facilidad de aprender a realizar una tarea en uso* como bueno o muy bueno, en su aspecto de facilidad de aprendizaje de los productos software: requisitos funcionales y no funcionales, Lista de casos de uso y descripción de casos de uso; un 90% de los ingenieros evalúa a *Presenta entendibilidad del mensaje en uso* como bueno o muy bueno, aspecto de operabilidad de los productos: descripción de casos de uso, código fuente y interfaz gráfica de usuario (GUI); otro 90% de los especialistas entrevistados evalúa a *Presenta utilización de memoria* como bueno o muy bueno, en la utilización de recursos de los productos software: código fuente y técnicas de codificación; asimismo, un 100% de los ingenieros evalúa a *Presenta rendimiento* como bueno o muy bueno, en su comportamiento en el tiempo de los productos software: casos de pruebas y reporte de pruebas de integración; y un 80% de los entrevistados evalúa a *Presenta rendimiento* en el producto de código fuente como bueno o muy bueno

Los resultados descritos con respecto a la calidad del producto de software en función de la implementación de la Metodología Ágil Iconix con intervención, determinan que las bondades de esta fase son nueve aspectos buenos o muy buenos, que abarca los siguientes aspectos: *Presenta integridad de implementación funcional necesaria* en los productos software: requisitos funcionales y no funcionales, descripción de casos de uso, interfaz gráfica de usuario (GUI) y código fuente; *Presenta cobertura de la implementación funcional necesaria,* en los productos software: requisitos funcionales y no funcionales, descripción de casos de uso, interfaz gráfica de usuario (GUI) y código fuente; *Presenta exactitud de cálculos,* en los productos software: código fuente y pruebas unitarias; *Presenta eliminación de fallas,* en los productos software: reporte de pruebas unitarias y reporte de pruebas de integración; *Presenta facilidad de aprender a realizar una tarea en uso,* en los productos software: requisitos funcionales y no funcionales, Lista de casos de uso y descripción de casos de uso; *Presenta entendibilidad del mensaje en uso,* en los productos software: descripción de casos de uso, código fuente e interfaz gráfica de usuario (GUI); *Presenta utilización de memoria,* en los productos software: código fuente y técnicas de codificación; *Presenta rendimiento,* en los productos software: casos de pruebas y reporte de pruebas de integración; *Presenta rendimiento,* en el producto software relacionado al código fuente.

Cálculo para contraste de la primera hipótesis específica

Ha: Si modificamos la fase de análisis de requisitos mediante la inclusión de ingeniería de requisitos, entonces se mejora la calidad del producto software.

Tabla 28. Medidas de resumen de la Calidad del Producto Software en función de la Análisis de Requisitos de la Metodología Ágil Iconix sin y con intervención

Análisis de requisito	N	Media	Desvia-ción estándar	Error estándar	95% del intervalo de confianza para la media		Mínimo	Máximo
					Límite inferior	Límite superior		
Análisis de requisito sin intervención	10	20,10	3,348	1,059	17,70	22,50	14	26
Análisis de requisito con intervención	10	25,70	1,767	,559	24,44	26,96	22	28
Total	20	22,90	3,878	,867	21,08	24,72	14	28

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

En la tabla 28, se tiene algunos estadígrafos descriptivos como la puntuación media de 20,10 de la calidad del producto software medida en términos del análisis de requisito de la Metodología Ágil Iconix sin intervención, con una desviación estándar de 3,348 puntos arriba o debajo de la media, cuyo intervalo para la media con un 95% de confianza varía de 17,7 hasta 22,50, asimismo, se tiene que el rango oscila de 14 puntos hasta 26 puntos. Análogamente se tiene, la puntuación media de 25,70 de la calidad del producto software medida en términos del análisis de requisitos de la Metodología Ágil Iconix con intervención, con una desviación estándar de 1,767 puntos arriba o debajo de la media, cuyo intervalo para la media con un 95% de confianza varía de 24,44 hasta 26,96, asimismo, se tiene que el rango oscila de 22 puntos hasta 28 puntos.

Los resultados revelan en todos los aspectos que la calidad del producto software medida en términos del análisis de requisitos de la Metodología Ágil Iconix con intervención es mayor que la calidad del producto software medida en términos del análisis de requisitos de la Metodología Ágil Iconix sin intervención.

Tabla 29. Test de Levene para la prueba de homocedasticidad y la prueba T- Student de la Calidad de Producto Software en función de la Análisis de Requisitos de la Metodología Ágil Iconix sin y con intervención

		Prueba de Levene de igualdad de varianzas		prueba t para la igualdad de medias						
									95% de intervalo de confianza de la diferencia	
		F	Sig.	t	gl	Sig. (bilateral)	Diferencia de medias	Diferencia de error estándar	Inferior	Superior
CALIDAD DEL PRODUCTO SOFTWARE CON RESPECTO A ANALISIS DE REQUISITO	Se asumen varianzas iguales	2,437	,136	-4,678	18	,000	-5,600	1,197	-8,115	-3,085
	No se asumen varianzas iguales			-4,678	13,652	,000	-5,600	1,197	-8,174	-3,026

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

La tabla 29, muestra el Test de Levene que supone que las varianzas de las muestras en análisis son iguales, es decir:

$$H_0 : \sigma^2_{AR\ \text{sin}\ int} = \sigma^2_{AR\ \text{con}\ int}$$

$$H_A : \sigma^2_{AR\ \text{sin}\ int} \neq \sigma^2_{AR\ \text{con}\ int}$$

El estadígrafo de prueba de Levene representado por F = 2,437 que asocia un p valor 0,136 que es mayor que el nivel de significancia de $\alpha < 0,05$, esto indica que la muestra aporta información suficiente para afirmar con un 95% de confianza y 5% de significancia, que las varianzas de la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix sin y con intervención, son iguales, con lo que se cumple con el supuesto de homocedasticidad.

Asimismo, en la tabla 29, se observa el Test T- Student que supone en las hipótesis de contraste, que las medias de la Calidad de Producto Software en función de la Análisis de Requisitos de la Metodología Ágil Iconix sin y con intervención, son iguales, caso contrario son diferentes, es decir:

$$H_0 : \mu_{AR\ \text{sin}\ int} = \mu_{AR\ \text{con}\ int}$$

$$H_A : \mu_{AR\ \text{sin}\ int} \neq \mu_{AR\ \text{con}\ int}$$

El estadígrafo de prueba T – Student t = - 4,678 con un p – valor de 0,000 que es menor que el nivel de significancia de $\alpha < 0,05$, esto indica que, la muestra aporta información

suficiente con un 95% de confianza y 5% de significancia, que la Calidad de Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix sin y con intervención, son diferentes y que la diferencia favorece a la Calidad de Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix con intervención, dado que esta diferencia de $\mu_{AR\ \text{sin}\ int} - \mu_{AR\ \text{con}\ int} = -5{,}6$.

Tabla 30. El análisis de varianza de la Calidad de Producto Software en función de la Análisis de Requisitos de la Metodología Ágil Iconix sin y con intervención

Fuentes de variación del Análisis de Requisito	Suma de cuadrados	gl	Media cuadrática	F	Sig.
Entre grupos	156,800	1	156,800	21,879	,000
Dentro de grupos	129,000	18	7,167		
Total	285,800	19			

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

En la tabla 30, se presenta el análisis de varianza con el que se explica las variaciones de la Calidad de Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix sin y con intervención, cuyo modelo es el siguiente:

$$Y_{ij} = \mu + \tau_i + \varepsilon_{ij}$$

Donde:

μ: *Media global de la Calidad de Producto Software*

τ_i: *Mide el efecto del Analisis de Requisitos de la Metodologia Ágil Iconix en su versión* sin *o con intervención*

ε_{ij}: *Es el error atribuible a la medicion dela Calidad de Producto Software*

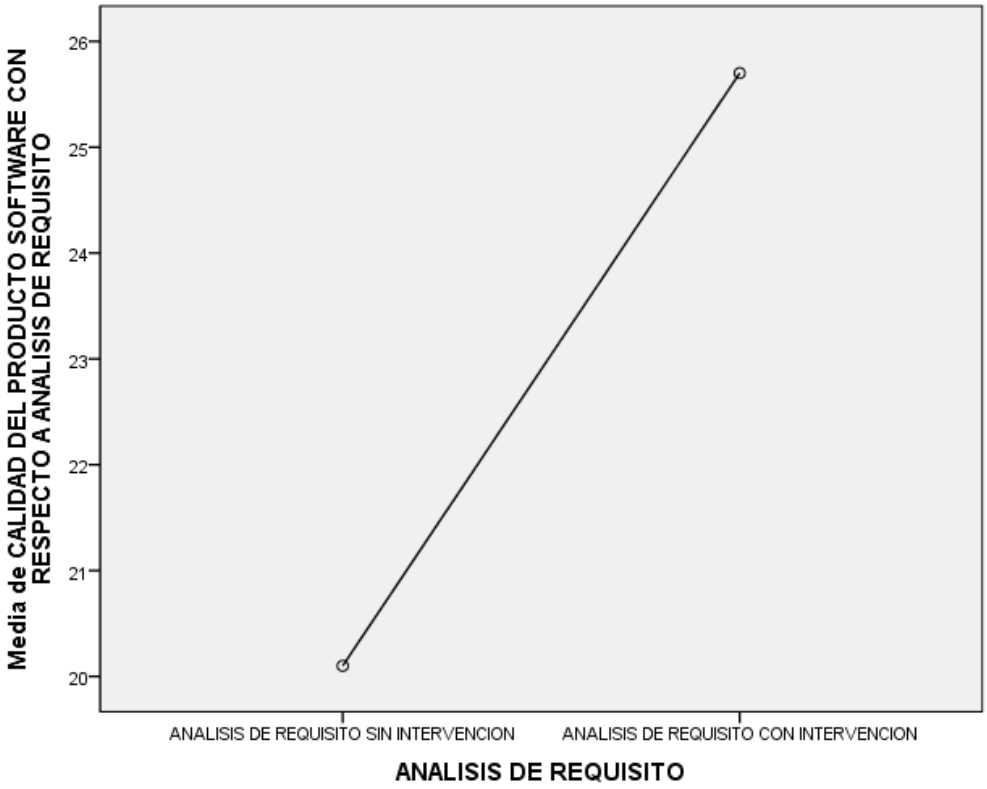
$$H_0 : \tau_{AR\ \text{sin}\ int} = \tau_{AR\ \text{con}\ int}$$

$$H_A : \tau_{AR\ \text{sin}\ int} \neq \tau_{AR\ \text{con}\ int}$$

Los resultados de la tabla 30, muestra el estadígrafo de prueba F = 21,879 cuyo p-valor = 0,000 es menor que el nivel de significancia de $\alpha < 0{,}05$ que indica que la muestra aporta información suficiente, con un 95% de confianza 5% de significancia, para afirmar que existe diferencia significativa entre los tratamientos representados el Análisis de Requisitos

de la Metodología Ágil Iconix sin y con intervención, que inciden en la Calidad de Producto Software; es decir se valida la hipótesis alterna que postula que $\tau_{AR\,\mathrm{sin}\,int} \neq \tau_{AR\,\mathrm{con}\,int}$. Estos resultados confirman la decisión evaluada con el T- Student descrito en la tabla 29.

Figura 34. Diagrama de medias de la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix sin y con intervención



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

El figura 34, indica que la media para la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix sin intervención, es menor que la media para la Calidad del Producto Software en función del Análisis de Requisitos de la Metodología Ágil Iconix con intervención.

Los resultados descritos determinan que el mejor tratamiento es la fase de Análisis de Requisitos de la Metodología Ágil Iconix con intervención, que incide significativamente

en la Calidad del Producto Software, expresado por la evaluación realizada por los ingenieros especialistas usando las normas NTP ISO/IEC 9126 y sus extensiones.

Cálculo para contraste de la segunda hipótesis específica

Ha: Si variamos la fase de diseño mediante la inclusión de patrones de diseño, entonces se mejora la calidad del producto software.

Tabla 31. Medidas de resumen de la Calidad del Producto Software en función del Diseño de la Metodología Ágil Iconix sin y con intervención

Fase de Diseño	N	Media	Desviación estándar	Error estándar	95% del intervalo de confianza para la media		Mínimo	Máximo
					Límite inferior	Límite superior		
Diseño sin intervención	10	23,00	4,372	1,382	19,87	26,13	16	30
Diseño con intervención	10	30,60	2,171	,686	29,05	32,15	28	33
Total	20	26,80	5,146	1,151	24,39	29,21	16	33

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

En la tabla 31, se tiene los estadígrafos descriptivos como la puntuación media de 23,00 de la calidad del producto software medida en términos del diseño de la Metodología Ágil Iconix sin intervención, con una desviación estándar de 4,372 puntos arriba o debajo de la media, cuyo intervalo para la media con un 95% de confianza oscila de 19,87 hasta 26,13, asimismo, se tiene que el rango oscila de 16 puntos hasta 30 puntos. Análogamente se tiene, la puntuación media de 30,60 de la calidad del producto software medida en términos del diseño de la Metodología Ágil Iconix con intervención, con una desviación estándar de 2,171 puntos arriba o debajo de la media, cuyo intervalo para la media con un 95% de confianza varía de 29,05 hasta 32,15, asimismo, se tiene que el rango oscila de 28 puntos hasta 33 puntos.

Los resultados revelan en todos los aspectos que la calidad del producto software medida

en términos del diseño de la Metodología Ágil Iconix con intervención es mayor que la calidad del producto software medida en términos del diseño de la Metodología Ágil Iconix sin intervención.

Tabla 32. Test de Levene para la prueba de homocedasticidad y la prueba T- Student de la Calidad de Producto Software en función del Diseño de la Metodología Ágil Iconix sin y con intervención

		Prueba de Levene de igualdad de varianzas		prueba t para la igualdad de medias						
									95% de intervalo de confianza de la diferencia	
		F	Sig.	t	gl	Sig. (bilateral)	Diferencia de medias	Diferencia de error estándar	Inferior	Superior
CALIDAD DEL PRODUCTO SOFTWARE CON RESPECTO A DISEÑO	Se asumen varianzas iguales	3,277	,087	-4,924	18	,000	-7,600	1,543	-10,843	-4,357
	No se asumen varianzas iguales			-4,924	13,183	,000	-7,600	1,543	-10,930	-4,270

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

La tabla 32, presenta el Test de Levene que supone que las varianzas de las muestras en análisis son iguales, es decir:

$$H_0 : \sigma^2_{D\,\text{sin}\,int} = \sigma^2_{D\,\text{con}\,int}$$

$$H_A : \sigma^2_{D\,\text{sin}\,int} \neq \sigma^2_{D\,\text{con}\,int}$$

El estadígrafo de prueba de Levene representado por F = 3,277 que asocia un p valor 0,087 que es mayor que el nivel de significancia de $\alpha < 0,05$, esto indica que la muestra aporta información suficiente para afirmar con un 95% de confianza y 5% de significancia, que las varianzas de la Calidad de Producto Software en función del Diseño de la Metodología Ágil Iconix sin y con intervención, son iguales, con lo que se cumple con el supuesto de homocedasticidad, que argumenta su validez del efecto del diseño en la calidad del producto software.

Asimismo, en la tabla 32, se presenta el Test T- Student que supone en las hipótesis de contraste, que las medias de Calidad del Producto Software en función del Diseño de la Metodología Ágil Iconix sin y con intervención, son iguales, caso contrario son diferentes, es decir:

$$H_0 : \mu_{D\ \text{sin}\ int} = \mu_{D\ \text{con}\ int}$$
$$H_A : \mu_{D\ \text{sin}\ int} \neq \mu_{D\ \text{con}\ int}$$

El estadígrafo de prueba T – Student t = - 4,924 con un p – valor de 0,000 que es menor que el nivel de significancia de $\alpha < 0{,}05$, esto indica que, la muestra aporta información suficiente con un 95% de confianza y 5% de significancia, que la Calidad de Producto Software en función del Diseño de la Metodología Ágil Iconix sin y con intervención, son diferentes y que la diferencia favorece a la Calidad de Producto Software en función de la fase de Diseño de la Metodología Ágil Iconix con intervención, dado que esta diferencia de $\mu_{D\ \text{sin}\ int} - \mu_{D\ \text{con}\ int} = -7{,}6$

Tabla 33. El análisis de varianza de la Calidad de Producto Software en función del Diseño de la Metodología Ágil Iconix sin y con intervención

Fuentes de variación de la Fase de Diseño	Suma de cuadrados	gl	Media cuadrática	F	Sig.
Entre grupos	288,800	1	288,800	24,246	,000
Dentro de grupos	214,400	18	11,911		
Total	503,200	19			

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

En la tabla 33, se presenta el análisis de varianza con el que se explica las variaciones de la Calidad de Producto Software en función del Diseño de la Metodología Ágil Iconix sin y con intervención, cuyo modelo es el siguiente:

$$Y_{ij} = \mu + \tau_i + \varepsilon_{ij}$$

Donde:

μ: *Media global de la Calidad de Producto Software*

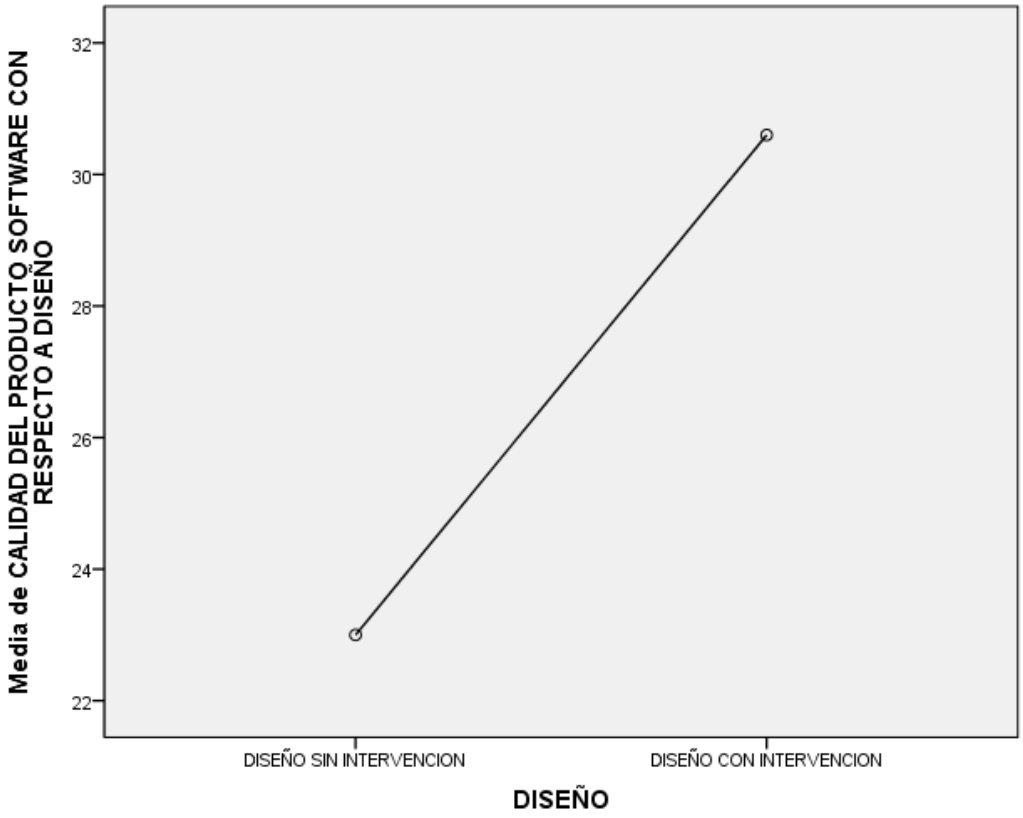
τ_i: *Mide el efecto del Diseño de la Metodologia Ágil Iconix en su versión sin o con intervención*

ε_{ij}: *Es el error atribuible a la medicion dela Calidad de Producto Software*

$$H_0 : \tau_{D\ \text{sin}\ int} = \tau_{D\ \text{con}\ int}$$
$$H_A : \tau_{D\ \text{sin}\ int} \neq \tau_{D\ \text{con}\ int}$$

Los resultados de la tabla 33, muestra el estadígrafo de prueba F = 24,246 cuyo p-valor = 0,000 es menor que el nivel de significancia de $\alpha < 0,05$ que indica que la muestra aporta información suficiente, con un 95% de confianza 5% de significancia, para afirmar que existe diferencia significativa entre los tratamientos representados por el Diseño de la Metodología Ágil Iconix sin y con intervención, que inciden en la Calidad de Producto Software; es decir se validad la hipótesis alterna que postula que $\tau_{D\,\mathrm{sin}\,int} \neq \tau_{D\,\mathrm{con}\,int}$. Estos resultados confirman la decisión evaluada con el T- Student descrito en la tabla 33.

Figura 35. Diagrama de medias de la Calidad del Producto Software en función del Diseño de la Metodología Ágil Iconix sin y con intervención



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

El figura 35, indica que la media para la Calidad del Producto Software en función del Diseño de la Metodología Ágil Iconix sin intervención, es menor que la media para la Calidad del Producto Software en función del Análisis de Requisito de la Metodología

Ágil Iconix con intervención.

Los cálculos de las tablas del 31 al 33, determinan que el mejor tratamiento es la fase de Diseño de la Metodología Ágil Iconix con intervención, que influye significativamente en la Calidad del Producto Software, expresado por la evaluación realizada por los ingenieros especialistas usando las normas NTP ISO/IEC 9126 y sus extensiones.

Cálculo para contraste de la tercera hipótesis específica

Ha: Si transformamos la fase de implementación mediante la inclusión de técnicas de programación orientada a objetos, entonces se mejora la calidad del producto software.

Tabla 34. Medidas de resumen de la Calidad del Producto Software en función de la Implementación de la Metodología Ágil Iconix sin y con intervención

Implementación	N	Media	Desviación estándar	Error estándar	95% del intervalo de confianza para la media		Mínimo	Máximo
					Límite inferior	Límite superior		
Implementación sin intervención	10	27,00	6,446	2,039	22,39	31,61	18	38
Implementación con intervención	10	38,00	2,449	,775	36,25	39,75	35	42
Total	20	32,50	7,373	1,649	29,05	35,95	18	42

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

En la tabla 34, se tiene los estadígrafos de resumen como la puntuación media de 27,00 de la calidad del producto software medida en términos de la implementación de la Metodología Ágil Iconix sin intervención, con una desviación estándar de 6,446 puntos arriba o debajo de la media, cuyo intervalo para la media con un 95% de confianza y 5% de significancia oscila de 22,39 hasta 31,61, asimismo, se tiene que el rango oscila de 18 puntos hasta 38 puntos. Análogamente se tiene, la puntuación media de 38,00 de la calidad del producto software medida en términos de la implementación de la Metodología Ágil Iconix con intervención, con una desviación estándar de 2,449 puntos arriba o debajo de la

media, cuyo intervalo para la media con un 95% de confianza y 5% de significancia oscila de 36,25 hasta 39,75, asimismo, se tiene que el rango oscila de 35 puntos hasta 42 puntos.

Los estadígrafos calculados revelan en todos los aspectos que la calidad del producto software medida en términos de la Implementación de la Metodología Ágil Iconix con intervención, es mayor que, la calidad del producto software medida en términos de la Implementación de la Metodología Ágil Iconix sin intervención.

Tabla 35. Test de Levene para la prueba de homocedasticidad y la prueba T- Student de la Calidad de Producto Software en función de la Implementación de la Metodología Ágil Iconix sin y con intervención

		Prueba de Levene de igualdad de varianzas		prueba t para la igualdad de medias						
									95% de intervalo de confianza de la diferencia	
		F	Sig.	t	gl	Sig. (bilateral)	Diferencia de medias	Diferencia de error estándar	Inferior	Superior
CALIDAD DEL PRODUCTO SOFTWARE CON RESPECTO A IMPLEMENTACION	Se asumen varianzas iguales	4,477	,049	-5,044	18	,000	-11,000	2,181	-15,582	-6,418
	No se asumen varianzas iguales			-5,044	11,546	,000	-11,000	2,181	-15,772	-6,228

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

La tabla 35, presenta el Test de Levene que supone que las varianzas de las muestras en análisis son iguales o son diferentes, es decir:

$$H_0 : \sigma^2_{I\ \text{sin}\ int} = \sigma^2_{I\ \text{con}\ int}$$

$$H_A : \sigma^2_{I\ \text{sin}\ int} \neq \sigma^2_{I\ \text{con}\ int}$$

El estadígrafo de prueba de Levene calculado a través de F = 4,477 que asocia un p valor 0,047 que es menor que el nivel de significancia de $\alpha < 0,05$, esto indica que la muestra no aporta información suficiente para afirmar con un 95% de confianza y 5% de significancia, que las varianzas de la Calidad de Producto Software en función de la Implementación de la Metodología Ágil Iconix sin y con intervención, son iguales, lo que implica que no se cumple con el supuesto de homocedasticidad; pero si cambiamos el nivel de significancia a $\alpha = 0,01$ y luego realizamos nuevamente la evaluación del estadígrafo de prueba de Levene

calculado a través de F = 4,477 que asocia un p valor 0,047 que es mayor que el nivel de significancia de $\alpha < 0{,}01$, este nuevo criterio de decisión estaría determinando que la muestra aporta información suficiente con un 99% de confianza y 1% de significancia que las varianzas de la implementación en sus dos versiones son iguales o que se cumple el supuesto de homocedasticidad, esto quiere decir que la fase de implementación de la Metodología Ágil Iconix en sus versiones sin y con intervención, no afecta a la varianza del error del diseño experimental: $Y_{ij} = \mu + \tau_i + \varepsilon_{ij}$

Asimismo, en la tabla 35, se presenta el Test T- Student que supone en las hipótesis de contraste, que las medias de la Calidad de Producto Software en función de la fase de Implementación de la Metodología Ágil Iconix sin y con intervención, son iguales, caso contrario son diferentes, es decir:

$$H_0 : \mu_{I\ \text{sin}\ int} = \mu_{I\ \text{con}\ int}$$

$$H_A : \mu_{I\ \text{sin}\ int} \neq \mu_{I\ \text{con}\ int}$$

El estadígrafo de prueba T – Student t = - 5,044 con un p – valor de 0,000 que es menor que el nivel de significancia de $\alpha < 0{,}05$, esto indica que, la muestra aporta información suficiente con un 95% de confianza y 5% de significancia, que la Calidad de Producto Software en función de la fase de Implementación de la Metodología Ágil Iconix sin y con intervención, son diferentes y que la diferencia favorece a la Calidad de Producto Software en función de la fase de Implementación de la Metodología Ágil Iconix con intervención, dado que esta diferencia de $\mu_{I\ \text{sin}\ int} - \mu_{I\ \text{con}\ int} = -11{,}00$.

Tabla 36. El análisis de varianza de la Calidad de Producto Software en función de la fase de Implementación de la Metodología Ágil Iconix sin y con intervención

Fuentes de variación de la Fase de Implementación	Suma de cuadrados	gl	Media cuadrática	F	Sig.
Entre grupos	605,000	1	605,000	25,444	,000
Dentro de grupos	428,000	18	23,778		
Total	1033,000	19			

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

En la tabla 36, se presenta el análisis de varianza con el que se explica las variaciones de la

Calidad de Producto Software en función de la fase de Implementación de la Metodología Ágil Iconix sin y con intervención, cuyo modelo es el siguiente:

$$Y_{ij} = \mu + \tau_i + \varepsilon_{ij}$$

Donde:

μ: *Media global de la Calidad de Producto Software*

τ_i: *Mide el efecto de la fase de implementación de la Metodologia Ágil Iconix en su versi*ón sin *o con intervenci*ón

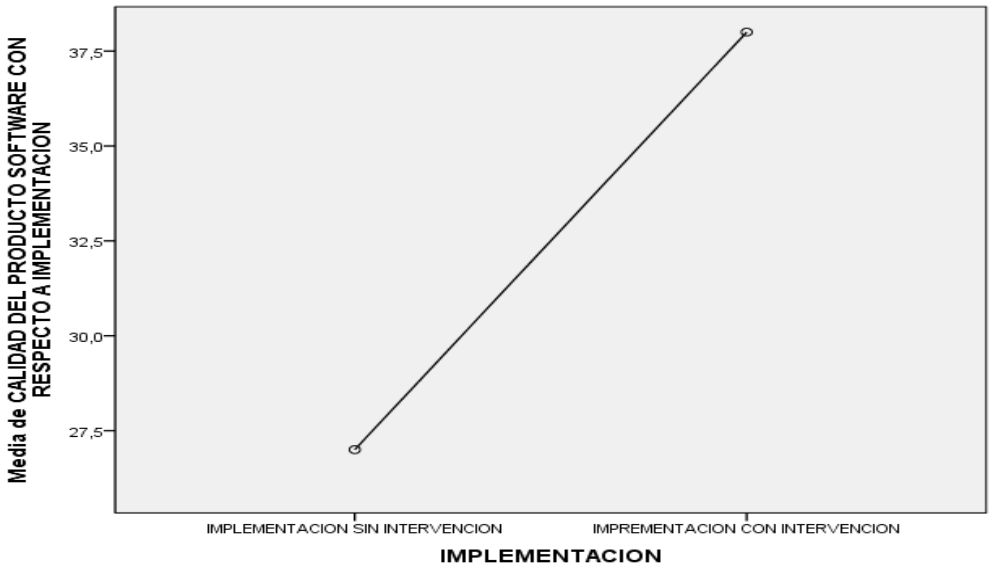
ε_{ij}: *Es el error atribuible a la medicion dela Calidad de Producto Software*

$$H_0 : \tau_{I \text{ sin } int} = \tau_{I \text{ con } int}$$

$$H_A : \tau_{I \text{ sin } int} \neq \tau_{I \text{ con } int}$$

Los resultados de la tabla 36, muestra el estadígrafo de prueba F = 25,444 cuyo p-valor = 0,000 es menor que el nivel de significancia de $\alpha < 0,05$ que indica que la muestra aporta información suficiente, con un 95% de confianza 5% de significancia, para afirmar que existe diferencia significativa entre los tratamientos representados por la fase de Implementación de la Metodología Ágil Iconix sin y con intervención, que inciden en la Calidad de Producto Software; es decir se valida la hipótesis alterna que postula que la media de los tratamientos son diferentes $\tau_{I \text{ sin } int} \neq \tau_{I \text{ con } int}$. Estos resultados confirman la decisión evaluada con el T- Student descrito en la tabla 35.

Figura 36. Diagrama de medias de la Calidad del Producto Software en función de la fase de Implementación de la Metodología Ágil Iconix sin y con intervención



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

El figura 36, indica que la media para la Calidad del Producto Software en función de la fase de Implementación de la Metodología Ágil Iconix sin intervención, es menor que la media para la Calidad del Producto Software en función de la fase de implementación de la Metodología Ágil Iconix con intervención.

Los cálculos de las tablas del 34 al 36, determinan que el mejor tratamiento es la fase de Implementación de la Metodología Ágil Iconix con intervención, que influye significativamente en la Calidad del Producto Software, expresado por la evaluación realizada por los ingenieros especialistas usando las normas NTP ISO/IEC 9126 y sus extensiones.

Cálculo para contraste de la hipótesis general

Ha: Si adaptamos la metodología ágil Iconix con la inclusión de técnicas de ingeniería de software, entonces se mejora la calidad del producto software, Lima, 2017.

Tabla 37. Medidas de resumen de la Calidad del Producto Software en función de la Metodología Ágil Iconix sin y con intervención

Metodología Ágil Iconix	N	Media	Desviación estándar	Error estándar	95% del intervalo de confianza para la media		Mínimo	Máximo
					Límite inferior	Límite superior		
Iconix sin intervención	10	70,10	13,178	4,167	60,67	79,53	54	94
Iconix con intervención	10	94,30	5,438	1,719	90,41	98,19	89	103
total	20	82,20	15,823	3,538	74,79	89,61	54	103

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

En la tabla 37, se observa que los estadígrafos de resumen como la puntuación media de 70,10 de la calidad del producto software medida en términos de la Metodología Ágil

Iconix sin intervención, con una desviación estándar de 13,178 puntos arriba o debajo de la media, cuyo intervalo para la media con un 95% de confianza y 5% de significancia oscila de 60,67 hasta 79,53, asimismo, se tiene que el rango de las puntuaciones de la calidad de producto software oscila de 54 puntos hasta 94 puntos. Análogamente se tiene, la puntuación media de 94,30 de la calidad del producto software medida en términos de la Metodología Ágil Iconix con intervención, con una desviación estándar de 5,438 puntos arriba o debajo de la media, cuyo intervalo para la media con un 95% de confianza y 5% de significancia oscila de 90,41 hasta 98,19, asimismo, se tiene que el rango oscila de 89 puntos hasta 103 puntos.

Los estadígrafos calculados revelan en todos los aspectos que la calidad del producto software medida en términos de la Metodología Ágil Iconix con intervención, es mayor que, la calidad del producto software medida en términos de la Metodología Ágil Iconix sin intervención.

Tabla 38. Test de Levene para la prueba de homocedasticidad y la prueba T- Student de la Calidad de Producto Software en función de la Metodología Ágil Iconix sin y con intervención

		Prueba de Levene de igualdad de varianzas		prueba t para la igualdad de medias						
									95% de intervalo de confianza de la diferencia	
		F	Sig.	t	gl	Sig. (bilateral)	Diferencia de medias	Diferencia de error estándar	Inferior	Superior
CALIDAD DEL PRODUCTO	Se asumen varianzas iguales	5,218	,035	-5,368	18	,000	-24,200	4,508	-33,671	-14,729
	No se asumen varianzas iguales			-5,368	11,978	,000	-24,200	4,508	-34,024	-14,376

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

La tabla 38, se observa el Test de Levene que supone que las varianzas de las muestras en análisis son iguales o son diferentes, es decir:

$$H_0 : \sigma^2_{MAI\ \text{sin}\ int} = \sigma^2_{MAI\ \text{con}\ int}$$

$$H_A : \sigma^2_{MAI\ \text{sin}\ int} \neq \sigma^2_{MAI\ \text{con}\ int}$$

El estadígrafo de prueba de Levene calculado a través de F = 5,218 que asocia un p valor 0,035 que es menor que el nivel de significancia de $\alpha < 0,05$, esto indica que la muestra no aporta información suficiente para afirmar con un 95% de confianza y 5% de significancia, que las varianzas de la Calidad de Producto Software en función de la Metodología Ágil Iconix sin y con intervención, son iguales, lo que implica que no se cumple con el supuesto de homocedasticidad; pero si cambiamos el nivel de significancia a $\alpha = 0,01$ y luego realizamos nuevamente la evaluación del estadígrafo de prueba de Levene calculado a través de F = 5,218 que asocia un p valor 0,035 que es mayor que el nivel de significancia de $\alpha < 0,01$, este nuevo criterio de decisión estaría determinando que la muestra aporta información suficiente con un 99% de confianza y 1% de significancia que las varianzas de las puntuaciones de la Calidad de los Productos Software en función de la Metodología Ágil Iconix sin y con intervención en sus dos versiones son iguales o que se cumple el supuesto de homocedasticidad, esto quiere decir que la Metodología Ágil Iconix en sus versiones sin y con intervención, no afecta a la varianza del error del diseño experimental:

$$Y_{ij} = \mu + \tau_i + \varepsilon_{ij}$$

Asimismo, en la tabla 38, se presenta el Test T- Student que supone en las hipótesis de contraste, que las medias de la Calidad de Producto Software en función de la Metodología Ágil Iconix sin y con intervención, son iguales, caso contrario son diferentes, es decir:

$$H_0 : \mu_{MAI\ \text{sin}\ int} = \mu_{MAI\ \text{con}\ int}$$

$$H_A : \mu_{MAI\ \text{sin}\ int} \neq \mu_{MAI\ \text{con}\ int}$$

El estadígrafo de prueba T – Student t = - 5,368 con un p – valor de 0,000 que es menor que el nivel de significancia de $\alpha < 0,05$, esto indica que, la muestra aporta información suficiente con un 95% de confianza y 5% de significancia, que la Calidad de Producto Software en función de la Metodología Ágil Iconix sin y con intervención, son diferentes y que la diferencia favorece a la Calidad de Producto Software en función de la Metodología Ágil Iconix con intervención, dado que esta diferencia de $\mu_{MAI\ \text{sin}\ int} - \mu_{MAI\ \text{con}\ int} = -24,200$.

Tabla 39. El análisis de varianza de la Calidad de Producto Software en función de la Metodología Ágil Iconix sin y con intervención

Fuentes de variación de la Metodología Ágil Iconix	Suma de cuadrados	gl	Media cuadrática	F	Sig.

Entre grupos	2928,200	1	2928,200	28,818	,000
Dentro de grupos	1829,000	18	101,611		
Total	4757,200	19			

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

En la tabla 39, se presenta el análisis de varianza con el que se explica las variaciones de la Calidad de Producto Software en función de la fuente de variación generada por la Metodología Ágil Iconix sin y con intervención, cuyo modelo es el siguiente:

$$Y_{ij} = \mu + \tau_i + \varepsilon_{ij}$$

Donde:

μ: $Media\ global\ de\ la\ Calidad\ de\ Producto\ Software$

τ_i: $Mide\ el\ efecto\ generado\ por\ la\ Metodologia$ Ágil $Iconix$ sin $o\ con\ intervencion$

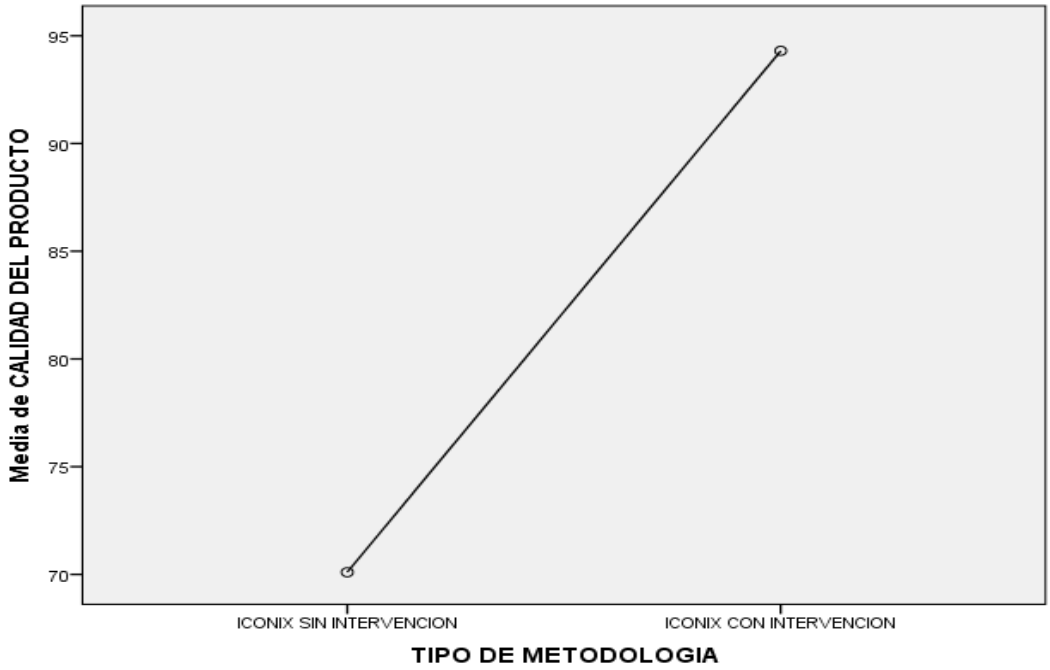
ε_{ij}: $Es\ el\ error\ atribuible\ a\ la\ medicion\ dela\ Calidad\ de\ Producto\ Software$

$$H_0 : \tau_{MAI\ \text{sin}\ int} = \tau_{MAI\ \text{con}\ int}$$

$$H_A : \tau_{MAI\ \text{sin}\ int} \neq \tau_{MAI\ \text{con}\ int}$$

Los resultados de la tabla 39, muestra el estadígrafo de prueba F = 28,818 cuyo p-valor = 0,000 es menor que el nivel de significancia de $\alpha < 0,05$ que indica que la muestra aporta información suficiente, con un 95% de confianza 5% de significancia, para afirmar que existe diferencia significativa entre los tratamientos representados por el efecto que ejerce la Metodología Ágil Iconix sin y con intervención, que inciden en la Calidad de Producto Software; es decir se valida la hipótesis alterna que postula que la media de los tratamientos son diferentes $\tau_{MAI\ \text{sin}\ int} \neq \tau_{MAI\ \text{con}\ int}$. Estos resultados confirman la decisión evaluada con el T- Student descrito en la tabla 38.

Figura 37. Diagrama de medias de la Calidad del Producto Software en función de la Metodología Ágil Iconix sin y con intervención



Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

Interpretación

La figura 37, indica que la media para la Calidad del Producto Software en función de la Metodología Ágil Iconix sin intervención, es menor que, la media para la Calidad del Producto Software en función de la Metodología Ágil Iconix con intervención.

Los cálculos de las tablas del 37 al 39, determinan que el mejor tratamiento es la Metodología Ágil Iconix con intervención, que influye significativamente en la Calidad del Producto Software, expresado por la evaluación realizada por los ingenieros especialistas usando las normas NTP ISO/IEC 9126 y sus extensiones.

V. DISCUSIÓN DE RESULTADOS

Nema, Rodríguez, Oivo y Tosun (2016), en el estudio "Análisis del concepto de deuda técnica en el contexto del desarrollo de software ágil: una revisión sistemática de la literatura", indican que las causas para incurrir en deuda técnica son; enfoque en la entrega rápida, problemas de arquitectura y diseño, los efectos más importantes; productividad reducida, degradación del software y mayor costo de mantenimiento. Según el presente estudio, tenemos el aporte de reducir la deuda técnica, porque aplicamos técnicas formales y probadas de ingeniería de software al intervenir la metodología ágil Iconix para mejorar la calidad del producto software, en función al contraste de la hipótesis general.

Yanga, Lianga y Avgerioub (2015), en la investigación "Un estudio de mapeo sistemático sobre la combinación de arquitectura de software y desarrollo ágil", analiza los métodos ágiles sobre: actividades y enfoques de arquitectura, prácticas ágiles, costos, beneficios, desafíos, factores y herramientas; el aspecto más destacado, que existe diferencia significativa entre las actividades de arquitectura y las prácticas ágiles empleadas. El estudio usa los principios de agilidad durante la intervención a la metodología ágil Iconix, donde se evidencia la combinación adecuada entre arquitectura y desarrollo con agilidad, donde el 100% de los ingenieros especialistas evalúan con la métrica *Presenta adaptabilidad a la organización y a la infraestructura de la misma,* como bueno o muy bueno, en su adaptabilidad al producto software, arquitectura técnica.

Silveira y Silva (2015), en el estudio "Adaptación de métodos ágiles: una revisión sistemática de la literatura", la industria de desarrollo de software ha adoptado métodos ágiles, porque son más flexibles y traen beneficios para administrar cambios de requerimientos, mejorar la productividad y alinear al cliente, la adaptación de métodos ágiles puede considerarse madura. En nuestro caso, luego del contraste de la primera hipótesis específica podemos afirmar que, los resultados revelan que la calidad del producto software medida en términos del análisis de requisitos de la Metodología Ágil Iconix con intervención es mayor que la calidad del producto software medida en términos del análisis de requisitos de la Metodología Ágil Iconix sin intervención.

Rodríguez, Musat y Yagüe (2010), en el estudio "Adopción de metodologías ágiles: un estudio comparativo entre España y Europa", según los resultados, la aplicación de metodologías ágiles en la industria española es un enfoque joven, en Europa, el 60% tenía

experiencia de 1 a 5 años y 20% más de 5 años, en España se adopta las metodologías ágiles a nivel local y personal, en europea presenta un nivel de madurez mayor, adoptan metodologías ágiles en proyectos para equipos distribuidos. En el Perú no hay experiencias de adaptación de metodologías agiles que mejore la calidad del producto software, el presente estudio es pionero en este aspecto, probado por los resultados donde, Los estadígrafos calculados revelan que la calidad del producto software medida en términos de la Metodología Ágil Iconix con intervención, es mayor que, la calidad del producto software medida en términos de la Metodología Ágil Iconix sin intervención.

Holvite, et al. (2017), en el estudio "Deuda técnica y desarrollo de software ágil prácticas y procesos: Una encuesta a los profesionales de la industria", opina que la deuda técnica se manifiesta y afecta los procesos de software, los resultados indican que las prácticas y procesos ágiles ayudan a reducir la deuda técnica, en particular, las técnicas que verifican y mantienen claridad de los productos software, implementando estándares de codificación y refactorización. Mediante la encuesta a los expertos se ha usado nueve métricas sobre implementación vale decir codificación, donde los resultados indican que la intervención a la metodología ágil Iconix (adaptación) tiene opiniones de bueno como máximo 80% y muy bueno como máximo 60%.

Britto (2014), en la tesis "Adaptación de un proceso de desarrollo de software basado en buenas prácticas", estudia el desarrollo de software con Iconix, se ha evaluado el proceso actual y desarrollado una adaptación. El nuevo proceso fue sometido a una prueba piloto y evaluado, se observa una mejora en el cumplimiento de buenas prácticas del 25% con respecto al proceso anterior. De acuerdo a la adaptación de la metodología ágil Iconix con técnicas formales de ingeniería de software, se verifica las tres hipótesis específicas de la adaptación con ingeniería de requisitos, patrones de diseño y principios de programación orientada a objetos como Solid, todas estas técnicas se alinean con la mejora del producto software.

Figueroa, Solis y Cabrera (2007), en el estudio "Comparación entre las metodologías tradicional y ágil", informan que para desarrollar software que presenten calidad de productos, debe elegirse el mejor proceso para un equipo y un proyecto, mediante los enfoques de procesos tradicionales y procesos ágiles. Durante el estudio se ha intervenido al análisis de requisitos con 5 procesos nuevos para desarrollo de análisis de requisitos, 9

procesos nuevos para desarrollo de diseño y 5 procesos nuevos para implementación, evidencia que se muestra en el anexo 6; siendo buenos y muy buenos según la opinión de los expertos.

Bona (2002), en el estudio "Avaliação de processos de software: Um estudo de caso em XP e Iconix", considera que Iconix es un proceso de desarrollo ágil, usa la orientación a objetos, tiene las fases de: análisis de requisitos, diseño preliminar, diseño e implementación, su esencia es la "definición de modelos de objetos a partir de los casos de uso". Al intervenir la metodología ágil Iconix mediante técnicas formales de ingeniería de software, hemos adaptado y mejorado varios procesos que según los expertos mejoran la calidad del producto software, de acuerdo a esta perspectiva, ya no es necesario las revisiones para cada fase, aun mas, se disminuye a tres fases de la metodología, siendo las fases de; análisis de requisitos, diseño e implementación, cumpliendo plenamente con los principios de agilidad en desarrollo de software.

Tarhan y Yilmaz (2013), en el estudio "Análisis sistemático y comparación del rendimiento de desarrollo y la calidad de productos software del proceso incremental y el proceso ágil", los resultados indican que el proceso ágil tiene mejor desempeño que el incremental en términos de productividad (79%), sobre el rendimiento de desarrollo y la calidad de productos software, el proceso ágil fue superior al proceso incremental. En el estudio al simplificar las fases de la metodología ágil Iconix, evidenciado en las tablas del anexo 6, donde la fase de análisis de requisitos presenta 4 procesos nuevos pero elimina la revisión de requisitos, en la fase de diseño se presenta 9 patrones, lo quiere decir que no necesariamente debe usarse los 9 patrones, podrían ser menos esto será función de la complejidad del software, también se elimina la revisión de diseño de detalle, en la fase de implementación al aplicar 5 principios para la generación de código limpio y también se elimina la revisión. Asimismo, evidenciamos en la investigación que se mejora la calidad interna y externa del software.

Kupiainena, Mäntyläa y Itkonena (2015), en el estudio "Uso de métricas en el desarrollo de software ágil y eficiente: Una revisión sistemática de la literatura", los resultados indican que las razones y los efectos del uso de métricas se centran en la medición de calidad de productos software, concluyen que el uso de métricas en el desarrollo de software ágil es similar al desarrollo de software tradicional. En la investigación se ha usado las normas NTP-ISO/IEC 9126-1 (2004). Ingeniería de software. calidad del producto; NTP-ISO/IEC 9126-2 (2004).

Ingeniería de software. calidad del producto. parte 2: métricas externas; NTP-ISO/IEC 9126-3 (2005). Ingeniería de software. calidad del producto. parte 3: métricas internas. Para evaluar la calidad del producto software.

Domínguez, Escalona, Mejías, Ross y Staples (2012), en el estudio "Evaluación de calidad para metodologías de ingeniería web basadas en modelos", indican que un estándar como ISO/IEC 9126 o ISO/IEC 25000 presenta un modelo para calidad de productos software; los resultados recomiendan a un conjunto de características y sub características de calidad, y definen una forma ágil de relacionar estas sub características de calidad con los atributos de los productos software. En el estudio para el modelo de calidad se usa la norma NTP-ISO/IEC 9126-1 (2004), para las métricas externas la norma NTP-ISO/IEC 9126-2 (2004) y para las métricas internas la norma NTP-ISO/IEC 9126-3 (2005); en los instrumentos para toma de datos se asocia las características, sub características, métricas y atributos para la encuesta a los expertos.

Hansen, Jonasson y Neukirchen (2011), en la investigación "Un estudio empírico de la arquitecturas de software y su efecto sobre la calidad del producto", el estudio considera a 1141 proyectos de código abierto en java, han calculado tres métricas de arquitectura de software que son: medición de clases por paquete, distancia normalizada y el exceso del grado de acoplamiento, la medida de estas métricas están relacionadas con las métricas del producto como: ratio de defectos, velocidad de descarga, métodos por clase y complejidad del método. En la investigación se aborda la complejidad del método y las clases por paquetes con patrones de diseño, a fin de crear una arquitectura de software robusta, y los métodos por clase con un principio de responsabilidad única, y estas son medidas en las fases de diseño e implementación para la metodología ágil Iconix.

Orantes (2006), en el estudio "Calidad de software usando metodologías ágiles para el desarrollo de software", indica que existen propuestas metodológicas con incidencia en el proceso de desarrollo, metodologías centradas en el producto software, dando importancia a la colaboración con el cliente y desarrollo incremental del software, manteniendo alta calidad de productos software. En la investigación, respecto al proceso de desarrollo se ha intervenido con nuevos proceso de ingeniería de software, como se indica en las hipótesis que han sido contrastadas afirmativamente, sea centrado en la mejora de calidad del producto software, y la colaboración con el cliente se evidencia en la formulación de la nueva fase de análisis de

requisitos, donde el cliente y usuario son importantes al definir, clasificar y priorizar los requisitos funcionales y no funcionales.

Freitas, Da Mota, Neto, O'Leary y Santana (2014) en la investigación "Usando un enfoque de métodos múltiples para comprender las pruebas en línea de producto de software (SPL) y la agilidad", el enfoque considera tres métodos: estudio de mapeo, estudio de caso y opinión de expertos, los resultados muestran cómo combinar agilidad y SPL, concluyen, que el enfoque requiere alto grado de esfuerzo para planificar, diseñar y realizar, pero ayuda a aumentar la comprensión sobre SPL. En el estudio se ha utilizado encuesta a expertos para validar el instrumento y obtener datos para la calidad el producto software, sobre las pruebas en línea del producto software, sea incluido las pruebas unitarias y las pruebas de integración que según los expertos están en la escala de bueno a muy bueno para la metodología ágil Iconix intervenida.

Da Mota, Do Carmo, Machado, McGregor, Santana y Romero (2010), en la investigación "Un estudio de mapeo sistemático de pruebas en líneas de productos de software", se evaluaron 120 estudios de 1993 a 2009, los resultados indican que las pruebas en línea son cubiertos para desarrollar un software. Las SPL según el presente estudio están cubiertas por las pruebas unitarias y pruebas de integración, que son evaluados por expertos la siguiente forma: que el 90% *Presenta integridad de implementación funcional necesaria* como bueno o muy bueno, 90% evalúan a *Presenta cobertura de la implementación funcional necesaria* como bueno o muy bueno, 90% evalúa a *Presenta exactitud de cálculos* como bueno o muy bueno, el 100% evalúan a *Presenta eliminación de fallas* como bueno o muy bueno, un 90% evalúa a *Presenta entendibilidad del mensaje en uso* como bueno o muy bueno, 90% evalúa a *Presenta utilización de memoria* como bueno o muy bueno, un 100% evalúa a *Presenta rendimiento* como bueno o muy bueno y un 80% evalúa a *Presenta rendimiento.*

Inayat, Salim, Marczak, Daneva y Shamshirband (2014), en la investigación "Un framework para estudiar la colaboración basada por los requisitos entre equipos ágiles: resultados de dos casos de estudio"; indican que durante el desarrollo de software con agilidad, la ingeniería de requisitos y los métodos ágiles comparten una colaboración entre interesados, los resultados indican que existe tendencias de colaboración, tendencia a la agrupación, reciprocidad de comunicación, e interacción entre los equipos ágiles. Daneva, Van der Veen, Amrit, Ghaisas,

Sikkel y Kum (2012), en el estudio "Priorización de requisitos ágiles en proyectos de sistemas tercerizados a gran escala: un estudio empírico", la aplicación de prácticas ágiles para la priorización de requisitos en proyectos distribuidos y tercerizados es una tendencia reciente, los hallazgos son: comprender las dependencias de los requisitos es importante para el despliegue exitoso de enfoques ágiles y el criterio de priorización es más importante para el establecimiento de grandes proyectos ágiles tercerizados. De acuerdo al presente estudio los requisitos funcionales y no funcionales son productos transversales a todo el desarrollo del software, la identificación, clasificación y priorización son desarrollados con aporte de ingeniería de colaboración, inmerso en el análisis de requisitos de la metodología ágil Iconix intervenida, que es parte de la primera hipótesis especifica de investigación contrastada afirmativamente.

Belfo (2012), en el artículo "Personas, dimensión organizacional y tecnológica para especificación de requerimientos de software", indican que la especificación de requisitos inapropiada es una razón del fracaso en desarrollo de software, los buenos requisitos se garantizan por la combinación correcta de personas, organización y tecnología; los resultados son: la comunicación y colaboración continua es el método más usado, sobre el usuario se debe hacer ASD más centrado en este como método útil de desarrollo de software ágil.

Baruah (2015), en la conferencia "Gestión de requerimientos en un entorno de software ágil", opina que la gestión de requisitos en la industria del software con requisitos cambiantes del lado del cliente, dificulta al desarrollador para que produzca un software de calidad, las metodologías ágiles de desarrollo de software, admiten cambios en los requisitos, pero se debe escuchar al cliente en todas las fases del desarrollo.

Kumar (2015), en el estudio "Investigando el cálculo de la recompensa de penalización de los usuarios de software y su impacto sobre la priorización de requisitos", según los resultados, para priorizar los requisitos con satisfacción del usuario, se debe seleccionar de manera cuidadosa aquellos requisitos para la implementación del producto software que tienen un máximo impacto en la satisfacción del usuario. La priorización de requisitos se incorpora en intervención a la metodología ágil Iconix, como se observa en el anexo 6.

Azadegan, Papamichail y Sampaio (2015), en la investigación "Aplicación de diseño de proceso colaborativo en la obtención de requisitos del usuario: Un caso de estudio",

indican que la obtención de requisitos es altamente colaborativa e involucra a muchas partes interesadas, hasta el 80% del costo del producto está determinado por las decisiones tomadas en colaboración con los interesados, durante las primeras etapas del ciclo de vida de los requisitos. La colaboración se ha incorporado en el análisis de requisitos de la metodología ágil Iconix intervenida, se muestra en el anexo 6.

Guerrero, Suárez y Gutiérrez (2013), en la investigación sobre "Patrones de diseño en el contexto de procesos de desarrollo de aplicaciones orientadas a la web", analizan patrones de diseño definidos por The Gang of Four (GoF), los resultados indican que de 23 patrones existentes, 8 patrones de diseño GoF son usados en desarrollo de software y los expertos usan 10 patrones de diseño GoF. Jiménez, Tello y Ríos (2014), en el estudio "Lenguajes de patrones de arquitectura de software: Una aproximación al estado del arte", concluyen que la evolución de la arquitectura de software ha llegado a niveles superiores, que permite resolver problemas de arquitectura con mejor calidad, encontrando métodos eficientes y de referencia al iniciar un diseño de lenguaje de patrones. En la investigación se ha incorporado los patrones de diseño abstract factory, singleton, adapter, decorator, facade, iterator, observer, state, y strategy en la fase de diseño, donde la segunda hipótesis especifica ha sido contrastada afirmativamente, aportando al desarrollo del diseño que mejora la calidad del producto software.

Pereira (2013), en la tesis "Principios de diseño SOLID aplicados para la mejora de código fuente en sistemas orientados a objetos", afirma que la calidad de productos software, está compuesto por el código fuente producido, hay diversas formas mejorar el código, entre ellos está los "Principios SOLID de diseño", se estudia mediante las métricas de la norma ISO/IEC 9126 para calidad de productos software, se concluye que los principios SOLID traen una percepción diferente de los problemas que afectan el código fuente de sistemas desarrollados y la calidad. En la investigación se ha incorporado los principios SOLID, que son; principio de responsabilidad única (SRP), principio abierto cerrado (OCP), principio de sustitución de Liskov (LSP), principio de segregación de interfaces (ISP), principio de inversión de dependencias (DIP), estos principios con los que se intervine la metodología ágil Iconix, generan mayor calidad de producto software, según los expertos, perteneciendo a la tercera hipótesis específica, contrastada afirmativamente.

Huanca (2015), en la tesis "Revisión sistemática de la calidad del software en prácticas ágiles", indica que el desarrollo de software ágil representa un alejamiento importante de los

enfoques tradicionales con planificación detallada, los resultados de estudios empíricos para la evaluación de la calidad en prácticas ágiles según el estándar ISO/IEC 25010, sugieren que las prácticas ágiles pueden ayudar a mejorar la calidad de productos software si son aplicadas correctamente. Se ha usado las normas NTP ISO/IEC 9126 y sus extensiones para evaluar la calidad del software con la metodología ágil Iconix, que para todos los casos presenta calidad bueno a muy bueno según los expertos, para las diversas métricas utilizadas; no se usa la norma ISO/IEC 25010, porque no existen métricas maduras para la mayoría de atributos de los producto software.

Samamé (2013), en la tesis titulada "Aplicación de una metodología ágil en el desarrollo de un sistema de información", usó la metodología ágil programación extrema y concluye: se aplicó Java y XML, se obtiene un sistema con portabilidad, la filosofía de programar y probar es un gran aporte, sintetizar o refactorizar el código genera un código más limpio y fácil de mantener.

Salvador (2013), en la tesis "Una revisión sistemática de usabilidad en metodologías ágiles", afirma que se han aplicado técnicas de evaluación de usabilidad en el desarrollo de software con metodologías ágiles, para mejorar la calidad de productos software, los resultados indican con mayor frecuencia que: prototipo rápido (40%), indagación individual (37%), pruebas formales de usabilidad (25%) y evaluaciones heurísticas (18%). Respecto a la característica usabilidad se ha evaluado las sub características: entendibilidad, facilidad de aprendizaje y operabilidad; que corresponde al análisis de requisitos y diseño de la metodología ágil Iconix intervenida y tiene relación con la primera y segunda hipótesis específica.

VI. CONCLUSIONES

Las conclusiones sobre las contrastación de las hipótesis específicas y general que se presentan a continuación fueron contrastadas con la pruebas de inferencia: T de Student, la prueba de homocedasticidad de varianza y análisis de varianza evaluadas con un 95% de confianza y 5% de significancia.

Primera conclusión

La modificación a la fase de análisis de requisitos mediante la inclusión de ingeniería de requisitos, mejora significativamente la calidad del producto software.

Segunda conclusión

La variación a la fase de diseño mediante la inclusión de patrones de diseño, mejora significativamente la calidad del producto software.

Tercera conclusión

La transformación a la fase de implementación mediante la inclusión de técnicas de programación orientada a objetos, como es SOLID, mejora significativamente la calidad del producto software.

Cuarta conclusión

Los resultados revelan que la adaptación de cambios a las fases de análisis de requisitos, diseño e implementación, en la metodología ágil Iconix con la inclusión de técnicas de ingeniería de software, mejoran significativamente la calidad del producto software.

VII. RECOMENDACIONES

Primera recomendación

Se debe investigar el uso de la metodología agil Iconix intervenida para del desarrollo de software con grupos de trabajo distribuido, porque la tendencia del trabajo futuro de un desarrollador de software es por Internet.

Segunda recomendación

En la línea de investigación de calidad del producto software, se debe estudiar métricas para evaluar la calidad del producto, específicamente cuando se realiza mantenimiento de software al haber desarrollado el producto software aplicando los patrones de diseño propuestos en el presente estudio.

Tercera recomendación

En la línea de investigación de desarrollo de software ágil, se debe estudiar la intervención a procesos agiles orientados para aplicaciones en móviles que genere mayor calidad del producto software.

VIII. REFERENCIAS BIBLIOGRÁFICAS

1. Alliance, Agile. (2001). *Agile manifesto*. Recuperado de https://www.agilealliance.org/agile101/the-agile-manifesto/. [Acceso octubre 2017].
2. Alexander, C. Ishikawa, S. Silverstein, M. Jacobson, M. Fiksdahl-King, I. y A. Shlomo, (1977). *A pattern language*. First Edition. New York: Oxford University Press.
3. Azadegan, A., Papamichail, K., & Sampaio, P. (2013). Applying collaborative process design to user requirements elicitation: A case study. *Computers in Industry, 64* (2013), 798–812.
4. Aurum, A., y Wohlin, C. (2005). *Managing both Individual and Collective participation in Software*. Berlin, Alemania: Editorial Heidelberg.
5. Banco Mundial (2018). *Atlas de los objetivos de desarrollo sostenible 2018: Basado en los indicadores del desarrollo mundial*, Washington, DC, Banco Mundial. DOI: 10.1596/978-1-4648-1250-7.
6. Barlow, J., Giboney, J., Keith, M., Wilson, D., & Schutzler, R, (2011). Overview and guidance on agile development in large organizations. *Communications of the Association for Information Systems, 29* (1), 25–44.
7. Baruah, N. (2015). Requirement Management in Agile Software Environment. *Procedia Computer Science.* 62, 81 – 83.
8. Belfo, F. (2012). Software Requirements Management through the Lenses of People, Organizational and Technological Dimensions. *International Journal of Web Portals*, 4(3):47-61.
9. Blé, C. (2010). *Diseño ágil con TDD*. Primera Edición. Madrid, España: Editorial Creative Commons.
10. Britto, J. (2014). *Adaptación de un proceso de desarrollo de software basado en buenas prácticas*. Tesis de Maestría. Unidad Académica Multidisciplinaria de Comercio. Universidad Autónoma de Tamaulipas. México.
11. Booch, G. (1996). *Análisis y diseño orientado a objetos*. Segunda Edición. Editorial: AddisonWesley. Díaz de Santos.
12. Bona, C. (2002). *Avaliação de processos de software: um estudo de caso em xp e iconix*. Tesis de Maestría. Programa de Pós-Graduação em Engenharia de Produção da Universidade Federal de Santa Catarina. Brasil.

13. Cano, C. (2015) *Revisión sistemática de comparación de modelos de procesos software*. Tesis de Maestría. Escuela de Posgrado. Pontificia Universidad Católica del Perú. Perú.

14. Chandel, M. (2018). *Cracking spring microservices interviews: java, spring boot y spring cloud*. Kindle Edition. Editorial Shunya Foundation.

15. Carrizo, D. & Rojas, J. (2016). Clasificación de prácticas en educción de requisitos en desarrollos ágiles: Un mapeo sistemático, *Ingeniare*. *24*(4), 654-662.

16. Cockburn, A. y Highsmith, J. (2001). *Agile software development: The people factor*. IEEE Computer.

17. Christou, I., Ponis, S., Palaiologou, E. (2010). Using the agile unified process in banking. *IEEE Software*, 72–79.

18. Delechamp, F. y Laugie, H. (2016). *Desarrolle una aplicación con java y eclipse*. Barcelona, España: Editorial ENI.

19. Díaz, J. (2001). *Lineamientos para la determinación del perfil de calidad de un producto de software*. Tesis de Maestría. Programas de Posgrado en Electrónica, Computación, Información y Comunicaciones. Instituto Tecnológico y de Estudios Superiores de Monterrey. México.

20. Daneva, M., van der Veen, E., Amrit, C., Ghaisas, S., Sikkel, K., Ajmeri, A., Ramteerthkar, U., y Wieringa, Smita. (2012). Agile requirements prioritization in large-scale outsourced system: An empirical study. *The Journal of Systems and Software. 86, 1333 – 1353.*

21. Damian, D., Kwan, I., y Marczak, S. (2010). Requirements-driven collaboration: Leveraging the invisible relationships between requirements and people. *Collaborative software engineering*, 57–76. Berlin: Springer.

22. Domínguez-Mayo, F., Escalona, M., Mejías, M., Ross, & Staples, G. (2012) Quality evaluation for model-driven web engineering methodologies. *Information and Software Technology, 54* (2012) 1265–1282.

23. Figueroa, R. Solis, C. y Cabrera, A. (2007). *Metodologías tradicionales vs. metodologías agiles*. Escuela de Ciencias en Computación. Universidad Técnica Particular de Loja. Ecuador.

24. Freeman, E.; Freeman, E.; Sierra, K. y B. Bates, (2009). *Head first: Desing pattenrs*. First Edition. California, USA: O'Reilly Media.

25. Fréderic D. y Laugie, H. (2016). *Desarrolle una aplicación con java y eclipse*.

Barcelona, España: Editorial ENI.

26. Freitas, I., da Mota, P., O'Leary, P., Santana, E. y Romero, S. (2007). Using a multi-method approach to understand Agile software product lines. *Information and Software Technology, 57,* 525–542.

27. Gamma, E.; Helm, R.; Jhonson, R. y Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Reimpresion 37. United States of America: Editorial Addison-Wesley.

28. García, F. y Pardo, C. (1998). *Introducción al análisis y diseño orientado a objetos*. Salamanca, España: Editorial América Ibérica.

29. Gary, K., Enquobahrie, A., Ibanez, L., Cheng, P., Yaniv, Z., Cleary, K., Kokoori, Muffih, S. y Heidenreich, J. (2011) Agile methods for open source safety-critical software. *Software: Practice and Experience, 41*, 945–962.

30. Grewal, H. y Maurer, F. (2007) Scaling agile methodologies for developing a production accounting systems for the oil and gas industry. *In: AGILE 2007, Washington, DC. IEEE*, 309–315.

31. Gómez, S. (2014). *Ingeniería del software*. Madrid, España: Editorial Delta

32. Guerrero, A., Suárez, J. y Gutiérrez, L. (2013) Patrones de Diseño en el contexto de Procesos de Desarrollo de Aplicaciones Orientadas a la Web. *Investigación Información Tecnológica, 24* (3), 103-114.

33. Jacobson, I.; Booch, G. y J. Rumbaugh, (2000) *El proceso unificado de desarrollo de software*. Primera Edición. Madrid: Pearson Educación S.A.

34. Jalali, S., Wohlin, C. (2010) Agile practices in global software engineering – A systematic map. In: IEEE International Conference on Global Software Engineering. Recuperado de http://www.wohlin.eu/icgse10a.pdf.

35. Jiménez, V., Tello, W. y Rios, J. (2014) .Lenguajes de Patrones de Arquitectura de Software Una Aproximación Al Estado del Arte. *Scientia Et Technica, 19* (4), 371-376.

36. Jurado, C. (2010). *Diseño ágil con TDD*. Bogotá, Colombia: SafeCreative.

37. Hansen, K., Jonasson, K. y H Neukirchen (2011). An empirical study of software architectures' effect on product quality. *Journal of Systems and Software, 84* (7), 1233-1243.

38. Highsmith, J. (2002) *Agile software development ecosystems*. Boston, USA: Addison Wesley.

39. Huanca, L. (2015) *Revisión sistemática de la calidad de software en prácticas*

agiles. Tesis de Maestría. Escuela de Posgrado. Pontificia Universidad Católica del Perú. Perú.

40. Holvite, J., Licorisch, S., Spinola, R., Hirinsalmy, S., MAcDonell, S., Mendes, T., Buchan, J. y V. Leppänen (2017). Technical debt and agile software development practices and processes: An industry practitioner survey. *Information and Software Technology*.
41. IEEE-CS/ACM. (1999). Joint task force on software engineering ethics and professional practices. Recuperado de http://www.acm.org/serving/se/code.htm
42. IEEE (2010). ISO/IEC 24765:2010 Systems and software engineering vocabualry. USA.
43. IEEE. (1998). Recommended practice for software requirements specifications, Institute of Electrical and Electronics Engineers, New York, USA, Std 830.
44. Inayat, I., Salwah, S., Marczak, S. y Daneva, M. (2014). A systematic literature review on agile requirements engineering practices and challenges. *Computers in Human Behavior, 51,* 915–929.
45. ISO 9000 (2005) Quality management systems - fundamentals and vocabulary. Geneva, Switzerland.
46. ISO/IEC 25010. (2011). Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Ginebra, Suiza.
47. ITU. (2015). *Stocktaking report 2015*. International Telecommunication Union, Geneva. Recuperado de http://handle.itu.int/11.1002/pub/80b519f2-en.
48. Karam, L. (2017). *Principios SOLID en la programación orientada a objetos*. Recuperado de https://apiumhub.com/es/tech-blog-barcelona/principios-solid/.
49. Katasonov, A. y Sakkinen, M. (2005). Requirements quality control: A unifying framework. *Requirements Engineering, 11*(1), 42–57.
50. Kitchenham, B. y Pfleeger, S. (1996). Software Quality: The Elusive Target. IEEE Software.
51. Kendall, R., Mark, A., Squires, S., Halverson, C. (2010). Physics-based code development project. *Computing in Science and Engineering, 12* (3), 22–27.
52. Kolfschoten, G.y Rouwette, E. (2006). Choice criteria for facilitation techniques: a preliminary classification. Conference on group decision and negotiation.

Universtatsverlag Karlsruhe, 1–8.

53. Kolfschoten, G. y De Vreede, G. (2007). *The collaboration engineering approach for designing collaboration processes*. Heidelberg: Springer.

54. Kupiainena, E., Mäntyläa, M. y Itkonena, J. (2015). Using metrics in Agile and Lean Software Development, A systematic literature review of industrial studies. *Information and Software Technology, 62*, 143–163.

55. Kumar, A. (2015). Investigating the penalty reward calculus of software users and its impact on requirements prioritization. *Information and Software Technology 65*, 56–68.

56. Leiva, I. y Villalobos, M. (2015). Método ágil híbrido para desarrollar software en dispositivos móviles. *Ingeniare, 23*(3), 473-488.

57. Leiva, A. (2016). *Principio de responsabilidad única SOLID 1ª parte*. Recuperado de https://devexperto.com/principio-responsabilidad-unica/. [Consultado Julio 2018]

58. Leiva, A. (2016). *Principio de sustitución de Liskov SOLID 3ª parte*. Recuperado: de https://devexperto.com/principio-de-sustitucion-de-liskov/. [Consultado julio 2018]

59. Liskov, B. y Wing, J. (1994). A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems, 16*(6).

60. Mainkar, P. (2017). *Expert android programming*. Birmingham, Reino Unido: Ediciones Packt.

61. Maiden, M. (1998). Problem domain categories in requirements engineering. *International Journal of Human–Computer Studies, 3*, 281–304.

62. Martin, R. (1996). *The liskov subtitution principle C++*. Report 8, nro. 3. p. 14-23.

63. Martin, R. (1996). *The dependency inversion principle C++*. Report 6, nro. 3. p. 61-66.

64. Martin, R. (2002). *Agile software development: principles, patterns, and practices*. NJ, USA: Pearson Education.

65. Martin, R. y Martin M. (2006). *Agile principles, patterns and practices in C#*. USA: Prentice Hall.

66. Martin, R. (2000). *Design principles and desing patterns*. Obtenido de <https://www.objectmentor.com/> [Consultado Agosto 2018].

67. Martin, R. (2003). *Agile software development: Principles, patterns, and*

practices. USA: Prentice Hall.

68. Medina, J. (2013). *Principios programación orientada a objetos*. España: Editorial Wiley.

69. Meyer, B. (1997). *Object oriented software construction*. Second Edition. Santa Barbara, California: Interactive Software Engineering Inc.

70. Naik, K. y Tripaty, P. (2008). *Software testing and quality assurance: Theory and practice*. First Edition. New Jersey, USA: John Wiley & Sons Inc.

71. Nema, W., Rodriguez, P., Oivo, M. y Tosun, A. (2016). Analyzing concept technical debt incontext agile software development, systematic literature review. *Information and Software Technology, 82*, 139–158.

72. Nerur, S. et al, (2005) Challenges of migrating to agile methodologies. *Communications of the ACM 48*, 5.

73. Nuseibeh, B. y Easterbrook, S. (2000). Requirements engineering: a roadmap. *Conference on the Future of Software Engineering*, ACM, New York, 35–46.

74. NTP-ISO/IEC 9126-1 (2004). *Ingeniería de software. Calidad del producto. Parte 1: Modelo de calidad.* Primera Edición. Lima: INDECOPI.

75. NTP-ISO/IEC 9126-2 (2004). *Ingeniería de software. Calidad del producto. Parte 2: Métricas externas*. Primera Edición. Lima: INDECOPI.

76. NTP-ISO/IEC 9126-3 (2005). *Ingeniería de software. Calidad del producto. Parte 3: Métricas internas*. Primera Edición. Lima: INDECOPI.

77. NTP-ISO/IEC 14598-1 (2005). *Tecnología de la información. Evaluación del producto. Parte 1: Visión general*. Primera Edición. Lima: INDECOPI.

78. NTP-ISO 14598-3 (2005). *Ingeniería de software. Evaluación del producto. Parte 3: Proceso para desarrolladores*. Primera Edición. Lima: INDECOPI.

79. Orantes, S. (2006). *Calidad de software en el uso de metodologías agiles para el desarrollo de software*. Centro de Investigación en Computación, Instituto Politécnico Nacional. México.

80. Palacios, A., García, R., Oliva, M. y Granollers, T. (2015). Exploración de patrones de interacción para su uso en la web semántica. *El profesional de la información, 24* (6), 749-758.

81. Pantoja, J. (2005). *Construindo softwares com qualidade e rapidez usando Iconix.* Recuperado de http://www.guj.com.br/article.show.logic?id=172. [Acceso junio 2017].

82. Pereira, A. (2013). *Princípios SOLID de design aplicados na melhoria de código*

fonte em sistemas orientados a objetos. Tesis. Universidade Federal de Lavras. Brasil.

83. Pijierro, M. (2017). *Principios SOLID: Principio de responsabilidad única*. Recuperado de https://medium.com/@mpijierro/principios-solid-principio-de-responsabilidad-%C3%BAnica-13eb4d5537c1. [Acceso diciembre 2017].
84. Pressman, R. (2002). *Ingeniería del software: Un enfoque práctico*. 5ta Edición. España: McGraw-Hill/Interamericana de España.
85. Pressman, R. (2005). *Ingeniería del software: Un enfoque práctico*. 6ta Edición. España: McGraw-Hill/Interamericana de España.
86. Prikladnicki, R. y Nicolas, J. (2010). Process models in the practice of distributed software development: a systematic review of literature. *Information and Software Technology, 52,* 779–791.
87. Robertson, S. (2001). Requirements trawling techniques for discovering requirements. *International Journal of Human–Computer Studies 55*, 405–421.
88. Robertson, S. y Robertson, J. (2006). *Mastering the requirements process*. Boston, Massachusetts: Pearson Education.
89. Rodríguez, D. y R. Harrison (1997). *Medición en la orientación a objetos*. University of Reading. School of Computer Science, Cybernetics & Electronic Engineering. UK.
90. Rodríguez, P., Musat, D. y Yagüe, A. (2010). Adopción de metodologías ágiles: un estudio comparativo entre España y Europa. *Revista Española de Innovación, Calidad e Ingeniería del Software*, 6(4).
91. Rosenberg, D. Stephens, M. y Collins-Cope, M. (2005). *Agile development with Iconix process: People, process, and pragmatism.* First Edition. EUA: Apress.
92. Rosenberg, D. y K. Scott (1999). *Use case driven object modeling with UML: A Practical Approach.* First Edition. EUA: Addison-Wesley.
93. Rosenberg, D. y K. Scott (2001). *Applyging use case driven object modeling with UML: Un Anotated e- Commerce Sample*. First Edition. United States: Addison-Wesley.
94. Rosenberg, D. y M. Stephens (2007) *Use case driven object modeling with UML: Theory and Practice.* First Edition. United States: Apress.
95. Sánchez, M. (2017). *Principios de la programación orientada a objetos*. España:

Ediciones 5.0.

96. Samamé, J. (2013). *Aplicación de una metodología ágil en el desarrollo de un sistema de información*. Tesis de Maestría. Escuela de Posgrado. Pontificia Universidad Católica del Perú. Perú.

97. Salvador, C. (2013). *Revisión sistemática de usabilidad en metodologías agiles*. Tesis de Maestría. Escuela de Posgrado. Pontificia Universidad Católica del Perú. Perú.

98. Santana, R. (2003) *Calidad en el desarrollo de software en el gobierno del estado de Tamaulipas y alternativas de mejora*. Tesis de Maestría. Unidad Académica Multidisciplinaria de Comercio. Universidad Autónoma de Tamaulipas. México.

99. Schmiedehausen, K. (2018). *Single page application architecture with angular*. Tesis. Technology and Communication. University of Applied Sciences.

100. Suarez. F. y Garzas, J. (2014). *I Jornada sobre calidad del producto software e ISO 25000*. Santiago de Compostela, España: Editorial: 233 Grados de TI S.L

101. Sun, X. y Zeng, Y. (2011). Formalization of design chain management using environment based design (EBD) theory. *Journal of Intelligent* Manufacturing, 24(3), 597–612.

102. Tarhan, A. y Yilmaz, S. (2013) Systematic analyses and comparison of development performance and product quality of Incremental Process and Agile Process. *Information and Software Technology, 56*, 477–494.

103. Scalone, F. (2006). *Estudio comparativo de los modelos y estándares de calidad del software*. Tesis de Magíster. Maestría en Ingeniería en Calidad. Universidad Tecnológica Nacional. Argentina.

104. Silveira, A. y Silva, F. (2015) Agile methods tailoring: A systematic literature review. *The Journal Systems and Software, 110*, 85-100.

105. UNESCO. (2015). *Informe sobre la ciencia: hacia 2030*. Recuperado de https://www.cooperacionib.org/informeciencia.php.

106. Sommerville, I. (2005). *Ingeniería del software*. Séptima Edición. España. Madrid: Pearson Educación.

107. UNESCO. (2017). Las piedras angulares para la promoción de sociedades del conocimiento inclusivas. Paris. Francia: Editorial Unesco

108. Wagner, S. (2013). *Software product quality control*. First Edition. Stuttgart Germany: Springer – Verlag Berlin Heidelberg.

109. Witold, S. (2014). *Software quality engineering: A practitioner's approach*. First Edition. IEEE. Montreal, Canada: John Wiley & Sons, Inc.

110. Wnuk, K., Regnell, B., y Karlsson, L. (2009). Feature Transition Charts for Visualization of Cross-Project Scope Evolution in Large-Scale Requirements Engineering for Product Lines, 89–98.

111. Yanga, C., Lianga, P. y Avgerioub, P. (2015). A systematic mapping study on the combination of software architecture and agile development. *The Journal of Systems and Software, 111*, 157–184.

112. Zhao, L., Macaulay, L., Adams, J. y Verscheuren, P. (2007). A pattern language for designing e. business architecture. Sience Direct.

IX. ANEXOS

Anexo 1. Matriz de Consistencia

TITULO: METODOLOGÍA ÁGIL ICONIX EN LA CALIDAD DEL PRODUCTO SOFTWARE, LIMA, 2017.

PROBLEMAS	OBJETIVOS	HIPÓTESIS	VARIABLES E INDICADORES	MÉTODO DE INVESTIGACIÓN
1. **Problema Principal** ¿De qué manera adaptar la metodología ágil Iconix para mejorar la calidad del producto software, Lima, 2017?. **2. Problemas Secundarios** a. ¿Cómo modificar la fase de análisis de requisitos para mejorar la calidad del producto software?. b. ¿Cómo variar la fase de diseño para mejorar la calidad del producto software?. c. ¿Cómo transformar la	**1. Objetivo General** Incluir técnicas de ingeniería de software a la metodología ágil Iconix mediante técnicas e instrumentos con la finalidad de mejorar la calidad del producto software, Lima, 2017. 2. **Objetivos Específicos** a. Desarrollar la fase de análisis de requisitos mediante la inclusión de ingeniería de requisitos a fin de mejorar la calidad del producto software. b. Desarrollar la fase de	**1. Hipótesis General** Si adaptamos la metodología ágil Iconix con la inclusión de técnicas de ingeniería de software, entonces se mejora la calidad del producto software, Lima, 2017. **2. Hipótesis Específicas** a. Si modificamos la fase de análisis de requisitos mediante la inclusión de ingeniería de requisitos, entonces se mejora la calidad del producto software.	**1. Variable Independiente** X: Metodología ágil Iconix. **Indicadores** X1: Análisis de requisitos X2: Diseño X3: Implementación **2. Variable Dependiente** Y: Calidad del producto software.	**1. Tipo de Investigación** Prospectivo, longitudinal y analítico. **2. Nivel de Investigación** Aplicativo. **3. Método** a. Deductivo e Inductivo b. Análisis y Síntesis c. Interpretación d. Estadístico **4. Diseño** Cuasi experimental. **5. Población** Esta compuesta por todos los productos software de análisis de requisitos, diseño e

fase de implementación para mejorar la calidad del producto software?.	diseño mediante la inclusión de patrones de diseño a fin de mejorar la calidad del producto software. c. Desarrollar la fase de implementación mediante la inclusión de técnicas de programación orientada a objetos con la finalidad de mejorar la calidad del producto software.	b. Si variamos la fase de diseño mediante la inclusión de patrones de diseño, entonces se mejora la calidad del producto software. c. Si transformamos la fase de implementación mediante la inclusión de técnicas de programación orientada a objetos, entonces se mejora la calidad del producto software.	**Indicadores** Y1: Calidad interna Y2: Calidad externa	implementación de la metodología ágil Iconix, con y sin intervención. **6. Muestra** La muestra se ha tomado por juicio de expertos, de los productos software de análisis de requisitos, diseño e implementación de la metodología ágil Iconix, con y sin intervención. **7. Técnicas** a. Encuesta. b. Análisis estadístico. **8. Instrumentos** a. Cuestionario. b. Paquete estadístico SPSS.

Anexo 2. Instrumentos para la toma de Datos

Instrumento Pre test: Calidad de producto software aplicando la metodología ágil Iconix sin intervención

Señor Experto:

En su respuesta tome en cuenta la abreviación de las fases como sigue: Análisis de requisitos (AR), diseño (D) e implementación (I); luego, sírvase evaluar la calidad del producto software generado con la metodología ágil Iconix sin intervención y dar respuesta a las preguntas formuladas, según la escala de Likert.

Variable: Calidad del producto software

(5) Totalmente de acuerdo (4) Parcialmente de acuerdo (3) Ni de acuerdo ni en desacuerdo (2) Parcialmente en desacuerdo (1) Totalmente en desacuerdo

Dimensión: Calidad interna y externa

Característica (Sub Característica)	**Producto software**	**Métrica Interna**	**Métrica Externa**	**Pregunta**	**5**	**4**	**3**	**2**	**1**
Funcionalidad (Aplicabilidad)	1. Requisitos funcionales y no funcionales. 2. Descripción de casos de uso. 3. Interfaz gráfica de usuario (GUI). 4. Código fuente.	1. Adecuación funcional. 2. Integridad de implementación funcional. 3. Cobertura de la implementación funcional.	1. Adecuación funcional 2. Integridad de implementación funcional 3. Cobertura de implementación funcional	**1.** ¿Presenta adecuación funcional necesaria? (AR)					
				2. ¿Presenta integridad de implementación funcional necesaria? (I)					
				3. ¿Presenta cobertura de la implementación funcional necesaria? (I)					
Funcionalidad	1. Código fuente.	1. Exactitud de	1. Exactitud de	**4.** ¿Presenta exactitud de cálculos? (I)					

(Precisión)	2. Pruebas unitarias.	cálculos.	cálculos.						
Fiabilidad (Madurez)	1. Reporte de pruebas unitarias. 2. Reporte de pruebas de integración.	1. Eliminación de fallas.	1. Densidad de fallas contra los casos de prueba. 2. Madurez de la prueba	**5.** ¿Presenta eliminación de fallas? (I)					
				6. ¿Presenta Densidad de fallas contra los casos de prueba? (AR)					
				7. ¿Presenta madurez de la prueba? (AR)					
Fiabilidad (Tolerancia a fallas)	1. Requisitos funcionales y no funcionales. 2. Reporte de pruebas unitarias.	1. Prevención de operación incorrecta.	1. Prevención de operación incorrecta.	**8.** ¿Presenta prevención de operación incorrecta? (D)					
Usabilidad (Entendibilidad)	1. Requisitos funcionales y no funcionales. 2. Descripción de casos de uso. 3. Interfaz gráfica de usuario (GUI).	1. Función de comprensión	1. Comprensión de entradas y salidas	**9.** ¿Presenta función de comprensión? (D)					
				10. ¿Presenta comprensión de entradas y salidas? (D)					
Usabilidad (Facilidad de aprendizaje)	1. Requisitos funcionales y no funcionales. 2. Lista de casos de uso. 3. Descripción de casos de uso.	1. Integridad de la documentación del usuario y/o facilidad de ayuda	1. Facilidad de aprender a realizar una tarea en uso	**11.** ¿Presenta integridad de la documentación del usuario y/o facilidad de ayuda? (D)					
				12. ¿Presenta facilidad de aprender a realizar una tarea en uso? (I)					
Usabilidad (Operabilidad)	1. Descripción de casos de uso. 2. Código fuente. 3. Interfaz gráfica de	1. Claridad de mensajes. 2. Claridad de la interfaz.	1. Entendibilidad del mensaje en uso	**13.** ¿Presenta claridad de mensajes? (AR)					
				14. ¿Presenta claridad de la interfaz? (AR)					
				15. ¿Presenta entendibilidad del mensaje					

	usuario (GUI).			en uso? (I)					
Eficiencia (Utilización de recursos)	1. Código fuente. 2. Técnicas de codificación.	1. Utilización de memoria.		**16.** ¿Presenta utilización de memoria? (I)					
Eficiencia (Comportamiento en el tiempo)	1. Casos de pruebas. 2. Reporte de pruebas de integración.		1. Rendimiento	**17.** ¿Presenta rendimiento? (I)					
Facilidad de Mantenimiento (Cambiabilidad)	1. Código fuente.	1. Registro de cambios		**18.** ¿Presenta rendimiento? (I)					
Facilidad de Mantenimiento (Estabilidad)	1. Diagrama de clases 2. Código fuente.	1. Impacto de la modificación	1. Ratio de éxito de cambios	**19.** ¿Presenta impacto de la modificación? (D)					
				20. ¿Presenta ratio de éxito de cambios? (D)					
Portabilidad (Adaptabilidad)	1. Arquitectura técnica (Diagrama de despliegue y de componentes).	1. Adaptabilidad al entorno organizacional (adaptabilidad a la organización y a la infraestructura de la misma)		**21.** ¿Presenta adaptabilidad a la organización y a la infraestructura de la misma? (D)					
Portabilidad (Instalabilidad)	1. Procedimiento de instalación.		1. Facilidad de reinstalación	**22.** ¿Presenta facilidad de reinstalación? (AR)					

Instrumento Post test: Calidad de producto software aplicando la metodología ágil Iconix con intervención

Señor Experto:

Sírvase evaluar la calidad del producto software generado con la metodología ágil Iconix con intervención aplicando; ingeniería de requisitos, patrones de diseño y técnicas de programación orientada a objetos (SOLID) y dar respuesta a las preguntas formuladas, según la escala de Likert.

Variable: Calidad del producto software

(5) Totalmente de acuerdo (4) Parcialmente de acuerdo (3) Ni de acuerdo ni en desacuerdo (2) Parcialmente en desacuerdo (1) Totalmente en desacuerdo

Dimensión: Calidad interna y externa

Característica (Sub Característica)	**Producto software**	**Métrica Interna**	**Métrica Externa**	**Pregunta**	**5**	**4**	**3**	**2**	**1**
Funcionalidad (Aplicabilidad)	1. Requisitos funcionales y no funcionales. 2. Descripción de casos de uso. 3. Interfaz gráfica de usuario (GUI). 4. Código fuente.	1. Adecuación funcional. 2. Integridad de implementación funcional. 3. Cobertura de la implementación funcional.	1. Adecuación funcional 2. Integridad de implementación funcional 3. Cobertura de implementación funcional	**1.** ¿Presenta adecuación funcional necesaria? (AR)					
				2. ¿Presenta integridad de implementación funcional necesaria? (I)					
				3. ¿Presenta cobertura de la implementación funcional necesaria? (I)					
Funcionalidad (Precisión)	1. Código fuente. 2. Pruebas unitarias.	1. Exactitud de cálculos.	1. Exactitud de cálculos.	**4.** ¿Presenta exactitud de cálculos? (I)					
Fiabilidad	1. Reporte de	1. Eliminación de	1. Densidad de	**5.** ¿Presenta eliminación de fallas? (I)					

(Madurez)	pruebas unitarias. 2. Reporte de pruebas de integración.	fallas.	fallas contra los casos de prueba. 2. Madurez de la prueba	**6.** ¿Presenta Densidad de fallas contra los casos de prueba? (AR)					
				7. ¿Presenta madurez de la prueba? (AR)					
Fiabilidad (Tolerancia a fallas)	1. Requisitos funcionales y no funcionales. 2. Reporte de pruebas unitarias.	1. Prevención de operación incorrecta.	1. Prevención de operación incorrecta.	**8.** ¿Presenta prevención de operación incorrecta? (D)					
Usabilidad (Entendibilidad)	1. Requisitos funcionales y no funcionales. 2. Descripción de casos de uso. 3. Interfaz gráfica de usuario (GUI).	1. Función de comprensión	1. Comprensión de entradas y salidas	**9.** ¿Presenta función de comprensión? (D)					
				10. ¿Presenta comprensión de entradas y salidas? (D)					
Usabilidad (Facilidad de aprendizaje)	1. Requisitos funcionales y no funcionales. 2. Lista de casos de uso. 3. Descripción de casos de uso.	1. Integridad de la documentación del usuario y/o facilidad de ayuda	1. Facilidad de aprender a realizar una tarea en uso	**11.** ¿Presenta integridad de la documentación del usuario y/o facilidad de ayuda? (D)					
				12. ¿Presenta facilidad de aprender a realizar una tarea en uso? (I)					
Usabilidad (Operabilidad)	1. Descripción de casos de uso. 2. Código fuente. 3. Interfaz gráfica de usuario (GUI).	1. Claridad de mensajes. 2. Claridad de la interfaz.	1. Entendibilidad del mensaje en uso	**13.** ¿Presenta claridad de mensajes? (AR)					
				14. ¿Presenta claridad de la interfaz? (AR)					
				15. ¿Presenta entendibilidad del mensaje en uso? (I)					
Eficiencia (Utilización de	1. Código fuente. 2. Técnicas de	1. Utilización de memoria.		**16.** ¿Presenta utilización de memoria? (I)					

recursos)	codificación.								
Eficiencia (Comportamiento en el tiempo)	1. Casos de pruebas. 2. Reporte de pruebas de integración.		1. Rendimiento	**17.** ¿Presenta rendimiento? (I)					
Facilidad de Mantenimiento (Cambiabilidad)	1. Código fuente.	1. Registro de cambios		**18.** ¿Presenta rendimiento? (I)					
Facilidad de Mantenimiento (Estabilidad)	1. Diagrama de clases 2. Código fuente.	1. Impacto de la modificación	1. Ratio de éxito de cambios	**19.** ¿Presenta impacto de la modificación? (D)					
				20. ¿Presenta ratio de éxito de cambios? (D)					
Portabilidad (Adaptabilidad)	1. Arquitectura técnica (Diagrama de despliegue y de componentes).	1. Adaptabilidad al entorno organizacional (adaptabilidad a la organización y a la infraestructura de la misma)		**21.** ¿Presenta adaptabilidad a la organización y a la infraestructura de la misma? (D)					
Portabilidad (Instalabilidad)	1. Procedimiento de instalación.		1. Facilidad de reinstalación	**22.** ¿Presenta facilidad de reinstalación? (AR)					

Anexo 3. Validación del Instrumento

Señor Experto, sírvase evaluar la validez del instrumento de acuerdo a la escala de Likert que se presenta asignado la siguiente puntuación según su criterio.

(1). Totalmente en desacuerdo (2). En desacuerdo (3). Ni de acuerdo ni en desacuerdo (4). De acuerdo (5). Totalmente de acuerdo

Indicadores	**Criterios**	**Puntuación**				
		(1)	(2)	(3)	(4)	(5)
Claridad	Está formulado con lenguaje apropiado					
Objetividad	Está expresado con preguntas objetivas observables					
Actualidad	Está adecuado al avance de la ciencia y la tecnología					
Organización	Tienen una organización lógica					
Suficiencia	Comprende los aspectos en calidad y cantidad					
Intencionalidad	Responde a los objetivos de la investigación					
Consistencia	Está basado en aspectos teóricos, científicos y técnicos					
Coherencia	Entre las dimensiones, indicadores, preguntas e índices					
Metodología	Responde a la operacionalización de la variable					
Pertinencia	Es útil para la investigación					

Correlaciones del Tau b de Kendall

El instrumento de recolección de datos diseñado para evaluar cada una de las metodologías propuestas, fue validado mediante la validez de contenido a través de 10 indicadores para medir la calidad de los ítems formulados, entre estos tenemos: la claridad, la objetividad, la actualidad, la organización, suficiencia, intencionalidad, consistencia, coherencia, metodología y pertinencia; los tres expertos consultados analizaron los ítems cuyas puntuaciones asignadas fueron procesadas con el Tau de

Kendall que evalúa las coincidencias entre las puntuaciones de los expertos, las correlaciones calculadas para el juicio de expertos son: $\tau = 0{,}583^*$; $\tau = 0{,}802^*$ y $\tau = 0{,}802^*$, que indican que existe una relación entre las calificaciones asignadas que varían de un rango de correlación moderada a una correlación alta, con mayor incidencia de la asociación alta, asimismo, las calificaciones porcentuales promedio de las calificaciones asignadas por los expertos es de 92,67% por instrumento, que indica que el instrumento cumple los requisitos para evaluar las fases de la Metodología Ágil Iconix sin intervención y la Metodología Ágil Iconix con intervención asociados a la Calidad del Producto Software.

Según las puntuaciones asignadas por los expertos que evaluaron el instrumento de recolección de datos que evalúa la Metodología Ágil Iconix con la Calidad del Producto Software, obtenemos la siguiente tabla**.**

Correlaciones del Tau b de Kendall

Tau_b de Kendall		Primer experto	Segundo experto	Tercer experto
Primer experto	Coeficiente de correlación	1,000	,583	,802*
	Sig. (bilateral)	.	,080	,016
	N	10	10	10
Segundo experto	Coeficiente de correlación	,583	1,000	,802*
	Sig. (bilateral)	,080	.	,016
	N	10	10	10
Tercer experto	Coeficiente de correlación	,802*	,802*	1,000
	Sig. (bilateral)	,016	,016	.
	N	10	10	10

* La correlación es significativa en el nivel 0,05 (bilateral).

Fuente: Instrumento de evaluación diseñado para los expertos

Anexo 4. Confiabilidad del Instrumento

La escala de medición para la calidad del producto software, consiste de 22 preguntas, según el estándar de la norma NTP ISO/IEC 9126 y sus extensiones, con cinco alternativas de respuestas cada uno. Previa a la recopilación de la información, se dará una breve información a los expertos acerca de los temas que se evalua, para que posteriormente complementen la escala de Likert de manera individual los expertos. Se espera una escala global para la calidad del producto software desarrollado con la metodología ágil Iconix, para la calidad interna y externa, con una fiabilidad según el coeficiente alpha de Crombach.

Índice de confiabilidad Alfa de Cronbach

Para evaluar la consistencia interna de la escala de medición a través de las puntuaciones asignadas por los expertos que miden los productos software mediante las variables Metodología Ágil Iconix con la Calidad del Producto Software, se ha calculado el estadístico como se muestra en la siguiente tabla.

Estadísticas de fiabilidad

Alfa de Cronbach	N de elementos
,930	22

Fuente: Instrumento de evaluación diseñado para medir las características de los Productos Software y la Calidad.

En forma correlativa, se evaluó la confiabilidad del instrumento de recolección de datos con el Alfa de Cronbach, cuyo coeficiente es de 0,93 para 22 ítems, este estadígrafo indica que el instrumento es altamente confiable, lo que quiere decir que los ítems del instrumento se combinan aditivamente generando una puntuación global, que mide apropiadamente la Metodología Ágil Iconix con la Calidad del Producto Software en la misma dirección, dado que los ítems están formulados en un sentido positivo.

Anexo 5. Tablas para el desarrollo de la Metodología Ágil Iconix sin Intervención

Luego de revisar las bases teóricas relacionadas al problema de investigación, para la metodología ágil Iconix sin intervención, se ha identificado: tareas, técnicas, productos software y responsables, que se muestra en las tablas siguientes.

Fase de análisis de requisitos de la metodología sin intervención

Tarea	Técnica	Producto software	Responsable
Identificar requisitos.	a. Entrevistas estructuradas y no estructuradas. b. Definir lo que el sistema debe hacer. c. Escribir al menos un caso de prueba para cada requisito.	a. Requisitos funcionales y no funcionales. b. Casos de prueba de aceptación.	Usuario Cliente Analista
Identificar objetos del mundo real Realizar el modelo de dominio	a. Identificar objetos clave del negocio b. Identificar objetos (sustantivos) en los requisitos funcionales y asignar al modelo de dominio c. Crear glosario de términos (sustantivos) d. Utilizar agregación y generalización	a. Glosario de términos b. Modelo de dominio inicial	Analista
Realizar prototipo de interfaz gráfica	a. Utilizar historia de eventos del usuario b. Utilizar los requisitos funcionales c. Diseñar interfaz gráfica básica	Prototipo de interfaz gráfica de usuario (GUI)	Programador Analista
Descubrir casos de uso	a. Utilizar requisitos funcionales b. Entrevistas	Lista de casos de uso	Usuario Cliente Analista
Modelar los casos de uso	a. Identificar roles y responsabilidades de actores b. Asociar actores con casos de uso c. Relacionar casos de uso d. Agrupar lógicamente casos de uso	a. Diagrama de casos de uso b. Paquete de casos de uso	Analista
Relacionar requisitos funcionales con los casos de uso	a. Asignar casos de uso a requisitos funcionales	Lista de relación entre requisitos funcionales, no funcionales y casos de uso	

Escribir el primer borrador de casos de uso	a. Utilizar glosario de terminos b. Asignar un nombre a cada interfaz grafica c. Utilizar la regla de dos párrafos d. Escribir el caso de uso como flujos de evento/respuesta e. Escribir el caso de uso con estructura sustantivo-verbo-sustantivo f. Escribir casos de uso en voz activa g. Reescribir los casos de uso	Primer borrador de casos de uso	

Fase de diseño de la metodología sin intervención

Tarea	Técnica	Producto software	Responsable
Reescribir el primer borrador para cada caso de uso	Desambiguar el caso de uso durante el análisis de robustez	Caso de uso reescrito	Analista
Identificar el primer corte de objetos para cada escenario del caso de uso	a. Copiar la descripción del caso de uso reescrito en el diagrama de robustez. b. Usar los objetos del modelo de dominio c. Crear un objeto interfaz por cada interfaz grafica d. Transformar verbos del caso de uso en objeto controlador e. Relacionar un caso de uso al diagrama de robustez cuando es invocado f. Utilizar las reglas para construir el diagrama de robustez	Diagrama de robustez	
Actualizar el modelo de dominio	a. Actualizar el modelo de dominio con nuevos objetos b. Asignar atributos a las clases entidad	Modelo de dominio actualizado	
Dividir modelo de dominio actualizado para	a. Coincidir las clases entidad del diagrama de robustez con parte del	Parte de modelo de dominio actualizado	Diseñador

cada caso de uso	modelo de dominio actualizado y dibujarlo		
Dibujar un diagrama de secuencia para cada caso de uso	a. Copiar la descripción del caso de uso b. Copiar objetos entidad, interfaz, y actores del diagrama de robustez c. Verificar que un mensaje del diagrama de secuencia es verbo en el caso de uso d. Hacer refactoring al diagrama de secuencia antes de codificar e. Crear el diagrama de clases	Diagrama de secuencia	Programador Diseñador
Actualizar el diagrama de clases para un caso de uso	a. Asignar operaciones a las clases a partir de mensajes del diagrama secuencia b. Establecer multiplicidad en las clases c. Depurar las clases, operaciones y atributos del diagrama de clases	Diagrama de clases	
Extraer controladores	a. Identificar controladores desde un diagrama de robustez	Lista de controladores	

Fase de implementación de la metodología sin intervención

Tarea	Técnica	Producto software	Responsable
Implementar la base de datos física	a. Escribir el script usando el diagrama de clases b. Ejecutar el script usando un DBMS	Base de datos física	Programador
Implementar código para clases entidad	a. Escribir o generar código fuente con una herramienta usando el diagrama de clases	Código fuente para clases entidad	Programador
Implementar código para las GUI	a. Generar código fuente usando una herramienta	Código fuente para GUI	Programador
Crear pruebas unitarias para cada controlador	a. Escribir código fuente para una prueba unitaria usando una herramienta	Prueba unitaria para cada controlador	Programador
Implementar código fuente para cada controlador	a. Escribir el código fuente siguiendo el flujo normal del diagrama de secuencia usando una herramienta b. Actualizar el diagrama de	Código fuente para cada controlador	Programador

	secuencia con la codificación		
Ejecutar pruebas unitarias para cada controlador	a. Ejecutar el módulo de cada prueba unitaria b. Modificar código fuente si la prueba unitaria muestra resultado incorrecto	Reporte de pruebas unitarias	Programador
Ejecutar pruebas de integración	c. Ejecutar la prueba de integración d. Modificar código fuente si la prueba de integración muestra resultado incorrecto	Reporte de pruebas de integración	Programador
Ejecutar pruebas de aceptación para cada caso de uso	a. Utilizar los casos de prueba de aceptación b. Ejecutar el módulo de un caso de uso c. Modificar código fuente si la prueba de aceptación muestra resultado incorrecto	Reporte de pruebas de aceptación	Programador Usuario Cliente

Anexo 6. Tablas para el desarrollo de la Metodología Ágil Iconix con Intervención

Luego de intervenir la metodología ágil Iconix mediante; ingeniería de requisitos, patrones de diseño y técnicas de programación orientada a objetos (SOLID), se ha identificado: tareas, técnicas, productos software y responsables, que se muestra en las tablas siguientes.

Fase de análisis de requisitos de la metodología con intervención

Tarea	Técnica	Producto software	Responsable
Obtener los requisitos de software	a. Lluvia de ideas b. Focus groups c. Análisis documental	Requisitos funcionales y no funcionales	a. Cliente b. Usuario final c. Experto de dominio d. Desarrollador
a. Clasificar los requisitos de software b. Categorizar los requisitos de software c. Evaluar la clasificación de requisitos de software d. Priorizar los requisitos de software	a. Sesiones de trabajo (Workshops) b. Votación de stakeholders	a. Requisitos funcionales y no funcionales clasificados b. Requisitos funcionales y no funcionales priorizados	a. Cliente b. Usuario final c. Experto de dominio d. Desarrollador
Crear casos de prueba	Escribir al menos un caso de prueba por cada requisito	Casos de prueba de aceptación	Usuario final
a. Identificar objetos entidad del mundo real b. Crear el modelo de dominio	a. Descubrir objetos clave del negocio b. Identificar sustantivos (objetos) en los requisitos y asignar al modelo de dominio c. Listar glosario de sustantivos	a. Glosario de sustantivos b. Modelo de dominio inicial	Analista
Realizar prototipo de interfaz gráfica	a. Utilizar historia de eventos del usuario b. Utilizar los requisitos funcionales	Prototipo de interfaz gráfica de usuario (GUI)	Programador Analista

	c. Diseñar interfaz gráfica básica		
a. Descubrir casos de uso b. Relacionar requisitos y casos de uso	a. Utilizar requisitos priorizados b. Asignar casos de uso a requisitos funcionales	Lista de relación entre requisitos funcionales, no funcionales y los casos de uso	Usuario Cliente Analista Experto de dominio
Modelar los casos de uso	a. Identificar roles y responsabilidades de actores b. Asociar actores con casos de uso c. Relacionar casos de uso d. Agrupar lógicamente casos de uso	a. Diagrama de casos de uso b. Paquete de casos de uso	Analista
Escribir el primer borrador de casos de uso	a. Utilizar glosario de sustantivos b. Asignar un nombre a cada interfaz grafica c. Utilizar la regla de dos párrafos d. Escribir el caso de uso como flujos de evento/respuesta e. Escribir el caso de uso con estructura sustantivo-verbo-sustantivo f. Escribir casos de uso en voz activa	Primer borrador de casos de uso	

Fase de diseño de la metodología con intervención

Tarea	Técnica	Producto software	Responsable
Reescribir el primer borrador para cada caso de uso	Revisar la descripción para cada caso de uso reescrito	Caso de uso reescrito	Analista
Identificar el primer corte de objetos para cada escenario del caso de uso	a. Usar los objetos del modelo de dominio b. Crear un objeto interfaz por cada interfaz grafica c. Transformar verbos del caso de uso en objeto	Objetos interfaz, entidad y controlador	

	controlador		
Actualizar el modelo de dominio	a. Actualizar el modelo de dominio con nuevos objetos b. Asignar atributos a las clases entidad	Modelo de dominio actualizado	
Crear familias de objetos relacionados	a. Declarar una interfaz para operaciones que crea objetos producto abstracto b. Implementar las operaciones para crear objetos producto concreto c. Declarar una interfaz para un tipo de objeto producto d. Definir un objeto producto para ser creado por AbstractFactory	Patrón abstract factory	a. Arquitecto de software b. Programador
Crear clase con instancia única	a. Definir una operación Instance b. Crear objetos únicos incluido su propia instancia	Patrón Singleton	
Crear clase reutilizable que coopere con clases no relacionadas	a. Definir la interfaz de dominio específica b. Definir una interfaz existente que necesita ser adaptada c. Adaptar la interfaz	Patrón Adapter	
Añadir dinámicamente responsabilidades a un objeto	a. Definir la interfaz de objetos para adicionar responsabilidades b. Definir un objeto para adicionar responsabilidades c. Mantener una referencia a un objeto component d. Definir una interfaz que se ajusta a su interfaz component e. Adicionar responsabilidades al objeto concreto decorator	Patrón Decorator	
Crear una interfaz única para un conjunto de interfaces de subsistemas	a. Definir la clase compiler responsable ante una petición b. Delegar peticiones de clientes a objetos del	Patrón Facade	

	subsistema c. Definir las clases subsistemas d. Implementar la funcionalidad del subsistema e. Realizar las tareas encargadas por compiler		
Crear acceso secuencial a los elementos de un objeto	a. Definir una interfaz de acceso a los elementos b. Implementar la interfaz c. Mantener la posición actual del recorrido d. Definir la interfaz para crear un objeto Iterator e. Implementar la interfaz de creación de Iterator	Patrón Iterator	
Definir dependencia de uno a muchos entre objetos	a. Crear objeto subject visto por observers b. Definir observer que actualiza los objetos por cambios en subject c. Almacenar estado de interés de objetos ConcreteObserver d. Mantener referencia para un objeto ConcreteSubject e. Implementar la interfaz de actualización de Observer	Patrón Observer	
Modificar comportamiento de un objeto cuando cambia su estado interno	a. Definir la interfaz de interés para los clientes b. Mantener una instancia de una subclase c. Definir una interfaz que encapsula el comportamiento de Context	Patrón State	
Definir una familia de algoritmos encapsulados e intercambiables	a. Declarar una interfaz (Strategy) común para todos los algoritmos b. Implementar (ConcreteStrategy) el algoritmo usando la interfaz Strategy c. Configurar (Context) con un objeto ConcretStrategy	Patrón Strategy	
Dibujar un	a. Copiar la descripción del	Diagrama de	Programador

diagrama de secuencia para cada caso de uso	caso de uso b. Dibujar objetos entidad, controlador e interfaz c. Verificar que un mensaje del diagrama de secuencia es verbo en el caso de uso d. Hacer refactoring al diagrama de secuencia antes de codificar e. Crear el diagrama de clases	secuencia	Diseñador
Actualizar el diagrama de clases para un caso de uso	a. Asignar operaciones a las clases a partir de mensajes del diagrama secuencia b. Establecer multiplicidad en las clases c. Depurar las clases, operaciones y atributos del diagrama de clases	Diagrama de clase	
Extraer controladores	Identificar controladores desde el diagrama de secuencia	Lista de controladores	

Fase de implementación de la metodología con intervención

Tarea	Técnica	Producto software	Responsable
Implementar la base de datos física	a. Escribir el script usando el diagrama de clases b. Ejecutar el script usando un DBMS	Base de datos física	Programador
Aplicar SRP	a. Identificar clases con responsabilidad única b. Identificar clases con más de una responsabilidad c. Crear clases con responsabilidad única d. Generar código para las clases con responsabilidad única	Código fuente para clases con responsabilidad única	Analista Programador
Aplicar OCP	a. Crear clases abstractas cuando se necesita b. Identificar clases que presentan herencia y composición c. Generar código para clases con herencia y composición	Código fuente para clases abierto y cerrado	Analista Programador

	d. No modificar código para adicionar funcionalidad e. Aplicar patrones strategy y template si es necesario f. Escribir código nuevo para adicionar funcionalidad		
Aplicar LSP	a. Identificar clases abstractas (tipo) b. Identificar clases concretas (subtipo) c. Modelar herencia entre clases d. Definir contratos de diseño de la clase base (tipo) e. Escribir código para implementar la herencia	Código fuente con principios Liskov	Analista Programador
Aplicar LSP	a. Identificar clases con varios clientes b. Crear interfaces específicas para cada cliente c. Agregar nuevas interfaces para nuevos requisitos d. Aplicar la separación por delegación con el patrón adapter cuando es necesario e. Aplicar la separación por herencia múltiple cuando es necesario f. Escribir código para implementar los métodos específicos	Código fuente con principio de segregación de interfaces	Analista Programador
Aplicar DIP	a. Identificar clases concretas para crear clases abstractas b. Analizar las dependencias entre módulos de código c. Analizar dependencias para inyección por constructor d. Analizar dependencias para inyección por interfaces e. Analizar dependencias para inyección por método f. Generar código con	Código fuente con principio de inversión de dependencias	Analista Programador

	inyección de dependencias		
Implementar código para las GUI	Generar código fuente usando una herramienta	Código fuente para GUI	Programador
Crear pruebas unitarias para cada controlador	Escribir código fuente para una prueba unitaria usando una herramienta	Prueba unitaria para cada controlador	Programador
Implementar código fuente para cada controlador	a. Escribir el código fuente siguiendo el flujo normal del diagrama de secuencia usando una herramienta b. Actualizar el diagrama de secuencia con la codificación	Código fuente para cada controlador	Programador
Ejecutar pruebas unitarias para cada controlador	a. Ejecutar el módulo de cada prueba unitaria b. Modificar código fuente si la prueba unitaria muestra resultado incorrecto	Reporte de pruebas unitarias	Programador
Ejecutar pruebas de integración	a. Ejecutar la prueba de integración b. Modificar código fuente si la prueba de integración muestra resultado incorrecto	Reporte de pruebas de integración	Programador
Ejecutar pruebas de aceptación para cada caso de uso	a. Utilizar los casos de prueba de aceptación b. Ejecutar el módulo de un caso de uso c. Modificar código fuente si la prueba de aceptación muestra resultado incorrecto	Reporte de pruebas de aceptación	Programador Usuario Cliente

Anexo 7. Patrones de Diseño Adaptado

A. Estructura del patrón Abstract Factory adaptado

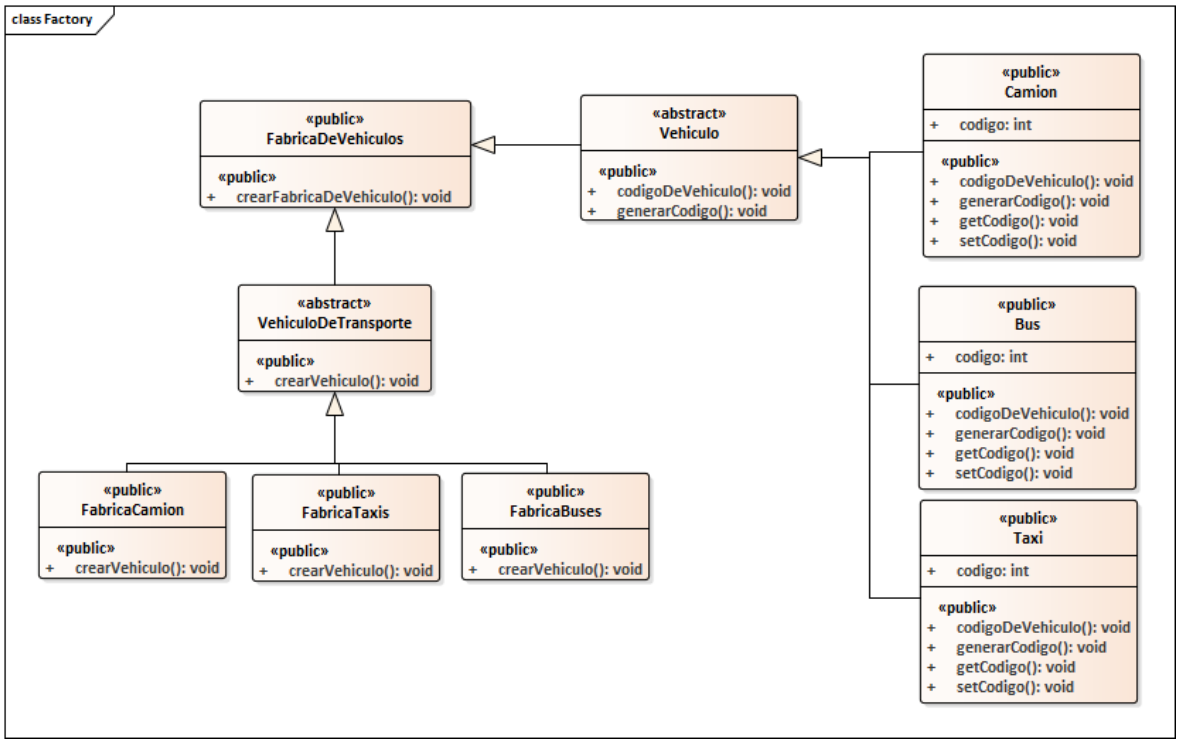


Figura 9.1. Estructura del patrón Abstract Factory adaptado para código java

Código adaptado a java del patrón Abstract Factory

Ahora se explica las clases que intervienen en la implementación del patrón Abstract Factory en código java.

Vehiculo.java:

Esta interface es común para todos los vehículo de nuestra FabricaDeVehiculos, en ella se declaran 2 métodos abstractos que serán comunes para los objetos a crear, sin importar si son Bus, Camion o Taxi, conocemos por regla que todos los métodos de una interfaz son abstractos, por ello no es necesario declararlos como tal.

```
package abstractfactory;
public interface Vehiculo {
    public void codigoDeVehiculo();
    public int generarCodigo();
}
```

VehiculoDeTransporte.java:

Esta interface será implementada por las diferentes fabricas de vehículos de la aplicación, cuenta con el método abstracto crearVehiculo(), que será común para cada fabrica y como su nombre lo dice, le permitirá a cada una implementar la lógica para crear sus objetos concretos.

```
package abstractfactory;
public interface VehiculoDeTransporte {
   public Vehiculo crearVehiculo();
}
```

FabricaDeVehiculos.java:

Esta clase será la fábrica principal, cuenta con un método estático que permitirá la creación de los diferentes tipos de vehículos, aplicamos el concepto de polimorfismo para ejecutar el llamado a la Fabrica correspondiente y crear el objeto concreto..

```
package abstractfactory;
public class FabricaDeVehiculos {
   public static void crearFabricaDeVehiculo(VehiculoDeTransporte factory) {
      Vehiculo objetoVehiculo = factory.crearVehiculo();
      objetoVehiculo.codigoDeVehiculo();
   }
}
```

FabricaBuses.java, FabricaCamion.java y FabricaTaxis.java:

Las clases se comportan de manera similar, implementan la interface VehiculoDeTransporte permitiendo crear los vehículos del tipo correspondiente y asignar el respectivo código de creación.

FabricaBuses.java:

```
package abstractfactory;
public class FabricaBuses implements VehiculoDeTransporte {
   public Vehiculo crearVehiculo() {
      Bus miBus = new Bus();
      miBus.setCodigo(miBus.generarCodigo());
      System.out.println("Se ha creado un nuevo Objeto Bus  ");
      return miBus;
   }
}
```

FabricaCamion.java:

```
package abstractfactory;
public class FabricaCamion implements VehiculoDeTransporte {
```

```
    @Override
    public Vehiculo crearVehiculo() {
        Camion miCamion = new Camion();
        miCamion.setCodigo(miCamion.generarCodigo());
        System.out.println("Se ha creado un nuevo Objeto Camion");
        return miCamion;
    }
}
```

FabricaTaxis.java:

```
package abstractfactory;
public class FabricaTaxis implements VehiculoDeTransporte {

    @Override
    public Vehiculo crearVehiculo() {
        Taxi miTaxi = new Taxi();
        miTaxi.setCodigo(miTaxi.generarCodigo());
        System.out.println("Se ha creado un nuevo Objeto Taxi");
        return miTaxi;
    }
}
```

Bus.java, Taxi.java y Camion.java:

Estas clases son las que instanciarán los objetos de tipo Vehículo (Buses, Camion y Taxis), implementan la interface Vehículo, y cada una permite generar un código aleatorio para identificar el vehículo creado.

Bus.java:

```
package abstractfactory;
public class Bus implements Vehiculo {
    private int codigo;
    public int generarCodigo() {
        int codigoBus = (int) (Math.random() * 9999);
        return codigoBus;
    }

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public void codigoDeVehiculo() {
        System.out.println("El Codigo del Bus es : " + getCodigo());
```

```
    }
}
```

Taxi.java:

```
package abstractfactory;
public class Taxi implements Vehiculo {
    private int codigo;
    public int generarCodigo() {
        int codigoTaxi = (int) (Math.random() * 9999);
        return codigoTaxi;
    }

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    @Override
    public void codigoDeVehiculo() {
        System.out.println("El Codigo del Taxi es : " + getCodigo());
    }
}
```

Camion.java:

```
package abstractfactory;
public class Camion implements Vehiculo {
    private int codigo;
    public int generarCodigo() {
        int codigoCamion = (int) (Math.random() * 9999);
        return codigoCamion;
    }

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    @Override
    public void codigoDeVehiculo() {
        System.out.println("El Codigo de la Camion es:" + getCodigo());
    }
}
```

Prueba del código java Abstract Factory

Esta clase permite iniciar el sistema, en ella creamos las instancias de Fabrica, mediante un menú de opciones se define y delega que Fabrica inicia el proceso de creación de los objetos que son seleccionados en el menú de opciones (JOptionPane).

```
package abstractfactory;
import javax.swing.JOptionPane;
public class NewMain {

    public static void main(String[] args) {
        FabricaCamion  Camion  = new FabricaCamion ();
        FabricaTaxis taxi = new FabricaTaxis();
        FabricaBuses buses = new FabricaBuses();
        String cad = "", salida;
        cad += "Ingrese la opción correspondiente para obtener el codigo del servicio\n";
        cad += "1. Codigo servicio de Taxis\n";
        cad += "2. Codigo servicio de Buses\n";
        cad += "3. Codigo servicio de Camion \n\n";
        try {
            do {
                try {
                    int opcion = Integer.parseInt(JOptionPane.showInputDialog(cad));
                    switch (opcion) {
                        case 1:
                            FabricaDeVehiculos.crearFabricaDeVehiculo(taxi);
                            break;
                        case 2:
                            FabricaDeVehiculos.crearFabricaDeVehiculo(buses);
                            break;
                        case 3:
                            FabricaDeVehiculos.crearFabricaDeVehiculo(Camion );
                            break;
                        default:
                            JOptionPane.showMessageDialog(null, "No es un valor de
consultavalido");
                            break;
                    }
                } catch (Exception e) {
                    JOptionPane.showMessageDialog(null, "No es un parametro de consulta
valido");
                }
                salida = JOptionPane.showInputDialog("Desea consultar otro codigo? S/N");
            } while (salida.toUpperCase().equals("S"));
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, "Bye!!!");
        }
    }
}
```

Salida del código java Abstract Factory

Es un patrón que utiliza muy bien los conceptos de herencia y polimorfismo, al utilizar las opciones seleccionadas y crear objetos como se está eligiendo (polimorfismo), también muestra como las clases concretan representan los objetos que crean las fábricas, mostramos el comportamiento en el terminal.

```
Se ha creado un nuevo Objeto Taxi
El Codigo del Taxi es : 4403
Se ha creado un nuevo Objeto Bus
El Codigo del Bus es : 9350
Se ha creado un nuevo Objeto Camion
El Codigo de la Camion es:3861
```

B. Estructura del patrón Singleton adaptado

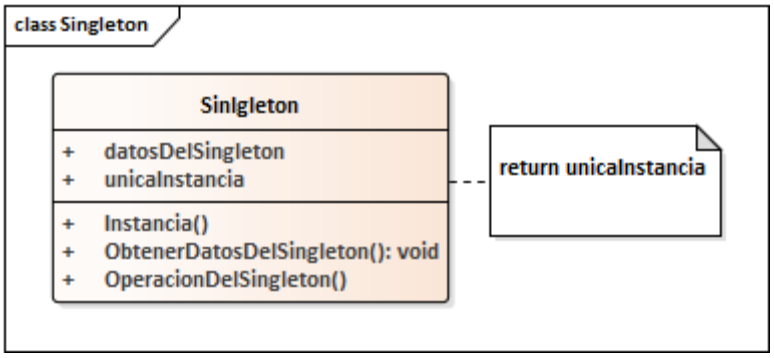


Figura 9.2. Estructura del patrón Singleton adaptado para código java.

Código adaptado a java del patrón Singleton

Se crea una clase Singleton, el cual tiene un constructor privado que es inaccesible desde otra clase, el método Instancia() retorna una instancia de la clase Singleton, de no existir instancia crea una, caso contrario, retorna la instancia creada.

```
package Singleton;
public class Singleton {
   private static Singleton unicaIntancia;
   private String datosDelSingleton;

   public static Singleton Instancia (){
      if (unicaIntancia == null){
         unicaIntancia = new Singleton();
      }else{
         System.out.println("No se puede instanciar un nuevo objeto"
               + " Singleton, se retorna : "+ unicaIntancia.hashCode());
      }
```

```
        return unicaIntancia;
    }
    public String getDatosDelSingleton() {
        return datosDelSingleton;
    }
    public void setDatosDelSingleton(String datosDelSingleton) {
        this.datosDelSingleton = datosDelSingleton;
    }
}
```

Prueba del código java Singleton

Para probar el patrón, se instancia tres objetos del tipo Singleton, se agregan datos a los objetos por separado y finalmente se procese a extraer e imprimir en consola los datos de las tres instancias.

```
package patronesdiseño;
import Singleton.Singleton;
public class PatronesDiseño {
    public static void main(String[] args) {
        /* Se instancia tres objetos del tipo Singleton*/

        Singleton s1 = Singleton.Instancia();
        Singleton s2 = Singleton.Instancia();
        Singleton s3 = Singleton.Instancia();

         /* se agregan datos a los objetos por separado  */
        s1.setDatosDelSingleton("Datos del primer objeto");
        s2.setDatosDelSingleton("Datos del segundo objeto");
        s3.setDatosDelSingleton("Datos del tercer objeto");

        /* se extraen los datos de las tres instancias */
        System.out.println(s1.getDatosDelSingleton());
        System.out.println(s2.getDatosDelSingleton());
        System.out.println(s3.getDatosDelSingleton());
    }
}
```

Salida del código java Singleton

En la consola se muestra que efectivamente la instancia Singleton, se instancia una vez y los datos que muestra son del último objeto, así mismo, el código hash 366712642, muestra que se trata del mismo objeto.

```
No se puede instanciar un nuevo objeto Singleton, se retorna: 366712642
No se puede instanciar un nuevo objeto Singleton, se retorna: 366712642
```

Datos del tercer objeto
Datos del tercer objeto
Datos del tercer objeto

C. Estructura del patrón Adapter adaptado

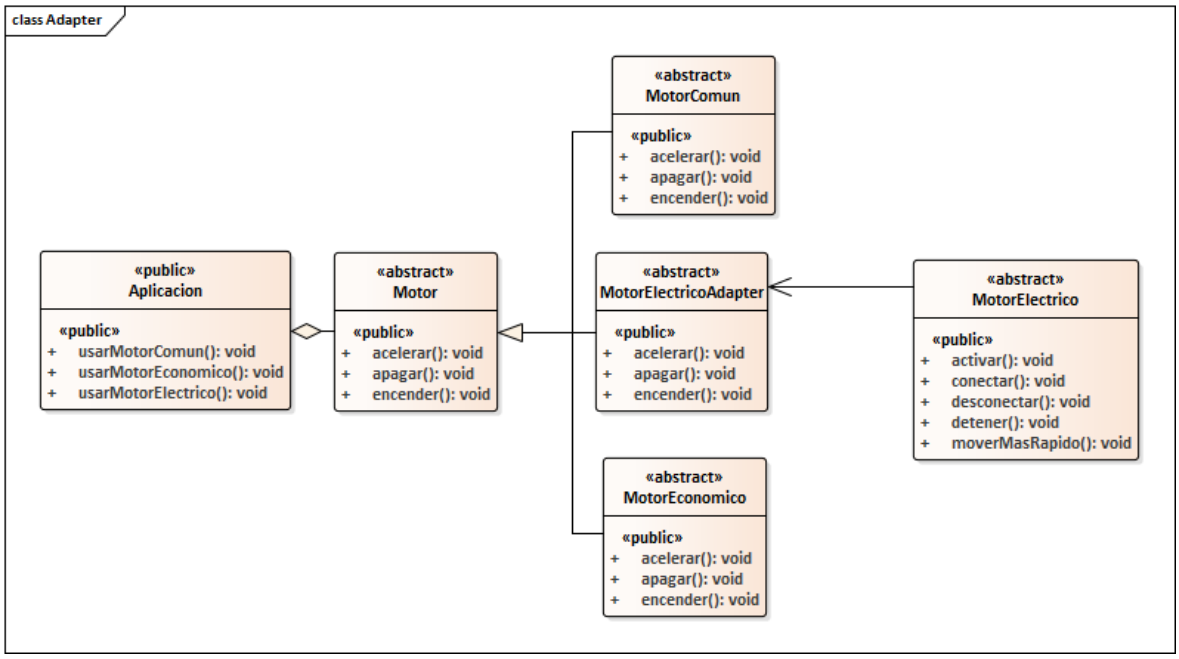


Figura 9.3. Estructura del patrón Adapter adaptado para código java.

Código adaptado a java del patrón Adapter

Explicamos cómo las clases intervienen en la implementación del patrón Adapter en código java.

Motor.java:

Esta interfaz es la que define los métodos que se heredan de los diferentes tipos de motores, provee los métodos comunes: encender, acelerar y apagar, para su funcionamiento.

```
package adapter;
public abstract class Motor {
   abstract public void encender();
   abstract public void acelerar();
   abstract public void apagar();
}
```

MotorComun.java y MotorEconomico.java

Estas clases representan la estructura de los motores normales con los que el sistema

funciona, básicamente heredan de la clase Motor y realizan el funcionamiento básico que se provee.

MotorComun.java

```
package adapter;
public class MotorComun extends Motor {
    public MotorComun() {
        super();
        System.out.println("Creando el motor comun");
    }

    @Override
    public void encender() {
        System.out.println("encendiendo motor comun");
    }
    @Override
    public void acelerar() {
        System.out.println("acelerando el motor comun");
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor comun");
    }
}
```

MotorEconomico.java

```
package adapter;
public class MotorEconomico extends Motor {
    public MotorEconomico(){
        super();
        System.out.println("Creando motor economico");
    }

    @Override
    public void encender() {
        System.out.println("Encendiendo motor economico.");
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando motor economico.");
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor economico.");
```

```
    }
}
```

MotorElectricoAdapter.java:

En la implementación de la clase Motor, se establece el puente mediante el cual la clase incompatible puede ser utilizada, la clase incompatible hereda de la clase abstracta Motor y mediante la implementación, realiza la comunicación con la clase a adaptar usando una instancia de la misma, cabe resaltar que el patrón se caracteriza por permitir esta compatibilidad entre clases que no comparten el mismo comportamiento.

```
package adapter;
public class MotorElectricoAdapter extends Motor {
    private MotorElectrico motorElectrico;
    public MotorElectricoAdapter() {
        super();
        this.motorElectrico = new MotorElectrico();
        System.out.println("Creando motor Electrico adapter");
    }

    @Override
    public void encender() {
        System.out.println("Encendiendo motorElectricoAdapter");
        this.motorElectrico.conectar();
        this.motorElectrico.activar();
    }

    @Override
    public void acelerar() {
        System.out.println("Acelerando motor electrico...");
        this.motorElectrico.moverMasRapido();
    }

    @Override
    public void apagar() {
        System.out.println("Apagando motor electrico");
        this.motorElectrico.detener();
        this.motorElectrico.desconectar();
    }
}
```

MotorElectrico.java:

Esta es la clase que se adapta, a pesar de ser un motor que posee características muy diferentes a los demás tipos de motores del sistema, clase que tiene comportamiento diferente a las demás clases, no puede heredar directamente de la clase abstracta Motor,

en lugar de esto, es accedida por la clase Adapter el cual le permite tener compatibilidad de los demás motores.

```
package adapter;
public class MotorElectrico {
    private boolean conectado = false;
    public MotorElectrico() {
        System.out.println("Creando motor electrico");
        this.conectado = false;
    }

    public void conectar() {
        System.out.println("Conectando motor electrico");
        this.conectado = true;
    }

    public void activar() {
        if (!this.conectado) {
            System.out.println("No se puede activar porque no "
                    + "esta conectado el motor electrico");
        } else {
            System.out.println("Esta conectado, activando motor"
                    + " electrico....");
        }
    }

    public void moverMasRapido() {
        if (!this.conectado) {
            System.out.println("No se puede mover rapido el motor "
                    + "electrico porque no esta conectado...");
        } else {
            System.out.println("Moviendo mas rapido...aumentando voltaje");
        }
    }

    public void detener() {
        if (!this.conectado) {
            System.out.println("No se puede detener motor electrico"
                    + " porque no esta conectado");
        } else {
            System.out.println("Deteniendo motor electrico");
        }
    }

    public void desconectar() {
        System.out.println("Desconectando motor electrico...");
        this.conectado = false;
    }
}
```

Prueba del código java Adapter

En la clase de prueba del sistema que usa los diferentes tipos de motores, instanciamos los diferentes tipos de motores, y se intenta acceder a los métodos comunes que se definen en la clase abstracta Motor.

```
package adapter;
import adapter.MotorElectrico;
public class Principal {
    public static void main(String[] args) {
        Principal miPrincipal = new Principal();
        System.out.println("\n***Motor Comun***");
        miPrincipal.usarMotorComun();
        System.out.println("******\n");
        System.out.println("***Motor Economico***");
        miPrincipal.usarMotorEconomico();
        System.out.println("******\n");
        System.out.println("***Motor Electrico***");
        miPrincipal.usarMotorElectrico();
        System.out.println("******\n");
    }

    private void usarMotorComun() {
        Motor motor = new MotorEconomico();
        motor = new MotorComun();
        motor.encender();
        motor.acelerar();
        motor.apagar();
    }

    private void usarMotorElectrico() {
        Motor motor = new MotorElectricoAdapter();
        motor.encender();
        motor.acelerar();
        motor.apagar();
    }

    private void usarMotorEconomico() {
        Motor motor = new MotorEconomico();
        motor.encender();
        motor.acelerar();
        motor.apagar();
    }
}
```

Salida del código java Adapter

En la consola se muestra como todas las instancias acceden a los mismos métodos:

encender(), acelerar() o apagar(), transparente al tipo de motor, el motor eléctrico tiene un comportamiento diferente pero gracias al adaptador, logra comportarse de forma similar. En conclusión, se puede utilizar una nueva clase sin afectar la lógica del sistema, utilizamos una clase que sirvió como puente o adaptador para la clase nueva, sin que eso afectara el código de nuestras clases existentes.

```
***Motor Comun***
Creando motor economico
Creando el motor comun
encendiendo motor comun
acelerando el motor comun
Apagando motor comun
******

***Motor Economico***
Creando motor económico.
Encendiendo motor economico.
Acelerando motor economico.
Apagando motor economico.
******

***Motor Electrico***
Creando motor eléctrico.
Creando motor Electrico adapter.
Encendiendo motorElectricoAdapter.
Conectando motor eléctrico.
Esta conectado, activando motor electrico.
Acelerando motor electrico.
Moviendo mas rapido...aumentando voltaje.
Apagando motor eléctrico.
Deteniendo motor eléctrico.
Desconectando motor eléctrico.
******
```

D. Estructura del patrón Decorator adaptado

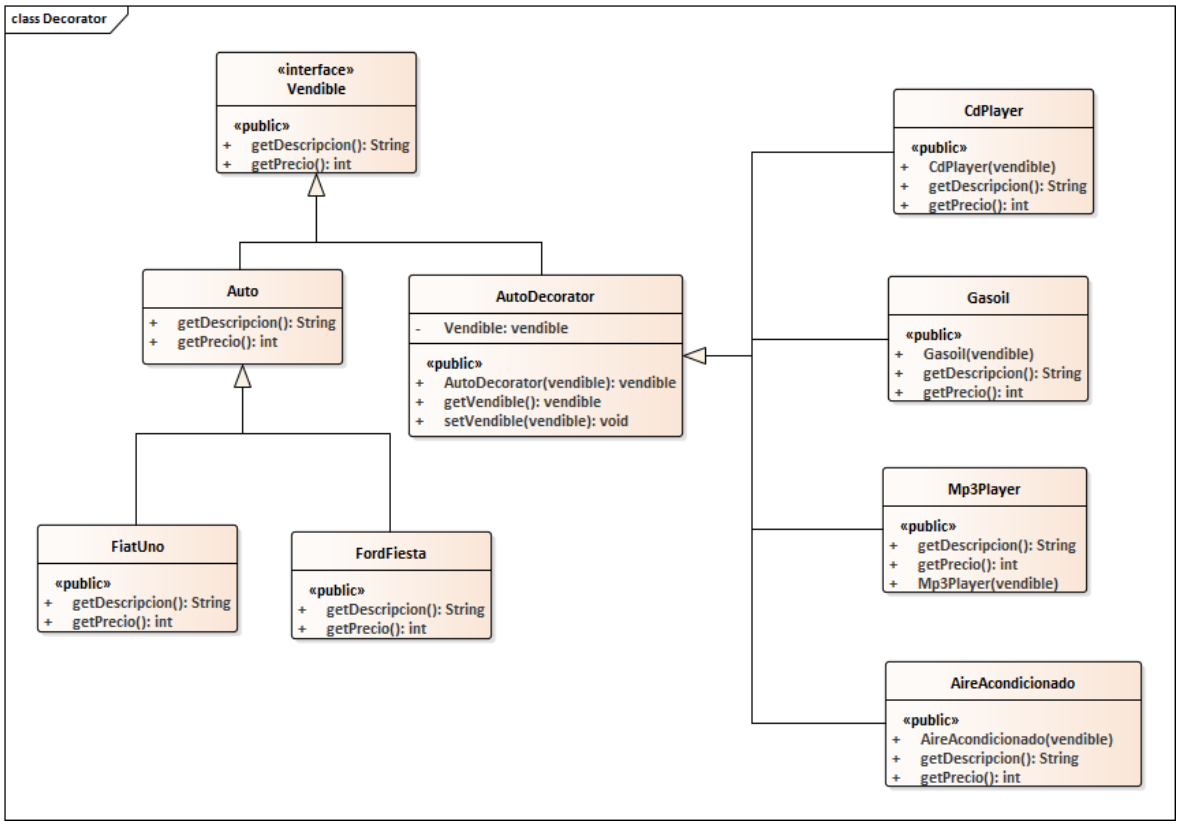


Figura 9.4. Estructura del patrón Decorator adaptado para código java.

Código adaptado a java del patrón Decorator

Explicamos cómo las clases intervienen en la implementación del patrón Decorator en código java.

Vendible.java:

Se crea una interface Vendible que declara los métodos: getDescripcion() y getPrecio(), los que exigirá su implementación en las clases que sean heredadas.

```
package decorator;
public interface Vendible {
   public String getDescripcion();
   public int getPrecio();
}
```

Auto.java:

Es la implementación de la interface Vendible, en esta clase se crea la lógica sobre los métodos heredados de la interface, adicionalmente se puede agregar atributos y métodos propios de la clase.

```
package decorator;
public class Auto implements Vendible {

      @Override
   public String getDescripcion() {
      throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.
   }

   @Override
   public int getPrecio() {
      throw new UnsupportedOperationException("Not supported yet."); //To change body of generated methods, choose Tools | Templates.
   }
}
```

FiatUno.java y FordFiesta.java:

Por la herencia, estas clases heredan los métodos de la clase Auto, aquí se implementan la descripción: tipo de auto y precio que retorna su precio. Ambas clases tiene comportamiento parecido.

FordFiesta.java

```
package decorator;
public class FordFiesta extends Auto {
   public String getDescripcion() {
      return "Ford Fiesta modelo 2018";
   }
   public int getPrecio() {
      return 35000;
   }
}
```

FiatUno.java

```
package decorator;
public class FiatUno extends Auto {

   public String getDescripcion() {
      return "Fiar uno modelo 2018";
   }
   public int getPrecio() {
      return 15000;
   }
}
```

AutoDecorator.java:

En esta clase se crea el decorador la esencia del patrón, luego se implementan múltiples

decoradores, en la clase se implementa la interface vendible, el cual regresa una instancia y mediante esta se agrega la funcionalidad a los objetos que dependan de esta clase.

```
package decorator;
public class AutoDecorator implements Vendible {
   private Vendible vendible;

      public AutoDecorator(Vendible vendible) {
       setVendible(vendible);
   }

   public Vendible getVendible() {
      return vendible;
   }

   public void setVendible(Vendible vendible) {
      this.vendible = vendible;
   }

   @Override
   public String getDescripcion() {
      throw new UnsupportedOperationException("Not supported yet.");
   }

   @Override
   public int getPrecio() {
      throw new UnsupportedOperationException("Not supported yet.");
   }
}
```

AireAcondicionado.java, CdPlayer.java, Gasoil.java, y Mp3Player.java:

Las clases que son accesorios se comportan de la misma forma, implementan los métodos de la interface vendible mediante el decorador (sirve como puente), así cuando se adjunte al objeto Auto, se podrá agregar la funcionalidad extra por medio del decorador.

AireAcondicionado.java

```
package decorator;
public class AireAcondicionado extends AutoDecorator {

   public AireAcondicionado(Vendible vendible) {
      super(vendible);
```

```
    }

    public String getDescripcion() {
        return getVendible().getDescripcion() + " + Aire Acondicionado (1500)";
    }

    public int getPrecio() {
        return getVendible().getPrecio() + 1500;
    }
}
```

CdPlayer.java

```
package decorator;
public class CdPlayer extends AutoDecorator {

    public CdPlayer(Vendible vendible) {
        super(vendible);
    }

    public String getDescripcion() {
        return getVendible().getDescripcion() + " + Cd Player ( 100) ";
    }

    public int getPrecio() {
        return getVendible().getPrecio() + 100;
    }
}
```

Gasoil.java

```
package decorator;
public class Gasoil extends AutoDecorator {

    public Gasoil(Vendible vendible) {
        super(vendible);
    }

    public String getDescripcion() {
        return getVendible().getDescripcion() + " + Gasoil ( 1200) ";
    }

    public int getPrecio() {
        return getVendible().getPrecio() + 1200;
    }
}
```

Mp3Player.java

```
package decorator;
```

```
public class Mp3Player extends AutoDecorator {
    public Mp3Player(Vendible vendible) {
        super(vendible);
    }

    public String getDescripcion() {
        return getVendible().getDescripcion() + " + Mp3Playero (300)";
    }

    public int getPrecio() {
        return getVendible().getPrecio() + 300;
    }
}
```

Prueba del código java Decorator

Para probar el patrón, se crea dos instancias del tipo Auto (auto y auto2), luego se agregan diferentes decoradores a cada objeto (para notar el comportamiento diferente), luego se imprime la descripción y su precio final.

```
package decorator;
public class NewMain {
    public static void main(String[] args) {
        Vendible auto = new FiatUno();
        auto = new CdPlayer(auto);
        auto = new Gasoil(auto);

        System.out.println(auto.getDescripcion());
        System.out.println("Su precio es : " + auto.getPrecio());

        Vendible auto2 = new FordFiesta();
        auto2 = new Mp3Player(auto2);
        auto2 = new AireAcondicionado(auto2);
        auto2 = new Gasoil(auto2);

        System.out.println(auto2.getDescripcion());
        System.out.println("Su precio es : " + auto2.getPrecio());
    }
}
```

Salida del código java Decorator

En la consola se muestra como los dos objetos se comportaron y fueron modificados por los decoradores, en la descripción se nota, que decoradores fueron agregados y el precio final es la suma de todos los precios de los decoradores.

Fiat es uno modelo 2018 + Cd Player (100) + Gasoil (1200)
Su precio es: 16300
Ford Fiesta modelo 2018 + Mp3Playero (300) + Aire Acondicionado (1500) + Gasoil (1200)
Su precio es: 38000

E. Estructura del patrón Facade adaptado

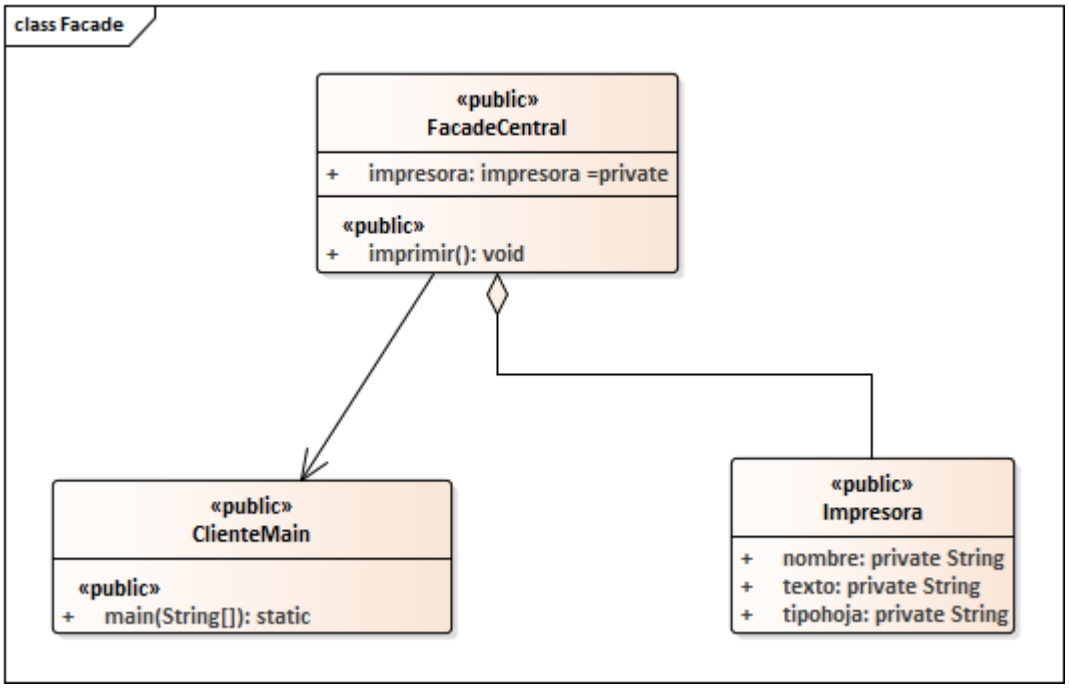


Figura 9.5. Diagrama del patrón Facade adaptado para código java.

Código adaptado a java del patrón Facade

Una aplicación para explicar las clases que intervienen en la implementación del patrón Facade en código java.

Impresora.java:

Se crea una clase Impresora el cual se conectara con la clase FacadeCentral, para poder permitirle acceder a un método que solo FacadeCentral le permita, se crean los atributos necesarios para el ejemplo y se encapsulan.

```
public class Impresora {
   private String nombre;
   private String tipoHoja;
   private String texto;
   public String getNombre() {
      return nombre;
   }
```

```
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getTipoHoja() {
        return tipoHoja;
    }

    public void setTipoHoja(String tipoHoja) {
        this.tipoHoja = tipoHoja;
    }

    public String getTexto() {
        return texto;
    }

    public void setTexto(String texto) {
        this.texto = texto;
    }
}
```

FacadeCentral.java:

La clase FacadeCentral crea una instancia de impresora, implementa el método imprimir, el cual modifica los atributos de impresora, así como, el de utilizar la instancia creada, de esta manera se restringe el acceso de los métodos a impresora desde otra clase que no sea FacadeCentral.

```
public class FacadeCentral {
    private Impresora impresora;
    public void imprimir(String texto) {
        impresora = new Impresora();
        impresora.setNombre("NICO-Printer");
        impresora.setTipoHoja("A4");
        impresora.setTexto(texto);
    }
}
```

Prueba del código java de Facade

Para probar el patrón, se crea una instancia de FacadeCentral, y mediante esta se llama al método imprimir, se le agrega el único parámetro que acepta el método e internamente se instancia a la impresora, luego se modifica sus atributos. De esta manera el usuario no interactúa con impresora, pero si logra modificarlo.

```
public class ClienteMain {
   public static void main(String[] args) {
      FacadeCentral facadeCentral = new FacadeCentral();
      facadeCentral.imprimir("Texto a imprimir");
   }
}
```

Salida del código java Facade

Al realizar el análisis del objeto en tiempo de ejecución, en memoria se tiene una instancia del tipo Impresora, el que tiene los datos que se muestran.

```
impresora.getNombre() => "NICO-Printer"
impresora.getTipoHoja() => "A4"
impresora.getTexto() => “Texto a imprimir”
```

F. Estructura del patrón Iterator adaptado

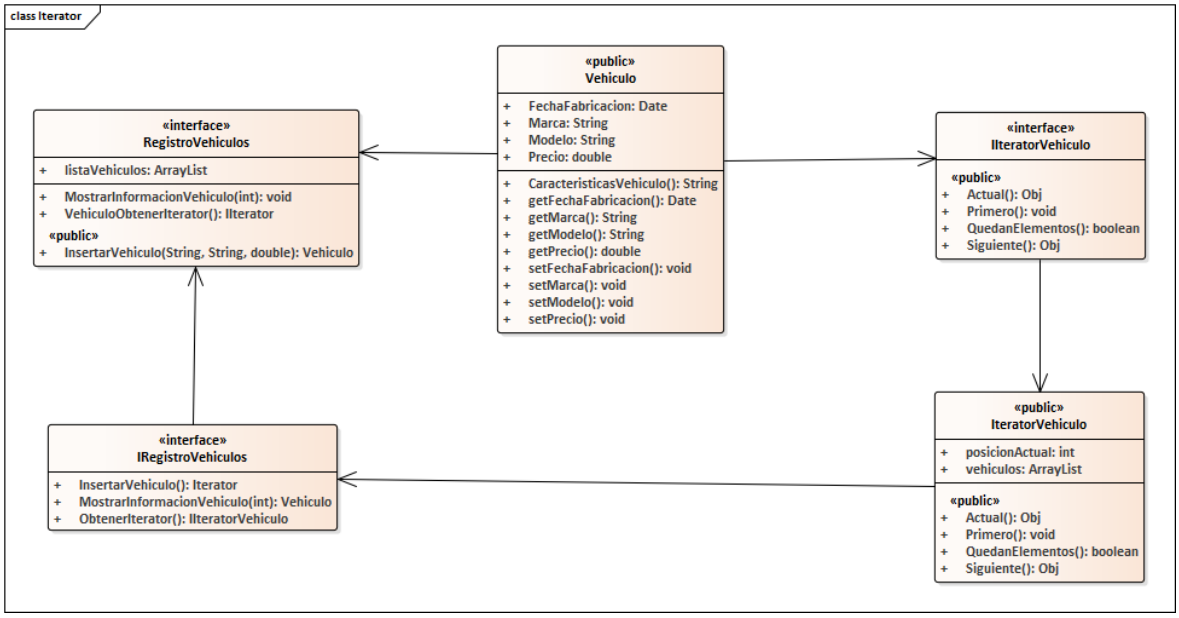


Figura 9.6. Estructura del patrón Iterator adaptado para código java.

Código adaptado a java del patrón Iterator

Explicamos las clases que intervienen en la implementación del patrón Iterator en código java.

Vehiculo.java:

Se crea la clase Vehiculo, se encapsula los atributos de Marca, Modelo,

FechaFabricacion y el Precio, se declara el método CaracteristicasVehiculo(), el cual retorna una cadena con las características del vehículo, con fines de impresión.

```
package Iterator;
import java.util.Date;

public class Vehiculo {
   private String Marca;
   private String Modelo;
   private Date FechaFabricacion;
   private double Precio;

   public Vehiculo(String marca, String modelo,
         Date fechaFabricacion, double precio) {
      this.Marca = marca;
      this.Modelo = modelo;
      this.FechaFabricacion = fechaFabricacion;
      this.Precio = precio;
   }

   public String CaracteristicasVehiculo() {
      return getMarca() + " " + getModelo() + " fabricado en "
            + getFechaFabricacion().toString() + " con un precio de "
            + getPrecio() + " Soles.\n";
   }

   public String getMarca() {
      return Marca;
   }

   public void setMarca(String Marca) {
      this.Marca = Marca;
   }

   public String getModelo() {
      return Modelo;
   }

   public void setModelo(String Modelo) {
      this.Modelo = Modelo;
   }

   public Date getFechaFabricacion() {
      return FechaFabricacion;
   }

   public void setFechaFabricacion(Date FechaFabricacion) {
      this.FechaFabricacion = FechaFabricacion;
   }
```

```
    public double getPrecio() {
        return Precio;
    }

    public void setPrecio(double Precio) {
        this.Precio = Precio;
    }
}
```

IRegistroVehiculos.java:

Se declara la interfaz que posteriormente será implementada en la clase RegistroVehiculos, se utiliza la interfaz para abstraer los métodos que serán implementados en las clases hijas, como herencia.

```
package Iterator;
public interface IRegistroVehiculos {
    void InsertarVehiculo(String marca, String modelo, double precio);
    Vehiculo MostrarInformacionVehiculo(int indice);

    IIteratorVehiculo ObtenerIterator();
}
```

RegistroVehiculos.java:

Por la herencia, la clase implementa los métodos abstraídos en la interfaz anterior, para el caso se trabaja con una lista de objetos del tipo Vehiculo, donde es necesario persistir el tipo de objetos en la lista.

```
package Iterator;
import java.util.ArrayList;
import java.util.Date;

public class RegistroVehiculos implements IRegistroVehiculos {
    private ArrayList listaVehiculos;
    public RegistroVehiculos() {
        this.listaVehiculos = new ArrayList();
    }

    public void InsertarVehiculo(String marca, String modelo, double precio) {
        Vehiculo v = new Vehiculo(marca, modelo, new Date(), precio);
        listaVehiculos.add(v);
    }

    public Vehiculo MostrarInformacionVehiculo(int indice) {
```

```
        return (Vehiculo) listaVehiculos.get(indice);
    }

    public IIterator VehiculoObtenerIterator() {
        return new IteratorVehiculo(listaVehiculos);
    }
}
```

IIteratorVehiculo.java:

En la interfaz IIteratorVehiculo se implementa el patrón Iterator, es quien en realidad hace todo el trabajo, podrá responder el primer vehículo, el vehículo actual, pasar al siguiente y decir cuántos vehículos quedan. Para la implementación, utilizamos una interfaz para poder abstraer las operaciones, mas no implementarlas.

```
package Iterator;
public interface IIteratorVehiculo {
    void Primero();
    Vehiculo Actual();
    Vehiculo Siguiente();
    boolean QuedanElementos();
}
```

IteratorVehiculo.java:

La clase implementa los métodos de la interfaz, adicionalmente debemos tener una referencia al listado completos de vehículos, el método que implementa utiliza el constructor de la clase en cuestión y en runtime inyecta a la lista, además, utilizamos una variable entera que mantiene en memoria el índice del elemento que se encuentra en el Iterator.

```
package Iterator;
import java.util.ArrayList;
public class IteratorVehiculo implements IIteratorVehiculo {
    private ArrayList vehiculos;
    private int posicionActual = -1;
    public IteratorVehiculo(ArrayList listado) {
        this.vehiculos = listado;
    }

    @Override
    public void Primero() {
        this.posicionActual = -1;
    }
```

```
    @Override
    public Vehiculo Actual() {
        if ((this.vehiculos == null)
                || (this.vehiculos.size() == 0)
                || (posicionActual > this.vehiculos.size() - 1)
                || (this.posicionActual < 0)) {
            return null;
        } else {
            return (Vehiculo) this.vehiculos.get(posicionActual);
        }
    }

    @Override
    public Vehiculo Siguiente() {
        if ((this.vehiculos == null)
                || (this.vehiculos.size() == 0)
                || (posicionActual + 1 > this.vehiculos.size() - 1)) {
            return null;
        } else {
            return (Vehiculo) this.vehiculos.get(posicionActual + 1);
        }
    }

    @Override
    public boolean QuedanElementos() {
        return (posicionActual + 1 <= this.vehiculos.size() - 1);
    }
}
```

Prueba del código java Iterator

Para probar el patrón, se crea cinco instancias del tipo Vehículo, estos vehículos se registran en la lista de vehículos mediante el Iterator, para probar la rotación de la posición al momento de consultar se hace un recorrido con while.

```
package Iterator;
public class NewMain {
    public static void main(String[] args) {
        IRegistroVehiculos registro = new RegistroVehiculos();
        registro.InsertarVehiculo("Volkswagen", "Polo", 12300);
        registro.InsertarVehiculo("Volkswagen", "Golf GTI", 18900);
        registro.InsertarVehiculo("Volkswagen", "Passat", 27000);
        registro.InsertarVehiculo("Volkswagen", "Scirocco", 32100);
        registro.InsertarVehiculo("Volkswagen", "Touareg", 21800);

        IIteratorVehiculo iterador = registro.ObtenerIterator();

        while (iterador.QuedanElementos()) {
```

```
            Vehiculo v = iterador.Siguiente();

            System.out.println(v.getMarca() + " " + v.getModelo() + " fabricado el " +
v.getFechaFabricacion().toString() + " (" + v.getPrecio() + " Soles)");
        }
    }
}
```

Salida del código java Iterator

En la consola se muestra, como los objetos se comportaron y fueron listados según se ordeno.

```
Volkswagen Polo fabricado el Tue Aug 21 02:16:10 PET 2018 (12300.0 Soles)
Volkswagen Golf GTI fabricado el Tue Aug 21 02:16:10 PET 2018 (18900.0 Soles)
Volkswagen Passat fabricado el Tue Aug 21 02:16:10 PET 2018 (27000.0 Soles)
Volkswagen Scirocco fabricado el Tue Aug 21 02:16:10 PET 2018 (32100.0 Soles)
Volkswagen Touareg fabricado el Tue Aug 21 02:16:10 PET 2018 (21800.0 Soles)
```

G. Estructura del patrón Observer adaptado

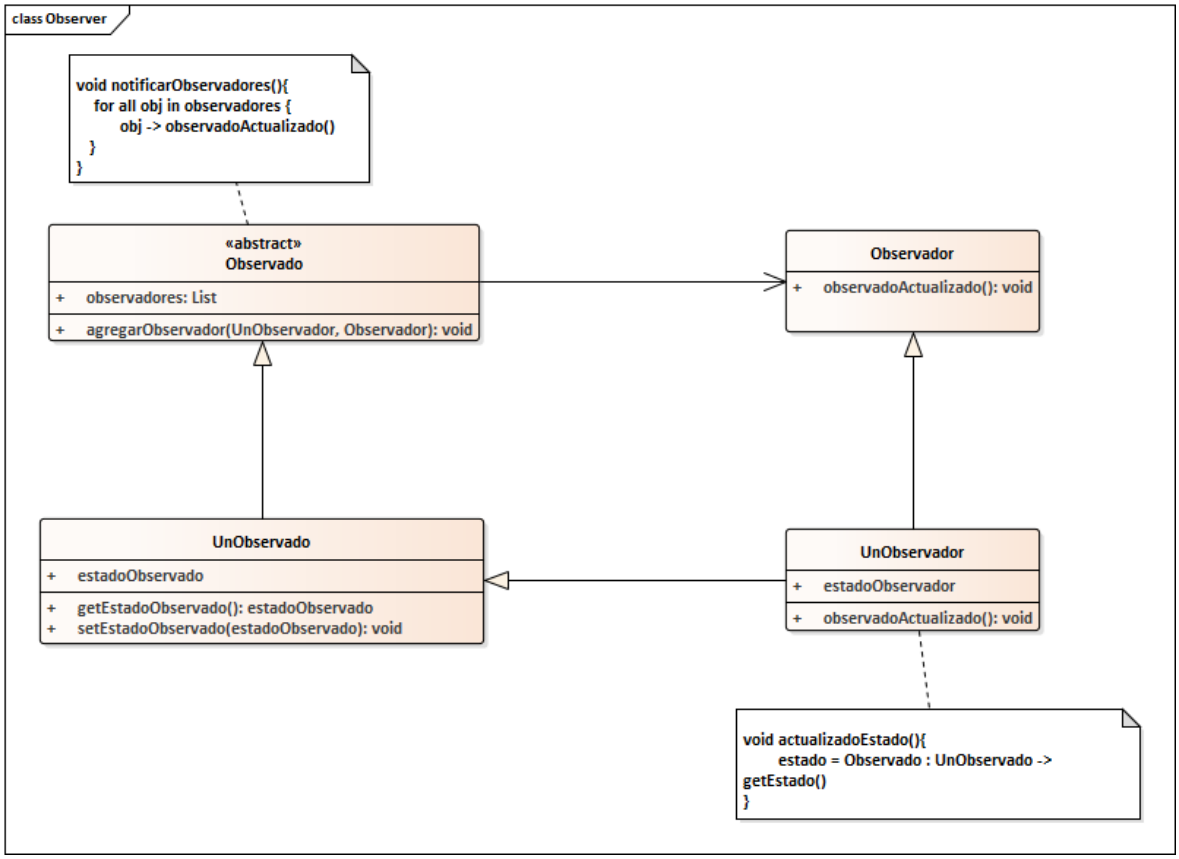


Figura 9.7. Estructura del patrón Observer adaptado para código java

Código adaptado a java del patrón Observer

Ahora desarrollamos las clases que intervienen en la implementación del patrón

Observer en código java.

IObservador.java:

Se crea una interface IObservador que declara un método observadoActualizado(), el cual es un void, en su implementación recién se agrega la lógica del observador.

```
package observer;
public interface IObservador {
   public void observadoActualizado();
}
```

Observado.java:

Según el diagrama, cada vez que se agregue un nuevo observador, los existentes serán notificados, por eso el método agregarObservador(), invoca el método notificarObservadores(); el método notificarObservadores() envía la notificación a cada observador a través de su propio método.

```
package observer;
import java.util.ArrayList;
Public abstract class Observado {
   private ArrayList<IObservador> observadores = new ArrayList<IObservador>();

   public Observado() {
   }
   public void agregarObservador(IObservador o) {
      observadores.add(o);
      notificarObservadores();
   }
   public void eliminarObservador(IObservador o) {
      observadores.remove(o);
   }
   public void notificarObservadores() {
      for (IObservador obj : observadores) {
         obj.observadoActualizado();
      }
   }
}
```

UnObservado.java:

La clase UnObservado extiende las propiedades de la clase Observado, en adición implementa un constructor vacío.

```
public class UnObservado extends Observado {
   public UnObservado() {
```

```
    }
}
```

UnObservador.java:

La clase implementa a la clase abstracta IObservador(), permitiendo implementar el método observadoActualizado(), el cual imprime en consola el comportamiento de la clase, se muestra cómo se va actualizando el estado del observador.

```
package observer;
public class UnObservador implements IObservador {
    private String nombre;
    public UnObservador(String nombre) {
        this.setNombre(nombre);
        System.out.println("Observador [" + this.nombre + "] creado");
    }
    public String getNombre() {
        return this.nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    @Override
    public void observadoActualizado() {
        System.out.println("Observador [" + this.getNombre() + "] recibe la notificación");
    }
}
```

Prueba del código java Observer

Para probar el patrón, se hace una instancia de la clase UnObservado, posteriormente se instancian tres objetos del tipo UnObservador con diferentes nombres (jose, juan y efrain). Se agregan los tres objetos creados a la lista de observadores y se pasa a comprobar las observaciones que se generan en la consola.

```
package observer;
public class Main {
    public static void main(String[] args) {
        UnObservado objObservado = new UnObservado();
        UnObservador objObservadorPepe = new UnObservador("jose");
        objObservado.agregarObservador(objObservadorPepe);
        UnObservador objObservadorJuan = new UnObservador("Juan");
        objObservado.agregarObservador(objObservadorJuan);
        UnObservador objObservadorMarta = new UnObservador("efrain");
        objObservado.agregarObservador(objObservadorMarta);
    }
```

```
}
```

Salida del código java Observer

En la consola se muestra como se crea un objeto que observara el comportamiento de los objetos que sean agregados, cuando se crea el objeto jose, el observador registra que: (1) fue creado (2) envía la notificación al único que está en la lista (jose). El procedimiento se repite, cada vez que se agrega al observador, la diferencia recae en la cantidad de notificaciones que se envía, cada vez que se agrega uno, se envía una notificación más.

```
Observador [jose] creado
Observador [jose] recibe la notificación
Observador [Juan] creado
Observador [jose] recibe la notificación
Observador [Juan] recibe la notificación
Observador [efrain] creado
Observador [jose] recibe la notificación
Observador [Juan] recibe la notificación
Observador [efrain] recibe la notificación
```

H. Estructura del patrón State adaptado

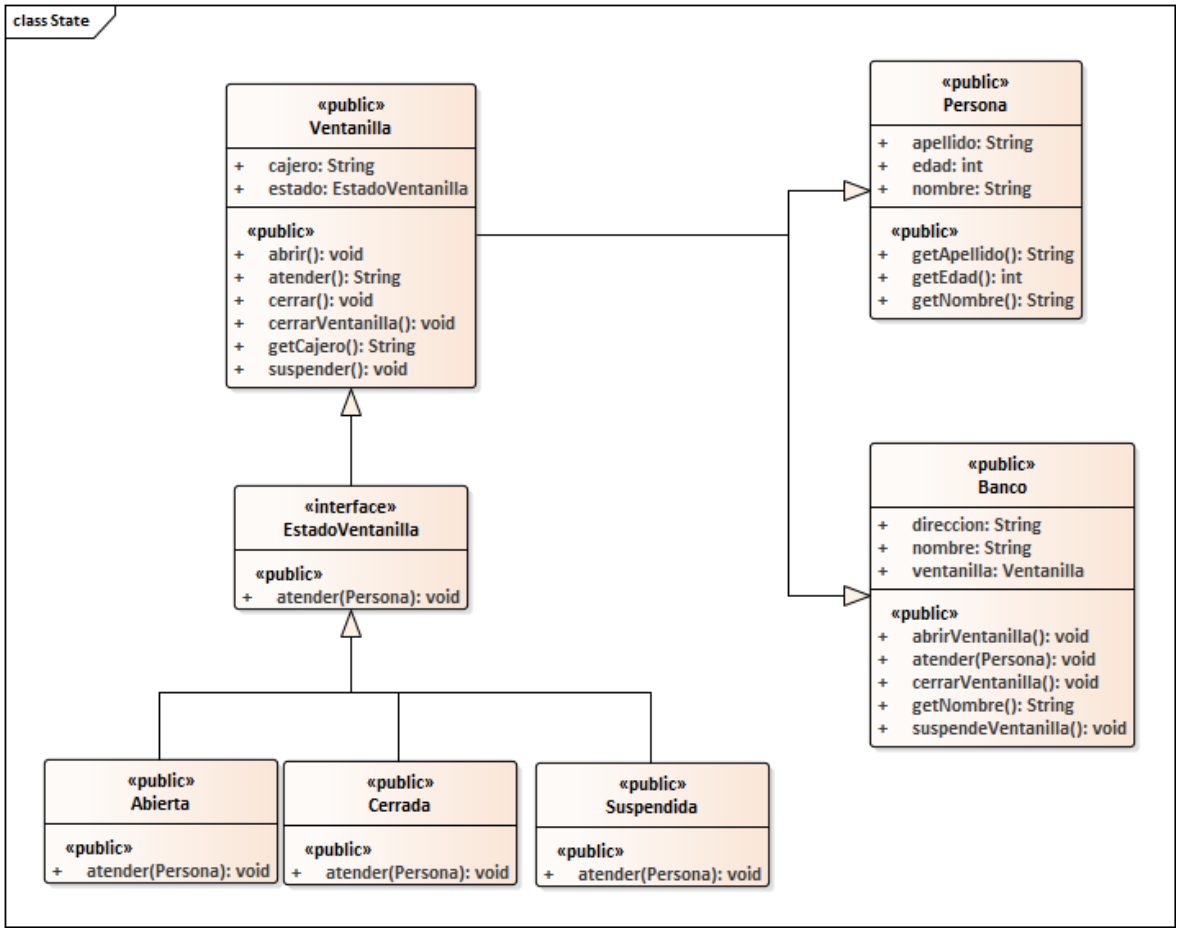


Figura 9.8. Estructura del patrón State adaptado para código java.

Código adaptado a java del patrón State

Persona.java:

Se crea una clase Persona que tendrá los atributos de: nombre, apellido y edad, necesarios para probar el patrón, la clase creada solo se referencia como un caso.

```
package State;
public class Persona {
    private String nombre;
    private String apellido;
    private int edad;

    public Persona(String nombre, String apellido, int edad) {
        setApellido(apellido);
        setNombre(nombre);
        setEdad(edad);
```

```
    }

    public String getApellido() {
        return apellido;
    }

    public int getEdad() {
        return edad;
    }

    public String getNombre() {
        return nombre;
    }

    public void setApellido(String apellido) {
        this.apellido = apellido;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

}
```

Banco.java:

Se crea una clase Banco que tendrá los atributos: nombre, dirección y ventanilla, este último del tipo Ventanilla, mediante la instancia ventanilla, se accederá a las implementaciones de EstadoVentanilla, el banco ofrece el método atender(), pero en realidad la instancia ventanilla responde ese método.

```
package State;
public class Banco {
    private String nombre;
    private String direccion;
    private Ventanilla ventanilla;

    public Banco() {
        ventanilla = new Ventanilla();
    }
    public  void  atender( Persona persona){
        System.out.println(persona.getNombre() + " Ingresa a la fila");
        ventanilla. atender(persona);
    }
```

```
    public void suspendeVentanilla(){
        ventanilla.suspender();
    }
    public void cerrarVentanilla(){
        ventanilla.cerrar();
    }
     public void abrirVentanilla(){
        ventanilla.abrir();
    }
      public String getNombre(){
        return nombre;
    }
}
```

EstadoVentanilla.java:

La interfaz EstadoVentanilla, permite implementar los diferentes estados que Ventanilla demanda, mediante esta técnica, el estado se mantiene constante y los objetos tienen un comportamiento diferente.

```
package State;
public interface EstadoVentanilla {
    public void atender(Persona persona);
}
```

Abierta.java, Cerrada.java y Suspendida.java

Las clases indicadas se comportan de la misma forma, implementan los métodos de la clase Ventanilla mediante la interfaz EstadoVentanilla que sirve como enlace.

Abierta.java

```
package State;
public class Abierta  implements EstadoVentanilla{

    @Override
    public void atender(Persona persona) {
        System.out.println("Atendiendo a : "+ persona.getNombre());
    }
}
```

Cerrada.java

```
package State;
public class Cerrada  implements EstadoVentanilla {

    @Override
    public void atender(Persona persona) {
        System.out.println("Ventanilla cerrada !");
```

```
    }
}
```

Suspendida.java

```
package State;
public class Suspendida implements EstadoVentanilla {

    @Override
    public void atender(Persona persona) {
        if (persona.getEdad() > 65) {
            System.out.println("Atendiendo a: " + persona.getNombre());
        } else {
            System.out.println("Espera unos minutos, primero atencion preferencial");
        }
    }
}
```

Ventanilla.Java

La clase Ventanilla cambia su comportamiento según el estado en que se encuentre. Si, está cerrada, no hay atención directamente; por eso, delega el método de atención a su estado y es este mismo estado quién toma la decisión de atender o no.

```
package State;
public class Ventanilla {
    private String cajero;
    private EstadoVentanilla estado;
    public Ventanilla() {
        estado = new Abierta();
    }

    public void suspender(){
        estado = new Suspendida();
    }

    public void cerrar(){
        estado = new Cerrada();
    }

    public void abrir(){
        estado = new Abierta();
    }
    public void atender(Persona persona){
        estado.atender(persona);
    }
    public String getCajero(){
        return cajero;
```

```
    }
}
```

Prueba del código java State

Para probar el patrón, se crean múltiples instancias del tipo Persona, se instancia también un objeto del tipo Banco, luego se utiliza el método del objeto Banco con las instancias de Persona, se debe comprobar el estado del banco al momento de cerrar la ventanilla.

```
package State;
public class NewMain {
    public static void main(String[] args) {
        Persona p1 = new Persona("Jose", "corilla", 25);
        Persona p2 = new Persona("Romel", "Huaraca", 75);
        Persona p3 = new Persona("Arones", "corilla", 55);
        Persona p4 = new Persona("pamela", "Flores", 63);
        Persona p5 = new Persona("Karla", "corilla", 19);
        Persona p6 = new Persona("Annie", "Martinez", 25);

        Banco b1 = new Banco();
        b1.atender(p1);
        b1.suspendeVentanilla();
        b1.atender(p2);
        b1.atender(p3);
        b1.cerrarVentanilla();
        b1.atender(p6);
    }
}
```

Salida del código java State

En la consola se muestra como el estado del banco se comporta de manera diferente y al momento de cerrar la ventanilla, no acepta que se atiendan a más personas, el estado se mantiene.

```
Jose Ingresa a la fila
Atendiendo a: Jose
Rommel Ingresa a la fila
Atendiendo a: Rommel
Arones Ingresa a la fila
Espera unos minutos, primero atención preferencial
Annie Ingresa a la fila
Ventanilla cerrada!
```

I. Estructura del patrón Strategy adaptado

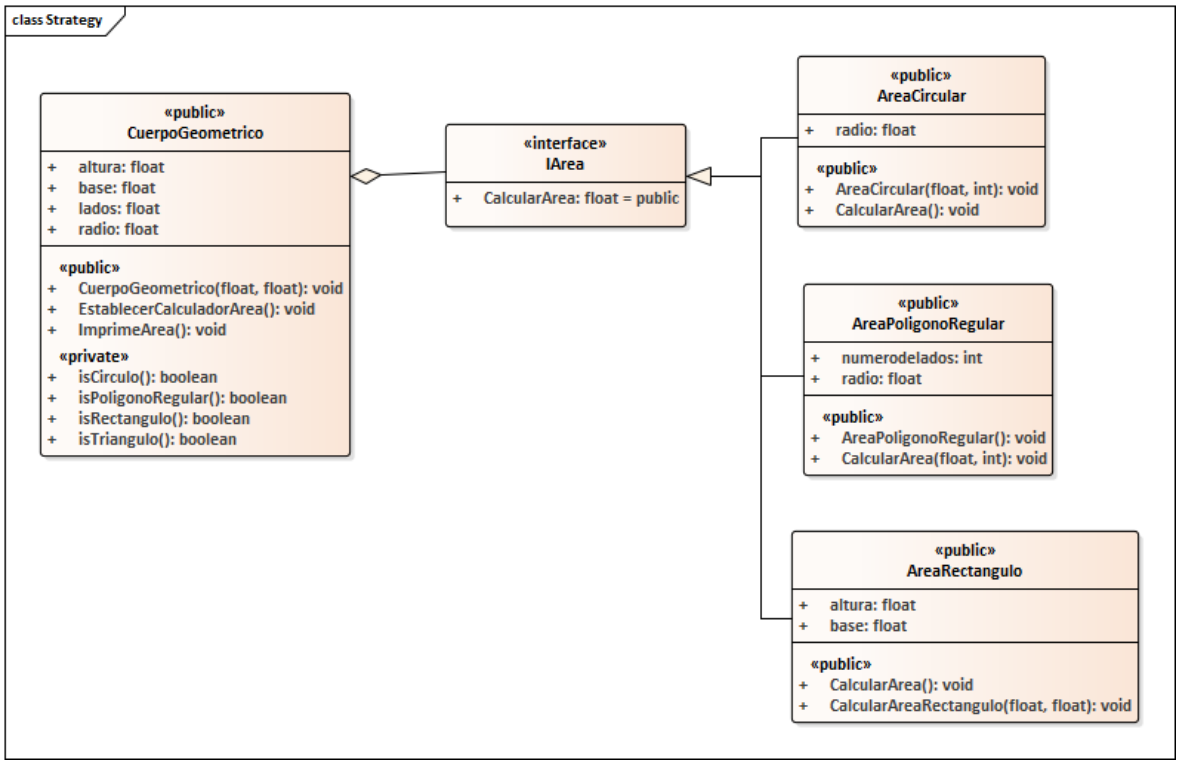


Figura 9.9. Diagrama del patrón Strategy adaptado para código java

Código adaptado a java del patrón Strategy

Ahora explicamos las clases que intervienen en la implementación del patrón Strategy en código java.

IArea.java:

La interfaz IArea, aquí definimos un método que nos ayudará a calcular el área de cualquier cuerpo geométrico, esta se implementará en la clase AbstractArea.

```
public interface IArea {
   float calculaArea();
}
```

AbstractArea.java:

La clase abstracta que implementará la interfaz IArea, donde definimos un constructor y alguna propiedad básica o común al resto de estrategias, es necesario tener definido la interface IArea.

```
public abstract class AbstractArea implements IArea {
   private String nombreFigura;
```

```
    public AbstractArea(String nombreFigura) {
        this.nombreFigura = nombreFigura;
    }

    @Override
    public abstract float calculaArea();

    public String getNombreFigura() {
        return nombreFigura;
    }

    protected void setNombreFigura(String nombreFigura) {
        this.nombreFigura = nombreFigura;
    }
}
```

AreaCircular.java:

Es la estrategia que define el cálculo del área de un círculo, se realiza multiplicando el valor del radio al cuadrado por el número PI.

```
public class AreaCircular extends AbstractArea {
    private float radio;
    public AreaCircular(float radio) {
        super("círculo");
        this.radio = radio;
    }

    @Override
    public float calculaArea() {
        return (float) (Math.PI * Math.pow(radio, 2));
    }
}
```

AreaPoligonoRegular.java:

Estrategia que define el cálculo del área de un polígono regular de “n” lados, conocido el radio de la circunferencia, se calcula multiplicando el número de lados, por el radio al cuadrado, por el seno de 2PI, dividido entre n y todo ello dividido entre 2.

```
public class AreaPoligonoRegular extends AbstractArea {
    private float radio;
    private int numeroDeLados;

    public AreaPoligonoRegular(int numeroDeLados, float radio) {
        super("polígono regular de " + numeroDeLados + " lados");
        this.numeroDeLados = numeroDeLados;
```

```
        this.radio = radio;
    }

    @Override
    public float calculaArea() {
        double numerador = numeroDeLados * Math.pow(radio, 2)
                * Math.sin((2 * Math.PI) / numeroDeLados);
        double denominador = 2;
        return (float) ((float) numerador / denominador);
    }
```

AreaRectangulo.java:

Estrategia que define el cálculo del área de un rectángulo, se lleva a cabo multiplicando el valor de la base por la altura.

```
public class AreaRectangulo extends AbstractArea {
    private float base;
    private float altura;

     public AreaRectangulo(float base, float altura) {
        super("rectángulo");
        if (base == altura) {
            super.setNombreFigura("cuadrado");
        }
        this.base = base;
        this.altura = altura;
    }

   @Override
    public float calculaArea() {
        return (base * altura);
    }
}
```

AreaTriangulo.java:

Estrategia que define el cálculo del área de un triángulo, se multiplica el valor de la base por la altura dividiéndolo entre 2.

```
public class AreaTriangulo extends AbstractArea {
    private float base;
    private float altura;

    public AreaTriangulo(float base, float altura) {
        super("triángulo");
        this.base = base;
```

```
        this.altura = altura;
    }

   @Override
    public float calculaArea() {
        return (base * altura) / 2;
    }
}
```

ConstructorIncorrectoException.java:

Clase que define la excepción de constructor incorrecto.

```
public class ConstructorIncorrectoException extends Exception {
    public ConstructorIncorrectoException() {
        super("El constructor utilizado no es correcto para ese cuerpo "
                + "geométrico.");
    }
}
```

PoligonoNoExisteException.java:

Clase que define la excepción de polígonos que no existen.

```
public class PoligonoNoExisteException extends Exception {
    public PoligonoNoExisteException() {
        super("El polígono no existe.");
    }
}
```

PoligonoNoSoportadoException.java:

Clase que define la excepción para polígonos irregulares.

```
public class PoligonoNoSoportadoException extends Exception {
    public PoligonoNoSoportadoException() {
        super("El polígono no está soportado.");
    }
}
```

CuerpoGeometrico.java:

Clase que implementa el patrón y que decidirá la estrategia a aplicar, cuenta con los métodos que establece la estrategia a desarrollar en función a los parámetros establecidos, en esta clase se utilizan las excepciones creadas (PoligonoNoExisteException excepción si no existe el polígono y

PoligonoNoSoportadoException excecepcion si es un polígono irregular). El método imprimeArea, imprime en pantalla el resultado calculado.

```
public class CuerpoGeometrico {
   private AbstractArea _estrategia;
   private final float radio;
   private final int lados;
   private final float base;
   private final float altura;

   public CuerpoGeometrico(float radio, int lados) throws
         ConstructorIncorrectoException {
      if(lados < 5)
         throw new ConstructorIncorrectoException();
      this.radio = radio;
      this.lados = lados;
      this.base = 0;
      this.altura = 0;
   }

   public CuerpoGeometrico(float radio) {
      this.radio = radio;
      this.lados = (int) Float.POSITIVE_INFINITY;
      this.base = 0;
      this.altura = 0;
   }

   public CuerpoGeometrico(float base, float altura, int lados) {
      this.radio = 0;
      this.lados = lados;
      this.base = base;
      this.altura = altura;
   }
   public void estableceCalculadoraArea() throws PoligonoNoExisteException,
         PoligonoNoSoportadoException {
      if (isCirculo()) {
         _estrategia = new AreaCircular(radio);
      } else if (isTriangulo()) {
         _estrategia = new AreaTriangulo(base, altura);
      } else if (isRectangulo()) {
         _estrategia = new AreaRectangulo(base, altura);
      } else if (isPoligonoRegular()) {
         _estrategia = new AreaPoligonoRegular(lados, radio);
      } else if (lados == 2 || lados == 0) {
         throw new PoligonoNoExisteException();
      } else {
         throw new PoligonoNoSoportadoException();
      }
   }
```

```
    public void imprimeArea() {
        System.out.println("El área del " + _estrategia.getNombreFigura()
                + " es: " + _estrategia.calculaArea());
    }

    private boolean isCirculo() {
        return lados == (int) Float.POSITIVE_INFINITY && radio != 0;
    }

    private boolean isPoligonoRegular() {
        return radio != 0 && lados != 0 && lados != 2;
    }

    private boolean isTriangulo() {
        return base != 0 && altura != 0 && lados == 3;
    }

    private boolean isRectangulo() {
        return base != 0 && altura != 0 && lados == 4;
    }
}
```

Prueba del código java Strategy

Para probar el patrón, se crea la instancia de CuerpoGeometrico, el cual es sometido a las diferentes estrategias, con parámetros diferentes a fin de probar si las estrategias heredadas son capaces de resolver la petición.

```
public class StrategyPrincipal {
    public static void main(String[] args) {
        try {
            CuerpoGeometrico p = new CuerpoGeometrico(2f);
            p.estableceCalculadoraArea();
            p.imprimeArea();
            p = new CuerpoGeometrico(2f, 3f, 3);
            p.estableceCalculadoraArea();
            p.imprimeArea();
            p = new CuerpoGeometrico(2f, 5);
            p.estableceCalculadoraArea();
            p.imprimeArea();
            p = new CuerpoGeometrico(Math.abs((float)(2f * Math.cos(90))),
                    Math.abs((float)(2f * Math.sin(90))), 4);
            p.estableceCalculadoraArea();
            p.imprimeArea();
            p = new CuerpoGeometrico(2f, 3f, 4);
            p.estableceCalculadoraArea();
            p.imprimeArea();
            p = new CuerpoGeometrico(2f, 12);
```

```
            p.estableceCalculadoraArea();
            p.imprimeArea();
            p = new CuerpoGeometrico(0, 12);
            p.estableceCalculadoraArea();
            p.imprimeArea();
        } catch (PoligonoNoExisteException | PoligonoNoSoportadoException |
                ConstructorIncorrectoException ex) {
            System.err.println(ex.getMessage());
        }
    }
}
```

Salida del código java Strategy

En la consola se muestra como CuerpoGeometrico.java, opta por una estrategia u otra, según el número de parámetros que se ingresa, también verifica si es posible calcular el área de ese polígono gracias a las excepciones creadas.

```
El área del círculo es: 12.566371
El área del triángulo es: 3.0
El polígono no está soportado.
El área del polígono regular de 5 lados es: 9.510565
El área del rectángulo es: 1.6023053
El área del rectángulo es: 6.0
El área del polígono regular de 12 lados es: 12.0
```

Anexo 8. Definición de Términos

Aseguramiento de calidad.- Un patrón planeado y sistemático de todas las acciones necesarias para proporcionar la confianza adecuada de que un artículo o producto cumple con los requisitos técnicos establecidos (IEEE, 2010).

Aseguramiento de calidad analítica.- Son todos los medios para analizar el estado de calidad de un producto (Wagner, 2013).

Aseguramiento de calidad constructiva.- Todos los medios utilizados en la construcción de un producto de manera que satisface sus requisitos de calidad (Wagner, 2013).

Colaboración basada en requisitos.- Es la colaboración impulsada por los requisitos durante su desarrollo y gestión para los productos de software posteriores, es decir, diseño, implementación, etc. (Damian, Kwan y Marczak, 2010).

Control de calidad del producto software.- Un proceso para especificar los requisitos de calidad, evaluar los artefactos creados, comparar lo deseado con la calidad real y tomar las medidas necesarias para corregir las diferencias (Wagner, 2013).

Deuda técnica.- Es una metáfora utilizada para comunicar las consecuencias de las prácticas deficientes usadas al desarrollar software.

Especificación de software.- Se define como una breve declaración de los requisitos que el software debe garantizar.

Evaluación de calidad.- Examen sistemático de la medida en que una entidad es capaz de cumplir los requisitos especificados (IEEE, 2010). Sinónimo quality assessment.

Ingeniería de requisitos.- Es parte de un proceso social centrado en el ser humano, llamado ingeniería de software (Damian, Kwan y Marczak, 2010).

La calidad interna.- representa la perspectiva del desarrollador y mantenedor (más adelante en el ciclo de vida de un sistema).

La calidad externa.- representa la perspectiva del mantenedor, operador y parcialmente del usuario final (en su aspecto de usabilidad).

La calidad en uso.- representa la perspectiva del usuario final.

Medida.- Variable a la que se le asigna un valor como resultado de la medida (IEEE, 2010). Sinónimo en ingeniería de software de métrica.

Medición.- Conjunto de operaciones que tienen el objeto de determinar el valor de una medida.

Patrón de diseño de interfaz de usuario.- Es una solución basada en el diseño de un

componente software que resuelve un problema de interacción de un usuario final con una interfaz (Palacios, et. al., 2015).

Pruebas en líneas de producto software (SPL).- Son pruebas aplicadas cuando se desarrolla software, siendo: unitarias, de integración, funcionales, de arquitectura y de sistema integrado.

Requerimiento de calidad.- Una demanda concreta y medible de un producto específico que tiene un impacto sobre un objetivo o factor de calidad de un producto software (Wagner, 2013).

ThinkLet.- Una actividad de colaboración, que genera un patrón conocido de colaboración entre personas que trabajan juntas hacia una meta.

Validación.- Es la actividad para evaluar si el resultado de un trabajo y sus requisitos se ajustan a las expectativas de los interesados (Wagner, 2013).

Verificación.- Es la actividad para evaluar si el resultado de un trabajo se ajusta a sus requisitos especificados (Wagner, 2013).