Modelo de referencia para la inclusión de sub-características de mantenibilidad al producto software durante el proceso de desarrollo

Trabajo de grado

Jennifer Dallán Erazo Martínez

Andrés Steven Florez Gómez

Director: PhD. Francisco José Pino Correa

Universidad del Cauca

Facultad de Ingeniería Electrónica y Telecomunicaciones

Departamento de Sistemas

Grupo de Investigación y Desarrollo en Ingeniería de Software

Línea de Investigación de Calidad de Proceso y Producto

Popayán, Junio de 2015

AGRADECIMIENTOS

A nuestras familias por su apoyo incondicional, esfuerzo, comprensión y confianza depositada en nosotros para cumplir con este objetivo.

Al PhD. Francisco José Pino Correa por su dedicación, tiempo, apoyo y todo el conocimiento brindado para el desarrollo de este trabajo.

A todos los docentes de la Universidad del Cauca que nos compartieron y transmitieron sus conocimientos.

A nuestros amigos y compañeros que nos brindaron su apoyo durante toda la carrera.

TABLA DE CONTENIDO

CAPITULO I: INTRODUCCIÓN 7

1.1. OBJETIVOS 8
1.1.1. OBJETIVO GENERAL 8
1.1.2. OBJETIVOS ESPECÍFICOS 8
1.2. ESTRATEGIA DE INVESTIGACIÓN UTILIZADA 9
1.3. ESTRUCTURA DEL DOCUMENTO 10

CAPITULO II: MARCO TEÓRICO Y ESTADO DEL ARTE 12

2.1. MARCO TEÓRICO 12
2.1.1. MANTENIMIENTO DE SOFTWARE 12
2.1.2. MANTENIBILIDAD DE SOFTWARE 12
2.1.3. MODELO DE REFERENCIA (ISO/IEC 33004) 13
2.1.4. ISO/IEC 12207, PROCESOS DE IMPLEMENTACIÓN DE SOFTWARE Y PROCESO DE OPERACIÓN DE SOFTWARE. 13
2.1.4.1. ISO/IEC 12207: Systems and software engineering – Software life cycle processes. 14
2.1.4.2. Procesos de implementación de software 14
2.1.4.2.1. Proceso de análisis de requisitos de software 14
2.1.4.2.2. Proceso de diseño arquitectural de software 14
2.1.4.2.3. Procesos de diseño detallado de software 14
2.1.4.2.4. Proceso de construcción de software 14
2.1.4.2.5. Proceso de Integración de software 15
2.1.4.2.6. Proceso de pruebas de evaluación de software 15
2.1.4.2.7. Proceso de operación de software 15
2.1.5. ATRIBUTO SOFTWARE 15
2.1.6. CERTIFICACIÓN DE MANTENIBILIDAD AQC LAB 15
2.2. ESTADO DEL ARTE 16
2.2.1. ESTRATEGIA DE BÚSQUEDA 16
2.2.2. CRITERIOS DE INCLUSIÓN Y EXCLUSIÓN 16
2.2.3. ESTUDIOS RELACIONADOS CON METODOLOGÍAS DE MANTENIMIENTO 17
2.2.3.1. Modelo de madurez de mantenimiento de software 17
2.2.3.2. Mejora de la calidad del proceso de mantenimiento 17
2.2.3.3. Metodología para el apoyo de mantenimiento basado en el estándar ISO/IEC 12207 17
2.2.3.4. Metodología de mantenimiento de software para pequeñas organizaciones 18
2.2.4. ESTUDIOS RELACIONADOS CON ATRIBUTOS DE MANTENIBILIDAD. 18
2.2.4.1. Factores claves que afectan la mantenibilidad de software 18
2.2.4.2. Impacto de atributos internos del producto software sobre la mantenibilidad 19
2.2.4.3. Modelo de atributos de mantenibilidad de software 19

2.2.4.4. Efecto de los patrones de diseño en la mantenibilidad de software 19
2.2.4.5. Arquitectura y mantenibilidad de software 19
2.2.5. ESTUDIOS RELACIONADOS CON MÉTRICAS DE MANTENIBILIDAD 20
2.2.5.1. Modelo práctico para medir la mantenibilidad... 20
2.2.5.2. Métricas de acoplamiento para predecir la mantenibilidad en diseños orientados a servicios.. 21
2.2.5.3. Métricas de mantenibilidad para sistemas orientados a objetos 21
2.2.6. APORTES.. 21

CAPITULO III: MODELO DE REFERENCIA PARA LA INCLUSIÓN DE SUB-CARACTERÍSTICAS DE MANTENIBILIDAD AL PRODUCTO SOFTWARE 23

3.1. FUNDAMENTOS TEÓRICOS 24
3.1.1. IDENTIFICAR LOS ATRIBUTOS DE SOFTWARE QUE INFLUYEN EN LA MANTENIBILIDAD DEL PRODUCTO. 24
3.1.2. AGRUPAR CONCEPTOS 30
3.1.3. FILTRAR ATRIBUTOS 33
3.2. CLASIFICACIÓN Y ANÁLISIS DE ATRIBUTOS DE MANTENIBILIDAD 35
3.2.1. CLASIFICAR LOS ATRIBUTOS IDENTIFICADOS.. 35
3.2.1.1. Clasificar los atributos por sub-caracteristicas y flujos de trabajo. 35
3.2.1.2. Clasificar los atributos de acuerdo a los procesos 43
3.2.2. ANALIZAR COMO ES POSIBLE POTENCIAR CADA SUB-CARACTERÍSTICA. 44
3.2.2.1. Sub-característica de reusabilidad.. 44
3.2.2.2. Sub-característica de capacidad para ser analizado.. 49
3.2.2.3. Sub-característica de modularidad 52
3.2.2.4. Sub-característica capacidad para ser modificado... 53
3.2.2.5. Sub-característica capacidad para ser probado.. 56
3.2.3. ESTRUCTURAR EL MODELO DE REFERENCIA GENERAL. 59
3.2.3.1. Descripción general del modelo.. 59
3.2.3.1.1. Objetivo 59
3.2.3.1.2. Procesos del modelo 59
3.2.3.1.3. Relaciones entre los procesos del modelo.. 60
3.2.3.1.4. Comunidad de interés.. 62
3.2.3.1.5. Contexto previsto de uso 62
3.2.3.1.6. Búsqueda de consenso 62
3.2.3.2. Descripción detallada de los procesos... 62
3.2.4. PLANTEAR LOS MODELOS DE REFERENCIA ESPECÍFICOS. 76

CAPITULO IV: APLICACIÓN DEL ESTUDIO DE CASO 83

4.1. ANTECEDENTES 83
4.2. DISEÑO 83
4.3. SUJETOS DE INVESTIGACIÓN Y UNIDAD DE ANÁLISIS.. 84

4.4. PROCEDIMIENTO DE CAMPO 87
4.5. PREPARACIÓN PARA LA RECOLECCIÓN DE DATOS 87
4.6. EJECUCIÓN DEL CASO 90
4.6.1. DATOS RECOLECTADOS 91
4.7. ANÁLISIS DE RESULTADOS 94
4.8. PLAN DE VALIDACIÓN 111
4.9. LIMITACIONES DEL ESTUDIO 112

CAPITULO V: CONCLUSIONES Y TRABAJO FUTURO 113

5.1. CONCLUSIONES 113
5.2. LECCIONES APRENDIDAS 114
5.3. TRABAJO FUTURO 115

REFERENCIAS BIBLIOGRÁFICAS 116

LISTA DE FIGURAS

Figura 1. Estrategia de Investigación 9
Figura 2. Proceso para la obtención de atributos 24
Figura 3. Vista especifica del modelo de referencia 61
Figura 4. Vista general del modelo de referencia 62
Figura 5. Estructura código B 85
Figura 6. Estructura código A 86

LISTA DE GRÁFICOS

Gráfico 1. Tasa de éxito ítem 1 95
Gráfico 2. Tasa de éxito ítem 2 96
Gráfico 3. Tasa de éxito ítem 3 97
Gráfico 4. Tasa de éxito ítem 4 98
Gráfico 5. Tasa de éxito ítem 6 99
Gráfico 6. Tasa de éxito ítem 7 100
Gráfico 7. Tasa de éxito ítem 8 101
Gráfico 8. Total porcentaje de éxito 102
Gráfico 9. Tasa de éxito y tiempo totales de cambios 103
Gráfico 10. Tasa de éxito en el encuentro de causa de fallos 104
Gráfico 11. Línea de tiempo ítem 4 106
Gráfico 12. Línea de tiempo ítem 5.1 106
Gráfico 13. Línea de tiempo ítem 5.2 107
Gráfico 14. Línea de tiempo ítem 5.3 107

LISTA DE TABLAS

Tabla 1. Relación entre métricas y atributos. 35
Tabla 2. Atributos finales 35
Tabla 3. Clasificación de los atributos por sub-características de mantenibilidad y flujos de trabajo.... 42
Tabla 4. Clasificación de los atributos y procesos. 44
Tabla 5. Prácticas para la sub-característica de reusabilidad.... 48
Tabla 6. Prácticas de documentación para la sub-característica de reusabilidad.... 49
Tabla 7. Procesos que componen el modelo de referencia.... 60
Tabla 8. Características de productos de trabajo de salida 76
Tabla 9. Prácticas aplicadas en el estudio de caso.... 87
Tabla 10. Duración ítem 6.... 92
Tabla 11. Tiempo modificaciones.... 93
Tabla 12. Tiempo análisis 94
Tabla 13. Frecuencia de éxito de modificación ítem 1.... 94
Tabla 14. Frecuencia de éxito de modificación ítem 2.... 95
Tabla 15. Frecuencia éxito de modificación ítem 3 96
Tabla 16. Frecuencia de éxito de modificación ítem 4.... 97
Tabla 17. Frecuencia de éxito de modificación ítem 6.... 98
Tabla 18. Frecuencia de éxito de modificación ítem 7.... 99
Tabla 19. Frecuencia de éxito de modificación ítem 8.... 100
Tabla 20. Porcentaje de éxito grupo A.....101
Tabla 21. Porcentaje de éxito grupo B 101
Tabla 22. Tiempo total por participante.... 102
Tabla 23. Cálculo TEF 104
Tabla 24. Cálculo TAF 105
Tabla 25. Tabla respuestas encuesta 109

CAPITULO I

INTRODUCCIÓN

La mantenibilidad es uno de los atributos de calidad esenciales, ya que las tareas de mantenimiento consumen una gran proporción del esfuerzo total gastado en el ciclo de vida del software [1]. El costo de esta etapa consume entre el 50% y el 80% de los recursos del proyecto [2] y el 66% de los costos del ciclo de vida del software son invertidos en el mantenimiento del producto [3]. Además, el 61% del tiempo que dedican los programadores al desarrollo es invertido en la etapa de mantenimiento, y sólo el 39% es empleado en nuevos desarrollos [4]. Lo anterior refleja que la etapa de mantenimiento: (1) requiere el mayor porcentaje de los costos del ciclo de vida del software, (2) incrementa el esfuerzo realizado, (3) impide que una gran parte del tiempo sea utilizado para nuevos desarrollos.

Dado que la etapa de mantenimiento genera altos costos durante el ciclo de vida del desarrollo de software, para facilitar su ejecución es conveniente tener en cuenta la característica de mantenibilidad de software. ISO/IEC 25010 [5] define la mantenibilidad como el grado de efectividad o eficiencia con la que un producto o sistema puede ser modificado y la divide en cinco sub-características: modularidad (modularity), reusabilidad (reusability), capacidad para ser analizado (analysability), capacidad para ser modificado (modifiability) y capacidad para ser probado (testability). Incluir estas sub-características al producto software podría ayudar a incrementar la mantenibilidad, reduciendo así el esfuerzo de mantenimiento, lo que permitiría usar estos mismos recursos para realizar más cambios o lograr los mismos cambios con menos recursos [6].

Al realizar la evaluación de mantenibilidad del producto software, si éste no incorpora las sub-características de mantenibilidad los resultados podrían ser negativos lo que generaría la necesidad de realizar cambios o mejoras al producto con el fin de incluir dichas sub-características. Pero como este proceso puede resultar costoso para las organizaciones, estas podrían decidir no realizarlo obteniendo como resultado un producto poco mantenible.

Se puede considerar que el problema de costos durante la etapa de mantenimiento reside en que no se tienen en cuenta las sub-características de mantenibilidad durante el desarrollo del producto software, por lo cual se hace necesario tener un modelo de referencia[1] que permita complementar el proceso de desarrollo incluyendo las sub-características de mantenibilidad durante las diferentes etapas del ciclo de vida del producto. Sin embargo, no se han encontrado soluciones que aborden este problema.

La mantenibilidad de software se descompone en una serie de atributos que pueden ser considerados durante diferentes etapas del proceso de desarrollo. Estudios previos han demostrado que una de las causas de los problemas en el mantenimiento del software es que a menudo la mantenibilidad no es una consideración importante durante las etapas de diseño e implementación de software [3]. Realizar más esfuerzo durante el ciclo de vida de desarrollo para hacer que el software sea mantenible puede reducir significativamente el total de los costos del software [7]. Por esto, es importante conocer

[1] Modelo de referencia de proceso: define un conjunto de procesos caracterizados en términos de propósito de proceso y salidas de proceso. (Definición tomada de: ISO/IEC 15504-2:2003)

los atributos de software que afectan a la mantenibilidad y clasificarlos en la etapa adecuada del proceso de desarrollo, de esta forma se podría lograr incluir las sub-características de mantenibilidad al producto software para conseguir un producto altamente mantenible.

El tener un modelo de referencia que cuenta con un conjunto de procesos en los cuales se agrupan los atributos que influyen sobre la mantenibilidad y que además permita identificar los atributos que afectan a cada sub-característica puede facilitar la inclusión de la mantenibilidad al producto software, logrando así conseguir un producto altamente mantenible. Actualmente, existe una certificación de mantenibilidad del producto software, la cual es realizada por AQC Lab [8], el modelo de referencia podría ser utilizado por las empresas con el fin de obtener esta certificación.

Bajo este contexto es importante definir un modelo de referencia de procesos que permita incluir sub-características de mantenibilidad al producto software. El alcance de este trabajo es identificar los atributos de software que influyen en la mantenibilidad, clasificarlos de acuerdo a la sub-característica adecuada y asignarlo a la etapa de desarrollo correspondiente de acuerdo a los procesos definidos en el Estándar Internacional ISO/IEC 15504-5:2011 [9].

En este sentido, el presente trabajo de investigación intenta dar respuesta a la siguiente pregunta de investigación:

¿Cómo incorporar sub-características de mantenibilidad al producto software durante el desarrollo del software?

1.1. Objetivos

A continuación se presentan los objetivos de este trabajo de investigación:

1.1.1. Objetivo General

- Proponer un modelo de referencia[2] que apoye la inclusión de sub-características de mantenibilidad al producto software durante el proceso de desarrollo.

1.1.2. Objetivos Específicos

- Determinar los aspectos básicos[3] que se deben tener en cuenta durante el desarrollo de software para incrementar la mantenibilidad del producto.
- Definir las prácticas base[4] que podrían potenciar cada una de las sub-características de mantenibilidad durante el proceso de desarrollo.
- Agrupar las prácticas base definidas en los diferentes procesos que harán parte del modelo de referencia de proceso a desarrollar.

[2] Modelo de referencia de proceso: define un conjunto de procesos caracterizados en términos de propósito de proceso y salidas de proceso. (Definición tomada de: ISO/IEC 15504-2:2003)
[3] Aspectos básicos: aspectos que son la base sobre los cuales se sustenta una cosa. (Definición tomada de: Diccionario de la lengua española © 2005 Espasa-Calpe)
[4] Prácticas base: actividades que se refieren al propósito de un proceso particular. Están descritas en un nivel abstracto, identificando “que” se debería hacer sin especificar “como”. (Definición tomada de: ISO/IEC 15504-5:2011)

- Evaluar el modelo de referencia propuesto mediante el método de estudio de caso implementándolo parcialmente en una entidad desarrolladora de software.

Con el cumplimiento de estos objetivos se desea aportar en:

Determinar los atributos de software que influyen sobre las diferentes sub-características de la mantenibilidad del producto.

Proponer un modelo de referencia que permita incluir al producto software las características de mantenibilidad a lo largo del proceso de desarrollo, para obtener un producto altamente mantenible.

Con el modelo de referencia propuesto se podrían disminuir los costos de mantenimiento durante el proceso de desarrollo y además las empresas podrían utilizar este modelo con el fin de obtener la certificación en mantenibilidad.

1.2. Estrategia de Investigación utilizada

Para el desarrollo de este trabajo de investigación se utiliza una estrategia de investigación (ver Figura 1) que se apoya en el método Investigación-Acción multiciclo con bifurcación [10] [11]. La estrategia de investigación comienza con un ciclo de investigación preliminar en el cual se establecen dos problemas principales: conceptual y metodológico. Lo anterior permite estructurar el trabajo en dos ciclos de investigación independientes: ciclo de investigación conceptual y ciclo de investigación metodológico. El ciclo de investigación metodológico está asociado con un ciclo de evaluación que permite valorar el modelo de referencia propuesto.

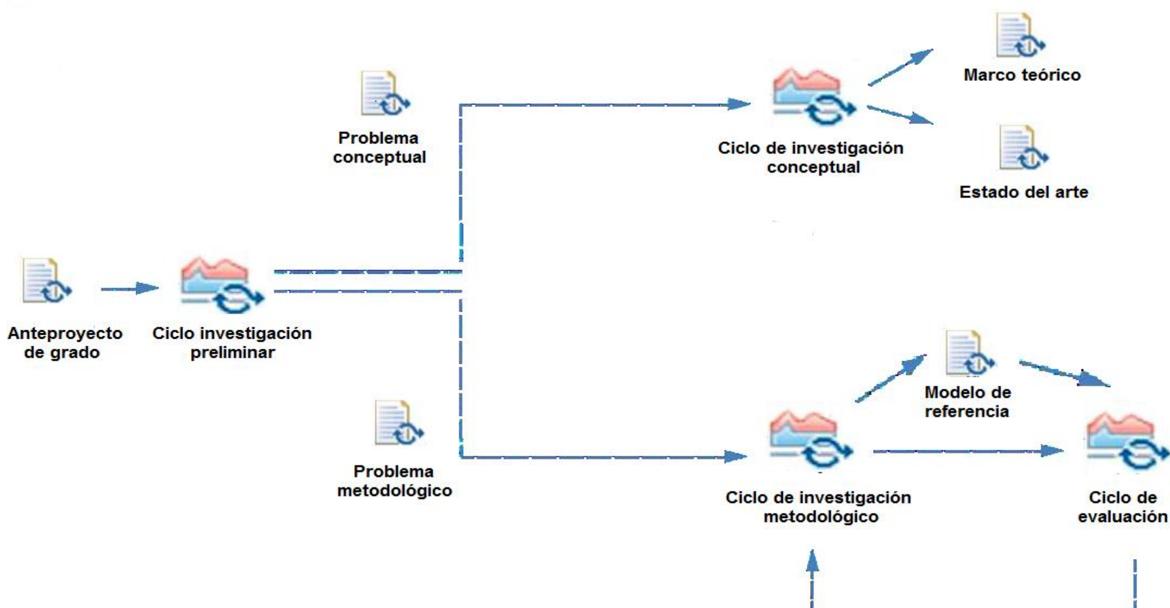

Figura 1. Estrategia de Investigación

Con el fin de conocer el contexto en el que se encuentra enmarcado el presente proyecto, se ejecuta el ciclo de investigación conceptual, el cual se compone de dos fases:

identificación del problema y estudio de la literatura. De estas fases se adquiere la información necesaria para estructurar el marco conceptual y el estado del arte. En la fase de identificación del problema se analizan las propuestas relacionadas con la incorporación de características de mantenibilidad al producto software durante el proceso de desarrollo. Por otra parte, en la fase de estudio de la literatura se revisan las características de mantenibilidad de software, los atributos que influyen sobre la mantenibilidad de software y se realiza un análisis de los trabajos relacionados.

Para la definición del modelo de referencia se ejecuta el ciclo de investigación metodológico, este ciclo está dividido en tres fases: identificación de los atributos software, clasificación de los atributos software y realización del modelo de referencia. En la fase de identificación de los atributos software se revisan y analizan los atributos del producto software y elementos del desarrollo de software que influyen en la mantenibilidad del producto. Para esto se hace una revisión de la literatura y se extraen los atributos software y sus definiciones propuestos en diversas investigaciones y posteriormente se obtiene una definición final que agrupa los conceptos encontrados en los diferentes estudios. En la fase de clasificación de los atributos software los elementos identificados se asignan a la etapa de desarrollo de software correspondiente, y se relacionan con las sub-características de mantenibilidad sobre las que influyen. Los atributos se clasifican de acuerdo a los procesos definidos en el estándar internacional ISO/IEC 15504-4:2011 [9] y de acuerdo a las sub-características de mantenibilidad definidas en el estándar internacional ISO/IEC 25010:2011 [5]. Por último en la fase de realización del modelo de referencia se definen las actividades necesarias para incorporar al producto software cada uno de los elementos identificados con el fin de potenciar las sub-características de mantenibilidad relacionadas. Se define el modelo de referencia que permita incorporar las características de mantenibilidad de software al producto durante el proceso de desarrollo.

Con el fin de evaluar el modelo de referencia propuesto se realiza un ciclo de evaluación que está ligado al de investigación metodológico. Durante éste se realiza la evaluación del modelo de referencia propuesto mediante un estudio de caso. Se divide en tres fases: diseño del estudio, ejecución del estudio y análisis y conclusiones. En la fase de diseño del estudio se establecen los objetivos del estudio de caso, se realiza el diseño y se elabora la estructura. Por su parte, en la fase de ejecución del estudio se prepara la actividad de recolección de datos y se recoge la evidencia. Finalmente, en la fase de análisis y conclusiones se analizan los resultados obtenidos en el estudio de caso.

En forma paralela se realizará la documentación adecuada desde el primer ciclo hasta el último.

1.3. Estructura del documento

A continuación se describe de manera general el contenido y organización de este documento.

El *Capítulo II – Marco teórico y estado del arte* está dividido en tres secciones: *Marco teórico* donde se encuentran los conceptos importantes para el desarrollo de esta investigación relacionados con mantenimiento de software, mantenibilidad de software y modelos de referencia de procesos; *Estado del arte* donde se presentan los trabajos relevantes por su relación con el tema de esta investigación y *Aportes* donde se describen las principales contribuciones de esta investigación.

El *Capítulo III – Modelo de referencia para la inclusión de sub-características de mantenibilidad al producto software* se compone de tres secciones: *Fundamentos teóricos* donde se obtienen los atributos de software que influyen sobre la mantenibilidad del producto; *Clasificación y análisis de atributos de mantenibilidad* donde encuentra la relación de los atributos identificados con las sub-características de mantenibilidad sobre las que influye y las etapas del proceso de desarrollo en las que se presentan, además de esto se analiza cómo es posible potenciar cada sub-característica y *Descripción de los procesos* donde se estructura el modelo de referencia general y se plantean los modelos de referencia específicos.

El *Capítulo IV – Aplicación del estudio de caso* contiene nueve secciones: *Antecedentes* donde se encuentran los estudios relacionados con mantenibilidad del software y se plantean la pregunta investigación principal y las preguntas adicionales; *Diseño* se definen el tipo del estudio de caso y las medidas usadas para indagar sobre las preguntas de investigación; *Sujetos de investigación y unidad de análisis*; *Procedimiento de campo* donde se establece el protocolo a seguir en el estudio de caso; *Preparación de recolección datos* que presenta los formatos utilizados; *Ejecución del caso* donde se presenta la preparación, ejecución, el seguimiento del estudio de caso y datos obtenidos en el estudio de caso; *Análisis de resultados* en el que se calculan y analizan las medidas con el fin de dar respuesta a las preguntas de investigación planteadas; *Plan de validación* que contiene las consideraciones para disminuir las amenazas a la validez del estudio de caso y *Limitaciones del estudio*.

El *Capítulo V – Conclusiones y trabajo futuro* que describe las conclusiones, lecciones aprendidas y trabajos futuros establecidos a partir de la investigación realizada.

CAPITULO II

MARCO TEÓRICO Y ESTADO DEL ARTE

Con el fin de tener una clara comprensión de este trabajo de investigación se hace importante precisar los estudios realizados y conceptos relacionados con el desarrollo del mismo. El objetivo de este capítulo es plantear: el marco teórico donde se desarrollan los conceptos o definiciones significativas para un mejor entendimiento de este trabajo, y el estado del arte donde se describen las investigaciones más relevantes relacionadas con el tema planteado por este trabajo de investigación.

2.1. Marco Teórico

2.1.1. Mantenimiento de Software

IEEE Standard 1219:1998 [12] define el mantenimiento de software como la modificación de un producto software después de su entrega para corregir fallas, mejorar el desempeño u otros atributos, o para adaptar el producto a un entorno modificado. Además de esto, el Standard define las siguientes categorías para el mantenimiento:

- **Mantenimiento Adaptativo:** Modificación de un producto software realizada después de la entrega para enfrentarse a los cambios en el entorno.
- **Mantenimiento Perfectivo:** Modificación de un producto software después de su entrega para mejorar el desempeño o mantenibilidad.
- **Mantenimiento Correctivo:** Modificación reactiva de un producto software después de su entrega para corregir fallas descubiertas.
- **Mantenimiento Preventivo:** Mantenimiento realizado con el propósito de prevenir problemas antes de que ocurran.
- **Mantenimiento Urgente:** Mantenimiento correctivo no programado realizado para que un sistema continúe operando.

2.1.2. Mantenibilidad de Software

ISO/IEC 25010 [5] define la mantenibilidad como capacidad del producto software para ser modificado efectiva y eficientemente. La mantenibilidad puede ser interpretada como una capacidad inherente del producto o sistema para facilitar las actividades de mantenimiento, o la calidad en uso experimentada por los mantenedores con el objetivo de realizar modificaciones al producto o sistema. Esta característica se divide en:

- **Modularidad (Modularity):** Grado en el cual un sistema o programa de computador está compuesto por componentes discretos de tal forma que un cambio en un componente tiene un impacto mínimo en los otros componentes.
- **Reusabilidad (Reusability):** Capacidad de un activo para ser utilizado en más de un sistema software o en la construcción de otros activos.
- **Capacidad para ser analizado (Analysability).** Facilidad para evaluar el impacto de un determinado cambio sobre el resto del software, diagnosticar las deficiencias o causas de fallos en el software, o identificar las partes a modificar.
- **Capacidad para ser modificado (Modifiability):** Capacidad del producto para ser modificado de forma efectiva y eficiente sin introducir defectos o degradar la calidad del

producto existente. Esta sub-característica puede ser influenciada por la capacidad para ser analizado y la modularidad.

- **Capacidad para ser probado (Testability):** Facilidad con la que se pueden establecer criterios de prueba para un sistema o componente y con la que se pueden llevar a cabo las pruebas para determinar si se cumplen dichos criterios.

2.1.3. Modelo de referencia (ISO/IEC 33004)

El propósito de un modelo de referencia de procesos es definir un conjunto de procesos que colectivamente puedan alcanzar el propósito principal de una comunidad de interés [13]. Esta norma define los requerimientos para los modelos de referencia de procesos, donde se indica lo que debe contener el modelo de referencia:

a) Una declaración del dominio del modelo de referencia de proceso.
b) Una descripción de procesos dentro del alcance del modelo de referencia de proceso.
c) Una descripción de la relación entre el modelo de referencia del proceso y su contexto de uso previsto.
d) Una descripción de la relación entre los procesos definidos dentro del modelo de referencia del proceso.

Además esta norma indica que el modelo de referencia de proceso debe documentar la comunidad de interés del modelo y las acciones tomadas para alcanzar un consenso dentro de esa comunidad de interés:

a) La comunidad de interés pertinente debe ser caracterizada o especificada.
b) El grado de alcance de consenso de la comunidad de interés debería ser documentado.
c) Si no se han tomado acciones para lograr el consenso, esto debería ser documentado.

En cuanto a los procesos definidos dentro del modelo de referencia esta norma establece que los mismos deben tener una única descripción e identificación. Estas descripciones de procesos son el elemento fundamental de modelo de referencia de procesos y deben cumplir con los siguientes requerimientos:

a) Un proceso debe ser descrito en términos de su propósito y salidas de proceso.
b) En cualquier descripción de proceso el conjunto de salidas del proceso debe ser el necesario y suficiente para alcanzar el propósito del proceso.
c) Las descripciones de procesos no deberían contener o implicar aspectos de características de calidad del proceso más allá de los más bajos niveles de su escala de medición esperada.

Por último esta norma señala que una salida de proceso describe una de las siguientes:

- Producción de un artefacto
- Un cambio significativo del estado.
- Cumplimiento de restricciones específicas.

2.1.4. ISO/IEC 12207, procesos de implementación de software y proceso de operación de software.

2.1.4.1. ISO/IEC 12207: Systems and software engineering – Software life cycle processes.

Este Estándar internacional establece un marco común para los procesos del ciclo de vida del software, con una terminología bien definida, que puede ser referenciada por la industria de software. Contiene procesos, actividades, y tareas que son aplicadas durante la adquisición de un producto software o servicio y durante el suministro, desarrollo, operación, mantenimiento y eliminación de productos software [14]. Esta norma agrupa las actividades que se deben realizar durante el ciclo de vida de un sistema software en siete grupos de procesos. Cada uno de estos procesos esta descrito en términos de propósito, salidas deseadas y listas de actividades y tareas que deben ser realizadas para alcanzar esas salidas. Los grupos de procesos definidos son:

- Procesos de acuerdo
- Procesos organizacionales de habilitación de proyectos
- Procesos de proyecto
- Procesos técnicos
- Procesos de implementación de software
- Procesos de soporte de software
- Procesos de reutilización de software

2.1.4.2. Procesos de implementación de software

Los procesos de implementación de software son utilizados para implementar como software un elemento específico del sistema. Estos procesos transforman comportamientos especificados, interfaces y restricciones de implementación en acciones de implementación que dan como resultado un elemento del sistema que satisface los requisitos del sistema. El grupo de procesos de implementación de software está conformado por seis procesos de bajo nivel [14]:

2.1.4.2.1. Proceso de análisis de requisitos de software

El propósito de este proceso es establecer los requerimientos de los elementos software del sistema.

2.1.4.2.2. Proceso de diseño arquitectural de software

El propósito de este proceso es proporcionar un diseño para el software que implemente y que pueda ser verificado con los requisitos.

2.1.4.2.3. Procesos de diseño detallado de software

El propósito de este proceso es proporcionar un diseño para el software que implemente y que pueda ser verificado con los requisitos y la arquitectura de software y que este lo suficientemente detallado para permitir la codificación y pruebas.

2.1.4.2.4. Proceso de construcción de software

El propósito de este proceso es producir unidades de software ejecutable que reflejen propiamente el diseño de software.

2.1.4.2.5. Proceso de Integración de software

El propósito de este proceso es combinar las unidades y componentes de software para producir ítems de software integrados, consistentes con el diseño de software, que demuestren que los requerimientos funcionales y no funcionales se satisfacen en una plataforma operacional completa o equivalente.

2.1.4.2.6. Proceso de pruebas de evaluación de software

El propósito de este proceso es confirmar que el producto software integrado cumple con los requerimientos definidos.

2.1.4.2.7. Proceso de operación de software

Este proceso pertenece al grupo de procesos técnicos y tiene como propósito operar el producto software en su entorno destinado y proporcionar soporte a los clientes del producto software.

2.1.5. Atributo software

ISO/IEC 25010 [5] define atributo como propiedad inherente o característica de una entidad que puede ser distinguida cualitativa o cuantitativamente por medios humanos o automatizados.

2.1.6. Certificación de mantenibilidad AQC Lab

Actualmente existe una certificación en mantenibilidad del producto software, la cual es realizada por AQC Lab [8]. Para ello se apoya en la norma internacional sobre calidad del producto software ISO/IEC 25010 SQuaRE. Se ha seleccionado la mantenibilidad como primera característica de calidad ya que el mantenimiento es una de las etapas del ciclo de vida de desarrollo más costosa, que genera el 80% de los costos del proceso de desarrollo, es una de las características más demandadas hoy en día por los clientes software y además porque realizar mantenimiento a un producto con baja mantenibilidad puede producir nuevos errores. El modelo de calidad que utiliza AQC Lab en sus evaluaciones define un conjunto de métricas de producto, a partir de las cuales se obtienen los valores de mantenibilidad y sus sub-características. En cuanto al proceso de evaluación cumpliendo con los requisitos de ISO/IEC 25040, AQC Lab realiza las siguientes actividades [15]:

- **Establecer los requisitos de la evaluación:** cuyo objetivo es definir el propósito de la evaluación, los requisitos de calidad que se deben considerar, las partes involucradas y el rigor de la evaluación.
- **Especificar la evaluación:** cuyo objetivo es determinar las métricas, técnicas y herramientas que se utilizarán para llevar a cabo la evaluación.
- **Diseñar la evaluación:** cuyo fin es definir el plan con las actividades de evaluación que se deben llevar a cabo.
- **Ejecutar la evaluación:** cuya meta es obtener las mediciones y aplicar los criterios de evaluación determinados en las actividades anteriores.
- **Concluir la evaluación:** que finalmente permite analizar los resultados y elaborar un informe descriptivo para que la organización evaluada conozca la calidad de su producto software.

2.2. Estado del Arte

Al realizar una búsqueda en la literatura acerca de la inclusión de características de mantenibilidad al producto software durante el proceso de desarrollo y temas relacionados, se han encontrados diferentes estudios relacionados con: metodologías de mantenimiento, atributos de mantenibilidad y métricas de mantenibilidad. Con este fin se definieron la estrategia de búsqueda y criterios de inclusión y exclusión presentados en las siguientes secciones.

2.2.1. Estrategia de Búsqueda

El estudio se realizó consultando bases de datos electrónicas, primero se seleccionaron las bases de datos para cumplir con los objetivos de la búsqueda acerca de la inclusión de características de mantenibilidad al producto software durante el proceso de desarrollo. Inicialmente la biblioteca digital utilizada para realizar la investigación fue Scopus, sin embargo debido a que varios artículos no se podían acceder gratuitamente, ésta fue cambiada por Google Scholar.

Para realizar la búsqueda automática en la biblioteca digital seleccionada, se utilizó la cadena de búsqueda definida a continuación:

“software product” AND “maintainability”

La cadena contiene dos partes que incluyen los conceptos necesarios para tener un gran alcance en la investigación acerca de la mantenibilidad del producto software. La primera parte está relacionada con los estudios que hacen referencia al producto software, la segunda parte corresponde a las investigaciones que realizan estudios de mantenibilidad. La cadena de búsqueda se componen de palabras en inglés, se emplea el conector booleano AND, para unir las dos partes de la cadena.

Esta cadena fue utilizada en la base de datos electrónica establecida inicialmente, los artículos obtenidos se examinaron para comprobar su relevancia en esta investigación.

La estrategia de búsqueda fue consultar en Google Scholar con la cadena de búsqueda definida, a los artículos encontrados se les aplicó el criterio de inclusión, los que cumplieron con este criterio fueron denominados estudios relevantes, a partir de los cuales se obtuvieron los denominados estudios primarios mediante la aplicación del criterio de exclusión [16]. Fue necesario, realizar una búsqueda manual, revisando las referencias de los artículos relevantes. El periodo de revisión para los estudios relevantes, incluye trabajos publicados entre los años 2005 y 2014.

2.2.2. Criterios de Inclusión y Exclusión

El principal criterio para incluir una publicación, fue que su investigación fuera pertinente al estudio realizado, es decir, la inclusión de características de mantenibilidad al producto software durante el proceso de desarrollo. El criterio de inclusión fue, leer el resumen de los artículos obtenidos a partir de la búsqueda automática y determinar si eran adecuados para la investigación, es decir, si planteaban una propuesta que incorpore características de mantenibilidad al producto software o si planteaban algún estudio relacionado con temas de mantenibilidad de software.

El criterio de exclusión fue leer completamente los estudios relevantes, los artículos que mencionaban la mantenibilidad de software pero que no la tenían como tema principal no fueron tenidos en cuenta para el estudio realizado.

Al realizar la búsqueda automática se obtuvo un total de 292 artículos, de los cuales: 124 abordaban el tema de atributos de software, 45 trataban sub-características de mantenibilidad, 123 tenían como tema principal métricas y mantenibilidad del producto. Luego de leer el resumen de estos artículos se obtuvieron los 47 estudios relevantes y finalmente los 12 estudios primarios.

A continuación se presenta los 12 estudios encontrados al seguir la estrategia de búsqueda definida y una discusión acerca de los mismos.

2.2.3. Estudios relacionados con metodologías de mantenimiento

2.2.3.1. Modelo de madurez de mantenimiento de software

April et. al. [17] proponen un modelo de madurez de mantenimiento de software (SMmm) para las actividades diarias del mantenimiento de software que permite evaluar y mejorar esta etapa. Este modelo aborda las actividades de mantenimiento, ayuda a identificar actividades de mejora de mantenimiento y está diseñado como un complemento para el modelo Capability Maturity Model integration (CMMi). En él se propone reagrupar los procesos de mantenimiento de software en tres grupos: Procesos primarios, que son procesos operacionales de mantenimiento de software, procesos de soporte que apoyan a los procesos primarios y por último los procesos organizacionales.

2.2.3.2. Mejora de la calidad del proceso de mantenimiento

Polo et. al. [18] presentan un enfoque del proceso de mantenimiento propuesto en MANTEMA una metodología para el mantenimiento de software que integra todas las actividades relacionadas con este proceso, se centra en el mantenimiento de software buscando solucionar el problema de los altos costos. Existen dos propuestas para resolver problemas de mantenimiento: soluciones técnicas y soluciones administrativas, en MANTEMA se propone una metodología que integra las soluciones administrativas, toma como base ISO/IEC 12207 y ayuda a las organizaciones a dirigir el proceso con adecuados niveles de calidad. Además define dos tipos de mantenimiento: mantenimiento planificable que incluye el mantenimiento correctivo no urgente, perfectivo preventivo y adaptativo y el mantenimiento no planificable que incluye mantenimiento correctivo urgente.

2.2.3.3. Metodología para el apoyo de mantenimiento basado en el estándar ISO/IEC 12207

Polo et. al. [19] presentan una metodología para el proceso del ciclo de vida del mantenimiento, que es una adaptación del estándar ISO/IEC 12207. ISO 12207 es un framework de referencia que cubre todos los aspectos del ciclo de vida del software, sin embargo no detalla cómo implementar las tareas y actividades incluidas en el proceso. MANTEMA incorpora una serie de actividades bien organizadas de acuerdo al tipo de mantenimiento, apoyando el control de documentación, se definen 5 tipos de mantenimiento: correctivo urgente, correctivo no urgente, perfectivo, preventivo y adaptativo. Además de esto, se considera la participación de 3 tipos de organizaciones en

el proceso de mantenimiento: cliente que es el propietario del software y requiere del servicio de mantenimiento; mantenedor es la organización que cumple con el servicio de mantenimiento; usuario que utiliza el software mantenido. De igual forma MANTEMA también proporciona plantillas estándar para cada uno de los documentos generados.

2.2.3.4. Metodología de mantenimiento de software para pequeñas organizaciones

Pino et. al. [4] presentan Agile MANTEMA como una propuesta metodológica para el mantenimiento de software que se centra en las pequeñas organizaciones. Especifica una estrategia de mantenimiento que define qué se necesita hacer, cuando, cómo y por quién. También establece una serie de elementos como los tipos de mantenimiento, niveles de servicio y niveles de capacidad, con el fin de manejar la complejidad que es inherente al proceso de mantenimiento y permitir a las pequeñas compañías definir su propio proceso de mantenimiento, tomando en cuenta sus características y necesidades particulares. El objetivo de Agile MANTEMA es hacer los elementos de proceso descritos en MANTEMA más livianos e incorporarlos en la gestión de proyectos agiles usando el método Scrum. Esta metodología describe un proceso, niveles de servicio, niveles de desempeño de proceso y niveles de capacidad de proceso.

Los estudios [4, 17-19] tienen como tema principal la etapa de mantenimiento de software, proponen modelos y metodologías de mantenimiento de software con el fin de controlar y mejorar este proceso, ya que esta etapa es la más costosa y conflictiva del ciclo de vida del software. Sin embargo, estas propuestas no tienen en cuenta la característica de mantenibilidad de software ni plantean soluciones para incorporarla al producto. Los estudios anteriores se centran únicamente en la etapa de mantenimiento del software al final del ciclo de vida del mismo, planteando una serie de actividades organizadas que permiten mejorar esta etapa con el fin de reducir el problema de altos costos que se presenta durante la misma.

2.2.4. Estudios relacionados con atributos de mantenibilidad.

2.2.4.1. Factores claves que afectan la mantenibilidad de software

Kumar [7] afirma que la mantenibilidad de un producto software es afectada por principios de diseño y arquitectura, se describen varios factores propuestos por diferentes investigadores en modelos de mantenibilidad de software, los cuales son de gran importancia en las evaluaciones de mantenibilidad. Entre estos factores se destacan: la estabilidad, facilidad de cambio, facilidad de análisis y facilidad de prueba, establecidos por la norma ISO 9126, y además de estos también se describen otros factores, planteados por diferentes autores, como: modularidad, documentación, facilidad de lectura, consistencia, simplicidad, capacidad de expansión, instrumentación, estandarización, lenguaje de programación, nivel de validación y pruebas, complejidad, trazabilidad, algunas propiedades estructurales o de código y habilidades de equipo de mantenimiento. Los factores mencionados en este estudio son de vital importancia para esta investigación, ya que permiten conocer los aspectos primordiales que debe tener un producto software para lograr un alto nivel de mantenibilidad.

2.2.4.2. Impacto de atributos internos del producto software sobre la mantenibilidad

Kozlov et. al. [20] evalúan el impacto que tienen algunos atributos internos del producto software sobre la característica de mantenibilidad, obteniendo como resultado que el número de líneas de comentario y número de módulos influyen sobre la misma y además se encuentran otros factores que influyen negativamente sobre esta, tales como: número de líneas de código, tipos de datos globales y locales, número total de ítems de entrada y salida, y complejidad de interfaz de los métodos de clase. El conocimiento acerca de la influencia que tienen estos atributos de calidad internos sobre la mantenibilidad puede ser utilizado a favor durante el proceso de desarrollo. Estos resultados sirven como base para mejorar la mantenibilidad de software en etapas tempranas del proceso de desarrollo de software, lo cual presenta un gran aporte para esta investigación.

2.2.4.3. Modelo de atributos de mantenibilidad de software

Hashim y Key [21] proponen un modelo que resalta la necesidad de mejorar la calidad del producto software de tal forma que el mantenimiento sea eficiente. En el artículo se destacan problemas asociados con el mantenimiento los cuales se relacionan con las deficiencias de la forma en la que los sistemas son desarrollados, tales como: falta de trazabilidad, falta de documentación, mal diseño e implementación, herramientas de desarrollo y técnicas y lenguajes de programación inapropiados. Además se presentan algunos atributos de mantenibilidad, que coinciden con los expuestos en [7]. El principal aporte de este estudio es que expone los problemas más importantes asociados al mantenimiento, los cuales deben ser tenidos en cuenta en esta investigación para buscar soluciones y así aumentar la facilidad de mantenimiento del producto software.

2.2.4.4. Efecto de los patrones de diseño en la mantenibilidad de software

Hegedus et. al. [22] realizan una investigación para determinar cuál es el efecto que tienen los patrones de diseño en la mantenibilidad de software, encontrando que todas las características ISO 9126 mejoran su calidad con el número de instancias de patrones, a partir de lo cual se concluye que los patrones de diseño tienen un impacto positivo en la mantenibilidad de software. El resultado obtenido es importante para esta investigación ya que se evidencia que es posible mejorar la mantenibilidad del producto software desde etapas tempranas del desarrollo de software, en este caso, haciendo uso de los patrones de diseño.

2.2.4.5. Arquitectura y mantenibilidad de software

Bosch y Bengtsson [23] proponen una técnica para analizar la mantenibilidad optima de una arquitectura software basada en un escenario específico. La arquitectura de software define las decisiones de software fundamentales y la descomposición del sistema en sus componentes principales y la relación entre estos componentes. Los autores resaltan la importancia de la arquitectura de software para cumplir con requerimientos de calidad ya que impacta y restringe a los atributos de calidad del sistema, tales como: el rendimiento y la mantenibilidad.

En los estudios [7, 20-23] se proponen atributos de mantenibilidad de software como: modularidad, documentación, facilidad de lectura, consistencia, simplicidad, capacidad de expansión, instrumentación, estandarización, lenguaje de programación, nivel de

validación y pruebas, complejidad, trazabilidad, algunas propiedades estructurales o de código, habilidades de equipo de mantenimiento, número de líneas de comentario, número de líneas de código, tipos de datos globales y locales, número total de ítems de entrada y salida, complejidad de interfaz de los métodos de clase y arquitectura de software. Además, se mencionan algunos problemas que afectan a la mantenibilidad del producto software como: falta de trazabilidad, falta de documentación, mal diseño e implementación, técnicas, herramientas de desarrollo y lenguajes de programación inapropiados. Todo lo anterior es muy importante y se debe tener en cuenta durante el proceso de desarrollo de software. Sin embargo, en los estudios mencionados, no se presentan propuestas que permitan incluir estos atributos al producto software durante el proceso de desarrollo, o que solucionen los problemas expuestos. Las anteriores son las principales diferencias que tienen los estudios revisados con ésta investigación.

2.2.5. Estudios relacionados con métricas de mantenibilidad

2.2.5.1. Modelo práctico para medir la mantenibilidad

Heitlager et. al. [24] realizan un estudio para la comprensión de los requerimientos mínimos que debe cumplir un modelo de mantenibilidad práctico basado en análisis de código fuente, esto son: las medidas deben ser independientes de la tecnología, tener una definición sencilla, ser simples de entender y explicar y deben permitir el análisis de causa raíz dando pistas claras de la relación entre las propiedades del nivel de código y nivel de calidad del sistema.

Con estos requerimientos, se ha formulado un modelo de mantenibilidad alternativo, este enlaza las características de mantenibilidad ISO 9126 con las medidas a nivel de código, la influencia de varias propiedades de código en características de mantenibilidad en un sistema software definidas son: el volumen tiene influencia sobre la facilidad de análisis del sistema, la complejidad por unidad tiene influencia sobre la facilidad de cambio y la facilidad de prueba, la duplicación influye sobre la facilidad de análisis y la facilidad de cambio y la unidad de prueba influye sobre la facilidad de análisis, la estabilidad y la facilidad de prueba del sistema. Posteriormente para cada propiedad en el código fuente se definen una o más medidas, para cada una de las propiedades anteriores se definen las siguientes métricas: para volumen líneas de código, años hombre en el fallo de los puntos de función y otras medidas de volumen como tamaño funcional, para complejidad por unidad complejidad ciclomática por unidad y otras medidas de complejidad como acoplamiento, fan-in, fan-out y medidas de estabilidad derivadas de estas, para duplicación bloques duplicados de más de 6 líneas, para tamaño por unidad líneas de código por unidad y para unidad de prueba cubrimiento de la unidad de prueba y número de sentencias assert.

El modelo propuesto en este estudio evita los problemas que se han encontrado en el índice de mantenibilidad (MI) como: análisis de la causa raíz, es decir, que el valor computado por el índice de mantenibilidad no proporciona las pistas necesarias para saber qué características de mantenibilidad han contribuido a la obtención de dicho valor ni que acción tomar para mejorarlo. Otros problemas son: la complejidad del promedio, la computabilidad, los comentarios, el entendimiento y el control.

2.2.5.2. Métricas de acoplamiento para predecir la mantenibilidad en diseños orientados a servicios

Perepletchikov et. al. [25] proponen una serie de métricas para cuantificar el acoplamiento estructural de los artefactos de diseño en sistemas orientados a servicios, estas métricas pretenden predecir las características de calidad de mantenibilidad del software orientado a servicios, pueden ser utilizadas como indicadores tempranos de características de software orientado a servicios, permitiendo a las organizaciones detectar problemas potenciales en las fases tempranas del ciclo de vida de desarrollo de software.

Existe una gran cantidad de métricas propuestas para medir varios aspectos de los atributos de calidad internos (estructurales), como acoplamiento, cohesión y complejidad, estos atributos no describen directamente la calidad de un producto, son usados para predecir atributos de calidad externos como por ejemplo la mantenibilidad. Aunque los atributos externos son los indicadores más importantes de calidad, solo pueden ser medidos eficientemente después de que el producto ha sido desarrollado, mientras que la medida de los atributos estructurales permite la predicción de la calidad de software en etapas tempranas del ciclo de vida del desarrollo. Las métricas definidas son aplicables ya sea a un sistema, a un servicio, a una interfaz de servicio, o un elemento base de implementación de servicio. El conjunto de métricas está conformado por dos subgrupos, el primero es llamado Métricas Primarias, el cual contiene nueve de ellas, y el segundo subgrupo es el de Métricas de Agregación conformado por ocho métricas.

2.2.5.3. Métricas de mantenibilidad para sistemas orientados a objetos

Dubey y Rana [26] realizan un estudio de la literatura relacionada con las métricas de orientación a objetos que pueden ser utilizadas para la evaluación de la mantenibilidad. Además, proponen un modelo de mantenibilidad basado en el análisis de la relación entre estas métricas y la mantenibilidad. Las métricas consideradas son: ponderado de métodos por clase, conjunto de respuesta por clase, falta de cohesión en los métodos, acoplamiento entre objetos, profundidad del árbol de herencia y profundidad del número de hijos por clase.

En los estudios [24-26] se proponen métricas de mantenibilidad de software. La diferencia que tienen estas métricas con las planteadas en la norma ISO 9126, es que las últimas no son medidas en el producto, sino en el desempeño de la actividad de mantenimiento por el equipo técnico. Por el contrario las propuestas en los estudios poseen un carácter predictivo y permiten medir el producto desde etapas tempranas del desarrollo para identificar los problemas que se presentan y corregirlos desde el inicio. Las métricas definidas en los estudios permiten identificar los aspectos que se tienen en cuenta en la evaluación de la mantenibilidad del software y al conocerlos, enfocarse en ellos durante el proceso de desarrollo para obtener un producto fácil de mantener. Aunque los estudios proponen los atributos que deben ser medidos en la mantenibilidad de software, no presentan propuestas que indiquen acciones de mejora cuando se obtienen bajos índices de mantenibilidad, ni como incluirlos al producto software durante su desarrollo.

2.2.6. Aportes

Este trabajo de investigación contribuye con los siguientes aportes:

- Determinar los atributos de software que influyen sobre las diferentes sub-características de la mantenibilidad del producto.
- Clasificar los atributos de software de acuerdo a las sub-características de mantenibilidad sobre las que influyen y el proceso de desarrollo en los que se presentan.
- Proponer un modelo de referencia que permita incluir al producto software las características de mantenibilidad a lo largo del proceso de desarrollo, para obtener un producto altamente mantenible.
- Con el modelo de referencia propuesto se podrían disminuir los costos de mantenimiento durante el proceso de desarrollo y además las empresas podrían utilizar este modelo con el fin de obtener la certificación en mantenibilidad realizado por AQC Lab.

CAPITULO III

MODELO DE REFERENCIA PARA LA INCLUSIÓN DE SUB-CARACTERÍSTICAS DE MANTENIBILIDAD AL PRODUCTO SOFTWARE

En este capitulo se presenta el desarrollo del ciclo de investigación metodologico realizado con el fin de obtener el modelo de referencia para la inclusión de sub-caracteristicas de mantenibilidad al producto software durante el proceso de desarrollo denominado MANTuS. Este ciclo contiene siete actividades principales divididas en tres grupos: fundamentos teóricos donde se definen los atributos de software considerados en el modelo de referencia propuesto, este grupo comprende las actividades: (i) Identificar los atributos de software que influyen en la mantenibilidad del producto, donde se obtienen los atributos de software referenciados en la literatura, los cuales son relevantes para esta investigación, (ii) Agrupar conceptos, donde se estrucutura un único concepto para cada uno de los atributos encontrados en los diferentes estudios revisados, (iii) Filtrar atributos, donde se realiza un analisis para identificar unicamente los atributos que se tienen en cuenta para el modelo de referencia; clasificación y análisis de atributos de mantenibilidad en el cual se identifican las sub-características y procesos en los que influye cada uno los atributos, este grupo abarca las actividades: (iv) Clasificar los atributos identificados, esto se realiza de acuerdo a las sub-características de mantenibilidad y a los procesos en los que se presentan, (v) Analizar como es posible potenciar cada sub-característica, en la cual se describe el analisis realizado para obtener las prácticas que podrían ayudar a potenciar cada una de las sub-caracteristicas de mantenibilidad teniendo en cuenta los atributos que la afectan; descripción de los procesos donde se especifica cada uno de los procesos del modelo de referencia en términos de su próposito, resultados y prácticas base, este grupo integra las actividades: (vi) Estructurar el modelo de referencia general, donde se definen los resultados del proceso, los cuales se asocian a las prácticas base de cada sub-característica, el modelo de referencia MANTuS cumple con los requisitos establecidos por la norma ISO/IEC 33004 para modelos de referencia de procesos, debido a que las acciones para alcanzar el consenso están fuera del alcance de este trabajo, este aspecto se encuentra documentado en el modelo como lo sugiere la norma, (vii) Plantear los modelos de referencia específicos, donde para cada una de las sub-características de mantenibilidad se exponen los procesos que permiten potenciarlas.

Ciclo de Investigación metodologico.

Para definir el modelo de referencia MANTuS se ha llevado a cabo el ciclo de investigación metodologico presentado en la estrategia de investigación definida en el *Capítulo I*. Durante este ciclo se realizaron las siguientes actividades:

- Identificar los atributos de software que influyen en la mantenibilidad del producto
- Agrupar conceptos
- Filtrar atributos
- Clasificar los atributos identificados.
- Analizar como es posible potenciar cada sub-característica
- Estructurar el modelo de referencia general
- Plantear los modelos de referencia específicos

El proceso seguido para obtener los atributos considerados en la construcción del modelo de referencia MANTuS se observa en la Figura 2. La descripción detallada de cada una de estas actividades se encuentra en la sección 3.1.

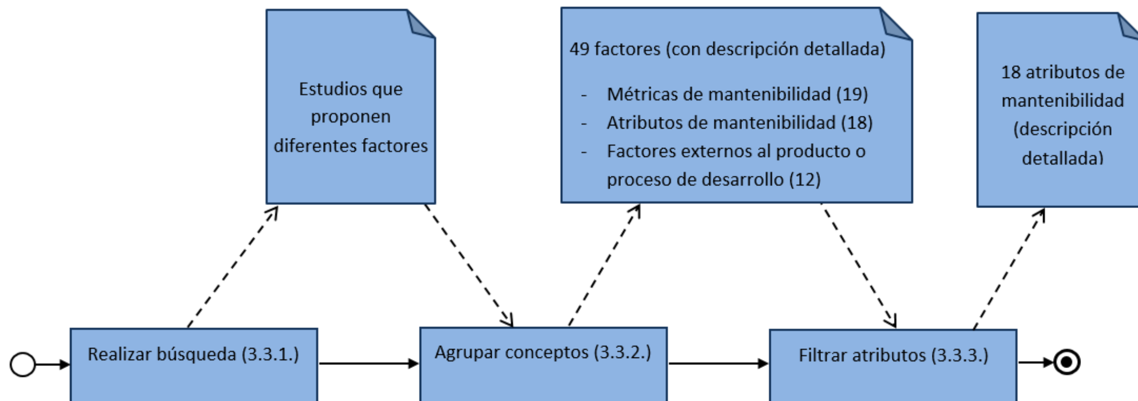

Figura 2. Proceso para la obtención de atributos

La asociación de los resultados obtenidos en cada uno de estas actividades constituyen el modelo de referencia propuesto. A continuación se presenta de manera detallada el desarrollo de cada una de estas actividades.

3.1. Fundamentos teóricos

3.1.1. Identificar los atributos de software que influyen en la mantenibilidad del producto.

Para identificar los atributos software que influyen en la mantenibilidad del producto, encontrados en los estudios relevantes mencionados en la sección 2.2.4, inicialmente se realizó una nueva búsqueda en la literatura, mediante la cadena de búsqueda: *"Software attribute" AND "maintainability"* la cual permitió obtener un total de 19 estudios que proponían diferentes factores relacionados con mantenibilidad. Los factores que afectan la mantenibilidad de software son aspectos vitales que juegan un papel importante en la evaluación de mantenibilidad, son conocidos también como factores de costo ya que son importantes indicadores del costo del mantenimiento de software [7] e incluyen propiedades del producto software, del mantenimiento, de los recursos, entre otros [27]. A continuación se presenta cada uno de los estudios encontrados con sus respectivos factores y definiciones dadas a cada uno de ellos:

En [28] se presentan:

- **Factor de atributos ocultos:** medida de los atributos visibles. Puede ser calculado sumando la visibilidad de cada atributo con respecto a otras clases del sistema.
- **Acoplamiento:** incrementa la comunicación entre clases, reduce el encapsulamiento e incrementa la complejidad del software.
- **Falta de cohesión:** la cohesión apoya el encapsulamiento y reduce la complejidad del software para ser entendido, implementado y probado.
- **Complejidad**
- **Densidad de control:** representa el porcentaje de sentencias de control en el código.
- **Factor de herencia de atributos:** porcentaje de atributos de clase que son heredados.

- **Factor de herencia de métodos:** se obtiene al dividir el número total de métodos heredados entre el número total de métodos.
- **Líneas de código fuente:** calcula el número de líneas de código fuente en el proyecto. Excluye las líneas con espacios en blanco y comentarios.
- **Longitud del programa:** número total de operadores y operandos en un programa.
- **Factor de métodos ocultos:** mide la visibilidad o invisibilidad de cada método con respecto a otras clases en el proyecto.
- **Porcentaje de miembros públicos y protegidos:** calcula el porcentaje de miembros públicos y protegidos de una clase con respecto a los otros miembros de la clase.
- **Número de clases:** calcula el número total de clases en un sistema.
- **Número de métodos:** calcula el número total de métodos en un sistema.
- **Factor de polimorfismo:** ayuda a medir el grado en el cual las clases en un sistema son polimórficas. El polimorfismo es utilizado en la programación orientada a objetos para realizar un ligado en tiempo de ejecución a una clase entre varias otras clases de la misma jerarquía. Ayuda a procesar instancias de clases, de acuerdo a su tipo de dato o clase.
- **Profundidad del árbol de herencia:** mide la posición de una clase en el árbol de herencia. Corresponde al número del nivel de una clase en la jerarquía de herencia.
- **Respuesta por clase:** es el número de métodos en una clase más el número de distintos métodos llamados por esos métodos.
- **Tamaño promedio de clase:** tamaño promedio de clase en número de líneas de código por clase.
- **Tamaño promedio de método:** tamaño promedio de método en términos de líneas de código por método.

En [29] se presentan los siguientes factores:

- **Arquitectura de tres capas:** una clara separación de preocupaciones entre la capa de presentación, negocio y persistencia es considerada una buena práctica. Cada capa debe mantenerse desacoplada de las capas por encima de ella, y depender solo de los componentes más generales en las capas inferiores.
- **Comentarios:** los comentarios deben ser significativos y de un tamaño efectivo, de tal forma que no influencien negativamente la facilidad de lectura del código.
- **Documentación**
- **Encapsulamiento:** dado que los métodos de Java sólo devuelven un objeto, los desarrolladores a menudo crean pequeñas clases de contenedores "salida" como una solución temporal. Esto introduce dependencias, tales como la creación de estructuras de objetos antes de llamar a un método, lo que puede conducir a problemas de mantenimiento.
- **Convenciones estándar de nombrado:** el uso de convenciones estándar de nombrado para paquetes, métodos y variables, facilita el entendimiento.
- **Herencia:** el uso de la herencia incrementa el número total de clases, de tal forma que debe ser usada con cuidado. Si una interface implementa varias clases, tiene el mismo efecto que herencias múltiples, lo cual lleva a confusión y baja mantenibilidad.
- **Librerías:** el uso de librerías propietarias puede significar baja mantenibilidad, porque los nuevos desarrolladores necesitarían familiarizarse con ellas.
- **Nombrado coherente:** los desarrolladores deben usar un esquema de nombrado consistente que permita al lector entender la relación entre métodos y clases. Las clases deben ser fáciles de identificar para facilitar el mapeo desde el dominio y requerimientos hacia el código.

- **Líneas de código**
- **Plataforma técnica apropiada:** está relacionado con requerimientos implícitos no documentados que emergen cuando un sistema se mueve a un entorno diferente.
- **Simplicidad:** el tamaño y complejidad de un sistema es crítico. Toma mayor tiempo identificar una clase cuando hay muchas clases. La presencia de muchas clases que están casi vacías es un signo de código que puede poseer baja mantenibilidad.
- **Uso de componentes:** la clases deben organizarse de acuerdo a la funcionalidad o de acuerdo a la capa de código en la que operan

En [25] se presentan:

- **Acoplamiento:** es una medida del grado al cual existe interdependencia entre módulos de software.
- **Cohesión:** es el grado al cual los elementos de un módulo contribuyen a una y solo una tarea.
- **Complejidad:** puede ser definida en términos del trabajo interno realizado por un módulo de software.

En [30] se presentan:

- **Acoplamiento:** los componentes con mayor acoplamiento se resisten más al cambio.
- **Complejidad:** los sistemas simples son más fáciles de comprender y de probar que los complejos.
- **Tamaño de unidad:** las unidades como piezas de funcionalidad de más bajo nivel deben mantenerse pequeñas para que sean centradas y fáciles de entender.
- **Redundancia:** el código duplicado debe ser mantenido en todos los lugares donde ocurre.
- **Tamaño de unidad de interfaz:** las unidades con muchos parámetros pueden ser síntoma de mal encapsulamiento.
- **Volumen:** entre más grande sea un sistema mayor será el esfuerzo requerido para mantenerlo, ya que es más información que se debe tener en cuenta.

En [31] se presentan los siguientes factores:

- **Acoplamiento del Sistema:** número de estructuras globales más el número de parámetros pasados, dividido entre el número total de estructuras de datos.
- **Anidación:** profundidad máxima de anidamiento del módulo, mide el porcentaje de módulos anidados.
- **Cohesión**
- **Comentarios Intramódulo:** número de líneas con comentarios, dividido entre el número total de líneas en el módulo, promediado sobre todos los módulos.
- **Complejidad**
- **Complejidad de entrada y salida:** número de líneas de código dedicadas a entrada/Salida, dividido entre el tamaño.
- **Consistencia:** falta de conflictos y contradicciones.
- **Documentación de apoyo**
- **Encapsulamiento**
- **Formato de sentencia:** porcentaje de sentencias no agrupadas por módulo, promediadas sobre todos los módulos.
- **Integridad de inicialización:** porcentaje de variables bien definidas (es decir variables declaradas antes de ser usadas) promediado sobre todos los módulos.

- **Nombrado:** número de identificadores y etiquetas con nombres significativos, dividido entre el número total de etiquetas e identificadores.
- **Tamaño:** sentencias de código no comentadas.
- **Tipos de datos globales:** número de tipos de datos globales, dividido entre el número total de tipos de datos definidos.
- **Tipos de datos locales:** número de tipos de datos locales, dividido entre el número total de tipos de datos definidos, promediado sobre todos los módulos.

En [27] se presentan los siguientes factores:

- **Complejidad Estructural**
- **Habilidad del equipo**
- **Mantenimiento de infraestructura:** por ejemplo, entorno de desarrollo, herramientas
- **Acoplamiento**
- **Complejidad**
- **Duplicación**
- **Tamaño del código**

En [32] se presentan:

- **Falta de cohesión**
- **Número promedio de líneas de comentario por módulo**
- **Número de comentarios**
- **Porcentaje de miembros privados**
- **Líneas de código**
- **Número de métodos**
- **Profundidad en el árbol de herencia**
- **Promedio de atributos por clase**
- **Complejidad ciclomática**
- **Porcentaje de número de métodos por clase**
- **Volumen**

En [7] se presentan los siguientes factores:

- **Capacidad de expansión:** describe que un cambio físico en la información, funciones computacionales, almacenamiento de datos o en tiempo de ejecución puede ser logrado fácilmente. La parametrización de constantes y el tamaño de estructuras de datos básicas generalmente mejora la capacidad de expansión.
- **Comentarios apropiados:** los comentarios deben ser expresivos y significativos, de tal forma que puedan mejorar la facilidad de lectura del código.
- **Consistencia:** indica que los productos de software asocian y contienen notación, terminología y simbología uniforme. El uso de convenciones de nombrado, procesos de Entrada/Salida, acoplamiento de módulos, que son reflejos de la consistencia.
- **Habilidades del equipo de mantenimiento**
- **Instrumentación:** la característica de instrumentación contiene ayudas que mejoran las pruebas de software.

En [20] se presentan los siguientes factores:

- **Proporción de comentarios**
- **Líneas de código**

- **Promedio de complejidad Ciclomática**
- **Tipos de datos globales:** número de tipos de datos globales dividido entre el número total de tipos de datos definidos.
- **Tipos de datos locales:** número de tipos de datos locales, dividido entre el número total de tipos de datos definidos promediado sobre todos los módulos.

En [33] se presentan los siguientes factores:

- **Arquitectura:** una clara distinción entre capas de presentación, negocio y persistencia.
- **Componentes:** las clases deben ser organizadas de acuerdo a su funcionalidad o a la capa de código en la que operan.
- **Encapsulamiento:**
- **Herencia:** el uso de la herencia aumenta el número de clases, lo cual disminuye la mantenibilidad, por lo cual debe ser utilizada con cuidado
- **Librerías:** el uso de librerías puede significar una gran cantidad de código, lo cual hace más difícil el mantenimiento.
- **Simplicidad:** el código no debe incluir sentencias que sean muy similares a otras.
- **Nombrado:** usar nombres que creen un sistema consistente que permita al lector entender la relación entre los métodos y las clases.
- **Comentarios:** algunos comentarios generales sobre las clases. La mayoría de los métodos deberían tener buenos comentarios.
- **Plataforma Técnica:** el uso de una plataforma estándar que sea ampliamente usada será bien conocida por los desarrolladores y además será soportada por otras compañías.

En [34] se presentan los siguientes factores:

- **Complejidad Ciclomática:**
- **Líneas de código:**
- **Número de comentarios:**
- **Complejidad lógica:** cuántas rutas de ejecución se encuentran en el código
- **Tamaño:** cuánto código está allí.

En [35] se presentan:

- **Complejidad ciclomática**
- **Líneas de código.**

En [36] se presentan:

- **Líneas de código:**
- **Complejidad ciclomática**

En [37] se presentan los siguientes factores:

- **Proporción de Comentarios:**
- **Número promedio de variables vivas:** una variable viva está viva en una declaración en particular sólo si se hace referencia a un cierto número de declaraciones antes o después de esa declaración.
- **Promedio Complejidad ciclomatica:** se define como promedio de complejidad ciclomática de todos los módulos.

- **Promedio de lapso entre variables vivas:** el lapso es el número de declaraciones entre dos referencias sucesivas del alguna variable

En [38] se presentan los siguientes factores:

- **Documentación**
- **Facilidad de lectura**
- **Facilidad de entendimiento:** comprensibilidad del software intenta estimar la correlación entre el estilo de la escritura de código fuente y la documentación correspondiente.

En [39] se presentan los siguientes factores:

- **Patrones de diseño:** intentan empaquetar soluciones probadas a problemas de diseño en una forma que hace que sea posible encontrar, adaptar y reutilizarlos.

En [24] se presentan los siguientes factores:

- **Comentarios:** es simplemente un código que se ha comentado, e incluso se trata de texto en lenguaje natural que a veces se refiere a las versiones anteriores del código.
- **Complejidad por unidad:** se refiere al grado de complejidad interna de las unidades de código fuente de la que es compuesta.
- **Duplicación:** es el grado de duplicación del código fuente. La excesiva duplicación es perjudicial para la mantenibilidad ya que hace que un sistema sea más largo de lo que necesita ser.
- **Líneas de código por unidad:**
- **Complejidad ciclomática por unidad:** dado que la unidad es la pieza más pequeña de un sistema que pueda ser ejecutado y probado de forma individual, tiene sentido calcular la complejidad ciclomática en cada unidad.
- **Tamaño de unidad:** el tamaño de las unidades a partir del cual un sistema está compuesto
- **Unidad de prueba:** pequeños programas escrito por los programadores para probar automáticamente su código una unidad a la vez.
- **Volumen:** El volumen global del código fuente influye en la analizabilidad del sistema.

En [21] se presentan los siguientes factores:

- **Complejidad:** supone reflejar a cierto grado, la dificultad en comprender o mantener códigos.
- **Documentación**
- **Estandarización:** un conjunto de estándares de programación deben estar disponibles como guía en la escritura de código para evitar la idiosincrasia entre programadores.
- **Facilidad de lectura:** grado en el cual un lector puede entender fácil y rápidamente un código fuente.
- **Herramientas de desarrollo y técnicas.**
- **Lenguaje de programación:** un código escrito en un lenguaje con el que los programadores de una organización no están familiarizados resulta en la dificultad de mantener el código reutilizado. El lenguaje utilizado también puede afectar la facilidad de lectura del programador.
- **Nivel de validación y pruebas:** un mayor tiempo gastado en el diseño de validaciones y programas de prueba da como resultado menos errores en el programa y en

consecuencia disminuye los costos de mantenimiento resultantes de la corrección de errores.
- **Trazabilidad:** habilidad para trazar una representación de diseño o componentes del programa actual, hacia los requerimientos.

En [22] se presentan los siguientes factores:

- **Patrones de diseño:** la proporción de líneas de código que forman parte de un patrón de diseño tiene una alta correlación con la mantenibilidad. Representan soluciones conocidas para problemas comunes de diseño en un contexto dado.

3.1.2. Agrupar conceptos

En los estudios anteriores, se halló que algunos de ellos presentaban factores de mantenibilidad similares, aunque no coincidían totalmente en el nombre o definición de los mismos, por lo que fue necesario estructurar un único concepto para cada factor. Para la unificación de los conceptos se analizaron las definiciones de cada factor expuestas en los artículos, realizando una comparación entre ellos, de tal forma que las definiciones que coincidían no fueron modificadas y en los casos donde se presentaban diferencias se unieron los conceptos, logrando así agruparlos en un único concepto. En los casos donde el nombre del factor no coincidía, pero su definición era similar se tuvo en cuenta el nombre que era utilizado un mayor número de veces en los estudios, al final de la definición de estos factores se indican los nombres de los conceptos agrupados por la misma. Como resultado de este análisis se obtuvieron 49 factores con su respectiva definición, los cuales se presentan a continuación:

- **Acoplamiento [25, 27-31]:** es una medida del grado de interdependencias que existe entre módulos de software. Los componentes bien acoplados son más resistentes al cambio. Agrupa el concepto: acoplamiento del sistema [31].
- **Anidación [31]:** profundidad máxima de anidamiento del módulo, mide el porcentaje de módulos anidados.
- **Arquitectura [29, 32, 33] :** una clara distinción entre la presentación, capa de negocio y la persistencia se considera como una buena práctica. Cada capa debe ser independiente de las capas por encima de él y depender sólo de las capas inferiores. El estándar IEE 1471-2000 [40] define la arquitectura como la organización fundamental de un sistema, expresada en sus componentes, las relaciones entre ellos y con su entorno, y los principios que gobiernan su diseño y evolución. Agrupa el concepto: arquitectura de tres capas [29].
- **Capacidad de expansión [7]:** un cambio físico a la información, funciones computacionales, o almacenamiento de datos puede ser realizado fácilmente.
- **Cohesión [25, 28, 29, 31, 32]:** grado de comunicación entre los métodos y variables miembros de una clase. Agrupa el concepto: falta de cohesión [28, 32].
- **Comentarios [7, 20, 24, 29, 31-34, 37] :** relación existente entre el número total de líneas de código y el número de líneas de comentarios. Agrupa los conceptos: Comentarios intramódulo [31], número de comentarios [32, 34], número promedio de líneas de comentarios por módulo [32], comentarios apropiados [7], proporción de comentarios [20, 32, 37].
- **Complejidad [21, 24, 25, 27-31, 34]:** dificultad en la comprensión o el mantenimiento de los códigos. La medición de la complejidad de un módulo implica medir el flujo de control del módulo, el flujo de datos e incluso las estructuras de datos utilizadas.

Agrupa los conceptos: complejidad estructural [27], complejidad lógica [34], complejidad por unidad [24].

- **Complejidad Ciclomática [20, 24, 32, 34-37]:** promedio de complejidad ciclomática de todos los módulos. Es el número de caminos lógicos individuales contenidos en un programa. Agrupa los conceptos: promedio de complejidad ciclomática [20, 37], complejidad ciclomática por unidad [24].
- **Complejidad de entrada y salida [31]:** número total de líneas de código dedicadas a entrada y salida, dividido entre el número de líneas de código.
- **Consistencia [7, 31]:** indica que el producto software asocia y contiene simbología, terminología y notación uniforme.
- **Densidad de control [28]:** porcentaje de sentencias de control en el código.
- **Documentación [21, 29, 31, 38]:** contiene explicaciones detalladas del diseño de software.
- **Duplicación [24, 27, 30]:** es el grado en que una secuencia de código fuente ocurre más de una vez. La excesiva duplicación es perjudicial para la mantenibilidad ya que hace que un sistema sea más largo de lo que necesita ser. Agrupa el concepto: redundancia [30].
- **Encapsulamiento [29-31, 33]:** hacer privadas las variables que son innecesarias para el tratamiento del objeto pero necesarias para su funcionamiento, así como las funciones que no necesitan interacción del usuario o que solo pueden ser llamadas por otras funciones dentro del objeto. Agrupa el concepto: tamaño de unidad de interfaz [30].
- **Estandarización [21, 29]:** un conjunto de estándares de programación (paquetes, clases, métodos y variables) debe estar disponible para actuar como una guía en la escritura de código y evitar la idiosincrasia entre los programadores. Agrupa el concepto: convenciones estándar de nombrado [29].
- **Facilidad de lectura [21, 38]:** grado al cual un lector pueden entender fácil y rápidamente el código fuente.
- **Facilidad de entendimiento [38]:** intenta estimar la correlación entre el estilo de la escritura de código fuente y la documentación correspondiente.
- **Factor de atributos ocultos [28]:** medida de los atributos visibles. Puede ser calculado sumando la visibilidad de cada atributo con respecto a otras clases del sistema.
- **Factor de métodos ocultos [28]:** ayuda a medir la visibilidad o invisibilidad de cada método con respecto a otras clases en el proyecto.
- **Formato de sentencia [31]:** porcentaje de sentencias sin agrupar por módulo, promediado sobre todos los módulos.
- **Habilidad del equipo de mantenimiento [7, 27]:** para llevar a cabo las tareas de mantenimiento, deben entender cómo el software se lleva a cabo antes de que puedan modificarlo. Las actividades incluyen llevar a cabo el cambio, la documentación, pruebas y presentación de informes. Agrupa el concepto: habilidad del equipo [27].
- **Herencia [28, 29, 33]:** se pueden crear nuevas clases partiendo de una clase o de una jerarquía de clases preexistente. Agrupa el concepto: factor de herencia de atributos, métodos [28].
- **Herramientas de desarrollo y técnicas [21, 27]:** agrupa el concepto: mantenimiento de infraestructura [27].
- **Instrumentación [7]:** contiene ayudas que mejoran las pruebas de software.
- **Integridad de inicialización [31]**: porcentaje de variables bien definidas

- **Lenguaje de programación [21] :** un código escrito en un lenguaje con el que el programador no está familiarizado genera dificultades en mantener el código reutilizado.
- **Librerías propietarias [29, 33]:** el uso de librerías propietarias y el estilo de codificación de los desarrolladores de estas librerías pueden causar una baja mantenibilidad. Agrupa el concepto: librerías [29, 33].
- **Líneas de código [20, 22, 24, 28, 29, 31, 32, 34-36]:** número total de líneas de código lógicas en el sistema, excluyendo los espacios en blanco y los comentarios. Agrupa los conceptos: líneas de código fuente [28], líneas de código por unidad [24].
- **Nivel de validación y pruebas [21]:** un mayor tiempo gastado en el diseño de validaciones y programas de prueba da como resultado menos errores en el programa y en consecuencia disminuye los costos de mantenimiento resultantes de la corrección de errores.
- **Nombrado Coherente [29, 31, 33]:** el uso de convenciones estándar para el nombrado de paquetes, clases, métodos y variables facilita el entendimiento. Hacer uso de un esquema de nombrado coherente permite al lector entender la relación entre métodos y clases. Las clases deben ser identificadas fácilmente para que puedan se mapeadas del dominio y requerimientos hacia el código. Agrupa el concepto: nombrado [31, 33].
- **Numero de clases [22, 28]:** calcula en número total de clases en el sistema.
- **Número de métodos [28, 32]:** número total de métodos en el sistema. Agrupa el concepto: porcentaje de números de métodos por clase [32].
- **Número promedio de variables vivas [37]**: se dice que una variable está viva en una sentencia si ésta es referenciada frecuentemente por sentencias antes y después de dicha sentencia. El promedio de variables vivas es la suma del total de variables vivas divididas entre el total de sentencias ejecutables.
- **Patrones de diseño [22, 39]:** buscan agrupar soluciones a problemas de diseño, de tal forma que sea posible encontrarlos, adaptarlos y reutilizarlos. La proporción de líneas de código que forman parte de un patrón de diseño tiene una alta correlación con la mantenibilidad.
- **Plataforma Técnica [29, 33]:** una parte importante del mantenimiento de los sistemas es la habilidad de adaptarse a diferentes entornos y muchos problemas con el mantenimiento de los sistemas están relacionados con requisitos implícitos no documentados que aparecen cuando un sistema es trasladado a un entorno diferente. El uso de una plataforma estándar de herramientas simplifica esta situación. Una plataforma estándar es bien conocida por los desarrolladores y será soportada por diferentes compañías. Agrupa el concepto: plataforma técnica apropiada [29].
- **Polimorfismo [28]:** es utilizado en la programación orientada a objetos para realizar un ligado en tiempo de ejecución a una clase entre varias otras clases de la misma jerarquía. Ayuda a procesar instancias de clases, de acuerdo a su tipo de dato o clase. Agrupa el concepto: factor de polimorfismo [28].
- **Porcentaje de miembros públicos y privados [28, 32]:** es el porcentaje de miembros públicos y protegidos de una clase con respecto a los otros miembros de la misma. Agrupa el concepto: porcentaje de miembros privados [32].
- **Profundidad de árbol de herencia [28, 32]:** posición de una clase en el árbol de herencia. Corresponde al número de nivel de una clase en la jerarquía de herencia.
- **Promedio de atributos por clase [32]:** número de atributos en una clase.
- **Promedio de métodos por clase [32]:** número de métodos por cada clase de un sistema.

- **Respuestas por clases [28]:** es el número de métodos en una clase más el número de distintos métodos llamados por esos métodos.
- **Simplicidad [29, 33]:** relacionado con el tamaño y la complejidad de un sistema. Toma más tiempo identificar una clase específica cuando hay muchas clases. La presencia de muchas clases casi vacías es una señal de código poco mantenible.
- **Tamaño [24, 27, 28, 30-32, 34]:** cantidad de código del sistema. Las unidades de funcionalidad de más bajo nivel deben mantenerse pequeñas para que sean centradas y fáciles de mantener. Agrupa el concepto: longitud de programa [28], tamaño de código [27], tamaño de unidad [24, 30], volumen [24, 30, 32].
- **Tamaño promedio de clase [28]:** tamaño promedio de clase en número de líneas de código por clase.
- **Tamaño promedio de método [28]:** tamaño promedio de método en términos de líneas de código por método.
- **Tipos de datos globales y locales [20, 31] :** número total de tipos de datos globales entre el número total de datos definidos, y número total de tipos de datos definidos entre el número promedio de tipos de datos definidos en todos los módulos. Agrupa los conceptos: tipos de datos globales [20, 31], tipos de datos locales [20, 31].
- **Trazabilidad [21]:** habilidad para trazar la representación de diseño o los componentes actuales del programa hacia los requerimientos.
- **Unidad de prueba [24]:** pequeños programas escrito por los programadores para probar automáticamente su código una unidad a la vez.
- **Uso de componentes [29, 33]:** las clases deben ser organizadas de acuerdo a la funcionalidad o a la capa de código en la que operan. Agrupa el concepto: componentes [33].

3.1.3. Filtrar atributos

Después de realizar un análisis de la definición de estos factores, se observó que algunos de estos no son atributos del producto, por lo cual no son relevantes para esta investigación y no se tuvieron en cuenta. A continuación se nombran dichos factores:

- Arquitectura
- Instrumentación
- Habilidades del equipo de mantenimiento
- Herramientas de desarrollo y técnicas
- Lenguajes de programación
- Librerías propietarias
- Nivel de validación y pruebas.
- Nombrado coherente
- Patrones de diseño
- Plataforma técnica
- Uso de componentes
- Unidad de prueba

Después de este primer filtro quedaron un total de 37 factores que influyen sobre la mantenibilidad, sin embargo, también se notó que 19 de estos factores correspondían a métricas y no a atributos del producto, por lo tanto, estas métricas se relacionaron a los atributos que miden. En la Tabla 1 se muestran la relación de los atributos con las métricas encontradas, basándose en lo hallado en la literatura y un análisis personal que

se apoya en la definición de las métricas y los atributos. A continuación se indica el resultado del análisis realizado.

- **Complejidad de entrada y salida:** como lo indica en nombre, ésta métrica se relaciona con el atributo de complejidad.
- **Densidad de control:** debido a que esta mide el porcentaje de sentencias de control en el código se considera que está relacionado con la complejidad, ya que al aumentar este porcentaje la complejidad también podría aumentar.
- **Formato de sentencia:** ya que este está métrica se refiere al porcentaje de sentencias no agrupadas, es decir, no más de una sentencia por línea, se considera que está relacionada con la facilidad de lectura, porque si se tienen muchas sentencias en una sola línea de código es posible que la facilidad de lectura disminuya.
- **Integridad de inicialización:** esta métrica mide el porcentaje de variables bien definidas, por lo cual se relaciona con la facilidad de lectura ya que cuando se declaran las variables antes de utilizarlas este atributo podría aumentar.
- **Promedio de atributos por clase:** entre más atributos tenga una clase la complejidad de la misma puede aumentar.
- **Variables vivas:** una variable está viva desde la primera a la última referencia dentro de un procedimiento, por lo tanto se considera que entre más variables vivas existan, la complejidad del código podría ser mayor.

Métricas	Atributos	Fuente
Complejidad ciclomática	Complejidad	[41-43]
Complejidad de entrada y salida	Complejidad	Propia
Densidad de control	Complejidad	Propia
Factor de atributos ocultos	Encapsulamiento	[43]
	Complejidad	[44]
Factor de métodos ocultos	Encapsulamiento	[43]
	Complejidad	[44]
Formato de sentencia	Facilidad de lectura	Propia
Integridad de inicialización	Facilidad de lectura	Propia
Líneas de código	Tamaño	[42, 43, 45]
	Complejidad	[41, 42]
Número de clases	Complejidad	[41]
Número de métodos	Complejidad	[41, 42]
	Tamaño	[46]
Número promedio de variables vivas	Complejidad	Propia
Porcentaje de miembros públicos y privados	Encapsulamiento	[28, 42]
Profundidad del árbol de herencia	Complejidad	[42, 45]
	Anidación	[42]
	Herencia	[42]
Promedio de atributos y métodos por clase	Complejidad	Propia
Respuestas por clases	Complejidad	[43]
	Acoplamiento	[41, 42]
Tamaño promedio de clase	Complejidad	[28]
	Facilidad de entendimiento	[28]

Tamaño promedio de método	Complejidad	[28]
	Facilidad de entendimiento	[28]
Tipo de datos globales y locales	Facilidad de lectura	[20]

Tabla 1. Relación entre métricas y atributos.

Por lo anterior, se obtuvieron en total 18 atributos software que se consideraron en la construcción del modelo de referencia. Estos atributos se presentan en la Tabla 2.

Atributos	Fuentes
Acoplamiento	[25, 27-31]
Anidación	[31]
Capacidad de expansión	[7]
Cohesión	[25, 28, 29, 31, 32]
Comentarios	[7, 20, 24, 29, 31-34, 37]
Complejidad	[21, 24, 25, 27-31, 34]
Consistencia	[7, 31]
Documentación	[21, 29, 31, 38]
Duplicación	[24, 27, 30]
Encapsulamiento	[29-31, 33]
Estandarización	[21, 29]
Facilidad de lectura	[21, 38]
Facilidad de entendimiento	[38]
Herencia	[28, 29, 33]
Polimorfismo	[28]
Simplicidad	[29, 33]
Tamaño	[24, 27, 28, 30-32, 34]
Trazabilidad	[21]

Tabla 2. Atributos finales

3.2. Clasificación y análisis de atributos de mantenibilidad

3.2.1. Clasificar los atributos identificados

Los atributos identificados anteriormente se clasificaron de acuerdo a dos aspectos: sub-características de mantenibilidad sobre las que influye y las etapas del proceso de desarrollo en las que se presenta.

3.2.1.1. Clasificar los atributos por sub-caracteristicas y flujos de trabajo.

Para realizar la clasificación por sub-caracteristicas de mantenibilidad fue necesario hacer una comparación entre las sub-características de mantenibilidad presentadas por las normas ISO/IEC 9126-1 [47] e ISO/IEC 25010 [5].

La mantenibilidad de software presenta diferentes sub-características, de acuerdo a la norma ISO/IEC 9126-1 esta se divide en cinco sub-características: capacidad para ser analizado (Analysability), capacidad para ser cambiado (Changeability), estabilidad (Stability), capacidad para ser probado (Testability) y cumplimiento de la mantenibilidad

(Compliance). Por otra parte la norma ISO/IEC 25010 divide la mantenibilidad en cinco características: modularidad (Modularity), reusabilidad (Reusability), capacidad para ser analizado (Analysability), capacidad para ser modificado (Modifiability) y capacidad para ser probado (Testability).

Debido a que la norma ISO/IEC 25010 se deriva de la ISO/IEC 9126, al realizar la comparación entre las sub-características definidas por estas dos normas se encuentra que la capacidad para ser analizado y la capacidad para ser probado se presentan en las dos normas. Además la sub-característica capacidad para ser modificado presentada en ISO/IEC 25010 es la combinación de las sub-características capacidad para ser cambiado y estabilidad, presentadas en la norma ISO/IEC 9126-1. Por último la norma ISO /IEC 25010 incluye dos nuevas sub-características a la mantenibilidad: modularidad y reusabilidad. Es por esto que las sub-características que se tuvieron en cuenta para realizar la clasificación de los atributos son las definidas en la norma ISO/IEC 25010.

Con el fin de clasificar estos atributos en cada una de las sub-características de mantenibilidad se realizó un análisis semántico teniendo en cuenta las definiciones tanto de los atributos encontrados como de las sub-características definidas en la norma ISO/IEC 25010. Además, para clasificar estos atributos en la etapa de desarrollo en la que se presentan, se tuvieron en cuenta los flujos de trabajo definidos en Rational Unified Process (RUP): Modelo de Negocio, requisitos, análisis y diseño, implementación, pruebas y despliegue. Los resultados obtenidos se presentan en la Tabla 3.

Para tener un criterio de clasificación se buscaron estudios donde se evidencia la relación entre el atributo con la sub-característica de mantenibilidad y el flujo de trabajo al que ha sido asignado. La búsqueda de estos estudios se realizó de la siguiente manera: Primero se dividieron equitativamente los atributos a buscar entre los dos integrantes. Después, al analizar el conglomerado se intercambiaron los atributos que no fueron encontrados por el otro integrante. Finalmente, los dos integrantes buscaron todos los atributos que no fueron hallados anteriormente. A continuación, se presenta el resultado de todo el proceso de búsqueda.

- **Acoplamiento:** En [48] se afirma que el acoplamiento en los sistemas software tiene un fuerte impacto negativo en la calidad de software y por lo tanto se debe mantener al mínimo durante la etapa de diseño. En [49] el autor se refiere al acoplamiento como propiedad de diseño. En [50] el autor se refiere al acoplamiento como un concepto integral de diseño. En [51] se indica que el acoplamiento ha sido identificado como uno de las propiedades básicas de la calidad del diseño del software. En [52] se afirma que el acoplamiento es un factor de calidad muy importante para el diseño y la implementación orientada a objetos. En [44] se evidencia que el acoplamiento debe ser considerado durante el diseño, porque se afirma que es una de las características importantes que brinda eficiencia al diseño en la orientación a objetos. En este estudio también se indica que al aumentar el acoplamiento también aumenta la complejidad del diseño, lo que hace que se necesite mayor esfuerzo para realizar las pruebas reduciendo así la capacidad para ser probado del diseño, lo cual demuestra que hay relación entre el acoplamiento y esta sub-característica. En [41] se demuestra que existe relación entre el acoplamiento con la capacidad para ser analizado y la reusabilidad del software, porque se dice que un bajo acoplamiento mejora estas dos sub-características. En [53] se afirma que el acoplamiento excesivo entre las clases de un sistema afecta la modularidad del mismo y también que la medida del acoplamiento es un buen indicador de la capacidad para ser probado. En [54] se realiza un estudio

donde se concluye que la métrica de acoplamiento entre objetos presenta correlación con la sub-característica de mantenibilidad capacidad para ser modificado. En [55, 56] se muestra que el acoplamiento se relaciona con la capacidad para ser modificado, la capacidad para ser probado y la reusabilidad del sistema, ya que se dice que cuando el acoplamiento aumenta la reusabilidad disminuye y se hace más difícil modificar y probar el sistema.

- **Anidación:** en [57] se dice que la anidación es un factor del código fuente y que un exceso en la anidación de un componente reduce la facilidad de lectura y la capacidad para ser probado. En [42] se afirma que la anidación está relacionada con la sub-característica de mantenibilidad de software capacidad para ser analizado ya que una excesiva anidación de clases lleva a que el código fuente sea difícil de entender. En [58] se considera importante tener en cuenta la profundidad de la anidación para evaluar la escalabilidad en la fase de diseño, debido a que la sub-característica capacidad para ser modificado incluye la escalabilidad, entonces este estudio justifica la relación entre la anidación y dicha sub-característica de mantenibilidad.

- **Capacidad de expansión:** en [59] se afirma que para simplificar el proceso de modificación del producto software después de su entrega con el fin de corregir fallas o mejorar algunos factores de desempeño, es decir, para que el sistema presente la sub-característica capacidad para ser modificado, se requieren algunas características, entre las que se encuentra la capacidad de expansión.

- **Cohesión:** en [44] se afirma que la cohesión es una característica importante que ayuda a la eficiencia del diseño. En [49] el autor se refiere a la cohesión como propiedad de diseño. En [60] el autor habla de la cohesión como un atributo de diseño más que de código y un atributo que puede ser usado para predecir las propiedades de implementación como "facilidad de depuración, facilidad de mantenimiento y facilidad de modificación". En [61] se dice que la cohesión es una de las propiedades de diseño más importantes. En [50] se argumenta que en el diseño orientado a objetos la cohesión en un gran beneficio, además el autor se refiere a la cohesión como un concepto integral de diseño. En [51] se indica que la cohesión ha sido identificado como uno de las propiedades básicas de la calidad del diseño del software. En [52] se afirma que la cohesión es un factor de calidad muy importante para el diseño y la implementación orientada a objetos. En [41] se afirma que una alta cohesión aumenta la capacidad para ser modificado y la modularidad del sistema. En el estudio [54] se concluye que la métrica falta de cohesión en métodos presenta correlación con la sub-característica de mantenibilidad capacidad para ser modificado. En [53, 62] se evidencia que la cohesión influye en la reusabilidad, ya que se dice que cuando hay alta cohesión los componentes tienden a tener alta mantenibilidad y reusabilidad. En [63] se afirma que cuanto menor es la cantidad de acoplamiento y la complejidad de los componente y más alto sea la cohesión más fácil será la capacidad para ser analizado, la estabilidad y la capacidad para ser probado del producto software. En [64] indican que se ha demostrado que las métricas de la cohesión son buenos predictores de la capacidad para ser probado, al encontrar una correlación clara entre ellas.

- **Comentarios:** en [3] se afirma que el uso de los comentarios es un asunto que afecta la calidad de la programación. En [42] consideran el uso constante de la información informal y la proporción de las líneas de código comentadas con el tamaño total del código fuente como factores que pueden contribuir a la capacidad para ser analizado,

capacidad para ser modificado y la reusabilidad. En [65] relacionan la capacidad para ser analizado con el porcentaje de comentarios del código.

- **Complejidad:** en [52] se afirma que la complejidad es un factor de calidad muy importante para el diseño y la implementación orientada a objetos. En [44, 66] se asegura que la complejidad influye en la capacidad para ser probado, se dice que cuando la complejidad es baja disminuye el esfuerzo necesario para probar el software. En [41] se señala que baja complejidad indica alta capacidad para ser analizado y alta capacidad para ser modificado, lo cual justifica que hay relación entre la complejidad y estas dos sub-características. En [42] identifican un número de métricas relacionadas con la reusabilidad. Las características de código fuente que relacionan estas métricas pertenecen a la alta modularidad, baja complejidad y buenas características de documentación.

- **Consistencia:** en [67] se afirma que la consistencia del sistema puede lograrse cubriendo la consistencia en cada fase de desarrollo de software y entre las diferentes fases de desarrollo de software.

- **Documentación:** en [68] se afirma que los documentos de diseño son una fuente de información importante, especialmente cuando los sistemas entran en la fase de mantenimiento. En [69] se dice que las metodologías típicas del desarrollo orientado a objetos requieren que se documente el análisis, el diseño arquitectural o a alto nivel y el diseño a bajo nivel. En [3] se dice que en la industria del software es necesario mantener una trazabilidad bidireccional durante todo el ciclo de vida de los sistemas software y también se afirma que es un aspecto importante para la facilidad de entendimiento y capacidad de ser modificado del software. En [41] se indica que la documentación contribuye a capacidad para ser analizado, la capacidad para ser modificado y la reusabilidad del software. En [46] se dice que la documentación es un factor importante para la sub-característica capacidad para ser probado ya que en las pruebas es importante tener documentación clara de los requerimientos y especificaciones.

- **Duplicación:** en [70] se expone que la redundancia es reconocida como un problema en el mantenimiento de software ya que incrementa el tamaño del programa y por lo tanto el esfuerzo de las actividades relacionadas con el tamaño como por ejemplo las inspecciones, es por esto que debe tenerse en cuenta durante la etapa de implementación. También se dice que es considerada un obstáculo para la capacidad para ser modificado del software, ya que un cambio que se debe realizar en una pieza de código debe realizarse también en sus duplicados. En [71] se establece que el código duplicado puede hacer que el sistema sea más grande, más complejo para analizar y más difícil para realizar cambios. Con el tiempo, la cantidad de duplicación en un sistema puede llegar a ser considerable y puede complicar el mantenimiento. En [72] se plantea que los programas a menudo tienen una gran cantidad de código duplicado, lo que hace más difícil la comprensión y el mantenimiento. Además plantean que los resultados de la duplicación son el aumento del tamaño y la complejidad del código, por lo que el mantenimiento del programa es más difícil. En [73] relacionan la duplicación con la capacidad para ser analizado y con la capacidad para ser modificado. En [74] consideran que la capacidad para ser analizado está afectada por el volumen, la duplicación, tamaño de la unidad y el acoplamiento. En [65] relacionan la duplicación con la capacidad para ser modificado.

- **Encapsulamiento:** en [44] se dice que debido a que cada clase encapsula ciertos atributos y métodos, hacer que esos atributos sean públicos o privados afecta el factor de encapsulamiento del diseño impactando la complejidad y la capacidad para ser probado del software. Este estudio también afirma que incrementar el encapsulamiento aumenta la capacidad para ser cambiado, esto justifica la relación entre encapsulamiento y la sub-característica capacidad para ser modificado, ya que ésta incluye a la sub-característica capacidad para ser cambiado. En [75] se indica que el encapsulamiento mejora la facilidad de entendimiento y la capacidad para ser analizado del software. En [41] se expone que alto encapsulamiento implica alta modularidad. En [42] se afirma que el encapsulamiento mejora la escalabilidad del sistema, y por lo tanto también su capacidad para ser modificado. En este estudio también se asegura que este atributo ayuda a mejorar la modularidad, ya que busca proteger los datos y funcionalidades específicas de un módulo de accesos no autorizados de otros módulos.

- **Facilidad de lectura:** en [76] se afirma que debido a que la facilidad de lectura puede afectar la calidad del software, los programadores deben preocuparse por ella. En [77] se dice que se debe inspeccionar la facilidad de lectura del código fuente para asegurar la mantenibilidad, portabilidad y reusabilidad del software. En [59] se enuncia que la facilidad de lectura afecta la capacidad para ser analizado del software, ya que simplifica el trabajo de identificar las modificaciones que se requieren hacer al sistema. En [78] se expone que la facilidad de lectura mejora la facilidad para ser entendido de un componente, lo cual mejora la capacidad para ser analizado y también la capacidad para ser cambiado del mismo.

- **Facilidad de entendimiento:** en [79] se dice que entre más facilidad de entendimiento tenga el código fuente es más fácil para el programador obtener información crítica sobre un programa y leer el código, de esta forma se ve la importancia que tiene considerar la facilidad de entendimiento en la etapa de implementación. En [45] se define a la facilidad de entendimiento como el nivel de dificultad para estudiar y entender el diseño y código del sistema, por lo tanto por definición se puede observar que la facilidad de entendimiento está relacionada con la etapa de implementación y la sub-característica de mantenibilidad capacidad para ser analizado. En este estudio también se afirma que este es un factor muy importante para fomentar la reusabilidad ya que la mayoría de las actividades de reusabilidad requieren que los ingenieros de software primero entiendan los componentes del software antes de realizar cambios o hacer las extensiones correspondientes. En [80] se afirma que la sub-característica capacidad para ser analizado puede ser alcanzada parcialmente a través de la facilidad de entendimiento.

- **Herencia:** en [50] el autor se refiere a la herencia como un concepto integral del diseño orientado a objetos. En [51] se indica que la herencia ha sido identificado como uno de las propiedades básicas de la calidad del diseño del software. En [81] se realiza un estudio para comprobar la influencia que tiene la herencia sobre la sub-característica de mantenibilidad capacidad para ser modificado, concluyendo que los sistemas sin herencia son más fáciles de modificar que los sistemas con herencia. En [44] se afirma que la herencia es una característica importante que ayuda a la eficiencia del diseño orientado a objetos. En este estudio se dice que la herencia incrementa la reusabilidad y también puede afectar negativamente la capacidad para ser probado del software, ya que entre más grande sea el árbol de profundidad mayor será el esfuerzo para realizar las pruebas. En [82] se afirma que la reusabilidad está limitada a mecanismos usuales

como la herencia y algunos patrones de diseño. En [66] se dice que la herencia ayuda en la reusabilidad, como también en otros factores como la complejidad y la capacidad para ser probado del software. En [53] se indica que la herencia permite una mejor reusabilidad del código.

- **Polimorfismo:** en [44] se evidencia que el polimorfismo debe ser considerado durante el diseño, porque se afirma que es una de las características importantes que brinda eficiencia al diseño en la orientación a objetos. En [42] se hace referencia al polimorfismo como una decisión de diseño que busca alcanzar la generalización y por lo tanto mejora la reusabilidad. También se afirma que facilita la capacidad para ser modificado del software.

- **Simplicidad:** en [83] se evidencia que la simplicidad debe ser considerada durante la etapa de pruebas, ya que se dice que entre más simple sea un componente resultará menos costoso probarlo. En [80] se dice que la propiedad simplicidad es importante para lograr la capacidad para ser analizado del software. En [83] se declara que el atributo de simplicidad está relacionado con la capacidad para ser probado. En [84] se afirma que la simplicidad está relacionada con la capacidad para ser probado y la reusabilidad del software.

- **Tamaño:** en [85] se evidencia que el tamaño de unidad debe ser considerado durante la implementación, ya que se afirma que es una propiedad del código fuente. En [86] se afirma que las líneas de código influyen en la capacidad para ser analizado del software, ya que afectan el tiempo necesario para diagnosticar errores. También se evidencia que las líneas de código afectan la capacidad para ser probado del software, ya que las pruebas necesitan cubrir todo el código y cuando el número de líneas de código incrementan, esta tarea se dificulta. En [87] se afirma que el número de líneas de código es la característica que más afecta la capacidad para ser cambiado del software, debido a que esta sub-característica está incluida en la capacidad para ser modificado, entonces lo anterior justifica la relación entre las líneas de código y la capacidad para ser modificado. En [53] se evidencia que el número de líneas de código afecta la reusabilidad del software, en el estudio se dice que una clase más grande es difícil de reutilizar, de mantener y de entender. En [24, 30, 85] se dice que el tamaño de unidad influencia la capacidad para ser analizado y la capacidad para ser probado de todo el sistema, ya que cuando se tienen grandes unidades se tiene menor capacidad para ser analizado y menor capacidad para ser probado.

- **Trazabilidad:** en [88, 89] se define la trazabilidad como una característica que se debe tener en cuenta en la especificación de los requisitos del software e indican que esta área permanece como un problema ampliamente reportada debido a que no hay un análisis de las fuentes de requerimientos utilizadas en los desarrollos de software. En [90]se afirma que en el proceso desarrollo de software, la trazabilidad y evolución de los requisitos es uno de los factores más relevantes para lograr software fiable y exacto. En [68] se señala que un diseño es considerado de mayor calidad siempre que se mantenga la trazabilidad con el código. Como el diseño representa una abstracción de la implementación, se espera que las clases en el diseño estén representadas en el código. En [91] se indica que los vacíos en la trazabilidad, como características de información ausentes reducen la capacidad para ser modificado y la variabilidad de los componentes. Además de esto en el estudio se encuentra que varias características como la capacidad para ser analizado dependen de la trazabilidad de artefactos para análisis de impacto o comprensión del programa.

La relación de algunos atributos con las sub-características de mantenibilidad no fueron encontradas en la literatura revisada, para estos casos se hizo un análisis por parte de los investigadores basándose en las definiciones tanto de los atributos como de las sub-características. A continuación se presenta el análisis realizado por atributo.

- **Anidación:** debido a que este atributo se ve afectado por la relación entre módulos, es importante considerarlo durante la etapa de diseño.
- **Complejidad:** este atributo afecta al sistema en general por lo cual debe ser considerado en todas las etapas del proceso de desarrollo del producto. Además, este atributo se relaciona con la modularidad del producto debido a que si éste tiene baja modularidad los componentes podrían tener muchas funcionalidades aumentado así la complejidad del software.
- **Consistencia:** ya que este atributo implica que el software tenga notación, terminología y simbología uniforme, favorece a la capacidad para entender el producto lo cual a su vez ayuda a mejorar la capacidad para ser analizado del mismo. De igual forma, al producto ser consistente los cambios necesarios pueden ser realizados con mayor facilidad, esto muestra la relación existente entre este atributo y la capacidad para ser modificado.
- **Duplicación:** este atributo incrementa el tamaño del programa y por lo tanto el esfuerzo de las actividades relacionadas con el tamaño como por ejemplo las pruebas, lo cual afecta la capacidad para ser probado.
- **Estandarización:** ya que este atributo implica tener un conjunto de estándares de programación definidos se evidencia que es importante considerarlo durante la etapa de implementación, además este puede mejorar la facilidad de lectura del código fuente, la facilidad de entendimiento del mismo y con esto la capacidad para ser analizado y modificado.
- **Facilidad de entendimiento:** entender bien el código puede ayudar a encontrar más fácilmente donde se deben realizar los cambios y así ejecutarlos con menor dificultad, es decir, que este atributo afecta la capacidad para ser modificado del software. De igual forma esto ayuda a que las pruebas puedan ser elaboradas con mayor facilidad, mejorando así la capacidad para ser probado.
- **Facilidad de lectura:** este atributo favorece el entendimiento del código, lo cual facilita la realización de las pruebas, evidenciando así la relación entre este atributo y la capacidad para ser probado del software.
- **Herencia:** cuando este atributo es utilizado en exceso puede aumentar la complejidad del software, lo cual dificulta el análisis del sistema, es decir, la capacidad para ser analizado.
- **Polimorfismo:** este atributo podría aumentar la complejidad del software, y por lo tanto influye en la capacidad para ser analizado y modificado.
- **Simplicidad:** este atributo se debe tener en cuenta durante todo el proceso de desarrollo buscando un sistema con baja complejidad. Debido a que en un sistema simple toma menor tiempo identificar una clase cuando hay muchas clases se hace más fácil realizar los cambios necesarios por lo cual aumenta la facilidad para ser modificado. Además, este atributo se relaciona con la modularidad del producto debido a que si éste tiene baja modularidad los componentes podrían tener muchas funcionalidades disminuyendo así la simplicidad del software.
- **Tamaño:** ya que el tamaño hace referencia al número de líneas de código, se evidencia que este atributo debe ser considerado en la etapa de implementación. Además, el tener un sistema de gran tamaño podría aumentar la complejidad del mismo, y por lo tanto afectar la capacidad para ser analizado.

- **Trazabilidad:** este atributo debe estar presente en el sistema en todas las etapas del proceso de desarrollo.

A continuación se presenta la Tabla 3 donde se encuentra la síntesis del análisis y clasificación realizados.

Sub-características / Atributos	Fuentes	Capacidad para ser Analizado	Modularidad	Capacidad para ser Modificado	Capacidad para ser Probado	Reusabilidad	Flujo de trabajo
Acoplamiento	[25, 27-31]	+	+	+	+	+	Análisis y Diseño (+)
Anidación	[31]	+		+	+		Análisis y Diseño (-) Implementación (+)
Capacidad de expansión	[7]			+			Despliegue (+)
Cohesión	[25, 28, 29, 31, 32]	+	+	+	+	+	Análisis y Diseño (+)
Comentarios	[7, 20, 24, 29, 31-34, 37]	+		+		+	Implementación (+)
Complejidad	[21, 24, 25, 27-31, 34]	+	-	+	+	+	Análisis y diseño (-) Implementación (-)
Consistencia	[7, 31]	-		-			Requisitos (+) Análisis y Diseño (+) Implementación (+) Pruebas (+) Despliegue (+)
Documentación	[21, 29, 31, 38]	+		+	+	+	Requisitos (+) Análisis y Diseño (+) Implementación (+) Pruebas (+) Despliegue (+)
Duplicación	[24, 27, 30]	+		+	-		Implementación (+)
Encapsulamiento	[29-31, 33]	+	+	+	+		Análisis y Diseño(+)
Estandarización	[21, 29]	-		-			Implementación (-)
Facilidad de lectura	[21, 38]	+		+	-	+	Implementación (+)
Facilidad de entendimiento	[38]	+		-	-	+	Análisis y Diseño (+) Implementación (+)
Herencia	[28, 29, 33]	-		+	+	+	Análisis y Diseño (+)
Polimorfismo	[28]	-		-		+	Análisis y Diseño (+)
Simplicidad	[29, 33]	+	-	-	+	+	Requisitos (-) Análisis y Diseño (-) Implementación (-)
Tamaño	[24, 27, 28, 30-32, 34]	-		+	+	+	Implementación (+)
Trazabilidad	[21]	+		+			Requisitos (-) Análisis y Diseño (+) Implementación (+) Pruebas (-)

Tabla 3. Clasificación de los atributos por sub-características de mantenibilidad y flujos de trabajo. (+) Indica que la relación es justificada por la literatura y (-) la relación es justificada por un análisis de los investigadores.

3.2.1.2. Clasificar los atributos de acuerdo a los procesos

Para dar una mejor estructura al modelo de referencia, se clasificaron los atributos de acuerdo a los procesos definidos en el estándar internacional ISO/IEC 15504-5 teniendo en cuenta la clasificación realizada previamente por flujos de trabajo. Los procesos que han tenido en cuenta son: análisis de requisitos de software, diseño arquitectural de software, diseño detallado de software, construcción de software, pruebas de evaluación de software, operación de software y gestión de la documentación, Fue necesario realizar un análisis semántico de las definiciones de los flujos de trabajo y los procesos definidos con el fin de obtener la relación entre los atributos de software y los procesos. Estos resultados se muestran en la Tabla 4.

Atributos	Procesos
Acoplamiento	Diseño Arquitectural Diseño Detallado
Anidación	Construcción de Software
Capacidad de expansión	Operación de Software
Cohesión	Diseño Detallado
Comentarios	Construcción de Software
Complejidad	Diseño Arquitectural Diseño Detallado Construcción de Software
Consistencia	Análisis de Requisitos de Software Diseño Arquitectural Diseño Detallado Construcción de Software Pruebas de Evaluación de Software Operación de software
Documentación	Análisis de Requisitos de Software Diseño Arquitectural Diseño Detallado Construcción de Software Pruebas de Evaluación de Software Operación de software
Duplicación	Construcción de Software
Encapsulamiento	Diseño Detallado
Estandarización	Construcción de Software
Facilidad de lectura	Construcción de Software
Facilidad de entendimiento	Diseño Arquitectural Diseño Detallado Construcción de Software
Herencia	Diseño Detallado
Polimorfismo	Diseño Detallado
Simplicidad	Análisis de Requisitos de Software Diseño Arquitectural Diseño Detallado Construcción de Software
Tamaño	Construcción de Software

Trazabilidad	Análisis de Requisitos de Software Diseño Arquitectural Diseño Detallado Construcción de Software Pruebas de Evaluación de Software

Tabla 4. Clasificación de los atributos y procesos.

3.2.2. Analizar como es posible potenciar cada sub-característica.

El modelo de referencia propuesto sigue el paradigma orientado a objetos ya que los estudios revisados en la literatura proponen atributos relacionados con este paradigma. Para cada una de las cinco sub-características se analizó cómo es posible potenciarla mediante la realización de prácticas que permitan incluir los diferentes atributos (presentados en la Tabla 2) que influyen sobre la misma, dichas prácticas son separadas de acuerdo al proceso en el cual deben realizarse, todo esto teniendo en cuenta la clasificación desarrollada (la cual se presenta en la sección 3.2). Aunque algunas prácticas coinciden para las diferentes sub-carácterísticas, se hace necesario describirlas para cada una de ellas debido a que la justificación difiere entre sub-características.

3.2.2.1. Sub-característica de reusabilidad.

Para potenciar la sub-característica de reusabilidad en el producto software, es conveniente considerar los diferentes atributos que afectan directamente esta sub-característica, teniendo en cuenta la relación establecida en la Tabla 3.

- **Acoplamiento:** es importante que el software cuente con bajo acoplamiento. Para lograr esto, durante el proceso de diseño se debe descomponer el sistema en pequeñas partes de funcionalidades (módulos) que tengan la menor interrelación posible. Para que estos módulos (paquetes, clases, métodos) sean fáciles de reutilizar es oportuno abstraer la funcionalidad de dichos módulos, así los módulos podrían ser reutilizados en diferentes contextos fuera del sistema original. Además de esto se debe reducir el número de respuestas por clases.

- **Cohesión:** es conveniente que el software tenga alta cohesión. Por esto, es primordial que los módulos (paquetes) del sistema cuenten con una fuerte cohesión funcional, lo que implica que las clases en el módulo estén relacionadas de acuerdo a su función, es decir, que dicho módulo tenga una única preocupación, lo cual debe ser considerado en la etapa de diseño. La separación de preocupaciones tiene un impacto positivo sobre la reusabilidad, ya que se facilita la localización de la funcionalidad que se pretende reutilizar.

- **Comentarios:** durante la etapa de implementación es pertinente hacer uso de comentarios en el código fuente, como mínimo la documentación dentro de éste debe explicar el propósito de las funciones, subrutinas, variables y constantes. Que un experto sea capaz de entender el código es muy importante para la reutilización. Si un programador no puede comprender la funcionalidad del código, es difícil que identifique que parte del código fuente le permite cubrir sus necesidades.

- **Complejidad:** para que un producto software tenga una baja complejidad, debe contar con bajo acoplamiento y alta cohesión. Además, se debe reducir el número de

métodos por clase. También se debe evitar la complejidad de estos métodos, es decir, incluir solamente los flujos de datos y de control necesarios.

- **Documentación:** es necesario que el producto software tenga una buena documentación, es decir, que sea completa y actualizada. Además de esto la documentación debe contar con un estándar adecuado. Lo anterior se refiere a usar símbolos convencionales en todos los diagramas, utilizar solo un método de documentación dentro de la organización, debe ser titulada con claridad y bien organizada, con secciones indicadas con precisión, los diagramas deben ser claros y separados.

- **Facilidad de lectura:** el código fuente debe ser fácil de leer para asegurar la reusabilidad del mismo. Este atributo podría ser mejorado teniendo en cuenta los siguientes aspectos al escribir el código: para mejorar la comprensión del código se debe indentar el mismo; las variables deben tener nombres descriptivos y se debe hacer poco uso de abreviaciones; los comentarios deben ser utilizados adecuadamente; evitar sentencias largas, es mejor mantener las líneas de código cortas, aunque esto implica separar las sentencias en múltiples líneas; hacer un correcto uso de paréntesis para mejorar la facilidad de lectura de las expresiones aritméticas y lógicas.

- **Facilidad de entendimiento:** el diseño y el código fuente deben tener una alta facilidad de entendimiento, lo cual se puede lograr teniendo en cuenta atributos como el tamaño, acoplamiento, cohesión. El acoplamiento y la cohesión afectan la capacidad de entendimiento porque un componente de un sistema no puede ser entendido sin referenciar a otros componentes con los que esté relacionado, además la separación de preocupaciones también afecta a este atributo porque entre más centradas están dichas preocupaciones será más fácil entenderlas. El tamaño del diseño y el código podría afectar la facilidad de entendimiento de los componentes software al disminuir o aumentar el esfuerzo necesario para entenderlos. Teniendo en cuenta lo anterior, si se hace necesario reutilizar una funcionalidad se hace más fácil ubicarla ya que esta estará localizada en un solo componente y por lo tanto las actividades de reutilización estarán limitadas a este componente.

- **Herencia:** este atributo debe ser considerado durante la etapa de diseño ya que ayuda a mejorar la reusabilidad del sistema al facilitar al desarrollador la localización de las clases reutilizables. Sin embargo, la herencia debe usarse con moderación, es decir, hasta cierto nivel de jerarquía porque ésta afecta a otros factores como la complejidad, capacidad para ser entendido, capacidad para ser modificado y probado. En [92] se recomienda una herencia de tres niveles de profundidad ya que se observó que hasta ese nivel las tareas de mantenimiento pueden ser realizadas rápidamente, pero cuando el nivel es superior a tres las tareas toman más tiempo.

- **Polimorfismo:** cuando se tenga que llevar a cabo una responsabilidad que dependa de un tipo (clase) es conveniente hacer uso del polimorfismo, es decir, asignar el mismo nombre a servicios implementados en diferentes objetos. Para esto, se debe hacer uso de las clases abstractas.

- **Simplicidad:** durante todo el proceso de desarrollo se debe considerar que el sistema se debe mantener simple, evitando la complejidad innecesaria. Esto se podría lograr

teniendo en cuenta solamente las funcionalidades básicas e ir incrementando lo necesario, de esta forma se evita incluir aspectos que agreguen complejidad al sistema. Además, se deben implementar solamente las funcionalidades que realmente se necesitan y no las que se cree que podrían ser necesarias. Otros aspectos a tener en cuenta son: dividir las tareas en sub-tareas entendibles, las cuales deben ser realizadas por una o pocas clases; mantener pequeños los métodos, donde cada uno de estos debería ser responsable de una única tarea; si se tiene muchas condiciones en un método, sería conveniente dividirlo en pequeños métodos; se debe intentar tener pequeñas clases.

- **Tamaño:** se debe evitar que el tamaño del código crezca innecesariamente. Para esto es importante evitar la duplicación, mantener el código simple.

En la Tabla 5 se observan las prácticas que se podrían realizar para incluir los atributos relacionados, en cada uno de los procesos con el fin de potenciar la sub-característica de modularidad.

Proceso	Prácticas	Atributo
Análisis de Requisitos de Software	• Incluir solamente los requisitos necesarios, es decir, aquellos que son indispensables para el buen funcionamiento del sistema, evitando así agregar complejidad al mismo. • Expresar la información suficiente para la comprensión de los requisitos, sin agregar datos innecesarios. • Enunciar en los requisitos solamente lo que el sistema debe hacer, no como debe hacerlo.	**Complejidad** **Simplicidad**
	• Identificar los requisitos de manera única, haciendo uso de un sistema de numeración. • Expresar los requisitos de manera sencilla, clara y sin ambigüedades • Expresar los requisitos no funcionales de manera cuantitativa, por ejemplo no decir que algo debe ser rápido, sino qué tan rápido debe ser. • Escribir los requisitos de manera organizada, con el fin de poder hacer un seguimiento a los mismos. • Redactar los requisitos en un lenguaje simple, sencillo y contundente para facilitar la comprensión de todos los participantes del proyecto. • Usar términos del dominio del problema con los que los clientes y usuarios estén familiarizados, los cuales deben ser documentados en el	**Documentación**

	glosario de términos de la ERS.	
Diseño Arquitectural de Software	• Descomponer el sistema en pequeña partes de funcionalidades (módulos). • Crear módulos (paquetes, clases, métodos) que tengan la menor interrelación posible. • Abstraer la funcionalidad de los módulos haciendo uso de clases y métodos abstractos.	**Acoplamiento** **Complejidad** **Facilidad de entendimiento** **Simplicidad**
	• Elaborar diagramas de diseño de alto nivel claros y separados.	**Facilidad de Entendimiento**
	• Dividir las tareas en sub-tareas entendibles. • Realizar funcionalidades con una o pocas clases.	**Complejidad** **Simplicidad**
	• Mantener desacoplada de las capas por encima. • Depender solo de los componentes más generales en las capas inferiores.	**Acoplamiento** **Complejidad** **Simplicidad**
Diseño Detallado de Software	• Elaborar diagramas de diseño de bajo nivel claros y separados.	**Facilidad de Entendimiento**
	• Tener como máximo 3 niveles de profundidad de herencia.	**Complejidad** **Facilidad de entendimiento** **Herencia** **Simplicidad**
	• Cuando se tiene que llevar a cabo una responsabilidad que dependa de un tipo, asignar el mismo nombre a servicios implementados en diferentes objetos.	**Complejidad** **Polimorfismo** **Simplicidad**
	• Tener en cuenta solamente las funcionalidades necesarias. • Dividir las tareas en sub-tareas entendibles. • Realizar funcionalidades con una o pocas clases. • Cada método deber ser responsable de una sola tarea. • Tener clases de tamaño adecuado. • Reducir el número de métodos por clase. • Incluir solamente los datos de entrada y salida necesarios en las funciones.	**Complejidad** **Simplicidad**

	• Cada módulo (paquete) debe tener una única preocupación, lo cual implica que las clases en el módulo estén relacionadas de acuerdo a su función.	**Cohesión Complejidad Facilidad de entendimiento Simplicidad**
Construcción de Software	• Hacer uso de comentarios adecuados en el código fuente. • Explicar el propósito de las funciones, subrutinas, variables y constantes.	**Comentarios Documentación Facilidad de entendimiento Facilidad de lectura**
	• Incluir solamente los flujos de control necesarios.	**Complejidad Facilidad de entendimiento Simplicidad**
	• Indentar el código. • Dar nombres descriptivos a las variables. • Hacer poco uso de abreviaciones. • Evitar sentencias largar. • Hacer correcto uso de los paréntesis.	**Facilidad de lectura Facilidad de entendimiento**
	• Implementar solamente las funcionalidades necesarias. • Mantener métodos simples, es decir, dividir los métodos que tengan muchas condiciones.	**Complejidad Facilidad de entendimiento Simplicidad Tamaño**
	• Evitar que el tamaño del código crezca innecesariamente. • Evitar la duplicación de código.	**Facilidad de entendimiento Tamaño**
Pruebas de evaluación de software	• Expresar el resumen de las pruebas que han sido realizadas. • Escribir cuales módulos se han probado y si se han probado a fondo. • Indicar claramente los errores encontrados.	**Documentación**

Tabla 5. Prácticas para la sub-característica de reusabilidad

Debido a que el proceso de gestión de la documentación es un proceso de soporte que apoya a los procesos de implementación de software, se definieron algunas prácticas que podrían realizarse en los diferentes procesos con el fin de potenciar la reusabilidad del producto. Esta información puede observarse en la Tabla 6.

GESTION DE LA DOCUMENTACIÓN	
Prácticas	**Proceso**
• Mantener Actualizado el documento de requisitos. • Cumplir con un estándar adecuado. • Titular el documento de requisitos con claridad. • Indicar las secciones de los documentos con precisión	**Análisis de Requisitos de Software**

• Usar símbolos convencionales en todos los diagramas. • Cumplir con un estándar adecuado. • Titular los diagramas y documentos de forma clara. • Mantener los diagramas y documentos bien organizados. • Indicar las secciones de los documentos con precisión.	**Diseño Arquitectural de software** **Diseño Detallado de Software** **Facilidad de Entendimiento**
• Titular el documento de pruebas con claridad. • Cumplir con un estándar adecuado. • Mantener el documento de pruebas actualizado. • Indicar las secciones de los documentos con precisión.	**Pruebas de Evaluación de Software**
• Titular los documentos con claridad. • Cumplir con un estándar adecuado. • Mantener el documento de pruebas actualizado. • Elaborar manuales de administración y de usuario. • Elaborar documentos de instalación, configuración, integración y migración. • Indicar las secciones de los documentos con precisión.	**Operación de software**

Tabla 6. Prácticas de documentación para la sub-característica de reusabilidad

3.2.2.2. Sub-característica de capacidad para ser analizado.

Para potenciar la sub-característica de capacidad para ser analizado en el producto software, es conveniente considerar los diferentes atributos que afectan directamente esta sub-característica, teniendo en cuenta la relación establecida en la Tabla 3:

- **Acoplamiento:** es fundamental que el software cuente con bajo acoplamiento. Para lograr esto, durante el proceso de diseño se debe descomponer el sistema en pequeñas partes de funcionalidades (módulos) que tengan la menor interrelación posible. Tener alto acoplamiento es señal de un diseño complejo que disminuye la facilidad de entendimiento y por lo tanto la capacidad para ser analizado del sistema. Además de lo anterior, se debe reducir el número de respuestas por clases ya que esto también afecta la facilidad de entendimiento. Este atributo se relaciona con el encapsulamiento.

- **Anidación:** es apropiado contar con un bajo nivel de anidación en el software ya que esto aumenta la facilidad de entendimiento y por lo tanto la capacidad para ser analizado del mismo. Para esto durante el proceso de diseño detallado se debe centrar la atención en el atributo de herencia para evitar la excesiva profundidad de anidación en los módulos (clases). También es apropiado en el proceso de construcción de software evitar las estructuras de control anidadas, ya que estas son más complejas que las estructuras de control secuenciales.

- **Cohesión:** es conveniente que el software tenga alta cohesión ya que aumenta la facilidad de entendimiento y por lo tanto la capacidad para ser analizado del mismo. Por esto, es primordial que los módulos (paquetes) del sistema cuenten con una fuerte cohesión funcional, lo que implica que las clases en el módulo estén relacionadas de acuerdo a su función, es decir, que dicho módulo tenga una única preocupación, lo cual debe ser considerado en la etapa de diseño. Para garantizar la cohesión es oportuno eliminar el código que no pertenece a la actividad realizada por el módulo (paquete), creando otro módulo para este.

- **Comentarios:** durante la etapa de implementación es pertinente hacer uso de comentarios en el código fuente, como mínimo la documentación dentro de éste debe explicar el propósito de las funciones, subrutinas, variables y constantes. Si un programador no puede comprender la funcionalidad del código, la capacidad para ser analizado del sistema disminuye.

- **Complejidad:** para que un producto software tenga una baja complejidad, debe contar con bajo acoplamiento, alta cohesión, bajo nivel de anidación y considerar los atributos encapsulamiento y polimorfismo. Además, se debe reducir el número de métodos por clase. También se debe evitar la complejidad de estos métodos, es decir, incluir solamente los flujos de datos y de control necesarios.

- **Consistencia:** el software debe ser consistente, para esto en el proceso de análisis de requisitos de software se debe verificar que los requisitos sean consistentes entre sí, es decir, no debe haber conflictos o incompatibilidad entre ellos. Además en todos los procesos se debe comprobar que los artefactos asocien y contengan notación, terminología y simbología uniforme, es apropiado hacer uso de convenciones de nombrado. Este atributo está asociado a otros como estandarización, facilidad de lectura, trazabilidad.

- **Documentación:** es necesario que el producto software tenga una buena documentación, es decir, que sea completa y actualizada. Además de esto la documentación debe contar con un estándar adecuado. Lo anterior se refiere a usar símbolos convencionales en todos los diagramas, utilizar solo un método de documentación dentro de la organización, debe ser titulada con claridad y bien organizada, con secciones indicadas con precisión, definiendo los términos utilizados, los diagramas deben ser claros y separados.

- **Duplicación:** la duplicación aumenta el tamaño del código y su complejidad, lo cual afecta la capacidad para ser analizado del software. La duplicación de código se puede resolver detectando el código duplicado, extrayéndolo en un nuevo procedimiento y reemplazando todas las instancias del código duplicado por llamadas al nuevo procedimiento.

- **Encapsulamiento:** este atributo tiene efecto sobre la capacidad para ser analizado del software ya que oculta la complejidad detrás de interfaces simples. Es apropiado tener en cuenta el ocultamiento de la información, esto se puede lograr ocultando los detalles de implementación dentro de los módulos (clases). Para esto, primero los diseñadores podrían identificar los servicios que el módulo (clase) proporcionara al sistema y al mismo tiempo identificar un conjunto de detalles de implementación que podrán cambiar durante el desarrollo, es decir, hacer privadas las variables que son innecesarias para el tratamiento del objeto pero necesarias para su funcionamiento, así como las funciones que no necesitan interacción del usuario o que solo pueden ser llamadas por otras funciones dentro del objeto. Cada uno de estos detalles de implementación debería encapsularse en un módulo (clase) independiente, así los cambios del sistema se limitan al módulo en cuestión.

- **Estandarización:** es apropiado tener un conjunto de estándares de programación como guía en la escritura de código para evitar la individualidad entre los

programadores, ayudando así a la capacidad para ser analizado del software. Se debe hacer uso de convenciones estándar para el nombrado de paquetes, clases, métodos y variables.

- **Facilidad de lectura:** el código fuente debe ser fácil de leer para asegurar la capacidad para ser analizado del mismo. Este atributo podría ser mejorado teniendo en cuenta los siguientes aspectos al escribir el código: para mejorar la comprensión del código se debe indentar el mismo; las variables deben tener nombres descriptivo, coherente y se debe hacer poco uso de abreviaciones; los comentarios deben ser utilizados adecuadamente; evitar sentencias largas, es mejor mantener las líneas de código cortas, aunque esto implica separar las sentencias en múltiples líneas; hacer un correcto uso de paréntesis para mejorar la facilidad de lectura de las expresiones aritméticas y lógicas.

- **Facilidad de entendimiento:** el diseño y el código fuente deben tener una alta facilidad de entendimiento, lo cual se puede lograr teniendo en cuenta atributos como el tamaño, acoplamiento, cohesión, estandarización. Además de esto, las clases deben ser identificadas fácilmente para que puedan ser mapeadas desde los requerimientos hacia el código.

- **Herencia:** este atributo debe usarse con moderación, es decir, hasta cierto nivel de jerarquía porque ésta afecta a otros factores como la complejidad, capacidad para ser entendido, capacidad para ser modificado y probado. También debido a que las clases heredadas podrían heredar clases o atributos o sobrescribir métodos de sus clases base, se hace difícil entender el comportamiento exacto de las clases simplemente con revisarlas una sola vez. En [92] se recomienda una herencia de tres niveles de profundidad ya que se observó que hasta ese nivel las tareas de mantenimiento pueden ser realizadas rápidamente, pero cuando el nivel es superior a tres las tareas toman más tiempo.

- **Polimorfismo:** cuando se tenga que llevar a cabo una responsabilidad que dependa de un tipo (clase) es conveniente hacer uso del polimorfismo, es decir, asignar el mismo nombre a servicios implementados en diferentes objetos. Para esto, se debe hacer uso de las clases abstractas. Sin embargo se debe tener en cuenta que el uso excesivo del polimorfismo puede reducir la capacidad para ser analizado del software.

- **Simplicidad:** este atributo se relaciona con los atributos acoplamiento, cohesión, anidación, encapsulamiento y polimorfismo. Durante todo el proceso de desarrollo se debe considerar que el sistema se debe mantener simple, evitando la complejidad innecesaria. Esto se podría lograr teniendo en cuenta solamente las funcionalidades básicas e ir incrementando lo necesario, de esta forma se evita incluir aspectos que agreguen complejidad al sistema. Además, se deben implementar solamente las funcionalidades que realmente se necesitan y no las que se cree que podrían ser necesarias. Otros aspectos a tener en cuenta son: dividir las tareas en sub-tareas entendibles, las cuales deben ser realizadas por una o pocas clases; mantener pequeños los métodos, donde cada uno de estos debería ser responsable de una única tarea; si se tiene muchas condiciones en un método, sería conveniente dividirlo en pequeños métodos; se debe intentar tener pequeñas clases.

- **Tamaño:** se debe evitar que el tamaño del código crezca innecesariamente. Para esto es importante evitar la duplicación, mantener el código simple.
- **Trazabilidad:** este atributo asegura que los artefactos software estén siempre consistentes y completos, lo cual apoya la capacidad para ser analizado del software. Para esto es importante verificar periódicamente que los artefactos de una etapa sean consistentes con los artefactos de la etapa anterior y actualizarlos cuando se presenten cambios.

En el Anexo A se observan las prácticas que se podrían realizar para incluir los atributos relacionados, en cada uno de los procesos con el fin de potenciar la sub-característica de capacidad para ser analizado.

3.2.2.3. Sub-característica de modularidad

Para potenciar la sub-característica de modularidad en el producto software, es conveniente considerar los diferentes atributos que afectan directamente esta sub-característica, teniendo en cuenta la relación establecida en Tabla 3:

- **Acoplamiento:** es importante que el software cuente con bajo acoplamiento, ya que esto permite que los módulos (paquetes, clases, métodos) sean más sencillos de diseñar, programar, probar y mantener. Durante el proceso de diseño se debe descomponer el sistema en pequeñas partes de funcionalidades (módulos) que tengan la menor interrelación posible. A nivel de procedimientos, funciones y métodos lo anterior se podría lograr reduciendo la cantidad y complejidad de los parámetros y disminuyendo al mínimo los parámetros por referencia. A nivel de clases se podría lograr un bajo acoplamiento reduciendo la dependencia entre clases, es decir, deberían existir pocas dependencias hacia otras clases desde una sola. A nivel de paquetes, la vinculación con otros paquetes debería ser mínima, es decir, las clases de un paquete no deben depender de las clases de otros paquetes. Además de esto se debe reducir el número de respuestas por clases.

- **Cohesión:** es conveniente que el software tenga alta cohesión. Por esto, es primordial que los módulos (paquetes) del sistema cuenten con una fuerte cohesión funcional, lo que implica que las clases en el módulo estén relacionadas de acuerdo a su función, es decir, que dicho módulo tenga una única preocupación, lo cual debe ser considerado en la etapa de diseño. Se debería disminuir las responsabilidades de una clase, así si una clase tiene muchas responsabilidades probablemente haya que dividirla en dos o más. A nivel de procedimientos o métodos es apropiado que estos realicen una única tarea.

- **Complejidad:** para que un producto software tenga una baja complejidad, debe contar con bajo acoplamiento y alta cohesión. Además, se debe reducir el número de métodos por clase. También se debe evitar la complejidad de estos métodos, es decir, incluir solamente los flujos de datos necesarios. De igual forma para reducir la complejidad del software es conveniente tener módulos (paquetes, clases) pequeños que se pueden diseñar y desarrollar de manera sencilla y que tengan lo mayor independencia posible del resto de la aplicación.

- **Encapsulamiento:** es apropiado tener en cuenta el ocultamiento de la información, esto se puede lograr ocultando los detalles de implementación dentro de los módulos (clases). Para esto, primero los diseñadores podrían identificar los servicios que el

módulo (clase) proporcionara al sistema y al mismo tiempo identifican un conjunto de estos detalles de implementación que podrán cambiar durante el desarrollo, es decir, hacer privadas las variables que son innecesarias para el tratamiento del objeto pero necesarias para su funcionamiento, así como las funciones que no necesitan interacción del usuario o que solo pueden ser llamadas por otras funciones dentro del objeto. Cada uno de estos detalles de implementación debería encapsularse en un módulo (clase) independiente, así los cambios del sistema se limitan al módulo en cuestión.

- **Simplicidad:** este atributo se relaciona con los atributos acoplamiento, cohesión y encapsulamiento. Durante todo el proceso de desarrollo se debe considerar que el sistema se debe mantener simple, evitando la complejidad innecesaria. Esto se podría lograr teniendo en cuenta solamente las funcionalidades básicas e ir incrementando lo necesario, de esta forma se evita incluir aspectos que agreguen complejidad al sistema. Además, se deben implementar solamente las funcionalidades que realmente se necesitan y no las que se cree que podrían ser necesarias. Otros aspectos a tener en cuenta son: dividir las tareas en sub-tareas entendibles, las cuales deben ser realizadas por una o pocas clases; mantener pequeños los métodos, donde cada uno de estos debería ser responsable de una única tarea; si se tiene muchas condiciones en un método, sería conveniente dividirlo en pequeños métodos; se debe intentar tener pequeñas clases.

En el Anexo B se observan las prácticas que se podrían realizar para incluir los atributos relacionados, en cada uno de los procesos con el fin de potenciar la sub-característica de modularidad.

3.2.2.4. Sub-característica capacidad para ser modificado

Para potenciar la sub-característica de capacidad para ser modificado en el producto software, es conveniente considerar los diferentes atributos que afectan directamente esta sub-característica, teniendo en cuenta la relación establecida en la Tabla 3.

- **Acoplamiento:** es fundamental que el software cuente con bajo acoplamiento. Para lograr esto, durante el proceso de diseño se debe descomponer el sistema en pequeñas partes de funcionalidades (módulos) que tengan la menor interrelación posible. Para que estos módulos (paquetes, clases, métodos) sean fáciles de modificar es oportuno abstraer la funcionalidad de dichos módulos. Tener alto acoplamiento hace que las modificaciones se propaguen a los módulos (paquetes, clases) que están fuertemente acoplados con el módulo que será modificado. Además de lo anterior, se debe reducir el número de respuestas por clases ya que esto también afecta la facilidad de entendimiento. Este atributo se relaciona con el encapsulamiento y la anidación.

- **Anidación:** es apropiado contar con un bajo nivel de anidación en el software ya que así se podría disminuir las interdependencias entre clases y facilitar la modificación de componentes. Para esto durante el proceso de diseño detallado se debe centrar la atención en el atributo de herencia para evitar la excesiva profundidad de anidación en los módulos (clases). También es apropiado en el proceso de construcción de software reducir las estructuras de control anidadas, ya que estas son más complejas que las estructuras de control secuenciales.

- **Capacidad de expansión:** para tener una alta capacidad de expansión es necesario que el producto cuente con: bajo acoplamiento, alta cohesión, simplicidad, baja complejidad, facilidad de entendimiento.

- **Cohesión:** es conveniente que el software tenga alta cohesión. Por esto, es primordial que los módulos (paquetes) del sistema cuenten con una fuerte cohesión funcional, lo que implica que las clases en el módulo estén relacionadas de acuerdo a su función, es decir, que dicho módulo tenga una única preocupación, lo cual debe ser considerado en la etapa de diseño. Al separar las responsabilidades, se facilita la localización de los módulos (paquetes, clases) que se van a ver afectados por el cambio. Para garantizar la cohesión es oportuno eliminar el código que no pertenece a la actividad realizada por el módulo (paquete), creando otro módulo para este.

- **Comentarios:** durante la etapa de implementación es pertinente hacer uso de comentarios en el código fuente, como mínimo la documentación dentro de éste debe explicar el propósito de las funciones, subrutinas, variables y constantes. Si un programador no puede comprender la funcionalidad del código, la capacidad para ser analizado del sistema disminuye.

- **Complejidad:** para que un producto software tenga una baja complejidad, debe contar con bajo acoplamiento, alta cohesión, bajo nivel de anidación y considerar los atributos encapsulamiento y polimorfismo. Además, se debe reducir el número de métodos por clase. También se debe evitar la complejidad de estos métodos, es decir, incluir solamente los flujos de datos y de control necesarios.

- **Consistencia:** el software debe ser consistente, para esto en el proceso de análisis de requisitos de software se debe verificar que los requisitos sean consistentes entre sí, es decir, no debe haber conflictos o incompatibilidad entre ellos. Además en todos los procesos se debe comprobar que los artefactos asocien y contengan notación, terminología y simbología uniforme, es apropiado hacer uso de convenciones de nombrado. Este atributo está asociado a otros como estandarización, facilidad de lectura, trazabilidad.

- **Documentación:** es necesario que el producto software tenga una buena documentación, es decir, que sea completa y actualizada. Además de esto la documentación debe contar con un estándar adecuado. Lo anterior se refiere a usar símbolos convencionales en todos los diagramas, utilizar solo un método de documentación dentro de la organización. La documentación debe ser titulada con claridad y bien organizada, con secciones indicadas con precisión, definiendo los términos utilizados, los diagramas deben ser claros y separados.

- **Duplicación:** la duplicación aumenta el tamaño del código y su complejidad, además dificulta la modificación del código ya cuando se realizan cambios en una instancia de código duplicado, se hace necesario buscar las otras instancias para efectuar el cambio correspondiente. La duplicación de código se puede resolver detectando el código duplicado, extrayéndolo en un nuevo procedimiento y reemplazando todas las instancias del código duplicado por llamadas al nuevo procedimiento.

- **Encapsulamiento:** este atributo introduce interfaces para manipular los datos de un módulo (clase) lo cual permite reducir la probabilidad de que un cambio en un módulo

se propague a otros módulos. Es apropiado tener en cuenta el ocultamiento de la información, esto se puede lograr ocultando los detalles de implementación dentro de los módulos (clases). Para esto, primero los diseñadores podrían identificar los servicios que el módulo (clase) proporcionara al sistema y al mismo tiempo identificar un conjunto de detalles de implementación que podrán cambiar durante el desarrollo, es decir, hacer privadas las variables que son innecesarias para el tratamiento del objeto pero necesarias para su funcionamiento, así como las funciones que no necesitan interacción del usuario o que solo pueden ser llamadas por otras funciones dentro del objeto. Cada uno de estos detalles de implementación debería encapsularse en un módulo (clase) independiente, así los cambios del sistema se limitan al módulo en cuestión. Además, se debe evitar el uso de variables globales y atributos públicos.

- **Estandarización:** es apropiado tener un conjunto de estándares de programación como guía en la escritura de código para evitar la individualidad entre los programadores, ayudando así a la capacidad para ser analizado del software. Se debe hacer uso de convenciones estándar para el nombrado de paquetes, clases, métodos y variables.

- **Facilidad de lectura:** el código fuente debe ser fácil de leer para asegurar la capacidad para ser modificado del mismo. Este atributo podría ser mejorado teniendo en cuenta los siguientes aspectos al escribir el código: para mejorar la comprensión del código se debe indentar el mismo; las variables deben tener nombres descriptivo, coherente y se debe hacer poco uso de abreviaciones; los comentarios deben ser utilizados adecuadamente; evitar sentencias largas, es mejor mantener las líneas de código cortas, aunque esto implica separar las sentencias en múltiples líneas; hacer un correcto uso de paréntesis para mejorar la facilidad de lectura de las expresiones aritméticas y lógicas.

- **Facilidad de entendimiento:** el diseño y el código fuente deben tener una alta facilidad de entendimiento, lo cual se puede lograr teniendo en cuenta atributos como el tamaño, acoplamiento, cohesión, estandarización. Además de esto, las clases deben ser identificadas fácilmente para que puedan ser mapeadas desde los requerimientos hacia el código.

- **Herencia:** este atributo debe usarse con moderación, es decir, hasta cierto nivel de jerarquía porque ésta afecta a otros factores como la complejidad, capacidad para ser entendido, capacidad para ser modificado y probado. También debido a que las clases heredadas podrían heredar clases o atributos o sobrescribir métodos de sus clases base, se hace difícil entender el comportamiento exacto de las clases simplemente con revisarlas una sola vez. En [92] se recomienda una herencia de tres niveles de profundidad ya que se observó que hasta ese nivel las tareas de mantenimiento pueden ser realizadas rápidamente, pero cuando el nivel es superior a tres estas tareas toman más tiempo.

- **Polimorfismo:** cuando se tenga que llevar a cabo una responsabilidad que dependa de un tipo (clase) es conveniente hacer uso del polimorfismo, es decir, asignar el mismo nombre a servicios implementados en diferentes objetos. Para esto, se debe hacer uso de las clases abstractas. Sin embargo se debe tener en cuenta que el uso excesivo del polimorfismo puede reducir la capacidad para ser modificado del software.

- **Simplicidad:** este atributo se relaciona con los atributos acoplamiento, cohesión, anidación, encapsulamiento y polimorfismo. Durante todo el proceso de desarrollo se debe considerar que el sistema se debe mantener simple, evitando la complejidad innecesaria. Esto se podría lograr teniendo en cuenta solamente las funcionalidades básicas e ir incrementando lo necesario, de esta forma se evita incluir aspectos que agreguen complejidad al sistema. Además, se deben implementar solamente las funcionalidades que realmente se necesitan y no las que se cree que podrían ser necesarias. Otros aspectos a tener en cuenta son: dividir las tareas en sub-tareas entendibles, las cuales deben ser realizadas por una o pocas clases; mantener pequeños los métodos, donde cada uno de estos debería ser responsable de una única tarea; si se tiene muchas condiciones en un método, sería conveniente dividirlo en pequeños métodos; se debe intentar tener pequeñas clases.

- **Tamaño:** se debe evitar que el tamaño del código crezca innecesariamente. Para esto es importante evitar la duplicación, mantener el código simple.

- **Tamaño de unidad:** Las unidades como piezas de funcionalidad de más bajo nivel deben mantenerse pequeñas para que sean centradas y fáciles de entender.

- **Trazabilidad:** este atributo asegura que los artefactos software estén siempre consistentes y completos. Para esto es importante verificar periódicamente que los artefactos de una etapa sean consistentes con los artefactos de la etapa anterior y actualizarlos cuando se presenten cambios.

En el Anexo C se observan las prácticas que se podrían realizar para incluir los atributos relacionados, en cada uno de los procesos con el fin de potenciar la sub-característica de capacidad de ser modificado.

3.2.2.5. Sub-característica capacidad para ser probado.

Para potenciar la sub-característica de capacidad para ser probado en el producto software, es conveniente considerar los diferentes atributos que afectan directamente esta sub-característica, teniendo en cuenta la relación establecida en la Tabla 3.

- **Acoplamiento:** es fundamental que el software cuente con bajo acoplamiento ya que un alto nivel de acoplamiento es señal de un diseño complejo, lo cual complica las pruebas de software y por lo tanto disminuye la capacidad para ser probado del producto. Para lograr un bajo acoplamiento, durante el proceso de diseño se debe descomponer el sistema en pequeñas partes de funcionalidades (módulos) que tengan la menor interrelación posible. Además de lo anterior, se debe reducir el número de respuestas por clases ya que esto también afecta la capacidad para ser probado, debido a que el tester debe inicializar instancias de las clases cuyos métodos son invocado por la clase que está siendo probada. Este atributo se relaciona con el encapsulamiento.

- **Anidación:** es apropiado contar con un bajo nivel de anidación en el software ya que esto aumenta la facilidad de entendimiento y por lo tanto la capacidad para ser probado del mismo. Para esto durante el proceso de diseño detallado se debe centrar la atención en el atributo de herencia para evitar la excesiva profundidad de anidación en los módulos (clases). También es apropiado en el proceso de construcción de software

evitar las estructuras de control anidadas, ya que estas son más complejas que las estructuras de control secuenciales.

- **Cohesión:** es conveniente que el software tenga alta cohesión ya que aumenta la facilidad de entendimiento y por lo tanto la capacidad para ser probado del mismo. Por esto, es primordial que los módulos (paquetes) del sistema cuenten con una fuerte cohesión funcional, lo que implica que las clases en el módulo estén relacionadas de acuerdo a su función, es decir, que dicho módulo tenga una única preocupación, lo cual debe ser considerado en la etapa de diseño. Para garantizar la cohesión es oportuno eliminar el código que no pertenece a la actividad realizada por el módulo (paquete), creando otro módulo para este.

- **Complejidad:** para que un producto software tenga una baja complejidad, debe contar con bajo acoplamiento, alta cohesión, bajo nivel de anidación y considerar los atributos encapsulamiento y polimorfismo. Además, se debe reducir el número de métodos por clase. También se debe evitar la complejidad de estos métodos, es decir, incluir solamente los flujos de datos y de control necesarios.

- **Documentación:** es necesario que el producto software tenga una buena documentación, es decir, que sea completa y actualizada. Además de esto la documentación debe contar con un estándar adecuado. Lo anterior se refiere a usar símbolos convencionales en todos los diagramas, utilizar solo un método de documentación dentro de la organización, debe ser titulada con claridad y bien organizada, con secciones indicadas con precisión, definiendo los términos utilizados, los diagramas deben ser claros y separados.

- **Duplicación:** la duplicación aumenta el tamaño del código y su complejidad, lo cual afecta la capacidad para ser probado del software. La duplicación de código se puede resolver detectando el código duplicado, extrayéndolo en un nuevo procedimiento y reemplazando todas las instancias del código duplicado por llamadas al nuevo procedimiento.

- **Encapsulamiento:** este atributo ayuda a disminuir la complejidad del software por lo podría aumentar la capacidad para ser probado. Es apropiado tener en cuenta el ocultamiento de la información, esto se puede lograr ocultando los detalles de implementación dentro de los módulos (clases). Para esto, primero los diseñadores podrían identificar los servicios que el módulo (clase) proporcionara al sistema y al mismo tiempo identificar un conjunto de detalles de implementación que podrán cambiar durante el desarrollo, es decir, hacer privadas las variables que son innecesarias para el tratamiento del objeto pero necesarias para su funcionamiento, así como las funciones que no necesitan interacción del usuario o que solo pueden ser llamadas por otras funciones dentro del objeto. Cada uno de estos detalles de implementación debería encapsularse en un módulo (clase) independiente, así los cambios del sistema se limitan al módulo en cuestión. Además, se debe evitar el uso de variables globales y atributos públicos.

- **Facilidad de lectura:** el código fuente debe ser fácil de leer para beneficiar la capacidad para ser probado del mismo. Este atributo podría ser mejorado teniendo en cuenta los siguientes aspectos al escribir el código: para mejorar la comprensión del código se debe indentar el mismo; las variables deben tener nombres descriptivo,

coherente y se debe hacer poco uso de abreviaciones; los comentarios deben ser utilizados adecuadamente; evitar sentencias largas, es mejor mantener las líneas de código cortas, aunque esto implica separar las sentencias en múltiples líneas; hacer un correcto uso de paréntesis para mejorar la facilidad de lectura de las expresiones aritméticas y lógicas.

- **Facilidad de entendimiento:** el diseño y el código fuente deben tener una alta facilidad de entendimiento, lo cual se puede lograr teniendo en cuenta atributos como el tamaño, acoplamiento, cohesión, estandarización. Además de esto, las clases deben ser identificadas fácilmente para que puedan ser mapeadas desde los requerimientos hacia el código.

- **Herencia:** este atributo debe ser considerado durante la etapa de diseño ya que ayuda a mejorar la capacidad para ser probado del sistema ya que los casos de prueba usados para los métodos padre pueden ser reutilizados para los métodos hijos, lo cual decrementa el esfuerzo de prueba. Sin embargo, la herencia debe usarse con moderación, es decir, hasta cierto nivel de jerarquía porque ésta afecta a otros factores como la complejidad, capacidad para ser entendido, capacidad para ser modificado y probado. También debido a que las clases heredadas podrían heredar clases o atributos o sobrescribir métodos de sus clases base, se hace difícil entender el comportamiento exacto de las clases simplemente con revisarlas una sola vez. En [92] se recomienda una herencia de tres niveles de profundidad ya que se observó que hasta ese nivel las tareas de mantenimiento pueden ser realizadas rápidamente, pero cuando el nivel es superior a tres las tareas toman más tiempo.

- **Simplicidad:** este atributo se relaciona con los atributos acoplamiento, cohesión, anidación, encapsulamiento y polimorfismo. Durante todo el proceso de desarrollo se debe considerar que el sistema se debe mantener simple, evitando la complejidad innecesaria. Esto se podría lograr teniendo en cuenta solamente las funcionalidades básicas e ir incrementando lo necesario, de esta forma se evita incluir aspectos que agreguen complejidad al sistema. Además, se deben implementar solamente las funcionalidades que realmente se necesitan y no las que se cree que podrían ser necesarias. Otros aspectos a tener en cuenta son: dividir las tareas en sub-tareas entendibles, las cuales deben ser realizadas por una o pocas clases; mantener pequeños los métodos, donde cada uno de estos debería ser responsable de una única tarea; si se tiene muchas condiciones en un método, sería conveniente dividirlo en pequeños métodos; se debe intentar tener pequeñas clases.

- **Tamaño:** se debe evitar que el tamaño del código crezca innecesariamente. Para esto es importante evitar la duplicación, mantener el código simple.

- **Tamaño de unidad:** las unidades como piezas de funcionalidad de más bajo nivel deben mantenerse pequeñas para que sean centradas y fáciles de entender.

En el Anexo D se observan las prácticas que se podrían realizar para incluir los atributos relacionados, en cada uno de los procesos con el fin de potenciar la sub-característica de capacidad de ser probado.

3.2.3. Estructurar el modelo de referencia general.

3.2.3.1. Descripción general del modelo

A continuación se presenta una descripción del modelo de referencia MANTuS. Esta sección contiene la declaración del objetivo del modelo de referencia, el contexto previsto de uso y las acciones que fueron tomadas para alcanzar el consenso.

3.2.3.1.1. Objetivo

El objetivo del modelo de referencia MANTuS es especificar, en términos de sus propositos y sus resultados, un conjunto de procesos que permitan incluir atributos de mantenibilidad al producto software con el fin de potenciar esta característica y que sean aplicables en el contexto de empresas desarrolladoras de software. Al igual que otros modelos de referencia el alcance de éste es la descripción concreta de los procesos y no la especificación detallada de los elementos particulares de la implementación, es decir, la descripción de los procesos establece lo que se debe lograr mas no determina como debe lograrse. Es por esto que los procesos pueden ser implementados de diferentes formas obteniendo el mismo resultado.

3.2.3.1.2. Procesos del modelo

El Modelo de Referencia MANTuS asume que la mantenibilidad de software es un aspecto fundamental del producto que debe ser considerado desde el incio del ciclo de vida del mismo. Es por esto que se establecen los siguentes seis procesos: Análisis de requisitos de software desde la perspectiva de mantenibilidad, Diseño arquitectural de software desde la perspectiva de mantenibilidad, Diseño detallado de software desde la perspectiva de mantenibilidad, Construcción de software desde la perspectiva de mantenibilidad, Pruebas de evaluación de software desde la perspectiva de mantenibilidad, Gestión de la documentación desde la perspectiva de mantenibilidad. En la Tabla 7 se presenta para cada proceso del modelo de referencia su nombre, identificador y propósito.

Proceso	Identificador	Propósito
Análisis de requisitos de software desde la perspectiva de mantenibilidad.	**DEV-MANT.1**	Establecer los requerimientos de los componentes del software teniendo en cuenta atributos que ayuden a potenciar la característica de mantenibilidad.
Diseño arquitectural de software desde la perspectiva de mantenibilidad	**DEV-MANT.2**	Proveer un diseño para el software, que permita incluir atributos de mantenibilidad al producto.
Diseño detallado de software desde la perspectiva de mantenibilidad.	**DEV-MANT.3**	Proporcionar un diseño para el software que sea lo suficientemente detallado que permita incluir atributos de software relacionados con la mantenibilidad del producto.
Construcción de software desde la perspectiva de mantenibilidad.	**DEV-MANT.4**	Producir unidades de software ejecutables que incluyan atributos de software para potenciar la mantenibilidad del producto.

Pruebas de evaluación de software desde la perspectiva de mantenibilidad.	**DEV-MANT.5**	Que la unidad de prueba incluya atributos que favorezcan a la mantenibilidad del producto.
Gestión de la documentación de software desde la perspectiva de mantenibilidad.	**SUP-MANT.1**	Desarrollar y mantener un registro de la información del software teniendo en cuenta atributos que permitan potenciar la mantenibilidad del producto.

Tabla 7. Procesos que componen el modelo de referencia

3.2.3.1.3. Relaciones entre los procesos del modelo

El modelo de referencia que apoya la inclusión de sub-características de mantenibilidad al producto software durante el proceso de desarrollo está compuesto por dos vistas: la vista específica para cada una de las cinco sub-características y la vista general para la característica de mantenibilidad, la cual agrupa las vistas específicas. En la Figura 3 se observa la vista específica del modelo de referencia con el número de prácticas base definidas para cada proceso de las sub-características de mantenibilidad. El modelo de referencia obtenido se basa en la información y análisis realizado en cada una de las actividades anteriores.

Los seis procesos que conforman el modelo de referencia MANTuS deben ser entendidos desde una vista general. Estos procesos no presentan relación entre si, son independientes, es decir, para alcanzar los resultados de un proceso no es necesario haber logrado los resultados de otro u otros procesos, lo cual da flexibilidad al modelo de referencia. En la Figura 4 se presenta la vista general de los procesos que conforman el modelo de referencia, se observa que estos procesos son independientes y que el proceso de Gestión de la documentación desde la perspectiva de mantenibilidad es transversal a los otros, ya que éste es un proceso de soporte que puede ser ejecutado durante los otros cinco procesos. La vista general del modelo de referencia combina y sintetiza las prácticas base definidas en los procesos de todas las sub-características de mantenibilidad.

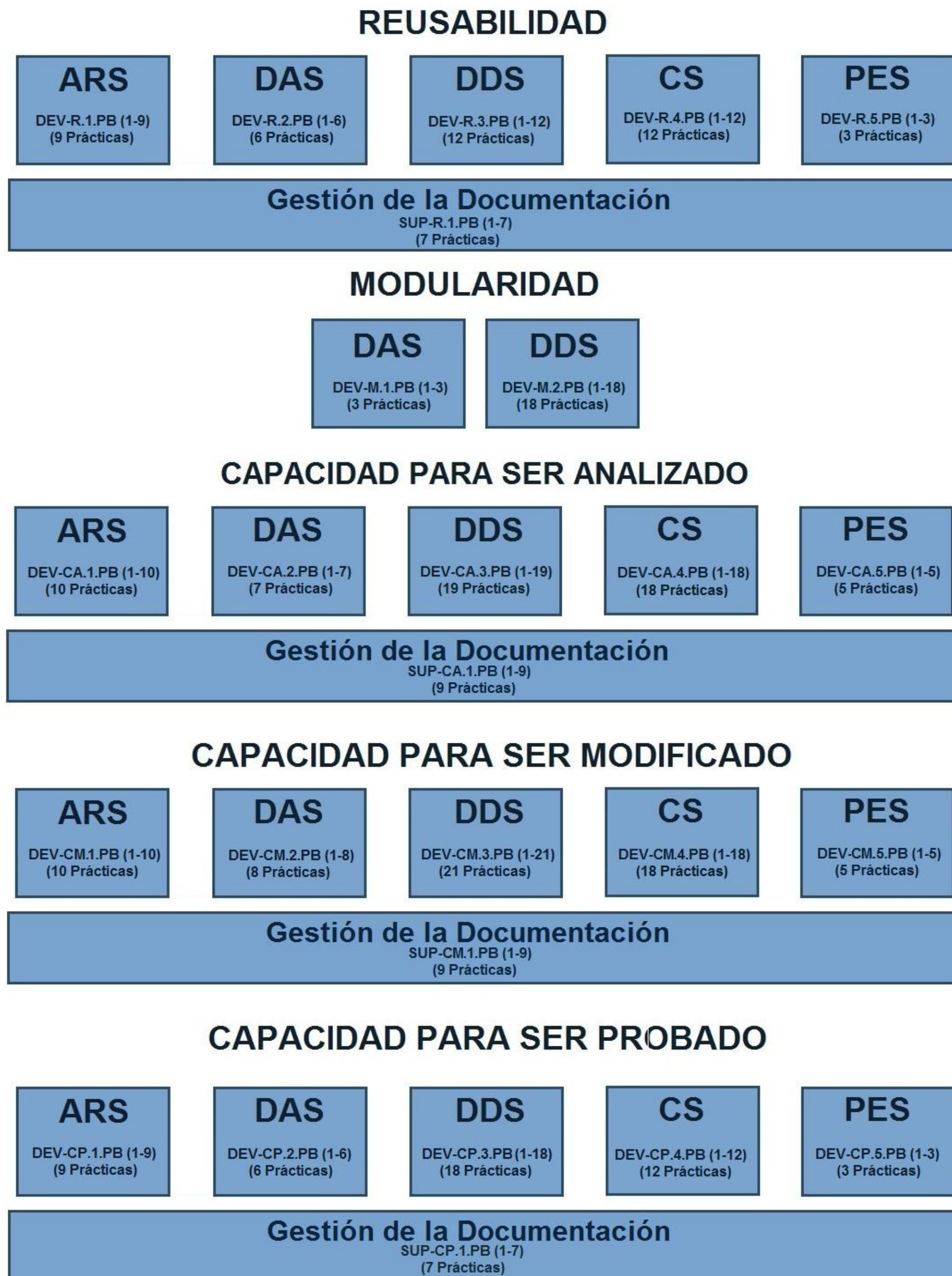

Figura 3. Vista especifica del modelo de referencia. ARS= Análisis de requisitos de software, DAS= diseño arquitectural de software, DDS= diseño detallado de software, CS= construcción de software, PES= pruebas de evaluación de software, OP= operación de software

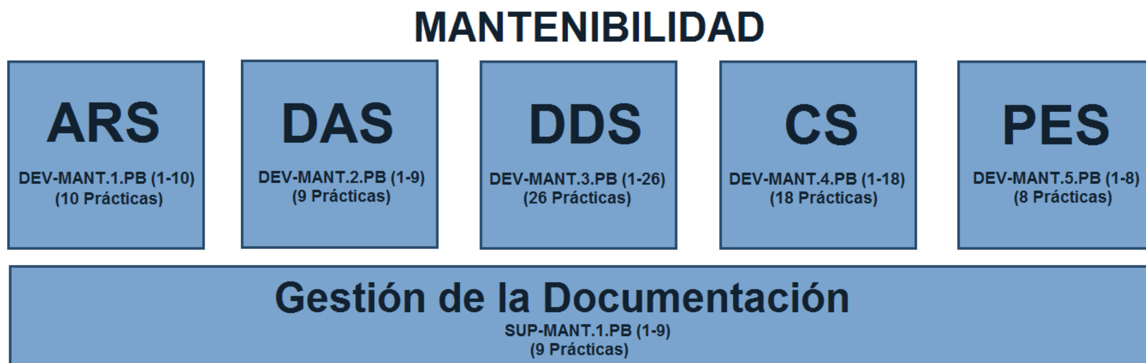

Figura 4. Vista general del modelo de referencia ARS= Análisis de requisitos de software, DAS= diseño arquitectural de software, DDS= diseño detallado de software, CS= construcción de software, PES= pruebas de evaluación de software, OP= operación de software

3.2.3.1.4. Comunidad de interés

El modelo de referencia MANTuS fue construido para ser aplicado en empresas desarrolladoras de software que deseen potenciar la mantenibilidad de sus productos, ademas empresas que esten interesadas en obtener la certificación de mantenibilidad aplicada por AQC.

3.2.3.1.5. Contexto previsto de uso

El modelo de referencia MANTuS fue construido para ser usado en contextos de desarrollo de software donde sea importante incluir al producto software atributos de mantenibilidad para potenciar esta característica. Asi mismo, la especificacion de los resultados permite determinar aspectos a incluir o mejorar en la implementación de los procesos existentes.

3.2.3.1.6. Búsqueda de consenso

La busqueda del consenso del modelo de referencia MANTuS está fuera del alcance de este trabajo de grado ya que ésta involucra socializar dicho modelo en diferentes contextos nacionales e internacionales para llegar a un acuerdo entre los miembros de la comunidad de interés. Sin embargo, se buscó mediante la realización de un estudio de caso la evaluación de un conjunto de prácticas definidas para el proceso de construcción de software, la cual permitió evidenciar la utilidad que tiene el modelo de referencia MANTuS para la comunidad interesada en la mantenibilidad del producto software. Como trabajo futuro se pretende distribuir el modelo de referencia a diferentes actores que están relacionados con el tema de mantenibilidad, como las empresas AQC y Fiding, esto con el fin de empezar a lograr el consenso del mismo.

3.2.3.2. Descripción detallada de los procesos

En esta sección se presentan las descripciones de los procesos del modelo de referencia MANTuS, en términos de sus propósitos y resultados. Los enunciados de los propósitos y resultados de los procesos son elementos obligatorios según el estandar ISO/IEC 33004.

La descripción de los procesos presentada en esta sección sigue la estructura utilizada para plantear los proceso en el estandar internacional ISO/IEC 15504-5. Estas descripciones incorporan el enunciado del propósito del proceso y el conjunto de resultados del proceso. El enunciado del propósito describe a un alto nivel el objetivo general de llevar a cabo el proceso. El conjunto de resultados del proceso describe los elementos con los cuales se demuestra el logro exitoso de su propósito, como la producción de un artefacto, un cambio significativo de estado o el cumplimiento de restricciones especificadas.

Teniendo en cuenta las prácticas base definidas en la sección 3.2.2 para cada una de las cinco sub-características de mantenibilidad, se obtuvo la estructura general del modelo de referencia para la característica de mantenibilidad, en la cual cada proceso está definido en términos de identificador, nombre, propósito, resultados y prácticas base. En la sección resultados del proceso cada resultado está asociado a las sub-características de mantenibilidad en las que influye, siendo CA= capacidad para ser analizado; M=modularidad; CM=capacidad para ser modificado; CP=capacidad para ser probado; R=reusabilidad. En la sección prácticas base se proporciona la definición de las actividades y las tareas necesarias para alcanzar el propósito del proceso y cumplir con los resultados del mismo, cada práctica base está asociada a uno o más resultados. Por último cada procesos está relacionado con los productos de trabajo de salida (PTS) correspondientes, cuyas características se encuentran la Tabla 8. A continuación se presenta la descripción de los seis procesos del modelo de referencia.

DEV-MANT.1 Análisis de requisitos de software desde la perspectiva de mantenibilidad

<table>
<tr><td>Identificador del proceso</td><td>DEV-MANT.1</td></tr>
<tr><td>Nombre del proceso</td><td>Análisis de requisitos de software desde la perspectiva de mantenibilidad</td></tr>
<tr><td>Propósito del proceso</td><td>El propósito del proceso de análisis de requisitos de software desde la perspectiva de mantenibilidad es establecer los requerimientos de los componentes del software teniendo en cuenta atributos que ayuden a potenciar la característica de mantenibilidad.</td></tr>
<tr><td>Resultados del proceso</td><td>Como resultado de la implementación exitosa del proceso de análisis de requisitos de software desde la perspectiva de mantenibilidad:
<table>
<tr><th></th><th>CA</th><th>M</th><th>CM</th><th>CP</th><th>R</th></tr>
<tr><td>a) Los requisitos de software son especificados con simplicidad.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>b) Los requisitos de software son especificados con baja complejidad.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>c) Los requisitos de software especificados son consistentes.</td><td>X</td><td></td><td>X</td><td></td><td></td></tr>
<tr><td>d) El documento de requisitos de software es simple.</td><td>X</td><td></td><td>X</td><td>X</td><td>X</td></tr>
</table>
</td></tr>
<tr><td>Prácticas base</td><td>DEV-MANT.1.PB1: Determinar los requisitos de software</td></tr>
</table>

	necesarios. Incluir solamente los requisitos necesarios, es decir, aquellos que son indispensables para el buen funcionamiento del sistema, evitando así agregar complejidad al mismo. [Resultados: a, b] **DEV-MANT.1.PB2: Asegurar la comprensión.** Expresar la información suficiente para la comprensión de los requisitos, sin agregar datos innecesarios. [Resultados: a, b] **DEV- MANT.1.PB3: Garantizar precisión.** Enunciar en los requisitos solamente lo que el sistema debe hacer, no como debe hacerlo. [Resultados: a, b] **DEV-MANT.1.PB4: Garantizar consistencia.** Verificar que los requisitos sean consistentes entre sí, es decir, no debe presentarse conflictos o incompatibilidad entre ellos. [Resultados: c] **DEV-MANT.1.PB5: Numerar requisitos.** Identificar los requisitos de manera única, haciendo uso de un sistema de numeración. [Resultados: d] **DEV-MANT.1.PB6: Garantizar claridad de requisitos.** Expresar los requisitos de manera sencilla y sin ambigüedades. [Resultados: d] **DEV-MANT.1.PB7: Expresar requisitos cuantitativamente.** Expresar los requisitos no funcionales de manera cuantitativa, por ejemplo no decir que algo debe ser rápido, sino qué tan rápido debe ser. [Resultados: d] **DEV-MANT.1.PB8: Garantizar orden en los requisitos.** Escribir los requisitos de manera organizada, con el fin de poder hacer un seguimiento a los mismos. [Resultados: d] **DEV-MANT.1.PB9: Formular requisitos concisos.** Redactar los requisitos en un lenguaje simple, sencillo y preciso para facilitar la comprensión de todos los participantes del proyecto. [Resultados: d] **DEV-MANT.1.PB10: Utilizar términos conocidos.** Usar términos del dominio del problema con los que los clientes y usuarios estén familiarizados, los cuales deben ser documentados en el glosario de términos del documento de especificación de requisitos de software. [Resultados: d]
Producto de trabajo de salida	
PTS-01 Especificación de requisitos que considera la mantenibilidad [Resultados: a, b, c, d]	

DEV-MANT.2 Diseño arquitectural de software desde la perspectiva de mantenibilidad

<table>
<tr><td>Identificador del proceso</td><td>DEV-MANT.2</td></tr>
<tr><td>Nombre del proceso</td><td>Diseño arquitectural de software desde la perspectiva de mantenibilidad</td></tr>
<tr><td>Propósito del proceso</td><td>El propósito del proceso de diseño arquitectural de software desde la perspectiva de mantenibilidad es proveer un diseño para el software, que permita incluir atributos de mantenibilidad al producto.</td></tr>
<tr><td>Resultados del proceso</td><td>Como resultado de la implementación exitosa del proceso de diseño arquitectural de software desde la perspectiva de mantenibilidad:
<table>
<tr><th></th><th>CA</th><th>M</th><th>CM</th><th>CP</th><th>R</th></tr>
<tr><td>a) El diseño arquitectural de software desarrollado cuenta con bajo acoplamiento.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>b) El diseño arquitectural de software desarrollado tiene baja complejidad.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>c) El diseño arquitectural de software elaborado es fácil de entender.</td><td>X</td><td></td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>d) El diseño arquitectural de software desarrollado es simple.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>e) El diseño arquitectural de software obtenido es consistente.</td><td>X</td><td></td><td>X</td><td></td><td></td></tr>
<tr><td>f) El diseño arquitectural de software obtenido cuenta con trazabilidad.</td><td>X</td><td></td><td>X</td><td></td><td></td></tr>
</table>
</td></tr>
<tr><td>Prácticas base</td><td>DEV-MANT.2.PB1: Descomponer el sistema. Descomponer el sistema en pequeña partes de funcionalidades (módulos). [Resultados: a, b, c, d]
DEV-MANT.2.PB2: Verificar la interrelación. Crear módulos (paquetes, clases, métodos) que tengan la menor interrelación posible. [Resultados: a, b, c, d]
DEV-MANT.2.PB3: Abstraer funcionalidades. Abstraer la funcionalidad de los módulos haciendo uso de clases y métodos abstractos. [Resultados: a, b, c, d]
DEV-MANT.2.PB4: Garantizar claridad. Elaborar diagramas de diseño de alto nivel claros y separados. [Resultados: c]
DEV-MANT.2.PB5: Fraccionar tareas. Dividir las tareas en sub-tareas entendibles. [Resultados: b, d]
DEV-MANT.2.PB6: Asegurar la simplicidad. Realizar</td></tr>
</table>

	funcionalidades con una o pocas clases. [Resultados: b, d] **DEV- MANT.2.PB7: Asegurar baja complejidad.** Tener paquetes lo más simples posible. [Resultados: b, d] **DEV-MANT.2.PB8: Verificar trazabilidad.** Verificar periódicamente que los artefactos de una etapa sean consistentes con artefactos de la etapa anterior. [Resultados: c, e, f] **DEV-MANT.2.PB9: Actualizar artefactos.** Actualizar los artefactos cuando se presenten cambios. [Resultados: c, e, f]
Producto de trabajo de salida	
PTS-02 Diseño de software de alto nivel que considera la mantenibilidad [Resultados: a, b, c, d, e, f]	
PTS-03 Arquitectura que favorece a la mantenibilidad [Resultados: a, b, d]	
PTS-04 Registro de trazabilidad [Resultados: e, f]	

DEV-MANT.3 Diseño detallado de software desde la perspectiva de mantenibilidad

Identificador del proceso	DEV-MANT.3
Nombre del proceso	Diseño detallado de software desde la perspectiva de mantenibilidad.
Propósito del proceso	El propósito del proceso de diseño detallado de software desde la perspectiva de mantenibilidad es proporcionar un diseño para el software que sea lo suficientemente detallado que permita incluir atributos de software relacionados con la mantenibilidad del producto.

Resultados del proceso

Como resultado de la implementación exitosa del proceso de diseño detallado de software desde la perspectiva de mantenibilidad:

	CA	M	CM	CP	R
a) El diseño detallado de software elaborado cuenta con bajo acoplamiento.	X	X	X	X	X
b) El diseño detallado de software tiene baja complejidad.	X	X	X	X	X
c) El diseño detallado de software es fácil de entender.	X		X	X	X
d) El mecanismo de herencia es utilizado correctamente en el diseño detallado obtenido.	X		X	X	X
e) El polimorfismo es aplicado adecuadamente en el diseño detallado realizado.	X		X		X
f) El diseño detallado de software es simple.	X	X	X	X	X
g) El diseño detallado de software tiene baja cohesión.	X	X	X	X	X
h) El diseño detallado de software cuenta con bajo nivel de anidación.	X		X	X	
i) El diseño detallado de software tiene en cuenta el encapsulamiento.	X	X	X	X	
j) El diseño detallado de software es consistente.	X		X		
k) El diseño detallado de software cuenta con trazabilidad.	X		X		

Prácticas base

DEV-MANT.3.PB1: Reducir respuestas. Reducir el número de respuestas por clases, incluyendo únicamente las necesarias. [Resultados: a, b, c, f]

DEV-MANT.3.PB2: Separar funcionalidades. Cada módulo (paquete) debe tener una única preocupación, lo cual implica que las clases en el módulo estén relacionadas de acuerdo a su función. [Resultados: b, c, f, g]

DEV-MANT.3.PB3: Garantizar la cohesión. Eliminar las funcionalidades que no pertenecen a la actividad realizada por un módulo (paquete), creando otro módulo para estas. [Resultados: b, c, f, g]

DEV-MANT.3.PB4: Disminuir métodos. Reducir el número de métodos por clase, incluyendo únicamente los necesarios. [Resultados: b, f]

	DEV-MANT.3.PB5: Verificar los flujos de datos. Incluir solamente los datos de entrada y salida necesarios en las funciones. [Resultados: b, f] **DEV-MANT.3.PB6: Garantizar claridad.** Elaborar diagramas de diseño de bajo nivel claros y separados. [Resultados: c] **DEV-MANT.3.PB7: Controlar niveles de herencia.** Tener como máximo 3 niveles de profundidad de herencia. [Resultados: b, c, d, f, h] **DEV-MANT.3.PB8: Utilizar polimorfismo.** Cuando se tiene que llevar a cabo una responsabilidad que dependa de un tipo, asignar el mismo nombre a servicios implementados en diferentes objetos, para esto se debe hacer uso de las clases abstractas [Resultados: b, e, f] **DEV-MANT.3.PB0: Determinar funcionalidades.** Tener en cuenta solamente las funcionalidades necesarias. [Resultados: b, f] **DEV-MANT.3.PB10: Fraccionar tareas.** Dividir las tareas en sub-tareas entendibles. [Resultados: b, f] **DEV-MANT.3.PB11: Asegurar simplicidad.** Realizar funcionalidades con una o pocas clases. [Resultados: b, f] **DEV-MANT.3.PB12: Separar responsabilidades.** Cada método deber ser responsable de una sola tarea. [Resultados: b, f] **DEV-MANT.3.PB13: Verificar el tamaño de las clases.** Las clases diseñadas deben incluir solamente lo necesario para cumplir su responsabilidad. [Resultados: b, f] **DEV-MANT.3.PB14: Identificar servicios.** Identificar los servicios que el paquete proporcionará al sistema. [Resultados: a, b, f, i] **DEV-MANT.3.PB15: Determinar detalles de implementación.** Identificar el conjunto de detalles de implementación que podrán cambiar durante el desarrollo, es decir, potenciar el ocultamiento de información. [Resultados: a, b, f, i] NOTA: hacer privadas las variables que son innecesarias para el tratamiento del objeto pero necesarias para su funcionamiento, así como las funciones que no necesitan interacción del usuario o que solo pueden ser llamadas por otras funciones dentro del objeto. **DEV-MANT.3.PB16: Encapsular detalles de implementación.** Encapsular en un paquete independiente, cada uno de los detalles de implementación, buscando que los cambios del sistema se limiten al paquete en cuestión. [Resultados: a, b, f, i]

	DEV-MANT.3.PB17: Identificar responsabilidades. Identificar las responsabilidades que la clase tiene en el sistema. [Resultados: a, b, f, i] **DEV-MANT.3.PB18: Encapsular métodos.** Mantener información y métodos privados que pueden ser cambiados en cualquier momento sin afectar a los otros objetos que dependan de ello. [Resultados: a, b, f, i] **DEV-MANT.3.PB19: Verificar variables globales.** Evitar el uso de variables globales. [Resultados: a, b, f, i] **DEV-MANT.3.PB20: Verificar atributos públicos.** Evitar el uso de atributos públicos**.** [Resultados: a, b, f, i] **DEV-MANT.3.PB21: Reducir parámetros por referencia.** Disminuir al mínimo los parámetros por referencia, incluyendo solo los inevitables. [Resultados: a, b, f] **DEV-MANT.3.PB22: Moderar dependencias.** Reducir dependencia entre clases, es decir, deberían existir pocas dependencias hacia otras clases desde una sola. [Resultados: a, b, f] **DEV-MANT.3.PB23: Restringir responsabilidades.** Disminuir las responsabilidades de una clase, buscando dividirla en dos o más si tiene muchas responsabilidades. [Resultados: b, f, g] **DEV-MANT.3.PB24: Asegurar baja complejidad.** Elaborar clases y métodos lo más simples posibles, incluyendo solamente las funcionalidades necesarias. [Resultados: b, f] **DEV-MANT.3.PB25: Verificar trazabilidad.** Verificar periódicamente que los artefactos de una etapa sean consistentes con artefactos de la etapa anterior. [Resultados: c, j, k] **DEV-MANT.3.PB26: Actualizar artefactos.** Actualizar los artefactos cuando se presenten cambios. [Resultados: c, j, k]
Producto de trabajo de salida	
PTS-05 Diseño de software de bajo nivel que considera la mantenibilidad [Resultados: a, b, c, d, e, f]	
PTS-04 Registro de trazabilidad [Resultados: e, f]	

DEV-MANT.4 Construcción de software desde la perspectiva de mantenibilidad

Identificador del proceso	DEV-MANT.4
Nombre del proceso	Construcción de software desde la perspectiva de mantenibilidad.
Propósito del proceso	El propósito del proceso de construcción de software desde la perspectiva de mantenibilidad es producir unidades de software ejecutables que incluyan atributos de software para

	potenciar la mantenibilidad del producto.
Resultados del proceso	Como resultado de la implementación exitosa del proceso de construcción de software desde la perspectiva de mantenibilidad:

	CA	M	CM	CP	R
a) Las unidades de software cuentan con buenos comentarios.	X		X		X
b) Las unidades de software tienen baja complejidad.	X	X	X	X	X
c) Las unidades de software son fáciles de entender.	X		X	X	X
d) Las unidades de software cuentan con alta facilidad de lectura.	X		X	X	X
e) Las unidades de software son simples.	X	X	X	X	X
f) Las unidades de software tienen el tamaño adecuado.	X		X	X	X
g) Las unidades de software cuentan con bajo nivel de anidación.	X		X	X	
h) Las unidades de software tienen un estándar definido.	X		X		
i) Las unidades de software evitan la duplicación.	X		X	X	
j) Las unidades de software son consistentes.	X		X		
k) Las unidades de software cuentan con trazabilidad.	X		X		

Prácticas base	**DEV-MANT.4.PB1: Utilizar comentarios.** Hacer uso de comentarios adecuados en el código fuente. [Resultados: a, c , d] **DEV-MANT.4.PB2: Describir propósitos.** Explicar el propósito de las funciones, subrutinas, variables y constantes. [Resultados: a, c, d] **DEV-MANT.4.PB3: Verificar los flujos de control.** Incluir solamente los flujos de control necesarios. [Resultados: b, c, e, f] **DEV-MANT 4.PB4: Organizar código.** El código fuente debe ser indentado. [Resultado: c, d] **DEV-MANT.4.PB5: Asignar nombres claros.** Dar nombres descriptivos a las variables. [Resultado: c, d] **DEV-MANT.4.PB6: Controlar abreviaciones.** Hacer poco uso de abreviaciones. [Resultado: c, d] **DEV-MANT.4.PB7: Controlar sentencias.** Evitar sentencias largas en el código fuente, es mejor mantener las líneas de

	código cortas, aunque esto implica separar las sentencias en múltiples líneas. [Resultado: c, d] **DEV-MANT.4.PB8: Controlar paréntesis.** Hacer correcto uso de los paréntesis para mejorar la facilidad de lectura de las expresiones aritméticas y lógicas. [Resultado: c, d] **DEV-MANT.4.PB9: Determinar funcionalidades.** Implementar solamente las funcionalidades necesarias. [Resultados: b, c, e, f] **DEV-MANT.4.PB10: Dividir métodos.** Mantener métodos simples, es decir, dividir los métodos que tengan muchas condiciones. [Resultados: b, c, e, f] **DEV-MANT.4.PB11: Controlar tamaño.** Evitar que el tamaño del código crezca innecesariamente. [Resultados: c, f] **DEV-MANT.4.PB12: Controlar nivel de anidación.** Evitar las estructuras de control anidadas innecesarias, ya que estas son más complejas que las estructuras de control secuenciales. [Resultados: b, c, e, f, g] **DEV-MANT.4.PB13: Evitar duplicación.** Detectar código duplicado, extrayéndolo en un nuevo procedimiento y reemplazando todas las instancias del código duplicado por llamadas al nuevo procedimiento. [Resultados: b, e, f, i] **DEV-MANT.4.PB14: Definir estándares.** Tener un conjunto de estándares de programación en la escritura de código para evitar la individualidad entre los programadores. [Resultados: d, h] **DEV-MANT.4.PB15: Utilizar convenciones.** Hacer uso de convenciones estándar para el nombrado de paquetes, clases, métodos y variables. [Resultados: d, h] **DEV-MANT.4.PB16: Controlar tamaño de unidad.** Las piezas de funcionalidad de más bajo nivel deben mantenerse pequeñas para que sean centradas y fáciles de entender. [Resultados: b, c, e, f] **DEV-MANT.4.PB17: Verificar trazabilidad.** Verificar periódicamente que los artefactos de una etapa sean consistentes con artefactos de la etapa anterior. [Resultados: c, j, k] **DEV-MANT.4.PB18: Actualizar artefactos.** Actualizar los artefactos cuando se presenten cambios. [Resultados: c, j, k]
Producto de trabajo de salida	
PTS-06 Unidad de software que considera la mantenibilidad [Resultados: a, b, c, d, e, f, g, h, i, j, k]	
PTS-04 Registro de trazabilidad [Resultados: j, k]	

DEV-MANT.5 Pruebas de evaluación de software desde la perspectiva de mantenibilidad.

<table>
<tr><td>Identificador del proceso</td><td>DEV-MANT.5</td></tr>
<tr><td>Nombre del proceso</td><td>Pruebas de evaluación de software desde la perspectiva de mantenibilidad.</td></tr>
<tr><td>Propósito del proceso</td><td>El propósito del proceso de pruebas de evaluación de software desde la perspectiva de mantenibilidad es que la unidad de prueba incluya atributos que favorezcan a la mantenibilidad del producto.</td></tr>
<tr><td>Resultados del proceso</td><td>Como resultado de la implementación exitosa del proceso de pruebas de evaluación de software desde la perspectiva de mantenibilidad:
<table>
<tr><th></th><th>CA</th><th>M</th><th>CM</th><th>CP</th><th>R</th></tr>
<tr><td>a) La unidad de prueba es consistente.</td><td>X</td><td></td><td>X</td><td></td><td></td></tr>
<tr><td>b) La unidad de prueba cumple con un estándar.</td><td>X</td><td></td><td>X</td><td></td><td></td></tr>
<tr><td>c) La unidad de prueba es fácil de entender.</td><td>X</td><td></td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>d) La unidad de prueba es fácil de leer.</td><td>X</td><td></td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>e) La unidad de prueba cuenta con trazabilidad.</td><td>X</td><td></td><td>X</td><td></td><td></td></tr>
<tr><td>f) El documento de pruebas es simple.</td><td>X</td><td></td><td>X</td><td>X</td><td>X</td></tr>
</table></td></tr>
<tr><td>Prácticas base</td><td>DEV-MANT.5.PB1: Garantizar trazabilidad. Hacer seguimiento de la unidad de prueba hasta los requisitos. [Resultados: a, e]
DEV-MANT.5.PB2: Utilizar convenciones de nombrado. Hacer uso de convenciones de nombrado para los casos de prueba, ya que esto ayuda a evitar comentarios. [Resultados: b, c, d]
DEV-MANT.5.PB3: Resumir pruebas. Expresar el resumen de las pruebas que han sido realizadas. [Resultados: f]
DEV-MANT.5.PB4: Indicar módulos probados. Escribir cuales módulos (paquetes, clases, métodos) se han probado y si se han probado a fondo. [Resultados: f]
DEV-MANT.5.PB6: Indicar resultados claros. Indicar claramente los errores encontrados. [Resultados: f]</td></tr>
<tr><th colspan="2">Producto de trabajo de salida</th></tr>
<tr><td colspan="2">PTS-07 Especificación de los casos de pruebas [Resultados: b, c, d]</td></tr>
<tr><td colspan="2">PTS-08 Registro de prueba [Resultados: b, c, d, f]</td></tr>
<tr><td colspan="2">PTS-04 Registro de trazabilidad [Resultados: a, e]</td></tr>
</table>

SUP-MANT.1 Gestión de la documentación desde la perspectiva de mantenibilidad

<table>
<tr><td>Identificador del proceso</td><td>SUP-MANT.1</td></tr>
<tr><td>Nombre del proceso</td><td>Gestión de la documentación de software desde la perspectiva de mantenibilidad.</td></tr>
<tr><td>Propósito del proceso</td><td>El propósito del proceso de gestión de la documentación de software desde la perspectiva de mantenibilidad es desarrollar y mantener un registro de la información del software teniendo en cuenta atributos que permitan potenciar la mantenibilidad del producto.</td></tr>
<tr><td>Resultados del proceso</td><td>Como resultado de la implementación exitosa del proceso de gestión de la documentación de software desde la perspectiva de mantenibilidad:
<table>
<tr><th></th><th>CA</th><th>M</th><th>CM</th><th>CP</th><th>R</th></tr>
<tr><td>a) Los documentos de análisis de requisitos son simples.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>b) Los documentos de diseño arquitectural de software son simples.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>c) Los documentos de diseño detallado de software son simples.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>d) Los documentos de pruebas evaluación de software son simples.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>e) Los documentos de operación de software son simples.</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>f) Los documentos son fáciles de entender.</td><td>X</td><td></td><td>X</td><td>X</td><td>X</td></tr>
<tr><td>g) Los documentos son consistentes.</td><td>X</td><td></td><td>X</td><td></td><td></td></tr>
<tr><td>h) Los documentos se adecuan a un estándar.</td><td>X</td><td></td><td>X</td><td></td><td></td></tr>
<tr><td>i) Los documentos son fáciles de leer.</td><td>X</td><td></td><td>X</td><td>X</td><td>X</td></tr>
</table>
</td></tr>
<tr><td>Prácticas base</td><td>SUP-MANT.1.PB1: Mantener documentación. Mantener actualizados y organizados los documentos. [Resultados: a, b, c, d, e, f, g, h, i]
SUP-MANT.1.PB2: Definir estándar. Cumplir con un estándar de documentación adecuado. [Resultados: a, b, c, d, e, f, g, h, i]
SUP-MANT.1.PB3: Asignar nombres claros. Titular los documentos y diagramas con claridad. [Resultados: a, b, c, d, e, f, g, h, i]
SUP-MANT.1.PB4: Asegurar precisión. Indicar las secciones de los documentos con precisión. [Resultados: a, b, c, d, e, f, g, h, i]</td></tr>
</table>

	SUP-MANT.1.PB5: Definir convenciones. Usar símbolos convencionales en todos los diagramas. [Resultados: b, c, f, g, h, i] **SUP-MANT.1.PB6: Realizar manuales.** Elaborar manuales de administración y de usuario de forma clara y concisa. [Resultados: e] **SUP-MANT.1.PB7: Elaborar documentos de soporte.** Elaborar documentos de instalación, configuración, integración y migración. [Resultados: e] **SUP-MANT.1.PB8: Utilizar estándar.** Verificar que los artefactos asocien y contengan notación, terminología y simbología uniforme. [Resultados: b, c, f, g, h, i] **SUP-MANT.1.PB9: Usar convenciones.** Hacer uso de convenciones de nombrado en los documentos. [Resultados: b, c, f, g, h, i]
Productos de trabajo de Salida	
PTS-09 Historial de cambio [Resultados: a, b, c, d, e, f, g, h, i]	
PTS-10 Plantilla [Resultados: a, b, c, d, e, f, g, h, i]	

Características de productos de trabajo

ID PT	Nombre de PTS	Características de PTS
PTS-01	Especificación de requisitos que considera la mantenibilidad	- Cada requerimiento es identificado - Cada requerimiento es único - Cada requerimiento es verificable o puede ser evaluado - Los requerimientos no son ambiguos - Los requisitos no deben presentar conflictos entre ellos.
PTS-02	Diseño de software de alto nivel que considera la mantenibilidad	- Describe la estructura general del software - Identifica los elementos del software requeridos - Identifica la relación entre los elementos del software - La relación entre los elementos del software debe ser la mínima posible - Consideraciones para los diagramas de alto nivel: - - Deben ser claros y separados - - Deben ser actualizados cuando se presenten cambios
PTS-03	Arquitectura que favorece a la mantenibilidad	- Proporciona una vista general del sistema - Incluye los patrones de arquitectura necesarios de acuerdo al problema - Utilizar una arquitectura que favorezca a la mantenibilidad. es conveniente utilizar una arquitectura como la tres capas, ya

		que brinda una clara separación de preocupaciones entre la capa de presentación, negocio y persistencia, lo cual es considerado una buena práctica. Cada capa debe mantenerse desacoplada de las capas por encima. Es recomendable verificar que las capas dependan solo de los componentes más generales en las capas inferiores
PTS-04	Registro de trazabilidad	- Identifica los requerimientos a hacer trazados - Contiene un mapeo de los requerimientos a los productos de trabajo del ciclo de vida - Provee un mapeo hacia adelante y hacia atrás de los requerimientos hacia productos de trabajo asociados a través de todas las fases del ciclo de vida - Debe mantenerse actualizado - NOTA: podría incluirse como una sección de un producto de trabajo definido
PTS-05	Diseño de software de bajo nivel que considera la mantenibilidad	- Proporciona un diseño detallado - Proporciona una descripción detallada de cada una de las partes de la solución - Proporciona un diseño completo y listo para ser programado - Incluye los patrones de diseño necesarios de acuerdo al problema - Consideraciones para los diagramas de bajo nivel: - - Deben ser claros y separados - - Deben ser actualizados cuando se presenten cambios
PTS-06	Unidad de software que considera la mantenibilidad	- Sigue un estándar de codificación establecido - El código es documentado - El código es indentado - El código no contiene duplicación - El código se debe mantener actualizado
PTS-07	Especificación de los casos de prueba	- Identifica el caso de prueba - Especificaciones de entrada - Especificaciones de salida - Utiliza convenciones de nombrado
PTS-08	Registro de prueba	- Registra los resultados de las pruebas realizadas al producto. - Identifica los elementos que fueron probados - Identifica la fecha en la que las pruebas fueron ejecutadas - Identifica la persona responsable de los resultados de la prueba

PTS-09	Historial de cambio	- Registros históricos de todos los cambios realizados a un producto de trabajo - Contiene: - - Descripción del cambio - -Información de la versión del producto de trabajo cambiado - - Fecha del cambio - -Persona encargada del cambio
PTS-10	Plantilla	- Define el estilo y forma esperados para los documentos - Define un estándar de documentación adecuado: - - Títulos claros - - Secciones - - Convenciones de nombrado - Define convenciones para los diagramas: - - Notación uniforme - - Terminología uniforme - - Simbología uniforme

Tabla 8. Características de productos de trabajo de salida

3.2.4. Plantear los modelos de referencia específicos.

Para las cinco sub-características de mantenibilidad se elaboraron las descripciones de proceso, siguiendo la misma estructura utilizada en la descripción del modelo de referencia general. A continuación se presenta la descripción de procesos para la sub-caracteristica reusabilidad. Las descripciones de procesos para las sub-características capacidad para ser analizado, modularidad, capacidad para ser modificado, capacidad para ser probado se encuentran en los Anexos A, B, C y D respectivamente.

DEV-R.1 Análisis de requisitos de software desde la perspectiva de reusabilidad

Identificador del proceso	DEV-R.1
Nombre del proceso	Análisis de requisitos de software desde la perspectiva de reusabilidad.
Propósito del proceso	El propósito del proceso de análisis de requisitos de software desde la perspectiva de reusabilidad es establecer los requerimientos de los componentes del software teniendo en cuenta atributos que ayuden a potenciar la sub-característica de reusabilidad.
Resultados del proceso	Como resultado de la implementación exitosa del proceso de análisis de requisitos de software desde la perspectiva de reusabilidad: a) Los requisitos de software son especificados con simplicidad. b) Los requisitos de software son especificados con baja complejidad. c) El documento de requisitos de software es simple.
Prácticas base	**DEV-R.1.PB1: Determinar los requisitos de software**

	necesarios. Incluir solamente los requisitos necesarios, es decir, aquellos que son indispensables para el buen funcionamiento del sistema, evitando así agregar complejidad al mismo. [Resultados: a, b] **DEV-R.1.PB2: Asegurar la comprensión.** Expresar la información suficiente para la comprensión de los requisitos, sin agregar datos innecesarios. [Resultados: a, b] **DEV-R.1.PB3: Garantizar precisión.** Enunciar en los requisitos solamente lo que el sistema debe hacer, no como debe hacerlo. [Resultados: a, b] **DEV-R.1.PB4: Numerar requisitos.** Identificar los requisitos de manera única, haciendo uso de un sistema de numeración. [Resultados: c] **DEV-R.1.PB5: Garantizar claridad de requisitos.** Expresar los requisitos de manera sencilla y sin ambigüedades. [Resultados: c] **DEV-R.1.PB6: Expresar requisitos cuantitativamente.** Expresar los requisitos no funcionales de manera cuantitativa, por ejemplo no decir que algo debe ser rápido, sino qué tan rápido debe ser. [Resultados: c] **DEV-R.1.PB7: Garantizar orden en los requisitos.** Escribir los requisitos de manera organizada, con el fin de poder hacer un seguimiento a los mismos. [Resultados: c] **DEV-R.1.PB8: Formular requisitos concisos.** Redactar los requisitos en un lenguaje simple, sencillo y preciso para facilitar la comprensión de todos los participantes del proyecto. [Resultados: c] **DEV-R.1.PB9: Utilizar términos conocidos.** Usar términos del dominio del problema con los que los clientes y usuarios estén familiarizados, los cuales deben ser documentados en el glosario de términos del documento de especificación de requisitos de software. [Resultados: c]

DEV-R.2 Diseño Arquitectural de Software desde la perspectiva de reusabilidad

Identificador del proceso	DEV-R.2
Nombre del proceso	Diseño arquitectural de software desde la perspectiva de reusabilidad.
Propósito del proceso	El propósito del proceso de diseño arquitectural de software desde la perspectiva de reusabilidades proveer un diseño para el software, que permita incluir atributos de reusabilidad al producto.
Resultados del	Como resultado de la implementación exitosa del proceso de

proceso	diseño arquitectural de software desde la perspectiva de reusabilidad: a) El diseño arquitectural de software desarrollado cuenta con bajo acoplamiento. b) El diseño arquitectural de software desarrollado tiene baja complejidad. c) El diseño arquitectural de software elaborado es fácil de entender. d) El diseño arquitectural de software es simple.
Prácticas base	**DEV-R.2.PB1: Descomponer el sistema.** Descomponer el sistema en pequeña partes de funcionalidades (módulos). [Resultados: a, b, c, d] **DEV-R.2.PB2: Verificar la interrelación:** Crear módulos (paquetes, clases, métodos) que tengan la menor interrelación posible. [Resultados: a, b, c, d] **DEV-R.2.PB3: Abstraer funcionalidades.** Abstraer la funcionalidad de los módulos haciendo uso de clases y métodos abstractos. [Resultados: a, b, c, d] **DEV-R.2.PB4: Garantizar claridad.** Elaborar diagramas de diseño de alto nivel claros y separados. [Resultados: c] **DEV-R.2.PB5: Fraccionar tareas.** Dividir las tareas en sub-tareas entendibles. [Resultados: b, d] **DEV-R.2.PB6: Asegurar la simplicidad.** Realizar funcionalidades con una o pocas clases. [Resultados: b, d]

DEV-R.3 Diseño Detallado de Software desde la perspectiva de reusabilidad

Identificador del proceso	DEV-R.3
Nombre del proceso	Diseño detallado de software desde la perspectiva de reusabilidad.
Propósito del proceso	El propósito del proceso de diseño detallado de software desde la perspectiva de reusabilidad es proporcionar un diseño para el software que sea lo suficientemente detallado que permita incluir atributos de software relacionados con la reusabilidad del producto.
Resultados del proceso	Como resultado de la implementación exitosa del proceso de diseño detallado de software desde la perspectiva de reusabilidad: a) El diseño detallado de software elaborado cuenta con bajo acoplamiento. b) El diseño detallado de software tiene baja complejidad. c) El diseño detallado de software es fácil de entender. d) El mecanismo de herencia es utilizado correctamente en el diseño detallado obtenido. e) El polimorfismo es aplicado adecuadamente en el diseño detallado realizado.

	f) El diseño detallado de software es simple. g) El diseño detallado de software tiene alta cohesión.
Prácticas base	**DEV-R.3.PB1: Reducir respuestas.** Reducir el número de respuestas por clases, incluyendo únicamente las necesarias. [Resultados: a, b, c, f, g] **DEV-R.3.PB2: Disminuir métodos.** Reducir el número de métodos por clase, incluyendo únicamente los necesarios. [Resultados: b, f, g] **DEV-R.3.PB3: Verificar los flujos de datos.** Incluir solamente los datos de entrada y salida necesarios en las funciones. [Resultados: b, f, g] **DEV-R.3.PB4: Garantizar claridad.** Elaborar diagramas de diseño de bajo nivel claros y separados. [Resultados: c, g] **DEV-R.3.PB5: Controlar niveles de herencia.** Tener como máximo 3 niveles de profundidad de herencia. [Resultados: b, c, d, f, g] **DEV-R.3.PB6: Utilizar polimorfismo.** Cuando se tiene que llevar a cabo una responsabilidad que dependa de un tipo, asignar el mismo nombre a servicios implementados en diferentes objetos. [Resultados: b, e, f, g] **DEV-R.3.PB7: Determinar funcionalidades.** Tener en cuenta solamente las funcionalidades necesarias. [Resultados: b, f, g] **DEV-R.3.PB8: Fraccionar tareas.** Dividir las tareas en sub-tareas entendibles. [Resultados: b, f, g] **DEV-R.3.PB9: Asegurar simplicidad.** Realizar funcionalidades con una o pocas clases. [Resultados: b, f, g] **DEV-R.3.PB10: Separar responsabilidades.** Cada método deber ser responsable de una sola tarea. [Resultados: b, f, g] **DEV-R.3.PB11: Verificar el tamaño de las clases.** Las clases diseñadas deben incluir solamente lo necesario para cumplir con su responsabilidad. [Resultados: b, f, g] **DEV-R.3.PB12: Separar funcionalidades**. Cada módulo (paquete) debe tener una única preocupación, lo cual implica que las clases en el módulo estén relacionadas de acuerdo a su función. [Resultados: b, c, f, g]

DEV-R.4 Construcción de Software desde la perspectiva de reusabilidad

Identificador del proceso	DEV-R.4
Nombre del proceso	Construcción de software desde la perspectiva de reusabilidad.
Propósito del proceso	El propósito del proceso de construcción de software desde la

	perspectiva de reusabilidad es producir unidades de software ejecutables que incluyan atributos de software para potenciar la reusabilidad.
Resultados del proceso	Como resultado de la implementación exitosa del proceso de construcción de software desde la perspectiva de reusabilidad: a) Las unidades de software cuentan con buenos comentarios. b) Las unidades de software tienen baja complejidad. c) Las unidades de software son fáciles de entender. d) Las unidades de software cuentan con alta facilidad de lectura. e) Las unidades de software son simples. f) Las unidades de software tienen el tamaño adecuado.
Prácticas base	**DEV-R.4.PB1: Utilizar comentarios.** Hacer uso de comentarios adecuados en el código fuente. [Resultados: a, c, d] **DEV-R.4.PB2: Describir propósitos.** Explicar el propósito de las funciones, subrutinas, variables y constantes. [Resultados: a, c] **DEV-R.4.PB3: Verificar los flujos de control.** Incluir solamente los flujos de control necesarios. [Resultados: b, c, f] **DEV-R.4.PB4: Organizar código.** El código fuente debe ser indentado. [Resultado: c, d] **DEV-R.4.PB5: Asignar nombres claros.** Dar nombres descriptivos a las variables. [Resultado: c, d] **DEV-R.4.PB6: Controlar abreviaciones.** Hacer poco uso de abreviaciones. [Resultado: c, d] **DEV-R.4.PB7: Controlar sentencias.** Evitar sentencias largas en el código fuente. [Resultado: c, d] **DEV-R.4.PB8: Controlar paréntesis.** Hacer correcto uso de los paréntesis. [Resultado: c, d] **DEV-R.4.PB9: Determinar funcionalidades.** Implementar solamente las funcionalidades necesarias. [Resultados: b, c, e] **DEV-R.4.PB10: Dividir métodos.** Mantener métodos simples, es decir, dividir los métodos que tengan muchas condiciones. [Resultados: b, c, e, f] **DEV-R.4.PB11: Controlar tamaño.** Evitar que el tamaño del código crezca innecesariamente. [Resultados: c, f] **DEV-R.4.PB12: Controlar duplicación.** Evitar la duplicación de código. [Resultados: c, f]

DEV-R.5 Pruebas de evaluación de software desde la perspectiva de reusabilidad

Identificador del proceso	DEV-R.5
Nombre del proceso	Pruebas de evaluación de software desde la perspectiva de reusabilidad.
Propósito del proceso	El propósito del proceso de pruebas de evaluación de software desde la perspectiva de reusabilidad es que la unidad de prueba incluya atributos que favorezcan a la reusabilidad del producto.
Resultados del proceso	Como resultado de la implementación exitosa del proceso de pruebas de evaluación de software desde la perspectiva de reusabilidad: a) El documento de pruebas de software es simple.
Prácticas base	**DEV-R.1.PB1: Resumir pruebas.** Expresar el resumen de las pruebas que han sido realizadas. [Resultados: f] **DEV-R.1.PB2: Indicar módulos probados.** Escribir cuales módulos (paquetes, clases, métodos) se han probado y si se han probado a fondo. [Resultados: f] **DEV-R.1.PB3: Indicar resultados claros.** Indicar claramente los errores encontrados. [Resultados: f]

SUP-R.1 Gestión de la documentación de software desde la perspectiva de reusabilidad

Identificador del proceso	SUP-R.1
Nombre del proceso	Gestión de la documentación de software desde la perspectiva de reusabilidad.
Propósito del proceso	El propósito del proceso de gestión de la documentación de software desde la perspectiva de reusabilidad es desarrollar y mantener un registro de la información del software teniendo en cuenta atributos que permitan potenciar la reusabilidad del producto.
Resultados del proceso	Como resultado de la implementación exitosa del proceso de gestión de la documentación de software desde la perspectiva de reusabilidad: a) Los documentos de análisis de requisitos son simples. b) Los documentos de diseño arquitectural de software son simples. c) Los documentos de diseño detallado de software son simples. d) Los documentos de pruebas evaluación de software son simples. e) Los documentos de operación de software son simples. f) Los documentos son fáciles de entender.
Prácticas base	**SUP-R.1.PB1: Mantener documentación.** Mantener

	actualizados y organizados los documentos. (Resultados: a, b, c, d, e, f) **SUP-R.1.PB2: Definir estándar.** Cumplir con un estándar adecuado. (Resultados: a, b, c, d, e, f) **SUP-R.1.PB3: Asignar nombres claros.** Titular los documentos y diagramas con claridad. (Resultados: a, b, c, d, e, f) **SUP-R.1.PB4: Asegurar precisión.** Indicar las secciones de los documentos con precisión. (Resultados: a, b, c, d, e, f) **SUP-R.1.PB5: Definir convenciones.** Usar símbolos convencionales en todos los diagramas. (Resultados: b, c, f) **SUP-R.1.PB6: Realizar manuales.** Elaborar manuales de administración y de usuario de forma clara y concisa. (Resultados: e) **SUP-R.1.PB7: Elaborar documentos de soporte.** Elaborar documentos de instalación, configuración, integración y migración. (Resultados: e)

CAPITULO IV

APLICACIÓN DEL ESTUDIO DE CASO

La evaluación del modelo de referencia MANTuS se realizó mediante la aplicación de un estudio de caso con estudiantes del programa de Ingeniería de sistemas de la Universidad del Cauca. Para la ejecución de este estudio de caso se tomó como referencia la plantilla protocolo para planeación de estudios de caso planteado en [93]. A continuación se describe cada uno de los pasos realizados en el estudio de caso: antecedentes, diseño, sujetos de investigación y unidad de análisis, procedimiento de campo, preparación de recolección de datos, ejecución de caso y análisis.

4.1. Antecedentes

En la literatura se encontraron algunos estudios relacionados con la mantenibilidad de software, los cuales se presentan en el *Capítulo II* de esta investigación. Aunque existen diversos enfoques que abordan la mantenibilidad, se debe resaltar que el área de aplicación de esta investigación es la inclusión de prácticas a las diferentes etapas del proceso de desarrollo que permitan potenciar la mantenibilidad del producto. Teniendo en cuenta lo anterior, la pregunta principal de investigación que se pretende responder con este estudio de caso es:

PP: ¿La utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS permite potenciar la mantenibilidad del producto?

Otras preguntas adicionales que se pueden plantear son:

PA1: ¿Aumenta la tasa de éxito para realizar cambios correctamente con la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS?

PA2: ¿La utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS permite que el tiempo invertido al realizar cambios al producto software genere los resultados esperados?

PA3: ¿Utilizar las prácticas base del proceso de construcción del modelo de referencia MANTuS permite aumentar la tasa de éxito en el encuentro de la causa de fallos?

PA4: ¿Disminuye el tiempo de análisis de fallos con la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS?

PA5: ¿Disminuyen los costos con la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS?

4.2. Diseño

Según el enfoque propuesto en [94], el tipo de diseño del estudio de caso para esta investigación es simple embebido. Simple porque se tiene un único contexto con el mismo estudio de caso y embebido porque son dos unidades de análisis (dos versiones del mismo código fuente). Este estudio de caso compara los resultados obtenidos al abordar las dos unidades de análisis. Por otro lado las medidas usadas para indagar sobre las preguntas de investigación definidas son: (i) número de funcionalidades modificadas

correctamente por todos los participante a un código sin prácticas y a un código que incluye las prácticas definidas en el modelo de referencia, (ii) el tiempo requerido para realizar cambios a un código sin prácticas y a un código que incluye las prácticas definidas en el modelo de referencia, relacionado con la sub-característica capacidad para ser modificado, (iii) tasa de éxito en el encuentro de la causa de los fallos a un código sin prácticas y a un código que incluye las prácticas definidas en el modelo de referencia, relacionado con la sub-característica capacidad para ser analizado, (iv) tiempo de análisis de fallos de un código sin prácticas y de un código que incluye las prácticas definidas en el modelo de referencia, relacionado con la sub-característica capacidad para ser analizado.

4.3. Sujetos de investigación y unidad de análisis

Los sujetos de investigación participantes en este estudio de caso fueron ocho estudiantes de últimos semestres del programa Ingeniería de Sistemas de la Universidad del Cauca. Para este estudio de caso se tuvieron dos unidades de análisis las cuales son dos versiones del mismo programa: una versión que evidencia la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS (código B) y otra que no las utiliza (código A).

A continuación se describe de manera general las características funcionales del programa. Es un programa implementado en C++ que simula algunas funcionalidades que se pueden realizar en un banco. Para ello el programa permite crear una cuenta bancaria, la cual puede ser de dos tipos:

Cuenta corriente donde se realizan las siguientes transacciones:

- Visualizar datos de la cuenta del cliente
- Consignar dinero
- Retirar dinero
- Consultar saldo

Cuenta de ahorros donde se realizan las siguientes transacciones:

- Visualizar datos de la cuenta del cliente
- Consignar dinero
- Retirar dinero
- Consultar saldo
- Realizar transferencias
- Ver las últimas dos transacciones realizadas.

Las Figura 5 y Figura 6 muestran la estructura del código B y el código A respectivamente. Se observa que el código B separa las funcionalidades de las clases en diferentes archivos .cpp donde se implementan las funciones, mientras que en el código A se implementa todas las funcionalidades en el main, las funciones de todas las clases están declaradas en un único archivo .cpp y además están vacías.

```
Banco.h

    class Persona{ ...  };

    class Cliente:public virtual Persona{ ... };

    class Cuenta{ ... };

    class Transaccion{ ... };

    class Cuenta_Ahorro:public Cuenta{ ... };

    class Cuenta_Corriente:public Cuenta{ ... };
```

```
Transaccion.cpp

    Transaccion::Transaccion(){}

    //Implementación métodos Set y Get
```

```
Persona.cpp

    Persona::Persona(){}

    //Implementación métodos Set y Get
```

```
Cliente.cpp

    //Clase cliente hereda de la clase persona
    Cliente::Cliente():Persona(){}
```

```
Cuenta.cpp

    Cuenta::Cuenta(){}

    //Implementación métodos Set y Get
```

```
Cuenta_Corriente.cpp

    Cuenta_Corriente::Cuenta_Corriente():Cuenta(){}

    int Cuenta_Corriente::BuscarCorrienteCedula(...){ ... }

    int Cuenta_Corriente::BuscarCorrienteCuenta(...)
    {
        int i;

        for (i = 0; i < con_corriente; i++)
        {
            if (cuenta_buscada == vec_corriente[i].GetNumCuenta())
            {
                return i;
            }
        }
        return -1;
    }

    void Cuenta_Corriente::ConsignarCuentaCorriente(...){ ... }

    void Cuenta_Corriente::RetirarCuentaCorriente(...){ ... }
```

```
main.cpp
    int MenuTipoCuenta()
    {
        ...
    }
    int main(...)
    {
        //Bloque de declaraciones
        ...
        do
        {
            ...
            switch (...)
            {
                case 1:
                    MenuTipoCuenta();
                    ...
                break;
                ...
                case 3:
                    ...
                    ob_cuenta_corriente.BuscarCorrienteCuenta(...);
                    ...
                break;
                ...
                case 5:
                    MenuTipoCuenta();
                    switch (...)
                    {
                        ...
                        case 2:
                            ...
                            switch (...)
                            {
                                ...
                            }
                        break;
                    }
                break;
            }
            ...
        }while (...);
    }
```

```
Cuenta_Ahorro.cpp

    Cuenta_Ahorro::Cuenta_Ahorro():Cuenta (){}

    int Cuenta_Ahorro::BuscarAhorroCedula(...){ ... }

    int Cuenta_Ahorro::GuardarTransaccion(...){ ... }

    int Cuenta_Ahorro::BuscarAhorroCuenta(...){ ... }

    float Cuenta_Ahorro::ConsignarCuentaAhorro(...){ ... }

    float Cuenta_Ahorro::RetirarCuentaAhorro(...){ ... }

    float Cuenta_Ahorro::Transferir(...){ ... }
```

Figura 5. Estructura código B

```
Banco.h

    class Persona{ ... };

    class Cliente:public virtual Persona{ ... };

    class Cuenta{ ... };

    class Cuenta_Ahorro:public Cuenta{ ... };

    class Cuenta_Corriente:public Cuenta{ ... };
```

```
Banco.cpp

    /*Clase Persona*/
    Persona::Persona(){}

    //Implementación métodos Set y Get

    /*Clase Cliente*/
    Cliente::Cliente():Persona(){}

    /*Clase Cuenta*/
    Cuenta::Cuenta(){}

    Cuenta::Cuenta(...){ ... }

    //Implementación métodos Set y Get

    void Cuenta::Consignar(){ /*Método vacío*/}

    /*Sobrecarga del operadores*/
    float Cuenta::operator+(...){ ... }

    float Cuenta::operator-(...){ ... }

    /*Clase Cuenta_ahorro*/
    Cuenta_Ahorro::Cuenta_Ahorro():Cuenta (){}

    Cuenta_Ahorro:: Cuenta_Ahorro(...){ /*Método vacío*/ }

    void Cuenta_Ahorro :: Retirar(){ /*Método vacío*/ }

    void Cuenta_Ahorro :: Transferir_dinero(){ /*Método vacío*/ }

    void Cuenta_Ahorro :: Consultar_transaccion(){ /*Método vacío*/ }

    /*Clase cuenta corriente*/
    Cuenta_Corriente::Cuenta_Corriente():Cuenta(){}

    Cuenta_Corriente:: Cuenta_Corriente(...){}

    void Cuenta_Corriente:: Girar_cheque(){ /*Método vacío*/ }
```

```
main.cpp

    int main(...)
    {
        //Declaración variables
        ...
        do{
            ...
            if(...){
                if(...){
                    ...
                }
                if(...){
                    ...
                }
                if(opcion=="3"){
                    ...
                    for(i=0;i<con_ahorro;i++){
                        if(consignar==vec_ahorro[i].Get_num_cuenta())
                        {
                            ...
                        }
                    }
                    ...
                }
                if(...){
                    ...
                }
                if(...){
                    ...
                    if(...){
                        if(...){
                            ...
                            if(...){
                                if(...){
                                    ...
                                }
                                if(...){
                                    ...
                                }
                            }
                        }
                        if(...){
                            ...
                            if(...){
                                if(...){
                                    ...
                                }
                                ...
                            }
                        }
                    }
                }
            }
        }while(...);
    }
```

Figura 6. Estructura código A

Para tener una vista más específica de los dos códigos ver Anexo E (CD:\Anexos\Anexo E) donde se puede observar que el código B incluye las prácticas de mantenibilidad especificadas en la Tabla 9, la cual indica con una X las prácticas que han sido aplicadas y las que no, para estas últimas se explica por qué no se incluyeron al código B.

ID Práctica	Nombre Práctica	¿Aplicada?		¿Por qué? (En caso de No aplicada)
		Si	No	
DEV-MANT.4.PB1	Utilizar comentarios	X		
DEV-MANT.4.PB2	Describir propósitos	X		
DEV-MANT.4.PB3	Verificar los flujos de control	X		
DEV-MANT 4.PB4	Organizar código		X	El código original ya está bien indentado
DEV-MANT.4.PB5	Asignar nombres claros	X		
DEV-MANT.4.PB6	Controlar abreviaciones	X		
DEV-MANT.4.PB7	Controlar sentencias		X	No existen sentencias largas en el código original

DEV-MANT.4.PB8	Controlar paréntesis	X		
DEV-MANT.4.PB9	Determinar funcionalidades		X	Porque no se cuenta con los requisitos y diseño para este problema.
DEV-MANT.4.PB10	Dividir métodos	X		
DEV-MANT.4.PB11	Controlar tamaño	X		
DEV-MANT.4.PB12	Controlar nivel de anidación	X		
DEV-MANT.4.PB13	Evitar duplicación	X		
DEV-MANT.4.PB14	Definir estándares	X		
DEV-MANT.4.PB15	Utilizar convenciones	X		
DEV-MANT.4.PB16	Controlar tamaño de unidad	X		
DEV-MANT.4.PB17	Verificar trazabilidad		X	Porque no se cuenta con los artefactos de etapas anteriores
DEV-MANT.4.PB18	Actualizar artefactos		X	Porque no se cuenta con los artefactos de etapas anteriores

Tabla 9. Prácticas aplicadas en el estudio de caso

4.4. Procedimiento de campo

Para llevar a cabo el estudio de caso, se estableció el siguiente protocolo: contextualización a los participantes, entrega de instrumentos para la realización del estudio de caso, ejecución de código a modificar, inicio aplicación, entrega de resultados y diligenciamiento de encuesta.

4.5. Preparación para la recolección de datos

Para la recolección de datos se utilizó una guía donde se indican las diferentes modificaciones a realizar y se presentan unas plantillas para ingresar algunos valores de tiempo y opiniones respecto a las modificaciones realizadas. El ítem 6 plantea cambio correctivo y los ítems 1, 2, 3, 4, 7, 8 cambios perfectivos. El ítem 5 se relaciona con el encuentro de causas de fallos. A continuación, se muestra la guía de ejemplo entregada a cada participante. Adicionalmente se indican las medidas (definidas en el diseño) con las que se relaciona cada ítem. Todas las guías diligenciadas se encuentran en el Anexo F (ver CD:\Anexos\Anexo F).

1. Añadir a todos los menús la opción de volver al anterior. (i, ii,)

Hora inicio	
Hora fin	
Número de opciones agregadas	
¿En cuales menús realizó cambios?	
¿Le resultó fácil encontrar la ubicación del código donde debía realizar los cambios? ¿Por qué?	

2. No permitir la existencia de dos cuentas con el mismo número. (i, ii)

Hora inicio	
Hora fin	
¿Le resultó fácil encontrar la ubicación del código donde debía realizar los cambios? ¿Por qué?	

¿Le resultó fácil realizar el cambio? ¿Por qué?	

3. Incluir los mensajes "cuenta inexistente" que falta en la función transferir cuando se desea realizar una transferencia y alguna de las dos cuentas no existen. Se debe probar los dos casos, es decir, que no exista la cuenta de origen o que no exista la cuenta de destino. (i, ii)

Hora inicio	
Hora fin	
¿Le resultó fácil encontrar la ubicación del código donde debía realizar los cambios? ¿Por qué?	
¿Le resultó fácil realizar el cambio? ¿Por qué?	

4. Cuando se crea una cuenta, no solicitar el valor del saldo al usuario, fijar este valor en $ 50000 para cuentas corrientes y $ 70000 para cuentas de ahorros. (i, ii, iv)

Hora inicio	
Hora de encuentro del código a modificar	
Hora fin	
¿Le resultó fácil encontrar la ubicación del código donde debía realizar los cambios? ¿Por qué?	
¿Qué pasos siguió para realizar el cambio?	

5. ¿Cuáles son las posibles causas que provocan el fallo del programa en diferentes funcionalidades? Crear dos cuentas de ahorro para visualizar los fallos. (ii, iii, iv)
Para cada fallo encontrado llenar la siguiente tabla:

5.1.

Hora inicio	
Hora fin	
Descripción del fallo	Se consigna en una cuenta de ahorro y el programa falla al consultar la opción visualizar datos del cliente.
Causa(s) del fallo (si ha sido encontrada)	
¿Le resultó fácil encontrar la ubicación del código donde debía realizar los cambios? ¿Por qué?	

5.2.

Hora inicio	
Hora de encuentro del código a modificar	
Hora fin	
Descripción del fallo	Se retira dinero de las dos cuentas y al consultar las dos últimas transacciones los resultados son erróneos.
Causa(s) del fallo (si ha sido encontrada)	
¿Le resultó fácil encontrar la ubicación del código donde debía realizar los cambios? ¿Por qué?	

5.3.

Hora inicio	
Hora de encuentro del código a modificar	
Hora fin	
Descripción del fallo	Cuando se hace una transferencia de dinero y se consultan las últimas dos transacciones el sistema no arroja los resultados esperados.
Causa(s) del fallo (si ha sido encontrada)	
¿Le resultó fácil encontrar la ubicación del código donde debía realizar los cambios? ¿Por qué?	

6. Corregir las causas de los fallos que se presentan en las diferentes funcionalidades cuando se tiene más de una cuenta de ahorro. (i, ii)
   Para cada causa del fallo corregida llenar la siguiente tabla.

Hora inicio	
Hora fin	
¿Le resultó fácil realizar el cambio? ¿Por qué?	
¿Dónde realizo el cambio?	
¿Qué tuvo que hacer para solucionar el fallo?	

7. No permitir la transferencia de dinero desde una cuenta hacia la misma. (i, ii, iv)

Hora inicio	
Hora de encuentro del código a modificar	
Hora fin	
¿Le resultó fácil encontrar la ubicación del código donde debía realizar los cambios? ¿Por qué?	
¿Le resultó fácil realizar el cambio? ¿Por qué?	

8. Permitir que el cliente pueda ver las últimas tres transacciones. (i, ii, iv)

Hora inicio	
Hora de encuentro del código a modificar	
Hora fin	
¿Le resultó fácil encontrar la ubicación del código donde debía realizar los cambios? ¿Por qué?	
¿Qué pasos siguió para realizar el cambio?	
¿Le resultó fácil realizar el cambio? ¿Por qué?	

También se utilizó una encuesta en Google Drive que fue diligenciada por las participantes al finalizar todas las modificaciones propuestas. La realización de la encuesta tiene el objetivo de conocer las opiniones de los participantes con respecto a características de mantenibilidad de cada uno de los códigos. A continuación se presentan las preguntas de la encuesta, todas las respuestas registradas se pueden ver en el Anexo G (ver CD:\Anexos\Anexo G).

1) ¿Le resulto fácil realizar las modificaciones? ¿Por qué?

2) ¿Le pareció fácil encontrar las causas de los fallos? ¿Por qué?

3) ¿Le pareció organizado el código? ¿Por qué?

4) ¿Le resulto fácil entender el código? ¿Por qué?

5) ¿Le resulto fácil entender el código? ¿Por qué?

6) ¿El código estaba bien documentado? ¿Por qué?

7) ¿Las variables tenían nombres claros y entendibles?

8) ¿Encontró código duplicado en el programa?

Nota: si su respuesta es sí conteste la pregunta 9, sino continúe con la 10.

9) Esta situación le afecto en la realización de los cambios solicitados. ¿Por qué?

10) La mantenibilidad de software es la capacidad del producto software para ser modificado efectiva y eficientemente. En su opinión ¿el código tiene alta mantenibilidad? ¿Por qué?

4.6. Ejecución del caso

La aplicación del estudio de caso siguió el procedimiento de campo mencionado anteriormente.

1. Preparación

Definición del protocolo a seguir durante la ejecución del estudio de caso:

- Contextualización a los participantes
- Entrega de instrumentos para la realización del estudio de caso: los participantes se dividen en dos grupos: Grupo A que trabajan con el código A el cual no incluye prácticas de mantenibilidad y Grupo B que trabajan con el código B el cual si incluye algunas prácticas de mantenibilidad definidas para el proceso de construcción. Posteriormente se le entrega a cada participante la guía del trabajo a realizar y el código al que le debe realizar las modificaciones. Por último se le envía al correo el enlace de la guía a diligenciar.
- Ejecución de código a modificar: cada participante tiene 5 minutos para ejecutar el código entregado, con el fin de que verifique todas las funcionalidades del programa.
- Inicio aplicación: los participantes desarrollan todos los ítems de la guía entregada, para cada ítem llevan el control del tiempo invertido llenando la tabla que se encuentra después de la descripción del ítem. Los participantes indican mediante comentarios las líneas de código en las que modifican. Los investigadores resuelven las dudas que tengan los participantes al diligenciara y realizar los ítems de la guía.
- Entrega de resultados: cuando el participante termine todos los ítems propuestos en la guía, entrega el código con las modificaciones realizadas y la guía totalmente diligenciada.

- Diligenciamiento de encuesta: una vez el participante entregue el código y la guía, éste diligencia la encuesta que se le envió al correo anteriormente, respondiendo a todas las preguntas que se encuentran en ella.

2. **Ejecución**
   El estudio de caso inició con la contextualización a los participantes, donde se explicó el objetivo del estudio de caso y la idea general del programa a modificar. Se indicó cómo se iba a desarrollar el ejercicio aplicativo y cuáles eran los artefactos a entregar, es decir el código con las modificaciones, la guía y encuesta diligenciadas.

3. **Seguimiento**
   Durante la realización del estudio de caso los investigadores resolvieron todas las inquietudes presentadas por los participantes y estuvieron atentos a que éstos consignaran los datos solicitados en las tablas de la guía, los cuales eran necesarios para calcular las métricas.

4.6.1. Datos recolectados

Con los tiempos registrados por los participantes al realizar las modificaciones solicitadas, se elaboraron dos tablas resumen con los tiempos invertidos (*Duración*) por cada uno de ellos y *¿Correcto?* que comprende los valores si o no, los cuales indican que la modificación fue realizada con éxito o no, respectivamente. Para observar los códigos modificados por cada uno de los participantes ver el Anexo H (CD:\Anexos\Anexo H).

Para calcular la *Duración* del ítem 6, fue necesario sumar los tiempos de modificación para la corrección de los tres fallos presentados en este ítem. Estos cálculos se pueden ver en la Tabla 10.

Participante		Fallo 6.1	Fallo 6.2	Fallo 6.3	Duración ítem 6
	T. fin	12:03:00	00:00	0	
	T. inicio	11:46:03	0	0	
PA1	**T. total**	**00:16:57**	**0**	**0**	00:16:57
	T. fin	11:53:30	11:55:30	11:58:30	
	T. inicio	11:50:23	11:53:30	11:55:30	
PA2	**T. total**	**00:03:07**	**00:02:00**	**00:03:00**	00:08:07
	T. fin	12:01:12	0	0	
	T. inicio	11:53:04	0	0	
PA3	**T. total**	**00:08:08**	**0**	**0**	00:08:08
	T. fin	11:50:10	0	0	
	T. inicio	11:45:10	0	0	
PA4	**T. total**	**00:05:00**	**0**	**0**	00:05:00
	T. fin	13:05:24	0	12:19:58	
	T. inicio	12:10:26	0	12:13:05	
PB1	**T. total**	**00:54:58**	**0**	**00:06:53**	01:01:51
	T. fin	07:30:20	0	07:53:30	
	T. inicio	07:05:20	0	07:51:00	
PB2	**T. total**	**00:25:00**	**0**	**00:02:30**	00:27:30

	T. fin	10:15:12	0	10:22:26	
	T. inicio	09:29:13	0	10:21:15	
PB3	**T. total**	**00:45:59**	**0**	**00:01:11**	00:47:10
	T. fin	11:27:38	0	12:36:31	
	T. inicio	10:35:10	0	12:32:20	
PB4	**T. total**	**00:52:28**	**0**	**00:04:11**	00:56:39

Tabla 10. Duración ítem 6

En la Tabla 11 se muestran los tiempos invertidos por los participantes al realizar las modificaciones de los ítems 1, 2, 3, 4, 6, 7, 8, y el tiempo total al realizar todos los cambios. Además la tabla incluye el cálculo del tiempo total y promedio invertido por todos los participantes para cada uno de los ítems.

Participante		ítem 1	ítem 2	ítem 3	ítem 4	ítem 6	ítem 7	ítem 8	Tiempo Total
	T. Fin	10:01:26	10:54:33	11:02:27	11:07:52		11:56:36	11:58:56	
	T. Inicio	09:34:04	10:08:15	11:00:03	11:04:05		11:51:07	11:57:30	
	Duración	**00:27:22**	**00:46:18**	**00:02:24**	**00:03:47**	**00:16:57**	**00:05:29**	**00:01:26**	
PA1	¿Correcto?	Si	No	No	Si	No	No	No	1:43:43
	T. Fin	10:29:35	11:02:30	11:17:50	11:24:40		12:06:10	12:08:15	
	T. Inicio	09:33:40	10:34:50	11:05:45	11:22:40		12:01:00	12:07:10	
	Duración	**00:55:55**	**00:27:40**	**00:12:05**	**00:02:00**	**00:08:07**	**00:05:10**	**00:01:05**	
PA2	¿Correcto?	No	No	No	Si	No	No	No	1:52:02
	T. Fin	10:27:20	11:08:10	11:21:10	11:32:04		12:20:01	12:23:50	
	T. Inicio	09:33:40	10:36:20	11:12:30	11:26:20		12:08:01	12:23:08	
	Duración	**00:53:40**	**00:31:50**	**00:08:40**	**00:05:44**	**00:08:08**	**00:12:00**	**00:00:42**	
PA3	¿Correcto?	Si	No	No	Si	No	No	No	2:00:44
	T. Fin	10:15:00	10:36:20	10:55:18	11:00:46		12:00:20	12:10:10	
	T. Inicio	09:36:18	10:15:42	10:36:26	10:58:25		11:54:10	12:03:50	
	Duración	**00:38:42**	**00:20:38**	**00:18:52**	**00:02:21**	**00:05:00**	**00:06:10**	**00:06:20**	
PA4	¿Correcto?	Si	Si	Si	Si	No	No	No	1:38:03
	T. total	02:55:39	02:06:26	00:42:01	00:13:52	00:38:12	00:28:49	00:09:33	
Grupo A	T. promedio	00:43:55	00:31:36	00:10:30	00:03:28	00:09:33	00:07:12	00:02:23	
	T. Fin	10:06:00	10:23:25	11:24:10	11:28:12		13:10:32	13:25:34	
	T. Inicio	09:30:00	10:06:20	11:07:22	11:25:00		13:00:20	13:10:35	
	Duración	**00:36:00**	**00:17:05**	**00:16:48**	**00:03:12**	**01:01:51**	**00:10:12**	**00:14:59**	
PB1	¿Correcto?	Si	Si	Si	Si	Si	Si	Si	2:40:07
	T. Fin	09:56:10	10:32:40	10:58:46	11:09:44		15:36:00	20:09:00	
	T. Inicio	09:33:10	10:11:00	10:42:50	11:03:48		15:21:00	20:01:00	
	Duración	**00:23:00**	**00:21:40**	**00:15:56**	**00:05:56**	**00:27:30**	**00:15:00**	**00:08:00**	
PB2	¿Correcto?	Si	Si	Si	Si	Si	Si	Si	1:57:02
	T. Fin	09:54:10	10:35:45	11:00:34	11:11:34		10:32:02	10:40:45	
PB3	T. Inicio	09:34:45	10:05:12	10:40:23	11:09:43	**00:47:10**	10:29:55	10:36:03	2:05:59

	Duración	**00:19:25**	**00:30:33**	**00:20:11**	**00:01:51**		**00:02:07**	**00:04:42**	
	¿Correcto?	Si	Si	Si	Si	Si	Si	Si	
	T. Fin	09:10:10	09:35:10	10:16:45	10:21:39		12:56:14	13:20:50	
	T. Inicio	09:00:00	09:13:30	09:55:00	10:19:27		12:52:10	13:04:28	
	Duración	**00:10:10**	**00:21:40**	**00:21:45**	**00:02:12**	**00:56:39**	**00:04:04**	**00:16:22**	
PB4	¿Correcto?	Si	Si	Si	Si	Si	Si	Si	2:12:52
	T. total	01:28:35	01:30:58	01:14:40	00:13:11	03:13:10	00:31:23	00:44:03	
Grupo B	T. promedio	00:22:09	00:22:44	00:18:40	00:03:18	00:48:17	00:07:51	00:11:01	

Tabla 11. Tiempo modificaciones

En la Tabla 12 se presenta para cada participante los tiempos invertidos al realizar el análisis del código para los ítems 4, 5, 7, 8 y el tiempo total.

Participante		Ítem 4	Ítem 5.1	Ítem 5.2	Ítem 5.3	Ítem 7	Ítem 8	T. Total
	T. Fin	11:04:05	11:30:00	11:40:45	11:44:20	11:51:07	11:57:30	
	T. Inicio	11:03:15	11:11:21	11:31:02	11:42:10	11:50:16	11:57:00	
	Duración	**00:00:50**	**00:18:39**	**00:09:43**	**00:02:10**	**00:00:51**	**00:00:30**	
PA1	¿Correcto?	Si	Si	Si	Si	No	No	0:32:43
	T. Fin	11:22:40	11:44:20	11:46:20	11:49:30	12:01:00	12:07:10	
	T. Inicio	11:21:30	11:27:25	11:45:05	11:47:20	12:00:10	12:06:15	
	Duración	**00:01:10**	**00:16:55**	**00:01:15**	**00:02:10**	**00:00:50**	**00:00:55**	
PA2	¿Correcto?	Si	No	No	No	No	No	0:23:15
	T. Fin	11:26:20	11:41:02	11:46:15	11:50:10	12:08:01	12:23:08	
	T. Inicio	11:24:40	11:33:03	11:42:11	11:47:10	12:05:20	12:22:01	
	Duración	**00:01:40**	**00:07:59**	**00:04:04**	**00:03:00**	**00:02:41**	**00:01:07**	
PA3	¿Correcto?	Si	No	No	No	No	No	0:20:31
	T. Fin	10:58:25	11:20:47	11:31:45	11:40:30	11:54:10	12:03:50	
	T. Inicio	10:57:35	11:15:20	11:21:52	11:32:20	11:52:30	12:02:00	
	Duración	**00:00:50**	**00:05:27**	**00:09:53**	**00:08:10**	**00:01:40**	**00:01:50**	
PA4	¿Correcto?	Si	No	No	No	No	No	0:27:50
	T. Fin	11:25:00	12:10:26	12:11:35	12:12:26	13:00:20	13:10:35	
	T. Inicio	11:24:30	11:21:40	12:10:27	12:11:35	13:00:10	13:05:15	
	Duración	00:00:30	00:48:46	00:01:08	00:00:51	00:00:10	00:05:20	
PB1	¿Correcto?	Si	Si	Si	Si	Si	Si	0:56:45
	T. Fin	11:03:48	19:05:00	19:39:00	19:50:30	15:21:00	20:01:00	
	T. Inicio	11:03:00	19:00:00	19:37:20	19:50:00	15:20:55	20:00:00	
	Duración	00:00:48	00:05:00	00:01:40	00:00:30	00:00:05	00:01:00	
PB2	¿Correcto?	Si	Si	Si	Si	Si	Si	0:09:03
	T. Fin	11:09:43	21:28:23	22:15:11	22:20:45	10:29:55	10:36:03	
	T. Inicio	11:08:56	21:14:33	22:13:56	22:19:34	10:28:36	10:35:11	
	Duración	00:00:47	00:13:50	00:01:15	00:01:11	00:01:19	00:00:52	
PB3	¿Correcto?	Si	Si	Si	Si	Si	Si	0:19:14

	T. Fin	10:19:27	10:35:05	10:35:05	12:32:18	12:52:10	13:04:28	
	T. Inicio	10:18:50	10:29:20	10:35:05	12:30:50	12:50:30	13:03:45	
	Duración	00:00:37	00:05:45	00:00:00	00:01:28	00:01:40	00:00:43	
PB4	¿Correcto?	Si	Si	Si	Si	Si	Si	0:10:13

Tabla 12. Tiempo análisis

4.7. Análisis de resultados

Para dar respuesta a las preguntas planteadas en este estudio de caso se realizó el análisis presentado a continuación.

PA1: ¿Aumenta la tasa de éxito para realizar cambios correctamente con la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS?

Con el fin de determinar la tasa de éxito en la ejecución de los cambios solicitados (modificaciones) a los participantes, a partir de los datos presentados en la Tabla 11 para cada cambio solicitado se generó una tabla de frecuencias que incluye tres columnas: *Modificación correcta* que toma los valores Si o No, *Frecuencia* que es el número de veces que se presenta cada uno de estos valores y *Frecuencia relativa* que indica el porcentaje de ocurrencia de cada valor. A partir de la frecuencia relativa se realizó un gráfico de torta donde es posible observar la tasa de éxito y que además incluye el tiempo total y promedio empleado por los participantes al realizar los cambios solicitados. Este análisis se realizó para los códigos A y B el cual es presentado a continuación.

Ítem 1: Añadir a todos los menús la opción de volver al anterior

Código A		
Modificación correcta	**Frecuencia**	**Frecuencia relativa (%)**
Si	3	75
No	1	25
Código B		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	4	100
No	0	0

Tabla 13. Frecuencia de éxito de modificación ítem 1

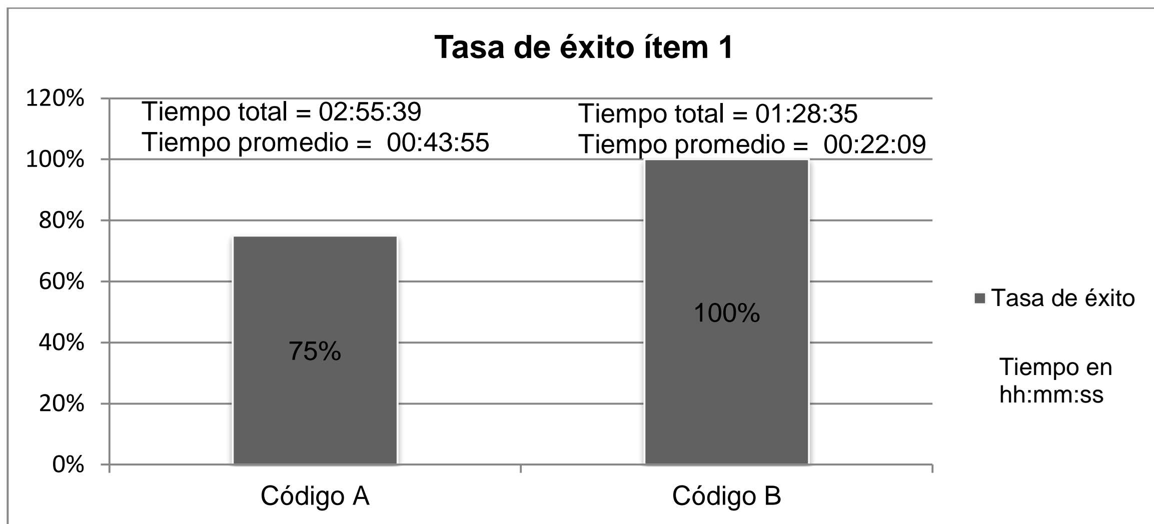

Gráfico 1. Tasa de éxito ítem 1

En el Gráfico 1 se observa que la tasa de éxito en la modificación del código B (100%) es mayor que la del código A (75%). Además, el tiempo promedio invertido por los participantes del grupo B (00:22:09) es menor respecto al tiempo promedio invertido por los participantes del grupo A (00:43:55). Estos resultados reflejan la opinión de los participantes, que se describe a continuación. Quienes modificaron el código A expresaron que aunque el cambio a realizar era sencillo tardaron debido a que el código contenía muchas llaves, era muy largo y no estaba separado por funciones, lo cual dificultó ubicar rápidamente donde debían incluir el código necesario para cumplir con esta funcionalidad. Por otra parte las personas que modificaron el código B opinaron que pudieron realizar el cambio con facilidad gracias a que los menús eran explícitos, a los comentarios incluidos en código y la estructura presentada por el mismo.

Ítem 2: No permitir la existencia de dos cuentas con el mismo número.

Código A		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	1	25
No	3	75
Código B		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	4	100
No	0	0

Tabla 14. Frecuencia de éxito de modificación ítem 2

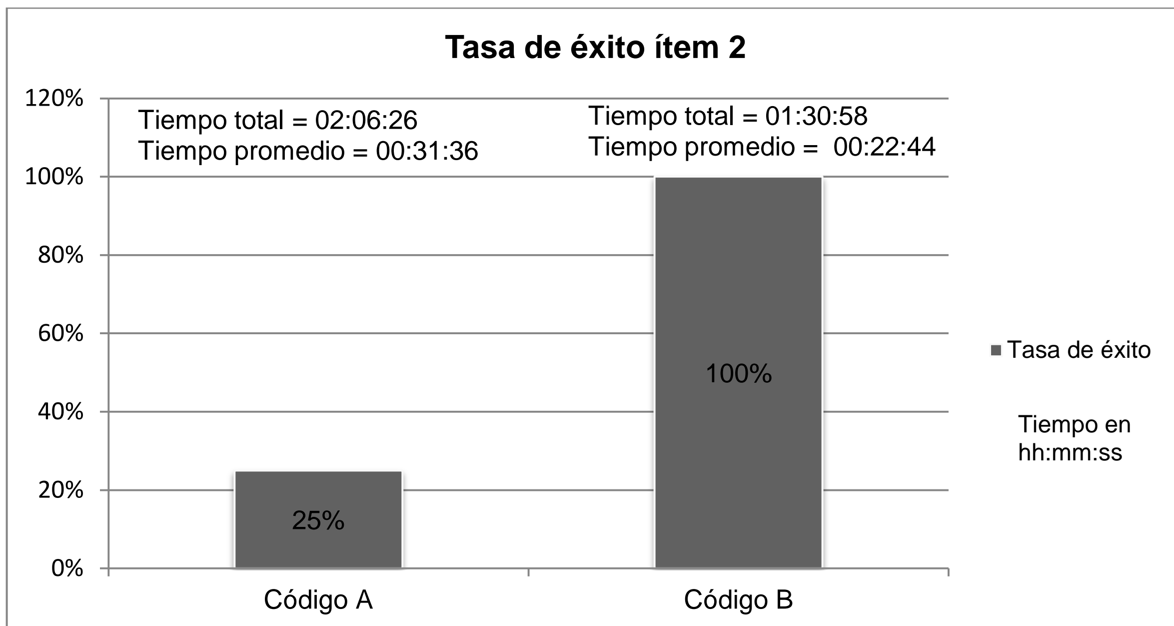

Gráfico 2. Tasa de éxito ítem 2

En el Gráfico 2 se puede observar que la tasa de éxito de modificación es mucho mayor para el código B (100%) que para el código A (25%) y aun así el tiempo promedio invertido por las personas del grupo A (00:31:36) es mayor que el invertido por el grupo B (00:22:44). Si bien todos los participantes del grupo A afirmaron que el cambio fue sencillo de hacer, al revisar la funcionalidad en el código se evidencia que el 75% de los participantes de este grupo realizaron la modificación de manera inadecuada, por ejemplo, dos de ellos aunque no se dejaba hacer la creación de cuentas repetidas el programa se quedaba en un ciclo debido a la lógica que utilizaron, la cual no solucionaba correctamente la modificación solicitada. Por otra parte, las personas del grupo B manifestaron que el cambio fue sencillo de realizar gracias a la documentación que presentaba el código y a las funciones que ya estaban implementadas, lo cual ayudo a que solo se agregaran unas condiciones para cumplir con lo solicitado.

Ítem 3: Incluir los mensajes “cuenta inexistente” que falta en la función transferir cuando se desea realizar una transferencia y alguna de las dos cuentas no existen. Se debe probar los dos casos, es decir, que no exista la cuenta de origen o que no exista la cuenta de destino.

Código A		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	1	25
No	3	75
Código B		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	4	100
No	0	0

Tabla 15. Frecuencia éxito de modificación ítem 3

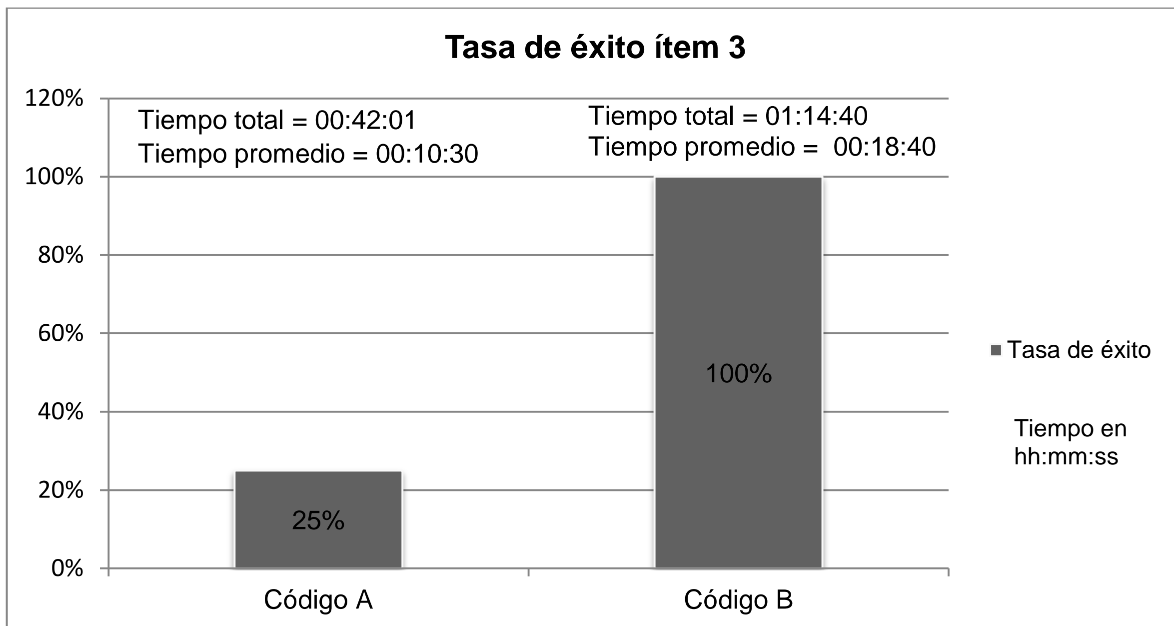

Gráfico 3. Tasa de éxito ítem 3

En el Gráfico 3 se observa que la tasa de éxito en la modificación para el código B (100%) es mucho mayor que la del código A (25%). Igualmente se observa que el tiempo invertido por los personas del grupo A (00:10:30) es menor que al invertido por el grupo B (00:18:40). Los resultados anteriores indican que aunque el grupo A realizó los cambios en menos tiempo, solo uno de ellos pudo incluir los mensajes de manera correcta.

Todas las personas del grupo A dijeron que les resultó fácil realizar la modificación ya que solo consistió en agregar unos mensajes, pero a la vez algunos afirmaron que lo difícil fue encontrar el lugar del código donde había que colocarlos porque no existían comentarios en esta parte y que fue confuso debido a las variables que se utilizaban para verificar la existencia de las cuentas. Al analizar los códigos, se verificó que la ubicación exacta para agregar los mensajes era difícil de hallar ya que esta parte presentaba varios ciclos y condicionales anidados, lo que produjo que el 75% de este grupo colocara los mensajes en lugares equivocados y finalmente no cumplieran satisfactoriamente con el cambio pedido. Por otra parte, los integrantes del grupo B afirmaron que esta modificación también fue sencilla de realizar gracias a los comentarios, indentación y estructura del código, además que por los métodos ya implementados solo fue necesario agregar unos mensajes.

Ítem 4: Cuando se crea una cuenta, no solicitar el valor del saldo al usuario, fijar este valor en $ 50000 para cuentas corrientes y $ 70000 para cuentas de ahorros.

Código A		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	3	75
No	1	25
Código B		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	4	100
No	0	0

Tabla 16. Frecuencia de éxito de modificación ítem 4

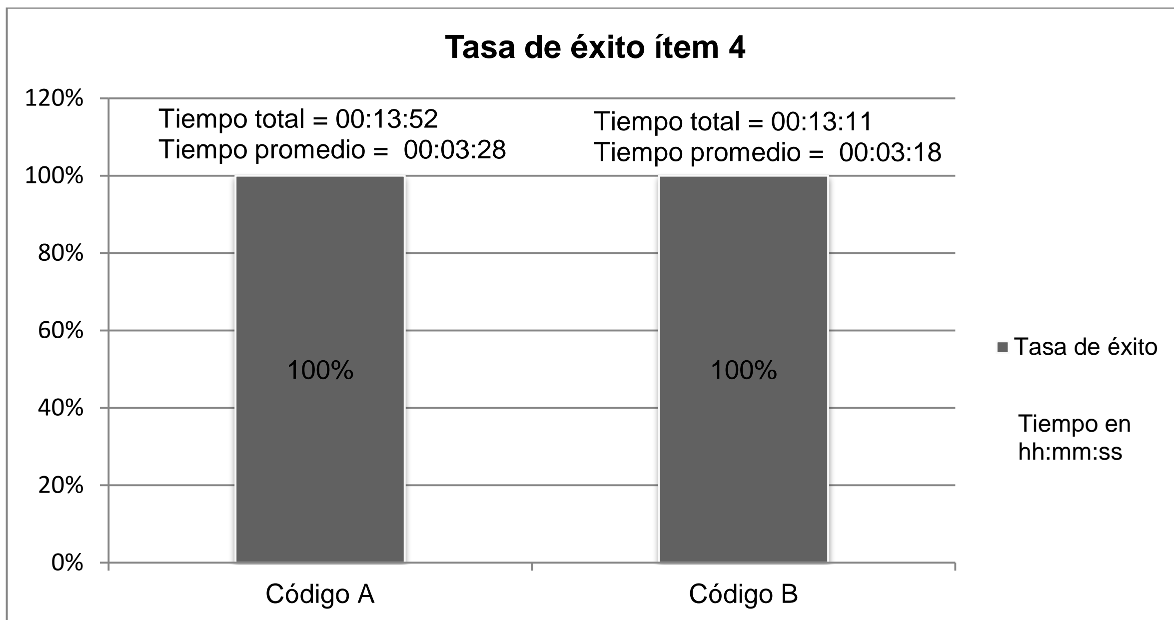

Gráfico 4. Tasa de éxito ítem 4

En el Gráfico 4 se puede observar que la tasa de éxito en la modificación para el código A (100%) es igual que la del código B (100%), sin embargo, el tiempo promedio invertido por los integrantes del grupo B (00:003:18) es menor respecto al invertido por el grupo A (00:03:28). Es importante resaltar que este era un cambio que se podía realizar más fácilmente en el código A pero aun así el tiempo promedio fue menor para el grupo B.

Las personas del grupo A afirmaron que este cambio fue realmente sencillo ya que por los puntos anteriores estaban más familiarizados con el código y solo tuvieron que cambiar los valores que recibía el constructor. Al analizar los códigos se evidencia que en algunos de ellos aunque la modificación se realiza correctamente, al momento de imprimir los datos esto se hace de forma inadecuada, es decir, se imprimen varias veces los mismos valores debido la lógica que utilizaron para realizar el cambio. Por otra parte, las personas del grupo B afirmaron que para ellos el cambio resulto fácil gracias al orden de los menús, a la indentación y a los comentarios presentes en el código, y al ya existir la función solo tuvieron que cambiar el valor en donde se fijaban los datos al objeto.

Ítem 6: Corregir las causas de los fallos que se presentan en las diferentes funcionalidades cuando se tiene más de una cuenta de ahorro.

Código A		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	0	0
No	4	100
Código B		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	4	100
No	0	0

Tabla 17. Frecuencia de éxito de modificación ítem 6

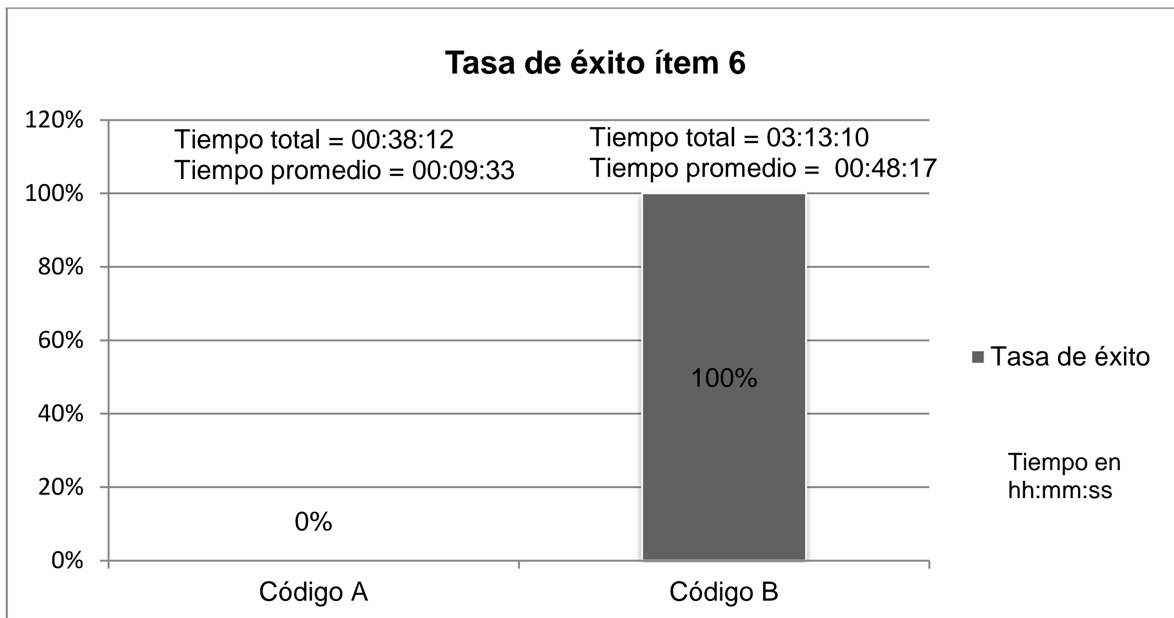

Gráfico 5. Tasa de éxito ítem 6

En el Gráfico 5 se observa que la tasa de éxito en la modificación del código B (100%) es mucho mayor que la del código A (0%). Igualmente se puede observar que el tiempo promedio invertido por los participantes del grupo A (00:09:33) es menor que el del grupo B (00:48:17). Lo anterior indica que los participantes del grupo A realizaron los cambios en menor tiempo pero ninguno de ellos logró solucionar los fallos.

Aunque el 75% de los participantes del grupo A afirmaron haber realizado fácilmente los cambios necesarios para solucionar los fallos, no corrigieron las zonas correctas del código ya que no habían identificado las verdaderas causas de los fallos. Por otra parte el único participante del grupo A que encontró las causas de los fallos (participante PA1), manifestó que no había podido realizar los cambios ya que la lógica del código le dificultó esta tarea. A pesar de que funcionalidad a corregir (guardar transacciones) presentaba la misma lógica en los dos códigos, todos los participantes del grupo B pudieron corregir los fallos presentados. La mayoría de ellos expresaron que la documentación del código fue de gran ayuda para identificar la función donde se debían realizar los cambios, de igual forma la ausencia de duplicación de código les permitió resolver dos fallos realizando un cambio en una función que era llamada desde diferentes partes del código.

Ítem 7: No permitir la transferencia de dinero desde una cuenta hacia la misma.

Código A		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	1	25
No	3	75
Código B		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	4	100
No	0	0

Tabla 18. Frecuencia de éxito de modificación ítem 7

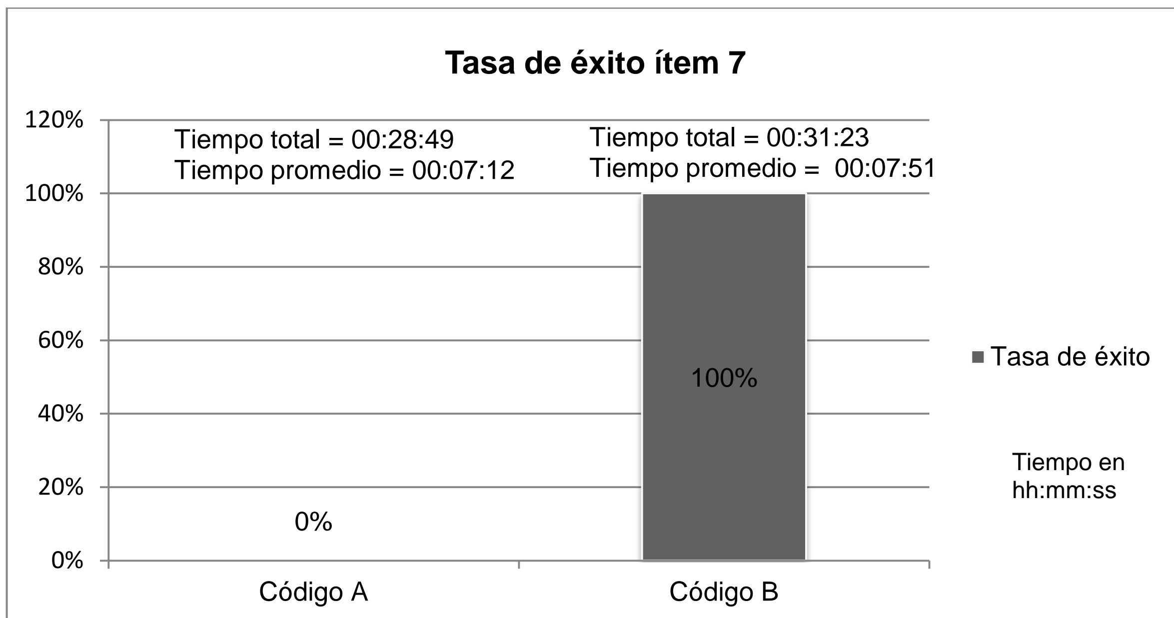

Gráfico 6. Tasa de éxito ítem 7

En el Gráfico 6 se observa que la tasa de éxito en la modificación es mucho mayor en el código B (100%) que en el código A (0%). Asimismo se puede ver que el tiempo promedio de modificación para el código B (00:07:51) es mayor que para el código A (00:07:12). Lo anterior evidencia que todos los participantes del grupo B pudieron realizar correctamente el cambio solicitado, de acuerdo con su opinión la modificación era muy sencilla y la realizaron rápidamente debido a que solo tuvieron que agregar un condicional. Por otra parte, todos los participantes del grupo A afirmaron que el cambio era muy fácil, uno de ellos manifestó que no pudo realizarlo correctamente debido a la cantidad de condicionales presentados en el código y a pesar de la opinión presentada acerca del cambio ningún participante del grupo A logó incluir exitosamente la funcionalidad. Al revisar los códigos modificados por el grupo A se pudo notar que zona del código donde se debía realizar el cambio tenía una lógica compleja que incluía una serie de ciclos anidados lo cual impidió que los cambios realizados fueran exitosos.

Ítem 8: Permitir que el cliente pueda ver las últimas tres transacciones

Código A		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	0	0
No	4	100
Código B		
Valor	**Frecuencia**	**Frecuencia relativa (%)**
Si	4	100
No	0	0

Tabla 19. Frecuencia de éxito de modificación ítem 8

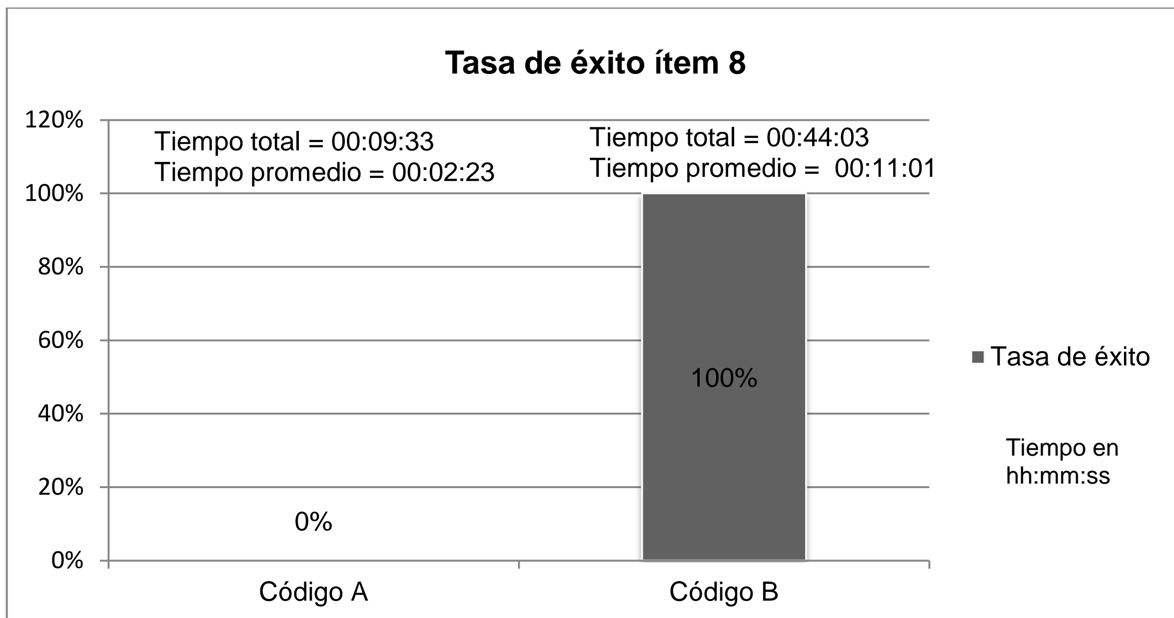

Gráfico 7. Tasa de éxito ítem 8

En el Gráfico 7 se observa que la tasa de éxito en la modificación para el código B (100%) es mucho mayor que la del código A (0%). También se puede ver que el tiempo promedio al realizar las modificaciones es menor en el código A (00:02:23) que en el B (00:11:01). Lo anterior indica que aunque el tiempo promedio para realizar las modificaciones en el código A fue menor, ninguno de los participantes del grupo A logró realizar el cambio correctamente, mientras que en el grupo B todos pudieron hacerlo. En el grupo A los cambios realizados no se vieron reflejados en el funcionamiento del programa debido a que está modificación dependía de los cambios realizados en la modificación 6 donde se obtuvo una tasa de éxito de modificación del 0%. En el grupo B los participantes manifestaron que aunque pudieron realizar la modificación correctamente gracias a la documentación y nombrado de variables en el código, la estructura del mismo aumentó el tiempo en la modificación ya que debían realizar cambios en diferentes archivos del programa.

Partiendo del análisis realizado anteriormente se generaron las Tabla 20 y Tabla 21 que presentan el total de cambios a realizar por todos los participantes, el total de cambios realizados con éxito y sin éxito para los códigos A y B respectivamente.

Grupo A		
		Porcentaje (%)
Si	9	32,1
No	19	67,9
Total	28	100

Tabla 20. Porcentaje de éxito grupo A

Grupo B		
		Porcentaje (%)
Si	28	100
No	0	0
Total	28	100

Tabla 21. Porcentaje de éxito grupo B

En el Gráfico 8 se presenta la tasa de éxito de todas las modificaciones realizadas por todos los participantes en cada uno de los códigos. Para dar respuesta a la pregunta 1, Se observa que la tasa de éxito de modificación es mucho mayor para el código B (100%) que para el A (32,1%), lo cual evidencia que el incluir las prácticas del modelo de

referencia definidas para el proceso de construcción aumenta la tasa de éxito de realizar los cambios correctamente.

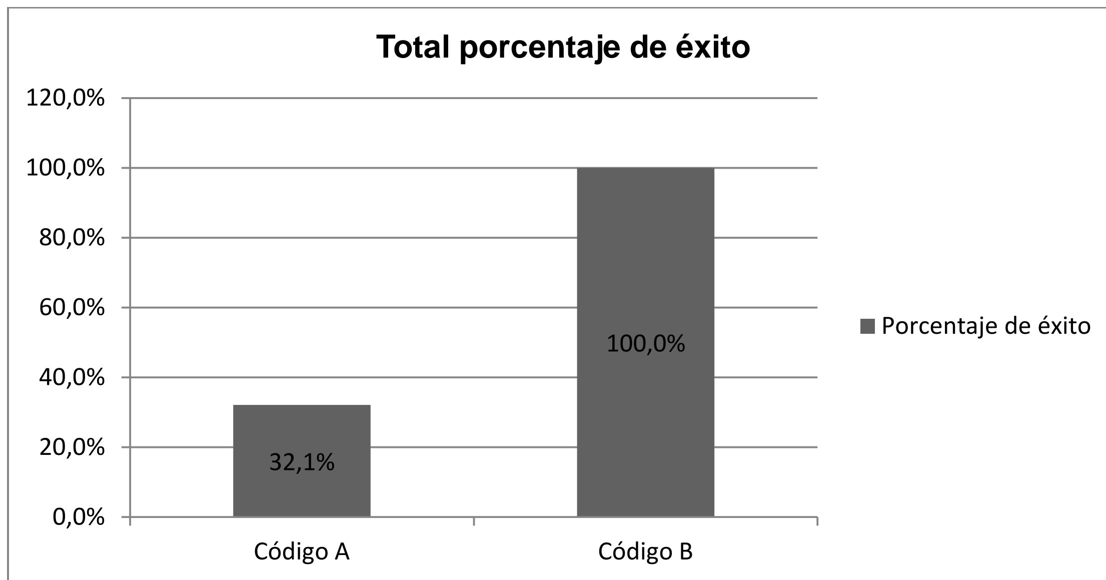

Gráfico 8. Total porcentaje de éxito

PA2: ¿La utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS permite que el tiempo invertido al realizar cambios al producto software genere los resultados esperados?

El tiempo total invertido para realizar todos los cambios al producto fue hallado para cada participante sumando los tiempos finales empleados en el desarrollo de cada ítem de la guía tanto en modificación como en análisis presentados en las Tabla 11 y Tabla 12. De igual forma se halló la tasa de éxito teniendo los siete ítems que solicitaban la realización de modificaciones al producto. Los datos obtenidos se presentan en la Tabla 22.

Participante	Tiempo de Análisis (hh:mm:ss)	Tiempo de Modificación (hh:mm:ss)	Tiempo Total (hh:mm:ss)	Tasa de éxito (%)
PA1	00:32:43	01:43:43	02:16:26	28,6
PA2	00:23:15	01:52:02	02:15:17	14,3
PA3	00:20:31	02:00:44	02:21:15	28,6
PA4	00:27:50	01:38:03	02:05:53	57,1
PB1	00:56:45	02:40:07	03:36:52	100
PB2	00:09:03	01:57:02	02:06:05	100
PB3	00:19:14	02:05:59	02:25:13	100
PB4	00:10:13	02:12:52	02:23:05	100

Tabla 22. Tiempo total por participante

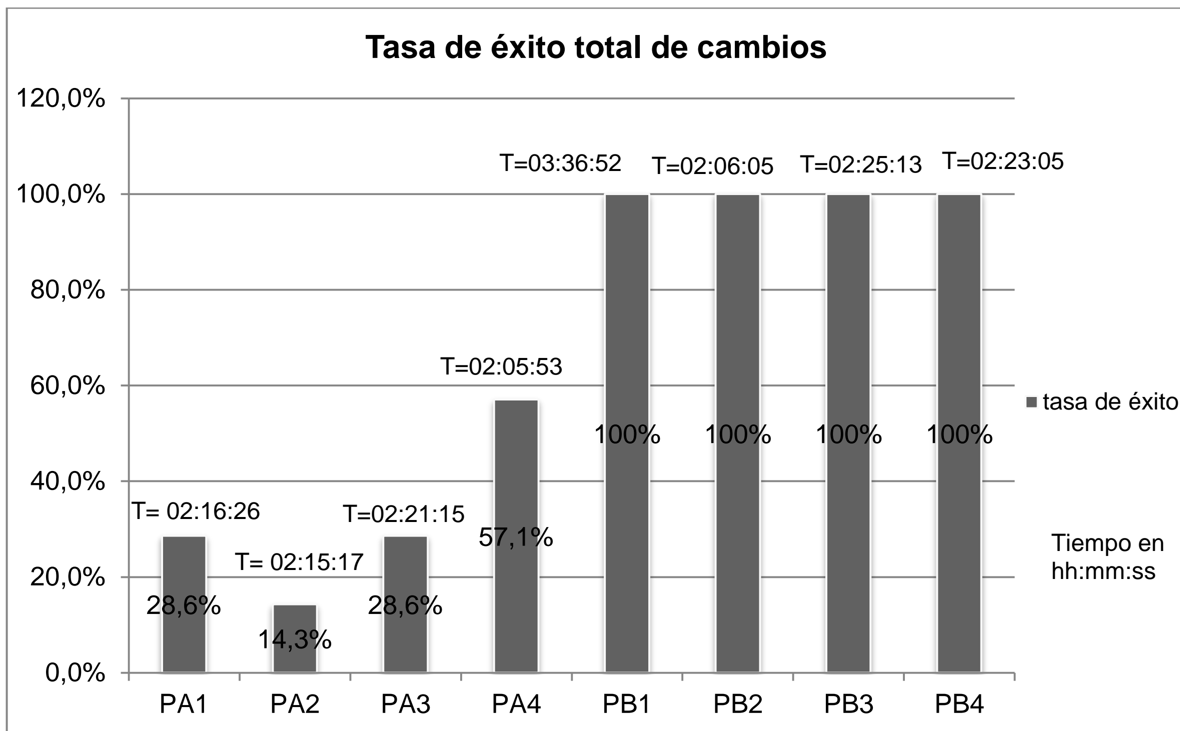

Gráfico 9. Tasa de éxito y tiempo totales de cambios

El Gráfico 9 presenta el tiempo total y la tasa de éxito para cada participante al realizar los cambios solicitados, en esta se puede observar que el participante del grupo A que realizó los cambios en menor tiempo fue PA4 con 02:05:53 mientras que para el grupo B el participante con menor tiempo fue PB2 con 02:06:05. A pesar de que estos dos tiempos difieren por poco (00:00:12), la tasa de éxito del participante PA4 (57,1%) es considerablemente menor que la del participante PB2 (100%), lo cual indica que el tiempo invertido por el participante PA4 no se ve reflejado en los resultados obtenidos al realizar las modificaciones. Al realizar este mismo análisis con el participante que obtuvo el segundo menor tiempo en cada uno de los grupos, se puede notar que el participante del grupo B PB4 invirtió mayor tiempo (02:23:03) con respecto al participante del grupo A PA2 (02:15:17), sin embargo logró realizar correctamente todos los cambios solicitados, lo cual se ve reflejado en su tasa de éxito (100%) que es mucho mayor a la del participante PA2 (14,3%).

Haciendo el mismo análisis para los participantes restantes de los dos grupos, se observa el mismo comportamiento, lo cual indica que la inclusión de las prácticas base del modelo de referencia al proceso de construcción permite que el tiempo invertido al realizar cambios al producto software genere los resultados esperados.

PA3: ¿Utilizar las prácticas base del proceso de construcción del modelo de referencia MANTuS permite aumentar la tasa de éxito en el encuentro de la causa de fallos?

La tasa de éxito en el encuentro de causa de fallos (TEF) fue hallada para cada uno de los participantes teniendo en cuenta los resultados del ítem 5 planteado en la guía, registrados en la Tabla 12. Los resultados obtenidos se presentan en la Tabla 23. Para esto se tuvo en cuenta la métrica tasa de éxito en el encuentro de fallos planteada en la norma ISO/IEC 9126 con la fórmula:

$TEF = 1 - (A/B), \quad 0 \leq TEF \leq 1$, entre más cercano a 1 mejor

Donde:

A = número de fallos cuyas causas no se han encontrado.

B = número de fallos registrados.

	Participante	A	B	$TEF = 1 - (A/B)$
Código A	PA1	0	3	1
	PA2	3	3	0
	PA3	3	3	0
	PA4	3	3	0
Código B	PB1	0	3	1
	PB2	0	3	1
	PB3	0	3	1
	PB4	0	3	1

Tabla 23. Cálculo TEF

En la Tabla 23 se observa que solo un participante (PA1) que trabajó con el código A pudo encontrar correctamente las causas a los tres fallos presentados, aunque todos afirmaron haberlas encontrado, al revisar las descripciones de las causa especificadas por cada uno de ellos, se observó que indicaban causas de fallos en zonas del código que no generaban algún tipo de problema. Por otra parte, los cuatro participantes que modificaron el código B encontraron todas las causas de los fallos presentados, manifestando que la documentación, indentación y separación del código por funciones fueron de gran ayuda para encontrar la causa del fallo.

Los valores obtenidos para TEF en la Tabla 23 indican que la tasa de éxito en el encuentro de causa de fallos para el grupo A es del 25% y para el grupo B es 100%. La representación gráfica de estos resultados se encuentra en el Gráfico 10, donde se puede verificar que incluir las prácticas base del modelo de referencia al proceso de construcción permite aumentar la tasa de éxito en el encuentro de la causa de fallos.

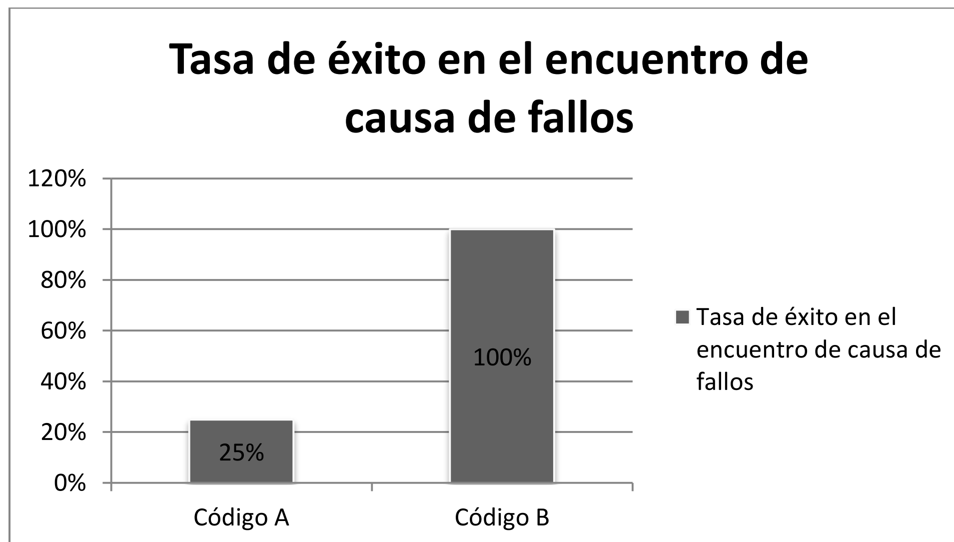

Gráfico 10. Tasa de éxito en el encuentro de causa de fallos

PA4: ¿Disminuye el tiempo de análisis de fallos con la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS?

El tiempo de análisis de fallos se halló para los ítems 4, 5 (que incluye tres fallos), 7, 8 planteados en la guía, haciendo uso de la métrica tiempo de análisis de fallo establecida en la norma ISO/IEC 9126 que se calcula mediante la fórmula:

$TAF = Sum(T)/N, \; 0 \leq TAF$, entre más pequeño sea TAF mejor

$T = Tout - Tin$

Donde:

Tout = tiempo en el que las causas del fallo son encontradas. Para este caso el Tout es la hora de encuentro de código a modificar indicado por los participantes en cada ítem de la guía y que se encuentra en la Tabla 12.

Tin = tiempo en el que el reporte del fallo es recibido. Para este caso el Tin es la hora de inicio de análisis de los fallos indicado por los participantes en cada ítem de la guía y que se encuentra en la Tabla 12.

N: número de fallos registrados.

Debido a que el número de fallos cuyas causas se deben hallar para cada ítem planteado en la guía es uno, al reemplazar N la métrica queda como:

$TAF = Tout - Tin$

Para esta métrica la norma ISO/IEC 9126 recomienda excluir el número de fallos cuyas causas no se han encontrado, por esta razón el análisis presentado excluye los datos donde el valor para la fila *Correcto* es no (ver Tabla 12). Debido a que ninguno de los participantes del grupo A realizó correctamente los cambios para los ítems 7 y 8, no fue posible realizar el análisis comparativo entre los resultados del grupo A y B.

En la Tabla 24 se muestran el valor de TAF hallado para cada participante en cada uno de los ítems de la guía, este es el mismo valor de *Duración* calculado en Tabla 12.

Participante	TAF ítem 4 (hh:mm:ss)	TAF ítem 5.1 (hh:mm:ss)	TAF ítem 5.2 (hh:mm:ss)	TAF ítem 5.3 (hh:mm:ss)
PA1	0:00:50	0:18:39	0:09:43	0:02:10
PA2	0:01:10			
PA3	0:01:40			
PA4	0:00:50			
PB1	0:00:30	0:48:46	0:01:08	0:00:51
PB2	0:00:48	0:05:00	0:01:40	0:00:30
PB3	0:00:47	0:13:50	0:01:15	0:01:11
PB4	0:00:37	0:05:45	0:00:00	0:01:28

Tabla 24. Cálculo TAF

A partir de los TAF mostrados en la Tabla 24, para cada ítem se realizó una línea de tiempo donde se ubican los TAF obtenidos por cada uno de los participantes que realizaron correctamente el ítem. El análisis para cada ítem es presentado a continuación:

Ítem 4: Cuando se crea una cuenta, no solicitar el valor del saldo al usuario, fijar este valor en $ 50000 para cuentas corrientes y $ 70000 para cuentas de ahorros.

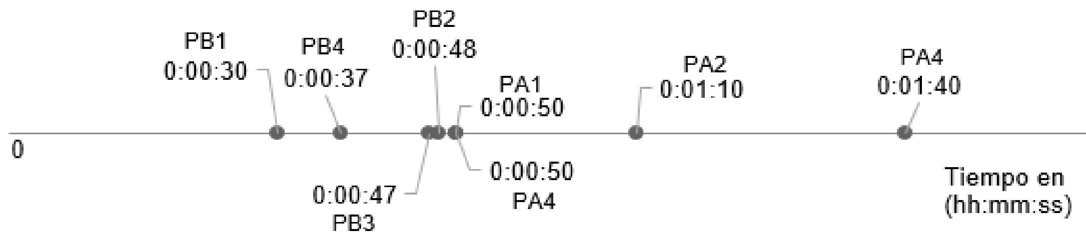

Gráfico 11. Línea de tiempo ítem 4

En el Gráfico 11 se observa que el participante que encontró el código a modificar en menor tiempo es PB1 con 30 segundos que pertenece al grupo B y en general todos los participantes del grupo B tuvieron un tiempo inferior que el menor tiempo del grupo A el cual es de 50 segundos. Aunque el cambio presentaba la misma dificultad para encontrar el código a modificar, los participantes del grupo A manifestaron que fue fácil de encontrar debido al conocimiento que ya habían adquirido del código, mientras que los participantes del grupo B afirmaron que fue sencillo debido a los comentarios y la organización del código.

Ítem 5.1: Se consigna en una cuenta de ahorro y el programa falla al consultar la opción visualizar datos del cliente.

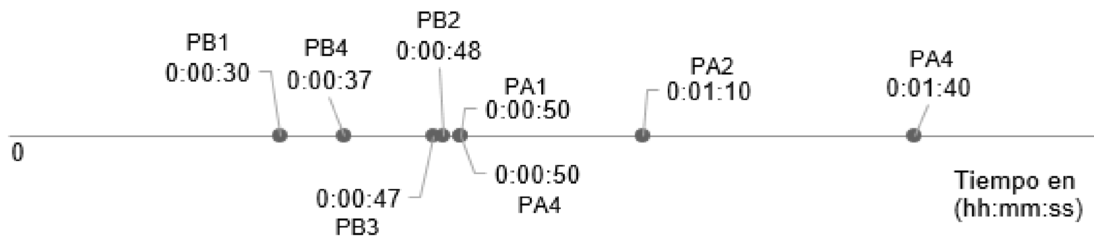

Gráfico 12. Línea de tiempo ítem 5.1

En el Gráfico 12 se puede observar que la mayoría de los participantes del grupo B encontraron correctamente la causa del fallo en un tiempo inferior respecto al único participante del grupo A que pudo encontrarla (participante PA1), el cual lo hizo en un tiempo de 00:18:29. Es importante resaltar que el participante PA1 solo encontró la zona general donde se encontraba el fallo, mas no detalló específicamente cual era la lógica y parte del código que generaba el error, aspecto que si fue hallado por todos los participantes del grupo B y que como se mencionó anteriormente la mayoría lo hicieron en menor tiempo. El participante PB1 fue el que más tiempo invirtió en el encuentro de la causa de fallos, gastándose incluso más tiempo que el participante PA1, pero esto se debe a que el participante PB1 encontró más fallos de los que estaban presentes en la guía, invirtiendo más tiempo en analizar, entender y encontrar la parte del código donde se presentaban los fallos.

Ítem 5.2: Se retira dinero de las dos cuentas y al consultar las dos últimas transacciones los resultados son erróneos.

Gráfico 13. Línea de tiempo ítem 5.2

En el Gráfico 13 se puede observar que solamente un participante del grupo A (participante PA1) pudo encontrar con éxito cual era la causa del fallo, mientras que todos los participantes del grupo B pudieron hacerlo. Además de esto es posible ver que todos los participantes del grupo B lograron encontrar la causa en un tiempo menor que PA1 quien se tardó 00:09:43 y como se mencionó anteriormente no encontró la causa exacta. Es importante resaltar que los participantes del grupo B identificaron la causa en un corto tiempo debido a que, como ellos lo manifestaron, esta era la misma que provocaba el fallo del ítem 5.1, es por esto que el participante PB4 tiene un tiempo de 0 ya que identificó inmediatamente que al resolver el fallo en 5.1, el fallo en 5.2 también desaparecía.

Ítem 5.3 Cuando se hace una transferencia de dinero y se consultan las últimas dos transacciones el sistema no arroja los resultados esperados.

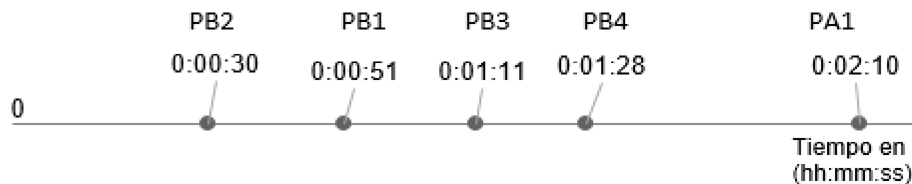

Gráfico 14. Línea de tiempo ítem 5.3

En el Gráfico 14 se observa que todos los participantes del grupo B encontraron exitosamente la causa del fallo y lo hicieron en un tiempo inferior al único participante del grupo A que pudo encontrarla (PA1), el cual necesito un tiempo de 00:02:10 y como se dijo anteriormente no encontró la causa con precisión. Los participantes del grupo B indicaron que pudieron identificar la causa del fallo con facilidad ya que hacia parte de la misma función que resolvieron encontrando los fallos anteriores.

Con todo el análisis anterior se puede notar que el tiempo de análisis de fallos disminuye con la inclusión de las prácticas base del modelo de referencia al proceso de construcción.

PA5: ¿Disminuyen los costos con la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS?

Con el fin de comparar los costos para cada uno de los grupos, se utilizó la siguiente fórmula:

$$Costo\ Total = FET * Costo\ de\ hora$$

Donde:

FET = Factor de éxito en el tiempo, calculado mediante la fórmula:

$$FET = \frac{T}{E}$$

Con:

T = tiempo invertido al realizar las modificaciones.

E = éxito al realizar las modificaciones, calculado mediante la relación entre el número de cambios realizados correctamente y los cambios totales realizados ($E = \frac{Cambios\ correctos}{Cambios\ totales}$).

Teniendo en cuenta los datos presentados en la Tabla 22, donde se observa el tiempo total invertido por cada participante tanto en el análisis como en la realización de las modificaciones planteadas en la guía, se calculó el tiempo total empleado por cada uno de los grupos al realizar los cambios.

Para el Grupo A se tiene:

$$T_A = (2{:}16{:}26 + 2{:}15{:}17 + 2{:}21{:}15 + 2{:}05{:}53)\ hh{:}mm{:}ss$$

$$T_A = 8{:}58{:}51\ hh{:}mm{:}ss \sim 9\ h$$

Y para el Grupo B:

$$T_B = (3{:}36{:}52 + 2{:}06{:}05 + 2{:}25{:}13 + 2{:}23{:}05)\ \ hh{:}mm{:}ss$$

$$T_B = 10{:}31{:}15 \sim 10{,}5\ h$$

Con respecto al éxito de cada uno de los grupos, de acuerdo con los datos presentados en Tabla 20 y Tabla 21 el éxito para el Grupo A (*E_A*) y para el Grupo B (*E_B*) son:

$$E_A = \frac{9}{28} \qquad y \qquad E_B = \frac{28}{28}$$

Con estos datos es posible calcular el costo total para cada uno de los grupos.

Grupo A

$$Costo\ total_A = \frac{9h}{(9/28)} * Costo\ de\ hora\ \$/h$$

$$Costo\ total_A = 28h * Costro\ de\ hora\ \$/h$$

Grupo B

$$Costo\ total_B = \frac{10{,}5h}{(28/28)} * Costo\ de\ hora\ \$/h$$

$$Costo\ total_B = 10{,}5h * Costro\ de\ hora\ \$/h$$

De lo anterior se observa que independiente del valor del *Costo de hora,* el costo para el Grupo A es mayor al costo para el Grupo B, lo cual indica que sí disminuyen los costos con la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS.

Análisis de la encuesta

La Tabla 25 resume las respuestas dadas por los participantes a las preguntas realizadas en la encuesta.

Pregunta	Grupo A		Grupo B	
	Si	No	Si	No
1) ¿Le resulto fácil realizar las modificaciones?	3	1	4	0
2) ¿Le pareció fácil encontrar las causas de los fallos?	3	1	4	0
3) ¿Le pareció organizado el código?	1	3	4	0
4) ¿Le resulto fácil entender el código?	2	2	4	0
5) ¿Le resulto fácil leer el código?	1	3	4	0
6) ¿El código estaba bien documentado?	0	4	4	0
7) ¿los variables tenían nombres claros y entendibles?	2	2	4	0
8) ¿Encontró código duplicado el en programa?	4	0	0	4
9) Esta situación le afecto en la realización de los cambios solicitados.	4	0		
10) La mantenibilidad de software es la capacidad del producto software para ser modificado efectiva y eficientemente. En su opinión ¿el código tiene alta mantenibilidad?	0	4	4	0

Tabla 25. Tabla respuestas encuesta

Para la pregunta 1, el 25% de los participantes del grupo A manifestó que los cambios no fueron sencillos de realizar ya que el código no estaba bien estructurado, el 75% restante afirmo que fue fácil realizar las modificaciones ya que las funcionalidades eran sencillas, sin embargo resaltaron que se presentaron dificultades al analizar donde se debían hacer los cambios porque el código contaba con muchos condicionales anidados. Uno de ellos indicó que no pudo resolver los cambios implicaban mayor complejidad. Pese a estas opiniones en el análisis de funcionalidad presentado anteriormente se pudo observar que no todos los cambios fueron realizados con éxito en este grupo. En cuanto a los participantes del grupo B todos afirmaron que las modificaciones fueron sencillas de realizar, debido a diferentes aspectos como: la estructura modular del código que separaba la complejidad de la declaración e implementación de las clases y la buena documentación e indentación del código.

En la pregunta 2, el 75% de los integrantes del grupo A afirmó que les pareció fácil encontrar las causas de los fallos debido a que el código era relativamente sencillo, a que hicieron seguimiento por consola y al código. Al 25% de los participantes de este grupo le resultó difícil encontrar las causas porque el código no estaba bien estructurado. Es importante resaltar que algunas personas creyeron haber encontrado las causas de los fallos, pero realmente no lo hicieron, encontrando supuestos errores que al modificarlos no corregían los fallos presentados. Por otra parte, a todos los participantes del grupo B les pareció fácil encontrar las causas de los fallos, esto lo atribuyeron a aspectos como: el orden y la documentación del código, lo cual sirvió de guía para encontrar algunas funcionalidades al solo seguir los comentarios.

Respecto a la pregunta 3 solo una persona del grupo A dijo que el código estaba organizado, pero comentó que no era fácil saber dónde cerraban las llaves, el resto de participantes afirmaron que el código no estaba bien estructurado porque era muy difícil saber dónde terminaban las sentencias, no se manejan funciones o métodos que

facilitaran el programa, presentaba mucho código repetido y no existía mucha documentación lo cual dificultaba la localización de algunas cosas del código. Por otra parte, todas los personas del grupo B afirmaron que el código si estaba bien estructurado porque contaba con buenas prácticas como la documentación, la indentación, organización, separación entre las declaraciones y la implementación y uso adecuado de instrucciones como switch.

Para la pregunta 4, dos de los integrantes del grupo A concluyeron que el código era fácil de entender porque las funciones no eran complejas, aunque se dificultaba entender donde terminaba cada sentencia. Las otras dos personas de este grupo afirmaron que era difícil de entender porque existían algunas funciones cuya implementación era compleja, el programa presentaba mucho código repetido y además contaba con variables mal nombradas, cuyo nombre no indicaba para que servían. El no entender bien el código pudo resultar clave para que los participantes no encontraran las causas específicas de los fallos y por ende no lograran resolverlas adecuadamente. Respecto al grupo B todos sus integrantes resaltaron que el programa era fácil de entender porque las funcionalidades eran sencillas y el código contaba con buena indentación, documentación y organización.

En la pregunta 5, el 75% de los participantes del grupo A concluyeron que el código no era fácil de leer debido a la estructura del programa, ya que todo la implementación estaba en el main y no existían funciones o métodos, lo cual hacía confuso el código, además los nombres de las variables y la falta de documentación dificultaba la lectura del mismo. Sólo un participante de este grupo afirmó que el código era fácil de leer y esto lo atribuyó a que el programa no requería mucho conocimiento de la plataforma. Por otra parte, a todos los participantes del grupo B les resultó fácil de leer el código debido a diferentes aspectos como: orden en el main, declaración de clases, implementación de métodos, indentación apropiada, variables y métodos explícitos y los comentarios ya que explicaban bien las funcionalidades.

Respecto a la pregunta 6, todos los participantes del grupo A opinaron que el código no presentaba buena documentación, ellos afirmaron que la falta de comentarios dificultó el entendimiento de algunas funciones complejas que eran de vital importancia en el flujo del código. Por el contrario, todos los participantes que trabajaron con el código B afirmaron que éste tenía buena documentación, ya que los comentarios describían métodos, funcionalidades y declaraciones de variables, lo cual facilitó el entendimiento del código.

En la pregunta 7, el 50% de los integrantes del grupo A concluyó que las variables no tenían nombres claros y entendibles, afirmaron que existían nombres parecidos, cortos, y de una sola letra, que generaban confusión. El otro 50% de este grupo afirmó que las variables estaban bien nombradas, ya que sus nombres indicaban claramente para que servían. Por parte del grupo B, todos los participantes indicaron que los nombres de las variables eran entendibles y reflejaban el valor que representaban al igual que los nombres de los métodos, que representaban claramente la funcionalidad de los mismos.

Para las preguntas 8 y 9, todos los participantes del grupo A afirmaron haber encontrado código duplicado en el programa y manifestaron que esto afecto su desempeño ya que este código debía haber sido implementado en una función para así evitar realizar el mismo cambio en diferentes partes del código. Respecto al grupo B los participantes no encontraron código duplicado por lo que no respondieron la pregunta 9.

En la última pregunta relacionada con la mantenibilidad del producto, todos los participantes que modificaron el código A manifestaron que este no tiene alta mantenibilidad por diferentes aspectos como: la falta de documentación, que dificulta la modificación del código; la gran cantidad de condicionales, que impide realizar cambios mayores de forma sencilla; la ausencia de métodos y funciones. Además comentaron que el código puede ser simplificado para evitar la redundancia y los menús pueden ser incluidos en funciones para facilitar su modificación

Por el contrario, todos los participantes que trabajaron con el código B concluyeron que éste si tiene alta mantenibilidad debido que las funcionalidades estaban separadas correctamente y el código estaba documentado e indentado. Uno de los participantes afirmó que las clases tenían un bajo acoplamiento y una alta cohesión, lo cual permitía que un cambio, solo afectara a la parte directamente relacionada con él y no a todo el programa. Otro participante concluyó que la estructura del código puede ser extendida fácilmente para agregar nuevas funcionalidades.

En general se puede observar que el incluir al código prácticas como documentación, nombrado coherente, controlar abreviaciones, determinar y separar funcionalidades, ayudó a los participantes del grupo B a realizar correctamente todas las modificaciones planteadas en la guía, pues como ellos mismos lo resaltaron este tipo de prácticas les ayudó a entender, leer, analizar mejor el código para encontrar la parte a modificar y así mismo realizar dichos cambios. Por otra parte se analiza que el código A al no estar programado con este tipo de prácticas generó mayor dificultad en la modificación del código, ya que como los participantes de este grupo lo indicaron, no pudieron entender ni leer bien el código, lo cual hizo que los cambios no se hicieran de forma correcta.

4.8. Plan de validación

Con el fin de disminuir la subjetividad de los resultados del estudio de caso, la información obtenida a partir de la encuesta fue soportada mediante el análisis cuantitativo realizado (ver sección 4.7), donde partiendo de las métricas planteadas se llegó a resultados que reflejaban las opiniones de los participantes.

Para disminuir las amenazas a la validez del estudio de caso, se consideraron los siguientes aspectos:

Validez de conclusión: para evitar el riesgo de variación en los resultados debido a diferencias individuales de los sujetos de investigación los participantes elegidos fueron estudiantes de últimos semestres de Ingeniería de Sistemas de la Universidad del Cauca los cuales tenían conocimientos de programación orientada a objetos y manejo del lenguaje C++. Lo anterior permitió reducir la heterogeneidad en el grupo de estudio ya que ellos tenían conocimientos y antecedentes similares.

Validez de constructo: en la recolección de datos se tuvieron en cuenta tanto medidas de tiempo como las observaciones individuales de los participantes para cada modificación, lo cual permitió hacer un análisis más completo comparando estos dos aspectos.

Validez interna: la guía planteaba modificaciones sencillas para evitar que los participantes se aburrieran o cansaran rápidamente, evitando así que se afectaran los resultados del estudio. De igual forma, las plantillas de recolección de datos fueron diseñadas en forma sencilla y clara para evitar confusión.

4.9. Limitaciones del estudio

La selección de los participantes podría reducir la validez externa del estudio de caso ya que al ser estudiantes, los sujetos no son seleccionados de una población general, lo cual podría limitar el poder de generalización de los resultados obtenidos.

El número de participantes en el estudio de caso podría ser insuficiente para generalizar los resultados obtenidos.

A pesar de que todos los participantes en el estudio de caso contaban con conocimientos previos tanto en orientación a objetos y manejo del lenguaje C++, se observó inconformidad respecto al lenguaje seleccionado para el estudio, ya que muchos comentaron que hace mucho tiempo no trabajan con él.

CAPITULO V

CONCLUSIONES Y TRABAJO FUTURO

En este trabajo de grado se ha propuesto un modelo de referencia para la inclusión de sub-características de mantenibilidad al producto software durante el proceso de desarrollo, este fue elaborado a partir de: (i) la identificación de los atributos de software que influyen en la mantenibilidad del producto, (ii) la clasificasión de los atributos identificados de acuerdo a las sub-características de mantenibilidad y a los procesos en los que se presentan, (iii) el analisis de como potenciar cada sub-característica, (iv) la estructuración del modelo de referencia y (v) la evaluación del modelo de referencia realizada a traves de un caso de estudio. Al realizar la búsqueda de trabajos relacionados con mantenibilidad de software, específicamente de atributos de mantenibilidad se pudo notar que son pocos los estudios que abordan este tema, por lo cual se puede resaltar que el análisis y clasificación de atributos de mantenibilidad presentados es un aporte importante de esta investigación. De la apliacion del estudio de caso se puede concluir que la inclusión de las prácticas base del modelo de referencia al proceso de construcción permite potenciar la mantenibilidad del producto software, ya que la capacidad para ser analizado y modificado del producto aumentan considerablemente.

Adicionalmente, con el fin de divulgar el presente trabajo de investigación se han escrito dos artículos:

- *“Análisis y clasificación de atributos de mantenibilidad: una revisión desde la literatura”* el cual se envió a la revista entre ciencia e ingeniería categoría B.
- *“Generando productos software mantenibles desde el proceso de desarrollo: El modelo de referencia MANTuS”* que ha sido enviado a la revista Ingeniare categoría A1 y que se encuentra en revisión para su publicación.

En este capítulo se presentan las conclusiones del trabajo de investigación, algunas recomendaciones y sugerencias sobre posibles trabajos futuros.

5.1. Conclusiones

- La mantenibilidad de software es una característica de calidad que se debe considerar desde etapas tempranas del ciclo de vida del producto y durante todo el proceso de desarrollo de software.

- Los procesos diseño arquitectural de software y diseño detallado de software son los que tienen mayor influencia sobre la mantenibilidad del producto, ya que la clasificación de atributos de mantenibilidad realizada indica que la mayoría de los atributos se presentan durante estos procesos.

- Los atributos de mantenibilidad que influyen sobre el mayor número de sub-características de mantenibilidad de acuerdo a la clasificación realizada son: acoplamiento, cohesión, complejidad y simplicidad. Por otra parte, los atributos consistencia, documentación y trazabilidad se presentan en todos los flujos de trabajo, por lo cual deben ser considerados durante todo el proceso de desarrollo.

- Las sub-características capacidad para ser analizado y capacidad para ser modificado del producto son las que más impacto tienen sobre la mantenibilidad del producto ya que todos los 18 atributos definidos influyen sobre estas sub-características.

- Las prácticas base definidas para los procesos del modelo de referencia son sencillas, esto con el fin de no aumentar el esfuerzo durante el desarrollo, sin embargo, es importante resaltar éstas deben considerarse durante el proceso de desarrollo y no en etapas posteriores del ciclo de vida del producto.

- Debido a que el modelo de referencia MANTuS está compuesto por una vista general y una vista específica el usuario potencial del mismo puede utilizar de manera segmentada el modelo que necesite para potenciar una sub-característica de mantenibilidad específica.

- Los resultados obtenidos en el estudio de caso aplicado demuestran que la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS permite potenciar la mantenibilidad del producto software.

- El estudio de caso desarrollado evidencia que la utilización de las prácticas base del proceso de construcción del modelo de referencia MANTuS permite disminuir los costos de la etapa de mantenimiento de software.

- De la aplicación del estudio de caso se observó que la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS ayudó a que el tiempo invertido en el mantenimiento del producto se vea reflejado en los resultados obtenidos, lo cual podría permitir que exista más tiempo disponible para nuevos desarrollos.

- Los resultados del estudio de caso evidencian que la utilización de prácticas base del proceso de construcción del modelo de referencia MANTuS genera una alta tasa de éxito en el análisis de código y en las modificaciones realizadas, lo cual indica que las sub-características capacidad para ser modificado y analizado del producto aumentan.

5.2. Lecciones aprendidas

- Es importante definir una estrategia de búsqueda con criterios de inclusión y exclusión que permita realizar una búsqueda detallada de la bibliografía para obtener los trabajos más relevantes de la temática a investigar. Si es necesario, se recomienda realizar una búsqueda manual, revisando las referencias de los artículos relevantes.

- El alcance del proyecto debe ser claro desde el inicio de la investigación, es decir, tener objetivos claros y concisos que puedan ser alcanzados durante el tiempo definido en el calendario del proyecto.

- Para el desarrollo de una investigación es importante contar desde el principio con una metodología de trabajo definida, a partir de la cual se empiecen a estructurar elementos que le den mayor formalidad a todo el trabajo de investigación. Esta metodología se debe tener clara desde el inicio de la investigación y debe estar enfocada en los objetivos del proyecto.

- Las afirmaciones teóricas deben ser sustentadas con referencias bibliográficas confiables con el fin de tener argumentos válidos en el desarrollo de la investigación.

- Es importante anexar a la investigación, todos los artefactos generados de tal forma que éstos sirvan como soporte a los resultados finales obtenidos y como apoyo a las conclusiones generadas al final de la investigación.

- La utilización del método estudio de caso sirve como un primer acercamiento a la aplicación de un planteamiento teórico en un ambiente real, dando la posibilidad de evaluar para poder identificar fortalezas y aspectos a mejorar.

- Antes de la ejecución del estudio de caso es de vital importancia definir claramente las medidas que se van a utilizar para realizar la recolección y análisis de los datos con el fin de dar respuestas a las preguntas de investigación planteadas.

- Las plantillas de recolección de datos deben ser diseñadas en forma clara y sencilla con el fin de evitar confusión en los participantes del estudio de caso y facilitar el análisis de los datos recolectados.

5.3. Trabajo futuro

Por cuestiones de tiempo y alcance algunos aspectos no fueron considerados en este trabajo de investigación y se plantean como trabajos que pueden ser abordados en el futuro.

- Se ha iniciado un estudio de caso con el fin de evaluar todo el modelo de referencia propuesto. Éste se está llevando a cabo en la asignatura Proyecto II del programa de Ingeniería de sistemas de la Universidad del Cauca, incorporando las prácticas del modelo de referencia a tres procesos de desarrollo diferentes (Scrum-XP, Tutelcan, UP-VSE).

- Realizar la validación del modelo de referencia propuesto en un entorno empresarial mediante un estudio de caso.

- Aplicar el modelo de referencia propuesto en una organización desarrolladora de software que desee obtener la certificación de mantenibilidad realizada por AQC, para validar si inclusión de las prácticas definidas permiten alcanzar este fin.

- Se pretende distribuir el modelo de referencia MANTuS a diferentes actores relacionados con el tema de mantenibilidad de software, como las empresas AQC y Fiding, con el fin de comenzar a lograr el consenso del mismo.

- Debido al enfoque presentado por este trabajo, donde se consideraron únicamente atributos de mantenibilidad que se presentan durante el proceso de desarrollo, otros factores relevantes para el mantenimiento del producto como control de versiones, herramientas de soporte, tiempo de vida del producto, entre otros, han quedado por fuera del mismo. Es por esto que se plantea como trabajo futuro la definición de prácticas que permitan abordar estos aspectos en las diferentes etapas del ciclo de vida del producto a las que correspondan.

REFERENCIAS BIBLIOGRÁFICAS[5]

[1] J. M. Conejero, E. Figueiredo, A. Garcia, J. Hernández, and E. Jurado, "On the relationship of concern metrics and requirements maintainability," *Information and Software Technology,* vol. 54, pp. 212-238, 2012.

[2] C. E. H. Chua, S. Purao, and V. C. Storey, "Developing maintainable software: The Readable approach," *Decision Support Systems,* vol. 42, pp. 469-491, 2006.

[3] J.-C. Chen and S.-J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *Journal of Systems and Software,* vol. 82, pp. 981-992, 2009.

[4] F. J. Pino, F. Ruiz, F. García, and M. Piattini, "A software maintenance methodology for small organizations: Agile MANTEMA," *Journal Of Software Maintenance And Evolution: Research And Practice,* vol. 24, pp. 851-876, 2011.

[5] I. S. Organization, "ISO/IEC 25010," in *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models*, ed, 2011.

[6] E. B. Swanson, "IS “maintainability”: should it reduce the maintenance effort?," *ACM SIGMIS Database,* vol. 30, pp. 65-76, 1999.

[7] B. Kumar, "A survey of key factors affecting software maintainability," in *2012 International Conference on Computing Sciences*, 2012, pp. 261-266.

[8] (22-03-2014). *AQC Lab: Laboratorio de Evaluación de la Calidad Software*. Available: http://www.alarcosqualitycenter.com/index.php/laboratorio-evaluacion-calidad-software

[9] I. S. Organization, "ISO/IEC 15504-5," in *An exemplar software life cycle process assessment model*, ed, 2011.

[10] F. J. Pino, M. Piattini, and G. Travassos, "Managing and Developing Distributed Research Projects by Means of Action-Research," *Rev. Fac. Ing. Univ. Antioquia,* vol. 68, pp. 61-74, 2010.

[11] J. McNiff, *Action research: Principles and practice*, 3rd ed. New York, USA: Routledge, 2013.

[12] IEEE, "IEEE Std 1219," in *IEEE Standard for Software Maintenance*, ed, 1998.

[13] I. S. Organization, "ISO/IEC 33004," in *Information technology - Process assessment - Requirements for Process Referents, Process Assessment and Organizational Maturity Models*, ed, 2012.

[14] I. S. Organization, "ISO/IEC 12207," in *Systems and software engineering — Software life cycle processes*, ed, 2008.

[15] J. Verdugo and M. Rodriguez, "Evaluación de la Calidad de Aplicaciones Software," *Dintel,* vol. 7, pp. 148 - 150, 2013.

[16] F. J. Pino, F. García, and M. Piattini, "Software process improvement in small and medium software enterprises: a systematic review," *Software Quality Journal,* vol. 16, pp. 237-261, 2008.

[17] A. April, J. Huffman Hayes, A. Abran, and R. Dumke, "Software Maintenance Maturity Model (SMmm): the software maintenance process model," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 17, pp. 197-223, 2005.

[18] M. Polo, M. Piattini, and F. Ruiz, "Improving the Quality of the Maintenance Process," in *The Second World Congress for Software Quality*, Pacifico Yokohama Conference Center (Tokyo Bay Arena), 2000, pp. 325 - 330.

[5] Estas fuentes han sido documentadas utilizando el estilo bibliográfico IEEE

[19] M. Polo, M. Piattini, F. Ruiz, and C. Calero, "MANTEMA: A complete rigorous methodology for supporting maintenance based on the ISO/IEC 12207 Standard," in *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*, 1999, pp. 178-181.

[20] D. Kozlov, J. Koskinen, M. Sakkinen, and J. Markkula, "Assessing maintainability change over multiple software releases," *Journal of Software Maintenance and Evolution,* vol. 20, pp. 31-58, 2008.

[21] K. Hashim and E. Key, "A software maintainability attributes model," *Malaysian Journal of Computer Science,* vol. 9, pp. 92-97, 1996.

[22] P. Hegedus, D. Bán, R. Ferenc, and T. Gyimóthy, "Myth or reality? Analyzing the effect of design patterns on software maintainability," *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity,* vol. 340 pp. 138-145, 2012.

[23] J. Bosch and P. Bengtsson, "Assessing optimal software architecture maintainability," in *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, 2001, pp. 168-175.

[24] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, 2007, pp. 30-39.

[25] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, "Coupling metrics for predicting maintainability in service-oriented designs," in *Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07)*, 2007, pp. 329-338.

[26] S. K. Dubey and A. Rana, "Assessment of maintainability metrics for object-oriented software system," *ACM SIGSOFT Software Engineering Notes,* vol. 36, pp. 1-7, 2011.

[27] W. Hordijk and R. Wieringa, "Surveying the factors that influence maintainability: research design," in *ACM SIGSOFT Software Engineering Notes*, 2005, pp. 385-388.

[28] S. C. Misra, "Modeling design/coding factors that drive maintainability of software systems," *Software Quality Journal,* vol. 13, pp. 297-320, 2005.

[29] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, 2012, pp. 306-315.

[30] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal,* vol. 20, pp. 287-307, 2012.

[31] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Software Maintenance, 1992. Proceerdings., Conference on*, 1992, pp. 337-344.

[32] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 367-377.

[33] B. Anda, "Assessing software system maintainability using structural measures and expert assessments," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, 2007, pp. 204-213.

[34] K. D. Welker, "The software maintainability index revisited," *CrossTalk,* pp. 18-21, 2001.

[35] M. Broy, F. Deissenboeck, and M. Pizka, "Demystifying maintainability," in *Proceedings of the 2006 international workshop on Software quality*, 2006, pp. 21-26.

[36] M. Pizka and F. Deißenböck, "How to effectively define and measure maintainability," *SMEF 2007,* p. 93, 2007.

[37] K. Aggarwal, Y. Singh, P. Chandra, and M. Puri, "Measurement of software maintainability using a fuzzy model," *Journal of Computer Science,* vol. 1, p. 538, 2005.

[38] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Reliability and maintainability symposium, 2002. Proceedings. Annual*, 2002, pp. 235-241.

[39] M. Vokáč, W. Tichy, D. I. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment," *Empirical Software Engineering,* vol. 9, pp. 149-195, 2004.

[40] IEEE, "IEEE Std 1471-2000," in *IEEE Recommened Practice for Architectural Description of Software-Intensive systems*, ed, 2000.

[41] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan, "How does Context Affect the Distribution of Software Maintainability Metrics?," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, 2013, pp. 350-359.

[42] Y. Zou and K. Kontogiannis, "Migration to object oriented platforms: A state transformation approach," in *Software Maintenance, 2002. Proceedings. International Conference on*, 2002, pp. 530-539.

[43] P. J. Vázquez Escudero, M. García, M. Navelonga, and F. J. García Peñalvo, "Métricas orientadas a objetos," 2001.

[44] S. Khalid, S. Zehra, and F. Arif, "Analysis of object oriented complexity and testability using object oriented design metrics," in *Proceedings of the 2010 National Software Engineering Conference*, 2010, p. 4.

[45] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. Von Staa, "On the reuse and maintenance of aspect-oriented software: An assessment framework," in *Proceedings of Brazilian Symposium on Software Engineering*, 2003, pp. 19-34.

[46] M. Bruntink and A. Van Deursen, "Predicting class testability using object-oriented metrics," in *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, 2004, pp. 136-145.

[47] I. S. Organization, "ISO/IEC 9126-1," in *Software Engineering – Product Quality – Part 1: Quality Model*, ed, 2001.

[48] F. Brito e Abreu and W. Melo, "Evaluating the impact of object-oriented design on software quality," in *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, 1996, pp. 90-99.

[49] M. Sefika, A. Sane, and R. H. Campbell, "Monitoring compliance of a software system with its high-level design models," in *Proceedings of the 18th international conference on Software engineering*, 1996, pp. 387-396.

[50] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *Software Engineering, IEEE Transactions on,* vol. 29, pp. 297-310, 2003.

[51] E. Capra, C. Francalanci, and F. Merlo, "An empirical study on the relationship between software design quality, development effort and governance in open source projects," *Software Engineering, IEEE Transactions on,* vol. 34, pp. 765-782, 2008.

[52] L. Etzkorn and H. Delugach, "Towards a semantic metrics suite for object-oriented design," in *Technology of Object-Oriented Languages and Systems, 2000. TOOLS 34. Proceedings. 34th International Conference on*, 2000, pp. 71-80.

[53] M. Badri, L. Badri, and F. Touré, "Empirical Analysis of Object-Oriented Design Metrics: Towards a New Metric Using Control Flow Paths and Probabilities," *Journal of Object Technology,* vol. 8, pp. 123-142, 2009.

[54] M. Ajrnal Chaumun, H. Kabaili, R. K. Keller, F. Lustman, and G. Saint-Denis, "Design properties and object-oriented software changeability," in *Software Maintenance and Reengineering, 2000. Proceedings of the Fourth European*, 2000, pp. 45-54.

[55] P. S. Sandhu and H. Singh, "A Reusability Evaluation Model for OO-Based Software Components," *International Journal of Computer Science,* vol. 1, pp. 259-264, 2006.

[56] P. S. Sandhu, P. Blecharz, and H. Singh, "A Taguchi approach to investigate impact of factors for reusability of software components," *Transactions on Engineering, Computing and Technology,* vol. 19, pp. 135-140, 2007.

[57] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information Systems Journal,* vol. 12, pp. 43-60, 2002.

[58] C. Gautam and S. S. Kang, "COMPARISON and IMPLEMENTATION of COMPOUND MEMOOD MODEL and MEMOOD MODEL," *International journal of computer science,* vol. 2, pp. 2394-2398.

[59] A. B. Al-Badareen, Z. Muda, M. A. Jabar, J. Din, and S. Turaev, "Software quality evaluation through maintenance processes," in *NAUN Conference of CONTROL*, 2010.

[60] J. M. Bieman and B.-K. Kang, "Measuring design-level cohesion," *Software Engineering, IEEE Transactions on,* vol. 24, pp. 111-124, 1998.

[61] H. Kabaili, R. K. Keller, and F. Lustman, "Cohesion as changeability indicator in object-oriented systems," in *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, 2001, pp. 39-46.

[62] K. Sirbi and P. J. Kulkarni, "Impact of Aspect Oriented Programming on Software Development Quality Metrics," *Global Journal of Computer Science and Technology,* vol. 10, 2010.

[63] G. Shahmohammadii and S. Jaliliii, "A Quantitative Evaluation of Maintainability of Software Architecture Styles."

[64] M. Ó. Cinnéide, D. Boyle, and I. H. Moghadam, "Automated refactoring for testability," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, 2011, pp. 437-443.

[65] N. Bawane and C. Srikrishna, "A novel method for quantitative assessment of software quality," *International Journal of Computer Science and Security,* vol. 3, p. 508, 2010.

[66] S. Chawla and R. Nath, "Evaluating Inheritance and Coupling Metrics," *International Journal of Engineering Trends and Technology,* vol. 4, pp. 2903-2908, 2013.

[67] A. B. AL-Badareen, M. H. Selamat, M. A. Jabar, J. Din, and S. Turaev, "The impact of software quality on maintenance process," *International Journal of Computers,* vol. 5, pp. 183-190, 2011.

[68] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, "Design-code traceability for object-oriented systems," *Annals of Software Engineering,* vol. 9, pp. 35-58, 2000.

[69] L. C. Briand, "Software documentation: how much is enough?," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*, 2003, pp. 13-15.

[70] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner*, et al.*, "Can clone detection support quality assessments of requirements specifications?," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010, pp. 79-88.

[71] B. S. Baker, "Parameterized duplication in strings: Algorithms and an application to software maintenance," *SIAM Journal on Computing,* vol. 26, pp. 1343-1362, 1997.

[72] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Static Analysis*, ed: Springer, 2001, pp. 40-56.

[73] J. P. Correia, Y. Kanellopoulos, and J. Visser, "A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 2009, pp. 61-70.

[74] P. Meananeatra, E. Rattanaleadnusorn, S. Rongviriyapanish, T. Kitcharoensup, T. Wisuttikul, and B. Charoendouysil, "Modeling code analyzability at method level in J2EE applications," in *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, 2013, pp. 61-66.

[75] H. Koziolek, "Sustainability evaluation of software architectures: a systematic review," in *Proceedings of the joint ACM SIGSOFT conference--QoSA and ACM SIGSOFT symposium--ISARCS on Quality of software architectures--QoSA and architecting critical systems--ISARCS*, 2011, pp. 3-12.

[76] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 73-82.

[77] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *Software Engineering, IEEE Transactions on,* vol. 36, pp. 546-558, 2010.

[78] M. Ilyas, M. Abbas, and K. Saleem, "A Metric Based Approach to Extract, Store and Deploy Software Reusable Components Effectively," *IJCSI International Journal of Computer Science Issues,* vol. 10, pp. 257-264, 2013.

[79] J. Kumar Chhabra, K. Aggarwal, and Y. Singh, "Code and data spatial complexity: two important software understandability measures," *Information and software Technology,* vol. 45, pp. 539-546, 2003.

[80] H. P. Breivold and I. Crnkovic, "Analysis of software evolvability in quality models," in *Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on*, 2009, pp. 279-282.

[81] R. Harrison, S. Counsell, and R. Nithi, "Experimental assessment of the effect of inheritance on the maintainability of object-oriented systems," *Journal of Systems and Software,* vol. 52, pp. 173-179, 2000.

[82] J. A. Díaz Pace and M. R. Campo, "Analyzing the role of aspects in software design," *Communications of the ACM,* vol. 44, pp. 66-73, 2001.

[83] P. Nikfard, M. K. Najafabadi, B. D. Rouhani, F. Nikpay, and H. B. Selamat, "An Empirical Analysis of a Testability Model," in *Informatics and Creative Multimedia (ICICM), 2013 International Conference on*, 2013, pp. 63-69.

[84] S. F. Ahmad, M. R. Beg, and M. Haleem, "A Comparative Study of Software Quality Models," *International Journal of Science, Engineering and Technology Research,* vol. 2, pp. pp: 172-176, 2013.

[85] T. Kuipers, J. Visser, and G. De Vries, "Monitoring the quality of outsourced software," in *Proc. Int. Workshop on Tools for Managing Globally Distributed*

Software Development (TOMAG 2007). Center for Telematics and Information Technology (CTIT), The Netherlands, 2007.

[86] R. K. Singh, A. Asthana, and D. K. Singh, "Approach to Software Maintainability Prediction Versus Performance," *International Journal of Soft Computing and Engineering,* vol. 2, pp. 51-54, 2012.

[87] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the relation between changeability decay and the characteristics of clones and methods," in *Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, 2008, pp. 100-109.

[88] L. Antonelli and A. Oliveros, "Fuentes Utilizadas por desarrolladores de Software en Argentina para Elicitar Requerimientos," in *WER*, 2002, pp. 106-116.

[89] R. L. Antonelli and A. Oliveros, "Traceability en la Etapa de Elicitación de Requerimientos," in *II Workshop de Investigadores en Ciencias de la Computación*, 2000.

[90] O. C. Gotel and A. C. Finkelstein, "An analysis of the requirements traceability problem," in *Requirements Engineering, 1994., Proceedings of the First International Conference on*, 1994, pp. 94-101.

[91] R. Brcina, S. Bode, and M. Riebisch, "Optimisation process for maintaining evolvability during software evolution," in *Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the*, 2009, pp. 196-205.

[92] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "Evaluating The Effect Of Inheritance On The Maintainability Of Object-Oriented Software," in *Empirical Studies of Programmers: Sixth Workshop*, 1996, p. 39.

[93] P. Brereton, B. Kitchenham, D. Budgen, and Z. Li, "Using a protocol template for case study planning," in *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering. University of Bari, Italy*, 2008.

[94] R. K. Yin, *Case Study Research: Design and Methods*, Third Editon ed. vol. 5: SAGE PUBLICATIONS, 2003.