



UNIVERSIDAD VERACRUZANA  
FACULTAD DE ESTADÍSTICA E  
INFORMÁTICA

---

REPORTE DEL TRABAJO:  
“FACILIDAD DE EVOLUCIÓN Y DE MANTENIMIENTO DE  
SOFTWARE”

MODALIDAD:  
MONOGRAFÍA

PRESENTA:  
MARIO EDUARDO DORANTES HERNÁNDEZ

DIRECTOR:  
DRA. MARÍA KAREN CORTÉS VERDÍN

CO DIRECTOR:  
MTRA. MARÍA DE LOS ÁNGELES ARENAS VALDÉS

XALAPA, VER. 2024

## Tabla de contenido

1. Introducción .....	4
2. Método.....	5
2. 1 Planeación .....	5
2.1.1 Preguntas de investigación .....	5
2.1.2 Palabras clave.....	6
2.1.3 Cadena de búsqueda.....	6
2.1.4 Selección de fuentes .....	7
2.1.5 Criterios de inclusión y exclusión .....	8
2.1.6 Proceso de selección de estudios primarios .....	8
2.1.7 Formato de extracción de datos .....	10
2.2 Ejecución .....	12
2.2.1 Resultados de la búsqueda.....	12
3. Resultados .....	14
3.1 Respuestas a las preguntas de investigación .....	19
4. Referencias .....	59

## Figuras

<b>Figura 1.</b> Diagrama de actividad del proceso de búsqueda. ....	9
<b>Figura 2.</b> Distribución estudios primarios por año de publicación. ....	17
<b>Figura 3.</b> Tipo de publicación. ....	18
<b>Figura 4.</b> Distribución de los estudios primarios seleccionados de acuerdo con las bases de datos consultadas. ....	18
<b>Figura 5.</b> Elementos de la Facilidad de Evolución, Brcina, Bode & Riebisch, 2009. ....	23
<b>Figura 6.</b> Elementos de la facilidad de mantenimiento, Cai, Liu, Zhang, Tong & Yang, 2010. ....	28
<b>Figura 7.</b> Elementos de la facilidad de mantenimiento, Hinceeranan & Rivepiboon, 2012. ....	29
<b>Figura 8.</b> Elementos facilidad de mantenimiento, Broy, Deissenboek & Pizka, 2007. ....	31
<b>Figura 9.</b> Elementos que conforman la facilidad de evolución / mantenimiento de un sistema de software ....	32
<b>Figura 10.</b> Modelo de facilidad de mantenimiento con uso de Code Smells, Wagey, Hendradjaya & Mardiyanto, 2015. ....	40

## Tablas

<b>Tabla 1.</b> Palabras clave. ....	6
<b>Tabla 2.</b> Formato de extracción de datos. ....	10
<b>Tabla 3.</b> Resultados del proceso de búsqueda. ....	12
<b>Tabla 4.</b> Estudios primarios seleccionados. ....	14
<b>Tabla 5.</b> Métricas relacionadas con facilidad de evolución o facilidad de mantenimiento. ....	49
<b>Tabla 6.</b> Elementos importantes para diseñar software fácil de evolucionar o mantener ....	55
<b>Tabla 7.</b> Patrones de diseño relacionados con la facilidad de evolución o mantenimiento. ....	56

## 1. Introducción

En este documento, se presenta el reporte de la revisión sistemática de la literatura, que trata el tema de Facilidad de evolución o mantenimiento del software. Se pretende investigar y analizar las diferentes propuestas para definir la facilidad de evolución o mantenimiento, así como saber los elementos que contribuyen al software fácil de evolucionar o mantener. Al igual que conocer enfoques existentes para el diseño y medición de software fácil de evolucionar o mantener.

Hoy en día vivimos en un mundo de constante cambio, donde todo el tiempo se deben buscar alternativas para mejorar productos, servicios y procesos, entre otras cosas. En el contexto del software es aún más notoria esta necesidad de mejora, debido a los requisitos cambiantes por parte del mercado y del entorno. Dada esta situación, los sistemas de software deben modificarse constantemente, ya sea para añadir nuevas funcionalidades o para cambios en la tecnología.

Por tal motivo, se busca que un sistema de software tenga la capacidad de evolucionar o mantener, ya que, gracias a esta característica, el sistema podrá adaptarse a cambios en el futuro sin sufrir demasiadas repercusiones. Es ideal considerar esta característica desde el inicio del desarrollo de un producto de software, para evitar estas repercusiones.

Algunas de estas repercusiones, es el retrabajo por parte de los ingenieros de software al tener que lidiar con un sistema que no contempló la evolución o mantenimiento, así como el costo de dicho retrabajo, ya que simplemente no se puede desechar un sistema de software por todo lo que conllevó realizarlo. Es importante mencionar, que la mayor parte de los costos de un producto de software, se relacionan con la evolución o mantenimiento de este para poder seguir sirviendo a los usuarios de manera satisfactoria.

Este trabajo surge por la necesidad de conocer el estado actual de la facilidad de evolución o mantenimiento. Es decir, saber cómo es que los distintos autores definen dicha característica, y cuáles son los elementos que contribuyen a esta. Asimismo, saber si existe alguna manera de diseñar software fácil de evolucionar o mantener, y cómo puede ser medido. Por último, identificar si existen experiencias o resultados al haber considerado la facilidad de evolución o mantenimiento al desarrollar un producto de software.

Este reporte incluye dos elementos importantes, el primero de ellos es el apartado del método, donde se incluyen elementos como lo son las preguntas de investigación, la cadena de búsqueda, proceso de selección de estudios, resultados de la búsqueda, etc. Por otro lado, el segundo elemento importante de este trabajo es el apartado de los resultados, donde se da respuesta a las preguntas de investigación planteadas.

## 2. Método

En el presente apartado, se muestra el método seguido para poder llevar a cabo la investigación. El método está basado en la propuesta del libro Evidence-Based Software Engineering and Systematic Reviews (Kitchenham, Budgen & Brereton. 2015) y se explica a continuación.

### 2.1 Planeación

En esta sección, se presentan los elementos que involucra la planeación, tal es el caso de las preguntas de investigación, la estrategia de búsqueda y la selección de estudios primarios, cada una de estas en su fase de planeación.

#### 2.1.1 Preguntas de investigación

Con el fin de lograr el objetivo de la RSL, se determinaron las siguientes preguntas de investigación:

**RQ1-** ¿Cuáles son las propuestas para definir la facilidad de evolución o facilidad de mantenimiento?

**Motivación:** Conocer los enfoques, propuestas que a lo largo del tiempo han existido para definir la facilidad de evolución o facilidad de mantenimiento en el contexto del software.

**RQ2 -** ¿Cuáles son los elementos o aspectos que contribuyen a la facilidad de evolución o facilidad de mantenimiento?

**Motivación:** Saber que elementos intervienen o definen la facilidad de evolución o facilidad de mantenimiento.

**RQ3 -** ¿Cómo se mide software fácil de evolucionar o fácil de mantener?

**Motivación:** Saber si existen métricas para medir la facilidad de evolución o facilidad de mantenimiento en el software.

**RQ4-** ¿Cómo se diseña software fácil de evolucionar o fácil de mantener?

**Motivación:** Saber si existen estrategias y/o patrones de diseño que consideren la facilidad de evolución o facilidad de mantenimiento.

**RQ5** – ¿Cuáles han sido las experiencias o resultados obtenidos al considerar la facilidad de evolución o facilidad de mantenimiento en el desarrollo de software?

**Motivación:** Saber si ha habido experiencias utilizando algún método o estrategia para la facilidad de evolución o facilidad de mantenimiento en el desarrollo de software.

### 2.1.2 Palabras clave

Aquí se muestran los términos (palabras clave) relacionados con el tema de este trabajo, los cuales fueron obtenidos tomando como base las preguntas de investigación. Tales términos son un predecesor de la cadena de búsqueda.

*Tabla 1. Palabras clave.*

Palabra clave	Término en inglés	Sinónimo o palabra relacionada	Término(s) de búsqueda
Facilidad de evolución	Evolvability		Evolvability
Facilidad de mantenimiento	Maintainability	Software maintainability	Software maintainability
Medición	Measurement	Metric	Measurement Measure Metric
Diseño de software	Software design	Software Design	Software design
Patrón	Pattern	Method Process Strategy Framework	Pattern Design pattern Architectural pattern Method Process Strategy Framework

### 2.1.3 Cadena de búsqueda

A continuación, se presentan las cadenas de búsqueda que fueron definidas tomando en cuenta los términos de búsqueda obtenidos anteriormente. Primeramente, se presentará una “cadena principal”, la cual se fue refinando hasta

su versión final y fue utilizada en dos bases de datos (IEEE Explore y Springer Link). Posteriormente, se presentarán otras dos cadenas, las cuales tienen gran semejanza a la cadena principal, pero por cuestiones de cada base de datos, como lo es que debían utilizar menos conectores booleanos, tuvieron que ser modificadas. Estas cadenas se emplearon como se muestra en el apartado 2.1.6 Proceso de selección de estudios primarios.

En el caso específico de Springer Link, además de realizar la búsqueda con la “cadena principal”, se seleccionaron algunos filtros para mejorar los resultados, dichos filtros fueron: Computer Science, Software Engineering, Conference Paper y Article. Por otro lado, para las base de datos de ACM, de igual manera para mejorar los resultados obtenidos, se utilizó un filtro para el tipo de contenido, en este caso se usó el de Research Article. Finalmente, referente a la base de datos de Elsevier, se utilizaron los filtros de: Research Articles, Computer Science e Engineering. Las versiones de todas las cadenas comentadas se muestran a continuación:

#### **“Cadena principal” – Utilizada en IEEE Xplore y Springer Link:**

(evolvability OR "software maintainability") AND (measurement OR measure OR metric OR "software design" OR pattern OR "design pattern" OR "architectural pattern" OR method OR process OR strategy OR framework) AND NOT (neural OR robotic\* OR geneti\* OR mechanical OR artificial OR evolutionary)

#### **Cadena utilizada en ACM Digital Library:**

("software evolvability" OR "software maintainability") AND (measurement OR measure OR metric OR "software design" OR pattern OR "design pattern" OR "architectural pattern" OR method OR process OR strategy OR framework) AND NOT (neural OR robotic\* OR geneti\* OR mechanical OR artificial OR evolutionary)

#### **Cadena utilizada en Science Direct (Elsevier):**

("Software Evolvability" OR "Software Maintainability") AND (metric OR "design pattern") AND NOT (mechanical OR artificial OR evolutionary)

#### **2.1.4 Selección de fuentes**

Para la selección de fuentes, se consideraron solo aquellas en las cuales se concentra la mayoría de la literatura relevante para este trabajo.

- IEEE Xplore
- ACM Digital Library
- Springer Link
- Science Direct (Elsevier)

### 2.1.5 Criterios de inclusión y exclusión

A continuación, se presentan los criterios de inclusión y exclusión para este trabajo, tales criterios fueron fundamentales para la selección de los estudios.

#### **Criterios de inclusión**

**CI-1.** Se considerarán solamente estudios primarios en inglés.

**CI-2.** El título debe hacer referencia a desarrollo de software.

**CI-3.** El título y/o abstract debe contener el término de búsqueda “Evolvability”, “Evolution” o “Maintainability”.

**CI-4.** Después de leer el abstract, se muestran indicios que el estudio contesta al menos una pregunta de investigación.

#### **Criterios de exclusión**

**CE-1.** Estudios primarios diferentes a un congreso o journal.

**CE-2.** No se considerarán estudios relacionados con el área de la biología.

**CE-3.** No es posible obtener acceso al material completo.

**CE-4.** Se trata de un estudio duplicado.

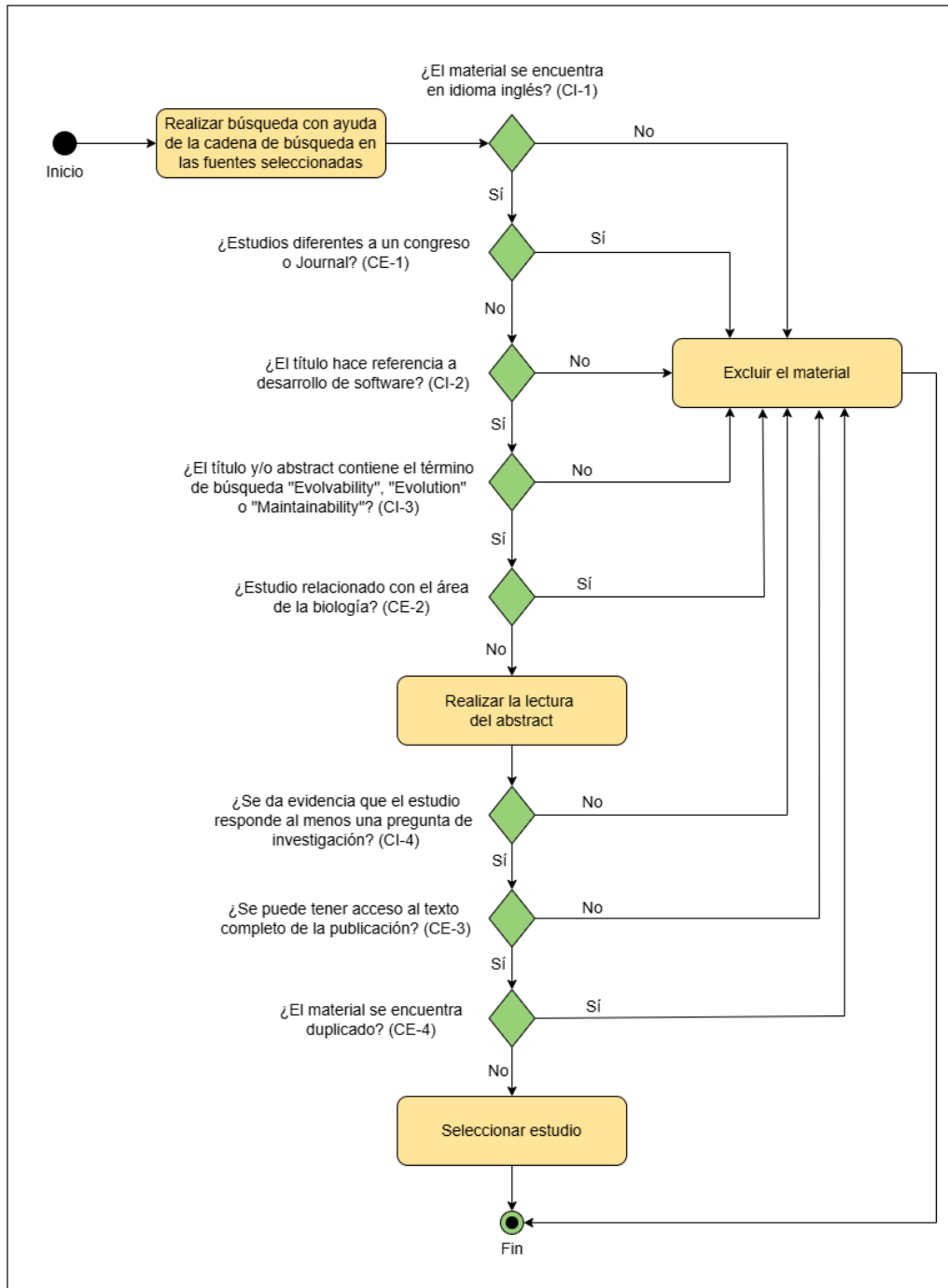
### 2.1.6 Proceso de selección de estudios primarios

Para el procedimiento de selección de estudios primarios, se tomaron en cuenta los siguientes pasos, mismos que se muestran en la Figura 1:

1. Realizar la búsqueda con ayuda de la cadena de búsqueda en las fuentes seleccionadas.
2. Considerar solamente estudios primarios en inglés. (CI-1)
3. Excluir estudios que sean diferentes a un congreso o journal. (CE-1)
4. Verificar que el título hace referencia a desarrollo de software. (CI-2)
5. Verifica que el título y/o abstract contiene el término de búsqueda “Evolvability”, “Evolution” o “Maintainability” (CI-3)
6. Excluir estudios relacionados con el área de la biología. (CE-2)
7. Realizar la lectura del abstract y verificar que se dé evidencia que el estudio responde al menos una pregunta de investigación. (CI-4)
8. Revisar que se pueda tener acceso al texto completo de la publicación (CE-3)
9. Revisar que el material no esté duplicado. (CE-4)



**Figura 1.** Diagrama de actividad del proceso de búsqueda.



### 2.1.7 Formato de extracción de datos

En la presente sección, se muestra el formato que se realizó para la extracción de datos (Tabla 2) en el cual se incluye la información necesaria para este trabajo. Esta tabla se trasladó a una hoja de Excel para recabar la información pertinente, así como para facilitar la administración de la información. De igual manera, dicha extracción se apoyó de la herramienta QualCoder, para una mejor gestión de dicho proceso. A continuación, se muestra el formato de extracción de datos:

**Tabla 2.** Formato de extracción de datos.

Datos del estudio					
Título	Título del artículo				
Autores	Los nombres de los autores				
Año	Año del artículo				
Fuente	Nombre de la base de datos				
Tipo de publicación	Clasificación del artículo				
Referencia o DOI	DOI del artículo o número de identificador del artículo				
Palabras clave	Términos clave del artículo				
Abstract o resumen	Abstract o resumen del artículo				
Pregunta(s) de investigación relacionada(s)	RQ1	RQ2	RQ3	RQ4	RQ5
	Se marca la casilla en caso de relacionars e con la pregunta	Se marca la casilla en caso de relacionars e con la pregunta	Se marca la casilla en caso de relacionars e con la pregunta	Se marca la casilla en caso de relacionars e con la pregunta	Se marca la casilla en caso de relacionars e con la pregunta
Extracción para la síntesis					
RQ1- ¿Cuáles son las propuestas para definir la facilidad de evolución o facilidad de mantenimiento ?	Propuestas	Información extraída relacionada con las propuestas para definir la facilidad de evolución o facilidad de mantenimiento			
	Enfoques	Información extraída relacionada con los enfoques para definir la facilidad de evolución o facilidad de mantenimiento			

RQ2- ¿Cuáles son los elementos o aspectos que contribuyen a la facilidad de evolución o facilidad de mantenimiento ?	Elementos	Información extraída relacionada con los elementos que intervienen en la facilidad de evolución o facilidad de mantenimiento
	Aspectos	Información extraída relacionada con los aspectos que intervienen en la facilidad de evolución o facilidad de mantenimiento
RQ3- ¿Cómo se mide software fácil de evolucionar o fácil de mantener?	Métricas	Información extraída relacionada con las métricas para medir la facilidad de evolución o facilidad de mantenimiento
RQ4- ¿Cómo se diseña software fácil de evolucionar o fácil de mantener?	Estrategias	Información extraída relacionada con las estrategias de diseño que consideren la facilidad de evolución o facilidad de mantenimiento
	Patrones	Información extraída relacionada con los patrones de diseño que consideren la facilidad de evolución o facilidad de mantenimiento
	Métodos	Información extraída relacionada con los métodos de diseño que consideren la facilidad de evolución o facilidad de mantenimiento
	Procesos	Información extraída relacionada con los procesos de diseño que consideren la facilidad de evolución o facilidad de mantenimiento
	Frameworks	Información extraída relacionada con los Frameworks de diseño que consideren la facilidad de evolución o facilidad de mantenimiento
RQ5- ¿Cuáles han sido las experiencias o resultados obtenidos al considerar la facilidad de evolución o facilidad de mantenimiento en el desarrollo de software?	Experiencias	Información extraída relacionada con las experiencias obtenidas al considerar la facilidad de evolución o facilidad de mantenimiento
	Resultados	Información extraída relacionada con los resultados obtenidos al considerar la facilidad de evolución o facilidad de mantenimiento
Notas	Notas del estudio	
Análisis de la información	Análisis de la información del estudio	

## 2.2 Ejecución

En esta sección se presenta lo sucedido en el proceso de búsqueda de los estudios primarios para este trabajo. Los resultados aquí presentados, abarcan las cuatro bases de datos consideradas, así como los snowballings correspondientes según sea el caso.

### 2.2.1 Resultados de la búsqueda

Para la realización de la RSL, se realizó el procedimiento de selección de estudios primarios presentado con anterioridad. En esta sección, se muestran los resultados obtenidos en las bases de datos contempladas, donde se describen los pasos del proceso de búsqueda que se aplicó. A continuación, se presentan dichos resultados en la Tabla 3:

*Tabla 3. Resultados del proceso de búsqueda.*

Fuente	Paso 1	Paso 2 (CI-1)	Paso 3 (CE-1)	Paso 4 (CI-2)	Paso 5 (CI-3)	Paso 6 (CE-2)	Paso 7 (CI-4)	Paso 8 (CE-3)	Paso 9 (CE-4)	Estudios seleccionados
IEEE Xplore	497	496	479	401	372	369	19	19	19	19
IEEE – Backward Snowballing	400	400	189	173	72	71	19	17	8	8
IEEE – Forward Snowballing	133	133	116	108	45	44	8	6	1	1
Springer Link	220	220	220	187	51	51	4	3	3	3
Springer – Backward Snowballing	73	71	43	41	10	10	6	6	1	1
Springer – Forward Snowballing	14	14	9	9	2	2	0	0	0	0
ACM	59	59	56	49	18	18	5	5	3	3
ACM – Backward Snowballing	64	64	37	37	21	21	4	4	1	1

ACM – Forward Snowballing	36	36	31	29	14	14	2	2	1	1
Science Direct - Elsevier	16	16	14	14	11	10	0	0	0	0
Total estudios seleccionados:										37

Para la base de datos de IEEE Xplore, como primer paso, se ingresó la cadena dando como resultado 497 estudios encontrados (no fue necesario aplicar algún filtro para dicha búsqueda). Luego para el paso dos, se mantenían 496 estudios al haber aplicado el CI-1 (Considerar solo estudios en inglés). Para el paso tres, se aplicó el CE-1 (excluir estudios diferentes a una conferencia o journal) reduciendo los resultados a 479. En el paso cuatro, se aplicó el CI-2 (el título hace referencia a desarrollo de software) quedando 401 estudios a considerar.

Posteriormente, en el paso cinco, se redujeron los resultados a 372 al aplicar el CI-3 (El título y/o abstract contiene el término de búsqueda “Evolvability”, “Evolution” o “Maintainability”). Llegado el paso seis del proceso, se aplicó el CE-2 (excluir estudios relacionados con el área de la biología) quedando 369 estudios. Para el paso siete, se redujeron los resultados a 19 al aplicar el CI-4 (Realizar la lectura del abstract y verificar que se da evidencia que se responde al menos una pregunta de investigación). Finalmente, en el paso ocho y nueve, al aplicar el CE-3 (acceso al texto completo de la publicación) y el CE-4 (revisar estudios duplicados) respectivamente, se mantuvieron esos 19 estudios, los cuales fueron los que se seleccionaron.

Dicho proceso de selección de los estudios primarios fue el mismo para las otras bases de datos y sus respectivos snowballings. La diferencia radica en el número de estudios obtenidos al ingresar la cadena de búsqueda a las bases de datos y finalmente los estudios seleccionados luego de aplicar el proceso de selección, ya que, a diferencia de IEEE, en las demás bases de datos existían menos estudios relevantes para este trabajo.

Para el backward snowballing de IEEE, se obtuvieron 400 estudios, de los cuales finalmente se seleccionaron 8. Referente al forward snowballing de dicha base, se obtuvieron 133 estudios, de los cuales solamente 1 se seleccionó. Por otro lado, para la base de datos de Springer Link, al ingresar la cadena de búsqueda se obtuvieron 220 estudios, de los cuales únicamente 3 fueron seleccionados. De tal modo que para el backward snowballing de Springer Link se obtuvieron 73 estudios,

de los cuales solo 1 se seleccionó, y referente al forward snowballing de esta base se obtuvieron 14 estudios, de los cuales ninguno fue seleccionado.

Siguiendo con el proceso de búsqueda y selección de los estudios, para la base de datos ACM, al ingresar la cadena se obtuvieron 59 estudios, de los cuales únicamente 3 fueron los seleccionados. Para el backward snowballing de dicha base se obtuvieron 64 estudios de los cuales 1 fue seleccionado, y para el forward snowballing, se obtuvieron 36 estudios, de los cuales también solo 1 fue seleccionado. Finalmente, para la base de datos Elsevier, luego de aplicar la cadena de búsqueda se obtuvieron únicamente 16 estudios, de los cuales no se seleccionó ninguno, por tal motivo, tampoco fue posible realizar algún snowballing para ampliar los resultados.

### 3. Resultados

A continuación, se muestran los hallazgos de esta RSL, dichos resultados abarcan las tres bases de datos de las cuales se seleccionaron estudios (IEEE, Springer, ACM). Además, también se muestran los hallazgos de los estudios obtenidos por snowballing.

Primeramente, como resultado de la selección de estudios primarios, se incluyeron 37 estudios en total (Tabla 3). Dicho esto, en la Tabla 4 se listan estos estudios junto con alguna de su información importante.

*Tabla 4. Estudios primarios seleccionados.*

#	ID	Nombre	Autor(es)	Año	Tipo de publicación
1	EP-01-IEEE	Optimisation Process for Maintaining Evolvability during Software Evolution	Robert Brcina, Stephan Bode, Matthias Riebisch	2009	Congreso
2	EP-02-IEEE	Analysis of Software Evolvability in Quality Models	H. P. Breivold; I. Crnkovic	2009	Congreso
3	EP-03-IEEE	A framework for assessing the evolvability characteristics along with sub-characteristics in AOSQ model using fuzzy logic tool	P. Kumar; S. K. Singh	2017	Congreso
4	EP-04-IEEE	Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study	H. P. Breivold; I. Crnkovic; R. Land; M. Larsson	2008	Congreso
5	EP-05-IEEE	Analyzing Software Evolvability	H. P. Breivold; I. Crnkovic; P. J. Eriksson	2008	Congreso

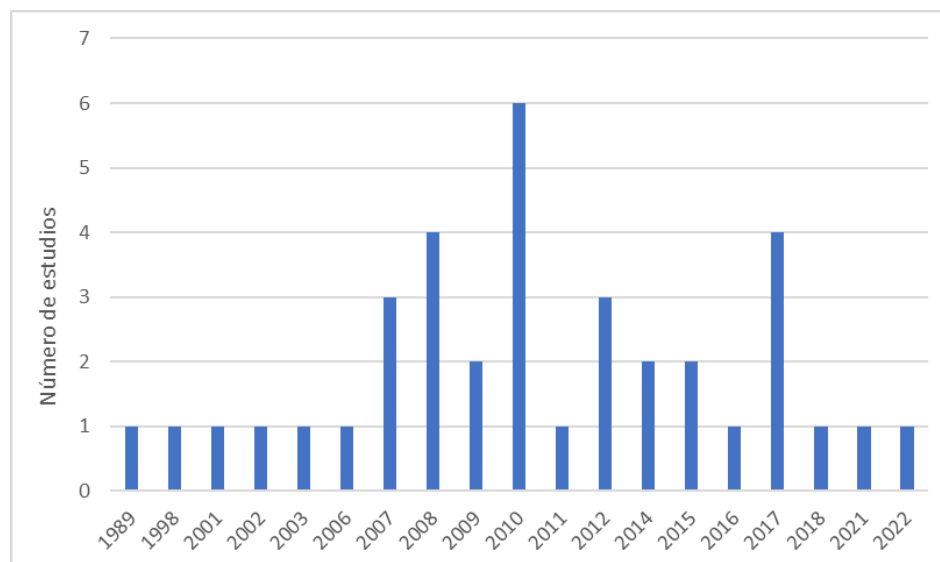
6	EP-06-IEEE	Optimum technology insertion into systems based on the assessment of viability	P. A. Sandborn; T. E. Herald; J. Houston; P. Singh	2003	Journal
7	EP-07-IEEE	Using dependency model to support software architecture evolution	H. P. Breivold; I. Crnkovic; R. Land; S. Larsson	2008	Congreso
8	EP-08-IEEE	Towards a practical maintainability quality model for service-and-microservice based systems	Justus Bogner, Stefan Wagner, Alfred Zimmermann	2017	Congreso
9	EP-09-IEEE	Evaluating software evolvability	H. P. Breivold, I. Crnkovic, and P. Eriksson	2007	Congreso
10	EP-10-IEEE	Dynamic and static views of software evolution	S. Cook, H. Ji, and R. Harrison	2001	Congreso
11	EP-11-IEEE	Using Software Evolvability Model for Evolvability Analysis	Breivold, H.P., and Crnkovic, I	2008	Congreso
12	EP-12-IEEE	Evolvability as a quality attribute of software architectures	S. Ciraci and P. van den Broek	2006	Congreso
13	EP-13-IEEE	On Automatically Collectable Metrics for Software Maintainability Evaluation	J. -P. Ostberg; S. Wagner	2014	Congreso
14	EP-14-IEEE	Analyzing the Effect of Design Patterns on Software Maintainability: A Case Study	S. Rochimah; P. G. Nuswantara; R. J. Akbar	2018	Congreso
15	EP-15-IEEE	A Quantitative Approach to Software Maintainability Prediction	L. Ping	2010	Congreso
16	EP-16-IEEE	Evaluating the Impact of Design Patterns on Software Maintainability: An Empirical Evaluation	H. K. Jun; M. E. Rana	2021	Congreso
17	EP-17-IEEE	An integrated measure of software maintainability	K. K. Aggarwal; Y. Singh; J. K. Chhabra	2002	Congreso
18	EP-18-IEEE	Software maintainability-a new 'ility'	D. A. Sunday	1989	Congreso
19	EP-19-IEEE	A proposal of software maintainability model using code smell measurement	B. C. Wagey; B. Hendradjaya; M. S. Mardiyanto	2015	Congreso
20	EP-20-IEEE	Why Is It Important to Measure Maintainability and What Are the Best Ways to Do It?	C. Chen; R. Alfayez; K. Srisopha; B. Boehm; L. Shi	2017	Congreso
21	EP-21-IEEE	A Controlled Experiment for Evaluating the Impact	M. Perepletchikov; C. Ryan	2011	Journal

		of Coupling on the Maintainability of Service-Oriented Software			
22	EP-22-IEEE	Evaluating maintainability of android applications	A. A. Saifan; A. Al-Rabadi	2017	Congreso
23	EP-23-IEEE	Evaluating Software Maintainability Using Fuzzy Entropy Theory	L. Cai; Z. Liu; J. Zhang; W. Tong; G. Yang	2010	Congreso
24	EP-24-IEEE	Research on maintainability evaluation of service-oriented software	Ma Zhe; Ben Kerong	2010	Congreso
25	EP-25-IEEE	A Practical Model for Measuring Maintainability	I. Heitlager, T. Kuipers, and J. Visser	2007	Congreso
26	EP-26-IEEE	A maintainability estimation model and tool	A. Hincheeranan and W. Rivepiboon	2012	Journal
27	EP-27-IEEE	A comparative Study of Maintainability Index of Open Source Software	Ganpati, Anita, A. Kalia, and H. Singh.	2012	Journal
28	EP-28-IEEE	Measuring Maintainability Index of a Software Depending on Line of Code Only	Najm, N.	2014	Journal
29	EP-01-SPRINGER	Evolvability Characterization in the Context of SOA	Jose L. Arciniegas H. & Juan C. Dueñas L.	2010	Congreso
30	EP-02-SPRINGER	Impact Evaluation for Quality-Oriented Architectural Decisions regarding Evolvability	Stephan Bode & Matthias Riebisch	2010	Congreso
31	EP-03-SPRINGER	Defining systems architecture evolvability - a taxonomy of change	Rowe, D., Leaney, J., Lowe, D.	1998	Congreso
32	EP-04-SPRINGER	A Bayesian Belief Network for Modeling Open Source Software Maintenance Productivity	Stamatia Bibi, Apostolos Ampatzoglou, Ioannis Stamelos	2016	Congreso
33	EP-01-ACM	A Generic Method for Identifying Maintainability Requirements Using ISO Standards	Khalid T. Al-Sarayreh, Asma Labadi & Kenza Meridji	2015	Congreso
34	EP-02-ACM	Questioning software maintenance metrics: a comparative case study	Dag I.K. Sjoberg, Bente Anda & Audris Mockus	2012	Congreso
35	EP-03-ACM	Design property metrics to maintainability estimation: a virtual method using functional relationship mapping	T.R. Gopalakrishnan Nair, Sri Aravindh & R.Selvarani	2010	Congreso



36	EP-04-ACM	Demystifying maintainability	M. Broy, F. Deissenboeck, and M. Pizka	2007	Congreso
37	EP-05-ACM	Impact of Abstract Factory and Decorator Design Patterns on Software Maintainability: Empirical Evaluation using CK Metrics	Kurmangali A, Rana M and Ab Rahman W.	2022	Congreso

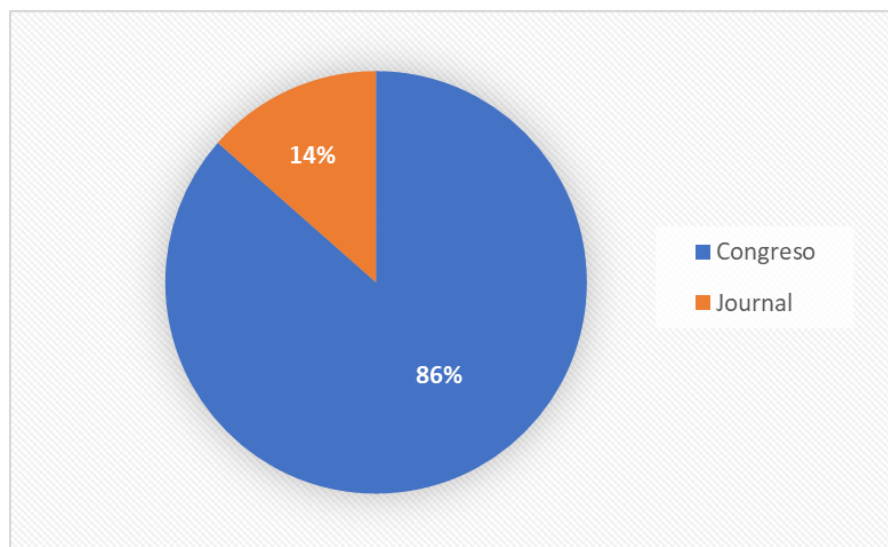
En la Figura 2 se muestra una gráfica con los estudios primarios por año de publicación. Como se puede notar, la mayoría de los estudios que fueron seleccionados para este trabajo, se encuentran publicados en el año 2010, seguidos de los años 2008 y 2017. Por otra parte, es interesante notar que posterior al año 2017 únicamente existe un estudio por año relevante para este trabajo. Esto es llamativo, siendo que la facilidad de evolución o mantenimiento es esencial para un sistema de software, exista la escasez de investigaciones en la actualidad.



*Figura 2. Distribución estudios primarios por año de publicación.*

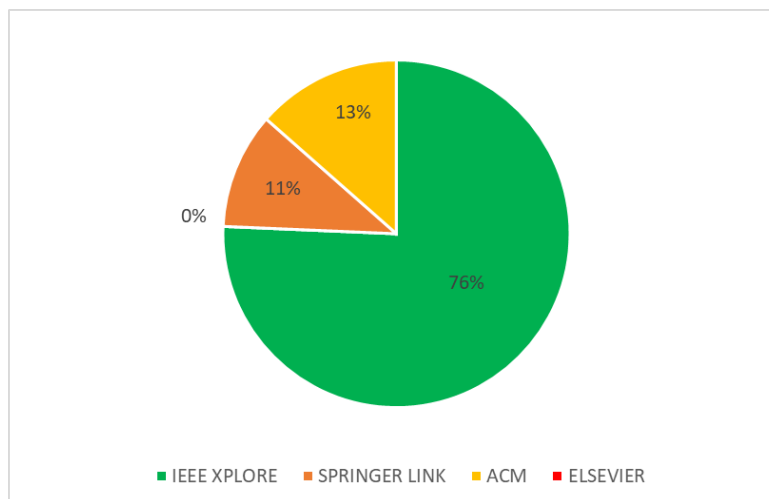
En la Figura 3, se presentan las proporciones de los estudios según su tipo de publicación. El 86% de los estudios seleccionados fueron publicados en congresos. Es interesante observar, como solo un 14% de los estudios seleccionados, es de tipo Journal, esto podría ser debido a que este tipo de trabajos tienen una revisión más estricta. Por tal motivo, se podría decir que la investigación en el tema tiene poco desarrollo. Esto reafirma lo comentado anteriormente en la

Figura 2, donde hasta en los años que se distribuyeron más estudios del tema, sigue siendo una cantidad relativamente pequeña de estudios primarios.



*Figura 3. Tipo de publicación.*

En la Figura 4, se muestra la distribución de los estudios primarios seleccionados de acuerdo con las bases de datos consultadas, mostrando que en IEEE Xplore es de donde se seleccionaron la mayoría de los estudios, teniendo el 76% (28 estudios) del total. Por otro lado, únicamente un 13% (5 estudios) de los estudios corresponden a ACM y el 11% (4 estudios) restante son los estudios obtenidos de Springer Link. Cabe resaltar que la base de datos Elsevier, se representa con el 0%, (0 estudios) ya que, en el proceso de selección de dicha base, ningún estudio cumplió con los criterios para ser incluido en el trabajo.



*Figura 4. Distribución de los estudios primarios seleccionados de acuerdo con las bases de datos consultadas.*

### 3.1 Respuestas a las preguntas de investigación

A continuación, se presentan las respuestas a las preguntas de investigación que fueron definidas para este trabajo.

#### **RQ1- ¿Cuáles son las propuestas para definir la facilidad de evolución o facilidad de mantenimiento?**

Esta pregunta de investigación fue respondida por 15 de los 37 estudios que fueron seleccionados. A continuación, se presentan los hallazgos:

Breivold, Crnkovic y Eriksson (2008) en el EP-05-IEEE definen la facilidad de evolución como “Un atributo que se relaciona con la capacidad del sistema para adaptarse a los cambios en sus requisitos a lo largo de la vida útil del sistema con el menor costo posible y manteniendo la integridad arquitectónica”. En una perspectiva similar Sandborn, Herald, Houston y Singh (2003) en el EP-06-IEEE definen el término facilidad de evolución como “Capacidad del sistema para admitir los requisitos de capacidad proyectados con el diseño y la arquitectura”.

Cook, Ji y Harrison (2001) en su estudio primario (EP-10-IEEE), definen la facilidad de evolución como “La capacidad de un producto de software de evolucionar para continuar sirviendo a su cliente de una manera rentable”. Por su parte, en el EP-12-IEEE, Ciraci y Broek (2006), definen la facilidad de evolución como “La capacidad de un sistema para sobrevivir a los cambios en su entorno, requisitos y tecnologías de implementación”.

En contraste, Jun y Rana (2021) en el EP-16-IEEE definen la facilidad de mantenimiento como el “El grado de eficacia y eficiencia en el que un sistema de software puede modificarse después de la implementación para su corrección, mejora, extensión o adaptación”.

Respecto al EP-18-IEEE, Sunday (1989) proporciona una definición de facilidad de mantenimiento del software, en esta, comenta que es una característica del software que refleja el grado de esfuerzo requerido para realizar las tareas que se listan en breve. En el estudio se añade que, si bien no es una definición de “diccionario”, esta proporciona una base para la discusión de las técnicas y el trabajo realizado en el desarrollo del sistema. A continuación, se listan las tareas involucradas:

- Corrección de errores
- Adición de características
- Eliminación de capacidades
- Adaptación / Modificación

Por otro lado, en el estudio primario de Cai, Liu, Zhang, Tong y Yang (2010) (EP-23-IEEE) se especifica que cumplir con los requisitos de calidad del sistema, es algo que se ha vuelto más importante que cumplir con los requisitos de funcionalidad. Se menciona que, para hacer evolucionar un producto, se gastan muchos recursos y tiempo de mantenimiento, de tal modo que, un producto con alta facilidad de mantenimiento puede ahorrar costos. Los autores comparten la definición del Glosario estándar de terminología de ingeniería de software de IEEE, donde la facilidad de mantenimiento se define como: “La facilidad con la que se puede modificar un sistema para corregir fallas, mejorar el rendimiento u otros atributos, o adaptarse a un entorno modificado”.

Zhe y Kerong (2010) en el EP-24-IEEE, definen la facilidad de mantenimiento del software como: “La dificultad para corregir errores y fallas defectuosas, para cumplir con los nuevos requisitos y la facilidad con la que se puede entender el sistema de software existente”. Por otro lado, Hinceeranan y Rivepiboon (2012) en el EP-26-IEEE mencionan que la facilidad de mantenimiento está definida por el glosario estándar de ingeniería de software IEEE como “La facilidad con la que un sistema o componente de software puede modificarse para corregir fallas, mejorar el rendimiento u otros atributos, o adaptarse a un entorno modificado”. Además, los autores añaden que medir la facilidad de mantenimiento del sistema de software en la fase de diseño, puede ayudar a que un diseñador de software deba mejorar dicha característica del sistema antes de entregarlo al cliente.

Referente al EP-27-IEEE, Ganpati, Kalia y Singh (2012) expresan que el propósito del mantenimiento del software es mantener el software operativo, así como prevenir y corregir fallas en el sistema y mejorar la funcionalidad. Los autores comparten la definición proporcionada por el IEEE, donde en este se define el mantenimiento como “El proceso de modificar un sistema o componente de software después de la entrega, para corregir fallas, mejorar el rendimiento u otros atributos, o adaptarse a un entorno modificado”. Además, añaden que la facilidad de mantenimiento está estrechamente relacionada con el mantenimiento del software.

En cuanto al estudio primario de Bogner, Wagner y Zimmermann (2017) (EP-08-IEEE) se destaca que algo de suma importancia para las organizaciones, es modificar un sistema de software de manera rápida y efectiva. Mencionan que el atributo de calidad relevante para esto es la facilidad de mantenimiento, la cual se define como “El grado de eficacia y eficiencia con el que se puede modificar un sistema de software para corregirlo, mejorarlo, ampliarlo o adaptarlo”.

Archiniegas y Dueñas (2010) indican en su estudio (EP-01-SPRINGER) que la facilidad de evolución tiene un efecto directo sobre las características del ciclo de vida de un sistema de software. Los autores describen la facilidad de evolución como: “La capacidad de anticipar las ubicaciones de las transformaciones en un sistema y su capacidad para adaptarse o modificarse para lograr un equilibrio entre costos y recursos, restricciones y efectos locales o globales”.

En el EP-02-SPRINGER, Bode y Riebisch (2010) usan una definición basada en Breivold et al (2007) y Rowe et al (1998). De tal modo que, se refieren a facilidad de evolución como: “La capacidad de un sistema de software a lo largo de su vida útil para adaptarse a los cambios y mejoras en los requisitos y tecnologías que influyen en la estructura arquitectónica del sistema, con el menor costo posible y manteniendo la integridad arquitectónica”.

Respecto al estudio primario de Rowe, Leaney y Lowe (1998) (EP-03-IEEE) hacen referencia a la facilidad de mantenimiento del IEEE 610, donde esta característica es definida como: “La facilidad con la que un sistema o componente de hardware puede conservarse o restaurarse a un estado en el que pueda realizar las funciones requeridas”. Por otro lado, los autores diferencian la facilidad de mantenimiento y la facilidad de evolución, pues resaltan la importancia de distinguir entre mantener el sistema en un estado en el que pueda realizar sus funciones requeridas, y en adaptarse a los cambios en los requisitos no especificados previamente.

Continuando con el punto anterior, los autores al diferenciar la facilidad de mantenimiento y facilidad de evolución, definen esta última como: “La capacidad de un sistema para adaptarse o hacer frente a cambios en los requisitos, en el entorno y las tecnologías de implementación”. Adicionalmente, los autores exploran cualidades de software que contribuyen a la facilidad de evolución de un sistema de software. De tal modo que, reestablecen su definición de facilidad de evolución, donde se refieren a esta como: “Un atributo que se relaciona con la capacidad de un sistema para adaptarse a cambios en sus requisitos a lo largo de la vida útil del sistema con el menor costo posible y manteniendo la integridad arquitectónica”.

Por su parte, Al-Sarayreh, Labadi y Meridji (2015) en el EP-01-ACM, exploran definiciones de facilidad de mantenimiento según estándares. En el estudio se proporcionan tres definiciones de esta característica, donde según el ISO 24765, se dice que es “La facilidad con la que un sistema o componente de software puede modificarse para cambiar o agregar capacidades, corregir fallas o defectos, mejorar el rendimiento u otros atributos, o adaptarse a un entorno modificado”. Mientras que, el IEEE 14764 define la facilidad de mantenimiento como “La capacidad del producto de software para modificarse y la facilidad de mantenimiento es la velocidad y la facilidad con la que un programa puede corregirse o cambiarse”. Por otro lado, el ISO 9126, define esta característica como “Una capacidad del producto de software para ser modificado. Las modificaciones pueden incluir correcciones, mejoras o adaptación del software a cambios en el entorno, y en los requisitos y especificaciones funcionales”. Cabe aclarar, que los autores no propusieron alguna definición para facilidad de mantenimiento, solo se limitaron a tomar en cuenta las definiciones propuestas en los estándares.

Vistas las definiciones presentadas como respuestas en esta RQ1, de facilidad de evolución y facilidad de mantenimiento de un sistema de software,

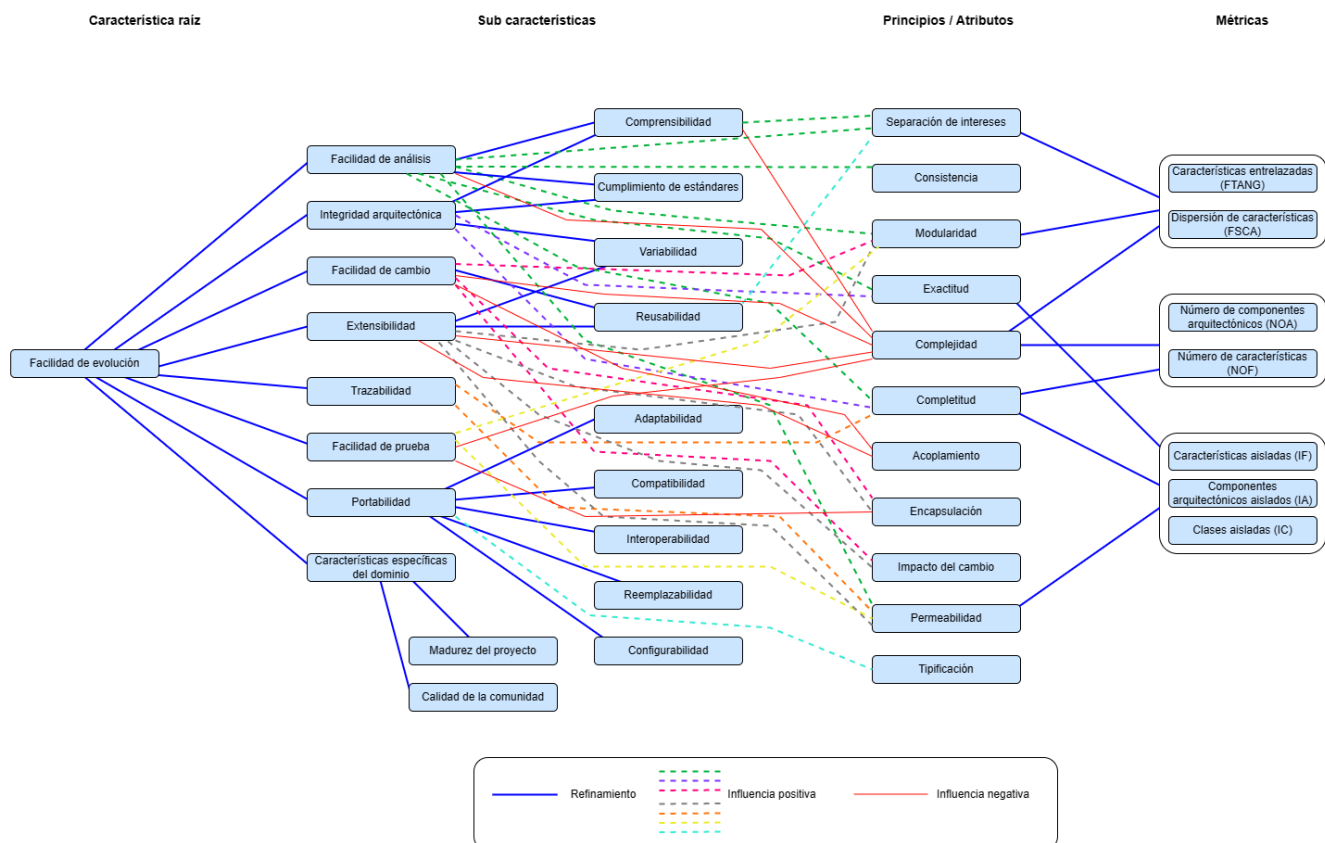
revelan diferentes perspectivas, aunque también muestran algunos puntos de coincidencia importantes. Primeramente, en cuanto a la facilidad de evolución algunos autores como Breivold, Crnkovic y Eriksson (2008), Bode y Riebisch (2010) y Rowe, Leaney y Lowe (1998) destacan su relación con la adaptación a los cambios en los requisitos y la capacidad de mantener la integridad arquitectónica del sistema. Por otro lado, autores como Ciraci & Broek (2006) enfatizan la capacidad del sistema para sobrevivir a cambios en el entorno y tecnologías de implementación. Mientras que autores como Cook, Ji y Harrison (2001) únicamente mencionan la importancia del software de evolucionar para continuar sirviendo al cliente de manera rentable.

En lo que se refiere a las definiciones de facilidad de mantenimiento presentan puntos en común. En general, se entiende como la capacidad de modificar un sistema de software para corregir fallas, mejorar el rendimiento o adaptarse a un entorno cambiado. Autores como Jun y Rana (2021), Bogner, Wagner y Zimmermann (2017), hacen hincapié en la importancia de la eficacia y la eficiencia en la realización de estas modificaciones. Además, se menciona que la facilidad de mantenimiento está estrechamente relacionada con el proceso de mantenimiento del software en sí.

## RQ2- ¿Cuáles son los elementos o aspectos que contribuyen a la facilidad de evolución o facilidad de mantenimiento?

Esta pregunta de investigación fue respondida por 25 de los 37 estudios seleccionados. Los hallazgos se muestran a continuación:

Primeramente, en el estudio primario de Brcina, Bode y Riebisch (2009) (EP-01-IEEE) se presenta un modelo de la facilidad de evolución como una característica raíz, la cual se refina en subcaracterísticas y estas a su vez se refinan con algunos atributos de software. A continuación, en la Figura 5 se muestra cómo se desglosa la característica de facilidad de evolución que presentan los autores en su estudio:



*Figura 5. Elementos de la Facilidad de Evolución, Brcina, Bode & Riebisch, 2009.*

Adaptado de Quality Model Regarding Evolvability, por Brcina, Bode & Riebisch, 2009.

Como se puede apreciar en la Figura 5, las subcaracterísticas que se refinan de la facilidad de evolución, tiene relación con ciertos principios/atributos de software, negativa o positivamente. Por ejemplo, la facilidad de cambio y la extensibilidad tienen una influencia negativa con el acoplamiento, esto debido a que,

si los módulos de un sistema dependen demasiado unos de otros, será difícil realizar cambios. Por otro lado, una influencia positiva se puede localizar en la facilidad de análisis y la consistencia, ya que, la consistencia en un sistema proporciona una base sólida, de tal modo que, será fácil de analizar el sistema en caso de necesitar realizar algún cambio.

Los autores del estudio se enfocan principalmente en las subcaracterísticas que tienen una relación inmediata con la facilidad de evolución, como se puede notar en la Figura 5 presentada anteriormente. Dichas subcaracterísticas son:

- Facilidad de análisis
- Integridad.
- Facilidad de cambio y extensibilidad.
- Trazabilidad.
- Facilidad de prueba.
- Portabilidad
- Características específicas del dominio.

Además, algunas subcaracterísticas se refinan en otras. Sin embargo, los autores solo se limitan a presentarlas, pero sin dar detalle sobre ellas. Tales subcaracterísticas son:

- Comprensibilidad
- Cumplimiento de estándares
- Variabilidad
- Reusabilidad
- Adaptabilidad
- Compatibilidad
- Interoperabilidad
- Reemplazabilidad
- Configurabilidad

Por otra parte, los atributos de software a considerar para la facilidad de evolución que presentan los autores no se describen a detalle. No obstante, como se muestra en la Figura 5, estos tienen influencias positivas o negativas con las subcaracterísticas de la facilidad de evolución. Estos atributos son:

- Separación de intereses
- Consistencia
- Modularidad
- Exactitud
- Complejidad
- Completitud



- Acoplamiento
- Encapsulación
- Impacto del cambio
- Permeabilidad
- Tipificación

Con respecto al trabajo de Breivold, Crnkovic y Eriksson (2008) que corresponde al EP-05-IEEE, los autores destacan que cualquier sistema que no considere explícitamente una o más de las subcaracterísticas de la facilidad de evolución, probablemente no podrá evolucionar adecuadamente. Dichas subcaracterísticas son las siguientes:

- Facilidad de análisis.
- Integridad.
- Facilidad de cambio.
- Portabilidad.
- Extensibilidad.
- Facilidad de prueba.
- Atributos específicos del dominio.

Los autores Breivold y Crnkovic en conjunto con Land y Larsson, en los estudios primarios EP-02-IEEE, EP-04-IEE, EP-07-IEEE y EP-11-IEEE vuelven a mencionar tales subcaracterísticas (presentadas en el EP-05-IEEE) para la facilidad de evolución de los sistemas de software.

Del mismo modo, en el estudio primario de Breivold, Crnkovic y Eriksson (2007) (EP-09-IEEE), se vuelven a mencionar tales subcaracterísticas. No obstante, en este estudio al ser más antiguo, los autores no consideraron la subcaracterística de “atributos específicos del dominio” como sí lo hicieron en sus estudios posteriores a ese año.

Kumar y Singh (2017) en el EP-03-IEEE identifican características importantes para el desarrollo de un sistema de software. Estas características se consideran como variables de entrada, que dirigen a una variable de salida, la cual es facilidad de evolución. A continuación, se listan estas variables:

Entrada:

- Extensibilidad
- Estabilidad de diseño
- Sustentabilidad
- Configurabilidad

Salida:

- Facilidad de evolución

Sin embargo, no se discute ninguna definición para las características que constituyen las variables de entrada. Por otra parte, los autores mencionan que los cuatro elementos de entrada son independientes, mientras que los elementos de salida, es decir, la facilidad de evolución es dependiente. Por lo que la salida depende de elementos independientes.

Por otro lado, Cook, Ji y Harrison (2001) en el EP-10-IEEE señalan que la esencia de la evolución de un producto de software radica en modificar partes de él. De tal modo que, el software debe analizarse, cambiarse, restablecerse y probarse. Los autores relacionan estas actividades con las subcaracterísticas de facilidad de mantenimiento según el ISO 9126:

- Facilidad de análisis.
- Facilidad de cambio.
- Estabilidad.
- Facilidad de prueba.
- Cumplimiento.

El estudio primario de Rochimah, Nuswantara y Akbar (2018) (EP-14-IEEE) presenta subcaracterísticas de facilidad de mantenimiento. Dichas subcaracterísticas se listan a continuación:

- Modularidad
- Reusabilidad
- Facilidad de prueba
- Facilidad de análisis

Ping (2010) en su trabajo de investigación (EP-15-IEEE) señala subcaracterísticas que debe tener un producto de software para considerarse fácil de mantener. Estas subcaracterísticas son:

- Facilidad de cambio
- Facilidad de prueba
- Facilidad de análisis
- Estabilidad

En el EP-16-IEEE, Jun y Rana (2021) indican que la facilidad de mantenimiento generalmente se compone de tres atributos. Tales atributos se listan a continuación:

- Facilidad de análisis.
- Facilidad de cambio.
- Facilidad de prueba.

Por su parte, en el estudio primario de Wagey, Hendradjaya y Mardiyanto (2015) (EP-19-IEEE), se identifican subcaracterísticas de la facilidad de

mantenimiento del software. Es importante mencionar, que solo presentaron dichas subcaracterísticas sin dar mayor detalle. Además, en el estudio tales subcaracterísticas están relacionadas con Code Smells, dicha relación será presentada más adelante en las respuestas a la RQ3. A continuación, se muestran las subcaracterísticas:

- Modularidad
- Reusabilidad
- Facilidad de análisis
- Facilidad de modificación
- Facilidad de prueba

Pereplechikov y Ryan (2011) en su investigación (EP-21-IEEE) señalan que las características de calidad del software se clasifican como internas o externas. Es así, como el diseño de cualquier producto de software posee características internas medibles como podrían ser acoplamiento, cohesión y tamaño, las cuales tienen un efecto causal en las características de calidad externas, como lo es la facilidad de mantenimiento. En este estudio, se presentan subcaracterísticas de facilidad de mantenimiento de acuerdo con el estándar ISO 9126:

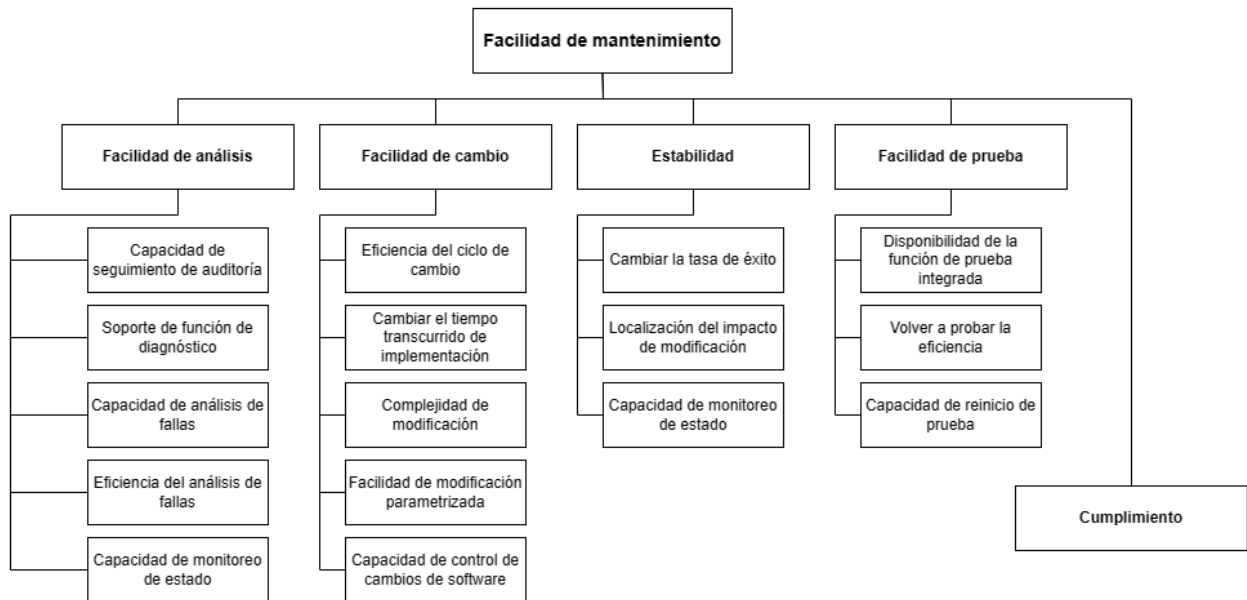
- Facilidad de análisis.
- Facilidad de cambio.
- Estabilidad.
- Facilidad de prueba.

Por otro lado, Saifan y Rabadi (2017) en el EP-22-IEEE exploran el tema de facilidad de mantenimiento del software desde una perspectiva de aplicaciones Android. Se destaca que el desarrollo de este tipo de aplicaciones aumentó exponencialmente, debido al fuerte aumento en los requisitos de los usuarios. Además, estas aplicaciones enfrentan desafíos competitivos y una rápida evolución en la tecnología. De tal modo que, al hacer estos desarrollos tan rápidamente, ha conducido a aplicaciones de baja calidad. En el estudio se mencionan elementos que afectan la facilidad de mantenimiento del software:

- Facilidad de análisis.
- Facilidad de cambio.
- Estabilidad.
- Facilidad de prueba.

En el estudio de Cai, Liu, Zhang, Tong y Yang (2010) (EP-23-IEEE) se indica que la facilidad de mantenimiento es una característica de calidad interna del software, la cual es importante en la evolución del producto de software. La facilidad de mantenimiento tiene cinco subcaracterísticas, las cuales a su vez se dividen en una serie de atributos. Es importante mencionar, que los autores comentan que un atributo es una entidad que se puede verificar o medir en el producto de software,

sin embargo, solo se limitan a mencionarlos como se muestra en la siguiente Figura 6:



*Figura 6. Elementos de la facilidad de mantenimiento, Cai, Liu, Zhang, Tong & Yang, 2010.*

Adaptado de The evaluation index for maintainability, por Cai, Liu, Zhang, Tong & Yang, 2010.

Los autores solo se limitan a presentar estos atributos relacionados con las subcaracterísticas de facilidad de mantenimiento, no obstante, no proporcionan alguna definición o métrica relacionada. Además, como se puede observar, para la subcaracterística de cumplimiento, no se da mayor detalle.

En el estudio primario de Zhe y Kerong (2010) (EP-24-IEEE), se indica que la arquitectura orientada a servicios proporciona una forma de construir un sistema distribuido con la funcionalidad como un servicio para las aplicaciones. En este estudio, se identifican elementos que pueden contribuir a la facilidad de mantenimiento del software orientado a servicios, los cuales se pueden dividir de la siguiente manera:

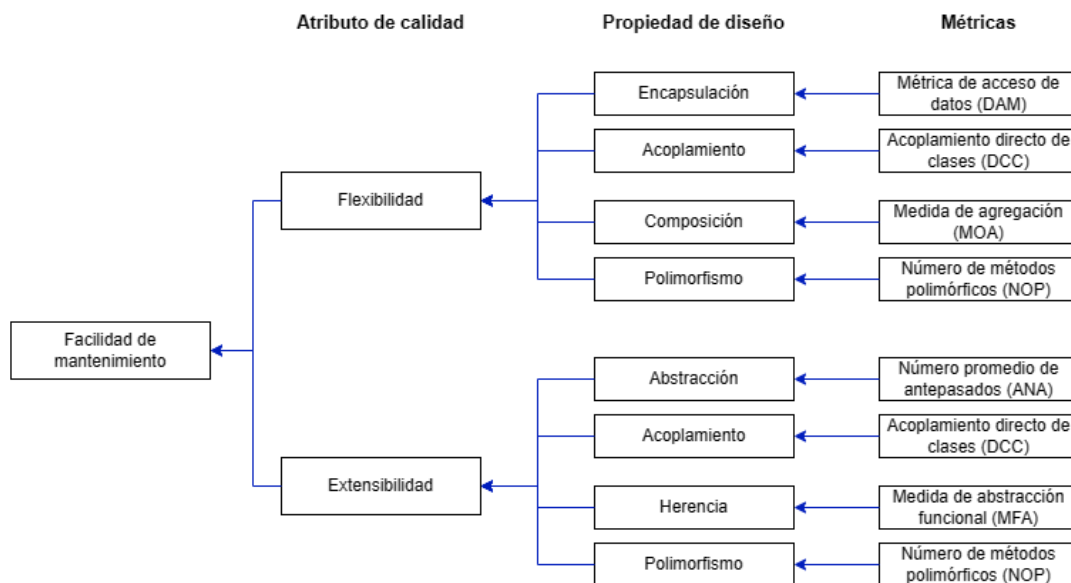
- Facilidad de análisis.
- Facilidad de cambio.
- Estabilidad.
- Facilidad de prueba.

Heitlager, Kuipers y Visser (2007) en el EP-25-IEEE, hacen referencia al modelo de calidad ISO 9126, el cual define 6 características principales de un producto de software, las cuales, a su vez, se subdividen en subcaracterísticas. En

el estudio, se mencionan las subcaracterísticas relacionadas con facilidad de mantenimiento de software:

- Facilidad de análisis.
- Facilidad de cambio.
- Estabilidad.
- Facilidad de prueba.
- Conformidad con la facilidad de mantenimiento.

En estudio de Hinceeranan y Rivepiboon (2012) (EP-26-IEEE) se propone que la flexibilidad y la extensibilidad son subcaracterísticas de la facilidad de mantenimiento del software, las cuales sirven como criterios para evaluar la facilidad de mantenimiento de un diagrama de clases. En el estudio se relata que la flexibilidad es una característica que permite la incorporación de cambios en un diseño, mientras que la extensibilidad se refiere a la presencia y el uso de propiedades de un diseño existente, permitiendo la incorporación de nuevos requisitos en el diseño del software. A continuación, en la Figura 7 se muestran estos atributos de calidad relacionados con la facilidad de mantenimiento, así como propiedades de diseño relacionadas:



*Figura 7. Elementos de la facilidad de mantenimiento, Hinceeranan & Rivepiboon, 2012.*

Adaptado de The structure of the Maintainability Estimation Model, por Hinceeranan & Rivepiboon, 2012.

De acuerdo con el trabajo de investigación de Ganpati, Kalia y Singh (2012) (EP-27-IEEE) se indica que la facilidad de mantenimiento es un atributo de calidad interno del software. En el estudio se señala que dicha característica se puede expresar en función de los siguientes elementos:

- Modularidad
- Portabilidad
- Legibilidad
- Facilidad de prueba
- Reusabilidad
- Flexibilidad
- Conformidad

En el EP-01-SPRINGER, Archiniegas y Dueñas (2010) plantean características que están relacionadas con la facilidad de evolución de un sistema de software. Es importante mencionar, que los autores únicamente se limitaron a nombrarlas, y no proporcionan información adicional. Estas características muestran a continuación:

- Adaptabilidad
- Facilidad de modificación
- Reemplazabilidad
- Extensibilidad
- Trazabilidad
- Variabilidad
- Adaptabilidad

Con respecto al trabajo de investigación de Bode y Riebisch (2010) (EP-02-SPRINGER) se identifican subcaracterísticas de facilidad de evolución, las cuales se basan en la norma ISO 9126. A continuación, se presentan estas subcaracterísticas relacionadas con la facilidad de evolución:

- Facilidad de análisis.
- Facilidad de cambio/Facilidad de modificación.
  - Extensibilidad.
  - Variabilidad.
  - Portabilidad.
- Reusabilidad.
- Facilidad de prueba.
- Trazabilidad.
- Cumplimiento de estándares.
- Cualidades de proceso.

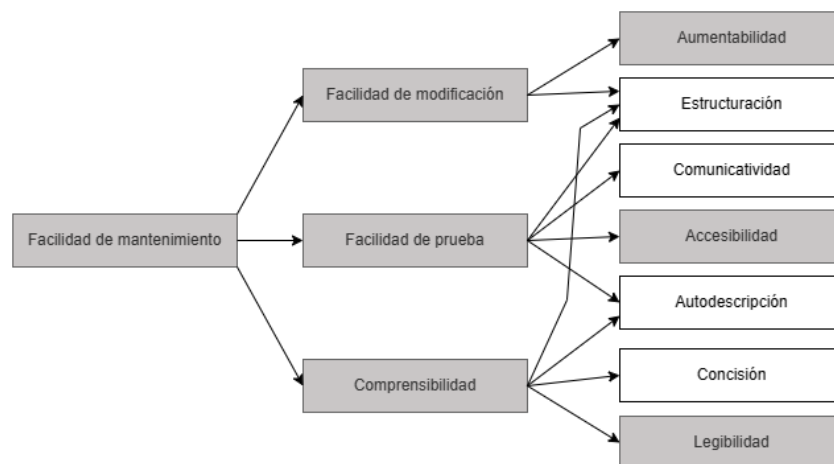
Por otro lado, Rowe, Leaney y Lowe (1998) en su estudio primario (EP-03-SPRINGER) exploran categorías relacionadas con el cambio en un sistema de software, y por lo tanto cualidades arquitectónicas que abordan dichas categorías. Los autores comentan que estas cualidades, contribuyen a la facilidad de evolución del software. A continuación, se presentan dichos elementos:

- Generalidad.
- Adaptabilidad.
- Escalabilidad.
- Extensibilidad.

En el trabajo de investigación de Al-Sarayreh, Labadi y Meridji (2015) (EP-01-ACM) se presentan elementos relacionados con la facilidad de mantenimiento del software:

- Facilidad de análisis.
- Facilidad de cambio.
- Estabilidad.
- Facilidad de prueba.

Por su parte, en el estudio de Broy, Deissenboek y Pizka (2007) (EP-04-ACM) se muestra un árbol de características de calidad de software propuesto por Boehm et. al 1978. En este se puede notar la característica de facilidad de mantenimiento, de la cual se desglosan tres elementos, facilidad de modificación, facilidad de prueba y comprensibilidad. A continuación, en la Figura 8, se muestra el árbol de características:

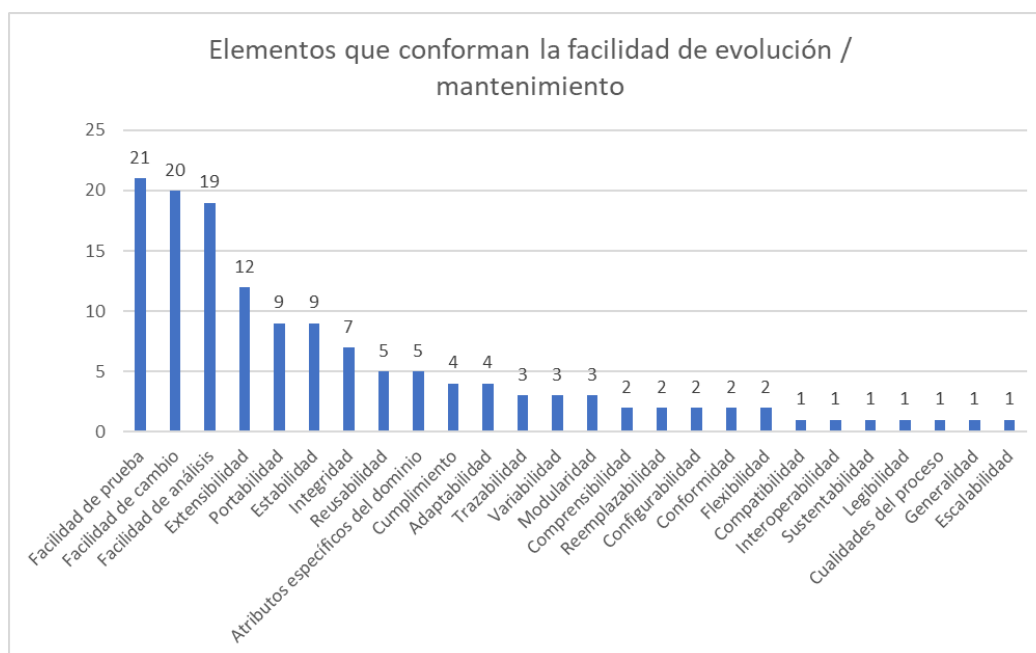


*Figura 8. Elementos facilidad de mantenimiento, Broy, Deissenboek & Pizka, 2007.*

Adaptado de Software Quality Characteristics Tree, por Broy, Deissenboek & Pizka, 2007.

En el estudio se comenta que los recuadros grises, se refieren a actividades, mientras que los recuadros sin color describen características del sistema de software. Donde se podría decir que, para poder mantener fácilmente un sistema de software, necesitamos modificarlo, esta actividad de modificación de alguna manera está influenciada por la estructura del sistema.

Tras revisar las respuestas en esta RQ2, se encuentran similitudes en los elementos propuestos en cada estudio. En la Figura 9, se pueden observar aquellos elementos que fueron más nombrados en los estudios seleccionados. La Figura 9 se muestra a continuación:



**Figura 9.** Elementos que conforman la facilidad de evolución / mantenimiento de un sistema de software

Como se puede observar, la subcaracterística de facilidad de prueba es la más mencionada en los estudios que dieron respuesta a esta RQ2. Donde en 21 de los 25 estudios que respondieron esta RQ, se indica que esta subcaracterística contribuye a la facilidad de evolución o mantenimiento de un sistema de software. Además, las subcaracterísticas de facilidad de cambio (o facilidad de modificación según algunos estudios) y facilidad de análisis de igual manera son de las subcaracterísticas más mencionadas en los estudios, donde en 20 y 19 estudios respectivamente se mencionan dichas subcaracterísticas.

Así mismo, existen otras subcaracterísticas que son mencionadas en al menos 3 estudios de los 25 que responden esta RQ, como lo son la trazabilidad, variabilidad y la modularidad. No obstante, también se mencionan en los estudios



algunas subcaracterísticas que únicamente se presentan en 2 o inclusive 1 estudio, como lo son la comprensibilidad, reemplazabilidad, configurabilidad, conformidad, flexibilidad, compatibilidad, entre otras.

De tal modo que, se consideran de mayor relevancia aquellas subcaracterísticas que tienen mayor coincidencia entre los estudios primarios que responden esta RQ. Es decir, facilidad de prueba, facilidad de cambio, facilidad de análisis, extensibilidad, portabilidad, estabilidad, e inclusive la integridad, son las subcaracterísticas cruciales para tener en cuenta para la facilidad de evolución o mantenimiento de un sistema de software. Tales subcaracterísticas son de importancia, debido a que permiten identificar, corregir y mejorar un sistema de software de la mejor manera a lo largo del tiempo, asegurando una mejora continua en el sistema.

Finalmente, no se presentó alguna subcaracterística en común que haya sido considerada por todos los estudios. Sin embargo, la subcaracterística de facilidad de prueba fue la más nombrada en los estudios incluidos. De tal modo que, queda abierta la posibilidad de realizar una investigación adicional sobre dicha subcaracterística, ya que podría ser una pieza clave para saber más al respecto sobre la facilidad de evolución.

### **RQ3- ¿Cómo se mide software fácil de evolucionar o mantener?**

En esta pregunta de investigación, 21 de los 37 estudios considerados, fueron los que incluyeron maneras de medir la facilidad de evolución o mantenimiento. Es relevante señalar que, solo en algunos estudios se presentan tal cual métricas, en la mayoría únicamente se identifican maneras de medir que se podrían tener en consideración. Dichas maneras de medir tienen relación con ciertos elementos que conforman la facilidad de evolución o mantenimiento. Los hallazgos se describen a continuación:

En el EP-01-IEEE, Brcina, Bode y Riebisch (2009) identifican ciertos elementos que contribuyen a la facilidad de evolución de un sistema de software. Adicionalmente, los autores al desglosar la facilidad de evolución en atributos medibles mencionaron métricas, las cuales se pueden observar en la Figura 5. Es importante mencionar que en el estudio no se proponen métricas específicas para la facilidad de evolución.

Métricas relacionadas con los atributos medibles:

- Características entrelazadas (FTANG - Feature Tangling)
- Dispersión de características (FSCA - Feature Scattering)
- Número de componentes arquitectónicos (NOA - Number of Architectural Components)
- Número de características (NOF - Number of Features)
- Características aisladas (IF - Insulated Features)
- Componentes arquitectónicos aislados (IA - Insulated Architectural Components)
- Clases aisladas (IC - Insulated Classes)

En cuanto al estudio primario de Breivold, Crnkovic y Eriksson (2008) (EP-05-IEEE) se presentan subcaracterísticas identificadas para la facilidad de evolución (estas fueron explicadas a mayor detalle anteriormente en la RQ2), para las cuales mencionan atributos de medición, que pueden servir para medir dichas subcaracterísticas. Estos son los siguientes:

- Facilidad de análisis: Los atributos de medición incluyen modularidad, complejidad y documentación
- Integridad: Los atributos de medición incluyen documentación arquitectónica.
- Facilidad de cambio: Los atributos de medición incluyen la complejidad, el acoplamiento, el impacto del cambio, la encapsulación, la reutilización y la modularidad.
- Portabilidad: Los atributos de medición incluyen mecanismos que facilitan la adaptación a diferentes entornos.

- Extensibilidad: Los atributos de medición incluyen modularidad, acoplamiento, encapsulación, impacto del cambio.
- Facilidad de prueba: Los atributos de medición incluyen complejidad, modularidad.
- Atributos específicos del dominio: Los atributos de medición dependen de los dominios específicos.

Tal como se puede notar, los autores exponen algo similar a lo presentado por los autores (Brcina, et al. 2009), donde se debe desglosar la característica de facilidad de evolución, hasta obtener atributos que puedan ser medibles. No obstante, en este estudio los autores (Breivold, et al. 2008) solo se limitaron a mencionar aquellos atributos de medición, que incluyen las subcaracterísticas de la facilidad de evolución. Es decir, no establecieron en su estudio alguna definición o breve descripción para tales atributos.

Así mismo, en los estudios primarios de Breivold, Crnkovic, Land y Larsson (2008) (EP-07-IEEE), así como Breivold y Crnkovic (2008) (EP-11-IEEE), se vuelven a mencionar las subcaracterísticas listadas anteriormente (EP-05-IEEE), junto con sus atributos de medición relacionados los cuales son fundamentales para medir la facilidad de evolución. No obstante, de igual manera solo mencionan tales atributos de medición, sin explicarlos o presentar alguna métrica relacionada.

Por su parte, Ostberg y Wagner (2014) en su investigación (EP-13-IEEE), se presentan métricas de facilidad de mantenimiento de software. En su estudio primero mencionan aquellas que según su criterio consideran inapropiadas, posteriormente proponen un nuevo conjunto de métricas que consideran más adecuadas para la evaluación de la facilidad de mantenimiento del software. A continuación, se presentan las métricas que los autores consideraron inapropiadas para evaluar la facilidad de mantenimiento:

- Cyclomatic Complexity by McCabe: Los autores consideran inapropiada esta métrica debido a que mencionan que se debe distinguir entre la complejidad, del gráfico de flujo de control, derivado de las características estáticas del código y la complejidad para comprender el código. Además, comentan que la complejidad de McCabe aumenta con cada decisión, por ejemplo, en un bloque *if* dentro del código, no obstante, esto no necesariamente aumenta la complejidad percibida por el humano.
- Metrics by Halstead: Los autores critican estas métricas debido a que no son intuitivas y consideran que la mayoría de los desarrolladores de software puedan imaginar algo como el volumen del sistema. Tales métricas son referentes a la longitud del programa, volumen, esfuerzo, tiempo, etc.
- Lines of Code (LOC): Los autores mencionan que la creencia es que un código más grande y por lo tanto con más líneas de código, da como resultado menor facilidad de mantenimiento. Sin embargo, un código con menos líneas de código no estructurado es más difícil de mantener a

comparación de un código con varias líneas de código altamente estructurado.

- Maintainability Index (MI): Referente a esta métrica, la crítica va relacionada a que esta métrica hace uso de la métrica de Halstead (relacionada con el volumen del programa), McCabe y LOC, las cuales no consideran adecuadas, por lo mencionado con anterioridad.

A continuación, se presenta una lista de las métricas que los autores consideran adecuadas para medir la facilidad de mantenimiento del software. Es importante destacar que los autores comentan que, aunque no están exentas de problemas, proporcionan fundamentos más claros para usarlas en la evaluación de la facilidad de mantenimiento:

- Nesting Depth.
- Comment Ratio.
- Clone Coverage.
- Bug Patterns.
- Test Results and Coverage.
- Coupling and Cohesión.

Por otro lado, en el trabajo de investigación de Rochimah, Nuswantara y Akbar (2018) (EP-14-IEEE) se plantean puntos de evaluación para subcaracterísticas de facilidad de mantenimiento. Es importante decir, que en su estudio no dieron detalle acerca de cómo obtener algún valor para estos puntos de evaluación. A continuación, se muestran las subcaracterísticas de facilidad de mantenimiento, junto con sus puntos de evaluación:

- Modularidad:
  - Coupling of component conformance (acoplamiento de la conformidad de los componentes)
  - Cyclomatic complexity (Complejidad ciclomática)
- Reusabilidad:
  - Reusability of assets (Reusabilidad de activos)
  - Conformance to coding rule (Conformidad con la regla de codificación)
- Facilidad de análisis:
  - System log completeness conformance (Cumplimiento de la integridad del registro del sistema)
  - Diagnosis function effectiveness (Eficacia de la función de diagnóstico)
  - Diagnosis function sufficiency conformance (Conformidad de suficiencia de la función de diagnóstico)
- Facilidad de prueba:

- Test function completeness conformance (Función de prueba conformidad completa)
- Autonomous testability (Capacidad de prueba autónoma)
- Test restartability (Capacidad de reinicio de prueba)

En cuanto al estudio primario de Ping (2010) (EP-15-IEEE) se señalan subcaracterísticas que debe tener un producto de software para ser considerado fácil de mantener. Dichas subcaracterísticas de facilidad de mantenimiento, tienen relación con métricas, las cuales se listan a continuación:

- Facilidad de cambio, facilidad de prueba y facilidad de análisis:
  - Líneas de código (LOC)
  - Complejidad ciclomática (Cyclomatic Complexity)
- Estabilidad
  - Acoplamiento entre objetos (Coupling between objects)

Jun y Rana (2021) en el EP-16-IEEE, proponen que la facilidad de mantenimiento del software tiene en cuenta la complejidad, la modularidad, el código muerto o duplicado y el lenguaje de programación para medir la métrica de facilidad de mantenimiento. Se comenta que a cada atributo se le asigna un peso diferente, la complejidad se le asigna el 50% debido a que un código más complejo genera más costos para modificar y probar. La modularidad se le asigna el 30% debido a que se debe tener en cuenta el número de módulos y el tamaño del código que deben mantenerse. Por otro lado, el 20% restante se distribuye igual entre el lenguaje de programación y el código muerto, debido a que un lenguaje de programación de bajo nivel aumenta el costo de mantenimiento, y el código duplicado aumenta el costo de mantenimiento y presenta factores de riesgo de seguridad.

A continuación, se muestra la fórmula (1) que comparten los autores para medir la facilidad de mantenimiento:

$$(1) M = Co(0.5) + Mo(0.3) + Dp(0.1) + PI(0.1)$$

Donde:

- M = Facilidad de mantenimiento (Maintainability)
- Co = Complejidad (Complexity)
- Mo = Modularidad (Modularity)
- Dp = Código duplicado / muerto (Duplicate/Dead Code)
- PI = Lenguaje de programación (Programming Language)

Los autores comentan que la modularidad del software mejora la cohesión dentro de cada módulo, pero reduce el alto acoplamiento entre cada módulo en una solución de software. Mencionan que la puntuación de la modularidad se deriva del número de módulos (número de clases) y el tamaño del módulo (líneas de código en un módulo). A continuación, se muestra la fórmula (2) para calcular la modularidad:

$$(2) Mo = (Mn * 0.5) + (Ms * 0.5)$$

Donde:

- Mo = Modularidad (Modularity)
- Mn = Numero de módulos (Number of Modules)
- Ms = Tamaño del módulo (Module Size)

Referente a la complejidad, se comenta que afecta las conexiones internas de un programa, de tal modo que mientras más conexiones existan en un programa, más complejo será este. Esta característica, se mide en base a la complejidad ciclomática y el acoplamiento, mientras mayor sea el resultado, más difícil es de mantener el sistema:

$$(3) Co = (Vg * 0.5) + (Cp * 0.5)$$

Donde:

- Co = Complejidad (Complexity)
- Vg = Complejidad ciclomática (Cyclomatic Complexity)
- Cp = Acoplamiento (Coupling)

Por otro lado, referente al código duplicado o muerto se refiere a un código similar o copiado y pegado, lo que podría ser dañino ya que aumentaría los costos de mantenimiento. Además, el código muerto se refiere al código que nunca se usa e incluye métodos y variables, lo que puede que sea difícil de entender. Respecto al lenguaje de programación se comenta que ciertos lenguajes de programación son específicos de la plataforma, de tal modo que el software sea difícil de mantener. De estos últimos dos elementos, no se presentan más información sobre cómo obtener algún valor cuantitativo.

Los autores Aggarwal, Singh y Chhabra (2002) (EP-17-IEEE), comentan que un sistema de software se mantiene mediante el uso integrado de código fuente y los documentos relacionados. Afirman que la legibilidad del código y la calidad de la documentación son factores que deben tenerse en cuenta al medir la facilidad de mantenimiento del software. Es así como los autores proponen los siguientes factores para la medición de la facilidad de mantenimiento:

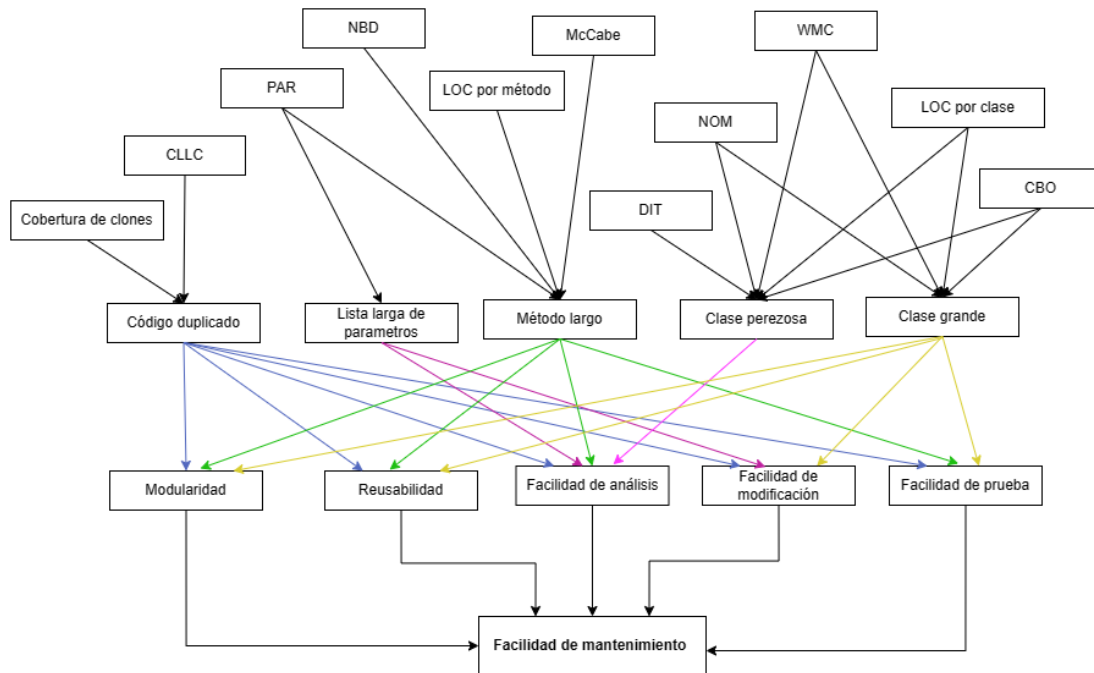
- Legibilidad del código fuente (Readability Of Source Code (RSC)).

- Calidad de la documentación (Documentation Quality (DOQ)).
- Comprensibilidad del software (Understandability of Software (UOS)).

Los autores argumentan que estas tres medidas presentadas anteriormente, son de naturaleza bastante subjetiva, de tal manera que se necesita alguna herramienta que no solo integre estos tres factores, sino que también maneje la naturaleza subjetiva de estos. La mejor opción es un Fuzzy Model para poder gestionar opiniones que pudiesen ser ambiguas, dudosas, contradictorias y divergentes. Los autores crearon 27 reglas, las cuales indican la facilidad de mantenimiento como muy buena / buena / promedio / pobre. A continuación, se muestra un ejemplo de estas reglas:

- Si (RSC es bueno) y (DOQ es alto) y (UOS es más) entonces (la facilidad de mantenimiento es muy buena)
- Si (RSC es promedio) y (DOQ es alto) y (UOS es más), entonces (la facilidad de mantenimiento es buena)
- Si (RSC es pobre) y (DOQ es alto) y (UOS es más) entonces (la facilidad de mantenimiento es promedio)
- Si (RSC es pobre) y (DOQ es bajo) y (UOS es menos) entonces (la facilidad de mantenimiento es pobre).

Por otro lado, Wagey, Hendradjaya y Mardiyanto (2015) en el EP-19-IEEE, proponen un gráfico de dependencia de atributos (ADG) en el cual el nivel bajo son los nodos sensores, donde cada uno de estos nodos sensores se elige según la conexión del siguiente nivel, el cual es donde se encuentran Code Smells, como atributos de nivel medio. Finalmente, el nivel alto son las subcaracterísticas de facilidad de mantenimiento basadas en el ISO 25010. A continuación, en la Figura 10 se presenta una adaptación del ADG que presentaron los autores:



**Figura 10.** Modelo de facilidad de mantenimiento con uso de Code Smells, Wagey, Hendradjaya & Mardiyanto, 2015.

Adaptado de ADG Maintainability model using code smell as middle level attribute, por Wagey, Hendradjaya & Mardiyanto, 2015.

En el estudio los autores solo decidieron utilizar 5 de los Code Smells de que presentó Fowler en 1999, argumentan que en investigaciones que realizaron (Geiger et al. (2006), Meananeatra et al. (2011), Bryton et al. (2010), Munro (2005), Crespo et al. (2005), Olbrich et al. (2010), Abilio et al. (2015)) encontraron conexión entre ciertas métricas de software y esos Code Smells, que se mencionan a continuación:

- Código duplicado.
- Lista larga de parámetros.
- Método largo.
- Clase perezosa.
- Clase grande.

Es importante mencionar que las métricas mencionadas en el estudio no presentan fórmulas o algo relevante para obtener los valores cuantitativos de éstas. Las métricas son las siguientes y se puede ver su relación con los Code Smells como se mostró anteriormente en la Figura 10:

- Cobertura de clonación (Clone Coverage (CC))
- Cobertura de línea lógica de clonación (Clone Logical Line Coverage (CLLC))



- Número de parámetros (Number of Parameter (PAR))
- Profundidad de bloque anidado (Nested Block Depth (NBD))
- Método línea de código (Method Line of Code (MLOC))
- Complejidad ciclomática de McCabe (McCabe Cyclomatic Complexity (MCC))
- Árbol de herencia de profundidad (Depth Inheritance Tree (DIT))
- Número de métodos (Number of Methods (NOM))
- Método ponderado en clase (Weighted Method in Class (WMC))
- Línea de código de clase (Class Line of Code (CLOC))
- Acoplamiento entre objetos (Coupling Between Objects (CBO))

En el trabajo de investigación de Chen, Alfayez, Srisopha, Boehm y Shi (2017) (EP-20-IEEE), se identifican métricas relacionadas con la facilidad de mantenimiento. Referente a la métrica de índice de facilidad de mantenimiento (MI), señalan que es una de las métricas más utilizadas en la industria para calcular el valor de la facilidad de mantenimiento del software. MI se calcula en función del volumen de Halstead, la complejidad ciclomática de McCabe y las líneas de código fuente, con una versión que incluye una relación código/comentario. De tal modo que, un MI más alto indica una mayor facilidad de mantenimiento, es decir, el código es más fácil de entender y los mantenedores pueden encontrar y corregir errores, realizar cambios o agregar nuevas funciones fácilmente al sistema.

Otra propuesta que mencionan los autores para medir la facilidad de mantenimiento del software es la deuda técnica. Según los autores Chen et al. (2017), la deuda técnica es una metáfora para explicar las consecuencias de tomar decisiones en el desarrollo de software no óptimas. Los autores también comentan que se ha demostrado que los Code Smells son un indicador importante que refleja aspectos de facilidad de mantenimiento de un sistema de software, al igual que otras métricas como las orientadas a objetos, como métodos ponderados por clase, falta de cohesión en los métodos, acoplamiento entre objetos, nivel de desacoplamiento, etc.

Los autores añaden que, en la práctica, la deuda técnica (TD) tiene la ventaja de poder identificar las partes particulares del software que en mayor medida necesitan mejorar la facilidad de mantenimiento. Por otro lado, las métricas de tipo índice de facilidad de mantenimiento (MI) tienen ventajas de localización y eficiencia similares, pero más a nivel de módulo que a nivel de línea de código. No obstante, en el estudio no se presentan fórmulas o algo similar para poder calcular el valor de la característica de facilidad de mantenimiento.

Perepletchikov y Ryan (2011) (EP-21-IEEE) identifican las subcaracterísticas de la facilidad de mantenimiento de un sistema de software, según el ISO 9126:

- Métrica de facilidad de análisis: Eficiencia del análisis de fallas (Failure Analysis Efficiency (FAE)).  
(4)  $FAE = SUMA(T)/N$ , donde T= Tiempo que toma analizar cada causa de falla (o tiempo necesario para localizar una falla de software) y N= número de fallas, de las cuales se encuentran causas. Cuanto más cerca de 0 sea el resultado, mejor.
- Métrica de facilidad de cambio: Complejidad de modificación (Modification Complexity (MC)).  
(5)  $MC = SUMA(T)/N$ , donde T= Tiempo de trabajo dedicado a cada cambio y N= número de cambios. Canto más cerca de 0 sea el resultado, mejor.
- Métrica de estabilidad: Localización del impacto de la modificación (Modification Impact Localization (MIL)).  
(6)  $MIL = A/B$ , donde A= número de impactos adversos emergentes(fallas) en el sistema después de las modificaciones y B= número de modificaciones realizadas. Cuanto más cerca de 0 sea el resultado, mejor.

Por otro lado, Saifan y Rabadi (2017) en su investigación (EP-22-IEEE) exploran elementos que afectan la facilidad de mantenimiento del software, así como métricas relacionadas. Los elementos son: facilidad de análisis, facilidad de cambio, estabilidad y facilidad de prueba. Es importante mencionar, que el estudio de los autores está relacionado con las aplicaciones para smartphones con sistema operativo Android. A continuación, se presenta una fórmula (7) que comparten los autores para calcular la facilidad de mantenimiento:

$$(7) \text{Facilidad de mantenimiento} = \text{Facilidad de análisis} + \text{Facilidad de cambio} + \text{Estabilidad} + \text{Facilidad de prueba}$$

Donde:

$$(8) \text{Facilidad de análisis} = cl\_wmc + cl\_comf + in\_bases + cu\_cdused$$

$$(9) \text{Facilidad de cambio} = cl\_stat + cl\_func + cl\_data$$

$$(10) \text{Estabilidad} = cl\_data\_publ + cu\_cdusers + in\_noc + cl\_func\_publ$$

$$(11) \text{Facilidad de prueba} = cl\_wmc + cl\_func + cu\_cdused$$

Para cada métrica, además de su definición, se listan los factores a los que afecta, así como su rango:

- **Cl\_wmc:**
  - Definición: Método ponderado por clase, mide la complejidad del método.
  - Afecta: Complejidad, comprensibilidad y cohesión.
  - Rango: 0-11
- **In\_bases:**
  - Definición: Número de clases de las que la clase hereda directamente o no.
  - Afecta: Complejidad y comprensibilidad.
  - Rango: 0-4
- **Cu\_cdused:**
  - Definición: El número de clases directamente utilizado por esta clase.
  - Afecta: Acoplamiento, la complejidad y reutilización de métodos.
  - Rango: 0-6
- **Cl\_cmof:**
  - Definición: Relación entre el número de líneas de comentarios y el número de líneas del código.
  - Afecta: El tamaño de la clase.
  - Rango: 0-100
- **Cl\_stat:**
  - Definición: Número de sentencias ejecutables.
  - Afecta: El tamaño de la clase.
  - Rango: 0-7
- **Cl\_data:**
  - Definición: El número total de atributos en la clase.
  - Afecta: Complejidad y el tamaño de la clase.
  - Rango: 0-25
- **Cl\_fun:**
  - Definición: El número total de funciones en la clase.
  - Afecta: Complejidad y cohesión.
  - Rango: 0-9
- **Cl\_fun\_publ:**
  - Definición: El número total de funciones públicas en la clase.
  - Afecta: Acoplamiento y polimorfismo.
  - Rango: 0-7
- **Cu\_cdusers:**
  - Definición: El número de clases que utilizan directamente esta clase.
  - Afecta: Acoplamiento, facilidad de cambio y reutilización de métodos.
  - Rango: 0-3
- **In\_noc:**

- Definición: El número de hijos de esta clase.
- Afecta: Reutilización de métodos y pruebas.
- Rango: 0-5
- Cl\_data\_publ:
  - Definición: El número total de atributos públicos en la clase.
  - Afecta: Encapsulación (Ocultación de información).
  - Rango: 0-7

Referente al EP-25-IEEE, Heitlager, Kuipers y Visser (2007) señalan medidas para calcular la facilidad de mantenimiento de un sistema de software, donde se dice que dichas métricas se basan en una comparación de las funciones requeridas y las funciones implementadas hasta el momento. A continuación, se presentan ejemplos proporcionados en el estudio, cabe destacar, que no se mencionaron medidas para todas las subcaracterísticas mencionadas en el estudio:

- Facilidad de análisis: Se sugiere la medida de "Registro de actividad", la cual es definida como la relación entre la cantidad de elementos de datos para los que se implementa el registro, frente a la cantidad de elementos de datos para los que las especificaciones requieren registro.
- Facilidad de cambio: Se sugiere la medida de "Impacto del cambio", la cual se calcula a partir del número de modificaciones realizadas y el número de problemas causados por estas modificaciones.

Hincheeranan y Rivepiboon (2012) en su trabajo de investigación (EP-26-IEEE), proponen que la flexibilidad y la extensibilidad son subcaracterísticas de la facilidad de mantenimiento del software tal como se presentó anteriormente en la Figura 7 (Hincheeranan et al. (2012)). Además, se presentan propiedades de diseño relacionadas con estas subcaracterísticas, dichas propiedades de diseño tienen relación con las métricas:

- Métrica de acceso de datos (DAM)
- Acoplamiento directo de clases (DCC)
- Medida de agregación (MOA)
- Número de métodos polimórficos (NOP)
- Número promedio de antepasados (ANA)
- Medida de abstracción funcional (MFA)

Los autores solo se limitaron a mencionar dichas métricas, no presentan fórmulas o definiciones relacionadas con éstas, sin embargo, los autores presentan las siguientes dos fórmulas (12) (13) para el cálculo de los atributos de calidad identificados: flexibilidad y extensibilidad.

$$(12) \text{ Flexibilidad} = 0.25 * \text{Encapsulación} - 0.25 * \text{Acoplamiento} + 0.5 * \text{Composición} + 0.5 * \text{Polimorfismo}$$

$$(13) \text{ Extensibilidad} = 0.5 * \text{Abstracción} - 0.5 * \text{Acoplamiento} + 0.5 * \text{Herencia} + 0.5 * \text{Polimorfismo}$$

Por otro lado, Ganpati, Kalia y Singh (2012) (EP-27-IEEE) realizaron un estudio comparativo de cuatro sistemas de software de código abierto. En dicho estudio, se expone que la métrica de software más utilizada para calcular la facilidad de mantenimiento es el índice de facilidad de mantenimiento (MI). Esta métrica consiste en una expresión polinomial la cual da como resultado un número que indica que tan fácil de mantener es un sistema de software. Los autores mencionan que un valor de MI superior a 85 indica que el sistema es altamente fácil de mantener, si el valor es entre 85 y 65, sugiere que es una facilidad de mantenimiento moderada, un valor por debajo de 65 indica un sistema difícil de mantener. A continuación, se presenta la formula (14) mostrada en el estudio:

$$(14) MI = 171 - 5.2 * \ln(aveV) - 0.23 * aveV(g) - 16.2 * \ln(aveLOC)$$

Donde:

- aveV = Volumen promedio Halstead (Average Halstead Volume)
- aveV(g) = Complejidad ciclomática promedio por módulo (Average Cyclomatic Complexity Per Module)
- aveLOC = Promedio de líneas de código por módulo (Average Lines of Code per Module)

Najm (2014) en su estudio primario (EP-28-IEEE) presentan un nuevo método para encontrar el valor de la métrica de índice de facilidad de mantenimiento (MI) únicamente con respecto a LOC (líneas de código). MI, es una medida para la facilidad de mantenimiento, la cual se compone de métricas ponderadas de Halstead (HV), Complejidad ciclomática de McCabe (CC) y Líneas de código (LOC). A continuación, se presenta lo necesario para calcular el MI con la fórmula original (15), y posterior a ello, la fórmula (18) que propone el autor basándose únicamente en LOC:

Para la fórmula original, es necesario primero medir las siguientes métricas en el código fuente del sistema:

- HV = Volumen de Halstead
- CC = Complejidad ciclomática
- LOC = Recuento de líneas de código fuente
- CM = Porcentaje de líneas de comentario (opcional)

Fórmula original:

$$(15) MI = 171 - 5.2 * \ln(HV) - 0.23 * (CC) - 16.2 * \ln(LOC)$$

Referente a la fórmula sugerida, el autor descubrió que LOC es el factor más importante en MI, además, LOC tiene ventajas como poder automatizar el proceso de conteo de las líneas de código, y que es una métrica intuitiva. Para calcular la fórmula sugerida es necesario considerar que:

$$(16) HV = 45 * LOC - 428$$

$$(17) CC = 0.22 * LOC + 1.9$$

De tal modo que, el autor concluye la siguiente ecuación sugerida:

$$(18) MI = 171 - 5.2\ln(HV) - 0.23 * (CC - 16.2\ln(LOC))$$

En el estudio de Bibi, Ampatzoglou y Stamelos (2016) (EP-04-SPRINGER) se identifican métricas de software, las cuales funcionan como predictores de la facilidad de mantenimiento de un sistema. En el estudio solo se limitan a mencionarlas junto con una breve descripción, la fase de desarrollo a la cual pertenecen y el atributo de calidad con el cual tienen relación. A continuación, se listan estas métricas junto con los datos relacionados:

- Profundidad del árbol de herencia (DIT): Número de nivel de herencia, donde 0 es para la clase raíz.
  - Fase: Diseño
  - Atributo de calidad: Herencia
- Numero de clases hijos (NoCC): Número de subclases que tiene la clase.
  - Fase: Diseño
  - Atributo de calidad: Herencia
- Acoplamiento de paso de mensajes (MPC): Número de sentencias de envío definidas en la clase.
  - Fase: Construcción
  - Atributo de calidad: Acoplamiento
- Respuesta para una clase (RFC): Número de métodos locales más el número de métodos llamado por métodos locales en la clase.
  - Fase: Construcción
  - Atributo de calidad: Acoplamiento
- Falta de cohesión de métodos (LCOM): Número de conjuntos de métodos que no interactúan entre sí en la clase.
  - Fase: Construcción
  - Atributo de calidad: Cohesión
- Acoplamiento de abstracción de datos (DAC): Número de tipos abstractos definidos en la clase.
  - Fase: Diseño
  - Atributo de calidad: Acoplamiento

- Método ponderado por clase (WMC): Complejidad ciclomática promedio de todos los métodos.
  - Fase: Construcción
  - Atributo de calidad: Complejidad
- Número de métodos (NOM): Número de métodos en la clase.
  - Fase: Diseño
  - Atributo de calidad: Tamaño
- Líneas de código (SIZE1/LOC): Número de punto y coma en la clase.
  - Fase: Construcción
  - Atributo de calidad: Tamaño
- Número de propiedades (SIZE2): Número de atributos y métodos en la clase.
  - Fase: Diseño
  - Atributo de calidad: Tamaño
- Acoplamiento entre objetos (CBO): Número de clases de las que una clase depende.
  - Fase: Diseño
  - Atributo de calidad: Acoplamiento
- Tamaño promedio del método (AMS): Número promedio de punto y coma en un método.
  - Fase: Construcción
  - Atributo de calidad: Complejidad

Sjoberg, Anda y Mockus (2012) en su trabajo de investigación (EP-02-ACM) exploran un conjunto de métricas para evaluar la facilidad de mantenimiento de un sistema de software, comentan que las seleccionaron debido a que están entre las más utilizadas y conocidas. A continuación, se listan estas métricas que mencionan los autores. Cabe aclarar, que no proporcionaron alguna definición o fórmulas para obtener resultados cuantitativos.

- Índice de facilidad de mantenimiento (MI).
- Medidas estructurales (SM): Los autores comentan que el conjunto más común de métricas para evaluar la facilidad de mantenimiento de código, son las medidas estructurales (SM), donde se incluyen las métricas CK. Se mencionan un subconjunto de las métricas CK las cuales incluye:
  - La medida de acoplamiento OMMIC (llamada a métodos en una clase no relacionada)
  - La medida de cohesión TCC (cohesión de una clase estrecha)
  - La medida de tamaño de clases WMC1 (número de métodos por clase)
  - Profundidad del árbol de herencia (DIT).
- Code Smells.
  - Feature Envy

- Good Class

Por otro lado, Gopalakrishnan, Aravindh y Selvarani (2010) en el EP-03-ACM investigan métricas CK, donde comentan que las métricas de diseño internas de la tecnología orientada a objetos están dirigidas principalmente por estas métricas, las cuales tienen un impacto en la facilidad de mantenimiento del software. Además, los autores mencionan dichas métricas junto con su umbral de valores recomendado. No obstante, no se presentan fórmulas o descripciones de dichas métricas. Éstas se presentan a continuación:

- WMC (Método Ponderado por Clase) - Umbral: Óptimo 20 y Máximo 100.
- RFC (Respuesta por Clase): Umbral: Óptimo 50 a 100 y Máximo 222.
- LCOM (Falta de Cohesión en los Métodos) - Umbral: [0, 1].
- CBO (Acoplamiento Entre Objetos): Umbral: Óptimo -5 y Máximo- 24.
- DIT (Profundidad del Árbol de herencia): Umbral: Óptimo-3 y Máximo-6.
- NOC (Número de hijos) Umbral: Máximo-5.

Kurmangali, Rana y Rahman (2022) (EP-05-ACM) realizaron un trabajo de investigación donde evalúan la facilidad de mantenimiento antes y después de aplicar patrones de diseño seleccionados. En este trabajo también, se mencionan métricas relacionadas con la facilidad de mantenimiento, que se utilizaron en el estudio para evaluar las soluciones de software, sin mostrar fórmulas de éstas. A continuación, se presentan las métricas empleadas en el estudio, junto con una breve descripción:

- WMC: métodos ponderados para una clase, vinculados con la complejidad y la facilidad de mantenimiento, cuyo valor bajo indica una mejor calidad del software.
- DIT: profundidad del árbol de herencia, indica la proximidad de una clase medida a su clase raíz.
- NOC: número de hijos de una clase.
- CBO: acoplamiento entre clases de objetos.
- RFC: respuesta para una clase, es decir, el número de posibles métodos invocados en respuesta a cualquier mensaje recibido por la clase.
- LCOM: falta de cohesión entre métodos, calcula el número de métodos que no están relacionados entre sí o son diferentes.

Finalmente, los autores comentan que se considera que cuanto más bajos son los resultados de las métricas anteriores, menos complejo y más fácil de mantener es el software.



Vistas las respuestas a esta RQ3, se puede apreciar que existen varias métricas relacionadas con la facilidad de evolución o facilidad de mantenimiento. A continuación, en la Tabla 5 se muestra de manera resumida, las métricas relacionadas con facilidad de evolución o facilidad de mantenimiento, que se encontraron en los estudios primarios.

*Tabla 5. Métricas relacionadas con facilidad de evolución o facilidad de mantenimiento*

Estudio	Autor (es)	Métrica (s)
EP-01-IEEE	Robert Brcina, Stephan Bode, Matthias Riebisch	FTANG - Feature Tangling FSCA - Feature Scattering NOA - Number of Architectural Components NOF - Number of Features IF - Insulated Features IA - Insulated Architectural Componets IC - Insulated Classes
EP-13-IEEE	J. -P. Ostberg; S. Wagner	CC - Cyclomatic Complexity Metrics by Halstead LOC - Lines of Code MI - Maintainability Index Nesting Depth Comment Ratio Clone Coverage Bug Patterns Test Results Test Coverage
EP-14-IEEE	S. Rochimah; P. G. Nuswantara; R. J. Akbar	Coupling of component conformance CC - Clyclomatic Complexity Reusability of assets Conformance to coding rule System log completness conformance Diagnosis function effectiveness Diagnosis function sufficiency conformance Test function completness conformance Autonomous testability Test restartability
EP-15-IEEE	L. Ping	LOC - Lines of Code CC - Cyclomatic Complexity CBO - Coupling between objects
EP-16-IEEE	H. K. Jun; M. E. Rana	Maintainability Metric
EP-17-IEEE	K. K. Aggarwal; Y. Singh; J. K. Chhabra	RSC - Readability of Source Code DOQ - Documentation Quality UOS - Understandability of Software
EP-19-IEEE	B. C. Wagey; B. Hendradjaya; M. S. Mardiyanto	CC - Clone Coverage CLLC - Clone Logical Line Coverage PAR - Number of Parameter NBD - Nested Block Depth MLOC - Method Line of Code MCC - McCabe Cyclomatic Complexity DIT - Depth Inheritance Tree NOM - Number of Methods WMC - Weighted Method in Class CLOC - Class Line of Code CBO - Coupling Between Objects
EP-20-IEEE	C. Chen; R. Alfayez; K. Srisopha; B. Boehm; L. Shi	MI - Maintainability Index TD - Technical Debt
EP-21-IEEE	M. Pereplechikov; C. Ryan	FAE - Failure Analysis Efficiency MC - Modification Complexity

		MIL - Modification Impact Localization
EP-22-IEEE	A. A. Saifan; A. Al-Rabadi	CI_WMC IN_BASES CU_CDUSED CI_CMOF CI_STAT CI_DATA CI_FUN CI_FUN_PUBL CU_CDUSERS IN_NOC CI_DATA_PUBL
EP-25-IEEE	I. Heitlager, T. Kuipers, and J. Visser	Registro de actividad Impacto del cambio
EP-26-IEEE	A. Hinceeranan and W. Rivepiboon	DAM - Data Access Metric DCC - Direct Class Coupling MOA - Measure of Aggregation NOP - Number of Polymorphic Methods ANA - Average Number of Ancestors DCC - Direct Class Coupling MFA - Measure of Functional Abstraction NOP - Number of Polymorphic Methods
EP-27-IEEE	Ganpati, Anita, A. Kalia, and H. Singh.	MI - Maintainability Index
EP-28-IEEE	Najm, N.	MI - Maintainability Index
EP-04- SPRINGER	Stamatia Bibi, Apostolos Ampatzoglou, Ioannis Stamelos	DIT - Depth Inheritance Tree NOCC - Number of Children Classes MPC - Message Passing Coupling RFC- Response for a Class LCOM - Lack of Cohesion of Methods DAC - Data Abstraction Coupling WMC - Weighted Method per Class NOM - Number of Methods LOC - Lines of Code SIZE2 - Number of Properties CBO - Coupling Between Objects AMS - Average Method Size
EP-02-ACM	Dag I.K. Sjoberg, Bente Anda & Audris Mockus	MI - Maintainability Index SM - Structural Measures CK Metrics OMMIC TCC - Tight Class Cohesion WMC - Weighted Method per Class DIT - Depth Inheritance Tree
EP-03-ACM	T.R. Gopalakrishnan Nair, Sri Aravindh & R.Selvarani	CK Metrics WMC - Weighted Method per Class RFC- Response for a Class LCOM - Lack of Cohesion of Methods CBO - Coupling Between Objects DIT - Depth Inheritance Tree NOCC - Number of Children Classes
EP-05-ACM	Kurmangali A, Rana M and Ab Rahman W.	WMC - Weighted Method per Class DIT - Depth Inheritance Tree NOCC - Number of Children Classes CBO - Coupling Between Objects RFC- Response for a Class LCOM - Lack of Cohesion of Methods

Se encontraron gran variedad de métricas relacionadas con la facilidad de evolución de un sistema de software, aunque es importante aclarar, que ciertas métricas son más próximas a la facilidad de evolución que otras. Entre las métricas más identificadas en los estudios, se pueden encontrar LOC, CC, CBO, MI, DIT y WMC. Métricas como LOC, CC y CBO, tienen impacto principalmente en el acoplamiento y polimorfismo de un sistema, mientras que DIT y WMC, están relacionadas con la herencia y la complejidad respectivamente. La métrica de MI es la que mayormente se menciona y para saber el valor de dicha métrica es necesario conocer elementos como el volumen, las líneas de código (LOC), la relación entre comentarios y la complejidad ciclomática (CC).

Finalmente, queda abierta la posibilidad de investigar estas métricas más mencionadas en los estudios. Esto ayudaría a comprender mejor el impacto de dichas métricas en los sistemas de software, específicamente en la facilidad de evolución.

#### **RQ4- ¿Cómo se diseña software fácil de evolucionar o mantener?**

En esta pregunta de investigación, 5 de los 37 estudios incluidos, son los que proponen maneras para poder diseñar un sistema de software considerando la facilidad de evolución o mantenimiento. Es relevante destacar que, en los estudios no se mencionan estrategias, métodos, procesos o frameworks para el diseño con miras a la evolución o mantenimiento del sistema. Sólo se presentan elementos a considerar, y en algunos casos ciertos patrones de diseño. Aunque los estudios no indicaron la aplicación de estos elementos y patrones durante la fase de diseño, su naturaleza sugiere que deben considerarse principalmente en esa etapa. Los hallazgos se discuten a continuación:

Brcina, Bode y Riebisch (2009) en el EP-01-IEEE, identifican principios de software a considerar durante el desarrollo para la facilidad de evolución. Sin embargo, no se explican a detalle, tampoco se menciona alguna estrategia o patrón para lograr o cumplir con estos elementos:

- Separación de intereses
- Consistencia
- Modularidad
- Exactitud
- Complejidad
- Completitud
- Acoplamiento
- Encapsulación
- Impacto del cambio
- Permeabilidad
- Tipificación

Es importante considerar los elementos anteriores durante el desarrollo, ya que, si bien los autores no proponen explícitamente incluirlos en la etapa de diseño, en el estudio se entiende que se deben contemplar en todas las etapas. El hecho de tener en cuenta dichos elementos, resalta lo obtenido en las respuestas a las RQ anteriores, en las que la modularidad, acoplamiento, consistencia, impacto al cambio, juegan un rol importante en la facilidad de evolución o mantenimiento.

Por otro lado, en el estudio primario de Kumar y Singh (2017) (EP-03-IEEE) se plantean características importantes para lograr la facilidad de evolución del software. Las cuales se consideran como variables de entrada, que conducen a una variable de salida, facilidad de evolución.

Entrada:

- Extensibilidad

- Estabilidad de diseño
- Sustentabilidad
- Configurabilidad

Salida:

- Facilidad de evolución

Se deben considerar estas variables durante las etapas de desarrollo, incluida la de diseño, ya que establecen una base sólida para la evolución del sistema y se necesitan para que la salida sea la facilidad de evolución. Del mismo modo, se podría intuir que al no considerar alguna de estas variables de entrada, se podría afectar el resultado de la variable de salida.

Jun y Rana (2021) en su estudio primario (EP-16-IEEE), analizan y evalúan dos soluciones de software en términos de atributos de calidad de facilidad de mantenimiento del software. La primera solución es una solución simple, es decir, una solución que no incorpora patrones de diseño. Por el contrario, la segunda solución de software es una solución refinada, es decir, incluye la aplicación de los siguientes tres patrones de diseño:

- Factory Method: Un patrón de diseño de creación que define una interfaz para crear objetos en una superclase, pero permite que las subclases modifiquen el tipo de objetos a crear.
- Singleton: Un patrón de diseño creacional para garantizar que una clase tenga exactamente una instancia y proporcione un punto de acceso global.
- Decorator: Un patrón de diseño estructural que proporciona una forma de modificar el comportamiento de objetos individuales sin la necesidad de crear una nueva clase derivada. Proporciona una alternativa a las subclases para una funcionalidad amplia flexible. Además, contribuye al principio de software open-closed, abierto para extensión, pero cerrado para modificación.

Posterior a aplicar tales patrones de diseño, los autores comentan que el resultado dado es positivo, ya que la aplicación y el uso efectivo de patrones de diseño ayudan a producir una mejor solución con alta calidad. Agregan que los patrones de diseño, especialmente en el factor de facilidad de mantenimiento, mejoran significativamente la calidad de la solución de software.

Por otro lado, en el trabajo de investigación de Bode y Riebisch (2010) (EP-02-SPRINGER) se proponen subcaracterísticas relacionadas con la facilidad de evolución de un sistema de software. Aquí se presentan propiedades de un buen diseño arquitectónico relacionadas con la facilidad de evolución:

- Baja complejidad
  - Abstracción

- Modularidad
  - Cohesión
  - Bajo acoplamiento
- Encapsulación
- Separación de intereses
- Herencia
- Simplicidad
- Exactitud
  - Consistencia
  - Completitud
- Integridad conceptual
- Granularidad adecuada
- Mapeo coherente a conceptos

Si bien lo presentado no son patrones o estrategias para diseñar software fácil de evolucionar, en el estudio se mencionan dichas propiedades para un buen diseño arquitectónico. Esto debido a que se sugiere que, al considerarlas dichas propiedades, se obtiene un beneficio en la arquitectura del sistema, lo cual, visto en varias definiciones de facilidad de evolución o mantenimiento, es algo crítico e importante.

Kurmangali, Rana y Rahman (2022) en el EP-05-ACM, comentan el tema de patrones de diseño, donde para demostrar el impacto de los patrones de diseño en la facilidad de mantenimiento del software, derivaron un escenario simple. En dicho escenario, se diseñaría e implementaría en Java una solución simple sin ningún patrón de diseño aplicado. Posteriormente, se aplicarían patrones de diseño como Abstract Factory y Decorator en un intento de mejorar la facilidad de mantenimiento de la primera solución de software.

En el estudio se señala que con el patrón Abstract Factory aporta un nivel adicional de abstracción a un sistema, lo que podría ser bueno para su facilidad de mantenimiento. Además, asegura un código limpio y ordenado con el cumplimiento del principio abierto-cerrado y de responsabilidad única. Respecto al patrón Decorator permite “adjuntar dinámicamente funcionalidades adicionales a un objeto” y hace que la extensión de características o comportamiento sea más fácil de mantener que la herencia. Finalmente se comenta que luego de diseñar el escenario del problema y aplicar los dos patrones de diseño Abstract Factory y Decorator, el análisis mostró que la aplicación de patrones de diseño influyó positivamente en la facilidad de mantenimiento del software.

Analizadas las respuestas a esta RQ4, se puede notar que algunos elementos son importantes al momento de diseñar software fácil de evolucionar o mantener (ver Tabla 6).

**Tabla 6.** Elementos importantes para diseñar software fácil de evolucionar o mantener

Estudio	Autor (es)	Elementos
EP-01-IEEE	Robert Brcina, Stephan Bode, Matthias Riebisch	Separación de intereses Consistencia Modularidad Exactitud Baja complejidad Compleitud Bajo acoplamiento Encapsulación Impacto del cambio Permeabilidad Tipificación
EP-03-IEEE	P. Kumar; S. K. Singh	Extensibilidad Estabilidad de diseño Sustentabilidad Configurabilidad
EP-02-Springer	Stephan Bode & Matthias Riebisch	Baja complejidad Abstracción Modularidad Cohesión Bajo acoplamiento Encapsulación Separación de intereses Herencia Simplicidad Exactitud Consistencia Compleitud Integridad conceptual Granularidad adecuada Mapeo coherente a conceptos

La separación de intereses, consistencia, modularidad, exactitud, baja complejidad, completitud, bajo acoplamiento y la encapsulación, son elementos importantes para considerar pensando en evolucionar o mantener un sistema de software. De igual manera, sería importante considerar las subcaracterísticas presentadas en las respuestas de la RQ2, ya que dichas subcaracterísticas también pueden ser un factor clave para diseñar pensando en la evolución o mantenimiento.

Por otro lado, también se encontró que algunos patrones de diseño pueden contribuir de manera positiva el diseño de un sistema fácil de evolucionar o mantener. A continuación, en la Tabla 7 se muestran los patrones de diseño encontrados en los estudios, que facilitan la evolución o mantenimiento de un sistema:

*Tabla 7. Patrones de diseño relacionados con la facilidad de evolución o mantenimiento*

Estudio	Autor (es)	Patrón (es) de diseño
EP-16-IEEE	H. K. Jun; M. E. Rana	Factory Method Singleton Decorator
EP-05-ACM	Kurmangali A, Rana M and Ab Rahman W.	Abstract Factory Decorator

Como se puede notar, el patrón de diseño Decorator se mencionó en ambos estudios, ya que contribuye a la facilidad de evolución o mantenimiento. Los otros tres patrones de diseño mostrados en la anterior Tabla 9, únicamente se mencionan una vez en los estudios. Lo cual aún puede dejar la incógnita de si en verdad contribuyen a la facilidad de evolución o mantenimiento de un sistema de software, o únicamente fueron factibles a las soluciones de software que fueron aplicados.

También se debe tener en cuenta que, al querer aplicar un patrón de diseño a un sistema de software, debe ser de acuerdo con las características del proyecto y sus necesidades. Puesto que no por aplicar patrones de diseño, se asegura que un sistema será fácil de evolucionar o mantener, se debe realizar un análisis previo para ver si el patrón de diseño cumple las necesidades. Finalmente, queda abierta la posibilidad de una investigación más enfocada en los patrones de diseño y su impacto positivo o negativo en la facilidad de evolución de un sistema de software. Además, es importante destacar, que no se encontraron estrategias, frameworks, métodos o procesos para diseñar software fácil de evolucionar o mantener un sistema.



## **RQ5- ¿Cuáles han sido las experiencias o resultados obtenidos al considerar la facilidad de evolución o mantenimiento en el desarrollo de software?**

Esta pregunta de investigación fue respondida únicamente por 3 de los 37 estudios incluidos. Cabe señalar que, las experiencias o resultados obtenidos considerados, son pocos. Los hallazgos a esta RQ se presentan a continuación:

Breivold, Crnkovic, Land y Larsson (2008) en el EP-04-IEEE, notaron mejoras visibles en la organización donde aplicaron el método ARchitecture Evolvability Analysis (AREA), que permite analizar la facilidad de evolución a nivel arquitectónico mediante las siguientes tres fases:

- **Fase 1:** Analizar las implicaciones de los estímulos de cambio en los requisitos en la arquitectura de software. Esta fase analiza la arquitectura para la evolución y comprende el impacto de los estímulos de cambio en la arquitectura actual.
- **Fase 2:** Analizar y preparar la arquitectura de software para adaptarse a los estímulos de cambio y posibles cambios futuros. Esta fase se centra en la identificación y mejora de los componentes que necesitan ser refactorizados
- **Fase 3:** Finalizar la evaluación. En esta fase, los resultados anteriores se incorporan, analizan y estructuran en una colección de documentos.

Las mejoras visibles que notaron los autores al aplicar el método AREA son las siguientes:

- Mejora en la documentación de la arquitectura, incluida la ruta de evolución de la arquitectura. Esto debido a que la transformación de la arquitectura y las sugerencias de soluciones de refactorización fueron parte del proceso de análisis de dicho método. De tal modo que, como resultado de las actividades de análisis y refactorización, se ha mejorado la documentación de las propuestas de solución de diseño e implementación. Gracias a esto, el equipo central de arquitectura y los equipos de implementación compartieron la misma visión sobre el camino de evolución de la arquitectura de software.
- Junto con la documentación de la ruta de evolución de la arquitectura, se enriquecen los modelos arquitectónicos y se facilita la trazabilidad de la evolución de la arquitectura del software.

Por otro lado, en el estudio primario de Jun y Rana (2021) (EP-16-IEEE), se midió la facilidad de mantenimiento de un sistema de software. Dicha métrica toma en cuenta la complejidad, modularidad, código muerto o duplicado y el lenguaje de programación, donde estos factores influyen en qué tan fácil o difícil es modificar o mantener un sistema de software. Los autores midieron dos soluciones de software,

una sin refinar y otra refinada, es decir, una la cual no incorpora patrones de diseño, y otra a la cual sí aplicaron patrones de diseño.

Al aplicar los patrones Factory, Singleton y Decorator al sistema de software, Jun y Rana obtuvieron resultados favorables, donde los valores para modularidad, número de módulos, tamaño de los módulos, complejidad, complejidad ciclomática, acopamiento, son mayores en el software donde no aplicaron los patrones de diseño. De tal modo que, el valor final de la facilidad de mantenimiento fue menor en el software donde se aplicaron los patrones de diseño, lo que indica que es más fácil de mantener.

En el EP-05-ACM, Kurmangali, Rana y Rahman (2022) aplican patrones de diseño a un sistema de software, centrándose en la mejora de la facilidad de mantenimiento. Comenzaron diseñando un escenario simple donde inicialmente se creó una solución en Java sin la implementación de ningún patrón de diseño, luego incorporaron patrones como Abstract Factory y Decorator con el objetivo de mejorar la facilidad de mantenimiento. Descubrieron que Abstract Factory añade un nivel extra de abstracción al sistema, donde no solo se contribuye a un código más ordenado, sino que también cumple con los principios de diseño, como el abierto-cerrado y la responsabilidad única. El patrón Decorator, permitió la incorporación dinámica de funcionalidades adicionales a un objeto, facilitando de manera notoria la extensión de características y comportamientos. Es así como, tras diseñar el escenario del problema y aplicar los patrones Abstract Factory y Decorator, la evaluación revela que la experiencia de aplicar estos patrones de diseño influyó positivamente en la facilidad de mantenimiento del software.

Esta RQ5 fue poco respondida, lo cual podría deberse al estado actual en el área, es interesante notar que existe escasez de investigaciones respecto a este tema. De igual manera, a pesar de que la facilidad de evolución y facilidad de mantenimiento es esencial para un sistema de software, no se le ha dado la consideración que debería tener. De tal modo que, se puede decir que, para identificar experiencias o resultados al tener en cuenta la facilidad de evolución o mantenimiento, primeramente, es fundamental considerar y priorizar dicha característica al desarrollar sistemas de software.

## 4. Referencias

- Aggarwal, K. K., Singh, Y., & Chhabra, J. K. (2002). An integrated measure of software maintainability. *In Annual Reliability and Maintainability Symposium. 2002 Proceedings*, 235-241. <https://doi.org/10.1109/RAMS.2002.981648>
- Al-Sarayreh, K. T., Labadi, A., & Meridji, K. (2015). A Generic Method for Identifying Maintainability Requirements Using ISO Standards. *In Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, 1-6. <https://doi.org/10.1145/2816839.2816929>
- Arciniegas H, J. L., & Dueñas L, J. C. (2010). Evolvability Characterization in the Context of SOA. *In Advances in Software Engineering: International Conference, ASEA 2010, Held as Part of the Future Generation Information Technology Conference, FGIT 2010, Jeju Island, Korea, December 13-15, 2010*, 242-253. [https://doi.org/10.1007/978-3-642-17578-7\\_25](https://doi.org/10.1007/978-3-642-17578-7_25)
- Bibi, S., Ampatzoglou, A., & Stamelos, I. (2016). A Bayesian belief network for modeling open source software maintenance productivity. *In Open Source Systems: Integrating Communities: 12th IFIP WG 2.13 International Conference, OSS 2016, Gothenburg, Sweden, 32-44*. [https://doi.org/10.1007/978-3-319-39225-7\\_3](https://doi.org/10.1007/978-3-319-39225-7_3)
- Bode, S., & Riebisch, M. (2010). Impact evaluation for quality-oriented architectural decisions regarding evolvability. *In European Conference on Software Architecture*, 182-197. [https://doi.org/10.1007/978-3-642-15114-9\\_15](https://doi.org/10.1007/978-3-642-15114-9_15)
- Bogner, J., Wagner, S., & Zimmermann, A. (2017). Towards a practical maintainability quality model for service-and microservice-based systems. *In Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, 195-198. <https://doi.org/10.1145/3129790.3129816>
- Brcina, R., Bode, S., & Riebisch, M. (2009). Optimisation process for maintaining evolvability during software evolution. *In 2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 196-205. <https://doi.org/10.1109/ECBS.2009.20>
- Breivold, H. P., & Crnkovic, I. (2009). Analysis of software evolvability in quality models. *In 2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, 279-282. <https://doi.org/10.1109/SEAA.2009.10>
- Breivold, H. P., Crnkovic, I., Land, R., & Larsson, M. (2008). Analyzing software evolvability of an industrial automation control system: A case study. *In 2008 The Third International Conference on Software Engineering Advances*, 205-213. <https://doi.org/10.1109/ICSEA.2008.16>

- Breivold, H. P., Crnkovic, I., Land, R., & Larsson, S. (2008). Using dependency model to support software architecture evolution. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering-Workshops*, 82-91. <https://doi.org/10.1109/ASEW.2008.4686324>
- Breivold, H. P., & Crnkovic, I. (2008). Using software evolvability model for evolvability analysis. *Mälardalen University*, 1-10.
- Breivold, H. P., Crnkovic, I., & Eriksson, P. (2007). Evaluating software evolvability. *Software Engineering Research and Practice in Sweden*, 96-103.
- Breivold, H. P., Crnkovic, I. (2009). Analysis of software evolvability in quality models. In *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, 279-282. <https://doi.org/10.1109/SEAA.2009.10>
- Breivold, H. P., Crnkovic, I., & Eriksson, P. J. (2008). Analyzing software evolvability. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, 327-330. <https://doi.org/10.1109/COMPSAC.2008.50>
- Cai, L., Liu, Z., Zhang, J., Tong, W., & Yang, G. (2010). Evaluating software maintainability using fuzzy entropy theory. In *2010 IEEE/ACIS 9th International Conference on Computer and Information Science*, 737-742. <https://doi.org/10.1109/ICIS.2010.35>
- Chen, C., Alfayez, R., Srisopha, K., Boehm, B., & Shi, L. (2017). Why is it important to measure maintainability and what are the best ways to do it?. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion*, 377-378. <https://doi.org/10.1109/ICSE-C.2017.75>
- Ciraci, S., & Van Den Broek, P. (2006). Evolvability as a Quality Attribute of Software Architectures. In *EVOL*, 29-31.
- Cook, S., He, J., & Harrison, R. (2001). Dynamic and static views of software evolution. In *Proceedings IEEE International Conference on Software Maintenance*, 592-601. <https://doi.org/10.1109/ICSM.2001.972776>
- Ganpati, A., Kalia, A., & Singh, H. (2012). A comparative study of maintainability index of open source software. *Int. J. Emerg. Technol. Adv. Eng*, 228-230.
- Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology*, 30-39. <https://doi.org/10.1109/QUATIC.2007.8>
- Hincheeranan, A., & Rivepiboon, W. (2012). A maintainability estimation model and tool. *International Journal of Computer and Communication Engineering*, 143-146.
- Jun, H. K., & Rana, M. E. (2021). Evaluating the Impact of Design Patterns on Software Maintainability: An Empirical Evaluation. In *2021 Third International Sustainability and Resilience Conference: Climate Change*, 539-548. <https://doi.org/10.1109/IEEECONF53624.2021.9668025>

- Kitchenham, B. A., Budgen, D., & Brereton, P. (2015). *Evidence-Based Software Engineering and Systematic Reviews*. CRC Press. <https://books.google.com.mx/books?id=bGfdCgAAQBAJ>
- Kumar, P., & Singh, S. K. (2017). A framework for assessing the evolvability characteristics along with sub-characteristics in AOSQ model using fuzzy logic tool. *In 2017 International Conference on Computing, Communication and Automation*, 339-344. <https://doi.org/10.1109/CCAA.2017.8229839>
- Kurmangali, A., Rana, M. E., & Ab Rahman, W. N. W. (2022). Impact of Abstract Factory and Decorator Design Patterns on Software Maintainability: Empirical Evaluation using CK Metrics. *In 2022 International Conference on Decision Aid Sciences and Applications*, 517-522. <https://doi.org/10.1109/DASA54658.2022.9765083>
- Najm, N. M. A. M. (2014). Measuring Maintainability Index of a Software Depending on Line of Code Only. *IOSR J. Comput. Eng.*, 64-69.
- Nair, T. G., Aravindh, S., & Selvarani, R. (2010). Design property metrics to maintainability estimation: a virtual method using functional relationship mapping. *ACM SIGSOFT Software Engineering Notes*, 1-6. <https://doi.org/10.1145/1874391.1874404>
- Ostberg, J. P., & Wagner, S. (2014). On automatically collectable metrics for software maintainability evaluation. *In 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, 32-37. <https://doi.org/10.1109/IWSM.Mensura.2014.19>
- Pereplechikov, M., & Ryan, C. (2011). A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. *IEEE Transactions on software engineering*, 449-465. <https://doi.org/10.1109/TSE.2010.61>
- Ping, L. (2010). A quantitative approach to software maintainability prediction. *In 2010 International Forum on Information Technology and Applications*, 105-108. <https://doi.org/10.1109/IFITA.2010.294>
- Rochimah, S., Nuswantara, P. G., & Akbar, R. J. (2018). Analyzing the effect of design patterns on software maintainability: A case study. *In 2018 Electrical Power, Electronics, Communications, Controls and Informatics Seminar (EECCIS)*, 326-331. <https://doi.org/10.1109/EECCIS.2018.8692876>
- Rowe, D., Leaney, J., & Lowe, D. (1998). Defining systems evolvability-a taxonomy of change. *Change*, 541-545.
- Saifan, A. A., & Al-Rabadi, A. (2017). Evaluating maintainability of android applications. *In 2017 8th International Conference on Information Technology*, 518-523. <https://doi.org/10.1109/ICITECH.2017.8080052>
- Sandborn, P. A., Herald, T. E., Houston, J., & Singh, P. (2003). Optimum technology insertion into systems based on the assessment of viability. *IEEE Transactions on*

components and packaging technologies, 734-738.  
<https://doi.org/10.1109/TCAPT.2003.820984>

Sjøberg, D. I., Anda, B., & Mockus, A. (2012). Questioning software maintenance metrics: a comparative case study. *In Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, 107-110.  
<https://doi.org/10.1145/2372251.2372269>

Sunday, D. A. (1989). Software maintainability-a new 'ility'. *In Proceedings., Annual Reliability and Maintainability Symposium*, 50-51.  
<https://doi.org/10.1109/ARMS.1989.49572>

Wagey, B. C., Hendradjaya, B., & Mardiyanto, M. S. (2015). A proposal of software maintainability model using code smell measurement. *In 2015 International Conference on Data and Software Engineering*, 25-30.  
<https://doi.org/10.1109/ICODSE.2015.7436966>

Zhe, M., & Kerong, B. (2010). Research on maintainability evaluation of service-oriented software. *In 2010 3rd International Conference on Computer Science and Information Technology*, 510-512. <https://doi.org/10.1109/ICCSIT.2010.5563562>