¿Cómo documentar?

La documentación en el desarrollo de software es un proceso indispensable que asegura que todos los aspectos del proyecto están claramente comunicados y registrados, para que los participantes puedan consultar y transmitir la información generada durante la ejecución. Para lograr este fin vamos a revisar los siguientes aspectos a considerar.

Aspectos Importantes a Considerar

- **Integración en el proceso de desarrollo:** Hacer de la documentación una parte integral del ciclo de vida del desarrollo de software es fundamental para su efectividad (Boehm, 1988).
- **Revisiones y feedback:** Involucrar a otros miembros del equipo en la revisión de la documentación asegura su calidad y relevancia (Carroll, 1998).
- **Accesibilidad:** Asegurarse de que la documentación esté fácilmente accesible para todos los miembros del equipo y usuarios finales es crucial para su uso efectivo (Boehm, 1988).
- **Claridad y concisión:** La documentación debe ser fácil de entender y directa, evitando tecnicismos innecesarios que puedan confundir a los lectores (Carroll, 1998).
- **Estructura:** Organizar la información de manera lógica y accesible es fundamental para facilitar la búsqueda y comprensión de la misma (Fielding, 2000).
- **Formato:** Utilizar un formato claro y profesional, como un documento en Word, Google Docs o PDF o cualquier otro formato que se adapte a las necesidades, haciendo uso de gráficos, diagramas y otros elementos visuales puede mejorar significativamente la comprensión y retención de la información (Fielding, 2000).
- **Actualización regular:** Mantener la documentación al día con los cambios en el software es esencial para su relevancia y utilidad (Boehm, 1988).
- **Uso de ejemplos:** Incluir ejemplos prácticos ayuda a ilustrar conceptos complejos y facilita la comprensión (Carroll, 1998).

Para realizar la documentación de manera efectiva es fundamental seguir una estructura organizada que aborde los aspectos importantes de un proyecto, A continuación. se va a presentar una serie de elementos para generar la documentación.

1. **Título de la propuesta:** Un título claro, conciso y que refleje el objetivo del proyecto.

2. **Resumen ejecutivo o de negocio:** Un breve resumen que describa el propósito del proyecto, los beneficios esperados y los resultados claves, Este apartado debe de captar la atención del lector y proporcionar una visión general del contenido (Schmidt, 2019).
3. **Problemática:** Contexto sobre el problema que se busca resolver y la necesidad del software. Es importante establecer la relevancia del proyecto en el ámbito actual (Pressman, 2014).
4. **Objetivos del proyecto:** Definir los objetivos específicos, con los cuales se le debe de dar solución a la problemática, Estos deben ser medibles y alcanzables (SMART) (Doran, 1981).
5. **Alcance del proyecto:** Detallar qué se incluirá y qué no se incluirá en el proyecto. Esto ayuda a gestionar las expectativas de los interesados y a evitar desviaciones en el alcance (Kerzner, 2017).
6. **Requisitos funcionales:** Listar las funcionalidades que el software debe tener. Esto puede incluir descripciones de características específicas y cómo se espera que interactúen los usuarios con el sistema (Sommerville, 2011).
7. **Requisitos no funcionales:** Consideraciones sobre rendimiento, seguridad, usabilidad, entre otros. Estos requisitos son cruciales para la aceptación del software (ISO/IEC 25010, 2011).
8. **Definir patrones de comportamiento**: Listar los comportamientos esperados a la hora de generar un reembolso o reclamo de un seguro o póliza.
9. **Plan de desarrollo:** Un cronograma que incluya las fases del proyecto, hitos y plazos. Este plan debe ser realista y flexible para adaptarse a cambios (Schwalbe, 2015).
10. **Recursos necesarios:** Detallar los recursos humanos y técnicos para llevar a cabo el proyecto. Esto incluye la identificación de roles y responsabilidades (Meredith & Mantel, 2017).
11. **Riesgos y mitigaciones:** Identificar posibles riesgos y cómo se planea mitigarlos. La gestión de riesgos es esencial para el éxito del proyecto (Hillson, 2017).
12. **Conclusiones:** Resumir los puntos clave y la importancia del proyecto. Este apartado debe reafirmar la necesidad del desarrollo propuesto (Kendall & Rollins, 2003).
13. **Anexos:** Incluir cualquier información adicional que pueda ser relevante, como diagramas, gráficos o estudios de caso.

La elección de la herramienta adecuada para documentar un proyecto de software es un aspecto crítico que puede influir en la claridad, accesibilidad y mantenibilidad de la documentación. A continuación, se presentan los aspectos clave a considerar al seleccionar una herramienta de documentación, junto con referencias que respaldan la importancia de cada aspecto.

- **Tipo de documentación:** Es fundamental determinar el tipo de documentación que se necesita crear, ya sea especificaciones técnicas, manuales de usuario, documentación de API, entre otros. Algunas herramientas son más adecuadas para ciertos tipos de documentos (Baker, 2018).
- **Colaboración:** La colaboración en tiempo real es un factor importante, especialmente en equipos distribuidos. Herramientas como Google Docs o Confluence permiten que varios usuarios trabajen simultáneamente y realicen comentarios, lo que mejora la comunicación y la eficiencia (Schmidt, 2019).
- **Facilidad de uso:** La facilidad de uso de la herramienta es crucial para su adopción por parte del equipo. Una interfaz intuitiva puede reducir la curva de aprendizaje y facilitar la creación de documentación (Fowler, 2018).
- **Formato de salida:** Es importante verificar los formatos de salida que ofrece la herramienta. Algunas permiten exportar documentos en formatos como PDF, HTML o Markdown, lo que puede ser útil para compartir o publicar la documentación (Sommerville, 2011).
- **Integración con otras herramientas:** La capacidad de integración con otras herramientas utilizadas en el proyecto, como sistemas de control de versiones (Git) o plataformas de gestión de proyectos (Jira, Trello), es un aspecto a considerar para facilitar el flujo de trabajo (Pressman, 2014).
- **Control de versiones:** Asegurarse de que la herramienta permite el control de versiones es esencial para mantener un historial de cambios en la documentación y facilitar la colaboración (Kerzner, 2017).
- **Accesibilidad:** La accesibilidad desde diferentes dispositivos y plataformas es importante, especialmente si el equipo trabaja de forma remota o en diferentes ubicaciones (Schwalbe, 2015).
- **Soporte y Comunidad:** Investigar si la herramienta cuenta con un buen soporte técnico y una comunidad activa puede ser útil para resolver problemas y obtener consejos sobre el uso de la herramienta (Hillson, 2017).

Te en cuenta que el listado anterior son aspectos tener en cuenta y la selección de la herramienta se debe de acoplar a la naturaleza de cada proyecto, sin embargo, se recomienda el uso de herramientas enfocadas a la gestión de proyecto, por lo general suelen ser más robustas y más integrales abarcando muchos de los aspectos mencionados anteriormente, pero también existen otras alternativas, para ello se deja un listado de herramientas comunes para documentación de proyectos de software

- **Editores Markdown:** Herramientas como Typora o Dillinger permiten crear documentación en formato Markdown, que es fácil de leer y escribir (Baker, 2018).
- **Wiki**: plataformas como Confluence o MediaWiki son ideales para crear documentación colaborativa y accesible (Fowler, 2018).
- **Gestores de Proyectos:** Herramientas como Jira o Trello pueden incluir secciones para documentación, permitiendo mantener todo en un solo lugar (Pressman, 2014).

1.1.1. Definición básica de la necesidad.

Para establecer los patrones de comportamiento, primero se debe de poner en consideración que es un reembolso o reclamo y cómo los usuarios pueden acceder a esos servicios, esto depende en gran medida de las legislaciones de cada país, por eso se recomienda que tenga estas cuatro procesos que no dependen de dichas legislaciones y se consideran propias de la generación del reembolso, sean considerados como procesos básicos.

- **Presentación de la solicitud:** El asegurado debe presentar una solicitud de reembolso a la compañía de seguros. Esto generalmente implica completar un formulario específico y proporcionar documentación que respalde la reclamación, como facturas y recibos de los gastos incurridos (Europa, n.d.).
- **Evaluación de la solicitud:** La compañía de seguros revisa la solicitud y la documentación presentada. Este proceso puede incluir la verificación de que los gastos son elegibles según los términos de la póliza.
- **Aprobación o denegación:** Una vez evaluada la solicitud, la compañía de seguros emite una decisión. Si se aprueba, se procederá a realizar el reembolso. Si se deniega,

se debe proporcionar una explicación al asegurado sobre las razones de la denegación (Federal Register, 2024).

- **Emisión del reembolso:** Si la solicitud es aprobada, el reembolso se emite al asegurado, generalmente a través de un cheque o transferencia bancaria.

A partir de esto se puede definir otra serie de procesos que dependen de cada necesidad, para la definición de estos se recomienda tener en cuenta las siguientes pautas.

- Los procesos deben ser comportamientos que se repiten en el ciclo de vida de un reembolso y que respondan a una necesidad o una oportunidad de mejora.
- Se puede definir un proceso en base a una funcionalidad que se requiera en el proyecto, pero se debe considerar si esa funcionalidad afecta a todo el ciclo de vida, entre menos partes del proceso de reembolso afecte es más probable que sea un proceso.
- Cada proceso se debe de documentar, y este debe ser sometido a votación, por parte del equipo.

A partir de estos procesos se empiezan a definir una serie de comportamientos o patrones, que corresponde a lo mínimo que se necesita para que el proceso finalice, para la definición de esos patrones se debe de tener en cuenta.

- Un patrón en un comportamiento recurrente y necesario para el proceso, ejemplo: para el proceso ***presentación de la Solicitud*** se pueden definir el siguiente patrón, el usuario debe de llenar un formulario con la información básica, ese formulario y todo el esfuerzo que conlleva esa tarea se considera un patrón y este comportamiento se repite para todas las solicitudes, a este tipo de patrón se le considera patrón base.
- Un patrón puede variar dependiendo de unas condiciones, ejemplo, se necesita presentar una serie de preguntas al usuario que tiene como finalidad establecer lo sucedido, pero estas preguntas dependen del tipo de reembolso que se va a solicitar, a pesar de que el comportamiento es presentar esa serie de preguntas, varía qué preguntas, a este patrón lo vamos a considerar patrón cambiante.

Estos patrones se convierten posteriormente en el insumo principal para definir las tareas para la documentación de estos patrones se recomienda el uso de la siguiente estructura.

- **Proceso:** Nombre del procesos asociado, no puede existir un patrón sin procesos.
- **Nombre del patrón:** Asigna un nombre claro y descriptivo que refleje la función del patrón. El nombre debe ser intuitivo para facilitar su identificación y uso.

- **Intención:** Describe brevemente el propósito del patrón. ¿Qué problema resuelve? ¿Cuál es su objetivo principal? Esto ayuda a los desarrolladores a entender cuándo y por qué utilizar el patrón.
- **Contexto:** Define el contexto en el que se aplica el patrón. Esto incluye las condiciones y situaciones específicas en las que el patrón es útil.
- Aprobado/Rechazado - Razón: Definición si el patrón fue evaluado y la razón del rechazo.
- **Tipo:** Se definen dos tipos, patrón base y cambiante, esto se traduce en la complejidad del patrón que sirve para futuras consideraciones.

Una vez definido los procesos se continúa con la documentación técnica, la documentación técnica base de un proyecto de software enfocado en los reembolsos o reclamos es esencial para garantizar la comprensión, mantenimiento y escalabilidad del sistema. A continuación, se presentan los componentes clave de esta documentación, junto con una breve descripción de cada uno:

- Requisitos del Software: Este documento detalla las necesidades y expectativas del cliente y los usuarios finales. Incluye requisitos funcionales (qué debe hacer el sistema) y no funcionales (rendimiento, seguridad, etc.). Es fundamental para guiar el desarrollo y asegurar que el producto final cumpla con las expectativas (Sommerville, 2011).
- Diseño del Sistema: Describe la arquitectura del software, incluyendo diagramas de flujo y descripciones de componentes. Este documento proporciona una visión general de cómo se estructurará el sistema y cómo interactúan sus partes (Booch, 2007).
- Plan de Pruebas: Este documento detalla la estrategia de pruebas que se utilizará para verificar que el software cumple con los requisitos especificados. Incluye tipos de pruebas (unitarias, de integración, de sistema) y criterios de aceptación (Beck, 2002).
- Manual de Usuario: Proporciona instrucciones sobre cómo utilizar el software. Incluye guías paso a paso, ejemplos y soluciones a problemas comunes. Es crucial para facilitar la adopción del software por parte de los usuarios finales (Nielsen, 1993).

- Guía de Instalación: Proporciona instrucciones sobre cómo instalar y configurar el software en diferentes entornos. Es esencial para asegurar que los usuarios puedan implementar el sistema sin problemas (Perry & Wolf, 1992).

Definir limitantes y restricciones en un proyecto de software es un proceso crítico que impacta en la planificación, diseño y ejecución del proyecto. A continuación, se presentan las definiciones y enfoques para identificar y documentar estas limitantes y restricciones, junto con referencias académicas pertinentes.

- Limitantes: Se refieren a las condiciones o factores que pueden restringir el alcance del proyecto. Estas pueden incluir recursos limitados, como tiempo, presupuesto y personal, así como limitaciones tecnológicas que pueden afectar la implementación del software (Schwalbe, 2015).
- Restricciones: Son condiciones específicas que deben cumplirse durante el desarrollo del proyecto. Estas pueden incluir requisitos legales, normativos o de calidad que el software debe satisfacer. Las restricciones son más rígidas que las limitantes y a menudo se derivan de políticas organizacionales o estándares de la industria (Kerzner, 2017).

Proceso para Definir Limitantes y Restricciones

- Identificación: El primer paso es identificar las limitantes y restricciones potenciales. Esto se puede lograr a través de sesiones de lluvia de ideas con el equipo del proyecto, entrevistas con interesados y análisis de documentos existentes (PMI, 2017).
- Clasificación: Una vez identificadas, es útil clasificar las limitantes y restricciones en categorías, como técnicas, organizacionales, legales y de recursos. Esta clasificación ayuda a entender mejor el impacto de cada factor en el proyecto (Baker, 2010).
- Documentación: Es fundamental documentar las limitaciones y restricciones de manera clara y concisa. Esto incluye describir cada limitante y restricción, su origen, y cómo afectará al proyecto. La documentación debe ser accesible para todos los miembros del equipo y los interesados (Schwalbe, 2015).
- Revisión y Actualización: Las limitantes y restricciones pueden cambiar a lo largo del ciclo de vida del proyecto. Por lo tanto, es importante revisar y actualizar regularmente esta información para reflejar cualquier cambio en el contexto del proyecto (Kerzner, 2017).

Definición de Estrategias de Mitigación: Una vez que se han analizado las limitantes y restricciones, se deben definir estrategias específicas para mitigarlas. Estas estrategias pueden incluir:

- Reducción: Implementar medidas para disminuir la probabilidad o el impacto de la limitante o restricción.
- Transferencia: Delegar la responsabilidad de la limitante a un tercero, como un proveedor o socio.
- Aceptación: Reconocer la limitante o restricción y planificar cómo manejar sus efectos si se materializa (PMI, 2017).
- Desarrollo de un Cronograma: Es esencial establecer un cronograma para la implementación de las estrategias de mitigación. Esto incluye definir plazos específicos y asignar responsabilidades a los miembros del equipo. Un cronograma bien estructurado ayuda a asegurar que las acciones se realicen de manera oportuna (Kerzner, 2017).
- Asignación de Recursos: Identificar y asignar los recursos necesarios para implementar las estrategias de mitigación es crucial. Esto puede incluir recursos humanos, financieros y tecnológicos. La planificación adecuada de recursos asegura que el equipo tenga lo necesario para abordar las limitantes y restricciones (Schwalbe, 2015).
- Monitoreo y Evaluación: Una vez implementadas las estrategias, es importante establecer un sistema de monitoreo y evaluación para medir la efectividad de las acciones tomadas. Esto permite realizar ajustes en el plan de acción según sea necesario y garantizar que se logren los objetivos del proyecto (Baker, 2010).

Consideraciones para la Definición de Tareas en Proyectos de Software

La definición de tareas en proyectos de software es un proceso fundamental que influye en la planificación, ejecución y éxito del proyecto. A continuación, se presentan aspectos clave que deben considerarse al definir estas tareas, respaldados por la literatura académica.

1. **Claridad y Especificidad :** La claridad en la definición de tareas es esencial para evitar ambigüedades que puedan llevar a malentendidos. Según Boehm (1981), una especificación precisa de los requisitos y tareas es crucial para el éxito del desarrollo

2. **Descomposición de Tareas:** La descomposición de tareas grandes en subtareas más pequeñas y manejables facilita la estimación del tiempo y el esfuerzo requerido. Esta técnica, conocida como "Work Breakdown Structure" (WBS), es ampliamente utilizada en la gestión de proyectos para mejorar la planificación y el control (Kerzner, 2017).
3. **Prioridad:** Establecer la prioridad de cada tarea es fundamental. El análisis de MoSCoW (Must have, Should have, Could have, Won't have) es una técnica que ayuda a clasificar las tareas según su importancia y urgencia, permitiendo una mejor gestión del tiempo y los recursos (Cohn, 2005).
4. **Dependencias:** Identificar las dependencias entre tareas es crucial para la planificación. Las dependencias pueden afectar el cronograma del proyecto, y su gestión adecuada es esencial para evitar retrasos (Schwalbe, 2015).
5. **Estimación de Tiempo y Recursos:** La estimación precisa del tiempo y los recursos necesarios para completar cada tarea es vital. Según McConnell (2006), la experiencia del equipo y la complejidad de la tarea son factores determinantes en la estimación de esfuerzos.
6. **Asignación de Responsabilidades:** La asignación de tareas a miembros específicos del equipo debe basarse en sus habilidades y experiencia. Esto no solo mejora la eficiencia, sino que también aumenta la motivación y el compromiso del equipo (Belbin, 2010).
7. **Revisión y Ajuste:** Establecer un proceso para revisar y ajustar las tareas a medida que avanza el proyecto es fundamental. La flexibilidad en la gestión de proyectos permite adaptarse a cambios en los requisitos y circunstancias (Agile Alliance, 2021).
8. **Documentación:** La documentación adecuada de cada tarea, que incluya descripciones y criterios de aceptación, es esencial para asegurar que todos los miembros del equipo comprendan las expectativas (Pohl, 2010).
9. **Comunicación:** Fomentar una comunicación abierta entre los miembros del equipo es vital para resolver dudas y asegurar que todos estén alineados con los objetivos del proyecto (Schmidt et al., 2016).
10. **Uso de Herramientas de Gestión de Proyectos:** La utilización de herramientas de gestión de proyectos, como Jira o Trello, puede facilitar la organización y visualización de las tareas, mejorando la colaboración y el seguimiento del progreso (Duncan, 2013).

Definición de Tiempos en Proyectos de Software

La estimación de tiempos en proyectos de software es un proceso crítico que permite a los equipos de desarrollo planificar adecuadamente sus actividades, asignar recursos y establecer plazos realistas. Existen diversas metodologías y estrategias que se pueden utilizar para realizar estas estimaciones, cada una con sus propias ventajas y desventajas.

Metodologías para la Estimación de Tiempos

1. Análisis de Datos Históricos: Esta técnica implica revisar proyectos anteriores para identificar patrones y promedios en los tiempos de desarrollo. Al analizar datos históricos, los equipos pueden hacer estimaciones más precisas basadas en experiencias (door3, nd).
2. Método de Puntos de Función: Este método mide el tamaño funcional del software y se basa en la complejidad de las funciones que se van a desarrollar. Los puntos de función permiten estimar el esfuerzo necesario para completar el proyecto, lo que facilita la planificación de tiempos y recursos (itequia, nd).
3. Estimación Ágil: Las metodologías ágiles, como Scrum, utilizan técnicas como el Planning Poker, donde los miembros del equipo estiman el esfuerzo requerido para completar tareas específicas. Este enfoque fomenta la colaboración y permite ajustar las estimaciones en función de la retroalimentación continua (LinkedIn, nd).
4. Descomposición del Trabajo: Esta estrategia implica dividir el proyecto en tareas más pequeñas y manejables. Al estimar el tiempo para cada tarea individual, se puede obtener una estimación más precisa del tiempo total del proyecto.
5. Técnica Delphi: Consiste en consultar a un grupo de expertos para obtener sus estimaciones sobre el tiempo requerido para completar un proyecto. Las estimaciones se discuten y refinan en varias rondas hasta llegar a un consenso.

Referencias bibliográficas

Anderson, D. J. (2010). Kanban: Successful evolutionary change for your technology business. Blue Hole Press.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., … & Thomas, D. (2001). Manifesto for Agile Software Development. Agile Alliance.

Hüttermann, M. (2012). DevOps for developers. Apress.

Beck, K. (1999). Extreme Programming Explained: Embrace Change. Addison-Wesley Professional.

Royce, W. W. (1970). Managing the development of large software systems. Proceedings of IEEE WESCON.

Doran, G. T. (1981). There's a S.M.A.R.T. way to write management's goals and objectives. Management Review, 70(11), 35-36.

Fowler, M. (2018). Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Hillson, D. (2017). Practical Project Risk Management: The ATOM Methodology. Berrett-Koehler Publishers.

ISO/IEC 25010. (2011). Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.

Kendall, G. I., & Rollins, S. C. (2003). Advanced Project Portfolio Management and the PMO: Multiplying ROI at Warp Speed. J. Ross Publishing.

Kerzner, H. (2017). Project Management: A Systems Approach to Planning, Scheduling, and Controlling. Wiley.

Meredith, J. R., & Mantel, S. J. (2017). Project Management: A Managerial Approach. Wiley.

Pressman, R. S. (2014). Software Engineering: A Practitioner's Approach. McGraw-Hill.

Schmidt, C. (2019). The Art of Project Management. O'Reilly Media.

Sommerville, I. (2011). Software Engineering. Addison-Wesley.

Schwalbe, K. (2015). Information Technology Project Management. Cengage Learning.

Baker, J. (2018). Effective Documentation for Software Projects. O'Reilly Media.

Fowler, M. (2018). Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Hillson, D. (2017). Practical Project Risk Management: The ATOM Methodology. Berrett-Koehler Publishers.

Kendall, G. I., & Rollins, S. C. (2003). Advanced Project Portfolio Management and the PMO: Multiplying ROI at Warp Speed. J. Ross Publishing.

Kerzner, H. (2017). Project Management: A Systems Approach to Planning, Scheduling, and Controlling. Wiley.

Pressman, R. S. (2014). Software Engineering: A Practitioner's Approach. McGraw-Hill.

Schmidt, C. (2019). The Art of Project Management. O'Reilly Media.

Schwalbe, K. (2015). Information Technology Project Management. Cengage Learning.

Sommerville, I. (2011). Software Engineering. Addison-Wesley.

Gruber, J. (2004). Markdown. Retrieved from https://daringfireball.net/projects/markdown/

Beck, K. (2002). Test-Driven Development: By Example. Addison-Wesley.

Booch, G. (2007). Object-Oriented Analysis and Design with Applications. Addison-Wesley.

Fowler, M. (2004). The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley.

McConnell, S. (2004). Code Complete: A Practical Handbook of Software Construction. Microsoft Press.

Nielsen, J. (1993). Usability Engineering. Morgan Kaufmann.

Perry, D. E., & Wolf, A. L. (1992). Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17(4), 40-52.

Sommerville, I. (2011). Software Engineering. Addison-Wesley.

Baker, B. (2010). Project Management: A Systems Approach to Planning, Scheduling, and Controlling. Wiley.

Kerzner, H. (2017). Project Management: A Systems Approach to Planning, Scheduling, and Controlling. Wiley.

PMI (Project Management Institute). (2017). A Guide to the Project Management Body of Knowledge (PMBOK® Guide). Project Management Institute.

Schwalbe, K. (2015). Information Technology Project Management. Cengage Learning.

Baker, B. (2010). Project Management: A Systems Approach to Planning, Scheduling, and Controlling. Wiley.

Helms, M. M., & Nixon, J. (2010). Exploring SWOT Analysis – Where Are We Now?. Journal of Strategy and Management, 3(3), 215-251.

Kerzner, H. (2017). Project Management: A Systems Approach to Planning, Scheduling, and Controlling. Wiley.

PMI (Project Management Institute). (2017). A Guide to the Project Management Body of Knowledge (PMBOK® Guide). Project Management Institute.

Schwalbe, K. (2015). Information Technology Project Management. Cengage Learning.

Agile Alliance. (2021). Agile 101. Recuperado de https://www.agilealliance.org/agile101/

Belbin, R. M. (2010). Team Roles at Work. Routledge.

Boehm, B. W. (1981). Software Engineering Economics. Prentice Hall.

Cohn, M. (2005). User Stories Applied: For Agile Software Development. Addison-Wesley.

Duncan, W. R. (2013). Project Risk Management. Wiley.

Kerzner, H. (2017). Project Management: A Systems Approach to Planning, Scheduling, and Controlling. Wiley.

McConnell, S. (2006). Software Estimation: Demystifying the Black Art. Microsoft Press.

Pohl, K. (2010). Requirements Engineering: Fundamentals, Principles, and Techniques. Springer.

Schmidt, K., et al. (2016). Communication in Software Development Teams: A Systematic Literature Review. Journal of Systems and Software, 117, 1-15.

Schwalbe, K. (2015). Information Technology Project Management. Cengage Learning.

DOOR3. (n.d.). Estimación del tiempo de desarrollo de software. Recuperado de https://www.door3.com/es/blog/software-development-time-estimation

Itequia. (n.d.). Estimación de Tiempo en Desarrollo de Software: Estrategias Eficientes. Recuperado de https://itequia.com/es/estimacion-de-tiempo-en-desarrollo-de-software-estrategias-eficientes/

LinkedIn. (n.d.). Metodología Ágil para estimar el tiempo de Proyectos TI. Recuperado de https://es.linkedin.com/pulse/estimando-el-tiempo-en-proyectos-de-tecnolog%C3%ADa-la-con-ojeda-montoya