REVISTA DIGITAL

TIA

Tecnología, Investigación y Academia -Red Avanzada – RITA

Publicación Facultad de Ingeniería y Red de Investigaciones de Tecnología

Buenas prácticas en la construcción de software

Best practices in software development

Camilo Alberto Prieto[1] Diego Alejandro Madrid[2]

Citar este documento:

Camilo Alberto Prieto, Diego Alejandro Madrid. Buenas prácticas en la construcción de Software. Revista Tecnol.Investig.Academia TIA, ISSN: 2344-8288, 10 (2), pp. 149-167. Bogotá-Colombia.

[1] Ingeniero de Sistemas, Universidad Distrital Francisco José de Caldas, Alvic, camilo.prieto1989@gmail.com, Colombia, ORCID: 0000-0003-1976-7802.

[2] Ingeniero de Sistemas, Universidad Distrital Francisco José de Caldas, Infotrack, diegomadrid26@gmail.com, Colombia., ORCID: 0000-0002-8832-0214

Vol. 10 No. 2
Artículo corto
Fecha de aceptación: junio 2022
ISSN: 2344 8288
Fecha de recepción: abril-2022.

Resumen

La cuarta revolución industrial ha llevado a una constante evolución tecnológica encaminando a las organizaciones a buscar una mejora continua, adaptando su negocio al mercado moderno y a la competitividad por medio de la innovación. Para esto las organizaciones tienen el desafío de crear sistemas de información, cada vez más grandes y complejos, en corto tiempo y sin sobrepasar los costos. Al contar con diferentes tecnologías y arquitecturas, las organizaciones tienen dificultades para madurar y estandarizar sus procesos de desarrollo de software. El uso de buenas prácticas en la construcción del software, son aplicables a cualquier tecnología y proporcionan un valor agregado al producto, generando una mayor acogida y apreciación en la industria.

En este artículo encontrará una guía de buenas prácticas que se deben tener en cuenta para construir aplicaciones basándose en estándares del mercado.

***Palabras clave*:** Arquitectura, Estándar, Patrón, Software.

Abstract

The fourth industrial revolution has led to a constant technological evolution leading organization to seek continuous improvement, adapting their business to the modern market and competitiveness through innovation, for this, organizations have the challenge of creating information systems, increasingly large and complex, in a short time and without exceeding costs. By having different technologies and architectures, organizations find it difficult to mature and standardize their software development processes. The use of good practices in the construction of the software are applicable to any technology and provide added value to the product, generating greater acceptance and appreciation in the industry.

In this article you will find a guide of good practices that you should consider to build applications according to market standards.

***Key Word**: Architecture, Standard, Pattern, Software.*

1. INTRODUCCIÓN

En la actualidad las organizaciones tienen la necesidad de apoyarse en los proyectos de software para mejorar y optimizar sus procesos, lo cual requiere de una inversión de tiempo y esfuerzo que es proporcional a los costos, con el fin de crear soluciones de manera ágil con equipos que den soporte a todo el ciclo de vida del Software.

Debido a este auge informático las organizaciones buscan tener tecnologías y arquitecturas modernas que permitan generar una mayor estabilidad y versatilidad a sus proyectos.

Un factor de fracaso en la ejecución de los proyectos es que las organizaciones no cuentan con un proceso de Diseño y Desarrollo de Software con una Arquitectura que soporte los atributos de calidad según las necesidades de negocio y que defina una base con estándares y buenas prácticas para el equipo de desarrollo.

También se evidencia que, en los equipos de desarrollo, no se cuenta con un proceso maduro en la transferencia del conocimiento a los integrantes del equipo a causa de falencias en las tecnologías, arquitecturas y estándares de las aplicaciones usadas. Esto se deriva en la práctica de antipatrones al iniciar las actividades de desarrollo, tales como:

- (copy and paste programming): Programar copiando y modificando código existente en lugar de crear soluciones genéricas.
- (reinventing the wheel): Enfrentarse a las situaciones buscando soluciones desde cero, sin tener en cuenta soluciones ya desarrolladas para afrontar los mismos problemas.

2. CONTENIDO

2.1 Estándares y buenas prácticas

La práctica de estos antipatrones deriva en tener aplicaciones con código de baja calidad, difíciles de mantener y poco escalables a nivel funcional afectando los costos y tiempos de los proyectos.

Artículo de reflexión. ISSN: 2344-8288 Vol. 10 No. 2. 2022 Bogotá-Colombia

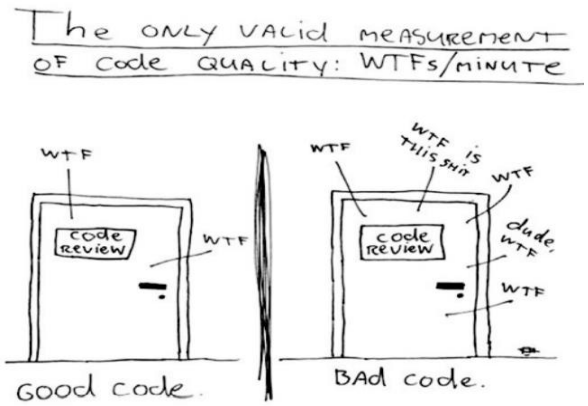
Por lo anterior, es importante aplicar estándares y buenas prácticas al momento de diseñar y escribir el código de las aplicaciones, que permitan tener mantenibilidad y escalabilidad del producto y, así mismo, promover estos estándares como una cultura organizacional.

2.1.2 Clean Code

Gran parte del tiempo de un desarrollador es dedicado a leer código. Esta tarea puede variar en su complejidad según la calidad de su escritura. Precisamente, este escenario es el que la filosofía Clean Code busca combatir.

El término *Clean Code* se le atribuye al ingeniero de software Robert Cecil Martin, siendo una filosofía de desarrollo de software que parte de la aplicación de técnicas sencillas para facilitar la tarea de lectura y escritura de código haciéndolo intuitivo para el lector. [7]
La implementación del principio "*KISS" (Keep it simple stupid)* busca manejar una simplicidad en la forma en que se estructura el código, para reducir al máximo la complejidad de lectura. [11]. Un buen código es entendible y requiere de poco esfuerzo y tiempo para que un desarrollador con experiencia pueda comprender su estructura y objetivo. Para esto, se parte de una serie de buenas prácticas:

Figura 1 Buen código vs mal código.



Fuente: R. Martin, Clean code: A handbook of agile software craftsmanship [8].

a) Nombre con sentido

La correcta definición de los nombres de variables, funciones, parámetros, clases y métodos es esencial para el entendimiento del código. Para esto, es necesario que cumplan con varios aspectos: [7]

Un nombre debe ser preciso y dar un entendimiento central de su funcionalidad. Éste puede ser grande, si así se requiere, siempre y cuando represente lo necesario. Sin embargo, es aconsejable mantener nombres cortos y entendibles. Se debe evitar que los nombres incluyan abreviaciones, prefijos, secuencias de números o palabras redundantes *(the-, a-, -object, -info, -data*) [8].

De igual manera, en cuanto a la estructura del nombramiento, no se deben utilizar palabras reservadas del lenguaje de programación, ni hacer uso de juegos de palabras. Es posible agregar un contexto a las variables agrupándolas en clases. Un correcto nombramiento evitará la creación de una documentación redundante del código.

b) Funciones

En su correcta definición, las funciones deben de ser reducidas buscando no sobrepasar las 20 líneas de código, evitando un anidamiento excesivo. Las funciones deben seguir el "*principio de responsabilidad única",* siendo creadas para seguir un único propósito correctamente. [9]
Al tener varias condiciones *(if/else - whil*e) que ocupan más de una línea de código, es adecuado implementar una llamada a un método, haciendo que la función quede más corta y fácil de leer, teniendo en cuenta que este método debe tener un nombre ampliamente descriptivo.

Teniendo en cuenta que la lectura del código se realiza de arriba hacia abajo, se debe seguir la regla "*the Stepdown rule*" [8]. Ésta busca que cada función esté seguida de otra que corresponda con el siguiente nivel de abstracción, buscando facilidad en la lectura al avanzar a través del archivo de código. Al estar en la parte superior, una función principal debe estar seguida de subniveles o subfunciones dependientes de ésta.

Artículo de reflexión. ISSN: 2344-8288 Vol. 10 No. 2. 2022 Bogotá-Colombia

Se debe buscar reducir, en su mayor medida, los argumentos que reciben las funciones. Lo ideal es no enviar ninguno o, en caso de enviarlos, nunca más de tres. Esto debido a que, un número alto de argumentos, significa un mayor tiempo para interpretar lo que representan y un aumento de la complejidad en el testeo. Para esto es aconsejable hacer uso de objetos que agrupen los argumentos cuando el número de éstos es mayor a tres.[9]

Una práctica importante es evitar duplicar código, teniendo en cuenta el principio "*DRY*" (dont repeat yourself), si ya hemos construido un código que tiene una funcionalidad similar, no debe repetirse, para esto es aconsejable hacer una refactorización y generalización buscando hacer uso del mismo código sin necesidad de repetirlo.

c) Comentarios

El uso de los comentarios debe darse únicamente para casos donde no es posible expresarse por medio del mismo código desde su correcto nombramiento y estructura. [8]. Es necesario evitar comentarios redundantes que no aportan ninguna información de valor en espacios donde el código se explica por sí solo. Es un caso común encontrar código con comentarios sobre el versionamiento de los cambios que se realizan, siendo una mala práctica al ser información redundante, que ya podemos obtener mediante los sistemas de control de versiones que facilitan esta tarea.

Las líneas de código comentadas deben ser eliminadas, no es una buena práctica mantener estas líneas que no están siendo utilizadas, ya que dificultan la lectura e interpretación del código, además es posible recuperarlas una vez borradas, mediante el uso apropiado de los controladores de versiones.

El código debe ser creado partiendo de la importancia que tiene una fácil interpretación por parte del lector. Por tal motivo, es de vital importancia la construcción y uso de estas prácticas, algunos de los comentarios posibles a usar de forma positiva son:[7]

- Comentarios legales sobre derechos de autor.
- Comentarios informativos cuando no es clara una funcionalidad.
- Comentarios sobre advertencias.
- Comentarios sobre utilización de librerías o recursos de terceros que no sean modificables.

d) Formato

Los archivos de código deben tener un límite máximo de quinientas líneas, buscando mantener una mínima distancia vertical entre los elementos, y de igual forma el ancho de las líneas de código no deben exceder los ciento veinte caracteres. Esto con el fin de poder realizar una lectura más rápida del código sin necesidad de hacer uso de la barra de desplazamiento horizontal.

Es importante que las variables sean declaradas lo más cerca posible de su implementación. Sin embargo, las variables que se usarán de instancia (durante todo el archivo) se declaran en la parte superior, haciendo más fácil el entendimiento para el desarrollador. [8]

Los equipos de desarrollo deben tener una serie de reglas para el estilo de código en cuanto a su indentación y formatos. Lo anterior con el fin de mantener el mismo estilo a lo largo de todo el proyecto al momento de compartir el código.

Un ejemplo de lo anterior es la teoría de las ventanas rotas que, aplicada al código, nombra la importancia de una cultura en el equipo de desarrollo, en la que prevalezca la importancia de cuidar las buenas prácticas en la construcción de software. Si no se cuidan estas prácticas, el mal manejo del código puede ser replicado por miembros del equipo provocando que pierdan el interés por hacerlo mejor y dejen de construir el software según los principios deseados.[9] Para promover las buenas prácticas en todo el equipo de desarrollo, es necesario evitar malos diseños y códigos difíciles de mantener.

e) Tratamiento de errores

Es importante mantener un manejo y tratamiento adecuado de los errores, con el fin de buscar que nuestro código sea estable ante cualquier eventualidad. Cada una de las excepciones generadas en el código, deberá proporcionar una información Suficiente para entender el contexto y ubicación del error. [9]

Es considerada una buena práctica la separación lógica del manejo de errores del negocio. De esta forma, el código se mantendrá limpio, facilitando el entendimiento. Además, posibilita la creación de una estructura de código generalizada que permita determinar la acción a tomar según cada tipo de error. [8]

f) Clases y objetos

Es importante mantener un único propósito para cada clase, haciendo uso del "*principio de responsabilidad única",* que fomenta la buena práctica de mantener clases pequeñas y bien organizadas en vez de clases extensas que pueden llegar a ser más complejas.

Haciendo uso del principio "*abierto/cerrado*" es necesario tener en cuenta que las clases deben estar abiertas para la extensión y cerradas para su modificación, siendo mejor práctica, realizar nuevos ajustes en el código extendiendo o introduciendo nuevas clases para evitar hacer una modificación de las existentes. [9]

Mediante el principio de inversión de dependencias se tiene en cuenta que las clases deben depender de abstracciones y no de detalles concretos, para así facilitar la ejecución de test y proporcionar un código más limpio y generalizado. [11]. Una buena práctica es mostrar los datos por medio la abstracción [9] de forma tal que no quede expuesta su implementación, teniendo en cuenta los conceptos de:

- **Acoplamiento:** Es el grado de conocimiento de una clase sobre otra. Un acoplamiento fuerte hace referencia a que las clases relacionadas necesitan conocer

muchos detalles internos de otras. Lo ideal es mantener un acoplamiento bajo que permita al desarrollador entender una clase sin tener que leer otras y poder modificarla sin llegar a alterarlas. [9]

- **Cohesión:** Es la medida de independencia de una clase, es decir, cómo una clase está diseñada para llegar a interactuar con esta misma. [9]

2.2 Arquitectura

Lo primero para construir una aplicación con calidad y estándares es contar con una definición de arquitectura. Para construir software es importante basarse en el análisis y diseño de una solución que cubra las necesidades de negocio y definir la estructura que guiará el desarrollo de las aplicaciones.

Para esto, la arquitectura permite diseñar el sistema que cubrirá los requerimientos funcionales y no funcionales del software. Debido a que se trata de un proceso creativo, las actividades dentro del proceso dependen del tipo de sistema a desarrollar, los antecedentes y la experiencia del arquitecto del sistema, así como de los requerimientos específicos de éste. Por lo tanto, es útil pensar en el diseño arquitectónico como un conjunto de decisiones a tomar, en vez de una secuencia de actividades [1].

En el análisis de la arquitectura se deben tener en cuenta tener tres factores importantes:

1) **Arquitectura Empresarial:** Este enfoque tiene un detalle bajo y un nivel de genericidad alto del sistema que permite definir las relaciones entre los principales activos de una empresa, incluyendo Estrategia, Calidad Organización, Procesos, Información, Aplicaciones y Tecnología. La arquitectura empresarial es el mapa que proporciona un entendimiento común de la organización. Ésta se usa para alinear la estrategia y los requerimientos tácticos, permitiendo analizar los efectos que tendrán las decisiones de sus directivos, mitigando riesgos y planificando las acciones correctas a ejecutar en el mejor momento para así gestionar las transformaciones que se deseen realizar [2].

2) **Arquitectura Solución:** Este enfoque tiene un detalle y un nivel de genericidad medio del sistema que define el diseño y la comunicación de estructuras de alto nivel para permitir y guiar el diseño y desarrollo de soluciones integradas que satisfagan las necesidades actuales y futuras del negocio. Además de los componentes de tecnología, la arquitectura de solución abarca los cambios a servicios, procesos, organización y modelos operativos [3].

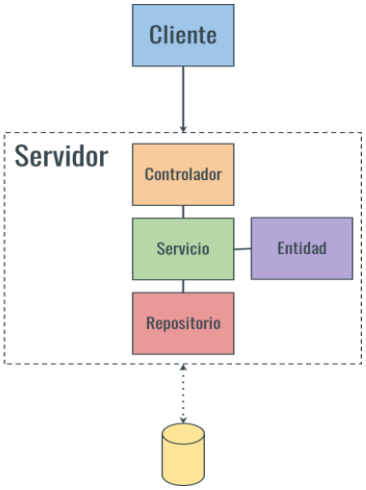
3) **Arquitectura Software:** Este enfoque tiene un detalle alto y un nivel de genericidad bajo del sistema que permite definir patrones de arquitectura, patrones de diseño, estilo arquitectónico, datos y tecnologías que se van a implementar en las soluciones.

2.3 Estilos Arquitectónicos

Es importante analizar e identificar qué estilo Arquitectónico es el más ideal a utilizar para la construcción de la aplicación teniendo en cuenta los atributos de calidad. Un estilo arquitectónico establece un marco de referencia que identifica y clasifica un conjunto de atributos y características que conforman las aplicaciones. [4]. Es importante mencionar que los estilos arquitectónicos no determinan la tecnología en la cual está construido el software, ni los detalles técnicos de cómo debe construirse, en su lugar, da ciertos lineamientos y características que debe cumplir un software para establecer que sigue un determinado estilo arquitectónico. Los estilos arquitectónicos más importantes son:

1) **Capas:** La arquitectura en capas es una de las más utilizadas, no solo por su simplicidad, sino porque es usada por defecto cuando no hay seguridad de la arquitectura a usar para determinada aplicación. La arquitectura en capas divide la aplicación en capas. Cada capa tiene un rol definido, como podría ser, una capa de presentación (UI), una capa de reglas de negocio (servicios) y una capa de acceso a datos (DAO). Sin embargo, este estilo arquitectónico no define cuántas capas debe tener la aplicación, por el contrario, se centra en la separación de la aplicación en capas (Aplica el principio de Separación de preocupaciones (SoC)).

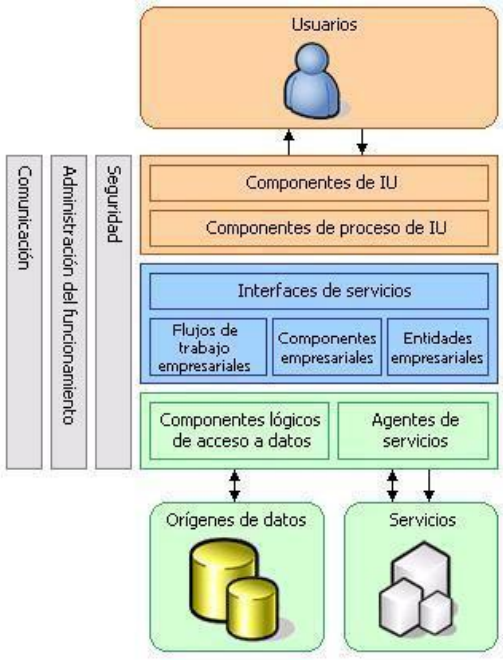
Figura 2 Arquitectura en capas.



Fuente: Elaboración Propia.

2) **Monolítico:** El estilo arquitectónico monolítico consiste en crear una aplicación autosuficiente, agrupando todo lo relacionado con el sistema dentro del mismo proyecto, sin contar con dependencias externas que integren su funcionalidad. En este sentido, sus componentes quedan acoplados y trabajan con los mismos recursos.

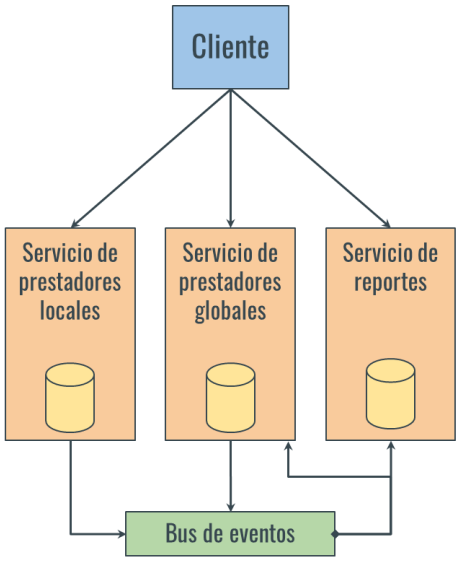
Figura 3 Arquitectura de un monolítico.



Fuente: Desarrollo de aplicaciones para ambientes distribuidos [5].

3) Microservicios: El estilo de Microservicios consiste en crear pequeños componentes de software con una sola responsabilidad sin importar la tecnología, lo que les permite desarrollarse de forma totalmente independiente del resto de componentes. Los microservicios han creado infraestructuras IT más adaptables y flexibles ya que, si se quiere modificar solamente un servicio, no es necesario alterar el resto de la infraestructura. Cada uno de los servicios se puede desplegar y modificar sin que ello afecte a otros servicios o aspectos funcionales de la aplicación.

Figura 4 Arquitectura Microservicios

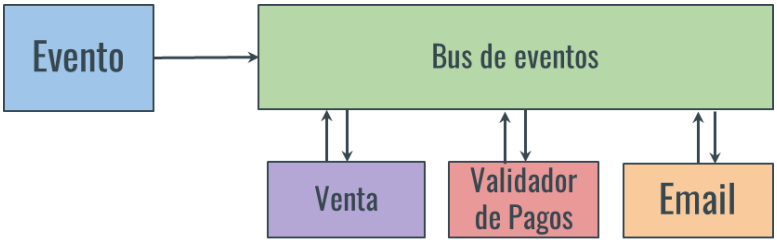


Fuente: Elaboración Propia.

2.4 Event-driven architecture: La arquitectura dirigida por eventos EDA (por sus siglas en inglés) es una arquitectura asíncrona y distribuida, pensada para crear aplicaciones altamente escalables.

En una arquitectura EDA, los componentes no se comunican de forma tradicional, en la cual se establece comunicación de forma síncrona, luego se obtiene una respuesta y se procede con el siguiente paso. Una arquitectura basada en eventos tiene un bajo nivel de acoplamiento porque los productores de eventos no saben cuáles consumidores de eventos están atentos a ellos, y el evento desconoce cuáles son sus consecuencias [11].

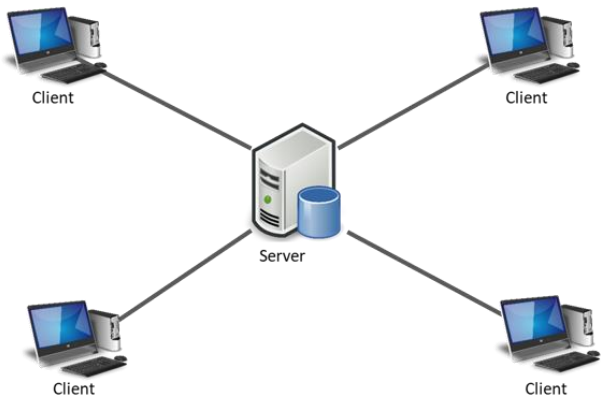
Figura 5 Arquitectura EDA



Fuente: Elaboración Propia.

2.5 Cliente Servidor: La arquitectura Cliente-Servidor es un modelo de aplicación distribuida, el cual está compuesto por el proveedor que es un servidor que brinda una serie de servicios o recursos los cuales son consumidos por el cliente.

Figura 6 Arquitectura cliente servidor



Fuente: Arquitectura cliente servidor [6]

2,6 Patrones de Diseño de software

Los patrones de diseño en la arquitectura de software, son soluciones a problemas de diseño, que han comprobado su efectividad resolviendo problemas similares en el pasado. Éstos tienen que ser reutilizables para resolver problemas parecidos en contextos diferentes [4].

Los patrones causaron un enorme impacto en el diseño de software orientado a objetos. Debido a que son soluciones probadas a problemas comunes, se convirtieron en un vocabulario para hablar sobre un diseño [1]. Por esto, es importante identificarlos y aplicarlos como una buena

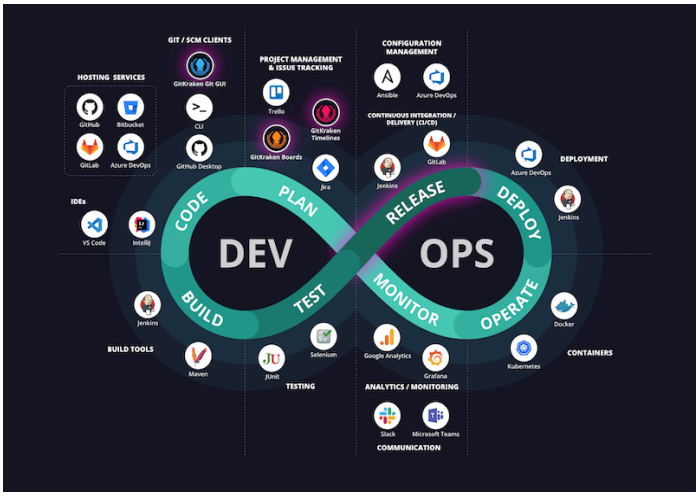
práctica y guía en la construcción de software. Los patrones de diseño tienen características en el desarrollo de software:

- **Plataforma común para desarrolladores:** Los patrones de diseño proporcionan una terminología estándar y son específicos para un escenario particular.
- **Mejores prácticas:** Aprender estos patrones ayuda a los desarrolladores a conocer el diseño de software de una manera fácil y rápida.
- Los patrones de diseño se dividen en tres tipos, los cuales se agrupan según el tipo de problema que buscan resolver, los tipos son:
- **Patrones creacionales:** Son patrones de diseño relacionados con la creación o construcción de objetos. Estos patrones intentan controlar la forma en que los objetos son creados, implementando mecanismos que eviten la creación directa de objetos.
- **Patrones estructurales:** Son patrones que tienen que ver con la forma en que las clases se relacionan con otras. Estos patrones ayudan a dar un mayor orden a las clases ayudando a crear componentes más flexibles y extensibles.
- **Patrones de comportamiento:** Son patrones que están relacionados con procedimientos y con la asignación de responsabilidad a los objetos. Los patrones de comportamiento engloban también patrones de comunicación entre ellos.

2.7 Herramientas

Antes de iniciar a construir software, es una buena práctica realizar un estudio de las tecnologías y herramientas que se van a utilizar para el ciclo de desarrollo de software. Una cultura que apoya esta idea es *DevOps*. En cada etapa, esta cultura se apoya en herramientas que permiten automatizar los procesos que soportan el desarrollo, tales como: *test* automatizados, integración continua, despliegue continuo, entre otros. El uso de herramientas beneficia los tiempos de desarrollo y la calidad de los entregables.

Figura 7 Herramientas DevOps,Fuente: Informe sobre herramientas DevOps 2020



Fuente:[12]

2.8 Metodología XP

La metodología XP "*Extreme programing*" (programación extrema), es conocida por ser una de las metodologías ágiles más exitosas en el desarrollo de software. Siendo habitual que se relacione con Scrum, buscan una implementación de forma eficiente y efectiva para la ejecución de proyectos. [13]

Con esta metodología se busca lograr construir productos con características muy ajustadas al cliente, siendo ampliamente flexible a contratiempos y cambios en el proyecto. La metodología XP se centra en mantener una continua comunicación con todos los involucrados en el proyecto, teniendo en cuenta la reutilización del código existente y la realimentación. [14]

5.1 Características metodología XP

- Plantea una comunicación constante entre el cliente y el equipo de desarrollo.
- Consigue una respuesta rápida a los cambios constantes.
- Se mantiene un cronograma de actividades flexibles.

Artículo de reflexión. ISSN: 2344-8288 Vol. 10 No. 2. 2022 Bogotá-Colombia

- Plantea que el software funcional prevalece sobre cualquier documentación.
- Los factores de éxito del proyecto son medidos según los requisitos del cliente y trabajo del equipo.

2.8 Modelo de la metodología XP

En la se tienen en cuenta ciertas variables en un proyecto: costo, tiempo, calidad y alcance. De estas variables se plantea que el alcance y la calidad serán definidos por el cliente y el costo, por el jefe del proyecto. El tiempo que durará el proyecto será definido libremente por el equipo de desarrollo, buscando mantener un equilibrio entre las variables involucradas en él. [13]

La metodología XP tiene un equipo con ciertos roles definidos:[15]

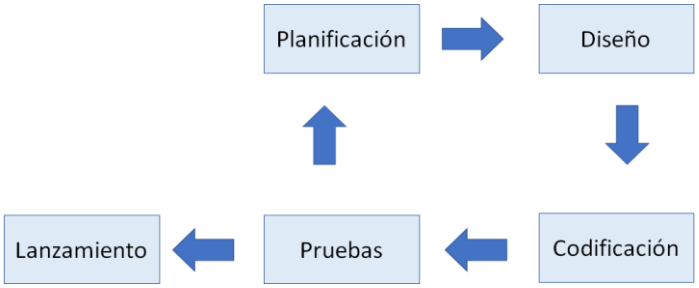
- **Cliente :** Los clientes son los responsables de conducir la gestión de objetivos de los proyectos marcando las necesidades y prioridades.
- **Desarrolladores:** Los desarrolladores se encargan de delimitar la duración de las tareas y gestionar los tiempos además de contribuir con la construcción y cumplimiento de los objetivos planteados por el cliente.
- **Testers:** El tester o encargado de pruebas tiene una comunicación continua con el cliente para alinear los resultados conseguidos frente a los requisitos estimados.
- **Tracker:** El tracker o encargado de seguimiento, tiene como objetivo que se tenga un control sobre las actividades que se realizan. Éste promueve la comunicación y relación constante con el cliente, definiendo hitos en la planeación y controlando la ejecución de tareas de los desarrolladores.
- **Coach:** El coach realiza la actividad de acompañamiento y orientación del equipo de trabajo y cliente, siendo guía para que todo salga de la manera correcta.
- **Manager :** El manager se encarga de coordinar la comunicación entre distintas partes, gestionando los recursos necesarios.

2.9 Ciclo de vida metodología XP

El ciclo de vida de la metodología XP incluye cinco fases: [15]

1) **Fase de planificación:** Parte del entendimiento de lo que el cliente necesita, partiendo de las historias de usuario, se priorizan y descomponen en versiones pequeñas. Cada iteración debe generar una nueva versión del software, siendo funcional y lista para probar y lanzar.
2) **Fase de diseño:** En esta fase se manejan versiones sencillas haciendo lo mínimo necesario para que funcione y se obtenga un prototipo.
3) **Fase de codificación:** En XP la programación es realizada en parejas, frente al mismo ordenador con posibilidad de intercambiar el mando. De esta manera, hay certeza de la calidad y universalidad del código, implementando el uso de buenas prácticas con el fin de que cualquier desarrollador pueda entender y trabajar el código. Así, se obtiene una programación planificada y correctamente estructurada.
4) **Fase de pruebas:** Es necesario realizar pruebas automáticas continuamente, siendo un factor clave al tratarse de proyectos a corto plazo. Teniendo en cuenta que el mismo cliente podrá hacer y sugerir nuevos tests.
5) **Fase de lanzamiento:** En esta fase se entrega el producto final al cliente, después de ser validado y probado en cada una de las historias de usuario planteadas, teniendo un software útil e incorporable al producto.

Figura 8 Fases del ciclo de vida XP



Fuente: Elaboración propia

La metodología XP se caracteriza por un ciclo de vida dinámico mediante ciclos de desarrollo cortos (iteraciones) buscando conseguir entregables funcionales. Cada una de las iteraciones comprende un ciclo completo de análisis, diseño, desarrollo y pruebas, pero utilizando un conjunto de reglas y prácticas de XP. Se suele emplear entre 10 y 15 iteraciones habitualmente. [13]

3 CONCLUSIONES

Una correcta cultura organizacional en el equipo de desarrollo promueve el uso de la integración de buenas prácticas, las cuales facilitan el entendimiento del grupo de trabajo en la codificación de proyectos, estandarizando y generalizando un lenguaje universal para el equipo.

La implementación de las prácticas sugeridas en este artículo contribuye a la reducción de tiempos y esfuerzos empleados en el momento de codificar proyectos de software, generando una reducción de costos para las organizaciones.

La adopción de estándares en la escritura de código como las reglas de Clean Code, permiten que las aplicaciones sean más simples de leer y entender, beneficiando la mantenibilidad de los productos.

La implementación de herramientas tecnológicas y marcos de trabajo en el proceso de desarrollo de software que permiten automatizar tareas optimizan los esfuerzos de los equipos de desarrollo.

La Arquitectura de un proyecto de software es indispensable para analizar, diseñar y definir cuál es el estilo arquitectónico que cubre las necesidades de negocio a nivel funcional y no funcional.

Referencias

[1] I. Sommerville, Ingeniería de software, 9 ed., Mexico D.F.: Sommerville, 2011.

[2] P. Robledo, «albatian innovation consulting,» 04 2017. [En línea]. Disponible: https://albatian.com/es/. [Último acceso: 23 05 2021].

[3] S. Foundation, «SFIA El marco global de habilidades y competencias para un mundo digital.» SFIA Foundation, 2003. [En línea]. Disponible: https://sfia-online.org/es/sfia-7/skills/solution-architecture. [Último acceso: 22 05 2021].

[4] O. Blancarte, Introducción a la arquitectura de Software, Ciudad de México: Oscar Javier Blancarte Iturral de, 2020, pp. 45-50.

[5] «Desarrollo de aplicaciones para ambientes distribuidos.» 03 2013. [En línea]. Disponible: https://laurmolina7821.wordpress.com/1-1-1-aplicaciones-monoliticas/. [Último acceso: 23 05 2021].

[6] O. Blancarte, «reactiveprogramming,» 2019. [En línea]. Disponible: https://reactiveprogramming. [Último acceso: 23 05 2021].

[7] "Clean Code: Código Limpio, ¿qué es? ¿cómo funciona?", *Blog HostGator México*, 2021. [Online]. code-codigo-limpio/. [Último acceso: 24 05 2021].

[8] R. Martin, Clean code: A handbook of agile software craftsmanship, 1.ª ed. Southampton, PA, Estados Unidos de América: Pearson, 2008.

[9] M. Dolores Monedero, "Guardianes del código", Paradigmadigit https://www.paradigmadigital.com/dev/guardianes-del-codigo/. [Último acceso: 25- May- 2021].

[10] R. C. Martin, "Summary of 'Clean code' by Robert C.Martin https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29.

[Último acceso: 25- May- 2021].

[11] I. Red Hat, «Red Hat,» 08 07 2020. [En línea].Disponible:https://www.redhat.com/es/resources/event-driven-architecture-hybrid-cloud-blueprint-detail. [Último acceso: 24 05 2021].

[12] «República Web,» 09 05 2020. [En línea]. Disponible: https://republicaweb.es/podcast/informe-sobre-herramientas-devops-2020/. [Último acceso: 25 05 2021].

[13] J. Grau, "La Metodología XP: la metodología de desarrollo de software más exitosa", Proagilist, 2021. [Online]. Disponible: https://proagilist.es/blog/agilidad-y-gestion-agil/agile-scrum/la-metodologia-xp/. [Último acceso: 25- May- 2021].

Publicación Facultad de Ingeniería y Red de Investigaciones de Tecnología Avanzada – RITA

REVISTA

TIA