Juan David Cerquera Salazar

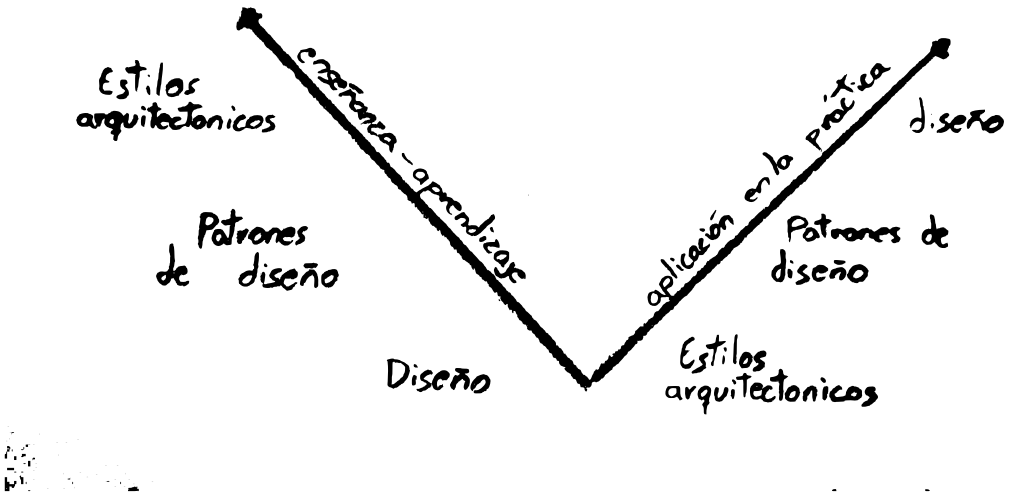
Soy Juan David Cerquera tengo 19 años, actualmente me encuentro cursando un tecnólogo de análisis y desarrollo software en el SENA y una ingeniería en la UNAD. Empecé a desarrollarme como programador desde finales de mis años escolares con proyectos básicos, lo que me ha permitido desarrollar gusto a la programación, aspiro a llegar ser un gran desarrollador FullStack.

Tengo alrededor de dos años de estudio en el ámbito de la programación y la tecnología, teniendo conocimientos en lenguajes de programación como java, c#, Python y JavaScript. También he tenido experiencia con base de datos MySQL, y con algunos framework como angular y spring boot.

Introducción a la Arquitectura de Software

En los primeros años de la construcción de software no existía el diseño del sistema como una etapa independiente la programación. Pues esta práctica se comienza a proponer, investigar y aplicar a principios de los 70.

Mientras que en el 90 se empiezan a ver la necesidad de desarrollar un nivel de abstracción superior al de diseño, aquí es donde surge la arquitectura de software.



“Las etapas de la fase Arquitectura del Sistema se enseñan-aprenden de forma opuesta a la forma en que se aplican en la práctica”.

1. Elección del estilo arquitectónico

2. Selección de los patrones de diseño

3. Diseño de componentes

Hay que tener en cuenta que jamás se contara con los requerimientos completos de un sistema, sino que para definir el diseño se tiene en cuenta los requerimientos que se tiene y los que llegaran.

A la arquitectura de software afecta a su entorno y su entorno la afecta a ella esto se denomina ABC (Architecture Business Cycle).

La arquitectura de software de un sistema es el resultado de combinar decisiones técnicas, sociales y del negocio. Pues los interesados del sistema se tarden o temprano requieran que tenga distintas características; Los requerimientos funcionales y no funcionales se deben tener en cuenta desde el principio ya que luego serán un gran problema.

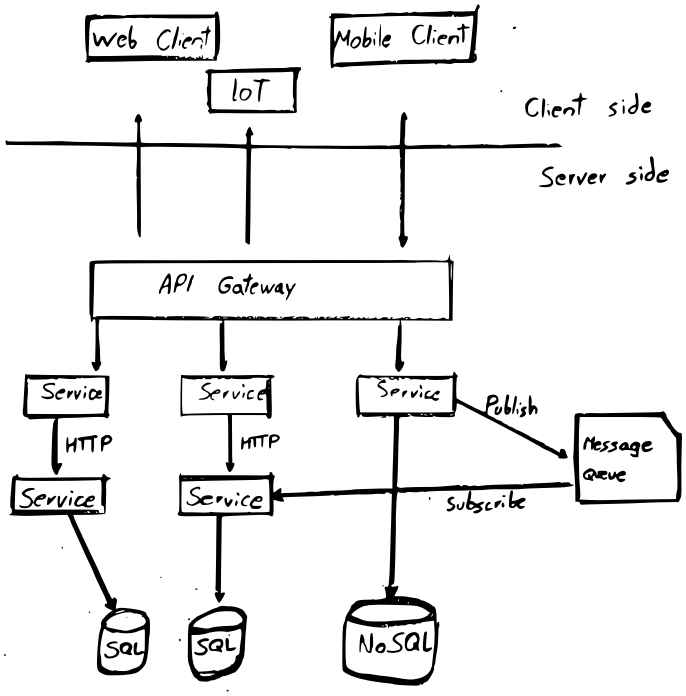
Reflexión:

Yo creo que la arquitectura de software es una herramienta fundamental a la hora de escribir software pues con esta se tendrá entendimiento de como podrá funcionar el sistema ya que este tiene un entendimiento total del sistema tanto técnico como sociales, como el entorno de negocio. Esto permite construir software de calidad que no solo es funcional si no que es adaptable y escalable con el tiempo.

Bibliografía:

(S/f). Researchgate.net. Recuperado el 23 de septiembre de 2024, de https://www.researchgate.net/publication/251932352_Introduccion_a_la_Arquitectura_de_Software

Arquitectura basada en microservicios para aplicaciones web



Tomando como referencia las arquitecturas que tienen grandes empresas como Netflix y Amazon basándose en la arquitectura de microservicios. Nos habla de que la arquitectura de microservicios se trata de pequeños servicios autónomos, pequeños y con su única implementación individual de funcionalidad y También llegando a tener distintos host o servicios interactuando con protocolos http mediante APIs RESTFull.

Características:

- La descomposición de varias de sus partes funcionales de forma independiente lo que hace que implementar algún requerimiento evitando el redespliegue de toda la aplicación.
- Centrado en características y funcionalidades de forma independiente

Ventajas.

- Esto nos da distintas ventajas como modularidad en las funcionalidades.
- Manejo independiente de despliegue de servicios

Desventajas.

- Alto consumo de memoria
- Tiempo para fragmentar distintos microservicios
- Necesidad de desarrolladores para la solución de problemas como latencia en la red o balanceo de cargas.

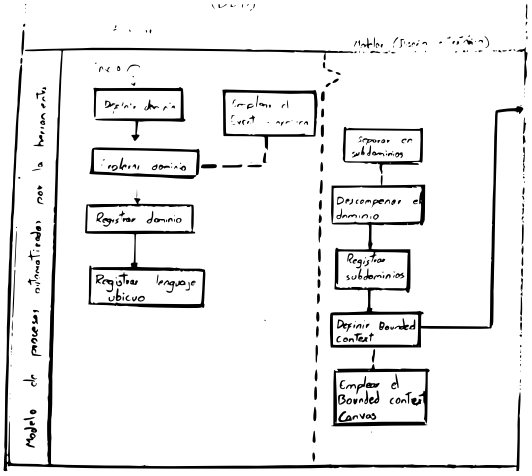
Reflexión:

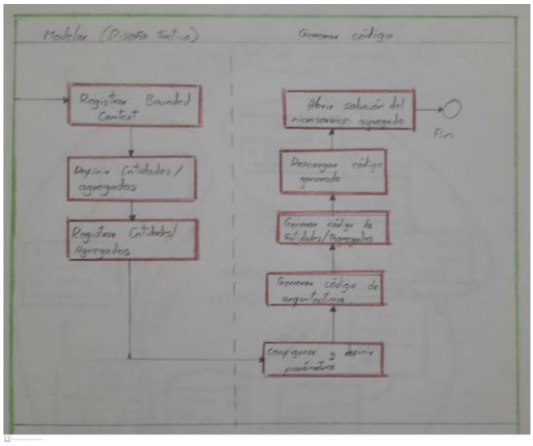
Según lo presenciado puedo decir que la arquitectura de microservicios ha sido un gran referente en la industria ya que a demostrado ser una arquitectura lo que facilita la adaptación a nuevos requerimientos sin afectar el funcionamiento global del sistema, teniendo gran modularidad lo cual sirve para tener una gran escalabilidad y adaptabilidad.

Bibliografía:

Velasco, J. I. P., Ruiz, A. I. R., & Alvira, H. A. P. (2019). Arquitectura basada en microservicios para aplicaciones web. *Tecnología Investigación y Academia*, *7*(2), 12–20. https://revistas.udistrital.edu.co/index.php/tia/article/view/13364

Herramienta para el modelado y generación de código de Arquitecturas de Software basadas en Microservicios y Diseño guiado por el dominio (DDD)





Desarrollo de una herramienta de modelado y generador de código de arquitectura de software basada en Microservicios y Diseño Dirigido por Dominio (DDD) que permita facilitar y acelerar el desarrollo de proyectos de software y la migración de sistemas heredados a nuevos servicios independientes, implementando nuevas tecnologías que permitirán al software tener interoperabilidad, modularidad, seguridad y escalabilidad.

Realizando pruebas con distintos arquitectos y desarrolladores de software expertos, donde se les dio instrucciones de cómo manejar la herramienta con el fin de implementarla en sus requerimientos individuales.

Reflexión:

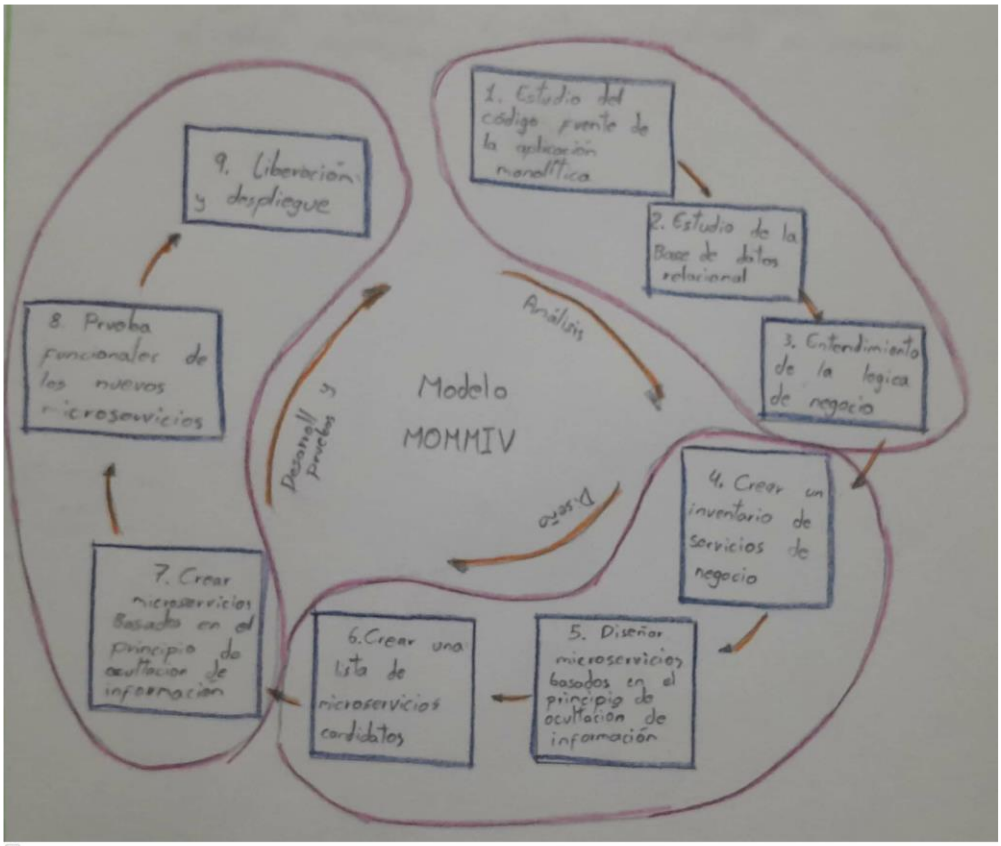
Creo que esta herramienta demuestra ser muy útil para poder generar arquitectura funcional, siendo esta de microservicios y DDD permite tener modularidad característica importante para tener escalabilidad y adaptabilidad en el sistema las cuales son importantes en la industria. Siendo este modelo ya esta probado por expertos se puede asegurar que es una Herramienta funcional.

Bibliografía:

Trebejo Loayza, W. J., & Sobero Rodríguez, F. (2022). Herramienta para el modelado y generación de código de Arquitecturas de Software basadas en Microservicios y

Diseño guiado por el dominio (DDD). *Revista peruana de computación y sistemas*, *4*(2), 3–14. https://doi.org/10.15381/rpcs.v4i2.24855

MOMMIV: Modelo para descomposición de una arquitectura monolítica hacia una arquitectura de microservicios bajo el Principio de Ocultación de Información



Se analizo que las empresas tienen softwares deficientes con el Patrón Arquitectónico Monolítico, el cual merma su funcionalidad con el paso del tiempo, estas necesitan una actualización, en este documento nos propone un modelo para migrar del patrón monolítico al patrón de microservicios. Siendo el punto de partida que no encontraron un modelo de migración usando el principio de Ocultación de información han creado el modelo MOMMIV (Modelo de Migración a Microservicios Versátil). Este estando dividido en Análisis, Diseño, Desarrollo.

Análisis: En esta etapa se analiza tanto el código fuente, componentes, dependencias y la base de datos relacional para tener la funcionalidad de la aplicación monolítica.

Diseño: Una vez analizado el software se procede a hace un “inventario” de las funcionalidades del programa, y se procede diseñar nuevos microservicios guiado por el principio de ocultación de información.

Desarrollo: se crean los microservicios basándose en el Principio de Ocultación de Información junto con su mecanismo de comunicación, Se realizan pruebas funcionales comparándolas con las funciones de la aplicación monolítica. Finalmente se procede a realizar el debido despliegue.

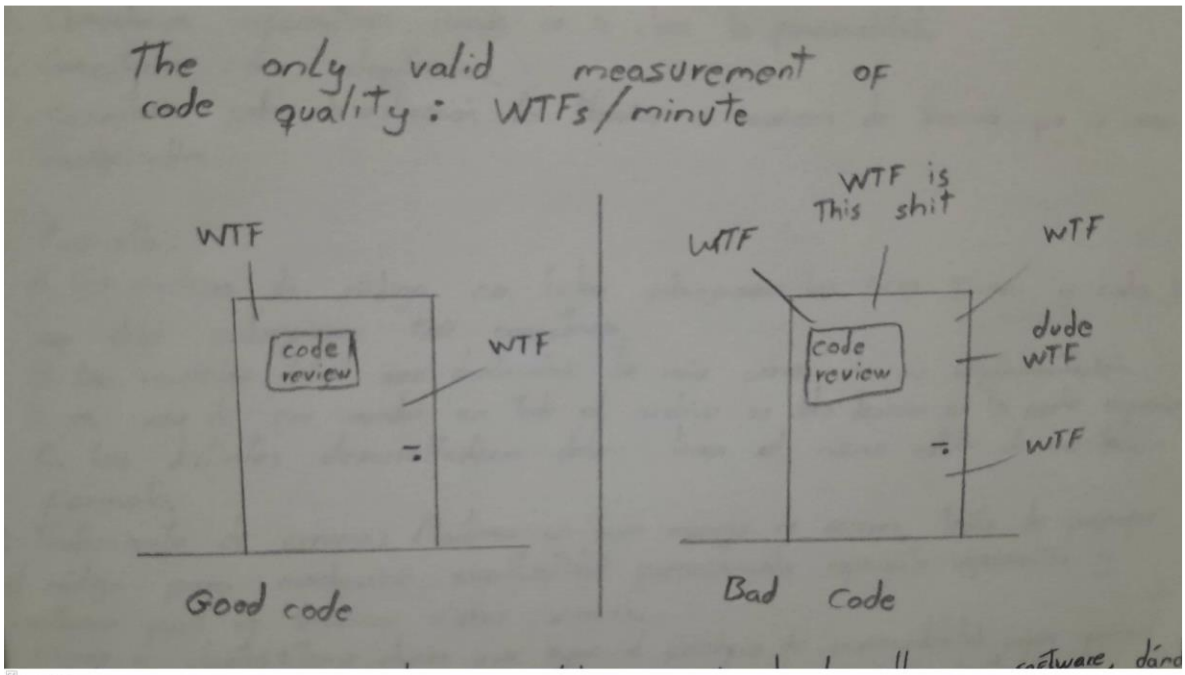
Reflexión:

Creo que el modelo MOMMIV proporciona un marco estructurado y versátil para la migración de arquitecturas monolíticas a microservicios manteniendo el principio de ocultación de información, ya que al dividir las tareas por secciones esta permite reconsiderar los resultados en cada una y devolverse a un paso anterior de ser necesario. Siendo este modelo divide las funcionalidades de la aplicación y busca construir un microservicio con cada una creo que es muy modular.

Bibliografía:

Velepucha, V., Flores, P., & Torres, J. (2019). MOMMIV: Modelo para descomposición de una arquitectura monolítica hacia una arquitectura de microservicios bajo el principio de ocultación de información. [MOMMIV: Model for the decomposition of a monolithic architecture towards a microservices architecture under the Principle of Information Hiding] *Revista Ibérica De Sistemas e Tecnologias De Informação,* , 1000-1009. Retrieved from https://login.bdigital.sena.edu.co/login?url=https://www.proquest.com/scholarly-journals/mommiv-modelo-para-descomposición-de-una/docview/2195127731/se-2

Buenas prácticas en la construcción de software



Nos habla de las buenas prácticas que debemos seguir al desarrollar un software, dándonos a conocer de malas prácticas que debemos evitar como no tener una arquitectura diseñada para realizar un software de calidad, la falta de comunicación entre los miembros de un equipo lo cual afecta negativamente los estándares de calidad del software, llevando a la práctica de los anti-patrones.

- copy and paste programming
- reinventing the Wheel

Buenas prácticas:

1. estándares y buenas prácticas: Aplicar estándares y buenas prácticas al momento de diseñar y escribir el código de las aplicaciones.
2. Clean Code, KISS: Mantener la simplicidad del código con el fin de facilitar la lectura del mismo.
3. Nombre con sentido: El nombre debe dar a entender su funcionalidad, siendo aconsejable mantener nombres cortos y evitar abreviaciones, prefijos y palabras redundantes y evitar usar palabras reservadas.
4. Funciones: Las funciones no deben ser muy complejas, siguiendo el principio de responsabilidad única y seguir la regla “the Stepdown rule”, “Ésta busca que cada función esté seguida de otra que corresponda con el siguiente nivel de abstracción”, en caso de usar parámetros evitar enviar más de tres y evitar repetir código.

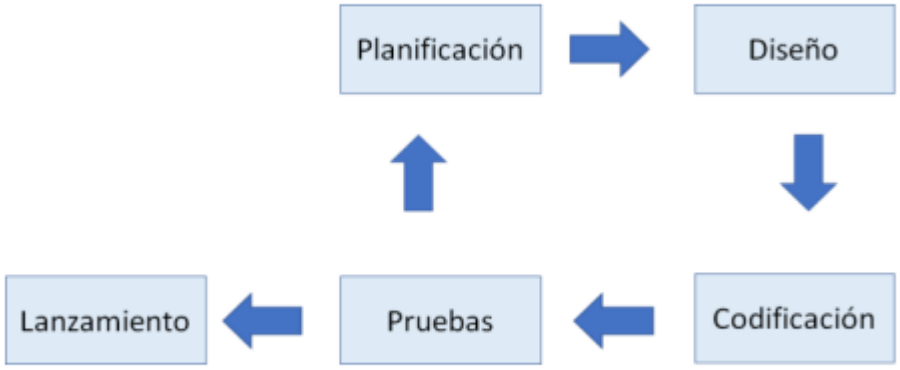
5. Comentarios: Se deben usar cuando el código se da a entender por sí solo. Se recomienda usar en estos casos:
    a. Comentarios legales sobre derechos de autor.
    b. Comentarios informativos cuando no es clara una funcionalidad.
    c. Comentarios sobre advertencias.
    d. Comentarios sobre utilización de librerías o recursos de terceros que no sean modificables

6. Formato:
    a. Un archivo de código no debe sobrepasar las 500 líneas y cada línea no debe sobrepasar 120 caracteres.
    b. las variables deben ser declaradas lo más cerca de su implementación o en caso de ser usadas en todo el archivo se debe declarar en la parte superior.
    c. Los distintos desarrolladores deben tener el mismo estilo de indentación y formato.
7. Tratamiento de errores: Mantener un buen manejo de errores trata de preparar el código para cualquier eventualidad proporcionando suficiente información y mantener fácil de localizar dichos errores.
8. Clases y objetos: Tener clases que sigan el “principio de responsabilidad” única para evitar clases muy grandes y complejas, tener en cuenta el principio de “abierto/cerrado” introduciendo nuevas clases para evitar hacer una modificación de las existentes, seguir el principio de “inversión de dependencias” para facilitar las pruebas y tener un código más limpio.
    a. Acoplamiento: Tener un acoplamiento bajo (nos dice que las clases relacionadas no debe conocer muchos detalles de otra).
    b. Cohesión: Es la independencia de cada clase.
9. Arquitectura: Diseñar el sistema que cubrirá los requerimientos funcionales y no funcionales del software.
    a. Arquitectura Empresarial: Definir correctamente los componentes y activos de una empresa para entender la organización.
    b. Arquitectura Solución: Definir el diseño y comunicaciones de alto nivel del software para guiar el diseño y desarrollo a satisfacer las necesidades.
    c. Arquitectura Software: “Definir patrones de arquitectura, patrones de diseño, estilo arquitectónico, datos y tecnologías que se van a implementar en las soluciones.”

10. Estilos Arquitectónicos: Antes de decidir cual estilo de arquitectura escoger se debe analizar las características de cada uno de estos.
    a. Capas: Este divide la aplicación en capas cada una de estas con un rol definido por ejemplo presentación, servicios etc.
    b. Monolítico: Aplicación autosuficiente, agrupando todo en el mismo proyecto.
    c. Microservicios: Dividir el software en pequeños componentes con una única responsabilidad Permitiendo al software ser independiente de cada una de sus partes.
    d. Event-driven architecture: Esta arquitectura no espera respuestas inmediatas, sino que espera a un evento y reacciona a una respuesta.
    e. Cliente Servidor: Un servidor que brinda una serie de servicios o recursos los cuales son consumidos por el cliente.
11. Patrones de Diseño de software: Soluciones a errores de diseño, siendo generales y aplicables en casos específicos.
    a. Patrones creacionales: “Estos patrones intentan controlar la forma en que los objetos son creados, implementando mecanismos que eviten la creación directa de objetos.”
    b. Patrones estructurales: Estos patrones se encargan de diseñar la comunicación que tienen las clases.
    c. Patrones de comportamiento: “Son patrones que están relacionados con procedimientos y con la asignación de responsabilidad a los objetos.”
12. Herramientas: Antes de construir un software es recomendable realizar una investigación sobre las distintas herramientas disponibles.
13. Metodología XP: Mantener la comunicación con el cliente para realizar un software ajustado.
    a. Cliente: Proporciona los requerimientos y las principales necesidades.
    b. Desarrolladores: Aportar en la construcción y objetivos planteados.
    c. Testers: se encarga de comunicarse con el cliente y definir los estándares de calidad.
    d. Tracker: Realiza el seguimiento de las tareas propuestas y la comunicación con el cliente.
    e. Coach: Orienta al equipo de trabajo y cliente para que todo se haga de manera correcta.
    f. Manager: “El mánager se encarga de coordinar la comunicación entre distintas partes, gestionando los recursos necesarios.”
14. Ciclo de vida metodología XP:

a. Fase de planificación: Entendimiento de los requisitos del cliente, con cada interacción generar una versión del software nueva y funcional.
b. Fase de diseño: “En esta fase se manejan versiones sencillas haciendo lo mínimo necesario para que funcione y se obtenga un prototipo.”
c. Fase de codificación: Desarrollado principalmente en pares para asegurar el entendimiento del software.
d. Fase de pruebas: Realización de pruebas automáticas constantemente.
e. 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



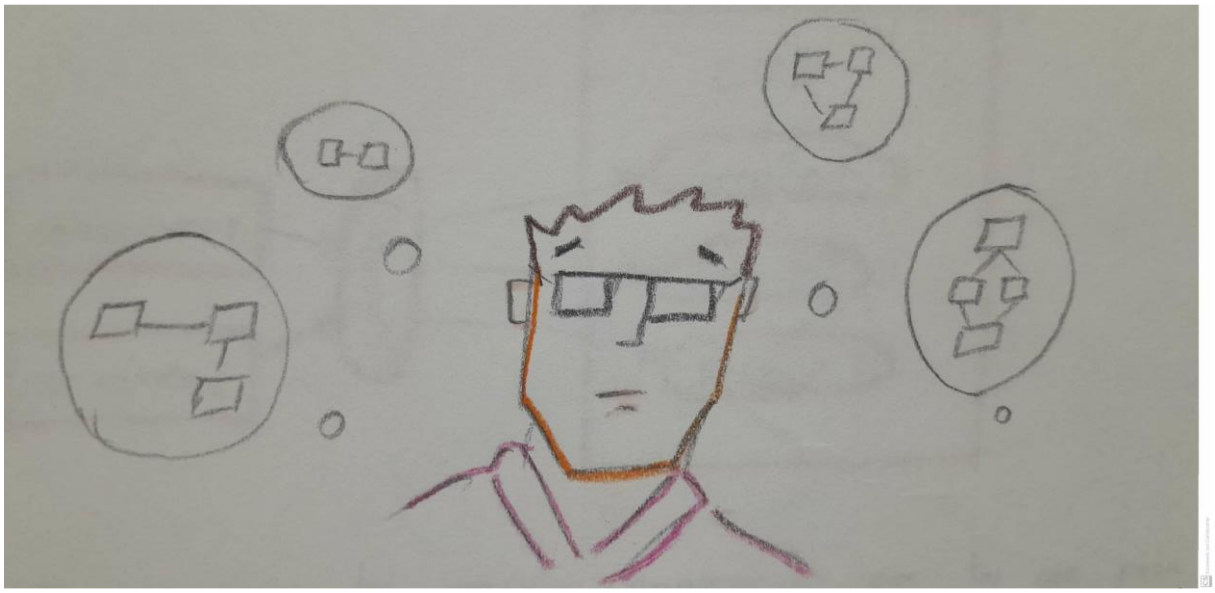
Fuente: Elaboración propia

Reflexión:
Considero que las buenas prácticas son importantes de tener en cuenta a la hora del análisis y desarrollo del software ya que permite que un proyecto sea entendible por distintos desarrolladores y sea más de escalar y adaptar, esto no solo hace que un software se de mayor calidad, sino que optimiza el proceso de desarrollo.

Bibliografía:

Prieto, C. A., & Madrid, D. A. (2022). Buenas prácticas en la construcción de software: Best practices in software construction. *Tecnología Investigación y Academia*, *10*(2), 149–166. https://revistas.udistrital.edu.co/index.php/tia/article/view/21794

Lenguajes de Patrones de Arquitectura de Software: Una Aproximación Al Estado del Arte



Nos habla de la evolución y de los estándares de software a través de los años incluyendo y haciendo énfasis en la arquitectura.

Años Cincuenta: Se tomaba muy en cuenta todas las ciencias relacionadas con el software y el hardware como las matemáticas, ciencias de la computación y demás.

Años Sesenta: Era informal y sin planificación a profundidad.

Años Setenta: Se introdujeron estándares de calidad al software, el clico de vida del software y se identificaron principios de diseño.

Años Ochenta: Busca las buenas prácticas, productividad y escalabilidad teniendo en cuenta las principales cosas beneficiosas para el software y el equipo de desarrollo.

Años Noventa: El software tomo gran fuerza en el área laboral teniendo grandes avances en el desarrollo de patrones de diseño y estandarización de procesos.

Ingeniería de Software: Se preocupa por los métodos y principios de diseño necesarios para el proceso de construcción del software.

Arquitectura de Software: Se centra en el cómo construir el sistema, asegurando que los atributos de calidad del software sean adecuados y que cumplan con las expectativas del usuario final.

Los Patrones de Diseño y los Lenguajes de Patrones: Estos fueron una adaptación de la propuesta del arquitecto Christopher Alexander, que en un principio descompuso problemas complejos en problemas pequeños y daba soluciones generales que funcionaban en distintas situaciones y a estas soluciones le llamo patrones, luego este concepto se adaptó al software.

Lenguajes de patrones: Son agrupaciones de patrones de diseño para resolver problemas complejos y recurrentes

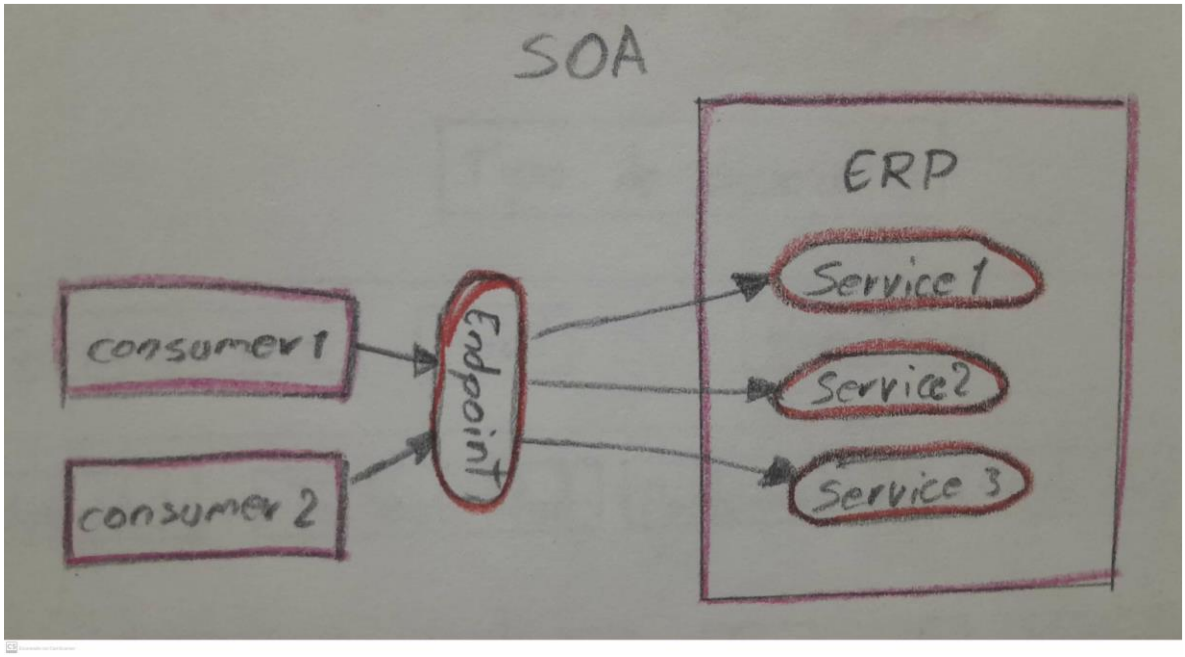
Reflexión:

Puedo ver que la arquitectura de software a tenido un gran crecimiento a través del tiempo siendo sus inicios informales en los años 50 hasta tener un enfoque formal, estandarizado, desarrollado y enfocado en la calidad. Siendo esta importante para resolver grandes problemas de manera más práctica.

Bibliografía:

Scientia Et Technica. (s/f). Redalyc.org. Recuperado el 23 de septiembre de 2024, de https://www.redalyc.org/pdf/849/84933912003.pdf

ARQUITECTURA DE SOFTWARE, ESQUEMAS Y SERVICIOS



Cada vez son más los requisitos empresariales por los que pasa el desarrollo de software teniendo tiempos de desarrollo cada vez más cortos. La reutilización y el bajo acoplamiento son fundamentales para mejorar el desarrollo de software. La arquitectura de microservicios permite adaptarse a este modelo empresarial por su modularidad, su tiempo de desarrollo, su adaptabilidad, flexibilidad y tiempo de mantenimiento siendo un sistema flexible a los cambios del entorno.

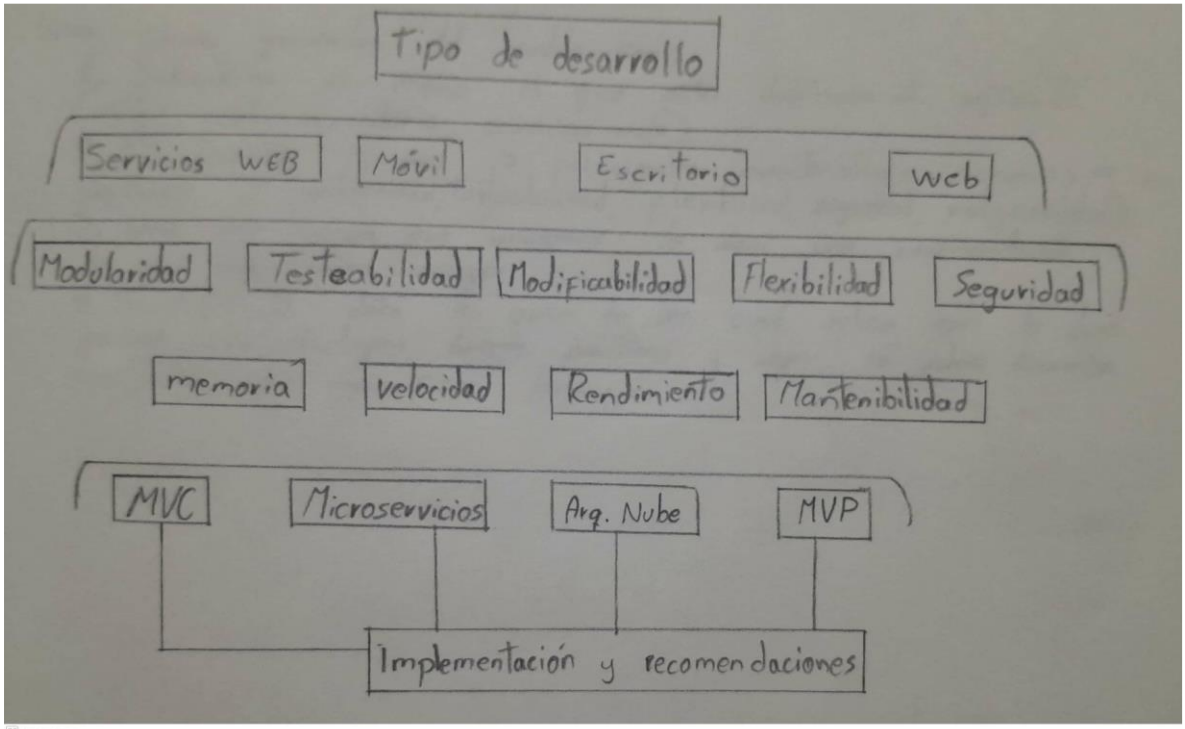
Reflexión:

Yo considero que la arquitectura SOA, al contar con gran modularidad, se ha consolidado en la industria por sus características de escalabilidad y flexibilidad. Estas cualidades son ideales para desarrollar productos de calidad en un entorno cambiante. Su capacidad para adaptarse a los requisitos empresariales y a los tiempos de desarrollo cada vez más cortos la convierte en una solución efectiva para enfrentar los desafíos actuales del desarrollo de software.

Bibliografía:

(S/f-c). Unirioja.es. Recuperado el 23 de septiembre de 2024, de https://dialnet.unirioja.es/servlet/articulo?codigo=4786655

Marco de Trabajo para Seleccionar un Patrón Arquitectónico en el Desarrollo de Software



Se realizo un estudio a desarrolladores y arquitectos de software profesionales sobre cuáles son las arquitecturas de software más usadas.

Luego de analizar las distintas arquitecturas, se define cuáles son las más usadas son las más usadas, en que se destaca cada una.

Siendo las más usadas
Arquitectura en la nube: Seguridad y flexibilidad.

MVC: Mantenibilidad, rendimiento, velocidad y memoria.

Microservicios: Mantenibilidad, Rendimiento, seguridad y flexibilidad.

MVP: Modificabilidad, Rendimiento, testeabilidad, flexibilidad, Modularidad.

También se analizó los distintos dispositivos a los que estaba enfocado esas arquitecturas:

Arquitectura en la nube: Aplicaciones web

MVC: Dispositivos móviles, Aplicaciones de escritorio, Aplicaciones basadas en la web

Microservicios: Aplicaciones web

MVP: Dispositivos móviles y Aplicaciones web

Con estos y otras investigaciones realizadas por el equipo de trabajo se llegó a desarrollar un modelo para escoger la arquitectura adecuada según sus requisitos.

Unos pasos generales del modelo son:

1. Selecciona el medio al que está destinado el software (Móvil, WEB, Escritorio, Servicios Web)
2. El usuario escoge las 3 principales características necesarias en su software (Modularidad, Testeabilidad, flexibilidad, seguridad, modificabilidad)
3. Una vez escoja sus opciones le dará una recomendación de que arquitectura escoger.
4. También le dará la opción de ver cómo aplicar, esta le dará pasos que incluyen buenas prácticas y como se podría desarrollar ese patrón sin especificar lenguajes o tecnologías.

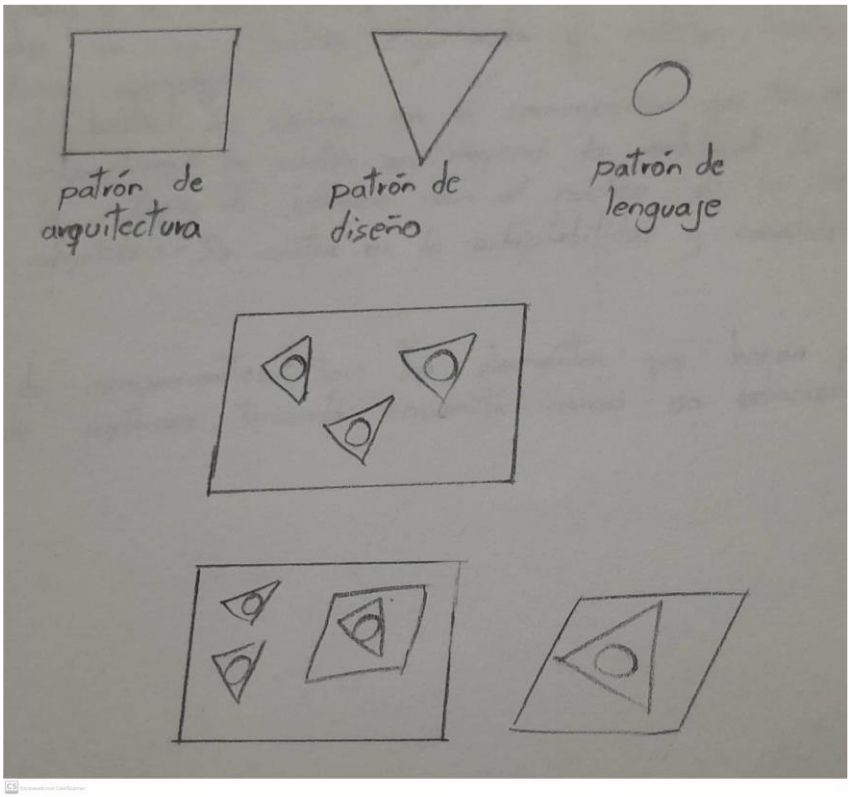
Reflexión:

Me parece que este marco de trabajo es una herramienta muy útil para el proceso inicial de un proyecto, ya que facilita la definición de la arquitectura adecuada para el sistema, considerando los requisitos necesarios. Además, proporciona una guía para implementar la arquitectura, esto ayuda a los desarrolladores a seguir buenas prácticas y orientar su trabajo de manera efectiva. Esto ayuda principalmente en etapas iniciales de proyecto optimizando en tiempo necesario.

Bibliografía:

(S/f-d). Proquest.com. Recuperado el 23 de septiembre de 2024, de https://www.proquest.com/docview/2562269745/32A222F5EEBC4B15PQ/1?accountid=31491&sourcetype=Scholarly%20Journals

Especificando una arquitectura de software



Este articulo busca contextualizar la arquitectura de software dejando claro algunos conceptos relacionados a este.

1. ARQUITECTURA: las arquitecturas son un proceso creativo y estas van muy ligadas a lo que los arquitectos involucrados consideren convenientes basados en las siguientes tres fuentes:
    a. Método: Puede ser visto como una manera concisa y documentada, mediante el cual la arquitectura es derivada desde los requerimientos del sistema y las restricciones tecnológicas.
    b. Intuición: Habilidad de concebir sin razonamiento consciente.
    c. Reutilización: La mayoría de los elementos de una arquitectura son adoptados de otras arquitecturas, ya que el arquitecto puede estar familiarizado con la problemática o con alguna arquitectura encontrada en la literatura técnica.
2. ELECCIÓN DE PATRONES DE REFERENCIA: es importante entender que los patrones se dividen en varios tipos, los principalmente son:
    a. Patrones de arquitectura: Afectan la estructura global del sistema
    b. Patrones de diseño: Afecta a subsistemas o componentes del sistema global y sus relaciones.

    c. Patrones de lenguajes: describen como implementar ciertos aspectos de un problema utilizando las características específicas de dicho lenguaje.
3. Los patrones arquitectónicos se dividen en cuatro categorías
    a. Del lodazal a la estructura: Se centra en la organización de componentes desorganizados en una estructura organizada y modular. Siendo especialmente útil en sistemas complejos.
    b. Sistemas distribuidos: Se centra en la comunicación que tiene el sistema
    c. Sistemas interactivos: Se centra en mejorar la usabilidad de la aplicación separando la interfaz de usuario del núcleo de la lógica.
    d. Sistemas adaptables: Se centra en la adaptabilidad y la escalabilidad de un sistema
4. DEFINICIÓN DE COMPONENTES: Son los elementos que hacen parte de la arquitectura de software. Teniendo encueta como se relacionan.

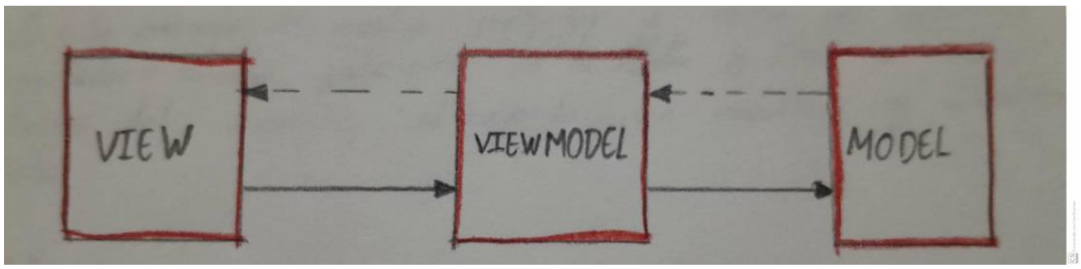
Reflexión:

Yo creo que el entendimiento correcto de la arquitectura de software es Importante para todo el equipo de desarrollo, pues establece una base conceptual para trabajar en sinergia en el proyecto. Conocer los principios, patrones y componentes que la conforman permite a los miembros del equipo colaborar de manera más efectiva.

Bibliografía:

Quilindo, L. A. (2022). Especificando una arquitectura de software: Specifying a software architecture. *Tecnología Investigación y Academia*, *10*(2), 136–148. https://revistas.udistrital.edu.co/index.php/tia/article/view/18080

Análisis comparativo de Patrones de Diseño de Software



artículo se detallan la estructura, componentes, ventajas y desventajas de los patrones de diseño.

1. Patrón Template Method - Modelo Plantilla: Es un patrón de diseño de comportamiento donde se define el esqueleto de un algoritmo en la superclase, mientras que, las subclases pueden sobrescribir los pasos del algoritmo sin la necesidad de cambiar su estructura.
    a. Ventajas
        i. Evita duplicación de código
        ii. Es fácil de entender e implementar
        iii. Flexible
    b. Desventajas
        i. Seguir el flujo es complejo a la hora de depurar
        ii. Poca mantenibilidad
2. Model-View-Controller MVC: Separado en tres componentes Modelo, vista, controlador.
    a. Estructura:
        i. Modelo: Se encarga de manipular y gestionar los datos.
        ii. Vista: Se encarga de mostrar las pantallas al usuario.
        iii. Controlador: Se encarga de procesar las instrucciones recibidas y comunicar a la vista y al modelo.
    b. Ventajas:
        i. Modularidad.
        ii. Mantenibilidad.
    c. Desventajas:
        i. Gran crecimiento del programa
        ii. Implementación complicada con lenguajes que no sean compatibles con el paradigma orientado a objetos.
3. Model-View-Presenter MVP: Separado en tres componentes Modelo, vista, Presentador.
    a. Estructura:

        i. Modelo: Contiene los datos y la lógica, oculta su implementación con interfaces.
        ii. Vista: Se encarga de mostrar las pantallas al usuario y recibir sus peticiones.
        iii. Presentador: Recibe las peticiones de la vista e invoca métodos del modelo, obteniendo un resultado y actualiza la vista.
    b. Ventajas:
        i. Flexibilidad
        ii. Permite la integración de distintas tecnologías
        iii. La vista y el modelo pueden ser testeados de manera independiente.
    c. Desventajas:
        i. Implementación Compleja.
        ii. No es adecuado para soluciones simples y pequeñas
4. Model Front Controller: Se centra en el manejo de peticiones, usando como punto inicial un controlador que realiza la gestión de solicitudes.
    a. Estructura:
        i. Controller: Procesa todos y gestionan las peticiones del sistema.
        ii. Dispatcher: Se encarga de mostrar las pantallas al usuario y navegar entre vistas.
        iii. Helper: ayuda al controlador y a la vista a terminar su procesamiento.
    b. Ventajas
        i. Existe una mejora significativa en la manejabilidad de la seguridad.
        ii. Es posible reutilizar código.
        iii. Evita tener distribuida la gestión de peticiones.
        iv. Control centralizado
        v. Seguridad de subprocesos
    c. Desventajas
        i. Poca escalabilidad.
        ii. Baja velocidad de respuesta
5. Model-View-ViewModel MVVM: permite aislar limpiamente la lógica de negocios y presentación de una aplicación de su interfaz de usuario.
    a. Estructura:

        i. Modelo: Es el responsable del acceso a la fuente de datos y de trabajar con esos datos.
        ii. Vista: Se encarga de mostrar las pantallas al usuario y recibir sus peticiones.
        iii. Vista del modelo: Encargada me transformar los datos en un formato que la vista lo requiera.
    b. Ventajas
        i. Facilita el mantenimiento
        ii. Disminuye la cantidad de código
        iii. Facilidad en pruebas unitarias
    c. Desventajas:
        i. Curva de aprendizaje alta
        ii. Depuración compleja

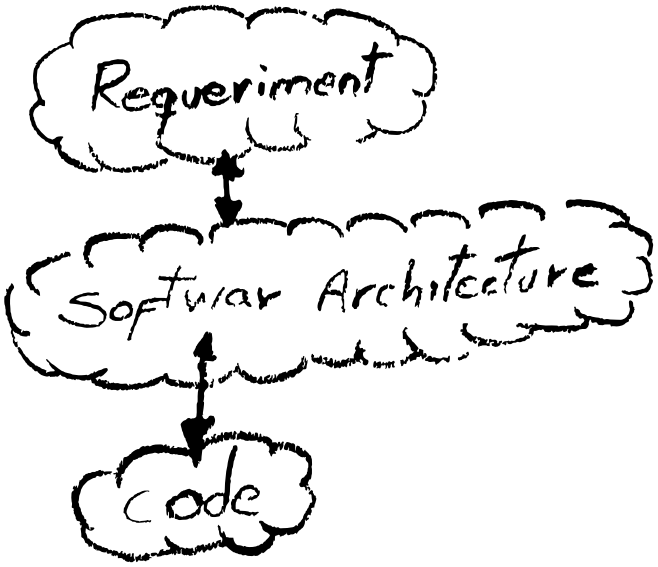
Reflexión:

Considero que conocer los distintos patrones arquitectónicos permite integrarse mejor a un marco de trabajo ya que se entenderá las ventajas y desventajas del sistema, así como también el funcionamiento fundamental del sistema de manera conceptual. Esto ayuda a la colaboración de distintas partes del equipo y tomar decisiones asertivas en cada etapa del proyecto, y tomar decisiones viables para el proyecto.

Bibliografía:

(S/f-e). Recuperado el 23 de septiembre de 2024, de http://file:///C:/Users/juan/Downloads/Dialnet-AnalisisComparativoDePatronesDeDisenoDeSoftware-9042927.pdf

Revisión de elementos conceptuales para la representación de las arquitecturas de referencias de software



Elementos clave:

1. Componente: elemento computacional primario de un sistema.
2. Conector: elemento que permite la interacción entre componentes.
3. Configuración: estructura de los componentes y conectores.
4. Restricción: limitaciones en la configuración y comportamiento.
5. Estilo arquitectónico: patrones de diseño para la organización de componentes.
6. Actor: entidad que interactúa con el sistema.
7. Rol: función desempeñada por un actor.
8. Servicio: funcionalidad ofrecida por un componente.

la representación de conocimiento de las arquitecturas de referencias de software requiere considerar los 8 conceptos clave identificados. El componente es el elemento arquitectural base, citado en el 100% de los trabajos revisados. La conceptualización de arquitecturas de software y arquitecturas de referencias de

software debe considerar estos elementos para garantizar una representación efectiva y eficiente.

El artículo examina en profundidad las diferentes formas de representar arquitecturas de software, destacando la importancia de comprender cómo se descomponen y definen sus elementos esenciales para lograr una representación clara y coherente. En particular, se exploran y analizan los métodos más utilizados para describir dichas arquitecturas, poniendo énfasis en el uso del lenguaje natural como una herramienta común y accesible en este ámbito. El lenguaje natural permite una comunicación más sencilla y comprensible entre los involucrados en el proceso de desarrollo de software, lo que lo convierte en una opción popular. No obstante, se señala que, a pesar de sus ventajas en términos de accesibilidad, este enfoque presenta limitaciones significativas cuando se trata de realizar un análisis más riguroso y detallado de las arquitecturas de software. En especial, el lenguaje natural puede carecer de la precisión y la estandarización necesarias para representar con exactitud todos los aspectos técnicos y complejos que involucra una arquitectura. Esto puede generar ambigüedades o interpretaciones incorrectas, lo que dificulta la implementación de soluciones eficientes. Por ello, el artículo sugiere que, aunque el lenguaje natural es útil en las primeras fases del diseño, es fundamental complementarlo con otras representaciones más formales o técnicas que permitan un análisis más profundo y riguroso de los sistemas. De este modo, se puede lograr una mejor estandarización en la descripción de arquitecturas de software, facilitando tanto su desarrollo como su posterior mantenimiento.

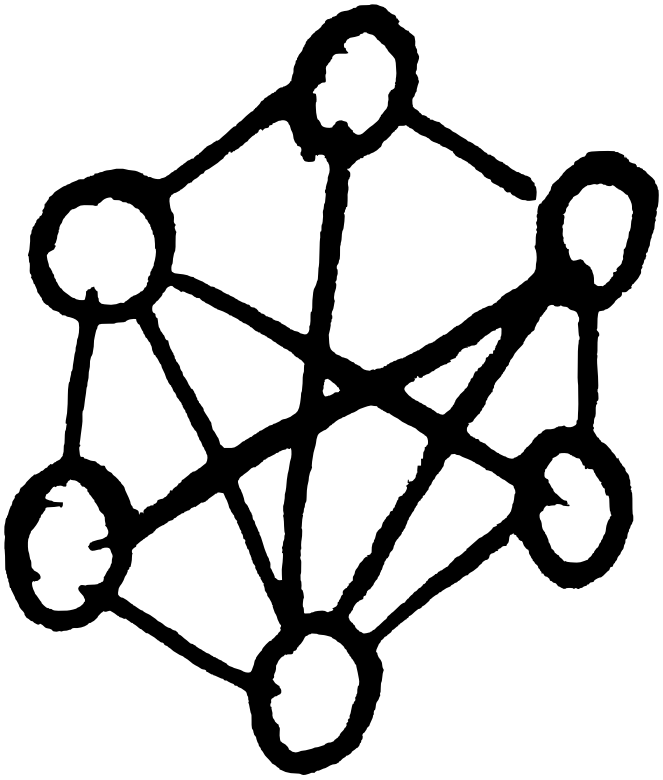
Reflexión:

Considero que Tener una base conceptual es de gran importancia ya que nos ayuda a comprender cómo se componen generalmente las arquitecturas, para comprender su funcionamiento, tener una comunicación adecuada con otros miembros del equipo y tener una mejor planeación ayudando así a resolver problemas.

Bibliografía:

Palmero, M. A. S., Martínez, N. S., & Grass, O. Y. R. (2019). Revisión de elementos conceptuales para la representación de las arquitecturas de referencias de software. *Revista cubana de ciencias informáticas*, *13*(1), 143–157. http://scielo.sld.cu/scielo.php?pid=S2227-18992019000100143&script=sci_arttext

Modelo Teórico para la Identificación del Anti-patrón “Stovepipe System” en la Etapa de la Implementación de una Arquitectura de Software



El análisis se centra en identificar anti-patrones de desarrollo de software, como el "Stovepipe System", que generan sistemas fragmentados y difíciles de integrar. Se propone el uso de redes neuronales y aprendizaje automático para detectar estos anti-patrones y mejorar la calidad del software.

Objetivos del análisis

1. Identificar anti-patrones de desarrollo de software.
2. Abordar el problema de los sistemas fragmentados y difíciles de integrar.
3. Mejorar la calidad del software.

Anti-patrón "Stovepipe System"

1. Genera sistemas fragmentados.
2. Difícil integración entre módulos.
3. Complica el mantenimiento y la escalabilidad.

Propuesta de solución

1. Uso de redes neuronales.
2. Análisis de grandes volúmenes de datos.
3. Detección de patrones que indican la presencia de anti-patrones.
4. Aprendizaje automático para identificar comportamientos repetitivos y problemáticos.

Beneficios

1. Mejora la calidad del software.
2. Evita problemas futuros asociados a la arquitectura del sistema.
3. Facilita la toma de decisiones correctivas a tiempo.

Palabras clave

1. Anti-patrones de desarrollo de software
2. Stovepipe System
3. Redes neuronales
4. Aprendizaje automático
5. Calidad del software
6. Arquitectura del sistema

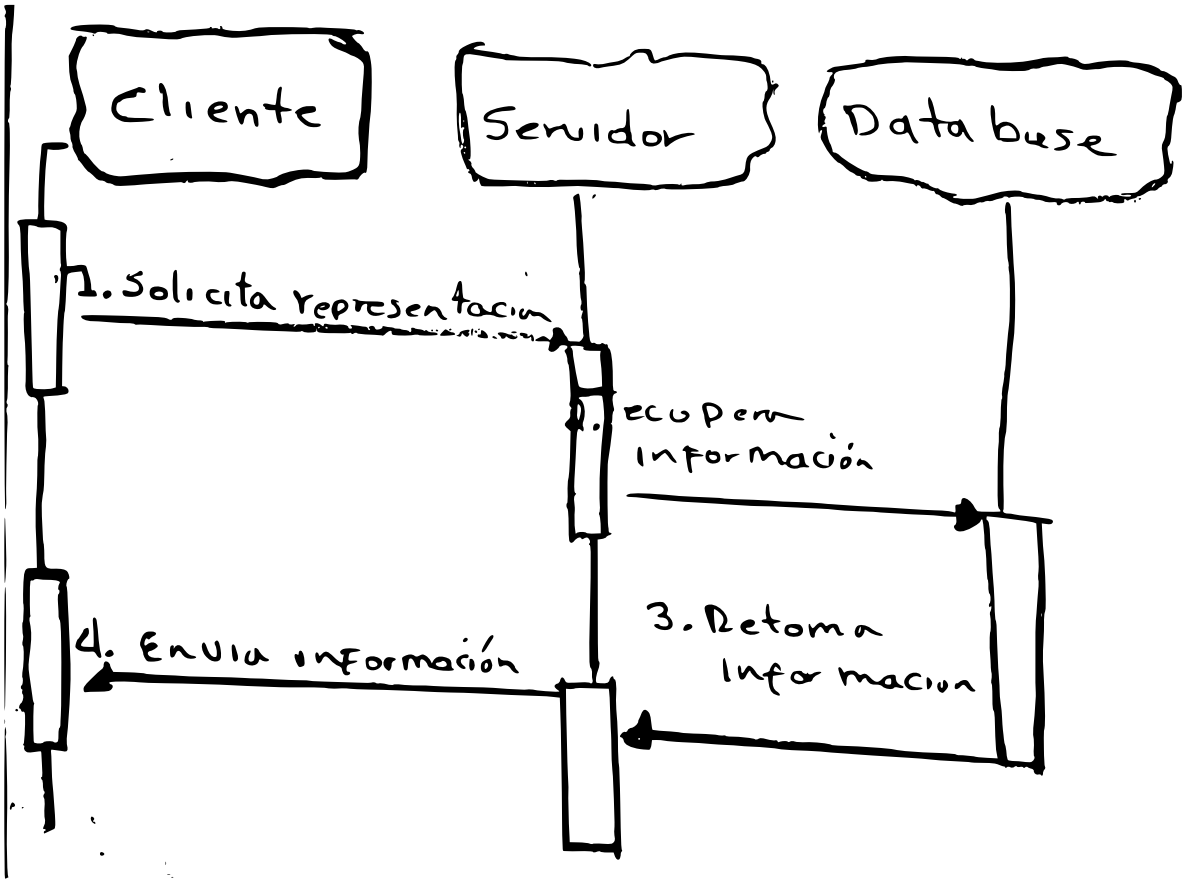
Reflexión:

Considero que identificar los anti-patrones como el "Stovepipe System" es crucial en el desarrollo de software, ya que estos pueden llevar a la creación de sistemas fragmentados y difíciles de integrar, afectando gravemente la calidad y mantenibilidad del producto final. Tener un sistema de redes neuronales que puedan identificar los anti-patrones sería un gran avance ya que ayudaría a evitarlos de manera fácil y rápida.

Bibliografía:

(S/f-f). Umsa.bo. Recuperado el 23 de septiembre de 2024, de http://revistasbolivianas.umsa.bo/pdf/rpgi/n1/n1_a23.pdf

Arquitectura basada en Microservicios y DevOps para una ingeniería de software continua



El proyecto SIGAP implementó una arquitectura basada en microservicios y principios DevOps para optimizar los procesos de ingeniería de software y aumentar la productividad del equipo de desarrollo. La arquitectura de microservicios permite descomponer la aplicación en servicios independientes, facilitando su mantenimiento y escalabilidad. La integración de DevOps promueve la colaboración, automatización y mejora continua, resultando en ciclos de entrega más rápidos y confiables.

Ventajas de la arquitectura de microservicios y DevOps

1. Mayor flexibilidad y adaptabilidad a cambios y requerimientos del cliente.
2. Mejora la calidad del software.
3. Facilita el mantenimiento y escalabilidad.
4. Permite entregas de software más ágiles y controladas.
5. Promueve la colaboración y automatización.

Beneficios del proyecto SIGAP

1. Mejora la productividad del equipo de desarrollo.
2. Optimiza los procesos de ingeniería de software.
3. Proporciona una base sólida para la evolución del software.
4. Es adaptable a otros dominios fuera del ámbito de SIGAP.

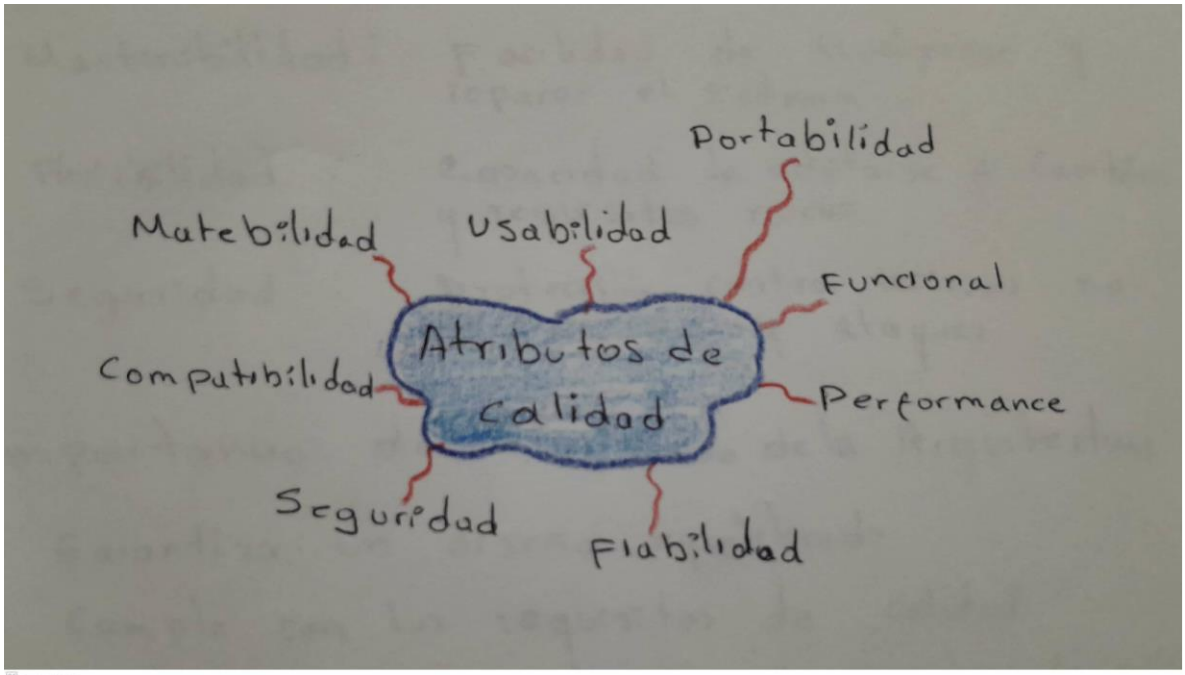
Reflexión:

Considero que la implementación de una arquitectura de microservicios y practicas DevOps demuestra efectividad en su caso, pero demostrando la posibilidad de ser implementado perfectamente en otros contextos y poder ser efectivo.

Bibliografía:

Mamani Rodríguez, Z. E., Del Pino Rodríguez, L., & Gonzales Suarez, J. C. (2020). Arquitectura basada en Microservicios y DevOps para una ingeniería de software continua. *Industrial data*, *23*(2), 141–149. https://doi.org/10.15381/idata.v23i2.17278

Atributos de Calidad y Arquitectura del Software



Este artículo analiza la arquitectura de software, destacando conceptos clave y su importancia en el desarrollo de sistemas robustos. Examina atributos de calidad como escalabilidad, mantenibilidad, flexibilidad y seguridad, y su relevancia en la elección de una arquitectura adecuada.

Conceptos clave de la arquitectura de software

1. Escalabilidad
2. Mantenibilidad
3. Flexibilidad
4. Seguridad
5. Modelado de la arquitectura desde diferentes perspectivas

Atributos de calidad

1. Escalabilidad: capacidad de crecer y adaptarse a nuevas demandas.
2. Mantenibilidad: facilidad de modificar y reparar el sistema.
3. Flexibilidad: capacidad de adaptarse a cambios y requisitos nuevos.
4. Seguridad: protección contra accesos no autorizados y ataques.

Importancia del modelado de la arquitectura

1. Garantiza un diseño equilibrado.

2. Cumple con los requisitos de calidad.
3. Evita sacrificar atributos de manera desproporcionada.

Beneficios

1. Desarrollo de sistemas robustos y eficientes.
2. Mejora la calidad del software.
3. Aumenta la productividad y reducción de costos.

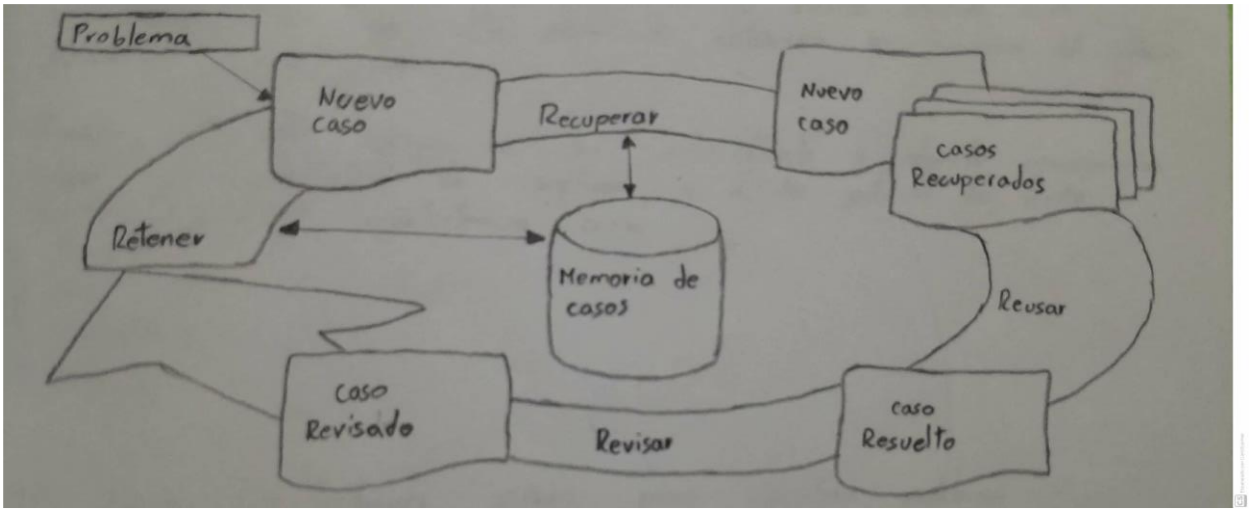
Reflexión:

Considero que entender los atributos de calidad de la arquitectura de software, como la escalabilidad, mantenibilidad, flexibilidad y seguridad, nos permite enfocar mejor nuestro sistema. Al conocer estos aspectos, podemos ofrecer soluciones que se adapten a las necesidades específicas de cada proyecto, creando sistemas más eficientes. Esto resulta en un mejor reconocimiento de problemas y requerimientos, facilitando una respuesta más efectiva y oportuna.

Bibliografía:

Bastarrica, M. C. (s/f). *Atributos de Calidad y Arquitectura del Software*. Upm.es. Recuperado el 23 de septiembre de 2024, de http://www.grise.upm.es/rearviewmirror/conferencias/jiisic04/Tutoriales/tu4.pdf

RADS: Una herramienta para reutilizar estrategias en diseños de arquitecturas de software.



RADS (Recuperación y Aplicación de Decisiones de Software).
Herramienta de almacenamiento de información relevante para la creación de la arquitectura de software.

Capturar información: Se basa en capturar experiencias de experiencias de diseño arquitectónico por las que pasan los arquitectos de software.

Análisis de casos: Se basa en analizar experiencias de otros arquitectos de casos anteriores para ayudar a diseñar la arquitectura.

Buen punto de partida: Proporciona un punto de partida valioso para el diseño de arquitecturas de software.

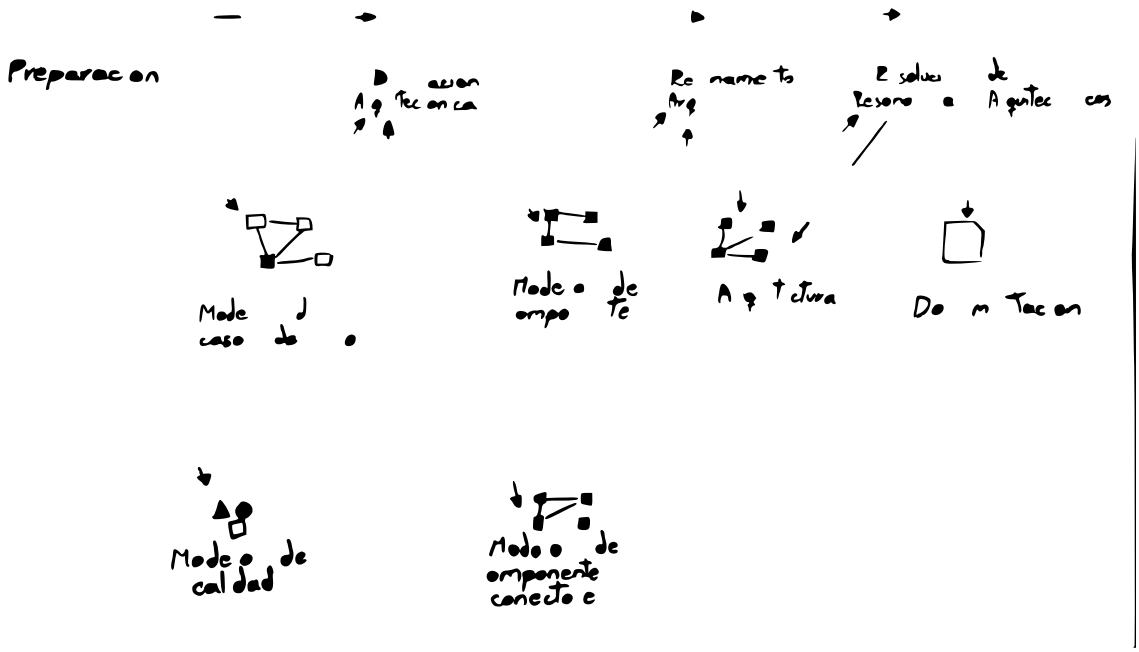
Reflexión:

Creo que la herramienta RADS, al recopilar experiencias de diferentes arquitectos de software, permite comprender y analizar de manera efectiva el proyecto específico en el que se está trabajando. Al comparar este contexto con experiencias pasadas, RADS se convierte en un punto de partida valioso para elegir una arquitectura adecuada. Esto no solo enriquece el proceso de diseño, sino que también brinda claridad y confianza al enfrentar nuevos desafíos arquitectónicos.

Bibliografía:

Carignano, M. C., Gonnet, S., & Leone, H. P. (2016). *RADS: una herramienta para reutilizar estrategias en diseños de arquitecturas de software*. https://core.ac.uk/download/pdf/296390232.pdf

Comparación de métodos para la arquitectura del software: Un marco de referencia para un método arquitectónico unificado



Este artículo presenta una comparación de métodos para la arquitectura del software, con el objetivo de identificar características deseables en un método de diseño arquitectónico. Se propone un marco de referencia para un método arquitectónico unificado, considerando características de calidad.

Objetivos

1. Comparar métodos de diseño y evaluación arquitectónica.
2. Identificar características deseables en un método de diseño arquitectónico.
3. Proponer un marco de referencia para un método arquitectónico unificado.

Beneficios

1. Mejora en la calidad del software.
2. Unificación de métodos de diseño arquitectónico.
3. Facilitación de la evaluación y comparación de métodos.

Palabras clave

1. Arquitectura de software
2. Calidad de software
3. Métodos de diseño arquitectónico

4. Métodos de evaluación arquitectónica
5. Características de calidad

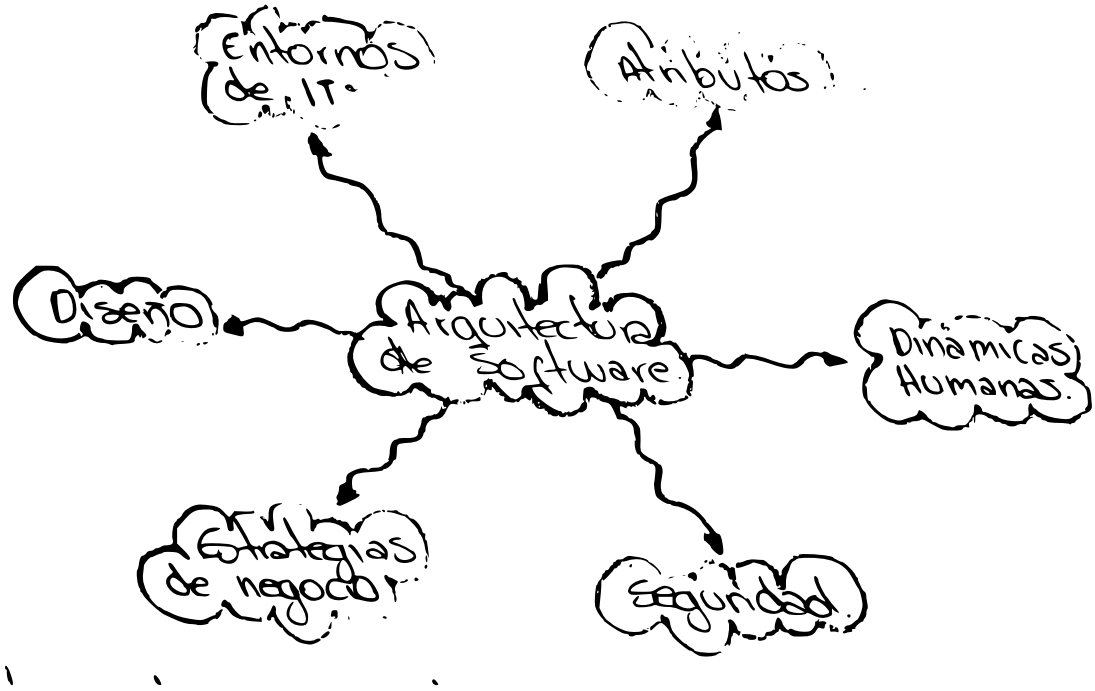
Reflexión:

Considero que la comparación de métodos en la arquitectura del software es clave para mejorar la calidad de los sistemas. Pues al unificar enfoques de diseño y evaluación nos facilita la comprensión de las necesidades principales de una buena arquitectura siendo adaptable a las necesidades específicas de cada proyecto.

Bibliografía:

Losavio, F. (2010). COMPARACIÓN DE MÉTODOS PARA LA ARQUITECTURA DEL SOFTWARE: UN MARCO DE REFERENCIA PARA UN MÉTODO ARQUITECTÓNICO UNIFICADO. *Revista De La Facultad De Ingenieria*, *25*(1), 71–87. https://ve.scielo.org/scielo.php?script=sci_arttext&pid=S0798-40652010000100008

Revisión de elementos conceptuales para la representación de las arquitecturas de referencias de software



La investigación analiza la representación de conocimiento de las arquitecturas de referencia de software, identificando elementos conceptuales y su importancia en la disciplina. Se revisan diferentes lenguajes de descripción arquitectónica (ADL) y se proponen ocho conceptos clave para la conceptualización de las arquitecturas de software y referencia: componente, conector, configuración, restricción, estilo arquitectónico, actor, rol y servicio.

Los elementos fundamentales son:

1. Componente.
2. conector.
3. configuración.
4. restricción.
5. estilo arquitectónico.
6. actor.
7. rol.
8. servicio.

Marco de referencia:

1. Zachman.
2. RM-ODP.
3. C4ISR.
4. TOGAF.
5. Modelo "4+1”.

Objetivos

1. Identificar elementos conceptuales para la representación de conocimiento de las arquitecturas de referencia de software.
2. Analizar la importancia de cada elemento en la disciplina.

Metodología

1. Revisión de literatura sobre lenguajes de descripción arquitectónica (ADL) y arquitecturas de referencia de software.
2. Análisis comparativo de los elementos conceptuales en diferentes ADL.

Resultados

1. Identificación de ocho conceptos clave para la conceptualización de las arquitecturas de software y referencia.
2. El componente es el elemento arquitectural base, citado en el 100% de los trabajos.

Palabras clave

1. Representación de conocimiento
2. Lenguajes de descripción arquitectónica (ADL)
3. Componente
4. Conector

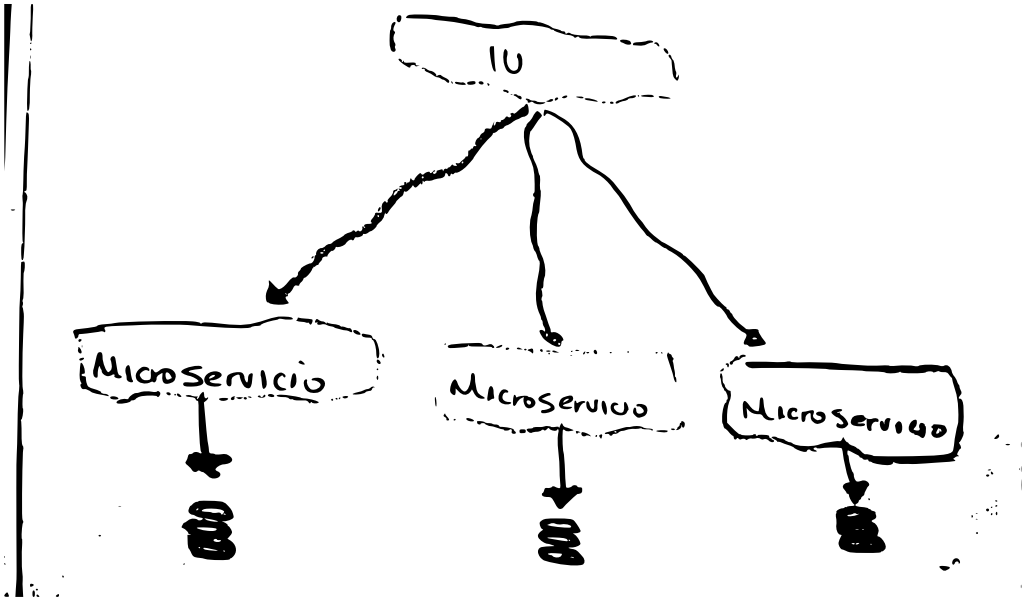
Reflexión:

Considero que tener una base conceptual es de gran importancia ya que nos ayuda a comprender cómo se componen generalmente las arquitecturas, nos sirve para poder resolver problemas, mejorar la comunicación entre los distintos miembros del equipo, tener una mayor resolución de problemas en nuestros casos específicos y tomas decisiones acertadas.

Bibliografía:

Palmero, M. A. S., Martínez, N. S., & Grass, O. Y. R. (2019). Revisión de elementos conceptuales para la representación de las arquitecturas de referencias de software. *Revista cubana de ciencias informáticas*, *13*(1), 143–157. http://scielo.sld.cu/scielo.php?pid=S2227-18992019000100143&script=sci_arttext

Diseño e implementación de una arquitectura de microservicios orientada a trabajar con transacciones distribuidas



Este artículo presenta el diseño e implementación de una arquitectura de microservicios orientada a trabajar con transacciones distribuidas. La arquitectura utiliza patrones como SAGA y Event Sourcing para gestionar transacciones asíncronas y síncronas, y permite la compensación de transacciones en caso de errores.

Arquitectura

API Gateway: Envía las solicitudes de los usuarios a los microservicios.

Microservicios: Gestionan paso a paso las transacciones solicitadas.

EventBridge: Esta toma estados en cada etapa de la transacción, esta aplicara filtros para el correcto funcionamiento de la transacción.

DynamoDB: Guarda la información sobre las transacciones que salieron mal.

Objetivos

1. Diseñar una arquitectura de microservicios para transacciones distribuidas.
2. Implementar patrones de diseño para gestionar transacciones asíncronas y síncronas.

3. Proporcionar una solución escalable y flexible para la gestión de transacciones.

Metodología

1. Utilización de Spring como framework de desarrollo.
2. Implementación de patrones SAGA y Event Sourcing.
3. Uso de Amazon Web Services (AWS) para la gestión de eventos y transacciones.

Arquitectura

1. Microservicios desplegados en instancias de EC2 de AWS.
2. Utilización de Load Balancer para balancear la carga entre instancias.
3. Conexión a EventBridge de AWS para la gestión de eventos.
4. Base de datos NoSQL (DynamoDB) para persistir información de transacciones.

Funcionalidades

1. Gestionar transacciones asíncronas y síncronas.
2. Compensar transacciones en caso de errores.
3. Visualizar transacciones en tiempo real.
4. Permitir la reversión de transacciones.

Palabras clave

1. Arquitectura de microservicios
2. Transacciones distribuidas
3. Patrones de diseño (SAGA, Event Sourcing)
4. Amazon Web Services (AWS)
5. Spring

Reflexión:

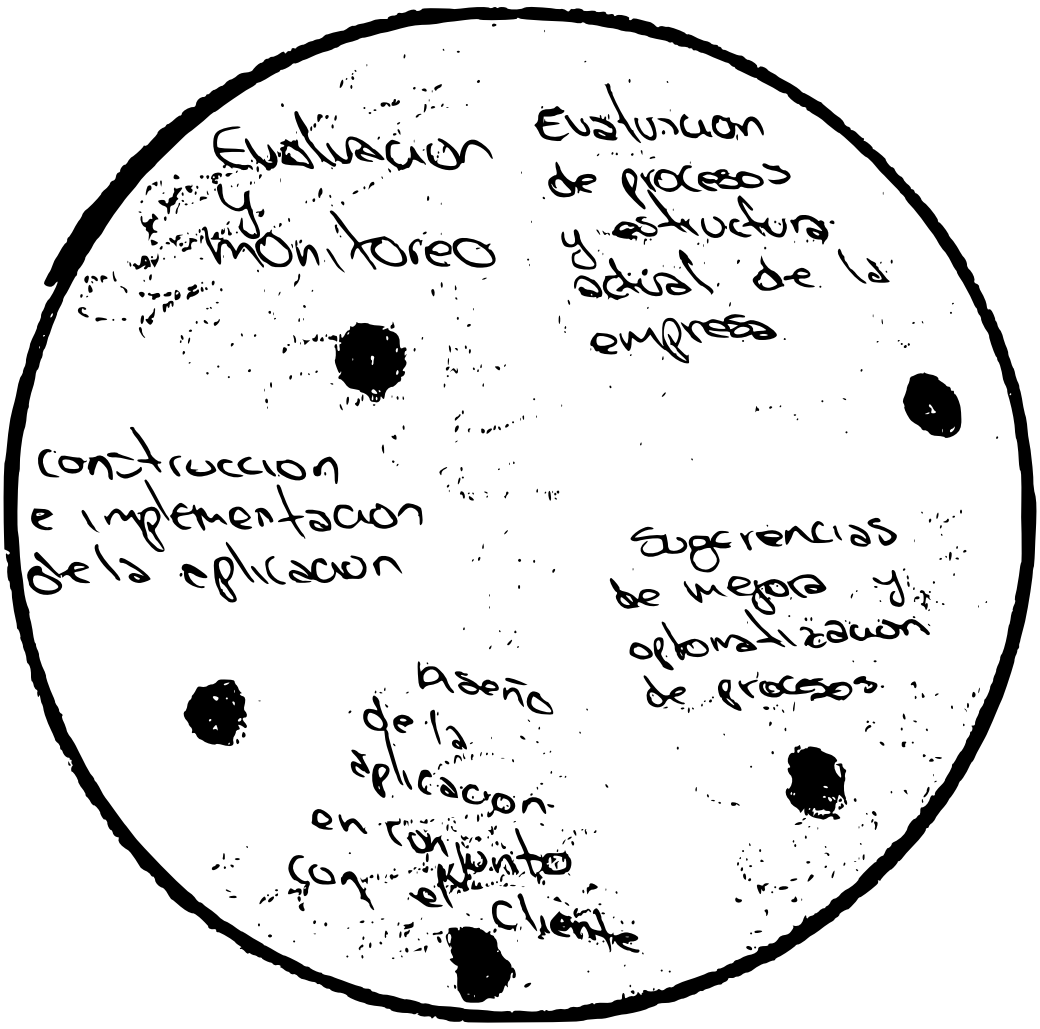
Yo pienso que la arquitectura propuesta maneja de forma asertiva la solución de un complejo desafío en las transacciones distribuidas. Aunque fomentaría al análisis continuo para optimizar la forma en la que se desarrolla para adaptar a las nuevas herramientas y tecnologías.

Bibliografía:

Vista de Diseño e implementación de una arquitectura de microservicios orientada a trabajar con transacciones distribuidas. (s/f). Utp.ac.pa. Recuperado el 23 de

septiembre de 2024, de https://revistas.utp.ac.pa/index.php/id-tecnologico/article/view/3783/4355

Integración de arquitectura de software en el ciclo de vida de las metodologías ágiles



La investigación explora cómo las arquitecturas de software y las metodologías ágiles se interrelacionan y se integran en el desarrollo de software, analizando cómo ambas pueden colaborar para optimizar el proceso de creación. Se enfoca en el alcance de esta integración, destacando la importancia de considerar los "requisitos significativos para la arquitectura". Estos requisitos son cruciales para garantizar que la arquitectura sea flexible y adaptable a los cambios frecuentes que caracterizan a los entornos ágiles. La investigación busca demostrar cómo una arquitectura bien definida puede facilitar la implementación de metodologías ágiles, permitiendo un desarrollo más eficiente y adaptado a las necesidades cambiantes del proyecto, al mismo tiempo que asegura la calidad y coherencia del software.

Objetivo:

Analizar la interrelación entre arquitectura de software y metodologías ágiles

Beneficios:

1. Desarrollo más eficiente
2. Adaptabilidad a cambios frecuentes
3. Calidad y coherencia del software
4. Mejora la colaboración entre equipos

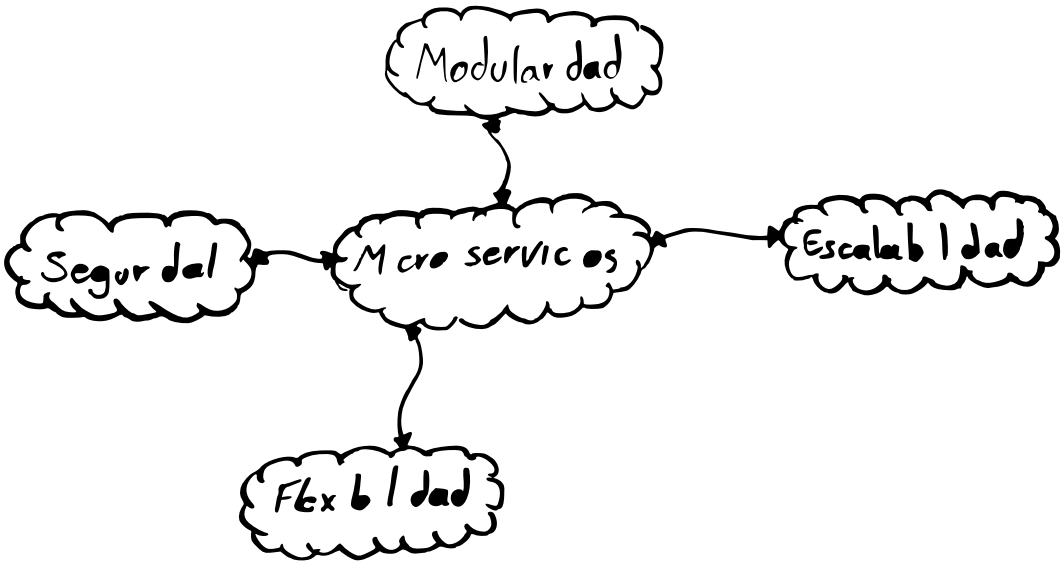
Reflexión:

Yo pienso que integras las metodologías agiles son tan importantes como implementar una arquitectura adecuada para cada proyecto, sin embargo, es muy complicado implementarlos ambos correctamente, Entonces me parece muy adecuado esta investigación para desarrollar una arquitectura ágil.

Bibliografía:

Navarro, M. E., Moreno, M. P., Aranda, J., Parra, L., Rueda, J. R., & Pantano, J. C. (2017). Integración de arquitectura de software en el ciclo de vida de las metodologías ágiles. *XIX Workshop de Investigadores en Ciencias de la Computación (WICC 2017, ITBA, Buenos Aires)*.

Desarrollo de aplicaciones basadas en microservicios: tendencias y desafíos de investigación



El estudio aborda los desafíos en la fase de desarrollo de aplicaciones basadas en microservicios, destacando las tendencias identificadas en cuanto a las características clave de esta arquitectura, como el modularidad, la seguridad y otros atributos esenciales. La investigación revela que la modularidad de los microservicios permite un desarrollo más ágil y flexible, facilitando la actualización y escalabilidad de los componentes de manera independiente. En términos de seguridad, se observa que, aunque cada microservicio puede tener controles de seguridad propios.

Tendencias Clave

1. Modularidad
2. Seguridad
3. Escalabilidad
4. Flexibilidad

Desafíos Principales

1. Coordinación y orquestación de módulos
2. Gestión de seguridad a nivel de sistema
3. Implementación de DevOps y automatización de pruebas

Ventajas

1. Escalabilidad
2. Mantenimiento
3. Flexibilidad

Reflexión:

En mi opinión, este artículo presenta tanto ventajas como desventajas al migrar a una arquitectura de microservicios, destacando tendencias clave como la modularidad, la seguridad, la escalabilidad y la flexibilidad. Aunque la modularidad permite un desarrollo más ágil y la escalabilidad mejora el rendimiento, la gestión de la seguridad y la coordinación de los módulos son desafíos significativos que deben considerarse.

Bibliografía:

(S/f-g). Proquest.com. Recuperado el 23 de septiembre de 2024, de https://www.proquest.com/docview/2348878316/963E0A5318424FD3PQ/3?accountid=31491&sourcetype=Scholarly%20Journals