Juan David Cerquera Salazar



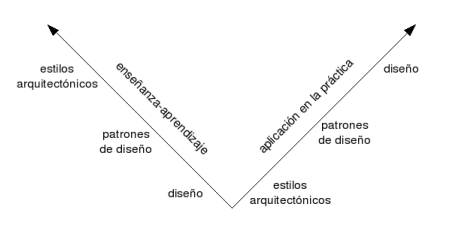
Soy Juan David Cerquera tengo 19 años, actualmente me encuentro cursando un tecnólogo de análisis y desarrollo software en el SENA y una ingeniería en la UNAD. Empecé a desarrollarme como programador desde finales de mis años escolares con proyectos básicos, lo que me ha permitido desarrollar gusto a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

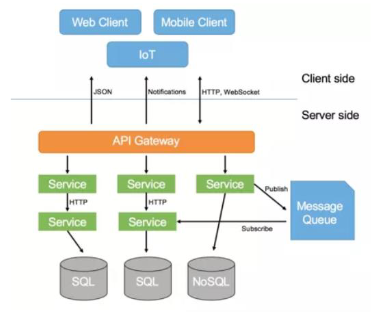
3. Diseño de componentes

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.

**Arquitectura basada en micro-servicios 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

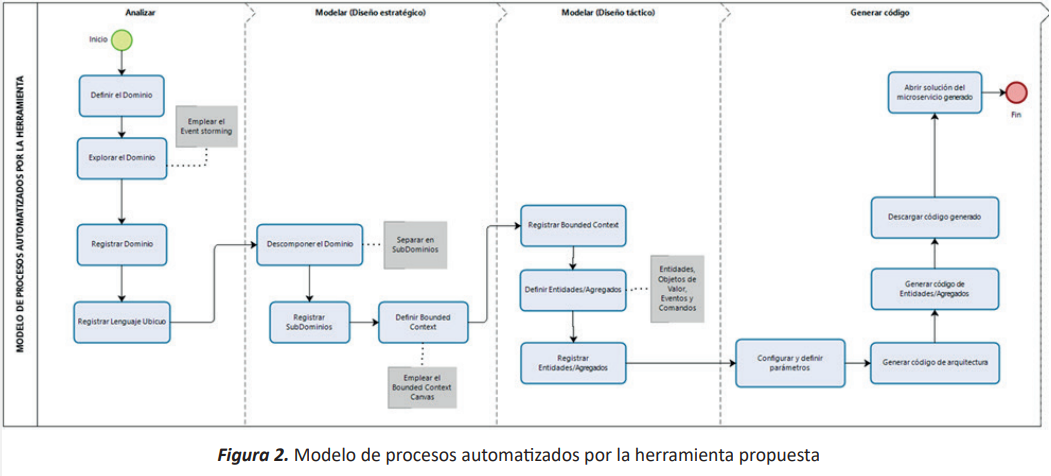
Ventajas.

* 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.

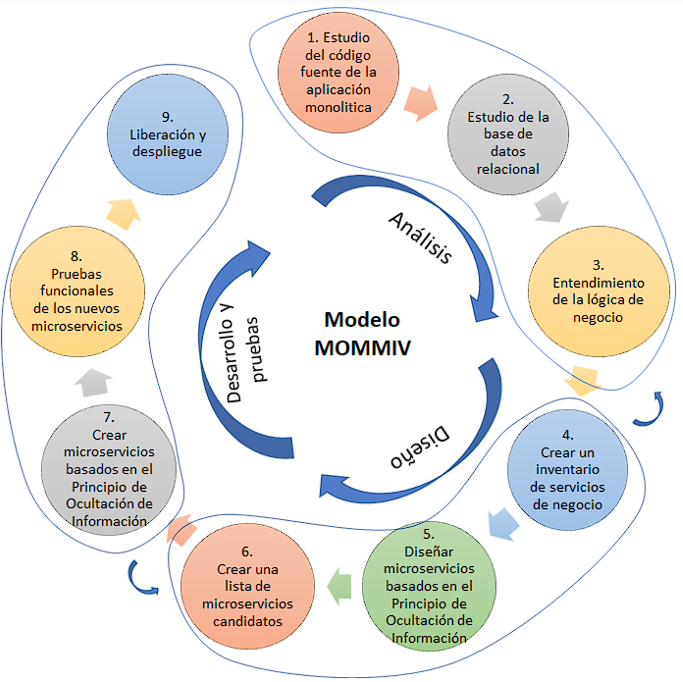
**Herramienta para el modelado y generación de código de Arquitecturas de Software basadas en Microservicios y Diseño guiado por el dominio (DDD)**

****

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.

**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.

**Buenas prácticas en la construcción de software**

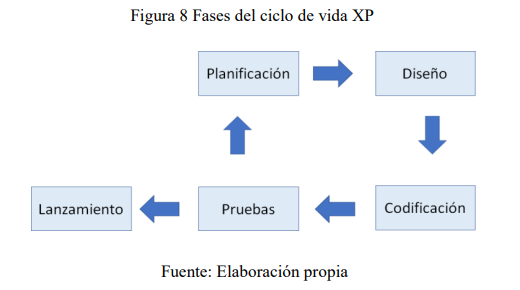
**Diagrama, Dibujo de ingeniería

Descripción generada automáticamente**Nos habla de las buenas practicas que debemos seguir al desarrollar un software, dándonos a conocer de malas practicas 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.

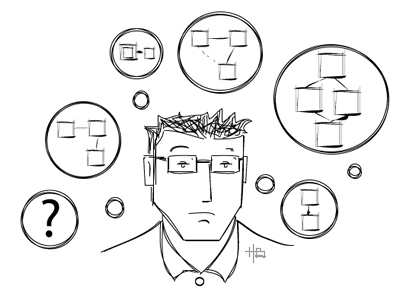
* copy and paste programming
* reinventing the Wheel

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:
   1. Comentarios legales sobre derechos de autor.
   2. Comentarios informativos cuando no es clara una funcionalidad.
   3. Comentarios sobre advertencias.
   4. Comentarios sobre utilización de librerías o recursos de terceros que no sean modificables
6. Formato:
   1. Un archivo de código no debe sobrepasar las 500 líneas y cada línea no debe sobrepasar 120 caracteres.
   2. 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.
   3. 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.
   1. Acoplamiento: Tener un acoplamiento bajo (nos dice que las clases relacionadas no debe conocer muchos detalles de otra).
   2. Cohesión: Es la independencia de cada clase.
9. Arquitectura: Diseñar el sistema que cubrirá los requerimientos funcionales y no funcionales del software.
   1. Arquitectura Empresarial: Definir correctamente los componentes y activos de una empresa para entender la organización.
   2. Arquitectura Solución: Definir el diseño y comunicaciones de alto nivel del software para guiar el diseño y desarrollo a satisfacer las necesidades.
   3. 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.
    1. Capas: Este divide la aplicación en capas cada una de estas con un rol definido por ejemplo presentación, servicios etc.
    2. Monolítico: Aplicación autosuficiente, agrupando todo en el mismo proyecto.
    3. Microservicios: Dividir el software en pequeños componentes con una única responsabilidad Permitiendo al software ser independiente de cada una de sus partes.
    4. Event-driven architecture: Esta arquitectura no espera respuestas inmediatas, sino que espera a un evento y reacciona a una respuesta.
    5. 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.
    1. Patrones creacionales: “Estos patrones intentan controlar la forma en que los objetos son creados, implementando mecanismos que eviten la creación directa de objetos.”
    2. Patrones estructurales: Estos patrones se encargan de diseñar la comunicación que tienen las clases.
    3. 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.
    1. Cliente: Proporciona los requerimientos y las principales necesidades.
    2. Desarrolladores: Aportar en la construcción y objetivos planteados.
    3. Testers: se encarga de comunicarse con el cliente y definir los estándares de calidad.
    4. Tracker: Realiza el seguimiento de las tareas propuestas y la comunicación con el cliente.
    5. Coach: Orienta al equipo de trabajo y cliente para que todo se haga de manera correcta.
    6. Manager: “El mánager se encarga de coordinar la comunicación entre distintas partes, gestionando los recursos necesarios.”
14. Ciclo de vida metodología XP:
    1. Fase de planificación: Entendimiento de los requisitos del cliente, con cada interacción generar una versión del software nueva y funcional.
    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: Desarrollado principalmente en pares para asegurar el entendimiento del software.
    4. Fase de pruebas: Realización de pruebas automáticas constantemente.
    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.”



**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.

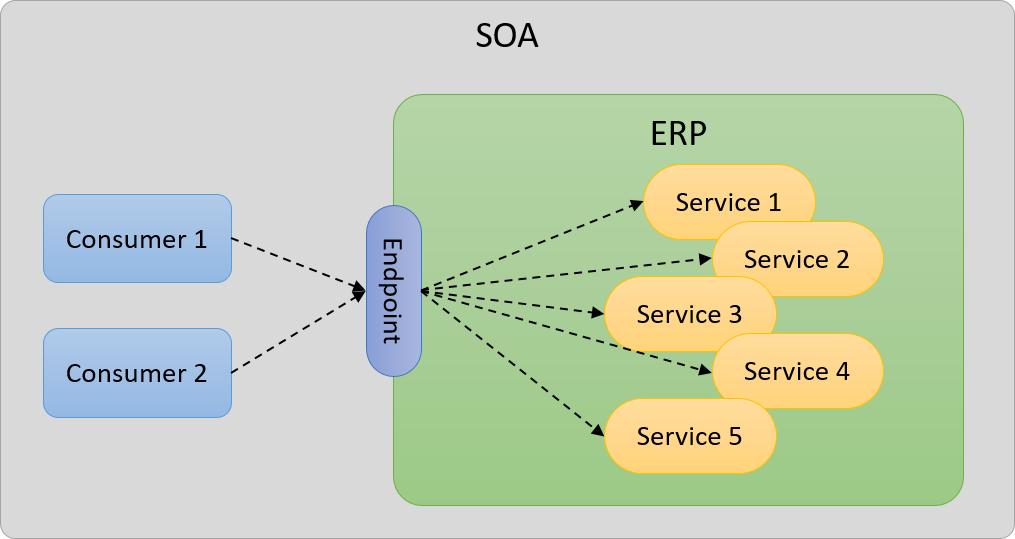
Ingeniería de Software: Se preocupa por los métodos y principios de diseño necesarios para el proceso de construcción del software.

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.

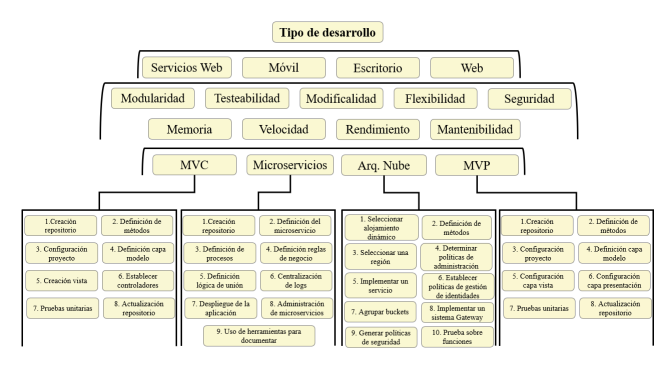
Lenguajes de patrones: Son agrupaciones de patrones de diseño para resolver problemas complejos y recurrentes

**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 mas 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.

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

**** **Clean architecture para mejorarG**

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

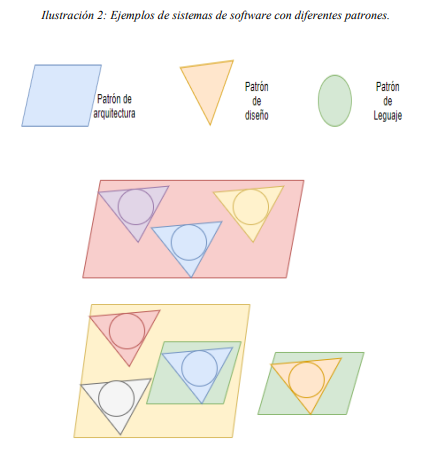
MVP: Dispositivos móviles y Aplicaciones web

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

Unos pasos generales del modelo son:

1. Selecciona el medio al que esta 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 como aplicar, esta le dará pasos que incluyen buenas prácticas y como se podría desarrollar ese patrón sin especificar lenguajes o tecnologías

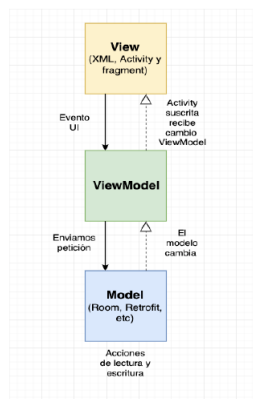
**Especificando una arquitectura de software**

****

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:
   1. 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.
   2. Intuición: Habilidad de concebir sin razonamiento consciente.
   3. 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:
   1. Patrones de arquitectura: Afectan la estructura global del sistema
   2. Patrones de diseño: Afecta a subsistemas o componentes del sistema global y sus relaciones.
   3. 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
   1. 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.
   2. Sistemas distribuidos: Se centra en la comunicación que tiene el sistema
   3. Sistemas interactivos: Se centra en mejorar la usabilidad de la aplicación separando la interfaz de usuario del núcleo de la lógica.
   4. 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.

**Análisis comparativo de Patrones de Diseño de Software**

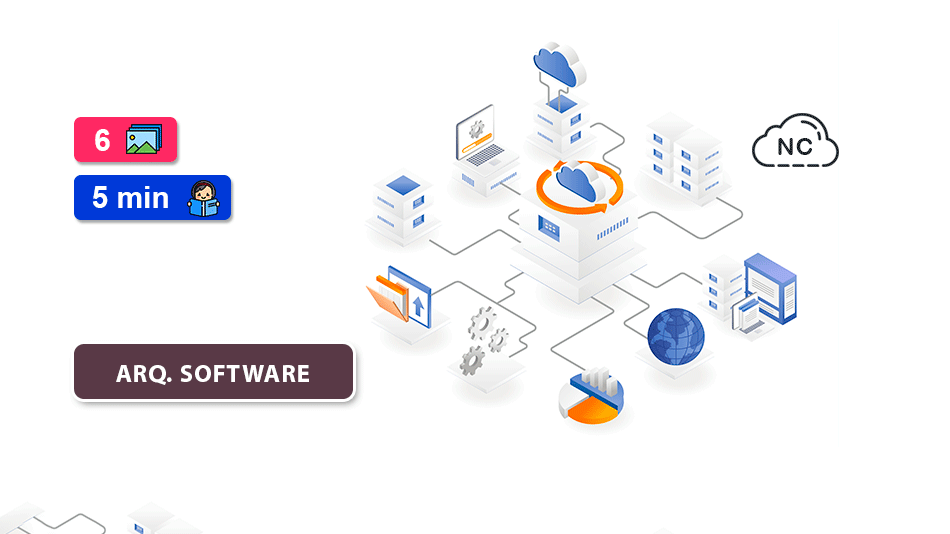
****

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.
   1. Ventajas
      1. Evita duplicación de código
      2. Es fácil de entender e implementar
      3. Flexible
   2. Desventajas
      1. Seguir el flujo es complejo a la hora de depurar
      2. Poca mantenibilidad
2. Model-View-Controller MVC: Separado en tres componentes Modelo, vista, controlador.
   1. Estructura:
      1. Modelo: Se encarga de manipular y gestionar los datos.
      2. Vista: Se encarga de mostrar las pantallas al usuario.
      3. Controlador: Se encarga de procesar las instrucciones recibidas y comunicar a la vista y al modelo.
   2. Ventajas:
      1. Modularidad.
      2. Mantenibilidad.
   3. Desventajas:
      1. Gran crecimiento del programa
      2. 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.
   1. Estructura:
      1. Modelo: Contiene los datos y la lógica, oculta su implementación con interfaces.
      2. Vista: Se encarga de mostrar las pantallas al usuario y recibir sus peticiones.
      3. Presentador: Recibe las peticiones de la vista e invoca métodos del modelo, obteniendo un resultado y actualiza la vista.
   2. Ventajas:
      1. Flexibilidad
      2. Permite la integración de distintas tecnologías
      3. La vista y el modelo pueden ser testeados de manera independiente.
   3. Desventajas:
      1. Implementación Compleja.
      2. 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.
   1. Estructura:
      1. Controller: Procesa todos y gestionan las peticiones del sistema.
      2. Dispatcher: Se encarga de mostrar las pantallas al usuario y navegar entre vistas.
      3. Helper: ayuda al controlador y a la vista a terminar su procesamiento.
   2. Ventajas
      1. Existe una mejora significativa en la manejabilidad de la seguridad.
      2. Es posible reutilizar código.
      3. Evita tener distribuida la gestión de peticiones.
      4. Control centralizado
      5. Seguridad de subprocesos
   3. Desventajas
      1. Poca escalabilidad.
      2. 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.
   1. Estructura:
      1. Modelo: Es el responsable del acceso a la fuente de datos y de trabajar con esos datos.
      2. Vista: Se encarga de mostrar las pantallas al usuario y recibir sus peticiones.
      3. Vista del modelo: Encargada me transformar los datos en un formato que la vista lo requiera.
   2. Ventajas
      1. Facilita el mantenimiento
      2. Disminuye la cantidad de código
      3. Facilidad en pruebas unitarias
   3. Desventajas:
      1. Curva de aprendizaje alta
      2. Depuración compleja

**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.

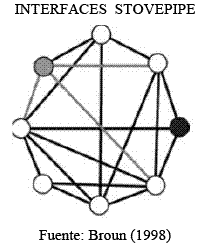
7. Rol: función desempeñada por un actor.

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.

**Modelo Teórico para la Identificacion del Antipatron “Stovepipe System” en la Etapa de la Implementacion 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

i. Anti-patrones de desarrollo de software

ii. Stovepipe System

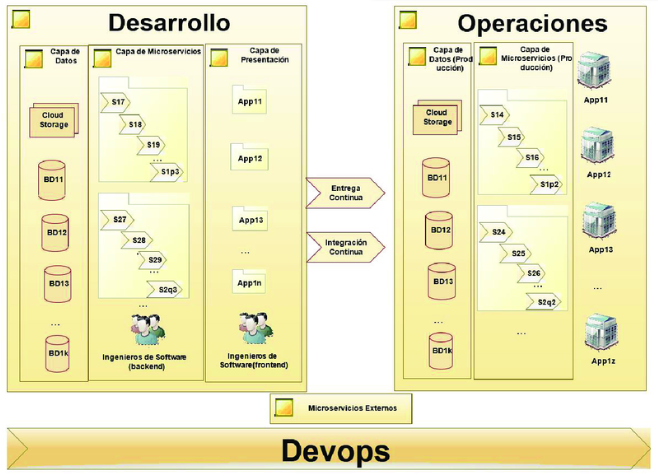
iii. Redes neuronales

iv. Aprendizaje automático

vi. Calidad del software

vii. Arquitectura del sistema

**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

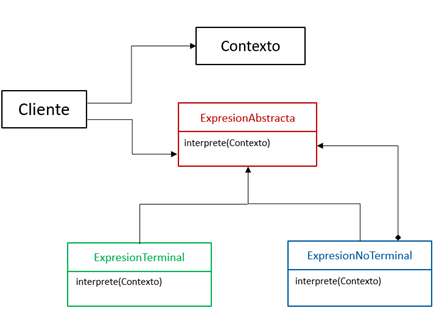
1. Mejora la productividad del equipo de desarrollo.

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**.**

**Marco de trabajo para la selección de la arquitectura de un proyecto de software mediante la aplicación de patrones arquitectónicos**



La investigación sobre patrones de software busca facilitar la selección y aplicación de patrones arquitectónicos adecuados para proyectos de desarrollo, considerando arquitecturas de información, navegación, interacción y visualización. Esto permite a los desarrolladores optimizar la experiencia del usuario y tomar decisiones informadas para asegurar el éxito del proyecto.

Objetivos de la investigación

1. Facilitar la selección de patrones arquitectónicos adecuados.

2. Optimizar la experiencia del usuario.

3. Mejorar la estructura y presentación de la información.

4. Alinear la arquitectura con los objetivos funcionales y técnicos del proyecto.

Beneficios de la investigación

1. Identificar la mejor estructura para organizar y presentar la información.

2. Optimizar la navegación y interacción.

3. Asegurar una presentación clara y coherente de los datos.

4. Tomar decisiones informadas para el desarrollo de la aplicación.

Palabras clave

i. Patrones de software

ii. Arquitecturas de información

iii. Navegación

iv. Interacción

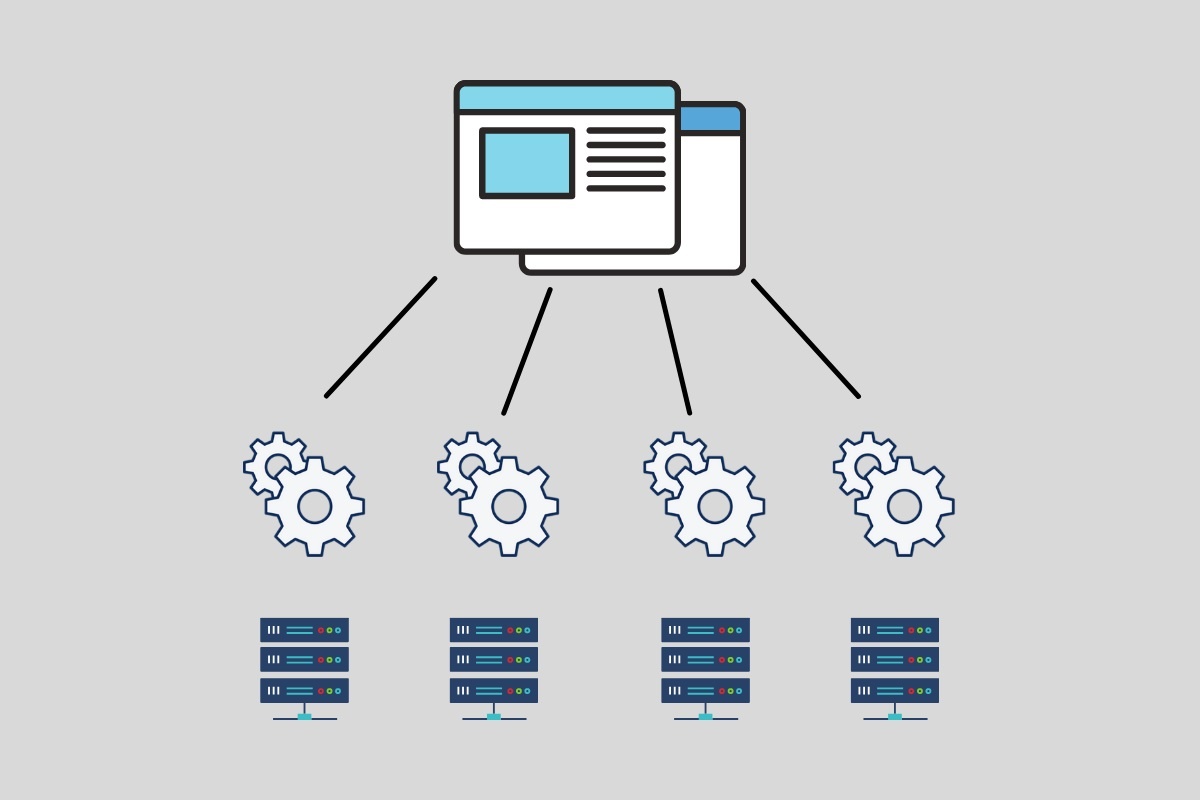
v. Visualización

vi. Experiencia del usuario

vii. Desarrollo de aplicaciones

viii. Arquitectura de software

**Implementación de una arquitectura de software orientada a microservicios en la Dirección de Tecnología de una Institución Universitaria**



El estudio analiza la implementación de una arquitectura de microservicios en una institución universitaria, destacando sus ventajas en escalabilidad, flexibilidad y mantenimiento. También explora tecnologías como micro-frontends y testing automático para mejorar la experiencia del usuario y garantizar la calidad del software.

Objetivos del estudio

1. Definir e implementar una arquitectura de microservicios.

2. Mejorar la escalabilidad, flexibilidad y mantenimiento del sistema.

3. Integrar tecnologías para mejorar la experiencia del usuario.

4. Garantizar la calidad del software mediante testing automático.

ventajas de la arquitectura de microservicios

1. Escalabilidad

2. Flexibilidad

3. Mantenimiento más sencillo

4. Desarrollo y implementación continua

Tecnologías exploradas

1. Micro-frontends: integración modular de interfaces

2. Testing automático: garantiza la calidad del software

3. Desarrollo y implementación continua

Beneficios

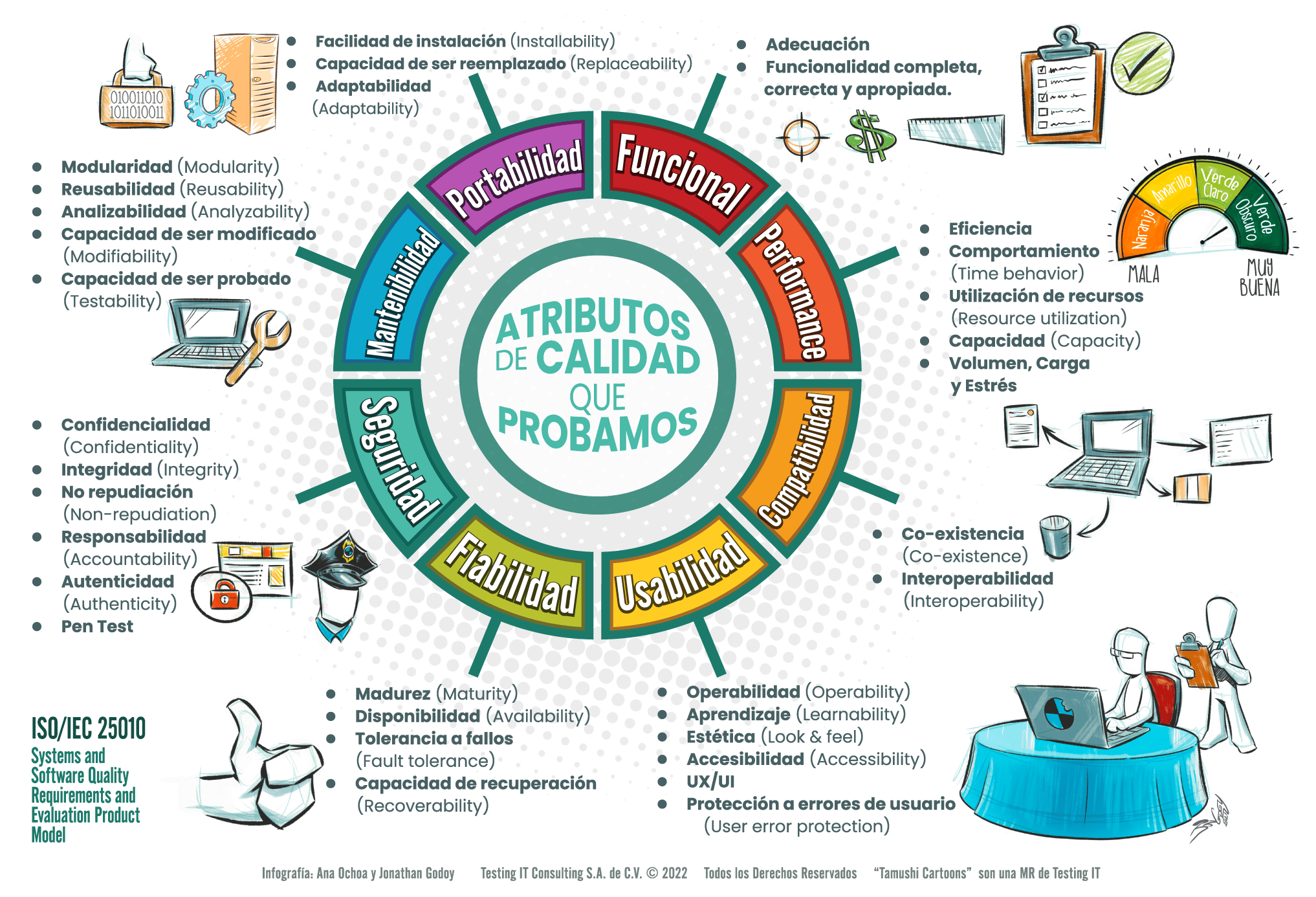
1. Mejora la experiencia del usuario

2. Aumenta la eficiencia y capacidad de respuesta del sistema

3. Reduce errores y acelera el ciclo de desarrollo

4. Se adapta a las necesidades dinámicas de la institución universitaria

**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.

**Clean architecture para mejorar el desarrollo de aplicaciones móviles en la empresa GMD**



El informe describe la implementación de Clean Architecture en la empresa GMD para mejorar el desarrollo de aplicaciones móviles. La arquitectura tradicional MVC y MVP no eran suficientes para manejar la complejidad creciente de los aplicativos. Clean Architecture, propuesta por Robert Martin, permite independencia de frameworks, facilita pruebas, y separa la lógica de negocio de la interfaz gráfica y la base de datos.

Objetivos

1. Mejorar el desarrollo de aplicaciones móviles.

2. Reducir la complejidad y el costo del mantenimiento.

3. Facilitar las pruebas y mejoras.

Metodología

1. Análisis de la situación actual.

2. Implementación de Clean Architecture.

3. Evaluación económica.

Resultados

1. Mejor distribución de software.

2. Reducción del costo del mantenimiento.

3. Facilidad para realizar pruebas unitarias y modulares.

4. Mejoras enfocadas a los casos de uso

Beneficios

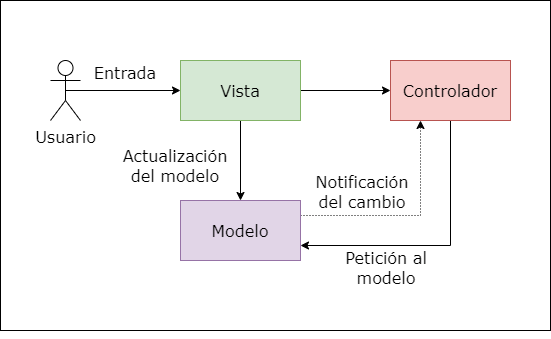
1. Independencia de frameworks.

2. Facilidad para cambiar de origen de datos.

3. Rediseño de interfaces sin afectar la lógica de negocio.

4. Reducción de costos a largo plazo.

**Arquitectura de software flexible y genérica para métodos del tipo Newton**



el análisis y diseño de una arquitectura orientada a objetos para resolver el problema de reúso en métodos del tipo Newton. La arquitectura propuesta abstrae las características principales de estos métodos utilizando patrones de arquitectura y diseño, permitiendo su uso en diferentes problemas no lineales.

Objetivos

1. Resolver el problema de reúso en métodos del tipo Newton.

2. Diseñar una arquitectura orientada a objetos para métodos no lineales.

Metodología

1. Análisis de las características principales de los métodos del tipo Newton.

2. Diseño de la arquitectura utilizando patrones de arquitectura y diseño.

3. Evaluación de la extensibilidad y flexibilidad utilizando métricas de abstracción e inestabilidad. Beneficios

1. Reúso de código en métodos del tipo Newton.

2. Interfaz genérica entre distintos métodos tipo Newton.

3. Mejora en la eficiencia y productividad en el desarrollo de software.

Palabras clave

i. Arquitectura orientada a objetos

ii. Métodos del tipo Newton

iii. Patrones de arquitectura y diseño

iv. Reúso de código

v. Extensibilidad y flexibilidad

vi. Métricas de abstracción e inestabilidad

**Modelo para la ayuda a la toma de decisiones en la selección de patrones de desarrollo de software.**



un modelo basado en técnicas de inteligencia artificial para ayudar a los desarrolladores, diseñadores y arquitectos de software a tomar decisiones sobre la aplicación de patrones de diseño en situaciones específicas. El modelo caracteriza los patrones según sus características y proporciona algoritmos para realizar el proceso de toma de decisiones.

Objetivos

1. Crear un modelo que ayude a los desarrolladores a aplicar patrones de diseño.

2. Investigar la aplicación de patrones de diseño en el desarrollo de arquitectura de software.

3. Desarrollar algoritmos para la toma de decisiones.

Problema

1. Falta de conocimientos sobre patrones de diseño entre desarrolladores.

2. Inmadurez en la aplicación de patrones de diseño.

Metodología

1. Investigación sobre patrones de diseño y su aplicación.

2. Desarrollo de un modelo basado en inteligencia artificial.

3. Creación de algoritmos para la toma de decisiones.

Beneficios

1. Mejora en la aplicación de patrones de diseño.

2. Reducción de errores en el desarrollo de software.

3. Aumento de la eficiencia y productividad.

Palabras clave

i. Patrones de diseño

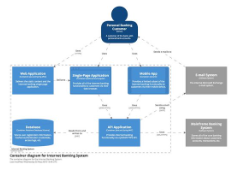
ii. Inteligencia artificial

iii. Toma de decisiones

iv. Arquitectura de software

v. Desarrollo de software

**Modelo de arquitectura de software para el procesamiento de datos en arquitecturas actuales**



La elección de una arquitectura de software adecuada es crucial para garantizar el éxito de una aplicación. La arquitectura debe considerar las necesidades particulares de cada proyecto y no solo las fortalezas del equipo de trabajo. Se propone un prototipo arquitectural para manejar datos masivos, basado en la lógica del negocio, infraestructura y herramientas existentes.

Objetivos

1. Proponer una arquitectura de software adecuada para manejar datos masivos.

2. Garantizar el éxito de la aplicación.

3. Considerar las necesidades particulares de cada proyecto.

Problema

1. La mala elección de una arquitectura puede generar obstáculos.

2. El manejo de datos masivos es un desafío para muchas arquitecturas.

Factores clave

1. Análisis de las necesidades particulares del proyecto.

2. Fortalezas del equipo de trabajo.

3. Lógica del negocio.

4. Infraestructura.

5. Herramientas existentes.

Beneficios

1. Mejora en la eficiencia y eficacia del sistema.

2. Solución efectiva para el manejo de datos masivos.

3. Guía comparativa para tomar decisiones.

Palabras clave

i. Arquitectura de software

ii. Manejo de datos masivos

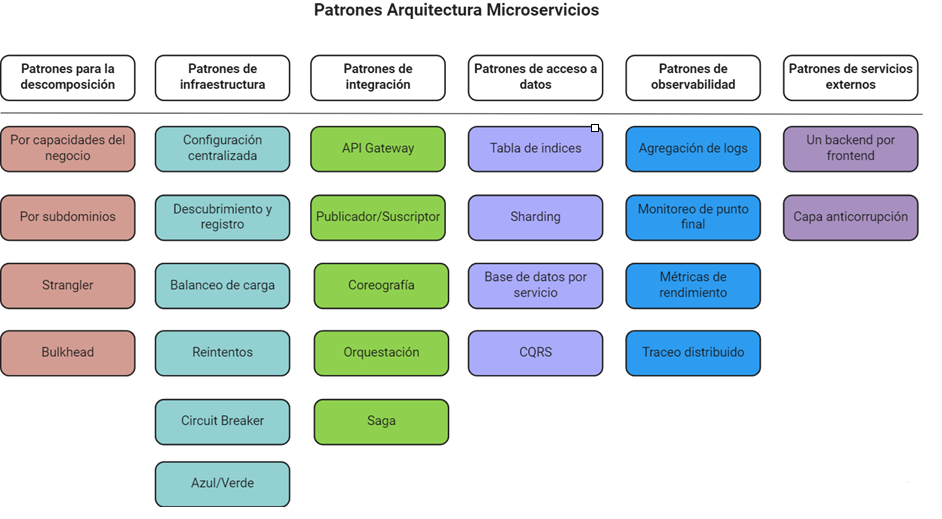
iii. Lógica del negocio

iv. Infraestructura

v. Herramientas existentes

vi. Prototipo arquitectural

**Adaptación de un patrón de software en seguridad a la arquitectura de un Microservicio**



Este trabajo investiga los riesgos de seguridad asociados con la arquitectura de microservicios y propone un patrón de seguridad llamado "Microservice Security Pattern API Gateway" (MSPAG) para fortalecer la seguridad en sistemas basados en microservicios. El patrón utiliza JSON Web Tokens (JWT) y criptografía H256 para proteger los puntos de acceso de los microservicios.

Objetivos

1. Investigar los riesgos de seguridad en la arquitectura de microservicios.

2. Proponer un patrón de seguridad efectivo para microservicios.

3. Fortalecer la seguridad en sistemas basados en microservicios.

problema

1. La naturaleza descentralizada de los microservicios aumenta la superficie de ataque.

2. Los ataques en sistemas basados en microservicios pueden tener consecuencias graves.

Ventajas de los microservicios

1. Modularidad

2. Escalabilidad independiente

3. Entrega continua

4. Flexibilidad tecnológica

Desafíos de seguridad

1. Accesos no autorizados

2. Superficie de ataque ampliada

3. Complejidad en la seguridad

Palabras clave

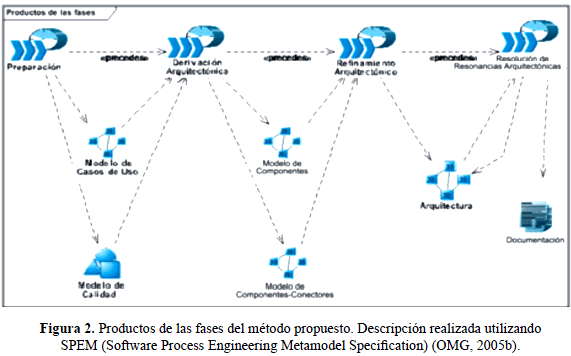
i. Patrón de seguridad

ii. JSON Web Tokens (JWT)

iii. Criptografía H256

iv. Desarrollo de software

**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

i. Arquitectura de software

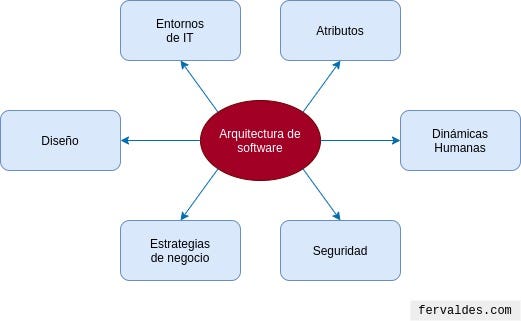
ii. Calidad de software

iii. Métodos de diseño arquitectónico

iv. Métodos de evaluación arquitectónica

v. Características de calidad

**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.

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

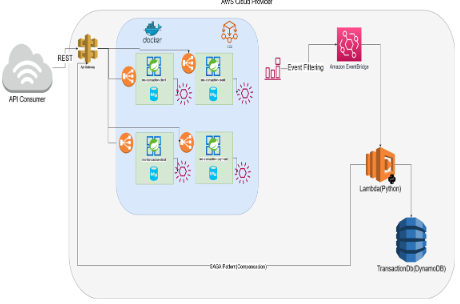
i. Representación de conocimiento

ii. Lenguajes de descripción arquitectónica (ADL)

iii. Componente

iv. Conector

**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.

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

i. Arquitectura de microservicios

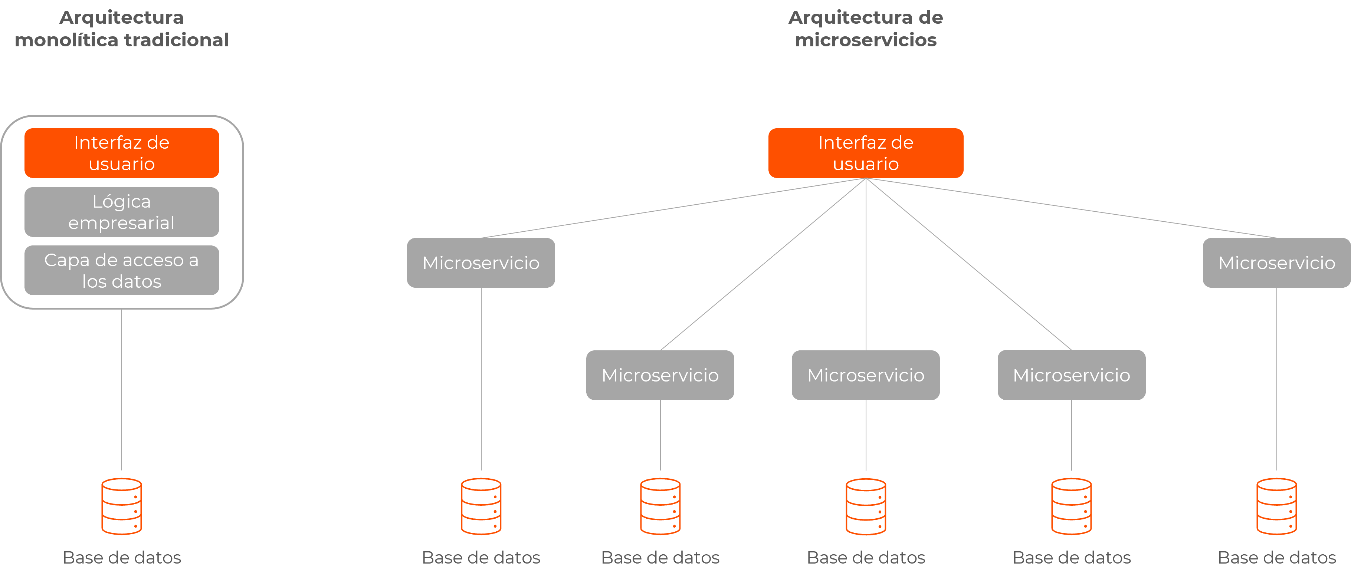
ii. Transacciones distribuidas

iii. Patrones de diseño (SAGA, Event Sourcing)

iv. Amazon Web Services (AWS)

v. Spring

**Arquitectura basada en micro-servicios para aplicaciones web**



Este artículo presenta una visión general de la arquitectura de microservicios, sus características, ventajas y desventajas. También se comparan las arquitecturas monolíticas y de microservicios, y se explora la comunicación entre microservicios.

Objetivos

1. Presentar la arquitectura de microservicios como una opción para el desarrollo de aplicaciones web.

2. Analizar las características, ventajas y desventajas de la arquitectura de microservicios.

3. Comparar las arquitecturas monolíticas y de microservicios.

Conceptos clave

1. Arquitectura de microservicios: enfoque de desarrollo de aplicaciones como un conjunto de servicios pequeños e independientes.

2. Características: escalabilidad, funcionalidad modular, libertad del desarrollador, uso de contenedores.

3. Ventajas: equipo de trabajo mínimo, escalabilidad, funcionalidad modular.

4. Desventajas: alto consumo de memoria, complejidad de gestión, pruebas complicadas.

Palabras clave

i. Arquitectura de microservicios

ii. Desarrollo de aplicaciones web

iii. Escalabilidad

iv. Flexibilidad

v. Comunicación entre microservicios

vi. Modelado de microservicios

**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, la gestión de la seguridad

Tendencias Clave

- Modularidad

- Seguridad

- Escalabilidad

- Flexibilidad

Desafíos Principales

- Coordinación y orquestación de módulos

- Gestión de seguridad a nivel de sistema

- Implementación de DevOps y automatización de pruebas

Ventajas

i. Escalabilidad

ii. Mantenimiento

iii. Flexibilidad

**A scalable architecture for automated monitoring of microservices**



El artículo presenta una arquitectura destinada al monitoreo de microservicios,

enfocándose en la supervisión de las métricas clave del rendimiento del sistema. La

propuesta se basa en el uso de sistemas de monitoreo avanzados como Prometheus,

OpenTSDB y Elastic APM para capturar y analizar datos críticos en tiempo real.

Prometheus se encarga de la recolección y consulta de métricas, mientras que OpenTSDB

se utiliza para almacenar series temporales de datos, y Elastic APM proporciona

capacidades de monitoreo y análisis de rendimiento de aplicaciones. La combinación de

estas herramientas permiten una visibilidad integral sobre el funcionamiento de los

microservicios, facilitando la identificación de cuellos de botella y problemas de

rendimiento.

Herramientas Utilizadas:

- Prometheus (recolección y consulta de métricas)

- OpenTSDB (almacenamiento de series temporales de datos)

- Elastic APM (monitoreo y análisis de rendimiento de aplicaciones)

Beneficios:

i. Visibilidad integral sobre el funcionamiento de los microservicios

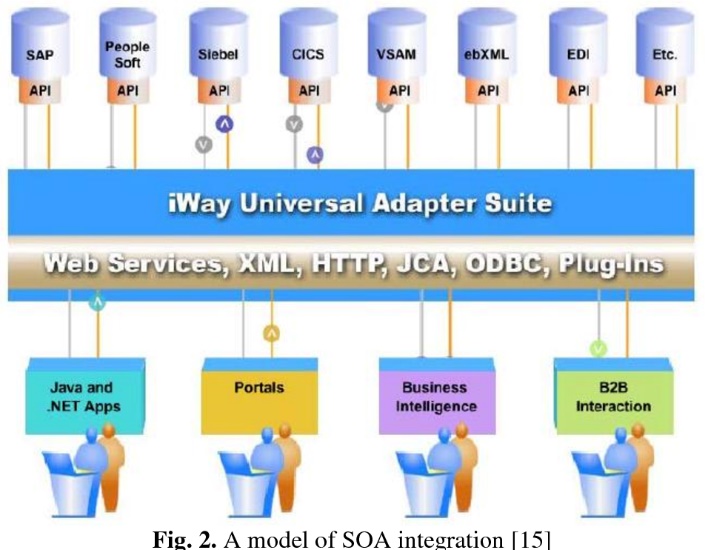
ii. Identificación de cuellos de botella y problemas de rendimiento

iii. Optimización del seguimiento de la salud y eficiencia del sistema

iv. Mejora de la capacidad de respuesta ante posibles incidencias

v. Rendimiento óptimo en entornos distribuidos

**Software Architectures - Present and Visions**



La arquitectura de software es un componente fundamental en el desarrollo de sistemas informáticos. En este artículo, se analiza el estado actual de las arquitecturas de software y se exploran las tendencias y visiones futuras

Estado Actual:

1 arquitecturas basadas en microservicios para mayor flexibilidad y escalabilidad

2 integración con Inteligencia Artificial (IA) y Machine Learning (ML) para mejorar la toma de decisiones

3 desarrollo de aplicaciones escalables y seguras para satisfacer las necesidades de los usuarios

4 importancia de la arquitectura de datos y la gestión de la complejidad para asegurar la eficiencia

Tendencias Futuras:

a. Arquitecturas serverless para reducir costos y mejorar la eficiencia

b. Computación en la nube y edge computing para mejorar la velocidad y la escalabilidad

c. Seguridad y privacidad de datos para proteger la información confidencial

d. Desarrollo de aplicaciones autónomas y adaptativas para mejorar la experiencia del usuario

Desafíos:

i. Gestión de la complejidad y la escalabilidad

ii. Integración de tecnologías emergentes

iii. Seguridad y privacidad de datos

iv. Desarrollo de habilidades y talento en arquitectura de software

**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:

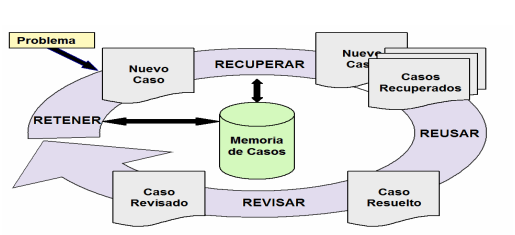
i. Desarrollo más eficiente

ii. Adaptabilidad a cambios frecuentes

iii. Calidad y coherencia del software

iv. Mejora la colaboración entre equipos

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.