Arquitectura de software

1. Introducción:

La arquitectura de software surge como una disciplina clave dentro del desarrollo de sistemas, permitiendo gestionar la complejidad inherente de los proyectos a través de un enfoque estructurado y organizado. En sus inicios, el diseño del sistema no se concebía como una etapa separada de la programación. Fue a partir de la década de 1970 cuando se empezó a explorar y aplicar la idea de descomponer programas en módulos, marcando un hito en la evolución del desarrollo de software.

A comienzos de la década de los 90, expertos como Mary Shaw y David Garlan sentaron los cimientos de lo que actualmente entendemos como arquitectura de software, destacando un grado de abstracción que sobrepasaba al del diseño convencional. Esta rama no solo se enfoca en las estructuras de los sistemas y sus interrelaciones, sino también en asegurar que estos posean características de calidad como desempeño, escalabilidad y seguridad.

2. Objetivos:

- Diseñar un marco abstracto y coherente que describa los componentes, sus interfaces y la comunicación entre ellos, asegurando claridad en el funcionamiento.
- Satisfacer los objetivos del sistema, tanto funcionales como no funcionales (mantenibilidad, flexibilidad, seguridad, auditoría, etc.), considerando las limitaciones tecnológicas disponibles.
- Proporcionar un diseño que permita la implementación efectiva del software en una arquitectura física, asignando tareas a componentes específicos.
- Diseñar arquitecturas óptimas según las tecnologías empleadas, evitando incompatibilidades, por ejemplo, asegurando que las arquitecturas sean adecuadas para sistemas en tiempo real o de múltiples capas.

- Crear sistemas capaces de crecer, adaptarse y comunicarse con otros, garantizando integración e interoperabilidad.
- Simplificar el diseño global para abordar problemáticas modernas, minimizar riesgos en costos, tiempos y calidad, y facilitar el mantenimiento a lo largo del ciclo de vida del sistema.

3. Arquitectura de software:

La arquitectura de software es el más avanzado en el desarrollo de sistemas. Ilustra la configuración global, las conexiones entre los elementos y la interacción de estos con su ambiente. Se parece a los planos de un inmueble, dado que establece cómo los componentes del software cooperan para alcanzar las metas funcionales y no funcionales del sistema. Consiste en un conjunto de patrones, abstracciones y decisiones que proporcionan un marco claro para el desarrollo y mantenimiento de sistemas complejos. Además, establece las bases para que el software sea escalable, flexible, mantenible y capaz de integrarse con otras tecnologías.

La arquitectura de software alcanza metas concretas, como, por ejemplo:

- Ofrecer una perspectiva integral del sistema.
- Disminuir la complejidad del progreso.
- Asegurar la satisfacción de requerimientos tanto funcionales como no funcionales.
- Conformarse a las tecnologías y limitaciones existentes.

Hay diversas formas de arquitectura que se emplean en función del contexto y las demandas del proyecto, tales como la arquitectura en capas, la arquitectura hexagonal, la arquitectura fundamentada en microservicios, entre otras. Cada una presenta beneficios y retos específicos, y su selección se basa en elementos como la magnitud del sistema, las metas y las limitaciones tecnológicas.

La arquitectura de software como método organizado no solo simplifica la implementación, sino también la administración durante todo el ciclo de vida del software, posibilitando a los programadores prever problemas y elaborar soluciones eficaces desde los primeros pasos.

3.1 Proceso de la arquitectura de software:

El proceso de arquitectura de software abarca una serie de acciones y decisiones sistemáticas que facilitan el diseño, la implementación y la administración de la estructura de un sistema

de software. Este procedimiento busca asegurar que la arquitectura satisfaga las necesidades funcionales y no funcionales, al mismo tiempo que se ajusta a las restricciones tecnológicas y organizativas. A continuación, se describen las fases fundamentales de este procedimiento:

3.1.1 Recopilación y Evaluación de Requisitos

Especificación: Se reconocen y registran los propósitos tanto funcionales como no funcionales del sistema. Este procedimiento requiere la implicación de los interesados para garantizar que las necesidades empresariales y las expectativas del usuario estén claramente establecidas.

Rendimiento: Una serie de necesidades prioritarias, que incluyen limitaciones tecnológicas, escalabilidad, desempeño y seguridad.

3.1.2 Definición de la Arquitectura Conceptual

Especificación: Se diseñan los componentes principales y sus interacciones, creando una vista abstracta del sistema. Aquí se seleccionan los estilos arquitectónicos apropiados (por ejemplo, arquitectura en capas, microservicios, hexagonal, etc.).

Resultado: Una representación inicial que describe cómo se estructurará el sistema y cómo se comunicará con su entorno.

3.1.3 Elección de Tecnologías para Tecnologías

Especificación: Se seleccionan las tecnologías y recursos más idóneos para poner en práctica la arquitectura conceptual. Esto abarca lenguajes de programación, marcos de referencia, bases de datos y servicios de almacenamiento en la nube.

Rendimiento: Un plan tecnológico que garantice la factibilidad y la compatibilidad del diseño arquitectónico con los instrumentos existentes.

3.1.4Diseño Específico de Elementos

Especificación: Cada elemento se detalla en profundidad, estableciendo su funcionalidad, interfaces, dependencias y contratos. Este diseño minucioso garantiza que los programadores tengan claro su proceso de implementación.

Rendimiento: Descripción técnica exhaustiva para cada elemento del sistema.

3.1.5 Validación y Evaluación de la Arquitectura

Especificación: Se realizan revisiones para verificar que la arquitectura cumple con los requisitos definidos y que es técnicamente factible. Esto incluye pruebas de prototipos y simulaciones para identificar posibles riesgos.

Resultado: Un diseño arquitectónico validado, con ajustes realizados según los hallazgos.

3.1.6 Documentación de la Arquitectura

Especificación: El diseño arquitectónico se registra en vistas concretas (vista lógica, vista de evolución, vista de procesos, vista física, etc.) de acuerdo con el modelo 4+1 o sustituto. Esto permite entender y conservar el sistema durante todo su ciclo de vida.

Rendimiento: Documentos de arquitectura asequibles y bien estructurados.

3.1.7 Ejecución y Control

Especificación: En el proceso de puesta en marcha del sistema, la arquitectura orienta a los equipos de desarrollo para asegurar la consistencia con el diseño previamente definido. Además, se monitorea el progreso del sistema para ajustarlo a modificaciones en las necesidades o el ambiente tecnológico.

Resultado: Un sistema operativo en consonancia con la arquitectura proyectada y flexible ante futuras exigencias.

3.2. Modelos y vistas arquitecturales

Los modelos y perspectivas arquitecturales son esenciales en la arquitectura del software, pues facilitan la representación de diversas visiones del sistema para tratar su diseño y comunicación de forma eficaz. Estas visiones son imprescindibles para cubrir las demandas de los diferentes interesados implicados en el desarrollo y conservación del software.

El Modelo de Vistas de Arquitectura 4+1, sugerido por Philippe Kruchten, es uno de los métodos más famosos para estructurar estas vistas. Este modelo expone el sistema desde cinco puntos de vista adicionales: cuatro enfoques fundamentales para los aspectos técnicos y una perspectiva extra para los casos de uso.

3.2.1 Vista Lógica

La visión lógica ilustra la estructura operativa del sistema desde el punto de vista del programador. En esta perspectiva, se reconocen los elementos fundamentales, tales como clases, módulos y las conexiones entre estos. Este método es crucial para entender el funcionamiento global del sistema y la disposición de sus componentes para alcanzar las metas propuestas.

3.2.2 Vista de Desarrollo

Desde el punto de vista de los equipos de desarrollo, la vista de desarrollo ilustra la disposición del código fuente en paquetes, bibliotecas y módulos, incluyendo también elementos vinculados con la gestión de versiones y las herramientas del ambiente de desarrollo. Este nivel de minuciosidad facilita la administración eficaz de las dependencias del sistema y asegura que el diseño arquitectónico sea consistente con la implementación.

3.2.3 Vista de Procesos

El enfoque de la vista de procesos es la interacción entre los elementos del sistema durante su ejecución. Establece procedimientos, actividades y los sistemas de comunicación entre ellos, tales como concurrencia y sincronización. Esta perspectiva es esencial para mejorar elementos como el desempeño, la escalabilidad y la habilidad del sistema para responder a demandas fluctuantes.

3.3.3 Vista Física

La perspectiva física refleja la ubicación de los elementos del sistema en la infraestructura de hardware. Incorpora servidores, redes, bases de datos y otros elementos físicos requeridos para el funcionamiento del sistema. Esta visión simplifica la organización de la infraestructura y garantiza que el diseño arquitectónico se ajuste a las necesidades y limitaciones del ambiente físico.

3.3.4 Vista de Casos de Uso (4+1)

La vista de casos de uso integra las perspectivas anteriores mediante escenarios específicos que ilustran cómo el sistema satisface los requisitos funcionales. Describe flujos de interacción entre los usuarios y el sistema, proporcionando una representación comprensible para los stakeholders no técnicos, como clientes o usuarios finales.

3.3.5 Ventajas de los Modelos y Vistas Arquitecturales

La utilización de modelos y perspectivas arquitecturales brinda varias ventajas. Promueven el entendimiento del sistema desde distintos puntos de vista, posibilitando una cooperación eficaz entre equipos técnicos y no especializados. Además, garantizan que todos los elementos esenciales de la arquitectura, desde la funcionalidad hasta la infraestructura física,

sean tenidos en cuenta y tratados de forma holística, potenciando la calidad y la sostenibilidad del sistema durante todo su ciclo de vida.

3.3. Diseño basado en el dominio (DDD)

El Diseño Dirigido por el Dominio (Domain-Driven Design, DDD) es una metodología enfocada en la creación de software que se sincroniza de manera profunda con los procesos, normas y metas empresariales. Esta metodología, propuesta por Eric Evans, sostiene que el desarrollo de software debe fundamentarse en un entendimiento profundo del dominio empresarial y en una cooperación constante entre los especialistas en el campo y los programadores.

En DDD, el dominio se refiere al área concreta de conocimiento o actividad que el software necesita respaldar. Este método fomenta una cooperación cercana entre los especialistas en el campo (stakeholders) y el equipo de desarrollo, lo que facilita la modelación del software de forma que represente con exactitud los procedimientos y normas empresariales.

Para conseguirlo, DDD emplea un lenguaje ubicuo, un vocabulario que todos los participantes del proyecto comparten. Este idioma, que proviene directamente del dominio, suprime las ambigüedades y garantiza que todos los participantes entiendan los términos y conceptos de igual forma.

3.3.1 Principios fundamentales

El Diseño de Software Directo se fundamenta en principios fundamentales que orientan el diseño del software para que sea representativo del sector empresarial:

Lenguaje Relevante: El equipo de desarrollo junto a los especialistas en negocios adoptan un lenguaje común que se emplea durante todo el proceso de desarrollo, desde las reuniones

hasta la codificación. Este idioma suprime ambigüedades y garantiza una comunicación nítida.

Formación del Dominio: El dominio se divide en componentes más reducidos y controlables, facilitando así la captura de las normas empresariales y su representación como entidades, agregados y servicios.

Contexto Definido (Contexto Limitado): Establece las fronteras precisas dentro del sistema, garantizando que cada segmento del modelo de dominio sea independiente y consistente en su contexto.

3.3.2 Componentes Clave:

Organizaciones: Se trata de elementos singulares con identidad en el dominio, tales como un cliente o un producto.

Objetos de Importancia: Son rasgos del dominio que carecen de identidad propia, como un intervalo de fechas o una ubicación.

Añadidos: Se reúnen entidades y objetos de valor bajo una entidad central que gestiona las relaciones.

Instaladores: Administran la conservación y restauración de entidades y agregados, funcionando como un nivel de abstracción para la base de datos.

Dominios de Servicios: Encapsulan operaciones complejas que no son propias de ninguna entidad o agregado concreto, pero resultan indispensables para el dominio.

3.3.3 Ventajas del DDD

El Diseño Basado en el Dominio facilita un vínculo íntimo entre el software y el negocio, garantizando que los sistemas representen con precisión las normas y procedimientos del dominio. Entre las principales ventajas sobresalen:

Mayor nitidez y exactitud: La aplicación de un lenguaje ubicuo evita confusiones entre desarrolladores e interesados.

Flexibilidad y capacidad de escalado: La división del dominio en contextos definidos promueve el modularidad, lo que facilita la escala y conservación de los sistemas.

Disminución de peligros: Al entender de manera más profunda el dominio, se disminuyen las equivocaciones en el diseño y desarrollo del software.

3.3.4 Casos de Uso del DDD

El DDD es perfecto para proyectos complejos con dominios extensos y profundos, en los que el entendimiento del negocio resulta crucial para el triunfo. Es especialmente beneficioso en sistemas fundamentados en microservicios, donde cada servicio puede simbolizar un entorno independiente y delimitado. Además, se aplica en áreas como las finanzas, el comercio electrónico y la manufactura, donde los sistemas deben gestionar normas y procedimientos específicos de la empresa.

3.3.5 La Función del DDD en las Arquitecturas Contemporáneas

En arquitecturas contemporáneas como los microservicios, el DDD se incorpora de forma orgánica, dado que cada microservicio puede ajustarse a un contexto específico del dominio. Esto no solo promueve la autonomía entre los elementos, sino que también asegura que cada componente del sistema esté diseñado para alcanzar una meta precisa y clara.

El DDD va más allá de una técnica; es una transformación cultural que fomenta la cooperación, la transparencia y el diseño estratégico, generando un software sólido, en concordancia con las metas empresariales y flexible ante posibles modificaciones futuras.

3.4. Arquitectura hexagonal

El modelo hexagonal, también denominado Arquitectura de Puertos y Adaptadores, fue sugerido por Alistair Cockburn con el objetivo de separar la lógica empresarial de las dependencias externas del sistema. Esta perspectiva hace posible que el núcleo de la aplicación sea autónomo de los aspectos técnicos, consiguiendo así una mayor adaptabilidad, escalabilidad y sencillez en el mantenimiento.

La arquitectura hexagonal organiza el software en tres capas principales: el núcleo del dominio, los puertos y los adaptadores. El núcleo del dominio contiene la lógica de negocio, mientras que los puertos definen las interfaces para interactuar con el exterior y los adaptadores implementan esas interfaces para conectar el sistema con elementos externos, como bases de datos, APIs o interfaces de usuario.

El diseño emplea una metáfora de un hexágono para simbolizar el núcleo principal del sistema, con lados que funcionan como puertos que permiten la interacción entre los adaptadores. Este modelo asegura que cualquier modificación en las dependencias externas no perjudique la lógica interna de la empresa, y al contrario.

3.4.1 Elementos Fundamentales

Núcleo del Dominio: Incluye la lógica empresarial pura, concebida de manera autónoma de cualquier aspecto tecnológico o técnico. Este núcleo establece las normas esenciales del sistema.

Los puertos: Son interfaces que facilitan la comunicación del núcleo con el entorno. dos clases:

Puertos de acceso: Administran las peticiones que se reciben en el sistema (como, por ejemplo, órdenes provenientes de una interfaz de usuario o un servicio).

Puertos de salida: Estipulan las interfaces por las que el núcleo se relaciona con servicios externos, tales como bases de datos o sistemas de terceros.

Adaptadores: Son desarrollos específicos de los puertos. Los adaptadores convierten las peticiones externas en un formato que pueda comprender el núcleo y a la inversa. Algunos ejemplos son controladores HTTP, almacenes de datos o integraciones con APIs externas.

3.5. Microservicios

La arquitectura de microservicios es una metodología de diseño de software que estructura una aplicación como un grupo de servicios pequeños, autónomos y profundamente unidos. Cada microservicio se encarga de una funcionalidad particular del sistema y interactúa con otros mediante interfaces claramente establecidas, como las APIs REST o los mensajes.

3.5.1 Características Fundamentales:

Recopilación: Los microservicios son independientes y tienen la capacidad de crecer, instalarse y expandirse de manera autónoma.

Especialización: Cada servicio se centra en solucionar un problema o desempeñar una función determinada en el sistema.

Escalable: Facilita la expansión de servicios individuales en función de la demanda, optimizando los recursos.

Capacitación tecnológica: Según las necesidades, los servicios pueden ser implementados en distintos lenguajes de programación y tecnologías.

3.5.2 Beneficios:

Promueve el desarrollo ágil, posibilitando que grupos pequeños operen en distintos servicios al mismo tiempo. Mejora la capacidad de resistencia, dado que las interrupciones en un servicio no impactan de manera directa a todo el sistema. Facilita la implementación de nuevas características.

La arquitectura de microservicios es perfecta para sistemas complejos y escalables, facilitando un desarrollo rápido y flexible ante cambios tecnológicos o empresariales.

3.6. Estrategias de descomposición de aplicaciones para microservicios

La descomposición en microservicios segmenta una aplicación monolítica en servicios pequeños e independientes, cada uno responsable de una función particular. Entre las tácticas más habituales se incluyen:

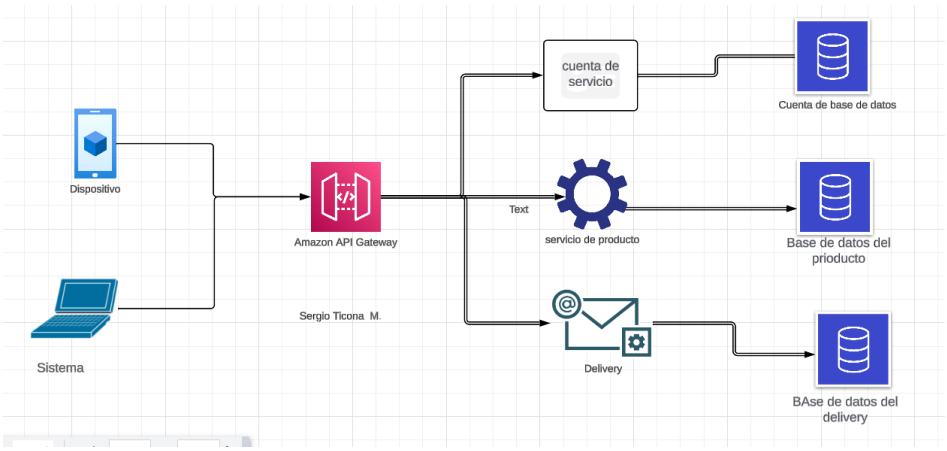
Estrategia de Descomposición	Descripción	Ventajas
Por funcionalidades	Divide la aplicación en servicios según funciones clave como 'usuarios' o 'pagos'.	Claridad funcional y fácil de implementar.
Basada en subdominios	Utiliza DDD para separar microservicios según áreas del negocio, como 'catálogo de productos'.	Alineación directa con las reglas de negocio.
Por componentes técnicos	Agrupa servicios en base a recursos técnicos compartidos, como bases de datos o autenticación.	Optimización de recursos compartidos.
Por casos de uso	Crea servicios según flujos principales del sistema, como 'procesar pedidos'.	Soporta procesos complejos con mayor flexibilidad.

3.7 Topología

Las topologías en una arquitectura de microservicios establecen la manera en que los servicios se organizan, interactúan entre ellos y se incorporan en un sistema. Estas estructuras son esenciales para garantizar el correcto desempeño, la escalabilidad y la conservación de los microservicios en diferentes contextos. A continuación, se muestran las topologías más relevantes empleadas en la práctica.

Topología	Descripción
Monolito Distribuido	Servicios parcialmente desacoplados, útil como transición hacia microservicios.
Equipos Independientes	Cada equipo gestiona un conjunto autónomo de servicios, fomentando la descentralización.
Orientada a Eventos	Los servicios se coordinan mediante mensajes asíncronos, reduciendo el acoplamiento.
Orquestación	Un servicio central coordina las interacciones, facilitando el control, pero introduciendo dependencias.
Coreografía	Los servicios coordinan sus interacciones de forma descentralizada, mejorando la resiliencia.

4. Ejemplo esquemático de un sistema basado en microservicios



5. Conclusiones

La arquitectura de software juega un papel crucial en la evolución de sistemas contemporáneos, facilitando la creación de soluciones escalables, adaptables y en

concordancia con las metas empresariales. Mediante métodos como los microservicios, la arquitectura hexagonal y el diseño basado en el dominio (DDD), se consigue desacoplar los elementos del sistema, lo que simplifica su mantenimiento, evaluación y progreso. Las tácticas de descomposición facilitan la estructuración de sistemas complejos en módulos autónomos, fomentando la resistencia y la capacidad de adaptación tecnológica.

Además, las perspectivas arquitectónicas, como la del modelo 4+1, proporcionan visiones complementarias que contribuyen a armonizar los intereses de los stakeholders técnicos y no técnicos, garantizando que el sistema satisfaga tanto las necesidades funcionales como las no funcionales. En conclusión, una arquitectura claramente establecida no solo incrementa la excelencia del software, sino que también disminuye los riesgos y asegura la viabilidad del sistema a lo largo del tiempo.

6. Bibliografía

Alistair Cockburn. (2005). Hexagonal Architecture. Consultado de

https://blog.hubspot.es/website/que-es-arquitectura-hexagonal

Atlassian. (n.d.). Microservices Architecture. Recuperado de

https://www.atlassian.com/es/microservices/microservices-architecture

Cristiá, M. (2007). Introducción a la Arquitectura de Software. Universidad Nacional de

Rosario. Recuperado de ResearchGate:

https://www.researchgate.net/publication/251932352_Introduccion_a_la_Arquitectura_de_So

ftware

Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software.

Addison-Wesley.

Gluo. (n.d.). Arquitectura de Software: ¿Qué es y qué tipos hay? Recuperado de

https://www.gluo.mx/blog/arquitectura-de-software-que-es-y-que-tipos-hay

HubSpot. (n.d.). Diseño Basado en el Dominio (DDD). Recuperado de

https://blog.hubspot.es/website/que-es-ddd

Kruchten, P. (1995). The 4+1 View Model of Architecture. IEEE Software, 12(6), 42–50.

Salguero, E. (n.d.). Arquitectura Hexagonal. Medium. Recuperado de https://medium.com/@edusalguero/arquitectura-hexagonal-59834bb44b7f

Wikipedia. (n.d.). Arquitectura de microservicios. Recuperado de https://es.wikipedia.org/wiki/Arquitectura_de_microservicios

Wikipedia. (n.d.). Modelo de Vistas de Arquitectura 4+1. Recuperado de https://es.wikipedia.org/wiki/Modelo_de_Vistas_de_Arquitectura_4%2B1