

**Tema**

INTRODUCCIÓN A LA ARQUITECTURA DE SOFTWARE

**Tutor**

Ing. Eduardo Mauricio Campaña Ortega

MIS. MDU. CCNA. CCIA.

PhD. (c) Ingeniería de Software

PhD. (c) Seguridad Información

**Fecha**

04/05/2025

**Tabla de contenido**

[INTRODUCCION A LA ARQUITECTURA DE SOFTWARE 4](#_Toc197257749)

[INTRODUCCIÓN 4](#_Toc197257750)

[OBJETIVO 4](#_Toc197257751)

[ARQUITECTURA DE SOFTWARE PARTE I 4](#_Toc197257752)

[ARQUITECTURA DE SOFTWARE PARTE II 9](#_Toc197257753)

[ARQUITECTURA DE SOFTWARE PARTE III 15](#_Toc197257754)

[CONCLUSIONES 20](#_Toc197257755)

[RECOMENDACIONES 21](#_Toc197257756)

[BIBLIOGRAFÍA 21](#_Toc197257757)

**ÍNDICE DE IMÁGENES**

[Figura 1. Analogía entre tipos de construcciones físicas y niveles de complejidad en la arquitectura de software 6](#_Toc197257553)

[Figura 2. La Mansión Winchester: una analogía con sistemas de software sin planeación arquitectónica clara 7](#_Toc197257554)

[Figura 3. La metáfora de las piedras: priorización de elementos clave en la arquitectura de software 9](#_Toc197257555)

[Figura 4. Modelo 4+1 vistas de la arquitectura de software según Kruchten 17](#_Toc197257556)

**ÍNDICE DE TABLAS**

[Tabla 1. Tipos de Requerimientos No Funcionales 13](#_Toc197257232)

# INTRODUCCION A LA ARQUITECTURA DE SOFTWARE

## INTRODUCCIÓN

La arquitectura de software constituye un componente esencial en el desarrollo de sistemas, ya que establece la estructura y el diseño general que guiará su construcción. Para comprender mejor su propósito y complejidad, puede compararse con la arquitectura tradicional: ambas disciplinas requieren planificación rigurosa, toma de decisiones estructurales y una orientación constante hacia la satisfacción de necesidades reales. Así como un arquitecto de edificaciones debe dominar aspectos técnicos y contextuales, el arquitecto de software debe comprender los requisitos del sistema, anticipar cambios y diseñar soluciones funcionales y sostenibles.

Dentro de esta disciplina, se destacan elementos clave como los componentes, las conexiones, los patrones arquitectónicos y la documentación, los cuales permiten definir el comportamiento y la adaptabilidad del sistema. Además, el modelado y diseño arquitectónico, mediante enfoques como el modelo 4+1 vistas y el uso de UML, resulta fundamental para representar, analizar y comunicar la estructura del software desde distintas perspectivas. Estos recursos no solo facilitan la colaboración entre equipos, sino que aseguran una base sólida para el desarrollo de sistemas escalables y coherentes con su entorno operativo.

## OBJETIVO

Explorar los principios fundamentales de la arquitectura de software mediante su analogía con la arquitectura tradicional, con el fin de comprender su propósito estructural, sus elementos esenciales y la relevancia del modelado y diseño arquitectónico como herramientas clave para construir sistemas robustos, escalables y alineados con las necesidades del entorno y los usuarios.

## ARQUITECTURA DE SOFTWARE PARTE I

La presentación, que introduce un tema tan relevante como la arquitectura de software fue desarrollada por Sorey Bibiana García Zapata; ella es ingeniera informática de la ciudad de Medellín, la cual además de contar con una especialización en desarrollo de software en el área FIT, se desempeña como desarrolladora para Microsoft. A lo largo de su exposición, se abordan conceptos claros y bien estructurados que facilitan la comprensión de la temática. Para iniciar, se presenta la introducción a lo que es la arquitectura de software.

La presentación aborda el tema de la arquitectura de software a través de una analogía con la ingeniería civil, comparando el rol del arquitecto de software con el del arquitecto encargado de diseñar edificaciones. Usar una analogía, permite que la comprensión resulte sencilla ya que muchas personas están familiarizadas con conceptos relacionados con construcciones, como edificios, casas o puentes. Por ello, para asimilar mejor los conceptos de la arquitectura de software, la presentación introduce algunas frases como ideas básicas sobre el concepto general de arquitectura, apoyándose también con ilustraciones.

En primer lugar, se cita a James M. Fitch, quien afirma: *“*La arquitectura es un instrumento cuya función principal es la de intervenir en favor del hombre*”*. Esta frase resalta la arquitectura como una herramienta al servicio del ser humano, útil para planear y materializar diversas construcciones o desarrollos: desde edificios, puentes y autopistas, hasta soluciones tecnológicas como software o sitios web.

A su vez, se presenta también una reflexión del arquitecto Glenn Murcutt, quien menciona: *“*La arquitectura debe ser una respuesta, no una imposición. La mayoría de los arquitectos que construyen edificios extraños aseguran que lo hacen porque ahora la tecnología lo hace posible. Poder hacer una cosa no legitima hacerla. Necesitamos soluciones para los problemas reales, no inventar problemas para poder empatar con nuevas soluciones.”

Este enfoque resalta la importancia de que la arquitectura, incluida la arquitectura de software, sea una respuesta concreta a necesidades reales. No se trata simplemente de aplicar tecnologías por el solo hecho de que existan, sino de generar soluciones útiles y pertinentes.

En este sentido, la arquitectura permite desarrollar productos de calidad, incluso cuando los recursos son limitados. No se debe asumir que la falta de tecnología o de medios impide lograr buenos resultados; con una planificación adecuada, es posible alcanzar desarrollos eficientes y funcionales. Por tanto, el propósito de la arquitectura es justamente brindar soluciones efectivas a las problemáticas reales.

Desde esta perspectiva, el rol del arquitecto de edificaciones y el del arquitecto de software comparten desafíos similares. Los retos a los que se enfrenta un ingeniero civil al desarrollar una gran obra guardan una estrecha relación con los que debe afrontar un arquitecto de software al momento de diseñar y construir una aplicación, ya sea de escritorio o web. Ambos deben considerar múltiples factores, tomar decisiones estructurales clave y asegurar que el resultado final cumpla con las necesidades del usuario y los objetivos del proyecto.

Ahora bien, es importante tomar en cuenta que se debe contar con un conjunto básico de habilidades y conocimientos para ejercer este tipo de roles.Por lo que, tanto en la arquitectura de edificaciones como en la de software, es fundamental poseer la capacidad para desarrollar una estructura sólida y funcional, lo que implica no solo habilidades técnicas, sino también un conocimiento profundo, y en muchos casos especializado el que permita llevar a cabo con éxito cualquier tipo de proyecto, ya sea una obra física o una solución tecnológica.

La arquitectura de software y la arquitectura tradicional comparten principios fundamentales en cuanto a diseño estructurado y planificación meticulosa. Ambas disciplinas buscan crear sistemas robustos y eficientes, ya sea en el ámbito digital o físico. Según (Elizondo, Cervantes Maceda, & Castro Careaga, 2015), la arquitectura de software se centra en la estructuración de sistemas complejos mediante la definición de sus componentes y relaciones, lo cual es análogo al proceso que siguen los arquitectos e ingenieros civiles al diseñar edificaciones y estructuras físicas. Esta analogía resalta la importancia de una planificación detallada y una comprensión profunda de los requisitos y restricciones del sistema, aspectos cruciales tanto en el desarrollo de software como en la construcción de infraestructuras físicas.

Por otro lado, para entender esta similitud, se complementa con una comparación entre distintos tipos de construcciones, resaltando que no es lo mismo diseñar una pequeña casa para perros, sin mayores detalles, que construir un edificio residencial completo, como los que conforman una urbanización. Los retos que enfrenta el diseñador o arquitecto en cada uno de estos casos son totalmente diferentes en complejidad, escala y recursos requeridos.

Asimismo, se presentan otros ejemplos contrastantes: una vivienda básica, con acabados mínimos, techo de teja y construida en su mayoría con cemento, frente a un edificio moderno, equipado con tecnología avanzada y respaldado por planos arquitectónicos mucho más complejos. También se menciona una casa mejor planeada, con elementos como garaje y molduras integradas, y otra aún más inusual, con un diseño poco común, que representa un verdadero desafío de ejecución para cualquier arquitecto.

Finalmente, se hace alusión a construcciones históricas como el **Taj Mahal**, que demandaron grandes cantidades de recursos, tiempo y esfuerzo colectivo. Estas edificaciones, por sus acabados excepcionales y su impacto cultural, siguen siendo íconos arquitectónicos a nivel mundial.



Figura 1. Analogía entre tipos de construcciones físicas y niveles de complejidad en la arquitectura de software

Esto demuestra que cada escenario de construcción plantea retos, condiciones y necesidades distintas. Por ello, se invita a reflexionar: ¿Qué herramientas, personas, presupuesto, conocimientos y tiempo se requieren para llevar a cabo cada uno de estos proyectos?

Al observar los ejemplos presentados, es posible analizar cada caso desde una perspectiva práctica: ¿Cuántas personas se necesitan para ejecutar el proyecto?, ¿Cuál es el presupuesto requerido?, ¿Qué conocimientos técnicos son necesarios?, ¿Y cuánto tiempo tomaría completarlo?

Claramente, construir un refugio para una mascota no demanda los mismos recursos que levantar un edificio de gran envergadura o diseñar un conjunto residencial completo. Cada proyecto conlleva una planificación específica, con fases de diseño, análisis y desarrollo que varían según su complejidad. En consecuencia, cada tipo de obra exige un equipo con distintos niveles de capacitación, así como una inversión proporcional de tiempo y recursos. La arquitectura, ya sea física o de software, siempre debe adaptarse a las características del problema que busca resolver.

Se puede afirmar, sin temor a equivocarse, que todas las consideraciones relacionadas con la arquitectura de edificaciones deben también tenerse en cuenta al momento de definir la arquitectura de software. Al igual que en los proyectos de construcción, donde se analizan aspectos como los recursos disponibles, la planificación, el diseño, el personal, el presupuesto y el tiempo, en el desarrollo de software estos factores también resultan fundamentales.

Por ello, la arquitectura de edificaciones y la arquitectura de software presentan una notable similitud, ya que ambas exigen una visión estructurada, una adecuada planeación y un conocimiento profundo del entorno en el que se ejecuta el proyecto.

En busca de un mejor enfoque del concepto de arquitectura de software, se presenta un caso particular como referencia a la famosa mansión Winchester. A modo de introducción, se muestra una vista panorámica de su fachada, lo que da pie a relatar la historia detrás de esta peculiar construcción: "En 1884, Sarah L. Winchester enviudó, heredando la fortuna del acaudalado fabricante de los rifles Winchester. Sin embargo, a pesar de su riqueza, una vidente de la época, a quien la desdichada viuda consultó, le aseguró que las almas de quienes habían muerto por causa de estos rifles estaban en pena. Años antes, Sarah también había perdido a su hija Annie, y la muerte de su esposo terminó por desequilibrar emocionalmente. Fue entonces cuando decidió comprar un extenso terreno en California, donde mandó a construir una casa que permaneció en obras constantes hasta el día de su muerte, 38 años después, en 1922. Según la vidente, solo de esta forma las almas hallarían un lugar donde residir hasta encontrar el reposo eterno."



Figura 2. La Mansión Winchester: una analogía con sistemas de software sin planeación arquitectónica clara

Este caso ilustra cómo una construcción puede extenderse indefinidamente debido a motivaciones muy particulares. A pesar de no seguir una planeación arquitectónica convencional, la mansión Winchester se convirtió en un proyecto de larga duración, sin un objetivo funcional claro, lo que contrasta con los principios fundamentales de una buena arquitectura, ya sea de edificaciones o de software.

Asimismo, la arquitectura de la mansión Winchester sorprendió a muchos y se escapó de los límites de la razón. No solo por su imponente estilo victoriano y sus características adelantadas a la época, como sus 160 habitaciones, tres ascensores, 47 chimeneas, sistemas de calefacción y alcantarillado, además de luces de gas que se activaban con solo presionar un botón, sino por las peculiaridades que la convierten en una construcción única.

Las imágenes presentadas muestran claramente estos detalles: una puerta situada al lado de una pared sin salida, una ventana que no comunica con ninguna estancia, fachadas con puertas que funcionan también como ventanas, y escaleras truncadas que no llevan a destino alguno, evidenciando una planificación arquitectónica completamente extraña.

En este punto surgen dos preguntas: ¿Qué relación tiene esta historia con la arquitectura de software? ¿Qué conexión existe entre una edificación tan inusual y el tema que se está abordando? La respuesta es que esta situación, trasladada al contexto del desarrollo de software, resulta ser más común de lo que debería. Cuando un desarrollador es asignado a la tarea de mantener y evolucionar un sistema legado, construido previamente por otra persona cuya arquitectura presenta fallas o carece de documentación adecuada, suele optar por reconstruir partes del sistema y crear sus propias rutas dentro del código.

Lo que ocurre finalmente es que, con el paso del tiempo, las aplicaciones de software terminan convirtiéndose en una "colcha de retazos" o en un verdadero laberinto, similar a la mansión Winchester, donde, si no se cuenta con una guía adecuada, es fácil perderse y resulta incierto volver a encontrar el camino correcto. Este representa un problema grave al que deben enfrentarse los desarrolladores de software. Con frecuencia, al integrarse a una empresa o asumir un proyecto existente, se encuentran con aplicaciones desarrolladas por terceros que no realizaron un trabajo riguroso en cuanto a la planeación ni en la creación de documentación de apoyo que facilite la comprensión del sistema y de sus funcionalidades.

En muchas ocasiones, los desarrolladores se ven en la necesidad de revisar código escrito por otras personas, el cual, en algunos casos, no está debidamente indentado, tabulado ni construido siguiendo buenas prácticas de desarrollo y diseño de software. Ante esta situación, no queda más opción que comenzar a integrar nuevo código propio dentro de la estructura existente. De esta manera, el software termina convirtiéndose en un sistema poco operable y aún más confuso para cualquier persona que, en el futuro, deba realizar tareas de mantenimiento o actualización.

Esta situación es bastante frecuente, pues, así como en la mansión Winchester se observaban elementos sin sentido, como una puerta que al abrirse revela únicamente una pared, en el software mal estructurado se presentan inconsistencias que dificultan su comprensión y evolución. Del mismo modo, es común encontrar numerosos errores y problemas al momento de realizar el mantenimiento de aplicaciones o sistemas de información que no fueron creados por el propio desarrollador, sino por terceros.

Para complementar la explicación, se presenta una frase de gran relevancia: "Programar sin una arquitectura en mente es como explorar una gruta solo con una linterna. No sabes dónde estás, dónde has estado ni hacia dónde vas" Danny Thorpe. Esta afirmación resulta sumamente acertada, ya que, al iniciar el desarrollo de un sistema de información sin una planeación previa, sin definir una arquitectura clara ni establecer un diseño estructurado, es inevitable que, con el paso del tiempo, el proyecto enfrente estancamientos y dificultades significativas en su evolución.

En estos casos, pueden surgir problemas que nunca fueron previstos, situando al desarrollador en un escenario oscuro y complejo del cual puede resultar difícil salir. Incluso, en ocasiones, puede ser necesario desistir en el proceso de desarrollo del software, dejando el sistema de información incompleto. Por esta razón, resulta fundamental aplicar una arquitectura adecuada, realizar un buen diseño del sistema y llevar a cabo un análisis detallado de los requerimientos del cliente. Solo así será posible construir un sistema sólido, completo y que pueda ser finalizado de manera satisfactoria.

De esta manera, la arquitectura se convierte en la base fundamental de un sistema de software, y debe ser diseñada no solo para satisfacer las necesidades actuales, sino también para dotar al software de las capacidades necesarias que le permitan mantenerse y evolucionar conforme a los requerimientos del negocio y a las solicitudes de los clientes. Como se pudo apreciar, la arquitectura desempeña un papel crucial en el desarrollo de software, ya que no solo permite anticiparse a los problemas que puedan surgir, sino que también facilita la evolución continua del sistema a lo largo del tiempo.

Por ello, en el ámbito de los sistemas, se sabe que constantemente surgen nuevos avances y tecnologías. Un software correctamente diseñado, será capaz de adaptarse a muchos de estos cambios. Aunque en ciertos momentos será necesario replantear algunas partes y modificar algunos desarrollos, en general, el sistema se adaptará eficazmente a nuevos entornos.

## ARQUITECTURA DE SOFTWARE PARTE II

Al momento de diseñar la arquitectura de un sistema, es fundamental considerar primero los aspectos más críticos e importantes, y posteriormente completar el diseño para satisfacer las necesidades generales del sistema. Siempre se debe tener en cuenta qué es lo primordial, ya que, si no se estructuran adecuadamente los elementos esenciales desde el inicio, será muy difícil organizarlos de manera correcta más adelante.



Figura 3. La metáfora de las piedras: priorización de elementos clave en la arquitectura de software

Como se aprecia en estas dos imágenes, aquí primero metieron las rocas más grandes y luego empezaron a meter las más pequeñas. Si ellos hubieran hecho lo contrario y hubieran puesto primero piedritas pequeñitas, tal vez después, cuando quisieran meter las piedras grandes no hubieran podido llevarse a cabo. Este ejemplo recuerda una historia interesante sobre un conferencista que abordaba el tema de cómo distribuir adecuadamente el tiempo. Él deseaba realizar una exposición que dejará una impresión duradera en los estudiantes y asistentes.

El día de su presentación, llegó al auditorio cargando un balde grande y varios objetos, lo cual causó intriga entre el público. Colocó el balde en el centro del escenario y comenzó a llenarlo con varias rocas grandes. Luego, preguntó al auditorio: “¿Está lleno el balde?”. Alguien respondió que sí. A continuación, el conferencista sacó una bolsa de arena y la vació en el balde. La arena se filtró entre los espacios que quedaban entre las piedras. Volvió a preguntar: “¿Está lleno el balde?”. Esta vez, alguien respondió con duda: “Tal vez no”. Entonces, el conferencista sacó una jarra con agua y la vertió en el balde, llenando los últimos espacios vacíos.

Nuevamente preguntó: “¿Ahora está lleno el balde?”, y los asistentes, más cautelosos, respondieron que tal vez no. El conferencista entonces cuestionó: “¿Qué enseñanza deja esta demostración?”. Un asistente respondió: “Que, aunque la agenda esté apretada, siempre habrá espacio para algo más”. A lo que el conferencista replicó: “No, ese no es el mensaje principal”. Explicó que, si no hubiera colocado primero las piedras grandes, luego no habría sido posible acomodarlas. Por tanto, en la vida es necesario identificar cuáles son las piedras más importantes, ejemplificando a Dios, la familia, el trabajo, entre otra, y darles prioridad, para luego ir acomodando los demás elementos.

Así también ocurre en el desarrollo de software, es fundamental saber identificar primero qué es lo más importante, ya que a esos elementos esenciales se les debe dedicar atención y trabajo prioritario. Posteriormente, en el transcurso del proyecto, se podrán ir diseñando y agregando otros componentes.

La arquitectura de software se compone de diversos elementos que definen su estructura, comportamiento y capacidad de adaptación. Estos elementos trabajan de forma conjunta para garantizar que el sistema cumpla con los objetivos funcionales y de calidad esperados. Comprender cada uno de ellos es esencial para diseñar soluciones robustas, escalables y mantenibles. Entre los elementos clave para la arquitectura de software se encuentran los siguientes:

**Componentes**

Los componentes son las unidades funcionales que constituyen el software, y cada uno de ellos desempeña tareas específicas, como el manejo de la interfaz de usuario, el procesamiento de la lógica de negocio o el acceso a bases de datos. Su correcta definición y separación permite una mayor modularidad, facilitando el desarrollo y mantenimiento del sistema. Según (Ken, 2023), los componentes representan piezas clave que pueden desarrollarse, reemplazarse o integrarse de forma independiente, lo que mejora la escalabilidad del sistema.

**Conexiones**

Las conexiones permiten la interacción entre los componentes y son responsables de cómo fluyen los datos y los comandos entre ellos. Estas conexiones deben ser claramente definidas mediante interfaces y protocolos que aseguren una comunicación eficaz. (Ojeda Montoya, 2023) destaca que las conexiones bien diseñadas no solo garantizan la interoperabilidad, sino que también contribuyen a la robustez general del sistema al reducir los acoplamientos innecesarios.

**Patrones de diseño y estilos arquitectónicos**

Los patrones de diseño constituyen soluciones reutilizables a problemas frecuentes en el desarrollo de software, mientras que los estilos arquitectónicos definen modelos estructurales de alto nivel. Patrones como MVC, repositorio o fábrica permiten organizar el código de forma lógica y eficiente. A su vez, estilos como la arquitectura de tres capas o la basada en microservicios ofrecen guías estructurales para el sistema completo. Como explica (Elizondo, Cervantes Maceda, & Castro Careaga, 2015), estos enfoques permiten abordar la complejidad del diseño mediante marcos conceptuales que ayudan a estructurar el sistema de manera coherente.

Además, de acuerdo con (Ken, 2023), es importante tomar en cuenta aquellos componentes que van desde el correcto análisis de requisitos como uso de tecnologías, que va acompañado de una correcta documentación, estos componentes son centrados en:

**Requisitos no funcionales y tecnologías**

Los requisitos no funcionales, como la escalabilidad, seguridad, rendimiento o disponibilidad, son elementos críticos de la arquitectura que influyen directamente en la experiencia del usuario y en la sostenibilidad del software. Estos requisitos deben guiar la selección de tecnologías, lenguajes, bases de datos y frameworks. Las herramientas tecnológicas elegidas impactan no solo en el diseño, sino también en la viabilidad y evolución del sistema.

**Documentación y mantenimiento**

Una arquitectura bien documentada facilita su entendimiento y adopción por todos los actores involucrados en el desarrollo. Diagramas, descripciones de componentes e interfaces, y registros de decisiones de diseño son recursos indispensables. Además, una buena arquitectura debe considerar su capacidad de evolución y mantenimiento, asegurando que el sistema pueda adaptarse a nuevas necesidades sin una reestructuración completa.

El arquitecto de software es el responsable de establecer el nivel de detalle, la estrategia y las herramientas necesarias para llevar a cabo una implementación que satisfaga tanto los requerimientos funcionales como los no funcionales del sistema. Es decir, el arquitecto de software debe definir de qué manera se realizará el desarrollo y cómo se organizarán los distintos elementos para lograr un producto sólido y sostenible.

Por otro lado, se revisará qué son los requisitos funcionales y no funcionales. Primero, es importante aclarar que ambos requisitos parten de lo que se conoce como los requerimientos del sistema. ¿Qué es un requerimiento? Un requerimiento es una característica que el sistema debe poseer o una restricción que debe satisfacer para ser aceptado por el cliente. Siempre será esencial trabajar de la mano con el cliente para definir estos aspectos.

Ahora bien, los requerimientos pueden clasificarse en funcionales o no funcionales. Un requerimiento funcional es aquel que describe la interacción entre el sistema y su entorno, independientemente de cómo se implemente. El entorno incluye tanto a los usuarios como a cualquier otro sistema externo que deba interactuar con el propio sistema. Por lo tanto, un requerimiento funcional abarca todo lo que rodea al sistema, ya sea los usuarios lo manejarán, qué otros programas o sistemas existentes deben integrarse, y en general, todos los elementos del ambiente que interactúan directamente con el sistema de información.

¿Y qué son los requerimientos no funcionales? Los requerimientos no funcionales describen aspectos del sistema que son visibles para el usuario, pero que no incluyen una relación directa con el comportamiento funcional del sistema. Algunos ejemplos de estos requerimientos pueden ser:

* **Interfaz gráfica**: define cómo será la apariencia visual que manejarán los usuarios, el diseño y disposición de los elementos de interacción.
* **Interfaz de hardware**: contempla los dispositivos periféricos necesarios para que el sistema funcione adecuadamente.
* **Interfaz de software**: especifica otros programas que deberán estar instalados o integrados para el correcto funcionamiento del sistema.
* **Desempeño**: establece qué tan rápido debe funcionar el sistema, si los resultados deben ser entregados en tiempo real, o de qué manera se almacenará la información.
* **Tolerancia a fallos**: señala qué medidas se tomarán para respaldar la información y qué dispositivos se requerirán para garantizar la continuidad del servicio en caso de fallas.
* **Seguridad**: determina el nivel de protección que debe ofrecer el sistema. Por ejemplo, un sistema bancario exigirá medidas de seguridad mucho más rigurosas que un sistema orientado a usuarios comunes.
* **Calidad del software**: define la necesidad de utilizar ciertos lenguajes de programación, plataformas específicas o sistemas operativos concretos.
* **Operación**: describe la manera en que el programa trabajará y qué entregables proporcionará.

Tabla 1. Tipos de Requerimientos No Funcionales

|  |  |  |
| --- | --- | --- |
| **Tipo** | **Descripción** | **Ejemplos** |
| **Rendimiento** | Define la velocidad de respuesta, tiempo de procesamiento y capacidad de manejo de carga del sistema. | El sistema debe responder en menos de 2 segundos bajo carga normal. |
| **Escalabilidad** | Indica la capacidad del sistema para adaptarse a un aumento en la carga o crecimiento futuro. | El sistema debe soportar 10,000 usuarios concurrentes con posibilidad de escalar a 100,000. |
| **Disponibilidad** | Especifica el porcentaje de tiempo que el sistema debe estar operativo. | El sistema debe tener una disponibilidad del 99.9% anual. |
| **Mantenibilidad** | Define la facilidad con la que el sistema puede ser modificado o corregido. | El sistema debe permitir aplicar parches sin necesidad de reiniciar. |
| **Portabilidad** | Determina la capacidad del software para funcionar en diferentes plataformas o entornos. | La aplicación debe correr en Windows, macOS y Linux sin cambios de código. |
| **Usabilidad** | Establece qué tan fácil es para los usuarios aprender y operar el sistema. | Un nuevo usuario debe poder utilizar el sistema básico con menos de 30 minutos de formación. |
| **Seguridad** | Define los mecanismos de protección de datos y control de acceso. | Los usuarios deben autenticarse mediante doble factor; los datos deben estar cifrados. |
| **Fiabilidad** | Indica qué tan consistente y libre de errores es el sistema durante su uso. | El sistema no debe presentar más de una falla crítica cada 6 meses. |
| **Tolerancia a Fallos** | Especifica cómo reacciona el sistema ante fallas o errores. | Si se pierde conexión a la base de datos, el sistema debe intentar reconectar 3 veces. |
| **Interoperabilidad** | Indica la capacidad del sistema para interactuar con otros sistemas o tecnologías. | El sistema debe integrarse con SAP y Salesforce mediante APIs REST. |
| **Requerimientos de Plataforma/Tecnología** | Establece restricciones sobre herramientas, tecnologías o lenguajes. | El sistema debe desarrollarse en Java y utilizar PostgreSQL como base de datos. |
| **Regulatorios/Legales** | Define requisitos derivados de leyes, normas o estándares específicos. | El sistema debe cumplir con el Reglamento General de Protección de Datos (GDPR). |

Esto resume lo que implica trabajar con los requerimientos funcionales y no funcionales de un sistema. Por lo tanto, el arquitecto de software es el encargado de analizar cómo se implementarán estos requerimientos, de qué manera se realizará el análisis funcional del sistema y cómo se cumplirán los requerimientos no funcionales.  
Además, debe ser una persona capaz de identificar las necesidades del negocio, evaluar las habilidades de su equipo de trabajo y considerar la viabilidad de las tecnologías disponibles para el desarrollo del software. El arquitecto debe saber liderar a su equipo, transmitir claramente las ideas del proyecto, seleccionar las tecnologías adecuadas y adaptarse a las necesidades específicas de la empresa que requiere el sistema de información.

Un buen arquitecto debe estar en capacidad de entender todas las condiciones a las que se verá sometido un sistema y proponer una solución acorde a cada escenario en particular. El buen arquitecto es aquel que se anticipa a los problemas; es quien, cuando se presenta una dificultad, es capaz de brindar una solución efectiva. Para los seres humanos siempre ha sido sencillo encontrar problemas. La crítica es constante y los errores son fáciles de señalar. Sin embargo, las personas que verdaderamente se destacan son aquellas que no solo detectan los problemas, sino que además son capaces de encontrar soluciones concretas para cada uno de ellos. El verdadero aporte de un arquitecto de software no radica únicamente en identificar los posibles inconvenientes, sino en ofrecer alternativas viables que permitan avanzar y mejorar continuamente el sistema.

Por tanto, la madurez de un arquitecto dará a las aplicaciones de software que tenga a su cargo una especificación coherente, permitiendo enfrentar un conjunto de riesgos mucho más reducido que en el caso de un arquitecto aprendiz. Aquí se evidencia la importancia de la experiencia. Las personas con mayor trayectoria en arquitectura de software son capaces de anticipar muchos problemas y de realizar desarrollos más estables y eficientes, en comparación con aquellas que apenas comienzan a adentrarse en el mundo del diseño y desarrollo de sistemas.

Y es que, como afirma Edward V. Berard: **"Desarrollar software de calidad basado en requisitos y caminar sobre el agua son cosas muy fáciles… si ambos están congelados"**. Esta frase encierra una gran verdad.

Si el agua está congelada, caminar sobre ella es posible. De igual manera, si los sistemas de información nunca evolucionaran y la tecnología no avanzara día a día, desarrollar software sería una tarea sencilla. Sin embargo, todo sistema de información debe estar diseñado de tal forma que pueda adaptarse a los cambios constantes del entorno, de las tecnologías, de las necesidades de los usuarios y de las organizaciones. Allí radica el gran desafío para el desarrollador de software: anticiparse a los problemas y construir sistemas capaces de evolucionar con el tiempo. Lo más difícil en la implementación es precisamente lograr esa capacidad de adaptación. Si las condiciones permanecieran estáticas, desarrollar software sería mucho más sencillo. Pero, como todo cambia continuamente, el software debe ser dinámico, flexible y preparado para afrontar las nuevas necesidades que inevitablemente surgirán.

Diseñar la arquitectura de software es como construir un puente en una zona sísmica: no basta con que sea sólido al inicio, debe poder resistir los movimientos que inevitablemente llegarán. En el mundo digital, estos "movimientos" se traducen en cambios constantes en los requerimientos del usuario, la evolución de las tecnologías y la presión del mercado. Como señalan (Bass, Clements, & Kazman, 2016), una buena arquitectura no es solo una solución técnica, sino una estrategia que permite evolucionar el sistema sin comprometer su estabilidad. Por ello, la capacidad de adaptación debe ser una propiedad inherente de la arquitectura, pensada desde las primeras decisiones del diseño.

## ARQUITECTURA DE SOFTWARE PARTE III

A diferencia de los arquitectos de edificaciones, los arquitectos de software son los encargados de construir directamente la base sobre la cual se desarrollará toda la arquitectura de los proyectos. Mientras que un arquitecto de edificaciones se limita a realizar planos y esquemas que luego son ejecutados por otras personas encargadas de la construcción, el arquitecto de software debe involucrarse activamente en la creación de los cimientos técnicos del sistema de información.

Esta diferencia explica por qué es común aceptar que un arquitecto de edificaciones puede destacarse sin haber colocado jamás un ladrillo. Sin embargo, no se puede decir lo mismo de un arquitecto de software que nunca haya programado. Un arquitecto de software debe conocer en profundidad la programación, pues está directamente implicado en la definición de cómo se implementará el sistema. Aunque su rol principal sea el diseño de la arquitectura y no necesariamente la codificación del sistema completo debe tener la capacidad de desarrollar componentes clave y comprender a fondo el proceso técnico. Solo así podrá tomar decisiones acertadas y guiar eficazmente al equipo de desarrollo.

Al igual que ocurre en una obra de construcción, un equipo de desarrollo de software debe contar con las habilidades necesarias para implementar correctamente el sistema y seguir las indicaciones del arquitecto, con el objetivo de alcanzar el éxito del proyecto. Para que esto sea posible, es fundamental conformar un equipo de desarrollo competente, con personal idóneo para cada una de las funciones involucradas. El éxito del proyecto depende en gran medida de la comunicación y colaboración efectiva entre el arquitecto de software y su equipo. Si no existe entendimiento mutuo, o si el equipo no sigue las directrices definidas por el arquitecto, el proyecto difícilmente podrá avanzar con éxito. Una de las dificultades comunes en el desarrollo de software es precisamente encontrar un equipo bien capacitado y alineado con los objetivos del proyecto. Sin embargo, aunque no siempre es una tarea sencilla, conformar un buen equipo de desarrollo es completamente posible con una adecuada selección, liderazgo y gestión.

Otra parte del símil con la arquitectura tradicional está en el uso de planos y maquetas, los cuales permiten a los arquitectos especificar y detallar cómo serán construidas las edificaciones. Estas herramientas visuales sirven para representar de forma tangible el proyecto que se va a desarrollar. En el ámbito de la arquitectura de software ocurre algo similar: los arquitectos de software también utilizan modelos que representan las especificaciones y necesidades técnicas de los sistemas.

De esta manera, los modelos y diagramas en el desarrollo de software cumplen una función análoga a la de los planos y maquetas en la arquitectura física. Estos artefactos requieren del uso de herramientas adecuadas, habilidades técnicas y conocimientos específicos, ya que servirán como guía fundamental en el proceso de construcción del sistema. Así como en arquitectura se elaboran distintas vistas del mismo proyecto, los arquitectos de software también generan diferentes representaciones del sistema: desde vistas generales que ofrecen una perspectiva amplia del sistema de información, hasta modelos más detallados que describen componentes internos, relaciones entre módulos, flujos de datos y estructuras específicas.

Dependiendo de la complejidad del proyecto y del enfoque de ingeniería adoptado, estos modelos pueden ser más globales o más específicos. Algunos muestran solo la interfaz o estructura exterior, mientras que otros profundizan en los elementos internos, como las funciones de los módulos, los servicios implicados o la organización lógica del sistema.

Los diagramas utilizados para representar el diseño y la distribución del software permiten visualizar distintas perspectivas de un mismo sistema, así como de las condiciones del entorno donde será desplegado. De forma similar a lo que ocurre en la arquitectura tradicional, donde se utilizan planos y maquetas para ilustrar la distribución eléctrica, hidráulica o estructural de una edificación desde distintos enfoques, en la arquitectura de software también se recurre a modelos que ofrecen diferentes vistas del sistema.

Cada diagrama en el contexto del software refleja un aspecto particular del sistema: su estructura lógica, la interacción entre componentes, la comunicación con sistemas externos o la experiencia del usuario, entre otros. Esta diversidad de perspectivas facilita una comprensión integral del sistema de información que se desea construir. Por ello, el uso de diagramas resulta fundamental. Estos no solo permiten representar el sistema desde diferentes ópticas, sino que, en conjunto, ofrecen una visión completa y coherente del proyecto. Gracias a esta representación múltiple, es posible analizar, diseñar y comunicar de forma más efectiva las soluciones arquitectónicas necesarias para garantizar el éxito del desarrollo.

Dentro del campo de la arquitectura de software, el modelo 4+1 vistas se han consolidado como una de las aproximaciones más completas y estructuradas para describir sistemas complejos. Este enfoque, propuesto por Philippe Kruchten, organiza la arquitectura en cinco perspectivas complementarias: lógica, de procesos, de desarrollo, de despliegue y escenarios, permitiendo comunicar eficazmente las decisiones arquitectónicas a distintos públicos. Según (Kruchten, 1995), cada vista responde a las preocupaciones de diferentes grupos de interés, y los escenarios actúan como el hilo conductor que valida y conecta las demás vistas. Esta separación de perspectivas favorece la comprensión, mantenimiento y evolución del sistema.

Dentro del ámbito de la arquitectura de software, existe un enfoque ampliamente reconocido conocido como el modelo **4+1 vistas**. Esta propuesta define un conjunto de perspectivas que permiten representar de manera integral el diseño y la arquitectura de un sistema de software. A través de este modelo, es posible visualizar el sistema desde cinco ángulos complementarios:

1. **Vista lógica**: Describe la funcionalidad que debe proporcionar el sistema a los usuarios finales, centrándose en la estructura de clases, objetos y relaciones.
2. **Vista de procesos**: Se enfoca en los aspectos dinámicos del sistema, como la comunicación y concurrencia entre procesos, subprocesos y tareas.
3. **Vista de desarrollo (implementación)**: Muestra cómo el sistema está organizado a nivel de módulos y componentes de software, facilitando la gestión del código fuente.
4. **Vista de despliegue**: Representa la distribución física del sistema, incluyendo nodos, servidores y conexiones de red donde los componentes serán ejecutados.
5. **Escenarios (o casos de uso)**: Funcionan como un eje transversal que vincula las demás vistas, proporcionando ejemplos concretos de cómo el sistema debe comportarse bajo determinadas condiciones.

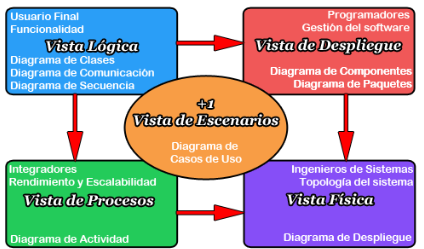


Figura 4. Modelo 4+1 vistas de la arquitectura de software según Kruchten

Aunque este modelo plantea su propia estructura de modelado, en la práctica, el lenguaje más utilizado para representar estas vistas es UML (Unified Modeling Language). UML se ha consolidado como la notación estándar para el modelado de sistemas de software, gracias a su flexibilidad y capacidad para capturar tanto los aspectos estructurales como los comportamentales de un sistema. Su amplio conjunto de diagramas facilita a los arquitectos de software el diseño de soluciones sólidas, escalables y adaptables a lo largo del tiempo.

Cada una de las vistas del modelo 4+1 puede representarse mediante uno o varios diagramas UML, lo que facilita su documentación y comprensión. UML (Unified Modeling Language) se ha convertido en el lenguaje estándar para la especificación, visualización y construcción de artefactos en sistemas de software, y proporciona un conjunto de diagramas que se alinean con cada una de las vistas del modelo.

Como afirman (Booch, Rumbaugh, & Jacobson, 2005), UML proporciona un lenguaje visual común que facilita la comunicación entre desarrolladores, diseñadores y otros interesados, además de ofrecer una base sólida para documentar y analizar decisiones arquitectónicas. Esta capacidad de adaptación y claridad ha consolidado a UML como la notación más utilizada para implementar modelos como el 4+1 vistas.

A continuación, se detalla la correspondencia entre las vistas del modelo 4+1 y los diagramas UML más comúnmente utilizados:

* **Escenarios (Casos de uso)**: Esta vista transversal se representa mediante los **diagramas de casos de uso**, que muestran las diferentes formas en que los actores (usuarios u otros sistemas) interactúan con el sistema. Permiten identificar los requisitos funcionales y constituyen el punto de partida para el modelado.
* **Vista lógica**: En esta vista se modela la lógica del negocio del sistema. Los diagramas UML más representativos son:
  + **Diagrama de clases**: Define la estructura de objetos, atributos, métodos y relaciones.
  + **Diagrama de estados**: Representa los distintos estados de un objeto y las transiciones entre ellos.
  + **Diagrama de colaboración (o de comunicación)**: Muestra cómo los objetos interactúan para llevar a cabo una funcionalidad.
* **Vista de desarrollo (implementación)**: Refleja la organización interna del sistema a nivel de código. Se emplean:
  + **Diagramas de componentes**: Representan los módulos de software, bibliotecas y su interacción. Aquí se especifica qué lenguaje de programación se utilizará, qué tecnologías estarán involucradas y qué motor de base de datos se integrará.
* **Vista física (despliegue)**: Describe la infraestructura donde será ejecutado el sistema. Se utilizan:
  + **Diagramas de despliegue**: Muestran la disposición física de los nodos (servidores, estaciones cliente, dispositivos periféricos) y la comunicación entre ellos.
* **Vista de procesos**: Enfocada en los aspectos dinámicos del sistema, como concurrencia y sincronización. Se pueden usar:
  + **Diagramas de actividad**: Describen flujos de trabajo y procesos del sistema.
  + **Diagramas de estado**: Ilustran los estados de los objetos en el tiempo.
  + **Diagramas de secuencia**: Muestran cómo los objetos interactúan entre sí mediante mensajes ordenados en el tiempo.

En conjunto, estas representaciones permiten tener una visión integral y detallada del sistema desde múltiples perspectivas, asegurando un diseño sólido, coherente y alineado con los requisitos del proyecto.

De esta manera, es posible presentar las distintas perspectivas de un sistema de software a partir de dos enfoques ampliamente reconocidos: el modelo **4+1 vistas** y el **lenguaje UML**, ambos orientados a satisfacer las necesidades de los distintos actores involucrados en el proceso de desarrollo. Cada una de estas vistas proporciona una representación específica del sistema, dirigida a un público particular dentro del equipo de desarrollo:

* **Vista lógica**: Orientada a los *analistas y diseñadores de software*, esta vista describe la estructura funcional del sistema. Se enfoca en representar los elementos clave de la lógica del negocio, como clases, relaciones, estados y comportamientos.
* **Vista de implementación (o de desarrollo)**: Dirigida a los *programadores y desarrolladores*, esta vista aborda la organización del código fuente, los módulos y los componentes reutilizables del sistema. Permite gestionar cómo se construirá e integrará el software.
* **Vista de casos de uso (escenarios)**: Pensada para los *usuarios finales y clientes*, esta vista detalla cómo los distintos actores interactúan con el sistema, definiendo permisos, funcionalidades y flujos de uso. Sirve como punto de partida para entender los requisitos funcionales.
* **Vista de procesos**: Enfocada en los *integradores del sistema*, esta vista modela los aspectos dinámicos del sistema, como la concurrencia, la sincronización, la escalabilidad y el rendimiento, asegurando que los distintos procesos funcionen adecuadamente de forma paralela o distribuida.
* **Vista de despliegue (física)**: Diseñada para los *ingenieros de infraestructura*, esta vista define la topología del sistema, es decir, cómo se distribuyen los distintos nodos físicos (servidores, clientes, redes), así como los aspectos relacionados con la instalación, entrega y comunicación del sistema.

De esta manera, es posible presentar las distintas perspectivas de un sistema de software a partir de dos enfoques ampliamente reconocidos: el modelo 4+1 vistas y el lenguaje UML, ambos orientados a satisfacer las necesidades de los distintos actores involucrados en el proceso de desarrollo. Cada una de estas vistas ofrece una representación específica que responde a los intereses de diferentes perfiles dentro del equipo, desde usuarios finales hasta ingenieros de infraestructura. Tal como explican (Kruchten, 1995) , el modelo 4+1 utiliza diferentes vistas como lógica, de procesos, de desarrollo, de despliegue y de casos de uso, para abordar preocupaciones particulares, mientras que UML proporciona los diagramas necesarios para representar visualmente cada una de ellas de manera coherente y estandarizada. Esta combinación resulta esencial para garantizar una comprensión integral, comunicación efectiva y documentación precisa en proyectos complejos de software.

Con esto concluimos esta introducción a la arquitectura de software. A lo largo de esta presentación, se ha abordado los conceptos fundamentales que permiten comprender la importancia de una arquitectura bien diseñada para el desarrollo exitoso de sistemas de información. Así como en la construcción de edificaciones es esencial contar con una planificación adecuada para evitar fallos estructurales, en el desarrollo de software, una arquitectura sólida es la base que garantiza escalabilidad, mantenibilidad, adaptabilidad y rendimiento del sistema. El diseño arquitectónico permite anticipar riesgos, asignar responsabilidades adecuadamente y construir soluciones tecnológicamente viables y alineadas con las necesidades del negocio.

Este ha sido un primer acercamiento. Más adelante, se profundizará en cada una de las vistas arquitectónicas, herramientas y metodologías clave que todo arquitecto de software debe dominar. Para conocer más de la temática pueden visitar el blog de la autora de esta presentación, donde encontrarán otros contenidos igualmente valiosos y complementarios. Continuaremos en las próximas sesiones con un análisis más detallado de cada componente de la arquitectura de software.

## CONCLUSIONES

La arquitectura de software comparte principios fundamentales con la arquitectura tradicional, como la planificación meticulosa, la toma de decisiones estructurales y la orientación a satisfacer necesidades reales.

A su vez, así como un arquitecto de edificaciones requiere conocimientos técnicos y visión estructural, un arquitecto de software debe dominar la programación y comprender profundamente los requerimientos del sistema para construir soluciones funcionales, sostenibles y adaptables, incluso con recursos limitados.

Por otro lado, la arquitectura de software se compone de elementos clave tales como: componentes, conexiones, patrones, requisitos y documentación, que, en conjunto, definen su estructura, comportamiento y capacidad de adaptación.

Es importante contemplar que un arquitecto de software no solo debe dominar estos elementos, sino también anticipar cambios tecnológicos y del negocio, brindando soluciones flexibles

Por último, el modelado y diseño de la arquitectura de software es fundamental para visualizar la estructura, el comportamiento y la interacción de los componentes dentro del entorno donde se desplegará el sistema.

Modelos clave como el 4+1 vistas de Philippe Kruchten, permiten analizar el sistema desde diferentes ángulos, facilitando la comunicación y comprensión entre los distintos grupos involucrados. Al integrar el modelo 4+1 con UML, se logra una visión integral que asegura el diseño y la implementación efectiva de soluciones flexibles y escalables.

## RECOMENDACIONES

Es fundamental considerar que un arquitecto de software no debe centrarse solo en las necesidades actuales del sistema, sino que anticiparse a posibles cambios futuros tanto tecnológicos como comerciales. Esto garantiza que el sistema se mantenga relevante y eficiente a lo largo del tiempo, adaptándose a nuevas demandas sin necesidad de rediseños costosos.

Por otro lado, la arquitectura de software debe facilitar la comunicación clara entre todos los grupos involucrados en el proyecto, por lo que, es recomendable optar por modelos como el 4+1 vistas y UML, lo que asegura que todos los actores, desde desarrolladores hasta clientes, comprendan cómo se estructura y opera el sistema, lo que optimiza la colaboración y la toma de decisiones.

## BIBLIOGRAFÍA

Bass, L., Clements, P., & Kazman, R. (2016). *Software Architecture in Practice* (3.ª edición ed.). Boston, MA: Addison-Wesley.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (Segunda edición (2nd ed.) ed.). Boston: Addison-Wesley.

Elizondo, P. V., Cervantes Maceda, H., & Castro Careaga, L. (2015). *Arquitectura de Software: Conceptos y Ciclo de Desarrollo.* México, D.F.: Cengage Learning Editores, S.A. de C.V.

Ken, A. (26 de 11 de 2023). *Arquitectura de software: ¿Qué es y qué tipos hay?* Obtenido de Gluo: https://www.gluo.mx/blog/arquitectura-de-software-que-es-y-que-tipos-hay

Kruchten, P. (1995). Architectural Blueprints—The “4+1” View Model of Software Architecture. *IEEE Software, 12*, 42–50. doi:https://doi.org/10.1109/52.469759

Ojeda Montoya, J. F. (13 de 9 de 2023). *Arquitectura de Software: Fundamentos, Características, Usos y Ejemplos.* Obtenido de Linkedin: https://es.linkedin.com/pulse/arquitectura-de-software-fundamentos-caracter%C3%ADsticas-y-ojeda-montoya