

Fundamentos de Arquitectura de Software

Guía didáctica

Unidad Académica Técnica y Tecnológica

Tecnología Superior en Transformación Digital de Empresas

Fundamentos de Arquitectura de Software

Guía didáctica

Carrera	PAO Nivel
▪ <i>Tecnología Superior en Transformación Digital de Empresas</i>	III

Autor:

Arguello Jacome Marco Ricardo
Daniel Alejandro Guamán Coronel



Asesoría virtual
www.utpl.edu.ec

Universidad Técnica Particular de Loja

Fundamentos de Arquitectura de Software

Guía didáctica

Arguello Jacome Marco Ricardo
Daniel Alejandro Guamán Coronel

Diagramación y diseño digital:

Ediloja Cía. Ltda.
Telefax: 593-7-2611418.
San Cayetano Alto s/n.
www.ediloja.com.ec
edilojacialtda@ediloja.com.ec
Loja-Ecuador

ISBN digital - 978-9942-39-816-1



**Reconocimiento-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)**

Usted acepta y acuerda estar obligado por los términos y condiciones de esta Licencia, por lo que, si existe el incumplimiento de algunas de estas condiciones, no se autoriza el uso de ningún contenido.

Los contenidos de este trabajo están sujetos a una licencia internacional Creative Commons **Reconocimiento-NoComercial-CompartirIgual 4.0 (CC BY-NC-SA 4.0)**. Usted es libre de **Compartir – copiar y redistribuir el material en cualquier medio o formato. Adaptar – remezclar, transformar y construir a partir del material citando la fuente, bajo los siguientes términos: Reconocimiento- debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios.** Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciante. **No Comercial-no puede hacer uso del material con propósitos comerciales. Compartir igual-Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original.** No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia. <https://creativecommons.org/licenses/by-nc-sa/4.0/>

Índice

1. Datos de información.....	8
1.1. Presentación de la asignatura	8
1.2. Competencias genéricas de la UTPL.....	8
1.3. Competencias específicas de la carrera	8
1.4. Problemática que aborda la asignatura	8
2. Metodología de aprendizaje.....	9
3. Orientaciones didácticas por resultados de aprendizaje.....	11
 Primer bimestre.....	 11
 Resultado de aprendizaje 1.....	 11
Contenidos, recursos y actividades de aprendizaje.....	11
 Semana 1	 11
 Unidad 1. Fundamentos de la arquitectura de software.....	 11
1.1. Arquitectura de software y su relación con la ingeniería de software.....	12
1.2. Definiciones de arquitectura de software	16
1.3. Importancia de la arquitectura de software	22
1.4. Beneficios de la arquitectura de software	25
1.5. El rol del arquitecto de software	27
Actividades de aprendizaje recomendadas	33
 Semana 2	 34
1.6. Arquitectura vs. Diseño	34
1.7. Elementos arquitectónicos	37
1.8. Partes interesadas (stakeholders).....	42
 Semana 3	 44
1.9. Descripción arquitectónica – Modelo conceptual ISO/IEC/IEEE 42010.....	44
1.10.Arquitectura lógica vs. Arquitectura física.....	51
Actividades de aprendizaje recomendadas	54
Autoevaluación 1	55

Semana 4	60
Unidad 2. Consideraciones y contextos para la definición y diseño de una arquitectura de software	60
2.1. Consideraciones para la definición de una arquitectura de software.....	60
2.2. Consideraciones para el diseño de una arquitectura de software .	62
2.3. Principios de diseño	64
2.4. ¿Qué hace que una arquitectura sea buena?.....	71
Actividades de aprendizaje recomendadas	74
Autoevaluación 2.....	75
Semana 5	80
Unidad 3. Calidad y atributos de calidad	80
3.1. Calidad del software	80
3.2. Calidad del diseño.....	81
3.3. Calidad de arquitectura o arquitectónica de software	81
3.4. Atributos de calidad.....	83
Actividades de aprendizaje recomendadas	92
Autoevaluación 3.....	93
Resultado de aprendizaje 2.....	96
Contenidos, recursos y actividades de aprendizaje.....	96
Semana 6	96
Unidad 4. Estilos y patrones arquitectónicos	96
4.1. Patrón de software	96
4.2. Estilos arquitectónicos.....	98
4.3. Beneficios de los estilos arquitectónicos	100
Semana 7	102
4.4. Estilos arquitectónicos comunes	102
Actividades de aprendizaje recomendadas	104
Semana 8	105
Actividades finales del bimestre	105
Actividades de aprendizaje recomendadas	106

Segundo bimestre	107
Resultado de aprendizaje 2.....	107
Contenidos, recursos y actividades de aprendizaje.....	107
Semana 9	107
4.5. Estilos y patrones arquitectónicos: patrones arquitectónicos	107
Semana 10	117
4.6. Patrón arquitectónico vs. Patrón de diseño.....	117
4.7. Evolución de las arquitecturas, software y tendencias	119
Actividades de aprendizaje recomendadas	124
Autoevaluación 4.....	125
Resultado de aprendizaje 3.....	128
Contenidos, recursos y actividades de aprendizaje.....	128
Semana 11	128
Unidad 5. Modelos de arquitectura	128
5.1. Modelos para descripción arquitectónica.....	129
Semana 12	139
Actividades de aprendizaje recomendadas	153
Autoevaluación 5.....	154
Semana 13	157
Unidad 6. Técnicas y tácticas para diseño, implementación y documentación de arquitecturas de software	157
6.1. Tipos de aplicaciones.....	158
6.2. Implementación de arquitectura.....	181
Semana 14	184
6.3. Pruebas y validación de arquitectura	184

Semana 15	191
6.4. Documentación de la arquitectura	191
Actividades de aprendizaje recomendadas	194
Autoevaluación 6.....	196
Semana 16	199
Actividades finales del bimestre	199
4. Solucionario	200
5. Referencias bibliográficas	210
5.1. Bibliografía básica	210
6. Anexos	214



1. Datos de información

1.1. Presentación de la asignatura



1.2. Competencias genéricas de la UTPL

- Comunicación oral y escrita.

1.3. Competencias específicas de la carrera

- Diseña modelos arquitectónicos de empresa para gestionar el alineamiento estratégico entre negocio y TI.
- Despliega infraestructura tecnológica para operar digitalmente los dominios arquitectónicos de una organización.

1.4. Problemática que aborda la asignatura

Concebir la arquitectura digital de una empresa a través del modelado, alineación y comprensión de las interacciones significativas entre el

negocio y las TIC en la era digital. La formación base en este núcleo académico girará en torno a formar los estudiantes con habilidades fuertes en la construcción de aplicaciones nativas en nube, lo que les permitirá a los estudiantes construir y entregar continuamente software escalable, resiliente y robusto en entornos de nube automatizados para un negocio digital.



2. Metodología de aprendizaje

Se utilizará la metodología de aprendizaje basado en problemas. Bajo este contexto se requiere que el estudiante haga uso de los conocimientos adquiridos hasta el momento como parte de su formación académica y/o formación profesional.

El aprendizaje basado en problemas es una metodología que implica que usted investigue y reflexione sobre un estudio de caso o escenario real donde puede existir una o varias soluciones ante un problema.

La capacidad de análisis es importante, especialmente porque siguiendo un proceso formal y utilizando técnicas de análisis podrá descomponer un problema y proponer una solución arquitectónica que incluya *hardware* y/o *software*, un despliegue que puede ser local o en la nube (*cloud*), y que pueda ser implementada a través de diversas tecnologías y lenguajes de programación. La propuesta de solución, desde el punto de vista estructural y dinámico, deben tener en cuenta la inclusión de atributos de calidad, estilos o patrones arquitectónicos.

Se abordan los conceptos básicos y fundamentales que se requiere conocer de la arquitectura de *software*. Además, se estudian modelos que permiten describir una arquitectura de *software*, y el proceso para llevar a cabo actividades de diseño e implementación de *software* a través del uso de estilos y patrones arquitectónicos. Tanto la documentación, diseño e implementación pueden conducirle a investigar y aplicar conceptos avanzados o nuevos para usted, es por esto que le sugiero revisar en esta guía la sección de bibliografía básica, complementaria, recursos educativos

abiertos disponibles en la web y a nexos que han sido seleccionados y preparados para su estudio. Algunos documentos digitales y ejemplos prácticos con explicación asociada se cargarán en la plataforma educativa virtual conforme avancemos las semanas de estudio del curso.



3. Orientaciones didácticas por resultados de aprendizaje



Primer bimestre

Resultado de aprendizaje 1

- Explica la descripción arquitectónica a través de vistas y escenarios arquitectónicos.

Para alcanzar este resultado de aprendizaje se analizaran los estilos y patrones arquitectónicos, enfocándonos en sus beneficios y enumerando los estilos arquitectónicos más comunes. Esto nos permitirá comprender los principios de diseño, estilos y patrones a aplicar en un diseño de arquitectura.

Contenidos, recursos y actividades de aprendizaje



Semana 1

Unidad 1. Fundamentos de la arquitectura de software

La arquitectura de software es una disciplina que se encarga de definir la estructura, componentes, módulos, interfaces y otros aspectos del software. Es decir, es la base sobre la cual se construye el software. La arquitectura de software permite tener una visión global del software, lo que facilita la toma de decisiones y la identificación de problemas. Además, permite la reutilización de componentes, lo que reduce el tiempo y costo de desarrollo.

Esta unidad hace referencia a los conceptos básicos y fundamentales para entender y comprender la arquitectura software. Estos le serán útiles al momento de proponer una o varias soluciones formales a un problema que

se resuelve con *software*. Es por ello que le invito a complementar cada uno de los temas de cada semana, apoyándose en la lectura comprensiva en bibliografía básica y complementaria, impresa o digital disponible en la web.

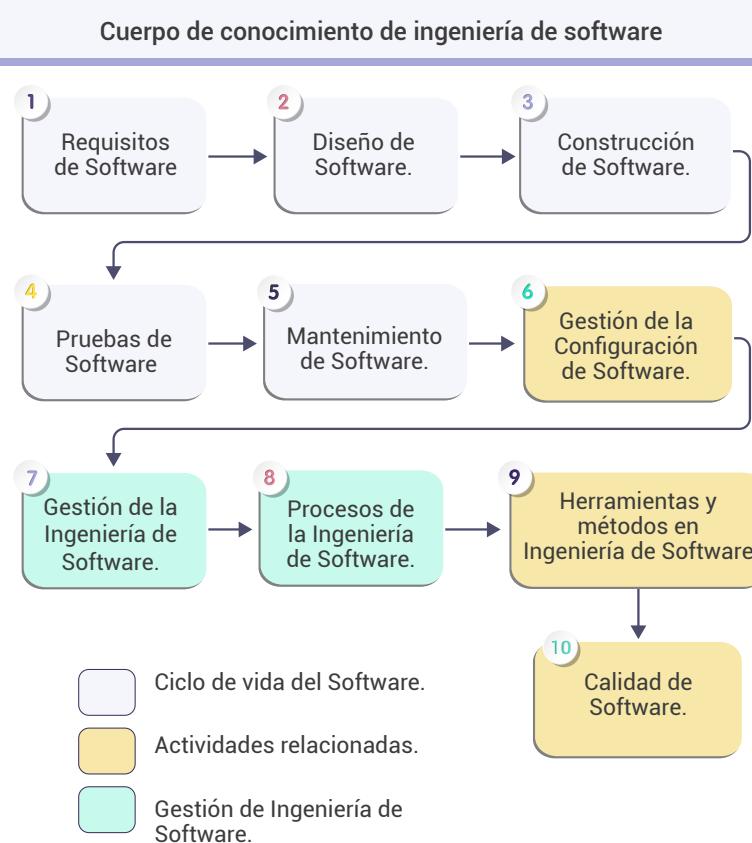
1.1. Arquitectura de *software* y su relación con la ingeniería de *software*

La [IEEE Computer Society](#), define a la Ingeniería de *software* como la “aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del *software*, es decir, la aplicación de la ingeniería al *software*”.

Una concepción errónea en una persona con conocimientos y habilidades en *software*, es pensar que todo se resuelve con programación. Adoptar el rol de ingeniero o arquitecto de *software*, implica tener también el conocimiento de otras áreas. Es lo que precisamente busca el SWEBOK (*Software Engineering Body of Knowledge*, Cuerpo de Conocimiento de Ingeniería de *Software*), el cual es un documento promovido por la IEEE Computer Society y creado por el Software Engineering Coordinating Committee, para desarrollar el área de conocimiento propio de la ingeniería del *software*.

Figura 1.

Áreas del cuerpo de conocimiento de la arquitectura de software



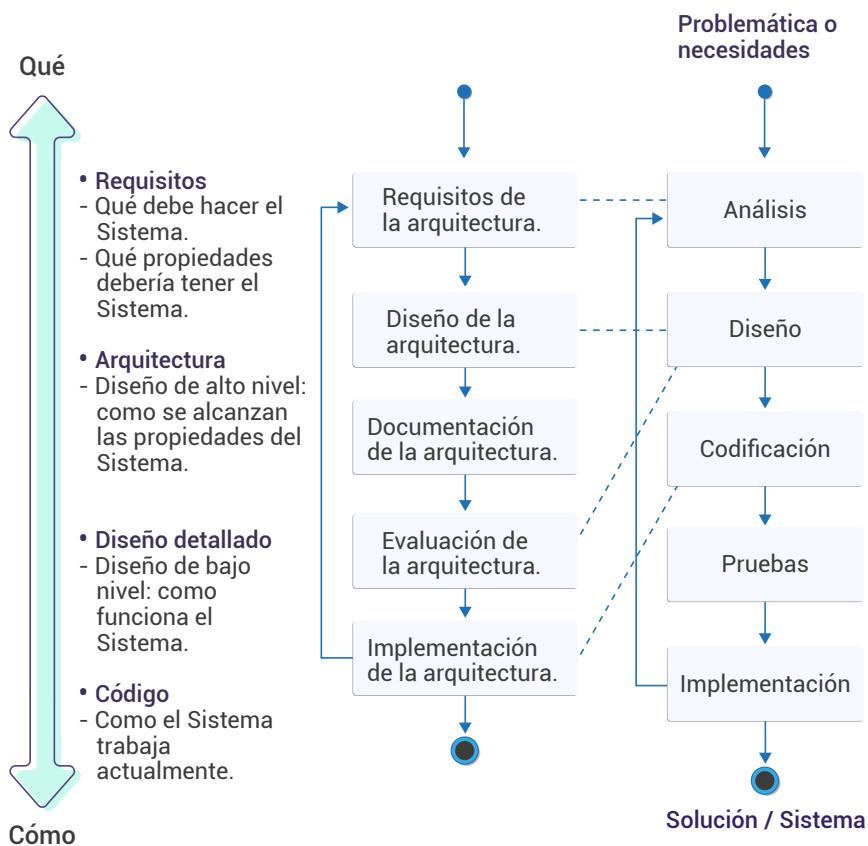
Nota. Argüello M., & Guamán D., 2023

Tal como se observa en la figura 1, en el presente curso estudiaremos cómo las áreas etiquetadas como: 1. Requisitos de Software, 2. Diseño de software, 3. Construcción de software, 4. Pruebas de software, 10. Calidad de software apoyan en el análisis, diseño, construcción, pruebas e implementación de una arquitectura de software.

El cuerpo de conocimiento de ingeniería de software propone 10 áreas donde destacan: los enfoques o metodologías de desarrollo (tradicionales, no tradicionales), las actividades que permiten gestionar el software y gestionar la calidad a través de herramientas y métodos, y las áreas de diseño y construcción del software a través de los cuales se puede definir la arquitectura, los componentes, las interfaces, y otras características de un sistema o componente.

En lo que respecta a la arquitectura de software, esta desempeña un papel esencial durante el ciclo de vida de desarrollo de software, sirviendo como puente entre los **requisitos** (**derivados de la fase de análisis**) e **implementación para proponer un diseño**.

Figura 2.
Fases del CVDS para definir, diseñar e implementar una arquitectura



Nota. Argüello M., & Guamán D., 2023

Una de las áreas dentro del SWEBOk hace referencia a los Requisitos de Software. Como se puede observar en la figura2, los requisitos son el insumo principal para iniciar con el diseño de una arquitectura de software. El diseño involucra hacer uso de modelos y diagramas que documentados permitirán elaborar los escenarios de prueba para evaluar la arquitectura desde el punto de vista de diseño y de implementación. Como se había mencionado anteriormente, enfocarse netamente en la programación de un sistema de software, no se podría considerar una opción adecuada,

especialmente cuando queremos que nuestro *software* satisfaga las necesidades del cliente.

Existen tres fundamentos que permiten comprender la arquitectura de *software* dentro del contexto de la ingeniería de *software*:

- Cada sistema de *software* tiene una arquitectura.
- Cada sistema de *software* tiene al menos una arquitectura.
- La arquitectura de *software* no es una fase del desarrollo.

Asociados a estos tres fundamentos, le pregunto ¿qué se le viene a la mente cuando escucha, Cliente-Servidor, Layers, Tiers, Publicador/Suscriptor, REST, SOA, MVC, orientación a objetos?, ¿escalabilidad, mantenibilidad, disponibilidad?, ¿vistas arquitectónicas, elementos arquitectónicos? Pues bien, estos son algunos de los conceptos propios de arquitectura de *software* que los estudiaremos en el presente curso y que serán de utilidad para diseñar y construir *software*.

Antes de iniciar con el estudio de los conceptos relacionados con arquitectura de *software*, es importante que conozcamos las fases del CVDS, su importancia, beneficios y roles que intervienen en cada una de ellas. En la figura 3 se evidencia algunos artefactos/entregables, actores/roles que son parte de la fase de análisis y diseño de *software*, y las tareas que cada uno de ellos realiza, esto será de utilidad más adelante cuando estudiemos el proceso para definir y diseñar una arquitectura.

Figura 3.
Fases de análisis y diseño de software



Nota. Argüello M., & Guamán D., 2023

1.2. Definiciones de arquitectura de software

La arquitectura de software ha sido definida por los autores desde diversos puntos de vista. Es así que Taylor y Medvidovic (2012), definen a la arquitectura software como el “**conjunto de decisiones de diseño** que gobiernan un sistema”. Cuando hablamos de sistema o sistemas informáticos, es importante recordar que estos se componen de tres partes fundamentales: **hardware** (por ejemplo, procesadores, memoria, discos, tarjetas de red), **software** (por ejemplo, programas o bibliotecas), y **datos**, que pueden ser transitorios (en memoria) o persistentes (en disco o ROM). Para Garlan (2000), la “arquitectura de software se define como

la descripción del sistema en términos de componentes funcionales, reutilizables e independientes que están interconectados y permiten la abstracción del sistema”.

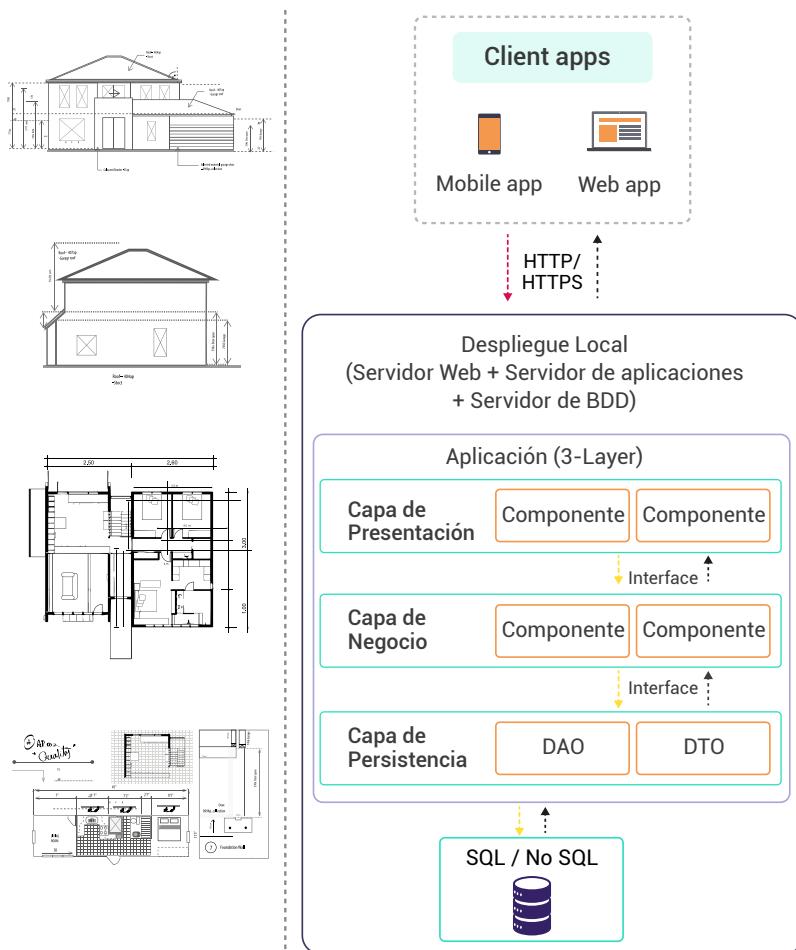
A diario usted interactúa con un sistema o solución informática a través de un equipo computacional o dispositivo electrónico y como profesional en formación le interesa conocer ¿qué hacen realmente las partes o componentes individuales de *software* y *hardware*?; ¿cómo funcionan juntos y cómo interactúan con el mundo que las rodea?, en otras palabras, lo que le interesa conocer es su arquitectura.

En otra definición dada por el Instituto de Ingeniería de Software Engineering Institute (2014), se afirma que la “arquitectura de un sistema intensivo en *software*, es la estructura o estructuras del sistema, que comprenden elementos de *software*, las propiedades visibles externas de esos elementos y las relaciones entre ellos”. En esta definición se identifican dos conceptos claves: **estructuras internas del sistema** (que son visibles y usadas por los desarrolladores) y **propiedades visibles externas** (que son visibles a los usuarios). Cuando se habla de estructura del sistema recuerde que existen dos tipos: **estructuras estáticas** y **estructuras dinámicas**. Las primeras definen los elementos de una arquitectura en tiempo de diseño, su disposición u organización. Mientras que las estructuras dinámicas definen sus elementos en tiempo de ejecución y sus interacciones.

Si hacemos una analogía, para explicar estos dos conceptos clave, pensemos cuando un arquitecto construye una casa, los planos impresos o digitales corresponden a la parte estructural, mientras que la distribución de agua potable a través de la tubería y la distribución de la energía eléctrica al accionar los interruptores de encendido y apagado corresponderían a la parte dinámica (ver figura 4). Por lo tanto, diseñar y construir una casa sin planos sería imposible, ¿verdad?

Figura 4.

Analogía construir casa vs estructuras o modelos para construir software
(Cliente – Servidor)



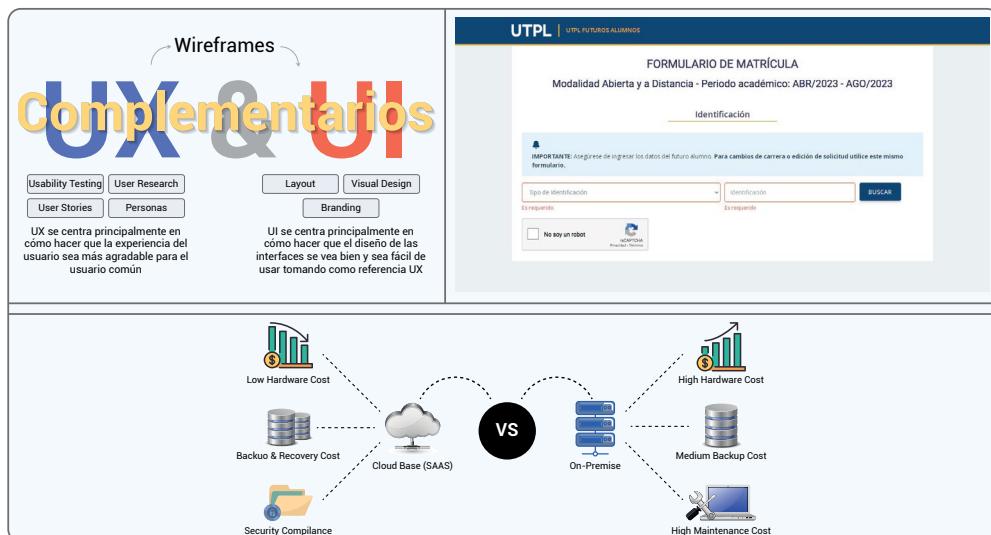
Nota. Argüello M., & Guamán D., 2023

Continuando con la definición dada por el SEI, las **propiedades visibles externas** en una arquitectura se manifiestan de dos maneras: la primera relacionada con el comportamiento externo visible (*lo que hace el sistema*) y la segunda relacionada con las propiedades de calidad (cómo *lo hace el sistema*). El **comportamiento externo visible de un sistema de software**, se refiere a las interacciones funcionales entre el sistema y su entorno (usuarios, otros sistemas o servicios) que tiene relación con conceptos de modelado de usuario UI (User Interface) / UX (User Experience). Las propiedades de calidad se refieren a una propiedad no funcional visible externamente de un sistema, lo que el usuario no lo visualiza, pero que

es parte del sistema, entre ellos constan por ejemplo la seguridad, el rendimiento, la escalabilidad, entre otros, evidenciados. Si hablamos de seguridad cuando se realiza un despliegue local, el propietario de la aplicación se encarga de instalar los certificados de seguridad, configurar los *firewalls*, configuración de protocolos de seguridad, todo ello para evitar intrusiones de terceros, mientras que cuando se realiza un despliegue en la nube dependiendo del modelo *cloud* que se utilice, el tema de seguridad puede ser gestionado por el proveedor. En relación con el rendimiento, por ejemplo, lo podríamos evidenciar cuando al realizar una transacción bancaria en línea o cuando deseamos obtener el reporte desde una aplicación disponible en la *web*, el tiempo de respuesta de dicha operación no supera los 30 segundos. Estos conceptos y ejemplos adicionales, los revisaremos a detalle en el capítulo 4 donde explicaremos la Calidad y atributos de calidad.

Despliegue local y despliegue en la nube son conceptos que serán de utilidad en la presente asignatura, por lo tanto, le invito a revisar algunas definiciones, ventajas e inconvenientes en [Software on premises vs. cloud: ventajas y desventajas](#) donde se describen las diferencias entre estos dos modelos de despliegue.

Figura 5.
UI vs UX y Despliegue local vs Cloud



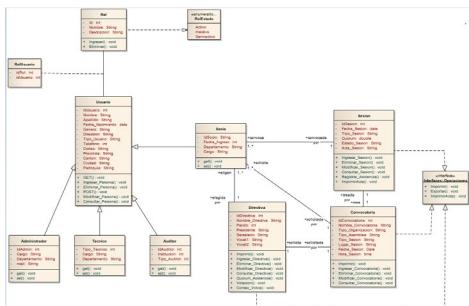
Nota. Tomado de *What is Cloud ERP Software? Benefits & System Options [Ilustración]*, por Tucakov, D., 2020, [Phoenixnap](#). CC BY 2.0

Para entender mejor los conceptos de arquitectura de software propuestos por los autores, le invito a analizar el siguiente escenario: “*en el sistema de matrícula en línea de UTPL admite una serie de transacciones diferentes para que los clientes realicen su inscripción o prematrícula, la actualicen, la cancelen, calculen los costos, realicen el pago, etc.*”. ¿Cómo piensa que lo hace, a través de qué dispositivos electrónicos puede interactuar con el sistema?, ¿el sistema interactuará con algún servicio, sistema o subsistema?, ¿cómo piensa que está desplegada la aplicación de forma local o en la nube?, ¿qué piensa usted?, le invito a revisar una posible respuesta.

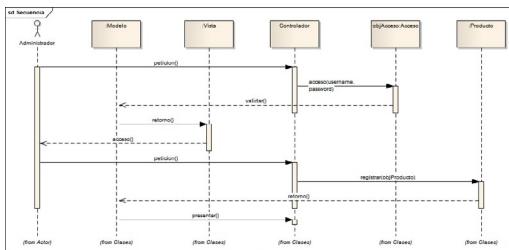
Las **propiedades de calidad** del sistema (*cómo lo hace*) incluyen el tiempo de respuesta promedio para una transacción bajo una carga específica, el rendimiento máximo que el sistema puede soportar, el número de peticiones que acepta el sistema, la disponibilidad del sistema y el tiempo requerido para reparar defectos. La **estructura estática** del sistema se representa de forma semiformal usando UML (Unified Modeling Language) donde se especifica las clases que serán usadas para implementar el estilo o patrón arquitectónico y la **estructura dinámica** permitirá visualizar a través de los diagramas de secuencia, actividades y comunicación, el modelo de solicitud / respuesta cuando por ejemplo se requiera explicar el escenario donde un cliente realiza operaciones sobre los módulos o funcionalidades de la aplicación (ver figura 6).

Figura 6.

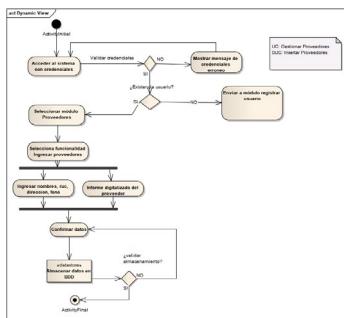
Representación de estructuras estáticas y dinámicas a través de diagramas UML



Estructura estática – Diagrama de clases



Estructura dinámica – Diagrama de secuencia



Estructura dinámica – Diagrama de actividades

Nota. Argüello M., & Guamán D., 2023

¿Se comprendió el escenario que se propuso como ejemplo? Se puede evidenciar primero que de una inquietud puntual podemos empezar a descomponer el problema o pensar en una posible solución y, en segundo

lugar, podemos evidenciar que la mayoría de los sistemas de software se basan en principios arquitectónicos que permiten a los componentes de hardware y software, los subsistemas y sus relaciones interactuar juntos para proporcionar una o muchas funcionalidades a los clientes o usuarios del sistema.

1.3. Importancia de la arquitectura de software

El diseñar y construir distintos tipos de software a pequeña, mediana y gran escala, nos conduce primeramente a entender las necesidades de los clientes, de las empresas y, al mismo tiempo, nos sugiere gestionar cuidadosamente las limitaciones presupuestarias, de recursos económicos y tecnológicos y de tiempo.

Usar una arquitectura para la construcción de software es importante, porque ayuda a prevenir muchos de los problemas relacionados con calidad que enfrentan actualmente los proyectos y productos de software. Si usted como parte de la formación profesional ha construido software, seguramente ha evidenciado que un proyecto de software no se trata solamente de tener un código limpio o buen código fuente que haga uso de estándares y buenas prácticas, sino que también a veces es necesario alejarse de la programación por un momento para tener claro el contexto en general que incluye documentación estandarizada, hardware, software y datos de forma integral, lo que nos lleva a pensar en su arquitectura.

Haciendo una analogía entre construir algo (casas, edificaciones) y el software, pensemos por un momento en lo siguiente: ¿qué se requiere para construir una casa para una mascota?, ¿qué se requiere para construir una casa para una familia? , y ¿qué se requiere para construir un edificio? (Ver figura 7).

Figura 7.

Arquitectura de software analogías



Diseñar un edificio

- Construido de manera más eficiente, oportuna por un equipo multidisciplinario teniendo en cuenta las características internas y externas.
- Requiere modelado detallado de planos y análisis estructurales.
- Proceso y etapas bien definidos.
- Herramientas de diseño y análisis estructural especializadas.



Diseñar una casa

- Construido de manera más eficiente y oportuna por un equipo.
- Requiere modelado y planos.
- Proceso bien definido.
- Herramientas de diseño especializadas.



Diseñar una casa para una mascota

- Puede ser construido por una persona.
- Requiere de un modelo mínimo.
- Proceso simple.
- Herramientas o materiales básicos.

Nota. Argüello M., & Guamán D., 2023

Para construir ese algo (edificaciones) es importante siempre conocer:

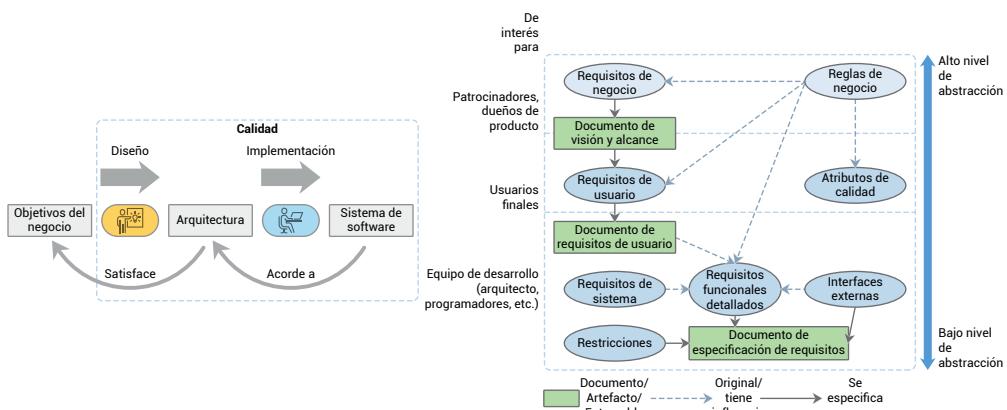
- Elementos principales, composición, orden y planificación.
- Propiedades de las estructuras/componentes o materiales que se utilizan.
- El rol de los actores (arquitecto, ingeniero eléctrico, etc.).
- Procesos y diseños/modelos que garanticen un buen resultado y de calidad.
- Tipos de edificación y características que deben cumplir.

Si relacionamos el contexto anterior con el *software*, la importancia de la arquitectura radica en que debemos tener en cuenta los requisitos de negocio, usuario y sistema para diseñar e implementar *software*

considerando atributos de calidad (ver figura 8). Esto quiere decir que podría existir escenarios donde un sistema de software inusual de alta calidad tenga un diseño arquitectónico pobre. Si no pensamos en la importancia de la arquitectura de software, el resultado final es algo que normalmente parece una gran bola de nieve llena de problemas que se derrumbará a corto o mediano plazo.

Figura 8.

El sistema se implementa acorde a una arquitectura que satisface los objetivos de negocio a partir del cual se generan documentos o artefactos



Nota. Argüello M., & Guamán D., 2023

Seguramente usted ha desarrollado software, a pequeña o gran escala, y cuando le han preguntado ¿funciona?, su respuesta seguramente ha sido ¡Por su puesto en mi máquina funciona y con 3 clientes concurrentes funciona muy bien!, sin embargo, cuando le preguntan acerca de su estructura o documentación asociada, seguramente su respuesta es el software no tiene una estructura estática y peor aún una arquitectura base, por lo cual se podría mencionar que su software no es de calidad.

La importancia de la arquitectura se evidencia cuando a nivel de diseño e implementación evitamos defectos secundarios, como por ejemplo, que cuando se implementa en un ambiente de producción o cuando se incrementa el número de usuarios concurrentes, su software no soporta nuevos requisitos funcionales, es demasiado lento, inseguro, frágil, inestable, difícil de implementar, mantener, cambiar, extender, etc.

Tenga presente que, la arquitectura de software es importante por razones que van desde lo no técnico hasta lo técnico. Entre ellas, por ejemplo,

la inclusión de propiedades de calidad y documentación que permite la comunicación entre las partes interesadas (**Stakeholders**) quienes deben entender y comprender la solución a un problema dentro de un contexto.



Finalmente, recuerde que una arquitectura, al ser resultado de un conjunto de decisiones de diseño, permitirá ser la base para representar en alto nivel el diseño de los prototipos iniciales que permitan su posterior codificación, implementación, integración y despliegue, con el objetivo de evaluar si los sistemas son funcionales, mantenibles y evolutivos.

1.4. Beneficios de la arquitectura de software

El objetivo de una arquitectura es identificar los requisitos que afectan la estructura y posterior comportamiento del *software*. Por lo tanto, una arquitectura debe verse como el resultado de un conjunto de decisiones de diseño en lugar de un conjunto de elementos arquitectónicos (componentes y conectores, que los estudiaremos más adelante). Una buena arquitectura reduce los riesgos asociados con la creación de una solución técnica que se implementa a nivel de programación.

Uno de los beneficios que nos brinda una arquitectura de *software*, es poder construir aplicaciones que aborden todas las inquietudes o necesidades relevantes dadas por los clientes, que se puedan implementar en la infraestructura de TI elegida (despliegue local o *cloud*) y que proporcione resultados que cumplan con las metas y objetivos del negocio.

Para lograr esto, cuando se piensa en una arquitectura de *software*, se sugiere tener presentes algunas preguntas o preocupaciones de alto nivel a las cuales debemos dar respuesta:

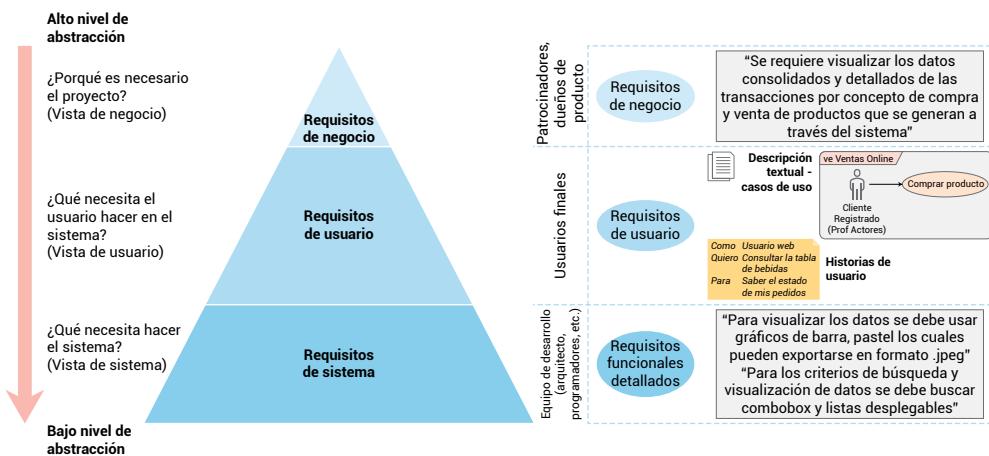
- ¿Cómo utilizarán los usuarios el sistema o la solución *software*?
- ¿Cómo se implementará y administrará el sistema en ambientes de desarrollo, pruebas y producción?

- ¿Cuáles son los requisitos no funcionales para la inclusión de seguridad, rendimiento, concurrencia, internacionalización y configuración?
- ¿Cómo se puede diseñar el sistema para que sea flexible, escalable y se pueda mantener a lo largo del tiempo?
- ¿Cuáles son las tendencias arquitectónicas que podrían afectar su aplicación ahora o después de su implementación o despliegue?

Estas preguntas y otras preguntas que usted defina, son de utilidad porque permiten abordar los tres tipos de requisitos que se muestran en la figura 9, entre los que constan requisitos de negocio, de usuario, de sistema, según Memory Jogger (2005), a través de los cuales se puede tomar decisiones a nivel de diseño, implementación y las compensaciones inherentes a la calidad.

Le invito a que revise el [anexo 1](#) donde se propone un ejemplo de un estudio de caso o escenario a partir del cual y se llevan actividades de análisis para entender el dominio del problema y especificar los requisitos a través del documento de especificación de requisitos ([ver anexo 2](#)).

Figura 9.
Tipos de requisitos



Nota. Argüello M., & Guamán D., 2023

Conocer y entender las necesidades dadas por los clientes, transformarlas y documentarlas como requisitos, apoyándonos en artefactos como el documento de especificación de requisitos, historias de usuario o casos

de uso, permitirán que la arquitectura a nivel de diseño e implementación tenga algunos beneficios entre los que destacan:

- Exponer de manera formal y documentada la estructura del sistema en alto nivel, ocultando algunos detalles de su implementación.
- Tener una visión clara y una hoja de ruta que el equipo de desarrollo utilizará para la construcción del *software*.
- Consistencia de enfoque y estándares, lo que lleva a una base de código bien estructurada.
- Abordar los requisitos de varias partes interesadas, haciendo uso de un proceso guiado que permita responder preguntas relacionadas con decisiones de diseño importantes, requisitos no funcionales, limitaciones y otras preocupaciones transversales.
- Hacer uso de fases, actividades y tareas como parte de un proceso que permite definir y diseñar una arquitectura de *software*.
- Utilizar un marco de referencia para identificar y mitigar los riesgos a nivel de proyecto y producto de *software*.
- Entender las alternativas de despliegue que puede tener una aplicación para que cumpla con los atributos de calidad establecidos en la fase de análisis y diseño.

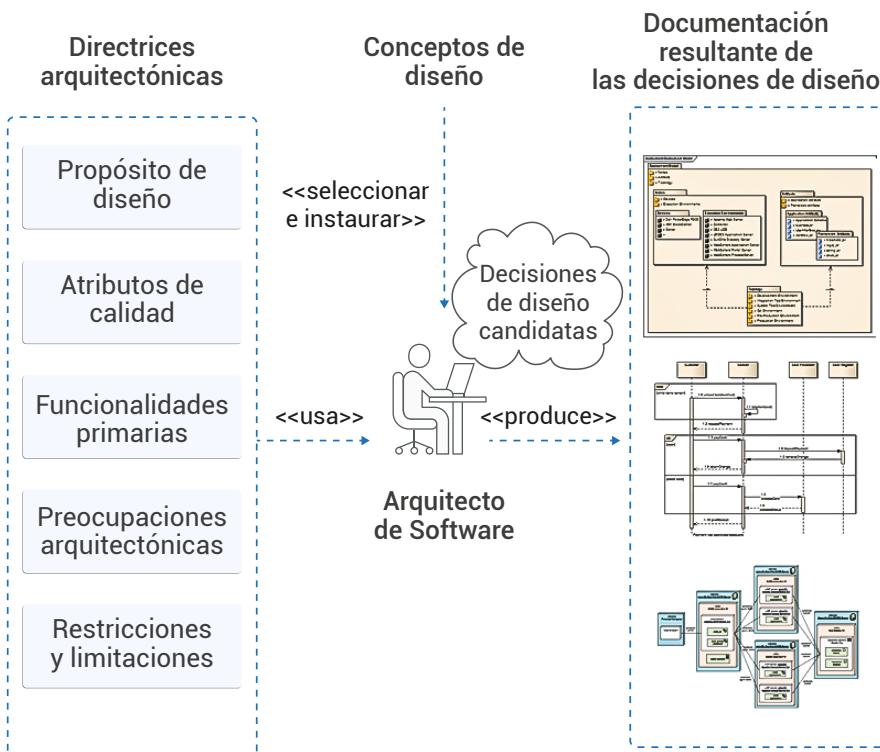
1.5. El rol del arquitecto de *software*

Ahora que conocemos el concepto de arquitectura de *software*, su importancia y algunos de los beneficios, se ha preguntado ¿quién o quiénes son los encargados de definir y diseñar una arquitectura? Seguramente su respuesta fue el **arquitecto de software**. Antes de considerar cuál y cómo realiza su trabajo un arquitecto, debemos comprender exactamente cuáles son sus responsabilidades, dónde están sus límites, qué áreas debe delegar a otros y cómo trabaja junto con los otros miembros del equipo para garantizar una entrega de *software* exitosa. La línea entre ser un **desarrollador de software** y ser un **arquitecto de software** es un poco compleja, ya que por lo general el arquitecto de software crea y ejecuta planes de diseño y desarrollo de *software*, incluido características de

calidad, restricciones y limitaciones arquitectónicas, reglas de negocio, el presupuesto y los plazos de entregas del *software*, mientras que el desarrollador de *software* sigue el plan de desarrollo de *software* e implementa los módulos a través de un lenguaje de programación realizando las pruebas unitarias del módulo desarrollado.

El **arquitecto** es responsable de diseñar, documentar y liderar la construcción de un sistema que satisfaga las necesidades de todos sus grupos de interés. El **arquitecto de software** tiene como función liderar el desarrollo de la arquitectura de *software* del sistema, que incluye promover y crear soporte para las decisiones técnicas clave que limitan el diseño y la implementación general del proyecto. A diferencia de otros roles, la visión del arquitecto de *software* es de amplitud y no de profundidad. Amplitud porque el arquitecto debe tener conocimientos de infraestructura, de TI, de planificación, de nuevas tecnologías, de toma de decisiones, de relaciones humanas, porque debe receptar y comunicar las decisiones de diseño. Cuando se menciona profundidad es porque el arquitecto no solo se enfoca en utilizar una tecnología para llevar a cabo la implementación de una arquitectura (ver figura 10).

Figura 10.
Rol del arquitecto de software



Nota. Argüello M., & Guamán D., 2023

Entre algunas de las características que posee un arquitecto de software y que usted podría revisarlas en bibliografía básica y complementaria constan las siguientes:

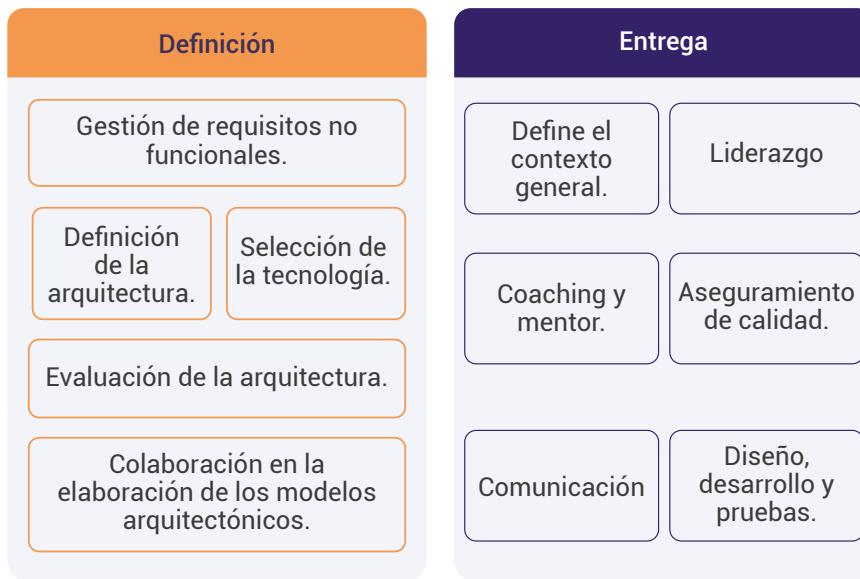
- Identificar e involucrar a las diferentes partes interesadas, con el objetivo de conocer estratégicamente a las personas que le brindarán los insumos necesarios y apoyarán en la definición y diseño de una arquitectura candidata.
- Comprender y captar los **concerns** (intereses, inquietudes o necesidades) dadas por las partes interesadas.
- Definir y diseñar una arquitectura que cubra todas o gran parte de los **concerns** dados por las partes interesadas.
- Asumir con liderazgo la definición y diseño de la arquitectura de un producto o sistema de software.

El arquitecto tiene como responsabilidades desarrollar y mantener una visión de alto nivel de los elementos principales del producto o sistema, que posteriormente se traduce en un diseño detallado, codificado, probado y desplegado. Si bien el término **desarrollador de software** es de fácil comprensión, **arquitecto de software** a veces es complejo definirlo, debido a que este debe pensar en la función de la arquitectura de software, del **panorama general** y, a veces, esto significa no pensar solamente en el código sino en el sistema de software en general (*hardware, software, procesos, infraestructura*).

Otro de los roles del arquitecto es asegurarse de que el panorama general que desarrolle sea el adecuado para un contexto. Entonces aquí es importante mencionar que cada problema o necesidad de los clientes tiene una serie de posibles soluciones arquitectónicas (candidatas) y cada arquitectura tiene una serie de posibles representaciones a través de documentación y diseños (diagramas). Ante ello, el rol del arquitecto es seleccionar y proponer una arquitectura que se ajuste al contexto y luego documentar esa arquitectura de forma tal que sea entendida por su equipo técnico y no técnico de trabajo y sobre todo que pueda ser comprendida por los **Stakeholders (partes interesadas)**. En la figura 11 se visualiza como el arquitecto de software debe tener un conocimiento amplio en diversos temas relacionados con las áreas propuestas en el SWEBOk que le permitan tomar decisiones de diseño importantes.

Figura 11.

Rol del arquitecto de software para la definición y entrega de software



Nota. Argüello M., & Guamán D., 2023

Alcanzar el rol de arquitecto de software, no es una tarea sencilla que se logra de la noche a la mañana, para ello hay que trabajar y tener presente que un arquitecto de software, entre otras cosas, necesita interactuar con clientes, gerentes de producto y personas involucradas en el proceso de desarrollo para proporcionar modelos y diseños iniciales que se pueden construir con diversas tecnologías y sobre todo tener habilidad técnica y habilidades blandas (*soft skills*), términos que le invito a revisarlos en [Soft Skills for Developers – The Ultimate Guide](#). En un proceso de desarrollo de software, un arquitecto de software tiene que revisar constantemente la documentación (diagramas) y el código fuente para garantizar la calidad del diseño, evitando su *degradación* y *complejidad*. Esto generalmente requiere un trabajo práctico en términos de desarrollo de prototipos, automatización del proceso de desarrollo, contribución de buenas prácticas en escritura de código, evaluación de tecnologías, entre otras.

Finalmente, el papel de un arquitecto de software incluye el trabajo colaborativo con cierto grado de humildad, esta colaboración también permite al arquitecto familiarizarse con las habilidades e intereses del equipo y compartir sus conocimientos y experiencias con el resto del equipo. Se menciona humildad porque se requiere garantizar que se

escuche a todo el equipo, ya que pueden tener experiencia o conocimiento más específico para el problema bajo estudio y habrá que aceptarlo.

Entonces la pregunta en este momento es usted, qué rol tiene o considera que tiene, ¿analista, diseñador, documentador, arquitecto, desarrollador, administrador de base de datos, gerente?, le invito por un instante a hacerse las siguientes preguntas, ¿qué rol desea o desearía tener?, ¿se está formando para ello?, ¿qué habilidades blandas o *soft skills* ha desarrollado o quiere desarrollar?

Muy bien, hemos culminado nuestra semana 1 de estudio y como parte de los resultados de aprendizaje, Ud. debería estar en capacidad de responder a las preguntas ¿qué entiende por arquitectura de software?, ¿en qué fase del proceso de desarrollo de software se propone y diseña una arquitectura software?, ¿por qué es importante tener en cuenta una arquitectura software?, y ¿cuáles son sus beneficios? Es necesario recordar algunos conceptos de ciclos anteriores relacionados con Fundamentos de Ingeniería de Software y Modelado, ya que los incluiremos de a poco en nuestro estudio, además no olvide revisar bibliografía básica y complementaria para reforzar los temas estudiados.

Lecturas en bibliografía básica, complementaria y recursos educativos abiertos

- Revise la bibliografía complementaria para reforzar sus conocimientos de ingeniería de software e identificar qué actividades y artefactos del proceso de software son requeridos en la arquitectura de software.
- **Libro de Somerville** – (Capítulo 2 - Procesos de software. Subtemas 2.1 Modelos de proceso de software, 2.1.2 Desarrollo incremental, 2.2.1 Especificación del software, 2.2.2 Diseño e implementación del software, 2.2.3 Validación del software, 2.2.4 Evolución del software).
- **Libro de Presman** – (Capítulo 9 – Diseño de la arquitectura. Subtemas 9.1.1 ¿Qué es la arquitectura?, 9.1.2 ¿Por qué es importante la arquitectura?).
- **Libro a rquitectura de s oftware, conceptos y ciclo de desarrollo** – (Capítulo 1 - La arquitectura y el desarrollo de software – subtemas 1.1 Visión general del desarrollo de los sistemas de software,

1.2 Definición de arquitectura de software, 1.5 Beneficios de la arquitectura, 1.6 El rol del arquitecto).

- **DD3_Arquitecturas de software** - Importancia de la arquitectura de software (p áginas 6,7).
- Revise el anexo 1 donde se propone un ejemplo de un estudio de caso o escenario referente a un dominio del problema.
- Revise el anexo 3 donde luego de entender y analizar el dominio del problema, se utiliza el documento de especificación de requisitos para transformar las necesidades dadas por las partes interesadas en requisitos funcionales y no funcionales.

Como parte de nuestra formación, le invito a llevar a cabo las actividades de aprendizaje recomendadas para esta semana.



Actividades de aprendizaje recomendadas

Con base a la revisión de la bibliografía básica, complementaria y recursos educativos abiertos sugeridos para su estudio y los que Ud. puede adicionar, por favor responda las siguientes preguntas que se proponen utilizando sus propias palabras.

1. ¿Cómo define usted a la arquitectura de software?
2. ¿Por qué es importante la arquitectura de software?
3. Cuando se requiera desarrollar aplicaciones software pequeñas, medianas, grandes cuyo tamaño por lo general se mide en líneas de código, ¿bajo qué escenarios es necesaria una arquitectura de software?
4. ¿Con qué fases del CVDS se asocia o relaciona la arquitectura software? Analice la figura 2 y argumente su respuesta.

Nota. conteste las actividades en un cuaderno de apuntes o en un documento Word.



Semana 2

Una vez que ha dado respuesta a las preguntas propuestas en la actividad recomendada de la semana 1, ha estudiado los conceptos y conoce la importancia de la arquitectura de software, es momento de estudiar algunos conceptos que son parte del estándar ISO/IEC/IEEE 42010:2011, Systems and software engineering - Architecture description ISO/IEC/IEEE 42010 (2011), y que le permitirán comprender términos tales como elementos arquitectónicos, *stakeholders* (interesados), *concerns* (interés, inquietud o responsabilidad). Además, en esta semana le invito a comprender la diferencia entre arquitectura lógica vs. arquitectura física, arquitectura vs. diseño.

1.6. Arquitectura vs. Diseño

Si recordamos otra de las áreas que el SWEBOK propone es el diseño de software, en dicha área el objetivo es representar de forma gráfica (modelos y diagramas) y documentar de forma textual los escenarios que permitirán implementar y evaluar la arquitectura de software.

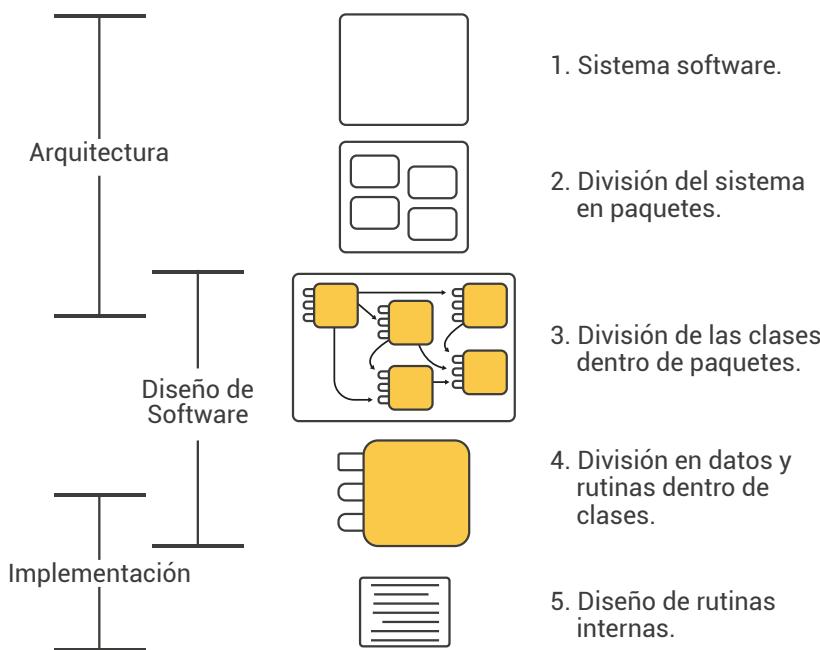
Seguramente usted ha desarrollado diversas aplicaciones de software hasta el momento, todas de forma diferente o con cierta similitud si hablamos de tecnologías, plataformas de implementación y enfoques de diseño para alcanzar un objetivo. Sin embargo, al diseñar su sistema de software, de todo el conjunto de opciones eligió solo una en el espacio de potenciales opciones, ¿verdad? Si lo hizo de esta manera, hemos utilizado la esencia del diseño, la cual trata de reducir el espectro de soluciones a una solución que funcione acorde al **contexto** en el que está trabajando o se desea trabajar.

En su libro *Design It!: From Programmer to Software Architect*, Michael Keeling (2017), menciona que “como sustantivo, **diseño** es la estructura nombrada (aunque a veces innombrable) o comportamiento de un sistema, cuya presencia contribuye a la resolución de una fuerza o fuerzas en ese sistema”. Además, menciona que la **arquitectura de software** muestra la estructura del sistema y oculta los detalles de implementación, centrándose en cómo los componentes del sistema interactúan entre sí. Para entender el concepto dado por Keeling, se puede mencionar que la

arquitectura y diseño se tratan correctamente como dos etapas separadas en el proceso de desarrollo del *software*, donde la arquitectura es general y la especificidad se conoce en el diseño derivado de dicha arquitectura (ver figura 12).

Figura 12.

Arquitectura vs Diseño de software



Nota. Argüello M., & Guamán D., 2023

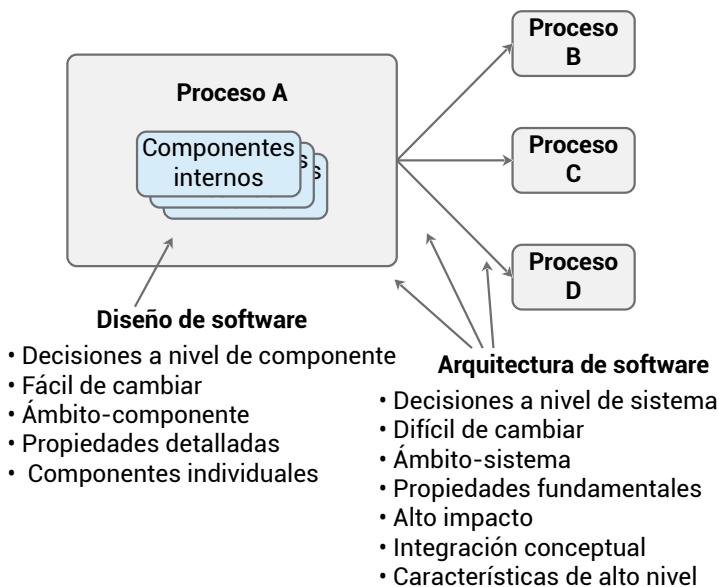
La **arquitectura requiere de una vista de alto nivel** y una experiencia considerable por parte del arquitecto de *software*. La arquitectura de un sistema es su esqueleto, es el nivel más alto de abstracción (estructura, comportamiento, interacción, atributos de calidad) de un sistema que ayuda a responder preguntas tales como ¿qué tipo de almacenamiento de datos está presente?, ¿cómo interactúan los componentes entre sí o con otros componentes?, ¿qué sistemas de recuperación existen?, ¿cuál es la mejor opción para el despliegue de los componentes? Entre otras. Dicho de otro modo, la arquitectura es considerada como un plan enfocado en la **optimización global del software**, que restringe el diseño de *software* para evitar errores conocidos a nivel de implementación tecnológica (*hardware* y *software*) y alcanzar los objetivos requeridos por el cliente, una empresa o una organización.

Por otra parte, el **diseño** de software, consiste en diseñar los módulos / componentes individuales, es decir, responder preguntas tales como ¿cuáles son las responsabilidades o funciones del módulo x?, ¿cuáles son las responsabilidades de la clase Y?, ¿cómo interactúa una interfaz con una o varias clases?, ¿qué puede hacer el módulo y qué no?, ¿qué patrones de diseño se pueden utilizar?, entre otras. **Diseño** es un plan que **se concentra en la implementación y optimización local**, que a menudo profundiza en suficiente detalle para que los desarrolladores implementen software consistente y profesional (ver figura 13).

Parte de aquellos detalles incluye la forma general o estilo del sistema de software (por ejemplo, cliente-servidor, basado en web, móvil nativo, distribuido, servicios, asíncrono vs. síncrono, etc.), la estructura del código dentro de las distintas partes del sistema de software (por ejemplo, si el código está estructurado como componentes, capas, características, puertos y adaptadores, etc.), la elección de tecnologías (es decir, lenguaje de programación, plataforma de implementación, etc.), la elección de marcos de desarrollo (por ejemplo, marco web MVC, marco de persistencia / ORM, etc.), la elección del enfoque / patrones de diseño y atributos de calidad asociados (por ejemplo, el enfoque del rendimiento, escalabilidad, disponibilidad, etc.).

Figura 13.

Características arquitectura vs Diseño de software



Nota. Argüello M., & Guamán D., 2023

En resumen, podríamos mencionar que, **la arquitectura de software** se enfoca más en las decisiones de diseño que dan forma a todo el sistema, mientras que **el diseño de software** hace énfasis a nivel de módulo / componente / clase que permite la implementación del sistema. Con esto se podría afirmar que, **toda arquitectura es un diseño, pero no todo diseño es una arquitectura**.

1.7. Elementos arquitectónicos

El término elemento arquitectónico hace referencia a las piezas fundamentales a partir de las cuales se construyen los sistemas de software. La naturaleza de un elemento arquitectónico depende mucho del **contexto y tipo de sistema**.

Perry & Wolf (1992), mencionan que “los elementos arquitectónicos incluyen elementos de procesamiento, elementos de datos y elementos de conexión. Los elementos de procesamiento son aquellos que realizan transformaciones en los datos, los elementos de datos son aquellos que contienen la información que se utiliza y transforma, y los elementos de conexión que mantienen unidas las diferentes piezas de la arquitectura”.

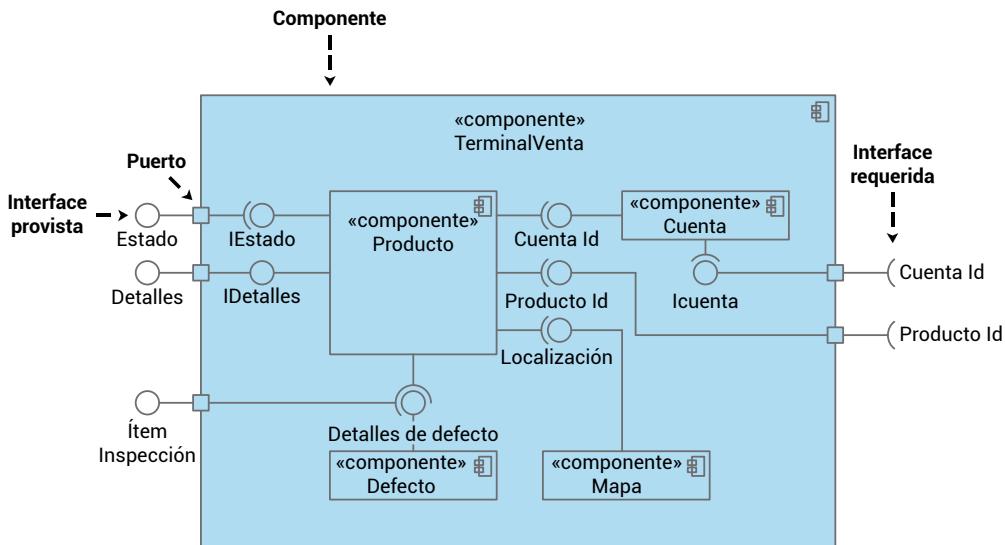
Los elementos arquitectónicos se identifican a nivel de diseño, por ejemplo cuando se representa el diagrama de componentes, donde deben existir conectores como interfaces provistas e interfaces requeridas, las cuales a través de su configuración permiten conectar a dichos componentes. Entonces, ¿qué son los componentes, conectores y puertos? Revisemos estos conceptos a continuación, ya que los utilizaremos para representar gráfica y formalmente una arquitectura de software.

Elementos arquitectónicos – Componentes:

El concepto de componente es la base de la arquitectura de software y es el concepto que las ADL (Architecture Description Language, por sus siglas en inglés) comparten por excelencia. Un componente es un elemento computacional que tiene un alto nivel de encapsulación y solo es posible interactuar con él a través de sus interfaces (provistas o requeridas), por lo tanto, un componente puede ser *simple* o *complejo*. En otra definición se considera a los componentes como los bloques de construcción los cuales conforman las partes de un sistema. En términos de Component-Based Software Development (CBSD, por sus siglas en inglés) un componente se puede definir desde el punto de vista de implementación como un paquete de código. Desde un punto de vista más general, una componente es un artefacto de software que se ha desarrollado para ser reutilizable, lo que implica que la podría utilizar en otro sistema, subsistema o servicio (ver figura 14).

Figura 14.

Representación de un componente y sus elementos (interfaces y puertos)



Nota. Tomado de *What is Component Diagram? [Ilustración]*, por Visual Paradigm, 2023, [Visual Paradigm](#). CC BY 2.0

Entre los principios que se debe tener en cuenta al momento de definir y diseñar un componente constan: *encapsulación*, *abstracción* y *modularidad*. A nivel de implementación con lenguajes de programación, los componentes pueden ser representados como módulos, clases, objetos o un conjunto de funciones relacionadas, según Len Bass, Paul Clements y Rick Kazman (2012).

Elementos arquitectónicos – Conectores:

Los conectores surgen de la necesidad de separar la **interacción** del **cálculo**, con ello se logra que los componentes sean más reutilizables y modulares. El uso de conectores permite mejorar el nivel de abstracción de la descripción de la arquitectura de software.

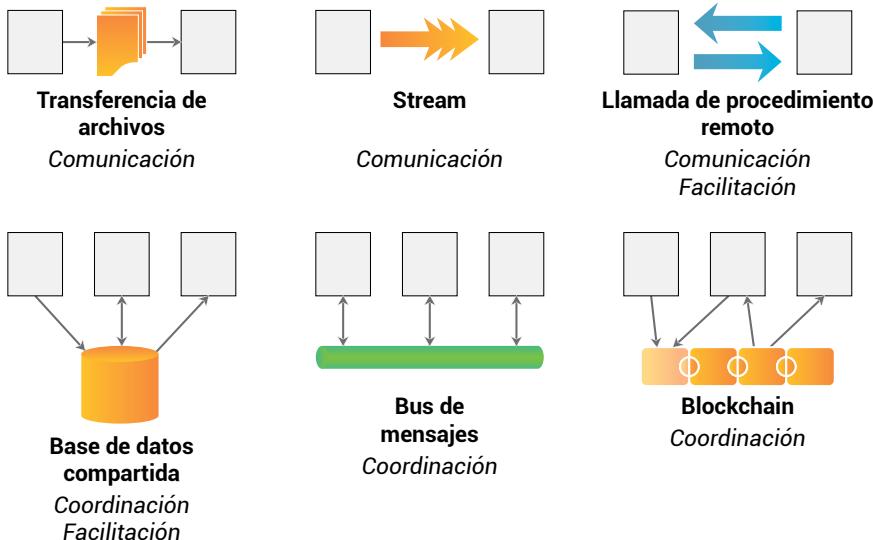
Los conectores ofrecen los servicios de: **comunicación** para transferir datos, **coordinación** para transferir el control, **facilitación** para habilitar y optimizar la interacción entre componentes y de **conversión** para ajustar la interacción entre interfaces incompatibles.

En la figura 15 se visualiza algunos ejemplos a través de los cuales los componentes se comunican haciendo uso de conectores para

permitir la transferencia de archivos, como bus de mensajes, llamada de procedimiento remoto, transferencia de archivos, entre otros.

Figura 15.

Ejemplos de tipos conectores de software



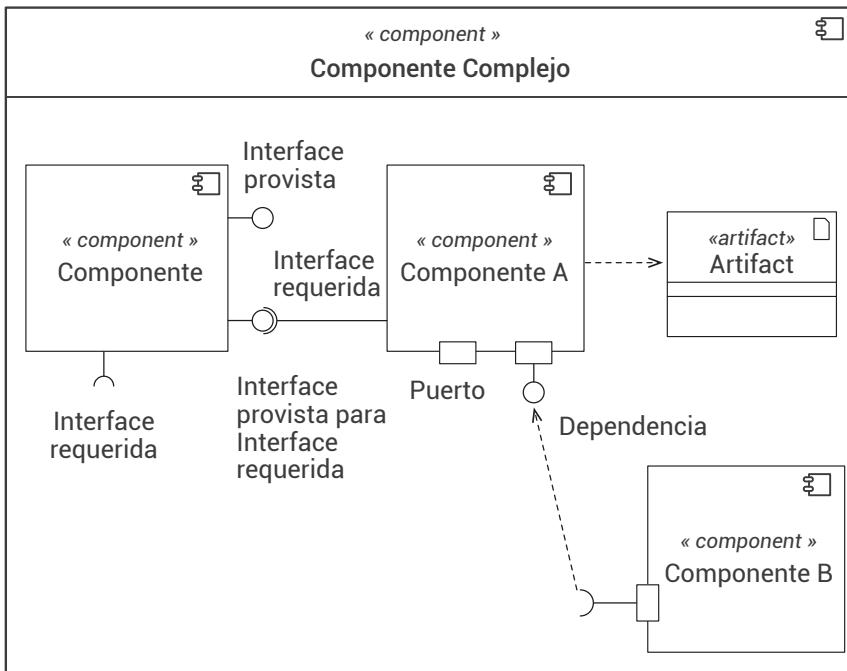
Nota. Tomado de *The Blockchain as a Software Connector [Ilustración]*, por Pautasso, C., 2016, [Slideshare](#). CC BY 2.0

Elementos arquitectónicos – Puertos:

A los puertos se los considera como puntos de interacción de elementos arquitectónicos (componentes y conectores). Por lo general, los puertos a nivel de implementación son de tipo *Interface + Played_Role* (*Rol que juega*). A nivel de implementación, una Interface es un conjunto de servicios publicados y también describe la firma de los servicios que se pueden invocar o solicitar a través de esa interface. En la figura 16 se visualiza una componente compleja dentro de la cual existe una relación entre componentes, conectores e interfaces.

Figura 16.

Componentes, interfaces y puertos a nivel de diseño



Nota. Argüello M., & Guamán D., 2023

A nivel de diseño arquitectónico, los componentes y conectores incorporan decisiones de cómo el sistema se estructura a través de un conjunto de elementos que tienen comportamiento en tiempo de ejecución (componentes) e interacciones (conectores). Por ejemplo, dependiendo del estilo o patrón arquitectónico, se hará uso de un conjunto de componentes para representar y definir las principales unidades de cálculo que usan los servicios, capas (layers), servidores, pipelines u otros. Por otro lado, los conectores son los medios de comunicación entre componentes, que permiten especificar por ejemplo llamada-retorno, operadores de sincronización de procesos, canalizaciones u otros.

Seguro usted se ha de preguntar ¿cuándo o dónde utilizaremos los elementos arquitectónicos a nivel de diseño y/o implementación de arquitectura de software? Si recordamos, uno de los beneficios de la arquitectura de software es la documentación, por tanto, el representar en vistas estructurales o dinámicas los componentes, conectores y puertos nos ayudarán a responder preguntas tales como:

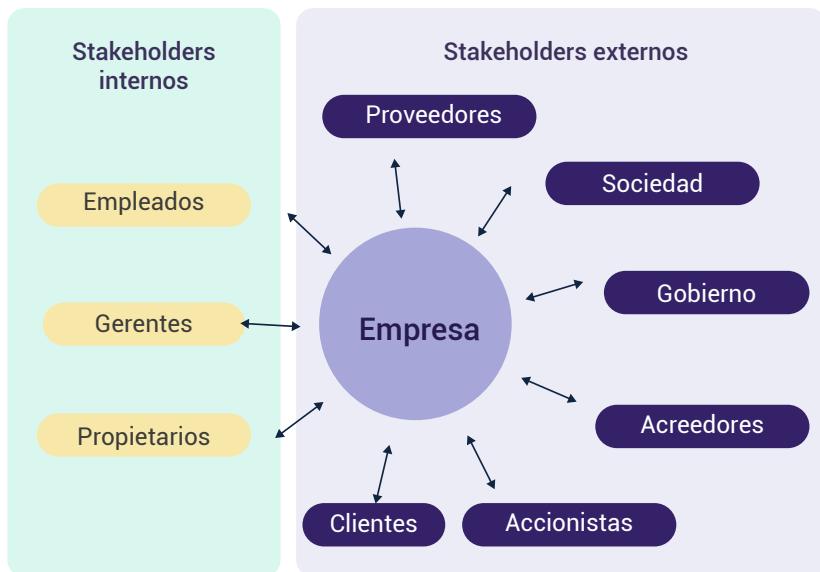
- ¿Cuáles son los principales componentes estructurales que permitirán representar y entender el comportamiento en tiempo de ejecución de una arquitectura del sistema?
- ¿Cómo es la interacción entre componentes a través de sus relaciones en tiempo de ejecución?
- ¿Cuáles son los principales almacenes de datos que comparten los componentes?
- ¿Qué partes del sistema o componentes tienen la capacidad de replicarse, reutilizarse o evaluarse?
- ¿Qué tipos de conectores y puertos se usan para intercambiar datos a través de los componentes del sistema?
- ¿Qué componentes del sistema pueden funcionar en paralelo?
- ¿Puede cambiar la estructura del sistema a medida que se ejecuta? y, de ser así, ¿cómo o dónde cambiaría?

En resumen, los elementos arquitectónicos (componentes, conectores, puertos) nos ayudan a formalizar, describir y evaluar una arquitectura de software, utilizando para ello algún modelo arquitectónico como por ejemplo 4+1 o C4. Y es que, para comprender y usar un modelo, es importante tener en cuenta que la gestión de un proyecto y producto de software han sido impulsados por las necesidades de las personas y organizaciones interesadas en el sistema, a las cuales se conoce como partes interesadas (*Stakeholders*).

1.8. Partes interesadas (*stakeholders*)

Una parte interesada es cualquier persona que tenga interés en el éxito del sistema, por ejemplo, el cliente, los usuarios finales, los desarrolladores, el director del proyecto, los encargados del mantenimiento e incluso los que comercializan el sistema, entre otros. Acorde al estándar IEEE 1471-2000 (Hilliard, 2000), un interesado, en el contexto de la arquitectura de software, se define como una persona, grupo o entidad con un interés, inquietud o responsabilidad (**concerns**) sobre la realización de la arquitectura (ver figura 17).

Figura 17.
Stakeholders en arquitectura de software



Nota. Argüello M., & Guamán D., 2023

Como usted conoce, los sistemas no solamente se utilizan, sino que se deben construir, poner en funcionamiento, se deben validar, se deben realizar mantenimientos, generalmente evolucionan, se mejoran, y por supuesto hay que pagar por ellos. Cada una de estas actividades involucra a un grupo de personas que tiene sus propios intereses y necesidades que el sistema de software debe satisfacer.

¿Por qué es importante identificar y trabajar con los *Stakeholders*? la respuesta a esta pregunta es porque en los proyectos de desarrollo de software se requiere la inclusión de todas o al menos la mayoría de las partes interesadas. Dependiendo del contexto del sistema, cuando no se puede tener presente a todas las partes interesadas (por ejemplo, al desarrollar un nuevo producto), al menos se debe seleccionar algunas personas que representarán al grupo y expresarán sus **intereses, inquietudes o necesidades (concerns)**.

El término **concerns** (intereses, inquietudes, necesidades o responsabilidades) se utiliza por dos razones, la primera es porque forma parte del estándar ISO/IEC/IEEE 42010:2011, y segundo porque es apropiado en el proceso de definición y diseño de una arquitectura de software que lo estudiaremos más adelante. Cada actor/persona de

las partes interesadas tiene sus propios ***concerns*** que puede ser una necesidad, un requisito, un objetivo, intenciones o aspiraciones a partir de las cuales se define y diseña una arquitectura. Algunos de los *concerns* pueden ser similares a los de otros actores o también pueden resultar contradictorios.

Las responsabilidades o inquietudes que tienen cada actor de las partes interesadas, pueden ser diversas como, por ejemplo, el propósito del sistema, proporcionar un determinado comportamiento de un componente en tiempo de ejecución, configurar una pieza de *hardware* para que tenga un rendimiento en particular, que el sistema sea fácil de personalizar, lograr un corto tiempo de comercialización o un bajo costo de desarrollo, emplear de manera remunerada a programadores que tengan una especialidad en particular, que el sistema proporcione una amplia gama de funciones, la viabilidad de construir y desplegar el sistema, los riesgos e impactos del sistema, mantenibilidad y capacidad de evolución del sistema.

Para comprender el modelo conceptual de descripción arquitectónica, le invito a revisar de forma breve su concepto en el siguiente apartado, ya que este lo abordaremos con más detalle en las siguientes semanas.



Semana 3

1.9. Descripción arquitectónica – Modelo conceptual ISO/IEC/IEEE 42010

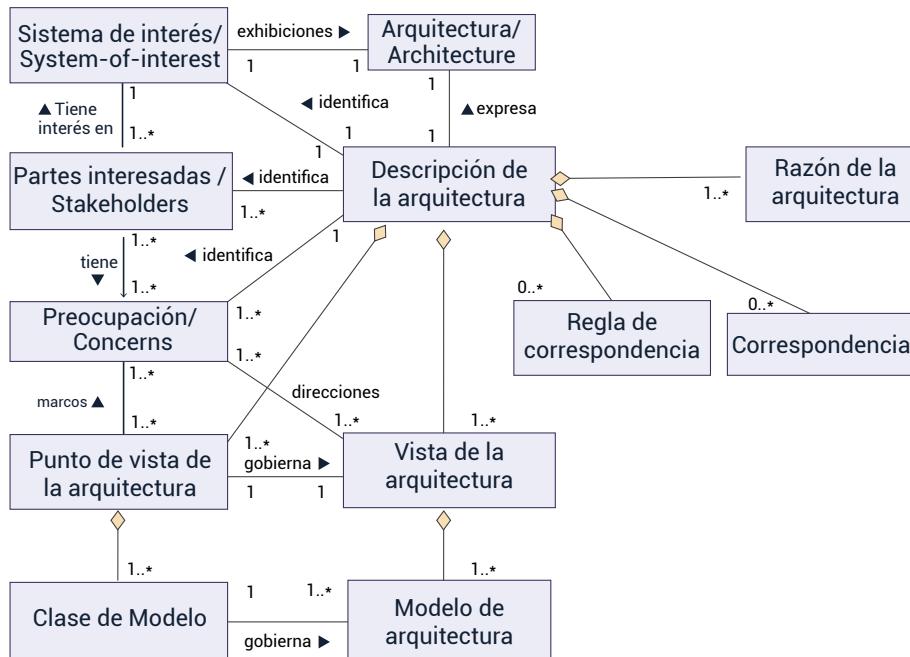
Definir y diseñar una arquitectura para un sistema de *software* puede resultar algo complejo. Parte del rol del arquitecto de *software* es elaborar documentación y/o modelos que permitan minimizar esta complejidad, con el objetivo de que el resto de personas puedan entenderla y comprenderla. Para ello, el arquitecto utiliza como base una Descripción Arquitectónica (AD, Architecture Description por sus siglas en inglés) a la cual se la define como un conjunto de productos que documentan una arquitectura de manera que sus partes interesadas (***Stakeholders***) pueden entender y visualizar que la arquitectura ha respondido a sus intereses o inquietudes (***concerns***).

Uno de los estándares que se utiliza para describir una arquitectura software es el estándar ISO/IEC/IEEE 42010 (ver figura 18). El proceso de descripción de una arquitectura software generalmente comienza con la identificación de las **partes interesadas (Stakeholders)** quienes tienen interés en que se diseñe y construya un sistema de software. Para ello las partes interesadas tienen sus propios intereses o necesidades (**concerns**). Para dar respuesta a esas preocupaciones, necesidades o intereses a las partes interesadas se debe mostrar de forma textual y gráfica la propuesta de diseño y arquitectura candidata, utilizando para ello **vistas arquitectónicas (Architecture view)** que servirán para que cada persona con un rol dentro del sistema pueda acorde a su **punto de vista (Architecture viewpoint)** entender y comprender los tipos de modelos (**Model Kind**) que se usan para representar las partes estructurales y de comportamiento de la arquitectura del sistema software (**Architecture model**). Las partes interesadas del sistema de interés (**System of Interest**) tienen necesidades o preocupaciones consideradas fundamentales para la arquitectura. Los sistemas de interés pueden ser productos y servicios de software, aplicaciones individuales, subsistemas, sistemas de sistemas, líneas de productos, familias de productos. Por lo tanto, la descripción arquitectónica identifica el sistema de interés cuya arquitectura está siendo expresada o documentada.

En el metamodelo se visualiza que las correspondencias (**Correspondence**) describen las dependencias entre los **elementos de la arquitectura** (partes interesadas, necesidades, vistas, etc.), así como entre las descripciones de toda la arquitectura.

Figura 18.

Metamodelo ISO/IEC/IEEE 42010



Nota. Tomado de *Architecture description ISO/IEC/IEEE 42010, por Systems and software engineering*.

En el metamodelo de la norma también se visualiza la **Arquitectura** (*Architecture*) a la que se define como los conceptos fundamentales o propiedades de un sistema (**System-of-interest**) en su entorno, materializados en sus elementos, relaciones y en los principios de su diseño y evolución. El **sistema** siempre se diseña y se construye para abordar los intereses, necesidades, preocupaciones, metas y objetivos (**Concerns**) de sus partes interesadas (**Stakeholder**).

Como podemos visualizar en el metamodelo, los fundamentos de la arquitectura de software son útiles para registrar las decisiones de arquitectura y el razonamiento detrás de ellas. Es por ello que el estándar no define descripciones de arquitectura específicas, vistas, puntos de vista, modelos o tipos de modelos, sino las propiedades que identifican los elementos que se pueden utilizar para su descripción.

La arquitectura de software, su descripción textual y gráfica son importantes porque permiten gestionar las diferentes vistas, puntos de vista y con ello controlar el desarrollo iterativo e incremental de un sistema

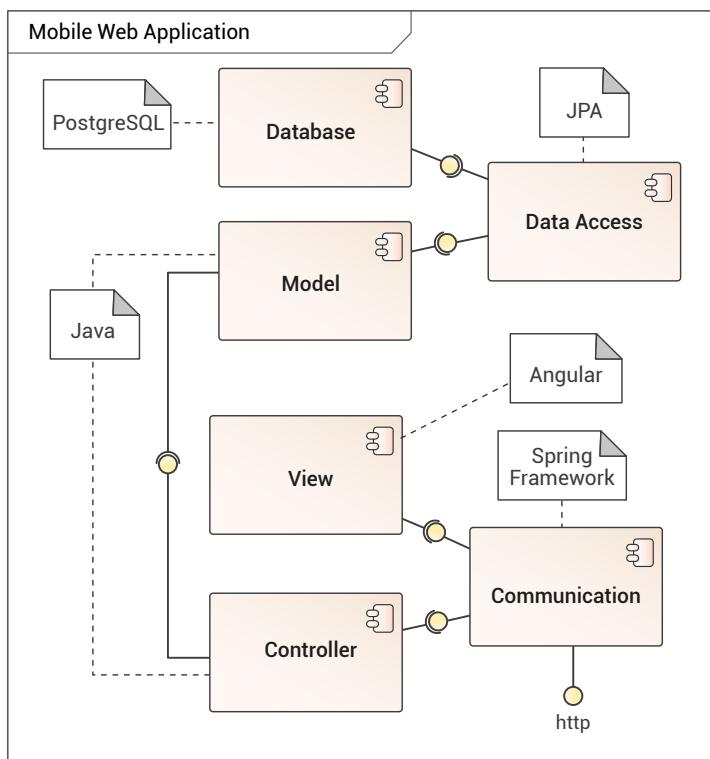
a lo largo de su ciclo de vida. A continuación, se define a cada uno de los conceptos que forman parte del estándar y que seguramente usted ya está familiarizado con lo que venimos estudiando en el presente curso, vista, punto de vista, vistas arquitectónicas.

1.9.1. Vista

Acorde a su definición, las vistas son consideradas como una representación de un sistema completo desde la perspectiva de un conjunto de intereses o necesidades relacionadas, según Hilliard (2000). Para describir una arquitectura software se puede utilizar una o varias vistas. En la figura 19 se visualiza los componentes que son utilizados para el desarrollo de una aplicación web, entre ellos destacan el modelo, la vista, el controlador, el acceso a los datos y algunas tecnologías que se pueden utilizar, más adelante estudiaremos que la vista de desarrollo incluye al diagrama de componentes.

Figura 19.

Vista de desarrollo que expone los componentes (diagrama de componentes) para representar el funcionamiento de MVC en una aplicación Web



Nota. Argüello M., & Guamán D., 2023

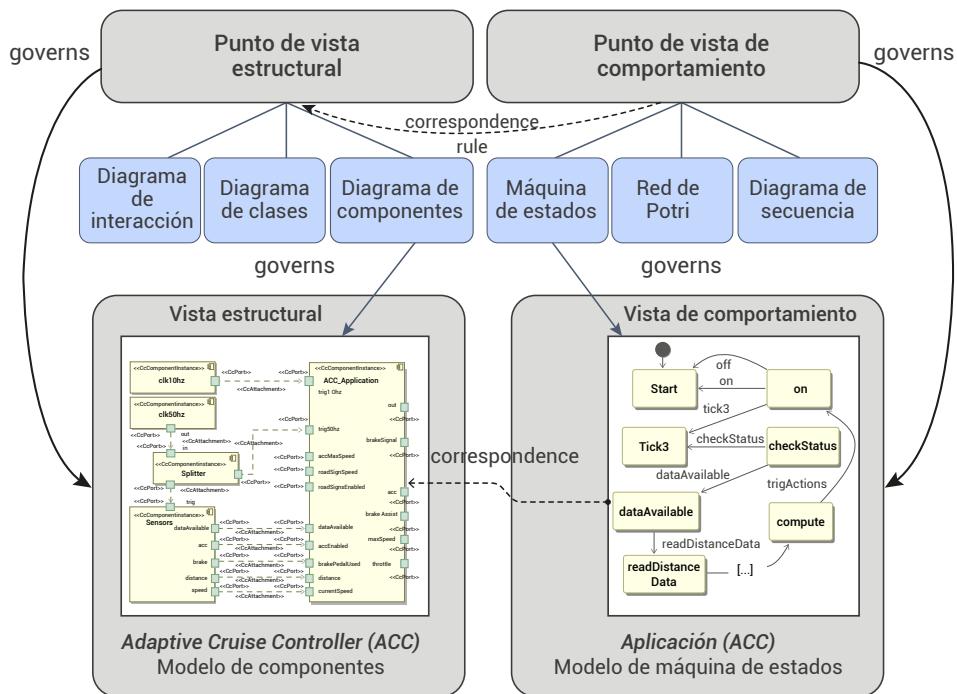
1.9.2. Punto de vista

Los puntos de vista definen la especificación de las convenciones para documentar y usar una vista que está conformada por un conjunto de diagramas. Los puntos de vista recordemos que son requeridos por las partes interesadas, por ejemplo, ingenieros de software, dueños de producto, arquitectos de software, desarrolladores, quienes a través del uso y comprensión de conjunto de diagramas permitirán entender acorde a su punto de vista cómo será la arquitectura del software. En la figura 20 se visualiza los tipos de elementos, las relaciones, así como la meta-information para describir la vista y que serán de utilidad al momento de utilizarlo por los puntos de vista. Los puntos de vista tratan de satisfacer a una audiencia distinta, cada una interesada en aspectos diferentes del sistema. Asociado a cada uno de los puntos de vista se define un lenguaje

especializado, que recoge el vocabulario y la forma de expresarse de la audiencia concreta a la que se dirige.

Figura 20.

Puntos de vista utilizados para explicar una vista que contiene diagramas asociados, en este caso de una aplicación ACC (Adaptive Cruise Controller)



Nota. Tomado de Software Architecture Modeling by Reuse, Composition and Customization, por Ivano, M., 2012, Universita Di L'Aquila, L'Aquila, Italy, Thesis, 1.

1.9.3. Vistas arquitectónicas

Las vistas arquitectónicas se consideran como la representación de un sistema desde la perspectiva de un conjunto de asuntos de interés. Las vistas que pueden ser de tipo estática, diseño, casos de uso, máquina de estados, actividades, interacción, despliegue, gestión de modelos, perfiles se utilizan para organizar y presentar los conceptos de uno o varios tipos de diagramas UML con el fin de proporcionar una notación visual como lo exponen Li, Lan y Avgeriou (2016). Se muestran las áreas sobre las cuales se puede usar una vista arquitectónica, un conjunto de diagramas y los principales elementos o conceptos UML que intervienen.

Tabla 1.
Vistas y diagramas UML asociados

Principal Área	Vista	Diagrama	Principales conceptos o elementos UML
Estructural		Diagrama de clases	association, class, dependency, generalization, interface, realization
	Diseño	Estructura interna	connector, interface, part, port, provided interface, role, required interface
		Diagrama de colaboración	connector, collaboration, collaboration use, role
		Diagrama de componentes	component, dependency, port, provided interface, realization, required interface, subsystem
	Casos de uso	Diagrama de casos de uso	actor, association, extend, include, use case, use case generalization
Dinámico	Máquina de estados	Diagrama de máquina de estados	completion transition, do activity, effect, event, region, state, transition, trigger
	Actividades	Diagrama de actividades	action, activity, control flow, control node, data flow, exception, expansion region, fork, join, object node, pin
	Interacción	Diagrama de secuencia	occurrence specification, execution specification, interaction, interaction fragment, interaction operand, lifeline, message, signal
		Diagrama de comunicación	collaboration, guard condition, message, role, sequence number
Físico	Despliegue	Diagrama de despliegue	artifact, dependency, manifestation, node
Gestión de modelos	Gestión de modelos	Diagrama de paquetes	import, model, package
	Perfiles	Diagrama de paquetes	constraint, profile, stereotype, tagged value

Nota. Fuente: Rumbaugh et al., (2004)

Para representar gráficamente las vistas se utiliza elementos, relaciones y diagramas, los cuales se construyen con UML como lenguaje que permite documentar la arquitectura y el cual propone como **Propiedades** definir información semántica sobre arquitecturas de software y sus elementos arquitectónicos.

Su información es adicional a las propiedades estructurales de la descripción de una arquitectura de software. Se puede asociar a cualquiera de los elementos arquitectónicos de una descripción de arquitectura de software (componentes, conectores, sistemas, interfaces, puertos, enlaces o archivos adjuntos). Como restricciones UML restringe el diseño de descripciones de arquitectura de software o estilos arquitectónicos a lo largo de toda su vida de ejecución.

Bien, una vez que hemos mencionado algunos de los conceptos y elementos que son parte de una arquitectura, es necesario realizarnos la siguiente pregunta ¿es lo mismo una arquitectura lógica que una arquitectura física?, le invito a que podamos revisar el siguiente tema, donde luego de su revisión podrá brindar una o varias respuestas a la pregunta.

1.10. Arquitectura lógica vs. Arquitectura física

La fase de análisis dentro del CVDS, es donde se presentan y analizan las posibles soluciones para identificar la que mejor se ajusta a las metas y objetivos que debe tener el software. Esta fase se considera vital para el diseño, codificación e implementación del software. La arquitectura de software abarca dimensiones como la arquitectura lógica y la arquitectura de despliegue o física.

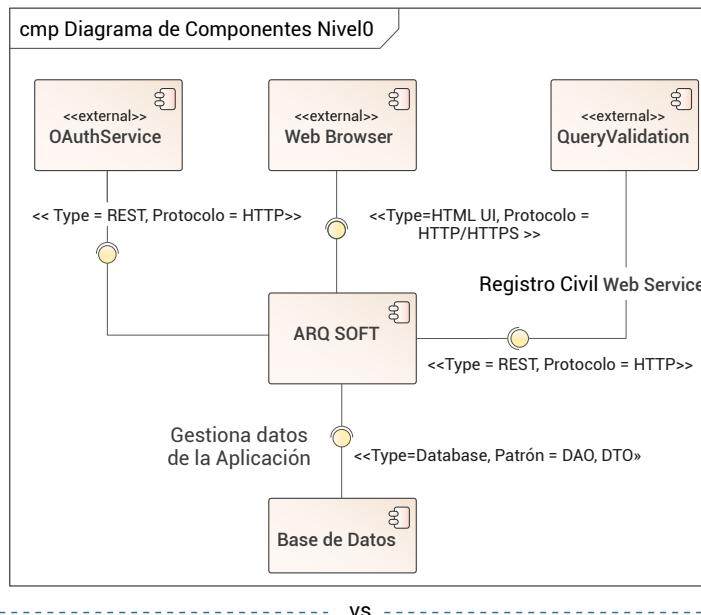
La **arquitectura lógica (logical architecture)**, describe el sistema en términos de su organización conceptual, de funcionamiento interno o interacción. Esta arquitectura se soporta en diagramas ya conocidos tales como casos de uso, clases, paquetes, componentes, subsistemas. La arquitectura lógica se refiere en sí a qué estilo o patrón arquitectónico utilizaremos para llevar a cabo la implementación del sistema en donde utilizaremos algún lenguaje o tecnología de programación (Java, .Net, PHP, etc.). En su forma de representación, tomaremos como ejemplo el patrón arquitectónico 3 “layers” o capas lógicas que lo explicaremos más adelante, donde visualmente se puede evidenciar que la distribución de los componentes como un **monolito o arquitectura monolítica** permitirá cumplir con algunos de los atributos de calidad como mantenibilidad, seguridad.

Por otro lado, la **arquitectura de despliegue o física (physical architecture)**, describe el sistema en términos de la asignación de procesos a las

unidades hardware de procesamiento (servidores) que funcionan despliegue local o en *cloud*, y la configuración de la red (componentes, nodos, dispositivos, protocolos o interfaces de comunicación interna y externa del sistema). En su forma de representación, similar a “*tiers*” o **capas físicas**. En la figura 21 se muestra una representación gráfica de la arquitectura lógica y física.

Figura 21.

Arquitectura lógica (Layer) vs Arquitectura Física (Tiers)

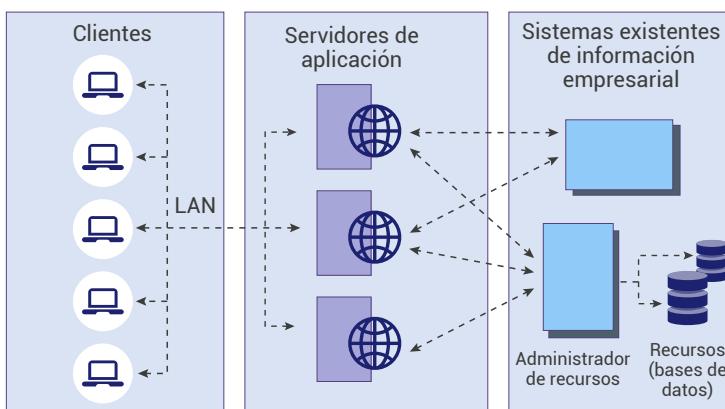


VS

Tier - Capa 1
Presentación

Tier - Capa 2
Negocio

Tier - Capa 3
Datos/Recursos



Nota. Tomado de *Three-tier architectures [Ilustración]*, por IBM, 2022, ibm.com. CC BY 2.0

Cuando se trabaja de forma colaborativa, dentro de los equipos de desarrollo de software, la arquitectura se considera un medio de comunicación que permite a todo comprender en alto nivel la estructura interna (*arquitectura lógica - layers*) que debe ser desplegada en un entorno (*arquitectura física - tiers*) para el correcto funcionamiento y cumplimiento de los atributos de calidad a nivel de todo el sistema.

Recuerde que la arquitectura es una **abstracción** que se centra en las **principales decisiones de diseño**, entre las que constan:

- **Estructura:** componentes y conexiones (relaciones).
- **Comportamiento:** responsabilidades de cada componente, algoritmos de alto nivel.
- **Interacción:** reglas que gobiernan y definen cómo se comunican los componentes.
- **Atributos de calidad:** estrategia para lograr la calidad a nivel estructural y de comportamiento.
- **Implementación:** lenguajes de programación, plataformas, bibliotecas, etc.

¿Se ha comprendido la diferencia entre lo lógico y lo físico de una arquitectura?, no se olvide que si aún tiene dudas puede contactar con su profesor tutor y complementar el estudio con bibliografía básica disponible de forma digital y en la web.

Conforme avanzamos nuestro estudio, una de las palabras que los ingenieros y arquitectos de software utilizamos para describir la arquitectura es diseño, y esto plantea la pregunta de si deberíamos usar las palabras **arquitectura** y **diseño** indistintamente, ¿son lo mismo?, ¿no es lo mismo?, ¿usted qué opina? A continuación, revisemos lo que mencionan algunos autores referentes a estos dos conceptos.

Con el estudio de estos temas finalizamos la semana 3. Es necesario que tenga claro los conceptos que forman parte del modelo de descripción arquitectónica tales como elementos arquitectónicos (componentes y conectores), stakeholders, concerns y sistema. Además, es importante entender la diferencia entre arquitectura lógica y arquitectura física, así como comprender la diferencia entre arquitectura y diseño. Recuerde apoyarse en bibliografía básica y complementaria disponible de forma digital o en la web, para complementar su estudio y no se olvide de desarrollar las actividades de aprendizaje recomendadas.

Lecturas en bibliografía básica, complementaria y recursos educativos abiertos

Revise la bibliografía complementaria para analizar el estándar ISO/IEC/IEEE 42010 y relacionarlos con conceptos que se usan en la descripción, diseño e implementación de una arquitectura de *software*.

- **Libro de Somerville** – (Capítulo 6 - Diseño arquitectónico – subtemas 6.2 Vistas arquitectónicas).
- **Libro de Presman** – (Capítulo 8 – Conceptos de diseño. Subtemas 8.1 Diseño en el contexto de la ingeniería de software. Capítulo 9 – Diseño de la arquitectura. Subtemas 9.1.3 Descripciones arquitectónicas).
- **Libro a rquitectura de s oftware, conceptos y ciclo de desarrollo** – (Capítulo 4 – Vistas – subtemas 4.4.1 Vistas lógicas, 4.4.2 Vistas de comportamiento, 4.4.3 Vistas físicas).
- **DD3_Arquitecturas de s oftware** - Componentes, conectores y relaciones (p áginas 7,8).



Actividades de aprendizaje recomendadas

1. Con base a la revisión de la bibliografía para su estudio y los que Ud. puede adicionar, por favor mencione al menos 5 similitudes y 5 diferencias entre arquitectura y diseño.
2. En sus propias palabras, a través de un pequeño ejemplo exponga un escenario de aplicación donde haya utilizado o piense que se utilicen elementos arquitectónicos (componentes, conectores, puertos).

Nota. conteste las actividades en un cuaderno de apuntes o en un documento Word.

3. Una vez que ha estudiado los conceptos relacionados a la u nidad, le invito a desarrollar la a utoevaluación con el fin de evaluar los conocimientos adquiridos hasta el momento.



Autoevaluación 1

1. ¿Cuál de las siguientes opciones se considera correcta respecto de un concepto de arquitectura de software?
 - a. La arquitectura de *software* de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema, que comprende elementos solamente las tecnologías y lenguajes de programación a utilizar.
 - b. La arquitectura de *software* de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema, que comprende elementos arquitectónicos con (vistas y puntos de vista) y las relaciones entre ellos y propiedades de ambos.
 - c. La arquitectura de *software* de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema, que comprende elementos de *software*, relaciones como agregación, composición y propiedades de ambos.
 - d. La arquitectura de *software* de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema, que comprende elementos de *software*, relaciones entre ellos y propiedades de ambos.
2. ¿A qué concepto se asocia el siguiente concepto? Es una representación de un conjunto coherente de elementos arquitectónicos, escritos y leídos por las partes interesadas del sistema.
 - a. Arquitectura de *software*.
 - b. Descripción arquitectónica.
 - c. Vista.
 - d. Punto de vista.

3. En arquitectura de software existen 3 categorías de estructuras que se puede utilizar para razonar sobre el sistema, estructuras de módulo, estructuras de componentes y conectores, estructuras de asignación. ¿A qué se refiere la estructura de módulo?
- Muestran cómo se estructurará el sistema como un conjunto de elementos que tienen comportamiento en tiempo de ejecución (componentes) e interacciones (conectores).
 - Muestran cómo se relacionará el sistema con las estructuras que no son de software en su entorno (por ejemplo, CPU, sistemas de archivos, redes, equipos de desarrollo, etc.).
 - Muestra la representación de una o más estructuras.
 - Muestran cómo se estructurará un sistema como un conjunto de códigos o unidades de datos que deben construirse o adquirirse.
4. En arquitectura de software existen 3 categorías de estructuras que se puede utilizar para razonar sobre el sistema, estructuras de módulo, estructuras de componentes y conectores, estructuras de asignación. ¿A qué se refiere la estructura de componentes y conectores?
- Muestran cómo se estructurará el sistema como un conjunto de elementos que tienen comportamiento en tiempo de ejecución (componentes) e interacciones (conectores).
 - Muestran cómo se relacionará el sistema con las estructuras que no son de software en su entorno (por ejemplo, CPU, sistemas de archivos, redes, equipos de desarrollo, etc.).
 - Muestra la representación de una o más estructuras.
 - Muestran cómo se estructurará un sistema como un conjunto de códigos o unidades de datos que deben construirse o adquirirse.

5. En arquitectura de software existen 3 categorías de estructuras que se puede utilizar para razonar sobre el sistema, estructuras de módulo, estructuras de componentes y conectores, estructuras de asignación. ¿A qué se refiere la estructura de asignación?
- Muestran cómo se estructurará el sistema como un conjunto de elementos que tienen comportamiento en tiempo de ejecución (componentes) e interacciones (conectores).
 - Muestran cómo se estructurará un sistema como un conjunto de códigos o unidades de datos que deben construirse o adquirirse.
 - Muestra la representación de una o más estructuras.
 - Muestran cómo se relacionará el sistema con las estructuras que no son de software en su entorno (por ejemplo, CPU, sistemas de archivos, redes, equipos de desarrollo, etc.).
6. ¿Cuál de las siguientes opciones son correctas respecto a la importancia de la arquitectura de software?
- Una arquitectura inhibirá o habilitará los atributos de calidad de conducción de un sistema.
 - Las decisiones tomadas en una arquitectura le permiten razonar y gestionar el cambio a medida que evoluciona el sistema.
 - Una arquitectura documentada mejora la comunicación entre las partes interesadas.
 - Una arquitectura define un conjunto de restricciones sobre la implementación posterior.

7. ¿Qué es el SWEBOK?
- a. Un estándar que guía al conocimiento presente en el área de la Ingeniería de software.
 - b. Un documento que guía al conocimiento presente en el área de la Ingeniería de software.
 - c. Una norma que guía al conocimiento presente en el área de la Ingeniería de software.
 - d. Un proceso que guía al conocimiento presente en el área de la Ingeniería de software.
8. ¿Cuál de las siguientes opciones es correcta? La arquitectura de un sistema define cuatro aspectos diferentes.
- a. Estructura estática, estructura dinámica, comportamiento visible externamente y propiedades de calidad.
 - b. Elementos arquitectónicos, vistas, puntos de vista y componentes.
 - c. Partes interesadas, arquitectos, desarrolladores, gerentes de proyecto.
 - d. Estructura estática, vista lógica, vista física, diseño y arquitectura.
9. A qué concepto se asocia el siguiente texto: “es un conjunto de productos que documenta una arquitectura de una manera que sus partes interesadas pueden entender y demuestra que la arquitectura ha respondido a sus preocupaciones”.
- a. Elementos arquitectónicos.
 - b. Partes interesadas.
 - c. Arquitectura.
 - d. Descripción arquitectónica.

10. A qué concepto se asocia el siguiente texto: "es la persona (o grupo) responsable de diseñar, documentar y liderar la construcción de una arquitectura que satisfaga las necesidades de todos sus stakeholders".
- a. El analista.
 - b. El diseñador.
 - c. El gerente de producto.
 - d. El arquitecto.

[Ir a solucionario](#)



Unidad 2. Consideraciones y contextos para la definición y diseño de una arquitectura de software

Ahora que entendemos los fundamentos de la arquitectura de software, es importante conocer algunas prácticas y consideraciones que se pueden utilizar al momento de definir y diseñar la arquitectura de un sistema de software. Recuerde que existe un conjunto de características comunes y principios de diseño que impulsan, influyen y dan forma a una arquitectura de software dentro de un contexto.

2.1. Consideraciones para la definición de una arquitectura de software

Antes de abordar esta unidad, es importante que revise el anexo 1 donde se propone un estudio de caso de un escenario para que usted como arquitecto entienda el dominio del problema y pueda utilizar algunas técnicas y consideraciones que le permitirán proponer una solución documentada apoyada del documento de arquitectura de software (anexo 2) y del documento de especificación de requisitos (anexo 3). Lo que se expone en los anexos 2 y 3 es una alternativa de solución, usted podría ampliarla y complementarla utilizando los conceptos que estudiaremos esta semana.

Es normal que, en etapas tempranas del proceso de software, no conozca completamente el tamaño y la extensión del sistema, cuál es el nivel de complejidad, cuáles son los riesgos más importantes o dónde existirán conflictos entre las partes interesadas. Sin embargo, lo que se recomienda para que una arquitectura tenga éxito, es tener en cuenta algunos principios como los propuestos en Nick Rozanski (2008), y consideraciones como los que se exponen a continuación:

- La arquitectura debe estar impulsada por las **concerns (interés, inquietud o responsabilidad)** de las partes interesadas, las cuales son el núcleo de un proceso de definición de arquitectura, pero de ninguna manera son las únicas aportaciones al proceso. En el anexo 1 se

propone un conjunto de *concerns* dados por los usuarios ante una problemática dentro de un contexto.

- La arquitectura debe estar soportada por **documentación** que evidencie el alcance, el contexto, las decisiones arquitectónicas, los principios y la propia solución para comunicar a las partes interesadas. El documento que se utiliza para ello es el documento de arquitectura de *software* que se propone en el anexo 2.
- La arquitectura debe garantizar que de forma iterativa se cumplan las decisiones y los principios arquitectónicos durante todo el ciclo de desarrollo hasta la implementación final.
- La arquitectura debe ser definida siguiendo un proceso compuesto de una serie de pasos o actividades, que permitan la definición clara de los objetivos, y de las entradas y salidas requeridas en cada paso, así como debe permitir su evaluación. Normalmente, en un proceso de definición de arquitectura, las salidas de una actividad se convierten en las entradas de las posteriores.
- La arquitectura debe ser tecnológicamente neutral. Es decir, no debe exigir que la solución se construya utilizando una tecnología, repositorio, patrón arquitectónico o estilo de desarrollo específicos, ni debe dictar ningún estilo para el modelado, diagramación o documentación en particular.
- La arquitectura debe alinearse con una metodología de desarrollo, buenas prácticas de ingeniería de *software* y usar estándares de gestión de la calidad como la norma ISO 9126, Camacho, Cardeso y Nuñez (2004), para que pueda integrarse y adaptarse a circunstancias particulares dentro del contexto. En el anexo 3 en los ítems 4.5 al 4.10 se exponen los requisitos asociados con los atributos de calidad.

El objetivo de definir una arquitectura es **desarrollar una arquitectura sólida y gestionar la producción y el mantenimiento de todos los elementos que la conforman**. Si recordamos, la arquitectura de *software* es el conjunto de decisiones de diseño que se implementan sobre el sistema. Estas decisiones deben tomar en cuenta actividades, conceptos, métodos y estrategias que permitan evaluar el cumplimiento de los atributos de calidad según el contexto de aplicación. Le invito a que revisemos algunas

consideraciones que se pueden usar para el diseño de una arquitectura de software.

2.2. Consideraciones para el diseño de una arquitectura de software

Cuando un arquitecto de software define una arquitectura, asume que su diseño evolucionará con el tiempo y que no puede saber todo lo que necesita desde el principio para diseñar completamente el sistema.

Por lo general, el diseño deberá evolucionar durante las etapas de implementación de la aplicación a medida que los requisitos funcionales dados por los clientes o la organización son incorporados y validados en cada incremento o versión del sistema final.

Algunos autores sugieren que para que un proyecto de software sea exitoso, los arquitectos de software deben seguir un conjunto de reglas, pautas y principios básicos que a nivel de diseño permitan dar respuesta a preguntas tales como ¿cuáles son las partes fundamentales de la arquitectura que representan un mayor riesgo si algo sale mal?, ¿cuáles son las partes de la arquitectura que tienen más probabilidades de cambio o evolución, o cuyo retraso genere impacto en el diseño?, ¿cuáles son sus suposiciones clave y cómo las probará/validará?, ¿qué condiciones pueden requerir una refactorización al diseño o implementación?

Para dar respuesta a estas y otras preguntas, estos son algunos de los principios clave para diseñar una arquitectura de software:

- **Construya para cambiar, en lugar de construir para durar:** considere que el sistema puede necesitar cambiar o evolucionar con el tiempo para abordar nuevos requisitos y desafíos tecnológicos o de empresa, especialmente cuando usamos un enfoque ágil, por lo tanto, hay que tener en cuenta la flexibilidad y escalabilidad que permita soportar aquello.
- **Modele para analizar y reducir riesgos:** utilice herramientas CASE de diseño, lenguajes de modelado como UML y modelos como 4+1 o C4 que permiten visualizar decisiones de arquitectura y diseño, y cuando se requiera analizar su impacto. Sin embargo, no formalice el modelo en la medida en que suprima la capacidad de iterar y adaptar el diseño fácilmente. Esto lo podemos evidenciar en el anexo 2 donde

se muestra como documentar el dominio de la solución usando UML y utilizando el modelo 4+1.

- **Utilice modelos y visualizaciones como herramienta de comunicación y colaboración:** la comunicación eficiente del diseño, las decisiones y los cambios continuos en el diseño son fundamentales para una buena arquitectura. Utilice modelos, vistas y otras visualizaciones de la arquitectura para comunicar y compartir su diseño de manera eficiente con todas las partes interesadas y para permitir una comunicación rápida cuando existan cambios en el diseño.
- **Identifique decisiones clave de ingeniería:** invierta tiempo en tomar decisiones clave de forma correcta, especialmente la primera vez y en etapas tempranas, para que el diseño sea más flexible y tenga menos probabilidades de tener inconvenientes con los cambios.
- **Comience con una arquitectura de referencia o base:** esto le ayudará a tener una visión general y correcta, a partir de lo cual podrá desarrollar arquitecturas candidatas a medida que prueba, mejora y refina las diferentes versiones de su arquitectura de manera iterativa e incremental.
- **No intente hacerlo bien la primera vez:** diseñe todo lo que pueda para comenzar a validar la trazabilidad entre el diseño versus los requisitos, objetivos y suposiciones dados por las partes interesadas.
- **Separación de intereses:** divida el sistema en distintas funciones con la menor superposición de funciones posible. El factor importante es la minimización de los puntos de interacción para lograr una **alta cohesión y un bajo acoplamiento**. Sin embargo, separar la funcionalidad en los límites incorrectos puede llevarnos a tener un alto acoplamiento y complejidad entre componentes y características.
- **Principio de responsabilidad única:** cada componente o módulo debe ser responsable solo de una característica o funcionalidad específica, en términos de diseño de aplicaciones, la funcionalidad específica debe implementarse en un solo componente y no debe duplicarse en ningún otro componente.

- **Minimice el diseño inicial:** diseñe solo lo necesario, ya que, en algunos casos, es posible que necesite un diseño y pruebas integrales por adelantado si el costo de desarrollo o una falla en el diseño es muy alto. En otros casos, especialmente para el desarrollo ágil, puede evitar realizar un gran diseño por adelantado, especialmente si los requisitos de su aplicación no están claros, o si existe la posibilidad de que el diseño evolucione con el tiempo. Evite hacer un gran esfuerzo de diseño prematuramente, a este principio a veces se conoce como YAGNI ("You Aren't Gonna Need It", "No lo vas a necesitar").
- **Agregue iterativamente detalles al diseño:** utilice varias iteraciones para asegurarse de seleccionar primero las decisiones de diseño a nivel general más importantes y luego concentrarse en los detalles. Un error común es profundizar primero en los detalles demasiado rápido y equivocarse en las decisiones importantes al hacer suposiciones incorrectas o al no evaluar su arquitectura de manera efectiva.



Recuerde que estas son algunas de las recomendaciones que espero le sean de ayuda cuando defina y diseñe una arquitectura de software. Ahora lo que Ud. se ha de preguntar es ¿qué características o principios hacen buena a una arquitectura de software? Le invito a que revisemos el siguiente tema (Principios de diseño) donde se da respuesta a la pregunta.

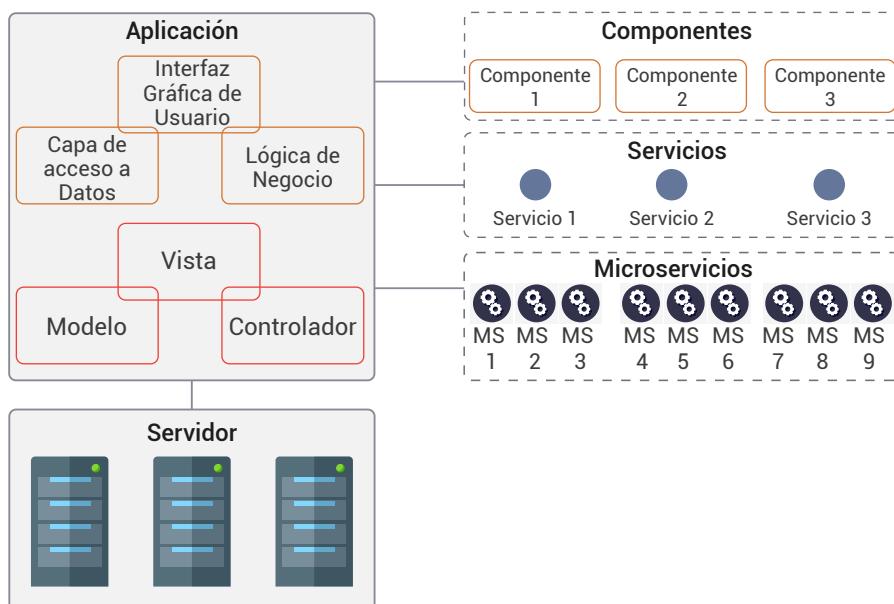
2.3. Principios de diseño

Recuerde que la arquitectura establece una base firme para cualquier proyecto de software. Una arquitectura sólida define los estándares técnicos, el diseño, la entrega y el soporte del producto de software. Al diseñar la arquitectura de software, se debe tener en cuenta los objetivos de desarrollo, infraestructura tecnológica y asegurarse de utilizar algunos principios de diseño tales como *descomposición modular, generar y probar, acoplamiento y cohesión*, que le invito a revisarlos a continuación.

2.3.1. Descomposición modular / Modularización

Es claro que la arquitectura determina los atributos de calidad, los cuales son características aplicables a todo el sistema. Por tanto, el diseño comienza pensando en el sistema como un todo. Para poder llegar a diseñar e implementar el sistema debemos descomponerlo en partes, donde cada una de ellas puede heredar todos o algunos de los requisitos del atributo de calidad del conjunto. En este momento recuerde cuando hablamos de monolito y como podíamos pasar por servicios hasta llegar a microservicios. Similar a un rompecabezas, es decir, cuando aparezcan nuevos componentes resultantes del análisis para ser usados en el diseño final, entonces la descomposición inicial debe acomodar esos nuevos componentes en su diseño o arquitectura candidata.

Figura 22.
Descomposición e inclusión de componentes



Nota. Argüello M., & Guamán D., 2023

La modularización es el proceso de dividir un sistema de software en múltiples módulos independientes donde cada uno de ellos funciona de forma independiente. Entre las ventajas de la modularización en la ingeniería y arquitectura de software constan la fácil comprensión y entendimiento del sistema, facilidad para el mantenimiento del sistema,

reutilización de los módulos según sus requisitos y evitar replicar componentes o código varias veces.

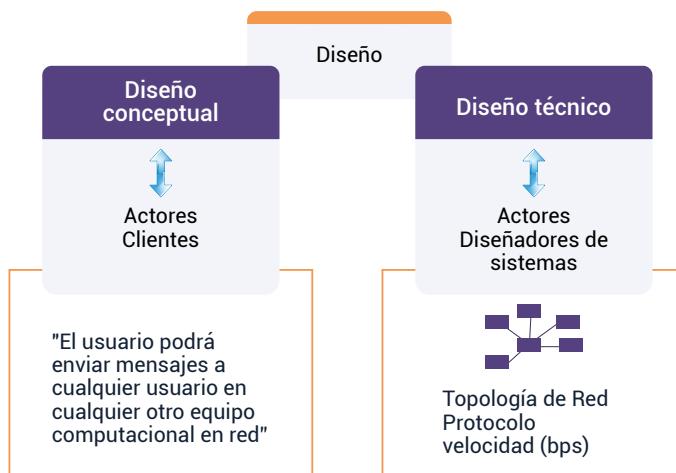
2.3.2. Generar y probar



Recuerde que el diseño o arquitectura candidata debe ser visto como una hipótesis. Una hipótesis, según la RAE, es una suposición de algo posible o imposible para sacar de ello una consecuencia. Esto quiere decir que hay que preguntarnos si el diseño actual satisface los requisitos, si no es así, debemos generar una nueva hipótesis (un nuevo diseño o refinar el actual).

El propósito de la fase de diseño en el ciclo de vida del desarrollo de software es producir una solución a un problema dado en el documento de Especificación de Requisitos de Software (ERS o SRS). El resultado de la fase de diseño es el Documento de Diseño de Software (DSS o SDD) con el cual podemos validar o testear el diseño y su implementación. Básicamente, el diseño es un proceso iterativo de dos partes, la primera es el *diseño conceptual* que le dice al cliente lo que hará el sistema y el segundo es el *diseño técnico* que permite a los desarrolladores de sistemas comprender el *hardware* y el *software* reales necesarios para resolver el problema del cliente.

Figura 23.
Diseño conceptual y técnico



Nota. Argüello M., & Guamán D., 2023

Algunas características del diseño conceptual y técnico del sistema se exponen en la tabla 2.

Tabla 2.
Diseño conceptual y diseño técnico

Diseño conceptual	Diseño técnico
<ul style="list-style-type: none"> ▪ Escrito en un lenguaje sencillo y natural, es decir, un lenguaje comprensible para el cliente. ▪ Explicación detallada sobre las características del sistema. ▪ Descripción de las funcionalidades del sistema. ▪ Es independiente de la implementación. ▪ Vinculado con el documento de especificación de requisitos. 	<ul style="list-style-type: none"> ▪ Diseño y componente de hardware. ▪ Funcionalidad y jerarquía de los componentes de software. ▪ Arquitectura de software. ▪ Red de arquitectura. ▪ Estructura de datos y flujo de datos. ▪ Componente de Entrada / Salida del sistema.

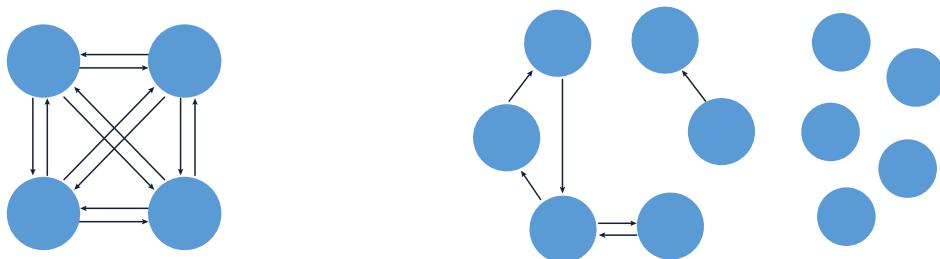
Nota. Argüello M., & Guamán D., 2023

2.3.3. Acoplamiento

El acoplamiento es la medida del grado de interdependencia entre los módulos. Un buen software debe considerar el bajo acoplamiento.

En la figura 24 podemos observar que los círculos representan a los componentes o módulos del diseño que serán implementados, en él se puede representar la comunicación, las dependencias que tiene y sobre todo la coordinación y comunicación entre componentes. Como regla general, los módulos son altamente acoplados si hacen uso de variables compartidas o si intercambian información de control. Los tipos de acoplamiento permiten visualizar si los mismos tienen un alto o bajo acoplamiento.

Figura 24.
Representación gráfica de acoplamiento



Contenido

Común

Externo

Control

Stamp

Datos

Alto acoplamiento

Muchas dependencias
Mas coordinación
Más flujo de información

Bajo acoplamiento

Algunas dependencias
Poca coordinación
Menos flujo de información

Desacoplado

Sin dependencias

Nota. Argüello M., & Guamán D., 2023

A nivel de acoplamiento, existen algunos tipos entre los que constan:

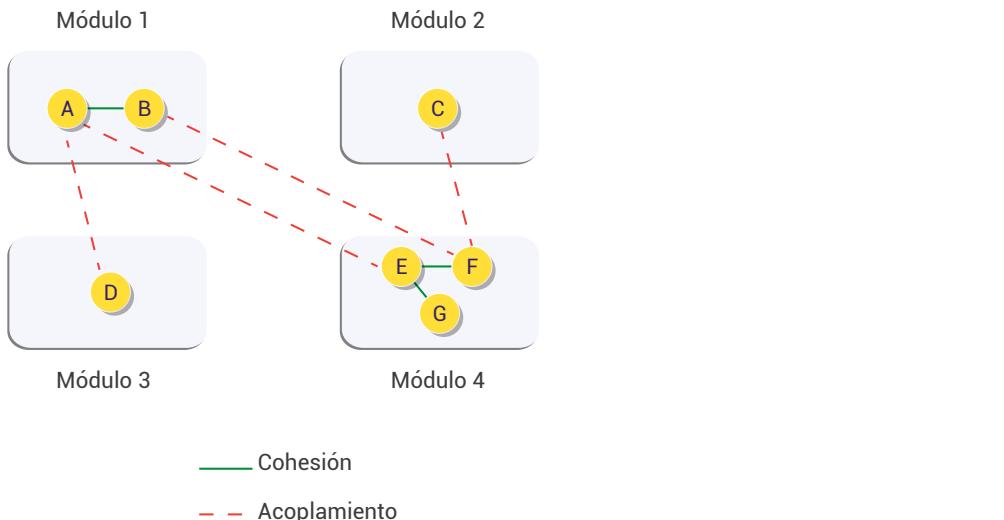
- **Acoplamiento de datos:** si la dependencia entre los módulos se basa en el hecho de que se comunican pasando solo datos, se dice que los módulos están acoplados a datos. En el acoplamiento de datos, los componentes son independientes entre sí (*bajo acoplamiento*). Por ejemplo, los sistemas de facturación al emitir el documento (factura) al cliente haciendo uso de servicios de facturación electrónica.

- **Acoplamiento de Stamp:** en el acoplamiento de sellos (*Stamp*), la estructura de datos completa se pasa de un módulo a otro módulo. Esto puede ser necesario debido a factores de eficiencia, sin embargo, esta elección la hace el arquitecto con conocimientos y habilidades técnicas.
- **Acoplamiento de control:** si los módulos o componentes se comunican pasando información de control, entonces se dice que tienen acoplamiento de control. Esto no puede ser tan beneficioso si los parámetros indican un comportamiento completamente diferente, sin embargo, resulta bueno si los parámetros permiten factorizar y reutilizar la funcionalidad. Un ejemplo de esto es la función de ordenación que toma una función de comparación como argumento.
- **Acoplamiento externo:** en el acoplamiento externo, los módulos dependen de otros módulos, generalmente externos al *software* que se está desarrollando o a un tipo particular de *hardware*. Por ejemplo, protocolos, archivo externo, formato de dispositivo, etc.
- **Acoplamiento común:** los módulos tienen datos compartidos, como estructuras de datos globales (*alto acoplamiento*). Los cambios en los datos globales significan rastrear todos los módulos que acceden a esos datos para evaluar el efecto del cambio. Una de las desventajas es la dificultad para reutilizar los módulos, la capacidad reducida para controlar el acceso a los datos y la capacidad de mantenimiento reducida.
- **Acoplamiento de contenido:** en un acoplamiento de contenido, un módulo puede modificar los datos de otro módulo o el flujo de control se pasa de un módulo a otro módulo (*alto acoplamiento*).

2.3.4. Cohesión

La cohesión es una medida del grado en que los elementos (internos) de un módulo están relacionados funcionalmente (ver figura 25). La cohesión es el grado en el que todos los elementos dirigidos a realizar una sola tarea están contenidos en el módulo o componente.

Figura 25.
Cohesión y acoplamiento



Nota. Argüello M., & Guamán D., 2023

Básicamente, la cohesión hace referencia a las relaciones internas que mantiene unido o compacto el módulo. Por tanto, se considera que un buen diseño de software tendrá una alta cohesión. Entre los tipos de cohesión constan los siguientes:

- **Cohesión funcional:** cada elemento esencial para un solo cálculo está contenido en el componente. Una cohesión funcional realiza la tarea y las funciones, esto se considera una situación ideal.
- **Cohesión secuencial:** un elemento genera algunos datos que se convierten en la entrada de otro elemento, es decir, el flujo de datos entre las partes. Ocurre de forma natural en los lenguajes de programación funcionales.
- **Cohesión comunicacional:** dos elementos operan sobre los mismos datos de entrada o contribuyen a los mismos datos de salida. Por ejemplo, al actualizar el registro en la base de datos y enviarlos a la impresora.
- **Cohesión procesal:** los elementos de cohesión procesal garantizan el orden de ejecución, donde las acciones están débilmente o poco conectadas y es poco probable que sean reutilizables. Por ejemplo, calcular el costo de cada asignatura que se matricula un estudiante,

imprimir el expediente del estudiante, calcular el costo total de la matrícula, imprimir la factura con los costos totales.

- **Cohesión temporal:** Los elementos están relacionados durante un tiempo predefinido. En un módulo conectado con cohesión temporal, todas las tareas deben ejecutarse en el mismo lapso de tiempo. Esta cohesión contiene el código para inicializar todas las partes del sistema.
- **Cohesión lógica:** Los elementos están relacionados lógicamente y no funcionalmente.
- **Cohesión coincidente:** los elementos no están relacionados y no tienen otra relación conceptual que la ubicación en el código fuente.



En resumen, para diseñar e implementar una aplicación debemos tener en cuenta la descomposición a nivel de componentes o funcionalidades y **sobre todo la alta cohesión y bajo acoplamiento**. Teniendo claro el significado de descomposición modular, cohesión y acoplamiento, es momento de conocer algunas consideraciones de diseño, tecnológicas y de despliegue para diseñar e implementar aplicaciones web, móviles y servicios.

2.4. ¿Qué hace que una arquitectura sea buena?

Es muy difícil definir y determinar cuáles son las características que hacen buena a una arquitectura de software, ya que no existe un estándar sobre lo bueno o lo malo y, por lo tanto, brindar una respuesta exacta a esta pregunta es complejo. Lo que sí está claro es que cada sistema tiene una arquitectura de software que con base en un proceso, consideraciones y principios fue analizada, diseñada, documentada, implementada y validada para operar dentro de un contexto específico.

La forma en la que se definen las directrices de la arquitectura suele depender de las necesidades de los clientes o el negocio, ya que pueden

ser muy rígidas o ajustables a un proyecto en específico. A veces pensamos que, solo mirando el producto final, se puede saber si la arquitectura de software es buena o mala. Esto definitivamente no es verdad, ya que la construcción del producto software es derivada de un proceso. Sin embargo, lo que si es cierto es que mirando el producto se puede encontrar algunas características de que algo es bueno, y más aún si lo experimentamos/utilizamos y tomamos en cuenta algunos indicadores que podrían evidenciar que el software tiene una buena arquitectura. A continuación, se mencionan algunas consideraciones que ayudan a determinar que una arquitectura es buena cuando:

- Todos los miembros del equipo y los *Stakeholders* entienden y comprenden los conceptos de dominio de la arquitectura (dominio del problema y dominio de solución).
- El software es fácil de usar y funciona como lo requiere el usuario final, lo que quiere decir que permite adaptarse a las necesidades de los clientes.
- Al encontrar algún problema en el software se puede llevar a cabo tareas de refactorización y mantenimiento sin afectar la arquitectura y su comportamiento.
- El software es flexible y permite su extensión, lo que indica que tiene una arquitectura que permite modificar o agregar nuevas funcionalidades a nivel de diseño e implementación y será usable y escalable a largo plazo.
- Se han tomado en cuenta a los atributos de calidad tales como usabilidad, rendimiento, escalabilidad, disponibilidad, interoperabilidad, entre otros.
- A nivel de diseño e implementación del software, los indicadores de calidad y deuda técnica (**architectural technical debt**) se encuentran dentro de los rangos definidos.
- Sobre el software se pueden llevar a cabo pruebas a nivel de caja blanca o caja negra sobre los componentes, funcionalidades o todo el sistema de manera integral.

Ahora que comprende la diferencia entre arquitectura y diseño, le invito a que revise algunos elementos clave que se proponen para, a más de incluir características que hacen buena a una arquitectura, también permitan que su diseño sea correcto. Algunas de las características claves indican que, un diseño se considera correcto cuando:

- Establece sistemas robustos, pero libres de *frameworks*. Cuando hablamos de *frameworks* nos referimos a los que permiten desarrollo, por ejemplo Laravel, Symfony, Spring Framework, Angular u otros. ¿Y por qué evitarlos en la medida de lo posible? Porque implementan o hacen uso de librerías o componentes que a veces no se utilizan y que podrían afectar a la estructura y calidad del *software*.
- El sistema hace uso de un repositorio de datos y este se alinea a las necesidades del negocio, no viceversa.
- Elige un conjunto de herramientas interoperables para optimizar procesos de desarrollo. Con los nuevos enfoques ágiles y metodologías como **SCRUM** y **DevOps**, existe la posibilidad de automatizar las pruebas, integración, despliegue en ambientes de desarrollo, preproducción y producción, por lo tanto, hay que utilizarlos y configurarlos de manera estratégica desde un inicio.

Seguramente con la revisión de bibliografía disponible de forma física, digital o en la web complementará su estudio. Al finalizar esta semana, espero que tengamos en cuenta consideraciones y principios que le permitirán analizar, definir y proponer una arquitectura de *software*. Es momento de reforzar los conocimientos previo al estudio de la siguiente unidad, para ello le invito a realizar las actividades de aprendizaje recomendadas. Luego de realizar las mismas lo espero la próxima semana para conocer el proceso para la definición y el diseño de una arquitectura de *software*.

Lecturas en bibliografía básica, complementaria y recursos educativos abiertos

- Conozca algunas de las consideraciones que se deben tomar en cuenta para el diseño y arquitectura de una solución *software*.
- **Libro de Somerville** – (Capítulo 6 - Diseño arquitectónico – subtemas 6.1 Decisiones en el diseño arquitectónico).

- **Libro de Presman** – (Capítulo 8 – Conceptos de diseño. Subtemas, 8.3.1 Abstracción, 8.3.4 División de problemas, 8.3.5 Modularidad, 8.3.7 Independencia funcional. Capítulo 9 – Diseño de la arquitectura. Subtemas 9.1.4 Decisiones arquitectónicas).
- Libro, a rquitectura de s oftware, conceptos y ciclo de desarrollo – (Capítulo 3 – Diseño, toma de decisiones para crear estructuras – subtemas 3.1 Diseño y niveles de diseño, 3.1.1 Diseño y arquitectura, 3.1.2 Niveles de diseño, 3.3 Principios de d iseño, 3.3.1 Modularidad, 3.3.2 Cohesión y acoplamiento, 3.3.3 Mantener simples las cosas).
- Revise el [anexo 1](#) donde se propone un ejemplo de un estudio de caso o escenario referente a un dominio del problema.
- Revise el [anexo 2](#) donde se utiliza el formato de documentación arquitectónica para de forma gráfica y textual describir una posible solución al problema (ver a nexo 2).
- Revise el [anexo 3](#) donde luego de entender y analizar el dominio del problema, se utiliza el documento de especificación de requisitos para transformar las necesidades dadas por las partes interesadas en requisitos funcionales y no funcionales.



Actividades de aprendizaje recomendadas

1. Con base a la revisión de la bibliografía básica, complementaria y recursos educativos abiertos sugeridos para su estudio y los que Ud. puede adicionar, por favor seleccione y describa 5 principios claves de la arquitectura que usted haya utilizado o utilizaría para el diseño y construcción de sus aplicaciones de software de tipo bancario.

Nota. conteste las actividades en un cuaderno de apuntes o en un documento Word.

2. Una vez que ha estudiado los conceptos relacionados a la u nidad, le invito a desarrollar la a utoevaluación con el fin de evaluar los conocimientos adquiridos hasta el momento.



Autoevaluación 2

1. La arquitectura puede ser vista desde diferentes contextos, entre ellos técnico, ciclo de vida del proyecto, negocio y profesional. Marque las opciones que se consideren correcta para cada contexto.
 - a. El contexto técnico incluye el logro de los requisitos de atributos de calidad.
 - b. El contexto dentro del ciclo de vida del proyecto, independientemente de la metodología de desarrollo de software que utilice, debe presentar un caso de negocio para el sistema y comprender los requisitos arquitectónicamente significativos.
 - c. En el contexto de negocio, el sistema creado a partir de la arquitectura debe satisfacer los objetivos de negocio dados por las partes interesadas, cada una de las cuales tiene diferentes expectativas para el sistema.
 - d. Dentro del contexto profesional, el arquitecto debe tener ciertas habilidades y conocimientos influenciados no solo técnico y la lectura, sino también por sus experiencias en proyectos similares.
 - e. Todas las anteriores.

2. ¿Cuál de las siguientes opciones se considera correcta?
- a. La arquitectura debe ser el producto de un solo arquitecto o un pequeño grupo de arquitectos con un líder técnico identificado.
 - b. La arquitectura debe ser el producto de un solo desarrollador o un pequeño grupo de desarrolladores con un líder técnico identificado.
 - c. La arquitectura debe ser el producto de analistas y diseñadores con un líder técnico identificado.
 - d. La arquitectura debe ser el producto de decisiones de diseño dadas por las partes interesadas con un líder técnico identificado.
3. ¿Cuáles de las siguientes opciones se considera adecuada, para considerar a una arquitectura buena?
- a. La arquitectura debe ser implementada con *Spring Framework*.
 - b. La arquitectura debe ser evaluada teniendo en cuenta solamente el rendimiento como atributo de calidad.
 - c. La arquitectura debe permitir la integración incremental de los módulos.
 - d. La arquitectura debe ser documentada usando vistas.
4. ¿Cuáles de las siguientes opciones se considera adecuada, para considerar a una arquitectura buena?
- a. La arquitectura debe depender de una versión particular de librerías y lenguajes de programación.
 - b. La arquitectura debe incluir módulos que cumplan con el principio de diseño, baja cohesión y alto acoplamiento.
 - c. La arquitectura debe incluir módulos bien definidos cuyas responsabilidades funcionales se asignen según los principios de ocultación de información y separación de preocupaciones.
 - d. La arquitectura debe incluir módulos que solamente producen datos.

5. ¿Cuáles de las siguientes opciones se considera adecuada, para considerar a una arquitectura buena?
 - a. Los módulos que producen datos deben estar dentro del mismo módulo que consumen datos.
 - b. Los módulos que producen datos deben estar separados de los módulos que consumen datos.
 - c. Los módulos que producen datos deben usar lenguajes y tecnologías de Back- End como HTML, CSS.
 - d. Los módulos que producen datos deben usar lenguajes y tecnologías de Front- End como NodeJS, Python.
6. ¿Cuáles de las siguientes opciones se considera adecuada, para considerar a una arquitectura buena?
 - a. La arquitectura debe contener un conjunto que abarque todas las áreas de conflicto de recursos, cuya resolución se especifica y mantiene claramente.
 - b. La arquitectura debe contener un conjunto específico de librerías y controles ActiveX.
 - c. La arquitectura debe contener un conjunto específico (y pequeño) de áreas de conflicto de recursos, cuya resolución se especifica y mantiene claramente.
 - d. La arquitectura debe contener un conjunto específico (y pequeño) de elementos arquitectónicos como vistas y puntos de vista.

7. ¿Cuáles de las siguientes opciones respecto de la arquitectura se consideran correctas?
- Las arquitecturas son buenas o malas dependiendo de lo que se visualiza en un prototipo.
 - Una arquitectura para un sistema académico es utilizada para un sistema bancario.
 - No existe una arquitectura intrínsecamente buena o mala. Las arquitecturas son más o menos adecuadas para algún propósito.
 - Una arquitectura es buena cuando usa como patrón Modelo – Vista – Presentador.
8. ¿Cuál de las siguientes opciones se considera correcta?
- Una arquitectura canaliza la creatividad de los desarrolladores, reduciendo el diseño y la complejidad del sistema.
 - Una arquitectura puede proporcionar la base para la creación de prototipos evolutivos.
 - El análisis de una arquitectura permite la predicción temprana de las cualidades de un sistema.
 - Una arquitectura es el artefacto clave que permite al arquitecto y al administrador del proyecto razonar sobre el costo y el cronograma.
 - Todas las anteriores.

9. ¿Cuáles de las siguientes opciones se consideran las principales decisiones que debe tomar y que ayudan a garantizar que considere todos los factores importantes al comenzar y luego desarrollar iterativamente el diseño de su arquitectura?
- a. Determinar el tipo de aplicación.
 - b. Determinar la estrategia de implementación y despliegue.
 - c. Determinar las tecnologías apropiadas.
 - d. Determinar los atributos de calidad.
 - e. Todas las anteriores.
10. Una buena forma de representar visualmente aspectos del alcance del sistema y comenzar el trabajo con las partes interesadas es a través de:
- a. Diagrama de despliegue.
 - b. Diagrama de casos de uso.
 - c. Diagrama de contexto.
 - d. Diagrama de clases.

[Ir a solucionario](#)



Semana 5

Como parte del cuerpo de conocimiento de ingeniería de software (SWEBOK) una de las áreas que es parte del presente estudio es la Calidad de Software, que se refiere a las características deseables de los productos de software, y a los procesos, herramientas y técnicas que se utilizan para lograr esas características.

Según la RAE (Real Academia Española), **calidad** se define como la “propiedad o conjunto de propiedades inherentes a algo, que permiten juzgar su valor”. Cuando se habla de la **calidad en software**, se debe considerar que al software lo usan personas (usuarios, clientes) y que es producido y/o modificado por personas (ingenieros de software, arquitectos de software, desarrolladores y otras personas que tienen roles técnicos) cuyas habilidades y conocimientos permiten entregar software de calidad. En consecuencia, la calidad no tiene que ver solo con lo que hace el software, sino también debe tener en cuenta el comportamiento mientras se ejecuta, su estructura interna, la gestión de la construcción y la documentación asociada. Entonces, ¿qué es la calidad del software?, revisemos a continuación algunos conceptos.

Unidad 3. Calidad y atributos de calidad

En semanas anteriores habíamos estudiado algunos conceptos y consideraciones que nos permitirán diseñar una arquitectura de software candidata o base. El concepto calidad y sus atributos tales como funcionalidad, confiabilidad, usabilidad, eficiencia, entre **otros**, son el medio que permite cumplir con los requisitos no funcionales derivados del análisis dentro de un dominio de un problema. Le invito a que revisemos algunos conceptos relacionados con calidad y que le servirán al momento de definir, diseñar e implementar una arquitectura de software.

3.1. Calidad del software

Para Pressman (2010), la “**calidad del software** se define como el proceso eficaz de software que tiene como objetivo crear un producto útil y proporcionar valor medible a quienes lo producen y a quienes lo utilizan”. Según Ian Sommerville (2011), la “calidad del software es el grado en el cual

el software posee una combinación deseada de atributos o requerimientos adicionales que el sistema debe satisfacer”.

En otra definición se menciona que la “calidad de software es la totalidad de rasgos y atributos de un producto de software que le apoyan en su capacidad de satisfacer sus necesidades explícitas o implícitas”, Camacho et al. (2004). De acuerdo a la definición del Instituto de Ingenieros Eléctricos y Electrónicos (IEEE Std 610-1990, 1990), la “calidad del software es el grado con el que un sistema, componente o proceso cumple con la combinación deseada de cualidades o atributos para satisfacer necesidades y expectativas del cliente”. ¿Se ha comprendido?, ahora y como estamos estudiando aspectos de arquitectura de software, le invito a revisar a qué nos referimos cuando se habla de calidad del diseño y calidad arquitectónica.

3.2. Calidad del diseño

La **calidad del diseño** se refiere a las características que los diseñadores especifican para un producto. En desarrollo del software, la calidad del diseño incluye el grado en el que se cumple las funciones y características especificadas en el modelo de requerimientos. La calidad de la conformidad se centra en el grado en el que la implementación se apega al diseño y en el que el sistema resultante cumple sus objetivos de requerimientos y desempeño.

3.3. Calidad de arquitectura o arquitectónica de software

La calidad arquitectónica de software tiene relación con las propiedades externas del sistema, las cuales se manifiestan de dos maneras diferentes: comportamiento externamente visible (lo que hace el sistema) y propiedades de calidad (cómo lo hace el sistema). El **comportamiento externamente visible** de un sistema de software, define las interacciones funcionales entre el sistema y su entorno. Mientras que las **propiedades de calidad** (a menudo denominadas características no funcionales), se refieren a cómo se comporta un sistema desde el punto de vista de un observador externo (cliente o usuario del sistema). Una **propiedad de calidad** es una propiedad no funcional externamente visible de un sistema entre las que constan el rendimiento, la seguridad o la escalabilidad.

Existe una amplia gama de características arquitectónicas externas que pueden ser de interés al momento de definir y diseñar una arquitectura, para ello el arquitecto de software podría hacerse preguntas tales como ¿cuál es el rendimiento máximo esperado del sistema al utilizar cierta infraestructura, despliegue local o *cloud*? , ¿cómo funciona el sistema cuando la carga es mínima y máxima?, ¿el sistema deberá permitir concurrencia de 100, 1000 o más usuarios? , ¿cómo se protege la información del sistema contra vulnerabilidades y accesos no autorizados?, ¿con qué frecuencia es probable que el sistema sufra caídas o fallos que impidan su disponibilidad?, ¿qué tan fácil es administrar, mantener y mejorar el sistema?, ¿con qué facilidad el sistema puede ser usado por personas con alguna capacidad especial?

Hay que tener en cuenta que las propiedades de calidad, tales como seguridad, rendimiento, disponibilidad, usabilidad u otro, varían en su aplicación acorde a cada tipo de sistema. La usabilidad, por ejemplo, es poco probable que sea importante para un proyecto de configuración de la infraestructura con poca o ninguna funcionalidad expuesta a los usuarios. Sin embargo, es probable que algunos tipos de sistemas tengan requisitos de propiedad de calidad similares y formas comunes de cumplirlos.

En otro escenario, por ejemplo, las aplicaciones de software modernas, incluyen características como orquestación, distribución, portabilidad, interoperabilidad, reutilización de componentes y en algunos casos respuestas en tiempo real, lo cual requiere tiempos de respuesta óptimos, por lo tanto, se requiere de una definición adecuada de la arquitectura para abordar estos requerimientos no funcionales.

Bajo estos escenarios, podemos observar que cuando se define una arquitectura candidata, siempre hay que identificar y exhibir las propiedades de calidad y los comportamientos externamente visibles requeridos para implementarlos a nivel de sistema. Recuerde, cuando vamos a diseñar una arquitectura que incluya atributos de calidad, siempre debemos tener presente el tipo de aplicación, ya que de ello dependerá la inclusión de algunas características internas o externas. Luego de conocer conceptos de calidad, es momento de estudiar los conceptos referentes a los atributos de calidad.

3.4. Atributos de calidad

En la bibliografía puede encontrar que a los atributos de calidad se los conoce también como *requisitos no funcionales*, *propiedades* o *características del sistema*, los cuales son útiles para mejorar la calidad del sistema o servicio y tener una gran influencia sobre la arquitectura resultante.

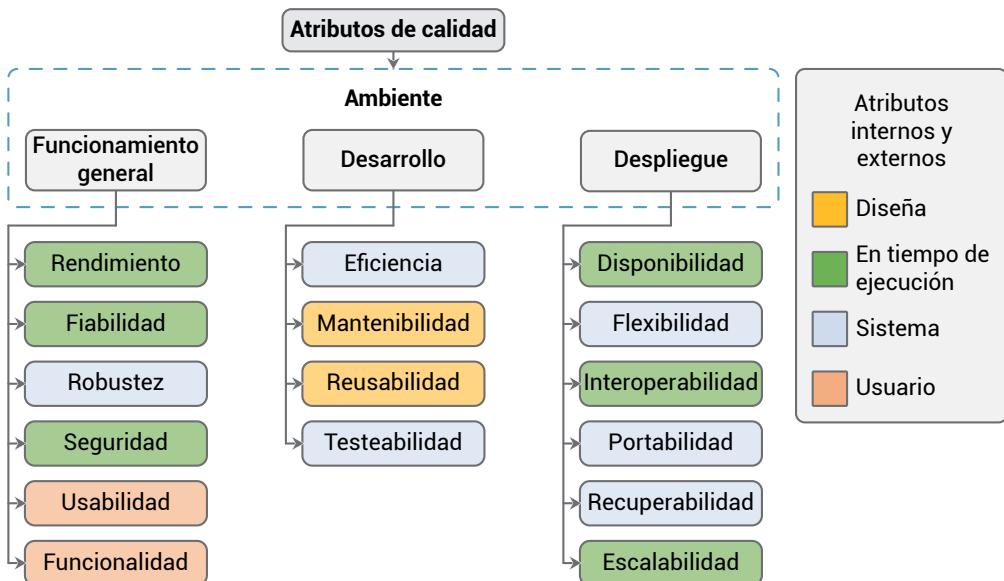
Los atributos de calidad, se definen como las propiedades o características que se desea que tenga un servicio o sistema de *software*, según Jamir Antonio Avila Mojica (2012). Les Bass Len, Paul Clements y Rick Kazman (2013), proponen dos categorías para agrupar a los atributos de calidad: *atributos de desarrollo* y *atributos de operación*. Los **atributos de desarrollo** relevantes desde una perspectiva de ingeniería de *software* están relacionados con facilidad de mantenimiento, facilidad de modificación, facilidad de comprensión, entre otros. Por otro lado, los **atributos de operación** son cualidades que evalúan el sistema en tiempo de ejecución, por ejemplo: rendimiento, confiabilidad, robustez, tolerancia a fallos, seguridad, entre otros.

La norma ISO 9126, como se expone en Pressman (2010), es una norma internacional que se utiliza para la evaluación de *software* y que se divide en cuatro partes que abordan temas como: *modelo de calidad*, *métricas externas*, *métricas internas*, y *métricas de calidad en el uso*. La norma es el resultado de la estandarización de la terminología de los atributos de calidad que se utilizan para construir y evaluar una arquitectura *software* que está guiada por **estilos arquitectónicos**, **patrones arquitectónicos**, **patrones de diseño**, **vistas arquitectónicas**.

La calidad y los atributos de calidad suelen ser transversales a los sistemas de *software*, por lo tanto, deben integrarse desde las bases del sistema que se está definiendo y construyendo. La adaptación de alto rendimiento, escalabilidad, seguridad, disponibilidad, etc. en una base de código ya existente, suele resultar compleja y requiere mucho tiempo, porque los diseños para admitir estas cualidades a menudo se definen en la forma general del sistema de *software* (las estructuras de alto nivel) en sí (ver figura 26).

Figura 26.

Atributos de calidad usados en ambientes de funcionamiento, desarrollo, despliegue y características internas y externas



Nota. Argüello M., & Guamán D., 2023

Los atributos de calidad que podrían aplicarse a los sistemas de software son diversos y no todos tienen la misma ponderación. Algunos son más aplicables que otros, según el ambiente y el tipo de sistema que se construya. Por ejemplo, un sistema para una empresa financiera cuyo funcionamiento sea en la web, probablemente tendrá un conjunto diferente de atributos de calidad a un sistema que funciona en una Intranet y que es utilizado dentro de la industria de las telecomunicaciones y también diferente a un sistema usado dentro de una institución de educación superior. Los atributos de calidad debemos tenerlos en cuenta durante el proceso de definición y diseño, desarrollo, operación y despliegue del software. Le invito a que revisemos el siguiente módulo didáctico, donde se define y expone algunas características de calidad interna y externa definidas en la norma ISO-9126-1 (INEN, 2014), y que nos ayudarán en la definición, diseño e implementación de una arquitectura de software.

Características de calidad interna y externa definido en ISOIEC 9126-1.

Seguramente Ud. ya ha realizado implementaciones de software que incluyan algunos atributos de calidad, ¿verdad?, recuerde que, como parte de la definición, diseño e implementación de una arquitectura de software,

es necesario siempre mirar y entender los atributos y características de calidad requeridas por el sistema. El software no es solamente un conjunto de instrucciones o procedimientos que consiguen un fin, no basta con que funcione, sino que tiene que entenderse como un producto de calidad.

Para complementar se puede mencionar que los atributos de calidad son esenciales en el software, especialmente la disponibilidad, interoperabilidad, flexibilidad, rendimiento, seguridad, testeabilidad. Con base en el estudio de caso propuesto en el [anexo 1](#) se puede encontrar algunos de los atributos de calidad que se resumen en los apartados 4.5 al 4.10 del documento de especificación de requisitos del [anexo 3](#).

En resumen y para su mejor comprensión, a continuación, se exponen algunos escenarios de aplicación en donde intervienen los atributos de calidad.

- **Ambiente de despliegue - Disponibilidad:** la disponibilidad se refiere a una propiedad del software que está listo para llevar a cabo su tarea cuando lo necesita. Esta es una perspectiva amplia y abarca lo que normalmente se llama confiabilidad. Básicamente, la disponibilidad se trata de minimizar el tiempo de interrupción del servicio al mitigar las fallas. Por ejemplo, cuando se requiere realizar una actualización de la aplicación en un ambiente de producción, el sistema deberá mantenerse operativo y activo para que los clientes puedan seguir utilizándolo.

Tabla 3.

Ambiente de despliegue - Disponibilidad

Escenario	Posibles valores
Origen	<i>Interno / externo: personas, hardware, software, infraestructura física, entorno físico</i>
Estímulo	<i>Fallo: omisión, bloqueo, sincronización incorrecta, respuesta incorrecta</i>
Artefacto	Procesadores del sistema, canales de comunicación, almacenamiento persistente, procesos.
Ambiente	Funcionamiento general: arranque, apagado, modo de reparación, funcionamiento degradado, funcionamiento sobrecargado

Escenario	Posibles valores
Respuesta	<p><i>Detectar la falla:</i> registrar la falla notificar a las entidades apropiadas (personas o sistemas).</p> <p><i>Recuperarse ante la falla:</i> deshabilitar la fuente de eventos que originan la falla y que ocasiona que el sistema no esté disponible temporalmente mientras se efectúa la reparación, corregir la falla o contener el daño que causa, operar en un modo degradado mientras se realiza la reparación.</p>
¿Cómo medir la respuesta?	<p>Tiempo o intervalo de tiempo cuando el sistema debe estar disponible.</p> <p>Porcentaje de disponibilidad (por ejemplo 99.999%).</p> <p>Tiempo para detectar la falla.</p> <p>Tiempo para reparar la falla.</p> <p>Tiempo o intervalo de tiempo en el que el sistema puede estar en modo degradado.</p> <p>Proporción (por ejemplo, 99%) o tasa (por ejemplo, hasta 100 por segundo) de una cierta clase de fallos que el Sistema debe prevenir, o manejar sin fallas.</p>

Nota. Argüello M., & Guamán D., 2023

- **Ambiente de despliegue - Interoperabilidad:** la interoperabilidad se refiere al grado en que dos o más sistemas/servicios pueden intercambiar información significativa de manera útil. Como todos los atributos de calidad, la interoperabilidad no es una propuesta de sí o no, sino que tiene matices de significado. Por ejemplo, cuando se requiere transmitir datos en formato JSON o XML desde un servicio externo desplegado en *cloud* hacia una aplicación *w eb*, es claro que debe existir una interface claramente definida entre la aplicación *w eb* y dicho servicio.

Tabla 4.
Ambiente de despliegue - Interoperabilidad

Escenario	Posibles valores
Origen	Un sistema/servicio
Estímulo	Una solicitud para intercambiar información entre sistemas/servicios.
Artefacto	Los sistemas que desean interoperar.
Ambiente	<i>Despliegue:</i> Los sistemas que desean interoperar son descubiertos en tiempo de ejecución o se conocen antes de ejecutar el sistema.

Escenario	Posibles valores
Respuesta	Se puede considerar una o todas de las siguientes: la solicitud es (apropiadamente) rechazada y las entidades apropiadas (personas o sistemas) son notificadas, la solicitud se acepta (apropiadamente) y la información se intercambia con éxito, la solicitud es registrada por uno o más de los sistemas involucrados.
¿Cómo medir la respuesta?	Se puede considerar una o más de los siguientes: porcentaje de intercambios de datos/información correctamente procesados, porcentaje de intercambios de datos/información correctamente rechazados.

Nota. Argüello M., & Guamán D., 2023

- **Ambiente de despliegue - Flexibilidad:** la flexibilidad tiene que ver con el cambio y el costo y el riesgo de realizar dichos cambios. Para planificar la flexibilidad, un arquitecto debe considerar tres preguntas: ¿qué puede cambiar?, ¿cuál es la probabilidad del cambio?, ¿cuándo se realiza el cambio y quién lo hace? Por ejemplo, cuando se requiere añadir un nuevo módulo o funcionalidad sobre la aplicación, está de manera predeterminada, no se podrá visualizar a menos que se configure alguna regla de negocio para que ciertos usuarios con un rol asignado la puedan visualizar.

Tabla 5.
Ambiente de despliegue - Flexibilidad

Escenario	Posibles valores
Origen	Usuarios finales, desarrolladores, administrador del sistema
Estímulo	Una directiva para agregar / eliminar / modificar funcionalidad o cambiar un atributo de calidad, capacidad o tecnología.
Artefacto	Código, datos, interfaces, componentes, recursos, configuraciones, etc.
Ambiente	<i>Despliegue:</i> Runtime, tiempo de compilación, tiempo de construcción, tiempo de iniciación, tiempo de diseño
Respuesta	Se puede considerar una o todas de las siguientes: realizar la modificación, testeo de la modificación, despliegue de la modificación
¿Cómo medir la respuesta?	Costo en términos de: número, tamaño, complejidad de los artefactos afectados, esfuerzo, tiempo calendario, dinero (desembolso directo o costo de oportunidad), grado en que esta modificación afecta otras funciones o atributos de calidad, nuevos defectos introducidos

Nota. Argüello M., & Guamán D., 2023

- **Ambiente de funcionamiento general - Rendimiento:** el rendimiento es cuestión de tiempo y la capacidad del sistema de software para cumplir con los requisitos dentro de dicho tiempo. Cuando ocurren eventos, interrupciones, mensajes, solicitudes de usuarios u otros sistemas, o algún elemento del sistema, debe responder a ellos a tiempo. Por ejemplo, cuando los usuarios que utilizan una aplicación generan 3.000 transacciones por minuto al realizar operaciones normales y estas transacciones se procesan con una latencia promedio de dos segundos.

Tabla 6.

Ambiente de funcionamiento general - Rendimiento

Escenario	Posibles valores
Origen	Internos o externos al sistema
Estímulo	Llegada de un evento periódico, esporádico o estocástico.
Artefacto	Sistema o uno o más componentes en el Sistema.
Ambiente	Modo de operación: normal, emergencia, pico de carga, sobrecarga.
Respuesta	Procesar eventos, cambiar el nivel de servicio
¿Cómo medir la respuesta?	Latencia, tiempo límite, rendimiento, fluctuación, tasa de errores

Nota. Argüello M., & Guamán D., 2023

- **Ambiente de funcionamiento general - Seguridad:** la seguridad es una medida de la capacidad del sistema para proteger los datos y la información del acceso no autorizado y, al mismo tiempo, brindar acceso a personas y sistemas autorizados. Una acción tomada contra un sistema informático con la intención de causar daño se llama ataque y puede adoptar varias formas. Puede ser un intento no autorizado de acceder a datos o servicios o modificar datos, o puede tener la intención de negar servicios a usuarios legítimos. Por ejemplo, cuando una persona autorizada o anónima intenta modificar los datos del sistema desde un sitio externo, bajo este escenario, la aplicación debe tener *logs* de auditoría para identificar dicha acción y en caso de alteración de los datos, estos se podrían restaurar evidenciando los cambios en los *logs*.

Tabla 7.*Ambiente de funcionamiento general - Seguridad*

Escenario	Possibles valores
Origen	Humano u otro sistema que puede haber sido identificado previamente (ya sea correcta o incorrectamente) o que puede ser actualmente desconocido. Una persona que realice el ataque sobre el sistema, puede ser de fuera de la organización o de dentro de la organización.
Estímulo	Se realiza un intento no autorizado para mostrar datos, cambiar o eliminar datos, acceder a los servicios del sistema, cambiar el comportamiento del sistema o reducir la disponibilidad.
Artefacto	Servicios del sistema, datos dentro del sistema, un componente o recursos del sistema, datos producidos o consumidos por el sistema
Ambiente	El sistema está en línea o fuera de línea, conectado o desconectado de una red, detrás de un firewall o abierto a una red, completamente operativo, parcialmente operativo o no operativo
Respuesta	<p>Las transacciones se llevan a cabo de tal manera que los datos o servicios están protegidos contra el acceso no autorizado.</p> <p>Los datos o servicios no se manipulan sin autorización.</p> <p>Las partes de una transacción se identifican con seguridad.</p> <p>Las partes de la transacción no pueden repudiar su participación.</p> <p>Los datos, recursos y servicios del sistema estarán disponibles para uso legítimo.</p> <p>El sistema rastrea las actividades dentro del registro de acceso o modificación, registrar intentos de acceder a datos, recursos o servicios, notificar a las entidades apropiadas (personas o sistemas) cuando se esté produciendo un ataque aparente.</p>
¿Cómo medir la respuesta?	Se puede considerar una o todas de las siguientes: cuánto de un sistema se ve comprometido cuando se ataca un componente o valor de datos en particular, cuánto tiempo pasó antes de que se detectara un ataque, cuántos ataques se resistieron, ¿Cuánto tiempo se tarda en recuperarse de un ataque exitoso?, cuántos datos son vulnerables a un ataque en particular.

Nota. Argüello M., & Guamán D., 2023

- **Ambiente de desarrollo - Testeabilidad:** la capacidad de prueba del software se refiere a la facilidad con la que se puede hacer que el software demuestre sus fallas a través de pruebas (generalmente basadas en la ejecución). Para que un sistema se pueda probar

adecuadamente, debe ser posible controlar las entradas de cada componente (y posiblemente manipular su estado interno) y luego observar sus salidas (y posiblemente su estado interno). Por ejemplo, cuando se realiza las pruebas unitarias sobre un componente del sistema completo que proporciona una interfaz para controlar su comportamiento ante un conjunto de entradas y observar las salidas.

Tabla 8.

Ambiente de desarrollo - Testeabilidad

Escenario	Posibles valores
Origen	Pruebas unitarias, pruebas de integración, pruebas de sistema, pruebas de aceptación, usuarios finales, ya sea ejecutando pruebas manualmente o utilizando herramientas de prueba automatizadas
Estímulo	Se ejecuta un conjunto de pruebas debido a la finalización de un incremento de codificación como una componente, clase, capa o servicio, la integración completa de un subsistema, la implementación completa del sistema, o la entrega del sistema al cliente.
Artefacto	Tiempo de diseño, tiempo de desarrollo, tiempo de compilación, tiempo de integración, tiempo de implementación, tiempo de ejecución
Ambiente	La parte del sistema que se está probando
Respuesta	Se puede considerar una o todas de las siguientes: ejecutar el conjunto de pruebas y capturar los resultados, capturar la actividad que resultó en la falla, controlar y monitorear el estado del sistema.
¿Cómo medir la respuesta?	Se puede considerar una o todas de las siguientes: esfuerzo por encontrar una falla o clase de fallas, esfuerzo por lograr un porcentaje dado de cobertura del sistema, probabilidad de que la siguiente prueba revele la falla, tiempo para realizar pruebas, esfuerzo para detectar fallas, dependencia de componentes para prueba, tiempo para preparar el entorno de prueba, reducción de la exposición al riesgo.

Nota. Argüello M., & Guamán D., 2023

Lecturas en bibliografía básica, complementaria y recursos educativos abiertos

- **Conozca**, a través de la revisión de bibliografía, cómo identificar y proponer el uso de atributos de calidad en su diseño y arquitectura de software.
 - **Libro de Somerville** – (Capítulo 24 – Gestión de la calidad. Subtemas 24.1 Calidad del software).

- **Libro de Presman** – (Capítulo 8 – Conceptos de diseño. Subtemas 8.2.1 Lineamientos y atributos de la calidad del software. Capítulo 14 – Conceptos de calidad. Subtemas 14.1 ¿Qué es calidad?, 14.2 Calidad del software, 14.2.3 Factores de la calidad ISO 9126, 14.2.4 Factores de calidad que se persiguen).
 - **Libro a rquitectura de s oftware, conceptos y ciclo de desarrollo** – (Capítulo 1 – La arquitectura y el desarrollo de software – subtemas 1.3 Arquitectura, atributos de calidad y objetivos de negocio).
 - **DD3_Arquitecturas de s oftware** - Calidad del software (página 5), Calidad arquitectónica (página 8), Atributos de calidad (páginas 9,10), ISO/IEC 9126 adaptado para arquitecturas de software (página 15), Relación entre Arquitectura de Software y Atributos de Calidad (p áginas 16,17,18).
- Revise el [anexo 1](#) donde se propone un ejemplo de un estudio de caso o escenario referente a un dominio del problema.
 - Revise el [anexo 2](#) donde se utiliza el formato de documentación arquitectónica para de forma gráfica y textual describir una posible solución al problema.
 - Revise el [anexo 3](#) donde luego de entender y analizar el dominio del problema, se utiliza el documento de especificación de requisitos para transformar las necesidades dadas por las partes interesadas en requisitos funcionales y no funcionales.

Como podemos evidenciar, los atributos de calidad y su uso combinado son de utilidad en etapas tempranas (diseño) y etapas tardías (implementación y despliegue) del sistema. Para ampliar y reforzar el estudio referente a los atributos de calidad, le invito a desarrollar las siguientes actividades recomendadas.



Actividades de aprendizaje recomendadas

Es hora de poner a prueba los conocimientos relacionados con el uso de atributos de calidad en los sistemas de software.

1. Usted como profesional en formación está familiarizado con el sistema que le permite llevar a cabo los procesos de prematrícula, inscripción y matrícula de la UTPL. Qué atributos piensa usted que se ha considerado o se podría considerar para que el sistema web o móvil que utiliza sea de calidad, especialmente cuando:
 - a. Un usuario que deseé minimizar el impacto de un error desea cancelar una operación del sistema en tiempo de ejecución.
 - b. La cancelación de una operación o transacción errónea se realiza en menos de un segundo.
 - c. Desea que la factura electrónica de la matrícula se valide contra los servicios externos del SRI – Ecuador.
 - d. Requiere utilizar como método de autenticación OAuth.

Nota. conteste las actividades en un cuaderno de apuntes o en un documento Word.

2. Una vez que ha estudiado los conceptos relacionados a la unidad, le invito a desarrollar la autoevaluación con el fin de evaluar los conocimientos adquiridos hasta el momento.



Autoevaluación 3

1. Cuál de las siguientes opciones se consideran correctas ante la afirmación de que los atributos de calidad son los factores generales que afectan:
 - a. El comportamiento del sistema en tiempo de ejecución.
 - b. El diseño de los componentes del sistema.
 - c. La experiencia del usuario.
 - d. El diseño de la base de datos del sistema.
 - e. Todas las anteriores.

2. Cuando se menciona que “el atributo se puede medir como un porcentaje del tiempo total de inactividad del sistema durante un período predefinido”, ¿a qué atributo de calidad nos referimos?
 - a. Seguridad.
 - b. Escalabilidad.
 - c. Mantenibilidad.
 - d. Disponibilidad.

3. Cuando se facilita el intercambio y reutilización de la información tanto interna como externamente de un sistema o entre sistemas, ¿a qué atributo de calidad nos referimos?
 - a. Escalabilidad.
 - b. Interoperabilidad.
 - c. Disponibilidad.
 - d. Rendimiento.
 - e. Todas las anteriores.

4. Cuando el sistema tiene la capacidad de cambiar o adaptarse a cambios sobre componentes, servicios, funciones y las interfaces al agregar o cambiar la funcionalidad de la aplicación para corregir errores o cumplir con nuevos requisitos comerciales, ¿a qué atributos de calidad nos referimos?
- Escalabilidad.
 - Interoperabilidad.
 - Mantenibilidad.
 - Rendimiento.
5. Cuando el sistema se puede medir en términos de latencia o rendimiento, donde Latencia es el tiempo necesario para responder a cualquier evento, ¿a qué atributos de calidad nos referimos?
- Escalabilidad.
 - Interoperabilidad.
 - Mantenibilidad.
 - Rendimiento.
6. Cuando la probabilidad de que un sistema no falle y de que realice su función prevista durante un intervalo de tiempo específico, ¿a qué atributos de calidad nos referimos?
- Escalabilidad.
 - Fiabilidad.
 - Rendimiento.
 - Interoperabilidad.
7. Cuando el sistema tiene la capacidad de manejar aumentos de carga sin impacto en el rendimiento del sistema, o la capacidad de ampliarse fácilmente, ¿a qué atributos de calidad nos referimos?
- Escalabilidad.
 - Fiabilidad.
 - Rendimiento
 - Interoperabilidad.
 - Todas las anteriores.

8. Cuando el sistema tiene la capacidad para reducir la posibilidad de acciones maliciosas o accidentales fuera del uso diseñado que afecten al sistema y evitar la divulgación o pérdida de información, ¿a qué atributos de calidad nos referimos?
- a. Escalabilidad.
 - b. Fiabilidad.
 - c. Rendimiento.
 - d. Seguridad.
9. Cuando se menciona que las interfaces de la aplicación deben diseñarse pensando en el usuario y el consumidor, de modo que sean intuitivas de usar, se puedan localizar y globalizar, brindar acceso a usuarios discapacitados y brindar una buena experiencia de usuario en general, ¿a qué atributos de calidad nos referimos?
- a. Escalabilidad.
 - b. Fiabilidad.
 - c. Experiencia de usuario / Usabilidad.
 - d. Seguridad.
10. Los atributos de calidad en términos del diseño de su aplicación se pueden clasificar en categorías como reusabilidad y mantenibilidad, ¿a qué tipo de categoría estos pertenecen?
- a. Diseño.
 - b. Sistema.
 - c. Tiempo de ejecución.
 - d. Experiencia de usuario.

[Ir a solucionario](#)

Resultado de aprendizaje 2

- Comprende el funcionamiento y aplicabilidad de principios de diseño, estilos y patrones arquitectónicos.

Para alcanzar este resultado de aprendizaje se analizarán los estilos y patrones arquitectónicos, enfocándonos en sus beneficios y enumerando los estilos arquitectónicos más comunes. Esto nos permitirá comprender los principios de diseño, estilos y patrones a aplicar en un diseño de arquitectura.

Contenidos, recursos y actividades de aprendizaje



Semana 6

Una vez que conocemos aspectos relacionados con calidad, en esta semana se estudian los fundamentos teóricos relacionados con estilos arquitectónicos. En gran medida la arquitectura del software tiene como soporte estructural un estilo arquitectónico, un patrón arquitectónico y patrones de diseño que se pueden aplicar de forma unificada o combinada dentro de un contexto, utilizando un vocabulario de diseño formal como UML y teniendo siempre presente el aseguramiento de la calidad del software. Para entender los conceptos de esta semana le invito a revisar el [anexo 2](#) (sección 7 y sección 8) donde se utiliza UML para representar las vistas de desarrollo y despliegue como propuesta de solución al estudio de caso del [anexo 1](#).

A continuación, revisemos algunos conceptos que serán de utilidad en la definición, diseño e implementación de una arquitectura de software.

Unidad 4. Estilos y patrones arquitectónicos

4.1. Patrón de software

Un patrón de software tiene como propósito compartir una solución probada y ampliamente aplicable a un problema de diseño particular, en una forma estandarizada y que se pueda reutilizar fácilmente, según Len

Bass et al. (2013). Para ello, los patrones de software deben ser capaces de proporcionar las siguientes cinco piezas de información (nombre, contexto, problema, solución, consecuencias). Le invito a comprender a qué se refiere cada una de las piezas.

1. **Nombre:** un patrón necesita un nombre significativo que nos permita identificar y discutir claramente el patrón, y lo que es más importante, usar su nombre como parte de nuestro lenguaje de diseño al discutir posibles soluciones a problemas de diseño.
2. **Contexto:** se considera como el escenario o situación común en la cual se puede aplicar el patrón.
3. **Problema:** cada patrón es una solución a un problema particular, por lo que parte de la definición del patrón debe ser una declaración clara del problema que resuelve y cualquier condición o restricción que deba cumplirse para que el patrón se aplique de manera efectiva. Los problemas a los que se refiere son particulares o recurrentes que pueden ocurrir en un contexto de diseño e implementación específico y para los cuales se define estrategias de implementación que incluye el uso de componentes, conectores, reglas y pautas para organizar las relaciones entre ellos.
4. **Solución:** el núcleo del patrón es una descripción de la solución al problema que este aborda. La descripción de la solución también debe especificar qué atributos de calidad proporcionan las configuraciones estáticas y en tiempo de ejecución de los elementos. La solución dada por un patrón está determinada y descrita por:
 - Un conjunto de tipos de elementos (por ejemplo, repositorios de datos, procesos y objetos).
 - Un conjunto de conectores o mecanismos de interacción (por ejemplo, llamadas a métodos, eventos o bus de mensajes).
 - Un diseño topológico o representación gráfica de los componentes.
 - Un conjunto de restricciones semánticas que cubren la topología, el comportamiento de los elementos y los mecanismos de interacción.

5. **Consecuencias:** la definición de un patrón de software debe incluir una declaración clara de los resultados y compensaciones que resultarán de su aplicación, con el objetivo de que le permitan decidir si es una solución adecuada al problema. Las consecuencias pueden ser positivas (beneficios) o negativas (costos).

Los patrones de software generalmente se organizan en tres grupos: *estilos arquitectónicos* que registran soluciones a nivel del sistema, *patrones arquitectónicos* que registran soluciones a problemas detallados de diseño de software y *expresiones idiomáticas* que capturan soluciones útiles para problemas específicos del lenguaje. Empecemos nuestro estudio conociendo más acerca de los estilos arquitectónicos, los cuales se aplican a estructuras a nivel de sistema.

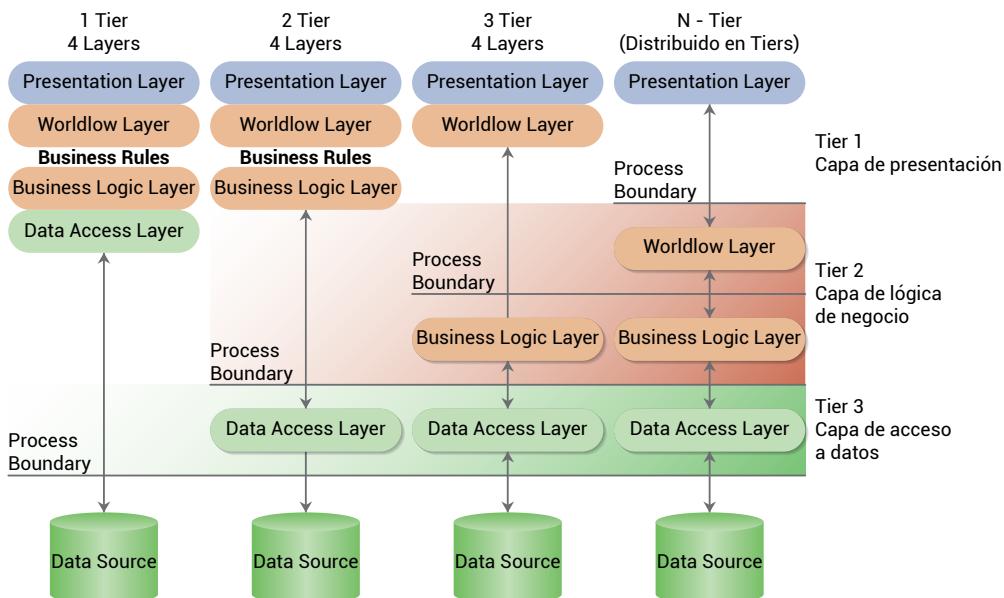
4.2. Estilos arquitectónicos

Según Richard Taylor y Nenad Medvidovic (2012), los “estilos arquitectónicos determinan un conjunto de reglas de diseño que identifican tipos de componentes, conectores y patrones que se pueden utilizar para constituir un sistema o subsistema de software junto con restricciones locales y globales de composición.”

Un estilo arquitectónico expresa un esquema de organización estructural fundamental para los sistemas de software, ya que proporciona un conjunto de tipos de elementos predefinidos, especificando sus responsabilidades e incluyendo reglas y pautas para organizar las relaciones entre ellos” (ver figura 27).

Figura 27.

Representación de una arquitectura distribuida en Capas (Arquitectura física)



Nota. Tomado de *N-Tier/(Layer) Architecture in C# [Ilustración]*, por Arghandabi, H., 2019, medium.com. CC BY 2.0

Para Len Bass (2013), los “estilos arquitectónicos se consideran como una colección de decisiones de diseño arquitectónico que son específicos para el sistema dentro de un contexto de desarrollo dado y recaban cualidades beneficiosas de cada sistema resultante”. Reflejan menos especificidad de dominio que los patrones arquitectónicos y son útiles para determinar todo, desde la estructura de subrutinas hasta estructura de aplicación de nivel superior.



Recuerde complementar estos conceptos con los que usted pueda buscar en bibliografía complementaria. Una vez conocidos los conceptos, revisemos cuáles son los beneficios que nos brindan el utilizar los estilos arquitectónicos.

4.3. Beneficios de los estilos arquitectónicos

Los estilos se pueden utilizar durante el proceso de definición y diseño como paso previo a la implementación de la arquitectura de varias formas. Richard Taylor y Nenad Medvidovic (2012), exponen los siguientes beneficios que tiene el utilizar estilos arquitectónicos:

- a. **Reutilización:** a nivel de diseño, cuando las soluciones son bien entendidas se pueden aplicar a nuevos problemas. A nivel de codificación, las implementaciones a nivel de componentes y el uso de estructuras se pueden compartir con otros estilos.
- b. **Comprensión:** a nivel de diseño, la representación y organización gráfica de elementos UML. A nivel de codificación, la organización o modularidad del sistema (por lo general en paquetes que incluyen clases y/o archivos).
- c. **Interoperabilidad:** a nivel de codificación compatible con la estandarización de estilos y patrones.
- d. **Estilo-especificidad:** desde el punto de vista de análisis, codificación e implementación habilitado por el espacio de diseño restringido correspondiente a experiencias o implementaciones previas utilizando un estilo o su combinación con algún patrón.
- e. **Base para la adaptación:** al considerar los estilos existentes, puede encontrar que ninguno de ellos realmente resuelve su problema, pero lo abordan parcialmente o lo abordan con algunas limitaciones. En estas situaciones, el estilo constituye un punto de partida para el proceso de diseño, pero actúa como una base para adaptarse a las limitaciones particulares de la situación actual.

Como lo menciona Nick Rozanski (2008), los “estilos arquitectónicos se diferencian por su objetivo y características de diseño e implementación, permitiendo la creación de arquitecturas con propiedades diferentes, asociando el estilo con atributos de calidad”. Definir una arquitectura con un estilo conocido puede tener dos beneficios inmediatos. En primer lugar, porque el seleccionar una solución probada y bien entendida para un problema, ayuda en la definición de los principios estructurales del sistema. En segundo lugar, si se conoce que la arquitectura se basa en un estilo conocido, esto ayuda a comprender sus características importantes.

Para lograr los beneficios que nos proporcionan los estilos arquitectónicos, se sugiere tener en cuenta algunas propiedades básicas, entre las que destacan:

- a. **Usar un vocabulario de elementos de diseño:** debido a que para su representación gráfica y formal se hace uso de tipos de componentes, conectores y elementos de datos. Por ejemplo, tuberías, filtros, objetos, servidores, entre otros.
- b. **Usar un conjunto de reglas de configuración:** especialmente las restricciones topológicas que determinan las composiciones permitidas de elementos. Por ejemplo, un componente puede estar conectado como máximo a otros dos componentes.
- c. **Analizar sus componentes y conectores:** los estilos arquitectónicos definen los componentes y conectores. Donde un conector de software es un bloque de construcción arquitectónico encargado de efectuar y regular interacciones entre componentes, por lo tanto, a nivel de diseño y codificación debemos tener en cuenta el rol que tiene un conector, el cual puede ser:
 - Conector de llamada a procedimiento (*remoting process call*).
 - Conector de datos compartidos (base de datos compartida).
 - Conector de paso de mensajes.
 - Conector de transmisión.
 - Conector de distribución.

Le invito a que revisemos los tipos que puede adoptar un conector, especialmente cuando vamos a llevar a cabo actividades de diseño e implementación con algún lenguaje de programación.

Ahora que usted conoce que son los estilos arquitectónicos, beneficios y algunas propiedades, seguro se ha de preguntar, ¿cómo se agrupan o clasifican los estilos arquitectónicos para comprender mejor su uso dentro de un contexto? En el siguiente tema damos respuesta a esta pregunta, empecemos.

Lecturas en bibliografía básica, complementaria y recursos educativos abiertos

■ Semana 6

Revise la bibliografía complementaria que le permitirá conocer los conceptos, tipología, ventajas, desventajas y uso de los estilos arquitectónicos más comunes.

- **Libro de Presman** – (Capítulo 9 – Diseño de la arquitectura. Subtemas 9.3 Estilos arquitectónicos, 9.3.1 Breve taxonomía de estilos de arquitectura).
- **Libro, a rquitectura de s oftware, conceptos y ciclo de desarrollo** – (Capítulo 3 – Toma de decisiones para crear estructuras – subtemas 3.4.1 Patrones).
- **DD3_Arquitecturas de s oftware** - Estilo Arquitectónico (páginas 19, 20).



Semana 7

4.4. Estilos arquitectónicos comunes

Los estilos arquitectónicos tienden a ser bastante unidimensionales en el sentido de que se enfocan en resolver un tipo de problema muy específico. Esto significa que, en todos los sistemas, excepto en los más simples (como algoritmos o problemas pequeños que se resuelven con software), normalmente es necesario combinar varios estilos para satisfacer los diversos problemas de diseño que la mayoría de los sistemas le presentan.

Entre los estilos arquitectónicos más comunes constan los siguientes:

- a. Estilos tradicionales influenciados por el lenguaje de programación (programa principal y subrutinas, orientado a objetos).
- b. En capas (máquinas virtuales, cliente-servidor).
- c. Flujo de datos (procesamiento por lotes, tuberías y filtros).
- d. Invocación implícita (publicador-suscriptor, dirigido por eventos).

A continuación, se presenta en resumen lo referente al estilo, los elementos de diseño que debemos tener en cuenta, la topología que se propone, ejemplos de uso, algunas ventajas, desventajas y sugerencias de cuando utilizar y evitar el uso de un estilo en específico.

Resumen: estilo, elementos de diseño, topología, ejemplos de uso, ventajas, desventajas y sugerencias.

Luego de visualizar en resumen los estilos arquitectónicos más comunes, seguro ha utilizado algún estilo arquitectónico, ¿verdad? Yo diría que sí, recuerde cuando usted cursó asignaturas como Programación de algoritmos, Programación avanzada, donde es básico el uso del estilo arquitectónico, *programa principal* y *subrutinas*, o *programación orientada a objetos*.

Conforme avanzó en sus estudios desarrolló aplicaciones web o móviles y lo más probable es que haya usado estilos arquitectónicos tales como Capas, *Publicador-suscriptor* o dirigido por eventos donde hizo uso de API o ESB para escribir servicios SOA o REST, ¿verdad?, es que a veces nos pasa eso, codificamos muchas de las veces sin un diseño base, pero olvidamos que los frameworks de desarrollo tienen como base el uso de un estilo arquitectónico o patrones arquitectónicos como lo revisaremos en las siguientes semanas.

Lecturas en bibliografía básica, complementaria y recursos educativos abiertos

- **Semana 7**
 - Revise la bibliografía complementaria que le permitirá conocer los conceptos, tipología, ventajas, desventajas y uso de los estilos arquitectónicos más comunes.
Libro de Somerville – (Capítulo 6 – Diseño arquitectónico. Subtemas 6.3.3 Arquitectura cliente – servidor, 6.3.4 Arquitectura de tubería y filtro, 6.4 Arquitectura de aplicación).
 - Revise el anexo 1 donde se propone un ejemplo de un estudio de caso o escenario referente a un dominio del problema.

- Revise el anexo 2 donde se utiliza el formato de documentación arquitectónica para de forma gráfica y textual describir una posible solución al problema (ver a nexo 2).
- Revise el anexo 3 donde luego de entender y analizar el dominio del problema, se utiliza el documento de especificación de requisitos para transformar las necesidades dadas por las partes interesadas en requisitos funcionales y no funcionales.

Para finalizar el estudio en esta semana, le invito a realizar las actividades de aprendizaje recomendadas y recuerde que diferentes estilos nos dan como resultado diferentes arquitecturas, cada una con propiedades muy diferentes. Un estilo no determina completamente la arquitectura resultante, pero si nos ayuda a tener un juicio individual de aplicación respecto de otros arquitectos o ingenieros de software. Un estilo define el dominio del discurso acerca del problema (dominio del problema) y acerca del sistema resultante (dominio de solución) acorde al tipo de aplicación que tenemos en mente construir.



Actividades de aprendizaje recomendadas

Una vez que ha revisado y estudiado los conceptos relacionados con estilos arquitectónicos, le invito a que gráficamente represente los siguientes escenarios:

1. Todos los ajustes de configuración, el entorno de la interfaz gráfica de usuario, la lógica de datos y el sistema de lógica de negocio se alojan en un mismo servidor. La representación gráfica debe mostrar que uno o muchos clientes realizan peticiones de datos sobre el servidor.
2. La arquitectura proporciona un entorno donde la interfaz gráfica de usuario y la lógica de negocio se ejecuta en un servidor y los datos se almacenan en otro servidor. La representación gráfica debe mostrar que los clientes realizan una petición al servidor donde se aloja la interfaz de usuario y la lógica de negocio y que la comunicación entre ambos servidores es a través de un *middleware*.
3. La arquitectura proporciona un entorno donde la interfaz gráfica de usuario se ejecuta en un servidor, la lógica de negocio se ejecuta en

otro servidor y los datos se almacenan otro servidor. La arquitectura requiere usar un *middleware* porque si el cliente envía la solicitud al servidor donde se aloja la aplicación, primero esta solicitud es recibida por una capa intermedia y finalmente esta solicitud se envía a cada servidor.

Nota. conteste las actividades en un cuaderno de apuntes o en un documento Word.



Semana 8

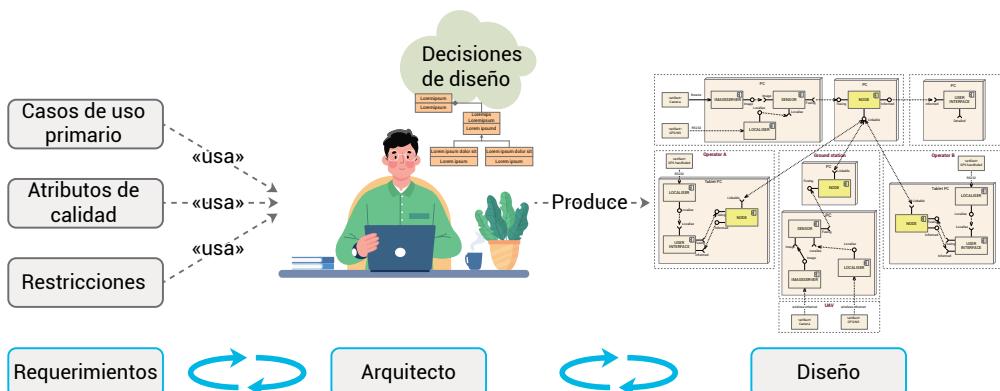


Actividades finales del bimestre

Es importante que vaya asociando los conceptos teóricos con la práctica en escenarios donde haciendo uso de herramientas de modelado e implementación a nivel de lenguajes de programación, involucre las tendencias, tecnologías de ingeniería y arquitectura de software para construir soluciones software con calidad. Otro de los objetivos del presente curso es utilizar un **lenguaje común, nomenclatura definida, estandarizada** y sobre todo tener el criterio para definir y diseñar arquitecturas de software para que las pueda implementar/codificar. Recuerde que los conceptos de arquitectura están definidos y es responsabilidad de los ingenieros y arquitectos de software llevar a cabo actividades de análisis, diseño y construcción de software donde haga uso de documentos y diagramas propios de la arquitectura (ver figura 28).

Figura 28.

El arquitecto de software y sus actividades relacionadas



Nota. Argüello M., & Guamán D., 2023



Actividades de aprendizaje recomendadas

Desarrolle el pequeño banco de preguntas que se cargará en la plataforma educativa para su autoestudio previo a la evaluación presencial.



Segundo bimestre

Resultado de aprendizaje 2

- Comprende el funcionamiento y aplicabilidad de principios de diseño, estilos y patrones arquitectónicos.

Para alcanzar este resultado de aprendizaje se analizaran los estilos y patrones arquitectónicos, enfocándonos en sus beneficios y enumerando los estilos arquitectónicos más comunes. Esto nos permitirá comprender los principios de diseño, estilos y patrones a aplicar en un diseño de arquitectura.

Contenidos, recursos y actividades de aprendizaje



Semana 9

4.5. Estilos y patrones arquitectónicos: patrones arquitectónicos

Continuamos con el estudio ahora de los patrones arquitectónicos, los cuales son utilizados para el análisis, diseño e implementación de abstracciones específicas de datos, funciones e interconexiones que, a través de un vocabulario y plantillas base de diseño, permiten el cumplimiento de atributos de calidad y reutilización de componentes a nivel de implementación o código. A continuación, revisemos algunas definiciones y tipos de patrones arquitectónicos más comunes.

■ Patrones arquitectónicos

Un patrón arquitectónico es un paquete de decisiones de diseño que se encuentra repetidamente en la práctica, tiene propiedades conocidas que permiten la reutilización de componentes, y describe una clase de arquitectura. El Software Engineering Institute (SEI) define a los patrones arquitectónicos como “una descripción de elementos y tipos de relaciones junto con una serie de restricciones sobre cómo estos son usados”. Taylor y Medvidovic (2012), mencionan que un “patrón arquitectónico es

un conjunto de decisiones de diseño arquitectónico que se aplican a un problema de diseño recurrente, y se parametrizan para tener en cuenta los diferentes contextos de desarrollo de software en los que ese problema aparece”.

En otra definición, Brown (2018), mencionan que “los patrones arquitectónicos son considerados dentro de la arquitectura software como plantillas específicas definidas por tipos de elementos y relaciones que se utilizan para solventar de diferentes formas un problema acorde a un contexto dado, utilizando para ello puntos de vista y vistas arquitectónicas como las propuestas en el modelo 4+1 View”.

Para Rozanski (2008), “los patrones arquitectónicos son una disciplina de resolución de problemas en ingeniería del software que ha surgido con mayor énfasis en la comunidad de orientación a objetos, aunque pueden ser aplicados en cualquier ámbito de la informática”. De forma concreta, Camacho et al. (2004) en su trabajo definen a los “patrones arquitectónicos como plantillas específicas que aportan una forma de resolución de problemas dentro de arquitecturas de software”.

En consecuencia, a las teorías expuestas, los patrones arquitectónicos detallan una solución puntual a inconsistencias de software, por tanto, los patrones presentan menor alcance que los estilos arquitectónicos. Los patrones de arquitectura forman parte de la llamada arquitectura lógica de un sistema, que de forma resumida comprende:

- El diseño de más alto nivel de la estructura interna del sistema.
- Los patrones y abstracciones necesarios para guiar la construcción del sistema de software.
- Los fundamentos para que analistas, diseñadores, programadores, testers, etc., trabajen en una línea común que permita cubrir restricciones y alcanzar los objetivos del sistema. Los objetivos del sistema, no son solamente funcionales, sino que también incluyen el mantenimiento, auditoría, flexibilidad e interacción con otros sistemas. Mientras que las restricciones limitan la construcción del sistema acorde a las tecnologías disponibles (*hardware* y *software*) para su implementación.

¿Existe alguna relación entre los tipos de estilos arquitectónicos y patrones?, para dar contestación a esta pregunta le invito a revisar algunos conceptos, características, ventajas y desventajas de los patrones arquitectónicos más comunes, entre los que destacan: Capas (*Layers*), Modelo-Vista-Controlador (MVC), Broker, Servicios (SOA), Microservicios.

- **Capas (*Layers*)**

Le invito a que revisemos el siguiente módulo didáctico, donde se define y expone algunas características del patrón arquitectónico en capas.

[**Capas \(*Layers*\).s.**](#)

- **Modelo-Vista-Controlador (*Model-View-Controller*)**

[**Modelo-Vista-Controlador \(*Model-View-Controller*\)**](#)

- ***Broker Pattern***

[**Broker Pattern**](#)

Se había comentado que el Web Service es un ejemplo del patrón *Broker*, para comprender mejor y como paso previo para entender la Arquitectura Orientada a Servicios (SOA) y Microservicios, es importante que revisemos que son los servicios. Los servicios web proporcionan rutinas que pueden ser utilizadas por cualquier desarrollador cuando hace uso de una arquitectura de software. Los servicios pueden estar alojados en un servidor, además son una forma de implementar la arquitectura orientada a servicios y son independientes de la plataforma con la que SOA puede descomponer las aplicaciones. A continuación, se expone el tema de servicios web.

- **Servicios web**

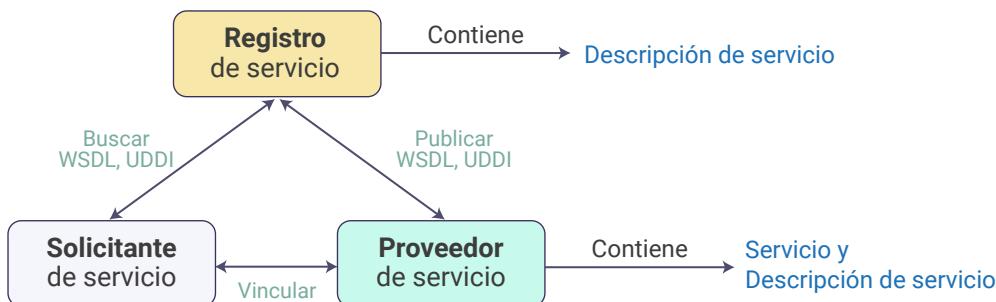
Según el W3C *Working Group* (2004), los servicios web son considerados como un software que permite la interacción entre diferentes máquinas conectadas por *Internet*, posee una interfaz WSDL (*Web Services Description Language*, por sus siglas en inglés) que se comunica con el cliente por medio de mensajes SOAP (*Simple Object Access Protocol*) en formato XML (*eXtensible Markup Language*), comúnmente transportados a través del protocolo HTTP (*Hypertext Transfer Protocol*) en conjunto con otros estándares de los servicios web. Según Sommerville (2011), un servicio web

es una función ofrecida, por una parte, a otra. Aunque el proceso puede asociarse a un proceso físico, la función es esencialmente intangible y, por lo general, no da por resultado la propiedad de alguno de los factores de producción. López (2011), menciona que los servicios web se describen así mismos como la lógica de negocio expuesto en la web como servicios por medio de interfaces y el uso de protocolos de Internet, que pueden ser buscados, suscritos e invocados por otro sistema o servicio.

La arquitectura de servicios web describe cómo crear instancias de los elementos e implementar las operaciones de manera interoperable. La arquitectura del servicio web interactúa entre **tres roles**: **proveedor de servicios, solicitante de servicios y registro de servicios**. La interacción involucra las **tres operaciones**: **publicar, buscar y vincular**. Estas operaciones y roles actúan sobre los artefactos de servicios web. Los artefactos del servicio web son el **módulo de software del servicio web y su descripción**.

El proveedor de servicios aloja un módulo asociable a la red (servicio web). Define una descripción de servicio para el servicio web y la publica en un solicitante de servicio o registro de servicio. Este solicitante de servicio utiliza una operación de búsqueda para recuperar la descripción del servicio localmente o desde el registro del servicio. Utiliza la descripción del servicio para vincularse con el proveedor de servicios e invocar con la implementación del servicio web. La figura 32 muestra las operaciones, los roles y su interacción.

Figura 29.
Roles, operaciones y artefactos del Servicio Web



Nota. Argüello M., & Guamán D., 2023

El **proveedor de servicio**, desde una perspectiva arquitectónica, es la plataforma la que aloja los servicios. El **solicitante de servicio** es la

aplicación que busca e invoca o inicia una interacción con un servicio. El navegador desempeña el papel de solicitante, impulsado por un consumidor o un programa sin interfaz de usuario. El **registro de servicio** es necesario para que los solicitantes de servicios encuentren el servicio y obtengan información vinculante para los servicios durante el desarrollo.

En la operación de **publicación**, se debe publicar una descripción del servicio para que un solicitante de servicio pueda encontrar el servicio. En la operación de **búsqueda**, el solicitante del servicio recupera la descripción del servicio directamente. Puede estar involucrado en dos fases diferentes del ciclo de vida del solicitante del servicio, a nivel de diseño, para recuperar la descripción de la interfaz del servicio para el desarrollo del programa, y en tiempo de ejecución, para recuperar el enlace del servicio y la descripción de la ubicación para la invocación. En la operación de **vinculación**, el solicitante del servicio invoca o inicia una interacción con el servicio en tiempo de ejecución utilizando los detalles de vinculación en la descripción del servicio para localizar, contactar e invocar el servicio. Le invito a conocer más acerca del concepto de servicio web, sus características y algunas ventajas en el siguiente módulo didáctico.

Servicios web

Bien, es momento de conocer el tema de la Arquitectura Orientada a Servicios (SOA).

- **Service-Oriented Architecture (SOA)**

Arquitectura Orientada a Servicios (*Service Oriented Architecture*, en sus siglas en inglés), tiene varios enfoques: de negocio, de tecnología y de organización. Desde el punto de vista de negocio, SOA permite a las organizaciones satisfacer los cambios en las necesidades de la empresa, mediante la implementación de servicios web. Desde el punto de vista tecnológico aumenta la flexibilidad, simplificando la adaptación de los sistemas existentes, mejorando la productividad de procesos y la usabilidad de las aplicaciones. Desde el punto de vista organizacional permite la consistencia en los procesos, ofrece rapidez de adaptación al cambio y mejora la cultura de servicio.

El SEI menciona que la arquitectura orientada a servicios es una forma de diseño, desarrollo, implementación y gestión de sistemas, en el cual:

- Los servicios proporcionan funcionalidad de negocio reutilizable.
- Los consumidores de servicios se construyen utilizando la funcionalidad de los servicios disponibles.
- Las definiciones de interfaz de servicio son artefactos de primera clase.
- Una infraestructura SOA permite el descubrimiento, la composición y la invocación de servicios.
- El intercambio de documentos basados en mensajes.

Serrano (2007), conceptualiza la arquitectura orientada a servicios en términos de servicios programables que ofrecen una funcionalidad encapsulada, débilmente acoplados, proporcionados por aplicaciones nuevas o existentes, con la finalidad de poder ser reutilizados por aplicaciones finales u otros servicios (clientes).

En SOA se hace uso de un bus de mensajes conocido generalmente como ***Enterprise Service Bus (ESB)***. La arquitectura del bus de mensajes describe el principio de utilizar un sistema de software que puede recibir y enviar mensajes utilizando uno o más canales de comunicación, para que las aplicaciones puedan interactuar sin necesidad de conocer detalles específicos entre sí. Un bus es un estilo para diseñar aplicaciones en las que la interacción entre aplicaciones se logra pasando mensajes (generalmente asíncronos) sobre un bus común. Las implementaciones más comunes de la arquitectura del bus de mensajes utilizan un enrutador de mensajería o un estilo de publicación - suscripción, y a menudo se implementan utilizando un sistema de mensajería como *Message Queue Server*. Muchas implementaciones consisten en aplicaciones individuales que se comunican utilizando esquemas comunes y una infraestructura compartida para enviar y recibir mensajes.

Un bus de mensajes ofrece la capacidad de manejar:

- **Comunicaciones orientadas a mensajes.** Toda comunicación entre aplicaciones se basa en mensajes que utilizan esquemas conocidos.
- **Lógica de procesamiento compleja.** Las operaciones complejas se pueden ejecutar combinando un conjunto de operaciones más

pequeñas, cada una de las cuales admite tareas específicas, como parte de un itinerario de múltiples pasos.

- **Modificaciones a la lógica de procesamiento.** Debido a que la interacción con el bus se basa en esquemas y comandos comunes, puede insertar o eliminar aplicaciones en el bus para cambiar la lógica que se usa para procesar los mensajes.
- **Integración con diferentes entornos.** Al utilizar un modelo de comunicación basado en mensajes y estándares comunes, puede interactuar con aplicaciones desarrolladas para diferentes entornos, como Microsoft .NET y Java.

Las variaciones en el estilo del bus de mensajes incluyen:

- **Enterprise Service Bus (ESB).** Basado en los diseños de bus de mensajes, un ESB usa servicios para la comunicación entre el bus y los componentes conectados al bus. Un ESB usualmente proporcionará servicios que transforman mensajes de un formato a otro, permitiendo a los clientes que usan formatos de mensajes incompatibles comunicarse entre sí.
- **Bus de Servicio de Internet (ISB).** Esto es similar a un bus de servicio empresarial, pero con aplicaciones alojadas en la nube en lugar de en una red empresarial. Un concepto básico de ISB es el uso de Identificadores de Recursos Uniformes (URI) y políticas para controlar el enrutamiento de la lógica a través de aplicaciones y servicios en la nube.

Continúe profundizando su conocimiento a través del siguiente módulo didáctico.

[Service-Oriented Architecture \(SOA\).](#)

Posterior a los servicios existe un concepto como microservicios, que si hacemos una analogía hemos empezado desde una aplicación monolítica, luego pasamos por el tema de servicios y estamos en capacidad de conocer los microservicios, pensemos, ¿qué ocurre cuando un vaso (monolito) se cae y se rompe? Puede dividirse en 2 o 3 partes verdad, pero, ¿qué ocurre si lo empezamos a esas 2 o 3 partes las hacemos más pequeñas?, es lo que nos da una pauta para hablar de microservicios.

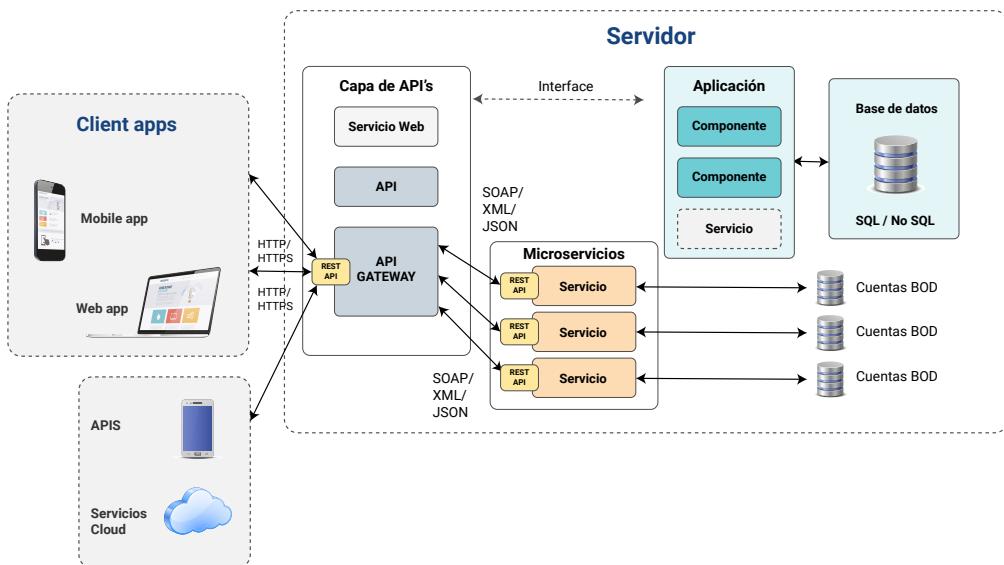
- **Microservicios**

Haciendo una analogía, hablar de microservicios es descomponer una aplicación en pequeñas partes, mucho más pequeñas que los servicios, es decir, tener un conjunto de microservicios, los cuales se convierten en varias pequeñas aplicaciones que funcionarán cada una con su propio repositorio de datos. El siguiente paso luego de los servicios es llegar a los microservicios, los cuales tienen su propia responsabilidad y una de las ventajas es que cuando se diseñan e implementan los equipos de desarrollo pueden trabajarlos de manera independiente a otros microservicios, la única dependencia entre ellos es la comunicación. A medida que los microservicios se comunican entre sí, deberá asegurarse de que los mensajes enviados entre ellos sigan siendo compatibles con versiones anteriores. Esto requiere cierta coordinación, especialmente cuando diferentes equipos son responsables del desarrollo, pruebas y despliegue de diferentes microservicios.

Evaluemos el siguiente ejemplo, donde tenemos una aplicación a través de la cual un usuario puede llevar a cabo el pedido y pago por concepto de la compra de un producto. Para realizar el proceso de pedido y pago se diseñan 4 pequeños servicios o microservicios los cuales deben ser llamados desde la aplicación utilizando para ello una API (*Application Programming Interface*) (ver figura 30).

Figura 30.

Ejemplo de diseño y uso de Microservicios



Nota. Argüello M., & Guamán D., 2023

El uso de una API Gateway o puerta de enlace es una de las características de los microservicios. En la figura 36 se visualiza que la aplicación llama a una API central que reenvía la llamada al microservicio que se requiera en un momento dado. En este ejemplo, se visualizan los servicios separados para el perfil de usuario, el inventario, los pedidos o la orden y el pago. Entre microservicios puede existir también comunicación entre sí. Por ejemplo, el servicio de pago puede notificar al servicio de orden cuando un pago se realiza correctamente. El servicio de orden podría llamar al servicio de inventario para ajustar el stock. No existe una regla clara de cuán grande puede ser un microservicio.

En el ejemplo anterior, el servicio de perfil de usuario puede ser responsable de datos como el nombre de usuario y la contraseña de un usuario, pero también la dirección de casa, imagen de perfil, favoritos, etc. También podría ser una opción para dividir todas esas responsabilidades en aún más microservicios. Algunas de las ventajas y desventajas que este patrón ofrece se muestran en la tabla 9.

Tabla 9.*Ventajas y desventajas de los microservicios*

Ventajas	Desventajas
Puede escribir, mantener e implementar cada microservicio por separado.	Al contrario de lo que cabría esperar, en realidad es más fácil escribir un monolito bien estructurado al principio y luego dividirlo en microservicios.
Una arquitectura de microservicios debería ser más fácil de escalar, ya que solo puede escalar los microservicios que necesitan escalar. No es necesario escalar las partes de la aplicación que se utilizan con menos frecuencia.	Con los microservicios, entran en juego muchas preocupaciones adicionales: comunicación, coordinación, compatibilidad con versiones anteriores, registro, etc.
Es más fácil reescribir partes de la aplicación porque son más pequeñas y están menos acopladas a otras partes	<p>Los equipos que pierdan la habilidad necesaria para escribir un monolito bien estructurado probablemente tendrán dificultades para escribir un buen conjunto de microservicios.</p> <p>Una sola acción de un usuario puede pasar a través de varios microservicios.</p> <p>Hay más puntos de falla, y cuando algo sale mal, puede llevar más tiempo identificar el problema</p>

Nota. Argüello M., & Guamán D., 2023

Con base a las ventajas y desventajas descritas para el patrón, es importante considerar que el patrón de microservicios es ideal para:

- Aplicaciones en las que determinadas piezas se utilizarán de forma intensiva y deben escalarse.
- Servicios que brindan funcionalidad a varias otras aplicaciones.
- Aplicaciones que se volverían muy complejas si se combinaran en un monolito.
- Aplicaciones en las que se pueden definir contextos delimitados claros.

¿Se ha comprendido los temas relacionados con estilos y patrones arquitectónicos?, recordemos que los patrones y su diseño se verifican implementándolos usando lenguajes y tecnologías de la programación, es por ello que le invito a revisar algunos ejemplos que se cargarán en la

plataforma educativa virtual y que espero sean de ayuda para el desarrollo de sus entregables.

Bien, una vez que hemos revisado los patrones arquitectónicos más utilizados, especialmente cuando se construye sistemas de *software* a nivel empresarial, le invito a que complemente su estudio revisando bibliografía complementaria disponible en la *web* o en recursos educativos abiertos. Seguramente en este momento le llama la atención conocer la diferencia o similitudes entre estilos y patrones, ¿verdad?, revisemos el siguiente apartado el cual puede ayudar a dar respuesta a la pregunta.



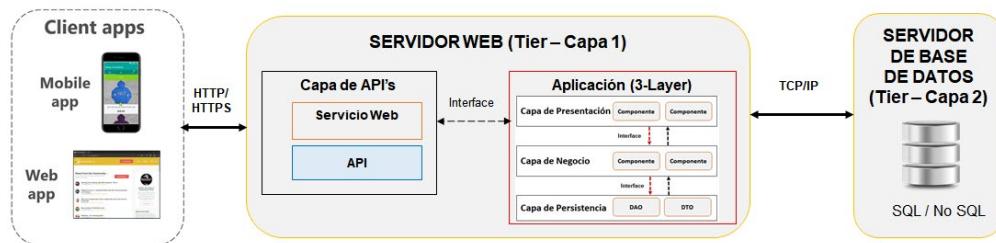
Semana 10

4.6. Patrón arquitectónico vs. Patrón de diseño

Sería interesante en este momento recordar los conceptos referentes a estilos y patrones. Empezaremos mencionando que los estilos arquitectónicos definen de forma general los componentes y conectores que debería tener una arquitectura de *software*. Los estilos dan respuesta al ¿qué?, se consideran menos específicos de dominio, lo que implica que nos permite visualizar de forma general cómo se estructurará física o lógicamente nuestro diseño de solución. Por otro lado, los patrones arquitectónicos definen las estrategias de implementación o codificación de esos componentes y conectores. Los patrones arquitectónicos dan respuesta al ¿cómo?, y se consideran más específicos de dominio, lo que implica que tenemos un poco más de detalle en los componentes y su lógica de implementación (ver figura 31).

Figura 31.

Estilo 2 Capas – Tiers (Físicas) vs Patrón arquitectónico 3 Capas – Layers (Lógicas)



Nota. Argüello M., & Guamán D., 2023

Es importante incluir un concepto relacionado con los patrones de diseño, a los cuales se consideran con nivel granular más fino que los anteriores y son independientes del dominio. Aquí cabe resaltar que una buena arquitectura por lo general hace uso de estilos, patrones o su combinación, teniendo en cuenta siempre la calidad. Como en nuestro estudio nos hemos enfocado en los estilos y patrones arquitectónicos, le invito a revisar algunas diferencias que se exponen en la figura 31.

Tabla 10.

Diferencias entre estilos arquitectónicos y patrones arquitectónicos

Estilo arquitectónico	Patrón arquitectónico
Solo describe el esqueleto estructural y general de las aplicaciones.	Existen en varios rangos de escala, comenzando con patrones que definen la estructura básica de una aplicación.
Son independientes del contexto al que puedan ser aplicados.	Partiendo de la definición de patrón, requieren de la especificación de un contexto del problema.
Cada estilo es independiente de los otros.	Depende de patrones más pequeños que contiene, patrones con los que interactúa, o de patrones que lo contengan.
Expresan técnicas de diseño desde una perspectiva que es independiente de la situación actual del diseño.	Expresa un problema recurrente de diseño muy específico, y presenta una solución para él, desde el punto de vista del contexto en el que se presenta.
Son una categorización de sistemas.	Son soluciones generales a problemas comunes.

Nota. Argüello M., & Guamán D., 2023

El estilo y el patrón arquitectónico imponen una transformación en el diseño de una arquitectura, pero el alcance del patrón es menor que el del estilo, ya que el patrón se concentra en un solo aspecto en lugar de toda la aplicación. Los patrones arquitectónicos abarcan aspectos de comportamiento dentro del contexto de la arquitectura. El uso en conjunto de los estilos y los patrones sirven para determinar la forma de la estructura general de un sistema, y es muy importante evaluar si un patrón es apropiado para la aplicación y el estilo arquitectónico en general.

Si bien los conceptos y la práctica a través de los estilos y patrones arquitectónicos es algo que todo ingeniero y arquitecto de software debería conocer, es importante también recordar y tener presente que toda arquitectura cumple con el criterio de evolucionar, cambiar, innovar haciendo uso para ello de componentes y conectores. Hablando del tema de evolución, a continuación, le invito a revisar algunas tendencias en lo que respecta a las arquitecturas de software con las cuales seguramente usted ya se está familiarizando.

4.7. Evolución de las arquitecturas, software y tendencias

La ingeniería de software en general ha reconocido que los sistemas de software requieren cambios y mejoras continuas para satisfacer los nuevos requisitos dados por los clientes u organizaciones que permitan su evolución. Según Michel y Daniel (2008), la evolución genera una degradación en la estructura del software, esto suele ocurrir debido a cambios en los requisitos que van desde los técnicos por el cambio de las plataformas tecnológicas, el mantenimiento excesivo, la erosión de la arquitectura, documentación insuficiente o inconsistente, funcionalidades duplicadas (código duplicado), falta de modularidad o cambios que provienen del negocio debido a la volatilidad inherente al contexto del sistema.

A quién de nosotros no le ha pasado que solo conocía de algoritmos o aplicaciones de escritorio que siguen un estilo como cliente – servidor, programación orientada, objetos o que funcionan en un solo equipo. En la actualidad seguramente usted ya está familiarizado con aplicaciones web, móviles, servicios, microservicios, Funciones como Servicios (FaaS) donde ya casi todo se implementa en *cloud* haciendo uso de modelos de servicios como *IaaS* (*Infraestructura as a Service*), *PaaS* (*Platform as a Service*), *SaaS*

(Software as a Service). El cambio y evolución es inevitable, por lo tanto, para que el software siga siendo útil, los cambios deben aplicarse cuando:

1. Surgen nuevos requisitos mientras el sistema está en uso o cuando está en proceso de desarrollo.
2. Hay nuevos requisitos dados por la empresa donde si hay errores en los sistemas deben ser reparados.
3. Se implementa nueva infraestructura de *hardware / software* para la operación o interacción de los componentes del sistema.
4. Se requiere alcanzar y mejorar los objetivos estratégicos de las organizaciones o empresas teniendo en cuenta los atributos de calidad.

La evolución de la arquitectura del *software* debería permitir cambios en el *software* de forma controlada sin comprometer la integridad del sistema y las invariantes. Sin embargo, cuando tiene lugar una evolución, los sistemas de *software* experimentan dos tipos principales de evolución (i) evolución interna y (ii) evolución externa, veamos de qué se trata esto.

1. **Evolución interna:** cambios de modelo en la topología de los componentes e interacciones, a medida que se crean o destruyen durante la ejecución. Este tipo de evolución implica la captura, la dinámica de métricas del sistema.
2. **Evolución externa:** cambios de modelo en la especificación de componentes y las interacciones requeridas para cumplir con los nuevos requisitos de los grupos de interés. Implica la adaptación de la arquitectura del *software*.

Dado que la evolución afecta a la arquitectura, los cambios en el proceso de evolución también deben reflejarse en la especificación (documentación) de la arquitectura del *software*. Los elementos que constituyen la arquitectura son críticos para soportar el proceso de ingeniería inversa e ingeniería directa sobre un sistema de *software* y, por extensión, su evolución, según Brown (2018).

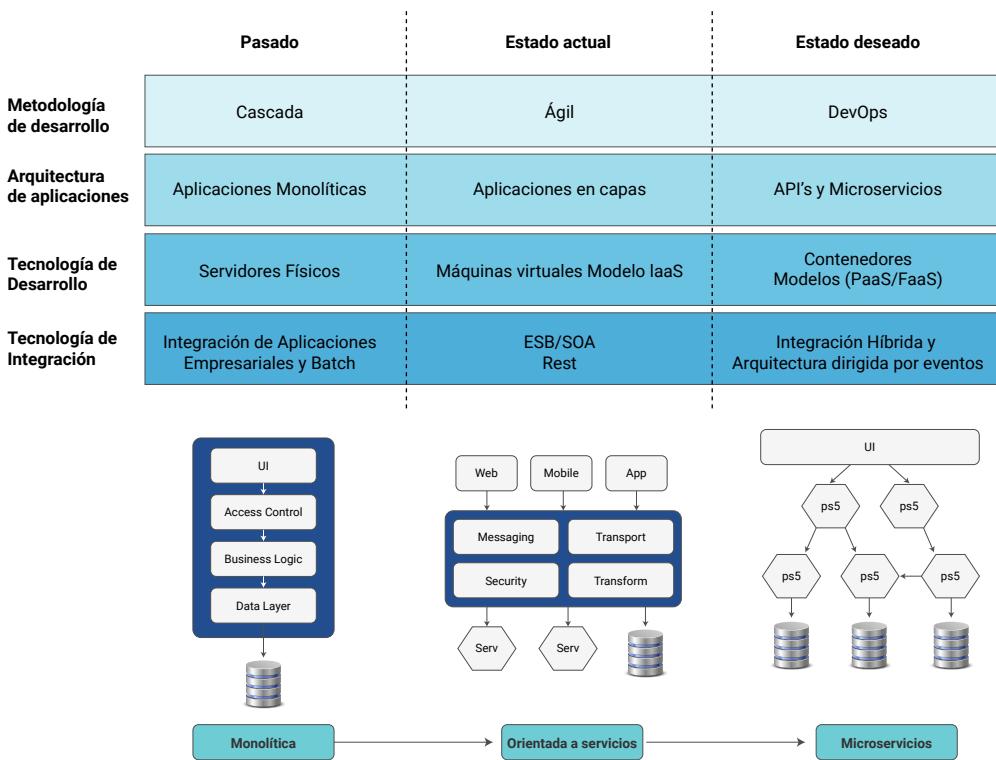
La arquitectura del *software* está en constante evolución a través del tiempo y resulta necesario su análisis, comprensión, reparación y mejora con el objetivo de estudiar la aplicación de patrones, atributos de calidad y

buenas prácticas de implementación donde se incluya nuevas tecnologías a nivel de *hardware* y *software* en nuevas soluciones de *software*.

En la figura 38 se visualiza como el *software* y las arquitecturas han evolucionado, pasando de lo que conocíamos como aplicaciones monolíticas hacia lo que se conoce como descomponer a este tipo de aplicaciones en pequeños servicios, microservicios y funciones que como criterio de implementación hacen uso de modelos o infraestructuras *cloud*. Lo que hace diferente a cada arquitectura es que por ejemplo en una aplicación monolítica se hace uso de un servidor generalmente físico (despliegue local) sobre el cual se distribuye la aplicación y la base de datos. Cuando hablamos de servicios pensemos en que descomponemos algunas capacidades de negocio del sistema para desplegarlos en servidores físicos, despliegue local o *cloud*. En este tipo de arquitectura, al igual que en monolítica, los datos se extraen desde un repositorio que está centralizado. Cuando se habla de microservicios, a los servicios se descomponen en componentes más pequeños y ya se incluye infraestructura como contenedores que por lo general se alojan en *cloud*. Otra de las características de los microservicios es que cada uno puede acceder a recuperar datos desde su propio repositorio alojado en *cloud*. El tema de funcionalidad, se puede tener capacidades muy específicas que pueden interoperar en *cloud* con otros sistemas, otros servicios u otras funciones; uno de los conceptos más conocidos es FaaS (*Function as a Service*).

Figura 32.

Evolución de las arquitecturas software a nivel de aplicación/concepto, infraestructura e implementación



Nota. Argüello M., & Guamán D., 2023

Para complementar y reforzar los temas relacionados con arquitectura monolítica, servicios, microservicios y FaaS y como estos se asocian con estilos y patrones arquitectónicos, le invito a revisar bibliografía complementaria disponible de forma digital o en la web. Espero que hasta el momento se vaya comprendiendo la temática de cada semana, por favor no olvide que en caso de tener dudas contactar con su profesor tutor a través de los diferentes medios: plataforma virtual, correo electrónico.

Lecturas en bibliografía básica, complementaria y recursos educativos abiertos.

- Revise la bibliografía complementaria que le permitirá conocer los conceptos, tipología, ventajas, desventajas y uso de los patrones arquitectónicos más comunes.

- **Libro de Somerville** – (Capítulo 6 – Diseño arquitectónico. Subtemas, 6.3 Patrones arquitectónicos, 6.3.1 Arquitectura en capas, 6.3.2 Arquitectura de repositorio, 6.3.3 Arquitectura cliente-servidor, 6.3.4 Arquitectura de tubería y filtro, 6.4 Arquitecturas de aplicación, 6.4.1 Sistemas de procesamiento de transacciones, 6.4.2 Sistemas de información, 6.4.3 Sistemas de procesamiento de lenguaje).
 - **Libro de Presman** – (Capítulo 9 – Diseño de la arquitectura. Subtemas 9.3.2 Patrones arquitectónicos).
 - **Libro Arquitectura de Software, conceptos y ciclo de desarrollo** – (Capítulo 3 – Toma de decisiones para crear estructuras – subtemas 3.4.1 Patrones).
 - **DD3_Arquitecturas de Software** - Patrón Arquitectónico (Páginas 21, 22, 23).
 - **DD3_Arquitecturas de Software** - Estilo Arquitectónico (Páginas 19,20).
- Revise las lecturas recomendadas y apóyese en bibliografía complementaria para diferenciar entre estilos y patrones arquitectónicos.
- **DD1_Arquitecturas software** - Patrones arquitectónicos (Páginas 17-53).

Lectura recomendada en la web: Estilos vs. Patrones Arquitectónicos.

- [Architectural Styles vs. Architectural Patterns vs. Design Patterns](#)

Lecturas recomendadas en la web: Evolución de las arquitecturas de Software.

- [Arquitecturas monolíticas vs. microservicios.](#)
- [Conversión de aplicaciones monolíticas en microservicios mediante el diseño basado en dominios.](#)
- [De una arquitectura tradicional a microservicios.](#)



Actividades de aprendizaje recomendadas

Ejercicio:

Busque en cualquier repositorio como GitHub, GitLab, Bitbucket una aplicación web o móvil que haga uso de un patrón arquitectónico y que esté construida en Java, PHP, o Visual Studio. Una vez identificada la aplicación, póngala en funcionamiento en su equipo local o en la nube y posterior a ello, represente de forma gráfica los componentes o elementos que son parte del patrón seleccionado.

Una vez que ha estudiado los conceptos relacionados a la unidad, le invito a desarrollar la autoevaluación con el fin de evaluar los conocimientos adquiridos hasta el momento.



Autoevaluación 4

1. ¿A qué estilo arquitectónico corresponde la siguiente afirmación? Distribuye el sistema generalmente en dos nodos, donde el cliente realiza solicitudes al servidor. En muchos casos, el servidor es una base de datos con lógica de aplicación representada como procedimientos almacenados.
 - a. Cliente – servidor.
 - b. Orientada a objetos.
 - c. Capas.
 - d. Publicador – suscriptor.

2. ¿A qué estilo arquitectónico corresponde la siguiente afirmación? Un paradigma de diseño basado en la división de responsabilidades para una aplicación o sistema en objetos individuales reutilizables y autosuficientes, cada uno de los cuales contiene los datos y el comportamiento relevante para el objeto.
 - a. Cliente – servidor.
 - b. Orientada a objetos.
 - c. Capas.
 - d. Publicador – suscriptor.

3. ¿A qué estilo arquitectónico corresponde la siguiente afirmación? Segrega la funcionalidad en segmentos separados, donde cada segmento es un nivel ubicado en una computadora físicamente separada.
 - a. Cliente – servidor.
 - b. Orientada a objetos.
 - c. Capas.
 - d. Publicador – suscriptor.

4. ¿A qué estilo o patrón arquitectónico corresponde la siguiente definición? Se refiere a aplicaciones que exponen y consumen funcionalidad como un servicio mediante contratos y mensajes.
- Broker*.
 - Cliente – servidor.
 - Arquitectura Orientada a Servicios (SOA).
 - Layers*.
5. El estilo arquitectónico en capas (*tiers*) es utilizado dentro de la categoría de:
- Comunicación.
 - Despliegue.
 - Dominio.
 - Estructura.
6. El estilo de arquitectura que prescribe el uso de un sistema de software que puede recibir y enviar mensajes utilizando uno o más canales de comunicación, de modo que las aplicaciones puedan interactuar sin necesidad de conocer detalles específicos entre sí es el que hace uso de:
- Capas.
 - Bus de Servicios Empresariales (ESB).
 - Orientación a objetos.
 - Cliente y servidor.
7. Si está creando una aplicación que va a tener servidor centralizado en un equipo local, al cual deben acceder los clientes dentro de una *intranet*, es posible que tenga uno o varios clientes que envíen solicitudes dicho servidor. En este caso, ¿qué tipo de arquitectura está implementando?:
- Orientación a servicios.
 - Ciente – servidor.
 - Microservicios.
 - Model – Vista – Controlador.

8. ¿Cuáles son algunos de los principios comunes para los diseños que utilizan el estilo arquitectónico en capas?
- a. Abstracción.
 - b. Encapsulación.
 - c. Alta cohesión.
 - d. Bajo acoplamiento.
9. Cuando usa MVC, ¿qué componente trabaja directamente con la base de datos para buscar, insertar, actualizar y eliminar datos?
- a. Modelo.
 - b. Vista.
 - c. Controlador.
 - d. Interface.
10. Cuando usa MVC, ¿en qué componente se procesan los datos después de recibir una solicitud de View y antes de actualizar cualquier cosa en nuestra base de datos con nuestro modelo?
- a. Modelo.
 - b. Vista.
 - c. Controlador.
 - d. Interface.

[Ir a solucionario](#)

Resultado de aprendizaje 3

- Elabora diseños arquitectónicos para diferentes tipos de aplicaciones en un dominio específico de negocio digital.

Para lograr este resultado de aprendizaje se analizarán los modelos de arquitectura más comunes y se describirán técnicas para el diseño, implementación y documentación de arquitecturas de software.

Hasta el momento usted debe conocer que la arquitectura de software provee conceptos y técnicas que permiten la definición, diseño, desarrollo y validación de sistemas medianos, grandes y complejos. Adicional a ello también debe conocer que a través de la arquitectura se describe la estructura (componentes y conectores) de un sistema de software, ocultando los detalles de bajo nivel y abstrayendo las características importantes de alto nivel, utilizando para ellos modelos para descripción arquitectónica. Le invito a revisar el [anexo 2](#) donde a partir del estudio de caso descrito en el [anexo 1](#), se propone una solución haciendo uso del modelo 4+1 donde se explican las vistas y se realizan los diagramas principales utilizando UML.

Contenidos, recursos y actividades de aprendizaje



Semana 11

Unidad 5. Modelos de arquitectura

Para hablar de modelos de arquitectura es importante conocer lo que es una vista, los tipos, los diagramas y la forma en que ellos pueden ser agrupados usando nomenclatura formal o semiformal, es necesario recordar y conocer los modelos que podemos usarlo para describir una arquitectura.

5.1. Modelos para descripción arquitectónica

Los modelos se crearon originalmente para admitir arquitecturas de sistemas empresariales. Estos pueden ser sistemas organizativos de sistemas, o pueden tener una estructura de gestión más simple para que la cartera de sistemas se pueda gestionar como un todo. Entre los modelos que nos ayudan a describir y documentar una arquitectura de software constan: (i) *4+1 View* en el cual se agrupan y distribuyen las vistas con el objetivo de utilizarlas acorde al tipo de sistema que queremos describir y (ii) el *Modelo C4* que es una técnica de notación gráfica ajustada para modelar la arquitectura de sistemas de software la cual se basa en una descomposición estructural de un sistema en contenedores y componentes y hace uso de técnicas de modelado existentes, como el Lenguaje de Modelado Unificado (UML) o los diagramas Entidad Relación (E-R) para la descomposición más detallada de los bloques de construcción arquitectónicos. Le invito a que iniciemos el estudio del modelo *4+1* y que recordemos algunos conceptos que anteriormente ya los había revisado, pero son necesarios.

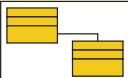
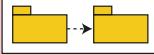
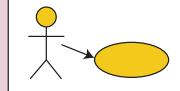
5.1.1. Modelo de arquitectura *4+1*

Si recordamos, la arquitectura del software se ocupa de la abstracción, la descomposición / composición, el estilo y la estética. Para describir una arquitectura de software, se puede hacer uso de un modelo compuesto por múltiples vistas o perspectivas con el objetivo de abordar eventualmente arquitecturas de sistemas medianos o grandes.

Según Kruchten (1995), el modelo *4+1* (ver figura 39) permite a través de un conjunto de vistas (*vista lógica*, *vista de desarrollo*, *vista de proceso*, *vista física* y *vista de escenarios*) representar de forma visual y documentada las características y modelos principales del software.

Figura 33.

Vistas del modelo 4+1

	Conceptual / Logical	Physical / Operational
Functional	<p>Logical/Structural view</p> <p>Perspective: Analysts, Designers Stage: Requirement analysis Focus: Object-oriented decomposition Concerns: Functionality Artefacts:</p> <ul style="list-style-type: none"> Class diagram Object diagram Composite structure diagram 	<p>Implementation/Developer view</p> <p>Perspective: Developers, Prof. mngs. Stage: Design Focus: Subsystem decomposition Concerns: software management Artefacts:</p> <ul style="list-style-type: none"> Component diagram Package diagram 
	<p>Use Case/Scenario view</p> <p>Perspective: End users Stage: Putting it altogether Concerns: Understandability, usability Focus: Feature decomposition Artefacts:</p> <ul style="list-style-type: none"> Use-case diagram User stories 	
Non-Functional	<p>Process/Behaviour view</p> <p>Perspective: System Integrators Stage: Design Focus: Process decomposition Concerns: Performance, scalability, throughput Artefacts:</p> <ul style="list-style-type: none"> Sequence diagram Communication diagram Activity diagram State (machine) diagram Interaction overview diagram Timing diagram 	<p>Deployment/Physical view</p> <p>Perspective: System Engineers Stage: Design Focus: Map software to hardware Concerns: System topology, delivery, installation, communication Artefacts:</p> <ul style="list-style-type: none"> Deployment diagram Network topology (not UML)

Nota. Adaptado de Documenting software architecture - <https://bit.ly/2LaXyZ0>

UML se puede describir como un lenguaje de modelado visual de propósito general para visualizar, especificar, construir y documentar un sistema de software. UML como lenguaje de modelado se utiliza de manera independiente al proceso de desarrollo que se utilice, sin embargo, y dependiendo del contexto del producto software que se requiera construir se sugiere utilizarlo en procesos como Proceso Racional Unificado (*RUP, Rational Unified Process por sus siglas en inglés*), centrado en la arquitectura, iterativo e incremental, proceso ágil.

De igual forma, en nuestro curso le invito a revisar los procesos iterativo e incremental para, siguiendo las fases del ciclo de desarrollo de software, enfocarnos en las primeras fases. ¿Recuerda las fases del ciclo de

desarrollo de software, las metodologías y modelos/procesos que se pueden usar para construir software?, si no lo recuerda, es el momento de tomar la bibliografía básica y complementaria y reforzar estos temas.

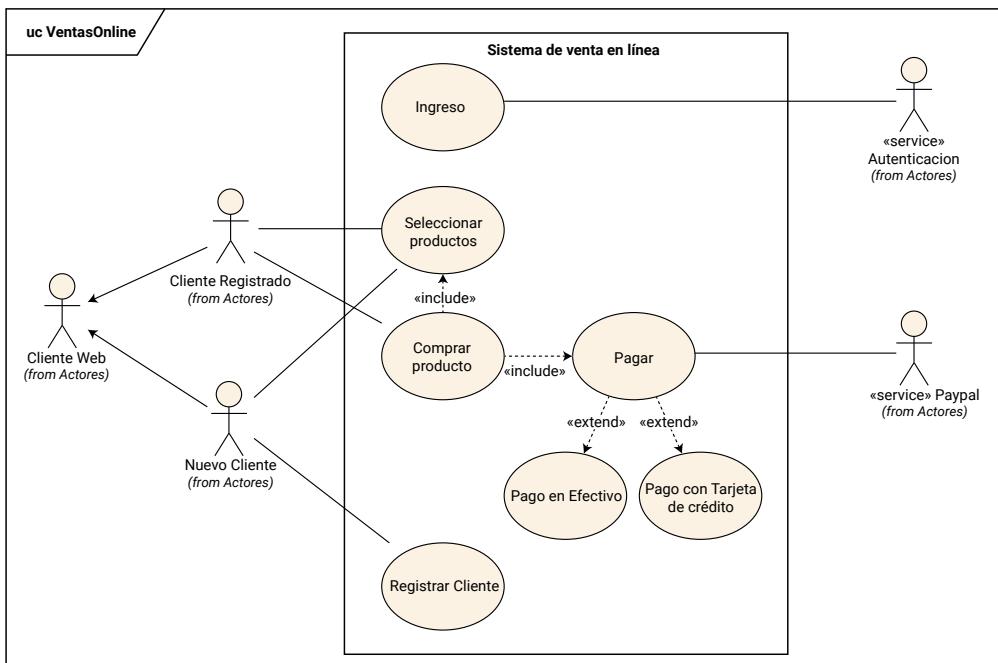
Una vez que ha revisado y recordado los temas relacionados con las metodologías ágiles y los procesos iterativo e incremental, nomenclatura UML acorde a los diagramas, aspectos básicos e introductorios de arquitectura, software y las fases del ciclo de vida de desarrollo del software revisemos como Krutchen agrupa los diagramas en vistas.

Krutchen en su modelo o marco de referencia propone 5 vistas, las mismas que se exponen a continuación:

5.1.1.1. Vista de escenarios o casos de uso

Los escenarios son en cierto sentido una abstracción de los requisitos más importantes. Su diseño se expresa mediante diagramas de escenarios de objetos y diagramas de interacción de objetos. Esta vista utiliza un pequeño subconjunto de escenarios importantes (por ejemplo, casos de uso) para mostrar que los elementos de los cuatro puntos de vista funcionan juntos sin problemas. Este punto de vista es redundante con los otros (de ahí el “+1”), pero juega dos roles críticos: (i) actúa como un conductor para ayudar a los diseñadores a descubrir elementos arquitectónicos durante el diseño de la arquitectura y (ii) valida e ilustra el diseño de la arquitectura, tanto en papel como punto de partida para las pruebas de un prototipo arquitectónico. Esta vista se apoya en el *diagrama de casos de uso*.

Figura 34.
Vista de escenarios - Diagrama de casos de uso



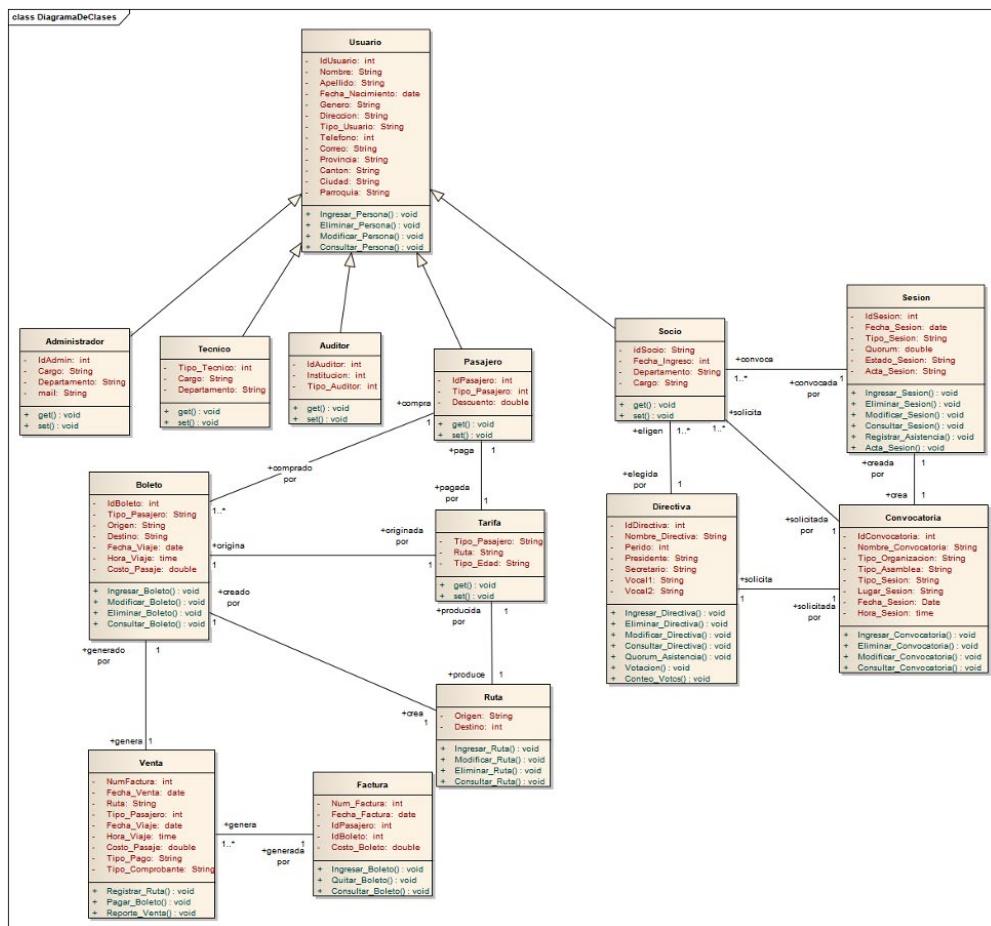
Nota. Argüello M., & Guamán D., 2023

5.1.1.2. Vista lógica o estructural

Para la construcción de esta vista se requiere principalmente entender y analizar los requisitos funcionales, es decir, lo que el sistema debe proporcionar en términos de servicios a usuarios finales. Esta vista hace uso del conjunto de abstracciones clave (por ejemplo, entidades de dominio, clases e interacciones entre ellas) resultantes de la descomposición del sistema y tomadas en su mayoría del dominio del problema. Esta vista, como es la que se usa para documentar las decisiones de diseño que permiten llevar a cabo la implementación o desarrollo, representa la descomposición del problema en objetos o clases de objetos y pone énfasis en los principios de abstracción, encapsulación y herencia. Esta descomposición no es solo por el bien del análisis funcional, sino que también sirve para identificar mecanismos comunes y elementos de diseño en las diversas partes del sistema. Uno de los principales diagramas en esta vista es el *diagrama de clases*, *diagrama de comunicación*.

Figura 35.

Vista lógica - Diagrama de clases



Nota. Arguello M., & Guamán D., 2023

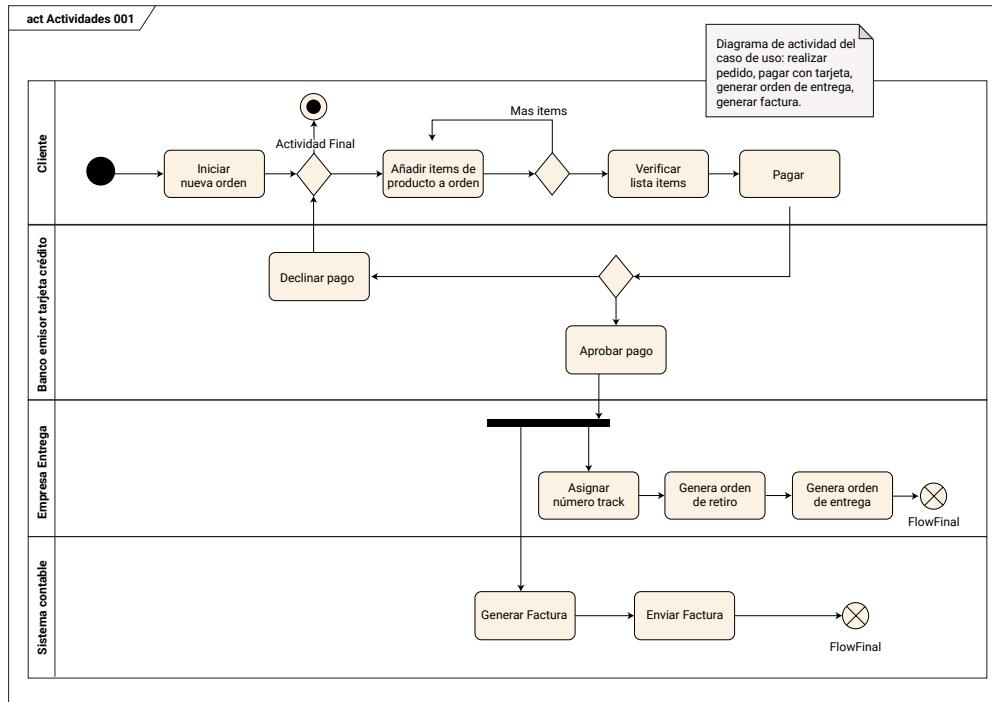
5.1.1.3. Vista de procesos o de comportamiento

Cuando se diseña y documenta una arquitectura de software es importante abordar aspectos concurrentes en tiempo de ejecución (tareas, subprocesos, procesos y sus interacciones). Esta vista tiene en cuenta algunos de los requisitos no funcionales tales como el rendimiento, la disponibilidad del sistema, la concurrencia y distribución, la integridad del sistema y la tolerancia a fallos. La vista de procesos se puede describir en varios niveles de abstracción, donde cada nivel aborda diferentes intereses o necesidades requeridos por el sistema de software. En el nivel más alto, la vista de procesos se puede ver como un conjunto de redes lógicas de ejecución independiente de programas de comunicación denominados

procesos, distribuidos a través de un conjunto de recursos de hardware conectados por una LAN o una WAN. Algunos de los diagramas que se pueden utilizar en esta vista son *diagrama de colaboración*, *diagrama de actividad*, *diagrama de estados* y *diagramas de secuencia*.

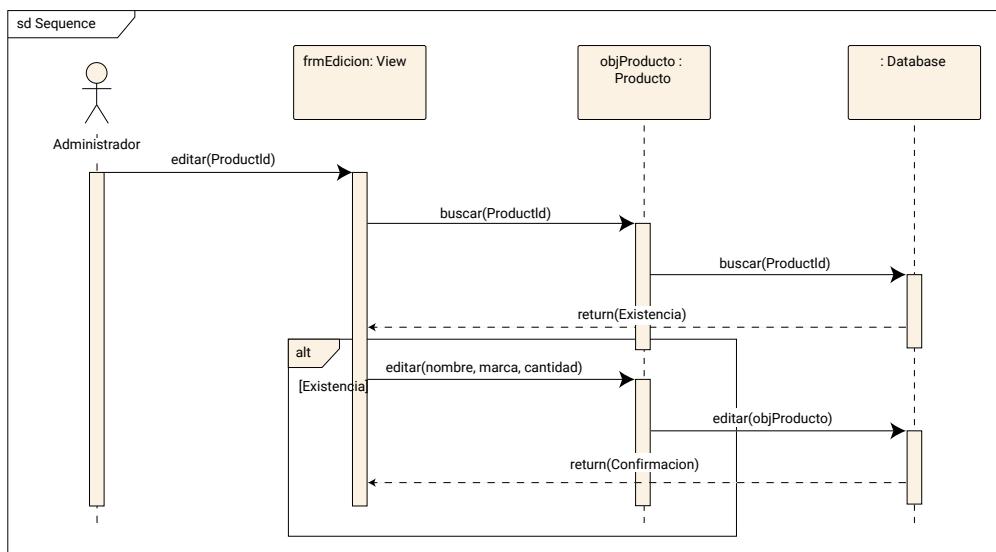
Figura 36.

Vista de procesos - Diagrama de actividades



Nota. Argüello M., & Guamán D., 2023

Figura 37.
Vista de procesos - Diagrama de secuencia



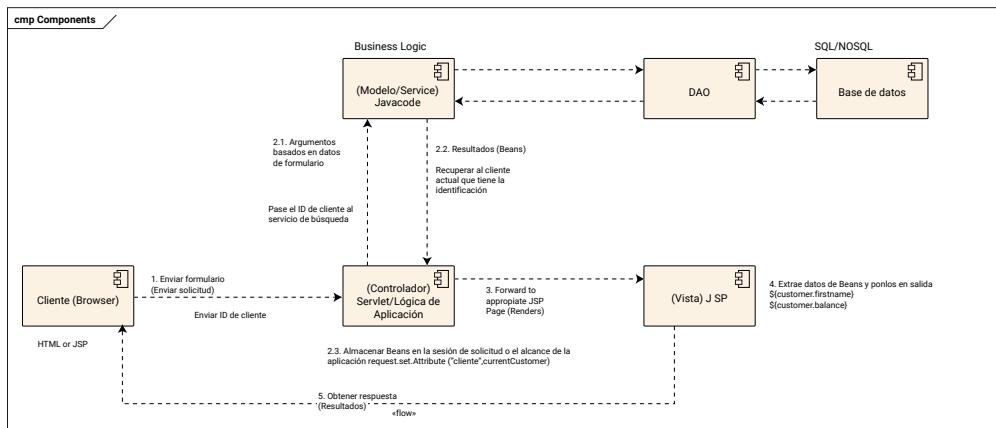
Nota. Argüello M., & Guamán D., 2023

5.1.1.4. Vista de implementación o de desarrollo

La vista de implementación se centra en la organización de los módulos o componentes del sistema en el entorno de desarrollo de software. Las vistas y los diagramas asociados permiten explicar el empaquetado del software en pequeños fragmentos (bibliotecas de programas o subsistemas) que llevados a la implementación pueden ser desarrollados por uno o más desarrolladores (equipo de desarrollo). Si recordamos los conceptos de estilos o patrones arquitectónicos, los módulos o componentes están organizados en una jerarquía de capas físicas o lógicas, donde cada capa tiene una responsabilidad definida y cada una de ella proporciona una interfaz bien definida para las capas superiores. La vista de desarrollo del sistema está representada por diagramas de módulos y subsistemas, que muestran las relaciones de “exportación” e “importación”. A través de esta vista se puede enumerar las reglas y decisiones de diseño que gobiernan la arquitectura de desarrollo entre las que constan la descomposición en pequeños módulos, el particionamiento, el agrupamiento, la visibilidad. La vista de desarrollo tiene en cuenta los requisitos internos relacionados con la facilidad de desarrollo, la gestión del software, la reutilización o los elementos comunes, y las limitaciones impuestas por el conjunto de herramientas o el lenguaje de programación.

La vista de desarrollo sirve como base para la asignación de requisitos, para la asignación del trabajo a los equipos (o incluso para la organización del equipo), para la evaluación y planificación de costos, para monitorear el progreso del proyecto, para razonar sobre la reutilización, portabilidad y seguridad del software. Es la base para establecer una línea de producto software. Esta vista se apoya en el *diagrama de componentes* y *diagrama de paquetes*.

Figura 38.
Vista de desarrollo - Diagrama de componentes



Nota. Argüello M., & Guamán D., 2023

5.1.1.5. Vista de despliegue o física

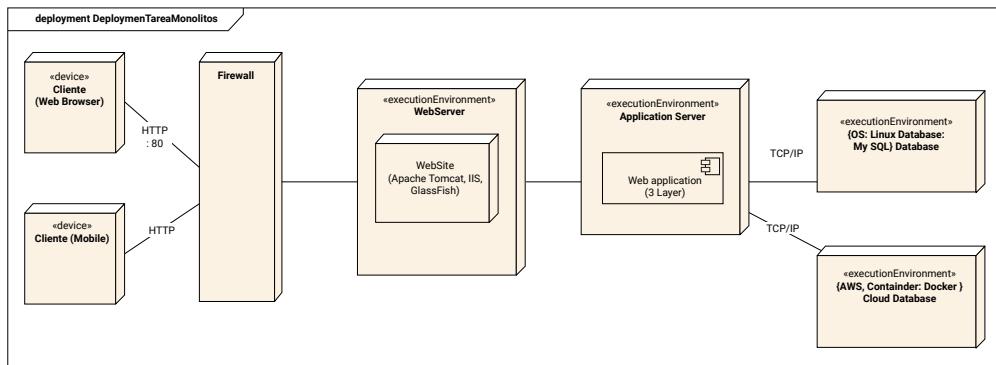
Aspecto importante en la documentación de una arquitectura de software es conocer dónde y cómo se va a desplegar nuestra solución de software, la misma que puede ser despliegue local, *cloud* o combinada. A través de esta vista se especifica que el sistema software se ejecuta en una red de *hardware* o nodos de procesamiento donde los diversos elementos identificados en los puntos de vista lógicos, de proceso y de implementación (redes, procesos, tareas y objetos) deben mapearse en los distintos nodos disponibles. Esta vista tiene en cuenta los requisitos funcionales y no funcionales del sistema tales como la disponibilidad del sistema, la confiabilidad (tolerancia a fallas), el rendimiento (rendimiento) y la escalabilidad, así como de algunos protocolos de transmisión de datos en *intranet* o *Internet* (TCP/IP, HTTP, FTP, UDP, etc.).

A través de esta vista se debe especificar las diversas configuraciones físicas que pueden existir para crear ambientes de desarrollo, pruebas,

preproducción, producción u otras que se requieran para la implementación acorde al tipo de sistema, ambiente de despliegue (despliegue local, *cloud*) y teniendo en cuenta los clientes o usuarios finales del sistema. Por lo tanto, la asignación del *software* a los nodos debe ser muy flexible y tener un impacto mínimo en el código fuente en sí. Esta vista se apoya en el *diagrama de despliegue*.

Figura 39.

Vista de Despliegue - Diagrama de despliegue



Nota. Argüello M., & Guamán D., 2023

En la Tabla 11 se expone el resumen de vistas, su descripción, partes interesadas, lo que se debe considerar y los diagramas UML asociados.

Tabla 11.*Resumen de vistas y diagramas*

	Vista Lógica	Vista de Procesos	Vista de Desarrollo	Vista Física	Vista de Escenarios
Descripción	Muestra el componente (objeto) del sistema, así como su interacción.	Muestra los procesos / reglas de flujo de trabajo del sistema y cómo se comunican esos procesos, se centra en la vista dinámica del sistema.	Proporciona vistas de bloques de construcción del sistema y describe la organización estática de los módulos del sistema.	Muestra la instalación, configuración e implementación de la aplicación de software.	Muestra que el diseño está completo al realizar la validación y la ilustración.
Fase	Análisis de requisitos	Diseño	Diseño	Diseño	Todas las fases
Enfoque	Descomposición orientada a objetos	Descomposición del proceso	Descomposición del subsistema	Mapeo del software a hardware	Descomposición de características
Partes interesadas	Usuario final, analistas y diseñadores	Integradores y desarrolladores de software	Programadores y gerentes de proyectos de software	Ingeniero de sistemas, operadores, administradores de sistemas e instaladores de sistemas	Todas las opiniones, puntos de vista de usuarios finales.
Considerar	Requisitos funcionales	Requisitos no funcionales	Organización del módulo de software (reutilización de la gestión de software, limitación de herramientas)	Requisito no funcional con respecto al hardware subyacente	Consistencia y validez del sistema
Diagrama UML	Clase, Estado, Objeto, Diagrama de comunicación	Diagrama de secuencia, actividades, máquina de estados	Diagrama de componentes y paquetes	Diagrama de despliegue, Diagrama de red	Diagrama de casos de uso, Historias de usuario

Nota. Argüello M., & Guamán D., 2023





Semana 12

Luego de revisar los elementos principales del modelo 4+1, es momento de conocer el modelo C4, el cual propone una notación gráfica ajustada para modelar la arquitectura de sistemas de software. Empecemos.

5.1.2. Modelo C4

Según lo que habíamos estudiado en semanas anteriores, la audiencia o personas que acorde a los puntos de vista requieran de documentación para entender la arquitectura de software, pueden apoyarse en vistas (como las propuestas en el modelo 4+1), las cuales hacen uso de algunos diagramas, vocabulario y nomenclatura UML.

Si hacemos una analogía entre los planos que un profesional de la industria de construcción utiliza para visualmente edificar una casa o edificio, en software a veces esto resulta complejo. Y es que cuando a un desarrollador de software se le solicita que comunique la arquitectura de un sistema de software mediante diagramas, lo más probable es que no los tenga, o si los tiene, estos sean confusas representaciones de cuadros y líneas o que la notación sea inconsistente (estereotipos, codificación de colores, formas, estilos de línea, etc.), use nombres ambiguos, relaciones no etiquetadas, terminología genérica, opciones de tecnología faltantes, abstracciones mixtas, etc.

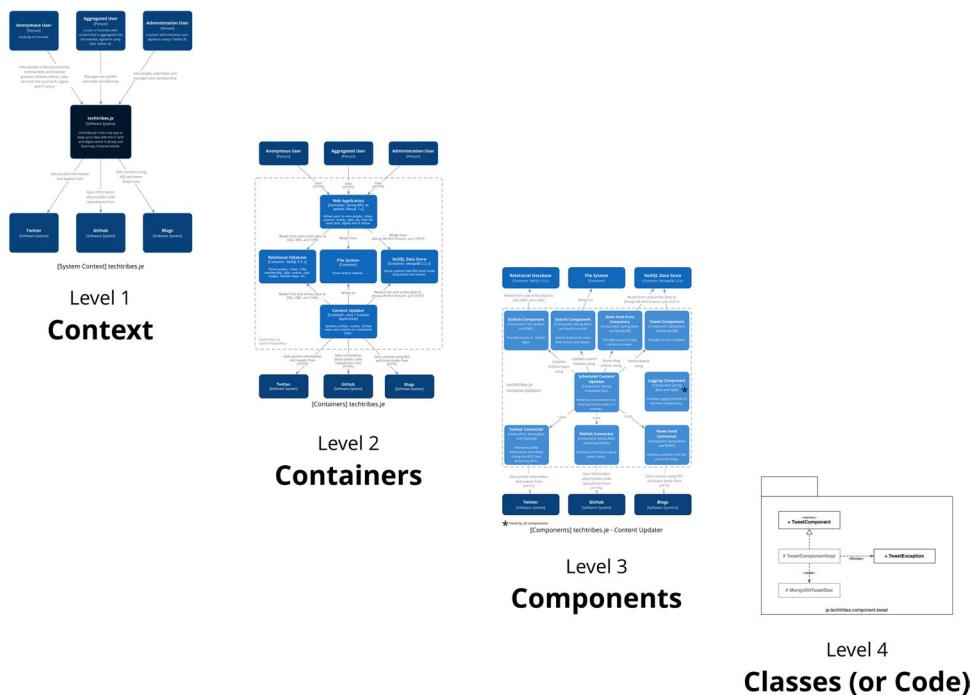
Para evitar estas inconsistencias podemos complementar la representación gráfica con algunos diagramas que a diferentes niveles de abstracción nos permiten visualizar la estructura estática de un sistema de software. Brown (2013), propone el modelo de arquitectura de software C4 (ver figura 46), en el cual se pone énfasis en representar el *Contexto*, *Contenedores*, *Componentes* y *Clases* (o Código, si desea tener una vista más genérica). A continuación, resumamos de que se trata cada una de las C que son parte del modelo:

1. **Contexto:** diagrama de alto nivel que establece la escena; incluidas las dependencias clave del sistema/servicio y las personas (actores / roles / personas / etc.).

2. **Contenedor:** diagrama que muestra las opciones de tecnología de alto nivel, cómo se distribuyen las responsabilidades entre ellas y cómo se comunican los contenedores.
3. **Componente:** para cada contenedor, un diagrama de componentes le permite visualizar los componentes lógicos clave y sus relaciones.
4. **Clases (o Código):** este es un nivel de detalle opcional que recoge los diagramas de clases UML de alto nivel, el cual sirve para explicar cómo se implementará (o se ha implementado) un componente en particular. Los diagramas UML que se realicen tienden a ser bocetos o diagramas borradores en lugar de modelos completos.

Figura 40.

Niveles del modelo C4



Nota. Adaptado de The C4 model for visualising software architecture - <https://c4model.com>

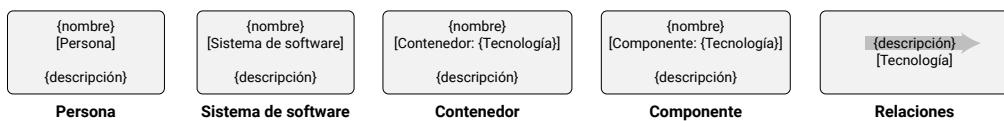
Brown (2013), menciona que el Modelo C4 se creó para ayudar a los equipos de desarrollo de software a describir y comunicar la arquitectura de software, especialmente durante las etapas tempranas de diseño y para

documentar retrospectivamente una base de código existente. Si hacemos una analogía, es una forma de crear mapas de código, en varios niveles de detalle, de la misma manera que usaría por ejemplo Google Maps para acercar y alejar un área o dirección que le interesa.

Respecto de la notación usada, el modelo C4 no prescribe ninguna notación en particular. Usa una notación semiformal simple que se puede representar a través del uso de pizarras, papel, notas adhesivas, fichas y una variedad de herramientas CASE de diagramación, dicha notación se expone en la figura 41.

Figura 41.

Notación usada en el modelo C4



Nota. Argüello M., & Guamán D., 2023

Aunque los diagramas del modelo C4 se crean utilizando una *notación de recuadros y líneas*, los diagramas centrales se pueden ilustrar utilizando UML, especialmente los relacionados con paquetes, componentes, nodos para el despliegue de la aplicación. Sin embargo, los diagramas UML resultantes tienden a carecer del mismo grado de texto descriptivo, porque agregar dicho texto no es posible (o fácil) con algunas herramientas CASE que usan UML.

Para crear los diagramas o modelos de código, primero necesitamos un conjunto común de abstracciones para a través de un lenguaje común describir la estructura estática de un sistema de software. A continuación, se exponen algunos de los conceptos usados en el modelo.

1. **Persona:** representa a uno de los usuarios que hacen uso o construye los sistemas de software (por ejemplo, actores, roles, personas, etc.).
2. **Sistema de software:** es el nivel más alto de abstracción y describe algo que ofrece valor a sus usuarios, sean humanos o no. Esto incluye el sistema de software que está modelando y los otros sistemas de software de los que depende su sistema de software (o viceversa). En muchos casos, un sistema de software es *propiedad de un solo equipo de desarrollo de software*.

3. **Contenedores:** cuando hablamos de contenedor, no se hace referencia a Docker o Google Cloud Platform. En el modelo C4, un contenedor representa una aplicación o un almacén de datos. Un contenedor es algo que debe ejecutarse para que funcione el sistema de software general, *similar a los ambientes de ejecución o nodos del diagrama de despliegue UML*. Se puede mencionar que un contenedor es esencialmente un contexto o límite dentro del cual se ejecuta algún código o se almacenan datos. Cada contenedor es un entorno de ejecución o algo implementable / ejecutable por separado, que normalmente (pero no siempre) se ejecuta en su propio espacio de proceso. En términos reales de software, un contenedor puede adoptar tipos como los que se exponen a continuación:
- 1.1. **Aplicación web del lado del servidor:** una aplicación web Java EE que se ejecuta en Apache Tomcat o GlassFish, una aplicación ASP.NET MVC que se ejecuta en Microsoft IIS (*Internet Information Server*), una aplicación Ruby on Rails que se ejecuta en WEBrick, una aplicación Node.js, etc
 - 1.2. **Aplicación web del lado del cliente:** una aplicación JavaScript que se ejecuta en un navegador web usando Angular, Backbone. JS, jQuery, etc
 - 1.3. **Aplicación cliente o de escritorio del lado del cliente:** una aplicación de escritorio de Windows escrita con WPF (Windows Presentation Foundation), una aplicación de escritorio de OS X escrita con Objective-C, una aplicación de escritorio multiplataforma escrita con JavaFX, Java Swing, etc
 - 1.4. **Aplicación móvil:** una aplicación de Apple iOS, una aplicación de Android, una aplicación de Microsoft Windows Phone, etc
 - 1.5. **Aplicación de consola del lado del servidor:** una aplicación independiente (por ejemplo, “*public static void main*”), un proceso por lotes, etc
 - 1.6. **Función sin servidor (Serverless):** una única función sin servidor (por ejemplo, Amazon, Lambda, función de Azure, etc.)
 - 1.7. **Base de datos:** un esquema o base de datos en un sistema de gestión de base de datos relacional o no relacional, almacén de documentos,

base de datos de gráficos, etc. Por ejemplo, MySQL, Microsoft SQL Server, Oracle Database, MongoDB, Riak, MariaDB, PostgreSQL, Cassandra, Neo4j, etc

- 1.8. **Blob o almacén de contenido:** un almacén *blobs* (por ejemplo, Amazon S3, Microsoft Azure Blob Storage, etc.) o una red de distribución de contenido (por ejemplo, Akamai, Amazon CloudFront, etc.)
 - 1.9. **Sistema de archivos:** un sistema de archivos local completo o una parte de un sistema de archivos en red más grande (por ejemplo, SAN (Storage Área Network), NAS (Network Attached Storage, etc.)
 - 1.10. **Script de shell:** un único *script* de *shell* escrito en Bash, etc
4. **Componente:** en el contexto del modelo C4, un componente es una agrupación de funcionalidades relacionadas encapsuladas detrás de una interfaz bien definida. Por ejemplo, cuando se utiliza un lenguaje como Java o C #, la forma más sencilla de pensar en un componente es que es una colección de clases de implementación detrás de una interface. Aspectos como la forma en que se empaquetan esos componentes (por ejemplo, un componente frente a muchos componentes por archivo JAR, DLL, biblioteca compartida, etc.) es una preocupación separada y ortogonal. Un punto importante a tener en cuenta es que todos los componentes dentro de un contenedor generalmente se ejecutan en el mismo espacio de proceso. En el modelo C4, los componentes no son unidades desplegables por separado.

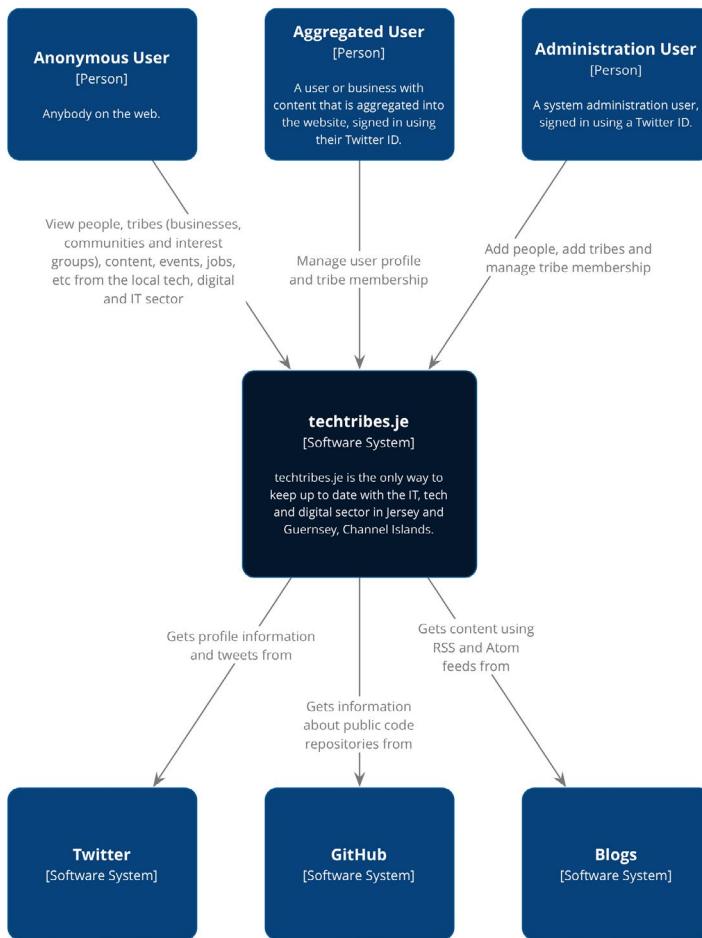
Muy bien, ¿lo tenemos claro? Recuerde apoyarse en bibliografía complementaria o consultar a su profesor tutor si tiene dudas, especialmente porque la tarea de la asignatura contempla el uso del modelo C4 para documentar su propuesta arquitectónica. Una vez conocido algunos conceptos que son parte del modelo, es momento de revisar cómo y qué representan los 4 diagramas o niveles principales del modelo C4.

5.1.2.1. Nivel 1: enlace del diagrama de contexto del sistema

Un diagrama de contexto del sistema es un buen punto de partida para diagramar y documentar un sistema de software, lo que le permite dar un paso atrás (requisitos) y ver el panorama general (ver figura 42). En su

representación gráfica, el contexto muestra el sistema como una caja en el centro, rodeado por sus usuarios y los otros sistemas o servicios con los que interactúa. Cabe recalcar que para llegar a representar el contexto debemos tener presente a las necesidades y requisitos de negocio, de usuario y de sistema.

Figura 42.
Nivel 1 del modelo C4 – Contexto



[System Context] techtribes.je

Nota. Adaptado de The C4 model for visualizing software architecture - <https://c4model.com>

Los detalles en este tipo de diagrama no son muy importantes, ya que esta se considera como una vista alejada o de nivel 0 que muestra una imagen global o general del panorama del sistema. El contexto debe incluir las

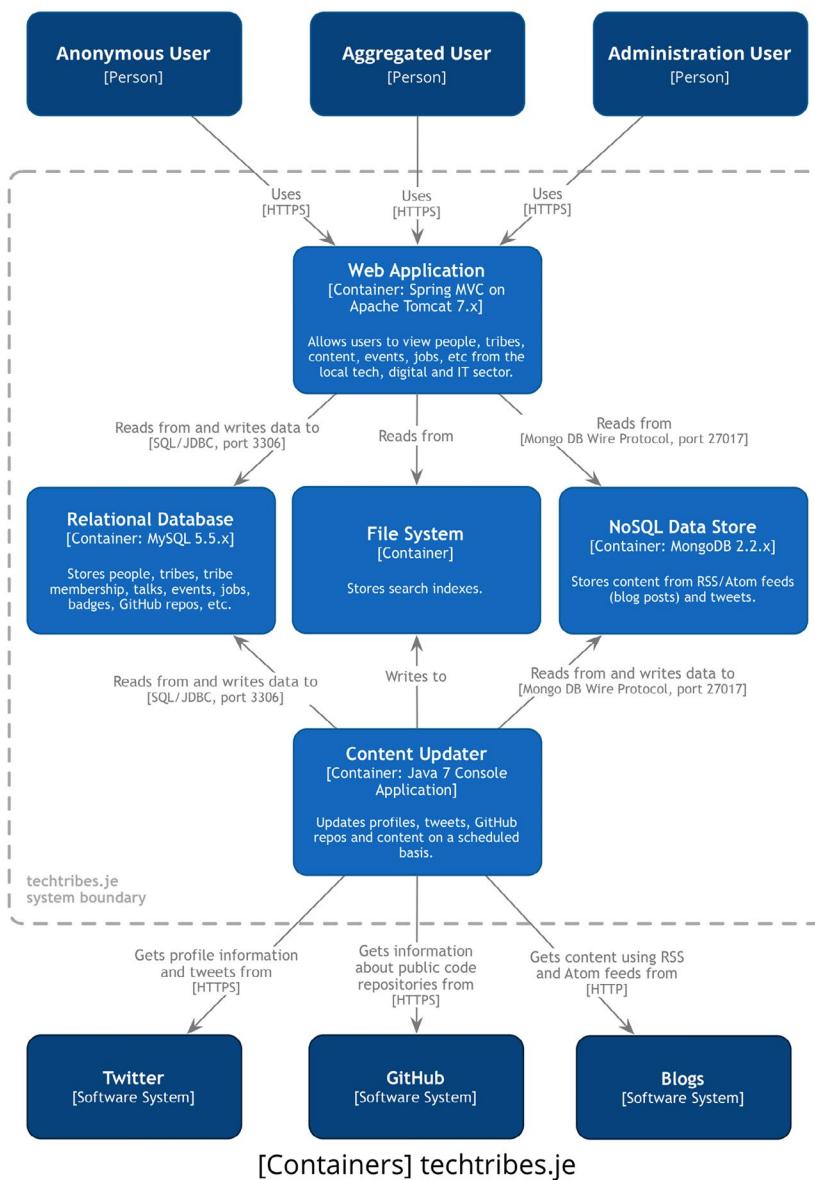
personas (actores, roles, personas, etc.), servicios y sistemas de *software* en lugar de tecnologías, protocolos y otros detalles de bajo nivel. A través de este diagrama podemos exponer nuestra posible solución a personas sin conocimientos técnicos. En resumen, este nivel expone lo siguiente:

- **Alcance:** un solo sistema de *software*.
- **Elementos primarios:** el sistema de *software* en el ámbito del contexto.
- **Elementos de apoyo:** personas (por ejemplo, usuarios, actores, roles o personas), servicios y sistemas de *software* (dependencias externas) que están directamente conectados al sistema de *software* dentro del alcance. Por lo general, cuando existen otros sistemas o servicios de *software*, estos se encuentran fuera del alcance o los límites de su propio sistema de *software*, es decir, no se tiene la responsabilidad ni la propiedad de ellos, sin embargo, habrá que representarlos gráficamente.
- **Público objetivo:** todas las partes interesadas, tanto técnicos como no técnicos, dentro y fuera del equipo de desarrollo de *software*.

5.1.2.2. Nivel 2: enlace de diagrama de contenedor

Una vez que conocemos cómo encaja el sistema en el entorno general de TI (Tecnología de la Información), uno de los siguientes pasos es acercarse al límite del sistema, utilizando para ello un diagrama de contenedor. Un contenedor es algo como una aplicación web del lado del servidor, una aplicación que contiene varias páginas (.html, .php, .cs), una aplicación de escritorio, una aplicación móvil, un esquema de base de datos, un sistema de archivos, etc. Esencialmente, un contenedor es una unidad ejecutable / implementable por separado (por ejemplo, un espacio de proceso separado) que ejecuta código o almacena datos (ver figura 43).

Figura 43.
Nivel 2 del modelo C4 – Contenedor



Nota. Adaptado de The C4 model for visualising software architecture - <https://c4model.com>

El diagrama de contenedor muestra en alto nivel la arquitectura del software y cómo se distribuyen las responsabilidades en ella. También muestra las principales opciones tecnológicas y cómo los contenedores se comunican entre sí. Es un diagrama simple, centrado en la tecnología

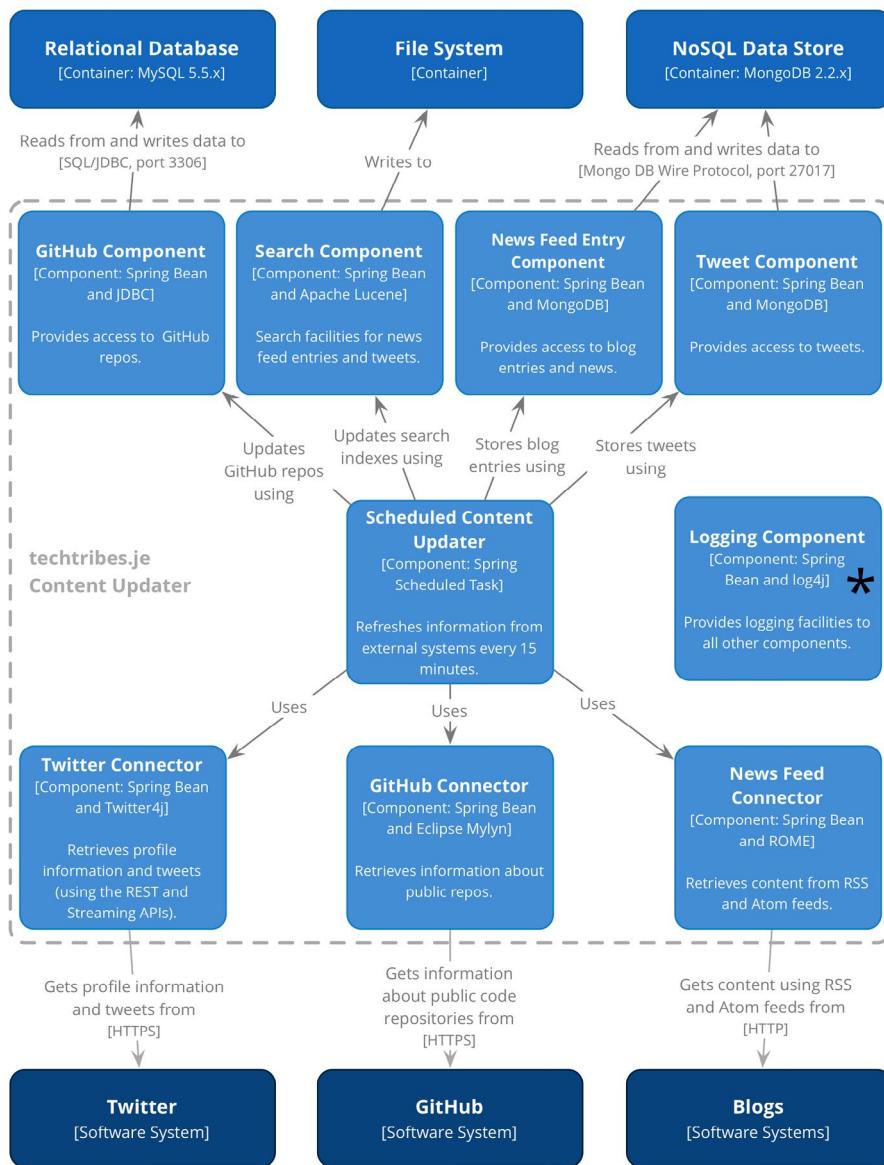
de alto nivel, que es útil tanto para los desarrolladores de *software* como para el personal de operaciones y soporte. En resumen, este nivel expone lo siguiente:

- **Alcance:** un solo sistema de *software*.
- **Elementos primarios:** contenedores dentro del alcance y contexto del sistema de *software*.
- **Elementos de apoyo:** personas, sistemas o servicios de *software* directamente conectados a los contenedores.
- **Público objetivo:** personal técnico dentro y fuera del equipo de desarrollo de *software*, incluidos arquitectos de *software*, desarrolladores y personal de operaciones / soporte.
- **Nota adicionales:** este diagrama no expone detalles sobre escenarios de implementación, agrupamiento, replicación, comutación por error, etc.

5.1.2.3. Nivel 3: enlace del diagrama de componentes

Una vez definido el Nivel 2, puede acercar y descomponer cada contenedor con el objetivo de identificar los principales bloques de construcción estructural del sistema y sus interacciones (ver figura 44).

Figura 44.
Nivel 3 del modelo C4 – Componentes



* Usado por todos los componentes

[Components] techtribes.je - Content Updater

Nota. Adaptado de The C4 model for visualising software architecture - <https://c4model.com>

El diagrama de componentes, que corresponde al Nivel 3, muestra cómo un contenedor se compone de una serie de componentes, expone cuáles son

cada uno de esos componentes, sus responsabilidades y los detalles de tecnología / implementación. En resumen, este tipo de diagramas expone lo siguiente:

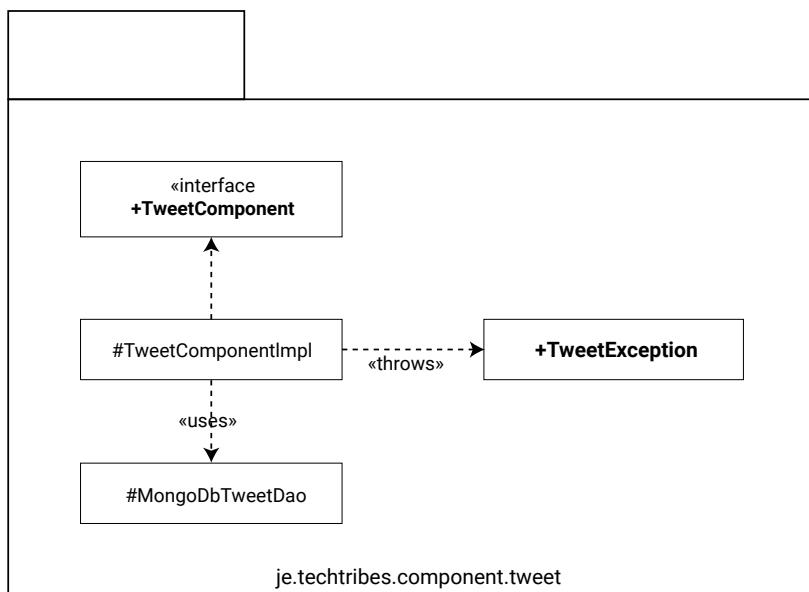
1. **Alcance:** un solo contenedor.
 - **Elementos primarios:** componentes dentro del alcance y contexto del sistema de software.
2. **Elementos de apoyo:** contenedores (dentro del sistema de software en el alcance y contexto), personas y sistemas de software que interactúan directamente conectados a los componentes.
3. **Público objetivo:** arquitectos y desarrolladores de software.

5.1.2.4. Nivel 4: enlace de código

Por último, y conociendo como cada componente se despliega en un contenedor, se puede ampliar o mirar a detalle para mostrar cómo se implementa como código usando lenguajes y tecnologías de programación. Para ello, en este nivel se utiliza diagramas de clases UML, diagrama de paquetes que agrupen las clases, diagramas de relación de entidades o similares (ver figura 45).

Figura 45.

Nivel 4 del modelo C4 – Código o Clases



Fuente: The C4 model for visualising software architecture - <https://c4model.com>

Este es un nivel de detalle opcional y, a menudo, se lo realiza para implementarlo en herramientas o IDE de desarrollo. Idealmente, este diagrama se generaría automáticamente utilizando herramientas (por ejemplo, una herramienta de modelado IDE, StarUML, Enterprise Architect, Rational Rose, o la que tenga asociado UML), y debería considerar mostrar solo aquellos atributos y métodos que le permitan exponer el contexto dentro del dominio del problema. Este nivel de detalle no es aconsejable para representar aquellos componentes más importantes o complejos. En resumen, este tipo de diagramas expone lo siguiente:

1. **Alcance:** un solo componente.
2. **Elementos primarios:** elementos de código (por ejemplo, clases, interfaces, objetos, funciones, tablas de bases de datos, etc.) dentro del componente acorde al alcance y contexto.
3. **Público objetivo:** arquitectos y desarrolladores de software.

En resumen, a través de los modelos podemos describir, documentar y visualizar una arquitectura de software utilizando para ello notación y nomenclatura UML, cuadros y líneas como notación semiformal que facilite

la comprensión y entendimiento a las partes interesadas para posterior a ello llevar a cabo la codificación e implementación de la solución.

Antes de finalizar el estudio durante estas semanas, le hago una pregunta ¿cuándo usar diagramas o modelado?, ¿son conceptos iguales o diferentes?, le invito a revisar el siguiente apartado donde se brinda una respuesta a la pregunta y como siempre por favor complemente con lectura en bibliografía disponible de forma digital o en la web para tener claro estos dos conceptos.

Diagramas vs. Modelado

A nivel académico y de la industria, los diagramas y los modelos se consideran importantes dependiendo de la calidad o densidad de la documentación arquitectónica que se requiera. Cuando diagramamos, normalmente creamos uno o más diagramas separados (que posteriormente acorde a un modelo podríamos agruparlos por vistas), a menudo con una notación *ad hoc*, utilizando por ejemplo herramientas CASE, despliegue local o *cloud*, papel y lápiz o una pizarra, es decir podemos tener mucha o poca semántica en los diagramas.

El lenguaje de dominio de las herramientas de diagramación es realmente solo cuadros y líneas, por lo que a veces resulta complejo conocer, por ejemplo, ¿qué dependencias tiene el componente X? Además, la reutilización de los elementos en todos los diagramas generalmente se realiza mediante duplicación (es decir, copiando y pegando, especialmente cuando se usa herramientas CASE), lo que le otorga la responsabilidad de mantener los diagramas sincronizados cuando cambia el nombre de dichos elementos. Es importante señalar aquí que, como hemos expuesto anteriormente, el modelo C4 se puede usarse independientemente de si está haciendo diagramas o modelando, pero hay algunas oportunidades interesantes cuando pasa de la diagramación al modelado.

Con el modelado, está construyendo un modelo no visual de algo (por ejemplo, la arquitectura de un sistema de software) y luego se crea diferentes vistas (que agrupan, por ejemplo diagramas) sobre ese modelo. Estos diagramas requieren un poco más de rigor, pero el resultado es una única definición de todos los elementos y las relaciones entre ellos. Esto, a su vez, permite que las herramientas CASE de modelado comprendan la semántica de lo que está tratando de hacer y proporcionen información adicional sobre el modelo (permitiendo realizar ingeniería directa e

ingeniería inversa). También permite que las herramientas de modelado proporcionen visualizaciones alternativas, a menudo de forma automática.

Una de las preguntas más frecuentes acerca del uso de los diagramas en sistemas de software grandes y complejos se refiere, por ejemplo, cuando comienza a tener más de 20 elementos (más las relaciones entre ellos) en un diagrama, el diagrama resultante visualmente comienza a desordenarse muy rápidamente, ¿verdad? Un enfoque para lidiar con esto es no mostrar todos los elementos o componentes en un solo diagrama, sino crear varios diagramas (recuerde el concepto, te divide y vencerás), uno por corte o que se pueda agrupar a través de paquetes, componentes, contenedores o clases. Este enfoque ciertamente puede ayudar, pero vale la pena preguntarse si, ¿los diagramas resultantes son útiles?, ¿los va a usar y, de ser así, para qué los va a usar? Aunque en el modelo C4 los diagramas de Contexto y Contenedor del sistema son muy útiles, los diagramas de Componentes para sistemas de software grandes a menudo tienen menos valor porque son más difíciles de mantener actualizados, y es posible que muy poca gente los vea de todos modos, especialmente si no están incluidos en documentación o presentaciones visuales.

Con esta comparativa breve hemos finalizado el estudio de los temas referentes a modelos, los cuales se usan para documentar y visualizar una arquitectura de software, como siempre le invito a complementar los temas realizando las actividades de aprendizaje recomendadas y a revisar bibliografía complementaria disponible en formato digital o en la web, a través de los cuales podemos reforzar algunos conceptos. Nos vemos en la siguiente semana.

Lecturas en bibliografía básica, complementaria y recursos educativos abiertos.

- Revise la bibliografía complementaria para conocer los modelos y las vistas o niveles que se utilizan para describir, documentar y comunicar una arquitectura de software.
 - **Libro Arquitectura de Software, conceptos y ciclo de desarrollo**
– (Capítulo 4 – Documentación: Comunicar la Arquitectura – subtemas 4.6.1 Vistas y más allá, 4.6.2 4+1 Vistas, 4.6.3 Puntos de vista y perspectivas).
 - **DD1_Arquitecturas software - 4+1 Vistas** (Páginas 5-16).

Lecturas recomendadas en la web: El modelo C4 de documentación para la Arquitectura de Software.

- URL: <https://bit.ly/3pQoV9R>

OCW1_OpenCourseWare.

- Arquitectura Física - Diagrama de Componentes (Páginas 25-31).
- Arquitectura Física - Diagrama.



Actividades de aprendizaje recomendadas

Proponga una solución a través del modelo C4 para automatizar el proceso de mantenimiento preventivo vehicular en una mecánica particular. Esto quiere decir que el personal de la mecánica posterior al finalizar el mantenimiento registre los datos del mantenimiento realizado a un vehículo particular en un sistema o aplicación móvil. Posterior al mantenimiento, el dueño del vehículo podrá recibir una alerta a través de su dispositivo móvil cuando esté próximo a retornar a realizar un nuevo mantenimiento (por ejemplo, cuando el siguiente cambio sea a los 5000 km que las alertas empiecen cuando registre 3500, 4000 y 4500 km). Considere un proceso que exponga de forma general el contexto, contenedor, componentes y de ser necesario, clases.

Una vez que ha estudiado los conceptos relacionados a la unidad, le invito a desarrollar la autoevaluación con el fin de evaluar los conocimientos adquiridos hasta el momento.



Autoevaluación 5

1. ¿Qué vista se encarga de la representación lógica de la estructura funcional del sistema, donde normalmente se supone que es un modelo de clase?
 - a. Vista física.
 - b. Vista de procesos.
 - c. Vista de desarrollo.
 - d. Vista lógica.

2. ¿Qué vista muestra los aspectos de concurrencia y sincronización de la arquitectura?
 - a. Vista física.
 - b. Vista de procesos.
 - c. Vista de desarrollo.
 - d. Vista lógica.

3. ¿Qué vista se encarga de la estructura del *software* en tiempo de diseño, la identificación de módulos, subsistemas y capas y las preocupaciones directamente relacionadas con el desarrollo de *software*?
 - a. Vista física.
 - b. Vista de procesos.
 - c. Vista de desarrollo.
 - d. Vista lógica.

4. ¿Qué vista se encarga de la identificación de los nodos en los que se ejecutará el *software* del sistema y la asignación de otros elementos arquitectónicos a estos nodos?
 - a. Vista física.
 - b. Vista de procesos.
 - c. Vista de desarrollo.
 - d. Vista lógica.

5. En el modelo C4, en el Nivel 1, ¿qué tipo de diagrama se utiliza?
 - a. Diagrama de clases.
 - b. Diagrama de secuencia.
 - c. Diagrama de contexto.
 - d. Diagrama de clases.
6. En el modelo C4, el Nivel 2 que hace referencia a los contenedores donde se desplegará la aplicación, ¿a qué tipo de diagrama es similar en el modelo 4+1?
 - a. Diagrama de contexto.
 - b. Diagrama de componentes.
 - c. Diagrama de despliegue.
 - d. Diagrama de clases.
7. En el modelo C4, el Nivel 3, ¿a qué tipo de diagrama es similar en el modelo 4+1?
 - a. Diagrama de contexto.
 - b. Diagrama de componentes.
 - c. Diagrama de despliegue.
 - d. Diagrama de clases.
8. En el modelo C4, el nivel 4 utiliza algunos diagramas UML para entrar en detalle de la solución, ¿cuáles son dichos diagramas?
 - a. Diagrama de contexto.
 - b. Diagrama de componentes.
 - c. Diagrama de despliegue.
 - d. Diagrama de clases.
9. Para la descripción y representación visual gráfica de una arquitectura de software, ¿qué tipo de notación se puede utilizar?
 - a. Informal o no formal.
 - b. Semiformal.
 - c. Formal.
 - d. Ninguna.

10. El lenguaje de modelado unificado UML, ¿a qué tipo de notación corresponde?
- a. Informal o no formal.
 - b. Semiformal.
 - c. Formal.
 - d. Lenguaje de descripción arquitectónica.

[Ir a solucionario](#)



Estamos próximos a finalizar el estudio de la asignatura. Hasta el momento vamos adquiriendo el conocimiento conceptual que combinado con la implementación práctica nos permite entender los conceptos relacionados con estilos, patrones y atributos de calidad. Por su puesto, hacer uso de modelos como el propuesto por Krutchen o por Simon Brown nos ayudarán a describir, visualizar y validar nuestra arquitectura de *software*. En estas semanas le animo a que complementemos el estudio revisando técnicas y tácticas para el diseño, implementación, documentación y validación de arquitecturas de *software* principalmente para aplicaciones web, móviles y servicios que son propuestas en bibliografía como Microsoft Corporation (2019), Brown (2019), Keeling (2017) y Rozanski (2008), y que para su comprensión han sido recopilados para estudiarlos durante estas últimas semanas, empecemos.

Unidad 6. Técnicas y tácticas para diseño, implementación y documentación de arquitecturas de software

Con los conceptos estudiados en semanas anteriores hemos cubierto las áreas de SWEBOK etiquetadas como: 1. Requisitos de Software, 2. Diseño de Software y 10. Calidad de Software, ahora es momento de conocer algunos conceptos propuestos para el área de 3. Construcción de Software y 4. Pruebas de Software las cuales apoyan en el diseño, construcción, pruebas e implementación de una arquitectura de *software*.

Recordemos que hemos estudiado el proceso de definición y diseño de arquitectura, *software* el cual proponía algunas fases para realizarlas de forma iterativa e incremental, ¿lo recuerda? Como todo proceso, se requiere de entradas para a través de *un conjunto de pasos o actividades* generen salidas como el diseño de una arquitectura. Las entradas requeridas para el diseño nos ayudan a formalizar los requisitos y restricciones que la arquitectura debe adoptar. Entre las entradas comunes constan los casos de uso y escenarios de uso, requisitos funcionales, requisitos no funcionales (incluidos atributos de calidad tales como rendimiento, seguridad, confiabilidad, entre otros), requisitos tecnológicos, el entorno de implementación y otras restricciones consideradas importantes.

6.1. Tipos de aplicaciones

En la vida diaria, como ingeniero y arquitecto de software, con los conocimientos adquiridos, usted ya puede proponer o de hecho ya ha creado soluciones que involucren aplicaciones móviles, web, cliente o servicios de forma individual o combinada.

Si recordamos, las **aplicaciones móviles** se pueden desarrollar como aplicaciones de cliente ligero o de cliente enriquecido. Las aplicaciones móviles de cliente enriquecido pueden admitir escenarios desconectados o conectados ocasionalmente (archivo con extensión apk que se instalan y funcionan solo en el dispositivo). Las aplicaciones móviles de cliente ligero (que hacen uso de navegador), solo admiten escenarios conectados. Los recursos del dispositivo pueden resultar una limitación al diseñar aplicaciones móviles. Por otro lado, las **aplicaciones cliente enriquecidas** se desarrollan normalmente como aplicaciones independientes con una interfaz gráfica de usuario que muestra datos usando una variedad de controles. Las aplicaciones cliente enriquecidas pueden diseñarse para escenarios desconectados y ocasionalmente conectados si necesitan acceder a datos o funciones remotos. En relación con los **servicios**, estos exponen la funcionalidad de negocio compartida y permiten a los clientes acceder a ellos desde un servicio o sistema local o remoto. Las operaciones de servicio son llamadas mediante mensajes, basados en esquemas XML, que se pasan a través de un canal de transporte o Bus de Servicios (ESB, *Enterprise Service Bus*). El objetivo de este tipo de aplicación es lograr un acoplamiento flexible entre el cliente y el servidor.

Finalmente, las **aplicaciones web** suelen admitir escenarios conectados y pueden admitir diferentes navegadores que se ejecutan en una variedad de sistemas operativos y plataformas. Como podemos analizar, cada tipo de aplicación se puede implementar utilizando una o más tecnologías que dependiendo de los escenarios, las limitaciones tecnológicas, así como las capacidades y la experiencia de su equipo de desarrollo, impulsarán su elección de tecnología (ver tabla 12).

Tabla 12.
Tipos de aplicación

Tipo de aplicación	Beneficios	Consideraciones
Aplicaciones móviles	Soporte para dispositivos portátiles. Disponibilidad y facilidad de uso para usuarios externos. Soporte para escenarios fuera de línea y ocasionalmente conectados.	Limitaciones de entrada y navegación. Área de visualización de pantalla limitada.
Aplicaciones cliente enriquecidas	Capacidad para aprovechar los recursos del cliente. Mejor capacidad de respuesta, rica funcionalidad de interfaz de usuario y experiencia de usuario mejorada. Interacción altamente dinámica y receptiva. Soporte para escenarios fuera de línea y ocasionalmente conectados.	Complejidad de implementación. Hacen uso de una plataforma específica.
Servicios	Interacciones débilmente acopladas entre cliente y servidor. Puede ser consumido por aplicaciones diferentes y no relacionadas. Soporte para interoperabilidad.	No dispone de interfaz gráfica de usuario. Depende de la conectividad de la red sin interrupciones.
Aplicaciones Web	Amplio alcance y una interfaz de usuario basada en estándares en múltiples plataformas. Facilidad de implementación y gestión de cambios.	Depende de la conectividad de red sin interrupciones. Dispone de una interfaz gráfica de usuario.

Nota. Argüello M., & Guamán D., 2023

En los siguientes apartados se exponen algunas consideraciones generales de diseño y los atributos clave para proponer la construcción de aplicaciones web, móviles y servicios que se proponen en Microsoft Corporation (2009). Esto involucra tener en cuenta conceptos tales como descomposición, capas, componentes, atributos de calidad, patrones arquitectónicos y las consideraciones tecnológicas para el despliegue de una solución.

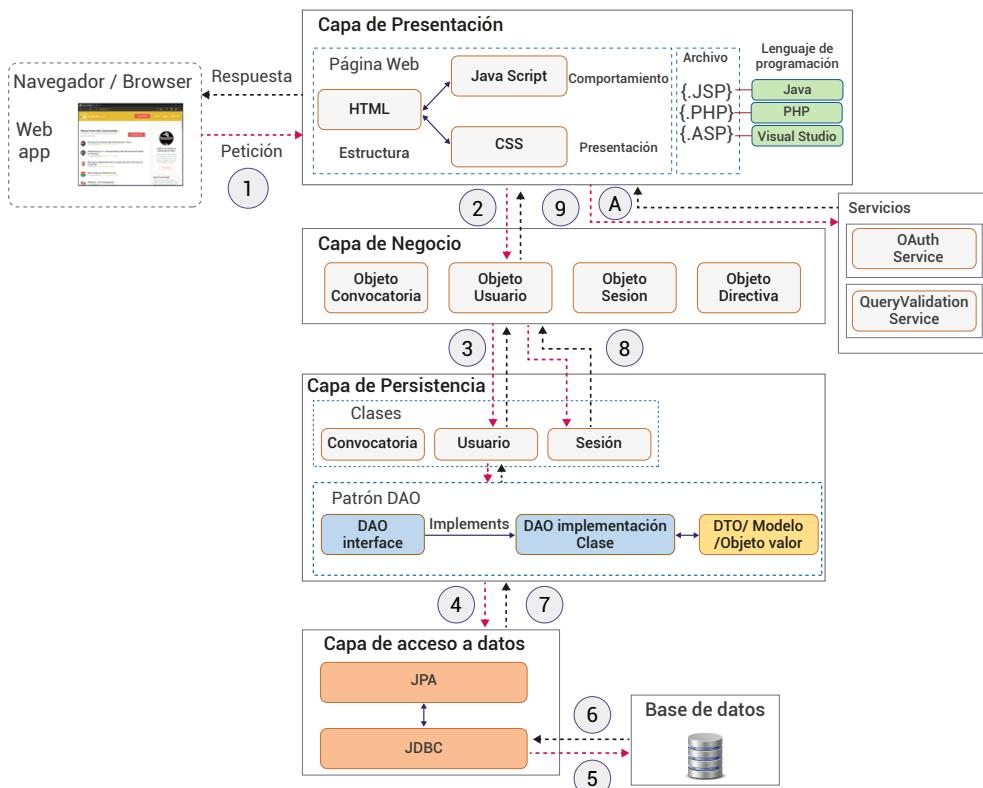
- **Diseñando aplicaciones web**

Seguramente usted conoce que una Aplicación Web es aquella que puede contener varias capas distintas y que permite a los usuarios acceder principalmente a través de un navegador web, donde el navegador crea solicitudes que pueden ser HTTP o HTTPS para URL específicas que se mapean a recursos alojados en un Servidor Web. El servidor procesa y responde a dichas solicitudes a través de páginas HTML que se pueden visualizar en la mayoría de navegadores. El núcleo de una aplicación web es su lógica del lado del servidor. El ejemplo típico es una arquitectura de tres capas compuesta por las capas de presentación, lógica de

negocio, persistencia y datos y que usaremos para explicar los siguientes apartados. La figura 52 ilustra el escenario cuando el usuario realiza una petición a través del navegador, el proceso inicia desde la capa de presentación la cual envía la petición a la capa inferior (negocio) para aplicar distintas reglas de negocio que permiten enviar o solicitar datos desde la base de datos. Con los datos recuperados desde la base de datos, estos retornan de capa en capa hasta llegar a la capa de presentación en donde se visualizan los resultados en la página web. Cabe indicar que la página web se compone de elementos de Front-End como HTML, Java Script y CSS básicamente.

Figura 46.

Capas y solicitud de acceso en una aplicación Web diseñada con patrón 3 - Layers



Nota. Argüello M., & Guamán D., 2023

Como se había estudiado en semanas anteriores, el patrón tres capas muestra la capa de presentación que generalmente incluye componentes lógicos de presentación e Interfaz gráfica de usuario, la capa de negocio

generalmente incluye lógica o reglas de negocio, flujo de trabajo de negocio y componentes de entidades de negocio y la capa de datos generalmente incluye el acceso a datos y componentes de agente de servicio.

Consideraciones generales de diseño

Al diseñar una aplicación web, el objetivo del arquitecto de software es minimizar la complejidad, recuerde alta cohesión y bajo acoplamiento, al separar las tareas en diferentes áreas de interés mientras diseña una aplicación segura y de alto rendimiento. Para lograr ello, se sugiere hacer uso de las siguientes consideraciones:

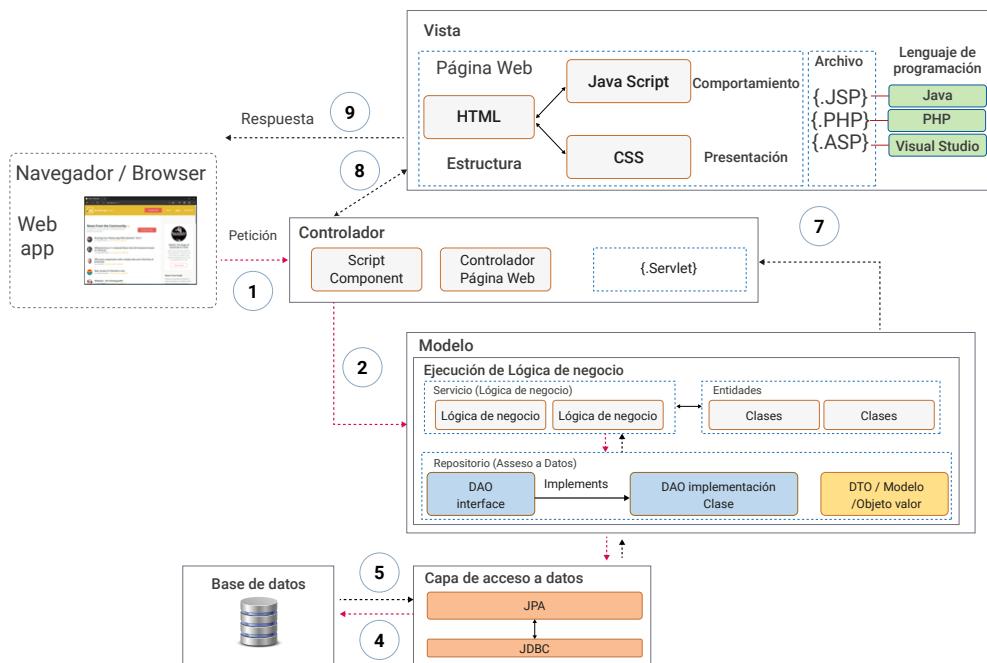
- **Descomponer la aplicación de forma lógica:** utilizando capas para dividir la aplicación de manera lógica en capas, con esto lograremos escalabilidad y crear código mantenable, evaluar, monitorear y optimizar el rendimiento de cada capa por separado.
- **Utilizar la abstracción para implementar un acoplamiento flexible entre capas:** esto se puede lograr mediante la definición de componentes de interfaz, como una fachada con entradas y salidas bien definidas que traduzca las solicitudes a un formato comprendido por los componentes dentro de la capa.
- **Comprender cómo se comunicarán los componentes entre sí:** esto requiere una comprensión de los escenarios de implementación que la aplicación debe admitir. Debe determinar si debe admitirse la comunicación a través de los límites físicos o los límites del proceso, o si todos los componentes se ejecutarán dentro del mismo proceso.
- **Considerar el almacenamiento en caché para minimizar los viajes de ida y vuelta del servidor.** Al diseñar una aplicación web, considere la posibilidad de utilizar técnicas como el almacenamiento en caché y el búfer de salida para reducir los viajes de ida y vuelta entre el navegador y el servidor web, y entre el servidor web y los servidores posteriores, esta estrategia puede ayudar a mejorar el rendimiento.
- **Considerar el registro y la instrumentación:** debe auditar y registrar las actividades en las capas y niveles de su aplicación. Estos registros se pueden utilizar para detectar actividad sospechosa que permita alertar de ataques de seguridad al sistema.

- **No intercambie datos confidenciales en texto plano a través de la red:** Siempre que requiera pasar datos confidenciales como una contraseña o una cookie de autenticación a través de la red, considere la posibilidad de primero cifrar y firmar los datos o utilizar algún algoritmo como por ejemplo Triple DES (3DES) o técnicas de cifrado como por ejemplo Secure Sockets Layer (SSL).
- **Diseñar la aplicación web para que se ejecute con una cuenta con privilegios mínimos.** Si un atacante logra tomar el control de un proceso o acceder a la aplicación, la identidad del proceso debe tener acceso restringido a ciertos componentes considerados de alto riesgo en el sistema para evitar posibles daños.

Una aplicación web también puede diseñarse e implementarse usando como patrón MVC. En la figura 47 se muestra la interacción de forma general desde que el cliente realiza una petición desde un navegador web, a diferencia de una aplicación en capas, en MVC la petición es enviada primero al controlador, quien interactúa con el modelo para recuperar los datos desde el repositorio local o externo, con los datos extraídos en la capa de modelo se mapea y realiza la persistencia de los datos que a través del controlador se podrán redireccionar a la página web que lo solicite.

Figura 47.

Capas y solicitud de acceso en una aplicación Web diseñada con patrón MVC



Nota. Argüello M., & Guamán D., 2023

Como estamos analizando un modelo de arquitectura en capas lógicas, a continuación, se exponen algunas consideraciones de diseño para cada una de ellas, especialmente cuando diseñamos una aplicación web, cliente o móvil:

- **Capa de presentación**

La capa de presentación muestra la interfaz de usuario (UI-Front End) y facilita la interacción del usuario con el sistema. El diseño debe centrarse en la separación de preocupaciones, donde la lógica de interacción del usuario está desacoplada de los componentes de la interfaz de usuario. En aplicaciones web, la capa de presentación consta de un componente del lado del servidor (que representa el HTML) y un componente del lado del cliente (el navegador o agente de usuario que ejecuta los scripts y muestra el HTML). Por lo general, toda la lógica de presentación existe en los componentes del servidor y los componentes del cliente solo muestran el HTML. Con técnicas del lado del cliente como JavaScript o AJAX, es posible ejecutar lógica en el cliente, generalmente para mejorar la experiencia del usuario.

- **Capa de lógica de negocio**

La capa de negocio, nos permite implementar la lógica y los flujos de trabajo de larga duración. El uso de una capa de negocio separada puede mejorar la capacidad de mantenimiento y la capacidad de pruebas de la aplicación, y permite centralizar y reutilizar funciones lógicas de negocio comunes. En la capa lógica se deben diseñar entidades de negocio que representen los datos acordes al contexto y que sean útiles para pasar datos entre componentes.

- **Capa de datos**

La capa de datos es útil para abstraer la lógica necesaria para acceder a la base de datos. Se diseña como un componente separado para que la aplicación sea más fácil de configurar y mantener, y ocultar los detalles de la base de datos de otras capas de la aplicación. A nivel de esta capa se puede usar el patrón de diseño Objeto de Acceso a Datos (conocido como DAO, *Data Access Object*) y Objetos de Transferencia de Datos (conocidos como DTO, *Data Transfer Object*) cuando interactúe con otras capas y se puede aprovechar la agrupación de conexiones para minimizar la cantidad de conexiones abiertas, así como para usar operaciones por lotes que reduzcan los viajes de ida y vuelta a la base de datos. La capa de datos también puede necesitar acceder a servicios externos usando agentes de servicio.

- **Capa de servicios**

El crear un componente o capa de servicios es importante cuando se planea implementar la capa de negocio en una infraestructura externa al sistema principal o cuando se requiere exponer la lógica negocio mediante un servicio web. Cuando la lógica de negocio reside en una infraestructura remota, se debe diseñar métodos detallados en los servicios para minimizar el número de viajes de ida y vuelta y proporcionar un acoplamiento flexible. A nivel de capa de servicios no hay que implementar reglas de negocio en una interfaz de servicio, ya que esto puede dificultar la estabilidad de la interfaz y generar dependencias innecesarias entre componentes y clientes (sistemas u otros servicios). Cuando se trabaja con servicios, la interoperabilidad es primordial, por ello hay que elegir los protocolos, mecanismos de transporte adecuados y decidir si la interfaz

expondrá SOAP, REST o ambos métodos (estos conceptos los explicaremos más adelante).

Consideraciones de despliegue

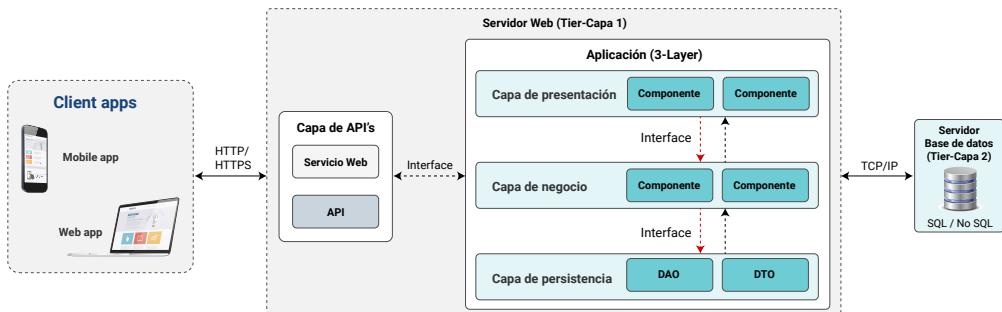
El diseño nos conduce a una codificación y su posterior implementación o despliegue, lo que implica poner en funcionamiento la solución sea este despliegue local o haciendo uso de *cloud* o su combinación. Al desplegar una aplicación *web*, debemos tener en cuenta la distribución de las capas y los componentes (arquitectura lógica y física), ya que podría verse afectado el rendimiento, la escalabilidad y la seguridad de la aplicación. Ante ello, existen opciones de *despliegue distribuida* y *despliegue no distribuida*, según los requisitos de negocio y las limitaciones de la infraestructura. *El despliegue no distribuido* generalmente maximizará el rendimiento al reducir la cantidad de llamadas que deben cruzar los límites físicos. Sin embargo, *el despliegue distribuido* le permitirá lograr una mejor escalabilidad y permitirá que cada capa esté protegida por separado. A continuación, se explica a detalle cada una de ellas.

- **Despliegue no distribuido**

En un escenario de despliegue no distribuido, todas las capas lógicamente separadas de la aplicación *web* están ubicadas físicamente en el mismo servidor *web*, excepto la base de datos. En este escenario, se debe considerar cómo la aplicación manejará varios usuarios simultáneos y cómo proteger las capas que residen en el mismo servidor. La figura 48 muestra este escenario.

Figura 48.

Despliegue no distribuido de una aplicación Web en servidores despliegue local. Se muestra 2 capas físicas Tiers y la aplicación se la distribuye usando el patrón Layers (Servidor Web contiene aplicación y acceso a datos)



Nota. Argüello M., & Guamán D., 2023

Para realizar el despliegue de una aplicación en un entorno no distribuido, hay que tener en cuenta las siguientes consideraciones:

Usar la implementación no distribuida si la aplicación web es sensible al rendimiento, porque las llamadas locales a otras capas reducen el impacto en el rendimiento que causarían las llamadas remotas entre niveles.

Si no necesita compartir la lógica de negocios con otras aplicaciones y solo la capa de presentación accederá a ella, se debe diseñar una interfaz basada en componentes para la capa de negocio.

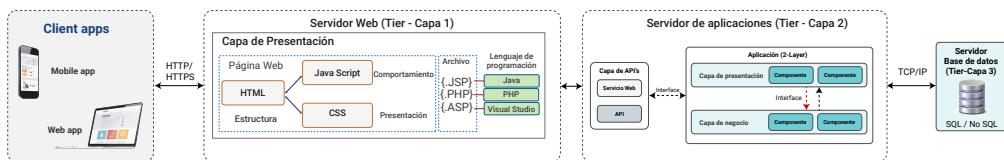
Proteger los datos confidenciales que se transmiten entre el servidor web y el servidor de la base de datos.

■ Despliegue distribuido

En un escenario de despliegue distribuido, las capas de presentación y de negocio de la aplicación web residen en niveles físicos separados y se comunican de forma remota. Por lo general, se debe ubicar las capas de acceso a datos y de negocios en el mismo servidor (ver figura 49).

Figura 49.

Despliegue distribuido de una aplicación Web en servidores despliegue local, se muestra 3 capas físicas y Layers para distribuir la lógica de la aplicación.



Nota. Argüello M., & Guamán D., 2023

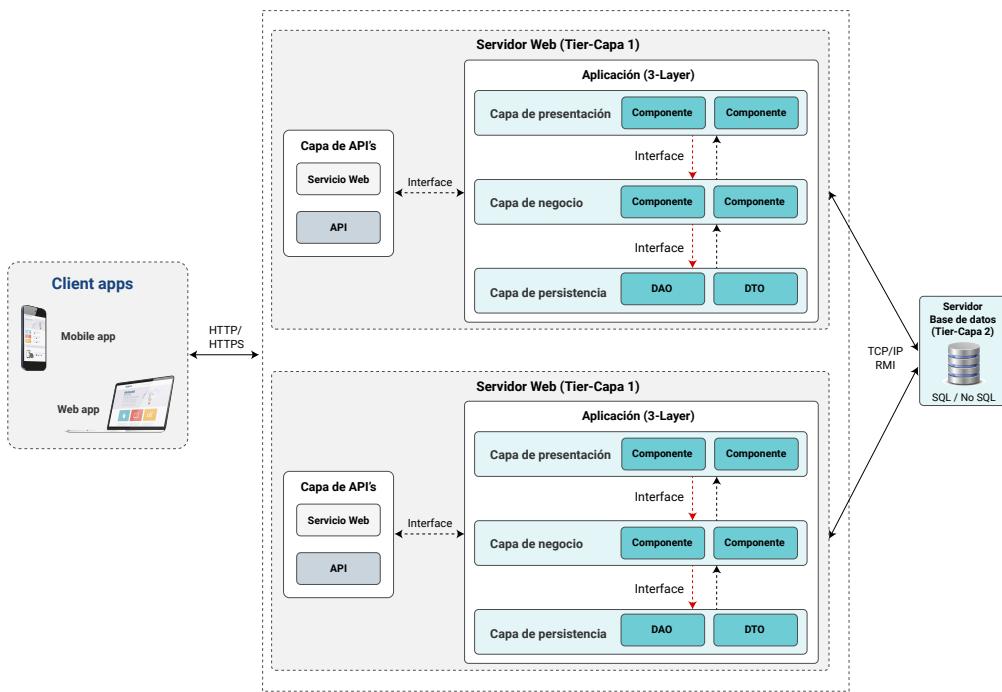
Para elegir un tipo de despliegue distribuido de una aplicación web, se sugiere considerar lo siguiente:

- No implementar la capa de negocio en niveles separados a menos que sea necesario, por ejemplo, cuando se requiera maximizar la escalabilidad o cuando las preocupaciones de seguridad le prohíben implementar dicha lógica empresarial en su servidor web que gestione el Front-end.
- Utilizar una interfaz basada en mensajes para la capa de negocio.
- Utilizar el protocolo TCP con codificación binaria para comunicarse desde la capa de presentación con la capa de negocio para obtener el mejor rendimiento.
- Proteger los datos confidenciales que se pasan entre diferentes niveles físicos.
- **Balanceo de carga**

Una característica importante al desplegar una aplicación web en varios servidores es poder equilibrar la carga para distribuir las solicitudes de modo que sean manejadas por diferentes servidores web. Esto ayuda a maximizar los tiempos de respuesta, el uso de recursos y el rendimiento (ver figura 50).

Figura 50.

Balanceo de carga en una aplicación web (Replica de aplicación en servidores despliegue local)



Nota. Argüello M., & Guamán D., 2023

Para llevar a cabo el balanceo de carga se sugiere tener en cuenta lo siguiente:

- Evitar la afinidad con el servidor al diseñar aplicaciones web, si es posible, porque esto puede afectar negativamente la capacidad de la aplicación para escalar horizontalmente. La afinidad del servidor ocurre cuando todas las solicitudes de un cliente en particular deben ser manejadas por el mismo servidor.
- Diseñar componentes sin estado, por ejemplo, una interfaz web que no tiene estado en proceso ni componentes de negocio con estado.
- Utilizar el balanceo de carga de red del sistema operativo como una solución de software para implementar la redirección de solicitudes a los servidores en una granja de aplicaciones.
- Utilizar la agrupación en clústeres para minimizar el impacto de las fallas de hardware.

- Particionar la base de datos en varios servidores de bases de datos si su aplicación tiene altos requisitos de entrada / salida.

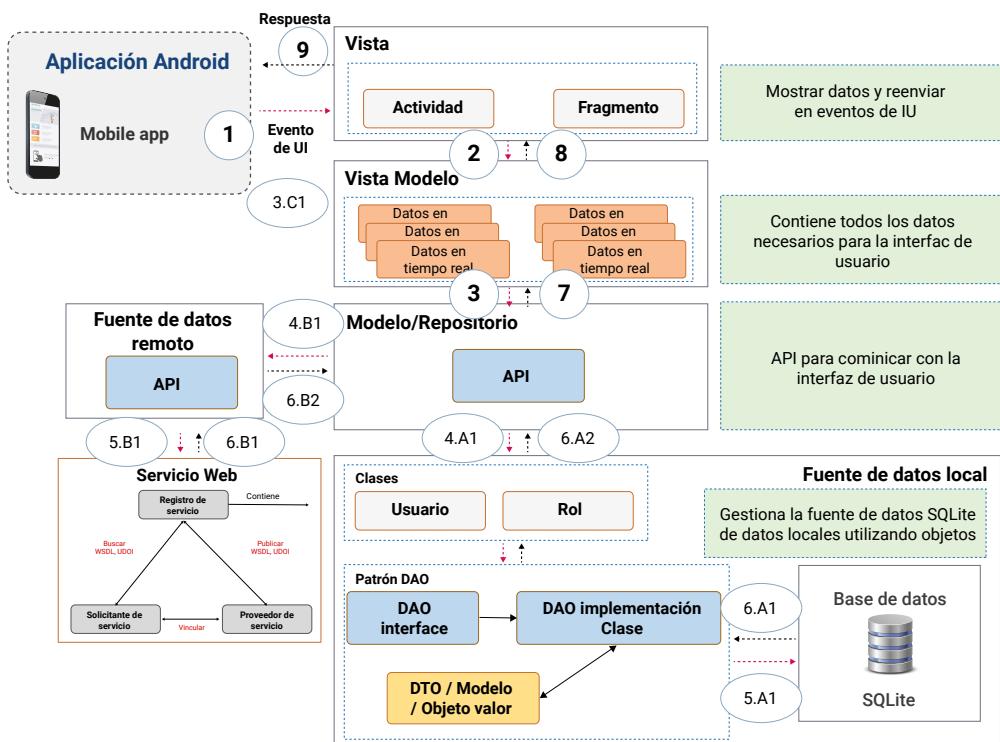
Una vez que conocemos algunas consideraciones y estrategias para diseñar una aplicación *web*, es momento de comprender cuándo y cómo las aplicaciones móviles son una solución adecuada. Esto incluye conocer sus componentes principales, algunos problemas existentes tales como la implementación, el uso de energía y sincronización, y consideraciones tecnológicas.

- **Diseñando aplicaciones móviles**

La estructura de una aplicación móvil normalmente está compuesta de capas de presentación o Vista que contiene los elementos de la interfaz gráfica de usuario, VistaModelo que se utiliza para presentar los datos en tiempo real cuando no se requiere de un viaje, hace la base de datos, una capa de Modelo/Repositorio que contiene una API para poder operar con una Fuente de datos remota o externa como los servicios o cuando se requiere hacer uso de la fuente de datos local que es provista por el sistema operativo. En el caso de requerir recuperar datos externos se utilizará componentes (API) o agentes de servicio y acceso a datos que permite operar la aplicación móvil con el de los servicios *web*, mientras que cuando se requiere usar la base de datos local se puede usar el patrón DAO para manejar el mapeo y persistencia de los datos que pueden ser consumidos localmente o de forma externa (*external storage*) (ver figura 51).

Figura 51.

Capas y solicitud de acceso en una aplicación móvil



Nota. Argüello M., & Guamán D., 2023

Consideraciones generales de diseño

Para diseñar una aplicación móvil se sugieren algunas pautas que permiten cumplir con los requisitos y funcionar eficientemente en escenarios comunes a este tipo de aplicaciones. Entre las consideraciones se sugiere:

- Diseñar una aplicación cliente Enriquecido cuando las solicitudes requieren procesamiento local y su funcionamiento ocasionalmente requiera de un escenario conectado.
- Diseñar una aplicación ligera cuando la aplicación dependa del procesamiento del servidor y siempre estará completamente conectada.
- Determinar los tipos de dispositivos sobre los cuales funcionará su aplicación, ya que esto permitirá tener en cuenta el tamaño y la resolución de la pantalla, las características de rendimiento de la

CPU, la memoria y el espacio de almacenamiento y la disponibilidad del entorno de la herramienta de desarrollo.

- Tener en cuenta los requisitos de los usuarios y las limitaciones de negocio, ya que su aplicación podría requerir utilizar *hardware* específico, como GPS o una cámara, lo que puede afectar no solo su tipo de aplicación, sino también el tipo de dispositivo.
- Considerar escenarios de conexión ocasional y de ancho de banda limitado cuando sea apropiado. Si su dispositivo móvil es un dispositivo independiente, no necesitará tener en cuenta los problemas de conexión. Cuando se requiere conectividad de red, las aplicaciones móviles deben manejar los casos en los que una conexión de red es intermitente o no está disponible. En este caso, es vital diseñar sus mecanismos de almacenamiento en caché, administración de estado y acceso a datos teniendo en cuenta la conectividad de red intermitente, comunicaciones por lotes para la entrega cuando la conectividad esté disponible.
- Diseñar una interfaz de usuario adecuada para dispositivos móviles, teniendo en cuenta las limitaciones de la plataforma. Los dispositivos móviles requieren una arquitectura más simple, una interfaz de usuario más simple y otras decisiones de diseño específicas para trabajar dentro de las restricciones impuestas por el *hardware* del dispositivo. Tenga en cuenta estas limitaciones y diseñe específicamente para el dispositivo en lugar de intentar reutilizar la arquitectura o la interfaz de usuario desde un escritorio o Aplicación web, ya que las principales limitaciones son la memoria, la duración de la batería, la capacidad de adaptarse a diferentes tamaños y orientaciones de pantalla, seguridad y ancho de banda de la red.
- Diseñar una arquitectura en capas adecuada para dispositivos móviles que mejore la reutilización y el mantenimiento. Dependiendo del tipo de aplicación, se pueden ubicar varias capas en el propio dispositivo. Utilice el concepto de capas para maximizar la separación de necesidades, mejorar la reutilización y el mantenimiento.
- Tener en cuenta las limitaciones de recursos del dispositivo, como la duración de la batería, el tamaño de la memoria y la velocidad del procesador. Cada decisión de diseño debe tener en cuenta la CPU, la

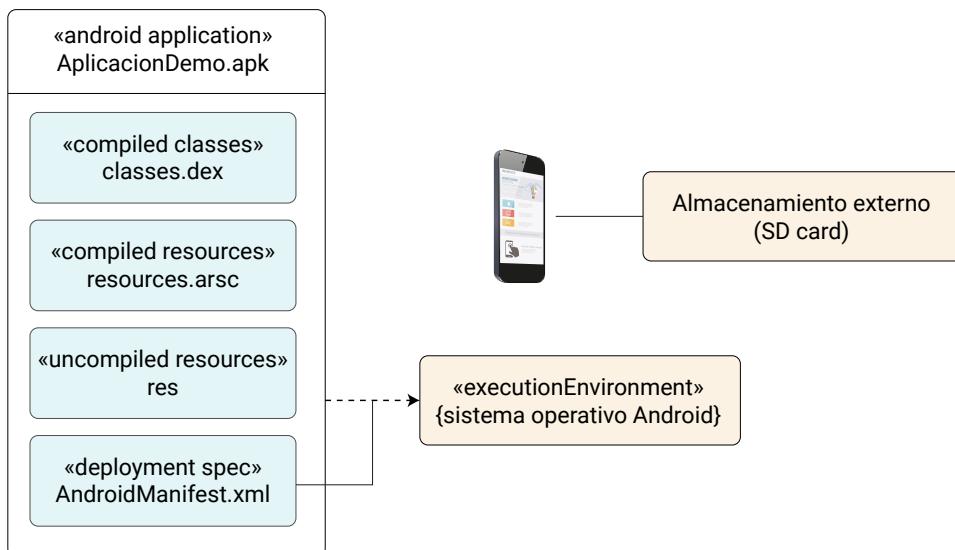
memoria, la capacidad de almacenamiento y la duración de la batería limitadas de los dispositivos móviles. La duración de la batería suele ser el factor más limitante en los dispositivos móviles. La retroiluminación, la lectura y escritura en la memoria, las conexiones inalámbricas, el hardware especializado y la velocidad del procesador tienen un impacto en el uso general de energía. Cuando la cantidad de memoria disponible es baja, el sistema operativo Windows Mobile puede pedirle a su aplicación que cierre o sacrifique los datos almacenados en caché, lo que ralentiza la ejecución del programa. Optimice su aplicación para minimizar su consumo de energía y memoria mientras considera el rendimiento durante este proceso.

Consideraciones de despliegue

Para desplegar una aplicación móvil se debe considerar los requisitos de los usuarios, la forma en la que se instalará y usará la aplicación y aspectos de seguridad cuando lo requiera (ver figura 52).

Figura 52.

Despliegue de aplicación móvil Android



Nota. Argüello M., & Guamán D., 2023

Para desplegar una aplicación móvil se sugiere tener en cuenta lo siguiente:

- Utilice las herramientas y tecnologías adecuadas para instalar la aplicación mediante comandos o procesos automatizados.

- Cuando la instalación y ejecución sea a través de una red, utilice protocolos HTTP/HTTPS y redes inalámbricas seguras, cuando la instalación sea manual use tarjetas de almacenamiento externo (SD card).
- Si la aplicación se ejecutará solo en un sitio específico dentro del dispositivo móvil y desea controlar manualmente la distribución, considere la implementación en tarjeta de almacenamiento externo (SD card).

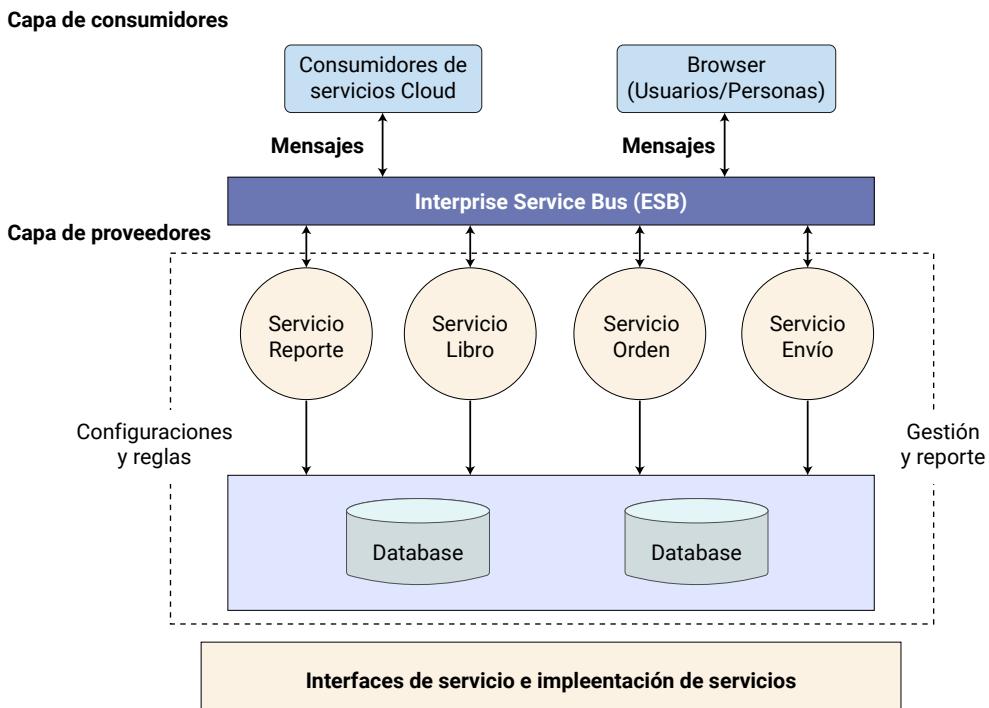
Seguramente hemos diseñado y desarrollado, aplicaciones web y móviles, ¿conocíamos las consideraciones al momento de poner en funcionamiento este tipo de aplicaciones? Muy bien, ahora es momento de conocer sobre la naturaleza y el uso de los servicios, las pautas generales para diferentes escenarios de servicio y algunos atributos clave que debe considerar en términos de desempeño, seguridad, implementación y consideraciones de tecnología. Empecemos.

- **Diseñando servicios**

Como se había mencionado en su momento, un servicio es una interfaz pública que proporciona acceso a una unidad de funcionalidad. Los servicios proporcionan literalmente algún servicio al sistema o cliente que lo invoca o que consume dicho servicio. El bajo acoplamiento y la capacidad de combinarse dentro de una aplicación cliente (escritorio, web, móvil), o combinarse dentro de otros servicios, para brindar una funcionalidad más compleja son algunas características de los servicios. Además, los servicios son distribuibles y se puede acceder a ellos desde un equipo remoto o desde el equipo en donde estos se desarrollan y ejecutan.

Los servicios están orientados a mensajes, lo que significa que las interfaces de servicio se definen mediante un archivo de lenguaje de descripción de servicios web (WSDL) y las operaciones o métodos se llaman mediante esquemas de mensajes basados en XML (*Extensible Markup Language*) que se pasan a través de un canal de transporte o Bus de Servicios Empresariales (ESB). Los servicios soportan el funcionamiento sobre entornos heterogéneos que permiten el uso de diversas tecnologías y lenguajes de programación, permitiendo con ello la interoperabilidad en la definición de mensaje / interfaz, esto quiere decir que si los componentes pueden comprender el mensaje y la definición de la interfaz, pueden utilizar el servicio independientemente de su tecnología base (ver figura 53).

Figura 53.
Arquitectura orientada a servicios



Nota. Argüello M., & Guamán D., 2023

Los servicios son flexibles por naturaleza y se pueden utilizar en una amplia variedad de escenarios y combinaciones, entre los que destacan:

- **Servicio expuesto a través de Internet:** este escenario describe un servicio que es consumido por una variedad de sistemas o servicios clientes a través de *Internet*. Este escenario también incluye servicios de empresa a empresa y servicios centrados en el consumidor.
- **Servicio expuesto a través de una *intranet*:** este escenario describe un servicio que es consumido a través de una *intranet* por un conjunto (generalmente restringido) de clientes internos o corporativos. Una aplicación de gestión de documentos de nivel empresarial sería un ejemplo de este escenario.
- **Servicio expuesto en la máquina local:** este escenario describe un servicio que consume una aplicación en la máquina local. Las decisiones de protección de mensajes y transporte deben basarse en los usuarios y los límites de confianza de la máquina local.

- **Escenario mixto:** este escenario describe un servicio que es consumido por múltiples aplicaciones a través de *Internet*, una *intranet* y / o la máquina local.

Consideraciones generales de diseño

En temas de servicios, hay que tener en cuenta el diseño para operaciones generales, cumplir con el contrato de servicio y anticipar solicitudes no válidas o solicitudes que llegan en forma incorrecta. Algunas pautas que se sugieren para su diseño se exponen a continuación:

- Considere el uso de un enfoque por capas para diseñar servicios y evite el alto acoplamiento entre capas. Separe las reglas de negocio y las funciones de acceso a los datos en distintos componentes/ servicios cuando corresponda.
- Utilice la abstracción para proporcionar una interface en la capa de negocio, esta abstracción se puede implementar mediante interfaces de objetos públicos, definiciones de interfaces comunes, clases base abstractas o mensajería.
- Diseñe operaciones de granularidad gruesa, lo que permitirá centrarse en las operaciones del servicio. Por ejemplo, cuando diseñe un servicio de consulta de clientes, debe proporcionar una operación que devuelva todos los datos en una llamada en lugar de requerir varias llamadas que devuelvan subconjuntos de datos.
- Diseñe los contratos de datos para extensibilidad y reutilización. Los contratos de datos deben diseñarse de modo que pueda ampliarlos sin afectar a los consumidores del servicio.
- Diseñe solo para el contrato de servicio, lo que implica que debe implementar y proporcionar solo la funcionalidad detallada en el contrato de servicio, y la implementación interna y los detalles de un servicio nunca deben estar expuestos a consumidores externos.
- Diseñe servicios que permitan aceptar solicitudes inválidas implementando la lógica de validación para comparar todos los mensajes con los esquemas apropiados y rechazar todos los mensajes inválidos.

- Evite el uso de servicios de datos para exponer tablas individuales en una base de datos. Esto puede generar problemas de dependencia para los consumidores del servicio.

Muy bien, espero se haya comprendido el tema de servicios, especialmente porque hacen uso de dos enfoques conocidos como REST y SOAP que se usan para la transmisión de datos en línea, ambos definen cómo diseñar Interfaces de Programación de Aplicaciones (API), las cuales permiten la comunicación de datos entre aplicaciones web. Revisemos a continuación dichos conceptos:

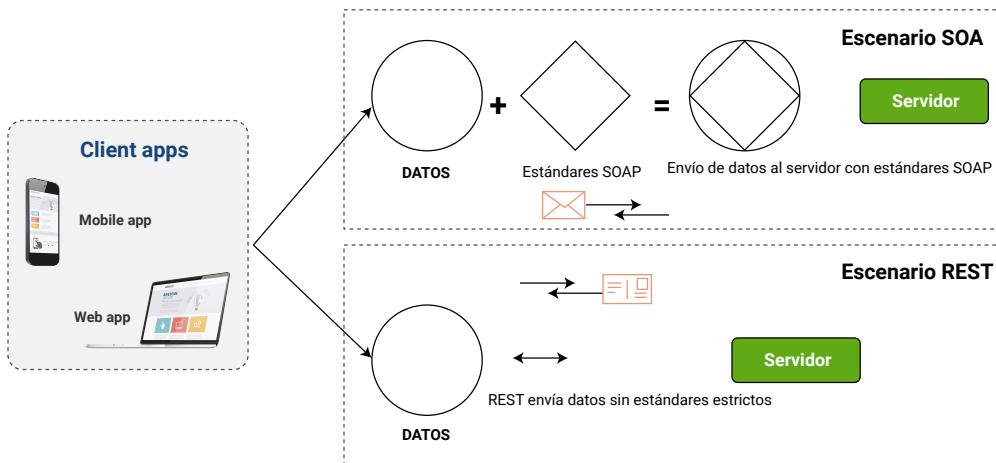
- **API** (*Application and Programming Interface*), permite que una pieza de software se comunique con otra pieza de software. Hay muchos tipos diferentes de API, por ejemplo, API de Facebook, API de Twitter, API de Google, etc. Sin embargo, dentro del diseño e implementación de software cuando se habla de estas API, la mayoría de las veces se refieren a API REST.
- **SOAP** (*Simple Object Access Protocol*), es un protocolo basado en mensajes en el que el mensaje se compone de un sobre XML que contiene un encabezado y un cuerpo. La idea principal detrás del diseño de SOAP era garantizar que los programas creados en diferentes plataformas y lenguajes de programación pudieran intercambiar datos de una manera fácil. SOAP significa Protocolo Simple de Acceso a Objetos.
- **REST** (*Representational State Transfer*), es un estilo arquitectónico que se basa en HTTP y funciona de manera muy similar a una aplicación web, pero sin interfaz gráfica de usuario, esto quiere decir que en lugar de que un usuario interactúe y navegue por las páginas web, las aplicaciones interactúan y navegan por los recursos REST utilizando la misma semántica que una aplicación web (URI). En REST, un recurso se identifica mediante un Identificador Uniforme de Recursos (URI), y las acciones que se pueden realizar contra un recurso se definen mediante verbos HTTP como GET, POST, PUT y DELETE. La interacción con un servicio REST se logra mediante la realización de operaciones HTTP contra un URI, que normalmente tiene la forma de una URL basada en HTTP. El resultado de una operación proporciona una representación del estado actual de ese recurso. Además, el resultado puede contener enlaces a otros recursos a los que puede desplazarse desde el recurso actual.

El error más común cuando se usa REST es pensar que solo es útil para operaciones CRUD contra un recurso, por el contrario, REST se puede utilizar con cualquier servicio que se pueda representar como una máquina de estado, es decir, siempre que pueda dividir o descomponer un servicio en estados distinguibles (como recuperado y actualizado), puede convertir esos estados en acciones y demostrar cómo cada estado puede conducir a uno o más estados.

Cualquier servicio web que se defina según los principios de REST puede denominarse servicio web **RESTful**. Un servicio RESTful usa los verbos HTTP normales de GET, POST, PUT y DELETE para trabajar con los componentes requeridos.

En comparación con REST, SOAP ofrece más flexibilidad de protocolo, por lo que puede utilizar protocolos de mayor rendimiento como TCP (Ver figura 54). SOAP admite los estándares WS-*, incluidos seguridad, transacciones y confiabilidad. La seguridad y confiabilidad de los mensajes garantizan que los mensajes no solo lleguen a su destino, sino también que esos mensajes no se hayan leído ni modificado durante el tránsito.

Figura 54.
Envío de mensajes SOA vs REST



Nota. Argüello M., & Guamán D., 2023

Similar a como se sugirieron algunas pautas para diseñar aplicaciones o servicios, para diseñar recursos REST se sugiere tener en cuenta lo siguiente:

- Usar un diagrama de estado o diagrama de comunicación para modelar y definir recursos que serán compatibles con su servicio REST.
- Elija un enfoque para la identificación de recursos, utilizando nombres significativos para los puntos iniciales de REST y los identificadores únicos, como parte de su ruta general, para instancias de recursos específicos.
- Analice si se deben admitir múltiples representaciones para diferentes recursos. Por ejemplo, si el recurso debe admitir un formato XML, Atom o JavaScript Object Notation (JSON) y conviértalo en parte de la solicitud de recursos.
- Evite valores QueryString para definir acciones en un URI.
- Utilice operaciones HTTP específicas como PUT o DELETE según corresponda para reforzar el diseño basado en recursos y el uso de una interfaz uniforme.
- Use el protocolo HTTP para utilizar una infraestructura web común (almacenamiento en caché, autenticación, tipos de representación de datos comunes, etc.).
- Asegúrese de que sus solicitudes GET sean seguras, lo que significa que siempre devuelvan el mismo resultado cuando se invocan. Considere hacer que sus solicitudes PUT y DELETE sean idempotentes, lo que significa que las solicitudes idénticas repetidas deberían tener el mismo efecto que una sola solicitud.

Una vez que hemos conocido dos de los enfoques importantes para transmitir datos en línea (REST y SOAP), es importante tener en cuenta algunas consideraciones para el despliegue de servicios.

Consideraciones de despliegue

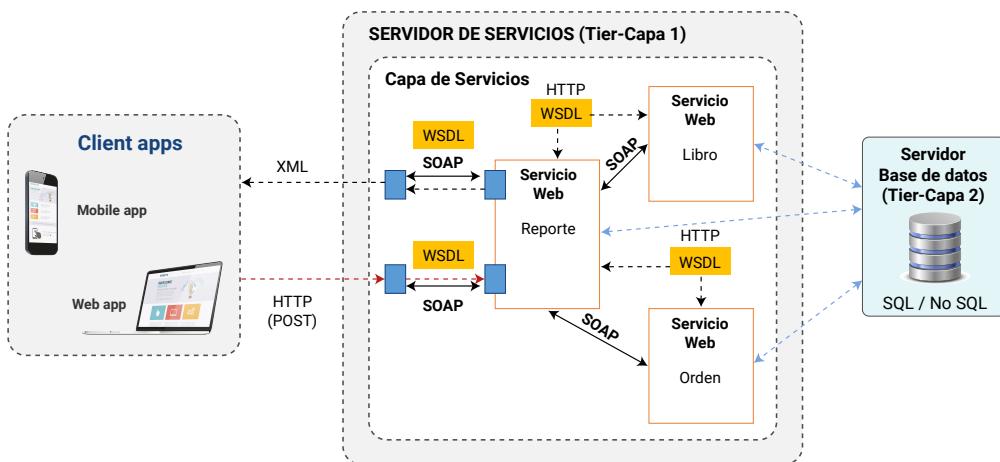
Los servicios generalmente hacen uso de un despliegue distribuido donde los servicios se pueden implementar en un cliente, un solo servidor o desplegarlos en varios servidores. Sin embargo, al implementar servicios, debe considerar los problemas de rendimiento y seguridad inherentes a los escenarios distribuidos y tener en cuenta las limitaciones impuestas por

el entorno de producción. Adicional a ello tenga en cuenta las siguientes pautas:

- Ubique la capa de servicio en el mismo nivel que la capa de negocio con el objetivo de mejorar el rendimiento de la aplicación.
- Cuando un servicio está ubicado en el mismo nivel físico que el consumidor del servicio, considere usar canalizaciones con nombre o memoria compartida para la comunicación.
- Si otras aplicaciones dentro de una red local acceden al servicio, considere usar TCP para la comunicación.
- Configure el *host* del servicio para utilizar la seguridad de la capa de transporte solo si los consumidores tienen acceso directo al servicio y las solicitudes no pasan por intermediarios.
- Desactive el seguimiento y la compilación en modo de depuración para todos los servicios, excepto durante el desarrollo y las pruebas.

En la figura 55 se muestra el escenario de despliegue de Servicios SOA sobre un Servidor despliegue local que aloja a todos los servicios que realizan peticiones y obtienen respuesta desde un servidor de base de datos. En un despliegue con SOA se visualiza que los mensajes hacen uso del protocolo SOAP y WSDL como formato del *Extensible Markup Language (XML)* usado para describir los servicios. Los mensajes SOAP se estructuran con etiquetas XML además de que interviene un lenguaje adicional para su definición, WSDL (*Web Services Description Language*) que se usa para generar una interfaz del Servicio Web, misma que será su punto de entrada. A éste se le conoce como Archivo WSDL o Contrato y una vez que el servicio está publicado, éste archivo estará accesible en una red por medio de un url que tiene terminación ?wsdl, por ejemplo <http://www.dneonline.com/calculator.asmx?wsdl>.

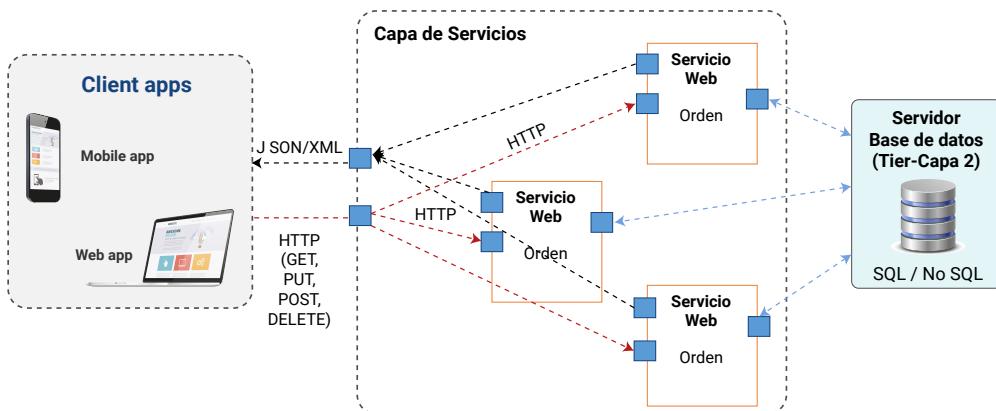
Figura 55.
Escenario de despliegue SOA



Nota. Argüello M., & Guamán D., 2023

En la figura 61 se muestra el escenario de despliegue de Servicios REST sobre un Servidor despliegue local que aloja a todos los servicios que realizan peticiones y obtienen respuesta desde un servidor de base de datos. Pensemos en el escenario cuando una persona hace uso de una aplicación web o móvil. Esta aplicación tiene dos botones “recuperar datos” y “cargar datos”. Cuando el usuario hace *clic* en el botón “recuperar datos”, la aplicación ejecuta una solicitud hacia el proveedor de servicios. En primer lugar, el controlador de puntos finales de la API (recuadros azules en la figura) convierte estas solicitudes en una función estándar (GET; PUT; POST; DELETE). Estas funciones hacen algo de lógica dentro de un proveedor de servicios. Esta lógica comprueba si el usuario tiene derecho a solicitar esos datos. Además, comprueba qué cantidad de datos pueden tener los usuarios. En segundo lugar, los datos se preparan para el usuario. Como resultado, el proveedor de API REST entrega datos en formato XML o JSON a la aplicación. Finalmente, la aplicación toma estos datos y realiza alguna representación gráfica en la interfaz gráfica para que el usuario pueda visualizarlo. En REST una URI tiene el formato. <http://api.example.com/recuperardatos/> o <http://api.example.com/recuperardatos/{criterio}>

Figura 56.
Escenario de despliegue REST



Nota. Argüello M., & Guamán D., 2023

Muy bien, en esta semana hemos conocido algunas consideraciones de diseño y de despliegue cuando la solución incluya aplicaciones web, móviles y servicios. Recuerde que conforme avanzamos con el estudio se descubren términos y conceptos que para complementarlos debe buscar bibliografía disponible de forma digital o en la web. Es momento de conocer algunas consideraciones para la implementación, validación y documentación de una arquitectura, le invito a continuar con el estudio.



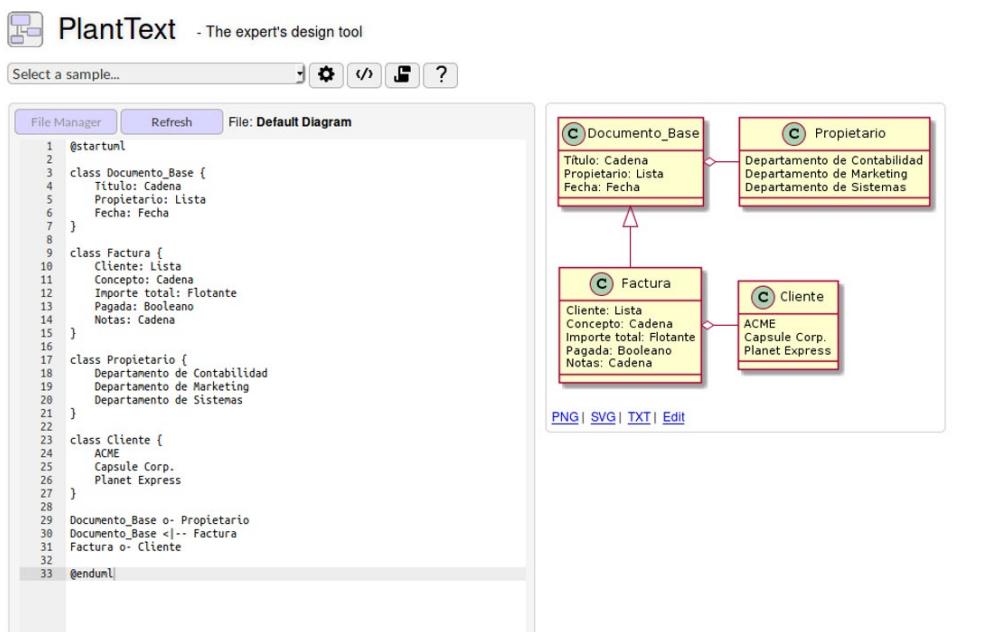
6.2. Implementación de arquitectura

La implementación es la fase de la ingeniería de software que no es opcional. El desarrollo basado en la arquitectura es una actividad de mapeo, donde mantener el mapeo significa asegurar que nuestra intención arquitectónica (diseño o arquitectura candidata) se refleje en la construcción de nuestro sistema de software (código). En el libro *Software Architecture in Practice*, Bass et al. (2013), se proponen cuatro técnicas para ayudar a mantener la coherencia entre el código y la arquitectura:

- **Incorporar el diseño en el código:** la arquitectura actúa como un modelo para la implementación. Esto significa que los desarrolladores e implementadores conocen qué estilo y patrón arquitectónico se está implementando, por ejemplo, Capas, publicador / subscriptor, MVC, Broker, etc, qué principios de diseño, paradigmas de programación y buenas prácticas se están usando (ver figura 57).

Figura 57.

Representando las clases con código en herramienta CASE



Nota. Argüello M., & Guamán D., 2023

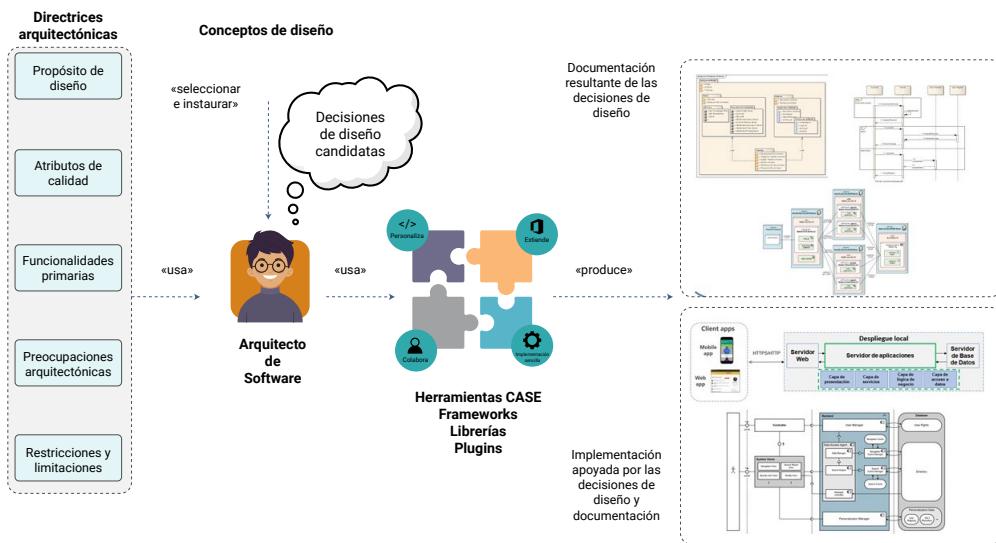
Cada desarrollador puede documentar la estructura arquitectónica en el código utilizando para ello comentarios de línea y de bloque, el objetivo de hacer esto es que cualquiera que tome el código conocerá algunas de las limitaciones o restricciones. Para la inclusión de comentarios se puede hacer uso de herramientas que pueden relacionar automáticamente el código y la arquitectura.

- **Marcos o frameworks de desarrollo:** un marco es un conjunto reutilizable de clases organizadas en torno a un contexto en particular. Un programador *utiliza los servicios proporcionados y predefinidos por frameworks* (ver figura 58), por ejemplo

SpringToolSuite, Ruby on Rails, los cuales se basan en MVC y está diseñado para aplicaciones web, móviles o servicios, herramientas CASE, librerías y *plugins*.

Figura 58.

Uso de herramientas CASE, frameworks, librerías, plugins para diseñar e implementar



Nota. Argüello M., & Guamán D., 2023

La mayoría de los *frameworks* ofrecen la capacidad de recopilar información del servidor web, mapear los datos para operar con la base de datos y utilizando para ello un conjunto de plantillas base. Este es un enfoque ideal, ya que se desarrolla la arquitectura basada en un patrón arquitectónico o de diseño conocido y se seleccionan tecnologías que brinden soporte de implementación para cada elemento arquitectónico.

- **Plantillas de código:** una plantilla de código es una colección de código dentro de la cual el desarrollador proporciona partes específicas de la aplicación. Es necesario utilizar una plantilla de código para cada componente crítico que debe tener un modo de espera activo. Coloque el código específico de la aplicación en lugares fijos dentro de la plantilla. Entre las ventajas de las plantillas de código se tiene por ejemplo que los componentes con propiedades similares se comportan de manera similar. La plantilla solo necesita depurarse una vez. Las partes complicadas pueden ser completadas por personal capacitado y entregadas a personal menos capacitado.

A nivel de implementación, el mantener el código y la arquitectura consistentes nos permitirá evitar la erosión y degradación de la arquitectura que suele ocurrir debido a que la implementación no tiene relación con la documentación arquitectónica representada a través de vistas o componentes. Otro de los factores que ocasionan la degradación es que los desarrolladores pueden tomar decisiones que no sean consistentes entre sí o con la arquitectura. Para prevenir la erosión, se puede hacer uso de herramientas que permitan validar el cumplimiento de las restricciones arquitectónicas, permitiendo agregar reglas de arquitectura que se aplican en tiempo de ejecución, compilación o despliegue de la arquitectura. Cuando la degradación ocurra en la documentación, es porque la documentación puede estar desactualizada y es necesario revisitar los requisitos y la documentación.



Semana 14

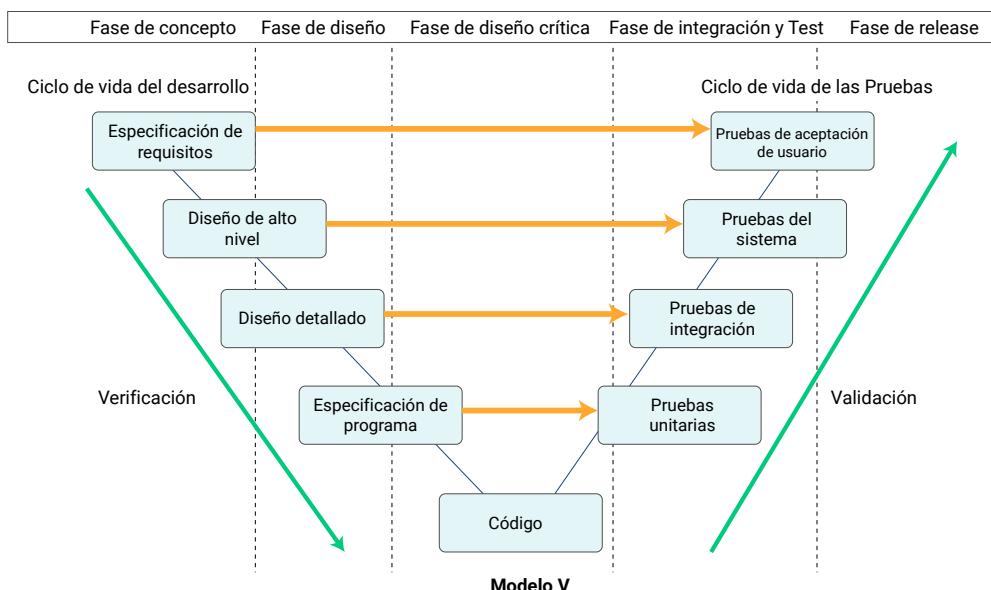
6.3. Pruebas y validación de arquitectura

6.3.1. Pruebas

Posterior a un diseño e implementación total o parcial de los componentes de software, debemos llevar a cabo las pruebas, las mismas que permiten mostrar si el software hace lo que está destinado a hacer y descubrir defectos antes de que el software se ponga en uso. El arquitecto y el líder de desarrollo debe participar activamente en planificación y ejecución de las pruebas, para interpretar los resultados, ya que es quien conoce las áreas sensibles del sistema, según Sommerville (2011) y Pressman (2010).

Para llevar a cabo el proceso de pruebas se debe usar como entrada datos imaginarios o aleatorios (datos de prueba). Las pruebas son parte de un proceso de verificación (técnicas de análisis estático) y validación (técnicas de análisis dinámico) (ver figura 59).

Figura 59.
Fases de la verificación y validación



Nota. Argüello M., & Guamán D., 2023

Verificación es hacernos la pregunta, ¿estamos construyendo el producto correctamente? La verificación es un proceso de validar documentos, diseño, código y componentes específicos para verificar si el software se ha construido o no de acuerdo con los requisitos. El objetivo principal es garantizar la calidad de la aplicación de software, el diseño, la arquitectura, etc.

Por otro lado, para entender la **Validación** debemos hacernos la pregunta ¿Estamos construyendo el producto adecuado? La validación es un mecanismo dinámico para probar y validar si el producto de software realmente satisface las necesidades exactas del cliente o no. El proceso ayuda a garantizar que el software cumpla con el uso deseado en un entorno apropiado.

Entre los objetivos que las pruebas persiguen se exponen los siguientes:

- (01) Demostrar al desarrollador y al cliente que el software cumple con sus requisitos. Para el software personalizado, significa que debe haber al menos una prueba para cada requisito en el documento de requisitos. Para productos de software genéricos, significa que debe haber pruebas para todas las funciones del sistema, además

de combinaciones de estas funciones, que se incorporarán en el lanzamiento del producto.

- (O2) Descubrir situaciones en las que el comportamiento del software es incorrecto, indeseable o no se ajusta a su especificación. La prueba de defectos se ocupa de eliminar el comportamiento indeseable del sistema, como fallas del sistema, interacciones no deseadas con otros sistemas, cálculos incorrectos e inconsistencia de datos.

Tanto para la verificación como para la validación existe un conjunto de actividades que debemos llevar a cabo, entre las que destacan inspecciones, reseñas, tutoriales y comprobación de escritorio para la verificación. Pruebas unitarias, prueba de caja negra, prueba de caja blanca, pruebas de unidad y pruebas de integración son actividades que se siguen para la validación. Cuando se ha implementado el software o algunos de sus componentes, se puede llevar a cabo las pruebas a través de 3 etapas, entre las que constan:

- **Pruebas de desarrollo:** el sistema se prueba durante el desarrollo para descubrir errores y defectos.
- **Prueba de lanzamiento:** un equipo de prueba independiente prueba una versión completa del sistema antes de que se lance a los usuarios.
- **Prueba de usuario:** los usuarios o usuarios potenciales de un sistema prueban el sistema en su propio entorno.

Nos enfocaremos en las pruebas de desarrollo las cuales incluyen todas las actividades de prueba que lleva a cabo el equipo que desarrolla o implementa el sistema, entre ellas constan las pruebas unitarias, prueba de componentes, pruebas de integración, pruebas funcionales, pruebas de sistema, pruebas de rendimiento/estrés. Revisemos brevemente cada una de ellas.

- **Pruebas unitarias:** es el proceso de probar componentes individuales de forma aislada en busca de defectos. En las pruebas unitarias los componentes individuales pueden ser funciones o métodos individuales dentro de una clase / componente, clases de objetos con varios atributos y métodos, componentes compuestos con interfaces definidas que se utilizan para acceder a su funcionalidad.

- **Prueba de componentes:** donde se integran varias unidades individuales para crear componentes compuestos. Las pruebas de componentes deben centrarse en probar las interfaces de los componentes.

En un *proceso de desarrollo iterativo*, la prueba del sistema se ocupa de probar un incremento que se entregará al cliente. En un proceso de *desarrollo en cascada*, la prueba del sistema se ocupa de probar todo el sistema. Las pruebas del sistema pueden tener dos fases distintas entre las que destacan las pruebas de integración y las pruebas funcionales:

- **Pruebas de integración:** se realizan después de integrar un nuevo componente para probar el sistema integrado. Cuando se descubre un problema, los desarrolladores intentan encontrar el origen del problema e identificar los componentes que deben depurarse. Las pruebas de integración, implica la integración de dos o más componentes que implementan cualquier función o característica del sistema y luego probar este sistema integrado.
- **Pruebas funcionales:** se prueba la versión del sistema que podría entregarse a los usuarios (*release*). El objetivo principal es validar el sistema para que cumpla con sus requisitos y garantizar que el sistema sea confiable. En este tipo de pruebas, cuando los clientes participan en las pruebas de lanzamiento, esto se denomina prueba de aceptación.
- **Pruebas de sistema:** donde algunos o todos los componentes de un sistema están integrados y el sistema se prueba como un todo. Las pruebas del sistema deben centrarse en probar las interacciones de los componentes.
- **Pruebas de rendimiento/estrés:** el objetivo de las pruebas de rendimiento es garantizar que el sistema pueda procesar la carga prevista. Una forma eficaz de descubrir defectos son las pruebas de estrés, es decir, diseñar pruebas alrededor de los límites del sistema. En las pruebas de rendimiento el sistema se estresa, haciendo demandas que superan los límites de diseño del software, por lo general haciendo uso de herramientas. Las pruebas de estrés tienen dos funciones (i) Prueba el comportamiento de falla del sistema en circunstancias en las que la carga colocada en el sistema excede la

carga máxima y (ii) estresa continuamente el sistema y puede causar defectos que normalmente no se descubrirían.



Recuerde que posterior a un diseño e implementación llevamos a cabo la fase de pruebas, que incluye seguir un proceso de verificación y validación y que dependiendo del enfoque que usemos ágil o tradicional podemos hacer uso de uno o varios tipos de pruebas. Como podrá visualizar hemos cubierto las fases del ciclo de vida de desarrollo de software, empezando desde los requisitos que pueden ser de negocio, usuario y sistema, luego hemos llevado a cabo actividades de diseño acorde al dominio del problema y dominio de la solución.

Con todos estos insumos hemos realizado la propuesta de solución haciendo uso de un estilo y patrón arquitectónico que representándolo a través de modelos como 4+1 o C4, permite documentar y visualizar los diagramas que serán de apoyo para llevar a cabo la fase de codificación, implementación y pruebas del software que usted proponga. Como siempre le invito a que nos apoyemos en bibliografía disponible en la web de forma digital para comprender algunos de los conceptos que resulten nuevos. Finalmente, nos resta por conocer algunos conceptos referentes a métodos para validación de la arquitectura y la manera en la que podemos documentar una arquitectura.

6.3.2. Validación de arquitectura

Cuando diseñamos e implementamos software, las pruebas nos ayudan a evaluar defectos en etapas tempranas, reducir el costo de mantenimiento y determinar que tan bien es el sistema y sus componentes. En el apartado anterior habíamos mencionado algunas consideraciones y técnicas para llevar a cabo la implementación y despliegue de una aplicación, sin embargo, cuando requerimos realizar las pruebas sobre el diseño de la arquitectura, Microsoft Corporation (2009) sugiere considerar lo siguiente:

- Definir claramente las entradas y salidas de las capas y componentes durante la fase de diseño.

- Considerar utilizar estilos y patrones arquitectónicos como Capas, MVC, Broker u otro, con lo cual podemos realizar pruebas unitarias sobre cada componente.
- Diseñar la capa de negocio como componente separada para implementar la lógica de negocio y los flujos de trabajo con el objetivo de mejorar la capacidad de prueba.
- Diseñar componentes con el objetivo de alcanzar la alta cohesión y bajo acoplamiento y poder realizar las pruebas individualmente.
- Diseñar una estrategia eficaz de registro y seguimiento de *issues* y *bugs* que le permita detectar o solucionar errores que, de otro modo, serían difíciles de encontrar.
- Proporcionar información de registro y seguimiento que pueda ser consumida por herramientas de supervisión o administración (*log's* de errores).

Cuando se habla de evaluación del diseño se puede considerar la evaluación por parte del diseñador dentro del proceso de diseño, la evaluación por pares dentro del proceso de diseño y el análisis por parte de terceros una vez diseñada la arquitectura. Expongamos algunas consideraciones de cada una de ellas.

- **Evaluación por parte del diseñador:** cada vez que el diseñador toma una decisión de diseño clave o completa un hito de diseño, se deben evaluar las alternativas elegidas dentro del contexto. La evaluación por parte del diseñador es la parte de prueba del enfoque de generar y probar el diseño de arquitectura. Algunos de los factores que el evaluador tiene en cuenta respecto de las decisiones de diseño incluyen:
 - La importancia de la decisión, esto quiere decir que cuanto más importante sea la decisión, más cuidado se debe tener al tomarla y asegurarse de que sea correcta.
 - El número de alternativas potenciales, esto se refiere a cuantas alternativas adicionales y el tiempo adicional que podría dedicar a evaluarlas. Lo más importante es tomar una decisión y continuar con el proceso de diseño, que estar absolutamente seguro de que se está tomando la mejor decisión, recuerde el

proceso de diseño, el cual es iterativo e incluye actividades de refinamiento.

- **Evaluación por pares:** al igual que en temas de codificación, a nivel de diseño arquitectónico estos pueden ser revisados por pares. Esta evaluación se puede llevar a cabo en cualquier punto del proceso de diseño donde exista una arquitectura candidata, o donde exista al menos una parte revisable coherente que cumpla con los requisitos. Para llevar a cabo la evaluación por pares, se puede seguir los siguientes pasos:
 1. Los revisores determinan una serie de escenarios de atributos de calidad para llevar a cabo la revisión. Estos escenarios pueden ser desarrollados por el equipo de revisión o por partes interesadas adicionales.
 2. El arquitecto presenta la parte de la arquitectura (componentes individuales o integrados) a evaluar. Los revisores se aseguran individualmente de que se comprenda la arquitectura.
 3. Para cada escenario, el diseñador recorre la arquitectura y explica cómo se satisface un escenario en particular. Los revisores a través de preguntas y respuestas determinan si el escenario satisface los requisitos y si alguno de los escenarios que se están considerando no se cumplirá.
 4. Se capturan los problemas potenciales, los cuales deben ser solucionados por los arquitectos o los diseñadores y las partes interesadas.
- **Análisis por parte de terceros:** el análisis por terceros incluye especialmente entregarles a los evaluadores artefactos o documentación que describan la arquitectura. Este tipo de análisis se realizan con pleno conocimiento y participación de todas las partes interesadas y en algunos casos se realizan de forma más privada. Las evaluaciones pueden ser realizadas por un individuo o por un equipo. El proceso de evaluación debe proporcionar un método para obtener los objetivos y preocupaciones que las partes interesadas importantes tienen con respecto al sistema con el objetivo de que la evaluación responda a si el sistema satisfará los objetivos de negocio.



6.4. Documentación de la arquitectura

A veces menos, es más. La documentación arquitectónica debe tener la información mínima necesaria para comprender el sistema, pero no más que eso. Incluso la mejor arquitectura será poco útil si las personas que la necesitan no entienden o no conocen lo que es o para qué sirve o no puede entenderlo lo suficientemente bien como para usarlo, construirlo o modificarlo. Un documento de arquitectura de software es un mapa del software que lo usamos para visualizar, cómo está estructurado el software. Sirve como apoyo para comprender los módulos y componentes del software sin tener que profundizar en el código. Es una herramienta para comunicarse con otros, desarrolladores y partes interesadas, sobre el software.

La documentación debe contener los tipos de esquemas arquitectónicos y diagramas, para ello se puede hacer uso de notación informal, semiformal y formal.

- **Notación informal:** es el tipo de diagramas más común. Puede representarse de cualquier forma. A menudo actúa como la forma más fácil y rápida de comunicación entre las partes interesadas. Entre las desventajas, es casi lo mismo comprender o confundirse, por eso se requiere una descripción detallada del diagrama. Las notaciones más informales son más fáciles de crear, pero ofrecen menos garantías. Diferentes notaciones son mejores (o peores) para expresar diferentes tipos de información.
- **Notación semiformal:** es un esquema o diagrama gráfico estándar que tiene ciertas reglas de creación. Entre las desventajas, no proporciona una descripción completa de cada elemento específico. Por lo tanto, requiere conocimiento de la semántica del diagrama específico. Los diagramas UML son semiformales, así como ciertos enfoques para crear diagramas. El diagrama de clases UML no le ayudará a razonar sobre la capacidad de programación, ni un diagrama de secuencia le dirá mucho sobre la probabilidad de que el sistema se entregue a tiempo.

- **Notación formal:** normalmente, las notaciones más formales requieren más tiempo y esfuerzo para crear y comprender, pero ofrecen menos ambigüedad y más oportunidades de análisis. De alguna forma es un Lenguaje para Describir la Arquitectura (ADL, *Architecture Description Language*). Permite generar código directamente a partir de los esquemas creados, por lo tanto, se requiere del uso de software y conocimientos especiales, tanto para la escritura como para la comprensión de las partes interesadas.

Por lo tanto, cuando hablamos de documentar una arquitectura, hay que elegir las notaciones y lenguajes de representación sabiendo los temas importantes que necesita capturar y razonar. Algunas consideraciones para elaborar la documentación se exponen a continuación:

- Evite las repeticiones, al igual que en programación y en la documentación de la arquitectura, cree enlaces o referencias, pero no escriba o represente gráficamente lo mismo varias veces.
- Reconozca la audiencia a la que está dirigida la documentación, esta es una regla importante, por ejemplo, la documentación para desarrolladores y alta gerencia puede diferir drásticamente. Por lo tanto, es necesario decidir para quién lo está haciendo, luego comprender qué necesitan estas personas en su documentación y qué respuestas intentan encontrar.
- Evite la ambigüedad, este es uno de los problemas más frecuentes con la documentación. Por ejemplo, si su solución está documentada en forma de diagramas, debe preguntarse, ¿lo entiende?, ¿conoce el contexto?, puede ser que sí, pero la persona que lo revisó puede que no lo conozca, y eso puede traer confusiones. Por lo tanto, siempre debe proporcionar contexto para sus diagramas y acompañarlos de una descripción. Y cuando sea el caso, haga uso de modelos como 4+1 y C4 model, ya que le permitirán primero hacer una descripción de alto nivel del sistema y luego sumergirse en los detalles. a nivel de componentes, contenedores y despliegue, y también tiene criterios razonablemente formales para describir cada elemento.
- Mantenga la relevancia, esto le permitirá encontrar un compromiso entre el tiempo invertido y la relevancia. Con ello podrá determinar si es más relevante mantener la documentación o es mejor

implementarlo con la documentación que se genere y actualice de a poco.

- Revise la documentación, debería revisar periódicamente la documentación. Por lo tanto, lo más probable es que encuentre vistas o diagramas que han perdido relevancia y que vale la pena actualizar.
- Comparta la documentación con las partes interesadas y solicite su opinión, especialmente cuando está comenzando a documentar la arquitectura.

Finalmente, tenga presente que para documentar una arquitectura lo que realiza es una combinación de representaciones gráficas (diagramas que pueden agruparse en vistas) acompañado de explicaciones textuales en la que se evidencie el uso de elementos arquitectónicos. Un ejemplo de las principales secciones que forman parte de la documentación arquitectónica y algunas aplicaciones, diseño y codificación se presentan en la sección de bibliografía complementaria y se cargarán en la plataforma educativa virtual, donde se visualiza las siguientes secciones dependiendo del modelo que utilice. Aunque en resumen el documento de arquitectura podrá visualizarlo con un ejemplo práctico en el anexo 2.

Muy bien, con la revisión de algunas pautas para llevar a cabo la documentación de la arquitectura de software de su solución, hemos finalizado el estudio de la asignatura en este segundo bimestre. Le animo a que podamos complementar los temas en caso de que se requiera, haciendo uso de bibliografía complementaria o consultando a su profesor tutor a través de la plataforma educativa virtual. Para complementar los conceptos le animo a desarrollar las siguientes actividades.

Lecturas en bibliografía básica, complementaria y recursos educativos abiertos.

- Revise la bibliografía complementaria para conocer y utilizar técnicas y estrategias que le permitan diseñar, codificar e implementar una aplicación web.
 - **Libro de Presman** – (Capítulo 13 – Diseño de Webapps. Subtemas 13.6 Diseño del contenido, 13.7 Diseño Arquitectónico).

- (Capítulo 20 – Prueba de aplicaciones web, 20.3 Pruebas de contenido, 20.4 Prueba de Interfaz de Usuario, 20.5 Prueba a nivel de componente).
 - **DD2_Arquitectura de aplicaciones**
Tiers y Layers explicado con ejemplos de modelo y código
(Páginas 2-31).
- Revise la bibliografía complementaria y lecturas recomendadas para conocer y utilizar técnicas y estrategias que le permitan diseñar, codificar e implementar una aplicación móvil y servicios web.
- **Lecturas recomendadas en la web:** Todo lo que necesitas saber sobre la arquitectura de aplicaciones móviles.
<https://www.peerbits.com/blog/all-about-app-architecture-for-efficient-mobile-app-development.html>
- Revise la bibliografía complementaria, identificar y seleccionar los tipos y técnicas de pruebas para verificar y validar su arquitectura de software.
- **Libro de Presman** – (Capítulo 17 – Estrategias de prueba de software subtemas 17.1 Un enfoque estratégico para la prueba de software, 17.1.1 Verificación y validación, 17.3.1 Pruebas de unidad, 17.3.2 Pruebas de integración, 17.5 Estrategia de pruebas para Webapps, 17.7 Pruebas del sistema).
 - **DD1_Arquitecturas software** - Validación de la arquitectura (Páginas 54-57).
 - **DD3_Arquitecturas de Software** - Evaluación de arquitecturas de software (Páginas 35-43).



Actividades de aprendizaje recomendadas

- Proponga un ejemplo en el cual se evidencie a través de nomenclatura informal y semiformal un problema (situación inicial) y una propuesta de solución a un problema dentro de su localidad,

empresa o negocio. Trate de utilizar los conceptos de diagrama de contexto, contenedores, componentes.

- Como estudiantes hacemos uso del sistema de matrícula en línea -UTPL, ¿qué tipo de pruebas y qué escenarios propone para llevar a cabo las pruebas si usted fuera el arquitecto o desarrollador de software?, exponga algunos escenarios que validaría o probaría ya en la fase de implementación actual del sistema.
- Una vez que ha estudiado los conceptos relacionados a la unidad, le invito a desarrollar la autoevaluación con el fin de evaluar los conocimientos adquiridos hasta el momento.



Autoevaluación 6

1. Cuándo representamos y diseñamos una componente de Interfaz Gráfica de Usuario (UI) o vista para una aplicación web, ¿cuál es la mejor opción para representar los componentes internos que interviene?
 - a. HTML, JavaScript, CSS, Ajax.
 - b. Jdbc, Odbc, Store Procedures.
 - c. Interface, extends, implements.
 - d. Python, Larabel, C#.
2. Cuando diseña una aplicación web y utiliza una capa de servicios para comunicarse con otros servicios externos a la aplicación, ¿qué podría utilizar?
 - a. JDBC (Java Database Connectivity).
 - b. ESB (Enterprise Service Bus).
 - c. ERP (Enterprise Resource Planning).
 - d. RSP (Remote Store Procedure).
3. ¿Cuál de las siguientes opciones se considera correcta cuando se requiere intercambiar mensajes al implementar servicios?
 - a. SOAP permite muchos formatos de datos diferentes.
 - b. REST permite usar muchos formatos de datos como por ejemplo XML, JSON, HTML.
 - c. SOAP hacer uso de WSDL (Web Services Description Language).
 - d. Todas las opciones son correctas.

4. ¿A qué concepto se asocia el siguiente texto? Se considera como una implementación de software autónoma, desarrollada, implementada, administrada y mantenida que respalda la funcionalidad empresarial específica para una empresa en su conjunto y es “integrable” por diseño.
- Servicio.
 - Microservicio.
 - SOAP.
 - WSDL.
5. A nivel de documentación e implementación, un servicio bancario, como por ejemplo “validar la calificación crediticia del cliente”, que describe la función comercial que implementa el negocio, se define mediante:
- Un sujeto.
 - Un nombre.
 - Un verbo.
 - Un adjetivo.
6. RPC (*Remote Procedure Call*), ESB (*Enterprise Service Bus*) y REST (*representational state transfer*) se consideran conectores. En este sentido, ESB, ¿qué estilo arquitectónico utiliza?
- Cliente – Servidor.
 - Capas.
 - Publicador – suscriptor.
 - Todos los anteriores.
7. Cuando implementa una aplicación móvil, ¿cuáles de las siguientes opciones son correctas?
- Para su instalación y despliegue se puede usar SD-card.
 - Desde la aplicación móvil se puede consumir servicios o microservicios.
 - La aplicación móvil puede hacer uso de dispositivos como cámara, GPS, micrófono u otro *hardware* interno.
 - Una aplicación móvil puede hacer uso de fuentes de dato interna como SQLite o fuentes de datos externa.

8. Cuando se utiliza REST, ¿cuáles son algunos de los métodos que se pueden realizar?
- a. GET.
 - b. POST.
 - c. PUT.
 - d. DELETE.
9. Cuando se despliega de forma separada una aplicación web y la base de datos en dos servidores ubicados físicamente en la sala de servidores (físicamente en un cuarto en UTPL), ¿qué tipo de despliegue está utilizando?
- a. Despliegue distribuido.
 - b. Despliegue no distribuido.
 - c. Despliegue con balanceo de carga.
 - d. Despliegue local (*on – premises*).
10. ¿Cuáles de las siguientes opciones se considera correcta respecto del objetivo de documentar una arquitectura?
- a. Comunicar a las partes interesadas una propuesta de solución.
 - b. Utilizar un conjunto de vistas para comunicar la arquitectura.
 - c. Contiene las decisiones y restricciones de diseño e implementación.
 - d. Es un documento basado en el estandar IEEE-830.

[Ir a solucionario](#)



Actividades finales del bimestre

¡Muy bien!, hemos completado el estudio de algunos conceptos que nos han permitido realizar aplicaciones prácticas para comprender la Arquitectura de Software. Para apoyar a su comprensión y aprendizaje se han seleccionado e incluido una serie de conceptos básicos y necesarios para que sirvan de ayuda al momento de analizar, diseñar, documentar e implementar una solución *software*.



Recuerde que usted va adoptando los roles de analista, diseñador, arquitecto de software, control de calidad con los cuales aplica las competencias para construir un producto de software.

La práctica y experiencia que vaya adquiriendo o ya la tenga dentro del ámbito profesional donde se desenvuelve actualmente, le ayudará a decidir sobre el uso de nomenclatura informal y semiformal que le permitirán representar diagramas a través de modelos como 4+1 o C4.

Hasta el momento y con ayuda de las unidades estudiadas en el primer y segundo bimestre, usted debe estar en capacidad de proponer una solución que hace uso de estilos y patrones arquitectónicos, y usar la documentación generada para validar y probar a nivel de estructura y de funcionamiento si la solución cumple con los requisitos dados por los clientes. Recuerde hacer uso de técnicas y estrategias para analizar, diseñar e implementar una solución que sea comprensible por las partes interesadas.



4. Solucionario

En esta sección se ubica el solucionario a cada una de las autoevaluaciones con su respectiva retroalimentación de ser necesario.

Autoevaluación 1		
Pregunta	Respuesta	Retroalimentación
1.	d	La arquitectura de <i>software</i> representa la estructura o las estructuras del sistema, que consta de componentes tanto internos como externos y las relaciones entre ellos.
2.	c	Las vistas representan un aspecto parcial de una arquitectura de <i>software</i> que muestran propiedades específicas del sistema que puede ser leído y entendido por las partes interesadas que lo requieran.
3.	d	Las estructuras de módulo se refieren a la implementación de los elementos arquitectónicos (componentes y conectores) a través de tecnologías y lenguajes de programación.
4.	a	La estructura de componentes y conectores se utiliza para representar a nivel de diseño y de forma abstracta los elementos arquitectónicos que permitirán llevarlos a una implementación.
5.	d	La estructura de asignación permite identificar como se intercambiará los datos, los mensajes a través del <i>hardware</i> donde se desplegará el sistema, permite además identificar los puertos, protocolos y tecnologías que pueden usarse.
6.	a, b, c, d	La arquitectura del <i>software</i> es importante porque todos los sistemas de <i>software</i> tienen una arquitectura. Incluso cuando una aplicación tiene una sola estructura con un elemento, hay una arquitectura. Hay sistemas de <i>software</i> que no tienen un diseño formal y otros que no tienen documentación formal, pero incluso estos sistemas también tienen una arquitectura.
7.	b	SWEBOk es un documento creado por el Software Engineering Coordinating Committee, promovido por el IEEE Computer Society, el cual es una guía al conocimiento en el área de la Ingeniería del Software.

Autoevaluación 1		
Pregunta	Respuesta	Retroalimentación
8.	a	Cuando se desea diseñar e implementar una arquitectura se debe tener en cuenta a la estructura estática para diseñar los diagramas, la estructura dinámica para analizar el comportamiento e intercambio de mensajes entre los componentes, el comportamiento visible externamente al momento de implementar los módulos y las propiedades de calidad acorde al contexto y tipo de aplicación.
9.	d	Todas las aplicaciones tienen una arquitectura, cuando no existe documentación que describe la estructura interna como externa será muy complejo llevar a cabo tareas a nivel de construcción, implementación, mantenimiento y evolución.
10.	d	El arquitecto de software es la persona que tiene conocimientos y habilidades que le permiten diseñar y proponer el uso de tecnologías para documentar, construir e implementar una arquitectura de software.

[Ir a la
autoevaluación](#)

Autoevaluación 2		
Pregunta	Respuesta	Retroalimentación
1.	e	Cuando definimos y diseñamos una arquitectura de software debemos pensar en aspectos técnicos y no técnicos que nos permitan construir soluciones escalables, funcionales, mantenibles, seguras y que satisfagan las necesidades de los clientes y los objetivos de negocio.
2.	a	La arquitectura no solo debe ser elaborada por una persona, sino que debe ser el resultado de necesidades dadas por los Stakeholders, a nivel técnico las decisiones de diseño deben ser conversadas y evaluadas por los arquitectos de software, especialmente cuando se trabaja en sistemas medianos o grandes.
3.	c, d	Una arquitectura no se puede considerar buena o mala si no se la implementa de manera iterativa y se visualiza su documentación arquitectónica asociada.
4.	c	Para que una arquitectura se considere buena, a más de implementar los componentes de forma incremental se debe identificar las responsabilidades de cada uno de ellos y hacer uso de los principios de diseño, entre los que se incluyen modularidad, alta cohesión y bajo acoplamiento.
5.	b	La arquitectura es el paso previo a la implementación, por tanto, se requiere primero identificar y evaluar los módulos que consumirán y producirán datos, con el objetivo de evitar problemas a nivel de rendimiento o escalabilidad.
6.	c	Pensar que en una sola iteración obtendremos una buena arquitectura no es algo normal, se deben evaluar los problemas o conflictos que puedan existir y proponer alternativas de diseño y escenarios en los que puedan ocurrir para mitigar los problemas.
7.	c	Dependiendo del tipo de aplicación, del contexto o dominio de la solución se diseña una arquitectura candidata y siempre recuerde que no todo diseño es una arquitectura, ni toda arquitectura es un diseño.
8.	e	Una arquitectura debe diseñarse para satisfacer una necesidad, debe documentarse para soportar su construcción e implementación y debe ser validada para cumplir con atributos de calidad.
9.	e	El tipo de aplicación, la forma en la que se piensa implementar la solución, las tecnologías disponibles acordes a un presupuesto y los atributos de calidad son consideraciones para diseñar y proponer una arquitectura de software.

Autoevaluación 2		
Pregunta	Respuesta	Retroalimentación
10.	c	Entender el contexto es el primer paso de un diseño semi – formal para comprender las necesidades y pensar en lo que sería el dominio de la solución ante un dominio de problema.

Ir a la
autoevaluación

Autoevaluación 3		
Pregunta	Respuesta	Retroalimentación
1.	e	Cuando se define y diseña una arquitectura debemos entender el problema para dar solución que cumpla con los objetivos y necesidades de las partes interesadas, y sobre todo que puedan entender y comprender las decisiones de diseño.
2.	a	Entender el problema y documentar su alcance acompañado de una representación gráfica que muestre el contexto del problema es fundamental en el inicio del proceso de definición y diseño arquitectónico.
3.	e	Entender las necesidades, transformarlas y documentarlas como requisitos de negocio, usuario, sistema, restricciones y riesgos permitirán mejorar la comunicación entre el arquitecto y las partes interesadas.
4.	c	El diagrama de contexto nos permitirá visualizar los procesos que intervienen en la solución, así como las tecnologías de información y comunicación se pueden utilizar.
5.	c	Los arquitectos y diseñadores se encargan de definir los elementos arquitectónicos (componentes y conectores) que posteriormente los desarrolladores los implementarán como módulos y funcionalidades del sistema.
6.	a, b, d	En el diseño de una arquitectura se debe tener en cuenta los requisitos que son visibles a nivel de sistema, aquellos que no son visibles, pero que son de utilidad en la construcción del software y especialmente la documentación representada a través de diagramas.
7.	e	Los diagramas que se asocian a las vistas arquitectónicas ayudan a describir la arquitectura desde el punto de diseño.
8.	a	El primer paso es identificar los objetivos de la arquitectura, es decir para qué se va a utilizar, por qué y cómo, luego se usa escenarios clave que son asociados a los procesos, se debe tener en cuenta el tipo de aplicación y pensar en los posibles problemas que pueden existir y como se los mitigará.
9.	b	La arquitectura candidata es un diseño inicial o base de lo que podría ser una solución al problema, luego hay que refinar los elementos arquitectónicos que intervienen, los atributos de calidad y las restricciones que pueden existir.
10.	b, c	Una vez que hemos definido una arquitectura candidata, es momento de comunicar a las partes interesadas y planificar los escenarios de validación del diseño de la arquitectura.

**Ir a la
autoevaluación**

Autoevaluación 4		
Pregunta	Respuesta	Retroalimentación
1.	a	La arquitectura cliente-servidor es un modelo informático en el que el servidor aloja, entrega y gestiona la mayoría de los recursos y servicios que consumirá el cliente. Este tipo de arquitectura tiene una o más computadoras cliente conectadas a un servidor central a través de una red o conexión a <i>Internet</i> .
2.	b	En la arquitectura orientada a objetos los (procesos, archivos, operaciones de E / S, etc.) se representan como un objeto. Los objetos son estructuras de datos en la memoria que pueden ser manipuladas por el sistema total (<i>hardware</i> y <i>software</i>) y proporcionan una descripción de alto nivel que permite una interfaz de usuario de alto nivel.
3.	c	La arquitectura de tres capas es una arquitectura bien establecida que organiza las aplicaciones en tres niveles informáticos, lógicos y físicos: el nivel de presentación o interfaz de usuario; el nivel de aplicación, donde se procesan los datos, y el nivel de datos, donde se almacenan y gestionan los datos asociados con la aplicación.
4.	c	Es un estilo de diseño de <i>software</i> en el que los servicios se proporcionan a los otros componentes mediante componentes de la aplicación, a través de un protocolo de comunicación a través de una red. Sus principios son independientes de los proveedores y otras tecnologías.
5.	b	La arquitectura en capas (<i>tiers</i>) es utilizado en la categoría de despliegue, especialmente para distribuir o balancear la carga de las aplicaciones o de los datos.
6.	b	Un ESB, o Bus de Servicio Empresarial, es un patrón mediante el cual un componente de <i>software</i> centralizado realiza integraciones con sistemas <i>backend</i> (y traducciones de modelos de datos, conectividad profunda, enrutamiento y solicitudes) y hace que esas integraciones y traducciones estén disponibles como interfaces de servicio para su reutilización por parte de nuevos usuarios. Aplicaciones .
7.	b	La arquitectura cliente-servidor es un modelo informático en el que el servidor aloja, entrega y gestiona la mayoría de los recursos y servicios que consumirá el o los clientes.
8.	a, b, c, d	Al diseñar una aplicación o sistema, el objetivo de un arquitecto de <i>software</i> es minimizar la complejidad al separar el diseño en diferentes áreas de interés. Estos son los cuatro principios que debe seguir al diseñar su aplicación: abstracción, encapsulación, alta cohesión, bajo acoplamiento.

Autoevaluación 4		
Pregunta	Respuesta	Retroalimentación
9.	a	El componente Modelo corresponde a toda la lógica relacionada con los datos con la que trabaja el usuario. Esto puede representar los datos que se transfieren entre los componentes Vista y Controlador o cualquier otro dato relacionado con la lógica empresarial.
10.	c	El controlador actúa como una interfaz entre los componentes Modelo y Vista para procesar toda la lógica empresarial y las solicitudes entrantes, manipular datos utilizando el componente Modelo e interactuar con las Vistas para generar el resultado final.

[Ir a la autoevaluación](#)

Autoevaluación 5		
Pregunta	Respuesta	Retroalimentación
1.	d	A través de la vista se captura los requisitos funcionales de la aplicación como descomposición de elementos estructurales o abstracciones. Es usada por los desarrolladores, gerentes de ingeniería.
2.	b	A través de la vista se captura el proceso, el comportamiento, la concurrencia de tareas y el flujo de información y los aspectos no funcionales.
3.	c	Vista que se centra en la gestión de la aplicación. Es utilizada por los gerentes y desarrolladores.
4.	a	Vista que representa el diseño de implementación o la infraestructura de una aplicación. En esencia, captura el mapeo de <i>hardware</i> de los componentes o procesos de la aplicación.
5.	c	El diagrama de contexto, a veces llamado diagrama de flujo de datos de nivel 0, se usa para definir y aclarar los límites del sistema de <i>software</i> .
6.	c	El diagrama de contenedor, que corresponde al nivel 2, muestra la forma de alto nivel de la arquitectura del <i>software</i> y cómo se distribuyen las responsabilidades en ella. Similar al diagrama de despliegue del modelo 4+1.
7.	b	El diagrama de componentes muestra cómo un contenedor se compone de una serie de componentes, además, cuáles son cada uno de esos componentes, sus responsabilidades y los detalles de tecnología / implementación.
8.	d	En el modelo C4, este es un nivel de detalle opcional y, a menudo, se lo desarrolla con herramientas CASE para modelar clases y paquetes.
9.	a,b,c	Una arquitectura debe ser descrita en blanco y negro, usando papel y lápiz o herramientas CASE que permitan ingeniería directa e ingeniería inversa.
10.	b	UML es un lenguaje de modelado estandarizado semiformal, que consta de un conjunto de diagramas, desarrollados para ayudar a los desarrolladores de sistemas y <i>software</i> a especificar, visualizar, construir y documentar los artefactos de los sistemas de <i>software</i> , así como para el modelado de negocios y otros sistemas que no son de <i>software</i> .

**Ir a la
autoevaluación**

Autoevaluación 6		
Pregunta	Respuesta	Retroalimentación
1.	a	El desarrollo Web <i>front-end</i> , también conocido como desarrollo del lado del cliente, es la práctica de producir HTML, CSS y JavaScript para un sitio web o aplicación web para que un usuario pueda verlos e interactuar con ellos directamente.
2.	b	Un Bus de Servicio Empresarial (ESB, por sus siglas en inglés), es una arquitectura similar a un bus que ayuda a integrar diversas aplicaciones y servicios en una empresa. Incorpora un motor de mensajería, integración de datos y capacidades de enrutamiento, servicios web y capacidades de análisis.
3.	b	REST tiene una arquitectura más flexible, esto quiere decir que ofrece pautas flexibles y permite a los desarrolladores implementar las recomendaciones a su manera. Por lo tanto, permite diferentes formatos de mensajería, como HTML, JSON, XML y texto sin formato, mientras que SOAP solo permite XML.
4.	a	Los servicios son unidades de software que realizan una función. Se utilizan para dividir problemas complejos en una serie de problemas más simples. Los servicios también están diseñados para ser implementados por separado.
5.	c	Los servicios se pueden diseñar como servicios de entidad, que representan de manera predecible entidades de negocio, o como servicios de tareas, que representan acciones específicas que implementan algún paso en un proceso, es por ello que se definen como verbos.
6.	c	El software ESB (<i>Enterprise Service Bus</i>) es una infraestructura central compartida que actúa como un punto de conectividad para cada aplicación, dispositivo o sistema en toda la empresa.
7.	a,b,c,d	Una aplicación móvil es un tipo de software de aplicación diseñada para ejecutarse en un dispositivo móvil, como un teléfono inteligente o una tableta.
8.	a,b,c,d	Los verbos HTTP comprenden una parte importante de la restricción de interfaz uniforme y nos proporcionan la contraparte de acción del recurso basado en sustantivos. Los verbos HTTP principales o más utilizados (o métodos, como se les llama correctamente) son GET, POST, PUT y DELETE.
9.	b,d	En una implementación no distribuida, todas las funciones y capas residen en un solo servidor, excepto la funcionalidad de almacenamiento de datos.

Autoevaluación 6

Pregunta	Respuesta	Retroalimentación
10.	a,b,c	Un documento de arquitectura de software es un mapa del software, que se lo utiliza para visualizar cómo está estructurado el software. A través de la documentación se comprende los módulos y componentes del software sin tener que profundizar en el código. Es una herramienta para comunicarse con otros, desarrolladores y partes interesadas.

Ir a la
autoevaluación



5. Referencias bibliográficas

5.1. Bibliografía básica

- Adobe, S. (2016). PhoneGap. Retrieved from <https://phonegap.com/>
- Anaya Lopez, E. (2011). *Implementación de controles de seguridad en Arquitecturas Orientadas a Servicios (SOA) para servicios web*. Instituto Politécnico Nacional.
- Aperador, M. (2012). Características de los Servicios web y como consumirlos.
- Arias, J., & Fernández, N. (2009). *Servicios Web*. Madrid-España.
- Barcelona Pons, D., Ruiz Ollobarren, A., Arroyo Pinto, D., Garcia Lopez, P., París, G., & Sanchez-Artigas, M. (2019). Comparison of FaaS Orchestration Systems. <https://doi.org/10.1109/ucc-companion.2018.00049>
- Bass Len, Clements, Paul, & Kazman Rick. (2012). *Software Architecture in Practice* (Third, Vol. 2nd). <https://doi.org/10.1024/0301-1526.32.1.54>
- Bass Len, Clements, Paul, & Kazman Rick. (2013). *Software Architecture in Practice, Second Edition*.
- Bass Len, & Kazman Rick. (1999). *Architecture-Based Development*.
- Brown Simon. (2013). *Software Architecture, technical leadership and the balance with agility*. Retrieved from <http://leanpub.com/software-architecture-for-developers>
- Brown Simon. (2018). *Software Architecture for Developers* (Vol. 1).
- Brown Simon. (2019). *Software Architecture for Developers* (Vol. 2).

Camacho, Cardeso, & Nuñez G. (2004). Arquitecturas de Software. *Guía de Estudio*, 1–58.

Chase, N. (2011). IBM- DeveloperWorks.

Clements, P., Kazman, R., & Mark, K. (2001). paul clements Evaluating a Software Architecture.

Felix Bachmann, Len Bass, Chastek, G., Donohoe Patrick, & Peruzzi Fabio. (2000). *The architecture based design method*.

Garlan David. (2000). Software architecture: A Travelogue. *Proceedings of the Conference on the Future of Software Engineering, ICSE 2000*, 11. <https://doi.org/10.1145/336512.336537>

Godfrey Michael, & German Daniel. (2008). The past, present, and future of software evolution. In *Frontiers of Software Maintenance, 2008. FoSM 2008*. (pp. 129–138).

Hilliard, R. (2000). ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems. *IEEE, Http:// Standards. Ieee. Org*, 12(16–20), 2000.

Ian Sommerville. (2011). *Ingeniería de Software*.

IEEE Std 610-1990. (1990). Std 610-1990. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*.

INEN. (2014). Ingeniería del software. Calidad del producto software. Modelo de calidad (ISO/IEC 9126-1:2001, idt). Retrieved from https://www.normalizacion.gob.ec/buzon/normas/nte_inen_iso_iec_9126-1.pdf

Ivano, M. (2012). Software Architecture Modeling by Reuse, Composition and Customization. *Universita Di L'Aquila, L'Aquila, Italy, Thesis*, 1.

Jamir Antonio Avila Mojica. (2012). Arquitectura de Software: Atributos de Calidad y Perspectivas, 41.

Keeling Michael. (2017). *Design It!: From Programmer to Software Architect*. Pragmatic Bookshelf.

- Kruchten, P. B. (1995). The 4+ 1 view model of architecture. *IEEE Software*, 12(6), 42–50.
- Li, Z., Liang, P., & Avgeriou, P. (2016). *Architecture viewpoints for documenting architectural technical debt*. *Software Quality Assurance: In Large Scale and Complex Software-intensive Systems*. Elsevier Inc. <https://doi.org/10.1016/B978-0-12-802301-3.00005-3>
- Losavio, F., & Guillén, C. (2010). Comparación de métodos para evaluación de arquitecturas de software, 25, 71–87.
- Memory Jogger. (2005). The Software requirements, 370.
- Microsoft Corporation. (2009). *Microsoft Application Architecture Guide (patterns and practices) 2nd Edition*.
- Nick Rozanski, E. W. (2008). *Software Systems Architecture*.
- Perry, D., & Wolf, A. (1992). Foundations for the Study of Software Architecture. *ACM Software Engineering Notes*, 17, 40–52.
- Pressman, R. S. (2010). *Ingeniería del Software: Un enfoque práctico*.
- Richard N. Taylor, Nenad Medvidovic, E. M. D. (2012). *Software architecture : foundations, theory, and practice*.
- Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *Unified modeling language reference manual, the*. Pearson Higher Education.
- SEI. (2020). Glossary. Retrieved from <https://www.sei.cmu.edu/architecture/start/glossary/index.cfm>
- Serrano, J. M. (2007). *Arquitectura de Software, Parte 1: Introducción* (Vol. 35).
- Smith, M., Saunders, P. R., & Lyons, L. (2018). *A Practical Guide to Microservices and containers*.
- Software Engineering Institute. (2014). Defining Software Architecture. Retrieved from <http://www.sei.cmu.edu/architecture/>
- Sprint Tool Suite. (2020). Retrieved from <https://spring.io/tools>

Systems and software engineering – Architecture description ISO/IEC/IEEE 42010. (2011). Retrieved from <http://www.iso-architecture.org/42010/index.html>

W3CWorkingGroup. (2004). Web Services Architecture. Retrieved from <https://www.w3.org/TR/ws-arch/>



6. Anexos

Anexo 1. Escenario o estudio de caso

Lo que se le solicita a usted es “**Diseñar y proponer una solución documentada para el desarrollo de un Sistema de Gestión Organizacional para pymes-Ecuador (ARQSOFT)**”.

Las necesidades se han redactado de forma tal que Ud. pueda analizarlas y con ellas proveer de una solución. Antes de ello, es necesario que tenga en cuenta lo siguiente:

Necesidades: luego de una reunión entre los involucrados del proyecto, se recaban y listan las siguientes necesidades, las cuales servirán para determinar alcance, tiempo, costos, recursos, riesgos, entre otros.

- Que exista un módulo a nivel de sistema que permita la gestión de Roles, Permisos y Usuarios (RPU).
- Que existan módulos y funcionalidades a nivel de sistema que permitan la gestión de convocatorias, directiva, sesiones, socios, compras, ventas y los que considere para ARQSOFT.
- Que el módulo de RPU permita la integración e interoperabilidad con el resto de módulos y componentes de la solución que Ud. proponga.
- Que el módulo de RPU sea el CORE (similar a un sistema de seguridad central) para gestionar el acceso a todos los componentes y módulos que forman parte de ARQSOFT.
- Que se tenga en cuenta aspectos de confidencialidad para los datos e información considerados sensibles, los cuales son intercambiados entre un emisor y uno o varios receptores (cliente(s), sistema(s), base de datos).

- Que las personas que hacen uso de ARQSOFT puedan trabajar sin inconvenientes en los horarios de 08h00 a 18h30.
- Que cada 30 días en el horario de 19h00 de forma automática se generen respaldos de los datos almacenados en la base de datos local de ARQSOFT y se almacenen en un repositorio de *Cloud*.
- Los datos que son parte de las operaciones sobre el sistema deben ser alojados en un repositorio local y en la nube.
- Que las operaciones de consulta de datos que se realizan desde la interfaz gráfica de cada funcionalidad no superen los 30 segundos.
- Que las operaciones de edición de datos que se realizan desde la interfaz gráfica de cada funcionalidad no superen los 5 segundos.
- Tanto el administrador como los usuarios del sistema pueden acceder al sistema desde cualquier lugar a los módulos y funcionalidades asociados a ellos.
- Todos los módulos deben tener opciones para realizar las operaciones CRUD.
- Todos los módulos deben tener funcionalidades para emitir o generar reportes que antes de ser impresos deben ser visualizados.
- Reportes como convocatorias, sesiones, directiva, socios, compras y ventas deben tener un espacio para la firma y marca de agua en la parte central de la hoja que se imprima.
- Que los reportes que generan las funcionalidades del sistema se puedan exportar en formato pdf, xls, xml, csv.
- ARQSOFT debe operar adecuadamente con 100 usuarios concurrentes.
- Las comunicaciones externas entre servidor(es) de datos, aplicación y cliente(s) del sistema deben hacer uso de protocolos como HTTPS o HTTSPS.
- Los datos considerados confidenciales deben estar encriptados, haciendo uso de algoritmos de encriptación y desencriptación.

- Las compras y ventas de productos deberán ser registrados a través de una interface que permita interoperabilidad con los servicios del Banco XYZ.
- Para constancia de que se ha realizado una compra o venta de uno o varios productos, el sistema deberá generar una factura, la cual será electrónica, para ello usará servicios provistos por el SRI o de proveedores para este fin.
- Para la construcción de los nuevos módulos de ARQSOFT se debe seguir utilizando como entorno de desarrollo Microsoft Visual Studio.
- Que la documentación derivada del proyecto y producto se gestione en un repositorio el cual está alojado en *Cloud*.
- Por cada acción importante que se realice sobre el sistema se debe considerar el almacenamiento de datos en archivos de *logs* o *logs* en tablas de un modelo de datos, estos datos deberán contener al menos fecha, hora, módulo, funcionalidad, acción.
- Todos los usuarios que tengan acceso al sistema a través de un rol podrán hacer uso del módulo de gestión de reportes. Este módulo deberá Ud. proponerlo la forma de realizarlo para satisfacer las necesidades.

Para conocer a detalle y clasificar lo que se requiere como parte de la Información Organizacional y Gestión Financiera donde intervienen módulos como RPU, Convocatorios, Sesiones, Socios, Compras y Ventas se expone lo siguiente.

Gestión de Roles, Permisos y Usuarios (RPU)

La gestión implica la realización de operaciones CRUD (Create, Read, Update, Delete) que, a través de la interacción entre el cliente, sistema(s), base de datos y/o servicios, permitirá:

- Crear, editar, eliminar, listar los usuarios.
- Crear, editar, eliminar, listar los módulos.
 - Los módulos que forman parte de ARQSOFT son:

- Gestión de Convocatorias.
- Gestión de Sesiones.
- Gestión de Socios.
- Crear, editar, eliminar, listar los roles.
 - Los tipos de roles que soporte el sistema serán: Administrador, Técnico1, Técnico2, Técnico 3, Auditor.
 - Rol Administrador: tendrá permisos para visualizar y operar sobre todos los módulos y funcionalidades del sistema.
 - Rol Técnico1: tendrá permisos para visualizar y operar sobre los módulos y funcionalidades:
- Gestión de Socios.
- Gestión de Convocatorias.
- Gestión de Sesiones.
 - Rol Técnico2: tendrá permisos para visualizar y operar sobre el módulo y funcionalidades de:
- Gestión de Socios.
 - Rol Auditor: tendrá permisos para visualizar y operar sobre el módulo y funcionalidades de:
- Gestión de Sesiones.

A través de RPU se podrá entre otras cosas:

- Asignar y desasignar módulos a roles.
- Asignar y desasignar roles a usuarios.
- Generar reportes de roles, permisos y usuarios.
- Generar *logs* de actividades de asignación/desasignación.
- Debe analizar todos los casos posibles, por ejemplo, considerar que a un usuario se le puede configurar uno o varios roles y que dicho usuario a través de su rol podrá tener acceso a uno o varios módulos

que forman parte del sistema y que dicho módulo tendrá una o varias funcionalidades.

- Además, considere algún algoritmo o tecnología para la generación de credenciales y para acceso al sistema a través de dichas credenciales.
- Considere utilizar para la autenticación de usuarios servicios externos como por ejemplo OAuth y servicios provistos por el Registro Civil del Ecuador (<https://www.registrocivil.gob.ec/web-service-2/>).

Gestión de convocatorias

Como parte de la *Información Organizacional* de cada pyme, uno de los procesos necesarios es la Gestión de Convocatorias. Esto implica que a nivel de sistema exista un módulo que a través de operaciones CRUD permita realizar los procesos de registro, edición, consulta y eliminación de convocatorias. Para proceder a crear una Convocatoria es necesario tener previamente configurado los Socios y Directiva a través de los módulos y funcionalidades respectivas, de no tenerlos se debe realizar primero este proceso.

Para registrar una Convocatoria se solicitará datos como el tipo de organización, el cual puede ser Cooperativa, Asociación, Organismo Comunitario. Dependiendo de la selección acorde a la Organización se desplegará los tipos de asamblea, los cuales pueden ser Asamblea General, Consejo de administración y Consejo de Vigilancia. El tipo de Sesión puede ser Ordinaria, Extraordinaria e Informativa y finalmente para registrar la convocatoria se requiere especificar el lugar, fecha de convocatoria y hora de registro de la convocatoria. Considere que una convocatoria es creada para llevar a cabo una sola sesión.

Gestión de directiva

En el módulo de Gestión de Directiva se puede llevar a cabo la configuración de la Directiva que regirá por un periodo de tiempo de 1 año calendario. Para la elección de presidente, secretario y vocales que forman parte de la Directiva, se debe elegir entre los Socios que han sido previamente configurados en el sistema en el módulo de Socios. Al ser un módulo de gestión recuerde lo que implica proponer para el correcto

funcionamiento, es decir que se pueda llevar a cabo el registro, edición, consulta, eliminación.

Para elegir la Directiva debe existir una Convocatoria de Socios. El proceso de elección inicia verificando que exista el quorum respectivo (más del 50 % de socios) para esto se utiliza una funcionalidad del sistema que permite tomar y registrar asistencia de los Socios a la Convocatoria. Una vez que se ha comprobado la existencia de quorum, de todos los Socios registrados en el sistema se propone uno o varios, luego de ello todos los Socios que físicamente se encuentran en la fecha y hora de la Convocatoria proceden a registrar su voto de forma anónima y en papeles para depositarlos en un ánfora donde de luego de forma manual se realizará el conteo. El Socio que obtenga el mayor número de votaciones será el presidente y el que le sigue en votos será el secretario. Estos dos actores son los principales dentro de la Organización, luego ellos son los encargados de seleccionar a los demás miembros de la Directiva. (*El proceso de votación y conteo de votos podría automatizarse como parte de la solución*).

Gestión de sesiones

La Sesión es parte de una Convocatoria, algunas de las reglas o requisitos de la misma son los siguientes:

- Para llevar a cabo una sesión debe existir una convocatoria asociada.
- La sesión deberá iniciar en la fecha y hora indicada de la convocatoria.
- El secretario o secretaria, a través de su dispositivo móvil o un equipo portátil, debe utilizar la funcionalidad del sistema que permite tomar lista de los asistentes a la sesión.
- La funcionalidad de registrar asistencia permita visualizar todos los socios y marcar su asistencia o inasistencia.
- Debe existir como quórum máximo el 50 % de los socios para iniciar la sesión.
- El presidente y secretario que son parte de la Directiva deben estar presentes como parte del quorum.

- En caso de que no exista quorum, el estado de la sesión será posergado, por tanto, se debe editar su fecha de convocatoria y sesión y finalmente se imprimirá un acta con el estado y observaciones de la sesión.
- En caso de que se instale la sesión, el secretario debe tomar nota de la sesión a través de la funcionalidad del sistema. En esta funcionalidad se puede añadir uno o varios ítems (líneas) y puntos tratados (líneas).
- Al finalizar la sesión se debe imprimir el documento que contenga los datos de la convocatoria y sesión donde consten los ítems, acuerdos o puntos tratados en la sesión.
- Recuerde que al ser un documento formal deberá visualizarse para imprimirse para ser firmado al final de las hojas.
- Los datos de cada sesión (actual o anterior) pueden ser recuperados para realizar actividades de visualización o edición.

Gestión de socios

Los socios se consideran como un conjunto de personas que forman parte de una pyme. Por tanto, a través de este módulo se puede:

- Registrar, editar, eliminar, listar socios.
 - Identificación, nombres, género, fecha de nacimiento, fecha de ingreso, provincia, cantón, ciudad y parroquia de residencia.
- Generar reportes de socios usando diferentes filtros o criterios de búsqueda.

Anexo 2. Documento de Arquitectura de Software

Software Architecture Document

(Documento de Arquitectura de Software)

Proyecto ARQSOFT

Elaborado por: Daniel Alejandro Guamán Coronel

Versión 1.0

12/03/2021

Historial de revisiones

Versión	Descripción de versiones / cambios	Responsable	Fecha
1.0	Versión inicial	Daniel Alejandro Guamán Coronel	12/03/2021

Bloque de aprobación

Versión	Comentarios	Responsable	Fecha
1.0	En revisión		12/03/2021

Tabla de contenidos

1. Introducción
 - 1.1. Propósito
 - 1.2. Ámbito
 - 1.3. Definiciones, Acrónimos, y Abreviaturas
 - 1.4. Referencias
 - 1.5. Resumen
2. Representación arquitectónica
3. Objetivos y restricciones arquitectónicas
4. Vista de casos de uso
 - 1.6. Actores
 - 1.7. Realización de casos de uso
 - 1.7.1. Caso de uso Inicio de Sesión
 - 1.7.2. Caso de uso Solicitar Reporte
 - 1.7.3. Caso de uso Gestionar Usuarios
5. Vista Lógica
 - 1.8. Resumen
 - 1.9. Realización de diagrama de Clases
6. Vista de Procesos
 - 1.10. Resumen
 - 1.11. Realización de diagrama de secuencia
 - 1.11.1. Diagrama de secuencia inicio de sesión usando el servicio OAuth
 - 1.11.2. Diagrama de secuencia Gestionar Usuarios (Registrar un Usuario en la Base de Datos)
7. Vista de desarrollo
8. Vista de implementación

1. Introducción

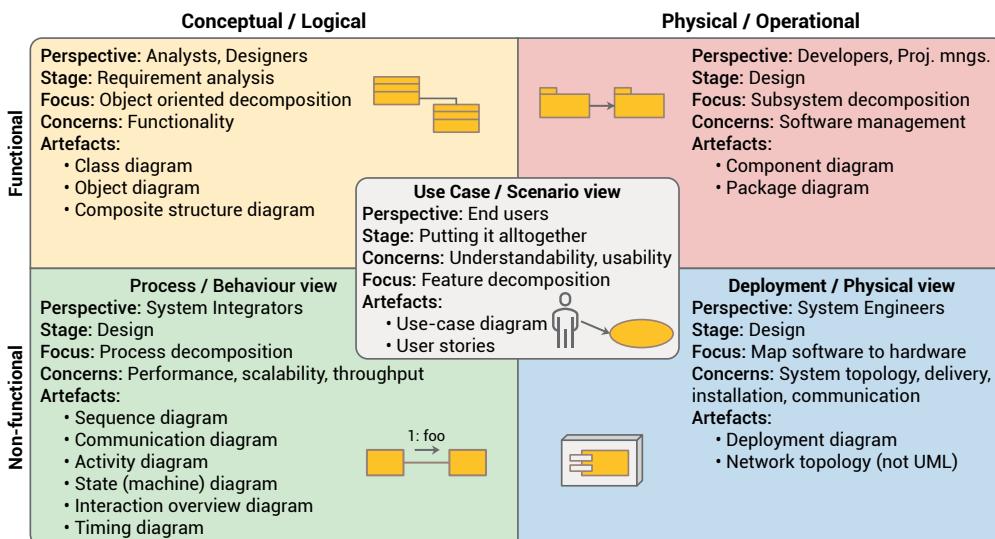
El presente documento proporciona una descripción general de alto nivel y explica la arquitectura del sistema ARQSOFT. El documento define los objetivos de la arquitectura, los casos de uso admitidos por el sistema, los estilos arquitectónicos y los componentes seleccionados. El documento proporciona una justificación para las decisiones de arquitectura y diseño tomadas desde la idea conceptual hasta su implementación.

1.11. Propósito

El Software Architecture Document (SAD, por sus siglas en inglés) proporciona una descripción general completa de la arquitectura del sistema ARQSOFT. El documento presenta una serie de vistas arquitectónicas para representar los diferentes aspectos del sistema. La estructura de este documento se basa en el modelo de arquitectura “4 + 1” propuesto por Philippe Kruchten [Kruchten].

Figura 60.

Modelo de arquitectura “4 + 1” propuesto por Philippe Kruchten [Kruchten]



El modelo “4 + 1” vistas permite a las partes interesadas apoyarse en la documentación para entender y comprender la arquitectura del software.

1.12. Ámbito

El ámbito de SAD es explicar la arquitectura del sistema ARQSOFT. Este documento describe los diversos aspectos de diseño que se consideran arquitectónicamente significativos. Estos elementos y comportamientos son fundamentales para orientar la construcción del sistema ARQSOFT y para entender el proyecto en su conjunto. Se alienta a las partes interesadas que requieran un conocimiento técnico del sistema a iniciar primeramente con la lectura de la documentación de la propuesta de proyecto y los documentos de Especificación de Requisitos de Software [SRS] desarrollados para este sistema.

1.13. Definiciones, acrónimos, y abreviaturas

- **HTTP** – Hypertext Transfer Protocol.
- **WWW** – World Wide Web.
- **GUI** – Graphical User Interface.
- **SAD** - Software Architecture Document.
- **SRS** – Software Requirements Specification.
- **UML** – Unified Modeling Language.
- **MVC** – Siglas del Patrón arquitectónico conformado por 3 partes que son: Modelo, Vista y Controlador.
- **Modelo** – Es la capa que interactúa con los datos.
- **Vista** – Contienen el código que permite visualizar los resultados en la GUI.
- **Controlador** – Contiene el código para responder a las peticiones que realiza el cliente.
- **Laravel** – Framework de código abierto para desarrollar aplicaciones y servicios web con PHP y MVC.
- **MySQL** – Es un sistema de gestión de bases de datos relacional.

1.14. Referencias

[ERS]: Especificación de Requisitos de Software (SRS, por sus siglas en inglés).

[CRC]: Tarjetas CRC (Clase, Responsabilidad, Colaboración).

[Kruchten]: The “4+1” view model of software architecture, Philippe Kruchten, November. 1995, <http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/Pbk4p1.pdf>

1.15. Resumen

Para documentar todos los aspectos de la arquitectura, el Documento de Arquitectura de Software (SAD) contiene las siguientes subsecciones.

Sección 2: describe el uso de cada vista.

Sección 3: describe los objetivos arquitectónicos y las restricciones del sistema.

Sección 4: describe las realizaciones de los casos de uso principales.

Sección 5: describe la vista de procesos principales del sistema.

Sección 6: describe la vista lógica del sistema, incluidas las definiciones de interface y operación.

Sección 7: describe vista de desarrollo que permitirá la construcción del sistema.

Sección 8: describe como el sistema se implementa a través del diagrama de despliegue.

2. Representación arquitectónica

Este documento detalla la arquitectura utilizando las vistas definidas en el modelo “4 + 1” [Kruchten]. Las vistas utilizadas para documentar el sistema ARQSOFT son:

1.16. Vista de casos de uso

Audiencia: todas las partes interesadas del sistema, incluidos los usuarios finales.

Área: describe el conjunto de escenarios y / o casos de uso que representan alguna funcionalidad central significativa del sistema. Describe los actores y los casos de uso del sistema, esta vista presenta las necesidades del usuario y se elabora más en el nivel de diseño para describir los flujos discretos y las restricciones con más detalle. Este vocabulario de dominio es independiente de cualquier modelo de procesamiento o sintaxis de representación.

Artefactos relacionados: diagrama de caso de uso, documentos de especificación de caso de uso.

1.17. Vista lógica

Audiencia: diseñadores.

Área: **requisitos funcionales**: describe el modelo de objeto del diseño. También describe las realizaciones de casos de uso más importantes y los requisitos de negocio del sistema.

Artefactos relacionados: diagrama de clases, diagrama de objetos.

1.18. Vista de procesos

Audiencia: integradores de sistemas.

Área: **requisitos no funcionales**: se ocupa de los aspectos dinámicos del sistema, explica los procesos del sistema y cómo se comunican, y se centra en el comportamiento del sistema en tiempo de ejecución.

Artefactos relacionados: diagrama de actividades, diagrama de secuencia.

1.19. Vista de implementación/Desarrollo

Audiencia: desarrolladores.

Área: **requisitos funcionales**: implementa el modelo de objeto del diseño. Implementa a nivel de sistema los requisitos de negocio.

Artefactos relacionados: diagrama de componentes, diagrama de paquetes.

1.20. Vista de despliegue

Audiencia: gerentes de implementación.

Área: **topología**: describe la asignación del *software* al *hardware* y muestra los aspectos distribuidos del sistema. Describe las estructuras de implementación potenciales, al incluir escenarios de implementación conocidos y anticipados en la arquitectura, permitimos a los implementadores hacer ciertas suposiciones sobre el rendimiento de la red, la interacción del sistema, etc.

Artefactos relacionados: modelo de implementación.

1.21. Vista de datos (*requerida cuando se trabaja con repositorio de datos relacional – no relacional*)

Audiencia: especialistas en datos, administradores de bases de datos.

Área: persistencia: describe los elementos persistentes arquitectónicamente significativos en el modelo de datos, así como cómo fluyen los datos a través del sistema.

Artefactos relacionados: modelo de datos.

1.22. Objetivos y restricciones arquitectónicas

Existen algunos requisitos clave y limitaciones del sistema que tienen una influencia significativa en la arquitectura, entre los que destacan:

1. El sistema está diseñado como una prueba de concepto para un sistema de gestión de datos e información principalmente para las pymes del Ecuador que debe ser desplegado en la web, sin embargo, puede escalar para adicionar nuevos módulos, funcionalidades o servicios que se construirán en el futuro. Por lo tanto, uno de los principales interesados en este documento y el sistema en su conjunto son los actuales y futuros arquitectos y diseñadores, no necesariamente los usuarios como suele ser el caso. Como resultado, uno de los objetivos de este documento es que pueda ser útil para los arquitectos, diseñadores y partes interesadas.
2. El sistema debe permitir comunicarse con varias API de sistemas, subsistemas o servicios de terceros. En este caso usaremos OAuth y los servicios expuestos por el Registro Civil para validar datos (<https://www.registrocivil.gob.ec/web-service-2/>). Definir cómo el sistema interactúa con estos sistemas de terceros es una preocupación principal de la arquitectura.
3. El sistema se construirá utilizando lenguajes de programación Java o PHP, y frameworks como Spring Tool Suite o Laravel. Utilizará un sistema RDBMS (Sistema de Gestión de Base de Datos Relacional) de código abierto (MySQL) para la persistencia de datos y se implementará en un servidor web Linux. Estos requisitos especiales de implementación requieren una consideración adicional en el desarrollo de la arquitectura.
4. La Especificación de Requisitos de Software [ERS], describe una serie de cambios anticipados que la aplicación podría enfrentar con el tiempo. Uno de los objetivos principales de la arquitectura del sistema es minimizar el impacto de estos cambios disminuyendo la

cantidad de código que se necesitaría modificar para implementarlos. La arquitectura busca hacer esto mediante el uso de principios de diseño como la modularización, alta cohesión, bajo acoplamiento y ocultación de información para aislar los componentes que probablemente cambiarán del resto del sistema.

5. Vista de casos de uso

El propósito de la vista de casos de uso es brindar un contexto adicional en torno al uso del sistema y las interacciones entre sus componentes. En el presente documento, cada componente se considera un actor de caso de uso. La sección 4.1 enumera los actores actuales y ofrece una breve descripción de cada uno en el contexto de uso general del sistema. En la sección 4.2, los casos de uso más comunes o generales se describen e ilustran utilizando diagramas de casos de uso de UML y diagramas de secuencia para aclarar las interacciones entre los componentes.

1.23. Actores

Actor	Descripción
Usuario	Este tipo de usuario, acorde a su rol, controlará el funcionamiento del sistema, podrá navegar por todas las Interfaces Gráficas de Usuario (GUI) del sistema para realizar operaciones sobre la aplicación.
OAuth Authentication Service	El OAuth Authentication Client, es un estándar abierto para la autorización de API, que nos permite compartir información entre sitios sin tener que compartir la identidad. Es un mecanismo utilizado a día de hoy por grandes compañías como Google, Facebook, Microsoft, Twitter, GitHub o LinkedIn, entre otras muchas.
QueryValidation Service	El objetivo principal de QueryValidation Service es actuar como enlace de comunicación entre un servicio externo (Registro civil) y nuestra aplicación.
Base de datos MySQL	Es el repositorio utilizado para almacenar los datos de la aplicación.

1.24. Realización de casos de uso

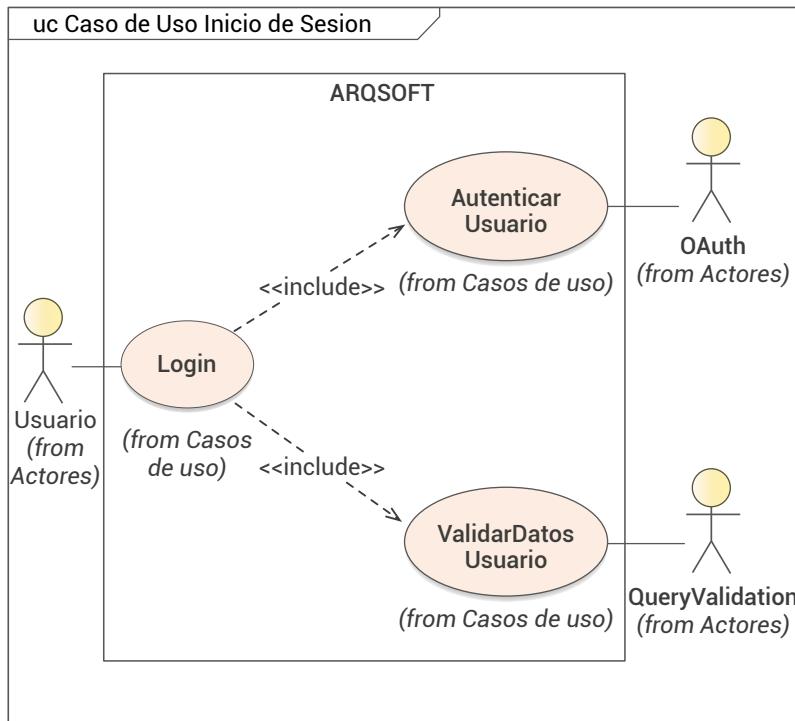
1.24.1. Caso de uso Inicio de sesión

El usuario debe ingresar las credenciales de acceso (nombre de usuario y contraseña) para realizar el proceso de *Login* o Logeo. Las credenciales se

autentican y el usuario es redirigido a la página de inicio de la aplicación. Tanto el nombre de usuario y contraseña se validan contra los servicios externos.

Figura 61.

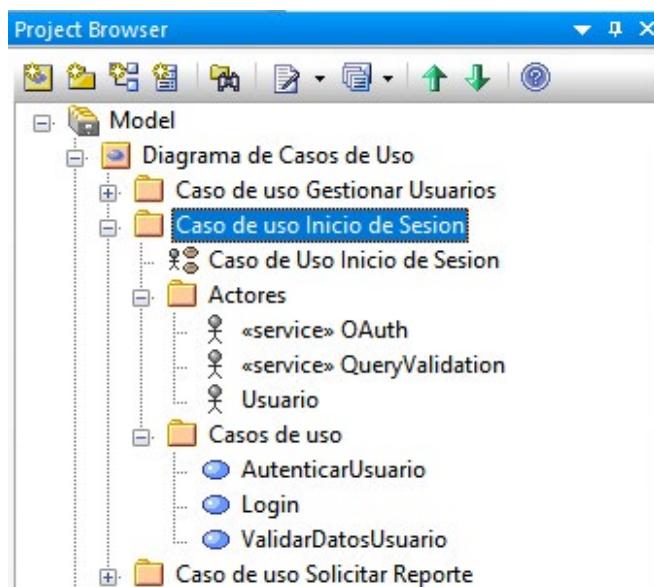
Caso de uso *Inicio de Sesión*



A continuación, se muestra la estructura de representación en la herramienta Enterprise Architect (herramienta CASE disponible en el Laboratorio Virtual UTPL).

Figura 62.

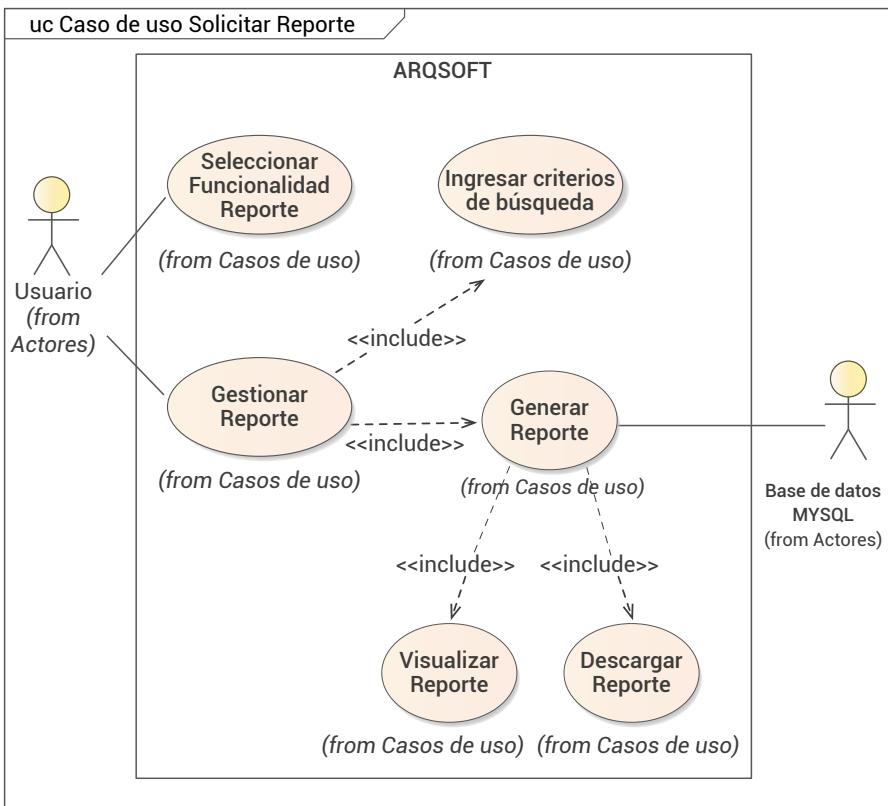
Herramienta CASE disponible en el Laboratorio Virtual UTPL



1.24.2. Caso de uso solicitar reporte

El usuario solicita a la aplicación un reporte que dependiendo del módulo o funcionalidades a las que tenga acceso podrá recuperar los datos e información. En este escenario, se ha obviado el proceso que inicia desde el acceso o Logeo del usuario a la aplicación. Un reporte puede tener varias alternativas de generación, una de ellas por ejemplo que se pueda aplicar criterios de búsqueda o filtros para obtener el reporte. El reporte una vez que es generado por la aplicación puede ser visualizado o descargado por el usuario.

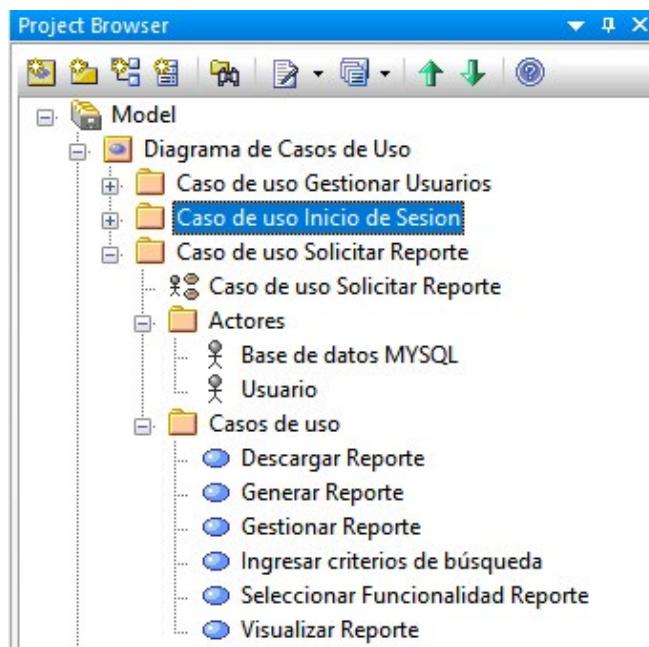
Figura 63.
Caso de uso Solicitar Reporte



A continuación, se muestra la estructura de representación en la herramienta Enterprise Architect (herramienta CASE disponible en el Laboratorio Virtual UTPL).

Figura 64.

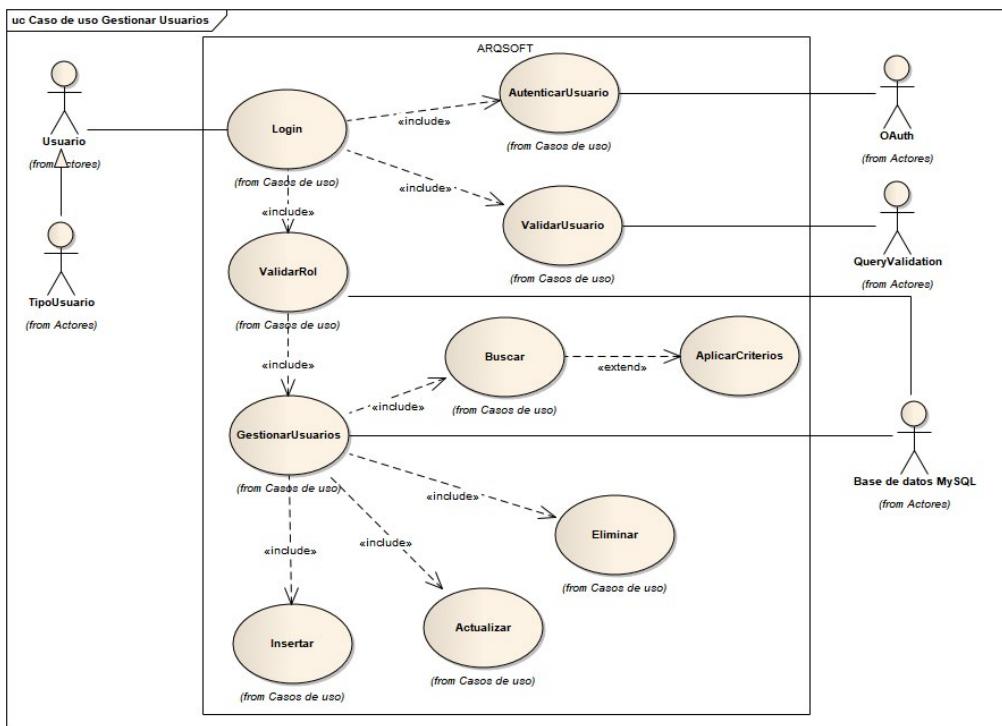
Herramienta CASE disponible en el Laboratorio Virtual UTPL - 1



1.24.3. Caso de uso gestionar usuarios

El usuario, dependiendo del rol que tenga configurado en la aplicación, puede realizar las operaciones CRUD, en este caso de usuarios. Para almacenar los datos en la base de datos.

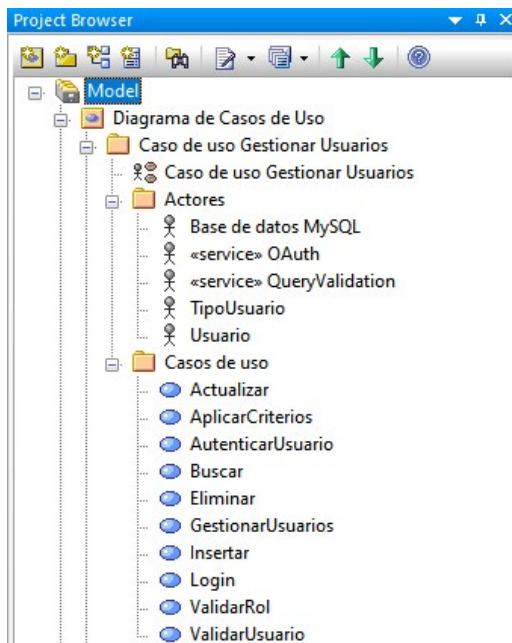
Figura 65.
Caso de uso Gestionar Usuarios



A continuación, se muestra la estructura de representación en la herramienta Enterprise Architect (herramienta CASE disponible en el Laboratorio Virtual UTPL).

Figura 66.

Herramienta CASE disponible en el Laboratorio Virtual UTPL - 2



6. Vista lógica

1.25. Resumen

El objetivo principal de la vista lógica es definir los componentes que conformarán el sistema y definir las interfaces a través de las cuales se comunicarán e interactuarán entre sí. El factor de toma de decisiones principal detrás de la definición de los componentes del sistema es la necesidad de aislar los componentes que probablemente cambiarán del resto del sistema. Al definir claramente las interfaces de estos componentes y ocultar sus implementaciones internas del resto del sistema, se puede minimizar el impacto de los cambios esperados. La sección de la Especificación de Requisitos de Software [ERS] describe los cambios que es probable que se realicen en el sistema. Un resumen de estos cambios y cómo la descomposición lógica de la arquitectura los aborda es el siguiente:

1. Cambios en la API de autenticación.

- a. La arquitectura aborda esto implementando las llamadas a la API de Autenticación en un componente de OAuthService

Client (ver tabla 6.1). El resto de la aplicación se comunicará con OAuthService solo a través de la interfaz expuesta por este componente. Por lo tanto, cualquier cambio en el sistema para hacer frente a los cambios en la API de OAuthService solo debe realizarse en la implementación interna de este componente.

2. Cambios en la API de consulta y validación.

- a. De manera similar a lo anterior, esto se soluciona implementando llamadas a la API de consulta de datos y validación de datos e información (*QueryValidation* - <https://www.registrocivil.gob.ec/consulta-de-datos-y-validacion-de-informacion/>) en un componente de cliente del Registro civil (ver tabla 6.1). Los cambios necesarios para hacer frente a los cambios en la API QueryValidation Service solo deben realizarse en la implementación interna de este componente y no en el resto del sistema.

Tabla 13.

Responsabilidades de los elementos

Elemento	Responsabilidades
OAuthService	<ul style="list-style-type: none">▪ Proporcionar una interfaz para la autenticación OAuth.▪ Manejar todas las comunicaciones con la API REST de OAuth.▪ Proporcionar una interfaz acorde al lenguaje de programación para que otros componentes la utilicen para acceder a la API de OAuth.
QueryValidation	<ul style="list-style-type: none">▪ Proporcionar una interfaz para la autenticación QueryValidation.▪ Manejar todas las comunicaciones con la API REST de QueryValidation.▪ Proporcionar una interfaz acorde al lenguaje de programación para que otros componentes la utilicen para acceder a la API de QueryValidation.
Repositorio de datos (MySQL)	<ul style="list-style-type: none">▪ Almacenar los datos en el almacén de datos MySQL.▪ Proporcionar una interfaz de consulta al repositorio de datos MySQL.

Elemento	Responsabilidades
Aplicación ARQSOFT	<ul style="list-style-type: none"> ▪ Presentar a los usuarios una interfaz de usuario basada en HTML accesible a través de un navegador web. ▪ Interactuar con otros componentes del sistema para permitir que los usuarios se autentiquen con Assembla, elijan un proyecto de Assembla para su análisis y analicen el proyecto elegido.

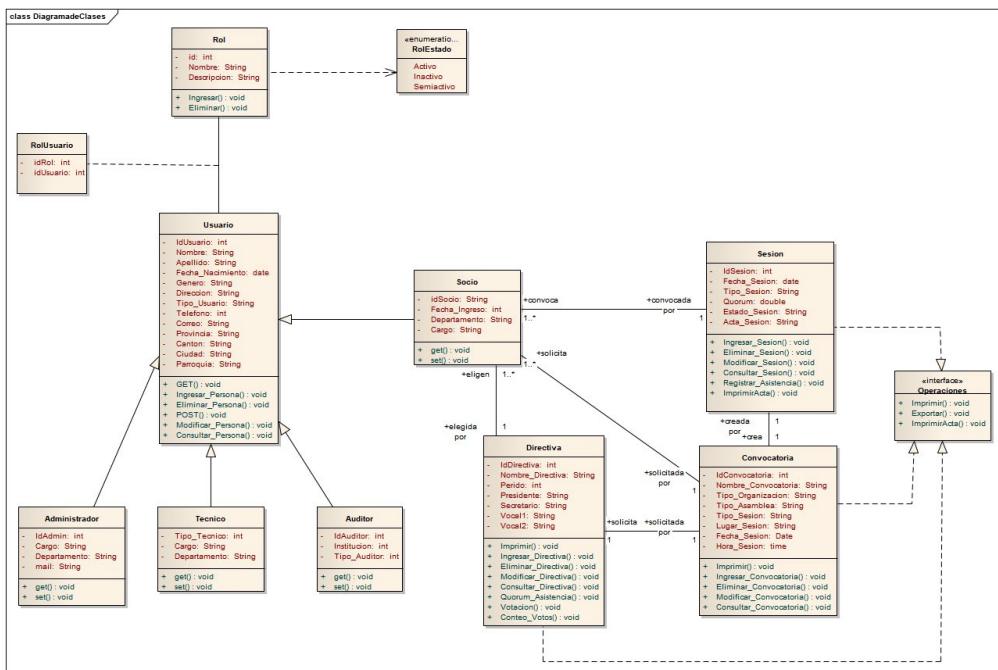
1.25.1. Realización de diagrama de clases

Recuerde que para realizar el diagrama de clases, es fundamental con base en el análisis utilizar como herramienta las **Tarjetas CRC para identificación de clases** (<https://creately.com/diagram/example/hmmj51oa2/CRC%20cards>).

Es importante que recuerde que entre las buenas prácticas para elaborar el diagrama de clases siempre nos apoyamos en las tarjetas CRC, ya que por ejemplo si sobre una clase no se puede identificar responsabilidades, es muy probable que esta no sea Clase, además que cuando a partir de la creación de objetos no podamos crear los mismos es porque debemos evaluar si realmente será una clase. Adicional a ello es importante que una clase siempre tenga responsabilidades y colaboradores, de lo contrario será una clase sin ninguna relación y se podría transformar en una Enumeración, por ejemplo.

Nombre de la Clase: Socio	
Responsabilidades	Colaboradores
Crear	SesiónConvocatoria
Actualizar	Directiva
Ser parte de una Sesión	
Ser parte de una Directiva	

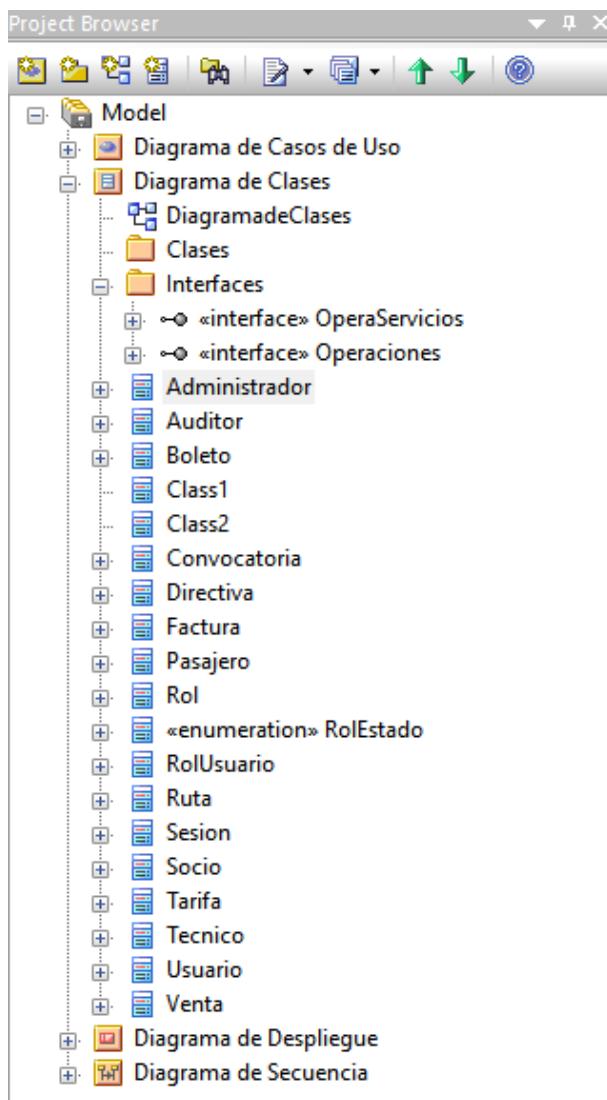
Figura 67.
Diagramas de clases



A continuación, se muestra la estructura de representación en la herramienta Enterprise Architect (herramienta CASE disponible en el Laboratorio Virtual UTPL).

Figura 68.

Herramienta CASE disponible en el Laboratorio Virtual UTPL - 3



3. Vista de procesos

1.26. Resumen

El objetivo de la vista de procesos es capturar tanto el flujo de intercambio de información entre procesos (por ejemplo, llamadas a la API REST o servicios, envío de datos a la base de datos) y la secuencia y el tiempo de estas comunicaciones entre procesos. Esto nos permite considerar cuestiones como la concurrencia y la confiabilidad.

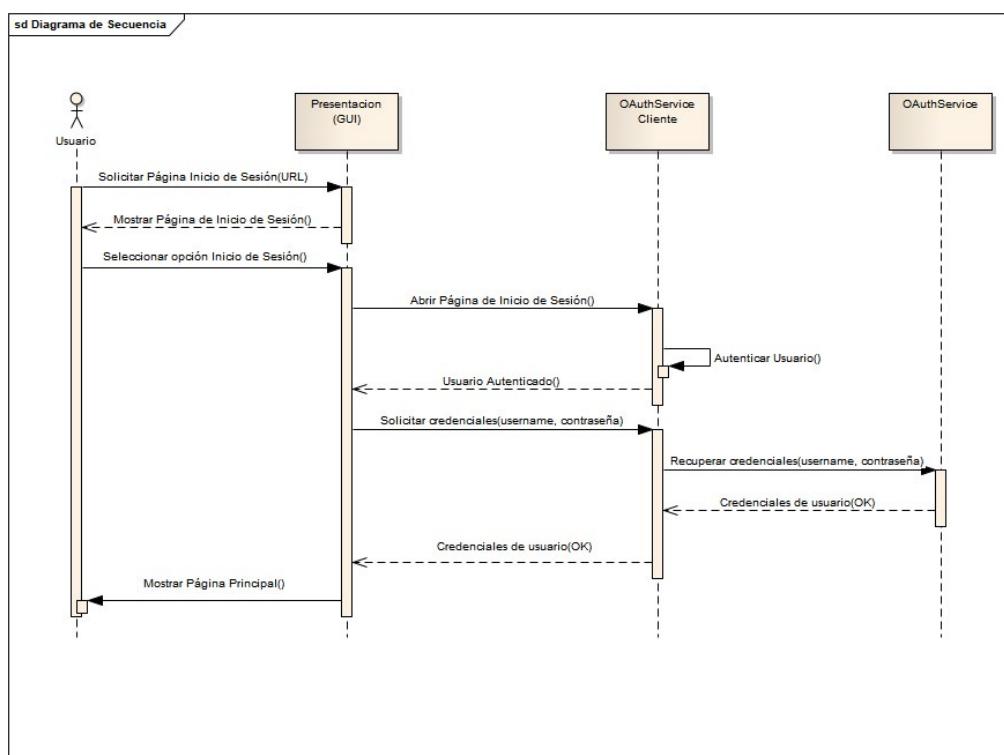
1.27. Realización de diagrama de secuencia

1.27.1. Diagrama de secuencia inicio de sesión usando el servicio OAuth

El usuario primero realiza la petición de la página para realizar la autenticación, para ello ingresa la URL con la dirección donde se encuentra implementada la aplicación. Posterior a ello y cuando se le visualiza la pantalla de inicio de sesión, ingresa las credenciales de acceso (nombre de usuario y contraseña) para realizar el proceso de Logeo. Las credenciales viajan desde el OAuthService Cliente hacia el Servicio que se encuentra alojado en *Cloud*. La emisión de la respuesta OK corresponde a que el usuario se ha validado correctamente y a partir de ello se visualizará la pantalla principal de la aplicación.

Figura 69.

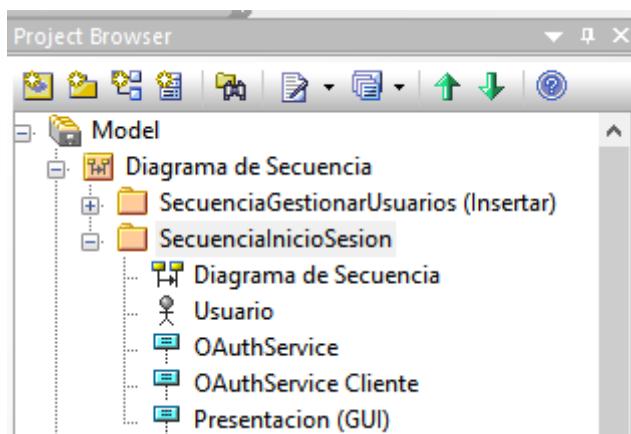
Diagrama de secuencia inicio de sesión usando el servicio OAuth



A continuación, se muestra la estructura de representación en la herramienta Enterprise Architect (herramienta CASE disponible en el Laboratorio Virtual UTPL).

Figura 70.

Herramienta CASE disponible en el Laboratorio Virtual UTPL - 4



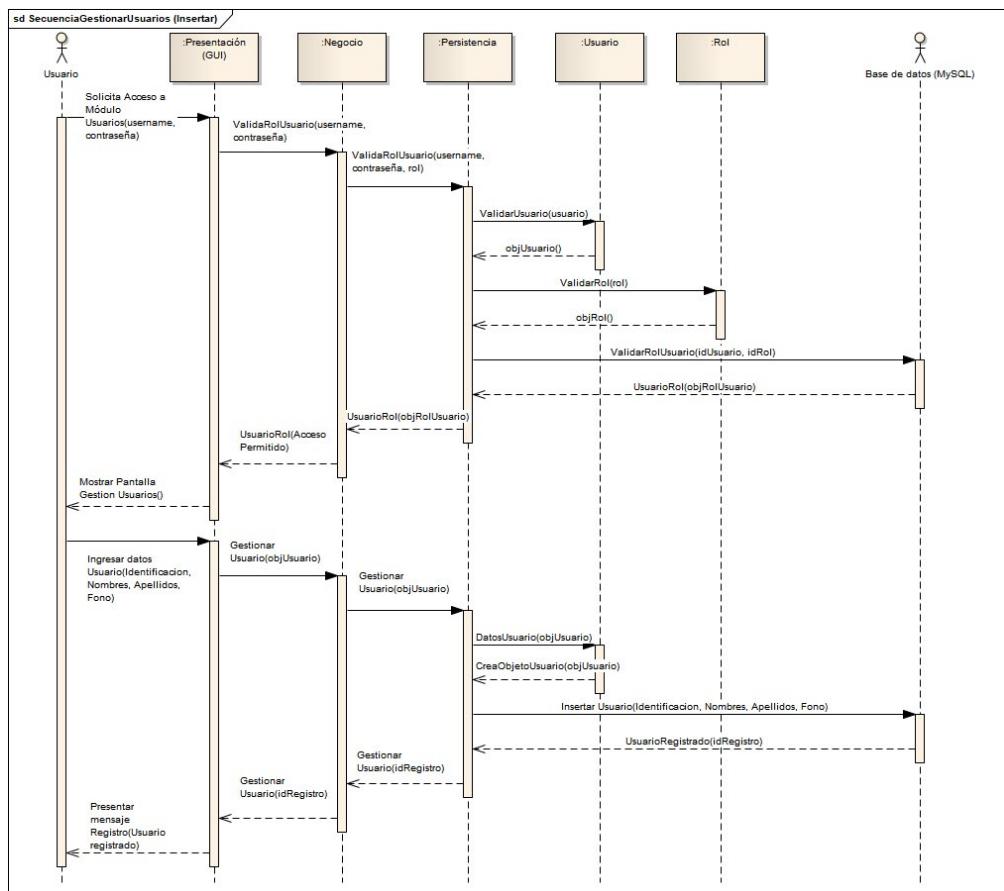
1.27.2. Diagrama de secuencia gestionar usuarios (registrar un usuario en la base de datos)

El proceso inicia verificando si el usuario a través de sus credenciales y el rol asociado tiene acceso al módulo y funcionalidades para realizar las operaciones CRUD de Usuarios. Si el usuario tiene acceso al módulo, se muestra los componentes de GUI para ingresar los datos que se soliciten para el registro.

Como se propone el uso de una arquitectura lógica en Capas (3-Layers), los datos se envían desde la capa de Presentación a la lógica de negocio donde se componen o descomponen los objetos para poder operarlos a través de la persistencia.

En la capa de persistencia lo que se realiza es un mapeo de los datos acorde a los objetos, es por ello que intervienen la clase Usuario, ya que se crea el objeto, se valida la creación del mismo y posterior a ello los datos son enviados a registrarse en la base de datos. Si el almacenamiento es exitoso, que es el escenario que se muestra en el presente diagrama, los datos se retornan desde la capa inferior hasta la superior que es la de presentación para informar al usuario que los registros se almacenaron correctamente.

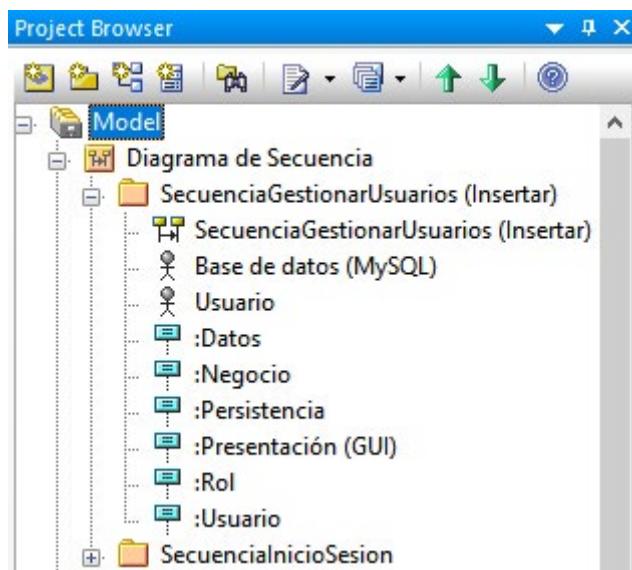
Figura 71.
Diagrama de secuencia Gestionar Usuarios



A continuación, se muestra la estructura de representación en la herramienta Enterprise Architect (herramienta CASE disponible en el Laboratorio Virtual UTPL).

Figura 72.

Herramienta CASE disponible en el Laboratorio Virtual UTPL - 5



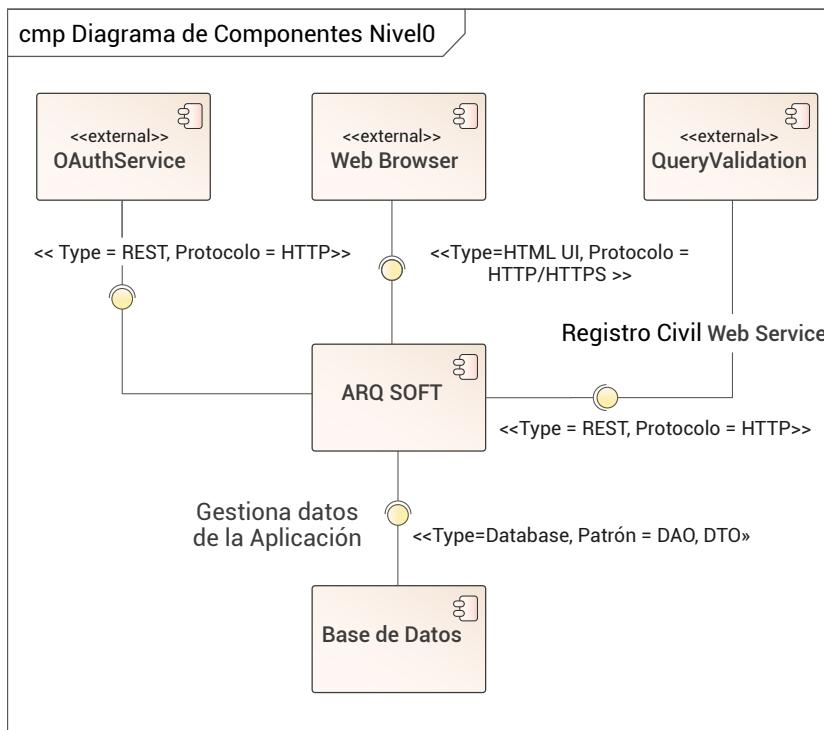
4. Vista de desarrollo

Esta sección describe la estructura general del modelo de implementación, la escritura de la aplicación acorde al patrón arquitectónico y patrones de diseño que se utilice y cualquier componente arquitectónicamente significativo. Por lo general, para gestionar el código se hace uso de herramientas tales como Github, Bitbucket.

El diagrama de componentes es importante para que los desarrolladores puedan implementarlo a nivel de aplicación. Es por ello que se puede realizar representaciones en alto nivel de abstracción (Diagrama de Componentes Nivel0) como se muestra en la figura.

Figura 73.

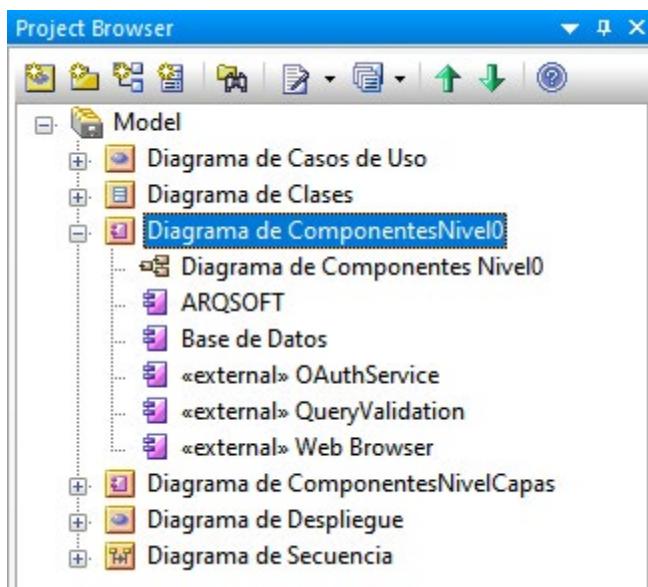
Diagrama de componentes Nivel 0



A continuación, se muestra la estructura de representación en la herramienta Enterprise Architect (herramienta CASE disponible en el Laboratorio Virtual UTPL).

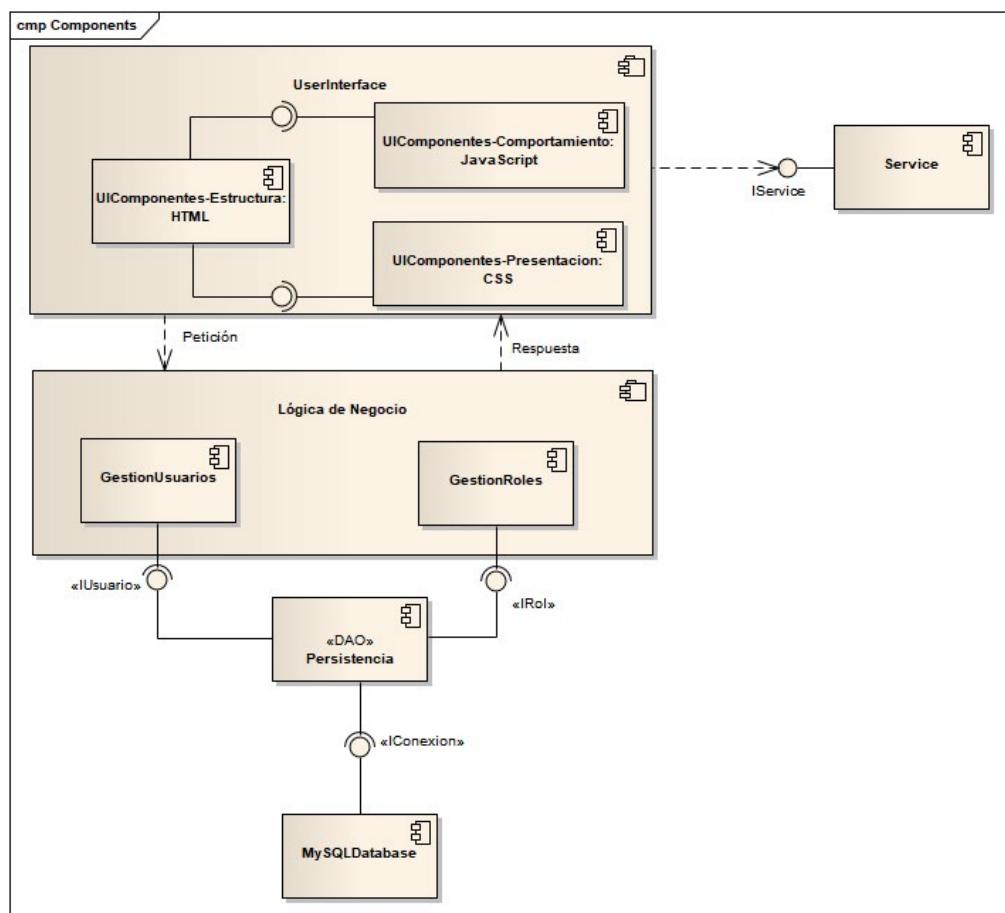
Figura 74.

Herramienta CASE disponible en el Laboratorio Virtual UTPL - 6



También se puede explorar cada componente de la aplicación para a través de interfaces provistas y requeridas, diseñar el comportamiento y la interacción que tendrán los componentes, como se puede ver en la figura donde se expone una representación en *Layers* (capas lógicas).

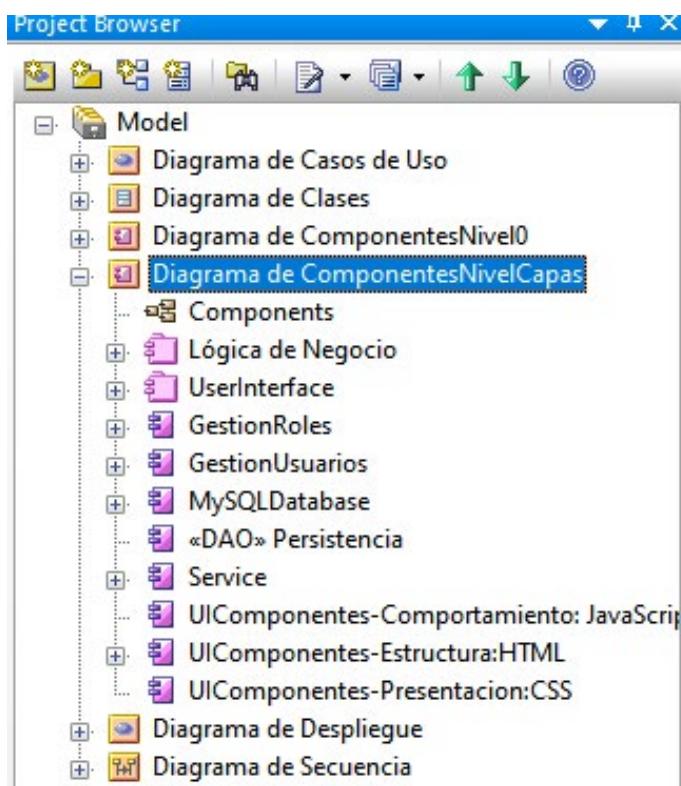
Figura 75.
Componentes



A continuación, se muestra la estructura de representación en la herramienta Enterprise Architect (herramienta CASE disponible en el Laboratorio Virtual UTPL).

Figura 76.

Herramienta CASE disponible en el Laboratorio Virtual UTPL - 7

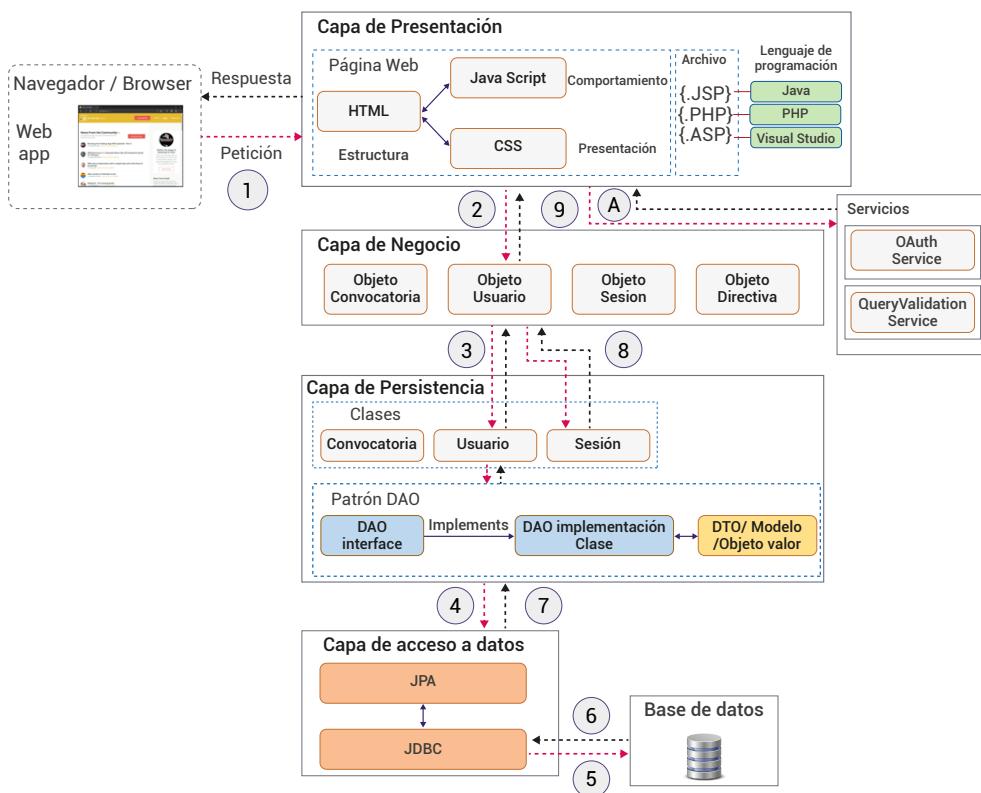


Otra forma de representar la interacción de los componentes que formarán parte de la solución y que ayudará a comprender a los desarrolladores los módulos a implementar a nivel de aplicación, es como se muestra en la figura, donde se indica que la petición dada por un usuario inicia cuando ingresa a través de teclado una petición, dicha petición en una arquitectura de *Layers* inicia en la capa de presentación en donde a través de la página HTML que está conformada por componentes de UI, y librerías que manejan el comportamiento (Java Script) y estilos como CSS permite enviar la petición a la capa de negocio.

La capa de negocio es la encargada de receptar la petición para agrupar o descomponer el objeto utilizando estructuras de datos, métodos y funciones que permitan aplicar las reglas de negocio y la lógica necesaria para enviar a procesar los datos en la base de datos. Cuando los datos se mapean en la capa de persistencia, en este caso utilizando patrones de diseño como DAO y DTO, el paso de los datos a su procesamiento en la base de datos es muy sencillo, y aunque se puede hacer uso de JPA o

Hibernate, dependiendo del lenguaje y tecnología de programación que se utilice, esto ayudará a que los datos mantengan la persistencia lo que quiere decir que si se envía a almacenar en la base de datos un conjunto de datos por ejemplo A, en la base de datos se almacene A y no por ejemplo B, C o A1. En un modelo en capas, el proceso es de dos vías, esto quiere decir que cuando se almacenen los datos en la base de datos deberá existir una respuesta que debe pasar por cada una de las capas como se visualiza en la figura, de lo contrario estaremos violando las reglas de la arquitectura ocasionando inconsistencias.

Figura 77.
Capas



5. Vista de implementación

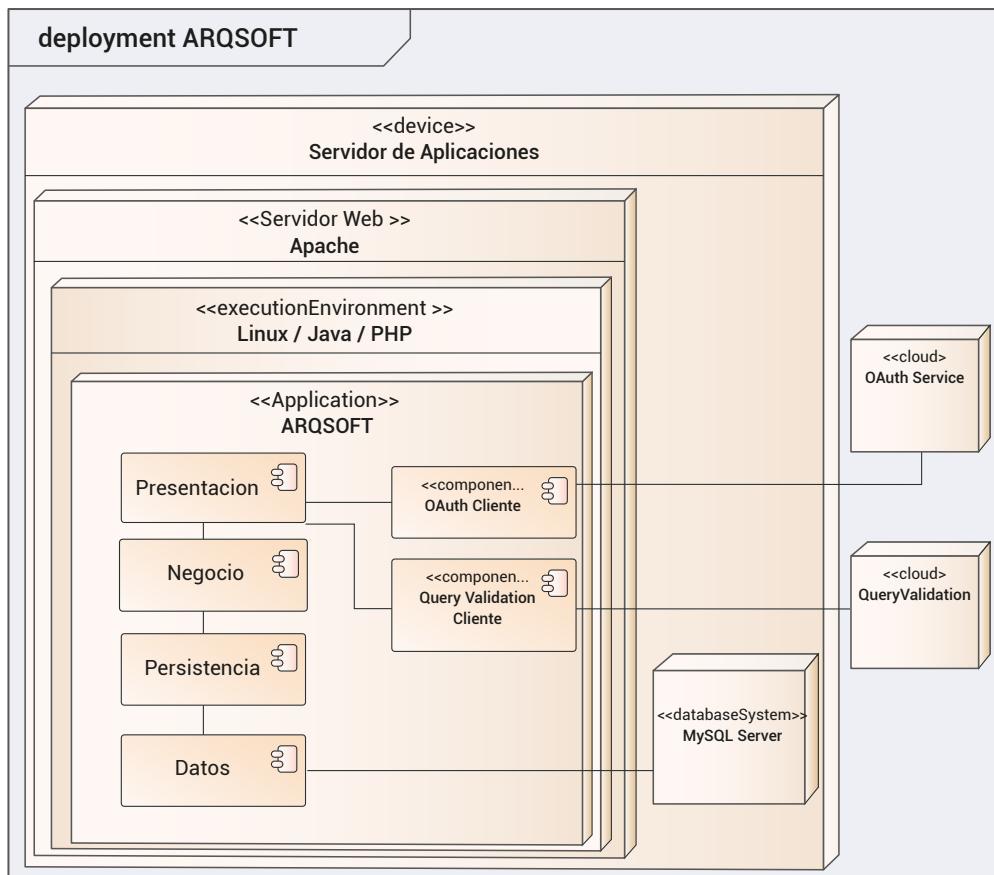
La aplicación web se alojará en un único servidor físico. Se utilizará un servidor web Apache que permita visualizar las páginas de la aplicación. Además, una instancia de MySQL Server también se alojará en el servidor físico para ayudar a la aplicación a conservar los datos. La aplicación

interactuará con API externas (OAuth Service y QueryValidation), de las cuales se desconocen los escenarios de implementación, ya que son externos a la aplicación y nuestra aplicación solo debe consumir las API que con base en una petición emiten una respuesta en formato JSON.

Los detalles de la implementación o despliegue de la aplicación se pueden ver a continuación.

Figura 78.

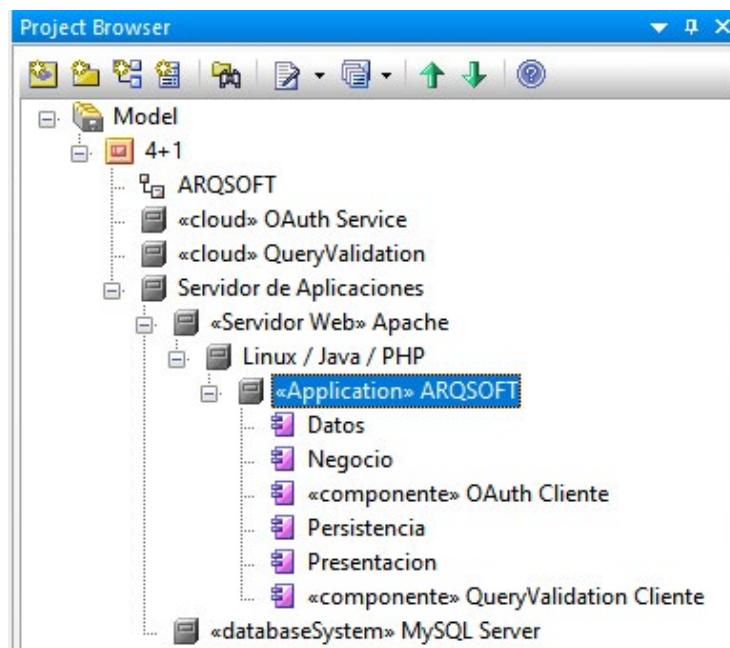
Deployment ARQ SOFT



A continuación, se muestra la estructura de representación en la herramienta Enterprise Architect (herramienta CASE disponible en el Laboratorio Virtual UTPL).

Figura 79.

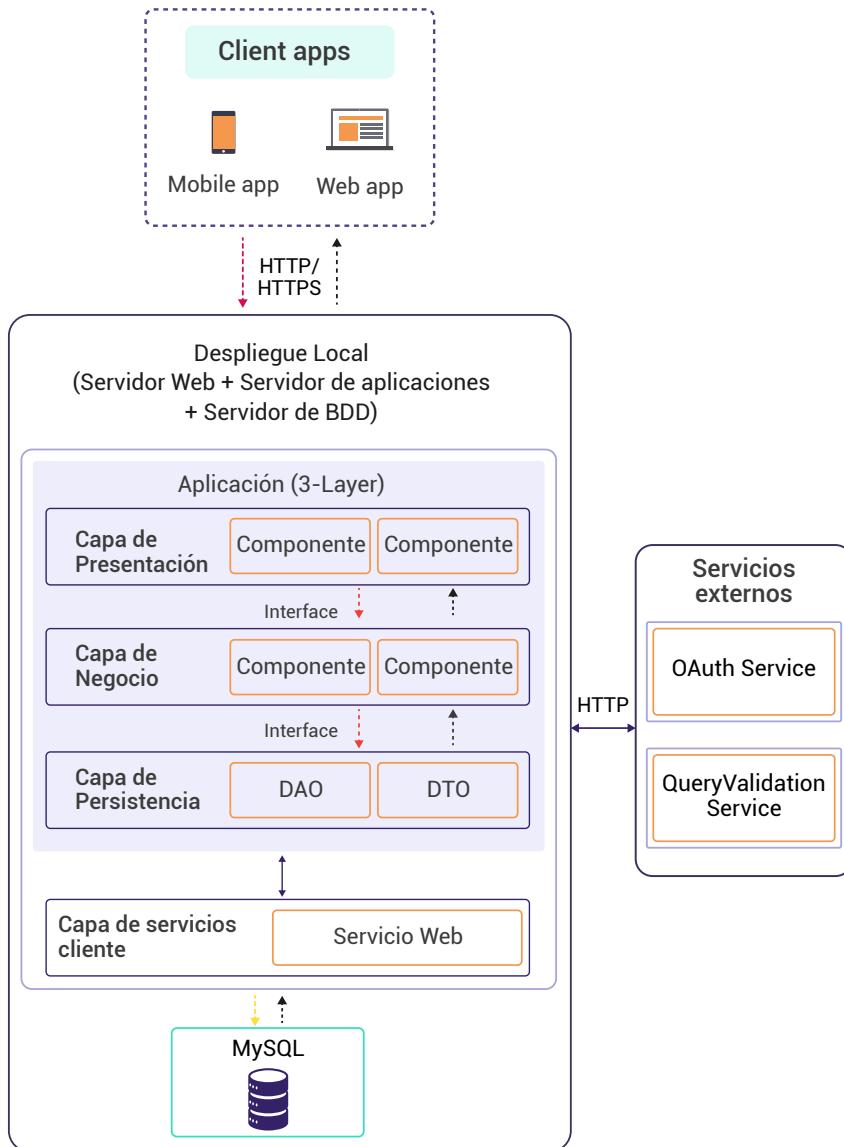
Herramienta CASE disponible en el Laboratorio Virtual UTPL - 8



Otra forma en la cual podemos exponer la propuesta de implementación o despliegue de la solución es como se muestra en la figura, con lo cual las partes interesadas pueden entender la propuesta arquitectónica.

Figura 80.

Propuesta de implementación o despliegue de la solución



Anexo 3. Documento de Especificación de Requisitos de Software

Software Requirements Specification

(Documento de Especificación de Requisitos de Software)

Proyecto ARQSOFT

Elaborado por: Daniel Alejandro Guamán Coronel

Versión 1.0

12/03/2021

Historial de revisiones

Versión	Descripción de versiones / cambios	Responsable	Fecha
1.0	Versión inicial	Daniel Alejandro Guamán Coronel	12/03/2021

Bloque de aprobación

Versión	Comentarios	Responsable	Fecha
1.0	En revisión		12/03/2021

Tabla de contenidos

1. Introducción
 - 1.12. Propósito
 - 1.13. Alcance
 - 1.14. Personal involucrado
 - 1.15. Definiciones, acrónimos y abreviaturas
 - 1.16. Referencias
 - 1.17. Resumen
2. Descripción General
 - 1.18. Perspectiva del producto
 - 1.19. Características de los Usuarios
 - 1.20. Restricciones
 - 1.21. Suposiciones y dependencias
3. Requisitos específicos
 - 1.22. Requerimientos Funcionales
 - 1.23. Requerimientos No Funcionales
4. Requisitos comunes de las Interfaces
 - 1.24. Interfaces de Usuario
 - 1.25. Interfaces de Hardware
 - 1.26. Interfaces de Software
 - 1.27. Interfaces de Comunicación
 - 1.28. Requisitos de Rendimiento
 - 1.29. Requisitos de Seguridad
 - 1.30. Requisitos de Usabilidad
 - 1.31. Requisitos de Disponibilidad
 - 1.32. Requisitos de Mantenibilidad
 - 1.33. Requisitos de Portabilidad

1. Introducción

El presente documento se utiliza para la Especificación de Requisitos Software (ERS) del Sistema de Gestión Organizacional para pymes - ARQSOFT. Esta especificación se ha estructurado basándose en las directrices dadas por el estándar IEEE Práctica Recomendada para Especificaciones Funcionales de Requisitos de Software ANSI/IEEE 830, 1998.

1.28. Propósito

El propósito del documento es definir las especificaciones funcionales, no funcionales para el desarrollo del sistema que permita automatizar los procesos de Gestión Organizacional (Gestión de Roles, Permisos y Usuarios (RPU), Gestión de Convocatorias, Gestión de Directiva, Gestionar de Sesiones, Gestión de Socios).

1.29. Alcance

Esta especificación de requisitos está dirigido a los arquitectos, desarrolladores y demás partes interesadas que desean conocer del Sistema ARQSOFT, el cual tiene por objetivo principal automatizar los procesos de Gestión Organizacional (Gestión de Roles, Permisos y Usuarios (RPU), Gestión de Convocatorias, Gestión de Directiva, Gestionar de Sesiones, Gestión de Socios).

1.30. Personal involucrado

Nombre	Juan Pérez
Rol	Responsable del Instituto de Economía Popular y Solidaria
Categoría Profesional	Gerente General
Responsabilidad	Coordinar las visitas a las pymes de la Región 7 Loja
Información de contacto	daguaman@utpl.edu.ec

Nombre	Andrea Villegas
Rol	Jefe de Administración y Finanzas
Categoría Profesional	Contadora de la pyme Ecolacteos de Loja
Responsabilidad	Contabilidad General de la Empresa
Información de contacto	abc@utpl.edu.ec

Nombre	Ana Carrillo
Rol	Jefe de Operaciones
Categoría Profesional	Recursos Humanos
Responsabilidad	Organizar el personal de logística, asesorías y servicios
Información de contacto	abc@utpl.edu.ec

Nombre	Roberto Méndez
Rol	Presidente de socios de la pyme
Categoría Profesional	Ingeniero Comercial
Responsabilidad	Organizar a los socios de la pyme
Información de contacto	abc@utpl.edu.ec

Nombre	Daniel Alejandro Guamán Coronel
Rol	Analista
Categoría Profesional	Analista de Sistemas
Responsabilidad	Ánalisis y diseño del Sistema
Información de contacto	daguaman@utpl.edu.ec

1.31. Definiciones, acrónimos y abreviaturas

Nombre	Descripción
Usuario	Persona que usará el sistema para gestionar procesos
ARQSOFT	Sistema de Gestión Organizacional para pymes
ERS	Especificación de Requisitos de Software
RF	Requerimiento Funcional
RNF	Requerimiento No Funcional
RPU	Roles, Permisos y Usuarios
CRUD	Create, Read, Update, Delete
Gestión	Término que se usa para especificar que el sistema permitirá realizar operaciones CRUD

1.32. Referencias

Título del Documento	Referencia
Estándar IEEE 830 – 1998	IEEE [IEEE Computer Society. Software Engineering Standards Committee, & IEEE-SA Standards Board. (1998). <i>ieee recommended practice for software requirements specifications</i> (Vol. 830, No. 1998). IEEE.]

1.33. Resumen

Este documento consta de tres secciones. En la primera se muestra la introducción y se proporciona una visión general de las especificaciones de recursos del sistema. En la segunda sección se realiza una descripción general del sistema, con el fin de conocer las principales funciones que este debe realizar, los datos asociados y los factores, restricciones, supuestos y dependencias que afectan al desarrollo del sistema, sin entrar en excesivos detalles. En la tercera sección se especifican los requisitos que debe satisfacer el sistema.

2. Descripción general

1.34. Perspectiva del producto

El sistema ARQSOFT, acorde a las restricciones arquitectónicas y tecnológicas iniciales, será un producto desarrollado en lenguaje de programación Java o PHP, haciendo uso de Frameworks como Spring Tool Suite o Laravel respectivamente. El sistema debe contener como módulos RPU, Convocatorias, Sesiones Socios. La gestión de todos los módulos implica la realización de operaciones CRUD a través de la interacción entre el cliente y sistema, base de datos o servicios.

1.35. Características de los Usuarios

Tipo de Usuario	ADMINISTRADOR
Formación	Ingeniero en TI o Afines.
Actividades	Tendrá permisos para visualizar y operar sobre todos los módulos y funcionalidades del sistema.

Tipo de Usuario	TÉCNICO 1
Formación	Ingeniero en TI o Afines.

Tipo de Usuario	TÉCNICO 1
Actividades	Tendrá permisos para visualizar y operar sobre los módulos y funcionalidades de: Gestión de Socios, Gestión de Convocatorias, Gestión de Sesiones.

Tipo de Usuario	TÉCNICO 2
Formación	Ingeniero en TI o Afines.
Actividades	Tendrá permisos para visualizar y operar sobre los módulos y funcionalidades de: Gestión de Socios.

1.36. Restricciones

1. El sistema está diseñado como una prueba de concepto para un sistema de gestión de datos e información principalmente para las pymes del Ecuador que debe ser desplegado en la web, sin embargo, puede escalar para adicionar nuevos módulos, funcionalidades o servicios que se construirán en el futuro. Por lo tanto, uno de los principales interesados en este documento y el sistema en su conjunto son los actuales y futuros arquitectos y diseñadores, no necesariamente los usuarios como suele ser el caso. Como resultado, uno de los objetivos de este documento es que pueda ser útil para los arquitectos, diseñadores y partes interesadas.
2. El sistema debe permitir comunicarse con varias API de sistemas, subsistemas o servicios de terceros. En este caso usaremos OAuth y los servicios expuestos por el Registro civil para validar datos (<https://www.registrocivil.gob.ec/web-service-2/>). Definir cómo el sistema interactúa con estos sistemas de terceros es una preocupación principal de la arquitectura.
3. El sistema se construirá utilizando lenguajes de programación Java o PHP y frameworks como Spring Tool Suite o Laravel. Utilizará un sistema RDBMS (Sistema de Gestión de Base de Datos Relacional) de código abierto (MySQL) para la persistencia de datos y se implementará en un servidor web Linux. Estos requisitos especiales de implementación requieren una consideración adicional en el desarrollo de la arquitectura.
4. El cliente requiere ver entregas continuas y funcionales de los prototipos de la solución para verificar su avance en el diseño y

construcción, por tanto, considere adoptar una metodología de desarrollo que satisfaga esta necesidad.

1.37. Suposiciones y dependencias

El equipo de arquitectos y desarrolladores de software tienen experiencia en el uso de tecnologías actuales. En los equipos computacionales en el que se implemente el sistema debe cumplir con las características antes mencionadas para el correcto funcionamiento del mismo. Se asume que los requisitos descritos en el documento se aceptan y validan por las partes interesadas. Además, se considera que existe el presupuesto para la realización del producto de software.

5. Requisitos específicos

1.38. Requerimientos funcionales

Identificación de requerimiento	RF01
Nombre de requerimiento	Autenticación y autorización de usuarios.
Características	Los usuarios deberán autenticarse para acceder a cualquier módulo o funcionalidad del sistema ARQSOFT.
Descripción del requerimiento	<p>La aplicación debe verificar a través de las credenciales de acceso (usuario y contraseña) si una persona está registrada en la base de datos de la aplicación y está autorizada para el acceso. Para ello el usuario debe:</p> <p>Ingresar el nombre de usuario.</p> <p>Ingresar la contraseña.</p> <p>La aplicación deberá validar las credenciales de acceso.</p> <p>Si las credenciales de acceso son correctas y el usuario se encuentra registrado en la base de datos, se autoriza su acceso y podrá visualizar la pantalla principal.</p>
Requerimiento No Funcional	
Prioridad del requerimiento: Alta	

Identificación de requerimiento	RF02
Nombre de requerimiento	Autenticación de Usuario contra servicio OAuth.

Identificación de requerimiento	RF02
Características	Los usuarios deberán autenticarse a través de un servicio OAuth.
Descripción del requerimiento	<p>La aplicación debe verificar a través de las credenciales de acceso (correo y contraseña) si una persona está autorizada para el acceso. Para ello el usuario debe:</p> <p>Ingresar la dirección de correo electrónico.</p> <p>Ingresar la contraseña del correo electrónico.</p> <p>Las credenciales de acceso se validarán contra el servicio <i>Authentication Client</i>. Si el acceso es correcto, el servicio emitirá un código HTTP 200 que significa autenticación correcta, caso contrario el código 400 que significa autenticación incorrecta.</p>
Requerimiento No Funcional	
Prioridad del requerimiento:	Alta

Identificación de requerimiento	RF03
Nombre de requerimiento	Gestión de Usuarios.
Características	El usuario con rol de Administrador podrá dar de alta o baja a los usuarios que utilizarán la aplicación.
Descripción del requerimiento	<p>La aplicación debe permitir, a través de una GUI, realizar las operaciones CRUD de los usuarios, para ello el usuario con rol Administrador ingresará datos como: Tipo de Identificación, Número de identificación, Nombres, Apellidos, correo electrónico, usuario y contraseña. Algunas consideraciones de los campos usuario y contraseña son:</p> <p>El texto del campo de usuario debe tener una longitud máxima de 8 caracteres y debe permitir letras en mayúsculas y minúscula.</p> <p>El texto del campo de contraseña debe contener caracteres alfa numéricos y su longitud.</p>
Requerimiento No Funcional	
Prioridad del requerimiento:	Alta

Identificación de requerimiento	RF04
Nombre de requerimiento	Gestión de Roles.

Identificación de requerimiento	RF04
Características	<p>El usuario con rol de Administrador podrá dar de alta o baja a los roles que servirán para asignar los permisos a los usuarios.</p>
Descripción del requerimiento	<p>La aplicación debe permitir registrar a los roles que se les podrá asignar a los usuarios a través de la GUI. Para registrar los roles se les solicitará:</p> <ul style="list-style-type: none"> Nombre del rol (Administrador, Técnico1, Técnico 2, Auditor). Descripción del rol. Estado del rol (activo, inactivo). Asignar y desasignar roles a usuarios. Generar reportes de roles, permisos y usuarios. Generar <i>logs</i> de actividades de asignación / desasignación.
Requerimiento No Funcional	
Prioridad del requerimiento: Alta	

Identificación de requerimiento	RF05
Nombre de requerimiento	Gestión de Roles, Permisos y Usuarios.
Características	El usuario con rol de Administrador podrá asignar/ desasignar los roles y permisos que se les da a los usuarios.

Identificación de requerimiento	RF05
Descripción del requerimiento	<p>La aplicación debe permitir asignar/desasignar los usuarios a uno o varios roles con los cuales un usuario podrá acceder a la aplicación. Los roles que debe tener registrado el sistema previamente son:</p> <p>Rol Administrador: se le otorga permisos para visualizar y operar sobre todos los módulos y funcionalidades de la aplicación.</p> <p>Rol Técnico1: se le otorga permisos para visualizar y operar sobre los módulos y funcionalidades para Gestión de Socios, Gestión de Convocatorias, Gestión de Sesiones.Rol Técnico2: se le otorga permisos para visualizar y operar sobre el módulo y funcionalidades para la Gestión de Socios.Rol Auditor: se le otorga permisos para visualizar y operar sobre el módulo y funcionalidades para la Gestión de Sesiones.</p> <p>La aplicación debe generar <i>logs</i> de actividades de asignación / desasignación de usuarios a roles.</p>
Requerimiento No Funcional	Prioridad del requerimiento: Alta

Identificación de requerimiento	RF06
Nombre de requerimiento	Consultar datos desde QueryValidationService.
Características	El usuario con rol de Administrador podrá consultar los datos de los usuarios que utilizarán la aplicación.

Identificación de requerimiento	RF06
Descripción del requerimiento	<p>La aplicación debe consultar datos personales registrados en el Sistema del Registro civil, para ello a través de la GUI se le solicitará que el usuario con rol de Administrador cuando vaya a registrar una persona ingrese como dato el número de identificación de la persona.</p> <p>Si la respuesta es correcta a la petición, el servicio web retornará en un archivo JSON los datos personales de la persona (identificación, nombres, apellidos, género, fecha de nacimiento, estado civil, entre otros que constan en el documento de identificación).</p> <p>Si no se obtiene respuesta a la petición desde el servicio web, este emitirá un mensaje de error 400.</p> <p>El error 400 debe ser interpretado por la aplicación para informar a través de la GUI al usuario.</p>
Requerimiento No Funcional	
Prioridad del requerimiento: Alta	

Identificación de requerimiento	RF07
Nombre de requerimiento	Gestión de Convocatorias.
Características	El usuario que tiene permisos para acceder al módulo de convocatorias puede registrar las solicitudes de las convocatorias.

Identificación de requerimiento	RF07
Descripción del requerimiento	<p>La aplicación debe permitir, a través de una GUI, realizar las operaciones CRUD de las convocatorias.</p> <p>Para crear una Convocatoria es necesario tener previamente configurado los Socios y Directiva a través de los módulos y funcionalidades respectivas.</p> <p>Los datos para crear una Convocatoria son:</p> <ul style="list-style-type: none"> Tipo de Organización, que puede ser: Cooperativa. Asociación. Organismo Comunitario. Tipo de Asamblea, que puede ser: Asamblea General. Consejo de Administración. Consejo de Vigilancia. Tipo de Sesión, puede ser: Ordinaria. Extraordinaria. Informativa. Lugar. Fecha de convocatoria. Hora de registro. <p>Una convocatoria es creada para llevar a cabo una sola sesión.</p>
Requerimiento No Funcional	
Prioridad del requerimiento:	Alta

Identificación de requerimiento	RF08
Nombre de requerimiento	Gestión de Directiva.
Características	El usuario que tiene permisos para acceder al módulo para registrar la directiva.
Descripción del requerimiento	<p>La aplicación debe permitir, a través de una GUI, realizar las operaciones CRUD de la Directiva para llevar a cabo el proceso de la sesión y convocatorias. Para ello se debe realizar los siguientes pasos:</p> <ul style="list-style-type: none"> La Directiva regirá por un periodo de tiempo de 1 año calendario. Se debe elegir presidente, secretario y vocales que forman parte de la Directiva. La Directiva debe ser elegida entre los Socios que han sido previamente registrados en el módulo de Socios.

Identificación de requerimiento	RF08
Requerimiento No Funcional	
Prioridad del requerimiento:	Alta
Identificación de requerimiento	RF09
Nombre de requerimiento	Gestión de Sesiones.
Características	El usuario que tiene permisos para acceder al módulo para registrar las sesiones.
Descripción del requerimiento	<p>La aplicación debe permitir, a través de una GUI, realizar las operaciones CRUD de las Sesiones, para llevar a cabo el proceso de la sesión se debe realizar los siguientes pasos:</p> <p>Para llevar a cabo una sesión debe existir una convocatoria asociada.</p> <p>El usuario debe utilizar la funcionalidad del sistema para registrar asistencia lista de los asistentes a la sesión.</p> <p>Para dar inicio a la sesión, debe validar la existencia de al menos el 50 % de los socios (quorum).</p> <p>En caso de que no exista quorum, el estado de la sesión se cambia a postergado.</p> <p>Si la sesión se posterga, se debe editar la fecha de convocatoria y sesión. Finalmente, se imprimirá un acta con el estado y observaciones de la sesión.</p> <p>Si la sesión se instala, el secretario debe tomar nota de la sesión a través de la funcionalidad de la aplicación. En esta funcionalidad se puede añadir uno a varios ítems que corresponden a los puntos tratados.</p> <p>Al finalizar la sesión se debe imprimir el documento que contenga los datos de la convocatoria y sesión donde consten los ítems, acuerdos o puntos tratados en la sesión. Los datos de cada sesión (actual o anterior) pueden ser recuperados para realizar actividades de visualización o edición.</p>
Requerimiento No Funcional	
Prioridad del requerimiento:	Alta

Identificación de requerimiento	RF10
Nombre de requerimiento	Gestión de Socios.
Características	En este módulo se puede llevar a cabo la gestión de los Socios que son un conjunto de personas que forman parte de una pyme.
Descripción del requerimiento	La aplicación debe permitir a través de una GUI, realizar las operaciones CRUD de los Socios, para llevar a cabo el proceso de la sesión se debe realizar solicitar los datos como identificación, nombres, género, fecha de nacimiento, fecha de ingreso, provincia, cantón, ciudad y parroquia de residencia de cada socio. La aplicación debe permitir generar reportes de socios usando diferentes filtros o criterios de búsqueda.
Requerimiento No Funcional	
Prioridad del requerimiento:	Alta

1.39. Requerimientos No Funcionales

Identificación de requerimiento	RNF01
Descripción del requerimiento	La aplicación debe hacer uso de componentes de Interfaz Gráfica de Usuario (GUI) que permitan que el usuario interactúe de forma intuitiva y sencilla.
Atributo de calidad asociado	Usabilidad
Prioridad del requerimiento:	Alta

Identificación de requerimiento	RNF02
Descripción del requerimiento	El sistema debe tener permitir intercambiar datos con otros sistemas, subsistemas, servicios.
Atributo de calidad asociado	Interoperabilidad
Prioridad del requerimiento:	Alta

Identificación de requerimiento	RNF03
Descripción del requerimiento	Los módulos de la aplicación usados en la autenticación y autorización deben permitir intercambiar datos con servicios externos.
Atributo de calidad asociado	Interoperabilidad
Prioridad del requerimiento:	Alta

Identificación de requerimiento	RNF04
Nombre de requerimiento	La aplicación debe garantizar que los datos de las transacciones realizadas se almacenen en un repositorio o en archivos.
Atributo de calidad asociado	Seguridad
Prioridad del requerimiento: Alta	

Identificación de requerimiento	RNF05
Nombre de requerimiento	Por cada acción realizada sobre los módulos o funcionalidades de la aplicación, se debe considerar el almacenamiento de datos en archivos de <i>logs</i> o en tablas de un modelo de datos. Estos datos deberán contener al menos fecha, hora, módulo, funcionalidad, acción y que se pueda llevar procesos de auditoría.
Atributo de calidad asociado	Seguridad
Prioridad del requerimiento: Alta	

Identificación de requerimiento	RNF06
Descripción del requerimiento	Permitir el correcto funcionamiento de la aplicación con 5, 10 o más usuarios. La aplicación debe funcionar correctamente con un número de hasta 10000 usuarios concurrentes.
Atributo de calidad asociado	Escalabilidad
Prioridad del requerimiento: Alta	

6. Requisitos comunes de las interfaces

1.40. Interfaces de usuario

La Interfaz Gráfica de Usuario (GUI, por sus siglas en inglés) dependiendo del lenguaje de programación, estará compuesta por páginas HTML (Java Script, CSS, AJAX) que contienen elementos gráficos como botones, listas y campos de textos. Ésta deberá ser construida específicamente para el sistema propuesto y, será visualizada a través de un navegador (Browser – Mozilla Firefox, Chrome, Safari u otro).

1.41. Interfaces de Hardware

Las interfaces de *Hardware* permitirán el desarrollo e implementación local de la aplicación, para ello se propone utilizar:

- 4 equipos computacionales portátiles que contiene *software* especializado
- 1 servidor con las siguientes características:
 - Marca: Dell.
 - Modelo: Poweredge 1950 III.
 - Procesador: Intel Xeon cuádruple.
 - Memoria: RAM 4 GB.
 - Disco Duro: 250 GB.
 - Sistema Operativo: Ubuntu 9.04 Server.
 - Servidor Web: Apache 2.0.3.
 - Motor de bases de datos: MySQL 5.1 Community Server.
 - Administrador de MySQL: phpMyAdmin 3.1.3.
 - Lenguaje de scripting: PHP 5.2.9.

1.42. Interfaces de *software*

Las interfaces de *software* permitirán la construcción e implementación de la solución, para ello se propone utilizar:

- Sistema Operativo: Linux.
- Spring Tool Suite.
- Larabel Framework.
- Navegador o Browser: Mozilla Firefox o Chrome.

1.43. Interfaces de comunicación

Las comunicaciones externas entre servidor(es) de datos, aplicación y cliente(s) del sistema deben hacer utilizando protocolos como TCP/IP, HTTPS o HTTSPS.

Algunas consideraciones relacionadas con los atributos de calidad que la aplicación debe contener se exponen a continuación.

1.44. Requisitos de rendimiento

Garantizar que las peticiones y respuestas entre componentes lógicos de la aplicación y su comunicación con la base de datos u otros servicios no afecte el desempeño de la aplicación, de la base de datos, y el tráfico de red.

1.45. Requisitos de seguridad

Garantizar la seguridad y confidencialidad de los datos que se generan desde la aplicación o que se transmiten entre la aplicación y otros servicios. Bajo este contexto, es necesario evitar la intrusión de personas no autorizadas utilizando certificados digitales de seguridad, firmas digitales, *firewalls* o algoritmos de encriptación de los datos considerables sensibles para la organización.

1.46. Requisitos de usabilidad

La aplicación *web* y cuando se considere adicionar una aplicación móvil, deben ser diseñadas y construidas a nivel de Front - End utilizando componentes de GUI que permitan que la aplicación sea intuitiva y sencilla, visualmente agradable a la vista y tenga en cuenta los principios de UX (Experiencia de Usuario) y UI (Interfaz de Usuario).

1.47. Requisitos de disponibilidad

La disponibilidad del sistema debe ser continua con un nivel de servicio para los usuarios las 24 horas, los 7 días de la semana, los 365 días del año, garantizando un esquema adecuado que permita ante un posible fallo en cualquiera de los componentes de la aplicación, contar con una contingencia, generación de alarmas.

1.48. Requisitos de mantenibilidad

El sistema debe disponer de documentación actualizada a nivel planificación, análisis, diseño, construcción e implementación que permita realizar operaciones de mantenimiento con el menor esfuerzo posible para las partes interesadas.

1.49. Requisitos de portabilidad

El sistema de estar diseñado para ejecutarse o desplegarse en *hardware* que tenga unas características a nivel de *software* (Sistema Operativo, Navegadores Web) para el correcto funcionamiento de la aplicación del sistema operativo que se ejecute.