



**UTPL**

*La Universidad Católica de Loja*

Vicerrectorado de Modalidad Abierta y a Distancia

## Fundamentos de Ingeniería de Software

Guía didáctica





Facultad Ingenierías y Arquitectura

## Fundamentos de Ingeniería de Software

### Guía didáctica

Carrera	PAO Nivel
Tecnologías de la Información	V

#### Autores:

Marco Patricio Abad Espinoza

#### Reestructurada por:

Danilo Rubén Jaramillo Hurtado



SIST\_3012

**Fundamentos de Ingeniería de Software**

**Guía didáctica**

Marco Patricio Abad Espinoza

**Reestructurada por:**

Danilo Rubén Jaramillo Hurtado

**Diagramación y diseño digital**

Ediloja Cía. Ltda.

Marcelino Champagnat s/n y París

edilojacialtda@ediloja.com.ec

[www.ediloja.com.ec](http://www.ediloja.com.ec)

**ISBN digital** -978-9942-25-770-3

**Año de edición:** abril, 2020

**Edición:** primera edición reestructurada en septiembre 2025 (con un cambio del 50%)

Loja-Ecuador



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 licenciatante. **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

<b>1. Datos de información .....</b>	<b>9</b>
1.1 Presentación de la asignatura.....	9
1.2 Competencias genéricas de la UTPL.....	9
1.3 Competencias del perfil profesional .....	9
1.4 Problemática que aborda la asignatura .....	10
<b>2. Metodología de aprendizaje .....</b>	<b>11</b>
<b>3. Orientaciones didácticas por resultados de aprendizaje.....</b>	<b>12</b>
<b>Primer bimestre .....</b>	<b>12</b>
<b>    Resultado de aprendizaje 1: .....</b>	<b>12</b>
<b>        Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>12</b>
<b>        Semana 1 .....</b>	<b>12</b>
Unidad 1. Introducción a la ingeniería de software.....	13
1.1. Origen de la ingeniería de software .....	13
1.2. Conceptos de Ingeniería de Software.....	16
Actividades de aprendizaje recomendadas .....	40
Autoevaluación 1.....	42
<b>        Resultado de aprendizaje 2: .....</b>	<b>45</b>
<b>        Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>45</b>
<b>        Semana 2 .....</b>	<b>45</b>
Unidad 2. Procesos de software .....	46
2.1. Modelos de proceso de software .....	46
2.2. Ciclo de vida del Software .....	54
Actividades de aprendizaje recomendadas .....	80
<b>        Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>80</b>
<b>        Semana 3 .....</b>	<b>80</b>
Unidad 2. Procesos de software .....	80
2.4. Scrum.....	80
Actividades de aprendizaje recomendadas .....	84

Autoevaluación 2.....	85
<b>Resultado de aprendizaje 3: .....</b>	<b>88</b>
Contenidos, recursos y actividades de aprendizaje recomendadas.....	88
<b>Semana 4.....</b>	<b>88</b>
Unidad 3. Proyectos de software .....	88
3.1. Definición de proyecto .....	89
3.2. Organización del proyecto .....	90
3.3. Tareas y productos de trabajo.....	93
3.4. Cronograma del proyecto .....	93
3.5. EDT .....	96
3.6. Fases del proyecto .....	97
Actividades de aprendizaje recomendadas .....	99
Autoevaluación 3.....	99
<b>Resultado de aprendizaje 4: .....</b>	<b>102</b>
Contenidos, recursos y actividades de aprendizaje recomendadas.....	102
<b>Semana 5.....</b>	<b>102</b>
Unidad 4. Análisis de software.....	102
4.1. Conceptos del análisis.....	102
4.2. Actividades del análisis .....	109
4.3. Gestión del análisis .....	119
Actividad de aprendizaje recomendada .....	123
Contenidos, recursos y actividades de aprendizaje recomendadas.....	123
<b>Semana 6.....</b>	<b>123</b>
Unidad 4. Análisis de software.....	123
4.4. Requerimientos .....	123
4.5. Proceso de ingeniería de requerimientos .....	132
Actividades de aprendizaje recomendadas .....	137
Autoevaluación 4.....	138
<b>Resultado de aprendizaje 1 y 2:.....</b>	<b>141</b>

<b>Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>141</b>
<b>    Semana 7 .....</b>	<b>141</b>
Unidad 5. Diseño de software .....	141
5.1. Diseño orientado a objetos con UML.....	141
5.2. Diseño del sistema vs. UI .....	143
5.3. Diagrama de contexto.....	143
5.4. Modelo de vistas 4+1.....	146
Actividades de aprendizaje recomendadas .....	160
<b>    Resultado de aprendizaje 1 a 4:.....</b>	<b>162</b>
<b>Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>162</b>
<b>    Semana 8 .....</b>	<b>162</b>
Actividades finales del bimestre .....	162
Actividades de aprendizaje recomendadas .....	162
<b>Segundo bimestre.....</b>	<b>163</b>
<b>    Resultado de aprendizaje 1 y 2:.....</b>	<b>163</b>
<b>    Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>163</b>
<b>    Semana 9 .....</b>	<b>163</b>
Unidad 5. Diseño de software .....	164
5.5. Análisis estructural y de comportamiento .....	164
5.6. Vista procesos.....	165
Actividades de aprendizaje recomendadas .....	170
<b>    Resultado de aprendizaje 3 y 4:.....</b>	<b>171</b>
<b>    Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>171</b>
<b>    Semana 10 .....</b>	<b>171</b>
Unidad 5. Diseño de software .....	171
5.7. Tarjetas CRC .....	171
5.8. Vista lógica: Diagrama de clases .....	174
Actividades de aprendizaje recomendadas .....	180
Autoevaluación 5.....	181

<b>Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>183</b>
<b>Semana 11 .....</b>	<b>183</b>
Unidad 6. Diseño arquitectónico .....	183
6.1. Decisiones en el diseño arquitectónico.....	184
6.2. Conceptos de diseño y arquitectura del sistema.....	186
6.3. Vista de Desarrollo componentes:.....	186
Actividad de aprendizaje recomendada .....	193
<b>Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>194</b>
<b>Semana 12.....</b>	<b>194</b>
Unidad 6. Diseño arquitectónico .....	194
6.4. Patrones arquitectónicos .....	194
6.5. Estilos arquitectónicos .....	198
Actividades de aprendizaje recomendadas .....	202
Autoevaluación 6.....	202
<b>Resultado de aprendizaje 2 y 3:.....</b>	<b>205</b>
<b>Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>205</b>
<b>Semana 13 .....</b>	<b>205</b>
Unidad 7. Diseño, construcción e implementación.....	206
7.1. Patrones de diseño .....	211
7.2. Conflictos en la implementación .....	214
<b>Resultado de aprendizaje 1 y 3:.....</b>	<b>217</b>
<b>Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>217</b>
<b>Semana 14 .....</b>	<b>217</b>
Unidad 7. Diseño, construcción e implementación.....	217
7.3. Del diseño a la implementación .....	218
Actividades de aprendizaje recomendadas .....	241
Autoevaluación 7.....	242
<b>Resultado de aprendizaje 2 y 3:.....</b>	<b>245</b>
<b>Contenidos, recursos y actividades de aprendizaje recomendadas.....</b>	<b>245</b>

<b>Semana 15</b>	<b>245</b>
Unidad 8. Pruebas del software	245
8.1. Conceptos relacionados con las pruebas	246
8.2. Pruebas de desarrollo	247
Actividades de aprendizaje recomendadas	258
Autoevaluación 8	259
<b>Resultado de aprendizaje 1 a 4:</b>	<b>262</b>
<b>Contenidos, recursos y actividades de aprendizaje recomendadas</b>	<b>262</b>
<b>Semana 16</b>	<b>262</b>
Actividades finales del bimestre	262
Actividades de aprendizaje recomendadas	262
<b>4. Solucionario</b>	<b>263</b>
<b>5. Glosario</b>	<b>277</b>
<b>6. Referencias bibliográficas</b>	<b>280</b>
<b>7. Anexos</b>	<b>282</b>





## 1. Datos de información

### 1.1 Presentación de la asignatura



### 1.2 Competencias genéricas de la UTPL

Trabajo en equipo.

### 1.3 Competencias del perfil profesional

- Diseñar aplicaciones de software que permitan mediante técnicas avanzadas de modelado dar solución a los requerimientos del cliente utilizando estándares de la industria.
- Asegurar la calidad tanto de los productos como de los procesos en los proyectos informáticos, utilizando buenas prácticas definidas por la industria para garantizar sistemas eficientes y negocios rentables.
- Gestionar la implementación de soluciones de negocio mediante la ejecución de proyectos de TI que cumplan adecuadamente los requisitos especificados por la organización.

## 1.4 Problemática que aborda la asignatura

Esta asignatura aborda la problemática de la debilidad en la falta de metodologías estructuradas para crear aplicaciones de software en el ámbito empresarial. Esta falta de metodología genera productos de baja calidad con un deficiente proceso de mantenibilidad y sin abordar completamente los requisitos del cliente. Esta materia, entonces, se enfocará en enseñar la implementación de procesos que garanticen un correcto proceso de planificación, análisis, diseño e implementación que permitan mediante estas soluciones ayudar a las empresas a ser más eficientes e incrementar su capacidad operativa.





---

## 2. Metodología de aprendizaje

---

La metodología de trabajo corresponde al Aprendizaje Basado en Proyectos, la cual permite a los estudiantes a más de aprender los conceptos y metodologías, aplicarlos en la creación de una solución a partir de una problemática, a esta problemática ellos la analizan, buscan estrategias de solución y proponen un proyecto que genere la solución desde distintos puntos de vista. Los estudiantes además contarán con un laboratorio virtual donde desarrollarán sus prácticas y experimentación.





### 3. Orientaciones didácticas por resultados de aprendizaje



#### Primer bimestre

##### Resultado de aprendizaje 1:

Conoce las principales áreas de conocimiento de la ingeniería de software.

Para alcanzar el resultado de aprendizaje 1, se llevará a cabo una introducción a los fundamentos de la ingeniería de software. Se estudiará el origen de la disciplina, sus conceptos fundamentales y las principales actividades involucradas en el desarrollo de software. Además, se abordará el desarrollo profesional en el campo, proporcionando una comprensión completa de cómo se organiza y gestiona el trabajo en esta área. Esto permitirá aplicar eficazmente estos conocimientos en proyectos reales y en su futura carrera profesional.

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.

##### Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



##### Semana 1

El desarrollo de software es uno de los ejes centrales de formación del ingeniero en Tecnologías de la Información, a lo largo de la carrera usted se ha formado en competencias de programación y de modelado de sistemas, sin embargo, para desarrollar software desde una perspectiva profesional, hacen

falta otras habilidades como la de analizar un problema de negocio y proponer soluciones que incluyan software, identificar las necesidades del cliente y transformarlas en un diseño antes de programar (construir) y hacer todo esto adoptando metodología que garantice un correcto trabajo en equipo donde diferentes personas trabajan en equipo para construir la solución con los niveles de calidad requeridos en un entorno de negocio. Este conjunto de habilidades, métodos y herramientas es lo que estudia la Ingeniería de software, y en esta asignatura estudiaremos sus fundamentos, los cuales se complementarán a futuro con otras asignaturas.

## **Unidad 1. Introducción a la ingeniería de software**

---

Esta unidad es introductoria y trataremos de poner en contexto los elementos y conceptos importantes de la ingeniería de software, y sobre todo trataremos de demostrar la importancia que tiene en la carrera y en el desarrollo profesional.

### **1.1. Origen de la ingeniería de software**

Para comprender la importancia de la ingeniería de software para las ciencias de la computación, nos remontaremos a finales de 1950, el cual se considera un período esencial en la era de la computación, según (Wirth, 1918), fue la época en la que los grandes ordenadores se pusieron a disposición de centros de investigación y universidades, y fueron utilizados principalmente en ingeniería y ciencias naturales, en esa época también ganaron importancia para su uso en empresas, y nació la profesión de los programadores, la cual era considerada una actividad sofisticada que era desarrollada por gurús de las computadoras.

A inicios de los 60 se crearon los primeros lenguajes de programación, el primero de ellos fue Fortran, que era un lenguaje orientado a trabajar con fórmulas matemáticas, y que fue desarrollado por IBM en 1957, luego apareció su sucesor denominado ALGOL en 1958. El primer lenguaje orientado al desarrollo de aplicaciones para negocios fue COBOL, desarrollado en 1962 por el Departamento de Defensa de los Estados Unidos. Esos sistemas

procesaban sus trabajos por lotes (*batch processing*), eso significaba que todos los trabajos se recolectaban antes de iniciar el procesamiento y tenían que colocarse en una cola de trabajos, no se empezaba a cargar otro trabajo hasta que finalice el primero, y puesto que los dispositivos de entrada como las cintas y tarjetas perforadas eran lentos, había un gran desperdicio del tiempo de procesador haciéndolo muy costoso.

En 1963, surgió el primer sistema de tiempo compartido, diseñado por John McCarthy en el MIT (*Massachusetts Institute of Technology*) y fue implementado en un ordenador DEC PDP-1, este sistema permitía múltiples usuarios trabajando al mismo tiempo, haciendo necesario guardar el estado de cada trabajo en la máquina. Dada su eficiencia, IBM y General Electric fabricaron sistemas de tiempo compartido para sus *mainframes* y, como consecuencia, la complejidad se incrementó considerablemente.

Las principales dificultades encontradas fueron que los sistemas se anunciaron, pero no podían entregarse a tiempo. Los problemas de operación se hicieron muy difíciles de resolver. El multiprocesamiento y la programación concurrente eran los ingredientes centrales de los sistemas de tiempo compartido y los programadores no contaban con la experiencia necesaria para trabajar en este tipo de aplicaciones, en consecuencia, se prometieron sistemas que no pudieron entregarse a tiempo y esto afectó a muchas compañías grandes y debido a ello estuvieron al borde del colapso.

Con este antecedente, en 1968 la OTAN (Organización del Tratado del Atlántico Norte), patrocinó una conferencia ([Software Engineering Reports](#)) en la cual se discutieron abiertamente las dificultades de la programación y se determinó que las técnicas de programación utilizadas en la época eran inadecuadas y que debían adoptarse nuevos métodos. En este contexto fueron acuñados los términos de ingeniería de *software* y crisis del *software*. Lo cual dio origen a nuevos esfuerzos por resolver la complejidad y hacer que la programación pase de ser un arte a una disciplina de ingeniería.

Ahora revise la lectura correspondiente, el [Tema 1 del OCW ingeniería de software I](#). Vaya a materiales de clase y descargue el documento MC-F-002 y estudie el apartado **Evolución histórica del desarrollo de software**. Revise la figura 1 para que se familiarice con el material.

### Figura 1

Presentación de contenidos de la asignatura

The screenshot shows a navigation menu from an OCW course. At the top, there's a logo for "OpenCourseWare UC" and a link to "Página Principal". Below the menu, there are several items: "Ingeniería del Software I (...)" which has a dropdown arrow; "Programa" and "Bibliografía" which also have dropdown arrows; and a blue-highlighted item "Materiales de Clase". To the right of these, there are four numbered items: "MC-F-001. Tema 0. Presentación de la asignatura.", "MC-F-002. Tema 1. Introducción a la Ingeniería del Software.", "MC-F-003. Tema 2. Lenguaje Unificado de Modelado (UML).", and "MC-F-004. Tema 3. Procesos de Ingeniería del Software.". The "Materiales de Clase" item is highlighted with a blue background.

Nota. Tomado de *Ingeniería del Software I* [Imagen], por Open Course Ware, 2011, [OCW](#), CC BY 4.0.

Como habrá notado, la ingeniería de software es una ciencia relativamente nueva y debido a la evolución de la tecnología cada vez tiene más desafíos, y los niveles de complejidad son cada vez más altos.

En razón de ello, vale la pena tratar de dar respuesta a las siguientes interrogantes:

- ¿Cuáles son los factores que inciden en la complejidad del software?,
- ¿qué cambios importantes en los métodos, técnicas y herramientas de ingeniería de software se han producido en los últimos años para atender los problemas que enfrentan los proyectos de desarrollo de software?,
- ¿Cuáles son las tendencias en el campo de la ingeniería de software?

## 1.2. Conceptos de Ingeniería de Software

En el apartado anterior, hemos realizado un breve recorrido por los orígenes de la ingeniería de software, el cual nos ha servido para plantearnos el porqué de su importancia, ahora vamos a revisar algunos conceptos importantes

En (Sommerville, 2016) se plantean algunos conceptos importantes en este campo, resumimos algunos de ellos en la Tabla 1.



**Tabla 1***Conceptos de Ingeniería de Software*

Preguntas	Respuesta
¿Qué es software?	Programas de cómputo y documentación asociada. Los productos de software se desarrollan para un cliente en particular o para un mercado en general.
¿Cuáles son los atributos del buen software?	El buen software debe entregar al usuario la funcionalidad y el desempeño requerido, y debe ser sustentable, confiable y utilizable.
¿Qué es ingeniería de software?	La ingeniería de software es una disciplina de la ingeniería que se interesa por todos los aspectos de la producción de software.
¿Cuáles son las actividades fundamentales de la ingeniería de software?	Especificación, desarrollo, validación y evolución del software.
¿Cuál es la diferencia entre ingeniería de software y ciencias de la computación?	Las ciencias de la computación se enfocan en teoría y fundamentos; mientras la ingeniería de ciencias de la software se enfoca en el sentido práctico del desarrollo y en la distribución de software computación?
¿Cuál es la diferencia entre ingeniería de software e ingeniería de sistemas?	La ingeniería de sistemas se interesa por todos los aspectos del desarrollo de sistemas basados en computadoras, incluidos hardware, software e ingeniería de procesos. La ingeniería de software es parte de este proceso más general.
	Se enfrentan con una diversidad creciente, demandas por tiempos de distribución limitados y desarrollo de software confiable.

Preguntas	Respuesta
¿Cuáles son los principales retos que enfrenta la ingeniería de software?	
¿Cuáles son los costos de la ingeniería de software?	Aproximadamente 60% de los costos del software son de desarrollo, y 40% de prueba. Para el software elaborado específicamente, los costos de evolución superan con frecuencia los costos de desarrollo.
¿Cuáles son los mejores métodos y técnicas de la ingeniería de software?	Aun cuando todos los proyectos de software deben gestionarse y desarrollarse de manera profesional, existen diferentes técnicas que son adecuadas para distintos tipos de sistema. Por ejemplo, los juegos siempre deben diseñarse usando una serie de prototipos, mientras que los sistemas críticos de control de seguridad requieren de una especificación completa y analizable para su desarrollo. Por lo tanto, no puede decirse que un método sea mejor que otro.
¿Qué diferencias ha marcado la Web a la ingeniería de software?	La Web ha llevado a la disponibilidad de servicios de software y a la posibilidad de desarrollar sistemas basados en servicios distribuidos ampliamente. El desarrollo de sistemas basados en Web ha conducido a importantes avances en lenguajes de programación y reutilización de software.
¿Cuáles son los principales retos que enfrenta la ingeniería de desarrollo de software confiable?	Se enfrentan con una diversidad creciente, demandas por tiempos de distribución limitados y
¿Cuáles son los costos de la ingeniería de software?	Aproximadamente 60% de los costos del software son de desarrollo, y 40% de prueba. Para el software elaborado específicamente, los costos de evolución superan con frecuencia los costos de desarrollo.

Preguntas	Respuesta
¿Cuáles son los mejores métodos y técnicas de la ingeniería de software?	Aun cuando todos los proyectos de software deben gestionarse y desarrollarse de manera profesional, existen diferentes técnicas que son adecuadas para distintos tipos de sistema. Por ejemplo, los juegos siempre deben diseñarse usando una serie de prototipos, mientras que los sistemas críticos de control de seguridad requieren de una especificación completa y analizable para su desarrollo. Por lo tanto, no puede decirse que un método sea mejor que otro.
¿Qué diferencias ha marcado la Web a la ingeniería de software?	La Web ha llevado a la disponibilidad de servicios de software y a la posibilidad de desarrollar sistemas basados en servicios distribuidos ampliamente. El desarrollo de sistemas basados en Web ha conducido a importantes avances en lenguajes de programación y reutilización de software.

Nota: Adaptado de *Software engineering* (10th ed.), por Sommerville, I., 2016, Pearson Education.

Para complementar esta información, tome nuevamente el documento del tema 1 del recurso OCW de Ingeniería de software 1, trabajado en el apartado anterior, revise los apartados “**Problemática del desarrollo de software**” y “**Contexto de la ingeniería de software**”.

Una vez consultados y revisados estos conceptos, debería ser capaz de describir con precisión los siguientes interrogantes:



- ¿Qué es el *software*?
- ¿Cuáles son los atributos de un buen *software*?
- ¿Cuáles son las actividades de la ingeniería de *software*?
- ¿Qué es una ingeniería?
- ¿Qué es y qué hace un ingeniero?
- ¿Qué es un sistema?
- ¿Qué es un proceso?
- ¿Qué es un proyecto?
- ¿Qué es un usuario y qué tipos de usuarios hay?

Con seguridad logró responder a cada una de las interrogantes planteadas, sin embargo, en ellas no se ha mencionado lo que es la ingeniería de *software*, por lo tanto, vamos a revisar otros conceptos importantes:

### **1.2.1. Ingeniería de software**

El glosario estándar de ingeniería de *software* del Instituto de Ingenieros Eléctricos y Electrónico (IEEE, por sus siglas en inglés) (IEEE Standard Glossary of Software Engineering Terminology, 1990), define a la ingeniería de *software* como:

La aplicación de un enfoque sistemático, disciplinado y cuantificable para el desarrollo, operación y mantenimiento del *software*, esto es la aplicación de ingeniería al *software*.

Otra definición dada en (Kung, 2014) dice:

"Ingeniería de *software* como disciplina es enfocada en la investigación, educación y aplicación de procesos de ingeniería y métodos para incrementar significativamente la productividad y la calidad del *software* mientras se reducen los costos del *software* y el tiempo de comercialización".

Si analizamos estas dos definiciones, podemos establecer que la ingeniería de software, rompe el esquema de la simple codificación de programas, y pasa al “estudio y aplicación de ingeniería al diseño, desarrollo y mantenimiento de productos de software” (Ahmed, 2016) y para entender lo que significa aplicar ingeniería a los procesos de desarrollo de software, vamos a remitirnos a los aspectos que plantean (Bruegge & Dutoit, 2018) respecto de lo que es la ingeniería de software:

- Es una actividad de modelado, lo cual permite a los ingenieros simplificar problemas complejos enfocándose en una parte de la realidad, los ingenieros pueden desarrollar varios modelos del sistema y del dominio de la aplicación.
- Es una actividad de resolución de problemas, y para ello usan los modelos que les permitan llegar a soluciones aceptables, para ello puede realizar actividades de experimentación y debido a las limitaciones en cuanto a recursos y tiempo suelen utilizar técnicas empíricas para evaluar las ventajas y desventajas de las diferentes alternativas de solución.
- Es una actividad de adquisición de conocimiento. Cuando se modela la aplicación, los ingenieros de software recolectan datos, los organizan en información y la formalizan en conocimiento.
- Es una actividad fundamentada en la razón. Cuando se adquiere conocimiento y se toma decisiones acerca del sistema o de su dominio de aplicación, los ingenieros también necesitan capturar el contexto en el cual se tomaron esas decisiones y el fundamento detrás de las mismas, este fundamento debe permitir a los ingenieros comprender las implicaciones de los cambios propuestos cuando se revisa estas decisiones.

A continuación, en el siguiente módulo didáctico se analiza el significado y las implicaciones de cada uno de estos aspectos, lo invito a revisar el siguiente módulo didáctico:

[Aspectos de la ingeniería de software](#)

A continuación, podemos encontrar algunos factores sobre el software con aspectos relacionados al mismo, ya en el transcurso de esta guía iremos ampliando dichos aspectos, la tabla 2 muestra estos factores.

**Tabla 2**

*Factores de calidad del software*

FACTORES	ASPECTOS CONSIDERADOS
Características operativas	<ul style="list-style-type: none"><li>• <b>Correcto:</b> ¿el software hace lo que quiero?</li><li>• <b>Fiable:</b> ¿Lo hace de forma fiable todo el tiempo?</li><li>• <b>Eficiencia:</b> ¿Se ejecutará en el HW lo mejor que pueda?</li><li>• <b>Integridad:</b> ¿Es seguro?</li><li>• <b>Facilidad de uso:</b> ¿Usabilidad, accesibilidad?</li></ul>
Soporte a cambio	<ul style="list-style-type: none"><li>• <b>Fácil mantenimiento:</b> ¿Puedo corregirlo?</li><li>• <b>Flexibilidad:</b> ¿Puedo cambiarlo fácilmente?</li><li>• <b>Facilidad de prueba:</b> ¿Puedo probarlo?</li></ul>
Adaptable a nuevos entornos	<ul style="list-style-type: none"><li>• <b>Reusabilidad:</b> ¿Podré reutilizar alguna parte?</li><li>• <b>Portabilidad:</b> ¿Podré usarlo en otro HW o SO?</li><li>• <b>Interoperabilidad:</b> ¿Podrá interactuar con otro sistema?</li></ul>

Nota. Jaramillo, D., 2025.

Aquí podemos considerar algunos aspectos por ejemplo ¿el software hace lo que quiero?, desde el punto de vista de quien lo necesita es muy importante por cuanto en algunas ocasiones el software hace lo que el programador desarrolla y no lo que el cliente quiere, aunque pueda parecer muy obvio esto suele suceder.

### **1.2.2. Participantes y roles**

Un proyecto de software requiere el trabajo en equipo de muchas personas, cada una de las cuales tiene diferentes habilidades e intereses en el proyecto, como ejemplo podemos mencionar el cliente, los usuarios, el gestor del proyecto, los desarrolladores y escritores técnicos, por tanto, que cada uno de estos asume un rol en el proyecto y cada rol tiene sus propias responsabilidades, puede haber varias personas ejecutando el mismo rol y un participante también puede ejecutar más de un rol.

Dependiendo del proyecto y del proceso de desarrollo adoptado para el proyecto, las personas pueden ejecutar diferentes roles. En la tabla 3, se describen algunos de los roles comunes en proyectos de desarrollo de software, así como sus responsabilidades.



**Tabla 3***Roles comunes en proyectos de desarrollo de software*

Rol	Responsabilidades	Ejemplo
Cliente	<ul style="list-style-type: none"><li>Define el problema o necesidad del negocio.</li><li>Proporciona los requerimientos de alto nivel para la definición del alcance.</li><li>Proporciona los criterios de éxito del proyecto.</li></ul>	Una empresa o su representante. Un gerente de un departamento.
Usuario	<ul style="list-style-type: none"><li>Provee el conocimiento respecto de las actividades que debe realizar en sus actividades diarias y que servirán para establecer los requerimientos del sistema.</li></ul>	Un vendedor. Un cajero de un banco. Una secretaria.
Gestor	<ul style="list-style-type: none"><li>Responsable de la organización del trabajo del equipo.</li><li>Contrata personas, asigna tareas, monitorea el avance del proyecto, gestiona o provee entrenamiento al equipo del proyecto, administra los recursos asignados por el cliente.</li><li>Responsable de la entrega de los resultados y la culminación del proyecto.</li></ul>	Gestor del proyecto. Scrum Máster
Desarrollador	<ul style="list-style-type: none"><li>Responsable de las actividades de especificación, diseño, implementación y pruebas del sistema.</li></ul>	Analista. Programador. Tester.



Rol	Responsabilidades	Ejemplo
Escritor técnico	<ul style="list-style-type: none"> <li>• Escribir la documentación que será entregada al cliente.</li> <li>• Entrevista a los desarrolladores, gestores y usuarios para comprender el sistema.</li> </ul>	Documentador. Elaborador de manuales de usuario.

Nota. Jaramillo, D., 2025.

### 1.2.3. Productos de trabajo

Para (Bruegge & Dutoit, 2018)los productos de trabajo son artefactos (elementos tangibles) que se producen durante el proceso de desarrollo, y estos pueden ser para consumo interno del equipo de proyecto en cuyo caso adoptan el nombre de productos de trabajo internos, y existen productos de trabajo que deben entregarse al cliente, a estos productos de trabajo se los conoce como entregables, los entregables se definen en el *contrato* y forman parte del alcance del proyecto, estos son los resultados visibles que agregan valor al cliente.

Tenga en cuenta que el *contrato* es el documento formal que firma el cliente con la empresa desarrolladora, donde se listan los productos que se entregarán, las condiciones de entrega y las acciones a tomar en caso de incumplimiento.

Los productos de trabajo interno pueden clasificarse en documentación técnica, reportes de avance, modelos, código fuente, por ejemplo, en documentación técnica se puede tener: Normas de programación, manuales de pruebas; en lo referente a modelos puede tener: modelos de software, modelos de datos. En cuanto a los entregables se puede tener



categorías como: Documentación de usuario, programas ejecutables, documentos de aprobación, por ejemplo, documentación se puede tener manuales de usuario, manual de configuración; programas ejecutables: módulos de la aplicación, librerías.

#### 1.2.4. Requerimientos

Los requerimientos de usuario son “descripciones de las propiedades necesarias y suficientes de un producto que satisfará las necesidades del consumidor” y los requerimientos de software son “descripciones de las propiedades necesarias y suficientes del software que deben cumplirse para asegurar que el producto logre el objetivo para lo que fue diseñado”(Gottesdiener, 2005)

Por tanto, los requerimientos de software son las especificaciones de las características que el producto de software debe satisfacer con el fin de lograr la aceptación de los usuarios. Aunque este tema se desarrollará con mayor detalle más adelante, es importante mencionar que los requerimientos de software pueden ser funcionales y no funcionales.

Los **requerimientos funcionales** describen las funciones que el sistema debe permitir en relación con las actividades del usuario para cumplir con sus actividades de negocio, y los **requerimientos no funcionales** especifican restricciones de operación del sistema.

Para ilustrar estos requerimientos, piense en un sistema para operar una biblioteca en una universidad, en donde se ha identificado los requerimientos descritos en la tabla 4 analícelos y establezca diferencias entre ellos.



Los requerimientos no funcionales se asocian a la calidad del producto, concretamente a ciertas características denominadas atributos de calidad, y tienen un alto impacto en las decisiones de diseño.

Recuerde que en todo momento puede acudir a su tutor para aclarar cualquier duda que se le presente durante su proceso de aprendizaje.



**Tabla 4**

Ejemplo de requerimientos funcionales y no funcionales para un sistema de biblioteca

ID	Tipo	Requerimiento
F01	Funcional	El sistema debe permitir crear una ficha de datos de los libros que incluya: título, autor(es), año de edición, editorial, tabla de contenidos, descriptores, materia.
F02	Funcional	El sistema debe permitir registrar una ubicación física de los libros.
F03	Funcional	Debe permitir la clasificación del libro de acuerdo al sistema decimal Dewey.
F04	Funcional	Debe permitir la creación de un código único del libro, para identificarlo en los diferentes procesos operativos.
F05	Funcional	El sistema permitirá la impresión de etiquetas identificativas que incluyan el título, el apellido del autor, el año de publicación, la ubicación física y el código único del libro.
F06	Funcional	El sistema debe permitir la búsqueda de los libros por autor, materia, descriptor, título, editorial.
F07	Funcional	El sistema debe permitir el registro de préstamos a estudiantes y docentes de la Universidad.
F08	Funcional	El sistema debe permitir el registro de las devoluciones y el reingreso del libro a circulación.
N01	No funcional	El sistema debe brindar protección de acceso con contraseña a los componentes de registro de material, préstamos y devoluciones sólo a los usuarios autorizados.
N02	No funcional	El tiempo máximo que debe demorar la entrega de un libro en préstamo es de 1 minuto.
N03	No funcional	El módulo de consultas debe ser una aplicación web.

ID	Tipo	Requerimiento
N04	No funcional	El sistema debe enviar notificaciones al usuario un día antes del plazo de devolución de los libros prestados.

Nota. Jaramillo, D., 2025.

Continuemos con el aprendizaje mediante la revisión de las actividades y desarrollo profesional en el software.

## Desarrollo y práctica profesional del software

### 1.3. Actividades de desarrollo de software

Como se mencionó en el apartado anterior, la ingeniería de software implica mucho más que la programación de aplicaciones, en este apartado vamos a revisar las actividades genéricas que se realizan durante el desarrollo de software, independientemente de la metodología o el enfoque que se utilice.

Antes de comenzar, es necesario hacer algunas precisiones respecto de los términos actividad, tarea y recursos.

Una actividad “Es un proceso que tiene lugar en el tiempo y en el espacio, y en el cual un agente actúa con unos objetivos determinados(Sánchez, 2012) las actividades, por tanto, son un conjunto de tareas que se ejecutan con el fin de obtener un artefacto, y pueden ser actividades generales en cuyo caso incluyen otras actividades específicas. Las

actividades se componen de tareas y pueden ser asignadas a un **recurso**. Una **tarea** es una unidad mínima de trabajo que puede ser gestionada, e implica la ejecución de los diferentes pasos que una persona debe ejecutar para completar las actividades.

Una actividad es asignada a un recurso (desarrollador), quien realiza las tareas necesarias para cumplir con la actividad, las tareas consumen recursos y producen entregables. Los **recursos** pueden ser suministros para desarrollar las tareas, tiempo y equipamiento.

En los procesos de planificación un gestor descompone el trabajo, en actividades y en tareas a las cuales les asigna recursos. Estos conceptos se entienden mejor con los ejemplos de la tabla 5



**Tabla 5***Ejemplos de actividades, tareas y recursos*

Ejemplo	Tipo	Descripción
Levantamiento de requerimientos.	Actividad	Permite obtener el artefacto Especificación de requerimientos, e incluye las tareas necesarias para obtener y validar los requerimientos del lado cliente.
Desarrolla el caso de prueba “Sin valor exacto” para el sistema de cajero automático.	Tarea	La tarea asignada al recurso Jhon (tester) se enfoca en verificar el comportamiento del cajero automático cuando se intenta retirar dinero y este no tiene las denominaciones de billete exactas para entregar el valor solicitado por el cliente. Incluye especificar el entorno de las pruebas, la secuencia de entradas a probar y las salidas esperadas.
Base de datos de cuentas.	Recurso	Incluye un ejemplo de la estructura de la base de datos de cuentas para poder realizar pruebas.

Nota. Adaptado de *Object-Oriented Software Engineering*, por Bruegge, B. y Dutoir, A., 2018, Prentice Hall.

Dependiendo de los autores, las actividades de desarrollo de software se plantearon originalmente como análisis, diseño implementación, pruebas y mantenimiento, pero esta percepción ha ido evolucionando con el tiempo y hoy en día se cuenta con diferentes enfoques y metodologías de desarrollo de las que nos ocuparemos en la siguiente unidad.

A continuación, se presenta una versión genérica de las actividades del proceso de desarrollo y le ayudarán a comprender en qué consiste el desarrollo de software. Las actividades del proceso de desarrollo de software son cuatro: especificación, diseño e implementación, validación y evolución del software, dónde cada una cumple con un objetivo importante en la construcción del software. A continuación, se analizan cada una de ellas:

### **1.3.1. Especificación del software**

Consiste en la primera actividad para comprender qué servicios se requieren del software. Se la conoce también como Ingeniería de requisitos y consiste en actividades de desarrollo y gestión.

Es importante aclarar que estas cuatro actividades:

- Estudio de factibilidad
- Obtención y análisis de requerimientos
- Especificación de requerimientos
- Validación de requerimientos

No se realizan una sola vez, sino que de acuerdo a la información o ajustes que se tengan que considerar se abordan las veces que sean necesario para definir los requerimientos de software.

También es importante el uso de herramientas de gestión de requisitos ya que permiten tener un mayor control del desarrollo de los requerimientos. Más adelante en la unidad 4, se indicará la manera de especificar los requisitos funcionales y no funcionales.

### **1.3.2. Diseño e Implementación del software**

En base a los requisitos de software definidos en la especificación se procede a diseñar la aplicación y luego a implementar. Al diseño se lo define como “el proceso de definir la arquitectura, los componentes, las interfaces y otras características de un sistema o componente” y “el resultado de ese proceso”. Desde el punto de vista de proceso, el diseño de software es la actividad del

ciclo de vida de la ingeniería de software en la que se analizan los requisitos de software para producir una descripción de la estructura interna del software que servirá de base para su construcción

### 1.3.3. Validación del software

En esta etapa se realiza una actividad fundamental que consiste en verificar que lo que se ha construido es lo que el cliente ha solicitado, evidentemente no es una actividad final, sino que se realiza desde que la especificación del software.

Las pruebas consisten en la verificación del software ante comportamientos esperados para un conjunto finito de casos de prueba adecuadamente seleccionados del dominio del problema.

Actualmente la visión de las pruebas de software ha cambiado de forma efectiva respecto a años anteriores. Las pruebas ya no se consideran como una actividad que se inicia después de que se complete la fase de codificación, sino más bien que las pruebas de software son o deberían ser a lo largo de todo el ciclo de vida de desarrollo y mantenimiento.

La planificación de las pruebas de software debería comenzar con las primeras etapas del proceso de requisitos de software y los planes y procedimientos de prueba deberían desarrollarse sistemáticamente y continuamente y refinarse a medida que avanza el desarrollo del software. Estas actividades de planificación de pruebas y diseño de pruebas proporcionan información útil para diseñadores de software y ayudan a resaltar posibles debilidades, como descuidos y contradicciones de diseño, u omisiones y ambigüedades en la documentación

### 1.3.4. Evolución del software

También conocida como mantenimiento del software, es una actividad fundamental ya que a medida que pasa el tiempo, las aplicaciones de software deben ser sometidas a procesos de modificación que extiendan su vida útil o mejoren sus características. Corrección de errores(bugs), adaptación a nuevos



entornos tecnológicos o agregado de funcionalidad son algunas de las tareas que incluye el mantenimiento del software, siendo una de las actividades que se repite periódicamente desde que empieza a utilizarse hasta su abandono definitivo.

Podemos acotar que siempre habrá que realizar actualizaciones al software ya desarrollado, estas actualizaciones deben realizarse bajo un modelo que permita gestionar los requerimientos sin dejar de lado la calidad del producto. Estas actualizaciones denominada gestión del cambio se puede realizar mediante prototipos o entrega incremental.

Para complementar el tema, y puesto que la presente asignatura se centra en Ingeniería de software orientada a objetos, a continuación, se comparte una infografía donde se describen las actividades que se desarrollan bajo este paradigma de programación, que en sí no constituye una metodología, sino que describe los procesos necesarios para construir software:

#### [Procesos de desarrollo de software orientado a objetos](#)

#### **1.4. Desarrollo profesional de software**

La programación, hoy en día es una actividad que muchos profesionales realizan para resolver sus propias necesidades, sin embargo, los ingenieros en sistemas, Tecnologías de la Información, informática o carreras afines, hacen desarrollo de software de manera profesional, eso significa que no desarrollan para sí mismos, sino que desarrollan para solucionar las necesidades sobre todo de empresas de diferentes sectores.

En este sentido, debe tomarse prácticas y estrategias de desarrollo a las que llamamos desarrollo profesional de software, además, evaluar el impacto de los fallos en los proyectos de ingeniería de software y las implicaciones éticas conjungan una serie de elementos fundamentales para comprender el papel del ingeniero en Tecnologías de la Información en lo referente al desarrollo de aplicaciones de software.

En la actualidad, la producción de software no se trata solamente de construir programas para solucionar un tema en concreto, sino de aplicar estrategias que permitan construir productos. Empiece conociendo cómo surge la ingeniería de software, para ello revise el tema Introducción a la ingeniería de software del REA Ingeniería de software I, diapositivas de la 1 a la 8.[\[1\]](#)

Terminada la lectura, podemos concluir que, el software de hoy en día requiere de tecnología sofisticada, más aún cuando se necesita procesar y compartir grandes cantidades de información entre usuarios y clientes. Una vez conocido el origen de la ingeniería de software, continúe revisando el tema Desarrollo de software profesional.

Finalizada la lectura del tema indicado podemos deducir las razones por las que es indispensable la construcción de software a nivel profesional, ya que en la actualidad las organizaciones no requieren de un simple programa que funcione de forma aislada, sino que dé soluciones integrales que contemplen las actividades que se realizan en cada uno de sus procesos.

Además, es importante que establezca la diferencia entre los dos tipos de software que se conocen: los Productos genéricos y los Productos personalizados. En ambos casos, la importancia radica no solamente en la satisfacción del cliente, sino también en los desarrolladores y los elementos derivados del producto software, que se reflejan en ciertos elementos conocidos como atributos.

En resumen, la ingeniería de software es una actividad que permite:

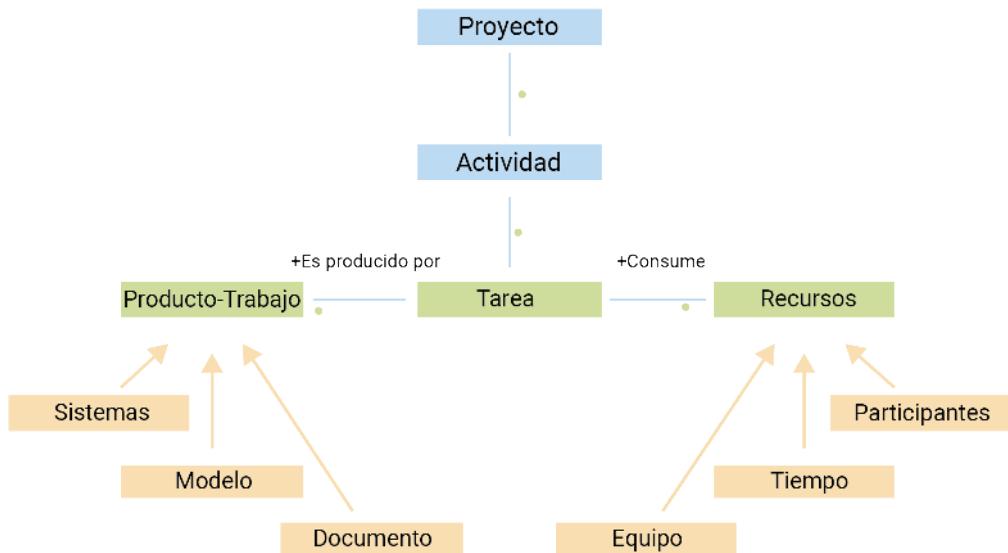
- **Modelado** para resolver problemas de complejidad tanto del proceso como del software.
- **Resolución de problemas** mediante el desarrollo de modelos que buscan la solución aceptable.
- **Adquisición de conocimiento**, debido a que los ingenieros de software recopilan datos, los organizan en información y los formalizan en conocimiento.

- **Impulsada por la lógica**, para que al adquirir conocimientos y tomar decisiones sobre el sistema o su dominio de aplicación, los ingenieros de software también puedan capturar el contexto en el que se tomaron las decisiones y la justificación detrás de estas decisiones.

En cada una de estas actividades, los ingenieros de software tienen que trabajar bajo las limitaciones de personas, tiempo y presupuesto, considerando que el cambio puede ocurrir en cualquier momento.

**Figura 2**

Conceptos de ingeniería de software representados como un diagrama de clases UML



Nota. Jaramillo, D., 2025.

En la figura 2, se muestra los conceptos relacionados con la ingeniería de software representados mediante un diagrama UML, que consiste en lo siguiente: un proyecto de software, cuyo propósito es desarrollar un sistema de software, está compuesto de varias actividades. Cada actividad está compuesta por una serie de tareas. Una Tarea consume recursos y produce un producto-trabajo. Un Producto-Trabajo puede ser un sistema, un modelo o un documento. Los recursos son: participantes, tiempo o equipo.

En el diagrama cada rectángulo representa un concepto. Las líneas entre los rectángulos representan diferentes relaciones entre los conceptos. Por ejemplo, la forma de los diamantes indica agregación: un proyecto incluye varias actividades, que incluye varias tareas. La forma del triángulo indica una relación de generalización.

## Fallas en la ingeniería de software

Los sistemas de software que se han construido a lo largo de los años se han integrado en todos los aspectos de la sociedad, por lo que parte de nuestra vida está ligada al software. A pesar de las buenas intenciones, existe un gran número de proyectos software que no han tenido éxito. Un gran porcentaje de proyectos nunca se terminaron, y peor aún, algunos de proyectos terminaron en desastre, causando pérdida de dinero, tiempo y tragedias de pérdida. Este es un tema importante que justifica el uso de la ingeniería de software para el desarrollo del software. A continuación, se listan algunos casos de las fallas atribuidas al software que han ocasionado serios problemas. Mencionaremos algunos:

- Error de año bisiesto.
- Fracaso del Mariner 1 (1962).
- Sobregiro del Banco de New York (1985).
- Accidente de un F-18 (1986).
- Error del milenio (2000): O "Y2K".
- Medicina: radioterapia.
- Transporte: Aerolínea American Airlines.
- Retrasado y con sobrepresupuesto.
- Tiempo de entrega.

Cada una de estas fallas es consecuencia de problemas en el software. Para algunos casos, los desarrolladores no anticiparon situaciones raras, por ejemplo, que una persona pueda vivir más de 100 años o los años bisiestos que afectan las fechas de vencimiento. En otros casos, los desarrolladores no esperaban que los usuarios utilizaran el sistema de forma activa, por ejemplo, explotando los agujeros de seguridad en el software de red. Y en otros casos

los fallos del sistema resultaron de fallas en la gestión del proyecto, por ejemplo, entrega tardía y exceso de presupuesto, entrega puntual de un sistema incorrecto o complejidad innecesaria (Bruegge & Dutoit, 2018).

Afortunadamente, no todos los proyectos terminan en los desastres que se indican anteriormente, pero sí en fracaso. Entonces, ¿Qué hace que un proyecto de software no tenga éxito? En pocas palabras, un proyecto de software sin éxito es uno que no cumple con las expectativas del cliente. Más específicamente, los resultados indeseables pueden incluir lo siguiente:

- Proyectos por encima del presupuesto.
- Excesiva planificación y/o pérdida de presencia en el mercado.
- No cumple con los requisitos del cliente.
- Menor calidad de la esperada.
- El rendimiento no satisface las expectativas.
- Demasiado difícil de utilizar.

En conclusión, la aparición de errores (bugs) no es anormal, está relacionada con el comportamiento del ser humano, lo complejo del problema y la responsabilidad en aplicar mecanismos de validación. Cuando un software controla dispositivos peligrosos como control aéreo, balístico y espacial, gestiona material peligroso o controla máquinas peligrosas, el más mínimo detalle cuenta para que no se convierta en una catástrofe de grandes magnitudes.

#### **1.4.1. Aspectos éticos**

Como en todas las profesiones la ética profesional está atada a un comportamiento adecuado y más aún cuando se trata de un profesional de desarrollo de software, ya que cualquiera que sea el rol a desempeñar en el proyecto de desarrollo, va a tener acceso a información importante y exclusiva de la organización, por lo tanto, se requiere de responsabilidades que va más allá de los conocimientos técnicos con el fin de precautelar la integridad y confidencialidad de la información.

Se puede resaltar que:

- La ingeniería de software **implica responsabilidades** mayores que el simple uso de habilidades técnicas.
- Los ingenieros de Software deben **comportarse de manera honesta y responsable** para que sean respetados como profesionales.
- El comportamiento responsable es mucho **más que simplemente actuar dentro de la ley**.

La ética en la ingeniería de software se refiere a los valores y principios que guían las decisiones de los desarrolladores en relación con los usuarios, los clientes y la sociedad. Algunas consideraciones clave son:

- **Responsabilidad profesional:** los ingenieros deben garantizar la calidad del software, procurando que sea seguro, confiable y cumpla con los objetivos para los cuales fue diseñado.
- **Respeto a la privacidad:** el manejo de datos personales exige confidencialidad y el diseño de mecanismos que eviten filtraciones o usos indebidos.
- **Transparencia:** informar a los usuarios sobre las limitaciones, riesgos o posibles errores del sistema.
- **Impacto social:** considerar las consecuencias que puede tener un software en la vida de las personas, por ejemplo, en sistemas de salud, educación o finanzas.
- **Propiedad intelectual:** reconocer el trabajo de otros, evitar el plagio y fomentar la innovación responsable.

Códigos como el de la Association for Computing Machinery (ACM) o el de la IEEE-CS orientan a los profesionales en la toma de decisiones éticas en escenarios complejos.

## Aspectos legales

El marco legal busca proteger a los creadores, usuarios y a la sociedad en general frente a los riesgos asociados al uso del software. Entre los más relevantes se encuentran:

- **Derechos de autor y licencias:** el software está protegido como obra intelectual, lo que implica respetar las licencias de uso (propietarias, libres o de código abierto) y evitar la piratería.
- **Protección de datos personales:** leyes como el Reglamento General de Protección de Datos (GDPR) en Europa o la Ley de Protección de Datos Personales en Latinoamérica obligan a garantizar el consentimiento, la transparencia y la seguridad en el manejo de información sensible.
- **Contratos y propiedad del software:** los acuerdos entre cliente y desarrollador deben establecer claramente la titularidad, uso y mantenimiento del software.
- **Ciberseguridad y delitos informáticos:** existen normativas que sancionan el acceso no autorizado a sistemas, la difusión de *malware* y los fraudes digitales.
- **Accesibilidad digital:** en muchos países, la legislación exige que el software sea accesible para personas con discapacidad, como parte de la inclusión digital.

[1] [Cálculo Simbólico y Numérico en Ecuaciones Diferenciales \(2014\)](#)

## Actividades de aprendizaje recomendadas

A continuación, le invito a que desarrolle las actividades de aprendizaje recomendadas:

1. Revise el siguiente video introductorio a la ingeniería de software para profundizar en sus conceptos básicos: [Ingeniería de software clase 1](#).
2. Con base en lo explicado en el apartado 1.2.4 y el ejercicio de la tabla 2, piense en 4 requerimientos funcionales y 4 requerimientos no funcionales para un **sistema de registro de entrega de paquetes (TRAMIL)**. [2] Anótelos en la siguiente tabla:

### *Lista de requerimientos funcionales y no funcionales*

ID	Tipo	Requerimiento
F01	Funcional	
F02	Funcional	
F03	Funcional	
F04	Funcional	
NF01	No Funcional	
NF02	No Funcional	
NF03	No Funcional	
NF04	No Funcional	

*Nota.* Copie la tabla en un Word o cuaderno para llenar.

3. Con base en la información provista en la presente unidad, desarrolle un ensayo de máximo 2 páginas, donde sustente el papel del ingeniero de Software y la importancia de su trabajo para las empresas y para la sociedad en general.

*Nota.* Complete las actividades en un cuaderno o documento Word.

4. En esta unidad se introdujeron los conceptos más importantes de la ingeniería de software, desde su origen hasta las actividades del proceso Orientado a Objetos. Para consolidar lo aprendido, desarrolle la **autoevaluación** de la **unidad 1**.



## Autoevaluación 1

1. El concepto de ingeniería de software surge como respuesta a la crisis del *software* tras una reunión organizada por la OTAN. ¿A qué situación hacía referencia dicha crisis?
  - a. La creciente incorporación de computadoras en empresas.
  - b. El nacimiento de la profesión de programador.
  - c. El desarrollo de los primeros sistemas de tiempo compartido.
  - d. La creación de ordenadores de menor tamaño.
2. Según la evolución histórica del *software*, ¿en qué año comenzaron a utilizarse los sistemas operativos y a qué era pertenecen?
  - a. 1960, primera era.
  - b. 1970, segunda era.
  - c. 1990, tercera era.
  - d. 2010, era de las TIC.
3. En el marco de las tendencias de mercado, ¿cuál de estas opciones caracteriza a la Arquitectura Orientada a Servicios (SOA)?
  - a. Proliferación de smartphones avanzados.
  - b. Sistemas abiertos con servicios accesibles externamente.
  - c. Expansión de la computación en la nube.
  - d. Enfoque en procesos y proyectos empresariales.
4. El término *software* incluye:
  - a. Programas, datos y documentación.
  - b. Programas y datos.
  - c. Programas y manuales de usuario.
  - d. Programas, procesos y personal.

5. Señala cuál de las siguientes afirmaciones no corresponde al *software*:

- a. Es intangible, no físico.
- b. No se daña como el *hardware*, pero puede degradarse.
- c. Se desarrolla a la medida de las necesidades.
- d. Carece de incertidumbre durante su construcción.

6. ¿Cuál de las siguientes prácticas no es adecuada para un proyecto de *software*?

- a. Reutilizar estándares y no reinventar la rueda.
- b. Aplicar la ley del mínimo esfuerzo.
- c. Aprender de experiencias previas mediante lecciones aprendidas.
- d. Seguir procesos en secuencia definidos junto al cliente.

7. Al aplicar principios de ingeniería de *software*, el resultado esperado es:

- a. Proyectos más eficientes.
- b. Equipos de trabajo motivados.
- c. *Software* de calidad.
- d. Documentación extensa y detallada.

8. El modelado en ingeniería de *software* se utiliza principalmente para:

- a. Reducir la complejidad de los sistemas.
- b. Generar representaciones gráficas del sistema.
- c. Facilitar la comunicación del equipo.
- d. Identificar problemas en la codificación.

9. ¿Cuándo resulta más conveniente el uso de modelos de *software*?

- a. En sistemas pequeños y fáciles de implementar.
- b. En sistemas con baja complejidad.
- c. En cualquier proyecto de *software*.
- d. En sistemas de gran tamaño y alta complejidad.

10. El modelo de dominio de aplicación se construye en la fase de:

- a. Implementación del sistema.
- b. Diseño arquitectónico.
- c. Diseño Orientado a Objetos.
- d. Levantamiento de requerimientos y análisis.

[Ir al solucionario](#)

[2] Podrá encontrar en la sección de materiales el caso de estudio TRAMIL.

## Resultado de aprendizaje 2:

Conoce e Identifica las fases del ciclo de vida de desarrollo de sistemas.

Para alcanzar el resultado de aprendizaje 2, se plantea la familiarización con las fases del ciclo de vida del desarrollo de sistemas. Se examinarán los diversos modelos de procesos de software y las metodologías de desarrollo, tanto tradicionales como ágiles. Este conocimiento permitirá identificar y aplicar adecuadamente cada fase del ciclo de vida en la gestión de proyectos de software, asegurando una planificación efectiva y una entrega exitosa del producto final.

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.

### Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



### Semana 2

Durante la primera semana, se realizó una introducción a lo que es la ingeniería de software, se analizaron algunos de los principios y se describió de manera breve algunas de las actividades de desarrollo desde la perspectiva Orientada a Objetos, sin embargo, en la práctica, los proyectos de desarrollo de software se abordan siguiendo un proceso de desarrollo de software, el cual consiste en primer lugar en un enfoque que en general puede ser considerado tradicional o predictivo y ágil.

En esta semana vamos a abordar los procesos de desarrollo de software más comunes, e intentaremos establecer los criterios necesarios para seleccionar uno u otro en función de las características del proyecto.

## Unidad 2. Procesos de software

---

### 2.1. Modelos de proceso de software

Un proceso de software es “un conjunto coherente de políticas, estructuras organizativas, tecnologías, procedimientos y artefactos que se necesitan para concebir, desarrollar, implantar y mantener un producto de software” y, un modelo de proceso es “una definición de alto nivel de las fases por las que transcurren los proyectos de desarrollo de software” (Melissa A. Russell, 2018), a los modelos de proceso también se los denomina como Ciclo de Vida del Desarrollo de Software (CVDS).

Un ingeniero o equipo de ingenieros de Software para resolver problemas reales de las organizaciones debe incorporar una estrategia de desarrollo que acompañe al proceso, métodos y herramientas; esta estrategia se llama modelo de procesos. Por lo tanto, debe seleccionar un modelo de procesos para la ingeniería de software según la naturaleza del proyecto y de la aplicación, los métodos y herramientas a utilizarse, los controles y entregas que se requieren (Sommerville, 2016).

En la figura 3, se especifican las actividades generales que todo proceso de desarrollo de software considera. Es importante recalcar que, al igual que las actividades, al momento de describir un proceso software, también se deben considerar los productos, roles y las precondiciones y postcondiciones.

**Figura 3**

Actividades del Desarrollo de Software



Nota. Jaramillo, D., 2025.

El proceso de desarrollo de software no es único y al no existir un proceso universal que sea efectivo para todos los contextos de proyectos de desarrollo, existe una diversidad de actividades y metodologías que hacen difícil automatizar todo un proceso de desarrollo de software.

Las actividades relacionadas al proceso de desarrollo difieren de acuerdo al entorno y muchos autores recomiendan actividades puntuales, por ejemplo, en (Sommerville, 2016) considera cuatro actividades: especificación, diseño e implementación, validación y evolución del software. Mientras que (Pressman, 2010) considera cinco actividades: comunicación, planeación, modelado, construcción y despliegue. De la misma manera (Bruegge & Dutoit, 2018) consideran seis actividades: obtención de requisitos, análisis, diseño del sistema, diseño de objetos, implementación y pruebas. Pero esto no quiere decir que son actividades totalmente distintas con enfoques diferentes sino más bien que cada una de estas actividades coincide en un alto porcentaje y las diferencias están en el enfoque y particularidades que cada autor desea que forme parte del proceso de desarrollo de software.

Vamos a determinar las diferencias entre tres modelos (cascada, incremental y orientado a la reutilización) y poder responder a las siguientes preguntas:

- ¿Son realmente diferentes los modelos?
- ¿Cuál modelo es el más adecuado para el desarrollo de software?

- ¿Elegir un modelo de desarrollo dependerá del tipo de problema?
- ¿Se pueden combinar los modelos?

A continuación, se analiza cada uno de estos modelos, con el objeto de profundizar en cada una de las actividades que forman parte del proceso, y especialmente argumentar de mejor manera la respuesta a las interrogantes anteriormente planteadas.

### 2.1.1. El modelo en cascada

Es el más básico de todos y ha servido como bloque de construcción para los demás paradigmas de ciclo de vida. Está basado en un ciclo convencional de una ingeniería y su visión es muy simple: el desarrollo de software se debe realizar siguiendo una secuencia de fases. Cada etapa tiene un conjunto de metas bien definidas y las actividades dentro de cada una contribuyen a la satisfacción de metas de esa fase. Leído el tema se podría concluir que la ventaja del modelo radica en su sencillez, ya que sigue los pasos intuitivos necesarios a la hora de desarrollar el software. Sin embargo, al aplicar a un determinado contexto puede ocurrir que:

- Los proyectos reales raramente siguen un flujo secuencial que propone el modelo. Siempre hay iteraciones y se crean problemas en la aplicación del paradigma. Normalmente, al principio, es difícil para el cliente establecer todos los requisitos de forma explícita.
- El cliente debe ser paciente ya que hasta que no llegue a etapas finales del proyecto no estará disponible una versión operativa del programa. Un error importante que no pueda ser detectado hasta que el programa esté funcionando, puede ser desastroso.

Por lo tanto, el modelo se puede utilizar para cierto tipo de problemas, donde los requisitos previamente ya estén establecidos y comúnmente no existan cambios.

## 2.1.2. Modelo incremental

El modelo incremental combina elementos del modelo en cascada con la filosofía interactiva de construcción de prototipos. Este modelo aplica secuencias lineales de forma escalonada mientras transcurre el tiempo donde cada secuencia lineal produce un incremento del software.

Terminada la lectura, podemos concluir que este modelo con respecto al modelo en cascada se caracteriza por la reducción de costos, sencillez al interactuar con el cliente y entregas más rápidas.

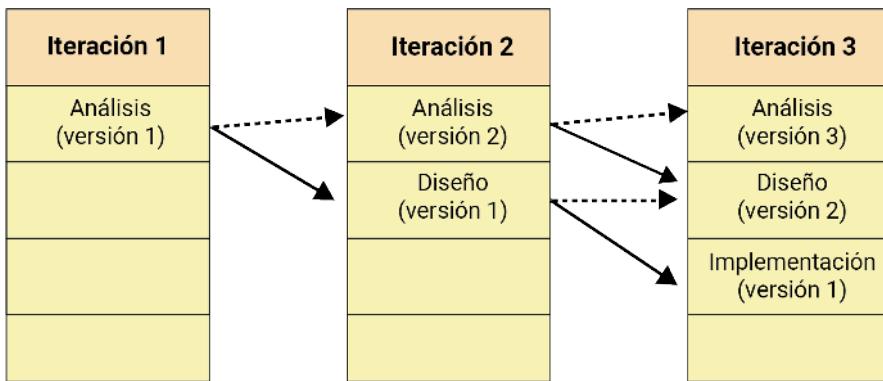
La mayoría de los proyectos de desarrollo de software se complementan con un enfoque “Iterativo”, de ahí que muchos autores sugieren el desarrollo “iterativo e incremental”, que consiste en planificar el desarrollo en diversos bloques temporales llamados iteraciones, dónde cada iteración se puede entender como miniproyectos.

En todas las iteraciones se repite un proceso de trabajo similar para proporcionar un resultado completo sobre el producto final, de manera que el cliente pueda obtener los beneficios del proyecto de forma incremental. Para ello, cada requisito se debe completar en una única iteración, el equipo debe realizar todas las tareas necesarias para completarlo (incluyendo pruebas y documentación) y que esté preparado para ser entregado al cliente con el mínimo esfuerzo necesario. De esta manera no se deja para el final del proyecto ninguna actividad arriesgada relacionada con la entrega de requisitos.

En la figura 4 se muestra la secuencia de las actividades del modelo Iterativo e Incremental, dónde con cada iteración el equipo evoluciona el producto a partir de los resultados completados en las iteraciones anteriores, añadiendo nuevos objetivos/requisitos o mejorando los que ya fueron completados. Un aspecto fundamental para guiar el desarrollo iterativo e incremental es la priorización de los objetivos/ requisitos en función del valor que aportan al cliente.

**Figura 4**

Proceso Iterativo e Incremental



Nota. Adaptado de *Object-oriented software engineering : using UML, patterns, and Java [Ilustración]*, por Bruegge, B. y Dutoit, A., 2018, Prentice Hall, CC BY 4.0.

### 2.1.3. Modelo de Ingeniería del Software orientada a la reutilización

Es el proceso de creación de sistemas de software a partir de un software existente, en lugar de tener que rediseñar desde el principio. El diseño basado en reutilización busca construir un producto software integrando componentes pre-existentes.

Se puede determinar que este modelo necesita de información del software o componente existente, es así que para desarrollar la primera actividad "Análisis de componentes", es necesario la especificación de requerimientos del software desarrollado, es decir, el punto de partida de las actividades del modelo son los requerimientos implementados.

Como se ha manifestado el modelo se puede utilizar en proyectos donde existan componentes desarrollados que se los pueda reutilizar, consecuentemente se requiere de información que en muchas ocasiones no existe, teniendo que el equipo de desarrollo enfrentarse a las siguientes dificultades:

- En muchas empresas no existe plan de reutilización (No se considera prioritario)

- Escasa formación del personal
- Resistencia del personal al uso de modelos basados en reutilización
- Pobre soporte metodológico del modelo
- Uso de métodos que no promueven la reutilización (Estructurados)
- Es necesario la implementación de métodos para:
  - Desarrollo para reutilización
  - Desarrollo con reutilización
- Los más importante ¿Quién soporta los gastos adicionales de la reutilización?

Complemente el estudio de este tema revisando el Tema 3. Procesos de Ingeniería de Software del REA **Ingeniería de Software I**. El documento analiza los ciclos de vida tradicionales, el ciclo de vida orientado a objetos y las metodologías de desarrollo de software.

Luego de la lectura y análisis se podrá dar cuenta que no solamente existen los tres modelos que hemos analizado, sino que existen otros modelos para el desarrollo de software como es: RAD, Espiral, Prototipos, entre otros. De estos modelos ¿Cuál es el más adecuado?, sin duda alguna que para responder de forma adecuada se necesitan de elementos adicionales que permitan elegir de forma coherente el modelo. A continuación, se presenta una lista de factores que afectan la elección del modelo de software:

- Naturaleza del proyecto
- Tamaño del proyecto
- Duración del proyecto
- Complejidad del proyecto
- Nivel y tipo de riesgo esperado
- Nivel de comprensión de los requisitos de usuario
- Nivel de comprensión del área de aplicación
- Involucramiento del cliente
- Experiencia de los desarrolladores
- Tamaño del equipo
- Interacción hombre-máquina

- Disponibilidad de herramientas y tecnología
- Versión del producto
- Nivel de consistencia requerido

En la tabla 6, se muestra una comparativa de los modelos más comunes y que se podrían utilizar en el desarrollo del software.



**Tabla 6**

Comparación de varios modelos de procesos con diferentes parámetros

Parámetros	Modelos de procesos					
	Cascada	Incremen-tal	Prototipos	RAD	Espiral	Ágil
Claridad en la especificación de requisitos	Nivel inicial	nivel inicial	a medio	nivel inicial	nivel inicial	cambia de forma incremental
Comentarios del usuario	No	No	Si	No	No	No
Velocidad para cambiar	Bajo	Alto	Medio	No	Alto	Alto
Previsibilidad	Bajo	Bajo	Alto	Bajo	Medio	Alto
Identificación de Riesgos	A nivel inicial	no	no	No	si	Si
Implementación Práctica	No	Bajo	Medio	No	Medio	Alto
Comprendibilidad	Simple	Intermedio	Intermedio	Intermedio	Difícil	muy complejo
Usabilidad	Básica	Media	Alta	Media	Media	Alta
Prioridad del cliente	nulo	nulo	intermedio	nulo	intermedio	alto
Costo	Bajo	bajo	alto	muy alto	costoso	muy costoso
organización de recursos	si	si	Si	si	no	no
Elasticidad	No	no	Si	si	no	muy alto

*Nota.* Jaramillo, D., 2025.

## 2.2. Ciclo de vida del Software

Para dar más luces sobre la selección de un modelo de proceso adecuado, es muy importante tener en cuenta el nivel de incertidumbre del proyecto, y para ello tomaremos como referencia la Guía de Agilidad (Melissa A. Russell, 2018), que establece 4 tipos de ciclos de vida y además recomienda un enfoque de acuerdo al nivel de incertidumbre de los proyectos, como se indica en el siguiente componente didáctico:

### Ciclos de vida

En la tabla 7 de la presente guía, se presentan algunas de las características de cada una de estas categorías de los ciclos de vida.



**Tabla 7**

Características de las diferentes categorías de ciclos de vida

Características				
Enfoque	Requerimientos	Actividades	Entregas	Metas
Predictivo	Fijos	Se ejecutan una sola vez durante todo el proyecto.	Una sola	Gestionar costos
Iterativo	Dinámicos	Se repite hasta que los resultados sean correctos.	Una sola	Solución correcta.
Incremental	Dinámicos	Se ejecuta una vez por incremento.	Entregas pequeñas frecuentes	Velocidad
Ágil	Dinámicos	Se repite hasta que sea correcto.	Entregas pequeñas frecuentes.	Valor del cliente a través de entregas y retroalimentación.

Nota. Adaptado de *Project Management Body Of Knowledge and Agile Practice Guide*, por Project Management Institute, 2017, Pennsylvania.

Continuemos con el aprendizaje mediante la revisión de las metodologías ágiles vs. tradicionales.

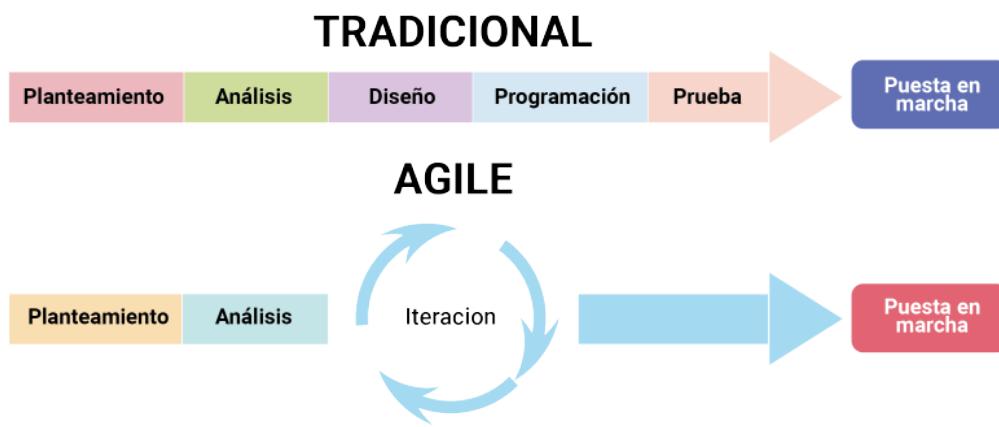
## Metodologías Ágiles vs Tradicionales

### 2.3. Metodologías ágiles vs. tradicionales

En la figura 5 se muestra la diferencia de actividades entre el modelo tradicional y el modelo ágil. Claramente, se puede apreciar que en el modelo tradicional se desarrolla todo el proyecto, mientras que en el ágil cada iteración es una parte del software, y cada parte se pone en marcha.

Figura 5

*Modelo tradicional versus modelos ágiles*



Nota. Jaramillo, D., 2025.

#### 2.3.1. Metodologías de desarrollo tradicionales

Una vez estudiados los diferentes modelos de proceso, habrá notado que las actividades genéricas para el desarrollo de software son análisis, diseño, implementación, pruebas, y entrega, sin embargo, existen muchas metodologías, sobre todo las que tienen enfoque ágil cuyas actividades, aunque tienen similitudes, tienen diferencias significativas.

Una metodología es “una forma de hacer o adoptar un modelo para ejecutar una tarea o un conjunto de tareas, de modo que el objetivo de la tarea se logre como se esperaba” (Ahmed A & Bhanu P, 2018) los cuales pueden ser aplicados en función de las circunstancias del trabajo que se tiene que

desarrollar. Considere como ejemplo la situación de construir un condominio de departamentos, en función de los requerimientos se tiene dos opciones, por un lado, se puede optar por utilizar estructuras prefabricadas y luego ensamblarlas para construir el edificio, o puede realizar una construcción con el método tradicional en el cual se construye todo con los materiales y herramientas básicas, en este caso la meta es la misma y los enfoques para conseguirlo son diferentes. La decisión sobre qué método seleccionar depende de los requerimientos del método, así como de las consideraciones tales como el plazo de entrega, los costos, la seguridad y la disponibilidad de los recursos y la tecnología. De la misma forma, una aplicación de software puede ser desarrollada utilizando metodologías conocidas o cualquier otra disponible para los desarrolladores.

En este apartado estudiaremos algunas metodologías de desarrollo de software que trabajan bajo el esquema predictivo o tradicional, recuerde que no se trata de profundizar en ellas, sino de conocer cuál es su enfoque, sus etapas, sus ventajas y desventajas, de modo que usted pueda decidir cuál sería la más conveniente para un proyecto específico.

#### a. Cascada (waterfall model)

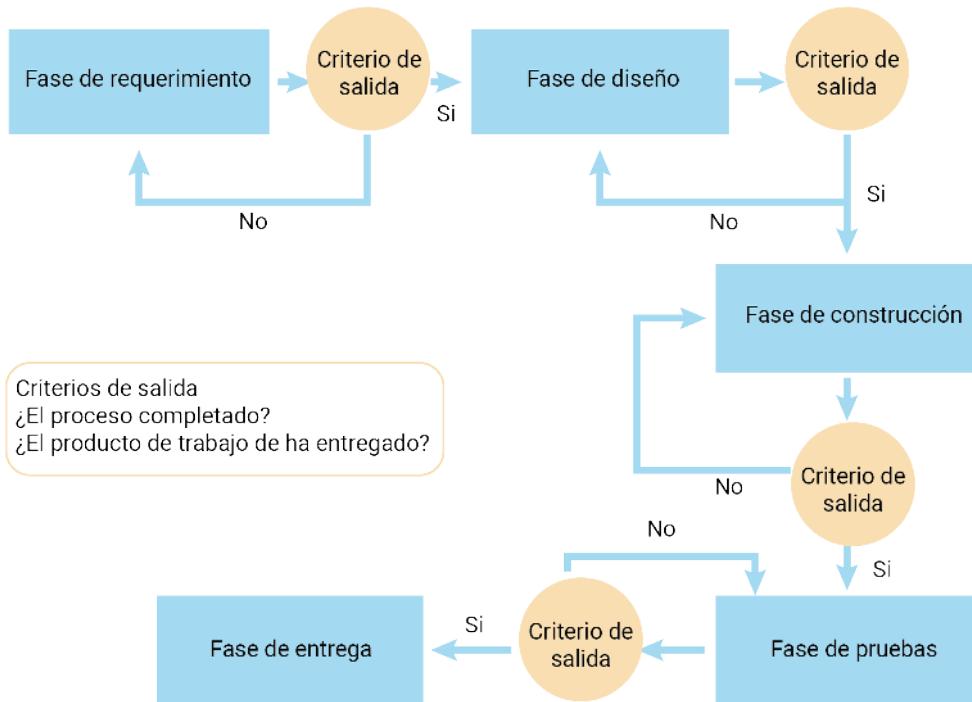
Fue documentado por primera vez por Benington Herber D., en 1956 y en 1970 fue modificado por Winston Royce. El propósito de esta metodología era reducir los costos de desarrollo, la especificación inicia a muy alto nivel hasta llegar a los niveles más bajos, a este enfoque se lo conoce como TOP-DOWN. Esta metodología fue la primera en usarse formalmente para el desarrollo de software, promueve la operación secuencial de sus procesos con planes claramente definidos. Ello ha llevado a que se denomine como dirigida por un plan.

En el modelo original de la metodología, cada paso se realizaba de manera secuencial, y no se avanza al siguiente, a no ser que se cumplan los criterios de salida del anterior, esto se puede apreciar en la figura 6. Los

criterios de salida están asociados a dos aspectos importantes, el primero es que los artefactos se hayan completado y el segundo es que estos hayan sido aceptados por parte del cliente.

**Figura 6**

*Control de calidad en el modelo en cascada*

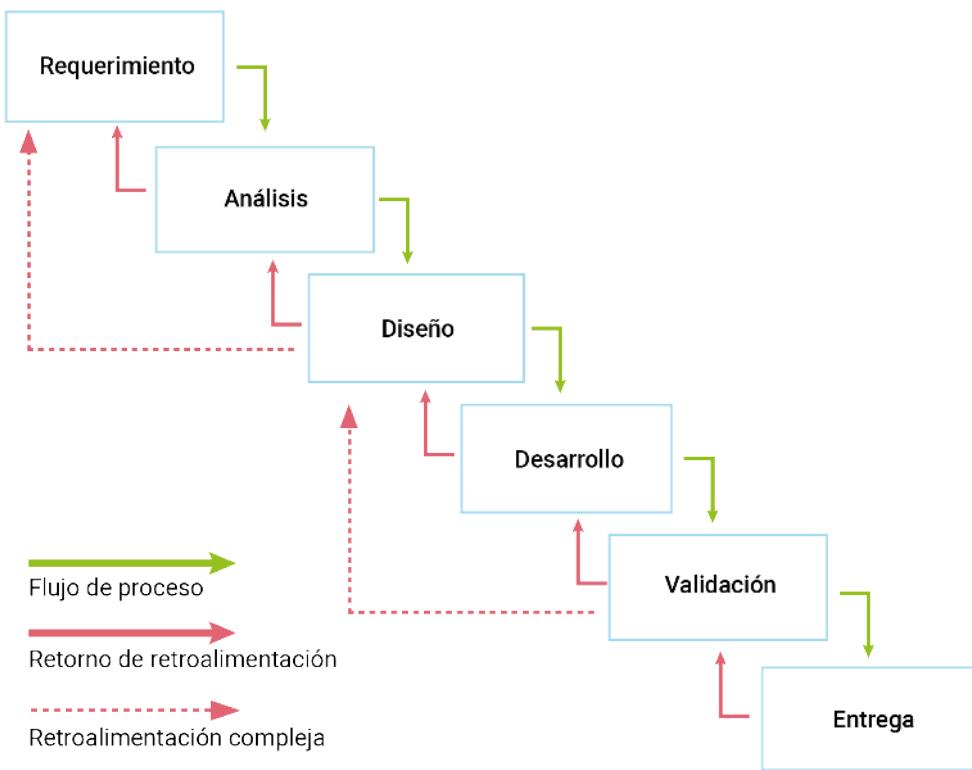


Nota. Adaptado de *Software Engineering in the Agile World* [Ilustración], por Ahmed, A. y Bhanu, P., 2018, Ndently Published, CC BY 4.0.

En la figura 7 se aprecia la versión de 1970 del modelo en cascada. La versión de Royce mejora el modelo original incluyendo bucles de retroalimentación de forma que la fase anterior puede ser revisitada.

**Figura 7**

Modelo en cascada de Royce



Nota. Adaptado de *Software development lifecycle models* [Ilustración], por Ruparelia, N., 2016, ACM SIGSOFT Software Engineering Notes, CC BY 4.0.

El modelo en cascada opera de la siguiente manera:

- En la fase de requerimientos, el equipo del proyecto se enfoca en realizar un gran esfuerzo en recolectar todos los requerimientos del cliente y de los usuarios, luego son documentados en un proceso exhaustivo, finalmente son validados y aprobados por el cliente, con ellos se elabora el documento de Requerimientos de Software, y una vez completado se usa como insumo para la etapa de diseño.
- En la fase de análisis, se desarrollan especificaciones detalladas del sistema y se produce la documentación necesaria para que los diseñadores puedan realizar el diseño preliminar del sistema, en caso

de inconsistencias o de que algo no quedó claro, se puede regresar a la fase de requerimientos y modificar la documentación correspondiente, con ello se revisaría nuevamente el análisis. Este documento de análisis también tiene que ser aprobado antes de pasar a la fase de diseño.

- En la fase de diseño, los diseñadores elaboran varios modelos del sistema y generan la documentación correspondiente que consiste en: el documento de diseño de interfaces, diseño de componentes, diseño de datos. Al igual en la fase anterior, en caso de encontrar fallos en el diseño durante esta fase, se regresa a la fase de análisis y en un escenario más complejo, se podría regresar a la fase de requerimientos.
- En la fase de desarrollo se construye el código de la aplicación, el resultado es el software terminado, probado y documentado, la siguiente fase que es la validación.
- En la fase de validación, se realizan las pruebas correspondientes para determinar si el sistema cumple con las especificaciones, y en caso de que no lo haga, se puede regresar a la fase de desarrollo y si el problema es más complejo, se regresaría a la fase de diseño.
- Una vez que se han completado exitosamente estas etapas, el sistema se entrega al cliente y usuarios, se entregan manuales, se preparan los datos para la operación del sistema y se capacita a los usuarios para su uso.

Es importante mencionar que, si bien es un modelo sólido, no es la mejor alternativa, o al menos no para cualquier proyecto, a continuación, se resumen algunas de sus ventajas y desventajas:

### **Ventajas:**

- Se planifica todo el proyecto al inicio, se establecen líneas base, hitos y plazos de entrega, ello facilita el monitoreo de proyecto, mejora la visibilidad de los procesos de desarrollo.
- Se puede obtener reportes de estado en cualquier momento durante la ejecución, y los interesados pueden tomar decisiones respecto de

cualquier cambio para asegurar que el proyecto avanza sin contratiempos. Los reportes de estado pueden ser internos y externos, los externos son para los interesados, y con base en ellos pueden solicitar cambios.

- Soporta controles de calidad: al finalizar cada fase, los artefactos desarrollados se pueden probar, y solo pasa a la siguiente fase si los artefactos cumplen con las especificaciones de las pruebas.
- Es adecuado para el desarrollo de productos muy grandes: Permite que el equipo sea de cualquier tamaño, desde unas pocas personas hasta cientos de ellas, y de ser necesario un proyecto muy grande puede ser desarrollado en corto tiempo por muchas personas.
- El modelo implica mucha documentación, por tanto, si un miembro del equipo abandona el proyecto, puede usar la documentación para ponerse al tanto de lo realizado y continuar con lo nuevo.

### **Desventajas:**

- Tiene una gran carga de trabajo en planificación, debido a que el producto se desarrolla por completo y no de manera incremental, y, por tanto, requiere muchas personas trabajando en el equipo.
- La carga operativa en cuanto a la gestión de personal es muy grande.
- La mayoría de las veces los usuarios no tienen conocimientos técnicos, por tanto, los requerimientos pueden tener muchas ambigüedades, y, en consecuencia, la planificación puede ser fallida.
- Todos los requerimientos deben congelarse al inicio del proyecto, ya que se usan como base para la planificación, los cambios en etapas avanzadas del proyecto resultan demasiado costosos, puesto que se necesita volver a etapas iniciales para implementarlos. En la realidad los clientes y usuarios suelen cambiar muchos requerimientos una vez iniciado el proyecto, por tanto, hoy en día es uno de los procesos menos convenientes.

## Cuando usar la metodología:

- En proyectos en los cuales se requiere que el proyecto se termine de acuerdo a un cronograma preestablecido.
- Cuando se requiere documentación muy formal.
- Proyectos sencillos, con poca incertidumbre baja, y bajo nivel de riesgo.

Continuemos aprendiendo otras metodologías de desarrollo tradicionales.

### b. Proceso unificado de desarrollo RUP

El Proceso Unificado de Desarrollo (RUP por sus siglas en inglés), fue creado por Grady Booch, Ivar Jacobson, y James Rumabugh, en 1998 con el acrónimo de RUP (Rational Unified Process), cuando trabajan para Rational Software como respuesta a los problemas existentes en la época con los procesos de desarrollo existentes.

De acuerdo a (Kroll & Kruchten, 2003) para crear RUP, sus autores se plantearon los principales problemas con el desarrollo de software, establecieron sus causas raíces y propusieron las mejores prácticas Rational de la ingeniería de software.

Los principales problemas o síntomas son los siguientes:

- Escasa comprensión de las necesidades de los usuarios finales.
- Imposibilidad de resolver los requerimientos cambiantes.
- Módulos que no empatan entre sí.
- El software muy difícil de mantener o extender.
- Descubrimiento tardío de fallas serias en el proyecto.
- Calidad del software muy pobre.
- Rendimiento del software
- Miembros del equipo que no colaboran entre sí, haciendo imposible saber reconstruir, o saber quién hizo, quién cambió, cuándo, cómo y por qué.

Proceso de compilación, construcción y liberación no confiable.

Luego identificaron las causas raíces de esos problemas, que se pueden resumir en las siguientes:

- Gestión de requerimientos improvisada.
- Comunicación imprecisa o ambigua.
- Arquitectura débil.
- Complejidad excesiva.
- Inconsistencias no detectadas en los requisitos, diseños e implementaciones.
- Pruebas insuficientes.
- Evaluación subjetiva del estado del proyecto.
- Fallos al momento de enfrentar riesgos.
- Propagación de cambios descontrolada.
- Automatización insuficiente.

Para resolver estos problemas, los autores propusieron algunas prácticas de la ingeniería de software que se detallan en el siguiente módulo didáctico:

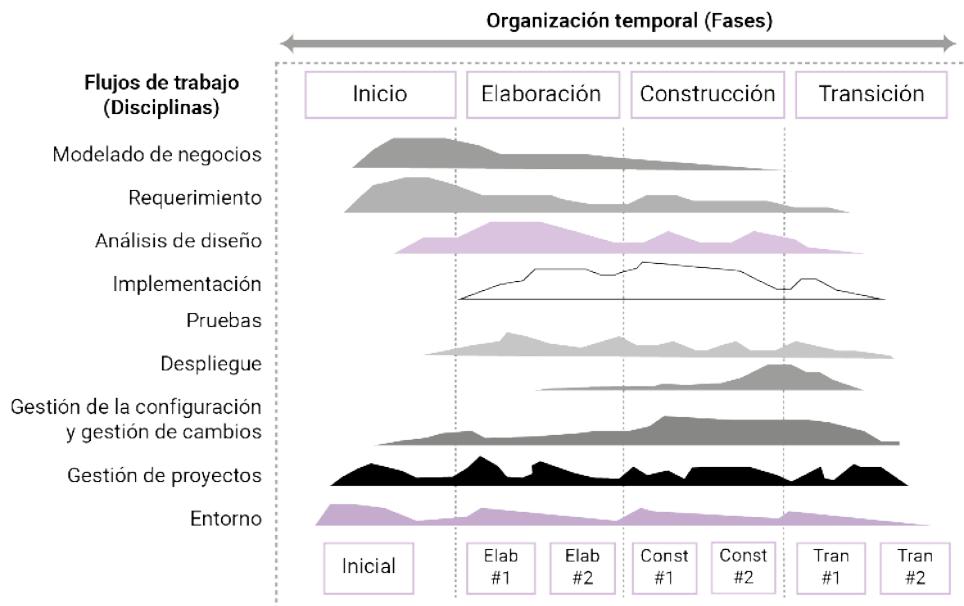
### Prácticas de la ingeniería de software

El RUP está organizado en dos dimensiones, horizontalmente podemos distinguir 4 fases: inicio, elaboración, construcción y transición, y verticalmente consta de nueve flujos de trabajo o disciplinas, en cada fase a partir de la segunda (elaboración) se desarrollan varias iteraciones, en las cuales se realizan todos los procesos establecidos en las disciplinas y producen en resultado ejecutable.

En la fase de inicio se busca definir el alcance del proyecto, y como artefacto resultante se obtiene un documento de visión, en la fase de elaboración se inicia el desarrollo, los artefactos resultantes son los programas, la documentación y el artefacto que marca el fin de la fase es la estabilización de la arquitectura de la aplicación, la tercera fase es la construcción, el resultado para el usuario son los componentes de la aplicación ya completos, la documentación asociada y en la fase de transición se completan todos los entregables, la documentación, se

capacita a los usuarios y se cierra el proyecto. Todas las disciplinas se ejecutan en todas las fases del proceso RUP, y la intensidad de trabajo requerido en cada disciplina, se ilustra con las montañas que recorren cada fase. Este proceso se ilustra en la figura 8.

**Figura 8**  
*El Proceso Unificado Rational (RUP)*



Nota. Adaptado de *The Rational Unified Process: An introduction* [Ilustración], por Krutchen, P., 2004, Massachusetts: Addison Wesley, CC BY 4.0.

### Ventajas:

- RUP admite el concepto de flujo de trabajo, y estos se pueden ejecutar en todas las iteraciones del proceso, por lo tanto, se produce un proceso iterativo incremental, que con cada iteración completa y mejora la aplicación resultante.
- La aplicación de las mejoras prácticas de la ingeniería de software, permite elevar la calidad de los resultados.

## **Desventajas:**

- La ejecución de todas las disciplinas resulta muy pesada en términos de mantenimiento de las fases del proyecto, para trabajar en RUP se requiere un equipo grande de personas.
- RUP exige mucha documentación.



## **Cuando utilizar RUP:**

- El RUP es un proceso de desarrollo robusto, que al basarse en iteraciones permite refinar el producto conforme avanza en el tiempo.
- Es muy recomendable cuando los niveles de incertidumbre y riesgo son altos y se requiere mucha formalidad en el proyecto, esto debido a la documentación.
- A pesar de que incluye procesos iterativos e incrementales, RUP no se considera un proceso de desarrollo ágil, debido a la cantidad de documentación que exige y a la cantidad de roles que deben operar, sin embargo, esto lo hace ideal para proyectos muy grandes.

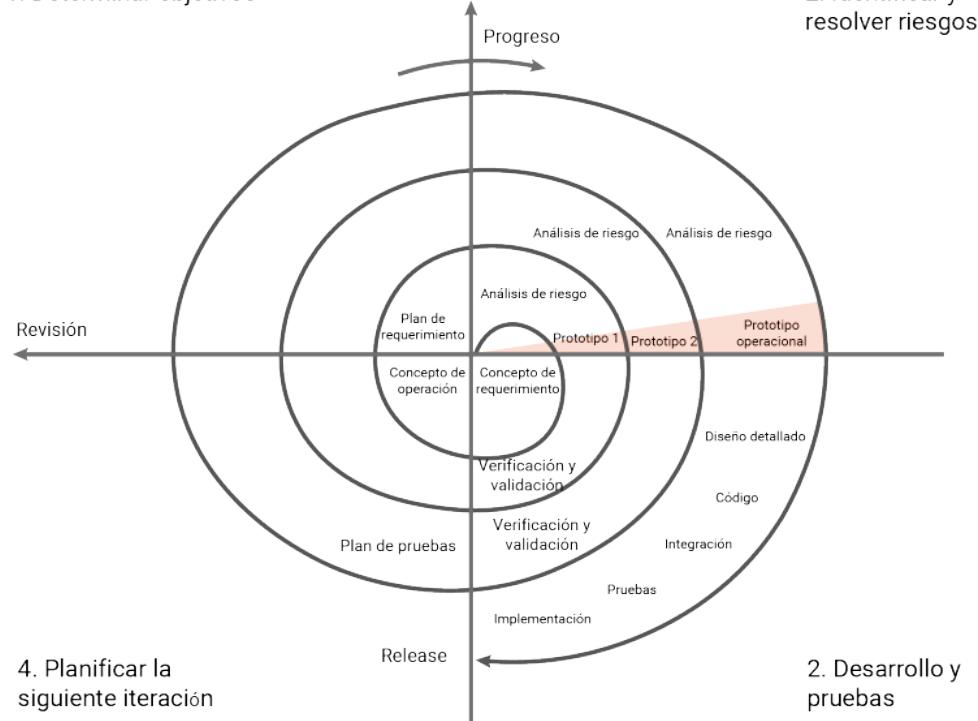
## **c. Modelo en espiral**

En 1986, Boehm, modificó el modelo en cascada, introduciendo varias iteraciones que se surgen en espiral hacia afuera, comenzando con pequeños incrementos, aplicando la filosofía start, small, think big (Ruparelia, 2016). El modelo en espiral representa un cambio en el paradigma orientado a especificaciones del modelo en cascada a uno dirigido por riesgos (ver figura 9, el modelo en espiral de Boehm).

**Figura 9**

*El modelo Espiral*

1. Determinar objetivos



Nota. Adaptado de *Software development lifecycle models* [Ilustración], por Ruparelia, N., 2016, ACM SIGSOFT Software Engineering Notes, CC BY 4.0.

En la figura 9 se aprecia el modelo en espiral, en él cada ciclo pasa por cuatro cuadrantes que son:

- Determinar objetivos.
- Evaluar alternativas e identificar y evaluar riesgos.
- Desarrollo y pruebas, y.
- Planificar la siguiente iteración.

Conforme avanza cada ciclo, se construye un prototipo, que se verifica contra los requerimientos y es validado mediante pruebas. Previo a ello, se identifican y analizan los riesgos con el fin de gestionarlos, luego estos son categorizados como riesgos relacionados con el rendimiento o

desarrollo. Si predominan los riesgos relacionados al desarrollo, el siguiente paso se realiza con el enfoque incremental del enfoque en cascada. Caso contrario, si predominan los riesgos de rendimiento, el siguiente paso es continuar con la espiral hasta el siguiente ciclo evolucionario. Esto asegura que el prototipo del segundo cuadrante tiene riesgos mínimos asociados a él. En este modelo, la gestión del riesgo se usa como base para determinar el tiempo y el esfuerzo que se usará para todas las actividades durante el ciclo, y también se usa para controlar los costos.

De acuerdo a (Ahmed, 2016), el modelo en espiral no es un modelo ágil, ya que el modelo en espiral no produce una versión ejecutable del sistema en cada iteración, si no prototipos que se usan para demostrar al cliente lo que el equipo está desarrollando.

#### **Ventajas:**

- En el modelo en espiral, se considera como buena práctica el construir un buen prototipo antes de construir el producto final, mismo que es refinado durante cada iteración.
- El cliente puede dar su retroalimentación respecto de cada prototipo y provee más información sobre partes equivocadas o faltantes en el producto.
- Una vez que el cliente ha dado su aprobación sobre los prototipos, el equipo comienza a desarrollar el producto.

#### **Desventajas:**

- El modelo en espiral puede resultar costoso por las muchas iteraciones para obtener el producto de software.
- Puede haber productos que se desarrollen en 1 sola iteración con los cual este modelo sería excesivo.

Cuando utilizar el modelo en espiral:

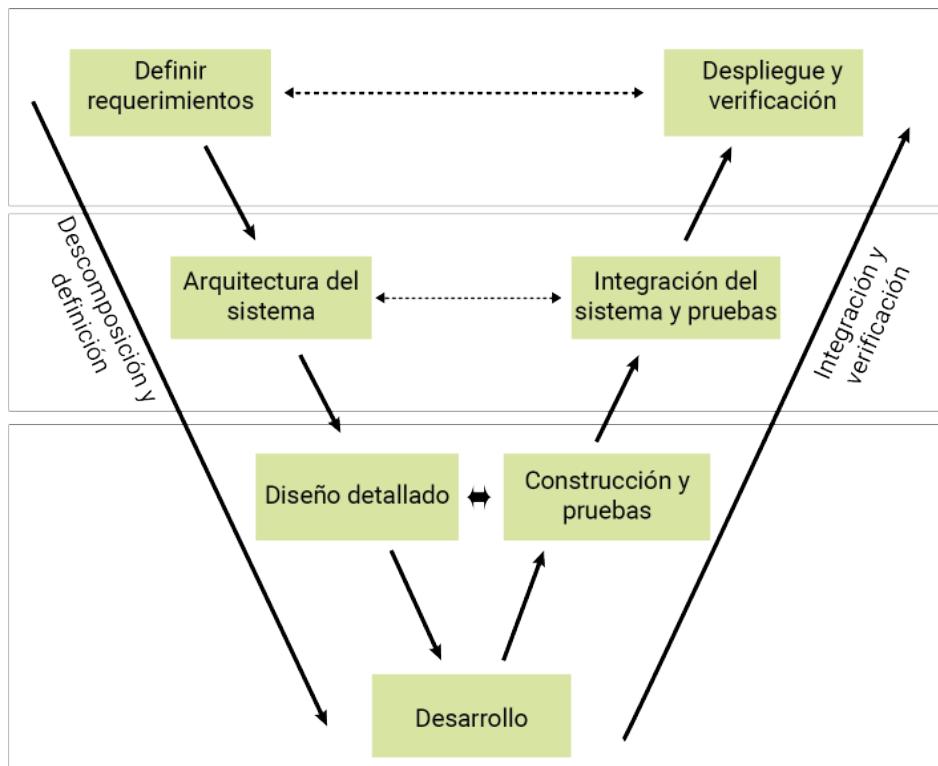
El modelo en espiral es conveniente utilizarlo para modelos en los que los niveles de riesgo son demasiado altos.

#### d. El modelo en V

Fue desarrollado por la NASA, se presenta como una variación del modelo en cascada se presenta en forma de una V, la parte de izquierda representa la evolución de los Requerimientos de Usuario en componentes más pequeños a lo largo del proceso de descomposición y definición, la parte derecha representa la integración de los componentes del sistema en niveles sucesivos de implementación y ensamblaje. En la figura 10 se aprecia este modelo. Visto verticalmente el modelo representa el nivel de descomposición del sistema, en la parte superior el sistema y en el fondo en nivel más bajo de descomposición.



**Figura 10**  
*El modelo en V*



Nota. Adaptado de *Software development lifecycle models* [Ilustración], por Ruparelia, N., 2016, ACM SIGSOFT Software Engineering Notes, CC BY 4.0.

Una característica del modelo en V es que es simétrico a lo largo de ambas ramas, por tanto, los procesos de verificación y aseguramiento de la calidad están definidos para la rama derecha durante el paso correspondiente en la rama izquierda, con ello es posible asegurar que los requerimientos y el diseño son verificables de forma específica, medible, alcanzable, realista y limitada en el tiempo (SMART por sus siglas en inglés).

## **Ventajas:**

- Introduce verificaciones de cada una de las fases, lo cual facilita la localización de fallos, esto lo hace mucho más robusto y completo que el modelo en cascada.
- El modelo es sencillo y fácil de aprender.
- Se hace explícita la iteración y el trabajo que se debe realizar, lo que produce un software de mejor calidad que en cascada.
- El cliente puede dar su retroalimentación y se involucra en las pruebas.

## **Desventajas:**

- A pesar de la retroalimentación del cliente, es difícil que el cliente exponga todos los requisitos.
- Al igual que en el modelo en cascada, el producto se entrega al finalizar el ciclo de vida.
- La metodología no contempla la posibilidad de regresar a etapas inmediatamente anteriores, lo cual sí puede ser necesario.

## **Cuando utilizar el modelo en V:**

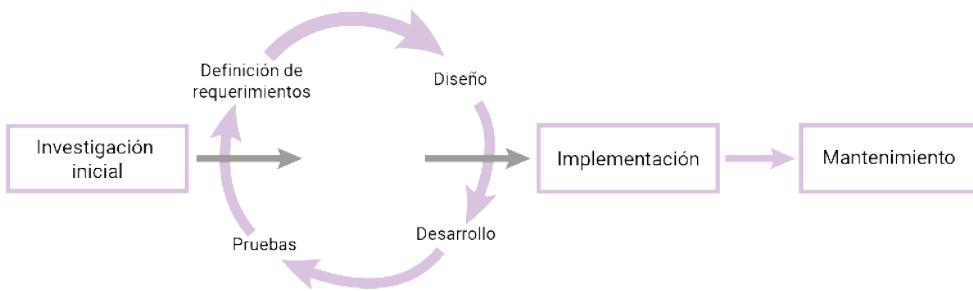
El modelo en V es una mejora al modelo en cascada, por lo que su uso es recomendado para aplicaciones de complejidad en incertidumbre baja, pero que requieran alta confiabilidad.

## **e. Desarrollo Rápido de Aplicaciones (RAD)**

Desarrollada por James Martin en 1991, utiliza prototipos como mecanismo de desarrollo iterativo, promueve una atmósfera colaborativa donde participan activamente los interesados del negocio en la creación de prototipos, casos de prueba y ejecutando pruebas unitarias. El esquema de trabajo de RAD se ilustra en la figura 11.

**Figura 11**

Metodología Rapid Application Development (RAD)



Nota. Adaptado de *Software development lifecycle models* [Ilustración], por Ruparelia, N., 2016, ACM SIGSOFT Software Engineering Notes, CC BY 4.0.

Uno de los aportes más significativos ha sido la inclusión de procesos iterativos e incrementales, gracias a ellos se consigue entregar valor al cliente lo más pronto posible y el equipo consigue entregar a tiempo un software que se adapta a las necesidades de los usuarios, a diferencia de los modelos predictivos o tradicionales que retrasan las entregas a los usuarios hasta el final del proyecto, con lo cual muchas cosas pudieron cambiar en el entorno del cliente.

Quedan sin mencionar metodologías como el modelo basado en prototipos, Análisis y Diseño Orientado a Objetos (OOAD) de Grady Booch y Object, Object Modeling Technique (OMT), las cuales han servido de base para el desarrollo del Lenguaje Unificado de Modelado y el Proceso Unificado de Desarrollo.

### 2.3.2. Metodologías agiles

En la década de los 90, el uso de procesos de desarrollo predictivos se había extendido y funcionaban bien con proyectos muy grandes y costosos, sin embargo, había mucho descontento por los tiempos de entrega y la cantidad de cambios que surgían en los clientes y usuarios. Para (Ahmed, 2016) la presencia de *Internet* hizo que muchas empresas consideran su potencial para los negocios, y contrataron el desarrollo de muchos sitios web, que con los métodos tradicionales no facilitaban el trabajo, esto llevó a la comunidad de

desarrolladores a pensar en métodos alternativos de desarrollar el trabajo que reduzcan los tiempos de entrega y eliminen obstáculos presentes en las metodologías tradicionales.

Esta situación llevó a un grupo de desarrolladores a crear una filosofía denominada Manifiesto Ágil, el cual establece los lineamientos y los principios sobre cómo debería ser el desarrollo de *software*. En la figura 12 se muestra una captura de pantalla del mismo, en él se expresan las aspiraciones de lo que los procesos de desarrollo ágiles deben tener para resolver los problemas de las metodologías tradicionales.



**Figura 12**

*Manifiesto Ágil*



Nota. Tomado de *Manifiesto por el Desarrollo Ágil de Software* [Ilustración], por Beck, K., Beedle, M., Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J. y Thomas, D., 2001, [agilemanifesto](#), CC BY 4.0.

Si analizamos brevemente este manifiesto, vemos que se resalta y se prioriza a los clientes, entregando resultados útiles antes que documentación, además la colaboración con el cliente ayuda a que el proceso se adapte rápidamente a los cambios, de modo que el producto siempre agrega valor al cliente y usuarios.

Un peligro al tratar esta filosofía es el de pensar que se trata de procesos informales y sin calidad, sin embargo, es necesario acotar que no tienen nada de ello, simplemente se enfocan en los resultados antes que en la documentación o los procesos, pero la calidad del producto resultante debe ser la misma que con un enfoque tradicional, naturalmente el hecho de que se adapte rápidamente a los cambios hará que el resultado genere mayor valor para los clientes, y más importante aún al colaborar con el cliente y los usuarios se logra que el producto resultante sea mucho más valioso para el cliente.

Ahora, le invito a revisar los principios del Manifiesto Ágil y conteste las siguientes preguntas, estos principios están en [Principios del manifiesto ágil](#).

A continuación, vamos a revisar brevemente dos de las metodologías ágiles más importantes.

#### a. Programación extrema (extreme programming)

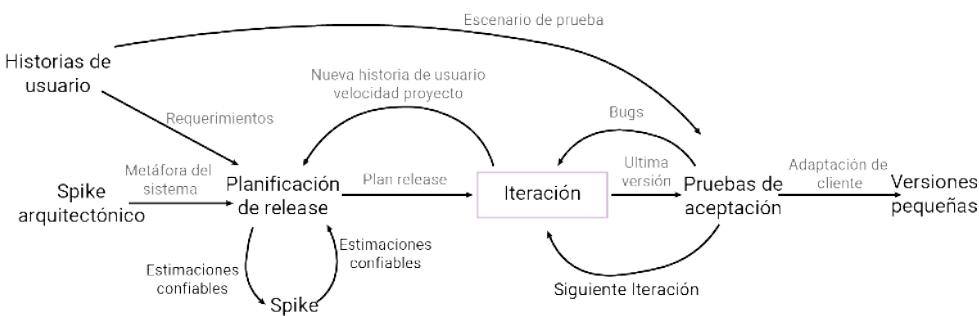
Fue desarrollada por Kent Beck en 1996, la Programación Extrema (XP) es uno de los primeros modelos de desarrollo de software ágiles. Su nombre surge del hecho de que en esta metodología el desarrollo se lleva al extremo, es usual que los programadores primero prueben su lógica de negocio antes de escribir cualquier código, y para asegurar que la misma se implementa correctamente trabajan en pares, mientras una persona escribe el código, la otra lo revisa simultáneamente.

Para (Ahmed, 2016) la metodología Programación Extrema permite a las compañías liberar sus productos lo antes posible, y examinar el mercado con un producto con pocas características, en caso de que no sea aceptado, se puede detener su desarrollo.

Un factor clave en esta metodología es que el cliente siempre está involucrado y es responsable de todos los aspectos relacionados con la especificación de requerimientos.

Las actividades de XP son: codificación, en la cual se genera código; pruebas, se considera que es la mejor forma lograr un producto de calidad; escucha, en la cual los desarrolladores escuchan lo que el cliente espera del producto, explica la lógica de negocio que está detrás de las características que espera del producto, y, diseño, que cuando el producto ya es grande después de muchas iteraciones es una tarea primordial para garantizar su correcto funcionamiento. En la figura 13 se aprecia la forma de trabajo de XP.

**Figura 13**  
*Flujo de procesos de XP*



Nota. Adaptado de *Extreme Programming:A gentle introduction* [Ilustración], por Don Wells, 2013, [ExtremeProgamming](#), CC BY 4.0.

## Ventajas

Permite co-localizar pequeños equipos de proyectos para trabajar en proyectos de desarrollo de software, no hay labores de gestión, debido a que los equipos se autoorganizan y solo hay un gestor de proyecto involucrado. Permite la construcción incremental de los productos de software, lo cual le permite incluir en el mercado productos de software muy temprano.

## Desventajas

Si se requiere un producto muy grande construido de manera rápida, puede no ser posible con XP, considere que un equipo de 5 a 6 personas trabajando con XP pueden producir 20 líneas de código por día, eso

significa que para un producto que requiera 100.000 líneas de código se requerirán aproximadamente 17 años asumiendo años de 300 días laborables(Ahmed, 2016)

### Cuando usar XP

Se debería usar cuando se necesita desarrollar una aplicación de forma incremental, en esta metodología no es necesario esperar a tener todos los requerimientos y sobre todo cuando hay riesgo de que el cliente cambie los requerimientos de manera muy frecuente. Es decir, el nivel de incertidumbre es muy alto. Es más conveniente utilizarla cuando se tiene pequeños equipos que trabajan en el mismo sitio debido a que se necesitan muchas reuniones y actualizaciones del producto muy frecuentes.

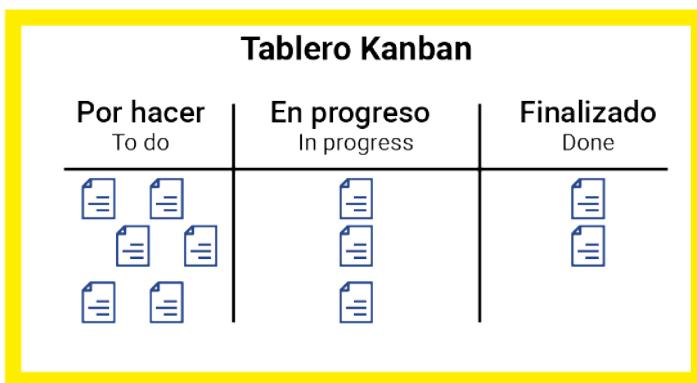
### b. Kanban

La metodología Kanban es un sistema visual de gestión de trabajo que ayuda a mejorar el flujo de tareas o procesos, haciéndolos más transparentes, eficientes y flexibles. Su origen se remonta a la industria automotriz japonesa, en Toyota, en la década de 1940, donde buscaban optimizar la producción “just-in-time” (justo a tiempo) y reducir inventarios excesivos.

En la práctica, Kanban implica usar un tablero (físico o digital) dividido en columnas que representan estados del trabajo: por ejemplo “Pendiente”, “En progreso” y “Finalizado”. Cada tarea se representa mediante una tarjeta que se mueve a través de estas columnas conforme avanza su ejecución. Esto permite ver de un vistazo cuál es el estado de cada tarea y detectar cuellos de botella o áreas donde el proceso se atasca. La figura 14 muestra un ejemplo del tablero

**Figura 14**

Ejemplo Tablero Kanban



Nota. Jaramillo, D., 2025.

**Algunos principios clave de Kanban son:**

- Visualizar el flujo de trabajo para que todos los miembros del equipo entiendan lo que se está haciendo y qué sigue.
- Limitar el número de tareas que están “en progreso” (WIP: work in progress) para evitar tener demasiadas tareas abiertas al mismo tiempo y que eso genere ineficiencia.
- Gestionar el flujo de trabajo para que las tareas avancen lo más suavemente posible, reduciendo retrasos, sobrecarga, desperdicios y cambios innecesarios.
- Hacer mejoras continuas basado en el aprendizaje del flujo real, detectar dónde se atasca el trabajo, medir tiempos, revisar procesos y ajustar.

## Ventajas

- Visibilidad del trabajo: al representar las tareas en un tablero, todos los miembros del equipo saben qué se está haciendo, qué falta y dónde hay bloqueos.
- Flexibilidad: se adapta fácilmente a diferentes entornos (desarrollo de software, producción, marketing, soporte, etc.) sin necesidad de cambios radicales en la organización.

- Reducción de cuellos de botella: al limitar el work in progress (WIP), se evita la sobrecarga de trabajo y se mejora el flujo, aumentando la eficiencia.
- Mejora continua: fomenta un enfoque iterativo donde se pueden analizar métricas, identificar problemas y optimizar procesos de manera constante.



## Desventajas

- Falta de plazos estrictos: Kanban no establece fechas límite por sí mismo, lo que puede ser un problema en proyectos con tiempos de entrega muy rígidos.
- Riesgo de saturación: si no se respetan los límites de WIP, el tablero puede llenarse de tareas y perder efectividad.
- Dependencia del compromiso del equipo: requiere disciplina para actualizar las tarjetas, respetar el flujo y mantener la transparencia; de lo contrario, el sistema se desordena.
- Menos planificación a largo plazo: está más enfocado en la gestión del flujo actual de trabajo que en la planificación estratégica de proyectos grandes o complejos.

### c. Scrum

La metodología Scrum es un marco de trabajo ágil diseñado para gestionar proyectos complejos, especialmente en el ámbito del desarrollo de software, aunque su aplicación se ha extendido a diferentes sectores gracias a su versatilidad y eficacia. A diferencia de los enfoques tradicionales, que suelen ser rígidos y secuenciales, Scrum se basa en ciclos iterativos e incrementales denominados sprints, los cuales tienen una duración fija que generalmente oscila entre una y cuatro semanas. Durante cada sprint, los equipos trabajan en un conjunto definido de tareas o funcionalidades priorizadas, con el objetivo de entregar un incremento funcional del producto al final de ese período. Una de las principales características de Scrum es su énfasis en la colaboración, la

transparencia y la capacidad de adaptación, lo que lo convierte en un marco muy adecuado en entornos cambiantes donde los requisitos pueden evolucionar constantemente.

El funcionamiento de Scrum se apoya en roles claramente definidos: el Product Owner, que representa los intereses del cliente y se encarga de gestionar el product backlog o lista priorizada de requerimientos; el Scrum Master, que actúa como facilitador, eliminando impedimentos y asegurando que el equipo siga los principios del marco; y el Development Team, un grupo multidisciplinario y autoorganizado que ejecuta el trabajo técnico. Además, Scrum cuenta con eventos o ceremonias clave como la Sprint Planning, donde se planifica el trabajo del sprint; el Daily Scrum o reunión diaria de seguimiento de máximo quince minutos; la Sprint Review, en la que se presenta el resultado a los interesados para recibir retroalimentación; y la Sprint Retrospective, orientada a la mejora continua de los procesos internos del equipo.

Entre las principales ventajas de Scrum se encuentran la capacidad de responder rápidamente a cambios, la entrega temprana y continua de valor al cliente, y la mejora de la motivación y la comunicación dentro del equipo. Sin embargo, también requiere disciplina, compromiso y una cultura organizacional que apoye la autonomía de los equipos

En el siguiente apartado de esta guía se detallará de forma más explícita esta metodología.



- ¿Cuál es el tiempo preferido en el cual se debe entregar resultados?
- ¿Cuál debería ser la mayor fortaleza de los equipos de trabajo?
- ¿Cómo debe ser la comunicación entre miembros del equipo?
- ¿Cómo abordan el tema de la calidad del producto?



## Actividades de aprendizaje recomendadas

A continuación, le invito a que desarrolle las actividades de aprendizaje recomendadas:

1. Revise el capítulo 2 de (Somerville, 2016), realice un mapa conceptual que le permita comprender los conceptos y modelos sobre los procesos de software.
2. Revise el tema de Ingeniería de software de cuarto limpio, busque este ejemplo en Internet o lo puede encontrar en Somerville, página 32, para profundizar sobre el aprendizaje de esta unidad.
3. Responda a las siguientes preguntas con relación a una metodología. Prepare preguntas para la tutoría semanal.
  - ¿Cuál es el tiempo preferido en el cual se debe entregar resultados?
  - ¿Cuál debería ser la mayor fortaleza de los equipos de trabajo?
  - ¿Cómo debe ser la comunicación entre miembros del equipo?
  - ¿Cómo abordan el tema de la calidad del producto?

Nota. Por favor, complete las actividades en un cuaderno o documento Word.



## Contenidos, recursos y actividades de aprendizaje recomendadas



### Semana 3

#### Unidad 2. Procesos de software

##### 2.4. Scrum

Varios autores consideran a SCRUM como la metodología ágil más popular para proyectos de software, sin embargo (Schwaber & Sutherland, 2017), definen a SCRUM como “Un framework en el cual las personas pueden abordar problemas complejos adaptativos, al tiempo que ofrecen productos

productivos y creativos del mayor valor posible". Su popularidad ha aumentado debido a su capacidad inherente para simplificar las cosas en un proyecto, así como para escalar cuando sea necesario.

SCRUM trabaja con 3 roles que son:

- SCRUM Master.
- (*Product Owner*) el propietario del producto y.
- (*Team*). Equipo del proyecto.

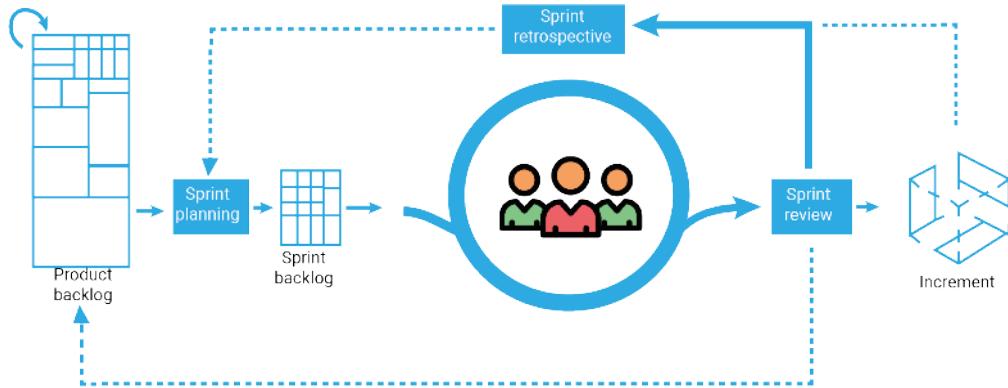
El papel del SCRUM Master es facilitar el trabajo del equipo, resolviendo obstáculos que les impidan lograr su objetivo. Su labor no es de gestor del proyecto, ya que, por principio de agilidad, los equipos son autoorganizados.

El *Product Owner* es el representante del cliente, su función es proporcionar los requisitos de software en forma de historias de usuario (*User Story*) al equipo del proyecto, también lleva a cabo pruebas de aceptación al final de cada iteración (*Sprint*) para asegurarse de que las características integradas del producto integradas en el *Sprint* se cumplen según las historias de los usuarios.

El equipo del proyecto es responsable de crear diseños de software, diseños de bases de datos, escribir código fuente y probar el código fuente. No hay especialistas en el equipo y cualquiera puede hacer el trabajo del proyecto. Sin embargo, es una buena idea tener una persona que sea propietaria de cada artefacto del proyecto. Entonces, aunque otros miembros del equipo también pueden crear artefactos del proyecto; la responsabilidad de cada artefacto del proyecto está claramente establecida.

El proyecto comienza con una épica (*Epic*), que es una gran historia creada por el cliente y que describe lo que se propone que haga el software. Luego el proceso se desarrolla como se muestra en la figura 15, en la que se aprecia el framework de SCRUM.

**Figura 15**  
Framework Scrum



Nota. Adaptado de *What is Scrum? [Ilustración]*, por Scrum.org, 2020, [Scrum.org](https://scrum.org), CC BY 4.0.

A continuación, se explica brevemente el funcionamiento del *framework*:

1. Se crea una **lista de historias de usuario** que incluyen todo lo que se conoce que debe incluir el proyecto en un artefacto llamado pila de pendientes del producto (*Product Backlog*), esta lista evoluciona en la medida que avanza el proyecto y se aclaran los requerimientos.
2. Del ***Product Backlog***, se toman grupos de historias de usuario y se desarrollan las iteraciones (*Sprints*), cada una dura alrededor de un mes, dando como resultado un incremento del producto (*Product Increment*)
3. Cada ***Sprint*** inicia con la planificación (*Sprint Planning*) en ella que establece su objetivo (*Sprint Goal*) y se determina lo que se puede entregar en ese incremento, produciendo otro artefacto que se conoce como pendientes del sprint (*Sprint Backlog*)
4. El **equipo trabaja** en el desarrollo del producto, al final de cada día se llevan a cabo reuniones diarias denominadas Scrum diarios (*Daily Scrum*), para evaluar el progreso hacia el objetivo del *Sprint* y monitorear el *Sprint Backlog*, ello permite optimizar el trabajo para alcanzar el *Sprint Goal*.
5. Al finalizar cada *Sprint*, se lleva a cabo una reunión denominada revisión del sprint (*Sprint Review*), en la que participan el SCRUM Master, el Team, el Product Owner y otros interesados clave, en ella el Product Owner

explica los elementos de la lista que se han terminado y cuáles han quedado pendientes, el equipo realiza una demostración del incremento, y el *Product Owner* lo aprueba, si algo queda pendiente se agrega a la lista de pendientes, además se establecen las metas para las próximas iteraciones.

6. Una vez aprobado el incremento se lleva a cabo una reunión del equipo denominada retrospectiva del *sprint* (*Sprint Retrospective*) en la cual el equipo se inspecciona a sí mismo y crea un plan de mejoras para las próximas iteraciones, el SCRUM Máster se asegura de que el evento se lleve a cabo y de que los asistentes entiendan su propósito.

## Ventajas de scrum

- El desarrollo es iterativo e incremental, por tanto, al final de cada iteración se entrega un reléase del producto, lo cual significa que en corto tiempo se puede colocar en el mercado una versión menor del producto.
- La planificación es un factor importante en Scrum, es muy fácil adaptar el proceso y entrar en mejora continua debido a las reuniones diarias, a la retrospectiva del *sprint*, y a la retroalimentación de parte cliente a través del *Product Owner*, lo cual permite que el producto esté siempre adecuado a las necesidades del negocio.
- A través de una técnica conocida como Scrum of Scrums, un clúster de equipos de proyecto puede trabajar en el mismo proyecto, por tanto, se puede escalar proyectos muy grandes y complejos.

## Desventajas

- Scrum está pensado para trabajar con pequeños equipos de proyecto trabajando en el mismo espacio, por tanto, no es conveniente para equipos de trabajo muy grandes a no ser que constituyan en un esquema Scrum of Scrums.
- Los proyectos que en los que se ha establecido costo y tiempo fijos, no pueden ser adaptados al entorno de Scrum, debido a que no es posible planificar todo el proyecto.

- Para trabajar con Scrum es necesario que los miembros del equipo sean experimentados, de otra manera no podrían autoorganizarse, ya que no hay un gestor del proyecto, si no es así, el proyecto puede terminar en caos.
- Si se requiere una persona con habilidades muy especializadas, puede resultar muy difícil trabajar con Scrum, puesto que se asume que todos los miembros del equipo son capaces de hacer cualquier clase de trabajo.

## Cuando usar SCRUM

Cuando se debe abordar proyectos muy complejos con altos niveles de incertidumbre, se cuenta con equipos de trabajo altamente experimentados y además se cuenta con el apoyo y la retroalimentación del cliente a lo largo de todo el proceso.

Se debe considerar además proyectos en los que el plazo y el costo no se hayan establecido como fijos, ya que la dinámica de Scrum hace muy difícil planificar en esos términos. Como mucho, se puede saber qué es lo que se planifica en cada *Sprint*.

Estimado estudiante, si usted desea más información puede ingresar y registrarse de manera gratuita a los sitios de [Agile Alliance](#) y [Scrum](#) para descargar libros y material de capacitación de uso libre.



### Actividades de aprendizaje recomendadas

A continuación, le invito a que desarrolle las actividades de aprendizaje recomendadas:

1. Elabore una lista de criterios que, con base en las características del proyecto, le permitan optar por un enfoque predictivo, iterativo, incremental o ágil.
2. Elabore un cuadro comparativo entre las metodologías de desarrollo tradicionales donde se evidencie el ciclo de vida, ventajas y desventajas.
3. Compare las metodologías XP y SCRUM, determine los escenarios en los cuales resultaría más conveniente una u otra.

4. Revise los siguientes videos:

- [#1. ¿Qué son las metodologías tradicionales en el desarrollo de software?](#)
- [#2. ¿Qué son las metodologías ágiles en el desarrollo de software?](#)  
Este video le ayudará para complementar su conocimiento sobre el origen y lo que son las metodologías ágiles.
- [#3. Scrum en 6 minutos - metodologías ágiles](#). Este vídeo le servirá para comprender los usos y el funcionamiento de Scrum.

Nota. Por favor, complete las actividades en un cuaderno o documento Word.

5. Desarrolle la autoevaluación de la unidad 3 para comprobar sus conocimientos.

### [Autoevaluación 2](#)

1. En el modelo en cascada, ¿en qué fase se definen los servicios, restricciones y metas del sistema?

- a. Análisis y definición de requerimientos.
- b. Diseño del sistema y del software.
- c. Implementación y pruebas unitarias.
- d. Mantenimiento del sistema.

2. ¿Qué modelo de desarrollo exige que cada fase concluya por completo antes de iniciar la siguiente?

- a. Cascada.
- b. Incremental.
- c. Orientado a la reutilización.
- d. Espiral.



3. El enfoque que consiste en crear una primera versión, mostrarla al usuario y luego mejorarla en distintas versiones hasta alcanzar el sistema final se denomina:



- a. Modelo en cascada.
- b. Modelo incremental.
- c. Modelo orientado a la reutilización.
- d. Modelo en espiral.



4. ¿Cuáles son las actividades centrales en el proceso de ingeniería de requerimientos?



- a. Especificación, análisis, implementación y pruebas.
- b. Análisis de componentes, modificación y diseño del sistema.
- c. Estudio de factibilidad, obtención y análisis, especificación y validación.
- d. Diseño, codificación, integración y despliegue.



5. ¿Cuál de los siguientes es un componente utilizado en procesos orientados a la reutilización?



- a. MVC.
- b. Servicios web.
- c. Sistemas remotos.
- d. Librerías compartidas.



6. En la etapa de especificación del software, ¿qué actividad se encarga de verificar que los requerimientos sean coherentes, completos y viables?



- a. Pruebas unitarias.
- b. Validación del software.
- c. Validación de requerimientos.
- d. Revisión de código.

7. ¿En qué actividad del proceso de diseño se determina la estructura global del sistema?

- a. Diseño arquitectónico.
- b. Diseño de interfaz.
- c. Diseño de componentes.
- d. Diseño de base de datos.

8. ¿Qué actividad del diseño se centra en la definición de la estructura de los datos del sistema?

- a. Diseño de interfaz.
- b. Diseño de componentes.
- c. Diseño de la base de datos.
- d. Diseño arquitectónico.

9. La fase en la que los desarrolladores prueban individualmente los componentes creados del sistema corresponde a:

- a. Prueba de desarrollo.
- b. Prueba de sistema.
- c. Prueba de aceptación.
- d. Prueba unitaria.

10. Dentro de la ingeniería de requerimientos, ¿qué estrategia se emplea para ayudar en la selección y validación de requerimientos?

- a. Obtención.
- b. Prototipado.
- c. Talleres de trabajo.
- d. Entrevistas.

[Ir al solucionario](#)



## **Resultado de aprendizaje 3:**

Identifica el ciclo de vida a utilizar en el desarrollo de un proyecto de software.

Para alcanzar el resultado de aprendizaje 3, se estudiará la definición de proyecto, su organización, las tareas y productos de trabajo asociados, y la creación de un cronograma. Con este conocimiento, será posible elegir y aplicar el ciclo de vida del desarrollo que mejor se ajuste a las necesidades del proyecto, garantizando una gestión eficaz y el logro de los objetivos establecidos.

### **Contenidos, recursos y actividades de aprendizaje recomendadas**

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



### **Semana 4**

En la unidad 2 se estudiaron los modelos de proceso de software y algunas de las metodologías de desarrollo más importantes, vale aclarar que, si bien las metodologías ágiles están de moda, no todos los proyectos deben desarrollarse utilizando una de ellas, por ello es importante que el ingeniero en Tecnologías de la Información, sea capaz de discernir cuál enfoque es el más adecuado de acuerdo a las características del proyecto.

Con esta base, resulta natural dar el siguiente paso hacia la gestión y análisis de los proyectos en sí, comprendiendo los factores que determinan su éxito o fracaso. ¡Es momento de abordar la siguiente unidad!

### **Unidad 3. Proyectos de software**

Una de las características del desarrollo de software es que todo debe desarrollarse bajo la figura de un proyecto. Esta concepción es importante porque rompe el esquema de trabajo de un programador.

En la presente unidad, vamos a resumir algunas de las características más relevantes de lo que es un proyecto de software, acotando que la gestión de proyectos como tal tiene un alcance mucho más grande del que trataremos aquí. De hecho, hay otra asignatura que se hará cargo de ese tema.

### 3.1. Definición de proyecto

Existen muchas definiciones de lo que es un proyecto, pero dado que el Cuerpo de Conocimiento de la Ingeniería de Software ((Washizaki, n.d.), acoge el estándar del *Project Management Institute* vamos a tomar la definición de ese estándar que establece en términos generales “Un proyecto es un esfuerzo temporal que se lleva a cabo para crear un producto, servicio o resultado único”, este concepto aplica para cualquier tipo de producto, en este caso al software, y, por tanto, este esfuerzo tiene un objetivo, tiene unos recursos y tiene algunas restricciones de alcance, costo y tiempo. Ello nos lleva a la conclusión de que los trabajos que tienen que ver con el desarrollo, modificación o mantenimiento de un producto de software, se constituyen en un proyecto.

Un proyecto, por lo tanto, tiene un grupo de personas a los que se denomina el equipo del proyecto y un grupo de participantes que son los miembros del equipo. Las personas que se verán afectadas por el desarrollo del proyecto o tienen participación en el proyecto o en su resultado se denominan interesados (*stakeholders*), el equipo de proyectos.

Debe contar con ellos para el desarrollo, ya que ellos aportan necesidades y pueden tomar decisiones respecto de la continuidad del proyecto.

Los participantes del equipo deben relacionarse de diferentes maneras y distinguimos algunas relaciones:

- **Reporte**, en el cual los participantes informan el estado de avance de sus tareas.
- **Decisión**, en la que la persona a cargo o líder toma decisiones respecto del proyecto, y.

- **Comunicación**, que se utiliza para intercambiar información con otros participantes, usuarios, clientes. La comunicación puede ser jerárquica, en cuyo caso será unidireccional y horizontal cuando se da entre miembros del equipo.

El PMI (*Project Management Institute, 2017*), define dos conceptos importantes con relación al alcance, el alcance del producto que es el “conjunto de características y funciones de un producto, servicio o resultado”, y el alcance del proyecto que es “el trabajo realizado para entregar un producto, servicio o resultado con las funciones y características especificadas”, además se afirma que en ocasiones el alcance del proyecto incluye el alcance del producto.

### 3.2. Organización del proyecto

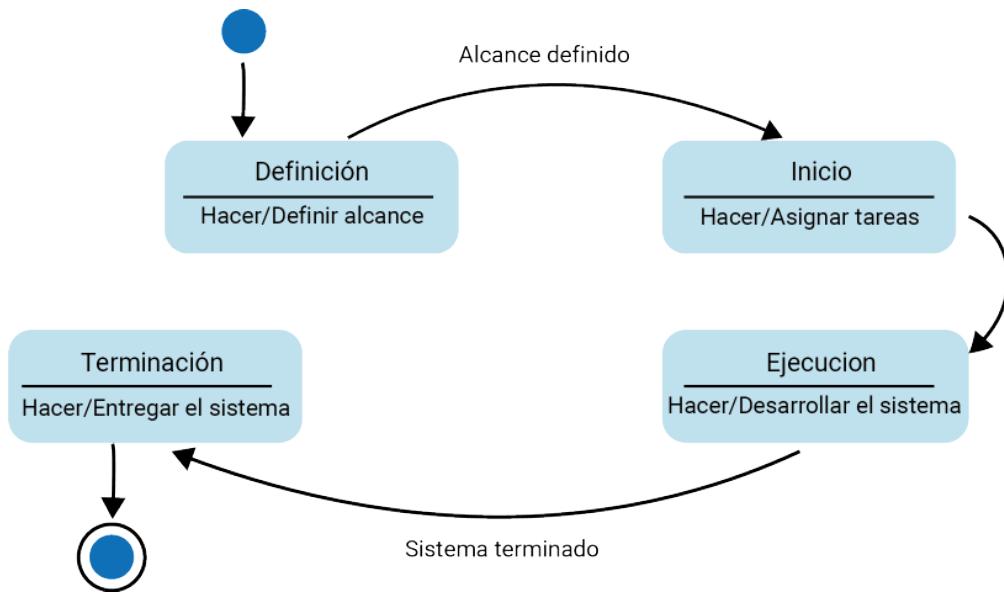
Un aspecto importante del proyecto es poder dar respuesta a preguntas como ¿quién es el responsable de qué parte del sistema?, ¿a quién se debe contactar cuando ocurren problemas con alguna versión específica de un componente?, ¿cómo se debe documentar un problema?, ¿cuáles son las restricciones del proyecto?, ¿cuáles son los criterios para evaluar el sistema?, ¿a quién se debe informar si aparecen nuevos requerimientos?, ¿quién es el encargado de hablar con el cliente? Según (Bruegge & Dutoit, 2018), desde la perspectiva de los desarrolladores, un proyecto consta de 4 componentes y de 4 etapas. Los componentes son:

- Producto de trabajo: cualquier ítem producido por el proyecto, tal como una pieza de código, un modelo o un documento, los productos de trabajo que van a manos del cliente.
- Cronograma: artefacto que especifica cuándo se debe completar el trabajo, puede representarse en diagramas de Gantt.
- Participante: es cualquier persona que participa o cumple una función en el proyecto.
- Tarea: es el trabajo que debe ser ejecutado por un participante del proyecto para crear un producto de trabajo.

Un proyecto se organiza en etapas o fases, cada una de las cuales tiene un propósito bien definido. La culminación de una fase se produce cuando se completa una condición de salida. El modelo genérico de las fases de un proyecto se indica en la figura 16.

**Figura 16**

Diagrama de estados de las etapas de un proyecto de software



Nota. Adaptado de *Object-oriented software engineering : using UML, patterns, and Java [Ilustración]*, por Bruegge, B. y Dutoit, A., 2018, Prentice Hall, CC BY 4.0.

Veamos el funcionamiento de cada etapa de este modelo genérico de gestión de proyectos.

- **Etapa de definición:** el gestor del proyecto, el cliente, un miembro clave del proyecto, y el arquitecto de software trabajan en la comprensión inicial de la arquitectura del sistema, específicamente los subsistemas que conformarán el sistema, el cronograma, el trabajo requerido y los recursos necesarios, esto se documenta en 3 artefactos: la declaración del problema, el documento de arquitectura inicial y el plan de gestión del proyecto.

- **Etapa de inicio:** el gestor del proyecto define una infraestructura, contrata a los participantes, organiza los equipos de trabajo, define los hitos principales y lanza el proyecto.
- **Etapa de ejecución:** los participantes desarrollan el sistema, reportan sus avances al líder de equipo, quien lleva a cabo el seguimiento del estado del trabajo de los desarrolladores e identifica posibles problemas, los líderes de equipo reportan el estado de avance al gestor del proyecto para evaluar el estado global del proyecto.
- **Etapa de terminación:** el resultado del proyecto se entrega al cliente y se guarda la historia del proyecto.

En un proyecto los participantes cumplen con un rol, el cual define un conjunto de tareas técnicas y de gestión que se espera de los participantes o del equipo, los roles pueden ser de gestión, de desarrollo y cruzados.

- **Roles de gestión:** son roles cuya función principal tiene que ver con la organización y ejecución del proyecto dentro de las restricciones, por ejemplo, se tiene el rol de gestor del proyecto y líder de equipo.
- **Roles de desarrollo:** tiene que ver con las actividades técnicas de desarrollo del sistema, estos incluyen: analistas, arquitectos de software, diseñadores y programadores.
- **Roles cruzados:** roles encargados de la coordinación entre equipos, son responsables de diseminar la información entre la estructura de comunicación del proyecto de un equipo a otro, y en ocasiones puede actuar como representante de un equipo a cargo de un subsistema, entre estos roles tenemos: Editor de documentación, gestor de la configuración y el tester.
- **Roles de consulta:** encargados de proveer soporte temporal en áreas donde los participantes del proyecto carecen de experiencia, la mayoría de las veces los usuarios y el cliente actúan como consultores en el dominio de aplicación, también puede haber consultores técnicos expertos en determinadas tecnologías o métodos y consultores no técnicos que ayudan a dirigir aspectos legales y de mercadotecnia.



Podemos mencionar los siguientes roles: cliente, usuario final, especialista de dominio de aplicación, especialista de dominio de solución.

### 3.3. Tareas y productos de trabajo

Una tarea se entiende como una asignación de trabajo bien definida asignado a un rol. Los grupos de tareas relacionadas se denominan actividades. Estas tareas son asignadas por el gestor del proyecto o el líder de equipo a un participante, y además monitorea su avance hasta la culminación.

Un producto de trabajo es un ítem tangible resultante de una tarea, como por ejemplo un modelo de objetos, un diagrama de clases, un código fuente, un documento o partes de él. Estos productos de trabajo están sujetos a fechas de vencimiento y muchos se alimentan de otras tareas. Los productos de trabajo que no son visibles para el cliente, se denominan productos de trabajo internos, y los que se entregan al cliente se denominan entregables.

La especificación del trabajo que debe completarse, se define en un paquete de trabajo, el cual incluye el nombre de la tarea, la descripción, los recursos necesarios, las dependencias o entradas y las salidas, así como las dependencias con otras tareas.

### 3.4. Cronograma del proyecto

Es un mecanismo que permite mapear las tareas que deben ejecutarse a lo largo del tiempo, cada tarea tiene un conjunto de atributos tales como fecha de inicio, fecha de fin, duración, recursos asignados y la dependencia con otras tareas.

Las notaciones más utilizadas para representar los cronogramas son el diagrama de Gantt y los diagramas Pert. El diagrama de Gantt es una especie de diagrama de barras que en sentido vertical representa la lista de actividades y en sentido horizontal se encuentran las unidades de tiempo y las barras relacionan las actividades con el tiempo en el que deben desarrollarse. En la figura 17 se puede apreciar un diagrama de Gantt utilizado para un



proyecto de implementación de escáneres en los centros universitarios de la UTPL y permite digitalizar las hojas de respuesta dadas por los estudiantes para ser enviados electrónicamente y calificados en la sede central.

Ambos diagramas fueron desarrollados utilizando la herramienta WBS Schedule Pro.

**Figura 17**

*Diagrama de Gantt para un proyecto de implementación de escáneres en centros universitarios digitalizar hojas de respuesta de evaluaciones de la UTPL*

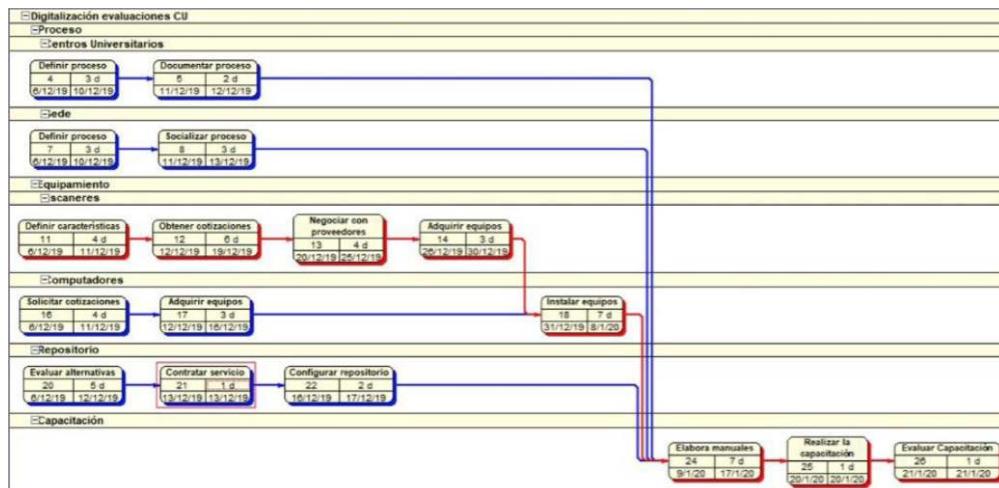


Nota. Jaramillo, D., 2025.

En la figura 18 se aprecia el mismo diagrama representado en diagrama PERT. Cabe resaltar que, si bien ambos diagramas muestran la misma información, cada uno de ellos ofrece una perspectiva diferente del cronograma, el diagrama Pert permite resaltar el tiempo de manera cronológica y el diagrama Pert se enfoca en las dependencias entre actividades, lo que se visualiza en color rojo es lo que se conoce como ruta crítica, esta ruta crítica se usa para establecer la duración del proyecto porque la secuencia de actividades con una duración más larga y, por tanto, las actividades que la conforman no se pueden retrasar.

**Figura 18**

Diagrama PERT para un proyecto de implementación de escáneres en centros universitarios digitalizar hojas de respuesta de evaluaciones de la UTPL



Nota. Jaramillo, D., 2025.

Para profundizar en el tema, revise el video: [Ms Project: crea un proyecto](#), donde encontrará información sobre cómo desarrollar el cronograma, y además encontrará información sobre cómo se realiza la planificación en un enfoque ágil.

Usted puede descargar software para ayudarle en el desarrollo de estos diagramas, algunos son gratuitos como Project Libre, y el caso de WBS Schedule Pro, puede obtener una licencia demostrativa por 30 días o adquirirlo.

En la presente unidad se ha desarrollado una explicación bastante elemental de lo que es la gestión de un proyecto de software, con el propósito de que se entienda que el desarrollo de software desde el punto de vista profesional no incluye solamente las actividades de programación de una sola persona, sino que implican el trabajo de un equipo de personas cumpliendo diferentes roles y que se sujetan a un cronograma para cumplir con un alcance en un tiempo determinado.

### 3.5. EDT

La Estructura de Desglose del Trabajo (EDT), conocida en inglés como *Work Breakdown Structure* (WBS), es una herramienta fundamental de la gestión de proyectos que consiste en dividir un proyecto complejo en partes más pequeñas, organizadas y manejables. Su propósito principal es facilitar la planificación, asignación de recursos, control y seguimiento de las tareas. En proyectos de software, el EDT permite descomponer el producto y los entregables en niveles jerárquicos: desde el proyecto en su totalidad, pasando por fases o módulos, hasta llegar a actividades específicas. De esta manera, se logra una visión clara de lo que debe desarrollarse y cómo se estructura el trabajo, lo que evita ambigüedades y asegura que nada importante quede fuera del alcance.

¿Cómo realizar un EDT en un proyecto de software?

Para la realización de un EDT podemos basarnos en temas como:

- Definir el alcance del proyecto.
  - Revisar el acta de constitución del proyecto y los requisitos del cliente.
  - Asegurarse de tener claridad sobre los entregables principales (ejemplo: sistema web, aplicación móvil, base de datos, manual de usuario, capacitación).
- Identificar los entregables principales.
  - Dividir el proyecto en grandes bloques de trabajo.
  - Por ejemplo: análisis de requisitos, diseño del sistema, desarrollo, pruebas, implementación y mantenimiento.
- Descomponer en niveles jerárquicos.
  - Cada bloque se divide en subentregables o paquetes de trabajo más pequeños.
  - Ejemplo: dentro de Desarrollo puede haber módulo de autenticación, módulo de reportes, API de integración.

- Definir paquetes de trabajo manejables.
  - Un paquete de trabajo debe ser lo suficientemente pequeño para poder asignarse a un responsable, estimar su costo y tiempo, y dar seguimiento.
  - Ejemplo: desarrollar formulario de login o diseñar base de datos de clientes.
- Representar en un diagrama jerárquico o numerado.
  - Usar diagramas tipo árbol o una estructura codificada.
    - Ejemplo numérico:
    - 1.0 Análisis de requisitos
    - 2.0 Diseño del sistema
      - 2.1 Diseño de base de datos.
      - 2.2 Diseño de Interfaz de Usuario.
    - 3.0 Desarrollo
      - 3.1 Módulo de autenticación.
      - 3.2 Módulo de reportes.
  - Validar con el equipo y stakeholders.

Revisar que el EDT cubra todo el alcance del proyecto y que no haya tareas duplicadas ni faltantes.

### 3.6. Fases del proyecto

#### 1. Inicio del proyecto

En esta fase se identifican las necesidades que justifica el proyecto y se define su alcance general. Se determinan los objetivos principales, los interesados (*stakeholders*) y los recursos iniciales. También se elabora el acta de constitución del proyecto, que oficializa su existencia.

## **2. Planificación**

Aquí se responde a la pregunta: ¿cómo se desarrollará el proyecto? Se establece el plan de trabajo, el cronograma, los costos, los recursos humanos y tecnológicos, así como los riesgos y sus estrategias de mitigación. Una herramienta clave es el EDT (Estructura de Desglose del Trabajo), que organiza los entregables en componentes manejables.

## **3. Ejecución**

Es la fase en la que se llevan a cabo las actividades planificadas. Se desarrolla el producto de software, se implementan los entregables y se realizan pruebas parciales para asegurar que cumplen con lo requerido. También se gestionan las comunicaciones y la coordinación del equipo de trabajo.

## **4. Monitoreo y control**

En paralelo a la ejecución, se supervisa el avance del proyecto para asegurarse de que cumpla con lo planificado. Se comparan resultados reales contra lo previsto, se corrigen desviaciones y se actualizan los planes si es necesario. Esta fase garantiza calidad, cumplimiento de plazos y control de costos.

## **5. Cierre del proyecto**

Es la fase final, en la que se entregan los resultados al cliente o usuario, se validan los objetivos alcanzados y se documentan las lecciones aprendidas. Además, se capacita a los usuarios finales y se formaliza la aceptación del proyecto.

Para el estudio nos basaremos en un ejemplo práctico de la Empresa TRAMIL, para lo cual le pido que revise el documento de Caso de estudio TRAMIL – descripción. Las fases de este caso pueden visualizarse de forma resumida en la siguiente infografía:

[Gestión de proyectos de software caso TRAMIL](#)

En conclusión, el ciclo de gestión de un proyecto de software aplicado al caso TRAMIL va desde definir el problema (inicio) hasta implantar y evaluar la solución (cierre), pasando por planificación, ejecución y control, lo que asegura que el sistema entregue valor real a la empresa y a sus clientes.



## Actividades de aprendizaje recomendadas



Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. Revise el siguiente video sobre La gestión de proyectos de software [Ingeniería del software - gestión de proyectos](#) - Raquel Martínez España.
2. Realice las etapas de planificación de un proyecto (personal), el mismo que será utilizado en el desarrollo de la presente asignatura.
3. La presente unidad es de tipo informativo, en la cual se han estudiado algunos conceptos y características de los proyectos. A continuación, le invito a responder las preguntas de la siguiente autoevaluación.



## Autoevaluación 3

1. Según las definiciones vistas en clase, ¿cuál de los siguientes ejemplos corresponde a un proyecto de software?
  - a. Redactar el documento de requerimientos del sistema.
  - b. Formar el equipo de trabajo.
  - c. Desarrollar una aplicación de ventas para un supermercado.
  - d. Contratar nuevo personal.
2. A las personas que tienen algún grado de participación en un proyecto se les denomina:
  - a. Participantes.
  - b. Equipo del proyecto.
  - c. Interesados.
  - d. Usuarios.

3. Respecto al alcance del proyecto, ¿cuál de las siguientes afirmaciones es válida?

- a. En ciertos casos, el alcance del proyecto incluye al alcance del producto.
- b. El alcance del proyecto es únicamente el tiempo disponible para terminarlo.
- c. El alcance del proyecto corresponde a las características del producto.
- d. El alcance del proyecto se refiere a la estructura del equipo.

4. Con relación a los interesados en un proyecto, ¿qué afirmación es correcta?

- a. Siempre forman parte directa del equipo de trabajo.
- b. Son personas o grupos que pueden influir en el proyecto o en sus resultados.
- c. Solo se consideran interesados quienes no se ven afectados por el proyecto.
- d. Tienen influencia únicamente al inicio del proyecto.

5. ¿Cuál de los siguientes elementos puede ser considerado un entregable de proyecto?

- a. Modelo de datos.
- b. Diagrama de casos de uso.
- c. Informe de pruebas del software.
- d. Manual de usuario.

6. En qué fase de un proyecto de software se busca entender los subsistemas que formarán parte del sistema final:

- a. Etapa de definición.
- b. Etapa de ejecución.
- c. Etapa de inicio.
- d. Etapa de cierre.



7. Dentro de los roles de un proyecto, un *tester* corresponde a:

- a. Rol de gestión.
- b. Rol de desarrollo y cruzado.
- c. Rol de desarrollo.
- d. Rol de consulta.

8. En un cronograma, una secuencia de actividades en rojo significa:

- a. La duración total del proyecto.
- b. Actividades sin recursos asignados.
- c. La ruta crítica.
- d. Actividades con sobrecarga de recursos.

9. El diagrama de Gantt representa el plan de trabajo completo, pero se utiliza especialmente para visualizar:

- a. La asignación de actividades en el tiempo.
- b. Las dependencias entre actividades.
- c. El nivel de importancia de cada actividad.
- d. Los recursos asignados por tarea.

10. ¿En cuál de los diagramas usados para representar el cronograma se evidencian con mayor claridad las dependencias entre actividades?

- a. Diagrama de Gantt.
- b. Diagrama de barras.
- c. Diagrama de actividades.
- d. Diagrama Pert.

[Ir al solucionario](#)



## **Resultado de aprendizaje 4:**

Identifica una metodología de desarrollo acorde a las características de un proyecto de software.

Para alcanzar el resultado de aprendizaje 4, se abordará la selección de la metodología de desarrollo más adecuada para un proyecto de software, optimizando así el proceso de desarrollo. Para ello, se abordará el análisis de requerimientos funcionales y no funcionales, la elaboración del documento de requerimientos y el proceso de ingeniería de requerimientos, que incluye la adquisición, análisis y validación de los mismos.

### **Contenidos, recursos y actividades de aprendizaje recomendadas**

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



### **Semana 5**

#### **Unidad 4. Análisis de software**

##### **4.1. Conceptos del análisis**

Bien, si ya pudo revisar los videos y descargó el texto indicado, comencemos revisando los conceptos del análisis.

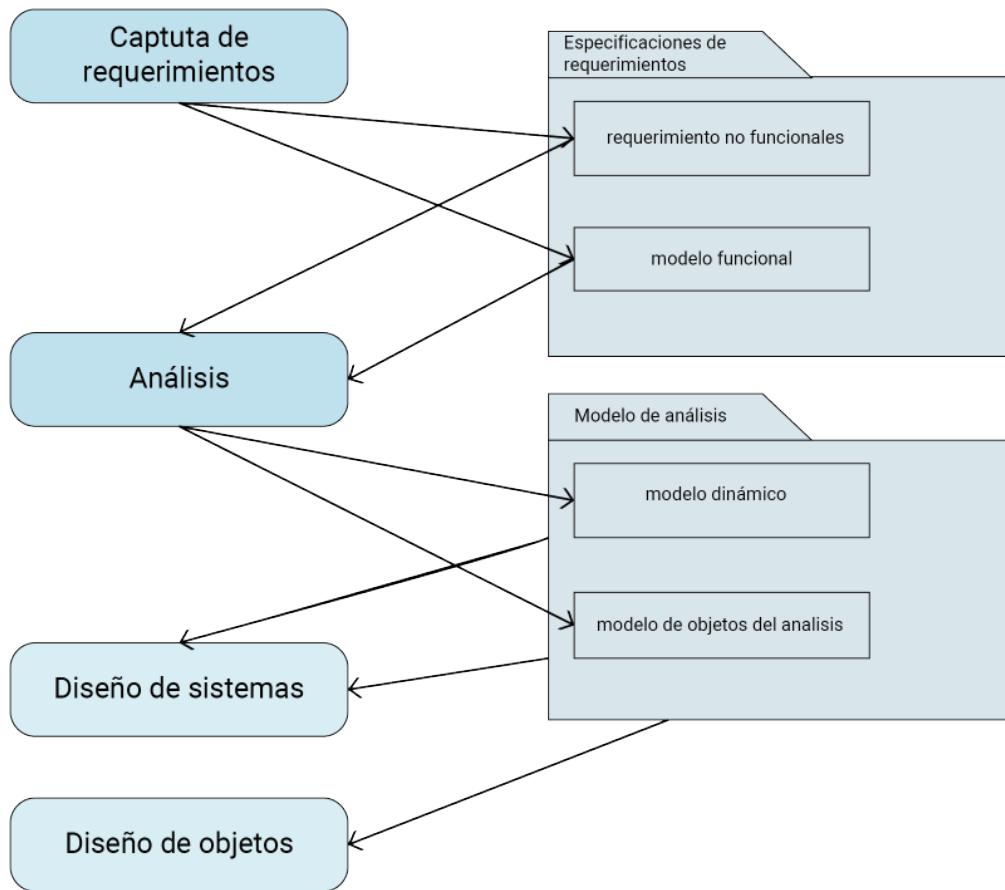
Realice un estudio previo del OCW de la Universidad de Salamanca, el apartado “Análisis orientado a objetos” descargando el documento tema 4: [Análisis orientado a objetos](#).

El problema es cómo crear modelos para describir el dominio de la aplicación utilizando un enfoque Orientado a Objetos, en el cual se especifica el comportamiento del sistema y las formas de interacción con los usuarios.

En la figura 19, se muestra una vista general del proceso, el cual tiene como punto de partida la captura de requerimientos y con ello se desarrollan las actividades que especifican los requerimientos funcionales y el modelo funcional; estas descripciones sustentan el modelo de análisis, el cual incorpora dos componentes esenciales que son: el modelo dinámico y el modelo de objetos del análisis.

**Figura 19**

*Vista general del proceso de IR*



Nota. Jaramillo, D., 2025.

Por tanto, el modelo de análisis está conformado por tres modelos individuales, cada uno de los cuales refleja aspectos diferentes del sistema, que serán usados posteriormente en la etapa de diseño. Estos modelos son:

- El Modelo funcional, que comprende los casos de uso y escenarios.
- El Modelo de objetos del análisis, representado por los diagramas de clases y objetos.
- El Modelo dinámico, representado por la máquina de estados y los diagramas de secuencia

Estimado estudiante, se le recuerda que si no comprende alguno de estos conceptos, tales como casos de uso, clases, objetos, máquinas de estados, etc. Para una mejor comprensión puede revisar el documento de [Advanced Praise for The UML](#), de los autores Rumbaugh, Jacobson, y Booch donde se explica claramente el esquema y funcionamiento de UML.

Para comprender el modelo de análisis, es necesario revisar algunos conceptos que configuran su estructura y funcionamiento.

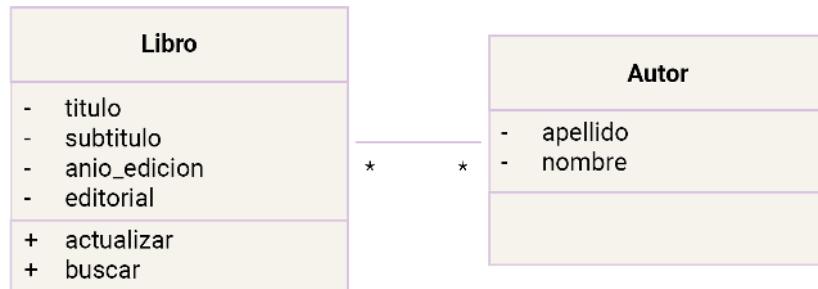
#### **4.1.1. Modelo de objetos del análisis.**

Hace énfasis en la representación del sistema que se construirá desde el punto de vista del usuario, el modelo de objetos del análisis muestra los conceptos individuales que son manipulados por el sistema incluyendo las relaciones con otros objetos.

Los objetos del análisis representan mediante diagramas UML los conceptos que son visibles para el usuario, entre ellos se tiene: clases, atributos y operaciones. En la figura 20. se denota dos conceptos referidos a un dominio de una biblioteca, en ellos tenemos la clase Libro, asociada a la clase Autor, cada una con sus respectivos atributos y operaciones.

**Figura 20**

Ejemplo de diagrama de clases UML



Nota. Jaramillo, D., 2025.

### Modelo dinámico

Es un modelo que se enfoca en modelar el comportamiento del sistema; su representación es a través de diagramas de secuencia, los cuales muestran las interacciones entre un grupo de objetos durante una ejecución particular de un caso de uso y, por otro lado, tenemos las máquinas de estados que se usan para representar el comportamiento de un objeto particular.

El modelo dinámico sirve para asignar responsabilidades a clases individuales y, durante el proceso, para identificar nuevas clases, asociaciones, y atributos que deberán añadirse al modelo de objetos del análisis.

No olvide que estos modelos representan conceptos a nivel de usuario, no son componentes reales del software, por tanto, no se debe incluir clases relacionadas con bases de datos, subsistemas, gestor de sesiones, red; sin embargo, estas deben verse como abstracciones de alto nivel de aquellas que se usarán posteriormente en el diseño.

#### 4.1.2. Objetos entidad, frontera y control

De acuerdo a (Bruegge & Dutoit, 2018), el modelo de análisis es en una abstracción de clases o subsistemas del diseño, la cual cumple con las siguientes características:

- Las clases del análisis se enfocan en la representación de los requerimientos funcionales; los no funcionales se consideran como requerimientos especiales y no se tienen en cuenta hasta el diseño e implementación del sistema. Por tanto, estas clases son más conceptuales, es decir son más evidentes en el dominio del problema.
- No es usual que las clases del análisis ofrezcan una interface en término de operaciones, pero su comportamiento se define mediante la asignación de responsabilidades expresadas a alto nivel y de manera menos formal.
- En lo referente a los atributos, estas clases definen aspectos más conceptuales y son reconocibles en el dominio del problema, a diferencia de las clases del diseño, en las cuales su especificación es cercana a lo que se representa en los lenguajes de programación.

Además, establece que las clases del análisis siempre encajan en uno de tres estereotipos, que se especifican a continuación:

- Objetos **entidad**, se utilizan para modelar información cuyo tiempo de vida es extenso y por lo general es persistente, estas clases se usan para modelar información o un concepto del mundo real, sobre el cual el sistema necesita dar seguimiento.
- Objetos **interface**, se usan para modelar la interacción entre el sistema y los actores. Esta interacción a menudo implica recibir, presentar información y peticiones desde y hacia los usuarios y los sistemas externos. Los objetos interface usualmente representan abstracciones de: ventanas, formularios, paneles, interfaces de comunicaciones, interfaces de impresoras, sensores, terminales, y API. No obstante, estas clases deben mantenerse a un nivel conceptual, es decir, se especifica lo que se obtiene con la interacción, sin decir cómo se ejecuta físicamente.

- Objetos **de control**, representan coordinación, secuencia, transacciones y control de objetos, se usan para encapsular el control de un caso de uso en concreto; también se incluye la representación de derivaciones y cálculos complejos tales como la lógica del negocio, que no puede asociarse con ninguna información concreta de larga duración, almacenada por el sistema.

Para modelar los aspectos dinámicos del sistema, se usan clases de control, debido a su capacidad para gestionar las acciones y los flujos de control principales, y asignando tareas a los objetos entidad e interface.

La notación UML para estos tres tipos de objetos, ofrece dos posibilidades, las cuales se muestran en la figura 21.

**Figura 21**

*Estereotipos para representar objetos entidad, control e interface.*

Alternativa 1:



Alternativa 2:

«entity»  
Cuenta

«Boundary»  
Interfaz del cajero

«control»  
Retirada de efectivo

Nota. Jaramillo, D., 2025.

Note que los nombres de los estereotipos en la alternativa 2, se han colocado en idioma inglés, debido a que estos son estándares.

Para ilustrar estos conceptos, analice el ejercicio siguiente, el cual, por simplicidad, se referirá a un elemento muy conocido en el mundo real, evitando la necesidad de hacer un proceso de especificación muy detallado.

Antes de revisar los resultados, tómese unos minutos para analizar el planteamiento, luego elabore una lista de objetos que logra identificar y clasifíquelos de acuerdo a los estereotipos indicados, finalmente, represéntelos gráficamente utilizando ambas alternativas de representación.

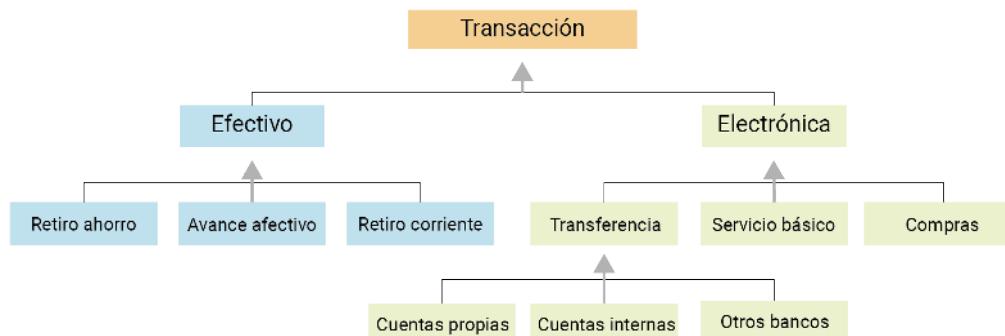
## Generalización y especialización

Por definición, la relación de herencia, permite organizar conceptos en jerarquías, el elemento en la parte superior de la jerarquía es el concepto más general y el elemento al final de la jerarquía es el más especializado, y entre los cuales, puede haber varios niveles intermedios.

La herencia puede tener origen en dos enfoques, el primero se conoce como generalización, que identifica conceptos abstractos, partiendo de conceptos de bajo nivel. Por otro lado, la especialización es una actividad de modelado que partiendo de conceptos generales o de alto nivel, identifica conceptos más específicos en la subclase.

En la figura 22, se muestra un ejemplo de una jerarquía de clases, obtenida a partir de la clase transacción, en ella se evidencian los diferentes tipos de transacciones que puede atender el cajero automático, observe que el criterio para obtener las subclases o clases hijas puede variar en función de los atributos en común y el propósito del sistema.

**Figura 22**  
*Jerarquía de clases*



Nota. Jaramillo, D., 2025.

## 4.2. Actividades del análisis

Una vez que ha revisado algunos de los conceptos más importantes de la etapa de análisis, vamos a explicar las actividades que permitirán transformar los casos de uso y escenarios en un modelo de análisis.

Según (Bruegge & Dutoit, 2018) , las actividades del análisis son las siguientes:

- Identificar objetos entidad
- Identificar objetos interfaz
- Identificar objetos de control
- Mapear casos de uso a objetos mediante diagramas de secuencia.
- Modelar interacciones entre objetos, mediante tarjetas CRC
- Identificar asociaciones
- Identificar agregaciones
- Identificar atributos
- Modelar el comportamiento dependiente del estado de objetos individuales.
- Modelar relaciones de herencia.
- Revisar el modelo de análisis.

La identificación de los diferentes tipos de objetos es el primer paso, luego es necesario crear diagramas de secuencia que nos permitan finalmente construir un diagrama de clases del análisis completo con sus respectivos atributos y relaciones.

El modelo de análisis también incluye el diagrama de estados, también conocido como máquina de estados.

Finalmente, es necesario validar el modelo de análisis, para ello es necesario chequear una serie de características establecidas en la tabla 8.

**Tabla 8***Características para validar el modelo de análisis*

Aspecto	Características
Correcto	<ul style="list-style-type: none"><li>• Glosario de objetos entidad comprensible para usuarios.</li><li>• Clases abstractas corresponden a conceptos de usuario.</li><li>• Descripciones correspondientes con definiciones de usuario.</li><li>• Nombres de objetos entidad e interfaz son sustantivos claros.</li><li>• Nombres de casos de uso y objetos de control son verbos.</li><li>• Errores identificados y gestionado.</li></ul>
Completo	<ul style="list-style-type: none"><li>• Todo objeto es requerido por al menos un caso de uso.</li><li>• Todo objeto se crea, modifica o destruye en un caso de uso.</li><li>• Todos los objetos accesibles desde un objeto interfaz.</li><li>• Procesos de inicialización de atributos identificados.</li><li>• Todos los atributos tienen su tipo de dato.</li><li>• Se identifican atributos cualificados.</li><li>• Asociaciones claras, multiplicidad definida.</li><li>• Cada objeto de control puede acceder a los objetos que participan en el caso de uso.</li></ul>
Consistente	<ul style="list-style-type: none"><li>• Nombres de casos de uso o clases únicos en el modelo.</li><li>• Si hay nombres similares, deben corresponder al mismo elemento.</li><li>• Herencia correctamente aplicada.</li></ul>



Aspecto	Características
Realista	<ul style="list-style-type: none"> <li>• Todas las características son factibles de implementar.</li> <li>• Evidencia clara de que se puede cumplir con requerimientos de rendimiento y confiabilidad.</li> </ul>

Nota. Jaramillo, D., 2025.

#### a. Identificar objetos entidad

Los objetos participantes forman las bases de un modelo de análisis, estos se identifican examinando cada caso de uso e identificando objetos candidatos. Para ello se debe trabajar en base a heurísticas como la de (Ahmed, 2016), que, de manera intuitiva, ayudan a identificar los elementos necesarios del modelo de clases, tales como objetos, atributos y asociaciones a partir de la especificación de requerimientos como se muestra en la tabla 9

**Tabla 9**

*Heurísticas de Abbot para el mapeo de partes del dialogo hacia modelos de componentes*

Parte del texto	Componente del modelo	ejemplo
Nombre propio	instancia	Alice
Nombre común	clase	oficial de campo
verbo en acción	operación	crea, emite, selecciona
verbo ser	herencia	es una clase de, es uno de
verbo tener	agregación	tiene, consta de, incluye
verbo modal	restricción	debe ser
adjetivo	atributo	descripción

Nota. Jaramillo, D., 2025.

El análisis del lenguaje natural tiene la ventaja de que se enfoca en los términos del usuario, no obstante, también se presentan limitaciones, como la imprecisión del lenguaje y, por tanto, la calidad depende del estilo de redacción del analista. Los desarrolladores pueden resolver esas limitaciones reescribiendo y clarificando la especificación de requerimientos, de modo que identifiquen y estandaricen objetos y términos. Otra limitante del lenguaje natural es que existen más sustantivos que clases relevantes, muchos de ellos corresponden a atributos o sinónimos de otros, esto hace que la tarea de identificar clases en especificaciones de software muy grandes consuma mucho tiempo.

Por consiguiente, la lista de heurísticas de Abbot trabaja bien para generar una lista de objetos candidatos a partir de descripciones cortas, tales como el flujo de eventos de un escenario o un caso de uso.

Como complemento a la heurística de Abbot, se puede usar la heurística para identificar objetos entidad propuesta en (Bruegge & Dutoit, 2018), que se describe a continuación:

Heurística para identificar objetos entidad: Se considerarán objetos entidad a los siguientes casos:

- Términos que los desarrolladores o usuarios necesitan para comprender el caso de uso.
- Sustantivos recurrentes en los casos de uso.
- Entidades del mundo real que el sistema necesita mantener.
- Actividades del mundo real que el sistema necesita conservar
- Fuentes de datos

La identificación de objetos entidad, debe contener un nombre en los términos que usan los usuarios y especialistas del negocio, agregando una breve descripción y listando algunos atributos que, de los objetos; esto permitirá a los desarrolladores tener mayor claridad respecto del dominio del problema.



Recuerde que los objetos entidad son todos aquellos elementos que pertenecen al dominio del problema y que el sistema necesita tener control.

### b. Identificar objetos interface

Como se había indicado anteriormente, los objetos interface (*boundary*) representan el mecanismo de comunicación del sistema con actores humanos o con otros sistemas, por lo tanto, se trata de identificar a todos los objetos que permitirían esta interacción, para ello, recolectan la información desde el actor y la trasladan a un formulario que puede ser utilizado tanto por objetos entidad, como por objetos de control.

Los objetos interface, no necesitan demasiados detalles de cómo se implementarán, pero sí es importante que especifiquen cómo deberá ser la interacción. Para identificar objetos Interface utilizaremos la siguiente heurística, propuesta por (Bruegge & Dutoit, 2018) Para identificar objetos interface, considere lo siguiente:

- Identifique controles de la interface que el usuario necesitará para iniciar el caso de uso, por ejemplo, BotondeReportedeEmergencia.
- Identifique formularios que se necesitaría el usuario para ingresar información al sistema. Por ejemplo, FormularioReporteEmergencia.
- Identifique avisos y mensajes que el sistema utiliza para responder al usuario.
- Cuando múltiples actores están involucrados en un caso de uso, identifique terminales de actor (ej. TerminalDespachador) para referirse a la interface en consideración.
- No modele aspectos visuales de la interface con objetos de interfaz de usuario.
- Siempre utilice términos de usuario final para describir interfaces; no utilice términos del dominio de la solución o de la implementación.

### c. Identificar objetos de control

Los objetos de control son responsables de coordinar las acciones entre los objetos entidad e interfaz, y no tienen una contraparte en el mundo real, usualmente se da una relación cercana entre objetos de control y casos de uso, el objeto de control se crea al inicio del caso de uso y desparece a su final. Estos objetos son los responsables de recolectar la información proveniente de los objetos interfaz.

Para identificar objetos de control, Bruegge y Dutoit (2010) proponen la siguiente heurística:

- Identifique un objeto de control por cada caso de uso.
- Identifique un objeto de control por actor en el caso de uso.
- La línea de vida de un objeto de control, debe cubrir la extensión del caso de uso o la extensión de la sesión de usuario. Si es difícil

identificar el inicio y el final de la activación de un objeto de control, el caso de uso correspondiente probablemente no tiene bien definidas las condiciones de entrada y de salida.

En este contexto, luego de revisar estos tres tipos de identificación, el siguiente módulo didáctico presenta, de manera guiada y aplicada, los casos de estudio organizados por tipo de identificación (entidad, interfaz y control). Se invita a revisarlo para comprender criterios, ejemplos estructurados y buenas prácticas que facilitan la construcción de modelos coherentes y verificables.

### Casos de estudio de identificación de objetos: entidad, interfaz y control

Concluida la revisión de los casos, se reconoce que la identificación de entidades, interfaces y controles constituye un fundamento esencial para estructurar modelos coherentes y garantizar la trazabilidad en sistemas orientados a objetos.

### **Mapeo de casos de uso a objetos mediante diagramas de secuencia**

Una vez identificados los objetos, se dibuja los diagramas de secuencia, estos se utilizan para representar el comportamiento de los objetos durante la ejecución de uno de los escenarios de un caso de uso, y a pesar de ser modelos no tan adecuados para comunicarse con el usuario, son mucho más precisos que los casos de uso para explicar el comportamiento del sistema, y permiten a los desarrolladores descubrir nuevos objetos, o aclarar aspectos confusos en la especificación de requerimientos.

Deberá dibujar un diagrama de secuencia por cada escenario relevante que se identifique en un caso de uso, se debe desarrollar un diagrama de secuencia por cada escenario. Para dibujar diagramas de secuencia, se debe tener en cuenta las consideraciones de la heurística siguiente, propuesta:

- La primera columna corresponde al actor que inicia el caso de uso.
- La segunda columna debe ser el objeto de control que usa el actor para iniciar el caso de uso.

- La tercera columna debe ser el objeto de control que gestiona el resto del caso de uso.
- Los objetos de control son creados por objetos interfaz que inician los casos de uso.
- Los objetos interfaz son creados por los objetos de control.
- Los objetos entidad son accedidos por objetos interfaz y por objetos de control.
- Los objetos entidad nunca acceden a los objetos interfaz objetos de control

## Identificación de asociaciones

Las asociaciones muestran las relaciones entre dos o más clases, por ejemplo, en un sistema académico tendríamos una lista de clases como: asignatura, docente, estudiante, matrícula, podríamos identificar asociaciones entre ellas: docente dicta asignatura, estudiante obtiene matrícula.

Para identificar asociaciones, es necesario tener en cuenta la heurística de Abbot, complementada con la establecida por (Bruegge & Dutoit, 2018), ello facilitará la tarea de identificar asociaciones.

- Examine los verbos.
- Nombre asociaciones y roles de manera precisa.
- Utilice calificadores tan frecuentemente como le sea posible, de modo que pueda identificar namespaces y atributos clave.
- Elimine cualquier asociación que pueda ser derivada de otras.
- No se preocupe por la multiplicidad hasta que el conjunto de asociaciones se estabilice.
- Demasiadas asociaciones, pueden hacer el modelo ilegible.

Vale resaltar la importancia de tener un número pequeño de asociaciones, de modo que sea fácil su lectura y comprensión, si no sucede esto los desarrolladores deben evaluar el modelo identificar todas las asociaciones que podrían resultar redundantes, conviene tener presente que mientras menos asociaciones se coloquen es mejor, siempre y cuando no se pierda información.

La identificación de asociaciones ofrece a los desarrolladores dos ventajas, primero la posibilidad de clarificar el modelo de análisis haciendo explícitas las relaciones entre objetos y en segundo lugar permite descubrir casos límite o que podrían considerarse excepcionales, por ejemplo, podría preguntarse, ¿sólo un OficialdeCampo reporta una emergencia?, o ¿es posible que más de uno lo haga?, o ¿puede haber reportes de emergencia anónimos?

Las asociaciones tienen las siguientes propiedades:

- Nombre, Es un atributo opcional y necesariamente único en todo el modelo, es una palabra o frase corta que describe la forma como se asocian las dos clases.
- Rol, Es una palabra, normalmente un sustantivo que identifica la función de cada clase con respecto a la asociación.
- Multiplicidad, Se coloca en cada extremo de la asociación y sirve para describir el número de instancias de la clase que se asocian con una instancia de la otra clase.

### Identificación de agregaciones

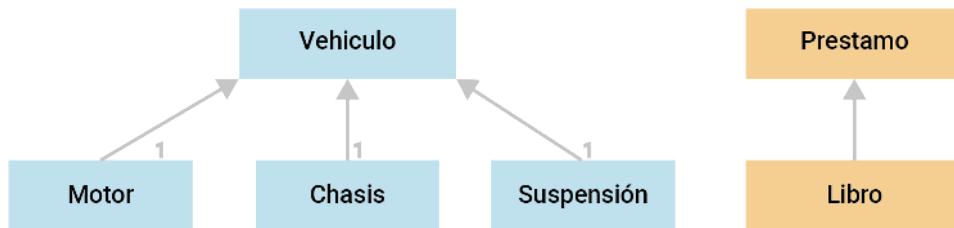
Las agregaciones son un tipo especial de asociaciones que denotan la relación Todo-Parte, existen dos tipos: la agregación de composición y la agregación compartida, la diferencia entre las dos es que la composición se utiliza para representar agregados estructurales, es decir agregaciones en las cuales, si desaparece el todo, desaparecen todas sus partes, en cambio en la agregación compartida ambas clases existen independientemente.

La agregación se representa como una asociación con la diferencia de que se coloca un rombo en la punta de la línea a la que llega el Todo, y en el caso de la agregación de composición el rombo se rellena totalmente como en la figura 23, en la cual se muestran dos ejemplos, uno de agregación y otro de composición. La composición que se encuentra a la izquierda denota la relación de un vehículo con sus partes, de acuerdo al sentido de la relación si desaparece el vehículo, desaparecen sus partes, en el caso de la derecha la

relación indica que un préstamo incluye muchos libros, sin embargo, no hay una dependencia estructural entre los libros y el préstamo, puesto que estos se pueden usar en otras instancias del préstamo.

**Figura 23**

Ejemplos de agregación por composición (izquierda) agregación compartida (derecha)



Nota. Jaramillo, D., 2025.

### Identificación de atributos

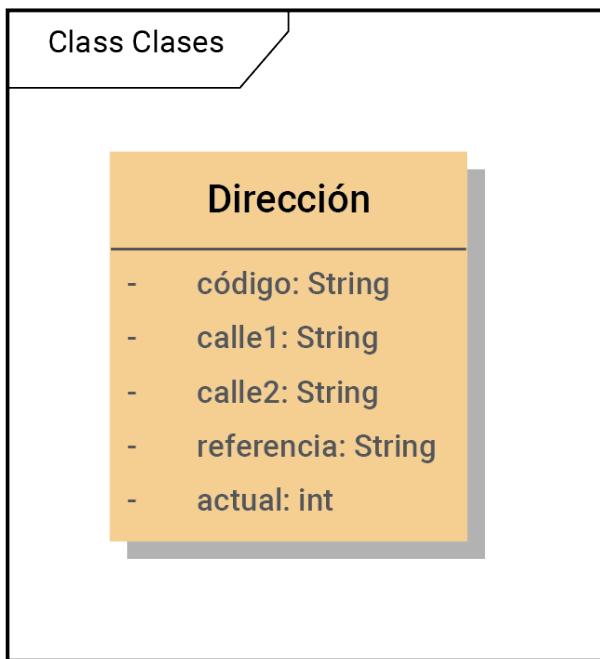
Los atributos son las propiedades (datos) de cada uno de los objetos individuales, al momento de identificarlos es necesario realizar un ejercicio de abstracción, es decir, se debe colocar únicamente los atributos que sean relevantes para la aplicación o el modelo de dominio para el que estamos modelando; los atributos son la especificación de estas propiedades y cada objeto tiene un valor para cada uno de ellos. Para identificar atributos, es necesario determinar la información que guarda cada actor de los objetos sobre los que opera durante los escenarios o casos de uso descritos y la definición de cada atributo implica la especificación del tipo de dato que se almacenará.

Es recomendable, además, identificar todas las asociaciones posibles entre las clases para evitar que se coloque como atributos, a características propias de las asociaciones.

En la figura 24 se puede apreciar los atributos para la <<clase Direccion>>, las cuales se identifican

**Figura 24**

*Atributos de la clase Dirección*



Nota. Jaramillo, D., 2025.

#### 4.3. Gestión del análisis

Para completar el modelo de análisis es necesario gestionarlo, ello implica documentar el modelo de análisis y asignar responsabilidades.

Estimado estudiante, estas actividades tienen un nivel de complejidad mayor, por ello asegúrese de haber comprendido la temática, los ejercicios y sobre todo de haber desarrollado con éxito la actividad propuesta antes de pasar al siguiente tema.

Las actividades del análisis no involucran simplemente el aspecto técnico, sino que es necesario gestionarlo, de forma tal que los equipos que trabajan en dicha actividad lo hagan coordinadamente y garanticen la consistencia en el uso de los recursos.

En este apartado se establece un formato para documentar los resultados de la etapa de análisis, luego la asignación de roles al equipo de trabajo y finalmente, el cómo manejar algunos incidentes relacionados a los procesos iterativos e incrementales.

#### **4.3.1. Documentación del análisis**

La documentación es un aspecto esencial en las actividades de desarrollo de software, muchas de las veces los desarrolladores se centran en producir el software, y no dedican mucho tiempo a documentar, sin embargo, hay que insistir en que la documentación es esencial para apalancar las siguientes etapas del proceso de desarrollo, y esto, independientemente de la metodología de desarrollo de software que vayamos a adoptar.

Al documento que describe las especificaciones necesarias lo denominaremos como DAR (Documento de Análisis de Requerimientos) y básicamente consiste en una versión ampliada del documento de requerimientos de software DRS, e incluye nueva información obtenida de las actividades del análisis.

El esquema básico del documento es el que se muestra en la siguiente figura, no obstante, existen muchas versiones y plantillas de documentación del análisis que pueden utilizarse, es preferible de hecho utilizar plantillas estandarizadas en la organización para la que se desarrolla o la que desarrolla el producto, y alternativamente se puede utilizar estándares internacionales de documentación de software como el ISO/IEC 26514:2008.

**Figura 25**

*Estructura del documento*

Documento de análisis de requerimientos	
1.	Introducción
2.	Sistema actual
3.	Sistema propuesto
3.1.	Visión general
3.2.	Requerimientos funcionales
3.3	Requerimientos no funcionales
3.4	Modelos del sistema
3.4.1.	Escenarios
3.4.2.	Modelo de casos de uso
3.4.3	Modelo de objetos
3.4.3.1.	Diccionario de datos
3.4.3.2.	Diagrama de clases
3.4.4	Modelos dinámicos

*Nota.* Jaramillo, D., 2025.

### **Asignación de responsabilidades**

Como se había expuesto anteriormente, el proceso de desarrollo de software no es una tarea que se puede hacer individualmente, especialmente en la actividad de análisis se requiere la participación de un gran número de individuos, entre clientes, usuarios, equipo de desarrollo con diferentes roles. En la tabla 10, se describen algunos de los roles que asumen los participantes en esta actividad.

**Tabla 10***Roles en el proceso de Análisis*

Rol	Descripción
Usuario final	Es el experto en el dominio de la aplicación, su papel es informar cómo funciona el sistema actual, el entorno de operación del futuro sistema y las tareas que este deberá soportar. Los usuarios finales pueden modelarse como uno o varios actores en función de las responsabilidades que estos tienen con relación al sistema
Cliente	Es quien contrata el sistema, y como propietario define su alcance tomando como referencia los requerimientos de los usuarios. El cliente sirve como integrador de la información relacionada con el dominio y resuelve las inconsistencias que puedan presentarse con relación a las expectativas de los usuarios.
Analista	Es el experto en el dominio de la aplicación, cuya actividad principal es modelar el sistema actual y generar información acerca del futuro sistema.
Arquitecto	Trabaja desde la perspectiva del sistema e intenta integrar el modelo de casos de uso con el modelo de objetos. Provee la filosofía del sistema e identifica omisiones en los requerimientos.
Documentador	Responsable de generar, integrar e indexar la documentación del análisis.

Nota. Jaramillo, D., 2025.

Cuente con la ayuda de su tutor y puede contactarlo por los medios indicados previamente.



## Actividad de aprendizaje recomendada

Es momento de aplicar sus conocimientos a través de las actividades que se han planteado a continuación:

1. Considere el caso de un sistema para controlar el ingreso de personas a un evento académico. El sistema comprende 4 casos de uso como registrar participantes, registrar ingreso al evento, emitir certificados y calificar el evento. Desarrolle lo siguiente:
  - a. Diagrama de casos de uso considerando los actores siguientes: operador, que es quien registra a los participantes, y emite los certificados; controlador, que es quien registra el ingreso de los participantes a cada uno de los eventos, y el participante que califica el evento.
  - b. Especificación de casos de uso.
  - c. Modelo de análisis que incluye: aplicar las heurísticas para identificar los objetos del análisis, elaborar diagramas de secuencia y de estados para refinar las características de los objetos y elaborar el diagrama de clases del análisis.

Nota. Por favor, complete la actividad en un cuaderno o documento Word.

## Contenidos, recursos y actividades de aprendizaje recomendadas



## Semana 6

### Unidad 4. Análisis de software

#### 4.4. Requerimientos

En el presente apartado vamos a estudiar los requerimientos de software, cómo identificarlos, cómo clasificarlos, cómo documentarlos y cómo utilizarlos para que formen parte del proyecto de desarrollo de software.

Para esta unidad, es necesario que usted tenga conocimientos sobre modelado de *software* con UML. Este conocimiento ustedes lo adquirieron en la asignatura de Modelado de sistemas.

#### **4.4.1. Requerimientos funcionales y no funcionales**

Una vez conocidos los conceptos preliminares de la ingeniería de *software* y la importancia del modelado de desarrollo del *software*, empezamos a partir de esta unidad hasta el final de la guía a describir las actividades del proceso de desarrollo de *software*. El objetivo de la presente unidad es la especificación de requerimientos de *software*, mediante un proceso formal conocido como Ingeniería de Requerimientos, cuyas actividades consisten en obtener información, luego analizarla con la ayuda de modelos y finalmente especificar los requerimientos clasificados en funcionales y no funcionales.

En la sección 1.2.4 ya definimos lo que son los requerimientos y se revisaron algunos ejemplos de requerimientos funcionales y no funcionales. En esta sección vamos a profundizar en el concepto, puesto que los requerimientos son la base para iniciar cualquier desarrollo de *software*.

A la hora de construir un *software*, pocos son realmente aquellos proyectos que culminan con éxito, es decir, aquellos que terminan dentro del plazo establecido, con el presupuesto asignado y sobre todo con los requisitos especificados. Una buena parte de proyectos fracasados radica en los problemas que se tienen a la hora de definir los requerimientos de sistema, especialmente cuando los proyectos son complejos.

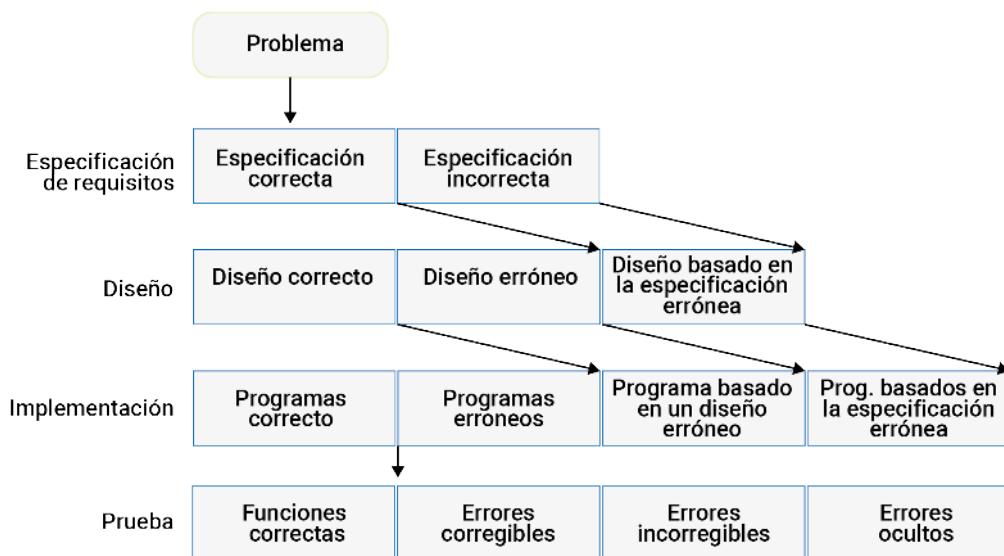
Entre los problemas más comunes que se tienen a la hora de construir software, desde el contexto del usuario, tenemos:

- No saben lo que quieren.
- Un sistema cuenta con muchos usuarios y ninguno tiene una visión de conjunto.
- No saben cómo hacer más eficiente la operación en su conjunto.
- No saben qué partes de su trabajo pueden transformarse en *software*.

- No saben detallar lo que saben de forma precisa.

Se requiere que los requerimientos sean definidos apropiadamente y no contengan errores. No detectar estos errores incrementará los costes (tiempo, dinero) de forma exponencial. En la figura 26 se puede observar cómo los errores que se generan en la etapa de especificación de requerimientos se extienden y afectan las demás etapas del desarrollo de software.

**Figura 26**  
Acumulación de errores



Nota. Adaptado de *Software Engineering: Modern Approaches* [Ilustración], por Braude, E. y Michael, E., 2016, Waveland Press, Inc., CC BY 4.0.

Ante esta situación surge la Ingeniería de Requisitos (IR), que trata de los principios, métodos, técnicas y herramientas que permiten descubrir, documentar y mantener los requisitos para sistemas basados en computadora de forma sistemática y repetible. En la tabla 11 se muestran algunos ejemplos de requerimientos.

**Tabla 11***Requerimientos de usuario y sistema - ejemplos*

Requerimientos de usuario	Requerimientos de sistema
1 El SGA-UTPL permitirá al estudiante consultar su expediente académico a través del dispositivo móvil previo método de autenticación.	2 El sistema debe solicitar al estudiante un usuario y una clave para autenticar su ingreso. 3 El sistema genera un reporte con todas las asignaturas que el estudiante se ha matriculado, no se consideran las asignaturas actuales que aún está cursando. 4 El estudiante dispondrá de la posibilidad de descargar el expediente académico en un formato de archivo PDF.
5 El SGA-UTPL permitirá al estudiante realizar su matrícula de acuerdo a la oferta académica y a la disponibilidad de cupos y paralelos en cada uno de los componentes académicos	6 El sistema permitirá al estudiante escoger las asignaturas ofertadas. 7 El sistema permitirá escoger el horario y paralelo por cada asignatura ofertada. 8 El sistema deberá informar al estudiante sobre la restricción para cursar alguna asignatura. 9 El sistema no mostrará las asignaturas que el estudiante ya tiene aprobadas. 10 El sistema deberá informar la cantidad de créditos en los

Nota. Jaramillo, D., 2025.

Como se puede observar, son definiciones en lenguaje natural que se atribuye a necesidades puntuales desde el punto de vista del estudiante y lo que esto conlleva a definir especificaciones más concretas llamados requerimientos de sistema que es algo más específico y más cercano de lo que se diseñará.

Tal como se indica, con frecuencia los requerimientos del sistema se clasifican como requerimientos funcionales y no funcionales.

Los requerimientos funcionales expresan la naturaleza del funcionamiento del sistema, mientras que los requerimientos no funcionales expresan restricciones sobre el espacio de posibles soluciones.

En concreto los requerimientos funcionales definen “qué debe” hacer el sistema, en cambio que los requerimientos no funcionales definen “cómo debe” ser el sistema. La distinción entre requerimientos funcionales y no funcionales no siempre resulta evidente.

Por ejemplo, Podríamos decir que en un principio la seguridad se puede interpretar como un requerimiento no funcional, no obstante, su elaboración puede conducir a nuevos requerimientos funcionales, como es, la necesidad de autenticar a los usuarios del sistema. Por lo tanto, más allá de si decidimos incluir este tipo de requisitos en una u otra sección, lo importante es identificarlos correctamente.



Para complementar el estudio revise el tema Requisitos, del texto *Ingeniería de Software I*, dónde se analizan los requisitos funcionales y no funcionales, así como el proceso para especificar requisitos.

Finalizada la lectura, podemos acotar que los requerimientos de usuario son más generales mientras que los requerimientos del sistema son más específicos.

Algunas acotaciones respecto de los Requerimientos de Usuario y los requerimientos del sistema, los Requerimientos de Usuario son más generales, por tanto, describen lo que el usuario espera hacer con el sistema; los

requerimientos del sistema son más específicos, por ello describen los servicios que deberá tener el sistema para atender a esas necesidades del usuario.

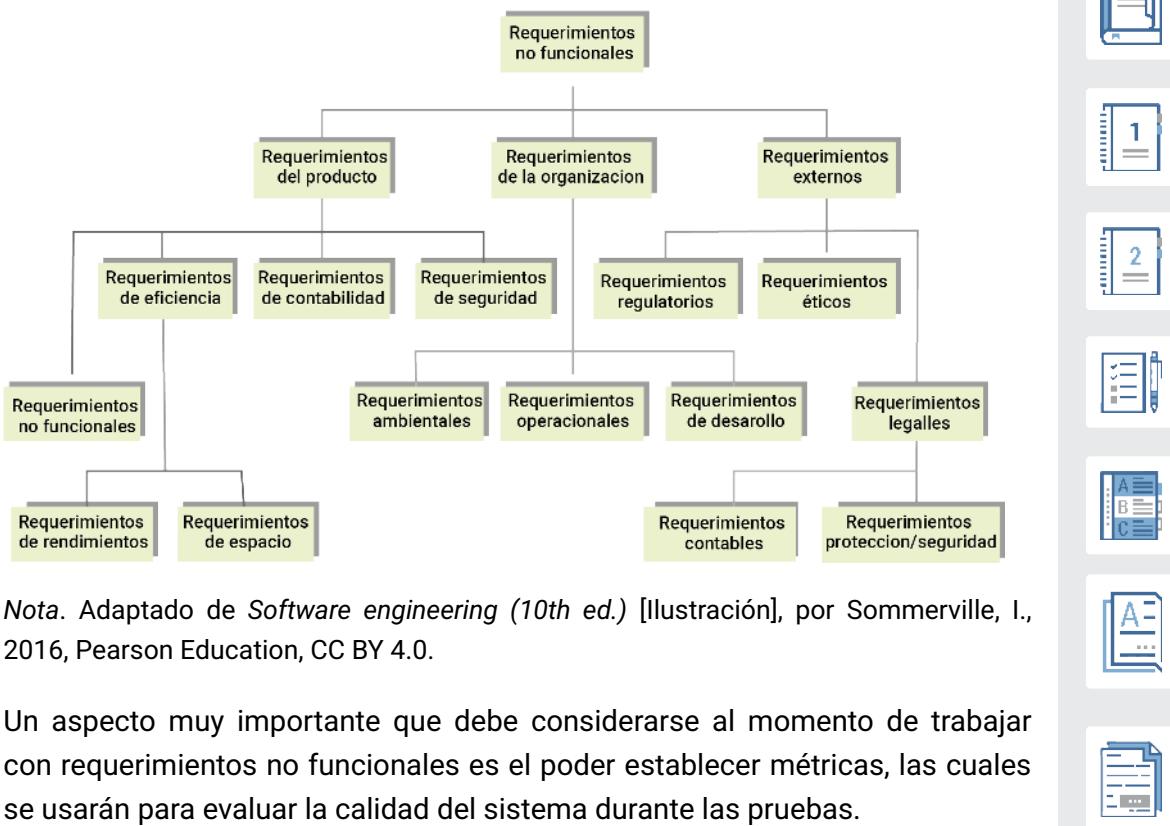
Esto se evidencia con claridad en los ejemplos de la tabla 5 especificados en la guía didáctica, en el primer ejemplo se especifica que “permitirá al estudiante consultar su expediente académico a través del dispositivo móvil previo método de autenticación”, ello deriva, por tanto, lo que el sistema debe hacer para poder cumplir con ese requerimiento de usuario, en este caso sería:

- Solicitar al estudiante un usuario y una clave para autenticar su ingreso,
- Generar un reporte con todas las asignaturas que el estudiante se ha matriculado, sin considerar las asignaturas actuales que aún está cursando,
- Posibilidad de descargar el expediente académico en un formato de archivo PDF.

Luego, la clasificación de los requerimientos del software en requerimientos funcionales y no funcionales es esencial para las siguientes etapas de desarrollo, como se había indicado en la sección 1.2.4 los requerimientos no funcionales son los que establecen los criterios de calidad del sistema y tienen una gran incidencia en el diseño de la arquitectura del sistema. Si analiza la figura 27 encontrará que los requerimientos no funcionales también tienen sus categorías denominadas: requerimientos del producto, requerimientos de la organización y requerimientos externos.

**Figura 27**

*Tipos de Requerimientos no Funcionales*



Nota. Adaptado de *Software engineering (10th ed.)* [Ilustración], por Sommerville, I., 2016, Pearson Education, CC BY 4.0.

Un aspecto muy importante que debe considerarse al momento de trabajar con requerimientos no funcionales es el poder establecer métricas, las cuales se usarán para evaluar la calidad del sistema durante las pruebas.

#### **4.4.2. El documento de requerimientos de software**

Los requerimientos de software deben ser documentados, ya que son el principal insumo del desarrollo de software, cualquiera de las metodologías estudiadas utiliza los requerimientos y dependiendo de si es ágil o tradicional se utilizarán diferentes formatos para documentarlos.

Hay que tener en cuenta que el documento de requerimientos no es útil solo para los desarrolladores, sino que sirve también para los usuarios, administradores y testers.

La especificación de requerimientos, requiere que cada requisito sea debidamente documentado y para ello existen plantillas que han sido desarrolladas de acuerdo al modelo o metodología, siempre basados en un estándar.

Dependiendo de la forma que se desee documentar los requisitos, existen plantillas para documentar requisitos de usuario y otra plantilla para los requisitos de software, pero para ciertos casos se puede utilizar una misma plantilla.

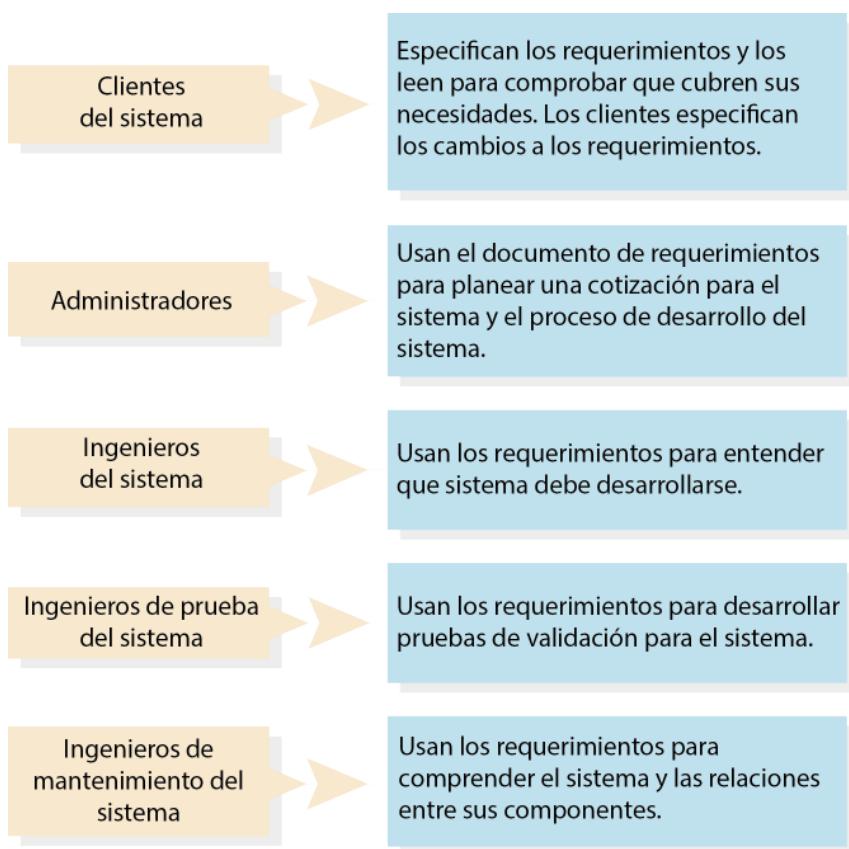
La mayoría de plantillas para documentar la especificación de requisitos de software (ERS), se basan en el estándar IEEE-830. En el [Anexo 1. Plantilla para documentar los requisitos de software](#), se presenta una de las múltiples plantillas para documentar requerimientos de software. Especificación de requisitos. Puede consultar en la web, ejemplos de especificación de requerimientos de software que utilizan diferentes plantillas, algunas con sugerencias de ciertas metodologías.

Tomando como referencia la plantilla que se indica en el anexo 1, la especificación de requisitos consiste en escribir los requisitos de forma coherente, clara y concisa, considerando una serie de atributos.

Para conocer sobre la forma de escribir los requerimientos ponga atención a la figura 28 donde se indican las formas de escribir una especificación de requisitos de sistema.

**Figura 28**

Consideraciones en la especificación de requisitos.



Nota. Jaramillo, D., 2025.

Podemos acotar que la “mejor forma” de escribir requisitos no existe. Lo más utilizado es el lenguaje natural. Cada requisito expresado en una frase corta (“el sistema hará ...”, “el usuario deberá...”, etc.) o lenguaje natural complementado con diagramas y/o notaciones formales, que depende de quien lee o quien escribe los requisitos.

En el documento de la ERS en la sección que corresponde se debe escribir los requisitos a un nivel de detalle suficiente como para permitir a los diseñadores diseñar un sistema que satisfaga estos requisitos, y que permita al equipo de pruebas planificar y realizar las pruebas que demuestren si el sistema satisface, o no, los requisitos. Todo requisito aquí especificado describirá

comportamientos externos del sistema, perceptibles por parte de los usuarios, operadores y otros sistemas. Esta es la sección más larga e importante de la ERS. Deberán aplicarse los siguientes principios (Sajid, 2021):

- El documento debería ser perfectamente legible por personas de muy distintas formaciones e intereses.
- Deberán referenciarse aquellos documentos relevantes que poseen alguna influencia sobre los requisitos.
- Todo requisito deberá ser unívocamente identificable mediante algún código o sistema de numeración adecuado.
- Lo ideal, aunque en la práctica no siempre realizable, es que los requisitos posean las siguientes características: Corrección, no ambiguos, completos, consistentes, clasificados, verificables, modificables y trazables.

Muy importante en este tema es analizar el estándar IEEE-830, y poner atención en las formas de especificar los requerimientos de software, que van desde expresarlos en lenguaje natural y explicarlos utilizando lenguajes estructurados, cada una de estas formas tiene sus ventajas y desventajas.

#### 4.5. Proceso de ingeniería de requerimientos

El trabajo con los requerimientos va más allá de simplemente recolectarlos y documentarlos, tanto es así que existe un proceso denominado Ingeniería de Requerimientos, que comprende cuatro actividades a nivel general que son: recopilación, especificación, validación y gestión de requisitos.

Revise el tema Procesos de ingeniería de requerimientos (Sommerville, 2016). Analice detenidamente la figura de la vista en espiral, dónde se indica cómo el proceso de ingeniería de requisitos permite llegar a especificar los requisitos de sistema.

El proceso de ingeniería de requerimientos consiste en el desarrollo de un conjunto de actividades que se desarrollan de forma iterativa y que están estrechamente relacionadas. Estas actividades dependen del proceso y metodología escogido para el desarrollo del proyecto.

- Adquisición y análisis de requerimientos
  - Descubrimiento de requerimientos
  - Clasificación y organización de requerimientos
  - Priorización y negociación de requerimientos
  - Especificación de requerimientos
- Validación de requerimientos

A continuación, vamos a estudiar y analizar estas actividades.

#### **4.5.1. Adquisición y análisis de requerimientos**

La captura de requerimientos es un proceso que amerita mucho cuidado para los desarrolladores, quienes deben utilizar algunas técnicas y herramientas para capturarlos y en función del modelo de procesos escogido, se capturarán en una sola etapa o se realizará durante las iteraciones. Esto no cambia los instrumentos que se pueden usar.

Desde el punto de vista de la IR, la adquisición u obtención permite en recopilar a través de diferentes estrategias las necesidades de parte de los interesados. El análisis consiste en utilizar apropiadamente las técnicas que permitan representar la información recopilada, para que a su vez pueda ser revisada y validada para continuar con la especificación.

Identificamos claramente las actividades que se relacionan a la obtención de información y las que se relacionan con el análisis de esa información. Es importante aclarar que otros autores no apegados a una metodología en particular sugieren actividades estándares que consiste en: obtención, análisis, especificación y validación. Mientras que las actividades de gestión están relacionadas con la gestión del cambio de los requerimientos.

El descubrimiento de información consiste en recopilar información, para lo cual pueden existir varias fuentes, como es: las personas (que se las conoce como interesados), la documentación existente o información externa a la organización (regulaciones gubernamentales, entre otras). Es función del analista establecer la mejor estrategia para recopilar información. Entre las estrategias más comunes tenemos:

- Entrevista
- Observación directa
- Cuestionario
- Talleres
- Análisis de la documentación, entre otros.

Con la información recopilada se realiza el análisis, y para ello se utilizan estrategias de modelado que ayudan a entender los procesos de la organización. Entre las estrategias de análisis más comunes tenemos:

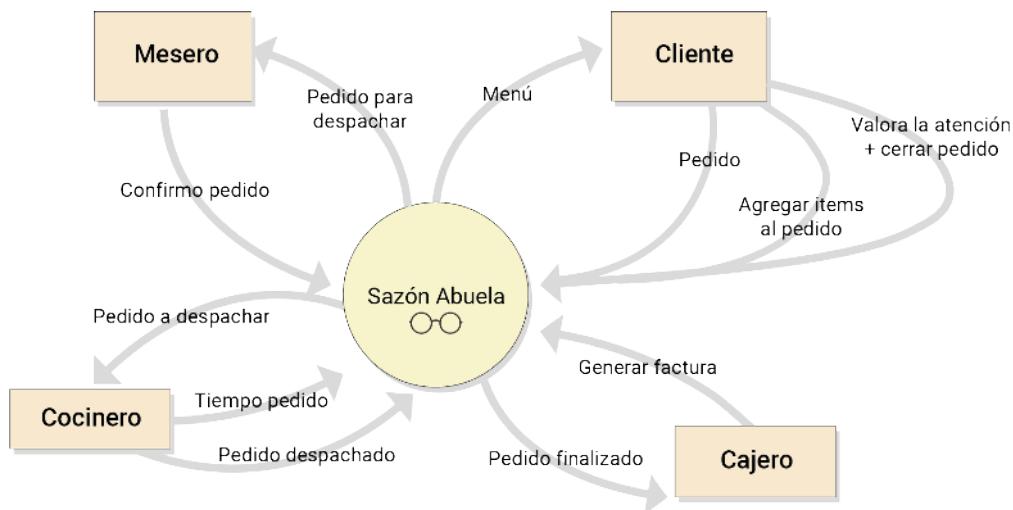
- Diagrama de contexto
- Mapa de proceso
- Escenarios
- Casos de uso
- Prototipos, entre otros.

Tanto las técnicas de obtención, como los modelos de análisis se utilizan cuantas veces sea necesario conforme se descubra la información.

Se pueden utilizar varias herramientas para representar los requerimientos tales como diagramas de contexto, mapas de procesos, escenarios, casos de uso, etc. Para nuestro propósito trabajaremos directamente con la especificación a partir de escenarios y casos de uso, debido al valor que aportan para el proceso de Ingeniería de Software Orientada a Objetos.

A continuación, en la figura 29 podemos ver un diagrama de contextos, este se puede realizar de forma manual si es necesario, su comprensión será entendible para los todos los participantes y podemos tener la primera visión de integración de los elementos del sistema

**Figura 29**  
Diagrama de contexto



Nota. Jaramillo, D., 2025.

¿Podría determinar sobre entorno se desarrollará el sistema? ¿Qué se presente hacer con el sistema? ¿Quiénes participarán?

#### 4.5.2. Validación de requerimientos

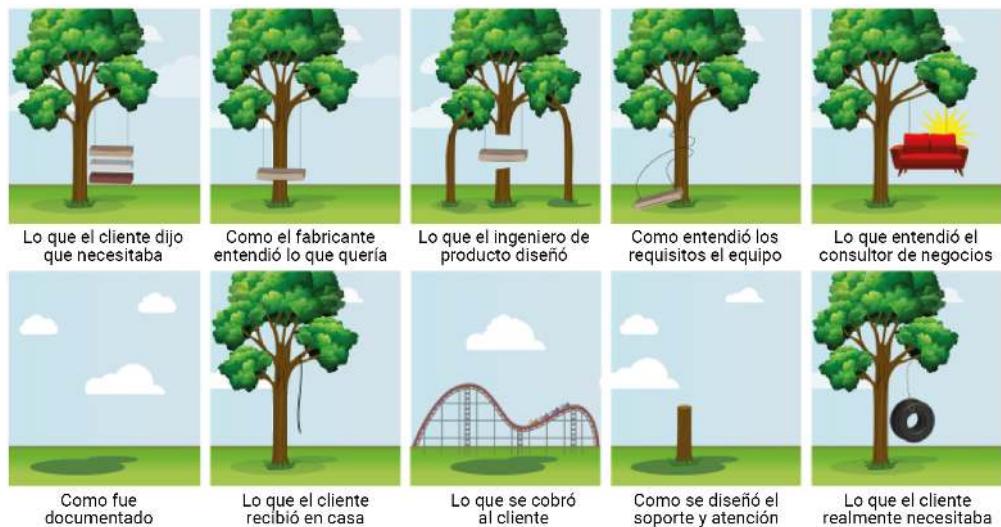
No todos los requerimientos se han obtenido pueden tener el sustento necesario, o pueden reflejar las necesidades reales del sistema, por ello es necesario realizar el proceso de validación.

Es necesario realizar la validación de los requerimientos previamente antes de comenzar con el desarrollo del software. Por lo tanto, los documentos de requisitos están sujetos a procedimientos de validación y verificación. Los requisitos pueden ser validados para asegurar que el ingeniero de software haya entendido los requisitos. También es importante verificar que un documento de requisitos se ajusta a los estándares de la compañía y que es comprensible, consistente y completo.

Un dilema común que suele haber entre los ingenieros de software es el identificar correctamente lo que el cliente necesita, que muchas de las veces difieren de lo que el cliente dice que necesita. En la figura 30 se aprecia brevemente este dilema, el cual refleja varios actores del proceso de desarrollo fallando en la comprensión de los requerimientos.

**Figura 30**

*Dilema entre lo que el cliente quiere y lo el cliente necesita*



Nota. Adaptado de *MI PORTAFOLIO DE ANALISIS ORIENTADO A OBJETOS* [Ilustración], por Aguilar, A. y Samaniego, R., 2010, [blospot](#) CC BY 4.0.

El problema es identificar lo que el cliente realmente necesita, lo que le permitirá resolver los problemas de negocio y así evitar cosas como: reproducir los problemas en el nuevo sistema, o agregar características innecesarias.

Podemos acotar que diferentes interesados que incluyen a representantes del cliente y desarrollador deben revisar el o los documentos de requerimientos. Lo normal es programar el momento adecuado para validar los requisitos, con el fin de cumplir con el objetivo de detectar cualquier problema antes de que los recursos se comprometan a desarrollar los requisitos. La validación de

requisitos se refiere al proceso de examinar el documento de requisitos para asegurarse de que define el software adecuado (es decir, el software que los usuarios esperan).

A continuación, en la tabla 12 puede ver una lista de requerimientos para el caso de estudio propuesto. Como verá aquí ya se utiliza una descripción técnica por cuando se obtienen a partir de una necesidad del cliente el cual no tiene esa experticia.

**Tabla 12**  
*Ejemplo de requerimientos*

Requerimiento	Descripción	Necesidad
Req1.	Registrar en el sistema la entrega de paquetes a los clientes al momento de realizar la entrega en la oficina	Nec2.
Re12.	Registrar la entrega de paquetes a los clientes al momento de la entrega en sus casas	Nec3.
Req3.	Registrar la entrega de paquetes a los clientes a través de la aplicación	Nec2, Nec3
Req5.	Cliente da seguimiento del estado de entrega de sus paquetes, a través de su funcionalidad en un navegador.	Nec4.

Nota. Jaramillo, D., 2025.



### Actividades de aprendizaje recomendadas

Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. En la web Ud. podrá encontrar el libro *Ingeniería de software de Ian Sommerville*, novena edición. Realice una lectura del capítulo 4, le ayudará a comprender más el tema. Además, elabore un pequeño resumen de la figura 4.12.

¿Cómo le fue con las preguntas planteadas en el apartado 4.24?  
Prepare un aporte para participar en la tutoría.

2. Revise los siguientes videos, para profundizar en la temática vista esta semana:

- [Diagrama de contexto FilmMagic](#).
- [Especificaciones de requisitos de software](#) (Universidad Católica de Murcia - UCAM).
- [Especificación de requisitos](#) (Universidad Nacional de Lanús - UNLa).

3. Finalmente, desarrolle la autoevaluación 4 para comprobar sus conocimientos.

### [Autoevaluación 4](#)

En las siguientes preguntas, escoja la opción correcta:

1. En el modelo de análisis, los casos de uso se agrupan dentro de:

- a. Modelo de objetos.
- b. Modelo funcional.
- c. Modelo dinámico.
- d. Modelo de arquitectura.

2. En el modelo dinámico de análisis, ¿qué tipo de diagrama describe cómo un objeto responde a distintos eventos a lo largo de su vida?

- a. Diagrama de secuencia.
- b. Diagrama de clases.
- c. Diagrama de estados.
- d. Diagrama de componentes.

3. Durante el análisis Orientado a Objetos, los objetos de control se caracterizan por:

- a. Ejecutar la lógica de los casos de uso.
- b. Representar la información persistente.
- c. Facilitar la interacción con los actores externos.
- d. Modelar la base de datos.

4. En un sistema de archivos con interfaz gráfica, objetos como archivo, disco y carpeta se consideran:

- a. Objetos de control.
- b. Objetos de entidad.
- c. Objetos de interfaz.
- d. Objetos temporales.

5. Una diferencia conceptual entre generalización y especialización es que:

- a. La generalización abstrae conceptos específicos en uno general, mientras que la especialización concreta un concepto general en casos específicos.
- b. Ambas son sinónimos, no existe diferencia.
- c. La generalización y la especialización siempre generan herencia múltiple.
- d. La generalización solo aplica a objetos y la especialización a clases.

6. Según la heurística de Abbot, la frase “Un hospital está compuesto por médicos y pacientes”, ejemplifica una relación de:

- a. Asociación simple.
- b. Agregación compartida.
- c. Agregación por composición.
- d. Herencia.



7. En la validación del modelo de análisis, cuando se afirma que el modelo se ha estabilizado, significa que:

- a. No volverán a hacerse cambios en absoluto.
- b. El número de cambios pendientes es mínimo.
- c. El modelo ya pasó a la etapa de codificación.
- d. El cliente aprobó sin observaciones.

8. Cuando los requisitos definidos no reflejan las necesidades reales de los usuarios, la causa más probable es:

- a. Mala redacción de los documentos.
- b. No se recolectó la información adecuada.
- c. Uso de una metodología incorrecta.
- d. Falta de herramientas de modelado.

9. El enunciado “El sistema debe soportar 50.000 usuarios simultáneos sin perder rendimiento”, corresponde a un requerimiento:

- a. Funcional.
- b. No funcional del producto.
- c. Organizacional.
- d. Externo.

10. La técnica de validación que permite a los clientes interactuar con un modelo ejecutable del sistema antes de su construcción final es:

- a. Revisión documental.
- b. Creación de prototipos.
- c. Generación de casos de prueba.
- d. Simulación matemática.

[Ir al solucionario](#)



## Resultado de aprendizaje 1 y 2:

- Conoce las principales áreas de conocimiento de la ingeniería de software.
- Conoce e identifica las fases del ciclo de vida de desarrollo de sistemas.

Para alcanzar los resultados de aprendizaje planteados, se promoverá la adquisición de un conocimiento integral de la ingeniería de software, incluyendo sus principales áreas y conceptos fundamentales. Además, se aprenderá a identificar y comprender las fases del ciclo de vida del desarrollo de sistemas, permitiéndoles aplicar estos conocimientos de manera efectiva en la gestión y ejecución de proyectos de software.

### Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



### Semana 7

#### Unidad 5. Diseño de software

##### 5.1. Diseño orientado a objetos con UML

Esta etapa del proceso de diseño comienza con el mapeo de subsistemas a plataformas de hardware y software que se representa en UML con diagramas de despliegue (*deployment diagram*), luego está el mapeo de información persistente, definir políticas de control de acceso, seleccionar el flujo global de control y definir condiciones de operación externas.

Para el mapeo a plataformas de *hardware y software*, de acuerdo con (Bruegge & Dutoit, 2018), es necesario plantearse las siguientes preguntas:

- ¿Cuál es la configuración de hardware del sistema?

- ¿Qué nodo es responsable de qué unidad?
- ¿Cómo realizar la comunicación entre nodos?
- ¿Qué servicios se realizan utilizando componentes de software existentes?
- ¿Cómo se encapsulan estos componentes?

La respuesta a estas preguntas suele derivar en la definición de subsistemas adicionales que permiten mover los datos de un nodo a otro, resolviendo problemas de concurrencia y de confiabilidad.

En el mercado también existen componentes listos para usar que permiten implementar servicios complejos de forma sencilla, entre estos podemos encontrar paquetes de Interfaz de Usuario o sistemas de administración de base de datos que encapsulados debidamente minimizan la dependencia entre componentes.

En cuanto a la gestión de datos, deberíamos preguntarnos lo siguiente:

- ¿Qué datos deberían ser persistentes?
- ¿Dónde se debe almacenar esta información?
- ¿Cómo será el acceso a la misma?

La persistencia de datos es un cuello de botella en el sistema de muchas formas diferentes, la mayor parte de la funcionalidad del sistema realiza operaciones de creación o manipulación de la información, lo cual implica que el acceso a la información persistente debería ser rápido y sobre todo confiable, de lo contrario el sistema sería lento y la información podría perderse fácilmente. Este problema se resuelve normalmente con un Sistema de Gestión de Base de Datos (DBMS) de los existentes en el mercado como Oracle, SQLServer, MySQL o algún otro.

Las políticas de control de acceso, se definen con base en la respuesta a las siguientes preguntas:

- ¿Quién accede a qué información?
- ¿Pueden los controles de acceso cambiar de forma dinámica?
- ¿Cómo se especifica y se implementa el control de acceso?

El control de acceso y la seguridad son grandes preocupaciones del sistema. Este debe ser consistente y la política debe ser capaz de definir quién puede o no acceder a cierta información y si debería ser igual para todos los subsistemas.

En lo relacionado con el flujo de control, nos podemos plantear las siguientes preguntas:

- ¿En qué secuencia se realizan las operaciones en el sistema?
- ¿El sistema es dirigido por eventos?
- ¿Puede el sistema manejar más de una interacción del usuario a la vez?

La selección del flujo de control tiene un alto impacto en las interfaces de los subsistemas. Si se selecciona un flujo de control dirigido por eventos, los subsistemas deberán proporcionar manejadores de eventos. Si en lugar de ello se seleccionan hilos, el subsistema debe garantizar exclusión mutua en secciones críticas.

En cuanto a las condiciones de operación externas (condiciones límite), las preguntas a plantearse son:

- ¿Cómo se levanta y cómo se baja el sistema?
- ¿Cómo se tratan los casos excepcionales?

El inicio y apagado del sistema representan un alto nivel de complejidad de sistema, especialmente en sistemas distribuidos, y tienen mucho impacto en las interfaces de todos los subsistemas.

## 5.2. Diseño del sistema vs. UI

## 5.3. Diagrama de contexto

El diagrama de contexto es una herramienta visual utilizada en el análisis de sistemas para representar de manera global e intuitiva cómo un sistema interactúa con su entorno. Es una técnica fundamental dentro de la ingeniería de software, especialmente en las fases iniciales de análisis y diseño de

sistemas, ya que permite identificar los límites del sistema y las principales interacciones con entidades externas, como usuarios, otras aplicaciones o dispositivos.

En esencia, el diagrama de contexto muestra un solo proceso que representa todo el sistema en estudio, colocado en el centro del diagrama. Alrededor de este proceso central se ubican los actores externos, que pueden ser personas, organizaciones o sistemas que intercambian información con el sistema. Las interacciones se representan mediante flechas que indican el flujo de información o datos entre el sistema y los actores externos. Es importante destacar que estas flechas pueden ser unidireccionales o bidireccionales, dependiendo de si la información fluye en un solo sentido o de manera recíproca.

El propósito principal del diagrama de contexto es proporcionar una visión macro del sistema, evitando entrar en detalles internos, pero dejando claro cómo se relaciona con su entorno. Esto facilita la comprensión de los límites del sistema, es decir, qué está dentro del sistema (lo que controla y gestiona) y qué está fuera (entidades externas que interactúan con él). Por esta razón, es frecuentemente el primer diagrama que se elabora dentro de un proyecto de desarrollo de software, ya que establece una base sólida para diagramas más detallados como los Diagramas de Flujo de Datos (DFD) de niveles inferiores.

En la práctica, la creación de un diagrama de contexto sigue varios pasos: primero, se identifica el sistema central; luego, se determinan los actores externos que interactúan con él; finalmente, se dibujan las interacciones de flujo de datos, especificando el tipo de información que se intercambia. Esta representación permite a analistas y desarrolladores comunicar claramente los requerimientos iniciales del sistema con clientes o partes interesadas, asegurando que todos comprendan el alcance y las responsabilidades del sistema antes de profundizar en su diseño interno.

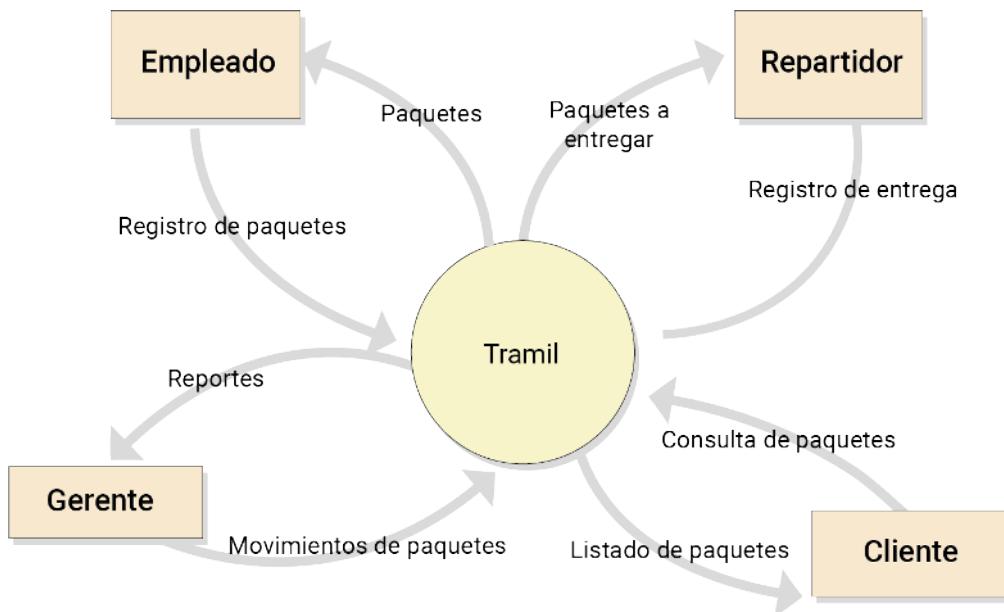
En resumen, el diagrama de contexto es una herramienta de alto nivel que proporciona una visión general, clara y simplificada del sistema y sus interacciones externas. Su fortaleza radica en su simplicidad y capacidad para

delimitar el alcance del sistema, sirviendo como punto de partida para un análisis más detallado y para la identificación de requerimientos funcionales y no funcionales.

En la figura 31 puede encontrar un ejemplo del diagrama de contexto en un nivel 0 del caso de estudio planteado para esta guía TRAMIL, es decir, la forma más general de ir representando las necesidades del usuario a través de este diagrama.

**Figura 31**

Diagrama de contexto-caso de estudio TRAMIL



Nota. Jaramillo, D., 2025.

Nótese que en este diagrama aun no estamos definiendo como se realizará por ejemplo el repartidor con una APP móvil para cuando registre la entrega en la dirección ni tampoco una página WEB para el cliente cuando necesite consultar sobre sus paquetes.

Para conocer lo que es y cómo especificar casos de uso, puede visitar la página web donde encontrará información muy importante sobre los casos de uso.

## 5.4. Modelo de vistas 4+1

El modelo “4+1” de Kruchten, es un modelo de vistas diseñado por Philippe Kruchten y que está acorde con el estándar IEEE 1471-2000 (Recommended Practice for Architecture Description of Software-Intensive Systems) que se utiliza para describir la arquitectura de un sistema software intensivo basado en el uso de múltiples puntos de vista.

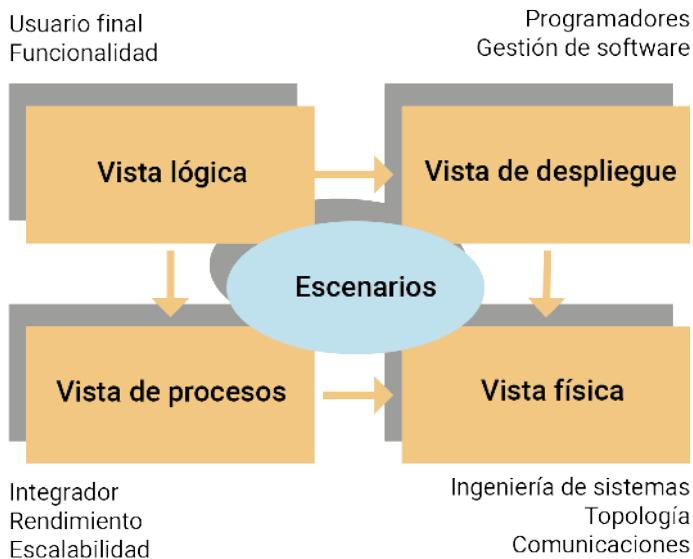
Lo que propone Kruchten es que un sistema software se documente y muestre con 4 vistas bien diferenciadas y relacionadas entre sí con una vista más, llamada vista “+1”. Estas 4 vistas las denominó Kruchten como: vista lógica, vista de procesos, vista de despliegue y vista física y la vista “+1” que tiene la función de relacionar las 4 vistas citadas, la denominó vista de escenario.

Cada una de estas vistas muestran toda la arquitectura del sistema software que se esté documentando, pero cada una de ellas ha de documentarse de forma diferente y ha de mostrar aspectos diferentes del sistema software. En la figura 32 se muestran las vistas del modelo.



**Figura 32**

Modelo de vistas 4+1 de Kruchten



Nota. Adaptado de *Architectural Blueprints – the “4+1” View Model of Software Architecture* [Ilustración], por Krutchen, P., 1995, IEE Software, CC BY 4.0.

Continuemos con el aprendizaje mediante la revisión de los escenarios y los diagramas de casos de uso.

## Casos de uso: escenarios y actores

### 5.4.1. Escenarios: Diagrama de casos de uso

Esta vista consiste en un pequeño conjunto de escenarios importantes (instancias de casos de uso más generales). Los escenarios son una abstracción de los requisitos más importantes, esriben secuencias de interacciones entre objetos, y entre procesos. Se utilizan para identificar y validar el diseño de arquitectura. Esta vista es también conocida como vista de escenarios o casos de uso. Esta vista es redundante con las otras (de ahí el "+1"), pero tiene propósitos distintos: como motor para descubrir los elementos arquitectónicos durante el diseño de la arquitectura y como una función de validación e ilustración.

## Diagramas de casos de uso

Los diagramas de casos de uso son un tipo de diagrama de comportamiento que sirve para describir lo que debe hacer un sistema desde el punto de vista de quien lo va a utilizar. Un modelo de casos de uso se construye mediante un proceso iterativo durante las reuniones, entre los desarrolladores del sistema y los clientes (y/o los usuarios finales) conduciendo a una especificación de requisitos sobre la que todos coinciden. Un caso de uso obtiene las acciones y comportamientos del sistema y cómo los actores interactúan.

Los casos de uso son utilizados durante la elicitation y análisis para representar la funcionalidad del sistema, enfocándose en el comportamiento del sistema desde el punto de vista externo. Un caso de uso describe una función proporcionada por el sistema que produce un resultado visible para un actor. Un actor representa a cualquier entidad que interactúa con el sistema, por ejemplo: usuarios del sistema, otro sistema o el entorno físico del sistema.

Luego de esta pequeña explicación nos adentramos en la construcción del Diagrama de casos de uso, pero muy importante antes que el mismo diseño es encontrar los elementos que vamos a utilizar.

## Actores

En la tabla 13 podemos encontrar los actores del sistema del caso de estudio, aquí una acotación, para poder establecer diferencias con los interesados no se ha considerado al usuario gerente, el mismo que no va a tener interacción con el sistema.

**Tabla 13***Actores del caso de estudio TRAMIL*

Actor	Nombre	Descripción
Act01	Bodeguero.	Persona que utilizará el sistema en la empresa
Act02	Repartidor	Persona que utilizará el sistema a través de un dispositivo, o
Act03	Cliente	Cliente externo a la organización. Dueño de los paquetes
Act04	Administrador	Encargada de las tareas de configuración y administración de funcionalidades del sistema

Nota. Jaramillo, D., 2025.

### Casos de uso del caso de estudio

En la tabla 14 se presenta un listado de casos de uso identificados de nuestro caso de estudio, esta tabla puede cambiar por el proceso de abstracción que se da al momento de trabajar con el caso. Se mapean los elementos que ya han trabajado, actores y requerimientos.

**Tabla 14**Casos de Uso - caso de estudio *TRAMIL*

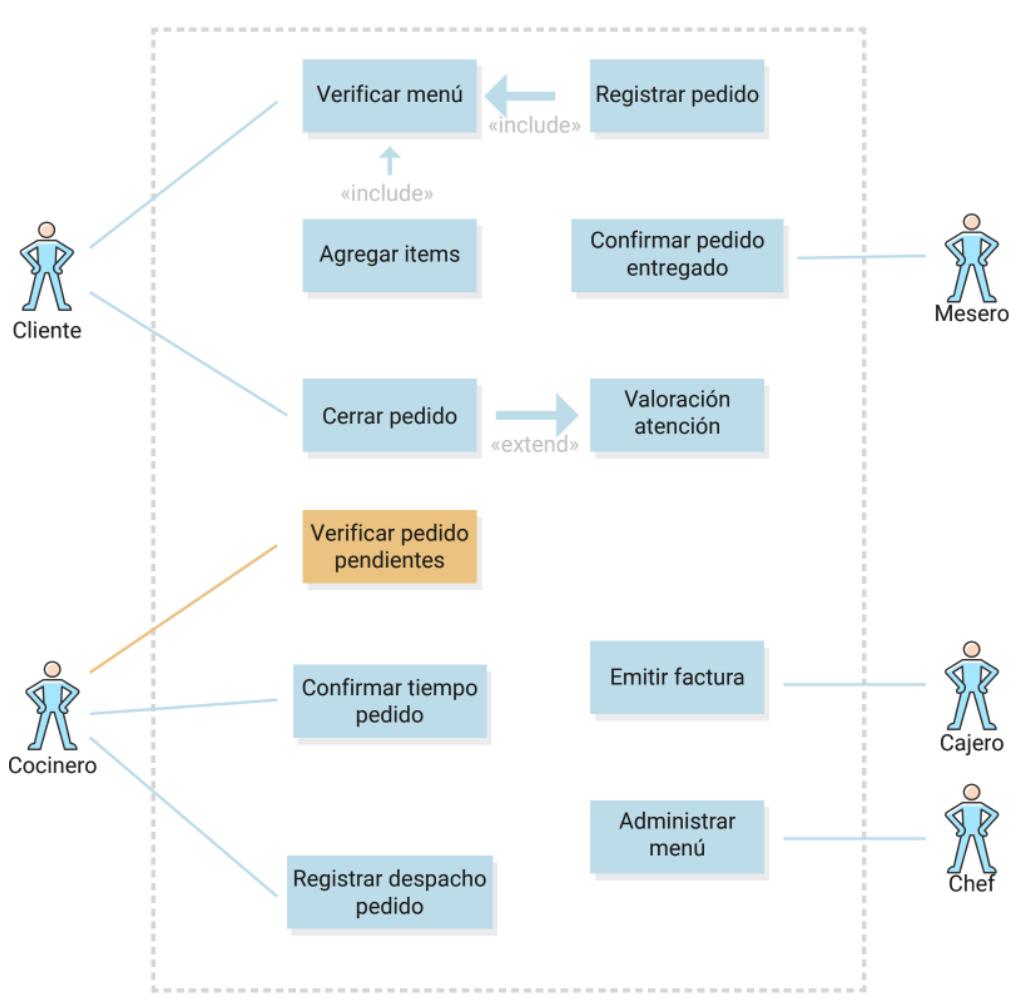
Caso Uso	Nombre	Actor	Req
CU01	RegistrarPaquete	Act01	Req01
CU02	RegistrarEntregaPaquete	Act01, Act02	Req04
CU03	RegistrarCliente	Act01	Req02
CU04	AgregarNuevaDireccion	Act01	Req03
CU05	NotificarCliente		Req04
CU06	RevisarEstadoPaquete	Act03	Req05
CU07	RegistrarseEnSistema	Act03	Req05
CU08	ValidarCliente	Act04	Req07
CU09	Administra Bodegueros	Act04	Req06
CU10	Administrador Repartidores	Act04	Req06

Nota. Jaramillo, D., 2025.

Un ejemplo de desarrollo de un diagrama de casos de uso podemos encontrar en la figura 33.

**Figura 33**

Diagrama de casos de uso



Nota. Jaramillo, D., 2025.

#### 5.4.2. Casos de uso y su especificación

En el ámbito de la ingeniería de software, la especificación de casos de uso es una técnica fundamental para capturar y documentar los requisitos funcionales de un sistema. Un caso de uso describe cómo los actores externos (usuarios u otros sistemas) interactúan con la aplicación para lograr

un objetivo concreto. La especificación, por su parte, consiste en detallar cada caso de uso mediante una narrativa estructurada que incluye el flujo normal de eventos, las excepciones, precondiciones, postcondiciones y reglas asociadas.

Este proceso tiene como finalidad principal ofrecer una visión clara y comprensible tanto para los desarrolladores como para los clientes o usuarios. A diferencia de los documentos técnicos complejos, la especificación de casos de uso utiliza un lenguaje cercano a la realidad del negocio, lo cual permite que todas las partes interesadas comprendan de manera uniforme cómo funcionará el sistema. Por ejemplo, en un sistema de gestión académica, un caso de uso podría ser “Registrar estudiante”, especificando paso a paso la interacción entre el administrador y la aplicación hasta completar el registro.

La importancia de esta práctica radica en varios aspectos. Primero, permite reducir ambigüedades en la fase de análisis, ya que cada escenario se documenta de forma precisa, evitando malentendidos entre el cliente y el equipo de desarrollo. Segundo, los casos de uso sirven como base para el diseño del sistema, pues a partir de ellos se pueden definir diagramas UML, modelos de clases y arquitecturas que soporten las interacciones descritas. Tercero, constituyen una referencia directa para las pruebas de software, dado que cada flujo de un caso de uso puede transformarse en un caso de prueba que valide el correcto funcionamiento del sistema.

Además, la especificación de casos de uso fomenta la comunicación interdisciplinaria, integrando la visión de analistas, diseñadores, programadores y usuarios finales. Este alineamiento contribuye a que el producto final responda efectivamente a las necesidades del negocio y no solo a supuestos técnicos.

Para una mejor comprensión de la estructura y elementos de un caso de uso, se recomienda revisar el [Anexo 2. Plantilla para especificar casos de uso](#), donde se presenta un ejemplo práctico de su aplicación.

Los formatos/plantillas para realizar la especificación de casos de uso varía de acuerdo a quienes están en el desarrollo, se presenta una adaptación de esta plantilla para el desarrollo de esta asignatura.

Aquí se puede notar algunas cosas importantes:

- Se encuentra la interacción entre lo que hará el actor del sistema y el Sistema, esto servirá para que el programador desarrolle la aplicación, así como el tester prueba la solución
- Encontramos objetos para nuestro modelo, atributos de esos objetos y operaciones.
- La secuencia en que se debe realizar el proceso

#### 5.4.3. Realización del Diagrama de casos de uso

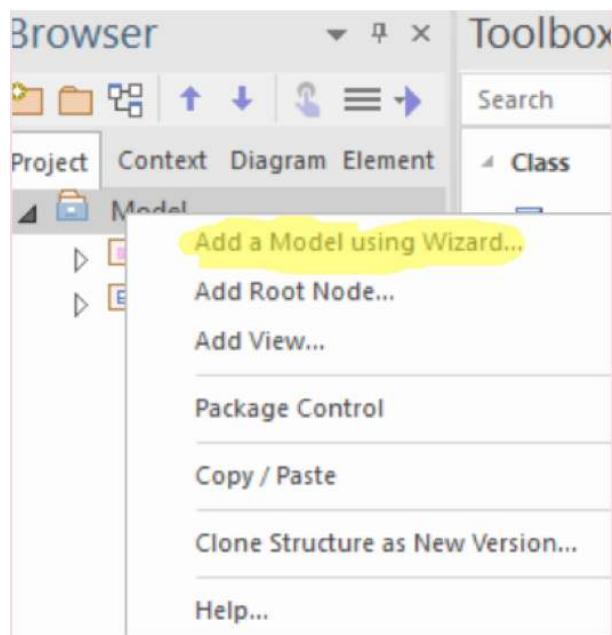
En la herramienta de modelado **ENTERPRISE ARCHITECT** (EA), se presentan varias plantillas para realizar el diagrama, a continuación, se muestra algunas el proceso:

A partir de un nuevo proyecto se creará el modelo, puede crear un nuevo diagrama usando el wizard (asistente) como se muestra en la figura 34.



**Figura 34**

EA-Creación nuevo diagrama

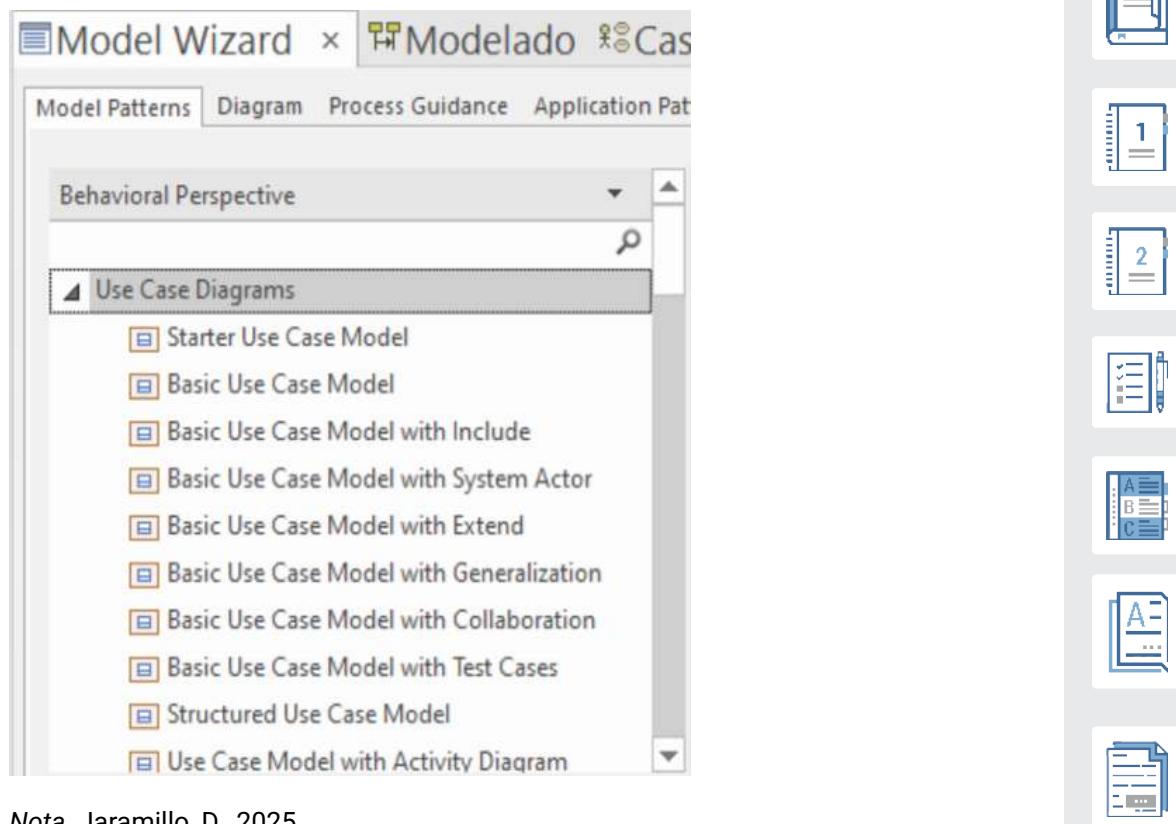


Nota. Jaramillo, D., 2025.

A partir de esto puede seleccionar en el wizard la plantilla que necesite, pues existen varias para realizar el diagrama de casos de uso como se muestra en la figura 35:

**Figura 35**

EA-opciones de realización de Diagrama de Casos de Uso



Nota. Jaramillo, D., 2025.

Cada una de estas plantillas presenta una forma diferente de representar el diagrama como se muestra en las figuras 36 y 37:

**Figura 36**

EA-opcion 1 para realizar DCU

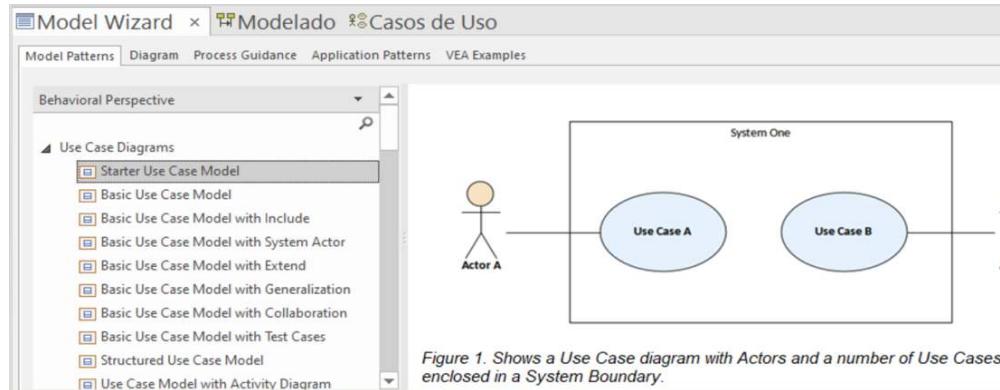


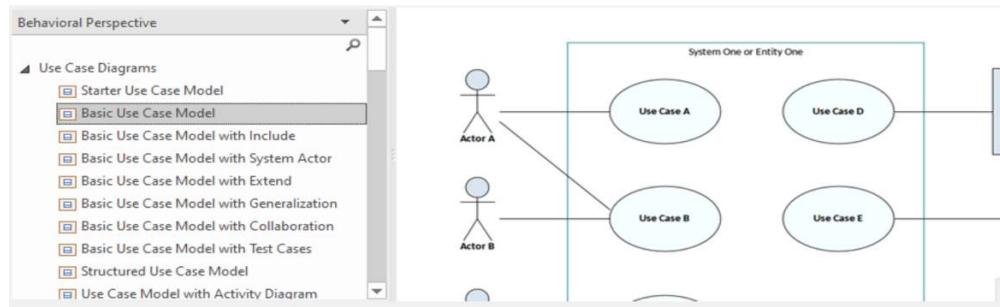
Figure 1. Shows a Use Case diagram with Actors and a number of Use Cases enclosed in a System Boundary.

Nota. Jaramillo, D., 2025.

Otra alternativa se presenta en la siguiente figura:

**Figura 37**

EA-opcion 2 para realizar DCU



Nota. Jaramillo, D., 2025.

Para empezar en este proceso se recomienda realizar con la primera plantilla. Cada herramienta tiene los elementos básicos para realizar los diagramas, en EA podemos encontrar en la figura 38:

**Figura 38**

Toolbox para desarrollar Diagrama de Casos de Uso

▲ Use Case

- Actor
- Use Case
- Boundary

▲ Use Case Relationships

- Use
- Associate
- Generalize
- Include
- Extend

Nota. Jaramillo, D., 2025.

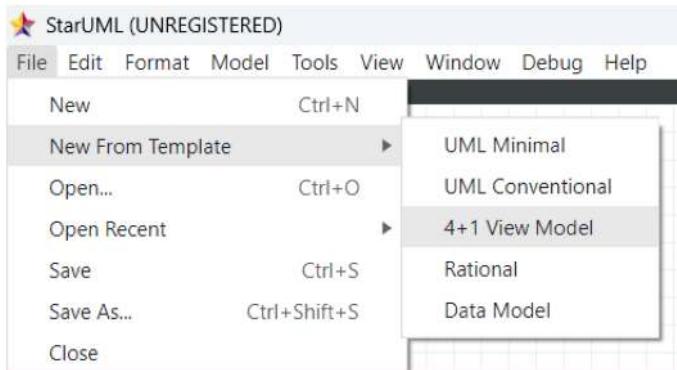
Vamos a mencionar otra herramienta para el modelado STARTUML con la finalidad de comprobar que estas herramientas son muy similares, para los otros diagramas utilizaremos únicamente EA.

En la figura 39 podemos encontrar una plantilla más específica para trabajar ya con el modelo 4+1, que no lo tiene EA (se tiene que crear sub paquetes).



**Figura 39**

STARUML-Creación del proyecto

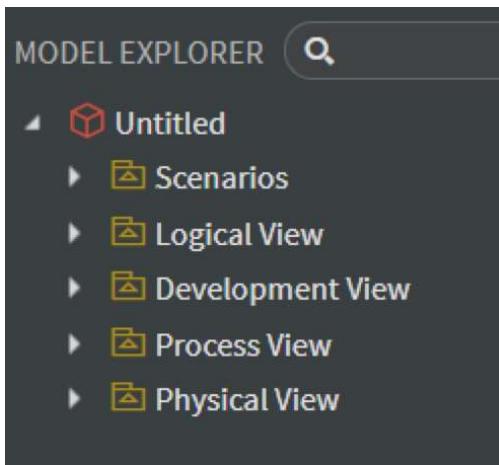


Nota. Jaramillo, D., 2025.

En la figura 40 se muestra cómo se crea el proyecto de trabajo

**Figura 40**

STARUML-Plantilla del proyecto View 4+1

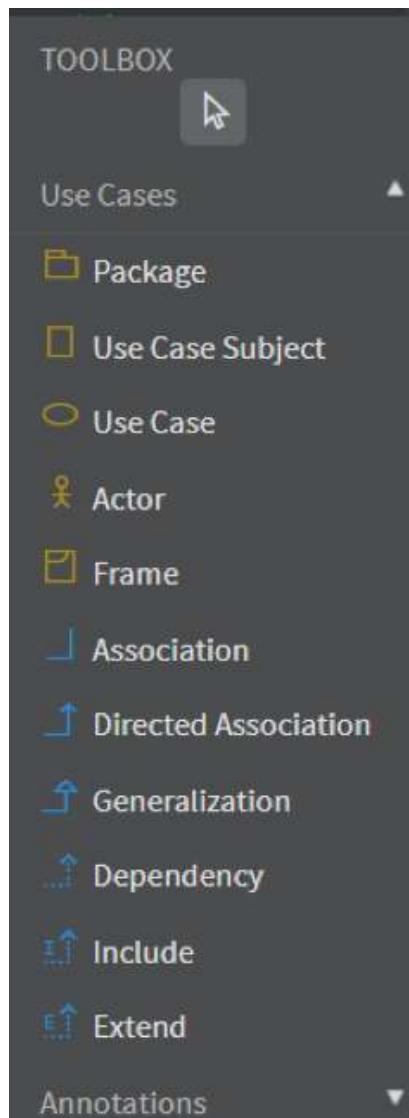


Nota. Jaramillo, D., 2025.

En la vista de escenarios se podrá entonces crear un diagrama de casos de uso y las herramientas que nos presentará son muy similares a las de EA como lo puede ver en la figura 41.

**Figura 41**

STARUML- ToolBox para DCU



Nota. Jaramillo, D., 2025.

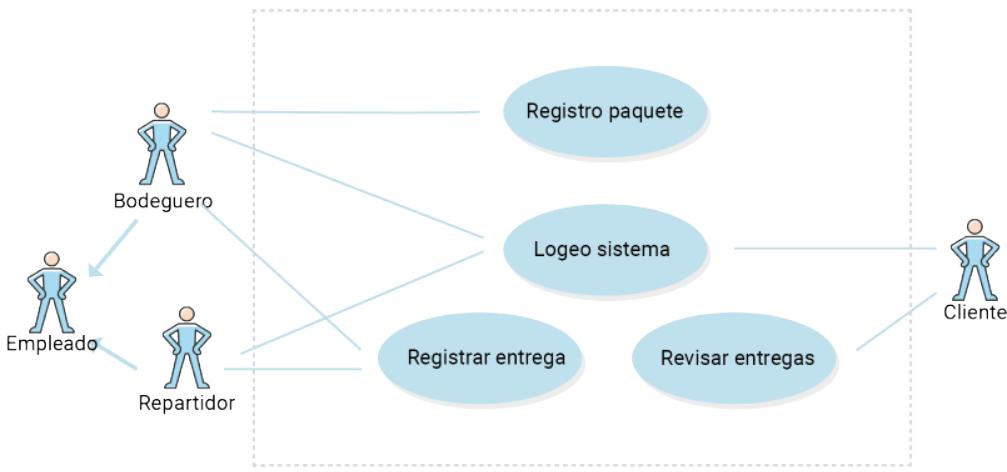
Si comparamos lo que nos presentan las dos herramientas son muy similares así también otras. Ud. Debe de acuerdo a su entorno de trabajo seleccionar la herramienta.

## Ejemplo: TRAMIL.

El caso de estudio lo puede encontrar en el repositorio con el título: <<EmpresaPaquetes>> está en las dos versiones EA y STARUML

**Figura 42**

Diagrama caso de uso - ejemplo TRAMIL



Nota. Jaramillo, D., 2025.

En el repositorio de la asignatura podrá encontrar el proyecto TRAMIL en EA que puede ayudarle sobre el caso de estudio para este ciclo académico. Como ejercicios de práctica se completarán este diagrama en las tutorías de clase.



### Actividades de aprendizaje recomendadas

Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. Instale, configure y familiarice con una herramienta de modelo que tenga acceso. Hay versiones gratuitas, tenga mucho cuidado con aplicaciones que puedan instalar virus en su computadora.
2. Realice un diagrama de casos de uso para que se familiarice con la herramienta, revise a qué hace referencia cada una de las opciones de la herramienta de este diagrama.

3. Consulte cuándo utilizar <<extend>> y cuando <<include>> prepare algo para la tutoría.



## Resultado de aprendizaje 1 a 4:

- Conoce las principales áreas de conocimiento de la ingeniería de software.
- Conoce e identifica las fases del ciclo de vida de desarrollo de sistemas.
- Identifica el ciclo de vida a utilizar en el desarrollo de un proyecto de software.
- Identifica una metodología de desarrollo acorde a las características de un proyecto de software.

### Contenidos, recursos y actividades de aprendizaje recomendadas



#### Semana 8

#### Actividades finales del bimestre

Estimado/a estudiante, la unidad 4 es más compleja y necesaria para el segundo bimestre, por lo que es necesario que ponga más atención y repase los ejercicios sobre requerimientos. Recuerde que se han colocado algunos videos que puede repasar.



#### Actividades de aprendizaje recomendadas

Refuerce su aprendizaje desarrollando las actividades que se describen a continuación:

1. Revise las unidades que se han tratado.
2. Participe en la evaluación presencial del primer bimestre.





## Segundo bimestre



### Resultado de aprendizaje 1 y 2:

- Conoce las principales áreas de conocimiento de la ingeniería de software.
- Conoce e identifica las fases del ciclo de vida de desarrollo de sistemas.

Para alcanzar los resultados de aprendizaje planteados, se abordará el diseño de software desde el análisis estructural y de comportamiento, comprendiendo la organización e interacción de los componentes del sistema. Además, se trabajará la vista de procesos para modelar y gestionar actividades dentro del ciclo de vida, relacionando cada fase con técnicas y representaciones que fortalezcan la construcción de sistemas confiables, eficientes y alineados con los requerimientos planteados.

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.

#### Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



#### Semana 9

El modelo de análisis es un paso intermedio para llegar del modelo de requerimientos al diseño, y del diseño a la implementación, por tanto, en este bimestre vamos a trabajar en realizar este proceso, recuerde que este contenido no está sujeto a ninguna metodología de desarrollo, lo que se estudia aquí debe aplicarse en cualquiera de ellas bien sea con un enfoque predictivo o con un enfoque ágil.

Desde el punto de vista de ingeniería, estamos estudiando el método de construcción de las aplicaciones. La metodología de desarrollo de software organiza las actividades del método con su enfoque particular en función de las restricciones del proyecto.

Para realizar el análisis Orientado a Objetos, se requiere el artefacto de especificación de los casos de uso, y para poder trabajar con el modelo de análisis es necesario contar con esas especificaciones.

Comience realizando una lectura de la Introducción a la unidad 5 de la guía didáctica, se hace una explicación general del tema y una estrategia para el trabajo. Tal como menciona, la guía, puede profundizar en el modelado revisando el texto: [\*The unified modeling language\*](#), que lo puede encontrar en la web.

Adicionalmente, le recomiendo utilizar la siguiente lista de videos para refrescar o aprender temas de modelos con UML.

- [Modelado de casos de uso UPV](#).
- [Modelado de clases con UML de la UCAM](#).

Además, le animo a revisar el libro de Penalvo, F. J. G., Martin, S. B., González, M.A.C. (2008,). Ingeniería del software. Retrieved March 27, 2020, from OCW-USAL [Web site](#).

## **Unidad 5. Diseño de software**

### **5.5. Análisis estructural y de comportamiento**

Dentro del proceso de diseño de software, dos enfoques complementarios resultan esenciales para comprender, organizar y representar un sistema: el análisis estructural y el análisis de comportamiento. Ambos proporcionan perspectivas distintas, pero necesarias, para garantizar que el sistema final cumpla con los requisitos funcionales y no funcionales establecidos.

El análisis estructural se centra en identificar y organizar los componentes estáticos del sistema. Su propósito es definir qué elementos conforman la aplicación, cómo se relacionan entre sí y qué responsabilidades asume cada uno. En términos de modelado, se traduce en la representación de clases, atributos, operaciones y asociaciones. Por ejemplo, un diagrama de clases en UML es una herramienta típica de este enfoque, ya que permite visualizar las entidades principales del dominio del problema y sus interrelaciones. Gracias al análisis estructural, los desarrolladores pueden construir un modelo claro de los datos y objetos del sistema, lo cual facilita la comprensión de la arquitectura y asegura la coherencia en la implementación. Además, este análisis proporciona la base para diseñar la persistencia de datos y definir la estructura de las bases de datos que soportará la aplicación.

Por otro lado, el análisis de comportamiento se orienta a la dinámica del sistema, es decir, a cómo interactúan los elementos definidos en el análisis estructural durante la ejecución. Aquí no solo interesa qué componentes existen, sino también cómo colaboran para cumplir con los casos de uso planteados. Diagramas como los de secuencia, comunicación o actividad en UML permiten modelar la interacción entre actores y objetos, mostrando la secuencia de mensajes, decisiones o flujos de control que conducen a un resultado. Este análisis es fundamental para validar que el sistema responderá de manera correcta frente a estímulos externos, que manejará adecuadamente la concurrencia y que cumplirá con los flujos de trabajo definidos en los requisitos.

La relación entre ambos enfoques es complementaria. Mientras que el análisis estructural ofrece una visión estática y organizacional del sistema, el análisis de comportamiento proporciona la perspectiva dinámica y funcional. Juntos permiten construir un diseño robusto, asegurando que cada componente tenga un propósito claro y que las interacciones entre ellos garanticen la satisfacción de los requisitos del cliente.

## 5.6. Vista procesos

### Diagrama de secuencia

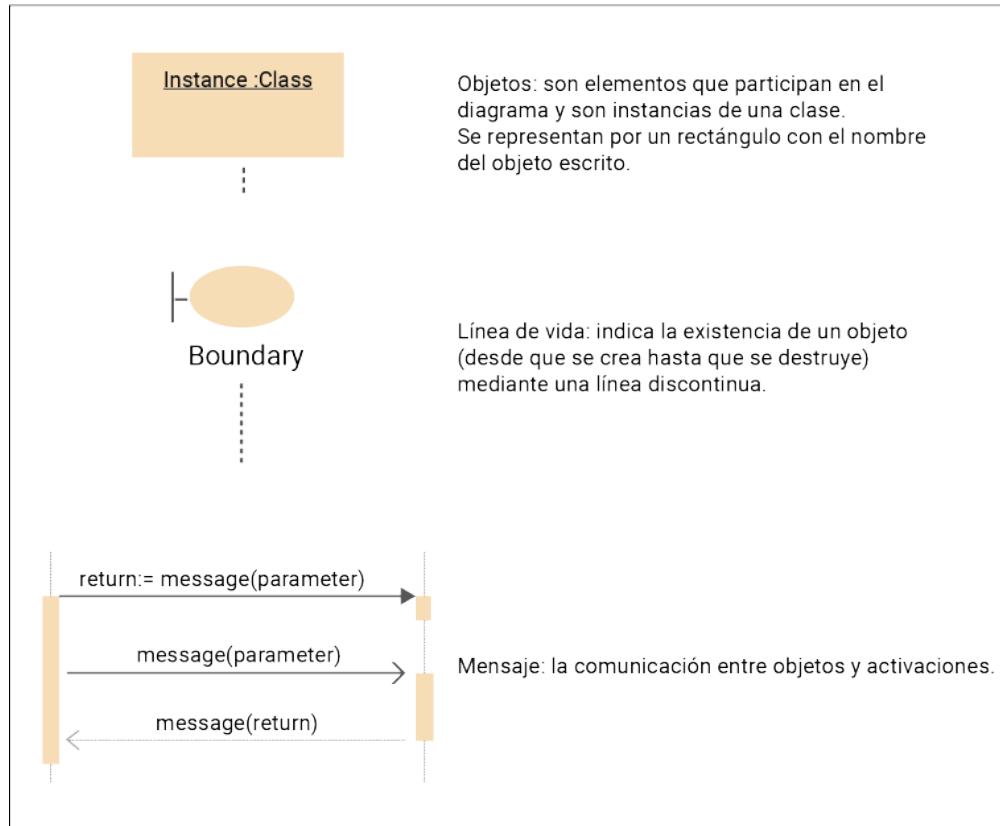
Un diagrama de secuencia es un gráfico bidimensional donde el eje vertical representa el tiempo, el eje horizontal muestra objetos del sistema y la interacción entre los objetos se representa mediante flechas que van de unos objetos a otros, ordenadas cronológicamente de arriba a abajo. Son un tipo de diagramas de interacción, que a su vez es una categoría de diagramas de comportamiento. Los diagramas de secuencia se utilizan cuando queremos expresar qué objetos se relacionan con qué objetos, enfatizando el orden en que lo hacen y qué tipo de mensajes se envían entre sí. También permiten expresar estructuras de control (condiciones y repeticiones), pero los diagramas de secuencia no están pensados para eso y debemos evitar incluir lógica procedural compleja en ellos.

Los elementos que participan en los diagramas de comunicación lo encontramos en la siguiente figura.



**Figura 43**

Elementos que participan en los diagramas de comunicación

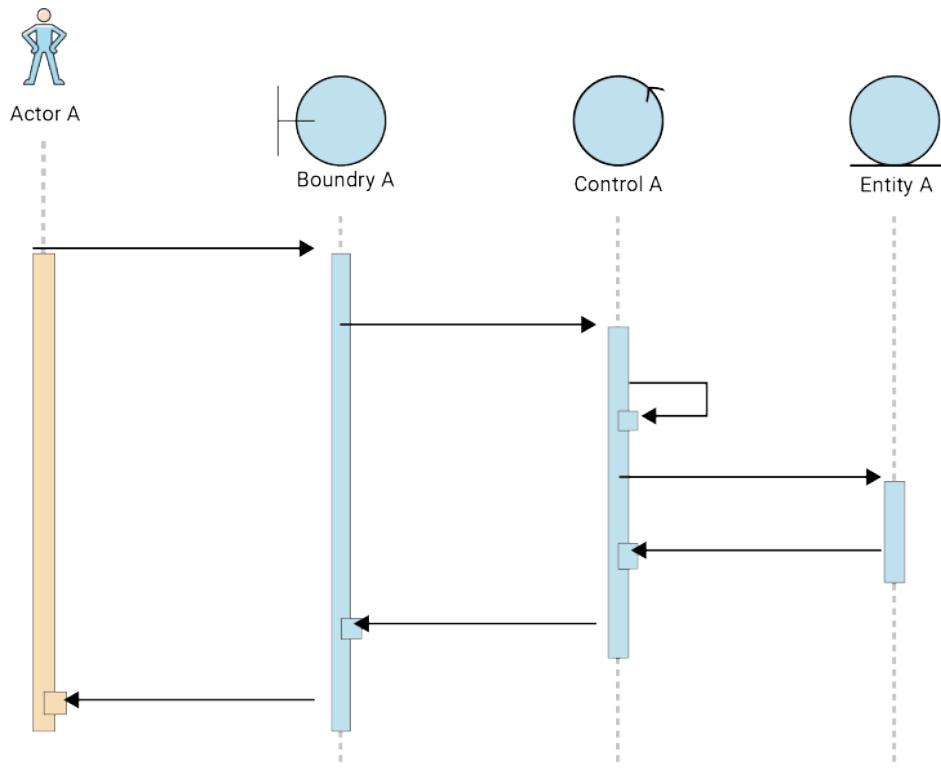


Nota. Jaramillo, D., 2025.

En la herramienta de trabajo EA, nos permite utilizar varias plantillas, desde una forma sencilla de hacerlo como se muestra en la figura 44.

**Figura 44**

Diagrama de Secuencias



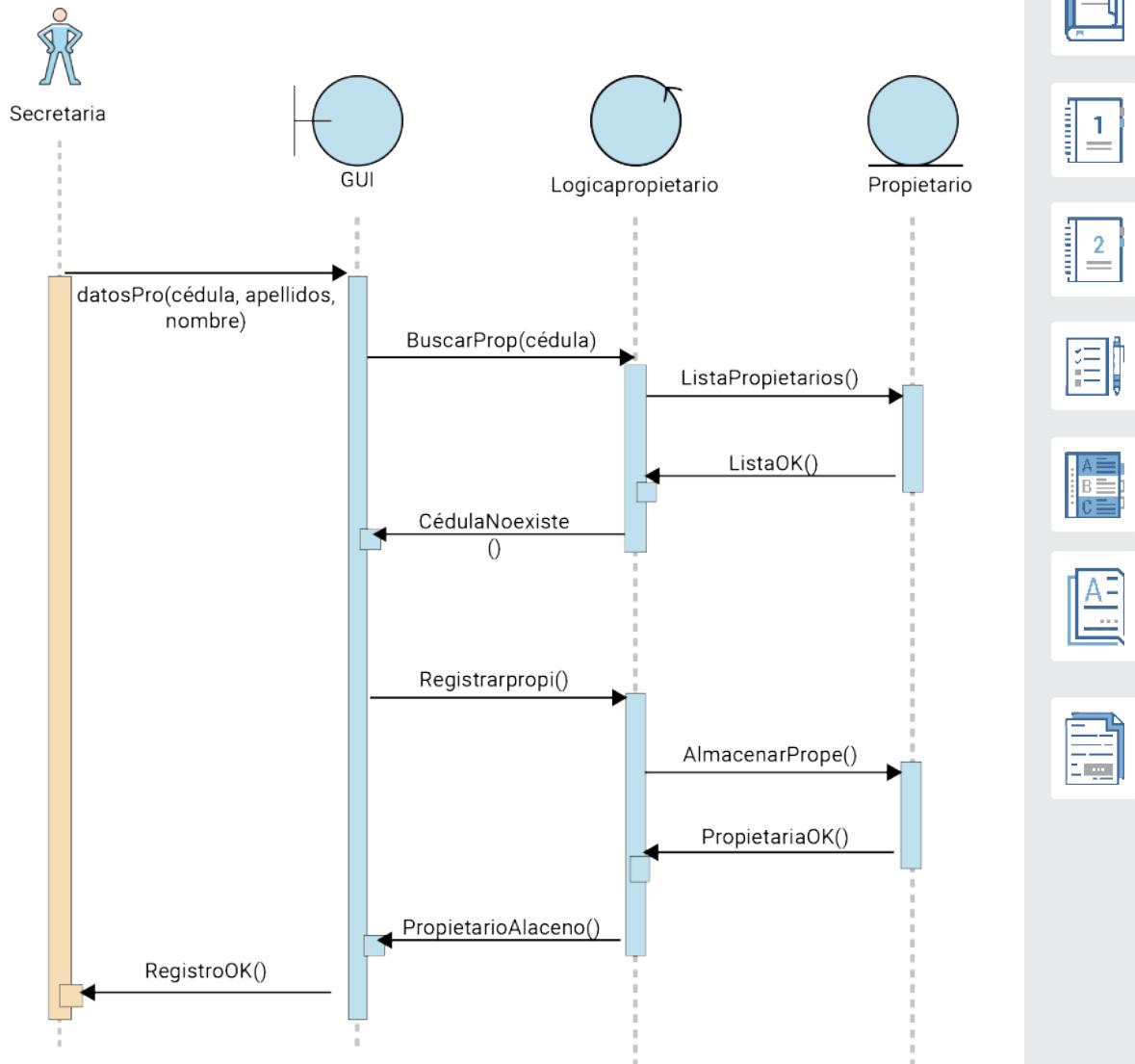
Nota. Jaramillo, D., 2025.

Existen varias formas de realizar el diagrama de secuencias en EA, Revise el recurso ¿Cómo construir un diagrama de secuencia?

Un ejemplo del diagrama de secuencia lo podemos encontrar en la figura 45 para ingresar un propietario

**Figura 45**

Diagrama de secuencia



Nota. Jaramillo, D., 2025.

Como se puede notar tenemos un actor que es el que desencadena la secuencia con unos datos y en este caso tres objetos uno de interfaz, uno de control y otro tipo entidad, ahí se colocara la secuencia de trabajo

En el repositorio de la asignatura encontrará proyectos realizados en EA donde se han desarrollado algunos ejemplos del diagrama de secuencias.



## Actividades de aprendizaje recomendadas

Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. Busque el archivo secuencia.pdf en los materiales y realice los diagramas que se presentan como práctica.
2. Realice en la herramienta seleccionada por lo menos unos dos diagramas de secuencia del caso de estudio que usted está proponiendo para trabajar.
3. Prepare preguntas para la tutoría con el profesor sobre este tema.



## Resultado de aprendizaje 3 y 4:

- Identifica el ciclo de vida a utilizar en el desarrollo de un proyecto de software.
- Identifica una metodología de desarrollo acorde a las características de un proyecto de software.

Para alcanzar los resultados de aprendizaje planteados, se orientará el proceso hacia la identificación del ciclo de vida y la metodología de desarrollo más adecuada para un proyecto de software. Además, se enfocarán en el diseño arquitectónico, comprendiendo las decisiones clave, conceptos fundamentales, patrones arquitectónicos y arquitecturas de aplicación. Este conocimiento les permitirá seleccionar y aplicar el enfoque más efectivo para el desarrollo de software según las características del proyecto.

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.

### Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



### Semana 10

### Unidad 5. Diseño de software

#### 5.7. Tarjetas CRC

Las tarjetas CRC (*Class-Responsibility-Collaboration*), son una herramienta utilizada en el diseño Orientado a Objetos para identificar, organizar y analizar las clases que formarán parte de un sistema. Cada tarjeta representa una

clase y contiene tres secciones: el nombre de la clase, sus responsabilidades (lo que debe hacer) y sus colaboraciones (otras clases con las que interactúa para cumplir sus funciones).

Esta técnica, creada por Ward Cunningham y Kent Beck, facilita el paso de los requisitos funcionales al modelo de clases, permitiendo visualizar de manera sencilla la distribución de responsabilidades en el sistema. Su uso resulta especialmente valioso en las primeras etapas del diseño, cuando se requiere experimentar y refinar la estructura del software antes de llevarla a diagramas UML o a la codificación.

La importancia de las tarjetas CRC dentro del diseño de software radica en varios aspectos. En primer lugar, promueven la colaboración en equipo, ya que pueden elaborarse en dinámicas grupales donde analistas y desarrolladores discuten y asignan responsabilidades. En segundo lugar, fomentan el buen diseño Orientado a Objetos, evitando la concentración de demasiadas funciones en una sola clase y distribuyendo las tareas de manera equilibrada. Finalmente, sirven como un puente entre los casos de uso y los modelos de implementación, al transformar interacciones en responsabilidades concretas para cada clase.

A continuación, se presentan dos ejemplos de Tarjetas CRC para el caso de estudio TRAMIL, organizados en la siguiente tabla:

**Tabla 15***Ejemplos de Tarjetas CRC*

Clase	Responsabilidades	Colaboraciones
Paquete	<ul style="list-style-type: none"><li>Almacenar información del paquete (código, peso, dimensiones, remitente, destinatario).</li><li>Registrar estado del paquete (en bodega, en tránsito, entregado).</li><li>Actualizar la ubicación y movimientos del paquete.</li></ul>	<ul style="list-style-type: none"><li><b>Cliente:</b> consulta el estado y seguimiento de su paquete.</li><li><b>EncargadoBodega:</b> registra la llegada y entrega en oficina.</li><li><b>Repartidor:</b> confirma entregas a domicilio mediante dispositivo móvil.</li><li><b>SistemaTracking:</b> actualiza estados en la base de datos.</li></ul>
Cliente	<ul style="list-style-type: none"><li>Registrarse en el sistema proporcionando datos personales y dirección.</li><li>Consultar el estado y seguimiento de sus paquetes.</li><li>Confirmar recepción de paquetes entregados en domicilio o en bodega.</li></ul>	<ul style="list-style-type: none"><li><b>Paquete:</b> consulta movimientos y estados de sus envíos.</li><li><b>EncargadoRegistro:</b> valida y autoriza el alta del cliente en el sistema.</li><li><b>SistemaNotificaciones:</b> recibe confirmación vía correo electrónico sobre el alta y cambios de estado de sus paquetes.</li></ul>

Nota. Jaramillo, D., 2025.

## 5.8. Vista lógica: Diagrama de clases

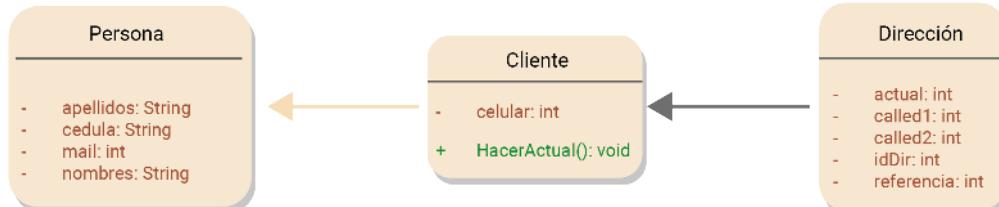
Los diagramas de clases son un tipo de diagramas estructurales. Los diagramas de clases están compuestos por las clases que componen el sistema, su estructura interna y sus relaciones con otras clases.

### Clases y objetos

Una clase es una descripción de un conjunto de objetos con las mismas propiedades (atributos) y el mismo comportamiento (operaciones). Tanto en programación orientada a objetos (POO) como en UML, una clase es una plantilla que representa un concepto del mundo real y se utiliza para crear objetos. Por ejemplo, podemos abstraer todos los productos de un supermercado en una clase **producto** junto con sus atributos (código, nombre, precio) y sus operaciones (obtener código, obtener precio, registrar un nuevo producto). A partir de una clase se pueden crear objetos, también llamados instancias de la clase en POO. Los objetos son concreciones de una clase: todos los objetos de una clase comparten las mismas operaciones, pero sus atributos pueden tener valores distintos. Las clases en UML se representan con un rectángulo dividido en 3 partes o “cajas”: la caja superior contiene el nombre de la clase, la del medio los atributos, y la inferior las operaciones. La figura 46 muestra un pequeño ejemplo de clases relacionadas mediante herencia y composición.

**Figura 46**

Clases, atributos y operaciones



Nota. Jaramillo, D., 2025.

### Relaciones entre clases

Las relaciones pueden ser:

- Asociación.
- Agregación.
- Composición.
- Dependencia.

También puede ocurrir que varias clases tengan operaciones o atributos en común y se requiera abstraerlos utilizando mecanismos conocidos de los lenguajes OO como la herencia e interfaces. UML también permite representar estas relaciones de clases mediante:

- Generalización (equivalente a la herencia en POO).
- Realización (equivalente a interfaces en Java).

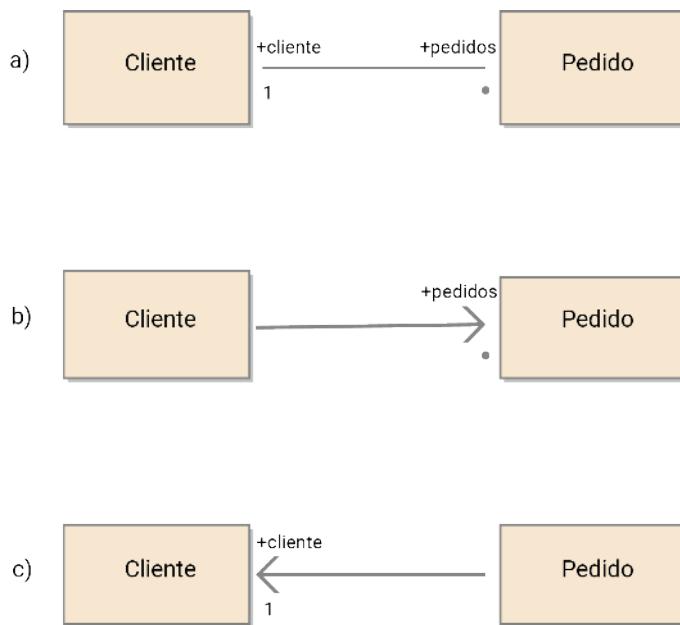
### Asociación:

Se representa mediante una línea continua, que puede estar o no acabada en una flecha en "V", según la navegabilidad. Además, se suelen indicar los roles y la multiplicidad. Si la flecha apunta de la clase A a la clase B, se lee como: "ClaseA tiene una ClaseB" y se dice que la asociación o navegabilidad es unidireccional. Si no dibuja la flecha se lee "Clase A tiene una ClaseB y ClaseB tiene una ClaseA" y se dice que la asociación o navegabilidad es bidireccional.

Supongamos una aplicación de gestión con las clases cliente y pedido, tal como se indica en la figura 47, cliente guarda información sobre los pedidos y pedido tiene información sobre el cliente que lo realizó. La navegabilidad es, por tanto, bidireccional (47.a). Puede ser que la aplicación guarde la lista de pedidos en el objeto cliente, pero no a la inversa. En este caso la navegabilidad de la asociación sería unidireccional y su representación en UML sería como se indica en 47.b. Para el caso que los pedidos contienen información del cliente, pero los clientes no guardan pedidos, la asociación sería unidireccional, tal como se indica en 47.c.

**Figura 47**

Representación de asociación en el diagrama de clases.



Nota. Jaramillo, D., 2025.

### Agregación y composición

Según (Bruegge & Dutoit, 2018) la agregación y la composición son “asociaciones que representan una relación entre un todo y sus partes”. Se puede leer como “ClaseB es un componente de ClaseA”, o también “ClaseA está compuesto por ClaseB”. Las diferencias entre ellas son: en la agregación, los objetos “parte” pueden seguir existiendo independientemente del objeto “todo”. Mientras que en la composición, la vida de los objetos compuestos está ligada a la del objeto que los compone, de manera que si el “todo” se destruye, las “partes” también se destruyen. La agregación se representa uniendo las dos clases (todo/parte) con una línea continua y poniendo un rombo hueco en la clase “todo”. La composición es igual, pero con el rombo relleno, como se muestra en la figura 48.

**Figura 48**

Ejemplo de agregación y composición



Nota. Jaramillo, D., 2025.

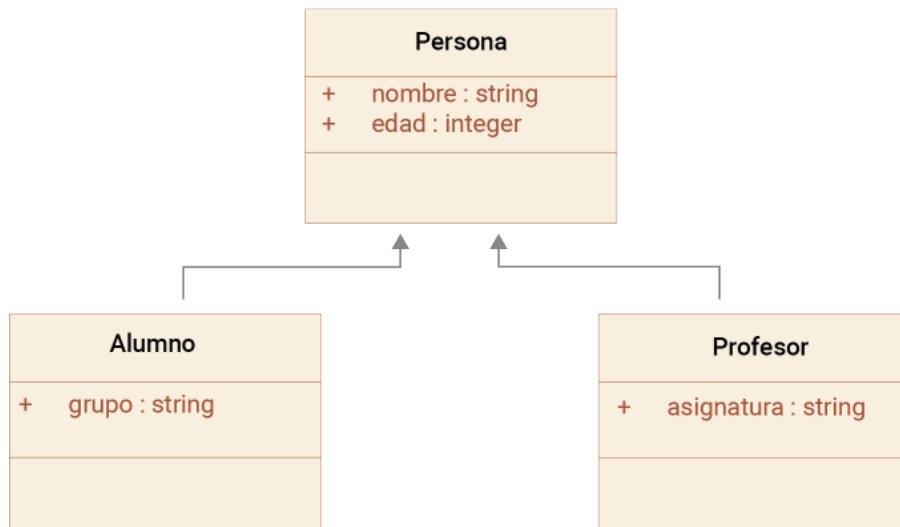
Tal como se muestra en la figura 48, la clase persona tiene una relación de agregación con la clase categoría, esto quiere decir que “una persona puede tener una categoría y que una categoría puede estar presente en muchas personas”. En este caso las categorías pueden existir por sí mismas. En la siguiente relación la clase persona tiene una relación de composición con la clase Domicilio, ya que domicilio es una parte inseparable de la persona, por lo que si no existiera una persona entonces el domicilio de esta debería desaparecer. Esta relación de composición es una relación fuerte, ya que una instancia arrastra a la otra en caso de eliminación.

### Generalización

La relación de generalización entre dos clases indica que una de las clases (la subclase) es una especialización de la otra (la superclase). Si ClaseA es la superclase y ClaseB la subclase, la generalización se podría leer como “ClaseB es una ClaseA” o “ClaseB es un tipo de ClaseA”. En Java (y la mayoría de los lenguajes orientados a objetos), la generalización se conoce como herencia. La herencia se utiliza para abstraer en una superclase los métodos y/o atributos comunes a varias subclases. A la subclase también se le puede llamar clase hija o clase derivada. A la superclase también se le puede llamar clase padre o clase base. La generalización se expresa con una línea acabada en una flecha triangular hueca, dibujada desde la clase hija hacia la clase padre. Por ejemplo, en la figura 49, la generalización (herencia) alumno y profesor son especializaciones de la clase persona.

**Figura 49**

Ejemplo de Generalización (Herencia).



Nota. Jaramillo, D., 2025.

En Java, una clase padre puede tener métodos abstractos, lo que significa que para esos métodos no se proporciona ninguna implementación. Una clase abstracta es una clase que tiene al menos un método abstracto, al tener uno o más métodos sin implementación, las clases abstractas no se pueden instanciar. Una clase que extiende a una clase abstracta debe implementar los métodos abstractos o bien volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta. En UML, tanto las operaciones como las clases abstractas se expresan poniendo el nombre de la operación o la clase en cursiva.

En la herramienta EA vamos a poder utilizar diferentes plantillas a través del asistente, a continuación, en las figuras 50 y 51 mostramos dos de las mismas:

## Figura 50

Diagrama de clases, EA-Ejemplo1

The screenshot shows the EA interface with the 'Structural Perspective' selected. In the left pane, under 'Class Diagrams', 'Starter Class Diagram' is highlighted. The main pane displays the 'Starter Class Diagram' pattern, which creates a simple Class diagram with two classes, 'Class A' and 'Class B', connected by an association named 'Association A'. Below the diagram, a caption reads: 'Figure 1. Shows a Class diagram with two classes connected by an Association relationships. The Association is named and an indicator shows how to read the relationship between the two classes.'

Nota. Jaramillo, D., 2025.

Otra plantilla que se puede seleccionar se muestra en la figura 51.

## Figura 51

Diagrama de Clases, EA-Ejemplo2

The screenshot shows the EA interface with the 'Structural Perspective' selected. In the left pane, under 'Class Diagrams', 'Basic Class Diagram with Attributes and Operations' is highlighted. The main pane displays the 'Basic Class Diagram with Attributes and Operations' pattern, which creates a Class diagram with two classes, 'Class A' and 'Class B', connected by an association named 'Association A'. Both classes have attributes and operations listed. Below the diagram, a caption reads: 'The Basic Class Diagram with Attributes and Operations pattern creates elements and a Class diagram that describes how two Classes are related to each other. The associations show semantic or structural relationships between the classes. Attributes have been added to the Classes which are features of the Class. Along with operations they give the classifier its essential characteristics. When an instance of a class is created the attributes are assigned actual values that identify the object amongst other instances.'

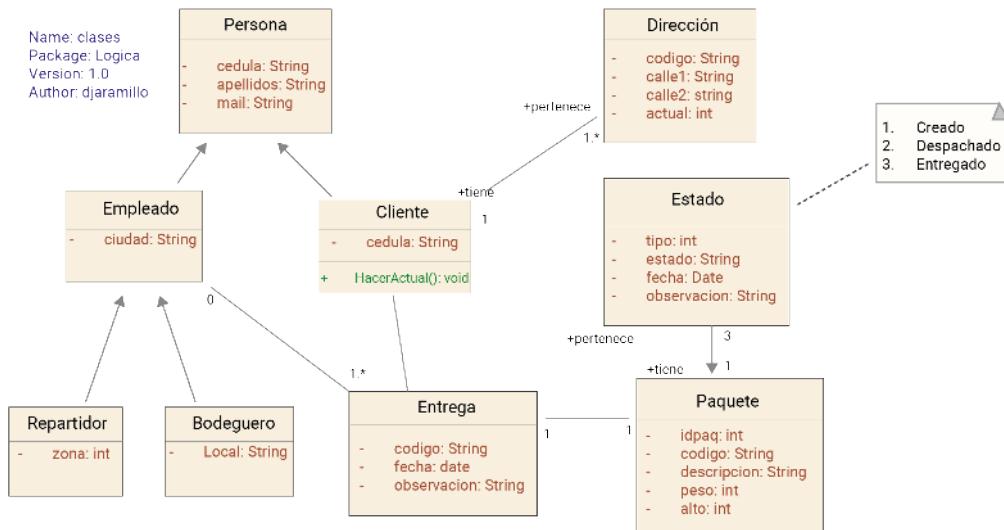
Nota. Jaramillo, D., 2025.

Para desarrollar el diagrama de clases, es necesario realizar lo siguiente:

- Identificar las clases.
- Identificar las relaciones.
- Elaborar el diagrama de clases utilizando herramienta

**Figura 52**

Diagrama de clases TRAMIL



Nota. Jaramillo, D., 2025.

Es esta aproximación inicial de la solución del caso de estudio la figura 52, pues no se muestra la dirección de la asociación, así como roles y multiplicidad.

En el proyecto presentado encontrará el diagrama completo reviselo.



### Actividades de aprendizaje recomendadas

Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. Revise el tema de roles y multiplicidad en los diagramas de clases para una mejor comprensión del tema.
2. Familiarícese con la herramienta de modelado que haya instalado, realice el diagrama de clases en la herramienta STARTUML del diagrama de clases que está presente en la figura 52.
3. Compruebe cuánto ha avanzado; la siguiente autoevaluación ayudará a repasar y asegurar lo visto en esta unidad.



## Autoevaluación 5

1. En el diseño Orientado a Objetos con UML, ¿qué diagrama se utiliza para representar el mapeo de subsistemas a plataformas de *hardware* y *software*?
  - a. Diagrama de clases.
  - b. Diagrama de casos de uso.
  - c. Diagrama de despliegue.
  - d. Diagrama de secuencia.
  
2. ¿Cuál de los siguientes aspectos no corresponde a una política de control de acceso en el diseño de *software*?
  - a. Quién accede a qué información.
  - b. Si los controles de acceso pueden cambiar dinámicamente.
  - c. La velocidad de procesamiento de los datos.
  - d. Cómo se implementa el control de acceso.
  
3. En el contexto de persistencia de datos, ¿qué elemento se emplea comúnmente para garantizar confiabilidad y rapidez en el acceso a la información?
  - a. Algoritmos de ordenamiento.
  - b. Sistema de Gestión de Base de Datos (DBMS).
  - c. Diagramas de flujo de datos.
  - d. Memoria RAM.
  
4. El diagrama de contexto representa:
  - a. Los algoritmos internos del sistema.
  - b. Un solo proceso que simboliza el sistema y sus interacciones con actores externos.
  - c. El detalle de cada módulo del *software*.
  - d. La arquitectura física del sistema.



5. ¿Cuál es el propósito principal del diagrama de contexto en el análisis de sistemas?

- a. Definir la arquitectura física del software.
- b. Representar el flujo interno de los procesos.
- c. Delimitar el alcance del sistema y sus interacciones externas.
- d. Diseñar la base de datos del sistema.

6. Según el modelo 4+1 de Kruchten, ¿qué vista sirve como vínculo entre las demás vistas?

- a. Vista lógica.
- b. Vista de procesos.
- c. Vista física.
- d. Vista de escenarios.

7. En los diagramas de casos de uso, ¿qué representa un “actor”?

- a. Una clase del sistema.
- b. Una entidad externa que interactúa con el sistema.
- c. Un módulo interno de software.
- d. Una base de datos relacional.

8. En el caso de estudio TRAMIL, ¿qué actor es responsable de registrar un paquete en el sistema?

- a. Cliente.
- b. Administrador.
- c. Bodeguero.
- d. Repartidor.

9. En la especificación de casos de uso, ¿qué elementos se deben documentar para evitar ambigüedades?

- a. Solo el flujo normal.
- b. El flujo normal, flujos alternos, precondiciones y postcondiciones.
- c. Exclusivamente los actores.



- d. Únicamente las reglas de negocio.
10. ¿Cuál es una ventaja clave de la especificación de casos de uso en el desarrollo de software?
- a. Permite generar directamente el código fuente.
  - b. Sustituye los diagramas UML.
  - c. Reduce ambigüedades y facilita la comunicación entre clientes y desarrolladores.
  - d. Elimina la necesidad de pruebas.

[Ir al solucionario](#)



## Contenidos, recursos y actividades de aprendizaje recomendadas



### Semana 11

El diseño del sistema es el modelo más importante para la construcción de aplicaciones, para construirlo se debe partir del modelo de análisis y luego seleccionar un estilo arquitectónico que permita satisfacer los requerimientos no funcionales, en esta unidad se revisarán algunos conceptos de arquitectura de software y se revisarán los principales patrones arquitectónicos y las arquitecturas.

### Unidad 6. Diseño arquitectónico

Al diseño del sistema se lo puede definir como la transformación de un modelo de análisis en un modelo de diseño, el cual incluye la descomposición en subsistemas y una descripción clara de las estrategias de construcción.

Cada uno de los aspectos que implica el diseño del sistema, que entre otras cosas debe incluir cuestiones como la identificación de metas de diseño, las cuales están relacionadas con elementos de calidad, descomposición inicial

del sistema en subsistemas más manejables que reducen la complejidad, y finalmente refinar la descomposición del sistema hasta que se cumplan las metas esperadas.

El proceso de diseño según (Bruegge & Dutoit, 2018), se alimenta del proceso de análisis y utiliza como insumos lo siguiente: conjunto de requerimientos no funcionales y restricciones, el modelo de casos de uso que describe las funcionalidades desde el punto de vista del actor, el modelo de objetos y los diagramas de secuencia, que además sirven como base para la comunicación entre el cliente y el equipo de desarrollo. Este modelo carece de información respecto de la estructura interna del sistema, y tampoco muestra información sobre la configuración del *hardware* y, por lo tanto, no se utiliza para la implementación.

El proceso de diseño, por tanto, es el primer paso para definir la estructura del sistema y deriva finalmente en modelos que sirven para la implementación. Para ello, la fase de diseño necesita de los siguientes insumos:

- Objetivos del diseño, que describen aspectos relacionados con la calidad del sistema que los desarrolladores deben optimizar.
- Arquitectura del *software*, que describe la descomposición en términos de responsabilidades de los subsistemas, dependencias entre subsistemas, mapeo de subsistemas a *hardware*, y las principales decisiones políticas, tales como flujos de control, control de accesos, y almacenamiento de información.
- Casos de uso de interfaz, que describen la configuración del sistema, inicio, apagado y manejo de excepciones.

## 6.1. Decisiones en el diseño arquitectónico

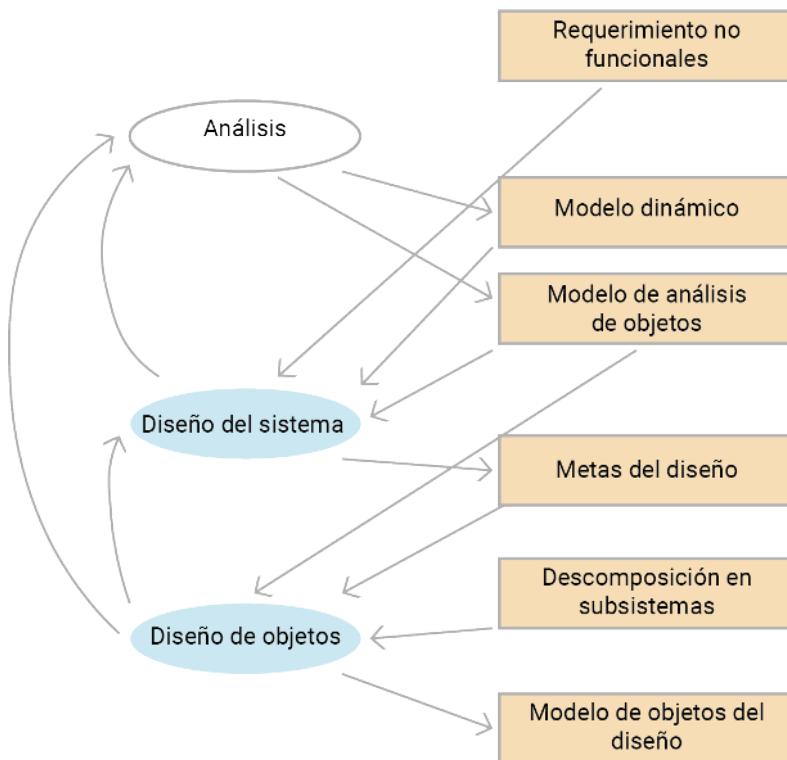
Como se había mencionado anteriormente, las decisiones de diseño arquitectónico se basan en los criterios de calidad y requerimientos no funcionales del sistema. De cómo se utilicen estos requerimientos depende en gran medida la calidad del sistema.

Estas decisiones derivan de los requerimientos no funcionales, y además guían las decisiones a ser tomadas por los desarrolladores cuando se necesita seleccionar de entre varias características a las prioritarias.

En la figura 53, se muestra la relación entre el diseño del sistema con otras actividades de la ingeniería del software.

**Figura 53**

Resumen de actividades de diseño.



Nota. Adaptado de *Object-oriented software engineering : using UML, patterns, and Java* [Ilustración], por Bruegge, B. y Dutoit, A., 2018, Prentice Hall, CC BY 4.0.

Las decisiones de diseño también tendrán impacto en el rendimiento, en la seguridad, en la mantenibilidad y otros atributos de calidad, de hecho, los atributos de calidad son clave al momento de tomar este tipo de decisiones.

Vale tener en cuenta los estándares de calidad del software como producto, en este caso un estándar ISO es la norma ISO9126.

## 6.2. Conceptos de diseño y arquitectura del sistema

Para abordar el tema del diseño, es importante primero entender algunos conceptos, por ello se recomienda poner especial cuidado en el estudio de estos temas y de ser posible profundizarlos utilizando otros recursos.

El diseño de software es una actividad compleja y combina varios elementos para poder obtener modelos de componentes que satisfagan las necesidades del cliente con los niveles de calidad requeridos, por tanto, antes de comenzar con la especificación de los procesos de diseño, es necesario estudiar los conceptos más importantes.

## 6.3. Vista de Desarrollo componentes:

Al igual que en la industria de la construcción, el diseño se puede apreciar desde diferentes puntos de vista, por ejemplo, la perspectiva, es una vista que muestra cómo se verá el edificio una vez terminado, la vista de planta muestra la distribución de ambientes, la vista de instalaciones sanitarias, la vista de seguridades, instalaciones eléctricas, etc, en conjunto constituyen el modelo completo que permitirá a los constructores visualizar todo lo que compone el edificio.

En el software, la situación es un poco más compleja debido a que los componentes son intangibles y por tanto necesitamos capacidad de abstracción para identificar los elementos necesarios en cada aspecto, se requiere además dominio del lenguaje de modelado UML, para poder describir cada uno de los aspectos que implica el software.

La arquitectura de un sistema a decir de Krutchen, se puede evidenciar en el modelo 4+1, y se describen en el texto con claridad, lo que sí se considera importante recalcar que, si bien se necesita un lenguaje para especificar la arquitectura, este no es tan importante hasta que ya se necesita describirla y

para eso se utilizan lenguajes de modelado como UML, y lenguajes de descripción arquitectónica ADL (Architecture Description Language), como AADL (Architecture Analysis & Design Language).



En este punto, es importante mencionar que el diseño y la arquitectura son esenciales para cualquier producto de software, independientemente del tipo de metodología que se use; no obstante, el nivel de especificación depende más que de la metodología de la naturaleza, complejidad y las regulaciones que norman el desarrollo del proyecto.

## Subsistemas

Para reducir la complejidad de un sistema de software, en tiempo de diseño es necesario descomponer el sistema en partes más pequeñas, a estas partes las denominamos subsistemas, diremos por tanto que “un subsistema es una parte reemplazable del sistema con interfaces bien definidas que encapsula el estado y el comportamiento de sus clases” ( Bruegge y Dutoit, 2010, p.218).

La descomposición en subsistemas corresponde a un proceso de descomposición lógico, no obstante, para dimensionar el trabajo, se puede decir que un subsistema corresponde a una cantidad de trabajo que puede asumir un desarrollador o un equipo de ellos puede asumir.

En un sistema orientado a objetos, un subsistema es una parte independiente del sistema que contiene un conjunto de clases relacionadas, y por tratarse de un sistema cada una de estas partes se encuentra relacionada con al menos una de las demás, no se puede concebir una parte aislada.

## Servicios e interfaces de subsistemas

Un servicio, es un conjunto de operaciones relacionadas que comparten un propósito, este conjunto de operaciones que comparte un subsistema con otros subsistemas se muestra como una interface de subsistemas, esta interface especifica el nombre de las operaciones, los parámetros con sus tipos de datos y sus valores de retorno.

En el diseño orientado a objetos, estas interfaces se implementan en la denominada Application Programmer Interface (API). En UML, estas interfaces se representan con conectores de ensamblaje (lollipop), como se muestra en la figura 27.

### Organización de subsistemas por capas

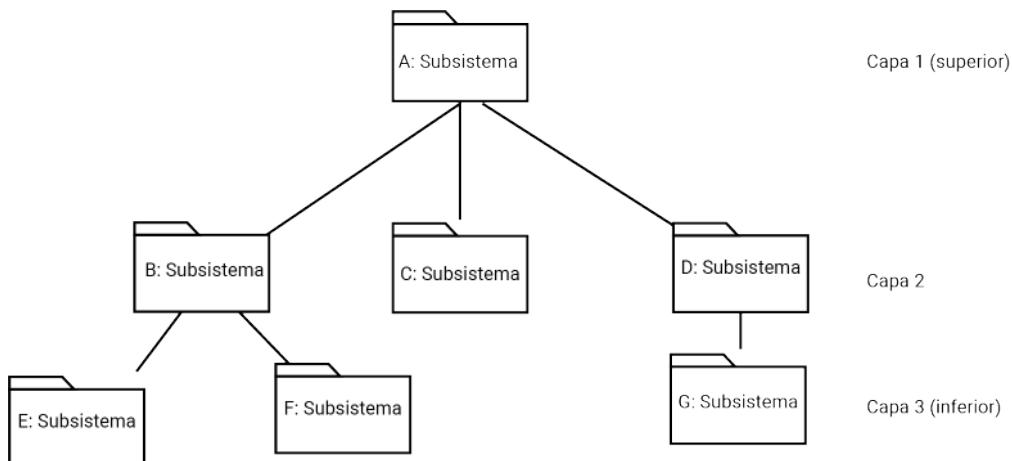
Las capas lógicas (layers), corresponden a la distribución jerárquica de los sistemas, que permite la organización de los subsistemas agrupados de modo que provean servicios similares.

Las capas se ordenan de modo que cada una depende solo de la capa en el nivel inferior, y no conoce de la existencia de otra capa.

Existen dos formas de organizar estas capas, para el primer caso hablaremos de una arquitectura cerrada, en la cual cada capa conoce únicamente a la capa inferior, y en una arquitectura abierta, una capa también puede acceder a niveles más profundos.

**Figura 54**

*Organización de los subsistemas por capas*



Nota. Jaramillo, D., 2025.

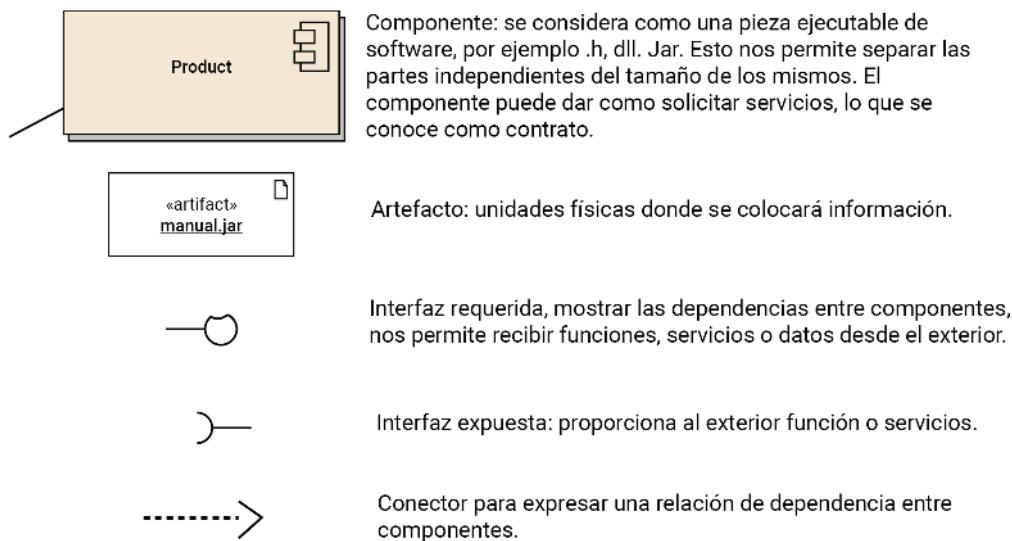
### 6.3.1. Diagrama de componentes

Mediante este diagrama se expresan como las piezas del software que forman un sistema. Este diagrama presenta un nivel de abstracción inferior que el diagrama de clases.

Los elementos que intervienen en los diagramas de comunicación se muestran en la siguiente figura.

**Figura 55**

*Elementos en los diagramas de comunicación*



Nota. Jaramillo, D., 2025.

En EA podemos encontrar varias plantillas para poder realizar el diagrama, como se muestra a continuación:

**Figura 56**

*Tipos de Diagramas de componentes en EA*

The screenshot shows the EA interface with the 'Diagram' tab selected. The left sidebar under 'Structural Perspective' has 'Component Diagrams' expanded, with 'Starter Component Diagram' highlighted. The main area is titled 'Starter Component Diagram' and contains the following text:  
The *Starter Component Diagram* pattern creates Components and a Component diagram that show connector indicating that the two Components share information via interfaces. The Components have these have been made visible on the diagram.

Below this is a diagram showing two components, 'Component A' and 'Component B', connected by an assembly connector. Each component has a note box: 'Notes that describe the Component.' for Component A and 'Notes that describe the Component.' for Component B.

Figure 1. Shows a Component diagram with two Components connected by an Assembly Connector.

**Discussion**

Nota. Jaramillo, D., 2025.

La figura nos presenta otra forma de realizar el diagrama en la herramienta.

**Figura 57**

*Diagrama de componentes-EA-2*

The screenshot shows the EA interface with the 'Diagram' tab selected. The left sidebar under 'Structural Perspective' has 'Component Diagrams' expanded, with 'Component Interfaces with XML Payload' highlighted. The main area is titled 'Component Interfaces with XML Payload' and contains the following text:  
The *Component Interfaces with XML Payload* pattern describes how two Components, representing logical parts of a system, interact via interfaces. The Information Flow allows the payload to be modeled and specified as one or more information items that are exchanged between the components.

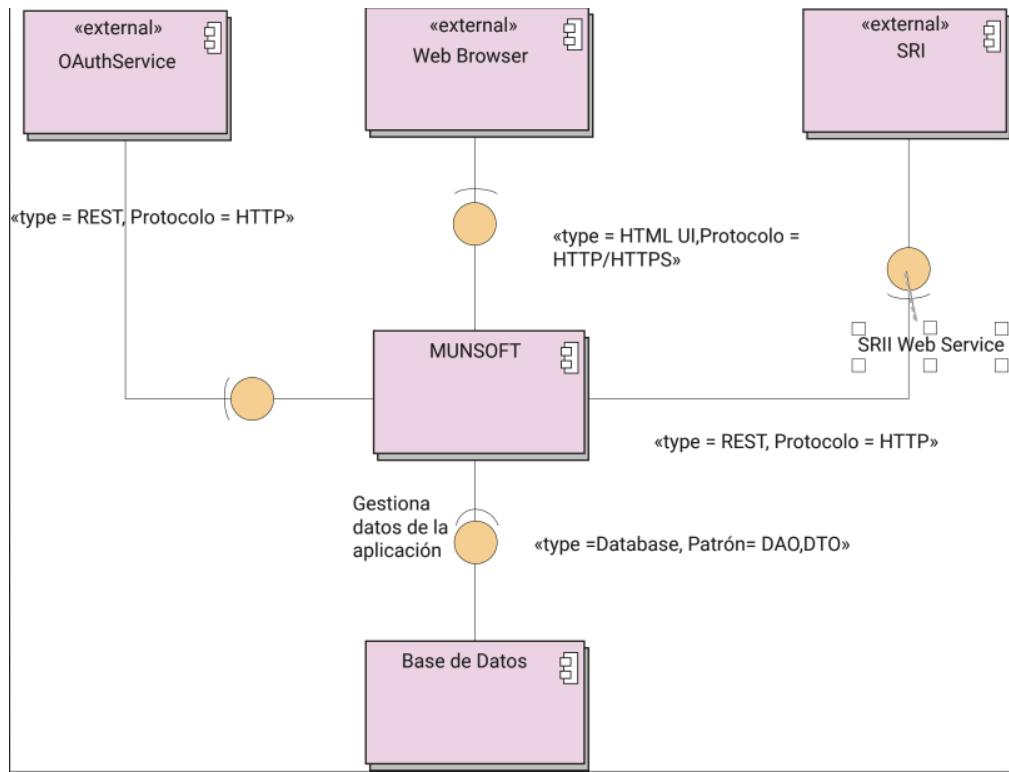
Below this is a diagram showing two components, 'Component A' and 'Component B', connected by an information flow. Component A has a port labeled 'Port a' and an interface labeled 'Interface Two'. Component B also has a port labeled 'Port a' and an interface labeled 'Interface One'. A dashed arrow labeled 'Element One' with the text '<xs:topLevelElement>' above it connects the two interfaces. A legend on the right defines symbols: a light blue square for 'Component', a teal square for 'Port', a yellow square for 'ProvidedInterface', a purple square for 'RequiredInterface', and a grey line for 'InformationFlow'.

Nota. Jaramillo, D., 2025.

A continuación, un diagrama de componentes que se presenta como ejemplo.

**Figura 58**

Ejemplo de diagrama de componentes



Nota. Jaramillo, D., 2025.

Puede revisar la siguiente página para una mejor comprensión: [Diagrama de Componentes UML](#). Asimismo, el siguiente video indica cómo realizar el [diagrama de Componentes](#) y en el repositorio podrá encontrar dentro del proyecto de caso de estudio del ciclo académico un ejemplo más detallado del mismo.

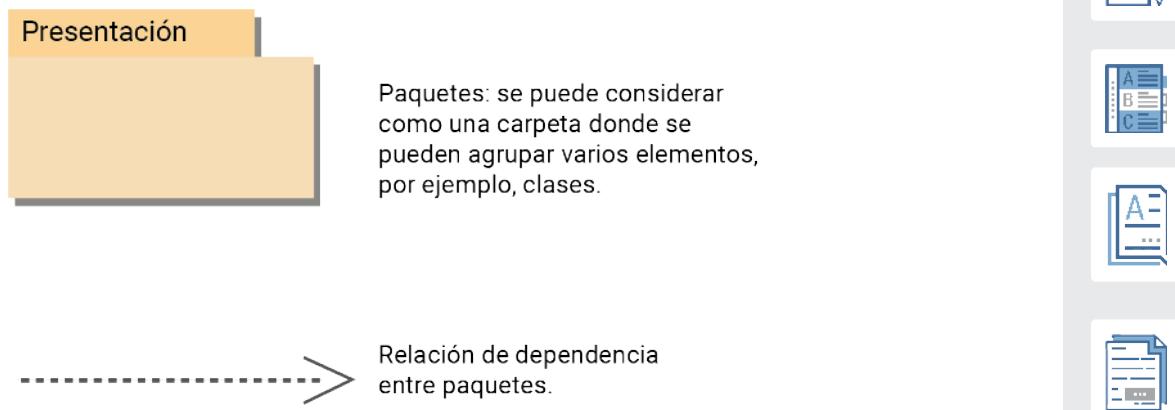
### 6.3.2. Paquetes

El diagrama de paquetes nos permite organizar de mejor manera. Cuando tenemos varias clases podemos ir organizándolas en paquetes, por ejemplo, un paquete de clases puras, un paquete donde colocamos las clases de interfaz y aquellas que las podemos considerar como generales, dependerá mucho cómo se quiera organizar los elementos que tengamos.

En el siguiente diagrama podemos encontrar:

**Figura 59**

*Elementos del diagrama de paquetes*



Nota. Jaramillo, D., 2025.

En la herramienta que estás utilizando podemos realizar este diagrama de varias formas, como se muestra en la parte izquierda de la figura, ahí se pueden seleccionar en este caso 5 diferentes opciones. A continuación, mostramos dos de ellas, una de la forma normal como se realiza y otra como un árbol jerárquico.

**Figura 60**

*Diagrama de paquetes EA-1*

Model Patterns Diagram Process Guidance Application Patterns VEA Examples

Structural Perspective

Package Diagrams

- Starter Package Diagram
- One Level Package Hierarchy
- Nested Package Hierarchy
- Package Dependencies
- Package Imports

Class Diagrams

Object Diagrams

Composite Structure Diagrams

Component Diagrams

Deployment Diagrams

- Starter Deployment Diagram
- Basic Deployment Diagram wi...
- Basic Deployment Diagram wi...
- Basic Deployment Diagram wi...
- Node Instance with Nested De...
- Node Instance with Deploy De...
- Single Device with Execution E...
- Database Server with Deploy...

### Starter Package Diagram

The Starter Package Diagram pattern creates a number of Packages and a Package diagram that describes the relationships between them. A package acting as a client is said to depend on another package acting as a supplier. This reliance is modeled on the diagram visually by a dashed line with the arrowhead pointing to the supplier package.

```
graph LR; PA[Package A] --> PB[Package B]
```

Figure 1. Shows a Package diagram with two packages that have dependency relationships indicating the reliance of one package on another.

### Discussion

Nota. Jaramillo, D., 2025.

En el proyecto que se presenta en el repositorio de la asignatura puede encontrar un ejemplo, así mismo puede revisar el siguiente video [diagrama de paquetes](#).



### Actividad de aprendizaje recomendada

Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. Revise.



## Semana 12

### Unidad 6. Diseño arquitectónico

#### 6.4. Patrones arquitectónicos

Un patrón o estilo arquitectónico es “un conjunto de principios generales que proporcionan marco de trabajo abstracto para una familia de sistemas” (Microsoft Corporation, 2009), estos principios son soluciones a problemas comunes de diseño de software, por ello es importante conocerlos y aplicarlos. En la siguiente infografía se resumen algunos patrones arquitectónicos de uso común.

#### Estilos arquitectónicos de uso común

El diseño de software es una actividad compleja, que requiere de una gran capacidad de abstracción para organizar subsistemas y componentes, de modo que el software que se construye cumpla con requerimientos funcionales y no funcionales, los cuales están atados directamente a la arquitectura.

Una de las formas de conseguir esto es el uso de patrones arquitectónicos, los cuales demuestran ser esquemas de organización de subsistemas probados que pueden usarse en diferentes situaciones.



En los últimos años, se ha dado un gran auge en el uso de un nuevo estilo arquitectónico denominado Arquitectura Orientada a Microservicios. En el siguiente video “[¿Qué es la arquitectura de microservicios?](#)”, encontrará algunas pautas para entender qué es y cómo funciona.

##### 6.4.1. Arquitectura capas

Arquitectura por capas en el desarrollo de sistemas

La arquitectura por capas es uno de los patrones más utilizados en el desarrollo de sistemas de software debido a su claridad, organización y facilidad de mantenimiento. Este patrón organiza la aplicación en niveles bien definidos, donde cada capa tiene responsabilidades específicas y se comunica únicamente con la capa inmediatamente inferior o superior. Su propósito principal es separar las preocupaciones, facilitando la escalabilidad, la reutilización de componentes y el mantenimiento del sistema a lo largo de su ciclo de vida.

Generalmente, la arquitectura por capas se estructura en cuatro niveles principales:

- Capa de **Presentación** (Interfaz de Usuario): es la encargada de la interacción con el usuario. Aquí se diseñan las pantallas, formularios o aplicaciones móviles que permiten al cliente o empleado comunicarse con el sistema.
- Capa de **Lógica de Negocio** (Dominio o Aplicación): contiene las reglas de negocio, procesos y operaciones que definen el funcionamiento del sistema. Su función es coordinar las acciones entre la presentación y los datos.
- Capa de **Acceso a Datos**: se encarga de gestionar la comunicación con la base de datos. Aquí se definen las consultas, inserciones, actualizaciones y eliminaciones de la información, aislando la lógica de negocio de los detalles técnicos del almacenamiento.
- Capa de **Datos** (Persistencia): incluye las bases de datos y sistemas de almacenamiento donde se guarda de manera persistente la información del sistema.

#### 6.4.2. Modelo vista controlador

La arquitectura Modelo-Vista-Controlador (MVC), es un patrón ampliamente utilizado en el desarrollo de sistemas de software, especialmente en aplicaciones web y móviles, porque separa de manera clara las

responsabilidades del sistema en tres componentes principales: Modelo, Vista y Controlador. Esta separación permite construir sistemas más organizados, fáciles de mantener y escalables.

- **Modelo:** el modelo representa la capa que maneja los datos y la lógica del negocio. Es responsable de gestionar la información, acceder a la base de datos, validar reglas de negocio y mantener la coherencia de los datos del sistema.
- **Vista:** la vista es la encargada de la presentación de la información al usuario. Incluye las pantallas, formularios, reportes y todo lo relacionado con la interfaz gráfica. La vista muestra datos que provienen del modelo, pero no tiene lógica de negocio, únicamente se enfoca en la interacción visual con el usuario.
- **Controlador:** el controlador actúa como intermediario entre el modelo y la vista. Recibe las acciones del usuario (*clicks*, solicitudes, entradas de datos), las procesa, invoca al modelo para realizar la lógica correspondiente y luego actualiza la vista con los resultados.

#### 6.4.3. Microservicios

La arquitectura de microservicios es un patrón de diseño de software que consiste en dividir una aplicación en múltiples servicios pequeños, independientes y autónomos, que se comunican entre sí mediante interfaces ligeras, como API REST o mensajería. Cada microservicio está enfocado en realizar una tarea o proceso específico, y puede desarrollarse, desplegarse y escalarse de forma independiente del resto.

Este enfoque contrasta con las aplicaciones monolíticas, donde todos los módulos están integrados en una sola estructura. Con los microservicios, las organizaciones logran mayor flexibilidad, escalabilidad y resiliencia, ya que un fallo en un servicio no necesariamente afecta a toda la aplicación.

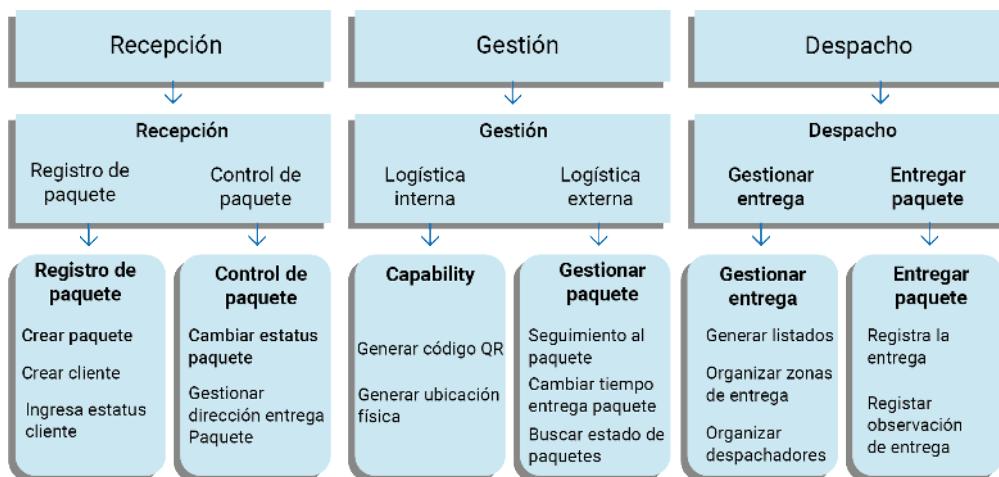
Las características principales de la arquitectura de microservicios incluyen:

- Descomposición en servicios pequeños: cada microservicio se centra en una funcionalidad concreta del sistema.
- Autonomía: los equipos pueden desarrollar y desplegar microservicios de manera independiente.
- Escalabilidad selectiva: solo se escalan los servicios que lo requieren, optimizando recursos.
- Comunicación ligera: los microservicios se comunican a través de API bien definidas.
- Tecnología heterogénea: cada servicio puede construirse con el lenguaje o la base de datos más adecuada.

Podemos encontrar una descomposición funcional con la ayuda de un mapa de capacidades.

**Figura 61**

Mapa de capacidades TRAMIL (Archimate)



Nota. Jaramillo, D., 2025.

Esto nos permitirá ir encontrando aquellas capacidades del sistema que puedan ser trabajadas como microservicios.

En este contexto, tras el análisis de los enfoques (arquitectura en capas, MVC y microservicios), se recomienda revisar el presente módulo didáctico, en el cual se exponen casos de estudio aplicados al contexto de la empresa TRAMIL.

### Casos de estudio de arquitecturas de software en TRAMIL

Concluida la revisión de los casos, puede establecerse que los distintos enfoques arquitectónicos aportan soluciones complementarias que fortalecen la organización, escalabilidad y mantenimiento de los sistemas en contextos empresariales como TRAMIL.

## 6.5. Estilos arquitectónicos

Los estilos arquitectónicos definen patrones de organización para los sistemas de software, estableciendo cómo se estructuran sus componentes y cómo interactúan entre sí. A continuación, se presentan algunos de los más relevantes.

### 6.5.1. Cliente servidor

El estilo arquitectónico cliente–servidor es uno de los modelos más utilizados en el desarrollo de sistemas distribuidos. Su funcionamiento se basa en la interacción entre dos entidades principales: el cliente y el servidor. El cliente representa al usuario o la aplicación que solicita un servicio, mientras que el servidor es el componente que responde a dichas solicitudes procesando la información, ejecutando operaciones y devolviendo resultados.

En este modelo, la comunicación entre cliente y servidor se realiza a través de protocolos de red, siendo el más común el HTTP/HTTPS en aplicaciones web. La ventaja principal es que se logra centralizar los recursos y la lógica de negocio en el servidor, mientras que los clientes se encargan de la interacción y presentación.

Entre las características del estilo cliente–servidor destacan:

- Separación de responsabilidades: los clientes gestionan la presentación y el servidor gestiona los datos y la lógica.
- Escalabilidad: es posible aumentar la capacidad del servidor para atender a más clientes.
- Mantenimiento centralizado: las actualizaciones se realizan en el servidor, sin necesidad de modificar cada cliente.
- Dependencia de red: su funcionamiento depende de la conectividad entre cliente y servidor.

A partir de estas características, resulta pertinente ilustrar su aplicación mediante un ejemplo aplicado.

### CASO DE ESTUDIO

En la empresa TRAMIL, dedicada a la gestión de paquetes y entregas, la arquitectura cliente–servidor puede aplicarse de manera efectiva para soportar las operaciones de la organización.

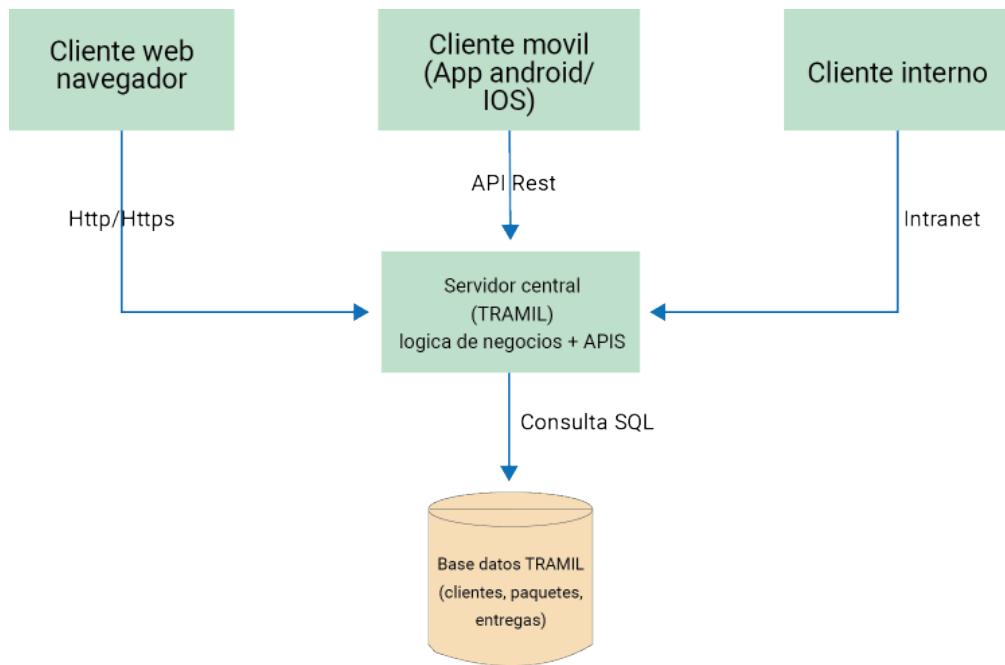
**Cliente:** Los clientes de TRAMIL pueden acceder al sistema a través de una aplicación web o móvil. Desde esta interfaz, el usuario realiza acciones como registrar un nuevo envío, consultar el estado de un paquete, o generar un reclamo. Del mismo modo, los empleados pueden acceder mediante clientes internos para registrar la llegada de paquetes, asignar rutas de entrega o emitir reportes.

**Servidor:** El servidor central de TRAMIL almacena toda la información relacionada con paquetes, clientes y entregas en una base de datos. Además, contiene la lógica de negocio que valida los registros, calcula rutas de entrega y gestiona la trazabilidad de los paquetes. Por ejemplo, cuando un cliente consulta el estado de un envío, el cliente (aplicación móvil) envía la solicitud al servidor. Este accede a la base de datos, obtiene el estado del paquete y devuelve la información al cliente para su visualización.

Gracias a este modelo, TRAMIL puede ofrecer un servicio centralizado, seguro y actualizado en tiempo real. Además, si la demanda crece, es posible fortalecer el servidor o implementar平衡adores de carga para atender a más usuarios simultáneamente.

El siguiente diagrama representa la arquitectura Cliente-Servidor aplicada al sistema de TRAMIL. Los clientes (web, móvil e internos) envían solicitudes al servidor central, el cual procesa las operaciones y consulta la base de datos para devolver la información solicitada.

**Figura 62**  
Arquitectura Cliente-Servidor TRAMIL



Nota. Jaramillo, D., 2025.

## Publicar

El estilo arquitectónico Publicar-Suscribirse (Publish-Subscribe, o Pub/Sub) es un patrón ampliamente utilizado en el desarrollo de aplicaciones modernas, especialmente en sistemas distribuidos y aplicaciones orientadas a eventos.

Este enfoque permite la comunicación asíncrona entre componentes, desacoplando a los emisores de información (publicadores) de los receptores (suscriptores). Esta separación facilita la escalabilidad, la flexibilidad y la adaptabilidad del sistema, aspectos fundamentales en entornos dinámicos y de alta complejidad.

En un sistema Pub/Sub, los publicadores envían mensajes a un canal o tema sin necesidad de conocer quién recibirá la información. Los suscriptores, a su vez, se registran en estos canales y reciben únicamente los mensajes de interés. Esta arquitectura reduce la dependencia directa entre módulos, lo que facilita el mantenimiento y la evolución de la aplicación. Además, permite que nuevos suscriptores se integren al sistema sin afectar a los publicadores existentes, promoviendo la modularidad y la expansión de funcionalidades sin alterar la estructura original.

El Pub/Sub se utiliza frecuentemente en aplicaciones en tiempo real, sistemas de notificación, plataformas de streaming de datos, comercio electrónico y servicios en la nube. Por ejemplo, en un sistema de e-commerce, un módulo de inventario puede publicar cambios en el stock, mientras que otros módulos, como la gestión de pedidos o las notificaciones a clientes, se suscriben a estos eventos para reaccionar automáticamente. De esta manera, se logra una sincronización eficiente entre componentes sin necesidad de llamadas directas o integración rígida.

Entre las ventajas del estilo Pub/Sub destacan su escalabilidad horizontal, el desacoplamiento de componentes, la posibilidad de procesar eventos en tiempo real y la flexibilidad para integrar nuevos servicios. Sin embargo, también plantea desafíos, como garantizar la entrega confiable de mensajes, manejar la consistencia de datos y monitorear el flujo de eventos, especialmente en sistemas distribuidos a gran escala. Herramientas y plataformas como Apache Kafka, RabbitMQ o Google Cloud Pub/Sub han surgido para implementar este estilo de manera eficiente, proporcionando mecanismos de persistencia, enrutamiento y recuperación de mensajes.



El estilo arquitectónico Publicar-Suscribirse representa una estrategia eficaz para el desarrollo de aplicaciones modernas que requieren comunicación asíncrona, modularidad y escalabilidad.



## Actividades de aprendizaje recomendadas

Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. Revisar el video [Microservicios | ¿qué son los microservicios?](#) [Introducción a microservicios](#), para conocer un poco más sobre la arquitectura orientada a microservicios.
2. Desarrolle la siguiente autoevaluación de la unidad 6 para comprobar sus conocimientos.



## Autoevaluación 6

1. La arquitectura del software se puede evidenciar a dos niveles: pequeño y grande. En relación con los patrones arquitectónicos, ¿a qué nivel corresponde el modelo Vista–Controlador?
  - a. Pequeño.
  - b. Grande.
  - c. Es independiente.
  - d. Ambos niveles simultáneamente.
2. Al diseñar la arquitectura del software, los requerimientos funcionales se implementan en:
  - a. A nivel de arquitectura.
  - b. En componentes externos.
  - c. En componentes individuales.
  - d. En la Interfaz de Usuario.



3. Documentar la arquitectura favorece la comunicación con los participantes porque:

- a. Es una presentación de alto nivel del sistema.
- b. Se elabora en una etapa temprana del desarrollo.
- c. Muestra cómo se organiza el sistema.
- d. Facilita la creación de código automáticamente.

4. El modelo de análisis es insuficiente para pasar a la implementación porque:

- a. Carece de información de la estructura interna del sistema.
- b. Las descripciones de casos de uso no son lo suficientemente detalladas.
- c. Faltan diagramas de clases.
- d. No incluye pruebas unitarias.

5. Para garantizar la disponibilidad en la arquitectura del sistema, se debe implementar:

- a. Activos críticos del sistema protegidos.
- b. Operaciones críticas en un componente individual.
- c. Incluir componentes redundantes.
- d. Uso exclusivo de bases de datos distribuidas.

6. No es posible que una aplicación cumpla con todos los atributos de calidad porque:

- a. Algunos pueden entrar en conflicto entre sí.
- b. Puede resultar demasiado costoso.
- c. Puede tomar demasiado tiempo implementarlo.
- d. Depende exclusivamente de la arquitectura elegida.

7. Según el modelo arquitectónico de Kruchten 4+1, las clases del sistema se aprecian en la:

- a. Vista de desarrollo.



- b. Vista del proceso.
  - c. Vista lógica.
  - d. Vista de casos de uso.
8. Los lenguajes utilizados específicamente para describir la arquitectura del sistema son:
- a. UML.
  - b. ADL.
  - c. C.
  - d. BPMN.
9. Desde el punto de vista ágil en el desarrollo de software, la arquitectura:
- a. Ya no se necesita, el sistema evoluciona con los requerimientos.
  - b. Debe ser más robusta que en las no ágiles.
  - c. El nivel de especificación depende de la complejidad del sistema, no de la metodología.
  - d. Se define solo al final del proyecto.
10. Al descomponer un sistema en subsistemas, la recomendación frente a cohesión y acoplamiento es:
- a. Priorizar la cohesión.
  - b. Aplicar la heurística  $7 \pm 2$ .
  - c. Minimizar el acoplamiento.
  - d. Equilibrar ambos en proporciones iguales.

[Ir al solucionario](#)

## Resultado de aprendizaje 2 y 3:

- Conoce e identifica las fases del ciclo de vida de desarrollo de sistemas.
- Identifica el ciclo de vida a utilizar en el desarrollo de un proyecto de software.

Para alcanzar los resultados de aprendizaje planteados, se fomentará una comprensión profunda de las fases del ciclo de vida del desarrollo de sistemas y aprenderán a elegir el ciclo de vida más adecuado para cada proyecto de software. Se hará énfasis en el diseño e implementación, incluyendo el diseño Orientado a Objetos con UML, el uso de patrones de diseño y la resolución de conflictos durante la implementación. Además, se estudiarán los conceptos esenciales de las pruebas de software, las pruebas de desarrollo y la gestión de las actividades de prueba. Este conocimiento les permitirá aplicar métodos efectivos para diseñar, implementar y asegurar la calidad en sus proyectos de software.

### Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



### Semana 13

Una vez que ha estudiado los estilos arquitectónicos, el diseño del sistema se convierte en una actividad más técnica que busca crear modelos que permitan la implementación de aplicaciones que cumplan con las especificaciones tanto funcionales como no funcionales. En esta unidad se estudiará el Diseño orientado a objetos.

## Unidad 7. Diseño, construcción e implementación

En esta unidad vamos a revisar las actividades de modelado del Diseño Orientado a Objetos. El propósito de esta unidad es crear el diseño de los subsistemas que deberán implementarse y construir los modelos de implantación.

Continuando con el estudio acerca del proceso de desarrollo de software, nos adentraremos en las actividades del diseño, el cual, como se indicó en la unidad 6, se trata de un proceso creativo, que requiere capacidad para resolver problemas, sin embargo, no por ello es improvisado, sino que es un proceso de alta complejidad, y, por lo tanto, requiere mucho conocimiento.

Las actividades de diseño, se orientan a decidir varios aspectos de implementación que provean soluciones que satisfagan tanto los requerimientos funcionales como los no funcionales y por ello han de evaluar diferentes alternativas. Cada decisión que se tome, tendrá alto impacto en el desempeño y calidad del producto resultante. Por consiguiente, las actividades de diseño se orientan a resolver los siguientes aspectos:

- Selección de productos de software de mercado o componentes heredados, los cuales pueden reducir costos.
- Mapeo de subsistemas a hardware.
- Diseño de la infraestructura de administración de la información persistente.
- Definición de una política de control de accesos a los objetos. Esta política impacta en la forma como estos objetos se distribuyen.
- Diseño del control de flujo global. Determinar la secuencia de operaciones impacta la interfaz del sistema.

El diseño y la implementación están estrechamente relacionados. Se puede usar UML para representar el diseño siempre y cuando se vaya a implementar utilizando algún lenguaje de programación como Java, Visual Studio, PHP u otros. El propósito de esta unidad es mostrar cómo el modelado del sistema y el diseño arquitectónico se ponen en práctica en el desarrollo de un diseño



Orientado a Objetos. Además, se busca introducir algunos temas del diseño y la implementación como la reutilización del software, la administración de la configuración y el desarrollo de código abierto.

El modelado del diseño implica cumplir algunas actividades, que intentaré resaltar a continuación.

El primero es el modelo de contexto, que en realidad lo que hace es ubicar al sistema en desarrollo, en relación con otros sistemas, ello es esencial porque permitirá identificar aspectos como las interfaces que se deben desarrollar para dichas interacciones, las posibles protecciones del sistema y también las responsabilidades; es decir, incluir en nuestro diseño los aspectos que no están cubiertos por otros sistemas y que son necesarios para cubrir los requerimientos. Si lo recuerda, esto era parte de la decisión respecto de lo que se debería hacer con los sistemas existentes o con otros que podrían resultar más adecuados comprarlos antes que desarrollarlos.

Esto lleva a identificar nuevos casos de uso o a complementar los casos de uso que ya tenemos definidos. Recuerde que los actores ya fueron identificados en las etapas iniciales del proceso de desarrollo y son personas, aplicaciones o dispositivos capaces de interactuar con nuestro sistema.

Otro de los aspectos que contempla este apartado es el Diseño de la arquitectura del sistema, para ello se deben utilizar los patrones arquitectónicos revisados en la unidad anterior.



Recuerde que cada uno de los componentes de la arquitectura del sistema, es un subsistema, el cual, a su vez, está conformado por clases que tienen relación entre ellos. Por ello, los conceptos sobre cohesión y acoplamiento estudiados, son necesarios al momento de estructurar los subsistemas.

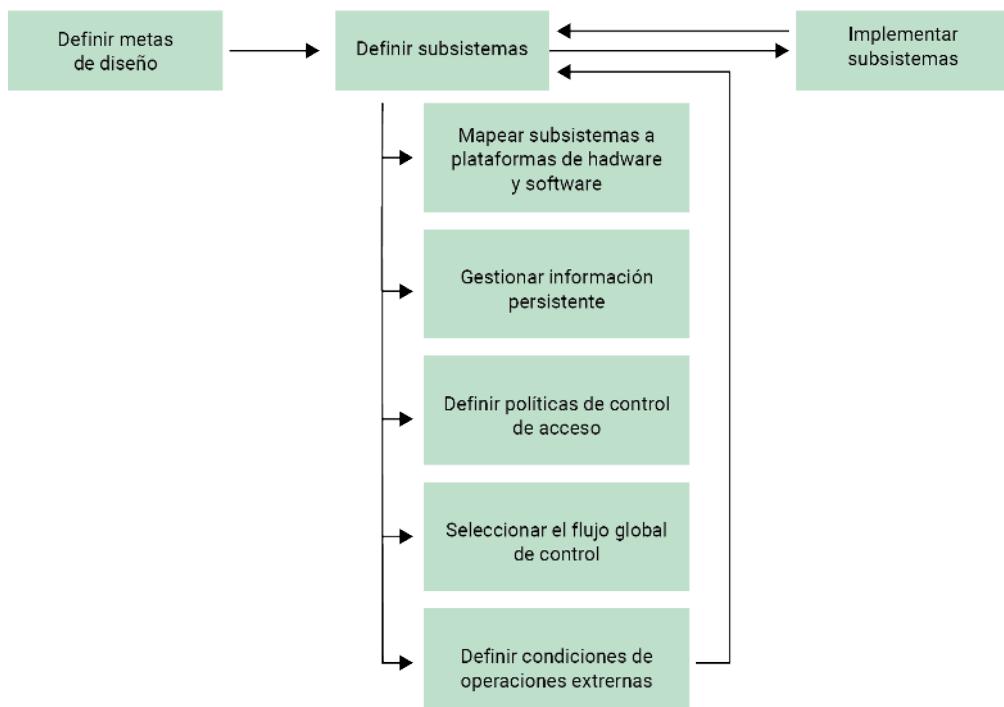
Con relación a los modelos de diseño, básicamente podemos hablar de modelos estructurales y modelos dinámicos, los cuales se deben complementar con la definición de interfaces entre componentes del sistema.

Para complementar esta información, vamos a intentar resumir los pasos para el modelado con UML. Para ello, revisemos el proceso general del diseño Orientado a Objetos del sistema, como se muestra en la figura 63.

Ya se ha comentado respecto de las metas del diseño, y la definición e implementación de subsistemas, los cuales se sustentan en el diseño arquitectónico. Ahora se revisarán algunas de las actividades planteadas en la figura 63, que son esenciales para el proceso de diseño.

Puesto que ya hemos acordado que la representación de los modelos se realizará con UML, empezaremos identificando los elementos de UML que se requieren en cada caso.

**Figura 63**  
*Actividades del diseño de sistemas*



*Nota.* Adaptado de *Object-oriented software engineering : using UML, patterns, and Java [Ilustración]*, por Bruegge, B. y Dutoit, A., 2018, Prentice Hall, CC BY 4.0.

## Mapeo de subsistemas a plataformas de hardware y software

El mapeo de subsistemas se hace a mediante la vista de despliegue, la cual según (Rumbaugh et al., 2005) muestra la organización física de los nodos en tiempo de corrida, esta vista se modela con los diagramas de despliegue.

Los elementos que conforman un diagrama de despliegue son:

- Conforme lo establecen (Rumbaugh et al., 2005), Un nodo es un recurso computacional en tiempo de corrida (dispositivo físico o ambiente de ejecución), el cual tiene como mínimo memoria y capacidad de procesamiento. Pueden colocárseles estereotipos para diferenciar clases de recursos tales como CPUs, dispositivos, memoria. Los nodos ejecutan los componentes.
- El (Wambler, 2005) define a un Artefacto, como una entidad autocontenido que provee servicios, puede verse como un elemento físico y hay varios tipos de artefactos tales como: archivos de modelos, archivos fuente, scripts, archivos ejecutables, tablas de bases de datos, entregables de desarrollo, documentos de procesamiento de textos y mensajes de correo.
- La presencia de un artefacto en un nodo se muestra colocándolo dentro del nodo con el estereotipo correspondiente. Un artefacto implementa un componente u otra clase, esto se representa con una línea discontinua etiquetada con <>manifest>, la cual denota dicha implementación, la flecha se dirige al componente, esta relación se denomina manifestación.

## Gestionar la información persistente

Entendemos como información persistente a aquella que permanece aún después de que el sistema se haya ejecutado, es decir los datos de la empresa o del cliente, esta información puede almacenarse en diferentes formatos, y en este punto es necesario decidir dónde y cómo se almacenará la información.

A continuación, se revisan brevemente los pasos requeridos para gestionar la información:

- Identificar objetos persistentes: Los objetos que normalmente son candidatos para persistir son los objetos entidad que se definieron durante la etapa de análisis, los demás objetos tienen una duración temporal en tiempo de ejecución, sin embargo, es necesario analizarlos a todos antes de tomar la decisión.
- Seleccionar estrategia de almacenamiento: Existen varias formas y formatos para almacenar la información persistente, esta decisión suele estar atada principalmente a los requerimientos no funcionales. Los formatos más comunes para almacenamiento en los que podemos elegir son:
  - **Archivos planos**: administrados por el sistema operativo, y se guardan como secuencias de bytes, tienen la ventaja de la simplicidad, sin embargo, puede ser complejo el acceso a los mismos, incluso la concurrencia puede verse comprometida, uno de los usos más comunes de este tipo de archivos es guardar rastros de auditoría.
  - **Bases de datos relacionales**: medio más ampliamente utilizado para el almacenamiento de información, la información se almacena en tablas y hay un conjunto de elementos que facilitan su manipulación, como los lenguajes de definición y lenguajes de consulta, lo cual las hace muy útiles para manejar grandes volúmenes de información y soportar concurrencia en miles de transacciones. Una de las desventajas son los costos.
  - **Bases de datos orientadas a objetos**: Son motores de base de datos capaces de almacenar objetos con sus relaciones, entre las ventajas que tienen es que permiten el uso de tipos de datos abstractos, favorecen la herencia, pero por otro lado el rendimiento es muy inferior al de las bases de datos relacionales.

## Políticas de control de acceso

En los sistemas multiusuario es necesario definir qué objetos pueden ser accedidos por los actores y que clase de acceso pueden tener en función del trabajo que deben realizar con el sistema, estas decisiones desde luego van relacionadas con los requerimientos de seguridad identificados.

Los accesos se pueden representar utilizando varias técnicas, como por ejemplo una tabla “acceso global” una “lista de control de accesos” y una asociación entre el actor y su capacidad.

El control de accesos también puede aplicarse a nivel de seguridad de acceso entre equipos de cómputo, para lo cual es necesario establecer un esquema que proteja sobre todo los datos, uno de los mecanismos más utilizados es el firewall que provee un control de acceso estático, o el uso de patrón de diseño proxy, para acceso dinámico.

Otros mecanismos para restringir accesos son la autenticación que normalmente, ofrece acceso con contraseña y la encriptación que provee mecanismo de protección a través de algoritmos de encriptación cuyo propósito es evitar que la información sea comprendida por intrusos, para poder interpretar la información es necesario desencriptarla.

## 7.1. Patrones de diseño

Los patrones de diseño son soluciones generales para problemas comunes de diseño de aplicaciones, son muchos los patrones que se pueden y se deben usar dependiendo de la situación que estemos resolviendo, para ello es importante estudiarlos y aplicarlos cuando se vea necesario.

Gamma et al (1995), definen a un patrón de diseño como una descripción de clases y objetos comunicándose entre sí, adaptada para resolver un problema de diseño general en un contexto particular, tienen amplio uso en el diseño de software y su uso permite resolver una gran cantidad de problemas comunes.

Se han creado diferentes tipos de patrones para diferentes propósitos, Blancarte (2016), presenta 3 tipos de patrones como son: **creacionales, estructurales y de comportamiento**, a continuación, se listan los patrones de diseño en estas categorías.

### 7.1.1. Patrones creacionales

Dentro de la ingeniería de software, los patrones de diseño son soluciones reutilizables y probadas a problemas comunes en el desarrollo de sistemas. En particular, los patrones creacionales se centran en el proceso de instanciación de objetos, es decir, en la forma en que se crean, configuran y presentan al sistema. Su propósito principal es abstraer la lógica de creación, evitando dependencias innecesarias y ofreciendo mayor flexibilidad en la construcción de software.

Los patrones creacionales son relevantes porque permiten desacoplar el código cliente de las clases concretas que se instancian, lo que facilita la extensibilidad y el mantenimiento. En lugar de que el cliente decida directamente qué clase crear, los patrones proporcionan mecanismos para delegar esta responsabilidad, promoviendo un diseño más robusto y adaptable a cambios futuros.

Aquí podemos encontrar algunos de estos patrones:



- Factory Method
- Abstract Factory
- Singleton
- Builder
- Prototype
- Object Pool

### 7.1.2. Patrones estructurales

En el campo de la ingeniería de software, los patrones de diseño estructurales se enfocan en la manera en que las clases y los objetos se organizan para formar estructuras más grandes y flexibles. Su objetivo principal es simplificar las relaciones entre componentes, favoreciendo la reutilización de código y reduciendo la complejidad del sistema. Estos patrones permiten que los objetos colaboren entre sí sin generar dependencias rígidas, lo cual facilita la escalabilidad y el mantenimiento del software.

Una característica clave de los patrones estructurales es que ofrecen soluciones para componer objetos y clases, de manera que las modificaciones internas no afecten al conjunto. En lugar de crear nuevas funcionalidades desde cero, aprovechan la integración de componentes ya existentes. Aquí podemos encontrar algunos de estos patrones:



- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

### 7.1.3. Patrones de comportamiento

Los patrones de diseño de comportamiento son un conjunto de soluciones orientadas a definir cómo los objetos y clases interactúan entre sí, distribuyendo de manera eficiente la comunicación y la responsabilidad. A diferencia de los patrones creacionales, que se centran en la instanciación de objetos, y de los estructurales, que se enfocan en su composición, los patrones de comportamiento buscan optimizar la colaboración y el flujo de mensajes dentro de un sistema.

Estos patrones son de gran importancia porque permiten reducir el acoplamiento entre componentes, favorecen la reutilización del código y facilitan la extensión de funcionalidades sin modificar drásticamente la arquitectura. Además, mejoran la claridad en el diseño al definir cómo se delegan las responsabilidades y cómo reaccionan los objetos frente a diferentes estímulos. En la siguiente figura se ilustran algunos de estos patrones.

**Figura 64**  
Patrones de comportamiento



Nota. Jaramillo, D., 2025.

## 7.2. Conflictos en la implementación

En este apartado se estudian algunas cuestiones comunes en el desarrollo de software profesional, que deben tener un enfoque para abordarlas. Entre estas cuestiones tenemos por ejemplo la reutilización de código, que puede ahorrar mucho dinero y tiempo de desarrollo y que hoy en día es bastante común; la administración de la configuración que se preocupa por el problema del

almacenamiento y control de las diferentes versiones del código que se van generando a lo largo del ciclo de desarrollo, cómo evitar conflictos entre estas sobre todo si se trabajan en equipo; y, el denominado Huésped-Objetivo, que en realidad de preocupa por las diferencias entre entorno de desarrollo y el ambiente de producción

A continuación, se plantean algunas preguntas que le ayudarán a profundizar en estos temas.

#### **Respecto de la reutilización de software.**

- Este aspecto aborda el problema de completar el desarrollo en el menor tiempo posible y minimizando costos. ¿Si los costos de producir el software se reducen a cero, en qué costos podría incurrirse? ¿De qué dependen estos costos?
- ¿A cuántos niveles se puede reutilizar el software? ¿Cuál es el más conveniente? ¿Cuál podría resultar más complejo? ¿Cuál el más costoso?
- ¿Qué métodos se pueden utilizar para determinar si una pieza de código es reutilizable? ¿En qué parte del diseño se debe evaluar esta posibilidad?
- ¿Se puede considerar al código abierto como software reutilizable?

#### **Respecto de la administración de la configuración.**

- ¿Qué problemas resuelve la administración de la configuración?
- ¿Es conveniente utilizar este tipo de herramientas cuando hay solo 1 o 2 desarrolladores?
- ¿Qué tan costoso resulta trabajar con estas herramientas?

#### **Respecto del desarrollo huésped-objetivo**

- ¿Qué estrategias se pueden utilizar para reducir el problema de incompatibilidad entre la plataforma de desarrollo y la plataforma de ejecución?
- ¿Con las plataformas web, móviles o soluciones en la nube, qué efecto tendría el problema planteado sobre huésped-objetivo?



Si ha tenido problemas sobre todo con los modelos desarrollados en UML, le invito a revisar nuevamente el tema de modelado con UML, y buscar información complementaria en la bibliografía recomendada.



## Resultado de aprendizaje 1 y 3:

- Conoce las principales áreas de conocimiento de la ingeniería de software.
- Identifica el ciclo de vida a utilizar en el desarrollo de un proyecto de software.

Para alcanzar estos resultados, se estudiarán las principales áreas de la ingeniería de software a través del diseño, construcción e implementación de sistemas, enfatizando la transición del diseño a la implementación. Asimismo, se abordarán aspectos de infraestructura, seguridad y persistencia, lo que permitirá seleccionar el ciclo de vida más adecuado y aplicar prácticas que garanticen proyectos de software sólidos y confiables.

### Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



### Semana 14

#### Unidad 7. Diseño, construcción e implementación

Los patrones de diseño son soluciones generales para problemas comunes de diseño de aplicaciones. Son muchos los patrones que se pueden y se deben usar dependiendo de la situación que estemos resolviendo, para ello es importante estudiarlos y aplicarlos cuando se vea necesario.

Para estudiar este tema, remítase a la guía didáctica en el apartado 7.2; en ella encontrará referencias para descargar material explicativo de este tipo de patrones.

## 7.3. Del diseño a la implementación

Para dar continuidad al estudio, resulta fundamental vincular los conceptos teóricos con su aplicación práctica. Es precisamente en este punto donde se establece el puente entre el diseño del sistema y su posterior implementación, asegurando que los modelos definidos no se queden únicamente en representaciones gráficas, sino que sirvan como base sólida para la construcción del software.

### 7.3.1. Modelo generado a partir del diagrama de clases

La importancia de reflejar los modelos diseñados tras un análisis de la problemática que se resolverá con la etapa de construcción del sistema o, en nuestro caso, uno de los componentes, es muy importante

Esto reflejará el esfuerzo que hemos realizado en la etapa de modelado y que se llegue a la construcción del sistema, es decir, vamos a llegar a la etapa de programación utilizando Java como el lenguaje base para la programación y la persistencia de la información

Empezaremos con una explicación de los recursos que utilizaremos para la construcción. En el repositorio de la asignatura encontrará instaladores de la versión de prueba

- El proyecto en *Enterprise Architect*, desarrollado durante esta guía como base, se utilizará para este apartado. Considere que la mayoría de las herramientas de modelado de UML nos permiten la generación de código a partir de nuestro modelo de clases.

### 7.3.2. Generación de código a partir de los modelos

La primera parte de este trabajo será obtener el código base para la explicación de la implementación de una determinada funcionalidad. Este código lo vamos a nombrar como modelo, puesto que estarán las clases en su versión inicial.

Es así que pasaremos desde el diagrama de clases de nuestro proyecto para la generación y depuración del código que está basado en *Enterprise Architect*.

Para ello, vamos a partir del diagrama de clases que se haya realizado, donde deben constar:

- Atributos: con sus nombres y sus tipos.
- Operaciones.
- Relaciones: con su multiplicidad y la dirección de las mismas.

¿Tiene claro la diferencia de agregación y composición en código?

Vamos a poner en consideración algunos temas que se presentan en el diagrama de clases y deben estar reflejados en la generación del código desde este diagrama:

#### a. Navegabilidad

Es importante determinar la dirección que tiene cada relación

- Si la flecha apunta de la ClaseA a la ClaseB, se lee como “ClaseA tiene una ClaseB” y se dice que la asociación o navegabilidad es unidireccional. Traducido a Java, esto significa que hay un atributo en la clase A que hace referencia a un objeto de la clase B.
- Si no dibujamos la flecha, se lee “ClaseA tiene una ClaseB y ClaseB tiene una ClaseA”, y se dice que la asociación o navegabilidad es bidireccional. En Java, ambas clases tendrían atributos que se hacen referencia recíprocamente.

#### Ejemplo:

Presentamos una asociación con las clases Cliente y Pedido. En el siguiente diagrama, Cliente guarda información sobre los pedidos y Pedido tiene información sobre el cliente que lo realizó. La navegabilidad es, por tanto, bidireccional, figura 65:

**Figura 65**

Asociación con las clases cliente y pedido



Nota. Jaramillo, D., 2025.

En la figura 66 podemos ver las 2 posibilidades que se presentarían al generar el código

**Figura 66**

Possibilidades de generación de código bidireccional

```
public class Cliente {  
    private String Nombre;  
    private ArrayList <Pedido> pedidos;  
}  
  
public class Pedido {  
    private int Orden;  
    private Cliente cliente;  
}
```

Nota. Jaramillo, D., 2025.

¿pero cuál sería lo correcto? Aquí podrían darse muchas interpretaciones, pero desde el punto de vista del autor de esta guía sería la segunda pues un atributo del pedido sería un cliente, mas no un atributo del Cliente una lista de pedidos. Nuevamente reiterar que la interpretación puede tener varios puntos de vista

Ejemplo de asociación unidireccional, donde se resuelve de acuerdo al sentido de la relación (figura 67).

**Figura 67**

Ejemplo de asociación unidireccional



```

public class Cliente {
    private String Nombre;
    private ArrayList <Pedido> pedidos;
}

public class Pedido {
    private int Orden;
    private Cliente cliente;
}
  
```



```

public class Cliente {
    private String Nombre;
}

public class Pedido {
    private int Orden;
    private Cliente cliente;
}
  
```

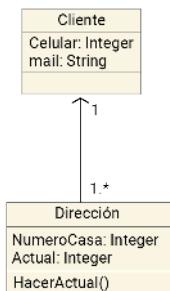
Nota. Jaramillo, D., 2025.

### b. Agregación

Esta relación nos presenta que el objeto dirección existe y que lo podremos tomar para relacionarlo con nuestro cliente, la figura 68 muestra dicha relación donde el rombo esta sin pintar.

**Figura 68**

Relación de agregación



```

package CLA_tramil;

public class Cliente extends Personas {
    private String Celular;
    private String mail;
    private Direccion myDireccion;

    public Cliente(String Identificacion, String Apellidos, String Nombres, //nombre
                  String Celular, String mail, Direccion myDireccion) { //hijo
        super(Identificacion, Apellidos, Nombres);
        this.Celular = Celular;
        this.mail = mail;
        this.myDireccion = myDireccion;
    }
}
  
```

Nota. Jaramillo, D., 2025.

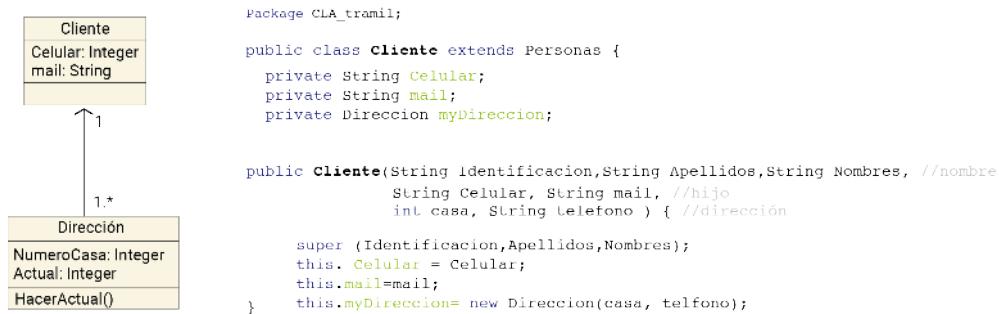
En el código nos podemos fijar que estamos igualando el atributo dirección con un parámetro que llega llamado Mydireccion.

### c. Composición

La composición nos indica principalmente que el objeto dirección no podrá existir por sí solo, es decir que únicamente podrá existir si existe el objeto cliente. El constructor refleja eso, al momento de hacer new Dirección (). En la figura 69 se muestra dicha relación donde está el rombo pintado

**Figura 69**

*Relación de composición*



Nota. Jaramillo, D., 2025.

El código que se presenta la composición y agregación no está generando lo que está representado en el diagrama ¿Qué le falta? Lea la nota al pie [\[3\]](#)

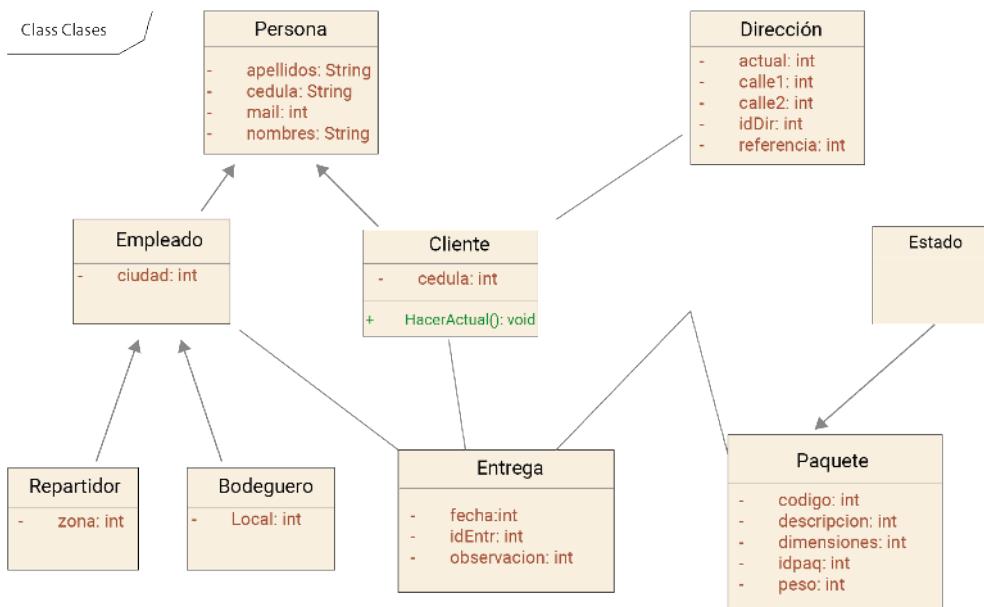
Recuerde la implementación de herencia.

- **Generación desde EA**

Vamos a considerar el siguiente diagrama de clases, que se muestra en la figura 70 pero antes ¿está completo? ¿Qué le falta? ¿es posible generar código así?:

**Figura 70**

Diagrama clases base para la generación de código



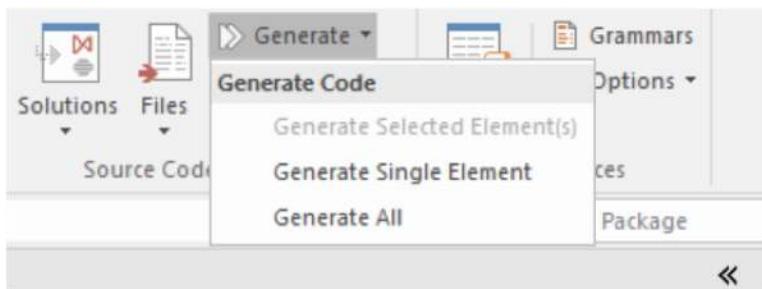
Nota. Jaramillo, D., 2025.

Podemos notar que existe relaciones de: herencia, composición y asociación, por lo que vamos a hacer el primer proceso de generación para poder determinar si lo que está representado y configurado en nuestro diagrama es realmente lo que queremos modelar o tenemos errores en la configuración de relaciones en esta herramienta

El proceso para la generación en EA lo podemos encontrar en la pestaña develop y luego en el ícono de generar, obteniendo las clases. Procederemos a generar todo el código y a continuación se presentará esta pantalla (figuras 71 y 72).

**Figura 71**

Opción de generación de código en EA

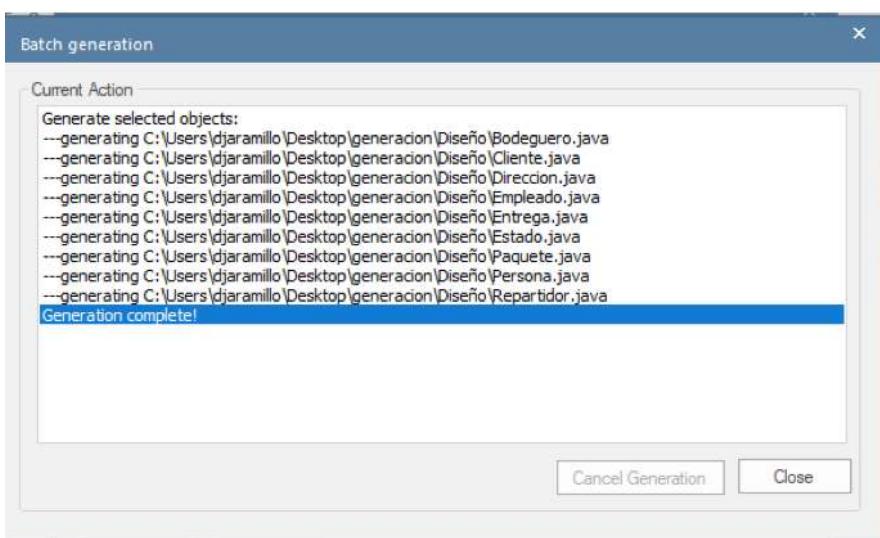


Nota. Jaramillo, D., 2025.

Posterior, se muestra la ejecución completa del proceso de generación de código en EA.

**Figura 72**

Ejecución del proceso de Generación de código EA



Nota. Jaramillo, D., 2025.

Ahora, observe la siguiente figura donde vamos a analizar las mismas desde nuestro IDE.

**Figura 73**

Vista desde el IDE



Este equipo	Nombre	Fecha de modificación
Universidad Técnica P	Bodeguero	1/8/2022 10:57
d Técnica Particular de	Cliente	13/9/2022 9:16
D	ddd.eapx	2/8/2022 23:27
s	Direccion	1/8/2022 10:57
ntos	Empleado	1/8/2022 10:57
	Entrega	1/8/2022 10:57
	Estado	1/8/2022 10:57
	Paquete	1/8/2022 10:57
	Persona	1/8/2022 10:57
	Repartidor	1/8/2022 10:57

Nota. Jaramillo, D., 2025.

Dónde se debe colocar, donde se generarán las clases, a partir de esto debemos revisar el resultado para realizar una corrección y obtener lo que se debe modelar. Este proceso puede resultar tedioso en los inicios, pero luego con el dominio de los temas se harán pocas generaciones.

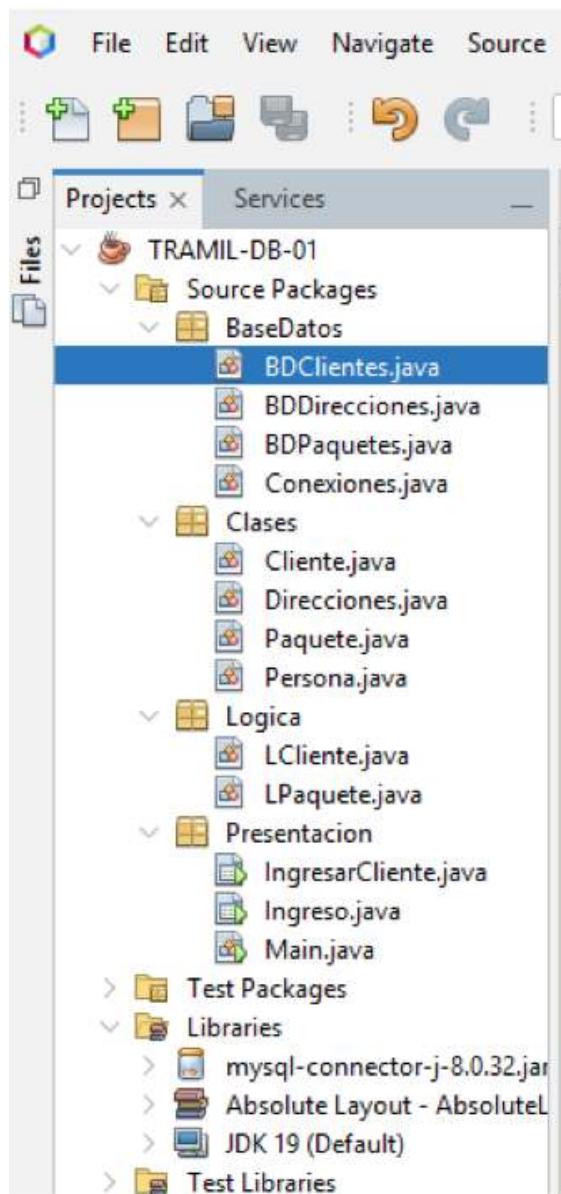
### 7.3.3. El Modelo Capas en la implementación

A partir del diagrama de clases vamos a obtener nuestro modelo, el mismo que nos permitirá trabajar en una IDE. Para explicación utilizaremos NetBeans y java como lenguaje de programación.

Creamos el entorno de desarrollo aplicando el patrón por capas como se muestra en la figura 74.

**Figura 74**

Proyecto Netbeans por capas



Nota. Jaramillo, D., 2025.

Como se puede apreciar tenemos nuestro modelo generado que se encuentra en el paquete de clases.

La figura 75 muestra la clase Cliente con un constructor que hace referencia:

- Generalización/Herencia desde la clase persona línea 11 palabra reservada extends
- A Composición pues se debe haber creado el objeto dirección para poder instanciar la clase Cliente
- Constructor que hace referencia a Generalización/Herencia línea 23 utilización de la palabra super

**Figura 75**

Clase cliente con sus atributos y relaciones

```
7  /*
8   * @author djaramillo
9   * Fecha 10 de agosto 2025
10  */
11  public class Cliente extends Persona { //herencia de persona
12    //atributos propios
13    int IdClie;
14    String celular;
15    // atributo crea la relación cliente-direccion
16    Direcciones direccion;
17
18    public Cliente() {
19    }
20
21    public Cliente(int IdClie, String celular, Direcciones direccion,
22                  String Cedula, String Apellidos, String Nombres, String Mail) {
23      super(Cedula, Apellidos, Nombres, Mail);
24      this.IdClie = IdClie;
25      this.celular = celular;
26      this.direccion = direccion;
27    }
}
```

Nota. Jaramillo, D., 2025.

Nótese en la figura 76 la clase persona, para poder ver la herencia/generalización esto se realiza por la generación de código desde el diagrama de clases.

**Figura 76**

Clase persona y su constructor

```
7  /**
8  * Fecha 10 de agosto 2025
9  * @author djaramillo
10 */
11 public class Persona {
12     String Cedula;
13     String Apellidos;
14     String Nombres;
15     String Mail;
16
17     public Persona(String Cedula,
18                 String Apellidos, String Nombres,
19                 String Mail) {
20         this.Cedula = Cedula;
21         this.Apellidos = Apellidos;
22         this.Nombres = Nombres;
23         this.Mail = Mail;
24     }
}
```

Nota. Jaramillo, D., 2025.

Para la explicación la realizaremos desde la capa de presentación donde en este caso esta nuestro formulario para el ingreso de un cliente, figura 77

**Figura 77**

Formulario de capa de presentación para ingresar un cliente

**Datos del Cliente**

Cedula

Nombres  Apellidos

mail

Celular

**Datos de la Dirección**

Código  Calle 1  Calle2

Referencia

Nota. Jaramillo, D., 2025.

Este formulario es muy sencillo. Como se puede notar se deberá ingresar todos los datos del cliente con su dirección para poder ser creado, es decir estamos trabajando con la premisa “una dirección no puede crearse por sí sola sino debe pertenecer a un cliente”.

El código en el botón Crear se muestra en la figura 78

**Figura 78**

Código que se ejecuta en el botón crear pantalla Cliente



```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {  
    // TODO add your handling code here:  
    Cliente objCliente = new Cliente(); //instanciando, creando  
    Direcciones objDireccion = new Direcciones();  
    LCliente objLCliente = new LCliente();  
    //setear atributos del cliente  
    objCliente.setCedula(this.jTextCedula.getText());  
    objCliente.setCedula(this.jTextNombre.getText());  
    objCliente.setApellidos(this.jTextApellidos.getText());  
    objCliente.setMail(this.jTextMail.getText());  
    objCliente.setCelular(this.jTextCelular.getText());  
    //setear atributos de la direccion  
    objDireccion.setCodigo(this.jTextCodigo.getText());  
    objDireccion.setCalle1(this.jTextCalle1.getText());  
    objDireccion.setCalle1(this.jTextCalle1.getText());  
    objDireccion.setReferencia(this.jTextReferencia.getText());  
    //set del objeto direccion en el objeto Direccion  
    objCliente.setDireccion(objDireccion);  
  
    try {  
        objLCliente.BuscarClienteBaseDatosXCedula(objCliente);  
    } catch (ClassNotFoundException ex) {  
        catch (SQLException ex) {  
            if (objCliente.getIdClie() == 0) {  
                try {  
                    // no existe el cliente  
                    objLCliente.AgregarCliente(objCliente);  
                } catch (ClassNotFoundException ex) {  
                    catch (SQLException ex) {  
                }  
            }  
        }  
    }  
}
```

Nota. Jaramillo, D., 2025.

La línea 242, 243 y 244 realizamos la instanciación de los objetos. La línea 246 a la 257 nos permite setear cada uno de los atributos del objeto cliente (que a su vez tiene dentro un objeto de tipo dirección), en este momento se tiene un objeto cliente sin el IdClie puesto que este será autogenerado al momento de insertar en la base de datos.

La linea 260 de la figura 78 a través hace una llamado a una clase de nivel inferior o sea Lógica a través del ObjLCliente para buscar a través del método que nos permitirá hacer una llamada a un objeto de la capa inferior (LOGICA) para ejecutar uno de sus métodos en este caso **buscarCliente(objCliente)**

**Figura 79**

Capa Lógica. Método BuscarIdCliente

```
41     public void BuscarIdCliente(Cliente objCliente)
42         throws ClassNotFoundException, SQLException {
43             ResultSet rs = objBDClientes.BusClieCedula(objCliente);
44             while (rs.next()){
45                 objCliente.setIdClie(rs.getInt(1));
46             }
47             rs.close();
48 }
```

Nota. Jaramillo, D., 2025.

El proceso es similar y en la línea 43 se hará una llamada (BusClieCedula) a la capa inferior que es la **capa de base de datos** a través del objeto objBDClientes.BusClieCedula() para buscar si existe el cliente con ese número de cedula, través de una sentencia sql como se muestra en la figura 80.

**Figura 80**

Capa Base de Datos. BuscarClienteCedula

```
36     public ResultSet BusClieCedula(Cliente objCliente)
37         throws ClassNotFoundException, SQLException {
38             Statement st = BLcon.AbrirConexion().createStatement();
39             String Sentencia = "SELECT * FROM Clientes where cedula = '" +
40                 objCliente.getCedula()+"'";
41             //System.out.println(Sentencia);
42             Resultset rs = st.executeQuery(Sentencia);
43             return rs;
44 }
```

Nota. Jaramillo, D., 2025.

Como se puede mostrar la linea 39 de la figura 80 se compone la sentencia y se ejecuta para que la misma regrese un ResourceSet a la capa Lógica (figura 79 línea 44) y a su vez retorno un 0 si no existe o el idClie con un valor si existiera. En el botón de la pantalla Crear (figura 78 línea 266) se validará que el cliente no exista para llamar al método de la lógica del cliente AgregarCliente() (figura 78 línea 268).

En la figura 81 se muestra como el método AgregarCliente se ejecutarán dos sentencias que llaman a la capa de base de datos para insertarCliente línea 33 e Insertar la dirección de un cliente línea 38.

**Figura 81**

*Paquete lógico del cliente método agregar cliente*

```
31     public void AgregarCliente(Cliente objCliente)
32         throws ClassNotFoundException, SQLException {
33             objBDClientes.InsertarCliente(objCliente);
34             // porque es autoincrementable, se llenara
35             // únicamente setear el idClie en el objeto
36             BuscarIdCliente(objCliente); //leno todos su atributos
37             // inserta la dirección
38             objBDDirecciones.InsertarDirecciones(objCliente);
39 }
```

Nota. Jaramillo, D., 2025.

Pero este método realiza algunas tareas. A través del método insertarCliente se hará la persistencia en la base de datos, figura 82

**Figura 82**

*Paquete de Base de Datos, Clase BDCliente, metodo Insertar Cliente*

```
27     public int InsertarCliente(Cliente objCliente)
28         throws ClassNotFoundException, SQLException {
29             String Sentencia = "Insert into Clientes (cedula,Nombres) values (?,?)";
30             PreparedStatement ps = BLcon.getConnection().prepareStatement(Sentencia);
31             ps.setString(1,objCliente.getCedula());
32             ps.setString(2,objCliente.getNombres());
33             return ps.executeUpdate(); //1 si se inserta -- 0 si no inserta
34 }
```

Nota. Jaramillo, D., 2025.

Posterior a eso se debe buscar el id del Cliente (línea figura 81 línea 36) para posteriormente llamar al método de insertadirección() en la capa de base de datos como se muestra en la siguiente figura.

**Figura 83**

Paquete de Base de Datos, Clase *BDDirecciones*, método *InsertarDireccion*

```
17 public class BDDirecciones {
18     Conexiones BLcon = new Conexiones();
19
20     public int InsertarDirecciones(Cliente objCliente)
21         throws ClassNotFoundException, SQLException {
22         String Sentencia = "Insert into Direcciones (calle1, calle2, idclie) values (?,?,?)";
23         PreparedStatement ps = BLcon.getConnection().prepareStatement(Sentencia);
24         ps.setString(1,objCliente.getDireccion().getCalle1());
25         ps.setString(2,objCliente.getDireccion().getCalle2());
26         ps.setInt(3,objCliente.getIdClie());
27         return ps.executeUpdate(); //1 si se inserta -- 0 si no inserta
28     }
29 }
```

Nota. Jaramillo, D., 2025.

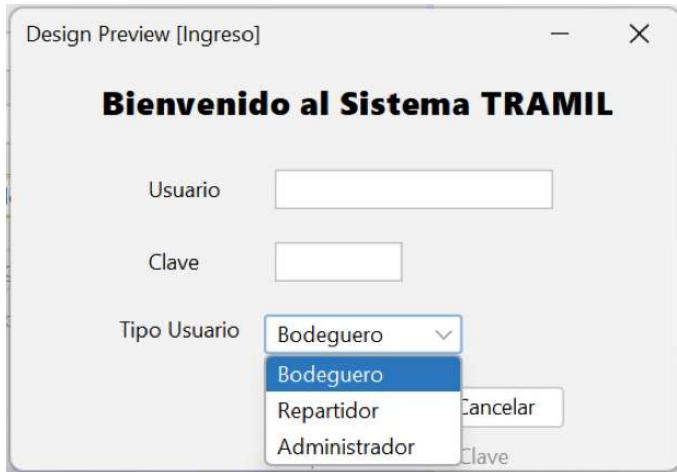
Algo que se debe considerar que él envió de parámetros siempre se cumple trabajando con objetos, para asegurar la propiedad de encapsulamiento.

Este es el trabajo que se realiza por capas así si existe algún problema por ejemplo que no está grabando bien deberíamos empezar revisando la capa de base de datos.

Algo adicional que vamos a considerar dentro de la explicación de esta etapa es a través de la figura 84.

**Figura 84**

Pantalla de Inicio del Sistema TRAMIL



Nota. Jaramillo, D., 2025.

Encontramos ahí un comboBox donde hace referencia al tipo de usuario... ¿de dónde tomamos esos datos? Analícelo.... esos tipos de usuarios se relacionan directamente con los **actores de nuestro diagrama de casos de uso**, entonces el proceso de modelado va tomando forma y sentido a todo lo que se ha realizado.

Un tema adicional que se hace referencia es a las malas prácticas de programación, a veces copiamos lo primero que encontramos. Analicemos la siguiente figura, si hacen el mismo proceso es decir buscar un cliente por el número de cedula.

## Figura 85

Método que se ejecuta a nivel del paquete de Base de Datos (1)

```
public ResultSet BusClieCedula(Cliente objCliente)
    throws ClassNotFoundException, SQLException {
    Statement st = BLcon.AbrirConexion().createStatement();
    String Sentencia = "SELECT * FROM Clientes where cedula = '" +
        objCliente.getCedula()+"'";
    //System.out.println(Sentencia);
    ResultSet rs = st.executeQuery(Sentencia);
    return rs;
}
```

Nota. Jaramillo, D., 2025.

A continuación, otro ejemplo de mala práctica.

## Figura 86

Método que se ejecuta a nivel del paquete de Base de Datos (1)

```
public int BuscarCliente(Cliente objCliente)
    throws ClassNotFoundException, SQLException {
    String Sentencia = "SELECT * FROM Clientes where cedula =?";
    PreparedStatement ps = BLcon.getConnection().prepareStatement(Sentencia);
    ps.setString(1,objCliente.getCedula());
    return ps.executeUpdate(); //1 si se inserta -- 0 si no inserta
}
```

Nota. Jaramillo, D., 2025.

La diferencia entre estas dos es que la primera se hace una concatenación de la sentencia sql por lo que se permitirá ingresar valores adicionales en la pantalla que nos pueden permitir una falla de seguridad mediante una Inyección SQL, mientras que la figura 86 no permite esto, la utilización de PreparedSatatment nos permite bloquear una posible vulnerabilidad a nuestro sistema.

A continuación, abordaremos el estudio de la infraestructura, la seguridad y la persistencia en el ámbito del software.

## Infraestructura, seguridad, persistencia

### 7.4. Infraestructura, seguridad y persistencia

En esta sección se abordarán los aspectos fundamentales que permiten garantizar el funcionamiento, la seguridad y la continuidad de los sistemas de información en un entorno real de implementación.

#### 7.4.1. La infraestructura

La implementación de un *software* en una organización no es simplemente el acto de instalar un programa en un conjunto de equipos de cómputo. Por el contrario, constituye un proceso complejo que requiere de una planeación cuidadosa, una gestión estratégica de recursos y, sobre todo, una infraestructura tecnológica adecuada que garantice la correcta operación del sistema. La infraestructura tecnológica es el conjunto de *hardware*, *software*, redes, plataformas de almacenamiento, medidas de seguridad y servicios asociados que permiten que una aplicación cumpla con los objetivos para los cuales fue diseñada.

En la actualidad, donde los sistemas de información se han convertido en el núcleo operativo de empresas, universidades, instituciones públicas y organizaciones sin fines de lucro, el análisis de la infraestructura adquiere un papel determinante. La calidad del *software* por sí sola no asegura el éxito de un proyecto; es indispensable contar con servidores robustos, conectividad suficiente, seguridad informática adecuada, respaldo de datos, escalabilidad y un equipo humano capacitado que administre la plataforma tecnológica.

Las principales consideraciones de la infraestructura tecnológica que deben tenerse en cuenta en el momento de implementar un *software*. Para ello, se desarrollarán secciones enfocadas en el *hardware*, el *software* de base, las redes y comunicaciones, la seguridad, la escalabilidad, la virtualización, la nube, la capacitación del talento humano y la gestión del cambio.

#### La infraestructura tecnológica como base de la implementación



Cuando se habla de infraestructura tecnológica, se hace referencia a todos los recursos técnicos que sostienen el ciclo de vida de un *software*. Desde los servidores físicos que alojan el sistema, hasta los *routers* que garantizan la conectividad, pasando por las plataformas de respaldo, los *firewalls* y los sistemas de monitoreo. Implementar un *software* sin una infraestructura adecuada sería como intentar construir un edificio sobre arena: tarde o temprano la solución colapsará.

Las organizaciones deben evaluar previamente con qué recursos cuentan y cuáles requieren adquirir o actualizar. Este análisis no solo involucra aspectos técnicos, sino también financieros y de gestión, pues la inversión en infraestructura suele representar una parte significativa del presupuesto de un proyecto de software.

### **Hardware y capacidad de procesamiento**

El *hardware* constituye el primer componente de la infraestructura tecnológica. Para implementar un *software*, se requiere definir si este será ejecutado en estaciones de trabajo, servidores locales, data centers propios o en entornos en la nube.

- **Servidores:** la selección del servidor es fundamental. Se deben considerar procesadores de alto rendimiento, memoria RAM suficiente, discos duros de estado sólido y redundancia para garantizar la disponibilidad del sistema.
- **Almacenamiento:** el *software* moderno maneja grandes volúmenes de datos. Por ello, es necesario contar con soluciones de almacenamiento robustas como SAN (*Storage Area Network*) o NAS (*Network Attached Storage*).
- **Dispositivos de usuario:** en el caso de aplicaciones cliente-servidor, las computadoras de los usuarios también deben cumplir con requisitos mínimos de *hardware*, evitando que la experiencia del usuario se vea afectada por equipos obsoletos.

### **Software de base y plataformas de soporte**

La infraestructura tecnológica también incluye el software de base, como sistemas operativos, bases de datos, servidores web y *middleware*. Cada uno de estos componentes deben seleccionarse de acuerdo con la compatibilidad y requerimientos del software a implementar.

- **Sistemas operativos:** dependiendo de la aplicación, puede ser necesario un sistema operativo específico (Windows Server, Linux, Unix). La decisión debe basarse en estabilidad, seguridad y soporte.
- **Bases de datos:** un sistema robusto debe contar con un motor de base de datos confiable como Oracle, SQL Server, MySQL o PostgreSQL. La elección dependerá del volumen de datos y de las características del software.
- **Servidores de aplicaciones y middleware:** algunas soluciones requieren servidores de aplicaciones (Tomcat, WebSphere, JBoss) que gestionen la comunicación entre la lógica del negocio y el usuario.

La implementación de un software depende en gran medida de la calidad de la conectividad. Una infraestructura de red deficiente puede provocar retrasos, caídas en el servicio y pérdida de confianza de los usuarios.

- **Velocidad y ancho de banda:** se debe garantizar que la red soporte el número de usuarios concurrentes.
- **Topología de red:** la arquitectura de red debe estar diseñada para evitar cuellos de botella.
- **Redundancia:** para sistemas críticos, es recomendable contar con enlaces redundantes que permitan mantener la operación en caso de fallos.
- **Seguridad en la comunicación:** el uso de VPN, firewalls y protocolos de encriptación es indispensable para proteger la información en tránsito.

#### 7.4.2. Seguridad

La seguridad es uno de los pilares de la infraestructura tecnológica. Un software puede funcionar correctamente, pero si no se protege adecuadamente la información que maneja, el proyecto estará condenado al fracaso.

- **Firewalls y sistemas de detección de intrusos:** protegen la infraestructura contra accesos no autorizados.
- **Políticas de autenticación y autorización:** implementación de contraseñas robustas, autenticación multifactor y gestión de roles.
- **Respaldo y recuperación:** copias de seguridad periódicas y Planes de Recuperación ante Desastres (DRP).
- **Cumplimiento legal:** dependiendo del país, se deben cumplir normativas como GDPR (Europa) o leyes locales de protección de datos.

#### 7.4.3. Persistencia

La persistencia se puede trabajar de dos maneras:

1. La primera realizando todo manualmente es decir que el programador define las clases con sus atributos y métodos, realiza la creación de la tabla a nivel de la base de datos e implementa las librerías de forma directa para que se pueda realizar Así mismo debe realizar las conexiones y configuraciones para el trabajo de la base de datos, figura 87.

**Figura 87**

Trabajo específico de BD manualmente

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class Conexiones {
    //Conectarse a la BDD
    public Connection con;
    public Connection getConnection () throws ClassNotFoundException, SQLException {
        String driver = "com.mysql.cj.jdbc.Driver";
        String url = "jdbc:mysql://localhost:3306/semanal3";
        Class.forName(driver);
        return DriverManager.getConnection(url,"root","12345678");
    }
    public Connection AbrirConexion() throws ClassNotFoundException, SQLException{
        con = getConnection();
        return con;
    }
    public void CerrarConexion() throws SQLException{
        con.close();
    }
}
```

Nota. Jaramillo, D., 2025.

2. La segunda a partir de herramientas que los diferentes framework lo permiten por ejemplo en Netbeans tenemos JPA donde se puede trabajar de los 2 caminos permitidos es decir A partir del modelo es decir cada clase se puede crear en la base de datos la tabla que permitirá almacenar la información es decir tener la persistencia. O la segunda forma si ya se tiene una tabla en la base de datos la herramienta nos permite cambiar las configuraciones de nuestras clases para que se pueda realizar es decir implementar la clase serializable en java.

**Figura 88**

JPA para trabajo de Base de Datos

Choose File Type

Project: TRAMIL-DB-01

Filter:

Categories:

- PlantUML
- Java
- Swing GUI Forms
- JavaBeans Objects
- AWT GUI Forms
- Unit Tests
- Micronaut
- Persistence
- Groovy
- Web Services

File Types:

- Entity Class
- Entity Classes from Database
- JPA Controller Classes from Entity Classes
- Persistence Unit
- DB Scripts from Entity Classes
- Database Schema

Description:

Creates an empty Java Persistence API entity class.

Nota. Jaramillo, D., 2025.

[3] El constructor refleja una relación <<un cliente tiene una dirección>> 1 a 1 porque solo tenemos una dirección. Lo correcto, es decir, el cliente tiene varias direcciones, para lo cual debemos colocar un array list de clientes.



### Actividades de aprendizaje recomendadas

Para afianzar sus conocimientos sobre las temáticas abordadas en esta unidad, desarrolle la autoevaluación y los ejercicios planteados. En caso de dudas, no olvide acudir a su tutor.

1. Desarrolle los siguientes ejercicios:

- Considere un sistema que incluya un servidor web y 2 servidores de base de datos. Ambos servidores de base de datos son idénticos: el primero actúa como servidor principal y el segundo como servidor redundante en caso de que el servidor 1 falle. Los

usuarios utilizan un navegador *web* para acceder a los datos mediante el servidor *web*. Además, tienen la opción de usar un cliente propietario que accede a la base de datos directamente. Dibuje un diagrama de despliegue en UML que represente el mapeo de *hardware* y *software*.



- b. Está diseñando las políticas de control de acceso para una tienda minorista basada en la *web*. Los clientes acceden a la tienda a través de la *web*, navegan por la información del producto, ingresan su dirección e información de pago y compran productos. Los proveedores pueden agregar nuevos productos, actualizar información del producto y recibir pedidos. El dueño de la tienda establece los precios minoristas, hace ofertas personalizadas a los clientes en función de sus perfiles de compra y ofrece servicios de *marketing*. Usted debe considerar tres actores: administrador de la tienda, proveedor y cliente. Diseñar una política de control de acceso para los tres actores. Los clientes pueden ser creados a través de la *web*, mientras que los proveedores son creados por el administrador de la tienda.
- c. Pruebe crear una pantalla para el caso de uso RegistrarPaquete. Puede exponerlo o preguntar con su tutor para ver cómo realiza la implementación. En los materiales podrá encontrar el proyecto utilizado en esta unidad.



2. Desarrolle la siguiente autoevaluación de la unidad 7 para comprobar sus conocimientos.



### Autoevaluación 7

1. Al establecer los objetivos de diseño en un proyecto de *software*, ¿con qué tipo de requisitos se encuentran principalmente vinculados?
  - a. Requerimientos funcionales.
  - b. Requerimientos no funcionales.
  - c. Expectativas de usuario.
  - d. Documentación técnica.

2. Al representar el contexto de un sistema, ¿qué aspecto describe mejor su alcance?

- a. La interacción con sistemas externos relacionados.
- b. Los límites internos y condicionantes del sistema.
- c. Las restricciones establecidas por el cliente.
- d. La arquitectura física del software.

3. El propósito central del diseño arquitectónico de un software es:

- a. Determinar los casos de uso a implementar.
- b. Definir las clases que satisfacen las necesidades del usuario.
- c. Identificar los subsistemas que integrarán la solución.
- d. Especificar el lenguaje de programación.

4. Dentro de los modelos de diseño, ¿a qué categoría pertenecen los que muestran cómo varían los objetos individuales frente a eventos?

- a. Modelos de subsistemas.
- b. Modelos de secuencia.
- c. Modelos de máquina de estados.
- d. Modelos de colaboración.

5. ¿Qué tipo de diagrama UML se utiliza para representar los dispositivos físicos de instalación y las conexiones entre ellos?

- a. Diagrama de componentes.
- b. Diagrama de despliegue (implantación).
- c. Diagrama de actividades.
- d. Diagrama de casos de uso.

6. ¿Cuál de los siguientes objetos es el candidato más adecuado para almacenamiento persistente?

- a. Objetos - entidad.
- b. Objetos frontera.
- c. Objetos de control.



- d. Objetos temporales.
7. Una desventaja de las bases de datos Orientadas a Objetos al gestionar persistencia es que:
- a. Representan tanto objetos como relaciones.
  - b. Su implementación suele ser compleja.
  - c. Presentan bajo rendimiento.
  - d. No permiten interoperar con SQL.
8. ¿Cuál es la principal ventaja de aplicar encriptación a la información sensible?
- a. Requiere desencriptarse para poder leerse.
  - b. Se genera rápidamente el código encriptado.
  - c. Si es robada, resulta ininteligible para terceros.
  - d. Reduce el espacio de almacenamiento.
9. ¿Qué aportan los patrones de diseño en el proceso de desarrollo de software?
- a. Brindan soluciones específicas para problemas amplios.
  - b. Proveen soluciones generales a problemas recurrentes.
  - c. Ayudan a reconocer componentes de software útiles en diseño.
  - d. Automatizan el proceso de codificación.
10. ¿Qué ventaja ofrecen las herramientas de gestión de configuración al trabajo colaborativo?
- a. Establecen estándares de codificación.
  - b. Guardan la información del proyecto en un repositorio central.
  - c. Previenen conflictos en el uso simultáneo del código.
  - d. Reducen el tiempo de compilación.

[Ir al solucionario](#)

## Resultado de aprendizaje 2 y 3:

- Conoce e identifica las fases del ciclo de vida de desarrollo de sistemas.
- Identifica el ciclo de vida a utilizar en el desarrollo de un proyecto de software.

Para alcanzar estos resultados, se explorarán las fases del ciclo de vida del desarrollo de sistemas junto con la identificación del ciclo más adecuado para cada proyecto. Se profundizará en los conceptos relacionados con las pruebas de software, incluyendo pruebas de desarrollo, unitarias e integración, lo que permitirá comprender su función dentro del ciclo de vida y asegurar la calidad del producto final.

### Contenidos, recursos y actividades de aprendizaje recomendadas

Recuerde revisar de manera paralela los contenidos con las actividades de aprendizaje recomendadas y actividades de aprendizaje evaluadas.



### Semana 15

Para concluir el estudio de la presente asignatura, vamos a estudiar lo que son y en qué consisten las pruebas del software, las cuales permitirán cerrar el ciclo de desarrollo incorporando procesos de pruebas del software para garantizar que el mismo cumple con las especificaciones funcionales y no funcionales.

### Unidad 8. Pruebas del software

En la ingeniería de software, para desarrollar productos de calidad que cumplan con las especificaciones, no basta el uso de una buena metodología de desarrollo o un diseño arquitectónico robusto y eficiente. Es necesario realizar pruebas de diferente tipo y en diferentes etapas del proceso. En los modelos ágiles se habla de una cultura de calidad, sin embargo, eso no significa solo hacer las cosas bien, sino probarlas más a menudo y la

diferencia fundamental entre las metodologías ágiles y las tradicionales es que en las primeras todo el mundo es responsable de la calidad y en las segundas hay un equipo encargado del aseguramiento de la calidad que son los especialistas en realizar las pruebas y reportar los resultados a los desarrolladores para que realicen las correcciones necesarias.

Independientemente de cuál sea el enfoque, en esta unidad vamos a estudiar los diferentes tipos de pruebas que se pueden aplicar al software.



Para comenzar, le invito a realizar el estudio de la introducción a la unidad 8 de la guía didáctica. Siga las instrucciones dadas.

Un aspecto a resaltar y que en la gran mayoría de proyectos sucede es el planteamiento de Dijkstra que menciona que “las pruebas pueden mostrar la presencia de errores, más no su ausencia”, con ello nos hace entender que por exhaustivo que sea el proceso de pruebas, es posible que se encuentren errores aun cuando las pruebas realizadas no los reporten, y muchas de las veces estos errores se producen cuando el producto está en manos del usuario, en un proceso tradicional se trata de un encuentro tardío y que puede resultar costoso al momento de corregir o incluso se puede enfrentar demandas, en el caso de las metodologías ágiles, la probabilidad de encontrarlo de manera inmediata una vez liberado un incremento es muy alta y, por tanto, la corrección impactará mucho menos que en la tradicional.

## 8.1. Conceptos relacionados con las pruebas

Ahora revisemos algunos conceptos y terminología común de las pruebas.

- **Prueba de software:** es todo proceso orientado a comprobar la calidad del software mediante la identificación de fallos en el mismo. La prueba implica necesariamente la ejecución del software.
- **Caso de prueba:** es un conjunto de entradas, condiciones de ejecución y resultados esperados, que han sido desarrollados para un objetivo

particular como, por ejemplo, ejercitarse en un camino concreto de un programa o verificar el cumplimiento de un determinado requisito.

- **Prueba exhaustiva/completa:** prueba ideal que proporciona la seguridad de que se han comprobado todas y cada una de las posibles causas de fallo.
- **Fallo:** es un efecto indeseado observado en las funciones o prestaciones desempeñadas por un software.
- **Error (defecto):** es una imperfección en el software que provoca un funcionamiento incorrecto del mismo.
- **Probar:** proceso de mostrar la presencia de un error en el software. Todos los conceptos estudiados definen una serie de parámetros que se usarán durante los procesos de prueba del software; por tanto, es conveniente que les dé una nueva mirada antes de pasar al siguiente apartado.
- **Depurar:** descubrir en qué lugar exacto se encuentra un error y modificar el software para eliminar dicho error.
- **Oráculo:** cualquier agente (humano o no) capaz de decidir si un programa se comporta correctamente durante una prueba y, por tanto, capaz de dictaminar si la prueba se ha superado o no.
- **Trazabilidad:** capacidad para interrelacionar dos o más entidades, inequívocamente identificables, dentro de una cadena de sucesos cronológicamente ordenada.

## 8.2. Pruebas de desarrollo

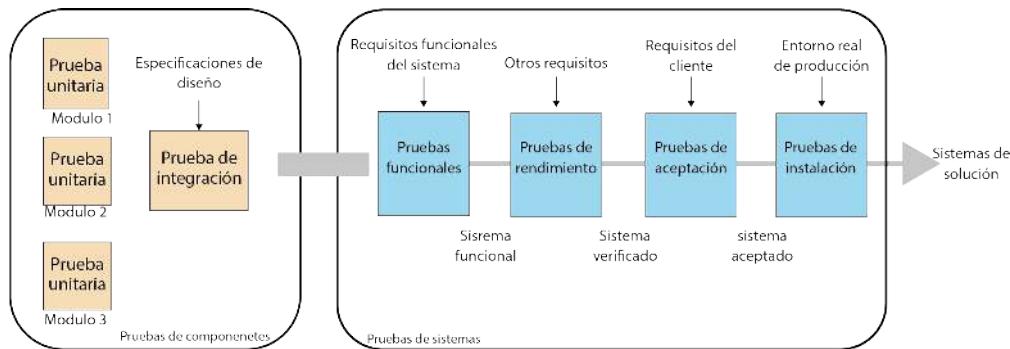
Habíamos indicado que las pruebas se realizan a diferentes niveles, unas las realizan los mismos desarrolladores o programadores y otras las ejecutan los especialistas en pruebas, también conocidos como testers.

Las pruebas de desarrollo son las que se realizan por parte de los desarrolladores y corresponden a 4 categorías.

Pero antes de estudiar a detalle en qué consisten las pruebas, en la figura 89, los autores Sánchez, Sicilia, y Rodríguez (2012), proponen un esquema sobre los niveles a los que se aplican las pruebas.

**Figura 89**

Niveles a los se aplican las pruebas de software



Nota. Jaramillo, D., 2025.

Las pruebas de desarrollo, por tanto, son las siguientes:

#### a. Pruebas de unidad

Aplicadas a un módulo individual a ver si cumple con la especificación, pero también es necesario determinar si no existen errores en el código por alguna bifurcación no controlada, por ello recomienda el uso de herramientas de prueba automatizadas que permitan probar todos los casos posibles. Al final de la unidad tendrá algunas referencias sobre herramientas para automatización de pruebas.

Tienen como propósito probar un módulo concreto con el fin de determinar de forma aislada si el mismo cumple con la especificación con la que fue creado, este mismo proceso se aplica a todos los elementos individuales y con ello se intenta asegurar que todos cumplen con su especificación, sin pensar en la colaboración con otros.



El desarrollo de este tipo de pruebas, puede tomar mucho tiempo y resultar costoso, por lo que de ser posible es recomendable utilizar herramientas de software que permitan automatizar este proceso.

Para poder realizar estas pruebas, es evidente que necesitamos definir los casos de prueba, para ello se proponen dos estrategias denominada pruebas de partición y las pruebas basadas en lineamientos, estos escenarios de prueba deben tener identificadas las variables de entrada y las salidas correctas, además, para los datos de entrada, se requiere datos de la operación normal y datos extremos para verificar el comportamiento del componente tanto en condiciones normales como en condiciones extremas de operación.

### b. Pruebas de componentes

En realidad, son pruebas aplicadas a subsistemas, son un poco más amplias que las anteriores y buscan la integración entre clases y de interfaces. Debe tener en cuenta la estrategia incremental para la integración, esto suele ser muy costoso de realizar.

El siguiente nivel de pruebas, son las pruebas de componentes, los componentes en este caso pueden ser subsistemas que se forman al combinar varias clases o conforme se haya establecido en el diseño.

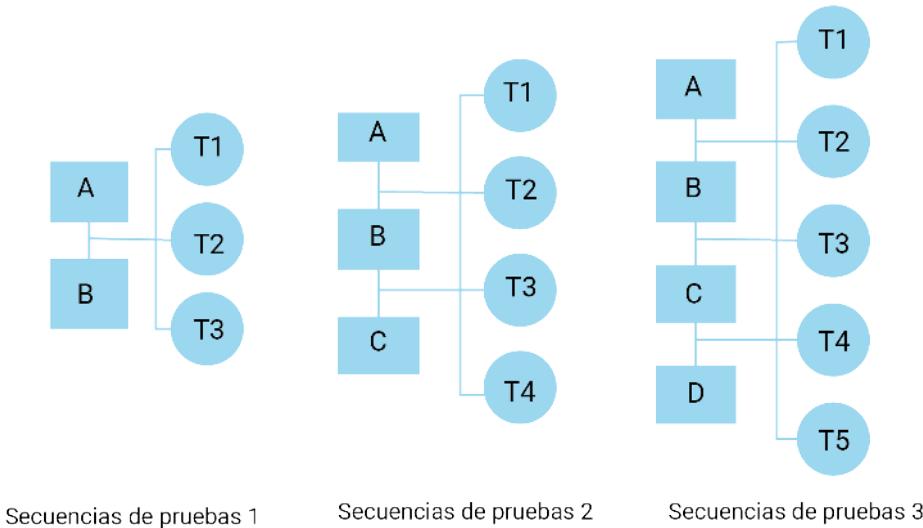
Una característica particular de este tipo de pruebas y la diferencia de las pruebas de unidad, es que se enfocan en probar las interfaces de las piezas individuales, para ello utilizan varias estrategias que permiten determinar si los componentes individuales colaboran correctamente entre sí.

Las interfaces pueden ser de diferente tipo, y por tanto es necesario diseñar pruebas para cada una de ellos, por ejemplo, si estamos verificando una interfaz de parámetro, será necesario determinar el funcionamiento correcto de la interfaz colocando casos de prueba en cada uno de ellos, verificando el número de parámetros, los tipos de datos que pasan y si en general el componente los procesa para obtener resultados correctos.

La recomendación para realizar este tipo de pruebas, es identificar el tipo de interfaz y por cada una de ellas establecer escenarios de prueba válidos.

A este tipo de pruebas también se las conoce como pruebas de integración, y es necesario recalcar que son pruebas incrementales a lo largo del ciclo de desarrollo del producto, sobre este tipo de pruebas en: <https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/Testing/Integration.html> del cual se muestra la estrategia incremental en la figura 36. Es decir, por cada componente que se agrega, es necesario realizar todas las pruebas, para determinar si el sistema se comporta correctamente con la inclusión del nuevo componente. En la figura 90 los recuadros con las letras son los componentes y los círculos representan los Test que se hacen en cada secuencia de pruebas.

**Figura 90**  
*Estrategia Incremental para pruebas de integración*



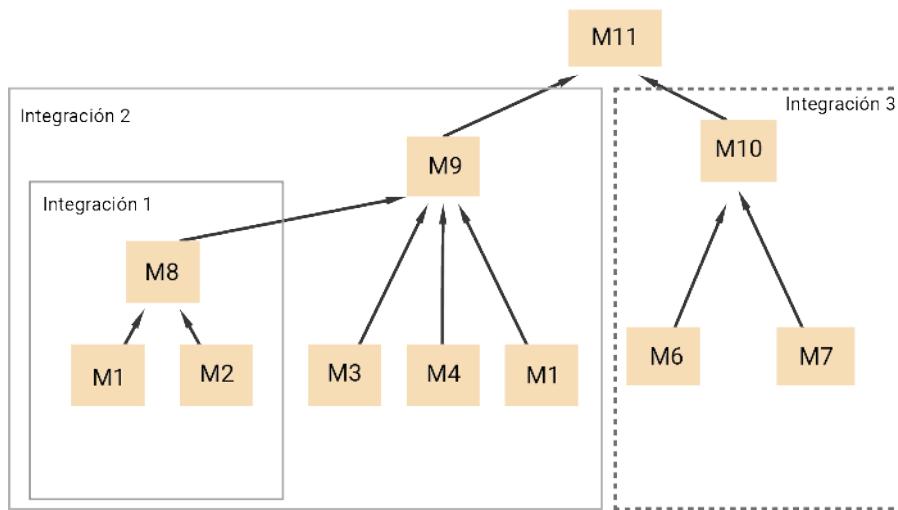
Nota. Jaramillo, D., 2025.

Sánchez, Sicilia, y Rodríguez (2012), añaden que estas pruebas se pueden realizar de tres maneras diferentes.

1. Integración **ascendente**, que consiste en incluir primero los componentes de infraestructura y progresivamente ir añadiendo los componentes funcionales, lo cual significa que se empieza por probar en una primera pasada los componentes de menor nivel, luego los que son invocados por los anteriores y así hasta llegar al nivel más alto. Esta estrategia se puede apreciar en la figura 91.

**Figura 91**

*Estrategia Incremental ascendente para pruebas de integración*

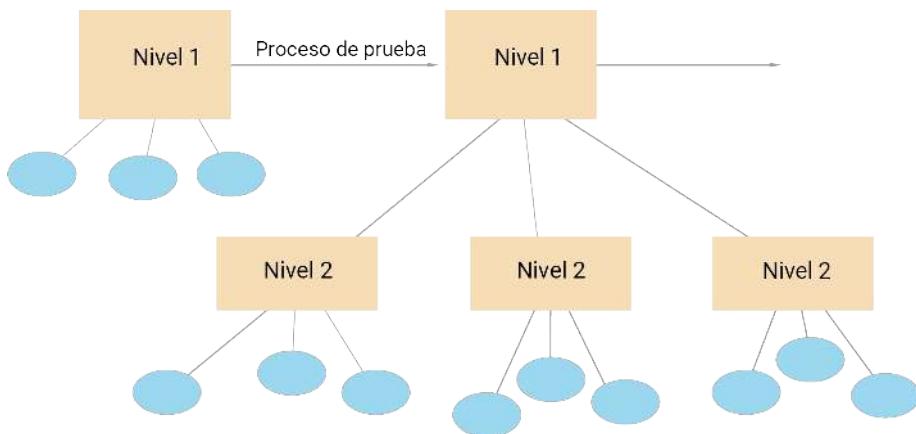


Nota. Jaramillo, D., 2025.

2. Integración **descendente**, en la cual se desarrolla un esqueleto del sistema al que se añaden componentes de manera progresiva, para ello se empieza probando en primera instancia el componente de más alto nivel, luego los que dicho componente utiliza y así hasta llegar hasta los de más bajo nivel, como se aprecia en la figura 92.

**Figura 92**

*Estrategia Incremental descendente para pruebas de Integración*



Nota. Jaramillo, D., 2025.

3. **Estrategia combinada**, en la cual se combinan las dos estrategias, el sistema se presente como un sándwich con 3 capas, la capa central es la que se desea probar, y mediante enfoque ascendente para probar la capa central, asumiendo que la inferior ya ha sido implementada, y se desciende a la capa de prueba desde la capa superior.

### c. Pruebas de sistema

estas pruebas están orientadas a determinar el funcionamiento correcto del sistema y se centran en atributos de calidad como la funcionalidad, el rendimiento. Además, incluyen pruebas de aceptación del cliente y de instalación para asegurarse que el cliente no tendrá inconvenientes.

El siguiente nivel de las pruebas, son las pruebas del sistema. En este punto la mayoría de fallos debieron haberse identificado y por tanto es necesario comprobar si el sistema completo satisface las especificaciones antes de pasar a producción. Sánchez, Sicilia, y Rodríguez (2012), sostienen que estas pruebas se enfocan en aspectos como la seguridad, la velocidad, la exactitud, la fiabilidad, interconexiones

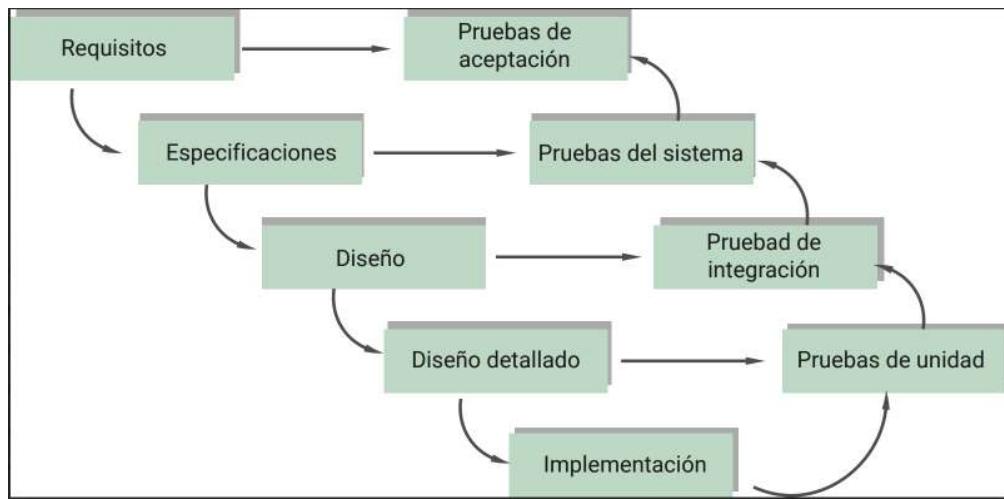
externas con otras aplicaciones, utilidades, dispositivos hardware o con el sistema operativo, y que las pruebas deben enfocarse desde las siguientes perspectivas:

1. **Pruebas de funcionalidad y operativa:** Buscan determinar si se cumple con las especificaciones funcionales dadas, son pruebas de caja negra.
2. **Pruebas de rendimiento:** Comprueban el cumplimiento de los requisitos no funcionales, por tanto, incluyen pruebas de desgaste que sirven para determinar el comportamiento del sistema cuando se lo pone a operar en sus límites; pruebas de volumen, que se realizan sobrecargando al sistema con datos; pruebas de configuración, que sirven para determinar el comportamiento en diferentes plataformas, además se realizan pruebas de compatibilidad, de regresión, de seguridad, de recuperación.
3. **Pruebas de aceptación:** Una vez que se ha determinado que el sistema cumple con todas las especificaciones, estas pruebas se realizan con los usuarios, quienes establecen los casos de prueba. Estas pruebas pueden ser de benchmarking o de comparación con valores de referencia, pruebas piloto en cuyo caso, se realiza una instalación para en un entorno controlado para determinar el comportamiento del sistema y proyectarlo al entorno real, además de las pruebas alfa, beta y pruebas en paralelo.
4. **Pruebas de instalación:** Consisten en realizar las pruebas necesarias para asegurar que el sistema se instala correctamente, se debe probar todas las opciones de instalación y desinstalación.

Una vez que se ha revisado todo el panorama del proceso y tipos de pruebas, en la figura 93, se expone los diferentes niveles a los que se realizan y la relación de las mismas con las etapas del ciclo de vida.

**Figura 93**

Niveles a los que se realizan las pruebas



Nota. Jaramillo, D., 2025.

Continuemos con el aprendizaje mediante la revisión de los tipos de pruebas.

## Tipos de pruebas

### 8.3. Tipos de pruebas: unitarias e Integración

#### Actividades y gestión de las pruebas

El proceso de pruebas suele ser muy formal precisamente para asegurar que todo funciona correctamente, sin embargo, debemos tener en cuenta que los productos de software son extremadamente complejos y para efectuar las pruebas es necesario contar con instrumentos como métricas que nos permitan establecer el nivel que se está logrando.

Las pruebas de software son una actividad necesaria que implica en muchos casos trabajo de equipos paralelos y el uso de técnicas y herramientas, por este motivo es necesario organizar las actividades de pruebas para monitorear el avance y los resultados obtenidos, por tanto, no se trata de actividades

marginales, sino que a medida que se incrementa la importancia, el tamaño y la criticidad del producto, se torna cada vez más importante y se necesitan dos enfoques:

- 1. Orientación individual hacia la calidad:** en el cual todo el equipo de desarrollo está implicado y realizan tanto el desarrollo como las pruebas unitarias correspondientes.
- 2. Equipo de aseguramiento de calidad:** en el cual se conforma un equipo independiente del de desarrollo, y son quienes se encargan de hacer pruebas a distintos niveles, principalmente las pruebas del sistema.

Es necesario medir el avance de las pruebas, y la estabilidad que se consigue con los productos o componentes completados, por ello es necesario analizar las métricas asociadas al proceso de pruebas. Las métricas son útiles para realizar estimaciones, proporcionan información para controlar el propio proceso y facilitan la identificación de riesgos asociados al producto, entre otros beneficios.

A continuación, se describen las métricas obtenidas durante el proceso de pruebas, y las que miden el mismo proceso de pruebas.

### **8.3.1. Medidas durante las pruebas**

Son medidas útiles para la planificación y diseño de pruebas de programas, permiten clasificar analizar los errores, establecer la densidad de los fallos o también determinar si se debe continuar o no con las pruebas.

En la siguiente infografía, se resumen las medidas utilizadas en esta categoría.

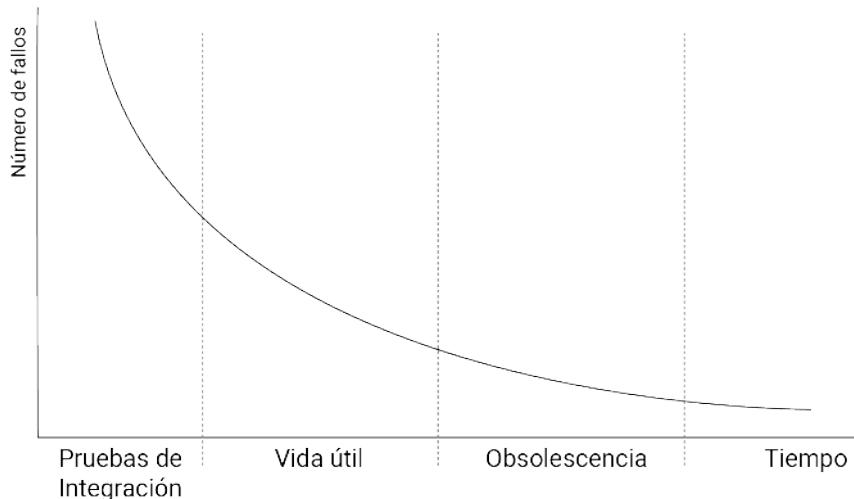
#### [Medidas utilizadas en pruebas](#)

Tras el análisis de la infografía, puede sostenerse que las métricas de prueba permiten evaluar confiabilidad, identificar tendencias y determinar con precisión la efectividad y finalización del proceso de validación.

En general, la intención de estas mediciones es determinar si el producto evoluciona hacia una versión más estable. En la figura 94, se puede apreciar cómo se debe ver la tendencia en las pruebas realizadas.

**Figura 94**

*Evolución de la tasa de fallos en sistemas de software*



Nota. Jaramillo, D., 2025.

### 8.3.2. Evaluación de las pruebas realizadas

Este tipo de medidas son útiles para el propio proceso de desarrollo en curso y para procesos futuros:

- Evaluación de la cobertura, sirven para determinar el nivel de compleción de las pruebas mediante comparación de los elementos cubiertos con el número total.
- Estimación de errores, la inclusión de errores ficticios, sirve para comprobar cuántos de estos errores son detectados, con ello se determina cuántos errores genuinos no se detectan.
- Medición de la efectividad, la relación entre el número de mutantes (modificaciones de código para determinar si se encuentra el fallo en las pruebas), en especial el número de mutantes identificados y el total de mutantes generados, se usa como medida de efectividad.

- Estudio de adecuación de las pruebas, Mide la efectividad relativa de las diferentes técnicas de prueba, sirve para determinar cuál es la técnica más adecuada en determinadas situaciones.



### 8.3.3. El proceso de prueba

Para integrar todos los aspectos que implica la realización de las pruebas, es necesario elaborar un Plan de Gestión de Pruebas, el cual se puede desarrollar considerando los siguientes aspectos:

- **Finalidad de la guía del proceso:** Se debe explicar el propósito de la realización de las pruebas, ya que en función de este propósito se deciden los escenarios de pruebas.
- **Gestión del proceso de prueba:** Se establece quienes serán las personas responsables, cuando se aplicarán las pruebas, las herramientas que se usarán, normas aplicables, medidas que se usarán como referencia, todo enmarcado en el ciclo de vida del software.
- **Documentación y productos de las pruebas:** Todo el proceso de pruebas debe ser formal, se puede usar estándares como el IEEE829 cuyo formato se coloca en el [anexo 3](#).
- **Estructura del equipo de pruebas:** Se refiere al grupo de personas que se harán cargo de las pruebas, pueden ser miembros internos, no necesariamente parte del equipo de desarrollo, el contar con un equipo interno, puede facilitar las cosas debido a que son personas que ya se conocen y conocen el sistema que se está desarrollando, no obstante, también pueden ser recomendable trabajar con equipos externos, que pueden aportar una visión independiente y sin prejuicios sobre lo que se está probando, aunque ello redunde en costos más elevados, de hecho alguno autores sugieren que los equipos externos actuarán sin compromisos con el equipo de desarrollo. Cabe acotar aquí la decisión final sobre cómo se constituirá el equipo depende también del presupuesto y de las metodologías que se apliquen tanto para el desarrollo como para las pruebas.
- **Medidas del proceso:** El uso de medidas o métricas del proceso siempre ayuda a mejorar el proceso de pruebas, también para obtener información



histórica que pueda utilizarse para mejorar proyectos futuros, entre otras cosas las medidas deben ser útiles para determinar si las actividades de prueba han sido eficaces, identificar las más eficientes, cuáles técnicas nos han provisto de mejores resultados.

- **Fin del proceso de pruebas:** El fin del proceso de pruebas está supeditado a factores como el nivel de calidad deseado, los costos y los niveles de riesgo que se podría identificar en determinado momento. Además, se debe tener en cuenta que el número de defectos identificados tiende a disminuir con el tiempo, como se muestra en la figura 43, además los equipos de pruebas tienen suficiente experiencia como para realizar procesos de prueba más efectivos.

Se ha concluido la unidad, y también el contenido de la asignatura, con toda seguridad, el estudio de estos temas le será de mucha utilidad, sobre todo si usted trabaja o piensa trabajar con áreas de desarrollo de software, revise nuevamente los temas indicados, busque la bibliografía recomendada y consulte a su tutor cualquier duda que tenga respecto del contenido del texto, la guía o cualquier otro material.



### Actividades de aprendizaje recomendadas

Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. Le invito a revisar el software de automatización de pruebas visitando el sitio Software Testing Help, allí se listan y se comentan un conjunto de herramientas open source para automatización de pruebas.
2. Desarrolle la autoevaluación de la unidad 8. Recuerde que debe desarrollarla por completo y tenga en cuenta que las partes que no resolvió correctamente, son las que debe repasar y, en caso de dudas, puede acudir a su tutor.



## Autoevaluación 8

1. ¿Cuál es la diferencia fundamental entre verificación y validación en pruebas de software?
  - a. Son sinónimos, no existe diferencia.
  - b. La verificación determina si el producto cumple con lo que espera el cliente y la validación si se construyó correctamente.
  - c. La verificación comprueba que el software se construyó de manera correcta y la validación confirma que satisface las expectativas del cliente.
  - d. La verificación mide calidad técnica y la validación mide velocidad de ejecución.
2. Las inspecciones podrían detectar más del 90 % de los defectos, pero en la práctica no alcanzan tal eficacia. ¿Por qué sucede esto?
  - a. No consideran las interfaces entre componentes.
  - b. El código puede resultar poco claro para quienes lo revisan.
  - c. Requieren herramientas automáticas costosas.
  - d. El tiempo de ejecución de las inspecciones es limitado.
3. Si durante las primeras pruebas de un proyecto se detectan gran cantidad de errores en los componentes iniciales, ¿qué significa esto?
  - a. Representa una curva normal de detección de errores.
  - b. Indica un posible riesgo de fallos en los siguientes componentes.
  - c. Muestra que el equipo de pruebas es muy experimentado.
  - d. Es una señal de que las pruebas fueron poco rigurosas.
4. En el contexto de pruebas de software, ¿qué se entiende por un fallo?
  - a. Un reporte formal de resultados de prueba.
  - b. Un error en la documentación técnica.
  - c. Un efecto indeseado en las funciones o prestaciones del sistema.



- d. Una imperfección menor que no afecta al usuario.
5. ¿A qué nivel se realizan pruebas enfocadas en comprobar si un objeto u operación específica cumple con su especificación?
- a. Pruebas de componentes.
  - b. Pruebas de integración.
  - c. Pruebas de sistema.
  - d. Pruebas de aceptación.
6. El objetivo principal de las pruebas de componentes es:
- a. Confirmar que el componente cumple con lo especificado.
  - b. Analizar la usabilidad del componente.
  - c. Revisar las interfaces entre módulos.
  - d. Medir el rendimiento del componente.
7. En las pruebas de integración, la estrategia de pruebas incrementales consiste en:
- a. Probar grupos de componentes ya certificados e ir agregando progresivamente los nuevos.
  - b. Seguir la misma lógica de las metodologías ágiles: desarrollar – probar – implantar continuamente.
  - c. Reprobar todos los componentes cada vez que se añade uno nuevo.
  - d. Solo probar los componentes más críticos.
8. Entre las estrategias de pruebas incrementales (ascendente, descendente y combinada), ¿cuál trabaja con tres capas de componentes?
- a. Estrategia combinada.
  - b. Estrategia ascendente.
  - c. Estrategia descendente.
  - d. Estrategia aleatoria.



9. ¿En qué aspectos se enfocan las pruebas de sistema?

- a. En la integración entre componentes.
- b. En la seguridad, velocidad y exactitud del sistema.
- c. En la usabilidad de la interfaz.
- d. En la corrección de código.

10. ¿Cuál es la ventaja principal de que el equipo de pruebas del sistema sea externo al desarrollo?

- a. Asegura independencia en los resultados.
- b. El equipo desconoce el sistema.
- c. Permite que el equipo interno se dedique al desarrollo.
- d. Reduce el costo de las pruebas.

[Ir al solucionario](#)



## Resultado de aprendizaje 1 a 4:

- Conoce las principales áreas de conocimiento de la ingeniería de software.
- Conoce e identifica las fases del ciclo de vida de desarrollo de sistemas.
- Identifica el ciclo de vida a utilizar en el desarrollo de un proyecto de software.
- Identifica una metodología de desarrollo acorde a las características de un proyecto de software.

### Contenidos, recursos y actividades de aprendizaje recomendadas



#### Semana 16

#### Actividades finales del bimestre

Es momento de preparar la evaluación presencial, y al igual que en el primer bimestre le recomiendo que realice una revisión de las actividades de aprendizaje de las unidades 6, 7 y 8, asegurándose de que comprende las temáticas tratadas, las actividades de final de unidad le permitirán afianzar su conocimiento.



#### Actividades de aprendizaje recomendadas

Continuemos con el aprendizaje mediante su participación en las actividades que se describen a continuación:

1. Como estrategia de estudio, le recomiendo nuevamente elaborar mapas mentales, pero sobre todo llevar a cabo los ejercicios indicados, ya que estas unidades son más prácticas.

*Nota.* Por favor, complete la actividad en un cuaderno o documento Word.





## 4. Solucionario

### Autoevaluación 1

Pregunta	Respuesta	Retroalimentación
1	c	La “crisis del software” estuvo ligada al salto de complejidad que trajeron los sistemas de tiempo compartido y entornos multiusuario, en un contexto con métodos y herramientas inmaduros. Esto generó retrasos, sobrecostos y gran cantidad de fallos.
2	a	Corresponden a la segunda era, donde el software era considerado como producto y había explotado la crisis del software, por lo que se desarrollaron sistemas operativos multiusuario.
3	b	SOA expone funcionalidades como servicios con interfaces estándar y accesibles externamente, facilitando interoperabilidad, integración entre proveedores, reutilización y bajo acoplamiento. La nube es un modelo de provisión distinto y la proliferación de smartphones o el enfoque en procesos no definen la arquitectura.
4	a	En efecto, el software no es solamente los programas, y desde el punto de vista de la ingeniería de software incluye los tres elementos mencionados, la alternativa d incluye personas y procesos que no corresponden al software y las demás están incompletas.
5	d	Todas las características, excepto la d, son aplicables al software, de hecho, el software es uno de los productos con mayor grado de incertidumbre durante su proceso de desarrollo.
6	b	En esta alternativa hay dos errores: primero, el cliente no siempre sabe lo que necesita ni conoce el proceso, por lo tanto, no puede indicar paso a paso lo que necesita y, segundo, un enfoque secuencial está demostrando que no es la mejor manera de abordar un proyecto de software.
7	c	La ingeniería de software se creó precisamente con el propósito de elevar la calidad de los productos de software, por lo tanto, todos los demás son efectos colaterales.



**Pregunta    Respuesta    Retroalimentación**

- 8      a      La principal razón para usar modelos en cualquier rama de la ciencia es incrementar la comprensión de sistemas complejos, aunque lo demás es consecuencia del modelado. Este no siempre es visual, muchos modelos pueden ser matemáticos y no necesariamente derivan en documentación.
- 9      d      Aunque podría ser deseable siempre usar modelos, estos tienen más sentido en sistemas grandes y complejos, en las demás alternativas, no habría un aporte significativo de los modelos.
- 10     d      El dominio de la aplicación corresponde a la comprensión del negocio, y esto se hace precisamente en el levantamiento de requerimientos y se consolida en el análisis.

[Ir a la autoevaluación](#)



## Autoevaluación 2

Pregunta	Respuesta	Retroalimentación
1	a	Los usuarios del sistema son los que permiten al analista establecer los servicios, identificar las restricciones y plantear las metas del sistema a desarrollar, por lo tanto, la participación del usuario es fundamental, por tal motivo la etapa que requiere una participación directa del usuario en el modelo en cascada es en la etapa de análisis y definición de requerimientos.
2	a	En el modelo en cascada, cada fase contempla el desarrollo de documentos debidamente autorizados, que básicamente son los insumos de la siguiente fase. Esto no quiere decir que, si en algún momento se necesita hacer algún alcance, se puede hacer siempre y cuando finalice una actividad.
3	b	En el desarrollo incremental, la especificación, el desarrollo y la validación están entrelazadas, lo que permite realizar una serie de pasos hacia la solución y retroceder cuando se detectan errores. Cada incremento incorpora algunas funciones que necesita el cliente, siendo los primeros incrementos los que incluyen las funciones más importantes o las más urgentes.
4	c	La ingeniería de requisitos es una disciplina que aporta al proceso de desarrollo de software con la definición de los requerimientos, por tal motivo considera inicialmente realizar el estudio de factibilidad para analizar la conveniencia de realizar o no el sistema, luego centra sus esfuerzos en la obtención de información y el análisis mediante modelos de requisitos, seguidamente, se realiza la especificación de los requerimientos y finalmente validarlos.
5	b	Los enfoques orientados a la reutilización se apoyan en componentes ya desarrollados que se pueden reutilizar y que pueden servir para diferentes aplicaciones. En este sentido, los servicios web, se pueden utilizar para desarrollos basados en componentes, ya que son una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones.
6	c	En la etapa de especificación del software, se determinan los requerimientos; por lo tanto, es necesario validarlos con la intervención del usuario, ya que las definiciones son a nivel de usuario.
7	a	El diseño arquitectónico, al ser la primera etapa en el proceso de diseño de software, por lo tanto, es la etapa donde se entiende cómo debe organizarse el sistema y cómo tiene que diseñarse la estructura global de ese sistema.

Pregunta	Respuesta	Retroalimentación
8	c	Cuando se ha establecido cómo funcionará cada componente, se procede con el diseño de la base de datos, dónde se diseñan las estructuras del sistema de datos y cómo se representarán en una base de datos en particular.
9	d	Cuando el desarrollador construye un componente, lo valida de forma aislada mediante pruebas unitarias. Estas verifican cada unidad de código sin integrar los demás componentes.
10	b	Los prototipos en la ingeniería de requisitos son una versión inicial del sistema, de tal manera que el usuario puede tener una idea clara de lo que el sistema realizará, así como validar los requisitos.

[Ir a la autoevaluación](#)



### Autoevaluación 3

Pregunta	Respuesta	Retroalimentación
1	c	Un proyecto es un esfuerzo que tiene un propósito que resuelve una necesidad de negocio, las otras alternativas se refieren a actividades que pueden o no estar asociadas con un proyecto. Recuerde que los trabajos que tienen que ver con el desarrollo, modificación o mantenimiento de un producto de software, se constituyen en un proyecto.
2	c	El grupo de personas que se conforman para trabajar en un proyecto se denomina equipo del proyecto. Cada persona es un participante, los usuarios no trabajan en el proyecto, aunque sí aportan las necesidades y los interesados opinan o son afectados por el trabajo del proyecto, pero no forman parte del equipo de proyectos.
3	a	El alcance del proyecto es global y comprende tanto lo relacionado con el proyecto como lo relacionado con el producto, las demás alternativas son incorrectas.
4	b	El papel de los interesados es esencial durante todo el proyecto, y pueden tener alta influencia e incluso pueden detener el desarrollo del proyecto. Las demás alternativas son falsas.
5	d	Un entregable es un producto de trabajo que se entrega a los clientes, en este caso, los productos de trabajo a, b y c son para uso interno del equipo del proyecto.
6	a	En la <b>etapa de definición</b> (previa a la ejecución), el arquitecto, el gestor del proyecto y los usuarios clave conciben la arquitectura del sistema y buscan entender los <b>subsistemas</b> que formarán parte del producto final. Esta etapa orienta el diseño de alto nivel antes de avanzar con el desarrollo.
7	c	Debido a que trabaja con el equipo de desarrollo y también con el lado del cliente para asegurarse que el producto hace lo que se espera.
8	c	En efecto, cuando se dibuja un cronograma, bien sea en un diagrama Pert o, en un diagrama de Gantt, la ruta crítica se dibuja y resalta con color rojo.
9	a	El diagrama de Gantt es una matriz en la que se representa verticalmente, las actividades y horizontalmente la escala de tiempo, las barras reflejan cuándo se desarrollará cada actividad en la escala de tiempo.

10

d

El diagrama Pert es un diagrama que forma un grafo en el cual los nodos son las actividades y las dependencias, entre estas se visualizan por líneas que las une; por tanto, estas dependencias se visualizan mejor en este tipo de diagramas, el diagrama de barras es igual que el de Gantt y el diagrama de actividades no se usa para representar cronogramas.

[Ir a la autoevaluación](#)



## Autoevaluación 4

Pregunta	Respuesta	Retroalimentación
1	c	El modelo funcional agrupa casos de uso para representar cómo interactúan los actores con el sistema. Esto facilita la visión de los requerimientos desde la perspectiva del usuario.
2	c	El diagrama de estados describe la secuencia de cambios de un objeto según los eventos recibidos. Es esencial para sistemas reactivos como controladores o software embebidos.
3	b	Los objetos de control gestionan la lógica de los casos de uso, coordinando entidades e interfaces. Ejemplo: un "GestorDeReserva" en un sistema de vuelos.
4	b	Los archivos, carpetas y discos representan información persistente; son objetos de entidad. Se distinguen de los de interfaz (pantallas) y control (lógica).
5	c	Generalización abstracta lo común de varias clases en una superclase. Especialización detalla la superclase en subclases específicas. Ejemplo: Vehículo → Auto, Bicicleta.
6	d	La composición implica una relación “todo-parte” fuerte: si el todo desaparece, también sus partes. Hospital y pacientes/médicos ejemplifican esta dependencia estructural.
7	b	Un modelo estabilizado significa que solo quedan cambios mínimos. Esto indica madurez en el análisis y solidez antes de avanzar al diseño y codificación.
8	c	La causa principal suele ser información mal recolectada. Una buena entrevista con usuarios, observación y análisis del contexto previenen discrepancias en los requisitos.
9	d	“Soportar 50.000 usuarios simultáneos” es un requisito no funcional del producto, pues establece condiciones de rendimiento, no funcionalidades específicas. Relaciona calidad y escalabilidad.
10	c	El prototipado permite validar con usuarios antes de la construcción final. Favorece la retroalimentación temprana y reduce riesgos de malentender necesidades.

[Ir a la autoevaluación](#)

## Autoevaluación 5

Pregunta	Respuesta	Retroalimentación
1	b	El diagrama de despliegue muestra cómo los componentes del sistema se asignan a <i>hardware</i> y <i>software</i> , útil para planificar rendimiento, disponibilidad y comunicación entre nodos.
2	c	La velocidad de procesamiento es un aspecto de rendimiento, no de seguridad. Una política de control de acceso se centra en permisos, restricciones y condiciones de uso de la información.
3	a	Un DBMS asegura persistencia, consistencia y rapidez en el acceso a datos. Ejemplos como MySQL o PostgreSQL ofrecen transacciones, índices y seguridad integrada.
4	b	El diagrama de contexto simplifica el sistema a un único proceso y muestra relaciones con actores externos, permitiendo delimitar alcance y fronteras del sistema.
5	a	El propósito central del diagrama de contexto es marcar el alcance del sistema, diferenciando claramente lo que está dentro y fuera, clave en el análisis inicial.
6	c	La vista de escenarios vincula las demás en el modelo 4+1, usando casos de uso para conectar las perspectivas lógica, de procesos, de desarrollo y física.
7	b	Un actor representa una entidad externa (usuario, sistema u organización) que interactúa con el <i>software</i> . Ejemplo: un cliente que consulta su cuenta en línea.
8	b	En el caso TRAMIL, el bodeguero registra paquetes en el sistema. Este rol refleja procesos internos de logística y control de inventario.
9	b	Documentar flujo normal, alterno, precondiciones y postcondiciones evita ambigüedades. Esto asegura claridad y facilita validar requisitos con los usuarios.
10	b	Los casos de uso reducen ambigüedad y facilitan la comunicación entre desarrolladores y clientes, sirviendo como puente entre requerimientos funcionales y diseño técnico.

[Ir a la autoevaluación](#)

## Autoevaluación 6

Pregunta	Respuesta	Retroalimentación
1	a	La arquitectura en pequeño se preocupa de los programas individuales, en tanto que la arquitectura en grande se encarga de modelar componentes de grandes sistemas. El Modelo Vista Controlador (MVC) es un estilo cuyo fin es separar los componentes de datos, de la lógica de negocio y de la visualización, por lo tanto, este aplica a grandes aplicaciones empresariales.
2	c	Los requerimientos funcionales corresponden a las operaciones que debe realizar el usuario y corresponden a la lógica de negocio. Estos se implementan en componentes individuales, ya que, a nivel de arquitectura, se representan los requerimientos no funcionales.
3	a	El desarrollar la arquitectura del sistema tiene ventajas evidentes en diferentes etapas y aspectos, en el capítulo 6 del texto básico se establecen tres ventajas de este enfoque, una de las cuales se manifiesta la comunicación con los participantes, y se establece que esto se da porque se trata de una presentación a alto nivel del sistema, que puede usarse para discutir sobre las decisiones del sistema.
4	a	El modelo de análisis es un paso intermedio, que permite tomar los requerimientos identificados en la fase de levantamiento y los transforma en conceptos compatibles con aplicaciones de software; sin embargo, no representa la estructura del sistema, esto lo hace el diseño.
5	c	La disponibilidad es un atributo de calidad que establece que el sistema estará disponible la mayor parte del tiempo, y en una forma de garantizar ante posibles fallos es la redundancia, establecida en la alternativa seleccionada.
6	a	Cada atributo de calidad se enfoca en diferentes características del sistema, muchas de las cuales pueden entrar en conflicto con otras, por ejemplo, si se quiere tiempo de facilidad de uso, se puede contraponer con la seguridad, Ej. En las aplicaciones bancarias, por más usables que sea, la seguridad obliga a que se coloquen controles como contraseñas, imágenes, preguntas secretas, etc., que hacen que la facilidad de uso se vea afectada.
7	c	El modelo propuesto por Kruchten comprende 5 vistas, una vista lógica, una vista de procesos, la vista física, la vista de desarrollo, y la vista de escenarios de casos de uso. De ellos la vista lógica representa el modelo de objetos y clases.



**Pregunta    Respuesta    Retroalimentación**

- 8      b      Existen diferentes tipos de lenguajes para representar los modelos del sistema, el lenguaje UML se usa para representar diferentes modelos del sistema, entre ellos la arquitectura, pero no logra un nivel de detalle suficiente; el lenguaje C es un lenguaje de programación y los ADL son lenguajes de descripción de la arquitectura y están destinados específicamente para ese propósito.
- 9      c      La metodología de desarrollo se enfoca en la forma de organizar las actividades de desarrollo, el detalle de la arquitectura es un elemento de construcción del producto que puede hacerse bien sean mediante un enfoque ágil o un enfoque tradicional. Esto depende del nivel de complejidad del producto que se esté desarrollando.
- 10     c      Las recomendaciones generales sobre cohesión y acoplamiento indican que se debe minimizar el acoplamiento y maximizar la cohesión, sin embargo, es preciso considerar que cualquier decisión en esa línea afecta a la otra, es decir, para incrementar la cohesión es preciso reducir la cohesión, por tanto, lo más recomendable es usar la heurística  $7 \pm 2$ , es decir, entre 5 y 9 niveles de descomposición en subsistemas.

[Ir a la autoevaluación](#)

## Autoevaluación 7

Pregunta	Respuesta	Retroalimentación
1	b	Los objetivos del diseño se centran específicamente en resolver los requerimientos no funcionales, que darán soporte a los requerimientos funcionales, los cuales se deben implementar en los componentes.
2	a	Cuando hablamos de contexto, siempre nos referimos a una vista global del sistema, y este, por tanto, no puede mostrar la parte interna del sistema, sino sobre todo los sistemas externos. Las restricciones no tienen cabida en esta apreciación.
3	c	Uno de los elementos más importantes del diseño es descomponer el sistema en subsistemas que cumplan características mínimas como la cohesión, acoplamiento y puedan organizarse en alguno de los estilos arquitectónicos estudiados.
4	c	Las respuestas a eventos están relacionadas con el comportamiento de los objetos, con la cual la única alternativa es la de las máquinas de estados.
5	b	Cada uno de los diagramas de UML tiene un propósito: los diagramas de componentes muestran los componentes lógicos que conforman el sistema, los diagramas de actividades muestran las secuencias de operaciones que deben realizarse y los diagramas de implantación, precisamente muestran los dispositivos con los componentes que correrán en cada uno de ellos.
6	a	El almacenamiento persistente se hace sobre aquellos elementos que necesitamos almacenar y actualizar en el tiempo. Los denominados objetos borde y de control solo existen en tiempo de corrida; por tanto, los objetos que pueden utilizarse para este tipo de almacenamiento son los objetos entidad.
7	c	Las bases de datos Orientadas a Objetos se especializan en almacenar los objetos como entidades completas, sin embargo, estas estructuras son complejas y las operaciones que deben realizarse se ven afectadas en el rendimiento, y uno de los motivos es porque no hay un modelo matemático que le sirva de base como sucede con las bases de datos relacionales.
8	c	La encriptación consiste en utilizar un algoritmo que oculta la información reemplazándola por un contenido ilegible por parte de un tercero, lo cual significa que, si alguien logra robar, no la podrá entender a no ser que la desencripte.

**Pregunta    Respuesta    Retroalimentación**

9        b        Por definición, un patrón recoge un conjunto de buenas prácticas aplicado en diferentes casos para resolver problemas particulares; en este caso, se trata de problemas de diseño.

10       c        Las herramientas de administración de la configuración permiten mantener diferentes versiones de componentes que pueden ser utilizados por varios participantes, guardando la consistencia y preservando el control de cambios; de este modo, evitan conflictos en el uso de dichos componentes.

[Ir a la autoevaluación](#)



## Autoevaluación 8

Pregunta	Respuesta	Retroalimentación
1	c	En la introducción al capítulo 8 del texto básico, se establecen los conceptos de verificación y validación, los cuales, a pesar de que suelen confundirse, tienen objetivos diferentes. La verificación se enfoca en comprobar que el software cumpla con sus especificaciones (funcionalidades y requerimientos no funcionales), por su parte, la validación se enfoca en asegurar que se cumplan con las expectativas del cliente, es decir, que el software le será útil.
2	b	Las inspecciones no sustituyen a las pruebas del software, debido a que no son suficientes al momento de detectar defectos surgidos por interacciones inesperadas.
3	b	La presencia de errores en las primeras versiones del producto suelen darse como un comportamiento normal. Sin embargo, si este número de detecciones incrementa de manera considerable, la única conclusión a la que se puede llegar es que podrían presentarse errores en componentes futuros; las otras alternativas, no hay forma de demostrar que sean ciertas.
4	c	De acuerdo con la definición de fallo dada en los conceptos de inicio del presente capítulo, la C sería la alternativa correcta.
5	a	Las pruebas de componentes validan cada módulo de manera aislada, asegurando que cumpla su especificación. Ejemplo: probar una clase de autenticación antes de integrarla al sistema completo.
6	a	Esto ayuda a detectar errores de interfaz tempranamente y evita que problemas pequeños se propaguen a niveles superiores.
7	a	La estrategia incremental implica reejecutar pruebas anteriores más el nuevo componente. Aunque costosa, asegura estabilidad progresiva y se conecta con la idea de regresión en pruebas continuas.
8	a	La estrategia combinada utiliza la estrategia ascendente y descendente y el sistema se presenta como un sándwich con 3 capas. La capa central es la que se desea probar, mediante enfoque ascendente prueba la capa central, asumiendo que la inferior ya ha sido implementada, y se desciende a la capa de prueba desde la capa superior.
9	b	Una vez realizadas las pruebas de componentes, donde se han determinado la validez de los componentes y las interfaces, el siguiente nivel son las pruebas del sistema, cuyo enfoque es precisamente aspectos como la velocidad, seguridad, exactitud, entre otros.



**Pregunta    Respuesta    Retroalimentación**

10

a

La ventaja más importante es, precisamente, la independencia de las pruebas son una característica deseable en cualquier equipo de pruebas o de auditoría. Las otras dos alternativas que son ciertas, no representan una ventaja.

[Ir a la autoevaluación](#)





## 5. Glosario

### Alcance

Identifica y limita lo que hará el sistema como parte del nuevo proyecto de desarrollo.

### Analista de negocios (BA)

Es el rol en un equipo de proyectos que tiene la responsabilidad de interactuar con los representantes de los interesados para obtener, analizar, especificar, validar y administrar los requisitos del proyecto. También se lo denomina analista de requisitos, analista de sistemas, ingeniero de requisitos, administrador de requisitos, analista de sistemas de negocios o simplemente analista.

### Arquitectura de software

Es la estructura del sistema, la cual comprende elementos de software, las propiedades externamente visibles de esos elementos y las relaciones entre ellos.

### Diagrama de contexto

Es un modelo de análisis que representa un sistema con un alto nivel de abstracción. El diagrama de contexto identifica objetos fuera del sistema que intercambian datos con el sistema, pero no muestra nada sobre la estructura interna o el comportamiento del sistema.

### Modelo 4+1 Vistas

Modelo desarrollado por Philippe Krutchen, que describe la arquitectura del software usando cinco vistas concurrentes, cada vista se refiere a un conjunto de intereses de diferentes *stakeholders* del sistema: vista lógica que describe

el modelo de objetos del diseño, vista de procesos que describe los aspectos de concurrencia y sincronización del diseño, vista física que describe el mapeo del *software* en el *hardware* y refleja los aspectos de distribución, y la vista de desarrollo describe la organización estática del software en su ambiente de desarrollo. A esta vista se agrega la vista de escenarios que representa una abstracción de los requisitos más importantes.

### **Objetivo de negocio**

Un beneficio comercial, financiero o no financiero, que una organización espera recibir como resultado de un proyecto o alguna otra iniciativa.

### **Patrón arquitectónico**

Son un conjunto de responsabilidades, reglas y directrices que se utilizan para expresar una estructura de organización base o esquema para un *software*, que sirven para determinar la organización, comunicación, interacción y relaciones entre ellos.

### **Patrón de diseño**

Son el esqueleto de las soluciones a problemas comunes en el desarrollo de *software*; brindan una solución ya probada y documentada a problemas de desarrollo de *software* que están sujetos a contextos similares.

### **Product champion**

Término utilizado para identificar a un representante de una clase de usuario específica que proporciona los requisitos de usuario para el grupo que representa.

### **Requisito**

Circunstancia o condición necesaria para algo. Es un concepto amplio que podría referirse a cualquier función, atributo, capacidad, característica o calidad necesaria de un sistema para que tenga valor y utilidad para un cliente,



organización, usuario interno u otro interesado. Se usa comúnmente en un sentido formal en el diseño de ingeniería, que incluye, por ejemplo, ingeniería de sistemas, ingeniería de software o ingeniería empresarial.

## **Requisitos de negocio**

Un conjunto de información que describe una necesidad comercial y que conduce a uno o más proyectos para entregar una solución con los resultados deseados. Los requisitos del negocio incluyen oportunidades comerciales, objetivos comerciales, métricas de éxito, una declaración de visión y alcance y limitaciones.

## **Stakeholder**

Término utilizado para identificar a una persona, grupo u organización que tiene interés o preocupación en una organización. Los interesados pueden afectar o ser afectados por las acciones, objetivos y políticas de la organización. Es común utilizar el término en inglés o su equivalente en español (interesado).

## **UML**

Abreviatura en inglés del Lenguaje de Modelado Unificado, que describe un conjunto de notaciones estándar para crear varios modelos visuales de sistemas, particularmente para el desarrollo de software Orientado a Objetos.

## **Visión**

Es una declaración que describe el concepto estratégico o el propósito final y la forma de un nuevo sistema.

## **EA**

La herramienta de modelado *Enterprise Architecture*, nos da una prueba de evaluación de 30 días.





## 6. Referencias bibliográficas

Ahmed, A. , & B. P. (2016). *Foundations of Software Engineering*.

Ahmed A, & Bhanu P. (2018). Software Engineering in the Agile World. N  
dently Published.

Bruegge, Bernd., & Dutoit, A. H. . (2018). *Object-oriented software  
engineering : using UML, patterns, and Java*. Prentice Hall.

Gottesdiener, E. (2005). *Software Requirements Memory Jogger*.

Melissa A. Russell. (2018). *Guía práctica de Ágil*. Independent Publishers  
Group : Project Management Institute.

*IEEE Standard Glossary of Software Engineering Terminology*. (1990).  
[https://ieeexplore.ieee.org/document/159342/?denied=\[8/9/2025](https://ieeexplore.ieee.org/document/159342/?denied=[8/9/2025)

Kroll, P, & Kruchten, P. (2003). Introducing the *Rational Unified Process*.  
*In Rational Unified Process Made Easy: A Practitioner's Guide to the  
RUP*.

Kung, D. (2014). *Object-Oriented Software Engineering An Agile Unified  
Methodology Solutions Manual*.

Pressman, R. (2010). *Ingeniería del Software, un enfoque práctico* (M.  
Hill, Ed.; Séptima).

Project Management Institute. (2017). *A guide to the project  
management body of knowledge*. Project Management Institute, Inc.

Rumbaugh, J., Jacobson Ivar, & Booch Grady. (2005). *The Unified  
Modeling Language Reference Manual*.

Ruparelia, N. B. (2016). Software development lifecycle models. ACM SIGSOFT Software Engineering Notes, 35(3), 8–13. <https://doi.org/10.1145/1764810.1764814>

Sajid, A. G. (2021). Comparative Analysis of Software Process Models in Software Development. *International Journal of Advanced Trends in Computer Science and Engineering*, 10(3), 2593–2599. <https://doi.org/10.30534/ijatcse/2021/1521032021>

Sánchez, S. , S. M. , y R. D. (2012). *Ingeniería de Software. Un enfoque desde la guía SWEBOK* (Alfaomega, Ed.). Alfaomega.

Sommerville, Ian. (2016). *Software engineering* (10th ed.). Pearson.

Wambler, S. (2005). *The Elements of UML TM 2.0 Style*.

Washizaki, H. (n.d.). *Guide to the Software Engineering Body of Knowledge*.

Wirth, N. (1918). *A Brief history of Software Engineering*.



---

## 7. Anexos

---

## Anexo 1. Plantilla para documentar los requisitos de software

### Especificación de requisitos de software proyecto

#### Historial de revisiones

Fecha	Versión	Descripción	Autor

#### 1. Introducción.

##### 1.1 Propósito.

##### 1.2 Alcance del proyecto.

#### 2. Descripción general.

##### 2.1 Perspectiva del producto.

##### 2.2 Módulos del producto.

ID	Módulo	Descripción

#### 2.3. Clases y características de usuario

N.º	Usuario	Descripción

#### 3. Características del sistema (requisitos funcionales por módulos)

##### 3.1 Módulo

ID	Requisito	Trazabilidad	Prioridad	Estabilidad

Trazabilidad: personas o documentos que han proporcionado información sobre el requisito.

Prioridad: para los clientes y usuarios (escala: alto, medio, bajo).

Estabilidad: probabilidad de que el requisito no cambie durante el resto del desarrollo (escala: alto, medio, bajo).

### Módulo

ID	Requisito	Trazabilidad	Prioridad

Trazabilidad: personas o documentos que han proporcionado información sobre el requisito.

Prioridad: para los clientes y usuarios (escala: alto, medio, bajo).

Estabilidad: probabilidad de que el requisito no cambie durante el resto del desarrollo (escala: alto, medio, bajo).

### Módulo

ID	Requisito	Trazabilidad	Prioridad

Trazabilidad: personas o documentos que han proporcionado información sobre el requisito.

Prioridad: para los clientes y usuarios (escala: alto, medio, bajo).

Estabilidad: probabilidad de que el requisito no cambie durante el resto del desarrollo (escala: alto, medio, bajo).

## 4. Reglas de negocio

ID	Definición	Tipo	Estática/ Dinámica	Fuente
RN-01				

## 5. Atributos de calidad

### Usabilidad

ID	Requisito	Trazabilidad	Prioridad
RNF-01			

### Rendimiento

ID	Requisito	Trazabilidad	Prioridad

## Seguridad

ID	Requisito	Trazabilidad	Prioridad

## Disponibilidad

ID	Requisito	Trazabilidad	Prioridad
RNF-01		Director	Medio

## Robustez

ID	Requisito	Trazabilidad	Prioridad

## Mantenibilidad

ID	Requisito	Trazabilidad	Prioridad

## Anexo 2. Plantilla para especificar casos de uso

**Tabla 1.**

Plantilla para especificar casos de uso

Caso de uso:	CU01.
Nombre:	Registrar paquete.
Autor:	Danilo Jaramillo H.
Fecha:	Mayo de 2025.
Descripción:	Se registrarán los datos del paquete que lleguen a las oficinas.
Actores:	Bodeguero.
Precondición	El cliente debe estar creado.
<b>Flujo normal</b>	
Actor	Sistema
1. Solicita la generación de un nuevo código para el paquete.	
	2. Genera nuevo código único para el paquete.
3. Ingresa datos del paquete de descripción, peso, observación.	
	4. Valida datos del paquete.

5. Ingresa la cédula del cliente para su búsqueda.	
	6. Recupera los datos del cliente como nombre, apellido, dirección y presenta los mismos.
7. Confirma la información ingresada y presentada.	
	<i>8. Registra los datos del nuevo paquete y presenta mensaje de confirmación del proceso.</i>
<b>Flujo alterno</b>	
5. Se permite realizar una búsqueda por apellidos/nombres.	
6. Se permite ingresar nuevo cliente.	
<b>Postcondición:</b> los datos del paquete.	
<b>Escenario:</b> el paquete contiene todos los datos del mismo y del cliente.	

Nota. Jaramillo, D., 2025.

## **Anexo 3. Esquema del plan de prueba según el estándar IEEE829**

1. Identificador del plan de pruebas.
2. Referencias.
3. Introducción.
4. Elementos de prueba (funciones).
5. Problemas de riesgo de *software*.
6. Características que no serán probadas.
7. Enfoque (estrategia).
8. Criterios de elementos aprobados/fallidos.
9. Criterios de suspensión y requisitos de reanudación.
10. Entregables de las pruebas.
11. Tareas de prueba restantes.
12. Necesidades del entorno.
13. Necesidades de personal y de capacitación.
14. Responsabilidades.
15. Cronograma.
16. Planificación de riesgos y contingencias.
17. Aprobaciones.
18. Glosario.