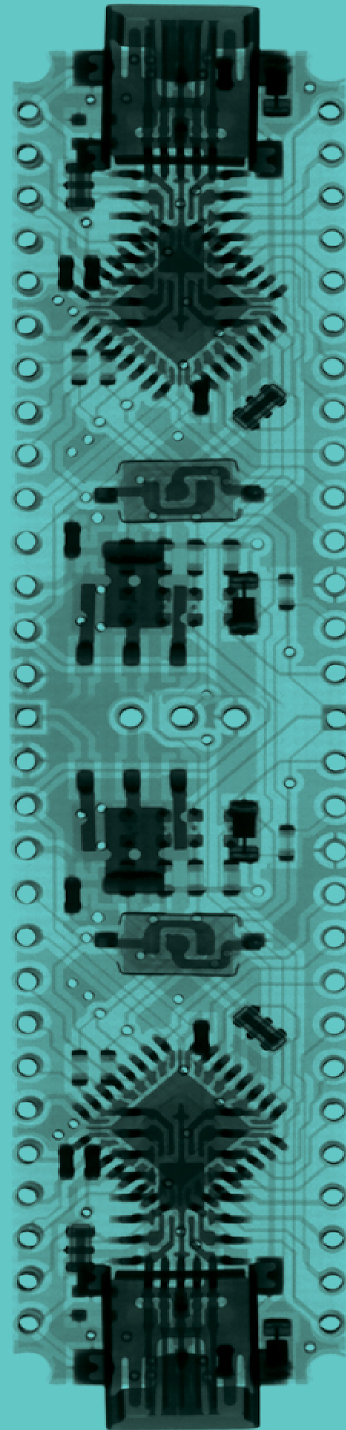
CIENCIAS NATURALES E INGENIERÍA



Modelado, arquitectura y comunicación en el desarrollo de sistemas de software: un enfoque práctico

PEDRO PABLO GONZÁLEZ PÉREZ

ISMAEL SOTO GALINDO

MATEMÁTICAS APLICADAS Y SISTEMAS

COLECCIÓN LIBRO DE TEXTO

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

Contenido

Portada

Legal

I. INTRODUCCIÓN

1.1. VISIÓN Y OBJETIVOS

1.2. ESCENARIO

1.3. CONTEXTO

1.4. ESTRUCTURA DEL MATERIAL

i. RELACIÓN DEL CONTENIDO CON PROGRAMAS DE ESTUDIO DE LA LICENCIUTURA EN INGENIERÍA EN COMPUTACIÓN

ii. DESCRIPCIÓN DE LA IMPORTANCIA DE LOS CONOCIMIENTOS A ADQUIRIR, ASÍ COMO DE LAS HABILIDADES Y ACTITUDES A DESARROLLAR

II. EL PAPEL DEL MODELADO EN EL DESARROLLO DE SOFTWARE

2.1. El modelado en el paradigma de desarrollo orientado a objetos

2.2. El Lenguaje Unificado de Modelado

III. EL PAPEL DE LA ARQUITECTURA EN EL DESARROLLO DE SOFTWARE

3.1. Arquitectura lógica

3.2. Arquitectura física

IV. LA IMPORTANCIA DE LA COMUNICACIÓN EN EL DESARROLLO DE SOFTWARE

4.1. El rol del usuario/cliente final en la comunicación en el desarrollo de software

4.2. El rol de los responsables/líderes de proyecto en la comunicación en el desarrollo de software

4.3. El rol de los desarrolladores en la comunicación durante el desarrollo de software

V. EL CICLO DE VIDA EN EL DESARROLLO DE SISTEMAS DE SOFTWARE

5.1. Modelos de cascada

5.2. Modelos incrementales

5.3. Modelos iterativos

VI. METODOLOGÍAS DE DESARROLLO DE SOFTWARE ORIENTADAS A MEJORAR LA VELOCIDAD Y LA CALIDAD DE LA ENTREGA DE VALOR

6.1. Lean versus Agile

6.2. DevOps

6.3. Análisis de las técnicas y prácticas avanzadas

VII. CONCLUSIONES

REFERENCIAS

GLOSARIO

UNIVERSIDAD AUTÓNOMA METROPOLITANA
Casa abierta al tiempo
Unidad Cuajimalpa

MATEMÁTICAS APLICADAS Y SISTEMAS
CIENCIAS NATURALES E INGENIERÍA

Modelado, arquitectura y comunicación en el desarrollo de sistemas de software: un enfoque práctico

Pedro Pablo González Pérez
Ismael Soto Galindo

Esta obra fue evaluada para su publicación por el Consejo Editorial de la Rectoría de la UAM Unidad Cuajimalpa, con base en los dictámenes solicitados a pares académicos mediante un esquema que preserva el anonimato mutuo. Estos dictámenes resultaron favorables.

D. R. © 2023, de esta edición:

Universidad Autónoma Metropolitana

Unidad Cuajimalpa

Av. Vasco de Quiroga 4871, col. Santa Fe Cuajimalpa

Alcaldía Cuajimalpa de Morelos

C. P. 05348, Ciudad de México

www.cua.uam.mx

ISBN: 978-607-28-3021-9

ISBN: 978-607-28-3020-2 (Colección)

Se prohíbe la reproducción total o parcial de esta obra, sea cual fuere el medio, electrónico o mecánico, sin el consentimiento por escrito de los titulares de los derechos.

I. INTRODUCCIÓN

La intención de este material es compartir con el lector el papel e importancia del modelado, la arquitectura y la comunicación en el proceso de desarrollo de software desde una perspectiva de investigación y desarrollo. Con este fin navegaremos sobre la columna vertebral de lo que, a nuestro parecer, es lo mínimo imprescindible a considerar para desarrollar software exitoso en el contexto actual. De forma específica, nos referimos al papel del modelado, la arquitectura y la comunicación como elementos clave que, amalgamados en un modelo de ciclo de vida de desarrollo de software (CVDS), son precursores de software de calidad.

A lo largo de cada uno de los epígrafes de este material, se hace énfasis en las estrategias, modelos, artefactos y herramientas comúnmente utilizados para comunicar de forma efectiva a los diferentes actores el significado e intención de las decisiones —arquitectónicas, de componentes, funcionales y de restricciones— que se toman durante el CVDS. Es importante notar que, aunque los temas aquí abordados no se desarrollan de forma exhaustiva, sí establecen un punto de referencia que pretendemos que pueda usarse como guía tanto por principiantes como por profesionales cuando emprendan proyectos de desarrollo de software.

Para mayor comprensión de los temas abordados en este material —modelado, arquitectura y comunicación de software— se ha procurado, a lo largo de los capítulos, ejemplificar la aplicación de dichos conceptos mediante dos casos de estudio: uno corresponde al desarrollo de un sistema cliente-servidor web para las reservaciones y ventas de una aerolínea, y el otro está relacionado con el desarrollo de una herramienta de simulación bioinformática para la experimentación in silico con vías de señalización celular (González-Pérez, P.P., Cárdenas-García, 2018; González-Pérez, P.P., Cárdenas-García, 2019).

1.1. VISIÓN Y OBJETIVOS

La visión del presente material es establecer un compendio de bases teóricas y prácticas que pueda fungir como columna vertebral de cualquier proyecto de desarrollo de software. En específico, que sea un soporte de gran valor para los proyectos de desarrollo de software a pequeña, mediana y gran escala donde los actores primordiales son alumnos de la licenciatura en Ingeniería en Computación integrados en equipos de trabajo colaborativo. De esta forma, los principales objetivos que persigue el presente material son los siguientes:

- Destacar la importancia del modelado en el desarrollo de software
- Resaltar la importancia de la arquitectura de software como la columna vertebral alrededor de la cual se erige el software
- Exaltar la necesidad de contar con mecanismos que promuevan la comunicación eficaz entre el cliente y el equipo de desarrollo
- Presentar prácticas y tendencias usadas por la industria de software actual
- Proponer un flujo de trabajo que relacione prácticas y herramientas que incentiven mejores resultados en el desarrollo de software

1.2. ESCENARIO

Imaginemos que te has unido a un nuevo equipo de desarrollo y, con la idea de familiarizarte con el proyecto en el cual se trabaja y la forma de trabajar del equipo, tu primera asignación es agregar una funcionalidad menor a uno de los módulos ya existentes. Comienzas a revisar el código, y lo primero que notas, además de la falta de comentarios, es que en el repositorio donde se hospeda el proyecto no hay referencia a documentación alguna. Es decir, no se cuenta con ninguna especificación de modelado, no se posee documentación detallada de la arquitectura lógica ni de la arquitectura física, y tampoco se cuenta con historial de las principales comunicaciones efectuadas mediante modelos, diagramas y otros artefactos que involucran tanto la parte del usuario/cliente como la del equipo de desarrollo. Además, has preguntado a tus compañeros y te han confirmado que, en general, dado que siguen enfoques de metodologías ágiles de desarrollo de software, la documentación no tiene tanta importancia como sí la tiene el código en funcionamiento. Encima, para tu mala fortuna, el programador que desarrolló el módulo ya no forma parte de la empresa, por lo que despejar tus dudas con él no es posible.

Seguramente, en este momento comienzas a sentirte abrumado; sin embargo, dado que eres un gran desarrollador, después de invertir bastante tiempo en la revisión y comprensión del código fuente, al fin has entendido la funcionalidad actual del software y comienzas a desarrollar la nueva funcionalidad solicitada. Hasta este punto, debería quedarnos claro que haber contado con documentación de modelado, arquitectura y comunicación te hubiese hecho mucho más fácil poner manos a la obra y hubiera acelerado considerablemente tu proceso de familiarización con el proyecto. Ahora que has terminado de desarrollar tu funcionalidad, tienes la misión de integrar tu código al sistema y ver cómo interactúa con los demás módulos, pero —de nuevo, para tu mala fortuna— no cuentas con documentación que te permita entender cuál es la arquitectura lógica del sistema y cómo los módulos interactúan entre sí, y tampoco tienes idea acerca de la arquitectura física correspondiente; es decir, en cuáles componentes hardware se ejecutarán los diferentes módulos de software, por lo que otra vez la tarea se hace lenta dada la prolongada curva de aprendizaje a la que debes someterte, además de que incurres en riesgos innecesarios.

Por último, una vez desplegada la funcionalidad, el usuario final ha descubierto un error que pone en peligro la ejecución del software, por lo que necesita ser corregido de inmediato. Por desgracia, ni el equipo ni la infraestructura con la que disponen están preparados para hacer cambios con la velocidad necesaria, y aunque en principio la funcionalidad era menor, por no saber dónde está el defecto y no poder solucionarlo en la ventana de tiempo requerida, el proyecto está en peligro de perderse.

Este escenario, además de parecer trágico, es común en equipos con prácticas laxas o con poca experiencia en el desarrollo de software. Sin importar el tamaño de la organización ni la experiencia de los desarrolladores, una especificación no entendida o no plasmada con el suficiente detalle, sumada a un diseño carente de soporte —modelos, diagramas, tablas, relaciones, etcétera— es muy probable que derive en resultados negativos. Por fortuna, a medida que la industria del desarrollo del software sigue madurando, resulta común encontrar frameworks (infraestructuras computacionales de soporte al desarrollo de software) que encapsulan buenas prácticas y que nos permiten incorporar principios que hace años resultaba oneroso implementar; un ejemplo de ello es la integración continua, introducida como práctica en los años 90 por los creadores de XP (Beck, 1999) y que hoy es piedra angular del desarrollo moderno de software.

Además de las tareas propias de un proyecto de software, existen actividades que, si bien pareciera que no están relacionadas de manera directa con el desarrollo, tenerlas en consideración mitigan buena parte de sus riesgos y problemas. Nos referimos a la deuda técnica. Bajo este concepto puede caber, por seguir nuestro escenario, la generación de documentación que soporte los módulos con los que está trabajándose y, de esa forma, futuros desarrolladores se beneficiarán de la documentación generada. No estamos hablando de documentar todo lo ya codificado, pero sí, de ir creando documentación a medida que se comienza a interactuar con código carente de documentación. Documentar con la idea de modelos y patrones en mente nos permitirá ir insertando artefactos en diferentes vistas, de las cuales, poco a poco, la arquitectura documentada emergerá. Recordemos que, si bien documentar nuestra arquitectura y nuestro software representa un costo, del escenario aquí simulado sabemos que el precio de no tenerlo es mucho más alto.

1.3. CONTEXTO

Hoy, como nunca, nuestra dependencia de sistemas o aplicaciones de software es más que evidente. Podríamos decir que estamos viviendo una era en la que el software es ubicuo y omnipresente, y resultaría difícil de explicar el avance que tenemos como sociedad sin él. Dicho lo anterior, es fácil advertir la importancia de contar con mecanismos, modelos, herramientas y —sobre todo— buenas prácticas, que nos permitan desarrollar software de calidad y reducir en lo posible los recursos invertidos en él, tales como tiempo, personas y dinero.

Mientras que jugadores de élite, como Amazon y Netflix, tienen la capacidad de desplegar nuevas funcionalidades de sistemas de software múltiples veces por día (Accelerate State of DevOps Report, 2022), aún encontramos equipos de desarrollo atascados en procesos de construcción, despliegue y liberación demasiado lentos y con una tasa de errores inaceptable para el mercado actual. El costo de no tener una funcionalidad en tiempo y forma puede llegar a ser muy alto para la empresa, pero más costosa resulta aún la incapacidad de identificar y solucionar de manera expedita un error en producción.

Pero ¿qué es lo que hace a los jugadores de élite ser tan productivos? Un primer análisis podría arrojarnos que el factor común en este tipo de empresas es la adopción de técnicas y prácticas avanzadas de desarrollo de software, entre las que se encuentran las siguientes: Lean (Coplien, J. O. y Bjørnvig, G.,2011; Forsgren, N., Humble, J. y Kim, G., 2018), Agile (Agile Alliance, 2013) y DevOps (Brown, D. 2015; Forsgren, N., Humble, J. y Kim, G., 2018), por mencionar algunas. Sin embargo, esto no sería más que la punta del iceberg y sería un error caer en la tentación de creer que la simple adopción de dichas prácticas nos asegurará el éxito al emprender nuestros proyectos. Resulta importante resaltar que con lo apenas dicho no estamos de ninguna manera promoviendo la no inclusión de las mencionadas prácticas a nuestro portafolio como ingenieros de software. Por el contrario: entendemos que su correcta y oportuna aplicación derivará en resultados positivos para nuestros proyectos. Sin embargo, hacemos énfasis en la aplicación correcta y oportuna de estas técnicas aunadas al modelado, arquitectura y comunicación, como aristas clave en el proceso de desarrollo

de proyectos de software, pues es ahí —en el proceso de desarrollo— donde reside la posibilidad de realmente hacer una diferencia positiva.

1.4. ESTRUCTURA DEL MATERIAL

El capítulo II está dedicado a discutir y ejemplificar el papel del modelado en el desarrollo de sistemas de software. Aquí se introduce el concepto de *modelo* y su papel en el proceso de desarrollo de software como herramienta muy útil para traducir los requerimientos del usuario a una propuesta de solución. Gran parte de este capítulo se dedica al Lenguaje Unificado de Modelado (UML, por su sigla en inglés), por medio de la discusión y construcción de modelos basados en diagramas UML clave, tales como diagramas de clase, diagramas de secuencia, diagramas de actividades, diagramas de paquetes y diagramas de componentes.

El crucial papel de la arquitectura en el desarrollo de sistemas de software se aborda en el capítulo III. Aquí la arquitectura se trata como el conjunto de decisiones de diseño significativas acerca de la organización y estructuración del software para satisfacer los requerimientos del usuario, los atributos de calidad deseados y otras propiedades relevantes del producto final. El capítulo hace especial énfasis en los términos arquitectura lógica y arquitectura física del software; ambos se discuten y ejemplifican de forma extensa. Especial atención se le da al patrón de arquitectura lógica de sistemas interactivos Modelo-Vista-Controlador (MVC), así como a algunas de las variantes ampliamente difundidas de este modelo.

En el capítulo IV se trata, de forma extendida y detallada, otra de las aristas clave a considerar en el desarrollo de sistemas de software, y fundamental en el éxito de todo el proceso de desarrollo: la comunicación. Aquí se presenta el desarrollo de software como un proceso en el que varios actores trabajan de manera conjunta y colaborativa, por lo que es esencial que se comuniquen bien entre sí para garantizar la comprensión, coordinación y toma de decisiones efectivas durante las diferentes etapas del proceso. De modo particular, se enfatiza la comunicación entre el cliente y el líder del proyecto u otros miembros del equipo de desarrollo durante la recolección y especificación de los requerimientos. A lo largo del capítulo se hace patente el papel crucial de los modelos como elemento de comunicación entre los diferentes actores del equipo de desarrollo.

El capítulo V está dedicado a revisar los principales ciclos de vida en el desarrollo de sistemas de software. El ciclo de vida de desarrollo de sistemas (CVDS) se presenta como un proceso de planeación, construcción,

implementación y pruebas de sistemas de software en el cual se considera también el tipo de hardware en que se ejecutará. En específico, en este capítulo se agrupan los ciclos de vida más utilizados durante el desarrollo de sistemas de software en tres grandes categorías: modelos de ciclos de vida en cascada, modelos de ciclos de vida incrementales y modelos de ciclos de vida iterativos. Se revisan las ventajas y desventajas de adoptar cada uno de estos.

Como complemento a los ciclos de vida tratados en el capítulo V, el capítulo VI presenta algunas metodologías de desarrollo de software orientadas a mejorar la velocidad y la calidad de la entrega de valor. En concreto, este capítulo revisa tres técnicas avanzadas muy difundidas en los últimos años: Lean, Agile y DevOps. El común denominador de estas técnicas es que las tres se orientan a asegurar la calidad, el aprendizaje y la entrega de valor al usuario final.

i. RELACIÓN DEL CONTENIDO CON PROGRAMAS DE ESTUDIO DE LA LICENCIUTURA EN INGENIERÍA EN COMPUTACIÓN

Como se describe en la tabla i.1, el contenido de este material aborda temas clave que se ofrecen en las siguientes Unidades de Enseñanza Aprendizaje (UEA) de la licenciatura en Ingeniería en Computación, cuyo contenido sintético se adjunta más abajo:

- UEA 4604037 Fundamentos de Ingeniería de Software
- UEA 4604038 Proyecto de Ingeniería de Software I
- UEA 4604044 Análisis y Diseño Orientado a Objetos
- UEA 4604048 Desarrollo de Software a Gran Escala

Tabla i.1. Relación del contenido del material con programas de estudio de la licenciatura en Ingeniería en Computación

Temática abordada en el material	*UEA que favorece*
El papel del modelado en el desarrollo de software	• Fundamentos de Ingeniería de Software • Proyecto de Ingeniería de Software I • Análisis y Diseño Orientado a Objetos • Desarrollo de Software a Gran Escala
El papel de la arquitectura en el desarrollo de software	• Proyecto de Ingeniería de Software I

	• Desarrollo de Software a Gran Escala
El papel de la comunicación en el desarrollo de software	• Desarrollo de Software a Gran Escala
El ciclo de vida en el desarrollo de sistemas de software	• Fundamentos de Ingeniería de Software • Proyecto de Ingeniería de Software I • Análisis y Diseño Orientado a Objetos
De los modelos de ciclo de vida a las principales actividades del proceso de desarrollo de software	• Proyecto de Ingeniería de Software I • Desarrollo de Software a Gran Escala

CONTENIDO SINTÉTICO DE LA UEA 4604037 FUNDAMENTOS DE INGENIERÍA DE SOFTWARE

1. El proceso de desarrollo de software como el conjunto estructurado de actividades requeridas para elaborar un sistema
   - Especificación de requerimientos
   - Conceptos y principios del diseño. Introducción a UML
   - Codificación. Estándares y procedimientos de programación. El paradigma de la programación estructurada. El paradigma orientado a objetos
   - Conceptos de la calidad del software

- Pruebas y mantenimiento del software

2. Modelos de clases y objetos
   - Diagramas de clases. Clases, atributos y métodos
   - Relaciones de clases. Herencia, agregación, asociación y dependencia
   - Tipos de clases: abstracta, estática y otras
   - Paquetes de clases
   - Diagramas de objetos. Objetos y relaciones entre ellos
3. Modelos de desarrollo de software
   - El modelo en cascada y sus ciclos de vida: pura, con fases solapadas, con subproyectos, con reducción de riesgos
   - El modelo evolutivo y sus ciclos de vida: espiral, entrega por etapas o incremental, entrega evolutiva o iterativo: diseño por planificación, cascada en V.
   - Minimización de desarrollos y sus ciclos de vida: componentes reutilizables. Diseño por herramientas
4. Metodologías ágiles de desarrollo de software
   - Concepto de la metodología ágil
   - Programación extrema
   - Scrum
   - Desarrollo dirigido por pruebas
   - Desarrollo dirigido por características
5. Casos de estudio

CONTENIDO SINTÉTICO DE LA UEA 4604038 PROYECTO DE INGENIERÍA DE SOFTWARE I

1. Presentación de un caso de estudio
   - Importancia de los sistemas de software en la actualidad
   - Aplicaciones del software. Software de sistemas, de gestión, de ingeniería, basado en web, científico y otros
2. Análisis de requerimientos
   - Extracción de los requerimientos del software

- Especificación de requerimientos

3. Diseño de software
    - Diagrama de componentes
    - Diagrama de clases
    - Diagrama de secuencia
4. Codificación del diseño de software
5. Pruebas de software
    - Verificación de la funcionalidad requerida en el software
6. Documentación adicional
    - Elaboración de manuales de usuario
7. Mantenimiento
    - Atención a las fallas detectadas
    - Evaluación del impacto de las fallas en el proyecto de software

CONTENIDO SINTÉTICO DE LA UEA 4604044 ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS

1. Introducción
    - Interacciones entre sistemas y entre subsistemas. Paso de mensajes
    - El lenguaje unificado de modelado (UML). El modelo estático. El modelo dinámico
    - Herramientas de software para hacer diagramas UML
2. Análisis orientado a objetos
    - Visión del proyecto
    - Extracción de requerimientos de un proyecto
    - Creación y afinación de diagramas de casos de uso
    - El modelo del dominio (contexto del problema)
3. Diseño orientado a objetos
    - Alcance del diseño
    - Especificación de datos persistentes
    - Creación y afinación de diagramas de bloques (subsistemas u objetos) y de despliegue

- Diagramas de clases y de objetos
- Descripción de interfaces de objetos y mensajes
- Especificación del comportamiento del sistema mediante diagramas de secuencia o de estados

4. Introducción a los patrones de diseño
   - Definiciones
   - Clasificación de patrones de diseño
   - Ventajas de utilizar patrones de diseño en sistemas orientados a objetos
   - Revisión de algunos patrones deeño
5. Implementación del diseño orientado a obje distos
   - Especificación de requerimientos del sistema
   - Diseño orientado a objetos
   - Implementación orientada a objetos
   - Pruebas del sistema

CONTENIDO SINTÉTICO DE LA UEA 4604048 DESARROLLO DE SOFTWARE A GRAN ESCALA

1. Los problemas que se presentan durante el desarrollo de proyectos de software a gran escala
   - Situación actual sobre los resultados de entrega de proyectos de software
   - Problemas de las metodologías tradicionales
   - Tipos de errores y riesgos más comunes en el desarrollo del software
2. Roles y responsabilidades en el desarrollo de proyectos de software a gran escala
   - El responsable del proyecto y los líderes de equipo
   - Los analistas del sistema
   - Los arquitectos del software y los encargados del diseño detallado
   - Los programadores y los responsables de la base de datos

- Probadores y soporte técnico

3. Diseño arquitectónico
    - Conceptos de la arquitectura del software
    - Patrones arquitectónicos
    - Modelado arquitectónico
4. Diseño de componentes
    - Modularización
    - Comunicación y dependencia entre módulos (ensamblaje)
    - Modelado del diseño
5. La gestión de la configuración
    - Identificación de los elementos de la configuración del sistema
    - Control de versiones
    - Control de cambios
6. Introducción a la gestión de calidad y pruebas
    - Introducción a la gestión de calidad en los proyectos a gran escala
    - Introducción a la gestión de pruebas en los proyectos a gran escala
7. Métricas
    - Principios de medición
    - Métricas estáticas y dinámicas
    - Tipos de métricas de software
    - Herramientas para métricas de software
8. Las actividades de mantenimiento y soporte técnico
    - Tipos de mantenimiento
    - Gestión de mantenimiento
    - Software de soporte técnico

ii. DESCRIPCIÓN DE LA IMPORTANCIA DE LOS CONOCIMIENTOS A ADQUIRIR, ASÍ COMO DE LAS HABILIDADES Y ACTITUDES A DESARROLLAR

El presente material pretende contribuir tanto a la adquisición de conocimientos como al desarrollo de habilidades de los alumnos de la licenciatura en Ingeniería en Computación durante su recorrido a lo largo del homónimo plan de estudios, y de forma muy particular, cuando cursan alguna de las siguientes UEA:

- Fundamentos de Ingeniería de Software
- Proyecto de Ingeniería de Software I
- Análisis y Diseño Orientado a Objetos
- Desarrollo de Software a Gran Escala

En la tabla ii.1. se describen los temas que aborda este material y su relación con los conocimientos a adquirir, así como las habilidades a desarrollar para cada una de las UEA antes relacionadas.

Tabla ii.1. Temas que aborda el presente material y su relación con las UEA del plan de estudios de la licenciatura en Ingeniería en Computación, los conocimientos a adquirir y las habilidades a desarrollar

Tema abordado en el material	*UEA que favorece*	*Conocimientos a adquirir*	*Habilidades a desarrollar*
El papel del modelado en el desarrollo de software	• Fundamentos de Ingeniería de Software	• Modelos de clases y objetos	• Efectuar el modelado y diseño orientado a objetos de sistemas de software
	• Proyecto de Ingeniería de Software I	• Diseño de software	• Efectuar el modelado y diseño orientado a objetos

			de sistemas de software
	• Análisis y Diseño Orientado a Objetos	• Diseño orientado a objetos	• Efectuar el modelado y diseño orientado a objetos de sistemas de software
	• Desarrollo de Software a Gran Escala	• Diseño de componentes	• Diseñar componentes de un sistema de software a gran escala
	• Proyecto de Ingeniería de Software I	• Diseño de la estructura de un sistema de software	• Diseñar la estructura de un sistema de software considerando capas, • módulos y clases
El papel de la arquitectura en el desarrollo de software	• Desarrollo de software a Gran Escala	• Diseño arquitectónico	• Diseñar arquitectónicamente un sistema de software a gran escala, considerando el patrón arquitectónico Modelo-Vista-Controlador
El papel de la comunicación en el desarrollo de software	• Desarrollo de Software a Gran Escala	• Papeles y responsabilidades en el desarrollo de software a gran escala	• Identificar los principales papeles y responsabilidades en el desarrollo de software a gran escala, así como la comunicación e interacción entre los mismos
El ciclo de vida en			

el desarrollo de sistemas de software	• Fundamentos de Ingeniería de Software	• Modelos de desarrollo de software	• Evaluar y seleccionar el ciclo de vida adecuado para el desarrollo de un sistema de software a pequeña escala
	• Proyecto de Ingeniería de Software I	• Fases y actividades en el desarrollo de software	• Evaluar y seleccionar el ciclo de vida adecuado para el desarrollo de un sistema de software a pequeña escala
	• Análisis y Diseño Orientado a Objetos	• Fases y actividades en el desarrollo de software orientado a objetos	• Evaluar y seleccionar el ciclo de vida adecuado para el desarrollo de un sistema de software a mediana escala
De los modelos de ciclo de vida a las principales actividades del proceso de desarrollo de software	• Proyecto de Ingeniería de Software I	• Ciclos de vida de desarrollo de software • Fases y actividades del proceso de desarrollo de software	• Aplicar las fases y actividades del proceso de desarrollo de software en la construcción de un sistema de software a pequeña escala
	• Desarrollo de Software a Gran Escala	• Ciclos de vida de desarrollo de software • Fases y actividades del proceso de	• Aplicar las fases y actividades del proceso de desarrollo de software en la construcción de un

		desarrollo de software	sistema de software a gran escala

II. EL PAPEL DEL MODELADO EN EL DESARROLLO DE SOFTWARE

Al término *modelo* se le atribuye una amplia variedad de acepciones, tanto en las ciencias como en las tecnologías. Este puede referirse casi a cualquier cosa, desde una construcción física en un caso dado, hasta un conjunto abstracto de ideas (Achinstein, P., 1987). Los modelos son construcciones mentales, abstracciones, que nos permiten representar objetos, sistemas o ideas de manera que nos faciliten obtener información de la cual inferir comportamientos o incluso experimentar con ellos sin necesidad de recurrir al objeto o sistema real. En general, utilizar modelos nos ayuda a trabajar con aspectos complejos de un problema en una estructura adecuada para el análisis formal.

En el proceso de desarrollo de software, los modelos son una herramienta muy útil para representar ideas y buscar su validación. Pensemos que construimos software con la finalidad de resolver problemas y optimizar soluciones que aporten valor al usuario a través de sus funcionalidades, por lo que es indispensable entender qué requiere el usuario y comunicar de manera eficaz la solución que se proponga. Además, tomemos en consideración que, habitualmente, un proyecto de software reúne diferentes perfiles con distintas responsabilidades, por lo que el reto de comunicar se hace mayor en medida que la audiencia se diversifica, y justo aquí es donde los modelos tienen un papel esencial porque nos permiten abordar nuestro proyecto desde distintos puntos de vista.

Aunque a menudo los conceptos *modelo* y *diagrama* se utilizan de forma indistinta, estos son diferentes en realidad. Por un lado, como hemos ya enfatizado, el modelo es la representación abstracta de *un objeto de estudio;* en nuestro caso, un sistema de software o los elementos que lo conforman. Por otro, el diagrama es un artefacto usado para presentar una de las vistas o dimensiones del objeto que estamos modelando. Es decir: podemos ver un modelo compuesto por uno o más diagramas. Una de las principales ventajas de modelar es lo económico y viable que resulta tener una representación del problema que se desea resolver y la velocidad con la que pueden hacerse ajustes al mismo a medida que se valida. Aunado a lo anterior, cuando modelamos, es bastante sencillo preservar el historial de cambios.

2.1. El modelado en el paradigma de desarrollo orientado a objetos

Para ejemplificar lo antes expuesto, examinemos el paradigma de desarrollo orientado a objetos (Meyer, B. 2002; Braude. J. E. 2003; Larman, C. 2003; Weitzenfeld, A. 2004; Schach, S. R. 2005; Bennett, S. M. 2007) donde, dada su popularidad, no es extraño que existan diferentes métodos que guían el análisis orientado a objetos (AOO) y el diseño orientado a objetos (DOO), y cada uno de estos se basa en modelos (a su vez, compuestos por diagramas) para presentar su aproximación. Entre estos métodos, es necesario hacer referencia a los siguientes:

- El método de Booch (Booch, G., 1998)
- El método de Rumbaugh (Rumbaugh, J., Blaha, M., Premerlani, W., Hedí F., Lorensen, W., 1996)
- El método Jacobson (Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G., 1992)
- El método de Coad y Yourdon (Coad, P., Yourdon, E., 1998)

Para describir con brevedad alguno de los métodos AOO y DOO antes mencionados, tomemos como ejemplo el método de Booch. Este propone dos tipos fundamentales de modelos: físico-estático y lógico-dinámico. A partir de estos se describen las características más relevantes del sistema y las consideraciones a tener en cuenta para su construcción.

El modelo físico-estático usa los siguientes diagramas para su representación:

- Diagrama de clases
- Diagrama de objetos
- Diagrama de módulos
- Diagrama de procesos

Por otra parte, el modelo lógico se basa, sobre todo, en los siguientes diagramas:

- Diagrama de transición de estados
- Diagrama de interacción
- Diagrama de actividades

2.2. El Lenguaje Unificado de Modelado

El Lenguaje Unificado de Modelado (UML, por sus siglas en inglés) (Rumbaugh, J., Jacobson, I., Booch, G., 2004; Fowler, M., Scott, K. 1999; Fowler, M., 2004) es —quizás— el lenguaje de modelado más usado en la industria del software. UML fue desarrollado en los años 90 con la finalidad de estandarizar las herramientas visuales (diagramas), elementos (paquetes, clases, etcétera), conexiones y sus atributos utilizados a la hora de modelar software. UML se compone principalmente de diagramas estructurales y de comportamiento.

Además, industrias de todo tipo han ido incorporando parte de UML a su kit de herramientas a la hora de representar su negocio. Aunque en los últimos años el uso de todas las características de UML ha ido disminuyendo, hay diagramas que se destacan por su vigencia y se mantienen como parte esencial del modelado orientado a objetos, como —por ejemplo— diagrama de clases, diagrama de secuencia, diagrama de actividades, diagrama de componentes y diagrama de despliegue.

2.2.1. Diagrama de clases

El diagrama de clases (Rumbaugh, J., Jacobson, I., Booch, G., 2004; Fowler, M., Scott, K. 1999; Fowler, M., 2004) permite modelar los conceptos clave del dominio de la aplicación o reglas de negocio como clases y relaciones entre clases. Es decir, las entidades y relaciones clave en los que se expresa el dominio de la aplicación se traducen a clases y relaciones entre clases, respectivamente. De igual forma, los nuevos conceptos que surgen durante el diseño detallado, tales como interfaces, clases de implementación, gestores de datos, conectores…, son casi siempre representados en un diagrama de clases como clases. De esta forma, y como se puede apreciar en la figura 2.1, los componentes básicos de un diagrama de clases son las clases y todas las posibles relaciones existentes entre estas, tales como generalización-especialización, agregación-composición, asociación, uso y realización.

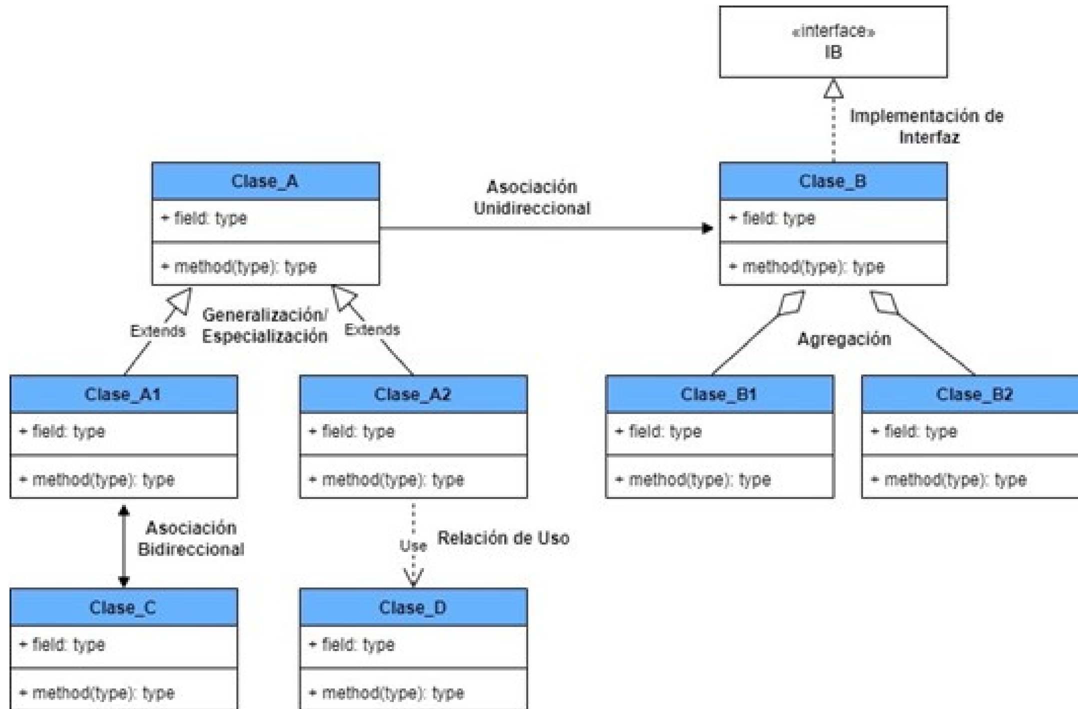


Figura 2.1. Clases y relaciones entre clases en un diagrama de clases en UML

La clase define la estructura y el comportamiento de los objetos que son instancias de esa clase. En los lenguajes orientados a objetos, cada objeto es una instancia (construcción concreta) de una clase. En otras palabras, la clase es la descripción de la estructura y el comportamiento que caracteriza a todos los objetos copias de esa clase.

Entre las clases se establecen diferentes tipos de relaciones; las más utilizadas son las siguientes:

Generalización-especialización. La generalización-especialización (Rumbaugh, J., Jacobson, I., Booch, G., 2004; Fowler, M., Scott, K. 1999; Fowler, M., 2004) es la relación entre una clase y una o más versiones refinadas de esa clase. La clase que se refina se denomina superclase, y la refinada, subclase. La generalización es también una relación transitiva y antisimétrica. La herencia es un mecanismo de implementación de la generalización-especialización en un lenguaje computacional. La aplicación más importante de la herencia es la simplificación conceptual que nos permite hacer en nuestro problema al reducir el número de características independientes y servir como mecanismo para formar complejas estructuras de datos por niveles de construcción, desde más abstractos hasta más concretos. Por ejemplo, en el dominio de aplicación de un sistema de

reservaciones y ventas de una aerolínea (ver figura 2.2), tanto un itinerario sencillo como un itinerario múltiple son tipos de itinerario; de igual forma, una tarjeta de crédito como una tarjeta de débito son tipos de tarjeta bancaria. Por lo tanto, la entidad "itinerario" podría ser vista como una superclase, mientras que las entidades "itinerario sencillo" e "itinerario múltiple", como las subclases que heredan todos los atributos y métodos declarados en la superclase "itinerario". En este mismo tenor, las entidades "tarjeta de crédito" y "tarjeta de débito" pueden ser consideradas tipos de "tarjeta bancaria"; esta última fungiría como superclase.

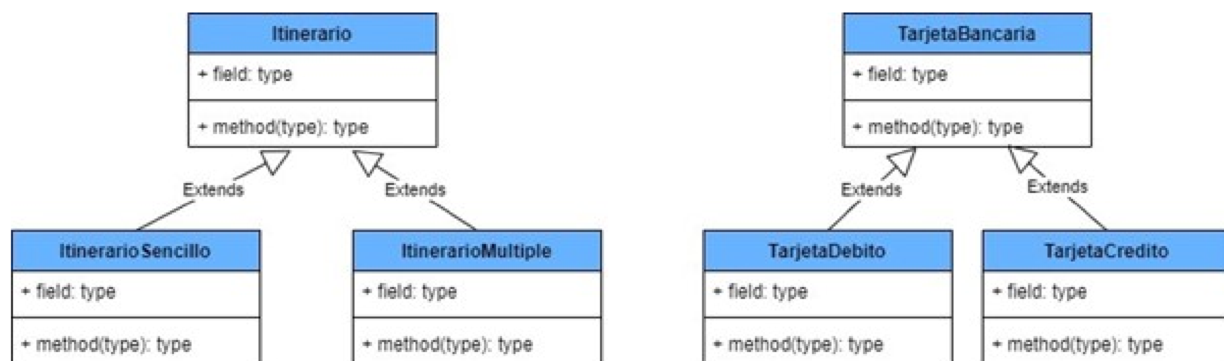


Figura 2.2. Ejemplos de relaciones de generalización-especialización en el dominio de un sistema de reservaciones y vuelos de una aerolínea

Agregación. La agregación (Rumbaugh, J., Jacobson, I., Booch, G., 2004; Fowler, M., Scott, K. 1999; Fowler, M., 2004), también conocida como relación *todo-parte*, expresa que un objeto "todo" es un agregado de otros objetos "parte". Sin embargo, no implica que el "todo" sea una entidad en la cual las "partes" sean fundamentales por su esencia o funcionamiento. En general, la agregación permite definir nuevos objetos ensamblando o componiendo objetos de otras clases, sin necesidad de recurrir a la generalización-especialización. La agregación de objetos puede ser construida, definida o variada dinámicamente, en tiempo de ejecución, mediante campos con los cuales los objetos compuestos hacen referencia a objetos componentes. En el contexto del ejemplo del sistema de reservaciones y ventas de una aerolínea, podríamos considerar la entidad "reservación de vuelo" como un contenedor o agregado constituido por partes más elementales, como "itinerario de vuelo", "datos del pasajero" y "datos de pago" (ver figura 2.3).

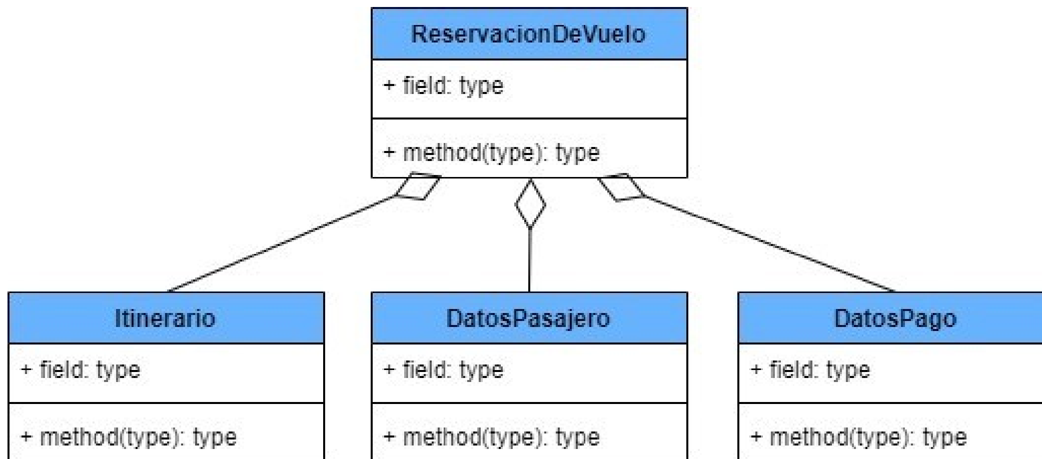


Figura 2.3. Ejemplos de relaciones de agregación en el dominio de un sistema de reservaciones y vuelos de una aerolínea

Asociación. Las asociaciones (Rumbaugh, J., Jacobson, I., Booch, G., 2004; Fowler, M., Scott, K. 1999; Fowler, M., 2004) representan relaciones entre instancias de clases. Una asociación describe un grupo de vínculos con estructura y semántica comunes. Regresando a nuestro ejemplo del sistema de reservaciones y ventas de una aerolínea (ver figura 2.4), ejemplos de asociaciones podrían ser "una reservación de vuelo que genera un comprobante de reservación" o "un pago modifica el status de la reservación de vuelo".

El nombre de una asociación suele expresar cierto sentido de recorrido de la asociación; por ejemplo, "genera" conecta a "reservación de vuelo" con "comprobante de reservación", mientras que "modifica" conecta a "pago" con "reservación de vuelo". La multiplicidad de una asociación se refiere al número posible de instancias de una clase que pueden estar relacionadas con una o más instancias de la otra clase asociada. La multiplicidad limita el número de objetos relacionados.

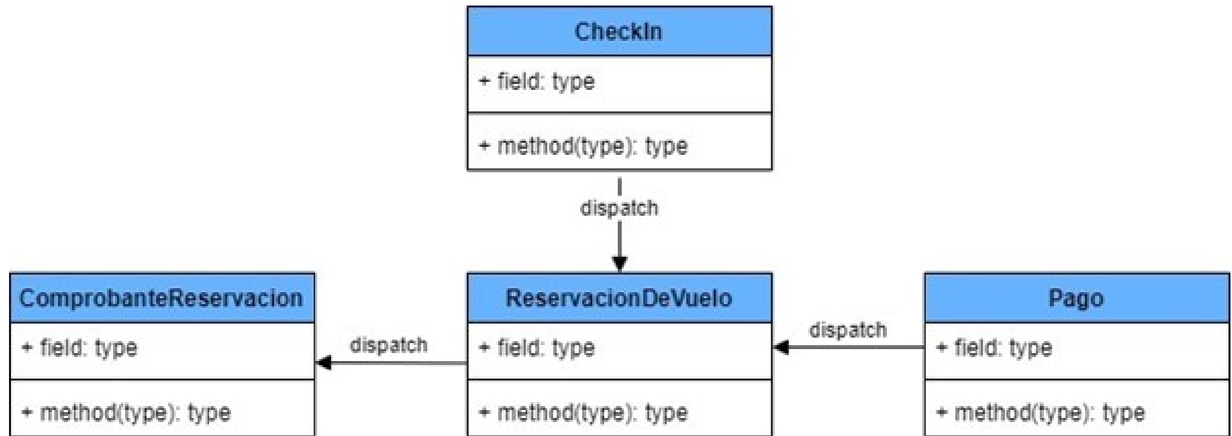


Figura 2.4. Ejemplos de relaciones de asociación en el dominio de un sistema de reservaciones y vuelos de una aerolínea

Uso o dependencia. La relación de uso o dependencia (Fowler, M., Scott, K. 1999; Fowler, M., 2004) es un tipo de asociación que representa la relación de servicio entre dos objetos instancias de dos clases diferentes. Es decir, un objeto instancia de una clase usa a otro objeto instancia de otra clase para llevar a cabo parte de su funcionalidad. En este tipo de relación, un objeto tiene el papel de cliente, y el otro, el de servidor. El objeto cliente es aquel que solicita el servicio, mientras que el objeto servidor es el proveedor del servicio solicitado. En una relación de uso o dependencia un cambio en la clase proveedora de servicio puede afectar la funcionalidad de la clase que requiere o solicita el servicio. Retomando de nuevo el ejemplo del sistema de reservaciones y ventas de una aerolínea (ver figura 2.5), podríamos pensar en las siguientes relaciones de uso o dependencia: “la expedición de un pase de abordar requiere que exista una reservación de vuelo”, o “la ejecución del pago de la reservación de vuelo requiere que se use la tarjeta bancaria”. Nótese que, en el primer ejemplo, la clase “pase de abordar” es la clase cliente, mientras que la clase “reservación de vuelo” es la clase proveedora de servicios. Por otra parte, en el segundo ejemplo, la clase “pago” tiene el papel de clase cliente, mientras que la clase “tarjeta bancaria” funge como clase proveedora de servicios.

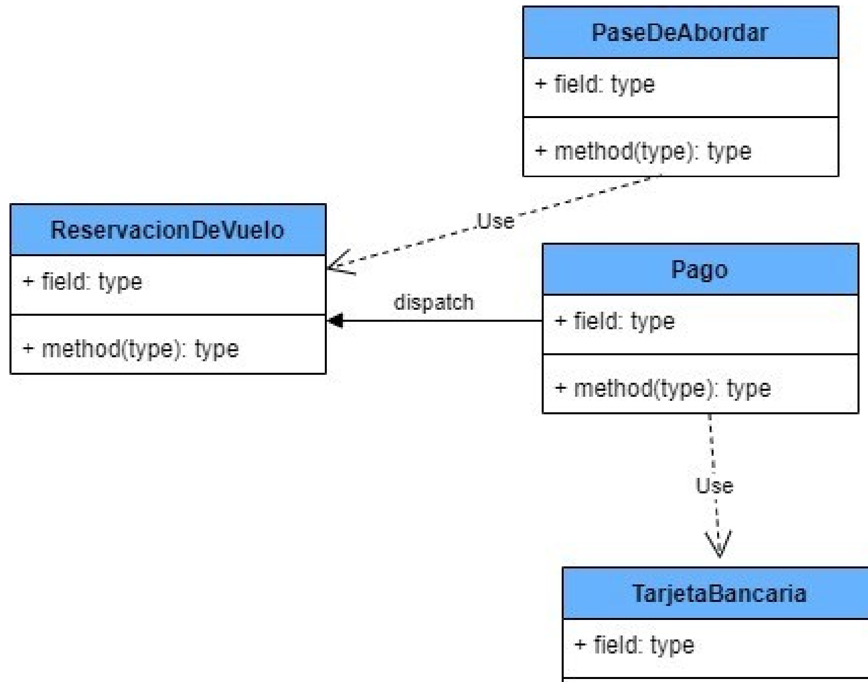


Figura 2.5. Ejemplos de relaciones de uso en el dominio de un sistema de reservaciones y vuelos de una aerolínea

Implementación de interfaz. Una interfaz de implementación (Fowler, M., Scott, K. 1999; Fowler, M., 2004) representa un conjunto de descripciones de operaciones (nombre de la operación, valor que retorna, número y tipo de parámetros que recibe), el cual puede ser utilizado por una clase para especificar los servicios (métodos) que proporcionará. Cuando una clase implementa una interfaz, se compromete a implementar cada una de las operaciones declaradas en dicha interfaz, ya que en una interfaz de implementación las operaciones solo son declaradas, no implementadas, por lo que le corresponde a la clase implementar las mismas. La relación implementación de interfaz puede verse como una alternativa a la herencia, y resulta de gran valor en los lenguajes de programación orientada a objetos que solo soportan la herencia simple o lineal (por ejemplo, Java), al permitir emular la semántica de lenguajes de programación orientada a objetos que soportan la herencia múltiple. Un ejemplo de implementación de interfaz en nuestro dominio de aplicación del sistema de reservaciones y ventas de una aerolínea (ver figura 2.6) viene dado al considerar que una “reservación de vuelo” debe implementar la interfaz “reservación”, así como que tanto las clases “tarjeta de crédito” y “tarjeta de débito” implementan la interfaz

“tarjeta bancaria”. Nótese que estas dos relaciones también podrían resolverse mediante una generalización-especialización.

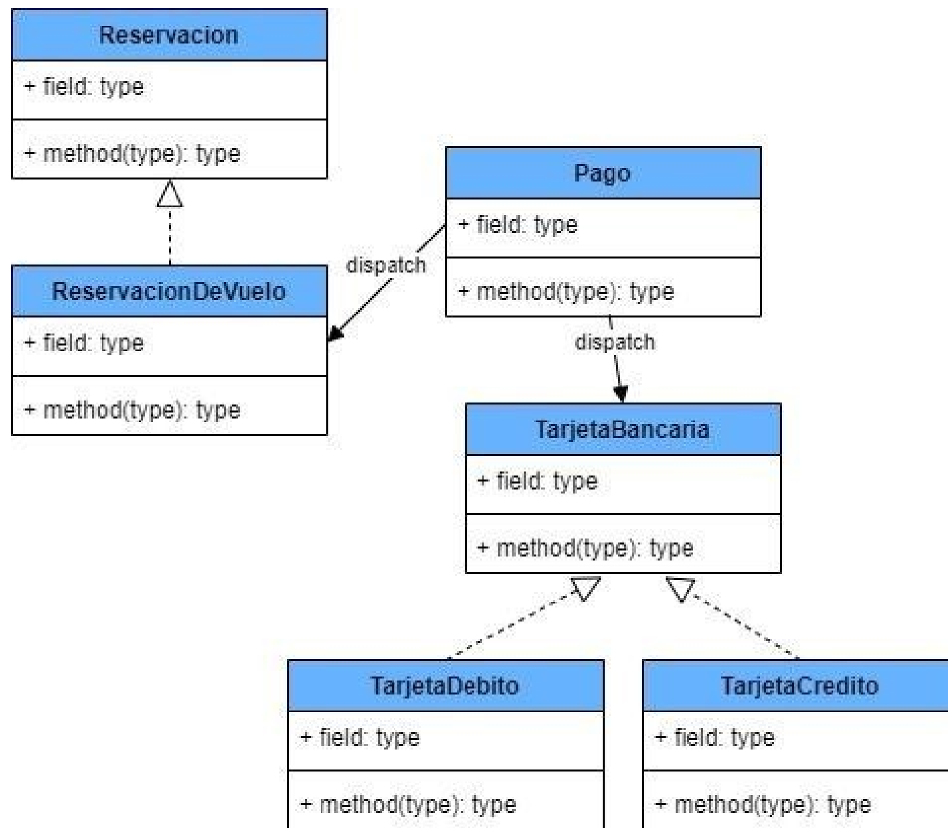


Figura 2.6. Ejemplos de relaciones interfaz de implementación en el dominio de un sistema de reservaciones y vuelos de una aerolínea

La figura 2.7 muestra la representación global en UML de cada una de las relaciones antes expuestas, ejemplificadas en el dominio de aplicación de un sistema de reservaciones y ventas de una aerolínea.

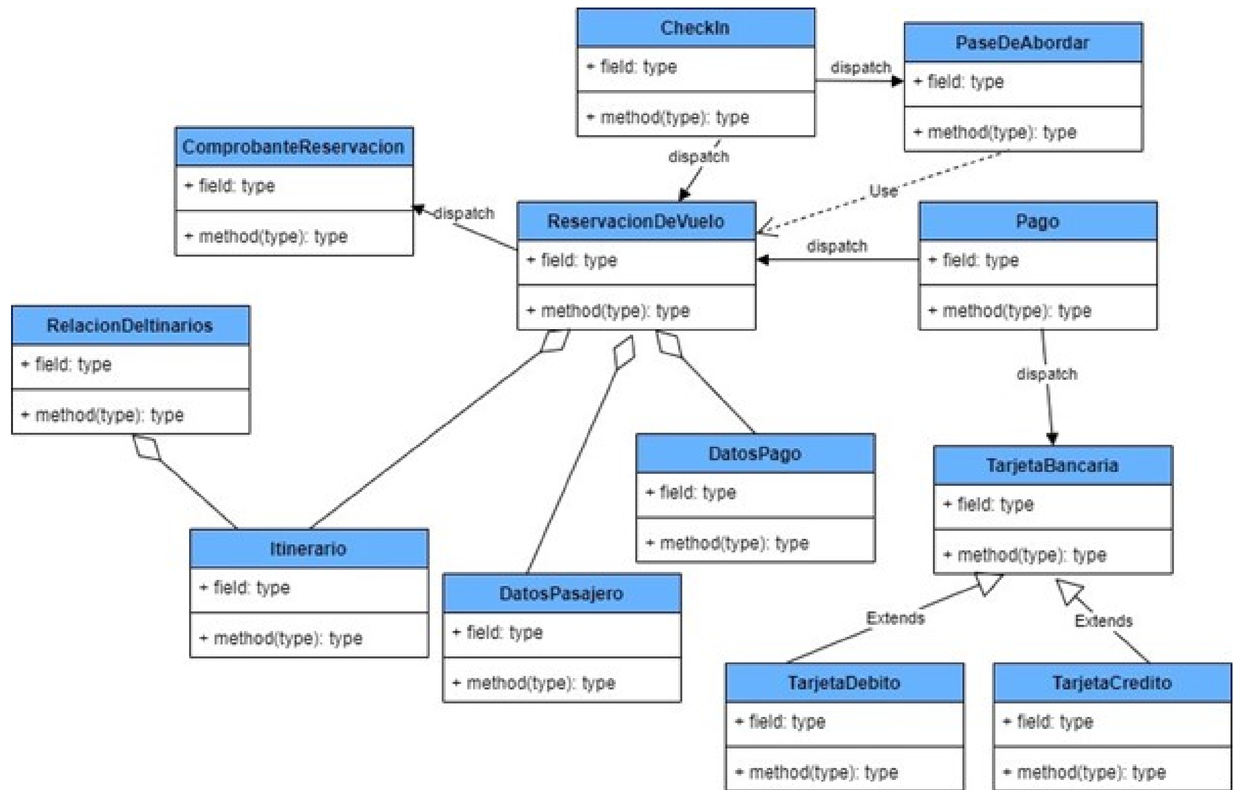


Figura 2.7. Diagrama de clases en UML que ilustra los diferentes tipos de relaciones entre clases identificadas durante el desarrollo de un sistema de reservaciones y ventas de una aerolínea

Para ejemplificar, desde otro dominio de aplicación, el modelado basado en diagramas de clases en UML, analicemos el siguiente fragmento correspondiente a la descripción del dominio de aplicación o reglas de negocio que guiarán el desarrollo de una herramienta de simulación bioinformática dedicada al modelado, simulación y experimentación in silico de vías de señalización intracelular de nombre Big Data-Cellulat (González-Pérez P.P., Cárdenas-García M., 2018; González-Pérez P.P., Cárdenas-García M., 2019):

"[…]Una vía de señalización intracelular describe las interacciones que ocurren entre elementos de señalización, tales como receptores transmembrana, proteínas, enzimas, factores de transcripción y genes. Los receptores transmembrana, las enzimas y los factores de transcripción son tipos de proteínas complejas. Además, otros tipos de proteínas que participan en las vías de señalización intracelular son las proteínas adaptadoras, las proteínas cinasas y las proteínas G, entre otras muchas familias. El

comportamiento de la vía de señalización intracelular viene dado por las interacciones que ocurren entre los elementos de señalización que la integran, siendo estas interacciones de dos tipos fundamentales: activación e inhibición[…]”.

Debido a que nuestra intención es centrarnos en el modelado basado en diagramas UML —y no, en la gestión de los requerimientos—, asumiremos que ya los requerimientos funcionales del sistema Big Data-Cellulat fueron recolectados, analizados, especificados y verificados/validados y, como resultado, ya se cuenta con una declaración del alcance del sistema de software a desarrollar. Dada esta asunción, la tabla 2.1 lista las principales clases identificadas como parte del dominio de la aplicación, así como las posibles relaciones existentes con otras clases. Nótese que la tabla 2.1 no incluye clases que representen aspectos de implementación, tales como interfaces gráficas de usuario, gráficos, controladores, etcétera. De forma complementaria, las figuras 2.8 y 2.9 ilustran dos diagramas de clases correspondientes al modelado de las entidades y relaciones descritas en la tabla 2.1.

Tabla 2.1. Principales clases identificadas en el dominio de la herramienta de simulación computacional Big Data-Cellulat

Clase	Tipo de relación existente	Clase relacionada
Estructura celular	-	-
Tejido celular	Generalización-especialización	Estructura celular
Célula	Generalización-especialización	Estructura celular
Compartimiento intracelular	Generalización-especialización	Estructura celular
Elemento de señalización	Agregación	Vía de señalización
Reactante	Generalización-especialización	Elemento de señalización
Reactante	Agregación	Estructura celular

Reacción química	Agregación	Estructura celular
Reacción ordinaria	Generalización-especialización	Reacción química
Reacción instantánea	Generalización-especialización	Reacción química
Reacción temporizada	Generalización-especialización	Reacción química
Gen	Generalización-especialización	Elemento de señalización
Proteína	Generalización-especialización	Elemento de señalización
Proteína cinasa	Generalización-especialización	Proteína
Proteína	Asociación	Proteína
Proteína	Asociación	Gen
Proteína adaptadora	Generalización-especialización	Proteína
Proteína G	Generalización-especialización	Proteína
Proteína compleja	Generalización-especialización	Proteína
Receptor transmembrana	Generalización-especialización	Proteína compleja
Factor de transcripción	Generalización-especialización	Proteína compleja

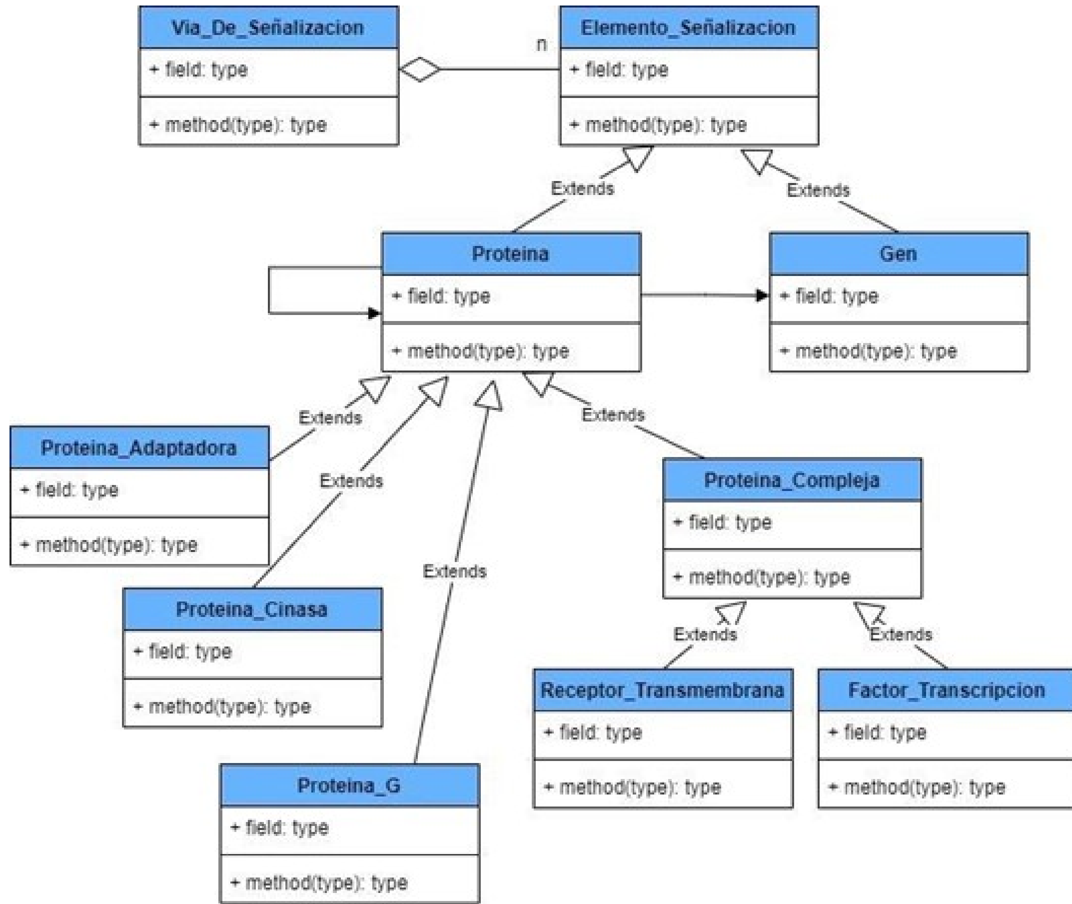


Figura 2.8. Modelado de una vía de señalización intracelular a través de un diagrama de clases en UML

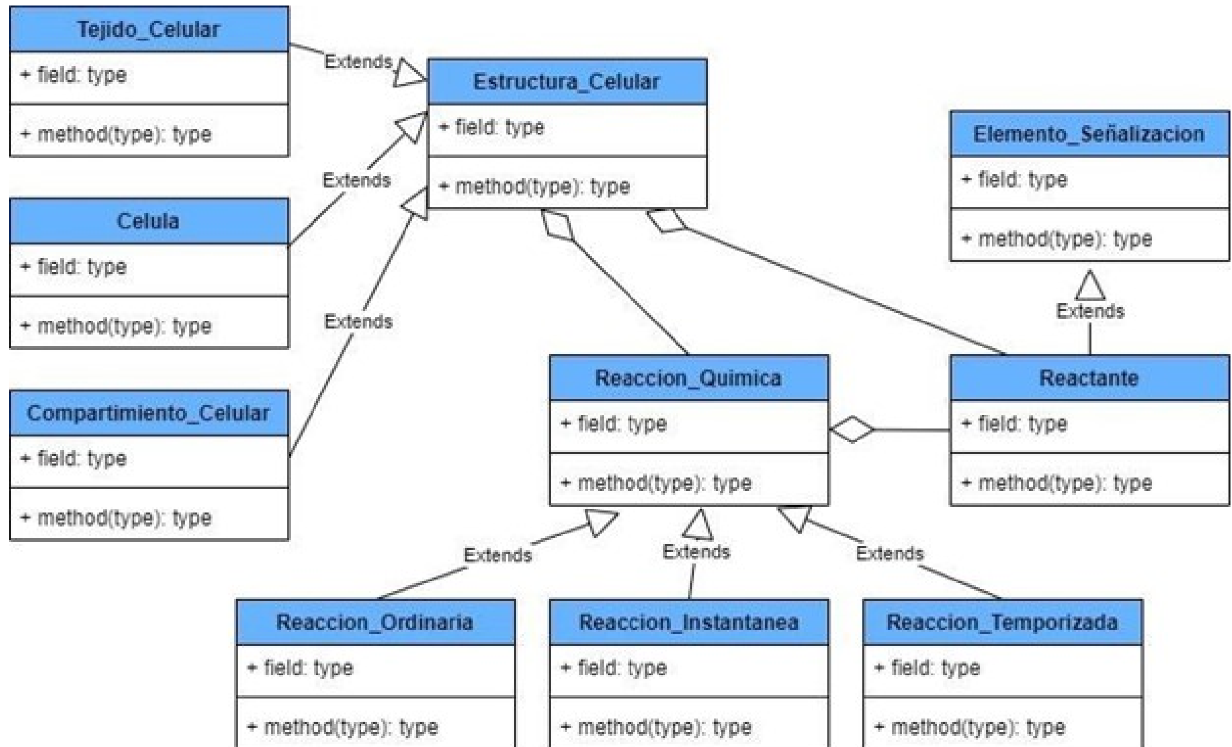


Figura 2.9. Modelado de las estructuras celulares, reacciones químicas y reactantes

2.2.2. Diagrama de secuencia

El diagrama de secuencia (Rumbaugh, J., Jacobson, I., Booch, G., 2004; Fowler, M., Scott, K. 1999; Fowler, M., 2004) modela el comportamiento del sistema a través de la interacción entre los objetos que lo conforman. Esto es, describe las secuencias de paso de mensajes entre los objetos que implementan el comportamiento de determinada funcionalidad del sistema.

Los principales componentes de un diagrama de secuencia son los objetos, la línea de vida de los objetos y los mensajes que se pasan los objetos. En consecuencia, se usan tres símbolos básicos en un diagrama de secuencia:

- Rectángulos, que representan objetos instancia de clases relacionadas entre sí mediante asociación, uso o agregación.
- Líneas verticales, comúnmente discontinuas, cada una de las cuales se inicia en la base del rectángulo y representa el tiempo de vida del objeto (línea de vida del objeto) durante la interacción.
- Flechas horizontales continuas y discontinuas, que representan mensajes y van desde la línea de vida de un objeto a la línea de vida de otro. Las

flechas horizontales continuas representan paso de mensaje desde un objeto originador de la petición hasta un objeto receptor, encargado de atender dicha petición, mientras que las flechas discontinuas representan el retorno o la respuesta que produce el objeto receptor a la petición recibida.

Además de lo anterior, un rectángulo vertical a lo largo de la línea de vida del objeto puede utilizarse como barra de activación para representar la duración de la activación del objeto, casi siempre, desde el momento en que el objeto recibe la petición hasta el momento cuando retorna el resultado.

El orden en que se dan los mensajes es de arriba hacia abajo y, casi siempre, de izquierda a derecha. A cada mensaje se asocia una etiqueta que corresponde al nombre del mensaje. La figura 2.10 ilustra los símbolos antes descritos en un diagrama de secuencia genérico, el cual muestra las diferentes interacciones que hay entre los objetos A, B, C, D y E.

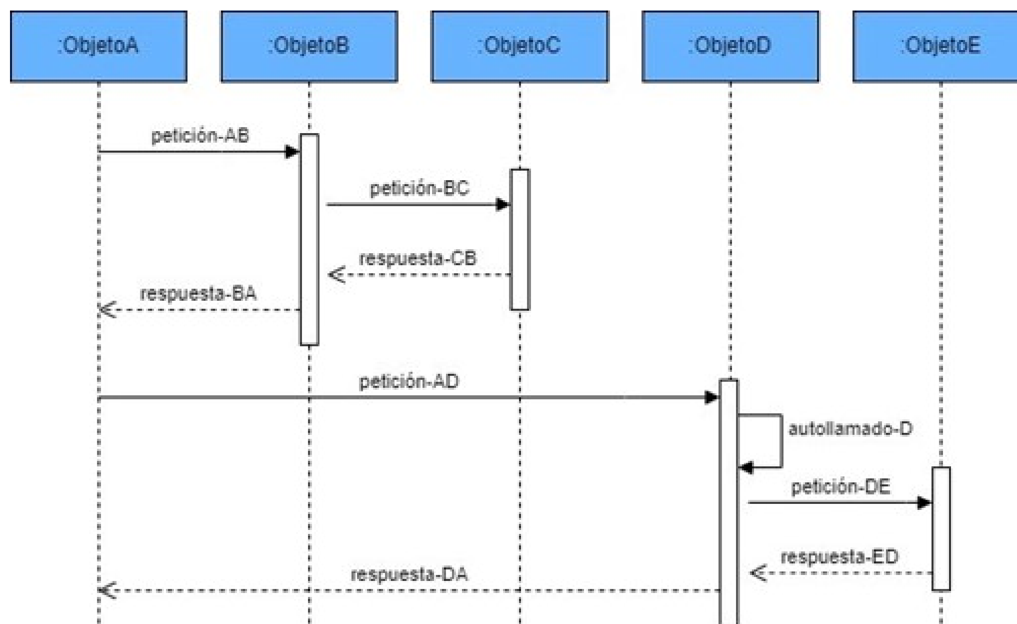


Figura 2.10. Diagrama de secuencia genérico que ilustra el uso de los principales símbolos que lo integran

Volviendo al caso de estudio del sistema de reservaciones y ventas de una aerolínea, la figura 2.11 muestra el diagrama de secuencia correspondiente a la funcionalidad “gestión de itinerarios”. Nótese que —además de las clases ya conocidas, tales como “itinerario” y “relación de itinerarios” (ver figura

2.7), cuyos objetos instancia se representan en el diagrama de secuencia—aquí también se incluyen nuevos objetos instancia de clases que representan interfaces de usuario y otras clases del dominio de la aplicación, las cuales fueron emergiendo en fases avanzadas del modelado o diseño de la solución.

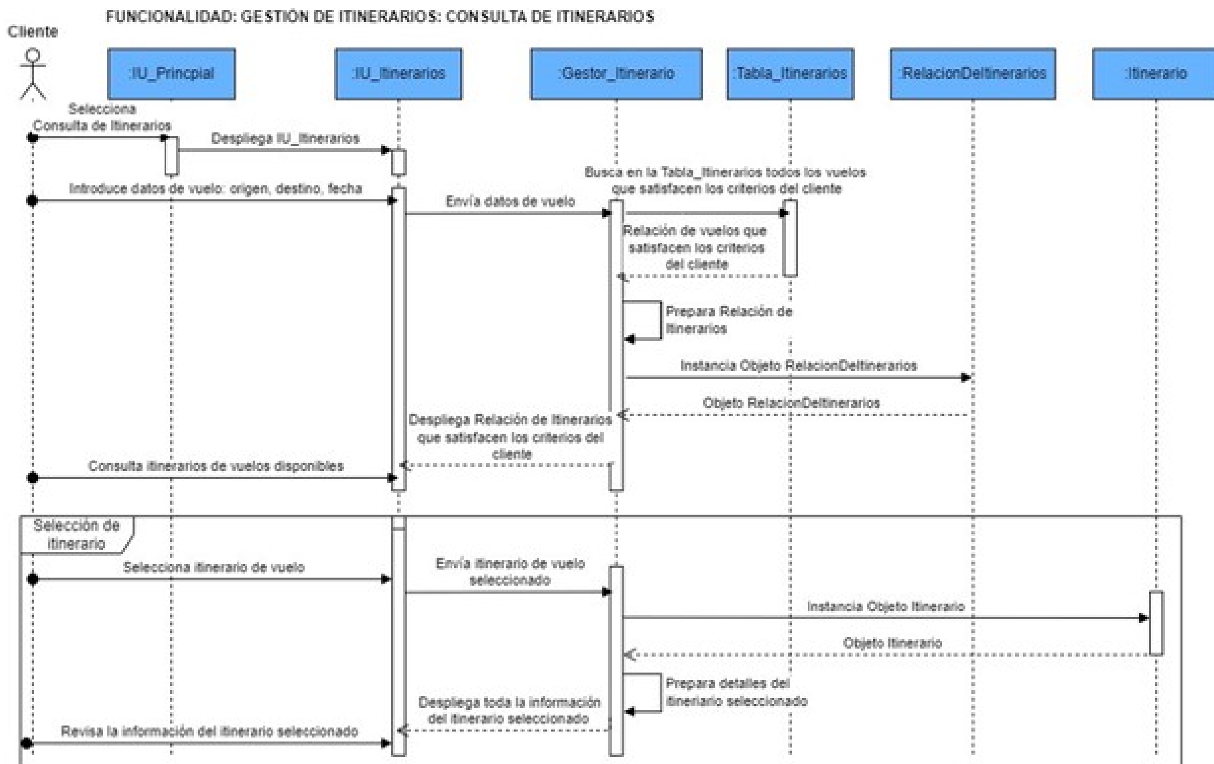


Figura 2.11. Diagrama de secuencia correspondiente a la funcionalidad “gestión de itinerarios” del sistema de reservaciones y ventas de una aerolínea

Por otra parte, en la figura 2.12 se ilustra el diagrama de secuencia correspondiente a una vía de señalización celular genérica. Nótese que las clases de las cuales son instancias los objetos que aquí interactúan ya fueron modeladas en el diagrama de clases ilustrado en la figura 2.8. Una característica que logra apreciarse de forma inmediata al observar este diagrama de secuencia es que no se indican los mensajes de retorno o respuesta desde el objeto receptor hacia el objeto originador, ya que estos no son necesarios. Es decir, por ejemplo, el ligando se une y activa al receptor sin que sea necesario que el receptor retorne ningún valor o notificación al ligando; lo mismo ocurre con las restantes interacciones, en las cuales la función del mensaje o interacción procedente del objeto originador ha sido

únicamente generar un efecto en el objeto receptor (proteína o factor de transcripción).

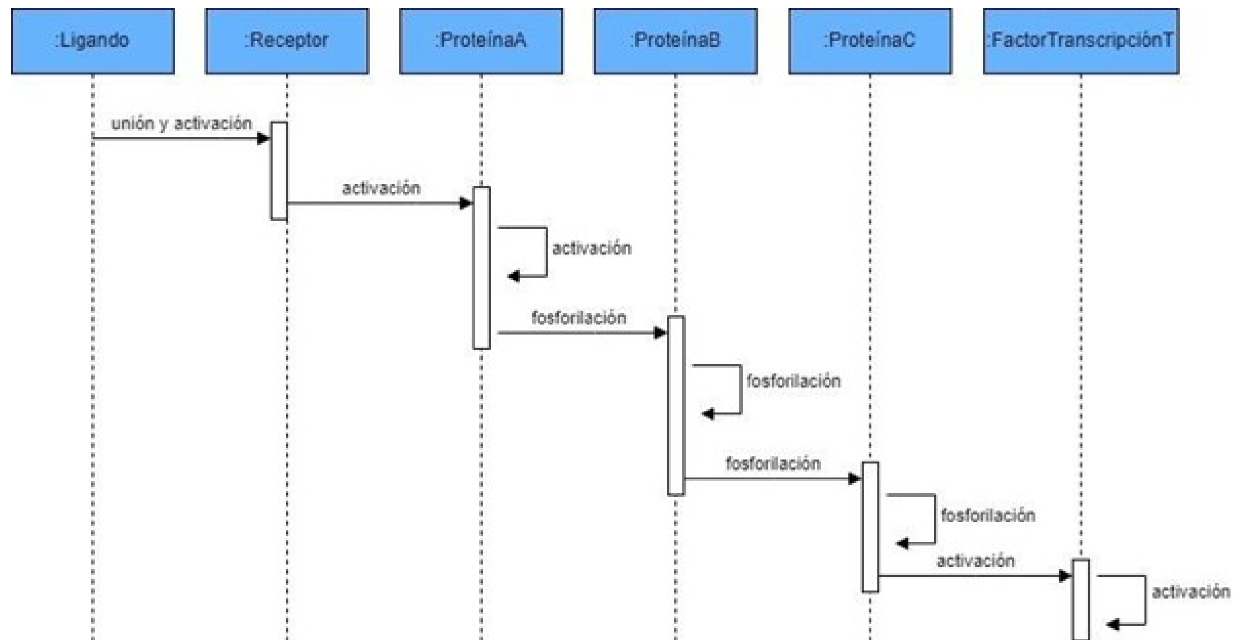


Figura 2.12. Diagrama de secuencia correspondiente a una vía de señalización celular genérica

2.2.3. Diagrama de actividades

Al igual que el diagrama de clases y el diagrama de secuencia, el diagrama de actividades (Rumbaugh, J., Jacobson, I., Booch, G., 2004; Fowler, M., Scott, K. 1999; Fowler, M., 2004) tiene un papel clave durante el modelado/diseño de un sistema de software. Mientras que el diagrama de clases captura los aspectos estáticos/estructurales y el diagrama de secuencia refleja los aspectos de interacción/comunicación entre los objetos instancia de las clases, el diagrama de actividades modela los estados de ejecución del cómputo (la ejecución del sistema) y el flujo de trabajo. El diagrama de actividades posee algunas semejanzas con el diagrama de flujo de datos. La principal diferencia entre ambos diagramas reside en que los diagramas de flujo de datos se limitan comúnmente a procesos secuenciales, mientras que los diagramas de actividades pueden manejar, además, procesos paralelos.

Una característica clave de los diagramas de actividades es el modelado de los hilos concurrentes, los cuales representan actividades que pueden realizar

de manera confluyente los diferentes objetos. El diagrama de actividades incluye los siguientes símbolos:

- Rectángulos de bordes redondeados, para representar estados de actividad. La interpretación del término estado de actividad depende de la perspectiva o nivel de detalle que ofrezca el diagrama de actividades. Considerando lo anterior, podemos decir que un *estado de actividad* puede representar desde una determinada tarea que debe ser ejecutada, ya sea por el propio sistema de software o por algún subsistema externo, hasta la ejecución de un método de una clase. En una perspectiva orientada a objetos, un estado de actividad suele representar la interacción de dos o más objetos para llevar a cabo la funcionalidad o funcionalidades relacionadas que encierra dicha actividad.
- Flechas, para representar transiciones simples de terminación. Una transición representa el fin de una actividad y el paso de la ejecución al siguiente estado de actividad en el diagrama. Una transición ocurre cuando un estado de actividad ha finalizado la ejecución de su cómputo.
- Rombos, para representar bifurcaciones. Una bifurcación establece las dos diferentes ramas que pueden seguirse después de evaluar una condición booleana.
- Barras gruesas horizontales, para representar sincronización. Una bifurcación corresponde a una división o unión de control, la cual se representa por una barra gruesa con varias flechas que llegan o salen. Cada flecha que llega a la barra de sincronización suele corresponder al final de la ejecución de un hilo concurrente.
- Círculos sólidos y círculos circunscritos, para representar el inicio y el fin del flujo de trabajo, respectivamente.

Los diagramas de actividades muestran el flujo de actividades, pero no los objetos que realizan dichas actividades. Podemos considerar que los diagramas de actividades son el punto de partida para el diseño, dado que nos muestran las principales actividades que el sistema debe ejecutar. La figura 2.13 ilustra los símbolos del diagrama de actividades antes descritos en un diagrama de actividades genérico.

En la figura 2.14 puede apreciarse el diagrama de actividades simplificado a nivel superior del sistema de reservaciones y ventas de una aerolínea. En este diagrama de actividades, cada una de las cinco funcionalidades superiores

que integran el sistema — “autenticación”, “consulta de itinerarios”, “reservaciones”, “pagos” y “checkin”— ha sido representada como un único estado de actividad. De forma complementaria, en la figura 2.15 se ilustra el diagrama de actividades correspondiente al refinamiento del estado de actividad “consulta de itinerarios”.

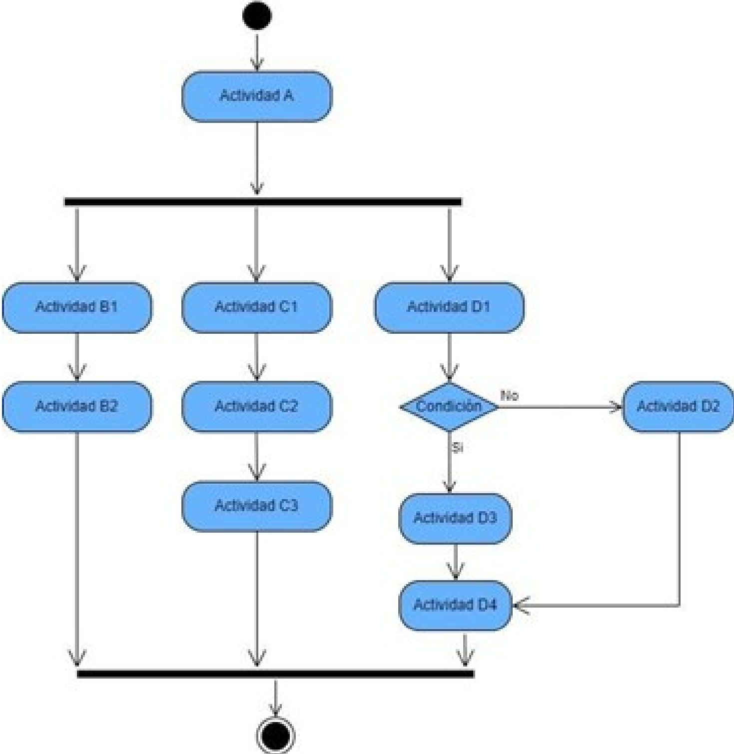


Figura 2.13. Diagrama de actividades genérico que ilustra el uso de los principales símbolos que lo integran

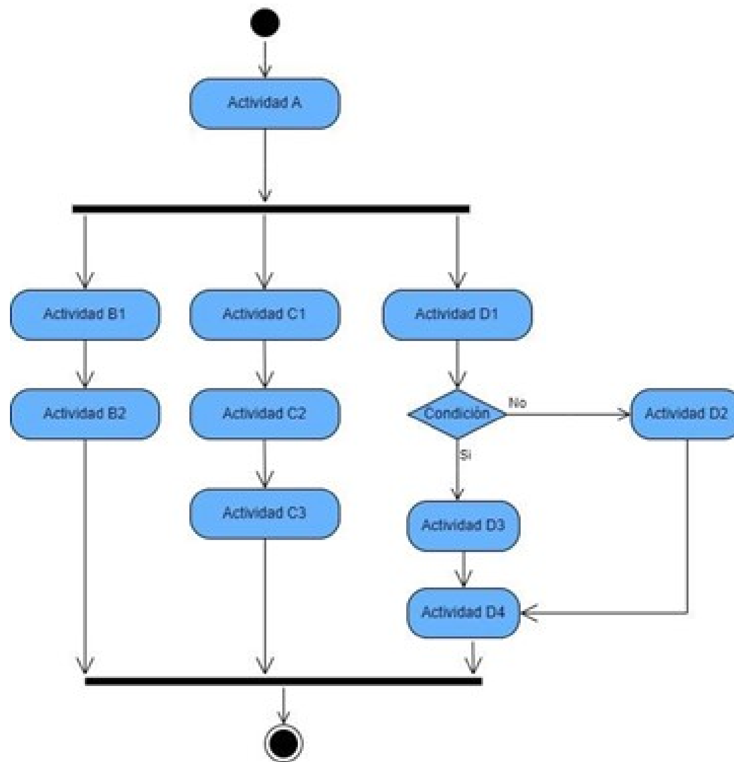


Figura 2.14. Diagrama de actividades simplificado correspondiente al sistema de reservaciones y ventas de una aerolínea

Figura 2.15. Diagrama de actividades de la funcionalidad “consulta de itinerarios” del sistema de reservaciones y ventas de una aerolínea. Nótese que este diagrama constituye un refinamiento del estado de actividad homónimo representado en la figura 2.14

Tornando al caso de estudio de la herramienta de simulación bioinformática Big Data- Cellulat, las figuras 2.16 y 2.17 muestran aspectos relacionados con el flujo de ejecución de la misma. En la figura 2.16 se muestra un diagrama de actividades simplificado de las principales actividades que engloba Big Data-Cellulat, mientras que la figura 2.17 muestra el diagrama de actividades correspondiente a la funcionalidad “gráficos”.

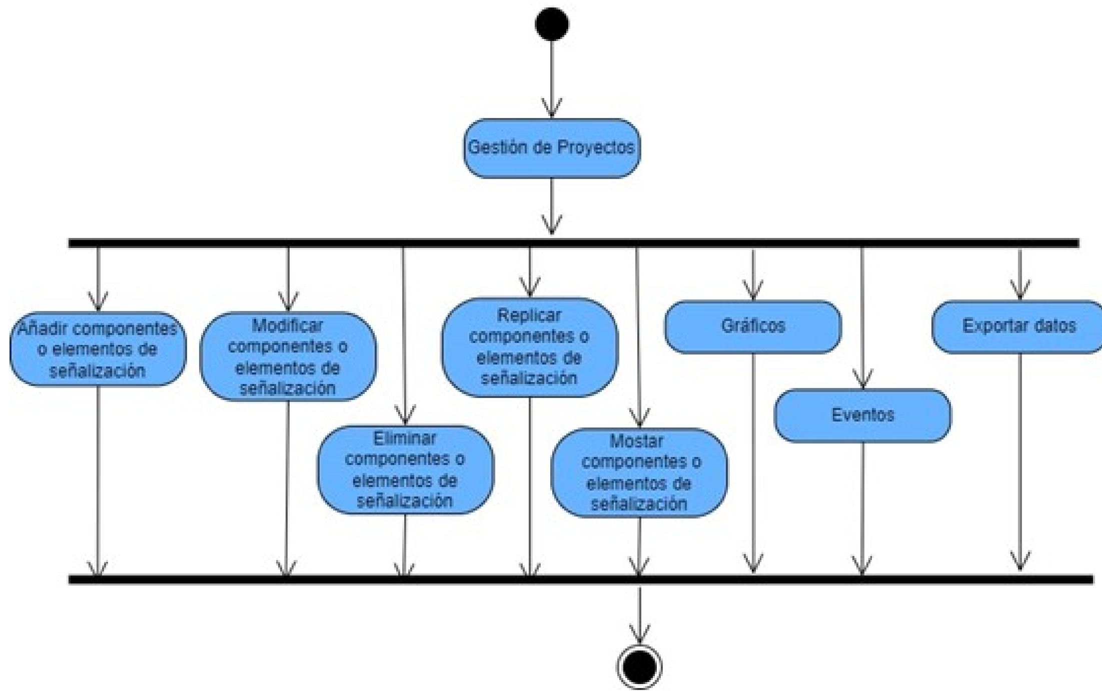


Figura 2.16. Diagrama de actividades simplificado correspondiente a la herramienta de simulación bioinformática Big Data-Cellulat

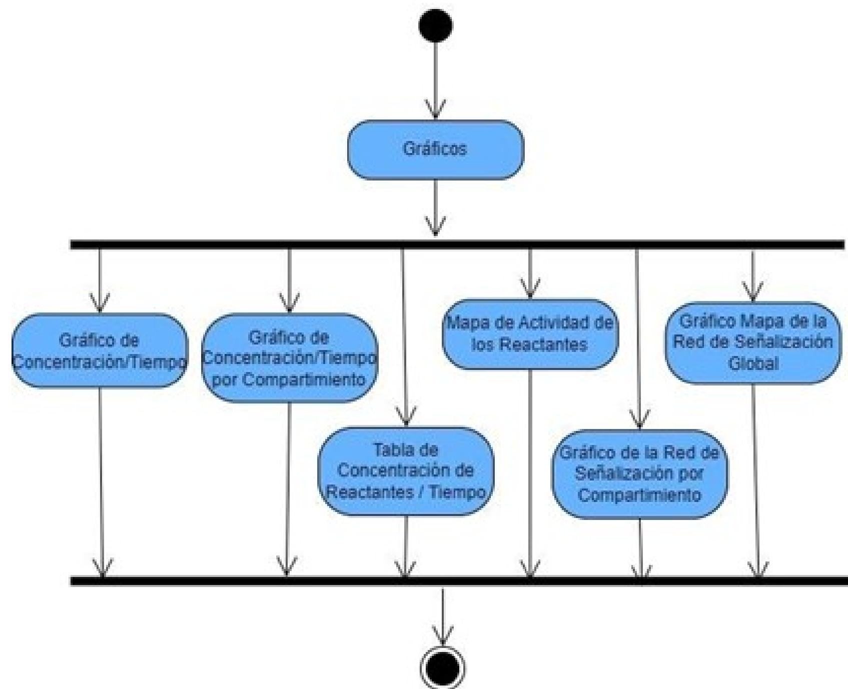


Figura 2.17. Diagrama de actividades correspondiente a la funcionalidad "gráficos" de la herramienta de simulación bioinformática Big Data-Cellulat

2.2.4. Diagrama de paquetes

El paquete de clases es una forma de organizar un grupo de clases estrechamente relacionadas. El paquete de clases funge como contenedor de clases; las clases contenidas en él pueden estar relacionadas a partir de diferentes criterios: aspecto del dominio del problema que modelan, tipo de servicio que proporcionan, tipo de capa lógica donde se ubican, etcétera.

El diagrama de paquetes (Rumbaugh, J., Jacobson, I., Booch, G., 2004; Fowler, M., Scott, K. 1999; Fowler, M., 2004) exhibe las dependencias entre los paquetes de clases que componen el modelo de diseño. El paquete "A" establece una relación de dependencia con el paquete "B", si al menos un elemento del paquete "A" requiere de al menos un elemento del paquete "B". Estos elementos son, por lo general, clases.

Como puede apreciarse en la figura 2.18, los principales elementos de un diagrama de paquetes son los siguientes:

- Paquete
- Relación de dependencia entre paquetes

Nótese en la figura 2.18 las relaciones que se establecen entre los paquetes A, B, C, D y E. El paquete A es el paquete de mayor jerarquía en el modelo; es decir, es el paquete superordinado. Esto significa que elementos del paquete A requieren de elementos ubicados en los paquetes B, C y D, pero ningún paquete en el modelo requiere de elementos contenidos en el paquete A. Por otra parte, el paquete E es el paquete de menor jerarquía en el modelo o paquete subordinado. Es decir, elementos contenidos en el paquete E son requeridos por elementos ubicados en otros paquetes (en este caso, paquetes B y C), pero el paquete E no requiere elementos contenidos en ningún otro paquete.

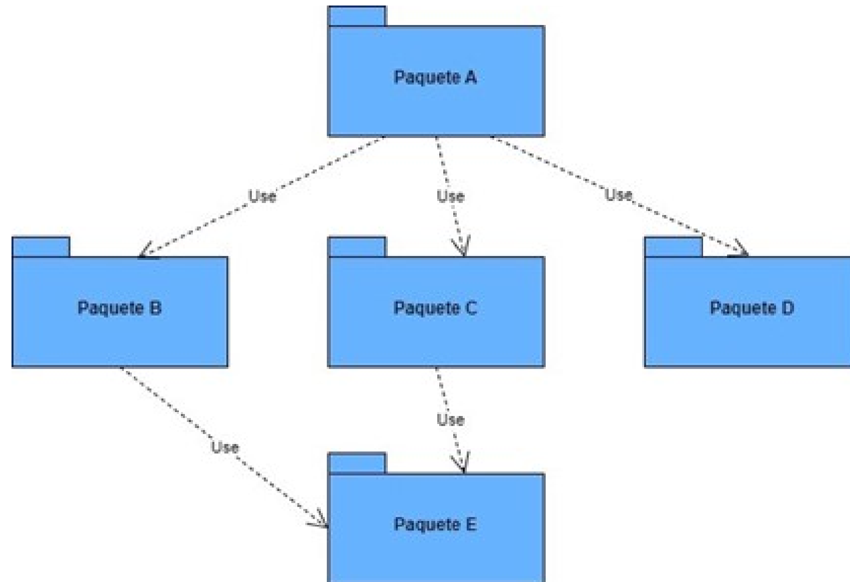


Figura 2.18. Diagrama de paquetes genérico que ilustra los principales componentes que lo integran

Retomando el caso de estudio del sistema de reservaciones y ventas de una aerolínea, las figuras 2.19 y 2.20 ilustran aspectos clave del modelado basado en diagramas de paquetes. La figura 2.19 muestra los paquetes y relaciones entre paquetes considerados en esta fase de modelado. Nótese el rol superordinado del paquete “Módulo Interfaces de Usuario”, el cual establece relaciones de dependencia con los restantes cinco módulos, los cuales representan diferentes aspectos de la lógica de la aplicación o reglas de negocio. De forma complementaria, el paquete “Módulo Itinerarios” tiene el papel de paquete subordinado, ya que solo ofrece servicios a otros paquetes (en este caso, a los paquetes “Módulo Interfaces de Usuario” y “Módulo Reservaciones”), pero no depende de ningún otro paquete.

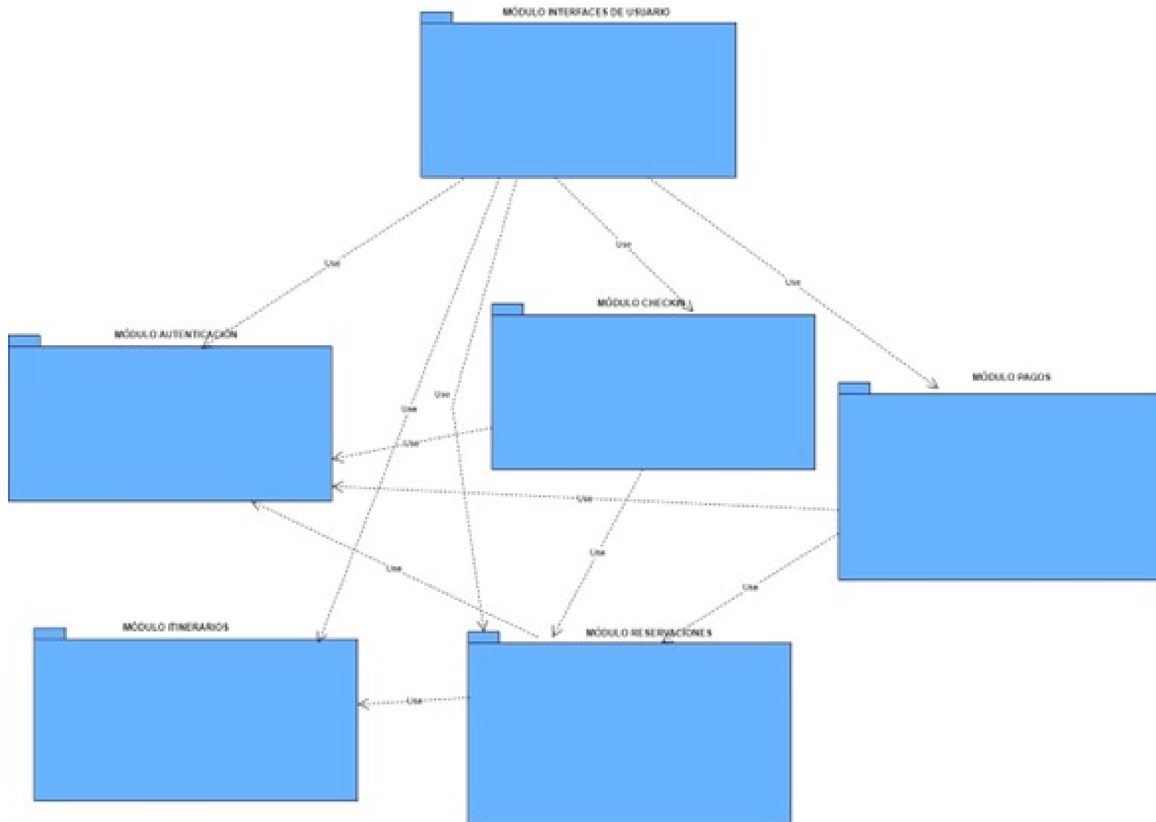


Figura 2.19. Diagrama de paquetes correspondiente al sistema de reservaciones y ventas de una aerolínea

Por su parte, la figura 2.20 corresponde a un modelado mucho más detallado, pues muestra las clases y relaciones entre clases al interior de cada paquete. Nótese que muchas de las clases y relaciones entre clases que aquí se exhiben ya se habían concebido en un primer nivel de modelado ilustrado en la figura 2.7.

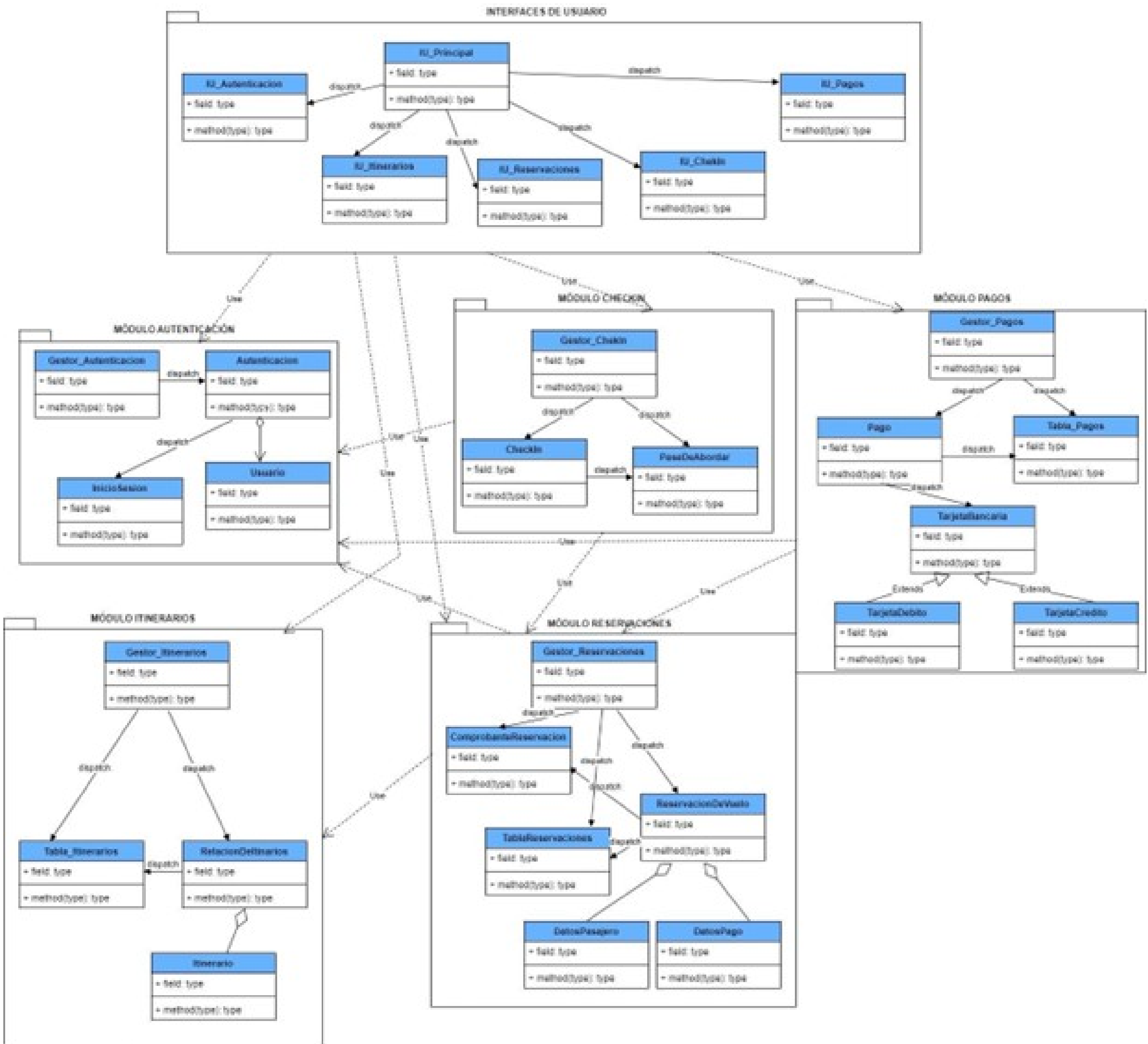


Figura 2.20. Diagrama de paquetes correspondiente al sistema de reservaciones y ventas de una aerolínea; muestra las clases contenidas en cada paquete

Por otra parte, la figura 2.21 muestra el modelado basado en diagramas de paquetes de la herramienta de simulación bioinformática Big Data-Cellulat. Nótese la compatibilidad entre este diagrama de paquetes y el diagrama de actividades ilustrado en la figura 2.16. Obsérvese el papel superordinado del paquete “Interfaces de Usuario”, el cual establece relaciones de dependencia con los restantes paquetes, los cuales representan diferentes aspectos de la lógica de la aplicación o reglas de negocio, tales como “Gestión de Reactantes”, “Gestión de Reacciones”, “Gestión de Simulación”, etcétera. De

forma particular, el paquete "Gestión de Simulación" establece una gran cantidad de dependencias respecto de otros paquetes.

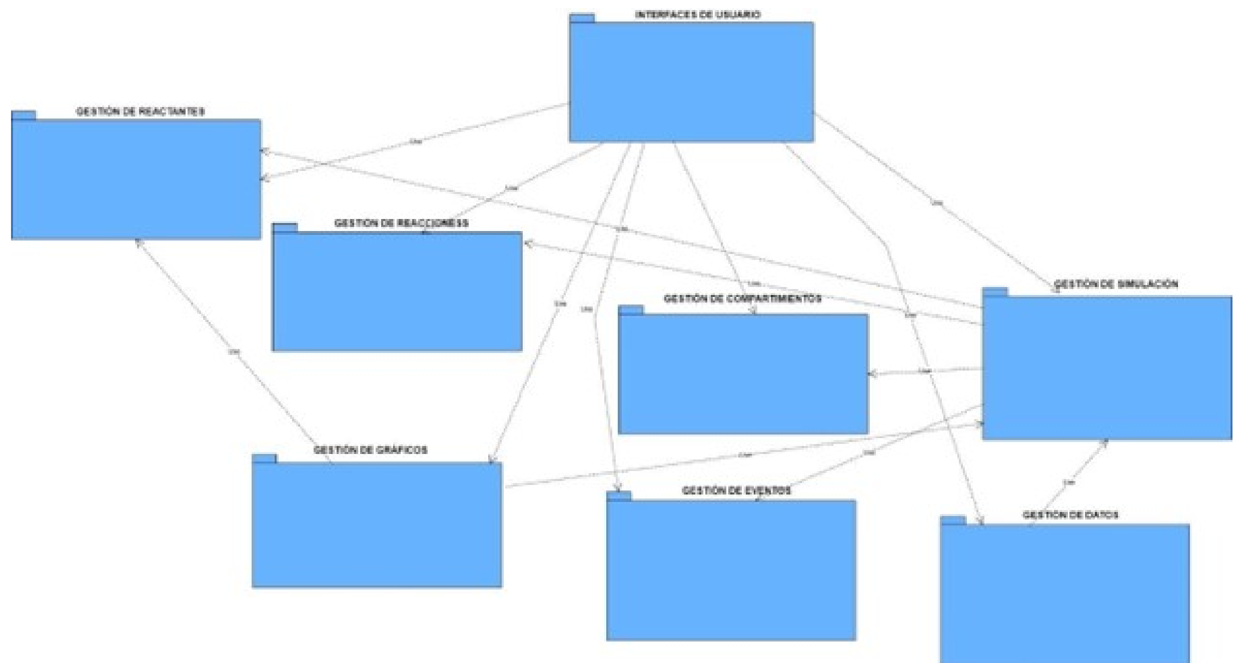


Figura 2.21. Diagrama de paquetes correspondiente a la herramienta de simulación bioinformática Big Data-Cellulat

2.2.5. Diagrama de componentes

Al igual que los diagramas de paquetes, los diagramas de componentes (Rumbaugh, J., Jacobson, I., Booch, G., 2004; Fowler, M., Scott, K. 1999; Fowler, M., 2004) describen la estructura del sistema de software. Sin embargo, a diferencia de los diagramas de paquetes, los cuales organizan los elementos lógicos del sistema de software —tales como clases y relaciones entre clases—, los diagramas de componentes pueden incluir elementos de software, de hardware y subsistemas externos o remotos con los cuales el sistema de software interactúa.

Como puede apreciarse en las figuras 2.22 y 2.23, los elementos más utilizados en un diagrama de componentes son los siguientes:

- Componente. Se representa a través de un rectángulo con dos pestañas en uno de los lados. Como ya indicamos, un componente puede representar aspectos lógicos, físicos o de dependencia externa del sistema de software.

- Interfaz del componente. Es común que a un componente siempre se asocie una interfaz, la cual permite establecer la comunicación con otro componente. La interfaz viene representada por una línea con un círculo no sólido en el extremo libre, mientras que el otro extremo de la línea se une al componente. El extremo de la línea con el círculo no sólido constituye el punto en el cual otros módulos que lo requieran pueden conectarse a dicha interfaz. La interfaz de un componente puede ser vista como el conjunto de operaciones públicas que ofrece dicho componente. De forma complementaria, cuando un módulo requiere conectarse a la interfaz de otro módulo, este hecho se representa con una línea que parte del módulo que requiere el servicio y que finaliza en el otro extremo en un semicírculo, el cual debe acoplarse al círculo no sólido que representa la interfaz del módulo que proporcionará el servicio (ver figura 2.22).
- Relación de dependencia. Aunque semánticamente difiere de la relación que se establece entre dos módulos mediante la interfaz, en muchas ocasiones, por facilidad, en un diagrama de componentes las relaciones entre los componentes se representan con flechas discontinuas que significan "dependencia". Al igual que una "relación de uso" en un diagrama de clases, en una relación de dependencia la flecha parte del componente dependiente y finaliza en el componente del que se depende para ejecutar la tarea (ver figura 2.23).

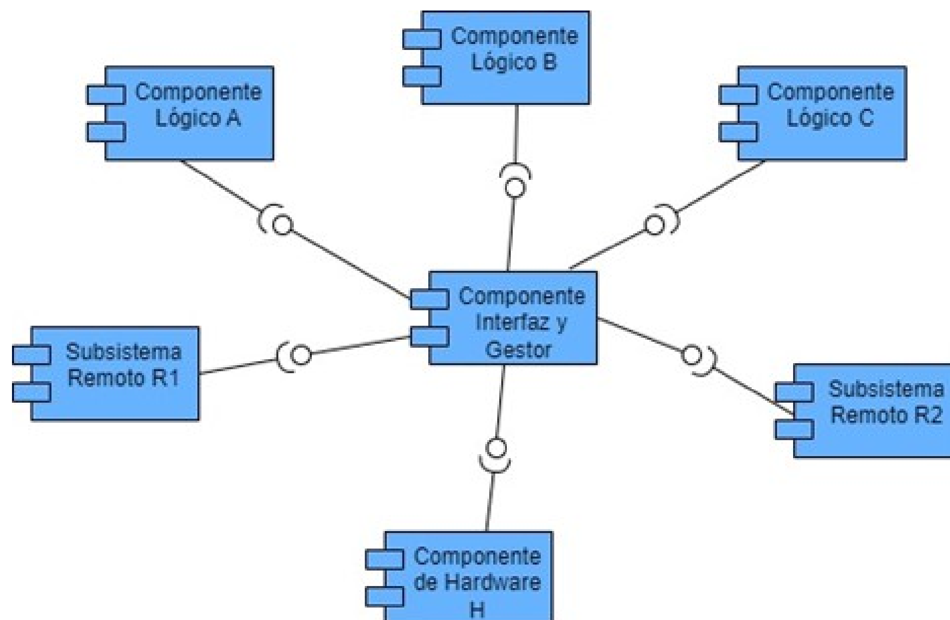


Figura 2.22. Diagrama genérico de componentes que ilustra las relaciones entre los componentes a través del símbolo de interfaz

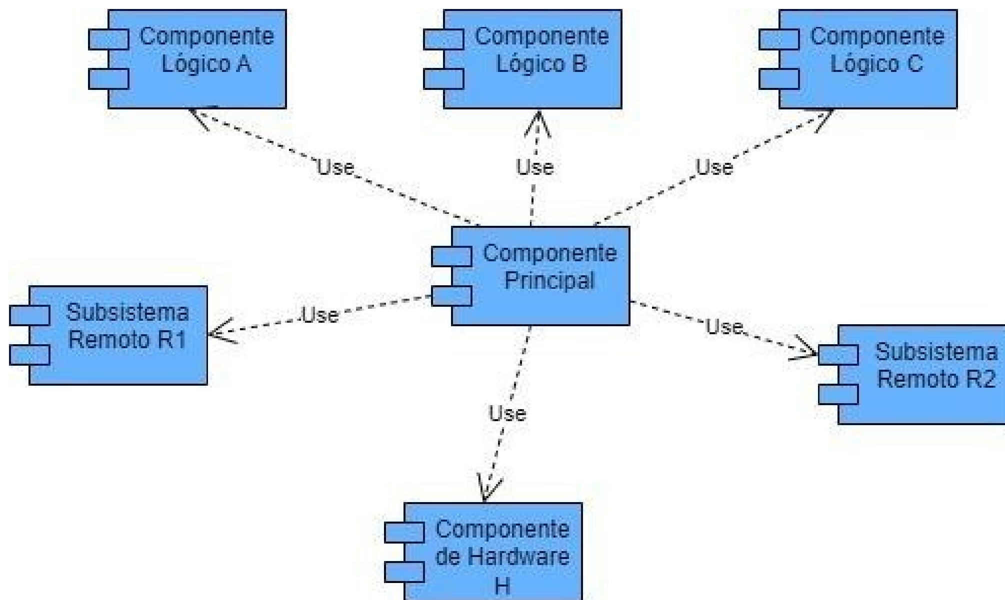


Figura 2.23. Diagrama genérico de componentes que ilustra las relaciones entre los componentes por medio del símbolo de dependencia

Tornando a nuestro caso de estudio del sistema de reservaciones y ventas de una aerolínea, la figura 2.24 ilustra el modelado basado en diagramas de componentes y diagramas de paquetes. Véase que la notación de paquete ha sido utilizada para indicar módulos o componentes en un nivel superior, que engloban otros componentes. Como podemos apreciar en esta figura, el modelo propone una primera aproximación al concepto de estructura o arquitectura de software, al separar y desacoplar componentes que representan aspectos de visualización/interacción (interfaces gráficas de usuario) de componentes que representan aspectos de la lógica de la aplicación y de componentes que corresponden a gestión de bases de datos y subsistemas externos.

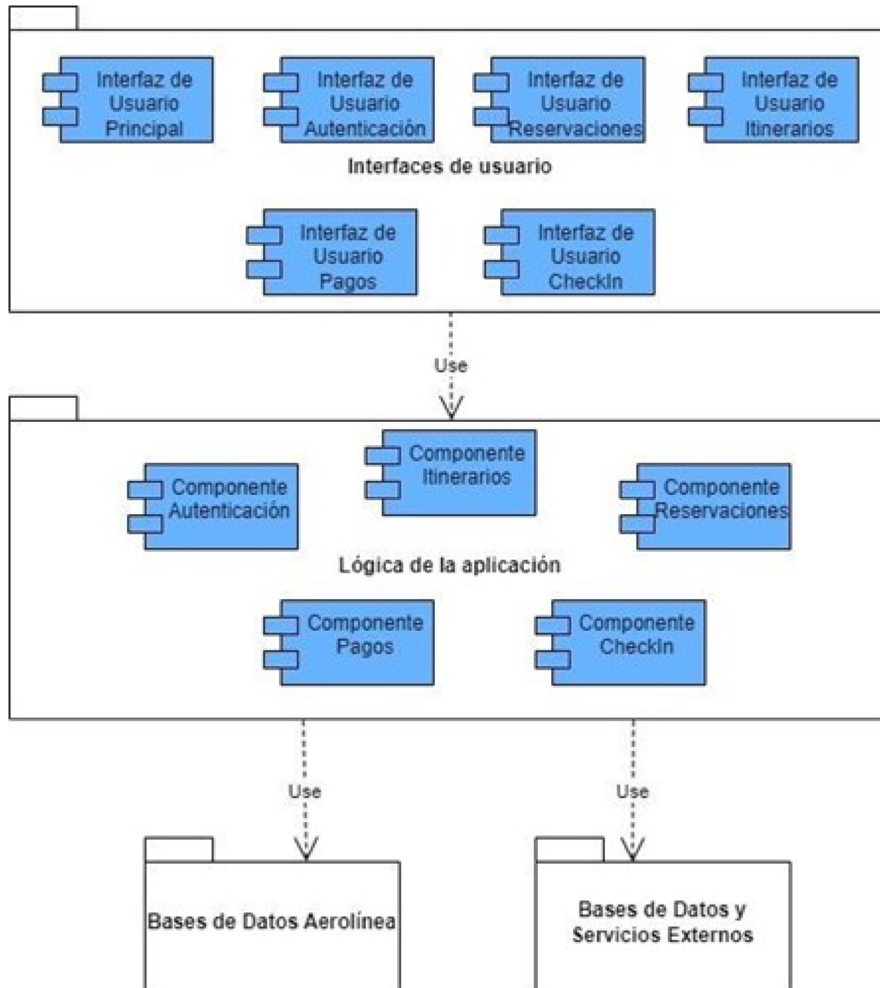


Figura 2.24. Diagrama de componentes correspondiente al sistema de reservaciones y ventas de una aerolínea

Por otra parte, en la figura 2.25 se ilustra el diagrama de componentes correspondiente al caso de estudio de la herramienta de simulación bioinformática Big Data-Cellulat. Aquí nuevamente la notación de paquete ha sido utilizada para indicar módulos o componentes en un nivel superior, que engloban otros componentes. Como puede apreciarse en esta figura, el modelo desarrollado encierra una primera aproximación al concepto de *estructura* o *arquitectura de software,* al separar y desacoplar componentes que representan aspectos de visualización/interacción (interfaces gráficas de usuario) de componentes que representan aspectos de la lógica de la aplicación y de componentes que corresponden a la gestión y exportación de los datos generados por la simulación.

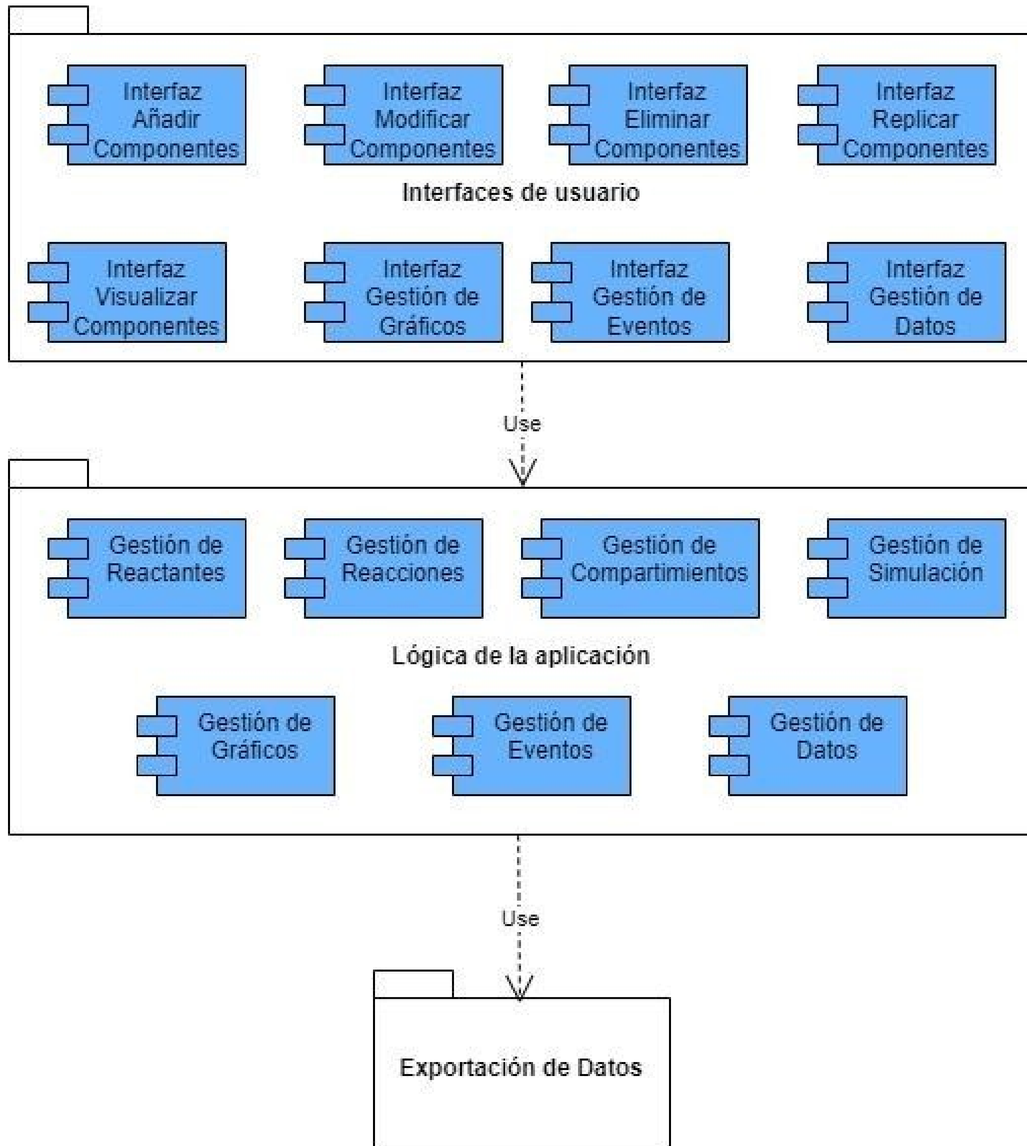


Figura 2.25. Diagrama de componentes correspondiente a la herramienta de simulación bioinformática Big Data-Cellulat

III. EL PAPEL DE LA ARQUITECTURA EN EL DESARROLLO DE SOFTWARE

La arquitectura de un sistema de software (Jacobson, I., Booch, G., Rumbaugh, J. 2000; Cervantes, H., Kazman, R., 2016; Bass, L., Clements, P. y Kazman, R.,2021; González- Pérez, P.P., Gómez Fuentes, M.C., Cervantes Ojeda, J., 2023) es el conjunto de decisiones de diseño significativas sobre la organización y estructuración del software para promover los atributos de calidad deseados y otras propiedades relevantes. Por *decisiones de diseño significativas* hay que entender aquellas determinaciones que pueden representar un punto de no retorno en el desarrollo o influir de forma directa en atributos de calidad, alcance y costos. Las decisiones de diseño significativas suelen ser costosas de revertir o cambiarse en la medida en que el proyecto avanza, por lo que es muy importante tener esto en cuenta al momento de diseñar arquitectónicamente. Por otra parte, con *promover los atributos de calidad* nos referimos a procurar que mediante nuestras resoluciones los atributos de calidad esperados por el cliente (volveremos a este punto más adelante) emerjan como consecuencia en el producto software resultante, y lo soporten. Las correctas decisiones arquitectónicas y su implementación tienen un papel fundamental para mantener el proyecto dentro del presupuesto y liberar los resultados en el tiempo planeado.

La arquitectura del software (Jacobson, I., Booch, G., Rumbaugh, J. 2000; Bass, L., Clements, P. y Kazman, R.,2021; González-Pérez, P.P., Gómez Fuentes, M.C., Cervantes Ojeda, J., 2023) especifica la organización del software, habiendo considerado los componentes que integrarán el sistema, las interfaces y los comportamientos que caracterizan a estos componentes, así como la forma como se establece la comunicación, interacción y colaboración entre dichos componentes. Esta integra todos los aspectos del software: lógicos, de proceso, de componentes, físicos… La arquitectura del software está afectada no solo por la estructura y el comportamiento sino también por el uso, la funcionalidad, el rendimiento, la flexibilidad, la reutilización, la facilidad de comprensión, las restricciones y compromisos económicos y tecnológicos, y la estética (Jacobson, I., Booch, G., Rumbaugh, J. 2000).

Hasta aquí hemos enfatizado la importancia de la arquitectura y por qué debemos tener especial cuidado a la hora de definirla; sin embargo, y por extraño que parezca, todo sistema de software posee una arquitectura, incluso si no se determinó a propósito. El punto es que el no haber participado de manera activa en su definición incrementará de modo significativo las posibilidades de que el proyecto de software no cumpla las expectativas del cliente o lo haga comprometiendo atributos de calidad, como la seguridad, el desempeño, la modularización y la escalabilidad, por mencionar algunas.

Un factor relevante cuando se toman decisiones acerca de la arquitectura del software es el hecho de saber que el problema que intenta resolverse, si no es igual, al menos podría ser muy similar a problemas que muchos otros arquitectos e ingenieros de software (e, incluso, nosotros mismos como desarrolladores de software) han enfrentado en algún momento, y que la experiencia acumulada, así como las decisiones tomadas han ido formando parte de lo que conocemos como *patrones arquitectónicos*.

Los patrones arquitectónicos son referencias que responden a los atributos de calidad que hemos identificado para nuestro sistema; son una guía de la cual podemos partir y, en caso necesario, extender, lo que acelerará nuestro proceso de desarrollo y lo robustecerá. Dado que el catálogo de patrones arquitectónicos es muy vasto, ha sido necesario crear categorías que ayuden a organizarlo. Dentro de estas categorías encontramos arquitecturas tanto físicas como lógicas entre las que destacan las que se relacionan a continuación.

3.1. Arquitectura lógica

La arquitectura lógica del sistema de software (conocida también como *arquitectura del software*) especifica la organización del software, considerando los componentes lógicos que la integrarán, tales como capas, paquetes, módulos, relaciones de dependencia entre módulos, clases, relaciones entre clases, etcétera. (González-Pérez, P.P., Gómez Fuentes, M.C., Cervantes Ojeda, J., 2023). La arquitectura lógica del software integra y estructura las diferentes fases de modelado desarrolladas que, de forma incremental, han capturado diferentes vistas físicas, de interacción, de flujo de ejecución…, a través del uso de diagramas de clases, de secuencia, de actividades, de paquetes y de componentes, entre otros.

Dentro de los principales enfoques sobre la arquitectura lógica del software, los más utilizados, generalizados y extendidos en las últimas décadas han sido los siguientes:

- Modelo arquitectónico orientado al flujo de datos
- Modelo arquitectónico orientado a objetos
- Modelo arquitectónico de sistemas de software interactivos
- Modelo arquitectónico de capas
- Modelo arquitectónico de pizarra o repositorio (del inglés, blackboard systems)

De los anteriores enfoques arquitectónicos, en este material nos centraremos en dos de ellos: 1) el modelo arquitectónico de sistemas interactivos —de forma particular, en el patrón arquitectónico Modelo-Vista-Controlador— y 2) el modelo arquitectónico de capas.

3.1.1. El modelo arquitectónico de sistemas interactivos: el patrón arquitectónico Modelo-Vista- Controlador

El modelo de organización del software interactivo —en general, referido como *arquitectura* o *patrón arquitectónico* Modelo-Vista-Controlador (MVC)— (Russell, J. (Editor), Cohn, R. (Editor), 2012; Pitt, C.; 2012; Yah, A., 2017; González-Pérez, P.P., Gómez Fuentes, M.C., Cervantes Ojeda, J., 2023) nace en el contexto del lenguaje orientado a objetos Smalltalk (Goldberg, A., Robson, D., Harrison, M.A. (Editor), 1983.), con el objetivo de proporcionar una real separación entre aspectos de presentación y visualización de aspectos ligados a la lógica de la aplicación. Según este

modelo arquitectónico, una aplicación interactiva se estructura separando la representación de los datos (modelo), la presentación de los datos (vista) y el comportamiento/lógica de la aplicación (controlador).

El patrón de arquitectura lógica MVC (ver figura 3.1) procura la separación entre la lógica de negocios y la vista, con la idea de separar responsabilidades y proporcionar una mejor distribución del trabajo y facilitar su mantenimiento. El patrón MVC ha gozado de gran popularidad durante las últimas décadas, y hoy en día ha sido extendido a otros patrones arquitectónicos, como Presentación-Abstracción-Control (PAC) (Qian, K. 2009; González-Pérez, P.P., Gómez Fuentes, M.C., Cervantes Ojeda, J., 2023), Modelo-Vista-Presentador (MVP) (Potel, M., 1996) y Modelo-Vista-Modelo de Vista (MVVM) (Microsoft, 2022).

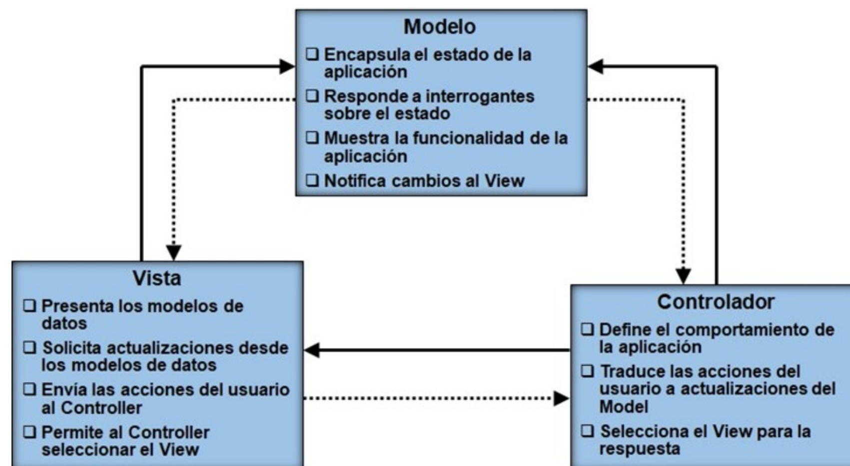


Figura 3.1. Patrón de arquitectura lógica Modelo-Vista-Controlador

- **Modelo.** Representa la estructura de los datos en la aplicación y las relativas operaciones (dependientes de la aplicación).
- **Vista.** Es responsable de la interacción con el usuario. Presenta los datos en diferentes formas; de aquí, que pueda haber más vistas en la medida en que sea necesario presentar los datos en formas diferentes. Recibe las entradas del usuario.
- **Controlador.** Atiende y recibe las acciones del usuario ejecutadas sobre la Vista, por lo que puede recuperar los datos proporcionados por este, trasladar los mismos en invocaciones de los métodos adecuados del modelo y seleccionar la vista apropiada (con base en el

estado y la preferencia del usuario). En general, el controlador escucha los eventos de entrada que le llegan de la vista, y reacciona como consecuencia sobre el modelo. Funge como intermediario-coordinador entre la vista y el modelo.

La figura 3.2 ilustra otra forma de ver los componentes del patrón MVC y la interacción que toma lugar entre los mismos. Las figuras 3.3, 3.4 y 3.5 muestran la representación estructural de los patrones Presentación-Abstracción-Control (PAC), Modelo-Vista-Presentador (MVP) y Modelo-Vista-Modelo de Vista (MVVM), respectivamente. Nótese que estos tres patrones arquitectónicos son herencia del patrón MVC.

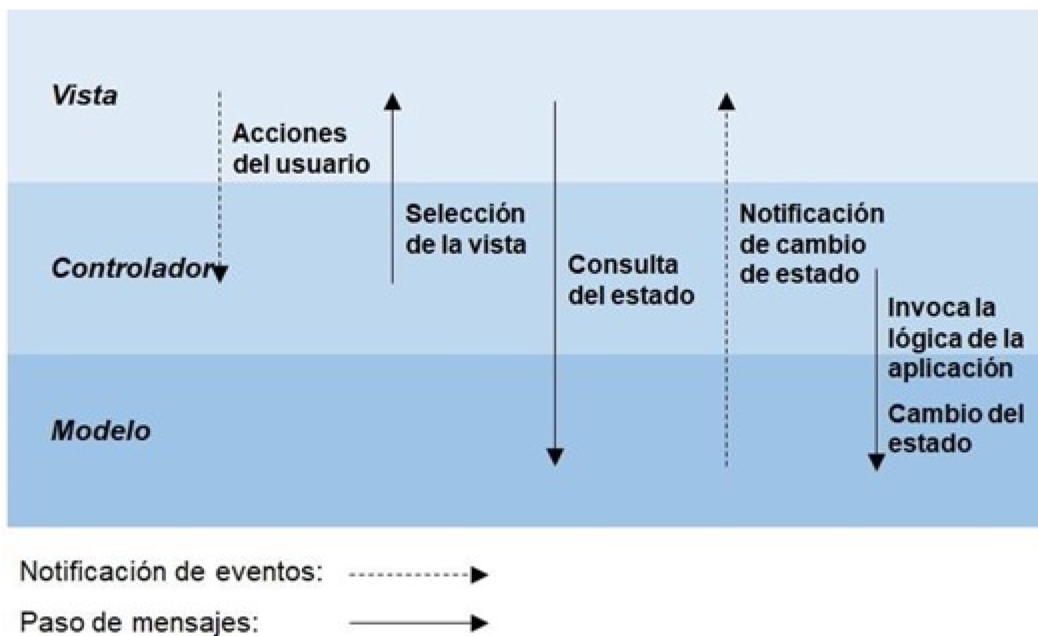


Figura 3.2. Capas del patrón MVC y la interacción que toma lugar entre las mismas

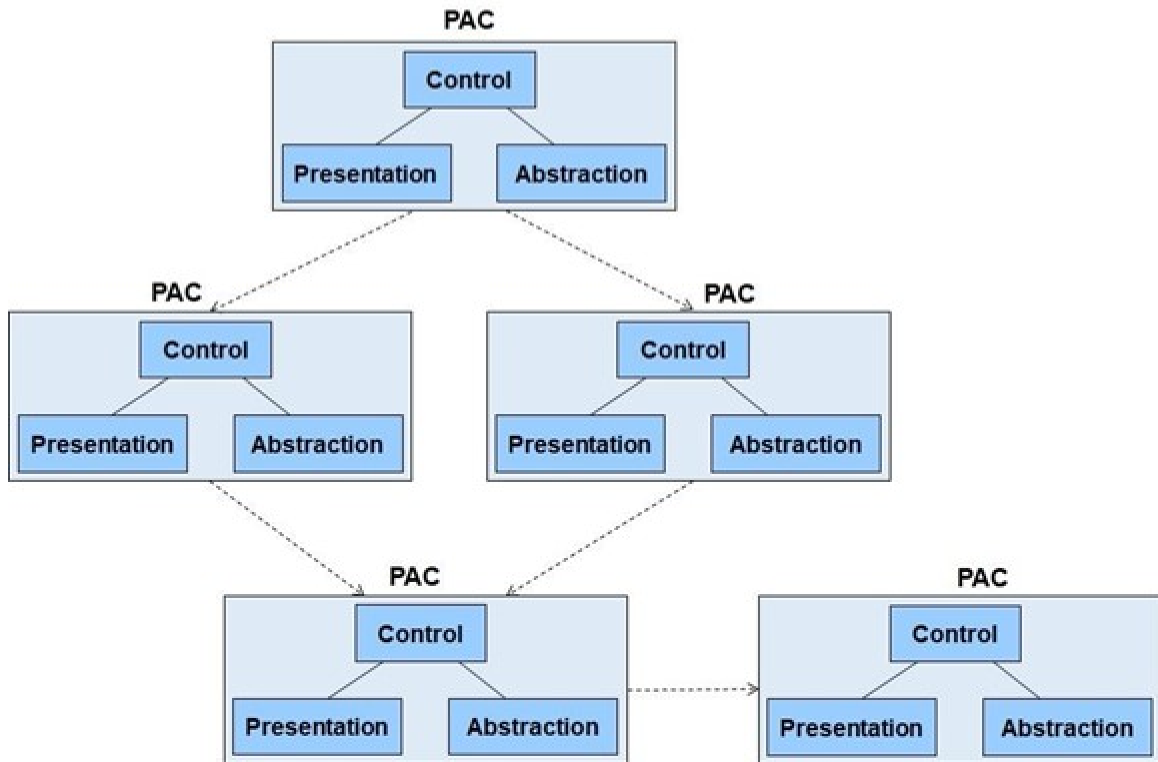


Figura 3.3. Patrón arquitectónico Presentación-Abstracción-Control (PAC)

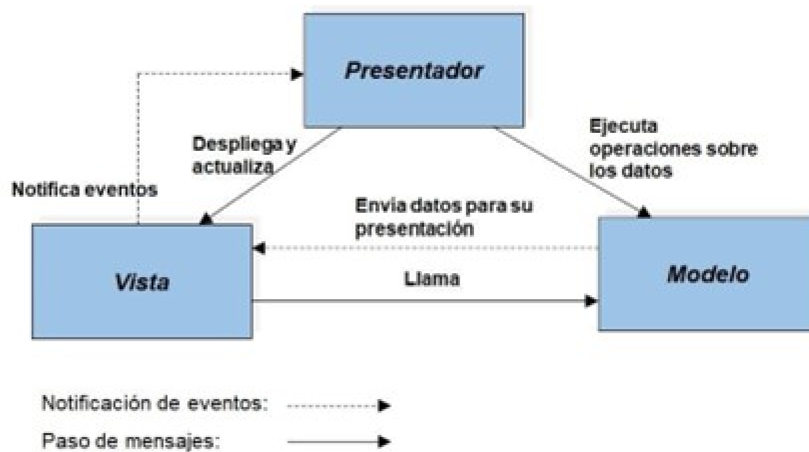


Figura 3.4. Patrón arquitectónico Modelo-Vista-Presentador (MVP)

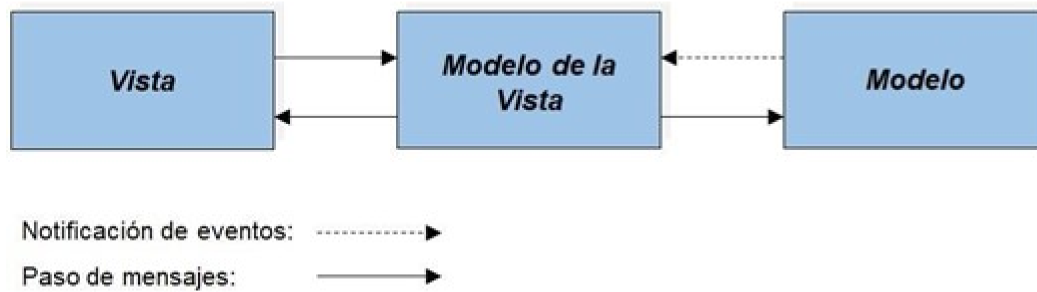


Figura 3.5. Patrón arquitectónico Modelo-Vista-Modelo de la Vista (MVVM)

Aquí es importante mencionar que mucho del desarrollo web actual implementa el patrón arquitectónico MVC o alguno de sus refinamientos o extensiones. Incluso frameworks de desarrollo de software, como los de Java (*Spring MVC, JSF, Struts e Hibernate*), los de PHP (Laravel y Symfony), React y Angular JS, implementan dicho patrón.

Para ilustrar el uso de algunas variantes de la arquitectura lógica MVC, retomaremos los dos casos de estudio que hemos tratado en las secciones previas: el sistema de reservaciones y ventas de una aerolínea y la herramienta de simulación bioinformática Big Data-Cellulat.

Con relación a nuestro primer caso de estudio, la figura 3.6 ilustra el uso de la arquitectura lógica MVC extendida a cuatro capas, al incorporar una capa dedicada a los servicios, mientras que la figura 3.7 propone un enfoque arquitectónico basado en el patrón PAC. Por otra parte, la figura 3.8 ilustra el diseño arquitectónico basado en el patrón MVC extendido a cuatro capas para la herramienta de simulación bioinformática Big Data-Cellulat.

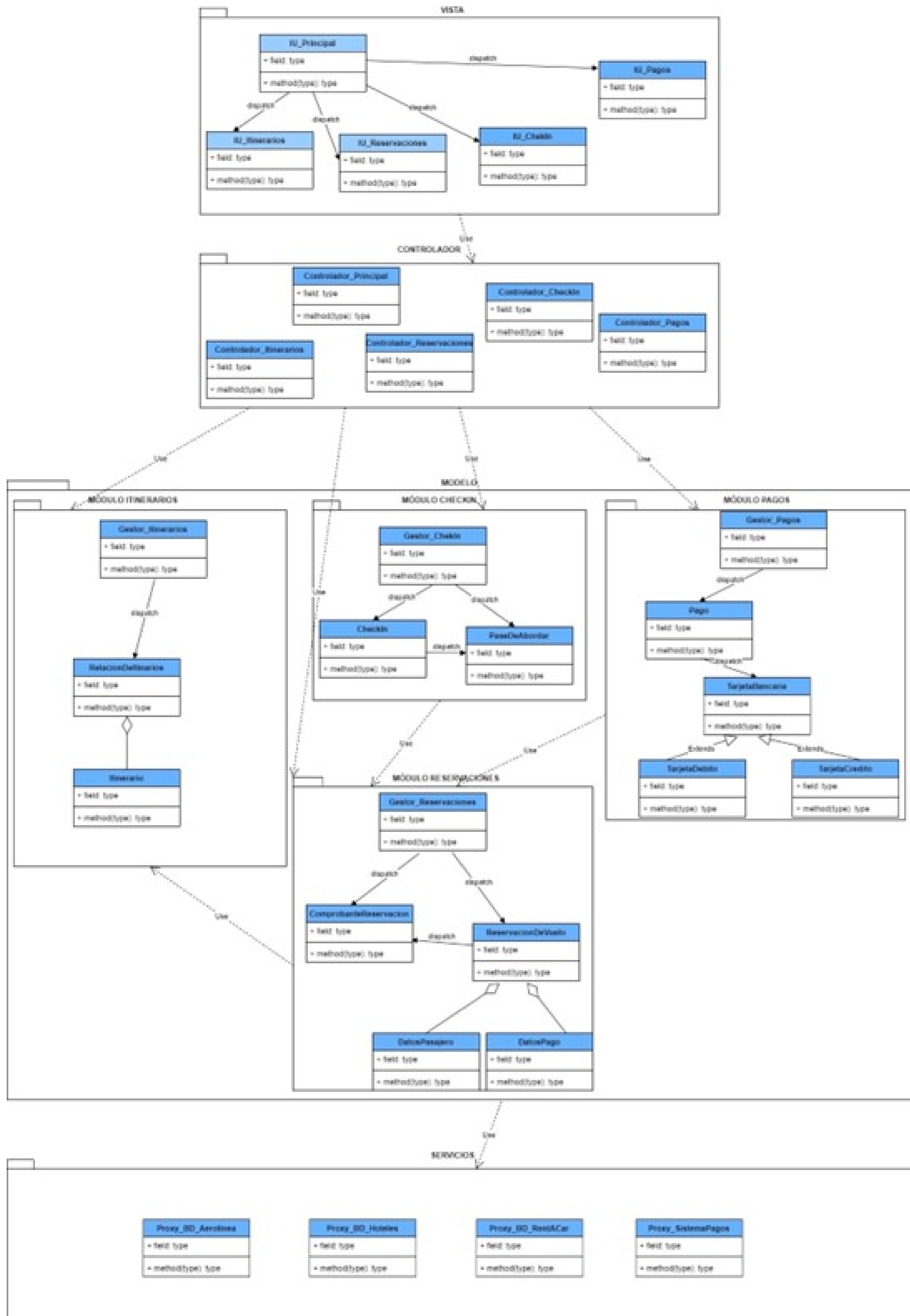


Figura 3.6. Patrón arquitectónico MVC extendido a cuatro capas, ensayado durante el diseño arquitectónico del sistema de reservaciones y ventas de

una aerolínea

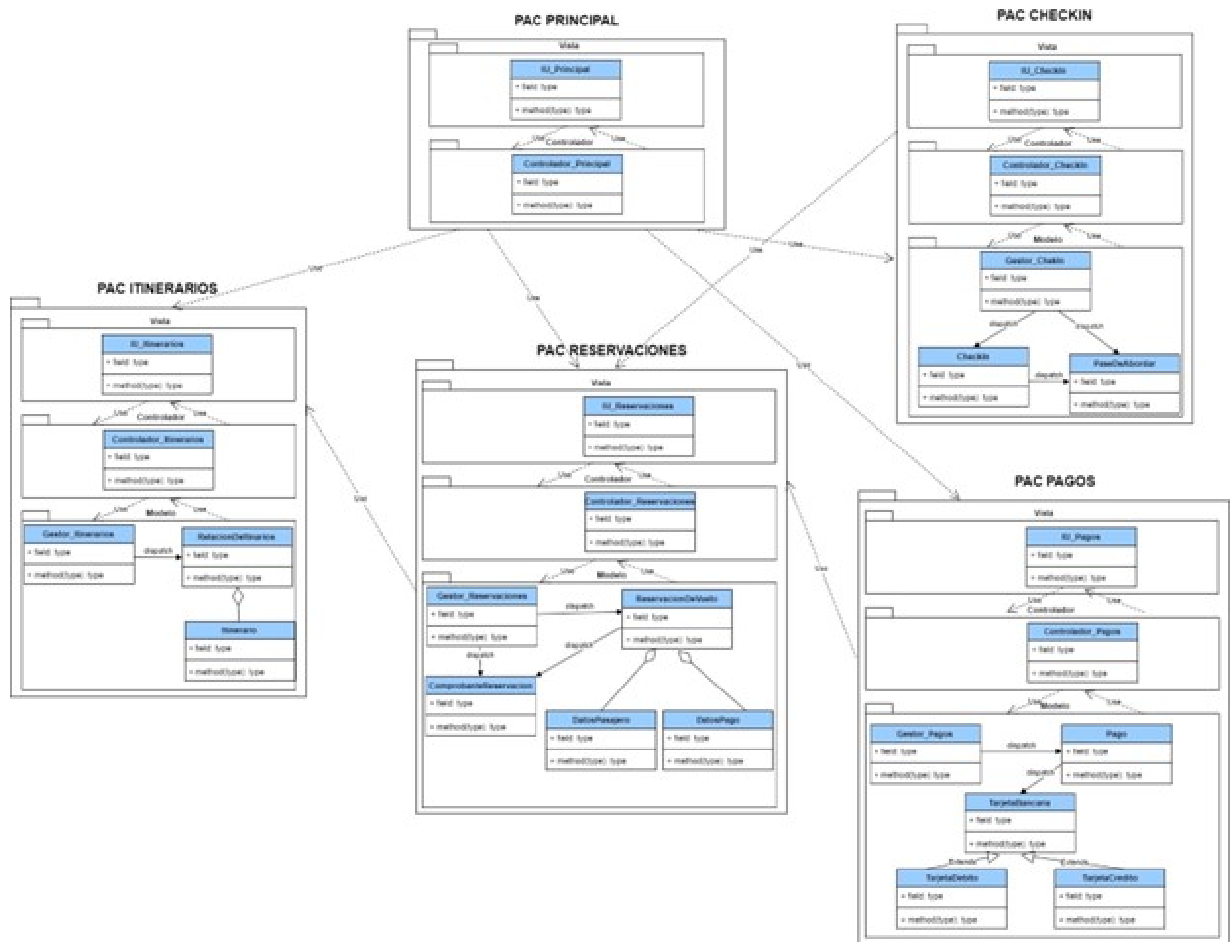


Figura 3.7. Patrón arquitectónico PAC, ensayado durante el diseño arquitectónico del sistema de reservaciones y ventas de una aerolínea

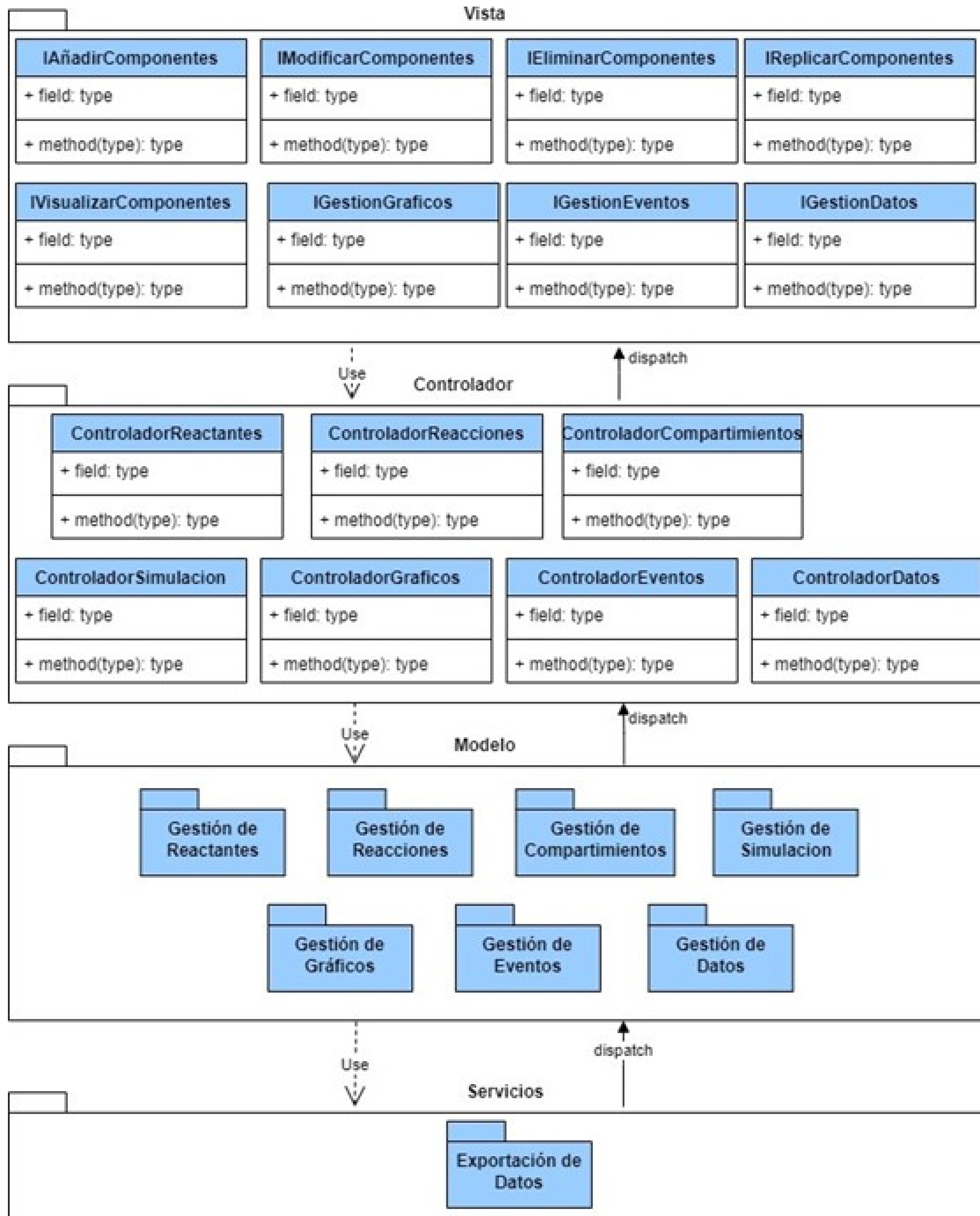


Figura 3.8. Patrón arquitectónico MVC extendido a cuatro capas, ensayado durante el diseño arquitectónico de la herramienta de simulación bioinformática Big Data-Cellulat

3.1.2. El patrón arquitectónico basado en capas

Este patrón ha sido durante muchos años el estándar de facto en equipos de desarrollo tradicionales debido a que su estructura de capas (presentación, negocio, persistencia, datos, etcétera) permite mapear fácilmente responsabilidades al interior del equipo, por lo que se provecha su organización (*frontend, backend,* bases de datos…). Se conforma por una pila de capas o tercios (también es conocida como *arquitectura n-tercios*) que aísla responsabilidad-funcionalidad, y en la que en una arquitectura cerrada por completo, cada capa solo podría comunicarse con las capas inmediatas superior e inferior; es decir, capas con las que mantiene contacto directo. En una arquitectura de capas abierta, el diseño permite que exista una o más capas a las que puede accederse de forma directa, aunque no sean adyacentes; por ejemplo, una capa de servicios. El patrón arquitectónico basado en capas suele encontrarse en aplicaciones de escritorio.

Como puede apreciarse en la figura 3.9, las capas más utilizadas en este tipo de arquitectura son las siguientes:

- Presentación
- Negocio o dominio
- Persistencia
- Datos

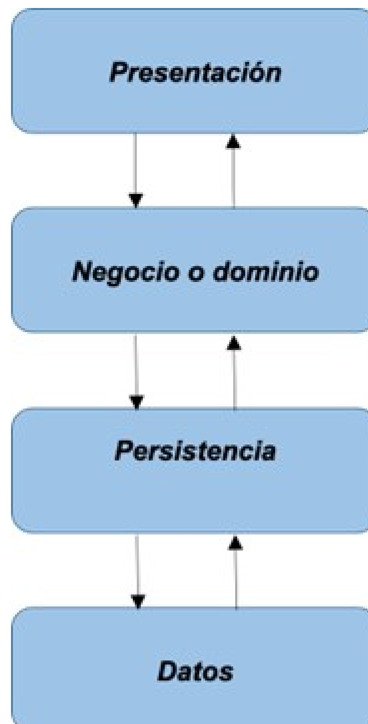


Figura 3.9. Capas más utilizadas en el patrón arquitectónico basado en capas

La capa de presentación es la encargada de gestionar la interacción entre el sistema y el usuario; es decir, cualquier código que permita al usuario obtener información y, por supuesto, proveerla. Es importante dejar en claro que el código en esta capa no tiene idea del negocio de la aplicación, sino que él nada más se encarga de responder a las interacciones del usuario y de delegar a la capa que le sigue (casi siempre la de negocio) el resolver qué hacer con la información/petición que le ha sido pasada. Por mencionar un ejemplo, en una aplicación web, la capa de presentación se encarga de cómo el HTML/JavaScript se presenta al usuario; lo mismo, para las interacciones que —mediante botones, combos, cajas de texto, etcétera— el usuario pudiera llevar a cabo. Sistemas de consola, páginas web, aplicaciones móviles y hasta Interfaces de Programación de Aplicaciones (APIs) por mencionar algunas, podrían sacar provecho de esta separación de responsabilidades.

En la capa de negocio residen los modelos que soportan las abstracciones del problema que intenta resolverse. Esto, en forma de software con la lógica necesaria que responde a los requerimientos de funcionalidad definidos con anterioridad. Una vez que la capa de presentación ha informado que existe una solicitud del usuario, la capa de negocio se encargará de procesar (en dependencia del usuario/solicitud) la información que le será devuelta al mismo, ya sea realizando operaciones o alguna validación o, por ejemplo, delegando a la capa de *siguiente*, la de datos, la gestión de información.

La capa de persistencia contiene el código que servirá para almacenar y actualizar los datos de la aplicación. Esto casi siempre se hace mapeando el modelo de clases y construyendo estructuras como DAOs (Objeto de Acceso a Datos). En algunas implementaciones la capa de persistencia y la capa de datos es la misma, por lo que podríamos encontrar aquí la base de datos y las consultas a ella.

Finalmente, la capa de datos se encarga de administrar el modo como la información es almacenada; por ejemplo, de forma relacional o no relacional. En esta capa encontramos bases de datos, sistema de archivos, entre otros.

Como ya hemos mencionado, el patrón MVC podría ser extendido agregando una capa de servicios. Esto es, en esencia, la combinación de un patrón n-capas (en este caso, dos capas) y el patrón MVC. Ambos patrones trabajan en conjunto con la idea de proveer separación de responsabilidades (los desarrolladores podrían trabajar de forma aislada en los diferentes componentes del modelo, la vista y el controlador, y consumir los servicios que otro equipo desarrolle) y resiliencia al permitir que la capa de servicios y la de presentación (donde residirán los componentes del MVC) sean desplegadas en diferentes nodos. La figura 3.10 ilustra el escenario recién discutido. Nótese la simpleza de modelo y el poder que la combinación de distintos patrones provee.

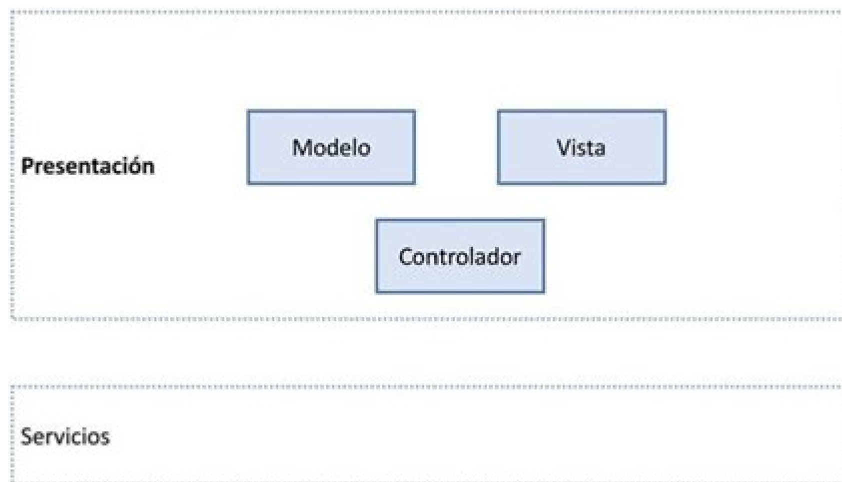


Figura 3.10. Combinación del patrón basado en capas y el patrón MVC

3.2. Arquitectura física

De forma complementaria a la arquitectura lógica, existen patrones arquitectónicos que atienden características propias de la implementación y del hardware sobre el cual se ejecutará el software, como las arquitecturas físicas; esto es, dónde y cómo será desplegado nuestro software. Es decir, la arquitectura física del software se refiere a la implementación de la arquitectura lógica en los componentes de hardware requeridos; es decir, procesadores, nodos, computadoras o cualquier otro tipo de hardware al que sea asignada alguna de las tareas o funcionalidades que deberá llevar a cabo el sistema software. Algunas de las arquitecturas físicas más relevantes son las siguientes:

- Arquitectura standalone
- Arquitectura cliente-servidor
- Arquitectura de sistemas distribuidos

3.2.1. Arquitectura física Standalone

Como se ilustra en la figura 3.11, una arquitectura monolítica o standalone soporta aplicaciones que, por razones de negocio, suelen correr localmente en un único dispositivo (hardware) y son funcionales por cuenta propia; es decir, no tienen dependencias externas para llevar a cabo su ejecución. Aunque muchas arquitecturas standalone se basan en el manejo de archivos para preservar su estado, también existen aquellas que hacen uso de bases de datos embebidas, como —por ejemplo— algunas de las aplicaciones que encontramos en nuestros teléfonos móviles.

El mantenimiento de una aplicación standalone se da al reemplazar la versión actual por una nueva y, aunque esto puede ocurrir valiéndose de un ejecutable, como en los sistemas operativos, también hay aplicaciones que con solo sustituir el archivo que las contiene es suficiente. Pensemos, por ejemplo, en una utilería de *shell*. Uno de los frameworks más usados en la industria del software para el desarrollo de aplicaciones standalone es Electron (Electron, https://www.electronjs.org/es/), el cual es usado en aplicaciones tan populares como Slack (Slack Technologies, https://slack.com/intl/es-la), VsCode (VsCode, https://code.visualstudio.com/), entre otras.

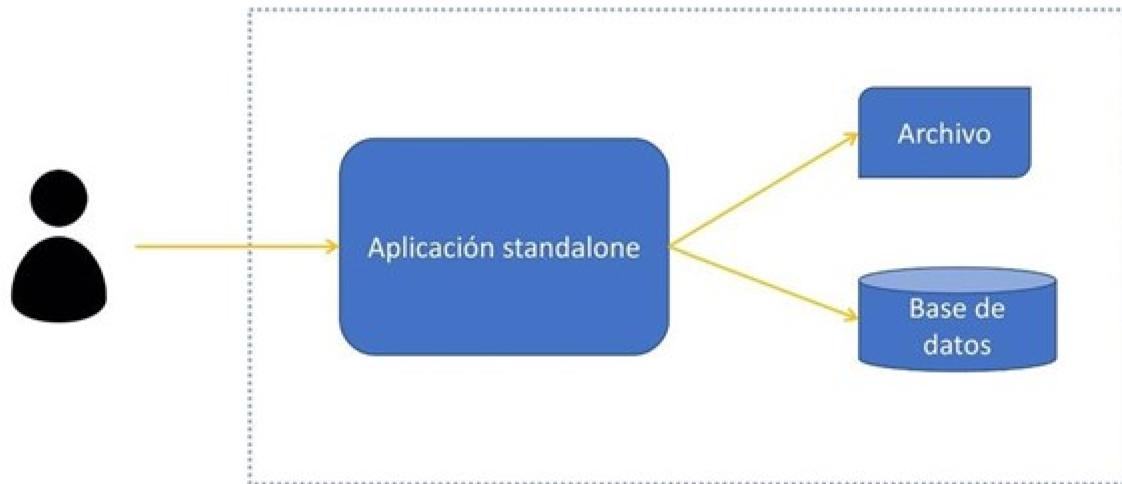


Figura 3.11. Arquitectura física monolítica o standalone. La línea discontinua indica un único procesador o dispositivo hardware

En la figura 3.12 se ilustra la arquitectura física monolítica de la herramienta de simulación bioinformática Big Data-Cellulat. Como puede apreciarse en esta figura, la arquitectura física propuesta nos indica que Big Data-Cellulat se ejecuta sobre un único procesador. Para una mejor comprensión de lo ilustrado en la figura 3.12, consúltese el correspondiente diagrama de componentes ilustrado en la figura 2.25 y el correspondiente patrón arquitectónico MVC extendido a cuatro capas, ilustrado en la figura 3.8.

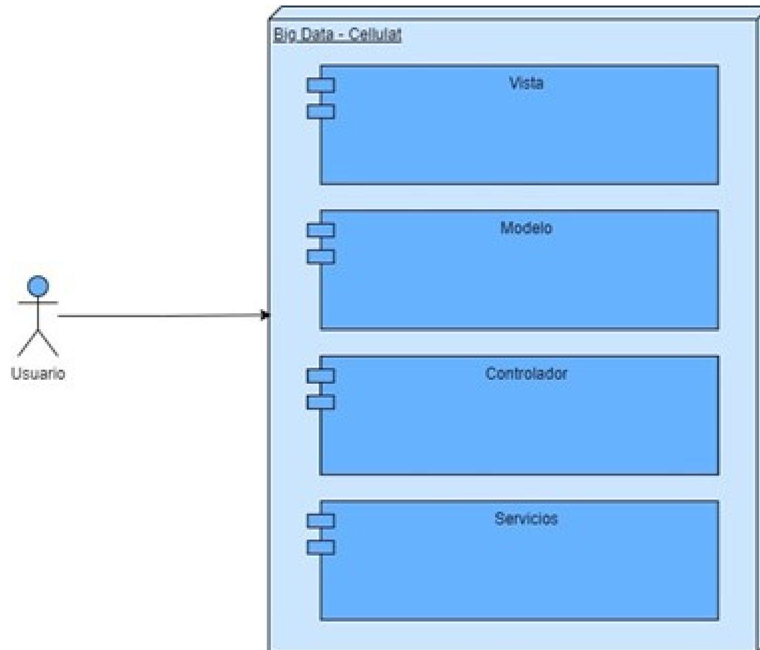


Figura 3.12. Arquitectura física monolítica (standalone) adoptada para la herramienta de simulación bioinformática Big Data-Cellulat

3.2.2. Arquitectura física cliente-servidor

Otro patrón de implementación de arquitectura física muy difundido y utilizado desde el surgimiento de Internet y hasta nuestros días es el cliente-servidor. La arquitectura física de un sistema cliente-servidor se conforma por uno o más servidores que proporcionan servicios y un conjunto de clientes que acceden a ellos y los usan. Desde el punto de vista físico, los clientes pueden ser diferentes tipos de hardware, tales como computadoras de escritorio, laptops, dispositivos móviles, etcétera, mientras que los servidores son comúnmente potentes equipos de cómputo, caracterizados por grandes recursos de memoria y velocidad de procesamiento. En este tipo de arquitectura, casi siempre la comunicación comienza cuando un cliente solicita un servicio, por lo que el servidor se mantendrá a la escucha de futuras solicitudes provenientes de los clientes. La mayoría de las aplicaciones web utilizan este patrón arquitectónico.

La arquitectura cliente-servidor es un tipo de arquitectura distribuida, mediante la cual los clientes pueden acceder a los recursos que proporcionan los servidores. En general, la interacción entre el cliente y el servidor se da por medio de dos tipos de mensaje: petición y respuesta. La

petición es un mensaje enviado por el cliente al servidor para solicitar un servicio particular. De forma complementaria, la respuesta es un mensaje enviado por el servidor al cliente para proporcionar el servicio solicitado. La interacción entre clientes y servidores ocurre a través de una estructura de comunicación conocida como *red*. En aplicaciones cliente-servidor web, esta red se refiere a Internet.

En la figura 3.13 puede apreciarse una representación esquemática de la arquitectura de sistema cliente-servidor web.

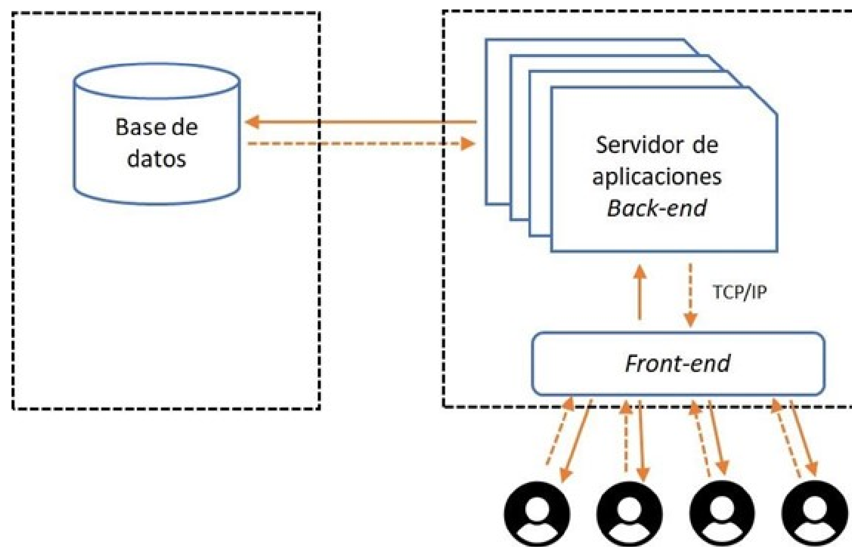


Figura 3.13. Arquitectura física cliente-servidor web

En la figura 3.14 puede apreciarse la arquitectura física cliente-servidor web propuesta para el sistema de reservaciones y ventas de una aerolínea. Como puede verse en esta figura, la arquitectura física propuesta nos indica que el sistema requiere varios procesadores (servidores) para su ejecución; destaca entre estos el servidor de aplicaciones y varios servidores de bases de datos. Para una mejor comprensión de lo ilustrado en la figura 3.14, consúltese el correspondiente diagrama de componentes ilustrado en la figura 2.24 y el correspondiente patrón arquitectónico MVC extendido a cuatro capas, ilustrado en la figura 3.6.

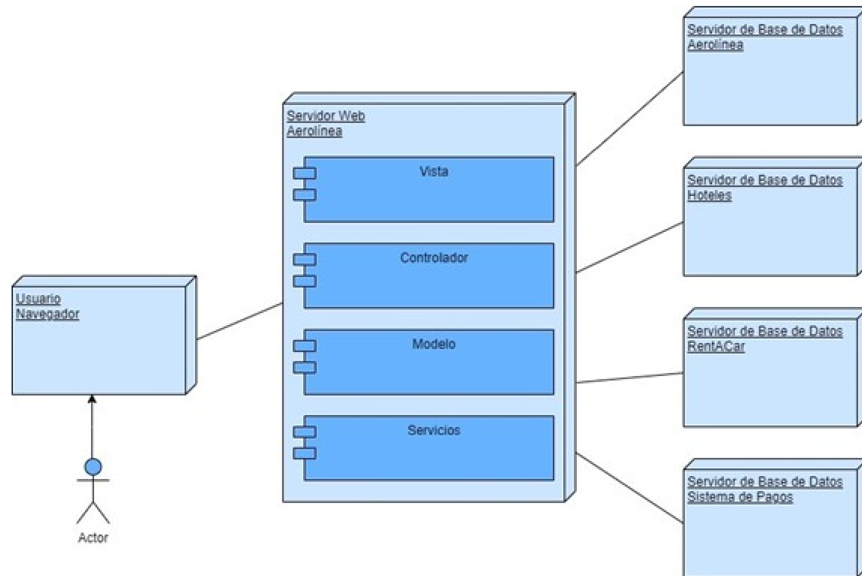


Figura 3.14. Arquitectura física cliente-servidor web adoptada para el sistema de reservaciones y ventas de una aerolínea

3.2.3. Arquitectura física sistemas distribuidos

Cuando nos referimos a *arquitectura física del tipo sistemas distribuidos*, solemos estar pensando en una arquitectura física que involucra varios nodos hardware (principalmente procesadores), donde los componentes lógicos (funciones, clases, módulos, subsistemas, etcétera) se encuentran asignados a dichos nodos. Los nodos y los componentes lógicos se comunican y coordinan sus actividades mediante el paso de mensajes. Una característica principal de este tipo de sistemas es la concurrencia, mientras que el control y el procesamiento se encuentra distribuido entre los nodos. En la figura 3.15 se ofrece una representación esquemática de la arquitectura física sistema distribuido.

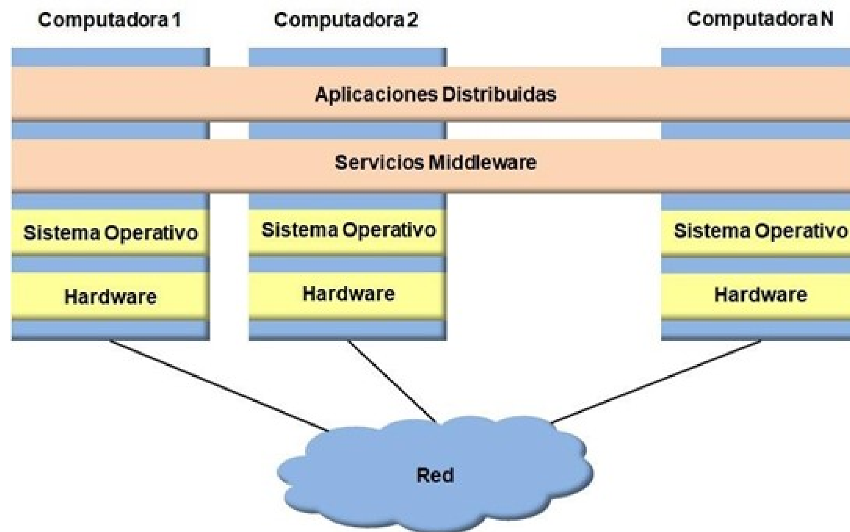


Figura 3.15. Arquitectura física sistema distribuido. Figura tomada de González-Pérez, P.P., Gómez-Fuentes, M.C., Cervantes-Ojeda, J., 2023

IV. LA IMPORTANCIA DE LA COMUNICACIÓN EN EL DESARROLLO DE SOFTWARE

En el desarrollo de sistemas de software, la comunicación es crucial para el éxito de cualquier proyecto que pretenda llevarse a cabo. El desarrollo de software es un proceso en el que varios actores trabajan de forma conjunta y colaborativa, por lo que la comunicación eficaz entre ellos es fundamental para garantizar la comprensión, coordinación y toma de decisiones efectiva durante las diferentes etapas que involucra el desarrollo de software.

La comunicación debe considerarse uno de los ejes principales de cualquier proyecto de desarrollo de software. A partir de una buena comunicación los requerimientos pueden ser recolectados, analizados, especificados y verificados/validados de forma adecuada. Aunado a lo anterior, una vez que entendemos qué busca solucionar el cliente, tenemos la tarea de comunicar los requerimientos a nuestro equipo de desarrollo de software, el cual, en conjunto con el arquitecto, trabajará en una solución que después presentarán al cliente; es decir, la respuesta al “cómo”.

La comunicación en un proyecto de desarrollo de software debe tratarse de forma multidimensional; es decir: deben establecerse mecanismos que permitan transmitir ideas de forma clara con la información suficiente, en dependencia del ámbito en el cual estemos desenvolviéndonos. No es lo mismo comunicarse con un equipo de desarrollo de software usando diagramas de secuencia (UML) para describir interacciones entre objetos, que usar un diagrama de flujo para comunicar este mismo proceso a un usuario del sistema. Estrategias y herramientas para conseguir este tipo de comunicación multidimensional o multivista, como modelos, diagramas y artefactos, ya se abordaron, en parte, en este material.

Aquí es importante hacer notar que la comunicación que se establece durante el desarrollo de sistemas de software no se restringe a la comunicación verbal entre los miembros del equipo de desarrollo. Más allá de esto, la comunicación incluye toda la documentación textual, gráfica y basada en modelos que va liberándose durante las diferentes etapas del proceso de desarrollo de software, desde la fase de gestión de los requerimientos hasta

la fase de calidad y pruebas; lo anterior facilita enormemente la transmisión de información entre los diferentes roles dentro del equipo de desarrollo.

Entre los principales actores encargados de liderar los diferentes tipos de comunicación que se establecen entre los involucrados en el desarrollo de un proyecto de software —tanto por parte del cliente como por parte del equipo de desarrollo— podemos identificar los siguientes:

- Usuario/cliente final
- Directivos/líderes de proyecto
- Desarrolladores de software
    - Analistas de sistema
    - Arquitectos del software
    - Encargados del diseño detallado
    - Programadores
    - Responsables de las bases de datos
    - Responsables de las pruebas
    - Responsables del soporte técnico

A continuación, analizaremos de forma más detallada el papel de algunos de estos actores en la comunicación durante el proceso de desarrollo de software (González- Pérez, P.P., Gómez Fuentes, M.C., Cervantes Ojeda, J., 2023) e ilustraremos el papel de los mismos utilizando los dos casos de estudio ya abordados, mediante aspectos del modelado y el diseño arquitectónico, en los capítulos precedentes.

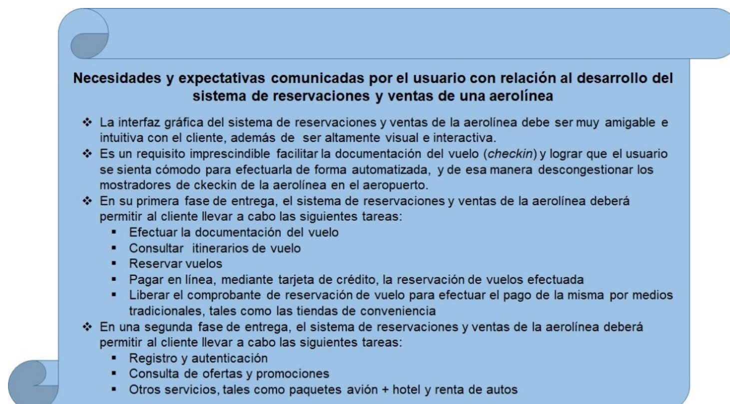
4.1. El rol del usuario/cliente final en la comunicación en el desarrollo de software

Los clientes o usuarios finales son uno de los principales actores de la comunicación en el desarrollo de sistemas de software. Son ellos quienes establecen las necesidades y expectativas del sistema que requiere desarrollarse, y son los beneficiarios directos del producto software. Para comprender las necesidades y objetivos de los clientes se necesita una comunicación clara y continua. Es mediante la comunicación efectiva entre el usuario y el equipo de desarrollo que es posible identificar errores en la comprensión de las necesidades del cliente y problemas potenciales, lo que da como resultado que se hagan ajustes en función de sus comentarios. La falta de comunicación con los clientes puede devenir en malentendidos, requisitos poco claros y soluciones insatisfactorias.

El usuario/cliente final comunica al equipo de desarrollo (sobre todo, al líder de proyecto o analistas) requerimientos funcionales, restricciones, así como otras características deseables, y ayuda al equipo de desarrollo a validar el diseño. Durante esta comunicación, comúnmente se usan diferentes técnicas, tales como el lenguaje verbal registrado, la comunicación textual (en general, por medio de cuestionarios, formularios, etcétera), la observación directa, los diagramas de flujo y otros artefactos, como casos de uso e historias de usuario.

En las figuras 4.1 y 4.2 se ilustran artefactos/modelos utilizados por el cliente para establecer comunicación con el equipo de desarrollo (de forma particular, con el líder de proyecto) del sistema de reservaciones y ventas de una aerolínea. Como puede apreciarse en estas dos figuras, entre los artefactos utilizados por el cliente para expresar sus necesidades y requisitos destacan la información textual (figura 4.1) y los diagramas de flujo (figura 4.2). Por otra parte, las figuras 4.3 y 4.4 muestran algunos de los artefactos/modelos utilizados por el cliente para establecer la comunicación con el líder de proyecto durante la recolección de los requerimientos para desarrollar la herramienta de simulación bioinformática Big Data-Cellulat. Como puede apreciarse en estas dos figuras, entre los recursos usados por el cliente para expresar sus necesidades y requisitos destacan la información textual (figura 4.3) y la información gráfica (figura 4.4). Esta última indica

los resultados visuales que debe producir la herramienta de simulación bioinformática.



Necesidades y expectativas comunicadas por el usuario con relación al desarrollo del sistema de reservaciones y ventas de una aerolínea

- La interfaz gráfica del sistema de reservaciones y ventas de la aerolínea debe ser muy amigable e intuitiva con el cliente, además de ser altamente visual e interactiva.
- Es un requisito imprescindible facilitar la documentación del vuelo (*checkin*) y lograr que el usuario se sienta cómodo para efectuarla de forma automatizada, y de esa manera descongestionar los mostradores de ckeckin de la aerolínea en el aeropuerto.
- En su primera fase de entrega, el sistema de reservaciones y ventas de la aerolínea deberá permitir al cliente llevar a cabo las siguientes tareas:
  - Efectuar la documentación del vuelo
  - Consultar itinerarios de vuelo
  - Reservar vuelos
  - Pagar en línea, mediante tarjeta de crédito, la reservación de vuelos efectuada
  - Liberar el comprobante de reservación de vuelo para efectuar el pago de la misma por medios tradicionales, tales como las tiendas de conveniencia
- En una segunda fase de entrega, el sistema de reservaciones y ventas de la aerolínea deberá permitir al cliente llevar a cabo las siguientes tareas:
  - Registro y autenticación
  - Consulta de ofertas y promociones
  - Otros servicios, tales como paquetes avión + hotel y renta de autos

Figura 4.1. Comunicación textual por parte del cliente al líder de proyecto, acerca de las necesidades y expectativas sobre el sistema de reservaciones y ventas de una aerolínea

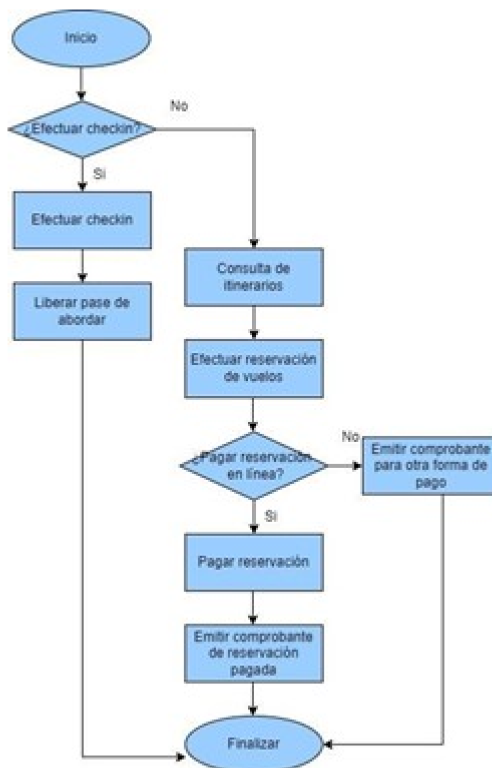


Figura 4.2. Comunicación mediante diagramas de flujo por parte del cliente al líder de proyecto acerca de las necesidades y expectativas sobre el sistema de reservaciones y ventas de una aerolínea

Necesidades y expectativas comunicadas por el usuario con relación al desarrollo de una herramienta de simulación de redes de señalización celular

- La herramienta de simulación bioinformática debe ser altamente visual, interactiva, intuitiva y amigable con el usuario bioinformático.
- La simulación de las vías y redes de señalización debe basarse en el algoritmo de Gillespie, un algoritmo de simulación de reacciones químicas, estocástico y discreto, que ha resultado muy difundido y ampliamente utilizado en las últimas décadas. Se proporciona la documentación requerida de dicho algoritmo.
- La herramienta de simulación bioinformática debe proporcionar al usuario la ejecución de las siguientes tareas:
  - Creación, modificación, visualización y eliminación de estructuras celulares, tales como tejidos, células y compartimientos celulares
  - Creación, modificación, visualización y eliminación de reacciones químicas y sus parámetros cinéticos, en un lenguaje muy cercano al natural
  - Creación, modificación, visualización y eliminación de reactantes y sus concentraciones micromolares en un lenguaje muy cercano al natural
  - Visualización e interacción de una amplia gama de gráficos que muestren el comportamiento de la ejecución de la simulación
  - Cosecha de datos (data farming) y exportación de los mismos en formato .csv.
- ...

Figura 4.3. Comunicación textual por parte del cliente al líder de proyecto, acerca de las necesidades y expectativas sobre la herramienta de simulación bioinformática

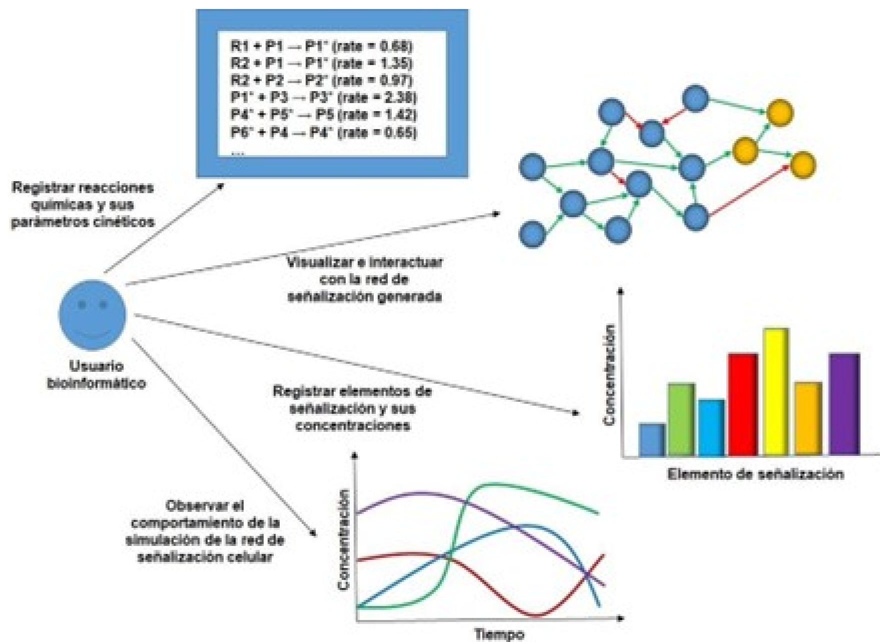


Figura 4.4. Comunicación gráfica por parte del cliente al líder de proyecto acerca de las necesidades y expectativas sobre la herramienta de simulación bioinformática

4.2. El rol de los responsables/líderes de proyecto en la comunicación en el desarrollo de software

El responsable del proyecto es el encargado principal de definir el proyecto de desarrollo de software y de asignar todos los recursos (financieros, humanos, tecnológicos…) para llevarlo a cabo. Además, colabora con el líder del proyecto en las tareas de administración, estimación y planificación del proyecto, así como en la revisión y aprobación de la planificación del proyecto. Por su parte, el líder de proyecto guía al equipo de desarrollo, que incluye arquitectos, analistas, programadores, responsables de diseño detallado, bases de datos, responsables de verificación y de pruebas y soporte técnico; también asigna el trabajo al equipo de desarrollo y responde a sus necesidades brindando soluciones oportunas.

Los responsables/líderes de proyecto suelen comunicar el estado del proyecto y establecer compromisos, basados en planeación y métricas, entre el cliente y el equipo de desarrollo de software. Para lo anterior se apoyan en diagramas de Gantt, cronogramas de trabajo, presentaciones y documentos escritos en lenguaje natural. Entre las principales métricas utilizadas por los responsables/líderes para comunicar el estado del proyecto se encuentran la planeación inicial en meses, quincenas o semanas, el tiempo ya invertido de dicha planeación, la cantidad y prioridad de las funcionalidades concluidas (diseñadas, implementadas y probadas unitaria e incrementalmente), el esfuerzo personas/mes, entre otras.

4.3. El rol de los desarrolladores en la comunicación durante el desarrollo de software

Los desarrolladores de software también tienen un papel importante en la comunicación. Estos profesionales son responsables de convertir las especificaciones y requisitos planteados por el cliente en un producto de software funcional. Para garantizar la coherencia en el diseño, la arquitectura y la implementación del software, los desarrolladores deben comunicarse de forma adecuada entre sí.

Mediante la comunicación eficiente, los desarrolladores garantizan que se alcancen los objetivos del proyecto, se discutan ideas acerca del desarrollo del mismo, se resuelvan problemas técnicos y se compartan conocimientos. Gran parte de esta comunicación se basa en compartir, discutir, refinar y traducir los modelos y artefactos desarrollados a lo largo de las diferentes fases del proceso de desarrollo de software. Para comprender mejor lo antes expuesto, pensemos en que un artefacto como el diccionario de entidades o clases, elaborado durante la fase de análisis, puede ser traducido a una primera vista estática de la solución, utilizando para ello un diagrama de clases.

4.3.1. Analistas, diseñadores y programadores

Una de las habilidades que se espera que tengan los desarrolladores de software —en particular, los analistas, diseñadores y programadores— es la de interpretar diagramas y especificaciones técnicas elaborados en fases previas del proceso de desarrollo para poder traducirlos durante las fases de análisis y diseño, con la finalidad de que exhiba una granularidad mucho más fina que la precedente, así como para trasladar la solución expresada mediante el diseño detallado a una solución plasmada en un lenguaje de programación (codificación). Además de lo anterior, los desarrolladores de software deben ser capaces de comprender y retroalimentar casos de uso e historias de usuario.

Para ejemplificar cómo los analistas deben interpretar las necesidades y expectativas del cliente, regresemos al caso de estudio relacionado con el desarrollo de la herramienta de simulación bioinformática. Como ya vimos en las figuras 4.3 y 4.4, el cliente utilizó la información textual (figura 4.3) y la información gráfica (figura 4.4) para expresar al líder de proyecto o

analista sus requisitos y necesidades acerca del producto software solicitado.

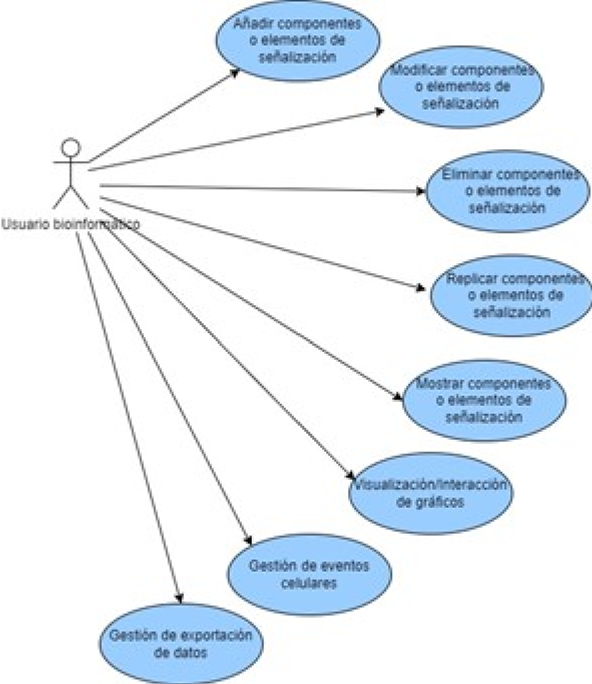


Figura 4.5. Uso de los diagramas de casos de uso como artefacto de comprensión y comunicación de las necesidades del cliente de la herramienta de simulación bioinformática. Esta comunicación va establecida tanto con el propio cliente como con el resto del equipo de desarrollo encargado de la fase de gestión de los requerimientos

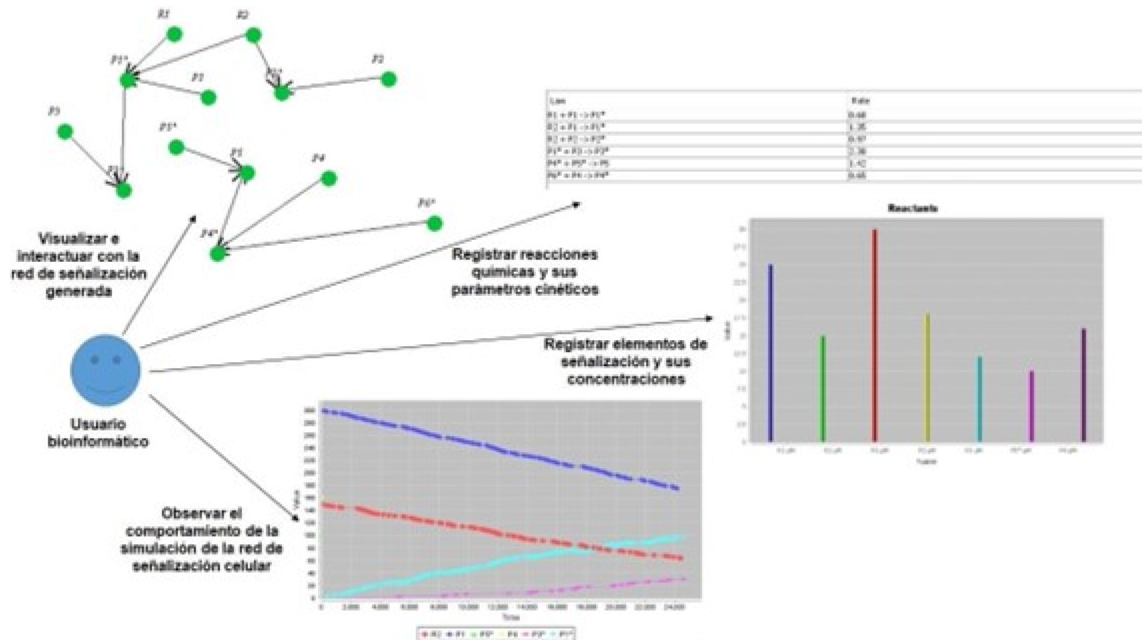


Figura 4.6. Uso de prototipos como artefacto de comprensión y comunicación de las necesidades del cliente de la herramienta de simulación bioinformática. Esta comunicación va establecida tanto con el propio cliente como con el resto del equipo de desarrollo encargado de la fase de gestión de los requerimientos

Como puede apreciarse en las figuras 4.5 y 4.6, los analistas traducen esta primera recolección de los requerimientos a una notación comprensible por ambas partes; es decir, tanto por el cliente como por el equipo de desarrollo. Nos referimos a los diagramas de casos de uso (figura 4.5) complementados con algunos de los gráficos que debe producir la herramienta de simulación bioinformática (figura 4.6).

4.3.2. Arquitectos

El arquitecto de software es también desarrollador de software. El papel del arquitecto exige un amplio dominio de los llamados *soft skills* o habilidades blandas. Como parte de su rol en el equipo de desarrollo, el arquitecto de software debe ejercer las siguientes funciones (González-Pérez, P.P., Gómez Fuentes, M.C., Cervantes Ojeda, 2023): 1) definir la arquitectura lógica del software, 2) definir la arquitectura física del software, 3) gestionar los requerimientos no funcionales, 4) seleccionar la tecnología y el manejo de los riesgos tecnológicos, 5) mejorar de manera continua la arquitectura del

software (tanto lógica como física) y 6) sugerir y colaborar en la implementación de la arquitectura lógica del software para asegurar que todos los aspectos de la arquitectura estén implementándose de forma correcta.

El arquitecto debe ser capaz de comunicarse tanto gerencialmente como al nivel del usuario y, por supuesto, expresarse de manera fluida en el campo técnico. Además, necesita ser empático y tener gran capacidad de manifestar ideas utilizando diferentes herramientas y lenguajes de modelado.

Una forma de comprender la comunicación entre diseñadores y arquitectos basada en los artefactos y modelos, es cuando una vista estática en el diseño, expresada a través de un diagrama de clases, se convierte en un diseño de arquitectura lógica y física en manos del arquitecto de software. Para ilustrar lo anterior, regresemos al caso de estudio del sistema de reservaciones y ventas de una aerolínea, y consideremos el modelo de diseño que ilustra la figura 2.20, el cual —aunque no constituye aún un modelo de arquitectura del software— muestra detalles estructurales correspondientes a una fase de diseño avanzada. Ahora bien, como se puede apreciar en la figura 3.6, el modelo de diseño liberado por los diseñadores se convierte, en manos del arquitecto, en un modelo de arquitectura lógica del software basado en una versión extendida del patrón arquitectónico MVC. El caso de estudio relacionado con el desarrollo de la herramienta de simulación bioinformática es otro escenario que nos permite ilustrar la comunicación entre diseñadores y arquitectos, al considerar el modelo de diseño detallado mostrado en la figura 2.21 y el correspondiente diseño arquitectónico basado en el patrón MVC extendido a cuatro capas mostrado en la figura 3.8.

4.3.3. Responsables de las pruebas

Como parte de su papel en el equipo de desarrollo, el responsable de las pruebas debe ejercer, entre otras funciones, la aplicación de las pruebas del software relacionadas con la recolección de los requerimientos, el análisis de los requerimientos, el diseño arquitectónico, el diseño detallado y la codificación.

El principal objetivo del responsable de las pruebas es desarrollar y ejecutar las pruebas que validarán que la funcionalidad desarrollada se desempeña como es esperado. Es común que el responsable de las pruebas establezca la comunicación con los otros miembros del equipo de desarrollo mediante diagramas de casos de uso e historias de usuario, además de

diagramas de transición de estado, diagramas de secuencia y diagramas de clases. De la información recabada se generan reportes fácilmente entendibles tanto por el equipo de ingeniería como por los usuarios.

V. EL CICLO DE VIDA EN EL DESARROLLO DE SISTEMAS DE SOFTWARE

Hasta ahora hemos hablado de modelado, arquitectura y comunicación a lo largo del ciclo de desarrollo de sistemas de software como tres aristas esenciales que, cuando son abordados de forma correcta, constituirán sin lugar a dudas un factor de éxito. En este punto, será necesario explorar los principios y conceptos que encierra en sí el propio ciclo de desarrollo de sistemas de software.

El ciclo de vida de desarrollo de sistemas (CVDS) (Pressman, R. S., 2010; Sommerville, I. 2011; Pfleeger, S. L., 2002; Gómez Fuentes, M.C., Cervantes Ojeda, J., González- Pérez, P.P., 2019) es un proceso de planeación, construcción, implementación y pruebas de sistemas de software, en el cual se considera también el tipo de hardware en el que se ejecutará. A esta primera aproximación debemos agregar el mantenimiento como una de las etapas a tomar en cuenta, pues si bien existen sistemas de vida fugaz, la mayoría del software necesitará mantenimiento continuo.

El CVDS es una representación global del proceso de desarrollo de software y determina el orden en el que se llevarán a cabo las fases y actividades propias de un proceso de este tipo, tales como gestión de los requerimientos, diseño arquitectónico, diseño detallado, implementación, pruebas, entre otras (Gómez Fuentes, M.C., Cervantes Ojeda, J., González-Pérez, P.P., 2019). Además, el CVDS indica el orden de las etapas involucradas en el desarrollo del software y nos proporciona un criterio para comenzar, pasar a la siguiente etapa, iterar —de ser el caso— y finalizar.

Como puede apreciarse en la figura 5.1, el CVDS puede verse como un proceso comúnmente conformado por las siguientes siete etapas:

- Planeación
- Gestión de los requerimientos
- Diseño
- Construcción
- Pruebas
- Despliegue

- Mantenimiento

Figura 5.1. Etapas del ciclo de vida de desarrollo de software

Una vez relacionadas las etapas del CVDS, es importante hacer notar que el orden de estas fases no es una receta con un proceso único; por el contrario, a lo largo de los años han sido propuestos diferentes modelos de implementación del CVDS, y estamos seguros de que seguirán emergiendo más en el futuro próximo. En general, aunque son muchos los modelos de CVDS, es posible agruparlos en tres principales categorías:

- Modelos de cascada
- Modelos incrementales
- Modelos iterativos

Además, estos modelos pueden clasificarse según su grado de predictibilidad y adaptabilidad de las siguientes formas:

- Modelos predictivos
- Modelos adaptativos

En los modelos predictivos, el usuario tiene claro lo que busca resolver; es decir: los requerimientos no tienen cambios y dan paso a una especificación y alcance fijos. Por otra parte, en los modelos adaptativos el usuario tiene idea de lo que quiere o necesita, pero no tiene una certeza absoluta, por lo que este tipo de modelos trabaja con base en artefactos que son evaluados por el usuario con la idea de obtener retroalimentación para la siguiente iteración. Mientras que los modelos en cascada se caracterizan por ser predictivos, los modelos iterativos e incrementales exhiben un comportamiento adaptativo.

Dada la naturaleza de los proyectos de software, es muy poco común encontrar modelos que sean puramente predictivos o adaptativos, puesto que

ser inflexible con los cambios en un proyecto (modelos ciento por ciento predictivos) puede derivar en un producto que no aporte valor al usuario. Es decir, restringir a una única entrada (recolección de requerimientos) no nos dejaría reaccionar a un posible cambio en los requerimientos que aporte más valor al usuario, por lo que el resultado del proyecto sería un software con poco o nulo valor. Por otra parte, y en el extremo opuesto, abrir la posibilidad a cambios indiscriminados nos llevaría a agotar los recursos destinados al proyecto y, quizás, a su cierre.

A continuación relacionaremos brevemente algunos modelos que han ido emergiendo a lo largo de la historia del desarrollo de software, así como sus principales características. Es necesario tomar en consideración que, aunque existen muchos modelos de desarrollo de software, resulta posible agruparlos como cascada, incrementales e iterativos. Esto es esperable porque —como sucede con muchas otras técnicas— los modelos han ido evolucionando, lo que ha dado pie a nuevos modelos.

5.1. Modelos de cascada

Los modelos de tipo cascada son una secuencia de etapas donde una etapa precede por completo a la etapa subsecuente; es decir, las etapas no pueden traslaparse, al menos en el modelo de cascada pura. Este tipo de modelo se ajusta bien a aquellos desarrollos donde no se espera que los requerimientos presenten cambios durante el tiempo de vida del software y, por lo tanto, dan paso a una especificación y alcance fijos. Este tipo de proyectos, aunque poco comunes, suelen ser de desarrollo de software donde el alcance está en absoluto acotado, pues atienden o están pensados para atender funcionalidades que son rutinas repetitivas de negocios.

5.1.1. Modelo de cascada pura

El modelo de cascada pura (Pressman, R. S., 2010; Sommerville, I. 2011) constituye uno de los primeros modelos de ciclo de vida que se ha seguido para desarrollar software. Ha sido muy usado durante muchos años, dada su baja curva de aprendizaje y rápida implementación. La cascada pura puede clasificarse como altamente predecible, pues dada una entrada, la salida debería ser única. Este tipo de modelo es ideal para desarrollos repetitivos donde los requerimientos son fijos. Se llama "cascada" porque cada una de sus fases debe ser completada antes de comenzar la siguiente. Al finalizar cada etapa debe haber un proceso de validación que habilita el paso a la siguiente etapa. Sin embargo, es posible que una vez comenzada la etapa subsecuente pueda regresarse a la anterior en caso de haberse detectado alguna anomalía u omisión. En la figura 5.2 se ilustran las etapas que suele integrar el ciclo de vida de cascada pura, mientras que en la figura 5.3 se relacionan las principales ventajas y desventajas que lo caracterizan.

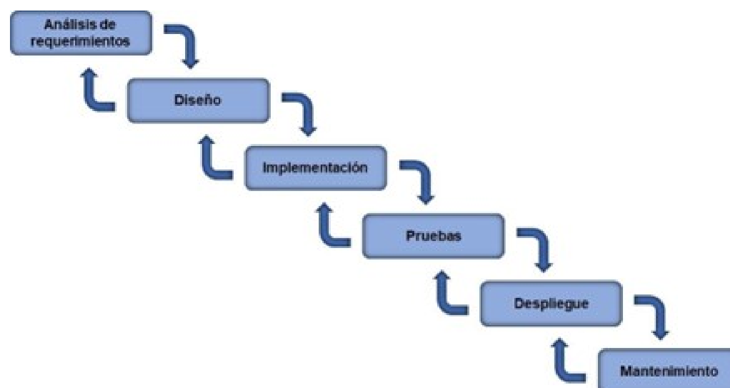


Figura 5.2. Etapas del ciclo de vida de cascada pura

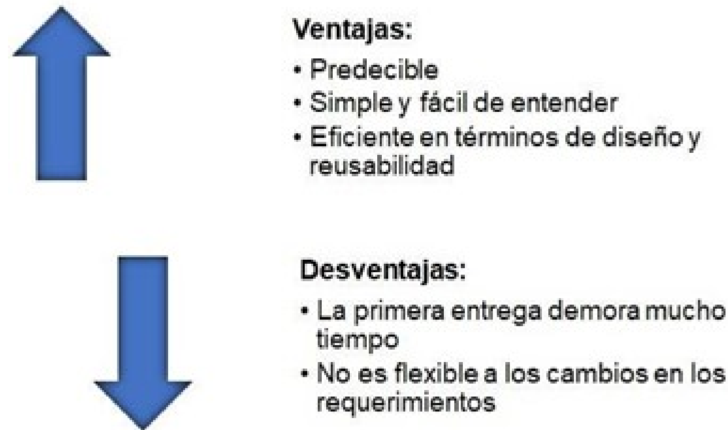


Figura 5.3. Ventajas y desventajas del modelo de cascada pura

5.1.2. Modelo en V

Una de las principales desventajas del modelo de cascada pura es que la fase de pruebas comienza en una etapa muy tardía del proyecto, por lo que se eleva considerablemente el riesgo de haber pasado por alto algún requerimiento o de que una inadecuada especificación de este se detecte demasiado tarde. El modelo en V es muy similar al modelo de cascada pura (Pressman, R. S., 2010; Sommerville, I. 2011). Sin embargo, como puede apreciarse en la figura 5.4, el modelo en V corre en paralelo etapas de validación para las fases de análisis de requerimientos y diseño, lo cual permite llegar a la implementación con una especificación bastante robusta. No obstante, el modelo en V no se libra de introducir riesgos similares a los ya identificados en el modelo de cascada porque también es un modelo predictivo. En la figura 5.5 se relacionan las principales ventajas y desventajas del modelo en V.

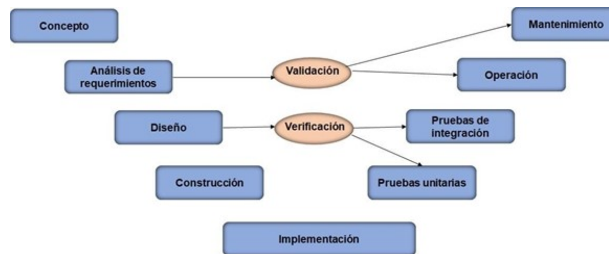


Figura 5.4. Etapas del ciclo de vida del modelo en V

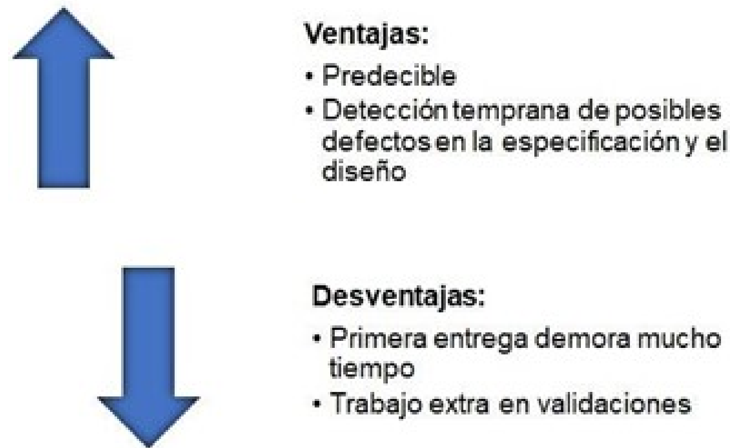


Figura 5.5. Ventajas y desventajas del modelo en V

5.1.3. Modelo Sashimi

El modelo Sashimi o modelo en cascada con traslape (Pressman, R. S., 2010; Sommerville, I. 2011), al igual que el modelo en V, es un modelo de cascada. Sin embargo, lo que lo hace diferente de la cascada pura es que da la posibilidad de comenzar una etapa siguiente sin la necesidad de esperar a que la etapa precedente se complete; es decir, las fases pueden traslaparse (ver figura 5.6). Por ejemplo, una vez recolectados requerimientos de alto nivel, podrían elegirse algunos de ellos, refinarlos y trabajar en su diseño —e, incluso, en su implementación— sin que sea indispensable que la etapa de análisis de requerimientos haya concluido por completo.

Con lo anterior se consigue una retroalimentación del diseño e implementación hacia la etapa de requerimientos. De esta forma, aunque el modelo sigue siendo una cascada, no es puramente predictivo, sino que incorpora matices de los modelos adaptativos. Este tipo de modelo responde mejor que el modelo de cascada pura y el modelo en V a proyectos donde se necesita comenzar lo antes posible con el desarrollo. En la figura 5.7 se relacionan las principales ventajas y desventajas del modelo Sashimi.

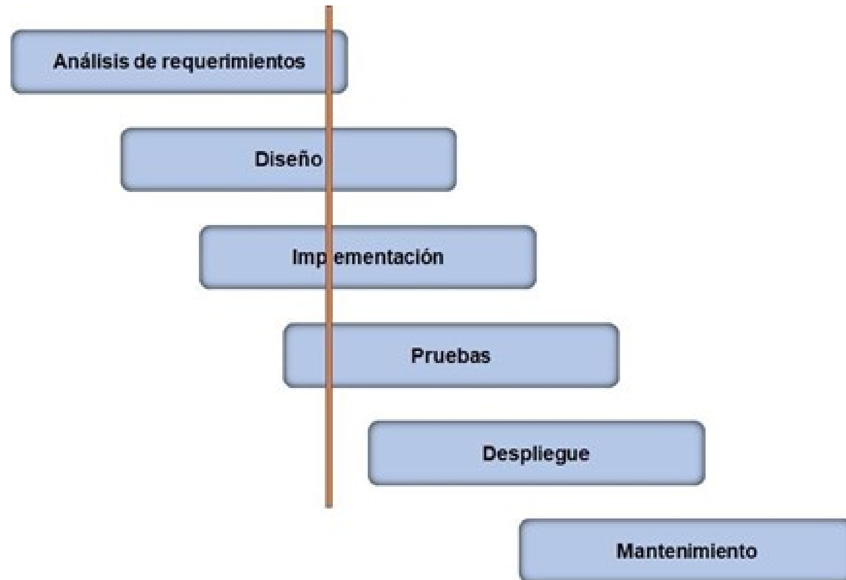


Figura 5.6. Fases del modelo Sashimi

Figura 5.7. Ventajas y desventajas del modelo Sashimi

5.2. Modelos incrementales

Como puede apreciarse en la figura 5.8, los modelos incrementales (Cockburn, A. 2008; Sommerville, I., 2011) se basan en la construcción incremental, y este proceso puede hacerse de manera adaptativa o predictiva. Cuando se sigue el enfoque adaptativo, se analiza, diseña, implementa, prueba y despliega iterativamente hasta que se llega al resultado deseado, lo que permite ir ajustando cualquier desviación en el producto de una iteración precedente durante la iteración sucesiva. Cada incremento puede corresponder a un módulo, una funcionalidad global o a un conjunto de requerimientos muy relacionados entre sí. Por otra parte, cuando se sigue el enfoque predictivo, los requerimientos y —en ocasiones— el diseño se realizan al inicio del proyecto, y una vez que esas etapas finalizan, se procede a la construcción del software mediante ciclos incrementales. En la figura 5.9 se describen las principales ventajas y desventajas de los modelos incrementales.

En la figura 5.10 se ilustra el uso del modelo incremental predictivo durante el desarrollo del sistema de reservaciones y ventas de una aerolínea. Nótese en esta figura cómo las fases de gestión de los requerimientos y diseño arquitectónico se llevaron a cabo al inicio del proyecto, y después se construyó el sistema de software mediante ciclos incrementales, donde cada incremento correspondió a la construcción de un nuevo módulo o macrofuncionalidad.

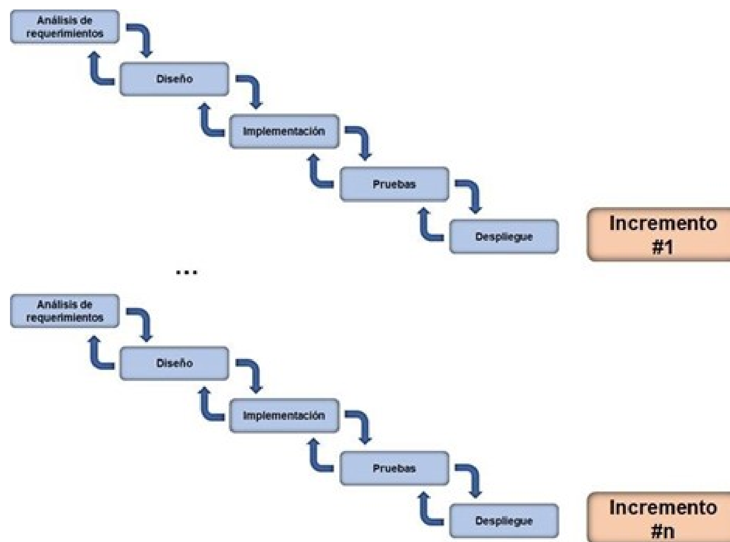


Figura 5.8. Modelos incrementales basados en la construcción incremental del producto software

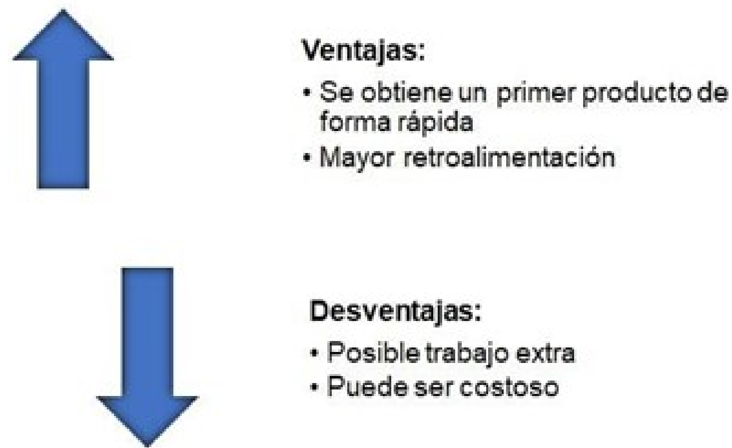


Figura 5.9. Ventajas y desventajas de los modelos incrementales

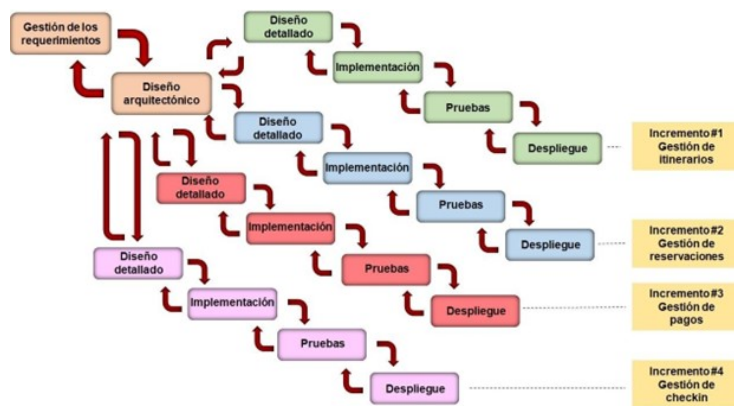


Figura 5.10. Uso del modelo incremental predictivo durante el desarrollo del sistema de reservaciones y ventas de una aerolínea

5.3. Modelos iterativos

Como su nombre lo indica, los modelos iterativos se caracterizan por introducir el concepto de *iteración* como parte de las principales fases del proceso de desarrollo. Es decir, cuando se sigue el enfoque de modelo iterativo, varias fases del proceso de desarrollo se componen de iteraciones. En cada iteración se refinan productos de análisis, diseño o implementación obtenidos en la iteración anterior o se crean nuevos de estos productos. Dos modelos representativos de esta categoría son el Proceso Unificado de Desarrollo de Software (UP, por su nombre en inglés) (Jacobson, I., Booch, G., Rumbaugh, J., 2000) y el modelo espiral (Pressman, R. S., 2010; Sommerville, I. 2011).

5.3.1. El Proceso Unificado de Desarrollo de Software

Más que un proceso, UP (Jacobson, I., Booch, G., Rumbaugh, J., 2000) puede considerarse un marco de desarrollo que propone una aproximación diferente a los enfoques que siguen los modelos anteriores, lo cual es notorio a simple vista en el diagrama que se muestra en la figura 5.11, donde pueden apreciarse las fases que componen a UP (incepción o inicio, elaboración, construcción y transición), así como los métodos/actividades de desarrollo de software que integra, tales como gestión del proyecto, modelado de negocio, análisis de requerimientos, diseño, implementación, pruebas, entre otros. Un detalle importante que hay que notar es que cada una de las fases (excepto la de inicio) puede tener más de una iteración, lo que permite obtener retroalimentación de forma continua. Esto nos brinda la oportunidad de ir ajustando nuestra solución. El traslape de los métodos/actividades a lo largo de las fases también es muy característico de este modelo.

UP se centra en la mitigación de riesgos por medio de un especial énfasis en la arquitectura; es decir, produce un prototipo funcional de la arquitectura desde etapas muy tempranas. Además, UP tiende a ser más adaptativo que predictivo, aunque esto puede variar dependiendo de los modelos y artefactos que utilicemos en su implementación, decisión que tendrá que tomarse en función de las necesidades del proyecto. Incluso pueden usarse diferentes modelos a lo largo de sus etapas. Por ejemplo, puede usarse una

aproximación ágil en la fase de elaboración y un modelo Sashimi en la fase de construcción.

En la fase de inicio se establece la razón de ser del proyecto y se determina su alcance, se adquiere una visión aproximada del proyecto en términos de los principales requerimientos, se desarrolla un primer análisis del negocio y se efectúan las primeras estimaciones, aún no precisas, del proyecto. La fase de inicio no es precisamente una fase de gestión de requerimientos sino una especie de fase de viabilidad.

La fase de elaboración consta de varias iteraciones. En esta se fomenta la recolección de requerimientos más detallados, se realiza el análisis y diseño de alto nivel con el objetivo de definir la arquitectura base del sistema, se inicia la construcción del sistema, se refina la visión que se tiene del proyecto, se identifican más requerimientos, se determina de forma más refinada el alcance del proyecto, se efectúa el análisis y mitigación de los principales riesgos.

Al igual que la fase de elaboración, la fase de construcción consta de varias iteraciones. A lo largo de esta fase continúan desarrollándose muchas de las actividades iniciadas en la fase de elaboración. En cada iteración se construye (diseño, implementación y pruebas) software que satisface un subconjunto de los requerimientos del proyecto. El software construido en cada iteración es software de calidad probado e integrado. La fase de construcción enfatiza las actividades de pruebas, despliegue, configuración y gestión de cambios.

La fase de transición también consta de varias iteraciones. Se centra, sobre todo, en las pruebas del sistema, el refinamiento del desempeño del sistema, así como en el lanzamiento del sistema para su producción, aunque es posible continuar con las actividades de construcción que no hayan sido concluidas en la fase homónima.

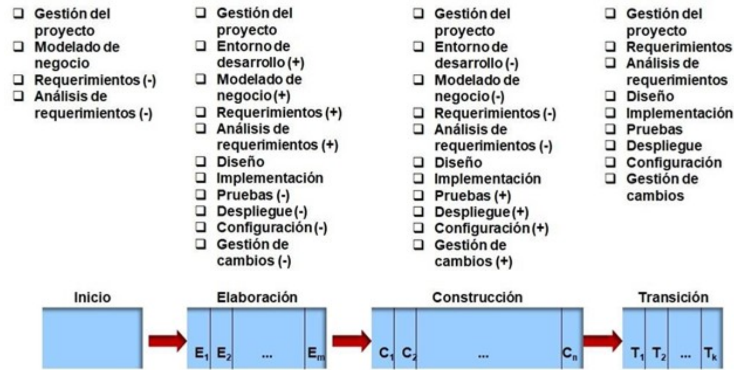


Figura 5.11. El Proceso Unificado de Desarrollo de Software (UP) como modelo iterativo

Aunque UP, en comparación con los modelos de ciclo de vida ya vistos, puede ser considerado complejo de implementar, debido a su popularidad y su robustez ha tenido diferentes variantes, lo que lo ha hecho más versátil y accesible a proyectos de software de pequeña y mediana escalas. Las variantes más conocidas de UP son las siguientes:

- Rational Unified Process
- Enterprise Unified Process
- Open UP-Versión ligera
- Agile UP-Versión ligera

Las principales ventajas y desventajas de UP se relacionan en la figura 5.12.

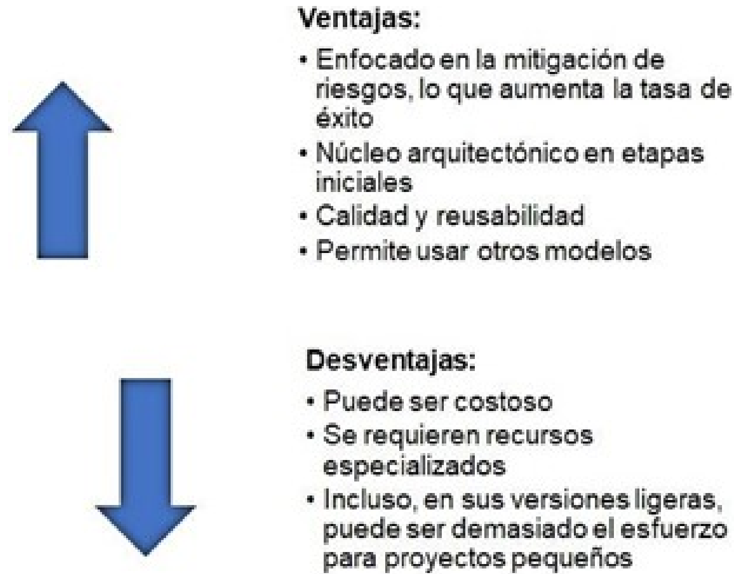


Figura 5.12. Ventajas y desventajas del Proceso Unificado de Desarrollo de Software

En la figura 5.13 se muestra el uso del modelo UP durante el desarrollo de la herramienta de simulación bioinformática Big Data-Cellulat. Como puede observarse en esta figura, la fase de elaboración abarcó cinco iteraciones, y enfatizó actividades como el modelado del negocio, la recolección de los requerimientos y el análisis de los requerimientos, aunque actividades como diseño, implementación y pruebas también fueron desarrolladas. La fase de construcción integró cuatro iteraciones, a través de las cuales se refinaron y completaron actividades iniciadas en la fase de elaboración, y se prestó particular importancia a las actividades relacionadas con el despliegue, la configuración y la gestión de cambios. Al final, la fase de transición requirió solo dos iteraciones, principalmente centradas en las pruebas, despliegue, configuración y gestión de cambios.

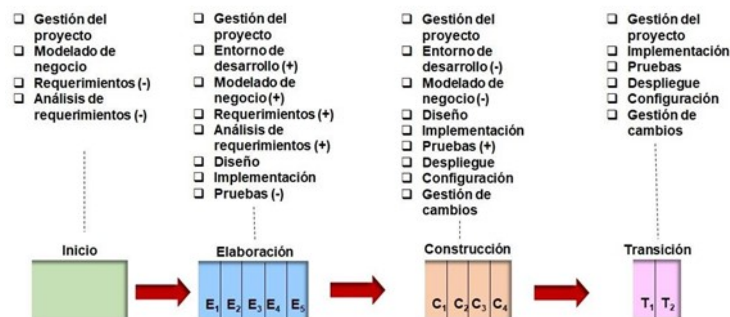


Figura 5.13. Uso del modelo UP durante el desarrollo de la herramienta de simulación bioinformática Big Data-Cellulat

5.3.2. El modelo en espiral

Otro ejemplo de modelo iterativo muy difundido en la industria del software es el modelo en espiral, el cual está orientado a riesgos, y divide el proyecto de software en un grupo de subproyectos. Comúnmente, cada subproyecto se centra en uno o más riesgos importantes hasta que todos estén controlados. Un riesgo puede referirse a requerimientos poco comprensibles, arquitecturas poco comprensibles, problemas de ejecución importantes, problemas con la tecnología a utilizar, entre otros. Como puede apreciarse en la figura 5.14, el modelo en espiral contempla cuatro principales fases: planeación, análisis de riesgos, construcción y retroalimentación. En la figura 5.15 se describen las principales ventajas y desventajas del modelo en espiral.

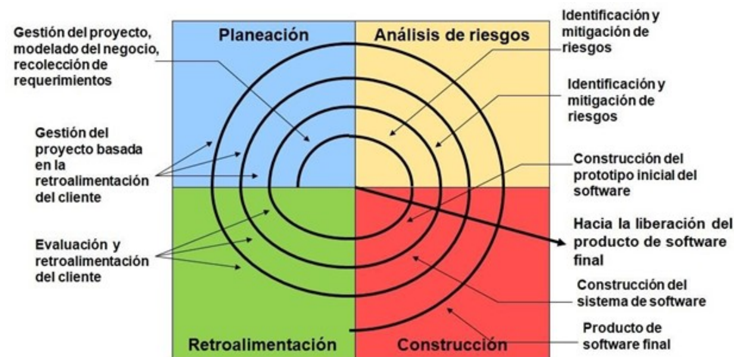
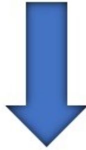


Figura 5.14. El modelo en espiral

Ventajas:

- Enfocado en la gestión de riesgos, lo que aumenta la tasa de éxito Mientras más tiempo y recursos se empleen, menores serán los riesgos
- Se puede iniciar un proyecto con una serie de iteraciones para reducir los riesgos
- Se pueden incorporar otros modelos de ciclo de vida como iteraciones de la espiral



Desventajas:

- Puede ser costoso
- Se requieren recursos especializados
- Se trata de un modelo complicado, el cual requiere de una gestión concienzuda, atenta y que exige conocimientos profundos.

Figura 5.15. Ventajas y desventajas del modelo en espiral

VI. METODOLOGÍAS DE DESARROLLO DE SOFTWARE ORIENTADAS A MEJORAR LA VELOCIDAD Y LA CALIDAD DE LA ENTREGA DE VALOR

Una vez discutido el importante papel que tienen el modelado, la arquitectura, la comunicación y los ciclos de vida en el desarrollo de software, veamos qué nos ofrecen las metodologías de desarrollo de software orientadas a mejorar la velocidad y la calidad de la entrega de valor, tales como Lean (Ries, E., 2011; Coplien, J. O. y Bjørnvig, G.,2011; Forsgren, N., Humble, J. y Kim, G., 2018), Agile (Agile Alliance, 2013) y DevOps (Brown, D. 2015; Forsgren, N., Humble, J. y Kim, G., 2018), y cómo sacar la máxima ventaja en la aplicación de estas prácticas.

6.1. Lean versus Agile

Lean y Agile son movimientos que han acompañado a la industria de desarrollo de software a lo largo de los últimos 20 años, y aunque es bastante común encontrarlos como parte de la estrategia y del conjunto de herramientas de cualquier equipo moderno de desarrollo de software, es importante señalar que su aplicación no está ligada exclusivamente al software. De hecho, Lean tiene su origen en la manufactura de autos (Ohno, T., 1988), que con el tiempo evolucionó en lo que conocemos hoy como *Lean manufacturing* (Abu Bakar, K. et Al., 2022), donde los principios Lean se aplicaron a muchas más industrias. Así como con Lean, los principios de Agile también pueden ser aplicados a cualquier iniciativa o proyecto sin importar su naturaleza. Quizá por eso sea que a menudo encontramos que ambos conceptos se confunden o se usan de manera errónea. Exploremos de manera breve las características de ambos movimientos, sus diferencias, similitudes y aporte al desarrollo de proyectos exitosos de software.

Como mencionamos antes, Lean y Agile en su forma más pura son movimientos o corrientes que obedecen a un conjunto de principios; Estos, a su vez, se han implementado en frameworks y metodologías creadas para diferentes industrias y emprendimientos a lo largo de los años. Para Lean, hay muchas iniciativas donde los principios Toyota han sido retomados, pero una de las más importantes es el libro *The Lean Startup* (Ries, E., 2011), cuyos valores y aplicación fueron abordados de tal forma que la industria en general se volcó a su implementación. Agile, por su parte, se rige por el Manifiesto Ágil (Agile Manifesto, https://agilemanifesto.org/iso/es/manifesto.html), que fue publicado en 2001 y que —al contrario de Lean— nació en el ámbito del desarrollo de proyectos de software, pero que —con su evolución— ha sido replicado en diferentes industrias.

Es importante entender que tanto Lean como Agile no garantizan mejorar la velocidad de un proyecto; es más, podemos afirmar que en organizaciones sin experiencia en su aplicación, al inicio serán una sobrecarga que desacelere el ritmo del proyecto. La idea principal que transmiten Lean y Agile es que debemos ser eficientes en nuestra forma de trabajar y, por ende, de potenciar la entrega de valor.

6.1.1. Diferencias entre Lean y Agile

Lean responde a la pregunta *¿qué vamos a desarrollar?*, y usa la experimentación como medio para descubrir la relevancia del producto a desarrollar, pero lo hace de forma eficiente y con la mejor calidad posible, con lo que se evita el desperdicio de recursos. Podemos resumir lo anterior en los siguientes puntos (ver figura 6.1):

- Maximizar calidad
- Ciclos de aprendizaje continuo
- Aprender a través de la experimentación en mercados reales
- Ajustar la estrategia en respuesta al aprendizaje adquirido
- Medir el éxito del experimento en mercados reales

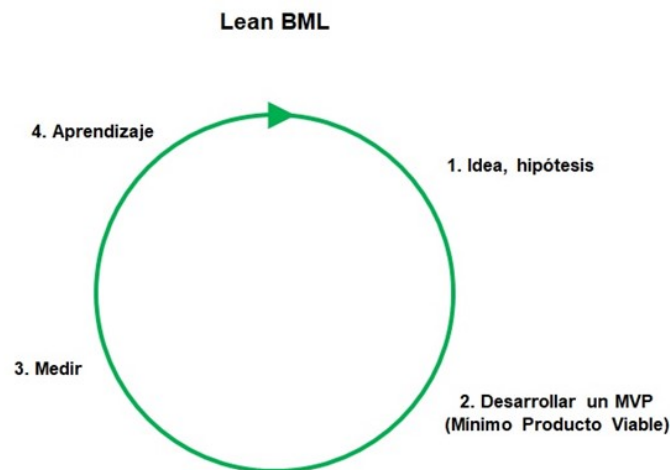


Figura 6.1. Principales aspectos considerados en Lean

Por otra parte, Agile se centra en cómo vamos a desarrollarlo; es decir, una vez entendido e identificado el objetivo, Agile plantea una serie de valores de los cuales se puede echar mano para mejorar la forma de colaborar y llevar a buen puerto el proyecto. Los siguientes puntos resumen los valores de Agile (ver figura 6.2):

- Mejorar e incrementar la colaboración multidiciplinaria
- Minimizar el trabajo en progreso

- Trabajar en ciclos iterativos que conlleven a cambios de forma progresiva y obtener retroalimentación de ellos
- Retrospectiva

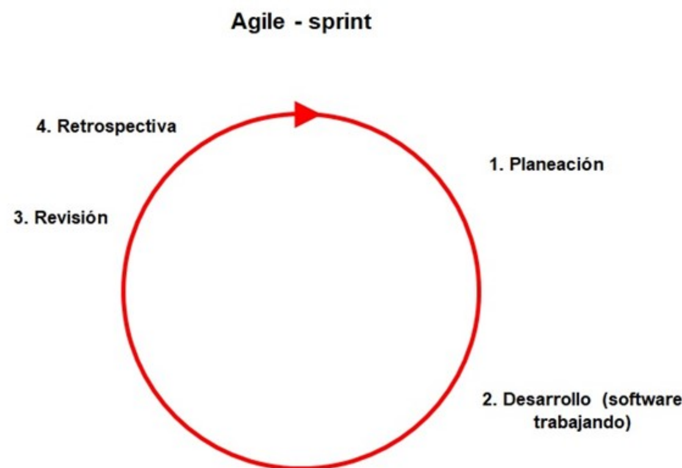


Figura 6.2. Principales aspectos considerados en Agile

En ambas aproximaciones los ciclos cortos forman parte del éxito de su implementación. Para Agile estos ciclos son conocidos, en general, como *sprints*, mientras que en Lean se les conoce como ciclos *BML* (por sus siglas en inglés), que significa *construye, mide, aprende*.

Tanto Lean como Agile cuentan con diversas implementaciones que gozan de gran popularidad en diferentes industrias. Por ejemplo, en Agile encontramos a Scrum, que durante muchos años ha sido usado en la organización de tareas y seguimiento en equipos de desarrollo de software. Scrum usa un tablero lógico donde las características (requerimientos funcionales y no funcionales) que conforman el proyecto son listados; entonces, el equipo se reúne y vota por los elementos de la lista que serán implementados en el siguiente sprint (el equipo acuerda la duración del sprint, y se deja fija por el resto del proyecto).

Una vez que el sprint concluye, el equipo se reúne y hace una presentación de los hitos alcanzados; luego se desarrolla una dinámica de retrospectiva donde se tocan puntos que incidirán en la mejora continua del proyecto. Después vuelve a votarse por los elementos que se incluirán en la siguiente iteración, y el ciclo se repite. Este tipo de iteraciones permite que en el enfoque Lean se analice y aprenda en cada iteración, por lo que es fácil

deducir que tanto Lean como Agile se complementan y se nutren uno del otro, pues el enfoque de no desperdiciar y sí agregar valor en cada iteración son factores que inciden directamente en los elementos elegidos para el siguiente sprint.

Además de Scrum, Kanban es otro framework que goza de gran popularidad entre los equipos de operaciones, dado que este se presta más a tareas de mantenimiento de software ya existente, así como de infraestructura.

Para ilustrar de forma parcial la aplicación de estas metodologías de desarrollo de software orientadas a mejorar la velocidad y la calidad de la entrega de valor, consideremos como escenario el ya conocido sistema de reservaciones y ventas de una aerolínea. Usando un enfoque Agile podríamos listar los siguientes requerimientos como elementos del tablero backlog.

Sistema de reservaciones y ventas de una aerolínea (backlog del proyecto):

- Hacer checkin
- Generar una reservación
- Realizar pago con tarjeta de crédito en mostrador
- Realizar pago con tarjeta de crédito en línea
- Imprimir pase de abordar
- Consultar estatus de la reservación
- Cambiar itinerario

La elección de las historias de usuario se hace basado en ponderación de importancia y tamaño del elemento. El propietario del producto decide cuáles elementos del tablero son más importantes y podrían generar más valor si se incluyen en el sprint. Los miembros del equipo evalúan el tamaño de cada historia y, según su capacidad, las incluyen en el siguiente sprint. Suponiendo que tenemos un equipo de tres personas, y que dada su experiencia han decidido que solo pueden aceptar 15 puntos, los elementos a desarrollar en el siguiente sprint serían los siguientes:

Sistema de reservaciones y ventas de una aerolínea (sprint backlog):

- Generar una reservación
- Realizar pago con tarjeta de crédito en línea

La técnica de ponderación de cada uno de los elementos del tablero es una combinación de experiencia tanto individual como de equipo, así como el seguimiento de métricas en otros proyectos similares. Por lo anterior, es común que este tipo de actividad vaya refinándose a medida que el proyecto avanza; sin embargo, es muy importante —aún si no se cuenta con la experiencia— que se cree una primera aproximación y que de ahí se generen métricas que ayuden a dirigir el avance del proyecto.

6.2. DevOps

En palabras de Donovan Brown (Brown, D. 2015), DevOps es la unión de personas, procesos y productos que habilitan la entrega continua de valor al usuario final, y aunque existen muchas otras definiciones, colaboración y entrega de valor son elementos base en la mayoría de ellas. DevOps, como movimiento, nació en respuesta a los conflictos que había entre los equipos de desarrollo y operaciones. Dichos conflictos se originaban en la premura del equipo de desarrollo por entregar nuevo software, y por la necesidad de mantener estable el software antes desplegado en los servidores por el equipo de operaciones.

En otras palabras, el equipo de desarrollo estaba concentrado en acelerar la entrega de cambios, y el de operaciones tenía la misión de reducir la cadencia de despliegues con la idea de asegurar la estabilidad del software existente en producción. Aunque con estrategias diferentes, lo que ambos equipos finalmente buscaban era entregar valor al usuario final, motivo por el cual la colaboración entre ambos equipos tuvo un papel fundamental en el origen de DevOps como movimiento (ver figura 6.3).

Figura 6.3. Principales actividades integradas en DevOps

Podemos resaltar que, de esta colaboración, ambos equipos intercambiaron prácticas y herramientas. Por poner un ejemplo, el equipo de operaciones adoptó el uso de repositorios de código base aplicados a la definición de infraestructura, práctica que luego fue conocida como *código* o *Ia C*, por sus siglas en inglés.

Del lado de desarrollo, los equipos adquirieron prácticas, como el uso de contenedores para replicar ambientes, y el uso de herramientas de automatización como, por ejemplo, servidores de integración continua, software de análisis estático y herramientas de testing que permitieron mover responsabilidades, que en un inicio se encontraban en el tercio del equipo de operaciones, al principio de la cadena de valor, justo donde el desarrollo comienza la definición misma del requerimiento.

Entonces, hemos hablado de DevOps como movimiento, pero también es fácil inferir —por su definición— que DevOps comparte principios y valores con Lean, además de que se vale de las prácticas de Agile para cumplir sus objetivos. Sin embargo, es importante resaltar que además de estos principios y prácticas, DevOps encuentra en el acrónimo CI-CD dos de sus prácticas más importantes: la integración continua y la entrega continua.

6.2.1. La integración continua en DevOps

Integración continua, o *CI*, es una práctica mediante la cual los equipos de desarrollo y operaciones realizan cambios al código base en un repositorio compartido entre los miembros del equipo (ver figura 6.3). Dichos cambios pasan a través de una batería de pruebas unitarias; cuando sean evaluados satisfactoriamente, se integrarán, y con ello se disparará un nuevo conjunto de pruebas de integración, con lo que se garantiza que los cambios introducidos en el repositorio no rompan el software que está funcionando o, en el caso de los equipos de operaciones, el ambiente que ha sido desplegado usando infraestructura como código. Como es de imaginarse, cuando alguna de las pruebas ejecutadas presenta algún error, el servidor de integración continua deberá reportar los problemas encontrados y el equipo de desarrollo o —en su caso— el de operaciones tendrá la obligación de corregir dicho error. Ese tipo de estrategia es conocida como *ciclo de retroalimentació*n y también es parte de las estrategias de Lean.

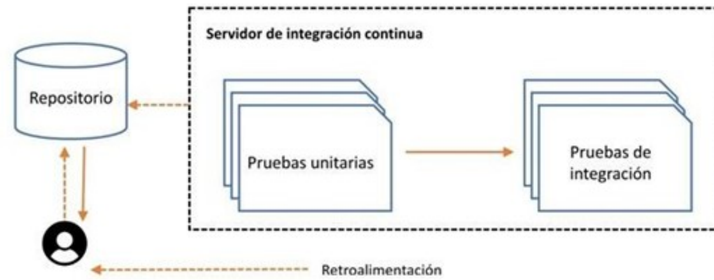


Figura 6.3. Integración continua (CI) en DevOps

6.2.2. La entrega continua en DevOps

Hablemos ahora de la entrega continua o CD. Esta práctica se encarga de llevar el código que ha sido convertido en software, mediante la integración continua, a los diferentes ambientes con los que el equipo de desarrollo y operaciones cuentan (ver figura 6.4). Podríamos pensar en un ambiente de desarrollo, de calidad, de preproducción y —al final— de uno productivo.

Es habitual encontrar que el software o el servidor de integración continua también soporta las tareas de despliegue a los distintos ambientes. Jenkins es un ejemplo de este tipo de software de automatización. Como es de esperarse, el proceso de llevar nuevas versiones del software desarrollado a los distintos ambientes involucra casos de prueba que, al ser pasados con éxito, aseguran la calidad del software desplegado, y en caso contrario dan retroalimentación al equipo de desarrollo sobre los cambios que deben realizarse en etapas tempranas.

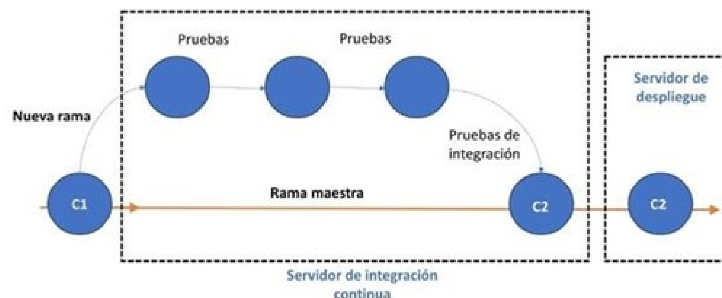


Figura 6.4. Entrega continua (CD) en DevOps

A medida que uno a uno de los ambientes se evalúan satisfactoriamente, banderas (por lo general llamadas *compuertas de aprobación*) son marcadas en verdadero para que el software de automatización entienda que es

momento de desplegar la versión de software en el siguiente ambiente. Es deseable que esta práctica, en la medida de lo posible, sea automatizada; sin embargo, en ambientes superiores, como producción o espejos de este, casi siempre las compuertas de aprobación están gobernadas por humanos que se encargan de dar el visto bueno en dependencia de las necesidades del negocio o de su estrategia para el despliegue de la próxima versión de software. No obstante, empresas como Netflix, Amazon y otras muchas tienen su proceso de entrega continua automatizado por completo, lo que permite que de forma directa se desplieguen a producción nuevas versiones de software en múltiples ocasiones durante la jornada. A la práctica de entregar software en producción de forma automatizada y sin intervención humana se le conoce como *despliegue continuo*.

Tanto la integración continua como la entrega continua han marcado un hito en la forma como los equipos de alta productividad trabajan. Por ello es importante que equipos involucrados en el desarrollo de software, sin importar su tamaño, adquieran este tipo de prácticas y las usen de forma correcta.

Por último, es importante que comprendamos que —como en todo movimiento— aunque los principios DevOps parezcan bien definidos, su interpretación e implementación es responsabilidad del equipo y debe ajustarse a las características tanto del equipo como del proyecto en cuestión. Un ejemplo de la evolución de este movimiento podría ser DevSecOps, que es una extensión de DevOps con énfasis en traer la seguridad al inicio de la cadena de valor, lo que crea aplicaciones seguras por definición. Más recientemente, se acuñó el término *MLOps,* que no es más que la práctica de DevOps en proyectos de aprendizaje automatizado (del inglés *machine learning*).

6.3. Análisis de las técnicas y prácticas avanzadas

Podríamos inferir que un factor común en el uso de las metodologías y prácticas antes mencionadas es la posibilidad de obtener valor a través de la entrega rápida de software que será evaluado por el usuario final, y del cual tendremos la posibilidad de aprender en cada iteración a través de los ciclos de retroalimentación que del mismo resulten.

¿Pero entregar rápido y obtener retroalimentación es sinónimo de proyectos de software exitosos? La respuesta inmediata es no, y la explicación es simple: acelerar el desarrollo sobre requerimientos imprecisos o inexistentes, o el *no* considerar los atributos arquitectónicos que los soportarán solo garantizará la entrega rápida de un producto no deseado o plagado de riesgos y defectos (o ambos).

VII. CONCLUSIONES

En este material hemos hecho gran énfasis en la importancia del modelado, la arquitectura y la comunicación efectiva en el desarrollo de sistemas de software. Como aquí hemos presentado y discutido, la arquitectura es la amalgama que une y mantiene todos los componentes de un sistema de software. Una arquitectura eficiente es aquella que integra metodologías, buenas prácticas, técnicas y herramientas con los requisitos que soportará. Además, una comunicación efectiva, basada en el modelado, es la clave del éxito para establecer dicha arquitectura. Partiendo de este último postulado, nuestro trabajo se centró en presentar técnicas y mecanismos de modelado y comunicación que aporten valor, en mayor o menor medida, al proceso de desarrollo de software, en dependencia de la etapa del ciclo de vida en el cual esté trabajándose, así como de las necesidades y expectativas del cliente final.

Hemos resaltado la importancia de la comunicación mediante modelos de los cuales emerja una arquitectura sostenible y efectiva para el proyecto en cuestión. Al introducir técnicas y prácticas como las tratadas en el precedente capítulo, podemos hacer hincapié en una arquitectura viva, que se retroalimenta en ciclos cortos que permiten su rápida validación, ya no nada más por medio de artefactos como diagramas y modelos sino también de software desplegado y en uso.

La comunicación entre los diferentes actores involucrados en el proceso de desarrollo de software, incluyendo el cliente final, debe mantenerse con una perspectiva multidimensional, dependiendo del ámbito y la audiencia con la que estemos comunicándonos. Arquitectos y diseñadores de software emplean modelos soportados por diagramas para transmitir y validar requerimientos, y la solución propuesta a estos. La experiencia de los más talentosos arquitectos e ingenieros se encuentra plasmada en patrones y modelos arquitectónicos que presentan soluciones a requerimientos no funcionales o características arquitectónicas que son comunes entre proyectos. Aunque su implementación no debe considerarse una solución ya hecha, sí define un punto de partida y sirve como referencia al momento de definir una arquitectura.

De DevOps, prácticas como integración continua, despliegue continuo y entrega continua forman parte de las herramientas con las que ingenieros de

software generan ciclos de aprendizaje y mejoras de los productos de software que desarrollan. Es importante notar que estas prácticas no son nuevas; sin embargo, el momento tecnológico en el que nos encontramos nos permite implementar dichas prácticas mediante herramientas que antes no había.

Entregar productos de software más rápido no garantiza que dicho software sea funcional para el usuario, por lo que es indispensable que antes de considerar prácticas que aceleren el desarrollo, se invierta tiempo y esfuerzo en tareas relacionadas con el correcto levantamiento de requerimientos, su especificación y validación y posterior definición de una arquitectura suficientemente documentada para que aporte valor sin que resulte onerosa, y cuyo mantenimiento sea constante; es decir, preservar la arquitectura viva durante el desarrollo de software y su posterior mantenimiento.

Solo uniendo lo abstracto de la ingeniería de software con prácticas que nos permitan acortar los ciclos de desarrollo y de retroalimentación podremos garantizar proyectos de desarrollo de software exitosos.

REFERENCIAS

Abu Bakar Kamarudin; Mohd Fazli Mohd. Sam; M.I. Qureshi. 2022. “Lean Manufacturing Design of a Two-Sided Assembly Line Balancing Problem Work Cell”, in Mohd Najib Ali Mokhtar; Zamberi Jamaludin; Mohd Sanusi Abdul Aziz; Mohd Nazmin Maslan; Jeeferie Abd Razak (eds.), *Intelligent Manufacturing and Mechatronics: Proceedings of SympoSIMM* 2021, Springer Nature.

Accelerate State of DevOps Report. 2022. https://services.google.com/fh/files/misc/2022_state_of_devops_report.pdf (fecha de la última consulta: 25 de julio de 2023).

Achinstein, P. 1987. *Los modelos teóricos.* Universidad Nacional Autónoma de México, Dirección General de Publicaciones.

Agile Alliance. 2013. What is Agile Software Development? https://www.agilealliance.org/agile101/ (fecha de la última consulta: 25 de julio de 2023).

Agile Manifesto. https://agilemanifesto.org/iso/es/manifesto.html (fecha de la última consulta: 18 de septiembre de 2023).

Alapati, S. (2018). Extreme Programming (XP) - sreenivas alapati. Medium. https://medium.com/@sreenivas/extreme-programming-xp-f0ad5066f737.

Bass, L., Clements, P. y Kazman, R. (2021). *Software Architecture in Practice* (4a. ed.). Addison-Wesley Professional.

Bennett, S. M. 2007. *Análisis y diseño orientado a objetos de sistemas*. McGraw-Hill.

Booch, G. 1998. *Object-Oriented Analysis and Design with Applications.* Addison Wesley.

Braude. J. E. 2003. *Ingeniería de software:* Una perspectiva orientada a objetos. AlfaOmega.

Brown, D. 2015. What is DevOps? https://www.donovanbrown.com/post/what-is-devops (fecha de la última consulta: 30 de junio de 2023)

Cervantes, H., Kazman, R. (2016). *Designing Software Architectures: A Practical Approach.* Pearson Education.

Coad, P., Yourdon, E. 1998. *Object-Oriented Analysis.* Second Edition. Yourdon Press Computing Series.

Cockburn, A. 2008. Using Both Incremental and Iterative Development. http://www.stsc.hill.af.mil/crosstalk/2008/05/0805Cockburn.html (fecha de la última consulta: 25 de septiembre de 2023).

Coplien, J. O. & Bjørnvig, G. (2011). *Lean Architecture: for Agile Software Development.* Wiley.

Electron. https://www.electronjs.org/es/ (fecha de la última consulta: 14 de septiembre de 2023).

Forsgren, N., Humble, J. y Kim, G. 2018. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations.* ITRevolution Press.

Fowler, M., Scott, K. 1999. *UML gota a gota*. Pearson Educación.

Fowler M. 2004. UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional.

Goldberg, A., Robson, D., Harrison, M.A. (Editor). 1983. Smalltalk-80: *The Language and its Implementation. A*ddison-Wesley.

Gómez Fuentes, M.C., Cervantes Ojeda, J., González-Pérez, P.P. 2019. *Fundamentos de Ingeniería de Software.* Universidad Autónoma Metropolitana, México.

González-Pérez P.P., Cárdenas-García M. (2018) "Inspecting the Role of PI3K/AKT Signaling Pathway in Cancer Development Using an In Silico Modeling and Simulation Approach". In: Rojas I., Ortuño F. (eds) *Bioinformatics and Biomedical Engineering*. IWBBIO 2018. Lecture Notes in Computer Science, vol 10813. Springer, Cham. https://doi.org/10.1007/978-3-319-78723-7_7.

González-Pérez P.P., Cárdenas-García M. (2019) "In Silico Modeling and Simulation Approach for Apoptosis Caspase Pathways". In: Fdez-Riverola F., Mohamad M., Rocha M., De Paz J., González P. (eds) Practical Applications of Computational Biology and Bioinformatics, 12th International Conference. PACBB2018 2018. *Advances in Intelligent Systems and Computing,* vol 803. Springer, Cham. https://doi.org/10.1007/978-3-319-98702-6_3.

González-Pérez, P.P., Gómez Fuentes, M.C., Cervantes Ojeda, J. 2023. *Desarrollo de Software a Gran Escala*. Universidad Autónoma Metropolitana, México.

Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach* (ACM Press) Addison-Wesley.

Jacobson, I., Booch, G., Rumbaugh, J. 2000. *El proceso unificado del desarrollo del software.* Pearson Addison-Wesley.

Larman, C. 2003. *UML y patrones: Una introducción al análisis y diseño orientado a objetos y al proceso unificado,* Pearson Education.

Meyer, B. 2002. *Construcción de software orientado a objetos.* Prentice Hall.

Microsoft. 2022. Model-View-ViewModel (MVVM). https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (fecha de la última consulta: 28 de julio de 2023).

Ohno, Taiichi. 1988. *Toyota Production System: Beyond Large-Scale Production.* CRC Press.

Pfleeger, S. L. 2002. *Ingeniería de software. Teoría y práctica.* Pearson Education. Pitt, C. 2012. Pro PHP MVC. Apress.

Potel, M. 1996. MVP: Model-View-Presenter. *The Taligent Programming Model for C++ and Java.* VP & CTO Taligent, Inc.

Pressman, R. S. 2010. Ingeniería del Software: Un enfoque práctico. McGraw Hill. Qian, K. 2009. *Interaction-oriented Software Architectures. Software Architecture and Design Illuminated.* Jones and Bartlett Illuminated.

Ries, Eric. 2011. *The Lean Startup. How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses.* Crown Business.

Rumbaugh, J., Blaha, M., Premerlani, W., Hedí F., Lorensen, W. 1996. *Modelado y diseño orientado a objetos. Metodología OMT.* Editorial Prentice Hall 1996.

Rumbaugh, J., Jacobson, I., Booch, G. 2004. *El Lenguaje Unificado de Modelado.Manual de Referencia.* Addison-Wesley.

Russell, J. (Editor), Cohn, R. (Editor). 2012. Model-View-Controller. Bookvika Publishing. Schach, S. R. 2005. Análisis y diseño orientado a objetos con UML y el proceso unificado. McGrawHill.

Slack Technologies. https://slack.com/intl/es-la (consultado el 14 de septiembre de 2023).

Sommerville, I. 2011. *Ingeniería del Software,* Pearson Education.

VsCode. https://code.visualstudio.com/ (consultado el 14 de septiembre de 2023). Weitzenfeld, A. 2004. *Ingeniería de software orientada a objetos con UML, Java e Internet.* Thomson.

Yah, A. 2017. *Learning MVC Architecture with PHP: to Exit Beginners, Before Entering Frameworks. Atom Yah.*

GLOSARIO

Sigla o término	Significado
Agile	Metodología de desarrollo de software orientada a mejorar la velocidad y la calidad de la entrega de valor. Agile se centra en la pregunta ¿cómo va a desarrollarse el proyecto?
AOO	Análisis Orientado a Objetos.
API	Interfaz de Programación de Aplicaciones.
BML	Ciclo corto de desarrollo: construye, mide, aprende.
CD	Entrega continua.
CI	Integración continua.
CVDS	Ciclo de Vida de Desarrollo de Software.
DAO	Objeto de Acceso a Datos (del inglés *data access object*).
DevOps	Metodología de desarrollo de software basada en la unión de personas, procesos y productos que habilitan la entrega continua de valor al usuario final.
DevSecOps	Extensión de DevOps con énfasis en traer la seguridad al inicio de la cadena de valor.
DOO	Diseño Orientado a Objetos.
Framework	Infraestructura computacional de soporte al desarrollo de software, la cual proporciona al usuario un valioso soporte en tareas relacionadas con la edición, interpretación, compilación, ejecución, incorporación de bibliotecas de código, así como una gran gama de recursos útiles en el desarrollo de software.
IaC	Infraestructura como Código.
Lean	Metodología de desarrollo de software orientada a mejorar la velocidad y la calidad de la entrega de valor. Lean responde a la pregunta *¿qué vamos a*

	desarrollar?
MLOps	Se refiere a la práctica de DevOps en proyectos de aprendizaje automatizado.
MVC	Patrón arquitectónico Modelo-Vista-Controlador.
MVVM	Patrón arquitectónico Modelo-Vista-VistaModelo.
PAC	Patrón arquitectónico Presentación-Abstracción-Control.
UML	Lenguaje Unificado de Modelado.
UP	Proceso Unificado de Desarrollo de Software.
XP	Programación Extrema (del inglés *Extreme Programming*).