ARQUITECTURA DE SOFTWARE

Conceptos y ciclo de desarrollo

ARQUITECTURA DE SOFTWARE

Conceptos y ciclo de desarrollo

Humberto Cervantes Maceda
Universidad Autónoma Metropolitana

Perla Velasco-Elizondo
Universidad Autónoma de Zacatecas

Luis Castro Careaga
Universidad Autónoma Metropolitana

Revisión técnica
Dr. René Mac Kinney Romero
Universidad Autónoma Metropolitana

CENGAGE Learning®

Australia • Brasil • Corea • España • Estados Unidos • Japón • México • Reino Unido • Singapur

Arquitectura de software. Conceptos y ciclo de desarrollo.
Humberto Cervantes Maceda, Perla Velasco-Elizondo y Luis Castro Careaga

Presidente de Cengage Learning Latinoamérica:
Fernando Valenzuela Migoya

Director editorial, de producción y de plataformas digitales para Latinoamérica:
Ricardo H. Rodríguez

Editora de adquisiciones para Latinoamérica:
Claudia C. Garay Castro

Gerente de manufactura para Latinoamérica:
Raúl D. Zendejas Espejel

Gerente editorial en español para Latinoamérica:
Pilar Hernández Santamarina

Gerente de proyectos especiales:
Luciana Rabuffetti

Coordinador de manufactura:
Rafael Pérez González

Editora:
Abril Vega Orozco

Diseño de portada:
MSDE | MANU SANTOS Design

Imagen de portada:
©Ixpert/Shutterstock

Composición tipográfica:
Karla Paola Benítez García

Datos para catalogación bibliográfica:
Cervantes Maceda, Humberto, Perla Velasco-Elizondo y Luis Castro Careaga.
Arquitectura de software. Conceptos y ciclo de desarrollo.
ISBN: 978-607-522-456-5

Visite nuestro sitio en:
http://latinoamerica.cengage.com

Impreso en México
1 2 3 4 5 6 7 18 17 16 15

CONTENIDO BREVE

CONTENIDO DETALLADO

ACERCA DE LOS AUTORES

El **Dr. Humberto Cervantes** es profesor-investigador de tiempo completo en la UAM-Iztapalapa desde 2004. En ese mismo año obtuvo un doctorado en *Ingeniería de software* por parte de la universidad Joseph Fourier en Grenoble, Francia. Además de realizar docencia e investigación dentro de la academia en temas relacionados con *Arquitectura de software*, desde 2006 realiza consultoría y tiene amplia experiencia en la implantación de métodos de arquitectura dentro de la industria. Ha recibido diversos cursos de especialización en el tema de *Arquitectura de software* en el *Software Engineering Institute* (SEI), y está certificado como ATAM Evaluator y *Software Architecture Professional* por parte del mismo. Para más detalles, visitar: www.humbertocervantes.net

La **Dra. Perla Velasco-Elizondo** es profesora-investigadora en la Universidad Autónoma de Zacatecas (UAZ). En 2008 obtuvo el grado de Doctora en Ciencias de la Computación por la Universidad de Manchester, Inglaterra. Durante 2011 fue investigadora posdoctoral en el *Institute for Software Research* de Carnegie Mellon University, Estados Unidos. En estas instituciones se especializó en los temas de *Desarrollo de software basado en composición y Arquitectura de software*. Sobre estos temas ha desarrollado actividades de docencia, investigación y consultoría. Ha tomado diversos cursos de especialización en el tema de *Arquitectura de software* en el *Software Engineering Institute* (SEI), está certificada como ATAM *Evaluator* y SOA *Architect Professional* por el SEI y como *Scrum Master* por la *Scrum Alliance*. Para más detalles, visitar: smarturl.it/pvelascoe

El **Mtro. Luis Castro Careaga** es profesor investigador de tiempo completo en la UAM Iztapalapa desde 1984. Tiene estudios de maestría en finanzas en el Instituto Autónomo de México en 1994 y en la maestría en Ciencias de la Computación del IIMAS de la UNAM en 1984. Es Ingeniero en Electrónica por la UAM en 1983. Ha realizado actividades de consultoría en sistemas de *software* para distintas organizaciones públicas y privadas desde 1984 a la fecha. Sus áreas de interés están sobre temas de *Ingeniería de software*, *Bases de datos y Arquitecturas de software*. Ha estado ligado al SEI en temas de *Personal Software Process*, *Team Software Process* y *Arquitecturas de software*. Por parte del SEI es ATAM *Evaluator* y *Software Architecture Professional*, y también ha sido PSP *Professional Developer*, PSP *Instructor*, TSP *Coach* y TSP *Mentor Coach*.

PRÓLOGO

En el contexto de la *ingeniería de software* el desarrollo de la *arquitectura de software* tiene que ver con la estructuración de un sistema para satisfacer los requerimientos de clientes y otros involucrados, en especial los requerimientos de atributos de calidad. En el momento tecnológico donde nos encontramos interactuamos con muchos sistemas de *software* que cada vez tienen necesidades más complejas en relación con atributos de calidad, como desempeño, disponibilidad, facilidad de uso, etc. Es por ello que la arquitectura es un tema fundamental.

A raíz de nuestra experiencia como profesores especializados en *ingeniería de software*, y de años de colaboración y consultoría en la industria del desarrollo de este campo, hemos sentido la necesidad de disponer de un texto introductorio en castellano relacionado con el tema de la *arquitectura de software*. Este libro es nuestra respuesta a esta necesidad. Hemos decidido elaborarlo con un énfasis importante hacia las bases teóricas, pero también tuvimos cuidado de proporcionar ejemplos prácticos que permiten relacionar la teoría con la realidad.

El capítulo 1 introduce los conceptos básicos y el ciclo de desarrollo de la *arquitectura de software*. El capítulo 2 presenta la influencia que tienen los requerimientos en dicha arquitectura. El capítulo 3 describe los métodos de diseño de las *arquitecturas de software*. El capítulo 4 muestra la importancia y la forma de documentar las *arquitecturas de software*. El capítulo 5 introduce la evaluación de las arquitecturas, por ejemplo, con estrategias de control de calidad y de toma de decisiones. El capítulo 6 presenta la influencia de la *arquitectura de software* durante la implementación del sistema. Para concluir, el capítulo 7 describe cómo considerar actividades del ciclo de desarrollo de la arquitectura en proyectos que utilicen métodos ágiles como el que abordamos de manera especial: *Scrum*.

En todos los capítulos empleamos un caso de estudio, que se incluye de manera completa como apéndice, para ejemplificar de mejor manera cada concepto o actividad del ciclo de desarrollo de arquitectura. De manera adicional proporcionamos preguntas para el análisis y referencias a otras fuentes en donde los lectores interesados podrán profundizar el conocimiento más allá de los fundamentos que presentamos en este material. El libro puede ser usado por practicantes y estudiantes de maestría y licenciatura interesados en diseño y desarrollo de sistemas. Por otro lado, al ser relativamente corto permite a los practicantes leerlo y estudiarlo sin que esto les requiera un tiempo excesivo, pues por lo habitual este es un recurso escaso.

AGRADECIMIENTOS

En conjunto agradecemos agradecemos a todas las personas que nos han ayudado en la elaboración de este libro, comenzando por nuestros revisores:

- Luis Carballo, de Bursatec.
- Eduardo Fernández, de la *Florida Atlantic University*.
- Iván González, de Quarksoft.
- Grace Lewis, del SEI.
- Eduardo Miranda, de *Carnegie Mellon*.
- Ismael Núñez, de *Ultrasist*.
- Eduardo Rodríguez, de la UAM Iztapalapa.
- Jorge Ruiz, de Quarksoft.

Agradecemos también a las personas del *Software Engineering Institute* (SEI), las cuales nos han recibido cálidamente desde hace varios años en el taller de educadores de arquitectura, y de quienes hemos recibido una amplia cantidad de conocimiento.

Agradezco a todas las personas que han contribuido de alguna u otra manera a brindarme el conocimiento y experiencia que tengo el gusto de compartir en este libro. En especial agradezco a la UAM-Iztapalapa, al SEI, al equipo de arquitectos y directores de la empresa Quarksoft, así como a mis alumnos.

HUMBERTO CERVANTES

A todas las personas que he conocido en instituciones académicas y de desarrollo de sistemas, quienes de muy diversas formas me han permitido compartir mi conocimiento, enfrentar retos nuevos y mantener vivo el deseo de nunca dejar de aprender, ¡muchas gracias!
De manera especial agradezco a la UAZ por el apoyo recibido durante la realización de este libro.

PERLA VELASCO-ELIZONDO

Muy agradecido con autoridades, colegas, estudiantes y egresados de la UAM por todo el apoyo brindado y el estímulo para seguir dando lo mejor de mí.

LUIS CASTRO

CAPÍTULO 1

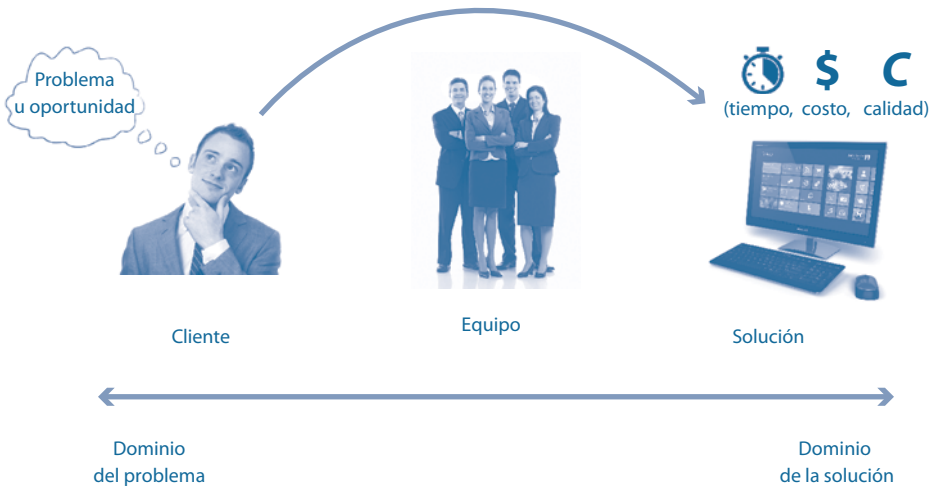
INTRODUCCIÓN: LA ARQUITECTURA Y EL DESARROLLO DE *SOFTWARE*

En la actualidad, el *software* está presente en gran cantidad de objetos que nos rodean: desde los teléfonos y otros dispositivos que llevamos con nosotros de forma casi permanente, hasta los sistemas que controlan las operaciones de organizaciones de toda índole o los que operan las sondas robóticas que exploran otros planetas. Uno de los factores clave del éxito de los sistemas es su buen diseño; de manera particular, el diseño de lo que se conoce como *arquitectura de software*.

Este concepto, al cual está dedicado el presente libro, ha cobrado una importancia cada vez mayor en la última década (Shaw y Clements, 2006). En este capítulo presentamos la introducción al tema y una visión general de la estructura de este libro.

1.1 VISIÓN GENERAL DEL DESARROLLO DE SISTEMAS DE *SOFTWARE*

Expresado de manera simplificada, el desarrollo de un sistema de *software* puede verse como una transformación hacia la solución técnica de determinada problemática u oportunidad con el fin de resolverla, como se muestra en la figura 1-1. Este cambio enfrenta a menudo restricciones en relación con el tiempo, el costo y la calidad.

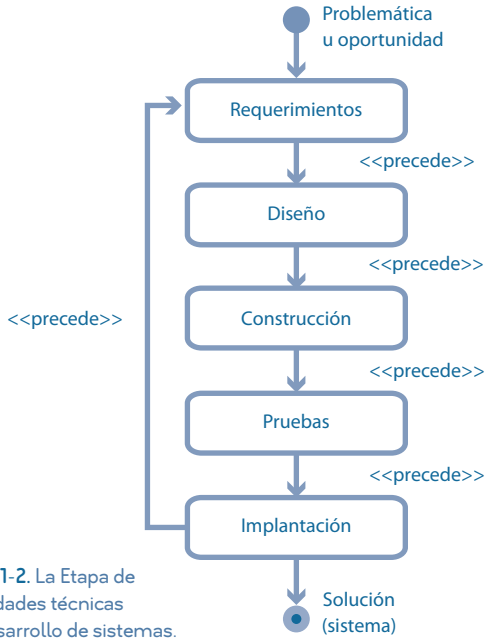


› **Figura 1-1.** Visión simplificada del desarrollo de sistemas.

Durante la transformación, que inicia en el dominio del problema y culmina en el de la solución, se llevan a cabo distintas *actividades técnicas,* las cuales se describen enseguida y se muestran en la figura 1-2.

- **Requerimientos.** Se refiere a la identificación de las necesidades de clientes y otros interesados en el sistema, y a la generación de especificaciones con un nivel de detalle suficiente acerca de lo que el sistema debe hacer.
- **Diseño.** En esta etapa se transforman los requerimientos en un diseño o modelo con el cual se construye el sistema. Hace alusión esencialmente a tomar decisiones respecto de la manera en que se resolverán los requerimientos establecidos previamente. El resultado del diseño es la identificación de las partes del sistema que satisfarán esas necesidades y facilitarán que este sea construido de forma simultánea por los individuos que conforman el equipo de desarrollo.
- **Construcción.** Se refiere a la creación del sistema mediante el desarrollo, y prueba individual de las partes que lo componen, para su posterior integración, es decir, conectar entre sí las partes relacionadas. Como parte del desarrollo de las partes, estas se deben diseñar en detalle de forma individual, pero este diseño detallado de las partes es distinto al diseño de la estructuración general del sistema completo descrito en el punto anterior.
- **Pruebas.** Actividad referida a la realización de pruebas sobre el sistema o partes de este a efecto de verificar si se satisfacen los requerimientos previamente establecidos e identificar y corregir fallas.
- **Implantación.** Llevar a cabo una transición del sistema desde el entorno de desarrollo hasta el entorno donde se ejecutará de forma definitiva y será utilizado por los usuarios finales.

Por simplificar, en estas actividades no se considera el mantenimiento, aunque también es muy importante en el desarrollo de sistemas.



› **Figura 1-2.** La Etapa de Actividades técnicas del desarrollo de sistemas.

Hay una relación de precedencia entre las actividades descritas: se precisa hacer por lo menos algo de requerimientos antes de diseñar; un tanto de diseño antes de construir; por lo menos algo de construcción antes de probar, y algunas pruebas antes de implantar. Que estas actividades se hagan por completo o de forma parcial, depende del tipo de ciclo de desarrollo que se elige, el cual va desde lo puramente secuencial (cascada) hasta lo completamente iterativo.

La *arquitectura de software* tiene que ver principalmente con la actividad de diseño del sistema; sin embargo, juega también un rol importante en relación con las demás actividades técnicas, como veremos más adelante.

1.2 DEFINICIÓN DE *ARQUITECTURA DE SOFTWARE*

Al igual que con diversos términos en *ingeniería de software*, no existe una definición universal del concepto de *arquitectura de software*.[1] Sin embargo, la definición general siguiente que propone el Instituto de Ingeniería de *Software* (SEI, por sus siglas en inglés) tiende a ser aceptada ampliamente:

> La ***arquitectura de software*** de un sistema es el conjunto de estructuras necesarias para razonar sobre el sistema. Comprende elementos de ***software***, relaciones entre ellos, y propiedades de ambos. (Bass, Clements y Kazman, 2012).

[1] Para muestra basta ver la colección de definiciones que mantiene el SEI en la página web http://www.sei.cmu.edu/architecture/start/glossary/community.cfm

De esta definición, el término "elementos de *software*" es vago, pero una manera de entenderlo es considerar de forma individual las partes del sistema que se deben desarrollar, las cuales se conocen como *módulos*. Por otro lado, las propiedades de estos elementos se refieren a las *interfaces*: los contratos que exhiben estos módulos y que permiten a otros módulos establecer dependencias o, dicho de otro modo, conectarse con ellos. El establecimiento de interfaces bien definidas entre elementos es un aspecto fundamental en la integración y la prueba exitosa de las partes de un sistema desarrolladas por separado, por ello juegan un rol esencial en la arquitectura.

Es importante señalar que el término "elementos" de la definición previa no se refiere únicamente a módulos. El diseño de un sistema requiere que se piense no solo en aspectos relacionados con el desarrollo simultáneo por un grupo de individuos, sino también con la satisfacción de requerimientos, la integración y la implantación. Para ello es necesario considerar tanto el comportamiento del sistema durante su ejecución como el mapeo de los elementos en tiempo de desarrollo y ejecución hacia elementos físicos. Por lo anterior, el término "elementos" puede hacer referencia a:

- Entidades dadas en el tiempo de ejecución, es decir, dinámicas, como objetos e hilos.
- Entidades que se presentan en el tiempo de desarrollo, es decir, lógicas, como clases y módulos.
- Entidades del mundo real, es decir, físicas, como nodos o carpetas.

Al igual que con los módulos, todos estos elementos se relacionan entre sí mediante interfaces u otras propiedades, y al hacerlo dan lugar a distintas *estructuras*. Es por ello que cuando se habla de la arquitectura de un sistema no debe pensarse en solo una estructura, sino considerarse una combinación de estas, ya sean dinámicas, lógicas o físicas.

1.3 ARQUITECTURA, ATRIBUTOS DE CALIDAD Y OBJETIVOS DE NEGOCIO

Además de ayudar a identificar módulos individuales que permitan llevar a cabo el desarrollo en paralelo de un sistema por parte de un equipo de desarrollo, la *arquitectura de software* tiene otra importancia especial: la manera en que se estructura un sistema tiene impacto directo sobre la capacidad de este para satisfacer los requerimientos, en particular aquellos que se conocen como *atributos de calidad* del sistema (de ellos hablaremos en detalle en el capítulo 2).

Ejemplos de estos atributos incluyen el desempeño, el cual tiene que ver con el tiempo de respuesta del sistema a las peticiones que se le hacen; la usabilidad (o facilidad de uso), relacionada con qué tan sencillo es para los usuarios hacer operaciones con el sistema, o bien, la modificabilidad (o facilidad de modificación), la cual tiene que ver con qué tan simple es introducir cambios en el sistema.

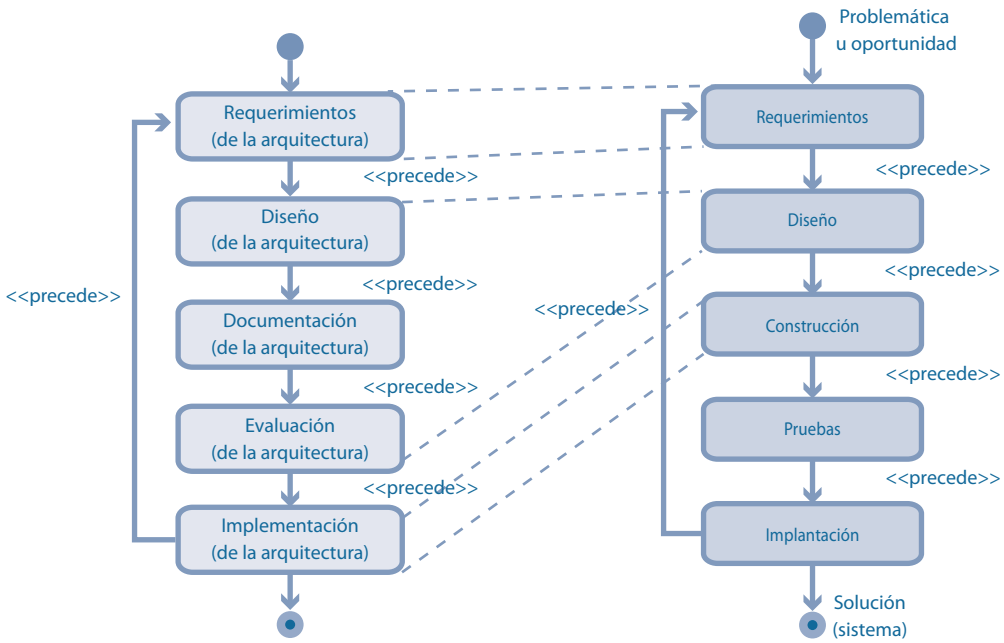
Las *decisiones de diseño* que se toman para estructurar un sistema permitirán o impedirán que se satisfagan los atributos de calidad. Por ejemplo, un sistema estructurado de manera tal que una petición deba transitar por muchos componentes implantados en nodos distintos antes de que se devuelva una respuesta podría tener un desempeño pobre.

De otro lado, un sistema estructurado a modo que los componentes sean altamente dependientes entre ellos (es decir, que estén altamente *acoplados*) limitará severamente la modificabilidad. De manera notable, la estructuración tiene un impacto mucho menor respecto a los requerimientos funcionales. Por ejemplo, un sistema difícil de modificar puede satisfacer plenamente los requerimientos funcionales que se le imponen.

Es de señalar que los atributos de calidad y otros requerimientos del sistema se derivan de lo que se conoce como *objetivos de negocio*. Estos objetivos pertenecen al dominio del problema y son las metas que busca alcanzar una compañía y que motivan el desarrollo de un sistema. Es por ello que algunos autores describen a la arquitectura como un "puente" entre los objetivos de negocio y el sistema en sí mismo. Ejemplos de estos objetivos se aprecian en el documento de visión del caso de estudio presentado en la sección 1 del apéndice del libro.

1.4 CICLO DE DESARROLLO DE LA ARQUITECTURA

De manera similar a lo expuesto en relación con las actividades técnicas para el desarrollo de sistemas, podemos hablar de un *ciclo de desarrollo* de la *arquitectura de software* que engloba actividades particulares. Estas se describen a continuación y se integran a las actividades técnicas del desarrollo de sistemas, como se muestra en la figura 1-3, independientemente de la metodología de desarrollo que se utilice, como veremos en el capítulo 7.



› **Figura 1-3.** Actividades asociadas al ciclo de desarrollo de la arquitectura (a la izquierda) y su mapeo dentro de las actividades técnicas del desarrollo de sistemas (a la derecha).

1.4.1 Requerimientos de la arquitectura

Esta etapa se enfoca en la captura, documentación y priorización de requerimientos que influyen sobre la arquitectura y que, por lo habitual, se conocen en inglés como *drivers* arquitectónicos.[2] Como se mencionó, los atributos de calidad juegan un rol preponderante respecto de los requerimientos, así que esta etapa hace énfasis en ellos. Otros requerimientos, como los *casos de uso* y *las restricciones*, son también relevantes para la arquitectura. El capítulo 2 se enfoca en esta etapa.

1.4.2 Diseño de la arquitectura

La etapa de diseño es probablemente la más compleja del ciclo de desarrollo de la arquitectura. Durante ella se definen las estructuras de las que se compone la arquitectura mediante la toma de decisiones de diseño. Esta

[2] Por desgracia no hemos encontrado en nuestro idioma una palabra adecuada que tenga pleno significado equivalente a *drivers*, por tal motivo, y aunque esta sea un tanto deficiente, la utilizaremos en el libro.

creación estructural se hace por lo habitual con base en dos clases de soluciones abstractas probadas, llamadas *patrones de diseño* y *tácticas*, al igual que en soluciones concretas como las elecciones tecnológicas, tales como los *frameworks*. El capítulo 3 describe de forma detallada esta etapa.

1.4.3 Documentación de la arquitectura

Una vez que ha sido creado el diseño de la arquitectura, es necesario darlo a conocer a otros interesados en el sistema, como desarrolladores, responsables de implantación, líderes de proyecto o el cliente mismo. La comunicación exitosa depende por lo habitual de que el diseño sea documentado de forma apropiada. A pesar de que durante el diseño se hace una documentación inicial que puede incluir bocetos de las estructuras, o bien capturas de las decisiones de diseño, la documentación formal involucra la representación sus estructuras por medio de *vistas*.

Una vista representa una estructura y contiene por lo habitual un diagrama, además de información adicional que apoya en la comprensión de este. La documentación de la arquitectura es el tema del capítulo 4.

1.4.4 Evaluación de la arquitectura

Dado que la *arquitectura de software* juega un rol crucial en el desarrollo, a efecto de identificar posibles riesgos o problemas es conveniente evaluar el diseño una vez que este ha sido documentado. La ventaja de la evaluación es que representa una actividad que puede realizarse de manera temprana (aun antes de codificar), y que el costo de corrección de los defectos identificados por medio de ella es mucho menor al costo que tendría enmendarlos después de que el sistema ha sido construido. El capítulo 5 trata en detalle este tema.

1.4.5 Implementación de la arquitectura

Una vez establecida la arquitectura, se construye el sistema. Durante esta etapa es importante evitar que ocurran desviaciones respecto del diseño definido por el arquitecto. En el capítulo 6 se habla en detalle acerca de la implementación y su relación con la arquitectura.

1.5 BENEFICIOS DE LA ARQUITECTURA

Se mencionó al principio de este capítulo que el desarrollo de sistemas enfrenta por lo general restricciones en relación con el tiempo, el costo y la calidad. Como se describe a continuación, enfatizar las actividades del ciclo de desarrollo de la *arquitectura de software* aporta diversos beneficios respecto de estas restricciones.

1.5.1 Aumentar la calidad de los sistemas

La relación entre arquitectura y calidad es directa: la arquitectura permite satisfacer los atributos de calidad de un sistema y estos son, a su vez, una de las dos dimensiones principales asociadas con la calidad de los sistemas, siendo la segunda el número de defectos. Hacer una inversión significativa en el diseño arquitectónico contribuye a reducir la cantidad de defectos, la cual, de otra forma, podría traducirse en fallas que impactan negativamente en la calidad.

Un ejemplo de falla asociada con un diseño deficiente ocurre cuando se pone un sistema en producción y se descubre que no da soporte al número de usuarios previsto en un principio.

1.5.2 Mejorar tiempos de entrega de proyectos

La *arquitectura de software* juega un rol importante para que los sistemas sean desarrollados en tiempo y forma. En principio, algunos de los elementos que se identifican dentro de las estructuras arquitectónicas ayudan directamente a llevar a cabo estimaciones más precisas del tiempo requerido para el desarrollo. Por otro lado,

una estructuración adecuada ayuda a asignar el trabajo y facilita el desarrollo en paralelo del sistema por parte de un equipo. Lo anterior optimiza el esfuerzo realizado y reduce el tiempo que toma el desarrollo del sistema.

El diseño de la arquitectura involucra con frecuencia la reutilización, ya sea de soluciones conceptuales o de componentes existentes, y esto ayuda también a reducir de manera significativa el tiempo de desarrollo. Por último, y relacionado con la calidad de manera directa, la reducción de defectos resultante de un buen diseño da como resultado una necesidad menor de volver a realizar el trabajo, lo cual contribuye a que los sistemas se entreguen en los plazos previstos.

1.5.3 Reducir costos de desarrollo

Respecto del costo de un sistema, la arquitectura también es fundamental. La reutilización es un factor importante en el momento de hacer un diseño arquitectónico porque ayuda a reducir costos. Por otra parte, es posible considerar la reutilización como un atributo de calidad del sistema y tomar decisiones de diseño al respecto con la finalidad de lograr una disminución de costos en el desarrollo de sistemas subsecuentes. Reiteramos asimismo que un buen diseño contribuye a aminorar la necesidad de volver a hacer el trabajo y facilita el mantenimiento, lo cual también conduce a bajar los gastos.

1.6 EL ROL DEL ARQUITECTO

Las actividades del ciclo de desarrollo son responsabilidad del rol del *arquitecto de software*, y esta función puede ser cubierta por uno o más individuos (en cuyo caso sería un equipo de arquitectura). Aunque de manera un tanto simplista, este arquitecto debe ser visto como un líder técnico cuya tarea principal es tomar decisiones de diseño pertinentes, a efecto de satisfacer los *drivers* arquitectónicos y demás requerimientos del sistema (debe tener idealmente un buen conocimiento del dominio del problema).

Además, el arquitecto debe comunicar sus decisiones y asegurar que durante la construcción del sistema estas sean respetadas por parte de los miembros del equipo de desarrollo. Los aspectos relacionados con requerimientos, comunicación del diseño y guía al equipo de desarrollo, requieren de una cantidad adecuada de habilidades "suaves" (es decir, no técnicas) incluyendo liderazgo, negociación, y excelente comunicación escrita y oral.

En la actualidad, el rol de arquitecto está presente en diversas compañías de desarrollo de *software*, aunque no forzosamente se tiene tanta claridad respecto de las actividades que ellos deben realizar. Aun con esto, es de destacar que en el año 2015, el sitio de *CNN Money* situaba a la labor de arquitecto de *software* como el primero de los cien mejores trabajos en Estados Unidos[3].

En la actualidad, por desgracia pocos arquitectos que laboran en la industria del desarrollo de *software* han recibido una formación teórica en el tema. Esto se debe a que no es sino hasta épocas recientes que se han establecido de manera más formal los conceptos relacionados con la *arquitectura de software* y que pocas instituciones ofrecen cursos enfocados en el tema. Con frecuencia, el desconocimiento de los principios relativos a este tema impacta de manera negativa en los proyectos de desarrollo.

[3] http://money.cnn.com/gallery/pf/2015/01/27/best-jobs-2015/

EN RESUMEN

Este capítulo comenzó dando un contexto general del desarrollo de sistemas. Enseguida definimos la *arquitectura de software* y hablamos de su importancia en relación con la satisfacción tanto de los objetivos de negocio como de los atributos de calidad. Después se hizo mención del ciclo de desarrollo de la arquitectura y, por último, del rol de arquitecto.

En el siguiente capítulo comenzaremos por describir la primera etapa del ciclo de desarrollo de la arquitectura: los requerimientos.

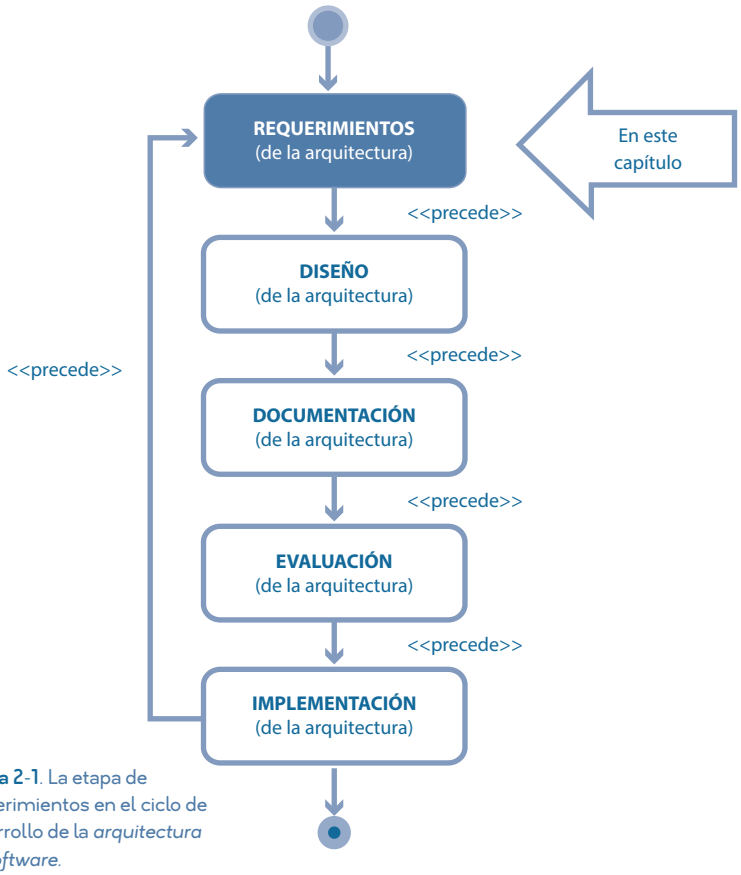
PREGUNTAS PARA ANÁLISIS

1. ¿Qué problemas podrían ocurrir en el desarrollo de *software* si no se le da importancia a la arquitectura?
2. ¿El diseño de la arquitectura cubre todo el diseño que se hace al desarrollar un sistema? En caso negativo, ¿qué otras actividades de diseño no arquitectónico se llevan a cabo en ese desarrollo?
3. Las interfaces juegan un rol esencial en la arquitectura: ¿qué pasa si no son consideradas antes de la construcción del sistema?
4. A pesar de que la definición del SEI habla de elementos de *software*, satisfacer algunos atributos de calidad requiere en ocasiones de una combinación entre *software* y *hardware*. ¿Qué ejemplo de ello daría usted?
5. ¿Puede una decisión de diseño impactar de forma positiva un atributo de calidad y, al mismo tiempo, afectar de manera negativa un atributo de calidad distinto? Dé un ejemplo.
6. ¿Por qué al momento de diseñar la arquitectura conviene hacer uso de soluciones probadas?
7. ¿Por qué es importante documentar la arquitectura?
8. ¿Por qué los atributos de calidad se llaman así?
9. ¿Tiene sentido que sea solo una persona, o unas pocas, que juegue el rol de arquitecto? ¿Por qué no realizar el diseño de la arquitectura con todo el equipo de desarrollo?
10. ¿Ha tenido experiencia con algún sistema que no haya podido ser desarrollado de forma adecuada debido a problemas relacionados con la arquitectura? En caso afirmativo, ¿cuáles fueron estos?

CAPÍTULO 2

REQUERIMIENTOS: IDENTIFICACIÓN DE *DRIVERS* ARQUITECTÓNICOS

En el capítulo anterior mencionamos que el desarrollo de la *arquitectura de software* involucra un proceso de cinco etapas. A partir de este capítulo iniciamos con las descripciones correspondientes. Como muestra la figura 2-1, comenzamos con la etapa de requerimientos. A efecto de proveer el contexto necesario, primero describiremos el término requerimientos en el ámbito general de la *ingeniería de software*, y después nos enfocaremos en los aspectos particulares de la etapa, sus conceptos fundamentales, las fuentes de información de las cuales hace uso, así como algunos de los métodos más conocidos que se utilizan para llevar a cabo de manera sistemática sus actividades.



› **Figura 2-1**. La etapa de requerimientos en el ciclo de desarrollo de la *arquitectura de software*.

2.1 REQUERIMIENTOS

En el primer capítulo se mencionó que la *arquitectura de software* tiene importancia especial porque la manera en que se estructura un sistema tiene impacto sobre la capacidad de este para alcanzar los objetivos de negocio. También se explicó que los objetivos de negocio del sistema son metas que busca alcanzar una organización y que motivan el desarrollo de un sistema.

Los objetivos de negocio del sistema pueden satisfacerse mediante comportamientos o características provistas por este. Por ejemplo, el objetivo de una organización que vende boletos de autobús, de ingresar a mercados internacionales, puede satisfacerse al permitir la compra en línea por medio de un portal y al tener facilidad para adaptar el sistema a diferentes idiomas, navegadores *web* y dispositivos móviles. Técnicamente hablando, y como se indica en la siguiente definición tomada de del libro *Software Requirements* (Wiegers, 2013), en el ámbito de la *ingeniería de software* las especificaciones de estos comportamientos y características reciben el nombre de *requerimientos*.

> Requerimiento es una especificación que describe alguna funcionalidad, atributo o factor de calidad de un sistema de *software*. Puede describir también algún aspecto que restringe la forma en que se construye ese sistema.

En este contexto es importante considerar que, además de los comportamientos y características, la definición anterior indica también que es posible que los requerimientos describan aspectos que restringen el proceso para desarrollar un sistema de *software*. Por ejemplo, hay casos en que los sistemas deben implementarse bajo consideraciones especiales sobre la fecha de inicio o terminación, límites presupuestales o el uso de determinado tipo de tecnologías. En las siguientes secciones describiremos con mayor detalle la importancia de distinguir entre diferentes clases de requerimientos y la relación que estos tienen con el concepto de *drivers arquitectónicos*.

La identificación de los requerimientos ocurre durante el análisis, actividad técnica del desarrollo de sistemas que describimos en el capítulo anterior. En el contexto de la *ingeniería de software*, la ingeniería de requerimientos es la disciplina que engloba las actividades relacionadas con la obtención, análisis, documentación y validación de estos.

2.2 REQUERIMIENTOS CON DISTINTOS TIPOS Y NIVELES DE ABSTRACCIÓN

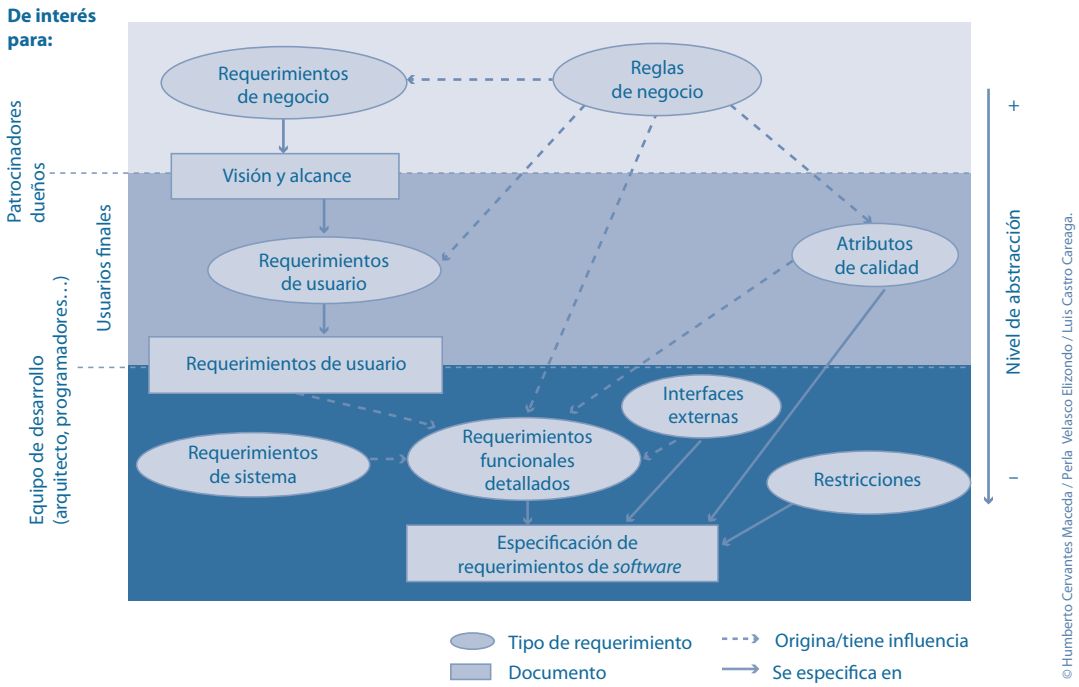
Aunque la ingeniería de requerimientos existe desde hace varios años, y a la fecha se han creado diversos métodos y herramientas que dan soporte a sus actividades, obtener y documentar requerimientos sigue siendo una tarea complicada. Recabarlos no solo consiste en escuchar al propietario o usuario final de un sistema, sino posteriormente escribir en una lista lo que él quiere que este haga. Además del propietario o usuario de un sistema, pueden existir otras personas interesadas en este; es importante reconocer correcta y oportunamente a todas ellas. Si bien los propietarios o los que hacen uso directo del sistema son más fáciles de identificar, quienes lo desarrollan, instalan y le dan mantenimiento, o bien, aquellos que patrocinaron su desarrollo (si no lo hicieron los dueños del sistema), son personas que también podrían estar interesadas en el sistema y que tal vez necesitarían tener requerimientos diferentes o adicionales a los de los propietarios o usuarios.

Debido a la heterogeneidad de los interesados en un sistema, no es extraño imaginar que la importancia que para alguno de ellos conlleva un requerimiento no sea la misma para los demás. Por ejemplo, la modificabilidad, la cual, como ya se mencionó, tiene que ver con qué tan simple es realizar cambios en un sistema, es un atributo de calidad más relevante probablemente para los desarrolladores que para los usuarios finales. De forma similar, incrementar los ingresos anuales es un requerimiento de interés para el dueño de un negocio, pero de menor importancia para esos clientes y desarrolladores. A efecto de facilitar su comprensión y manejo, es conveniente que los requerimientos sean clasificados en tipos y niveles de abstracción de acuerdo con su naturaleza.

En el contexto de la ingeniería de requerimientos se reconocen varias clases de estos, las cuales se ubican en distintos niveles de abstracción. La figura 2-2 las muestra en una versión adaptada de la figura original del modelo propuesto (Wiegers, 2013). En este modelo aparecen tres niveles. El nivel superior corresponde a tipos de requerimientos que describen aspectos del negocio. El nivel medio habla de cuestiones referidas a la interacción con los usuarios. Por último, el nivel inferior corresponde a los requerimientos que describen situaciones o elementos detallados que se necesitan para realizar el diseño del sistema. En el modelo también aparecen los documentos en dónde generalmente se especifican estos requerimientos.

La figura 2-3 muestra ejemplos de requerimientos en los niveles descritos anteriormente: a) requerimientos de negocio, que describen metas de una organización; b) requerimientos de usuario como casos de uso (Cockburn, 2001) o historias de usuario (Cohn, 2004) que describen los servicios que los usuarios pueden llevar a cabo por medio del sistema, o c) requerimientos funcionales detallados, como alguna especificación particular en el nivel de la interfaz de usuario.

En las secciones siguientes describimos en más detalle algunos tipos de requerimientos del modelo presentado antes.



› **Figura 2-2**. Tipos de requerimientos según Wiegers.

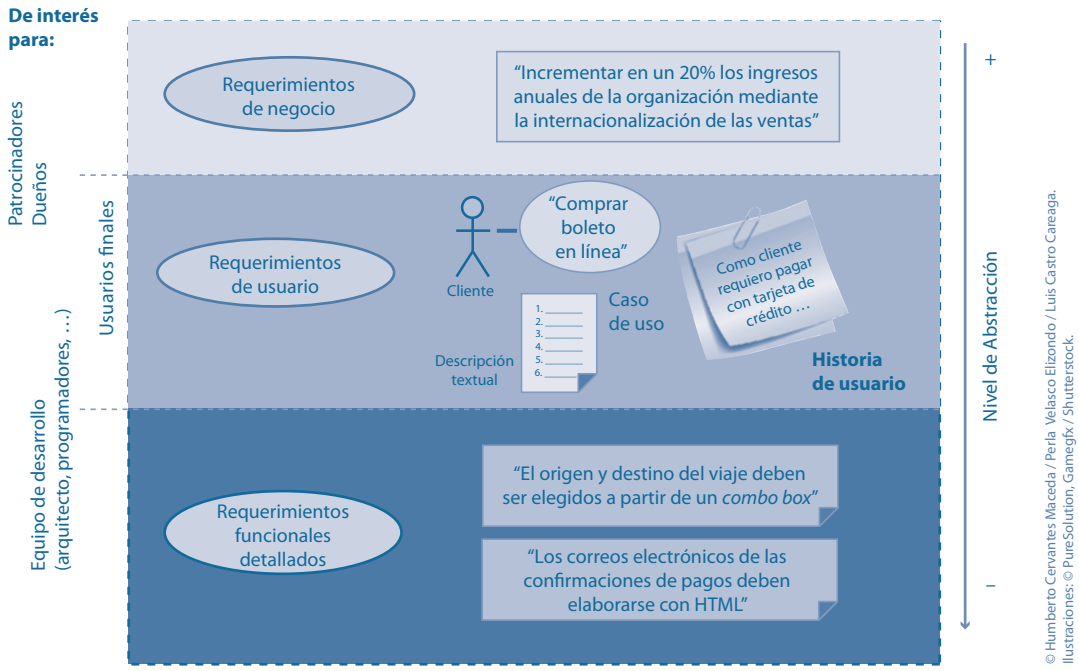
2.2.1 Requerimientos de usuario y requerimientos funcionales

Los requerimientos de usuario y los requerimientos funcionales especifican aspectos de carácter funcional sobre los servicios que pueden realizar los usuarios a través del sistema. Sin embargo, como se ilustra en la figura 2-3, se refieren a aspectos de diferente nivel de abstracción. Los requerimientos de usuario especifican servicios que por lo habitual dan soporte a procesos de negocio que los usuarios podrán llevar a cabo mediante el sistema, por ejemplo: "Comprar boleto (de autobús) en línea". Los funcionales describen detalles finos de diseño y/o implementación relacionados a los requerimientos de usuario, por ejemplo: "El origen y destino del viaje deben ser elegidos a partir de un *combo box*[1]". Ya mencionamos que los requerimientos de usuario se especifican por lo general mediante técnicas como casos de uso (en metodologías tradicionales) o historias de usuario (en metodologías ágiles).

En el conjunto de los requerimientos de usuario es posible distinguir dos tipos: primario y secundario. El primario describe servicios fundamentales que los usuarios desean llevar a cabo por medio del sistema, por ejemplo: "Comprar boleto (de autobús) en línea". Son fundamentales en tanto que soportan directamente los procesos de negocio clave de la organización. En contraste, el secundario describe acciones necesarias para dar soporte a la realización de los servicios especificados en los casos de uso o historias de usuario de los primarios. Para el ejemplo de caso de uso primario mencionado antes, casos de uso secundarios podrían ser: Acceder al sistema, Realizar alta o Enviar comprobante de compra.

Para facilitar la referencia a estos dos tipos, en lo sucesivo nos referiremos a ellos como requerimientos funcionales (que no deben confundirse con los funcionales detallados).

[1] Un *combo box* es una lista desplegable para hacer una selección en un conjunto de opciones predefinidas.



› **Figura 2-3**. Ejemplos de tipos específicos de requerimientos.

2.2.2 Atributos de calidad

Los atributos de calidad especifican características útiles para establecer criterios sobre la calidad del sistema. Actualmente no existe un "listado único" de atributos de calidad relevantes para diseñar la *arquitectura de software*. Sin embargo, estándares para la evaluación de la calidad, como el ISO/IEC 9126 (*International Standard Organization*, 2001) o modelos de calidad como los definidos por McCall's (Pressman, 2001) o Boehm (Boehm, Brown y Lipow, 1976), pueden ser de utilidad para guiar al arquitecto durante la identificación y la especificación de este tipo de *drivers arquitectónicos*. La figura 2-4 presenta los atributos de calidad que considera el estándar ISO/IEC 9126.

›**Figura 2-4**. Atributos de calidad considerados por el estándar ISO/IEC. 9126.

De manera similar, no existe un consenso universal acerca de los valores que deben tomar estos atributos, pues ello depende del contexto particular en el que operará el sistema. Sin embargo, es muy importante que los valores esperados se indiquen de manera que, durante su diseño e implementación, puedan ser evaluados.

A lo largo de este libro, hablaremos frecuentemente de las siete categorías de atributos de calidad definidos en Bass, Clements y Kazman, (2012):

1. **Disponibilidad.** Indicador acerca de si el sistema se encuentra en una condición operable cuando requiere ser utilizado.
2. **Seguridad.** Indicador del grado de protección ante usos o accesos inapropiados del sistema.
3. **Desempeño**. Indicador sobre la cantidad de trabajo realizado por el sistema considerando tiempo y recursos.
4. **Facilidad de prueba.** Indicador acerca de la facilidad con la cual se elaboran pruebas efectivas para el sistema.
5. **Modificabilidad.** Indicador referente al costo de realizar cambios en el sistema.
6. **Usabilidad**. Indicador sobre la facilidad con la cual el sistema puede ser utilizado por los usuarios.
10. **Interoperabilidad.** Indicador acerca de la facilidad del sistema para intercambiar información con otros sistemas mediante interfaces.

2.2.3 Restricciones

Como el nombre sugiere, las restricciones describen aspectos que limitan el proceso de desarrollo del sistema. Para facilitar su manejo se distinguen dos subclases:

I. Restricciones técnicas.
II. Restricciones administrativas.

Las restricciones técnicas se refieren a menudo a solicitudes expresas sobre el uso, durante el desarrollo del sistema, de productos de *software* provistos por terceros, métodos de diseño o implementación, productos de *hardware* o lenguajes de programación. Por ejemplo, una restricción técnica especificando que el *software* manejador de base de datos utilizado en un sistema debe ser gratuito (*freeware*).

Las restricciones administrativas describen aspectos que restringen el proceso de desarrollo del sistema. Estas restricciones a menudo se refieren a aspectos relacionados con el costo y tiempo de desarrollo, así como con el equipo de desarrollo. Por ejemplo, una restricción administrativa especificando que el sistema debe desarrollarse en un periodo no mayor a 12 meses.

2.2.4 Reglas de negocio e interfaces externas

Las reglas de negocio especifican políticas, estándares, prácticas o procedimientos organizacionales y/o gubernamentales que rigen o restringen la forma en que se realizan las actividades o procesos de una organización. La mayor parte de estas reglas existen por lo general fuera del contexto de un sistema. Por esta razón, como se aprecia en la figura 2-2, influyen en la especificación de otros tipos de requerimientos. Por ejemplo, durante el desarrollo de un sistema, una regla de negocio que especifica el procedimiento que usa una compañía para calcular un descuento sobre el valor de un boleto de autobús tiene influencia en el establecimiento de los requerimientos funcionales relacionados con la compra de boletos en línea.

Por otra parte, las interfaces externas especifican los detalles sobre las interfaces necesarias para que el sistema pueda comunicarse con componentes externos de *software* o *hardware*. En el contexto de la compra de boletos de autobús en línea, el diseño y la eventual implementación de la historia de usuario. "Como cliente requiero pagar con tarjeta de crédito" podría necesitar la comunicación con un elemento externo de negocios electrónicos para procesar los pagos.

2.2.5 Consideraciones importantes

Aunque modelos como el de la figura 2-2 proveen un marco conceptual para guiar la identificación y la documentación de requerimientos, es importante ser cuidadosos en su uso. Existen situaciones en las que, por ejemplo, atributos de calidad como "alta facilidad de prueba" o "alta modificabilidad" habrían podido derivarse de una restricción administrativa como: "El periodo de desarrollo del sistema no debe ser mayor a 1 año". En la figura 2-3, sin embargo, no existe una relación explícita entre estos dos tipos de requerimientos.

De forma similar, restricciones administrativas de carácter presupuestal o temporal podrían determinar el número de integrantes del equipo de desarrollo (otra restricción administrativa), la elección de productos de *hardware* en el cual se implantará el sistema, o la incorporación de componentes comerciales para dar soporte a determinados servicios de este (restricciones técnicas).

Por otra parte, algunos requerimientos son en ocasiones contradictorios entre sí. Por ejemplo, si para un sistema se necesita tanto "alta facilidad de prueba" como "alta modificabilidad", esta última podría lograrse particionando el sistema de forma más fina. Sin embargo, esto impactaría probablemente en la "alta facilidad de prueba" debido al tiempo requerido para llevar a cabo las pruebas de unidad e integración de todas estas partes. Si además existen restricciones administrativas, las cuales especifiquen que el desarrollo y la entrega del sistema se deben llevar a cabo en un tiempo corto, promover la alta modificabilidad representaría también una dificultad.

Por lo anterior, es importante negociar este tipo de contradicciones con los interesados del sistema para resolverlas o minimizarlas oportunamente. Por lo habitual, ese tipo de negociaciones se realizan considerando el impacto que tienen los requerimientos contradictorios en la satisfacción de los objetivos de negocio del sistema. Para facilitar la negociación es importante que los requerimientos tengan un indicador sobre su prioridad. La mayoría se debe priorizar aunque algunos no lo necesitan pues tienen que atenderse sin distinción, por ejemplo, las restricciones. De manera ideal, los requerimientos que tienen mayor impacto en la satisfacción de los objetivos de negocio deberían tener una prioridad más alta.

2.3 *DRIVERS ARQUITECTÓNICOS*

A pesar de que los tipos de requerimientos tienen una razón de ser, no a todos se les da la misma relevancia para el diseño de la arquitectura, aun si cuentan con una prioridad alta para el negocio. Por ello y por el hecho de que el tiempo para realizar el diseño está por lo habitual acotado, el profesional encargado debe enfocarse únicamente en los requerimientos de mayor influencia respecto de la forma que tomarán los elementos que componen las estructuras arquitectónicas. En el contexto de la creación de *software*, a este subconjunto de requerimientos se le conoce como *drivers arquitectónicos*. De esta forma, en el contexto del proceso de desarrollo de la arquitectura:

> La etapa de requerimientos se centra en la identificación, documentación y priorización de *drivers*.

Estos *drivers arquitectónicos* se clasifican en tres clases:

1. *Drivers* funcionales.
2. *Drivers* de atributos de calidad.
3. *Drivers* de restricciones.

2.3.1 *Drivers* funcionales

Los *drivers* funcionales son un subconjunto de los requerimientos funcionales que describimos en la sección 2.1.7. Esta clase de *drivers*, y particularmente los requerimientos de usuario primarios, son importantes porque proveen

información relevante para llevar a cabo la descomposición funcional del sistema y asignar estas funcionalidades a elementos específicos en la arquitectura.

Con ello, estos *drivers* se eligen considerando:

- Su relevancia en la satisfacción de los objetivos del negocio del sistema.
- La complejidad técnica que representa su implementación.
- El hecho de que representan algún escenario relevante para la arquitectura.

En la sección 2.1.1 del caso de estudio que conforma el apéndice de este libro se encuentra el modelo de casos de uso el cual incluye, entre otros, Registrarse en el sistema, Consultar corridas y Comprar boleto. Considerando los criterios anteriores, en la sección 2.1.2 se describe cómo los casos de uso primarios Consultar corridas y Comprar boleto son elegidos como *drivers* funcionales.

2.3.2 *Drivers* de atributos de calidad

Por lo habitual, todos los requerimientos de atributos de calidad son *drivers* de la arquitectura, independientemente de su prioridad. Sin embargo como el tiempo para realizar el diseño arquitectónico es acotado, para producir un diseño inicial es preferible considerar nada más un subconjunto de los requerimientos de atributos de calidad.

Al igual que con los *drivers* funcionales, los requerimientos de esta clase se eligen por lo habitual considerando estos criterios:

- Su relevancia en la satisfacción de los objetivos del negocio del sistema, es decir, su importancia para el cliente.
- La complejidad técnica que representa su implementación, esta es su importancia para el arquitecto.

Teniendo en cuenta los criterios anteriores, en la sección 2.2 del apéndice se ejemplifica y discute la elección de atributos de calidad como desempeño y disponibilidad como *drivers* del caso de estudio de este libro.

2.3.3 *Drivers* de restricciones

Como se mencionó antes, en contraste con los *drivers* funcionales y de atributos de calidad, los requerimientos de este tipo no tienen prioridad. Por esta razón, todas las restricciones son *drivers* de la arquitectura.

Como ejemplos en la sección 2.3 del apéndice, se observan algunas restricciones técnicas y administrativas asociadas al sistema del caso de estudio.

2.3.4 Otra información

Aun cuando los *drivers* descritos anteriormente especifican gran parte de la información necesaria para comenzar a diseñar un sistema, existen otros requerimientos que también podrían ser considerados por el arquitecto. Es el caso de las reglas de negocio y las interfaces externas, que podrían ser relevantes si estos determinan, de algún modo, la forma de las estructuras o los elementos arquitectónicos.

Como ejemplo retomemos el caso de uso: "Comprar boleto (de autobús) en línea". Si hay una regla de negocio, la cual indique que una vez seleccionada la corrida y los asientos, el cliente tiene hasta 24 horas para reservar y, eventualmente, completar la compra, esto podría influir en el arquitecto a usar una infraestructura de transacciones autentificadas que tienen duraciones de sesión definidas.

De modo similar, la historia de usuario: "Como cliente requiero pagar con tarjeta de crédito", podría necesitar la comunicación con un componente externo de negocios electrónicos que se encargue de los pagos. Conocer el protocolo de comunicación utilizado por el componente externo de procesamiento de pagos es importante para definir las interfaces de los componentes del sistema que se comunican con él.

2.3.5 Influencia de los *drivers arquitectónicos* en el diseño de la arquitectura

Aun cuando la información de los *drivers* influye en las decisiones de diseño tomadas por el arquitecto durante el diseño de la arquitectura, tal influencia no está igualmente repartida en todos ellos. Los de requerimientos funcionales pueden a menudo ser implementados fácilmente si se consideran en un contexto "aislado". No obstante, en un ámbito en donde también existen restricciones y expectativas sobre atributos de calidad relacionadas a ellos, implementarlos puede no ser una tarea trivial.

Por ejemplo, si se toma el caso de uso: "Comprar boleto (de autobús) en línea" como un *driver* de requerimiento funcional, en términos generales este podría ser especificado en función de la secuencia de acciones siguiente: 1) mostrar al usuario las opciones de viajes disponibles para el destino de su interés; 2) recibir del usuario la opción seleccionada; 3) formalizar la compra mediante el cobro del boleto, y 4) actualizar el número de asientos disponibles en el viaje correspondiente.

Hablando en términos de *arquitectura de software*, una alternativa válida sería asignar estas cuatro acciones a un solo componente, considerando las interfaces necesarias para la modificación de información en las bases de datos correspondientes, así como la comunicación con el componente externo que procesará el cobro del boleto. Si este requerimiento se presenta en un contexto en donde además se necesita que el procesamiento de la compra deba realizarse en un tiempo no mayor a cinco segundos, es evidente que la estrategia de diseño descrita no es la mejor.

Asignar estas cuatro acciones a un solo componente podría propiciar "cuellos de botella" en horarios de mayor uso del sistema y, por consiguiente, afectar el tiempo de respuesta. De forma similar, tal diseño propicia que el componente sea un punto de vulnerabilidad. Si este falla, el tiempo requerido para restaurarlo impedirá que la compra se complete en un tiempo no mayor a cinco segundos. Para tal escenario sería más adecuado un diseño que considere un particionamiento en donde algunas de las funciones independientes puedan asignarse a diferentes componentes, y algunos de estos, a su vez, pudieran estar replicados.

En la actualidad, muchos productos de *software* comerciales ofrecen soporte para la implementación de esquemas de replicación, de modo que el arquitecto podría considerarlos durante el diseño. Sin embargo, si en el proyecto existen restricciones presupuestales respecto del tipo de soluciones de soporte que pueden ser adquiridas, el uso de estos productos podría limitarse a los que sean gratuitos o tengan un costo bajo.

Lo anterior ejemplifica que aunque la funcionalidad es un aspecto fundamental en todos los sistemas de *software*, no es lo único que determina la forma de la arquitectura. Todo sistema tiene asociados, en mayor o menor medida, restricciones y requerimientos de atributos de calidad. Minimizar o ignorar la importancia de esta información durante el desarrollo de la arquitectura es sumamente riesgoso porque puede ser un factor importante del fracaso de un proyecto de *software*.

Es importante recordar que:

> del conjunto de *drivers arquitectónicos*, los atributos de calidad y las restricciones son los requerimientos que tienen mayor influencia sobre el diseño de la *arquitectura de software*.

2.4 FUENTES DE INFORMACIÓN PARA LA EXTRACCIÓN DE *DRIVERS ARQUITECTÓNICOS*

En esta sección presentamos en mayor detalle algunos de los artefactos que contienen información para extraer los *drivers arquitectónicos*.

2.4.1 Documento de visión y alcance

En términos generales, el documento de visión y alcance contiene información referente a por qué desarrollar un sistema, y al alcance de este, mediante descripciones textuales acerca de su contexto, las necesidades que resuelve, la descripción de sus usuarios, entre otras. El modelo presentado en la figura 2-2 considera este documento e indica que los requerimientos de negocio del sistema están especificados ahí. El documento de visión y alcance sirve de base y/o resulta complementario durante la identificación de *drivers* porque es útil, por ejemplo, al realizar actividades de priorización de estos respecto de su relevancia en la satisfacción de los objetivos de negocio del sistema.

Si bien en la práctica hay variaciones en el número de elementos de información contenidos en el documento de visión y alcance, gran parte de ellos puede ser utilizada como datos de entrada de varios métodos de la etapa de requerimientos. La figura 2-5 presenta, según su autor (Wiegers, 2013) la estructura y elementos de información sugeridos para el documento. En la sección 1 del apéndice se presenta un ejemplo de documento de visión y alcance con una versión simplificada del formato mostrado en la figura 2-5.

1. **Requerimientos de negocio**
   - 1.1 Antecedentes
   - 1.2 Oportunidad de negocio
   - 1.3 Objetivos de negocio y criterios de éxito
   - 1.4 Necesidades del cliente o del mercado
   - 1.5 Riesgos de negocio
2. **Visión de la solución**
   - 2.1 Enunciado de visión
   - 2.2 Características principales
   - 2.3 Suposiciones y dependencias
3. **Alcance y limitaciones**
   - 3.1 Alcance de la versión inicial
   - 3.2 Alcance de las versiones subsecuentes
   - 3.3 Limitaciones y exclusiones
4. **Contexto de negocio**
   - 4.1 Perfiles de los interesados en el sistema
   - 4.2 Prioridades del proyecto
   - 4.3 Ambiente de operación

› **Figura 2-5**. Ejemplo de estructura y elementos de información sugeridos para el documento de visión y alcance.

2.4.2 Documento de requerimientos de usuario

Los casos de uso y las historias de usuario son elementos utilizados para especificar necesidades de los clientes, es decir, para especificar los servicios que estos pueden realizar por medio del sistema (véase la figura 2-3). En el contexto de la ingeniería de requerimientos, en el modelo de la figura 2-2, tal información es parte del documento de requerimientos de usuario. Es válido que este no se genere de manera formal, sobre todo en proyectos de desarrollo que usan metodologías ágiles. Lo importante en esta etapa es que la información esté disponible para el arquitecto.

Los casos de uso (Cockburn, 2001) , además de tener una representación visual en forma de elipse en el diagrama de casos de uso, incluyen una descripción textual que detalla la secuencia de interacciones entre el sistema y los actores. Los elementos en la descripción textual pueden variar, aunque por lo habitual se incluye el nombre del caso de uso, actores involucrados, objetivos de negocio relacionados, precondiciones y poscondiciones, así

como descripciones de los flujos de interacciones principales, alternativas y de error. En conjunto al diagrama de casos de uso y las descripciones textuales correspondientes, se le conoce como modelo de casos de uso.

Las historias de usuario (Cohn, 2004) son especificaciones cortas escritas en una o dos frases utilizando el lenguaje del usuario final. Estas descripciones se complementan por lo habitual tanto con conversaciones que definen el detalle del servicio especificado en la historia de usuario como con descripciones textuales de las pruebas que servirán para determinar si esta fue atendida durante el diseño y la implementación. Debemos destacar que en el título de esta sección usamos la palabra funcional para hacer explícito que las historias de usuario a las que nos referimos describen requerimientos de usuario. Esto es importante porque en algunas ocasiones las historias se emplean para describir otros tipos de requerimientos, por ejemplo, de atributos de calidad.

2.4.3 Documento de especificación de requerimientos

Además del de visión y alcance y del de requerimientos de usuario, otro documento representativo de la ingeniería de requerimientos es el de Especificación de Requerimientos (SRS, por sus siglas en inglés). Como se observa en la figura 2-2, este documento contiene elementos para especificar los atributos de calidad, interfaces externas y restricciones y requerimientos funcionales que, como lo hemos mencionado, representan información relevante a efecto de identificar los *drivers*.

2.5 MÉTODOS PARA LA IDENTIFICACIÓN DE *DRIVERS ARQUITECTÓNICOS*

Hasta el momento, en este capítulo nos hemos limitado a abordar la etapa de requerimientos en términos de descripciones muy generales sobre las tareas de identificación, especificación y priorización de *drivers arquitectónicos*. Sin embargo, poco hemos dicho acerca de cómo llevar a cabo estas actividades. En la actualidad existen algunos métodos y modelos que pueden ser utilizados de una forma más sistemática por el arquitecto en su trabajo en aquella etapa. Algunos de ellos son:

1. Taller de atributos de calidad (QAW, por sus siglas en inglés).
2. Método de diseño centrado en la arquitectura (ACDM, por sus siglas en inglés).
3. Funcionalidad, usabilidad, confiabilidad, desempeño, soporte (FURPS+).

En las siguientes secciones describiremos estos métodos.

2.5.1 Taller de atributos de calidad (QAW)

El taller de atributos de calidad (Barbacci, Ellison, Lattanze, Stafford, Weinstock y Wood, 2008), o QAW[2], es un método desarrollado por el SEI que define un proceso para llevar a cabo de forma sistemática la identificación de *drivers* de atributos de calidad.

El método se basa en realización de un taller, que dura de uno a dos días e incluye diversas actividades para lograr la identificación, especificación y priorización de requerimientos de atributos de calidad. Tiene dos características distintivas: la primera es que considera la participación activa de los diversos tipos de interesados, por ejemplo los dueños, usuarios directos, dueños, patrocinadores, desarrolladores o los que le darán mantenimiento; y la segunda es que utiliza escenarios para la especificación de los *drivers* de atributos de calidad.

El método considera también la participación de un grupo de facilitadores (un líder y uno o más secretarios) con experiencia en este tipo de talleres, quienes apoyan la realización en tiempo y forma de las actividades.

Los escenarios de atributos de calidad son análogos a los casos de uso o historias de usuario funcionales respecto de que especifican requerimientos de atributos de calidad.

[2] En inglés: *Quality Attribute Workshop*.

Aunque en la práctica existen variaciones en relación con los elementos de información que debe contener un escenario de tales atributos, se recomienda que este se elabore en términos de la fuente y el estímulo que produce determinada respuesta del sistema, las partes del sistema afectadas por el estímulo, el contexto de ejecución en el cual se producirá dicha repuesta y, por sobre todo, la medida de calidad asociada a la respuesta producida. La figura 2-6 muestra un extracto de un escenario para el atributo de calidad de desempeño. El escenario corresponde al sistema de venta de boletos de autobús que hemos establecido; en específico, describe el funcionamiento esperado del sistema cuando se lleva a cabo la operación de búsqueda de corridas. En el contexto del caso de estudio de este libro, en la sección 2.2 del apéndice se encuentran, entre otros, ejemplos de escenarios de atributos de calidad, como desempeño y disponibilidad.

…	
Estímulo:	Petición de búsqueda de corridas de autobús.
Fuente del estímulo:	Un usuario del sistema.
Contexto de operación:	En un momento normal de operación con no más de 99 usuarios conectados al sistema.
Artefacto:	El sistema.
Respuesta:	El sistema procesa la petición y muestra la lista de corridas correspondientes.
Medida de respuesta:	Se procesa la petición en un tiempo no mayor a 15 segundos.
…	

› **Figura 2-6**. Extracto de un escenario para el atributo de calidad de desempeño.

Para llevar a cabo el taller de atributos de calidad es necesario que la organización para la cual se desarrollará el sistema tenga claridad sobre el contexto de operación de este, así como de sus principales funcionalidades, expectativas de calidad y restricciones.

Como se ilustra en la figura 2-7 el taller define un proceso secuencial de ocho etapas, las cuales describimos a continuación.

1. **Introducción y presentación de los participantes.** Uno de los facilitadores describe el objetivo, las fases y las actividades a realizarse en el taller y permite que los demás participantes se presenten.
2. **Presentación de la organización dueña del sistema.** Un representante de la organización para la cual se desarrollará el sistema explica respecto de este, su contexto, los objetivos de sus principales funcionalidades, las expectativas de calidad y las restricciones. Mientras transcurre esta explicación, uno de los facilitadores recaba la información que considera relevante para identificar *drivers arquitectónicos*.
3. **Presentación del boceto de la arquitectura.** Aun cuando no existe todavía una arquitectura definida para el sistema, durante el taller se solicita al arquitecto de *software* de la organización presentar un boceto o primera versión de ella, la cual él considera que satisfará los requerimientos descritos en el paso anterior. De forma similar, durante la presentación uno de los facilitadores registra la información importante para identificar *drivers arquitectónicos*.
4. **Identificación de los *drivers* arquitectónicos.** Con base en la información presentada hasta el momento, los facilitadores definen una primera versión del conjunto de *drivers arquitectónicos* del sistema poniendo énfasis especial en los que corresponden a atributos de calidad. Después los exhiben a los participantes del taller y, mediante una dinámica de lluvia de ideas, los invitan a hacer preguntas

› **Figura 2-7**. Etapas del taller de atributos de calidad.

y/o comentarios con el propósito de refinarlos. Al término de esta etapa debe haber un consenso sobre la cantidad y el tipo de *drivers* que hayan sido identificados.

5. **Generación de escenarios.** Durante esta etapa los diversos interesados en el sistema generan los escenarios "crudos" o básicos para los atributos de calidad identificados. Esta creación se lleva a cabo mediante una lluvia de ideas en la cual cada participante propone un escenario para un atributo relevante en su contexto. Un facilitador guía esta lluvia de ideas de modo que al final de ella se tenga especificado al menos un escenario para cada uno de los *drivers arquitectónicos* identificados en el paso anterior. Para la especificación de escenarios se utiliza el formato de la figura 2-6.
6. **Consolidación de escenarios.** Uno de los facilitadores pide a los participantes identificar los escenarios que son similares; esto se hace con el propósito de consolidarlos y generar un conjunto reducido, pero completo, de estos. Al término de la etapa debe haber un consenso sobre el número y tipo de los escenarios especificados.
7. **Priorización de escenarios.** Por medio de una dinámica de votación, en esta etapa se lleva a cabo la priorización de los escenarios especificados en el paso anterior. Los facilitadores otorgan a cada participante un número determinado de votos y los invitan a asignarlos a los que consideran de mayor prioridad. De manera ideal, los votos asignados a los participantes individuales corresponde a 30% del número total de escenarios. La votación ocurre en dos rondas, en cada una de las cuales ellos utilizan la mitad de sus votos.

8. **Refinamiento de escenarios.** En esta etapa se especifican con mayor detalle los escenarios de más prioridad, por lo regular cuatro o cinco. Dicho detalle incluye por ejemplo, información sobre los objetivos negocio que soporta el escenario, definiciones de los atributos de calidad o problemas identificados para dichos escenarios.

Al término de la realización de estas etapas se debe tener identificado y/o especificado: un conjunto de *drivers arquitectónicos*, una serie de escenarios correspondientes a los de atributos de calidad (algunos especificados con más detalle que otros) y una lista con los escenarios de atributos de calidad priorizados.

2.5.2 Método de diseño centrado en la arquitectura (ACDM – etapas 1 y 2)

El método de diseño centrado en la arquitectura (Lattanze, 2008), o ACDM[3] por sus siglas en inglés, fue desarrollado por Anthony Lattanze y define un proceso para apoyar la realización de diversas tareas de las etapas del ciclo de desarrollo arquitectónico (no solo de la de requerimientos). La figura 2-8 presenta la estructura y fases definidas por este método, las cuales son las ocho siguientes:

1. Identificación de *drivers arquitectónicos*.
2. Especificación del alcance del proyecto.
3. Creación o refinamiento de la arquitectura.
4. Revisión de la arquitectura.
5. Decisión de llevar o no la arquitectura a producción.
6. Experimentación.
7. Planeación de la implementación.
8. Implementación.

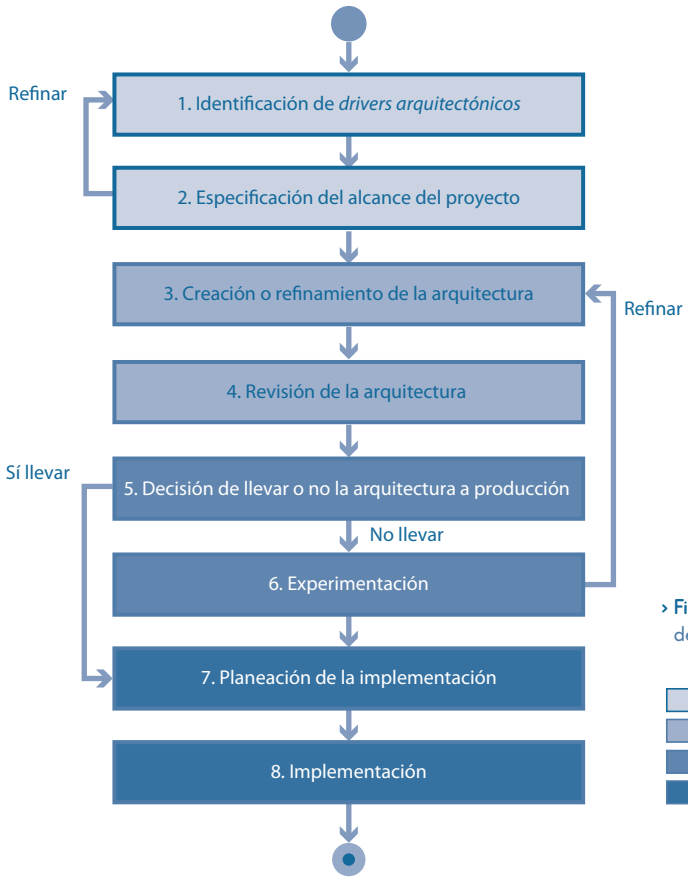
Puesto que estamos describiendo métodos que dan soporte a la etapa de requerimientos, solo abundaremos en las etapas 1 y 2. Durante ellas se llevan a cabo actividades que permiten la identificación, especificación y priorización de *drivers arquitectónicos*.

El propósito de la identificación de *drivers* es reunirse con representantes de la organización que requiere el sistema, a efecto de registrar la mayor cantidad de información relacionada con los *drivers arquitectónicos*. Sin embargo, los *drivers* obtenidos en esta etapa quedarán en forma "cruda" pues no se realiza algún análisis, refinamiento o consolidación exhaustiva de ellos.

Para dar soporte a esta etapa, el método propone definir siete roles en el equipo de desarrollo: administrador del proyecto, ingeniero de soporte técnico, arquitecto de *software* líder, ingeniero de requerimientos, experto en tecnología, ingeniero de calidad e ingeniero de producción. Respecto de los roles en la organización que requiere el sistema, no hay un conjunto definido, y la decisión sobre qué involucrados deben participar en la reunión para identificar los *drivers arquitectónicos* la toma el equipo de desarrollo. Sin embargo, se menciona que podrían tomar en conjunto esta decisión ingenieros y expertos de dominio, desarrolladores, diseñadores, integradores, capacitadores, personal de adquisiciones, usuarios finales, administradores del sistema, directivos de la organización, administradores y personal de ventas.

Las actividades para dar soporte a la identificación ocurren en el marco de un taller de *drivers arquitectónicos*, el cual tiene naturaleza similar al QAW. Dependiendo del tamaño o complejidad del sistema, durante la etapa de identificación de *drivers arquitectónicos* podría realizarse más de un taller. El método provee un proceso definido, así como una serie de técnicas, recomendaciones y formatos que pueden utilizarse para facilitar la planeación y realización del taller.

[3] *Architecture Centric Design Method.*



› **Figura 2-8**. Etapas del método de diseño centrado en la arquitectura.

El proceso necesita que algunos de los miembros del equipo de desarrollo asuman tres roles requeridos en el marco del taller: un facilitador —por lo habitual desempeñado por el ingeniero de requerimientos—, un cronometrador y uno o más secretarios.

El taller se compone por lo habitual de seis fases:

1. **Introducción y presentación de los participantes.** Un representante del equipo de desarrollo de la arquitectura, que la mayoría de las veces es el administrador del proyecto, describe el objetivo, las etapas y las actividades a realizarse en el taller y permite que se presenten los demás participantes.
2. **Descripción del sistema.** Un representante de la organización explica el contexto y los objetivos del sistema. Mientras esto transcurre, los secretarios recaban la información relevante a efecto de identificar los *drivers arquitectónicos*.
3. **Identificación de descripciones operacionales.** Un representante del equipo de desarrollo de la arquitectura lleva a cabo diversas técnicas para propiciar que los representantes de la organización hagan descripciones acerca de las formas en que ellos se imaginan que van a interactuar con el sistema. Los secretarios y el resto del equipo escriben la información recabada en formatos sugeridos por el método y con notas personales, respectivamente.
4. **Identificación de atributos de calidad.** De forma similar al paso anterior, un representante del equipo de desarrollo de la arquitectura lleva a cabo diversas técnicas para propiciar que los representantes de la organización describan los atributos de calidad del sistema. Los secretarios y el resto del equipo

anotan la información recabada en formatos sugeridos por el método y por medio de notas personales, respectivamente.

5. **Identificación de restricciones técnicas y de negocio**. De forma similar al paso anterior, un representante del equipo de desarrollo lleva a cabo diversas técnicas para propiciar que quienes representan a la organización describan las restricciones técnicas y de negocio del sistema. Los secretarios y el resto de los miembros escriben la información recabada en formatos sugeridos por el método y mediante notas personales, respectivamente.
6. **Resumen y acciones siguientes.** El facilitador presenta un resumen de la información recabada e indica a los asistentes que serán contactados posteriormente para obtener mayores detalles sobre esta.

Concluido el taller, el equipo de la *arquitectura de software* se debe reunir para consolidar la información y generar los documentos necesarios, como casos de uso, escenarios de atributos de calidad, entre otros. Además del conjunto de *drivers*, durante esta etapa se crea también una primera versión de un plan maestro de diseño, que se irá actualizando durante las etapas restantes de método.

Por otra parte, el propósito de la etapa de especificación del alcance del proyecto es analizar, consolidar y priorizar los *drivers arquitectónicos* obtenidos en la etapa de identificación de estos. Del análisis se deben generar las especificaciones finales de los *drivers*. Al término, todos ellos deben estar tanto especificados de forma clara y completa como priorizados. Esto hace que sean comprendidos por los interesados en el sistema, incluyendo al equipo de diseño de la arquitectura. Para la especificación de los *drivers* de requerimientos funcionales, el método sugiere echar mano de casos de uso; para la de los de requerimientos de atributos de calidad se utilizan escenarios. De manera similar que para la fase anterior, el método provee plantillas y guías para llevar a cabo el análisis, consolidación y priorización de los *drivers*. Si es necesario, en esta etapa se realizan también actualizaciones al plan maestro de diseño.

2.5.3 FURPS+

Más que un método, FURPS+ es un modelo creado por Hewlett-Packard, el cual define una clasificación de atributos de calidad de *software*: Funcionalidad (*Functionality*), Usabilidad (*Usability*), Confiabilidad (*Reliability*), Desempeño (*Performance*) y Soporte (*Supportability*). El modelo considera también restricciones de diseño, de implementación, físicas y de interfaz. El signo + con el que termina el acrónimo se refiere a todas estas restricciones.

Las restricciones de diseño y de implementación denotan aspectos que restringen el diseño del sistema y su codificación, respectivamente. Las de interfaz describen aspectos relevantes acerca del comportamiento de los elementos externos con los que el sistema debe interactuar. Por último, las restricciones físicas especifican propiedades que debe tener el *hardware* en el que este se instalará.

Por lo tanto, el modelo considera todos los *drivers arquitectónicos*. Es importante mencionar también que el Proceso Unificado de Rational (RUP[4], por sus siglas en inglés), un método para el análisis, diseño, implementación y documentación de sistemas orientados a objetos, toma en cuenta el uso de FURPS+ de forma explícita (Kruchten, 1995).

2.5.4 Comparación de los métodos y el modelo

Con el propósito de proveer al lector una una vista consolidada de las características de los métodos descritos en las secciones anteriores, presentamos la siguiente tabla comparativa.

[4] *Rational Unified Process.*

	Taller de atributos de calidad (QAW)	Método de diseño centrado en la arquitectura (ACDM)	FURPS+
Tipo	Taller	Taller	Modelo
Duración	Uno a dos días, dependiendo de: tamaño del proyecto, estado actual del diseño, lugar de realización del taller y número de participantes.	Uno o más talleres, con duración de uno a dos días, que se llevan a cabo de forma centralizada o distribuida, dependiendo de: tamaño del proyecto, estado actual del diseño, lugar de realización del taller y número de participantes.	Puede ser utilizado dentro del contexto de un método de desarrollo para dar soporte a la especificación y documentación de *drivers* de la arquitectura.
Mecánica y enfoque	Se llevan a cabo presentaciones y lluvia de ideas, con la participación activa de diversos interesados en el sistema.	Se llevan a cabo presentaciones y lluvia de ideas, con la participación activa de diversos interesados en el sistema.	n/a
Provee un proceso que especifica las actividades, funciones y artefactos de entrada y salida.	Sí	Sí	n/a
Participantes	Se definen dos roles principales por parte del equipo de desarrollo: líder y secretario. Se asumen otros por parte de la organización cliente, pero no se especifican.	Se definen cuatro roles principales por parte del equipo de desarrollo: líder del equipo de diseño de la arquitectura, facilitador, cronometrador y secretario, además de la participación de alguno de trece posibles por parte de la organización que requiere el sistema.	n/a
Entradas	Información sobre el contexto de operación del sistema, así como de información general sobre las principales funcionalidades, expectativas de calidad y restricciones de este.	Información sobre el contexto de operación del sistema, así como de información general sobre las principales funcionalidades, expectativas de calidad y restricciones de este.	n/a
Salidas	Un conjunto de *drivers arquitectónicos*, una serie de escenarios de atributos de calidad (algunos especificados con más detalle que otros) y una lista con los escenarios de atributos de calidad priorizados.	Todos los *drivers arquitectónicos* del sistema completamente documentados y priorizados.	n/a
Criterios de terminación	Que se hayan especificado completamente los escenarios de atributos de calidad de mayor prioridad.	Que todos los *drivers arquitectónicos* hayan sido completamente documentados.	n/a
Considera las tres clases de *drivers* principales: requerimientos funcionales, atributos de calidad y restricciones.	No. El énfasis está sobre *drivers* de atributos de calidad.	Sí	Sí

EN RESUMEN

En este capítulo describimos la primera etapa del ciclo de desarrollo de la *arquitectura de software*: los requerimientos de esta. Su objetivo es la identificación, especificación y priorización de los *drivers arquitectónicos*.

Los *drivers* son todos aquellos requerimientos que tienen mayor influencia sobre la forma que tomarán los elementos de los que se componen las estructuras de la arquitectura.

En la etapa de requerimientos se reconocen tres tipos de *drivers arquitectónicos*:

1. *Drivers* funcionales.
2. *Drivers* de atributos de calidad.
3. *Drivers* de restricciones.

Aunque es cierto que los requerimientos funcionales son un aspecto fundamental para todo sistema, en muchas ocasiones su implementación puede resultar trivial y estar basada en una arquitectura simple si no hay otro tipo de *drivers*. Sin embargo, la presencia de requerimientos de atributos de calidad y de restricciones es causa de que a menudo el arquitecto deba considerar varias opciones de diseño para satisfacer de la mejor forma los *drivers* identificados. Por esta razón, de todos los *drivers,* se dice que los de atributos de calidad y las restricciones son los que tienen mayor influencia sobre el diseño de la *arquitectura de software*.

Durante el capítulo también describimos algunos métodos y modelos de atributos de calidad que podrían ser utilizados por el arquitecto para realizar la identificación, especificación y priorización de *drivers arquitectónicos* de una forma más sistemática. Ejemplos de estos métodos y modelos incluyen el taller de atributos de calidad (QAW), el método de diseño centrado en la arquitectura (ACDM) y el modelo FURPS+.

En el siguiente capítulo trataremos la segunda etapa a realizarse durante el ciclo de desarrollo de la arquitectura: el diseño.

PREGUNTAS PARA ANÁLISIS

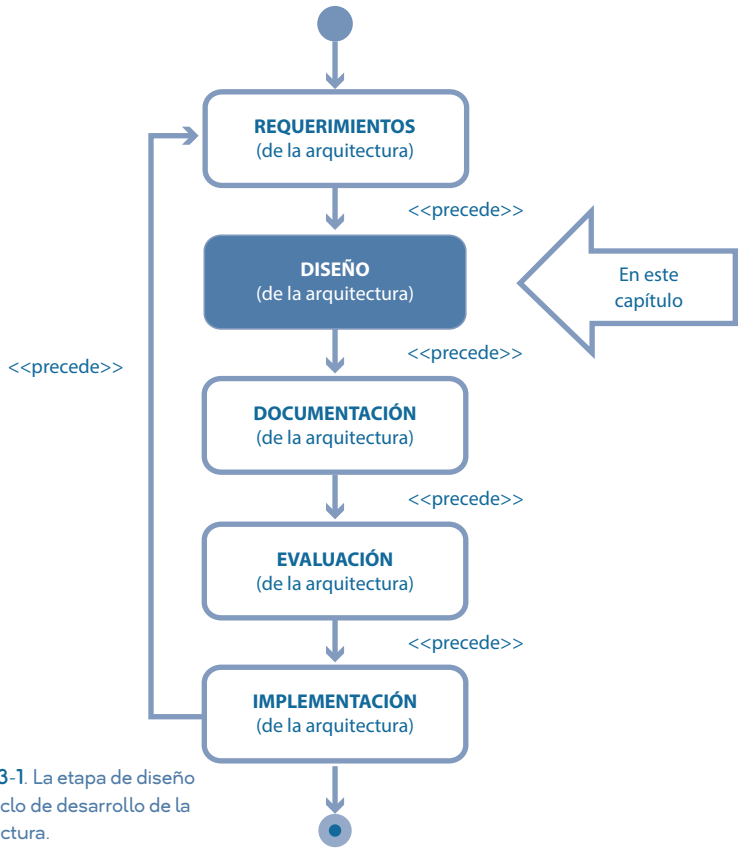
1. Considere esta frase: "Los requerimientos de usuario y los atributos de calidad son conceptos ortogonales". ¿Está de acuerdo? ¿Por qué sí o no?
2. Suponga que para determinado sistema se han identificado como los principales *drivers* los de atributos de calidad, seguridad y desempeño. Por ello han sido priorizados con una escala alta, lo cual significa que la implementación del sistema debe satisfacerlos por completo. ¿Qué implicaciones tiene esta demanda?
3. Considere la siguiente especificación de atributo de calidad: "El sistema deberá tener una interfaz amigable con pantallas ligeras". ¿A qué atributo de estos corresponde? ¿La especificación es de utilidad para el desarrollo de la arquitectura?
4. ¿Podría una restricción administrativa generar restricciones técnicas? Para contestar esta pregunta considere el caso de que haya una de tipo administrativo especificando que el sistema debe desarrollarse en un periodo no mayor a 12 meses.
5. Considere que una compañía de desarrollo de sistemas va a desarrollar un sistema de análisis automático de comentarios escritos por los clientes en las redes sociales de una cadena de restaurantes. Identifique tres objetivos de negocio del sistema asociados a este sistema.
6. Considere el sistema descrito en la pregunta anterior. Proponga dos *drivers* de requerimientos funcionales y discuta: i) su relevancia en la satisfacción de los objetivos de negocio del sistema identificados, y ii) su impacto en la descomposición funcional y asignación de responsabilidades de la aplicación.
7. Considere los atributos de calidad de disponibilidad, seguridad, desempeño e interoperabilidad para el sistema descrito en la pregunta 5. ¿Cuáles elegiría como *drivers* de atributos de calidad? Para contestar esta pregunta considere las descripciones de atributos de calidad presentadas en la sección 2.2.2 y los criterios de elección mencionados en la sección 2.3.2.
8. Considere el sistema descrito en la pregunta 5. Liste y describa las interfaces necesarias de este con componentes externos de *software* o *hardware*.
9. En el contexto de la *ingeniería de software*, la ingeniería de requerimientos es la disciplina que comprende las actividades relacionadas con obtención, análisis, documentación y validación de requerimientos. ¿El arquitecto de *software* debería participar en el proceso de tal ingeniería? ¿Por qué?
10. ¿Es posible integrar un método como QAW durante un proceso tradicional de ingeniería de requerimientos? Si la respuesta es afirmativa, discuta en qué forma se haría.

CAPÍTULO 3

DISEÑO: TOMA DE DECISIONES PARA CREAR ESTRUCTURAS

En el capítulo anterior nos enfocamos en la etapa de requerimientos y, más específicamente, en aquellos que influyen sobre la *arquitectura de software*, también conocidos como *drivers*. Este capítulo se enfoca en la etapa siguiente del ciclo de desarrollo arquitectónico, la de diseño, como se muestra en la figura 3-1.

Iniciaremos describiendo de forma general el concepto y los niveles de diseño, seguidos de la exposición de un proceso de diseño de la arquitectura. Luego nos enfocaremos en los principios y conceptos de diseño usados como parte del proceso, y más adelante describiremos la manera en que se diseñan las interfaces. Por último, el capítulo analizará algunos métodos usados actualmente para diseñar las arquitecturas.



› **Figura 3-1**. La etapa de diseño en el ciclo de desarrollo de la arquitectura.

3.1 DISEÑO Y NIVELES DE DISEÑO

A pesar de que el concepto de diseño es ubicuo, definirlo no es algo simple. Puede ser visto como una actividad que traduce una idea en un plano, o modelo, a partir del cual se puede construir algo útil, ya sea un producto, un servicio o un proceso. El aspecto importante de esta descripción es la concreción de la idea, que se realiza mediante la toma de decisiones. Un aspecto fundamental es que un diseño adecuado parte de las necesidades de un usuario. No importa qué tan ingenioso o estético sea este, si no satisface las necesidades de quien lo utiliza, entonces no es adecuado.

3.1.1 Diseño y arquitectura

Recientemente se ha propuesto una definición más formal de diseño (Ralph y Wand, 2009):

> El diseño es la especificación de un objeto, creado por algún agente, que busca alcanzar ciertos objetivos, en un entorno particular, usando un conjunto de componentes básicos, satisfaciendo una serie de requerimientos y sujetándose a determinadas restricciones.

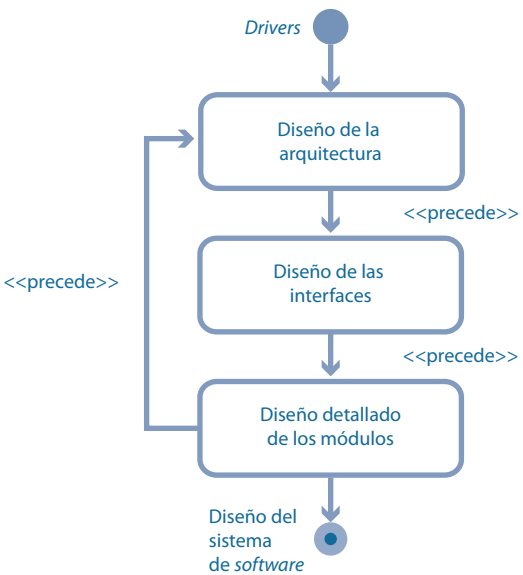
Podemos relacionar esta definición con diversos aspectos referentes a la arquitectura que hemos tratado hasta el momento:

1. El *objeto* se refiere a las distintas estructuras (físicas, lógicas, de ejecución) que componen la *arquitectura de software*.
2. El *agente* es el(los) arquitecto(s) de *software* u otros encargados del diseño.
3. Los *objetivos* son la satisfacción de los requerimientos que influyen a la arquitectura (los *drivers*) y la partición del sistema con el fin de realizar estimación o guiar su desarrollo.
4. El *entorno* se refiere tanto al contexto de uso del sistema, por parte de los usuarios finales, como al ambiente en que se desarrolla el sistema.
5. Los *componentes básicos* son los elementos de diseño, o bien, los conceptos de diseño, que discutiremos en la sección 3.4.
6. El *conjunto de requerimientos* incluye en de la lista de *drivers,* tanto los requerimientos funcionales como los no funcionales (principalmente los atributos de calidad).
7. Las *restriccione*s son todas las limitaciones impuestas ya sea por el cliente o por la organización de desarrollo; también son parte de los *drivers*.

3.1.2 Niveles de diseño

En el desarrollo de *software*, el diseño no se lleva a cabo únicamente en el nivel de la arquitectura. Podemos identificar en general tres niveles distintos de diseño:

1. Diseño de la arquitectura: este nivel se enfoca en la toma de decisiones en relación con los *drivers* de la arquitectura y la creación de estructuras para satisfacerlos. En la sección siguiente hablaremos de un proceso general de este nivel de diseño.
2. Diseño de las interfaces: este nivel ocurre parcialmente cuando se diseña la arquitectura, pero la mayor parte del trabajo ocurre una vez que este proceso ha concluido. Es en este momento en que: 1) se identifican todos los módulos y otros elementos faltantes requeridos para soportar la funcionalidad del sistema, y 2) se diseñan sus interfaces, es decir, los contratos que deben satisfacer estos módulos y otros elementos de acuerdo con los lineamientos que establece el propio diseño de la arquitectura. De este nivel hablaremos en la sección 3.5.



› **Figura 3-2**. Representación de los niveles de diseño.

3. Diseño detallado de los módulos: este nivel ocurre generalmente durante la construcción del sistema. Una vez que se han establecido los módulos y sus interfaces, se pueden diseñar los detalles de implementación de esos módulos previo a su codificación y prueba. Este nivel no será descrito en el libro, ya que los detalles de implementación de los módulos, en general, no son de naturaleza arquitectónica.

Por lo habitual estos tres niveles se llevan a cabo de forma ordenada en el tiempo, es decir, el diseño de la arquitectura precede al de las interfaces, el cual, a su vez, precede al diseño detallado de los módulos (véase la figura 3-2).

Sin embargo, es importante observar que este orden temporal no implica que se deba realizar todo el diseño de determinado nivel para luego llevar a cabo el del nivel siguiente: es posible, y muchas veces recomendable, seguir un enfoque iterativo en el cual se diseña en cada iteración una parte de la arquitectura, luego, una porción de las interfaces, y después se realiza el diseño detallado (y tal vez la construcción) de una parte de los módulos y demás elementos.

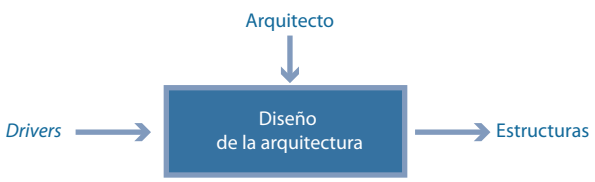
3.2 PROCESO GENERAL DE DISEÑO DE LA ARQUITECTURA

La figura 3-3 presenta el diseño de arquitectura visto como una caja negra. A la izquierda se muestran las entradas de esta actividad, que son los *drivers*, y a la derecha aparecen las salidas, la cuales son las estructuras. En la parte superior se muestra al arquitecto, quien es el principal responsable de la actividad. Por lo anterior podemos decir que en el contexto del ciclo de desarrollo de la *arquitectura de software*:

> La etapa del diseño de la arquitectura puede verse como una transformación, que realiza el arquitecto, de los *drivers* hacia las distintas estructuras que componen a la arquitectura.

El diseño de la *arquitectura de software* se realiza por lo habitual siguiendo un enfoque de "divide y vencerás". El problema general, que es realizar el diseño de toda la arquitectura, se divide en problemas de menor tamaño, que son realizar el diseño de partes de la arquitectura, y que pueden ser resueltos de manera más fácil. Lo anterior se traduce en seguir un proceso iterativo como se muestra en el lado derecho de la figura 3-4. Este procedimiento consiste en elegir en cada iteración un subconjunto de todos los *drivers* de la arquitectura y tomar decisiones de diseño al respecto, lo cual resulta en estructuras que permiten satisfacerlos.

El diseño resultante se evalúa, se elige enseguida otro subconjunto de los *drivers*, y se procede de la misma manera hasta que se completa el diseño. El proceso termina cuando se considera que se han tomado suficientes decisiones de diseño para satisfacer el conjunto de *drivers*, o bien, cuando concluye el tiempo que el arquitecto tiene asignado para realizar las actividades de diseño.

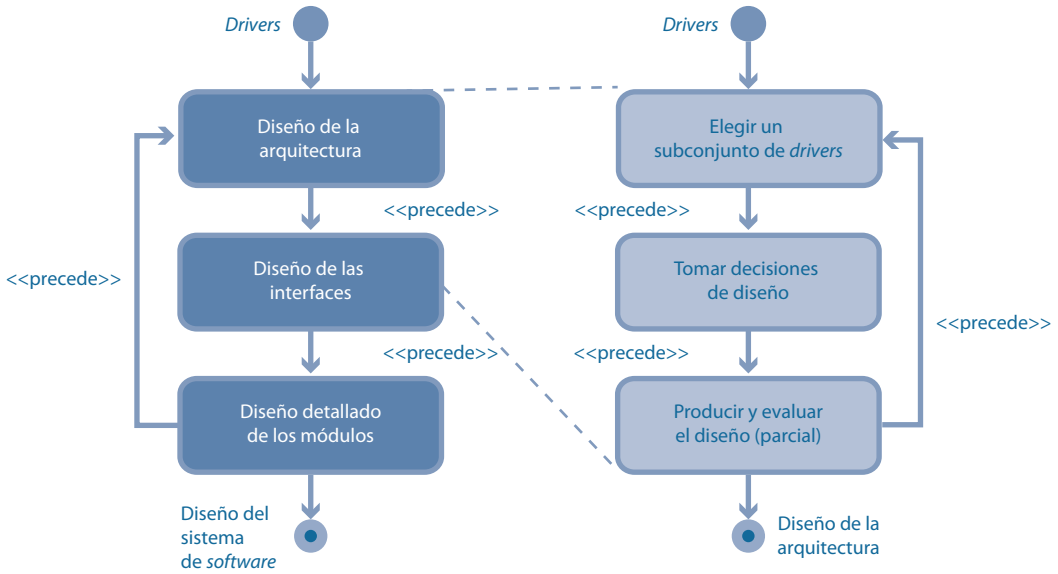


› **Figura 3-3**. Diseño como caja negra (las entradas están a la izquierda, las salidas, a la derecha, y los responsables, arriba).

En el diseño de la *arquitectura de software*, una de las ventajas de seguir el enfoque "divide y vencerás", es que resulta más simple y realista diseñar de forma iterativa e incremental, que tratar de tomar todos los *drivers* y, de una sola vez, producir un diseño que los satisfaga a todos.

Podemos ver un ejemplo de ello en el caso de estudio del apéndice (sección 3). La segunda iteración de diseño se enfoca en un subconjunto de los casos de uso, mientras que la tercera lo hace en un atributo de calidad: el desempeño.

El proceso de diseño involucra la toma de decisiones. Dado que los sistemas rara vez son completamente innovadores, muchos de los problemas de diseño con los que se enfrenta el arquitecto ya han sido atacados pre-



› **Figura 3-4**. Proceso general de diseño de la arquitectura.

viamente por otras personas. Por ello, una parte considerable de la toma de decisiones involucra la identificación, selección y adecuación de soluciones existentes con el fin de resolver los subproblemas de diseño que se deben atender como resultado de seguir el enfoque "divide y vencerás".

Utilizar soluciones existentes en vez de "reinventar la rueda" aporta diversos beneficios: ahorra tiempo y mejora la calidad debido a que por lo habitual ya han sido probadas y refinadas. En relación con ello, podemos retomar la cita de Freeman Dyson, la cual expresa que un buen ingeniero es quien hace un diseño que funciona con el menor número posible de ideas originales (Dyson, 1979). Es importante señalar que emplear soluciones existentes no es una limitante a la creatividad en el diseño, y que la originalidad reside más bien, en la identificación y combinación de aquellas.

En la sección 3.4 hablaremos de conceptos de diseño, que son justamente soluciones probadas a problemas recurrentes de diseño. El uso de estos conceptos se puede apreciar en el caso de estudio del apéndice (sección 3), dentro de cada una de las iteraciones. Por ejemplo, en la primera iteración se emplea un concepto de diseño llamado patrón arquitectónico de Capas.

Una vez que se eligen soluciones existentes y se adecúan, se producen estructuras enfocadas al problema que se está resolviendo y las cuales son parte del conjunto de estructuras lógicas, dinámicas o físicas de la arquitectura del sistema. En el caso de estudio podemos ver un ejemplo de esto en las casillas con la leyenda "Estructuras resultantes y responsabilidades de los elementos". Cabe señalar que durante el proceso de diseño, muchas veces es necesario hacer ajustes en las diversas estructuras obtenidas previamente al buscar satisfacer un *driver* nuevo.

3.3 PRINCIPIOS DE DISEÑO

Uno de los aspectos primordiales dentro del diseño de la *arquitectura de software* es facilitar la realización de cambios. Para lograrlo se aplican por lo habitual principios bien establecidos que se describen a continuación.

3.3.1 Modularidad

Un módulo es una parte del sistema que tiene una interfaz bien definida, además de que su desarrollo se asigna a un individuo o un equipo de trabajo (Parnas, 1972). La modularidad se logra al descomponer en partes el siste-

ma. La modularidad es importante en general porque permite, por una parte, dividir y hacer paralelo el desarrollo del sistema, y por la otra, facilitar tanto la realización de cambios en él como su comprensión.

El desarrollo en paralelo se logra asignando el trabajo de diseño detallado e implementación de los módulos a distintos desarrolladores una vez que las interfaces de los módulos han sido definidas. Llevar a cabo los cambios y la comprensión se facilita si los módulos tienen cohesión alta, como se describe en el punto siguiente.

Dos dificultades asociadas con la modularidad residen en identificar la granularidad correcta y en encontrar criterios para descomponer el sistema en módulos apropiados, ya que distintos particionamientos tienen consecuencias sobre la manera en que el sistema soporta los atributos de calidad. Un ejemplo de esto es que una granularidad muy fina impacta negativamente sobre la facilidad de prueba porque hay que generar un número mayor de casos de prueba, aunque a cambio esto pudiera influir de modo positivo sobre la modificabilidad.

En la segunda iteración de diseño del caso de estudio del apéndice, sección 3.2, podemos apreciar la descomposición del sistema en módulos que soportan la funcionalidad.

3.3.2 Cohesión alta y acoplamiento bajo

Cohesión alta y acoplamiento bajo son otros principios fundamentales dentro del diseño. El concepto de cohesión es una medida que hace referencia a que los módulos deben estar enfocados hacia tareas o "preocupaciones" particulares y relacionadas semánticamente.

La cohesión alta es una característica deseable del diseño, pues simplifica la realización de cambios en el código. Dicho de otro modo, los módulos no deben ser "todólogos" y mezclar código de funciones distintas.

Por ejemplo, un módulo que mezcla tanto código relacionado con la interacción con los usuarios como la lógica del programa, así como lógica enfocada a realizar persistencia, es un módulo de baja cohesión, pues implementa múltiples tareas de forma simultánea lo cual complicará su modificación.

El concepto de acoplamiento se refiere a qué tanto depende un módulo de otro. El acomplamiento bajo es una característica deseable del diseño, y al igual que la cohesión alta, facilita la realización de cambios en el código. Al tener acoplamiento bajo, las modificaciones que se le hacen a un módulo no impactan en otros que dependen del módulo cambiado. A esto se le conoce también como prevenir el "efecto de onda".

Este tipo de acoplamiento se logra mediante el principio de encapsulamiento, el cual hace referencia a que los detalles de implementación de la interfaz o el contrato del módulo deben estar ocultos a otros módulos dependientes. Por ello, para lograr un acomplamiento bajo es fundamental hacer un buen diseño de las interfaces, como se describe en la sección 3.5.

En el caso de estudio del apéndice se aprecia el principio de cohesión alta y acoplamiento bajo en la estructura resultante de la segunda iteración de diseño (sección 3.2). La cohesión alta se observa en los módulos que se especializan ya sea en aspectos de interacción con el usuario, manejo de la lógica de negocio, o bien persistencia de los datos. El acoplamiento bajo se observa, por ejemplo, en el hecho de que los módulos encargados de realizar la persistencia ocultan el detalle de que esta se realiza en una base de datos relacional.

3.3.3 Mantener simples las cosas

Un principio adicional que es fundamental en el diseño se conoce como principio KISS[1] (del inglés *Keep it simple and straightforward*, es decir, "Manténlo sencillo y directo"). Se refiere a mantener el diseño lo más simple posible con el fin de limitar la complejidad. Existen métricas que permiten cuantificar la complejidad pero, *grosso modo*, el limitarla se refiere en general a tratar de reducir al mínimo necesario el número de dependencias, encontrar una granularidad apropiada para los módulos y, además, evitar que una operación involucre en su ejecución una cantidad innecesaria de módulos.

Un aspecto adicional a considerar es que el diseño debería enfocarse en satisfacer los atributos de calidad requeridos para el sistema y no más de estos. Algunas veces se diseñan soluciones extremadamente elabora-

[1] http://www.princeton.edu/~achaney/tmve/wiki100k/docs/KISS_principle.html

das con el fin de que en un futuro el sistema se extienda de maneras no contempladas al momento del diseño. No obstante, si la modificabilidad no es un atributo de calidad deseado para el sistema, ese esfuerzo es innecesario y, además, puede resultar en la introducción de complejidad adicional.

3.4 CONCEPTOS DE DISEÑO

Como se describió previamente, al momento de diseñar es conveniente buscar soluciones probadas a los problemas recurrentes de diseño. Existen varias categorías de estas soluciones, las cuales se describen a continuación.

3.4.1 Patrones

Uno de los conceptos fundamentales de diseño son los patrones, que son soluciones conceptuales a problemas recurrentes a la hora de diseñar. Tiene como origen la arquitectura civil y, más específicamente, la obra del arquitecto Christopher Alexander y sus colegas, quienes escribieron el libro *Un lenguaje de patrones,* el cual describe una serie de soluciones a problemas de diseño para niveles distintos: geográfico, urbanístico, de barrio e incluso de detalles de construcción (Alexander, Ishikawa y Silverstein, 1977). Ese texto es un catálogo de patrones cuya particularidad es que cada uno de ellos tiene un nombre específico, como por ejemplo, "Comunidad de 7000", "Gradiente de intimidad", o bien, "Reino de niños".

El hecho de que los patrones tengan un nombre resulta fundamental pues la idea es que con tan solo hacer referencia a, por ejemplo, "Gradiente de intimidad", un arquitecto civil pueda comunicar a sus colegas la solución que está usando sin tener que explicarla en detalle. El conjunto de nombres de los patrones de diseño forma un vocabulario o lenguaje de esta actividad, y es por ello que el libro de Alexander tiene ese título.

En el desarrollo de *software*, la idea de Alexander ha sido ampliamente aceptada (de hecho, parece que mucho más que en la arquitectura civil). En 1994 apareció el primer catálogo de patrones de diseño que, hasta la fecha, sigue siendo una obra de referencia, y que se llama *Patrones de diseño: elementos de software orientado a objetos reutilizables* (Gamma, Helm, Johnson y Vlissides, 1994). Este libro, cuyos autores son conocidos como el "GoF", o "*Gang of Four*" (banda de los cuatro) documenta 23 patrones enfocados a problemas de diseño de granularidad relativamente fina.

Al igual que Alexander y sus colegas, el GoF creó un lenguaje de diseño al asignar a sus patrones nombres como Fábrica (*Factory*), Intermediario (*Proxy*) u Observador (*Observer*). El catálogo sigue una estructura muy similar a la del libro de Alexander y sus colegas, y la descripción de los patrones incluye además del nombre, el contexto del problema, la solución conceptual, las implicaciones del empleo de la solución, y algunos ejemplos de implementación.

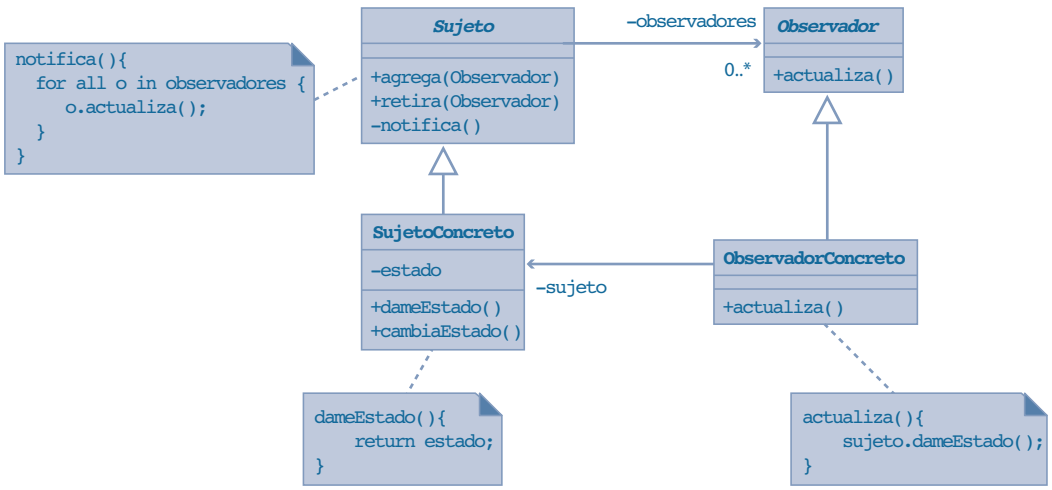
Desde la aparición del libro de GoF han surgido muchos catálogos con patrones de diseño de *software* enfocados en distintas áreas. Existen, por ejemplo, catálogos de patrones dirigidos a la estructuración general del sistema, también llamados *estilos arquitectónicos* o *arquitecturas de referencia* (Microsoft, 2009), de integración de aplicaciones (Hohpe y Woolf, 2003), arquitectónicos (Buschmann, Henney y Schmidt, 2007), para desarrollo de aplicaciones empresariales (Fowler, 2002), para desarrollo de aplicaciones basadas en servicios (Erl, 2009) o en la nube (Homer, Sharp, Brader, Narumoto y Swanson, 2014), por citar unos pocos.

Un aspecto importante a resaltar es que los patrones son soluciones conceptuales. Esto significa que no pueden usarse de forma directa, sino que deben ser "instanciados", es decir, adecuados al contexto y al problema específico que se busca resolver.

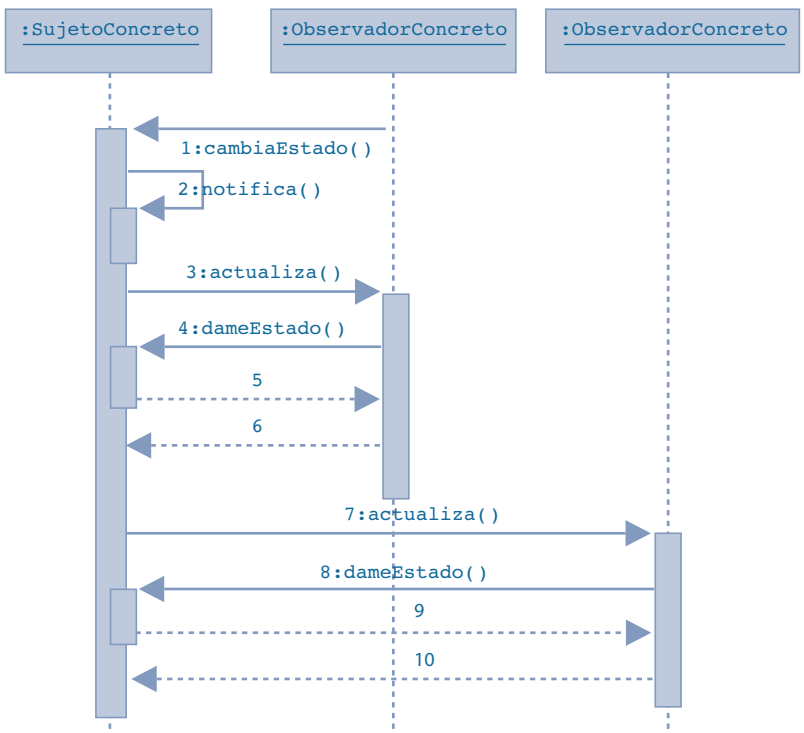
La figura 3-5 muestra un ejemplo de patrón de diseño llamado Observador (*Observer*), proveniente del libro de GoF, cuyo propósito es "definir una dependencia entre objetos de uno a muchos, de modo que cuando el estado del objeto con dependientes múltiples cambie, todos ellos sean notificados y actualizados de forma automática". Las figuras a) y b) muestran la solución desde un punto de vista conceptual por medio del modelo estructural y el de comportamiento.

La manera en que funciona Observador es que los objetos observadores se registran ante un objeto sujeto del cual están interesados en ser avisados de sus cambios de estado [método `agrega(Observador)`]. Cuando ocurre la modificación del estado, se invoca el [método `notifica()`] del sujeto que, a su vez, envía un mensaje a todos los observadores informando de ella [método `actualiza()`]. Al recibir la notificación, los observadores solicitan el estado actualizado del sujeto [método `dameEstado()`].

a) Modelo estructural:



b) Modelo de comportamiento:



› **Figura 3-5**. Modelos estructural (estático) y de comportamiento (dinámico) para el patrón Observador (llave: UML).

Durante el proceso de diseño tiende a cambiar el tipo de patrones utilizados. Al diseñar desde cero un sistema, la clase de patrón con el que inicia son los estilos arquitectónicos o también las arquitecturas de referencia. Conforme avanza el proceso, se utilizan patrones de arquitectura y de diseño.

Un estilo arquitectónico expresa un esquema de organización fundamental para los sistemas de *software*. Establece un conjunto predeterminado de tipos de elementos, especifíca sus responsabilidades e incluye reglas y guías para organizar las relaciones entre estos (Rozanski y Woods, 2005). Al establecer la estructuración inicial de un sistema de *software* se combinan por lo general varios estilos arquitectónicos. Algunos ejemplos de ellos son:

- Filtros y tuberías.
- Capas.
- Cliente/servidor.
- *N*-Tercios.
- Par a par.
- Publicador-suscriptor.

Las arquitecturas de referencia son diseños predefinidos que son usados por lo general para desarrollar un tipo particular de aplicación. En Microsoft (2009) se encuentra un catálogo diverso de estas para aplicaciones (como *Web*, Cliente Rico o Móviles). Cualquiera de estas incluye por lo habitual diversos estilos arquitectónicos y patrones de diseño. En general, el concepto de arquitectura de referencia ha sido adoptado de forma más amplia por los practicantes que el de estilo arquitectónico.

En el caso de estudio del apéndice, sección 3.1, se aprecia el uso de estilos arquitectónicos y patrones, como son Capas y *N*-Tercios en la primera iteración, *Data Mapper* (Mapeador de datos) en la segunda, y *Lazy Acquisition* (Adquisición tardía) en la tercera.

3.4.2 Tácticas

Las tácticas son conceptos de diseño que influyen sobre el control de la respuesta a un atributo de calidad particular. A diferencia de los patrones, no presentan soluciones conceptuales detalladas, sino que son técnicas probadas de las ciencias de la computación con las que se resuelven problemas en aspectos particulares relacionados con diversos atributos de calidad. La figura 3-6a muestra la estructura general de las tácticas: al recibir el sistema un estímulo, el uso de ellas permite que este responda de manera medible en relación con determinado atributo de calidad.

En el momento de escribir este libro hay un solo catálogo de tácticas, el cual está en el libro *Software Architecture in Practice* (Bass, Clements y Kazman, 2012). Ahí, este catálogo se muestra de forma gráfica como una serie de árboles en cuya raíz se encuentra una categoría particular de atributo de calidad. El catálogo describe tácticas para siete categorías de atributos de calidad: desempeño, disponibilidad, seguridad, modificabilidad, usabilidad, facilidad de pruebas e interoperabilidad. Debajo de cada categoría se encuentran estrategias y, finalmente, en en las hojas del árbol hay tácticas específicas.

La figura 3-6b muestra el árbol de tácticas asociado con la categoría de atributo de calidad de desempeño. Este catálogo debe entenderse de la forma siguiente: el desempeño está relacionado con la recepción de eventos (entrada a la izquierda) y con la generación de una respuesta a estos en un tiempo acotado por determinadas restricciones (salida a la derecha). Una estrategia para mejorar el desempeño es la administración de la demanda de recursos, y una táctica específica para lograrlo es el aumento de la eficiencia de cálculo.

Un ejemplo más concreto es que si se tiene un sistema el cual recibe eventos que deben ser procesados, y este procedimiento involucra un algoritmo, hacer más eficiente a este ayuda a que el sistema tenga el desempeño deseado. En el apéndice (sección 3.3) se observa el uso de tácticas en la tercera iteración de diseño.

Las tácticas y los patrones son complementarios. Por ejemplo, para lograr mejor desempeño si se tiene gran cantidad de observadores registrados que deben ser notificados, es posible combinar el patrón Observador descrito anteriormente con la táctica "Introducir concurrencia". Para lograrlo, esta debe aplicarse en el momento en que el sujeto invoca el método `actualiza()`, de manera que distintos observadores puedan ser notificados de forma concurrente. Así no habrá que esperar hasta que un observador termine de procesar la notificación de actualización para que el siguiente comience su trabajo, lo cual ayuda a aumentar el desempeño de la solución.

› **Figura 3-6**. Ejemplo de catálogo de tácticas para el atributo de calidad de desempeño.

3.4.3 *Frameworks*

Tanto patrones como tácticas son conceptos de diseño abstractos que deben adecuarse al problema particular y ser implementados posteriormente en el código. Existen, sin embargo, otros conceptos de diseño, los cuales no son abstractos, sino que son código concreto: los *frameworks* (marcos de trabajo). Estos son elementos reutilizables de *software* que proveen funcionalidad genérica y se enfocan en la resolución de un problema específico, como puede ser la construcción de interfaces de usuario o la persistencia de objetos en una base de datos relacional. Al igual que con la palabra *drivers*, en este libro usaremos el término en inglés *frameworks,* ya que se usa comúnmente en la práctica.

En la actualidad existen muchos *frameworks* (véase figura 3-7) enfocados a resolver una gran diversidad de problemas recurrentes, como la creación de interfaces de usuario tanto locales como *web*, la comunicación remota, la seguridad, la persistencia, etcétera. A pesar de que los *frameworks* son elementos de código, los consideramos conceptos de diseño pues su elección es en sí una decisión la cual muchas veces influye sobre la satisfacción de los *drivers*.

Frameworks, patrones y tácticas están relacionados pues, en general, los primeros implementan una variedad de patrones y de tácticas. Un ejemplo de ello son los *frameworks* enfocados en la creación de interfaces de usuario que, frecuentemente, implementan el patrón llamado Modelo-Vista-Controlador.

A pesar de que los *frameworks* ofrecen soluciones que pueden ser utilizadas de forma más directa que los patrones y las tácticas, tienen algunos inconvenientes, por ejemplo, una pronunciada curva de aprendizaje, la dificultad de elección dada la gran cantidad de ellos, su evolución constante y obsolecencia rápida, el hecho de que se requiere incluir todo un *framework* aun si solo se usa una parte pequeña de sus funcionalidades, y el soporte pobre cuando son *frameworks* poco populares de fuente libre.

No obstante estos inconvenientes, hoy en día es difícil imaginar un diseño de un sistema para dominios aplicativos tales como las aplicaciones *web*, que no involucre una variedad amplia de *frameworks*. En el caso de estudio (sección 3.1) se puede apreciar el uso de estos en la primera iteración de diseño.

› **Figura 3-7**. Algunos nombres de *frameworks* populares.

3.4.4 Otros conceptos de diseño

Existen otros conceptos de diseño que pueden ser usados durante el proceso de diseño una arquitectura; algunos de ellos son:

- **Modelos de dominio.** Modelo de los conceptos asociados a un problema específico, como puede ser el dominio hospitalario. Describe las abstracciones con sus atributos y las relaciones entre ellas. Un modelo de dominio es por lo habitual independiente de la solución y, de hecho, un mismo modelo de dominio podría usarse para crear distintas soluciones. Como parte del desarrollo es necesario en general crear un modelo de dominio para el problema en cuestión, pero también se pueden reutilizar modelos existentes (Evans, 2003).
- **Componentes COTS (*Commercial Off-the-Shelf*).** Partes de aplicaciones, o bien, aplicaciones completas listas para ser integradas. Ejemplos de componentes COTS incluyen *middleware* (es decir, *software* que se ubica entre el sistema operativo y las aplicaciones) tales como canales de integración de servicios (*Enterprise Service Bus*). Este tipo de componentes se incorpora por lo habitual de forma binaria previa configuración mediante algún mecanismo previsto para tal propósito.
- **Servicios *web*.** Son, de forma simplificada, interfaces que encapsulan sistemas completos, los cuales pueden ser parte de la compañía para la cual se desarrolla el sistema, o bien, pertenecer una empresa distinta. Al incorporar servicios *web* en del diseño, se reutilizan partes de negocio enteras de otras organizaciones, como podría ser el servicio de compra de boletos de una línea aérea.

3.5 DISEÑO DE LAS INTERFACES

Una de las clases de estructura que se generan en el momento de realizar el proceso de diseño descrito en la sección 3.2 son las dinámicas, es decir, las compuestas por entidades que existen en tiempo de ejecución. Estas estructuras pueden ser representadas mediante diagramas de secuencia u otros similares que ilustran la manera en que colabora en tiempo de ejecución un conjunto de elementos para soportar un *driver* particular, ya sea un caso de uso o un escenario de atributo de calidad.

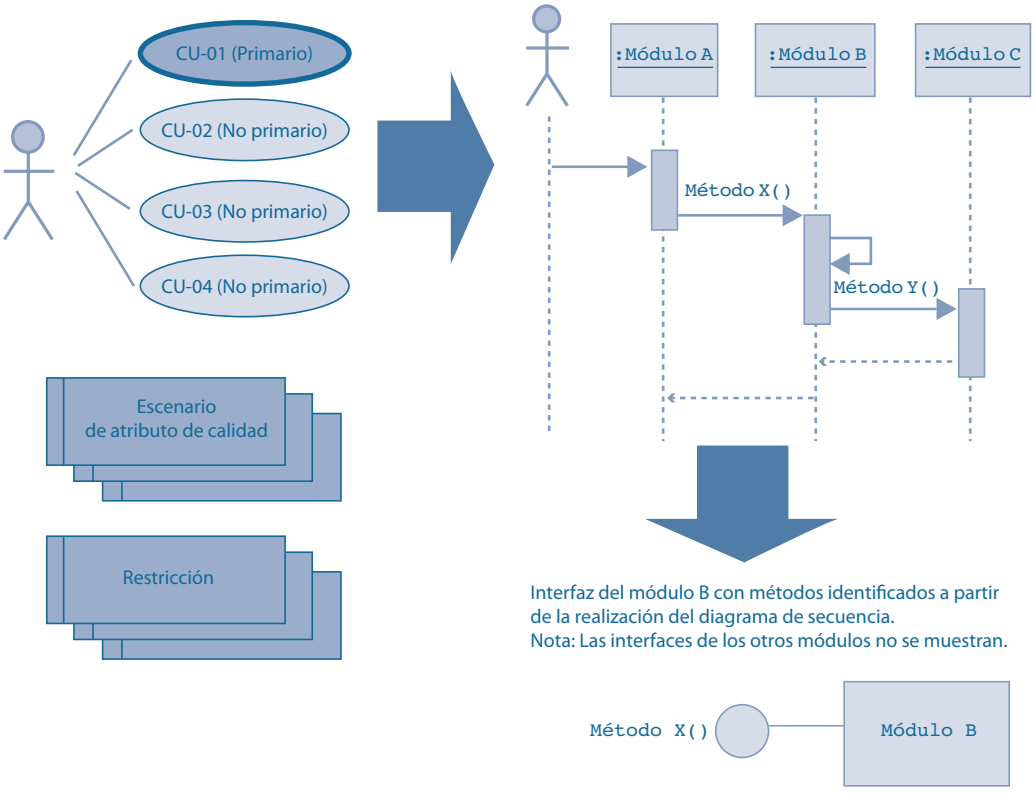
Al momento de generarse estas estructuras dinámicas se identifican los mensajes que intercambian los elementos que colaboran en la interacción. El conjunto de mensajes que recibe un elemento conforma la interfaz de este, es decir, el contrato al que debe apegarse a efecto de participar en la interacción. Lo anterior puede observarse en la sección 3.2 del apéndice (Interfaces de los elementos).

Es importante señalar que, en general, durante el diseño de la arquitectura se identifican por lo habitual solo interfaces para los elementos que soportan los *drivers*, como se puede apreciar en la figura 3-8a. Los *drivers*, sin embargo, representan únicamente un subconjunto de los requerimientos del sistema, por lo cual es necesario identificar interfaces y módulos para satisfacer en su totalidad el conjunto de requerimientos del sistema. Lo anterior es especialmente relevante en el contexto de los casos de uso del sistema porque durante el diseño de la arquitectura solo se tratan por lo general los casos de uso primarios.

a) Interfaces que se identifican durante el diseño arquitectónico:

Casos de uso, escenarios de atributos de calidad y restricciones. Los *drivers* se muestran resaltados y son usados para realizar el diseño arquitectónico.

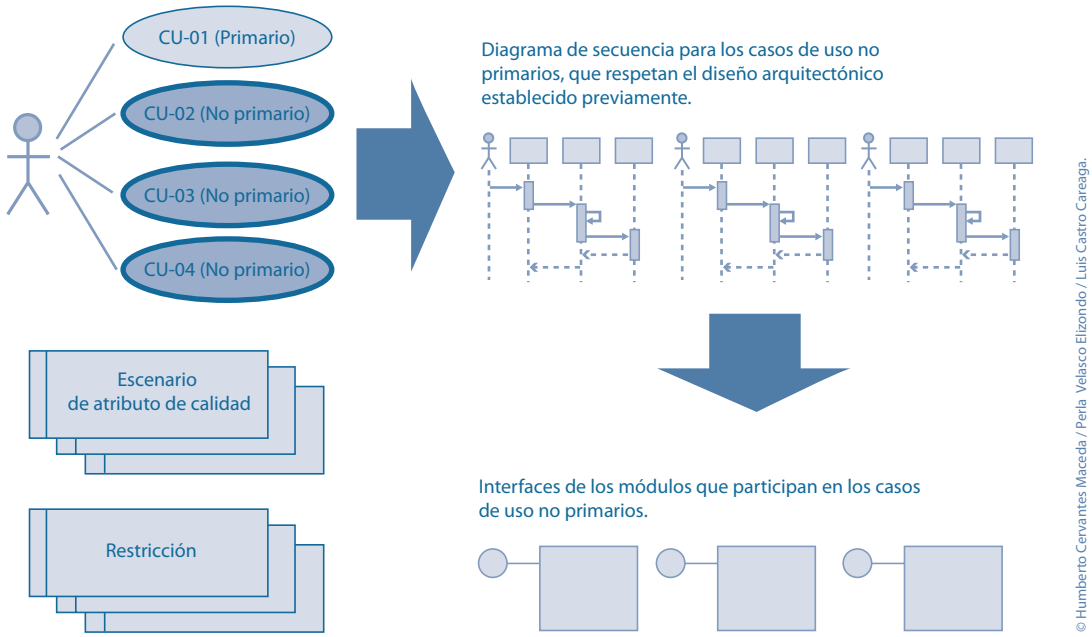
Diagrama de secuencia para CU-01 (primario), de acuerdo a las estructuras físicas y lógicas (no mostradas).



Interfaz del módulo B con métodos identificados a partir de la realización del diagrama de secuencia.
Nota: Las interfaces de los otros módulos no se muestran.

b) Interfaces que se identifican posteriormente al diseño arquitectónico:

Una vez que el diseño arquitectónico ha sido establecido, se consideran los requerimientos faltantes que son principalmente los casos de uso no primario (resaltados).



› **Figura 3-8**. Interfaces que se identifican durante el diseño arquitectónico y posteriormente al mismo.

Por lo anterior, una vez que la arquitectura ha sido diseñada es necesario realizar un ejercicio similar al que se hace durante el diseño de la arquitectura en relación con la generación de estructuras dinámicas. Esto tiene como finalidad identificar tanto los módulos como otros elementos que soportarán los casos de uso no primarios, así como sus interfaces (véase la figura 3-8b).

La diferencia aquí es que tal ejercicio se realiza apegándose al diseño de la arquitectura y no debería introducir cambios importantes en este. Si, por ejemplo, se estableció diseñar el sistema de tres capas, los casos de uso no primarios deben incluir elementos que se ubiquen en estas capas y tomar como ejemplo las estructuras diseñadas para soportar los casos de uso primarios. Muchas veces este trabajo no es realizado por el arquitecto sino por otros miembros del equipo, pero requiere que él comunique les su diseño antes de iniciar.

Diseñar las interfaces es una actividad clave en el desarrollo del sistema. Sin embargo, además de los parámetros y valores de retorno se deben considerar algunos otros aspectos, como el manejo de excepciones. Es necesario cuidar también que las interfaces no expongan detalles de implementación a efecto de lograr un acoplamiento bajo. Una vez establecidas las interfaces, servirán como entrada para el diseño detallado de los módulos y otros elementos (tema que no se expone en este libro).

3.6 MÉTODOS DE DISEÑO DE ARQUITECTURA

Es importante llevar a cabo el proceso de diseño de la arquitectura de manera sistemática y para ello existen diversos métodos:

1. Diseño guiado por atributos (ADD, por sus siglas en inglés).
2. Método de diseño centrado en la arquitectura (ACDM, por sus siglas en inglés). Ver sección 2.3.2.
3. Método de definición de arquitectura de *Viewpoints* y *Perspectives*.

En las siguientes secciones describimos estos métodos.

3.6.1 Diseño guiado por atributos (ADD)

El método diseño guiado por atributos (*Attribute Driven Design*, o ADD) es una propuesta del SEI para realizar el diseño de las arquitecturas (Wojcik *et al*., 2006). Conlleva un enfoque de "divide y vencerás", pues la arquitectura se diseña de manera iterativa y en cada iteración se toma una parte o elemento y se descompone en subelementos.

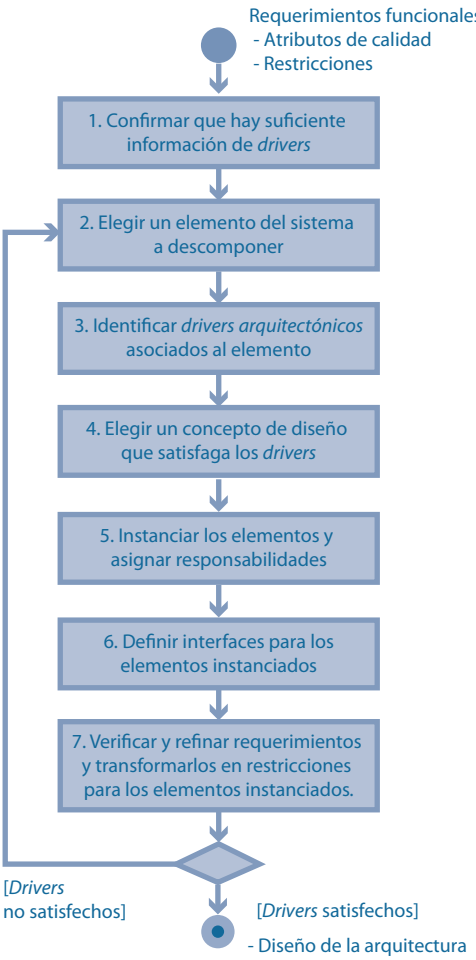
El método ADD recibe como entrada una lista de *drivers* arquitectónicos que incluyen escenarios de atributos de calidad, requerimientos funcionales primarios (por lo habitual casos de uso) y restricciones. Los pasos del método (mostrados en la figura 3-9) son:

1. **Confirmar que se tiene información suficiente sobre los *drivers* arquitectónicos**. El enfoque en este paso es asegurarse de que se cuenta con los datos suficientes acerca de los distintos *drivers* asociados al sistema y que están priorizados.
2. **Elegir un elemento del sistema a descomponer**. El elemento puede ser el sistema completo, si es un desarrollo nuevo, o un elemento obtenido de una iteración anterior. Existen diversos criterios de elección, los cuales incluyen el conocimiento que se tiene acerca de la arquitectura al momento y de los riesgos.
3. **Identificar *drivers* arquitectónicos asociados al elemento**. En esta etapa se elige una parte del conjunto inicial de *drivers* y se relacionan con el elemento elegido.
4. **Elegir un concepto de diseño que satisfaga los *drivers***. En este paso se selecciona un concepto general de diseño (ya sea patrones o tácticas) en relación con el elemento y los *drivers* elegidos. Este elemento es descompuesto mediante la aplicación de patrones asociados a tal concepto.
5. **Instanciar los elementos y asignar responsabilidades**. En esta fase se crean instancias de elementos derivados de los patrones y se describen sus responsabilidades.
6. **Definir interfaces para los elementos instanciados**. En este paso se identifican propiedades para los elementos instanciados y, más específicamente, las interfaces de los mismos.
7. **Verificar y refinar requerimientos y transformarlos en restricciones para los elementos instanciados.** En este paso se verifica si se han satisfecho los *drivers* y, en caso necesario, se refinan y se asocian a los elementos identificados durante la iteración.
8. **Repetir los pasos anteriores** para elementos que requieran un refinamiento mayor hasta cubrir la mayoría de los *drivers*.

Las iteraciones en el método ADD se llevan a cabo mientras sea necesario tomar decisiones de diseño adicionales para satisfacer los *drivers* arquitectónicos o hasta que termine el tiempo estipulado para diseñar. A la salida, el método produce el diseño de la arquitectura visto como un conjunto de estructuras que aún deben ser documentadas a efecto de comunicarlas a los demás involucrados. En la sección 3 del caso de estudio se ejemplifica la manera en que se lleva a cabo el método durante varias iteraciones de diseño.

Respecto de los métodos que hay, ADD es probablemente el que proporciona la guía más detallada para el diseño de la arquitectura. Algunos de sus inconvenientes son que no se especifica claramente cuánto abarca una iteración y que no proporciona criterios precisos de cuándo termina la actividad de diseñar.

Por otro lado, los conceptos de diseño que menciona el ADD son tácticas y patrones. Sin embargo, no hace mención de otros conceptos, por ejemplo los *frameworks*, por lo que es posible que el método produzca una arquitectura puramente conceptual la cual resultar complicada de mapear hacia las tecnologías para su implementación.



›**Figura 3-9**. Etapas del método ADD.

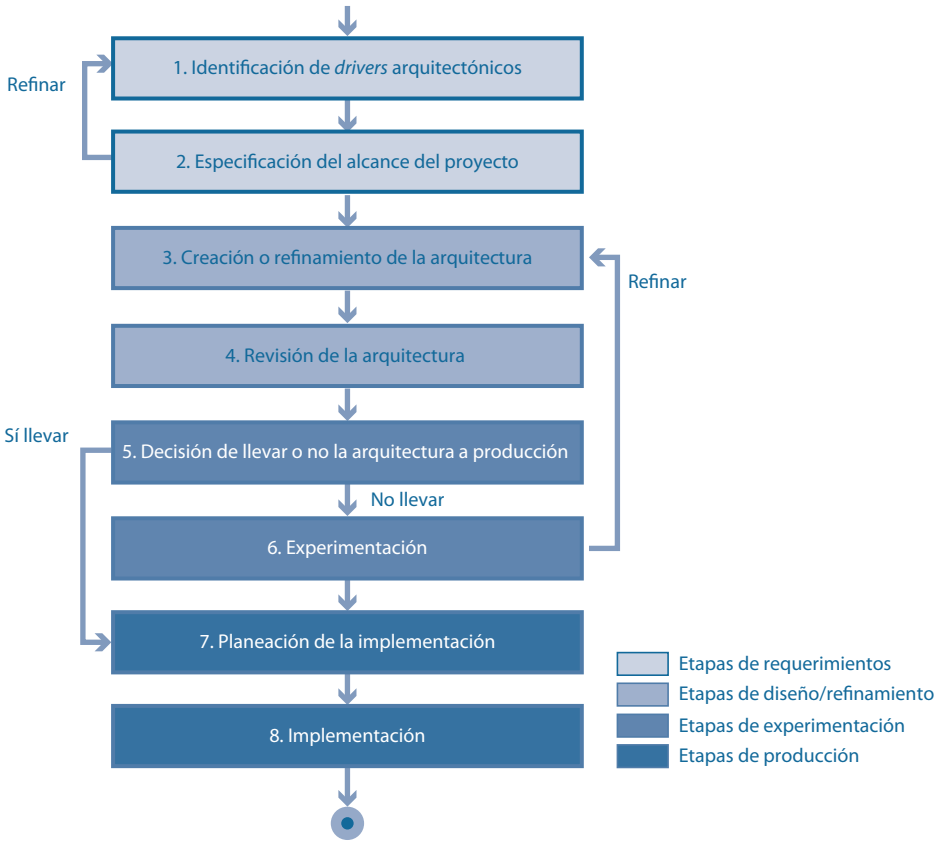
Por último, ADD no da una guía sobre qué partes del sistema se deben atacar tanto en las iteraciones iniciales como en las subsecuentes.

3.6.2 ACDM (etapa 3)

El método de diseño centrado en la arquitectura, o ACDM, por sus siglas en inglés, (Lattanze, 2008), fue introducido en el capítulo anterior (sección 2.3.2); sin embargo, en este se presenta nuevamente, ya que una de sus etapas cubre las actividades relacionadas con el diseño de la arquitectura. Cabe resaltar que para su autor, Anthony Lattanze, la filosofía subyacente a este método es usar la arquitectura como un plano para el proyecto entero y no solo como un artefacto técnico que se desarrolla una vez y se ignora posteriormente en las etapas de diseño e implementación del sistema.

La figura 3-10 presenta la estructura y etapas definidas por el ACDM, y en relación con el diseño, la etapa relevante es la tercera: Creación o refinamiento de la arquitectura.

El propósito principal de la etapa 3 es que el equipo correspondiente establezca el diseño arquitectónico inicial, o refine este como resultado de la evaluación. Las entradas de esta etapa son los *drivers* resultantes de las



› **Figura 3-10**. La etapa 3 del método de diseño centrado en la arquitectura se enfoca en la creación o el refinamiento de esta.

fases presentadas en el capítulo anterior y, en caso de que no sea la primera iteración del método, de una arquitectura parcial y una lista de problemas identificados en la evaluación. Los pasos de esta incluyen:

1. Establecer el contexto.
2. Realizar la partición del sistema con base en el contexto.
3. Asignar responsabilidades a los elementos y sus relaciones.
4. Documentar decisiones de diseño, responsabilidades de elementos y relaciones.
5. Cambiar de perspectiva si es necesario.
6. Repetir los pasos 2 a 5 hasta que se diseñen las interfaces.

Un aspecto interesante del método es que no considera diseñar y documentar como etapas distintas, siendo que en general la documentación se considera como una etapa diferente al diseño en el desarrollo arquitectónico (véase el capítulo 4). En la etapa 3 se crea o se refina la arquitectura y se crea o se actualiza la documentación.

Durante la primera iteración, el diseño parte de un diagrama de contexto, y el sistema, visto como un todo, es descompuesto en otros elementos. El autor del método recomienda, sin embargo, que no se dedique un tiempo excesivo a la etapa 3 en la primera iteración del diseño de la arquitectura, y que, mejor, se cree un bosquejo que se irá refinando con base en el principio iterativo general del método.

Lattanze hace énfasis en la importancia de registrar las decisiones de diseño mediante un "cuaderno de diseño", y sugiere que literalmente sea una libreta de papel. Como paso para delinear la arquitectura, el método

recomienda que se llegue hasta la definición de las interfaces entre los elementos y que el planteamiento de estas sea lo suficientemente detallado como para que equipos o individuos distintos construyan los elementos que las implementen sin problemas de comunicación y sin necesidad de tomar decisiones para diseñar, las cuales impidan que el sistema satisfaga los atributos de calidad.

El diseño de la arquitectura termina por lo habitual cuando provee guía suficiente para poder llevar a cabo el diseño detallado y la construcción de los módulos y otros elementos.

Las salidas de esta etapa son: un diseño documentado inicial o refinado, un plan maestro actualizado y, opcionalmente, una matriz de trazabilidad con la que se relacionen los elementos de diseño a los *drivers*.

De manera general el método que propone ACDM es muy interesante, pues busca alejarse completamente del enfoque del *Big Design Up Front* (BDUF) (realización del diseño completo en un solo bloque al inicio del proyecto) y se orienta mucho más hacia un enfoque iterativo y de retroalimentación temprana.

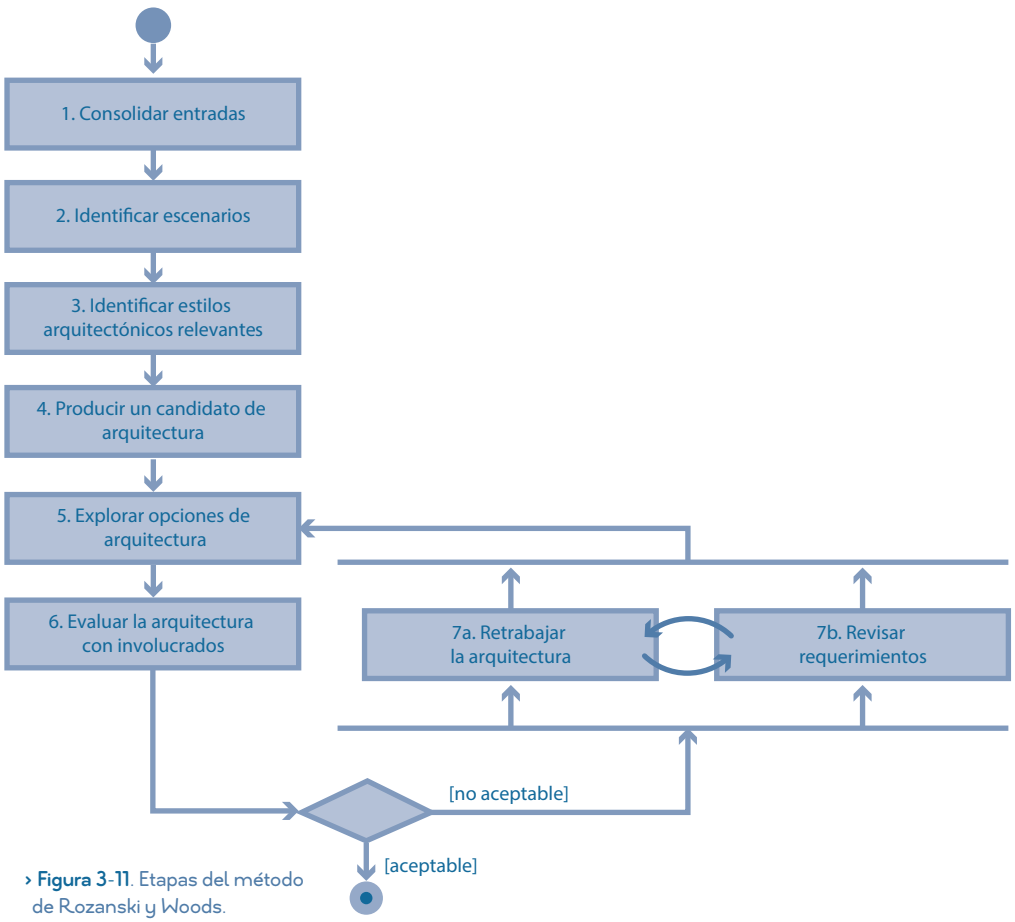
Una posible desventaja de este método es que puede ser difícil de implantar en organizaciones muy orientadas al desarrollo de sistemas bajo un enfoque secuencial (en cascada), en el cual se diseña toda la arquitectura sin realizar iteraciones. Al igual que el ADD, el ACDM es un método iterativo, aunque las repeticiones del primero corresponden a la actividad de diseño mientras que las del segundo incluyen la evaluación. A diferencia del ADD, la etapa 3 de ACDM proporciona diversos criterios y guías para diseñar.

3.6.3 Método de definición de arquitecturas de Rozanski y Woods

En su libro *Software Systems Architecture* (Rozanski y Woods, 2005), los autores describen un proceso de definición de arquitectura. Este cubre actividades que van más allá de la toma de decisiones de diseño y abarca en cierta forma todo el ciclo de desarrollo arquitectónico. Sus pasos se muestran en la figura 3-11 y se plantean a continuación:

1. **Consolidar las entradas.** El propósito de este paso es entender, validar y refinar las entradas iniciales (alcance y definición de contexto, preocupaciones de involucrados). A la salida se tiene la información de las entradas, pero consolidadas, es decir, que se han eliminado las inconsistencias principales, además de que se han dado respuestas a las preguntas abiertas y se ha generado una lista de áreas que requieren mayor exploración.
2. **Identificar escenarios.** El propósito de este paso es identificar un conjunto de escenarios arquitectónicos que ilustran los requerimientos más importantes del sistema. Se toman las entradas consolidadas y se produce una lista de dichos escenarios.
3. **Identificar estilos arquitectónicos relevantes.** El propósito es identificar uno o más estilos arquitectónicos probados que sirvan como base para la organización general del sistema. Se toman como entrada los escenarios, y se produce una lista de los estilos.
4. **Producir una arquitectura candidata.** El propósito de este paso es crear una primera versión de la arquitectura para que el sistema refleje las preocupaciones primarias y funcione como base para la evaluación y el refinamiento posteriores. A la entrada se toma el estilo arquitectónico, y a la salida se produce una versión preliminar de las vistas.
5. **Explorar opciones arquitectónicas.** El propósito de este paso es investigar diversas posibilidades para el sistema y tomar decisiones clave de arquitectura para elegir entre estas opciones. Esta exploración se hace mediante la aplicación de escenarios a los modelos para verificar si las decisiones tomadas hasta el momento los soportan. A la entrada se toman los diversos productos de los pasos anteriores, y a la salida se generan vistas más detalladas de algunas partes de la arquitectura.

6. **Evaluar la arquitectura con los involucrados.** El propósito de este paso es evaluar la arquitectura junto con los involucrados clave a efecto de identificar problemas y deficiencias en ella y conseguir que los involucrados la acepten. Como entrada se toma la documentación producida hasta el momento, y como salida se obtienen comentarios de revisión.
7. **a) Trabajar de nuevo la arquitectura.** El propósito de este paso es trabajar sobre los problemas que pudieran haber surgido durante la tarea de evaluación. Se hace por lo habitual de modo paralelo al paso b. Las entradas son los comentarios acerca de la arquitectura, y las salidas son las correcciones al diseño.
   b) Revisar los requerimientos. El propósito de este paso es considerar cambios en los requerimientos originales que deban hacerse a raíz de la evaluación. Las entradas son los comentarios acerca de la arquitectura, y las salidas son correcciones a los requerimientos.



› **Figura 3-11**. Etapas del método de Rozanski y Woods.

El método de Rozanski y Woods es muy similar al ACDM. Sin embargo, a diferencia del ADD, no proporciona pasos tan detallados para realizar las actividades de toma de decisiones de diseño en sí mismas. Es interesante la idea de que a partir de la evaluación sea necesario corregir los requerimientos del sistema.

3.6.4 Comparación de métodos

A continuación se muestra un cuadro comparativo de los métodos de diseño.

	ADD	ACDM	Método de Rozanski y Woods
Mecánica y enfoque	Diseño iterativo mediante la descomposición de elementos de manera recursiva	Iteraciones de diseño, documentación y evaluación	Iteraciones de diseño, documentación y evaluación
Participantes	Arquitecto	Arquitecto y otros involucrados	Arquitecto y otros involucrados
Entradas	*Drivers*	*Drivers* y alcance	Escenarios
Salidas	Esbozos de vistas	Vistas	Vistas
Criterios de terminación	Se satisfacen *drivers*.	Los experimentos ya no revelan riesgos o estos son aceptables	Los interesados están de acuerdo en que el diseño satisface sus preocupaciones
Conceptos de diseño utilizados	Tácticas y patrones	Estilos arquitectónicos, patrones y tácticas	Estilos arquitectónicos y patrones

EN RESUMEN

En este capítulo tratamos el diseño, que es la actividad central asociada al desarrollo de *arquitecturas de software*. Presentamos el concepto general de lo que es el diseño y los principios que aplican al diseñar la arquitectura de un sistema de *software*. Se presentaron también diversos conceptos de diseño, los cuales se utilizan para producir las estructuras que conforman la arquitectura. Los conceptos de diseño incluyen los patrones, las tácticas y los *frameworks*, así como algunos conceptos adicionales. Por último, se presentaron diversos métodos con los que se lleva a cabo de forma sistemática el diseño de la arquitectura. Al final de este proceso se tiene una arquitectura, sin embargo, es necesario comunicar el resultado a los distintos involucrados, lo cual será el tema del siguiente capítulo.

PREGUNTAS PARA ANÁLISIS

1. ¿El diseño de la arquitectura de sistemas de *software* es distinto al de otros productos? Para responder aplique la definición general este presentada en la sección 3.1.1 en relación con otro dominio (por ejemplo, el diseño de un automóvil) y contrastarla con el diseño de la arquitectura.
2. ¿Cómo acota el diseño de la arquitectura al diseño detallado de los módulos? Piense, por ejemplo, en un escenario que tenga que ver con el desempeño y con un tiempo de respuesta limitado.
3. En el momento de su diseño, algunos sistemas requieren más creatividad que otros, ¿cuáles serían ejemplos de ello?
4. Considere un programa que lee datos de un archivo, realiza un procesamiento sobre los datos leídos e imprime el resultado mostrado en pantalla. Elija dos esquemas distintos de modularización de este programa y discuta con un(a) compañero(a) qué diferencias pueden resultar de ellas acerca de cualesquier atributos de calidad, como reutilización o modificabilidad.
5. Suponga dos módulos A y B conectados entre ellos. El módulo B contiene un arreglo que guarda datos y proporciona en su interfaz un método que regresa el arreglo. ¿Cómo es el acoplamiento entre A y B? Considere qué sucedería si posteriormente se decide cambiar el arreglo por una lista ligada, y proponga una manera de resolver el problema si es que existe.

Continúa...

Continuación...

PREGUNTAS PARA ANÁLISIS

6. Para elegir un concepto de diseño sobre otro conviene listar los beneficios y desventajas de ambos en relación con el problema que se quiere resolver. Considere las tácticas de redundancia pasiva y de redundancia activa con los que se resuelven problemas de disponibilidad. Discuta con un(a) compañero(a) los pros y los contras de ambas opciones.
7. No existe un consenso sobre en qué momento del diseño se deben elegir las tecnologías. Algunos autores sugieren hacer primero un diseño puramente conceptual y luego ligarse a ellas. Sin embargo, en la práctica se observa a menudo que al menos una parte se elige en etapas tempranas de tal proyecto. ¿Qué dificultades podría haber con el enfoque que inicia con un diseño solo conceptual? ¿En qué tipo de proyectos o situaciones sería recomendable?
8. Discuta con un(a) compañero(a) las posibles razones de por qué en el diseño de la arquitectura solo se consideran casos de uso primarios y no todo el conjunto de funcionalidad.
9. Supongamos un sistema pequeño con un diseño "tradicional" de tres capas (presentación, negocio y datos). En ellas se ubican módulos con los que se soporta tres casos de uso del sistema. Considere dos estrategias de desarrollo para un equipo de tres personas: asignación "por capa" y asignación "por caso de uso". En la primera estrategia, un ingeniero es asignado a cada capa, en la segunda, otro ingeniero es asignado a cada caso de uso. Discuta con un(a) compañero(a) si ambas estrategias requieren un mismo nivel de detalle en la especificación de las interfaces de los módulos. Justifique la respuesta.
10. ¿Cómo integraría un método de diseño, como el ADD, en un proceso de desarrollo iterativo?