CENGAGE Learning

Humberto Cervantes Maceda
Perla Velasco-Elizondo
Luis Castro Careaga

ARQUITECTURA DE SOFTWARE

Conceptos y ciclo de desarrollo

l: 0000021597
.1
7
6

034 R4 2016 _ 1267 MAT: 000021547
005.1
-C22
A71
2016

ARQUITECTURA DE SOFTWARE

Conceptos y ciclo de desarrollo

UNIVERSIDAD TÉCNICA DEL NORTE
BIBLIOTECA
Ibarra - Ecuador

UNIVERSIDAD TECNICA DEL NORTE
BIBLIOTECA
Vía de adquisición: Compra
Documento No.: 31-01-2016 / 1267
Fecha: 29-01-2016
Valor unitario: 28.00
Código de Barras: 055301
Anexos:

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

CENGAGE Learning®

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

© D.R. 2016 por Cengage Learning Editores, S.A. de C.V., una Compañía de Cengage Learning, Inc.
Corporativo Santa Fe
Av. Santa Fe núm. 505, piso 12
Col. Cruz Manca, Santa Fe
C.P. 05349, México, D.F.
Cengage Learning® es una marca registrada usada bajo permiso.

DERECHOS RESERVADOS. Ninguna parte de este trabajo amparado por la Ley Federal del Derecho de Autor podrá ser reproducida, transmitida, almacenada o utilizada en cualquier forma o por cualquier medio, ya sea gráfico, electrónico o mecánico, incluyendo, pero sin limitarse a lo siguiente: fotocopiado, reproducción, escaneo, digitalización, grabación en audio, distribución en internet, distribución en redes de información o almacenamiento y recopilación en sistemas de información, a excepción de lo permitido en el Capítulo III, Artículo 27 de la Ley Federal del Derecho de Autor, sin el consentimiento por escrito de la Editorial.

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 Cosegraf: abril 2015
Progreso No. 10, Col. Centro
Ixtapaluca, Edo de México

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

CONTENIDO BREVE

PRÓLOGO xv

CAPÍTULO 1

INTRODUCCIÓN: LA ARQUITECTURA Y EL DESARROLLO DE *SOFTWARE* 1

CAPÍTULO 2

REQUERIMIENTOS: IDENTIFICACIÓN DE *DRIVERS* ARQUITECTÓNICOS 9

CAPÍTULO 3

DISEÑO: TOMA DE DECISONES PARA CREAR ESTRUCTURAS 29

CAPÍTULO 4

DOCUMENTACIÓN: COMUNICAR LA ARQUITECTURA 49

CAPÍTULO 5

EVALUACIÓN: ASEGURAR LA CALIDAD EN LA ARQUITECTURA 73

CAPÍTULO 6

IMPLEMENTACIÓN: CONVERTIR EN REALIDAD LAS IDEAS ARQUITECTÓNICAS 93

CAPÍTULO 7

ARQUITECTURA Y MÉTODOS ÁGILES: EL CASO DE *SCRUM* 105

APÉNDICE

CASO DE ESTUDIO 119

GLOSARIO 161

BIBLIOGRAFÍA 165

CONTENIDO DETALLADO

ACERCA DE LOS AUTORES xiii

PRÓLOGO xv

CAPÍTULO 1

INTRODUCCIÓN: LA ARQUITECTURA Y EL DESARROLLO DE *SOFTWARE* 1

1.1 VISIÓN GENERAL DEL DESARROLLO DE SISTEMAS DE *SOFTWARE* 2

1.2 DEFINICIÓN DE *ARQUITECTURA DE SOFTWARE* 3

1.3 ARQUITECTURA, ATRIBUTOS DE CALIDAD Y OBJETIVOS DE NEGOCIO 4

1.4 CICLO DE DESARROLLO DE LA ARQUITECTURA 5

- **1.4.1** Requerimientos de la arquitectura 5
- **1.4.2** Diseño de la arquitectura 5
- **1.4.3** Documentación de la arquitectura 6
- **1.4.4** Evaluación de la arquitectura 6
- **1.4.5** Implementación de la arquitectura 6

1.5 BENEFICIOS DE LA ARQUITECTURA 6

- **1.5.1** Aumentar la calidad de los sistemas 6
- **1.5.2** Mejorar tiempos de entrega de proyectos 6
- **1.5.3** Reducir costos de desarrollo 7

1.6 EL ROL DEL ARQUITECTO 7

- EN RESUMEN 8
- PREGUNTAS PARA ANÁLISIS 8

CAPÍTULO 2

REQUERIMIENTOS: IDENTIFICACIÓN DE *DRIVERS* ARQUITECTÓNICOS 9

2.1 REQUERIMIENTOS 10

2.2 REQUERIMIENTOS CON DISTINTOS TIPOS Y NIVELES DE ABSTRACCIÓN 11

- **2.2.1** Requerimientos de usuario y requerimientos funcionales 12
- **2.2.2** Atributos de calidad 13
- **2.2.3** Restricciones 14
- **2.2.4** Reglas de negocio e interfaces externas 14
- **2.2.5** Consideraciones importantes 15

2.3 *DRIVERS ARQUITECTÓNICOS* 15
2.3.1 *Drivers* funcionales 15
2.3.2 *Drivers* de atributos de calidad 16
2.3.3 *Drivers* de restricciones 16
2.3.4 Otra información 16
2.3.5 Influencia de los *drivers arquitectónicos* en el diseño de la arquitectura 17
2.4 FUENTES DE INFORMACIÓN PARA LA EXTRACCIÓN DE *DRIVERS ARQUITECTÓNICOS* 17
2.4.1 Documento de visión y alcance 18
2.4.2 Documento de requerimientos de usuario 18
2.4.3 Documento de especificación de requerimientos 19
2.5 MÉTODOS PARA LA IDENTIFICACIÓN DE *DRIVERS ARQUITECTÓNICOS* 19
2.5.1 Taller de atributos de calidad (QAW) 19
2.5.2 Método de diseño centrado en la arquitectura (ACDM-etapas 1 y 2) 22
2.5.3 FURPS+ 24
2.5.4 Comparación de los métodos y el modelo 24
■ EN RESUMEN 26
■ PREGUNTAS PARA ANÁLISIS 27

CAPÍTULO 3

DISEÑO: TOMA DE DECISIONES PARA CREAR ESTRUCTURAS 29
3.1 DISEÑO Y NIVELES DE DISEÑO 30
3.1.1 Diseño y arquitectura 30
3.1.2 Niveles de diseño 31
3.2 PROCESO GENERAL DE DISEÑO DE LA ARQUITECTURA 32
3.3 PRINCIPIOS DE DISEÑO 33
3.3.1 Modularidad 33
3.3.2 Cohesión alta y acoplamiento bajo 34
3.3.3 Mantener simples las cosas 34
3.4 CONCEPTOS DE DISEÑO 35
3.4.1 Patrones 35
3.4.2 Tácticas 37
3.4.3 *Frameworks* 38
3.4.4 Otros conceptos de diseño 39
3.5 DISEÑO DE LAS INTERFACES 40
3.6 MÉTODOS DE DISEÑO DE ARQUITECTURA 41

3.6.1 Diseño guiado por atributos (ADD) 42
3.6.2 ACDM (etapa 3) 43
3.6.3 Método de definición de arquitecturas de Rozanski y Woods 45
3.6.4 Comparación de métodos 46
EN RESUMEN 47
PREGUNTAS PARA ANÁLISIS 47

CAPÍTULO 4

DOCUMENTACIÓN: COMUNICAR LA ARQUITECTURA 49
4.1 ¿QUÉ SIGNIFICA DOCUMENTAR? 50
4.2 DOCUMENTACIÓN EN EL CONTEXTO DE *ARQUITECTURA DE SOFTWARE* 51
4.3 RAZONES PARA DOCUMENTAR LA ARQUITECTURA 51
4.3.1 Mejorar la comunicación de información sobre la arquitectura 51
4.3.2 Preservar información sobre la arquitectura 52
4.3.3 Guiar la generación de artefactos para otras fases del desarrollo 52
4.3.4 Proveer un lenguaje común entre diversos interesados en el sistema 53
4.4 VISTAS 54
4.4.1 Vistas lógicas 55
4.4.2 Vistas de comportamiento 56
4.4.3 Vistas físicas 57
4.5 NOTACIONES 59
4.5.1 Notaciones informales 59
4.5.2 Notaciones semiformales 60
4.5.3 Notaciones formales 60
4.6 MÉTODOS Y MARCOS CONCEPTUALES DE DOCUMENTACIÓN DE ARQUITECTURA 61
4.6.1 Vistas y más allá 62
4.6.2 4+1 Vistas 64
4.6.3 Puntos de vista y perspectivas 64
4.6.4 ACDM (etapas 3 y 4) 65
4.6.5 Otros 67
4.6.6 Comparación de los métodos y marcos conceptuales 67
4.7 RECOMENDACIONES PARA ELABORAR LA DOCUMENTACIÓN 68
EN RESUMEN 70
PREGUNTAS PARA ANÁLISIS 70

2.2 Fase del problema 122
2.3 Objetivos de negocio 122
3. VISIÓN DE LA SOLUCIÓN 123
3.1 Fase de visión 123
3.2 Características del sistema 123
4. ALCANCE 124
5. CONTEXTO DEL SISTEMA 124
5.1 Interesados 124
5.2 Diagrama de contexto 125
5.3 Entorno de operación 125
6. INFORMACIÓN ADICIONAL 125

SECCIÓN 2: REQUERIMIENTOS DE LA ARQUITECTURA 127
2.1 *Drivers* funcionales 128
2.1.1 Modelo de casos de uso 128
2.1.2 Elección de casos de uso primarios 129
2.2 *Drivers* de atributos de calidad 129
2.3 *Drivers* de restricciones 130

SECCIÓN 3: DISEÑO DE LA ARQUITECTURA 131
3.1 Primera iteración: estructuración general del sistema 132
3.2 Segunda iteración: integración de la funcionalidad a las capas 135
3.3 Tercera iteración: desempeño en capa de datos 138

SECCIÓN 4: DOCUMENTACIÓN 143
4.1 Generar una lista de vistas candidatas 144
4.2 Combinar las vistas 145
4.3 Priorizar las vistas 146
4.4 Ejemplo de vista 146

SECCIÓN 5: EVALUACIÓN 153
5.1 Realización de la evaluación 154
5.1.1 Identificación de las decisiones arquitectónicas 154
5.1.2 Generación del árbol de utilidad 154
5.1.3 Análisis de las decisiones arquitectónicas 155
5.2 Resultados de la evaluación 158

SECCIÓN 6: CONCLUSIÓN 159

GLOSARIO 161

BIBLIOGRAFÍA 165

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 Rodriguez, 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.

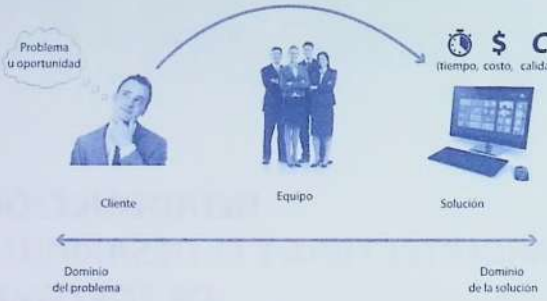


Figura 1-1 Visión simplificada del desarrollo de sistemas

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.

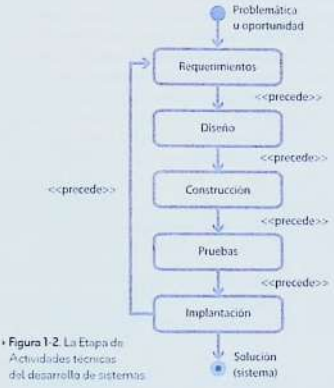


Figura 1-2. La Etapa de Actividades técnicas del desarrollo de sistemas

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.

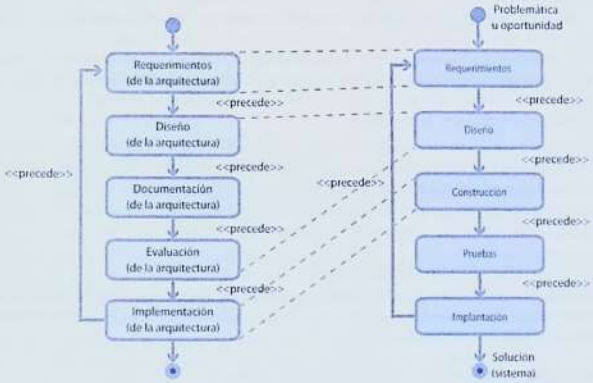
Las *decisiones de diseño* que se toman para estructurar un sistema permitirán o impedirán que se satisfagan los atributos de calidad. Por ejemplo, un sistema estructurado de manera tal que una petición deba transitar por muchos componentes implantados en nodos distintos antes de que se devuelva una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

UNIVERSIDAD TECNICA DEL NORTE BIBLIOTECA

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.

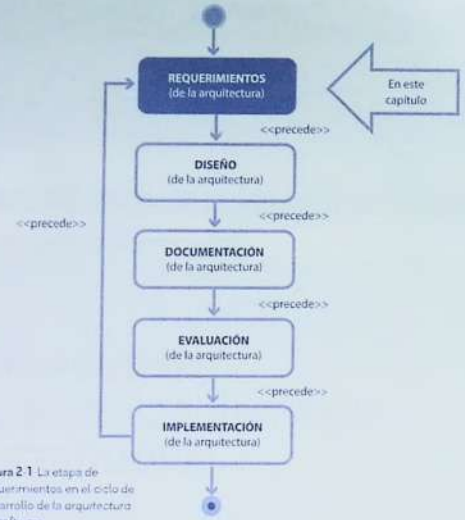
PREGUNTAS PARA ANÁLISIS

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 autobus, 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

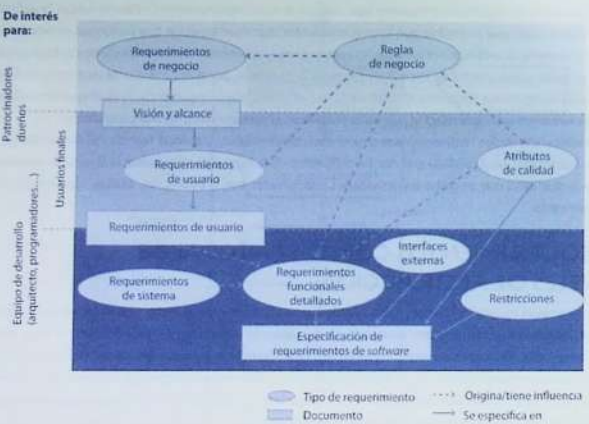
Aunque la ingeniería de requerimientos existe desde hace varios años, y a la fecha se han creado diversos métodos y herramientas que dan soporte a sus actividades, obtener y documentar requerimientos sigue siendo una tarea complicada. Recabarlos no solo consiste en escuchar al propietario o usuario final 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.

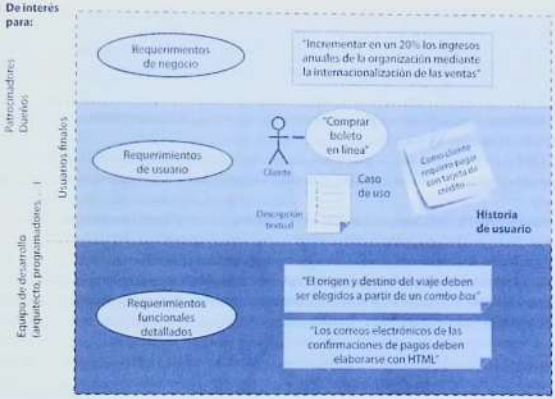
2.2.1 Requerimientos de usuario y requerimientos funcionales

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).

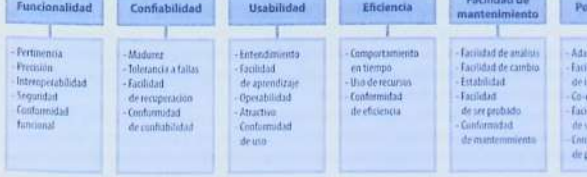
[1] Un combo box es una lista desplegable para hacer una selección en un conjunto de opciones predefinidas.



› 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

UNIVERSIDAD TÉCNICA DEL NORTE BIBLIOTECA

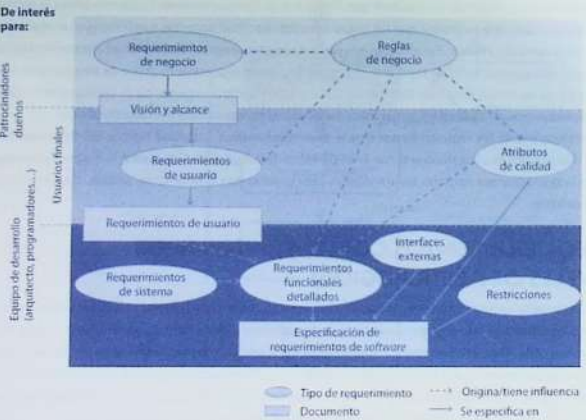


Figura 2-2 Tipos de requerimientos según Wiegers.

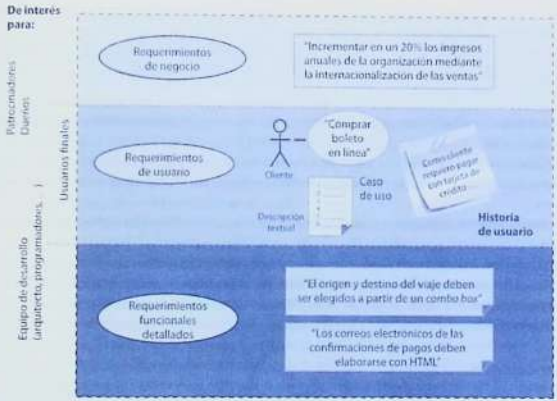
2.2.1 Requerimientos de usuario y requerimientos funcionales

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).

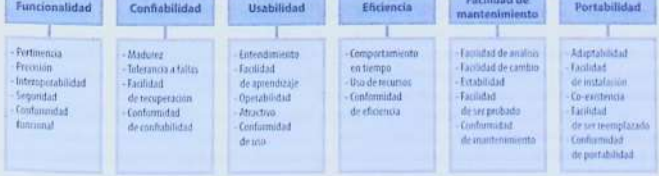
[1] Un combo box es una lista desplegable para hacer una selección en un conjunto de opciones predefinidas.



› 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

UNIVERSIDAD TÉCNICA DEL NORTE BIBLIOTECA

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

© Karlmarx Cerpa Morales / Archivo personal Pearson Educación de México

› **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[1], 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.

[1] 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.

UNIVERSIDAD TECNICA DEL NORTE BIBLIOTECA

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.

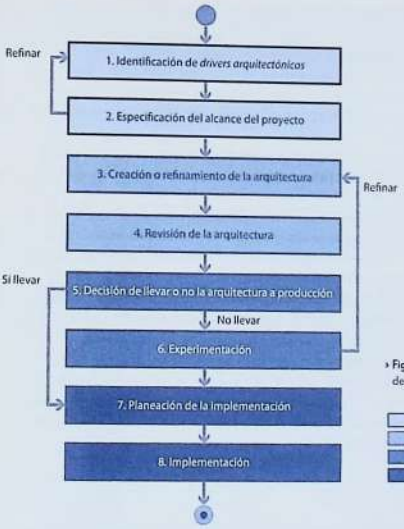
Puesto que estamos describiendo métodos que dan soporte a la etapa de requerimientos, solo abundaremos en las etapas 1 y 2. Durante ellas se llevan a cabo actividades que permiten la identificación, especificación y priorización de *drivers arquitectónicos*.

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 tiene 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 enfasis 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

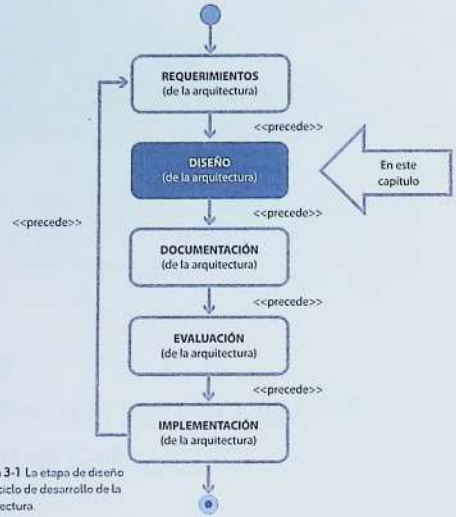
1. Considere esta frase: "Los requerimientos de usuario y los atributos de calidad son conceptos ortogonales". ¿Está de acuerdo? ¿Por qué sí o no?
2. Suponga que para determinado sistema se han identificado como los principales *drivers* los de atributos de calidad, seguridad y desempeño. Por ello han 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.

© Humberto Cervantes Maceda / Perla Velasco Elizondo / Luis Castro Cancela

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 *restricciones* 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.

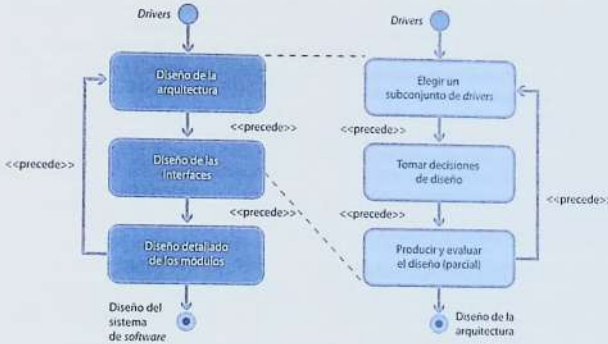
› Figura 3-3 Diseño como caja negra (las entradas están a la izquierda, las salidas, a la derecha, y los responsables, arriba)

© Humberto Cervantes Maceda Erika Victoria Estrada Aguilar Clara Garduño

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.

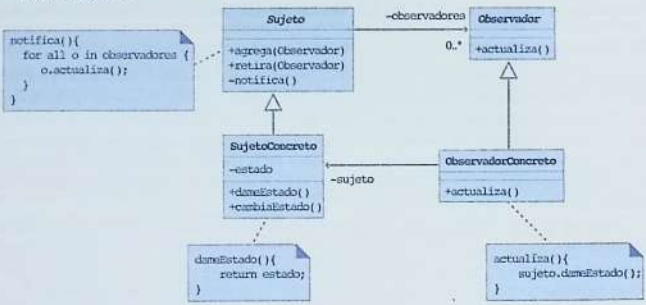
Desde la aparición del libro de GoF han surgido muchos catálogos con patrones de diseño de *software* enfocados en distintas áreas. Existen, por ejemplo, catálogos de patrones dirigidos a la estructuración general del sistema, también llamados *estilos arquitectónicos* o *arquitecturas de referencia* (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.

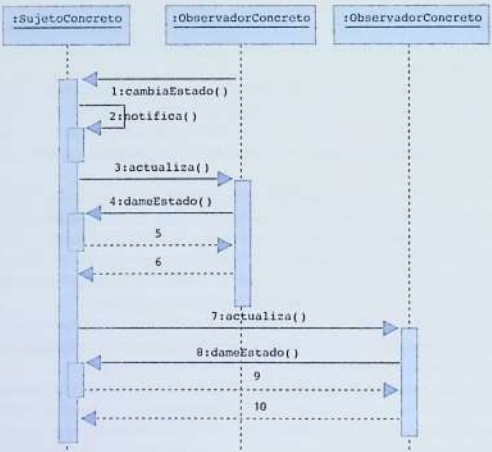
La figura 3-5 muestra un ejemplo de patrón de diseño llamado Observador (*Observer*), proveniente del libro de GoF, cuyo propósito es "definir una dependencia entre objetos de uno a muchos, de modo que cuando el estado del objeto con dependientes múltiples cambie, todos ellos sean notificados y actualizados 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:



© Humberto Cervantes Maceda - Fiete Velasco Elizondo / Juan Carlos Cuevas

› 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, especifica 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

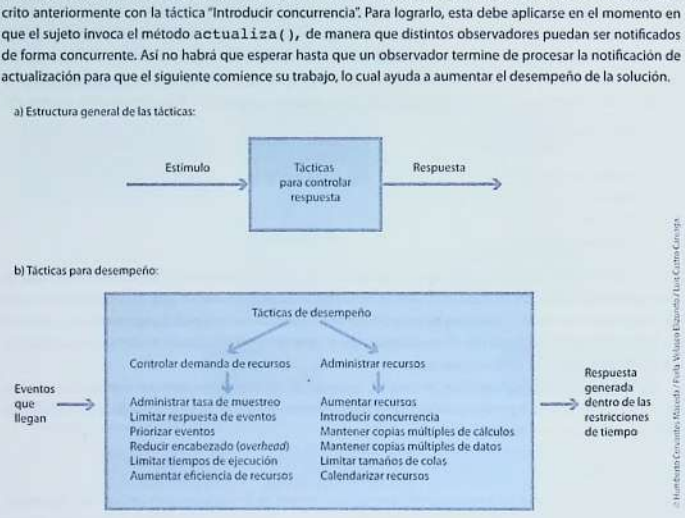
Las tácticas son conceptos de diseño que influyen sobre el control de la respuesta a un atributo de calidad particular. A diferencia de los patrones, no presentan soluciones conceptuales detalladas, sino que son técnicas probadas de las ciencias de la computación con las que se resuelven problemas en 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.



© Humberto Cervantes Maceda / Perla Velasco Elizondo / Luis Gerardo Gallegos

› 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.

© Humberto Cervantes Maceda y Perla Velasco Elizondo - Luis Carlos Cuevas

› **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.

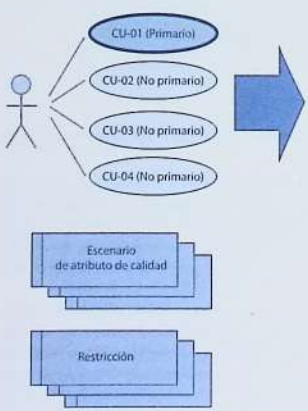
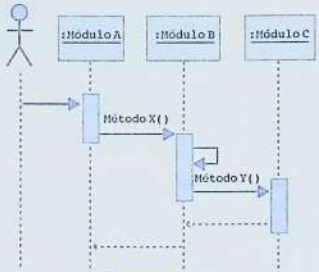


Diagrama de secuencia para CU-01(primario), de acuerdo a las estructuras físicas y lógicas (no mostradas).

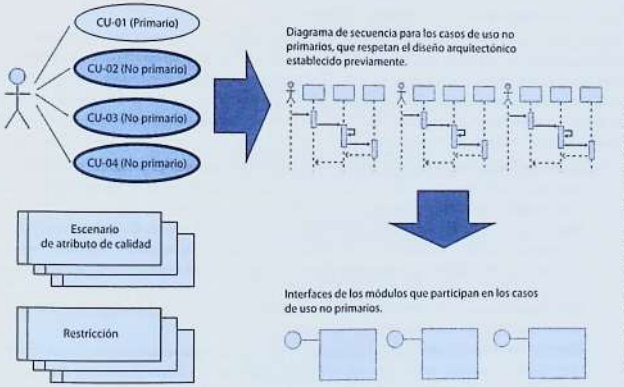


Interfaz del módulo B con métodos identificados a partir de la realización del diagrama de secuencia.
Nota: Las interfaces de los otros módulos no se muestran.

© Humberto Cervantes Maceda y Rafael Antonio Gómez Alcalá, Universidad Autónoma Metropolitana

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:

© Humberto Cervantes Maceda | Perla Velasco Elizondo | Ana Isabel Garcia

UNIVERSIDAD TECNICA DEL NORTE

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.

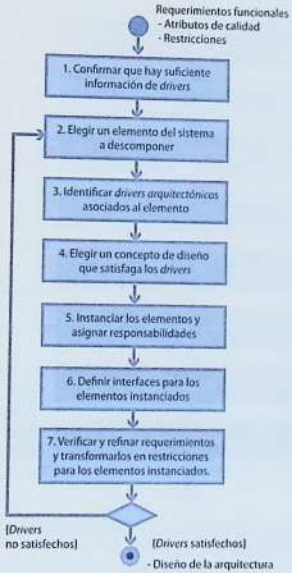
El método ADD recibe como entrada una lista de *drivers* arquitectónicos que incluyen escenarios de atributos de calidad, requerimientos funcionales primarios (por lo habitual casos de uso) y restricciones. Los pasos del método (mostrados en la figura 3-9) son:

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

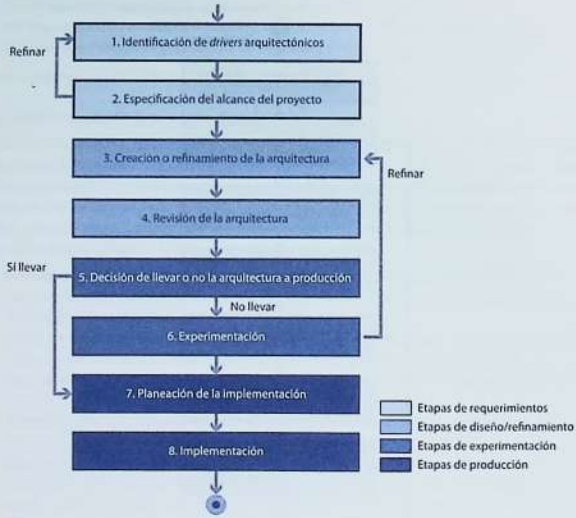
Por último, ADD no da una guía sobre qué partes del sistema se deben atacar tanto en las iteraciones iniciales como en las subsecuentes.

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



© Humberto Cervantes Maceda / Frida Velázquez Elizondo / Luis Castro Carvajal

› **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

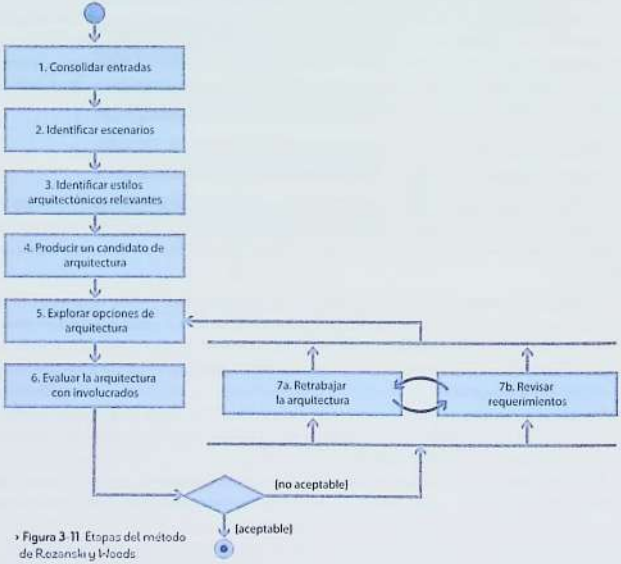
En su libro *Software Systems Architecture* (Rozanski y Woods, 2005), los autores describen un proceso de definición de arquitectura. Este cubre actividades que van más allá de la toma de decisiones de diseño y abarca en cierta forma todo el ciclo de desarrollo arquitectónico. Sus pasos se muestran en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

CAPÍTULO 4

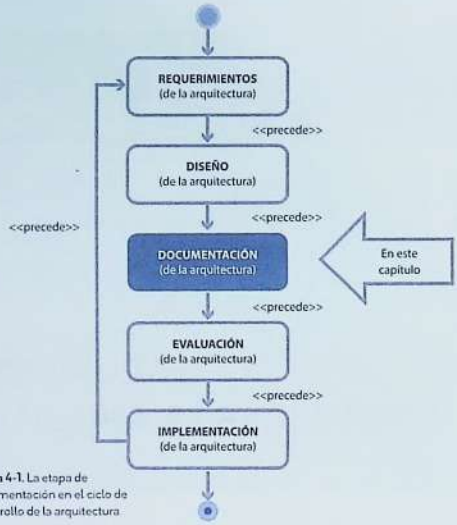
DOCUMENTACIÓN: COMUNICAR LA ARQUITECTURA

En el capítulo anterior explicamos que, durante la etapa de diseño, el arquitecto traduce los *drivers* en las estructuras arquitectónicas que conforman el sistema. Sin embargo, todo este esfuerzo sería inútil si, por ejemplo, durante la implementación de estas los desarrolladores tuvieran retrasos frecuentes porque no entienden completamente dichas estructuras.

En este capítulo describiremos la etapa de documentación de la arquitectura, resaltada en la figura 4-1. La etapa se enfoca en la elaboración de documentación con la que se comuniquen de manera eficiente las estructuras de la arquitectura a los diversos interesados en el sistema.

Describiremos de manera inicial qué se entiende por documentar en un contexto general. Después trasladaremos esta noción al ámbito de la *arquitectura de software* y discutiremos varias razones importantes para documentar una arquitectura. Presentaremos posteriormente los conceptos relevantes en esta etapa, las notaciones, así como algunos de los enfoques más conocidos que pueden utilizarse para soportar las actividades en esta etapa.

Concluiremos el capítulo enumerando una serie de recomendaciones útiles para generar documentación arquitectónica de buena calidad.



› **Figura 4-1.** La etapa de documentación en el ciclo de desarrollo de la arquitectura

© Humberto Cervantes Maceda y Perla Velasco Elizondo y Luis Castro Carranza

4.1 ¿QUÉ SIGNIFICA DOCUMENTAR?

En prácticamente cualquier entorno, conocer acerca de las políticas, procesos y productos que empleamos en nuestras actividades diarias es fundamental para la realización correcta de estas. En consecuencia, tener acceso a fuentes documentales adecuadas que describan la información necesaria resulta muy útil comparado con tener que extraerla directamente de las personas que la conocen o, peor aún, no tener esta información.

En términos generales, la palabra documentar se asocia al proceso de elaborar un documento, o conjunto de documentos, que tienen como propósito comunicar información relevante de un proceso, producto o entidad. Además de comunicarla, documentar permite que la información persista. Esto es muy importante si pensamos que en muchas ocasiones información relevante para soportar nuestras actividades laborales diarias no se documenta y es conocida solo por algunas personas dentro de la organización. Cuando ellas dejan de laborar en la organización, recuperar la información puede ser complicado o quizás imposible.

Ejemplos comunes de documentación incluyen las guías de usuario, manuales de políticas y procedimientos, así como tutoriales, documentos técnicos o guías de referencia rápida. Con los avances actuales en materia de tecnologías de la información, la documentación no solo puede estar plasmada en un conjunto de papeles. Medios como el video, los documentos electrónicos o los *wikis* se utilizan en la actualidad para plasmarla. Dependiendo del dominio o contexto de la organización, también se pueden variar las notaciones utilizadas para especificar la información en todos estos medios.

4.2 DOCUMENTACIÓN EN EL CONTEXTO DE *ARQUITECTURA DE SOFTWARE*

En el capítulo 1 mencionamos que la *arquitectura de software* es el término utilizado para referirse a las estructuras que conforman un sistema. En el capítulo 3 explicamos que durante la etapa de diseño, y con base en la información obtenida durante la fase de requerimientos, el arquitecto define esas estructuras mediante la toma de decisiones de diseño. De esta forma, en el contexto del ciclo de desarrollo de la *arquitectura de software*,

> la etapa de documentación se centra en la generación de documentos que describen las estructuras de la arquitectura con el propósito que esta información pueda ser comunicada de manera eficiente a los diversos interesados en el sistema

Aunque en la figura 4-1 la etapa de documentación de la arquitectura aparece después de la de diseño, en la práctica es difícil imaginar que se lleva a cabo estrictamente después de que todo el diseño arquitectónico está terminado. Muchas de las actividades en la etapa de diseño se realizan de manera concurrente con actividades de esta. Un ejemplo es que durante el diseño se hace uso de herramientas de modelado al crear las estructuras; estos modelos son una parte importante de la documentación.

4.3 RAZONES PARA DOCUMENTAR LA ARQUITECTURA

Al inicio del capítulo mencionamos que, pese a haber diseñado la mejor arquitectura, todo este trabajo puede ser poco productivo si personas como los diseñadores y desarrolladores del sistema tienen problemas al realizar sus tareas porque no existe documentación que la describa y, en cambio, hay información al respecto que no les ha sido comunicada. En la presente sección abundamos sobre estas y algunas otras razones por las cuales es importante documentar la arquitectura:

1. Mejorar la comunicación de información sobre la arquitectura.
2. Preservar la información sobre la arquitectura.
3. Guiar la generación de artefactos en otras fases del ciclo de vida.
4. Proveer un lenguaje común entre diversos interesados en el sistema.

4.3.1 Mejorar la comunicación de información sobre la arquitectura

Es claro que una vez diseñada la arquitectura deberá presentarse en algún momento a los diversos interesados en el sistema. Si bien es cierto que durante su diseño algunos modelos preliminares o bocetos pudieron haber sido generados, y el arquitecto podría emplearlos como apoyo para clarificar información durante la presentación de la arquitectura, es posible que su utilidad sea limitada pues podrían estar especificados en una notación no estándar o podrían contener información muy general. En este caso, el arquitecto tendría que comunicar de forma verbal la información faltante.

Una de las ventajas de la comunicación verbal es que es inmediata; sin embargo, se sabe ampliamente que esta dista de ser efectiva siempre. La efectividad de la comunicación verbal se ve disminuida en ocasiones debido a la presencia de ambigüedades, las cuales, se refieren al hecho de que la información comunicada puede ser interpretada de más de una forma. La ambigüedad se genera a menudo por limitaciones de las personas al usar

un lenguaje, deficiencias propias de este o ambas. La omisión de información es también un factor que afecta la eficiencia de la comunicación verbal. Por ejemplo, durante la presentación el arquitecto podría omitir información relevante acerca de la arquitectura por considerarla obvia para su audiencia.

En este contexto afirmamos que la existencia de documentación adecuada podría minimizar todos estos problemas. El arquitecto podría utilizar la documentación de la arquitectura como un elemento de soporte durante la presentación verbal de esta a los diferentes involucrados en el sistema.

4.3.2 Preservar información sobre la arquitectura

La documentación es persistente por naturaleza. De esta forma, la existencia de documentación sobre la arquitectura, y el acceso a aquella misma, podría ser de utilidad para resolver varias situaciones que pueden presentarse durante el desarrollo de un sistema.

Por ejemplo, si nuevos miembros se unen al equipo de desarrollo, pueden usar la documentación para comprender de forma más rápida la arquitectura del sistema. Esto es positivo y adecuado, comparado con el escenario de depender de la disponibilidad del arquitecto para conocer de su propia voz toda la información al respecto.

De forma similar, la existencia de documentación permite que cuando personas clave involucradas en el diseño de la arquitectura se ausentan, el resto del equipo de desarrollo puede consultarla para obtener la información requerida sin tener que esperar a que estas personas regresen.

Otro ejemplo es que las actividades de mantenimiento o actualización a un sistema pueden llevarse a cabo de manera más localizada si existe un documento que describe la arquitectura de este. Este ejemplo es más valioso si pensamos en que el sistema en turno es uno legado, el cual fue diseñado y desarrollado por personal que ya no labora en la compañía.

Todos los cambios realizados a la arquitectura deberían ser registrados en la documentación. De esta forma, estos pueden preservarse y ser conocidos por aquellos que tienen acceso a la documentación de la arquitectura.

4.3.3 Guiar la generación de artefactos para otras fases del desarrollo

En los capítulos anteriores expusimos que la *arquitectura de software* representa la estructura con la cual se proporcionará lo que el cliente o usuario espera del sistema. Por tal razón la arquitectura es un artefacto muy importante para guiar otras actividades dentro del ciclo de desarrollo de sistemas. Aunque todas ellas se llevan a cabo por personas que pertenecen a diferentes grupos de interesados en el sistema, por ejemplo, los líderes de proyecto o quienes desarrollan o prueban el sistema, su realización requiere que todas estas personas tengan una visión compartida y se desenvuelvan en un contexto de trabajo común atendiendo a esta. Como se ilustra en la figura 4-2, la documentación de la arquitectura es este contexto de trabajo común que permite guiar y restringir la forma en que se llevan a cabo varias actividades durante determinadas fases del ciclo de desarrollo del sistema.

Por ejemplo, al contener información acerca de todas las estructuras que conforman la arquitectura, la documentación arquitectónica sirve como fuente para las personas encargadas de la planeación del desarrollo del sistema. De forma similar, al contener información sobre el particionamiento y la funcionalidad que albergan los elementos que conforman las estructuras, así como de las relaciones de estos elementos en la arquitectura, la documentación arquitectónica es útil como guía para las actividades relacionadas con la construcción y la prueba del sistema.

En la sección 4.4 explicaremos que además de los de *software*, los elementos que conforman las estructuras arquitectónicas pueden ser también de *hardware*; por ejemplo los equipos de cómputo en donde se instalan los elementos de *software*. Para promover algún *driver*, durante el diseño el arquitecto pudo haber decidido colocar los equipos de cómputo en ubicaciones geográficamente dispersas. Al contener esta información, la documen-

un lenguaje, deficiencias propias de este o ambas. La omisión de información es también un factor que afecta la eficiencia de la comunicación verbal. Por ejemplo, durante la presentación el arquitecto podría omitir información relevante acerca de la arquitectura por considerarla obvia para su audiencia.

En este contexto afirmamos que la existencia de documentación adecuada podría minimizar todos estos problemas. El arquitecto podría utilizar la documentación de la arquitectura como un elemento de soporte durante la presentación verbal de esta a los diferentes involucrados en el sistema.

4.3.2 Preservar información sobre la arquitectura

La documentación es persistente por naturaleza. De esta forma, la existencia de documentación sobre la arquitectura, y el acceso a aquella misma, podría ser de utilidad para resolver varias situaciones que pueden presentarse durante el desarrollo de un sistema.

Por ejemplo, si nuevos miembros se unen al equipo de desarrollo, pueden usar la documentación para comprender de forma más rápida la arquitectura del sistema. Esto es positivo y adecuado, comparado con el escenario de depender de la disponibilidad del arquitecto para conocer de su propia voz toda la información al respecto.

De forma similar, la existencia de documentación permite que cuando personas clave involucradas en el diseño de la arquitectura se ausentan, el resto del equipo de desarrollo puede consultarla para obtener la información requerida sin tener que esperar a que estas personas regresen.

Otro ejemplo es que las actividades de mantenimiento o actualización a un sistema pueden llevarse a cabo de manera más localizada si existe un documento que describe la arquitectura de este. Este ejemplo es más valioso si pensamos en que el sistema en turno es uno legado, el cual fue diseñado y desarrollado por personal que ya no labora en la compañía.

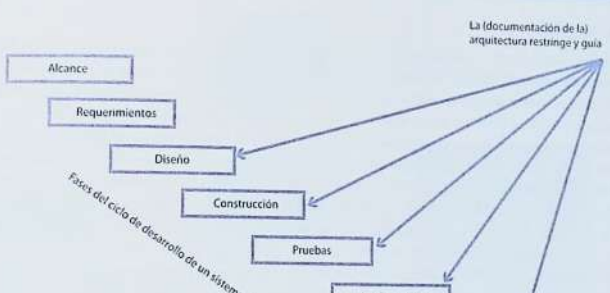
Todos los cambios realizados a la arquitectura deberían ser registrados en la documentación. De esta forma, estos pueden preservarse y ser conocidos por aquellos que tienen acceso a la documentación de la arquitectura.

4.3.3 Guiar la generación de artefactos para otras fases del desarrollo

En los capítulos anteriores expusimos que la *arquitectura de software* representa la estructura con la cual se proporcionará lo que el cliente o usuario espera del sistema. Por tal razón la arquitectura es un artefacto muy importante para guiar otras actividades dentro del ciclo de desarrollo de sistemas. Aunque todas ellas se llevan a cabo por personas que pertenecen a diferentes grupos de interesados en el sistema, por ejemplo, los líderes de proyecto o quienes desarrollan o prueban el sistema, su realización requiere que todas estas personas tengan una visión compartida y se desenvuelvan en un contexto de trabajo común atendiendo a esta. Como se ilustra en la figura 4-2, la documentación de la arquitectura es este contexto de trabajo común que permite guiar y restringir la forma en que se llevan a cabo varias actividades durante determinadas fases del ciclo de desarrollo del sistema.

Por ejemplo, al contener información acerca de todas las estructuras que conforman la arquitectura, la documentación arquitectónica sirve como fuente para las personas encargadas de la planeación del desarrollo del sistema. De forma similar, al contener información sobre el particionamiento y la funcionalidad que albergan los elementos que conforman las estructuras, así como de las relaciones de estos elementos en la arquitectura, la documentación arquitectónica es útil como guía para las actividades relacionadas con la construcción y la prueba del sistema.

En la sección 4.4 explicaremos que además de los de *software*, los elementos que conforman las estructuras arquitectónicas pueden ser también de *hardware*; por ejemplo los equipos de cómputo en donde se instalan los elementos de *software*. Para promover algún *driver*, durante el diseño el arquitecto pudo haber decidido colocar los equipos de cómputo en ubicaciones geográficamente dispersas. Al contener esta información, la documen-



› Figura 4-2 La arquitectura guía y restringe determinadas actividades durante algunas fases del ciclo de desarrollo del sistema

tación arquitectónica permite guiar las actividades de las personas encargadas de la implantación o el mantenimiento de sistema.

Ya hemos mencionado que en métodos como el ACDM (Lattanze, 2008) la documentación de la arquitectura es un documento indispensable para llevar a cabo de manera efectiva las actividades de evaluación. En el capítulo 5 describiremos con mayor detalle la etapa de evaluación de la arquitectura y observaremos que la identificación de problemas depende casi totalmente de la existencia de una buena documentación de la arquitectura.

4.3.4 Proveer un lenguaje común entre diversos interesados en el sistema

La documentación describe las estructuras arquitectónicas con el propósito de que la información relacionada pueda ser comunicada eficientemente a los diversos interesados en el sistema. Alrededor de esta idea ya hemos señalado cómo el documentar la arquitectura ayuda a disminuir algunas limitaciones de la comunicación verbal, a minimizar en algunos casos la necesidad del usar comunicación verbal para describir la arquitectura, o bien, a guiar actividades diversas durante el desarrollo de un sistema. Sin embargo, es importante mencionar una característica muy importante que permite que estos aspectos positivos ocurran: la documentación de la arquitectura debe ser elaborada utilizando una notación o lenguaje que tenga sintaxis y semántica bien definida; esto es, una que al ser observada o utilizada por los diversos interesados en el sistema se interprete de la misma forma por todos ellos. Cuando la documentación de la arquitectura está elaborada usando una notación adecuada, esta notación y las estructuras representadas por medio suyo se convierten en un lenguaje común entre los interesados.

En la sección 4.5 proveemos más detalles sobre este punto al describir los diversos tipos de notaciones que se pueden utilizar para elaborar la documentación de la arquitectura.

UNIVERSIDAD TÉCNICA DEL NORTE

4.4 VISTAS

Ya hemos explicado que la documentación de la arquitectura describe sus estructuras con el propósito que la información sobre estas pueda ser comunicada eficientemente a los diversos interesados en el sistema. Dichas estructuras se conforman por diversos elementos que, de acuerdo con la perspectiva desde la cual son descritos, adquieren una identidad particular. Por ejemplo, los elementos que adquieren la forma de unidades de implementación son utilizados por lo habitual cuando se hace una descripción sobre cómo está implementada la arquitectura. Sin embargo, estos mismos elementos pueden convertirse en una o más instancias al hacer una descripción sobre lo que ocurre cuando la arquitectura se ejecuta. De forma similar, cuando se elabora una descripción sobre cómo y dónde se encuentra instalada la arquitectura, las estructuras que la conforman pueden incluir elementos de *hardware* o incluso algunos que representan ubicaciones geográficas.

De esta forma, la heterogeneidad de las estructuras y sus elementos hace que resulte muy complicado describirlos a todos en un solo documento. Además, y debido a la diversidad de los interesados en el desarrollo de un sistema, no todos los elementos, estructuras y perspectivas son relevantes para cada uno de ellos. De esta forma, para facilitar el manejo de toda la información, un concepto ampliamente utilizado al elaborar la documentación es el de *vista*.

> Una vista describe una o más estructuras de la arquitectura en términos de los elementos que la conforman.

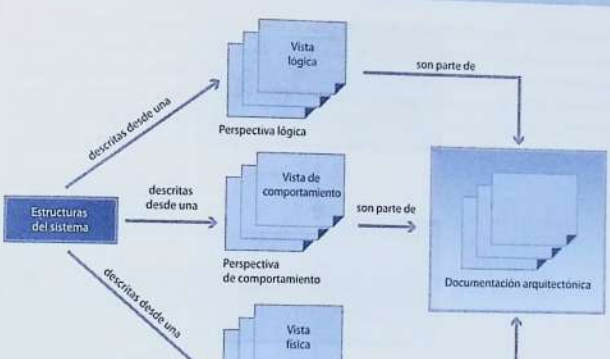
En términos generales una vista se conforma por: 1) un diagrama en donde se representan los elementos de la estructura, y 2) información textual que ayuda comprender este. Los diagramas permiten exhibir los elementos de la estructura de forma que es más fácil visualizarlos; la información textual describe por lo habitual las *propiedades y relaciones* definidas por elemento representado en el diagrama. Aunque en esta sección proveemos algunos ejemplos, en la sección 4.5 describiremos con mayor detalle las notaciones que pueden utilizarse para generar los diagramas y la información textual.

En el contexto de la etapa de documentación se reconocen varios tipos de vistas, por ejemplo, vistas de proceso, de desarrollo, de contexto o de implantación. De manera independiente al número total de vistas, es recomendable que el arquitecto documente al menos una de las siguientes tres clases:

1. Vistas lógicas.
2. Vistas de comportamiento.
3. Vistas físicas.

Como se observa en la figura 4-3, estas vistas toman sus nombres de las perspectivas más comunes desde las cuales un sistema puede ser descrito. Generar vistas dentro de estas tres clases es muy importante para comunicar eficientemente los aspectos clave de la arquitectura a los distintos interesados en el sistema. Si bien es cierto que cada vista describe aspectos específicos del sistema, es necesario mantener un mapeo entre la información que cada una de las vistas existentes contiene. De tal forma, usar el mismo nombre para referirse a un elemento de la arquitectura que aparece en varias vistas es una práctica indispensable. En la sección 4.7 se describen esta y otras recomendaciones relacionadas a la elaboración de vistas.

En las siguientes secciones describiremos algunas vistas tomando como ejemplo el caso referente al sistema de venta de boletos de autobús del que hemos hablado en el libro. Sin embargo, la sección 4 del apéndice contiene la documentación completa para el diseño presentado en el apéndice en la sección 3.



› **Figura 4-3.** Perspectivas, vistas y documentación arquitectónica. Figura adaptada de (Lattanze, 2008)

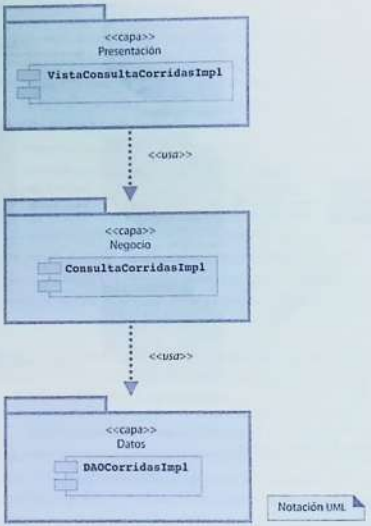
4.4.1 Vistas lógicas

Estas vistas describen las estructuras arquitectónicas en términos de elementos que toman la forma de unidades de implementación considerando tanto las propiedades como las relaciones u organización de cada una de estas. Las propiedades incluyen aspectos como, por ejemplo, su nombre, las funcionalidades o responsabilidades que le han sido asignadas, las interfaces que define o el lenguaje de programación utilizado para su implementación. La información sobre las relaciones u organización de las unidades se describen aspectos como, por ejemplo, qué funcionalidades o responsabilidades *ofrece a* o *espera* de otras unidades, o bien, cómo se agrupan en ensambles o jerarquías.

Algunas de las unidades de implementación que se utilizan frecuentemente en este tipo de vistas incluyen *clases*, *paquetes*, *módulos* o *subsistemas*. Asimismo, las relaciones o formas de organización utilizadas con frecuencia incluyen *es-parte-de* (que se emplea para representar una relación de agregación o composición), *depende-de* (la cual se usa para indicar que la unidad requiere de otra para funcionar correctamente), y *es-un* (que se utiliza para indicar que la unidad es "hija" de otra unidad la cual funge como "padre" y, por lo tanto, la unidad "hija" posee algunas de sus características).

La figura 4-4 muestra un ejemplo de vista lógica. Esta describe parte de la estructuración general del sistema de venta de boletos de autobús —usado como caso de estudio en este libro— en términos de capas que denotan distintas responsabilidades del sistema. La vista muestra también algunos de los módulos en estas capas, específicamente los que soportan la funcionalidad de consultar corridas. La relación <<*usa*>> se utiliza para indicar el tipo de vínculo entre los elementos de las vistas; <<*usa*>> es el mecanismo para indicar una relación *depende-de* en UML[1].

[1] *Unified Modeling Language*, por sus siglas en inglés.



© Humberto Cervantes Maceda, Perla Velasco-Elizondo, Raúl Cardoso Cariaga

› Figura 4-4. Ejemplo de una vista lógica

4.4.2 Vistas de comportamiento

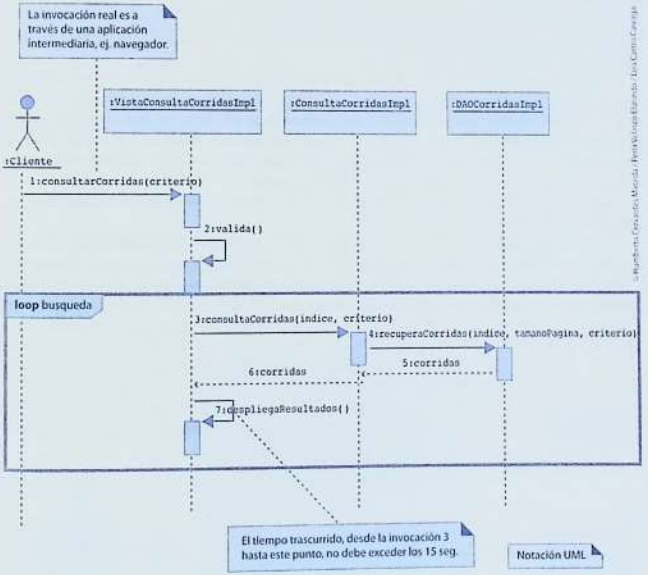
Las vistas de comportamiento describen estructuras cuyos elementos denotan entidades visibles en tiempo de ejecución, por ejemplo, *instancias, procesos, objetos, clientes, servidores o almacenes de datos*.

Además del nombre, el tipo y la funcionalidad ofrecidos, las propiedades de los elementos en este tipo de vistas incluyen por lo habitual información sobre valores específicos de atributos de calidad que pueden observarse cuando se ejecuta el sistema, por ejemplo, confiabilidad, desempeño o seguridad. Por otra parte, las relaciones describen la naturaleza de los mecanismos o protocolos de comunicación utilizados para la comunicación entre los elementos de la vista. De esta forma, *flujo-de-datos, llamada-a-procedimiento-remoto, acceso-compartido, broadcasting*, o incluso *SOAP*, son ejemplos válidos de relaciones que pueden emplearse durante la documentación de vistas de comportamiento (Clements, et al., 2010).

Es importante mencionar que en algunos enfoques de diseño y documentación, como en los descritos en Clements et al. (2010), las relaciones en este tipo de vistas se denotan como conectores.[2] Cuando este es el caso, el arquitecto puede definir y especificar propiedades para estos de la misma manera que se hace para elementos como procesos, clientes o servidores.

[2] En términos generales, un conector es un elemento arquitectónico que se usa para representar algún tipo de interacción entre componentes.

La figura 4-5 muestra un ejemplo de vista de comportamiento que describe aspectos relacionados con la funcionalidad para consultar corridas en nuestro ejemplo del sistema de venta de boletos de autobús. Como se puede observar, los elementos en esta vista son objetos, y las relaciones entre ellos son llamadas síncronas a procedimientos. Haciendo uso de comentarios se comunica que, mediante el uso de la táctica de paginación, el tiempo transcurrido desde que el cliente realiza una consulta hasta que se despliegan los resultados no debe exceder 15 segundos.



▸ Figura 4-5. Ejemplo de una vista de comportamiento

4.4.3 Vistas físicas

Estas vistas describen estructuras conformadas por elementos "físicos" que mantienen algún tipo de relación con los de las estructuras documentadas en otras vistas. Ya hemos mencionado que al documentar una arquitectura es relevante también saber, por ejemplo, cuáles equipos de cómputo albergan las unidades de implementación, cuáles ejecutan las instancias generadas a partir de ellas, y dónde se localizan físicamente estos. Cuando se documenta qué equipos las albergan, es posible que en uno solo estén más de una unidad de implementación. Por

esta razón es también importante documentar las localizaciones exactas de las unidades de implementación en las unidades de almacenamiento de los equipos de cómputo.

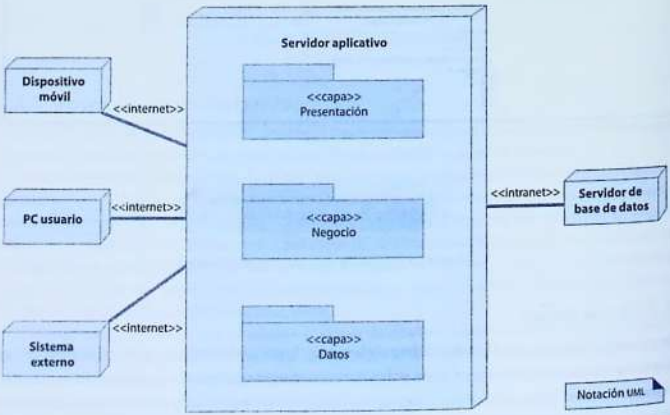
Con base en lo anterior podemos decir que en esta vista se pueden utilizar elementos de dos categorías: de *hardware* y de *software*.

Los equipos de cómputo y dispositivos como sensores o alarmas son quizá los elementos más comunes a considerar en la categoría elementos de *hardware*. Algunas de las propiedades que podrían considerarse al documentar algunos de ellos incluyen valores específicos acerca de información como modelo, marca, tipo de procesador, cantidad de memoria o capacidad de almacenamiento. En la mayoría de los casos, los valores que toman estas propiedades promueven la satisfacción de *drivers arquitectónicos*. Por ejemplo, un tiempo de respuesta de *x* segundos, requerido para la ejecución de determinado proceso, puede ser promovido mediante el uso de un equipo de cómputo con un procesador tipo *y*.

Respecto de los elementos de *software*, ejemplos válidos en estas vistas incluyen unidades de implementación como clases, paquetes, módulos o subsistemas, unidades visibles a tiempo de ejecución como instancias, procesos, objetos, clientes o servidores, o bien, almacenes de datos, así como estructuras de directorios. Para estos elementos se podrían o no definir propiedades. Cuando es el caso, estas denotan aspectos y valores relevantes en el contexto de implantación o producción del sistema, por ejemplo, permisos de acceso, sistema operativo, cantidad de memoria, almacenamiento o ancho de banda requerido.

Dada la naturaleza de estos elementos en las vistas físicas, relaciones como *reside-en* o *se-ejecuta-en*, son utilizadas con frecuencia.

La figura 4-6 ejemplifica una vista física en la cual se indica en qué equipos residen las capas resultantes de la estructuración general del sistema de ventas de boletos de autobús. La vista muestra también otros elementos físicos relevantes para el sistema y, haciendo uso de estereotipos en UML, la naturaleza de los canales de comunicación que permiten la interacción de todos estos elementos.



› Figura 4-6 Ejemplo de una vista física

Es importante mencionar que las vistas físicas pueden emplearse también para documentar aspectos de asignación de trabajo, por ejemplo, para indicar elementos de las vistas lógicas a las personas del equipo de desarrollo que serán responsables de su implementación.

4.5 NOTACIONES

En la sección 4.3.4 mencionamos que un aspecto fundamental para que la documentación cumpla el objetivo de comunicar de manera eficiente información sobre las estructuras es el uso de notaciones con una sintaxis y semántica que minimice, o idealmente erradique, la posibilidad de ambigüedad en su interpretación. Se reconoce por ello que las notaciones formales funcionan mejor, pues usan un lenguaje preciso, esto es, uno con sintaxis y semántica bien definidas, por lo habitual basadas en algún fundamento matemático. Sin embargo, no siempre es necesario ni práctico emplear este tipo de notaciones, y por ello existen otras que podrían ser también adecuadas para documentar la *arquitectura de software*.

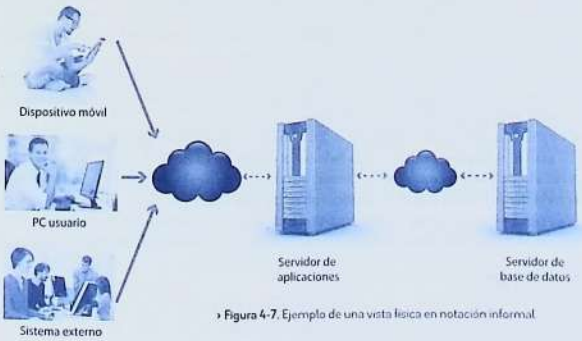
En las siguientes secciones describiremos algunas notaciones que, de acuerdo con nivel de formalidad, pueden clasificarse en:

1 Notaciones informales.
2 Notaciones semiformales.
3 Notaciones formales.

4.5.1 Notaciones informales

Como su nombre lo sugiere, estas notaciones son todas aquellas que no utilizan un lenguaje preciso. Esto es, ni su sintaxis ni su semántica son formales y, por lo tanto, pueden resultar inadecuadas si no se utilizan atendiendo determinadas recomendaciones. Por ejemplo, incluir siempre una descripción en lenguaje natural que explique el significado de cada elemento de la notación, utilizar los mismos elementos de la notación para representar lo mismo en todas las vistas. En la sección 4.7 hablaremos de otras recomendaciones para reducir la ambiguedad cuando se utiliza una notación informal para documentar la arquitectura.

La figura 4-7 muestra un ejemplo de vista física elaborada usando una notación informal. Como lo hemos mencionado, si no se incluye una descripción sobre el significado de cada elemento que aparece en la vista, su interpretación puede ser equivocada. Por ejemplo, no es claro por qué se usan diferentes tipos de líneas para



› **Figura 4-7.** Ejemplo de una vista física en notación informal.

conectar los elementos ni tampoco por qué unas son undireccionales y otras bidireccionales. Tampoco es aclarado el significado de las dos nubes.

4.5.2 Notaciones semiformales

En sentido estricto, las notaciones semiformales tampoco están basadas en un lenguaje preciso. La noción de "semiformal" describe más bien la situación de que utilizan una sintaxis y semántica "aceptada ampliamente" dentro de un dominio de aplicación para la cual se tiene. Ejemplos de estas notaciones incluyen el lenguaje de modelado unificado (UML) o la notación entidad-relación.

Es importante mencionar que estas notaciones no han sido creadas de manera exclusiva para documentar *arquitecturas de software*, por lo que en algunos casos resultan limitadas para expresar aspectos específicos en este contexto. A partir de la versión 2 de UML se han realizado algunas adiciones para facilitar el modelado arquitectónico. Sin embargo, siguen siendo limitadas en particular para documentar vistas dinámicas. Por esta razón, en Clements *et al.* (2010) se proveen algunas guías para el uso de UML 2.x en la documentación de arquitectura.

Al igual que las notaciones informales, las semiformales deben ser utilizadas con cuidado para evitar ambigüedad en su interpretación.

Las figuras 4.4 a la 4.7, presentadas en secciones anteriores, muestran ejemplos de vistas de arquitectura en UML.

4.5.3 Notaciones formales

Como lo mencionamos antes, las notaciones formales utilizan un conjunto de conceptos arquitectónicos, los cuales se describen usando un lenguaje con fundamento matemático, como algún tipo de lógica o de álgebra. Tales conceptos en estos lenguajes son por lo habitual: componente –el cual representa un elemento que realiza procesamiento o almacenamiento de datos–; conector –con el que se hace representación de canales de flujo de datos o de control entre componentes–, y configuración –el cual representa la forma particular en la que los componentes y conectores se relacionan en el sistema que se describe.

A pesar de no ser muy utilizadas en la práctica, en la academia existen muchas notaciones formales de las cuales se puede hacer uso para documentar la arquitectura. La mayoría de ellas son reconocidas como lenguajes de descripción de arquitecturas, o ADL,[3] por sus siglas en inglés. Al ser notaciones creadas con el fin de documentar arquitecturas, estas notaciones no presentan las limitaciones de las informales y semiformales. Además, debido a su tipo de fundamento matemático, estas notaciones pueden analizarse mediante el uso de métodos formales. Por ejemplo, al utilizar los valores provistos para propiedades, como tiempo de ejecución, periodicidad y prioridad de un conjunto de procesos, se podrían usar métodos de planificación de tareas para determinar si pueden ejecutarse de forma concurrente dentro de un lapso de tiempo determinado.

Algunos de los ADL más conocidos son *Acme*[4], creado en la Universidad Carnegie Mellon; xADL[5], creado tanto por miembros de esta misma universidad como el SEI, la Armada de los Estados Unidos y *Honeywell Technology Center*, y xADL[6] creado en la Universidad de California Irvine.

La figura 4-8 muestra un pequeño extracto de la documentación de una vista lógica del sistema de nuestro caso de estudio usando el lenguaje de descripción de arquitectura Acme. La vista incluye información sobre dos componentes (`VistaConsultaCorridas` y `ConsultaCorridasImpl`) y un conector (*rmi*). Los componentes y conectores tienen puertos y roles, denotados con las palabras `Port` y `Role`, los cuales funcionan como interfaces que les permiten la interacción. La forma particular en la que los componentes y conectores se relacionan en este sistema, es decir, la configuración, está especificada en el bloque de información dentro de la palabra

[3] *Architecture Description Language*.
[4] http://www.cs.cmu.edu/~acme/
[5] http://www.aadl.info/
[6] http://isr.uci.edu/projects/xarchuci/index.html

`Attachments`. La documentación muestra también la definición de algunas propiedades, denotadas con la palabra `Property`, relevantes para los componentes descritos; por ejemplo `conexionesMax`, para especificar el número de conexiones concurrentes que puede soportar el componente, y `latenciaMax`, para especificar el tiempo máximo de procesamiento de una solicitud.

```
System RedADM = {
    ...

    Component VistaConsultaCorridas = {
        Port p_envia;
        Property codigoFuente: externalFile = "consulta.jsp";

    };

     Component ConsultaCorridasImpl  = {
        Port p_recibe;
        Property conexionesMax: integer=100;
        Property latenciaMax: integer= 2;
        Property codigoFuente: externalFile = "consulta.java";
     };

    Connector rmi = {
        Role r_cliente;
        Role r_cliente
    };

    Attachments {
        cliente.envia to rmi.r_cliente;
        servidor.recibe to rmi.r_cliente;
    }
        ...

}
```

› Figura 4-8. Ejemplo de vista en notación formal

© Humberto Cervantes Maceda / Perla Velasco Elizondo / Ivan Castro Careaga

4.6 MÉTODOS Y MARCOS CONCEPTUALES DE DOCUMENTACIÓN DE ARQUITECTURA

Hasta este momento nos hemos limitado a describir la etapa de documentación en términos de elaborar vistas utilizando algún tipo de notación. Sin embargo, poco hemos dicho sobre cómo decidir de manera más sistemática cuáles vistas y de qué modo elaborarlas. En el soporte existente en este rubro es posible distinguir entre métodos y marcos conceptuales para la documentación de arquitecturas. Mientras los métodos describen de manera más explícita tanto las entradas requeridas como la secuencia de acciones que comprenden y las salidas generadas, los marcos conceptuales proveen un conjunto de conceptos que deben considerarse al documentar la arquitectura. Los métodos y marcos que presentamos abajo existen de forma independiente o están incrustados en un método específico de desarrollo de arquitectura.

En este capítulo describimos los siguientes métodos o marcos conceptuales:

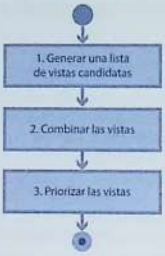
1. Vistas y más allá (*Views* and *Beyond*).
2. 4+1 Vistas (4+1 *Views*).
3. Método de diseño centrado en la arquitectura (ACDM).
4. Puntos de vista y perspectivas (*Viewpoints and Perspectives*).

4.6.1 Vistas y más allá

Vistas y más allá (Clements et al., 2010), o *Views and Beyond*, como expresa su nombre en inglés, es un método propuesto por el SEI para soportar la documentación de la *arquitectura de software* de un sistema. Considera tres categorías de vistas arquitectónicas relevantes, recomendaciones sobre qué documentar de cada una de ellas, así como una considerable cantidad de información adicional que puede complementar la documentación de la arquitectura.

Las categorías de vistas consideradas en Vistas y más allá son: vistas de módulos, vistas de componentes y conectores (C&C[1]), y vistas de asignación. En términos generales, todas estas son análogas a las vistas lógicas, de comportamiento y físicas descritas en la sección 4.4. La información adicional considerada en este método corresponde a la del sistema, la cual es relevante para todas la vistas, por ejemplo, descripción del sistema, relación entre vistas, glosarios, restricciones de negocio y *drivers arquitectónicos*.

Para apoyar al arquitecto en la tarea de documentar el diseño de la arquitectura, Vistas y más allá define un proceso secuencial simple en términos de las etapas que se ilustran en la figura 4-9 y se describen a continuación:



› **Figura 4-9.** Etapas del método vistas y más allá

© Humberto Cervantes Maceda - Profa. Velocia Elizondo - Tany Castro Cerezo

1. **Generar una lista de vistas candidatas.** Utilizando una tabla que en sus filas incluya el conjunto de interesados en el sistema, y en sus columnas, los diferentes tipos de vistas, el arquitecto identifica las vistas relevantes para el diseño en turno. De acuerdo con cada vista, el arquitecto debe indicar el nivel de detalle con que estas requieren ser documentadas.
2. **Combinar las vistas.** A efecto de reducir el número de vistas a documentar, el arquitecto lleva a cabo en esta etapa un análisis acerca de la tabla generada en el paso anterior. Podría detectar casos en los que, por ejemplo, algunas vistas requieren ser documentadas de forma muy general y que las necesitan muy pocas personas. Podría también detectar que existen vistas utilizadas para satisfacer los requerimientos de información de más de un interesado en el sistema, y que la combinación de ellas en una sola no implica mucha complejidad sino, por el contrario, ayuda a satisfacer los de todos estos interesados.
3. **Priorizar las vistas.** Este paso se lleva a cabo para decidir cuáles vistas, de la lista final, deben elaborarse primero. La priorización se realiza considerando los requerimientos de información para continuar con el diseño detallado del sistema o su implementación.

[1] *Component-and-Connector Views*.

Vistas y más allá no hace especificación explícita del tipo de notación a utilizar, pero sí énfasis especial en hacer uso de una que minimice la ambiguedad en su interpretación. Para elaborar las vistas en cada una de las categorías, el método sugiere utilizar un conjunto de estilos arquitectónicos; por ejemplo, descomposición, uso o capas para las vistas de módulos. Se recomienda consultar Clements *et al.* (2010) para mayores detalles sobre los estilos en cada categoría de vistas.

El método también sugiere utilizar una plantilla, con siete partes fundamentales, para organizar las vistas y otra información generadas. La estructura de la plantilla se ilustra en la figura 4-10. Como puede observarse, la estructura de la plantilla incluye:

1. **Representación primaria de la vista.** Es una representación gráfica de la vista en términos de sus principales elementos y relaciones, así como el texto correspondiente para describir de forma general la información anterior.
2. **Catálogo de elementos.** Esta sección hace las veces de un catálogo que contiene los elementos de la representación primaria describiéndolos con detalle. La información explícita acerca de estos incluye, propiedades, interfaces o comportamiento.
3. **Diagrama de contexto.** Permite ubicar al lector de la documentación sobre cuál parte del sistema es la que se describe en la vista.
4. **Guía de variabilidad.** Describe los posibles puntos de variación permisibles, si es que existen, de los elementos y relaciones representados en la vista. Ejemplos de estas variaciones incluyen rangos en los valores de las propiedades de los componentes, o variaciones en los protocolos de comunicación soportados por ellos.
5. **Antecedentes de la arquitectura.** Explica las razones que soportan las decisiones de diseño tomadas por el arquitecto durante el diseño de elementos y relaciones representadas en la vista.
6. **Otra información.** En esta sección se describe información relevante extra que aplique a la vista. Algunos ejemplos de ella: reglas de negocio, información sobre el equipo de desarrollo o descripción del sistema.

En el contexto del caso de estudio del libro, en la sección 4.4 del apéndice se puede encontrar un ejemplo de documentación de vista, con el uso del método de Vistas y más allá.

1. Representación primaria
2. Catálogo de elementos
   2.1 Elementos y sus propiedades
   2.2 Relaciones y sus propiedades
   2.3 Interfaces
   2.4 Comportamiento
3. Diagrama de contexto
4. Guía de variabilidad
5. Antecedentes de la arquitectura
   5.1 Justificación de las decisiones de diseño
   5.2 Análisis de resultados
   5.3 Suposiciones
6. Otra información

› **Figura 4-10** Plantilla de documentación sugerida en Vistas y más allá

© Pearson Educación, Marcela Perez Mejia, Ernesto Jiménez Cerrato

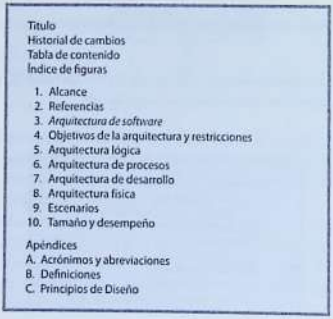
4.6.2 4+1 Vistas

4+1 Vistas es un marco conceptual para la documentación de arquitectura propuesto por Philippe Kruchten (Kruchten, 1995). Como su nombre lo indica, este marco considera la noción de vista como concepto principal; recomienda de modo específico la elaboración cinco vistas:

1. **Vista lógica.** Describe la arquitectura desde la perspectiva de los usuarios finales, es decir, considerando los elementos principales que proveen a estos los servicios del sistema.
2. **Vista de procesos.** Explica la arquitectura haciendo mayor énfasis en los elementos que en tiempo de ejecución permiten soportar aspectos como concurrencia, rendimiento o escalabilidad.
3. **Vista de desarrollo.** Describe la arquitectura en términos de elementos relevantes para los desarrolladores del sistema.
4. **Vista física.** Explica los componentes físicos del sistema (es análoga a las vistas físicas descritas en la sección 4.4.3)
5. **Vista "+1"** tiene la función de relacionar los elementos en las otras cuatro vistas por medio de casos de uso o escenarios que ilustren cómo interactúan todos estos elementos.

Cada una de estas vistas puede ser elaborada usando una plantilla, y es posible representar sus estructuras usando algún estilo o patrón arquitectónico.

Aunque 4+1 Vistas no especifica explícitamente el tipo de notación a utilizar, sí sugiere organizar la documentación arquitectónica en dos documentos. El primero, llamado Documento de arquitectura, contiene principalmente las vistas anteriormente descritas. Su estructura está representada en la figura 4-11. El segundo documento, denominado Guía sobre el diseño de la arquitectura, contiene información para entender el porqué de las decisiones de diseño tomadas por el arquitecto.



› Figura 4-11. Estructura del Documento de arquitectura sugerida en 4+1 Vistas

© Humberto Cervantes Maceda - Perla Velasco Elizondo - Raúl Cortés Camargo

4.6.3 Puntos de vista y perspectivas

Como explicamos en el capítulo anterior, Puntos de vista y perspectivas (Rozanski y Woods, 2005) es un método de diseño de arquitectura. De esta forma, lo que describiremos en esta sección es el enfoque de documentación que este método considera.

Recordando, Puntos de vista y perspectivas plantea que la arquitectura debe estructurarse considerando un conjunto de vistas, las cuales tienen que ser elaboradas tomando en cuenta la información definida por *un punto de vista*. De manera adicional, el método hace hincapié en la noción de *perspectiva arquitectónica* que denota un conjunto de actividades, tácticas, riesgos y aspectos de importancia para enriquecer una vista respecto de cómo esta soporta un conjunto de atributos de calidad.

Cada punto de vista está definido en términos de una descripción, aspectos de interés (por ejemplo sincronización o latencia), modelos notacionales a utilizar (por ejemplo máquinas de estado, de flujo de datos o diagramas), posibles problemas durante su elaboración (por ejemplo abrazo mortal o ambiguedad), audiencia (por ejemplo desarrolladores o administradores del sistema) y aplicabilidad.

El método define siete puntos de vista, y estos indican a su vez el tipo y el número de vistas a documentar:

1. **Funcional:** referente a vistas que describen elementos funcionales del sistema.
2. **Información:** hace alusión a vistas que explican cómo los elementos almacenan, manipulan, administran y distribuyen la información del sistema.
3. **Concurrencia:** hace referencia a vistas que describen la manera en que los elementos de la arquitectura soportan aspectos relacionados con la concurrencia.
4. **Desarrollo:** referente a vistas que especifican cómo los elementos deben ser implementados, probados, o modificados.
5. **Implantación:** alude a vistas que señalan cómo los elementos deben ser implantados en su ambiente de producción.
6. **Operacional:** hace referencia a vistas que indican el modo en que los elementos deben utilizarse en su ambiente de producción.
7. **Contexto:** alude a vistas que describen las dependencias, relaciones e interacciones de los elementos de la arquitectura y su ambiente.

En contraste con los métodos anteriores, este no sugiere de manera explícita alguna estructura para organizar la documentación generada. Sin embargo, hay determinado énfasis respecto del uso en UML como notación preferida.

4.6.4 ACDM (etapas 3 y 4)

ACDM (Lattanze, 2008) es un método de diseño de *arquitectura de software*; en los capítulos 2 y 3 hemos descrito partes de él. Durante la etapa 3, Creación o refinamiento de la arquitectura, se define el diseño inicial o se refina la versión actual de este en relación con los resultados obtenidos en la etapa 4, Revisión de la arquitectura (véase la figura 4-12).

Ya mencionamos también que, en el ACDM, el diseño y la documentación no son etapas mutuamente excluyentes. Así, una parte de las actividades de la etapa 3 incluye la creación o refinamiento de la arquitectura y la creación o actualización de la documentación correspondiente. Estas actividades consideran la participación de los siguientes roles: administrador del proyecto, arquitecto de *software* líder, líder científico, ingeniero de requerimientos, ingeniero de calidad, ingeniero de soporte e ingeniero de producción.

ACDM considera las vistas estáticas, dinámicas y físicas que, en términos generales, son análogas a las lógicas, de comportamiento y físicas, descritas en la sección 4.4. Contempla además la noción de vistas mixtas utilizadas para documentar información necesaria para, por ejemplo, comunicar la asignación de elementos de vista dinámica a elementos de la vista física o a los equipos responsables de su implementación, o bien, comunicar la relación entre los *drivers arquitectónicos* y los elementos que los soportan en las diferentes vistas.

Para la selección de las vistas, el ACDM sugiere adoptar una estrategia similar a la de Vistas y más allá. Esto es, identificar a los interesados en el sistema y, considerando los tipos de vistas, identificar para cada uno de estos las vistas que les son relevantes, así como el nivel de detalle con que requieren ser documentadas.

© Humberto Cervantes Maceda / Perla Velasco Elizondo / Ivan Castro Careaga

› **Figura 4-12**. Etapas del ACDM en donde se elabora la documentación de la arquitectura

Cabe señalar que el ACDM sugiere organizar la documentación de acuerdo con la estructura presentada en la figura 4-13. ACDM no hace énfasis en el uso de una notación específica.

1. Portada
2. Descripción del documento
3. Descripción de proyecto
4. *Drivers arquitectónicos*
5. Contexto del sistema
6. Nivel *n* de descomposición del sistema
   - 6.1 Descripción
   - 6.2 Representación primaria
   - 6.3 Razones de la descomposición
   - 6.4 Catálogos de elementos y relaciones
   - 6.5 Interfaces
7. Mapeo entre perspectivas
8. Índices por palabras, glosarios y acrónimos

© Humberto Cervantes Maceda / Perla Velasco Elizondo / Ivan Castro Careaga

› **Figura 4-13**. Estructura del Documento de arquitectura sugerida en el ACDM.

4.6.5 Otros

Además de los métodos y marcos conceptuales descritos anteriormente, existen algunos otros recursos que el arquitecto puede utilizar durante la documentación de la arquitectura. A continuación los describimos de modo breve.

El estándar ISO/IEC/IEEE 42010:2011, Descripción de arquitecturas, provee un marco conceptual para describir arquitecturas. Da también una especificación de lo que debe incluir un documento para considerarlo adecuado a este estándar. Entre sus conceptos relevantes para este capítulo hemos de mencionar los de vista y puntos de vista. El marco conceptual de este estándar es utilizado en el método Puntos de vista y perspectivas, descrito en la sección 4.6.3.

El modelo 4 Vistas de Siemens (Hofmeister, Nord y Soni, 1999) se crea a partir de un estudio realizado sobre la práctica de actividades de *arquitectura del software* en la industria. Establece que la documentación de la arquitectura debe llevarse a cabo considerando cuatro vistas principales: conceptual, de módulos, de ejecución y de código.

Finalmente, *Rational* ADS es una adaptación del 4+1 Vistas, creada para soportar la documentación de arquitecturas complejas, como las empresariales, de *e-business* o de sistemas incrustados. Además de las cinco vistas consideradas en 4+1 Vistas, las cuales son renombradas, *Rational* ADS considera cuatro vistas adicionales. Las nueve vistas resultantes (requerimientos no funcionales, casos de uso, dominio, experiencia del usuario, lógica, procesos, implementación, implantación y prueba) se agrupan de acuerdo con cuatro puntos de vista: requerimientos, diseño, realización y verificación.

4.6.6 Comparación de los métodos y marcos conceptuales

Presentamos la siguiente tabla comparativa con el propósito de proveer al lector un resumen de las características de los métodos y marcos conceptuales descritos en las secciones anteriores.

	Vistas y más allá	4+1 Vistas	Puntos de vista y perspectivas	Método de diseño centrado en la arquitectura (ACDM)
Tipo	Método	Modelo	Marco conceptual	Método
Mecánica y enfoque	Definido como una secuencia de pasos que soporta la identificación de las vistas a documentar.	Puede ser utilizado dentro del contexto de un método de desarrollo para soportar la especificación y documentación de vistas arquitectónicas.	Modelo que puede ser utilizado en el contexto de un método de desarrollo para soportar la especificación y documentación de vistas arquitectónicas.	Método iterativo e incremental que soporta la identificación y documentación de vistas arquitectónicas.
Provee un proceso que especifica las actividades, roles y artefactos de entrada y salida	Sí	No aplica	No aplica	Sí
Participantes	Arquitecto de software.	No aplica	No aplica	Se asume la participación de los siguientes roles: administrador del proyecto, arquitecto de software, líder, líder científico, ingeniero de requerimientos, ingeniero de calidad, ingeniero de soporte e ingeniero de producción.

Continúa...

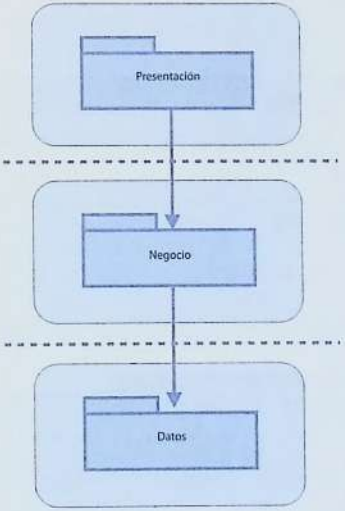
Continuación...

	Vistas y más allá	4+1 Vistas	Puntos de vista y perspectivas	Método de diseño centrado en la arquitectura (ACDM)
Tipo	Método	Modelo	Marco conceptual	Método
Entradas	Conjunto de interesados en el sistema e información proveniente de la etapa de diseño de la arquitectura.	No aplica	No aplica	*Drivers* de la arquitectura.
Salidas	Lista de vistas a documentar junto con el orden en que deben ser elaboradas.	No aplica	No aplica	Conjunto de vistas arquitectónicas documentadas.
Criterios de terminación	Tener un conjunto de vistas a documentar y el orden en que deben ser elaboradas.	No aplica	No aplica	Todas las vistas arquitectónicas están documentadas.
Considera las tres vistas fundamentales: lógicas, de comportamiento y físicas	Sí	Sí	Sí	Sí
Considera vistas adicionales	No	Sí	Sí	No
Considera información adicional del sistema que complementa la de las vistas (por ejemplo descripción del sistema, relación entre vistas, glosarios, restricciones de negocio y *drivers* arquitectónicos)	Sí	Sí	Sí	Sí
Provee soporte para documentar atributos no funcionales específicos	No	No	Sí	No
Da soporte para identificar o priorizar las vistas a documentar	Sí	No	No	Sí
Sugiere cómo organizar la documentación generada	Sí	Sí	No	Sí
Notación utilizada	No indica	No indica	Hace determinado énfasis en el uso en UML.	No indica

4.7 RECOMENDACIONES PARA ELABORAR LA DOCUMENTACIÓN

En la sección anterior describimos algunos métodos y marcos conceptuales útiles durante la documentación de la arquitectura. Un aspecto común en estos métodos es que se enfocan en el "qué" y ofrecen poca orientación sobre el "cómo". Esto es, se centran en indicar qué vistas se debe documentar y qué información de cada vista verter en documentos, pero ninguno explica con detalle cómo documentar adecuadamente las vistas.

Si bien, la selección de notaciones de documentación puede ofrecer determinado soporte en este sentido, a veces no resulta suficiente. Por ejemplo, la vista de la figura 4-14 se define usando elementos de notación de UML, pero sin la presencia del arquitecto difícilmente puede utilizarse para comunicar de modo eficiente la información que contiene. Algunas preguntas que surgen al observar esta vista son: ¿qué tipo de elementos representan los paquetes?, ¿qué clase de relaciones denotan las flechas que conectan los paquetes?, ¿qué elementos representan los cuadros debajo de los paquetes?, ¿qué significan las líneas punteadas horizontales?



› Figura 4-14. Ejemplo de vista de mala calidad

A efecto de evitar este y otro tipo de problemas, durante la etapa de documentación se recomienda atender los aspectos siguientes e incrementar así la calidad de la documentación:

1. Escribir la documentación desde la perspectiva de la persona que la va a utilizar. El tipo de lenguaje usado en los documentos requeridos por los desarrolladores del sistema no es el más adecuado para comunicar la arquitectura al propietario de este.
2. En caso de usar notaciones propietarias o informales, incluir un cuadro de notación en cada diagrama para evitar ambigüedad en su interpretación.
3. Usar un nivel adecuado de abstracción o detalle considerando el tipo usuario de la documentación. El tipo de información que necesitan los desarrolladores no es el mismo que el requerido por quienes instalarán el sistema una vez desarrollado.

© Humberto Cervantes Maceda, Pedro Velasco Elizondo, Iván Castro Enríquez.
UNIVERSIDAD TECNICA DEL NORTE BIBLIOTECA

4. Cuidar aspectos de presentación como gramática, ortografía y legibilidad de las representaciones gráficas.
5. Minimizar el uso de acrónimos.
6. Usar nombres descriptivos para los elementos, relaciones y propiedades usados en una vista.
7. Maximizar la consistencia de información entre elementos que aparecen en las diferentes vistas y los otros artefactos generados.
8. Capturar siempre el porqué de las decisiones de diseño relevantes en cada vista.
9. Realizar una evaluación de la documentación antes de hacerla disponible.
10. Actualizar de manera periódica la documentación.

EN RESUMEN

En este capítulo abordamos la tercera etapa del ciclo de desarrollo de la arquitectura: la documentación. Su objetivo es generar la descripción de los elementos que conforman las estructuras arquitectónicas para que esta pueda ser comunicada de manera eficiente a los diversos interesados en el sistema.

Explicamos que para facilitar tal descripción, un concepto utilizado ampliamente durante la elaboración de los documentos de la arquitectura es el de vista. Los tipos de vista que se emplean con frecuencia son:

1. Vistas lógicas.
2. Vistas de comportamiento.
3. Vistas físicas.

PREGUNTAS PARA ANÁLISIS

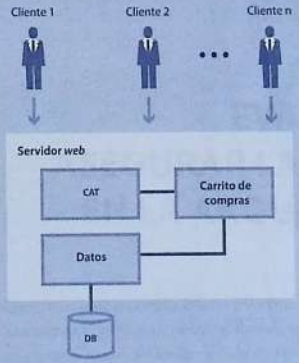
1. Hablando de documentación de arquitectura se reconocen varios tipos de vistas. Discuta con un(a) compañero(a) por qué se recomienda documentar siempre vistas lógicas, de comportamiento y físicas.
2. Discuta con un(a) compañero(a) las ventajas y desventajas de las notaciones informales respecto de las semiformales.
3. Discuta con un(a) compañero(a) en qué contextos son de mayor utilidad las notaciones formales de documentación de arquitectura.
4. ¿Considera que la arquitectura de un sistema puede ser documentada por completo usando una notación formal? Explique su respuesta.
5. En el contexto de ADL, ¿cuál es la diferencia entre una interfaz y un puerto?
6. Considere que una compañía de desarrollo de sistemas de *software* va a desarrollar un sistema de análisis automático de comentarios escritos en las redes sociales por los clientes de una cadena de restaurantes. Proponga una vista física para este sistema usando notación informal y discuta el porqué de los elementos, relaciones y propiedades en la vista.
7. Evalué la calidad de la vista generada en la pregunta anterior en relación con algunos de los aspectos listados en la sección 4.7. En específico los aspectos 2 a 6.

Continúa...

Continuación...

PREGUNTAS PARA ANÁLISIS

8. Considere que una compañía de sistemas va a desarrollar un sistema para una tienda de vinos, el cual permitirá a los clientes comprar utilizando un navegador *web*. Suponga que el arquitecto utiliza la vista mostrada a continuación para describir al equipo de desarrollo la arquitectura de este sistema. En su explicación, él menciona que "... el sistema se ejecuta en la parte superior de una base de datos denotada por DB, y además contiene un modelo de dominio denominado Datos, un catálogo de vinos denotado por CAT, y un carrito de compras que permite registrar las botellas que un cliente quiere adquirir".
   Discuta con un(a) compañero(a): ¿qué tipo de vista es esta? ¿Le parece que la vista es adecuada?



9. Evalúe la calidad de la vista presentada en la pregunta anterior respecto de algunos aspectos listados en la sección 4.7. En específico los aspectos 1 a 6.
10. Considere el sistema descrito en la pregunta 8. Suponga que este puede ser instalado y ejecutado en diferentes plataformas de *software* y *hardware*. ¿Cómo se podría representar ello en una vista?

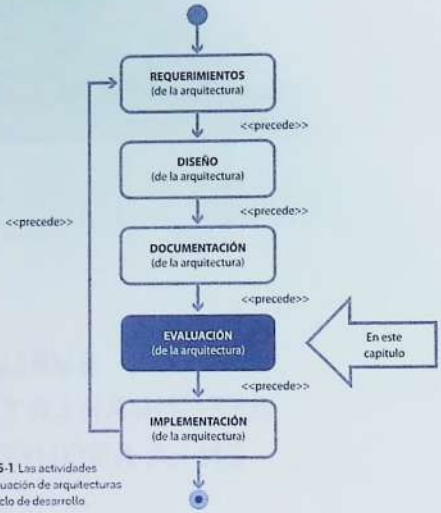
CAPÍTULO 5

EVALUACIÓN: ASEGURAR LA CALIDAD EN LA ARQUITECTURA

En los capítulos anteriores planteamos formas de especificar, diseñar y documentar *arquitecturas de software*. Una vez que se cuenta con una arquitectura especificada, diseñada y documentada, es posible iniciar la construcción del diseño del sistema a partir de la misma. Sin embargo, dada la importancia de las decisiones que incorpora, es conveniente asegurarse de que sea correcta y satisfaga además los *drivers arquitectónicos*.

La evaluación de arquitecturas permite la detección de incumplimientos y riesgos asociados, que pueden ser causas de retrasos o problemas muy serios en los proyectos a materializar. Este capítulo se enfoca en la etapa de evaluación del ciclo de desarrollo de la arquitectura, como muestra la figura 5-1.

Iniciaremos planteando de forma general el concepto de evaluación, para después enfocarnos en la práctica evaluativa de la arquitectura y los principios que ella conlleva. Terminaremos el capítulo con la descripción de algunos métodos usados en la actualidad para realizar las evaluaciones.



› **Figura 5-1** Las actividades de evaluación de arquitecturas en el ciclo de desarrollo

5.1 CONCEPTOS DE EVALUACIÓN

Molestia y frustración son las reacciones que se tienen cuando los sistemas fallan o no hacen lo que el cliente espera. La evaluación es una técnica con la que los desarrolladores cuentan para evitar que las fallas lleguen a los usuarios finales o se presenten en momentos en los que corregirlas es complicado y costoso. En la evaluación se examina un producto o subproducto para ver si cumple con criterios de calidad y qué tanto se desvía de ellos.

Las desviaciones se clasifican en dos tipos: desviaciones de las necesidades reales de los usuarios del producto, y desviaciones a la construcción correcta del producto. Las evaluaciones que buscan la detección de las primeras se conocen como *validaciones*, y se hacen por lo habitual sobre los productos terminados, aunque también pueden practicarse tanto a documentos de requerimientos o de diseño como a cualquier otro artefacto de trabajo. En las validaciones participan todos los interesados en el uso del producto.

Las evaluaciones que buscan desviaciones de la construcción correcta se conocen como *verificaciones* y se hacen por lo habitual sobre los artefactos intermedios que los proyectos generan al ir concretándose. En ellas participan casi de manera exclusiva los mismos miembros de los equipos de desarrollo.

Las verificaciones y validaciones se consideran como unas de las mejores prácticas en la *ingeniería de software* para la detección temprana de errores, defectos y/o riesgos en los distintos artefactos o partes de los sistemas en desarrollo. En un nivel más específico, tanto las revisiones e inspecciones, como la elaboración de prototipos o experimentos, implementan muchos de sus objetivos. En un nivel más detallado y para el contexto de las *arquitecturas de software*, las inspecciones se especializan en evaluaciones de arquitectura.

5.2 EVALUACIÓN DE ARQUITECTURAS

La arquitectura es la base para el diseño y la construcción de un producto de *software*. Por lo mismo, es importante asegurarse de que no se desvía de manera significativa de los *drivers arquitectónicos* (requerimientos funcionales, de atributos de calidad y de restricciones), así como tener confirmado que sea técnicamente correcta para que no se diseñen o construyan artefactos basados en arquitecturas erróneas o incompletas (Clements *et al*., 2008).

La evaluación consiste en revisar, inspeccionar o recorrer los artefactos de la *arquitectura de software* buscando inconsistencias, errores o desviaciones respecto de los *drivers*. Sin ella, la implementación del sistema incurre en riesgos, los cuales, en caso de que se concreten, tienen costos e impactos en el proyecto que pueden ser desproporcionados. Por ello, una estrategia inteligente es incluir una etapa de control de calidad de los artefactos de requerimientos y diseño de las *arquitecturas de software*.

La evaluación de las arquitecturas permite conocer si los requerimientos obtenidos fueron correctos y completos, o bien, si han cambiado, y qué tan satisfactorias fueron las decisiones tomadas durante el diseño de la arquitectura. Se aplica principalmente a los artefactos generados en las etapas de requerimientos y diseño de arquitectura que ya se documentaron de manera idónea, aunque puede hacerse durante su realización o cuando el sistema ya está terminado.

5.3 PRINCIPIOS DE LA EVALUACIÓN

Las evaluaciones tienen como principio ser procedimientos que detecten lo antes posible los errores para que no afecten fases posteriores o el uso del producto. Buscan además que los requerimientos de este (final o intermedio) sean satisfechos, señalando los riesgos en los productos desarrollados a efecto de realizar acciones de mitigación de aquellos (Clements et al., 2008).

Un defecto es cualquier elemento en un artefacto (documentación o código) que provoque una falla en el sistema una vez que esté en operación, o bien, se refiere a que el sistema no cumpla con un criterio de calidad específico. La realización de sistemas es una actividad intelectual muy compleja, lo cual, de manera natural, la hace propensa a que se cometan o introduzcan defectos en ella.

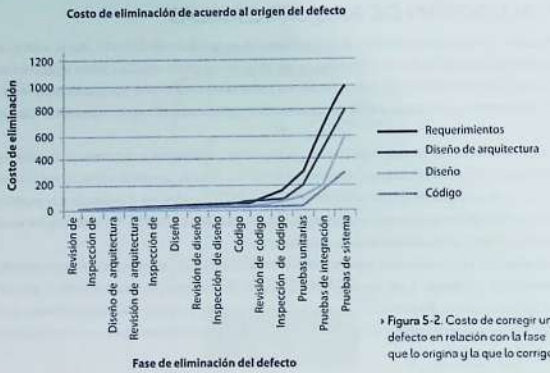
En el caso de las *arquitecturas de software*, los defectos se deben a elementos de un artefacto que provoquen una falla al sistema en operación, o bien, que este no cumpla con un *driver* arquitectónico específico.

5.3.1 Detección temprana de defectos en la arquitectura

El desarrollo de sistemas de *software* tiene una naturaleza incremental, de tal forma que si un producto intermedio contiene defectos, las siguientes actividades que lo usen como una entrada, se llevarán a cabo con una entrada incorrecta, generando productos intermedios erróneos y, por lo mismo, incrementando el número de defectos (se agregarán aumentos sobre bases erróneas). Tanto más tiempo pase entre la generación del defecto y su corrección, cuanto más serán los productos intermedios involucrados pues se da un efecto multiplicador.

Si el error es encontrado rápidamente, no hay productos adicionales por ajustar, solo el que lo contiene, por lo cual su corrección es menos compleja y, en consecuencia, más económica. La figura 5-2 incluye resultados de estudios de varios proyectos en donde se muestra el costo de eliminación de un defecto (en minutos), dependiendo de la fase en la que se le descubre y elimina.

Por su esencia, la arquitectura es la base para los diseños y la construcción del sistema. Encontrar defectos arquitectónicos cuando ya se tienen desarrollados varios componentes del sistema puede implicar costos serios de rehacer trabajo, así como retrasos en el proyecto.



› **Figura 5-2.** Costo de corregir un defecto en relación con la fase que lo origina y la que lo corrige.

© Humberto Cervantes Maceda, Pedro Velázquez Valenzuela, Luis Gerardo Cárdenas

5.3.2 Satisfacción de los *drivers arquitectónicos*

Las evaluaciones deben siempre enfocarse en la satisfacción de los *drivers arquitectónicos* (requerimientos de los usuarios y de otros grupos involucrados con el sistema) pues son la base de la calidad del producto. En los productos terminados, los usuarios son las personas que utilizarán el sistema. En los productos intermedios no siempre son estas, sino quien va a emplear el producto intermedio; por ejemplo, un diseñador es usuario de los productos generados del análisis de requerimientos y, probablemente, de los del diseño de la arquitectura, los cuales espera que estén realizados en modo correcto además de, en esencia, libres de defectos.

5.3.3 Identificación y manejo de riesgos

Un riesgo es un evento posible en el futuro, que en caso de ocurrir tendrá un efecto sobre las metas del proyecto. Se caracteriza por la probabilidad de que ocurra y por el impacto que tendrá en las metas del proyecto. Una buena administración de proyectos incluye estar al pendiente de los riesgos muy probables con potenciales impactos significativos. Por su naturaleza, cualquier riesgo asociado a la arquitectura tiene casi siempre una consecuencia muy alta y negativa en las metas del proyecto.

Las evaluaciones permiten tener visibilidad sobre los riesgos más probables y costosos a los que pueda enfrentarse el proyecto. Esto facilita la toma de decisiones para evitar o disminuir el impacto de los riesgos, con lo cual se establecen estrategias proactivas de la administración del proyecto para que este pueda alcanzar sus objetivos.

En la tabla siguiente se muestran de manera general los riesgos más frecuentes y de mayor impacto en proyectos de desarrollo de *software*.

Categoría	Descripción
Requerimientos funcionales	Las funciones construidas en el producto no satisfacen lo que los usuarios necesitan.
Requerimientos no funcionales	El sistema es incapaz de satisfacer los atributos de calidad que espera el usuario.
Desconocimiento técnico	El personal no cuenta con el conocimiento necesario para plantear una solución técnica que satisfaga los requerimientos de los usuarios.

5.4 CARACTERÍSTICAS DE LOS MÉTODOS DE EVALUACIÓN DE ARQUITECTURAS

Existen varios métodos para evaluar los productos de *arquitectura de software*, de los cuales se distinguen:

1. *Por el momento en el que se realiza la evaluación*. Esto es, mientras se define la arquitectura (diseños parcialmente terminados), en el momento en que ya se terminó de definir (diseños terminados) o cuando ya se desarrolló el sistema basado en ella (sistema terminado).
2. *Por el personal que la realiza*. Personas que están desarrollando o desarrollaron la arquitectura, o bien, personal que no participó en su desarrollo.

En las siguientes subsecciones se describen con más detalle.

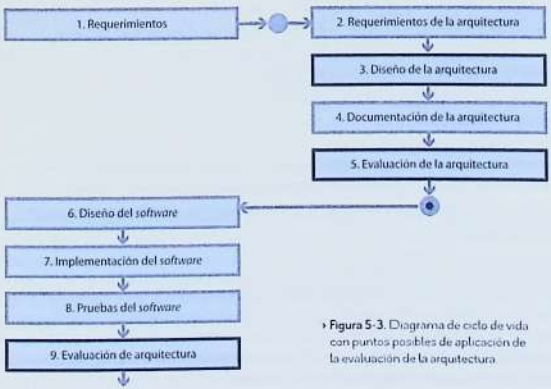
5.4.1 Producto que se evalúa: diseños o productos terminados

La evaluación de arquitectura se aplica tanto a diseños para detectar errores de manera temprana como a productos terminados para evaluar riesgos de colocar en producción el sistema terminado.

En el caso de la evaluación de diseños, por lo habitual se busca que estos sean correctos con base en la especificación disponible de los *drivers arquitectónicos*, y que los cubra de manera adecuada. Sin embargo, también se puede pretender que cumpla con las necesidades de los usuarios aun cuando estos no pueden ver un producto terminado.

En el caso de productos terminados, las evaluaciones buscan asegurar que satisfagan los *drivers* actuales, así como que el producto haya sido construido de manera correcta pues algunos defectos por incumplimientos de *drivers* pueden ser causados por una implementación errónea. Estas evaluaciones permitirán tomar acciones para mitigar riesgos referentes a poner sistemas en producción, o bien, respecto de sistemas que ya están en producción. La mitigación de riesgos puede incluir no salir a producción, generar proyectos de reemplazo, etcétera.

Los métodos de evaluación se incorporan en los procesos utilizados para el desarrollo del producto o de la arquitectura. En la figura 5-3 se resaltan las etapas en las cuales pueden realizarse evaluaciones de arquitectura.



› **Figura 5-3.** Diagrama de ciclo de vida con puntos posibles de aplicación de la evaluación de la arquitectura.

Humberto Cervantes, Rafael Pérez Torres Elizondo, Luis Lopez Carvajal

En el caso del ACDM visto en el capítulo 2, la evaluación se hace en la etapa 4: revisión de la arquitectura. En procesos como RUP, las evaluaciones están indicadas en cada paso hasta la finalización del producto.

5.4.2 Personal que lleva a cabo la evaluación

De acuerdo con sus principios, las evaluaciones de *arquitecturas de software* buscan asegurar que los productos satisfagan los *drivers arquitectónicos*. Por ello son indispensables evaluaciones en las que participen interesados en el sistema, por ejemplo, los usuarios finales, que son algunos de quienes definen los *drivers*. Sin embargo, si solo se llevan a cabo estas evaluaciones, los productos evaluados tienen por lo regular un número muy grande de defectos que ellas serán incapaces de detectar y eliminar, quedando latentes en el producto final, y por lo mismo es muy probable que este tenga problemas de calidad graves.

Lo anterior es una de las razones para que el personal encargado de desarrollar el sistema y, en particular, el de la arquitectura, realice evaluaciones para detectar de manera anticipada los errores en la arquitectura. Lo anterior hará que la evaluación del producto terminado hecha por aquellos interesados externos al equipo de desarrollo sea más efectiva, pues se asegura que el producto terminado ya cuenta con un número reducido de defectos.

El equipo de desarrollo del sistema cuenta además con el conocimiento de los elementos que componen la arquitectura y sus interrelaciones, de tal forma que puede evaluar elementos muy específicos y/o críticos de ella.

5.5 MÉTODOS DE EVALUACIÓN DE ARQUITECTURAS

Las evaluaciones de arquitecturas se realizan de distintas maneras: pueden ir desde revisiones simples hasta las que se hacen en talleres especializados de revisión, todo ello junto con la elaboración de experimentos y modelos que permitan evaluar si se cumplen o no determinadas características de la arquitectura.

Cada método de evaluación tiene ventajas y desventajas, siendo el contexto en el cual se usa lo que influye para obtener los mayores beneficios de los mismos. Durante el diseño de la arquitectura son más recomendables revisiones personales y ágiles, así como el desarrollo de experimentos o prototipos. Al final, convienen revisiones realizadas por personal externo al proyecto para detectar tanto situaciones en que el comportamiento del sistema no se dará como está previsto, es decir, *puntos de sensibilidad* de la arquitectura, como *relaciones de equilibrio* entre atributos de calidad que se contradicen entre sí y estos no han sido notados por el personal del proyecto.

Otras evaluaciones están inmersas en el proceso, como ocurre con el ACDM, el cual por su naturaleza iterativa plantea revisiones durante cada iteración de su proceso de diseño.

A continuación se describen los métodos más reconocidos para evaluar arquitecturas:

- Revisiones e inspecciones.
- Recorridos informales de diseño.
- ATAM (*Architecture Tradeoff Analysis Method*), o Método de evaluación de equilibrios de arquitectura.
- ACDM (*Architecture Centric Design Method*), o Método de diseño centrado en la arquitectura.
- ARID (*Active Reviews for Intermediate Designs*), o Revisiones activas para diseños intermedios.
- Desarrollo de prototipos o experimentos.

5.5.1 Revisiones e inspecciones

Las revisiones e inspecciones están entre las mejores prácticas de *ingeniería de software* para aumentar la productividad y la calidad de los equipos de desarrollo. Las primeras se asocian por lo regular a la evaluación hecha a un artefacto por la persona o personas que lo crearon. Las segundas tienen relación con la evaluación que se le hace a un artefacto por personas que no lo realizaron. En la figura 5-4 se muestran las entradas, salidas, roles de personas y herramientas que se manejan en las revisiones o inspecciones.

› **Figura 5-4.** Revisiones e inspecciones como caja negra (entradas y salidas)

Las inspecciones en el contexto de desarrollo de *software* fueron definidas por primera vez por Fagan (Fagan, 1976). A partir de ese momento se ha visto como una de las mejores prácticas hasta el presente, existiendo gran número de referencias que las explican de manera detallada, como Wong (Wong, 2006).

Para que sean efectivas y logren sus metas de detección de defectos las revisiones e inspecciones, tienen que realizarse con un nivel de formalidad alto siguiendo un proceso definido y cumpliendo con criterios mínimos de calidad. Este tipo de evaluaciones no son frecuentes. En su lugar encontramos revisiones informales por parte de los propios diseñadores y arquitectos, las cuales aunque son muy útiles son menos efectivas que las formales.

Los procesos que las describen plantean criterios de entrada (artefacto a ser evaluado y listas de verificación), pasos a seguir (forma en que se realizarán la evaluación, la corrección de defectos y la verificación de que su eliminación satisfactoria) y criterios de finalización (artefacto evaluado y con todos los defectos eliminados).

Por lo regular, la forma de hacerlas es seguir una lista de verificación, tomar un elemento de esta y revisar el artefacto por completo, observando si cumple con ese elemento. En caso de que no cumpla, se marca como defecto y se corrige. Si lo cumple, el elemento se marca para indicar que el artefacto lo cumple. Se recomienda llevar a cabo primero una revisión personal del artefacto, y posteriormente una inspección de otros desarrolladores. Con la primera se eliminan defectos que podrían distraer fácilmente a los inspectores de encontrar errores más costosos en potencia.

Otra forma de efectuar una revisión o inspección es mediante el uso de los artefactos en lugar de solo hacerlas de manera estática. Implica que los artefactos que están en revisión se usen como está previsto; por ejemplo, si se toma una porción del diseño de la arquitectura, se utilizará para derivar el diseño de la solución e incluso para generar un módulo en código. Este tipo de revisión recibe el nombre de *revisión activa*, y es muy útil para encontrar defectos importantes que pueden tener gran impacto en las metas del proyecto de desarrollo.

Estudios de muchos proyectos que utilizan revisiones e inspecciones siguiendo procesos formales plantean que en el momento de la evaluación encuentran de 50 a 70% de los defectos (Jones, 2010). Estos resultados son mucho mejores que los de las pruebas tradicionales, las cuales encuentran entre 30 y 50% de los defectos. Revisiones e inspecciones más informales hallan también este número de fallas.

La cantidad de defectos que se detectan depende de varios factores:

- **Conocimiento del inspector acerca del artefacto a evaluar.** Es necesario que el inspector de un documento de diseño tenga conocimiento y experiencia en los aspectos técnicos que incorporan los artefactos de arquitectura; de otra manera, serán muy pocos o triviales los defectos que encuentren.
- **Velocidad a la que se hace la revisión o la inspección.** Se ha observado que mientras más rápido se revise o inspeccione un documento, menos defectos se encuentran. Un número genérico es revisar o inspeccionar a una velocidad máxima de cuatro páginas por hora.

Los artefactos a revisar o inspeccionar son: documentación de los *drivers*, escenarios de atributos de calidad, diseños de arquitectura, diagramas de arquitectura, documentos de descripción de la arquitectura, entre otros.

Los métodos de evaluación ATAM, ACDM en su etapa 4, y ARID son en realidad inspecciones ajustadas especialmente a *arquitecturas de software*.

5.5.2 Recorridos informales al diseño

El procedimiento más ampliamente usado es cuando el equipo de arquitectura hace una presentación del diseño de esta a otras audiencias. Se usa con frecuencia porque es muy fácil de realizar y no requiere de mucho esfuerzo; sin embargo, es demasiado limitada en su capacidad de encontrar elementos de riesgo de la arquitectura.

Es una presentación por lo regular de una a dos horas, la cual no tiene un proceso definido y puede usarse para mostrar el tipo de diseño que sea, incluido el arquitectónico. Los asistentes pueden o no ser personal técnico, además, no necesariamente se aplican escenarios de requerimientos funcionales o de atributos de calidad. Los asistentes hacen preguntas al equipo que llevó a cabo la arquitectura, y este da las respuestas correspondientes, registrando los problemas que pueden descubrirse en ella.

Aun con sus limitaciones, es un ejercicio ampliamente recomendado pues además de apoyar la evaluación del diseño de la arquitectura, también es un ejercicio de comunicación de esta.

Estos recorridos pueden mejorarse al incorporarles más elementos de formalidad, como los que pueden encontrarse en otros métodos, como el ATAM y la etapa 4 del ACDM, que se verán en las secciones siguientes.

5.5.3 Método de análisis de equilibrios de la arquitectura (ATAM)

Uno de los procesos de evaluación para *arquitecturas de software* más conocidos es el Método de análisis de equilibrios de la arquitectura, o ATAM, (por sus siglas en inglés). Desarrollado por los investigadores del SEI, tiene como objetivo "evaluar las consecuencias de las decisiones arquitectónicas respecto de los requerimientos de *drivers* del sistema" (Kazman, Klein y Clements, 2000).

Su nombre se debe a que no solo se evalúa que una *arquitectura de software* cumpla con la calidad exigida, sino que analiza las interacciones de los distintos atributos de esta y cómo ellos se equilibran o restringen unos a otros. Puede realizarse más de una vez durante el diseño de la arquitectura, o bien, una vez concluido, y aplicarse además a productos de *software* terminados o en proceso. Es definido por Kazman, Klein y Clements (2000), y se explica con mayor amplitud en el libro de Clements, Kazman y Klein (*Evaluating Software Architectures*, 2002).

El ATAM toma como entradas la documentación de la arquitectura y cualquier otro artefacto que haya sido producido por el diseño de esta. El resultado de hacer una evaluación usando el método es una lista de riesgos en ella, puntos de sensibilidad o que requieren atención, y puntos de equilibrios entre atributos de calidad.

Los roles principales en una evaluación con el ATAM son el arquitecto de *software*, los interesados en el sistema con la arquitectura y el personal que apoya la realización de la evaluación con este método, conocido como equipo ATAM. La figura 5-5 ilustra las entradas, las salidas y otros aspectos del método.

› **Figura 5-5.** El ATAM como caja negra (entradas y salidas).

© Rentería Cedano, Mauricio Ernesto; Vidal Carreras, Luis Carlos Enrique

El método se propuso en un inicio para evaluar los diseños de la arquitectura, así como para detectar de manera temprana cualquier defecto o riesgo, y así fuera resuelto este antes de entrar al desarrollo del producto. Sin embargo, ha sido empleado además tanto en productos legados como en recién terminados para evaluar su arquitectura (cuando se cuenta con ella) y su nivel de cumplimiento de los *drivers*.

El ATAM es uno de los procedimientos más elaborados propuestos por el SEI, tanto que requiere de personal capacitado para poder guiar su práctica. Es común que las organizaciones que requieran una evaluación ATAM no dispongan de este personal, por ello deben apoyarse en personal externo. El SEI es una de las organizaciones que pueden ser contactadas para hacer evaluaciones de esta clase.

Tal situación es importante porque define la forma en que se llevan a cabo las evaluaciones ATAM. De manera cronológica, se da en cinco fases:

1. **Preparación de la evaluación.** Incluye todo el protocolo de contacto entre organizaciones, contratación y alcance de la evaluación. Dado que una evaluación con el ATAM requiere que participen distintos actores, es indispensable que sea planeada y se den las condiciones que permitan su adecuada realización. Por lo mismo, todas inician con una preparación, la cual consiste en determinar su duración más adecuada, asegurar que asistirán todos los interesados en el sistema, integrar los equipos, al igual que definir las fechas y cada uno de los aspectos de logística, como sala de juntas, suministros y herramientas (proyectores, pizarrones, cuadernos, etcétera).
   Puede tardar varias semanas y se realiza principalmente por teléfono, videoconferencia y/o correo electrónico. Personal de contacto y los posibles líderes de cada equipo son quienes por lo habitual la llevan a cabo.
2. **Familiarización con la evaluación.** Es la primera parte de la evaluación, en donde por un lado el equipo ATAM se familiariza con el proyecto y la arquitectura, y por el otro, el equipo de la organización a ser evaluada hace lo propio con el método ATAM.
   Durante esta familiarización se efectúa una evaluación preliminar y se prepara lo necesario (lista de interesados a participar, escenarios, árbol de utilidad, etc.) para efectuar la fase evaluativa. Se realiza por lo habitual en uno o dos días, y participa solo el equipo ATAM, así como el de arquitectura de la organización a ser evaluada.
3. **Ajustes previos a la evaluación.** La familiarización descubre elementos arquitectónicos que deben ser ajustados, tanto en forma y documentación, como en riesgos no considerados por el equipo de arquitectura. Lo anterior puede motivar a que este ajuste elementos antes de realizar la evaluación definitiva. Dependiendo de los cambios a realizar, la fase puede tomar desde unos minutos hasta varias semanas.
4. **Evaluación.** En esta fase se incorporan todos los interesados en el proyecto que estarán afectados por las decisiones de arquitectura. Ellos propondrán escenarios que deben ser cubiertos por esta y serán la base del análisis de la misma. Se identifican los riesgos más significativos, los equilibrios entre distintos atributos de calidad y los puntos de sensibilidad que ella pueda tener. La etapa toma por lo habitual un día o dos y participan los equipos ATAM e interno, así como los interesados en el proyecto.
5. **Elaboración del reporte final.** La última parte consiste en generar un reporte con todos los elementos identificados, con explicaciones y, en algunos casos, recomendaciones. También se recolecta y guarda la información a efecto de mejorar el proceso de ATAM para futuras evaluaciones. Se realiza por el equipo ATAM, y puede tardar varios días.

Las fases de familiarización y de evaluación se llevan a cabo con nueve pasos progresivos en los que se dan discusiones controladas para determinar si la arquitectura cumple o no con los requerimientos de atributos de calidad, y si cuando son incumplimientos deben ser considerados para efectos de corrección u otro tipo de decisiones respecto del producto. En la figura 5-6 se indican los pasos del ATAM.

© Humberto Cervantes Maceda, Pablo Velasco Elizondo, Luis Castro Camargo

› **Figura 5-6.** Pasos del ATAM

Los pasos pueden tener distintos participantes y objetivos, dependiendo de la fase del ATAM en la que se esté. A continuación se describen los pasos de una evaluación utilizando el ATAM para la fase de familiarización.

Fase de familiarización, paso 1: presentación del ATAM. El equipo ATAM presenta el método a los miembros del equipo de arquitectura: se describen sus objetivos, los pasos, las técnicas que serán utilizadas y los resultados esperados.

Fase de familiarización, paso 2: presentación de los *drivers* de negocio. El director del proyecto presenta a los miembros del equipo ATAM varios aspectos acerca del sistema en evaluación, lo cual incluye sus requerimientos relevantes, sus restricciones significativas, las metas de negocio que debe satisfacer cuando esté en producción, los involucrados principales y los elementos clave tomados en cuenta para el diseño de la arquitectura.

Esta presentación permite que el equipo ATAM tenga mejor entendimiento acerca del sistema, y los siguientes pasos de la evaluación sean realizados de manera adecuada. El equipo registra los *drivers* de la arquitectura.

Fase de familiarización, paso 3: presentación de la arquitectura. El equipo encargado de la *arquitectura de software* presenta al equipo ATAM la arquitectura generada durante el diseño, indicando sus aspectos principales, así como las restricciones tomadas en cuenta, por ejemplo, sistemas operativos o sistemas con los que ella interactuará, y cualquier otro aspecto que apoye las decisiones tomadas al diseñar. Esta presentación es de tipo técnico y el equipo ATAM registra las principales decisiones de arquitectura.

Fase de familiarización, paso 4: identificación de las decisiones de arquitectura. El equipo ATAM estructura y ordena toda la información obtenida en los pasos 2 y 3, para tener una base sólida para el análisis, resuelve dudas con el equipo de arquitectura, como las formas en que el sistema reaccionará ante escenarios de crecimiento, interconectividad, seguridad, etcétera.

Fase de familiarización, paso 5: generación del árbol de utilidad. Dado que son muy pocos integrantes del equipo ATAM y del de arquitectura, no es posible utilizar lluvias de ideas para obtener los escenarios con los cuales probar esta. Por ese motivo se usa una técnica la cual parte de la concepción de que la utilidad de un sistema se basa en los atributos de calidad.

Para ello se plantea un *árbol de utilidad*, el cual tiene precisamente a "la utilidad" como raíz, y las ramas son los atributos de calidad que contribuyen más a la utilidad del sistema. A su vez, cada atributo se ramifica en aspectos más específicos. El último nivel del árbol consiste en escenarios o ejemplos concretos de cómo se piensa que la arquitectura soporte esos aspectos.

Este árbol permite pasar de un nivel de abstracción alto de un atributo de calidad a escenarios concretos que la arquitectura debe realizar. A cada escenario se les asigna tanto un valor de prioridad para ser cumplido por esta, como uno de la complejidad técnica que representará efectuarlo.

El árbol es elaborado por el equipo ATAM con el apoyo del equipo de arquitectura. Crearlo es fundamental para la evaluación porque es el elemento con el que se establecen el alcance y las prioridades de los *drivers* que la arquitectura debe cumplir. Este paso es relativamente parecido a las actividades que se realizan durante el taller de atributos de calidad (QAW) descrito en el capítulo 2.

Fase de familiarización, paso 6: análisis de las decisiones arquitectónicas. Se analizan estas decisiones basándose en la información de los pasos 3, 4 y 5. El equipo de evaluación revisa los escenarios de mayor prioridad, el arquitecto resuelve dudas o hace aclaraciones, y se detectan de manera preliminar los riesgos, equilibrios y puntos de sensibilidad.

En este punto concluye la fase de familiarización. Podemos ver un ejemplo de ella en la sección 5 del caso de estudio del apéndice. Con los resultados preliminares del paso 6, el equipo encargado de la arquitectura puede desear hacer ajustes a esta.

La fase de evaluación iniciará una vez que se hayan hecho los ajustes y estén las condiciones para llevarla a cabo: instalaciones, logística y participantes. Los pasos se describen a continuación.

Fase de evaluación, paso 1: presentación del ATAM. El equipo ATAM presenta el método a los participantes. Se describen sus objetivos, los pasos, las técnicas que serán utilizadas y los resultados que se espera obtener de él.

Fase de evaluación, paso 2: presentación de los *drivers* de negocio. El director del proyecto presenta a los involucrados los aspectos más relevantes acerca del sistema en evaluación, lo cual incluye sus requerimientos importantes, restricciones relevantes, las metas de negocio que debe satisfacer cuando esté en producción, los interesados principales y los elementos más trascendentes tomados en cuenta para el diseño de la arquitectura. De esta manera, los participantes conocerán las razones y requerimientos clave del sistema a ser desarrollado.

Fase de evaluación, paso 3: presentación de la arquitectura. El equipo de arquitectura presenta a los participantes la *arquitectura de software* generada durante el diseño, indicando sus aspectos principales, así como las restricciones tomadas en cuenta: sistemas operativos, sistemas con los que interactuará y cualquier otro aspecto que apoye las decisiones tomadas al diseñar. Esta presentación es de tipo técnico, pero en un nivel que los participantes puedan entender. El equipo ATAM registra cualquier decisión arquitectónica nueva o cambio que pudo haber sucedido de lo que percibieron en este paso durante la fase de familiarización.

Fase de evaluación, paso 4: identificación de las decisiones arquitectónicas. El equipo ATAM expone a los participantes toda la información obtenida en la etapa de familiarización con los ajustes que pudieron salir en los pasos 2 y 3, a efecto de tener una base sólida para el análisis en los siguientes pasos.

Fase de evaluación, paso 5: presentación del árbol de utilidad. El equipo ATAM presenta a los asistentes el árbol de utilidad obtenido en la fase de familiarización. Se explica su estructuración, y se muestran los principales escenarios identificados, así como su importancia y la complejidad para su implementación.

Fase de evaluación, paso 6: análisis de las decisiones arquitectónicas. El equipo ATAM expone a los participantes las principales decisiones arquitectónicas identificadas en la fase de familiarización.

El equipo de evaluación revisa junto con los participantes los escenarios de mayor prioridad, y el arquitecto resuelve dudas o hace aclaraciones, se muestran los riesgos, equilibrios y puntos de sensibilidad identificados.

Fase de evaluación, paso 7: generación y priorización de los escenarios a ser considerados durante el análisis. El equipo ATAM se basa en el árbol de utilidad, y mediante una técnica de lluvia de ideas, guía en este punto a los participantes en la obtención de escenarios adicionales a los que el árbol de utilidad proporciona para que propongan otros, por ejemplo, de uso del sistema, así como de atributos de calidad en el uso futuro del mismo, como puede ser crecimiento, adaptaciones, etcétera. Después se hace una consolidación de los escenarios propuestos, eliminando los duplicados y determinando su importancia para el negocio por medio de una votación.

Cada escenario nuevo se coloca con su importancia en el árbol de utilidad, y el equipo de arquitectura determina la complejidad de implementación.

Fase de evaluación, paso 8: realización del análisis de los enfoques de la arquitectura basándose en los escenarios nuevos del árbol de utilidad. El equipo ATAM hace que el equipo de arquitectura describa cómo funcionará esta y resolverá los escenarios más significativos del árbol de utilidad.

Al hacer la descripción, el equipo de arquitectura permite a los asistentes conocer si existen puntos de sensibilidad, las relaciones de equilibrio entre atributos de calidad, o bien, riesgos en la ejecución de los escenarios. El equipo de evaluación documenta todos los hallazgos.

Fase de evaluación, paso 9: presentación del equipo ATAM de los resultados de la evaluación a los involucrados participantes. Este paso resume las entradas a la evaluación y los hallazgos de esta: puntos de sensibilidad, relaciones de equilibrio entre atributos de calidad y riesgos. También se describen las sugerencias para dar los pasos siguientes: ajustes a la arquitectura, o bien, elaboración de prototipos o experimentos adicionales.

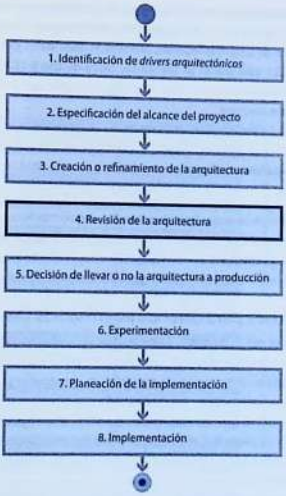
La última fase de una evaluación ATAM es la generación del reporte final. En ella, el equipo ATAM prepara y entrega este a la organización.

Después de una evaluación con el método, a los planes de proyectos debe incorporarse acciones que implementen las estrategias de mitigación de riesgos, y debe haber un seguimiento adecuado para asegurar que estos sean mitigados, o bien, que sean aceptados por todos los interesados del proyecto.

5.5.4 ACDM (etapa 4)

El método de diseño centrado en la arquitectura ACDM (Lattanze, 2008) fue presentando en la sección 2.3.2. En ella señalamos sus ocho etapas, las cuales volvimos a mostrar en la figura 5-7 señalando que la cuarta corresponde a la evaluación de la arquitectura.

El ACDM es un método integral, el cual se recomienda utilizarlo siguiendo estrictamente las etapas que propone, y eso hace diferente la evaluación respecto de los métodos evaluativos que pueden utilizarse junto con distintos procesos de desarrollo.



© Humberto Cervantes Maceda / Pedro Velasco Elizondo / Rafael Carmona Cejudo

› Figura 5-7. Etapas del ACDM.

La etapa 4 del ACDM plantea que la evaluación puede ser realizada por el equipo de desarrollo o por uno externo y pone énfasis en que la arquitectura sea capaz de satisfacer los requerimientos y escenarios funcionales, sin dejar a un lado los de atributos de calidad.

Como es parte integral de un método que sigue todo el ciclo de vida de la arquitectura, las entradas para realizarlo son la especificación de *drivers* y el documento de diseño de la arquitectura. Su salida son problemas y riesgos detectados en ella para satisfacer los *drivers*.

La etapa 4 de revisión que hace el ACDM se lleva a cabo mediante un taller en el que participan distintos roles del equipo de desarrollo: equipo de arquitectura, equipo de requerimientos, soporte, calidad, procesos y operación.

El taller tiene las actividades siguientes:

1. **Introducción.** Los asistentes se presentan entre sí y se expone de manera general el ACDM, además de cómo se realizará la etapa 4.
2. **Revisión del contexto y restricciones de negocio.** Se presentan a los asistentes todas las consideraciones de negocio descubiertas en la etapa 1 y analizadas en la 2, como presupuesto, fechas comprometidas, recursos humanos, estructura organizacional del proyecto de negocio, comercialización y aspectos legales.
3. **Revisión de los requerimientos y las restricciones técnicas.** Se presenta a los participantes un breve recordatorio acerca de requerimientos funcionales y de los de atributos de calidad, al igual que de restricciones técnicas clave en el proyecto.

4. **Presentación de la arquitectura.** Se presenta a los asistentes la arquitectura que satisface los *drivers*, indicando cuáles son estos, así como estrategia de diseño, contexto del diseño, arquitectura obtenida, influencias externas o de sistemas legados, problemas principales y riesgos.
5. **Análisis funcional.** En este punto se toman los casos de uso y se revisa cómo la arquitectura debe manejarlos para que sean llevados a cabo. Cualquier situación que impida su realización se toma como un problema de la arquitectura.
6. **Análisis estructural.** Se toman escenarios de atributos de calidad y se revisa que la arquitectura pueda manejarlos. Cualquier situación que impida su realización se toma como un problema de la arquitectura.
7. **Resumen: revisión de los problemas encontrados.** En este paso se resumen los hallazgos encontrados en los pasos 5 y 6.

Después de la etapa 4 del ACDM, la etapa 5 plantea tomar la decisión de hacer otra iteración en las etapas 1 a 4, o bien, pasar a las siguientes etapas del método. La etapa 4 es muy efectiva pues integra de manera natural la evaluación de la arquitectura con la especificación de los requerimientos y el diseño arquitectónicos, así como con la toma de decisiones acerca de seguir mejorando el diseño arquitectónico o avanzar a las siguientes etapas del método.

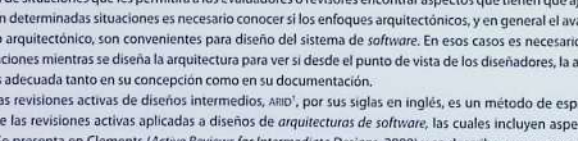
5.5.5 Revisiones activas para diseños intermedios (ARID)

Las revisiones llevan siempre el riesgo de que quienes se dedican a ellas hagan un trabajo deficiente, ya sea por tener un nivel técnico bajo o por una experiencia escasa respecto del producto que están evaluando. También puede ocurrir que para cuando se debe evaluar la arquitectura el proyecto ya está presionado, y las revisiones específicas, o no se hacen o son deficientes, o bien, son tardías.

Las revisiones activas plantean que los encargados de ellas deben ejecutar los productos a ser evaluados para detectar errores, excesos y carencias. Las realizadas a diseños evalúan productos de estos mediante la aplicación de situaciones que les permitirá a los evaluadores o revisores encontrar aspectos que tienen que ajustarse.

En determinadas situaciones es necesario conocer si los enfoques arquitectónicos, y en general el avance del diseño arquitectónico, son convenientes para diseño del sistema de *software*. En esos casos es necesario aplicar evaluaciones mientras se diseña la arquitectura para ver si desde el punto de vista de los diseñadores, la arquitectura es adecuada tanto en su concepción como en su documentación.

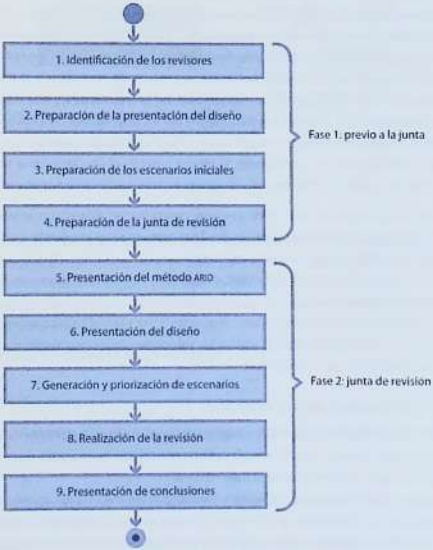
Las revisiones activas de diseños intermedios, ARID[1], por sus siglas en inglés, es un método de especialización de las revisiones activas aplicadas a diseños de *arquitecturas de software*, las cuales incluyen aspectos del ATAM. Se presenta en Clements (*Active Reviews for Intermediate Designs*, 2000) y se describe con mayor amplitud en Clements. *et al.* (2008) En la figura 5-8 se muestran sus entradas, salidas y participantes. Es notable que el ARID nombre a un *diseñador de la arquitectura*, sin embargo, puede considerarse idéntico al rol de *arquitecto* que se ha utilizado en los capítulos anteriores.



› **Figura 5-8.** El ARID como caja negra (entradas y salidas)

[1] *Active Reviews for Intermediate Designs*

El método ARID plantea dos fases y nueve pasos; de estos, cuatro son actividades necesarias para tener la junta de revisión, y cinco se realizan durante la junta. En la figura 5-9 se muestran sus fases, y después se describen con mayor detalle.



› **Figura 5-9**. Fases y pasos del ARID.

Fase 1: previo a la revisión

Paso 1: identificación de los revisores. Se selecciona a las personas idóneas para que participen en la revisión.

Paso 2: preparación de la presentación del diseño. El creador de la arquitectura se prepara para presentar el diseño de la misma a efecto de que personas con conocimiento técnico sean capaces de entender los principales componentes arquitectónicos y sus interacciones.

Paso 3: alistamiento de los escenarios iniciales. Se preparan escenarios para ilustrar los conceptos básicos de la arquitectura y facilitar así a los diseñadores de programas y/o programadores el entendimiento de estos. Los escenarios son ejemplos, y los revisores podrán utilizarlos para realizar la revisión o generar códigos de los que consideren más adecuados.

Paso 4: preparación de la junta de revisión. Se hace toda la logística de preparación de la junta: reproducción de materiales, invitaciones, reservaciones de salas de juntas, refrigerios, al igual que infraestructura, como proyectores, pizarrones, etcétera.

Fase 2: Junta de revisión

Paso 5: presentación del ARID. El facilitador presenta el método a los asistentes.

Paso 6: presentación del diseño. El creador de la arquitectura presenta a los asistentes el diseño de esta, teniendo como objetivo que entiendan sus características y las principales decisiones hechas en él. No hay discusión acerca de lo idóneo de las decisiones, solo se resuelven dudas acerca de su comprensión.

Paso 7: generación y priorización de escenarios. Al igual que en el ATAM, los asistentes sugieren escenarios y los priorizan. Esos escenarios serán aquellos sobre los cuales se realizará la revisión del diseño de la arquitectura.

Paso 8: realización de la revisión. Los asistentes toman los escenarios y el diseño de la arquitectura para intentar un diseño detallado de la solución que use la arquitectura y que resuelva el escenario. Ejercitar esto permitirá saber si la arquitectura es utilizable por los diseñadores de *software*, o conocer la identificación de aspectos complementarios, de clarificación, o de corrección de errores.

Paso 9: presentación de conclusiones. El último paso en el ARID es resumir todos los hallazgos encontrados durante la revisión para que sean resueltos por el diseñador de la arquitectura.

En general, tanto ATAM como ARID son muy parecidos; las diferencias se encuentran principalmente en el punto de vista que se tiene para la evaluación. En el primero se asume que la arquitectura es realizable como está y se evalúa el posible sistema final para ver que cumpla con los *drivers*. En el segundo no se asume que es realizable, sino que justo es lo que se va a evaluar, obteniendo hallazgos centrados en la descripción de la arquitectura y la coherencia técnica de esta.

En el ARID los revisores o evaluadores requieren tener el nivel técnico necesario para entender los aspectos detallados del diseño arquitectónico y plantear soluciones de programación que lo utilicen. En el caso del ATAM no es indispensable tal conocimiento.

5.5.6 Prototipos o experimentos

Los diseños de cualquier tipo plantean ideas que se implementarán posteriormente en programas que serán los constituyentes del sistema de *software*. El sistema construido mostrará de manera muy clara sus capacidades e incapacidades, pero su corrección será muy costosa.

Los prototipos o experimentos buscan mostrar que las ideas de los diseños son o no realizables, permitiendo la adquisición de conocimiento que puede ser muy valioso sin tener que desarrollar el sistema por completo. También permiten obtener conocimiento sobre tecnología nueva o desconocida para el arquitecto, y observar así que su uso en la arquitectura será o no riesgoso.

Un prototipo es una solución muy reducida de desarrollo rápido que permite afirmar o negar las suposiciones en las cuales se basa la arquitectura. Un experimento es el planteamiento de una situación práctica con el que se confirma o niega una hipótesis en la cual se fundamenta la arquitectura.

En la figura 5-10 se muestran las entradas y salidas de prototipos o experimentos. La realización de ambos tiene como entradas los riesgos detectados durante el diseño, las hipótesis en las que están basadas las decisiones arquitectónicas y, finalmente, los escenarios relevantes que debe cubrir la arquitectura.

Los prototipos son propuestos por los arquitectos o diseñadores, y los llevan a cabo los desarrolladores. Utilizan herramientas e infraestructura de desarrollo y de comunicaciones, así como dispositivos especiales cuando lo requieren. Sus salidas son una conclusión que confirma o niega la hipótesis inicial, una lista de riesgos actualizada con base en las conclusiones y cualquier problema detectado en la arquitectura al momento de realizar el prototipo o experimento.

La elaboración de prototipos o experimentos es un mecanismo de verificación y/o validación de la arquitectura, pues permite conocer de manera práctica si esta es capaz de cumplir o no con determinados escenarios de atributos de calidad.

› **Figura 5-10**. Desarrollo de prototipos como caja negra (entradas y salidas)

Los prototipos o experimentos proporcionan también conocimiento nuevo a los arquitectos acerca de las soluciones planteadas, de tal forma que pueden ajustarse para asegurar que la implementación de la arquitectura en un sistema se lleve a cabo con menores riesgos y de manera más eficiente: los prototipos, por ejemplo, sirven con frecuencia como punto de partida o elementos reutilizables para los equipos de desarrollo.

Los usos de los prototipos o experimentos son:

- Obtención de información acerca de riesgos arquitectónicos.
- Prospección tecnológica.
- Herramientas de aprendizaje.

Los resultados que se esperan son conocimiento nuevo, confirmación o rechazo de hipótesis y elementos básicos de reutilización.

Las limitaciones de los prototipos o experimentos son:

- La escala del prototipo o experimento nunca será la misma que la del sistema completo, lo cual puede generar imprecisiones en las conclusiones.
- Son actividades que deben formar parte del proyecto a efecto de que sean visibles para todos los involucrados.
- Son consumidores de recursos y tiempo del proyecto.

5.5.7 Comparación de métodos de evaluación

Ya revisados de manera general los métodos evaluativos, es conveniente tener un resumen de sus ventajas y desventajas y en qué contextos pudieran ser más útiles unos que otros. A continuación se muestra un cuadro comparativo de todos ellos.

	Revisiones e inspecciones	Recorridos informales al diseño	ATAM	Etapa 4 del ACDM	ARID	Prototipos o experimentos
Tipo	Mejor práctica	Mejor práctica	Método	Método	Método	Mejor práctica
Duración	Dependen del tamaño del artefacto a revisar. Típicamente cuatro páginas por hora (máximo)	Una a dos horas	Tres a cinco días	Medio día a dos días	Cuatro horas	Un día, máximo, por prototipo
Mecánica y enfoque	Revisión del artefacto apoyado en listas de revisión	Presentación del diseño de la arquitectura a una audiencia	Elección de *drivers arquitectónicos*, preparar escenarios y confrontarlos a la arquitectura con apoyo de facilitadores	Se aplican escenarios a la arquitectura con apoyo de facilitadores	Se toma la arquitectura y se intenta utilizar para hacer acciones tanto de diseño como de programación y encontrar así deficiencias	Se toma una hipótesis y se diseña un experimento para asegurar su validez mediante el desarrollo de *software* que implemente el experimento
Provee un proceso que especifica las actividades, funciones y artefactos de entrada y salida	Sí (formales) No (informales)	No	Sí	Sí	Sí	No
Participantes	Quien elaboró el artefacto o colegas	Variados	Interesados, arquitectos y facilitadores	Interesados, arquitectos y facilitadores	Diseñadores y/o programadores	Diseñadores y/o programadores, así como arquitectos
Entradas	Cualquier artefacto a ser evaluado	Diseño de la arquitectura	Documentación del diseño de la arquitectura	• Drivers de la arquitectura • Documentación del diseño de la arquitectura	Documentación del diseño de la arquitectura	Porciones críticas del diseño de la arquitectura
Salidas	Lista de defectos encontrados y corregidos	Lista informal de observaciones	Lista de riesgos, puntos de sensibilidad y defectos encontrados	Lista de problemas de la arquitectura para toma de decisión	Lista de deficiencias del diseño de la arquitectura	Confirmación o negación de la hipótesis y puntos de sensibilidad
Criterios de Terminación	Todos los defectos detectados están corregidos	Se ha recorrido la arquitectura y no hay más preguntas	Identificados riesgos, no riesgos, puntos de sensibilidad y equilibrios	Se han aplicado todos los escenarios relevantes a la arquitectura	Tiempo límite y los defectos de la documentación de la arquitectura identificados	Prototipo funcionando o no en un tiempo límite
Contexto de utilidad	Cualquier documento o artefacto	Cualquier diseño	Diseños arquitectónicos de sistemas grandes y críticos que incluyan tecnología nueva	Diseños arquitectónicos de sistemas grandes y críticos que incluyan tecnología nueva	Factibilidad de diseños de arquitectura, y partes seleccionadas de ellos	Puntos específicos en los que se basan los riesgos más importantes de la arquitectura

EN RESUMEN

En este capítulo describimos las prácticas de las evaluaciones para asegurar que la arquitectura es correcta y completa, además de útil para su implementación por medio del diseño y la construcción de programas. La corrección se plantea de acuerdo con los *drivers arquitectónicos* identificados tanto en la etapa de requerimientos como en el momento de la evaluación.

También mostramos las ventajas y beneficios de la detección temprana de defectos para evitar repetir trabajo en etapas posteriores del desarrollo. Examinamos distintos tipos de evaluaciones, algunas de carácter general como las revisiones e inspecciones, y otras configuradas especialmente para evaluar arquitecturas, por ejemplo el ATAM, la etapa 4 del ACDM y el ARID.

Mencionamos además que los prototipos y experimentos pueden considerarse como evaluaciones de conceptos o hipótesis sobre las cuales se fundamenta el diseño de la *arquitectura de software* de los sistemas.

Si un proyecto llega al punto de las evaluaciones de arquitectura, al realizarlas se podría asegurar que esta ya es correcta, completa y útil. Sin embargo, falta afianzar que la implantación del sistema se apegue a lo indicado en aquella. Este es el tema del que se ocupa el siguiente capítulo.

PREGUNTAS PARA ANÁLISIS

1. ¿Por qué es conveniente hacer una evaluación de arquitectura para un producto de *software* ya terminado?
2. ¿Cuál es la diferencia entre una verificación y una validación?
3. ¿Cuáles son más efectivas, las revisiones o las pruebas para la detección de defectos?
4. ¿Por qué es deseable una revisión personal antes de una inspección?
5. ¿Qué riesgos se asocian con las inspecciones?
6. Como una lista de revisión debe ser breve para que su aplicación sea efectiva, ¿cuáles criterios a revisar deberían integrarla en términos generales?
7. ¿Por qué el método ATAM se hace en dos etapas?
8. ¿Cuáles son las diferencias entre el método ATAM y el método ARID? Si en un proyecto debiera usted llevar a cabo tanto el uno como el otro, ¿cuál de los dos aplicaría primero y por qué?
9. ¿Cuáles son los riesgos que detecta principalmente el ARID?
10. ¿Qué riesgos elimina la creación de prototipos?

CAPÍTULO 6

IMPLEMENTACIÓN: CONVERTIR EN REALIDAD LAS IDEAS ARQUITECTÓNICAS

Se piensa con frecuencia que el trabajo del equipo de arquitectura concluye cuando se tiene un diseño que ha sido evaluado y cubre satisfactoriamente con los requerimientos, pues se asume que el equipo de desarrollo implementará el diseño de la arquitectura tal y como fue especificado. Sin embargo, la falta de seguimiento durante la implementación del sistema puede hacer que el diseño de la arquitectura no quede plasmado adecuadamente.

El capítulo describe aspectos de la implementación influidos por la *arquitectura de software* y explica que sus discrepancias con esta son fuente de problemas durante el desarrollo y la operación del sistema. En la figura 6-1 se muestra el ciclo de desarrollo arquitectónico y se ubica la etapa de implementación dentro de este.

› **Figura 6-1**. La etapa de implementación del sistema en el ciclo de desarrollo de la arquitectura.

6.1 CONCEPTO DE IMPLEMENTACIÓN DE *SOFTWARE*

Las personas no relacionadas con los aspectos técnicos de los sistemas solo están interesadas en que estos les sean útiles y resuelvan sus necesidades de control y disponibilidad de información. La implementación de los sistemas genera el producto de *software* que será utilizado por los distintos usuarios.

La implementación se hace por lo habitual con base en arquitecturas preestablecidas, y las técnicas de diseño y construcción de sistemas se apegan a ellas. La implementación es la acción de transformar especificaciones en programas que serán funcionales para los usuarios.

La implementación comprende:

- Diseñar la estructura general del sistema basándose en la arquitectura. Lo anterior se refiere a identificar todos los módulos que permitirán soportar los requerimientos funcionales.
- Diseñar de manera detallada los módulos del sistema basándose en la arquitectura y los requerimientos funcionales.
- Desarrollar y/o adquirir los módulos especificados en el diseño.
  - Programar código de cada uno de los módulos funcionales que no son adquiridos, de acuerdo a la especificación de sus interfaces.
  - Incorporar cada elemento adquirido.
  - Probar de manera individual cada uno de los módulos.

- Integrar los módulos desarrollados y/o adquiridos y probar que juntos funcionan como se espera.
- Probar los distintos niveles de integración hasta lograr la totalidad del sistema.
- Corregir cualquier error detectado en cada uno de los pasos.

Las actividades de implementación incorporan a la mayoría de los recursos humanos involucrados en el desarrollo de los sistemas.

6.2 LA ARQUITECTURA Y LA IMPLEMENTACIÓN DEL SISTEMA

La arquitectura es un conjunto de descripciones de la estructura de los elementos del sistema. Estas son abstractas y no se concretan hasta la implementación del sistema por medio de programas o elementos de *software* y *hardware* que interactuan entre sí.

Con frecuencia, el sistema resultante no cumple de manera adecuada con los *drivers*. Las razones pueden ser por una arquitectura errónea o porque los desarrolladores no siguen la arquitectura, o ambas.

Cuando la arquitectura no es la adecuada para los requerimientos del sistema, su implantación genera un producto que no satisface bien lo que necesitan los usuarios. Esto puede deberse a una deficiente identificación de requerimientos y *drivers* y/o a un diseño arquitectónico deficiente. Tales situaciones deberían identificarse por medio de una evaluación de la arquitectura, como se indicó en el capítulo 5.

El diseño de la estructura general en término de módulos y el diseño detallado de estos últimos deben considerar los elementos que señale la arquitectura. Cuando los desarrolladores no la siguen, crearán diseños de un sistema con una estructura no documentada y esta tendrá comportamientos no previstos desde el punto de vista de los *drivers*. Por lo mismo, la implementación de tales diseños constituirá un sistema que no se puede asegurar que satisfaga los *drivers*.

6.3 PRINCIPIOS DE LA IMPLEMENTACIÓN DE *SOFTWARE*

Para realizar la implementación del *software* deben seguirse algunos principios que aseguren el éxito de la implementación:

Principio 1. Generar diseños detallados de los módulos y otros elementos acordes con la arquitectura. El diseño detallado de módulos y otros elementos debe apegarse a la especificación de las interfaces y ceñirse a lo indicado en la arquitectura del sistema, pues de lo contrario podrían impactar negativamente en la integración y en la satisfacción de los *drivers*.

Principio 2. Solo el equipo o persona responsable de la arquitectura puede realizar ajustes en esta. En caso de encontrar errores, deficiencias o faltantes arquitectónicos, el equipo de implementación no debe intentar hacer las acciones de corrección o eliminación. Todas esas fallas deben notificarse siempre al equipo o persona responsable de la arquitectura para que sean resueltas y documentadas de manera adecuada.

Principio 3. Realizar la programación y/o adquisición de código existente acorde con los diseños detallados. Dado el principio 1, se acepta que los diseños detallados de los módulos y otros elementos cumplen con la arquitectura. Esto implica que toda la programación basada en estos diseños detallados cumplirá a su vez con los lineamientos que establece la arquitectura.

Principio 4. Ajustar los diseños detallados al encontrar errores o deficiencias. En caso de que la programación y/o la adquisición de código existente encuentre errores, deficiencias o faltantes en los diseños detallados de los módulos, el equipo de implementación debe hacer la resolución en estos antes de ajustar el código de los programas.

Al seguir los principios anteriores se eliminan los motivos de las inconsistencias entre implementación y arquitectura.

6.4 DESVIACIONES DE LA IMPLEMENTACIÓN RESPECTO DE LA ARQUITECTURA

Una *desviación* es la situación en que el diseño detallado de un módulo o su programación no son consistentes con las descripciones indicadas en la arquitectura. Las desviaciones provocan no asegurar que el sistema resultante satisfaga los *drivers*.

Se generan por el equipo de implementación y son de dos tipos:

- *La implementación corrigió errores, defectos u omisiones de la arquitectura, pero esta no fue corregida o ajustada.* Tales desviaciones tienen como consecuencia que las correcciones no fueron analizadas y diseñadas de acuerdo con criterios arquitectónicos, lo cual aumenta los riesgos de no satisfacer los *drivers*. De manera adicional, la documentación pierde valor pues no refleja la arquitectura real del sistema de *software*.

  Estas desviaciones se dan con frecuencia por desarrolladores muy proactivos que no siguen los procesos de control de cambios, y también suceden por falta de comunicación o empatía entre el equipo de implementación y el de arquitectura.
- *La implementación ignoró elementos de la arquitectura.* Al igual que el tipo anterior, estas desviaciones tienen como consecuencia el incremento del riesgo de no satisfacer los *drivers* arquitectónicos una vez que la implementación esté concluida.

  Estas desviaciones se dan por lo habitual por desarrolladores que desconocen la arquitectura o tienen poca disciplina para seguir las especificaciones arquitectónicas.

Las desviaciones son factores de riesgo del sistema al momento de colocarlo en producción. Cuando son detectadas, hay que tomar una decisión a efecto de resolverlas, o bien, no hacerlo. Este concepto forma parte de lo que se conoce como deuda técnica, ya que el proceso de toma de decisión sigue criterios financieros en relación con dejar que las deudas aumenten o sean pagadas.

Tanto más tiempo se mantengan las desviaciones, cuanto más aumentará la deuda técnica y, en consecuencia, el costo de retrabajar la arquitectura y/o la implementación de esta. El incremento en esa deuda se debe a los "intereses" que gana con el tiempo cuando no es resuelta.

Cuando se eliminan las desviaciones hay un costo asociado a rehacer el trabajo (retrabajo) de la arquitectura y/o implementación en el momento en que este se realice, incluidos los "intereses" ganados por la deuda técnica desde que ocurrió la desviación hasta que se corrige.

Los modelos de crecimiento de deuda mantienen una tendencia exponencial respecto del tiempo, de tal forma que en algún momento pueden ser tan grandes que con frecuencia la mejor decisión es no hacer la corrección y tomar los riesgos implícitos en las desviaciones.

6.5 PREVENCIÓN DE DESVIACIONES

Al hacer el análisis del manejo de la deuda técnica resulta claro que la mejor estrategia es la prevención de las desviaciones. Esta radica en evitar las causas que generan las desviaciones, de entre las cuales, las principales son:

- Falta de entrenamiento en los equipos de implementación.
- Carencia de seguimiento de los procesos de control de cambios.
- Comunicación deficiente entre el equipo de arquitectura y el de implementación.

6.5.1 Entrenamiento de diseñadores y programadores

El equipo de implementación está formado por personas que deben tener conocimientos técnicos necesarios para desarrollar los diseños detallados de los módulos y sus implementaciones de acuerdo con la arquitectura. Los conocimientos técnicos necesarios son establecidos por la arquitectura. Contar con los recursos humanos adecuados para la implementación depende del reclutamiento y la capacitación.

Respecto del reclutamiento, este debe considerar los conocimientos y habilidades necesarios para los distintos roles involucrados con la implementación. A efecto de asegurar que sea adecuado, se plantean prácticas como la revisión del currículo, entrevistas, exámenes técnicos y pruebas de habilidades verbales y matemáticas. Estas revelan deficiencias en los aspirantes, las cuales pueden o no ser resueltas con entrenamiento, y dan a conocer posibles riesgos relacionados con actitudes inadecuadas para el trabajo en equipo o las relaciones con los involucrados en los proyectos.

El entrenamiento es consecuencia de la detección de deficiencias en los recursos humanos, las cuales pueden ser técnicas, de conocimiento del problema a resolver, o bien, acerca de la arquitectura o los procesos que deben usarse durante la implementación. A efecto de tener entrenamientos efectivos se recomienda llevar una secuencia personalizada de estos en todos los miembros del equipo, y que cada entrenamiento tenga criterios de evaluación de los conocimientos y/o habilidades aprendidos.

Debe considerarse cualquier medio que pueda apoyar el aprendizaje: intrucción presencial o a distancia, *e-learning*, tutoriales, libros, presentaciones, seminarios, etcétera.

Es muy importante considerar que el equipo de implementación debe ser instruido en la *arquitectura de software*. Deben dedicarse tiempo y recursos para un adecuado entrenamiento acerca de ella. Una manera dinámica de hacerlo es que el arquitecto tenga un rol de mentor de la arquitectura para el equipo de implementación.

6.5.2 Desarrollo de prototipos o experimentos

Como se mencionó en el capítulo 5, los prototipos o experimentos son mecanismos para evaluar si un concepto arquitectónico funciona o no. En el ámbito de la implantación, aquellos que son realizados para la evaluación servirán como elementos de aprendizaje para el equipo que implementa, pues describen la forma en que determinados aspectos deben ser implementados de acuerdo con el punto de vista del equipo de arquitectura. Por ello es recomendable que todos los prototipos o experimentos estén disponibles para el equipo de implementación, con lo cual se previene que dicho equipo construya elementos del sistema conforme a su propia interpretación, evitando así las posibles desviaciones.

6.5.3 Otras acciones de prevención

Por lo habitual la principal accion preventiva es seguir los procesos para la obtención de una arquitectura y para hacer una implementación siguiéndola. En caso de no contar con un proceso, la primera acción es crearlo e implantarlo.

Los defectos de una arquitectura generada siguiendo de manera deficiente el proceso, o bien, sin proceso alguno, con frecuencia son fuentes de desviaciones durante la implementación. Un proceso adecuado y su realización disciplinada contribuyen a la prevención de desviaciones.

El método ARID descrito en el capítulo 5 apoya en buena medida la prevención pues permite identificar subespecificaciones en la documentación de la arquitectura, así como situaciones que serían desviaciones en potencia durante la implementación.

6.6.3 Auditorías

Las pruebas tienen una efectividad reducida para la detección de defectos y desviaciones, además de que exigen gran cantidad de esfuerzo y tiempo para su realización. En cambio, las auditorías son un mecanismo alterno a las pruebas, más económico en esfuerzo y tiempo para localizar indicios de desviaciones o errores arquitectónicos. Se seleccionan de preferencia algunos módulos críticos y se analizan por expertos técnicos o personal del equipo de arquitectura buscando desviaciones. La auditoría solo busca indicios, si los encuentra, se hacen revisiones más detalladas a efecto de tener información más específica que sirva para la toma de decisiones.

La toma de decisiones se da respecto al manejo del riesgo del proyecto, el cual tiene que ver con decisiones de seguir o no seguir. En caso de no continuar, se debe optar entre si se va a retrabajar el módulo o se va a rehacer o, incluso, rehacer el sistema entero. Lo anterior proporciona visibilidad acerca del riesgo que puede existir en el sistema debido a las desviaciones.

El objetivo de las auditorías es tener de manera rápida elementos de información para la toma de decisiones, no tanto corregir las desviaciones detectadas. Las acciones pueden considerar hacer auditorías más detalladas, pruebas de estrés, desempeño o seguridad, rediseños, reprogramación de módulos, extensión de los periodos de prueba, etcétera.

6.7 RESOLUCIÓN DE LAS DESVIACIONES: SINCRONIZACIÓN DE LA ARQUITECTURA Y LA IMPLEMENTACIÓN

De acuerdo con lo señalado en el capítulo 5 y en la sección 6.4 sobre defectos no identificados oportunamente y la deuda técnica (errores identificados pero no corregidos), tanto más tiempo pase entre la generación de la desviación y su eliminación, cuanto mayor será el costo para resolverla. Por esta razón se busca eliminar cuanto antes las desviaciones o errores detectados. Esta resolución puede implicar correcciones a cada uno de los elementos: arquitectura, diseños detallados o programación.

6.7.1 Desviaciones en el diseño detallado

Resolver desviaciones en el diseño detallado de un módulo antes de que se realice la programación del mismo es más económico, pues al no tener artefactos generados, solo se tienen los costos de retrabajo del diseño.

Asumiendo que las desviaciones son que el diseño no respeta la arquitectura, las acciones de corrección deben ser rehacer los elementos del diseño que no se apegan a las especificaciones de la arquitectura. Estos elementos pueden incluir diagramas, descripciones de las responsabilidades e interfaces de componentes, casos de prueba, nomenclatura y/o seudocódigo.

Las desviaciones deben quedar registradas de alguna manera para que el retrabajo necesario sea visible por medio de ajustes al plan de proyecto que incluya las actividades de corrección. Esto implicará un control de incidencias y un control de cambios.

6.7.2 Desviaciones en la programación

Cuando se identifican las desviaciones en la programación, estas son respecto del diseño detallado del módulo, el cual debe estar apegado a la arquitectura. Estas desviaciones no son evidentes en relación con la arquitectura, y se tratan por lo habitual contra el diseño. Cuando hay supervisión del equipo de diseño o del de arquitectura, son fácilmente manejables. Si esto no ocurre, posiblemente solo se detectarán hasta la fase de prueba y, por ende, puede darse en un momento en que el costo de retrabajo sea tan grande que la corrección implique gastos que salen del presupuesto original del proyecto, por lo que se tomarán otro tipo de decisiones.

La programación puede generar desviaciones por desconocimiento de aspectos de la arquitectura y por subespecificación o defectos en el diseño detallado de los módulos. Las correcciones necesarias se dan de acuerdo con la fuente de la desviación; si son de la programación respecto del diseño, entonces se corrige el código.

Cuando la desviación es producto de subespecificaciones y/o errores en el diseño, entonces hay que seguir el proceso de control de cambios para hacer los ajustes correspondientes en el diseño y tener un control de versiones adecuado. Esto puede provocar que algunos de los productos de la programación ya terminados requieran retrabajarse, pues fueron realizados con una versión del diseño que tenía errores y, por lo mismo, presentan también desviaciones.

6.7.3 Defectos y/o subespecificación en la arquitectura

Cuando el origen de las desviaciones se debe a errores en la arquitectura o a subespecificación en la misma, los diseños detallados de los módulos y su programación han sido elaborados sobre bases erróneas y se pierde en consecuencia el efecto unificador que la arquitectura tiene sobre ellos. Lo anterior da la posibilidad de generar varios diseños detallados de módulos que difieren entre sí en aspectos arquitectónicos.

El procedimiento a seguir debe ser el siguiente:

- El diseñador y/o desarrollador de los módulos encuentra un aspecto arquitectónico erróneo o subespecificado.
- Se notifica la situación al equipo de arquitectura.
- Se evalúa el impacto o deuda técnica asociada.
- En caso de que se decida eliminar la deuda técnica, se hacen las adecuaciones a la arquitectura correspondientes.
- En caso de que no se decida eliminar la deuda técnica, se documenta y reconoce su existencia.
- Si los cambios a la arquitectura son importantes, se vuelve a evaluar esta para asegurar que sigue cumpliendo con los *drivers*.
- Se elabora una versión arquitectónica nueva.
- Se comunican la versión reciente y los cambios a la arquitectura al diseñador y/o desarrollador de los módulos.
- El diseñador y/o desarrollador de los módulos retrabaja los diseños afectados por los cambios en la arquitectura y genera versiones nuevas de estos.
- En caso de ser dos entidades distintas, el diseñador de los módulos comunica a los desarrolladores tanto la versión nueva como los cambios en el diseño.
- El equipo de programación retrabaja los programas afectados por los cambios en los diseños de los módulos y genera versiones nuevas de estos.
- Se da por terminada la desviación.

Note que debe manejarse el control de cambios y de versiones en todos los artefactos implicados con la desviación.

6.7.4 Cuándo conviene no resolver las desviaciones

En caso de que la deuda técnica sea muy grande, y por los tiempos del proyecto y los recursos disponibles sea imposible resolver las desviaciones, la decisión puede ser aceptar las desviaciones con sus riesgos y costos asociados. Decisiones de ese tipo se realizan por el equipo de dirección del proyecto y los usuarios/clientes, pues pueden estar involucrados aspectos de manejo de contratos y otras obligaciones.

Tomar la decisión implicará seguir una serie de pasos:

- Determinar las acciones necesarias para evitar que la desviación se multiplique en otros diseños de módulos.
- Determinar los riesgos adquiridos por mantener la desviación.
- Determinar acciones requeridas para manejar los riesgos una vez que el sistema sea puesto en producción, por ejemplo, horarios especiales de operación de ciertos módulos y/o determinados tipos de usuarios, adquisición de equipos o espacio de almacenamiento, aumento del personal de soporte, etcétera.
- Incorporar las acciones en los planes de trabajo para que sean realizadas al momento en que el sistema sea puesto en producción.

Acciones adicionales pueden ser: crear nuevas fases del proyecto para generar versiones que resuelvan parcial o totalmente las desviaciones en el mediano plazo, renegociaciones de contratos, entre otras.

EN RESUMEN

En este capítulo presentamos las consideraciones que deben tenerse una vez que el desarrollo del sistema procede a partir de una arquitectura. En primer lugar establecimos en qué consiste la implementación del *software* y la relación que tiene con la arquitectura.

A continuación definimos los conceptos de desviación y deuda técnica que influyen en la relación entre la implementación del *software* y la arquitectura de este.

Por el efecto adverso que tienen, las desviaciones deben evitarse, y por ello indicamos distintas estrategias que previenen o minimizan su aparición. Sin embargo, es posible que ocurran, por ello describimos las distintas prácticas para identificarlas durante la implementación del *software*.

De manera posterior planteamos las acciones necesarias tanto para resolver las desviaciones como para no hacerlo, dependiendo de su naturaleza.

PREGUNTAS PARA ANÁLISIS

1. Suponiendo que se tiene una arquitectura correcta y especificada de modo adecuado, ¿cuáles son los riesgos arquitectónicos más importantes durante la implementación?
2. ¿Qué tipo de actividades debe realizar el equipo de arquitectura durante la implementación?
3. Suponga una situación en la que el equipo de *arquitectura de software* concluye y entrega esta al equipo de implementación y luego se desentiende del proyecto. ¿Cuáles son los riesgos asociados a este comportamiento?
4. Considere una situación hipotética en la que un miembro del equipo de implementación encuentra un defecto arquitectónico, lo corrige a efectos de lo que él está realizando y continúa con su trabajo. ¿Cuáles son los riesgos relacionados con este comportamiento?
5. Suponga que bajo el criterio de aprender sobre la marcha, en un proyecto se contrata a un equipo de implementación que no tiene experiencia con los elementos descritos en la *arquitectura de software*. Se le indica implementar de inmediato el sistema sin apoyo alguno del equipo encargado de la *arquitectura de software*. ¿Cuáles son los riesgos asociados a este comportamiento?
6. ¿En qué situaciones no es posible el pago de la deuda técnica?
7. Suponga que una auditoría de *arquitectura de software* hecha a 20 módulos detecta que dos tienen desviaciones significativas respecto de esta. ¿Qué toma de decisiones sería la más adecuada?
8. Considere una situación hipotética en la que se hallan desviaciones significativas en relación con la arquitectura durante el diseño detallado. ¿Qué toma de decisiones sería la más adecuada?
9. Suponga que se detectan desviaciones significativas respecto de la arquitectura durante las pruebas de aceptación. ¿Qué toma de decisiones sería la más adecuada?

CAPÍTULO 7

ARQUITECTURA Y MÉTODOS ÁGILES: EL CASO DE *SCRUM*

En el contexto de la *ingeniería de software*, los métodos ágiles son un conjunto de métodos ligeros muy utilizados actualmente para soportar el desarrollo de sistemas. Su naturaleza ligera hace que sean poco prescriptivos, lo cual genera en muchos casos que los aspectos relacionados con el desarrollo de la arquitectura sean poco claros para quienes los practican. En este capítulo nos enfocamos en el *Scrum*, un método ágil muy popular ahora. Discutiremos en particular cómo actividades del ciclo de desarrollo de la arquitectura pueden realizarse en este contexto.

7.1 MÉTODOS ÁGILES

En la *ingeniería de software*, los métodos ágiles son métodos de desarrollo de sistemas con un enfoque iterativo e incremental. Sin embargo, en contraste con los métodos tradicionales que tienen este mismo enfoque, se caracterizan por equipos de personas multifuncionales (los integrantes tienen las habilidades necesarias), auto-organizados (quienes los integran requieren poca dirección), así como por iteraciones de desarrollo cortas en tiempo en las que siempre se produce una parte operable del producto final.

En el año 2001, antes de que se acuñara formalmente el término métodos ágiles, un grupo compuesto por críticos de los enfoques de desarrollo de *software* basados en procesos se reunió para definir las características que distinguen a los métodos ágiles de otros métodos. De esta reunión resultó el *Manifiesto ágil*, que contiene el conjunto de valores y principios que deben atender los métodos ágiles (Beck *et al.*, 2001). Los valores en el manifiesto son:

1. Individuos e interacciones por encima de procesos y herramientas.
2. *Software* funcionando por encima de documentación extensiva.
3. Colaboración con el cliente por encima de la negociación contractual.
4. Respuesta ante el cambio por encima de seguir un plan.

De estos cuatro valores se derivan los siguientes doce principios:

1. Satisfacer al cliente por medio de la entrega temprana y continua de *software* funcional.
2. Son bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo.
3. Entregar con frecuencia *software* funcional, prefiriendo periodos de semanas y no de meses.
4. Trabajo conjunto y cotidiano entre clientes y desarrolladores.
5. Construcción de proyectos en un ambiente con individuos motivados, dándoles la confianza y el respaldo que necesitan.
6. Comunicación cara a cara como forma eficiente y efectiva de comunicación.
7. El *software* funcional como principal medida de avance.
8. Procesos ágiles como medio para promover el desarrollo sostenido.
9. Atención continua a la excelencia técnica y al buen diseño.
10. Simplicidad como el arte de maximizar la cantidad de trabajo que no se hace.
11. Equipos de desarrollo auto-organizados.
12. Adaptación regular a circunstancias cambiantes.

Además de estos valores y principios en el manifiesto, los métodos ágiles se apoyan en diversas prácticas utilizadas durante el desarrollo de sistemas. Algunas relevantes incluyen el modelado dirigido por historias, programación en pares, desarrollo guiado por pruebas (Beck, 2002) o integración continua (Duvall, Matyas y Glover, 2007).

Como se nota, la filosofía de los métodos ágiles tiene mucha flexibilidad y dinamismo y difiere de aquella en la que el desarrollo de un proyecto realiza estimaciones, planes y diseños que, en general, son estables y generados desde las primeras etapas.

En la actualidad se observa un incremento en la adopción de métodos ágiles, además de que hay varios estudios en los cuales se reporta que su uso ha contribuido a reducir la complejidad y el riesgo presentes en varios métodos tradicionales de desarrollo (Cohn, 2009). De manera similar, se reporta que los métodos ágiles ayudan a lograr un mejor tratamiento de requerimientos cambiantes, aumentar la productividad de los equipos y mejorar la satisfacción de los clientes.

Existen varios métodos ágiles. La figura 7.1 muestra una gráfica con los más populares de acuerdo con un reciente estudio anual publicado por VersionOne (VersionOne, 2013). En el resto del capítulo nos enfocaremos en el que se reporta como el más popular: *Scrum*. Primero describiremos el método, y posteriormente discutiremos cómo actividades del ciclo de desarrollo de la arquitectura pueden llevarse a cabo en este contexto.

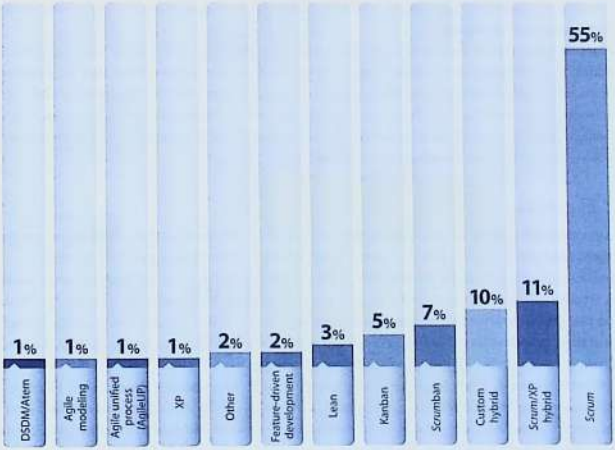


Figura 7-1. Los métodos ágiles más utilizados de acuerdo con el estudio anual publicado en 2014 por VersionOne

7.2 SCRUM

Scrum es un método ágil que puede aplicarse a casi cualquier proyecto para dar soporte a la administración de este. Es muy conocido en la actualidad por su uso en proyectos de desarrollo de *software*. Es en este contexto en el que se realiza nuestra descripción del método en este capítulo.

Como todo método ágil, *Scrum* permite de forma iterativa e incremental el desarrollo de *software*. Atendiendo al *Manifiesto ágil*, en *Scrum* un equipo multifuncional y auto-organizado crea de manera gradual un producto en varias iteraciones cortas. Cada iteración permite inspeccionar el rendimiento del equipo, así como el producto resultante, para luego, si es necesario, llevar a cabo oportunamente las adaptaciones requeridas.

Scrum define un conjunto pequeño de roles y un proceso que describimos en las siguientes secciones.

7.2.1 Los roles

Scrum define tres roles principales:

Propietario del producto (*product owner*): responsable de maximizar tanto el valor del producto que se está construyendo como el trabajo del equipo de desarrollo. Entre sus principales funciones están definir,

(re)priorizar y comunicar los requerimientos del producto, así como revisar y aprobar el trabajo y los resultados correspondientes en cada iteración. El propietario debe interactuar activa y regularmente con el equipo.

Equipo de desarrollo (*team*): grupo multifuncional y auto-organizado de personas encargadas de la construcción del producto. Entre sus principales responsabilidades está decidir el número de requerimientos a desarrollar en una iteración, así como la estrategia para llevarlos a cabo. El equipo se compone por lo habitual de entre cinco y nueve personas y no existen roles más específicos para los integrantes (por ejemplo el de programador o de diseñador).

Maestro *Scrum* (*Scrum master*): responsable de asegurar que el marco de trabajo de *Scrum* sea entendido y adoptado por todos los involucrados. Puede verse de esta forma como un facilitador para el propietario del producto y el equipo de desarrollo. En contraste con roles como el de líder de proyecto, que podrían parecer análogos, el maestro *Scrum* no indica al equipo qué se debe hacer en cada iteración. El maestro *Scrum*, sirve de ayuda al propietario y al equipo en el sentido que les ayuda a eliminar dificultades durante el proceso de desarrollo y promover el buen uso de prácticas ágiles de trabajo.

7.2.2 El proceso

El proceso de *Scrum* comprende un conjunto de iteraciones, las cuales tienen una duración fija, no pueden ser extendidas y se realizan una tras otra, sin interrupciones, hasta que el proyecto se considera terminado.

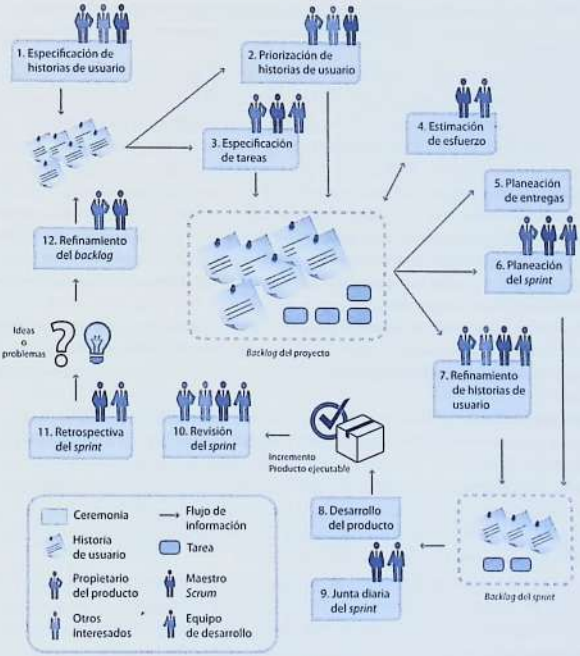
En *Scrum*, una iteración recibe el nombre de *sprint*, y su duración oscila por lo habitual entre una y cuatro semanas. En cada uno de ellos se lleva a cabo un conjunto de ceremonias organizadas en un patrón de actividades del tipo planeación-desarrollo-demostración-retrospectiva. Sin embargo, hay otras ceremonias que requieren realizarse antes del primer *sprint*. La figura 7.2, que describimos a continuación, muestra todas ellas, así como sus participantes, su secuencia y los principales artefactos producidos.

Como lo explicamos antes, *Scrum* define únicamente los roles de propietario del producto, equipo de desarrollo y maestro *Scrum*. Sin embargo, como se aprecia en la figura, en algunas ceremonias del proceso es productivo contar con la participación de otros interesados en el desarrollo del sistema (por ejemplo los usuarios finales). El maestro *Scrum* está presente en todas las ceremonias.

Al igual que cualquier proyecto de desarrollo de *software*, el proceso de *Scrum* inicia con la identificación de los requerimientos del sistema. El maestro *Scrum* se reúne con el propietario para especificarlos y priorizarlos (ceremonias 1, especificación de historias de usuario, y 2, priorización de historias de usuario), y en estas ceremonias podría requerirse la participación de otros interesados. En *Scrum* los requerimientos pueden ser determinados como historias de usuario, las cuales, como explicamos en el capítulo 2, son especificaciones cortas expresadas en el lenguaje del usuario final. La priorización de tales historias se orienta al propietario del producto debido a que, en esencia, se considera la relevancia de ellas en la satisfacción de los objetivos de negocio del sistema. De esta forma la priorización podría denotar, por ejemplo, el orden en el cual el propietario del producto desea que le sea entregada la funcionalidad.

El equipo de desarrollo toma en cuenta las historias de usuario a efecto de especificar una lista de tareas relacionadas (ceremonia 3, especificación de tareas). Esto se debe a que, para que a una historia se le considere como terminada, no solo hay que codificarla, sino que tiene tareas relacionadas como analizar o preparar la infraestructura necesaria para su implementación y pruebas, documentarla en los manuales correspondientes, generar su instalador en los equipos del cliente, etcétera. De ser necesario, el propietario del producto podría participar en la especificación de tareas. Las historias de usuario y tareas resultantes, como se ilustra en la figura 7-2, conforman lo que se conoce como *backlog* del proyecto.

En *Scrum*, el esfuerzo para completar una historia de usuario, y sus tareas asociadas, se especifica mediante *puntos de la historia* que son medidas relativas de complejidad que el equipo de desarrollo asigna en una ce-



© Humberto Cervantes Maceda / Pedro Velasco Elizondo / Luis Carlos Carriga Gutiérrez / Carriegh, PureSolutions, Spotlight media / Shutterstock

INSTITUTO TECNOLÓGICO DEL NORTE BIBLIOTECA

›Figura 7-2. Principales ceremonias y artefactos del proceso de *Scrum*

remonia posterior del *Scrum* (la 4, estimación de esfuerzo). Para el caso de las historias de usuario, la medida de esfuerzo debería establecerse considerando tareas de diseño, codificación y pruebas. Una vez estimado el trabajo requerido para completar los elementos del *backlog*, este podría ser revisado nuevamente por el propietario pues los resultados de la estimación podrían cambiar su visión acerca de las prioridades que él asignó antes, en la ceremonia 2.

Considerando la información en el *backlog* del proyecto en este punto, el propietario y el equipo de desarrollo acuerdan aspectos generales relacionados con las entregas (ceremonia 5, planeación de entregas). Esto es, acuerdan sobre cómo es que el Propietario del Producto irá obteniendo incrementalmente del equipo la funcionalidad esperada. Considerando el número de elementos en el *backlog*, así como las prioridades que tienen y el esfuerzo *requerido* para completarlos, también se establece la cantidad de *sprints* del proyecto y su duración, además de aspectos relacionados con el costo del proyecto.

Podemos decir que en este punto inicia un *sprint* de *Scrum*, y lo que se describe a continuación es un conjunto de ceremonias organizadas en el patrón de actividades planeación-desarrollo-demostración-retrospectiva que mencionamos antes.

La planeación consiste en que el propietario del producto y el equipo de desarrollo acuerdan qué elementos del *backlog* del proyecto se van a desarrollar en el primer *sprint* (ceremonia 6, planeación de *sprint*). De manera ideal, las historias de usuario y tareas relacionadas se seleccionan considerando la prioridad definida por el propietario y la complejidad de la implementación (denotado por la medida de esfuerzo determinada en la ceremonia 4). En *Scrum* es importante que el conjunto de elementos seleccionados pueda ser implementado en un *sprint*, lo cual se determina por lo habitual considerado la *velocidad* del equipo de desarrollo. La velocidad es una medida que establece el número de puntos de la historia completados por un equipo en un *sprint*. Si bien, la medida fluctúa en los *sprints* iniciales, tiende a estabilizarse después del tercero. Si la velocidad del equipo es menor al número de puntos de historia a completar en el primer *sprint*, se debe realizar una descomposición de las historias de usuario o tareas considerando una menor granularidad. Las historias y tareas a desarrollar en el *sprint* conforman lo que se conoce como *backlog* del *sprint*.

Antes de iniciar las actividades de desarrollo es necesario definir los criterios de aceptación de las historias de usuario. De esta forma, el propietario del producto y el equipo de desarrollo trabajan en la definición de las pruebas que deben realizarse a los elementos del *backlog* que las requieran (ceremonia 7, refinamiento de historias de usuario).

El desarrollo se refiere a la implementación del producto. Esto lo efectúa el equipo durante un periodo de varios días según la duración del *sprint* (ceremonia 8, desarrollo del producto). Como ya lo mencionamos, para el caso de las historias de usuario la implementación del producto incluye tareas de diseño, codificación y pruebas. Durante el desarrollo, el equipo se reúne diariamente para inspeccionar el progreso y, en su caso, realizar los ajustes necesarios para completar el trabajo restante en tiempo y forma (ceremonia 9, junta diaria del *sprint*). Si el equipo completara el trabajo antes de terminar el *sprint*, podría llevar a cabo más actividades, por ejemplo, completar el siguiente elemento más importante en el *backlog* del producto.

Una vez concluido el periodo de desarrollo del producto, inicia la demostración. El equipo se reúne con el propietario y, si es conveniente, otros involucrados relevantes, para revisar y demostrar lo que se ha construido (ceremonia 10, revisión del *sprint*). Como lo hemos mencionado, dependiendo del número del *sprint*, la revisión puede ser de un incremento funcional del producto, o bien, del producto final. Las historias de usuario aprobadas y las tareas completadas satisfactoriamente son removidas del *backlog* del producto.

Después de la demostración sigue la retrospectiva. El equipo discute sus impresiones acerca del trabajo realizado durante el *sprint* (ceremonia 11, retrospectiva del *sprint*). Como se ilustra en la figura 7-2, en esta reunión se genera un conjunto de ideas de mejora o problemáticas relacionadas con el trabajo realizado, que necesitan ser atendidas. Considerando esta información se realiza una actualización del *backlog* (ceremonia 12) la cual, como se muestra en la figura, podría comprender regresar historias de usuario del *backlog* del *sprint* al *backlog* del proyecto (si no fueron aprobadas por el propietario del producto) o generar historias de usuario nuevas. De manera similar, tareas nuevas podrían ser agregadas al *backlog* del proyecto. Sin embargo, se debe ser cuidadoso pues añadirle más elementos podría impactar de manera negativa en la fecha de término del producto establecida. Si fuera el caso, se debe considerar reducir el alcance de este.

Hasta este punto terminaría un *sprint*, y se puede iniciar uno nuevo considerando, en general, la secuencia de ceremonias a partir de la sexta. Más *sprints* se llevan a cabo hasta que se alcanza el número establecido de ellos para el proyecto y, de manera ideal, se entrega el producto final al propietario de este.

7.3 ¿GRAN DISEÑO AL INICIO O DEUDA TÉCNICA?

El proceso *Scrum*, y por lo habitual, cualquiera asociado a un método ágil, es por naturaleza ligero y poco detallado. Por ello se debe ser cuidadoso en su adopción, ya que podría generar confusiones importantes en sus practicantes, en especial si estos tienen poca experiencia en el diseño y el desarrollo de sistemas. Por ejemplo, el valor del *Manifiesto ágil* "*Software* funcionando por encima de documentación extensiva" podría ser interpretado como "No documentación". De forma similar, la ausencia tanto del rol de arquitecto como de una ceremonia explícita de diseño de arquitectura podría ser interpretada como que en *Scrum* no se hace trabajo arquitectónico. Esto sería incorrecto. Aunque los métodos ágiles promueven la comunicación cara a cara, el desarrollo incremental y las actividades de diseño "justo en tiempo" (*just in time*), no significa que las decisiones de diseño, incluidas las de arquitectura, deban ser tomadas de forma oportunista o poco deliberada.

Una percepción recurrente en el desarrollo ágil es que adoptar un enfoque *Big Design Up Front* (*discutido en la sección 3.6.2*), esto es, tener un diseño completo del sistema o de su arquitectura antes de comenzar con su codificación, es malo. En lugar de ello, en muchos métodos ágiles se prefiere un enfoque "diseño emergente", lo cual significa que el diseño del sistema va a ir "emergiendo" de su implementación. De esta forma, los practicantes de *Scrum* podrían evitar dedicar mucho tiempo para tomar decisiones de diseño, incluidas las de la arquitectura, y podrían preferir tomarlas "justo a tiempo".

En nuestra opinión, el enfoque de diseño emergente no es siempre adecuado o posible. Existen casos en los que los sistemas a construir tienen que ver con algo completamente nuevo para el equipo de desarrollo, y por ello no es tan evidente qué decisiones de diseño tomar de forma inmediata. Minimizar o ignorar la importancia de esta situación es riesgoso y podría ser causa de muchos problemas en el futuro.

En el contexto de la planeación de sistemas, el término deuda técnica es una metáfora utilizada para referirse a la "deuda" que el equipo de desarrollo adquiere, y debe asumir, al hacer omisiones durante el diseño de estos con el fin de acelerar la implementación para cumplir fechas de entrega o expectativas de los clientes. Una deuda técnica podría ser un costo alto (reflejado, por ejemplo, en salarios o tiempo requerido) para realizar cambios a un sistema a efecto de incluir una funcionalidad nueva, o bien, proveer esta misma a más usuarios sin degradar los tiempos de respuesta.

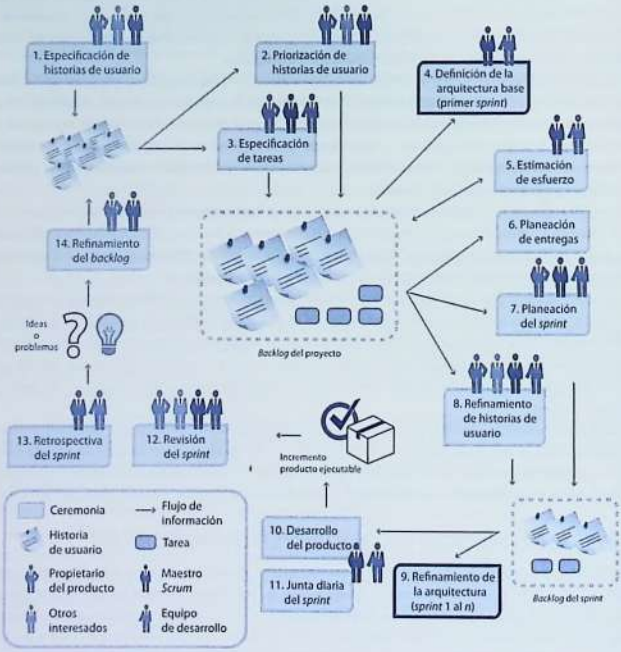
Aunque en la práctica la deuda técnica es inevitable debido a diversos factores, que van desde requerimientos cambiantes hasta la ignorancia del equipo de desarrollo respecto de que está adquiriendo una deuda de esa clase (lo cual puede ocurrir con equipos con poca experiencia), realizar acciones para minimizarla, o no hacerlas y asumir la deuda y "sus intereses", debe ser un aspecto a evaluar en todo proyecto de desarrollo, sea ágil o no.

7.4 DESARROLLO DE ARQUITECTURA EN *SCRUM*

En la actualidad, y en contraste con otras actividades incluidas en el proceso *Scrum*, no hay información muy establecida sobre cómo o en qué momento se deben realizar las actividades relacionadas con el desarrollo de arquitectura. Sin embargo, a partir de la información pública de muchos practicantes de *Scrum*, así como de experiencias propias, en las siguientes secciones describiremos en el contexto de este método ágil una instancia de un enfoque de "diseño planeado incremental", y aspectos de soporte relacionados. Este enfoque difiere del de diseño emergente descrito en la sección anterior. En este contexto usamos la palabra "planeado" para denotar que el diseño se define de manera deliberada antes de la implementación, mediante la toma de decisiones reflexionadas. Empleamos la palabra "incremental" para denotar que el diseño se define e implementa gradualmente hasta que el producto está terminado; por ello las actividades iniciales de esta etapa no requieren un diseño completo. En esta instancia hemos considerado la inclusión de algunas ceremonias adicionales en el proceso original de *Scrum*.

7.4.1 Soporte de un enfoque de diseño planeado incremental

La figura 7.3 muestra una versión adaptada del proceso *Scrum*, discutido en la sección 7.2.2, en la cual hemos incluido dos ceremonias que soportan el desarrollo de arquitectura adoptando un enfoque "diseño planeado incremental": 4. Definición de la arquitectura base, y 9. Refinamiento de la arquitectura. Como puede apreciarse, ambas ceremonias se llevan a cabo por el equipo de desarrollo.



© Humberto Cervantes Maceda / Perla Velasco Elizondo / Luis Castro Carvajal
Ilustraciones: Cuéllar & Asociados, Spin Estudio / Shutterstock

› Figura 7-3. Proceso de *Scrum* con ceremonias de desarrollo de arquitectura

A continuación describimos los detalles de estas ceremonias.

Ceremonia 4. Definición de la arquitectura base

La primera ceremonia propuesta, la número 4, es una obligatoria cuyo objetivo principal radica en la toma de decisiones orientadas a definir un diseño arquitectónico base que pueda ser refinado posteriormente. Durante ella es preferible utilizar conceptos de diseño existentes en vez de "reinventar la rueda". Sin embargo, y como lo mencionamos en el capítulo 3, lo anterior no debiera ser una limitante a la creatividad durante el diseño. Como se observa en la figura, esta ceremonia se halla fuera de lo que en *Scrum* se considera un *sprint*, por lo tanto, solo se realiza una vez.

Con el propósito de no perder agilidad se recomienda que durante esta ceremonia el equipo de desarrollo se enfoque únicamente en las siguientes tareas:

1. Seleccionar estilo (o patrón) arquitectónico o una arquitectura de referencia para la estructuración general del sistema, instanciar los elementos del patrón o arquitectura de referencia y asignarles responsabilidades a dichos elementos.
   Esta tarea se debe llevar a cabo considerando el dominio de aplicación y/o el tipo de sistema a desarrollar. Por ejemplo, el patrón arquitectónico de capas es utilizado con frecuencia para estructurar sistemas *web* (Microsoft, 2009). De esta forma, es común instanciar el patrón con tres capas asignando responsabilidades a cada una de ellas como sigue: una capa permite la solicitud de los servicios, otra proporciona estos, y una más soporta la persistencia de datos que necesitan los servicios.
2. Seleccionar una estrategia de implantación para el patrón arquitectónico seleccionado.
   La tarea hace referencia a una estrategia que permita la instalación y puesta en marcha de los elementos del patrón de arquitectura seleccionado. Por ejemplo, una estrategia de implantación que podría utilizarse en sistemas *web* estructurados en tres capas es el de ubicar estas como sigue: la capa que soporta la solicitud de los servicios, en un equipo tipo cliente ligero; la que proporciona los servicios solicitados, en un servidor de aplicaciones *web*, y la que soporta los servicios de persistencia de datos, en un servidor de datos.
3. Realizar una selección inicial de tecnologías para soportar la implementación del patrón arquitectónico seleccionado.
   Por ejemplo, la capa que soporta los servicios de persistencia de datos podría estar implementada en *Hibernate*[1], que es un *framework* de mapeo objeto-relacional para la plataforma Java que facilita mapear entre una base de datos relacional tradicional y el modelo de objetos de un sistema.
4. Documentar la arquitectura resultante considerando las decisiones tomadas.
   Se recomienda documentar aspectos lógicos (de estructuración del sistema) y físicos (de implantación). De manera ideal, la información sobre la selección tecnológica se debe añadir a estos diagramas.
5. Verificar el diseño generado.
   Esta tarea se encarga de verificar que el diseño:
   a) Permite realizar las historias de usuario con mayor valor de negocio (denotado generalmente por su prioridad). Las historias a las que nos referimos aquí podrían corresponder en naturaleza a los *drivers* de requerimientos de usuario que discutimos en el capítulo 2.
   b) Atiende las restricciones de diseño identificadas. Las restricciones a las que nos referimos aquí podrían corresponder en naturaleza a los *drivers* de restricciones que discutimos en la sección 2.1.7.3.
   c) Atiende principios generales de diseño como modularidad, cohesión alta, acoplamiento bajo y simplicidad que discutimos en el capítulo 3, sección 3.3.
   d) Promueve el desarrollo paralelo e incremental del sistema.

[1] www.hibernate.org

Estas decisiones son muy similares a lo que mostramos en la iteración inicial del caso de estudio del apéndice.

Es importante notar que esta ceremonia precede intencionalmente a la de estimación de esfuerzo (ceremonia 5). En la sección 7.2.2 explicamos que esta estimación se realiza asignado puntos de historia a las historias de usuario y tareas del *backlog* del proyecto. Si bien, en el contexto de *Scrum* no se espera tener una estimación de esfuerzo exacta en los primeros *sprints*, consideramos que contar con un diseño inicial de la arquitectura podría contribuir a mejorarla. Conocer la naturaleza de los elementos arquitectónicos contribuye a efectuar estimaciones de complejidad más precisas. De manera similar, diseñar la arquitectura involucra con frecuencia la reutilización de decisiones de diseño. Tener esta información explícita permite hacer estimaciones basadas en datos de proyectos anteriores.

Ceremonia 9. Refinamiento de la arquitectura

La segunda ceremonia propuesta, la número 9, tiene como objetivo la toma de decisiones orientadas a refinar el diseño arquitectónico para completar las historias de usuario en el *backlog* del *sprint*. Nuestra estrategia para no perder la agilidad es plantear esta como opcional durante los *sprints* 1 al n. La decisión de realizarla o no se basa en la información disponible sobre la complejidad técnica de cada historia de usuario en el *backlog* del *sprint* y los criterios de aceptación asociados a esta (información generada en las ceremonias 5 y 8, respectivamente).

En nuestra experiencia, una historia de usuario con una medida alta de complejidad tiene asociados por lo habitual criterios de aceptación para atributos de calidad que difieren de los que comúnmente se presentan en el dominio de aplicación del sistema a desarrollar. Esto hace evidente en muchos casos un reto técnico que requiere una solución de diseño más reflexionada. Como lo discutimos antes, tomar una decisión de diseño apresurada para resolver un reto técnico representa un riesgo que podría ser minimizado al efectuar actividades de la arquitectura en ese respecto.

Si el equipo de desarrollo decide realizar esta ceremonia, se recomienda que para cada historia de usuario con alta complejidad se enfoque en las siguientes tareas:

1. Seleccionar uno o más conceptos de diseño que satisfagan los criterios de aceptación correspondientes. Si aplica, instanciar el o los conceptos, y asignar responsabilidades en los elementos correspondientes.
2. Si el concepto seleccionado requirió generar elementos nuevos en la arquitectura, actualizar la documentación actual. Actualizar, si es necesario, la información sobre la selección tecnológica.
3. Verificar el diseño generado. En específico, verificar que las decisiones de diseño tomadas no afectan en:
   a) Realizar las historias de usuario con mayor valor de negocio.
   b) Atender las restricciones de diseño identificadas.
   c) Atender principios generales de diseño como modularidad, cohesión alta, acoplamiento bajo y simplicidad.
   d) Promover el desarrollo paralelo e incremental del sistema.

7.4.2 Especificación de atributos de calidad y restricciones

En el capítulo 2 comentamos que los atributos de calidad junto con las restricciones son los requerimientos que tienen mayor influencia sobre el diseño de la *arquitectura de software*. Esto aplica para arquitecturas diseñadas en un contexto ágil o no; por ello se hace uso de esta información en las ceremonias propuestas anteriormente. En esta sección damos algunas recomendaciones sobre cómo documentar esta información y utilizarla para beneficio de otras ceremonias, además de las propuestas para el diseño de la arquitectura.

Durante la ceremonia de especificación de historias de usuario (ceremonia 1) es muy importante que el maestro *Scrum* identifique información sobre atributos de calidad y restricciones asociadas al producto de *software*. Sin embargo, para no perder agilidad, solo debería documentar la información relevante para la crea-

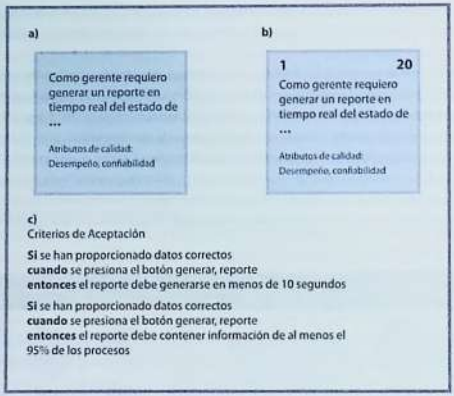
ción de la arquitectura. Mencionamos anteriormente que esta noción de relevancia se asocia al hecho de que esos atributos y restricciones representen un reto técnico para el equipo de desarrollo durante las actividades de implementación.

Como explicamos en el capítulo 2, las restricciones describen aspectos que limitan las decisiones que se pueden tomar para diseñar y desarrollar el sistema; incluyen con frecuencia requerimientos sobre el uso de productos de *software* y de *hardware*, métodos de diseño o implementación, o lenguajes de programación específicos. Es importante no solo documentar esta información, sino mantenerla siempre visible para todo el equipo de desarrollo, pues las restricciones aplican al sistema entero. El uso del tablero Kanban es una práctica frecuente en *Scrum* para mantener a la vista el estado actual del proyecto respecto del trabajo terminado, el trabajo en proceso y el trabajo por realizar (Cohn, 2009). En la figura 7-4 se presenta el boceto de un tablero Kanban con una columna para las restricciones como medio para mantenerlas observables a los miembros del equipo de *Scrum*.

Ilustraciones: © Gumenyk / Shutterstock

› Figura 7-4. Tablero Kanban con información sobre restricciones

Por otra parte, la información sobre atributos de calidad podría documentarse en las historias de usuario correspondientes escribiendo en la parte inferior el nombre o nombres del atributo a manera de recordatorio. Durante la ceremonia 5 puede utilizarse esta información como recordatorio de que la historia de usuario involucra un reto técnico que requiere una solución de diseño más reflexionada. De esta forma, la medida de complejidad asignada a la historia debería reflejar esta situación, como se muestra en la figura 7-5 b. Durante la ceremonia 7 tanto el recordatorio de atributo de calidad como la medida de complejidad de la historia de usuario son de utilidad para el equipo de desarrollo para soportar la definición de las pruebas relacionadas a tales atributos, lo cual se ilustra en la figura 7-5 c.



© Humberto Cervantes Maceda y Perla Velasco-Elizondo / Luis Cuéllar Enrique

›**Figura 7-5**. Propuesta para documentar atributos de calidad en historias de usuario

7.4.3 ¿Vistas de arquitectura?

En el capítulo 4 explicamos que, en los métodos de desarrollo tradicionales, las *vistas* son artefactos comunes para documentar la arquitectura de un sistema. Una vista describe una o más estructuras arquitectónicas en términos de los elementos que las conforman. Una vista se conforma, en términos generales, por: 1) un diagrama en donde se representan los elementos de la estructura, y 2) información textual que ayuda a la comprensión de dicho diagrama.

En un contexto ágil la generación de vistas no es una actividad carente de valor. Por el contrario, como lo hemos discutido antes, disponer de documentación apropiada de la arquitectura facilitará la comprensión por parte del equipo de desarrollo de qué partes tiene el sistema que van a construir y cuáles son las decisiones de diseño relevantes a la construcción de estas partes. Sin embargo, en un contexto ágil es importante cuidar que la documentación arquitectónica contenga el nivel necesario de información para soportar el *sprint* en curso y evitar incluir aquella que no aporta valor en el desarrollo de dicho *sprint*.

Tomando en cuenta la forma de trabajo de los equipos ágiles, es preferible adoptar un enfoque de documentación simple y efectivo. Por ejemplo, es recomendable optar por el uso de notaciones informales para elaborar los diagramas. De manera similar, se sugiere que los diagramas correspondientes a la arquitectura base (generados en la ceremonia 4), al igual que artefactos como los *backlogs*, estén siempre visibles para el equipo. Con ello, durante el desarrollo del sistema los diagramas pueden utilizarse como puntos de referencia en las discusiones sobre las decisiones de diseño específicas que deben aplicarse para el refinamiento de la arquitectura durante los *sprints*.

En un entorno ágil la documentación debería evolucionar a la par de la arquitectura y el desarrollo del sistema. Sin embargo, según nuestra experiencia, no siempre es necesario que los bocetos resultantes del refinamiento arquitectónico (generados en la ceremonia 9) estén siempre visibles para el equipo que desarrolla. Sin embargo, sí son artefactos de utilidad durante las retrospectivas y, en algunos casos, en las juntas diarias de *sprint*.

7.4.4 El arquitecto de *software*

Ya hemos explicado antes que el arquitecto de *software* es el responsable de definir, documentar, mantener y vigilar la implementación del diseño de la arquitectura. De manera ideal, en el contexto de *Scrum* estas responsabilidades deberían ser compartidas por varios miembros del equipo de desarrollo. Sin embargo, puede darse el caso en que una sola persona asuma este rol. De ser así, ella podría fungir como asesor o facilitador en materia de desarrollo de arquitectura, dejando al equipo las principales responsabilidades sobre esta.

Es importante notar que en un contexto como el de *Scrum* todo miembro del equipo que realiza actividades de arquitectura debería ser capaz también de llevar a cabo otras funciones propias de este rol. Con esto queremos decir que, por ejemplo, un arquitecto que no codifica es contradictorio y, en consecuencia, poco común en el contexto de *Scrum*.

EN RESUMEN

En este capítulo abordamos el tema de los métodos ágiles. Describimos el proceso que sigue *Scrum*, el cual es un método ágil muy popular actualmente, y discutimos cómo actividades del ciclo de desarrollo de la arquitectura pueden realizarse en este proceso.

Planteamos en específico el rol de arquitecto de *software* y la forma en que la arquitectura se desarrolla en *Scrum*. En este método ágil, este rol se comparte por lo habitual por varios miembros del equipo de desarrollo. Cuando se asume por un solo miembro, este funge únicamente como asesor o facilitador y deja al resto del equipo las principales responsabilidades sobre la misma.

Respecto de la forma en que en *Scrum* se desarrolla la arquitectura describimos un proceso, adaptado a partir del utilizado comúnmente, el cual define dos ceremonias orientadas a tal desarrollo.

PREGUNTAS PARA ANÁLISIS

1. En el contexto de *Scrum* para el desarrollo de sistemas de *software*, discuta con un(a) compañero(a) la relevancia de contar con una arquitectura y su documentación para: i) mejorar la comunicación al interior del equipo de desarrollo; ii) preservar la información sobre la arquitectura; iii) guiar la codificación del sistema, y iv) proveer un lenguaje común entre este equipo, el propietario del producto y el maestro *Scrum*.
2. Hay una frase popular que dice algo así: "...usted no necesita un diseño para construir una casa para un perro, pero es mejor tener un poco un diseño si usted va a construir un rascacielos". En el contexto del desarrollo ágil, discuta con un(a) compañero(a) cuándo esta frase es apropiada para el diseño de arquitectura. Justifique su respuesta.
3. Considere que, utilizando *Scrum*, una compañía de desarrollo de sistemas va a crear uno de análisis automático de comentarios escritos en las redes sociales por los clientes de una cadena de restaurantes. ¿Qué decisiones de diseño deberían tomarse antes de iniciar el primer *sprint*?, ¿cuáles se deberían documentar y comunicar al equipo de desarrollo? y ¿cuáles serían las consecuencias de no hacerlo?

APÉNDICE

CASO DE ESTUDIO

SECCIÓN 1
INTRODUCCIÓN

En este apéndice presentaremos un caso de estudio que muestra la manera en que se llevan a cabo las actividades relacionadas con el ciclo de desarrollo de la arquitectura. Para dicho caso no se asume una metodología de desarrollo en particular, sin embargo, se asume que ya se tiene elaborado un documento de visión y alcance para el sistema. En el documento se establece de manera temprana en el desarrollo y se incluye aquí porque contiene información de contexto similar a la que se presenta durante el taller de atributos de calidad (QAW), visto en la sección 2.3.1 del capítulo 2, en los pasos 2 y 3 los cuales corresponden a la presentación de negocios y a la presentación del boceto de la arquitectura, respectivamente.

La presentación de negocios corresponde a la sección 4 del documento de visión y alcance (Contexto de negocio), y la presentación del boceto de la arquitectura corresponde al diagrama de contexto de la sección 5.2 de este apéndice. Otros puntos relevantes de este documento en relación con la arquitectura son los siguientes:

- **Sección 3.2. Características del sistema.** Lista los requerimientos de alto nivel del sistema, tanto funcionales como no funcionales, además de sus prioridades.
- **Sección 5.3. Entorno de operación.** Provee diversas restricciones.
- **Sección 4. Alcance.** Proporciona información acerca de los requerimientos que se van a incluir en cada entrega.

A continuación se presenta el documento.

DOCUMENTO DE VISIÓN Y ALCANCE

1. INTRODUCCIÓN

El presente documento describe la visión y el alcance del sistema. Permite establecer el acuerdo inicial con el cliente acerca del desarrollo que se va a realizar.

2. CONTEXTO DE NEGOCIO

2.1 Antecedentes

La empresa Autotransportes de México (ADM) es líder en el transporte de pasajeros en México. En la actualidad opera varias rutas de autobuses hacia varios estados del país.

2.2 Fase del problema

A pesar de que ADM es hoy en día el líder del mercado de autotransporte en algunos estados, existe el riesgo de que sus competidores capten una mayor parte de este, dado que la compra de boletos de viaje se realiza únicamente en puntos de venta físicos (taquillas de las terminales de autobús y módulos dispersos en centros comerciales).

La competencia de ADM ha estado trabajando en una estrategia de modernización de sus esquemas de ventas y, por lo tanto, en la actualidad ofrecen la posibilidad de adquirir boletos en línea. Por lo anterior, la mesa directiva de ADM ha decidido impulsar el proyecto de creación de un sistema de reservación y venta de sus boletos de forma electrónica. Adicionalmente, la compañía desea entablar relaciones de negocio con empresas dedicadas a la venta de paquetes de viaje todo pagado (VTP) que requieran de transporte vía terrestre.

Hoy en día las empresas VTP no pueden adquirir de forma sencilla boletos de autobús de ADM, por lo cual es necesario proporcionar mecanismos que faciliten la integración del sistema de boletaje de ADM a los sistemas de estas.

2.3 Objetivos de negocio

ID	Descripción del objetivo de negocio
ON-1	Incrementar en 50% las ventas de boletos mediante la implantación de un sistema de reservación y venta vía *web* y dispositivos móviles.
ON-2	Proporcionar mecanismos, los cuales permitan integrar el sistema a terceros que requieran la reservación y compra de boletos a más tardar en 12 meses.
ON-3	Colectar datos con el fin de mejorar continuamente el servicio 4 meses después de su puesta en marcha.
ON-4	Liberar el sistema antes del 31 de diciembre de 2015

3. VISIÓN DE LA SOLUCIÓN

3.1 Fase de visión

RedADM será un sistema moderno de boletería. Permitirá realizar la consulta de rutas, la reservación y compra segura de boletos por medio de un navegador *web* en un primer momento, y con dispositivos móviles después. El sistema proporcionará, además, mecanismos que permitan a terceros, tales como las empresas de ventas de VTP, interactuar con este.

3.2 Características del sistema

ID	Descripción	Prioridad	Objetivo de negocio asociado
CAR-01	El sistema debe permitir realizar la compra de boletos desde un navegador web.	Alta	ON-1
CAR-02	El sistema debe permitir realizar la impresión y re-impresión de boletos.	Alta	ON-1
CAR-03	El sistema debe permitir realizar consulta de rutas y horarios de corridas.	Alta	ON-1
CAR-04	El sistema debe permitir realizar cancelación de boletos.	Alta	ON-1
CAR-05	El sistema debe permitir a un administrador generar reportes de actividad de usuarios y sistemas externos.	Media	ON-3
CAR-06	El sistema debe exportar servicios *web* que permitan a aplicaciones externas realizar las operaciones de consulta, compra y cancelación de boletos.	Media	ON-2
CAR-07	El sistema debe soportar de manera simultánea por lo menos cien interacciones (usuarios o sistemas externos)	Alta	ON-1
CAR-08	El sistema debe garantizar la seguridad de los datos que se ingresan.	Alta	ON-1
CAR-09	El sistema debe estar disponible 24 horas al día y, en caso de falla, retomar actividad en, máximo, cinco minutos.	Media	ON-2
CAR-10	El sistema debe integrarse con redes sociales (*Facebook*, *Twitter* y *Foursquare*)	Baja	ON-1
CAR-11	El sistema debe permitir dos niveles de usuario: cliente y administrador.	Alta	ON-1
CAR-12	El sistema debe permitir realizar pagos con tarjeta de crédito conectándose a un servicio de terceros.	Alta	ON-1

4. ALCANCE

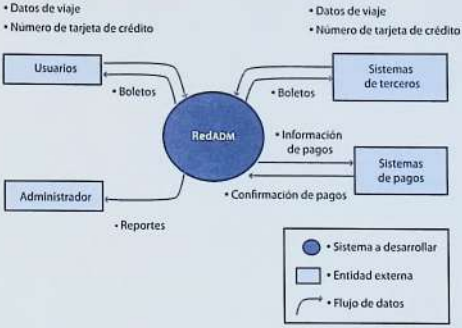
Número de entrega	Tema principal	ID de características a incluir
1.0	Funcionalidad básica.	CAR-01, CAR-02, CAR-03, CAR-04, CAR-07, CAR-08, CAR-11, CAR-12
2.0	Estabilidad del sistema e integración con terceros.	CAR-05, CAR-09, CAR-06
3.0	Visibilidad.	CAR-10

5. CONTEXTO DEL SISTEMA

5.1 Interesados

Nombre	Descripción	Responsabilidades
Juan Pérez (dueño del sistema)	Director de operación.	• Aprobar la visión y el alcance del sistema. • Proporcionar acceso a instalaciones y a procedimientos existentes. • Aprobar entregas del proyecto.
Martha Ramírez	Representante de usuarios.	• Proveer y validar requerimientos de interacción de usuarios finales con el sistema. • Validar prototipos. • Mostrar la manera en que se llevan a cabo las tareas que serán automatizadas.
Francisco Gómez	Representante de terceros.	• Dar requerimientos para definir interfaces de servicio. • Proporcionar entorno de pruebas de servicios y apoyar en la realización de las mismas.
Ramón Salas	Administrador del sistema.	Proporcionar y validar requerimientos relacionados con la administración y las necesidades de recolección de datos del sistema.
Sandra López	Administradora de la base de datos.	Brindar información y acceso a la base de datos (BD) existente.
Iván González	Representante de infraestructura.	• Apoyo en la implantación del sistema. • Apoyo para proveer acceso externo al sistema.
Raúl Ochoa	Líder de proyecto.	Coordinar el proyecto y representar al equipo de desarrollo.

5.2 Diagrama de contexto



© Humberto Cervantes Maceda / Perla Velasco Elizondo / Luis Castro Carranza

5.3 Entorno de operación

El sistema será usado desde navegadores *web* y deberá soportar los siguientes:

- Internet Explorer 10+
- Firefox 10+
- Google Chrome 17+

Los dispositivos móviles siguientes deberán ser soportados:

- iPhone/iPod Touch, iOS 6+
- Android 4+

Se deberá descartar el uso de *flash* y *applets*. El sistema se ejecutará en un servidor que será adquirido para albergar la aplicación.

Se debe usar el servidor de base de datos legado con Oracle v11.2.

La integración con sistemas externos, incluyendo el sistema de autorización de pagos, se lleva a cabo mediante servicios *web*.

6. INFORMACIÓN ADICIONAL

Los criterios de aceptación del proyecto incluyen:

- Sistema instalado, y funcionando, en la sede de ADM en nueve meses.
- Servicios *web* publicados y accesibles, a mas tardar en 12 meses, que permitan el acceso al sistema a terceros.
- Protocolo de pruebas de aceptación aprobado al 100%.
- Entregados juntos tanto el código fuente como el manual técnico y de administración.

SECCIÓN 2

REQUERIMIENTOS DE LA ARQUITECTURA

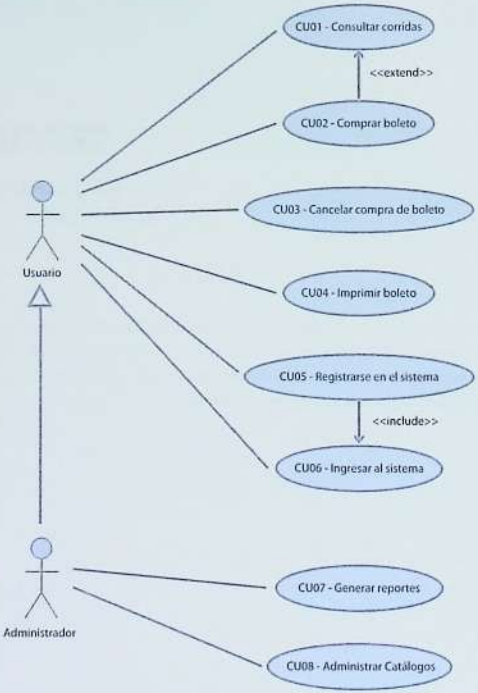
Una vez que el desarrollo de un sistema es aceptado y la visión y el alcance han sido acordados y validados por el cliente, comienza por lo habitual la realización de actividades relacionadas con la ingeniería de requerimientos. En este caso de estudio nos enfocaremos en los requerimientos que influyen directamente en la arquitectura, es decir, los *drivers* que incluyen requerimientos funcionales, atributos de calidad y restricciones.

2.1 *Drivers* funcionales

Para este caso de estudio, los requerimientos funcionales que especifican las interacciones entre los usuarios y el sistema son especificados usando la técnica de casos de uso (Cockburn, 2001).

2.1.1 Modelo de casos de uso

En general, los casos de uso del sistema se derivan de las caracteristicas que se describen en el documento de visión y alcance, y una de estas puede dar lugar a uno o más de ellos. A continuación se presentan tanto el modelo de casos de uso para nuestro sistema, usando un diagrama de UML, como una descripción breve de los mismos.



© Humberto Cervantes, Ricardo Peña Mena y Raúl Gabriel Ciuffoli

ID	Descripción	Característica asociada
CU-01	Permite consultar corridas de autobuses con base en diversos criterios (tipos de boleto, fechas de salida y de regreso, orígenes y destinos).	CAR-01 y CAR-06
CU-02	Permite comprar uno o más boletos, con o sin descuento, una vez que se ha realizado la consulta de corridas y se muestran los resultados de la misma.	CAR-01 CAR-06
CU-03	Permite cancelar un boleto comprado.	CAR-04
CU-04	Permite imprimir o reimprimir un boleto comprado.	CAR-02
CU-05	Permite darse de alta en el sistema con el fin de realizar la compra e impresión de boletos.	CAR-11
CU-06	Permite ingresar al sistema, una vez que el usuario se ha dado de alta, a efecto de hacer consultas, reimprimir boletos, o para cuestiones administrativas.	CAR-11
CU-07	Permite generar diversos tipos de reportes para análisis de comportamientos (y mejora del sistema).	CAR-05
CU-08	Permite realizar altas, bajas y cambios de las diversas entidades del sistema (rutas, autobuses, corridas, etcétera). Nota: en realidad este caso representa a múltiples casos de uso.	CAR-03

2.1.2 Elección de casos de uso primarios

Los casos de uso primarios se describen en la sección 2.2.1, y para este sistema son los siguientes:

ID	Criterios de elección (justificación)
CU-02	La compra de boletos es la razón primaria de ser de este sistema (CN-1).
CU-01	Es necesario disponer de la funcionalidad de consulta de corridas para poder comprar los boletos. Por otro lado, este caso de uso debe soportar el acceso concurrente y otros atributos de calidad descritos en la siguiente sección.

2.2 *Drivers* de atributos de calidad

En esta sección se describen los atributos de calidad relevantes para el sistema, los cuales se identificaron usando la técnica de escenarios (véase la sección 2.3.1, capítulo 2), y su priorización. La priorización se realiza en este caso considerando dos dimensiones que pueden tomar valores alto, medio y bajo: la importancia para el cliente/negocio y la dificultad de implementación. De acuerdo con la priorización, los atributos de calidad más relevantes son EAC-01 y EAC-02.

El primer escenario de atributo de calidad se muestra desglosado en sus seis partes, y los demás se describen de forma directa.

ID	Categoría	Escenario	Prioridad
EAC-01	Desempeño	**Fuente de estímulo:** un usuario **Estímulo:** selecciona la opción "buscar" una vez que ha elegido las ciudades de salida y destino, así como las fechas de su viaje **Artefacto:** en la pantalla de consulta (CU-01). **Entorno:** en un momento normal de operación y hay cien usuarios conectados al sistema. **Respuesta:** el sistema procesa la petición y muestra la lista de corridas. **Medida de la respuesta:** en un tiempo no mayor a dos segundos.	(A, A)
EAC-02	Disponibilidad del sistema	Ocurre una falla interna del sistema que detiene su operación en un momento en que el estaba funcionando de manera normal. El sistema vuelve a operar normalmente en un periodo no mayor a cinco minutos.	(M, A)
EAC-03	Seguridad	Un atacante intercepta la comunicación entre la PC de un usuario y el sistema mientras tal cliente realiza la compra de un boleto (CU-02) en un momento normal de operación. El atacante no obtiene ninguna información en claro (no cifrada).	(A, B)
EAC-04	Usabilidad	Un usuario captura con errores los campos de un formato en el navegador mientras hace la compra de un boleto (CU-02), en un momento normal de operación. El sistema resalta los campos llenados erróneamente y describe instrucciones de llenado.	(M, A)
EAC-05	Facilidad de prueba	Un probador realiza examinaciones unitarias de los componentes del sistema encargados de la lógica de negocio y la persistencia de datos en tiempo de desarrollo. El total de los componentes pueden ser probados de forma unitaria sin modificaciones en el código de estos componentes.	(B, B)
EAC-06	Modificabilidad	Un desarrollador cambia el idioma de la interfaz de usuario del sistema en tiempo de desarrollo/ mantenimiento. El nuevo idioma se agrega de manera exitosa sin requerir el cambio y la recompilación del código.	(M, B)

2.3 *Drivers* de restricciones

A continuación se presenta la lista de restricciones asociadas al sistema. Se pueden identificar dos categorías de estas: 1) las que expresa el cliente de forma explícita, que en este caso se derivan del documento de visión y alcance, y 2) las que son parte de la organización de desarrollo y que pueden no aparecer en este documento.

Tipo de restricción	Descripción
Del cliente	• Tiempos de entrega y presupuestos establecidos • Véase la sección 5.3 del documento de visión y alcance
De la organización de desarrollo	Los ingenieros de los que se dispone están familiarizados con los siguientes *frameworks* de Java: Java Server Faces (JSF), *Spring* e *Hibernate*[1]

[1] http://www.javaserverfaces.org/
http://projects.spring.io/spring-framework/
http://hibernate.org/

SECCIÓN 3

DISEÑO DE LA ARQUITECTURA

En esta sección se describe la manera en que se realiza el diseño de la arquitectura del sistema cuyos *drivers* fueron listados previamente. Para este caso de estudio, el diseño se lleva a cabo de acuerdo al método ADD (ver sección 3.6.1 del capítulo 3) y por ello se presenta el diseño por medio del resumen de varias iteraciones. Se asume que el paso 1 del ADD (Confirmar que hay suficiente información acerca de los *drivers* o requerimientos) ya ha sido completado (es la información de la sección anterior), y el diseño comienza en el paso 2 en cada una de las iteraciones.

3.1 Primera iteración: estructuración general del sistema

En la primera iteración, y como parte del paso 2 del ADD, el arquitecto elige el sistema como elemento a descomponer, dado que se está creando un sistema partiendo de cero (del único punto de partida arquitectónico que se dispone es del diagrama de contexto en el documento de visión y alcance).

En el paso 3 identifica los *drivers* que va a tratar durante la iteración. Dado que se trata de una iteración inicial, el enfoque está en la estructuración general del sistema con el fin de soportar los distintos *drivers* y apegarse a las restricciones.

Una vez que el arquitecto tiene claros los *drivers* procede, en el paso 4 del ADD, a elegir conceptos de diseño para descomponer el elemento elegido anteriormente. En este caso, estos conceptos son patrones de diseño estructurales y de implantación: capas y *N*-tercios.

Ya hecha la elección, en el paso 5 se generan elementos nuevos mediante la instanciación de los patrones y se les asignan responsabilidades. Ello da como resultado estructuras que en este caso son tanto físicas como lógicas.

En el paso 6 se definen interfaces para los elementos instanciados. Sin embargo, dado que se trata de una descomposición estructural de nivel alto, estas no se detallan todavía.

Resumen de iteración		
Elemento a descomponer	El sistema (iteración inicial).	
***Drivers* elegidos para la iteración**	Dado que es la iteración inicial, el arquitecto debe proponer una estructuración general del sistema. Para hacerlo toma en cuenta el conjunto de casos de uso primarios y escenarios de atributos de calidad, así como las siguientes restricciones (véase sección 5.3 del documento de visión): • Uso desde navegadores web y dispositivos móviles. • Integración de sistemas externos por medio de servicios web. • Integración de servidor de correo legado.	
Conceptos de diseño	**Concepto**	**Justificación**
	Estilo o patrón arquitectónico de capas (Fowler, 182)	Las capas permiten aislar de forma lógica distintas responsabilidades del sistema: los aspectos relacionados con la interacción con el usuario (capa de presentación), con la conexión de sistemas externos (capa de integración), el manejo de la lógica de negocio (capa de negocio) y la persistencia de los datos (capa de datos). Esto permitirá hacer cambios en la interfaz de usuario y asignar componentes a los ingenieros de manera más simple.
	Estilo o patrón arquitectónico de *N*-tercios.	Con la elección de este estilo se estructura el sistema de un punto de vista de implantación pues las capas de la aplicación se ubican en distintos "tercios" (o nodos físicos). Esto permite aumentar la seguridad y facilita escalar el sistema y su mantenimiento.
	Frameworks JSF[3], *Spring*[4] e *Hibernate*[5].	Estos se apegan a las capas definidas y son aquellos con los que el equipo de desarrollo está familiarizado.

Continúa...

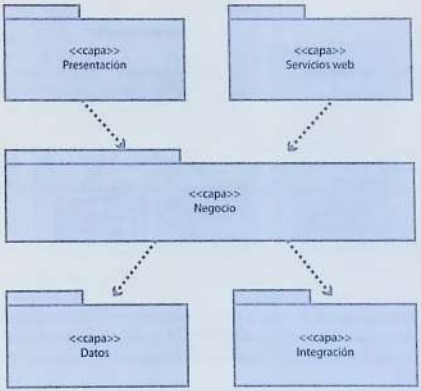
[3] Hemos optado por dejar el nombre en inglés y la página del libro correspondiente (Buschmann, Henney y Schmidt, 2007).
[4] http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
[5] http://projects.spring.io/spring-framework/
[6] http://hibernate.org/

Continuación...

Resumen de iteración

Estructuras resultantes y responsabilidades de los elementos

Perspectiva lógica (simbología: UML)



Elemento	Responsabilidad	Propiedades
Capa de presentación	Engloba todos los componentes encargados de recibir interacción por parte del usuario y de mostrarle resultados.	Lenguaje = Java *Framework* = JSF
Capa de servicios *web*	Expone servicios *web* para soportar la integración de sistemas externos.	Lenguaje = Java *Framework* = Spring-WS
Capa de negocio	Engloba componentes de coordinación que se encargan de llevar a cabo la ejecución de los casos de uso del sistema.	Lenguaje = Java *Framework* = Spring
Capa de datos	Engloba componentes encargados de almacenar objetos en una base de datos, ocultando del resto de la aplicación el hecho de que se trata de una base de datos relacional.	Lenguaje = Java *Framework* = Hibernate
Capa de integración	Permite la integración con sistemas externos (por ejemplo, sistema de cobros).	Protocolo = SOAP

Continúa...

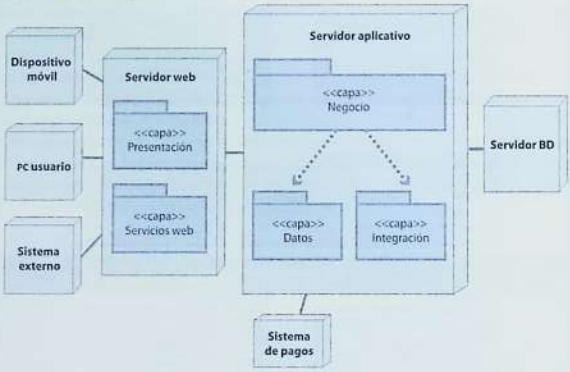
UNIVERSIDAD TÉCNICA DEL NORTE

Continuación...

Resumen de iteración

Estructuras resultantes y responsabilidades de los elementos

Perspectiva física (simbología UML)



© Humberto Cervantes, Micaela Avila Velasco Fuentes, Elsa Cecilia Camargo

Elemento	Responsabilidad	Propiedades
PC del usuario	Computadora personal mediante la cual acceden los usuarios y administradores del sistema.	Navegador = Internet Explorer 10+, Firefox 10+, Google Chrome 17+
Dispositivo móvil	Dispositivos por medio de los cuales se accederá en el futuro a la aplicación.	OS = iPhone/iPod Touch, iOS 6+ Android 4+
Servidor *web*	El servidor web alberga las capas de presentación y servicios web.	Tipo = Tomcat
Servidor Aplicativo	El servidor aplicativo alberga las capas de negocio, datos e integración de la aplicación.	Memoria = Por definir Procesador = Por definir
Servidor BD	Este servidor alberga la base de datos (del otro servidor legado).	Motor = Oracle v11.2
Sistema externo	Sistema externo que interactua con el sistema que se desarrolla mediante servicios web.	
Sistema de pagos	Sistema externo a través del cual se realizan los pagos.	

Continúa...

Continuación...

Resumen de iteración	
Interfaces de los elementos	Interfaces lógicas entre: • Capas de presentación y negocio. • Capas de servicios *web* y negocio. • Capa de negocio y datos. • Capa de negocio e integración. Interfaces físicas entre: • PC de usuario, dispositivo móvil, sistemas externos y sistema de pagos. • Servidores aplicativos y BD.

Al final de esta iteración se revisa si ya se han tomado decisiones de diseño para satisfacer los *drivers* de la arquitectura. Se han hecho muchas, pero no suficientemente detalladas, por ello es necesario realizar iteraciones adicionales.

3.2 Segunda iteración: integración de la funcionalidad a las capas

En la segunda iteración de diseño, el arquitecto se enfoca en identificar la manera en que se soportará la funcionalidad primaria del sistema. Para ello, toma las capas de este como elementos a descomponer (paso 2 del ADD). Los *drivers* para esta iteración (elegidos en el paso 3) son los casos de uso primarios, CU-01 y CU-02. Respecto de la elección de los conceptos de diseño (paso 4), se comienza por seleccionar el patrón *Domain Object*, que tiene como propósito encapsular funcionalidades distintas del sistema en bloques autocontenidos llamados objetos de dominio.

Se selecciona también el patrón *Data Mapper*, que es un tipo particular de *Domain Object* con el cual se encapsulan los aspectos de acceso a la base de datos relacional. A efecto de permitir la realización de pruebas, los objetos de dominio refinan mediante los patrones *Explicit Interface* y *Encapsulated Implementation* para separar claramente la interfaz y la implementación.

A continuación se instancian elementos a partir de los patrones (paso 5). El resultado es que en cada una de las capas se crean componentes enfocados en soportar cada uno de los casos de uso. Una vez identificados los componentes dentro de las capas, se procede a definir interfaces para ellos (paso 6). Por esta razón se realiza un análisis dinámico de las interacciones entre los componentes para soportar los flujos asociados a los casos de uso. Para este análisis se usan diagramas de secuencia de UML, y como resultado se identifican los métodos asociados a las interfaces de los componentes.

Al final de esta iteración ya se tiene más claro cuáles serán los componentes necesarios para soportar la funcionalidad primaria del sistema, y esto sirve de marco de referencia para identificar aquellos que implementan el resto de la funcionalidad. Sin embargo, no se han tomado decisiones más específicas para satisfacer los escenarios de atributos de calidad, por lo que la iteración siguiente se enfoca en este aspecto.

Resumen de iteración	
Elemento a descomponer	Las distintas capas del sistema.
***Drivers* elegidos para la iteración**	Casos de uso primarios: • CU-01: consultar comidas. • CU-02: comprar boleto.

Continúa...

Continuación...

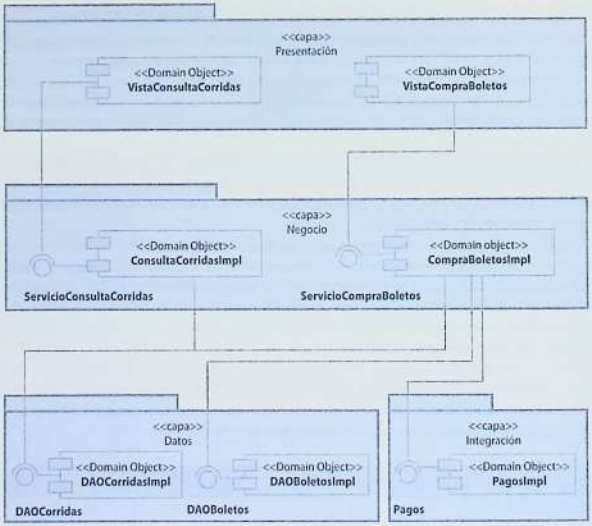
Resumen de iteración

Conceptos de diseño

Concepto	Justificación
Patrón Objeto de dominio ("Domain Object", 208)	Permite aislar las funcionalidades del sistema a efecto de soportar el desarrollo en paralelo y la modificabilidad.
Patrón Interfaz explícita ("Explicit Interface", 281) e Implementación encapsulada ("Encapsulated Implementation", 381)	Tener una separación de la interfaz y la implementación de los objetos de dominio facilita la realización de pruebas unitarias.
Patrón Mapeador de datos ("Data Mapper", 540)	También conocido como Objeto de acceso a datos (*Data Access Object*, o DAO), es un tipo particular de *Domain Object* que permite encapsular los aspectos de acceso a la base de datos relacional para favorecer la modificabilidad.

Estructuras resultantes y responsabilidades de los elementos

Perspectiva lógica (simbología UML)



Continúa...

Continuación...

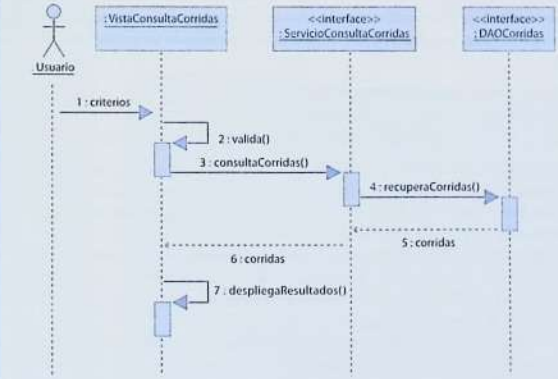
Resumen de iteración

Estructuras resultantes y responsabilidades de los elementos

Elemento	Responsabilidad	Propiedades
VistaConsultaCorridas	Este componente es responsable del despliegue y la navegación de las pantallas asociadas al CU-01.	Lenguaje = Java *Framework* = JSF
VistaCompraBoletos	Este componente es responsable del despliegue y la navegación de las pantallas asociadas al CU-02.	Lenguaje = Java *Framework* = JSF
ConsultaCorridasImpl	Este componente es responsable de proporcionar funciones para soportar el CU-01.	Lenguaje = Java Sesión = *Stateless*[1]
CompraBoletosImpl	Responsable de proporcionar funciones para soportar el CU-02.	Lenguaje = Java Sesión = *Stateless*
DAOCorridasImpl	Este componente tiene la función de realizar la persistencia de las entidades de tipo Corrida en la base de datos relacional y de aislar los detalles de persistencia del resto de la aplicación.	Lenguaje = Java *Framework* = *Hibernate*
DAOBoletosImpl	Lleva a cabo la persistencia de las entidades de tipo Corrida en la base de datos relacional y de aislar los detalles de persistencia del resto de la aplicación.	Lenguaje = Java *Framework* = *Hibernate*
PagosImpl	Este componente se comunica con el sistema externo para realizar los pagos en la compra de boletos.	Lenguaje = Java *Framework* = *Axis*[1]

Interfaces de los elementos

El siguiente diagrama de secuencia en UML muestra la manera en que interactúan los componentes lógicos para soportar el flujo principal del CU-01 (consultar corridas). Cabe señalar que se muestran mensajes de un objeto a sí mismo a efecto de facilitar el entendimiento de la manera en que se soporta la interacción.



[1] Los componentes *stateless* carecen de estado asociado permitiendo, por ejemplo, que se les acceda de forma concurrente. http://axis.apache.org/axis/java/

Continúa...

Continuación...

Resumen de Iteración

Interfaces de los elementos

Los métodos asociados a las interfaces de los componentes se detallan a continuación:

Interfaz	Detalles/Descripción
ServicioConsultaCorridas	`Corridas[ ]consultaCorridas(Criterios)` recupera corridas de acuerdo con los criterios (invocación síncrona)

DAOCorridas	`boolean creaCorrida(Corrida)`
	`Corridas [ ] recuperaCorridas(String criterio)`
	`boolean actualizaCorrida(Corrida)`
	`boolean borraCorrida (Corrida)`
	Nota: en general, los **DAO** proporcionan métodos de creación, recuperación, actualización y borrado (**CRUD**[1]), por lo que estos últimos se identifican aun si no son usados en el análisis dinámico.

3.3 Tercera iteración: desempeño en capa de datos

En la tercera iteración, el arquitecto decide llevar a cabo decisiones de diseño más detalladas que le permitan satisfacer el escenario de atributo de calidad de desempeño (EAC-01) que resultó prioritario. Recordemos este escenario:

> *Un usuario selecciona la opción "buscar" una vez que ha elegido las ciudades de salida y destino, así como las fechas en la pantalla de consulta (CU-01) en un momento normal de operación y hay hasta cien usuarios conectados al sistema. Este procesa la petición y muestra la lista de corridas en un tiempo no mayor a dos segundos.*
>
> Donde:
>
> - Momento normal de operación = 100 usuarios.
> - Momento de sobrecarga = 500 usuarios simultáneos.
> - Dos segundos incluye únicamente tiempo de procesamiento y no tránsito en red.

En esta iteración, el arquitecto decide enfocarse en la capa de datos, que será entonces el elemento a descomponer (paso 2 del ADD). El *driver* primario seleccionado para esta iteración es el escenario EAC-01 (paso 3 del ADD), y respecto de los conceptos de diseño elegidos en el paso 4 del ADD, estos incluyen los patrones *Lazy Acquisition* e *Eager Acquisition*, así como las tácticas de Mantener copias múltiples y Manejo de recursos.

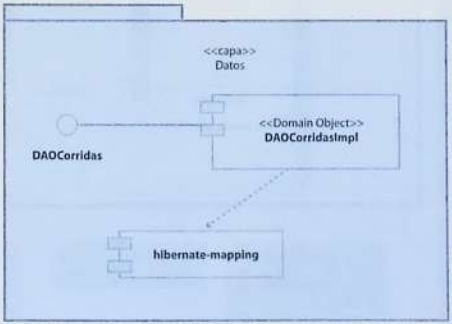
Aunque esos patrones y tácticas son los conceptos de diseño elegidos, forman parte de las opciones de configuración del *framework* usado en la capa de persistencia, por lo que la instanciación de estos conceptos (paso 5 del ADD) requiere solo la configuración del *framework*, que en este caso se lleva al cabo mediante un archivo en formato XML propio al *framework*.

Para el paso 6 del ADD se realiza de nuevo un análisis dinámico que aquí resulta en un refinamiento de las interfaces identificadas en la iteración anterior. Al final de esta iteración, el arquitecto concluye que en el nivel de la capa de datos ha tomado suficientes decisiones de diseño para satisfacer el atributo de calidad de desempeño que fue el *driver* de la iteración. Sin embargo, estas decisiones no son suficientes para satisfacer el escenario pues el desempeño requiere de otras decisiones en las demás capas del sistema, las cuales deberán hacerse en iteraciones subsecuentes.

---

[1] CRUD: Create, Read, Update and Delete.

Resumen de iteración		
Elemento a descomponer	Capa de datos	
***Drivers* elegidos para la iteración**	EAC-01	
Conceptos de diseño	**Concepto**	**Justificación**
	Patrones Adquisición temprana (*Eager Acquisition*, 509) y Adquisición tardía (*Lazy Acquisition*, 507)	El uso de *Eager Acquisition* en determinadas situaciones mejora desempeño pues se evita que *Hibernate* haga varios **SELECT** para objetos dependientes que deben ser cargados de forma simultánea. El uso de *Lazy Acquisition* en determinadas situaciones mejora desempeño al no cargar de forma inmediata propiedades o dependencias que podrían no usarse (por ejemplo, datos que no se muestran en una pantalla).
	Táctica Mantener copias múltiples Cachés	Uso de *process-level cache* (caché a nivel de proceso) para objetos que cambian con poca frecuencia. Uso de *query cache* (caché de consultas) para almacenar resultados de las consultas.
	Táctica Manejo de recursos Paginación	Las consultas no regresan todos los datos de la base sino un subconjunto.
Estructuras resultantes y responsabilidades de los elementos	Perspectiva lógica (simbología: UML)	



Continúa...

Continuación...

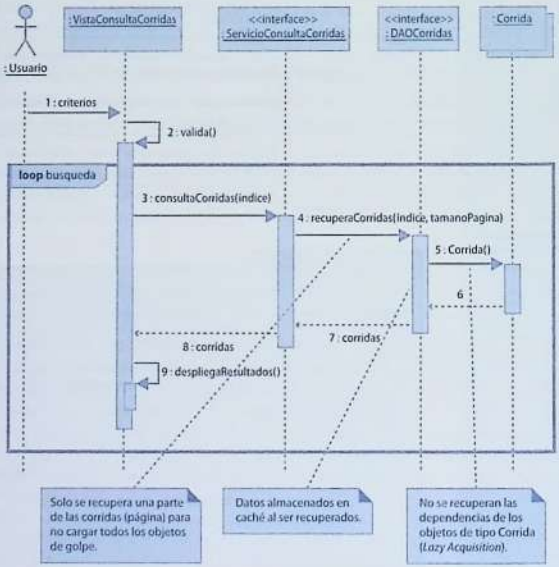
Resumen de iteración

Estructuras resultantes y responsabilidades de los elementos

Elemento	Responsabilidad	Propiedades
DAOCorridasImpl	Provee una implementación de los métodos CRUD declarados en la interfaz DAOCorridas.	Lenguaje = Java Framework = *Hibernate*
hibernate-mapping	Archivo de configuración de *hibernate* en el cual se configuran los parámetros correspondientes a *Eager Acquisition*, *Lazy Acquisition* y Cachés.	Lenguaje = XML

Interfaces de los elementos

El siguiente diagrama de secuencia de UML muestra la manera en que interactúan los componentes lógicos para soportar el flujo principal del CU-01:



© Humberto Cervantes, Micaela Peña Salinas Estrada y Iván Castro Cienfuegos

Continúa...

Continuación...

Resumen de iteración

Interfaces de los elementos

Los metodos asociados a las interfaces de los componentes se detallan a continuación (las llamadas a sí mismo de cada objeto no se muestran pues no son públicas).

Interfaz	Detalles/Descripción
ServicioConsultaCorridas	`Corrida[]consultaCorridas(indice)` recupera corridas a partir del índice.
DAOCorridas	`Corrida[]recuperaCorridas(indice, tamanoPagina)` recupera un bloque de corridas con base en el índice y tamaño de página.

Como parte del diseño sería necesario hacer más iteraciones para definir estructuras con las cuales satisfacer los demás *drivers* de la arquitectura. Por falta de espacio, estas no se incluyen.

SECCIÓN 4
DOCUMENTACIÓN

En esta sección se ilustra la manera en que el arquitecto lleva a cabo la etapa de documentación de la arquitectura del sistema. El método utilizado en esta etapa es Vistas y más allá (véase en el capítulo 4 la sección 4.6.1). De esta forma, el diseño producido en la etapa anterior se documentará haciendo uso de vistas en las siguientes categorías:

a) Módulos.
b) Componentes y conectores.
c) Localización.

De manera adicional se generarán documentos relevantes respecto de, por ejemplo, justificación de las decisiones de diseño y documentación de interfaces. Como lo indica el método Vistas y más allá, para elaborar las vistas en las tres categorías se usarán estilos arquitectónicos, y su selección y priorización se llevará a cabo siguiendo estos pasos:

1. Generar una lista de vistas candidatas.
2. Combinar las vistas.
3. Priorizar las vistas.

4.1 Generar una lista de vistas candidatas

Como se explicó en el capítulo 4, en este paso el arquitecto identifica las vistas relevantes para documentar el diseño en turno. A efecto de facilitar esta tarea, debe apoyarse de una tabla que en sus filas incluya el conjunto de interesados en el sistema, y en sus columnas, las vistas organizadas en las tres categorías mencionadas anteriormente. Estas vistas consideran estilos particulares, por ejemplo, Descomposición, Uso, Capas o Modelo de datos.

La tabla A-1 muestra la información correspondiente a nuestro caso de estudio. El conjunto de interesados proviene del documento de visión y alcance presentado antes. Para cada vista se indica el nivel de detalle que se requiere al documentarla.

	Vistas de módulos					Vistas de componentes y conectores					Vistas de localización			Otra documentación					
	Descomposición	Uso	Generalización	Capas	Modelo de datos	Filtros y tuberías	Datos compartidos	Cliente-servidor	Punto-punto	Comunicación entre procesos	Puesta en marcha	Instalación	Asignación de trabajo	Documentación de interfaces	Diagrama de contexto	Mapeo entre vistas	Variabilidad	Análisis	Descripción de decisiones Arquitectónicas
Líder del proyecto	m	m		m	m						d		d		d				
Equipo de desarrollo (y mantenimiento)	d	d		d	d			d			m	d		d	d				d
Equipo de pruebas	d	d		d	d			d			m	m		d	d				
Representante de infraestructura	m	m		p				m			d	d							
Arquitecto	d	d		d	d			d			d	d	d	d	d				d
Representante de usuarios								p			p								
Director de operación											m		d		d				

d = Muy detallada
m = Considerable detalle
p = Poco detalle

Tabla A-1. Lista de vistas candidatas a documentar

Las vistas de módulos describen las estructuras de la arquitectura en términos de elementos que corresponden a unidades de implementación. Como se indica en la tabla, para este caso de estudio los estilos arquitectónicos seleccionados son: Capas, Descomposicion, Uso y Modelo de datos. El estilo Capas permite detallar relaciones de uso entre grupos lógicos de módulos llamados capas. De esta forma, es útil para explicar la estructura general del sistema definida por el arquitecto. En términos generales, el estilo Descomposición permite describir la estructura interna de los módulos. Por lo tanto, es conveniente para explicar la estructura interna de las capas del sistema. El estilo Uso permite describir las dependencias funcionales entre módulos; por esta razón es un candidato para explicar esas dependencias entre las capas y los módulos que las conforman. Por último,

el estilo Modelo de datos se considera también como candidato porque con él se detalla información acerca de entidades de datos sobre las que opera el sistema. Durante el diseño no se identificaron relaciones de especialización-generalización entre elementos de la arquitectura que requirieran ser documentadas con el estilo Generalización. Por tal razón, este no se encuentra en el conjunto de vistas candidatas.

Por otra parte, las vistas de componentes y conectores describen la arquitectura del sistema en términos de elementos visibles en tiempo de ejecución. Para nuestro caso de estudio se ha elegido el estilo Cliente-servidor debido a que permite especificar la interacción entre un conjunto clientes y un servidor, todos geográficamente dispersos, mediante un modelo de invocación síncrona. El resto de los estilos arquitectónicos en esta categoría de vistas no fue considerado, puesto que los modelos de interacción que manejan no son requeridos en el sistema.

Las vistas de localización permiten detallar las relaciones entre los elementos documentados en otras vistas y los elementos "físicos" del sistema. En nuestro caso de estudio hemos considerado como candidatos los siguientes estilos: Implantación, Instalación y Asignación de trabajo. El primero es útil para indicar los elementos de *hardware* en los que se ejecutan los descritos en las vistas de componentes y conectores. Con este estilo daremos a conocer los detalles sobre la separación en diversos tercios del sistema, como las PC de usuario, sistemas externos, servidor *web*, servidor aplicativo y servidor de base de datos. Con el estilo Instalación se describe la ubicación de las unidades de implementación (los módulos) en la estructura de archivos del ambiente de producción del sistema. El estilo Asignación de trabajo detalla la relación entre los módulos sistemáticos y el personal que los desarrollará.

En lo que se refiere a la generación de otra documentación, además de la justificación de las decisiones arquitectónicas se decidió generar la documentación de interfaces y un diagrama de contexto de alto nivel. Con la primera se detallan los métodos que las interfaces especifican y que utilizan los componentes para interactuar y soportar los casos de uso. Como se ilustró en secciones anteriores, el diagrama de contexto es de utilidad para describir las entidades principales que interactúan con el sistema desarrollar.

Como se aprecia en la tabla A-1, para la mayoría de la vistas candidatas se asignó un nivel de detalle alto (d muy detallada). Esto se debe a que la mayoría de los involucrados con el sistema son personal técnico. Todos ellos usan la información contenida en estas vistas para guiar y restringir las actividades que realizan.

UNIVERSIDAD TECNICA DEL NORTE BIBLIOTECA

4.2 Combinar las vistas

Una vez generada la tabla de interesados y vistas, el arquitecto debe analizarla con el propósito de reducir número de vistas a documentar. Durante el análisis se consideró principalmente la utilidad de las vistas de forma independiente, y que al combinar una con otras no se introdujera complejidad en su elaboración, mantenimiento e interpretación. A partir de este análisis se decide generar las vistas listadas a continuación.

Vistas de módulos:

- Vista 1: Capas, Descomposición y Uso (muy detallada).
- Vista 2: Modelo de datos (muy detallada).

Vistas de comportamiento y localización:

- Vista 3: Cliente-servidor (muy detallada).
- Vista 4: Implantación e Instalación (muy detallada).

En la categoría de módulos, la vista Capas es muy útil para introducir la estructura general del sistema a los miembros actuales y futuros de los equipos técnicos (desarrollo, prueba, implantación). Sin embargo, dado que el número de capas es reducido y que estas corresponden a capas cuya semántica es conocida (vista, servicios, negocio, integración y datos), consideramos que la información de esta vista se puede combinar junto con la de los estilos Descomposición y Uso sin introducir complejidad en su elaboración, interpretación y mantenimiento.

Se decidió documentar la vista con el estilo Modelo de datos sin combinarla puesto que contiene información de naturaleza diferente de las de Descomposición y Uso.

Las vistas Cliente-servidor se documentarán de forma independiente, pues se utilizarán con frecuencia para ilustrar los casos de uso primarios. Las vistas Implantación e Instalación se pueden consolidar en una sola sin introducir mucha complejidad en su elaboración, mantenimiento e interpretación.

4.3 Priorizar las vistas

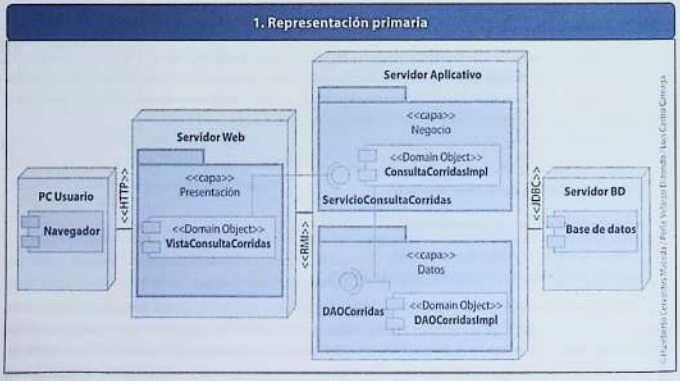
En este último paso del método, el arquitecto decide cuáles vistas, de la lista anterior, se deben elaborar primero. Para ello se consideró principalmente la utilidad de ellas durante las etapas del desarrollo del sistema. A continuación se muestra la priorización generada:

Prioridad	Nombre de vista
1	Vista 1: Capas, Descomposición y Uso.
	Vista 2: Modelo de datos.
2	Vista 3: Cliente-servidor.
	Vista 4: Implantación e Instalación.

La utilidad de las vistas 1 y 2 es alta puesto que, como lo mencionamos antes, son punto de partida para guiar y restringir actividades realizadas por los miembros de los equipos técnicos del proyecto. Una vez producidas, el arquitecto puede elaborar las vistas 3 y 4.

4.4 Ejemplo de vista

Una vez priorizadas las vistas, se debe documentar cada una. A efecto de ilustrar el resultado de esta actividad, la figura A-2 contiene la documentación generada para la vista 3, Cliente-servidor, el CU-01: consultar corridas, y documentos en los que se empleó la plantilla sugerida por el método Vistas y más allá.



Continúa...

Continuación...

2. Catálogo de elementos

a) Elementos:

Elemento	Responsabilidad	Propiedades
Navegador	Accede a la página generada por VistaConsultaCorridas.	
VistaConsultaCorridas	Genera las páginas relacionadas con la consulta de corridas.	Lenguaje = Java *Framework* = JSF Sesión = *Stateless* Tipo = *Singleton*[1] Número de solicitudes = por definir
ConsultaCorridasImpl	Esta instancia implementa los métodos de la interfaz SrvConsultaCorridas.	Lenguaje = Java *Framework* = JSF Sesión = *Stateless* Tipo = *Singleton* Número de solicitudes = por definir
DAOCorridasImpl	Esta instancia implementa los métodos CRUD declarados en la interfaz DAOCorridas.	Lenguaje = Java *Framework* = por definir Sesión = por definir Inicialización = *Lazy*
Base de datos	Base de datos.	

b) Relaciones:

Nombre	Responsabilidad	Propiedades
***Request/Response* (L)**	Permite la comunicación entre una instancia que proporciona un servicio y otra que lo requiere.	Comunicación = Síncrona Protocolo = MI (Method Invocation o Invocación a método) Modo = Local (L)
***Request/Response* (R)**	Permite la comunicación entre una instancia que proporciona un servicio y otra que lo requiere.	Comunicación = Síncrona Protocolo = HTTP, HTTPS, RMI (*Remote Method Invocation* o Invocación a método remoto) Modo = remoto (R)

Continúa...

[1] El uso del término *Singleton* se refiere al asegurar que solo exista una instancia del elemento al que se hace referencia, por ejemplo, una clase.

[2] *Request/Response* se refiere a una forma de comunicación entre dos elementos en donde el primero envía una solicitud al segundo y este le responde.

Continuación...

2. Catálogo de elementos

c) Interfaces:[1]

ServVistaConsultaCorridas

Nombre de método	consultarCorridas.	
Precondiciones	La cadena provista con los criterios de búsqueda es correcta y no vacía.	
Poscondiciones	Ninguna.	
Parámetros y retorno		
Nombre	**Tipo**	**Descripción**
criterio	*String*	Cadena con los criterios de búsqueda de la corrida. Incluye ciudades de salida y destino, así como fechas de salida y regreso.
Valor de retorno	*void*	
Excepciones		
Nombre	**Descripción**	
IllegalArgumentException	Indica que a un método se le ha pasado un argumento inapropiado.	
BusinessComponentException	Señala que la instancia del componente de la capa de negocios solicitado ha regresado un error o una advertencia.	

ServConsultaCorridas

Nombre de método	consultaCorridas.	
Precondiciones	• La cadena provista con los criterios de búsqueda es correcta y no vacía. • El índice de la página provisto es correcto y no vacío.	
Poscondiciones	Ninguna.	
Parámetros y retorno		
Nombre	**Tipo**	**Descripción**
criterio	*String*	Cadena con los criterios de búsqueda de la corrida. Incluye ciudades de salida y destino, así como fechas de salida y regreso.
índice	int	Índice de la página que contiene los datos a recuperar.

Continúa...

[1] Por razones de espacio no se muestran todos los métodos en las interfaces de los componentes listados en el catálogo de elementos.

Continuación...

2. Catálogo de elementos

ServConsultaCorridas

Valor de retorno	Corridas [ ]	Arreglo de objetos tipo Corrida con las corridas que corresponden al criterio provisto.

Excepciones	
Nombre	**Descripción**
IllegalArgumentException	Indica que a un método se le ha pasado un argumento inapropiado.
DAOException	Indica que el DAO solicitado ha regresado un error o una advertencia.

DAOCorridas

Nombre de método	recuperaCorridas.
Precondiciones	La cadena provista con los criterios de búsqueda es correcta y no vacía.
Poscondiciones	True.

Parámetros y retorno		
Nombre	**Tipo**	**Descripción**
criterio	*String*	Cadena con los criterios de búsqueda de la corrida. Incluye ciudades de salida y destino, así como fechas de salida y regreso.
indice	int	Índice de la página que contiene los datos a recuperar.
tamanioPagina	int	Tamaño de la página que contiene los datos a recuperar.
Valor de retorno	Corridas [ ]	Arreglo de objetos tipo Corrida con las corridas que corresponden al criterio provisto.

Excepciones	
Nombre	**Descripción**
IllegalArgumentException	Indica que a un método se le ha pasado un argumento inapropiado.
SQLException	Señala que el servidor de base de datos ha regresado un error o una advertencia.

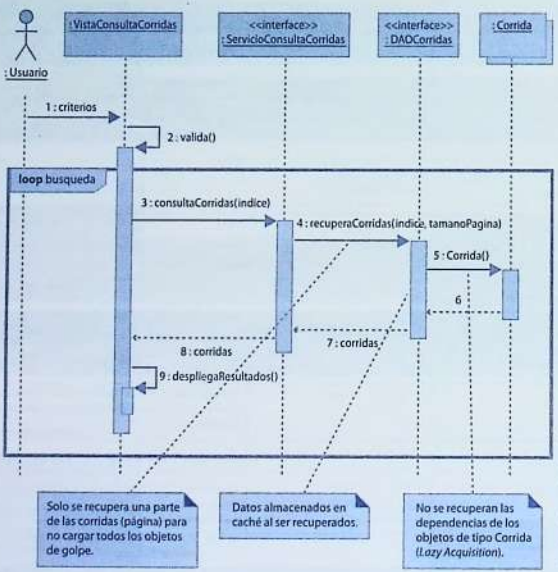
Continúa...

Continuación...

2. Catálogo de elementos

d) Comportamiento:

El siguiente diagrama muestra cómo los elementos descritos en esta vista interactúan para soportar el CU-01: consultar corridas. Incluye algunas anotaciones relacionadas con escenarios de atributos de calidad relevantes al caso de uso.



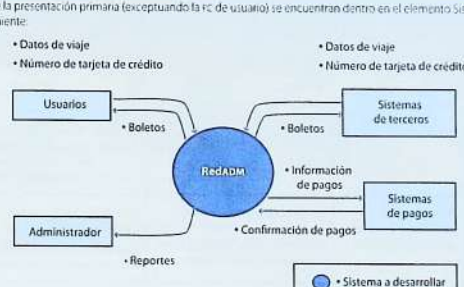
© Humberto Cervantes Maceda, Perla Velasco Elizondo, Luis Castro Careaga

Continúa...

Continuación...

3. Diagrama de contexto

Los elementos de la presentación primaria (exceptuando la FC de usuario) se encuentran dentro en el elemento Sistema a desarrollar del diagrama siguiente:



Heriberto Cervantes Miranda / Perla Velarde Estrada / Eric Cortés Cabrera

4. Guía de variabilidad

No aplica

5. Descripción de decisiones arquitectónicas

a) Justificación

A continuación se detalla la justificación de las decisiones arquitectónicas en términos de los escenarios de atributos de calidad (EAC) relevantes al caso de uso CU-01, consultar corridas.

	Decisión	Justificación	Comentarios
EAC-01	Uso del patrón *Lazy Acquisition* (507).	El patrón *Lazy Acquisition* puede promover el desempeño al no cargar un objeto, y sus propiedades y dependencias, hasta que no es absolutamente necesario.	Este patrón se emplea en el nivel de capa de datos. Su implementación requiere modificar un archivo de configuración del framework usado en la capa de datos (Hibernate).
EAC-01	Uso del patrón *Eager Acquisition*.	El patrón *Eager Loading* puede promover el desempeño pues evita la ejecución de varias cláusulas SELECT cuando se carga de forma simultánea un objeto y sus propiedades y dependencias.	Este patrón se usa en el nivel de capa de datos. Su implementación requiere modificar un archivo de configuración del framework usado en la capa de datos (Hibernate).

Continúa...

Continuación...

5. Descripción de decisiones arquitectónicas

	Decisión	Justificación	Comentarios
EAC-01	Táctica: Mantener copias múltiples mediante el uso de cachés.	El uso de cachés promueve el desempeño, pues permiten almacenar datos de manera transparente para que las futuras solicitudes de estos puedan procesarse más rápida.	Esta táctica se utiliza en el nivel de capa de datos. Se lleva a cabo para objetos que cambian con poca frecuencia. Su implementación requiere activar el "process-level cache" en un archivo de configuración del *framework* usado en la capa de datos (*Hibernate*).
EAC-01	Táctica: manejo de recursos, mediante el uso de paginación.	El uso de paginación promueve el desempeño pues hace que una consulta no regrese todos los datos sino un subconjunto.	Este patrón se usa en el nivel de capa de datos. Su implementación requiere modificar la implementación del DAO correspondiente.

b) Análisis de resultados.
Esta sección la redactará el arquitecto una vez realizada la evaluación.

c) Supuestos:
- Se asume que no hay fallas en los canales que comunican a los clientes y el servidor.
- Los tiempos de procesamiento esperados incluyen únicamente duración de procesamiento y no el adicional por el tránsito en los enlaces de red.

6. Otra información

› **Figura A–2.** Ejemplo de documentación de una vista arquitectónica

SECCIÓN 5

EVALUACIÓN

UNIVERSIDAD TECNICA DEL NORTE
BIBLIOTECA

En esta sección se presentan tanto la manera en que se evalúa el sistema como ejemplos del resultado de su realización al sistema diseñado y documentado en las secciones previas.

Para cada escenario se hará el análisis en una forma llamada Análisis de enfoques arquitectónicos. Dado que el *driver* de disponibilidad no tiene importancia alta, no será considerado.

Análisis de enfoques arquitectónicos	
Escenario núm. 1	Compra de boleto con pago de tarjeta de crédito desde interfaz *web*
Atributo(s)	Funcionalidad, interoperabilidad y usabilidad
Entorno	Un cliente compra un boleto desde una interfaz *web* usando una tarjeta de crédito válida con fondos
Estímulo	Solicitud de compra de un boleto con información de viaje y de cobro
Respuesta	Funcionalidad: el boleto se compra y queda registrado para los datos que dio el cliente, y el cobro se hace correctamente en la tarjeta de crédito por medio de su banco Interoperabilidad: el sistema se comunica con el sistema del banco respetando protocolos de comunicación y seguridad Usabilidad: el usuario realiza de manera intuitiva la operación desde su navegador

Decisiones arquitectónicas	Punto de sensibilidad	Equilibrio	Riesgo	No-riesgo
Patrón Capas. Sistemas *N-tercios*, *framework* (R, *Spring* e *Hibernate* (sección A.3.1). Primera iteración estructuración en capas	Con el diseño el punto de sensibilidad es de una transacción a la vez, pues no hay indicaciones de manejo de concurrencia ni número de servidores, etcétera. Es decir, no se ve en la documentación de diseño alguna evidencia de que pueda soportar más de una transacción simultánea	Hay equilibrio entre mantenibilidad y desempeño pues las capas facilitan aquella aunque pueden reducir este. Se pueden utilizar para tal tipo de aplicaciones pues la reducción en el desempeño es aceptable.	R1: nada se sabe del detalle de la integración con el banco, como manejo de seguridad, disponibilidad, comunicación síncrona y asíncrona, etcétera	NR1: Mantenibilidad por la separación de capas NR2: Integración de las capas entre sí NR3: Usabilidad por medio de una Interfaz *web*
Capa de presentación	Una transacción simultánea bajo el argumento del punto de sensibilidad de la decisión arquitectónica anterior	Facilidad de uso y desempeño, pues la capa de presentación involucra otras capas, lo que incrementa el número de elementos procesando las transacciones y afectando el desempeño. Sin embargo, la decisión permite un desempeño adecuado para este tipo de aplicaciones	R2: en el diagrama de componentes, la capa de presentación solo se esquematiza sin describir sus componentes R3: no hay evidencia de que la arquitectura haga manejo de transacciones simultáneas	NR1: Mantenibilidad por la separación de capas NR3: Usabilidad por medio de una Interfaz *web*
Capa de servicios *web*	No hay información para determinar el punto de sensibilidad de cuántos servicios *web* pueden manejarse	Por el uso de *Spring* ws hay equilibrios entre la interoperabilidad y la facilidad de mantenimiento, seguridad y desempeño	R4: no hay detalle suficiente para ver con *Spring* ws el manejo de los servicios *web*	NR1: Mantenibilidad por la separación de capas NR2: Integración de las capas entre sí

Continúa...

Continuación...

Análisis de enfoques arquitectónicos				
Escenario núm. 1	Compra de boleto con pago de tarjeta de crédito desde interfaz web.			
Decisiones arquitectónicas	**Punto de sensibilidad**	**Equilibrio**	**Riesgo**	**No-riesgo**
CompraBoletoImpl-Stateless	No se puede determinar cuántas sesiones de compra puede manejar el componente.		R5: la compra debe manejar un estado en la sesión, y al utilizar un elemento *Stateless*, aquel debe ser llevado por otro elemento de la arquitectura, el cual no se señala en algún lugar del diseño.	NR1: Mantenibilidad por la separación de capas.
Razonamiento	Este escenario es fundamental y las decisiones arquitectónicas planteadas pueden implementarlo. Sin embargo, la falta de detalles genera riesgos respecto de los cuales hay que asegurarse de manera más precisa que la arquitectura pueda manejar.			
Diagrama arquitectónico	Diagrama CE-DIS-01. Diagrama CE-DIS-02. Diagrama CE-DIS-03.			

Análisis de enfoques arquitectónicos				
Escenario núm. 2	Consulta de corridas desde agencias de viajes.			
Atributo(s)	Funcionalidad e interoperabilidad.			
Entorno	Desde su propia aplicación, una agencia de viajes solicita una consulta de corridas para un trayecto dado en una fecha determinada.			
Estímulo	Solicitud de consulta de corridas con información de viaje (fechas, horarios, origen y destino).			
Respuesta	Funcionalidad: el sistema devuelve las corridas disponibles y espacios para los datos proporcionados en la consulta. Interoperabilidad: El sistema de la agencia se comunica con el sistema en cuestión respetando protocolos de comunicación y seguridad.			
Decisiones arquitectónicas	**Punto de sensibilidad**	**Equilibrio**	**Riesgo**	**No-riesgo**
Capa de servicios web	No hay información para determinar el punto de sensibilidad de cuántos servicios web de consulta pueden manejarse.	Por el uso de *Spring* ws hay equilibrios entre la interoperabilidad y la facilidad de mantenimiento, seguridad y desempeño.	R6: no hay detalle suficiente para ver el manejo de los ws con *Spring* ws.	NR1: Mantenibilidad por la separación de capas. NR2: Integración de las capas entre sí.
Razonamiento	Este escenario es fundamental y las decisiones arquitectónicas planteadas pueden implementarlo. Sin embargo, la falta de detalles genera riesgos respecto de los cuales hay que asegurarse de manera más precisa que la arquitectura pueda manejar.			
Diagrama arquitectónico	Diagrama CE-DIS-01. Diagrama CE-DIS-02. Diagrama CE-DIS-03.			

Continúa...

Continuación...

Análisis de enfoques arquitectónicos	
Escenario núm. 3	Mil usuarios concurrentes consultando, y tiempo de respuesta menor a dos segundos.
Atributo(s)	Desempeño.
Entorno	En el momento del escenario hay 999 usuarios consultando al sistema acerca de comidas para viaje.
Estímulo	Un cliente hace una consulta desde el portal *web*.
Respuesta	Funcionalidad: el sistema devuelve la información de la consulta. Desempeño: el usuario recibe la respuesta en menos de dos segundos.

Decisiones arquitectónicas	Punto de sensibilidad	Equilibrio	Riesgo	No-riesgo
Patrón Capas. Sistemas *N-tercios*, *framework* de, *Spring* e *Hibernate*.	Una transacción, de acuerdo con la documentación del diseño.		R7: la arquitectura no puede manejar el escenario de desempeño.	
Razonamiento	Este escenario es fundamental, y las decisiones arquitectónicas planteadas no pueden implementarlo pues no hay evidencia de uso de patrones y/o estrategias para resolver aspectos relacionados con el desempeño.			
Diagrama arquitectónico	Diagrama CE-DIS-01. Diagrama CE-DIS-02. Diagrama CE-DIS-03.			

5.2 Resultados de la evaluación

Al analizar los escenarios propuestos para la evaluación, se tienen identificados los siguientes resultados planteados como riesgos y no-riesgos:

Riesgos

- R1: nada se sabe del detalle de la integración con el banco, como manejo de seguridad, disponibilidad, comunicación síncrona y asíncrona, etcétera.
- R2: en el diagrama de componentes, la capa de presentación solo se esquematiza sin describir sus elementos detallados.
- R3: no hay evidencia de que la arquitectura haga manejo de transacciones simultáneas.
- R4: no hay detalle suficiente para ver con *Spring* ws el manejo de los servicios *web*.
- R5: la compra debe manejar un estado en la sesión, y al utilizar un elemento *Stateless*, aquel debe ser llevado por otro elemento de la arquitectura, el cual no se señala en algún lugar del diseño.
- R6: no hay detalle suficiente para ver con *Spring* ws el manejo de los ws .
- R7: la arquitectura no puede manejar el escenario de desempeño.

No-riesgos

NR1: Mantenibilidad por la separación de capas.

NR2: Integración de las capas entre sí.

NR3: Usabilidad por medio de una Interfaz *web*.

SECCIÓN 6

CONCLUSIÓN

La exposición del caso de estudio mostró la forma en que se realizan los principales pasos del ciclo de vida de la arquitectura. El caso cubrió los requerimientos, el diseño, la documentación y la evaluación. Quedan pendientes tanto la resolución de los riesgos detectados en la evaluación como la implementación del sistema usando la arquitectura ajustada.

El caso de estudio destacó la obtención de diseños robustos junto con una adecuada documentación de ellos. De manera adicional, aun realizando diseños de este tipo fue necesario evaluar el del caso para detectar posibles riesgos que deben ser resueltos antes de comenzar la implementación del sistema.

Los principales riesgos coinciden con los que por lo habitual se encuentran en la mayoría de los proyectos: la subespecificación de los diseños dejando los aspectos detallados de la arquitectura a lo que interprete el equipo de implantación.

GLOSARIO

A

ACDM (*Architecture Centric Design Method*). Método de diseño centrado en la arquitectura. Es un método de desarrollo de arquitectura.

Acoplamiento. Medida que permite establecer el grado de dependencia entre dos elementos relacionados. Se habla de acoplamiento alto si los cambios en un elemento requieren de modificaciones en el que está conectado con él, lo cual significa que hay una dependencia grande entre ellos. El acoplamiento alto es indeseable pues hace que los cambios en un sistema sean más costosos.

ADD (*Attribute Driven Design*). Diseño guiado por atributos. Es un método de diseño de arquitectura.

Arquitectura de referencia. Diseño arquitectónico predefinido que se usa por lo habitual para desarrollar una clase particular de aplicación.

Arquitectura de software. Conjunto de estructuras necesarias para razonar sobre el sistema. Comprende elementos de *software*, relaciones entre ellos, y propiedades de ambos. Permiten satisfacer los requerimientos, en particular los atributos de calidad, y distribuir el trabajo.

ATAM (*Architecture Tradeoff Analysis Method*). Método de análisis de equilibrios de la arquitectura. Método para evaluar las consecuencias de las decisiones arquitectónicas respecto de los *drivers* de la misma.

ARID (*Active Review for Intermediate Designs*). Revisiones activas para diseños intermedios. Este método representa una especialización de la técnica de revisiones activas aplicada al diseño de la *arquitectura de software*.

Atributos de calidad. Características medibles con las que se expresan de manera cuantitativa aspectos de calidad del sistema relevantes para quienes usan u operan este o los que lo desarrollan. Ejemplos: usabilidad, facilidad de mantenimiento y modificabilidad.

Auditoría. Procedimiento que evalúa durante la implementación un artefacto (o parte de él) crítico para detectar desviaciones respecto de la *arquitectura de software*.

B

BDUF (*Big Design Up Front*). Gran diseño al inicio. Se refiere a la creación de un diseño extensivo del sistema, antes de proceder a la construcción, en particular cuando el ciclo de desarrollo es secuencial o en cascada. Lo anterior es indeseable pues al generar un diseño completo del sistema sin realizar aunque sea una parte de implementación, se corre el riesgo de producir un diseño inadecuado o que deba ser corregido posteriormente. Lo anterior se debe a que durante la construcción se identifican aspectos que muchas veces se desconocen al momento de diseñar.

C

Caso de uso. Requerimiento que expresa una interacción entre un usuario y el sistema y permite al primero alcanzar un objetivo que le aporta valor.

Cohesión. Medida que hace referencia a que los módulos estén enfocados hacia tareas o "preocupaciones" particulares y semánticamente relacionadas. La cohesión alta es deseable.

Concepto de diseño. Primitivas utilizadas en el proceso de diseño de la arquitectura que incluyen patrones, tácticas, *frameworks* y otros elementos reutilizables.

D

Defecto. Condición que puede generar una falla en el sistema de *software*.

Desviación. Situación en la que un diseño o un elemento de programación no es consistente con las descripciones indicadas en la arquitectura.

Deuda técnica. Costos en los que se incurre debido a omisiones durante el diseño de un sistema a efecto de acelerar la implementación para cumplir fechas de entrega o expectativas de los clientes.

Documentación de arquitectura. Documentos que describen las estructuras de la arquitectura con el propósito de que la información al respecto pueda ser comunicada de manera eficiente a diversos interesados en el sistema.

Documentar. Elaboración de un documento, o conjunto de documentos, con el propósito de comunicar información relevante acerca de un proceso, producto o entidad.

Documento de visión y alcance. Documento que contiene información referente a por qué se requiere desarrollar un sistema y el alcance de este, mediante descripciones textuales sobre su contexto, necesidades que resuelve, descripción de sus usuarios, entre otras.

Dominio de aplicación. Área o contexto en donde un sistema funciona y son válidos determinados conceptos y comportamientos.

***Drivers* de la arquitectura.** Subconjunto de los requerimientos del sistema que se caracterizan por influir directamente en el diseño de la arquitectura de este. Los *drivers* incluyen por lo habitual atributos de calidad, requerimientos funcionales primarios y restricciones.

E

Elemento. Una parte del sistema que tiene determinadas características (la interfaz es parte de ellas) y, por lo habitual, se conecta con otros. Los elementos pueden existir en tiempo de desarrollo o ejecución, y representar unidades lógicas, dinámicas o físicas. Ejemplos: módulos, componentes, objetos o nodos físicos.

Estilo arquitectónico. Esquema de organización general para los sistemas de *software*. Establece los tipos de elementos, sus responsabilidades y las reglas para crear relaciones entre los elementos. Ejemplos de estos estilos: Filtros y tuberías o Cliente/servidor.

Estructura. Conjunto de elementos con propiedades visibles de forma externa y relaciones entre ellos. La *arquitectura de software* se compone de múltiples estructuras.

Evaluación de la arquitectura. Actividad enfocada a evaluar si el diseño de la arquitectura satisface o no los *drivers* de la misma. En caso de que eso no suceda, se identifican riesgos en el diseño de la arquitectura.

F

Falla. Situación que ocurre cuando un sistema de *software* en ejecución deja de funcionar, trabaja parcialmente o lo hace de manera errónea. Las fallas son provocadas por los defectos.

***Frameworks* de desarrollo.** 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.

FURPS+. Creado por Hewlett-Packard, es un modelo, el cual define una clasificación de atributos de calidad de *software* que incluye: funcionalidad (*Functionality*), usabilidad (*Usability*), confiabilidad (*Reliability*), desempeño (*Performance*) y soporte (*Supportability*).

H

Historia de usuario. Especificaciones de requerimientos de usuario escritas en una o dos frases utilizando el lenguaje del usuario final.

I

Implementación. Transformación de especificaciones en programas que serán funcionales para los usuarios.

Inspección. Evaluación llevada a cabo por personal diferente al que construyó un artefacto, con el propósito de detectar defectos siguiendo una lista de revisión.

Interesado. Persona o grupo afectado directa o indirectamente por el desarrollo de un sistema.

Interfaz. Especificación contractual que permite realizar el intercambio de información entre dos elementos. Tiene por lo habitual métodos con parámetros y valores de retorno, así como posibles excepciones. También puede tener pre- y poscondiciones asociadas a los métodos.

M

Métodos ágiles. Métodos ligeros de desarrollo de sistemas. Tienen un enfoque iterativo e incremental y se caracterizan por equipos de desarrollo multifuncionales y auto-organizados e iteraciones de desarrollo cortas en tiempo en las que se produce siempre una parte operable del producto final.

Modelo de dominio. Modelo de los conceptos asociados al problema específico. Describe abstracciones de estos conceptos con sus atributos y las relaciones que existen entre ellas.

Módulo. Unidad lógica que existe en tiempo de desarrollo y tiene una interfaz bien especificada. Son también unidades de asignación de trabajo pues por lo habitual los crea un ingeniero o un equipo específico.

O

Objetivo de negocio. Meta o metas que buscan alcanzar una organización y que motivan el desarrollo del sistema.

P

Patrón de diseño. Solución conceptual a un problema recurrente de diseño. Se describe por lo habitual mediante un nombre, un contexto y descripción del problema, una pormenorización de aquella solución y las consecuencias o implicaciones de la aplicación del patrón. El conjunto de nombres de estos patrones conforma un vocabulario de diseño.

***Product owner* (Propietario del producto).** Rol definido en el método ágil *Scrum*. Su responsabilidad es maximizar el valor del producto que se está construyendo, así como el trabajo del equipo de desarrollo.

Prototipo. Artefacto funcional mínimo que implementa algunos aspectos de riesgo de la *arquitectura de software*.

***Viewpoints and Perspectives* (Puntos de vista y perspectivas).** Método de diseño de arquitectura.

Q

QAW (*Quality Attribute Workshop*). Taller de atributos de calidad. Método para la especificación de atributos de calidad.

R

Regla de negocio. Término utilizado para referirse a políticas, estándares, prácticas o procedimientos organizacionales y/o gubernamentales que rigen o restringen la forma en que se llevan a cabo las actividades o procesos de una organización.

Requerimiento de negocio. *Véase* Objetivo de negocio.

Requerimiento. Especificación que describe una funcionalidad, atributo de calidad o restricción de un sistema de *software*.

Requerimiento de usuario. Especificación de los servicios que pueden realizar los usuarios por medio del sistema.

Requerimiento funcional. Especificación detallada de la funcionalidad del sistema, que en general complementa a la especificación de los casos de uso, y cuyo propósito es facilitar el diseño y la construcción del sistema.

Restricción. Tipo de requerimiento que limita las decisiones de diseño que se pueden tomar. Las restricciones pueden ser técnicas o administrativas.

Retrabajo. Correcciones que deben hacerse a artefactos intermedios y/o finales para eliminar defectos o reducir la deuda técnica.

Revisión personal. Evaluación realizada por quien construyó un artefacto la cual tiene como finalidad detectar defectos siguiendo una lista de revisión.

Revisión por colegas. *Véase* Inspección.

Riesgo. Evento que en caso de ocurrir tendrá un efecto (positivo o negativo) en los objetivos del proyecto.

S

Scrum. Método ágil de administración de proyectos, incluidos los de desarrollo de *software*.

***Scrum master* (Maestro *Scrum*).** Rol definido en el método ágil *Scrum*. Su responsabilidad es asegurar que el marco de trabajo de este sea entendido y adoptado.

***Sprint*.** Término utilizado para referirse a una iteración en el método ágil *Scrum*, cuya duración oscila entre una y cuatro semanas.

SRS (*Software Requirements Specification*). Documento de especificación de requerimientos. Documento que contiene una descripción completa de los requerimientos del sistema que se va a desarrollar.

T

Tablero Kanban. Tablero utilizado para mantener visible el estado actual de un proyecto respecto del trabajo terminado, trabajo en proceso y trabajo por realizar.

Táctica. Estrategia de diseño que influye sobre el control de la respuesta de un atributo de calidad particular. Por ejemplo, una táctica que permite aumentar la disponibilidad es la introducción de redundancia.

***Team*.** Equipo de desarrollo. Definido en el método ágil *Scrum*, es el rol cuya responsabilidad radica en la construcción del producto.

U

UML (*Unified Modeling Language*). Lenguaje de Modelado Unificado. Lenguaje gráfico de modelado de sistemas de *software* que se ha vuelto el estándar de facto.

V

Validación. Revisión realizada sobre entregables intermedios o finales de un proyecto para asegurar que satisfacen las necesidades de los usuarios finales.

Verificación. Revisión realizada sobre entregables intermedios o finales de un proyecto para asegurar que cumplen con los requerimientos definidos del proyecto.

Vista. Representación de una estructura. Contiene por lo habitual un diagrama (presentación primaria), además de información adicional que sirve de apoyo para comprenderlo.

Vista de comportamiento. Describe estructuras cuyos elementos denotan entidades visibles en tiempo de ejecución, por ejemplo instancias, procesos, objetos, clientes, servidores o almacenes de datos.

Vista física. Describe estructuras conformadas por elementos "físicos" que mantienen algún tipo de relación con los de las estructuras documentadas en otras vistas.

Vista lógica. Describe las estructuras arquitectónicas en términos de elementos que toman la forma de unidades de implementación considerando las propiedades y las relaciones u organización de cada una de estas unidades.

***Views and Beyond* (Vistas y más allá).** Método de documentación de arquitectura.

4+1 Views (4+1 Vistas). Es un marco conceptual para soportar la documentación de arquitectura.

BIBLIOGRAFÍA

Alexander, C., Ishikawa, S. & Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Barbacci, M. R., Ellison, R., Lattanze, A. J., Stafford, J. A., Weinstock, C. B. & Wood, W. G. (2008). *Quality Attribute Workshops (QAWs)*. Carnegie Mellon University. CMU/SEI-2003-TR-016.

Bass, L., Clements, P. & Kazman, R. (2012). *Software Architecture in Practice* (3d Edition). Addison-Wesley Professional.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, R., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas, D. (2001). *Manifesto for Agile Software Development*. http://agilemanifesto.org/

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc.

Boehm, B. W., Brown, J. R. & Lipow, R. (1976). *Quantitative evaluation of software quality*. Proceedings of the 2nd International Conference on Software Engineering (ICSE) , pp. 592-605.

Buschmann, F., Henney, K. & Schmidt, D. (2007). *Pattern-Oriented Software Architecture Vol. 4: A Pattern Language for Distributed Computing*. Wiley.

Clements, P. (2000). *Active Reviews for Intermediate Designs*. Carnegie Mellon University. CMU/SEI-2000-TN-009.

Clements, P., Kazman, R. & Klein, M. (2008). *Evaluating Software Architectures Methods and Case Studies*. Addison-Wesley.

Cockburn, A. (2000). *Writing Effective Use Cases (First edition)*. Addison-Wesley.

Cohn, M. 2004. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co.

Duvall, P., Matyas, S. & Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk (First edition)*. Addison-Wesley Professional.

Dyson, F. (1979). *Disturbing the Universe*. Basic Books.

Erl, T. (2009). *SOA Design Patterns*. Prentice Hall PTR.

Evans, E. (2003) *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Prentice Hall.

Fagan, M. E. (1976). *Design and Code Inspections to reduce errors in program development*. IBM Systems Journal, 15(3), pp. 182-211.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Hofmeister, C., Nord, R. & Soni, D. (1999). *Applied Software Architecture.* Addison-Wesley Longman Publishing Co.

Hohpe, G. & Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions.* Addison-Wesley Longman Publishing Co.

Homer, A., Sharp, J., Brader, L., Narumoto, M. & Swanson , T. (2014). *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications.* Microsoft patterns & practices.

International Standard Organization. (2001). *International Standard 9126. Information Technology-Software Product Evaluation–Quality characteristics and guidelines for their use.*

Jones, Capers. (2010). *Software Engineering Best Practices.* Prentice Hall.

Kruchten, P. (1995). *The 4+1 View Model of Architecture.* IEEE Software 12 (6), pp. 42-50.

Lattanze, A. (2008). *Architecting Software Intensive Systems: A Practitioners Guide.* Auerbach Publications.

Microsoft. (2009). *Microsoft Application Architecture Guide.* Microsoft Corporation.

Parnas, D. (1972). *On the criteria to be used for decomposing systems into modules.* Communications of the ACM, 15 (12), pp.1053-1058.

Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach* (5th edition). McGraw-Hill Higher Education.

Ralph, P. & Wand, Y. (2009). *A Proposal for a Formal Definition of the Design Concept.* Lecture Notes in Business Information Processing , 14, pp. 103-136.

Rozanski, N. & Woods, E. (2005). *Software Systems Architecture.* Addison-Wesley Professional.

Shaw, M. & Clements, P. (2006). *The Golden Age of Software Architecture.* IEEE Software 23 (2), pp. 31-39.

VersionOne (2013). *State of Agile Survey.* http://stateofagile.versionone.com/8th-annual-state-of-agile-form/

Wiegers, K. E. (2013). *Software Requirements* (Second ed.). Microsoft Press.

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R. & Wood, W. (2006). *Attribute-Driven Design (ADD)*, Version 2.0. Carnegie Mellon University, CMU/SEI-2006-TR-023.

Wong (2006). *Modern Software Review-Techniques and Technologies.* IRM Press.

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 operan las sondas robóticas que exploran otros planetas. Uno de los factores clave del éxito de los sistemas es su diseño eficiente; de manera particular, el diseño de la arquitectura de software.

En el contexto de la ingeniería de software el desarrollo de la arquitectura 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 con necesidades cada vez más complejas, como el desempeño, disponibilidad o facilidad de uso, razón por la cual la arquitectura es un tema fundamental.

En esta obra se describen claramente los procesos y estructuras necesarias para diseñar e implementar de manera más eficiente arquitecturas de sistemas de software.

Entre las características más relevantes del libro destacan las siguientes:

- Tiene un enfoque importante hacia las bases teóricas, pero también proporciona ejemplos prácticos que permiten relacionar la teoría con la realidad. Por esta razón puede ser usado por profesionales y estudiantes de maestría y licenciatura.
- En cada capítulo se emplea un caso de estudio para ejemplificar cada concepto o actividad del ciclo de desarrollo de arquitectura.
- Se discute cómo el desarrollo de la arquitectura puede realizarse en el contexto del método ágil *Scrum*.
- Se incluyen preguntas para el análisis y referencias que permiten a los lectores profundizar en el tema.

BIBLIOTECA UTN
055301

ISBN-13: 978-607-522-456-5
ISBN-10: 607-522-456-4

CENGAGE Learning

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