CENGAGE Learning

Humberto Cervantes Maceda
Perla Velasco-Elizondo
Luis Castro Careaga

ARQUITECTURA DE SOFTWARE

Conceptos y ciclo de desarrollo

ARQUITECTURA DE SOFTWARE

Conceptos y ciclo de desarrollo

Humberto Cervantes Maceda
Universidad Autónoma Metropolitana

Perla Velasco-Elizondo
Universidad Autónoma de Zacatecas

Luis Castro Careaga
Universidad Autónoma Metropolitana

Revisión técnica

Dr. René Mac Kinney Romero
Universidad Autónoma Metropolitana

CENGAGE Learning®

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

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

Presidente de Cengage Learning Latinoamérica:
Fernando Valenzuela Migoya

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

Editora de adquisiciones para Latinoamérica:
Claudia C. Garay Castro

Gerente de manufactura para Latinoamérica:
Raúl D. Zendejas Espejel

Gerente editorial en español para Latinoamérica:
Pilar Hernández Santamarina

Gerente de proyectos especiales:
Luciana Rabuffetti

Coordinador de manufactura:
Rafael Pérez González

Editora:
Abril Vega Orozco

Diseño de portada:
MSDE | MANU SANTOS Design

Imagen de portada:
©Ixpert/Shutterstock

Composición tipográfica:
Karla Paola Benítez García

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

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

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

CONTENIDO 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 mas 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

CAPÍTULO 5

EVALUACIÓN: ASEGURAR LA CALIDAD EN LA ARQUITECTURA 73

5.1 CONCEPTOS DE EVALUACIÓN 74

5.2 EVALUACIÓN DE ARQUITECTURAS 75

5.3 PRINCIPIOS DE LA EVALUACIÓN 75

5.3.1 Detección temprana de defectos en la arquitectura 75

5.3.2 Satisfacción de los *drivers arquitectónicos* 76

5.3.3 Identificación y manejo de riesgos 76

5.4 CARACTERÍSTICAS DE LOS MÉTODOS DE EVALUACIÓN DE ARQUITECTURAS 77

5.4.1 Producto que se evalúa: diseños o productos terminados 77

5.4.2 Personal que lleva a cabo la evaluación 78

5.5 MÉTODOS DE EVALUACIÓN DE ARQUITECTURAS 78

5.5.1 Revisiones e inspecciones 78

5.5.2 Recorridos informales al diseño 80

5.5.3 Método de análisis de equilibrios de la arquitectura (ATAM) 80

5.5.4 ACDM (etapa 4) 84

5.5.5 Revisiones activas para diseños intermedios (ARID) 86

5.5.6 Prototipos o experimentos 88

5.5.7 Comparación de métodos de evaluación 89

- EN RESUMEN 91
- PREGUNTAS PARA ANÁLISIS 91

CAPÍTULO 6

IMPLEMENTACIÓN: CONVERTIR EN REALIDAD LAS IDEAS ARQUITECTÓNICAS 93

6.1 CONCEPTO DE IMPLEMENTACIÓN DE *SOFTWARE* 94

6.2 LA ARQUITECTURA Y LA IMPLEMENTACIÓN DEL SISTEMA 95

6.3 PRINCIPIOS DE LA IMPLEMENTACIÓN DE *SOFTWARE* 95

6.4 DESVIACIONES DE LA IMPLEMENTACIÓN RESPECTO DE LA ARQUITECTURA 96

6.5 PREVENCIÓN DE DESVIACIONES 96

6.5.1 Entrenamiento de diseñadores y programadores 97

6.5.2 Desarrollo de prototipos o experimentos 97

6.5.3 Otras acciones de prevención 97

6.6 IDENTIFICACIÓN DE DESVIACIONES: CONTROLES DE CALIDAD 98
6.6.1 Identificación de desviaciones durante el diseño y la programación 99
6.6.1.1 Verificaciones del diseño detallado de los módulos 99
6.6.1.2 Verificaciones de la programación 99
6.6.2 Pruebas 99
6.6.3 Auditorías 100
6.7 RESOLUCIÓN DE LAS DESVIACIONES: SINCRONIZACIÓN DE LA ARQUITECTURA Y LA IMPLEMENTACIÓN 100
6.7.1 Desviaciones en el diseño detallado 100
6.7.2 Desviaciones en la programación 100
6.7.3 Defectos y/o subespecificación en la arquitectura 101
6.7.4 Cuándo conviene no resolver las desviaciones 101
- EN RESUMEN 103
- PREGUNTAS PARA ANÁLISIS 103

CAPÍTULO 7

ARQUITECTURA Y MÉTODOS ÁGILES: EL CASO DE *SCRUM* 105
7.1 MÉTODOS ÁGILES 106
7.2 *SCRUM* 107
7.2.1 Los roles 107
7.2.2 El proceso 108
7.3 ¿GRAN DISEÑO AL INICIO O DEUDA TÉCNICA? 111
7.4 DESARROLLO DE ARQUITECTURA EN *SCRUM* 111
7.4.1 Soporte de un enfoque de diseño planeado incremental 112
7.4.2 Especificación de atributos de calidad y restricciones 114
7.4.3 ¿Vistas de arquitectura? 116
7.4.4 El arquitecto de *software* 117
- EN RESUMEN 118
- PREGUNTAS PARA ANÁLISIS 118

APÉNDICE

CASO DE ESTUDIO 119
SECCIÓN 1: INTRODUCCIÓN 121
DOCUMENTO DE VISIÓN Y ALCANCE 122
1. INTRODUCCIÓN 122
2. CONTEXTO DE NEGOCIO 122
2.1 Antecedentes 122

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

CAPÍTULO 1

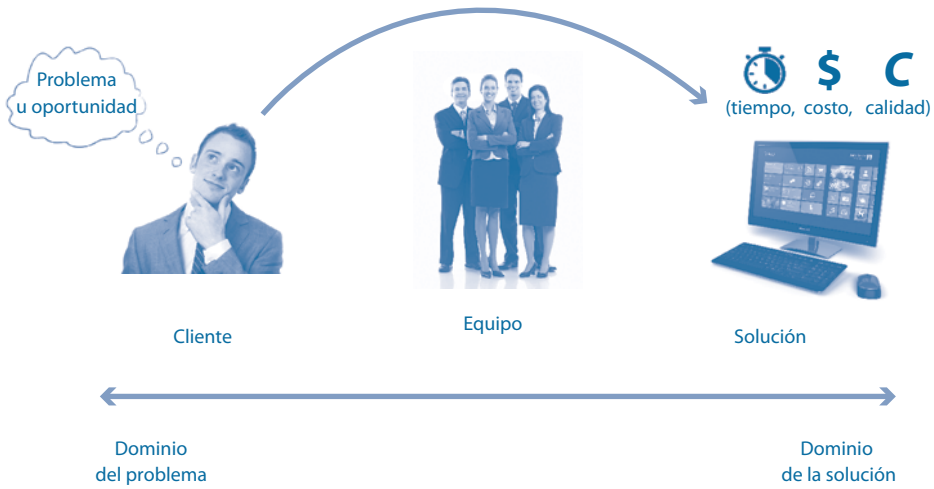
INTRODUCCIÓN: LA ARQUITECTURA Y EL DESARROLLO DE *SOFTWARE*

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

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

1.1 VISIÓN GENERAL DEL DESARROLLO DE SISTEMAS DE *SOFTWARE*

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

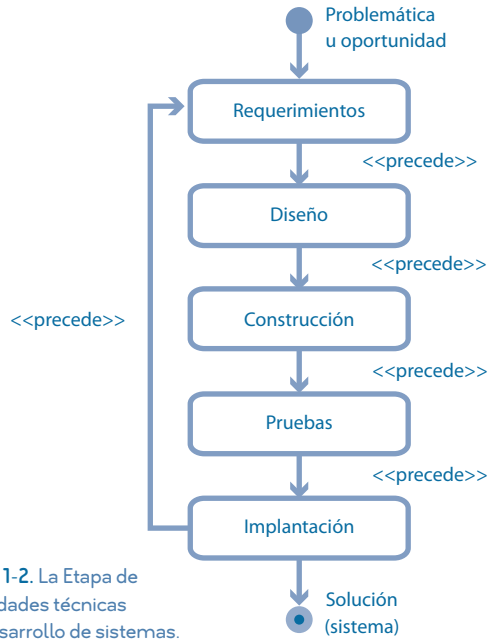


› **Figura 1-1.** Visión simplificada del desarrollo de sistemas.

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

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

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



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

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

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

1.2 DEFINICIÓN DE *ARQUITECTURA DE SOFTWARE*

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

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

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

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

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

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

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

1.3 ARQUITECTURA, ATRIBUTOS DE CALIDAD Y OBJETIVOS DE NEGOCIO

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

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

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

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

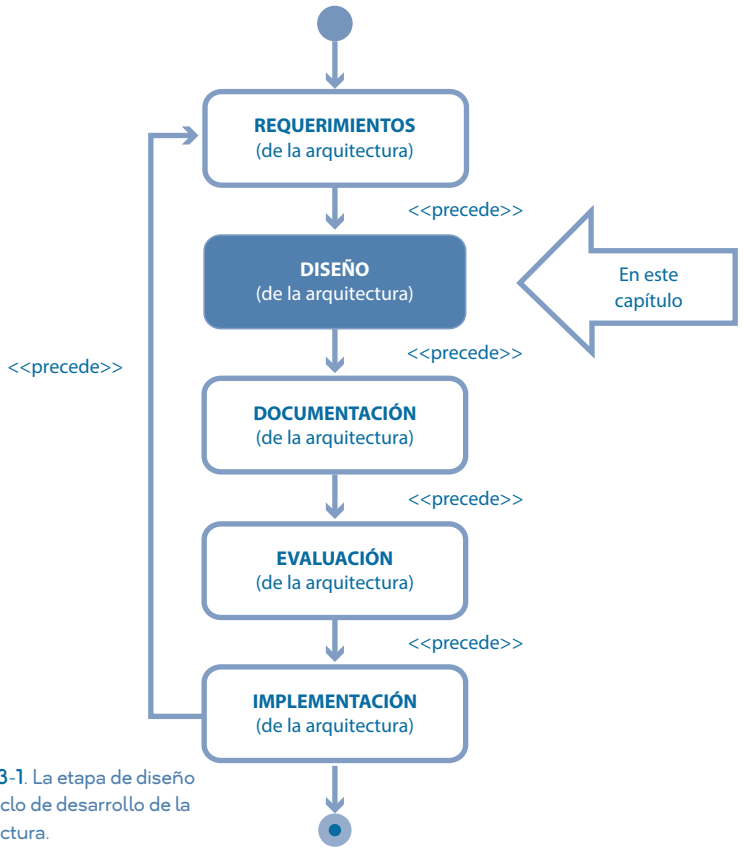
Es de señalar que los atributos de calidad y otros requerimientos del sistema se derivan de lo que se conoce como *objetivos de negocio*. Estos objetivos pertenecen al dominio del problema y son las metas que busca alcanzar una compañía y que motivan el desarrollo de un sistema. Es por ello que algunos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

CAPÍTULO 3

DISEÑO: TOMA DE DECISIONES PARA CREAR ESTRUCTURAS

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

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



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

3.1 DISEÑO Y NIVELES DE DISEÑO

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

3.1.1 Diseño y arquitectura

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

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

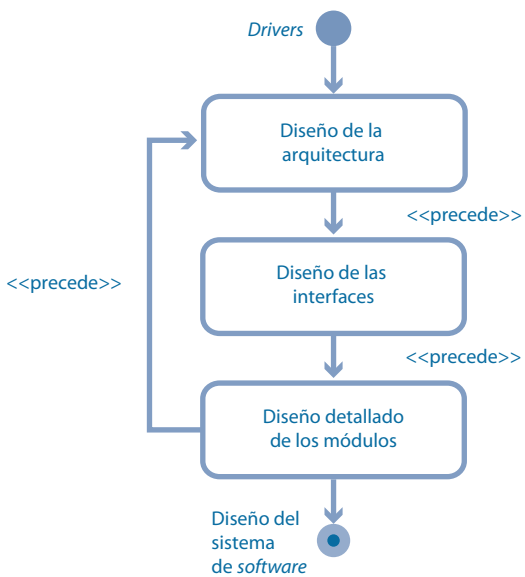
Podemos relacionar esta definición con diversos aspectos referentes a la arquitectura que hemos tratado hasta el momento:

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



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

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

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

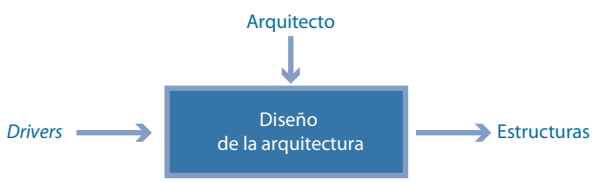
3.2 PROCESO GENERAL DE DISEÑO DE LA ARQUITECTURA

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

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

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

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

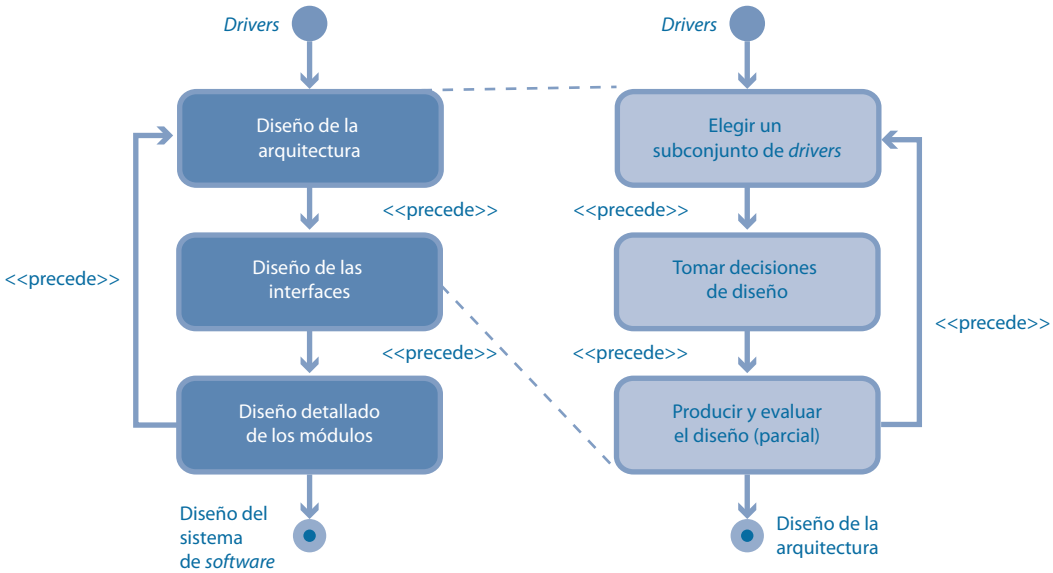


› **Figura 3-3**. Diseño como caja negra (las entradas están a la izquierda, las salidas, a la derecha, y los responsables, arriba).

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

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

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



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

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

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

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

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

3.3 PRINCIPIOS DE DISEÑO

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

3.3.1 Modularidad

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

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

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

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

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

3.3.2 Cohesión alta y acoplamiento bajo

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

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

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

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

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

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

3.3.3 Mantener simples las cosas

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

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

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

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

3.4 CONCEPTOS DE DISEÑO

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

3.4.1 Patrones

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

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

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

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

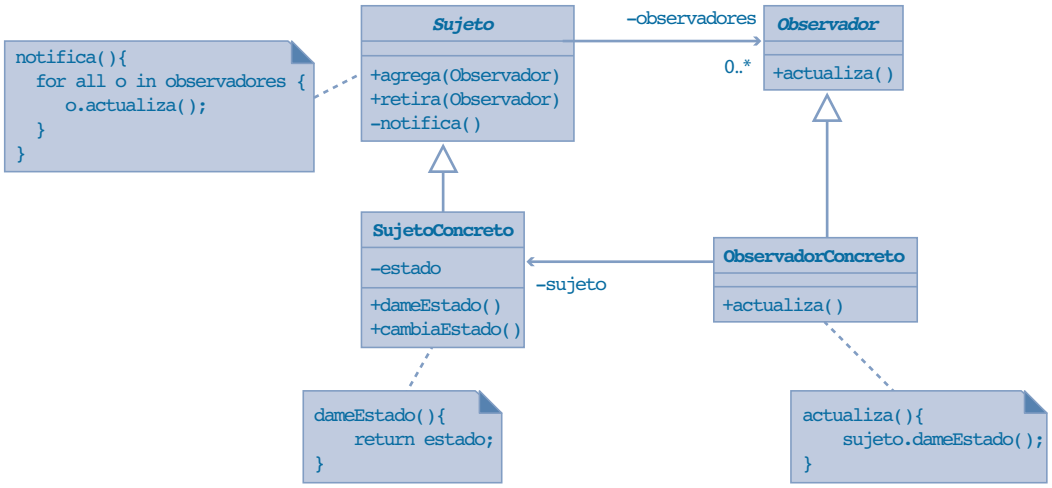
Desde la aparición del libro de GoF han surgido muchos catálogos con patrones de diseño de *software* enfocados en distintas áreas. Existen, por ejemplo, catálogos de patrones dirigidos a la estructuración general del sistema, también llamados *estilos arquitectónicos* o *arquitecturas de referencia*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

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

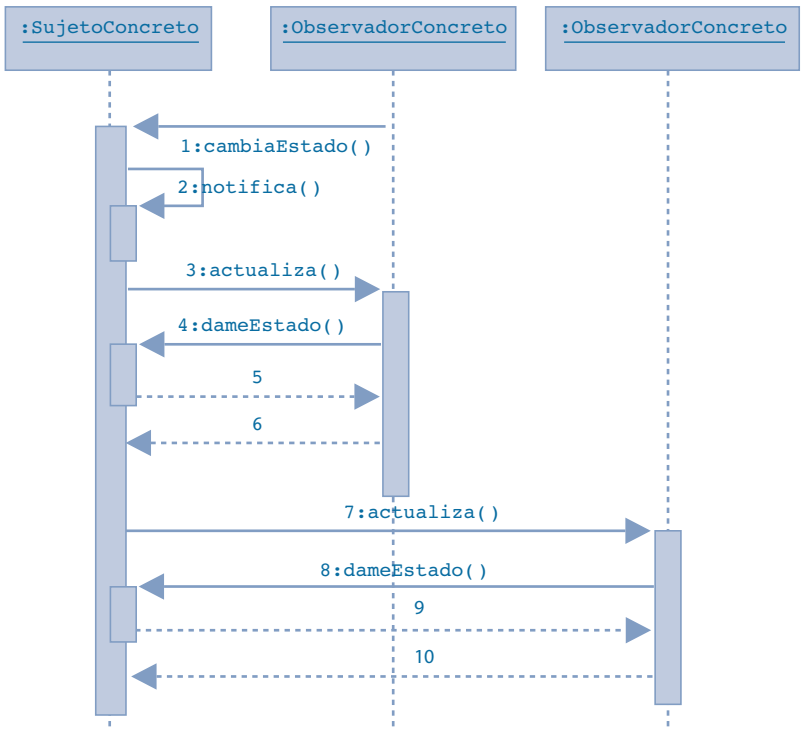
La figura 3-5 muestra un ejemplo de patrón de diseño llamado Observador (*Observer*), proveniente del libro de GoF, cuyo propósito es "definir una dependencia entre objetos de uno a muchos, de modo que cuando el estado del objeto con dependientes múltiples cambie, todos ellos sean notificados y actualizados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

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

a) Modelo estructural:



b) Modelo de comportamiento:



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

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

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

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

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

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

3.4.2 Tácticas

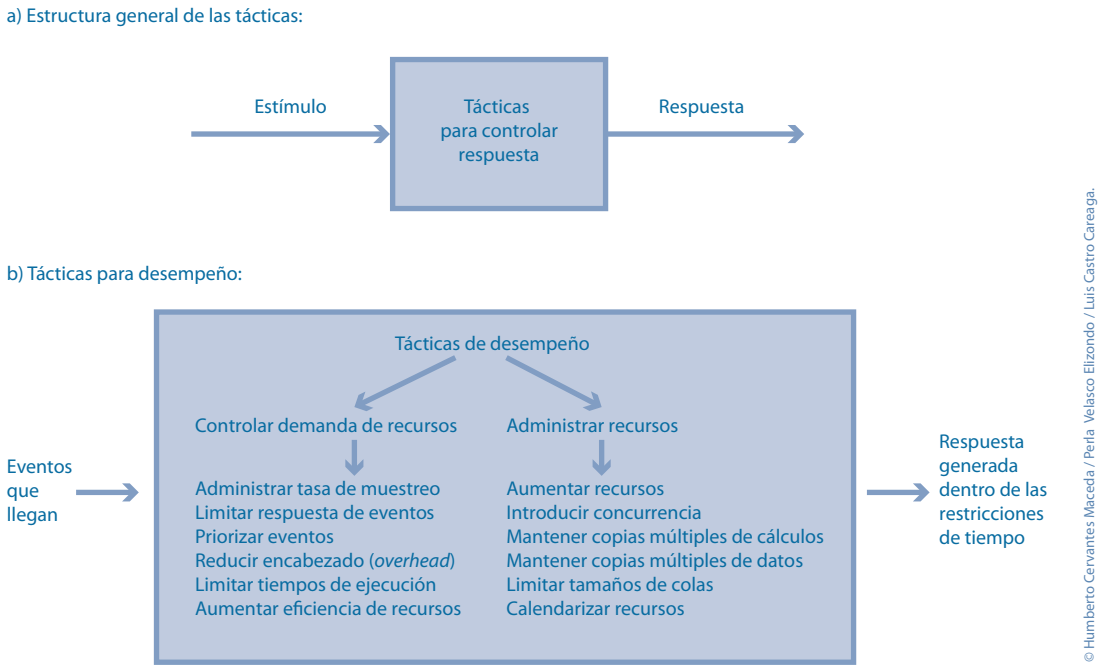
Las tácticas son conceptos de diseño que influyen sobre el control de la respuesta a un atributo de calidad particular. A diferencia de los patrones, no presentan soluciones conceptuales detalladas, sino que son técnicas probadas de las ciencias de la computación con las que se resuelven problemas en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

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

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

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

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



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

3.4.3 *Frameworks*

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

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

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

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

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

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

3.4.4 Otros conceptos de diseño

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

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

3.5 DISEÑO DE LAS INTERFACES

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

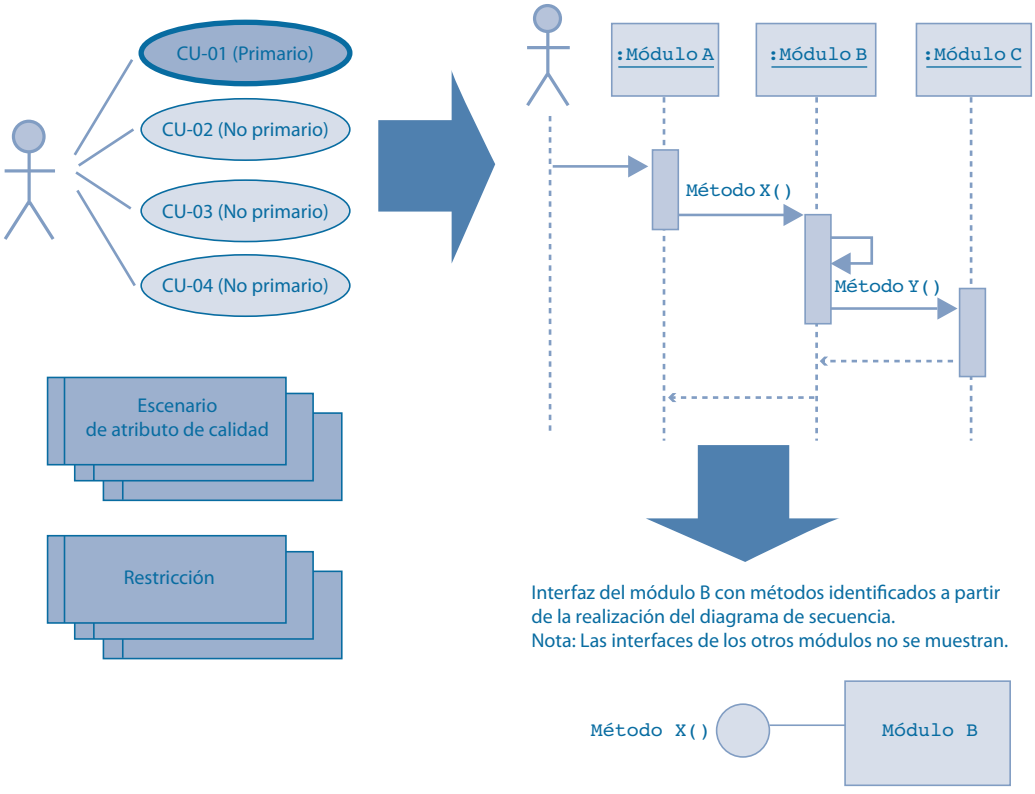
Al momento de generarse estas estructuras dinámicas se identifican los mensajes que intercambian los elementos que colaboran en la interacción. El conjunto de mensajes que recibe un elemento conforma la interfaz de este, es decir, el contrato al que debe apegarse a efecto de participar en la interacción.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

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

a) Interfaces que se identifican durante el diseño arquitectónico:

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

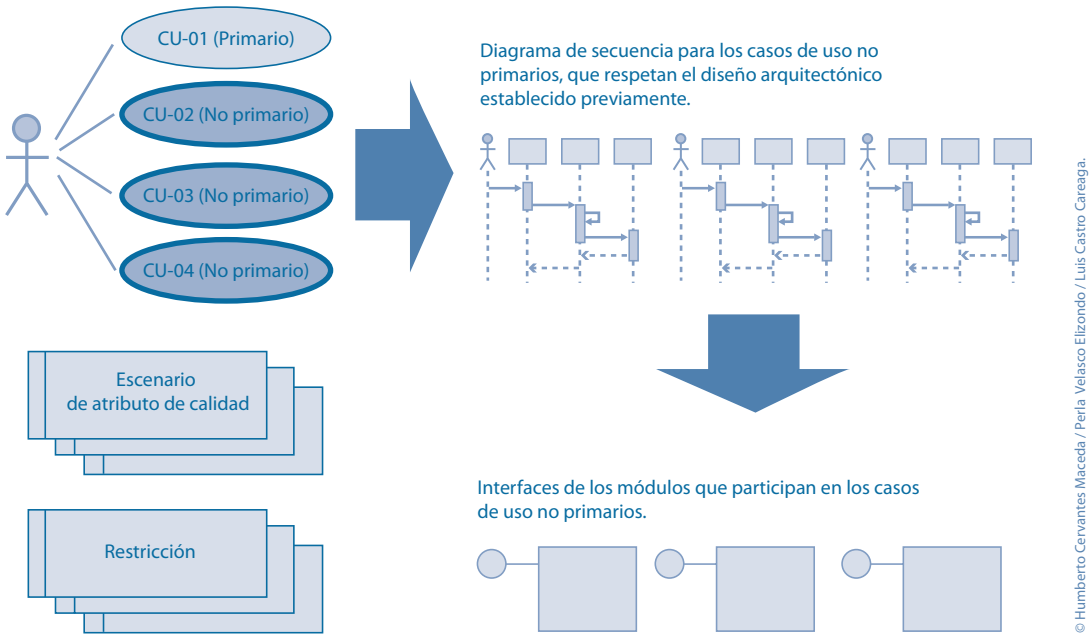
Diagrama de secuencia para CU-01(primario), de acuerdo a las estructuras físicas y lógicas (no mostradas).



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

b) Interfaces que se identifican posteriormente al diseño arquitectónico:

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



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

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

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

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

3.6 MÉTODOS DE DISEÑO DE ARQUITECTURA

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

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:

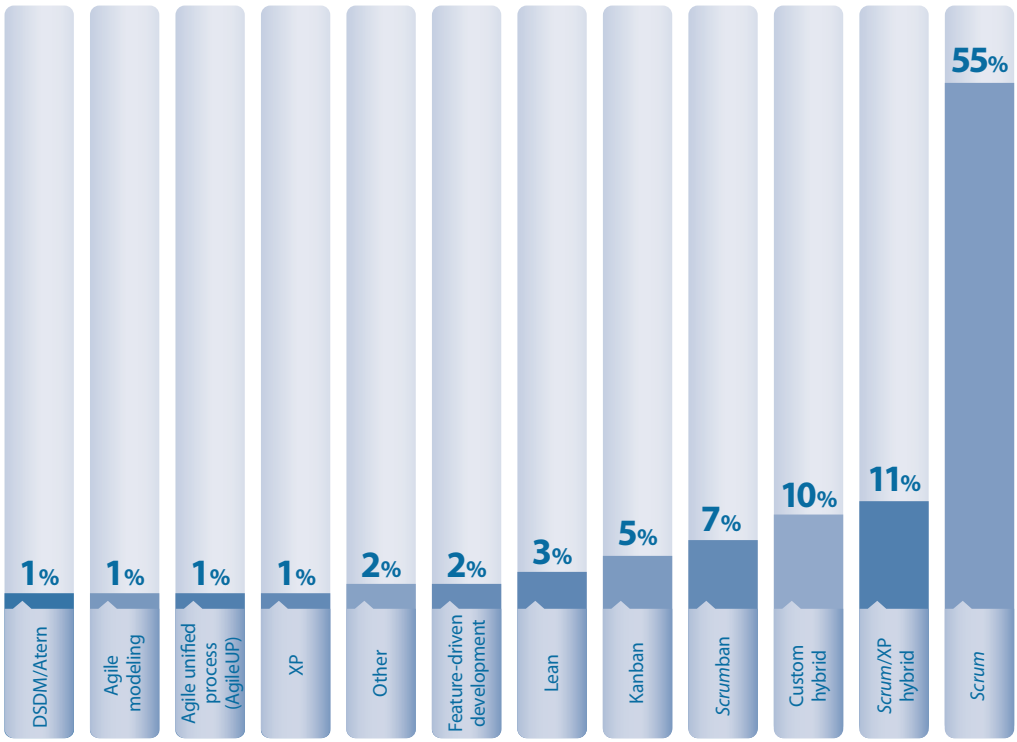
1. Satisfacer al cliente por medio de la entrega temprana y continua de *software* funcional.
2. Son bienvenidos los requisitos cambiantes, incluso si llegan tarde al desarrollo.
3. Entregar con frecuencia *software* funcional, prefiriendo periodos de semanas y no de meses.
4. Trabajo conjunto y cotidiano 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.

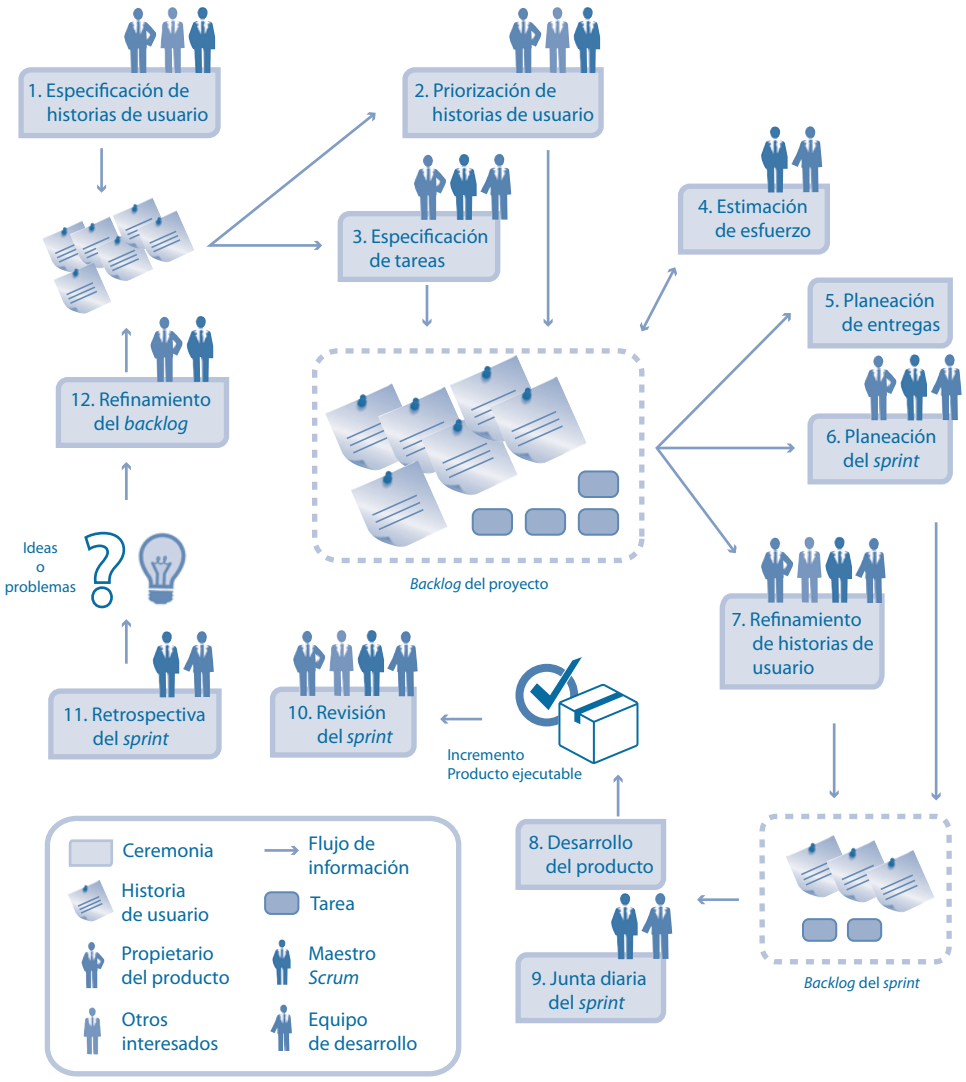
En *Scrum*, una iteración recibe el nombre de *sprint*, y su duración oscila por lo habitual entre una y cuatro semanas. En cada uno de ellos se lleva a cabo un conjunto de ceremonias organizadas en un patrón de actividades del tipo planeación-desarrollo-demostración-retrospectiva. Sin embargo, hay otras 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-



›**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.

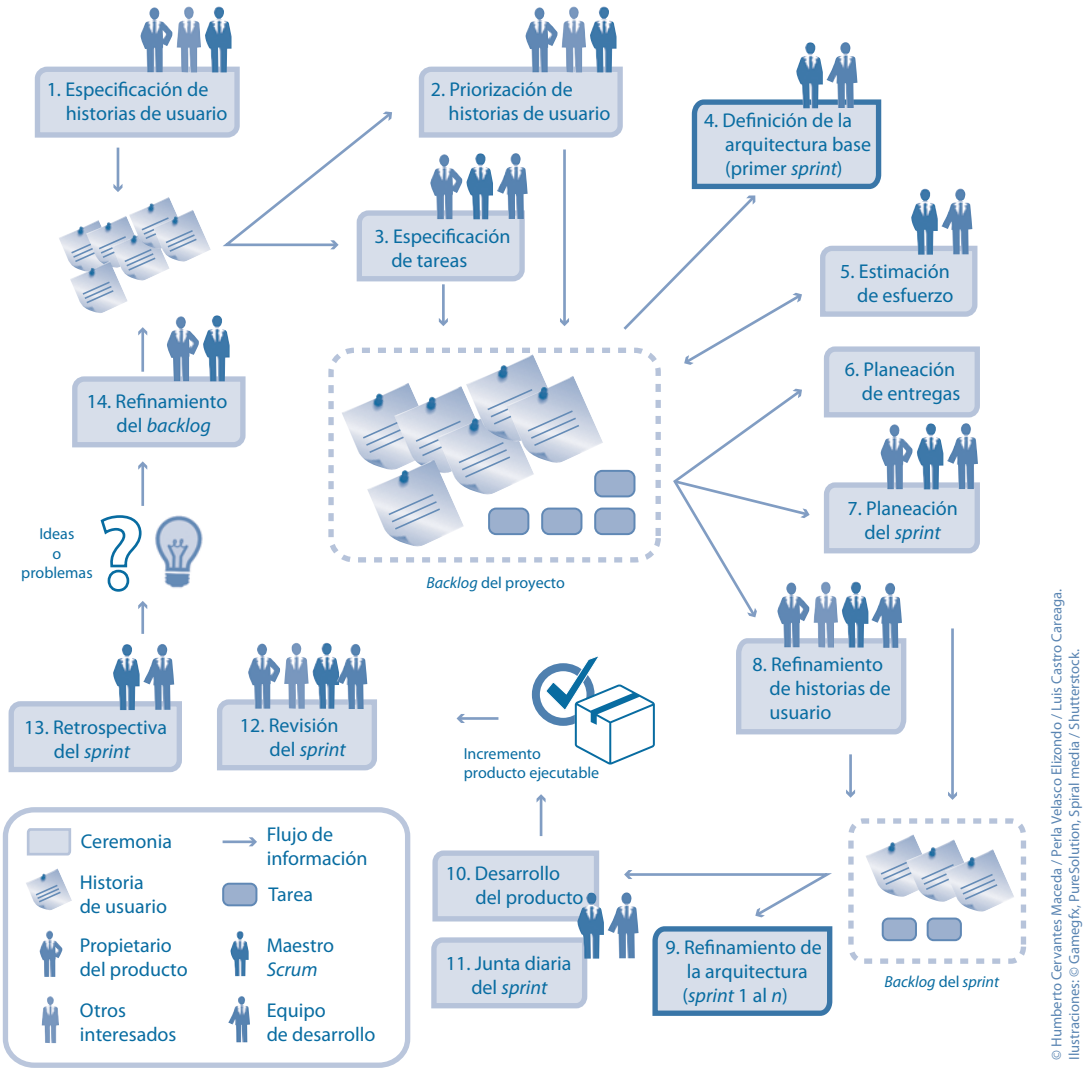
Aunque en la práctica la deuda técnica es inevitable debido a diversos factores, que van desde requerimientos cambiantes hasta la ignorancia del equipo de desarrollo respecto de que está adquiriendo una deuda de esa clase (lo cual puede ocurrir con equipos con poca experiencia), realizar acciones para 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.



› **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

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.

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

CENGAGE Learning®

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