Una Teoría para el Diseño de Software

Maximiliano Cristiá
Ingeniería de Software
Licenciatura en Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Mayo de 2010

Índice

1. ¿Por qué diseñar software... y no más bien no hacer nada? 1
- 1.1. Diseño para el cambio . . . 3
- 1.2. Calcular el diseño . . . 3

2. El diseño como fase del ciclo de desarrollo del sistema 4
- 2.1. Diseño y métodos formales . . . 4

3. Los tres niveles estructurales 5

4. Breve introducción histórica 6

5. El ciclo de vida de la arquitectura 7
- 5.1. La influencia de los requerimientos no funcionales en la definición de la arquitectura de un sistema . . . 8

6. La diferencia entre arquitectura y diseño 9

7. Estilos arquitectónicos 10

1. ¿Por qué diseñar software... y no más bien no hacer nada?

Definición 1 (Diseño de Software). *Diseñar un sistema de software significa:*

- *Descomponerlo o dividirlo en elementos de software.*

  Es decir, pensar al software como un conjunto de elementos que interactúan entre sí, y no como un bloque indiviso.

- *Asignar y describir una función para cada uno de esos elementos.*

- *Establecer las relaciones entre esos elementos.*

  Las relaciones entre los elementos representan las posibles interacciones entre ellos. Un elemento de software puede ser referenciado, usado o accedido por otros elementos de muchas formas diferentes. Cada una de estas posibilidades es una relación entre elementos de software.

El resultado de diseñar un sistema es una de las tantas descripciones que debe realizar un ingeniero de software. A esta descripción se la conoce como *diseño de software*, *arquitectura de software* o *estructura de software*. Entonces diseño, arquitectura o estructura de software son términos más o menos similares que usaremos como sinónimos a menos que se indique algo más preciso.

Pero, ¿por qué es conveniente diseñar un software? ¿Por qué simplemente no se programa el software sin pensar en diseñarlo? ¿No es así como trabaja la mayoría de los grupos de programadores? ¿Por qué perder tiempo en pensar cómo dividir un programa en partes, cuando el tiempo en la industria del software es siempre tan escaso?

La respuesta es muy sencilla: es conveniente dedicar tiempo y esfuerzo en diseñar un sistema de software porque es más barato desarrollarlo y mantenerlo, que hacerlo como se lo hace habitualmente. En los siguientes párrafos estudiaremos brevemente algunos aspectos económicos o industriales relativos a la producción de software que, en nuestra opinión, corroboran la afirmación anterior.

La producción de un sistema de software se puede dividir en dos grandes etapas: desarrollo y mantenimiento. Durante el desarrollo el programa se escribe desde cero (o casi si se utilizan porciones de código previamente desarrolladas) hasta que se entrega al cliente la primera versión. Durante el mantenimiento se desarrollan nuevas versiones hasta que el sistema se vuelve obsoleto y es descartado. Se estima que en promedio el desarrollo ocupa el 33 % del esfuerzo total de producción en tanto que el 67 % restante se dedica al mantenimiento, aunque diversos autores y estudios difieren en las cantidades exactas [1, 2, 3]. El porcentaje dedicado al mantenimiento puede ser notablemente más alto si dentro de él se considera la creación de una línea de productos.

El mantenimiento consiste esencialmente en cambiar, agregar o eliminar líneas de código fuente al software tal y como está en cada momento. Esto se hace tanto para corregir, adaptar o mejorar la versión actual. Dado que normalmente los requerimientos para el sistema no se conocen de antemano, sino que se van conociendo a medida que el sistema se usa, la mayor parte del mantenimiento se trata de incorporar a la versión actual estos nuevos requerimientos. Esto ocurre, incluso, durante el desarrollo pues una cantidad no despreciable de requerimientos van apareciendo a medida que el cliente ve los primeros prototipos o versiones no productivas. En resumen, gran parte del costo asociado con la producción de un sistema de software proviene de la necesidad de incorporar cambios a una versión del sistema (antes o después de la entrega) con el fin de:

- Cambiar, agregar o eliminar código fuente
- Corregir, adaptar, mejorar
- Elaborar una línea de productos
- Agregar requerimientos tardíos
- Modificar requerimientos erróneos

Dentro de los costos originados por la incorporación de cambios, el más alto es el ocasionado por la necesidad de tener que volver a testear todo o una gran parte del sistema cada vez que se introduce un cambio.

Claramente, entonces, los cambios no son sólo inevitables o inherentes a la producción de software sino que en muchas ocasiones son un excelente negocio. Por lo tanto no tiene sentido buscar estrategias para reducir la cantidad de cambios. Por el contrario, se deben buscar principios, metodologías, técnicas y herramientas de forma tal de cumplir con dos objetivos básicos de diseño:

- **Incorporar cada cambio con el menor costo posible**
- **Evitar que cada cambio degrade la integridad conceptual del sistema**

  Si esto no se tiene en cuenta, incorporar los primeros cambios puede ser barato pero cada vez será más costoso.

Es imposible lograr los objetivos anteriores sin haber tenido en cuenta los posibles cambios que sobrevendrán. En consecuencia se deben *analizar los cambios posibles, la línea evolutiva del sistema, los productos que de él pueden derivarse*, etc. Dado que introducir cambios tiene que ver con modificar líneas de código fuente, entonces es *fundamental descomponer, dividir u organizar bien el código fuente, es decir, diseñarlo bien*. Por lo tanto, dedicar esfuerzo para diseñar el sistema resulta más económico que no hacerlo porque permitirá que los costos de mantenimiento, e incluso parte del costo de desarrollo, disminuyan sensiblemente dado que se trata fundamentalmente de introducir cambios. Y como el costo de mantenimiento es por mucho el mayor de todos, entonces cualquier estrategia que tienda a reducirlo es esencial a la producción de software.

El diseño tiende a reducir los costos de mantenimiento porque reduce el costo de cambio, que es la principal fuente de costos, en consecuencia es una actividad esencial a la producción de software.

Por lo tanto, es imperativo definir una *teoría de diseño* que tenga en cuenta los costos derivados de la necesidad de introducir cambios al sistema.

1.1. Diseño para el cambio

Uno de los principios de la Ingeniería de Software es el principio de *Diseño para el Cambio*. Este principio sugiere anticipar en el diseño del sistema los cambios que probablemente se quieran incorporar en el futuro. Simplemente dice que hay que pensar y poner esfuerzo en anticipar los cambios más probables y hacer algo para reducir su impacto en el momento en que haya que incorporarlos al sistema. No dice cómo lograrlo, pero si lo dijera no sería un principio. Claramente, de todo lo antedicho se deduce que cualquiera sea la teoría de diseño que se elija, esta debe basarse en el principio de Diseño para el Cambio.

1.2. Calcular el diseño

Uno de los aspectos fundamentales de cualquier teoría de diseño de software es que los ingenieros deben tener claro que un diseño tiene una componente importante de intuición, experiencia y creatividad, pero que tiene una componente más importante de cálculo. Es decir, el diseño debe ser fruto de alguna regla de cálculo. Por eso se habla de *diseño calculado*[1]. Claramente esta regla de cálculo no será completamente formal (matemática) ni será sencilla, pero sí debe ser objetiva y clara.

Una de los objetivos esenciales de cualquier regla de cálculo para el diseño de software debe ser estipular claramente el criterio por el cual el sistema debe descomponerse en 2, 3 o 534 elementos de software, por qué tienen que ser esos elementos y no otros, por qué esos elementos tienen que tener tales y cuales funciones y no otras, porqué esos elementos se tienen que relacionar de tales o cuales maneras y no de otras. Como se muestra en la Figura 1 el diseño un sistema puede tener una cantidad cualquiera de elementos de software. El ingeniero se enfrenta entonces al dilema de determinar no solo el número correcto de elementos en que debe descomponer el sistema sino las características estructurales fundamentales de esos elementos. Esto debería hacerse de una manera más o menos objetiva siguiendo una serie de pasos claros y que siempre lleven a un diseño, sino excelente, muy bueno.

Este criterio, llamado *criterio de descomposición* o *criterio de diseño*, es el núcleo de cualquier teoría de diseño de software, es la regla de cálculo, es la herramienta básica de cualquier ingeniero de software que trabaje en diseño. El criterio de diseño le da al ingeniero una regla por medio de la cual hacer la misma cosa de dos formas diferentes:

1. Determinar los elementos de software que debe tener su diseño y las características fundamentales de esos elementos (interfaces, relaciones, especificaciones, etc.).

[1] Traducción de *calculational design*.

Figura 1: ¿En cuántos y en cuáles elementos de software debe ser dividido un sistema? La respuesta debe darla el criterio de diseño.

2. Verificar si un diseño dado es correcto o no, puesto que puede analizar si ese diseño es el resultado de haber aplicado la regla.

Cualquier criterio de diseño debe someterse al principio de Diseño para el Cambio de forma tal de, repetimos, cumplir los dos objetivos básicos de diseño:

- **Incorporar cada cambio con el menor costo posible**
- **Evitar que cada cambio degrade la integridad conceptual del sistema**

porque de lo contrario ese criterio no estaría teniendo en cuenta la justificación industrial que sustenta la necesidad de diseñar el software.

Ejercicio 1. Suponga que un ingeniero decide diseñar el problema de la guardia de hospital como un único programa sin ninguna subrutina y que otro lo diseña con una subrutina cada cinco líneas de código. Analice brevemente las ventajas y desventajas de cada opción y determine si alguna de ellas tuvo en cuenta el principio de Diseño para el Cambio y los dos objetivos de diseño que mencionamos más arriba. Justifique su respuesta.

2. El diseño como fase del ciclo de desarrollo del sistema

El diseño de software forma parte de la fase del ciclo de desarrollo del sistema denominada Arquitectura del Sistema. Considerando el modelo de desarrollo de cascada, la fase de definición de la arquitectura se ejecuta luego de la Ingeniería de Requerimientos. Es decir que después de contar con una lista más o menos estable de requerimientos, los arquitectos del sistema están en condiciones de pensar, concebir y documentar una arquitectura para el sistema.

2.1. Diseño y métodos formales

Como ya hemos visto, un modelo funcional describe los requerimientos del usuario mediante algún formalismo lógico o matemático. Por lo tanto, el diseño de un sistema y su modelo funcional son dos *descripciones* diferentes. En otras palabras son dos formas diferentes pero complementarias de ver un sistema. Podemos decir que el diseño y el modelo representan dos dimensiones ortogonales de un sistema como lo sugiere la Figura 2.

En efecto, dado un cierto conjunto de requerimientos, un sistema que los satisfaga funcionalmente puede haber sido diseñado de varias formas diferentes. Por ejemplo, un sistema operativo puede haber sido diseñado monolíticamente o estratificadamente. Es decir, un modelo funcional puede ser estructurado de diferentes formas; el modelo funcional no fija necesariamente una única forma de descomponer el sistema. Más aun, en la mayoría de los casos es un error tomar el modelo funcional como diseño del sistema.

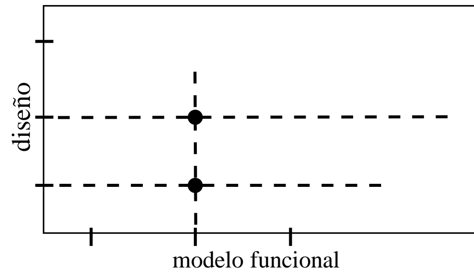

Figura 2: A un cierto modelo funcional puede corresponderle más de un diseño. Por ejemplo, un sistema operativo puede diseñarse monolíticamente o en estratos.

Pero entonces, ¿dónde encajan en el diseño los lenguajes formales para especificación que vimos anteriorment? Básicamente se trata de que la función deseada de cada elemento del diseño se especifique con uno o varios de esos lenguajes. En efecto, la función que se le asigna a cada elemento del diseño es una porción de los requerimientos o ayuda a implementar parte de los requerimientos. Por ejemplo, en un sistema de gestión para PyMES una parte del sistema será la encargada de la facturación, entonces se podría usar, digamos, Z para especificar el significado de "facturar".

De todas formas, no siempre es necesario o económicamente viable o conveniente especificar formalmente la función de todos los elementos en que se descompone un sistema. Justamente, el hecho de primero diseñar (descomponer) y luego especificar, permite que antes de especificar se seleccionen las porciones más complejas, críticas o convenientes con el fin de ser especificadas formalmente. A las restantes se les adosará una especificación informal.

Ejercicio 2. Considere la definición de diseño 1 y el problema de la guardia de hospital visto anteriormente. Describa coloquialmente dos diseños para ese problema y detalle al menos una relación entre algunos elementos de sus diseños.

3. Los tres niveles estructurales

En la teoría de diseño que veremos más adelante, la estructura de un sistema de software se realiza o define en tres niveles de abstracción diferentes, listados a continuación desde el más abstracto al más detallado[2]:

1. Elección del estilo arquitectónico

   Cada estilo arquitectónico representa a una familia de sistemas de software que comparten características estructurales fundamentales. Mediante un estilo arquitectónico se puede definir a grandes rasgos la estructura de un sistema de software de dimensión industrial. Existe alrededor de una docena de estilos arquitectónicos.

2. Selección de los patrones de diseño

   Cada patrón de diseño representa una solución general reusable para un problema que se da con frecuencia en el diseño de software. No es posible en general definir la estructura completa de un software usando uno o dos patrones de diseño. Existen tal vez más de 200 patrones de diseño.

3. Diseño de componentes

[2] Se incluye una sumarísima explicación de cada concepto para que el lector tenga una idea muy superficial de lo que hablamos. El desarrollo en detalle de cada uno de estos niveles es, precisamente, el objetivo del resto de este capítulo.

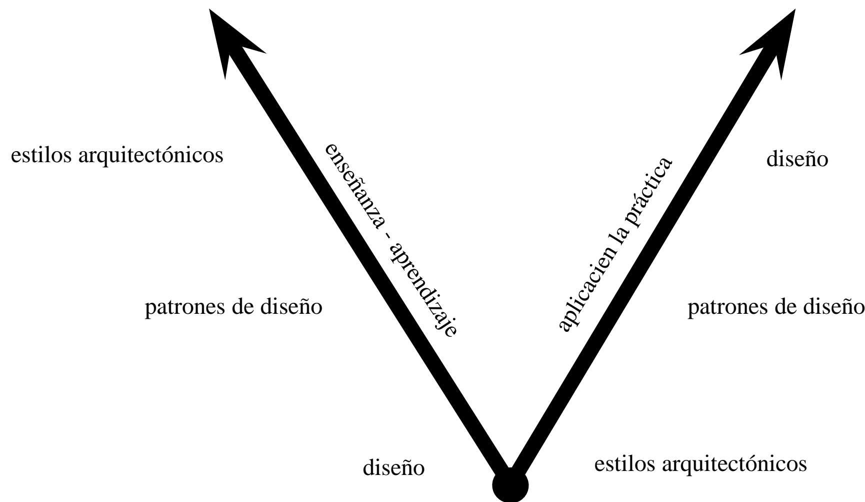

Figura 3: Las etapas de la fase Arquitectura del Sistema se enseñan-aprenden de forma opuesta a la forma en que se aplican en la práctica.

Cuando un ingeniero de software aborda la definición de la estructura de un sistema debe comenzar por trabajar en el más alto nivel de abstracción para luego ir refinando dicha estructura incorporando más detalles. Usualmente la primera decisión que debe tomar el ingeniero es el o los estilos arquitectónicos en los que se basará la estructura del sistema, la función que tendrán los componentes principales de aquellos y sus relaciones. Luego podrá seleccionar patrones de diseño para finalizar definiendo los componentes de menor nivel de abstracción o dando más detalles sobre los componentes ya definidos. Es muy común que se deban realizar varias iteraciones entre los niveles 2 y 3; si bien no debería ser frecuente, puede ocurrir que el estilo arquitectónico deba ser modificado. Esto último puede darse cuando los requerimientos desde los que parte el arquitecto tienen un nivel de calidad muy pobre. Aun teniendo que alterar la primera decisión estructural, es siempre significativamente más barato recalcular o redefinir el diseño que tener que hacerlo una vez que hay grandes porciones de código ya escritas. Por consiguiente, el equipo de arquitectos debe disponer de un tiempo razonable para describir la arquitectura con un nivel de detalle suficiente como para tener cierta confianza de que los cambios más probables podrán ser incorporados a bajo costo y sin degradar la integridad conceptual del sistema.

Nos interesa resaltar que, por otra parte, el proceso de enseñanza-aprendizaje de la teoría de diseño de software discurre en el sentido opuesto al que se aplica en la práctica como se intenta graficar en la Figura 3. En otras palabras, se enseña (aprende) primero a diseñar componentes, luego se aprende (enseña) a utilizar patrones de diseño y finalmente se aprende (enseña) lo relativo a estilos arquitectónicos.

4. Breve introducción histórica

En los primeros años de la construcción de software no existía el diseño del sistema como una etapa independiente de la programación. Pensar la descomposición del programa antes de programar se comienza a proponer, investigar y aplicar a principios de la década del 70 cuando se comienza a distinguir entre pequeños y grandes programas o sistemas [4]. Precisamente uno de los pioneros en esté área fue David L. Parnas con sus trabajos seminales en ocultación de información y desarrollo de familias de programas [5, 6, 7], seguido por el trabajo de Liskov y Guttag referido al diseño basado en tipos abstractos de datos [8, 9, 10]. Más o menos en la misma época pero con un impacto en la industria mucho mayor Tom DeMarco, Edward Yourdon, Larry Constantine y Glenford Myers

desarrollaron y consolidaron el análisis y diseño estructurado [11, 12, 13].

Los patrones de diseño nacen como un concepto de la arquitectura y la ingeniería civil a partir de los trabajos del arquitecto Christopher Alexander entre 1977 y 1979. En 1987 Kent Beck and Ward Cunningham comienzan a experimentar con la idea de aplicar patrones de diseño al desarrollo de sistemas de software [14, 15]. Los patrones de diseño como una rama del diseño orientado a objetos ganaron popularidad luego de la publicación de [16] en 1994. Ese mismo año se realizó la primera *Pattern Languages of Programming Conference*.

Recién a principios de la década del 90 algunos investigadores, principalmente ligados a la Carnegie-Mellon University y al Software Engineering Institute, comienzan a ver la necesidad de investigar y desarrollar un nivel de abstracción superior al del diseño, al que llamaron *arquitectura de software*. Este trabajo se condensa principalmente en el libro pionero de Mary Shaw y David Garlan [17] de 1996 y en el volumen más elaborado de Len Bass, Paul Clements y Rick Kazman [18] publicado en 1998. Precisamente la segunda edición de este último [19] junto a otro volumen más especializado producido también en el SEI [20] y al trabajo de un grupo de ingenieros de software de Siemens [21] son tal vez las expresiones más modernas y acabadas de la especialidad.

5. El ciclo de vida de la arquitectura

En esta sección veremos cómo la arquitectura del sistema afecta a su entorno y este también afecta a la arquitectura. El material de esta sección es un resumen del capítulo 1 de [19].

Antes de analizar las influencias mutuas entre la arquitectura y su entorno, es conveniente introducir el concepto de *interesado* o *involucrado* en el sistema[3]. Los interesados en una arquitectura de software de un cierto sistema son todas aquellas personas u organizaciones que tienen alguna injerencia o interés en el sistema. Un listado incompleto es el siguiente: programadores, administradores, testers, usuarios finales, dueño del sistema, organizaciones con las cuales el sistema interactúa, bancos que financian la construcción del sistema, etc. Nunca se repetirá la cantidad de veces suficiente lo importante que es determinar con la mayor precisión posible el conjunto de interesados en el sistema antes de comenzar a delinear su arquitectura. Determinar el conjunto de interesados en el sistema es responsabilidad o injerencia de los ingenieros a cargo de la Ingeniería de Requerimientos, como primer paso antes de comenzar con la captura de los requerimientos. Tanto los ingenieros de requerimientos como el equipo responsable de la arquitectura del sistema, más tarde o más temprano, deberán interactuar con todos y cada uno de los interesados para validar diferentes aspectos del sistema. Un interesado relevante que no sea consultado tempranamente sobre la arquitectura del sistema, será una fuente de problemas en el futuro.

Las influencias mutuas entre la arquitectura de un sistema y su entorno se denominan *Architecture Business Cycle* o ABC y se condensan en la Figura 4. La arquitectura de software de un sistema es el resultado de combinar decisiones técnicas, sociales y del negocio. Los ingenieros de software deben convencerse que su trabajo está influenciado por el negocio y el entorno social que los rodea. Es ilusorio e ingenuo suponer que uno podrá determinar la arquitectura únicamente basándose en consideraciones técnicas. Los interesados, algunos de ellos más que otros, más tarde o más temprano presionarán al arquitecto del sistema para que la arquitectura tenga tal o cual característica que no necesariamente es la mejor desde el punto de vista técnico. Por ejemplo, el gremio de empleados estatales puede presionar para que un sistema de administración impositiva preserve una interfaz con el usuario en modo texto dado que la mayor parte de los usuarios finales (es decir los afiliados al gremio) que domina esta tecnología es renuente a capacitarse o adaptarse a interfaces más modernas. Algo semejante ocurre con las influencias provenientes de la organización que desarrolla el sistema. Por ejemplo, si la mayor parte de los programadores tiene cierta fluidez en Java y toda su tecnología, diversas gerencias y grupos dentro de la organización presionarán explícita o implícitamente para que

[3]Ambos términos son nuestra traducción para *stakeholder*.

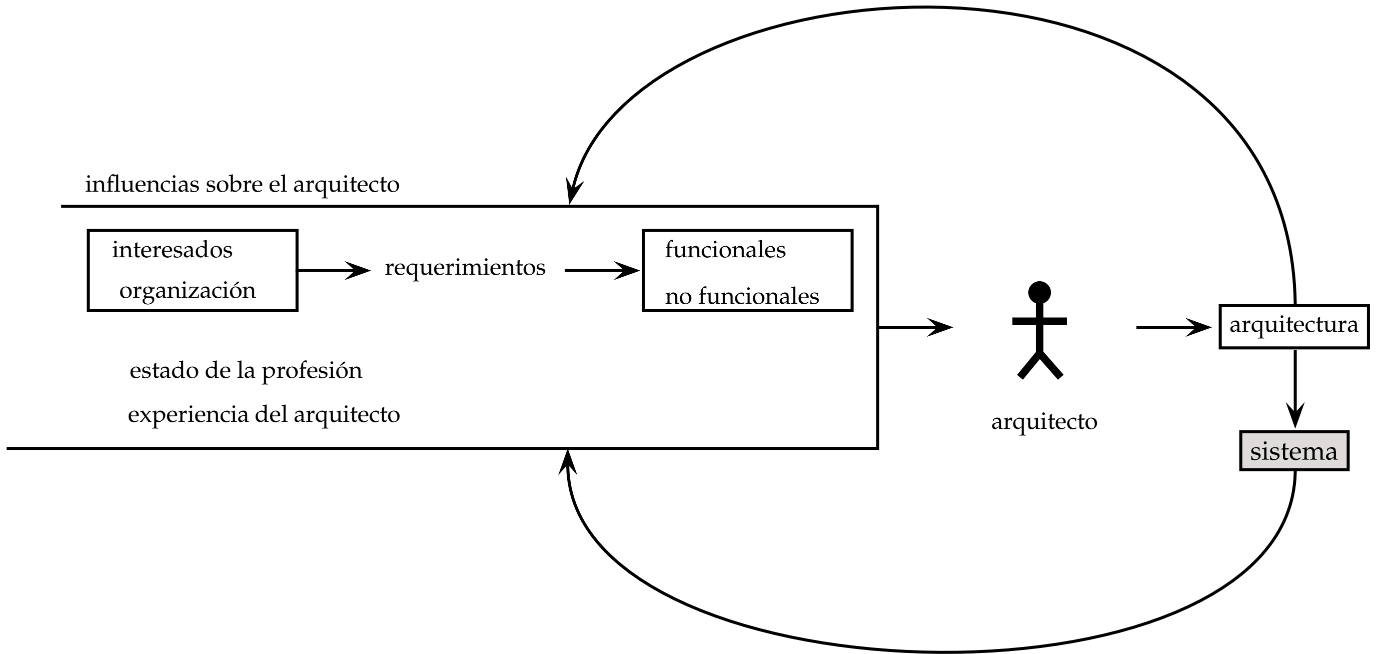

Figura 4: Influencias mutuas entre la arquitectura y su entorno.

la arquitectura se oriente hacia esa tecnología. Los conocimientos técnicos del arquitecto así como también el estado del arte de la práctica profesional de la Ingeniería de Software también son fuertes condicionantes no técnicos a la hora de definir la arquitectura de un sistema. Si el arquitecto domina el DOO entonces sus arquitecturas tenderán a seguir esa técnica; si se está ideando un sistema de comercio electrónico, la industria y la cultura técnica imperantes presionan para utilizar webservices independientemente de que sea la tecnología óptima para el sistema en cuestión.

El ciclo ABC se cierra pues una arquitectura exitosa tenderá a convertirse en la referencia obligada dentro de la organización para estructurar sistemas semejantes o elaborar líneas de productos. Asimismo los usuarios finales o los clientes serán renuentes a definir una nueva arquitectura para un nuevo sistema si la anterior fue la base para un sistema que les brindó buenas prestaciones. Finalmente, ciertos sistemas basados en ciertas arquitecturas logran rebasar las fronteras de una organización y se convierten en estándares de-facto o referencias obligadas de la industria por lo que terminan teniendo una influencia importante sobre la práctica profesional (tal es el caso, por ejemplo, de la tecnología Web la cual fue pensada por Tim Berners-Lee para compartir los resultados de la investigación dentro del Laboratorio Europeo de Física de Partículas (CERN) de Ginebra pero que terminó convirtiéndose un una tecnología de alcance universal).

5.1. La influencia de los requerimientos no funcionales en la definición de la arquitectura de un sistema

En la Figura 4 se presenta a los requerimientos como una influencia determinante en la concepción de la arquitectura de software del sistema. Más aun los requerimientos se dividen en dos grandes clases: *funcionales* y *no funcionales* (también llamados *cualidades* o *atributos de calidad* del sistema). Normalmente los requerimientos funcionales no ocupan solo la primera posición sino la única a la

hora de definir la arquitectura del sistema. Si bien hasta cierto punto esto puede ser razonable suele ser una fuente inagotable de problemas. Usualmente requerimientos no funcionales o cualidades tales como modificabilidad, seguridad, desempeño, tolerancia a fallas, testeabilidad, etc. no son tenidas en cuenta ni por los arquitectos del sistema ni por la mayoría o todos los interesados. En general la necesidad de que el sistema verifique algunas de estas cualidades se percibe recién cuando este entra en producción o en las etapas finales del desarrollo.

El problema de esta situación radica en que por lo general lograr que un sistema casi terminado tenga un desempeño superior, sea modificable, seguro o tolerante a fallas (ni que hablar de que verifique varias de estas cualidades simultáneamente) es casi imposible si no se pensó desde su concepción, es decir si esas cualidades no se incorporaron conscientemente en la arquitectura del sistema. No ocurre lo mismo, en la mayoría de los casos, con el requerimiento de agregar nuevas funcionalidades; suele ser costoso pero en general no requiere rehacer el sistema por completo. Los requerimientos funcionales por lo general son *discretos* en el sentido de que agregando o modificando algunas líneas de código en unos pocos lugares es suficiente para implementarlos; mientras que los requerimientos no funcionales son, por lo común, *densos* en el sentido de que es necesario agregar o modificar código en todas partes para implementarlos. En consecuencia no prever un requerimiento no funcional suele ser mucho más costoso que no tener en cuenta un requisito funcional. Actualmente se considera que los requerimientos no funcionales deben guiar la definición de la arquitectura del sistema tanto como los funcionales.

6. La diferencia entre arquitectura y diseño

En la sección 3 decíamos que la estructura de un sistema de software debe abordarse según tres niveles de abstracción: estilos arquitectónicos, patrones de diseño y diseño de componentes. Si bien son tres niveles de abstracción diferentes, la mayor diferencia se da entre los patrones de diseño y el diseño de componentes, por un lado, y los estilos arquitectónicos por el otro. Sin embargo, la definición 1 no da pistas sobre cómo abordar esos tres niveles, ni cómo expresar la estructura de un sistema en cada nivel, ni cuál es la diferencia entre ellos (particularmente entre los dos inferiores y el nivel arquitectónico). Por este motivo proponemos una definición específica para el primer nivel estructural.

Definición 2 (Nivel Arquitectónico)**.** *El nivel arquitectónico de la estructura de un sistema es aquella descripción donde se utilizan conectores diferentes a llamada a procedimiento y/o se imponen restricciones estructurales importantes entre los componentes y/o se utilizan distintos tipos de componentes.*

Antes de analizar la definición clarificaremos algunos conceptos que en ella aparecen:

Componente. Entidad computacional activa que implementa algún requisito funcional o parte de este.

Conector. Mecanismo que mediatiza la comunicación, coordinación o cooperación entre componentes.

Tipo componente. Componentes que comparten características estructurales, en particular, y fundamentalmente, los mismos tipos de interfaz.

Tipo interfaz. Forma de interacción con el entorno semántica y estructuralmente única.

Hechas estas aclaraciones analizaremos brevemente la definición propuesta. En primer lugar notar que la definición no habla de la arquitectura del sistema sino de una *descripción* particular de la estructura, por lo que decimos que esta definición refina o complementa a la definición 1. Observar

que la definición refiere a las mismas actividades que se llevan a cabo en el diseño solo que en un nivel de abstracción diferente. En este sentido, la arquitectura *es* diseño.

La diferencia distintiva con respecto al nivel del diseño radica en que según nuestra definición la descripción del nivel arquitectónico implica el uso de elementos de software que no tienen una representación directa en la mayoría de los lenguajes de programación; es decir, elementos abstractos con los cuales trabaja el arquitecto y que los programadores deberán refinar y proyectar sobre la tecnología de implementación disponible. En el nivel arquitectónico (no en el del diseño) se debe hacer hincapié en componentes con interfaces complejas y/o con relaciones complejas entre ellos y, primordialmente, conectores semánticamente más ricos. En este sentido la definición 2 sigue las ideas planteadas originalmente por Shaw y Garlan [17] en cuanto a que en el nivel arquitectónico los conectores son tratados (o deberían ser tratados) como elementos de primer nivel y no como subalternos de los componentes, reducidos a llamada a procedimiento o memoria compartida. Este es, sin dudas, uno de los aportes fundamentales que separan a la arquitectura de los otros dos niveles estructurales según se los concebía tradicionalmente pues tanto en los patrones de diseño como en el diseño de componentes la única forma de conectar dos componentes es mediante llamada a procedimiento. Considerar a los conectores como elementos de primer nivel implica entender que una buena arquitectura depende tanto de las relaciones entre sus componentes como de los componentes en sí. Más aun, dado que los conectores en el nivel arquitectónico son abstracciones no directamente representables en el lenguaje de programación, suele ser muy común que estos se conviertan en componentes al ser descriptos en los niveles inferiores. En otras palabras, un conector arquitectónico puede ser tan complejo que se requiera un buen número de líneas de código para implementarlo, lo que implica que a nivel de diseño esas líneas de código deben ser asignadas a elementos que no implementan requisitos funcionales y que deben ser diseñados.

Observar que el Diseño Orientado a Objetos (DOO) no entra dentro del nivel arquitectónico pues solo utiliza llamada a procedimiento, no hay restricciones entre sus componentes (cualquier objeto puede invocar servicios de cualquier otro) y solo se utiliza un tipo de componente (los objetos). En este sentido el DOO es una forma de diseño de relativo bajo nivel de abstracción.

También podemos determinar si una decisión estructural es arquitectónica o no por medio del Criterio de Localía, el cual dice que una decisión estructural no es de diseño (y por lo tanto es arquitectónica) sí y sólo sí un sistema que verifica esa decisión puede ser *extendido* a un sistema que no la verifica [22]. Por ejemplo, estructurar un sistema siguiendo el estilo cliente/servidor es una decisión arquitectónica puesto que el sistema puede ser extendido a un sistema *peer-to-peer* que no verifica el estilo. Por otro lado, si ese mismo sistema utiliza, por ejemplo, el patrón de diseño Strategy, al extenderlo a un sistema *peer-to-peer* seguirá utilizándolo, por lo que el uso del patrón es una decisión de diseño (y no arquitectónica).

Sin embargo, no siempre es simple diferenciar claramente entre el nivel de diseño y el nivel arquitectónico. En última instancia es el ingeniero el que traza la línea que divide ambos niveles en cada proyecto. Precisamente, no disponemos aun de una notación que permita expresar siempre el nivel arquitectónico sin adentrarse en el detalle del diseño. Sin embargo, ciertas estructuras permiten describir este nivel, o son más útiles en ese nivel que en el nivel del diseño. Una buena aproximación para describir la arquitectura sin describir el diseño yace en el concepto de *estilos arquitectónicos* que desarrollamos en siguiente sección.

7. Estilos arquitectónicos

Los estilos arquitectónicos son una generalización y abstracción de los patrones de diseño.

Definición 3 (Estilo Arquitectónico)**.** *Caracteriza una familia de sistemas que están relacionados por compartir propiedades estructurales y funcionales.*

También puede definirse como la descripción de los tipos componente y de los patrones de interacción entre ellos.

Notar que, a diferencia de los patrones de diseño, la definición apunta a describir sistemas completos y no partes de sistemas. Nadie supone que podrá describir un sistema completo mediante el patrón de diseño Composite o Abstract Factory o Command, pero sí puede hacerlo mediante un estilo arquitectónico.

Otra forma de caracterizar un estilo arquitectónico es respondiendo las siguientes preguntas, entre otras:

- ¿Cuál es el vocabulario de diseño, es decir los tipos de componentes y conectores?
- ¿Cuáles son los patrones estructurales permitidos?
- ¿Cuál es el modelo computacional subyacente?
- ¿Cuáles son los invariantes esenciales?
- ¿Cuáles son algunos ejemplos comunes de su uso?
- ¿Cuáles son las ventajas y desventajas?
- ¿Cuáles son las variantes (especializaciones y deformaciones) comunes?
- ¿Cuál es la metodología de desarrollo?

Referencias

[1] B. P. Lientz and E. B. Swanson, *Software Maintenance Management.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1980.

[2] F. P. Brooks, *The mythical man-month. Essays on Software Engineering. Anniversary edition.* Addison-Wesley, 1995.

[3] J. R. McKee, "Maintenance as a function of design," in *AFIPS '84: Proceedings of the July 9-12, 1984, national computer conference and exposition.* New York, NY, USA: ACM, 1984, pp. 187–193.

[4] F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," in *Proceedings of the 4. Fachtagung der GI Programmiersprachen.* London, UK: Springer-Verlag, 1976, pp. 80–89.

[5] D. L. Parnas, "On the criteria to be used in descomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[6] ——, "Designing software for ease of extension and contraction," in *ICSE '78: Proceedings of the 3rd international conference on Software engineering.* Piscataway, NJ, USA: IEEE Press, 1978, pp. 264–277.

[7] D. L. Parnas, P. C. Clements, and D. M. Weiss, "The modular structure of complex systems," in *ICSE '84: Proceedings of the 7th international conference on Software engineering.* Piscataway, NJ, USA: IEEE Press, 1984, pp. 408–417.

[8] B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Not.*, vol. 9, no. 4, pp. 50–59, 1974.

[9] J. Guttag, "Abstract data types and the development of data structures," *SIGPLAN Not.*, vol. 11, no. SI, p. 72, 1976.

[10] B. Liskov and J. Guttag, *Abstraction and specification in program development.* Cambridge, MA, USA: MIT Press, 1986.

[11] T. DeMarco, *Structured Analysis and System Specification.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1979.

[12] G. J. Myers, *Composite Structure Design.* New York, NY, USA: John Wiley & Sons, Inc., 1978.

[13] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1979.

[14] R. Smith, "Panel on design methodology," in *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum).* New York, NY, USA: ACM, 1987, pp. 91–95.

[15] K. Beck and W. Cunningham, "Using pattern languages for object-oriented program," in *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum).* New York, NY, USA: ACM, 1987, pp. 9–16.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Patrones de diseño.* Addison Wesley, 2003.

[17] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline.* Upper Saddle River: Prentice Hall, 1996.

[18] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice.* Reading: Addison-Wesley, 1998.

[19] ——, *Software architecture in practice (second edition).* Boston: Pearson Education, 2003.

[20] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting Software Architectures: Views and Beyond.* Pearson Education, 2002.

[21] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stad, *Pattern-Oriented Software Architecture — A System of Patterns.* John Wiley Press, 1996.

[22] A. H. Eden and R. Kazman, "Architecture, design, implementation," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2003, pp. 149–159.