Introducción a la arquitectura de software

Octubre 2012

1 ¿Qué es una Arquitectura de Software?

Al igual que con diseño, hay muchas definiciones de arquitectura:

- La arquitectura es el diseño de más alto nivel.
- La arquitectura son el conjunto de decisiones que le dan estructura al resto del diseño, las más difíciles de modificar, y por tanto, las que deberían ser tomadas lo antes posible, con el mayor cuidado.
- La arquitectura se ocupa de posibilitar (o garantizar) el cumplimiento de los requerimientos no funcionales.

Muchas definiciones consideran que la arquitectura es una parte del diseño, nosotros vamos a seguir esa postura.

También podemos diferenciar:

- **Arquitectura de software** es la que se ocupa de la forma de estructurar una sistema de software.
- **Arquitectura de sistema, organizacional o de integración** es la que se ocupa de las relaciones entre sistemas de software para conformar un sistema más grande.

Algunos autores definen más categorías arquitecturales. Por el momento nosotros nos vamos a concentrar en la Arquitectura de Software y sólo ocasionalmente vamos a considerar las demás.

2 Cuestiones arquitectónicas

La arquitectura ataca varios problemas, que conocemos como cuestiones arquitectónicas (**Architectural Concerns**). Podemos reunirlas en dos grandes grupos:

2.1 Físicas

1. Hardware, despliegue, ambientes (infraestructura)
2. Distribución física de los componentes lógicos (**nodos**)
3. Tecnologías críticas (alineado con la idea de "difícil de cambiar"): lenguajes, ciertos frameworks, etc

2.2 Conceptuales/lógicas

La actividad de diseño con frecuencia consiste en dividir una pieza de software en subsistemas menores, definir las responsabilidades de cada uno de ellos y la forma en que se comunican entre sí. Cuando pensamos en el diseño de más bajo nivel, los subsistemas podrían ser clases u objetos, y la forma de comunicarse entre ellos será a través de mensajes. En un nivel medio, los subsistemas podrían ser *módulos*, cada uno con una diferente responsabilidad funcional.

La idea de *dividir y conquistar* se mantiene vigente cuando pensamos en arquitectura. Sin embargo, la división en subsistemas se puede analizar de dos maneras:

- **Según sus responsabilidades funcionales.** Por ejemplo en un sistema administrativo de la universidad podría haber un subsistema que se ocupe de las inscripciones, otro de pagar sueldos a los docentes, sistemas contables, asistencia, etc.
- **Según el tipo de responsabilidad**, separando el dominio de otras responsabilidades no funcionales como seguridad, persistencia, interfaz de usuario, transaccionalidad, distribución, logging, debugging, etc.

A cada una de esas responsabilidades que tiene un sistema de software que no pasa por "lo funcional" la denominamos ***concern*** (o *aspecto*). Un concern es una preocupación o un problema que tenemos que responder y que entrecruza los requerimientos funcionales (es *cross-aplicación*). Hay dos concerns en los que nos vamos a concentrar por encima de los otros: la **persistencia** y la **interfaz de usuario**:

- La **interfaz de usuario** es la parte del sistema que se ocupa de la interacción con el usuario, es decir, mostrarle información al usuario y permitirle tomar acciones sobre esa información, modificarla, navegar entre las diferentes vistas o actividades, etc.
- Por **persistencia** nos referimos a la parte del sistema que se ocupa de garantizar que la información del sistema perdure en el tiempo. En muchos casos, para hacer esto se necesita almacenar los datos en un dispositivo de almacenamiento permanente (como un disco rígido). Por lo tanto, parte del problema a resolver por la persistencia son los mecanismos para decidir cuándo y cómo mover los datos desde el espacio de trabajo (por ejemplo, memoria RAM) al almacenamiento permanente.

En la parte que queda de la materia vamos a trabajar sobre el diseño de esos concerns, en particular:

- Técnicas específicas para diseñar interfaces de usuario y/o persistencia
- Formas de integrar a esos dos aspectos/concerns con el dominio.

Como se ve, no es nuestro objetivo estudiar la arquitectura en profundidad, como tema. Apenas vamos a pasar por algunos conceptos, a modo introductorio. El objetivo es que al diseñar podamos tener un panorama más grande, que incluya a todo el sistema y no sólo al dominio. Ese panorama nos lo da la arquitectura.

3 Criterios de decisión arquitectónicos

Para tomar decisiones de diseño, nuestros criterios son ante todo funcionales, pero también consideramos criterios no funcionales, a los que llamamos cualidades de diseño. En los que más hincapié hacemos son **mantenibilidad**, **extensibilidad**, **simplicidad**, **robustez**.

Para tomar decisiones arquitectónicas nuestros criterios son ante todo no funcionales, y si bien los anteriores siguen siendo fundamentales, agregamos como particularmente interesantes a: **escalabilidad**, **disponibilidad**, **seguridad**, **trazabilidad**, **eficiencia**, **recuperación ante errores**.

Empiezan a tener más notoriedad limitaciones no tecnológicas (y acá hay un cierto solapamiento de arquitectura con gestión de proyectos): **costo**, **tiempo de desarrollo**, **cuestiones legales**, **conformación del equipo**.

A modo de resumen, una arquitectura incluye por ejemplo las siguientes decisiones:

- Módulos funcionales, responsabilidades y comunicaciones.
- Cómo atacar cada concern, qué herramientas utilizar, etc.
- La tecnología a utilizar, comenzando por el lenguaje, sistema operativo, frameworks.
- El hardware necesario y la forma de distribuir las responsabilidades entre los diferentes ***nodos***.

Cabe recordar que al ser la arquitectura una parte del diseño, valen las mismas aclaraciones sobre las influencias y "entradas" que marcamos al hablar de diseño. Por ejemplo para poder definir una arquitectura puede ser conveniente tener en cuenta el contexto en el cual será utilizada, el equipo de trabajo, etc.

4 Patrones

Para resolver problemas de arquitectura, no tenemos **la** verdad, sino "best practices" que pueden o no aplicar al contexto en el que estamos inmersos. Exactamente igual que en el diseño. Así surge nuevamente la idea de patrón: soluciones conocidas a problemas frecuentes.

4.1 Arquitectura física

Al construir sistemas empresariales es frecuente que estos se ejecuten en forma distribuida, es decir, con la intervención de varias computadoras o *nodos* de procesamiento. Esto puede ser necesario por varios motivos:

- Para obtener un incremento de performance al combinar la capacidad de procesamiento de las diferentes máquinas.
- Para garantizar tolerancia a fallos mediante redundancia: dos máquinas capaces de ejecutar la misma tarea, en caso que una falle la otra puede obtener su lugar.
- Por una necesidad de distribución.

Nos vamos a concentrar por ahora en esta última, porque es la que resulta más cercana a los temas que vamos a ver a continuación. En un sistema multiusuario, cada usuario necesita poder acceder a la aplicación desde su propia computadora, pero todos deben compartir el mismo repositorio centralizado de datos. Eso obliga a tener parte de la aplicación distribuida (en las máquinas de los usuarios) y parte de la aplicación centralizada (en un servidor o varios).

Según cómo repartamos el peso de la lógica entre la parte centralizada (servidor) y la parte distribuida (clientes), podemos clasificar las aplicaciones de la siguiente manera:

- Stand Alone (no distribuida)
- Cliente Pesado
- Cliente Liviano
- Mixtas, como RIA (Javascript o Flash) o actualizaciones automáticas (el más viejo es Java WebStart).

Y esto dispara varios problemas que por ahora sólo vamos a enunciar e iremos profundizando a lo largo de las próximas clases:

- **Stateless vs. stateful.** Es decir, qué conocimiento tiene el servidor de las tareas que está realizando cada uno de sus clientes.
- **Seguridad**. La distribución representa problemas de seguridad, ya que los clientes son entornos menos controlados que los servidores.
- **Actualización.** ¿Cómo mantener a los clientes actualizados?
- **Comunicación** entre clientes y servidores, traslado de datos, conversiones, validaciones, etc.
- **Duplicación:** Con frecuencia nos vemos obligados a duplicar cierta lógica, por ejemplo las validaciones (en el cliente **y** en el servidor).

4.2 Arquitectura lógica

Como es natural esperar, la arquitectura lógica intentará desacoplar entre sí a los componentes que resuelven los diferentes concerns: persistencia, interfaz de usuario y dominio. A veces a eso se lo suele llamar "división en capas"; sin embargo, la organización en capas es sólo una de las múltiples formas en las que se puede lograr ese objetivo. A las diferentes formas de encarar este problema se los denomina ***estilos arquitectónicos***.

En clase no vamos a pasar por todos los estilos, les dejamos como lectura el paper de David Garlan y Mary Shaw, que menciona estilos interesantes como

- batch sequential,
- pipes & filters,
- object oriented,
- layers,
- database/blackboard,
- comunicating processes
- microkernel,
- interpreter/virtual machine,
- rule based.

En este curso nos interesa remarcar que dividir en capas no es la única forma de garantizar el desacoplamiento entre los concerns, y creemos que en muchos casos no es la mejor. Según Garlan y Shaw, en una **arquitectura en capas**, las mismas se organizan jerárquicamente como una pila, de forma que cada una le da servicios a la capa inmediata superior y es cliente de la capa inmediata inferior. En muchos casos, las capas internas están vedadas para las exteriores, esto hace que el sistema sea muy rígido y a menudo burocrático. Además la obligación de la comunicación en un solo sentido restringe el aprovechamiento de muchos estilos interesantes de diseño, como Observers, Commands, Orden superior, entre otros.

La idea de capa, sin embargo, se presta a múltiples acepciones, que conviene no confundir:

- **Capa como "nivel de abstracción"**. El ejemplo claro es el modelo OSI de redes o el conjunto de herramientas desde que yo digo "File new" hasta que se mueven los platos de un disco rígido y se inducen campos magnéticos en él. El concepto de nivel de abstracción nos parece central a la construcción de sistemas sin embargo no es lo que pasa en un sistema de tres capas, no es cierto que la ui sea más abstracta que el dominio y éste más abstracto que la persistencia, la relación es otra. La relación de "más abstracto" en todo caso se da entre el hibernate, el jdbc, la base de datos, el driver del disco. Algo similar se puede encontrar a nivel ui: wicket, que genera html y código del browser para manejarlo, placa de video.
- **Capas "lógicas"** (layer): presentación-dominio-persistencia, aunque a veces se entienden de otras maneras.
- **Capas "físicas"** (tier): máquinas que uso.

4.3 Otras taxonomías

Si les interesa, pueden ver algunas categorizaciones de patrones arquitecturales

1. Patrones sobre problemas lógicos
    a. Modelado de presentación, y comunicación entre presentación y modelo: MVC, MVVM, etc *(que veremos próximamente)*
    b. Comunicación entre componentes del modelo distribuido: colas de mensajes (pull-push, publish-subscribe), ESBs, repositorio (base de datos, primo hermano de la memoria compartida). Los EIP de Hohpe caen en gran medida en estas últimas dos categorías http://www.eaipatterns.com/toc.html
    c. Comunicación entre componentes del modelo centralizado: los que ya vimos

de paso de mensajes, call and return, continuation, memoria compartida

2. Patrones sobre integración del diseño y arquitectura
    a. Los EAP de Fowler http://martinfowler.com/eaaCatalog/
    b. El patrón Home/DAO

5 Consideraciones generales (divagando un poco)

1. **Arquitectura evolutiva y anticipada**, diseño anticipado vs evolutivo. Si bien creemos que el gran diseño anticipado no funciona, consideramos que una noción de arquitectura muy general es necesaria desde el vamos, y cambiarla sobre la marcha es altamente costoso. Es decir, ¿podemos manejar un diseño a fuerza de refactoring? En gran parte sí. ¿Podemos manejar una arquitectura a fuerza de refactoring? En mucha menor medida. Ejemplos cotidianos:
    a. Cambiar un motor de persistencia relacional por otro no es trivial, por más abstracciones que pongas encima de éste. Ni hablar de cambiarlo por otro no relacional. No querés hacer eso.
    b. Meter una cola de mensajes a mitad del proyecto es algo factible, pero no es para tomar a la ligera.
    c. Cambiar la forma de consumir o exponer una funcionalidad vía red de REST a SOAP es algo que tampoco querés hacer.
    d. Pasar de calcular estadísticas en la base de datos a un MapReduce, ehm, esto se puede hacer, no es la muerte de nadie.
    e. Cambiar un procesamiento online a batch no te cuesta mucho. Lo contrario te podría obligar a rediseñar todo.
    f. Migrar una aplicación muy chiquita de PHP a Ruby demoró 2.5 semanas de un programador part-time. Migrarla tres meses después hubiera sido imposible.
    g. Cambiar de procesos a threads depende el lenguaje y framework podría ser simple. Cambiar de cualquiera de estos dos a IO sincrónico (poll/select) puede ser la muerte
    h. Quizás con aspectos o instrumentación de bytecode pelada podrías agregar tracing y monitoreo a una aplicación existente (en teoría), en cualquier momento del proyecto.
    i. ¿Agregar internacionalización? Si no arrancaste internacionalizando, vas a parirla.
2. Una arquitectura no es un diagrama, como un diseño no es hacer cajitas.
3. Tampoco basta con tener sólo buen criterio de diseño, es necesario tener un conocimiento profundo de una gran gama de tecnologías, incluyendo los protocolos de red empleados, los detalles de implementación de los motores, las particularidades del lenguaje, etc. Contrariamente a lo que podría pensarse, un buen "arquitecto", aunque trabaja a un mayor nivel de abstracción que el desarrollador full time (porque el arquitecto también debería estar en el código de cada día), debe tener un nivel de conocimiento de más bajo nivel de las tecnologías subyacentes. **Dicho de otra forma, el arquitecto debería ser aún más técnico que el programador.**
4. Los grandes problemas de arquitectura, al final y a alto nivel, terminan siendo los

mismos de siempre:

a. Encapsular o anular el estado (**stateful vs. stateless**) y controlar los efectos, no solo para simplificar el entendimiento del programa, sino porque los códigos puros son más fáciles de escalar y
b. Control en la **comunicación**: determinar qué componente tiene control sobre cuál otro, minimizar la complejidad computacional (O) del programa, minimizar la dependencia en la secuencia de las operaciones, manejar errores, ser expresivos para que otros puedan entenderla
c. **Duplicaciones**: Si bien es un objetivo de la arquitectura evitar duplicaciones, a veces necesitaremos introducir repeticiones:
    i. De datos: por cuestiones de eficiencia y uso efectivo del canal de comunicaciones, a veces distintos nodos deberán tener copias de la misma información, o incluso, múltiples representaciones de la misma, y mantenerlas consistentes (**actualización**)
    ii. De lógica: por los mismos motivos, y también por **seguridad**. a veces tendremos que implementar la misma lógica en distintos lenguajes (por ejemplo, SQL, Javascript y Java o Ruby) y en distintos nodos

6 Material de lectura

6.1 Obligatoria

- Decisiones de diseño para construir una aplicación
- Who needs an architect? Artículo crítico sobre el rol del arquitecto en el equipo de desarrollo de software
- Is design dead? Otro artículo de Martin Fowler que revisa el estado actual del diseño
- **Estilos arquitectónicos**: Paper de David Garlan y Mary Shaw, 1994, Carnegie Mellon University (en especial las secciones 1, 2 y 3)

6.2 Material de lectura complementaria

- Catálogo de patterns de Martin Fowler (versión web) y versión PDF.
- Catálogo de Enterprise Integration Patterns
- Beautiful Architecture, Diomidis Spinellis, Cap 1 (preview)