Planos Arquitectónicos: El Modelo de "4+1" Vistas de la Arquitectura del Software*

Philippe Kruchten

Abstract

Este artículo presenta un modelo para describir la arquitectura de sistemas de software, basándose en el uso de múltiples vistas concurrentes. Este uso de múltiples vistas permite abordar los intereses de los distintos "stakeholders" de la arquitectura por separado: usuarios finales, desarrolladores, ingenieros de sistemas, administradores de proyecto, etc., y manejar los requisitos funcionales y no funcionales separadamente. Se describe cada una de las cinco vistas descritas, conjuntamente con la notación para captarla. Las vistas se diseñan mediante un proceso centrado en la arquitectura, motivado por escenarios y desarrollado iterativamente.

1 Introducción

Todos hemos visto muchos libros y artículos donde se intenta capturar todos los detalles de la arquitectura de un sistema usando un único diagrama. Pero si miramos cuidadosamente el conjunto de cajas y flechas que muestran estos diagramas, resulta evidente que sus autores han trabajado duramente para intentar representar más de un plano que lo que realmente podría expresar la notación. ¿Es acaso que las cajas representan programas en ejecución? ¿O representan partes del código fuente? ¿O computadores físicos? ¿O acaso meras agrupaciones de funcionalidad? ¿Las flechas representan dependencias de compilación? ¿O flujo de control? Generalmente es un poco de todo.

¿Será que una arquitectura requiere un estilo único de arquitectura? A veces la arquitectura del software tiene secuelas de un diseño del sistema que fue muy lejos en particionar prematuramente el software, o de un énfasis excesivo de algunos de los aspectos del desarrollo del software: ingeniería de los datos, o eficiencia en tiempo de ejecución, o estrategias de desarrollo y organización de equipos. A menudo la arquitectura tampoco aborda los intereses de todos sus "clientes".

Varios autores han notado este problema, incluyendo a David Garlan y Mary Shaw [7], Gregory Abowd y Robert Allen [1], y Paul Clements [4].

El modelo de 4+1 vistas fue desarrollado para remediar este problema. El modelo 4+1 describe la arquitectura del software usando cinco vistas concurrentes. Tal como se muestra en la Figura 1, cada vista se refiere a un conjunto de intereses de diferentes stakeholders del sistema.

- La vista lógica describe el modelo de objetos del diseño cuando se usa un método de diseño orientado a objetos. Para diseñar una aplicación muy orientada a los datos, se puede usar un enfoque alternativo para desarrollar algún otro tipo de vista lógica, tal como diagramas de entidad-relación.
- La vista de procesos describe los aspectos de concurrencia y sincronización del diseño.
- La vista física describe el mapeo del software en el hardware y refleja los aspectos de distribución.
- La vista de desarrollo describe la organización estática del software en su ambiente de desarrollo.

Los diseñadores de software pueden organizar la descripción de sus decisiones de arquitectura en estas cuatro vistas, y luego ilustrarlas con un conjunto reducido de casos de uso o escenarios, los cuales constituyen la quinta vista. La arqutitectura evoluciona parcialmente a partir de estos escenarios.

*Artículo publicado en *IEEE Software* 12(6), Noviembre 1995. Traducido por María Cecilia Bastarrica en Marzo 2006

En Rational, aplicamos la fórmula de Dwayne Perry y Alexander Wolf [9] de manera independiente para cada vista:

Arquitectura del software = {Elementos, Formas, Motivación/Restricciones}

Para cada vista definimos un conjunto de elementos (componentes, contenedores y conectores), captamos la forma y los patrones con que trabajan, y captamos la justificación y las restricciones, relacionando la arquitectura con algunos de sus requisitos.

Cada vista se describe en lo que llamamos "diagrama" (blueprint) que usa su notación particular. Los arquitectos también pueden usar estilos de arquitectura para cada vista, y por lo tanto hacer que coexistan distintos estilos en un mismo sistema.

El modelo de 4+1 vistas es bastante genérico: se puede usar otra notación y herramientas que las aquí descritas, así como también otros métodos de diseño, especialment para las descomposiciones lógica y de procesos.

2 El Modelo de 4+1 Vistas

La arquitectura del software se trata de abstracciones, de descomposición y composición, de estilos y estética. Trambién tiene relación con el diseño y la implementación de la estructura de alto nivel del software.

Los diseñadores construyen la arquitectura usando varios elementos arquitectónicos elegidos apropiadamente. Estos elementos satisfacen la mayor parte de los requisitos de funcionalidad y performance del sistema, así como también otros requisitos no funcionales tales como confiabilidad, escalabilidad, portabilidad y disponibilidad del sistema.

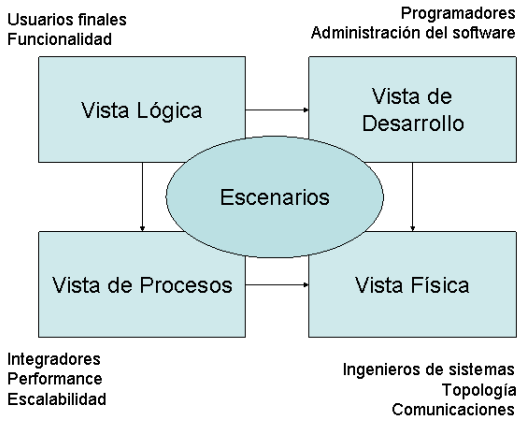

Figure 1: Modelo de "4+1" vistas

3 La Arquitectura Lógica

La arquitectura lógica apoya principalmente los requisitos funcionales –lo que el sistema debe brindar en términos de servicios a sus usuarios. El sistema se descompone en una serie de abstracciones clave, tomadas (principalmente) del dominio del problema en la forma de *objetos* o *clases de objetos*. Aquí se aplican los principios de abstracción, encapsulamiento y herencia. Esta descomposición no sólo se hace para potenciar el análisis funcional, sino también sirve para identificar mecanismos y elementos de diseño comunes a diversas partes del sistema.

Usamos el enfoque de Booch/Rational para representar la arquitectura lógica, mediante *diagramas de clases* y *templates de clases* [3]. Un diagrama de clases muestra un conjunto de clases y sus relaciones lógicas:

asociaciones, uso, composición, herencia y similares. Grupos de clases relacionadas pueden agruparse en categorías de clases. Los templates de clases se centran en cada clase individual; enfatizan las operaciones principales de la clase, e identifican las principales características del objeto. Si es necesario definir el comportamiento interno de un objeto, esto ser realiza con un diagrama de transición de estados o diagrama de estados. Los mecanismos y servicios comunes se definen como *utilities de la clase*.

Notación. La notación para la vista lógica se deriva de la notación de Booch [3]. Esta se simplifica considerablemente de tal modo de tener en cuenta solamente los items relevantes para la arquitectura. En particular, los numerosos adornos disponibles son bastante inútiles a este nivel de diseño. Usamos Rational Rose para apoyar el diseño lógico de la arquitectura.

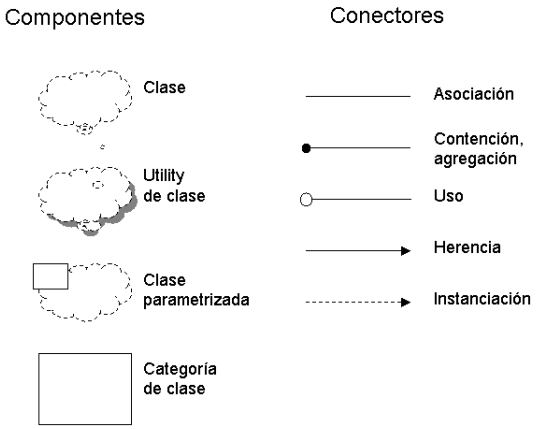

Figure 2: Notación para la vista lógica

Estilo. El estilo usado para la vista lógica es el estilo de orientación a objetos. La principal guiía para el diseño de la vista lógica es el intentar mantener un modelo único y coherente de objetos a lo largo de todo el sistema, para evitar la especialización prematura de las clases y mecanismos particulares o de un procesador.

Ejemplos. La Figura 3 muestra las principales clases que forman parte de la arquitectura de una muestra de PBX que desarrollamos en Alcatel.

Un PBX establece comunicaciones entre terminales. Un terminal puede ser un teléfono, una línea troncal (i.e. una línea a la oficina central), una línea de unión (i.e. de un PBX privado a una línea PBX), o una característica de una línea telefónica.

Diferentes tarjetas de interfaz de línea soportan distintas líneas. El objeto controlador decodifica e inyecta todas las señales en la tarjeta de interfaz de la línea, traduce las señales específicas desde y hacia un conjunto pequeño y uniforme de eventos: comenzar, deterner, dígito, etc. El controlador tiene también todas las restricciones *hard* de tiempo real. Esta clase tiene muchas subclases a las que proporciona distintos tipos de interfaces.

El objeto terminal mantiene el estado de una terminal, y negocia los servicios para esa línea. Por ejemplo, usa los servicios del *plan de numeración* para interpretar el discado.

El objeto *conversación* representa un conjunto de terminales que participan de una conversación. Usa los *servicios de traducción* (directorio, mapeo de direcciones lógicas a físicas, rutas), y *servicios de conexión* para establecer una ruta de voz entre los terminales.

Para sistemas mucho más grandes, que contienen varias docentas de clases de relevancia para la arquitectura, la Figura 3b muestra un diagrama de clases de alto nivel para un sistema de control de tráfico aéreo desarrollado por Hughes Aircraft of Canada que contiene 8 categorías de clases (i.e. grupos de clases).

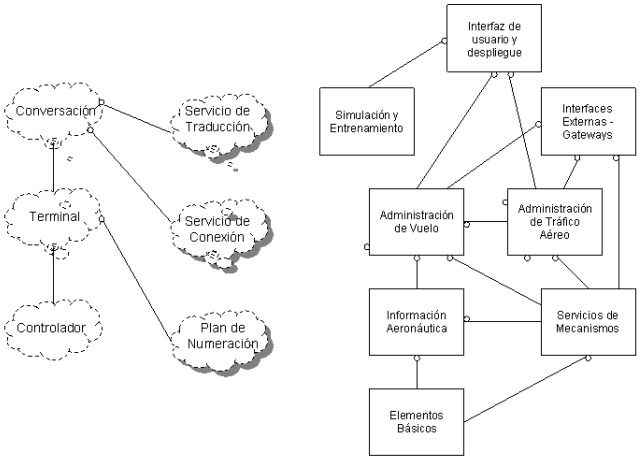

Figure 3: (a) Diagrama lógico del Télic PBX; (b) Diagrama de un sistema de control de tráfico aéreo

4 La Vista de Procesos

La arquitectura de procesos toma en cuenta algunos requisitos no funcionales tales como la performance y la disponibilidad. Se enfoca en asuntos de concurrencia y distribución, integridad del sistema, de tolerancia a fallas. La vista de procesos también especifica en cuál hilo de control se ejecuta efectivamente una operación de una clase identificada en la vista lógica.

La arquitectura de procesos se describe en varios niveles de abstracción, donde cada nivel se refiere a distintos intereses. El nivel más alto la arquitectura de procesos puede verse como un conjunto de redes lógicas de programas comunicantes (llamados "procesos") ejecutándose en forma independiente, y distribuidos a lo largo de un conjunto de recursos de hardware conectados mediante un bus, una LAN o WAN. Múltiples redes lógicas pueden usarse para apoyar la separación de la operación del sistema en línea del sistema fuera de ínea, así como también para apoyar la coexistencia de versiones de software de simulación o de prueba.

Un *proceso* es una agrupación de tareas que forman una unidad ejecutable. Los procesos representan el nivel al que la arquitectura de procesos puede ser controlada tácticamente (i.e., comenzar, recuperar, reconfigurar, y detener). Además, los procesos pueden replicarse para aumentar la distribución de la carga de procesamiento, o para mejorar la disponibilidad.

Partición. El software se particiona en un conjunto de *tareas* independientes: hilo de control separado que puede planificarse para su ejecución independiente en un nodo de procesamiento.

Podemos entonces distinguir:

- *tareas mayores* son elementos arquitectónicos que pueden ser manejados en forma unívoca. Se comunican a través de un conjunto bien definido de mecanismos de comunicación inter-tarea: servicios de comunicación sincrónicos y asincrónicos basados en mensajes, llamados a procedimientos remotos, difusión de eventos, etc. Las tareas mayores no debieran hacer suposiciones acerca de su localización con otras tareas dentro de un mismo proceso o un mismo nodo de procesamiento.

- *tareas menores* son tareas adicionales introducidas localmente por motivos de implementación tales como actividades cíclicas, almacenamiento en un buffer, time-out, etc.). Pueden implementarse en Ada por ejemplo, o como hilos de control liviano (threads). Pueden comunicarse mediante rendezvous o memoria compartida.

El flujo de mensajes y la carga de procesos puede estimarse en base al diagrama de procesos. También es posible implmentar una vista de procesos "vacía", con cargas dummy para los procesos y medir entonces su performance en el sistema objetivo [5].

Notación. La notación que usamos para la vista de procesos se expande de la notación originalmente propues por Booch para las tareas de Ada yse centra solamente en los elementos arquitectónicamente relevantes (Figura 4).

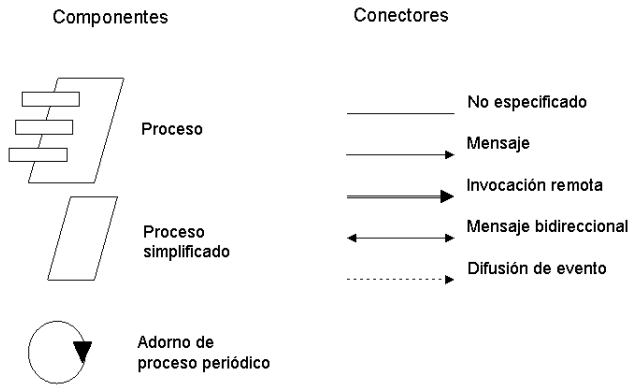

Figure 4: Notación para el diagrama de procesos

Hemos usado el producto Universal Network Architecture Services (UNAS) de TRW para diseñar e implementar un conjunto de procesos y tareas (con sus respectivas redundancias) como redes de procesos. UNAS contiene una herramienta –el Software Architects Lifecycle Environment (SALE)– el cual apoya dicha notación. SALE permite describir gráficamente la arquitectura de procesos, incluyendo la especificación de las posibles rutas de comunicación inter-tareas del cual se puede generar automáticamente el correspondiente código fuente Ada o C++. La generación automática de código permite hacer cambios fácilmente a la vista de procesos.

Estilo. Varios estilos podrían servir para la vista de procesos. Por ejemplo, tomando la taxonomía de Garlan y Shaw [7] tenemos: tubos y filtros, o cliente/servidor, con variantes de varios clientes y un único servidor o múltiples clientes y múltiples servidores. Para sistemas más complejos, podemos usar un estilo similar a la forma de agrupación de procesos del sistema ISIS descrito por Kenneth Birman con otra notación y otras herramientas [2].

Ejemplo. La Figura 5 muestra una vista de procesos parcial para el sistema PBX. Todas las terminales son adminsitradas por un único *proceso terminal*, el cual es manejado a través de mensajes en sus colas de input. Los objetos controladores se ejecutan en alguna de las tres tareas que componen el proceso controlador: una *tarea cíclica de baja tasa* que chequea todas las terminales inactivas (200ms), pone toda terminal que se torna activa en la lista de búsqueda del la *tarea cíclica de alta tasa* (10ms), la cual detecta cualquier cambio de estado significativo, y lo pasa a la *tarea controladora principal* la cual interpreta el cambio y lo comunica mediante un mensaje con el terminal correspondiente. Aquí el mensaje pasa dentro del controlador a través de memoria compartida.

5 Vista de Desarrollo

La vista de desarrollo se centra en la organización real de los módulos de software en el ambiente de desarrollo del software. El software se empaqueta en partes pequeñas –bibliotecas de programas o subsistemas– que pueden ser desarrollados por uno o un grupo pequeño de desarrolladores. Los subsistemas se organizan en una jerarquía de *capas*, cada una de las cuales brinda una interfaz estrecha y bien definida hacia las capas superiores.

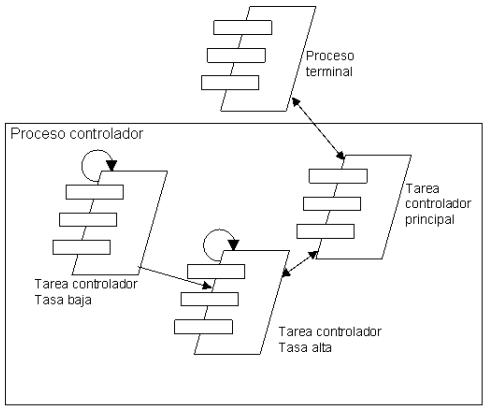

Figure 5: Diagrama (parcial) de procesos para Télic PBX

La vista de desarrolla tiene en cuenta los requisitos internos relativos a la facilidad de desarrollo, administración del software, reutilización y elementos comunes, y restricciones impuestas por las herramientas o el lenguaje de programación que se use. La vista de desarrollo apoya la asignación de requisitos y trabajo al equipo de desarrollo, y apoya la evaluación de costos, la planificación, el monitoreo de progreso del proyecto, y también como base para analizar reuso, portabilidad y seguridad. Es la base para establecer una línea de productos.

La vista de desarrollo de un sistema se representa en diagramas de módulos o subsistemas que muestran las relaciones exporta e importa. La arquitectura de desarrollo completa sólo puede describirse completamente cuando todos los elementos del softare han sido identificados. Sin embargo, es posible listar las reglas que rigen la arquitectura de desarrollo – partición, agrupamiento, visibilidad– antes de conocer todos los elementos.

Notación. Tal como se muestra en la Figura 6, usamos una variante de la notación de Booch limitándonos a aquellos items relevantes para la arquitectura.

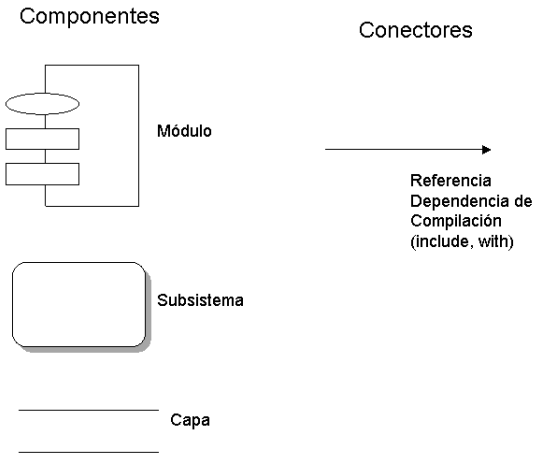

Figure 6: Notación para el diagrama de desarrollo

El ambiente de desarrollo Apex de Rational apoya la definición e implementación de la arquitectura de

desarrollo, la estrategia de capas antes descrita, y el cumplimiento de las reglas de diseño. Se puede dibujar la arquitectura de desarrollo en Rational Rose a nivel de módulos y subsistemas, en ingeniería hacia adelante y reversa a partir de código fuente Ada y C++.

Estilo para la vista de desarrollo. Recomendamos adptar el *estilo de capas* para la vista de desarrollo, definido en 4 a 6 niveles de subsistemas. Cada capa tiene una responsabilidad bien definida. La regla de diseño es que un subsistema en una cierta capa sólo puede depender de subsistemas que estén o bien en la misma capa o en capas inferiores, de modo de minimizar el desarrollo de complejas redes de dependencias entre módulos y permitir estrategias de desarrollo capa por capa.

Ejemplo de Arquitectura de Desarrollo. La Figura 7 representa la organización del desarrollo en cinco capas de la línea de productos de sistemas de control de tráfico aéreo desarrollados por Hughes Aircraft de Canadá [8]. Esta es la arquitectura de desarrollo correspondiente a la arquitectura lógica que se muestra en la Figura 3b.

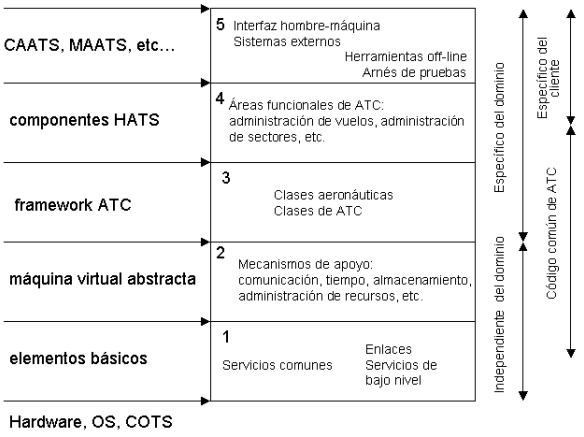

Figure 7: Las 5 capas del Sistema de Tráfico Aéreo de Hughes (HATS)

Las capas 1 y 2 constituyen la infraestructura distribuida independiente del dominio que es común a toda la línea de productos y la independiza de las variaciones de la plataforma de hardware, sistema operativo, o productos comerciales tales como administradores de bases de datos. La capa 3 agrega a esta infraestrucura un framework ATC para forma una *arquitectura de software dependiente del dominio*. Usando este framework, en la capa 4 se construye una paleta de funcionalidad. La capa 5 es dependiente del cliente y del producto, y contiene la mayor parte de las interfaces con el usuario y con sistemas externos. Tantos como 72 subsistemas forman parte de la capa 5, cada uno de los cuales contiene entre 10 y 50 módulos, y puede representarse en diagramas adicionales.

6 Arquitectura Física

Mapeando el software al hardware

La arquitectura física toma en cuenta primeramente los requisitos no funcionales del sistema tales como la disponibilidad, confiabilidad (tolerancia a fallas), performance (throughput), y escalabilidad. El software ejecuta sobre una red de computadores o nodos de procesamiento (o tan solo nodos). Los variados elementos identificados –redes, procesos, tareas y objetos– requieren ser mapeados sobre los variados nodos. Esperamos que diferentes configuraciones puedan usarse: algunas para desarrollo y pruebas, otras para emplazar el sistema en varios sitios para distintos usuarios. Por lo tanto, el mapeo del software en los nodos requiere ser altamente flexible y tener un impacto mínimo sobre el código fuente en sí.

Notación para la arquitectura física. Los diagramas físicos pueden tornarse muy confusos en grandes sistemas, y por lo tanto toman diversas formas, con o sin el mapeo de la vista de procesos.

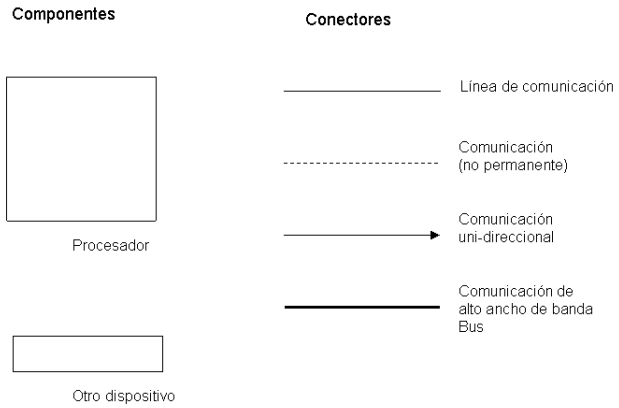

Figure 8: Notación para el diagrama físico

UNAS de TRW nos brinda los medios de datos para mapear la arquitectura de procesos en la arquitectura física permitiendo realizar una gran cantidad de clases de cambios en el mapeo sin modificar el código fuente.

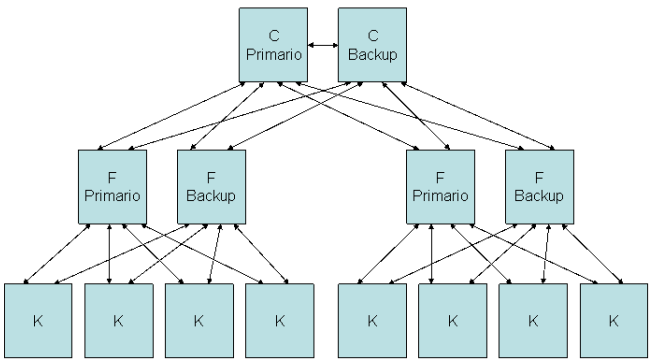

Figure 9: Diagrama físico de PABX

Ejemplo de diagrama físico. La Figura 9 muestra una configuración de hardware posible para un gran PABX, mientras que las Figuras 10 y 11 muestran el mapero de la arquitectura de procesos en dos arquitecturas físicas diferentes, que corresponden a un PABX pequeño y uno grande, respectivamente. C, F y K son tres tipos de computadores de diferente capacidad que soportan tres tipos diferentes de ejecutables.

7 Escenarios

Todas las partes juntas

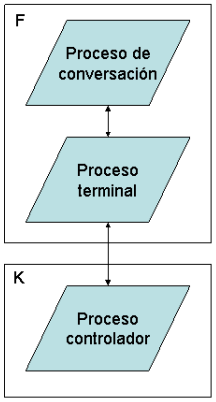

Figure 10: Una pequeña arquitectura física de PABX con emplazamiento de procesos

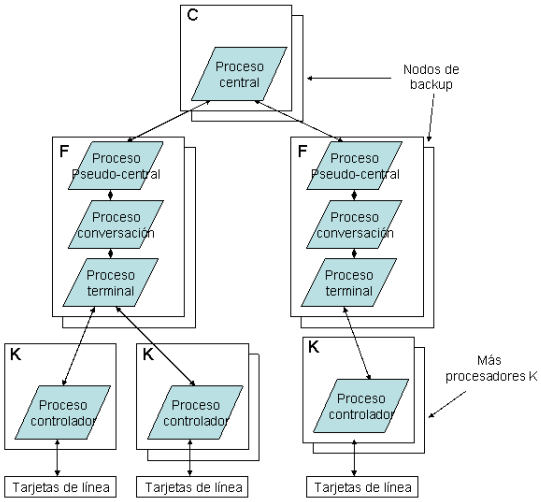

Figure 11: Diagrama físico para un PABX más grande incluyendo emplazamiento de procesos

Los elementos de las cuatro vistas trabajan conjuntamente en forma natural mediante el uso de un conjunto pequeño de *escenarios* relevantes –instancias de casos de uso más generales– para los cuales describimos sus *scripts* correspondientes (secuencias de interacciones entre objetos y entre procesos) tal como lo describen Rubin y Goldberg [10]. Los escenarios son de alguna manera una abstracción de los requisitos más importantes. Su diseño se expresa mediante el uso de diagramas de escenarios y diagramas de interacción de objetos [3].

Esta vista es redundante con las otras (y por lo tanto "+1"), pero sirve a dos propósitos principales:

- como una guía para descubrir elementos arquitectónicos durante el diseño de arquitectura tal como lo describiremos más adelante
- como un rol de validación e ilustración después de completar el diseño de arquitectura, en el papel y como punto de partido de las pruebas de un prototipo de la arquitectura.

Notación para escenarios. La notación es muy similar a la vista lógica para los componentes(ver Figura 2), pero usa los conectores de la vista de procesos para la interacción entre objetos (ver Figura 4). Nótese que las instancias de objetos se denotan con líneas sólidas. Para el diagrama lógico, capturamos y administramos los diagramas de escenarios de objetos usando Rational Rose.

Ejemplo de escenario. La Figura 12 muestra un fragmento del PABX pequeño. El script correspondiente podría ser:

1. el controlador del teléfono de Joe detecta y valida la transición desde colgado a descolgado y envía un mensaje para despertar la objeto terminal correspondiente.
2. el terminal reserva recursos y le indica al controlador que emita cierto tono de discado.
3. el controlador recibe los dígitos y los transmite hacia el terminal.
4. el terminal usa el plan de numeración para analizar el flujo de dígitos.
5. cuando se ingresa una secuencia válida de dígitos, el terminal abre una conversación.

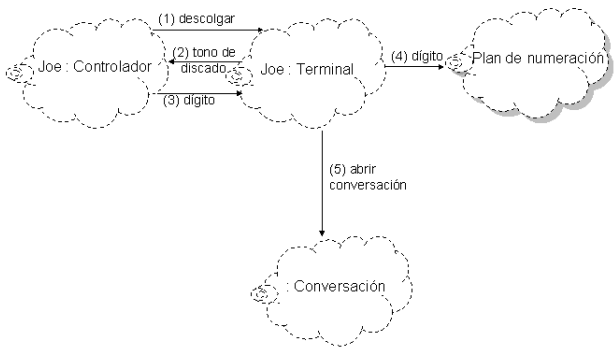

Figure 12: Embrión de un escenario de una llamada local–fase de selección

8 Correspondencia entre las Vistas

Las distintas vistas no son completamente ortogonales o independientes. Los elementos de una vista están conectados a los elementos de las otras vistas siguiendo ciertas reglas y heurísticas de diseño.

De la vista lógica a la vista de procesos. Identificamos varias características importantes de las clases de la arquitectura lógica:

- Autonomía: ¿Los objetos son activos, pasivos o protegidos?
  - un objeto *activo* toma la iniciativa de invocar las operaciones de otros objetos o sus propias operaciones, y tiene el control completo sobre la invocación de sus operaciones por parte de otros objetos.
  - un objeto *pasivo* nunca invoca espontáneamente ninguna operación y no tiene ningún control sobre la invocación de sus operaciones por parte de otros objetos.
  - un objeto *protegido* nunca invoca espontáneamente ninguna operacioón pero ejecuta cierto arbitraje sobre la invocación de sus operaciones.
- Persistencia: ¿Los objetos son permanentes o temporales? ¿Qué hacen ante la falla de un proceso o un procesador?
- Subordinación: ¿La existencia o persistencia de un objeto depende de otro objeto?
- Distribución: ¿Están el estado y las operaciones de un objeto accesibles desde varios nodos de la arquitectura física, ydesde varios procesos de la arquitectura de procesos?

En la vista lógica de la arquitectura consideramos que cada objeto es activo y potencialmente "concurrente", i.e. teniendo comportamiento en paralelo con otros objetos, y no prestamos más atención al grado preciso de concurrencia que requerimos para alcanzar este efecto. Por lo tanto, la arquitectura lógica tiene en cuenta sólo el aspecto funcional de los requisitos.

Sin embargo, cuanto definimos la arquitectura de procesos, implementar cada objeto con su propio thread de control (e.g., su propio proceso Unix o tarea Ada) no es muy práctico en el estado actual de la tecnología debido al gran overhead que esto impone. Más aún, si los objetos son concurrentes, deberá haber alguna forma de arbitraje para invocar sus operaciones.

Por otra parte, ser requiere múltiples threads de control por varias razones:

- para reaccionar rápidamente a ciertas clases de estḿulos externos, incluyendo eventos relativos al tiempo
- para sacar partido de las múltiples CPUs en un nodo, o los múltiples nodos en un sistema operativo
- para aumentar la utilización de la CPU, asignando la CPUa otras actividades mientras algún thread de control está suspendido esperando que otra actividad finalice (e.g., acceso a cierto dispositivo externo, o acceso a otro objeto activo)
- para priorizar actividades (y potencialmente mejorar la respuesta)
- para apoyar la escalabilidad del sistema (con procesos adicionales que compartan la carga)
- para separar intereses entre las diferentes áreas del software
- para alcanzar una mayor disponibilidad del sistema (con procesos de backup)

Usamos dos estrategias concurrentemente para determinar la cantidad correcta de concurrencia y definir el conjunto de procesos que se necesitan. Considerando el conjunto de posibles arquitecturas físicas, podemos proceder o bien:

Inside-out Comenzando a partir de la arquitectura lógica: definir las tareas agentes que multiplexan un único thread de control entre múltiples objetos activos de una clase; los objetos cuya persistencia o vida está subordinada a un objeto activo también se ejecutan en ese mismo agente; muchas clases que requieren ser ejecutadas con mutua exclusión, o que requieren sólo un pequeño procesamiento comparten el mismo agente. Este clustering prosigue hasta que se reducen los procesos hasta un número razonablemente pequeño que aún permite distribución y uso de los recursos físicos.

Outside-in Comenzando con la arquitectura física: identificar los estímulos externos (requerimientos) al sistema, definir los procesos cliente para manejar los estímulos y procesos servidores que sólo brindan servicios y que no los inician; usar la integridad de los datos y las restricciones de serialización del problema para definir el conjunto correcto de servidores, y asignar objetos a los agentes cliente y servidor; identificar cuáles objetos deben ser distribuidos.

El resultado es el mapeo de las clases (y sus objetos) en un conjunto de tareas y procesos de la arquitectura de procesos. Típicamente existe una tarea *agente* para una clase activa con algunas variaciones: varios agentes para una clase dada para aumentar el throughput, o varias clases mapeadas en un mismo agente porque sus opraciones se no se invocan frecuentemente o para garantizar su ejecución secuencial.

Nótese que esto no es un proceso lineal y determinístico que nos lleva a una arquitectura de procesos óptima; requiere una serie de iteraciones para lograr un *compromiso* aceptable. Hay numerosas otras formas de hacerlo, tal como lo establecen por ejemplo Birman et al. [2] o Witt et al. [11]. El método preciso a usar en la contrucción del mapeo está fuera del alcance de este artículo, pero podemos ilustrarlo con un pequeño ejemplo.

La Figura 13 muestra cómo un pequeño conjunto de clases de un sistema de control de tráfico aéreo hipotético puede mapearse en procesos.

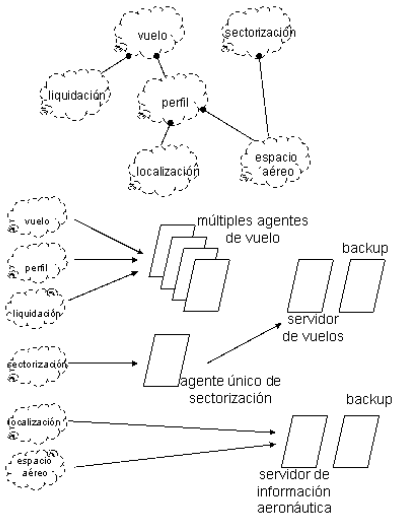

Figure 13: Mapeo de la vista lógica a la vista de procesos

La clase *vuelo* se mapea a un conjunto de *agentes de vuelo*: existen muchos vuelos a procesar, una alta tasa de estímulos externos, el tiempo de respuesta es crítico, la carga debe distribuirse entre múltiples CPUs. Más aún, los aspectos de persistencia y distribuión del procesamiento aéreo se difieren a un *servidor de vuelos*, el cual está duplicado por motivos de disponibilidad.

Un *perfil* de vuelo o una *liquidación* siempre están subordinadas a un vuelo, y a pesar que son clases complejas, ellas comparten los mismos procesos que la clase vuelo. Los vuelos se distribuyen en varios procesadores, de forma notable para el despliegue y las interfaces externas.

Una clase *sectorización*, que establece una partición del espacio aéreo para la asignación de jurisdicción de controladores de vuelos, debido a sus restricciones de integridad puede ser manejada solamente por un agente único, pero puede compartir el proceso servidor con el vuelo: las modificaciones son infrecuentes.

Localización y *espacio aéreo* y otra información aeronáutica estática son objetos protegidos, compartidos entre muchas otras clases, y raramente modificados; se mapean en su propio servidor, y se distribuye a otros procesos.

De la lógica al desarrollo. Una clase se implementa generalmente como un módulo, por ejemplo un tipo de la parte visible de un *paquete* Ada. Las clases grandes se descomponen en múltiples paquetes. Colecciones de

clases íntimamente relacionadas –categorías de clases– se agrupan en subsistemas. Deben también considerarse otras restricciones para la definición de subsistemas tales como la organización del equipo de desarrollo, el tamaño esperado del código (típicamente 5K a 20K SLOC por subsistema), grado de reuso y comonalidad esperado, principio de distribución en capas (visibilidad), políticas de liberación, y administración de la configuración. Por lo tanto, generalmente terminamos con una vista que no tiene necesariamente una relación uno a uno con la vista lógica.

Las vistas lógica y de desarrollo son muy cercanas, aunque se refieren a distintos asuntos. Hemos encontrado que cuanto mayor es el proyecto, mayor es también la distancia entre estas dos vistas. Similarmente para las vistas de procesos y física: cuanto mayor el proyecto, mayor es la distancia entre estas vistas. Por ejemplo, si comparamos las figuras 3b y 7, no existe una correspondencia uno a uno de las categorías de clases y las capas. Si tomamos la categoría "Interfaces externas–Gateway", su implementación se distribuye a lo largo de varias capas: los protocolos de comunicación están en los subsistemas dentro o debajo de la capa 1, los mecanismos generales de gateways están en los subsistemas de la capa 2, y los gateways específicos reales están en los subsistemas de la capa 5.

De procesos a físico. Los procesos y grupos de procesos se mapean sobre el hardware físico disponible en varias configuraciones para testing o distribución. Birman describe algunos esquemas eleaborados para realizar este mapeo dentro del proyecto Isis [2].

Los escenarios se relacionan esencialmente con la vista lógica, en términos de cuáles clases se usan y con la vista de procesos cuando las interacciones entre objetos involucran más de un thread de control.

9 Confeccionando el Modelo

No toda arquitectura de software requiere las "4+1" vistas completas. Las vistas que no son útiles pueden omitirse de la descripción de arquitectura, tales como la vista física si hay un único procesador, y la vista de procesos si existe un solo proceso o programa. Para sistemas muy pequeños, es posible que las vistas lógica y de desarrollo sean tan similares que no requieran descripciones independientes. Los escenarios son útiles en todas las circunstancias.

9.1 Proceso Iterativo

Witt et al. indican 4 fases para el diseño de arquitectura: bosquejo, organización, especificación y optimización, subdivididos en 12 pasos [11]. Indican que puede ser necesario algún tipo de backtrack. Creemos que este enfoque es muy "lineal" para proyectos ambiciosos y novedosos. Al final de las cuatro fases se tiene muy poco conocimiento para validar la arquitectura. Abogamos por un desarrollo más iterativo, donde la arquitectura se prototipa, se prueba, se mide, se analiza y se refina en sucesivas iteraciones. Además de permitir mitigar los riesgos asociados a la arquitectura, este desarrollo tiene otros beneficios asociados para el proyecto: construcción en equipo, entrenamiento, familiarización con la arquitectura, adquisiciónn de herramientas, ejecución de procedimientos y herramientas, etc. (Hablamos de un prototipo evolutivo, que crece lentamente hasta convertirse en el sistema, y no de un prototipo desechable, exploratorio.) Este enfoque iterativo también permite refinar los requisitos, madurarlos y comprenderlos más profundamente.

Un enfoque dirigido por escenarios La funcionalidad más crítica del sistema se captura en forma de escenarios (o casos de uso). Críticos se refiere a: funciones que son las más importantes, la razón de existir del sistema, o que tienen la mayor frecuencia de uso, o que presentan cierto riesgo técnico que debe ser mitigado.

Comienzo:

- Se elige un pequeño número de escenarios para cierta iteración basado en el riesgo y la criticidad. Los escenarios pueden sintetizarse para abstraer una serie de requisitos de usuario.
- Se bosqueja una arquitectura. Los escenarios se describen para identificar las abstracciones mayores (clases, mecanismos, procesos, subsistemas) como lo indican Rubin y Goldberg [10] –descomponiéndolos en secuencias de pares (objeto, operación).

- Los elementos de la arquitectura descubieros se ponen en las 4 vistas de arquitectura: lógica, de procesos, de desarrollo y física.
- Se implementa la arquitectura, se prueba, se mide, y se analiza para detectar errores o potenciales mejoras.
- Se recogen las lecciones aprendidas.

Loop:

La siguiente iteración puede entonces comenzar mediante:

- reestudiando los riesgos,
- extendiendo la paleta de escenarios a considerar,
- seleccionando una serie de escenarios que permitirán mitigar el riesgo o cubrir una mayor parte de la arquitectura.

Entonces:

- Intentar describir los escenarios de la arquitectura preliminar,
- descubrir elementos de arquitectura adicionales, o algunos cambios que es necesario aplicar a la arquitectura para dar cabida a estos escenarios,
- actualizar las 4 vistas de arquitectura,
- revisar los escenarios existentes basándose en los cambios,
- actualizar la implementación (el prototipo de la arquitectura) para dar apoyo al nuevo conjunto extendido de escenarios,
- probar y medir bajo sobrecarga en lo posible en el ambiente de ejecución objetivo,
- las 5 vistas se revisan para detectar potenciales simplificaciones, reutilización, y comonalidades,
- actualizar las guías de diseño y justificación del mismo,
- recoger las lecciones aprendidas.

End loop

El prototipo inicial de la arquitectura evoluciona hasta convertirse en el sistema real. Con suerte, luego de 2 o 3 iteraciones, la arquitectura se vuelve estable: no se encuentran nuevas abstracciones mayores, ni subsistemas, ni procesos, ni interfaces. El resto de la historia está dentro de la tónica del diseño, donde de hecho, el desarrollo puede continuar usando métodos y procesos muy similares.

La duración de estas iteraciones varía considerablemente: con el *tamaño* del proyecto, con el *número de personas* involucradas y su familiaridad con el dominio y el método, y con el *grado de novedad* del sistema con respecto a la organización de desarrollo. Por lo tanto la duración de una iteración puede ser de 2 a 3 semanas para un pequeño proyecto (e.g. 10KSLOC), o entre 6 y 9 meses para un gran sistema de comando y control (e.g. 700KSLOC).

10 Documentación de la Arquitectura

La documentación producida durante el diseño de la arquitectura se captura en dos documentos:

- un *Documento de Arquitectura del Software*, cuya organización sigue las "4+1" vistas (ver la figura 14 por un punteo típico)
- un documento de *Guías del Diseño del Software*, que captura (entre otras cosas) las decisiones de diseño más importantes que deben respetarse para mantener la integridad de la arquitectura del sistema.

Pagina de titulo
Historia de cambios
Tabla de contenidos
Lista de figuras
1. Alcance
2. Referencias
3. Arquitectura del software
4. Objetivos y restricciones de la arquitectura
5. Arquitectura logica
6. Arquitectura de procesos
7. Arquitectura de desarrollo
8. Arquitectura fisica
9. Escenarios
10. Tamaño y performance
11. Cualidades
Apendices
A. Siglas y abreviaturas
B. Definiciones
C. Principios de diseño

Figure 14: Punteo de un documento de Arquitectura de Software

11 Conclusión

El modelo de "4+1" vistas ha sido usado conéxito en varios proyectos grandes con o sin ajustes locales en su terminología [3]. Realmente permitió a los distintos stakeholders encontrar lo que querían acerca de la arquitectura del software. Los ingenieros de sistemas se enfocaron en la vista física, y luego en la vista de procesos. Los usuarios finales, los clientes, y los especialistas en datos en la vista lógica. Los administradores de proyectos, las personas de configuración del software en la vista de desarrollo.

Se han propuesto y discutido otra serie de vistas, tanto dentro de Rational como en otras partes, como por ejemplo Meszaros (BNR), Hofmeister, Nord y Soni (Siemens), Emery y Hilliard (Mitre) [6], pero en general hemos visto que estas otras vistas propuestas pueden reducirse a una de las cuatro vistas aquí propuestas. Por ejemplo una vista de costo&planificación puede verse como una vista de desarrollo, una vista de datos puede verse como una vista lógica, una vista de ejecución puede ser una combinación de las vistas física y de procesos.

Vista	*Lógica*	*Proceso*	*Desarrollo*	*Física*	*Escenarios*
Componentes	Clase	Tarea	Módulo, subsistema	Nodo	Paso, script
Conectores	asociación, herencia, contención	rendez-vous, mensaje, broadcast, RPC, etc.	dependencia de compilación, sentencia "with", "include"	medio de comunicación, LAN, WAN, bus	
Contenedores	Categoría de clase	Proceso	Subsistema (biblioteca)	Subsistema físico	Web
Stakeholders	Usuario final	Diseñador, integrador	Desarrollador, administrador	Diseñador de sistema	Usuario, desarrollador
Intereses	Funcionalidad	Performance, disponibilidad, tolerancia a fallas, integridad	Organización, reuso, portabilidad, líneas de productos	Escalabilidad, performance, disponibilidad	Comprensibilidad
Herramientas	Rose	UNAS/SALE DADS	Apex, SoDA	UNAS, Openview, DADS	Rose

Table 1: Resumen del modelo de "4+1" vistas

Agradecimientos

El modelo de "4+1" vistas debe su existencia a varios colegas de Rational, de Hughes Aircraft de Canadá, de Alcatel, y de otras partes. En particular quisiera agradecer por sus contribuciones a Ch. Thompson, A. Bell, M. Devlin, G. Booch, W. Royce, J. Marasco, R. Reitman, V. Ohnjec, y E. Schonberg.

References

[1] Gregory D. Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 9–20, Los Angeles, California, USA, 1993.

[2] Kenneth P. Birman and Robbert Van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. Wiley-IEEE Computer Society Press, April 1994.

[3] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings Pub. Co., Redwood City, California, 2nd edition, 1993.

[4] Paul Clements. From Domain Model to Architectures. In A. Abd-Allah et al., editor, *Focused Workshop on Software Architecture*, pages 404–420, 1994.

[5] A. R. Filarey, W. E. Royce, R. Rao, P. Schmutz, and L. Doan-Minh. Software First: Applying Ada Megaprogramming Technology to Target Platform Selection Trades. In *TRI-Ada*, pages 90–101, 1993.

[6] David Garlan. Proceedings of the first internal workshop on architectures for software systems. Technical Report CMU-CS-TR-95-151, Carnegie Mellon University, Pittsburgh, 1995.

[7] David Garlan and Mary Shaw. An Introduction to Software Architecture. *Advances in Software Engineering and Knowledge Engineering*, 1, 1993. World Scientific Publishing Co.

[8] Phillipe Kruchten and Ch. Thompson. An object-oriented, distributed architecture for large scale ada systems. In *Proceedings of the TRI-Ada'94 Conference*, pages 262–271, Baltimore, November 1994. ACM.

[9] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992. ACM Press.

[10] Kenneth S. Rubin and Adele Goldberg. Object behavior analysis. *Communications of the ACM*, 35(9):48–62, 1992.

[11] Bernard I. Witt, Terry Baker, and Everett W. Merrit. *Software Architecture and Design–Principles, Models, and Methods*. Van Nostrand Reinhold, New York, 1994.