Tema 2. Arquitecturas software

Desarrollo de Software
Curso 2020-2021
3º Grado Ingeniería Informática

Dto. Lenguajes y Sistemas Informáticos
ETSIIT
Universidad de Granada

12 de marzo de 2021

Tema 2. Arquitecturas software

Contenidos

Tema 2. Arquitecturas software

En este capítulo abordaremos la parte del desarrollo del software que se refiere a la arquitectura. En la sección 2.1 haremos un repaso del concepto de arquitectura del software y su evolución histórica. En la sección 2.2 definiremos el concepto de estilo arquitectónico y presentaremos algunos estilos de uso más común. La sección 2.3 la dedicaremos a presentar modelos actuales de especificación de arquitecturas. Por último (sección 2.4), se incluye una sección conclusiva a modo de reflexión sobre el papel del Ingeniero Informático y su figura de arquitecto software en el proceso de desarrollo software. Para las dos primeras secciones, salvo otras referencias, nos basaremos en el libro "Software architecture", de Mary Shaw and David Garlan (Shaw and Garlan, 1996). Para tercera sección, después de exponer el estándar IEEEP1471, siguiendo sobre todo el trabajo de Richvid Hilliard "Using the UML for Architectural Description" (Hilliard, 1999), utilizaremos la propuesta de Nick Rozansky (Rozanski, 2011) en su libro "Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition" como concreción del estándar de IEEE. También se usará alguna otra bibliografía adicional que se indicará en cada lugar, como el libro "Documenting Software Architectures: Views and Beyond, Second Edition" de Judith Stafford *et al.* (Stafford et al., 2010).

2.1. Introducción

La arquitectura software es el nivel más elevado en el diseño software, que se refiere a la estructura general del sistema: los componentes que contiene y la relación entre ellos mediante conectores, así como los patrones que guían estas relaciones y restricciones de los mismos.

La arquitectura incluye también las estructuras globales de control; los protocolos para la comunicación, sincronización y acceso a los datos; la forma de distribuir responsabilidades a los distintos elementos de diseño; la distribución física; el escalado y el rendimiento; la evolución, etc.

Un componente es una unidad que proporciona alguna funcionalidad claramente distinguible de otras, o un almacén de datos. Algunos ejemplos de componentes son clientes, servidores, bases de datos, filtros, capas, tipos abstractos de datos, etc. Se pueden considerar

como subsistemas si son simples o como sistemas completos si son compuestos (es decir, el componente tiene a su vez otros componentes que forman parte de él).

Un conector es el elemento arquitectónico que modela y controla la interacción entre los componentes.

Algunas interacciones son muy comunes y sencillas, como las llamadas a procedimientos y el acceso a variables compartidas, mientras que otras son más complejas, tales como los protocolos ciente-servidor, protocolos de acceso a las bases de datos, multidifusión asíncroma de eventos y flujo de datos por tubería (pipe data stream).

Además muestra la relación de la estructura con los requisitos del sistema al nivel más abstracto, el nivel arquitectónico, incluyendo requisitos no funcionales de calidad como capacidad, rendimiento, consistencia.

El que los componentes puedan ser considerados subsistemas, o incluso sistemas, independientes, permite que puedan ser reutilizados, formando parte de otros sistemas.

2.2. Estilos arquitectónicos

Un estilo arquitectónico (o patrón arquitectónico) define tipos de componentes y conectores y un conjunto de restricciones sobre la forma en la que pueden combinarse estos elementos.

Algunos de ellos se han intentado clasificar, como puede apreciarse en la Figura 2.1. Esta clasificación puede ayudar a entender os estilos y relacionar unos con otros.

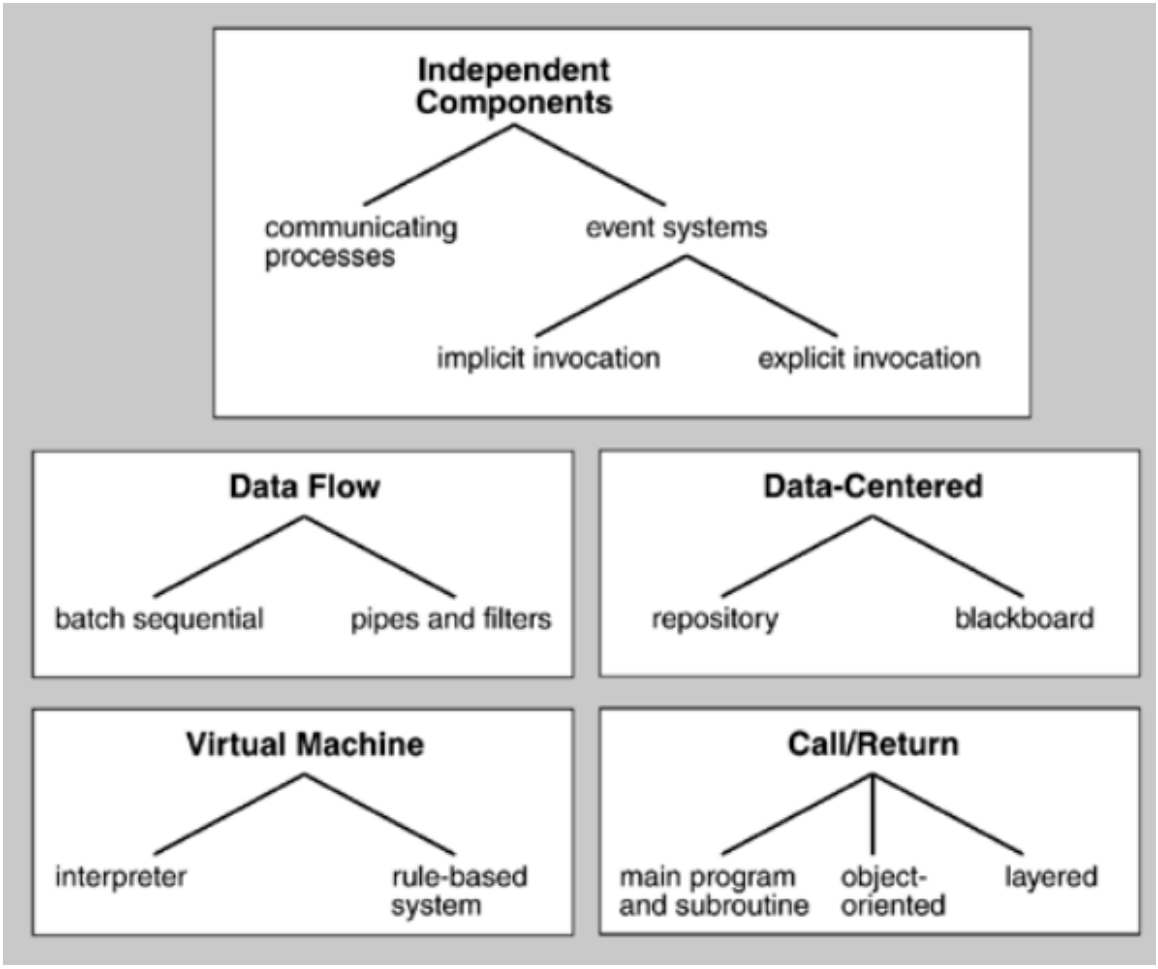


Figura 2.1: Diagrama de componentes usando el estilo arquitectónico "Control de procesos" para el sistema de control de la velocidad de crucero. [Fuente: (Bass et al., 2003)]

2.2.1. Estilo *Tubería y filtro*

En este estilo los componentes son los filtros (filter) y los conectores son las tuberías (pipes). Las tuberías transportan la corriente de datos (data stream). El filtro recibe una corriente de datos de entrada y los va procesando de forma incremental, realizando con ellos una transformación y empezando a poner los datos ya transformados en la salida aún cuando todavía no haya consumido todo los datos de entrada que le llegan. Un ejemplo aparecen en la Figura 2.2.

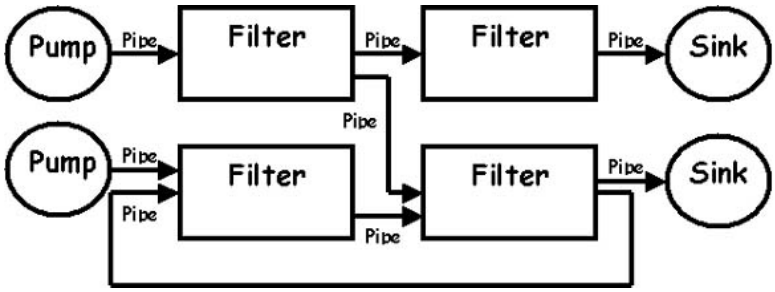


Figura 2.2: Estructura del estilo arquitectónico *Tubería y filtro*. [Fuente: (Garfixia, Accessed March 4, 2020)]

Invariantes del estilo

- Los filtros son entidades independientes, no comparten su estado con otros filtros
- Los filtros no conocen a sus filtros vecinos, tan sólo pueden imponer restricciones a los datos que le entran y garantizar que cumple con restricciones a los datos que saca como salida. Ni siquiera debería afectar a la salida final del sistema la forma en la que cada filtro realizan el procesamiento incremental de los datos

Algunos subestilos o tipos más epecíficos del estilo *Tubería y filtro* son:

- Estilo ‘ *Tubería lineal* (pipelines).- Todos los filtros están en una única secuencia (ver Figura 2.3)
- Estilo *Tubería limitada*.- Restringe la cantidad máxima de datos que pueden estar en un momento dado en una tubería
- Estilo *Secuencial por lotes* (sequential batch).- Es un caso extremo en el que cada filtro procesa todos los datos de entrada en conjunto, como una única entidad. En este caso las tuberías realmente no tienen sentido porque no se requiere una corriente de datos. Realmente éste se considera un estilo independiente

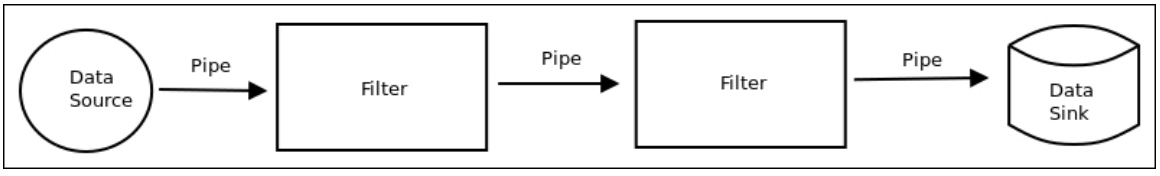


Figura 2.3: Estructura del estilo arquitectónico *Tubería y filtro*. [Fuente: (Balachandran Pillai, 2017)]

El mejor ejemplo de este estilo lo proporciona Unix en su intérprete de comandos (shell). Para ello proporciona una notación que represente las tuberías, es decir, que conecte los componentes (el carácter "—") junto con mecanismos de ejecución para implementar las tuberías.

Por ejemplo, para obtener el total de ficheros en el directorio actual que contienen la palabra "Maze" en su nombre:

```
ls | grep Maze | wc -l
```

Otros ejemplos se dan en el procesamiento de señales o en la programación paralela.

Ventajas

- Diseño fácil: Los sistemas con este estilo son muy fáciles de ser entendidos
- Reusabilidad: Se pueden construir filtros por la unión de dos filtros, siempre que haya acuerdo en la información que pasa de uno a otro
- Mantenibilidad, incluyendo mejoras: Se pueden añadir nuevos filtros o reemplazar los que ya existen
- Permiten análisis especializado, como rendimiento y análisis de interbloqueos
- Concurrencia: cada filtro puede realizar una tarea y actuar en paralelo con otros

Inconvenientes

- No permiten proceso interactivo, sólo programación por lotes
- Bajo rendimiento: Si no se programan bien se pueden producir cuellos de botella en los filtros más lentos
- Complejidad en la programación: para que el rendimiento sea alto, requieren de una programación compleja

2.2.2. Estilo *Abstracción de datos y organización OO*

Un estilo que transparenta las características modulares cuando se usa un lenguaje de programación modular u OO. Así, en este estilo, la representación de los datos y las operaciones que éstos pueden hacer se encapsulan: Los componentes son instancias de tipos abstractos de datos (TADs) en lenguajes modulates u objetos en lenguajes OO. Los conectores son las vías de comunicación entre los componentes (asociaciones, en terminología OO). Una conexión concreta se hace operativa cuando se realiza entre ellos alguna llamada

o invocación (o envío de mensajes, en terminología OO) a procedimientos o funciones (o métodos, en terminología OO). A los componentes también se les llama gestores por ser los encargados de preservar la integridad de los datos que contienen.

En la Figura 2.4 se muestra un ejemplo. "op" es la invocación (procedure call) que se hace para conectar un objeto con otro.

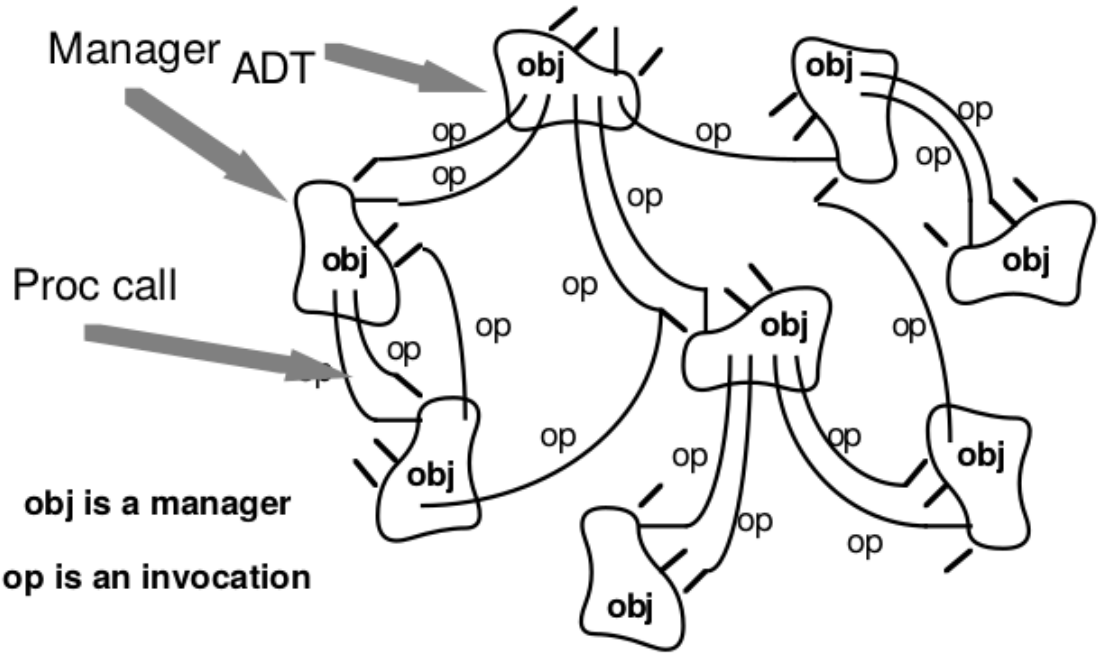


Figura 2.4: Ejemplo del estilo arquitectónico *Abstracción de datos y organización OO*. [Fuente: (Shaw and Garlan, 1996, pg. 23)]

Invariantes del estilo

- Encapsulamiento: Datos y operaciones sobre ellos están encapsulados
- El objeto es responsable de preservar la integridad de su estado
- Ocultación: La implementación de las operaciones están ocultas a otros objetos. Los datos también pueden ocultarse

Ventajas

- Gracias al ocultamiento de las operaciones es posible cambiar la implementación de éstas sin que afecte a sus clientes

- Gracias al encapsulamiento, los diseñadores pueden descomponer la resolución de un poblema como un conjunto de agentes con responsabilidades específicas que cooperan para la resolución del mismo

Inconvenientes

- Los componentes deben conocerse entre ellos para poder interaccionar (piénsese como alternativa en el estilo *Tubería y filtro*)
- Puede haber efectos colaterales entre objetos que están conectados entre sí directa o indirectamente

2.2.3. Estilo *Basado en eventos*

Una alternativa a la llamada a procedimientos propia del estilo *Abstracción de datos y organización OO* (a partir de ahora, estilo OO, para simplificar) es la posibilidad de no hacer una invocación directa a un procedimiento o función (un método en OO) sino de forma indirecta. Para ello, los componentes con métodos cuya invocación dependa del estado de otros componentes, deben subscribirse a una lista de partes interesadas en este otro componente. Cuando en este otro componente se produce un cambio de estado (evento), lo retransmite o anuncia a los subscriptores de la lista, o lo publica y otro componente recoge la publicación para transmitirla a los subscriptores. Los subscriptores reciben el anuncio y realizan las operaciones necesarias a partir de la información que les ha sido comunicada.

Variantes Hay dos variantes dentro de este estilo:

- Invocación implícita (estilo *Manejador de eventos* o *Publicar/subscribir*.- El componente que anuncia un evento no conoce quiénes son sus clientes a la escucha. La forma de comunicación es mediante un tercer componente intermediario (manejador de evento) que es el que recoge los cambios publicados y sí conoce a los componentes que están interesados para retransmitírselos.
- Invocación explícita (patrón *Observador*[1]).- El componente en el que se produce el evento tiene él mismo la lista de subscriptores y les anuncia a todos el cambio que se ha producido.

En la Figura 2.5 puede verse la diferencia entre el patrón de diseño o subestilo *Observador* y el subestilo *Manejador de eventos*.

[1]Obsévese que este no es un patrón arquitectónico sino de diseño. Este es un ejemplo de la fuerte relación que hay entre ellos, hasta tal punto de que muchos autores no hacen tal distinción, considerando además que diferentes patrones pueden aplicarse sobre un mismo sistema de forma jerárquica.

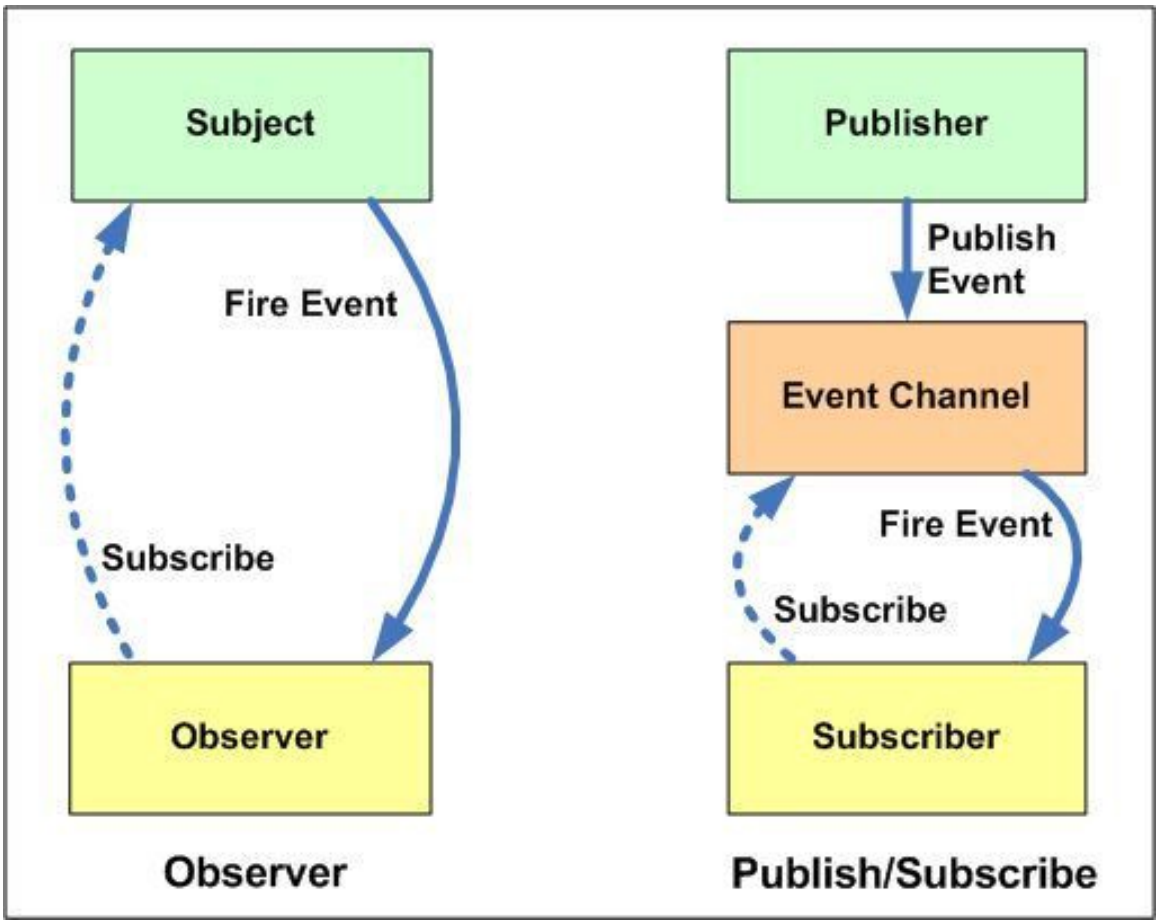


Figura 2.5: Diferencia entre el estilo arquitectónico *Basado en eventos* y el patrón de diseño *Observador*. [Fuente: https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

```
public class Observable
{
    public ActionEvent<Object, Object> onStateChange;

    public void eventRelease()
    {
        // do something
        // choose appropriate args
        onStateChange.invoke(null, null);
    }

    /* versus ************************************************/
    Observator[] observers;
    public void informObservators()
```

```
    {
        foreach (Observer observer in observers)
            // choose appropriate args
            observer.updateState(null, null);
    }
}

public class Observer
{
    public Observer(Observable observable)
    {
        observable.onStateChange += updateState;
    }
    /* versus ************************************************/
    public void updateState(Object arg1, Object arg2)
    {

    }
}
```

Un ejemplo de uso del estilo de invocación implícita son los depuradores software. Cuando el depurador se para en un punto de interrupción (breakpoint), se anuncia esta parada. Los que reciben el anuncio operan en consecuencia, por ejemplo, el editor de textos para ponerse en la línea de parada y la ventana con el estado de las variables se actualiza.

Ventajas

- Reusabilidad: Pueden añadirse componentes al sistema sólo añadiéndolos a la lista de subscriptores
- Evolución más sencilla del sistema: Los componentes pueden reemplazarse (cambiando incluso su interfaz) sin que ellos afecte a las interfaces del resto de componentes con los que se relaciona (pues lo hacen de forma indirecta)

Inconvenientes

- Pérdida de control: Los componentes no saben realmente cómo afecta lo que hacen en otros componentes, ni siquiera el orden en el que los cambios se producen en los otros componentes (sobre todo en el estilo "Manejador de eventos"
- Difícil comprobación de integridad: Derivada de la pérdida de control también se hace difícil establecer las condiciones de ejecución de una operación concreta, frente a la tradicional llamada a procedimientos (estilo OO o estilo "Organización programa

principal/subrutinas" (ver más abajo) para los que bastan las precondiciones y post-condiciones de una función o procedimiento para conocer su comportamiento en un momento determinado.

2.2.4. Estilo *Modelo-Vista-Controlador* (MVC)

Fue descrito por primera vez para su aplicación en Smalltalk Reenskaug (1979). La idea fundamental es separar la funcionalidad del sistema ("lógica del negocio" o "dominio de la aplicación") de la interfaz de usuario. En la actualidad está ampliamente extendido. Ejemplos de uso son Ruby on Rails y

Se compone de tres elementos:

- Modelo: Conjunto de clases que representan la lógica de negocio de la aplicación (clases deducidas del análisis del problema). Encapsula la funcionalidad y el estado de la aplicación.

- Vista: Representación de los datos contenidos en el modelo. Para un mismo modelo pueden existir distintas vistas.

- Controlador: Es el encargado de interpretar las ordenes del usuario. Mapea la actividad del usuario con actualizaciones en el modelo. Puesto que el usuario ve la vista y los datos originales están en el modelo, el controlador actúa como intermediario y mantiene ambos coherentes entre sí.

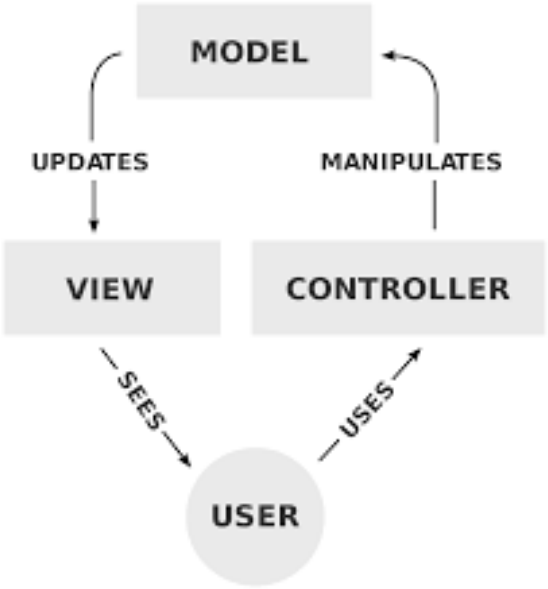


Figura 2.6: Estructura del estilo arquitectónico MVC de Controlador ligero (ver más abajo). [Fuente: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller]

Ventajas

- El desarrollo es más sencillo y limpio.
- Facilita la evolución por separado de vista y modelo.
    - Cualquier modificación que afecte al dominio de la aplicación, implica una modificación sólo en el modelo. Las modificaciones a las vistas no afectan al modelo, simplemente se modifica la representación de la información, no su tratamiento.
- Incrementa la reutilización y la flexibilidad.

Incovenientes

- Curva de aprendizaje muy pendiente, pues muchas veces este estilo combina tecnologías múltiples que los desarrolladores
- Puede añadir complejidad superflua si la aplicación no lo requiere

Variantes Hay principalmente dos variantes en este estilo:

- Controlador ligero o Modelo activo: La actualización de las vistas es hecha directamente desde el modelo (Figura 2.6). Para mantener la independencia de ambas, en esta opción es necesario el uso a su vez del estilo "Basado en eventos"

- Controlador pesado o Modelo pasivo: el controlador conoce las vistas e interactúa con ellas para notificarles que se han hecho cambios en el modelo de forma que las vistas pedirán los datos al modelo (lo cuál puede hacerse directa o indirectamente)

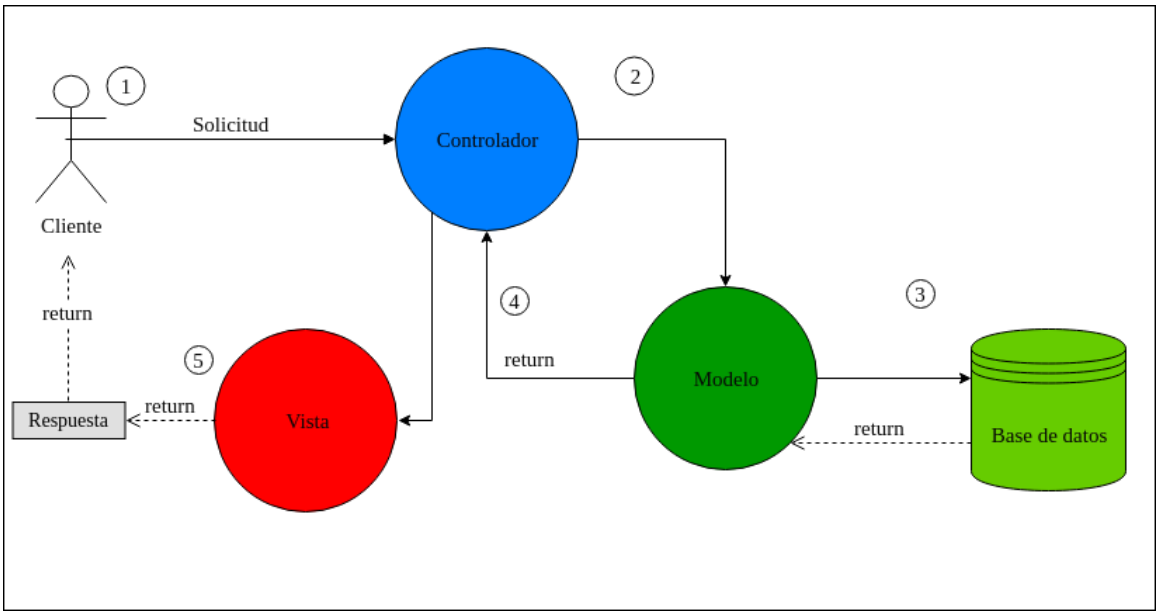


Figura 2.7: Estructura del estilo arquitectónico MVC de Controlador pesado. [Fuente: https://articulosvirtuales.com/articles/educacion/que-es-el-modelo-vista-controlador-mvc-y-como-funciona]

Un ejemplo de implementación del estilo MVC del controlador ligero con el estilo de eventos de invocación implícita es el que hace la librería Java SWING. En ella, se ha pasado del clásico MVC a una "arquitectura de modelo separable" (separable model architecture) en la que el controlador y la vista están fuertemente acoplados en una única clase llamada $UIObject$ (o $UIDelegate$) (Ver Figura 2.8[2]).

[2]El elemento UI Manager es usado para almacenar información general a la apariencia de todos los componentes (color de fondo por defecto, tipo de letra por defecto, tipos de bordes, etc.), estando todos manejados por el mismo UI Manager.

Figura 2.8: Implementación del estilo MVC hecha por Java SWING.

Un ejemplo de utilización del estilo de controlador pesado es el que realiza el entorno de trabajo Angular JS para desarrollo Web, escrito en JavaScript (ver Figura 2.9).

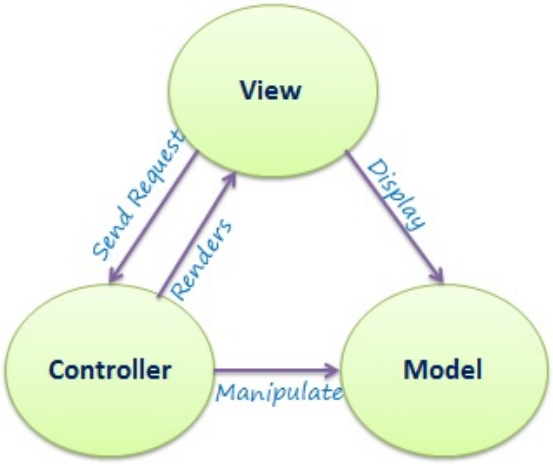


Figura 2.9: Estructura del estilo arquitectónico MVC pesado implementado por Angular JS. [Fuente: https://w3tutoriels.com/angularjs/angularjs-mvc/]

2.2.5. Estilo *Sistema por capas*

En este estilo el sistema se organiza por capas, de forma que cada capa sirve a la capa superior y es cliente de la capa inferior. Cuando una capa sólo puede ver a las adyacentes podemos hablar de que se trata de una máquina virtual de cara a su capa cliente. La Figura 2.10 proporciona un ejemplo con sólo tres capas, mientras que la Figura 2.11 proporciona un diagrama genérico.

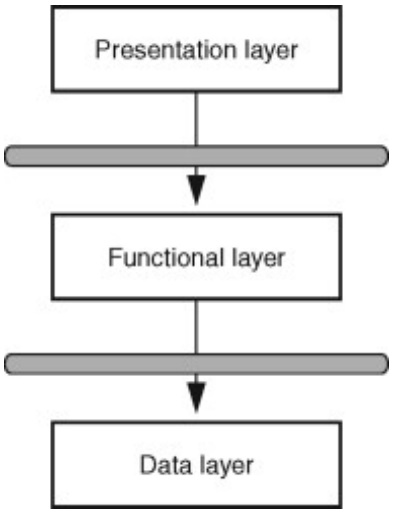


Figura 2.10: Estructura del estilo arquitectónico "Sistema por capas" con sólo tres capas. [Fuente: (Züllighoven, 2005)]

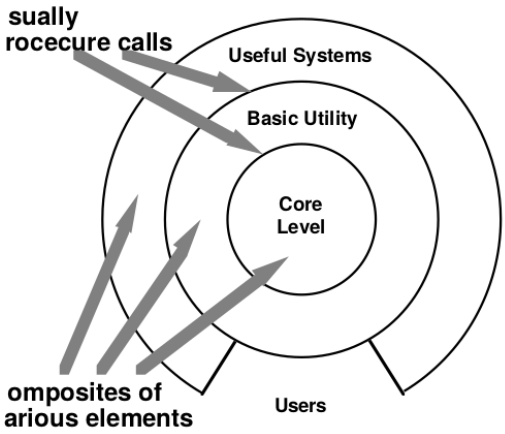


Figura 2.11: Estructura del estilo arquitectónico "Sistema por capas" genérico. [Fuente: (Shaw and Garlan, 1996, pg. 25)]

Un ejemplo de este estilo es la forma en la que interactúa todo el software de un ordenador (capas: firmware y sistema operativo, utilidades, aplicaciones de usuario). Otro son las capas de un Sistema de Gestión de Bases de Datos (SGBD) (capas: física –gestor de datos y metadatos en disco–, lógica –gestor de operaciones de consulta y actualización de datos como entidades provistas de significado para el usuario–, de aplicación: gestión de vistas y solicitudes de usuario).

Ventajas

- Soportan diseños basados en niveles de abstracción incremental.

- Reusabilidad: permiten la sustitución de la implementación de una capa por otro código, siempre que se mantega su interfaz (igual que en el estilo OO.

Incovenientes

- Difícil adaptación: No todos los sistemas pueden ser concebidos de forma jerárquica. Incluso si lo hacen pueden perder eficiencia frente a un estilo que permita mayor cohesión entre componentes.

2.2.6. Estilo *Repositorio*

Está compuesto por un componente como estructura central de datos (almacén de datos o repositorio) y el resto de componentes, externos, que operan sobre él.

Variantes Se considera que existen dos variantes:

- Estilo "Repositorio básico": Cuando es una petición externa hacia un componente externo la que dispara una petición para que se ejecute un proceso concreto de ese componente que a su vez solicitará información al componente central. Un ejemplo es el uso de una base de datos distribuida.

- Estilo *Pizarra* (blackboard): Es el propio estado en el que se encuentra el componente central el que dispara el proceso a realizarse en un componente externo o "fuente de conocimiento" (knowledge source).

Un ejemplo de la estructura del estilo *Pizarra* puede verse en la Figure 2.12.

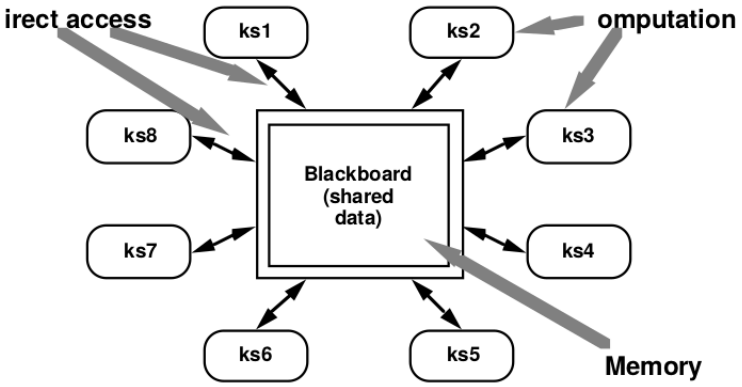


Figura 2.12: Estructura del estilo arquitectónico *Pizarra*. [Fuente: (Shaw and Garlan, 1996, pg. 26)]

2.2.7. Estilo *Intérprete*

Permite crear una máquina virtual que sirva de puente entre el nivel de computación básico que está disponible (programa objeto) para interactuar con el hardware y el nivel lógico que es entendido por la semántica de una aplicación.

Usualmente está formado por cuatro componentes:

- El motor que interpreta (o intérprete en sí)
- El contenedor con el pseudocódigo a ser interpretado
- Una representación del estado en el que se encuentra el motor de interpretación
- Una representación del estado en el que se encuentra el programa que está siendo simulado

La Figure 2.13 muestra la estructura del estilo *Intérprete*.

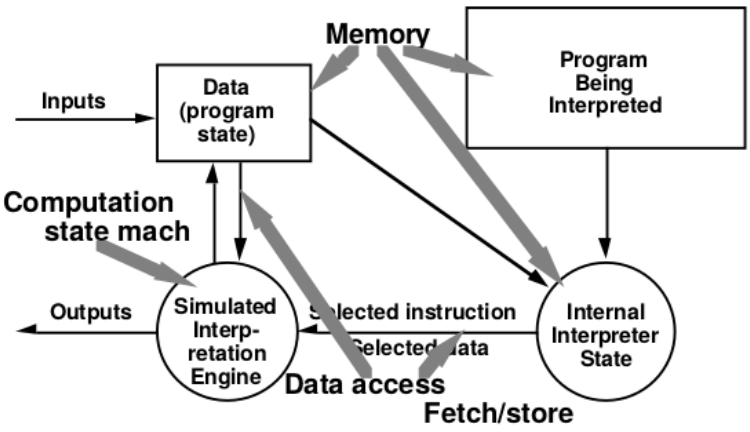


Figura 2.13: Estructura del estilo arquitectónico *Intérprete*. [Fuente: (Shaw and Garlan, 1996, pg. 27)]

Un ejemplo de intérprete es la máquina virtual Java (JVM) disponible para cualquier plataforma. EL código de entrada son los bytecodes (ficheros .class resultados de la compilación de los .java).

2.2.8. Estilo *Control de procesos*

Su propósito es el de mantener las propiedades de salida (variables controladas) de un proceso en un nivel o cercano a un nivel de referencia (set point). Se usa sobre todo en el entorno industrial.

Variantes Existen algunas variantes:

- *Ciclo abierto* (Figure 2.14).- En muy pocos casos, cuando todo el proceso es completamente predecible, no es necesario vigilar el proceso (controlar el estado de las variables y reaccionar en consecuencia).

- *Ciclo cerrado* (Figure 2.15).- Es necesario supervisar el sistema para corregir la salida según el cambio en los valores de las variables de entrada.

  - *Control de procesos retroalimentado* (Figure 2.16)
  - *Control de procesos preventivo* (Figure 2.17)

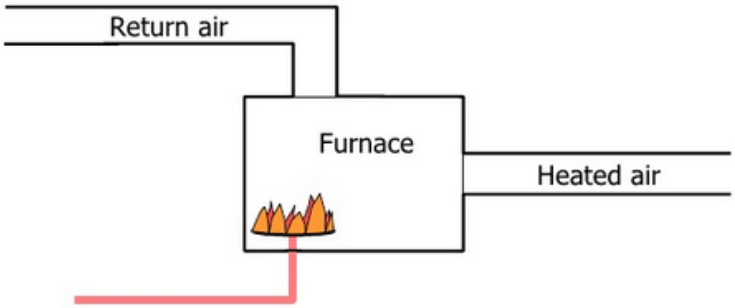


Figura 2.14: Ejemplo de sistema con estilo arquitectónico *Control de procesos de ciclo abierto*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

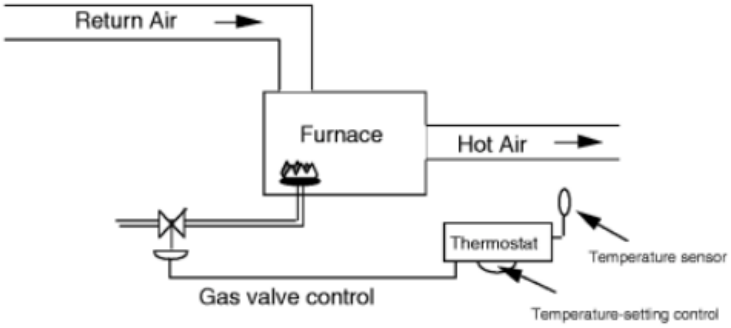


Figura 2.15: Ejemplo de sistema con estilo arquitectónico *Control de procesos de ciclo cerrado*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

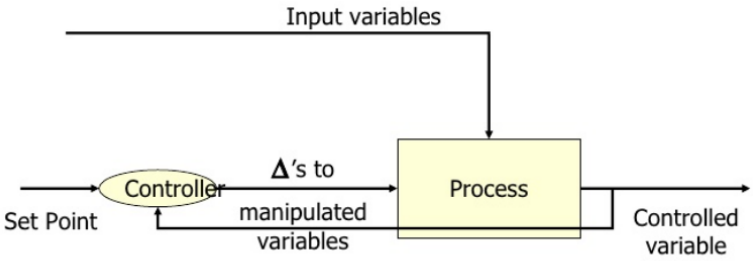


Figura 2.16: Estructura del estilo arquitectónico *Control de procesos retroalimentado*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

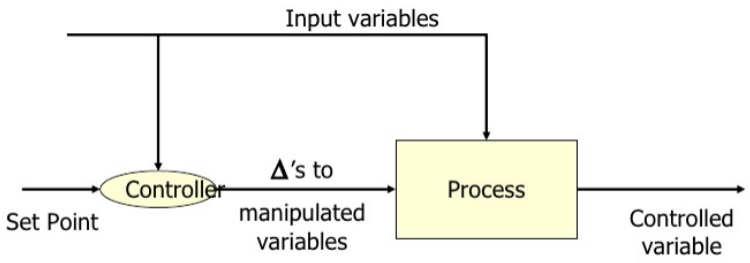


Figura 2.17: Estructura del estilo arquitectónico Co *ntrol de procesos preventivo.* [Fuente: (Shaw and Garlan, 1996, pg. 30)]

La Figura 2.18 muestra un ejemplo de un sistema de control de procesos retroalimentado: un sistema de control de la velocidad de crucero. En concreto es un diagrama de bloques del sistema.

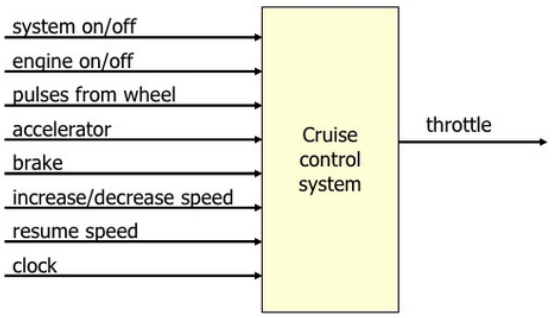


Figura 2.18: Diagrama de bloques del sistema de control de la velocidad de crucero. [Fuente: (Shaw and Garlan, 1996, pg. 52)]

Solución mediante arquitectura OO (*Abstracción de datos y organización OO*)

Ejemplo de solución propuesta por la arquitectura de Booch: 2.19

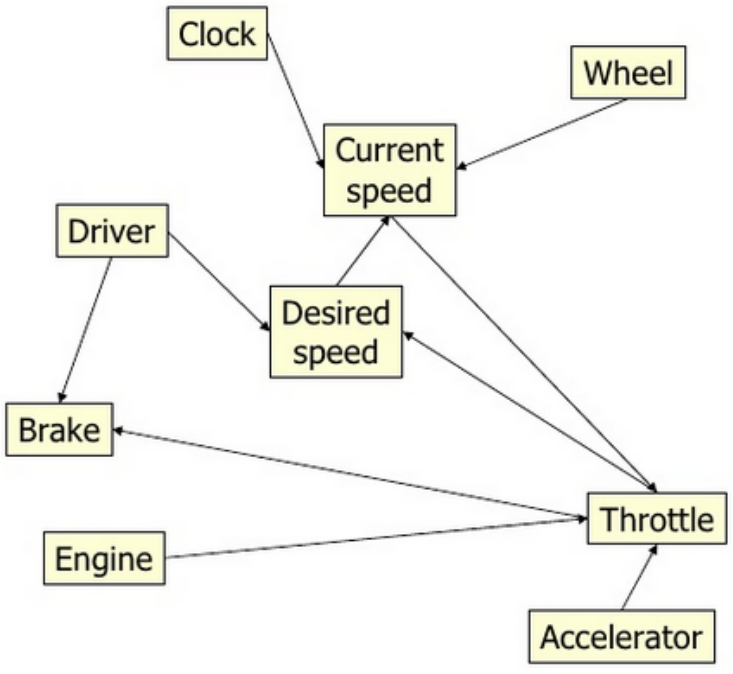


Figura 2.19: Diagrama de clases para el sistema de control de la velocidad de crucero. [Fuente: (Phillips et al., 1999)]

Diagrama del estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero propuesto por David Garlan. 2.21

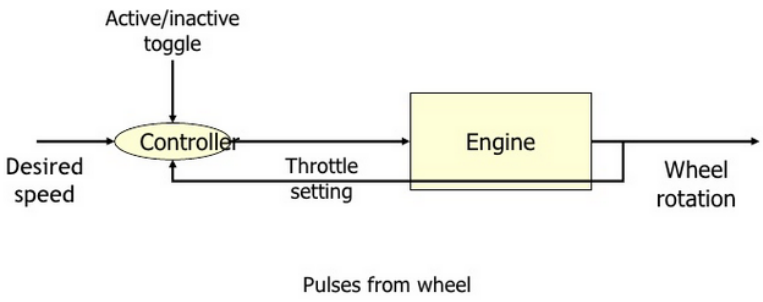


Figura 2.20: Diagrama del estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero. [Fuente: (Phillips et al., 1999)]

Diagrama de componentes usando el estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero propuesto por David Garlan. 2.21

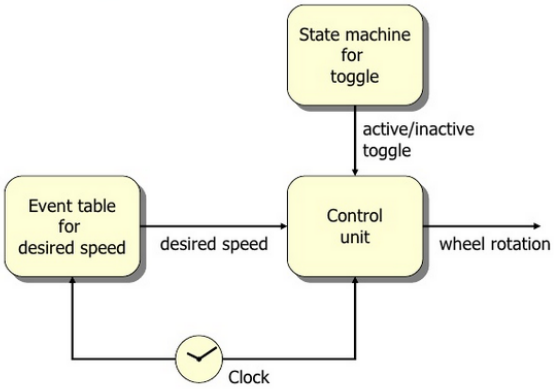


Figura 2.21: Diagrama de componentes). [Fuente: (Phillips et al., 1999)]

2.2.9. Otros estilos arquitectónicos

Hay otras muchas arquitecturas, algunas de ellas se verán más adelante:

- Estilo de *Procesos distribuidos*.- Dentro de éstas existen arquitecturas diversas según criterios diversos, como la topología que relaciona los procesos o el protocolo de comunicación entre procesos. Por ejemplo, atendiendo a la topología de la red podemos tener estilos arquitectónicos en anillo o en estrella.

  - Estilo *Cliente/Servidor*.- Un tipo particular de arquitectura de procesos distribuidos de uso muy generalizado. El servidor representa a los procesos que dan servicio y el cliente a los procesos que reciben el servicio. El servidor no conoce a los clientes por adelantado, mientras que el clente conoce al servidor y accede a él mediante llamada a procedimientos remota (remote procedure call, RPC).

- Estilo *Organización programa principal/subrutinas*.- Es el estilo que proyecta directamente la forma de funcionamiento de los lenguajes de programación que son usados por el sistema cuando éstos no permiten modularidad.

- Estilo *Sistema de transición de estados*.- Utilizado por sistemas reactivos que definen estados y transiciones entre ellos para pasar de un estado a otro.

- Estilos específicos de un dominio.- Como por ejemplo estilos concretos para la aviación, los videojuegos o los sistemas de gestión de vehículos. Al estar el domino restringido el estilo o patrón es más específico y permite generar código a partir de ella de forma automática o semi-automática.

En la realidad, los estilos se combinan de forma que en un solo sistema puede aplicarse más de uno.

Se puede considerar que un componente implementa a su vez otro estilo y así sucesivamente, de forma que se relacionan entre ellos de forma jerárquica. El caso más claro es el que tiene como estilo raíz el estilo *Tubería y filtro*, como por ejemplo ocurre en las tuberías de Unix, que pueden programarse mediante la línea de comandos.

También es posible que un mismo componente forme parte de más de un estilo, porque tenga conectores de varios estilos, como los estilos *Repositorio*, *Tubería y filtro* y *Control de procesos*. Así, la interfaz tendrá una parte específica para cada tipo de conector.

2.3. Notaciones actuales para descripción arquitectónica

2.3.1. El estándar IEEE P1471: Aplicación de distintos puntos de vista según las partes interesadas en un sistema software

> **NOTA**: Para evitar numerosas citas literales, debe tenerse en cuenta que el texto de este apartado ha sido traducido, extraído o resumido del trabajo de Rich Hilliard "Using UML for Architectural Description" (Hilliard, 1999).

En 1999 surgió un estándar (*prácticas recomendables*[3]), el P1471, elaborado por el IEEE Architecture Working Group (Group, 1999) [4] para regular las arquitecturas de sistemas software "intensivos" [5].

Este estándar regula en definitiva la manera en la que debe hacerse la *descripción arquitectónica* (DA) de un sistema software.

Objetivos

Los objetivos de IEEE para el P1471 son:

1. Asumir una interpretación amplia del concepto de arquitectura en sistemas software intensivos, es decir, sistemas basados en computadores, incluyendo aplicaciones software, sistemas de información, sistemas embebidos, sistemas de sistemas, lineas de productos y familias de productos donde el software juegue un papel fundamental en el desarrollo, operación o evolución del sistema.

[3] En IEEE hay tres tipos distintos de estándar: estándar en sí, prácticas recomendables y guías.

[4] Este estándar fue modificado en 2011 por el estándar ISOIECIEEE42010.

[5] Un sistema software intensivo es un sistema software o un sistema general donde el software influye de forma fundamental en el diseño, construcción, desarrollo y evolución del sistema.

2. Establecer un marco de trabajo conceptual y un vocabulario para hablar de los aspectos arquitectónicos del sistema, de forma que todos puedan entenderlo. Esto supone ponerse de acuerdo en los significados de algunos términos como *arquitectura*, *descripción arquitectónica* (DA) y *vista*.

3. Identificar y declarar prácticas arquitectónicas sólidas, pues habiendo numerosas propuestas falta proporcionar las bases para que las propuestas puedan definirse, contrastarse y aplicarse.

4. Permitir la evolución de estas prácticas conforme evolucionan tecnologías que sean relevantes.- IEEE reconoce la rápida evolución de las prácticas software arquitectónicas, tanto a nivel industrial como de investigación, y por eso P1471 pretende ser un marco de referencia para que estas prácticas puedan comunicarse, documentarse y compartirse. Por ello, el marco debe ser suficientemente general para acoger las técnicas actuales y suficientemente flexible para que pueda evolucionar.

Ingredientes básicos

El P171 contiene tres ingredientes básicos:

1. Un conjunto de normas de definición de términos tales como *descripción arquitectónica*, *vista arquitectónica* o *punto de vista arquitectónico*.

2. Un marco conceptual que sitúa esos términos en el contexto de los múltiples usos posibles de DAs para la construcción, análisis y evolución de sistemas.

3. Un conjunto de requerimientos sobre una DA de un sistema.

P1471 no exige para una DA consideraciones sobre los sistemas, proyectos, empresas, procesos, métodos o herramientas, sólo que la DA cumpla con los estándares, que se definen con requerimientos ("debe cumplir con").

Es importante destacar que el estándar IEEE P1471 se ha desarrollado de forma que es independiente de la notación.

Entidades conceptuales

En P1471 hay un conjunto básico de entidades conceptuales que son usadas en la definición del estándar y que es necesario conocer:

- Descripción arquitectónica.- Colección de productos para documentar la arquitectura de un sistema. No especifica el formato o el medio pero sí unos requerimientos mínimos de contenidos que reflejen las prácticas actuales y el consenso industrial.

- Interesado (stakeholder).- Cualquier individuo, clase o componente externo, institución o rol interesado en el sistema. Algunos ejemplos son: cliente del sistema, usuarios, operadores, personal encargado del mantenimiento del sistema, desarrolladores, comerciales, etc.

- Inquietud (concern).- Cualquier tipo de preocupación sobre la arquitectura que tengan las partes interesadas en el sistema, como las garantías de seguridad, fiabilidad, o mantenibilidad.

- Vista.- Cada una de las partes en las que se divide una DA. P1471 no exige unas vistas concretas pues éstas dependen de la técnica usada, sino que lo deja a criterio de los usuarios del estándar. Las vistas son los mecanismos para separar los distintos intereses, tanto para reducir la complejidad del sistema como para ajustarse a las necesidades de las distintas partes interesadas. A su vez, cada vista puede estar compuesta de viarios "modelos arquitectónicos", cada uno con un esquema distinto de representación, por ejemplo un modelo puede usar un diagrama de clases UML y otro un diagrama de componentes UML.

- Punto de vista.- Contiene las reglas por las que se rigen las vistas concretas. Un aspecto interesante es la idea de reusabilidad de los puntos de vista (que pueden recogerse en una "librería de puntos de vista"), de forma que puedan aplicarse a otros sistemas, considerándolos como referencias o patrones[6]. Para elegir los puntos de vista debemos partir de un inquietud concreto y, a partir de él, elegir un punto de vista con vistas concretas que se ajusten a él. P1471 no propone puntos de vista específicos, sólo que los que se elijan se entiendan y documenten bien. En la DA debe siempre quedar claro el punto de vista que se aplica en cada una de las vistas para hacer éstas inteligibles.

La Figura 2.22 representa el modelo conceptual del estándar P1471.

[6]De aquí deriva el concepto de patrón o estilo arquitectónico.

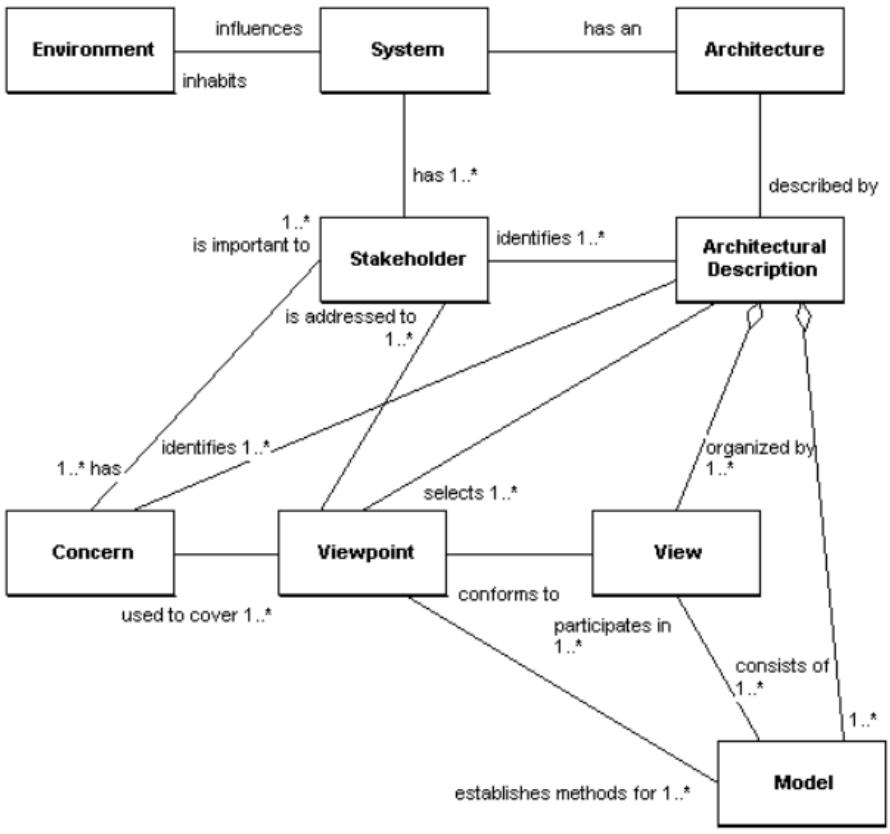


Figura 2.22: Modelo conceptual del estándar IEEE P1471. [Fuente: (Hilliard, 1999)]

CRITERIO DE CALIDAD: Seguridad

La seguridad software es la protección contra toda tipo de amenaza que pueda impedir el correcto funcionamiento del software, tanto debida a ataques intencionados como a accidentes o catástrofes.

CRITERIO DE CALIDAD: Fiabilidad

La fiabilidad software es la probabilidad de que el sistema funcione sin fallos debidos al diseño durante un período específico de tiempo. Garantizar la misma en software muy complejo se hace más difícil.

CRITERIO DE CALIDAD: Mantenibilidad

La mantenibilidad software se define como el grado en el que una aplicación puede comprenderse, repararse y mejorarse. Una baja mantenibilidad eleva de forma considerable el costo de un proyecto software.

Una DA debe identificar a sus partes interesadas y las inquietudes que tengan, y debe dar respuesta a todos estas inquietudes para que sea **completo**.

CRITERIO DE CALIDAD: Completitud

La completitud de una DA de un sistema software es la condición de que da respuesta a las inquietudes de todos las partes interesadas en el sistema.

Cómo declarar un punto de vista

Debe especificar los siguientes elementos:

- Nombre del punto de vista
- Partes interesadas con inquietudes abordados por este punto de vista
- Intereses/inquietudes abordados por este punto de vista
- Lenguaje, técnicas de modelado y métodos analíticos usados
- Fuente, si hay alguna, del punto de vista (autor, citas)

Otros aspectos no obligatorios:

- Cualquier tipo de método de comprobación de la consistencia o completitud incluido por el método usado que pueda aplicarse en los modelos concretos elaborados para una vista
- Cualquier técnica de evaluación o análisis que pueda aplicarse a los modelos elaborados para esa vista
- Cualquier heurística, patrón (en el sentido de estilo o patrón arquitectónico) u otra guía que sirva de ayuda para sintetizar las vistas asociadas con el punto de vista o sus modelos

Cómo realizar una DA

La forma de proceder es que el arquitecto software elija los puntos de vista arquitectónicos desde una lista predefinida de ellos (la librería de puntos de vista) en atención a cada uno de las inquietudes que el sistema debe cubrir. De forma alternativa puede escoger un patrón arquitectónico que de respuesta a cada inquietud concreto.

Una vez elegido cada punto de vista será necesario describirlo como se ha especificado anteriormente, incluyendo entre otros el lenguaje y métodos de modelado a usar.

Una herramienta de modelado (y lenguaje) muy apropiado es UML, pues también está desarrollado de forma independiente del proceso, pudiéndose utilizarse para cualquier tipo de software que se quiera desarrollar. La guía del usuario de UML precisamente proporciona cinco distintos puntos de vista (aunque los llama vistas entrelazadas): dirigida por los casos de uso, centrada en la arquitectura, iterativa, de proceso incremental, ... Hilliard (1999).

Hilliard Hilliard (1999) propone cuatro formas distintas de usar UML para aplicar el estándar IEEE P1471:

1. "Listo para usar" (out of the box) o "predefinido".- Se trata de usar UML como conjunto de herramientas de notación, de forma que para cada vista podemos usar uno o más diagramas UML. Se puede considerar cada tipo de diagrama como un posible lenguaje aplicable a un punto de vista, según la Tabla 2.1.

2. "Extensión ligera".- Se trata de usar los mecanismos de extensión ligera que proporciona UML (etiquetas, estereotipos, restricciones) para añadir información en la DA. Se proponen dos posibilidades:

   - Usar extensiones de UML.- Se amplía UML con un conjunto concreto de estereotipos, valores etiquetados (restricciones), y restricciones concretas en un dominio de aplicaciones
   - Usar variantes UML.- El metamodelo UML se deja como está pero sobre él se construyen variantes.

   El problema de este enfoque es que cada punto de vista necesita su propia extensión, lo que lo hace muy complicado de aplicar.

3. "UML como marco integrador".- Otra posibilidad es que el arquitecto software, o el equipo e empresa, adopten el metamodelo UML como marco integrador para hacer la DA, es decir, la DA se describa por completo con UML y los distintos puntos de vista y las vistas se relacionen entre sí a través de los distintos diagramas UML. Para ello hay que entender la relación entre el estándar P1471 y UML: en UML se considera el concepto de "técnica diagramática" (TD) que permite regular las distintas vistas. Así, una TA permite a los arquitectos software documentar un tipo concreto

de diagrama y añadir extensiones. Con este enfoque, se definen los puntos de vista y se asocia a cada una de sus vistas uno o varios diagramas. Pero además se puede proporcionar notación para representar toda la DA de manera global, por medio de un diagrama que incluye a las partes interesadas, los distintos intereses, los puntos de vista y las vistas (véase como ejemplo la Figura 2.23). Asímismo se deben integrar las vistas de manera que se garantice la consistencia entre ellas. UML proporciona como mecanismo para lograrlo, la relación "refine" por la cuál se representan los mismos elementos el distintos niveles de abstracción. Faltaría sin embargo una forma de indetificar elementos que representan lo mismo en distintas vistas o puntos de vista. Por ejemplo, un elemento en una vista estructural puede ser un componente y en una vista de datos representarse como interfaz de acceso a los datos.

4. "Fuera de la ontología "UML".- A menudo nos encontraremos con puntos de vista para los que no existen diagramas UML y será más fácil usar otras notaciones o técnicas que extender UML. Algunos ejemplos son los diagramas de GANTT, de presupuestos y organizativos para un punto de vista de gestión o los modelos de bloques y de fallos si se trata de un punto de vista de la fiabilidad.

Punto de vista	Tipo de diagrama UML
Estructural	Diagrama de componentes; diagrama de clases
Conductual	Diagrama de interacción; diagrama de actividad; diagrama de estados
Usuario	Diagrama de casos de uso; diagrama de interacción
Distribución	Diagrama de despliegue; diagrama de interacción

Tabla 2.1: Relación entre los puntos de vista arquitectónicos y los tipos de diagrama UML [Fuente: (Hilliard, 1999).

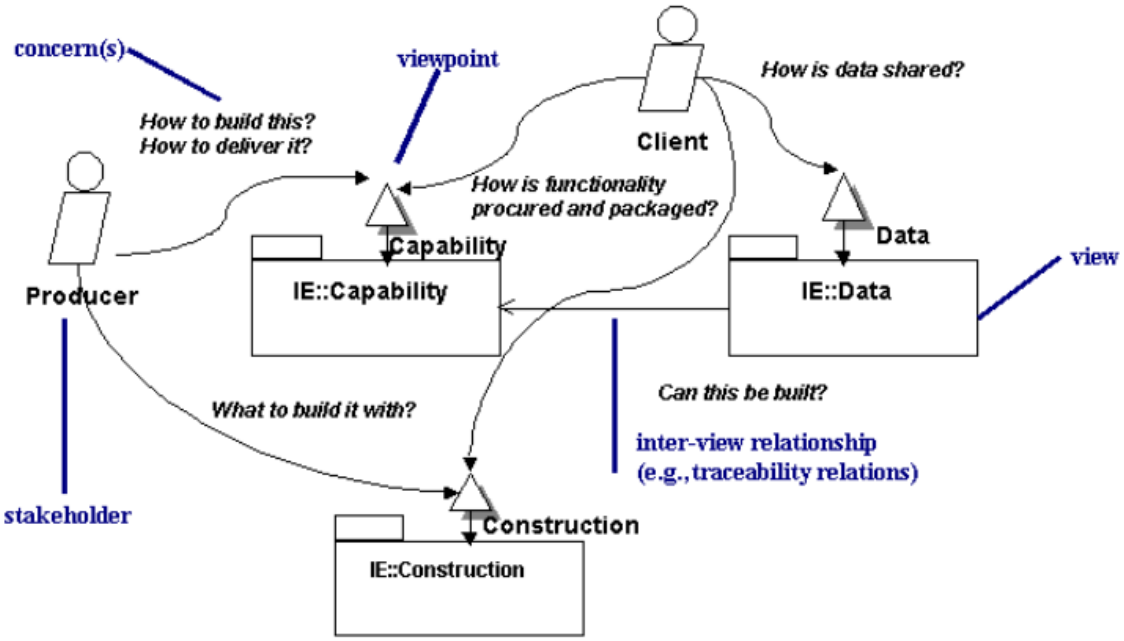


Figura 2.23: Un fragmento de una visión general de un sistema. Los iconos son las partes interesadas, las inquietudes se representan con flechas, los puntos de vista con triángulos y las vistas con paquetes. [Fuente: (Hilliard, 1999)]

2.3.2. Una propuesta concreta basada en el estándar IEEE P1471 (Rozanski, 2011)

> **NOTA**: Para evitar numerosas citas literales, debe tenerse en cuenta que para el resto de esta sección hemos traducido, extraído o resumido del trabajo de Rozansky (Rozanski, 2011).

La propuesta de este estándar se ha desarrollado por algunos autores con cierto detalle. Como ejemplo, Rozansky (Rozanski, 2011) ha detallado una guía sobre cómo identificar y comprometer a las partes interesadas en el sistema (Rozanski, 2011, Cap. 9), cómo identificar las inquietudes o cómo identificar y usar escenarios (Rozanski, 2011, Cap. 10).

Cómo identificar y comprometer a las partes interesadas en el sistema

DEFINICIÓN: Parte interesada

Una parte interesada en la arquitectura de un sistema es un individuo, equipo, organización y tipo de ellos con algún interés en la realización del sistema.

Es muy importante identificar correctamente a los más importantes y a partir de ahí intentar resolver sus inquietudes o preocupaciones. Los grupos más importantes de partes interesadas son:

- Los más afectados por las decisiones arquitectónicas, como usuarios, operadores o los clientes que pagan por la realización del sistema.
- Los que influyen en la forma y en el éxito del desarrollo, como los clientes
- Los que deben incluirse por razones organizativas o políticas, tales como los administrativos, un equipo encargado de la gestión de la arquitectura global, o alguna persona con autoridad en la empresa.

No tener en cuenta las preocupaciones de las partes interesadas más importantes desde el principio hará que cuando las planteen fuercen el rediseño, con un aumento significativo de los costes de desarrollo.

Una parte interesada que haga bien su trabajo debe:

- Estar informado y con la experiencia suficiente para tomar decisiones adecuadas
- Comprometerse en participar en el proceso de desarrollo y ser capaz de tomar decisiones difíciles si llega el caso
- Tener la autoridad necesaria para tomar decisiones y evitar que sean revertidas por otros en el futuro
- Ser capaces de representar a todo un grupo con los mismos intereses y llegar a tener criterios compartidos

Según el papel y las preocupaciones, las partes interesadas se clasifican en distintos tipos. Dependiendo del sistema unos tendrán más importancia que otros, pero rechazar algún tipo dará problemas futuros. Los tipos propuestos son los siguientes:

- Clientes o adquisidores: supervisan que el sistema sea desarrollado
- Asesores: supervisan que el sistema se ajuste a los estándares y regulación legal
- Comunicadores: explican el sistema a otras partes interesadas mediante documentación y material de entrenamiento
- Desarrolladores: construyen y despliegan el sistema a partir de sus especificaciones, o lideran equipos que lo hagan
- Mantenedores: gestionan la evolución del sistema una vez en explotación

- Ingenieros de producción: diseñan, despliegan y gestionan los entornos hardware y software en los que el sistema se construirá, probará y ejecutará

- Proveedores: construyen y/o proporcionan el hardware, software o la infraestructura donde se ejecutará el sistema

- Equipo de apoyo: proporcionan apoyo a usuarios del producto o sistema cuando se ejecuta

- Administradores del sistema: ejecutan el sistema una vez lanzado

- Equipo de prueba: prueba el sistema para asegurar que está preparado para ser usado

- Usuarios: definen la funcionalidad del sistema y hacen uso del mismo

Es necesario mantener un equilibrio entre el número de partes interesadas y la facilidad para llegar a un consenso. El arquitecto debe gestionar el proceso de toma de decisiones y tener una idea clara de la importancia relativa de cada uno de los grupos de partes interesadas. Esto será necesario a la hora de defender una decisión arquitectónica ante partes interesadas que consideren que sus inquietudes están siendo ignoradas.

El propio arquitecto debe considerarse a sí mismo una parte interesada en el sistema.

Se proponen algunos ejemplos para aprender a identificar y elegir de forma apropiada a las partes interesadas.

EJEMPLO: Proyecto de desarrollo a partir de software comercial ("off-the-shelf")

Estos proyectos requieren la selección, adaptación e implementación a partir de un paquete software, de forma que se desarrolla menos software que en un sistema tradicional. Pero el papel de las partes interesadas sigue siendo vital.
Como ejemplo, una empresa A, fabricante de hardware, quiere usar un sistema de planificación de recursos empresariales (Enterprise Resource Planning, ERP) para manejar mejor todos los aspectos de la cadena de producción, desde la petición hasta la entrega. Los gestores han considerado la construcción del sistema mediante un paquete software comercial 'personalizable" ("custom off-the-shelf", COTS) combinado con algún software de desarrollo propio para algunos aspectos más especializados de la funcionalidad. El nuevo sistema debe ser desplegado en un año, a tiempo para una reunión con los accionistas que están considerando una futura financiación de la empresa.
Los adquirientes del sistema incluyen al director gerente, que autorizará la financiación del proyecto, junto con el departamento de compras y los representantes informáticos, que evaluarán distintos paquetes ERP.
Los usuarios del sistema cubren un amplio rango de empleados internos, entre los que se incluyen los que procesan los pedidos, las compras, la financiación, la fabricación y la distribución.
Los desarrolladores, administradores del sistema, ingenieros de producción y mantenedores y personal del departamento de informática así como los asesores se elegirán desde el equipo de prueba de aceptación interna. Los comunicadores incluyen a los entrenadores internos, y el soporte es proporcionado por el equipo interno de ayuda (help desk), posiblemente en unión con los proveedores COTS.

EJEMPLO 2: Proyecto de desarrollo de un producto software

Un producto software es usualmente desarrollado por un proveedor especializado, con el desarrollo a menudo parcialmente financiado por inversores externos. Los usuarios previstos del producto estarán en otras organizaciones que, se espera, comprarán el producto una vez que esté completo. Las partes interesadas para tales proyectos a menudo se reparten entre varias organizaciones.

Como ejemplo, una empresa B, un proveedor de software educativo, quiere desarrollar un producto que será utilizado por los profesores universitarios para administrar sus horarios de clases. La empresa B ha formado una sociedad con una universidad local y ha obtenido algunos fondos de capital de riesgo para el producto. El sistema se ejecutará en PCs y será económico y fácil de usar.

Los adquirientes en este caso incluyen directores gerentes y jefes de producto de la empresa B, el socio educativo (la universidad), y representantes de los capitalistas de riesgo.

Los usuarios del sistema son profesores universitarios y personal administrativo; tenga en cuenta que estos usuarios en realidad no existen todavía como tales porque nadie ha comprado el producto todavía. La representación del usuario deberá obtenerse de alguna otra manera (por ejemplo, hablando con algunos usuarios potenciales del producto).

Los desarrolladores y mantenedores son personal de desarrollo de productos de la empresa B, y los asesores son tomados de las tres empresas asociadas. No hay partes interesadas que sean administradores del sistema real en este ejemplo (esto debe tenerse en cuenta en la arquitectura, por ejemplo, haciendo que el sistema se autogestione). La empresa B y/o las universidades que compran el producto pueden proporcionar el personal de mantenimiento.

Los comunicadores incluyen autores técnicos de la empresa A que escriben la guía de usuario.

Los ingenieros de producción proporcionan los entornos de desarrollo y prueba para la empresa B. También proporcionan y administran la infraestructura para fabricar CDs de productos y distribuir actualizaciones de software a los usuarios a través de Internet.

EJEMPLO 3: Desarrollo subcontratado

Un proyecto de desarrollo subcontratado involucra a una organización que utiliza los servicios de otra para proporcionar sistemas o servicios que normalmente proporcionaría con sus propios recursos internos. Estos proyectos dan como resultado que las partes interesadas que normalmente se encontrarían dentro de la organización adquiriente se encuentren dentro de la organización de servicios externos. Esto puede dificultar la identificación e interacción con estas partes interesadas.
La empresa C, una empresa financiera consolidada, desea expandir su presencia en Internet con la capacidad de comercializar una gama de servicios financieros a clientes últimos. Estos servicios están dirigidos a los residentes del país donde tiene su sede la empresa C, así como a algunos clientes internacionales. La empresa C planea contratar el desarrollo y uso del sistema a un desarrollador web establecido.
Los adquirientes incluyen directores gerentes que autorizarán la financiación del proyecto. Los usuarios incluyen clientes ordinarios, que accederán al sitio web público (como en el ejemplo anterior, estos todavía no existen), junto con el personal administrativo interno, que llevará a cabo sus funciones de trabajo de respaldo.
Los administradores del sistema son personal de la empresa de desarrollo web. Los asesores incluyen el personal interno de contabilidad y abogados de la empresa C, así como los reguladores financieros externos de cualquier país en el que la empresa C desee comerciar.
Los comunicadores, los ingenieros de producción y el personal de mantenimiento son proporcionados por la empresa C y/o la empresa de desarrollo web.

Cuando las partes interesadas aún no existan como grupo, se pueden identificar "partes interesadas de proximidad" ("proxy stakeholders"), personas o grupos que representen las inquietudes de las partes interesadas reales y aseguren que se les da la misma importancia que a las inquietudes de las otras partes interesadas. Por ejemplo, los clientes potenciales pueden ser representados por los gestores de productos del grupo de marketing, que tengan los resultados de pruebas de marketing, o por miembros de la población de usuarios objetivo que estén dispuestos a involucrarse en la concepción de los productos.

Cómo identificar las inquietudes

El concepto de inquietud (del inglés, concern) está definido por Rozansky (Rozanski, 2011, Cap. 6) de una forma más detallada que la que da el estándar IEEE P1471.

DEFINICIÓN: inquietud (concern)

Una inquietud sobre una arquitectura es un requisito, objetivo, restricción, intención o aspiración que una parte interesada tenga para dicha arquitectura.

Por tanto, no sólo incluyen a los requisitos funcionales y no funcionales, que son inquietudes específicas, sin ambigüedad y medibles, sino que también incluyen otras inquietudes que se formulan de forma mucho más vaga y poco definida pero que no por ello es menos importante para el interesado que la formula. Incluso puede ser más importante que algunos requisitos específicos.

EJEMPLO: Mantener la reputación del servicio proporcionado

Un minorista tiene una gran reputación en la calidad del servicio y la respuesta que da a sus clientes. Esto hace que tenga algunos objetivos y aspiraciones para una nueva tienda en línea que quiere construir:

- Los valores, la ética y la reputación del minorista deben reflejarse en la apariencia y el funcionamiento de la tienda en línea y sus procesos de respaldo.
- En todo momento, el sitio web debe tratar de presentar una cara "humana" al cliente (incluso aquellas partes de la misma que están completamente automatizadas).
- La tienda en línea debe ser fácil de usar para los clientes que tienen una experiencia limitada con los ordenadores y con el comercio electrónico.
- La tienda en línea debe ser "responsiva" (rápida de cargar y responder a las acciones del cliente) independientemente de la velocidad de conexión a Internet del cliente.
- La tienda en línea debe cubrir todos los aspectos de la experiencia de compra, incluido un catálogo actualizado y navegable, un sistema seguro de compra en línea, el seguimiento de un pedido y la gestión de devoluciones.

Excepto el último elemento, ninguno de los otros pueden considerarse requisitos formales y medibles, y el último es realmente una declaración del ámbito que se desea cubrir. Sin embargo, si el sistema no cumple con estos objetivos y aspiraciones, probablemente será visto como un fracaso.

Rozansky define además otros dos criterios de clasificación de las inquietudes. El primero las clasifica según si:

- se refieren al problema (inquietudes enfocadas en el problema), como la filosofía o

estrategia del negocio, y responden a las pregunta por qué y qué, o si

- se refieren a la solución (inquietudes enfocadas en la solución), casi siempre derivadas, de forma directa o indirecta de las primeras, como la estrategia TIC, que deriva de la filosofía del negocio, y que responden a las preguntas cómo y con qué.

El segunda las clasifica según si:

- son agentes que influyen y dirigen la toma de decisiones en una cierta dirección (por ejemplo, un objetivo para el negocio o para la tecnología), o si
- son agentes restrictivos sobre las decisiones que puedan tomarse (por ejemplo, un estándar o una política de actuación que estemos obligados a seguir.

Se pueden ver ejemplos de inquietudes clasificadas según estos dos criterios en la Figura 2.24.

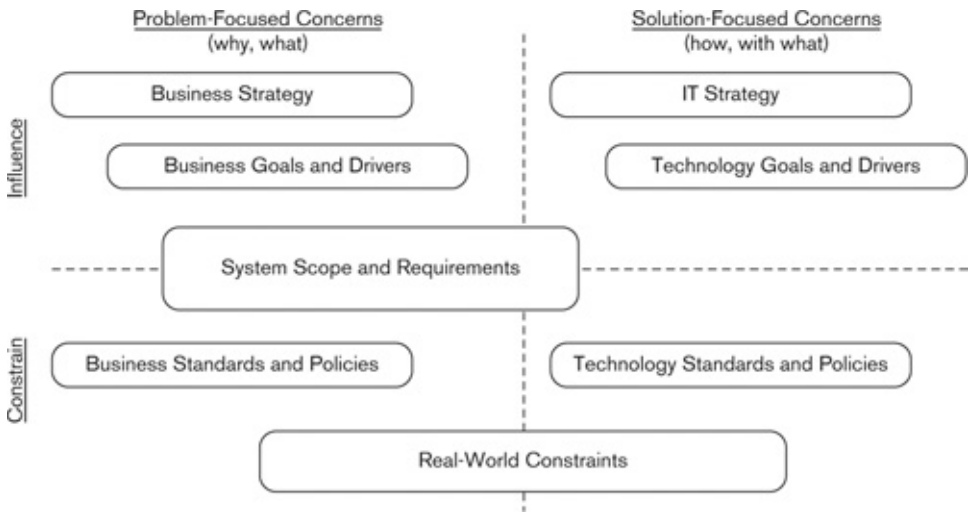


Figura 2.24: Inquietudes según dos criterios de clasificación. [Fuente: (Rozanski, 2011, Cap. 8)]

Cómo identificar y usar escenarios

El arquitecto debe en todo momento ser capaz de asignar prioridades de forma correcta en caso de necesidades de distintas partes interesadas que sean incompatibles entre sí.

A menudo es posible que el arquitecto se olvide de algunas prioridades del sistema impuestas por algunas partes interesadas y se deje llevar por sus propias preferencias personales. Para evitar esto se recomienda el uso de una técnica que le permitirá poder comprobar continuamente que las ideas que desarrolle funcionen de verdad en la práctica. Se trata del

uso de escenarios en la arquitectura, que permiten modelar algunas interacciones entre el sistema y entidades externas que sean especialmente clarificadoras, mediante la descripción de un "escenario arquitectónico".

DEFINICIÓN: Escenario arquitectónico

Un "escenario arquitectónico" es una descripción bien definida de una interacción entre una entidad externa y el sistema. Define el evento que dispara el escenario, la interacción que inicia la entidad externa y la respuesta que se espera del sistema.

Algunos ejemplos de escenarios son:

- Un conjunto particular de interacciones con sus usuarios a las que el sistema debe poder responder
- El procesamiento que debe realizarse automáticamente en un momento determinado, como a fin de mes
- Una situación particular de carga máxima que podría ocurrir
- Una demanda que un regulador externo pueda hacer de un sistema
- Cómo debe responder el sistema a un tipo particular de fallos
- Un cambio que un encargado de mantenimiento podría necesitar hacer en el sistema
- Cualquier otra situación a la que el diseño del sistema deba poder hacer frente

Para hacer un uso efectivo de los escenarios se pueden seguir las siguientes pautas de actuación antes de modelar los escenarios concretos:

- Enfocarse en un conjunto concreto de escenarios, contando con las partes interesadas para priorizar cuáles son los más importantes para guiar la toma de decisiones
- Usar escenarios distintos, evitando una tendencia a modelar escenarios parecidos que reducen mucho su utilidad
- Incluir el uso de escenarios que tengan en cuenta criterios de calidad (ver más adelante las perspectivas y su relación con los puntos de vista)
- Incluir el uso de escenarios fallidos, como los que consideran que haya problemas de falta de información, sobrecarga, fallos de seguridad, etc., interesantes en especial para ayudar a garantizar los criterios de calidad

- Involucrar lo más posible a las partes interesadas, que pueden ralentizar el proceso de descripción arquitectónica al proporcionar numerosa información, pero que lo van a enriquecer y hacerlo más real al ser capaces de ver aspectos y prioridades que el arquitecto puede desconocer por completo. Son además ellos los que deben priorizar los distintos aspectos del sistema

Rozansky distingue dos tipos básicos de escenarios:

- Escenarios funcionales.- Responden al qué, se relacionan con los requisitos funcionales y se suelen definir como una secuencia de eventos externos (derivados generalmente de un caso de uso) a los que el sistema debe responder de una forma particular. Para describirlos suele proporcionarse la siguiente información:

  1. Descripción general: una breve descripción de lo que el escenario debe ilustrar

  2. Estado del sistema: el estado del sistema antes de que ocurra el escenario (si es significativo). Esto suele ser una explicación de cualquier información que ya debería estar almacenada en el sistema para que el escenario sea significativo

  3. Entorno del sistema: cualquier observación importante sobre el entorno en el que se ejecuta el sistema, como la falta de disponibilidad de sistemas externos, el comportamiento particular de la infraestructura, las restricciones basadas en el tiempo, etc.

  4. Estímulo externo: una definición de lo que hace que se dispare el escenario, como los datos que llegan a una interfaz, la entrada del usuario, el paso del tiempo o cualquier otro evento de importancia para el sistema

  5. Respuesta requerida del sistema: una explicación, desde el punto de vista de un observador externo, de cómo el sistema debe responder al escenario

EJEMPLO: Escenario funcional en un sistema que sumariza los datos de entrada

Actualización estadística incremental

- Descripción general: cómo el sistema maneja un cambio en algunos de los datos base existentes
- Estado del sistema: ya existen estadísticas de resumen para el trimestre de ventas al que se refieren las estadísticas incrementales. Las bases de datos del sistema tienen suficiente espacio para hacer frente al procesamiento requerido para esta actualización
- Entorno del sistema: el entorno de implementación funciona normalmente, sin problemas
- Estímulo externo: una actualización de un subconjunto de las transacciones de ventas para el trimestre anterior llega a través de la interfaz externa "Datos de Carga Masiva" ("Bulk Data Load")
- Respuesta requerida del sistema: los datos entrantes deberían activar automáticamente el procesamiento estadístico en segundo plano para actualizar las estadísticas resumidas del trimestre afectado para reflejar los datos actualizados de las transacciones de ventas. Las estadísticas de resumen anteriores deben permanecer disponibles hasta que las nuevas estén listas

- Escenarios de calidad del sistema.- Responde al cómo debe reaccionar el sistema a un cambio en el entorno para garantizar uno o más criterios de calidad. No siempre los cambios en el entorno pueden modelarse como estímulos de entrada, como ocurre en los escenarios funcionales. Para describirlos suele proporcionarse la siguiente información:

  1. Descripción general: una breve descripción de lo que el escenario debe ilustrar
  2. Estado del sistema: el estado del sistema antes de que ocurra el escenario, si el comportamiento especificado en el escenario depende de ello. Para escenarios de calidad, esto puede necesitar definir aspectos del estado de todo el sistema (como un nivel de carga en todo el sistema) en lugar de la información almacenada en el sistema
  3. Entorno del sistema: cualquier observación importante sobre el entorno en el que se ejecuta el sistema, como la falta de disponibilidad de sistemas externos, el comportamiento particular de la infraestructura, las situaciones basadas en el tiempo, etc.

4. Cambios en el entorno: una explicación de lo que ha cambiado en el entorno del sistema que hace que ocurra el escenario. Esto podría ser cambios o fallas en la infraestructura, cambios en el comportamiento del sistema externo, ataques de seguridad, modificaciones requeridas o cualquiera de los otros cambios en el entorno que requieren que el sistema posea una propiedad de calidad particular para tratar con ellos

5. Comportamiento requerido del sistema: una definición de cómo debe comportarse el sistema en respuesta al cambio en su entorno (por ejemplo, cómo debe responder el sistema, desde un punto de vista de rendimiento cuantificable, a un aumento definido en el número de solicitudes que llegan por minuto)

EJEMPLO: Tres escenarios de calidad que sumarizan los datos de entrada

1. **Problemas de tamaño en la actualización diaria de datos**
    - Descripción general: cómo se comporta el procesamiento del sistema al final del día cuando los volúmenes de datos regulares se superan repentinamente
    - Estado del sistema: el sistema tiene estadísticas resumidas en su BD que ya se han procesado, y los elementos de procesamiento del sistema se cargan poco a poco, a la velocidad actual de carga
    - Entorno del sistema: el entorno funciona correctamente; los datos llegan a una velocidad constante de $1,000 - 1,500$ artículos/hora
    - Cambios en el entorno: la tasa de actualización de datos en un día en particular aumenta repentinamente a $4,000$ artículos por hora
    - Comportamiento requerido del sistema: cuando comienza el procesamiento al final del día, el sistema debe procesar los datos de ese día en un tiempo de procesamiento que no exceda un límite configurable por el sistema. Si se excede, el sistema debería dejar de procesar datos y descartar el trabajo en proceso, dejando el resumen estadístico anterior en su lugar y registrar un mensaje de diagnóstico (incluyendo causa y acción tomada) en el sistema de monitoreo
2. **Fallo en la instancia de la base de datos de resúmenes**
    - Descripción general: cómo se comporta el sistema cuando falla la BD donde quiere escribirse
    - Entorno del sistema: el entorno de implementación funciona bien
    - Cambios en el entorno: al escribir estadísticas de resumen en la base de datos, el sistema recibe una excepción que indica que la escritura falló (por ejemplo, la BD está llena)
    - Comportamiento requerido del sistema: debe dejar de procesar inmediatamente el conjunto de estadísticas en el que está trabajando y cualquier trabajo en progreso. El sistema debe registrar un mensaje fatal en el sistema de monitoreo y apagarse
3. **Necesidad de una dimensión adicional de resumen**
    - Descripción general: cómo el sistema puede hacer frente a la necesidad de ampliar el procesamiento estadístico proporcionado
    - Entorno del sistema: el entorno de implementación funciona normalmente, como se entregó inicialmente
    - Cambios en el entorno: surge la necesidad de admitir una nueva dimensión en las estadísticas de resumen para resumir las ventas por tipo de opción de pago utilizado
    - Comportamiento del sistema requerido: el equipo de desarrollo debe poder agregar un nuevo procesamiento requerido sin cambiar la estructura general del sistema (como interfaces o interacciones entre elementos) y con un esfuerzo total de menos de 4 personas-semana

Hay que tener en cuenta que los escenarios que identifican situaciones de fallo, a menudo no dan las respuestas de forma muy concreta ni robusta y pueden ser inaceptables. Pero al escribir el escenario ya se puede discutir con las partes interesadas en el sistema sobre el problema en cuestión y pensar en formas más concretas de afrontar las situaciones especiales.

Catálogo de puntos de vista

Asimismo, Rozansky ha propuesto un catálogo con siete distintos puntos de vista a tener en cuenta en todo sistema software ((Rozanski, 2011, parte III)) (ver Figura 2.25):

- Punto de vista contextual.- Describe las relaciones, dependencias e interacciones entre el sistema y su entorno (personas, sistemas y otras entidades externas con las que interactúa). Este punto de vista es el que requieren muchas partes interesadas en el sistema, ayudándoles a entender las responsabilidades del sistema y su relación con la institución.

- Punto de vista funcional.- Describe los elementos funcionales del sistema, sus responsabilidades, interfaces e interacciones de alto nivel. Es la piedra angular de la mayoría de las DAs y la primera parte que suelen leer todos las partes interesadas. Es además el que orienta la forma de otros puntos de vista, tales como el informativo, el concurrente, el de despliegue y otros. Tiene además un impacto muy significativo en las propiedades de calidad del sistema, tales como la mantenibilidad, la seguridad y el rendimiento.

- Punto de vista informacional (o de datos).- Describe la forma en la que el sistema almacena, manipula, gestiona y distribuye la información. Desarrolla una vista completa, pero sólo a alto nivel, de la estructura estática de datos y el flujo de la información. Su objetivo es dar respuesta a preguntas fundamentales sobre el contexto, la estructura, la propiedad, la latencia, las referencias y la migración de información.

- Punto de vista concurrente.- Describe la estructura de concurrencia del sistema y mapea los elementos funcionales a unidades concurrentes para identificar de forma clara qué partes del sistema se ejecutan de forma concurrente y cómo se coordinan y controlan. Para ellos es necesario crear modelos que muestren el proceso y las estructuras de hebra que usará el sistema, así como los mecanismos de comunicación entere procesos para coordinar estas operaciones.

- Punto de vista de desarrollo.- Describe la arquitectura del proceso de desarrollo software. Las vistas de desarrollo comunican los aspectos de la arquitectura que necesitan las partes interesadas implicados en la construcción, prueba, mantenimiento y mejora del sistema.

- Punto de vista de despliegue.- Describe el entorno en el que el sistema será utilizado y las dependencias que tiene el sistema con éste. Incluye el entorno hardware que requiere el sistema (nods de procesamiento, conexiones de red, capacidades de almacenamiento en disco), los requisitos técnicos de cada elemento externo y el mapeo entre los elementos software y el entorno de ejecución que lo ejecutará.
- Punto de vista operacional.- Describe cómo el sistema será utilizado, administrado y mantenido una vez en fase de explotación. Entre las tareas más significantes que debe describir están las de instlación, gestión y uso del sistema. El objetivo de este punto de vista es el de identificar las estrategias de todo el sistema para responder a las inquietudes que las partes interesadas puedan tener sobre la operabilidad del sistema e identificar soluciones que den respuesta.

El punto de vista contextual describe las relaciones, dependencias e interacciones entre el sistema y su entorno (las personas, los sistemas y las entidades externas con las que interactúa).

Los puntos de vista funcional, informacional y concurrente caracterizan la organización fundamental del sistema.

Los puntos de vista de desplieque y operacional definen el sistema una vez puesto en explotación.

El punto de vista de desarrollo se usa para gestionar la creación del sistema.

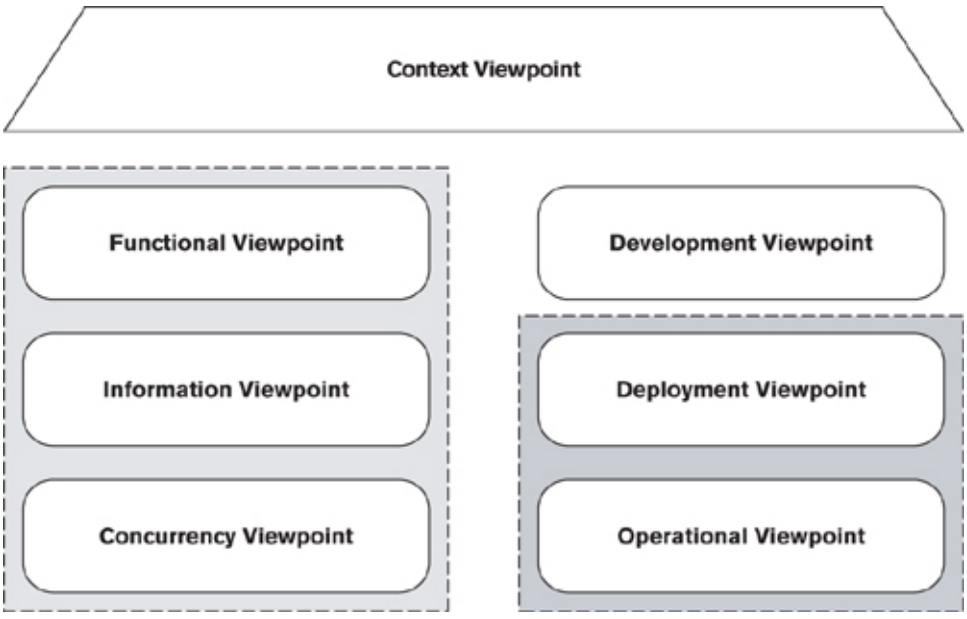


Figura 2.25: Agrupamiento de puntos de vista propuesto por Rozansky. [Fuente: (Rozanski, 2011)]

Este catálogo no está cerrado. No no hay que aplicar todos los puntos de vista, sino que dependerá de cada sistema concreto. Además pueden surgir nuevos puntos de vista.

Un primer paso para empezar a realizar una DA es entender la arquitectura del sistema, las habilidades y la experiencia de las partes interesadas, el tiempo disponible para el desarrollo, etc. para después seleccionar puntos de vista y vistas.

Plantilla para definir un punto de vista

Rozansky propone una plantilla (Tabla 2.2 para definir cómo aplicar un punto de vista en un proyecto software concreto.

Detalle	Descripción
Inquietudes	Aquéllas que preocupan a las partes interesadas, identificando a aquéllos con más interés en este punto de vista
Modelos	Los modelos más importantes que se usarán para representar las distintas vistas, junto con las notaciones usadas y las actividades a realizar para construirlos
Problemas y errores comunes	Aquéllo que deba ser evitado, junto con técnicas a usar para reducir el riesgo
Lista de comprobación	Todas las cuestiones que no deban olvidársenos al desarrollar el punto de vista y cuándo deben revisarse para ayudar al buen desarrollo (correcto, completo, preciso)

Tabla 2.2: Plantilla para describir la aplicación de un punto de vista en un proyecto concreto. [Fuente: (Rozanski, 2011)

Descripción detallada de un punto de vista: el punto de vista contextual

Veremos aquí una descripción detallada de cómo aplicar este punto de vista con algunos ejemplos (Rozanski, 2011, cap. 16).

Este punto de vista debería ser el primero en ser descrito aunque muchas veces no se incluye en la DA porque se supone que se conoce, y se aborda en primer lugar el punto de vista funcional, quizás añadiendo sólo un mero "diagrama de contexto". Es muy importante describirlo de forma detallada porque puede ser necesario para cumplir algunos requerimientos del sistema y porque en el resto de puntos de vista podemos necesitar referirnos a los distintos elementos del sistema externo.

Para la descripción, adaptaremos la plantilla para puntos de vista (Tabla 2.2) a este punto de vista concreto, tal y como aparece en la Tabla 2.3.

Detalle	Descripción
Definición	Describe las relaciones, dependencias e interacciones entre el sistema y su entorno
Inquietudes	Ámbito del sistema y responsabilidades; identificación de las entidades externas y servicios y datos usados; naturaleza y características de las entidades externas; identificación y responsabilidades de las intefaces externas; naturaleza y características de las interfaces externas; otras interdependencias externas; impacto del sistema en su entorno; y completitud, consistencia y coherencia global
Modelos	Modelo de contexto y escenarios de interacción
Problemas y errores comunes	Entidades exernas incorrectas u omitidas; dependencias implícitas omitidas; descripción de interfaces imprecisas u omitidas; contexto o ámbito implícito o asumido; interacciones especialmente complicadas; abuso de jerga
Partes interesadas	Todos las partes interesadas, pero especialmente los compradores, usuarios y desarradores
Aplicabilidad	Todos los sistemas

Tabla 2.3: Plantilla de descripción del punto de vista contextual. [Fuente: (Rozanski, 2011)

Inquietudes

Las inquietudes más importantes desde este punto de vista son:

- Ámbito del sistema y responsabilidades.- Considera cuáles son las responsabilidades principales del sistema. Debe ser breve y fácil de entender por todos las partes interesadas (no debe confundirse con la definición completa de los requisitos del sistema). Generalmente el ámbito viene ya impuesto, pero si no lo está, debemos definirlo a partir de consultar a las partes interesadas en el sistema. A veces se puede querer dejar de forma explícita lo que el sistema excluye.

EJEMPLO: Comercio electrónico al por menor

Un ejemplo de las responsabilidades que deben incluirse dentro del ámbito del sistema son:

- Presentar el catálogo de productos, con imágenes y descripción de cada producto
- Proporcionar una herramienta de búsqueda flexible (por nombre de producto, tipo, palabra clave, tamaño, etc.
- Aceptar pedidos de productos
- Aceptar pago con tarjeta de crédito (con aprobación asíncrona y notificación al cliente
- Proporcionar interfaces de conexión automática con un sistema de soporte para la entrega del producto

Como ejemplos de capacidades excluidas en una primera versión del sistema, tenemos:

- La capacidad para modificar o cancelar pedidos (sólo se podrá hacer por teléfono)
- La posibilidad de pagar de otra forma distinta a usar tarjeta de crédito
- Muestra del stock y posibilidad de reservar hasta disponibilidad

- Identificación de las entidades externas, servicios y datos usados.- Una entidad externa o actor es un sistema, organización o persona con el que interactúa de alguna forma nuestro sistema (ofreciendo/consumiendo servicios o datos a/de nuestro sistema). Algunos ejemplos son:
  - Otro sistema dentro de la empresa/institución que crea el sistema a modelar (se les suele referir como "sistemas internos" o "sistemas propios")
  - Otro sistema que forma parte de una empresa/institución distinta (nos referimos a ellos como "sistemas externos")
  - Una pasarela u otro componente software que oculta a otros sistemas (externos o internos)
  - Un almacen de datos externo al sistema (por ejemplo una base de datos compartida o un almacén de datos)

  - Un periférico u otro dispositivo físico externo al sistema (como servidores de mensajería compartida o robots de búsqueda)
  - Un usuario, un tipo de usuario u otra persona o rol tales como operadores o personal de apoyo

- Naturaleza y características de las entidades externas.- Pueden afectar de forma significativa a nuestro sistema. Algunos ejemplos son la estabilidad y disponibilidad de una entidad externa, su rendimiento, la localización física, la calidad de los datos, etc. En definitiva, criterios de calidad externos de esas entidades (no internos a ellas), es decir, que afecten al exterior, y por tanto a nuestro sistema. También podemos necesitar considerar las entidades externas que no son sistemas, como por ejemplo si un usuario habla el mismo lenguaje natural de nuestro sistema o si la pasarela para usar un fax compartido tiene características de funcionamiento que deban tenerse en cuenta.

  EJEMPLO: Sistema de reserva de viajes

  Estos sistemas intercambian información con otros similares de cualquier parte del mundo. Algunos de ellos, por ejemplo los situados en lugares exóticos, pueden tener interrrupciones en su disponibilidad debido a diferencias horarias o mayor índice de fallos, lo cual puede hacer que nuestro cliente pierda su reserva, algo enormemente dañino.
  Para evitarlo, las interfaces de los sistemas de reserva de viajes tienen que ser muy bien diseñadas, por ejemplo permitiendo "idempotencia" (repetir la misma operación un número determinado de veces sin que dé error), grabar los intentos en una base de datos y notificarlos a los operadores del sistema o permitir continuar transaciones o transferencias de datos en el punto en el que dio error en vez de empezar desde el principio.

- Identidad y responsabilidades de las interfaces externas. Hay que identificarlas todas, entendiendo que cada una de ellas puede cubrir alguno de los siguientes servicios:

  - Proveedor/consumidor de datos, identificando los contenidos, ámbito y significado de los datos transferidos
  - Proveedor/consumidor de servicios, identificando la semántica de la petición (incluyendo parámetros), las acciones a realizar para satisfacer la petición, los datos a devolver, acciones ante excepciones así como errores, estados, etc. a devolver
  - Proveedor/consumidor de eventos, junto con la identificación de dichos eventos, su significado, contenido, volumen y tiempos de ocurrencia probables

- Naturaleza y características de las interfaces externas.- Puede haber una gran diferencia entre la calidad de las conexiones con las entidades externas (las interfaces externas) y los sistemas en sí, pudiendo la interfaz ser el cuello de botella o factor restrictivo en la interacción, por ejemplo cuando la conexión es de bajo ancho de banda y poco fiable pero el sistema con el que nos conecta es altamente resiliente. Algunos ejemplos de características de las interfaces incluyen:

  - Volúmenes esperados (número de peticiones o transferencias, tamaño de los datos, fluctuaciones por temporadas, crecimiento esperado con el tiempo)
  - Si las interacciones están programadas (para momentos predefinidos), ocurren como respuesta a eventos o son ad hoc
  - Si las interacciones están completamente automatizadas, o completamente manuales (como cuando el usuario guarda un fichero o envía un email) o algo intermedio
  - Si las interacciones son transaccionales, i.e. tienen que realizarse por completo
  - Estado crítico y exigencias de tiempo, como por ejemplo cuando se requiere que una interacción particular se termine antes del final del día para poder ser registrada en el sistema contable
  - Si las interacciones son de proceso por lotes (grandes conjuntos de datos son tratados como una unidad), basada en mensajes o de emisión secuencial (streaming)
  - El nivel de seguridad requerido (autenticación, autorización, confidencialidad, etc.
  - El nivel de servicio que puede esperarse de la interfaz (en términos de tiempo de respuesta, latencia, escalabilidad, disponibilidad, etc.)
  - La naturaleza técnica de la interfaz y los protocolos usados (estándar abierto o propietario)
  - Formatos de datos y de ficheros

- Otras interdependencias externas.- A veces existen otro tipo de dependencias que no son de datos, de funciones o de eventos, desde o hacia en sistema a implementar, y que pueden ser difíciles de identificar. En esta inquietud debemos identificar la naturaleza de estas dependencias y su impacto en la arquitectura del sistema (qué funcionalidades deben añadirse a nuestro sistema para poder tener en cuenta estas dependencias)

EJEMPLO: Sistema de venta electrónica al por menor

Como ejemplo, en la Figura 2.26 se muestra un sistema de comercio electrónico (e-Commerce System) de venta al por menor, que iteracciona con varias entidades software externas[a]. Este es un ejemplo donde se muestran interdependencias externas entre los sistemas de reparto (Fulfillment System) y de cuentas de cliente (Customer Accounts System). El sistema de reparto tiene su propia lista de direcciones de reparto verificadas para cada cliente y rechazará cualquier petición que se haga a una dirección que no tenga registrada. Sin embargo, esta lista es una copia de la lista del sistema de cuentas de cliente. Cuando un cliente hace un pedido y además actualiza la dirección de envío, el sistema debe tener en cuenta esta dependencia, así como el tiempo para que la lista de direcciones se actualice en el sistema de reparto. Si no, los pedidos podrían ser rechazados porque el sistema de reparto aún no tenga esa dirección registrada.
El impacto que esta dependencia puede tener en la arquitectura es el de permitir el reenvío de la petición al sistema de reparto si ha sido rechazada la anterior, o retrasar las peticiones que conlleven actualización de la dirección de reparto para dar tiempo a que se actualicen las direcciones en el sistema de reparto.

[a]En este diagrama, se utiliza el estereotipo <<system>> para reflejar el sistema y el resto de "cuadros", que aparecen sin estereotipo, significa que son entidades externas software.

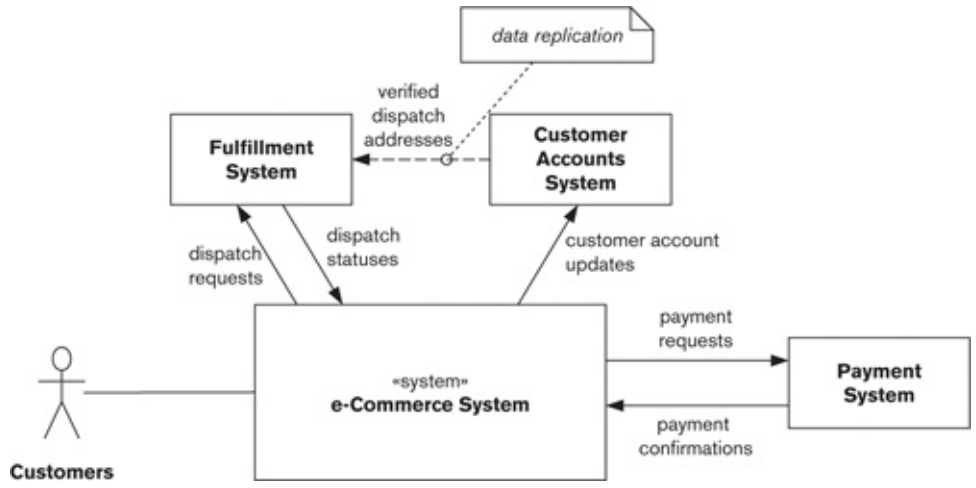


Figura 2.26: Un diagrama de contexto elaborado con UML. [Fuente: (Rozanski, 2011)]

- Impacto del sistema en su entorno.- Se trata del impacto del sistema ya en explotación, tanto dentro de la organización como a nivel externo, e incluye aspectos como los siguientes:

  - Cualquier sistema que dependa del nuestro y que pueda requerir cambios a nivel funcional, de interfaces o mejoras en el rendimiento o en la seguridad
  - Cualquier sistema que deba dejar de utilizarse cuando el nuestro entre en explotación
  - Cualquier dato que deba migrarse a nuestro sistema

- Completitud, consistencia y coherencia global.- A menudo nuestro sistema será parte de un sistema mucho mayor, incluso compartido entre varias organizaciones mediante redes públicas o privadas. Este "paisaje global" puede llegar a ser muy complejo y difícil de entender.

  Una preocupación (inquietud) de los usuarios particulares de nuestro sistema es que la solución global les provea de la funcionalidad que necesitan independientemente de qué sistema concreto implemente cada función o gestione cada conjunto de datos.

  > EJEMPLO: Sistema de venta electrónica al por menor
  >
  > En los primeros sistemas de comercio electrónico se ponía todo el interés en proporcionar catálogos de productos de gran calidad para hacer los sitios atractivos a los compradores y ser competitivos. Sin embargo, se descuidaba los procesos de pago, reparto o tratamiento de excepciones. Esto finalmente les daba mala reputación en el servicio al cliente, incluso llegando a la quiebra.

  Es cierto que esta inquietud es más responsabilidad del arquitecto de la empresa que del arquitecto de la aplicación, pero si se tiene en cuenta por el arquitecto de la aplicación puede aumentar la probabilidad de éxito de forma significativa. Es mucho más apreciable por los usuarios una aplicación consistente y coherente que una fragmentada e inconsistente.

  Al menos se debe garantizar que los procesos de negocio más importantes tengan una cobertuda adecuada, sea por sistemas manuales o automatizados. Del mismo modo, todos los datos que requieran los procesos deben almacenarse en algún lugar (por nuestro sistema o de forma externa) y ser accesibles a todos aquellos sistemas que los necesiten.

- Inquietudes para cada interesado en el sistema.- En la Tabla 2.4 se muestran las inquietudes que cada una de las partes interesadas en el sistema puede tener desde el punto de vista contextual.

Tipo de interesado	Inquietudes
Compradores (clientes)	Ámbito y responsabilidades del sistema; identificación de las entidades externas y servicios y datos usados; impacto de los sistemas en su entorno
Asesores	Todas las inquietudes
Comunicadores	Ámbito y responsabilidades del sistema; identificación y responsabilidades de las entidades externas; identidad y responsabilidades de las interfaces externas
Desarrolladores	Todas las inquietudes
Ingenieros de producción	Naturaleza y características de las interfaces externas; impacto del sistema en su entorno
Administradores del sistema	Todas las inquietudes
Equipo de prueba	Todas las inquietudes
Usuarios	Ámbito y responsabilidades del sistema; identificación de las entidades externas y servicios y datos usados; completitud, consistencia y coherencia global

Tabla 2.4: Inquietudes de las distintas partes interesadas desde el punto de vista contextual. [Fuente: (Rozanski, 2011, Cap. 16)

Modelos: El modelo de contexto El modelo más importante y muchas veces el único es el "modelo de contexto", que presenta una imagen global del sistema completo y su relación con el exterior, y que suele incluir los siguientes tipos de elementos:

- El sistema en sí
- Las entidades externas
- Las interfaces

Notación

Debe contemplarse además la notación que será usada, siendo una de las más comunes UML (que no tiene diagrama de contexto como tal pero que puede ser creado a partir de un diagrama de casos de uso o de un diagrama de clases). Se representará cada componente de la siguiente forma:

- El sistema en sí como un componente y usando estereotipos.
- Los sistemas externos pueden representarse como componentes o actores UML, usando el estereotipo <<external>>.

- Las entidades externas que representan a usuarios que interactúan con el sistema serán representadas siempre como actores UML.

- Las interfaces entre las entidades externas y el sistema pueden representarse en UML como flujos de información, dependencias o asociaciones con navegabilidad incluida (flechas vs líneas).

Puede verse un ejemplo en la Figura 2.27. Obsérvese que falta la navegabilidad en las líneas.

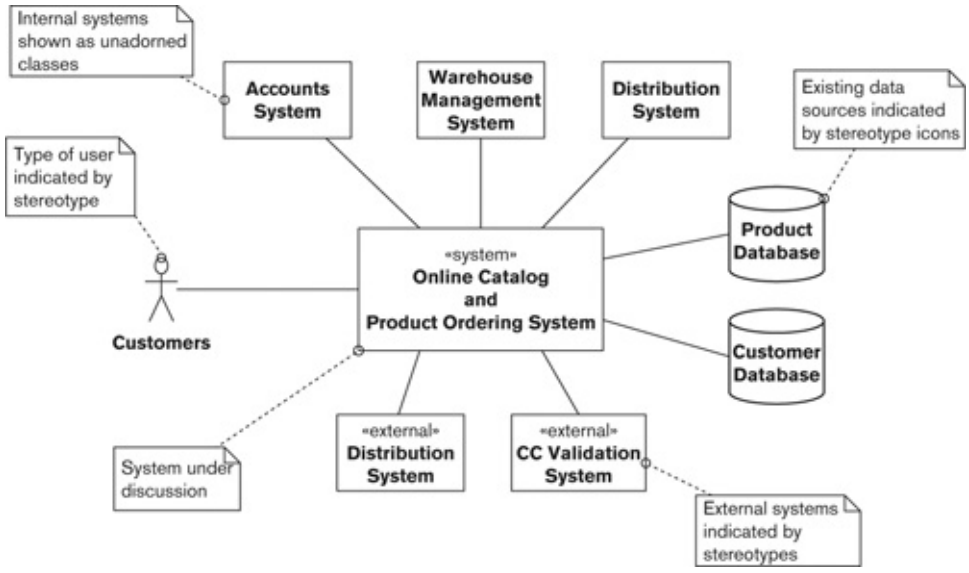


Figura 2.27: Un diagrama de contexto elaborado con UML. [Fuente: (Rozanski, 2011)]

Puesto que UML no proporciona un soporte específico ni potente para los diagramas de contexto, otra posibilidad es usar diagramas informales con notación de "cuadros y líneas", que puede ser más simple. Puede verse un ejemplo en la Figura 2.28.

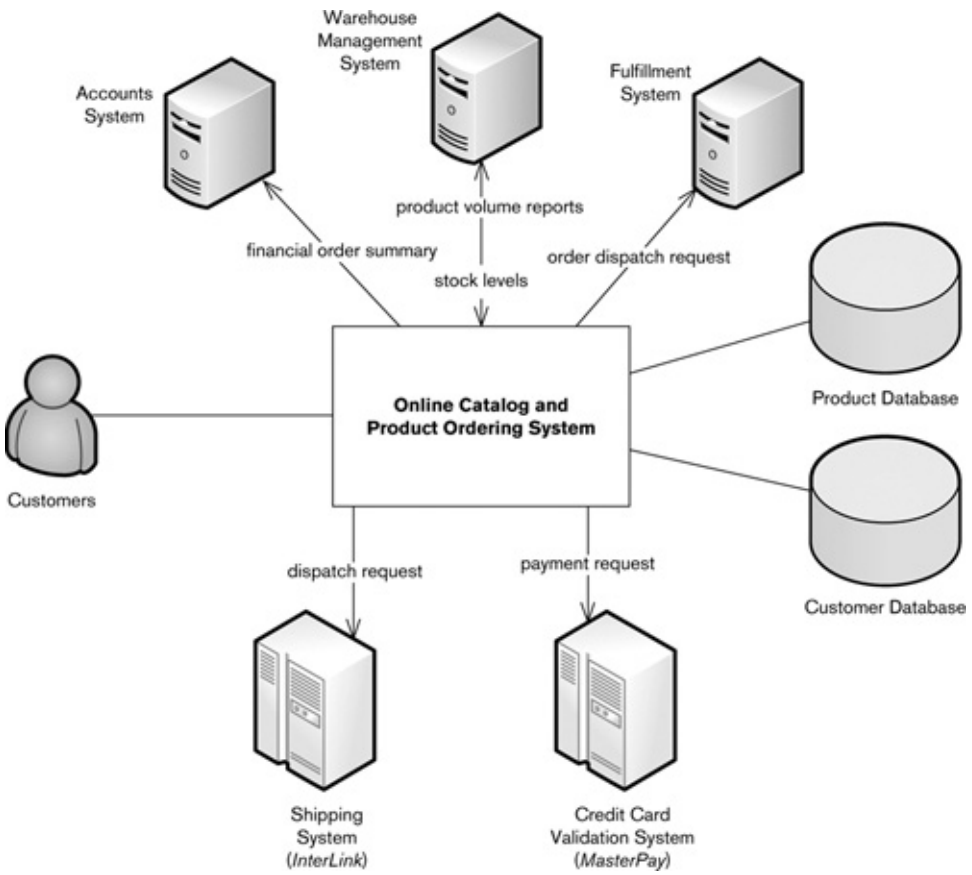


Figura 2.28: Un diagrama de contexto elaborado de forma libre con "cuadros y líneas". [Fuente: (Rozanski, 2011)]

La ventaja es que puede ser mucho más expresivo que UML y más fácil de entender por parte de algunos de las partes interesadas en el sistema. El problema es que obliga a explicar la notación.

Actividades

Para realizar el diagrama de contexto hay que llevar a cabo las siguientes actividades previas, que deben formar parte del documento elaborado para describir la aplicación de este punto de vista en el sistema a desarrollar:

- Revisar los objetivos del sistema
- Revisar los requisitos funcionales claves
- Identificar las entidades externas

- Definir las responsabilidades de las entidades externas
- Identificar las interfaces entre el sistema y cada entidad externa
- Identidicar y validar las definiciones de las interface
- Hacer un seguimiento del flujo de control e información entre el sistema y las entidades externas y añadir las nuevas interfaces que surjan
- Si se van a describir escenarios o casos de uso concretos, recorrerlos para validar el modelo y añadir cualquier entidad externa o interfaz que sea necesaria

Modelos: Escenarios de interacción

Aunque están originalmente pensados para modelar la interacción entre objetos en el desarrollo OO, pueden adaptarse para ilustrar escenarios arquitectónicos para cualquier tipo de sistemas, siempre que los elementos que intervengan y sus interfaces estén bien definidos. Puede consultarse más sobre el uso de escenarios en la DA, en el apartado 2.3.2.

Si bien es poco probable que se tenga tiempo para modelar todos los escenarios en los que participará nuestro sistema, puede ser útil modelar algunos de los más complicados, polémicos o menos entendidos, especialmente cuando el uso del sistema no está claro o hay desacuerdo entre las partes interesadas.

Notación

Para describir un escenario basta con usarse listas de interacción textual (a diferencia de cuando se definen los casos de uso completos), o bien diagramas de secuencia de UML.

Actividades

Descritas en el apartado 2.3.2.

Problemas y errores comunes

Se incluyen los siguientes:

- Entidades externas ausentes o incorrectas
- Dependencias implícitas entre entidades externas no consideradas, y que pueden afectar al propio sistema
- Descripciones imprecisas o ausentes de una interfaz externa
- Nivel de detalle inapropiado

- Degeneración del entorno, o efectos progresivos de cambios no controlados que pueden no apreciar las partes interesadas
- Contexto o ámbito implícito o asumido
- Complicación de interacciones
- Abuso de jergas tecnológicas que pueden no entender las partes interesadas

Lista de comprobación

Se propone la siguiente:

- ¿Has consultado con todos las partes interesadas quiénes están interesados en el punto de vista contextual (probablemente sean todos)?
- ¿Has identificado todas las entidades externas al sistema y sus resonsabilidades más relevantes?
- ¿Comprendes bien la naturaleza de cada interfaz con cada entidad externa y está documentada en un nivel apropiado de detalle?
- ¿Has considerado las posibles dependencias entre las entidades externas con las que hay que interactuar? ¿Están documentadas estas dependencias implícitas en la DA?
- ¿Ilustra de forma adecuada el diagrama de contexto todas las interfaces entre el sistema y su entorno con las suficientes definiciones aclaratorias en el diagrama?
- ¿Han acordado formalmente todos las partes interesadas con los contenidos del modelo contextual? ¿Está documentado en algún lugar?
- ¿Está situado el modelo contextual bajo algún sistema formal de control de cambios?
- ¿Se sigue el proceso de control de cambios? ¿Se consulta a las partes interesadas para que den su consentimiento formal?
- ¿Se coloca en modelo contextual en algún lugar fácilmente accesible por todos, tal como una carpeta compartida?
- ¿Has identificado todas las capacidades o requerimientos básicos del sistema y están documentandos en el nivel de detalle apropiado?
- ¿Es la definición del ámbito consistente internamente?

- ¿Identifica el entorno cualquier posible restricción tecnológica como por ejemplo plataformas de uso exigidas?
- ¿Está el entorno especificado en un nivel apropiado de detalle, equilibrando brevedad con claridad y completitud?
- ¿Has explorado un conjunto de escenarios realistas de interacciones entre el sistema y actores externos?
- ¿Les parece claro el contexto, el ámbito y posibles implicaciones a otros equipos con los que debes interactuar?
- ¿Has comprobado si en el modelo contextual existe alguna información que parezca obvia y debería ser explícitamente descrita pero se ha omitido?
- Tienen los procesos de negocios más importantes una cobertura adecuada, tanto por los sistemas o los procesos manuales definidos?
- ¿Están todos los datos requeridos para los procesos de negocio principales almacenados en algún lugar, interna o etrenamente?
- ¿Está formulada de forma coherente la solución global?

Descripción detallada de otro punto de vista: el punto de vista funcional

Veremos ahora una descripción detallada de cómo aplicar este otro punto de vista con algunos ejemplos (Rozanski, 2011, cap. 17).

El resto de puntos de vista están descritos con detalle en los capítulos 18 a 22 del libro de Rozansky (Rozanski, 2011).

Para la descripción, utilizaremos la plantilla para puntos de vista (Tabla 2.2), y la rellenaremos tal y como aparece en la Tabla 2.5.

El punto de vista funcional de un sistema define los elementos arquitectónicos que implementan las funciones del sistema. Documenta la estructura funcional del sistema, incluyendo elementos funcionales clave, sus responsabilidades, sus interfaces y las interacciones entre ellos. Muestra cómo el sistema llevará a cabo las funciones que se requieren del mismo (requisitos funcionales).

Constituye la piedra angular de la mayoría de las DAs y probablemente la vista más fácil de entender por parte de las partes interesadas. Generalmente este punto de vista dirige además la definición de otros puntos de vista, sobre todo de los de información, concurrencia, desarrollo y despliegue. Generalmente se dedica mucho tiempo a refinar la estructura definida en este punto de vista.

Detalle	Descripción
Definición	Describe los elementos funcionales del sistema en ejecución, así como sus responsabilidades, interfaces e interacciones más importantes
Inquietudes	Capacidades funcionales, interfaces externas, estructura interna y filosofía del diseño funcional
Modelos	Modelo de la estructura funcional
Problemas y errores comunes	Interfaces pobremente definidas Responsabilidades no bien entendidas Infraestructuras modeladas como elementos funcionales Vista sobrecargada Diagramas sin definiciones de elementos Dificultades para reconciliar las necesidades de distintos partes intersadas Nivel de detalle erróneo "Elementos divinos" Demasiadas dependencias
Partes interesadas	Todos
Aplicabilidad	Todos los (sub)sistemas

Tabla 2.5: Descripción del punto de vista funcional. [Fuente: (Rozanski, 2011)

Uno de los desafíos más importantes al describirlo es hacerlo al nivel adecuado de detalle, enfocándose en lo que es más significativo desde el punto de vista arquitectónico (lo que tiene mayor impacto en las partes interesadas) y dejando los detalles para los diseñadores (a nivel de patrones, habría que usar patrones arquitectónicos, y dejar los patrones de diseño para un uso posterior por parte de los diseadores). Se debe evitar documentar detalles de implementación física tales como servidores o infraestructura en este punto de vista, para no complicar los modelos y confundir a las partes interesadas, y dejar esos detalles para el punto de vista del despliegue.

Inquietudes

Las inquietudes más importantes desde este punto de vista son:

- Capacidades funcionales.- Definen qué funciones debe llevar a cabo el sistema y cuáles no (de forma explícita o implícita) porque queden fuera del ámbito del sistema o porque se provean en cualquier otro sitio.
- Interfaces externas.- Son los datos, eventos y flujos de control entre nuestro sistema y

otros sistemas. Pueden ir en los dos sentidos: desde o hacia nuestro sistema.

La definición de las interfaces debe incluir tanto la sintaxis (estructura de los datos, llamadas a procedimientos) como la semántica (significado o efecto).

- Estructura interna.- Normalmente puede haber varias formas de diseñar un sistema que cumpla con un conjunto de requisitos. Puede ser construido como una entidad monolítica o como colección de componentes de bajo acoplamiento; puede ser construido a partir de cierto número de paquetes estándar unidos mediante algún middleware apropiado o escrito desde cero; las funciones pueden cubrirse mediante servicios proporcionados por sistemas externos a través de una red o implementadas por la organización, etc. El desafío es saber elegir entre las distintas posibilidades una arquitectura que sea apropiada para satisfacer tanto los requisitos funcionales como los criterios de calidad (requisitos no funcionales).

  La estructura interna del sistema se define mediante sus elementos internos, qué hace cada uno de ellos y cómo interactúan entre sí. Esta organización interna puede tener un gran impacto en las propiedades de calidad del sistema. Por ejemplo, en un sistema complejo que traspasa los límites de la empresa es generalmente más difícil garantizar la seguridad que si se trata de un sistema sencillo que se ejecuta en un par de ordenadores situados juntos.

- Filosofía de diseño funcional.- La mayoría de las partes interesadas en nuestro sistema sólo lo estarán en lo que el sistema hace y en sus interfaces con usuarios y otros sistemas. Sin embargo, algunos otros pueden estar interesados en saber cómo se ajusta la arquitectura a una serie de principios de diseño robusto. Otros, especialmente nuestros clientes, pueden querer de forma implícita que el sistema esté bien diseñado porque eso lo pondrá en explotación de forma más rápida, barata y fácil.

  Algunos ejemplos de características de diseño que pueden formar parte de una buena filosofía de diseño son:

  - Coherencia: de forma que se ha hecho una descomposición en elementos correcta, para que interaccionen formando un todo
  - Cohesión: con todas las funciones relacionadas incluidas en un mismo elemento para realizar diseños menos proclives a errores
  - Consistencia: en cuanto a que los mecanismos y decisiones de diseño se aplican a través de toda la arquitectura facilitando el diseño y el mantenimiento
  - (Bajo) acoplamiento e interdependencia (separación de las inquietudes/funciones): de forma que haya pocas dependencias entre distintos elementos y facilite el diseño y el mantenimiento, aunque un mayor acoplamiento (sistemas monlíticos) puede aumentar la eficiencia

- Extensibilidad[7], flexibilidad funcional[8] y generalidad, vs simplicidad: mayor extensibilidad, flexibilidad funcional y capacidad de generalización hace a los sistemas más difíciles de implementar e ineficientes pero más fáciles de evolucionar; la excesiva simplicidad los hace baratos y eficientes pero difícles de evolucionar e incluso pueden no cumplir todos los requisitos

Estas características de diseño tienen siempre un efecto positivo en algunas cualidades del sistema (criterios de calidad), tales como capacidad de evolución, flexibilidad y mantenibilidad, pero también en otras que parecen tener una relación no tan directa, tales como rendimiento, seguridad, escalabilidad, etc.

Los principios y los patrones arquitectónicos son buenas técnicas para hacer que el sistema cumpla con ciertas características de diseño y pueden guiar a los diseñadores a tomar decisiones de diseño que doten al sistema de las características que se consideren más importantes de garantizar.

- Inquietudes para cada interesado en el sistema.- En la Tabla 2.6 pueden verse las inquietudes más importantes para las distintas partes interesadas en el punto de vista funcional.

Tipo de interesado	Inquietudes
Clientes	Capacidades funcionales básicas e interfaces externas
Asesores	Todas las inquietudes
Comunicadores	Potencialmente todas las inquietudes, hasta cierto límite según el contexto
Desarrolladores	Estructura interna y cualidades de diseño básicas; capacidades funcionales e interfaces externas
Administradores del sistema	Filosofía de diseño funcional básica, interfaces externas y posiblemente la estructura interna
Equipo de prueba	Estructura interna y cualidades de diseño básicas; capacidades funcionales e interfaces externas
Usuarios	Capacidades funcionales básicas e interfaces externas

Tabla 2.6: Inquietudes de los distintos tipos de partes interesadas en el punto de vista funcional. [Fuente: (Rozanski, 2011, Cap. 17)]

[7]Facilidad para añadir nueva funcionalidad.
[8]Facilidad para modificar en el futuro las funciones ya proporcionadas.

Modelos: Modelo de la estructura funcional

Contiene normalmente los siguientes elementos:

- Elementos funcionales.- Un elemento funcional es una parte del sistema ejecutable (tiene sentido en el tiempo de ejecución) y bien definida (en oposición a lo que se define para el tiempo de diseño) con responsabilidades específicas y que presenta interfaces bien definidas que permiten su coneción con otros elementos. En el nivel de abstracción más simple es un módulo de código software, pero también puede ser un paquete de aplicación, o un almacén de datos o incluso todo un sistema completo.

- Interfaces.- Una interfaz es un mecanismo bien definido por el que un elemento puede acceder a las funciones de otro. Se define mediante entradas, salidas y semántica de cada operación ofrecida por el elemento, junto con la naturaleza de la interacción que se requiere para invocar una operación concreta, por ejemplo llamada (síncrona) a procedimiento remoto (Remote Procedure Call, RPC), mensajería (invocación asíncrona), eventos, interrupciones, etc.

- Conectores.- Son las partes de la arquitectura que unen los distintos elementos para que puedan interactuar. Un conector define la interacción entre los elementos que lo usan y permite que la naturaleza de la interacción se considere aparte de la semántica de la operación que se invoca. La naturaleza de las interacciones entre elementos pueden ser fuertemente asociadas a la forma en la que los elementos se conectan.

  Sin embargo, el nivel de especificación de los conectores depende del tipo de interfaz. Si se usa RPC, apenas habrá que decir nada, sólo indicar qué elementos están conectados entre sí. Si se usa una interfaz basada en mensajería (como por ejemplo "Representational State Transfer" (REST), que está basado en el protocolo HTTP, se puede definir un conector como un elemento específico que proporciona la posibilidad de permitir la interacción entre dos entidades a través de él. Por ejemplo, en un servicio web basado en REST (servicio RESTFul), un conector puede ser una API que permita conectar una entidad con otra, como un servicio REST, de forma segura, pasado la información en formato JSON o XML.

- Entidades externas.- Lo normal es que no tengan consideración desde el punto de vista funcional, sino desde el punto de vista de contexto y en el de concurrencia y el de despliegue (en estos últimos para especificar cómo se ejecuta en hebras o en procesos y cómo se empaqueta, respectivamente. Todo lo relacionado con la infraestructura también debe ser considerado desde el punto de vista de despliegue en vez de desde el funcional.

  Por ejemplo, puede hacerse referencia al uso de colas de mensajes como conectores

pero el agente concreto que proporciona las colas debe ser parte del punto de vista de despliegue y no del funcional.

Notación

Para este punto de vista podemos usar distintas técnicas de representación de modelos:

- Diagrama de componentes UML.- La mayor ventaja de usar UML es que es ampliamente conocido y flexible. El diagrama principal para el punto de vista funcional es el diagrama de componentes, que muestra los elementos del sistema, las interfaces y las conexiones entre los elementos.

  EJEMPLO: Sistema de vigilancia de la temperatura

  La Figura 2.29 muestra la estructura funcional de un sistema de vigilancia de temperatura usando UML. El sistema está formado por dos componentes internos: el subsistema que obtiene la temperatura (Variable Capture), con la interfaz VariableReporting, y el subsistema que dispara la alarma (Alarm Initiator), con la interfaz LimitCondition. El primero de ellos interactúa con un sistema externo que muestra la temperatura (Temperature Monitor) y que invoca a la interfaz VariableReporting, de la cual se da más información: es una RPC en XML que usa el protocolo HTTP (por tanto se usa de forma asíncrona, de forma más parecida a las interfaces de tipo mensajería que a las RPCs) y que permite un máximo de 10 invocaciones simultáneas.

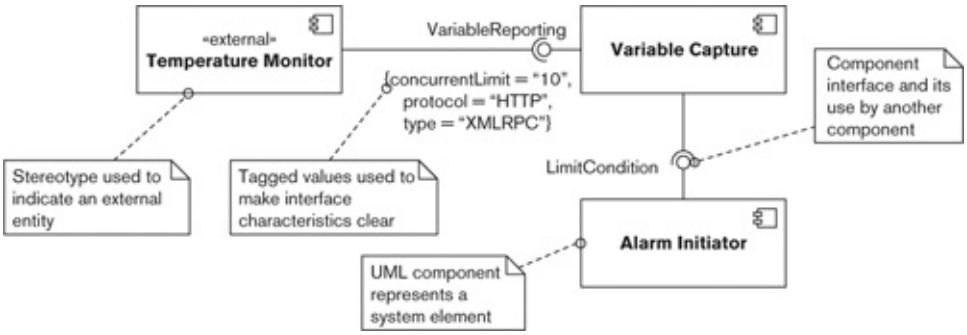


Figura 2.29: Ejemplo de una Estructura Funcional en UML. [Fuente: (Rozanski, 2011, Cap. 17)]

Además del estereotipo <<external>> ya visto cuando se trató el punto de vista contextual, otro estereotipo usado para el punto de vista funcional es el <<infrastructure>>,

para indicar los elementos de la infraestructura del sistema que tenga una función diferenciada.

En el diagrama se ha optado por la representación de la interfaz mediante un pequeño icono chupete (lollipot) que por una clase con el estereotipo <<interface>>. Para dar más información sobre una interfaz, podemos usar valores etiquetados que nos digan de qué tipo es, el protocolo usado o el numero de conexiones concurrentes permitidas.

Los conectores entre las interfaces se pueden representar como dependencias UML y flujos de información, como se describe en el siguiente ejemplo.

EJEMPLO: Tienda Web dentro de un entorno software ya existente en la empresa

La Figura 2.30 muestra un diagrama de componentes UML para describir la estructura funcional de un sistema sencillo. El sistema proporciona una tienda Web (Web Shop) que usan los clientes para compras a partir de un catálogo en línea y que será parte de un entorno software ya existente en la empresa.
El modelo muestra que el sistema interactúa con cinco entidades externas: los navegadores web de los tres tipos de usuario más importantes (personal de servicio al cliente, clientes y administrados del catálogo de productos), y dos sistemas externos (sistema de reparto e inventario). El sistema en sí está compuesto por cinco componentes principales, unidos por distintos tipos de conectores (incluyendo HTML sobre HTTP, mensajería por pulicación/subscripción y una interfaz externa LU 6.2 (desarollada por IBM).
Los clientes hacen el pedido a través de la tienda Web (Web Shop), que interactúa a su vez con el gestor de pedidos (Order Processor), el catálogo de productos (Product Catalog) y el sistema de información del cliente (Customer Information System). Los administradores del catálogo mantienen el catálogo a través de su interfaz basada en la Web (Catalog Management Interface) y los que forman parte del equipo de servicio al cliente mantienen la información del cliente (Customer Information System) mediante un programa cliente de interfaz dedicada (Customer Care Interface). Cuando se desea consultar el stock de un producto determinado, el catálogo de productos (Product Catalog) obtiene la información del inventario (Stock Inventory), que es un sistema que ya existe.
Hay también cierta información en cuanto a la naturaleza de las interacciones entre componentes. Sabemos que pueden acceder al sistema simultáneamente hasta 1000 clientes, 80 personas del servicio al cliente y 15 administradores del catálogo. Además, la interacción entre el catálogo de productos y el inventario tiene lugar mediante un protocolo específico (que seguramente era un tecnología que ya existía por ser el inventario parte del software actual de la empresa). Podemos asumir según el ejemplo que la comunicación entre componentes que no ha sido descrita se lleva a cabo con algún procedimiento RPC (que será definido en algún otro lugar de la DA de forma clara).
Uno de los aspectos interesantes que se reflejan en este diagrama es la cantidad de deducciones no obvias que se pueden hacer a partir de él. Las responsabilidades de los componentes no está claras, tampoco los detalles de las interfaces ni los detalles de la interacción entre los distintos componentes. Esto nos hace entender la necesidad de añadir descripciones de texto y otros diagramas que modelen el sistema de forma complementaria, por ejemplo, las interacciones entre componentes puede mostrarse modelando escenarios del sistema.

- Otras notaciones formales de diseño.- UML no es la única notación de diseño bien definida adecuada para el desarrollo de software. Existen otras notaciones estructuradas más antiguas (como Yourdon, Jackson System Development y Object Modeling Technique de James Rumbaugh) que se han aplicado con éxito a problemas de desarrollo de software durante muchos años. El problema es que tienden a ser bastante débiles para describir los conceptos (como elementos a gran escala, interfaces, opciones de implementación, etc.) que son importantes para los arquitectos. Los métodos más antiguos tampoco se enseñan ni se usan ampliamente en la actualidad, por lo que puede ser difícil encontrar soporte de herramientas y carecen de la familiaridad general que tiene UML para la mayoría de las personas

- Lenguajes de descripción de arquitectura (Architecture description languages, ADLs): los lenguajes que admiten directamente los conceptos que interesan a los arquitectos de software se conocen generalmente como ADL. Se ha creado una gran cantidad de ADLs (incluidos Unicon, Wright, xADL, Darwin, C2 y AADL). El gran atractivo de los ADLs es que brindan soporte nativo para algunas de las cosas que necesitamos capturar y razonar en nuestros diseños arquitectónicos (como componentes y conectores). Sin embargo, casi todos los ADL se han desarrollado en el entorno de investigación y tienden a sufrir una serie de inconvenientes prácticos, incluida la falta de familiaridad de las partes interesadas en ellos, un alcance relativamente limitado (a menudo sólo permite que se representen los componentes y conectores), y una inevitable falta de soporte en herramientas maduras. Por estas razones, no se puede recomendar ninguno para uso cotidiano

- Diagramas de "cuadros y líneas": muchos arquitectos utilizan un diagrama de estructura funcional basado en notación personalizada de cuadros y líneas. Dicho diagrama debe mostrar sólo los elementos funcionales y sus interfaces y debe vincular los elementos a las interfaces que usan con un dispositivo gráfico claro (generalmente una flecha, posiblemente con alguna anotación) que indica el uso de un conector. Al igual que con cualquier notación personalizada, se debe definir claramente el significado de la notación para evitar confusiones. La Figura 2.31 presenta el diagrama de "cuadros y líneas" equivalente al diagrama UML de la Figura 2.30. A veces es conveniente usar estos diagramas para vender las propiedades y beneficios del sistema a algunas partes interesadas y dejar los detalles técnicos para un diagrama UML que usen sólo algunas partes interesadas

- Bocetos: Permiten crear una sensación menos formal, introduciendo una notación ad hoc para representar cada uno de los aspectos de la vista que sean significativos para el sistema. A menudo es necesario para comunicar de manera efectiva aspectos esenciales de la vista a las partes interesadas no técnicas. El problema es que pueden conducir a una visión mal definida y confusión entre las partes interesadas. Al igual

que con el diagrama de "cuadros y líneas", puede evitarse esto utilizando un boceto como añadido a una notación de vista más formal (como UML) y utilizando diferentes anotaciones para diferentes grupos de partes interesadas

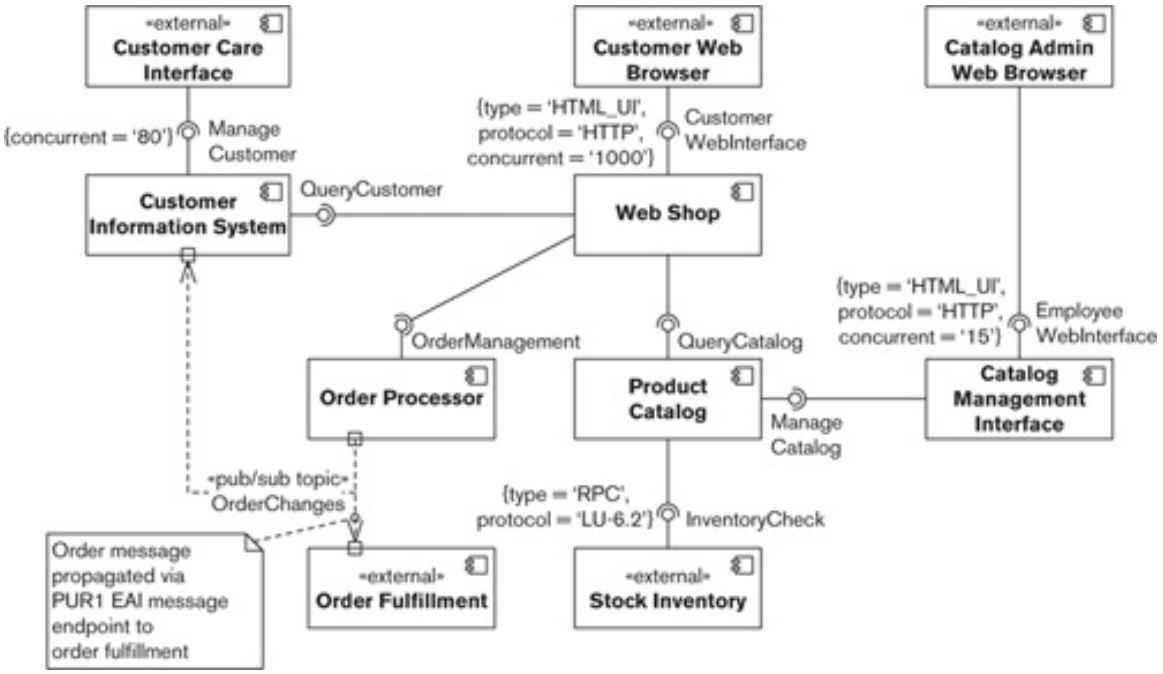


Figura 2.30: Ejemplo de diagrama de componentes UML para modelar una tienda Web. [Fuente: (Rozanski, 2011, Cap. 17)]

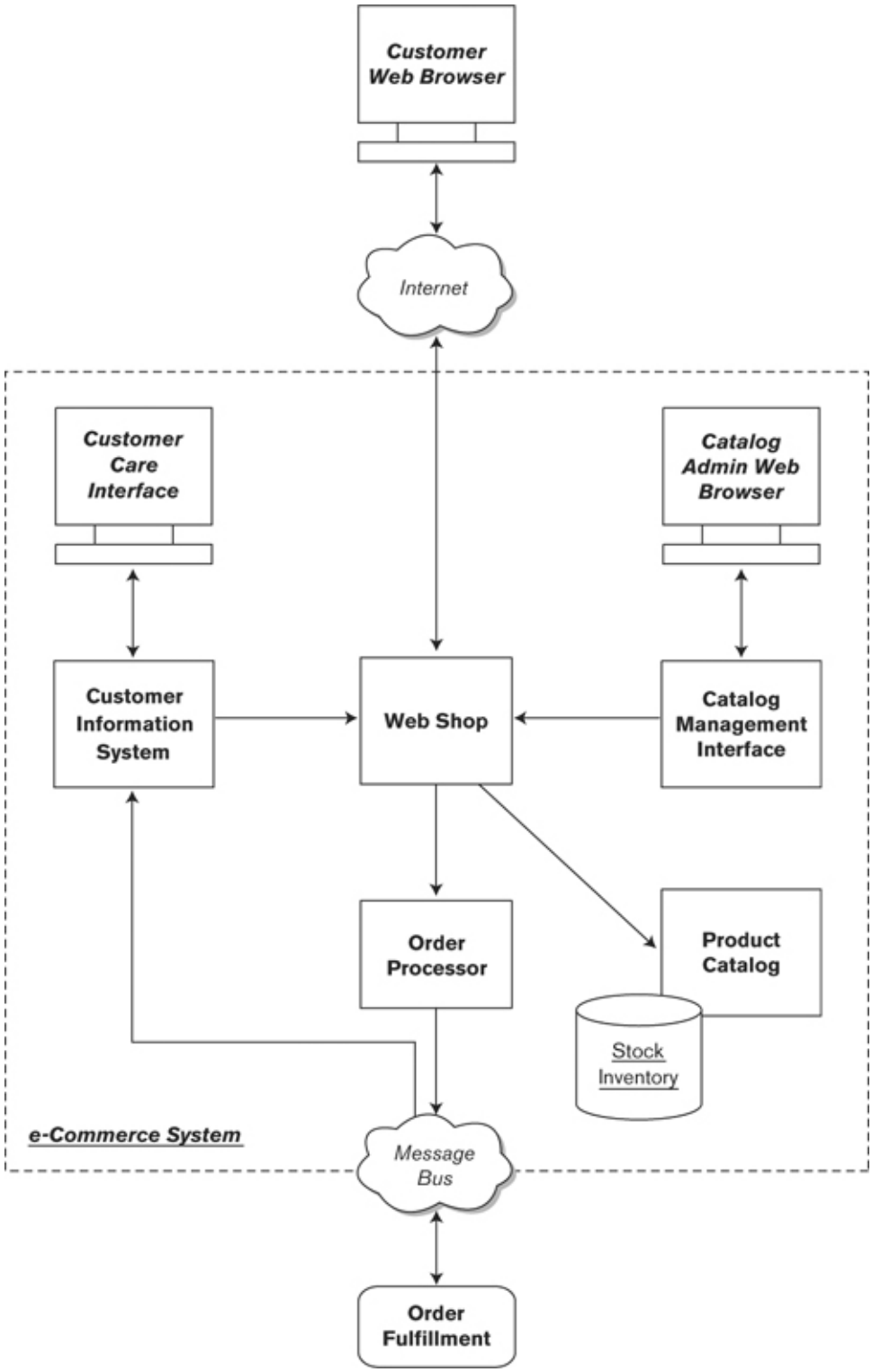


Figura 2.31: Ejemplo de diagrama de "cuadros y líneas" equivalente al diagrama de componentes UML para modelar una tienda Web que aparece en la Figura 2.30. Se ha usado una notación propia: las interfaces de usuario externas se representan por un icono que simboliza el monitor del ordenador, y los sistemas externos de respaldo se representan por rectángulos con esquinas redondeadas. Los almacenes de datos se representan por un icono que simboliza un tambor de discos y las interfaces funcionales (Internet, el bus de mensajes –Message Bus–) se representan mediante una nube. El ámbito del sistema incluye a todos los elementos dentro del rectángulo de línea discontinua. [Fuente: (Rozanski, 2011, Cap. 17)]

Actividades

- Identificar los elementos.- Hay muchas formas de hacerlo, y el método correcto a utilizar depende del tipo de sistema y el enfoque de desarrollo de software que estemos utilizando. Por ejemplo, enfoques procedurales clásicos, orientación a objetos o enfoques basados en componentes, influyen en la identificación de componentes de diferentes maneras. De todos modos, en todos ellos pueden seguirse los siguientes pasos:

  1. Trabajar a partir de los requisitos funcionales, derivando responsabilidades clave a nivel del sistema
  2. Identificar los elementos funcionales que desempeñarán esas responsabilidades
  3. Evaluar el conjunto identificado con los criterios de diseño deseables
  4. Repetir la secuencia para refinar la estructura funcional hasta que juzgue que es sólida, usando uno o más métodos de refinamiento entre los siguientes:
     - Generalización: identifica algunas responsabilidades comunes en una serie de elementos e introduce una cantidad de elementos más generales que pueden reutilizarse en todo el sistema para realizar estas tareas. La generalización es particularmente importante como parte de una arquitectura de línea de producto o empresa más grande para permitir la reutilización de activos de software en una serie de productos o sistemas similares
     - Descomposición: divide un elemento grande y complejo en varios subelementos más pequeños. Para sistemas grandes, a menudo necesitamos dividir los elementos funcionales de nivel superior en elementos a nivel de subsistema más manejables para permitir su diseño y construcción
     - Composición: reemplazar varios elementos funcionales pequeños con un elemento más grande que incluye todas las funciones de los más pequeños. La composición se usa típicamente cuando se ha identificado un gran número de elementos funcionales pequeños pero similares. En tales casos, a menudo tiene sentido desde una perspectiva arquitectónica reemplazar los elementos más pequeños con un solo elemento grande que pueda factorizar la parte común entre los más pequeños y reducir la cantidad de interacciones que requiere el sistema
     - Replicación: repetir un elemento del sistema o una pieza de procesamiento. Un ejemplo es la validación de datos, donde se identifica un elemento de validación para los datos entrantes y luego los replica en varias interfaces externas del sistema. La replicación puede traer beneficios de rendimien-

to, pero se debe tener cuidado para mantener consistentes los componentes replicados[9]

Si se está utilizando un estilo arquitectónico para guiar el proceso de diseño, el proceso es ligeramente diferente porque implicará crear una instanciación del estilo de modo que las responsabilidades a nivel del sistema se asignen a elementos del estilo. Esta actividad está estrechamente relacionada con el siguiente paso: asignar responsabilidades a los elementos.

- Asignar responsabilidades a los elementos.- Una vez que se han identificado los elementos candidatos, la siguiente actividad consiste en asignarles responsabilidades claras, es decir, la información gestionada por el elemento, los servicios que ofrece a otras partes del sistema y las actividades que inicia, si es que no se ha completado ya en el paso anterior

[9]Nota de la profesora: Salvo en sistemas que tengan restricciones muy cortas de tiempo de desarrollo y que no vayan a tener ningún mantenimiento, este método de refinamiento está muy desaconsejado porque el aumento de rendimiento se hace a costa de un enorme riesgo de inconsistencia en los componentes duplicados.

EJEMPLO: Responsabilidades de dos elementos de la aplicación de comercio electónico de venta al por menor antes descrita

- Elemento: Tienda Web
  - Proporcionar a los clientes una interfaz HTML accesible desde el navegador Web
  - Gestionar todos los estados relacionados con la sesión de la interfaz del cliente
  - Interaccionar con otras partes del sistema para permitir que el cliente pueda ver el catálogo de productos y el stock, comprar y consultar información sobre su cuenta
- Elemento: Sistema de Información al Cliente
  - Gestionar toda la información persistente relacionada con los clientes del sistema
  - Proporcionar una interfaz sólo-consulta al cliente que le permita consultar la información a la que tenga acceso
  - Proporcionar una interfaz programable de gestión de información que pueda usarse para crear aplicaciones de gestión de información de clientes
  - Proporcionar una interfaz dirigida por eventos de manejo de mensajes que acepte las líneas de detalle de los pedidos hechos por los clientes y los cambios de estado realizados sobre dichos pedidos

- Diseñar las interfaces.- Debe incluir las operaciones que ofrece la interfaz; la entrada, salidas, condiciones previas y efectos de cada operación; y la naturaleza de la interfaz (mensajería, RPC, servicio web, etc.)

  El enfoque de "Diseño por Contrato" desarrollado para OO por Bertrand Meyer es aquí también muy apropiado, de forma que las interfaces se especifican a través de "contratos" que utilizan condiciones previas, condiciones posteriores e invariantes para definir con precisión el comportamiento y las relaciones de cada operación.

  La notación apropiada para la definición de interfaces depende del tipo de interfaz y de quién necesita comprender esta información (considerando factores como la tecnología de implementación probable, los antecedentes del equipo de desarrollo y los tipos de interfaces que deben describirse). Las siguientes son algunas notaciones comunes de definición de interfaz:

- Lenguajes de programación.- Se pueden definir directamente mediante el uso de un lenguaje de programación para especificar la signatura de la operación junto con texto o código para la semántica de la operación. Este enfoque es simple pero lo vincula con el estilo, los supuestos y las limitaciones del lenguaje de programación particular, lo cuál no es lo ideal, especialmente si se utilizan múltiples tecnologías. Es el enfoque más apropiado para librerías software u otros ejemplos donde se usa un solo lenguaje de programación para implementar todo el sistema
- Lenguajes de definición de interfaz (Interface Deginition Laguage, IDL).- Desarrollados para admitir tecnología de sistemas distribuidos de lenguaje mixto. Existen IDLs para CORBA, para .NET, o para Web Services Description Laguage (WSDL), una tecnología XML para describir servicios web. Independientes de la tecnología de implementación, tienden a permitir instalaciones más simples que los lenguajes de programación,resultando más adecuados para definir interfaces arquitectónicas. Siempre que las partes interesadas puedan leerlos (o que se les enseñe a leerlos), los IDLs constituyen una buena opción para definir signaturas de operaciones
- Enfoques orientados a datos.- Permite describir las interfaces únicamente en términos de mensajes que se intercambian. Como ejemplos se incluyen interfaces a las que se accede a través de sistemas de mensajería e interfaces definidas en términos de intercambio estructurado de documentos (por ejemplo, interfaces orientadas a documentos y basadas en servicios web con mensajes definidos mediante el esquema XML). Recomendado en interfaces basadas en eventos que se definen en términos del intercambio de eventos comerciales en lugar de la invocación de operaciones

Cualquiera que sea la notación que se use para describir las interfaces, no hay que olvidar que una interfaz es mucho más que una simple definición de cómo llamar a las operaciones. Desafortunadamente, ninguno de los enfoques que hemos descrito ofrece facilidades para definir la semántica de la interfaz, por lo que una definición clara de una interfaz implicará el uso de lenguaje natural o lenguajes especializados como Object Restraint Language (OCL) para lograr esto. Una definición de interfaz debe comunicar con precisión las condiciones previas y posteriores de cada operación y cómo se deben combinar las operaciones para realizar una función útil (preferiblemente con ejemplos). Cualquier reducción en la descripción puede causar problemas importantes al utilizarlas.

- Diseñar los conectores.- Los elementos de un sistema deben relacionarse entre sí para lograr los objetivos del sistema, de forma que al identificar las responsabilidades de un elemento aparece la necesidad de interactuar con otros para poder llevarlas a cabo.

Las interacciones tienen lugar a través de conectores de algún tipo que vinculan los elementos que delegan responsabilidades con las interfaces ofrecidas por los elementos en los que éstas se delegan. A veces, el tipo de conector requerido es evidente (como una simple llamada a procedimiento), mientras que en otros casos deberá pensar detenidamente si necesita comunicación sincrónica o asincrónica, la resiliencia requerida del conector, la latencia aceptable de interacciones a través de él, etc. Para cada vía de comunicación entre elementos requerido por la arquitectura, debe agregarse un conector al modelo para admitirla (ya sea RPC, mensajería, transferencia de archivos u otros mecanismos)

- Comprobar la trazabilidad funcional.- La especificación de requisitos funcionales define una serie de funciones que el sistema debe proporcionar. Es necesario comprobar la trazabilidad para asegurarse de que la estructura funcional propuesta cumpla con todos los requisitos funcionales, de forma que no falten en la estructura ninguna función o haya alguna incompleta. Para hacerlo de manera formal se puede usar una tabla que cruce los requisitos funcionales con los elementos del modelo funcional encargados de implementarlas

- Recorrer escenarios comunes.- Puede ser extremadamente valioso y esclarecedor recorrer con algunas partes interesadas, los escenarios comunes de uso del sistema, utilizando el punto de vista funcional para mostrar cómo se comportará el sistema en cada caso. Será especialmente útil hacerlo con miembros del equipo de pruebas, el equipo de desarrollo y los administradores del sistema. Debe explicarse cómo interactuarían los elementos del sistema para implementar el escenario, de forma que puedan identificarse debilidades arquitectónicas, malentendidos, o elementos omitidos

- Analizar las interacciones.- Es útil analizar la estructura elegida desde el punto de vista del número de interacciones entre elementos tomadas durante escenarios de procesamiento comunes, para tratar de reducir las interacciones a un conjunto mínimo sin distorsionar la coherencia de los componentes funcionales (bajar el acoplamiento manteniendo la cohesión). Por lo general, es un paso importante hacia un sistema eficiente y confiable. A veces habrá que compensar la reducción de interacciones entre elementos para que no resulte en una estructura distorsionada con redundancia indeseable o partición de elementos inapropiada

- Analizar la flexibilidad.- Un sistema que funcione siempre está bajo presión para cambiar, por lo que la arquitectura debe ser flexible. Dentro de ella, la estructura funcional es uno de los principales factores que afectan a la flexibilidad de los sistemas de información. Es útil analizar algunos escenarios de "qué pasaría si" que revelen el impacto de posibles cambios futuros en el sistema. Un problema común en este punto es que los cambios implicados para aumentar la flexibilidad pueden entrar en conflicto

con los sugeridos por el análisis de las interacciones. Por lo tanto, es importante encontrar el equilibrio adecuado entre complejidad y flexibilidad

Problemas y errores comunes

Se incluyen los siguientes:

- Interfaces pobremente definidas.- Muchas veces se definen muy bien los elementos, sus reponsabilidades y relaciones entre sí pero se descuidan las interfaces, lo que puede llegar a malas interpretaciones en los equipos de desarrollo, que pueden terminar en un sistema poco fiable. Las interfaces deben incluir las operaciones, su semántica y ejemplos cuando sea posible

- Responsabilidades mal entendidas.- Deben definirse formalmente todas las resposabilidades de cada elemento para que los equipos de desarrollo no olviden ninguna o las repitan en más de un elemento

- Infraestructura modelada como si fueran elementos funcionales.- La infraestructura se debe modelar en el punto de vista de despliegue; hacerlo en el funcional, salvo que sea importante para entender la vista, le añade complejidad innecesaria

- Vista sobrecargada.- El punto de vista funcional es la piedra angular de la DA, la vista central, pero no el conjunto de todas las vistas (por ejemplo, incluyendo elementos de los puntos de vista de despliegue, concurrente o de información. Si se optara por una vista compuesta de todos los puntos de vista, habría que especificarlo claramente

  EJEMPLO: Diagrama de una vista funcional sobrecargada

  La Figura 2.32 muestra un ejemplo de vista funcional sobrecargada. Además de UML se ha añadido notación ad hoc: las líneas discontinuas en los elementos dentro del cuadro del nodo servidor ("Server Node"). Es difícil entender su significado sin preguntar al arquitecto o leer documentación adicional: representan procesos independientes del Sistema Operativo, lo cual no debería formar parte del punto de vista funcional. Tampoco debería estar en este punto de vista nada relacionado con el despliegue ni los distintos ordenadores que participan o el software externo (no deberían por tanto aparecer los elementos del nodo servidor. Por otro lado, los elementos del nodo servidor son ambiguos y los propios desarrolladores o personal del equipo de prueba necesitarán más detalle (en los puntos de vista de despliegue, de información, concurrente, etc.) para poder implementar y probar la solución.

- Diagramas sin definiciones de elementos.- A menudo se representa la estructura funcional (relaciones entre los elementos) sin dar una definición clara y precisa de cada elemento que todas las partes interesadas puedan entender

- Dificultades para reconciliar las necesidades de distintas partes interesadas.- A menudo no todas las partes interesadas pueden entender un mismo diagrama. Puede proporcionarse un diagrama específico para las partes técnicas y otro para las no técnicas (las partes interesadas en el negocio), que puede ser una simplificación del primero y una notación más sencilla

- Nivel de detalle inadecuado.- Hay que evitar diagramas tan detallados que bajen a capas de diseño demasiado detallado (por ejemplo, bajar a más de un tercer nivel de detalle posiblemente sea demasiado), pero también evitar ser tan genéricos que no se entiendan las ideas ni tampoco garantizar criterios de calidad (por ejemplo, quedarse en un primer nivel de detalle en sistemas muy grandes)

- Evitar un elemento "divino".- A menudo en el diseño OO existe el riesgo de distribuir mal las responsabilidades entre los objetos y dejar casi todo a un sólo objeto muy grande, el gestor. Este problema, llamado el problema del "objeto divino", tiene un homólogo al hacer una DA, de forma que un solo elemento acumule la mayor parte de la resposabilidad del sistema, haciéndolo demasiado complejo y difícil de entender. Además caerá sobre él casi toda la resposabilidad de asegurar los distintos criterios de calidad (tales como rendimiento, escalabilidad y fiabilidad) y más difícil de garantizarlos dada su complejidad

  EJEMPLO: Diagrama de componentes UML con un elemento divino

  La Figura 2.33 muestra un diagrama de componentes UML que puede adolecer del problema del elemento divino. En concreto, el Gestor de Clientes (Customer Management) parece tener una enorme responsabilidad, con el resto de elementos interactuando con él.

- Demasiadas dependencias.- El problema opuesto al del elemento divino es el teneder diagramas con muchas dependencias entre elementos, con una aspecto como de arañas luchando por el control. Cuando las interacciones son muy complejas, los sistemas son más difíciles de diseñar y construir, y pueden dar problemas de funcionamiento poco eficaz y baja capacidad de mantenimiento

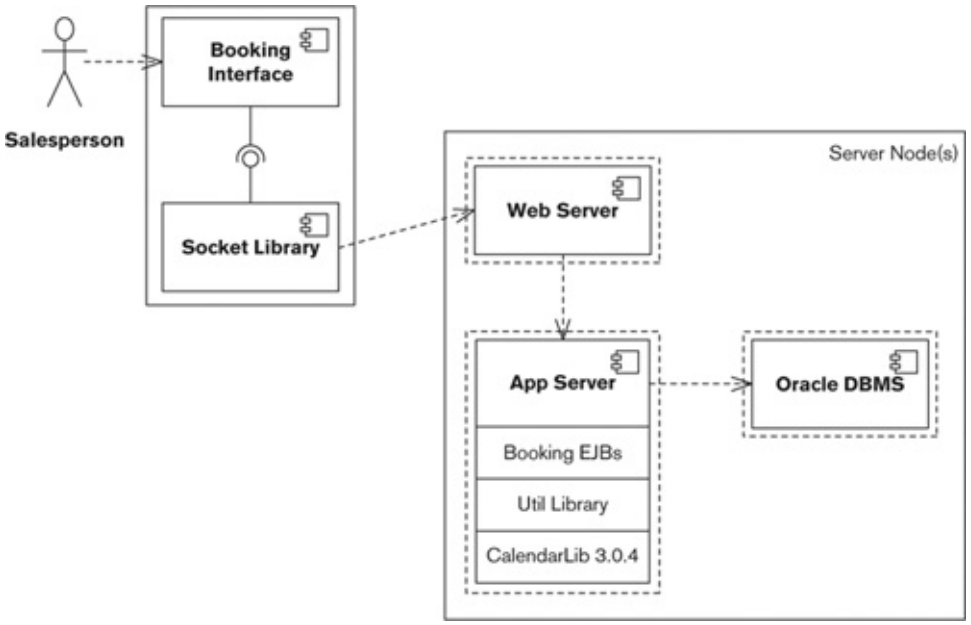


Figura 2.32: Ejemplo de diagrama de una vista funcional sobrecargada. [Fuente: (Rozanski, 2011, Cap. 17)]

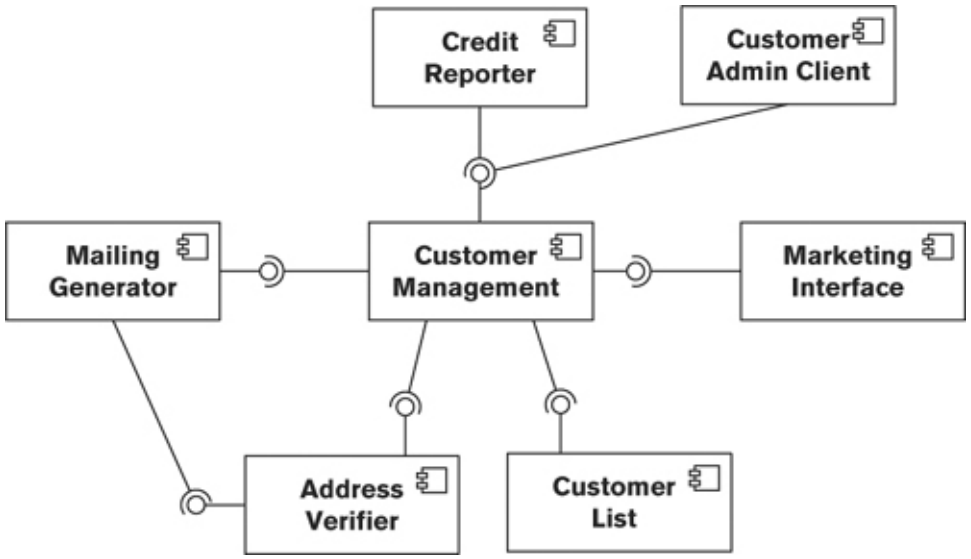


Figura 2.33: Ejemplo de diagrama funcional que adolece del problema del elemento divino. [Fuente: (Rozanski, 2011, Cap. 17)]

Lista de comprobación

Se propone la siguiente:

- ¿Tienes menos de 15 a 20 elementos de nivel superior?
- ¿Tienen todos los elementos un nombre, responsabilidades claras e interfaces claramente definidas?
- ¿Tienen lugar todas las interacciones de elementos a través de interfaces y conectores bien definidos que unan las interfaces?
- ¿Exhiben los elementos un nivel apropiado de cohesión?
- ¿Exhiben los elementos un nivel apropiado de acoplamiento?
- ¿Has identificado los escenarios de uso importantes y los has utilizado para validar la estructura funcional del sistema?
- ¿Has verificado la cobertura funcional de la arquitectura para asegurarte de que cumple con los requisitos funcionales?
- ¿Has definido y documentado un conjunto apropiado de principios de diseño arquitectónico y tu arquitectura cumple con estos principios?
- ¿Has considerado cómo es probable que la arquitectura haga frente a posibles escenarios de cambio en el futuro?
- ¿La presentación de la vista tiene en cuenta las preocupaciones y capacidades de todos los grupos de partes interesadas? ¿Actuará la vista como un vehículo de comunicación eficaz para todos estos grupos?

Adición de "perspectivas" al estándar P1471 basado en puntos de vista

Rozansky hace caer en la cuenta que, a diferencia de los puntos de vista y de sus vistas, que desglosan el sistema en partes en cierta medida independientes, hay una serie de inquietudes que son comunes a muchas de las vistas y deberían reflejarse en la DA como comunes a ellas. Se refieren más bien a requisitos no funcionales, y entre ellos, a criterios de calidad. Aunque algunos los han catalogado como puntos de vista, él destaca que no es oportuno hacerlo así por ser requisitos transversales a distintos puntos de vista.

La seguridad: un ejemplo de inquietud "transversal" Para explicarlo pone un ejemplo relacionado con el criterio de seguridad. En su opinión, este criterio de calidad no suele abordarse como debe durante el ciclo de vida del proyecto. Una razón es que no es fácil garantizar un nivel apropiado de seguridad y se tiende a declinar en otros la responsabilidad, cuando es la empresa la que está obligada por ley a garantizar la seguridad en el software que desarrolle.

La razón por la que se trata de una inquietud transversal y por lo tanto no puede considerarse como un punto de vista más es que es una inquietud para distintos puntos de vista:

- Punto de vista funcional.- El sistema de identificar y autentificar a sus usuarios, y defender el sistema frente a ataques externos.
- Punto de vista informacional.- El sistema de controlar distintos tipos de acceso a la información (leer, borrar, actualizar, insertar) y puede requerir aplicar criterios de seguridad con distintos niveles de granularidad.
- Punto de vista operacional.- El sistema debe mantener y distribuir información secreta (palabras clave) según los sistemas de seguridad que en ese momento estén en uso.

Posiblemente también en los puntos de vista de desarrollo, de concurrencia y de despliegue haya algunos aspectos que se vean afectados por la necesidad de garantizar la seguridad.

Perspectivas arquitectónicas Rozansky define el concepto de "perspectiva arquitectónica" de la siguiente forma:

DEFINICIÓN: Perspectiva arquitectónica

Colección de actividades, tácticas y guías arquitectónicas usadas para asegurar que el sistema cumple con un conjunto relacionado de criterios de calidad que deban ser considerados de forma transversal, es decir, por un conjunto diverso de vistas arquitectónicas.

No se trata de nada nuevo, las perspectivas son los requisitos no funcionales, pero lo importante de la propuesta es que se ofrece un mecanismo para que la arquitectura sistematice la implementación necesaria para su cumplimiento.

Para implementar las perspectivas, se añade el concepto de "táctica arquitectónica", desarrollado por investigadores software del Carnegie Mellon Software Engineering Institute (SEI), y algo modificado por el enfoque de Rozansky, que la define como un enfoque provado que puede usarse para conseguir una propiedad (criterio) de calidad concreto.

Así, para garantizar un criterio de calidad no sólo se debe revisar su cumplimiento en los modelos arquitectónicos sino que se deben seleccionar y probar algunas "tácticas arquitectónicas" que solucionen casos específicos que la arquitectura no pueda abordar.

Se podría hacer una equivalencia entre tácticas y patrones, de forma que las tácticas arquitectónicas son para los requisitos no funcionales como los patrones de diseño son para los requisitos funcionales. Pero las tácticas son bastante más generales, son sólo guías generales.

Un ejemplo de táctica arquitectónica para garantizar el rendimiento completo del sistema podría ser definir diferentes prioridades de procesamiento para las distintas partes de la carga de trabajo del sistema y gestionarlas mediante un planificador de procesos basado en prioridades, algo como lo que hacen los sistemas operativos para la gestión de procesos no interactivos de cálculo intensivo.

La perspectiva proporciona un marco para guiar y formalizar el cumplimiento de un criterio de calidad, de forma que desde éste se tenga en cuenta cómo garantizarlo en las distintas vistas arquitectónicas a las que les afecte, tomando las decisiones arquitectónicas que sean necesarias.

Se destacan de ella tres características principales:

- Es un *almacén de conocimiento* de gran utilidad, ayudando a revisar de forma rápida cómo cada modelo arquitectónico garantiza un determinado criterio de calidad sin necesitar usar documentos más detallados
- Es una *guía* eficiente si se trabaja en un dominio de aplicación desconocido y sus inquietudes, problemas y soluciones
- Es un *soporte a la memoria* cuando se conoce el dominio, para garantizar que no se nos olvida nada importante

Las perspectivas deben aplicarse en las etapas más iniciales del diseño de la arquitectura, aunque se haga de manera informal, para que se puedan garantizar mejor los criterios de calidad sin que sea al alto coste de rediseñar por no haberlas tenido en cuenta a tiempo.

Plantilla para definir una perspectiva arquitectónica

Se debe ser sistemático a la hora de definir una perspectiva. La propuesta de Rozansky consiste en la consideración de los siguientes aspectos para describir perspectiva en un proyecto concreto (Tabla 2.7).

Perspectivas más importantes

También se proponen algunas perspectivas más importantes que deben abordarse en sistemas de información de gran tamaño (Tabla 2.8).

En la realidad, no todas las perspectivas pueden aplicarse a todos los puntos de vista. La Figura 2.34 muestra un ejemplo de aplicación de varias perspectivas a los distintos puntos

Detalle	Descripción
Aplicabilidad	Qué puntos de vista son más probables que se vean afectados. Por ejemplo, si se trata de la capacidad de evolución, el punto de vista funcional estará más afectado que el operacional
Inquietudes	Define los criterios de calidad que son abordados por la perspectiva
Actividades	Define los pasos para aplicar las perspectivas a los distintos puntos de vista y de ahí a las vistas, de forma que se adopten decisiones en el diseño arquitectónico de las vistas para garantizar desde cada vista el criterio de calidad de la perspectiva
Tácticas arquitectónicas	Descripción de las tácticas más importantes para garantizar los criterios de calidad
Problemas y errores	Menciona errores comunes y pautas para reconocerlos y evitarlos
Lista de comprobación	Un listado con todas las cuestiones que no deben olvidársenos para que lo podamos repasar (inquietudes, tácticas, trampas, errores comunes)
Documentación adicional	La descripción de la perspectiva debe ser concisa. Los detalles adicionales pueden describirse en otro lugar y ser referidos aquí

Tabla 2.7: Plantilla para describir la aplicación de una perspectiva en un proyecto concreto. [Fuente: (Rozanski, 2011)]

de vista, mediante una tabla 2x2 donde sólo se colorea una celda si corresponde a una perspectiva y un punto de vista sobre el que se puede aplicar dicha perspectiva.

Como criterio para seleccionar los puntos de vista que abordarán cada perspectiva se aconseja:

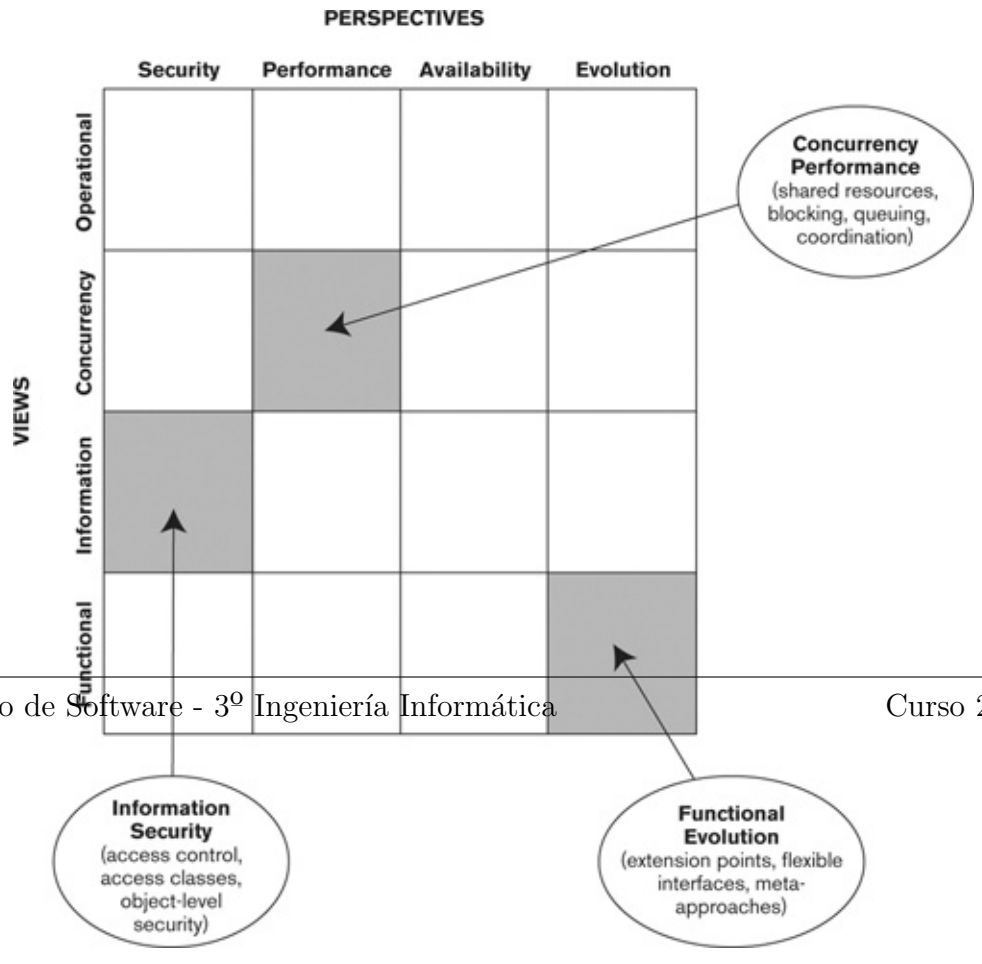
- Tener en cuenta las inquietudes de las partes interesadas en el sistema
- La importancia relativa de los distintos criterios de calidad en el sistema concreto
- La propia experiencia y el juicio profesional

Elaborar esta tabla puede ser útil para el arquitecto software como paso previo a la aplicación de las perspectivas.

Para estas celdas afectadas, deberá darse información (representada con texto en círculos) con el detalle de cómo traducir el criterio de calidad en el contexto del punto de vista concreto (criterios más específicos, inquietudes, guías para el diseño, etc.).

Perspectiva	Descripción
Accesible	Capacidad del sistema de poder ser usado por personas con alguna discapacidad
Deslocalizable	Capacidad del sistema para superar problemas debidos a la localización concreta de sus partes y la distancia entre ellos
Disponible y resiliente	Asegura que el sistema esté disponible cuando sea necesitado y que se restablezca después de posibles fallos
Eficiente y escalable	Cumpliendo con los requerimientos de rendimiento y manejando de forma satisfactoria los incrementos en la carga de trabajo
Evolutivo	Garantiza que el sistema pueda responder a cambios posibles en el mismo
Fácil de usar	Capacidad de facilitar a los usuarios un trabajo eficaz
Factible	Capacidad para que el sistema pueda ser diseñado, construido, lanzado y utilizado dentro de las restricciones de rescursos impuestas, tales como personas, presupuesto, tiempo y materiales
Internacionalizable	Capacidad del sistema de funcionar de la misma forma, independientemente de idiomas, países o grupos culturales
Regulable	Habilidad del sistema para cumplir con las leyes internacionales, regulación quasi-legal (normas éticas), políticas de la empresa y otras reglas y estándares
Seguro	Garantiza el acceso controlado a los recursos del sistema

Tabla 2.8: Perspectivas más importantes en proyectos de gran envergadura [Fuente: (Rozanski, 2011)]



Descripción detallada de una perspectiva: la perspectiva de seguridad

DEFINICIÓN: Datos sensibles

Se llama "datos sensibles" a aquella información que debe ser protegida frente al acceso no autorizado.

Hoy en día la seguridad es clave en la mayor parte de sistema de información, ya que pueden ser sistemas distribuidos y usar internet u otras redes y necesitan garantizar que sólo puedan acceder a cada recurso los que estén autorizados.

Los especialistas en seguridad utilizan una terminología específica:

- 'Principales".- Los actores o subsistemas software con acceso identificado
- "Recursos".- Partes del sistema que tienen acceso controlado
- "Políticas".- Define el acceso legítimo a cada recurso
- "Mecanismos de seguridad".- Usado por los principales del sistema para obtener el acceso que necesitan

La Figura 2.35 refleja las relaciones entre estos elementos.

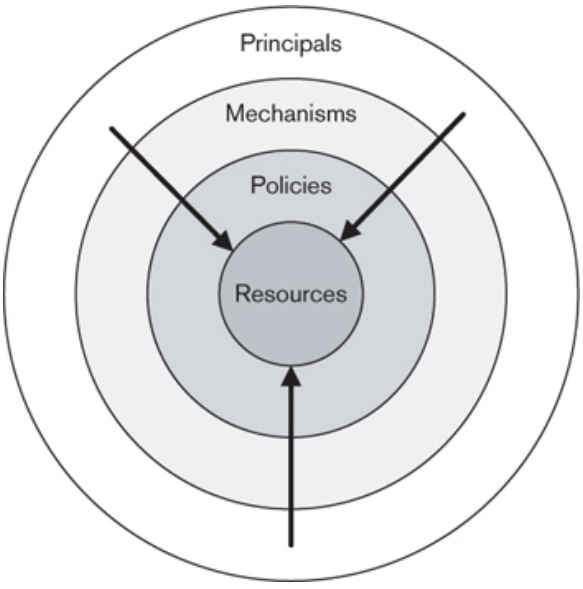


Figura 2.35: Relación entre principales, políticas, mecanismos y recursos. [Fuente: (Rozanski, 2011)]

Estos elementos son muy diferentes según el tipo de sistema. En todo caso hay que tener en cuenta que la seguridad no es un estado binario sino un proceso de gestión de riesgos que equilibra los riesgos probables de seguridad con los costes que requiere garantizarlas.

Se describe la perspectiva siguiendo la plantilla propuesta en la Tabla 2.7:

Aplicabilidad

Se trata de ver cómo afecta la seguridad a los distintos puntos de vista. La Tabla 2.9 responde teniendo en cuenta los siete puntos de vista considerados hasta ahora.

Punto de vista	Aplicabilidad
Contextual	Permite identificar claramente las conexiones externas al sistema y cómo deben protegerse de un uso malicioso para evitar la vulnerabilidad del sistema, incluso cambiando dichas conexiones
Funcional	Permite identificar los elementos funcionales del sistema que deben protegerse y en consecuencia pueden tener que implementar políticas concretas de seguridad
Informacional	Debe identificar los datos del sistema que deben ser protegidos, y los modelos de datos modificados para implementar los diseños concretos de seguridad, como por ejemplo, dividiéndolos según su sensibilidad
Concurrente	El diseño de seguridad puede señalar la necesidad de aislar distintas piezas del sistema en diferentes elementos de ejecución (como hebras), afectando así a la estructura de la concurrencia del sistema
De desarrollo	Identifica guías y restricciones que los desarrolladores software necesitan conocer para garantizar la política de seguridad.
De despliegue	El diseño de seguridad puede afectar de forma considerable a este punto de vista, incluyendo hardware o software seguro o añadiendo algunas decisiones de despliegue para prevenir los riesgo de seguridad
Operacional	También es muy importante para garantizar las políticas de seguridad la forma en la que se usa el sistema una vez en explotación. Este punto de vista debe dejar muy claras las asunciones y responsabilidades de seguridad de forma que se reflejen en los procesos operacionales

Tabla 2.9: Aplicación de la perspectiva de seguridad a los distintos puntos de vista. [Fuente: (Rozanski, 2011)]

CRITERIO DE CALIDAD: Confidencialidad

La confidencialidad (considerada aquí como un aspecto dentro de la perspectiva general de seguridad), se define como la capacidad de garantizar el acceso a un recurso sólo a los que están legitimamente autorizados.

CRITERIO DE CALIDAD: Integridad

La integridad (considerada aquí como un aspecto dentro de la perspectiva general de seguridad), se define como la capacidad de garantizar que la información no pueda ser modificada de forma no detectable.

CRITERIO DE CALIDAD: Disponibilidad

La disponibilidad (considerada aquí como un aspecto dentro de la perspectiva general de seguridad, más allá de su acepción en el nivel operativo), se define como la capacidad de garantizar que ningún ataque del sistema bloquee el acceso al sistema o a una parte del mismo.

CRITERIO DE CALIDAD: Trazabilidad

La trazabilidad (considerada aquí como un aspecto dentro de la perspectiva general de seguridad), se define como la capacidad de garantizar que se pueda conocer sin ambigüedad la secuencia de pasos que fueron dados para llevar a cabo una acción, retrotayendo los pasos hasta llegar al principal que la inició.

Inquietudes

Se trata de describir cómo abordar las inquietudes de seguridad, empezando por definir las partes vulnerables (recursos) y terminando por describir los mecanismos concretos de seguridad que serán utilizados. Para un enfoque sistemático, se propone describir cada uno de los siguientes aspectos:

- Recursos.- Las partes que requieren acceso controlado puede requerirlo bien por contener información importante o bien por realizar operaciones sensibles. Se deben establecer los mecanismos de seguridad apropiados (de acceso o de ejecución) para protegerlas.

- Principales.- Es necesario que el sistema permita el acceso identificado de cada principal para poderles dar los "privilegios de acceso" o autorizaciones adecuadas.

- Políticas.- Deben detallar el tipo de acceso de cada tipo de principal (empleados, administradores, gestores, etc.) para cada tipo de información. Además, deben describir cómo se controlará la ejecución de operaciones del sistema sensibles. También definirá restricciones de integridad, tales como reglas y comprobaciones a aplicar en los almacenes de datos y en la protección de documentos frente a cambios no autorizados.
- Amenazas.- Se deben identificar para añadir los mecanismos de seguridad necesarios para afrontarlas, equilibrando asimismo la garantía de la seguridad y la usabilidad del sistema (el índice de riesgo de cada amenaza concreta dependerá del tipo de sistema).
- Confidencialidad.- Si la información no sale del sistema, suele garantizarse mediante el control de acceso. Si hay transmisión a otros sistemas suele garantizarse mediante encriptación.
- Integridad.- Suele garantizarse mediante firma encriptada (firma electrónica o huella digital).
- Disponibilidad.- Para garantizarla hay que diseñar el sistema pensando en los posibles riesgos que pueden derivarse como consecuencia de ataques piratas.

  [

  label=accounting]DEFINICIÓN: Contabilización (accounting) “Las partes implicadas en la transacción (emisor – receptor) tienen que aceptar los datos de la relación creada.” Minubeinformatica.com

  [

  label=norepudio]DEFINICIÓN: No repudio “Informes externos que los sistemas generan sobre eventos, actuaciones, usos, etc de si mismos. Siguiendo y analizando la información generada en estos informes se puede localizar por donde se ha producido un error o ataque e intentar resolverlo. Son los llamados logs del sistema.” Minubeinformatica.com

- Trazabilidad.- El mecanismo más común para garantizarla en sistemas centralizados es el de contabilización (del inglés, accounting) (Definición **??**). En sistemas distribuidos se suelen usar mensajes encriptados como prueba de que lo envía un principal concreto (no repudio) (Definición **??**).
- Detección y recuperación.- Puede ir más allá de simples acciones tecnológicas, teniendo que involucrar también a personas y establecer procedimientos de acción

- Mecanismos de seguridad.- Para establecerlos se debe contar con expertos en seguridad que conozcan mejor las tecnologías disponibles de seguridad y cómo combinarlas para garantizar la seguridad del sistema

Sin embargo, en este apartado es suficiente con describir los recursos y pasar después a decribirlos con más detalle y clasificarlos, continuando con la descripción de las políticas de seguridad, así como las amenazas y los mecanismos de seguridad siguiendo la secuencia de actividades propuesta por Rozansky (ver Figura (Rozanski, 2011)).

Actividades

La Figura 2.36 muestra una propuesta de la secuencia de actividades que deben llevarse a cabo para aplicar la perspectiva de seguridad.

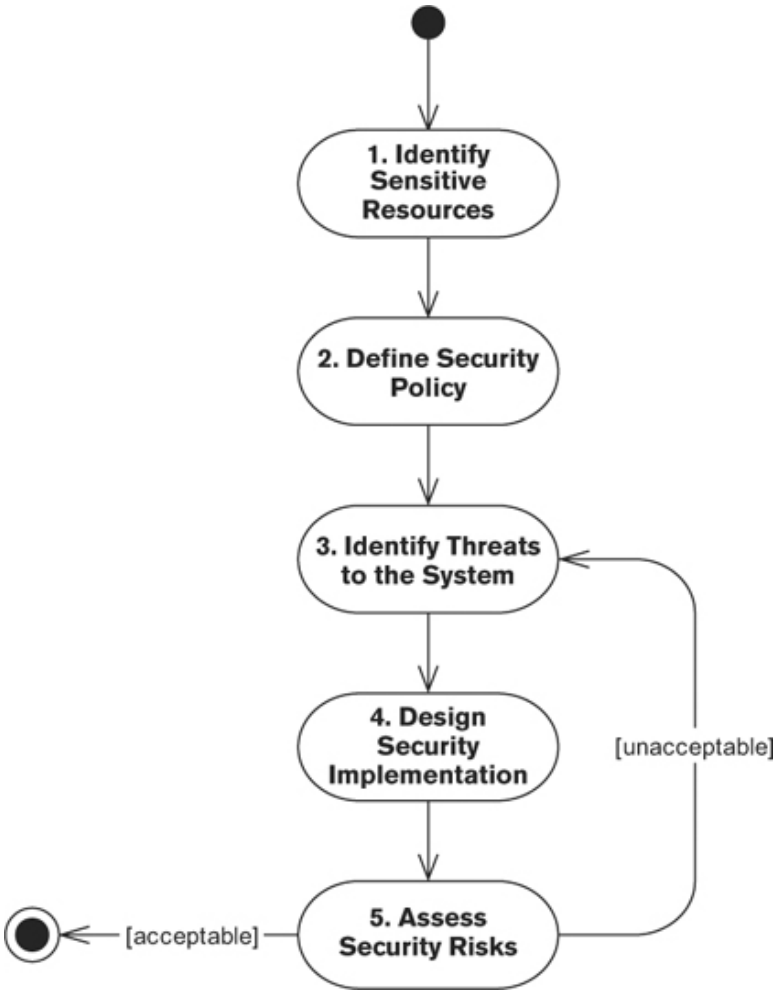


Figura 2.36: Secuencia de actividades a realizar para aplicar la perspectiva de seguridad. [Fuente: (Rozanski, 2011)]

Se empieza por una clasificación de los recursos y principales (paso 1) y se sigue con la descripción de las políticas de seguridad a adoptar (paso 2).

Luego se sigue de forma cíclica con la identificación de las amenazas del sistema (paso 3), la descripción de los mecanismos de seguridad para proteger al sistema frente a las amenazas, de forma que se garantice la confidencialidad, integridad y disponibilidad del sistema (paso 4) y la evaluación de los riesgos de seguridad (paso 5), de forma que se volverá al paso 3 si se considera que aún son demasiado altos.

Para cada paso, se definen a su vez un conjunto de sub-actividades, y las notaciones usadas más usuales, tal y como aparece en la Tabla 2.10.

Actividad principal	Sub-actividades	Notación
1. Identificar recursos sensibles	Clasificar recursos sensibles	Tablas/texto Diagramas usados en p.v. funcional o informativo
2. Definir la política de seguridad	Definir tipos o clases de principales según rol y tipos de accesos. Definir tipos o clases de recursos según uniformidad en el control del acceso. Definir conjuntos de control de acceso (operaciones que pueden realizarse en cada tipo de recurso y tipos de principales que pueden hacerlas). Identificar las operaciones sensibles del sistema y definir los tipos de principales que tienen acceso a ellas. Identificar los requisitos de integridad (situaciones del sistema donde los recursos puedan ser modificados (información) o ejecutados (operaciones) sin permiso.	Tablas
3. Identificar las amenazas al sistema	Identificar las amenazas Caracterizar las amenazas	Tablas/texto "Árbol de ataques"[10]
4. Diseñar la implementación de la seguridad	Diseñar una forma de mitigar las amenazas. Diseñar un enfoque de detección y recuperación. Considerar la tecnología. Integrar la tecnología.	Notación de cada vista afectada (Opc.) diagrama UML como en p.v. funcional para mostrar modelo de seguridad global
5. Establecer los riesgos de seguridad	Establecer los riesgos	Tablas

Tabla 2.10: Sub-actividades y notación para cada actividad para abordar la perspectiva de seguridad. [Fuente: (Rozanski, 2011)]

Paso 1: Identificación de los recursos

Para el ejemplo del sistema de comercio electrónico y partiendo del punto de vista funcional, los tipos de recursos que pueden identificarse a partir de los diagramas, en función del tipo de acceso son:

- Gestión de las cuentas de usuarios, y de los clientes en especial
- Gestión del catálogo de productos, y de los precios en especial
- Gestión de las compras

Partiendo del punto de vista de la información, los tipos de recursos que pueden identificarse según su sensibilidad son:

- Registros de cuentas de clientes
- Catálogo de productos
- Precios de los productos
- Registro de operaciones realizadas por clientes (compras)

Los principales pueden también clasificarse (en roles) en función de los recursos a los que pueden acceder:

Principales

- Cliente
- Gestor de catálogo
- Vendedor (quien prepara un pedido)
- Repartidor (quien entrega un pedido)
- Administrador
- Superusuario

La Figura 2.37 muestra la Tabla 25-2 Rozanski (2011) en donse se identifican los recursos en la Tienda Web.

Resource	Sensitivity	Owner	Access Control
Customer account records	Personal information of value for identity theft or invasion of privacy	Customer Care Group	No direct data access
Descriptive product catalog entries	Defines what is for sale and its description; if maliciously changed, could harm the business	Stock Management Group	No direct data access
Pricing product catalog entries	Defines pricing for catalog items; if maliciously or accidentally modified, could harm the business or allow fraud	Pricing Team in Stock Management Group	No direct data access
Business operations on customer account records	Needs to be controlled to protect data access and integrity	Customer Care Group	Access to individual record or all records by authenticated principal
Descriptive catalog operations	Needs to be controlled to protect data access and integrity	Stock Management Group	Access to catalog modification operations by authenticated principal
Pricing catalog modification operations	Needs to be controlled to protect data access and integrity	Pricing Team	Access to price modification operations by authenticated principal, with accountability of changes
...	...	...	...

Figura 2.37: Tabla 25-2 donde se identifican algunos de los recursos sensibles en el sistema de la Tienda Web. [Fuente:(Rozanski, 2011, Tabla 25-2)]

Paso 2: Definición de la política de seguridad

La Figura 2.38 muestra la Tabla 25-3 Rozanski (2011) en donde se identifican las políticas de control de acceso en la Tienda Web.

	User Account Records	Product Catalog Records	Pricing Records	User Account Operations	Product Catalog Operations	Price Change Operations
Data administrator	Full with audit	Full with audit	Full with audit	All with approval and audit	All with audit	All with approval from a product price administrator
Catalog clerk	None	None	None	All	Read-only operations	None
Catalog manager	None	None	None	Read-only operations with audit	All	All with audit
Product price administrator	None	None	None	None	Read-only operations	All with audit
Customer care clerk	None	None	None	All with audit	Read-only operations	None
Registered customer	None	None	None	All on own record	Read-only operations	None
Unknown Website user	None	None	None	None	Read-only operations	None

Figura 2.38: Tabla 25-3 donde se muestra parte del resultado de definir la política de seguridad para el sistema de Tienda Web. [Fuente:(Rozanski, 2011, Tabla 25-3)]

En este paso se han descrito 7 tipos distintos de principales según el control de acceso y 5 tipos distintos de recursos sensibles.

Paso 3: Identificación de las amenazas del sistema
En este paso hay que construir un "modelo de amenazas", que parte de cada recurso identificado y considera cada posible amenaza, el impacto (consecuencias) en el sistema y la probabilidad de que ocurra.

Para elaborarlo, debe responderse a las siguientes preguntas:

- ¿Quién es probable que quiera saltarse la política de seguridad?
- ¿Qué motivación tiene el atacante para atacar el sistema?
- ¿Cómo tratará de saltarse la política de seguridad?

- ¿Cuáles son las principales características del atacante (sofisticación, compromiso, recursos, etc.?

- ¿Cuáles son las consecuencias de que se salte la política de esta forma?

Hay que considerar ataques tanto externos como internos, cómo afectan los ataques al entorno (por ejemplo al proveedor de servidores –hosting–, o a un entorno de computación en la nube). También se debe contar con los recursos de seguridad implementados por los proveedores de servicios o por el entorno en general.

La notación más común es la de texto y tablas pero también pueden usarse árboles de ataques: resultados de amenazas por categorías y probabilidad de que ocurran, puestos en forma jerárquica.

Se pueden considerar dos subactividades para este paso:

- Identificar las amenazas, considerando la sensibilidad de los recursos y los posibles tipos de atacantes

- Caracterizar las amenazas, identificando los recursos objetivo, las consecuencias y la probabilidad del ataque

El cuadro **??** muestra el ejemplo de un árbol de ataque para la amenaza de obtener los detalles de la tarjeta de crédito de un cliente en un sistema de comercio electrónico. En esta notación, suele empezarse con un paso previo: identificar los objetivos de cada atacante. Se definirá un árbol por cada objetivo. Un objetivo incluye el conjunto de amenazas al sistema que el atacante puede intentar para lograrlo.

El árbol de ataque es un método de representación del modelo de amenaza de un sistema. Se basa en la técnica de los árboles de fallos en el diseño de sistemas críticos.

Un árbol de ataque representa los posibles ataques que el sistema puede sufrir para que un atacante logre un objetivo particular. La raíz del árbol es el objetivo que el atacante está tratando de lograr, y las ramas del árbol clasifican los diferentes tipos de ataques que el intruso podría intentar para lograr el objetivo.

Pueden representarse gráficamente (como una estructura de árbol con nodos y enlaces) o textualmente usando títulos numerados anidados.

[

label=arbolAtaque]EJEMPLO: Árbol de ataque
Este es un posible árbol de ataque con el objetivo de extraer los datos de la tarjeta de crédito del cliente de un sitio web de comercio electrónico:
Objetivo: obtener los datos de la tarjeta de crédito del cliente.

1. Extraer detalles de la base de datos del sistema
   - *a*) Acceder a la base de datos directamente
     1) Descifrar / adivinar contraseñas de bases de datos
     2) Romper / adivinar las contraseñas del sistema operativo que permiten evitar la seguridad de la base de datos
     3) Explotar una vulnerabilidad conocida en el software de la base de datos
   - *b*) Acceder a los detalles a través de un miembro del personal de administración de la base de datos
     1) Sobornar a un administrador de base de datos (DBA)
     2) Realizar ingeniería social por teléfono / correo electrónico para engañar al DBA para que revele detalles
2. Extraer detalles de la interfaz web
   - *a*) Configurar un sitio web ficticio y enviar por correo electrónico a los usuarios la URL para engañarlos para que introduzcan los detalles de la tarjeta de crédito
   - *b*) Descifrar / adivinar contraseñas para cuentas de usuario y extraer detalles de la interfaz web del usuario
   - *c*) Enviar a los usuarios un programa troyano por correo electrónico para grabar pulsaciones de teclas / interceptar el tráfico web
   - *d*) Atacar el servidor de nombres de dominio para secuestrar el nombre de dominio y use el ataque de sitio ficticio de 2.1
   - *e*) Atacar el software del servidor del sitio directamente para tratar de encontrar lagunas en su seguridad o configuración o para aprovechar una vulnerabilidad conocida en el software
3. Encuentrar detalles fuera del sistema
   - *a*) Realizar ingeniería social por teléfono / correo electrónico para que el personal de servicio al cliente revele los detalles de la tarjeta
   - *b*) Dirigir un ataque de ingeniería social a los usuarios mediante el uso de detalles públicos del sitio para hacer contacto (ver también 2.1)

Paso 4: Diseño de los mecanismos de seguridad a implementar
Hay que considerar tecnologías específicas para garantizar la política de seguridad defendiendo al sistema de los ataques identificados. Algunos ejemplos son: cortafuegos, comunicación SSL, criptografía, etc.

Este proceso de diseño da como resultado una serie de decisiones que deben incorporarse en la arquitectura, afectando probablemente a los puntos de vista funcional, de información, de implementación y operacional.

El cuadro **??** muestra un ejemplo de medidas de seguridad a adoptar para las amenazas identificadas en el árbol de ataque del ejemplo **??**, en un sistema de comercio electrónico.

Hay que notar que este paso no suele incluir una descripción de tecnologías a usar sino más bien en explicar cómo se garantiza la seguridad.

[

label=disSeguridad]EJEMPLO: Diseño de medidas de seguridad a adoptar

- Aislar las máquinas de la base de datos de la red pública utilizando la tecnología de firewall de red
- Aislar las partes sensibles a la seguridad del sistema de la red pública utilizando la tecnología de firewall de red
- Analizar las rutas en el sistema para verificar si hay vulnerabilidades posibles
- Organizar pruebas de penetración para ver si los expertos pueden encontrar formas de entrar en el sistema
- Identificar una estrategia de detección de intrusos que permita reconocer las violaciones de seguridad
- Capacitar al personal de administración y servicio al cliente (de hecho, probablemente todo el personal) para evitar ataques de ingeniería social y acatar estrictos procedimientos de protección de la privacidad de la información del cliente
- Diseñar el sitio Eeb para que una cantidad mínima de información del usuario (idealmente, ninguna) sea visible públicamente
- Diseñar el sitio Web para que la información confidencial (por ejemplo, números de tarjetas de crédito) nunca se muestren en su totalidad (por ejemplo, muestre solo los últimos cuatro dígitos para permitir que los usuarios legítimos identifiquen sus tarjetas en las listas)
- Aplicar de forma rutinaria actualizaciones software relacionadas con la seguridad a todo el software de terceros utilizado en el sistema
- Revisar el código del sistema para detectar vulnerabilidades de seguridad utilizando herramientas de análisis e inspección experta
- Recordar constantemente a los usuarios las precauciones de seguridad que deben tomar (por ejemplo, no revelar las contraseñas a nadie, incluido su personal; verificar las URL antes de ingresar información, etc.)

Paso 5: Evaluación de los riesgos de seguridad
No hay que olvidar que la seguridad nunca puede garantizarse de forma total. En este paso hay que hacer balance de riesgo/coste según las medidas adoptadas. Si el resultado es demasiado costoso o los riesgos no tratados demasiado altos, debe volverse al paso 3 de indetificación de las amenazas del sistema.

La Figura 2.39 muestra la Tabla 25-4 Rozanski (2011) en donde se muestra como ejemplo la evaluación de tres de los riesgos identificados en el sistema de la Tienda Web.

Risk	Estimated Cost	Estimated Likelihood	Notional Cost
Attacker gains direct database access	$8,000,000	0.2%	$16,000
Web-site flaw allows free orders to be placed and fulfilled	$800,000	4.0%	$32,000
Social-engineering attack on a customer service representative results in hijacking of customer accounts	$4,000,000	1.5%	$60,000
. . .	. . .	. . .	. . .

Figura 2.39: Tabla 25-4 donde se muestra la evaluación de tres de los riesgos identificados en el sistema de la Tienda Web. [Fuente:(Rozanski, 2011, Tabla 25-4)]

Tácticas arquitectónicas

Se proporciona la siguiente lista de posibles tácticas:

1. Aplicar los principios de seguridad que gocen de reconocimiento. Algunos ejemplos son:
   - Asignar el mínimo nivel de privilegios posible
   - Asegurar los accesos más débiles
   - Defensa en profundidad (esquema de protección por capas)
   - Separar y modularizar por responsabilidades
   - Hacer diseños de seguridad simples, que faciliten su análisis
   - No apoyarse en la oscuridad, sino asumir que los posibles atacantes puedan conocer el sistema tan bien como nosotros
   - Usar por defecto los criterios de seguridad (para claves, permisos de acceso, etc.)
   - Seguridad en fallos, no sólo cuando el sistema funciona bien
   - Asumir que las entidades externas no son confiables

    - Auditar/monitorizar eventos sensibles

2. Autenticar a los principales (personas, ordenadores, subsistemas, etc.
3. Autorizar el acceso
4. Asegurar la privacidad de la información
5. Asegurar la integridad de la información
6. Asegurar la trazabilidad
7. Proteger la disponibilidad
8. Integrar las tecnologías de seguridad
9. Proporcionar administración de la seguridad (forma parte del punto de vista operacional
10. Usar la infraestructura de seguridad de terceras partes

Problemas y errores comunes

Se destacan los siguientes:

- Políticas de seguridad complejas
- Tecnologías de seguridad disponibles no probadas
- Sistema no diseñado para responder frente a fallos
- Ausencia de facilidades de administración de seguridad
- Enfoque dirigido por la tecnología
- Fallo al considerar las fuentes para medir tiempos
- Exceso de confianza en la tecnología (nunca estamos seguros 100 %)
- Requisitos o modelos de seguridad no bien definidos
- Considerar la seguridad para el final, sin tenerla en cuenta desde el inicio de la DA
- Ignorar las amenazas internas
- Asumir que el cliente es seguro

- Embeber la seguridad en el código de la aplicación, de forma que el modelo de seguridad queda implementado de forma diseminada en distintas partes del código de la aplicación
- Seguridad por piezas, en vez de considerarla y abordarla como un todo
- Uso de tecnologías de seguridad ad hoc

Lista de comprobación

Se proporcionan dos listas, una para capturar todos los requerimientos de seguridad:

- ¿Has identificado los recursos sensibles del sistema?
- ¿Has identificado los conjuntos de principales que necesitan acceder a los recursos?
- ¿Has identificado las necesidades que tiene el sistema de garantizar la integridad de la información?
- ¿Has identificado las necesidades de disponibilidad del sistema?
- ¿Has establecido una política de seguridad para definir las necesidades de seguridad del sistema, junto con los principales y los permisos de acceso que tiene cada uno a cada recurso, y cuándo debe comprobarse la integridad de la información?
- ¿Es la política de seguridad lo más simple posible?
- ¿Has recorrido un modelo formal de amenazas para identificar los riesgos de seguridad del sistema?
- ¿Has considerado tanto las amenazas externas como las internas al sistema?
- ¿Has considerado cómo el entorno de despliegue del sistema alterará las amenazas del sistema?
- ¿Has recorrido escenarios de ejemplos junto con las partes interesadas en el sistema de forma que puedan entender la política de seguridad planificada y los riesgos del sistema?
- ¿Has revisado los requisitos de seguridad con expertos externos en seguridad?

La otra lista proporcionada es directamente aplicable en la DA:

- ¿Has abordado cada amenaza del modelo de amenazas con la profundidad necesaria?

- ¿Has usado lo más posible las tecnologías de seguridad de terceras partes?
- ¿Has realizado un diseño global integrado de la solución dada para garantizar la seguridad?
- ¿Has considerado los principios estándares de seguridad a la hora de diseñar la infraestructura de seguridad?
- ¿Es tu infraestructura de seguridad lo más simple posible?
- ¿Has definido una forma de identificar brechas de seguridad y de recuperar el sistema frente a ellas?
- ¿Has aplicado los resultados de la perspectiva de seguridad a todos los puntos de vista afectados?
- ¿Han revisado tu diseño de seguridad expertos externos en seguridad?

Descripción detallada de otra perspectiva: la perspectiva de evolución

El software, como su nombre indica ("soft"), es "flexible", de forma que las partes interesadas esperan que un sistema software pueda evolucionar muy rápidamente. Por otro lado, a menudo hay que hacer cambios como consecuencia de requisitos mal comprendidos que el usuario constata cuando empieza a usar el sistema, o el cambio comercial rápido. El enfoque iterativo permite a los usuarios comenzar a usar algunas partes de él mucho antes de estar terminado, proporcionando retroalimentación temprana a los desarrolladores. El problema es la presión constante mientras no se termina por completo, de cambiar el comportamiento del sistema, con la consiguiente necesidad en algunos casos de cambiar su arquitectura.

El software es fácil de cambiar sólo si el cambio se considera explícitamente durante su desarrollo.

DEFINICIÓN: Evolución

Conjunto de todos los posibles tipos de cambios que un sistema puede experimentar durante su vida útil.

La perspectiva Evolución aborda las preocupaciones relacionadas con el manejo de la evolución durante la vida útil de un sistema y, por lo tanto, es relevante para la mayoría de los sistemas de información a gran escala debido a la cantidad de cambios que la mayoría de los sistemas necesitan manejar.

CRITERIO DE CALIDAD: Capacidad de evolución (sistema evolutivo)

La capacidad de evolución de un sistema es su flexibilidad ante cambios inevitables en fase de explotación, con un aumento moderado de los costes de desarrollo necesarios para proporcionar tal flexibilidad

La perspectiva pretende que se cumpla con el criterio de calidad "capacidad de evolución". Tal y como hicimos con la perspectiva de seguridad, se describirá esta perspectiva siguiendo la plantilla propuesta en la Tabla 2.7.

Aplicabilidad

Se trata de ver cómo afecta la evolución a los distintos puntos de vista, considerando que afectará más a sistemas de mayor uso, tanto intensivo como extensivo. La Tabla 2.11 responde teniendo en cuenta los siete puntos de vista considerados hasta ahora.

Punto de vista	Aplicabilidad
Contextual	Puede necesitar mostrar entidades externas, interfaces o interacciones que formarán parte sólo en versiones futuras del sistema
Funcional	Debe reflejarse la evolución a nivel funcional si los requisitos de evolución son significativos
Informacional	Debe hacerse un modelo informacional flexible si se requiere evolución de la información o del entorno
Concurrente	Las necesidades evolutivas pueden condicionar el empaquetamiento de algún elemento en particular o alguna restricción en la estructura concurrente (v.g., que sea muy simple)
De desarrollo	Los requisitos evolutivos pueden tener ato impacto sobre el entorno que desarrollo que se necesita definir (v.g. forzando guías de portabilidad)
De despliegue	Generalmente la evolución no afecta a esta vista
Operacional	Generalmente la ecolución tiene poco impacto en esta vista

Tabla 2.11: Aplicación de la perspectiva de evolución a los distintos puntos de vista. [Fuente: (Rozanski, 2011, Cap. 28)]

Inquietudes

Se destacan las siguientes:

- Gestión de productos.- En las metodologías ágiles de desarrollo de software (Agile, Scrum, Extreme Programming (XP), etc.) se ha incorporado la idea de ver el software como producto futuro en el mercado, estudiando las necesidades del futuro cliente, así como las amenazas y las oportunidades del entorno donde se utilizará para realizar una hoja de ruta para planificar y supervisar su desarrollo. El papel de propietario del producto, a menudo lo desempeña el arquitecto, que debe trazar el rumbo futuro para el sistema que está desarrollando.

  Ya sea que se reconozca formalmente, como en las metodologías ágiles, o no, la gestión del producto es importante porque proporciona un contexto y una dirección para todos los cambios que ocurren en el sistema. Esto ayuda a que los cambios potenciales sean priorizados sistemáticamente y les permite ser considerados en el contexto de una hoja de ruta, para evitar el desarrollo de un conjunto de características incoherentes.

- Magnitud del cambio.- Cuando se realiza la DA de un sistema pensando que sobre él sólo se realizarán pequeños cambios, puede tener que enfrentarse a grandes costos si se tuviera que enfrentar en el futuro a cambios mucho mayores que incluso pueden llevar a la realización de un sistema completamente nuevo.

- Dimensiones del cambio.- Es necesario identificar las dimensiones de cambio requeridas, para poder acotar mejor la evolución del sistema. Entre ellas se proponen las siguientes:

  - Evolución funcional: incluye cualquier cambio en las funciones que proporciona el sistema, desde simples correcciones de defectos en un extremo de la escala hasta la adición o reemplazo de subsistemas completos en el otro
  - Evolución de la plataforma: muchos sistemas necesitan evolucionar en términos de las plataformas de software y hardware en las que se implementan. Esto puede incluir migración de plataformas (v.g. de servidores basados en Windows a servidores basados en Linux), así como la ampliación de las plataformas que el sistema puede usar (v.g., transferencia de productos a nuevas plataformas, ampliación de las plataformas cliente existentes basadas en PC a otras basadas en la Web o uso de apps para móviles)
  - Evolución de la integración: la mayoría de los sistemas están integrados en otros, de forma que los cambios de estos otros obligan a que evolucione el nuestro. No tiene que cambiar la funcionalidad pero sí la forma en como se integra con otros sistemas
  - Crecimiento en el uso: la mayoría de los sistemas exitosos experimentan un crecimiento en el uso durante su vida útil que puede deberse a muchos factores, como un aumento en el número o la complejidad de las transacciones, un aumento en

el número de usuarios o la necesidad de administrar y almacenar grandes cantidades de datos. Si el sistema proporciona un servicio de Internet exitoso, este crecimiento podría ser sustancial e impredecible

- Probabilidad del cambio.- Identificar los distintos tipos de cambios que podrían ser necesarios es algo fácil, pero evaluar la probabilidad de que los cambios sean realmente necesarios puede ser mucho más difícil. Puesto que ser flexible a los cambios agrega complejidad y gastos, es importante poder afinar estas probabilides
- Temporización del cambio.-Estimar el momento probable para realizar el cambio requerido también es una preocupación importante. Cuanto más lejos esté la necesidad de un cambio, menos probable es que el cambio sea realmente necesario en su forma actualmente identificada. Los requisitos para los cambios que no tienen fecha de entrega asociada pueden ser de menor prioridad que los cambios con fechas firmes a corto plazo adjuntas
- Cuándo pagar por el cambio.- Hay dos estrategias para planificar el cambio en nuestro sistema:

  1. Diseñar el sistema más flexible posible ahora para facilitar el cambio posterior. Esto se identifica mejor con el enfoque de metasistema, donde la estructura y las funciones de información del sistema se definen en tiempo de ejecución mediante datos de configuración
  2. Crear el sistema más simple posible para satisfacer las necesidades inmediatas y enfrentar el desafío de hacer cambios solo cuando sea absolutamente necesario. Esta es más la idea de las metodologías ágiles (por ejemplo los mantras de XP de "Haz es sistema más simple posible" y "No lo vas a necesitar", que captan la lección de que tratar de adivinar el futuro y construir el sistema más flexible posible es un negocio costoso, arriesgado y complejo)

  Una de las principales diferencias entre estas dos estrategias es cuándo se paga por el cambio. El desarrollo de sistemas altamente flexibles cuesta mucho más que el desarrollo de sistemas simples y rígidos, por lo que el costo del cambio se carga al principio del ciclo de vida del sistema si sigue la primera estrategia. La compensación es que espera que estos costos iniciales se paguen con cambios más baratos y rápidos más adelante. Desarrollar el sistema más simple posible cuesta menos por adelantado porque es más simple y rápido de entregar, pero cada cambio posterior probablemente costará más porque no tiene un mecanismo existente para implementarlo.

  Conseguir el equilibrio correcto entre estas dos posiciones extremas evita el desperdicio del esfuerzo inicial o los enormes costos de cambio posteriores y nos ayuda a encontrar una posición que minimice los costos generales de desarrollo.

- Cambios dirigidos por factores externos.- No todos los cambios están bajo nuestro control o los de las partes interesadas más inmediatas, algunos pueden ser impuestos por personas o grupos fuera de nuestra esfera de influencia, como por ejemplo un nuevo jefe del departamento TIC que cambie a una estrategia de "comprar en vez de construir".

  Los ejemplos de cambio impulsado externamente incluyen los siguientes:

  - El final de la vida útil de los componentes hardware o software que se planea usar como parte de la arquitectura. Si la empresa exige que los sistemas solo puedan ejecutarse en hardware y software admitidos por el proveedor, éstos deben poder proporcionarnos la hojas de ruta para sus productos que identifiquen cuándo es probable que termine su vida útil
  - Cambios en las interfaces con entidades externas, como pasar a un nuevo protocolo, formato de datos, contenido de datos o modelo de interacción
  - Cambios en la regulación externa, que pueden conducir a requisitos más estrictos para la continuidad del negocio, validación, retención de datos, auditoría o control
  - Cambio organizacional que puede conducir a diferentes prioridades, requisitos modificados o cambios en la población de usuarios y el perfil de transacciones

- Complejidad del desarrollo.- En casi todos los casos, el apoyo a la evolución aumenta la complejidad del diseño de un sistema, a veces en gran medida y también puede traer problemas relacionados con la fiabilidad del sistema y el tiempo requerido para entregar las primeras partes del sistema. En algunos casos, la complejidad puede incluso convertirse en un obstáculo para la evolución del sistema

- Preservación del conocimiento.- Una preocupación importante para cualquier sistema es cómo preservar el conocimiento requerido para realizar cambios significativos en el sistema a medida que pasa el tiempo, pues las personas se trasladan a otros proyectos, los recuerdos se desvanecen y los entornos técnicos disponibles cambian

- Fiabilidad del cambio.- Desde la corrección de errores más simple hasta la remodelación más compleja, cualquier cambio en el sistema puede tener un impacto negativo en el sistema implementado, por lo que es esencial contar con un conjunto de procesos y tecnologías para hacer que este proceso sea lo más fiable posible. Las pruebas automatizadas, los procesos repetibles y bien entendidos, los entornos de desarrollo estables y la gestión efectiva de la configuración son factores clave para abordar esta preocupación a medida que el sistema evoluciona

Actividades

Se proponen 4 actividades en esta perspectiva, tal y como se muestra en el diagrama de flujo de la Figura 2.40.

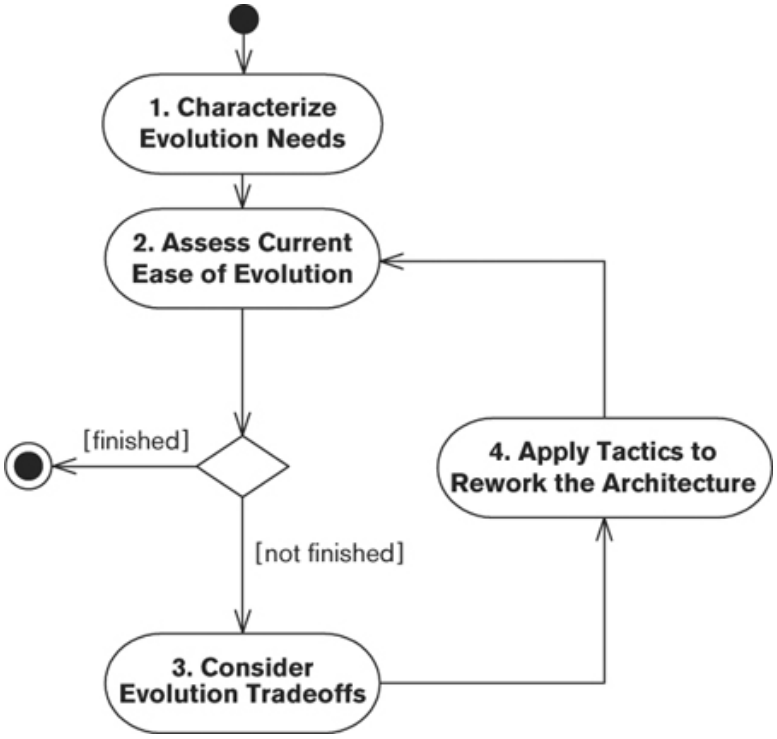


Figura 2.40: Curso de actividades a realizar para describir un sistema desde la perspectiva de evolución. [Fuente: (Rozanski, 2011, Cap. 28)]

Paso 1: Caracterizar las necesidades evolutivas

En este paso debemos volver a la especificación de requisitos y determinar qué es probable que tenga que cambiar con el tiempo. A menudo en la especificación de requisitos no aparece de forma explícita nada relacionado con la evolución del sistema. Debemos buscar en los requisitos algunos indicios de los siguientes tipos:

- Funciones diferidas: cualquier parte de los requisitos del sistema que defina explícitamente extensiones futuras, o funciones que no necesitan ser entregadas inicialmente
- Lagunas en los requisitos: probablemente requisitos de evolución disfrazados, que no pudieron definirse inicialmente debido a un análisis de requisitos incompleto

- Requisitos vagos o indefinidos: que indican que esta área del sistema no se comprende bien
- Requisitos abiertos: por ejemplo, términos tales como "similar a" o "incluyendo" o "etc.", en la definición de requisitos del sistema, sugieren que se requerirán extensiones similares a los casos especificados explícitamente

Para cada requisito evolutivo identificado, debemos describirlo utilizando la siguiente plantilla:

1. Tipo de cambio requerido: Se debe clasificar cada tipo de evolución en una de las dimensiones de cambio descritas anteriormente (funcional, de plataforma, de integración o de crecimiento)
2. Magnitud de cambio requerida: Ahora establecemos cuánto esfuerzo necesitará cada tipo de evolución. ¿Es sólo corrección de defectos, o se requerirán cambios a gran escala y de alto riesgo en el sistema? Una forma útil de presentar esto es el esfuerzo requerido en proporción al esfuerzo inicial de desarrollo del sistema
3. Probabilidad de cambio: Se trata de evaluar la probabilidad de que cada uno de los tipos de cambio identificados sea realmente necesario. Esto permite concentrarnos en aquéllos que tienen más probabilidades de ocurrir
4. Escala de tiempo de los cambios requeridos: ¿Se requieren los cambios en un calendario concreto e inmediato (una entrega por fases)? ¿O son necesidades vagas de posibles cambios futuros a hacer dependiendo de factores externos (como el crecimiento del sistema)?

A partir de esta descripción debemos priorizarlos según la importancia global y el tipo de evolución que las partes interesadas esperan del sistema. Una propuesta es ordenarlos dividiendo su magnitud relativa entre el número de meses que pensamos que quedan para que se necesite que el requisito esté implementado y enfocar el esfuerzo en los dos primeros.

Notación: Basta un enfoque de "texto y tablas".

Paso 2: Evaluar la facilidad para evolucionar en la actualidad

El objetivo de esta actividad es llegar a saber si los cambios requeridos pueden realizarse en el tiempo previsto a un coste razonable. Para ello debemos revisar los requisitos de evolución identificados (especialmente los de mayor prioridad) y pensar (sin identificar los detalles) cómo debería cambiar el sistema para cumplir con el requisito (magnitud, dificultad y riesgo del conjunto de cambios que habría que hacer).

Notación: Descripción textual.

Paso 3: Considerar las contrapartidas de la evolución

Debe considerarse si se debe hacer el esfuerzo de crear un sistema flexible durante el desarrollo inicial o si diferir este esfuerzo hasta que se requieran cambios en el sistema. La decisión depende en gran medida del tipo de sistema, la probabilidad de que los cambios sean realmente necesarios y el nivel de confianza que tiene en poder hacer cambios importantes de forma fácil cuando sea necesario, en lugar de durante el desarrollo inicial.

El resultado de este paso es describir la decisión tomada (cómo evolucionará el sistema y en qué punto se colocará el soporte para la evolución en el sistema).

Notación: Descripción textual.

Paso 4: Revisar la arquitectura

Considerando la mejor estrategia de evolución identificada, se deben cambiar las vistas (puntos de vista) afectadas.

Tácticas arquitectónicas

Aislar los cambios

Los cambios pequeños, por ejemplo los que afectan a un sólo módulo software, no suelen ser problemáticos. El problema aparece cuando sus efectos se propagan a través de varias partes diferentes del sistema simultáneamente.

El desafío arquitectónico consiste en diseñar una estructura de sistema sólida para que los cambios requeridos estén tan confinados como sea posible. Los siguientes principios generales de diseño pueden ayudarnos a localizar los efectos del cambio:

- Encapsulación: elementos fuertemente encapsulados con interfaces bien definidas y flexibles ayudan a aislar el cambio. Si las estructuras de datos internas de cada elemento no son visibles a sus clientes, los cambios internos a un elemento no se propagarán hacia fuera
- Separación de inquietudes (bajo acoplamiento): es más fácil aislar los cambios cuando las tareas funcionales se asignan a elementos concretos en vez de disgregarlas entre varios de ellos

- Cohesión funcional. también es más fácil aislar un cambio sin existe una alta cohesión, es decir, todas las funciones de un elemento están fuertemente relacionadas entre sí

- Mínima redundancia (punto único de definición): tanto datos como código y configuraciones deben definirse/implementarse una sola vez, para evitar tener que hacer cambios en varias partes diferentes del sistema, que, si no se hace bien, pueden llevar a inconsistencias

Crear interfaces extensibles

Los cambios en las interfaces son los que tienen mayor propagación y por tanto los más costosos. Por ejemplo, agregar un parámetro obligatorio a una función de uso frecuente implica cambiar cada parte de código que llame a esa función, recodificar y volver a probarla. Por tanto, vale la pena invertir en el diseño de cierto nivel de flexibilidad en las interfaces si parece probable que el sistema experimente cambios significativos.

Algunas técnicas que podemos usar incluyen las siguientes:

- Sustituir las APIs que tienen un gran número de parámetros individuales por otras que pasan objetos u otros tipos de datos estructurados. Por ejemplo, un método *CrearEmpleado* con argumentos: nombre y apellido del empleado, fecha de nacimiento y DNI podría reemplazarse por un método que tenga como único argumento un objeto *Empleado*

- Podemos utilizar un enfoque similar con interfaces de información. Por ejemplo, usando una tecnología de mensaje autodescriptivo como XML para definir formatos de mensaje, y permitiendo añadir elementos opcionales al mensaje, de forma que se puedan ampliar los mensajes con poco o ningún impacto en los elementos del sistema que no necesitan usar la forma extendida de la interfaz

Debemos tener en cuenta que estos enfoques no están exentos de costes. Llevar la flexibilidad de la interfaz al extremo implicaría eliminar por completo la escritura estática de sus interfaces y establecer los tipos de todos los parámetros de solicitud en tiempo de ejecución. Un enfoque tan flexible puede ser más difícil de entender y probar y también puede ser menos eficiente. Es algo parecido a lo que hacen los lenguajes no tipados. También puede introducir muchos problemas sutiles en el sistema porque es difícil verificar la información que falta en un momento dado.

Los detalles para crear interfaces flexibles dependen del entorno tecnológico y del dominio del problema del sistema. Sin embargo, es importante para las inquietudes de evolución, considerar el grado de flexibilidad que se requiere en las interfaces más importantes y cómo lograrlo.

Aplicar técnicas de diseño que faciliten el cambio

Se dan aquí una serie de principios, estilos y patrones de diseño que pueden ayudar a que un sistema sea más fácil de cambiar:

- Los patrones de abstracción y estratificación facilitan el cambio de una parte del sistema con un impacto mínimo en otras
- Los patrones de generalización facilitan el manejo de nuevos casos de uso o tipos de datos, al especializar la funcionalidad de propósito general existente de una manera apropiada para el nuevo caso de uso
- Los patrones de inversión de control (como los manejadores de eventos o "inyección de dependencia") y los patrones de devolución de llamada (se envía un mensaje a un elemento –generalmente un servicio externo que requiere conexión asíncrona– con dirección de respuesta –remitente– para que el servicio conecte con el cliente cuando termine de procesar la petición) ayudan a proteger los elementos de nivel superior de la arquitectura frente a los detalles de implementación de los elementos de nivel inferior

Aplicar estilos arquitectónicos basados en metamodelos

Si tiene requisitos importantes sobre la evolución del sistema, puede valer la pena considerar la adopción de un estilo arquitectónico general que se centre particularmente en apoyar el cambio. Los sistemas basados en metamodelos (o metasistemas) proporcionan un grado muy alto de flexibilidad en algunos dominios problemáticos (particularmente los sistemas de bases de datos que requieren una evolución significativa del esquema).

Los enfoques metamodelo desglosan el procesamiento y los datos del sistema en sus bloques de construcción fundamentales y usan configuraciones de tiempo de ejecución para ensamblarlos en componentes completamente funcionales. Los cambios en los requisitos a menudo se pueden hacer cambiando el metamodelo, en lugar de tener que cambiar los componentes de software subyacentes. Un ejemplo son los sistemas de gestión de contenidos (del inglés Content Management System, CMS), que se especializan en la creación de blogs y periódicos en línea, wikis, comercio, etc. Cuando se aplican en educación, suelen llamarse herramientas de gestión de contenidos de aprendizaje (del inglés Learning Content Management System, LCMS), siendo un ejemplo destacado para nosotros Moodle, por ser el sistema en el que se basa la plataforma PRADO de la Universidad de Granada. Otro ejemplo se da a continuación:

EJEMPLO: Uso de un estilo arquitectónico basado en un metamodelo

Un banco de inversión necesita capturar y procesar los detalles de sus productos, las cuales hacen uso de diversos instrumentos financieros (compras de bonos, transacciones de divisas, transacciones del mercado monetario, operaciones de capital, transacciones derivadas, etc.). De forma regular inventan nuevos tipos de productos financieros, lo que significa que el requisito de apoyar la evolución funcional es muy significativo.
Una arquitectura tradicional identificará un cierto número fijo de tipos de productos ofrecidos, implementando funcionalidad para cada uno de ellos e intentando un procesamiento genérico reutilizable para los aspectos comunes. Al aparecer un nuevo producto, habría que cambiar el sistema para darle cabida.
Por el contrario, una arquitectura basada en metamodelos comienza considerando conceptos/entidades fundamentales como clientes (contrapartes), monedas, fechas de negociación y liquidación, límites de negociación, colecciones de productos (libros) para un comerciante en particular, etc. En lugar de desarrollar un sistema para procesar un conjunto particular de tipos de transacciones o productos, el arquitecto diseña un sistema para proporcionar un conjunto de facilidades para implementar los conceptos subyacentes junto con otras de configuración basada en datos para permitir que los implementadores del sistema definan los tipos de productos que desean ofrecer en términos de estos conceptos subyacentes. Más tarde, cuando se requieren nuevos tipos de productos, se agregan cambiando los datos de configuración, en lugar del código del sistema.

La contrapartida de no necesitar reprogramarse sino sólo reconfigurarse es que son mucho más complejos de desarrollar y menos eficientes, lo que puede limitar su aplicabilidad en entornos donde el rendimiento es una preocupación importante.

Construir puntos de cambio en el software

Una estrategia intermedia es adoptar soluciones de diseño que admitan ciertos tipos de cambios en lugares específicos del sistema, lo que requiere identificar los lugares donde puede haber cambios críticos ("puntos de cambio") y especificar los mecanismos a adoptar para lograrlos.

Además del uso de diversos patrones de diseño que introducen alguna forma de punto de cambio (Fachada, Cadena de Responsabilidad y Puente, etc.) se pueden seguir los siguientes enfoques generales:

- Hacer que los elementos sean reemplazables: generalmente implica que la interfaz a un elemento y su implementación se mantengan separadas para que otros elementos

dependan sólo de la interfaz. Esto permite cambiar el comportamiento del sistema reemplazando un elemento en el momento de la compilación o, con algunas tecnologías y lenguajes de programación, en tiempo de ejecución

- Controlar el comportamiento mediante la configuración: por ejemplo, las entradas, salidas y precisión requeridas de un elemento de procesamiento estadístico en el sistema para permitir que algunos aspectos de la operación del sistema cambien con el tiempo sin modificar su implementación itemUtilizar datos autodescriptivos y procesamiento genérico: ciertos tipos de procesamiento, como conversión de formato, a menudo se pueden realizar de una manera más genérica si se conoce la estructura de los datos de entrada, por lo que se recomienda el uso de un flujo de datos autodescriptivo (como XML) de tal manera que se use la estructura de los datos entrantes para guiar el procesamiento (como cambio de formato)

- Separar el procesamiento físico y lógico: útil cuando el formato de datos cambia con frecuencia, pero no la funcionalidad que se necesita hacer con ellos. Si el software procesa el formato físico de los datos y luego realiza el procesamiento lógico sobre los resultados, será mucho más fácil adaptarlo a un cambio en el formato físico

- Dividir los procesos en pasos: podemos introducir un posible punto de cambio si cada paso de un proceso se programa como un elemento separado

Al igual que otras decisiones de arquitectura de software, debemos ser cautos al introducir puntos de cambio, sopesando el coste derivado de la creación y mantenimiento de cada punto de cambio con la probabilidad de que se use y su importancia dentro de las necesidades de las partes interesadas.

Usar puntos de extensión estándar

Un enfoque relacionado con el anterior es construir puntos de extensión dentro de una tecnología estándar que proporcione esta posibilidad gratuita y flexible de permitir evolución en el sistema. Por ejemplo, la plataforma J2EE permite crearlos para añadir de forma fácil soporte a nuevos tipos de bases de datos (a través de la interfaz JDBC), y a sistemas externos (a través de la interfaz JCA).

Para ello podemos construir adaptadores personalizados que nos permitan utilizar facilidades de integración de aplicaciones estándar para conectar con nuestros sistemas internos y con los paquetes que necesitemos usar, evitando la construcción de mecanismos propios de integración en nuestro sistema.

Hacer cambios fiables

Un cambio mal diseñado puede provocar efectos secundarios graves que causen problemas importantes en el sistema en explotación. Se proponen las siguientes estrategias para ayudar a que los cambios sean fiables:

- Gestor de la configuración del software: permite controlar los cambios en los módulos software e identificar y recuperar las distintas versiones del sistema
- Proceso de compilación automatizado: junto con el control de las versiones que serán las entradas al proceso de compilación/linkado/construcción del software, es importantecrear un sistema automatizado para este proceso que garantice el mismo resultado para las mismas entradas
- Análisis de dependencias: hay muchas herramientas para automatizar este análisis, y usarlas una vez que comencemos a construir el sistema puede ayudar a resaltar dependencias de las que de otro modo no habríamos estado al tanto
- Proceso de lanzamiento automatizado: crear y mantener sistemas para el lanzamiento automatizado que empaqueten el sistema y lo preparen requiere tiempo y esfuerzo, pero en nuestra experiencia siempre es más barato que la alternativa de hacerlo de forma manual
- Creación de mecanismos para deshacer los lanzamientos fallidos: por muchas pruebas que hagamos puede ocurrir que el lanzamiento falle y debamos volver a una versión anterior, para lo cual debemos asegurarnos de tener alguna forma semiautomática de hacerlo, por ejemplo, scripts para revertir a versiones anteriores de software y deshacer los cambios de estado y modelo de datos introducidos por una versión
- Gestión de la configuración del entorno: también debemos controlar los entornos de desarrollo y producción utilizados para crear y ejecutar el sistema. Estos procesos pueden estar menos respaldados por las herramientas existentes, pero es importante administrar cuidadosamente las versiones exactas de las herramientas de desarrollo y las plataformas de implementación, así como la información de configuración precisa para ellas, para evitar la inestabilidad causada por desajustes entre diferentes entornos
- Pruebas automatizadas: debemos asegurarnos de tener un conjunto completo de pruebas, y de que puedan probarse automáticamente en sistemas grandes, para poder evaluar el impacto de un cambio en el comportamiento del sistema
- Integración continua: para lograr detectar errores en los cambios lo más pronto posible, podemos integrar continuamente las partes cambiantes del sistema en lugar de intentar

la integración de todo al final del proceso (integración "big bang"), para ello hay que reunir los cambios del sistema con la mayor frecuencia posible y probar el resultado (al menos una vez al día en la mayoría de los casos)

Preservar los entornos de desarrollo

Una vez que un proyecto ha proporcionado una cantidad significativa de funcionalidad, el entorno de desarrollo original a menudo se desmantela o evoluciona. Con el tiempo, puede llegar fácilmente al punto en el que nadie conoce el conjunto exacto de compiladores, sistemas operativos, parches, bibliotecas, herramientas de compilación, etc., que se utilizan para crear el sistema. Esto puede ser un problema particular para los desarrolladores de productos que admiten una amplia gama de plataformas y versiones de productos.

Parte de la responsabilidad del arquitecto es preservar el entorno de desarrollo (compiladores, sistemas operativos, parches, bibliotecas, herramientas de compilación, etc.) de alguna manera, para que sea posible añadir cualquier nueva funcionalidad con el tiempo. Para ellos podemos registrar claramente los detalles del entorno de desarrollo requerido y asegurarnos de que se conserve suficiente hardware y software para que el entorno se pueda recrear con precisión. Una forma de hacerlo es usar herramientas de virtualización de hardware para crear una imagen autocontenida de todo el entorno de software, que se pueden guardar en el disco y aparecer más tarde exactamente en el mismo estado en que se encontraban cuando se guardaron.

Problemas y errores comunes

- Priorización incorrecta de las dimensiones.- Centrarse en las dimensiones evolutivas incorrectas puede dar como resultado una arquitectura que es más compleja y costosa de construir que las alternativas más simples y, sin embargo, no es particularmente fácil de cambiar cuando es necesario. Debemos por tanto hacer primero un estudio previo para estar seguros de enfocarnos en las dimensiones adecuadas

- Cambios que nunca suceden.- Brindar soporte para cualquier cambio futuro requiere una sobrecarga en términos de diseño, implementación y, a menudo, sobrecarga de tiempo de ejecución, por lo que respaldar una serie de cambios que no suceden puede ser un costo innecesario para su sistema. Debemos por tanto brindar soporte a los cambios que consideremos que serán realmente necesarios

- Impactos de la evolución en criterios críticos de calidad.- Si nos centramos demasiado en el objetivo de flexibilidad podríamos hacer un sistema que sea muy fácil de cambiar pero que no cumpla con una o más propiedades de calidad fundamentales, como el

rendimiento o la disponibilidad, o tan complejo que descuide otras propiedades como la seguridad o la internacionalización debido a la falta de tiempo. Por tanto debemos asegurarnos de mantener el equilibrio entre la flexibilidad y los otros criterios de calidad importantes para el sistema

- Dependencia excesiva de hardware o software específico.- Dificultan el cambio. Para evitarlas, debemos evaluar el uso de componentes especializados en la arquitectura y asegurarnos de que los beneficios que aportan superan las barreras que se oponen al cambio. También debemos conocer las hojas de ruta de los proveedores y otros factores que pueden limitar la vida útil de los componentes especializados y abstraer las interfaces que haya a componentes especializados para que podamos intercambiarlos sin demasiado impacto

- Ambientes de desarrollo perdidos.- los entornos de desarrollo a menudo están sujetos a cambios y evolución independientes a medida que pasa el tiempo. El problema al intentar recrear un entorno de desarrollo o prueba es que a menudo no está claro exactamente qué se necesita para hacerlo (versiones de bibliotecas, compiladores, lenguajes de scripting, parches software, versión de sistema operativo, modelos de componentes hardware, etc.). Para reducir el riesgo, cada vez que se introduce un elemento externo en el entorno de desarrollo, debemos registrar su nombre, versión y origen junto con el motivo de su inclusión

- Gestión de lanzamiento ad hoc.- Es importante organizar y administrar el proceso de administración de versiones con el mismo cuidado que el proceso de creación y prueba del sistema. Para ello debemos invertir en un proceso de lanzamiento automatizado para lograr confiabilidad y repetibilidad

Listas de comprobación

Lista de comprobación para captura de los requisitos:

- ¿Has considerado qué dimensiones evolutivas son más importantes para tu sistema?

- ¿Confías en que has realizado un análisis suficiente para confirmar que tu priorización de las dimensiones evolutivas es válida?

- ¿Has identificado cambios específicos particulares que se requerirán y la magnitud de cada uno?

- ¿Has evaluado la probabilidad de que cada uno de esos cambios sea realmente necesario?

Lista de comprobación para la DA:

- ¿Has realizado una evaluación arquitectónica para establecer si tu arquitectura es lo suficientemente flexible como para satisfacer las necesidades evolutivas del sistema?
- Cuando el cambio es probable, ¿tu diseño arquitectónico contiene el cambio en la medida de lo posible?
- ¿Has considerado elegir un estilo arquitectónico inherentemente orientado al cambio? Si es así, ¿has evaluado los costos de hacerlo?
- ¿Has cambiado los costos de tu apoyo a la evolución por las necesidades del sistema en su conjunto? ¿Alguna propiedad de calidad crítica se ve afectada negativamente por el diseño que has adoptado?
- ¿Has diseñado la arquitectura para acomodar sólo aquellos cambios que crees que serán necesarios?
- ¿Puedes recrear sus entornos de desarrollo y prueba de manera confiable?
- ¿Puedes construir, probar y lanzar de manera confiable y repetible el sistema, incluida la capacidad de revertir los cambios si salen mal?
- ¿Es tu enfoque evolutivo elegido la opción más barata y menos arriesgada de entregar el sistema inicial y la evolución futura requerida?

2.4. Sobre la Ingeniería Informática y el arquitecto software

La Ingeniería Informática es todavía una ingeniería muy nueva. La informática surge en el mundo anglosajón como disciplina mixta entre ciencia e ingeniería ("Computer Science and Engineering") y aún le queda mucho por alcanzar la madurez como ingeniería.

Pensemos lo que es más específico del ingeniero de cualquier otra rama, que nadie puede hacer por él: Elaborar proyectos y dirigirlos.

Cuando hablamos de elaborar proyectos es que él (o ella) es el responsable de la obra que se produzca aplicando el mismo y por eso debe firmarlo y registrarlo en el colegio correspondiente. Tiene responsabilidades tanto civiles como penales por actitudes negligentes en su elaboración. Eso conlleva asumir una numerosa lista de responsabilidades, no sólo en cuanto a la elaboración técnica de la obra (desglosada en distintos planos y documentación adicional), sino en cuanto a la elaboración del presupuesto y a la garantía de una serie de requisitos de seguridad, accesibilidad (reducción de barreras arquitectónicas), estudio de

impacto ambiental, y otras normas de obligado cumplimiento, además de considerar criterios éticos que puedan anticipar el mayor bien común antes de que tenga que ser prescrito por ley.

Cualquier empresa, del tamaño que sea, no puede elaborar un proyecto ni dirigirlo sin tener a un ingeniero que asuma esa responsabilidad. No suelen contratar a un ingeniero o a un arquitecto para tareas básicas como la de elaborar planos detallados (según la rama, pueden ser planos de estructuras, de instalaciones, de cableados, circuitos impresos, etc.), es decir, como delineantes, y si se hace el trabajo queda perfectamente diferenciado del que tiene el ingeniero.

Eso sí, el ingeniero tiene enorme independencia y no debe tomar decisiones arriesgadas que no haya deliberado prudentemente, atendiendo a su propio sentido de la responsabilidad, sabiendo además que él es el responsable, tanto legal como moral, de las consecuencias de la ejecución de su obra.

¿Es esa la situación de un ingeniero informático? El ingeniero informático en general no está considerado así. Él no da la cara por sus proyectos, no tiene responsabilidad legal. Es la empresa la que concibe el proyecto, asume la responsabilidad y encarga una elaboración más detallada a un ingeniero o un grupo de ellos. La empresa impone sus propios criterios, también éticos. La legislación que existe para el resto de disciplinas ingenieriles no se aplica en la ingeniería informática.

¿Qué nos falta para llegar a la madurez? Quizás se pueda dar respuesta intentando contestar a una serie de preguntas:

- ¿Somos conscientes de la responsabilidad al elaborar la DA de un proyecto, de cumplir con los requisitos no funcionales que vienen impuestos por ley? (seguridad, protección de datos, accesibilidad, ...)
- ¿Somos conscientes de nuestra responsabilidad para estar al día de las nuevas disposiciones legales que afecten a la elaboración de proyectos de ingeniería software?
- ¿Somos conscientes de la responsabilidad al elaborar la DA de un proyecto, de las implicaciones éticas (por ejemplo, derivadas de un uso inicuo del mismo) que pueden derivarse de nuestro proyecto, aún no estando todavía reguladas por ley?
- ¿Somos conscientes, a la hora de realizar una planificación presupuestaria y un calendario, de los daños personales (despidos laborales, reducciones de jornadas y/o salarios), económicos y comerciales que pueden derivarse de elaborarlos de forma negligente?

¿Cómo cambiaría el software si para su planificación o su elaboración fuera un ingeniero el responsable legal y la empresa no pudiera asumir esas atribuciones?

Existe en la actualidad en la Ingeniería Informática cierto vacío legal que no ocurre en las otras ingenierías. Si nadie pudiera desarrollar software (salvo para uso particular) si

no estuviera firmado y dirigido por un Ingeniero Informático, al menos software de cierta envergadura,[11] posiblemente no se produciría software a bajo coste que resulte altamente vulnerable, difícil de usar para personas con algunas limitaciones, susceptible de ser empleado para negocios fraudulentos, que no proteja los datos personales o incluso que haga un uso fraudulento de ellos.

Por otro lado, ¿qué puede aportar esta ingeniería, por ser más reciente, a las otras ingenierías más clásicas?

Como conclusión, consideramos como una parte muy importante de esta asignatura, en cuanto a su contribución para ir madurando hacia una mejor formación como ingenieros informáticos, ser capaces de elaborar una DA completa a partir de un supuesto práctico, usando la propuesta de Rozansky (Rozanski, 2011) o cualquier otra en la que se tengan en cuenta los criterios de calidad y se permita incorporar a ellos todas las consideraciones, tanto legales como éticas, que sean responsabilidad del ingeniero.

[11] En el resto de ingenierías, las obras menores las pueden proyectar y dirigir los ingenieros técnicos y las mayores los ingenieros superiores.

Bibliografía

Anand Balachandran Pillai. *Software Architecture with Python*. Packt Publishing, 2017. URL https://learning.oreilly.com/library/view/software-architecture-with/9781786468529/ch08s04.html.

Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (2nd Edition)*. Addison-Wesley Professional, 2003.

Garfixia. Pipe-and-filter, Accessed March 4, 2020. URL http://www.dossier-andreas.net/software_architecture/pipe_and_filter.html.

IEEE Architecture Working Group. Ieee p1471/d5.0 information technology - draft recommended practice for arhitectural description. Technical report, 1999. URL http://www.pithecanthropus.com/~awg/.

Rich Hilliard. Using the uml for architectural description. *Proceedings of UML'99, Lecture Notes in Computer Science*, 1723, 1999.

Greg Phillips, Rick Kazman, Mary Shaw, and Florian Mattes. Process control architectures, 1999. URL https://www.slideshare.net/ahmad1957/process-control.

Trygve Reenskaug. A note on dynabook requirements. Technical report, Xerox PARC, 1979. URL http://folk.uio.no/trygver/1979/sysreq/SysReq.pdf.

Nick Rozanski. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, Second Edition*. Addison-Wesley Professional, 2011. URL https://learning.oreilly.com/library/view/software-systems-architecture/9780132906135/.

Mary Shaw and David Garlan. *Software Architecture*. Prentice Hall, New Jersey, 1996.

Judith Stafford, Robert Nord, Paulo Merson, Reed Little, James Ivers, David Garlan, Len Bass, Feliz Bachmann, and Paul Clements. *Documenting Software Architectures: Views and Beyond, Second Edition*. Addison-Wesley Professional, EE.UU., 2010.

Heinz Züllighoven. *Object-Oriented Construction Handbook*. Science Direct, EE.UU., 2005.