Tema 2. Arquitecturas de 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

La *arquitectura software* es el nivel más elevado en el diseño software, que se refiere a la estructura general del sistema, con dos elementos básicos:

- *Componentes*: unidades que proporcionan alguna funcionalidad claramente distinguible de otras. Pueden ser compuestos or otros a su vez (subsistemas)
- *Conectores*: modelan y controlan la interacción entre los componentes
- *Patrones* que guían estas relaciones y restricciones de los mismos

Un *estilo arquitectónico* (o patrón arquitectónico) define:

- Tipos de componentes y conectores
- Conjunto de restricciones sobre la forma en la que pueden combinarse estos elementos

Algunos de ellos se han intentado clasificar:

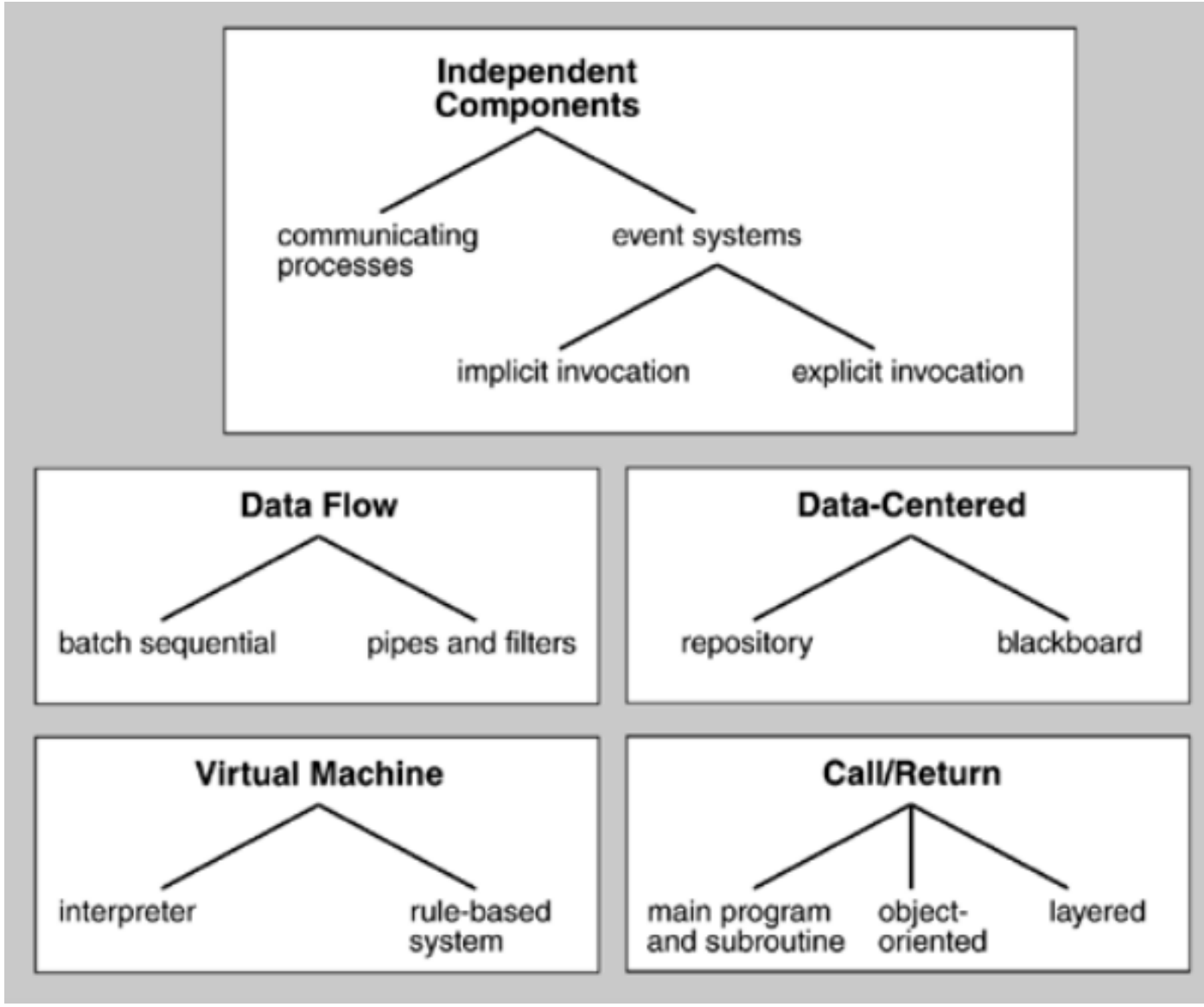


Figura 1: Diagrama de componentes usando el estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero. [Fuente: (Bass et al.,

Estilo *Tubería y filtro*

Componentes:

- 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
- Tubería: Transportan la corriente de datos (data stream)

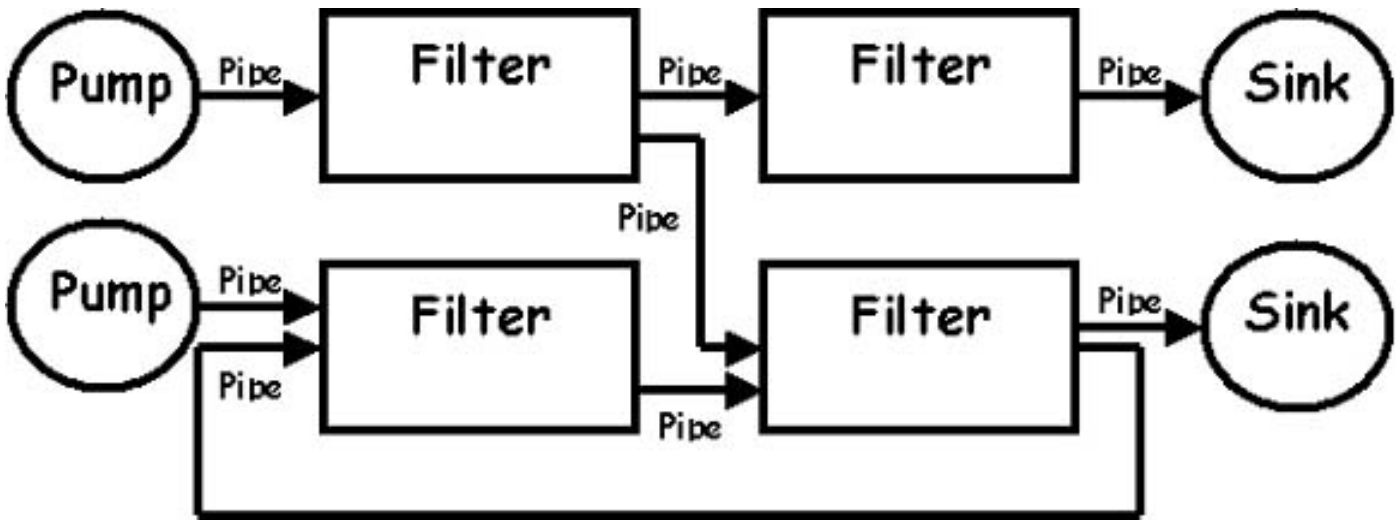


Figura 2: Estructura del estilo arquitectónico *Tubería y filtro*. [Fuente: (Garfixia, Accessed March 4, 2020)]

Ejemplo: shell de Unix:

Estilo *Abstracción de datos y organización OO*

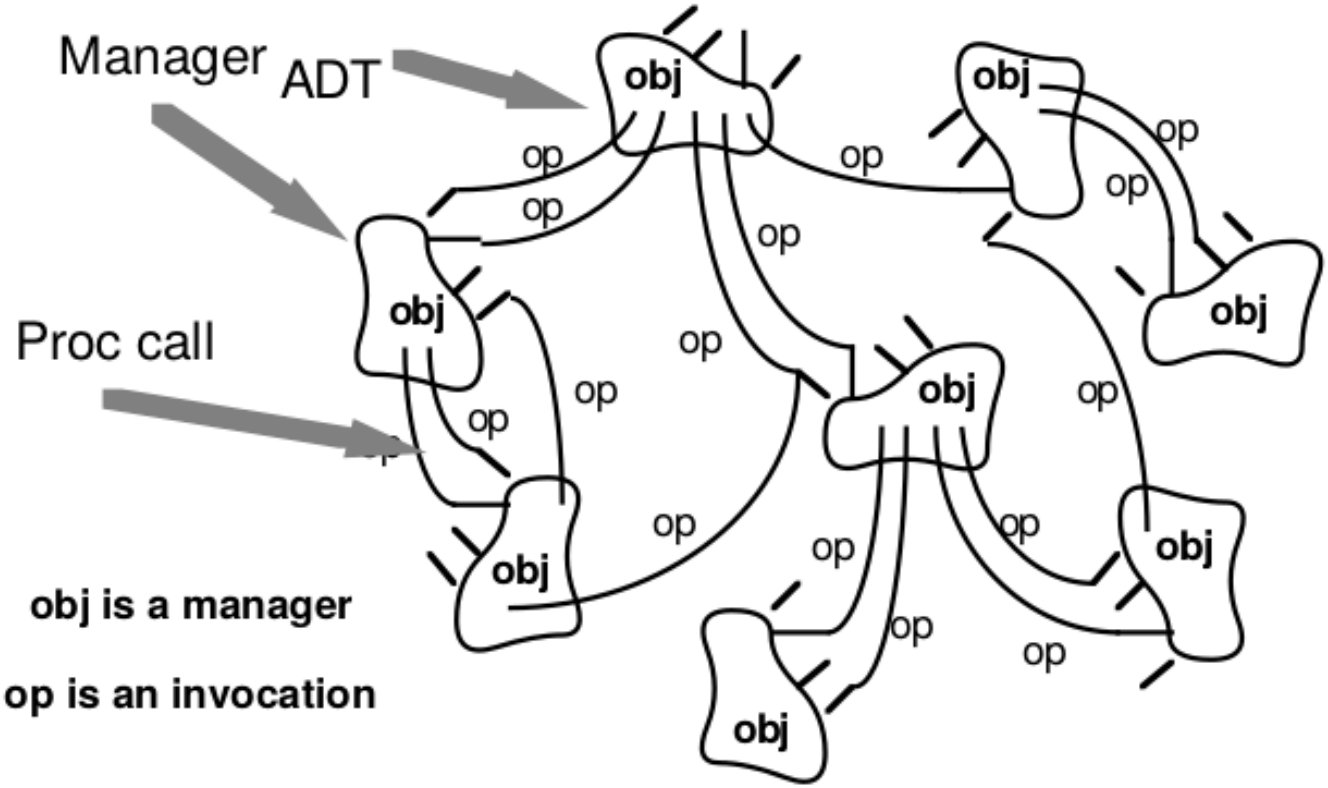


Figura 3: Ejemplo del estilo arquitectónico *Abstracción de datos y organización OO*. [Fuente: (Shaw and Garlan, 1996, pg. 23)]

Estilo *Basado en eventos*

Variantes:

- Invocación implícita (estilo *Manejador de eventos* o *Publicar/subscribir*
- Invocación explícita

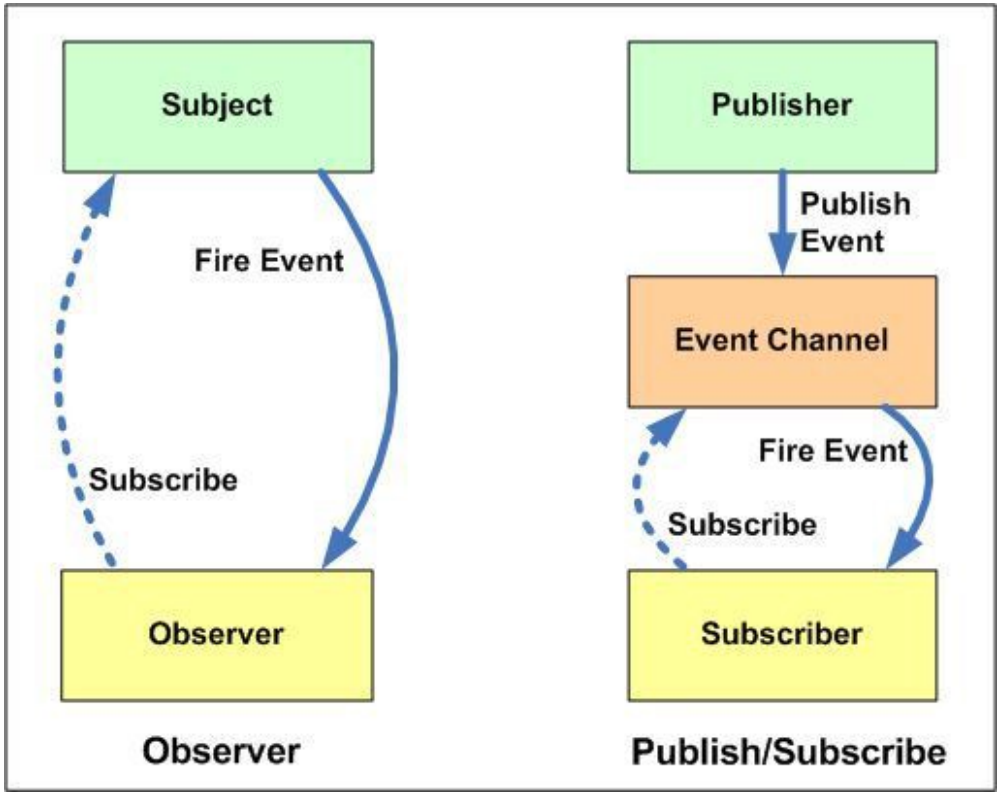


Figura 4: Diferencia entre el estilo arquitectónico *Basado en eventos* y el patrón de diseño *Observador*. [Fuente: https://hackernoon.com/observer-vs-pub-sub-pattern-50d3b27f838c

Estilo *Basado en eventos*

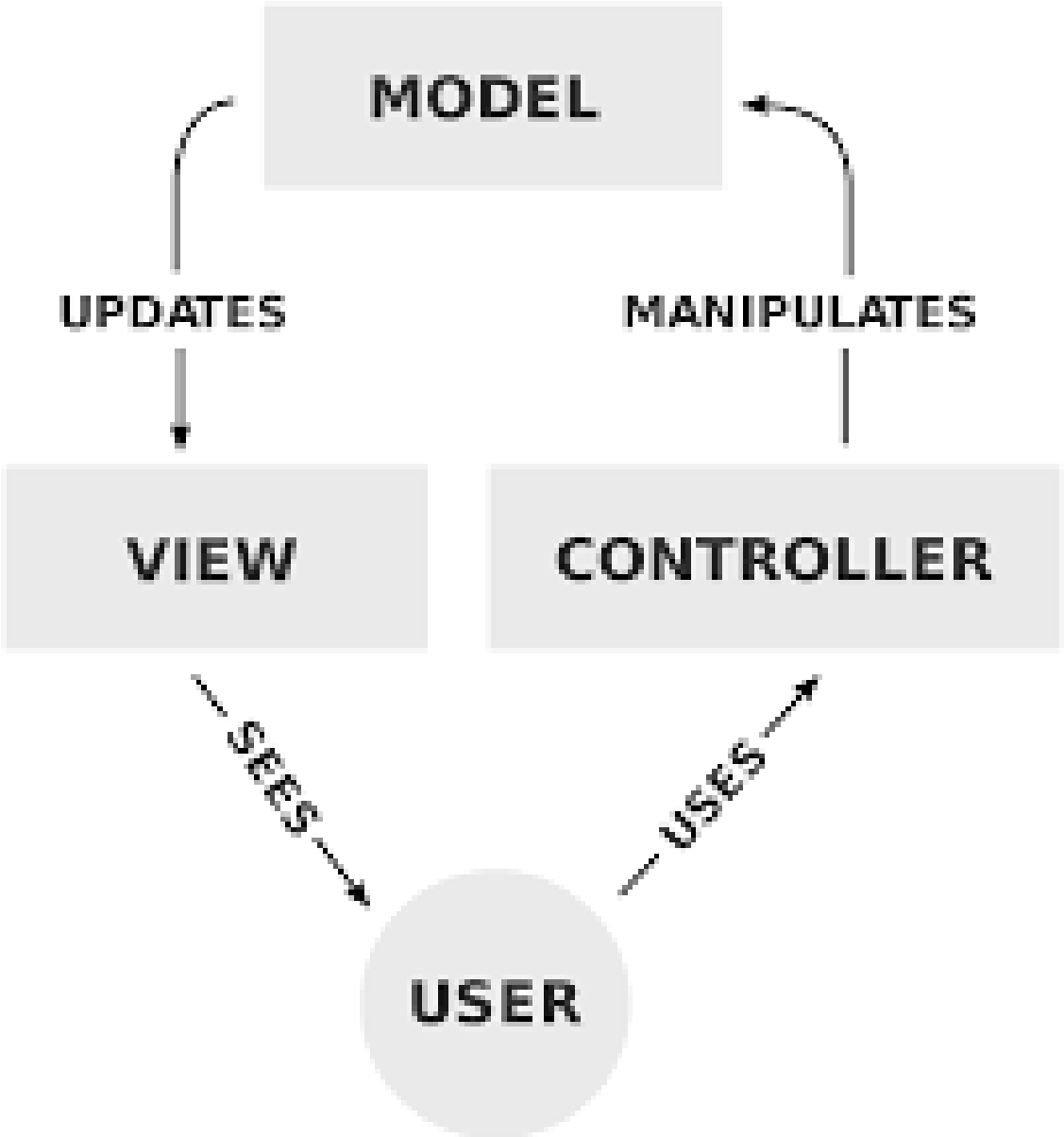
```
public class Observable
{
    public ActionEvent<Object, Object> onStateChange;

    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)
    {
```

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.

Variantes:

- Controlador ligero o Modelo activo
- Controlador pesado o Modelo pasivo


MODEL
UPDATES
MANIPULATES
VIEW
CONTROLLER
SEES
USES
USER

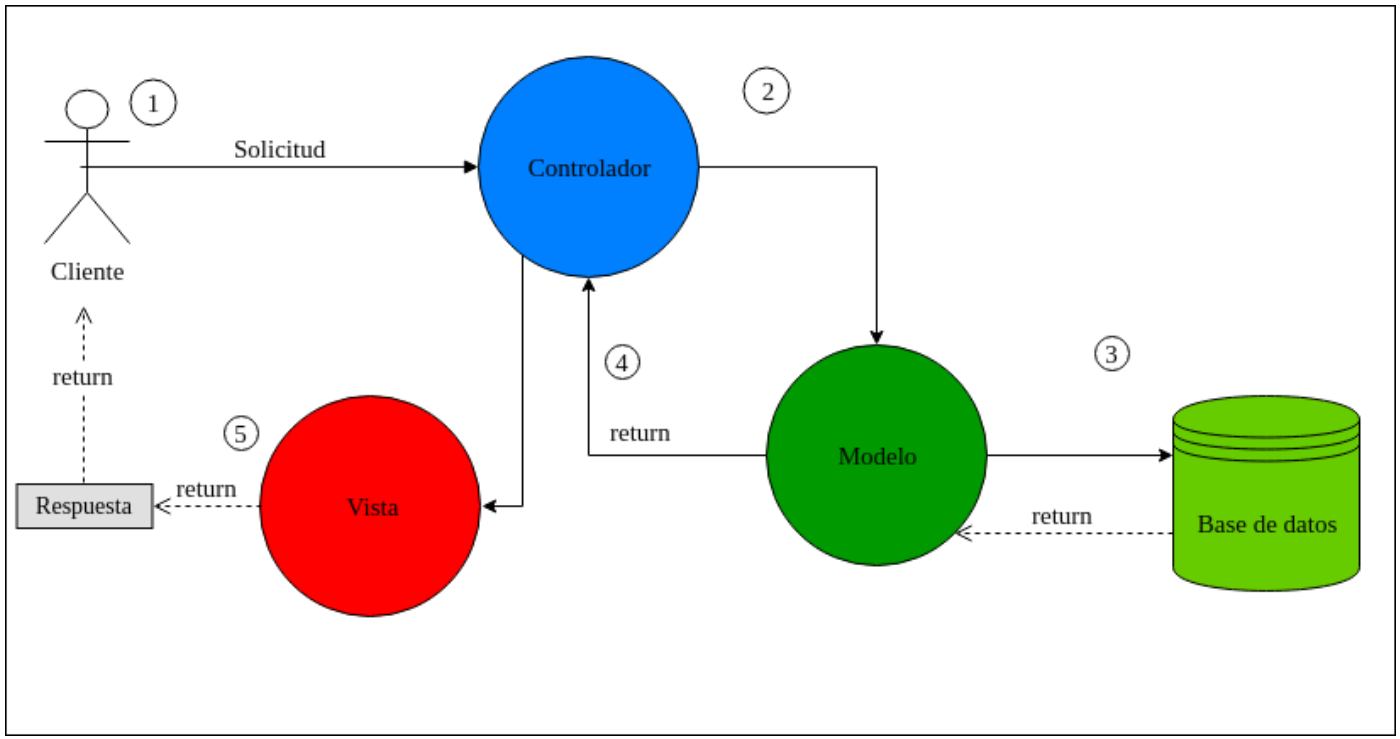


Figura 6: Estructura del estilo arquitectónico MVC de Controlador pesado. [Fuente: https://articulosvirtuales.com/articles/educacion/que-es-el-modelo-vista-controlador-mvc-y-como-funciona]

Figura 7: Implementación del estilo MVC hecha por Java SWING.

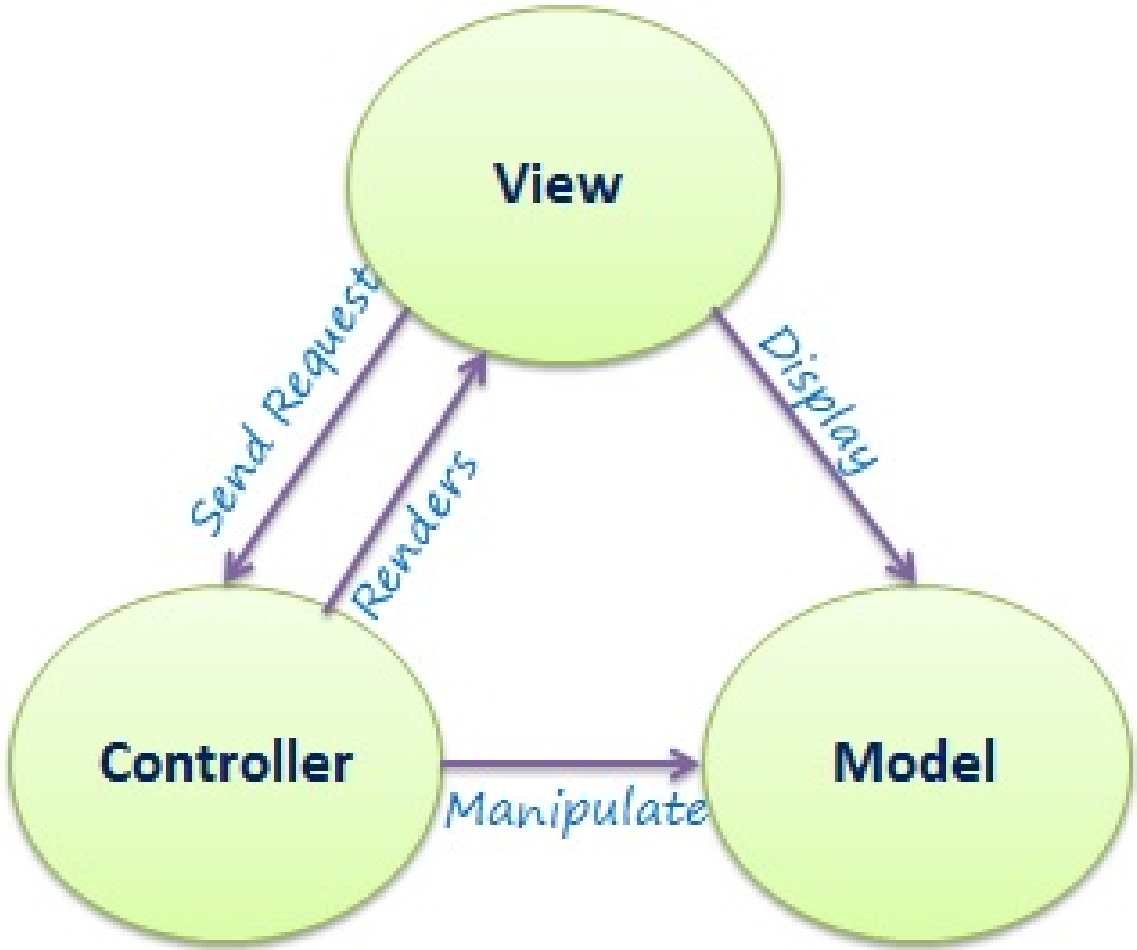


Figura 8: Estructura del estilo arquitectónico MVC pesado implementado por Angular JS. [Fuente: https://w3tutoriels.com/angularjs/angularjs-mvc/]

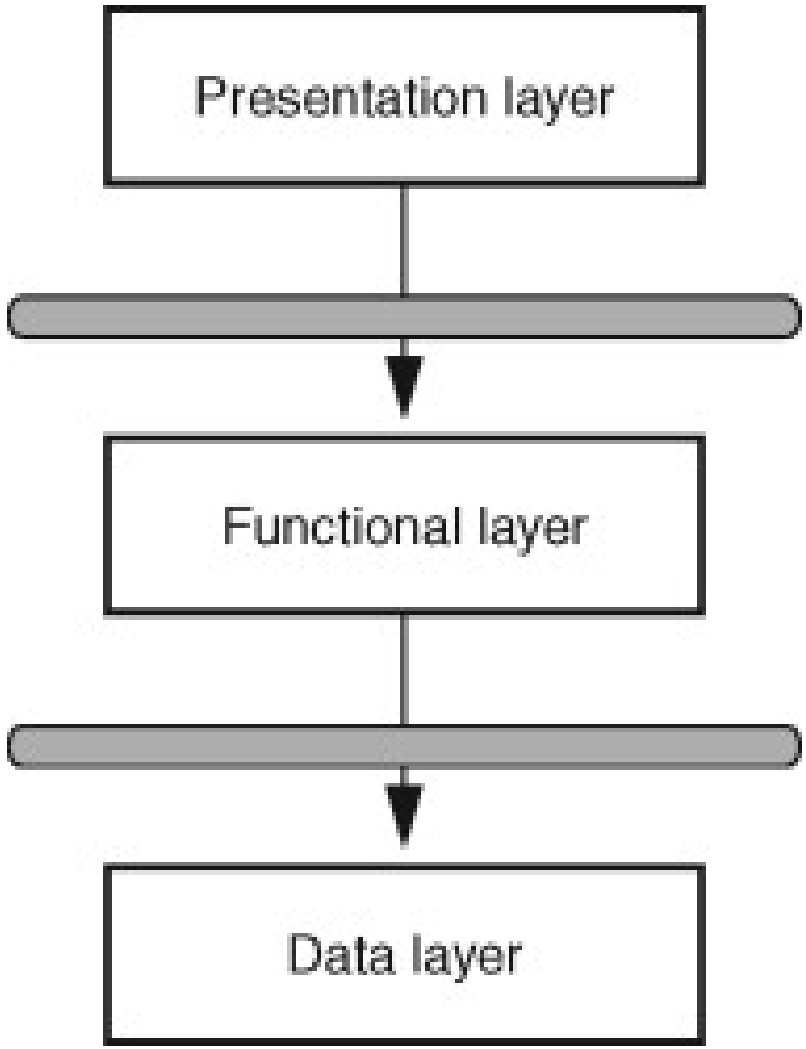


Figura 9: Estructura del estilo arquitectónico *Sistema por capas* con sólo tres capas. [Fuente: (Züllighoven, 2005)]

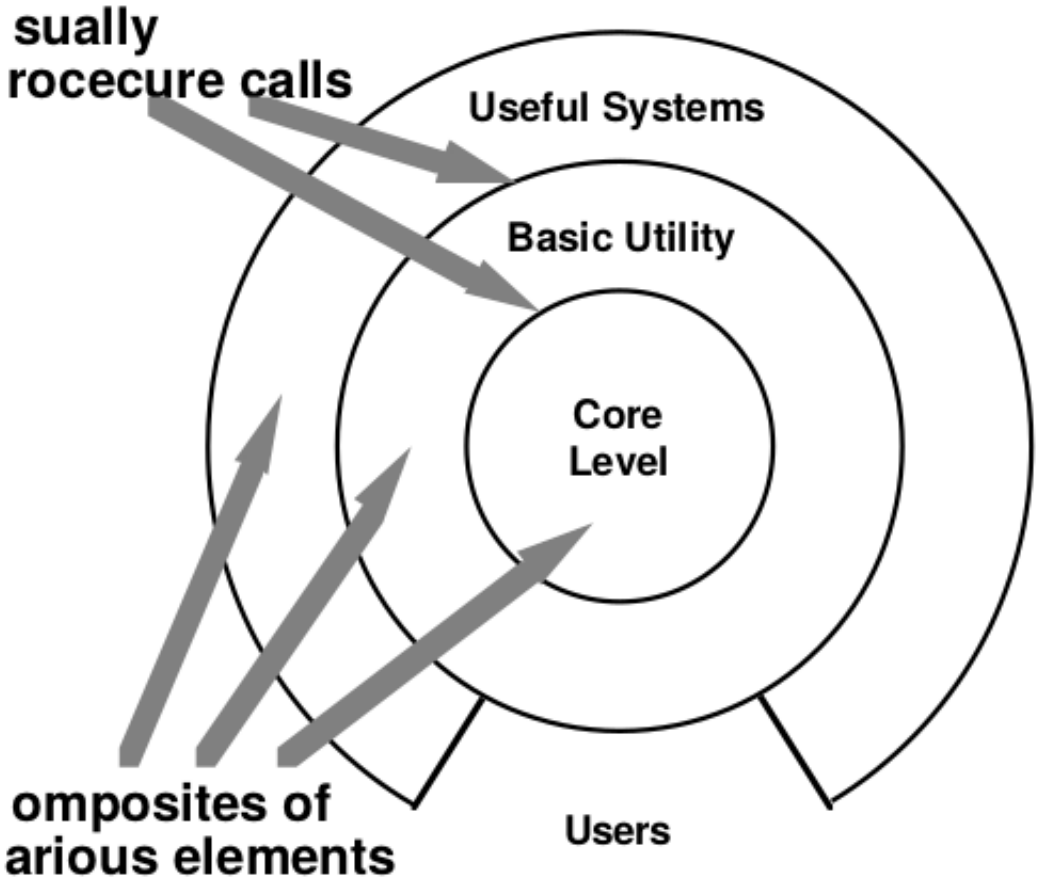


Figura 10: Estructura del estilo arquitectónico *Sistema por capas* genérico. [Fuente: (Shaw and Garlan, 1996, pg. 25)]

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).

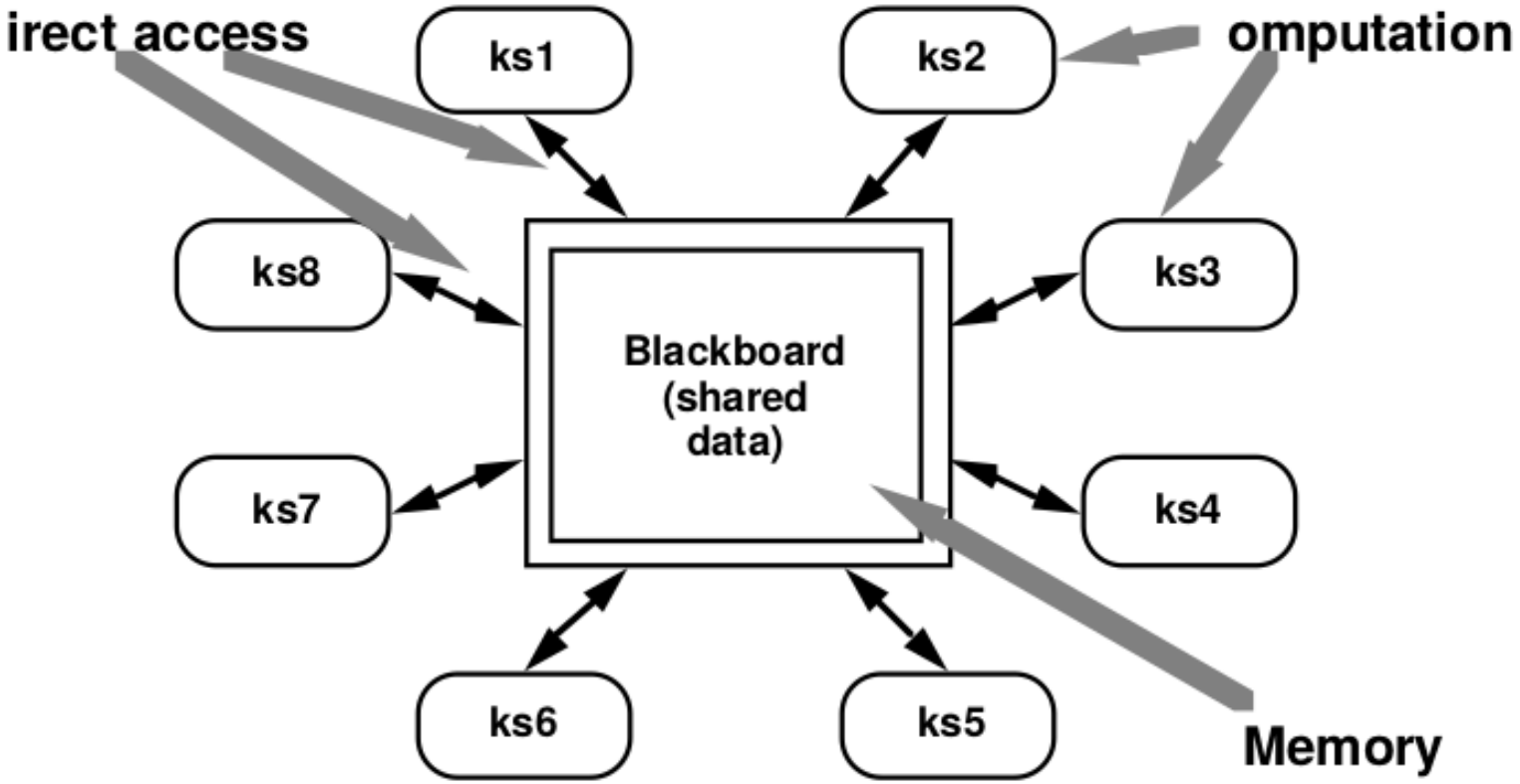


Figura 11: Estructura del estilo arquitectónico *Pizarra*. [Fuente: (Shaw and Garlan, 1996, pg. 26)]

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

Estilo *Intérprete*

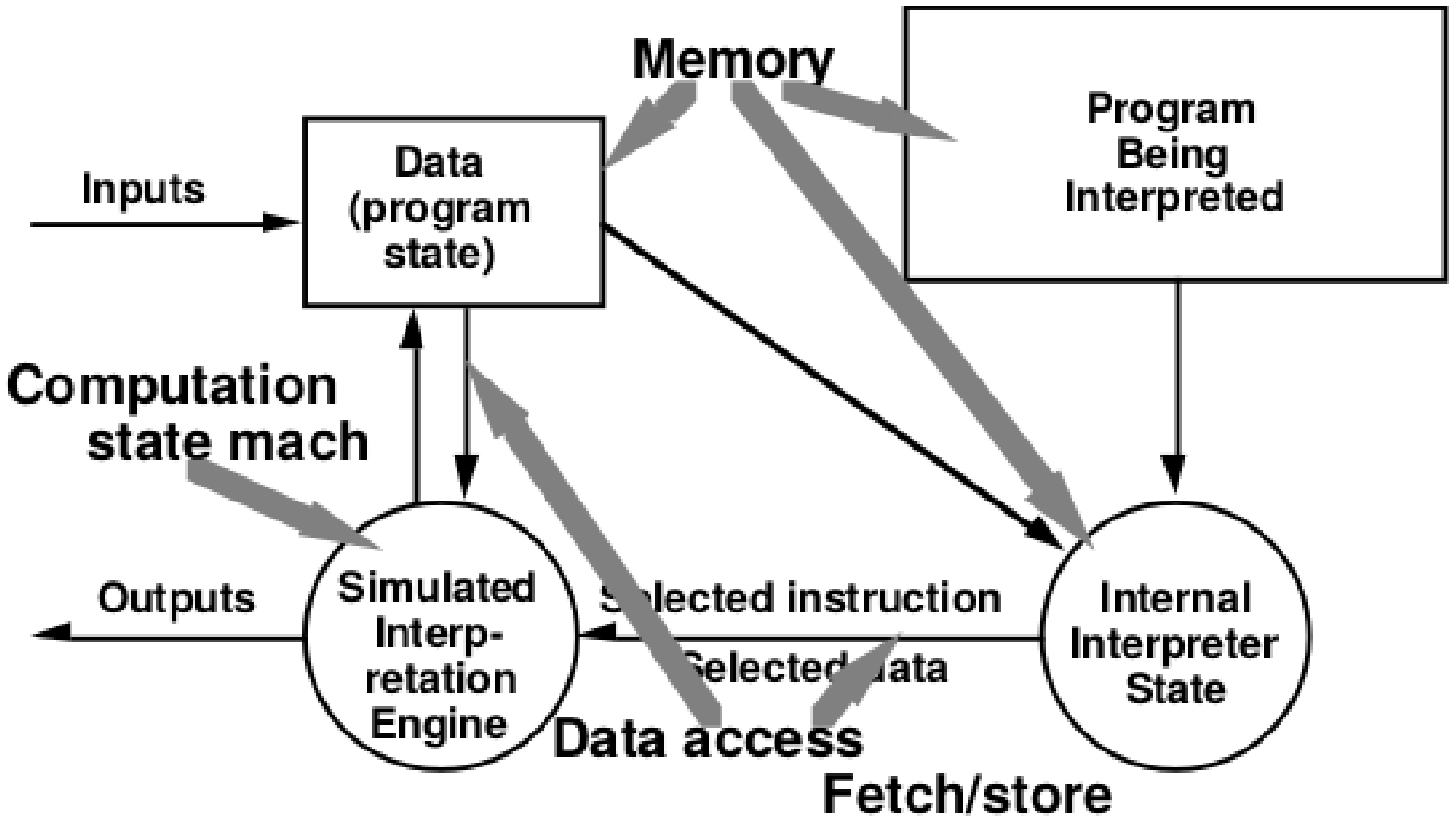


Figura 12: Estructura del estilo arquitectónico *Intérprete*. [Fuente: (Shaw and Garlan, 1996, pg. 27)]

Variantes:

- *Ciclo abierto* (Figure 13).- 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 14).- 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 15)
  - *Control de procesos preventivo* (Figure 16)

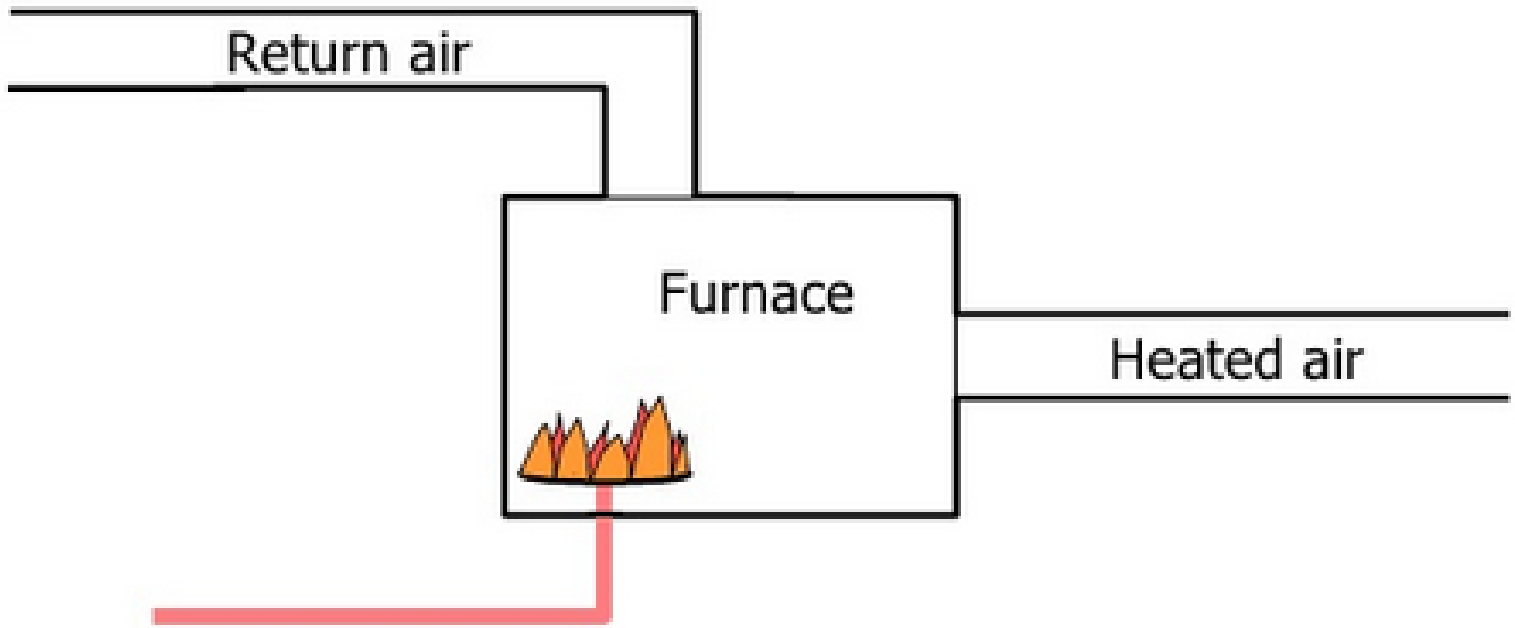


Figura 13: Ejemplo de sistema con estilo arquitectónico *Control de procesos de ciclo abierto*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

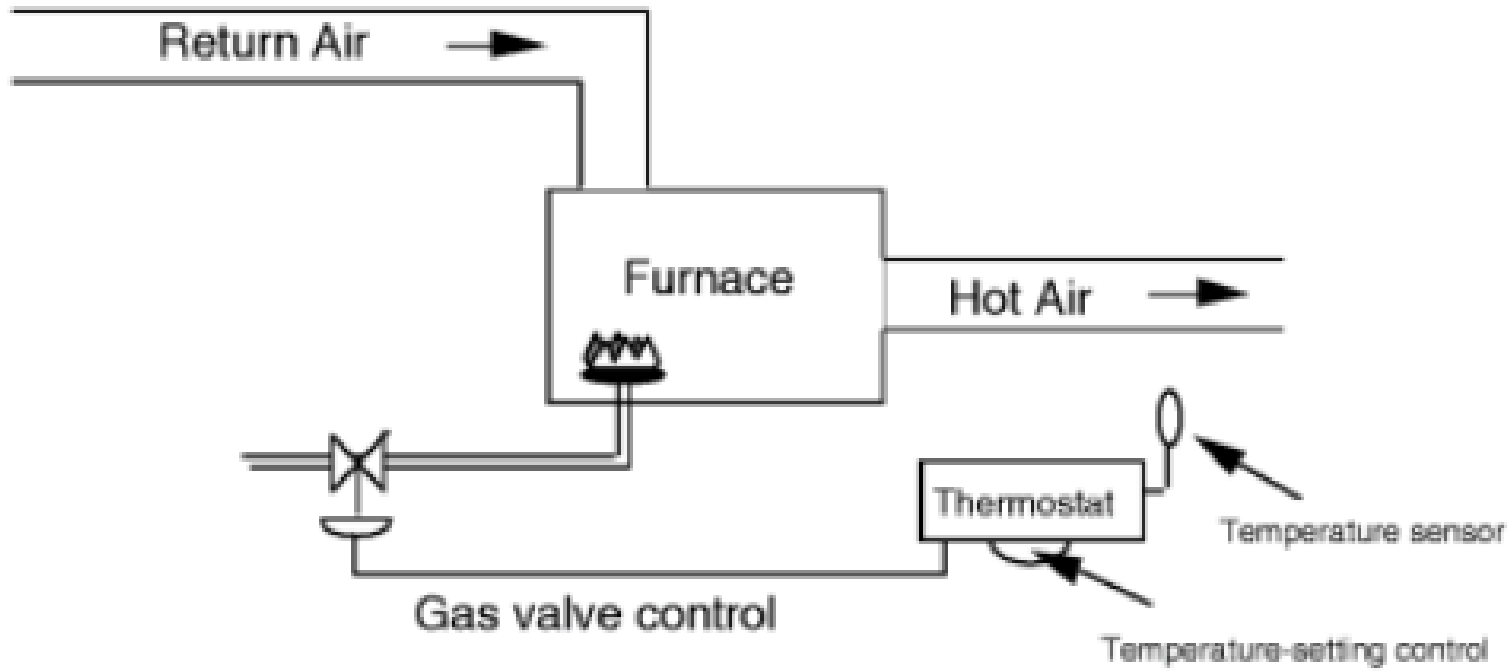


Figura 14: Ejemplo de sistema con estilo arquitectónico *Control de procesos* de ciclo cerrado. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

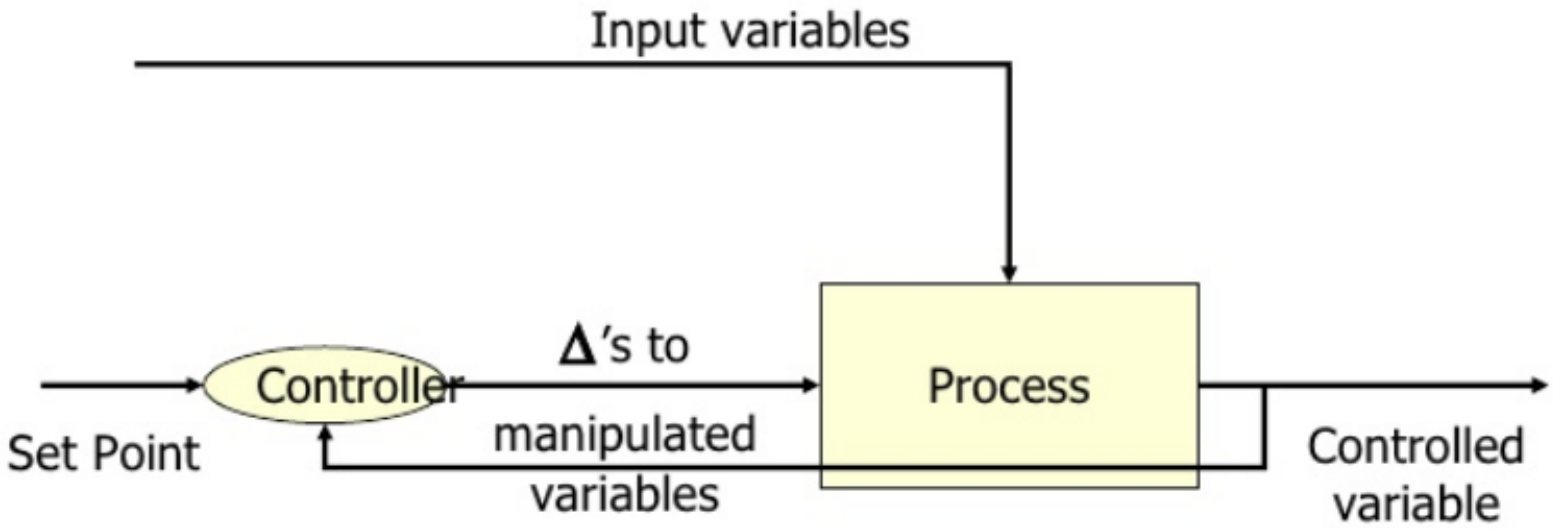


Figura 15: Estructura del estilo arquitectónico *Control de procesos retroalimentado*. [Fuente: (Shaw and Garlan, 1996, pg. 29)]

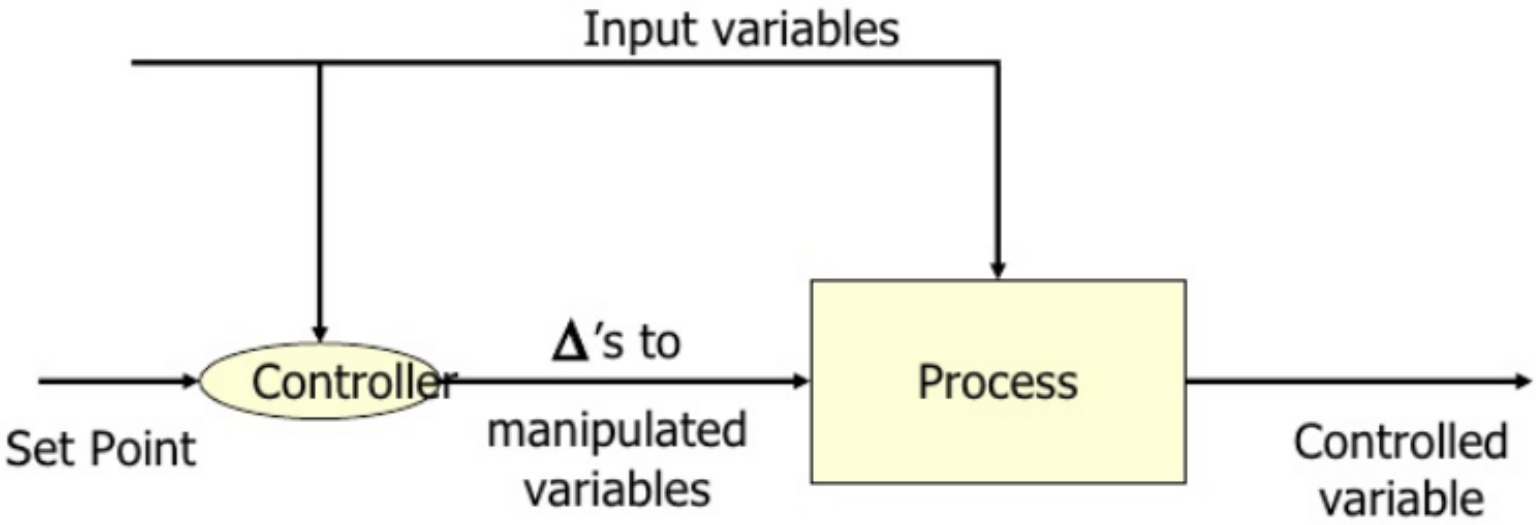


Figura 16: Estructura del estilo arquitectónico *Control de procesos preventivo*. [Fuente: (Shaw and Garlan, 1996, pg. 30)]

Ejemplo control velocidad de crucero (*preventivo*): diagrama de bloques

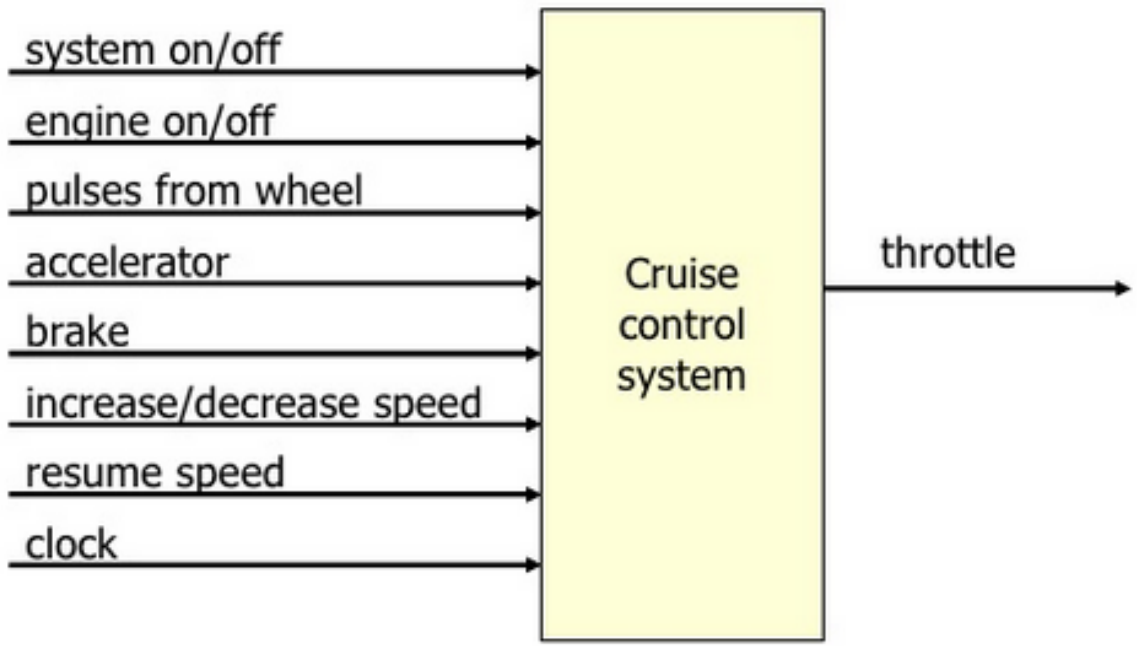


Figura 17: Diagrama de bloques del sistema de control de la velocidad de crucero. [Fuente: (Shaw and Garlan, 1996, pg. 52)]

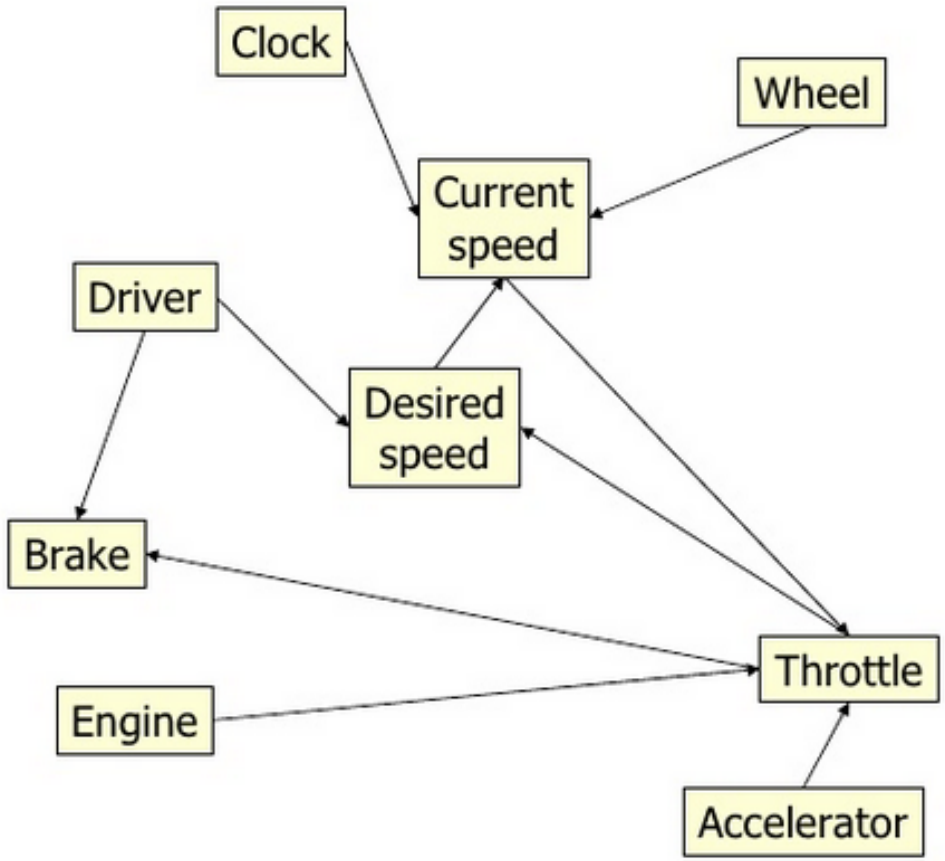


Figura 18: Diagrama de clases para el sistema de control de la velocidad de crucero. [Fuente: (Phillips et al., 1999)]

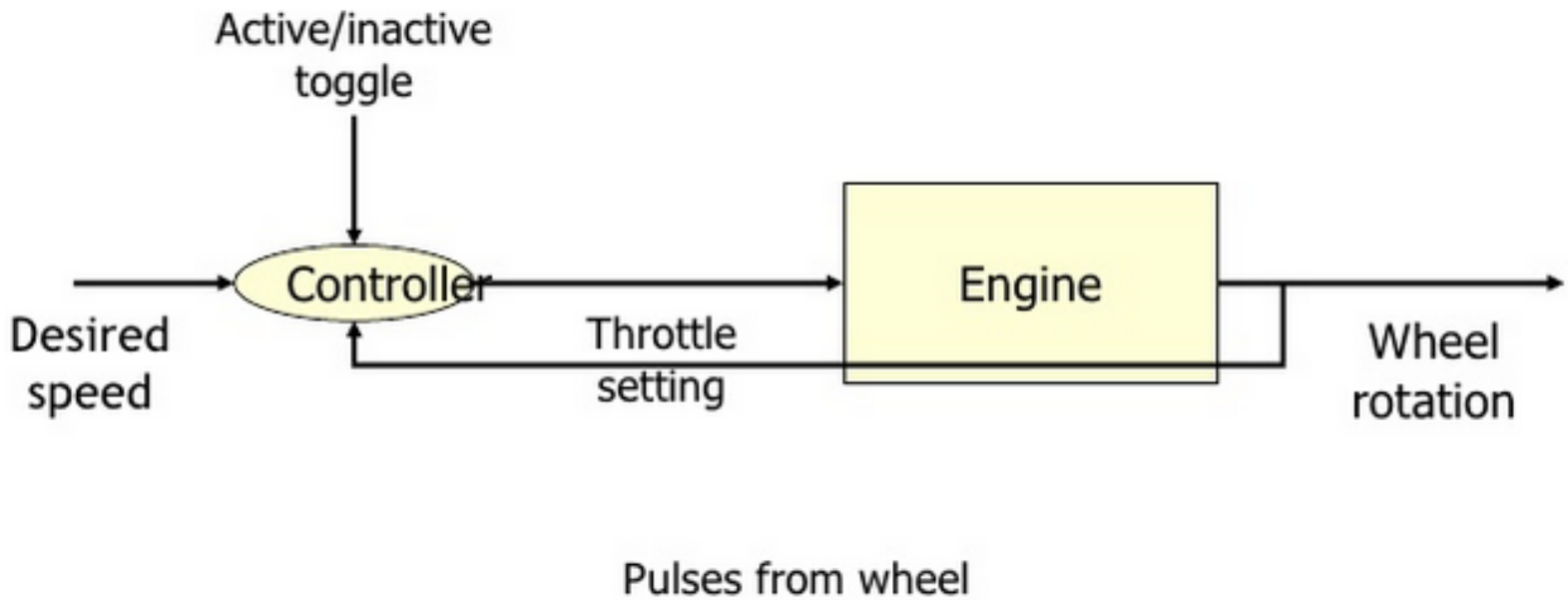


Figura 19: Diagrama del estilo arquitectónico *Control de procesos* para el sistema de control de la velocidad de crucero. [Fuente: (Phillips et al., 1999)]

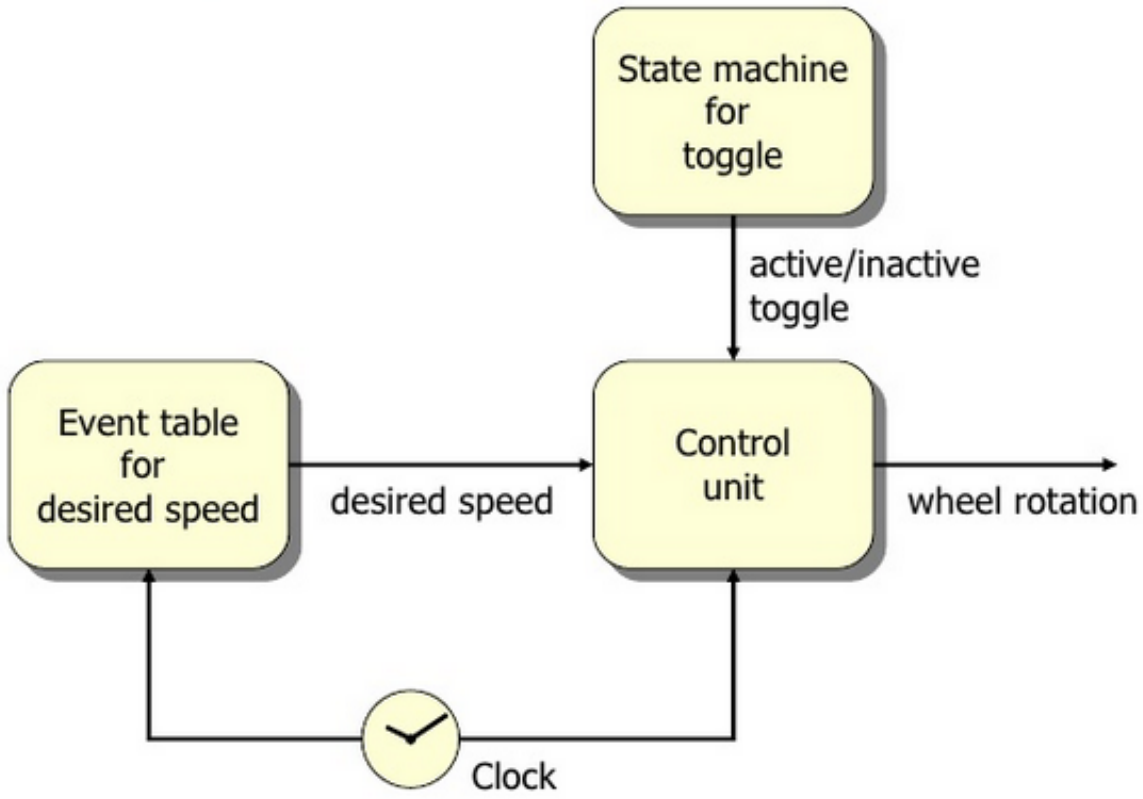


Figura 20: Diagrama de componentes). [Fuente: (Phillips et al., 1999)]

- Estilo de *Procesos distribuidos*
  - Estilo *Cliente/Servidor*
- Estilo *Organización programa principal/subrutinas*
- Estilo *Sistema de transición de estados*
- Estilos específicos de un dominio

En la realidad, los estilos se combinan de forma que en un sólo 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.
Un mismo componente puede formar 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.

- El estándar IEEE P1471 (actualizado en 2011 por ISO/IEC/IEEE42010)
- La propuesta de Rozansky (Rozanski, 2011)

Los objetivos de IEEE para el P1471 son:

1. Asumir una interpretación amplia del concepto de arquitectura en sistemas software intensivos
2. Establecer un marco de trabajo conceptual y un vocabulario para hablar de los aspectos arquitectónicos del sistema, como los términos:
   - "Arquitectura"
   - "Descripción arquitectónica" (DA)
   - "Vista"
3. Identificar y declarar prácticas arquitectónicas sólidas
4. Permitir la evolución de estas prácticas conforme evolucionan tecnologías que sean relevantes. El marco debe ser suficientemente general para acoger las técnicas actuales y suficientemente flexible para que pueda evolucionar

Entidades conceptuales consideradas:

- Descripción arquitectónica.- Colección de productos para documentar la arquitectura de un sistema.
- Parte interesada (stakeholder).- Cualquier individuo, clase o componente externo, institución o rol interesado en el sistema
- Inquietud (concern).- Cualquier área de inquietud de la arquitectura que tengan las partes interesadas en el sistema
- 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
- Punto de vista.- Contiene las reglas por las que se rigen las vistas concretas

Modelo conceptual

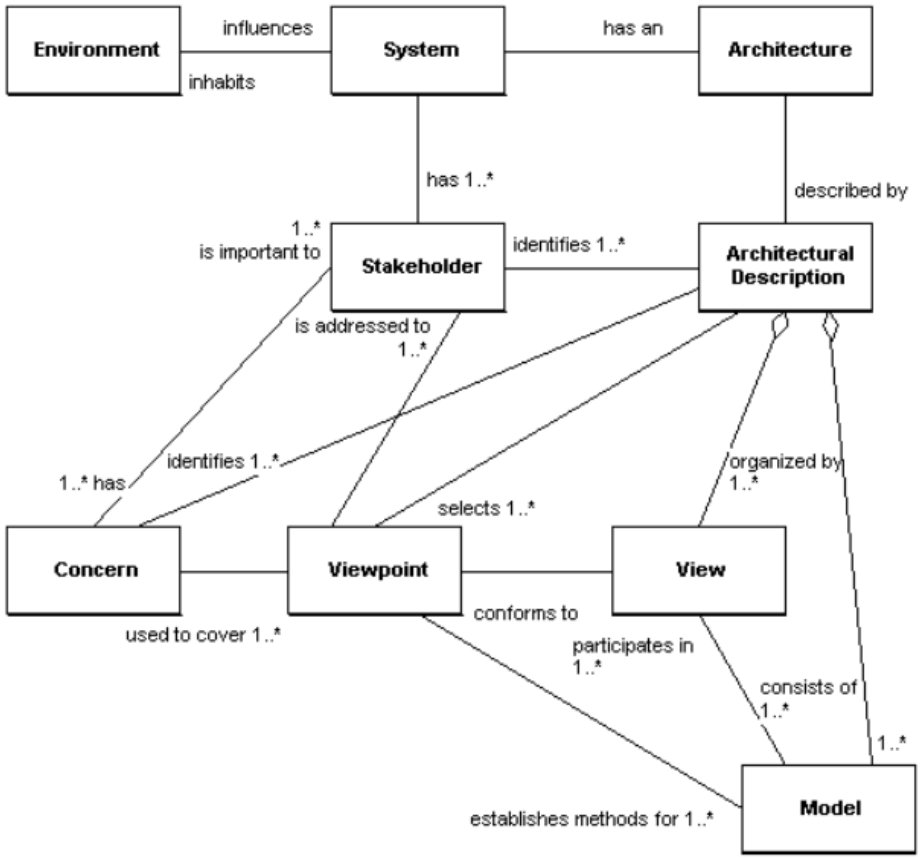


Figura 21: Modelo conceptual del estándar IEEE P1471. [Fuente: (Hilliard, 1999)]

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.

CRITERIO DE CALIDAD: Completitud

La completitud de una DA de un sistema software es la condición de que da respuesta a las inquietudes de todas 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 intereses abordados por este punto de vista
- Intereses abordados por este punto de vista
- Lenguaje, técnicas de modelado y métodos analíticos usados
- Fuente, si hay alguna, del punto de vista (autor, citas)

El arquitecto software debe elegir los puntos de vista arquitectónicos desde una lista predefinida de ellos (la librería de puntos de vista) en atención a cada una 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.
Puede usarse UML como lenguaje de modelado: Hilliard Hilliard (1999) propone cuatro formas distintas de usarlo para aplicar el estándar IEEE P1471:

1. "Listo para usar" (out of the box) o "predefinido"
2. "Extensión ligera"
   - Usar extensiones de UML
   - Usar variantes UML
3. "UML como marco integrador"
4. "Fuera de la ontología "UML"

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

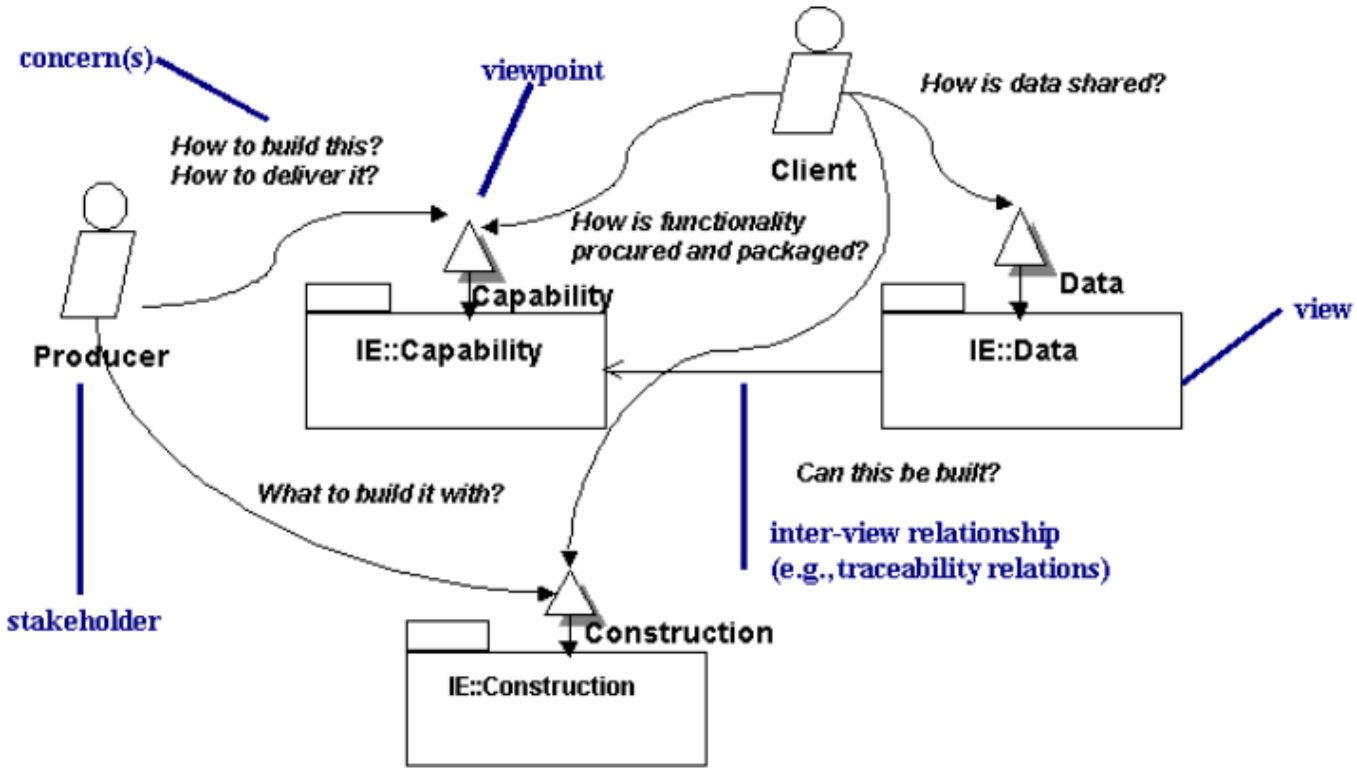


Figura 22: Un fragmento de una visión general de un sistema. Los iconos son las partes interesadas, las inquietudes se representan con flechas, los puntos de vista con triángulos y las vistas con paquetes. [Fuente: (Hilliard, 1999)]

Rozansky hace una propuesta concreta para implantar el estándar IEEE P1471(Rozanski, 2011), donde detalla los siguientes aspectos:

- Cómo identificar y comprometer a las partes interesadas en el sistema
- Cómo identificar las inquietudes
- Cómo identificar y usar escenarios
- Catálogo de puntos de vista
- Descripción detallada de los distintos puntos de vista
- Adición de "perspectivas" al estándar P1471 basado en puntos de vista
- Descripción detallada de las distintas perspectivas

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.

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.

Responsabilidades de una parte interesada en el sistema:

- 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

Tipos de partes interesadas:

- 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 desplegado
- 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

EJEMPLO: Proyecto de desarrollo "off-the-shelf"

Una empresa A; fabricante hardware, quiere usar un sistema "Enterprise Resource Planning" (ERP) para manejar mejor todos los aspectos de la cadena de producción usando un paquete comercial "custom off-the-shelf" (COTS) combinado con algún software de desarrollo propio para aspectos más especializados. El sistema debe lanzarse en un año, a tiempo para una reunión con accionistas que están considerando una futura financiación de la empresa.

Los **adquirientes** del sistema incluyen al director gerente, que autorizará la financiación del proyecto, al departamento de compras y a los representantes informáticos, que evaluarán distintos paquetes ERP. Los **usuarios** del sistema son empleados internos que gestionan pedidos, compras, financiación, fabricación y distribución. Los **desarrolladores**, **administradores del sistema**, **ingenieros de producción** y **mantenedores y personal del departamento TI** 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 desarrollado por un proveedor especializado parcialmente financiado por inversores externos. Los usuarios previstos del producto estan en empresas que comprarán el producto. Las partes interesadas a menudo se reparten entre varias organizaciones.
Ej: una empresa B, proveedora de software educativo, creará producto para que profesores universitarios administren sus horarios de clases. Ha formado una sociedad con una universidad local y obtenido fondos de capital de riesgo con otra. Los **adquirientes** son directores gerentes y jefes de producto de la empresa B, del socio educativo, y representantes de los accionistas. Los **usuarios del sistema** son profesores universitarios y personal administrativo (representados por usuarios potenciales del producto). Los **desarrolladores y mantenedores** son personal de desarrollo de la empresa B, y los **asesores** son tomados de las tres empresas asociadas. La empresa B y/o las universidades compradoras pueden proporcionar el **personal de mantenimiento**. Los **comunicadores** son autores técnicos de la empresa B que escriben la guía de usuario. Los **ingenieros de producción** proporcionan los entornos de desarrollo y prueba de la empresa B y la infraestructura para fabricar CDs y distribuir actualizaciones.

EJEMPLO 3: Desarrollo subcontratado

Una empresa subcontrata los servicios de otra para proporcionar sistemas o servicios. Las partes interesadas pueden encontrarse dentro de la empresa subcontratada, dificultando su identificación e interacción.
Ej: La empresa C, una empresa financiera consolidada, desea expandir su presencia en Internet comercializando una gama de servicios financieros a clientes últimos, residentes del país donde tiene su sede la empresa C o en otros países. La empresa C planea contratar el desarrollo y uso del sistema a un desarrollador web establecido.
Los **adquirientes** son directores gerentes que autorizarán la financiación del proyecto. Los **usuarios** son clientes ordinarios, que accederán al sitio web público (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** son 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.

DEFINICIÓN: inquietud (concern)

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

Algunas pueden formularse de manera vaga pero no son prescindibles.

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.
- En todo momento, el sitio web debe tratar de presentar una cara "humana" al cliente (incluso en las automatizadas por completo).
- Debe ser fácil de usar para los clientes que tienen una experiencia limitada con los ordenadores y con el comercio electrónico.
- Debe ser "responsiva" (rápida de cargar y responder a las acciones del cliente) independientemente de la velocidad de conexión a Internet del cliente.
- 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.

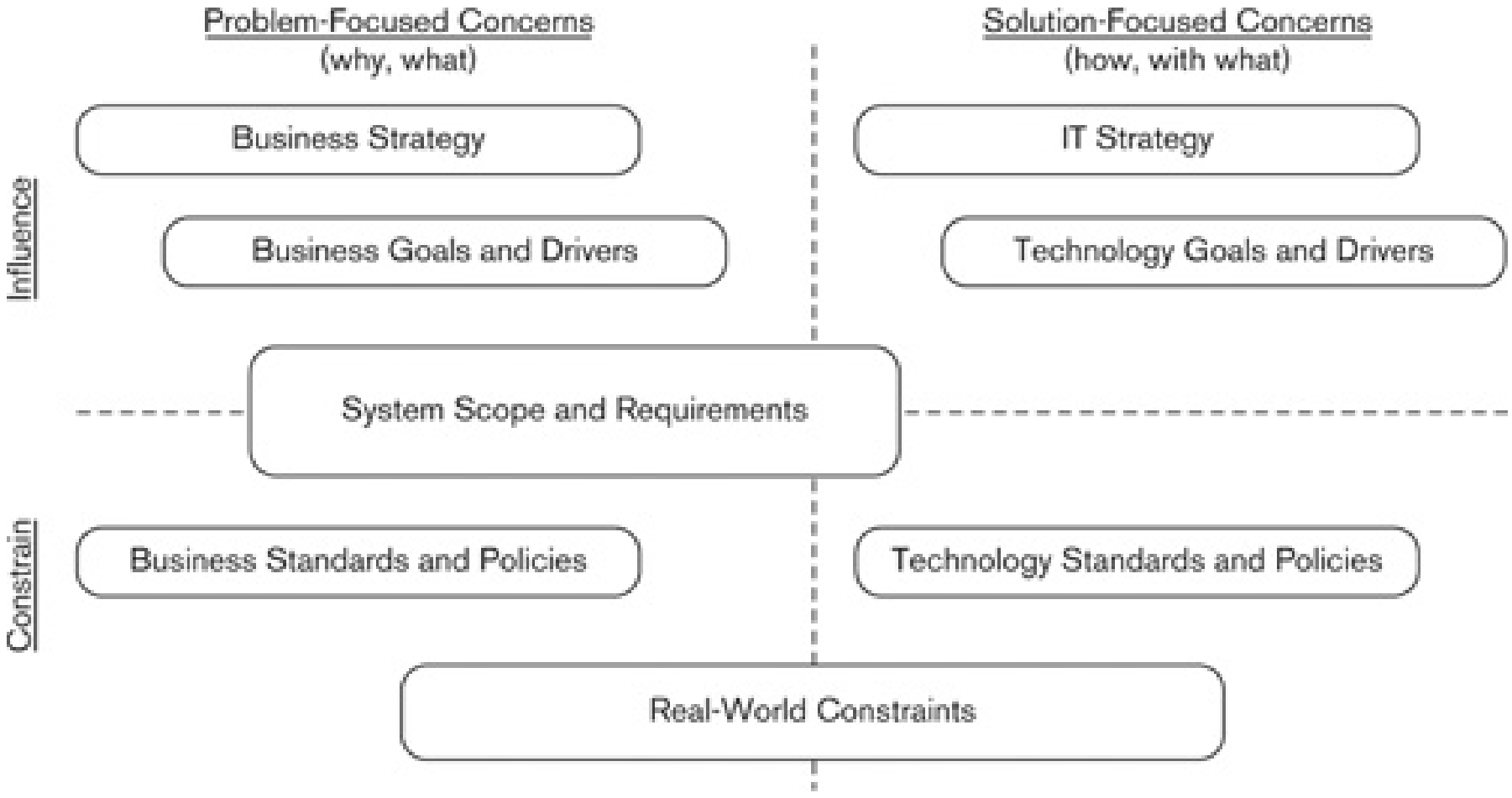


Figura 23: Inquietudes según dos criterios de clasificación. [Fuente: (Rozanski, 2011, Cap. 8)]

DEFINICIÓN: Escenario arquitectónico

Un "escenario arquitectónico" es una descripción bien definida de una interacción entre una entidad externa (actor) 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.

¿Por qué usarlos? 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

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

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.
- 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.

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 deben procesarse en segundo plano. Las estadísticas de resumen anteriores deben permanecer disponibles hasta que las nuevas estén listas

EJEMPLO: Tres escenarios de calidad que sumarizan datos de entrada

1. Problemas de tamaño en la actualización diaria de datos

- Descripción general: cómo se comporta el procesamiento al final del día cuando los volúmenes de datos crecen repentinamente
- Estado del sistema: el sistema tiene estadísticas resumidas en su BD que ya se han procesado, y los elementos de procesamiento se cargan 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: 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

EJEMPLO: Tres escenarios de calidad que sumarizan datos de entrada (cont.)

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

EJEMPLO: Tres escenarios de calidad que sumarizan datos de entrada (cont.)

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

Catálogo de puntos de vista

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

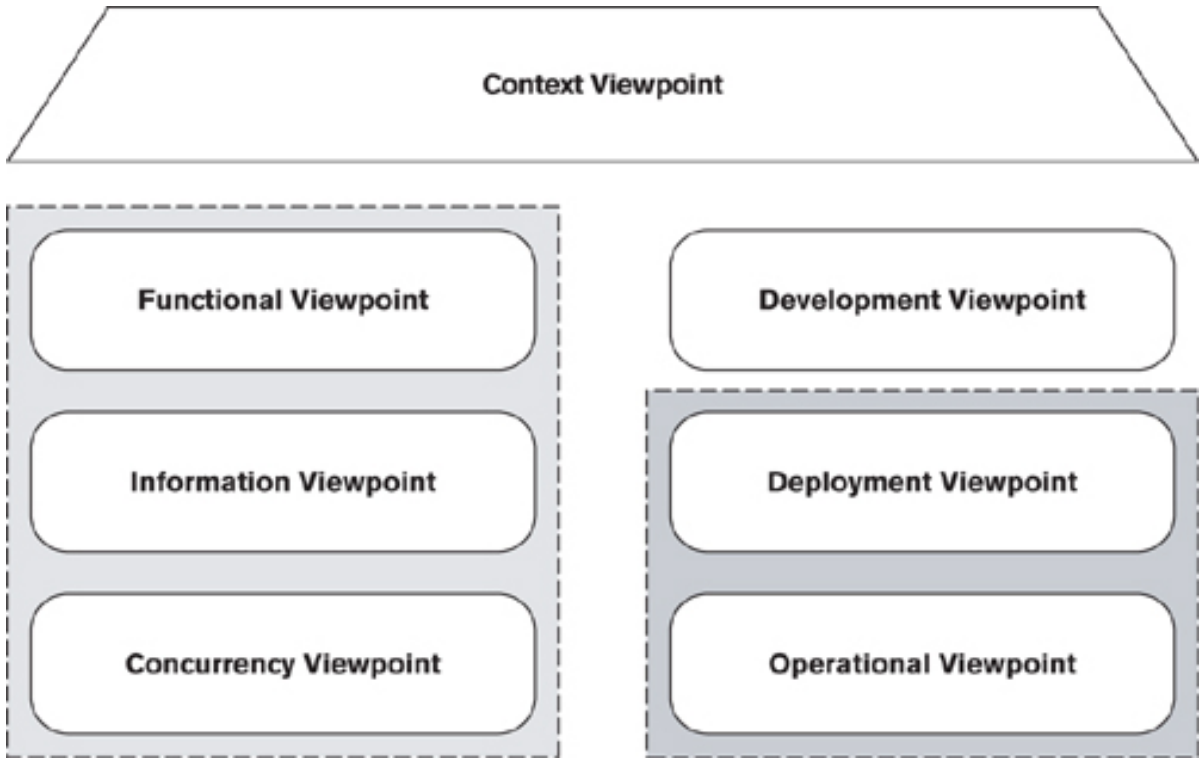


Figura 24: Agrupamiento de puntos de vista propuesto por Rozansky. [Fuente: (Rozanski, 2011)]

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

Plantilla para definir un punto de vista

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

- Debería ser el primero en ser descrito aunque muchas veces no se incluye en la DA, quizás añadiendo sólo un mero “diagrama de contexto”
- 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

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

Las inquietudes más importantes desde este punto de vista son:

- Ámbito del sistema y responsabilidades
- Identificación de las entidades externas (actores), servicios y datos usados
- Naturaleza y características de las entidades externas
- Identidad y responsabilidades de las interfaces externas
- Naturaleza y características de las interfaces externas
- Otras interdependencias externas
- Impacto del sistema en su entorno
- Completitud, consistencia y coherencia global
- Inquietudes para cada parte interesada en el sistema

- Modelos: El modelo más importante es el "modelo de contexto"
- Presenta una imagen global del sistema completo y su relación con el exterior
- Suele incluir los siguientes tipos de elementos:
  - El sistema en sí
  - Las entidades externas
  - Las interfaces
- Notación: UML no tiene diagrama de contexto como tal pero 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, usando el estereotipo <<external>>.
  - Las entidades externas que representan a usuarios que interactúan con el sistema serán representadas siempre como actores.
  - 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).

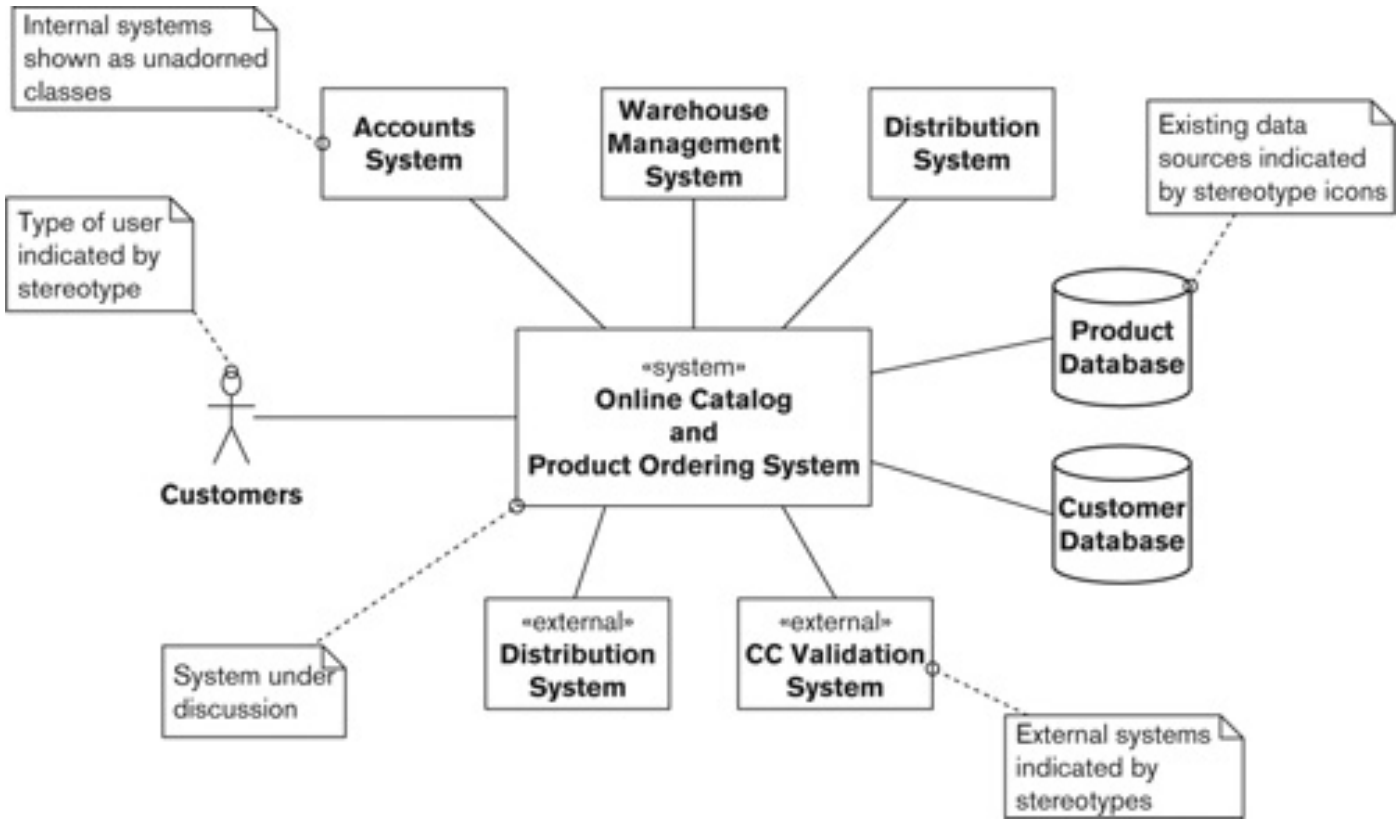


Figura 25: Un diagrama de contexto elaborado con UML. [Fuente: (Rozanski, 2011)]

Nota: Obsérvese que falta la navegabilidad en las líneas

- Notación alternativa: usar diagramas informales con notación de "cajas y líneas"
- Ventajas:
  - Puede ser más simple de utilizar
  - Puede ser mucho más expresivo que UML y más fácil de entender por parte de algunos de las partes interesadas en el sistema
- Inconveniente: Obliga a explicar la notación

Ejemplo de "modelo de contexto" usando notación de "cajas y líneas"

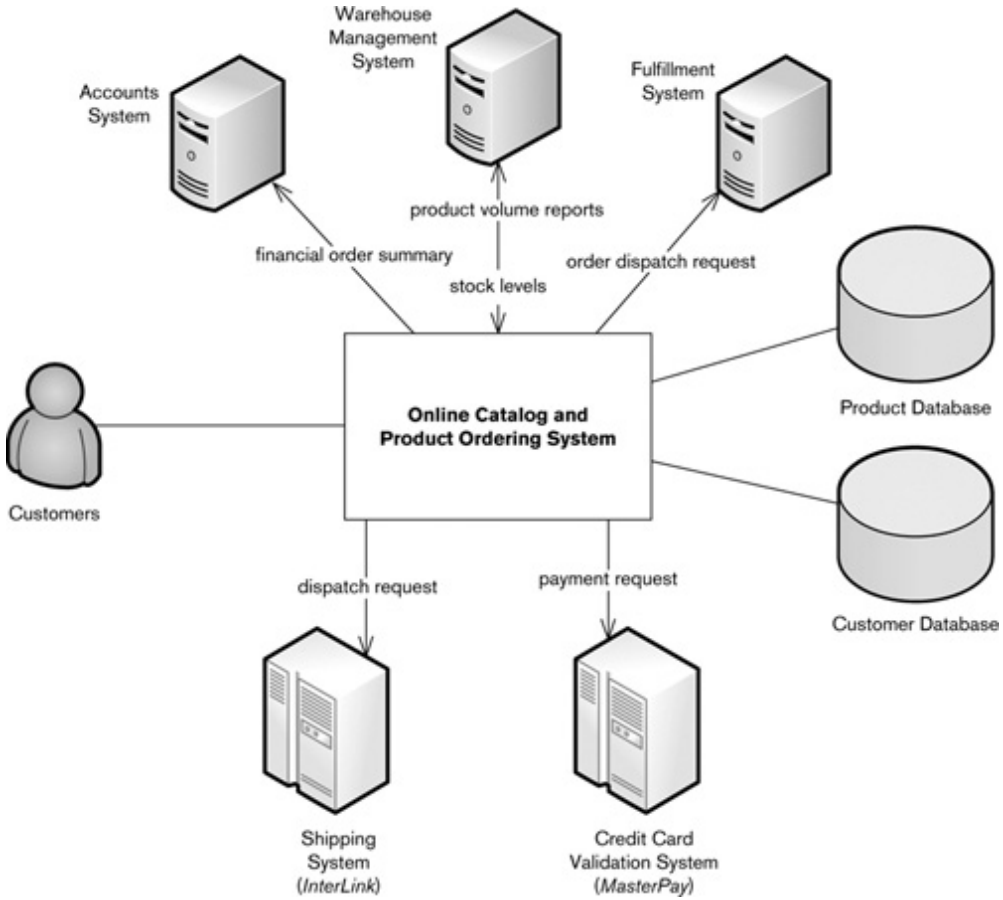


Figura 26: Un diagrama de contexto elaborado de forma libre con "cajas y líneas". [Fuente: (Rozanski, 2011)]

- 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

A veces es útil modelar algunas interacciones entre el sistema y entidades externas que sean especialmente clarificadoras, mediante la descripción de un "escenario arquitectónico".
Cómo describir un escenario:

- Basta con usar listas de interacción textual (a diferencia de cuando se definen los casos de uso completos), o bien diagramas de secuencia de UML
- 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

- 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

- 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 verificación

- ¿Has consultado con todas las partes interesadas quiénes están interesadas 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 todas 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?

Lista de verificación (cont.)

- ¿Se coloca el 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

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 distintas partes interesadas Nivel de detalle erróneo "Elementos divinos" Demasiadas dependencias
Partes interesadas	Todos
Aplicabilidad	Todos los (sub)sistemas

Tabla 4: Descripción del punto de vista funcional. [Fuente: (Rozanski, 2011)

- Capacidades funcionales.- Qué funciones debe llevar a cabo el sistema y cuáles no
- Interfaces externas.- Datos, eventos y flujos de control desde nuestro sistema hacia otros sistemas o viceversa. 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 (entidad monolítica vs. colección de componentes de bajo acoplamiento; paquetes estándar unidos mediante algún middleware apropiado vs escrito desde cero; las funciones cubiertas por servicios externos vs 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

Inquietudes (cont.)

- Filosofía de diseño funcional.- 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[1], flexibilidad funcional[2] 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

[1] Facilidad para añadir nueva funcionalidad.

[2] Facilidad para modificar en el futuro las funciones ya proporcionadas.

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 (cont.)

Tipo de parte interesada	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 5: Inquietudes de los distintos tipos de partes interesadas en el punto de vista funcional. [Fuente: (Rozanski, 2011, Cap. 17)]

Elementos: Contiene normalmente los siguientes elementos:

- Elementos funcionales.- Una parte del sistema ejecutable y bien definida con responsabilidades específicas y que presenta interfaces bien definidas que permiten su coneción con otros elementos (distintos niveles de abstracción desde un módulo software a un paquete, almacén de datos o todo un sistema completo.
- Interfaces.- 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.-Unen los distintos elementos para que puedan interactuar, definiendo la interacción entre los elementos que lo usan y permitiendo 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.

El nivel de especificación de los conectores depende del tipo de interfaz: Si se usa RPC, bastará 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.- No consideradas desde el punto de vista funcional, sino desde los de contexto, concurrencia y despliegue (en estos últimos para especificar cómo se ejecuta en hebras o en procesos y cómo se empaqueta, respectivamente.
  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.

- Diagrama de componentes UML.- La mayor ventaja 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.
- Otras notaciones formales de diseño.- 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. Débiles para describir los conceptos que son importantes para los arquitectos. 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): admiten directamente los conceptos que interesan a los arquitectos. Se ha creado una gran cantidad de ADLs (incluidos Unicon, Wright, xADL, Darwin, C2 y AADL). Casi todos 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

- Diagramas de "cuadros y líneas": deben mostrar sólo los elementos funcionales y sus interfaces y deben 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 29 presenta el diagrama de "cuadros y líneas" equivalente al diagrama UML de la Figura 28. 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. Al igual que con el diagrama de "cuadros y líneas", puede utilizarse un boceto como añadido a una notación de vista más formal (como UML) y utilizando diferentes anotaciones para diferentes grupos de partes interesadas

EJEMPLO: Sistema de vigilancia de la temperatura

La Figura 27 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.

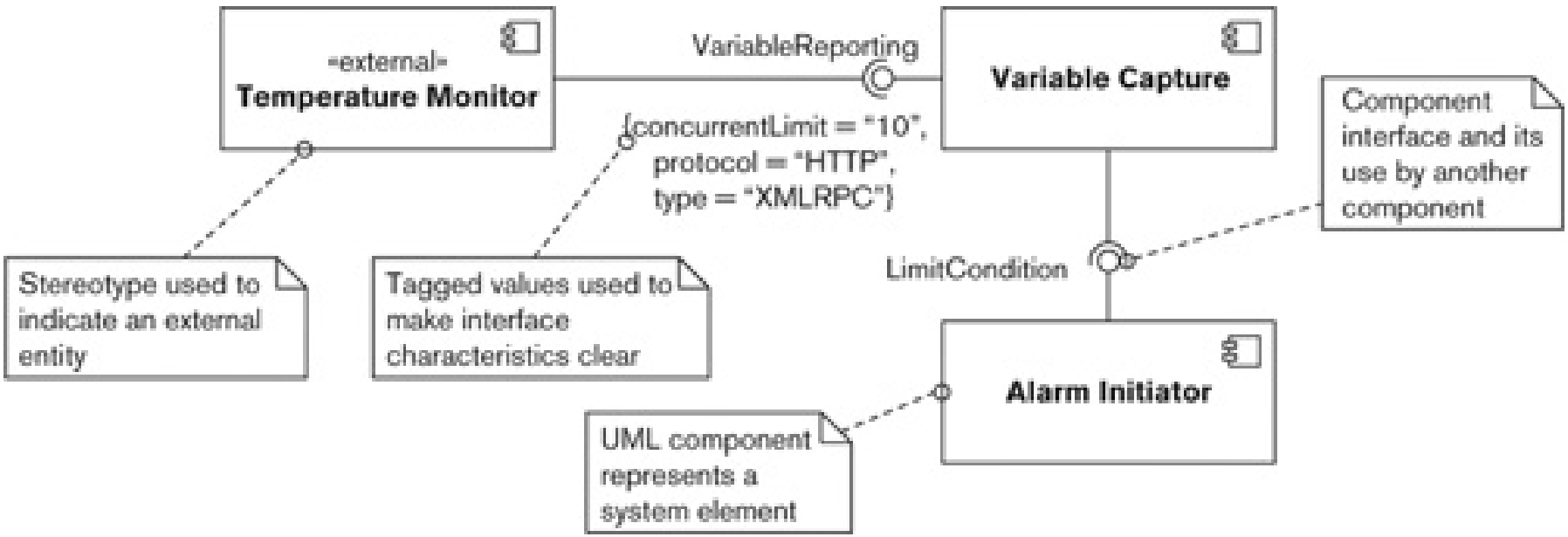


Figura 27: Ejemplo de una Estructura Funcional en UML. [Fuente: (Rozanski, 2011, Cap. 17)]

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

La Figura 28 describe la estructura funcional de una tienda Web para compras a partir de un catálogo en línea y que será parte de un entorno software ya existente en la empresa.
El sistema interactúa con cinco entidades externas: los navegadores web de los tres tipos de usuario más importantes y dos sistemas externos. El sistema en sí está compuesto por cinco componentes principales, unidos por distintos tipos de conectores.
Los clientes hacen el pedido a través de la tienda Web, que interactúa a su vez con el gestor de pedidos, el catálogo de productos y el sistema de información del cliente. Los administradores del catálogo mantienen el catálogo a través de su interfaz basada en la Web y los que forman parte del equipo de servicio al cliente mantienen la información del cliente mediante un programa cliente de interfaz dedicada. Cuando se desea consultar el stock de un producto determinado, el catálogo de productos obtiene la información del inventario, que es un sistema que ya existe.

EJEMPLO: Tienda Web dentro de un entorno software ya existente en la empresa (cont.)

Hay también cierta información en cuanto a la naturaleza de las interacciones entre componentes: pueden acceder al sistema simultáneamente hasta 1000 clientes, 80 personas del servicio al cliente y 15 administradores del catálogo. La interacción entre el catálogo de productos y el inventario tiene lugar mediante un protocolo específico.
Problemas del diagrama: 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.

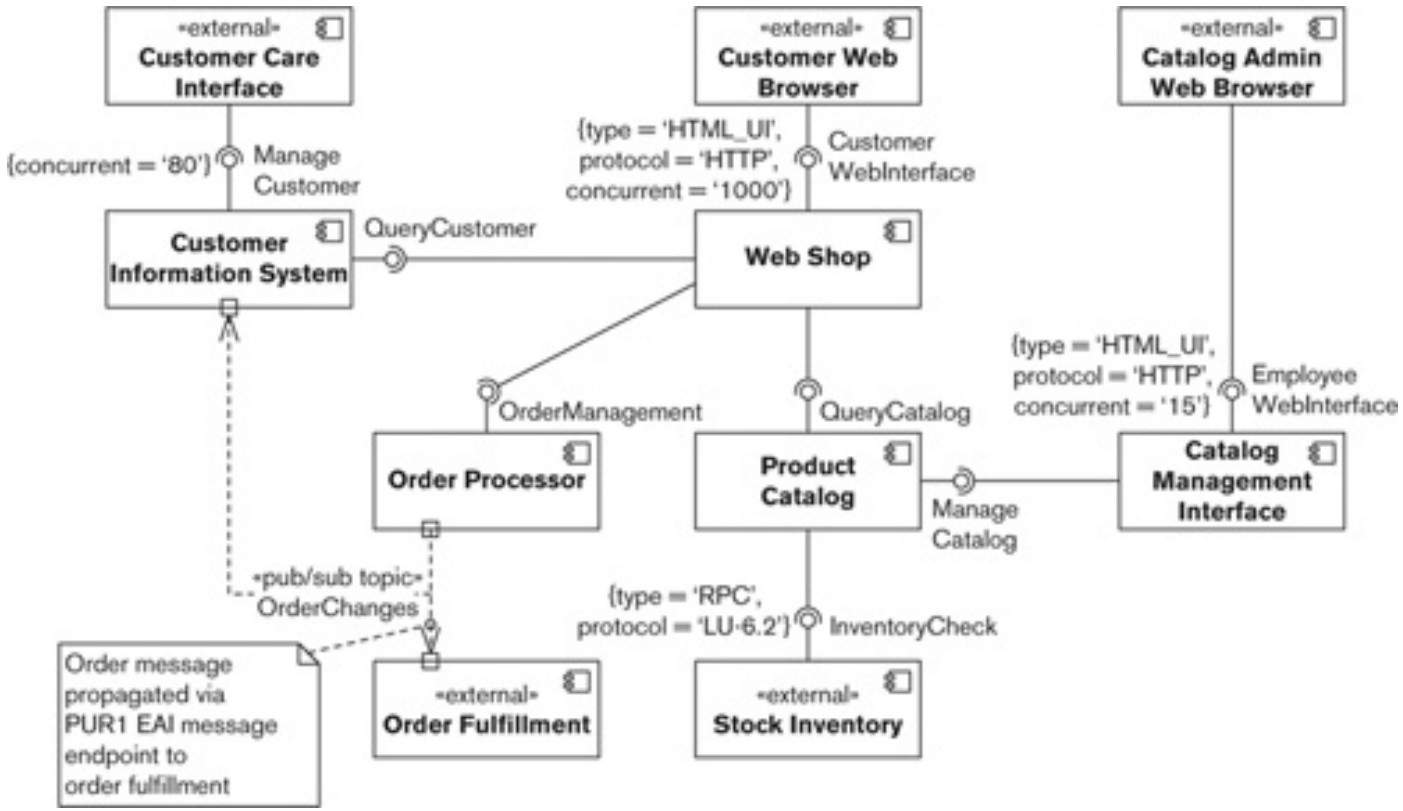


Figura 28: Ejemplo de diagrama de componentes UML para modelar una tienda Web. [Fuente: (Rozanski, 2011, Cap. 17)]

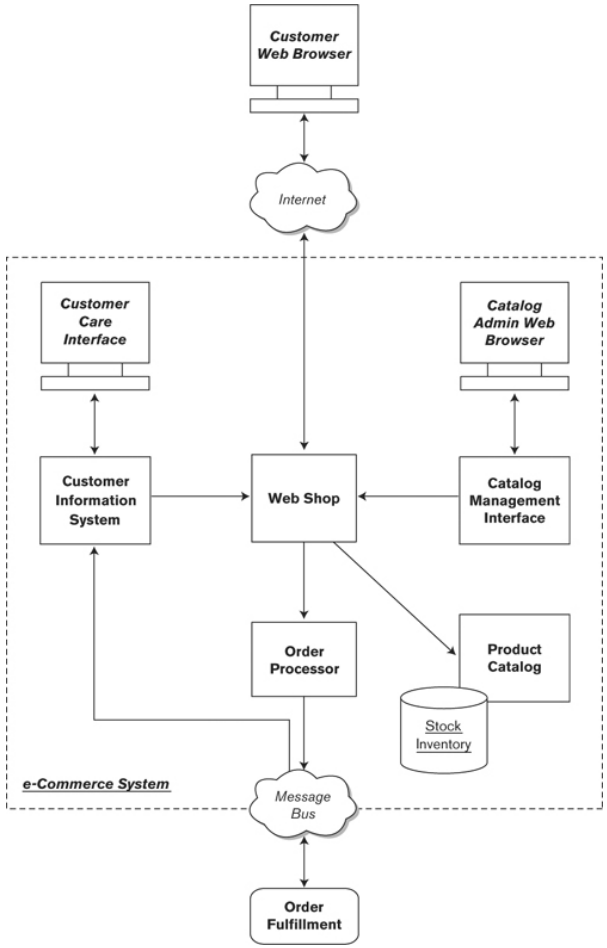


Figura 29: Equivale al diagrama de componentes UML de la Figura 28. Notación propia: icono monitor del ordenador (interfaces externas), rectángulos redondeados (sistemas externos de respaldo), icono tambor de discos (almacenes de datos), nube (interfaces funcionales). El ámbito del sistema incluye a todos los elementos dentro del rectángulo de línea discontinua. [Fuente: (Rozanski, 2011, Cap. 17)]

Identificar los elementos.- aunque el enfoque de desarrollo de software que estemos usando (procedural clásico, OO o enfoque basados en componentes) influye en la identificación de componentes de diferentes maneras, en todos ellos pueden seguirse los siguientes pasos:

1. Trabajar a partir de los requisitos funcionales, derivando responsabilidades clave
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 sea sólida, usando uno o más métodos de refinamiento entre los siguientes:
   - Generalización: permite la reutilización de activos de software en una serie de productos o sistemas similares
   - Descomposición: en sistemas grandes, permite dividir los elementos funcionales de nivel superior en elementos a nivel de subsistema más manejables
   - Composición: a veces es bueno 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: aumenta el rendimiento pero se hace a costa de un enorme riesgo de inconsistencia en los componentes duplicados

Actividades: 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

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 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 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 y sus cambios de estado

Debe incluir entrada, salidas, condiciones previas y efectos de cada operación; y la naturaleza de la interfaz (mensajería, RPC, servicio web, etc.)
Se recomienda enfoque de "Diseño por Contrato" que utilizan condiciones previas, condiciones posteriores e invariantes para definir con precisión el comportamiento y las relaciones de cada operación.
Algunas notaciones comunes de definición de interfaz:

- Lenguajes de programación.- 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 (IDLs).- 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.
- Enfoques orientados a datos.- Describen las interfaces únicamente en términos de mensajes que se intercambian (por ejemplo, interfaces orientadas a documentos y basadas en servicios web con mensajes definidos mediante el esquema XML). Recomendado en interfaces basadas en eventos definidas en términos del intercambio de eventos comerciales

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)

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

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

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

- 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 30 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 representan las 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.- 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, 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 muy bajas de diseño (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".- Cuando un solo elemento acumula 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 31 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

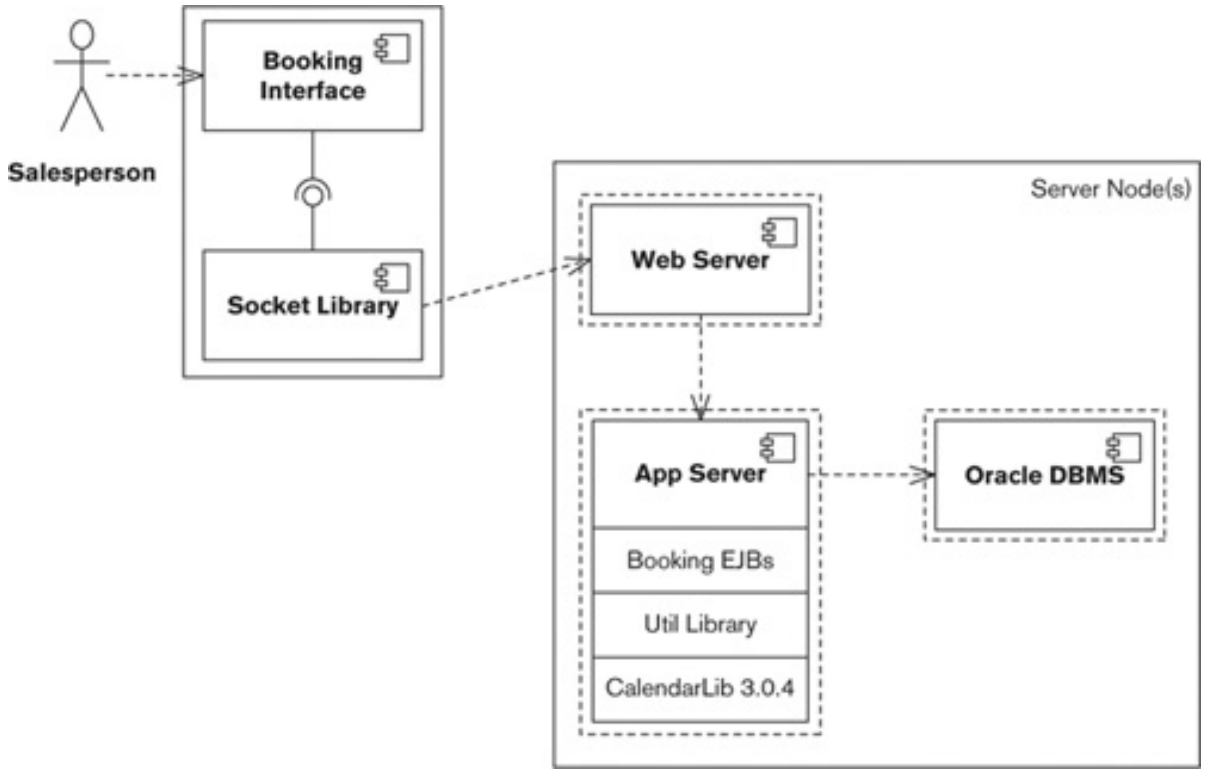


Figura 30: Ejemplo de diagrama de una vista funcional sobrecargada. [Fuente: (Rozanski, 2011, Cap. 17)]

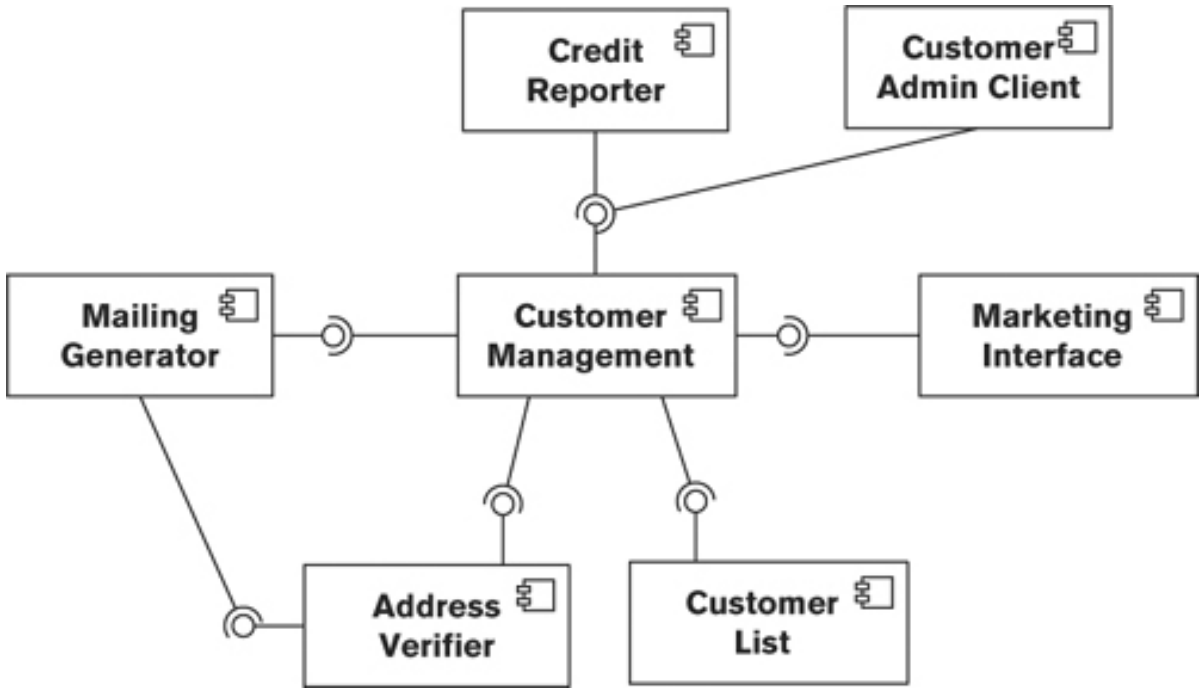


Figura 31: Ejemplo de diagrama funcional que adolece del problema del elemento divino. [Fuente: (Rozanski, 2011, Cap. 17)]

Lista de verificación

- ¿Tiene 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?
- ¿Ha identificado los escenarios de uso importantes y los ha utilizado para validar la estructura funcional del sistema?
- ¿Ha verificado la cobertura funcional de su arquitectura para asegúrese de que cumple con los requisitos funcionales?
- ¿Ha definido y documentado un conjunto apropiado de principios de diseño arquitectónico y su arquitectura cumple con estos principios?
- ¿Ha 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?

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.

¿Por qué no es un punto de vista la seguridad? Porque es una inquietud para distintos puntos de vista, entre otros:

- 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.

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.

Perspectivas más importantes

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, desplegado 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 6: Perspectivas más importantes en proyectos de gran envergadura [Fuente: (Rozanski, 2011)]

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 verificación	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 sr concisa. Los detalles adicionales pueden describirse en otro lugar y ser referidos aquí

Tabla 7: Plantilla para describir la aplicación de una perspectiva en un proyecto concreto [Fuente: (Rozanski, 2011)]

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

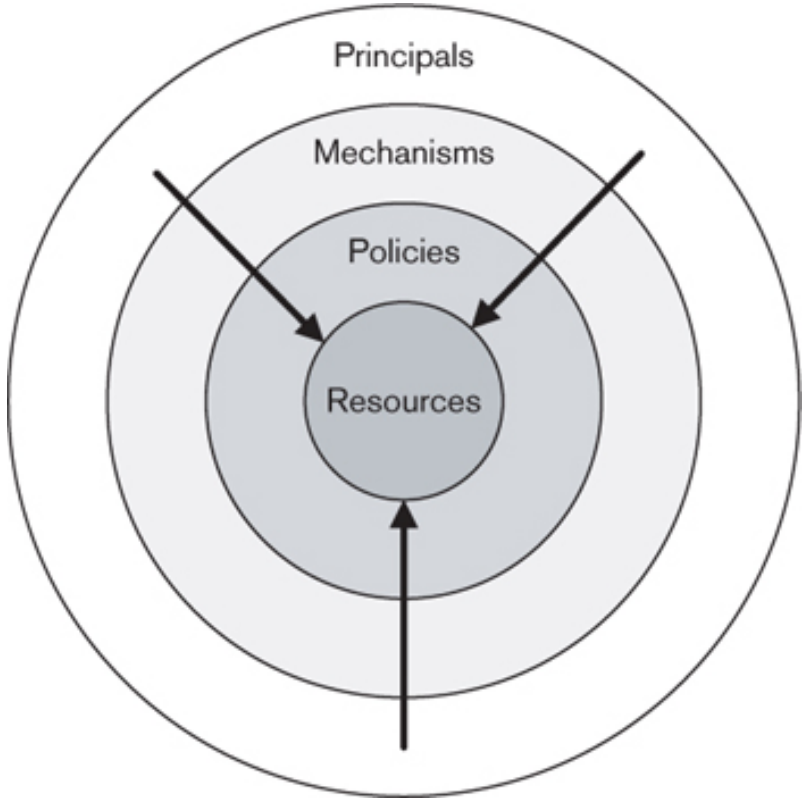


Figura 32: Relación entre principales, políticas, mecanismos y recursos. [Fuente: (Rozanski, 2011)]

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.

Recordatorio: plantilla para describir una perspectiva

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 verificación	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 8: Plantilla para describir la aplicación de una perspectiva en un proyecto concreto. [Fuente: (Rozanski, 2011)]

Aplicabilidad

Punto de vista	Aplicabilidad
Contextual	Permite identificar claramente las conexiones externas al sistema y cómo deben protegerse de un uso maliciosa 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 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

- Recursos
- Principales
- 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
- Integridad
- Disponibilidad
- Trazabilidad
- 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

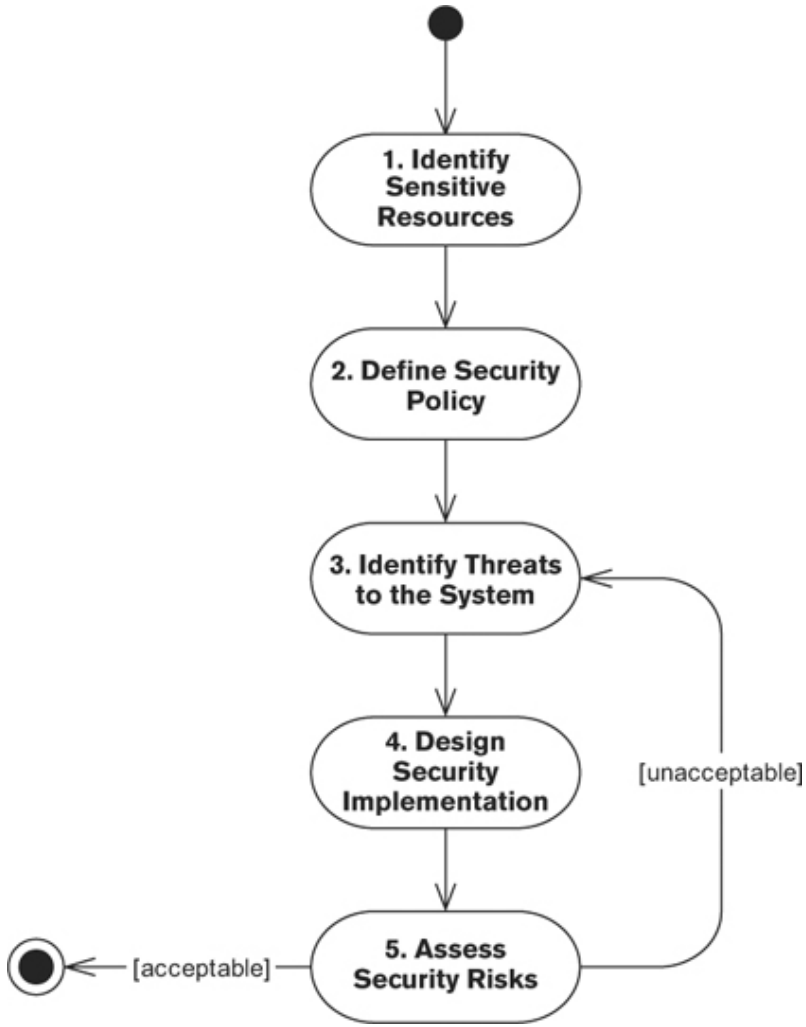


Figura 33: Secuencia de actividades a realizar para aplicar la perspectiva de seguridad. [Fuente: (Rozanski, 2011)]

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 por tipo de recurso y tipos de principales autorizados). 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 amenazas	Identificar las amenazas Caracterizar las amenazas	Tablas/texto "Árbol de ataques"[3]
4. Diseñar implementación de 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 riesgos	Establecer los riesgos de seguridad	Tablas

Tácticas arquitectónicas

1. Aplicar los principios de seguridad que gocen de reconocimiento. Ej:
   - 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

- 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

1. Lsita 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 cuando 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 d 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?

2. Lista 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?

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.

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 Evolución 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:

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

Cómo afecta la evolución a los distintos puntos de vista:

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

Inquietudes

- Gestión de productos.- 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
  - Evolución de la plataforma
  - Evolución de la integración
  - Crecimiento de uso

Inquietudes

- Probabilidad del cambio.- Puesto que ser flexible a los cambios agrega complejidad y gastos, es importante poder afinar estas probabilides
- Temporización del cambio.- 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
  2. Crear el sistema más simple posible para satisfacer las necesidades inmediatas y enfrentar el desafío de hacer cambios solo cuando sea absolutamente necesario (la idea de las metodologías ágiles)

  Conseguir el equilibrio correcto entre estas dos posiciones extremas evita el desperdicio del esfuerzo inicial o los enormes costos de cambio posteriores.
- Cambios dirigidos por factores externos.- No todos los cambios están bajo nuestro control o los de las partes interesadas más inmediatas

  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
  - Cambios en las interfaces con entidades externas
  - Cambios en la regulación externa
  - Cambio organizacional

Inquietudes

- Complejidad del desarrollo.-En algunos casos, la complejidad puede 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.- 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

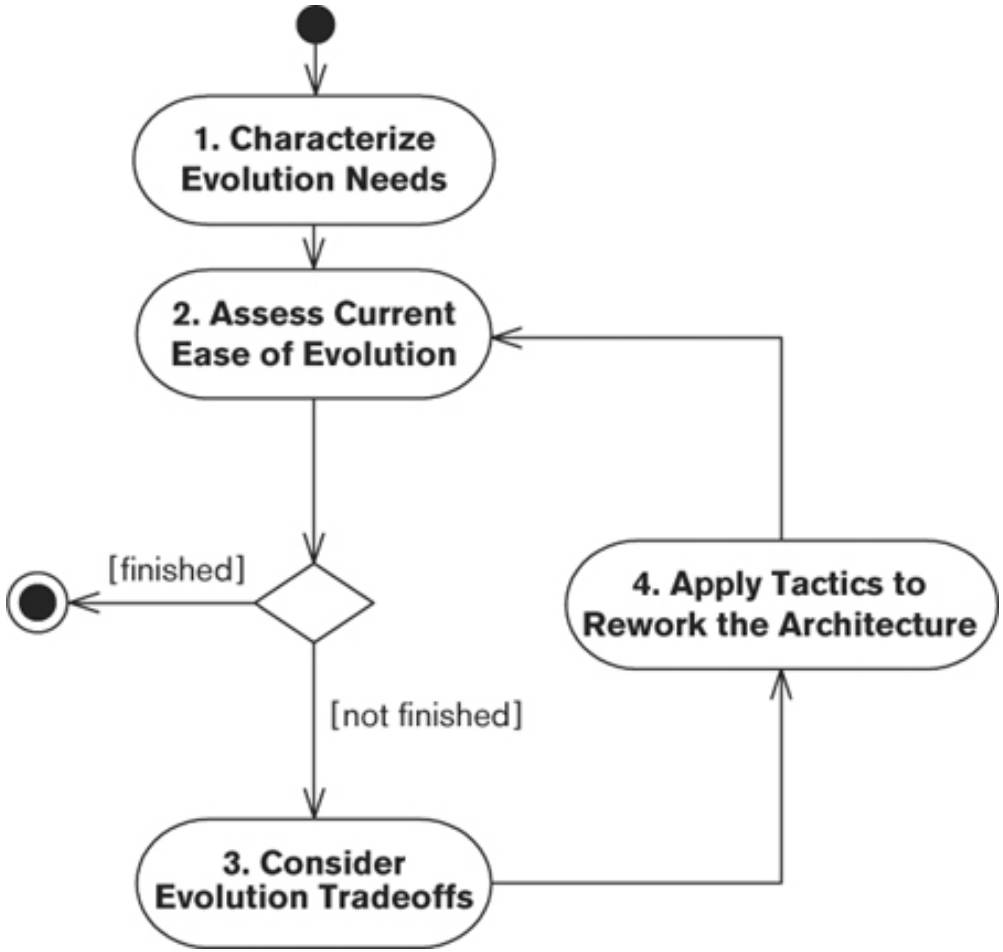


Figura 34: Curso de actividades a realizar para describir un sistema desde la perspectiva de evolución. [Fuente: (Rozanski, 2011, Cap. 28)]

Debemos volver a los requisitos y buscar 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

Caracterizar las necesidades evolutivas

Para cada requisito evolutivo identificado, debemos describirlo utizando 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 solo 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"

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

- 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

Considerando la mejor estrategia de evolución identificada, se deben cambiar las vistas afectadas.

Los siguientes principios generales de diseño pueden ayudarnos a localizar y aislar 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

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

Principios, estilos y patrones de diseño que pueden ayudar a que un sistema sea más fácil de cambiar:

- Patrones de abstracción y estratificación: facilitan el cambio de una parte del sistema con un impacto mínimo en otras
- Patrones de generalización: facilitan el manejo de nuevos casos de uso o tipos de datos
- Patrones de inversión de control (como manejadores de eventos) y de devolución de llamada: ayudan a proteger los elementos de nivel superior de la arquitectura frente a los detalles de implementación de los elementos de nivel inferior

- 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
- 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

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.
En una arquitectura tradicional, al aparecer un nuevo producto, habría que cambiar el sistema para darle cabida.
Una arquitectura basada en metamodelos considera conceptos/entidades fundamentales (clientes, monedas, fechas de negociación y liquidación, límites de negociación, colecciones de productos para un comerciante en particular, etc.). El arquitecto diseña un sistema que proporcione facilidades para implementar los conceptos subyacentes, junto con otras de configuración basada en datos para que los implementadores definan los tipos de productos que desean ofrecer en términos de estos conceptos subyacentes. Cuando se requieran nuevos tipos de productos, se agregarán cambiando los datos de configuración, en lugar del código.

DEFINICIÓN: Puntos de cambio

Lugares del sistema donde puede haber cambios críticos.

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
- Controlar el comportamiento mediante la configuración itemUtilizar datos autodescriptivos (como XML) y procesamiento genérico
- 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
- Dividir los procesos en pasos: podemos introducir un posible punto de cambio si cada paso de un proceso se programa como un elemento separado

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.

- 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

Estrategias para ayudar a que los cambios sean fiables:

- Usar un gestor de la configuración del software
- Incluir un proceso de compilación automatizado
- Automatizar el análisis de dependencias
- Incluir un proceso de lanzamiento automatizado
- Crear mecanismos semiautomáticos para deshacer los lanzamientos fallidos
- Gestionar la configuración del entorno
- Realizar pruebas automatizadas
- Integrar las partes cambiantes de forma continua

- 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

- Priorización incorrecta de las dimensiones
- Programación de cambios que nunca suceden
- Impact negativo de la evolución en criterios críticos de calidad
- Dependencia excesiva de hardware o software específico
- Pérdida del ambiente de desarrollo
- Gestión de lanzamiento ad hoc

- ¿Ha considerado qué dimensiones evolutivas son más importantes para su sistema?
- ¿Confía en que ha realizado un análisis suficiente para confirmar que su priorización de las dimensiones evolutivas es válida?
- ¿Ha identificado cambios específicos particulares que se requerirán y la magnitud de cada uno?
- ¿Ha evaluado la probabilidad de que cada uno de sus cambios sea realmente necesario?

2.Lista de comprobación para la DA

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

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 de un 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,[4] 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?

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

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.

Referencias

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.

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.

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.

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