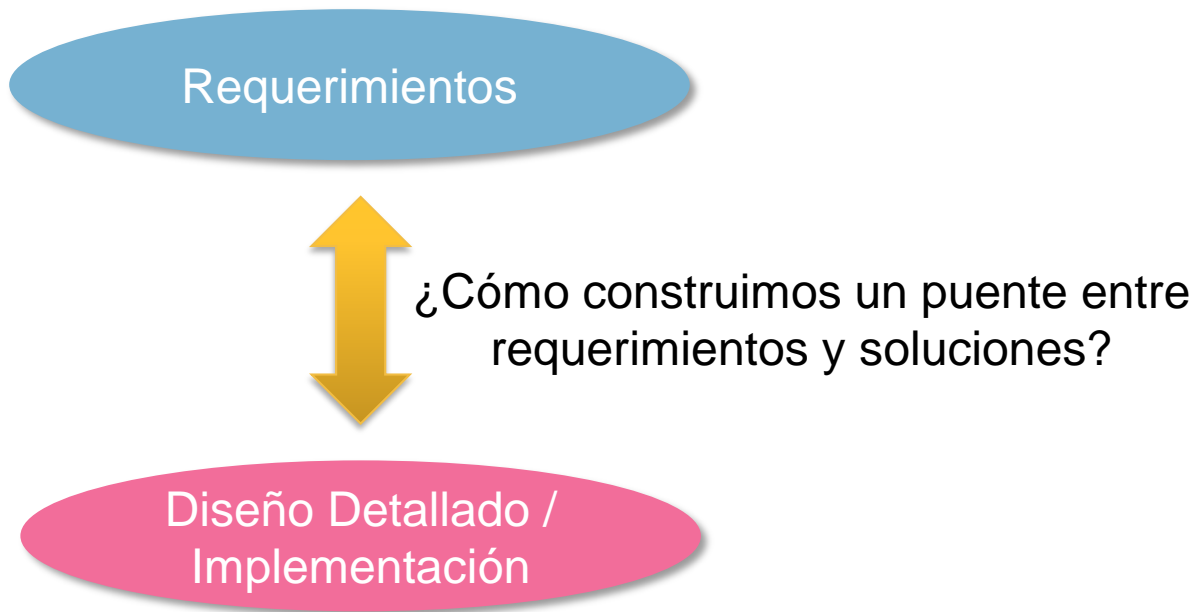




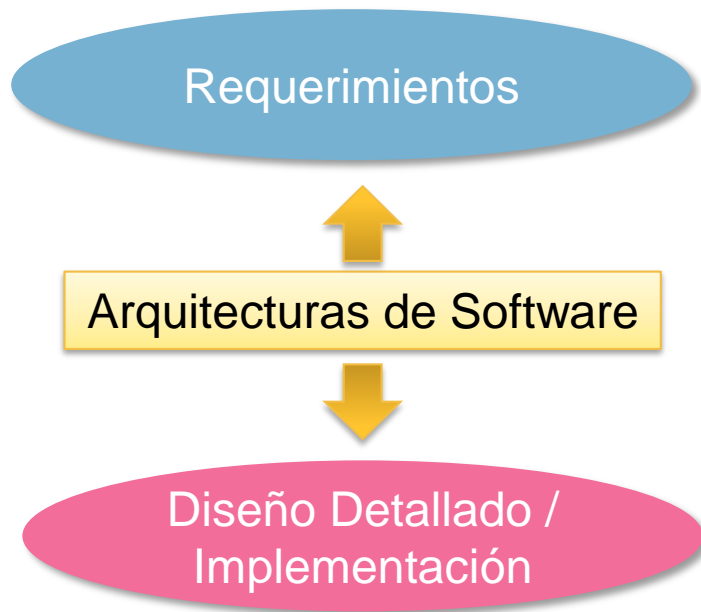
# Arquitecturas de Software



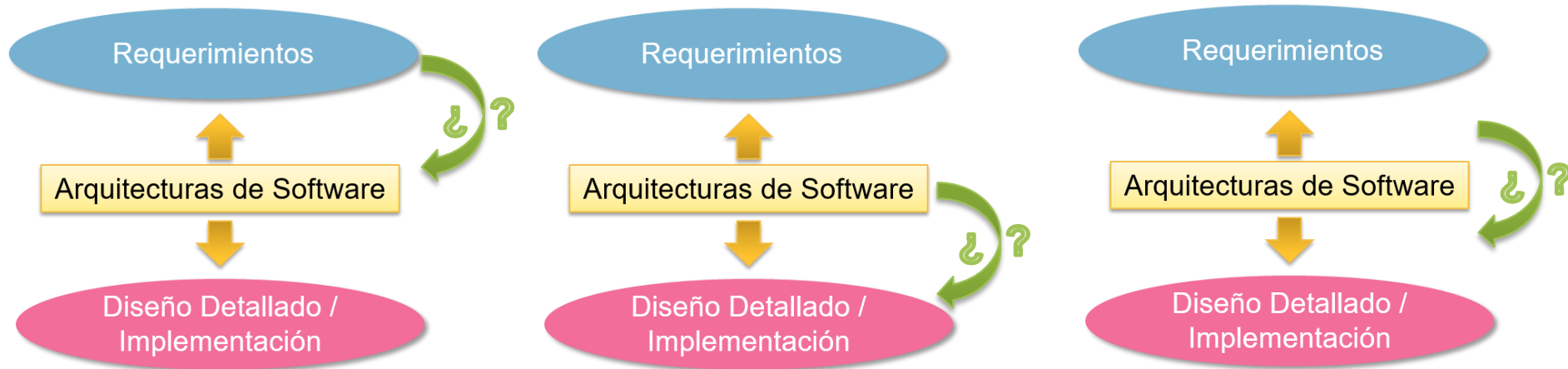
# El Gran Problema



# Una Respuesta Posible



# Que origina nuevas preguntas...



# Reseña Histórica

1968 – Dijkstra introduce la noción de “niveles de abstracción” y de Sistemas operativos organizados en “capas”.

1969 – Conferencia NATO\*, P. I. Sharp sostiene:

*Pienso que tenemos algo, aparte de la ingeniería de software: algo de lo que hemos hablado muy poco pero que deberíamos poner sobre el tapete y concentrar la atención en ello. Es la cuestión de la arquitectura de software. La arquitectura es diferente de la ingeniería. Como ejemplo de lo que quiero decir, echemos una mirada a OS/360. Partes de OS/360 están extremadamente bien codificadas.*

*Partes de OS, si vamos al detalle, han utilizado técnicas que hemos acordado constituyen buena práctica de programación. La razón de que OS sea un **amontonamiento amorfo de programas es que no tuvo arquitecto**. Su diseño fue delegado a series de grupos de ingenieros, cada uno de los cuales inventó su propia arquitectura. Y cuando esos pedazos se clavaron todos juntos **no produjeron una tersa y bella pieza de software**.*

\* La primera conferencia sobre Ingeniería de Software fue en 1968, financiada por la OTAN

# Reseña Histórica

Además sostuvo (1969):

*Lo que sucede es que las especificaciones de software se consideran especificaciones funcionales. **Sólo hablamos sobre lo que queremos que haga el programa.***

***Es mi creencia que cualquiera que sea responsable de la implementación de una pieza de software debe especificar más que esto. Debe especificar el diseño, la forma;** y dentro de ese marco de referencia, los programadores e ingenieros deben crear algo. Ningún ingeniero o programador, ninguna herramienta de programación, nos ayudará, o ayudará al negocio del software, a maquillar un diseño feo.*

*El control, la administración, la educación y todas las cosas buenas de las que hablamos son importantes; pero **la gente que implementa debe entender lo que el arquitecto tiene en mente.***

Luego, por unos años, “arquitectura” fue una metáfora de la que se echó mano cada tanto, pero sin precisión semántica ni consistencia pragmática.

# Reseña Histórica

## 1975 – Brooks\* (diseñador del OS/360)

Utilizaba el concepto de arquitectura del sistema para designar “**la especificación completa y detallada de la interfaz de usuario**” y consideraba que el arquitecto es un agente del usuario, igual que lo es quien diseña su casa. El concepto de AS actual está alejado de esa línea de pensamiento.

Distinguía entre arquitectura e implementación; mientras aquella decía *qué* hacer, la implementación se ocupa de *cómo*.

A diferencia de Dijkstra (formalismo matemático), Brooks considera las variables humanas.

*\*The mythical man-month*

# Reseña **Histórica**

- Década del 70 – **Diseño estructurado** y los primeros **modelos explícitos** de desarrollo de software.
- David Parnas desarrolla temas como:
  - **Descomposición** de Sistemas
  - **Ocultamiento** de la Información
  - **Estructuras** de Software
  - Familias de Programas

*Enfatizando siempre la búsqueda de **calidad del software**, medible en términos de economías en los procesos de desarrollo y mantenimiento.*



# Reseña Histórica

- Decía Parnas:

***“Las decisiones tempranas de desarrollo serían las que probablemente permanecerían invariantes en el desarrollo ulterior de una solución***

*Esas “decisiones tempranas” constituyen de hecho lo que hoy se llamarían **decisiones arquitectónicas.**”*

*La elección de la estructura correcta sintetiza,  
como ninguna otra expresión, el programa y  
la razón de ser de la Arquitectura de Software.*

# Reseña Histórica

- Sobre las familias de programas  
(o un anticipo de los estilos arquitectónico)

“Una familia de programas es un **conjunto de programas** (no todos los cuales han sido contruidos o lo serán alguna vez) a los cuales **es provechoso o útil considerar como grupo**. Esto evita el uso de conceptos ambiguos tales como “similitud funcional” que surgen a veces cuando se describen dominios.

Por ejemplo, los ambientes de ingeniería de software y los juegos de video no se consideran usualmente en el mismo dominio, aunque podrían considerarse miembros de la misma familia de programas en una discusión sobre herramientas que ayuden a construir interfaces gráficas, que casualmente ambos utilizan.”

# Reseña Histórica

- 1984, Mary Shaw

Reinvidica las **abstracciones de alto nivel**, reclamando un espacio para esa reflexión y augurando que el uso de esas abstracciones en el proceso de desarrollo pueden resultar en “**un nivel de arquitectura de software en el diseño**”.

- Mary Shaw. “Abstraction Techniques in Modern Programming Languages”. *IEEE Software*, Octubre, pp. 10-26, 1984.
- Mary Shaw. “Large Scale Systems Require Higher- Level Abstraction”. *Proceedings of Fifth International Workshop on Software Specification and Design*, IEEE Computer Society, pp. 143-146, 1989.

*Años 80': Programación Estructurada → Programación Orientada a Objetos*

# Reseña Histórica

- 1992, Perry y Wolf
  - El primer estudio en que aparece la expresión “arquitectura de software” en el sentido en que hoy lo conocemos.
  - Proponen concebir la AS por analogía con la arquitectura de edificios, una analogía de la que luego algunos abusaron, otros encontraron útil y para unos pocos consideraron inaceptable.
- Presentan un modelo para la arquitectura de software que consiste en tres componentes: elementos, forma y razón (rationale).
  - **Elementos:** son elementos ya sea de procesamiento, datos o conexión.
  - **Forma:** las propiedades de, y las relaciones entre, los elementos, es decir, restricciones operadas sobre ellos.
  - **Razón:** proporciona una base subyacente para la arquitectura en términos de las restricciones del sistema, que lo más frecuente es que se deriven de los requerimientos del sistema.

*La década de 1990, creemos, será la década de la arquitectura de software. Usamos el término “arquitectura” en contraste con “diseño”, para evocar nociones de codificación, de abstracción, de estándares, de entrenamiento formal (de los arquitectos de software) y de estilo.*

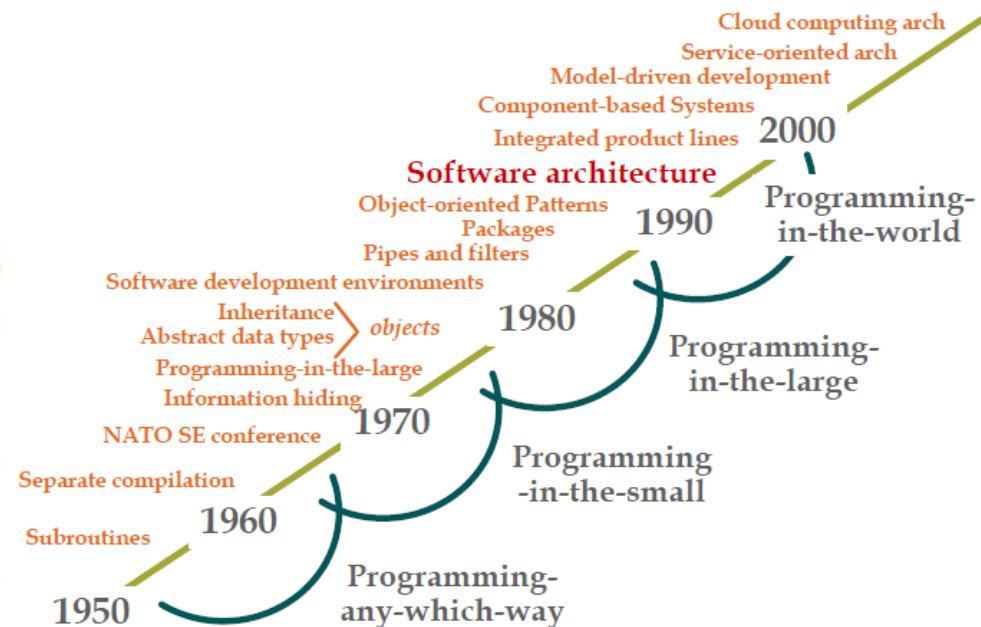
# Reseña Histórica

- 1994, Surge también la programación basada en **Componentes**.
- 1995, Surgimiento de los patrones, cristalizada en dos textos fundamentales:
  - Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Reading, Addison-Wesley, 1995.
  - Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal. *Pattern-oriented software architecture – A system of patterns*. John Wiley & Sons, 1996.
- 1996, Paul Clements, promueve un modelo donde la AS debe ser más de **integración de componentes** pre-programados que de programación.
- Finales de los 90's:
  - Homogenización de la terminología en Arquitectura de Software
  - Tipificación de los estilos arquitectónicos
  - Lenguajes de descripción de arquitectura (ADLs)
  - Se consolidó la concepción de las vistas arquitectónicas, reconocidas en todos y cada uno de los frameworks generalizadores que se han propuesto (4+1, TOGAF, RM/ODP, IEEE)

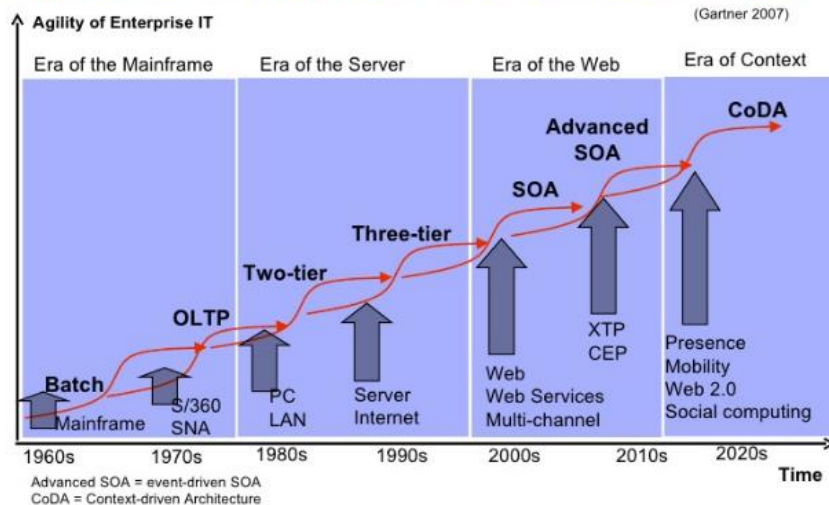
# Reseña Histórica

- 2000, Roy Fielding, presentó el modelo REST, el cual establece definitivamente el tema de las tecnologías de Internet y los modelos orientados a servicios y recursos.
- Se publica la versión definitiva de la recomendación IEEE Std 1471, que :
  - Procura homogenizar y ordenar la nomenclatura de descripción arquitectónica.
  - Homologa los estilos como un modelo fundamental de representación conceptual.
  - ANSI/IEEE 1471-2000, *Recommended Practice for Architecture Description of Software-Intensive Systems*.
- Siguiente paso...
  - Vistas, Viewtypes y estilos...

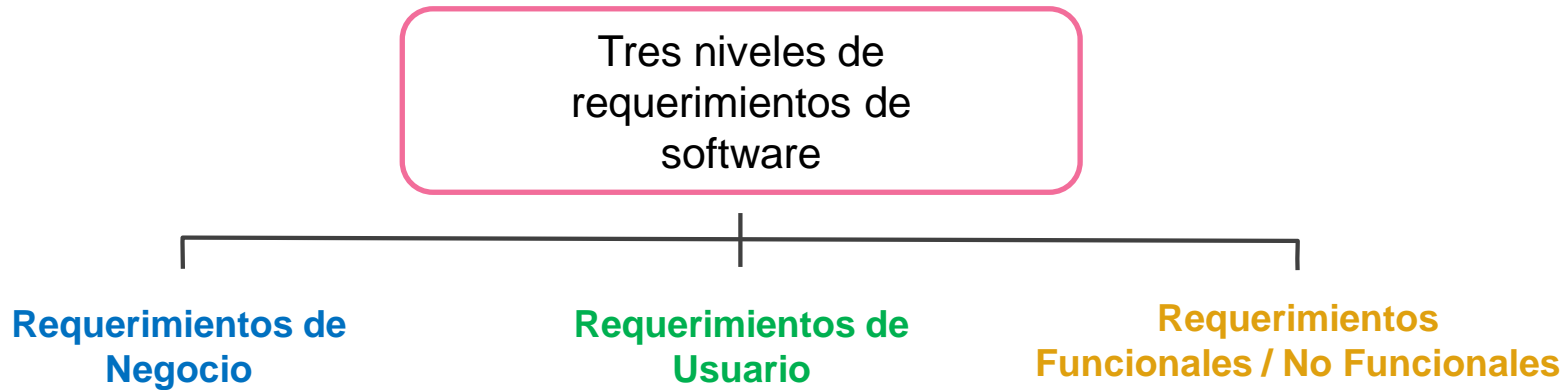
# Contexto



## Software Architecture in the Context of History

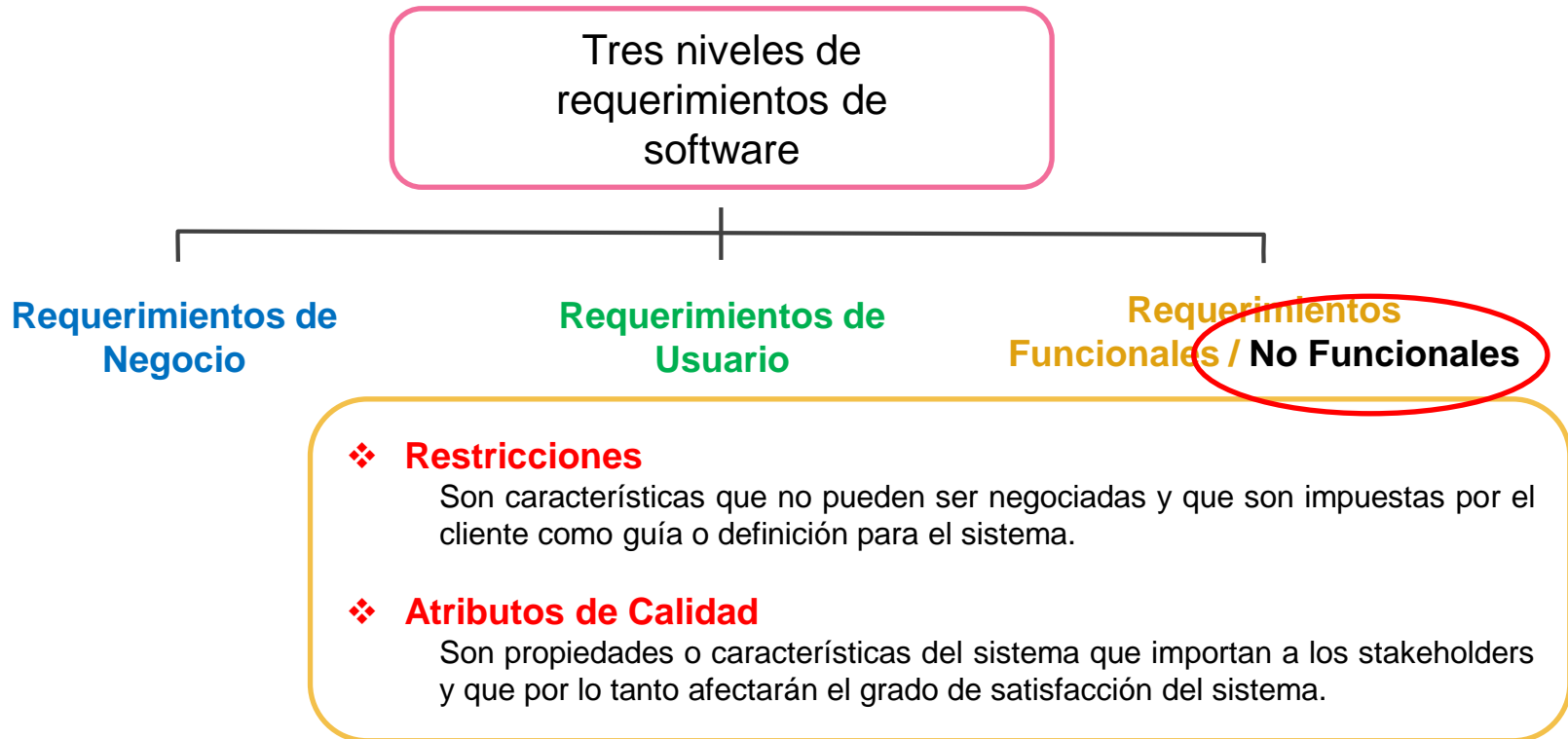


# Requerimientos de Software

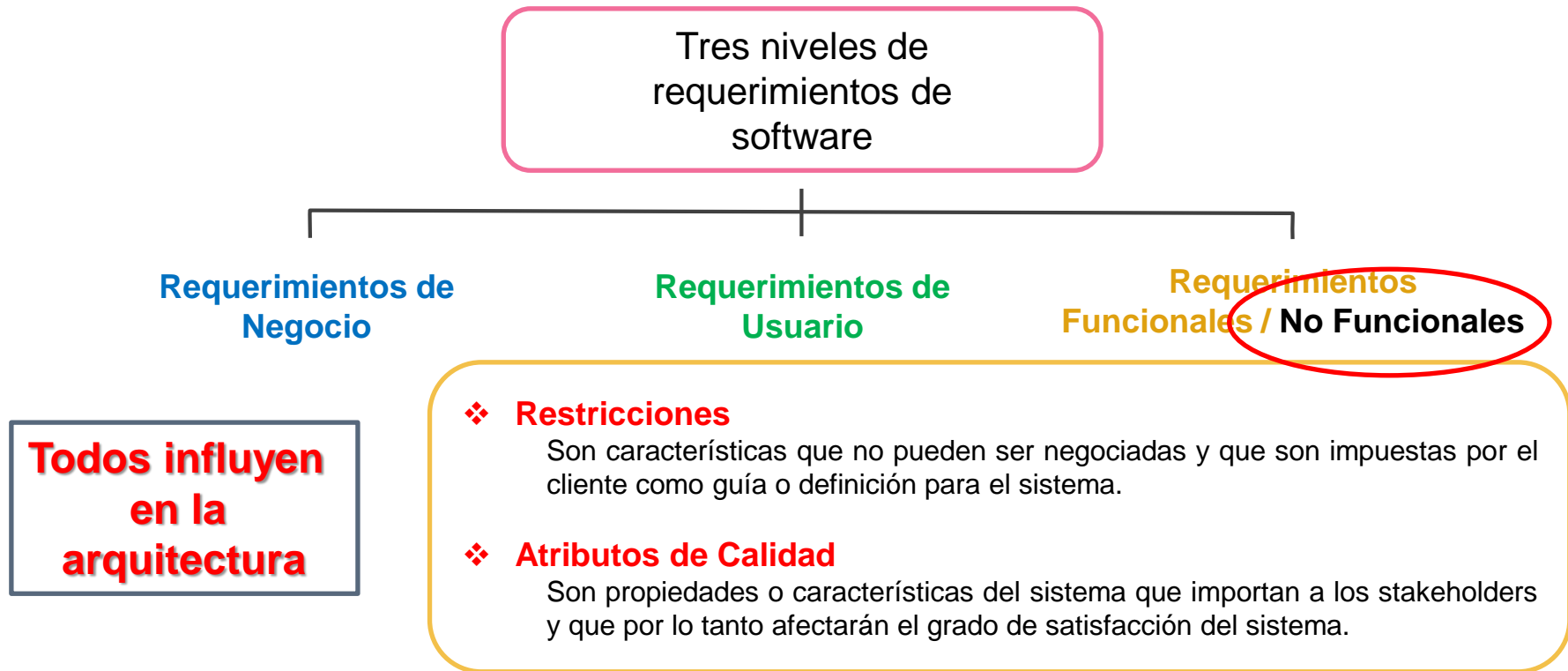




# Requerimientos de Software



# Requerimientos de Software





# Atributos de Calidad

# Atributos de Calidad

- La funcionalidad «de negocio» es sólo una parte de lo que un sistema debe hacer.
- Además, están los atributos de calidad (“ilities”), que hablan de características específicas que debe tener el sistema (anteriormente llamados “requerimientos no funcionales”).
  - Ejemplo: portabilidad, flexibilidad, usabilidad.}
- Necesitamos conocerlos para definir una arquitectura.
- En muchos casos, se afectan entre si. Por ejemplo:  
Portabilidad vs. Performance o Flexibilidad vs. Performance

*“Software quality is the degree to which software possesses a desired combination of attributes.”*

[IEEE Std. 1061]

# Atributos de Calidad

- Suelen estar pobremente especificados, o directamente no especificados (“un requerimiento que no es testeable, no es implementable”).
- En general no se analizan sus dependencias.
- La importancia de estos atributos varía con el dominio para el cual se construye el software.
- Además de requerimientos funcionales y atributos de calidad, el ingeniero de software debe identificar correctamente restricciones.
- Las “tácticas” de arquitectura no son fines en si mismas, son formas de alcanzar atributos de calidad deseados.
- El atributos de calidad que suele ser más importante: la flexibilidad (“facilidad de cambios”).

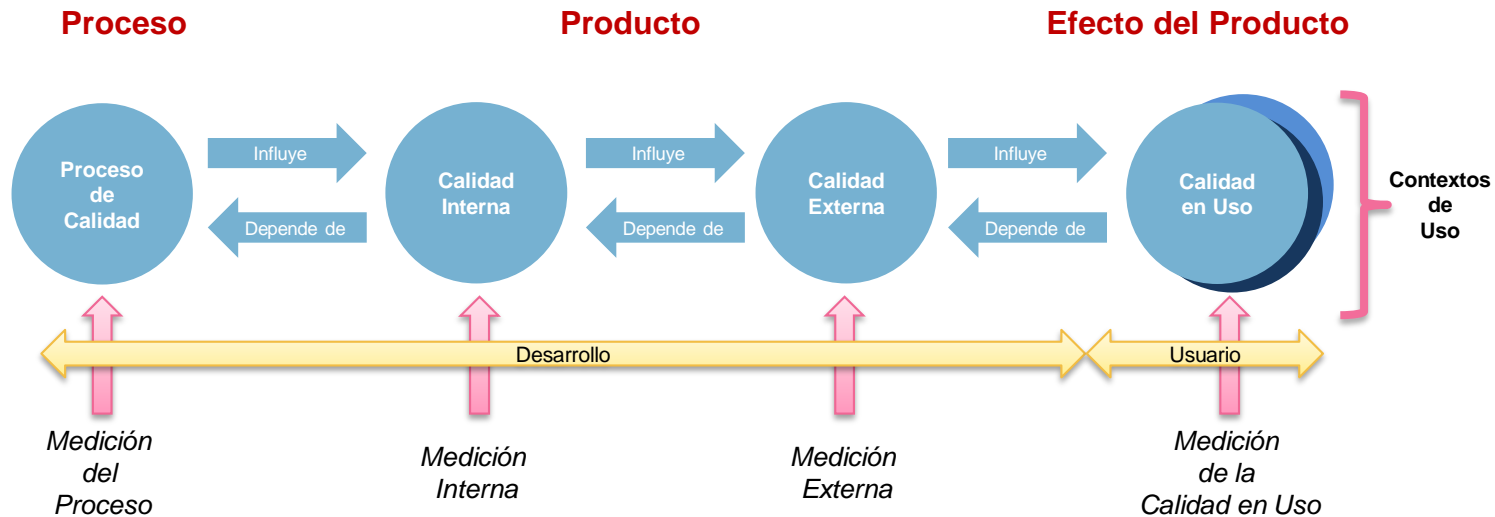
# Atributos de Calidad

✓ Diferentes aspectos de la calidad:

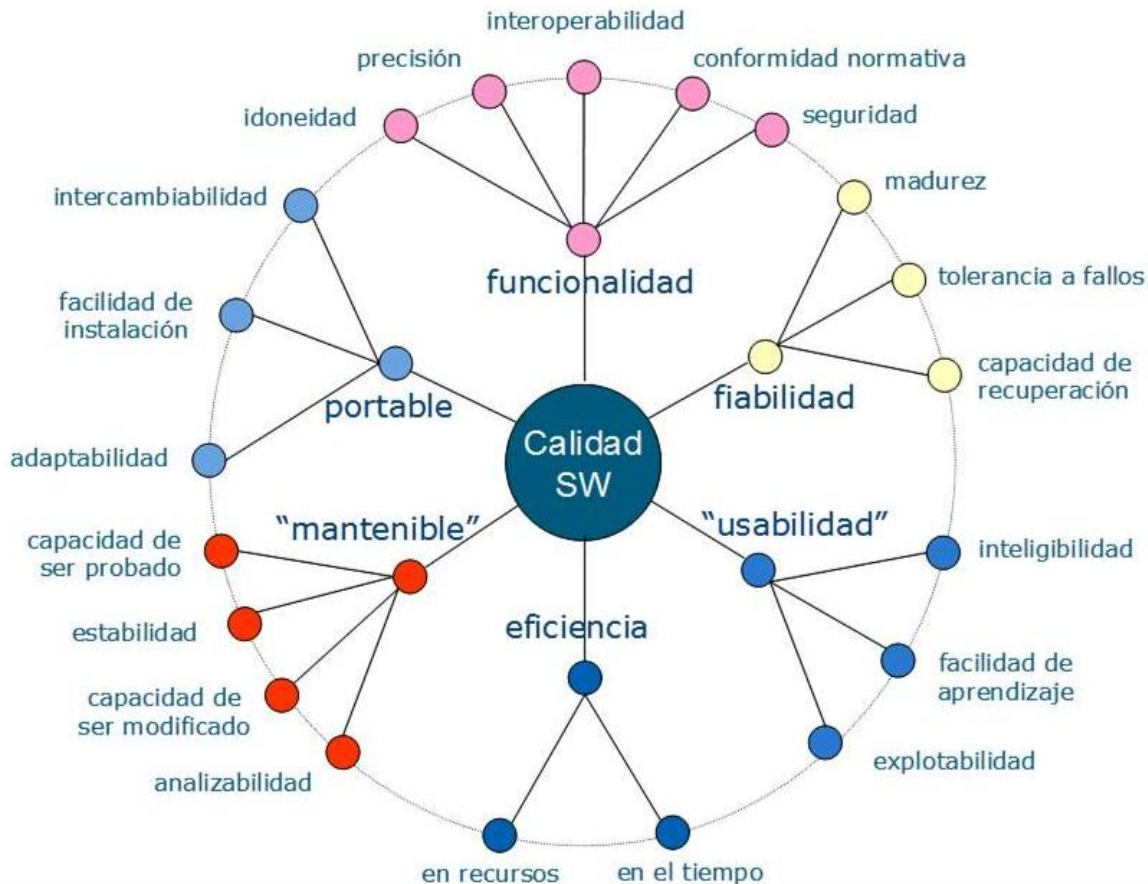
- ☐ **Interna:** medible a partir de las características intrínsecas, como el código fuente.
- ☐ **Externa:** medible en el comportamiento del producto, como en una prueba.
- ☐ **En uso:** durante la utilización efectiva por parte del usuario.

# Atributos de Calidad

- ✓ Distintas clasificaciones de atributos de calidad
  - ✓ Estándares y Certificaciones
    - ✓ SEI, IEEE, etc...



# Atributos de Calidad





# Atributos de Calidad

## Fiabilidad - Disponibilidad

- Relacionada con fallas (“failures”) en el sistema y sus consecuencias asociadas.
- Un “failure” ocurre cuando un sistema no entrega más un servicio de acuerdo con su especificación.
- Esas “failures” son observables por los usuarios (personas u otros sistemas).
- Error <> Defect <> Fault <> Failure
- Tiempo de reparación = tiempo hasta que la falla no es más observable.
- Disponibilidad = probabilidad de que un sistema esté disponible cuando se lo necesite.
- Los “downtimes” programados no se consideran.
- Relativamente fácil de especificar, difícil de verificar.

# Atributos de Calidad

## Facilidad de Cambios

- Relacionada con el costo de los cambios. Uno de los atributos de calidad más difíciles de expresar.
- Temas importantes:
  - ¿Qué puede cambiar?
    - Funcionalidad
    - Plataforma
    - Otros atributos de calidad
    - Interfaces
- ¿Quién y dónde se hace el cambio?
  - Usuarios, desarrolladores, administradores
  - Código, configuración, parametrización
- Una vez que un cambio se especifica, debe ser diseñado, implementado, probado y liberado.

# Atributos de Calidad

## Performance

- Relacionada con el tiempo que le lleva al sistema responder a un evento que ocurre (interrupciones, mensajes, pedidos de usuarios o paso del tiempo).
  - Latencia: tiempo entre la llegada del estímulo y el inicio de la respuesta del sistema
  - “Jitter”: variación en la latencia
  - Deadlines: límites de tiempo para un proceso
  - Throughput: cantidad de transacciones que el sistema puede procesar en un período de tiempo
  - Eventos no procesados
- Difícil de expresar. Depende de volúmenes del sistema, equipamiento en uso y versiones de sistema operativo y otros software de base.

# Atributos de Calidad

## Seguridad

- Habilidad de un sistema para resistir usos no autorizados y seguir proveyendo sus servicios a usuarios legítimos. Algunos temas que incluye:
  - *Nonrepudiation*: mecanismos para asegurar que quienes hicieron algo no puedan negarlo.
  - *Confidencialidad*: propiedad por la cual datos o servicios son protegidos de accesos no autorizados.
  - *Integridad*: propiedad por la cual datos o servicios se brindan como fue previsto.
  - *Disponibilidad* (en el contexto de seguridad): que un sistema esté disponible para su uso legítimo.
  - *Auditabilidad*: habilidad de un sistema para hacer un seguimiento de actividades realizadas.

# Atributos de Calidad

## Usabilidad

- Relacionada con la facilidad con la cual un usuario puede cumplir una tarea o utilizar un servicio ofrecido por el sistema y el tipo de soporte que provee el sistema.
  - Aprender la funcionalidad del sistema.
  - Usar el sistema eficientemente.
  - Minimizar el impacto de los errores.
  - Adaptar el sistema a las necesidades de los usuarios.
  - Aumentar confianza y satisfacción.

# Atributos de Calidad

## Escalabilidad

- Una medida de qué tan bien una solución sigue cumpliendo con sus requerimientos al cambiar los volúmenes del problema que resuelve.

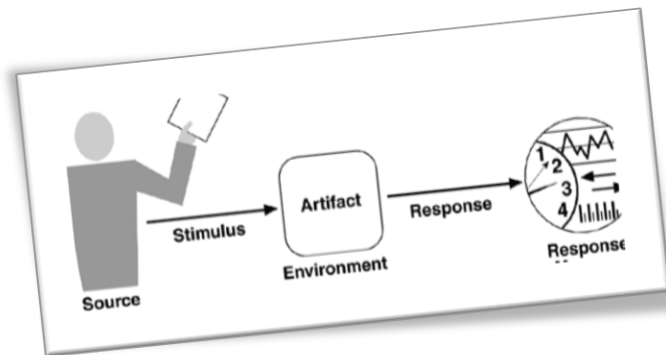
## Portabilidad

- Facilidad de un sistema para poder ser operado en distintas plataformas.

## Facilidad de Testing

- Posibilidad de ver el estado interno de la aplicación.

# Especificación de Atributos de Calidad



## Quality Attribute Scenario (SEI)

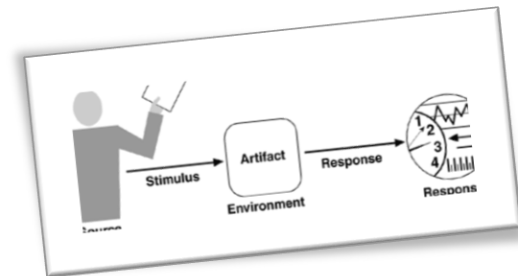
- Fuente del estímulo: Interna o externa
- Estímulo: condición que debe ser tomada en cuenta al llegar al sistema
- Entorno: condiciones en las cuales ocurre el estímulo
- Artifact: el sistema o partes de él afectadas por el estímulo
- Response: qué hace el sistema ante la llegada del estímulo
- Response measure: cuantificación de un atributo de la respuesta

# Especificación de Atributos de Calidad

## Escenario de performance:

*“Se requiere que el sistema sea capaz de realizar las operaciones de autenticación y cobro en a lo sumo 1 segundo ya que, de otro modo, no sería bien recibido por los usuarios, ni los choferes de colectivos. Esto es especialmente crítico en horas pico.”*

- Fuente: Pasajero
- Estímulo: Se aproxima o inserta una tarjeta para realizar el pago de un pasaje
- Entorno: En operación normal y hora pico
- Artefacto: Terminal de Cobro
- Respuesta: Se concreta la operación de la venta del pasaje.
- Medición de la respuesta: tiempo de respuesta total  $\leq 1$  seg.





# La arquitectura de un sistema de software:

- ☐ Define el sistema en términos de elementos e interacción entre ellos.
- ☐ Muestra correspondencia entre requerimientos y elementos del sistema construido.
- ☐ Resuelve atributos de calidad en el nivel del sistema, como escalabilidad, flexibilidad, confiabilidad y performance.

# Analogías con la ingeniería civil...

- ❑ Estilos arquitectónicos: colonial, victoriano, griego
  - Paradigmas de organización de sistemas de software: pipes, layers, events, repositories.
- ❑ Conocimientos específicos para un estilo en particular: cárceles, fábricas automotrices, hospitales, hoteles 5 estrellas
  - Arquitecturas para un dominio específico, llamadas arquitecturas de referencia.

# La estructura de los sistemas

- ❑ La arquitectura trata sobre la estructura de los sistemas
  - ❑ Cómo el sistema se descompone en partes
  - ❑ Cómo esas partes interactúan
- ❑ Pero esto lleva a la pregunta: ¿Qué tipos de estructuras?
  - Del código
  - Run-time
  - De deployment
  - Del entorno de desarrollo
  - Work breakdown structures
- ❑ Cada una de estas estructuras puede ser la base para una vista arquitectónica (architectural view)
  - Históricamente el foco estuvo en vistas de código.

# Las definiciones más aceptadas

(Bass, Clements)

La arquitectura de software de un sistema de computación es el conjunto de estructuras necesarias para razonar sobre el sistema, y comprende elementos de software, relaciones entre ellos y propiedades de ambos.



- ❑ Estilo o patrón arquitectónico
  - ✓ Una descripción de tipos de relaciones y elementos, junto con restricciones sobre cómo deben usarse (ej. “client server”).
  
- ❑ Arquitectura de referencia
  - ✓ Una división común de funcionalidad mapeada a elementos que cooperativamente implementan esa funcionalidad y flujos de datos entre ellos.

# Tres Principios Fundamentales

- ☐ Toda aplicación tiene una arquitectura.
- ☐ Cada aplicación tiene al menos un arquitecto.
- ☐ La “Arquitectura” no es una fase del desarrollo.

La arquitectura es el conjunto de decisiones principales de diseño de un sistema de software.


# Temas fundamentales (Clemens – 1996)

- 
- 
- Diseño o selección de la arquitectura
    - Cómo crear o elegir.
  - Representación de la arquitectura
    - Cómo comunicar.
  - Evaluación y análisis de la arquitectura
    - Cómo validar y verificar.
  - Desarrollo y evolución basados en arquitectura
    - Cómo construir.
  - Recuperación de la arquitectura
    - Cómo descubrir arquitecturas subyacentes (legacy)

# ¿Qué hace que una arquitectura sea “*buen*a”?


- ✓ Producto de un **único arquitecto o un pequeño grupo de arquitectos** con un claro líder (Brooks, Mills y otros). “Integridad conceptual”.
- ✓ El equipo de arquitectura debe contar con **requerimientos funcionales** y **atributos de calidad** requeridos que sean claros.
- ✓ La arquitectura debe estar **documentada**.
- ✓ La arquitectura debe ser **revisada** por los “stakeholders”.
- ✓ Debe ser **evaluada cuantitativamente** antes de que sea tarde.
- ✓ Debe permitir una implementación **incremental**.
- ✓ Módulos bien definidos basados en el **ocultamiento** de la información.
- ✓ Interfaces claramente definidas.
- ✓ Usa un grupo **pequeño y claro** de patrones de interacción.

# Arquitecturas de Software - Características


- 
- **Representación de alto nivel** de la estructura del sistema describiendo las partes que lo integran.
  - Puede incluir los **patrones** que supervisan la composición de sus componentes y las restricciones al aplicar los patrones.
  - Trata **aspectos del diseño y desarrollo** que no pueden tratarse adecuadamente dentro de los módulos que forman el sistema.




# Arquitecturas de Software - Objetivos

- 
- **Comprender** (abstracción) y mejorar la estructura de las aplicaciones complejas.
  - **Reutilizar** dicha **estructura** (o partes de ella) para resolver problemas similares.
  - Analizar la **corrección** de la aplicación y su grado de cumplimiento respecto a los requisitos iniciales.
  - Permitir el estudio de algunas propiedades específicas del dominio.

# Arquitecturas de Software - Objetivos

- 
- Planificar la **evolución** de la aplicación, identificando las partes mutables e inmutables de la misma, así como los costos de los posibles cambios.
  - Facilitar la **adaptación al cambio**:
    - Composición
    - Reconfiguración
    - Reutilización
    - Escalabilidad
    - Mantenibilidad, etc.

# Arquitecturas de Software - Objetivos

- 
- Organización a alto nivel del sistema, incluyendo aspectos como:
    - La descripción
    - Análisis de propiedades relativas a su estructura y control global
    - Los protocolos de comunicación
    - Protocolos de sincronización utilizados
    - La distribución física del sistema y sus componentes, etc

# Arquitecturas de Software – Fuera del alcance



➤ ¿De qué no se ocupa?

- Diseño detallado
- Diseño de algoritmos
- Diseño de estructuras de datos

# Más definiciones aceptadas

(Garlan y Shaw, 1993)

Una colección de componentes computacionales - o, simplemente, **componentes** - en conjunto con una descripción de las **interacciones** entre estos componentes, es decir, de los **conectores**.

- ❑ La arquitectura de software define
  - **Componentes**: lugar de almacenamiento o computo:
    - Filtros, bases de datos, objetos, TADs
  - **Conectores**: Mediadores entre componentes
    - Llamadas a procedimientos
    - Pipes
    - Broadcast
  - **Propiedades**: Información para construcción y análisis
    - Pre/Post condiciones, invariantes

# Vistas

- ❖ Las vistas arquitectónicas representan un aspecto parcial de una arquitectura de software que muestran propiedades específicas del sistema.
- ❖ Una única representación resultaría demasiado compleja.
- ❖ Cada vista representa un comportamiento particular del sistema.
- ❖ La **descripción** de un sistema complejo **no** es **unidimensional**.
- ❖ Surgen de la agrupación de **elementos arquitectónicos** en “tipos”.
- ❖ Relevancia: depende del **propósito**. Por ejemplo:
  - enunciar la misión de implementación,
  - análisis de atributos de calidad,
  - generación automática de código,
  - planificación,
  - etc.

# Vistas

- ❖ Las vistas exponen **atributos de calidad** en diferente grado.  
Ejemplos:
  - Vista modular: portabilidad...
  - Vista de deployment: performance, confiabilidad...
- ❖ Reflejan **decisiones arquitectónicas**.
- ❖ Enfatizan **aspectos** e ignoran otros para que el problema sea abordable.
- ❖ Es clave saber cuáles son las vistas relevantes y vincularlas.
- ❖ **Ninguna vista es “LA” arquitectura.**

# Vistas

- ❖ Las vistas arquitectónicas representan un aspecto parcial de una arquitectura de software que muestran propiedades específicas del sistema.
- ❖ Una única representación resultaría demasiado compleja.
- ❖ Cada vista representa un comportamiento particular del sistema.
  - **Estructura de módulos** (*asignación de trabajo, es parte de o comparte el mismo secreto que*).
  - **Estructura de uso** (*programas, depende de la corrección de*).
  - **Estructura de procesos** (*procesos, brinda trabajo computacional a*).

(David Parnas  
1974)



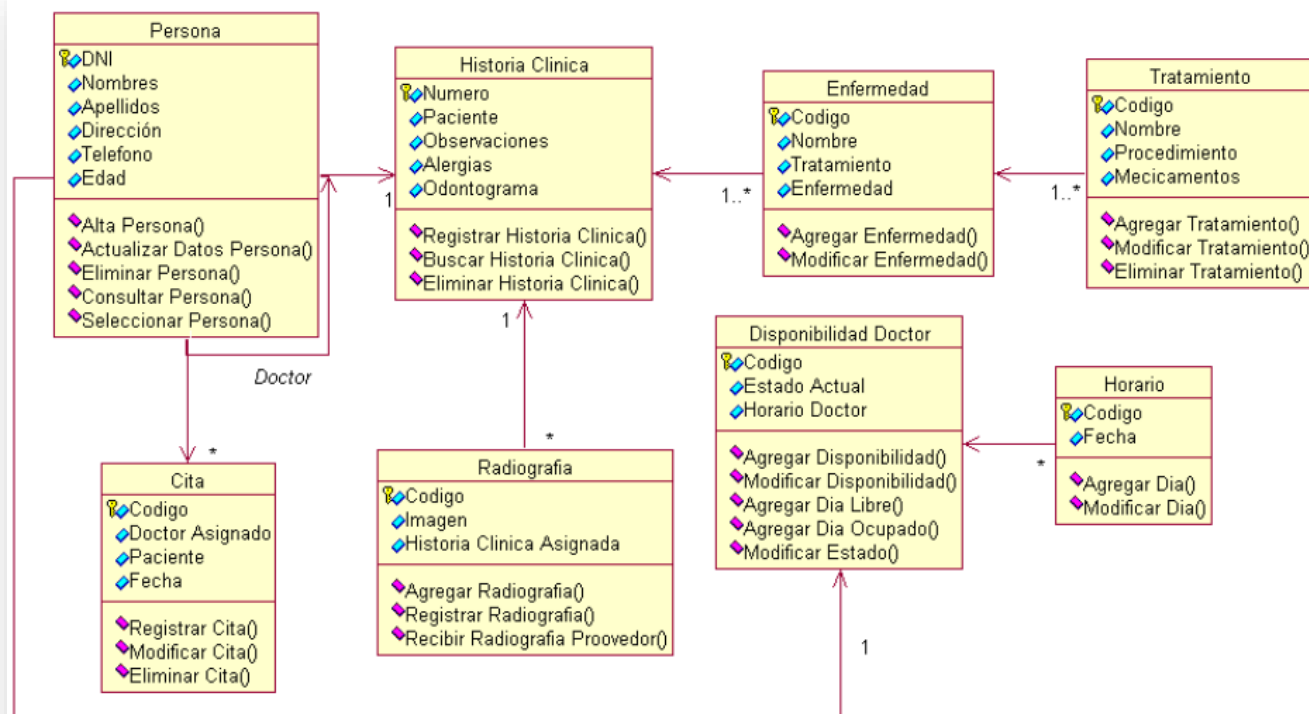
# Vistas

- ❖ Las vistas arquitectónicas representan un aspecto parcial de una arquitectura de software que muestran propiedades específicas del sistema.
- ❖ Una única representación resultaría demasiado compleja.
- ❖ Cada vista representa un comportamiento particular del sistema.

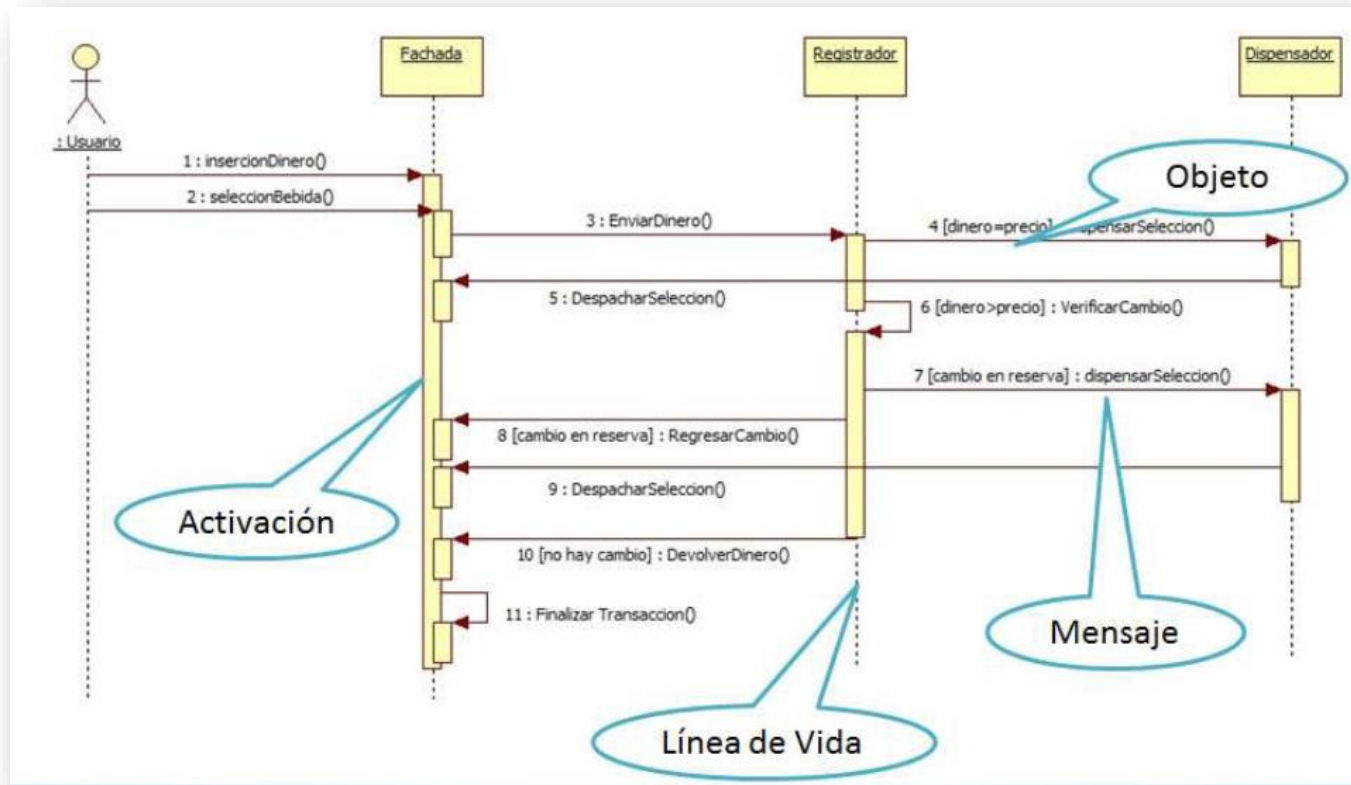
## Vista 4+1 (Kruchten, 1995)



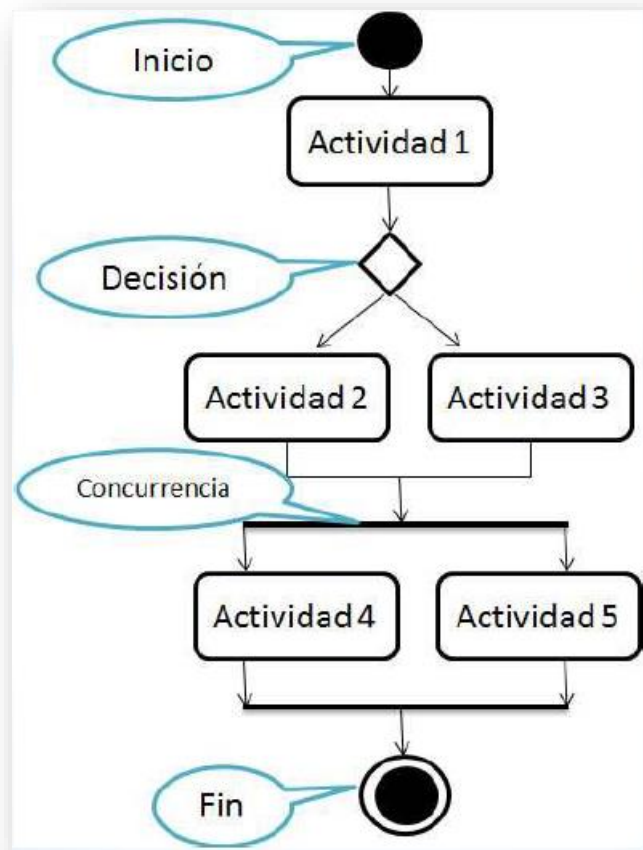
# 4+1 – Vista Lógica



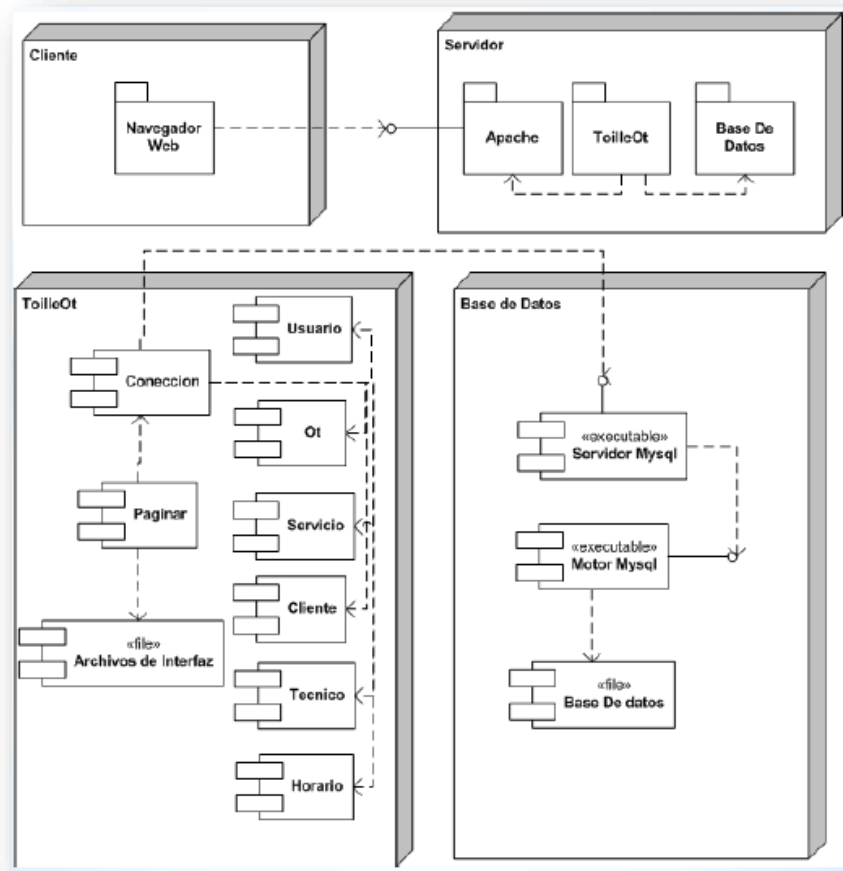
# 4+1 – Vista Lógica



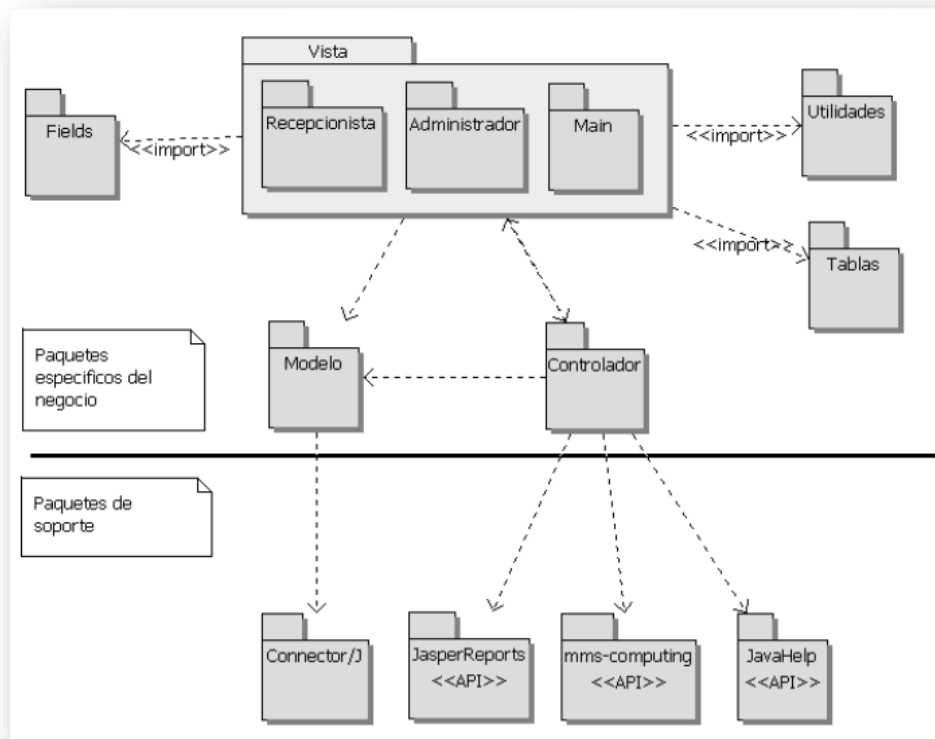
# 4+1 – Vista de Procesos



# 4+1 – Vista de Desarrollo

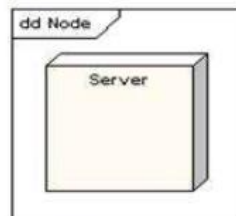


# 4+1 – Vista de Desarrollo

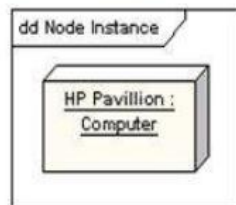


# 4+1 – Vista Física

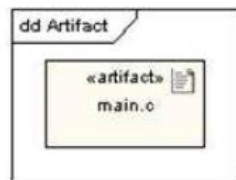
## NODO



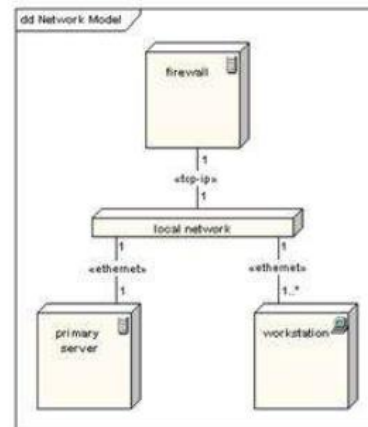
## INSTANCIA DE NODO



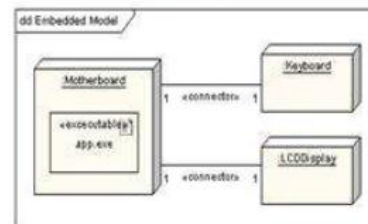
## ARTEFACTO



## ASOCIACION

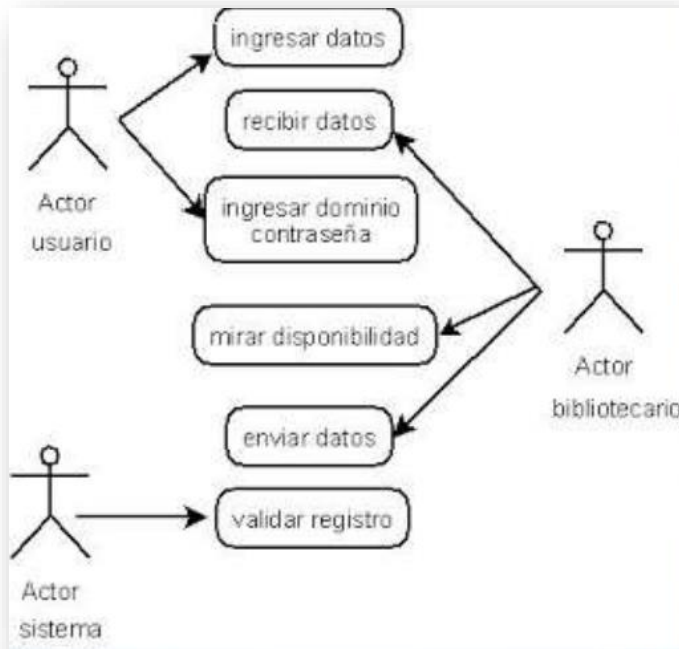


## NODO COMO CONTENEDOR



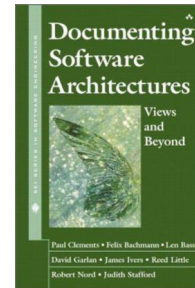


# 4+1 – Escenarios de CU





# Viewtypes



- Un sistema tiene varias estructuras
- Las estructuras pueden dividirse principalmente en tres grupos:
  - *De Módulos*: los módulos son unidades de implementación, una forma de ver al sistema basada en el código (visión estática).
  - *De Componentes y Conectores*: aquí los elementos son unidades de run-time **independientes** y los conectores son los mecanismos de comunicación entre esos componentes (visión dinámica).
  - *Estructuras de asignación o asignación (“allocation”)*: Muestra la relación entre elementos de software y los elementos en uno o más entornos externos en los que el software se crea y ejecuta.

Llamamos “**Viewtypes**” a las vistas de arquitecturas orientadas a estas tres estructuras

# Module **Viewtype**

- ❖ Un **módulo** es una unidad de código que implementa un conjunto de responsabilidades.
  - Una clase, una colección de clases, una capa o cualquier descomposición de la unidad de código.
  - Nombres, responsabilidades, visibilidad de las interfaces.
- ❖ Tipos de diagramas:
  - Descomposición (“es parte de”): los módulos tiene relación del tipo “es un submódulo de”
  - Usos (“depende de”): los módulos tienen relación del tipo “usa a”. Se dice que un módulo A usa a B, si la correcta ejecución de B es necesaria para la correcta ejecución de A (no es lo mismo que invocación)
  - Clases (“se comporta como”): los módulos en este caso son clases y las relaciones son de herencia.

# Module Viewtype

## ➤ Utilidad

### Construcción

---

Organizar el código fuente.

### Análisis

---

Trazabilidad de Requerimientos.

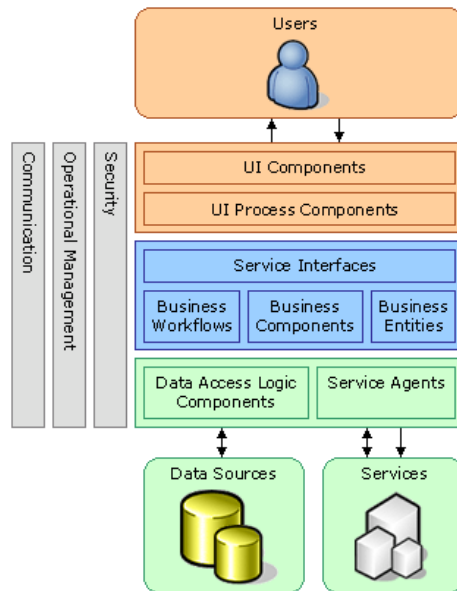
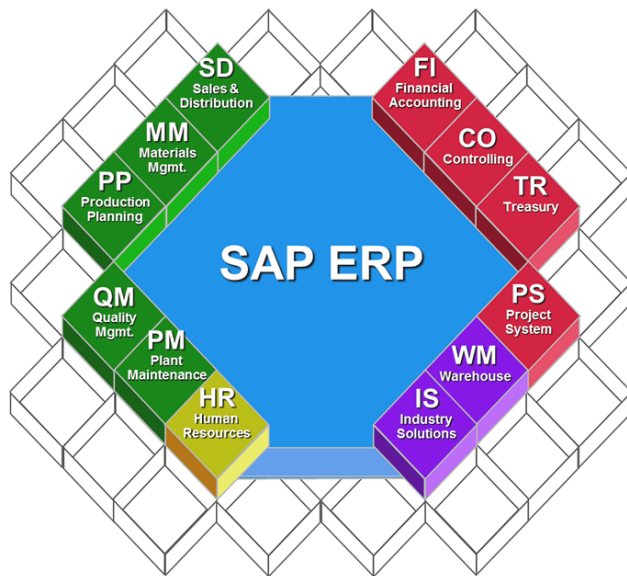
Análisis de Impacto.

### Comunicación

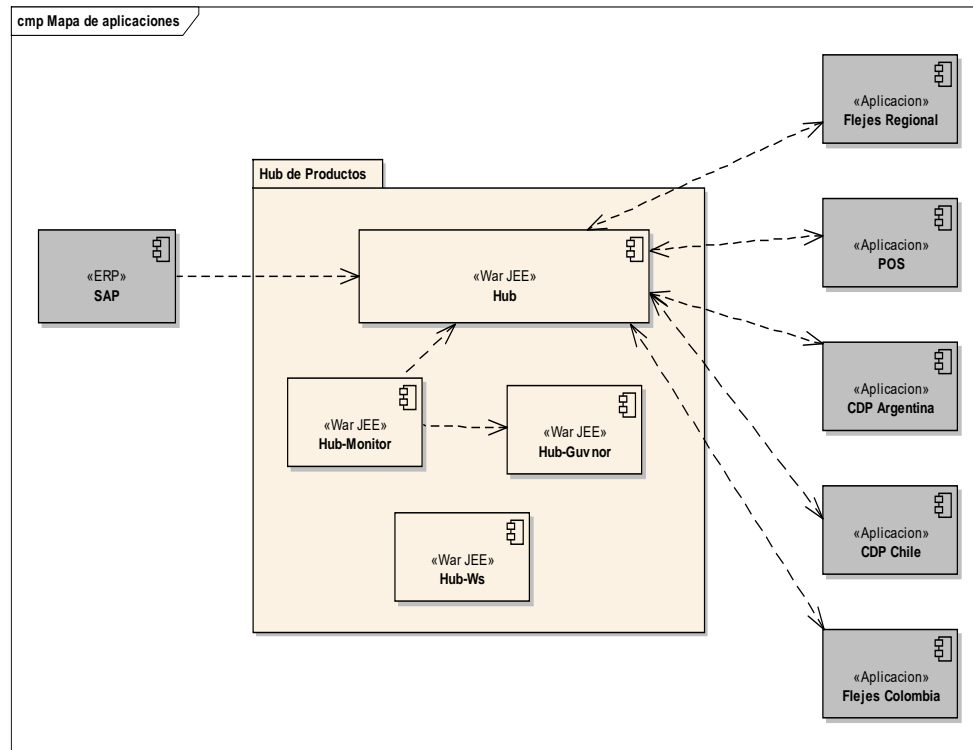
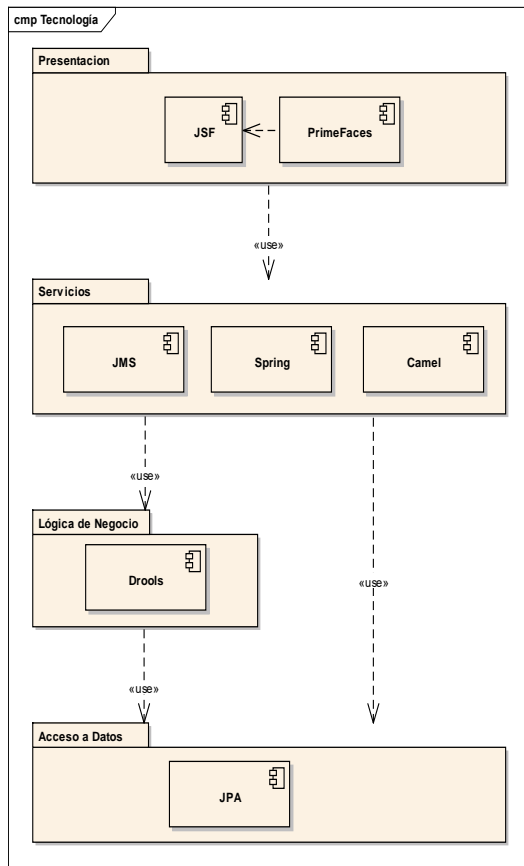
---

Pueden ser utilizadas para explicar las funcionalidades del sistema a alguien no familiarizado con el mismo.

# Module Viewtype



# Module Viewtype



# C&C Viewtype

- ❖ Estas estructuras están centradas en **procesos** que se comunican.
- ❖ Sus elementos son entidades con manifestación runtime que consumen recursos de ejecución y contribuyen al comportamiento en ejecución del sistema.
- ❖ La configuración del sistema es un grafo conformado por la asociación entre **componentes** y **conectores**.
- ❖ Las entidades runtime son **instancias** de tipos de conector o componente.
- ❖ Los componentes son entidades independientes y sólo se relacionan e interactúan a través de conectores.

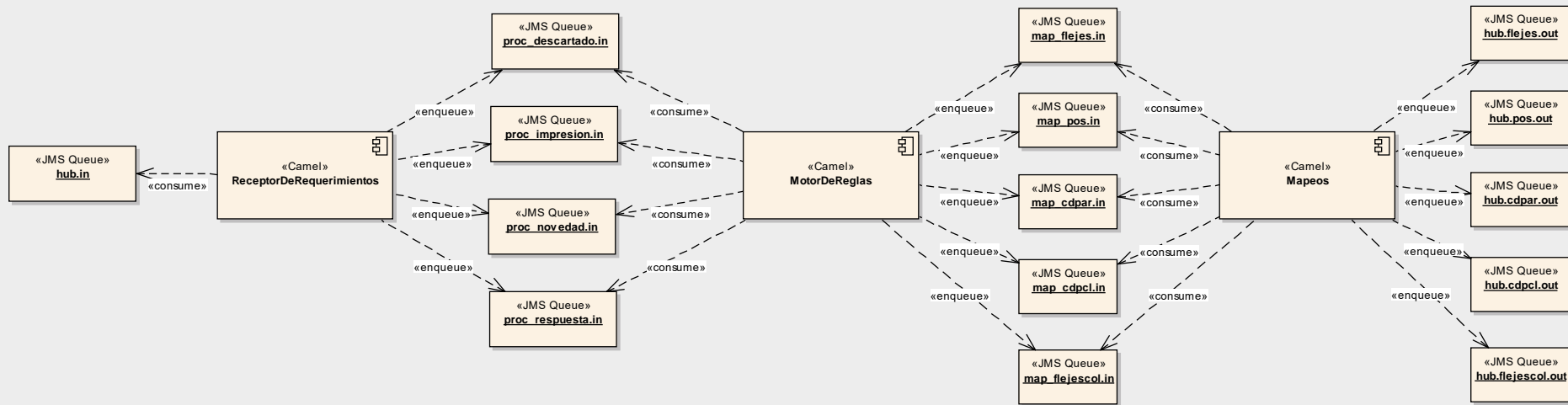
# C&C Viewtype

- ❖ La relación es **attachment**: Indica qué componentes están vinculados con qué conectores.
- ❖ Formalmente siempre se asocian puertos de componentes con puertos de conectores (llamados roles)



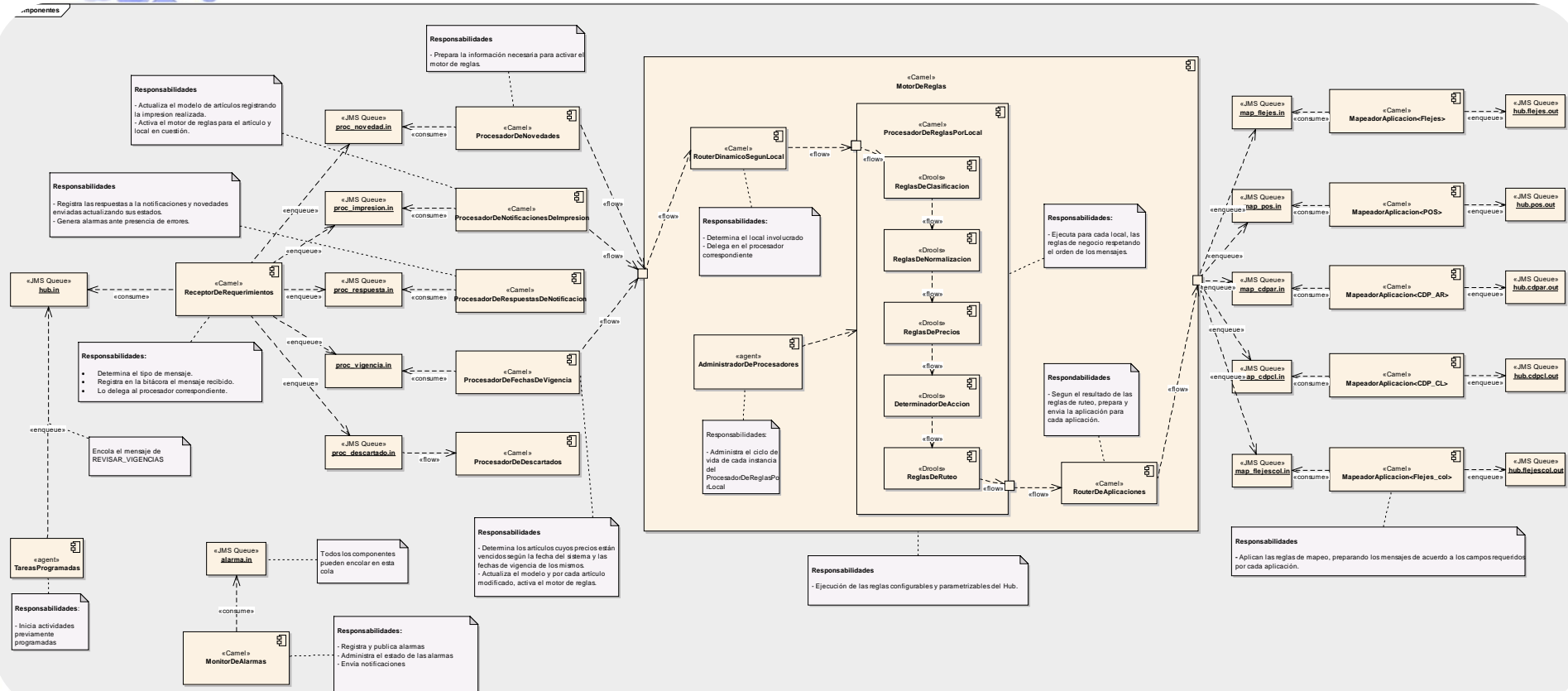
# C&C Viewtype

cmp Componentes Conceptual





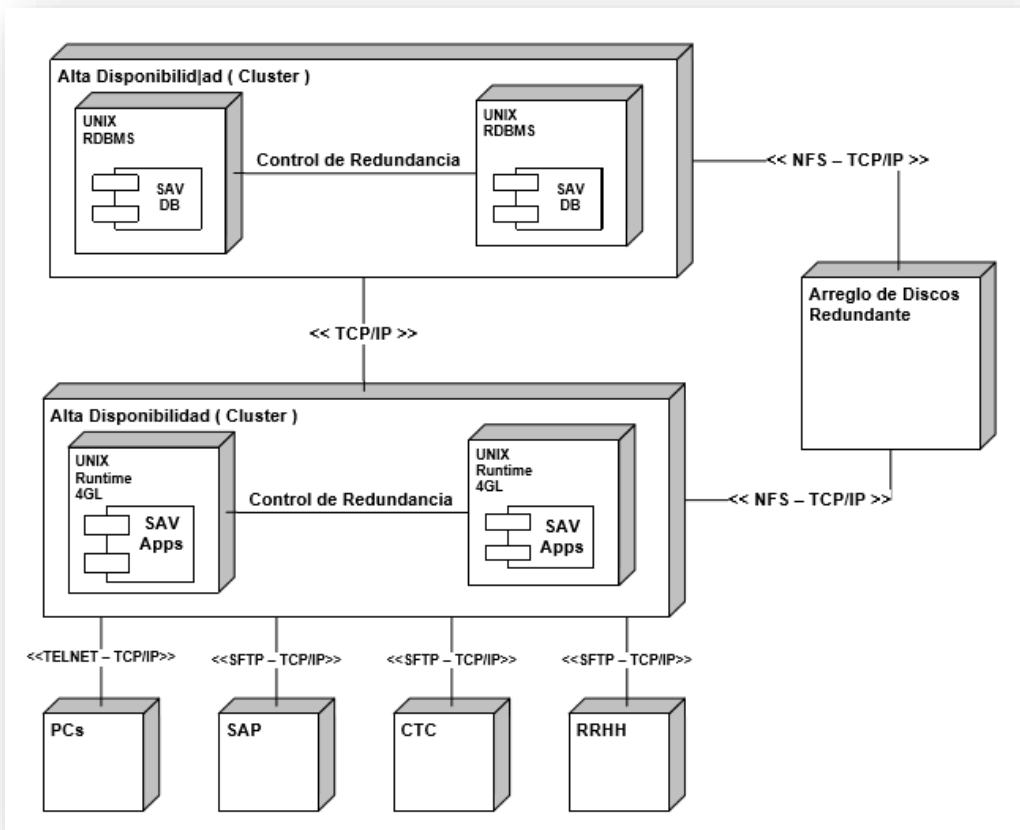
# C&C Viewtype

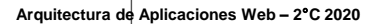


# Allocation **Viewtype**

- ❖ **Deployment:** muestra cómo el software se asigna a hardware y elementos de comunicación.
- ❖ **Implementación:** muestra cómo los elementos de software se mapean a estructuras de archivos en repositorios de control de la configuración o entornos de desarrollo.
- ❖ **Asignación de trabajo (“work assignment”):** asigna la responsabilidad del desarrollo y la implementación a equipos de programadores.

# Allocation Viewtype





# Estilos de Viewtypes

## Module

---

Descomposición

---

Usos

---

Generalización

---

Capas (Layers)

---

## C&C

---

Pipe & Filter

---

Publish & Suscribe

---

Shared Data

---

Client - Server

---

## Allocation

---

Deployment

---

Implementation

---

Work Assignment

---



# Estilos Arquitectónicos C&C

Un estilo arquitectónico determina el vocabulario de componentes y conectores que puede ser usado, así como un conjunto de restricciones de cómo pueden ser combinados.

Un estilo arquitectónico define una familia de sistemas en términos de patrón de organización estructural.

# Propiedades

- Un vocabulario para los elementos de diseño
  - Tipos de componentes y conectores
  - Por ejemplo: clases, invocaciones, “pipes”, clientes
- Reglas de composición
  - Un estilo tiene restricciones topológicas que determinan cómo se puede hacer la composición de los elementos.
  - Por ejemplo: los elementos de un “layer” se pueden comunicar sólo con los del “layer” inferior.
- Semántica para esos elementos
- Idealmente, criterios para la evaluación de una arquitectura o formas de analizarla; generación de código.
- Importante: un estilo arquitectónico no define la funcionalidad de un sistema. Desde ese punto de vista es algo “abstracto”.

# Arquitecturas Heterogéneas

- Resultan de la combinación de distintos estilos
- Por ejemplo:
  - Los componentes de un sistema “layered” pueden tener una estructura interna que use otro estilo.
  - Una arquitectura hecha con JEE probablemente resulte en una arquitectura heterogénea que incluya:
    - Layered
    - Repository
    - Independent components
    - Information hiding → Objects

En sistemas medianos / grandes, es más probable que un estilo arquitectónico describa una parte de un sistema que al sistema completo.



# Módulos vs Componentes

➤ **Módulos:** entidad en tiempo de diseño.

Enfatiza en encapsulamiento: “information hiding” e interfaces

➤ **Componentes:** tienen entidad en tiempo de ejecución y de despliegue.

# Componentes y Conectores

- Colección de módulos de software (**Componentes**) interactuando a través de un paradigma de comunicación bien definida (**conectores**)
- Los componentes son los bloques de construcción para describir una arquitectura.
- No existe aún una notación estándar.

# Componentes y Conectores

## Tipos de Componentes

---

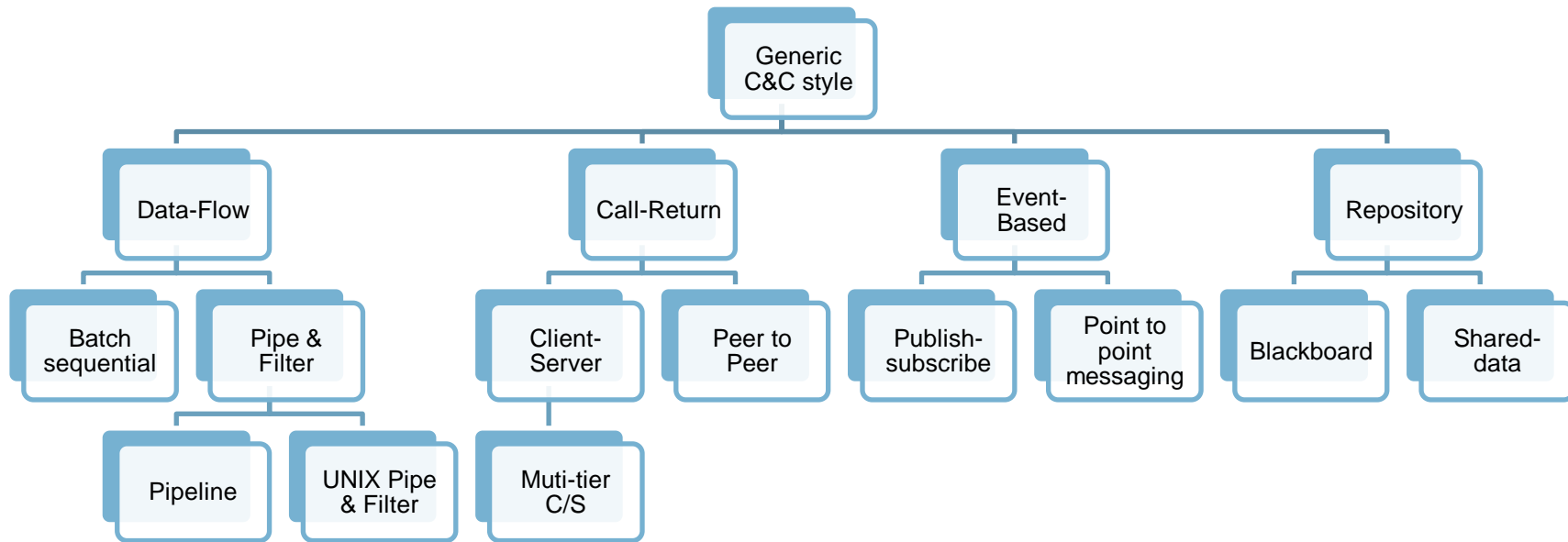
- **Computacional:** realiza el procesamiento en algún orden.  
Ej. función matemática, filtros.
- **Memoria:** mantiene una colección de datos persistentes.  
Ej. bases de datos, sistemas de archivos, tablas de símbolos.
- **Manejador:** contiene estado + operaciones asociadas. El estado es mantenido entre invocaciones de operaciones.  
Ej. TAD, Servidores.
- **Controlador:** gobierna la secuencia de tiempo de otros eventos.  
Ej. módulo de control de alto nivel, scheduler.

## Tipos de Conectores

---

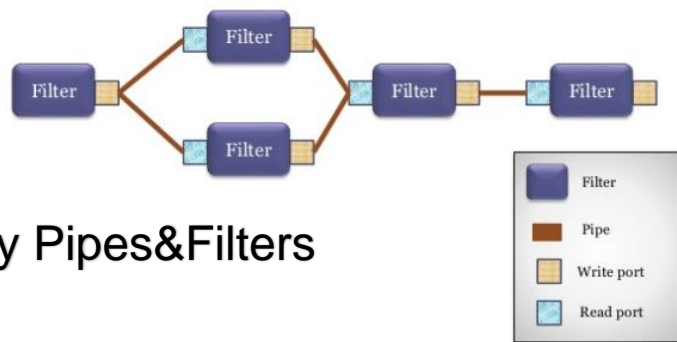
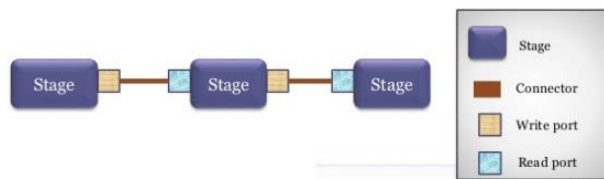
- **Procedure call**  
Simple thread de control entre el invocador (called) y el invocador (callee). Ej. RPC.
- **Data flow**  
Interacción de procesos a través de flujos de datos. Ej. pipes
- **Implicit invocation**  
El proceso se inicia hasta que un event o ocurra. Ej. listas de correo.
- **Message passing**  
La interacción se realiza a través de transferencia explícita o de datos discretos. Ej. TCP/IP.
- **Shared data**  
El acceso a datos es concurrente, con algún esquema de bloqueo para prevenir los conflictos. Ej. Pizarra, bases de datos compartidas.

# Taxonomía de Estilos Arquitectónicos



# Data Flow

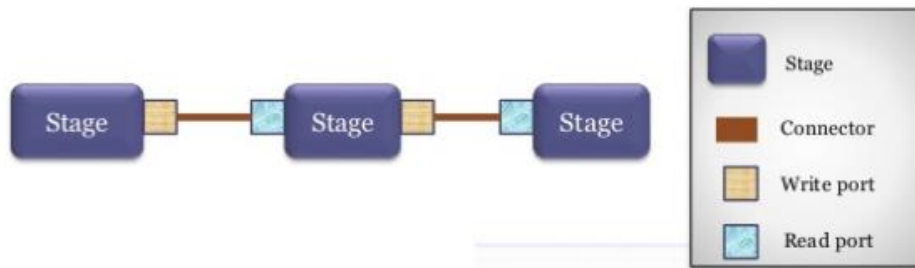
- ❑ La estructura del sistema está basada en transformaciones sucesivas a datos de input.
- ❑ Los datos entran al sistema y fluyen a través de los componentes hasta su destino final.
- ❑ Los componentes son de run-time.
- ❑ Normalmente un componente (de control) controla la ejecución del resto de los componentes



Dos estilos: Batch Sequential y Pipes&Filters

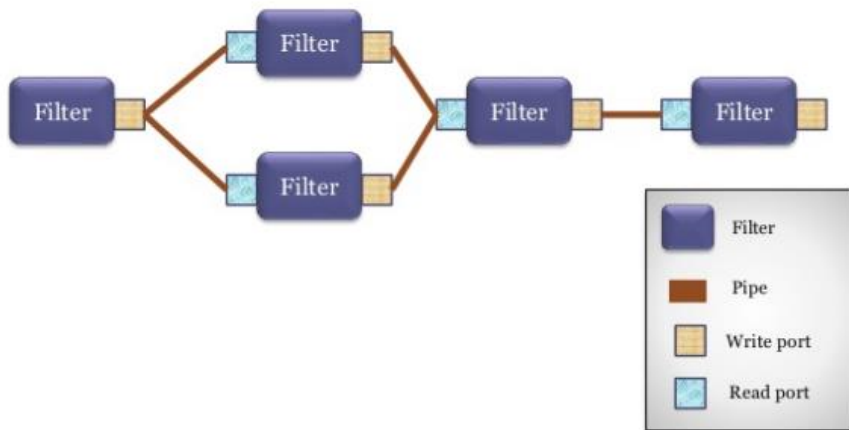
# Data Flow - Batch

- ❑ Cada paso se ejecuta hasta ser completado, y recién después puede comenzar el siguiente paso.
- ❑ Usado en aplicaciones clásicas de procesamiento de datos.
- ❑ Muchas veces usadas a partir de un “proceso off line”.



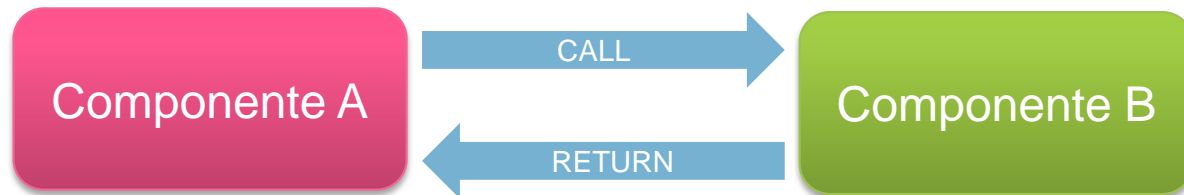
# Data Flow – Pipes&Filters

- ❑ Cada componente tiene inputs y outputs. Los componentes leen “streams” de datos de su input y producen streams de datos en sus outputs de forma continua.
- ❑ Filters: ejecutan las transformaciones.
- ❑ Pipes: conectores que pasan streams de un filtro a otro.



# Call Return

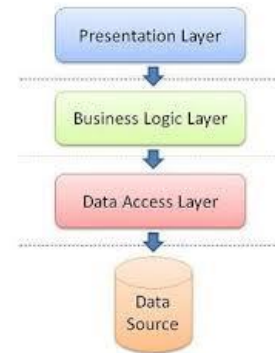
- ❑ Un componente llama o invoca a otro y se queda esperando la respuesta.





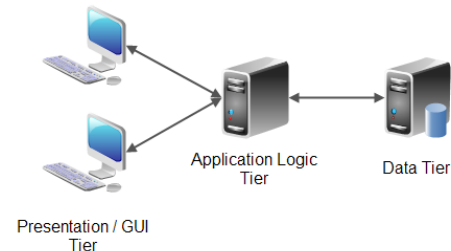
# Layered o Multi-tier

- Cada nivel provee servicios
  - Oculta en nivel siguiente
  - Provee servicios al nivel anterior
- En muchos casos el “bajar” de nivel implica acercarse al hardware o software de base
- Los niveles van formando “virtual machines”
  - Ventajas: portabilidad, facilidad de cambios, reuso.
  - Desventajas: performance, difícil de encontrar la abstracción correcta, puede implicar salteo de niveles.
- Ejemplos:
  - arquitectura de 3 capas (presentación, reglas de negocio, acceso a datos)



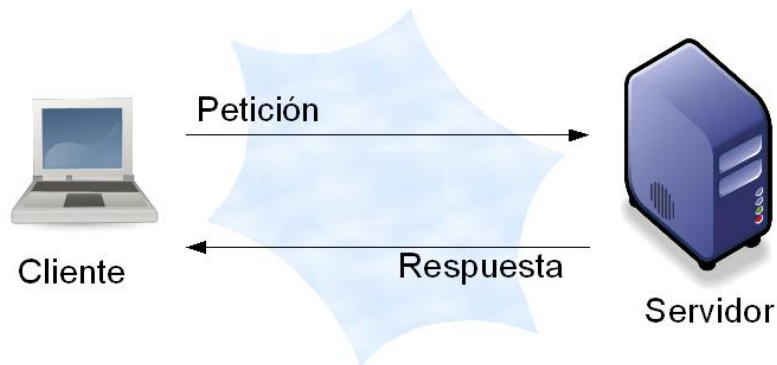
**Layer:** capa lógica.

**Tier:** capa física o entorno de ejecución.



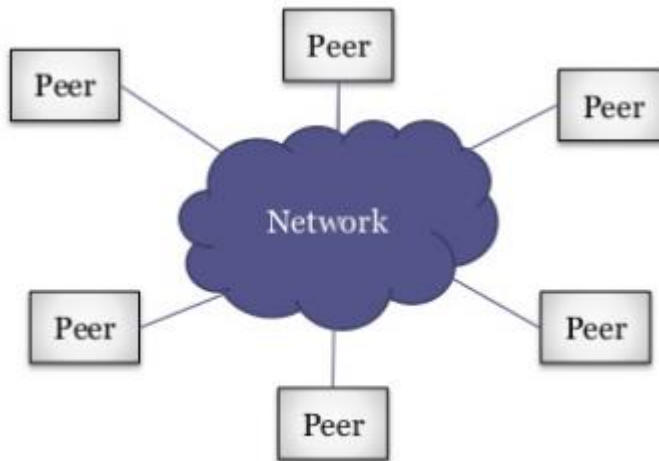
# Client / Server

- 2-Tier
- Los componentes son clientes (acceden a servicios) y servidores (proveen servicios)
- Los servidores no conocen la cantidad o identidad de los clientes.
- Los clientes saben la identidad de los servidores.
- Los conectores son protocolos basados en Call/Return



# Peer to Peer

- Nodos autónomos e iguales (peers) se comunican entre sí a través de la red.



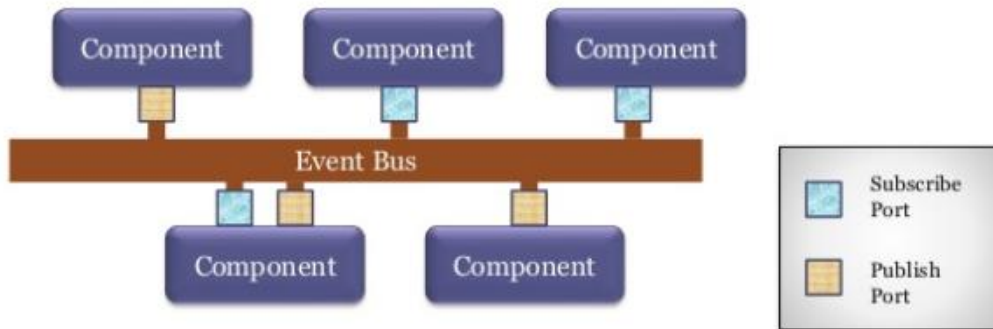
# Event Based

- EDA
  - Event Driven Architecture



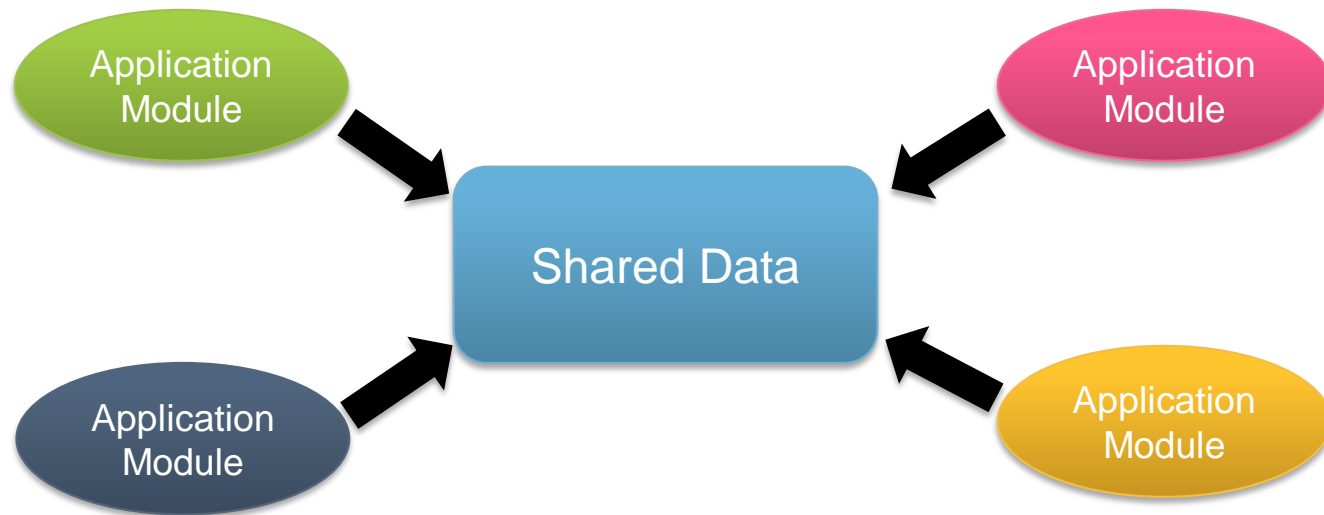
# Publish & Subscribe

- Componentes se suscriben a un canal para recibir mensajes de otros componentes.



# Centradas en datos

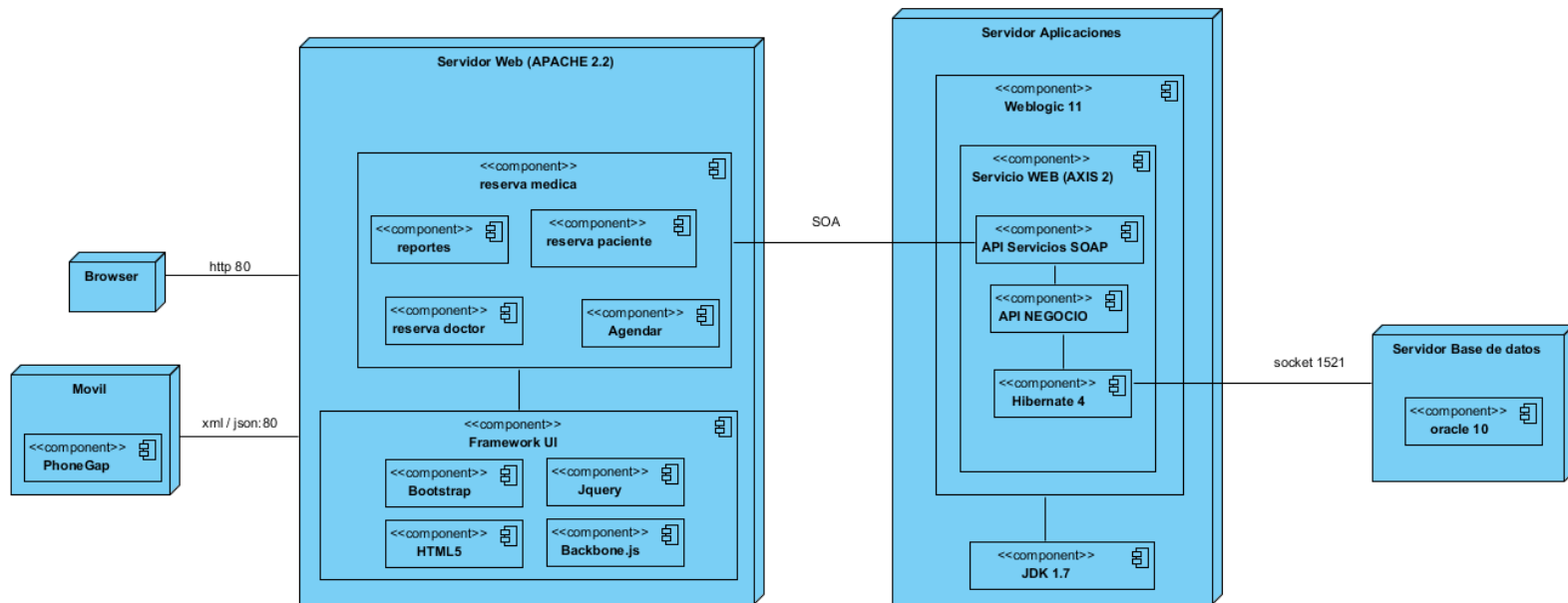
- Estructura de datos central (normalmente una base de datos) y componentes que acceden a ella.
- Gran parte de la comunicación está dada por esos datos compartidos.



# Documentación de Arquitecturas

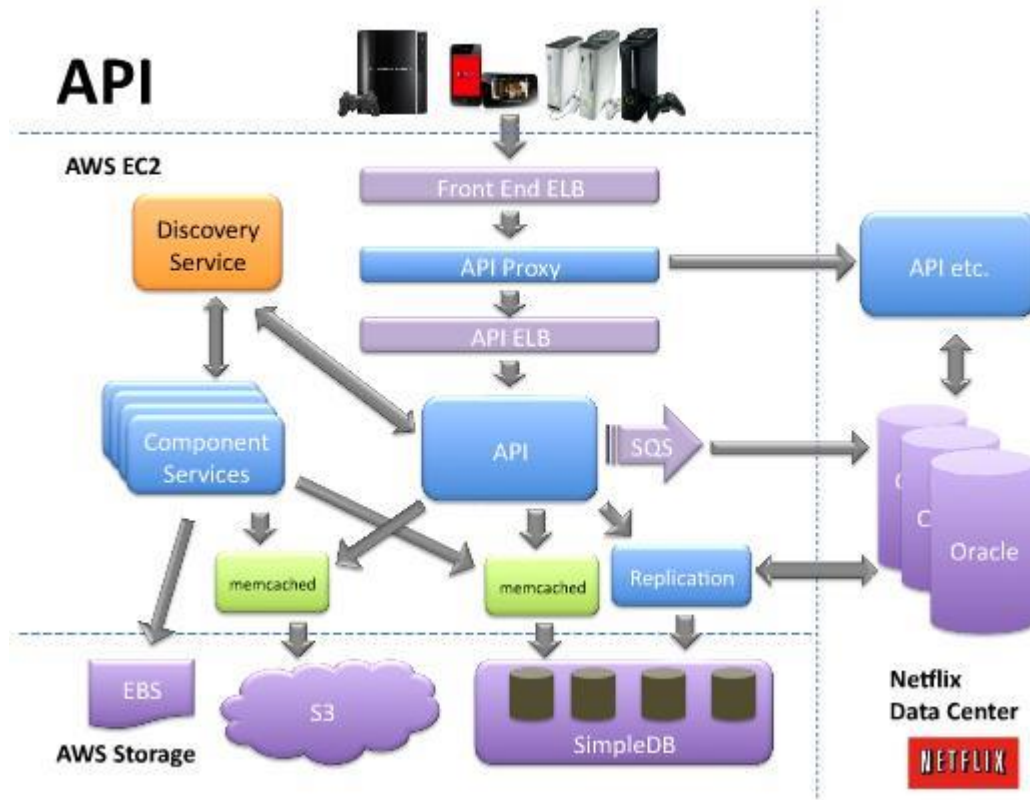
- ❖ Descripción de los requerimientos
  - Contexto del negocio, razón de ser para el producto, dominio
- ❖ Descripción del contexto
  - Sistemas con quienes interactúa, interfaces externas
- ❖ Uso de diagramas de arquitectura
  - Con prosa y descripción de cajas y líneas
- ❖ Consideración de restricciones de implementación
  - En la medida en que impactan la arquitectura
- ❖ Explicación del diseño arquitectónico
  - Como ataca los requerimientos y las restricciones de diseño
  - Alternativa

# Lo vi en Internet...

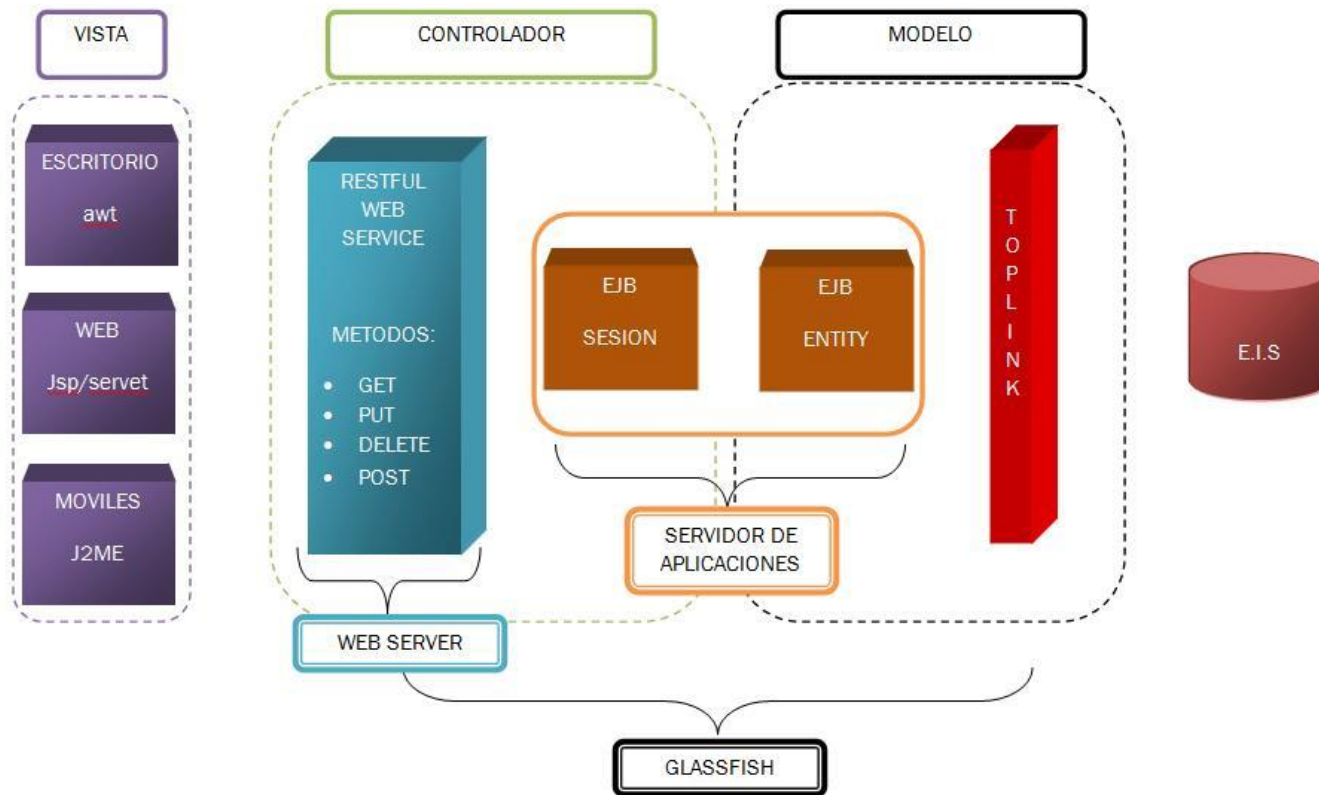




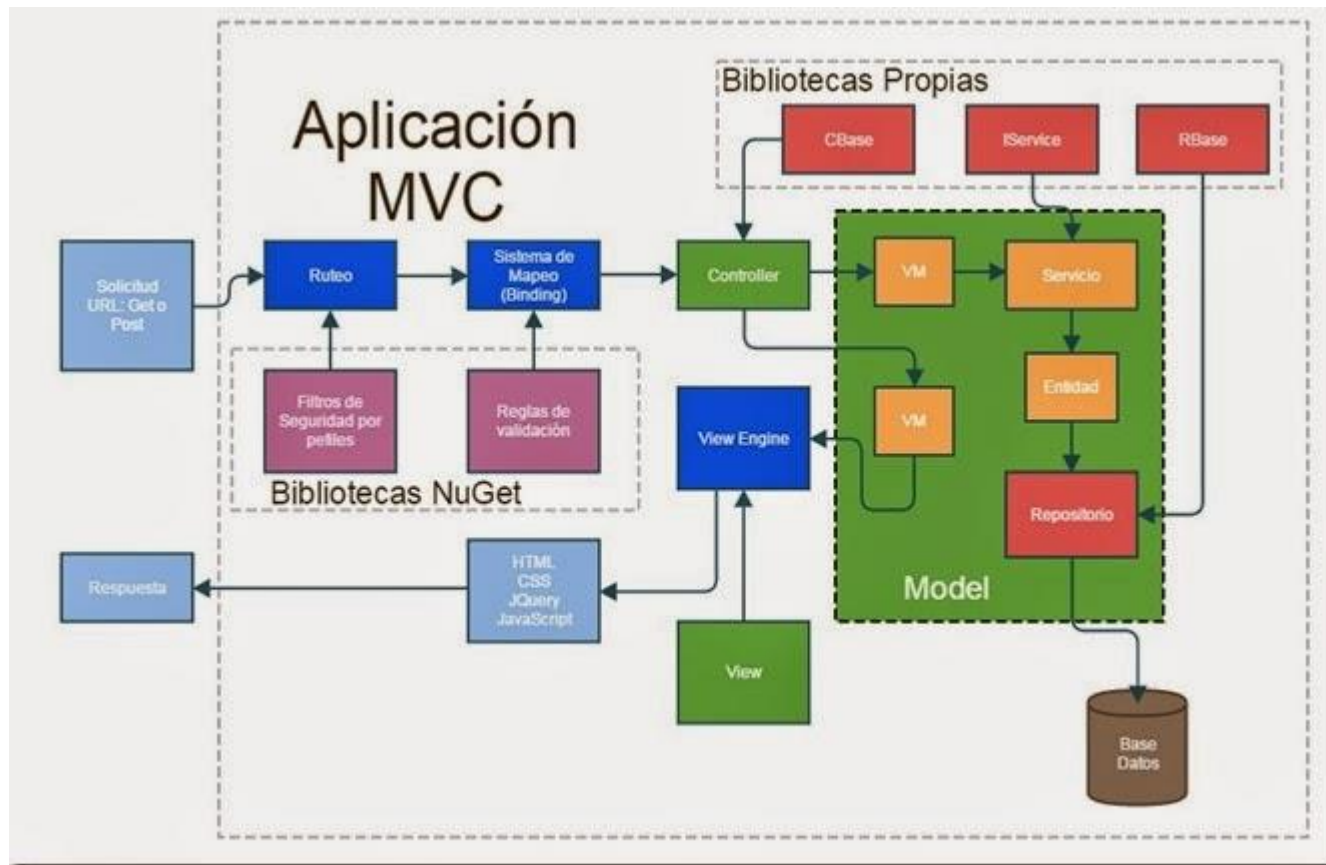
# Lo vi en Internet...



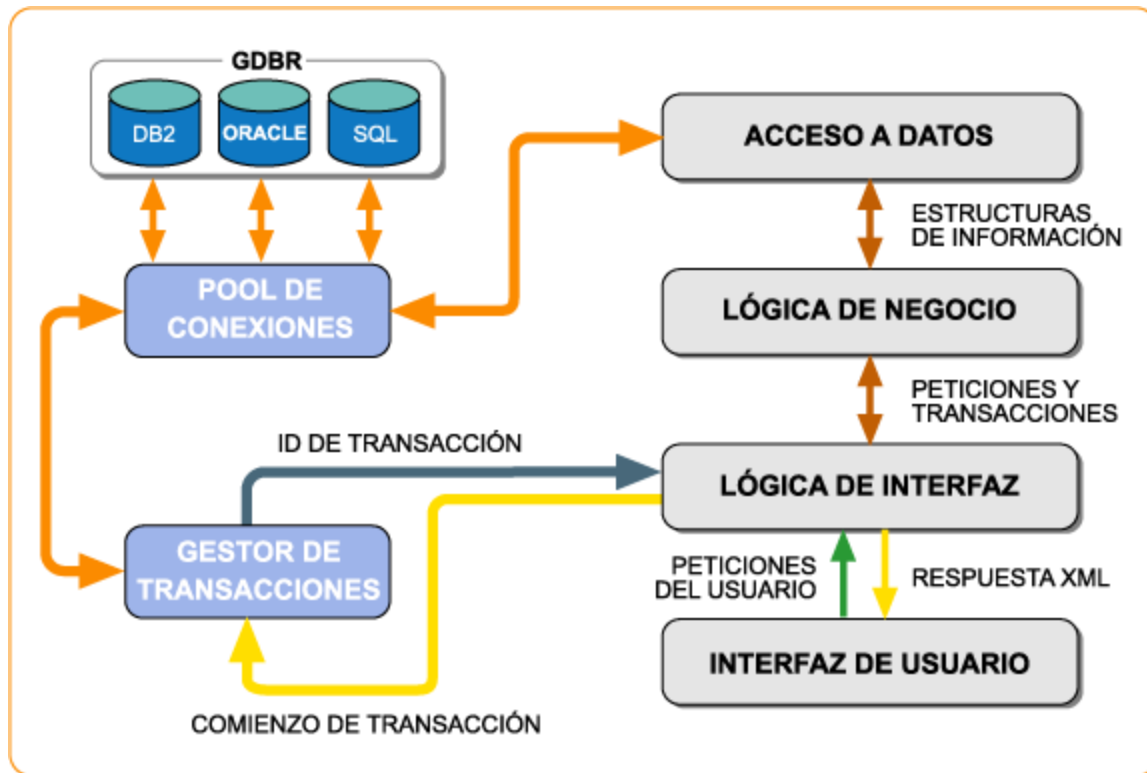
# Lo vi en Internet...



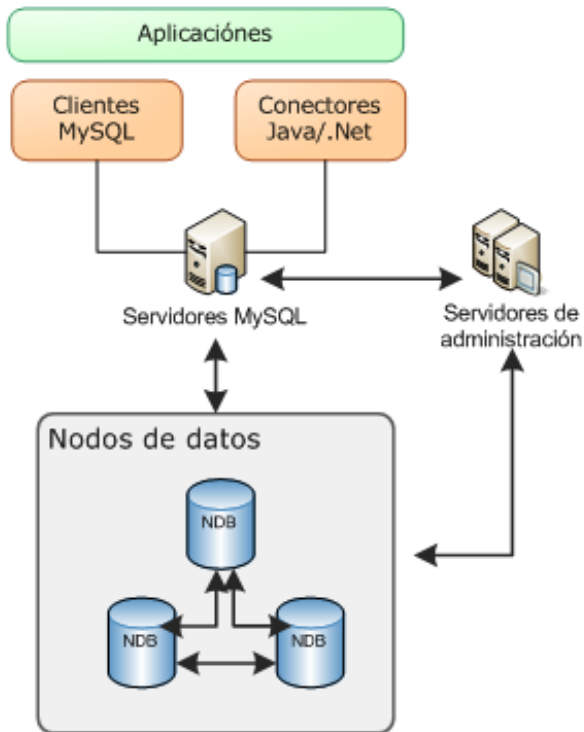
# Lo vi en Internet...



# Lo vi en Internet...



# Lo vi en Internet...





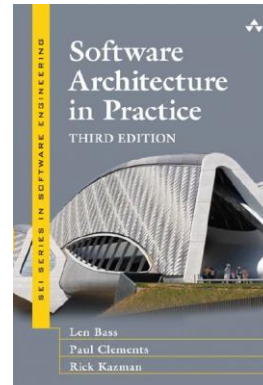
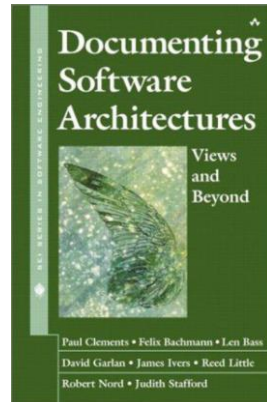
¿Preguntas?



# Bibliografía



- Documenting Software Architecture. Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford. 2002.
- Software Architecture in Practice. Len Bass, Rick Kazman, Paul Clements. 2012.
- Carlos Billy Reynoso, Introducción a la Arquitectura de Software, 2004.



# Papers Fundacionales



- Frederick Brooks Jr. *The mythical man-month*. Reading, Addison-Wesley, 1975.
- David Parnas. "On the Criteria for Decomposing Systems into Modules." *Communications of the ACM* 15(12), pp. 1053-1058, Diciembre de 1972.
- David Parnas. "On a 'Buzzword': Hierarchical Structure", *Programming Methodology*, pp. 335-342. Berlin, Springer-Verlag, 1978.
- David Parnas. "On the Design and Development of Program Families." *IEEE Transactions on Software Engineering* SE-2, 1, pp. 1-9, Marzo de 1976.
- Mary Shaw. "Abstraction Techniques in Modern Programming Languages". *IEEE Software*, Octubre, pp. 10-26, 1984.
- Mary Shaw. "Large Scale Systems Require Higher- Level Abstraction". *Proceedings of Fifth International Workshop on Software Specification and Design*, IEEE Computer Society, pp. 143-146, 1989.
- Dewayne E. Perry y Alexander L. Wolf. "Foundations for the study of software architecture". *ACM SIGSOFT Software Engineering Notes*, 17(4), pp.40-52, 1992.
- Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Reading, Addison-Wesley, 1995.
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal. *Pattern-oriented software architecture – A system of patterns*. John Wiley & Sons, 1996.