Grado en Ingeniería Software

Tema 1
Concepto de Arquitectura Software y Principios de Diseño

Índice de la Presentación

- Concepto de Arquitectura Software
- Principios de Diseño Software
- Patrones y Estilos
- Estándares
- Arquitecturas y Diseño UML
- Conclusiones

Concepto de Arquitectura Software

- Una **Arquitectura Software (AS)** representa la versión moderna del diseño de un sistema software.
- Las arquitecturas representan el diseño de un sistema software complejo desde distintos puntos de vista, según los intereses de diversos usuarios (“stakeholders”).

Concepto de Arquitectura Software

- Definición de Arquitectura Software (Perry & Wolf, 1991).
- **AS = {componentes, forma, lógica}**

Una arquitectura software es una tripleta de elementos arquitectónicos como clases y paquetes UML que se conectan con una determinada forma.
Diferentes arquitecturas pueden presentar formas diferentes (formas de organizar dichos componentes)
La lógica hace referencia a las decisiones que se toman para seleccionar un determinado componente o un patrón de diseño

Concepto de Arquitectura Software

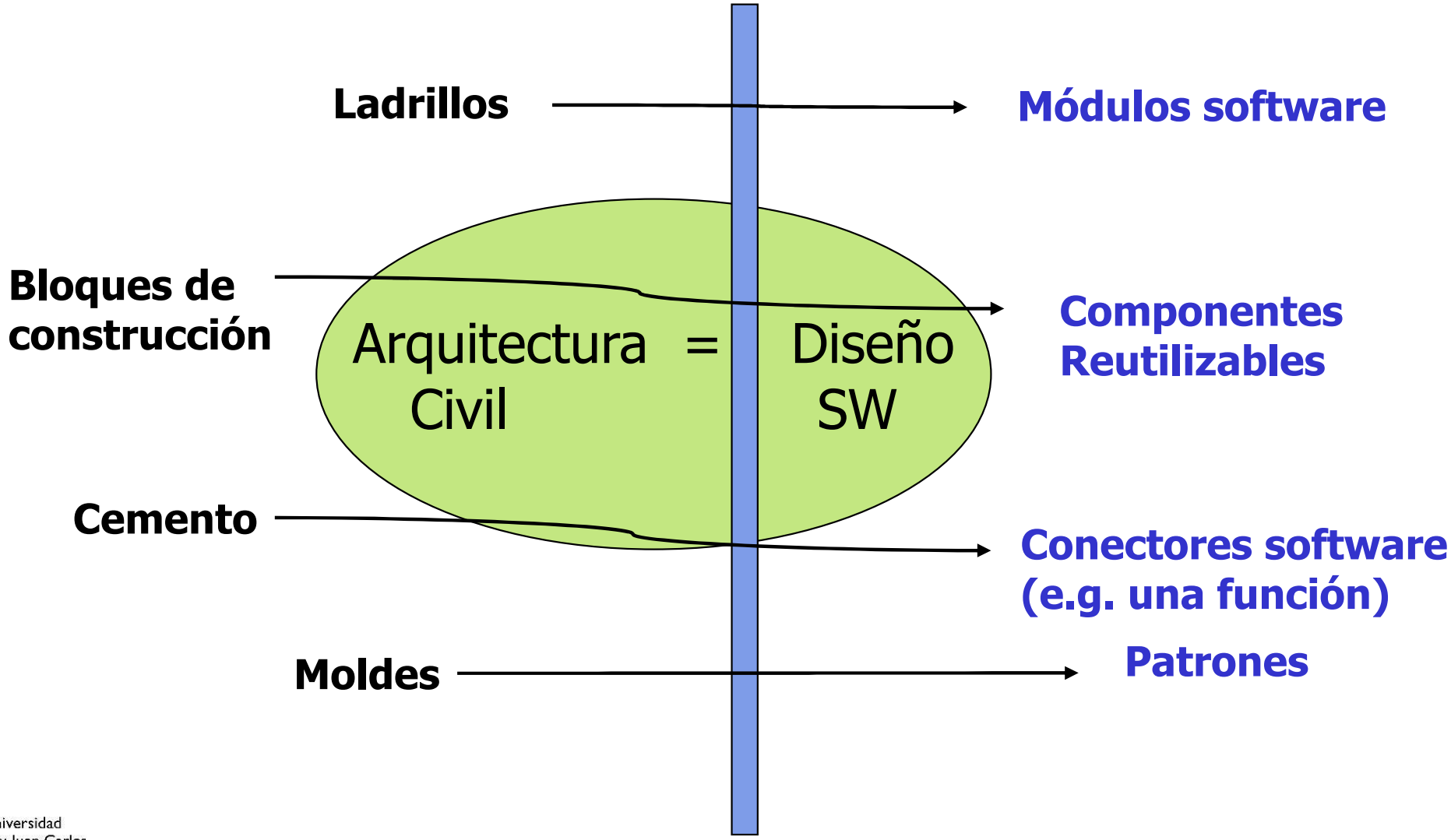
Distintos tipos de arquitecturas

Diferentes Estilos

Son arquitecturas distintas con diferentes estilos. Cada una sirve para una misión diferente. Los componentes son distintos en algunos casos y se unen de manera diferente.

Concepto de Arquitectura Software

Es una analogía comparando la arquitectura de casas y construcciones civiles con la arquitectura software

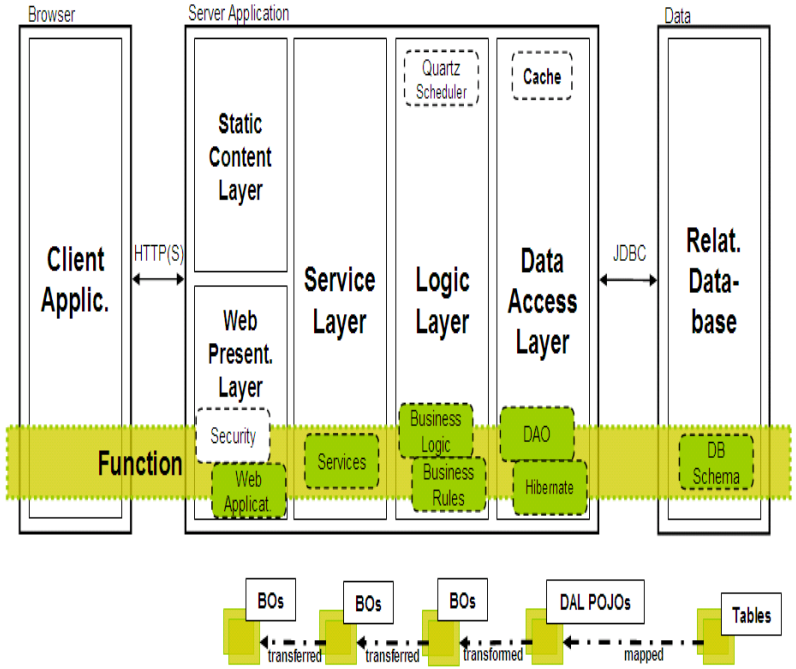


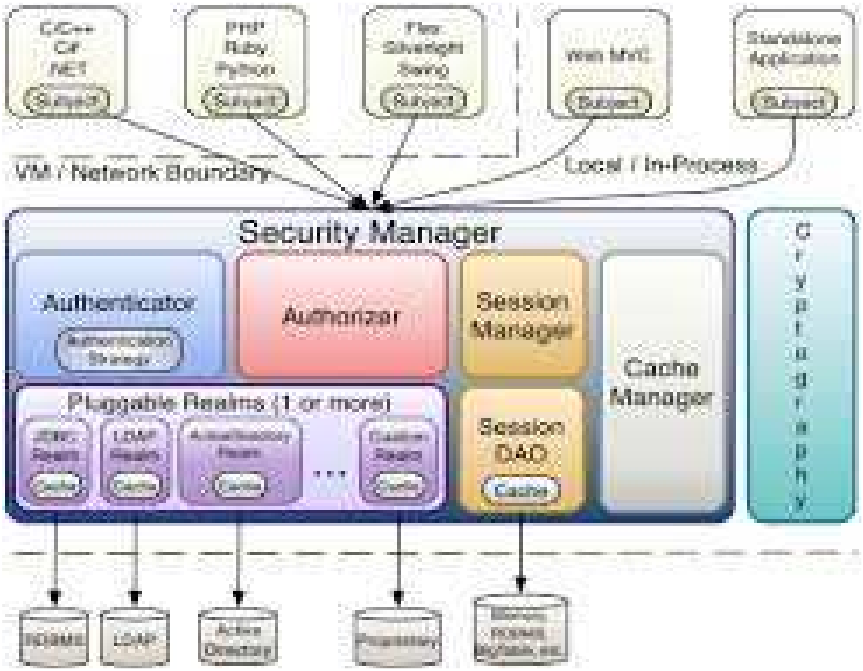
Concepto de Arquitectura Software

- Los diseñadores actuales se denominan Arquitectos Software
- La disciplina de las Arquitecturas Software tienen como misión identificar los requisitos funcionales y no funcionales que permitan construir la arquitectura de un sistema
- Las Arquitecturas Software son la piedra angular de toda fase de diseño software

Concepto de Arquitectura Software

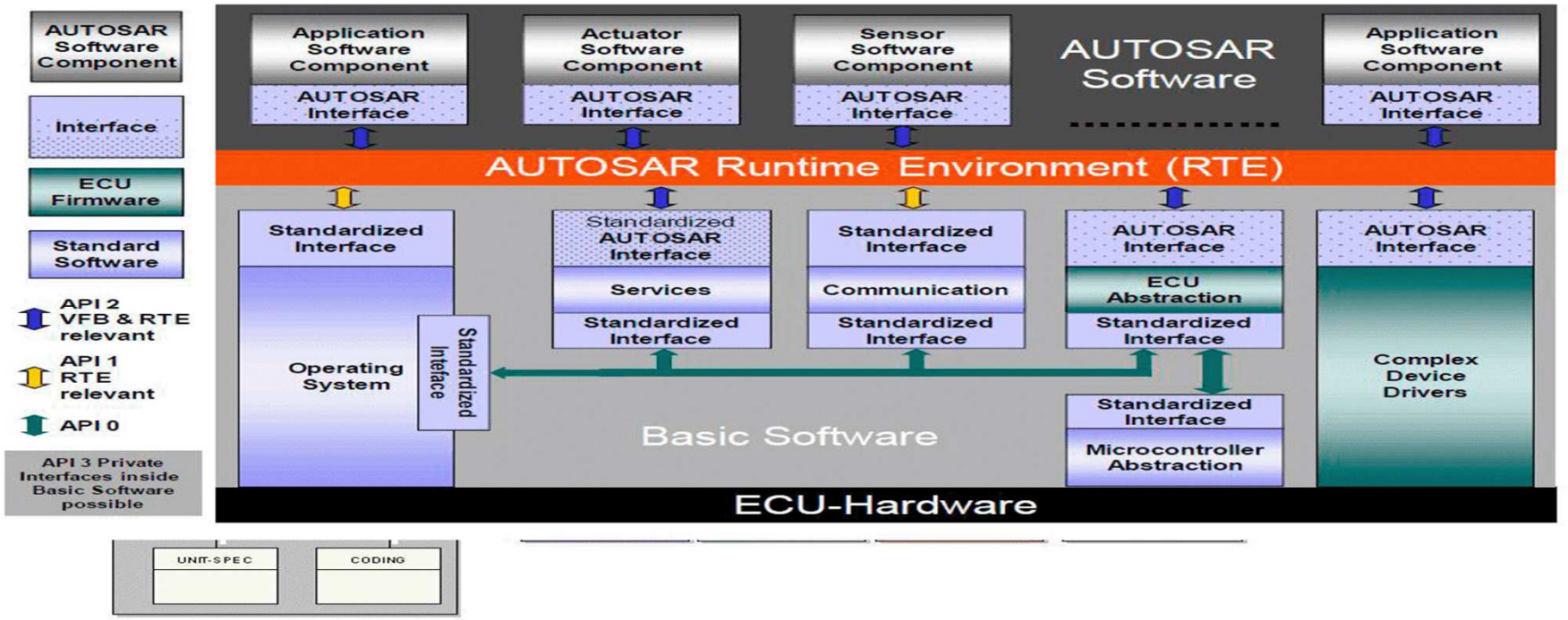
- Las arquitecturas software deben estar siempre alineadas con el código o implementación existente para evitar lo que se conoce como "*architectural mismatch*"
- Las arquitecturas software se "erosionan" o degradan (perdida de calidad) durante las fases de evolución del sistema
- Arquitecturas software con diferente grado de detalle





Concepto de Arquitectura Software

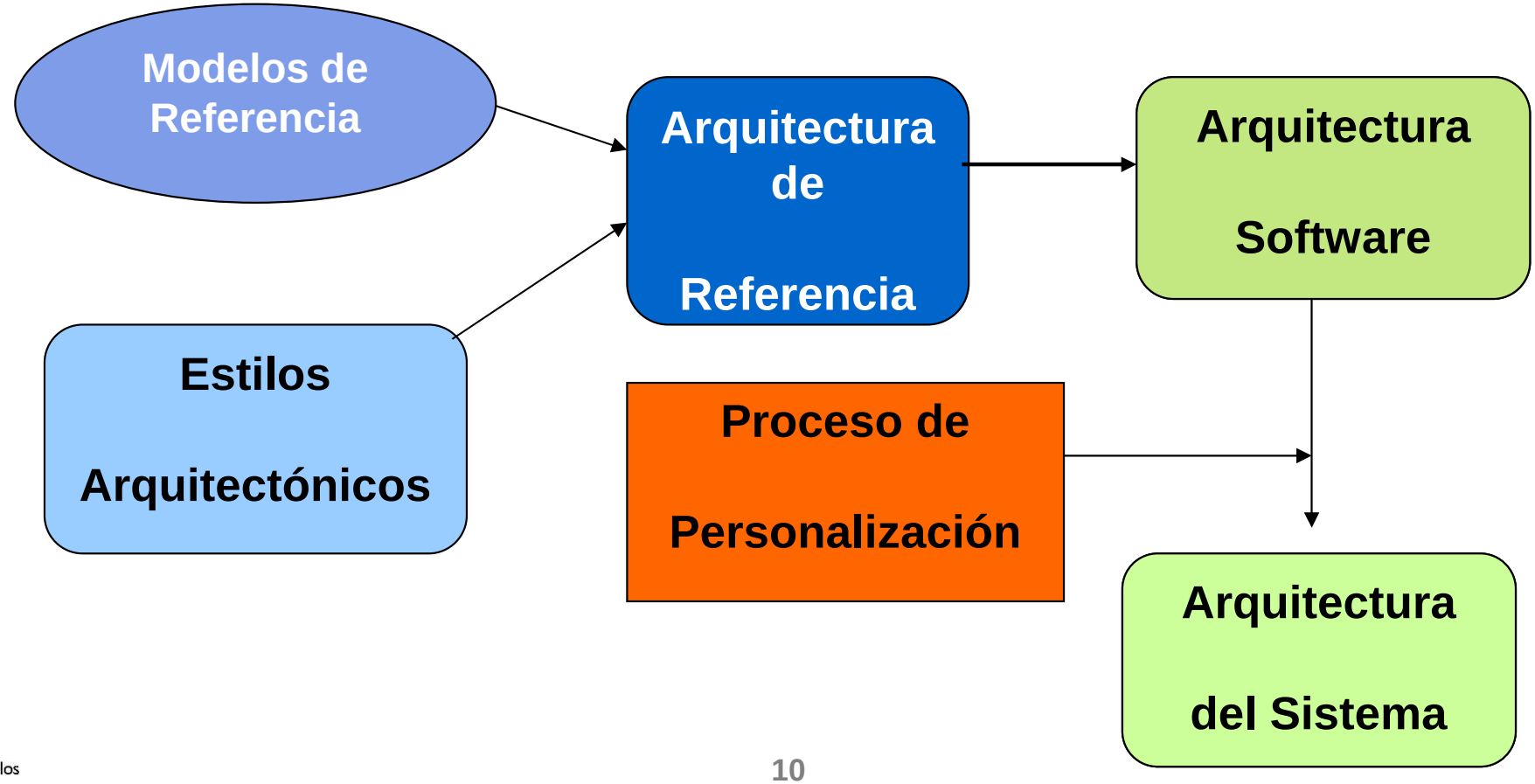
- Las Arquitecturas de Referencia son diseños software de alto nivel que describen las partes fundamentales de un conjunto de sistemas en un dominio particular
- E.g., Partes estándar de un compilador, Modelo OSI TCP/IP, arquitecturas en automoción (AUTOSAR), etc.



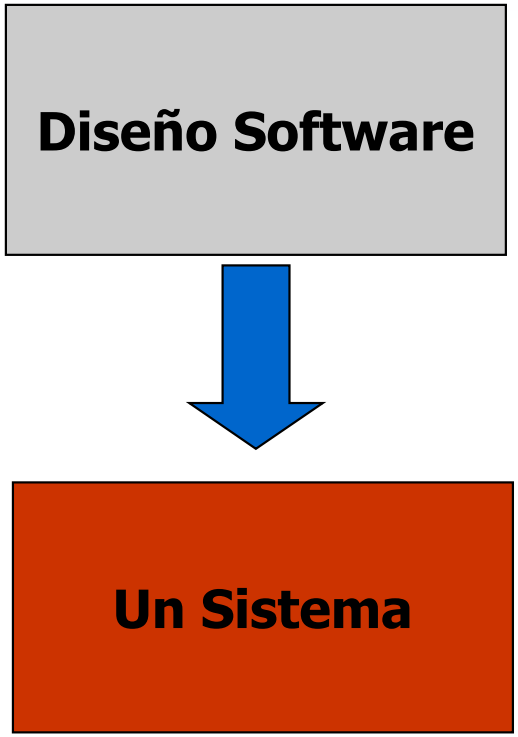
Concepto de Arquitectura Software

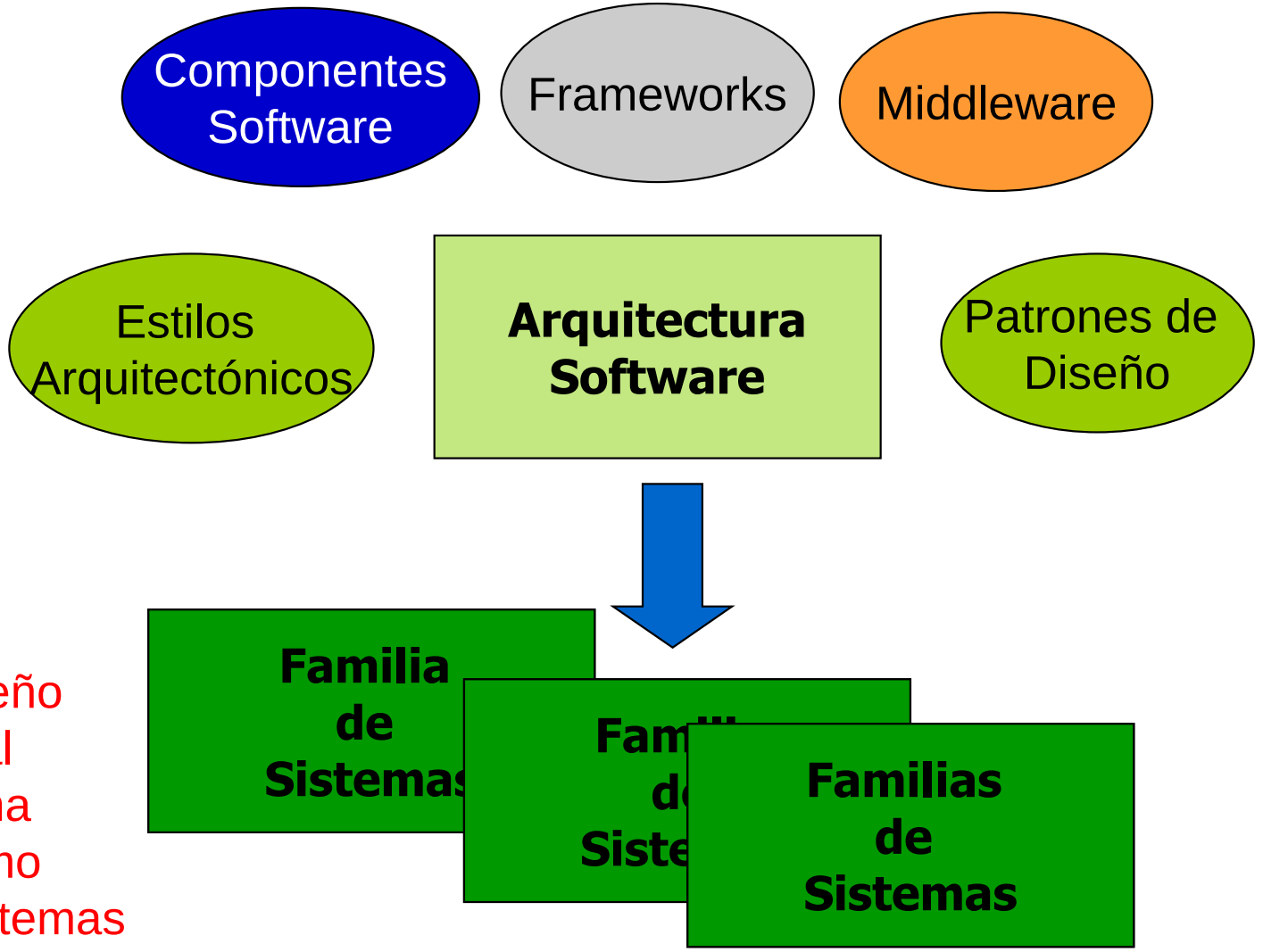
Modelos de Referencia: son una división de funcionalidad con un flujo de datos entre las partes que lo componen (ej: partes estándar de un compilador).

Arquitecturas de Referencia: son la asociación de un modelo de referencia sobre los componentes de software Y los flujos de datos entre dichos componentes.



Concepto de Arquitectura Software



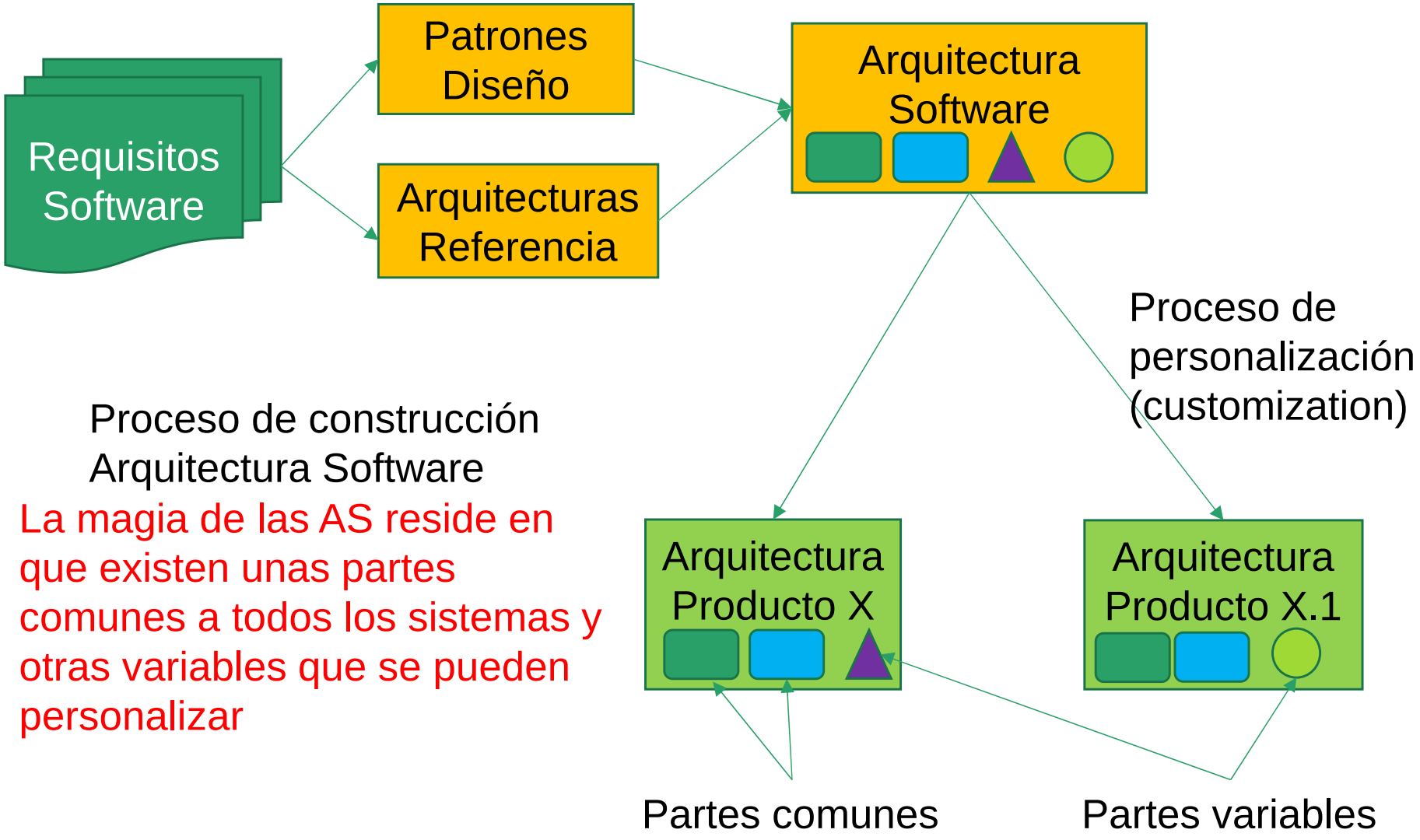


De la idea antigua de un diseño por cada sistema pasamos al nuevo concepto en el que una misma arquitectura sirve como diseño de un conjunto de sistemas similares

Concepto de Arquitectura Software

- La Arquitecturas del Producto Software es una personalización (derivación) de una arquitectura software para un producto determinado
- Se basa en la personalización de partes variables ("**software variability**") y el empleo de partes comunes reutilizables
- La arquitectura de un producto no se construye generalmente desde cero
- Refleja las características del producto concreto

Concepto de Arquitectura Software



La magia de las AS reside en que existen unas partes comunes a todos los sistemas y otras variables que se pueden personalizar

Principios de Diseño Software

- **Separación de responsabilidades** (“separation of concerns”): Evitar solapamientos de funcionalidad
- Hacer explícita la comunicación entre niveles
- Realizar abstracciones para diseñar niveles con **bajo grado de acoplamiento** (“loose coupling”)
- Modularizar el diseño
- Descomponer en subsistemas y módulos
- Las arquitecturas se rigen por un conjunto de principios de diseño software que no se deben violar

Principios de Diseño Software

Cohesion y Acoplamiento son principios de diseño básico en todo sistema software

- **Acoplamiento ("Coupling"):** Dependencias entre bloques o módulos de una arquitectura.
- **Cohesión ("Cohesion"):** Dependencias entre variables y métodos dentro de un componente o módulo.

Principios de Diseño Software

- No duplicar la funcionalidad de componentes
- No sobrecargar (“overload”) la funcionalidad de componentes (anti-patrón God Class)
- Definir interfaces y contratos entre componentes de manera clara
- Un componente no tiene porque conocer los detalles internos de otro componente (“information hiding”)
- Alta cohesión→Bajo acoplamiento
- El bajo acoplamiento de consigue mediante la abstracción, separación de responsabilides y ocultamiento de la información
- Evitar dependencias circulares

Principios de Diseño Software

Como conseguir "loose coupling"

- **Principio de Hollywood**: "Don't call us, we call you"
- **Dependency Inversion of Control**: Un módulo define una interfaz que otros módulos la realizan. Los módulos en los superiores no dependen de los módulos en niveles inferiores.
- **Dependency Injection**: Transfiere la responsabilidad de creación y enlace de módulos de software a un framework externo configurable

Un ejemplo de IoC sucede cuando ponemos el ratón en una ventana de Windows, nos proporciona información sobre el programa que se ejecuta sin pedírselo al programa principal

Patrones y Estilos

- La construcción de arquitecturas software se basan en los patrones de diseño y estilos arquitectónicos ya definidos.
- Un **Patrón de Diseño Software** (“design pattern”) es una solución de diseño para un problema recurrente en un contexto determinado.
- Son soluciones de diseño reutilizables ya probadas.

Patrones y Estilos

- Cronología e historia de los patrones software
  - The Timeless Way of Building (C. Alexander, 1979).
  - Lenguajes de patrones para programas OO (Cunningham y Beck, 1987).
  - Patrones de diseño (Gamma et al., 1995, GoF book).
  - Estilos arquitectónicos (Buschmann et al., 1996-2000).
  - Patrones de arquitecturas de aplicaciones empresariales (M. Fowler, 2002).
  - Patrones actuales centrados en tecnologías y plataformas (e.g.: Java, CORBA, Web).

Patrones y Estilos

Existen diversas categorías (Librería Básica de Patrones)

- **Creacionales**
  - Abstract Factory, Builder, Singleton…
- **Estructurales**
  - Adapter, Facade, Proxy
- **Comportamiento**
  - Mediator, Observer….
- **Concurrencia**
  - Scheduler, Lock, Thread Pool
- **Interacción** (Cunningham & Beck)
- **Workflow**

Patrones y Estilos

Estilos Arquitectónicos (Garlan & Shaw, 1995; Clements et al., 1998-2003)

- Modelo-Vista-Controlador (MVC)
- Capas (Layers, Tiers)
- Filtros y Tuberías (Pipe & Filter)
- Cliente/Servidor (Client/Server)
- Pizarra (Blackboard)
- Maquina virtual
- Basados en componentes
- Message bus
- Peer-to-Peer
- SOA y Micro-servicios

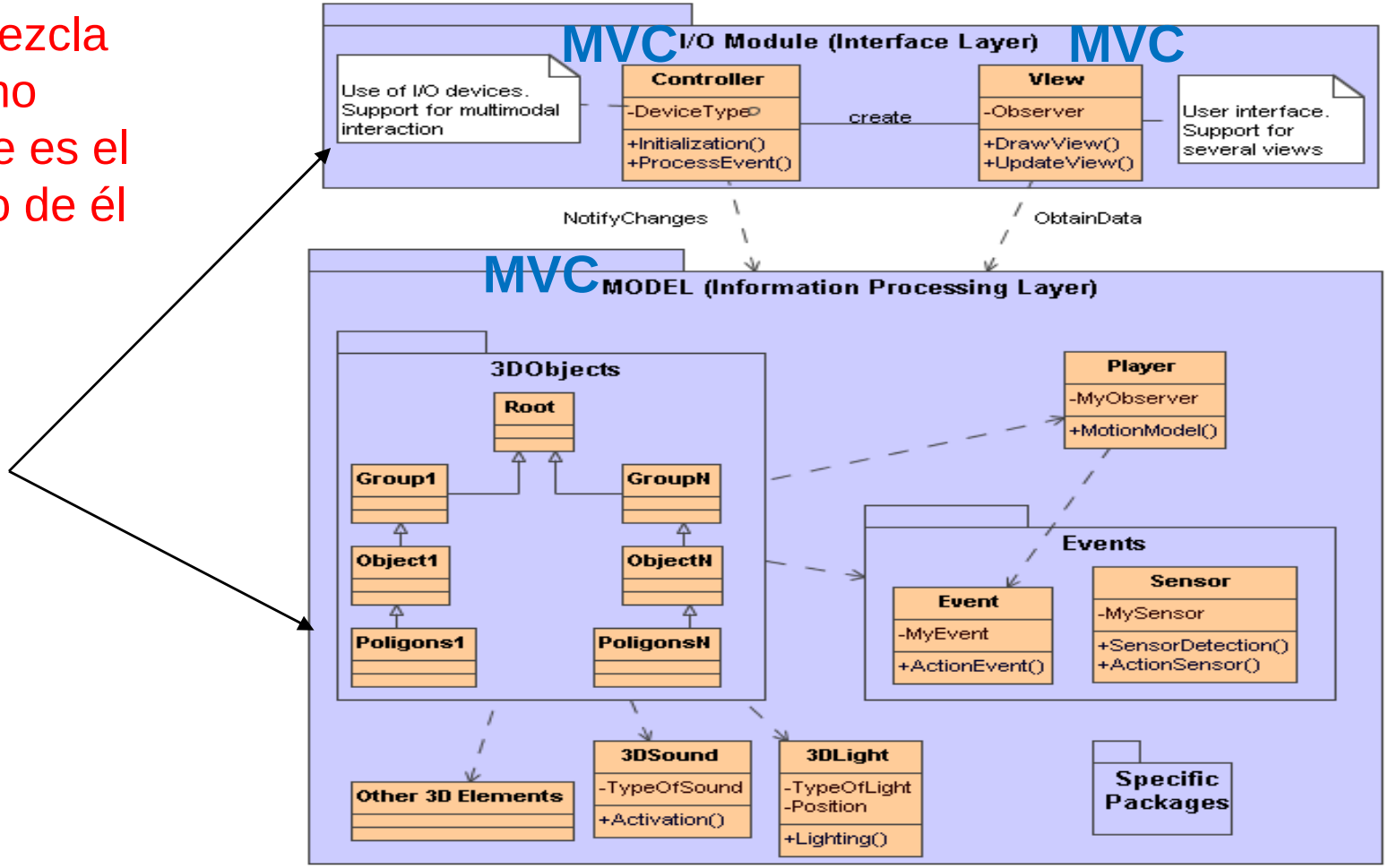
Un estilo arquitectónico de diferencia de un patrón de diseño en el tamaño. Los estilos involucran subsistemas mientras que los patrones son un conjunto más reducido de clases. Ambos sirven para el mismo propósito que es organizar los elementos de una AS. Evitan reinventar soluciones de diseño ya probadas.

Patrones y Estilos

Ejemplo de arquitectura heterogénea (mezcla de estilos)

Ejemplo de AS mezcla de dos estilos. Uno predominante que es el de capas y dentro de él un estilo MVC.

Capas (layered)



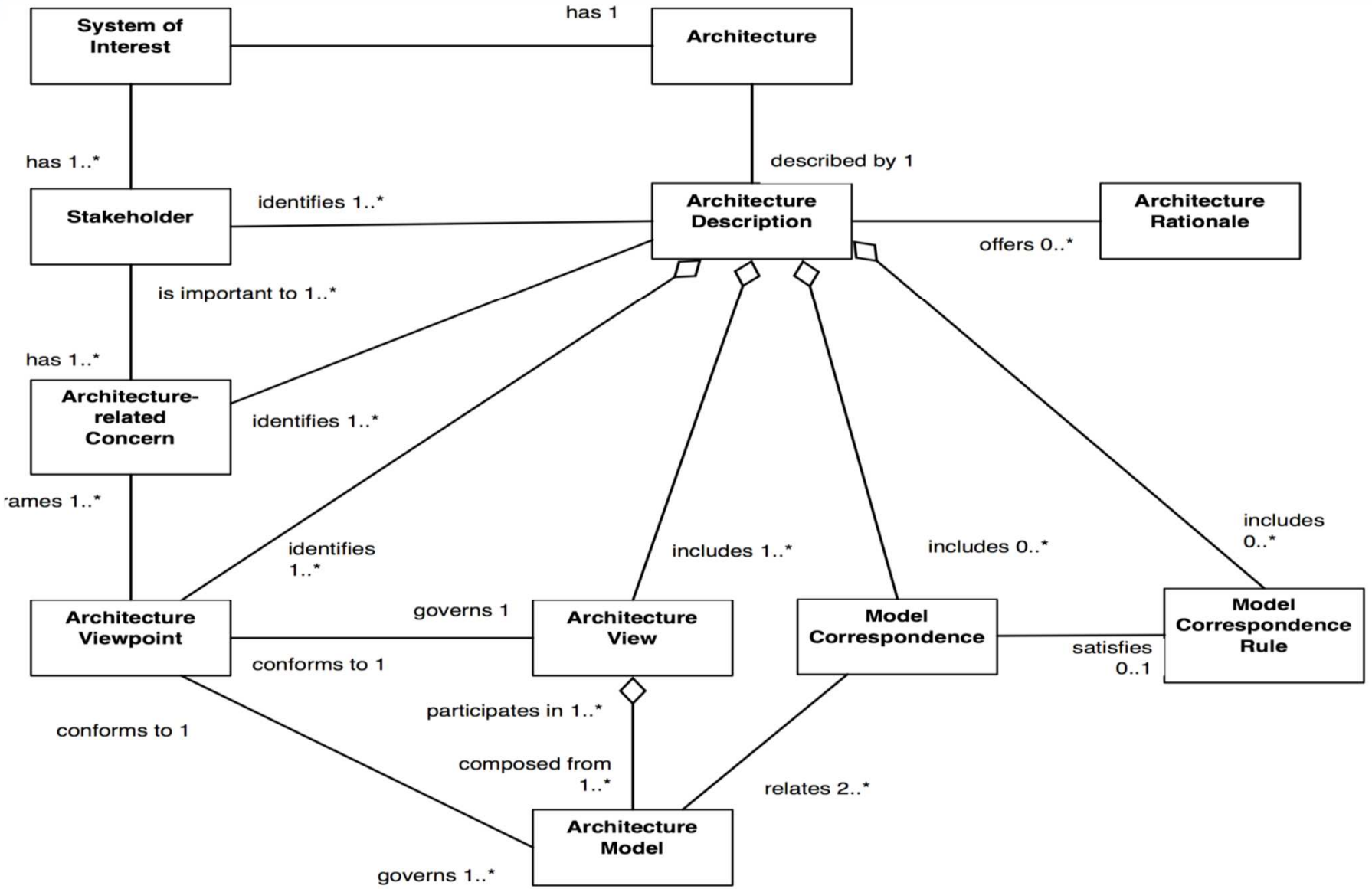
Estándar ISO 42010

- Nuevo estándar ISO/IEC 42010 (2009) en sustitución del IEEE 1471-2000, como práctica recomendada para la descripción de sistemas intensivos software.
- Mantiene el modelo de vistas arquitectónicas.
- Incluye como novedad soporte para lógica y decisiones de diseño.
- Alineación con otros estándares como RM-ODP y Enterprise Architecture (EA).

Estándar ISO 42010

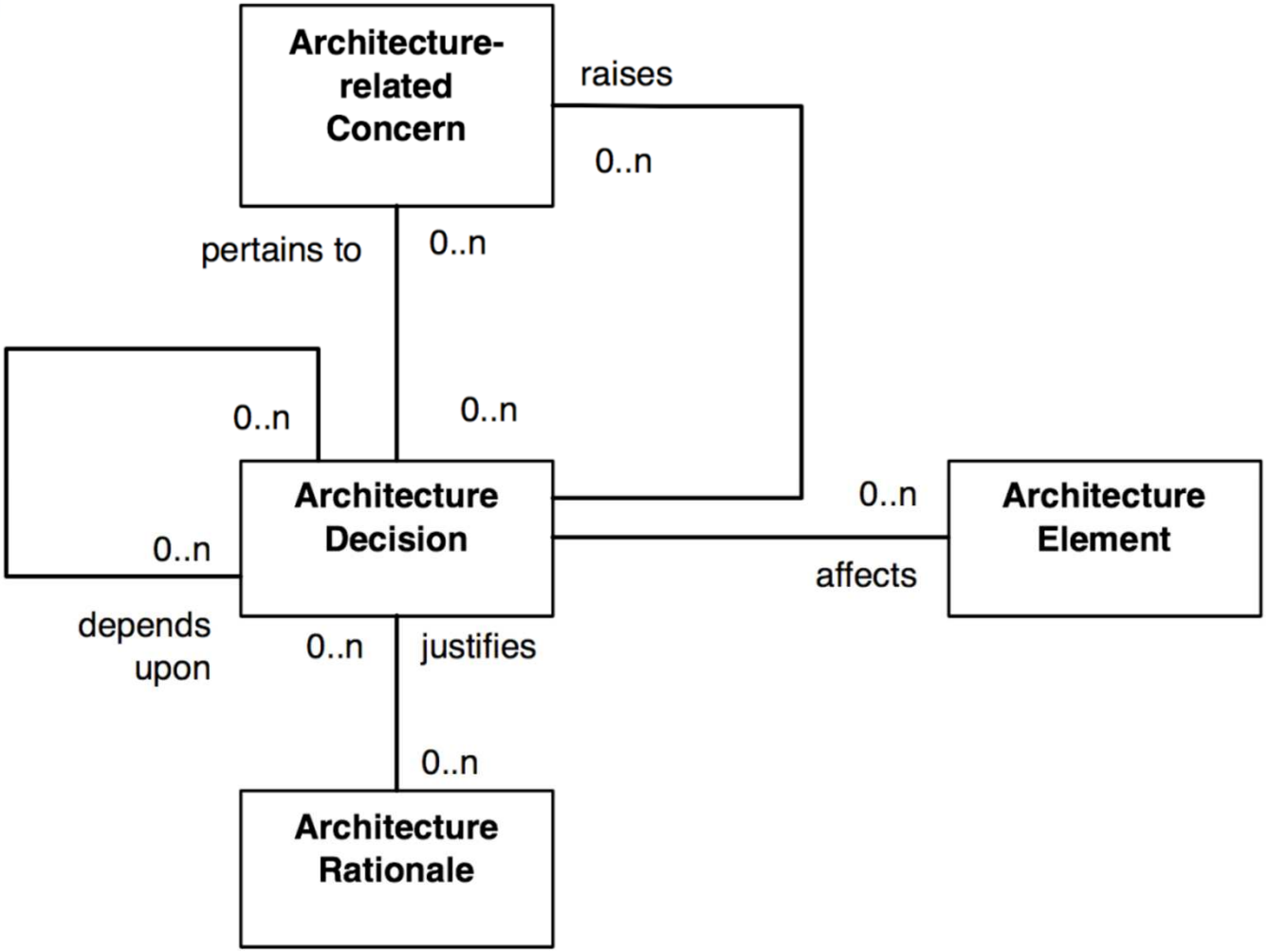
- Conceptos y Terminología:
  - **Architecture**: fundamental conception of a system in its environment embodied in elements, their relationships to each other and to the environment, and principles guiding system design and evolution.
  - **Architecting**: activities of conceiving, defining, describing, documenting, certifying proper implementation of, maintaining and improving an architecture throughout a system's life cycle.
  - **Architecture decision**: choice made from among possible options that addresses one or more architecture-related concerns.
  - **Architecture rationale**: explanation or justification for an architecture decision.
  - **Architecture view**: work product representing a system from the perspective of architecture-related concerns.
  - **Architecture viewpoint**: work product establishing the conventions for the construction, interpretation and use of architecture views and associated architecture models.

Estándar ISO 42010



Estándar ISO 42010

Esta es la parte novedosa del nuevo estándar que incluye elementos específicos en el modelo para capturar y representar las decisiones de diseño. De esta manera se equipara al mismo nivel una clase UML que la captura de una decisión de diseño en el formato que sea.



Conclusiones

- El proceso de construcción de arquitecturas software es *iterativo*, basado en la experiencia del arquitecto software, y utiliza patrones de diseño y estilos arquitectónicos.
- Las AS son una *combinación heterogénea de patrones y estilos* en los que suele predominar uno de ellos.
- Las *vistas se utilizan para documentar una arquitectura* que interesa a *diferentes tipos de usuarios*.
- Existen *Arquitecturas Específicas de Dominio* (automoción – AUTOSAR-, aviónica)
- Actualmente, *las AS son consideradas como el resultado de un conjunto de decisiones de diseño*.
- El nuevo estándar ISO 42010 refleja por primera vez la importancia de las decisiones de diseño en AS.

Lecturas adicionales

- Definiciones de AS (SEI): **http://www.sei.cmu.edu/architecture/definitions.html**
- Arquitectura de Referencia AUTOSAR https://www.autosar.org/
- Inversion of control / dependency injection http://stackoverflow.com/questions/6550700/inversion-of-control-vs-dependency-injection
- Bredmeyer (Sotware Architecture) https://www.bredemeyer.com/
- Microservices (Style and Patterns) https://microservices.io/

Grado en Ingeniería Software

Tema 2
Arquitecturas de Sistemas de Complejos en la Industria 4.0

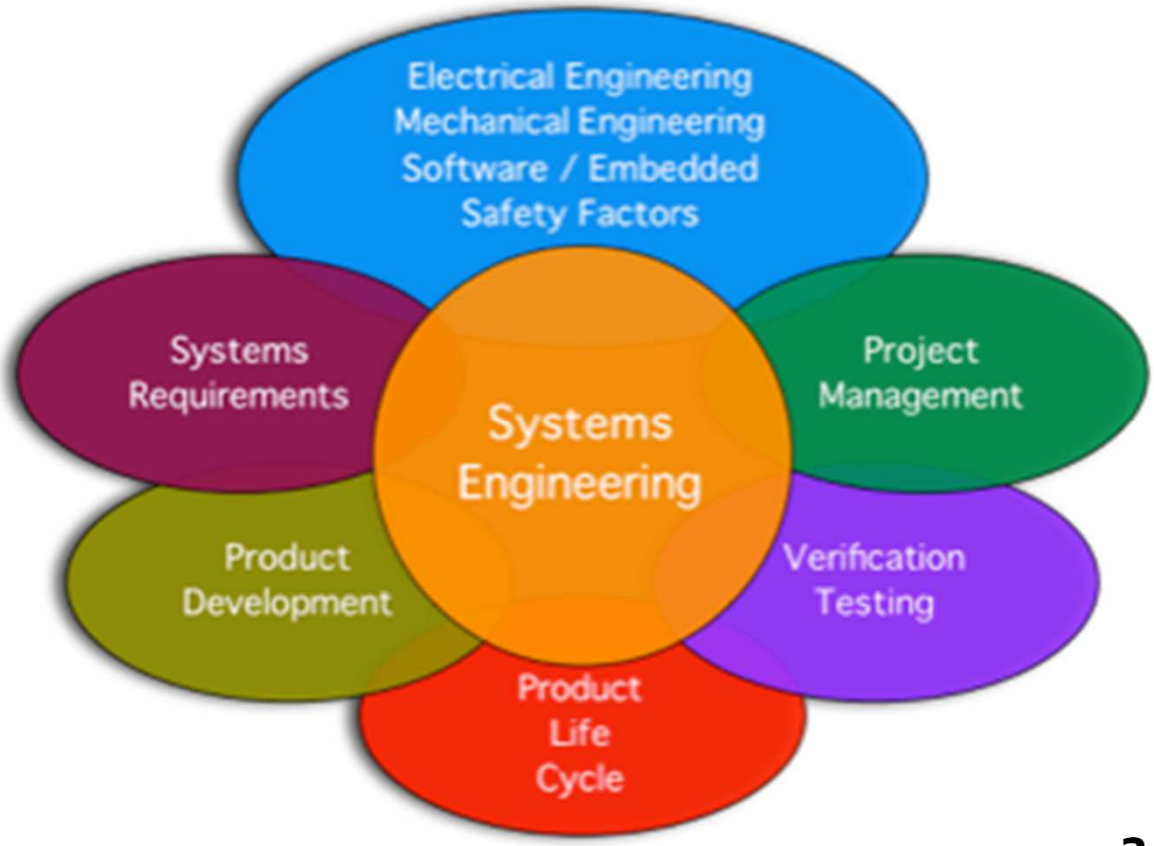
Índice de la Presentación

- Ingeniería de Sistemas
- Arquitecturas de Sistemas Intensivos y Complejos
- Sistemas de Sistemas (SoS)
- Construcción de la Arquitectura Software
- Evolución de la Arquitectura
- Arquitecturas en la Industria 4.0
- Conclusiones

Ingeniería de Sistemas

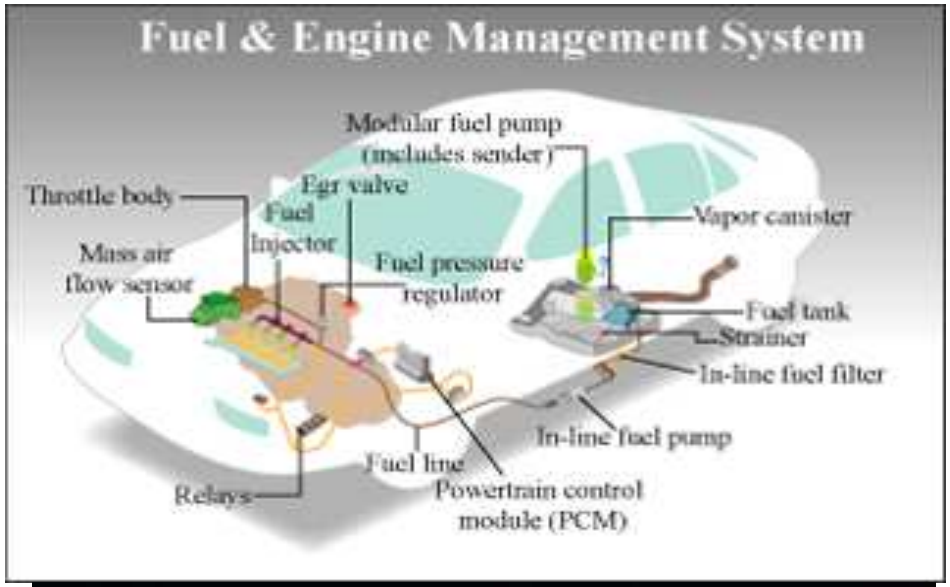
- "***Systems Engineering (SE)***" es una disciplina que involucra diversas sub-disciplinas para la construcción de sistemas intensivos y complejos
- **Sub-disciplinas**: Gestión de proyectos, Gestión de la configuración (SCM), hardware, Ingeniería Software, Verificación, Ingeniería de requisitos, etc.

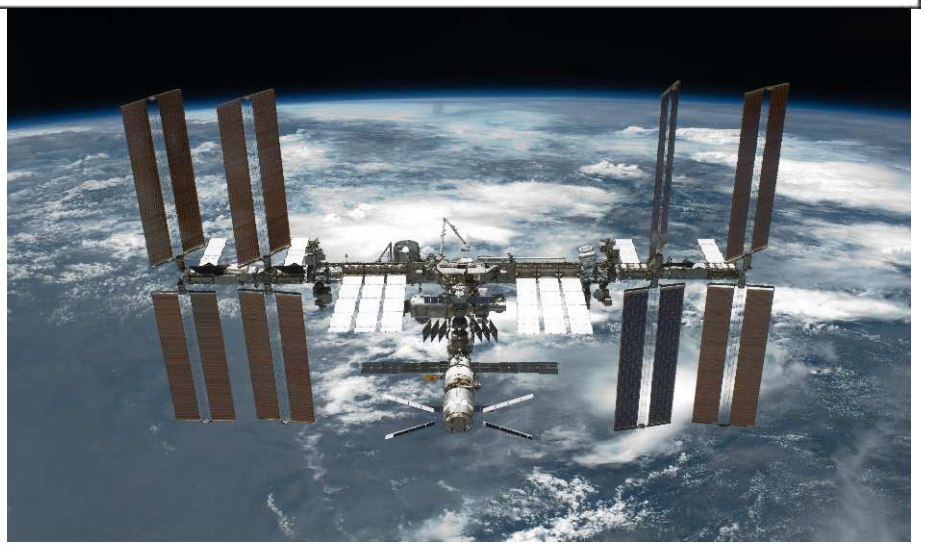
INCOSE (International Council on Systems Engineering) se encarga de los diversos estándares y certificaciones para Ingeniería de Sistemas complejos

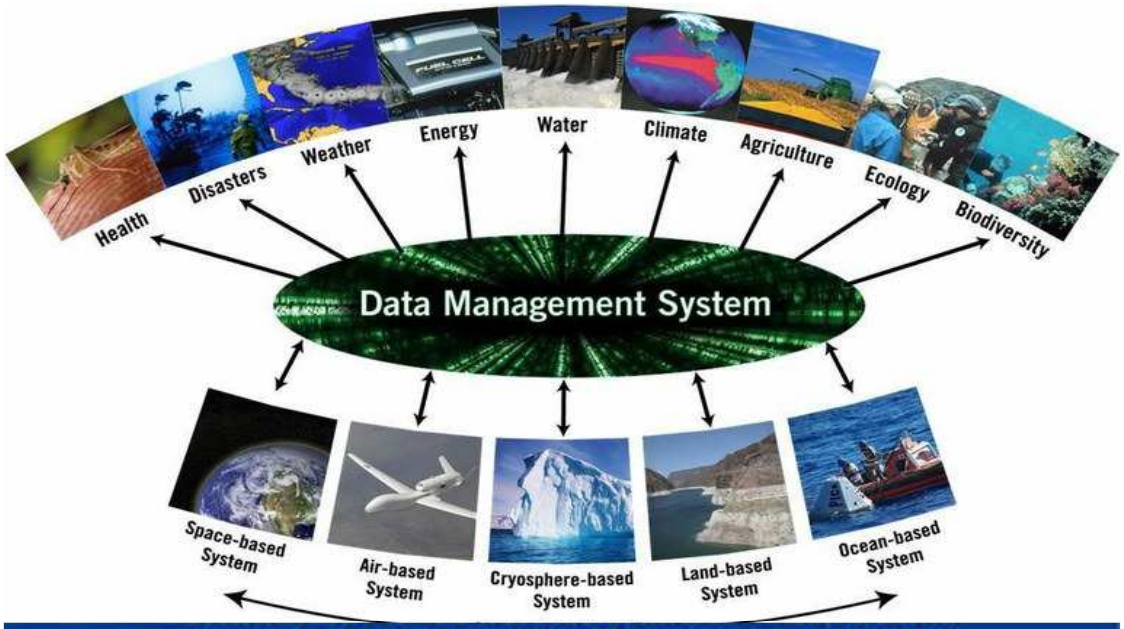


Ingeniería de Sistemas

Ejemplos de sistemas complejos







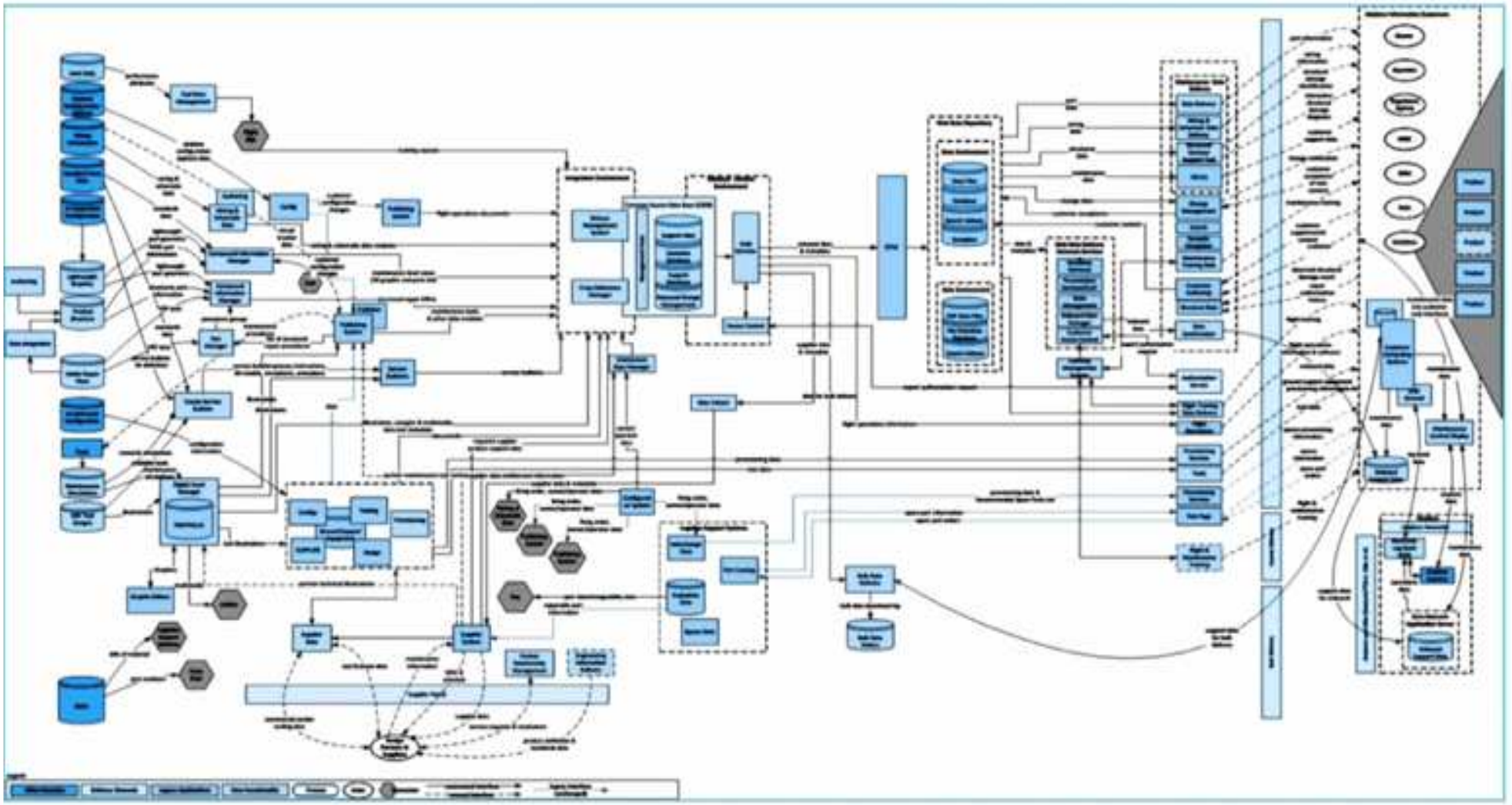
Ingeniería de Sistemas

- Se trabaja dentro de un dominio especifico para la construcción de productos que involucran HW y SW
- Los procesos y métodos para la construcción de sistemas complejos involucran a diversos “expertos en el dominio” y sub-disciplinas relacionadas
- La integración de productos HW/SW es uno de los aspectos importantes de la Ingeniería de Sistemas
- Dominios específicos:
  - Sistemas de información complejos (e.g., ATC)
  - Plataformas para trenes, aviónica, civiles (e.g., 112)
  - Infraestructuras civiles y de ingeniería (e.g, presas, energía)
  - Plataformas de transporte

Arquitecturas de Sistemas Intensivos y Complejos

Ejemplo I: Sistema empresarial

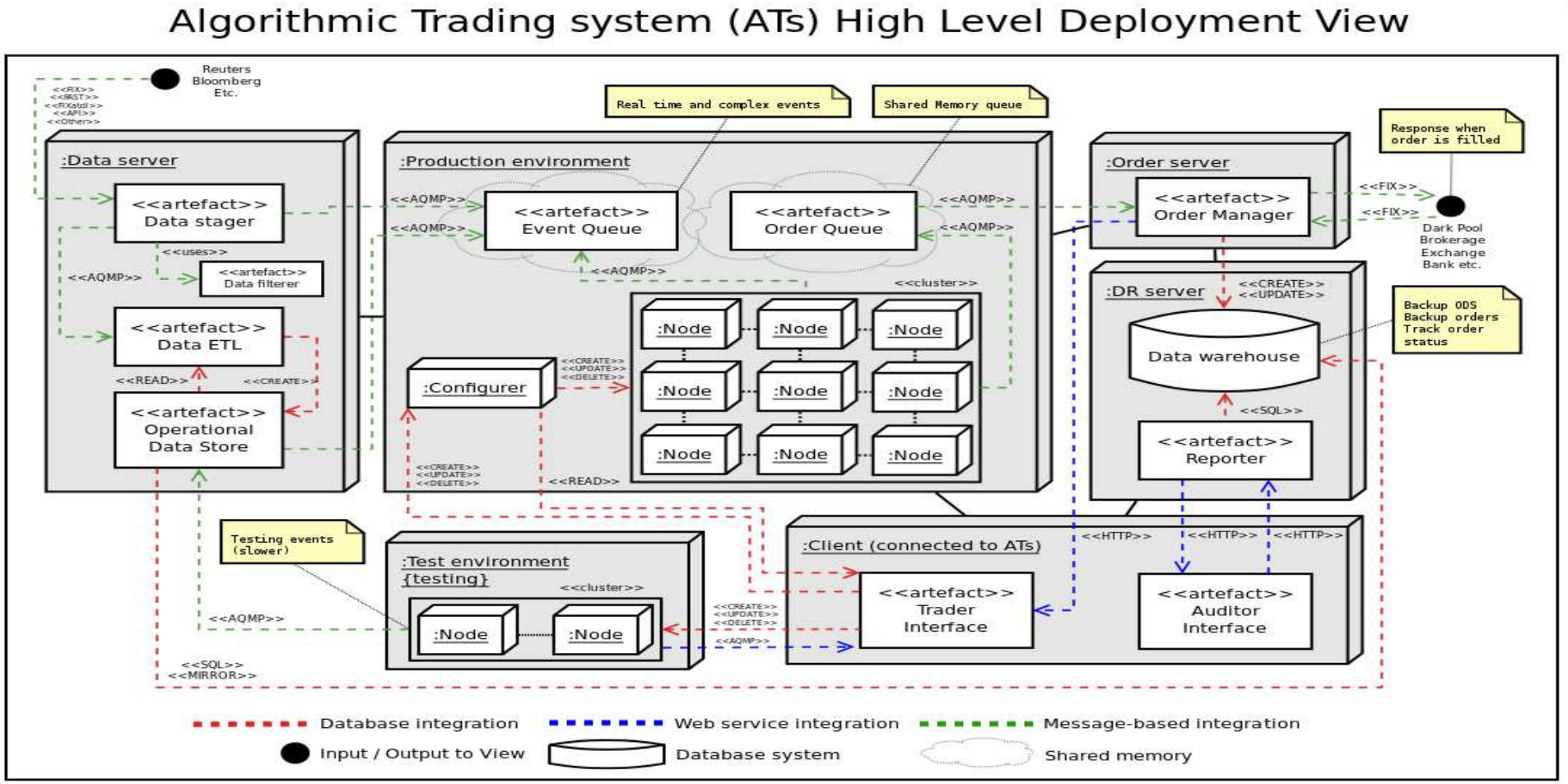
Sistema altamente complejo sin una arquitectura aparentemente definida o visible



Arquitecturas de Sistemas Intensivos y Complejos

Ejemplo 2: Sistema de Trading

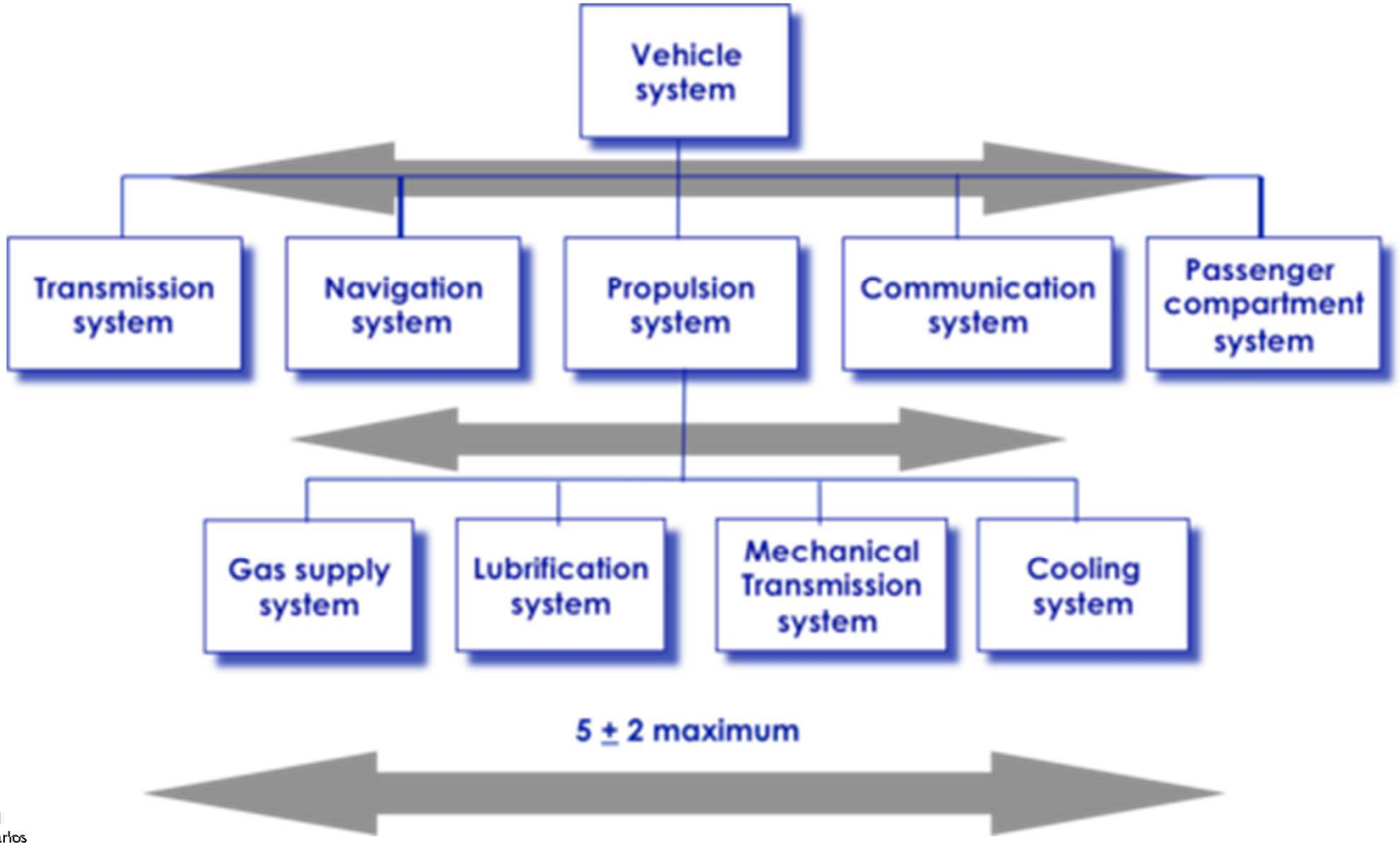
En este diseño al menos vemos una arquitectura de capas, subsistemas y componentes bien definidos, pero no los detalles internos



Arquitecturas de Sistemas Intensivos y Complejos

Ejemplo 3: Software vehículo

No hay detalle de las partes internas de la arquitectura, solo subsistemas y buses de conexión entre los diferentes módulos



Ejemplo 4: Apache Hadoop

Es el detalle de los módulos funcionales de Hadoop y sus relaciones extraído con Ingeniería Inversa. Es práctica mente imposible ver nada pero da idea de su complejidad.

Sistemas de Sistemas (SoS)

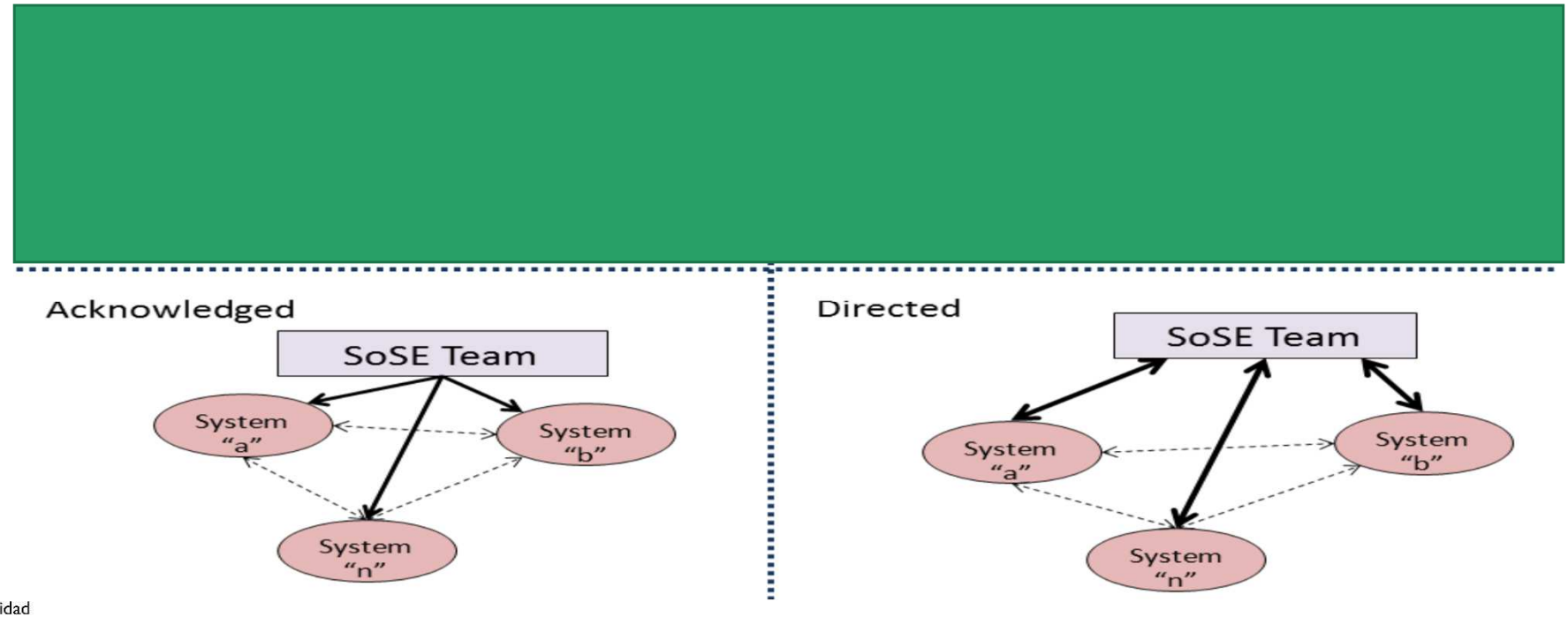
- **Systems of Systems (SoS)** es un término moderno que recoge aspectos de la Ingeniería de Sistemas del concepto de ULS ("Ultra-Large Systems") del SEI (*Software Engineering Institute*)
- SoS se refiere a la construcción moderna de sistemas complejos compuestos por diversos sub-sistemas
- **Systems of Systems Engineering (SoSE)** es de un ámbito superior a SE y acuñado en el ámbito militar pero expandiéndose al civil
- Al igual que en SE, involucra diferentes dominios, disciplinas y expertos
- Los diferentes sub-sistemas se integran para proporcionar una funcionalidad de orden superior

A SoS is an integration of a finite number of constituent systems which are independent and operatable, and which are networked together for a period of time to achieve acertain higher goal. (Jamshidi 2009)

Sistemas de Sistemas (SoS)

Tipos de SoS basados en la independencia de sus constituyentes

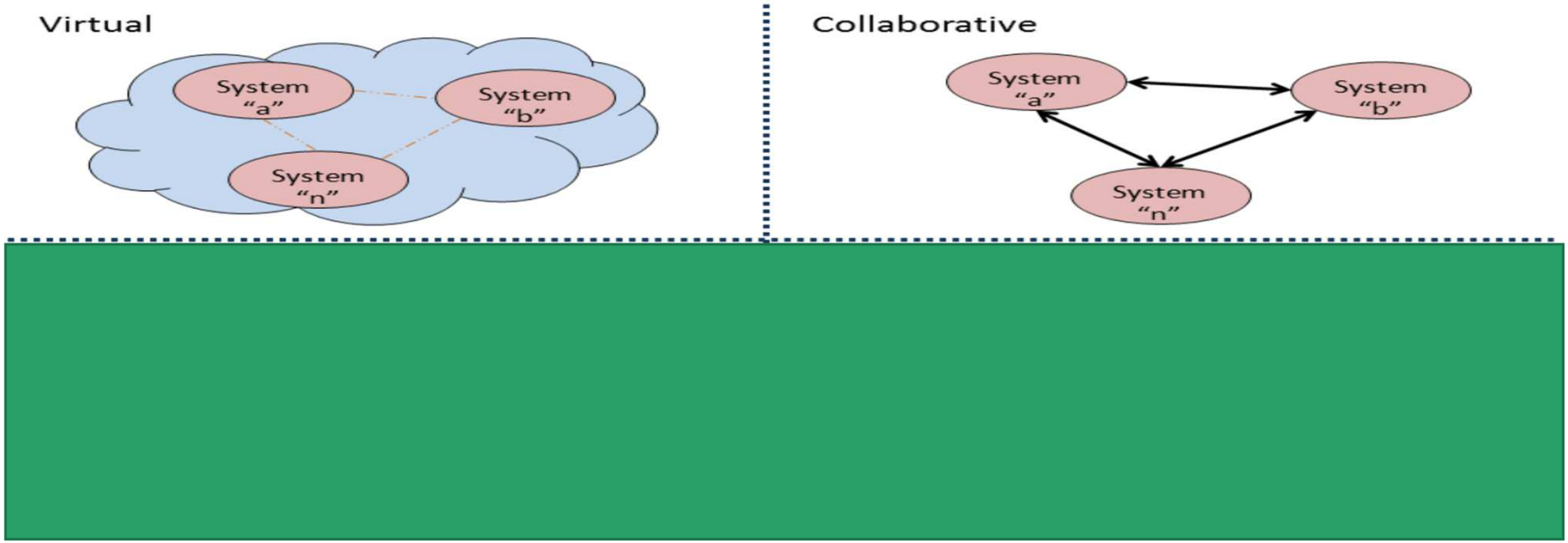
- **Dirigidos**: orientado a cumplir un propósito específico en el que sus sub-sistemas constituyentes están subordinados al SoS. Los componentes de los subsistemas pueden operar independientemente, pero subordinados al control central (e.g., sistemas de salud)
- **Reconocidos**: el SoS tiene objetivos reconocidos pero los constituyentes retienen objetivos independientes respecto a su desarrollo. Los cambios en el sistema se basa en acuerdos (e.g., sistemas de control y mando militar)



Sistemas de Sistemas (SoS)

Tipos de SoS basados en la independencia de sus constituyentes

- **Colaborativos**: los componentes del sistema interactúan de manera más o menos voluntaria para cumplir un objetivo. Los actores centrales deciden como permitir o denegar un servicio (e.g., sistemas de respuesta regionales frente a crisis)
- **Virtuales**: el SoS carece de una gestión centralizada. Un comportamiento de orden superior suele aparecer de manera no visible como consecuencia de las interacciones entre los constituyentes (e.g. Servicios de Internet)



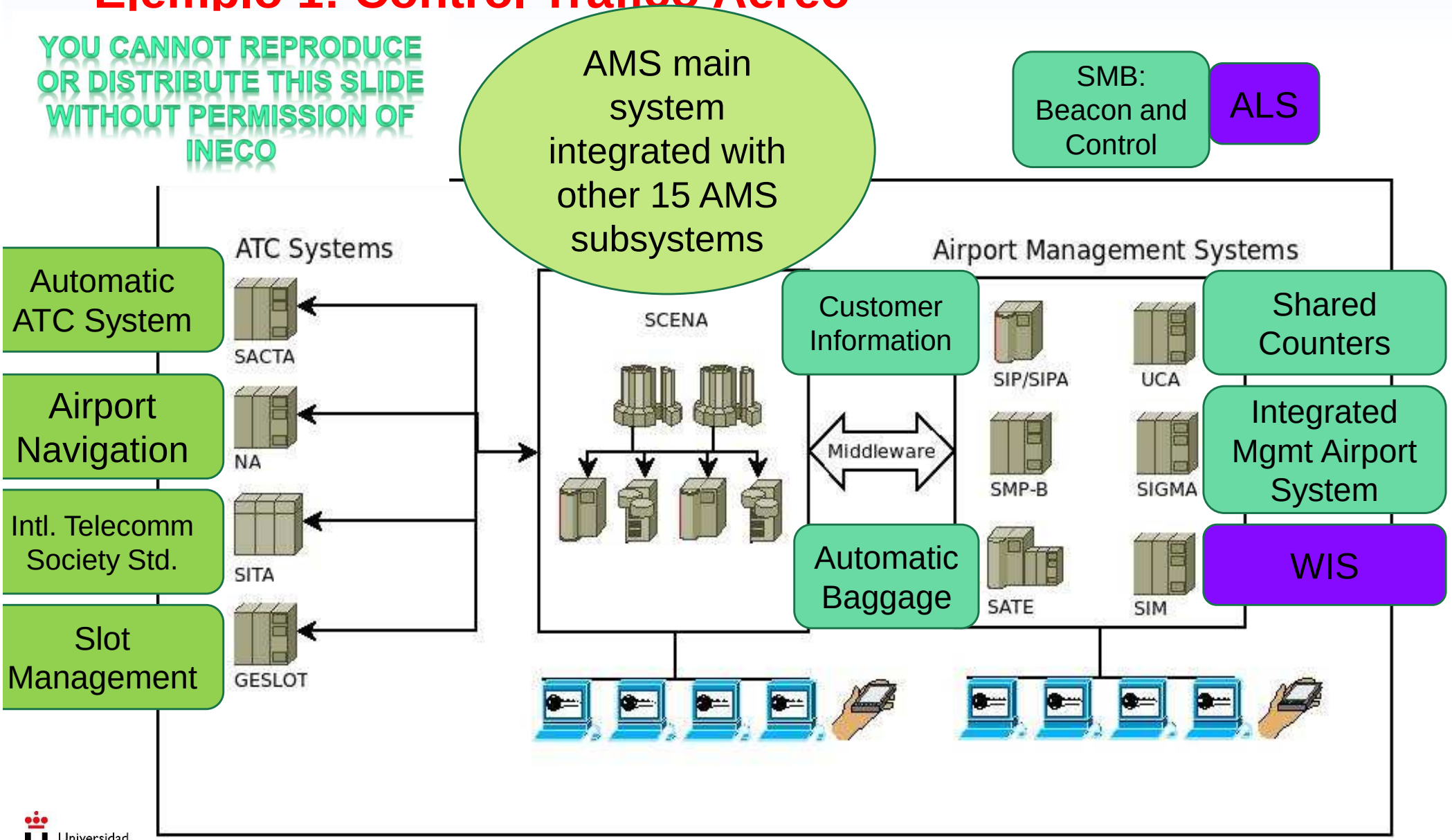
Sistemas de Sistemas (SoS)

Ejemplo 1: Control Tráfico Aéreo (AMS)

- SoS complejo en tiempo real para operaciones aeroportuarias
- Utiliza una variedad de sensores y otros dispositivos para recolectar información
- Requisitos de seguridad importantes y regulatorios
- Integración de Middleware AMS con sistemas de terceros (“third-party systems”)
- Los subsistemas AMS deben cooperar de forma estrecha
- Gestionar cambios y actualizaciones en subsistemas críticos (e.g, cambia la categoría del aeropuerto y requiere nuevos sistemas)

Sistemas de Sistemas (SoS)

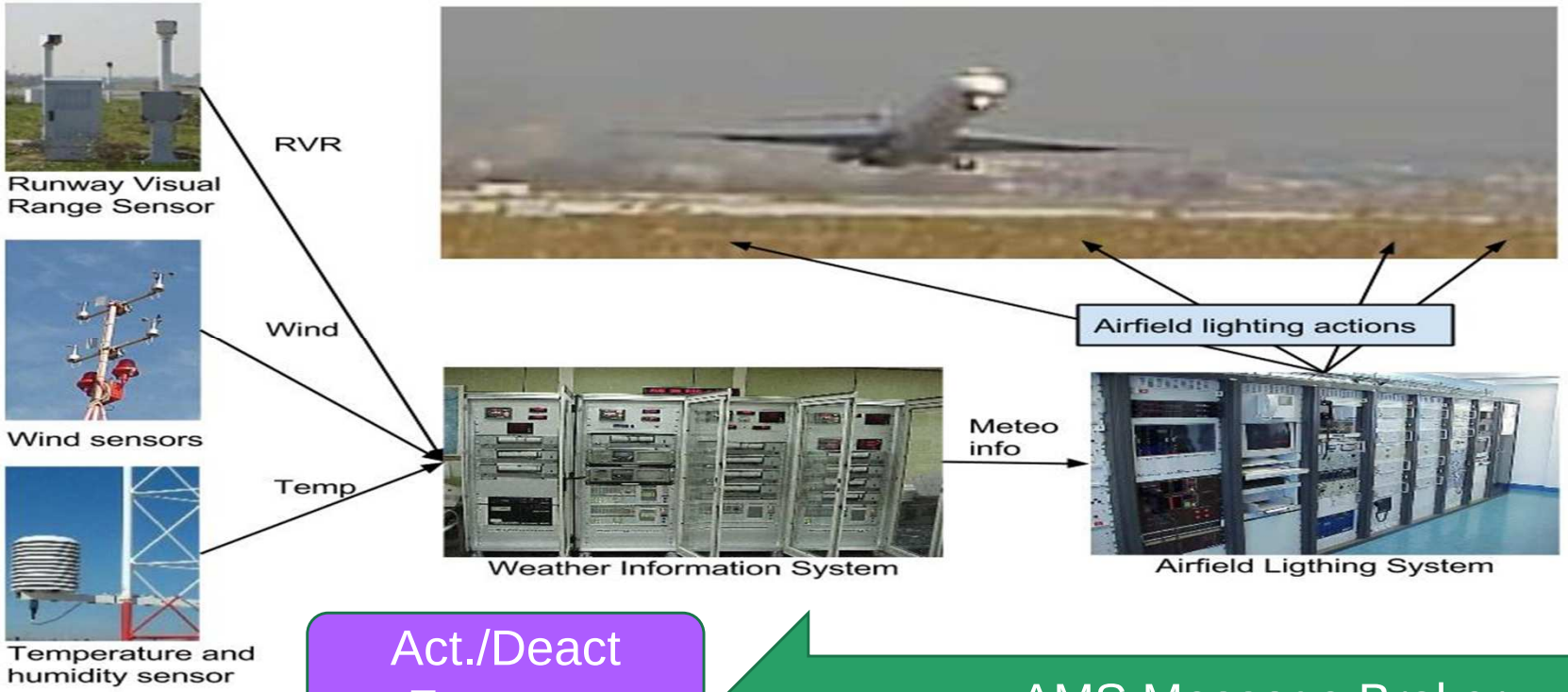
Ejemplo 1: Control Tráfico Aéreo



Sistemas de Sistemas (SoS)

Ejemplo 1: Control Tráfico Aéreo

Tareas colaborativas entre WIS ("Weather Information Systems") y ALS ("Airfield Lightning System")



Act./Deact Features

AMS Message Broker

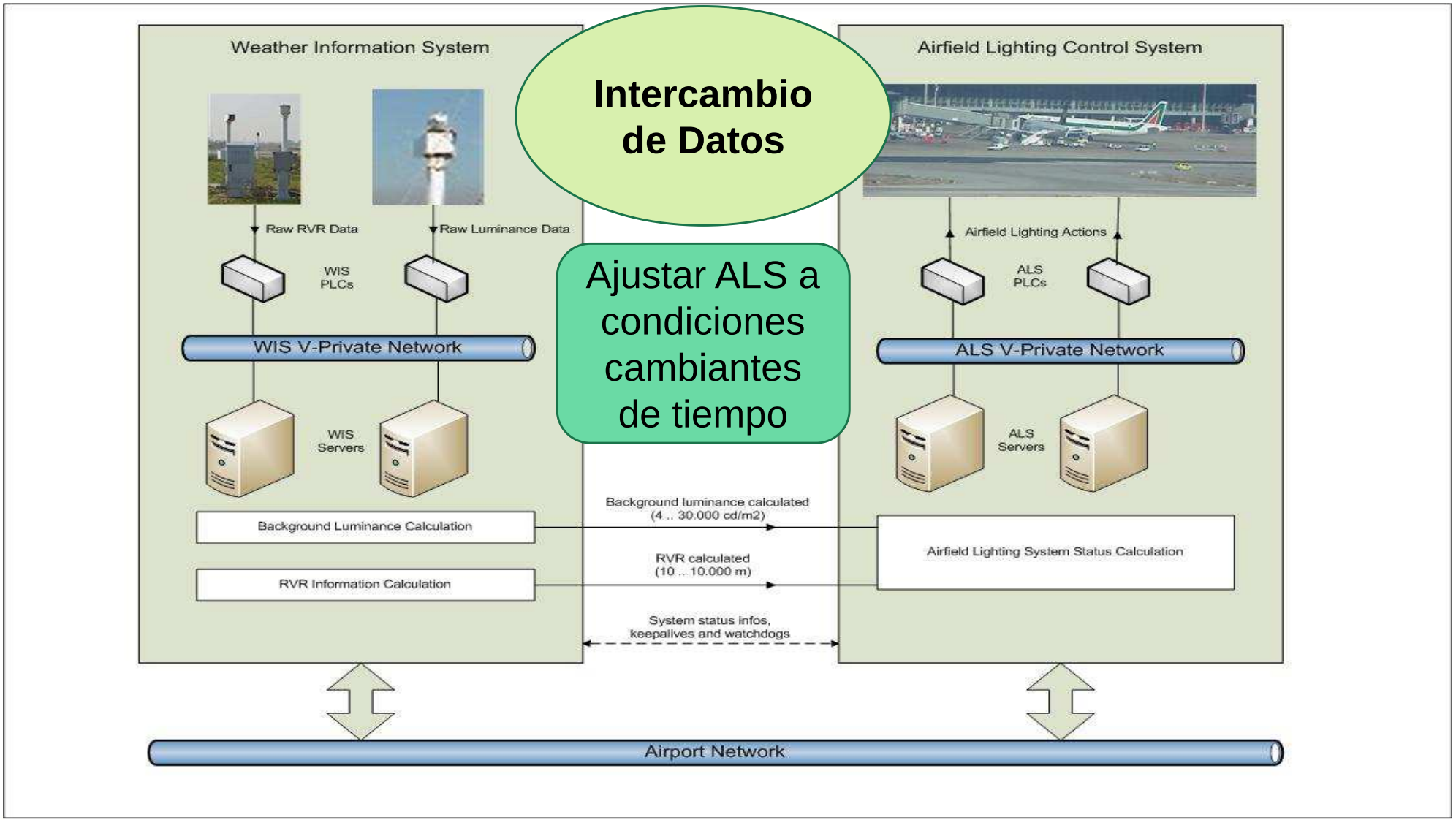
Cancel Operations

Send Priority Messages

Send/Receive Data

Sistemas de Sistemas (SoS)

Ejemplo 1: Control Tráfico Aéreo



Sistemas de Sistemas (SoS)

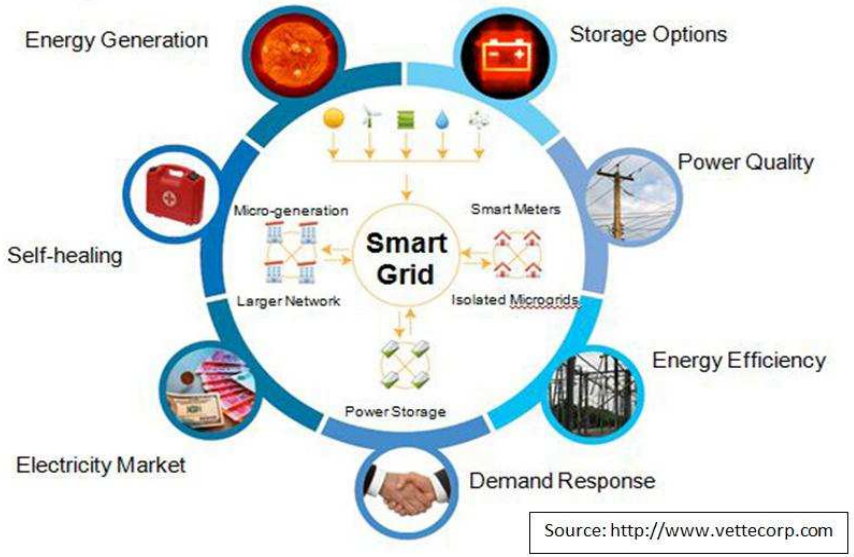
Ejemplo 2: Smart Grids

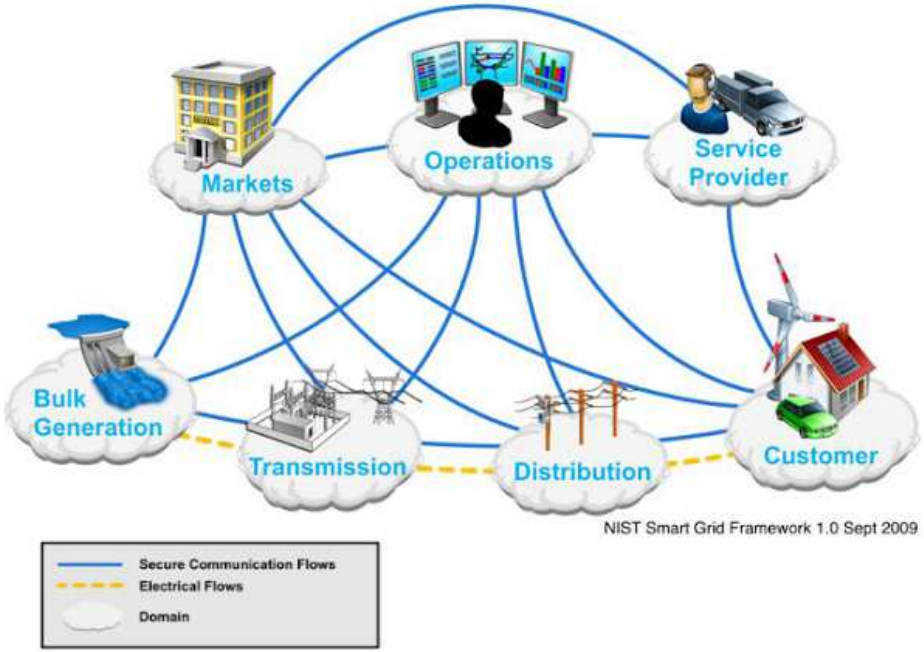
- Sistemas grandes para proporcionar electricidad
- Se encargan de la distribución de energía
- Gestionan diferentes tipos de fuentes de energía (e.g., plantas energéticas, solares, nucleares, geológicas)
- Su gestión pasa de ser centralizada-> descentralizada
- Las redes de energía heterogéneas demandan nuevas formas de gestión
- Diferentes elementos que hay que interconectar y gestionar
- Son sistemas ***pervasivos*** en cuanto a comunicación
- Los sistemas de energía inteligente (“Smart Energy Systems” y “Smart Distribution Grids”) promocionan la ***sostenibilidad*** del sistema (Green Systems) a través de la reducción del consumo de energía

Sistemas de Sistemas (SoS)

Ejemplo 2: Smart Grids

- Disponen de diferentes elementos de: Control y Medición
- Medición inteligente de la energía
- Necesidad de una infraestructura flexible que escale la red de manera eficiente
- Necesitan además ser interoperables y poseer capacidades en tiempo real para satisfacer picos de demanda o caídas de red
- Múltiples sensores que miden datos en tiempo real de diferentes fuentes
- Necesidades de Big Data

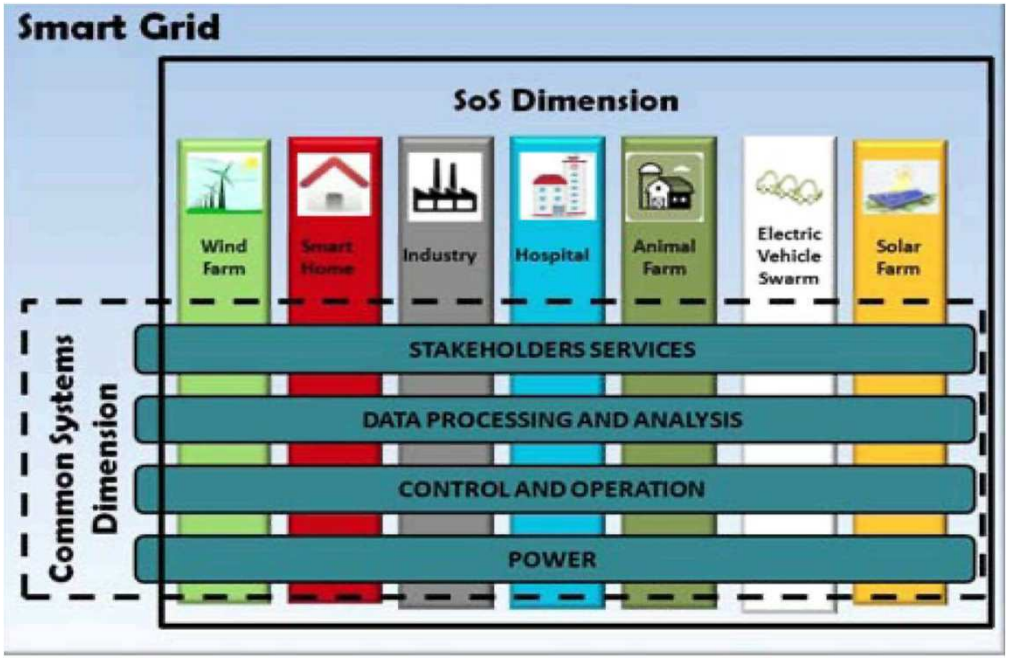


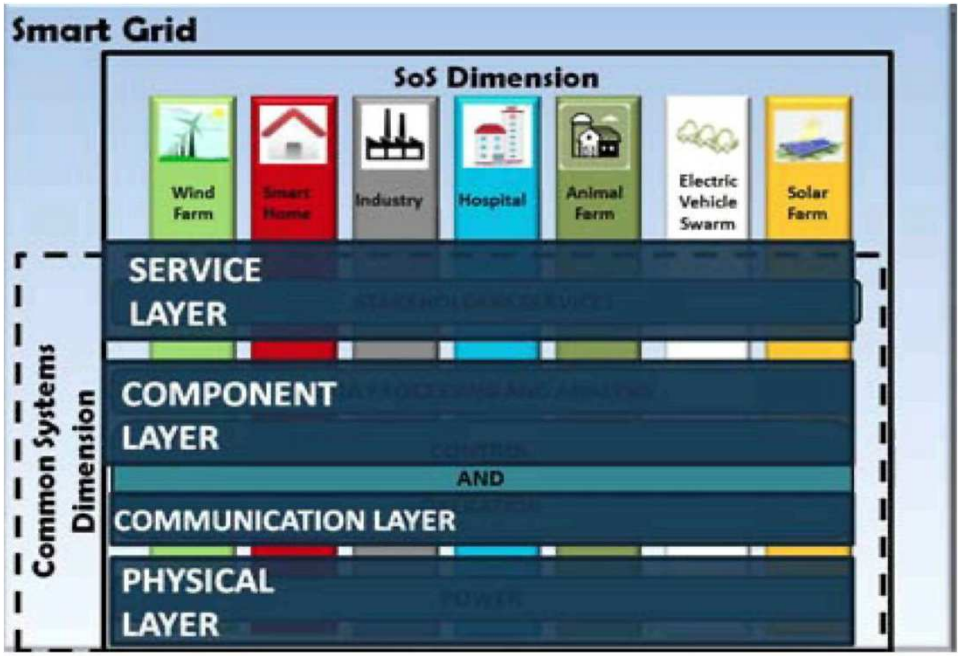


Sistemas de Sistemas (SoS)

Ejemplo 2: Smart Grids

- Dimensiones derivadas de proyectos Españoles/E.U. [Pérez et al, SeSoS, 2016]
- Sistemas: Energía, Operación y Control, Procesado Datos, Servicios proporcionados a usuarios
- Cálculos intensivos (e.g., emitir miles de facturas en horas, cálculo de las mejores franjas horarias de consumo, etc.)





Sistemas de Sistemas (SoS)

Ejemplo 3: Sistema Integrado de Vigilancia Exterior (SIVE)

- Detectar a larga distancia las embarcaciones que se aproximen a nuestro litoral.
- Identificar el tipo de embarcación y a sus tripulantes con el fin de comprobar la posible actuación ilegal de los mismos.
- Coordinar el seguimiento de la embarcación, utilizando para ello los medios marítimos, aéreos y terrestres con que cuenta la Guardia Civil.
- Interceptar a los presuntos delincuentes o auxiliar a los inmigrantes irregulares

Sistemas de Sistemas (SoS)

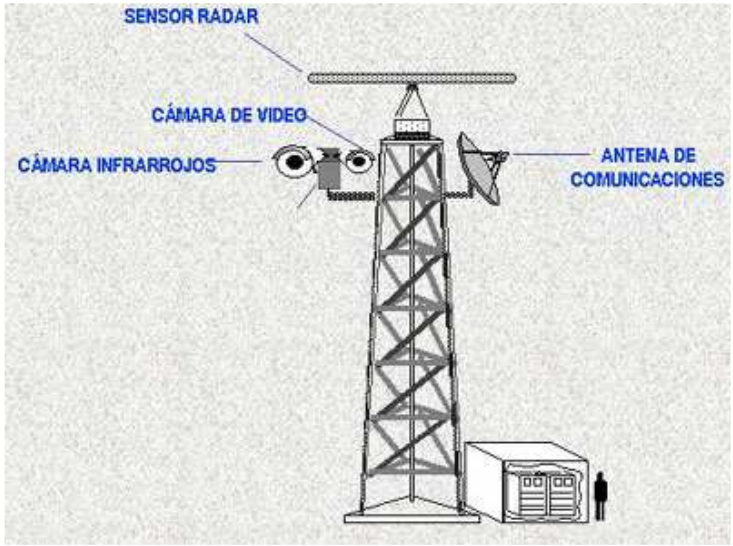
Ejemplo 3: Sistema Integrado de Vigilancia Exterior (SIVE)

(13.09.2014). Recuperado el 14 de septiembre de 2015 del sitio web de El País: http://goo.gl/FX06M6

Sistemas de Sistemas (SoS)

Ejemplo 3: Sistema Integrado de Vigilancia Exterior (SIVE)

- **Subsistema de detección (estación de sensores):** detectan embarcaciones a larga distancia (10 Km) y transmite la señal a varios monitores de TV. Incluye cámaras de video diurnas de gran alcance y cámaras nocturnas de infrarrojos
- **Subsistema de comunicaciones:** comunicaciones en tiempo real de imágenes, voz y datos
- **Subsistema de mando y control**: Centralización de señales, control de estaciones y emisión de órdenes a unidades de interceptación



Sistemas de Sistemas (SoS)

Ejemplo 3: Sistema Integrado de Vigilancia Exterior (SIVE)

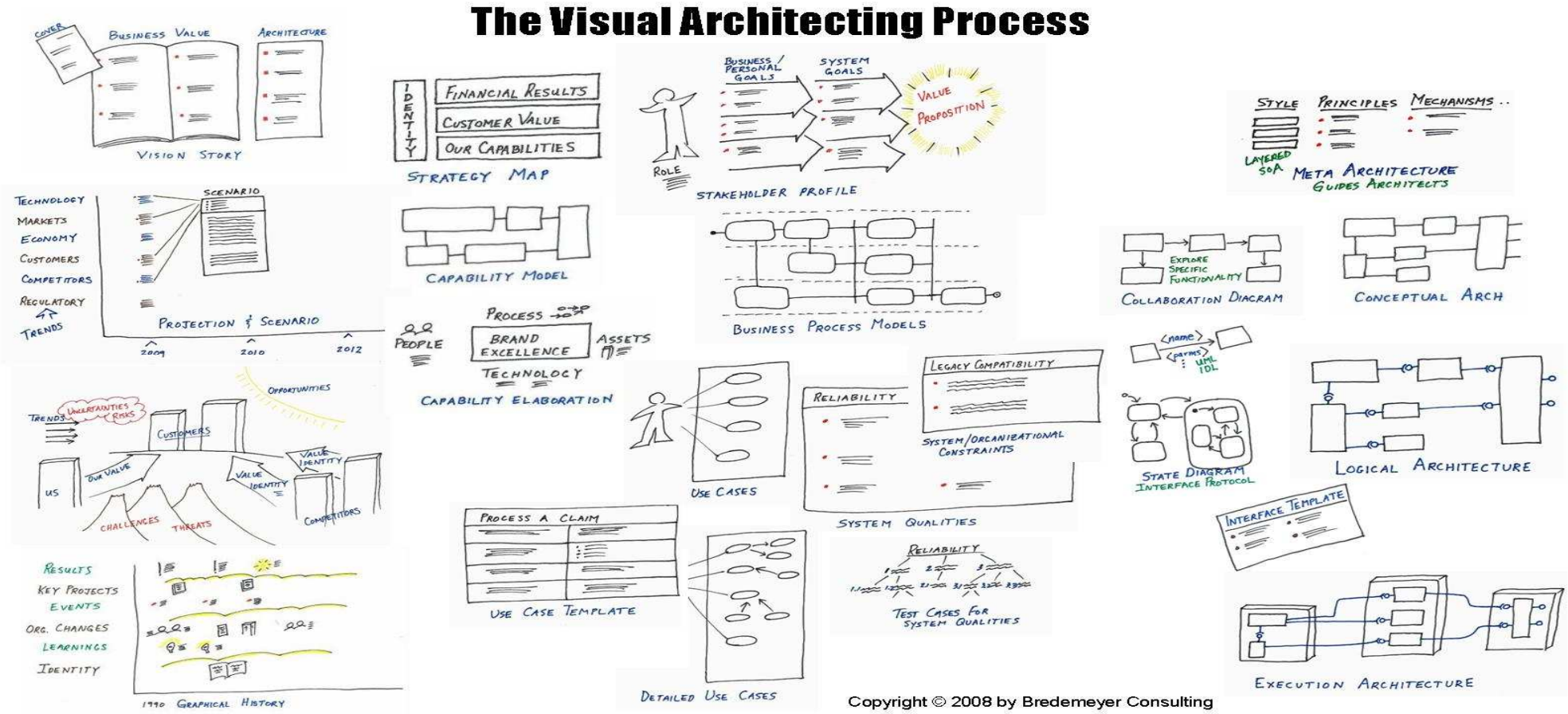
Construcción de la AS

- Seleccionar cuantas vistas hacen faltan (e.g, casos de uso, dinámica, despliegue, lógica)
- Seleccionar los diagramas apropiados para cada vista
- El mayor problema viene de la vista estructural del sistema Clases y Paquetes)
- En la dinámica describir los aspectos más complejos de interacción
- Para sistemas con alto grado de interacción el estilo MVC es adecuado
- Para sistemas críticos, tiempo real y que utilizan sensores, las arquitecturas basadas en eventos son útiles

Construcción de la AS

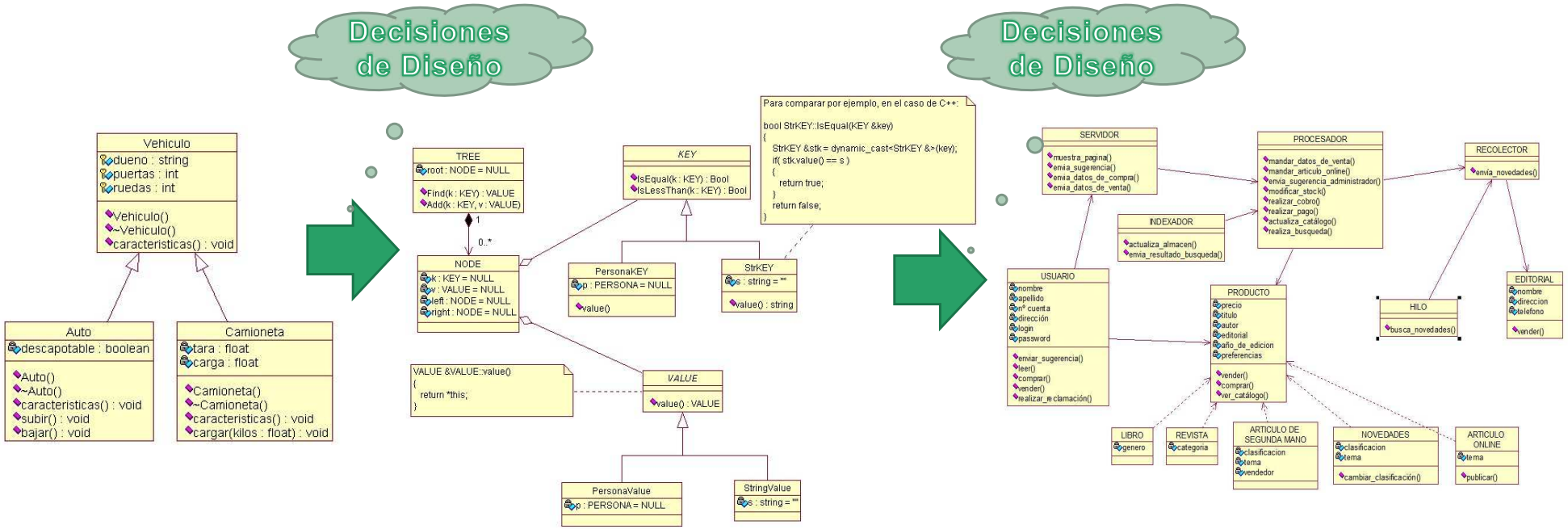
¿ Cómo crear la Arquitectura ?

No resulta fácil comenzar a hacer un “sketch” o diagrama de la arquitectura de un sistema complejo para lo cual no hay reglas bien definidas y que parece un proceso creativo y visual.



Construcción de la AS

- Una vez seleccionados los estilos arquitectónicos ver si existen patrones de diseño que solucionen problemas concretos (e.g., Observer)
- Normalmente suele haber una combinación de patrones
- La arquitectura se diseña en varias iteraciones donde se va refinando el diseño

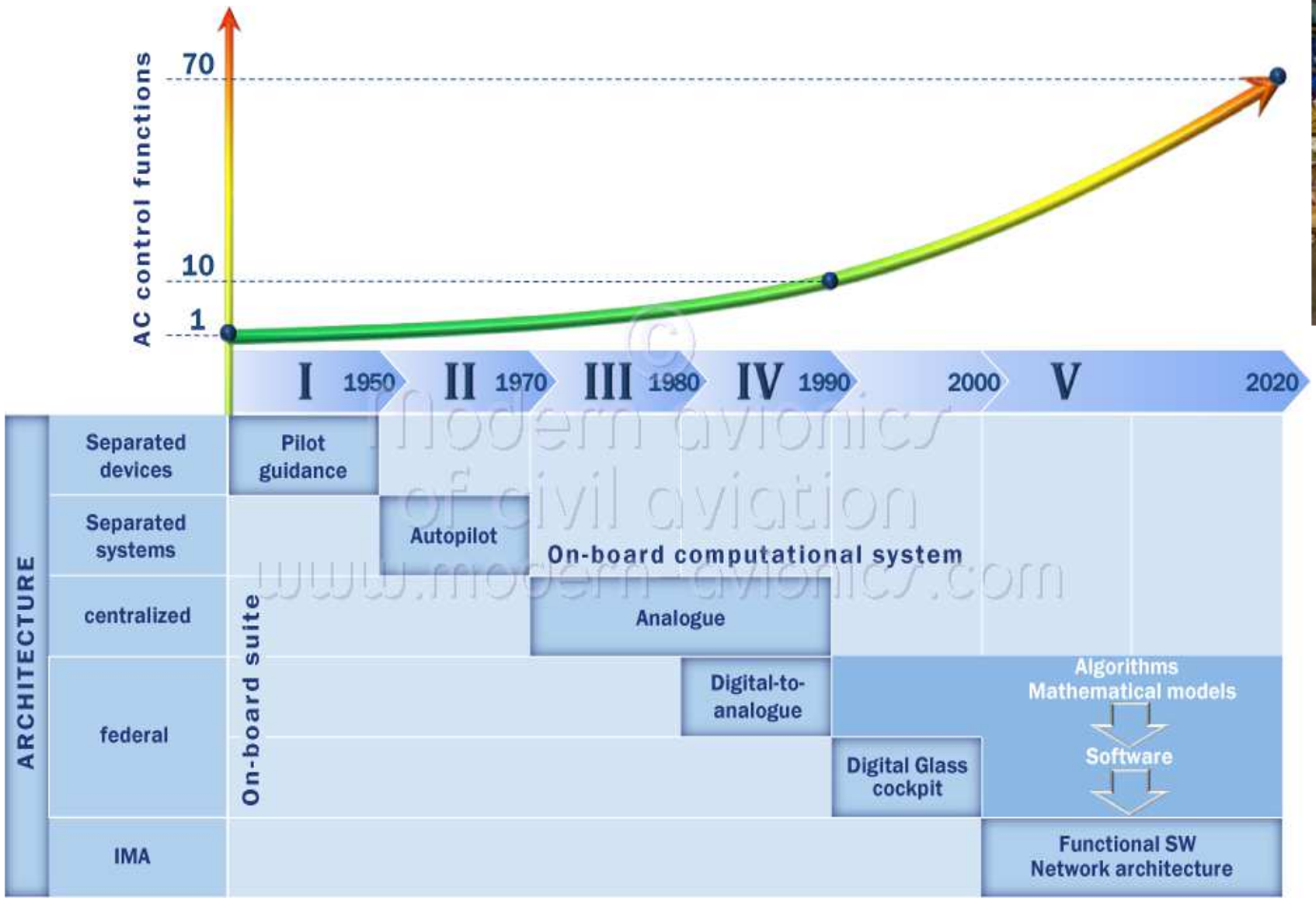


Iteraciones en el diseño de la arquitectura

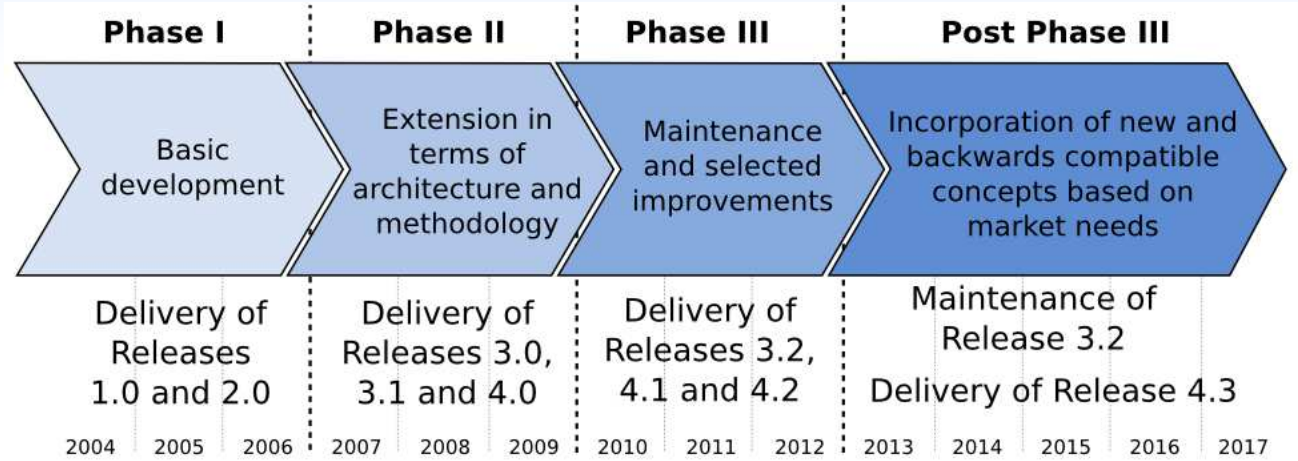
Evolución de la Arquitectura

- Una buena arquitectura debe mantenerse en el tiempo
- Debe ser suficientemente flexible para soportar cambios motivados por obsolescencia tecnológica o nuevos requisitos
- La evolución de la arquitectura debe reflejar el histórico de los cambios

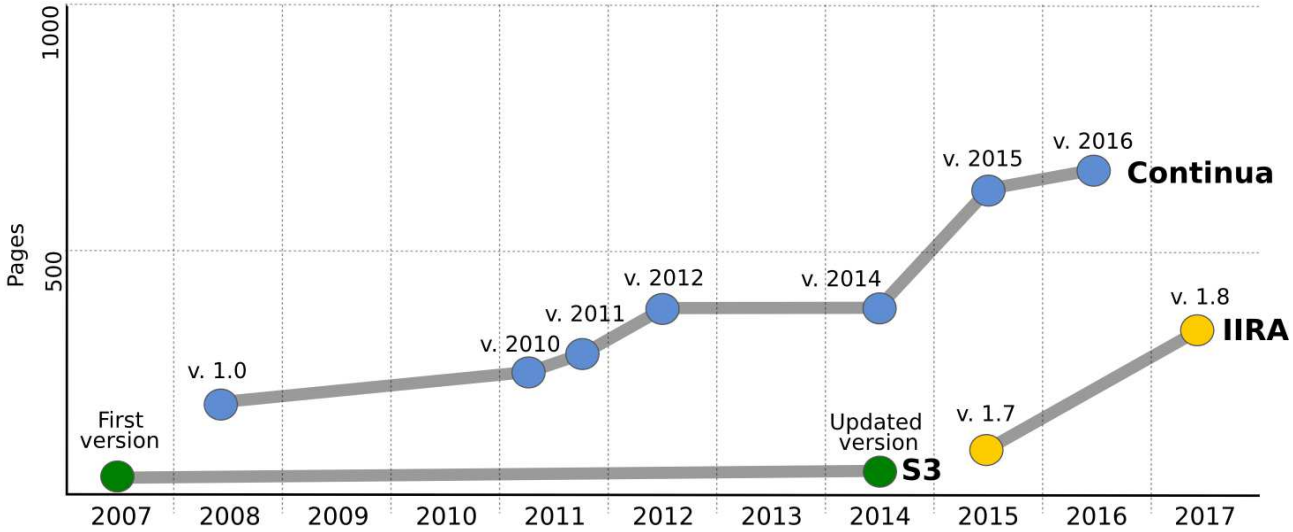
Evolución de la Arquitectura



Evolución de la Arquitectura



Fases de evolución
y releases en AUTOSAR



Actualizaciones en
Arquitecturas de Referencia

IIRA: Industrial Internet Reference Architecture
S3: Amazon Simple Storage Service
Continua: Personal Tele-health Ecosystem

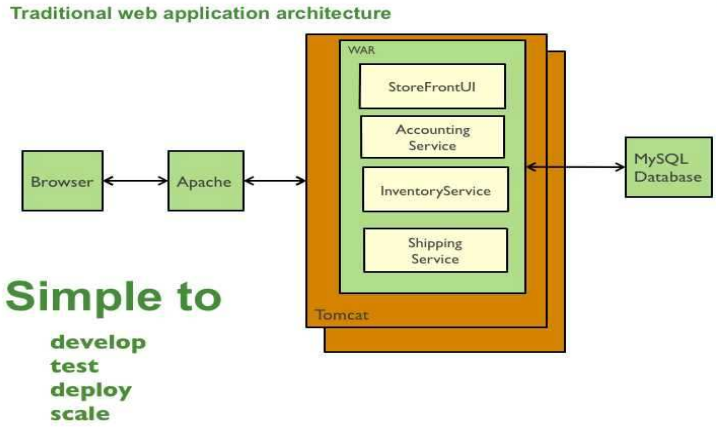
Evolución de la Arquitectura

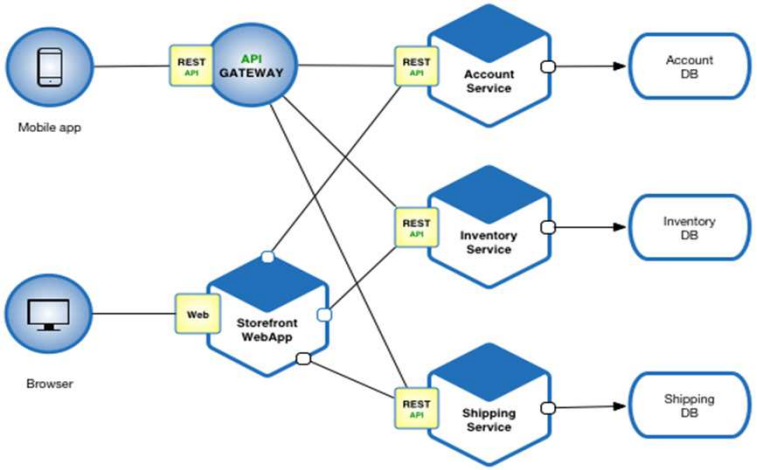
Arquitecturas de Micro-servicios

- Son arquitecturas de servicios para aplicaciones débilmente acopladas
- Permiten una continua entrega y despliegue de aplicaciones complejas
- Facilitan a las organizaciones evolucionar la “pila” tecnológica (capas de tecnología)
- Permiten la descomposiciones de capacidades de negocio
- Existe una colección de patrones para arquitecturas de micro-servicios
- Objetivo: Descomponer una arquitectura monolítica en micro-servicios

Evolución de la Arquitectura

De arquitecturas monolíticas a Micro-servicios





Simplicidad de desarrollo y despliegue

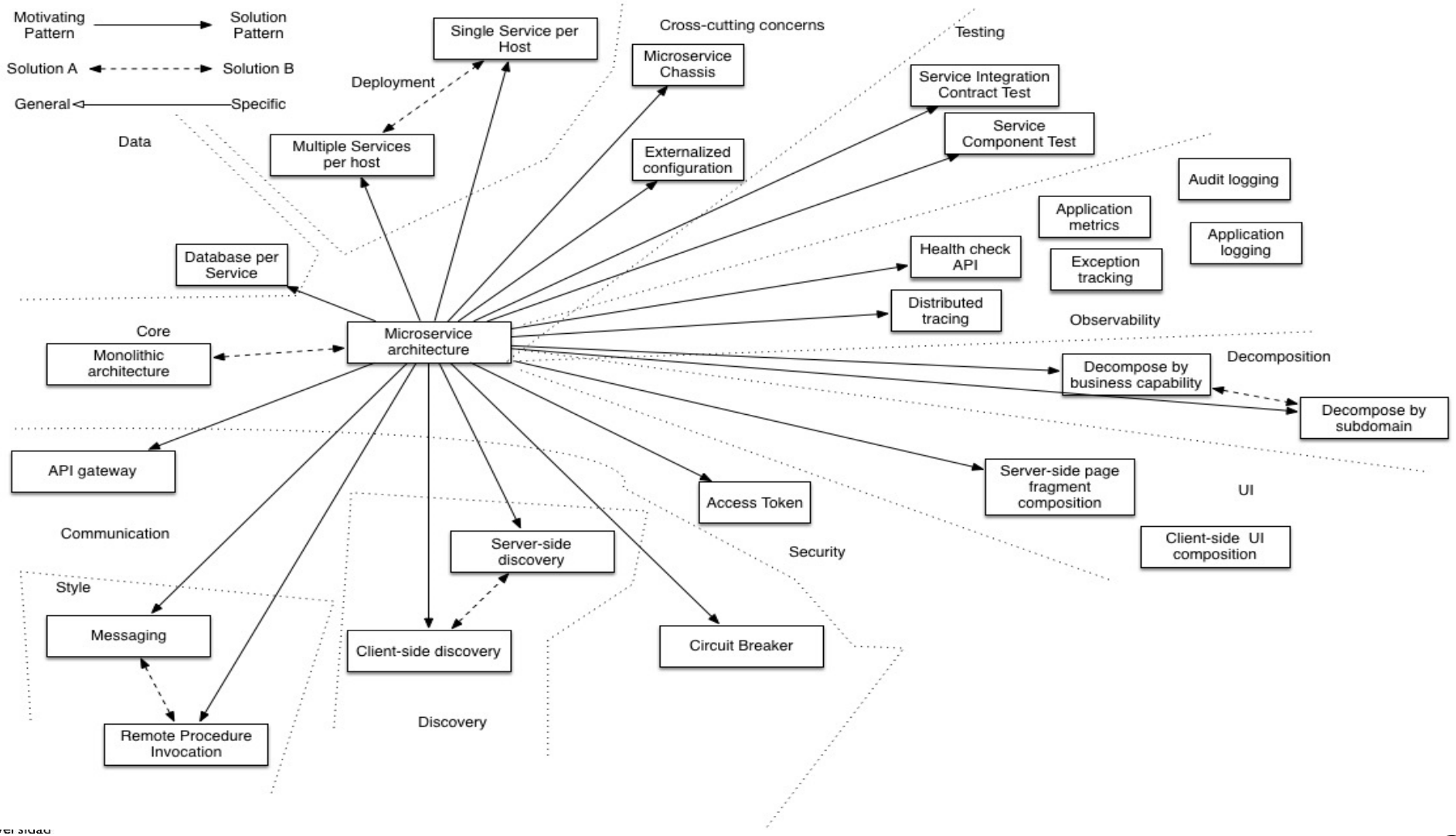
Difícil para despliegue continuo
Sobrecarga del contenedor Web
Difícil de escalar en varias dimensiones
Requiere una pila tecnológica durante largo tiempo

Micro-servicios son pequeños y fáciles en desarrollar
El desarrollo puede hacerse de manera independiente
Elimina parte de la dependencia tecnológica

Complejidad de sistemas distribuidos
Complejidad del testing
Complejidad del despliegue de diferentes MS
Consumo de memoria (de N instancias a NxM o más)

Evolución de la Arquitectura

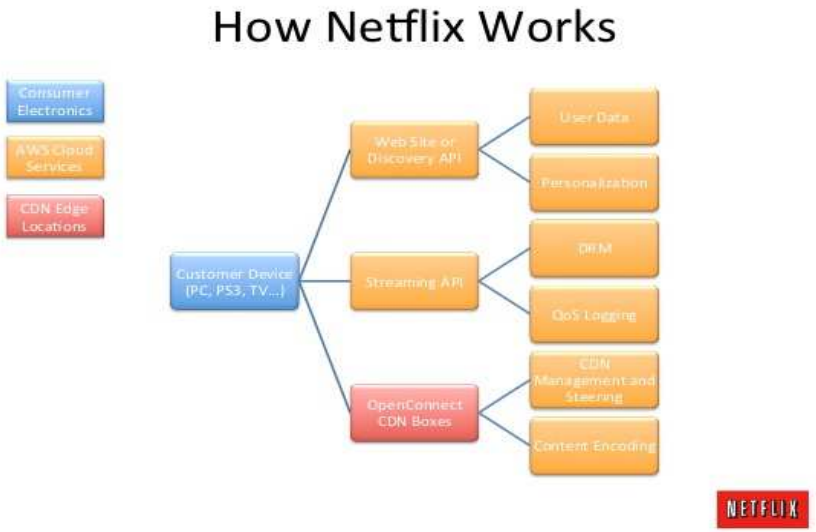
Patrones de Micro-servicios

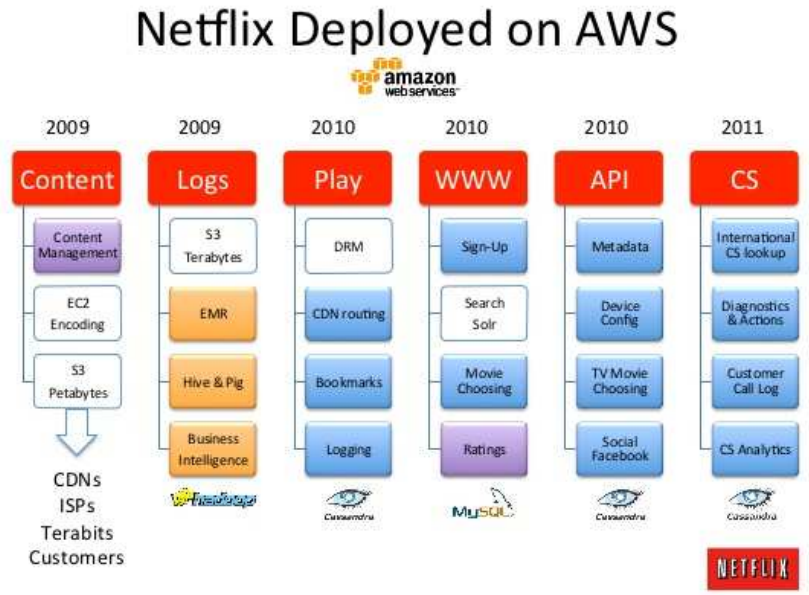


Evolución de la Arquitectura

Netflix

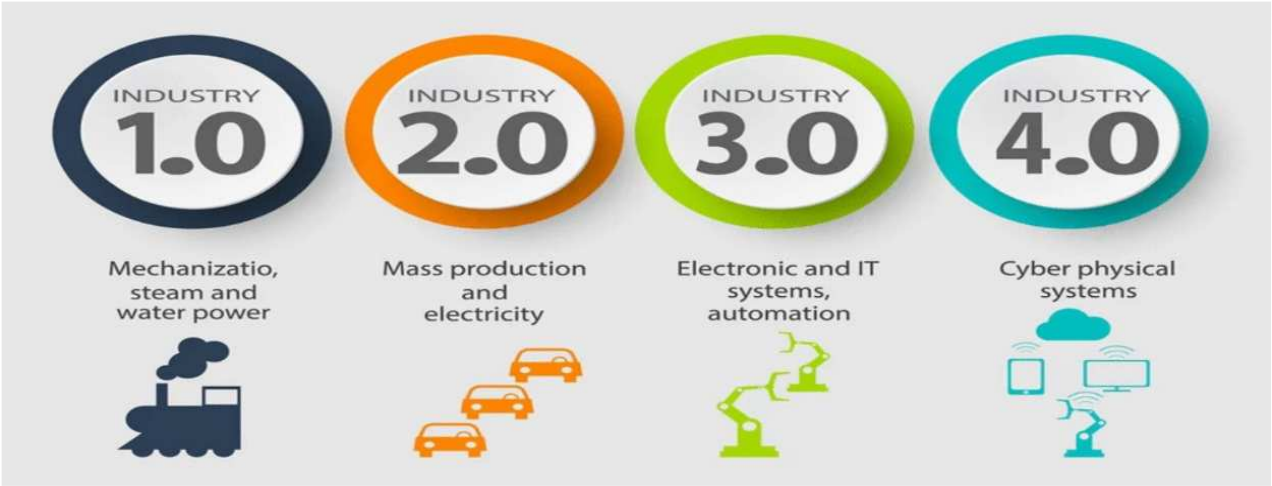
- Es un servicio de video streaming responsable del 60% del tráfico de Internet en 2020-2021 (Netflix, Youtube, Flow)
- Es una SOA transformada en arquitectura de micro-servicios
- Netflix API 20.000 Millones de peticiones / mes
- Cada llamada invoca una media de 6 llamadas al back-end de servicios





Sistemas en la Industria 4.0

- Orígenes de la Industria 4.0
  - El concepto de “Industrie 4.0” surge en una feria de Hannover (Alemania) en 2011
  - La idea es la de automatizar y digitalizar las fábricas alemanas, especialmente las que incluyen más software
  - El concepto de “Industry 4.0 - I4.0” (Europa) e “Industrial Internet of Things -IIoT” (USA) son equivalentes



Sistemas en la Industria 4.0

Objetivos de la I4.0

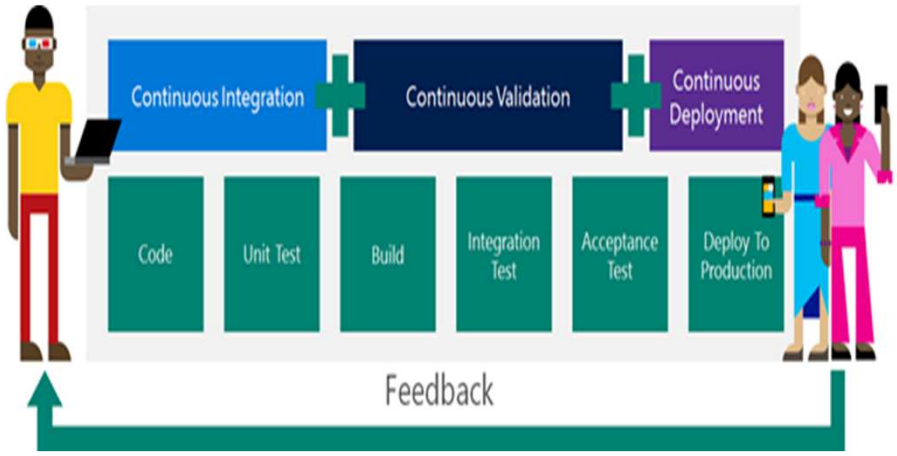
- Es una nueva revolución industrial de software para gestionar la complejidad de los sistemas software actuales
- La creación de factorías o fábricas inteligentes (“Smart factories”) incluye tecnologías y nuevos métodos de producción HW/SW como: IoT, Cypher-Physical Systems, Do it yourself!! (Robots, 3D printing)
- Optimizar la robotización de la Industria 3.0 (logística)
- Automatización e intercambio de datos en tecnologías de manufacturas

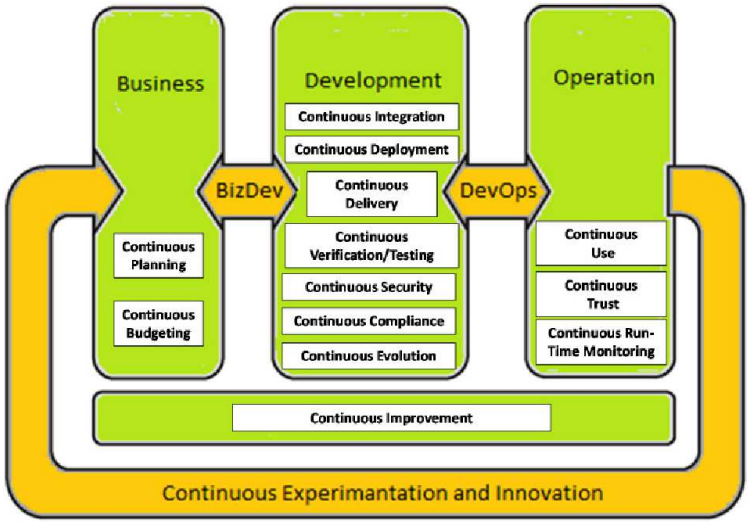
Sistemas en la Industria 4.0

- Tecnologías en la I4.0
  - **Big data y analística de datos**: colección de diferentes fuentes de datos para toma de decisiones inteligente
  - **Robots autónomos**: Robots que pueden interactuar con humanos
  - **Simulación**: Optimizacion y simulación del proceso productivo (e.g. sistemas de realidad virtual)
  - **Cyberseguridad**
  - **Industrial IoT**: Permitir a los robots y otros dispositivos comunicarse e interaccionar entre ellos y facilitar la toma de decisiones de manera descentralizada
  - **Cloud**: Compartición de datos fuera de los límites de la empresa
  - **Integración vertical y horizontal**

Sistemas en la Industria 4.0
Principios y Métodos

- **Interconexión**: Capacidad de los sistemas y máquinas, sensors y personas para comunicarse mediante protocolos de Internet de las Cosas (“Internet of Things (IoT)”)
- **Transparencia de la información**: Proporcionan a los operadores de la I4.0 información suficiente para la toma de decisions descentralizada
- **Continuous software engineering (CSE) y estrategia DevOps (Development + Operations)**

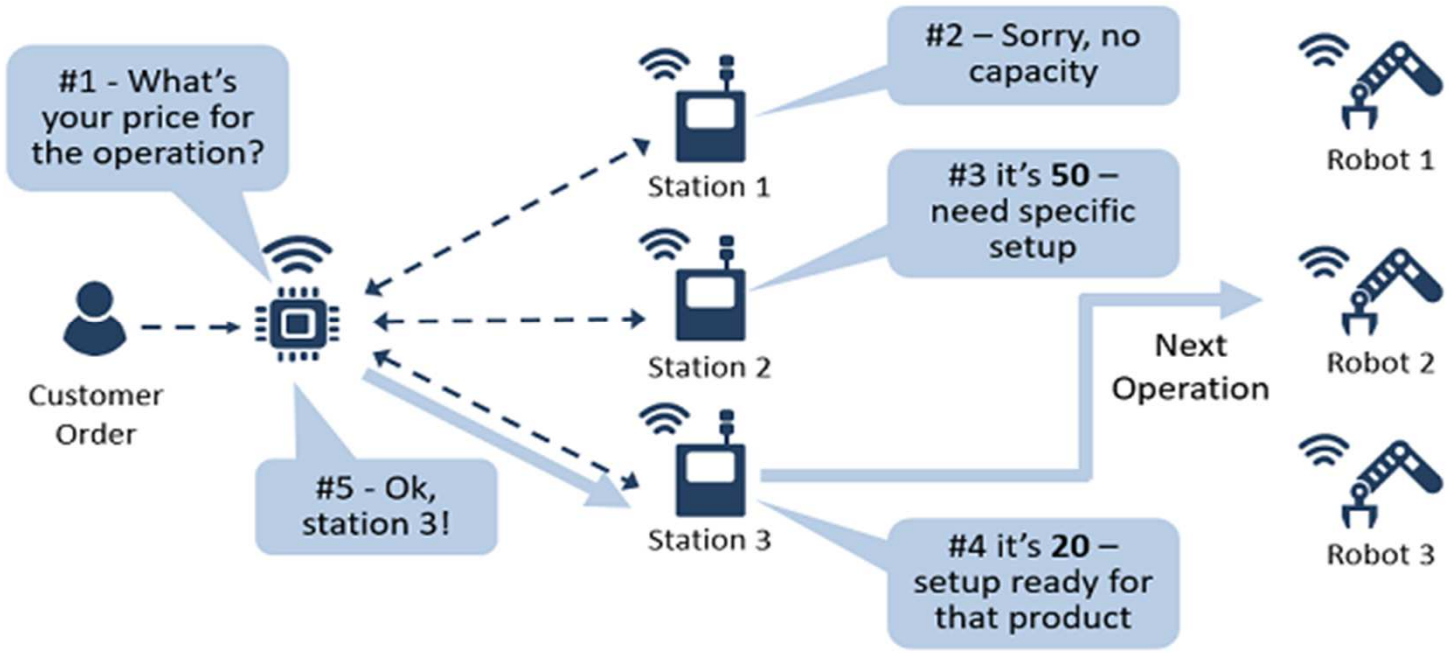




Sistemas en la Industria 4.0

Decisiones descentralizadas

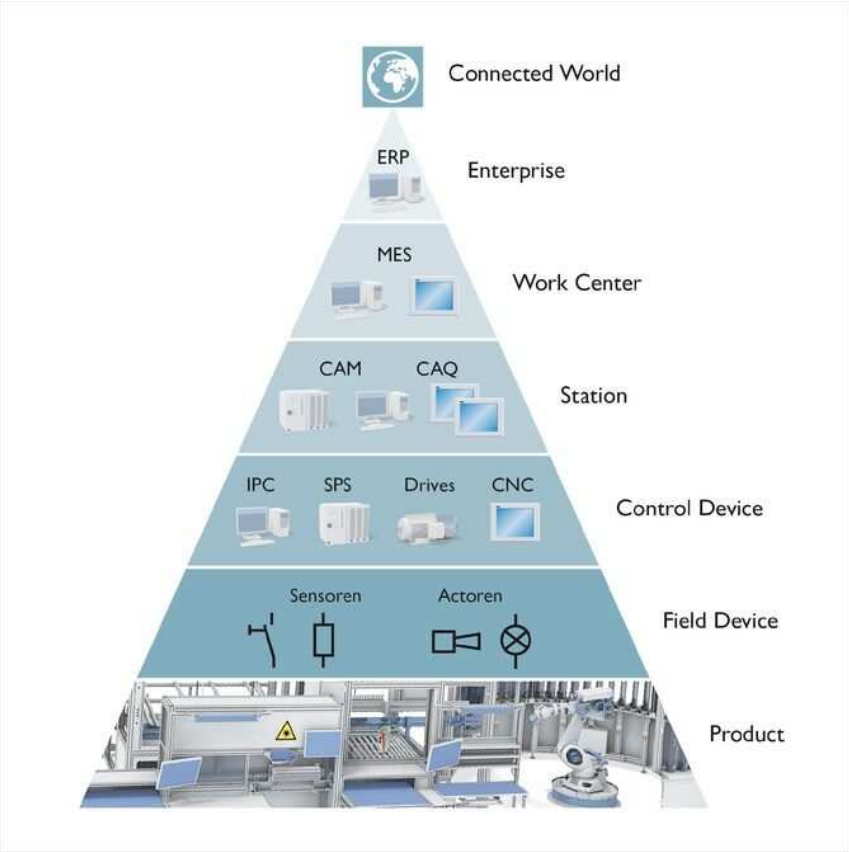
- **Interconexión**: Capacidad de los sistemas ciberfísicos para tomar decisions por si mismos y ejecutar las tareas de la manera más autónoma possible.



Sistemas en la Industria 4.0

Arquitecturas I4.0

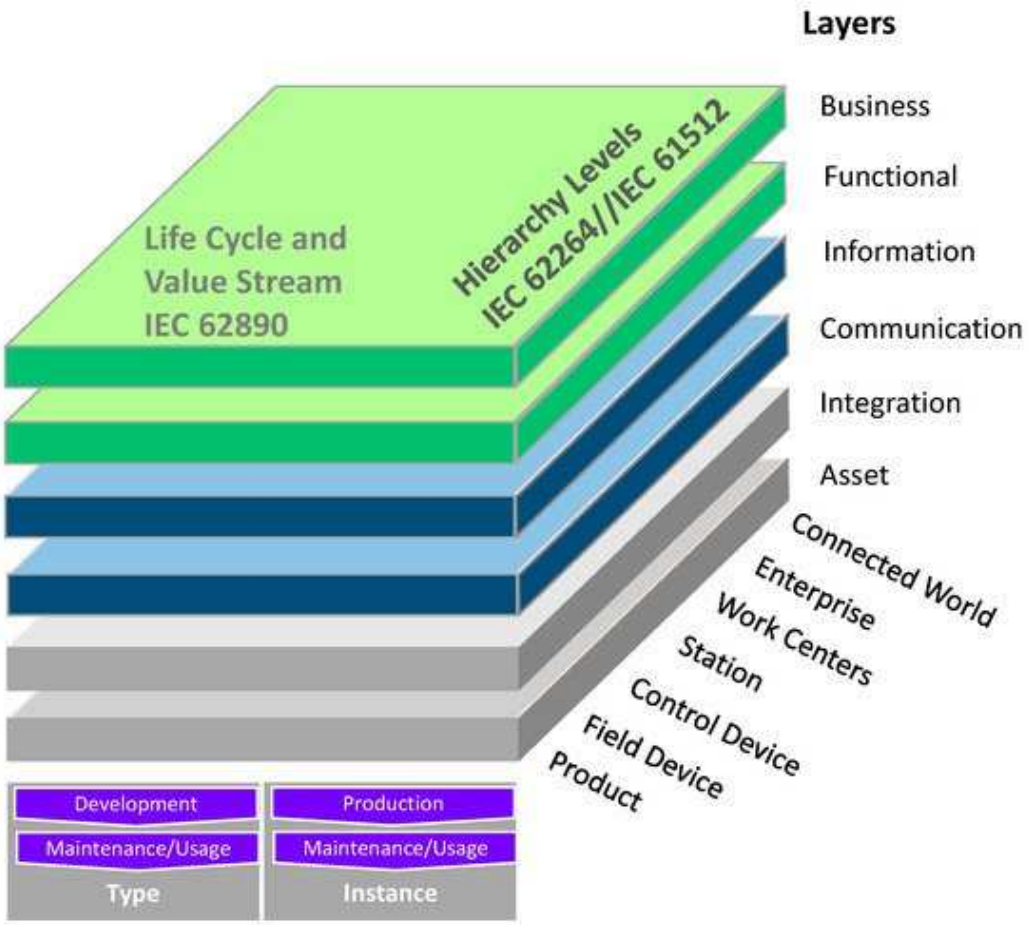
- Existen diversas arquitecturas de referencia para la Industria 4.0
- La mayoría se basan en el concep de RAMI 4.0 (Reference Architecture Model for I4.0)
- Multitud de protocolos y estándare para conectar los diferentes sistemas y plataformas
- Todos los elementos físicos de una factoría 4.0 se convierten en gemelos digitales (“Digital Twins”)



Sistemas en la Industria 4.0

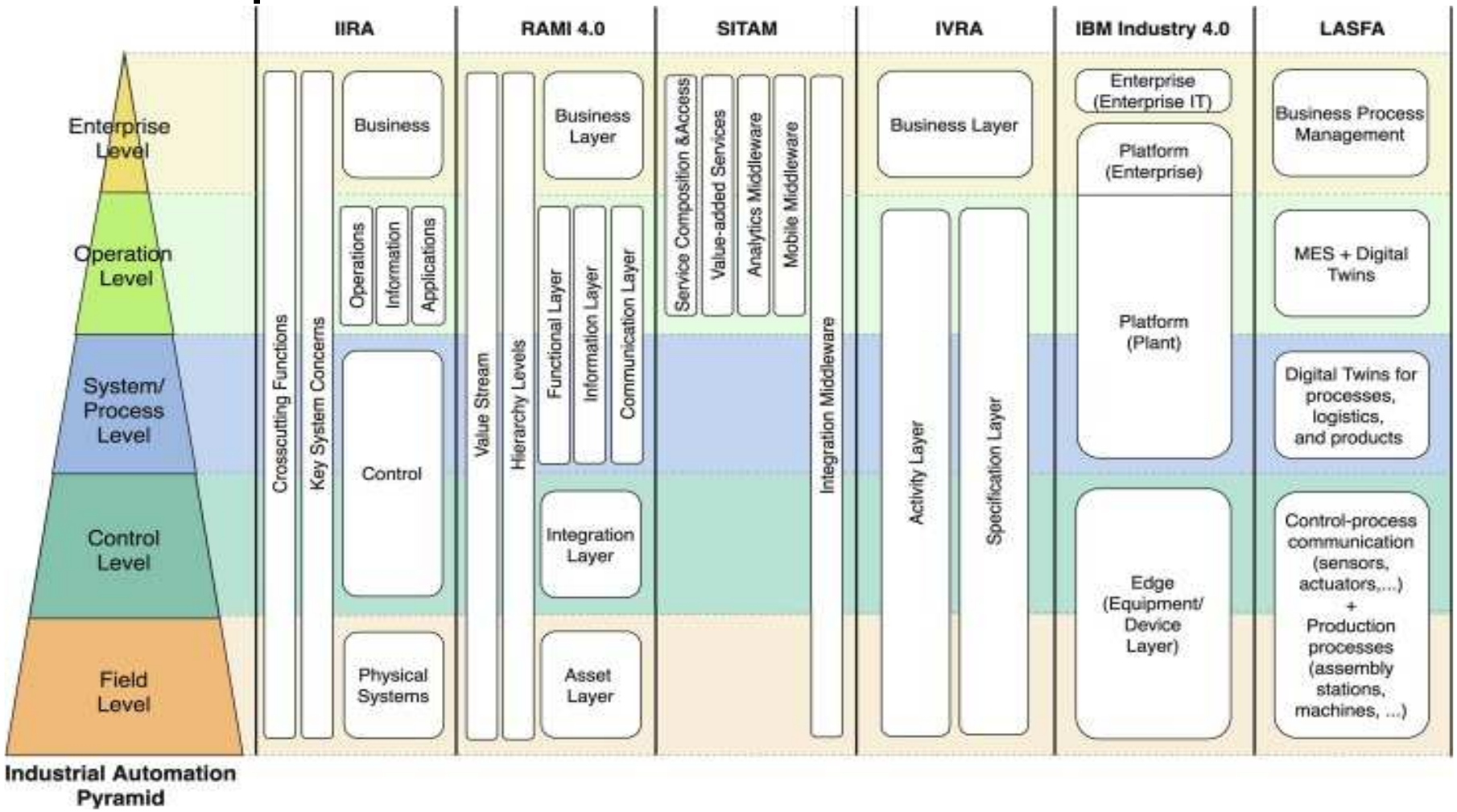
Arquitectura RAMI 4.0

- Arquitectura 3D presentada en 2015
- Reference architecture model for Industrie 4.0
- Describe los niveles jerárquicos de un sistema de manufacturas
- Los datos generados estan disponibles digitalmente
- La infraestructura IT se muestra en el eje vertical



Sistemas en la Industria 4.0

Otras arquitecturas I4.0 de referencia



Conclusiones

- Los sistemas de software intensivos son complejos de diseñar y mantener
- Las arquitecturas de un SoS debe ser flexibles y abiertas para soportar cambios en el tiempo
- La evolución de la AS debe reflejar el histórico de la versiones y de las decisiones de diseño tomadas
- La integración, despliegue e innovación continua son desafíos de la empresas y la construcción de sistemas modernos y complejos de hoy en día
- Las arquitecturas de la Industria 4.0 conectan los sistemas ciber-físicos con métodos inteligentes de producción
- En I4.0, los elementos del proceso productivo se convierten en gemelos digitales

Lecturas adicionales

- INCOSE: https://www.incose.org/
- Apache Hadoop: https://es.wikipedia.org/wiki/Apache_Hadoop
- Arquitectura IIRA: https://www.iiconsortium.org/IIRA.htm
- Arquitectura micro servicios: https://microservices.io/patterns/microservices.html
- NETFLIX y micro servicios: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/
- Ejemplos de MS: http://eventuate.io/exampleapps.html
- NGINX: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/
- Industria 4.0: https://www.i-scoop.eu/industry-4-0/
- Eclipse BaSyX: https://www.eclipse.org/basyx/

Grado en Ingeniería Software

Tema 3
Estilos y Patrones de Diseño Software

Índice de la Presentación

- Introducción
- Estilos Arquitectónicos
- Patrones de Diseño
- Vistas Arquitectónicas
- Diseño con UML
- Ejemplos de Arquitecturas
- Conclusiones

Introducción

- Las arquitecturas software utilizan un conjunto de estilos arquitectónicos y patrones de diseño software para construirlas
- Son elementos de conocimiento reutilizable
- Las AS son heterogéneas, los estilos y patrones se suelen combinar
- Básicamente, la diferencia entre estilos y patrones reside en el tamaño (los estilos son mas grandes en términos de clases y subsistemas que los patrones)
- Los estilos son una de las primeras decisiones de diseño a considerar

Introducción

- Los estilos y los patrones determinan la forma de la arquitectura

Diferentes estilos sirven para distintos propósitos. Un chalet tiene una forma diferente que un bloque de pisos aunque los componentes que se utilicen para su construcción sean los mismos.

Algunos Estilos Arquitectónicos

- Tuberías y filtros (Pipes and filters)
- Estilo por capas (Layers)
- Cliente-servidor (Client-Server)
- Pizarra (blackboard)
- Modelo-Vista-Controlador (MVC)
- Máquina virtual
- Dirigido por eventos (Event-driven)
- Conexión a pares (Peer-to-Peer)
- Orientado a servicios (SOA)

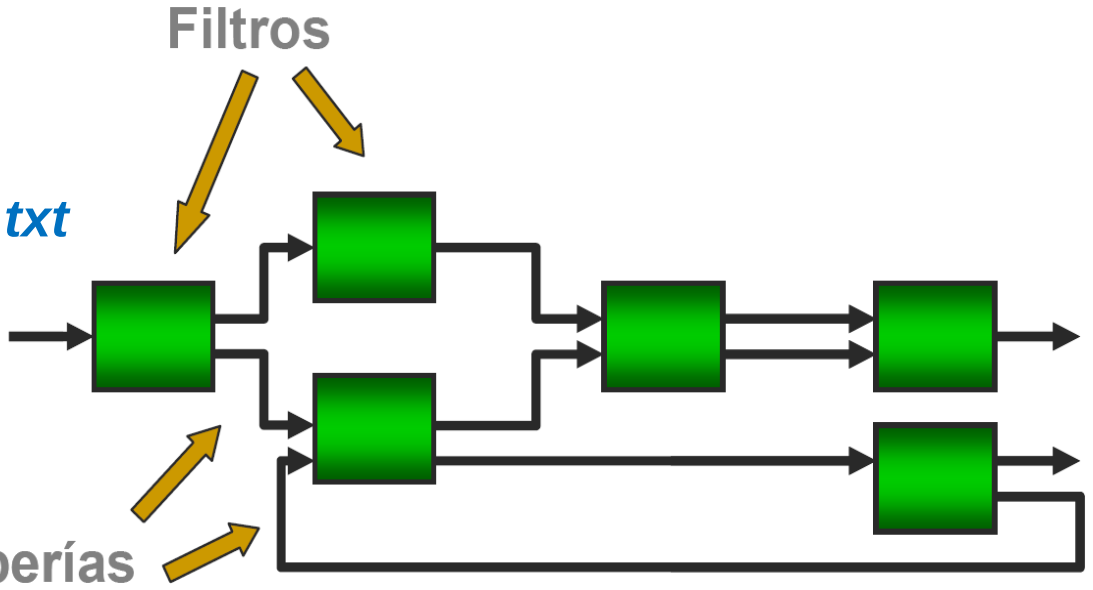
Describen la AS con una granularidad de sus componentes poco detallada.

Estilo Pipe and Filter

*Un **filtro** lee flujos de datos de sus entradas y genera flujos de datos a sus salidas.*

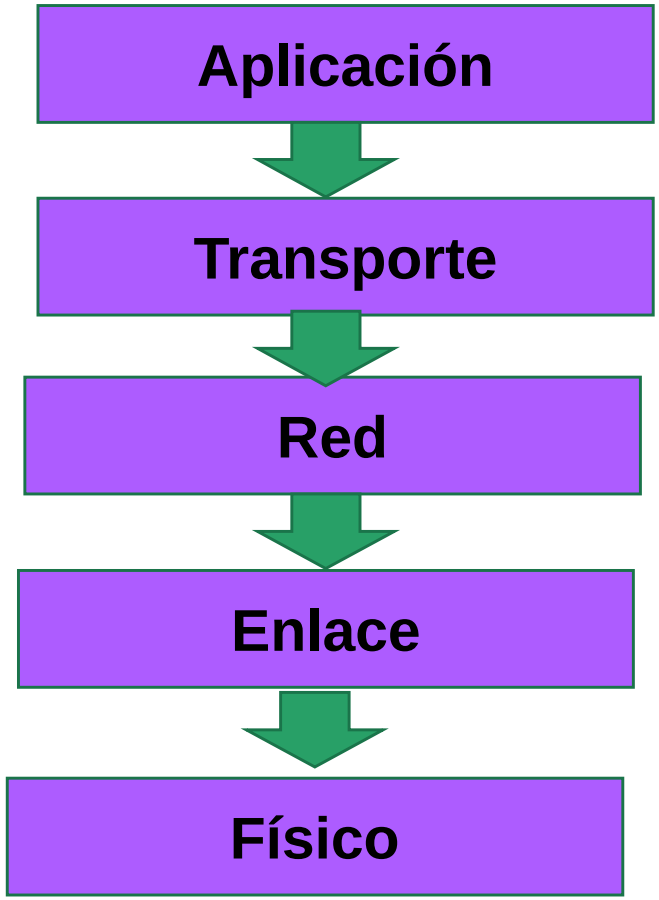
- *El **filtro** transforma los datos incrementalmente (se comienza a generar salida antes de acabar de leer toda la entrada).*
- *La **tubería** transmite el flujo de datos.*

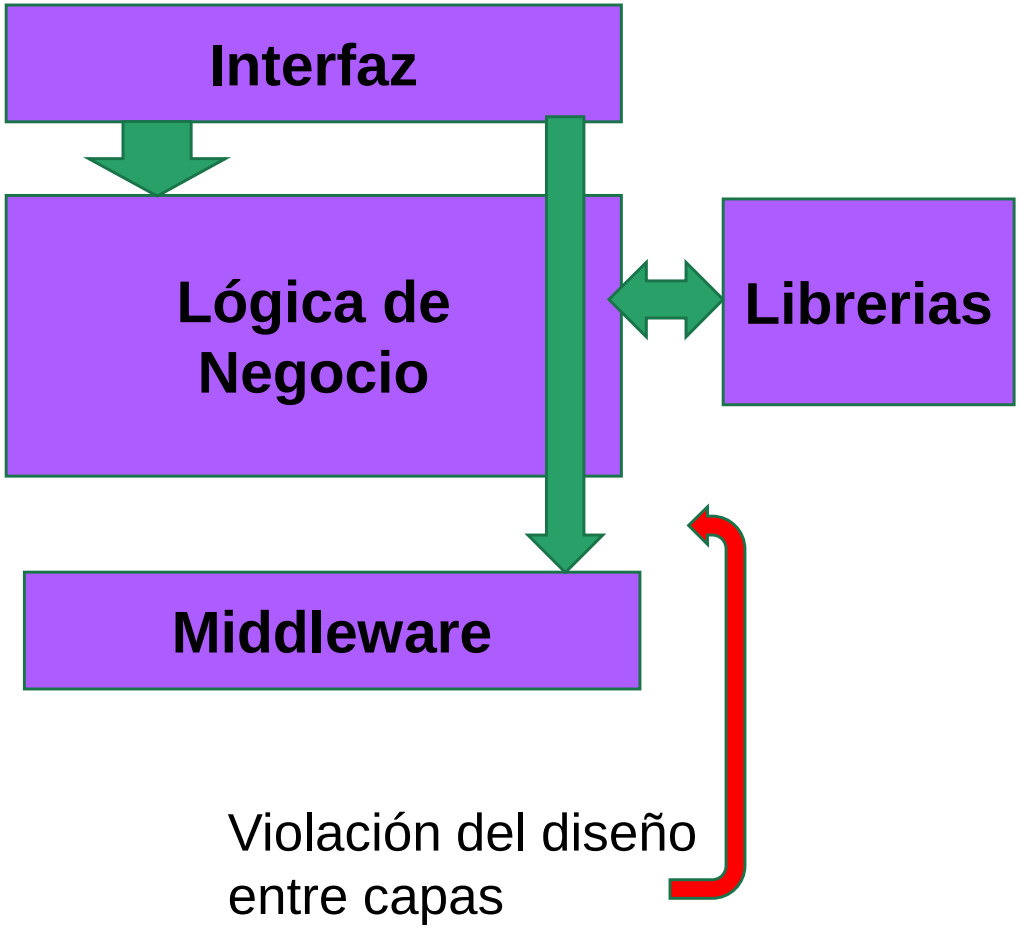
Comando Unix
ls –al | grep "fichero" > salida.txt



Estilos por Capas

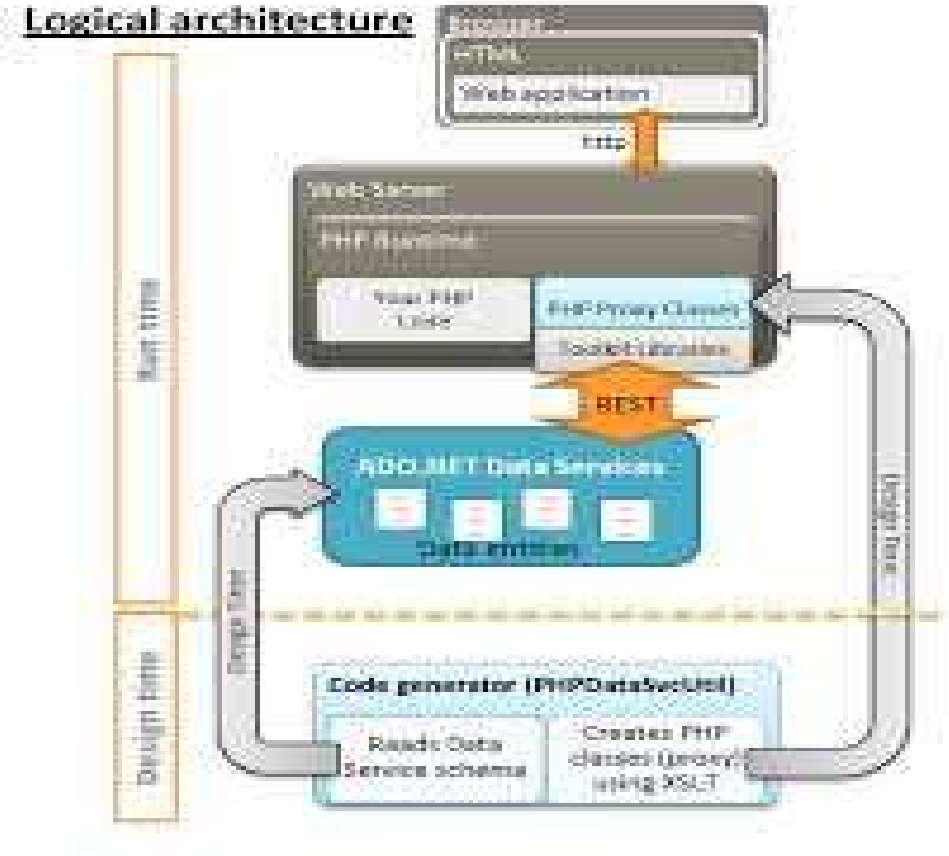
- Cada capa soporta un API (servicios) para que lo utilice la capa superior. A veces las capas no están claras y hay llamadas directas entre elementos de una capa (pero violan los principios de diseño). Podemos encontrar capas horizontalmente.

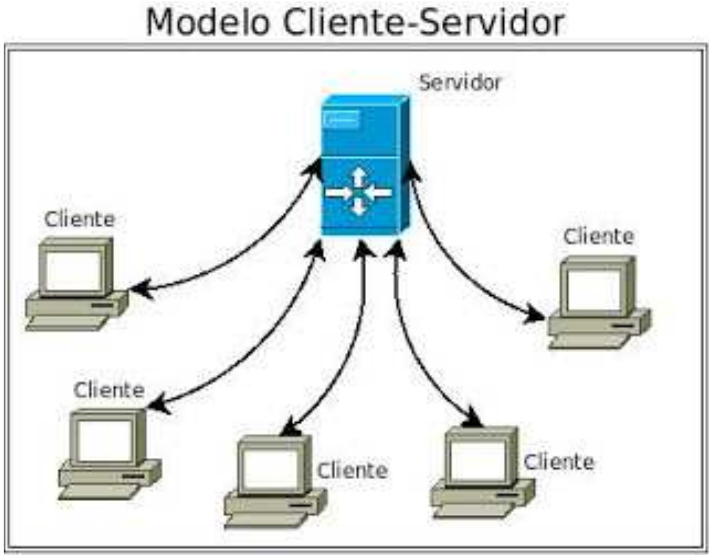




Estilo Cliente-Servidor

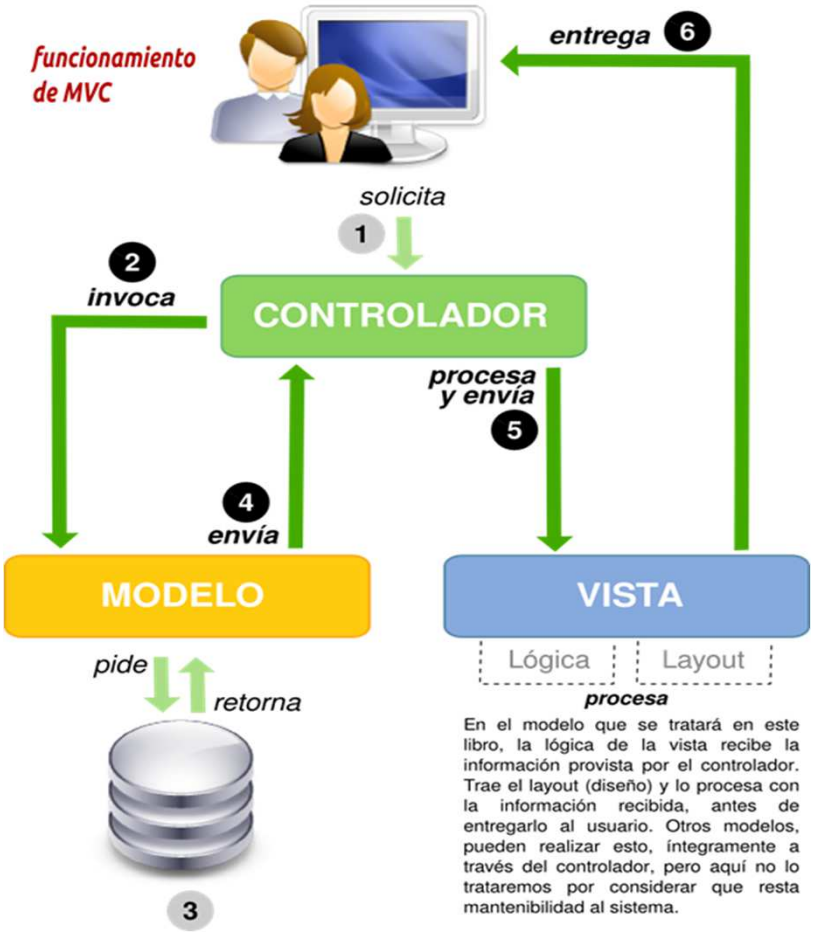
- Modelo utilizado en Internet
- Los clientes acceden a datos y aplicaciones en los servidores
- C/S distribuidos (redes peer-to-peer)
- Peticiones asíncronas y respuestas a eventos





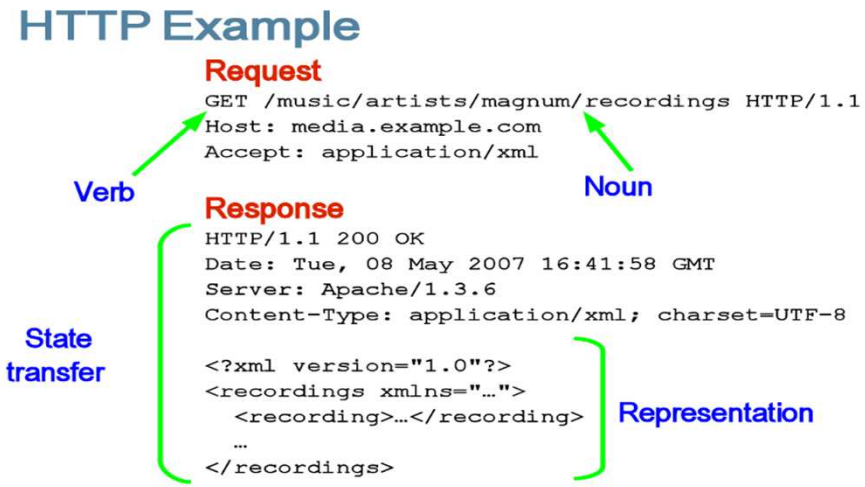
Estilos MVC

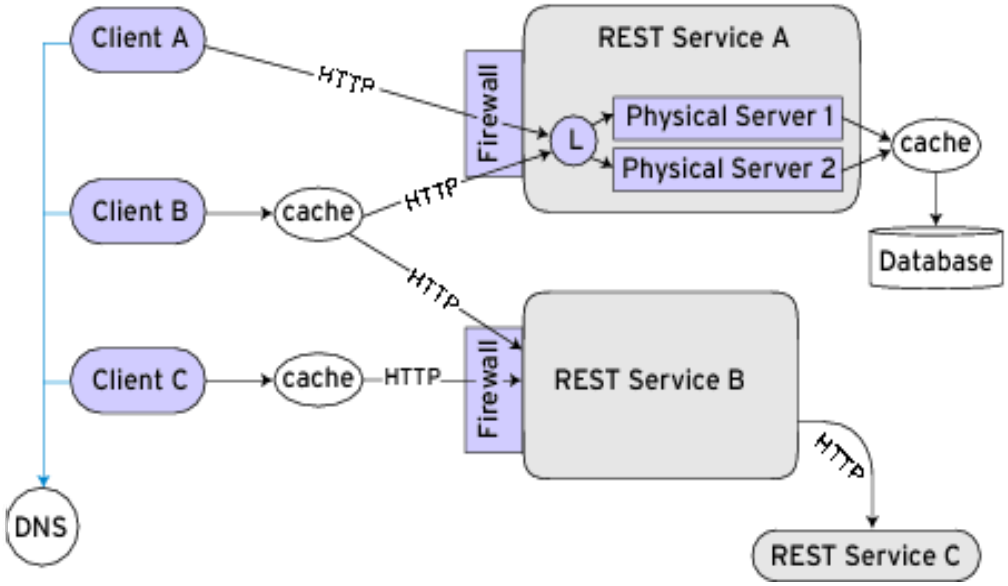
- Estilo para sistemas interactivos (desde años ‘80)
- Diferentes vistas de los mismos datos
- Posibilidad de interacción multimodal



Estilo REST

- REST or Restful (Representational State Transfer)
- Utilizado en para acceso a servicios Web y desarrollo Web
- Es una arquitectura cliente-servidor sin estado (stateless)
- Proporciona una interfaz uniforme
- Es cacheable





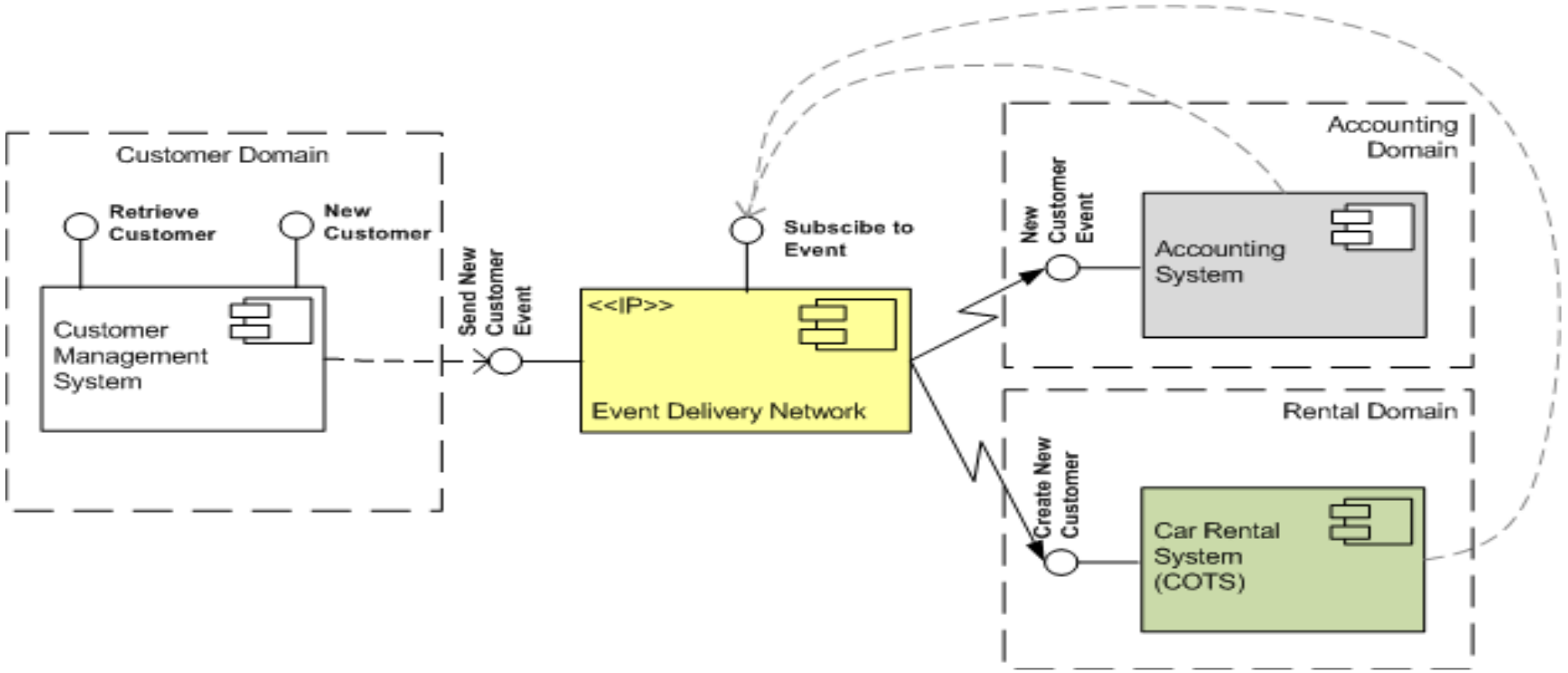
Estilo REST

Desde el punto de vista de la arquitectura de mejora los siguientes atributos de calidad:

- **Performance**: Las interacciones entre las páginas Web y servidores mejoran las prestaciones
- **Scalability**: para soportar un mayor de número de componentes e interacciones entre ellos
- **Simplicity**: de la interfaz mediante URIs
- **Modifiability**: de los componentes que cambian (incluso si la aplicación está en ejecución)
- **Reliability**: mejora de la resistencia a fallos
- **Portability**: cuando se mueven componentes de código con sus datos

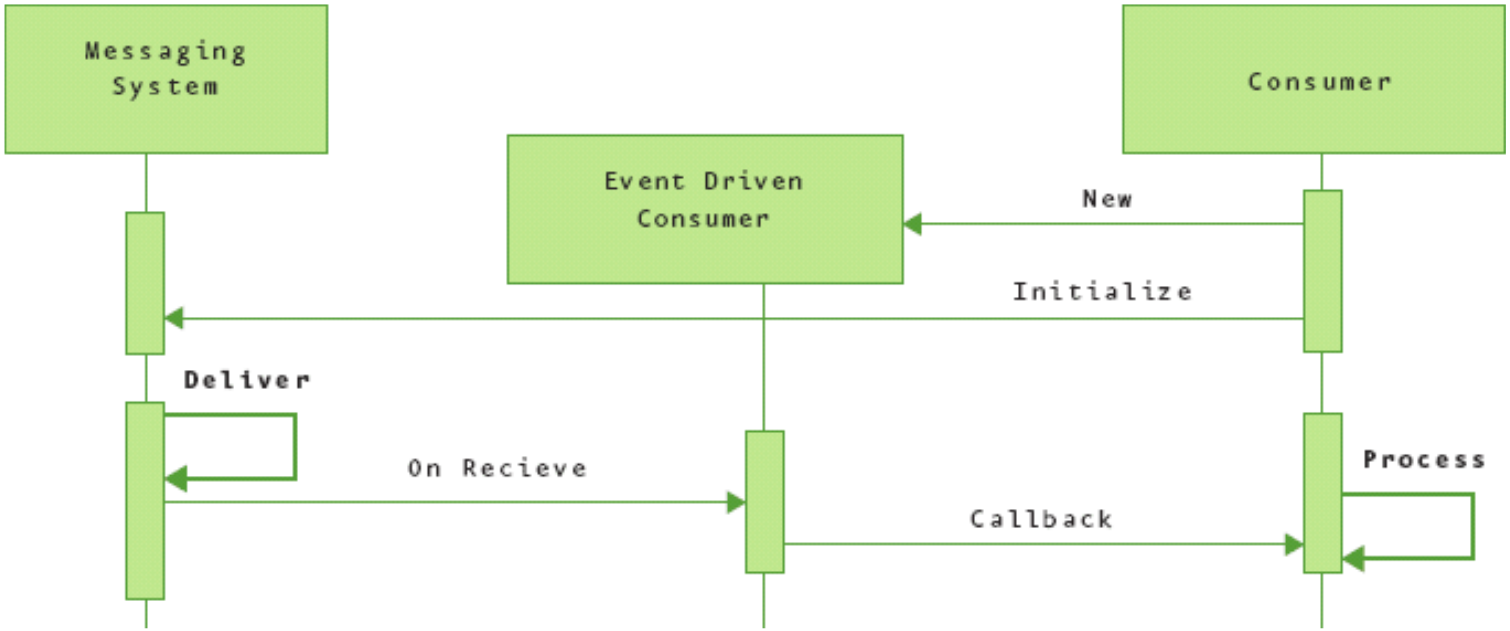
Estilos por Eventos

- Las arquitecturas dirigidas por eventos (EDA Event-driven Architectures) se basan en la detección, consumo y reacción a eventos
- Un evento representa un cambio significativo en el estado de un sistema
- Los eventos se transmiten entre sistema poco acoplados (“loosely coupled”) mediante mensajes



Estilos por Eventos

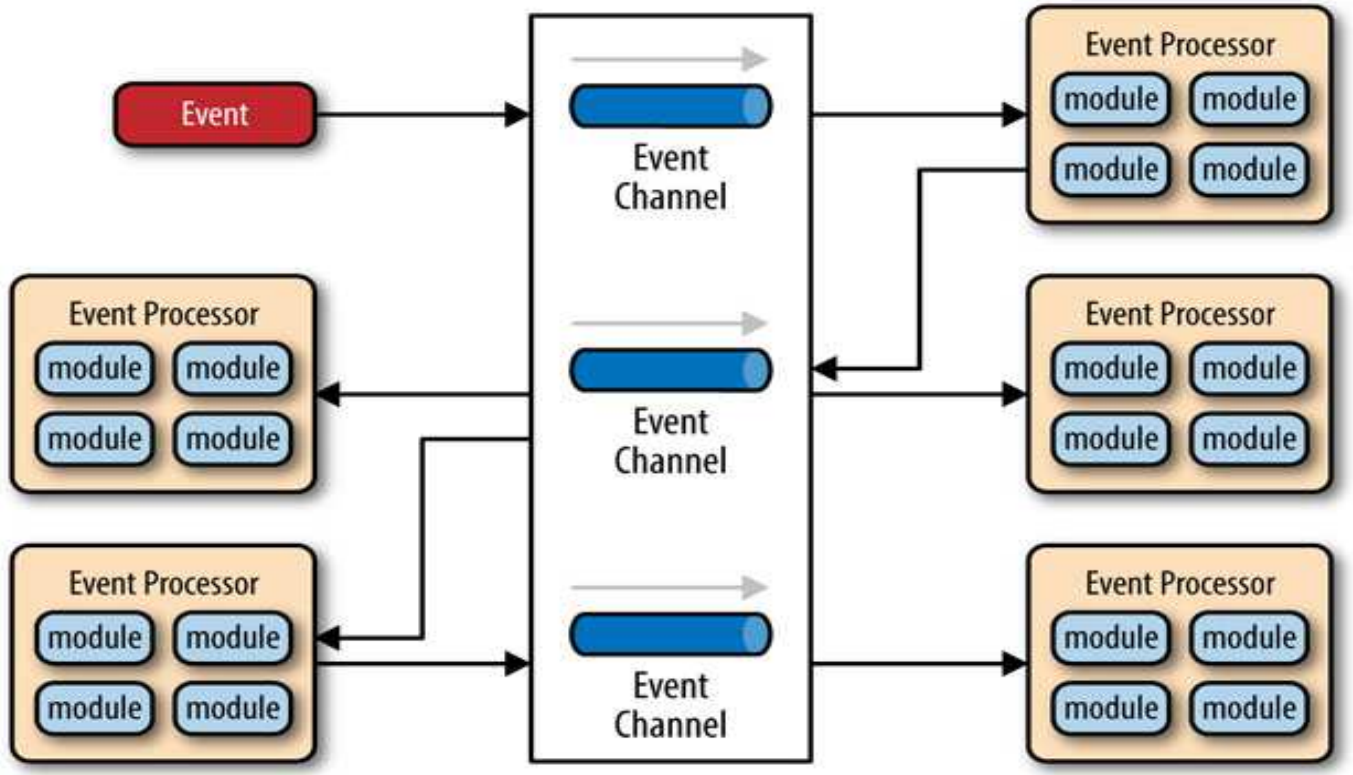
- Una arquitectura EDA tiene emisores de eventos, consumidores y canales para transmitir los eventos
- Los emisores no conocen a los consumidores de los eventos
- Los consumidores son responsables de reaccionar a los eventos
- Java Swing es un EDA (ActionListener, ActionEvent)
- Existe un modulo encargado de la gestión de los eventos
- Monitorizan los datos del entorno y analizan los eventos para proporcionar la respuesta más adecuada



Estilos por Eventos

Diferentes estilos o formas de procesar un evento:

- **Eventos simples**: relacionados con cambios medibles en una condición (e.g., un sensor que detecta temperatura)
- **Eventos en "stream":** son eventos que se envían en cadena a suscriptores
- **Eventos complejos**: representan confluencia de eventos y requieren intérpretes de eventos sofisticados



Patrones de Diseño

- La construcción de arquitecturas software se basan en los patrones de diseño y estilos arquitectónicos ya definidos.
- Un **Patrón de Diseño Software** (“design pattern”) es una solución de diseño para un problema recurrente en un contexto determinado.
- Son soluciones de diseño reutilizables ya probadas.

Patrones de Diseño

- Diferentes categorías de patrones (creacionales, comportamiento, estructura)
- Mas de 40 tipos de patrones de diseño
- Existen patrones específicos para soluciones empresariales (basados en Java)
- Los más recientes son los patrones para arquitecturas de micro servicios

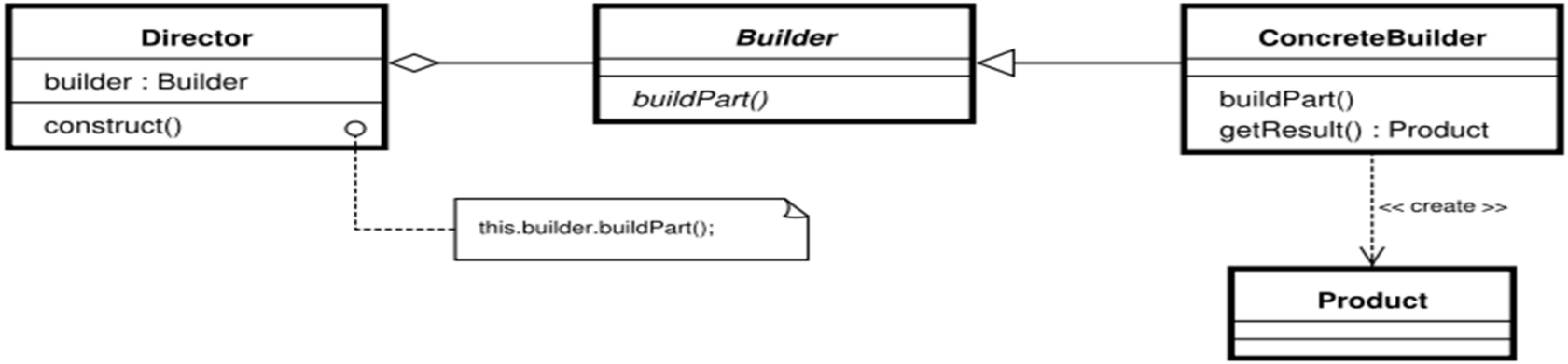
Patrones de Diseño Clásicos

- **Patrones creacionales**: patrones de diseño software que solucionan problemas de creación de instancias
- **Patrones estructurales**: solucionan problemas de composición (agregación) de clases y objetos
- **Patrones de comportamiento**: ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos

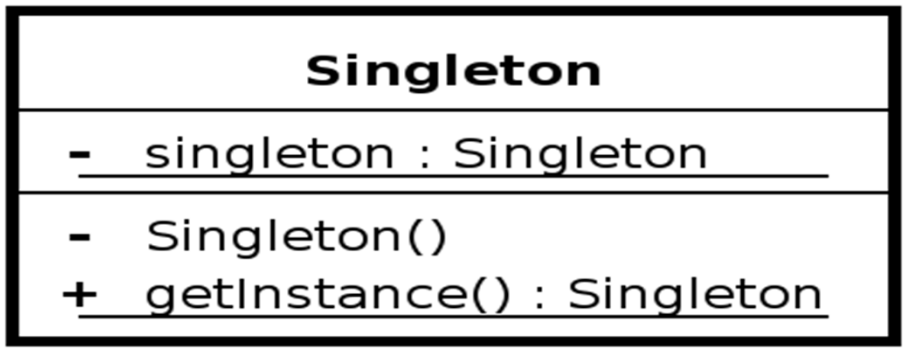
Esta es la clasificación clásica de patrones de diseño del libro de Gamma (GoF) aunque existen otras clasificaciones.

Patrones Creacionales

- **Builder:** abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto



- **Singleton:** garantiza la existencia de una única instancia para una clase. Permite restringir la creación de objetos a una clase

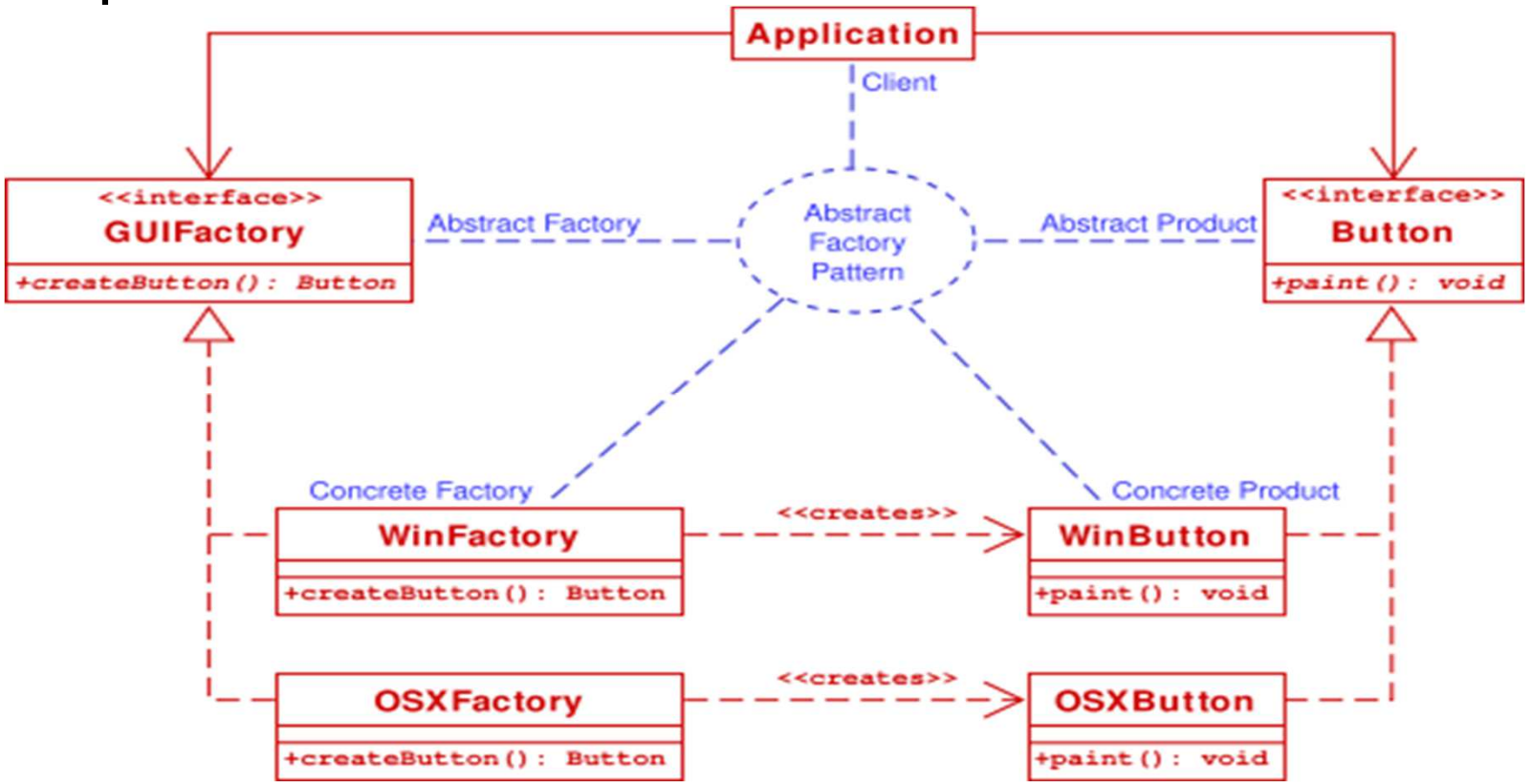


Patrones Creacionales

- **Abstract Factory:** permite trabajar con objetos de familias relacionadas y haciendo transparente el tipo de familia que se esté utilizando
- **Factory Method:** centraliza en una clase constructora la creación de objetos de un subtipo determinado, ocultando al usuario la diversidad de casos particulares que se pueden prever con el fin de elegir el subtipo que crear. Es una simplificación del *Abstract Factory*

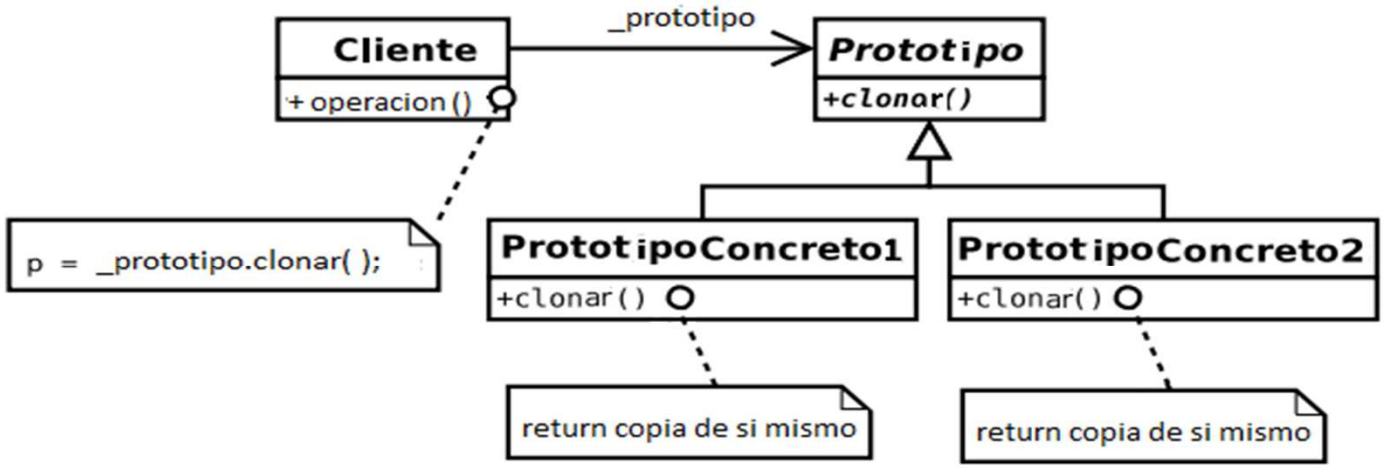
Patrón Abstract Factory

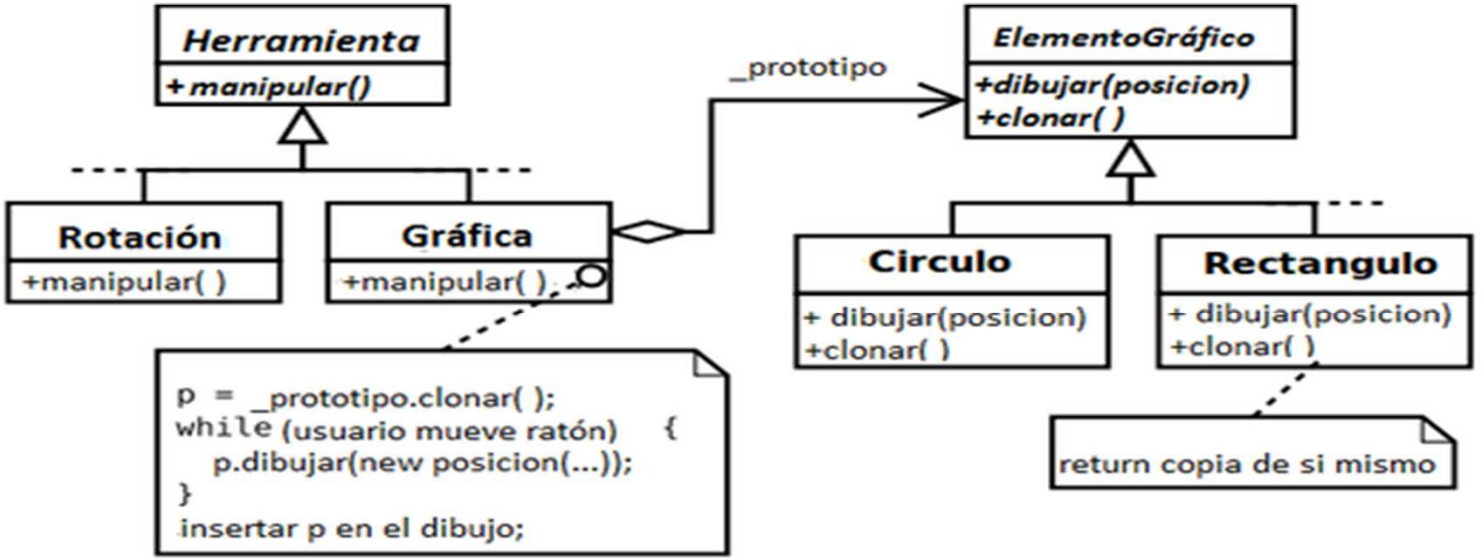
- **Propósito:** Encapsula un grupo de fábricas (“factories”) individuales bajo un tema común y utiliza interfaces genéricos para crear objetos concretos. Una factoria es una interfaz para crear objetos o familias de objetos sin especificar sus clases concretas



Patrón Prototype

Propósito: tiene como finalidad crear nuevos objetos duplicándolos



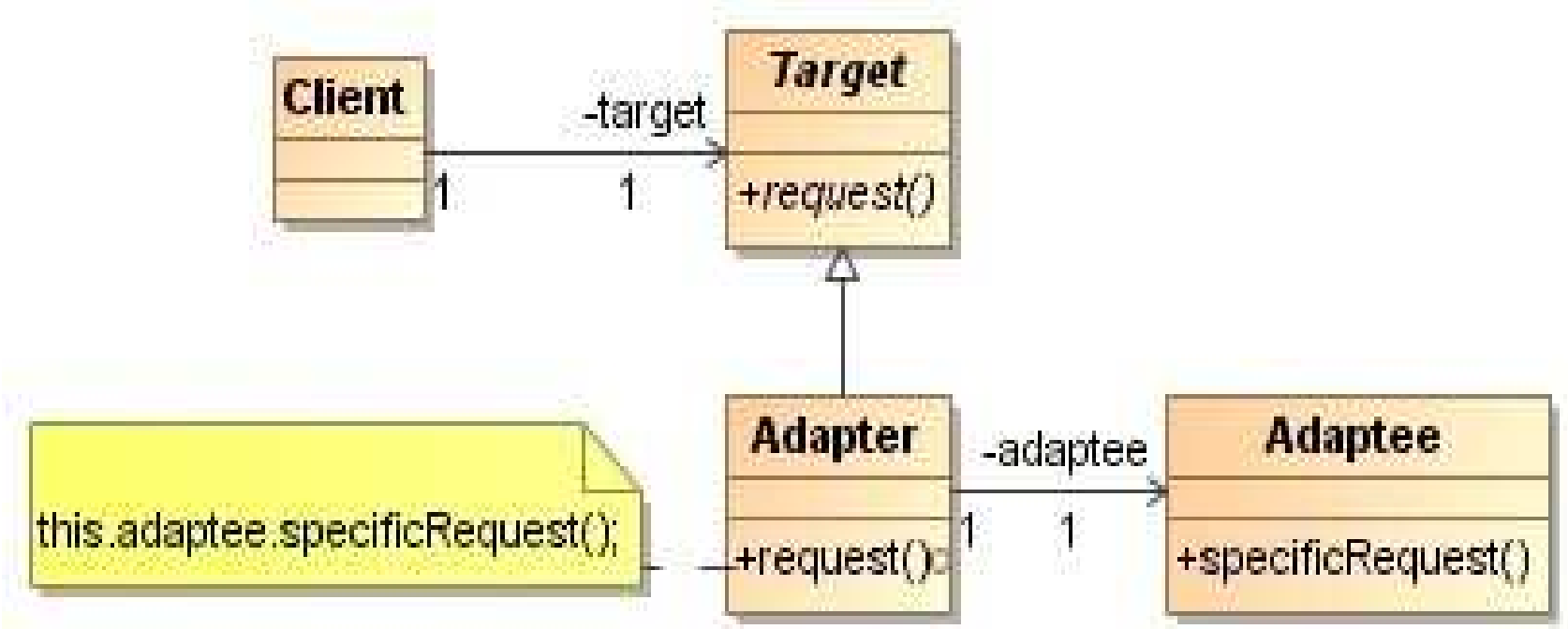


Patrones Estructurales

- **Adapter (wrapper):** Transforma una interfaz en otro. Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla
- **Decorator:** Añade funcionalidad a una clase dinámicamente
- **Facade:** Provee de una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema
- **Proxy, Composite, Bridge, etc.**

Patrón Adapter

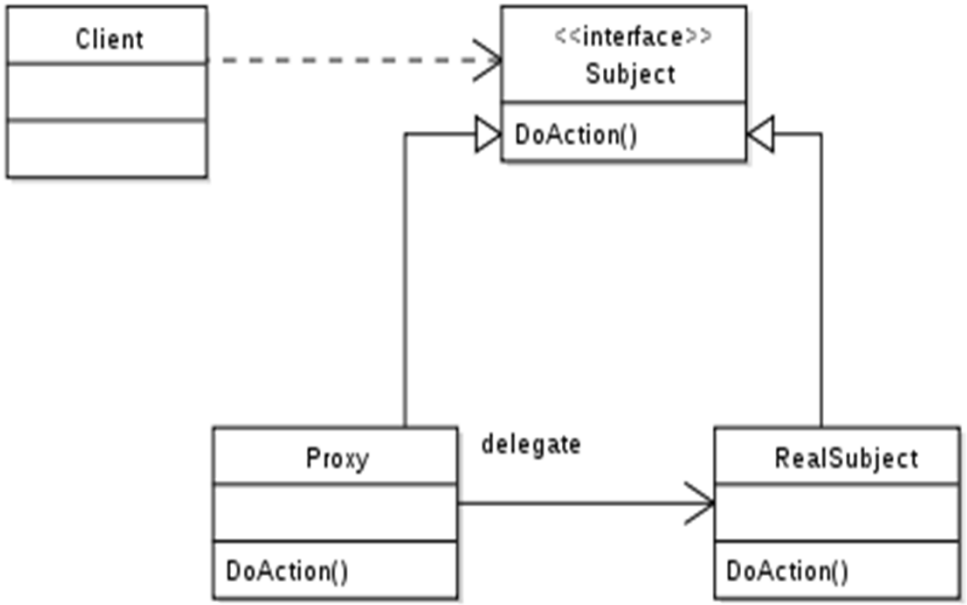
- **Target** define la interfaz específica del dominio que *Client* usa.
- **Client** colabora con la conformación de objetos para la interfaz *Target*.
- **Adaptee** define una interfaz existente que necesita adaptarse
- **Adapter** adapta la interfaz de *Adaptee* a la interfaz *Target*

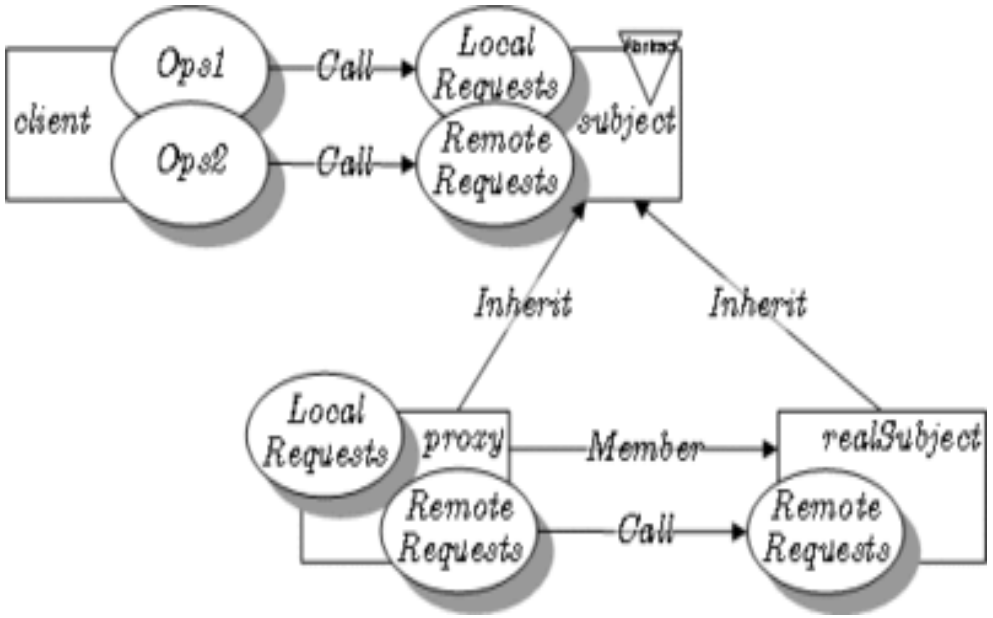


Tipicamente utilizado en programas antiguos para proporcionar un front-end moderno. Por ejemplo, wrappers para programas de sistemas bancarios escritos en COBOL se les proporciona un envoltorio o front-end Web que es el que accede a los datos

Patrón Proxy

Propósito: Proporciona un intermediario de un objeto para controlar su uso. Es una clase funcionado como una interfaz de otro elemento (ej: una conexión de red, un servidor)

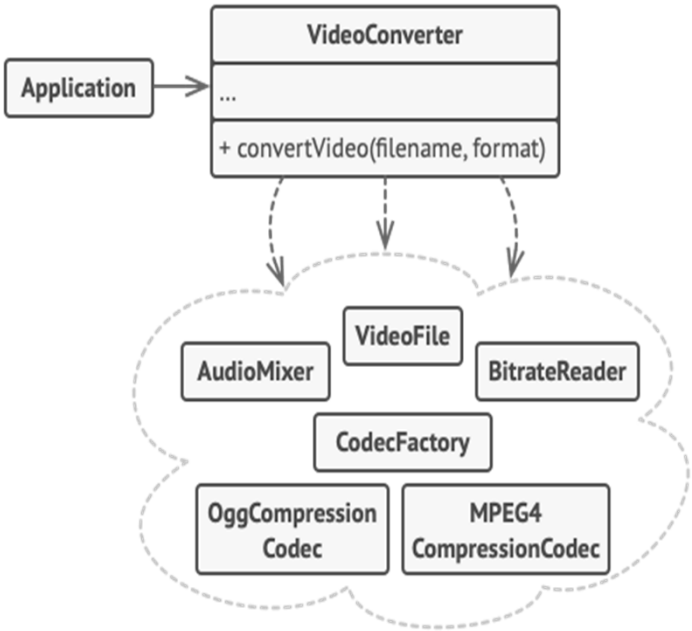


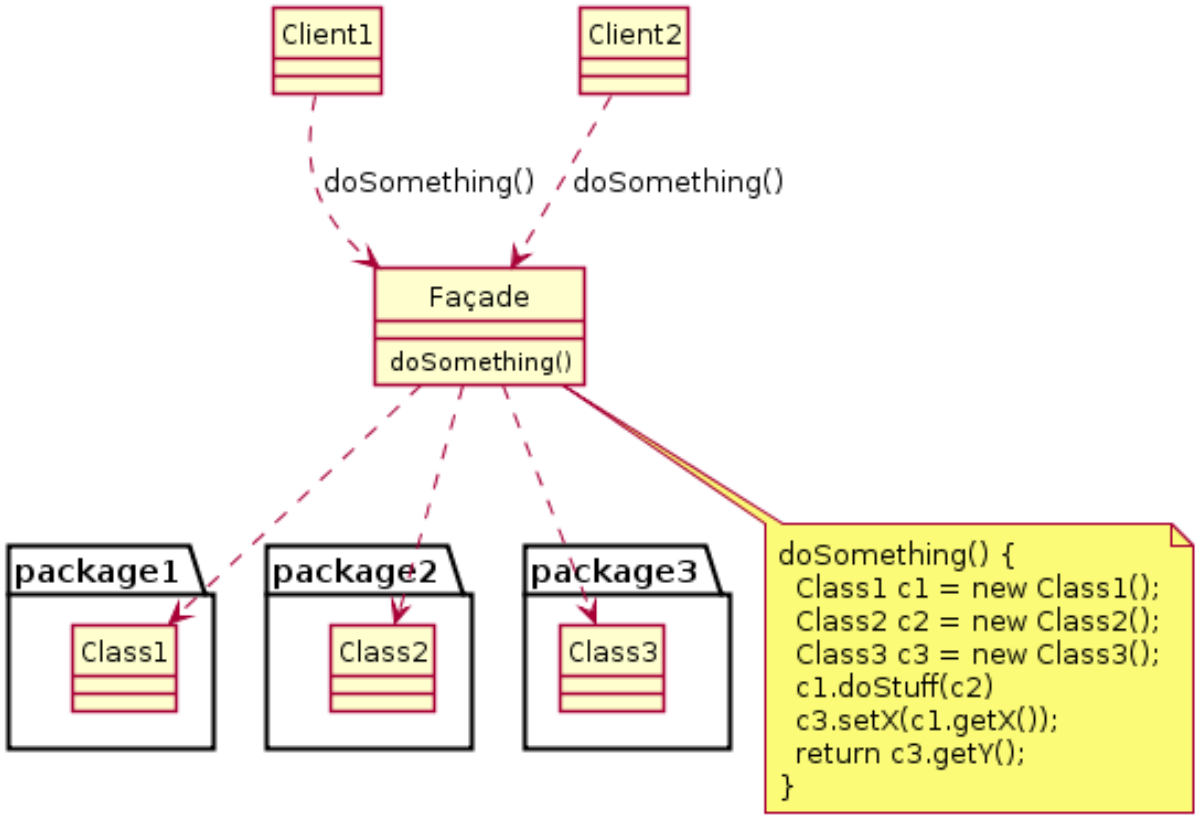


Muy habitual en sistemas Web para hacer la caché de las páginas más frecuentemente visitadas.

Patrón Facade

Propósito: Proporciona una interfaz simplificada a una librería, un framework o un conjunto de clases complejas. Es una alternativa al abstract factory y es similar al patron Mediator.



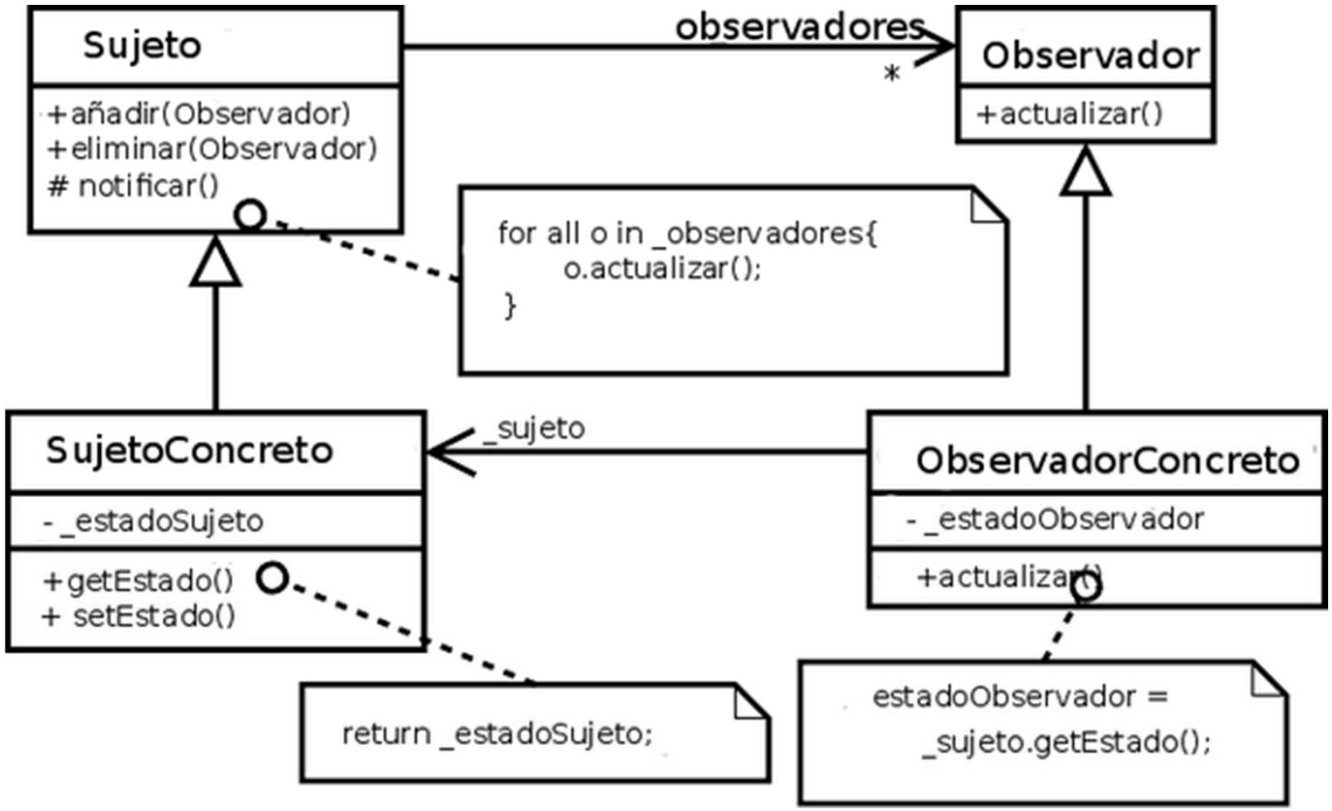


Patrones de Comportamiento

- **Chain of responsibility:** Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada
- **Strategy**: Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución
- **State**: Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno
- **Visitor**: Define nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera
- **Command, Interpreter, Iterator, Observer, Mediator, etc**

Patrón Observer

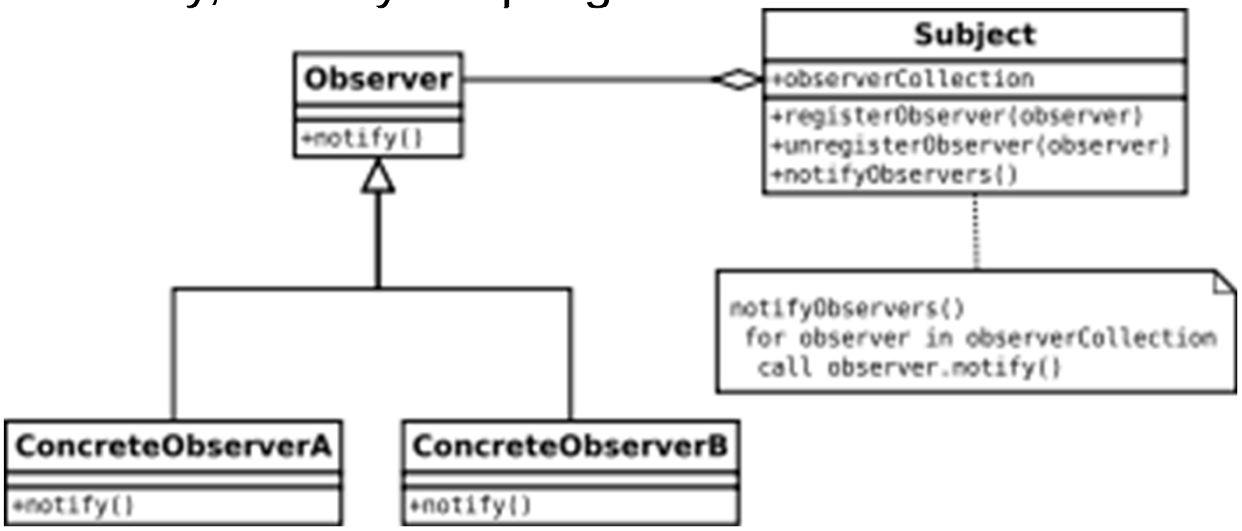
Propósito: Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él



Observer es un caso particular del patrón publish/subscribe

Patrón Publish-Subscribe

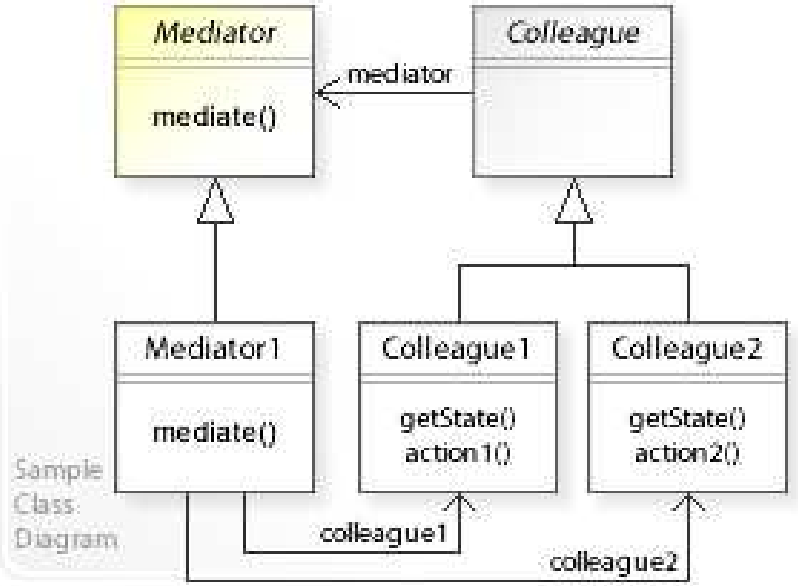
- **Propósito**: remitentes envían mensajes no programados a suscriptores. Este patrón es parte del middleware orientado a mensajes (MOM). El patrón Observer es un caso particular en el que un objeto denominado "Subject" mantiene una lista de dependientes denominados Observers, que notifican automáticamente cambios de estado. Se utiliza en sistemas de eventos distribuidos
- **Ventajas**: Scalability, loosely coupling

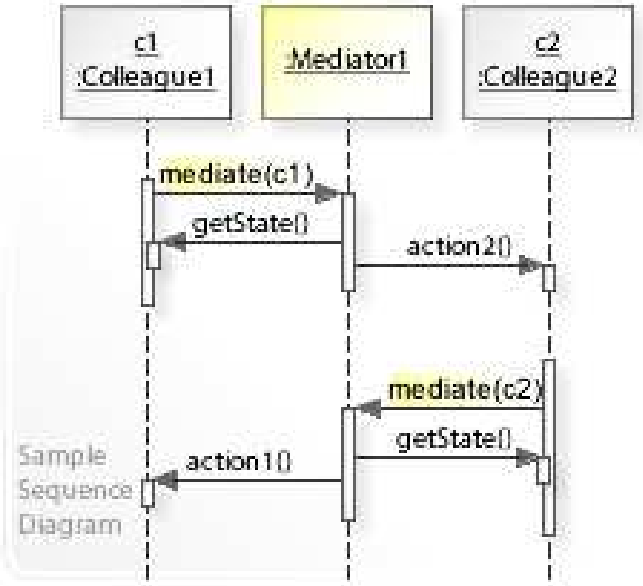


Este patrón atraviesa las redes mientras que observer monitoriza el estado interno de los objetos dentro de un sistema. Se utiliza por ejemplo para suscribirse a noticias o actualizaciones de un periódico digital

Patrón Mediator

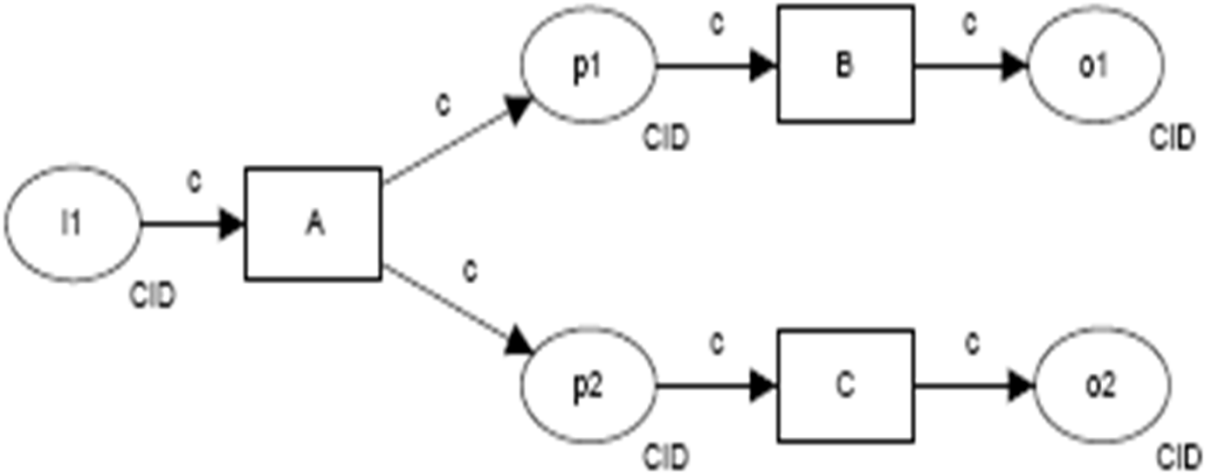
- **Propósito**: encapsula la comunicación entre objetos a través de un objeto denominado “mediator”.
- **Ventajas**: Reduce el acoplamiento entre componentes haciendo que se comuniquen indirectamente a través del elemento “mediator”. Es útil durante tareas de mantenimiento y refactorización del código ya que reduce las dependencias en la comunicación entre objetos.





Otros Patrones: Parallel Split

- **Propósito**: Es un cierto punto del flujo donde un thread individual se divide en varios threads que pueden ejecutarse en paralelo permitiendo la ejecución simultánea de varias actividades. Uso potencial en orquestración de servicios Web (BPEL). Es un patrón básico de control de flujo



Muy utilizado en reservas de viajes u hoteles vía Web para hacer consultas paralelas a diversas bases de datos

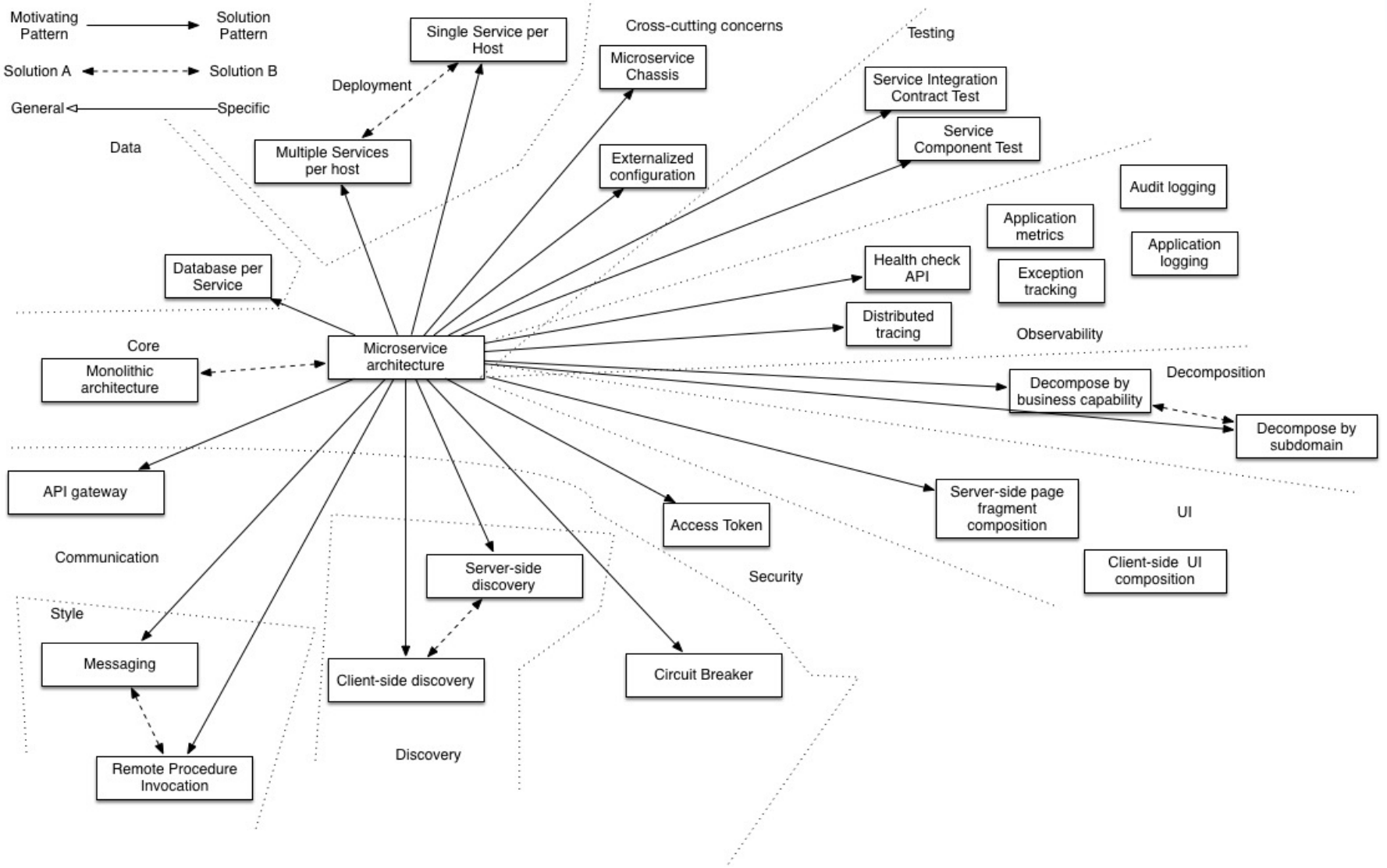
Otros Patrones: Inversion of Control

Origen: Basado en el principio de Hollywood donde se dan respuestas típicas a los actores.

Propósito: El flujo de ejecución tradicional se invierte de manera que se especifican respuestas deseadas a solicitudes concretas, en lugar de que el programador especifique la secuencia de acciones. Popularizado con el uso de frameworks, como por ejemplo el Framework de Spring. En Java hay 6 formas de implementar IoC

Ejemplo: uso un programa para hacer preguntas pero en un sistema de ventanas le cedo el control para que el actúe y me de por ejemplo el nombre de un programa.

Otros Patrones: Arquitecturas de Micro-servicios (MSA)



Otros Patrones: Micro Servicios

Patrones de descomposición: (i) por capacidades de negocio, (ii) por subdominio

Patrón de base de datos por servicio

Patrón API Gateway

Patrones de descubrimiento de cliente y de servidor

Patrón de invocación remota de mensajes

Patrón Circuit Breaker

Patrón Access Token

Patrones de observación

Patrones de interfaz de usuario

Existen distintas clasificaciones o taxonomías de patrones.
Esta es una de ellas.

Arquitecturas y Diseño UML

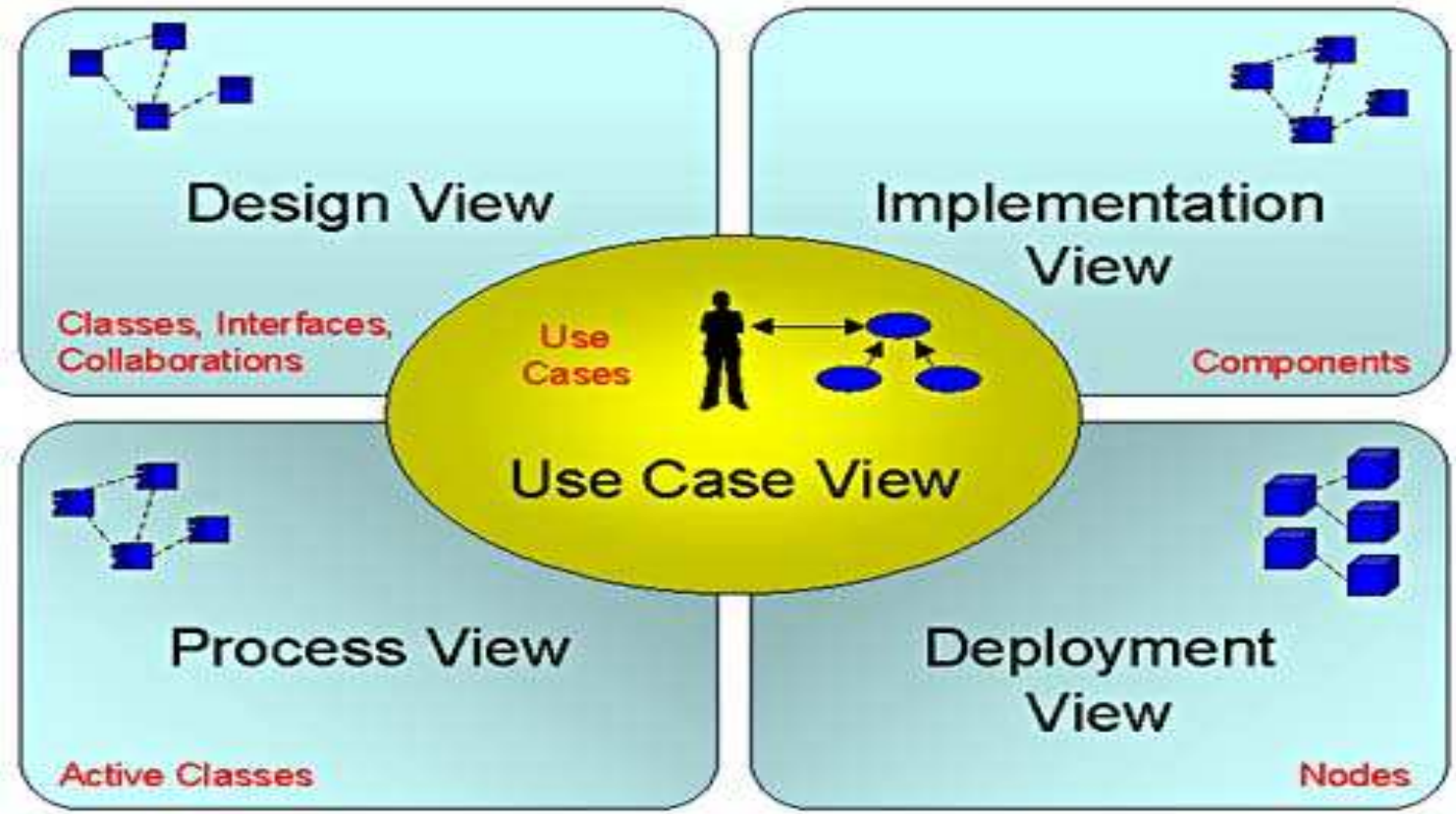
- Vistas y Documentación:
  - Las arquitecturas deben documentarse como un conjunto de vistas arquitectónicas.
  - Utilizar solamente aquellas vistas necesarias o de interés para los “stakeholders”.
  - Uso de UML 2.x con herramientas de modelado UML. **NO Word ni Visio si utilizas UML!!!**
  - Decidir de antemano que diagramas son necesarios para describir la arquitectura de nuestro sistema (complejo).

Arquitecturas y Diseño UML

- Más de 9 tipos de diagramas UML para representar las distintas vistas arquitectónicas.
- **No es necesario utilizar todos los diagramas UML para describir una arquitectura software**.
- Existen perfiles UML como RM-ODP o SysML para sistemas distribuidos o para describir la ingeniería de sistemas. Un perfil de UML se instala como un plug-in de una herramienta y proporciona una librería gráfica de símbolos
- Uso de frameworks propios (TOGAF, Zachman, DODAF, MODAF, RM-ODP, GERAM, etc.) para describir arquitecturas de sistemas distribuidos, empresariales o militares.

Arquitecturas y Diseño UML

Modelo de vistas “4+1” (Kruchten, 1995)



Fue el primer modelo de vistas arquitectónicas propuesto y que luego se ha utilizado en todas las herramientas UML para tener diferentes tipos de diagramas. Una vista representa la perspectiva de un usuario en la arquitectura.

Vistas Arquitectónicas

- Permiten describir el sistema y la arquitectura desde diferentes puntos de vista o perspectiva
- Cada vista representa los intereses de un grupo de usuarios
- Existen diferentes modelos de vistas (Kruchten, Soni, Woods & Rozanski)
- Cada vista se representa mediante un conjunto de diagramas UML

Vistas Arquitectónicas

- **Vista lógica**: captura las entidades lógicas (software) de un sistema y cómo se interconectan.
- **Vista física**: captura las entidades físicas (hardware) de un sistema y sus interrelaciones.
- **Vista de despliegue**: captura cómo las entidades lógicas se reparten en entidades físicas.
- **Vista de comportamiento**: captura el comportamiento esperado del sistema o partes de éste.
- **Vista de casos de uso**: captura los requisitos funcionales que serán ofrecidos por el sistema de información.

Diseño con UML

- Es una notación que incluye diferentes tipos de diagramas para modelar y diseñar sistemas software.
- Permite modelar vistas arquitectónicas a través delos diferentes diagramas UML
- Captura la información acerca de la estructura estática y el comportamiento dinámico de un sistema
- Dispone de un lenguaje de restricciones (OCL) para especificar pre-condiciones y post-condiciones
- Es generalista pero para algunos dominios dispone de “perfiles” específicos (e.g., SysML, MARTE)

Para describir la arquitectura software de un sistema no es necesario utilizar todos los diagramas UML. Solamente aquellos que sean necesarios. Para ello es necesario conocer en que vistas arquitectónicas estamos interesados. Por ejemplo, al cliente final solo le interesaría la vista de casos de uso mientras que al encargado de desplegar los servidores solo le interesa la vista de despliegue.

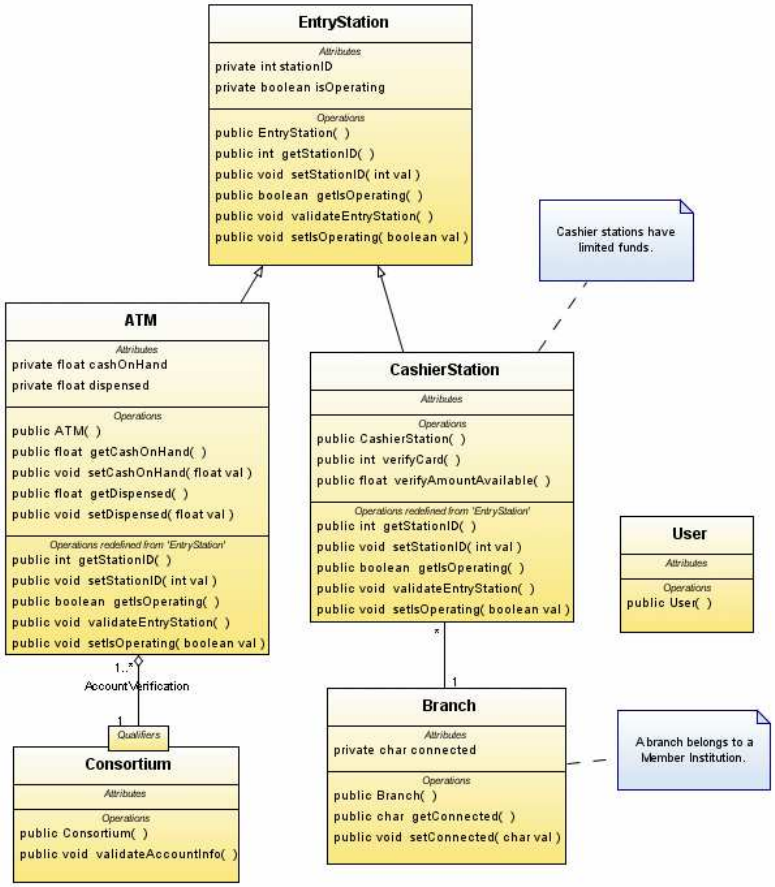
Vistas y Diagramas con UML

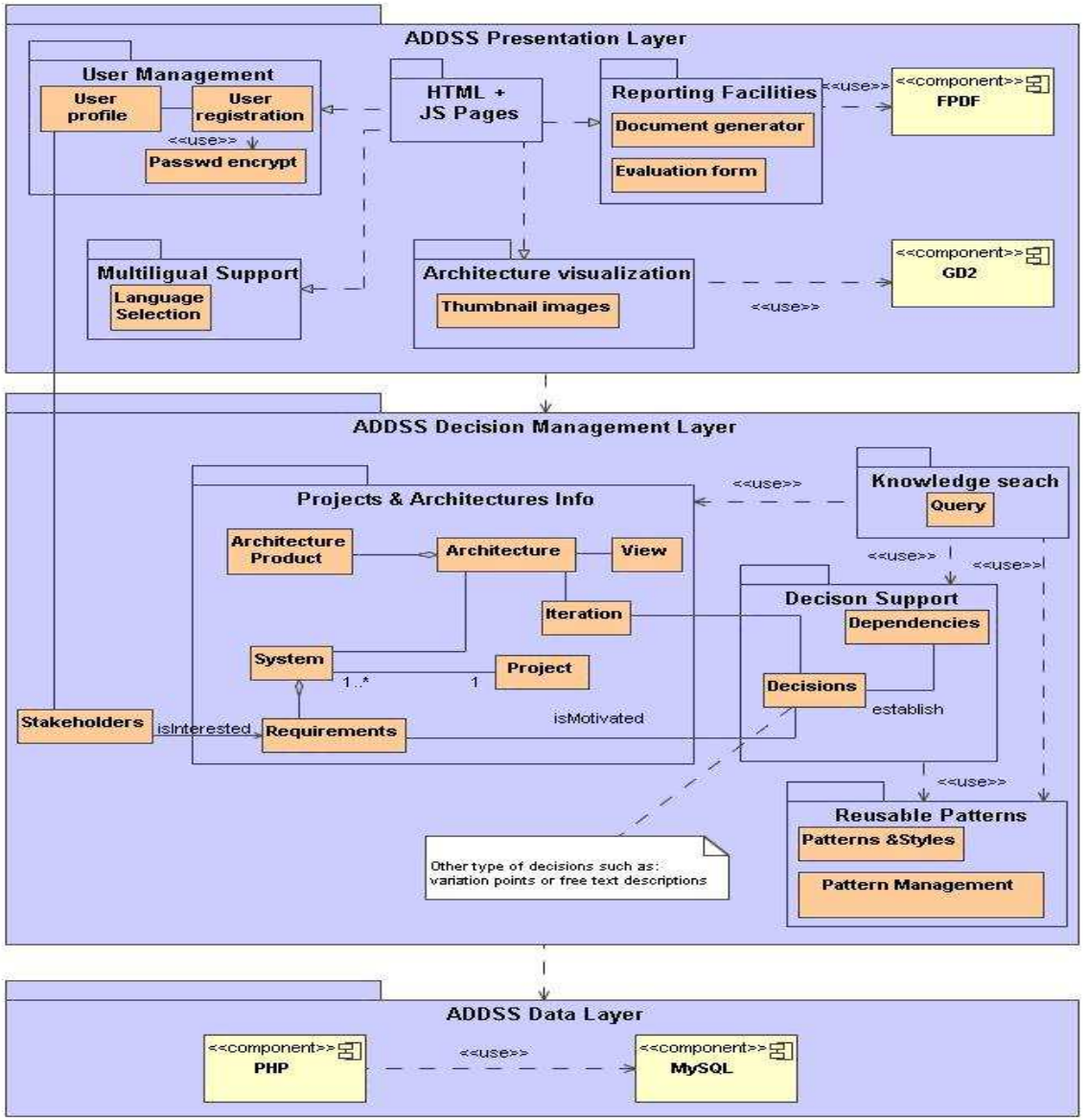
- **Vista Estática (lógica)**
  - Diagrama de clases, diagrama de objetos
- **Vista de Casos de uso**
  - Diagrama de casos de uso
- **Vista de Despliegue**
  - Diagrama de despliegue
- **Vista de Proceso**
  - Diagrama de secuencias, diagramas de estado
- **Vista de Implementación**
  - Diagrama de componentes

Arquitecturas y Diseño UML

Ejemplo de vista Estática

Diagrama clases y paquetes

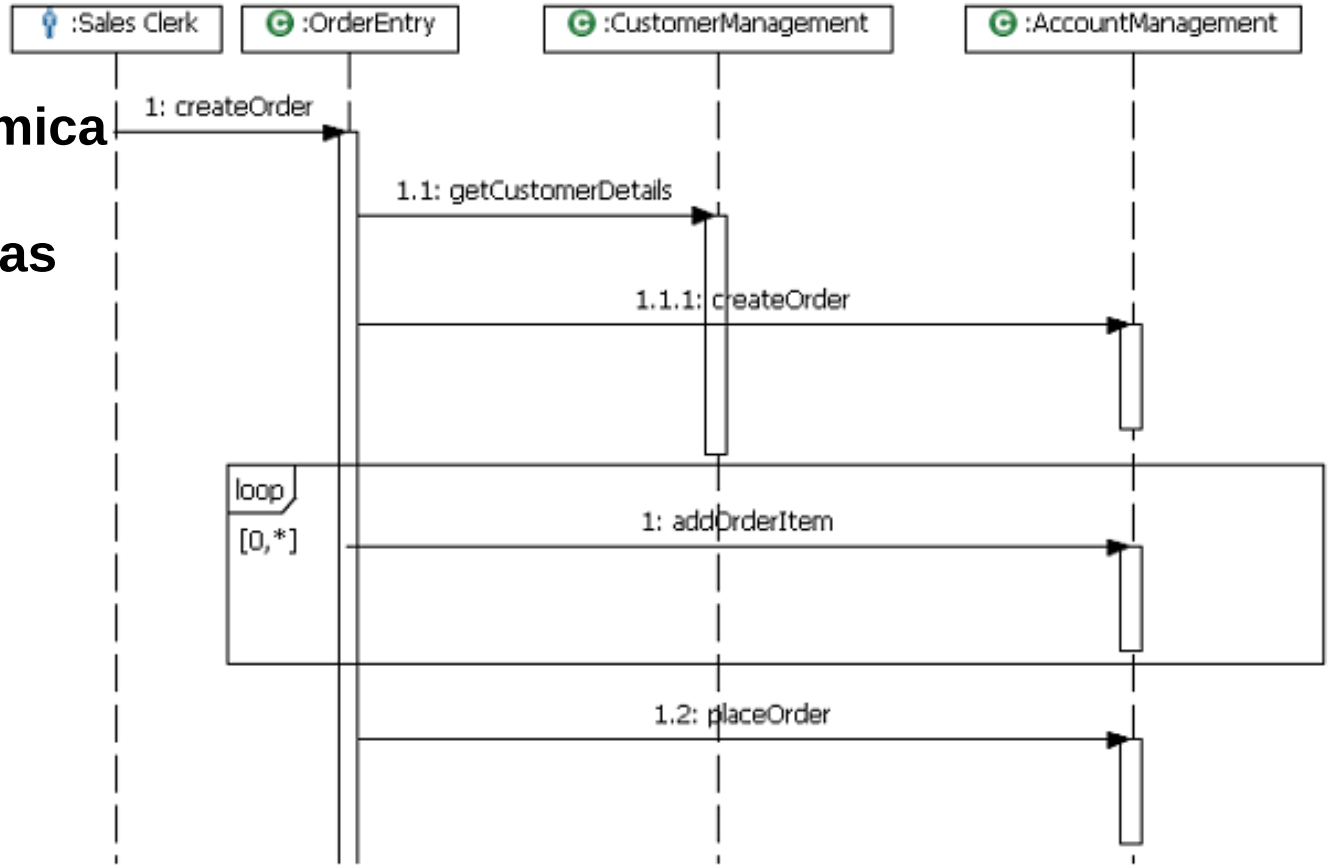




Arquitecturas y Diseño UML

Ejemplo de vista Dinámica

Diagrama de secuencias



Vistas y Diagramas con UML

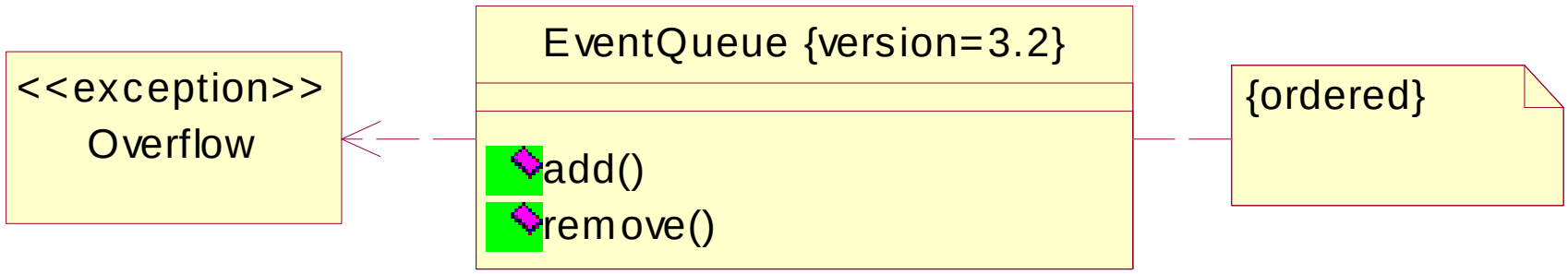
Area	Vista	Diagramas	Principales Conceptos
Estructural	Vista Estática	Diagrama de Clases	Clase, asociación, generalización, dependencia, realización, interface.
	Vista de Casos de Uso	Diagrama de Casos de Uso	Caso de uso, actor, asociación, extención, inclusión, generalización.
	Vista de Implementación	Diagrama de Componentes	Componente, interface, dependencia, realización.
	Vista de Despliegue	Diagrama de Despliegue	Nodo, componente, dependencia, localización.
Dinámica	Vista de Máquina de Estados	Diagramas de Estados (ciclo de vida de objeto)	Estado, evento, transición, acción.
	Vista de Actividades	Diagrama de Actividades	Estado, actividad, transición, concurrencia (fork), reunión (join).
	Vista de Interacción	Diagrama de Secuencia	Interacción, objeto, mensaje, activación.
		Diagrama de Colaboración	Colaboración, interacción, rol de colaboración, mensaje.
Gestión del Modelo	Vista de Gestión del Modelo	Diagrama de clases	Paquete, subsistema, modelo.
Extensibilidad	Todos	Todos	Restricciones, estereotipos, valores etiquetados.

UML: Comportamiento Dinámico

- Vista de Interacción: modela como interactúan los objetos para realizar una funcionalidad del sistema
  - Diagrama de secuencias
- Vista de Máquina de estados: modela el ciclo de vida de una instancia de una clase en estados y transiciones.
  - Diagrama de transición de estados
- Vista de Actividades: modela flujos de trabajo (workflows)
  - Diagrama de Actividades

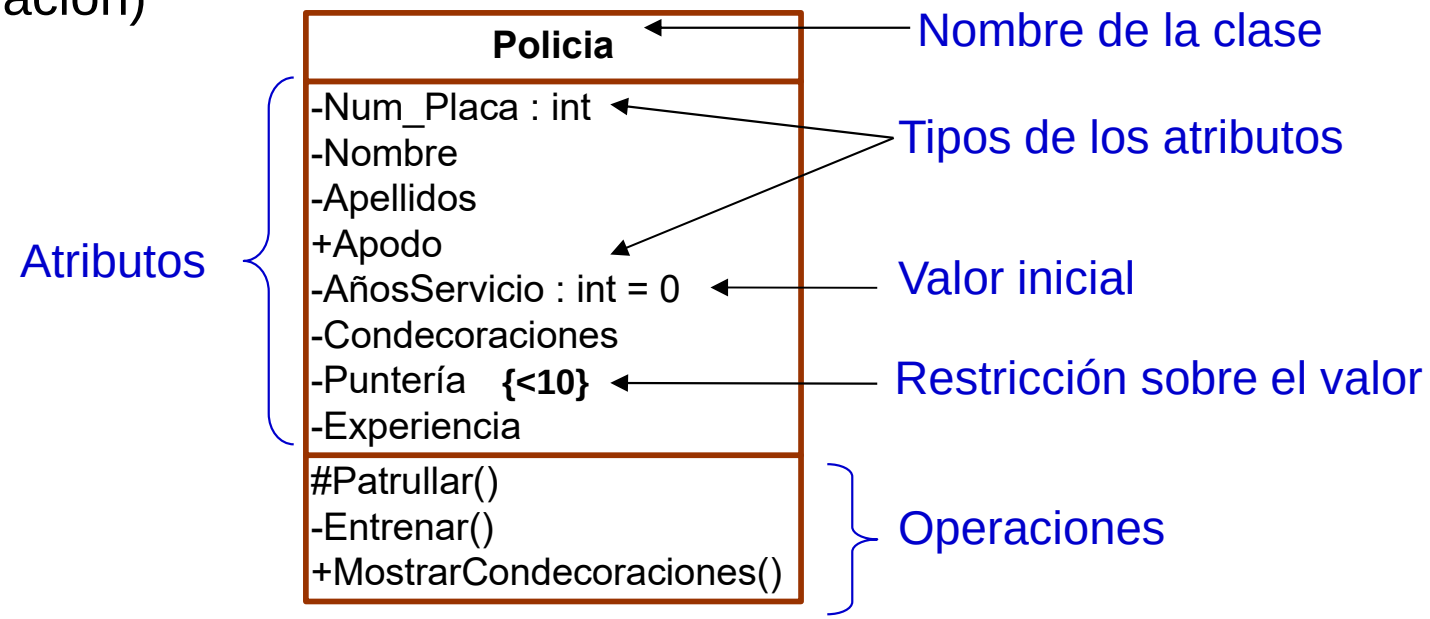
UML: Extensiones

- **Estereotipos**:
  - Extienden la semántica del elemento sobre el que se aplica
  - Permite representar una variación de un elemento existente que posee otra intención, o distinción de uso
  - Se expresan generalmente (entre << y >>)
- **Valores etiquetados**
  - Extiende las propiedades de un elemento de UML, permitiendo añadir nueva información en la especificación del elemento
  - Cadenas con el nombre de la etiqueta, un signo igual y un valor
- **Restricciones**
  - Notación formal con OCL



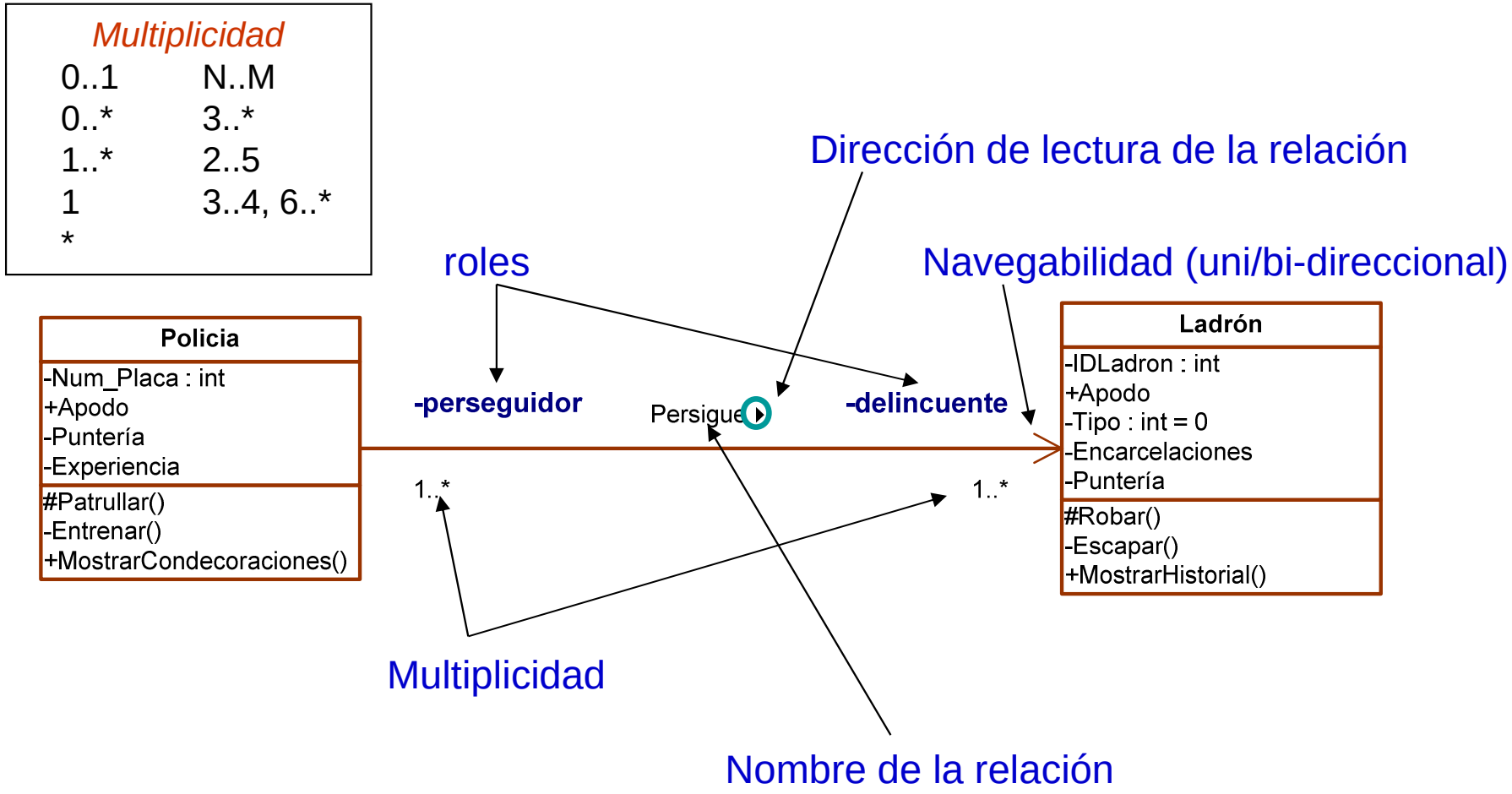
UML: Diagrama de Clases

- Representan la vista estática del sistema y describen conceptos (generalmente) del mundo real
- Nombre clase (entidad), atributos, métodos
  - **(-) Privado**: Los atributos/operaciones son visibles solo desde la propia clase.
  - **(#)** Los atributos/operaciones **protegidos** están visibles para la propia clase y para las clases derivadas de la original
  - **(+)** Los atributos/operaciones **públicos** son visibles a otras clases (cuando se trata de atributos se está transgrediendo el principio de encapsulación)



UML: Relación de Asociación

- Conexión semántica entre instancias de clases
- Proporciona una conexión para el envío de mensajes



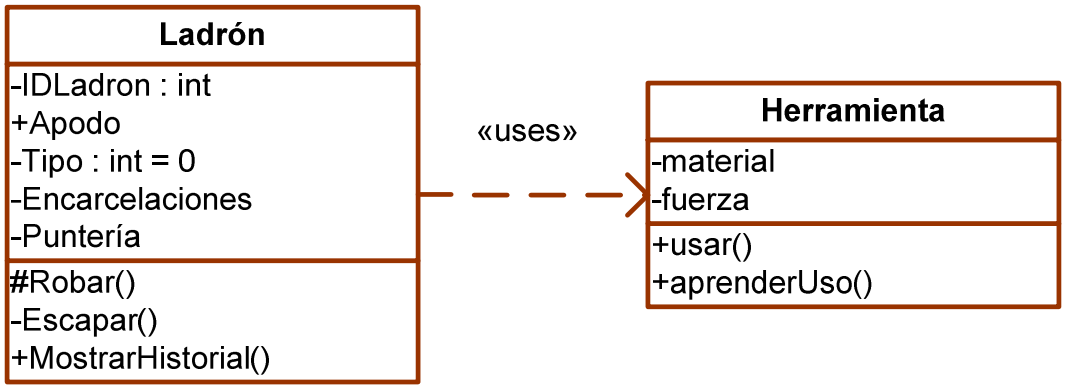
UML: Relación de Agregación

- Lógica (agregación o composición débil): la parte puede pertenecer a varios agregados

- Física (o composición fuerte): las partes sólo existen asociadas al compuesto (acceso a través de él)

UML: Relación de Dependencia

- Indica una relación semántica entre dos o más elementos del modelo en la cual un cambio al elemento proveedor puede requerir un cambio

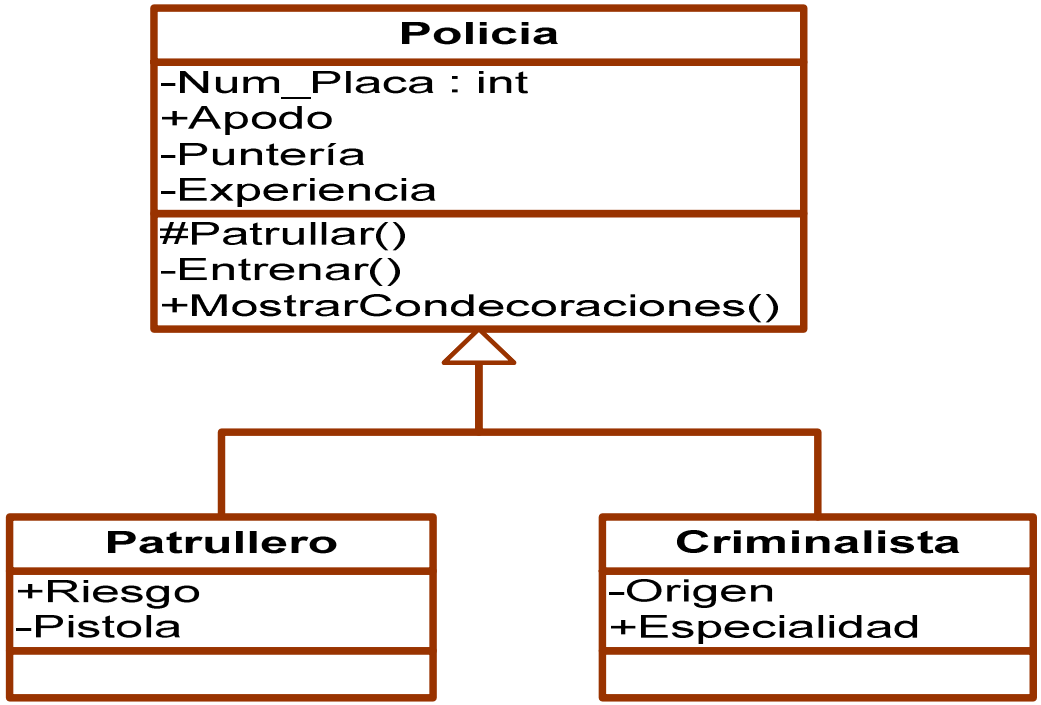


Mecanismos de extensión de UML

• Estereotipos	<<excepción>>
• Valores etiquetados	{versión=3.1}
• Restricción	{edad>18}

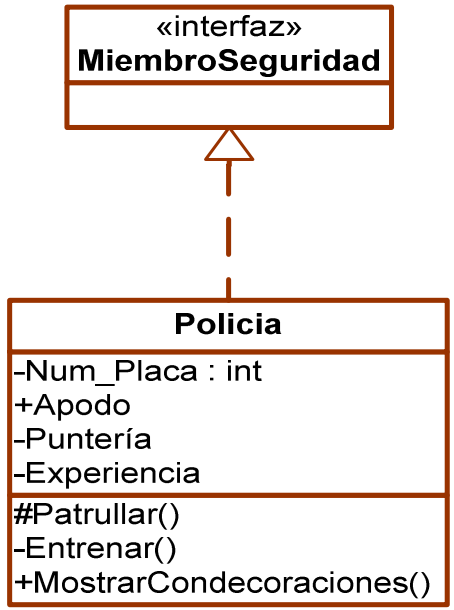
UML: Relación de Generalización

- Relación taxonómica entre una descripción general y otra más específica que la extiende
- Relación *"es un tipo de"*
- Representación del concepto de herencia



UML: Relación de Realización

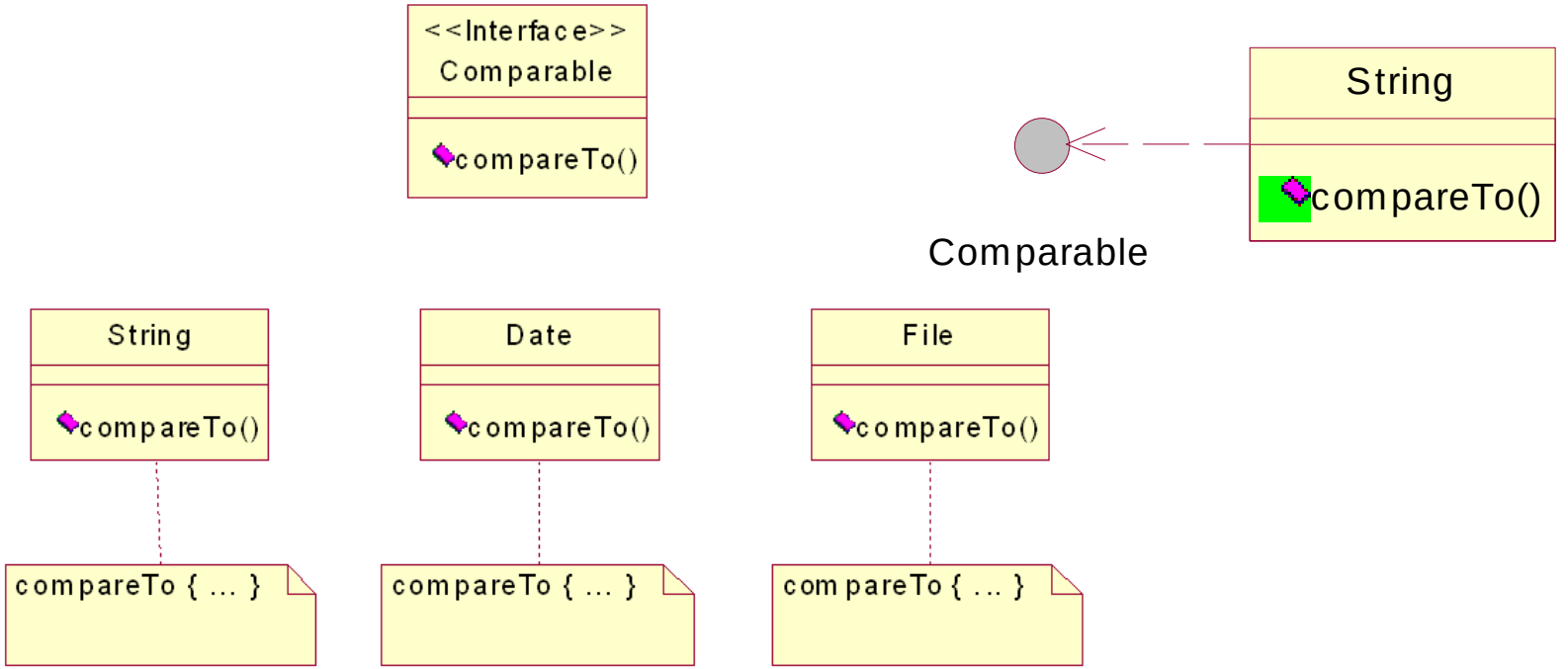
- Es una relación que implica que la parte realizante cumple con una serie de especificaciones propuestas por la clase realizada
- Situaciones de aplicación:
  - Interfaces y las clases y componentes que lo implementan
  - Casos de uso y colaboraciones que lo realizan



UML: Interfaces

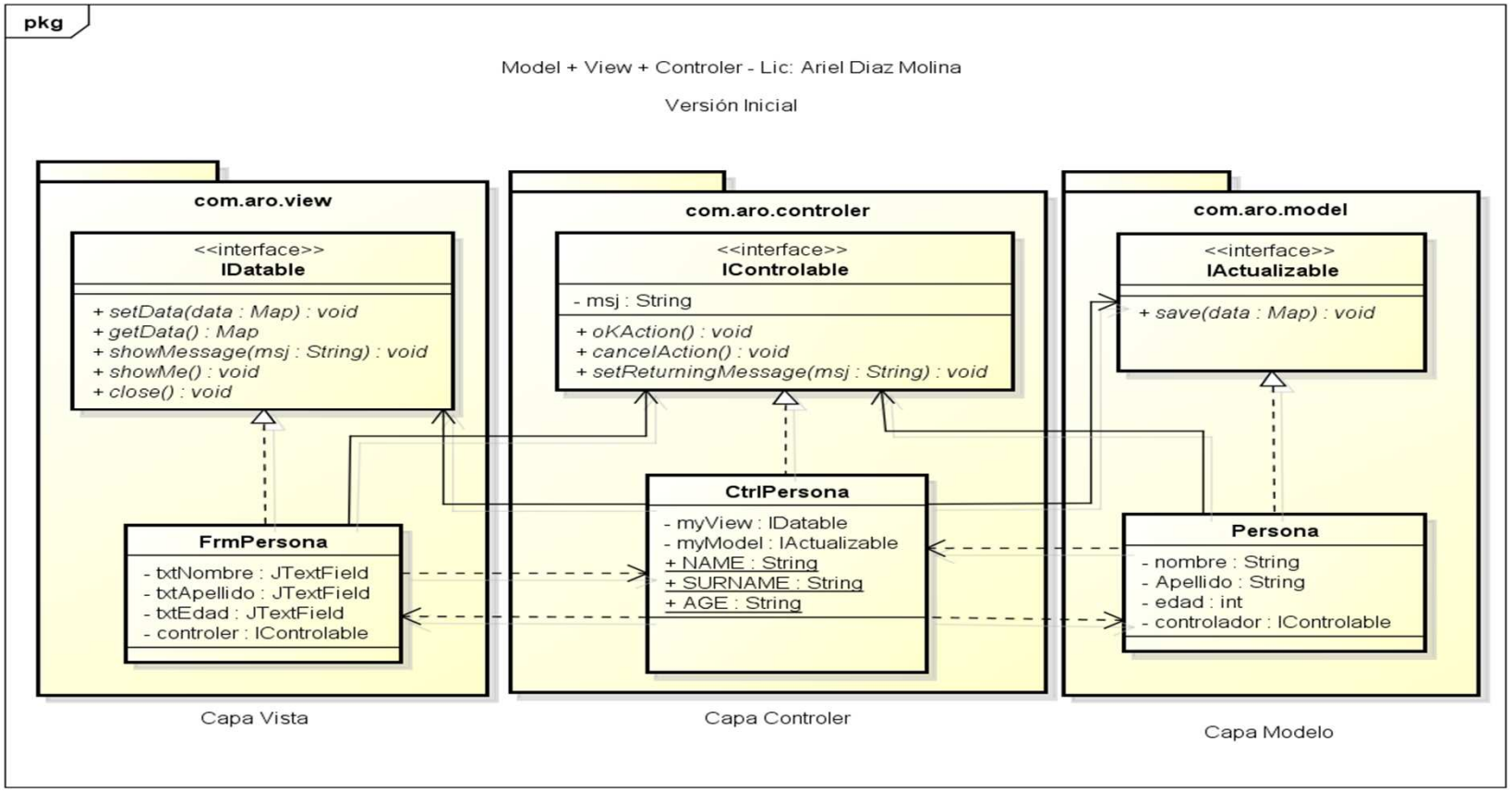
Describen un protocolo de comportamiento sin especificar su implementación.

- Contienen operaciones pero no atributos.
- Una interfaz puede ser implementada por varias clases



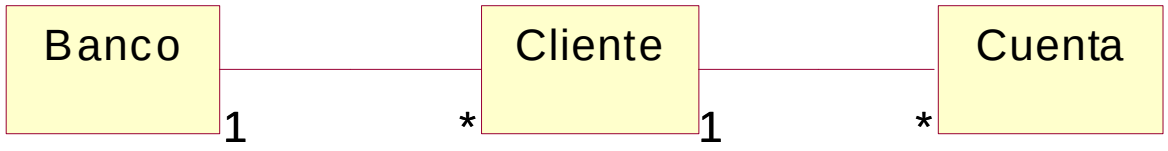
UML: Diagramas de Clases y Paquetes

- Utiliza paquetes UML para dividir grandes responsabilidades
- Un paquete UML puede utilizar los servicios de otros
- Los paquetes contienen clases y componentes UML

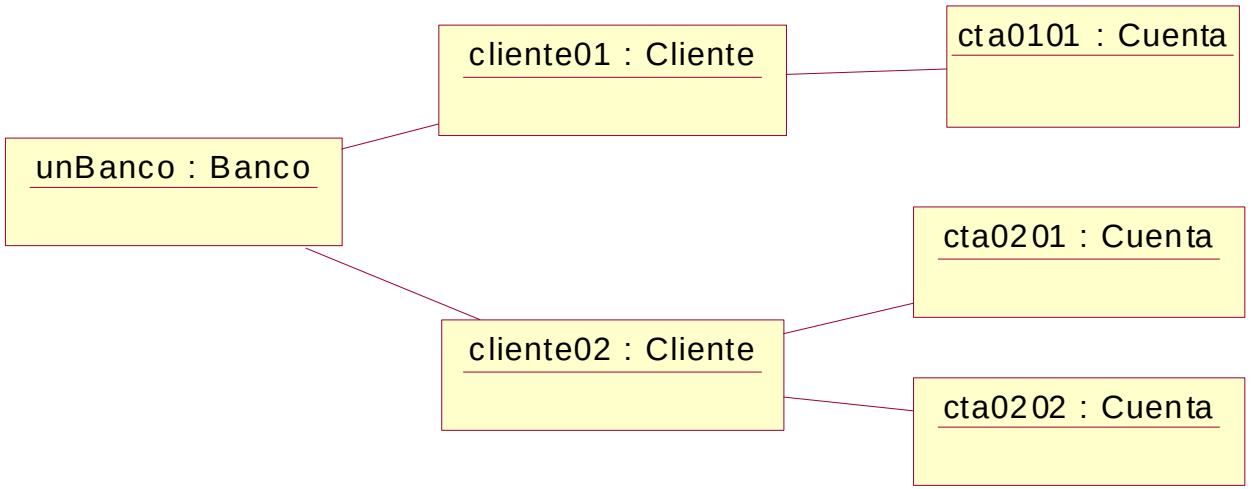


UML: Diagramas de Clases y Objetos

- **Diagrama de Clases**: muestra la abstracción de una parte del dominio.



- **Diagrama de Objetos**: representa una situación concreta del dominio. No se suelen utilizar mucho



UML: Diagrama de Componentes

- Modelan el empaquetado físico del sistema en unidades reutilizables denominadas “componentes”.
- Cada componente implementa una o más clases del diseño, definen interfaces e incluyen código fuente o ejecutable.
- Suelen utilizarse para hacer referencia a una librería externa.

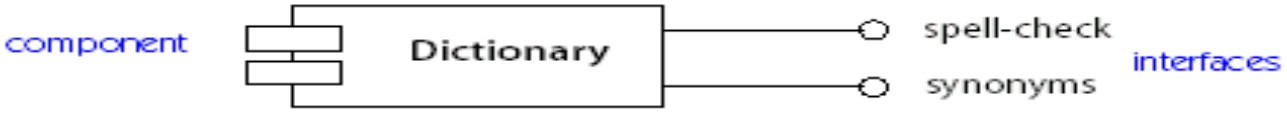


Figure 9-1. *Component with interfaces*

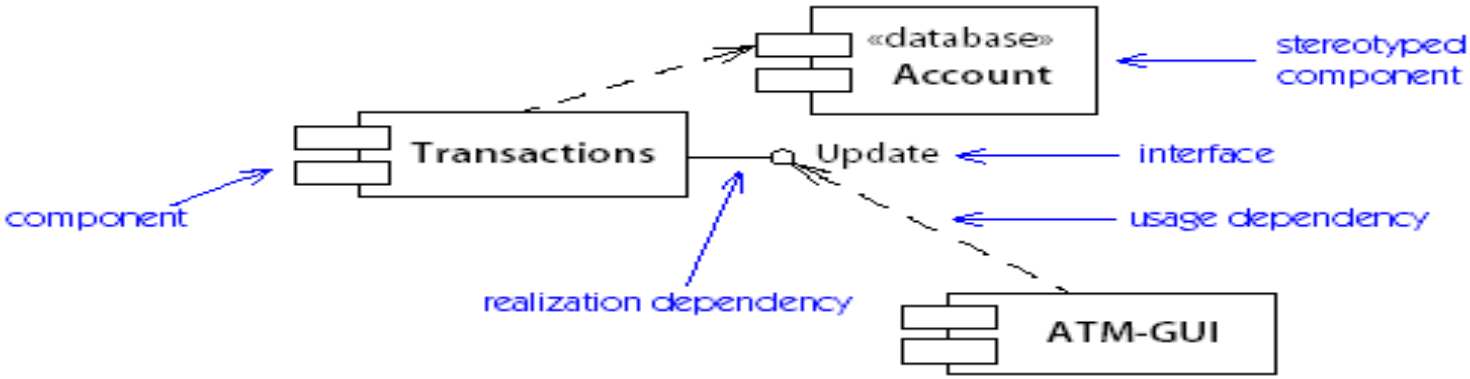


Figure 9-2. *Component diagram*

UML: Diagrama de Despliegue

- Modela los nodos y unidades funcionales de un sistema así como la comunicación entre ellos.
- Se modela como la Vista de Despliegue

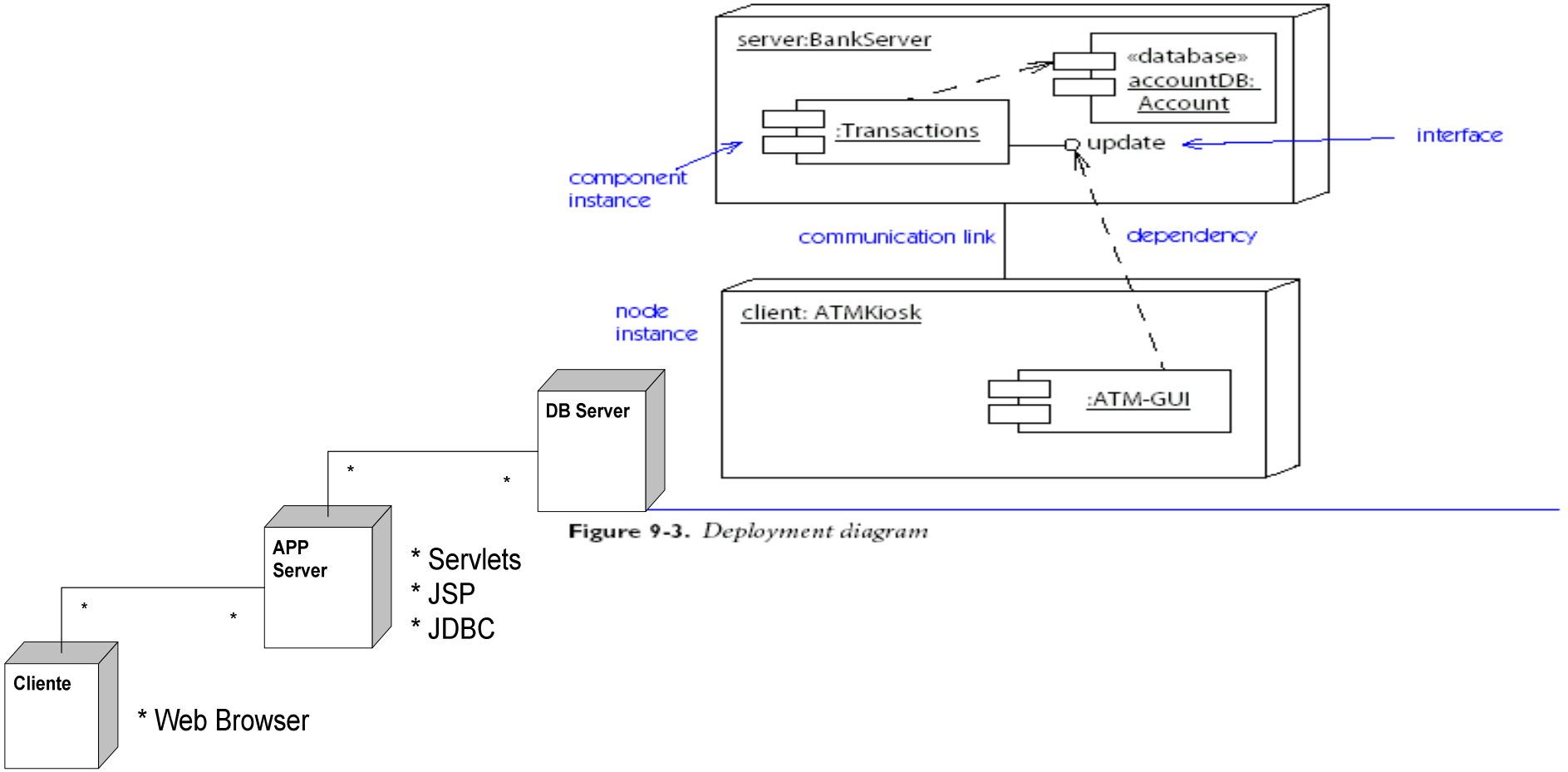
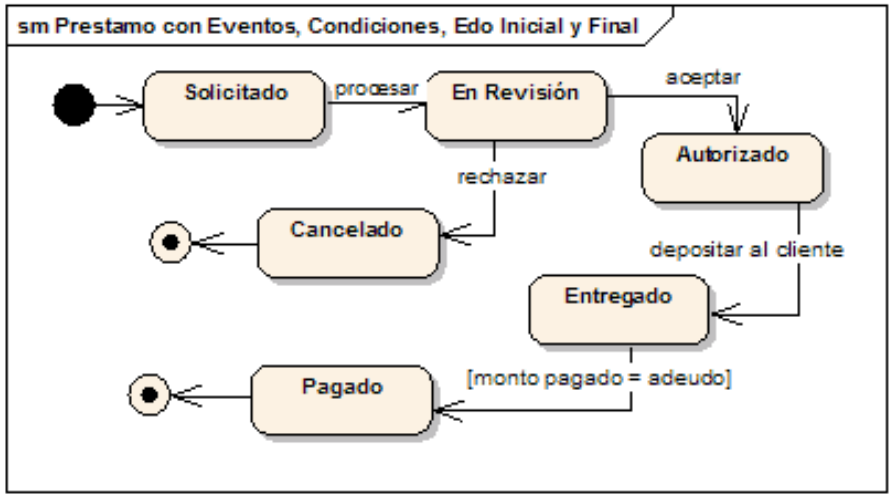
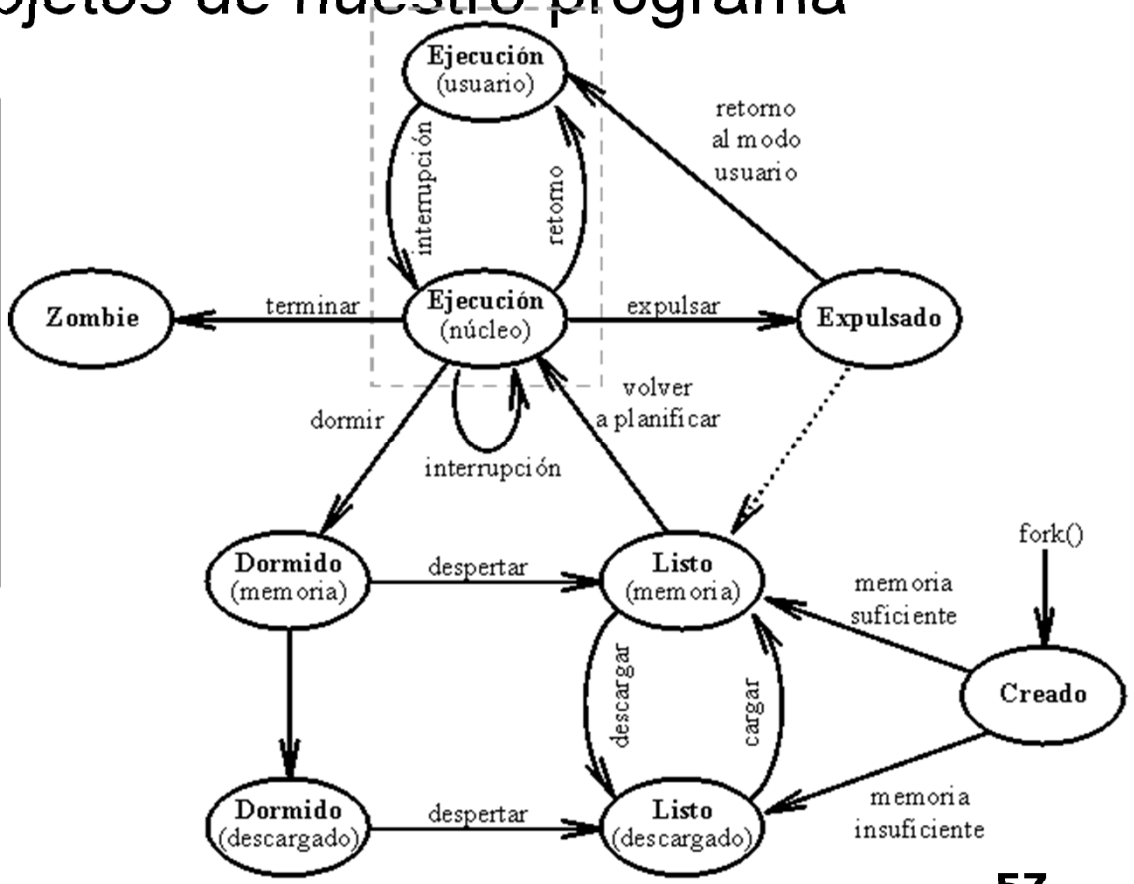


Figure 9-3. *Deployment diagram*

UML: Diagrama de Estados

- Definen estados y transiciones entre ellos motivados por eventos
- Invocaciones síncronas y asíncronas en el tiempo
- Se utiliza cuando estamos interesados en los estados y transiciones de los objetos de nuestro programa



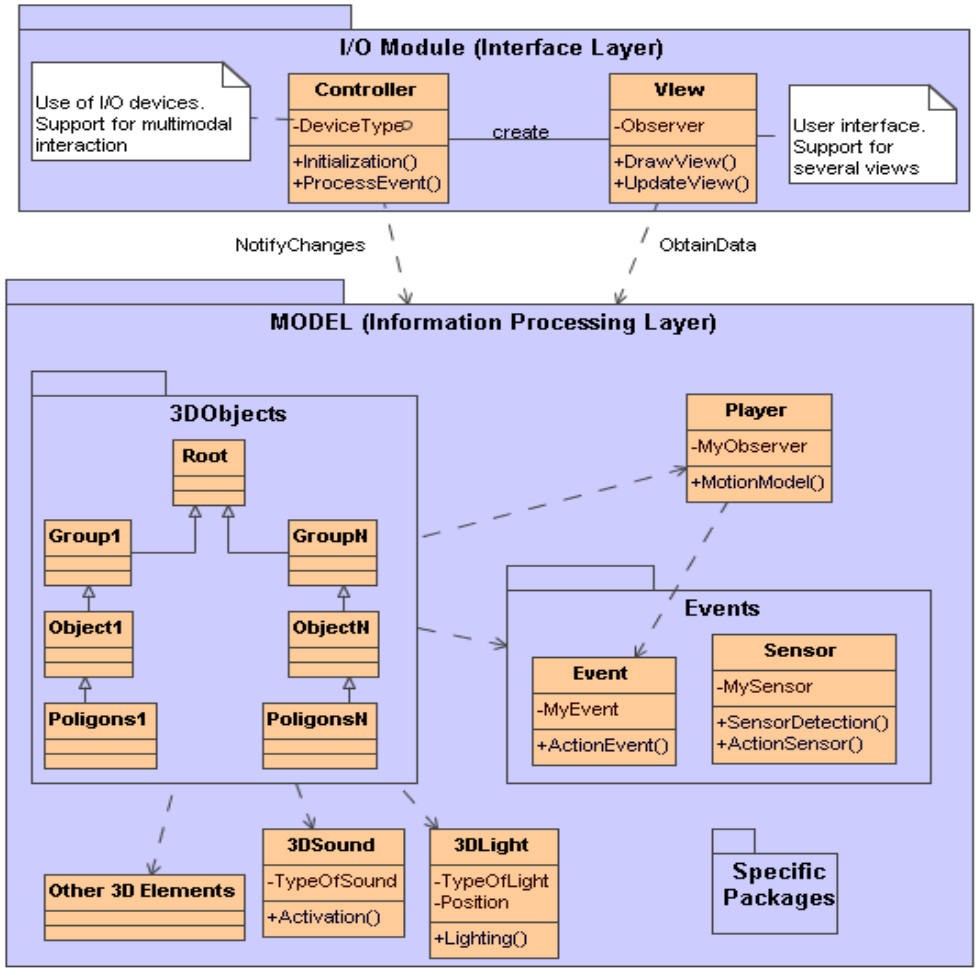


Ejemplos de Arquitecturas

Arquitectura heterogénea (combinación de estilos)

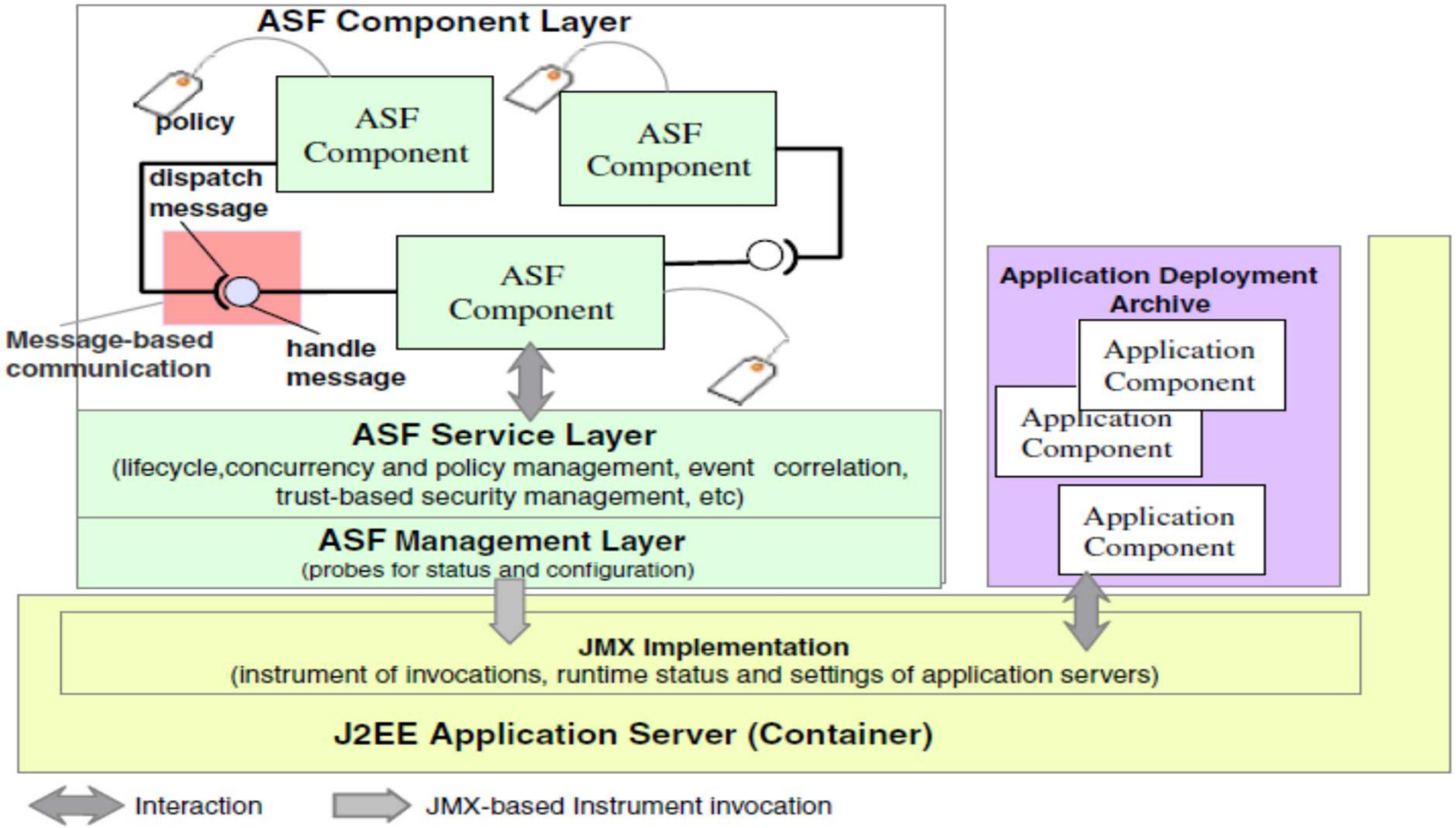
Tenemos el estilo por capas como predominante. Luego podemos observar un estilo MVC. Además, podemos ver algo de gestión de eventos e incluso una jerarquía de objetos. Al menos 2 o 3 estilos se encuentran en dicha arquitectura.

El estilo predominante es el más visible a los ojos del arquitecto software.



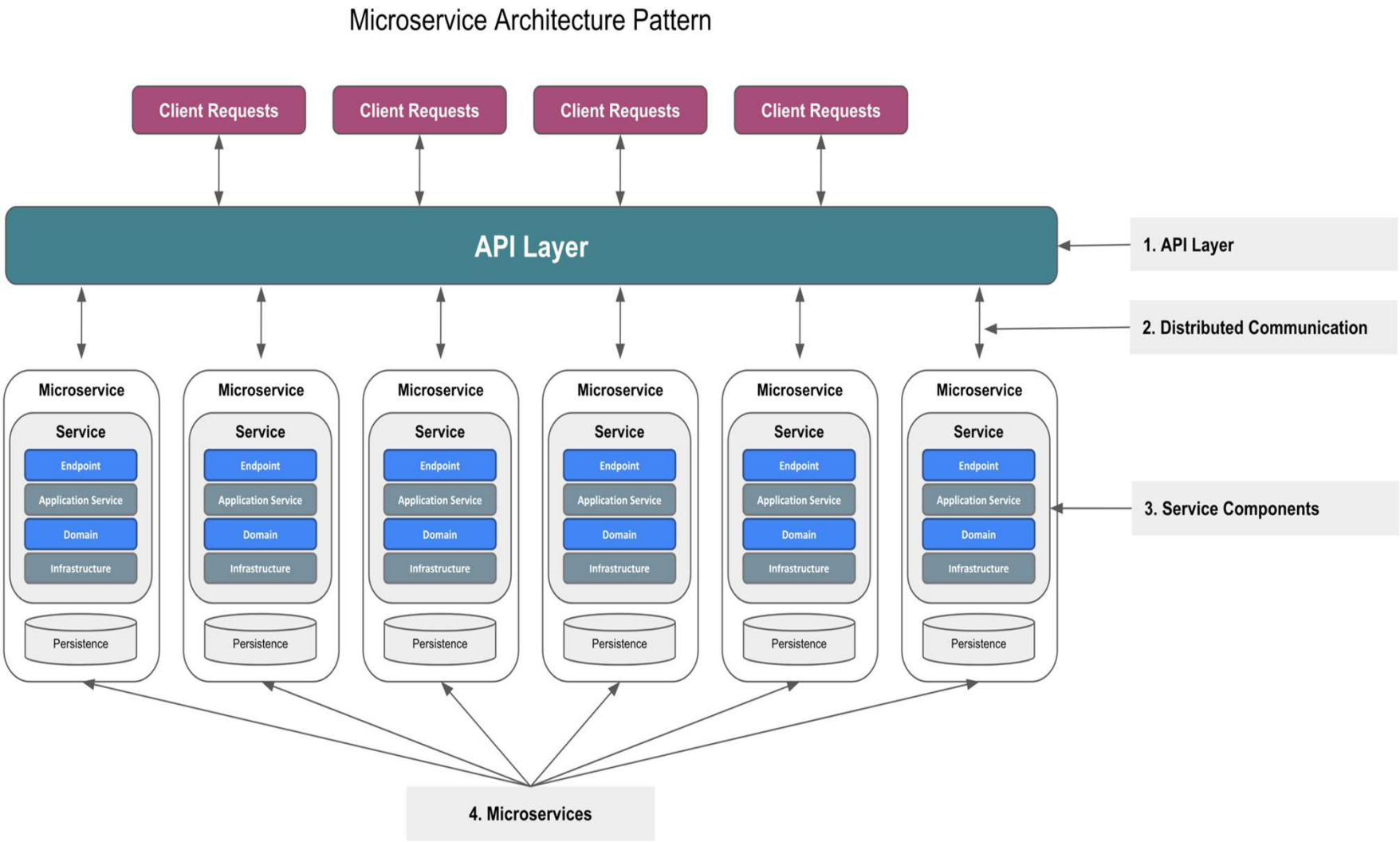
Ejemplos de Arquitecturas

Arquitectura por Capas (ASF: Adaptive Server Framework)



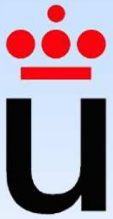
Aquí se muestran 3 capas verticales y una de ellas tiene una capa horizontal añadida en color morado

Ejemplos de Arquitecturas



Conclusiones

- Los patrones de diseño son soluciones codificadas de conocimiento reutilizables que han sido probadas.
- Los estilos arquitectónicos definen los grandes subsistemas ya áreas funcionales de la arquitectura software.
- Patrones y estilos se utilizan en la construcción de arquitecturas de referencia y software.
- UML ofrece una notación para describir y documentar arquitecturas (No es la única!) con diversos tipos de diagramas.
- Cada stakeholder tienen diferentes intereses y puntos de vista que se representan mediante diferentes diagramas. No es necesario utilizar todos los diagramas del UML, sólo los necesarios.



Universidad Rey Juan Carlos

Grado en Ingeniería Software

Tema 4

Decisiones de Diseño y Conocimiento Arquitectónico

Índice de la Presentación

- Introducción
- Caracterización de Decisiones
- Vista y Procesos
- Dependencias y Evolución
- Herramientas de AKM
- Decisiones en Contextos GSD/Ágil/OSS
- Reflection
- Conclusiones

Introducción

- Una **Arquitectura Software** debe ser considerada como el resultado de un conjunto de decisiones de diseño (ADD – Architectural Design Decisions) mas que como un conjunto de componentes y conectores (Bosch, 2004).
- Las decisiones de diseño permiten conocer los motivos subyacentes de una decisión tomada y las alternativas evaluadas.
- Permiten seguir mejor la evolución de una arquitectura software.

Introducción

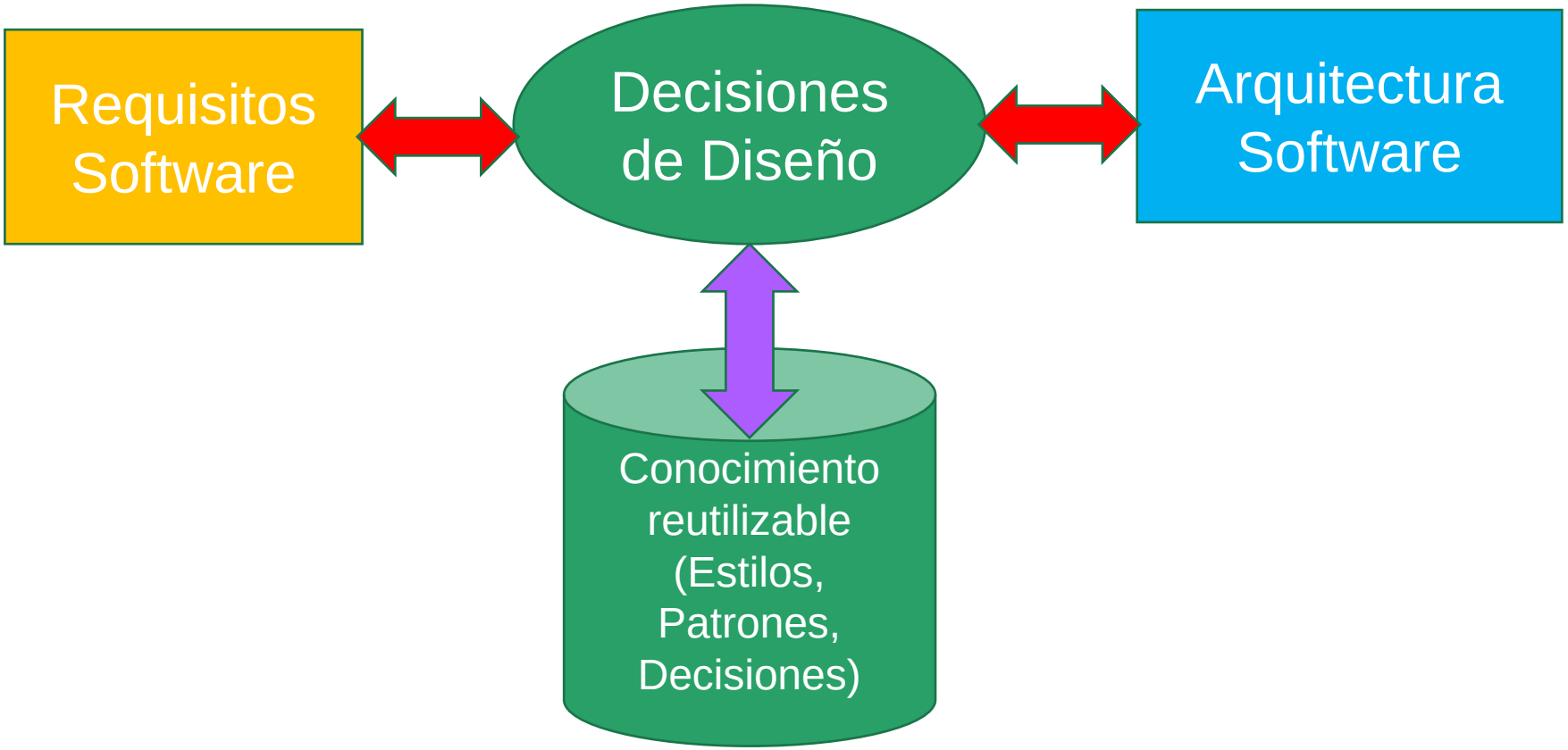
- Son necesarias para comprender las alternativas evaluadas durante el proceso de diseño.
- Son un conocimiento implícito, subyacente y tácito que reside en la mente de los arquitectos software y que es necesario hacer explícito para que pueda ser (re)utilizable y transferible.
- Es necesario cambiar los hábitos de diseño actuales para favorecer su captura.

Introducción

- Las ADD cubren el hueco entre requisitos y diseño
- Facilitan que el conocimiento experto no se evapore
- Son útiles para enseñar a diseñadores poco expertos
- Permiten comprender el motivo de decisiones buenas y malas
- Son reproducibles

Introducción

Las decisiones de diseño cubren el hueco entre los requisitos software (de donde provienen) y las arquitecturas software (el resultado de tomar un conjunto de decisiones). Las decisiones capturadas se pueden considerar un conocimiento reutilizable



Caracterización de Decisiones

- Caracterizar el conocimiento que se pretende almacenar sobre las decisiones de diseño es el primer paso.
- Tres aproximaciones:
  - Plantillas de texto (Tyree & Akerman, Capilla, Babar)
  - Ontologías (Kruchten)
  - Anotaciones en código (Zimmermann)

Para capturar el conocimiento de las ADD es necesario caracterizarlas o describirlas de alguna manera ya que se expresan en lenguaje natural

Caracterización de Decisiones

- Las plantillas de texto definen un conjunto de atributos que capturan el conocimiento de las decisiones y su motivación.
- Algunas aproximaciones utilizan un conjunto menor de atributos y otros opcionales definidos por el arquitecto software para reducir el coste de capturar un gran número de atributos.
- Las plantillas deben ser flexibles en cuanto a número de ítems o atributos a capturar

Caracterización de Decisiones

- Atributos para describir una decisión
  - Nombre: Asignar un nombre corto a la decisión
  - Descripción de la decisión: Descripción corta pero explicativa
  - Estatus: Si la decisión esta "pendiente/aprobada/rechazada"
  - Categoría
  - Lógica: El mótivo de tomar dicha decisión
  - Responsable
  - Fecha
  - Pros/Cons
  - Etc…

Si una decisión de diseño fuera seleccionar el estilo arquitectónico Cliente/Servidor podemos caracterizar esta con la lista de ítems o atributos que se muestran en la transparencia.

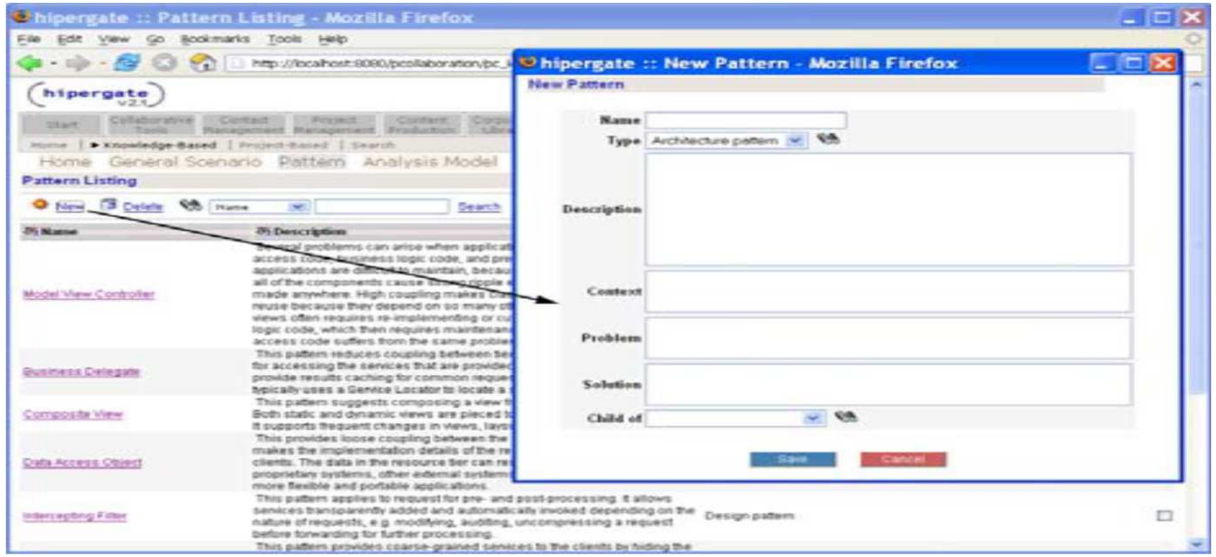
Caracterización de Decisiones (Tyree y Akerman, 2005)

14 atributos fijos para caracterizar las DDs

- Issue: Problema de diseño
- Decision: Descripción de la decisión
- Status: Pendiente, Decidida, Aprobada
- Constraints: Restricciones respecto al entorno
- Positions: Alternativas
- Related decisions
- Related requirements
- Related artifacts: otros elementos del diseño

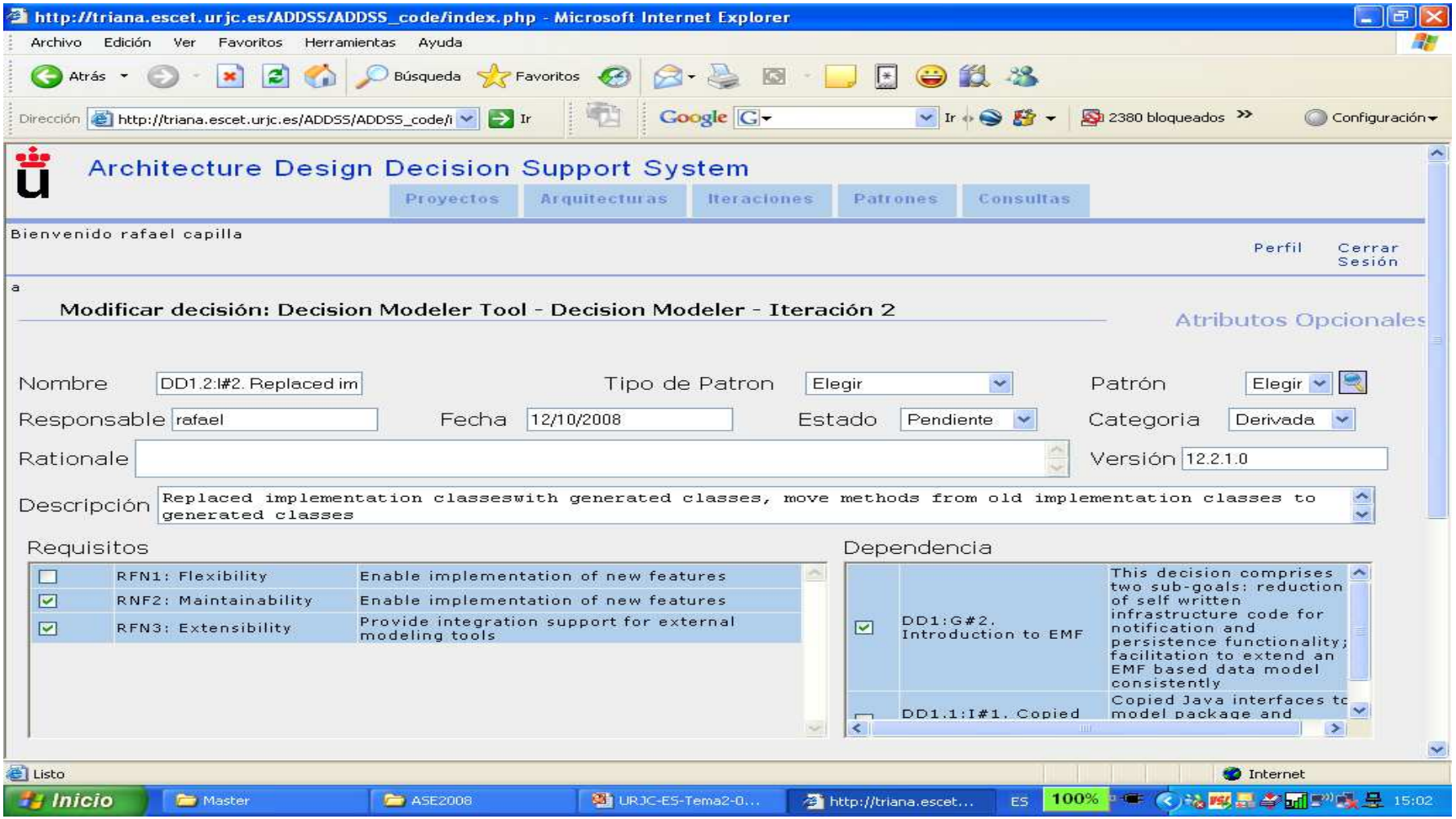
Caracterización de Decisiones
(Captura de AK con la herramienta PAKME, 2006)

- Basada en la plantilla de Tyree
- Relación con atributos de calidad (QA)
- Soporta compartición de conocimiento (knowledge sharing)
- Captura patrones de diseño (design patterns)
- Permite describir “scenarios”
- PAKME: Captura, Mantenimiento, Recuperación, Presentación



Caracterización de Decisiones

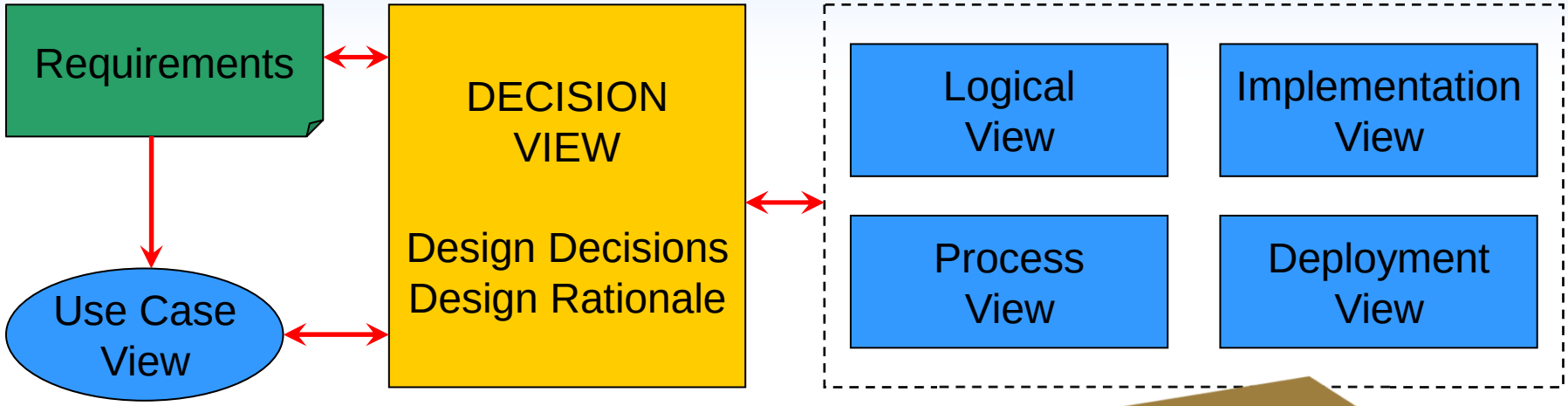
Ejemplo con herramienta ADDSS (URJC)



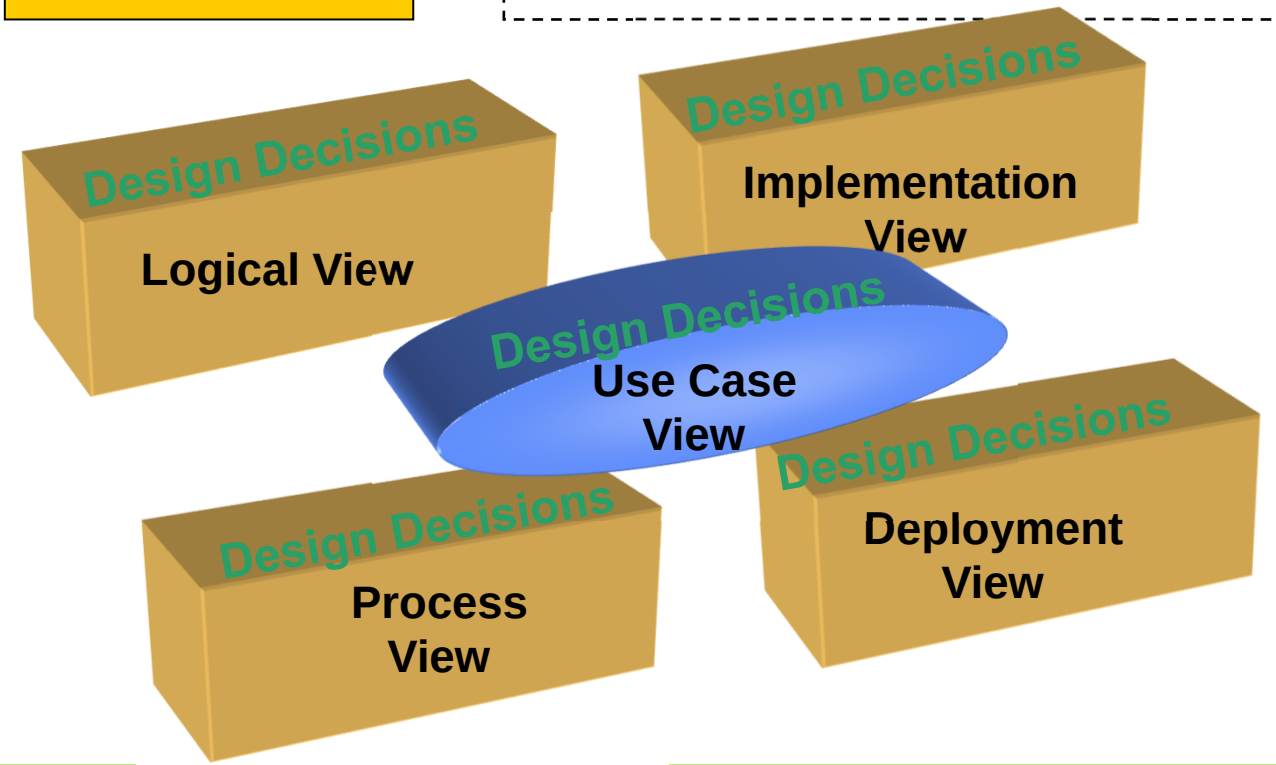
Vista y Procesos

- Las decisiones de diseño son elementos de primera clase definidos en el estándar ISO/IEC 42010
- Se capturan y documentan junto con las arquitecturas software
- Representan un nuevo punto de vista frente a los tradicionales de Kruchten, Soni, Rozanski, etc.

Vista de la Decisión



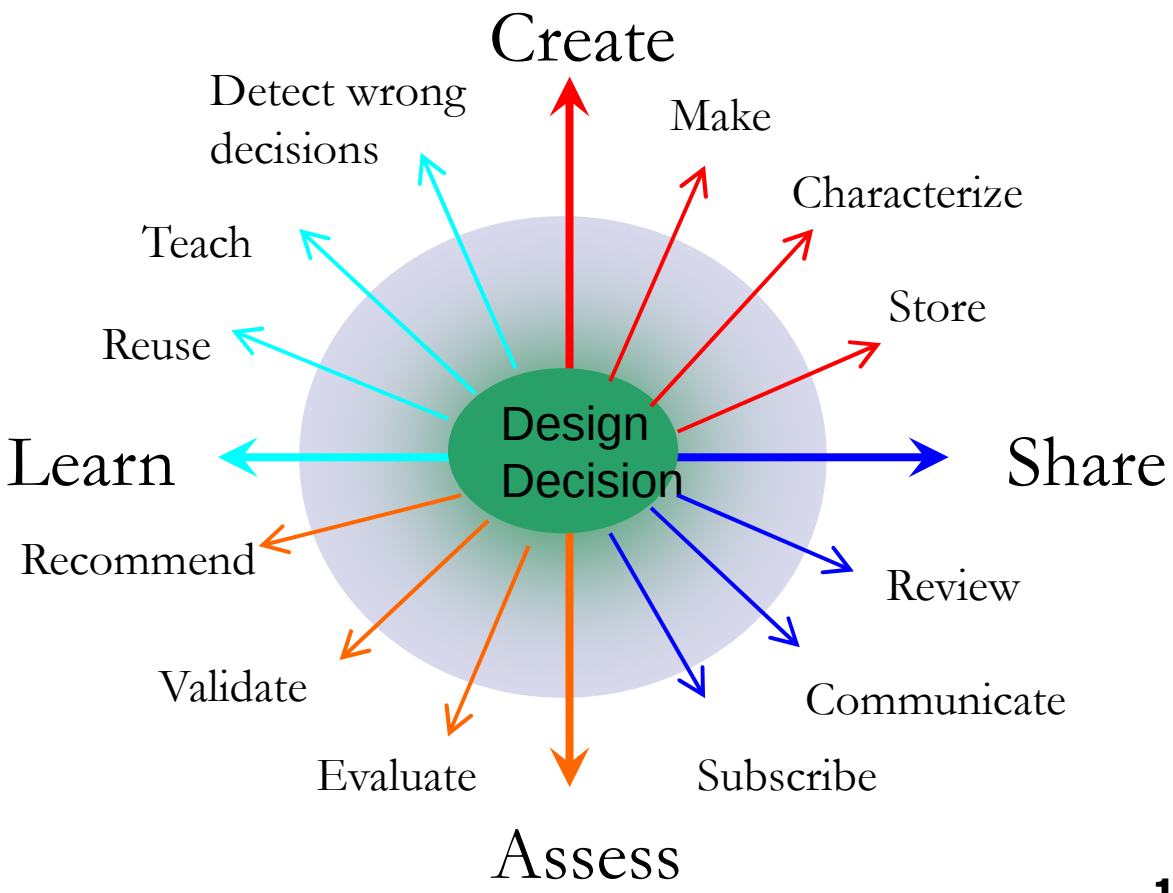
Lavista de la decisión es una nueva vista arquitectónica que es transversal a las restantes vistas del modelo 4+1



Vista y Procesos

- La captura y representación de ADDs es solamente una de las actividades de la Gestión del Conocimiento Arquitectónico (AKM – Architecture Knowledge Management)

Son los posibles procesos que surgen para crear una decisión, compartirla, utilizarlas para asesorar a arquitectos menos expertos y aprender de ellas.



Vista y Procesos

- Capturar las ADD y documentarlas junto con su arquitectura es el primer paso.
- Establecer relaciones de traceabilidad con requisitos y arquitecturas.
- Compartir las decisiones con otros stakeholders ("**knowledge sharing**"): Documentos, RSS
- Mantener y evolucionar las decisiones a lo largo del tiempo ("**Decision History**") en un histórico de decisiones

Dependencias y Evolución

- Las herramientas de AKM suelen incluir forma de establecer dependencias explícitas a requisitos y arquitecturas desde las decisiones.
- Las dependencias entre decisiones, REQ y AS permite establecer una traceabilidad completa para:
  - Averiguar las causas y orígenes de cambios en el diseño ("**backward traceability**").
  - Estimar el análisis de impacto en el diseño por cambios en los requisitos ("**forward traceability**").

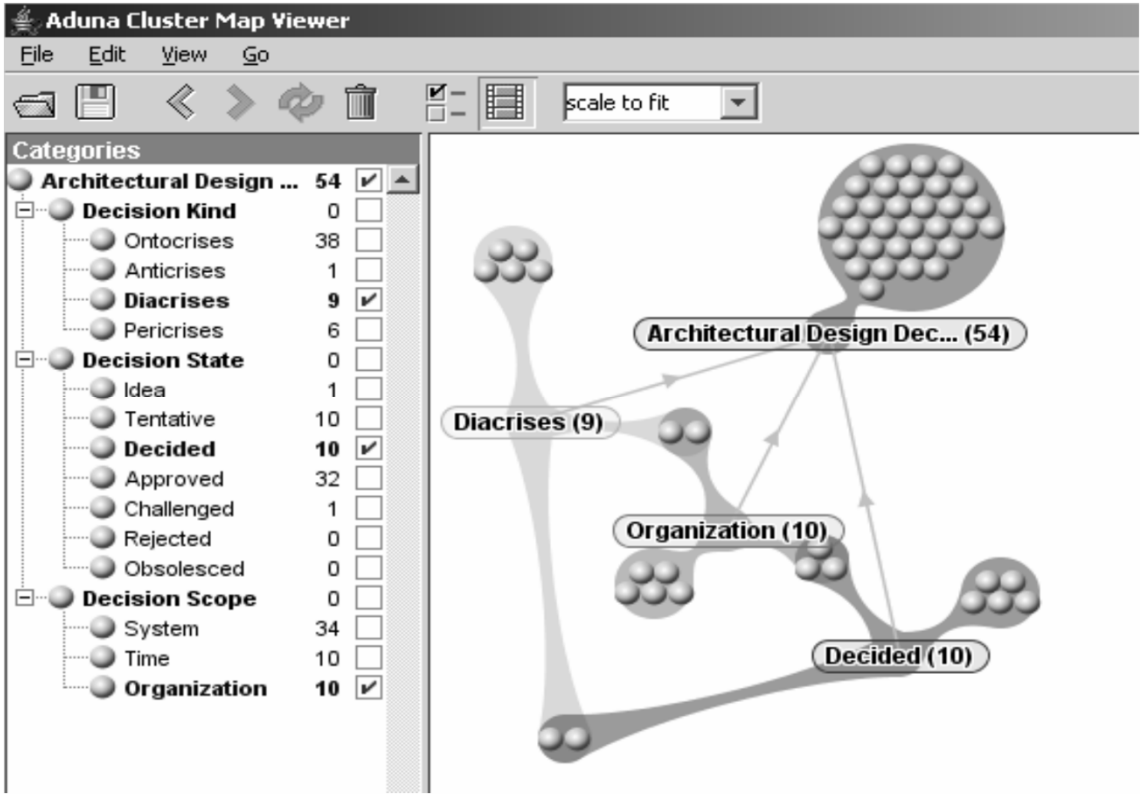
ADD: Traceabilidad Interna

- Las dependencias entre decisiones de diseño constituyen una traceabilidad interna para conocer el impacto en una decisión cuando otra se modifica.
- Si la granularidad de las decisiones es fina, la red de dependencias entre decisiones puede crecer enormemente.
- Limitar decisiones a nivel a clase en la arquitectura.

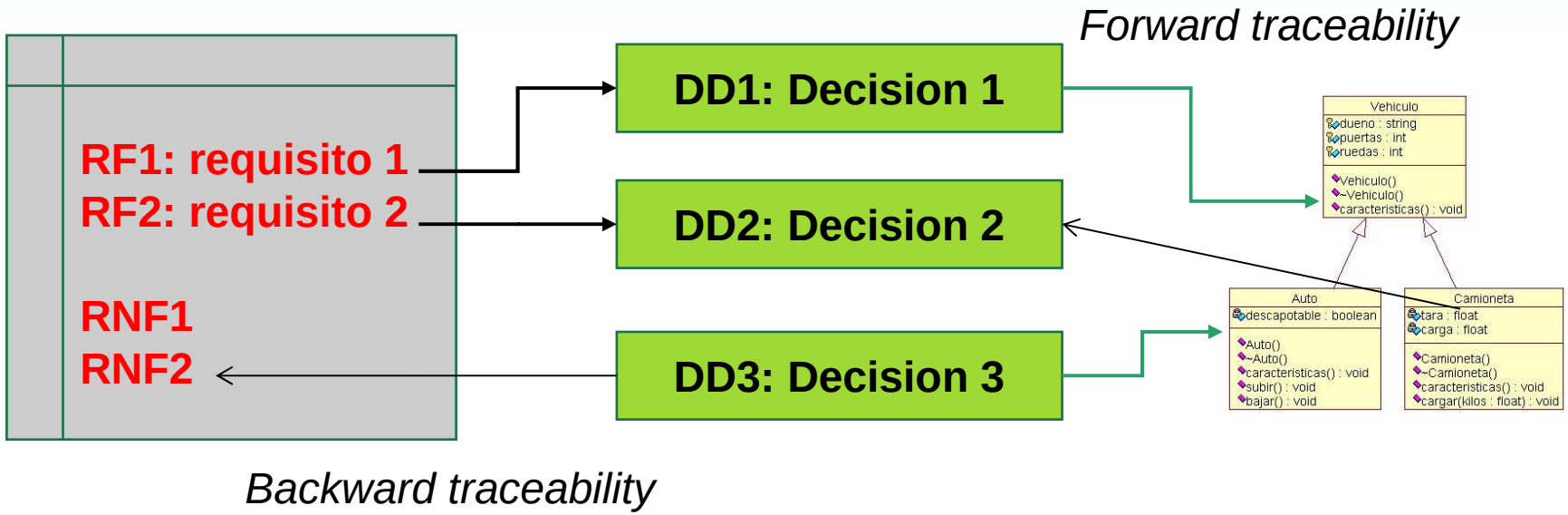
Traceabilidad Interna (Ontologías)

Decision A relacionada con **Decision B**
Decision A excluye a **Decision B**
Decision A permite **Decision B**
Decision A es conflicto con **Decision B**

Pueden existir conflictos entre decisiones incompatibles o contrarias que tenemos que resolver. O bien, una decisión puede excluir a otra (e.g. Si utilizamos una arquitectura de 2 capas no es posible incluir el middleware X)



ADD: Traceabilidad Externa



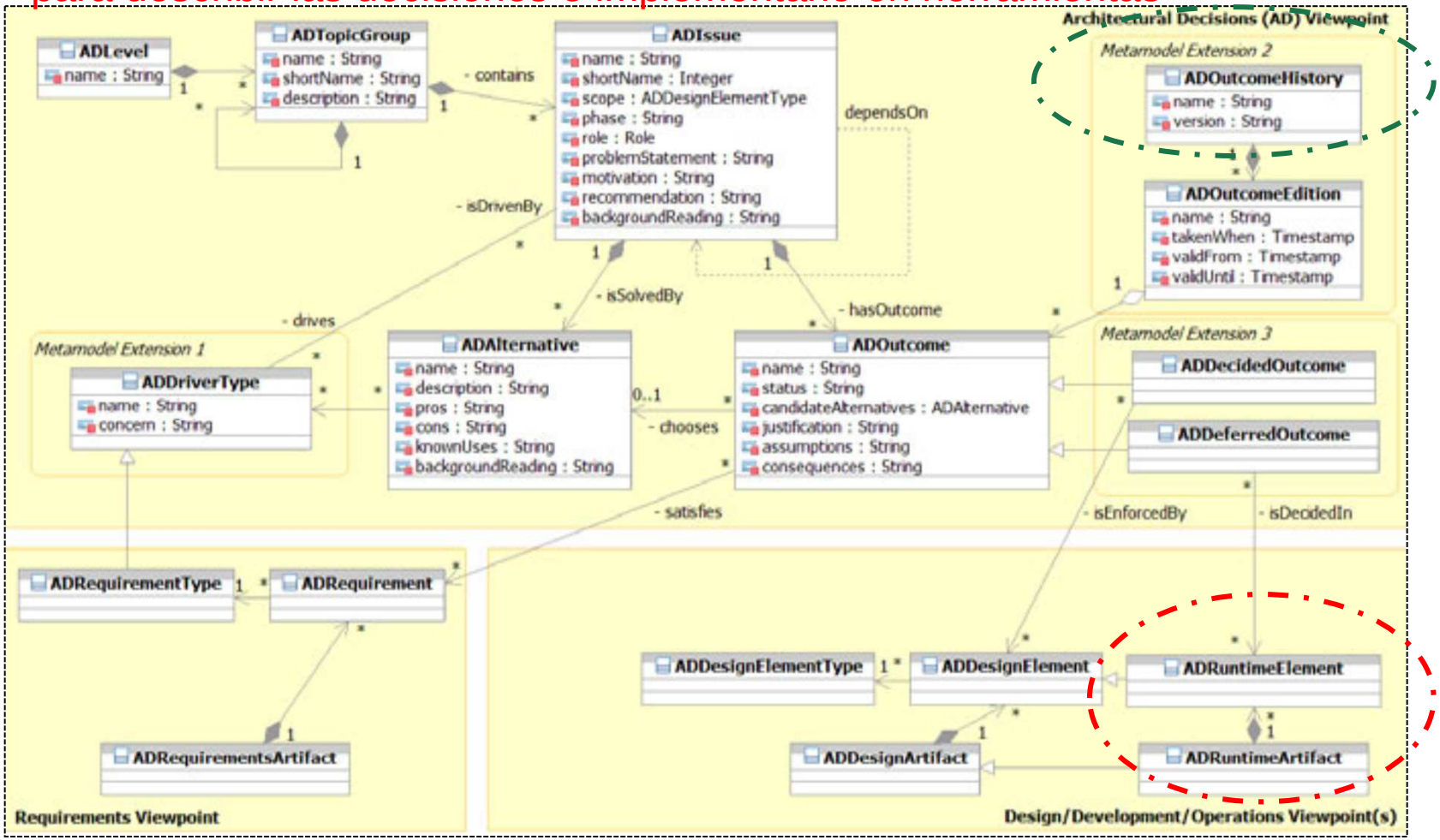
Gracias a que las decisiones se sitúan entre los requisitos y las arquitecturas podemos definir enlaces de traceabilidad para por ejemplo, BACKWARD: conocer el origen (el requisito) de una decisión de diseño mal tomada (un patrón mal seleccionado) o bien conocer FORWARD: el impacto en el diseño de un cambio en los requisitos

Evolución de Decisiones

- Un campo versión en las decisiones permite gestionar un histórico de las decisiones para:
  - Conocer la historia de las decisiones tomadas.
  - Para revertir cambios si fuera necesario.
  - Conocer las historia de la arquitectura y sus cambios.
- El histórico de decisiones debe mantenerse separado de la red de decisiones actual.

Meta-modelo con Soporte para Evolución

Los metamodelos ayudan a describir de manera lógica los ítems necesarios para describir las decisiones e implementarlo en herramientas



Evolución de Decisiones

- Posibilidad de recrear decisiones pasadas con el Histórico
- Posibilidad de revertir decisiones
- Posibilidad de conocer las alternativas consideradas y su lógica y usarlas para entender como se diseñó un sistema
- Enseñar y asesorar a arquitectos software con menos experiencia

Herramientas de AKM

- Desde 2005 se han creado un conjunto de herramientas de investigación prototipo para la captura y compartición de decisiones de diseño.
- Muchas de ellas son herramientas Web.
- **PAKME**: Herramienta Web para la captura de decisiones basada en la plantilla de Tyree & Akerman (2005) con soporte para atributos de calidad (Babar et al).
- **Archium**: Herramienta de modelado que permite la captura de DD con avisos sobre decisiones incompatibles.

Herramientas de AKM

- **AREL**: Herramienta de modelado UML que permite incluir DD en los modelos de clases. Soporta dependencias y e histórico de decisiones.
- **ADDSS**: Herramienta Web para la captura de conocimiento. Soporta personalización de atributos, evolución de las arquitecturas y dependencias. Las ultimas versiones incorporan mecanismos de compartición.
- **The Knowledge Architect**: Herramienta plug-in Word que permite marcar las decisiones de diseño con otros elementos del documento Word.

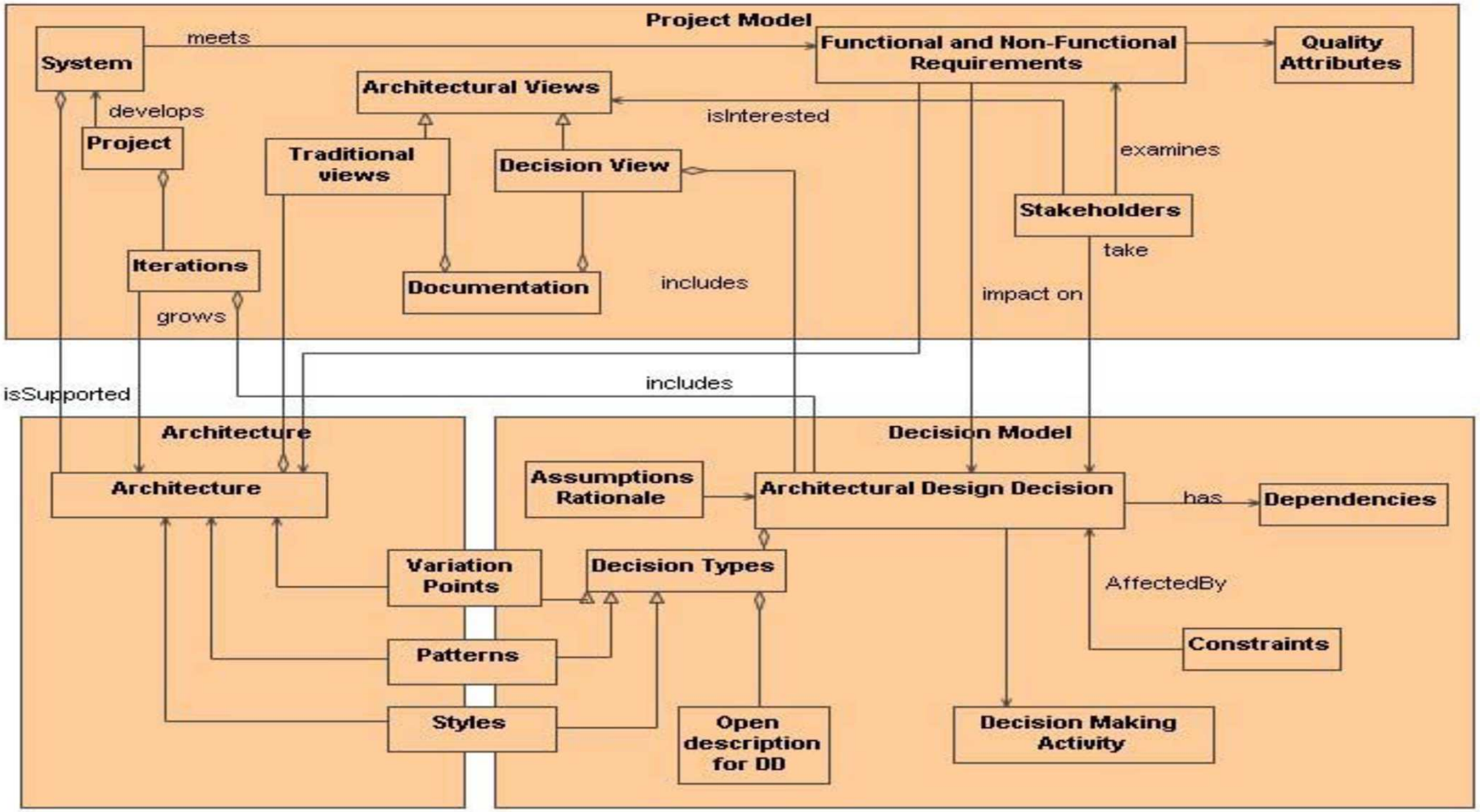
Herramientas de AKM

- **EAGLE**: Herramienta Web para compartir conocimiento. Incluye RSS y gestión de usuarios. Basada en la plataforma Hipergate.
- **AdWiki**: Herramienta Web similar a EAGLE. Almacena más de 400 decisiones SOA.

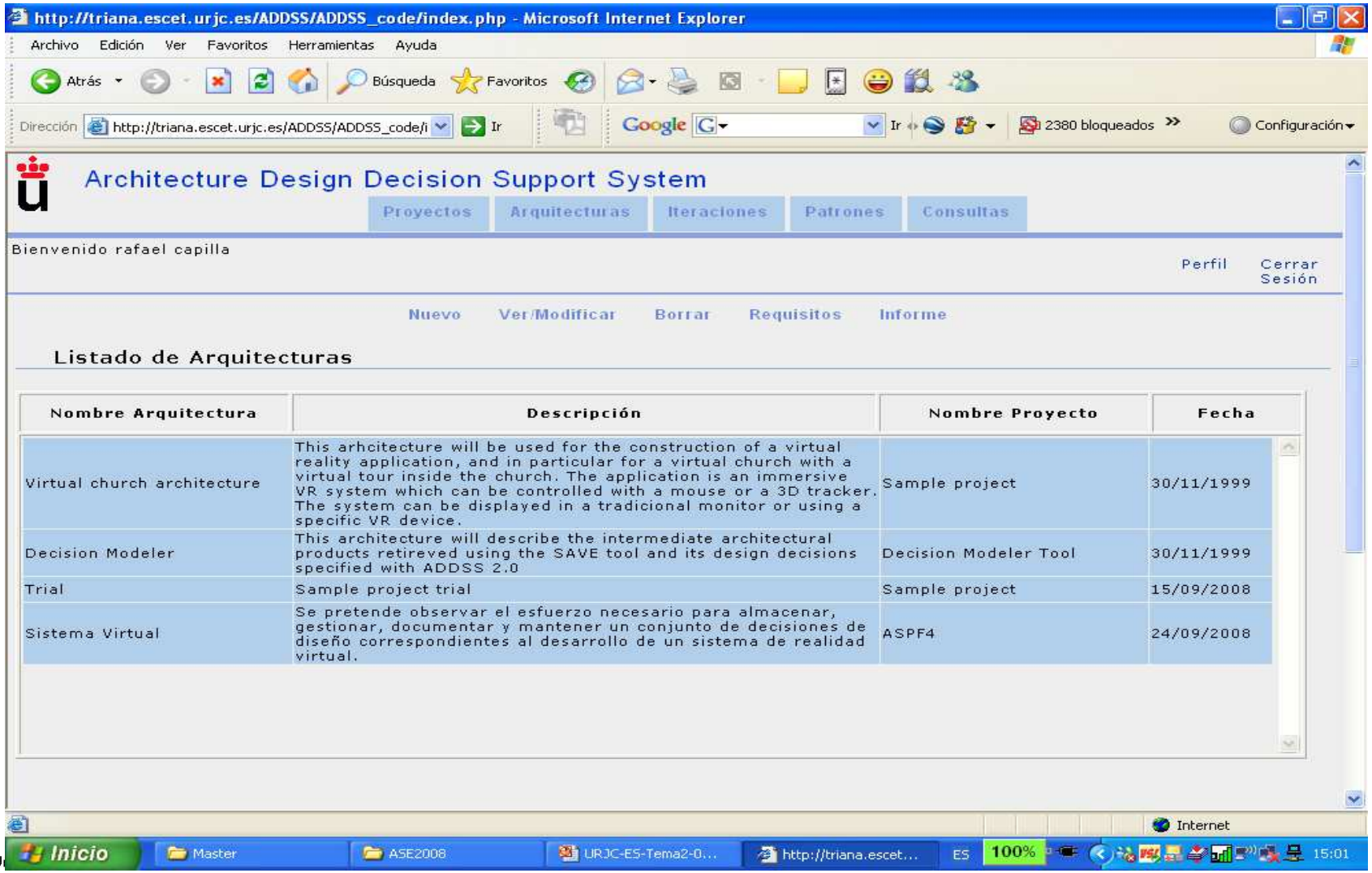
Herramienta ADDSS (2006-2011)

- Architecture Design Decision Support System
  - Captura DD con sus arquitecturas.
  - Soporte para vistas arquitectónicas.
  - No permite modelado UML.
  - Permite atributos opcionales y obligatorios para describir las decisiones.
  - Categoría y estatus de las decisiones.
  - Dependencias a otras decisiones.
  - Dependencias entre requisitos, decisiones y arquitecturas.
  - Soporte RSS y Chat.
  - Documentación PDF automática.
  - Visualiza la evolución de las arquitecturas.

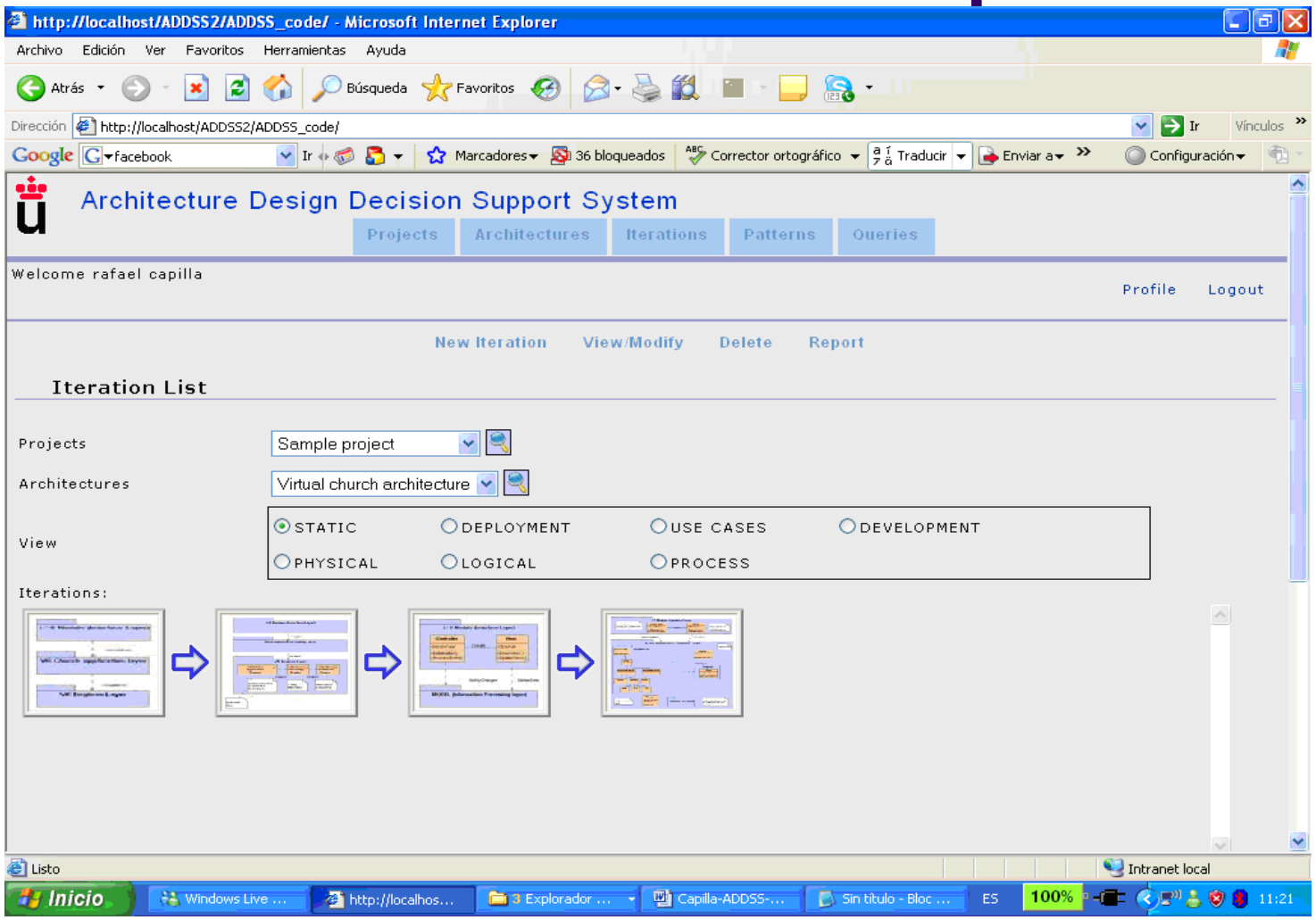
Meta-modelo de ADDSS



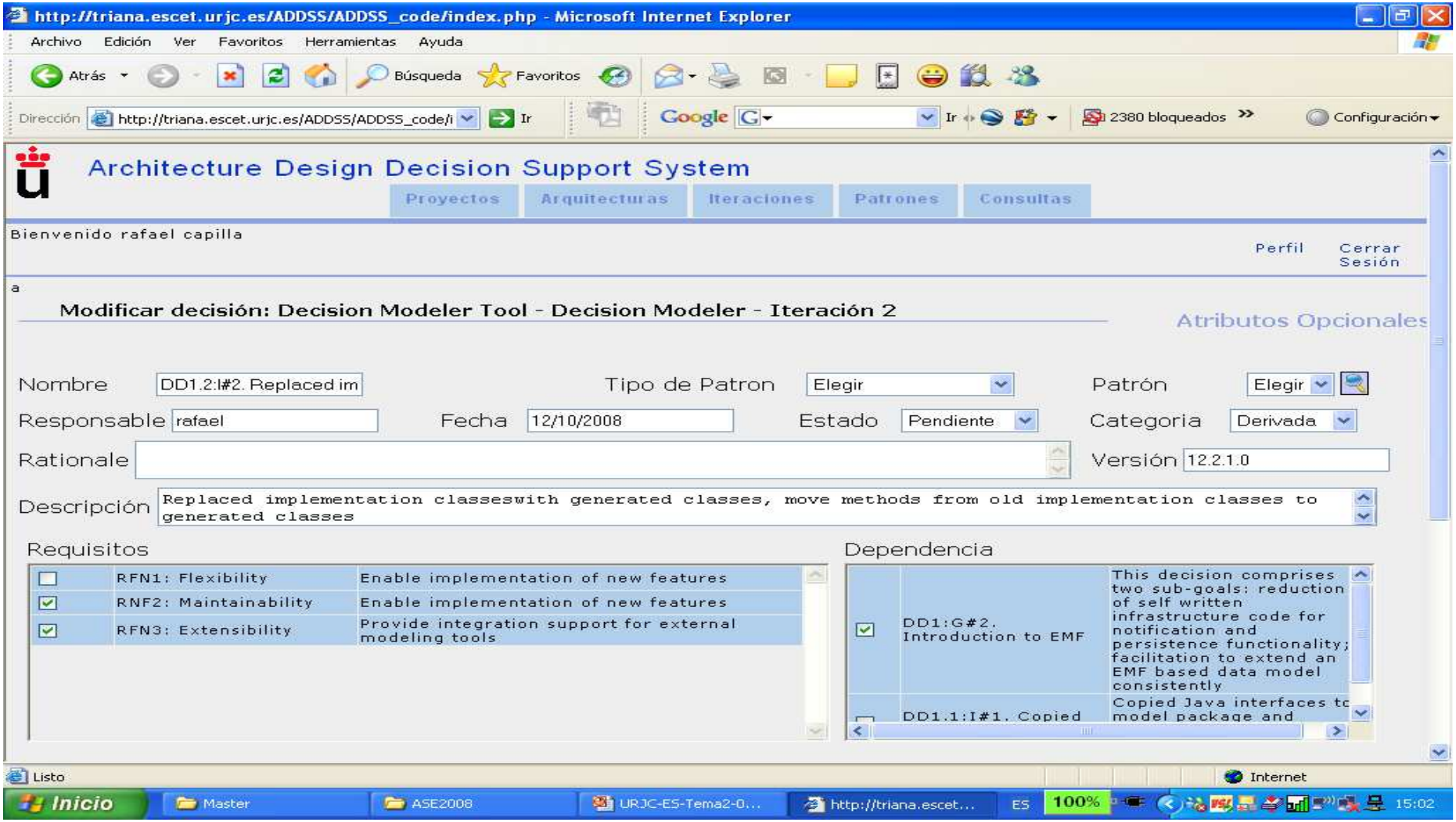
ADDSS: Decisiones



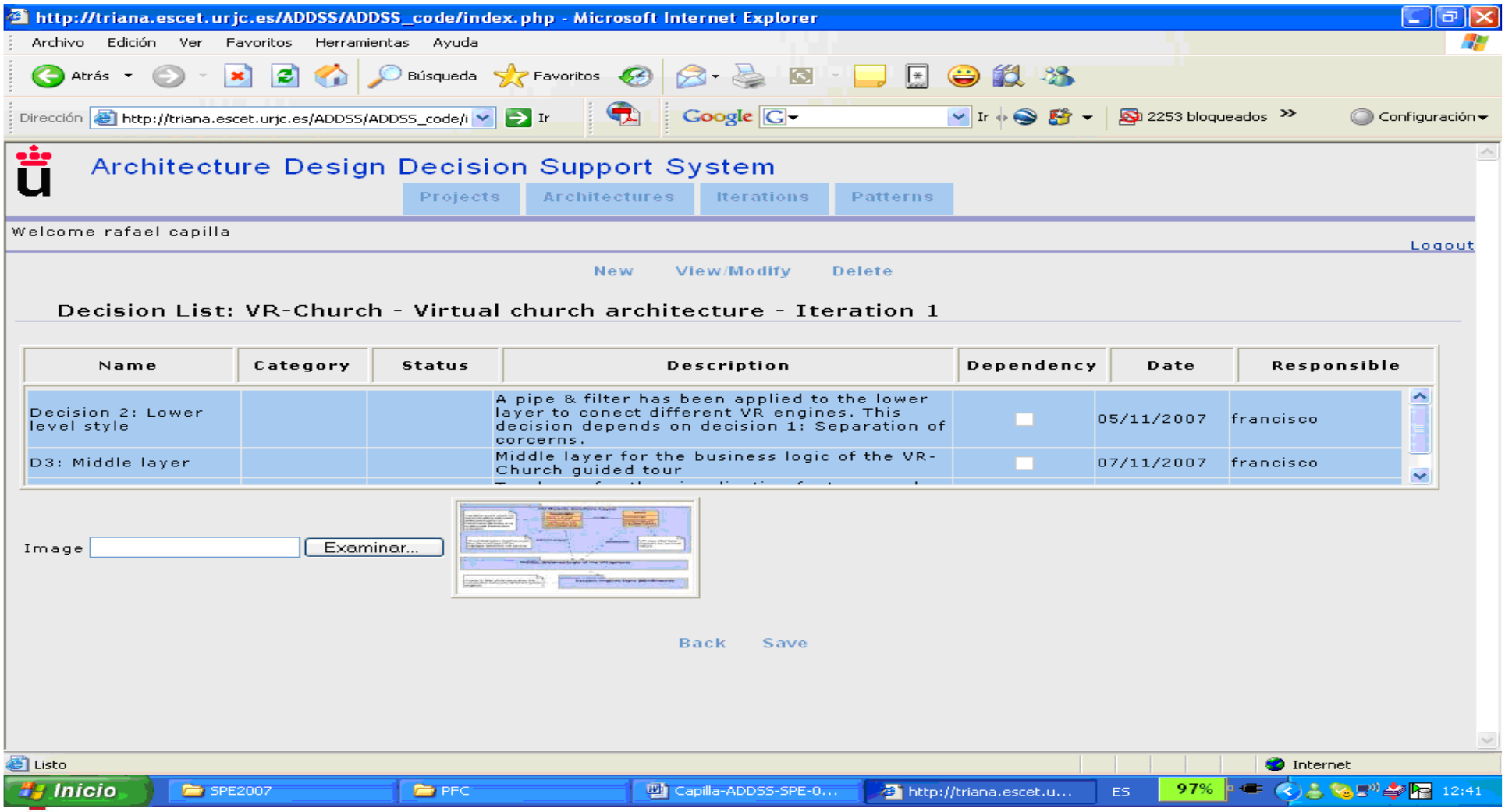
ADDSS: Evolución de la Arquitectura



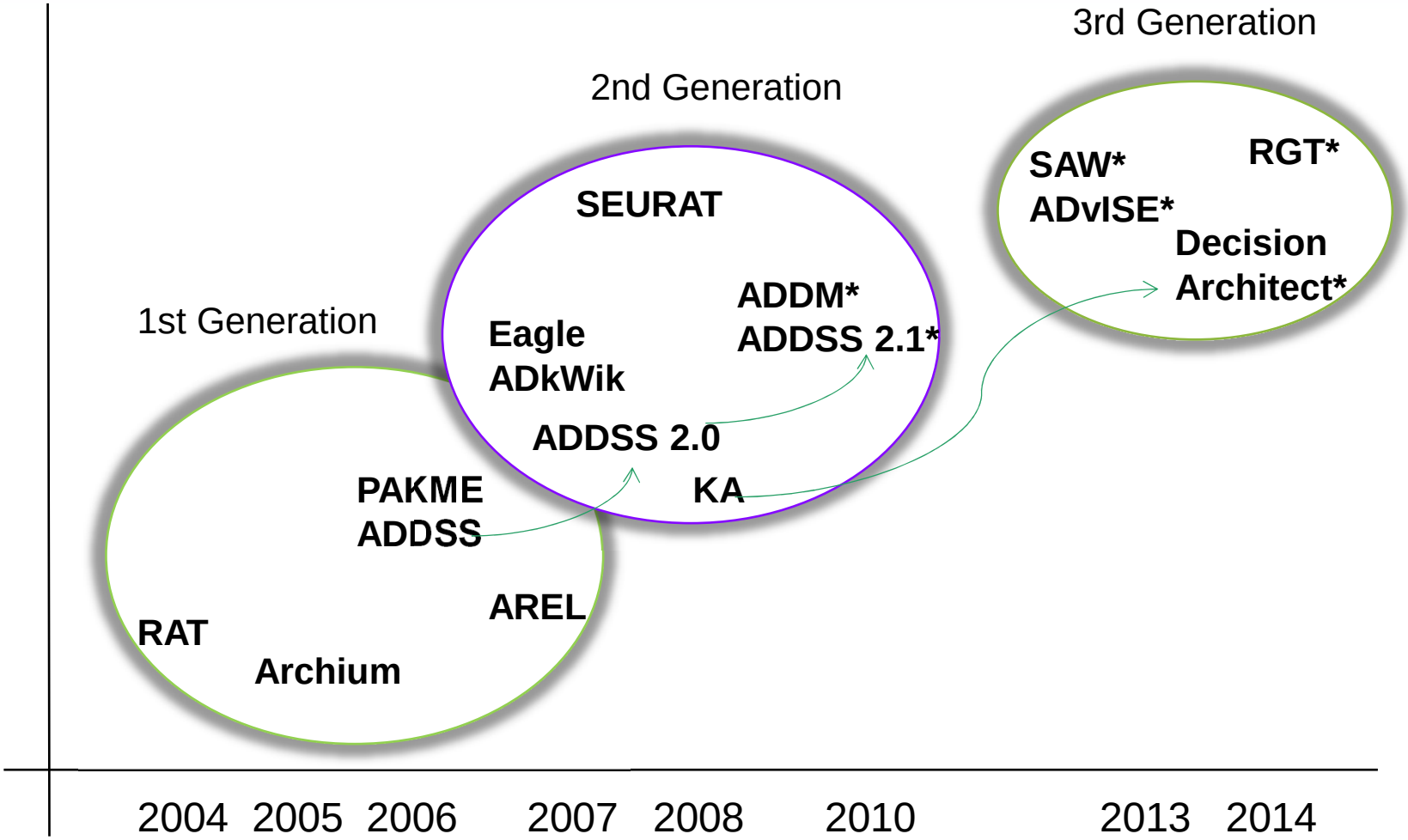
ADDSS: Trazas a Requisitos y DD



ADDSS: Traza a Arquitecturas



Generaciones de Herramientas de AKM



Herramientas de AKM Recientes

- **SAW (Software Architecture Warehouse)**: Herramienta Web para equipos distribuido. Ofrece reutilización de conocimiento y capacidades colaborativas. SAW
- SAW captura, gestiona, comparte y analiza AK reulitizable de múltiples modelos de decision. La compartición de conocimiento es mediante un Wiki con un modelo de decisión difusa.

Herramienta SAW (Nowak, Pautasso)

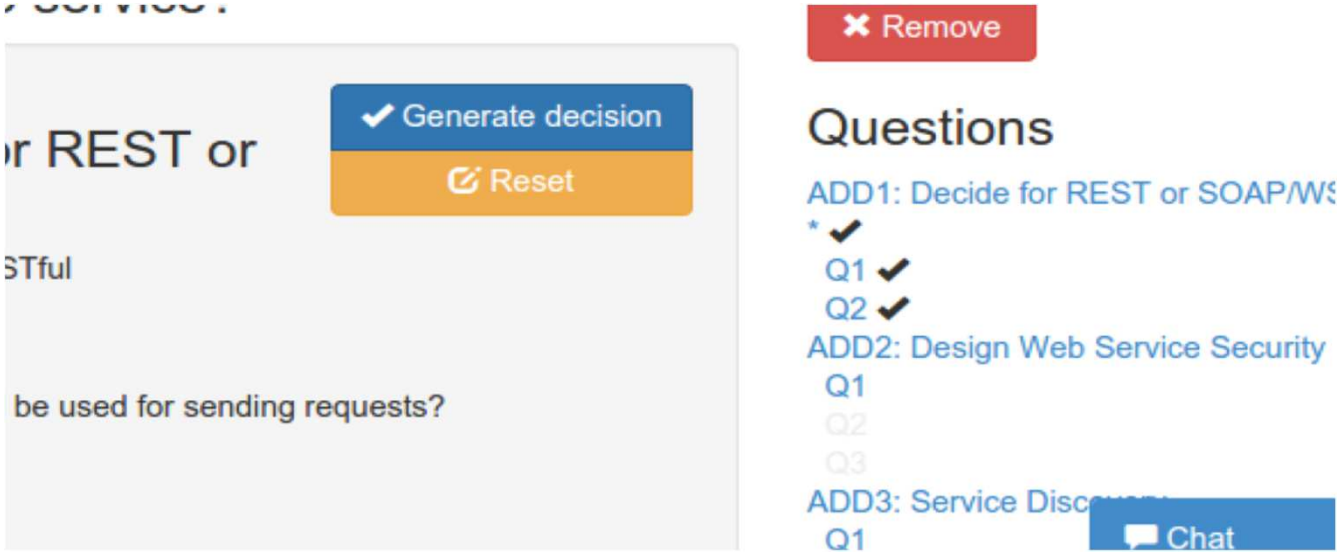
- Captura, gestiona, comparte y analiza AK en formato Wiki
- Búsqueda por texto y navegación por tags
- Gestiona consenso entre decisiones

Herramientas de AKM Recientes

- **ADvISE (Architectural Design Decision Support Framework):** Soporta la captura y representación de ADD mediante el método QOC ("Question-Option-Criteria"). Permite crear decisiones reutlizables con un cierto grado de incertidumbre y generar preguntas basadas en las decisiones tomadas.
- Junto con la herramienta VbMF (View-based Modeling Framework) proporciona una vista de components y conectores (C&C) de la arquitectura y enlaces entre arquitecturas y decisiones.

Herramientas de AKM Recientes

- **CoCoADvISE (Constrainable Collaborative Architectural Design Decision Support Framework):** Proporciona un soporte semi-automático para toma de decisiones y documentación.
- Permite colaboración entre usuarios.
- Utiliza modelos de ADD reutilizables mediante QOC.
- Extiende QOC con dependencias entre opciones y decisiones.



Herramientas de AKM Recientes

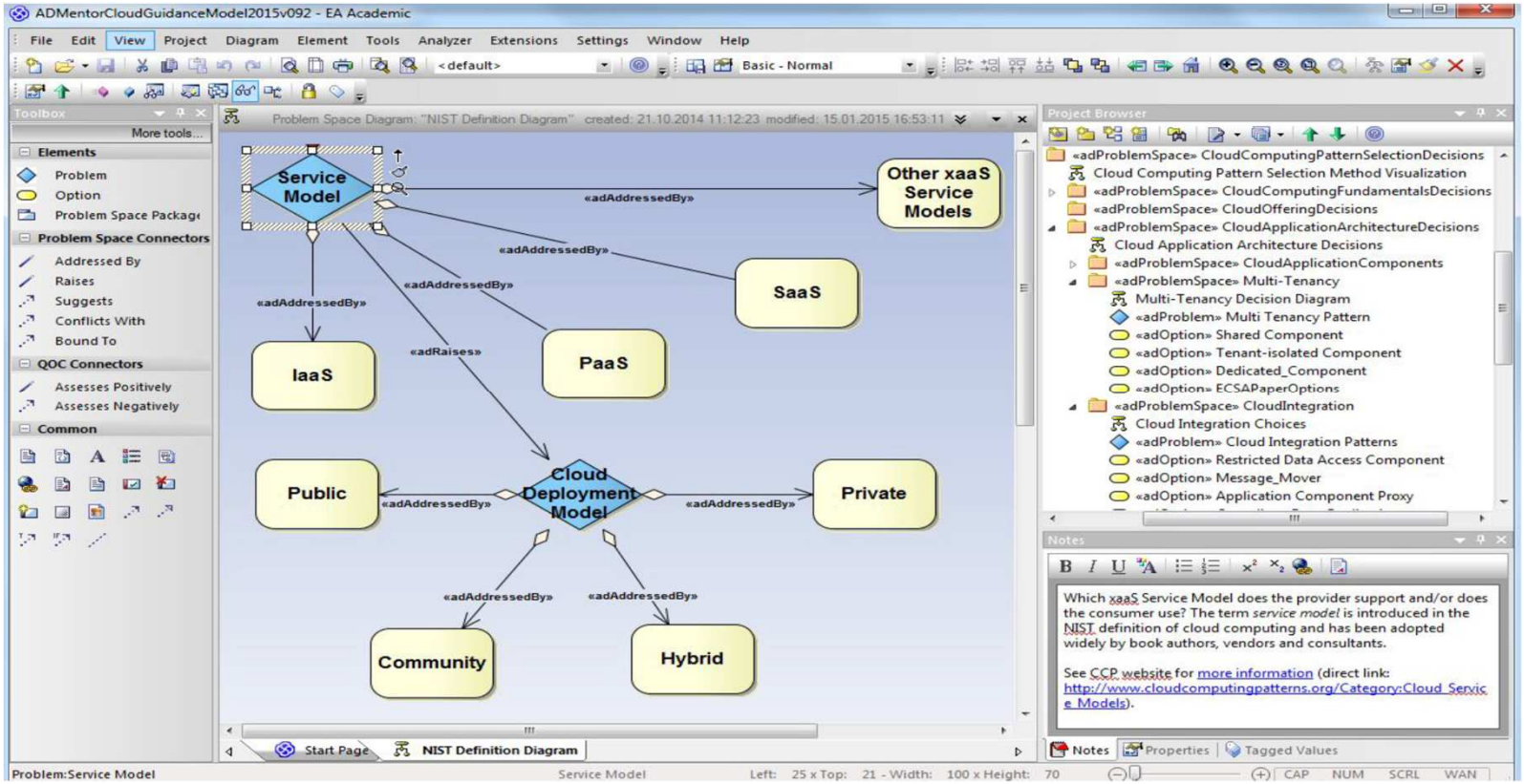
- **RGT(Repertory Grid Tool)**: Permite la captura y análisis de ADDs y proceso de decision grupal de manera que las decisions de priorizan mediante unos pesos según asignando un valor entre 1 y 100.
- RGT soporta una vista cronológica de las decisiones para describer los cambios a lo largo del tiempo.
- Permite establecer dependencias decisiones.

Herramientas de AKM Recientes

- **Decision Architect**: Construida sobre el Sparx Systems' Enterprise Architect (EA), y permite documentar decisiones de una manera fácil y establecer trazas con los elementos de diseño.
- Su modelo conceptual se basa en 5 puntos de vista ("viewpoints" para soportar los diferentes aspectos de los usuarios ("stakeholders") involucrados en el proceso de creación de la arquitectura.
- Integra la documentación de las decisiones en la herramienta como plataforma de modelado.

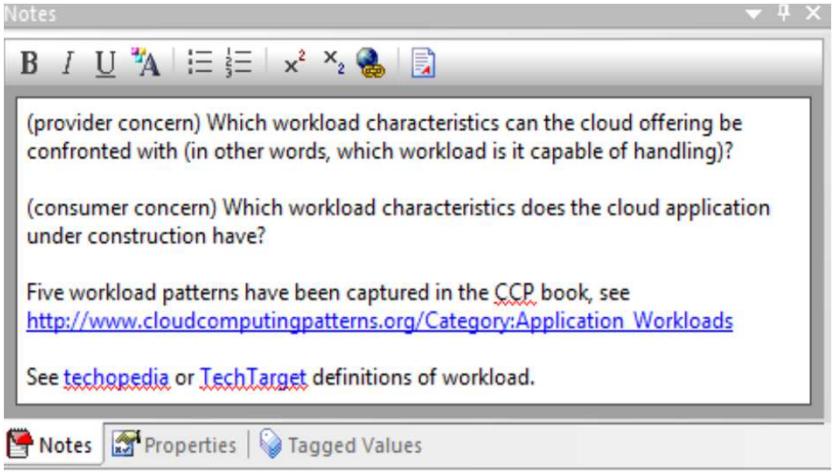
Herramienta ADMentor

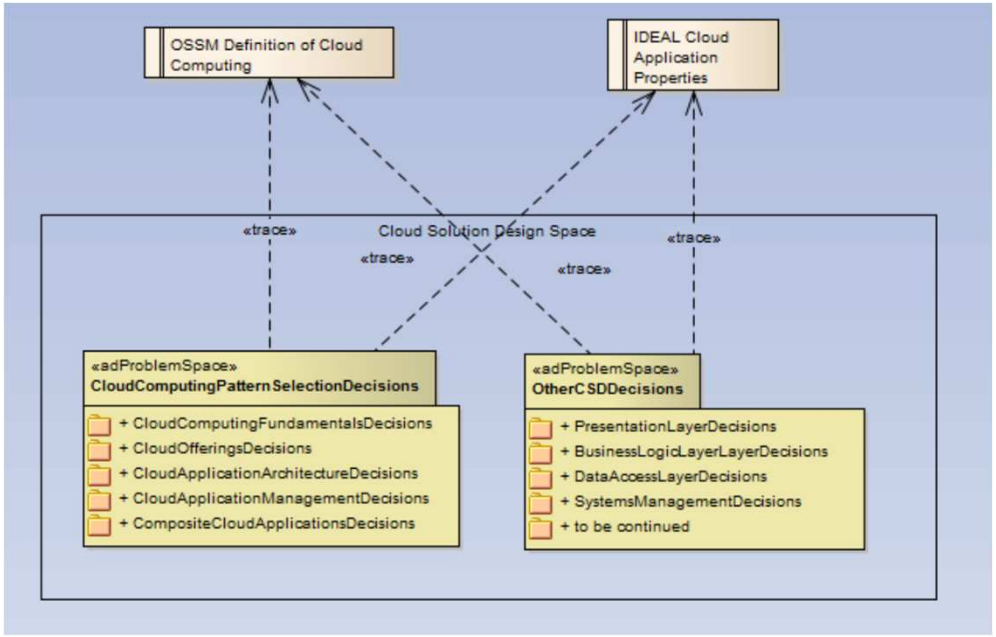
- Sobre Enterprise Architect (EA)
- Enlace a patrones de diseño
- Proporciona analíticas



Herramienta ADMentor

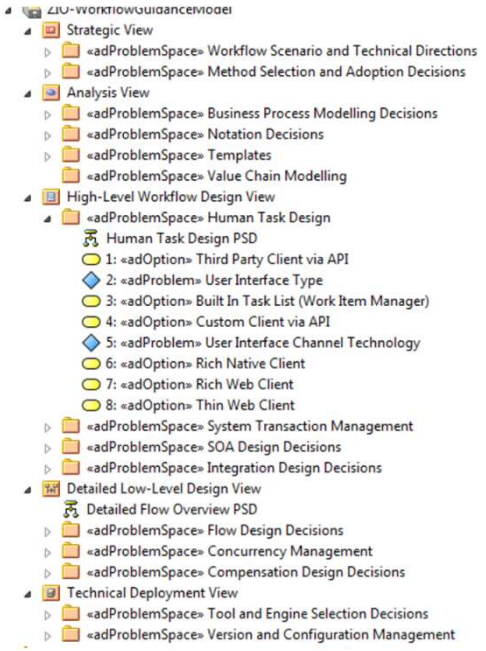
- Texto enriquecido y enlaces Web
- Modelado de AK y herramienta aplicado en la arquitectura de la empresa ABB

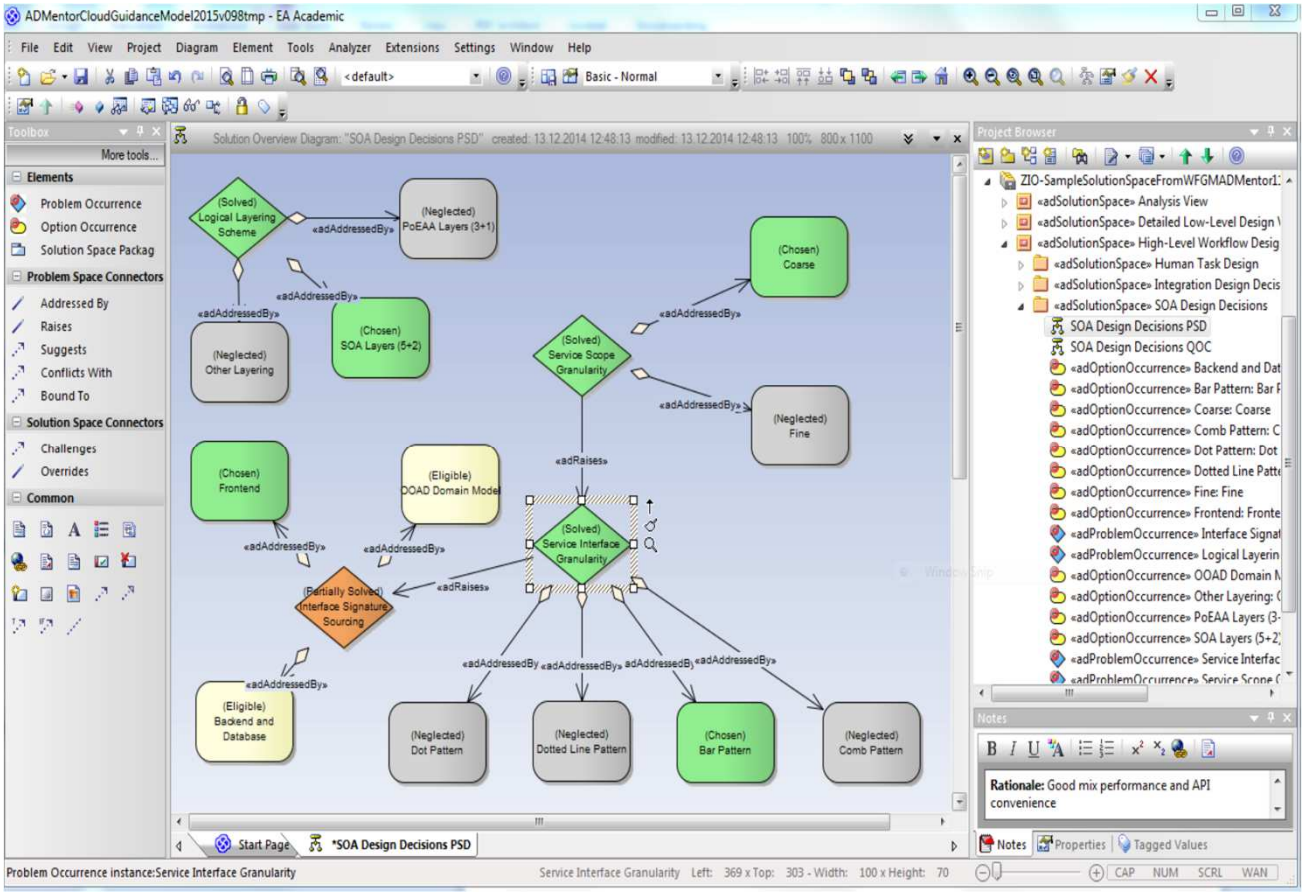




Herramienta ADMentor

- Método QOC (Question-Option-Criteria) para el espacio del diseño
- Interfaz con el espacio de la solución (decisiones tomadas)





Decisiones en Contextos GSD/Ágil/OSS

- Diferentes contextos pueden influir en la toma de decisiones.
- Aspectos como rapidez, esfuerzo, coste o decisiones distribuidas afectan al proceso de toma de decisiones.
- Contextos: Global Software Development (GSD), Ágil, OSS (open source software).

Decisiones en Contextos GSD/Ágil/OSS

- Son útiles aquellas herramientas que permiten la compartición y los aspectos colaborativos entre usuarios para tomar decisiones en un entorno de desarrollo global de software (GSD).
- Herramientas como SAW, ADvISE, RGT, Eagle, Adkwik
- Diseminación y evaluación (Groupware) entre diversos usuarios

Decisiones en Contextos GSD/Ágil/OSS

- Los proyectos con desarrollo software ágil necesitan más rapidez, menos documentación y centrados en personas, por lo que puede ser necesario capturar pocos ítems para las decisiones.
- Capturar menos ítems implica menor esfuerzo y coste en documentación.
- Las decisiones en proyectos ágiles están limitadas por el tiempo asignado a cada Sprint.

Decisiones en Contextos GSD/Ágil/OSS

- Los proyectos con desarrollo software ágil pueden conducir a una intención baja de capturar decisiones debido a conflictos y prioridades con los clientes.
- Más difícil de adaptar el uso de AKM a proyectos ágiles debido a la variedad de métodos (Scrum, FDD, etc.)
- Ejemplo: Feature-driven development (FDD) tiene hitos bien definidos comparado a Scrum y que benefician a la calidad el producto y las decisiones.

Decisiones en Contextos GSD/Ágil/OSS

- Los proyectos OSS son difíciles de gestionar con decisiones de diseño
- Compartición de decisiones es factible en algunos proyectos OSS (E.g, FOSS)
- Problemas para sincronizar las decisiones
- Outsourcing estratégico en proyectos SOA para proporcionar guías de diseño y selección de tecnologías al arquitecto software
- Diseminación y evaluación Groupware

Decisiones en Contextos GSD/Ágil/OSS

ROLES

AK Activity	GSD	Agile	Open Source
Architecting	Groupware decisions	Reduce decision scope/ funneled decision making	Benevolent dictator, voting, commit-sponsor
Sharing	Communication through tele-conferencing, instant messaging, e-mail, video conference	Maximize communication bandwidth among team members. Stand-up meetings, fast iterations.	Disseminate, e-mail lists, forums. Share by example.
Learn	Socialization, internationalization	Agile socialization	Open source internationalization
Evaluation	Groupware evaluation	Evaluate decisions with a limited scope, not strategic ones	Distributed evaluation

Reflection

- **Reflection**: Reflexión, Meditación, Reflectividad?
- Acerca del razonamiento lógico en la toma de decisiones
- ¿Porqué hemos tomado una decisión?
- ¿Es una decisión, buena, mala, adecuada?
- ¿En qué nos hemos basado para tomarla?
- ¿Cuál es el contexto del problema que puede afectar a nuestra decisión?
- Etc…..

Reflection

- Debemos evitar toma de decisiones segadas influenciadas
- Tenemos que contar con toda la información del contexto del problema disponible y otras fuentes de conocimiento
- Hay que hacerse preguntas (Reflection) antes de tomar una decisión
- ¿Porqué escogemos una opción y no otra?

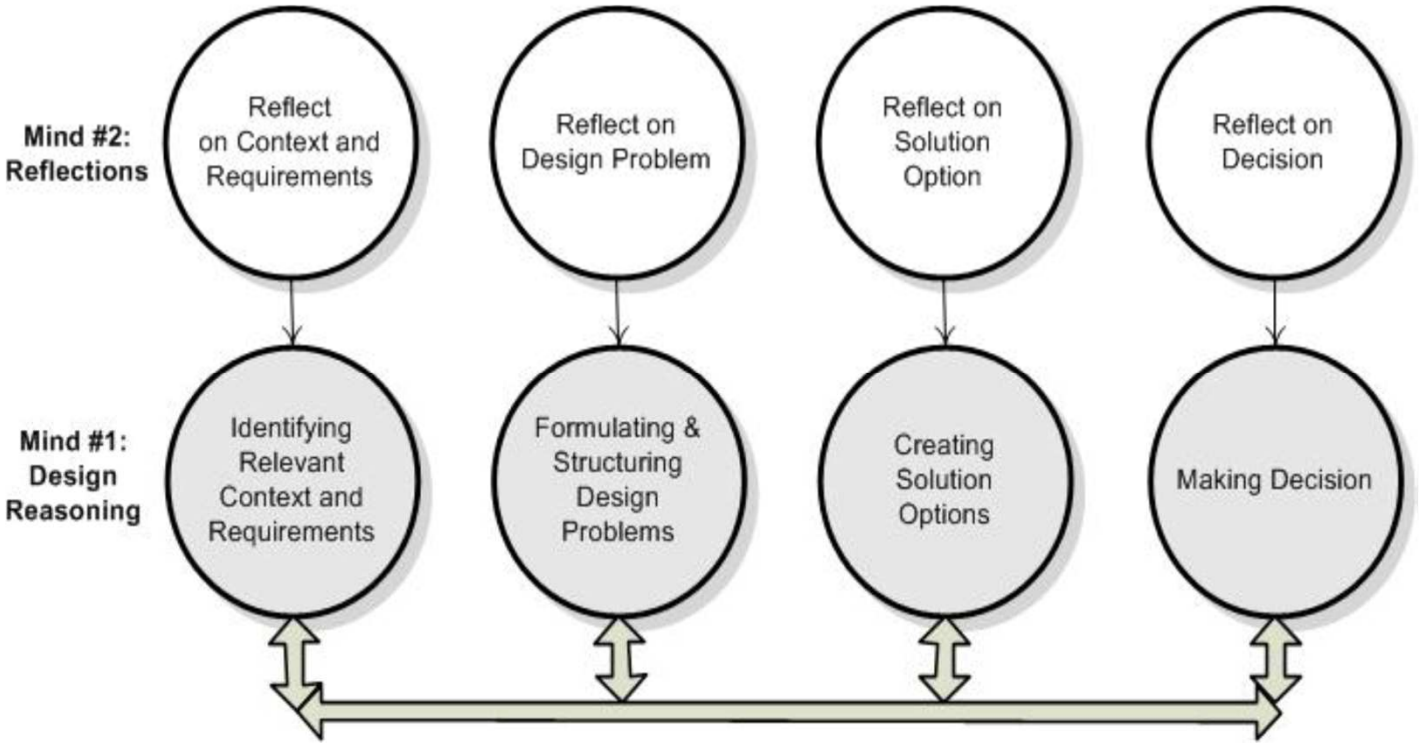
Reflection

- No podemos tomar buenas decisiones si la información es incompleta
- Las decisiones sesgadas conducen a una mala interpretación de los hechos o argumentos
- Es importante contar con hipótesis previas o asunciones
- El pensamiento reflectivo desafía la toma de decisiones
- La reflexión incrementa la calidad de las decisiones tomadas

Reflection

Modelo Mind1/Mind2

Es un modelo en el que las personas que tomas las decisiones (Mind1), éstas son desafiadas y puestas en duda por otras personas (Mind2) para obtener decisiones con más calidad



Reflection

Mind 1

- **Identificar el contexto relevante y requisitos**: Recoger información del contexto de problema, factores que afectan a las DDs y requisitos funcionales y no funcionales.
- **Formular y estructurar problemas de diseño**: El arquitecto software debe plantear los problemas de diseño principales.
- **Plantear soluciones y alternativas**: Cada solución parcial resuelve problemas de diseño. Los problemas y sus soluciones deben co-evolucionar. Cuando se toman decisiones pueden aparecer problemas nuevos.
- **Seleccionar una solución**: Los diseñadores realizan una elección lógica después de analizar y evaluar opciones.

Reflection

Mind 2

- **Reflexionar sobre contextos y requisitos**: Se reflexiona sobre los contextos y requisitos hasta que estos sean completos, relevantes y precisos.
- **Reflexionar sobre problemas de diseño**: Se evalúan si los problemas de diseño están bien articulados y la reflexión debe desafiar a problemas no bien articulados.
- **Reflexionar sobre soluciones de diseño**: Desafía a las soluciones de diseño y sus alternativas (e.g., plantear soluciones sin alternativas, soluciones aceptadas sin requisitos, soluciones que no resuelven ningún requisito).
- **Reflexionar sobre decisiones de diseño**: Se reflexiona sobre la validez de la solución de diseño, los pros y los cons, riesgos, etc.

Conclusiones I

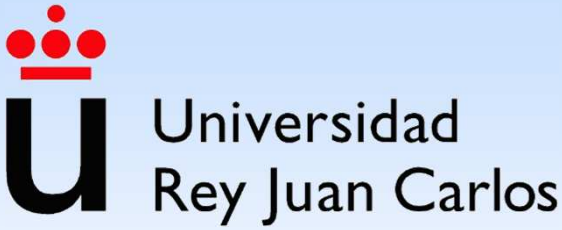
- Las decisiones de diseño ayudan a comprender el porqué del proceso de construcción arquitectónica.
- Permiten cubrir el hueco tradicional existente entre requisitos y diseño.
- Difíciles de administrar cuando el número de decisiones aumenta. Se debe capturar aquellas que sean claves.
- Reducen la necesidad de procesos de ingeniería inversa.
- Permiten determinar responsabilidad y el porqué de decisiones erróneas o sub-óptimas tomadas.
- Permiten conocer cuales fueron las alternativas de diseño.

Conclusiones II

- Múltiples barreras en la adopción de AK
  - Carencia de motivación e incentivos
  - La utilidad no se percibe
  - Carencia de herramientas adecuadas
  - Sobrecarga en la captura de decisiones
- Las ADD retienen el conocimiento
- Útiles en empresas con alta rotación de personal o cuando los creadores de la arquitectura ya no están disponibles.
- La reflexión en las decisiones nos permite evaluar su calidad y el porqué de las decisiones tomadas

Lecturas adicionales

- Herramienta ADMentor: http://www.ifs.hsr.ch/ADMentor-Tool.13201.0.html?&L=4
- Formato MADR: https://adr.github.io/madr/



Grado en Ingeniería Software

Tema 5
Evaluación de Arquitecturas Software

Índice de la Presentación

- **Introducción**
- **Atributos de Calidad**
  - Clasificación de atributos
- **Escenarios**
- **Evaluación de Arquitecturas**
- **ATAM**
- **Conclusiones**

Introducción

- Las arquitecturas software NO sólo son el resultado de responder a un conjunto de Requisitos Funcionales (RF), sino que los Requisitos No Funcionales (RNF) o Atributos de Calidad tienen un impacto directo en la arquitectura final.
- Los RNF determinan la forma final de la arquitectura.

Introducción

- Un RF solo puede ser utilizado una vez en el diseño software y afectar a una parte concreta de la arquitectura.
- Un RNF puede ser usado varias veces en el proceso de diseño y afectar a diversas partes de la arquitectura.
- Los RNF son necesarios para construir un sistema y una arquitectura de calidad.

Introducción

- Es necesario evaluar soluciones de diseño candidatas cuando existen más de una.
- La evaluación de arquitecturas se realiza mediante sus atributos de calidad.
- ADD (Architecture Driven Design) para soportar QA en las fases tempranas de diseño.
- La evaluación es difícil ya que evaluamos un diseño software, no un programa.

Atributos de Calidad

- **Definición de Atributo de Calidad (QA)**: Es un requisito no funcional que mide una propiedad del sistema asociada a su calidad. Representan objetivos de negocio e intereses de los distintos actores que intervienen en el sistema.

Atributos de Calidad

- **Requisitos No Funcionales o Atributos de Calidad**: Nos ayudan a seleccionar entre 2 o más arquitecturas candidatas con funcionalidad igual o similar.

- Igualmente, los podemos aplicar a la selección de otros elementos, como por ejemplo:

  - Selección de plataformas y/o tecnología (Dockers, Kubernetes)
  - Selección de patrones arquitectónicos
  - Selección de servicios (AWS)
  - Selección de hardware

Atributos de Calidad

- Es necesario especificar y priorizar qué atributos de calidad son los más importantes o imprescindibles para el diseño de nuestro sistema.
- La arquitectura no será definitiva hasta satisfacer todos los RF y RNF especificados al inicio.
- Es complicado satisfacer a la vez todos los QA que el cliente desea y puede ser que algunos no puedan satisfacerse

Atributos de Calidad

- **Valores cuantitativos de QA)**: Son valores numéricos o rangos de valores numéricos que se asignan a los QA con el fin de evaluarlos (Prestación de CPU < 3 ms)

  Sino transformamos los aspectos de calidad en valores cuantitativos NO es posible evaluar al arquitectura. Por ejemplo, un cliente puede desear que las consultas a la Base de Datos sean rápidas pero hay que estimar cuanto de rápidas (en segundos o milesegundos) queremos que sean.

Atributos de Calidad

- **Árbol de Utilidad (Utility Tree)**: Organiza y clasifica los atributos de calidad en forma de árbol de la siguiente manera:

- -ATRIBUTO QA X
- - Sub-atributo nivel 1
- - Sub-atributo nivel 1
- - Sub.atributo nivel 2
- - Sub-atributo nivel 3
- - Escenario 3.1
- - Escenario 3.2
- - Sub-atributo nivel 3
- - Escenario 3.3

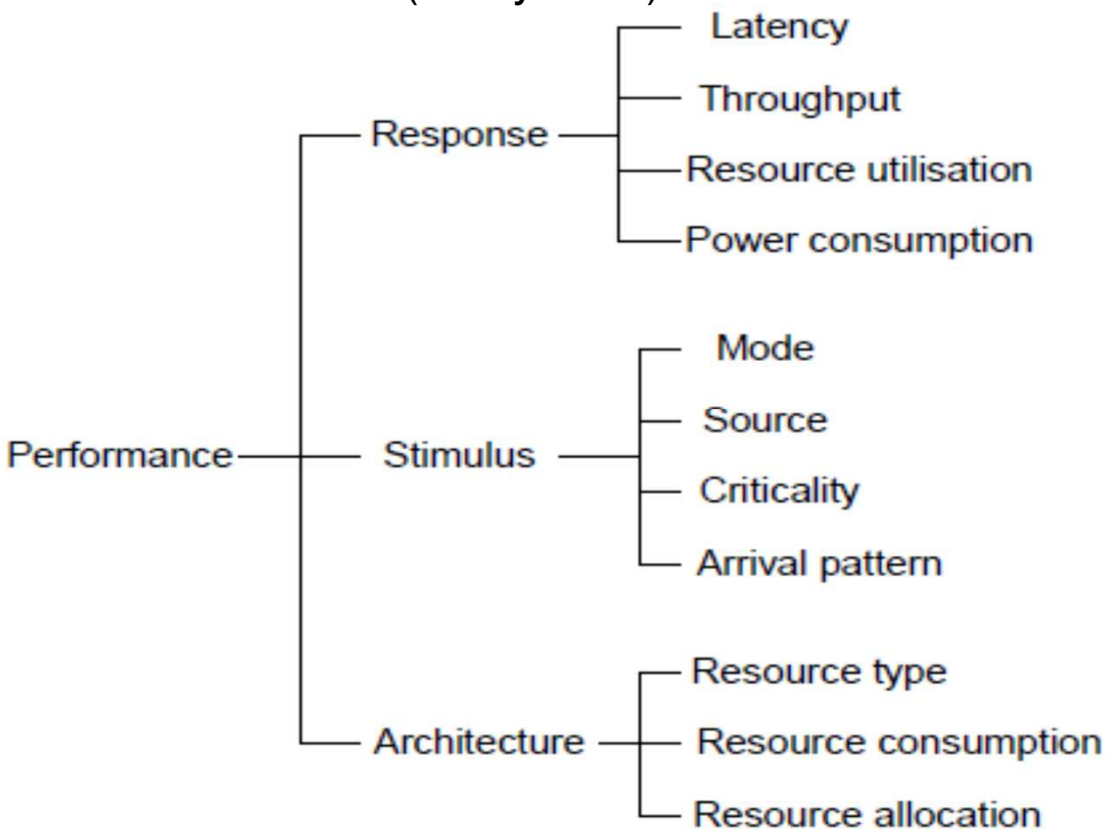
Atributos de Calidad

Taxonomía genérica de QA

- **Runtime**
  - Functionality, Performance, Security, Availability…
- **Non Runtime**
  - Portability, Reusability, Modifiability
- **Business**
  - Cost, Time to market

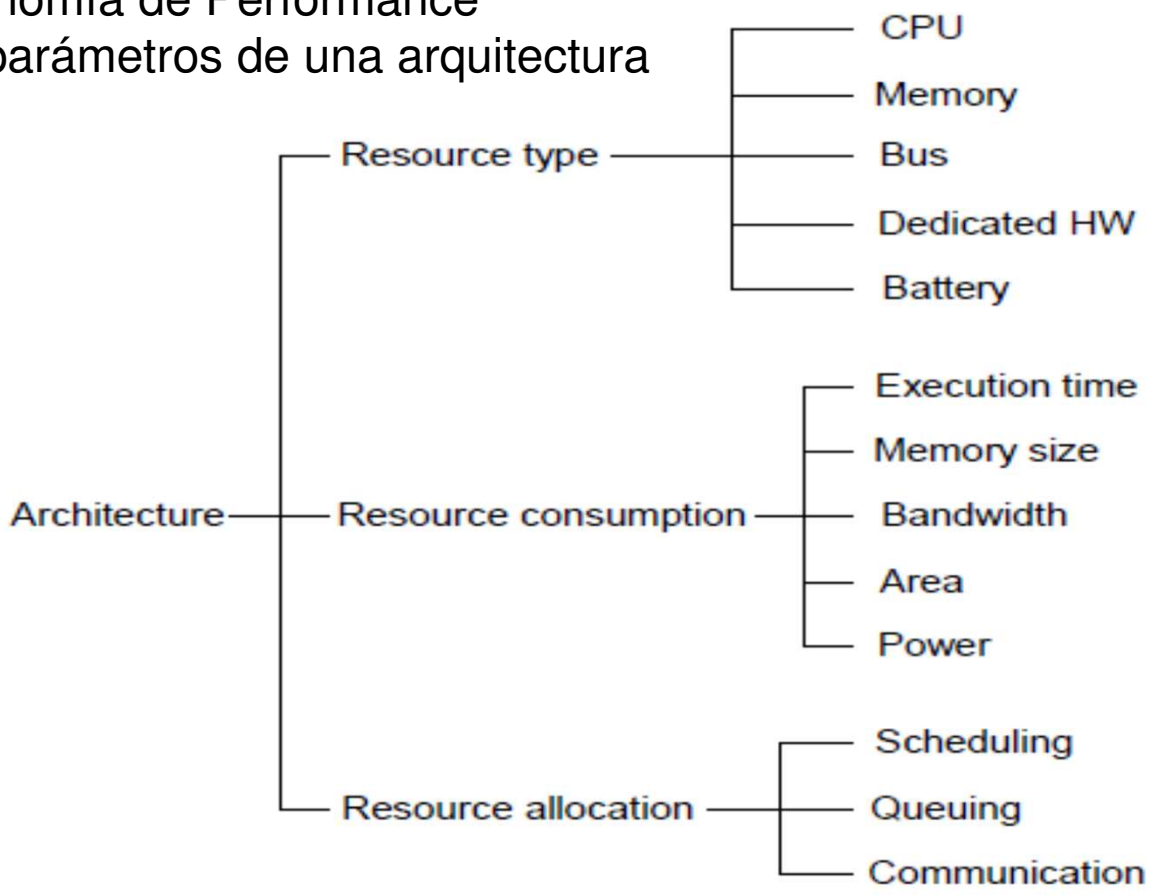
Atributos de Calidad

Taxonomía de Performance (Utility Tree)



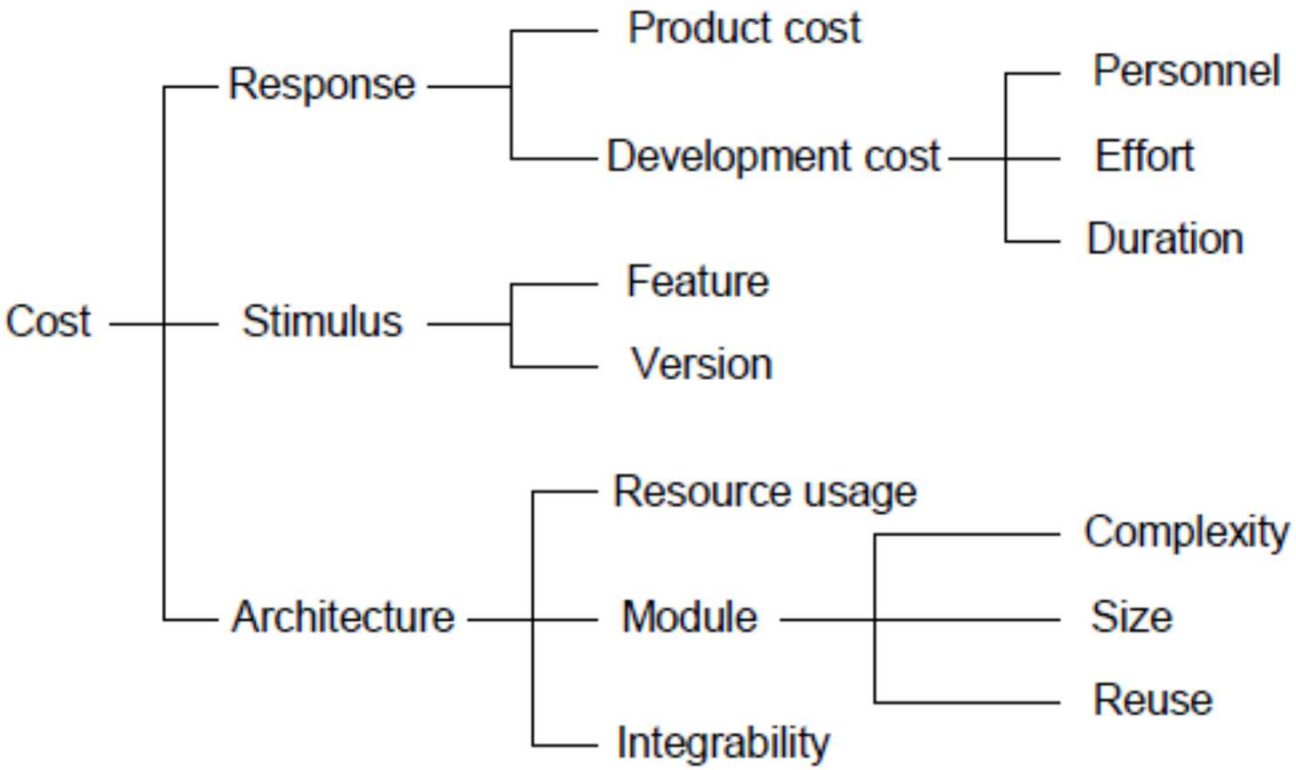
Atributos de Calidad

Taxonomía de Performance
con parámetros de una arquitectura

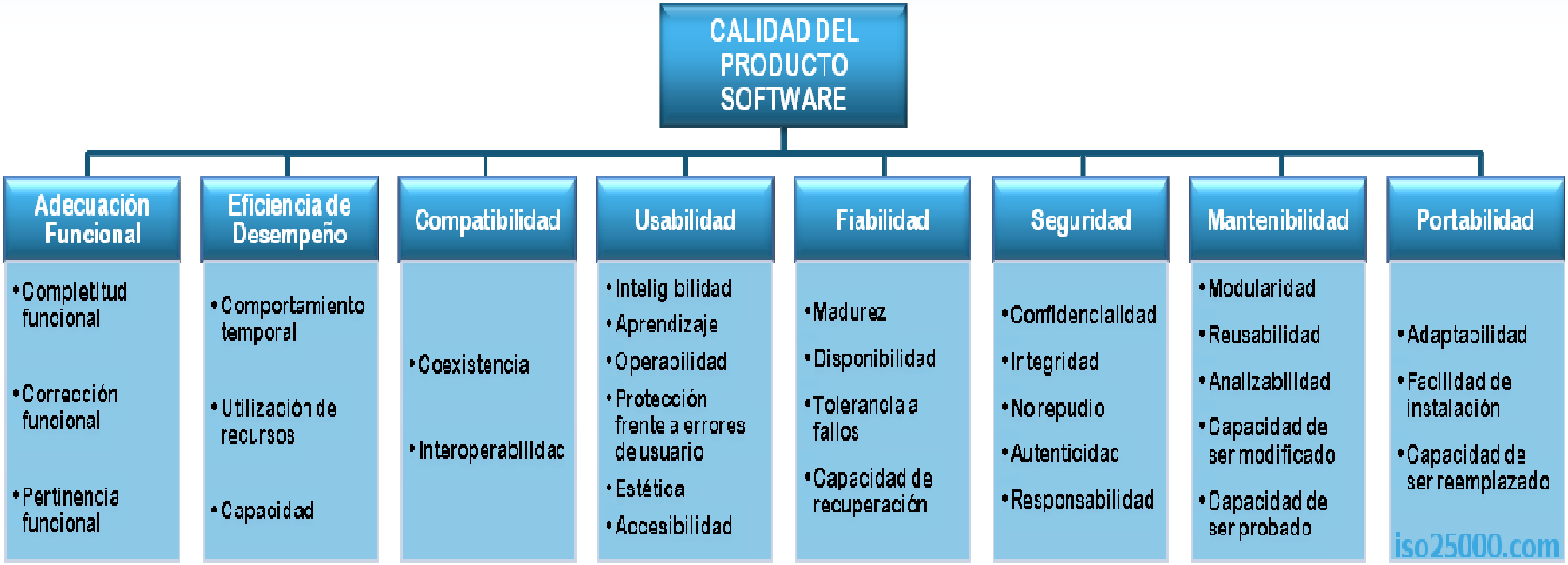


Atributos de Calidad

Taxonomía de Cost



ISO 25010



https://iso25000.com/index.php/normas-iso-25000/iso-25010

Escenarios

- **Concepto de Escenario**: Es una situación posible en la que el usuario interactúa con el sistema y suele ir asociado a la evaluación de atributos de calidad.

- **Escenarios generales**: Independientes del sistema (e.g. Los sistemas transaccionales bancarios soportan sistemas de pago electrónico
- **Escenarios concretos**: Específicos de un sistema particular (e.g. El sistema bancario X debe soportar un sistema de pago via Paypal con disponibilidad 24 h)

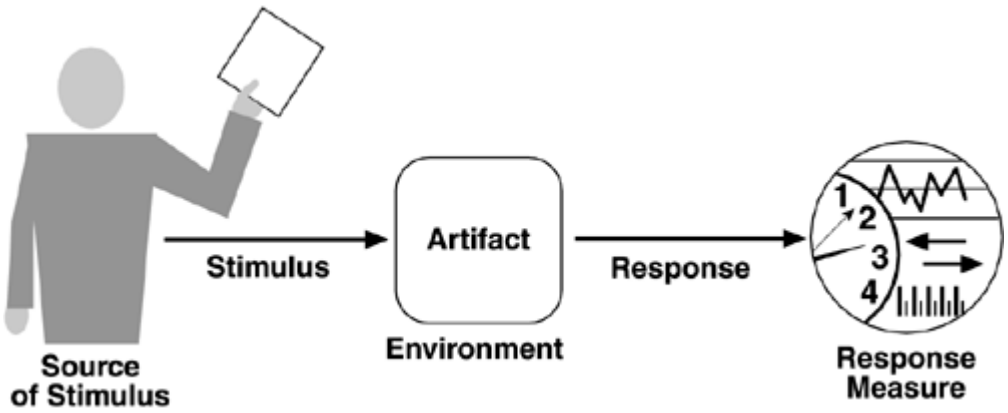
Escenarios

Atributo	Subatributo	Parte del sistema	Escenarios
Performance	Data latency	Base de datos	Maximize storage latency on customer DB to 200 ms
	Transaction throughput		Maximize average throughput to the authentication server
Availability	Hardware failure	Disco SSD	Restart after disk failure in less than 5 minutes

El utility tree lo podemos construir en forma de tabla. Por cada atributo y subatributo (si existe) debemos indicar a que parte del sistema afecta el QA. Para cada atributo y parte del sistema debemos escribir posibles escenarios de uso. En el ejemplo hay 1 sólo escenario pero debemos escribir varios

Escenarios

- Los escenarios concretos es la única manera que tenemos de cuantificar de forma medible los atributos de calidad de un sistema.



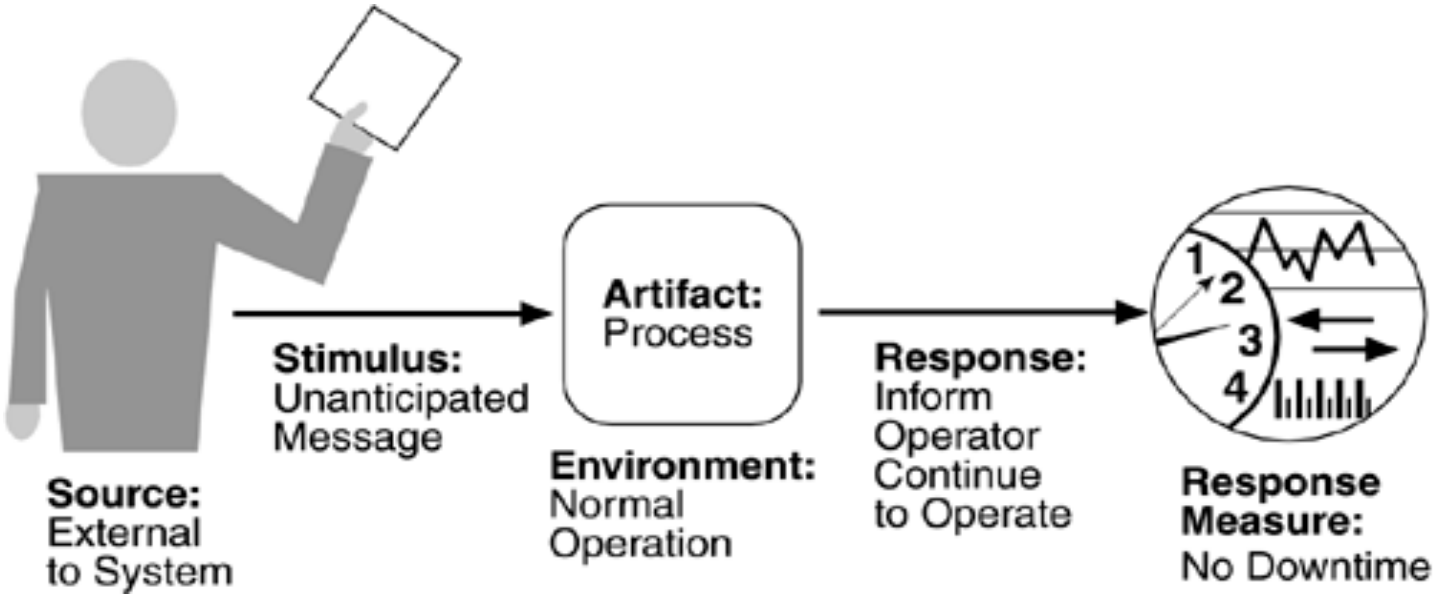
Escenarios

“when 100 users initiate ‘complete payment’ transition, the payment component, under normal circumstances, will process the requests with an average latency of three seconds.”

- Estimulo: Iniciar transacción
- Fuente del estímulo: Usuarios
- Entorno: circunstancias normales
- Elemento afectado: Componente de pago
- Efecto o respuesta: Transacción completada
- Medida de la respuesta: latencia en segundos

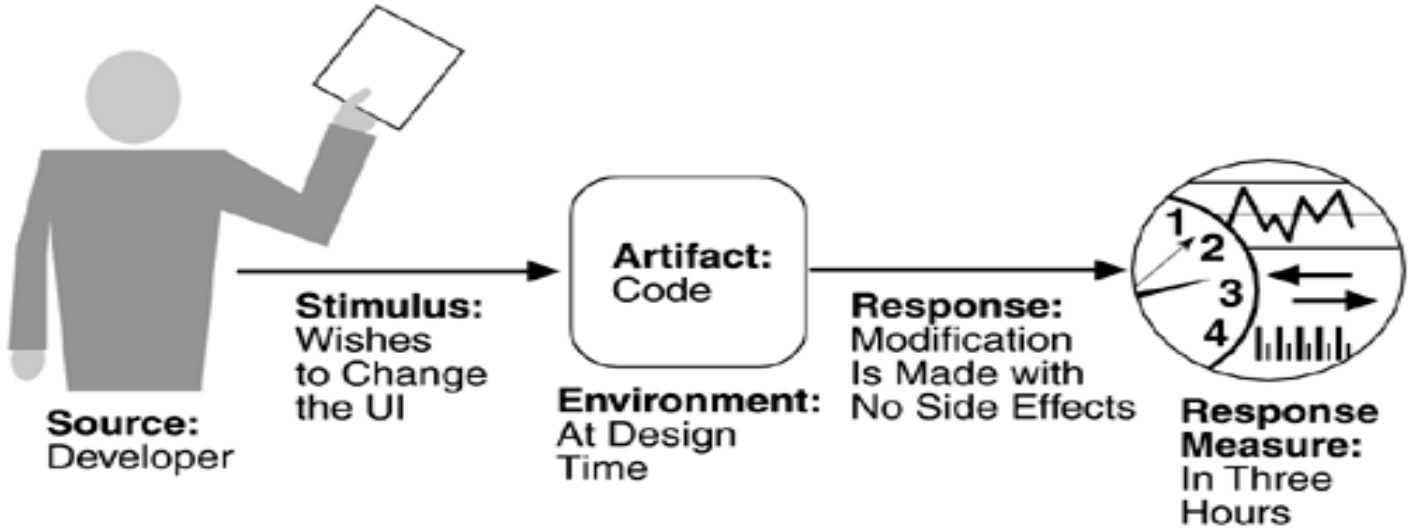
Escenarios

- **Ejemplo de escenario para Availability**



Escenarios

- **Ejemplo de escenario para Modifiability**



Evaluación de Arquitecturas

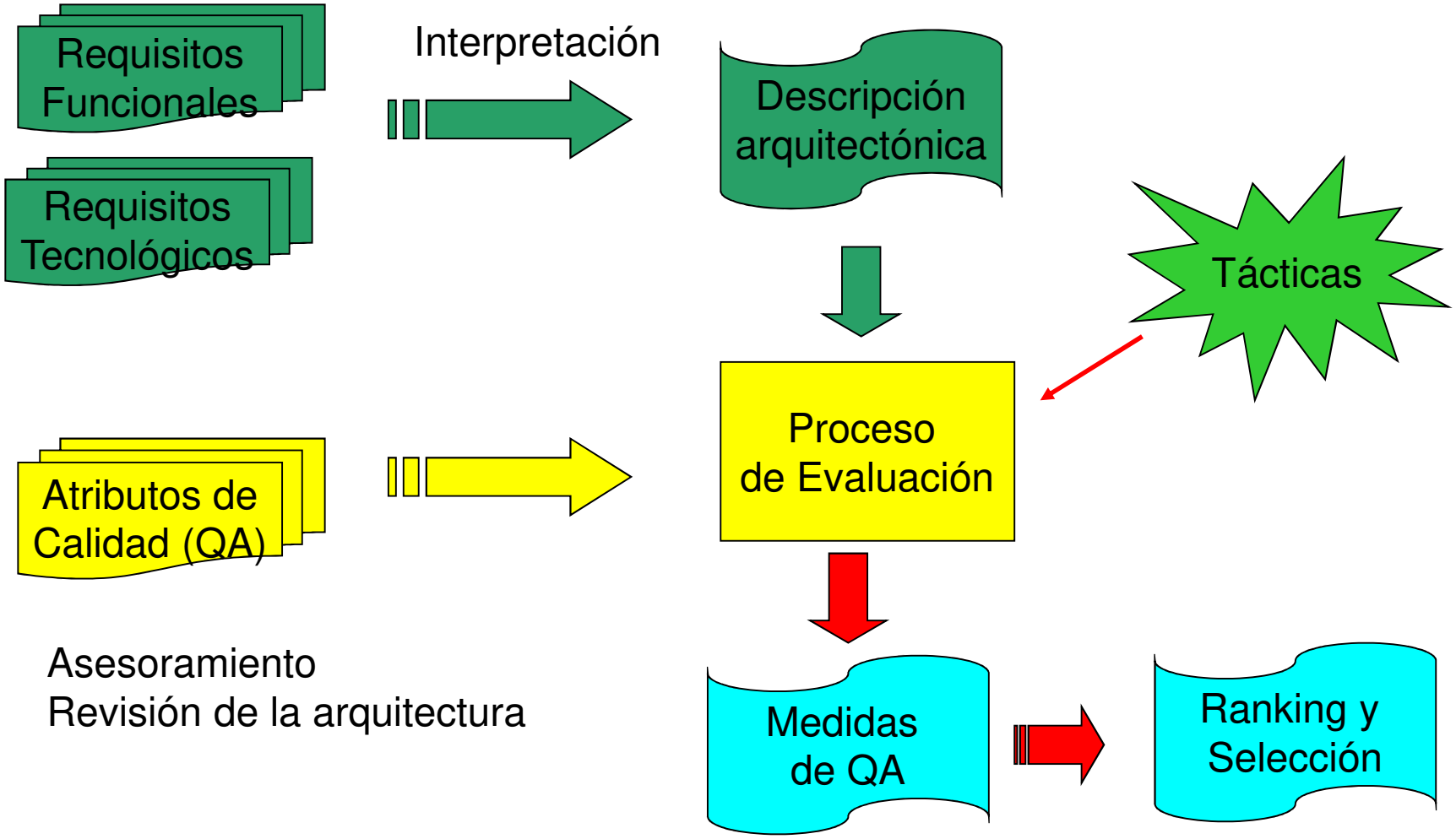
- El objetivo es **evaluar** una o varias soluciones de diseño candidatas para comprobar si satisfacen un conjunto de atributos de calidad.
- Es necesario establecer un **equilibrio** entre atributos dependientes. Por ejemplo, si incremento “throughput” incremento “latency, lo cual no es bueno.
- Hay que priorizar la evaluación de atributos.

Evaluación de Arquitecturas

- **Ranking QA**: Consiste en asignar a cada atributo evaluado y peso y clasificar éstos de acuerdo a un criterio con el fin de tomar una decisión a la hora de seleccionar el elemento arquitectónico candidato mejor o más adecuado.

- **Tácticas de razonamiento**: Son estrategias para satisfacer un conjunto de atributos de calidad. El resultado de una táctica implica una o varias decisiones de diseño para transformar una arquitectura software o para seleccionar un determinado servicio (en SOA).

Evaluación de Arquitecturas



Evaluación de Arquitecturas

Proceso de evaluación y selección

1. Definir QA basados en RNF
2. Construir el Utility Tree descomponiendo los QA en sub-atributos y utilizando algunas de las clasificaciones de QA aceptadas.
3. Definir escenarios de uso para cada atributo y parte del sistema a evaluar.
4. Definir valores admisibles en cada escenario.
5. Priorizar y ordenar los escenarios.
6. Evaluar los escenarios para cada QA (simulación, fórmulas).
7. Utilizar tácticas (no son obligatorias pero deseables)
8. Ordenar los resultados (ranking).
9. Seleccionar los escenarios mejores o más adecuados.
10. Selección de las soluciones de diseño (decisión de diseño) basados en los resultados del proceso de evaluación.

Métodos para Evaluar Arquitecturas

- **ATAM**: Architecture Trade-off Evaluation Methods
- **SAAM**: Software Architecture Evaluation Method
- **ARID**: Active Reviews for Intermediate Design
- **SARA**: Software Architecture Review and Assesment
- **CBAM**: Cost Benefit Analysis Method
- **SBAR, PASA**

ATAM: Architecture Trade-off Evaluation Method

- Utilizado durante años para evaluar arquitecturas software
- Aplicado en diversos dominios: automoción, financiero, defensa
- Objetivo: asesorar sobre las consecuencias de decisiones alternativas en el contexto de atributos de calidad (QAs)
- Es uno de los métodos de evaluación de arquitecturas más utilizados

ATAM: Objetivos

- Descubrir riesgos que pueden crear problemas en Qas
- Descubrir decisiones de diseño sin riesgo que permitan incrementar la calidad del sistema y realizar los objetivos de negocio
- Descubir puntos sensibles ("sensivity points") en los que un cambio pequeño produce una diferencia significativa en un QA
- Descubrir tensiones ("trade-offs") entre Qas que afecten a más de un QA

Ejemplo de sensivity point y trade-off: *el back-up de una base de datos afecta positivamente a la fiabilidad y negativamente a las prestaciones*

ATAM: Beneficios

- Clarificar los requisitos de los atributos de calidad
- Mejorar la documentación de la arquitectura
- Identificar riesgos en las fases tempranas del diseño
- Conseguir una pre-evaluación de calidad de un sistema antes de ser construído

ATAM: Stakeholders (Roles en ATAM)

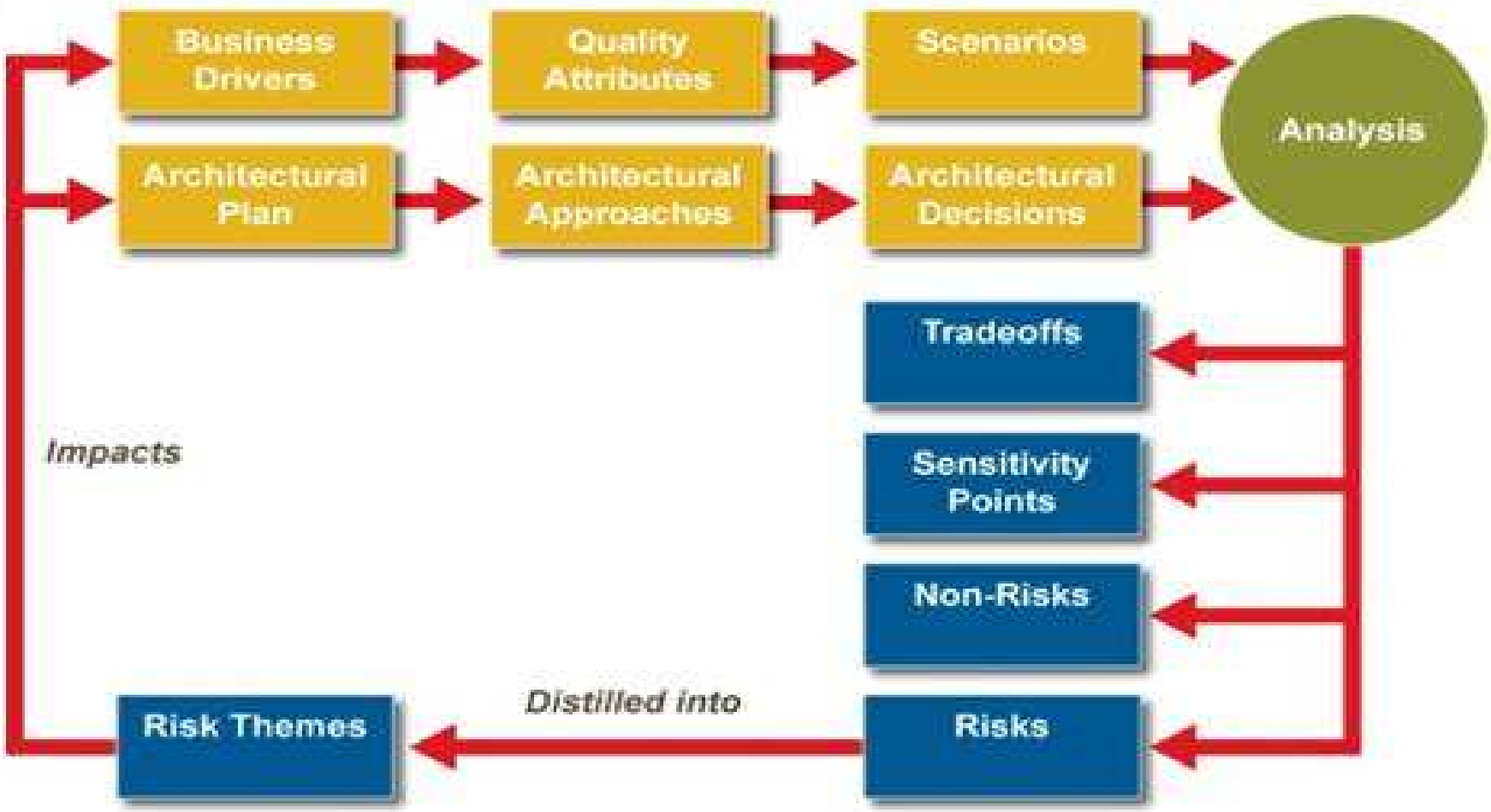
Clientes

Deben presentar los objetivos de calidad del sistema y negocio

Equipo ATAM

Moderador	Modera las discusiones interminables para que el proceso dure 2/3 días
Redactor Escenarios	Parte del equipo ATAM que redacta los posibles escenarios
Controlador tiempo	Controla el tiempo de discusión de los escenarios
Monitor/Observador del proceso	Observa que se cumplen los pasos de ATAM y como mejorar el proceso
Entrevistador usuarios	Lanza preguntas y problemas a los usuarios para determinar qué aspectos de calidad son más importantes

ATAM: Pasos


Business Drivers
Quality Attributes
Scenarios
Analysis
Architectural Plan
Architectural Approaches
Architectural Decisions
Tradeoffs
Impacts
Sensitivity Points
Non-Risks
Distilled into
Risk Themes
Risks

ATAM Fase de presentación: Paso 1

Presentación de ATAM

- El método se presenta a los evaluadores y stakeholders
- Se explican sus fases y duración
- Se identifican los stakeholders que van a participar y con que roles

ATAM Fase de presentación: Paso 2

Objetivos de Negocio

- Describir el contexto de negocio del sistema
- Describir los requisitos funcionales de alto nivel
- Describir los requisitos de calidad de alto nivel
  - Ej: Coste, Prestaciones, Seguridad, etc. A veces no es necesario indicar atributos de calidad sino objetivos de calidad (e.g. deseo que esta parte del sistema X sea seguro)
  - Centrarse en los QA que guían al diseño
  - Requisitos críticos de calidad que son imprescindibles para el éxito del sistema

ATAM Fase de presentación: Paso 3

Presentar la Arquitectura

- Describir la arquitectura existente en términos generales y sus partes funcionales principales
- Describir las restricciones técnicas (HW/SW) y middleware
- Describir otros sistemas con los que interactúa
- Describir los estilos arquitectónicos utilizados para soportar los QA

ATAM Fase de Análisis e Investigación: Paso 4

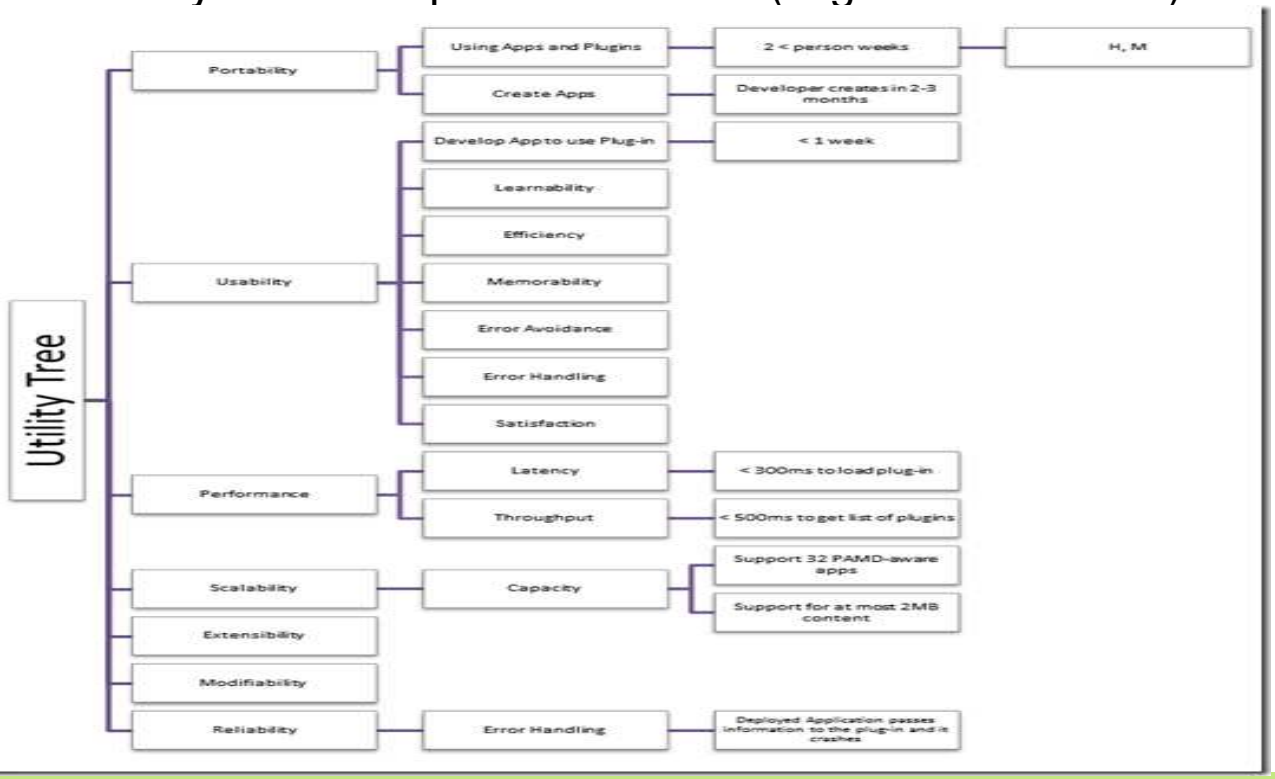
Identificar Aproximaciones Arquitectónicas

- Identificar aquellas partes de la arquitectura que son susceptibles de ser medidas con atributos de calidad
- Identificar los estilos arquitectónicos y patrones de diseño predominantes:
  - Capas
  - Cliente-Servidor
  - Publish-Subscribe
  - Pipe&Filter

ATAM Fase de Análisis e Investigación: Paso 5

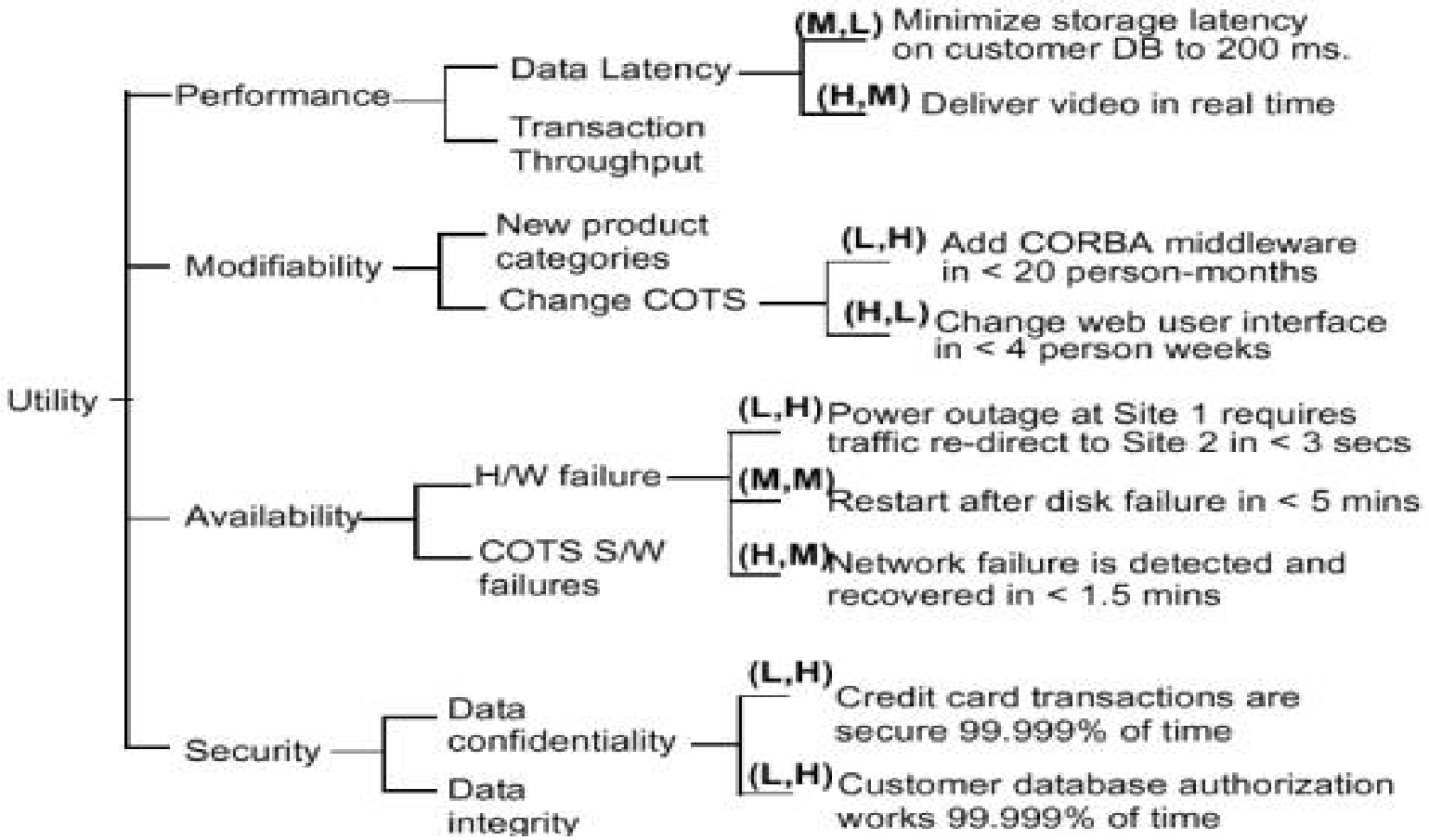
Generar el Utility Tree

- Identificar, priorizar y seleccionar los QA más relevantes (ej: performance, modifiability, security, etc.)
- Describir escenarios (“scenarios”) para cada rama del utility tree
- Salida: caracterizar y priorizar requisitos de calidad específicos
  - Importancia y dificultad para el sistema (High/Medium/Low)



ATAM Fase de Análisis e Investigación: Paso 5

Generar el Utility Tree



ATAM Fase de Análisis e Investigación: Paso 5

Describir los escenarios

- Representan los interés de los *stakeholders*
- Permiten entender los QA
- Cubren un rango de hechos previstos y no previstos
- Anticipan las partes más críticas del sistema
- Son una forma clara de representar el que causa un *estímulo* y su *respuesta*

Ejemplos

Use case scenario: *Remote user requests a database report via the Web during peak period and receives it within 5 seconds.*

Growth scenario: Add a new data server to reduce latency in scenario 1 to 2.5 seconds within 1 person-week.

ATAM Fase de Análisis e Investigación: Paso 5

Escenarios

Estímulo, *Entorno*, *Respuesta*

Ejemplos

Use case scenario: ***Remote user requests a database report via the Web during peak period*** *and receives it* ***within 5 seconds***.

Growth scenario: **Add a new data server** to reduce latency in scenario 1 to 2.5 seconds **within 1 person-week**.

ATAM Fase de Análisis e Investigación: Paso 6

Analizar Aproximaciones Arquitectónicas

El equipo de evaluación prueba las aproximaciones arquitectónicas desde el punto de vista de los QA para:

- Identificar riesgos en la arquitectura
- Presentar arquitecturas alternativas. No hacen falta que sean completas, solo la parte que afecta a un determinado QA
- Identificar elementos arquitectónicos correspondientes a atributos con mayor prioridad
- Identificar y documentar los riesgos de las diferentes alternativas, sensivity points (puntos críticos) y trade-offs (tensiones entre los diferentes QAs)

ATAM Fase de Análisis e Investigación: Paso 6

Preguntas sobre QAs

Intentan probar estilos para identificar y extraer decisiones relacionados con los aspectos de calidad del sistema

Ejemplo con Performance

- ¿Como son las prioridades asignadas a los mensajes para que lleguen más rápido?
- ¿ En cuanto tiempo deben recibirse los mensajes?
- ¿ Cual es el tiempo de respuesta del servidor cuando se reciben 100 peticiones en 5 segundos?

ATAM Fase de Análisis e Investigación: Paso 6

Identificar Riesgos

Riesgo

"Rules for writing business logic modules in the second tier of your 3-tier style are not clearly articulated. This could result in replication of functionality thereby compromising modifiability of the third tier"

No Riesgo

"Assuming message arrival rates of once per second, a processing time of less than 30 ms, and the existence of one higher priority process, a 1 second soft deadline seems reasonable"

ATAM Fase de Análisis e Investigación: Paso 6

Sensitivity Points

Sensitivity point

"Changing the timing scheme from a harmonic framework to a non-harmonic framework would be easy, but due to implied timing dependencies, there would be far reaching impacts to other modules"

No Riesgo

"In order to achieve the required level of performance in the discrete event generation component, assembly language had to be used thereby reducing the portability of this component"

ATAM Fase de Análisis e Investigación: Paso 6

Trade-offs

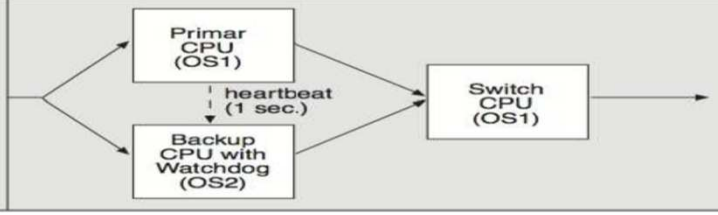
Trade-off

"Si incrementamos los bits para la criptografía (security) para codificar las imágenes de video se reducen las prestaciones del sistema o bien se incrementa el coste (Cost) del hardware"

ATAM Fase de Testing: Paso 7

Discutir y Priorizar Escenarios

- Los “stakeholders” producen escenarios basados en un proceso de discusión (“brainstorming”)
- Los escenarios del utility tree sirven como ejemplo para facilitar este paso y añadir nuevos escenarios
- Cada “stakeholder” dispone de un número de votos (0,3 * #scenarios) para priorizar los escenarios

Scenario #: A12	Scenario: Detect and recover from HW failure of main switch.			
Attribute(s)	Availability			
Environment	Normal operations			
Stimulus	One of the CPUs fails			
Response	0.999999 availability of switch			
Architectural decisions	Sensitivity	Tradeoff	Risk	Nonrisk
Backup CPU(s)	S2		R8	
No backup data channel	S3	T3	R9	
Watchdog	S4			N12
Heartbeat	S5			N13
Failover routing	S6			N14
Reasoning	Ensures no common mode failure by using different hardware and operating system (see Risk 8) Worst-case rollover is accomplished in 4 seconds as computing state takes that long at worst Guaranteed to detect failure within 2 seconds based on rates of heartbeat and watchdog Watchdog is simple and has proved reliable Availability requirement might be at risk due to lack of backup data channel ... (see Risk 9)			
Architecture diagram				

ATAM Fase de Testing: Paso 8

Volver a Analizar Aproximaciones Arquitectónicas

- Los escenarios priorizados sirven de entrada para volver al paso 6
- Propósito: documentar otras aproximaciones arquitectónicas, riesgos y no riesgos, sensivity points y trade-offs
- Si fuera necesario modificar o añadir escenarios hay que volver al paso 5

ATAM Fase de Informes: Paso 9

Presentar Resultados

- Recapitular sobre los pasos de ATAM y proporcionar resultados:
  - Aproximaciones arquitectónicas evaluadas (*arquitecturas candidatas evaluadas*)
  - Utility tree y escenarios
  - Riesgos/No riesgos
  - Sensivity points y tradeoffs
- Decidir entre actualizaciones o “update”

ATAM: Pasos

ATAM Fase I (Pasos 1-6)

- Grupos pequeños
- Generalmente dura 1 día
- Interacción informal entre pasos

ATAM Fase II (Pasos 7-9)

- Grupos más grandes
- Generalmente dura 2 días
- Recapitular y elaborar la documentación

Conclusiones I

- Los atributos de calidad de una arquitectura son imprescindibles para evaluar la calidad del diseño, elegir la mejor alternativa, y obtener un sistema que satisfaga los requisitos de calidad exigidos.
- La evaluación de QA en una arquitectura software debe ser equilibrada.
- Existen tensiones e interacciones entre los QA
- Los QA tienen impacto en la arquitectura software
- ATAM es uno de los diversos métodos existentes para evaluar arquitecturas software y son procesos manuales difíciles de automatizar

Conclusiones II

- La priorización en la selección de atributos es fundamental en las fases tempranas de diseño.
- La definición de escenarios resulta clave para evaluar las distintas situaciones de interacción del usuario con el sistema.
- Los procesos de “brainstorming” son claves para la selección y toma de decisiones finales entre un conjunto de alternativas
- Se recomienda seleccionar pocos atributos de calidad (2-4)

Lecturas adicionales

- Atributos de calidad: http://en.wikipedia.org/wiki/List_of_system_quality_attributes
- QA trade-off (Microsoft): http://msdn.microsoft.com/en-us/library/bb402962.aspx

Es lícita la inclusión en una obra propia de fragmentos de otras ajenas de naturaleza escrita, sonora o audiovisual, así como la de obras aisladas de carácter plástico o fotográfico figurativo, siempre que se trate de obras ya divulgadas y su inclusión se realice a título de cita o para su análisis, comentario o juicio crítico. Tal utilización solo podrá realizarse con fines docentes o de investigación, en la medida justificada por el fin de esa incorporación e indicando la fuente y el nombre del autor de la obra utilizada.