Universidad Nacional de Tucumán

Universidad Nacional de Tucumán
Facultad de Ciencias Exactas y Tecnología

Ingeniería de Software II

Arquitectura de Software

Mg. Héctor A. Valdecantos

Arquitectura de software: definiciones 1/4

La arquitectura del software aporta un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software de un sistema de información.

[Contreras Chavez Estanislao; Stronguilo Leturia Maria Del Pilar]

Arquitectura de software: definiciones 2/4

La arquitectura es la organización fundamental de un sistema implementado en sus componentes, las relaciones entre sí, y con el ambiente, y los principios que guían su diseño y evolución.

[IEEE 1471]

Arquitectura de software: definiciones 3/4

El conjunto de **decisiones significativas** sobre la organización de un sistema de software, la selección de los **elementos estructurales** y sus **interfaces** por las que se compone el sistema, junto con su **comportamiento** tal como se especifica en las **colaboraciones entre esos elementos**, la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente más grandes, y el **estilo arquitectónico** que guía esta organización: estos elementos y sus interfaces, sus colaboraciones y su composición.

[(Kruchten: The Rational Unified Process. También citado en Booch, Rumbaugh y Jacobson: The Unified Modeling Language User Guide, Addison-Wesley, 1999)]

Arquitectura de software: definiciones 4/4

El conjunto de estructuras necesarias para razonar sobre el sistema, que comprende elementos de software, relaciones entre ellos, y propiedads de ambos.

[SEI glossary, Len Bass et al. “Software architecture in practice”, 2013]

Arquitectura de software: definiciones 4/4

El conjunto de estructuras necesarias para razonar sobre el sistema, que comprende elementos de software, relaciones entre ellos, y propiedades de ambos.

[SEI glossary, Len Bass et al. “Software architecture in practice”, 2013]

estructuras

- módulos: determina una estructura de descomposición. Estática. Unidades de implementación, asignables a equipos; diagramas de clases;
- componentes y conectores: determina la forma que los elementos interactúan en runtime. Dinámica. Servicios, unidades compiladas, y conectadas.
- componentes y conectores: relación entre el software y el hardware. Cómo los elementos del software se asignan a la infraestructura que los soporta.

Arquitectura de software: para qué sirve?

- Qué comportamiento es el sistema capaz de realizar?
- Qué interacciones tiene con otros sistemas?
- Como se manejan propiedades como seguridad?

- Provee una base para determinar agendas y resalta qué capacidades son necesarias en el equipo.
- La arquitectura de software debe
  - ser diseñada, analizada, documentada, e implementada usando técnicas conocidas

[Len Bass, p. 20-21 pdf]

Arquitectura de software: para qué sirve?

- Vehículo para la comunicación
  - Reflejar claramente la intensión de diseño
- Propone diagramas para la implementación
  - En el sentido de “arquitecturar” una decisión
- Porta los atributos de calidad del sistema
  - Requerimientos no funcionales
- Provee una base para determinar agendas y resalta qué capacidades son necesarias en el equipo.
- La arquitectura de software debe
  - ser diseñada, analizada, documentada, e implementada usando técnicas conocidas

[Len Bass, p. 20-21 pdf]

Arquitectura de software: para qué sirve?

- Vehículo para la **comunicación**
  - Reflejar claramente la intensión de diseño
- Propone **diagramas** para la implementación
  - En el sentido de “arquitecturar” una decisión
- Porta los **atributos de calidad** del sistema
  - Requerimientos no funcionales
- Provee una **base para planificar**: e.g. atacar riegos, agendas y capacidades necesarias en el equipo
- La arquitectura de software debe
  - ser diseñada, analizada, documentada, e implementada usando técnicas conocidas en una etapa inicial

[Len Bass, p. 20-21 pdf]

Diagrama de arquitectura?

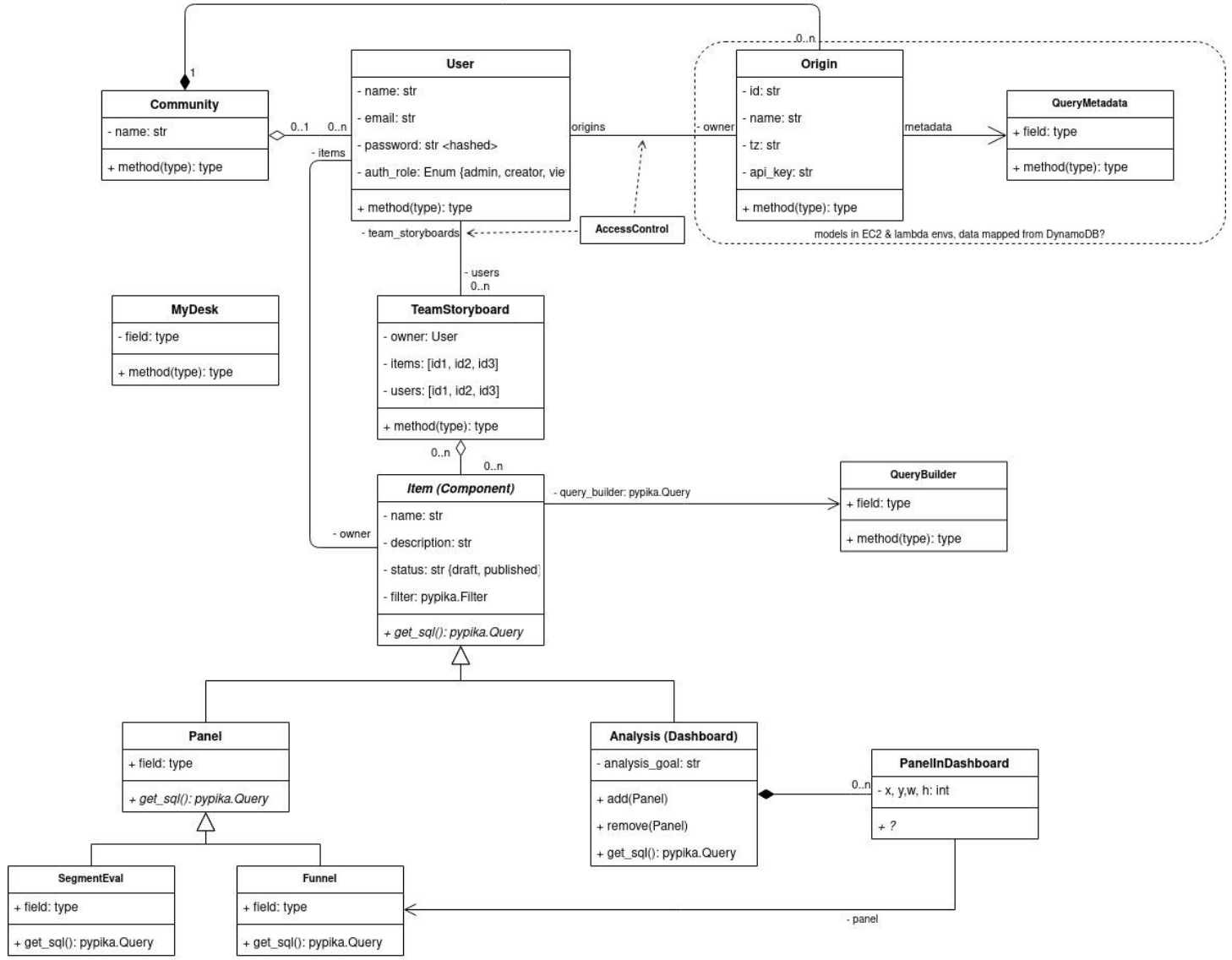


Diagrama de arquitectura?

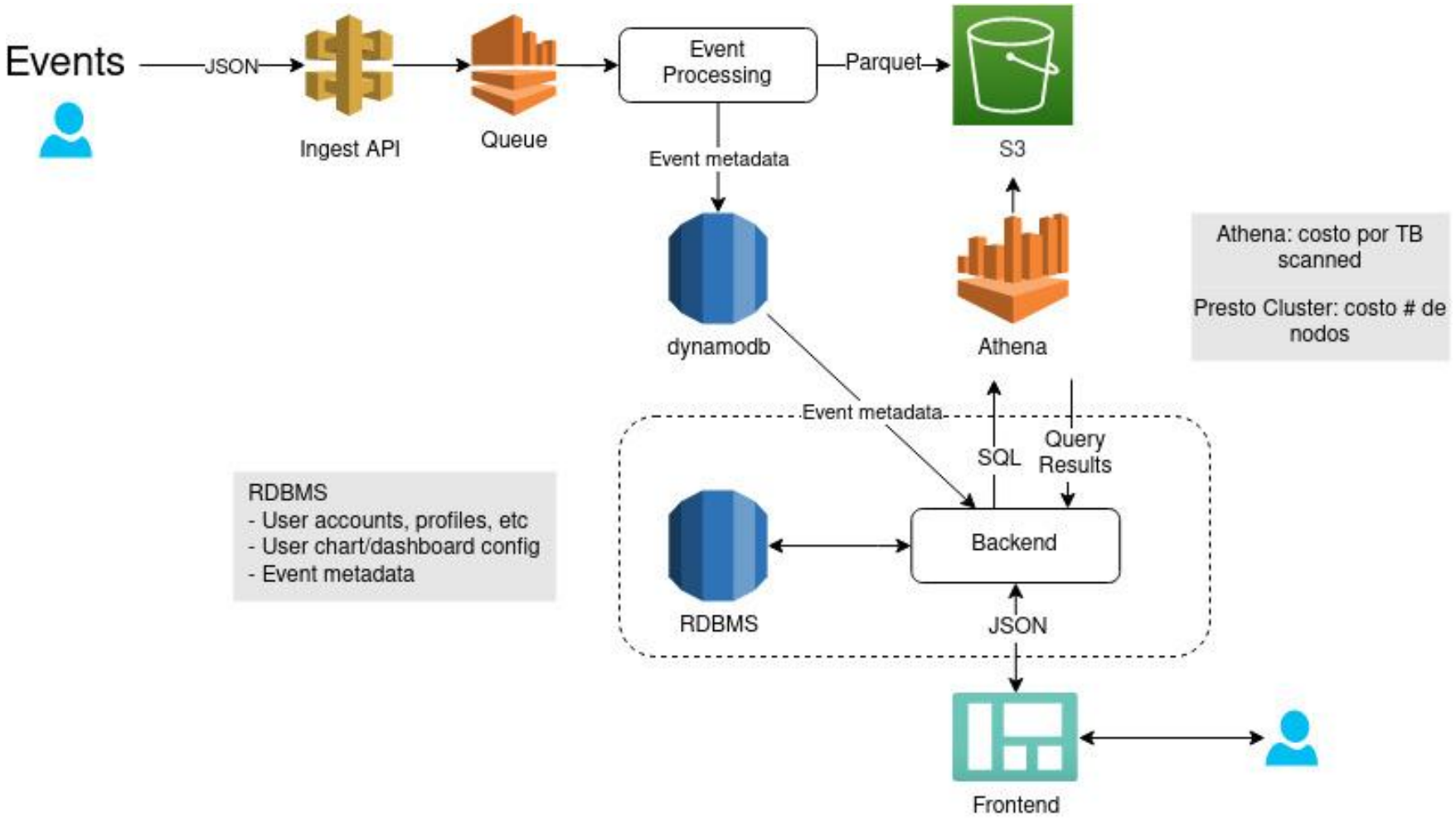
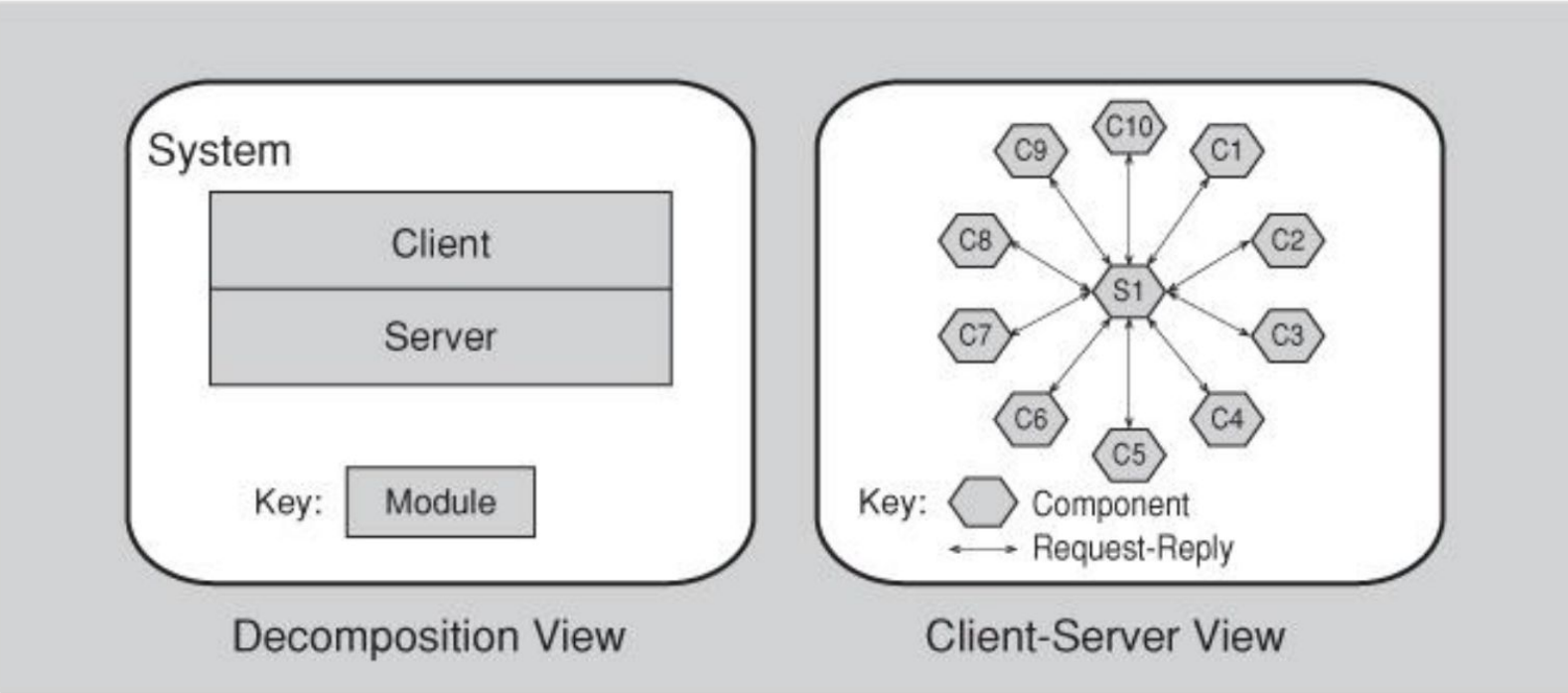
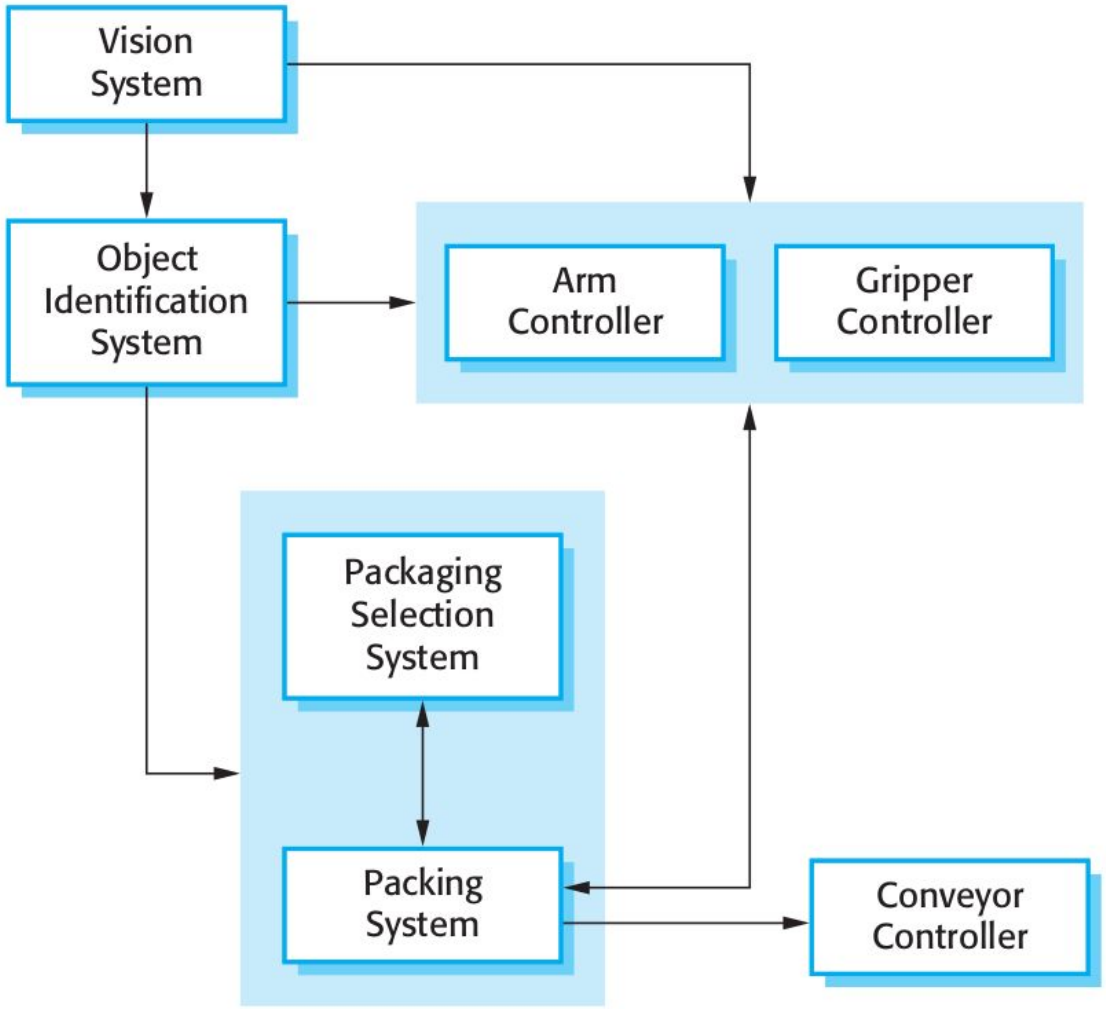


Diagrama de arquitectura?



[Clements, Paul, Rick Kazman, and Len Bass. "Software Architecture in Practice." (2013).]

Diagrama de arquitectura?



[Ian Sommerville. Software Engineering (9th ed.). Addison-Wesley, (2010), p. 149]

Arquitectura de software

CMU - Software Engineering Institute

- Definiciones clásicas
- Definiciones modernas

https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=513807

Arquitectura de software

Who Needs an Architect? by Martin Fowler,

"things that people perceive as hard to change."

https://www.youtube.com/watch?v=DngAZyWMGR0

Arquitectura de software: definición

El conjunto de **estructuras** necesarias para razonar sobre el sistema, que comprende **elementos** de software, **relaciones** entre ellos, y **propiedades** de ambos.

[SEI glossary, Len Bass et al. “Software architecture in practice”, 2013]

- Nos indica qué documentar
- Enfatiza la pluralidad de estructuras en un sistema
- Las estructuras existen para alcanzar un objetivo de negocio
- Propiedades refiere a características como performance, manejo de fallas, recursos compartidos, redundancia, etc
- Evita incluir “early” or “major” design decisions (iterativos-incremental)
- Las estructuras son fáciles de identificar, aunque no toda estructura es arquitectónica.

Estructuras arquitectónicas [Len Bass]

Una estructura es simplemente un conjunto de elementos unidos por una relación. Una estructura arquitectónica ayuda a razonar acerca del sistema y de sus propiedades como un todo. Tipos de estructuras arquitectónicas:

- **Relativa a los módulos**
  - Estructura estática (código, datos), qué es el sistema
  - Estructura de descomposición de módulos, responsabilidades (clases, capas)
  - Áreas con funcionalidades asignadas
  - Refleja división de trabajo para los equipos, unidades de implementación.
  - **Modificabilidad, trazabilidad, comprensibilidad, portabilidad..**
- **Relativa a los componentes y conectores**
  - Estructura dinámica; cómo los elementos interactúan en runtime.
  - incluye componentes (comportamiento): servicios, peers, clientes, filtros) y conectores como vehículo para la comunicación (sync op, pipes, call back, etc)
  - Reflejan las decisiones de **performance, security, paralelismo, availability..**
- **Relativa de asignación o ubicación** (allocation structures)
  - Como el software se relaciona con estructuras de hardware.
  - Organizacional, de desarrollo, de instalación y ambiente de ejecución.
  - **Portabilidad, configuración, performance, seguridad, disponibilidad..**

Estructuras no arquitectónicas

Considerar las estructuras no arquitectónicas inhiben concebir el sistema como un todo y entender sus propiedades.

- Estructura de un sector de código
- Archivos de configuraciones
- Detalles de bajo nivel (La arquitectura omite detalles)
  - Información autocontenida en un solo elemento que no se ramifica a otro.
- Estructuras que no muestren su relaciones con otras estructuras (privadas)

Siempre hay una arquitectura

Todos los sistemas de software poseen elementos y relaciones entre ellos que respaldan algún tipo de razonamiento.

- Dada la intangibilidad del software, se puede considerar que la arquitectura es una abstracción
- Caso trivial: un elemento define una arquitectura.
- Todo sistema define una arquitectura, pero no todas las arquitecturas son conocidas
  - Sin documentación, los arquitectos se fueron, sin código fuente controlado, etc
- Diferencias entre arquitectura y la representación de la arquitectura: la importancia de documentar!

Arquitectura de software

- La arquitectura es una abstracción
- Todo sistema de software tiene una arquitectura
- La arquitectura incluye comportamiento
- No todas las arquitecturas son buenas arquitecturas
- Es el link crítico entre el diseño y los requerimientos
- Sirve para estimar costos (arch de sistema si)
- Resulta útil para planificar tareas y manejar el riesgo

[Len Bass]

Qué debe saber un arquitecto de software

Principalmente tiene que tener experiencia, pero también manejar estas cinco categorías de conocimiento:

- Principios arquitectónicos
  - Lo fundamental… los objetivos de negocio se traducen en atributos de calidad, y estos en diseño arquitectónico.
- Estilos y patrones arquitectónicos
  - Fortalezas y debilidades de Blackboard, Pipes and filters, cliente servidor, Peer-to-peer, MVC, SOA, Microservice, EDA, etc
- Procesos de arquitectura
  - Los Procesos para hacer concretos los principios de diseño.
- Conocer los ambientes donde el sistema corre
  - Nube, on-premise, bare metal, main frame
- Manejar frameworks y herramientas

[Len Bass]

Conocimiento Arquitectura de Software

Categorías de conocimientos para la toma de decisiones arquitecturales

- Principios arquitectónicos
  - Metas de negocios -> requerimientos de calidad -> arquitectura
  - Cuales son las metas de negocio? Cual es el presupuesto?
  - Como los requerimientos de calidad soportar estas metas?
  - Qué diseños de arquitecturas alcanzan los requerimientos de calidad?
- Como expresamos los requerimientos?
  - Las metas de negocio y los requerimientos de calidad deben estar expresados de manera concreta.

Conocimiento Arquitectura de Software

Categorías de conocimientos para la toma de decisiones arquitecturales

- Estilos y patrones arquitectónicos
  - Los sistemas no se diseñan desde cero
  - Los patrones y estilos arquitectónicos detallan:
    - Componentes principales
    - Coordinación entre componentes
    - Patrones de asignación de recursos
  - Elegir los estilos que soportan los atributos de calidad más dominantes:
    - Compensar con otros atributos de calidad
    -

Conocimiento Arquitectura de Software

Categorías de conocimientos para la toma de decisiones arquitecturales

- Procesos de arquitectura
  - Como voy de conocer los requerimientos a obtener el diseño? como descubro mis metas de negocio?
  - Definir procesos:
    - Extraer metas de negocio, mapear metas con atributos de calidad, y atributos de calidad con diseño.

Conocimiento Arquitectura de Software

Categorías de conocimientos para la toma de decisiones arquitecturales

- Conocer los ambientes donde el sistema corre
  - Se necesita saber del negocio y del ambiente de ejecución.
  - Ambiente de negocio:
    - Clientes y proveedores
    - Políticas regulatorias: compliance
    - Leyes, regulaciones gubernamentales
    - Mejoras tecnológicas
    - Mercado, tendencias económicas y tecnológicas
  - Ambiente de ejecución
    - Cloud
    - Embebido
    - Bare-metal
    - Híbrido

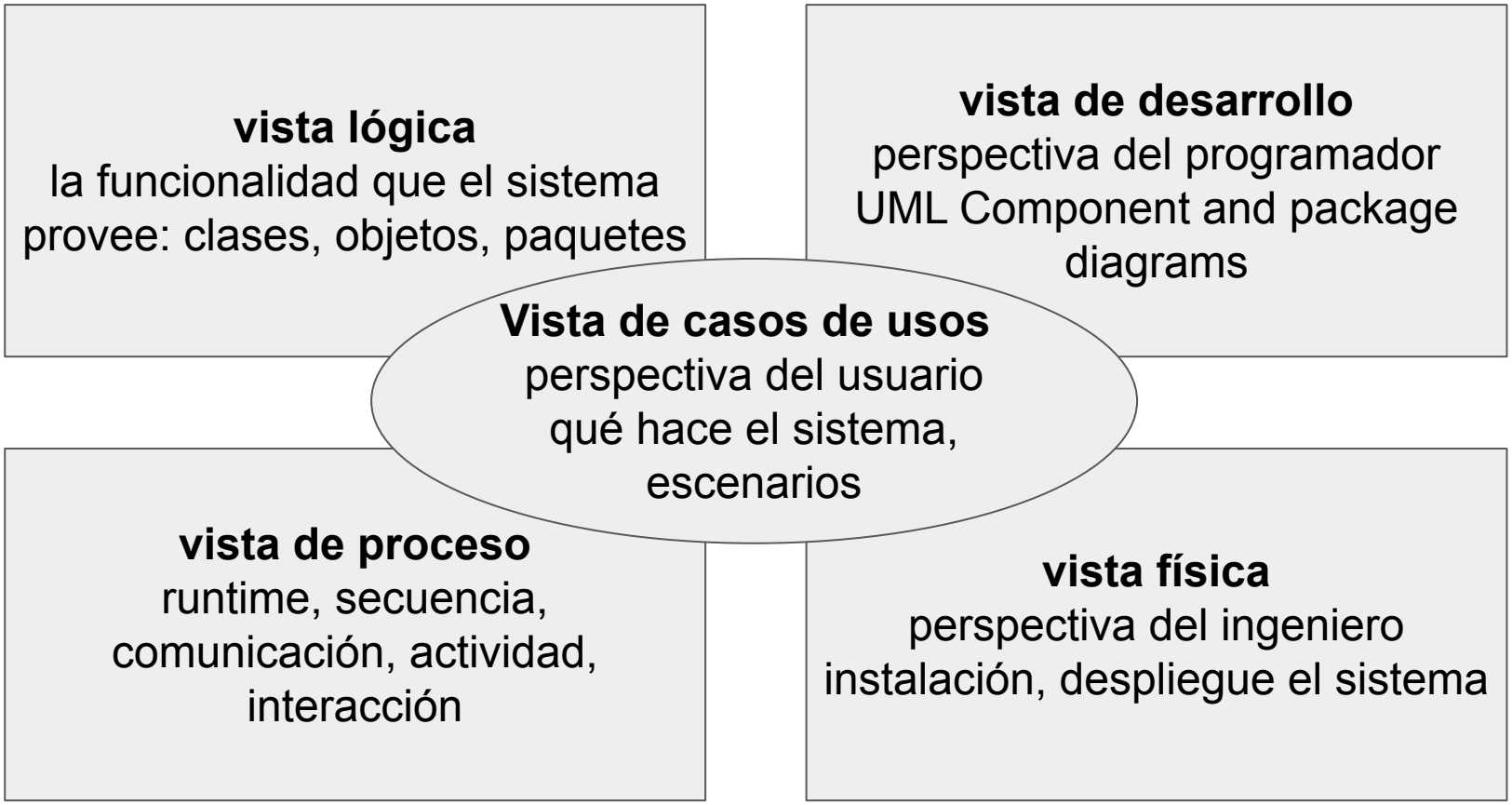
Conocimiento Arquitectura de Software

Categorías de conocimientos para la toma de decisiones arquitecturales

- Manejar frameworks y herramientas
  - Representan decisiones arquitectónicas
  - Miles de herramientas y frameworks para elegir

Arquitectura: 4 + 1 view model

Según el punto de vista de diferentes stakeholders



[Kruchten, Philippe (1995, November). Architectural Blueprints — The “4+1” View Model of Software Architecture. IEEE Software 12 (6), pp. 42-50.]

Arquitectura y requerimientos

Un requerimiento de software es una descripción, un modelo, o una especificación del sistema de software que:

- Es agnóstica del diseño o interfaz de usuario
- Altamente relacionada a un dominio específico

Arquitectura y requerimientos

Entre los tipos de requerimientos encontramos:

- Requerimientos funcionales:
  - lo que el sistema debe hacer
- Requerimientos no funcionales:
  - Modo de cumplir con la funcionalidades
  - Imponen límites al sistema: e.g. seguridad, privacidad, accesibilidad, performance, calidad, conformidad
- Restricciones:
  - Cómo el sistema debe ser desarrollado
  - e.g. Java, Waterfall, Eclipse, Windows

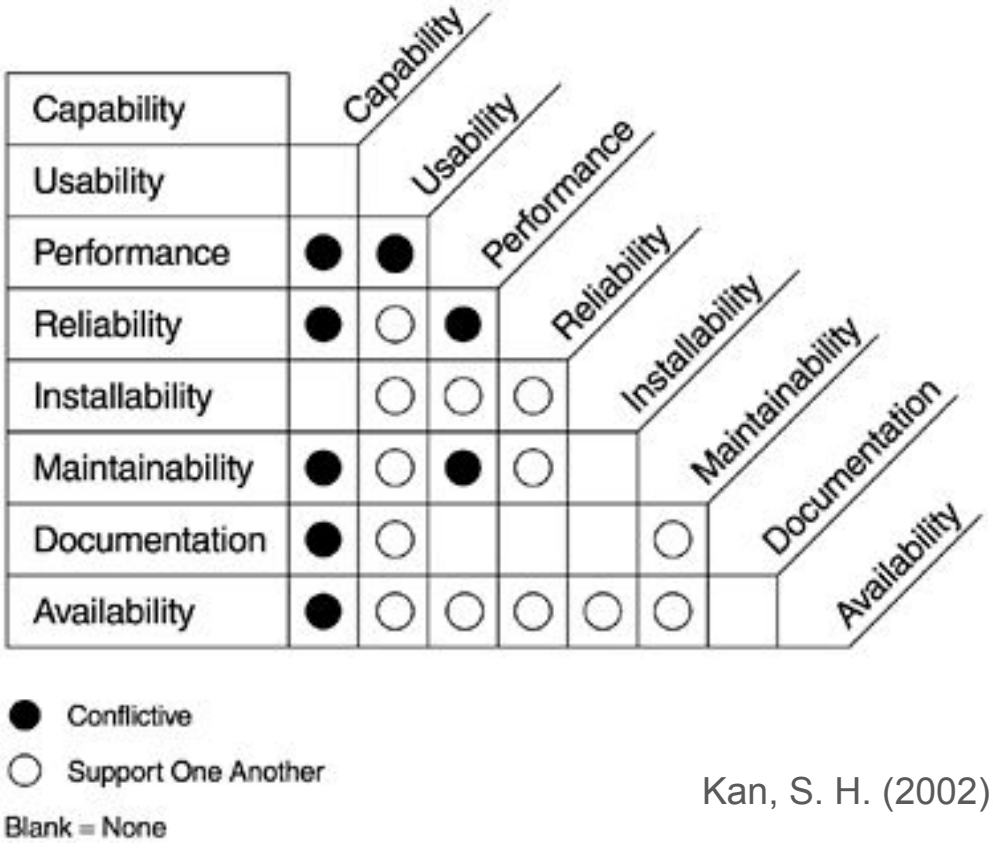
Arquitectura y requerimientos

Qué tipo de requerimientos son los siguientes?

- La página debe cargar en menos de 10 ms
- Se accede a los registros del paciente con un usuario y clave, los registros deben estar encriptados
- El protocolo de comunicación debe ser encriptado usando Transport Layer Security (TLS)
- El sistema debe estar programado en Java

Atributos de calidad

- Requerimientos funcionales
- Requerimientos no-funcionales



Kan, S. H. (2002).

Estilos y patrones de arquitecturas conocidos

- Monolithic application
- Blackboard
- Pipes and filters
- Cliente servidor
- Peer-to-peer
- Tres capas (o arq. en capa en general)
- MVC
- Microservice
- Object request broker
- Event-Driven Architecture

Bibliografía

Clements, Paul, Rick Kazman, and Len Bass. "Software Architecture in Practice." (2013).

Valdecantos, Héctor. “Principios y patrones de diseño de software en torno al patrón compuesto Modelo Vista Controlador para una arquitectura de aplicaciones interactivas”. Capítulo 2. FACET-UNT, (2010).

Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. Documenting Software Architectures: Views and Beyond. Addison Wesley, September (2002).

Kruchten, Philippe (1995, November). Architectural Blueprints — The “4+1” View Model of Software Architecture. IEEE Software 12 (6), pp. 42-50.

Kan, Stephen H. Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc., (2002).

Fowler, Martin. "Who needs an architect?." IEEE Software 20.5 (2003): 11-13.

Pressman, Roger S. Software engineering: a practitioner's approach. 7th edition. Palgrave Macmillan, (2010).

Ian Sommerville. Software Engineering (9th ed.). Addison-Wesley Publishing Company, USA, (2010).

Universidad Nacional de Tucumán

Universidad Nacional de Tucumán
Facultad de Ciencias Exactas y Tecnología

Ingeniería de Software II

Estilos de Arquitectura de Software

Mg. Héctor A. Valdecantos

Event-Driven Architecture

- Los componentes se comunican a través de la producción, detección y reacción a eventos.
- Los eventos se procesan de forma en tiempo real y asincrónica a los consumidores interesados, lo que permite un diseño más reactivo, flexible y escalable.
- Aplicaciones que necesitan manejar grandes volúmenes de cambios con una latencia mínima, como monitoreo, análisis e interacciones con el cliente.
- **Escalabilidad**: puede escalar fácilmente agregando más producers, consumers o brokers de eventos según sea necesario. El procesamiento asincrónico permite que el sistema maneje una gran cantidad de eventos sin generar cuellos de botella.
- **Responsiveness**: el procesamiento en tiempo real permite respuestas rápidas a condiciones cambiantes, como sistemas de monitoreo o servicios en línea.
- **Flexibility**: desacopla los componentes al permitirles comunicarse a través de eventos en lugar de llamadas directas, lo que reduce las dependencias y facilita la modificación o el reemplazo de componentes individuales.
- Complejidad: eventos asincrónicos, la garantía de la coherencia de los eventos y el manejo de fallas. (pérdida o duplicación de eventos)
- Confiabilidad: difícil garantizar la confiabilidad del sistema sin mecanismos extensos de manejo de errores.
- Consistencia de datos: consistencia eventual es común, la consistencia de datos puede demorarse. Difícil la consistencia inmediata o estricta.

N Layer Architecture

- Cada capa es responsable de funciones y preocupaciones específicas. Las capas típicas son: presentación, lógica de negocios, acceso a datos, aunque la cantidad y el propósito de cada capa pueden variar según los requisitos de la aplicación.
- El enfoque en capas fomenta la separación de concerns, donde cada capa tiene una responsabilidad distinta y se comunica solo con las capas adyacentes.
- Promueve la modularidad, la separación de preocupaciones y la capacidad de mantenimiento aislando la funcionalidad de la aplicación en unidades discretas y manejables. Este diseño apunta a simplificar el desarrollo, las pruebas y el mantenimiento, al mismo tiempo que mejora la organización del código.
- **Mantenibilidad**: Dado que cada capa es responsable de una función específica, las actualizaciones o modificaciones se pueden realizar normalmente dentro de una capa sin afectar a las demás.
- **Testability**: Cada capa se puede probar de forma independiente.
- **Reutilización**: La lógica compartida, como las funciones de acceso a datos, se puede aislar en capas dedicadas, lo que permite que otras aplicaciones reutilicen.
- Performance: A medida que las solicitudes pasan por varias capas, esto puede introducir sobrecarga y latencia adicionales.
- Flexibilidad: Las dependencias entre capas crean una estructura rígida, dificulta la adaptación de la arquitectura a nuevos requisitos.
- Escalabilidad: Para escalar, se debe replicar toda la pila de capas o se deben desacoplar capas individuales, lo que puede complicar el diseño.

Broker Pattern Architecture

- Arquitectura de software distribuida en la que los componentes, que normalmente se encuentran en diferentes nodos de red, se comunican a través de un broker que coordina la comunicación.
- El intermediario actúa como un middleware, que maneja las solicitudes de los clientes, las dirige al servidor o servicio adecuado y administra las respuestas de vuelta a los clientes. Este patrón es común en arquitecturas orientadas a servicios y de microservicios, donde facilita la interacción entre componentes independientes y distribuidos.
- El objetivo principal es desacoplar clientes y servicios, lo que permite la escalabilidad y la flexibilidad al permitir que diferentes componentes se comuniquen sin necesidad de dependencias directas.
- **Escalabilidad**: se distribuyen las solicitudes entre varios servicios para administrar la carga. Se pueden agregar o eliminar servicios sin afectar los componentes del cliente.
- **Modularidad**: modulariza naturalmente las aplicaciones, ya que cada servicio o componente se puede desarrollar, implementar y actualizar de forma independiente, lo que promueve un acoplamiento flexible.
- **Interoperabilidad**: al estandarizar la comunicación a través del intermediario, los servicios y clientes heterogéneos, posiblemente escritos en diferentes lenguajes o alojados en diferentes plataformas, pueden comunicarse sin problemas.
-
- Rendimiento: el intermediario introduce latencia adicional, ya que cada solicitud pasa

Object Request Broker (ORB) Architecture

- Plantea un middleware que facilita la comunicación entre objetos distribuidos en una red, independientemente de su ubicación física o del lenguaje de programación. Broker que gestiona la interacción entre objetos clientes y servidor mediante la localización, la llamada y la gestión de instancias de objetos en toda la red. Permite una comunicación transparente y sin fisuras entre componentes distribuidos ocultando las complejidades de la comunicación de red.
- **Interoperabilidad**: objetos implementados en diferentes plataformas, lenguajes, se comunican, permitiendo la integración de componentes heterogéneos.
- **Transparencia**: abstrae las complejidades de la comunicación en red, logra interacción transparente con el objeto remoto como si fuera local.
- **Modularidad**: modulariza los servicios, cada servicio se desarrolla y mantiene de independientemente. Logra un bajo acople de componente.
- Rendimiento: las llamadas de red administradas por el ORB introducen latencia, lo que afecta el rendimiento de las aplicaciones con interacciones remotas de alta frecuencia.
- Complejidad: la implementación del middleware ORB requiere una configuración y una gestión sofisticadas, especialmente cuando se trata de seguridad, tolerancia a fallas y descubrimiento de servicios.
- Escalabilidad: el middleware se ve sobrecargado cuando se escala. Se convertirse en un cuello de botella, necesita redundancia y equilibrio de carga.
- Examples: CORBA, EJB with , Java RMI, Ice (Internet Communications Engine)

Model View Controller Architecture

- Divide una aplicación en tres componentes principales: Modelo, Vista y Controlador. El Modelo representa los datos y la lógica empresarial, la Vista maneja la interfaz de usuario y la visualización, y el Controlador actúa como intermediario, administrando la entrada del usuario y coordinando las actualizaciones entre el Modelo y la Vista.
- El objetivo principal es separar las responsabilidades de una aplicación interactiva
- **Mantenibilidad**: cada componente (modelo, vista, controlador) se puede modificar de forma independiente sin afectar a los demás.
- **Testability**: la lógica de negocio está separada de la interfaz de usuario, lo que simplifica la escritura de pruebas unitarias para las capas del modelo y del controlador.
- **Reusability**: la lógica del modelo y del controlador se puede reutilizar en diferentes vistas o interfaces (por ejemplo, web, móvil), permite la reutilización en distintas plataformas.
- Complejidad: en proyectos pequeños puede introducir una complejidad innecesaria, cada feature puede requerir modificaciones en modelo, vista, controlador.
- Performance: las actualizaciones del modelo a la vista requieren coordinación a través del controlador, lo que puede afectar el rendimiento en aplicaciones en tiempo real.
- Ejemplos: frameworks web ASP.NET MVC y Ruby on Rails, Qt.

Pipe & Filter Architecture

- Los datos fluyen a través de una secuencia de elementos de procesamiento, llamados filtros, conectados por tuberías que transfieren datos entre ellos. Cada filtro realiza una transformación específica en los datos y luego los pasa al siguiente filtro en la secuencia.
- Permite el procesamiento de datos modular, reutilizable y componible desglosando las transformaciones complejas en una serie de etapas simples y aisladas.
- **Modularidad**: cada filtro es autónomo, lo que permite que los filtros se desarrollen, prueben y modifiquen independientemente de otros filtros.
- **Reutilización**: los filtros se pueden reutilizar en diferentes tuberías o aplicaciones, ya que realizan transformaciones discretas y estandarizadas en los datos.
- **Escalabilidad**: Se pueden agregar filtros adicionales al pipeline sin interrumpir los componentes existentes, lo que permite que el sistema se escale a medida que se requieren nuevas transformaciones.
- Performance: Dado que los datos deben pasar por varios filtros, el rendimiento puede verse afectado, especialmente para conjuntos de datos grandes, ya que cada filtro introduce una sobrecarga de procesamiento y una posible latencia.
- Manejo de errores: puede ser complejo, ya que los errores pueden propagarse a través del pipeline antes de ser detectados.
- Flexibilidad: puede dificultar la implementación de flujos de trabajo complejos y no lineales o transformaciones de datos dinámicas.
- Ejemplos: aplicaciones ETL, UNIX shell programs “cat file1 | sort | grep algo”,

Blackboard Architecture

- Para sistemas que requieren resolución colaborativa de problemas. Consta de tres componentes principales: la pizarra, que contiene el estado del problema; las fuentes de conocimiento, que son módulos independientes que aportan conocimientos o soluciones; y un componente de control, que coordina la activación de las fuentes de conocimiento en función del estado de la pizarra.
- Resolver colaborativamente e incrementalmente los problemas. Para problemas complejos y no estructurados que se benefician de diversos enfoques de solución.
- **Modularidad**: cada fuente de conocimiento es un módulo autónomo, lo que permite agregarlas, eliminarlas o actualizarlas de forma independiente sin afectar a las demás.
- **Flexibilidad**: la arquitectura se adapta fácilmente a los cambios o adiciones de nuevas fuentes de conocimiento, lo que la hace adaptable a las estrategias de resolución de problemas en evolución.
- Escalabilidad: se pueden integrar nuevas fuentes de conocimiento a medida que aumenta la complejidad del problema, lo que permite que el sistema maneje espacios de problemas más grandes y complejos.
- Performance: con múltiples fuentes de conocimiento que monitorean y contribuyen a Blackboard, la arquitectura puede sufrir ineficiencias, especialmente si hay una gran cantidad de fuentes de conocimiento e interacciones.
- Complejidad: diseñar el componente de control para activar y administrar de manera eficiente las fuentes de conocimiento puede ser un desafío, especialmente a medida que aumenta la cantidad de fuentes.

Arquitectura en capas

- Desarrollar partes del sistema independientemente
  - separación de intereses y poca interacción entre las partes
- Cada capa es un conjunto de módulos que brinda un conjunto cohesivo de servicios a través de una interfaz pública
- Relaciones de uso unidireccionales entre capas sup-inf.
  - Evitar relaciones circulares
- Portabilidad y modificabilidad
- Diseñar capas estrictas es costoso
  - Un mal diseño es costoso
- Puede degradar la performance
- Típicamente hay tres capas (Presentación: cliente web, Negocio: server web, Persistencia: server BD)
- Protocolo TCP/IP
- **Favorece**: Modularity, Maintainability,Reusability, Testability
- **No favorece**: Performance, scalability, flexibility

Broker pattern

- Servicios distribuidos a través de múltiples servidores
  - Separa usuario de servicios de los proveedores anteponiendo un intermediario (broker, media la comunicación)
  - Brida interfaces de servicios para los usuarios y los brokers se encargan de enviar las peticiones a los servidores
  - El cliente ignora la identidad, lugar, y características del servidor.
- Énfasis en disponibilidad: los servers son dinámicamente elegidos y pueden ser reemplazados por los brokers
- Agrega complejidad e indirección entre el cliente y el servidor
  - Resulta en una latencia (performance)
- Debuguear puede ser complicado porque son ambientes dinámicos
- Los brokers son un punto de ataque (seguridad)
- Si un broker no está bien diseñado puede ser un punto único de fallo
  - O convertirse en un cuello de botella
- Provee una buena modificabilidad, disponibilidad, y performance (los broker pueden actuar como routers)

MVC

- Arquitectura que surge de las aplicaciones interactivas
  - Las GUI son modificadas más frecuentementes
  - Permite mapear el modelo mental del usuario a la vista
  - Multiple vistas de un modelo
- *Un modelo contiene parte de los datos y lógica del sistema*
- *Una vista muestra una parte de los datos*
- *Un controlador media entre la vista y el modelo para notificar cambios de estados en el sistema y eventos de interacción*
- Agrega complejidad
- Favorece la modificabilidad de las vistas
- Logra componentes débilmente acoplados, mejora la testeabilidad
- Rails, Django, ASP.NET, Qt, Spring MVC, Java swing

Tubería y Filtros

- Provee una estructura para procesar streaming de datos
- *Los datos entran a un filtro por un port de entrada, son transformados, y pasados al port de salida y salen por una tubería al próximo filtro.*
- *Los datos fluyen a través de cadenas secuenciales de filtros*
  - *una cadena de filtros representa un sistema*
- No es una buena opción para sistemas interactivos porque no permite ciclos para capturar el feedback de usuarios.
- Sus componentes son genéricos con bajo acoplamiento, lo que favorece la reusabilidad y composición.
- Componentes independientes pueden correr en paralelo
- UNIX shell programs
  - cat file1 | sort | grep algo
- Compiladores:
  - cód. fuente →LEX→SYN→SEM→OPT→CODIGO→ cód. máquina
- Map-reduce pattern, procesamiento de peticiones de Apache web serv.

Peer-to-peer

- Sistema distribuido constituido por entidades de igual importancia que cooperan y colaboran para brindar un servicio a multiple usuarios
- Alta disponibilidad y buen escalabilidad
- Los componentes interactúan directamente, sin intermediarios
- *No hay un componente crítico*
  - *interacción simétrica entre pares*
  - *cada entidad provee y consume servicios similares y usan el mismo protocolo*
- *Cada componente es un cliente y servidor*
- *Hay una red de pares para descubrir pares para interactuar*
- *A veces existen supernodos para indexar la búsqueda de peers*
- Se busca que los pares tengan capacidades solapadas para incrementar la performance del servicio
- La seguridad, consistencia de datos, backup, y recuperación se tornan complejos de manejar.
- Los atributos de calidad mejoran cuando más grande la red
- BitTorrent, eMule, eDonkey, VoIP,

Microservice

- *Servicios pequeños de deploy independiente, que corren en un único procesador que se comunican por un mecanismo liviano*
- *Un conjunto de servicios brindan una meta de negocio*
- Usualmente comunicación HTTP/REST con JSON
- Favorece la modificabilidad y la escalabilidad
- Sistemas pueden ser divididos en múltiples componentes de servicio
- Complejidad en distribuir responsabilidades
- Organizado en capacidades y prioridades de negocio
  - Equipos de desarrollo multi-funcionales
  - "You build it, you run it."
- Microservicios tienen endpoints inteligentes para procesar información y aplicar lógica, y simples pipes por donde fluye la información
- implica una variedad de tecnologías y plataformas
- están diseñados para recuperarse del fallo (alta disponibilidad)
- Netflix, eBay, Amazon

Blackboard

- *Modelo de computación distribuida*
- *sistema que cuenta con dos componentes principales:*
  - *Pizarra: estructura de datos compartida*
  - *Agentes: mantienen la lógica específica del dominio*
- *Los agentes cooperan como especialistas funcionales observando la pizarra por actualizaciones para incorporar en sus procesos.*
- *Los resultados se vuelcan a la pizarra*
- Relevante para el procesamiento de eventos
- Favorece la performance operacional en el uso de la información pero poseen un elemento crítico que hay que tener en cuenta para la seguridad. Concurrencia
- Query optimization problems
-

Bibliografía

Clements, Paul, Rick Kazman, and Len Bass. "Software Architecture in Practice." (2013).

Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. Documenting Software Architectures: Views and Beyond. Addison Wesley, September (2002).

Pressman, Roger S. Software engineering: a practitioner's approach. 7th edition. Palgrave Macmillan, (2010).