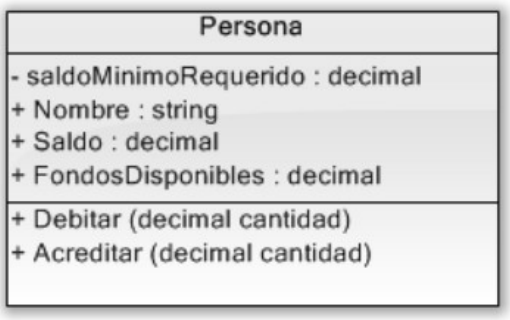
Arquitectura y Diseño de Sistemas – Preguntas sobre teoría

1. ***Explicar el principio SOLID de Responsabilidad Única***

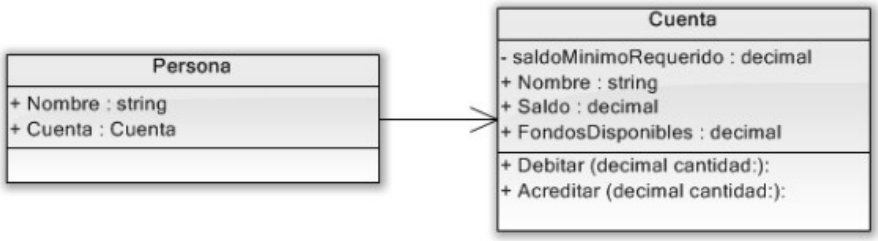
   Single-Responsibility Principle:

   Una clase debe tener una única razón para hacer cambios.

   En el contexto de SRP, definimos responsabilidad como “una razón para hacer cambios”. Si puedes pensar en más de un motivo para hacer un cambio en una clase, entonces esa clase tiene más de una responsabilidad.



- La clase Persona maneja tanto los datos filiatorios de la persona como la información de su saldo de cuenta
- ¿Qué sucedería si ahora se permite que una cuenta sea compartida por más de una persona?
- La clase Cuenta no tiene noción sobre quién la posee.
- Persona puede o bien exponer la propiedad Cuenta, o replicar la interfaz de Cuenta, delegando la implementación de sus métodos



2. ***Explicar el principio SOLID de Abierto/Cerrado***

   Open-Closed Principle

   Las entidades de software (clases, módulos, funciones, etc) deben estar abiertas para extensión pero cerradas para modificación.

   Cuando un solo cambio en un programa resulta en una cascada de cambios de sus módulos dependientes, el diseño huele a rigidez (el diseño es difícil de cambiar). El OCP aconseja refactorizar el sistema para que futuros cambios de ese tipo no causen más modificaciones. Si el OCP está bien aplicado, entonces más cambios de este tipo son logrados añadiendo nuevo código, no cambiando el antiguo código que ya funciona.

   Los módulos que conforman el Open-Closed Principle tienen dos atributos primarios. Son:

   1. Abiertos para extensión

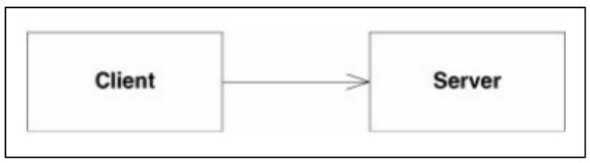
      Esto significa que el comportamiento del módulo puede ser extendido. Cuando los requerimientos de la aplicación cambien, seremos capaces de extender el módulo con

nuevos comportamientos que satisfagan esos cambios. En otras palabras, seremos capaces de cambiar lo que módulo hace.

2. Cerrado para modificación

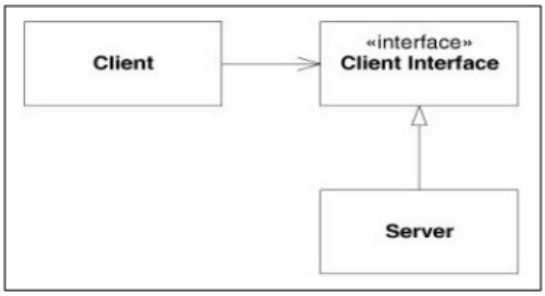
   Extendiendo el comportamiento de un módulo no resulta en cambios al código fuente o los binarios del módulo. La versión ejecutable del módulo, ya sea una librería enlazable, una DLL, o un .jar de Java, permanece sin cambios.

¿Cómo es posible que el comportamiento de un módulo sera modificado sin cambiar su código fuente? ¿Cómo podemos cambiar lo que hace el módulo sin hacer cambios en el módulo? Esto se logra gracias a la abstracción. Es posible crear abstracciones que son fijas y sin embargo representan un grupo ilimitado de posibles comportamientos. Las abstracciones son clases abstractas base y el grupo ilimitado de posibles comportamientos son representados por todas las clases derivadas posibles.



No soporta OCP

- Las clases Client y Server son clases concretas.
- Si por algún motivo la clase o implementación del Server es modificada, entonces la clase Client también debe ser modificada.



Soporta OCP

- Se agrega una interfaz intermedia, ClientInterface, entre Client y Server.
- Si, por algún motivo, la implementación del servidor cambia, el cliente probablemente no requiera cambios.
- La clase ClientInterface es cerrada para modificación aunque si está abierto para extensión.

3. *Explicar el principio SOLID de Sustitución de Liskov*

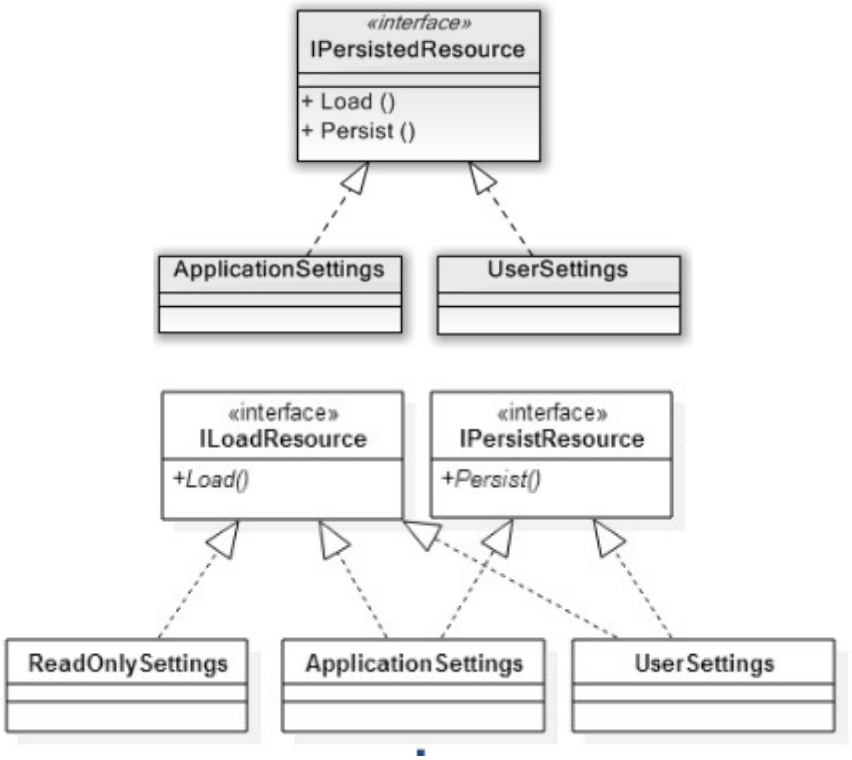
Liskov Subtitution Principle

Los subtipos deben poder sustituibles por sus tipos bases.

Barbara Liskov escribió este principio en 1988:

Lo que se quiere es algo como la siguiente propiedad de sustitución: Si para cada objeto o1 del tipo S hay un objeto o2 del tipo T tal que para todos los programas P definidos en términos de T, el comportamiento de P no se ve afectado cuando o1 es sustituido por o2, entonces S es un subtipo de T.

La importancia de este principio se vuelve obvia cuando consideras las consecuencias de violarlo. Supongamos que tenemos una función *f* que toma como argumento un puntero o referencia a una clase base *B*. Supongamos que hay clases *D* derivadas de *B* las cuales cuando las pasan a *f* disfrazadas de *B* causan un mal comportamiento de *f*. Entonces *D* viola LSP. Claramente *D* es frágil (el diseño es fácil de romper) en presencia de *f*.



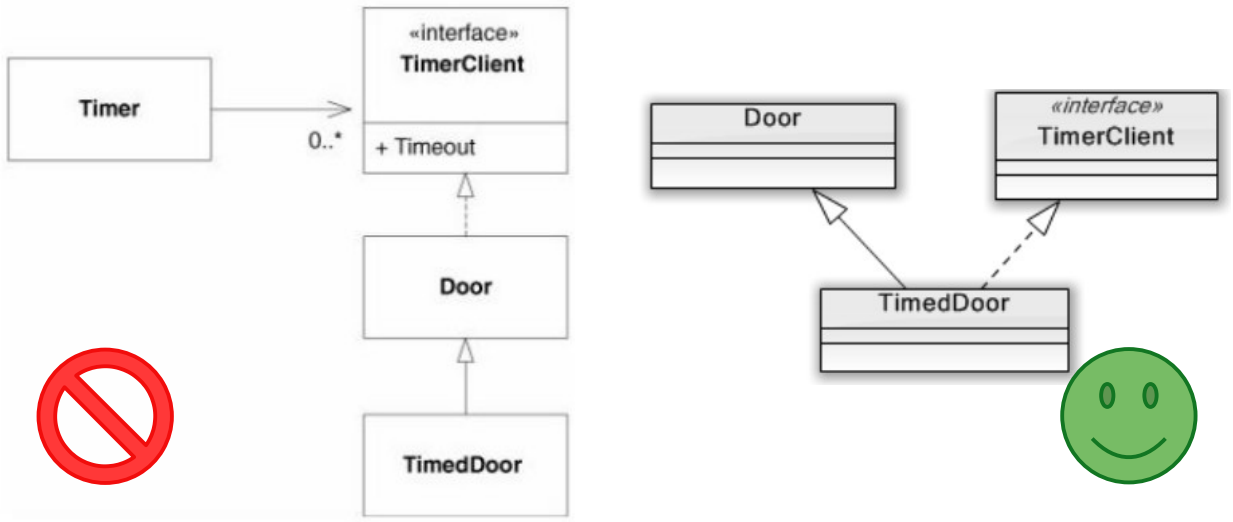
4. ***Explicar el principio SOLID de Segregación de la Interfaz***

   Interface-Segregation Principle

   Los clientes no deben estar forzados a depender de métodos que no usan

   Este principio lidia con las desventajas de interfaces “gordas”. Las clases que tienen interfaces “gordas” son clases cuyas interfaces no son cohesivas. En otras palabras, las interfaces de la clase pueden ser divididas en grupos de métodos. Cada grupo sirve a un conjunto diferente de clientes. Entonces, algunos clientes usan un grupo de funciones y otros clientes usan los otros grupos.

   Las clases gordas causan un extraño y dañino acoplamiento entre sus clientes. Cuando un cliente fuerza un cambio en la clase gorda, todos los otros clientes se ven afectados. Por lo tanto, los clientes solamente deben depender de métodos a los cuales realmente llamen. Esto puede ser logrado dividiendo la interface de la clase gorda en muchas interfaces específicas de cada cliente. Cada interface específica de cada cliente declara solamente aquellas funciones que invoca ese cliente en particular, o grupo de clientes. La clase gorda puede entonces heredar todas las interfaces específicas de cada cliente e implementarlas. Esto rompe la dependencia de los clientes de los métodos que no invocan, y permite a los clientes ser independientes uno del otro.



5. ***Explicar el principio SOLID de Inversión de la Dependencia***

Dependency Inversion Principle

La programación procedural tradicional crea una estructura de dependencia en la que la política depende del detalle. Esto es desafortunado ya que las políticas son entonces vulnerables a los cambios en los detalles. La programación orientada a objetos invierte esa estructura de dependencia en la que tanto el detalle como las políticas dependen de abstracciones, y las interfaces de servicio son usualmente propiedad de sus clientes.

De hecho, esta inversión de dependencias es el sello de calidad de un buen diseño orientado a objetos. No importa el lenguaje en el que el programa es escrito. Si sus dependencias están invertidas, tiene un diseño orientado a objetos. Si sus dependencias no son invertidas, tiene un diseño procedural.

El principio de inversión de dependencias es el mecanismo de bajo nivel fundamental detrás de muchos de los beneficios reclamados para las tecnologías orientadas a objetos. Es críticamente importante para la construcción de código que es resiliente al cambio. Ya que las abstracciones y los detalles están aislados uno del otro, el código es mucho más fácil de mantener.

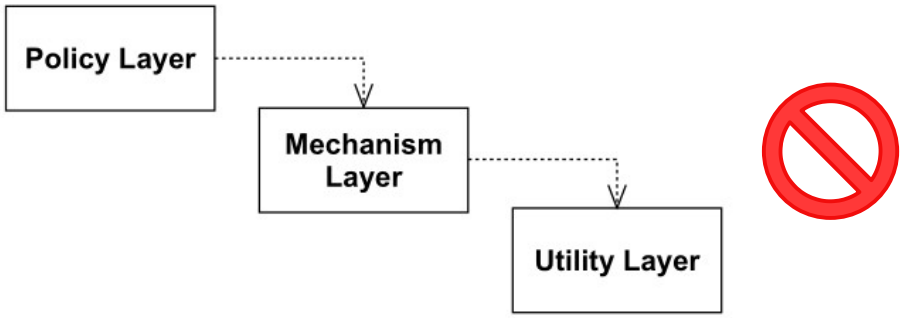


Figure 11-1 Naive layering scheme

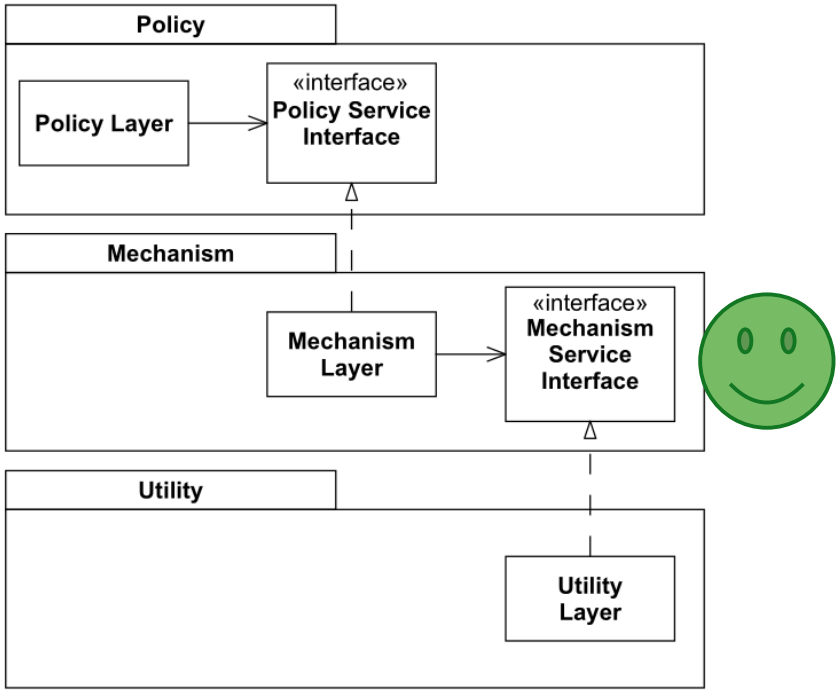


Figure 11-2 Inverted Layers

ANEXO:

Smells de diseño

1. *Rigidez*: el sistema es difícil de modificar porque todo cambio fuerza muchos otros cambios en otras partes del sistema
2. *Fragilidad*: las modificaciones causan que el sistema se rompa en lugares que no tienen relación conceptual con la parte que fue modificada
3. *Inmovilidad*: es difícil desenredar el sistema en componentes que puedan ser reusadas en otros sistemas
4. *Viscosidad*: hacer las cosas bien es más difícil que hacer las cosas mal
5. *Complejidad innecesaria*: el diseño contiene infraestructura que no añade ningún beneficio directo
6. *Repetición innecesaria*: el diseño contiene estructuras repetidas que podrían ser unificadas bajo una sola abstracción
7. *Opacidad*: el sistema es difícil de leer y entender. No expresa bien su intención.

6. Explicar qué es la arquitectura de un sistema

La arquitectura es el núcleo del diseño y del desarrollo de un sistema de software.

Toda la arquitectura es diseño, pero no todo el diseño es arquitectura. La arquitectura representa las decisiones de diseño que le dan forma a un sistema, donde la importancia es medida por el costo de cambio (Grady Booch)

La organización general del sistema (Garlan & Shaw, 1994)

Las partes que componen el software y sus relaciones (Kruchten)

La organización fundamental de un sistema, expresada a través de sus componentes, las relaciones entre ellos y el ambiente, y los principios que gobiernan su diseño y evolución (IEEE 1995)

Conjunto de decisiones de diseño que se hacen en etapas tempranas del proyecto (Anónimo)

Una arquitectura es el conjunto de decisiones importantes sobre la organización de un sistema de software, la selección de los elementos estructurales y sus interfaces por los cuales el sistema está compuesto, junto con su comportamiento como es especificado en las colaboraciones entre estos elementos, la composición de estos elementos estructurales y de comportamiento en subsistemas progresivamente más grandes, y el estilo de arquitectura que guía a esta organización (estos elementos y sus interfaces, sus colaboraciones y composición). (Booch, Rumbaugh y Jacobson en UML User Guide, Addison-Wesley, 1999)

Es el más alto nivel conceptual de un sistema en su ambiente. La arquitectura de un sistema de software (en algún punto en el tiempo) es su organización o estructura de componentes importantes interactuando a través de interfaces y dichos componentes siendo compuestos de, sucesivamente, componentes e interfaces más pequeños. (RUP / IEEE)

7. Enumerar tres razones por las cuales es importante la arquitectura de un sistema

1. Inhibe/habilita los atributos de calidad del sistema. Si un sistema cumplirá con sus atributos de calidad es determinado (pero no garantizado) por la arquitectura.
2. Las decisiones tomadas en una arquitectura permiten administrar los cambios a medida que el sistema evoluciona. La arquitectura debe propiciar que la mayoría de los cambios sean locales.
3. El análisis de una arquitectura permite predicciones tempranas acerca de las cualidades del sistema. Evaluando la arquitectura podríamos predecir si el sistema cumplirá con los atributos de calidad deseados.
4. Una arquitectura documentada mejora la comunicación entre los interesados
    - La arquitectura provee una abstracción del sistema que los interesados pueden usar para entenderlo
    - La arquitectura provee un lenguaje común en el cuál los distintos intereses pueden ser expresados, negociados y resueltos a un nivel manejable para sistemas grandes y complejos.
5. Es el portador de las decisiones de diseño más importantes y difíciles de cambiar
    - La arquitectura refleja las decisiones de diseño más tempranas que tiene gran influencia en el desarrollo, instalación y mantenimiento.
    - Cambiar alguna de esta decisiones causaría un efecto dominó sobre otras.
6. Define un conjunto de restricciones sobre la implementación subsecuente. La implementación de un sistema debe conformar las decisiones de diseño prescriptas por la arquitectura.
7. Influye en la estructura organizacional

- La arquitectura define la descomposición de más alto nivel del sistema
- En grandes proyectos, muchas veces se asignan diferentes porciones del sistema a diferentes grupos.

8. Provee las bases para un prototipo evolutivo
   - Permite hacer un prototipo del esqueleto del sistema
9. Permite al arquitecto y gerente del proyecto razonar acerca del costo y la planificación del sistema. La descomposición de alto nivel del sistema permite, además de crear equipos especializados en las distintas partes, tener un mejor conocimiento de esas partes y hacer estimaciones más precisas de los componentes, y luego, del sistema completo.
10. Puede ser creada como un modelo reusable y transferible que sea el corazón de una línea de productos
11. El desarrollo basado en una arquitectura pone atención en el ensamblaje de los componentes, y no solamente en su creación
    - El desarrollo basado en arquitectura generalmente se enfoca en componer/ensamblar partes que fueron desarrolladas separadamente:
      - Componentes comerciales de terceros
      - Software de código abierto
      - Aplicaciones disponibles en espacios públicos
      - Servicios online
12. Restringiendo las alternativas de diseño, la arquitectura canaliza la creatividad de los desarrolladores, reduciendo el diseño y la complejidad del sistema. Beneficios: mayor reuso, diseños más regulares y simples, tiempos de selección menores, mayor interoperatividad.
13. Puede ser la base para la capacitación de un nuevo integrante del equipo

8. Explicar tres tipos de actividades que debe realizar un Arquitecto de Software

1. *Gestión de requerimientos no funcionales:* los interesados también deben definir los requerimientos no funcionales que necesitan. Para que se puedan satisfacer, éstos deben ser:
   - Específicos
   - Medibles
   - Realizables
   - Comprobables
2. *Definición de la arquitectura:* trata de introducir estructura, lineamientos, principios y liderazgo a los aspectos técnicos de un proyecto de software.
3. *Selección de la tecnología:* se trata de manejar riesgos, reducir riesgos donde hay incertidumbre o alta complejidad; introducir riesgo donde haya beneficios para obtener. Algunos factores a tener en cuenta:

- Costo
- Relación con proveedores
- Interoperabilidad
- Entrega
- Dependencias con otras tecnologías
- Curva de aprendizaje
- Licenciamiento
- Estrategia de la tecnología
- Soporte
- Políticas de actualización
- Conocimiento del equipo

4. *Evaluación de la arquitectura:* el arquitecto debe verificar que la arquitectura...
    - Satisface los requerimientos no funcionales. Puede desarrollar una batería de tests que permitan evaluar los requerimientos no funcionales durante el ciclo de vida del proyecto. De esta manera podemos tener monitoreados estos criterios durante todo el proyecto.
    - Provee los fundamentos necesarios para el resto del código
    - Funciona como una plataforma para resolver los problemas de negocio subyacentes
5. *Colaboración de arquitectura:* el arquitecto de software es el responsable de asegurar que la arquitectura ha sido entendida por todos los interesados en el sistema de software.
6. *Propietario de la visión global:* el arquitecto de software es el responsable de conocer la visión global de la arquitectura, para asegurar que sea cuidada y evolucionada a lo largo del proyecto, ya sea para lograr mejoras o a medida que cambien los requerimientos.
7. *Liderazgo:* el arquitecto de software es la persona indicada para tomar el liderazgo técnico del proyecto.
8. *Coaching y mentoring:* el arquitecto de software debe proveer asistencia a los desarrolladores para resolver problemas particulares o sortear algún impedimento que los bloquee, aportar su experiencia y promover que se comparta el conocimiento entre los miembros del equipo.
9. *Aseguramiento de la calidad:* el arquitecto de software debe:
    - Definir estándares de codificación
    - Delinear principios de diseño a cumplir
    - Utilizar herramientas de análisis de código junto con integración continua para evaluar el cumplimiento de los mismos.
    - Promover el testeo unitario automático
    - Utilizar herramientas de análisis de cobertura de tests
10. *Diseño, desarrollo y testing:* el arquitecto podría experimentar el mismo sufrimiento que el resto de los desarrolladores, lo que lo ayudaría a entender cómo la arquitectura es vista desde la perspectiva de los desarrolladores.

9. Explicar para qué sirve un escenario de atributo de calidad

Un escenario de atributo de calidad es un mecanismo que permite caracterizar/capturar aspectos de atributos de calidad de una forma que puede ser evaluado y utilizado en diseño. Un escenario de atributo de calidad plantea la especificación de requisitos de calidad. Se compone de seis partes.

1. Estímulo: la condición a ser considerada cuando arriba al sistema.
2. Fuente de estímulo: es la entidad (humano, otro sistema, u otro actor) que generó el estímulo.
3. Ambiente: las condiciones sobre las que ocurre el estímulo. Entorno del sistema cuando recibe el estímulo (ej. el sistema está en una condición de sobrecarga, está operativo, etc)
4. Artefacto: el artefacto (componente, equipo, sistema entero) que es estimulado.
5. Respuesta: define las acciones que el artefacto debería realizar como respuesta al estímulo.
6. Medida de la respuesta: la medida de la respuesta por la cual el artefacto es evaluado. La respuesta debe ser medible para probar que se cumple el requerimiento.

10. Explicar qué es una táctica de atributos de calidad y cómo se usa

Una táctica es una decisión de diseño que influencia el control de la respuesta de un atributo de calidad.

Una táctica es una decisión de diseño que da respuesta a necesidades planteadas por un atributo de calidad.

Una táctica arquitectural es un enfoque establecido y probado que puede ser usado para ayudar a lograr un atributo de calidad (por ejemplo, definiendo diferentes prioridades de procesamiento para diferentes partes del volumen del trabajo y manejándolas usando un planificador de procesos basado en prioridades para lograr satisfacer la performance total del sistema). Cada perspectiva identifica y describe las tácticas más importantes para lograr sus atributos de calidad.

No se deben confundir tácticas con patrones de diseño. Aunque tácticas y patrones son ambos fuentes de conocimiento sobre diseño, una táctica es mucho más general y menos limitante que un patrón de diseño clásico, porque no asigna una estructura particular de software, sino que provee una guía general de cómo diseñar un aspecto del sistema en particular. *(N. Rozanski & E. Woods, Software Systems Architecture, p. 43)*

11. Explicar tres atributos de calidad

Atributo externo	Descripción
Disponibilidad	El grado en el cuál los servicios del sistema están disponibles cuando y dónde son necesitados
Instalabilidad	Que tan fácil es de instalar, desinstalar y reinstalar la aplicación
Integridad	El grado en el cuál el sistema protege contra la pérdida e imprecisión de datos
Interoperatividad	Qué tan fácilmente el sistema puede interconectarse e intercambiar información con otros sistemas o componentes
Performance (Rendimiento)	Que tan rápido y predeciblemente el sistema responde a las entradas del usuario y otros eventos
Confiabilidad	Cuanto tiempo funciona el sistema antes de experimentar una falla
Robustez	Qué tan bien el sistema responde a condiciones operativas inesperadas
Protección	Qué tan bien el sistema protege contra lesiones o daño

Seguridad	Qué tan bien el sistema protege contra acceso no autorizado a la aplicación o sus datos
Usabilidad	Qué tan fácil es para las personas aprender, recordar y usar el sistema
Atributo interno	Descripción
Eficiencia	Qué tan eficientemente el sistema usa los recursos de la computadora
Modificabilidad	Qué tan fácil es mantener, cambiar, mejorar y reestructurar el sistema
Portabilidad	Qué tan fácil el sistema puede hacerse funcionar en otros ambientes operativos
Reusabilidad	A qué grado los componentes pueden ser usados en otros sistemas
Escalabilidad	Qué tan fácil el sistema puede crecer para manejar más usuarios, transacciones, servidores y otras extensiones
Verificabilidad	Qué tan fácilmente los desarrolladores y testers pueden confirmar que el software fue implementado correctamente

12. Explique la diferencia entre estilos y patrones de arquitectura

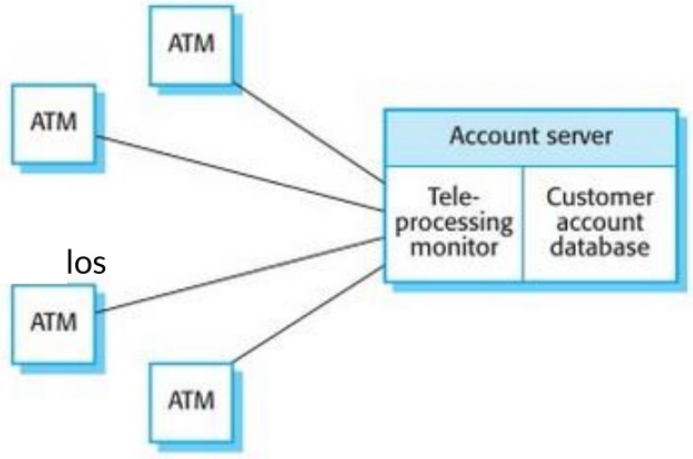
Un estilo arquitectónico es una colección de decisiones de diseño arquitectónico que:

1. son aplicables a un contexto de desarrollo,
2. dado un sistema particular, restringe las decisiones de diseño de arquitectura sobre dicho sistema, y
3. garantiza ciertas calidades del sistema resultante

Un patrón arquitectónico es una colección de decisiones de diseño arquitectónico que son aplicables a problemas de diseño recurrentes, y que están parametrizados para tener en cuenta los diferentes contextos de desarrollo de software en el que surge el problema.

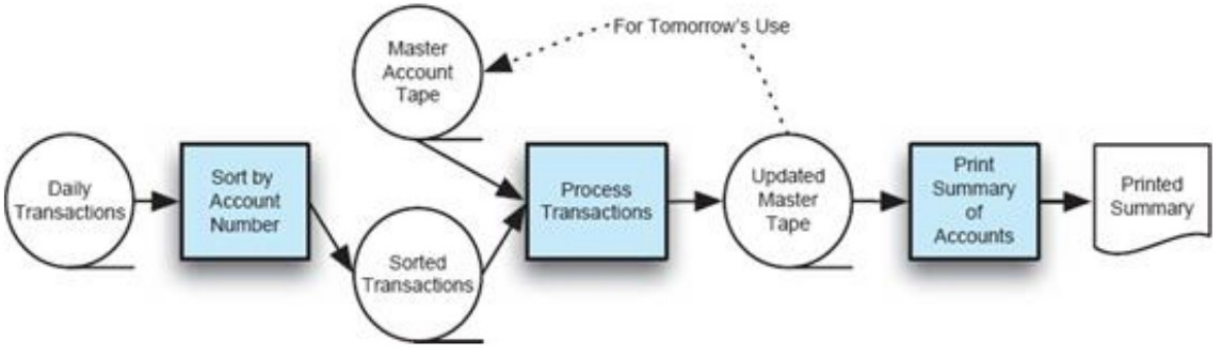
	Estilo	Patrón
Alcance	Aplican a un contexto de desarrollo, por ej.: sistemas altamente distribuidos, sistemas intensivos en GUI	Aplican a problemas de diseño específicos, por ej.: el estado del sistema debe presentarse de múltiples formas, la lógica de negocio debe estar separada del acceso a datos
Abstracción	Son muy abstractos para producir un diseño concreto del sistema	Son fragmentos arquitectónicos parametrizados que pueden ser pensados como una pieza concreta de diseño
Relación	Un sistema diseñado de acuerdo a las reglas de un único estilo puede involucrar el uso de múltiples patrones	Un único patrón puede ser aplicado a sistemas diseñados de acuerdo a los lineamientos de múltiples estilos

13. Explicar las características que presenta una arquitectura cliente/servidor



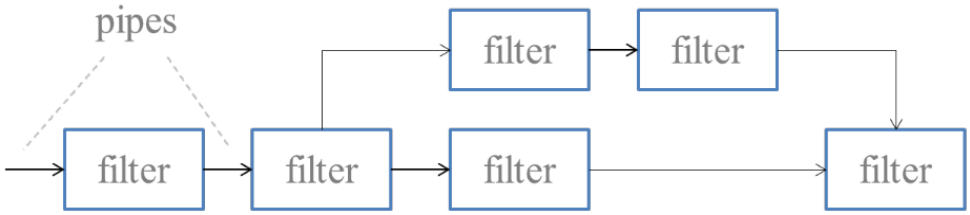
DESCRIPCIÓN	Clientes le envían requerimientos al servidor, el cuál los ejecuta y envía la respuesta (de ser necesario). La comunicación es iniciada por el cliente.
COMPONENTES	Clientes y Servidor
CONECTORES	Llamadas a procedimientos remotos
ELEMENTOS DE DATOS	Parámetros y valores de retorno
TOPOLOGÍA	Dos niveles. Múltiples clientes haciendo requerimientos al servidor
RESTRICCIONES ADICIONALES	Prohibida la comunicación entre clientes
CUALIDADES	• Sencilla y muy utilizada • Centralización de cómputos y datos en el server • Mantenible • Un único servidor puede atender a múltiples clientes
USOS TÍPICOS	• Apps con datos y/o procesamiento centralizados y clientes GUI • Stacks de protocolos de red • Aplicaciones empresariales
PRECAUCIONES	• Condiciones de la red vs. Crecimiento de clientes

14. Explicar las características que presenta una arquitectura batch/secuencial



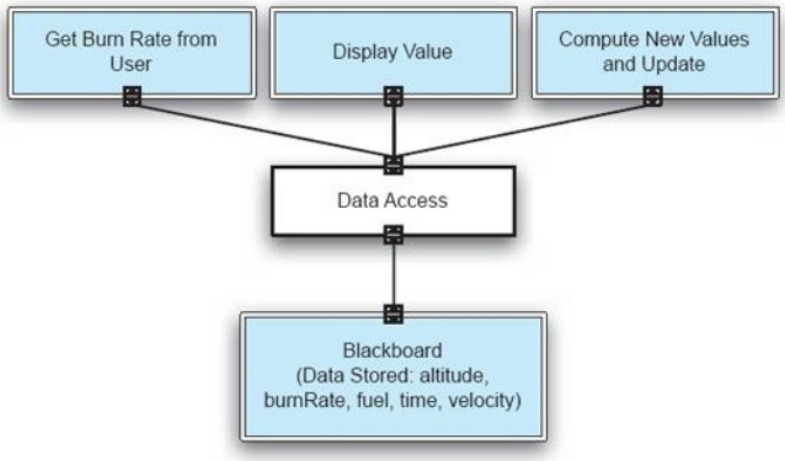
DESCRIPCIÓN	Programas separados y ejecutados en orden. Los datos son pasados como un lote de un programa al siguiente.
COMPONENTES	Programas independientes
CONECTORES	Distintos tipos de interfaces: desde la humana hasta web services
ELEMENTOS DE DATOS	Lotes de datos pasados de un programa al siguiente
TOPOLOGÍA	Lineal
RESTRICCIONES ADICIONALES	Se ejecuta un programa a la vez, hasta que termina
CUALIDADES	• Sencillez • Ejecuciones independientes
USOS TÍPICOS	Procesamiento de transacciones en sistemas financieros
PRECAUCIONES	• Cuando se requiere interacción entre componentes • Cuando se requiere concurrencia entre componentes

15. Explicar las características que presenta una arquitectura pipes & filters



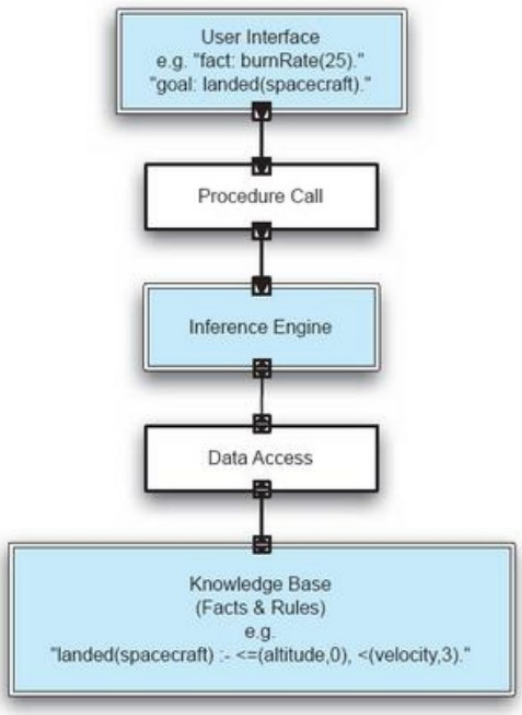
DESCRIPCIÓN	Programas separados y ejecutados, potencialmente de manera concurrente. Datos son pasados como un stream de un programa al siguiente
COMPONENTES	Programas independientes (filtros)
CONECTORES	Routers explícitos de streams de datos
ELEMENTOS DE DATOS	Stream de datos
TOPOLOGÍA	Pipeline (conexiones en T son posibles)
RESTRICCIONES ADICIONALES	
CUALIDADES	• Filtros mutuamente independientes • Estructura simple de streams de entrada/salida facilitan la combinación de componentes • Flexibilidad: agregar, eliminar, cambiar y reusar filtros
USOS TÍPICOS	• Aplicaciones sobre sistemas operativos • Procesamiento de audio y video • Web servers (procesamiento de requerimientos HTTP)
PRECAUCIONES	• Cuando estructuras de datos complejos deben ser pasadas entre filtros • Cuando se requiere interacción entre filtros

16. Explicar las características que presenta una arquitectura tipo blackboard



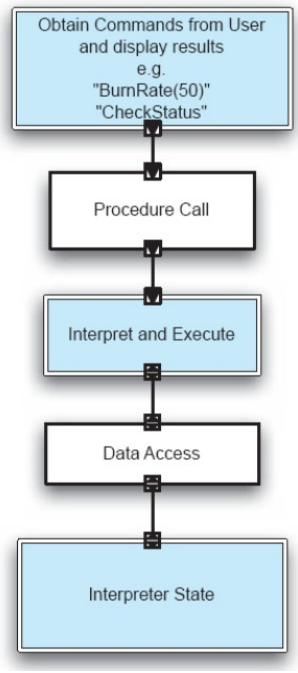
DESCRIPCIÓN	Programas independientes que acceden y se comunican a través de un repositorio de datos global (blackboard)
COMPONENTES	Blackboard (estructura central de datos) Programas independientes operando sobre la pizarra
CONECTORES	Acceso al blackboard (referencia directa a memoria, llamada a procedimiento, consultas a la base de datos, etc)
ELEMENTOS DE DATOS	Datos almacenados en el blackboard
TOPOLOGÍA	Estrella, con un blackboard al medio
RESTRICCIONES ADICIONALES	Detección de cambios en el blackboard • Polling sobre el blackboard • Blackboard Manager se encarga de notificar cambios
CUALIDADES	La solución completa a un problema no tiene que ser preplanificada. La evolución del estado determina las estrategias a ser adoptadas
USOS TÍPICOS	• Resolución de problemas heurísticos en inteligencia artificial • Compiladores
PRECAUCIONES	• Si la interacción entre programas “independientes” necesita de reglas de regulación compleja • Cuando los datos en el blackboard están sujetos a cambios frecuentes y se requiere propagarlos entre todos los componentes participantes

17. Explicar las características que presenta una arquitectura basada en reglas



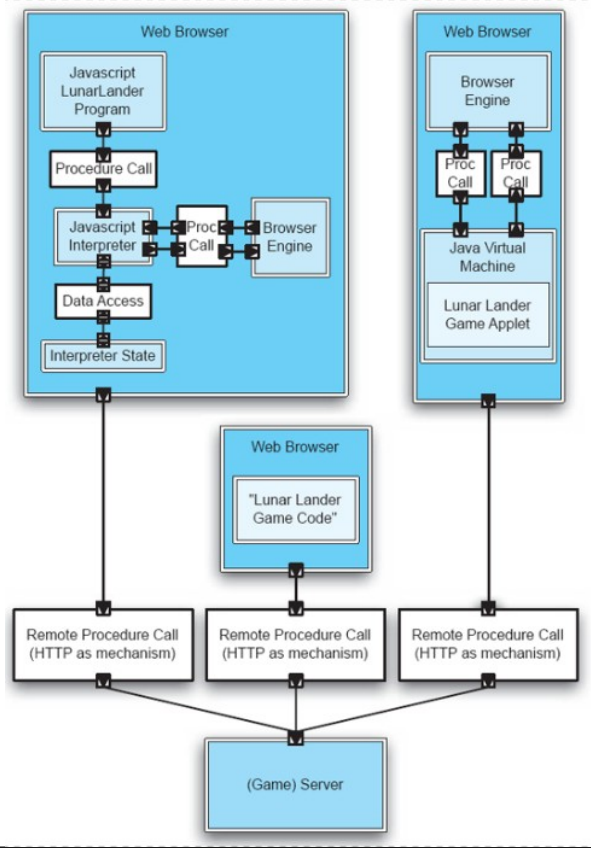
DESCRIPCIÓN	El motor de inferencias parsea la entrada del usuario • Si es un hecho/regla, la agrega a su base de conocimiento • Si es una consulta (goal), obtiene las reglas aplicables desde la base de conocimiento e intenta resolverla
COMPONENTES	Interfaz de usuario Motor de inferencias Base de conocimiento
CONECTORES	Las componentes están estrechamente conectados a través de: • llamadas a procedimientos • acceso a datos compartidos
ELEMENTOS DE DATOS	Reglas/hechos y consultas
TOPOLOGÍA	3 capas altamente acopladas: • Interfaz de usuario • Motor de inferencia • Base de conocimiento
RESTRICCIONES ADICIONALES	
CUALIDADES	• Modificabilidad: el comportamiento de la aplicación puede ser modificado agregando o eliminando reglas dinámicamente • Facilita el prototipado de sistemas pequeños
USOS TÍPICOS	Cuando el problema puede ser entendido como una cuestión de resolver repetidamente un conjunto de predicados
PRECAUCIONES	Cuando se tiene una gran cantidad de reglas, entender las interacciones entre múltiples reglas afectadas por los mismos hechos llega a ser difícil

18. Explicar las características que presenta una arquitectura basada en intérprete



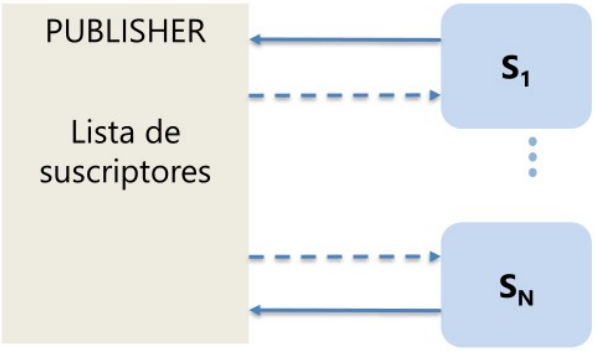
DESCRIPCIÓN	Parsea y ejecuta comandos de entrada, actualizando el estado mantenido por el intérprete
COMPONENTES	• Intérprete de comandos • Estado del programa interpretado • Estado del intérprete • Interfaz de usuario
CONECTORES	Los componentes están estrechamente conectados a través de: • Llamadas a procedimientos • Acceso a datos compartidos
ELEMENTOS DE DATOS	Comandos
TOPOLOGÍA	3 capas altamente acopladas El estado puede estar separado del intérprete
RESTRICCIONES ADICIONALES	
CUALIDADES	• Es posible obtener un comportamiento altamente dinámico, donde el conjunto de comandos se va modificando • La arquitectura del sistema puede ser constante mientras se crean nuevas capacidades a partir de primitivas existentes
USOS TÍPICOS	End-user programming
PRECAUCIONES	Cuando se necesita un procesamiento rápido (el código interpretado tarda mucho más que el código ejecutable)

19. Explicar las características que presenta una arquitectura de código móvil



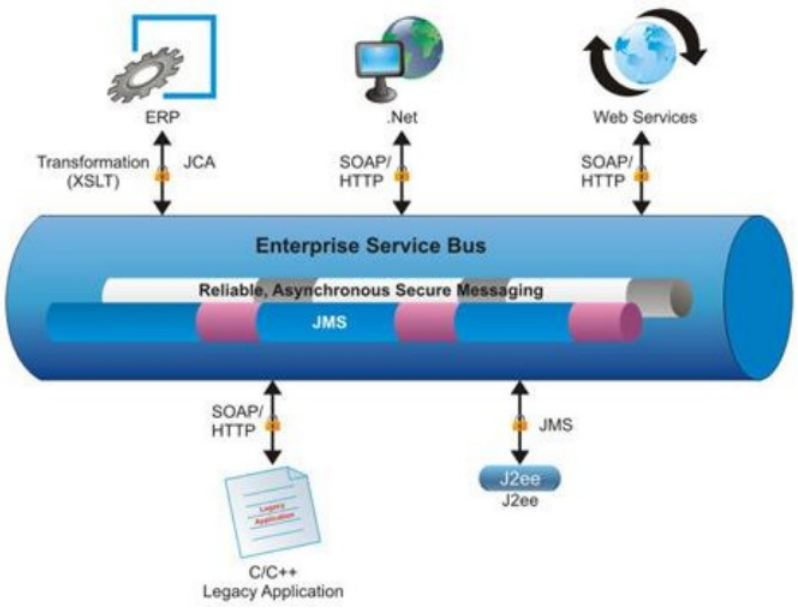
DESCRIPCIÓN	El código se mueve para ser interpretado en otro host
COMPONENTES	• Dock de ejecución (recepción y deployment de código y estado) • Intérprete/compilador de código
CONECTORES	Protocolos de red y elementos para empaquetar código y datos para transmisión
ELEMENTOS DE DATOS	• Representación de código como datos • Estados del programa • Datos
TOPOLOGÍA	Red
RESTRICCIONES ADICIONALES	
CUALIDADES	• Adaptabilidad dinámica • Toma ventaja del poder de procesamiento del host • Confianza: se incrementa al permitir migrar a un nuevo host de manera simple
USOS TÍPICOS	Cuando se procesan grandes conjuntos de datos en locaciones distribuidas (es más eficiente que el código se mueva al lugar donde se encuentran esos datos)
PRECAUCIONES	• Seguridad: la ejecución de código importado abre la puerta a malware • Cuando el costo de transmisión excede al costo de ejecución • Cuando las conexiones de red no están disponibles

20. Explicar las características que presenta una arquitectura de tipo publish/subscriber



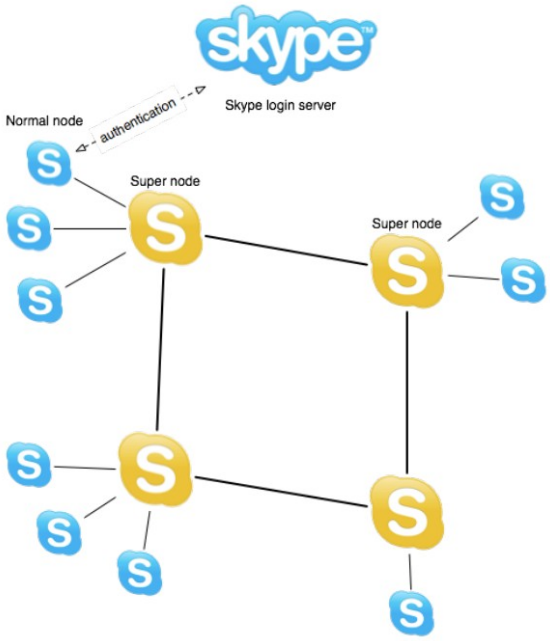
DESCRIPCIÓN	Subscribers se registran/desregistran para recibir mensajes o contenidos específicos. Cuando el Publisher publica, el mensaje es enviado a los Subscribers
COMPONENTES	• Publishers • Subscribers • Proxies para manejar la distribución
CONECTORES	Llamadas a procedimientos pueden ser usadas dentro de un programa Protocolos de red son más frecuentes
ELEMENTOS DE DATOS	• Suscripciones • Notificaciones • Información publicada
TOPOLOGÍA	Subscribers se conectan a Publishers en forma directa o pueden recibir notificaciones de intermediarios
RESTRICCIONES ADICIONALES	
CUALIDADES	Altamente eficiente para distribuir información en un solo sentido con muy bajo acoplamiento de componentes
USOS TÍPICOS	• Distribución de noticias • GUIs • Juegos en red multiplayer
PRECAUCIONES	Cuando la cantidad de Subscribers para un tópico es muy grande, un protocolo especial puede ser necesario
VARIANTES	• Usos específicos del estilo puede requerir pasos particulares en la suscripción/desuscripción • Soporte para el matching complejo de intereses y la información disponible puede ser provisto por los intermediarios

21. Explicar las características que presenta una arquitectura basada en eventos



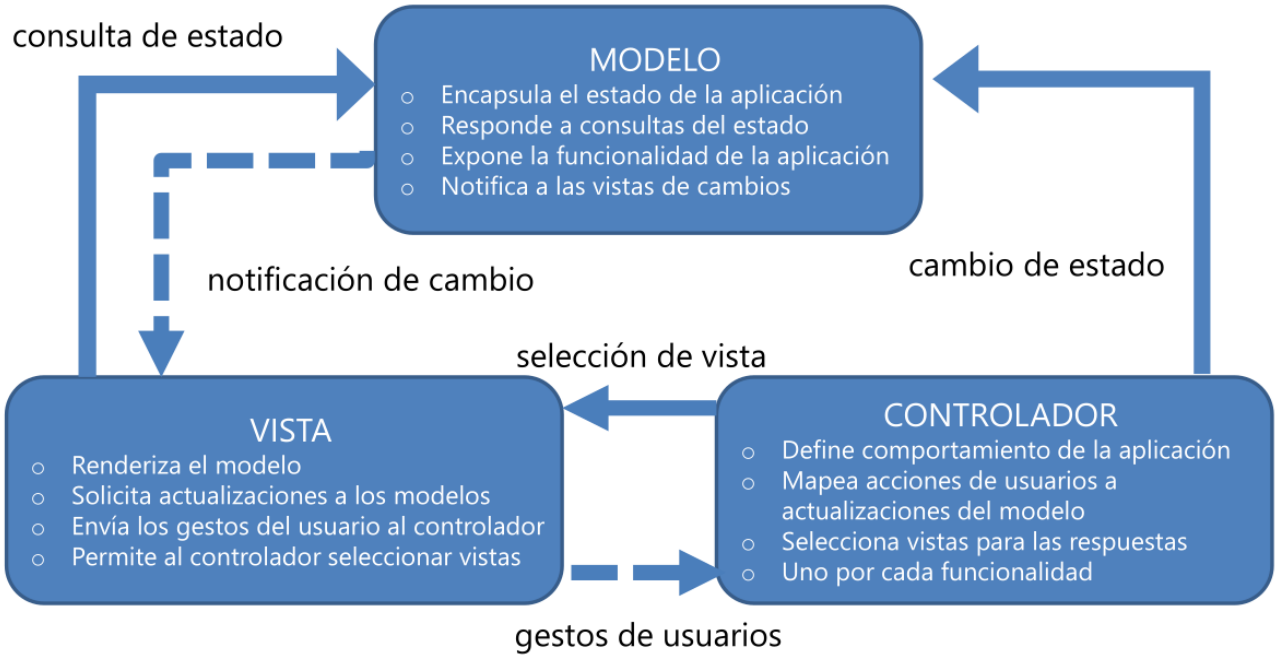
DESCRIPCIÓN	Componentes independientes que asincrónicamente emiten y reciben eventos comunicados a través de event-buses
COMPONENTES	Generadores y/o consumidores de eventos independientes y concurrentes
CONECTORES	Event-bus. Podría existir más de uno
ELEMENTOS DE DATOS	Eventos
TOPOLOGÍA	Los componentes se comunican con el event-bus, no directamente entre ellos
RESTRICCIONES ADICIONALES	
CUALIDADES	• Altamente escalable • Fácil de evolucionar • Efectivo para aplicaciones heterogéneas altamente distribuidas
USOS TÍPICOS	• Software de UI • Aplicaciones de área amplia que involucran partes independientes (mercados financieros, logística, redes de censado)
PRECAUCIONES	No existen garantías que un evento sea procesado, ni cuándo lo será
VARIANTES	Comunicación push o pull

22. Explicar las características que presenta una arquitectura de tipo peer-to-peer



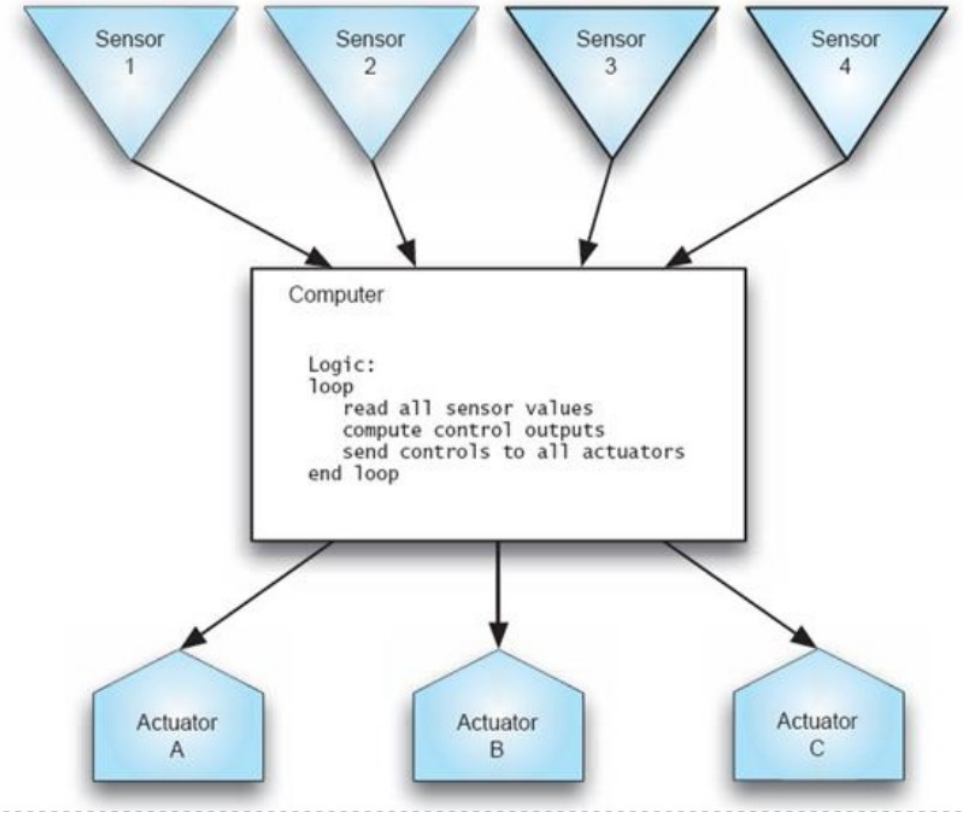
DESCRIPCIÓN	Estado y comportamientos son distribuidos entre pares que pueden actuar tanto como clientes como servidores
COMPONENTES	Peers: componentes independientes que tienen su propio estado y control
CONECTORES	Protocolos de red, generalmente propietarios
ELEMENTOS DE DATOS	Mensajes de red
TOPOLOGÍA	Red que puede tener conexiones redundantes entre peers. Puede variar arbitraria y dinámicamente
RESTRICCIONES ADICIONALES	
CUALIDADES	• Altamente robusto de cara a la falla de un nodo • Escalable, en términos de acceso a los recursos y poder de cómputo • Computación distribuida
USOS TÍPICOS	Cuando las fuentes de información y operación están distribuidas: • File sharing • Mensajería instantánea • Grid computing
PRECAUCIONES	• Cuando la recuperación de la información es crítica en tiempo y no puede hacer frente a la latencia propuesta por el protocolo • Seguridad

23. Explicar las características que presenta el patrón de arquitectura MVC



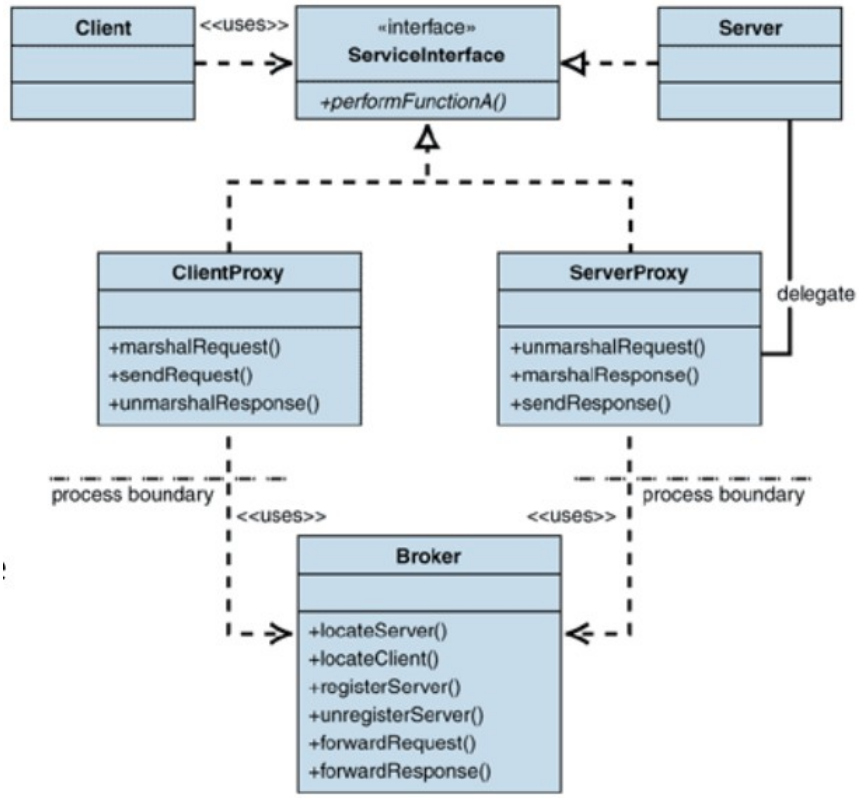
DESCRIPCIÓN	Separa la funcionalidad del sistema entre los componentes modelo, vista y controlador
COMPONENTES	• Modelo: es la representación de los datos o el estado de la aplicación. Contiene (o cuenta con la interfaz) la lógica de la aplicación • Vista: es un componente intefaz de usuario. Produce representaciones del modelo para el usuario y/o permite alguna manera de ingresar datos • Controller: maneja la interacción entre el modelo y la vista, trasladando las acciones de usuario en cambios al modelo o la vista
RELACIONES	La relación *notifica* conecta instancias del modelo, vista y controlador
RESTRICCIONES	• Debe existir al menos una instancia de vista, modelo y controlador • El componente modelo no debe interactuar directamente con el controlador
DEBILIDADES	• Puede ser demasiado complejo en aplicaciones con interfaces de usuario simples • Las abstracciones modelo-vista-controlador pueden no ser adecuadas para algunas herramientas de interface de usuario

24. Explicar las características que presenta el patrón de arquitectura SENSE-COMPUTE-CONTROL



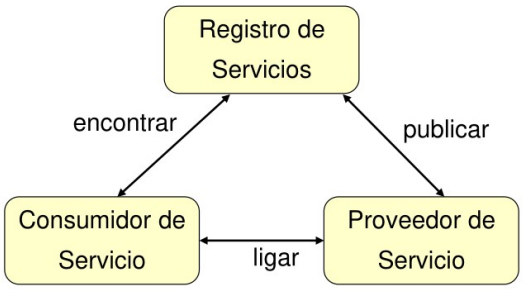
DESCRIPCIÓN	• Una computadora es embebida en alguna aplicación • Los sensores de distintos dispositivos están conectados a la computadora • Los sensores son interrogados para determinar su valor • La computadora también tiene asociados dispositivos Actuators • La computadora envía señales a los dispositivos a través de sus Actuators, logrando así controlar el sistema
COMPONENTES	• Sensores: sus valores son leídos por la computadora • Computadora: ejecuta un conjunto de leyes o funciones de control • Actuators: reciben la salida de la computadora
RELACIONES	Frecuencia del reloj: tasa máxima en que los valores de los sensores pueden cambiar, o en la sensibilidad con la que los Actuators pueden recibir actualizaciones
RESTRICCIONES	
DEBILIDADES	

25. Explicar las características que presenta el patrón de arquitectura BROKER



DESCRIPCIÓN	El patrón define un componente broker en tiempo de ejecución que oficia de mediador de la comunicación entre clientes y servidores
COMPONENTES	• Cliente: solicita un servicio • Servidor: provee servicios • Broker: intermediario responsable de localizar un servidor apropiado para cumplir la petición del cliente, reenviar la solicitud al servidor, y devolver los resultados al cliente • Proxy del lado del cliente: intermediario que resuelve la comunicación real con el broker, incluye clasificar, enviar y desclasificar mensajes • Proxy del lado del servidor: intermediario que resuelve la comunicación real del servidor con el broker, incluye clasificar, enviar y desclasificar mensajes
RELACIONES	La relación *adjuntar* asocia clientes (opcionalmente proxy-cliente) y servidores (opcionalmente proxy-server) con brokers
RESTRICCIONES	El cliente sólo se puede asociar con un broker (potencialmente vía proxy-cliente). El servidor solo se puede asociar con un broker (potencialmente vía proxy-server)
DEBILIDADES	• El broker agrega un nivel de indirección (aumenta la latencia y puede resultar en el cuello de botella) • El broker resulta el punto de falla • Puede ser un objetivo de ataques de seguridad • Puede ser difícil de testear

26. Explicar las características que presenta una arquitectura SOA

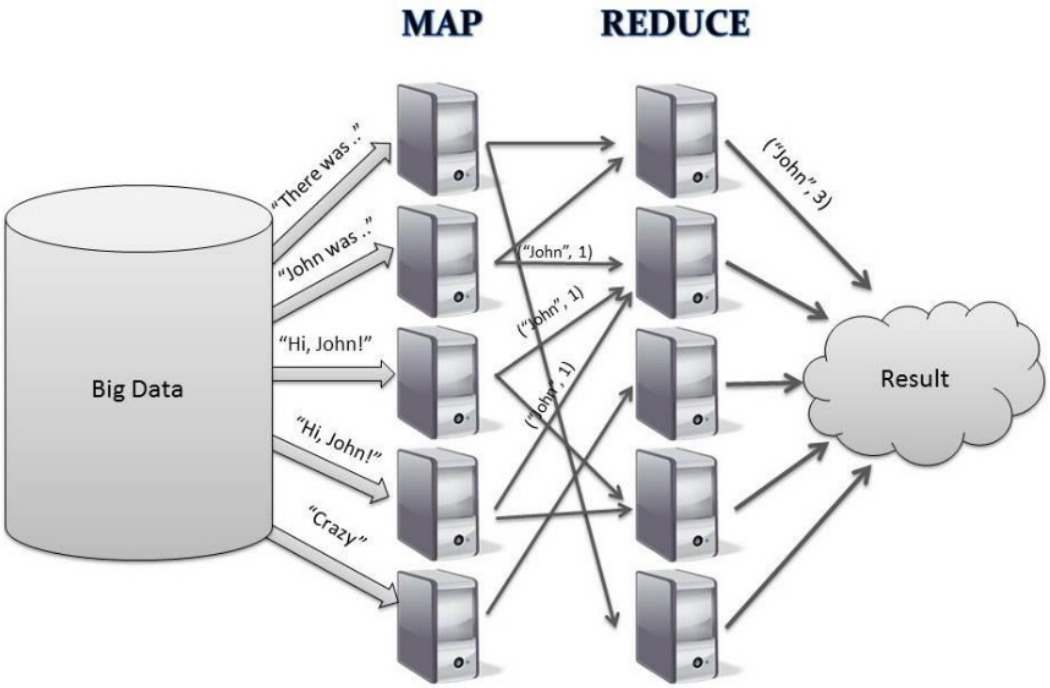


DESCRIPCIÓN	SOA es una arquitectura de software donde todos los servicios y procesos implementados en software están diseñados como servicios a ser consumidos a través de una red
COMPONENTES	• Proveedor de servicio: cualquier entidad que aloja a un servicio web disponible a través de la red ◦ Crea una descripción de servicio ◦ Despliega el servicio en un entorno de ejecución para hacerlo disponible a otras entidades sobre la red ◦ Publica la descripción del servicio en uno o más registros de servicios ◦ Recibe mensajes invocando el servicio por parte de los consumidores del servicio • Consumidor de servicio: puede ser cualquier “entidad” que requiera de un servicio disponible en la red ◦ Encuentra la descripción de un servicio publicada en un registro de servicios ◦ Aplica la descripción del servicio para ligar e invocar al servicio web alojado en el proveedor del servicio • Registro de servicio (intermediario): permite ligar a los proveedores de servicios con los consumidores. Una vez que se han ligado, las interacciones se realizan directamente entre el proveedor y el consumidor del servicio. ◦ Acepta pedidos de los proveedores de servicios para publicar y difundir las descripciones de los servicios web ◦ Permite que los consumidores de servicios busque en la colección de descripciones de servicios contenida dentro del registro
RELACIONES	• Publicar: registración o anuncio del servicio • Encontrar: búsqueda de un servicio que satisfaga determinadas condiciones • Ligar: crea una relación cliente-servidor entre el proveedor y el consumidor del servicio
RESTRICCIONES	
DEBILIDADES	• Complejidad en el diseño e implementación, debido al binding dinámico y el uso de meta-datos • Overhead en la performance del middleware (intermediario) que se interpone entre servicios y clientes • Carencia de garantía de performance, ya que los servicios se comparten y no están bajo el control del cliente

PROPIEDADES	• Visión lógica: los servicios son una abstracción de lo que los programas, bases de datos y procesos de negocio son capaces de hacer • Abstracción: SOA esconde los detalles subyacentes de implementación (lenguajes, procesos, estructuras de bases de datos, etc) • Relevancia de mensajes: un servicio es definido en término de los mensajes que se intercambian entre el agente proveedor y el consumidor, no en términos de las propiedades de los agentes en sí mismos • Metadatos: un servicio es descripto por meta-datos procesables automáticamente • Operaciones: un servicio tiende a depender de un número pequeño de operaciones con mensajes relativamente largos y complejos • Red: los servicios están definidos para ser usados sobre una red • Independencia de plataforma: los mensajes son enviados en un formato estandarizado definido, usualmente XML, enviados a través de interfaces
BENEFICIOS	• SOA permite que los agentes que participan en el intercambio de mensajes estén débilmente acoplados. Esto permite una mayor flexibilidad ◦ Las componentes funcionales y sus interfaces están separadas (nuevas interfaces pueden ser fácilmente añadidas) ◦ Funcionalidad vieja y nueva se pueden encapsular como componentes de software que proporcionen y solicitan servicios • El control de los procesos de negocio puede ser aislado • Los servicios se pueden incorporar de forma dinámica en tiempo de ejecución

	SOAP	REST
Origen	Modelo de llamadas entre aplicaciones del tipo Llamada a Procedimiento Remoto (RPC)	Modelo cliente-servidor sobre HTTP
Transmisión de mensajes	HTTP y RPC (otros protocolos son posibles)	HTTP
Restricciones	• Seguridad • Transacciones • Interpretación de mensaje • Esquema de direccionamiento	• Único esquema de direccionamiento (URI) • Operaciones limitadas (CRUD)
Sistema de tipos	Simple, comparable al de muchos lenguajes de programación	No tiene
Semántica	Las aplicaciones que interactúan necesitan acordar cómo interpretar los mensajes	Si bien la sintaxis es bastante autodescriptiva, no está garantizada
Tradeoffs	• Completitud • Estandarización de la interoperabilidad • Mensajes estructurados	• Simplicidad • Mínimo overhead • Performance
Calidad de servicio	La familia de tecnologías de Servicios Web tiene soporte para la seguridad, disponibilidad, manejo de transacciones, etc.	No brinda soporte intrínseco. Apropiado para operaciones read-only

27. Explicar las características que presenta una arquitectura MAP-REDUCE

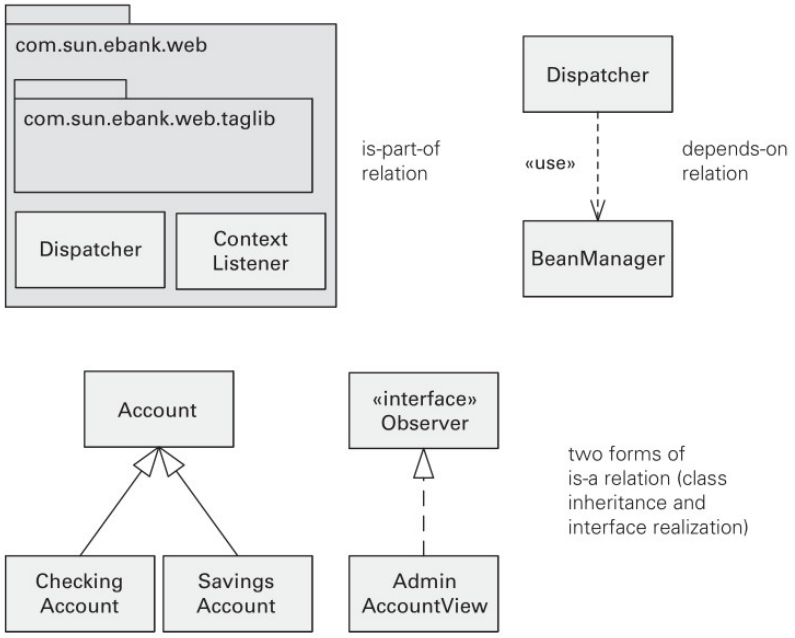


DESCRIPCIÓN	Map-Reduce provee un marco de trabajo para analizar un conjunto de datos grande y distribuido, que se ejecuta en paralelo sobre un conjunto de procesadores. El paralelismo permite baja latencia y alta disponibilidad
COMPONENTES	• Función Map: toma una lista de pares en un dominio de datos y devuelve una lista de pares en un dominio diferente (datos intermedios). Su propósito es filtrar el data set, determinando si un registro va a esta involucrado en el procesamiento posterior. El resultado de esta función es tomado por la infraestructura que junta los pares con la misma clave y los agrupa, creando un grupo por cada una de las claves • Función Reduce: es aplicada en paralelo a cada grupo, produciendo una colección de valores para cada dominio. El conjunto de datos de salida es siempre mucho más chico que el de entrada.
RELACIONES	
RESTRICCIONES	
DEBILIDADES	

Cuando no es apropiado este patrón:

- Si no existen grandes volúmenes de datos (ya que el overhead de Map-Reduce no se justifica)
- Si no se puede dividir el conjunto de datos inicial en subconjuntos de tamaño uniforme (en este caso, las ventajas del paralelismo se diluyen)
- Si se tienen operaciones que requieran múltiples “reduce” (aunque es posible, es más difícil de orquestar)

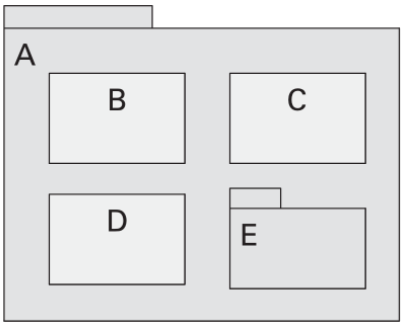
28. Explicar para qué sirve una vista de módulos (a partir de un estilo)



DESCRIPCIÓN	La vista de módulos es la descripción de la principal implementación de unidades, o módulos, de un sistema, junto con las relaciones entre esas unidades. Es poco probable que la documentación de una arquitectura de software pueda ser completada sin al menos una vista de módulos.
COMPONENTES	Módulos, los cuales son implementaciones de unidades de software que proveen un conjunto coherente de responsabilidades
RELACIONES	• *Es parte de*, la cuál define una parte o toda la relación entre el submódulo (la parte) y el módulo agregado (el todo) • *Depende de*, la cuál define una relación de dependencia entre dos módulos. Los estilos específicos del módulo explican lo que significa dependencia. • *Es un*, la cuál define una relación de generalización/especialización entre un módulo más específico (el hijo) y un módulo más general (el padre)
RESTRICCIONES	Diferentes vistas de módulo pueden imponer restricciones específicas en la topología
PARA QUÉ SIRVE	• Provee un plano general de la construcción del código • Facilita el análisis de impacto • Planificación de desarrollo incremental • Permite el análisis de traceabilidad de requerimientos • Explica la funcionalidad de un sistema y la estructura del código base • Ayuda en la definición de asignaciones de trabajo, calendarios de implementación e información de presupuesto • Muestra la estructura de información a ser persistente

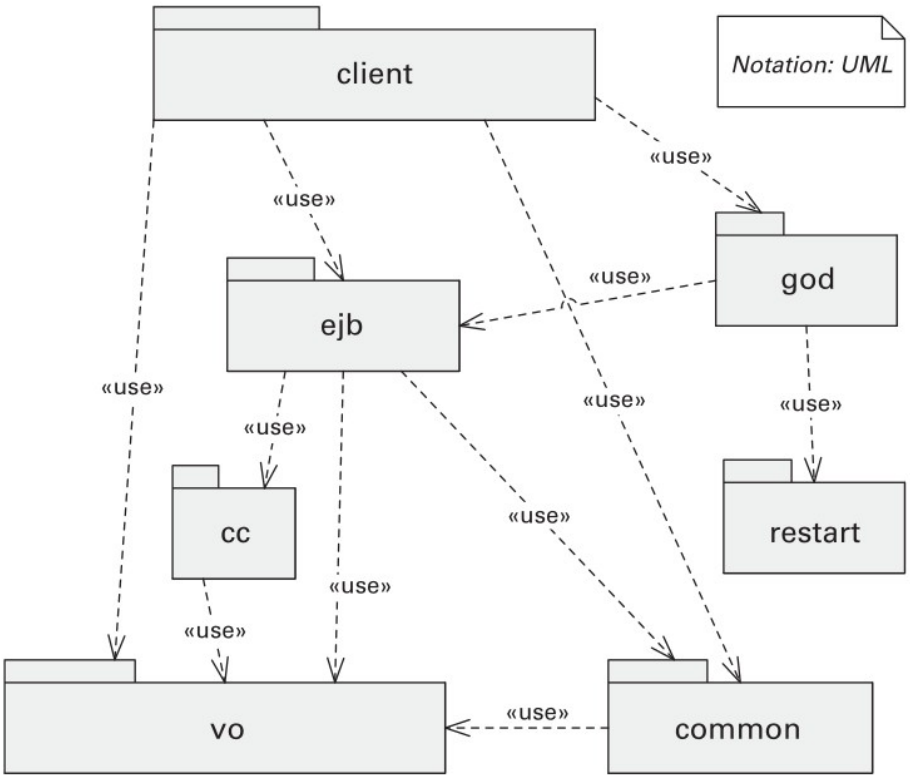
(P. Clements, Documenting Software Architectures, p. 55)

Estilo de descomposición



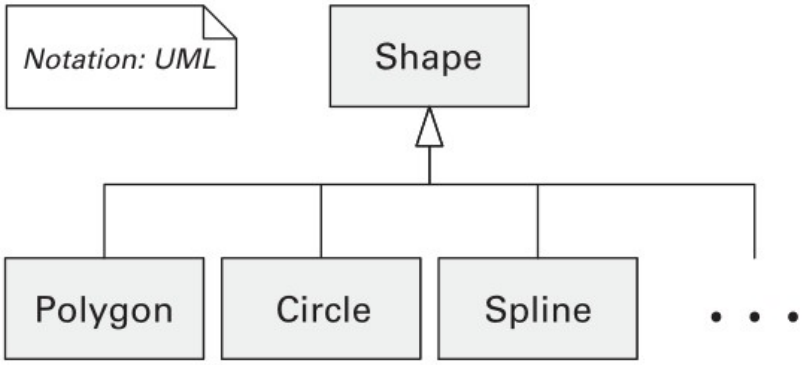
DESCRIPCIÓN	El estilo de descomposición es usado para descomponer un sistema en unidades de implementación. Una vista de descomposición describe la organización del código como módulos y submódulos y muestra cómo las responsabilidades del sistema son particionadas a través de ellas
COMPONENTES	Módulos
RELACIONES	Relación *descomposición*, la cuál es una forma de la relación *es-parte-de*. La documentación debe especificar el criterio usado para definir la descomposición
RESTRICCIONES	• Los ciclos no son permitidos en el gráfico de *descomposición* • Un módulo puede tener solamente un padre
PARA QUÉ SIRVE	• Razonar y comunicar a los recién llegados la estructura del software de manera digerible • Proveer input para la asignación de trabajo • Razonar sobre la localización de los cambios

Estilo de usos



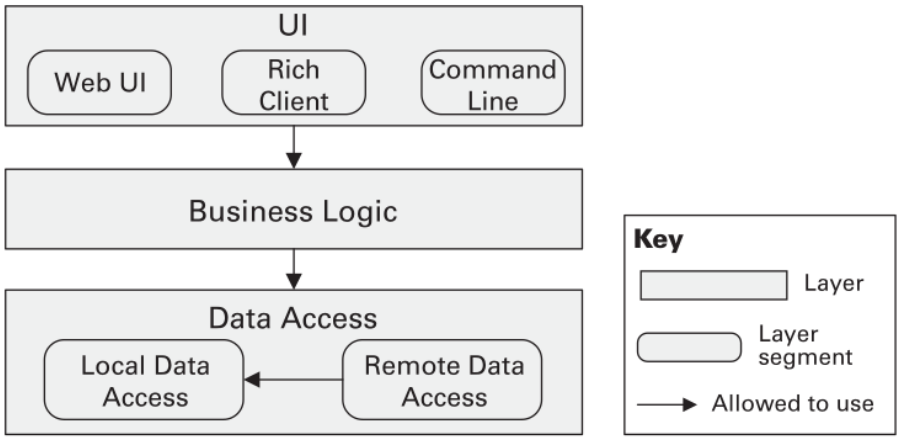
DESCRIPCIÓN	El estilo de usos muestra cómo los módulos dependen unos de otros. Es útil para planificación porque ayuda a definir subconjuntos e incrementos del sistema siendo desarrollado
COMPONENTES	Módulos
RELACIONES	Relación *usa*, la cuál es una forma de la relación *depende-de*. El módulo A *usa* el módulo B si A *depende-de* la presencia y el correcto funcionamiento B para satisfacer sus propios requerimientos
RESTRICCIONES	El estilo de usos no tiene restricciones topológicas. Sin embargo, si las relaciones *usa* presentan ciclos, mucha complejidad, o largas cadenas de dependencias, la habilidad de la arquitectura para ser entregada en subconjuntos incrementales será dañada
PARA QUÉ SIRVE	• Planificar desarrollo incremental y en subconjuntos • Debugging y testing • Medir el efecto de los cambios

Estilo de generalización



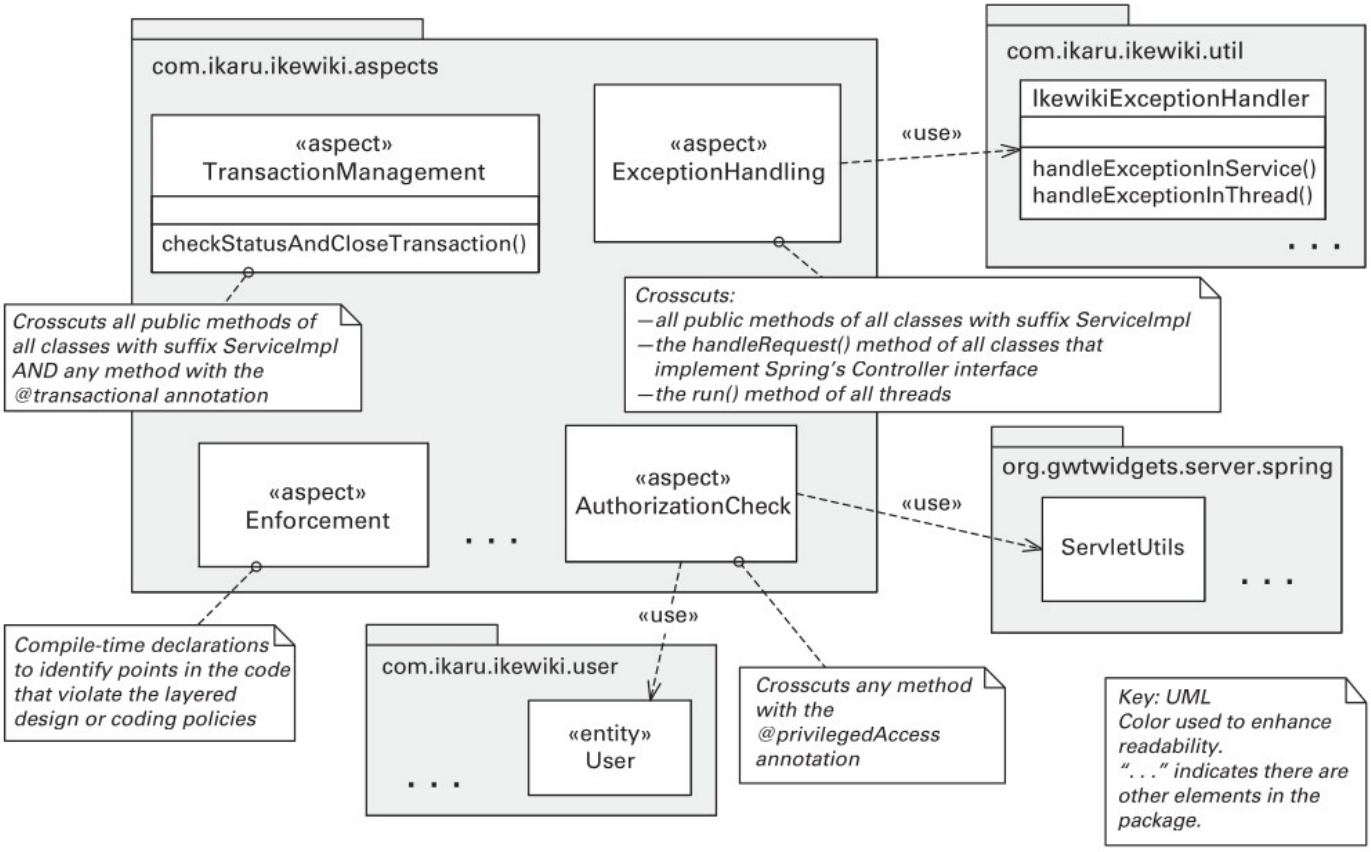
DESCRIPCIÓN	El estilo de generalización emplea la relación *es-un* para soportar la extensión y la evolución de arquitecturas y elementos individuales. Los módulos en este estilo son definidos de manera que capturen puntos en común y variaciones
COMPONENTES	*Módulos*. Un módulo puede tener la propiedad “abstracto” para indicar que no contiene una implementación completa
RELACIONES	*Generalización*, la cual es una especialización de la relación *es-un*. La relación puede ser especializada más para indicar, por ejemplo, si es herencia de clase, herencia de interface, o realización de interface
RESTRICCIONES	• Un módulo puede tener múltiples padres, aunque la herencia múltiple es generalmente considerada una aproximación de diseño peligroso • Los ciclos en la relación *generalización* no son permitidos; es decir, un módulo hijo no puede ser generalizado por uno más de sus módulos ancestros en una vista
PARA QUÉ SIRVE	• Expresar herencia en diseños orientados a objetos • Describir la evolución y extensión incrementalmente • Capturar los puntos en común, con variaciones como hijos • Soportar el reúso

Estilo de capas



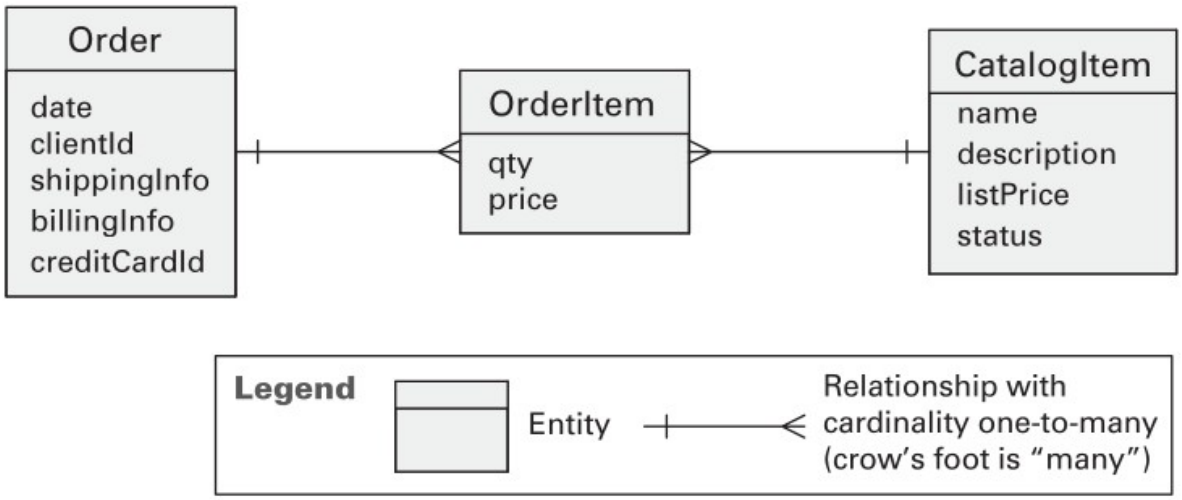
DESCRIPCIÓN	El estilo de capas coloca juntas capas (agrupaciones de módulos que ofrecen un conjunto cohesivo de servicios) en una relación unidireccional *pemitido-usar* unos con otros
COMPONENTES	*Capas*. La descripción de una capa debe definir qué módulos contiene la capa
RELACIONES	*Permitido-usar*, la cuál es una especialización de la relación genérica *depende-de*. El diseño debe definir las reglas de uso de la capa (por ejemplo, “Una capa tiene permitido usar cualquier capa inferior.”) y cualquier excepción permitida
RESTRICCIONES	• Cada pieza de software es asignada a exactamente una capa • Hay por lo menos 2 capas (generalmente 3 o más) • Las relaciones *permitido-usar* no deben ser circulares (es decir, una capa inferior puede usar una capa superior)
PARA QUÉ SIRVE	• Promover la modificabilidad y portabilidad • Administrar la complejidad y facilitar la comunicación de la estructura del código a los desarrolladores • Promover el reuso • Lograr la separación de asuntos

Estilo de aspectos



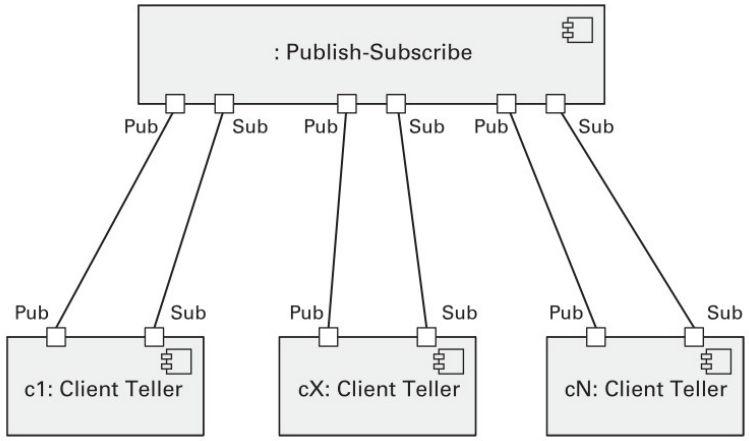
DESCRIPCIÓN	El estilo de aspectos muestra módulos de aspecto que implementan asuntos transversales y como están vinculados a otros módulos en el sistema
COMPONENTES	*Aspecto*, el cuál es un módulo que contiene la implementación de un asunto transversal
RELACIONES	*Transversal*, la cual une un módulo de aspecto a un módulo que será afectado por la lógica transversal de ese aspecto
RESTRICCIONES	• Un aspecto puede ser transversal a uno más módulos regulares así como a módulos de aspectos • Un aspecto que es transversal a sí mismo puede causar recursión infinita, dependiendo la implementación
PARA QUÉ SIRVE	• Modelar asuntos transversales en diseños orientados a objetos • Mejorar la modificabilidad

Modelo de datos



DESCRIPCIÓN	El modelo de datos describe la estructura de las entidades de datos y sus relaciones
COMPONENTES	*Entidad de datos*, la cuál es un objeto que mantiene información que necesita ser almacenada o de alguna manera representada en el sistema. Entre sus propiedades se incluyen nombre, atributos de datos, clave primaria y reglas para garantizar permisos a los usuarios para acceder a la entidad
RELACIONES	• Relaciones *uno-a-uno*, *uno-a-muchos*, y *muchos-a-muchos*, las cuales son asociaciones lógicas entre las entidades de datos • *Generalización/especialización*, la cual indica una relación *es-un* entre las entidades • *Agregación*, la cuál convierte una relación en una entidad agregada
RESTRICCIONES	Las dependencias funcionales deben ser evitadas
PARA QUÉ SIRVE	• Describir la estructura de la data usada en el sistema • Realizar un análisis de impacto de los cambios en el modelo de datos; análisis de extensibilidad • Hacer cumplir la calidad de datos evitando la redundancia e inconsistencia • Guiar la implementación de módulos que acceden a los datos

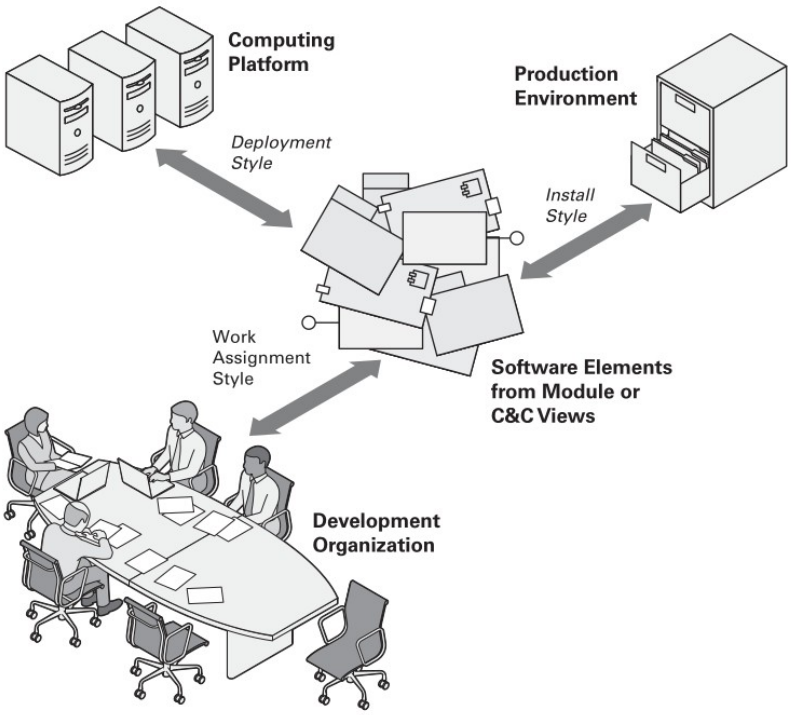
29. Explicar para qué sirve una vista de componentes y conectores



DESCRIPCIÓN	Una vista de componentes y conectores (C&C) muestra elementos que tienen presencia en ejecución, tales como procesos, objetos, clientes, servidores y almacenamientos de datos. Estos elementos son llamados *componentes*. Adicionalmente, las vistas C&C incluyen como elementos los caminos de interacción, tales como enlaces de comunicación y protocolos, flujos de información y accesos a almacenamiento secundario. Estas interacciones son representados como *conectores*.
COMPONENTES	• Componentes: unidades principales de procesamiento y almacenamientos de datos. Un componente tiene un conjunto de *puertos* a través de los cuales interactúa con otros componentes (vía conectores) • Conectores: caminos de interacción entre componentes. Los conectores tiene un conjunto de roles que indican cómo los componentes pueden usar un conector en interacciones.
RELACIONES	• Adjuntos: los puertos de los componentes son asociados con los roles del conector para generar un gráfico de componentes y conectores • Delegación de interface: en algunas situaciones los puertos de los componentes son asociados con uno o más puertos en una subarquitectura “interna”. Similar para los roles de un conector
RESTRICCIONES	• Componentes pueden ser adjuntos solamente a conectores, no a otros componentes • Conectores pueden ser adjuntos solamente a componentes, no a otros conectores. • Adjuntos pueden ser hechos solamente entre roles y puertos compatibles • Delegación de interface puede ser definida solamente entre dos puertos compatibles (o dos roles compatibles) • Los conectores no pueden aparecer aislados; un conector debe estar adjunto a un componente
PARA QUÉ SIRVE	• Mostrar cómo trabaja el sistema • Guiar el desarrollo especificando la estructura y el comportamiento de los elementos en ejecución • Ayudar a los arquitectos y otros a razonar acerca de los atributos de calidad del sistema en ejecución, tales como performance, confiabilidad y disponibilidad

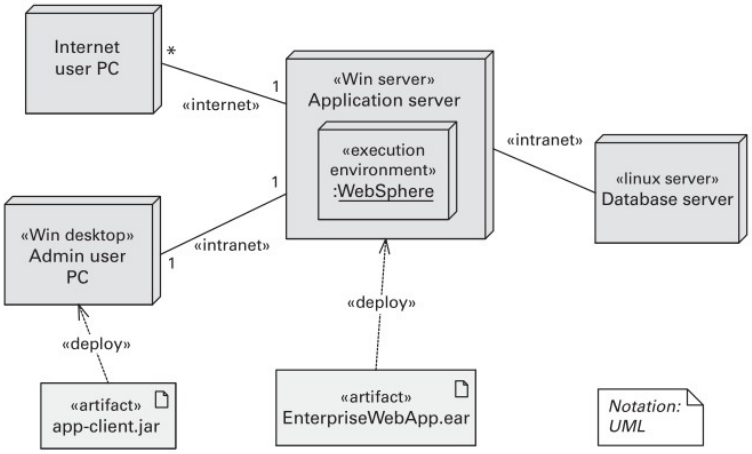
(P. Clements, Documenting Software Architectures, p. 126)

30. Explicar para qué sirve una vista de asignación (a partir de un estilo)



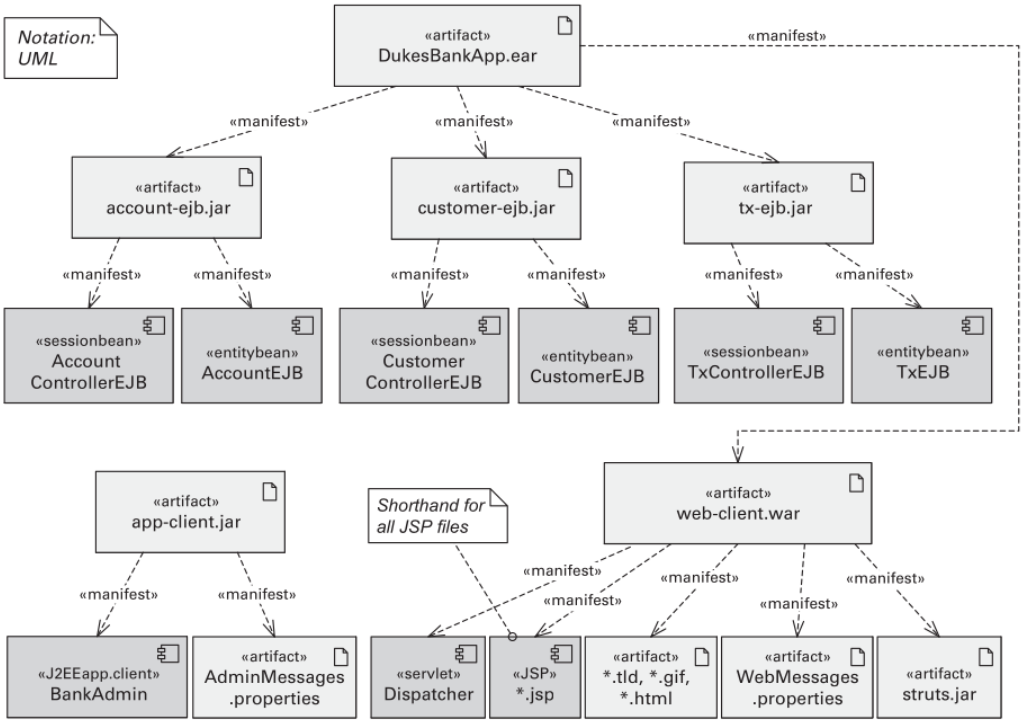
DESCRIPCIÓN	Los estilos de asignación describen el mapeo entre la arquitectura de software y su ambiente
COMPONENTES	• Elemento de software: tiene propiedades que son requeridas del ambiente • Elemento de ambiente: tiene propiedades que son provistas al software
RELACIONES	*Asignado-a*: un elemento de software es mapeado (asignado a) un elemento del ambiente. Las propiedades dependen del estilo en particular
RESTRICCIONES	Varía respecto al estilo
PARA QUÉ SIRVE	

Estilo de deployment (despliegue)



DESCRIPCIÓN	El estilo de deployment describe el mapeo de componentes y conectores en la arquitectura de software al hardware de la plataforma de cómputo
COMPONENTES	• Elemento de software: elementos de la vista C&C. Entre las propiedades útiles a documentar se incluyen las características significativas requeridas del hardware, tales como procesamiento, memoria, requerimientos de capacidad y tolerancia a fallos • Elemento de ambiente: hardware de la plataforma de cómputo (procesador, memoria, disco, red tales como router, banda ancha, firewall y bridge, etc). Las propiedades útiles de un elemento de ambiente son aspectos que influencian en la decisión de asignación
RELACIONES	• *Asignado-a*: unidades físicas en las cuales los elementos de software residen durante la ejecución. Las propiedades incluyen ya sea que la asignación pueda cambiar en tiempo de ejecución o no • *Migra-a, copia-migra-a,* y/o *ejecución-migra-a*: si la asignación es dinámicas. Las propiedades incluyen el disparador que causa la migración
RESTRICCIONES	La topología de asignación es irrestringida. Sin embargo, las propiedades requeridas del software deben ser satisfechas por las propiedades provistas del hardware
PARA QUÉ SIRVE	• Es útil para analizar performance, disponibilidad, confiabilidad y seguridad • Ayuda a la estimación de costos cuando se evalúa las opciones de compra de hardware

Estilo de instalación



DESCRIPCIÓN	El estilo de instalación describe el mapeo de componentes en la arquitectura de software a un sistema de archivos en el ambiente de producción
COMPONENTES	• Elemento de software: un componente C&C. Las propiedades requeridas de un elemento de software, si la hay, usualmente incluye requerimientos en los ambientes de producción, tales como un requerimiento de soporte de Java o una base de datos, o permisos específicos en el sistema de archivos • Elemento de ambiente: un ítem de configuración, como un archivo o una carpeta. Las propiedades provistas de un elemento de ambiente incluyen indicadores de las características provistas por los ambientes de producción.
RELACIONES	• *Asignado-a*: un componente es asignado a un ítem de configuración • *Contención*: un ítem de configuración es contenido en otro
RESTRICCIONES	Los archivos y carpetas son organizados en una estructura de árbol, siguiendo una relación *está-contenido-en*
PARA QUÉ SIRVE	• Entender la organización de los archivos y carpetas del software instalado puede ayudar a los desarrolladores, deployers y operadores en sus tareas • Las propiedades requeridas de los elementos de software pueden ser usados para ayudar en el análisis de opciones de compra para ambientes de producción

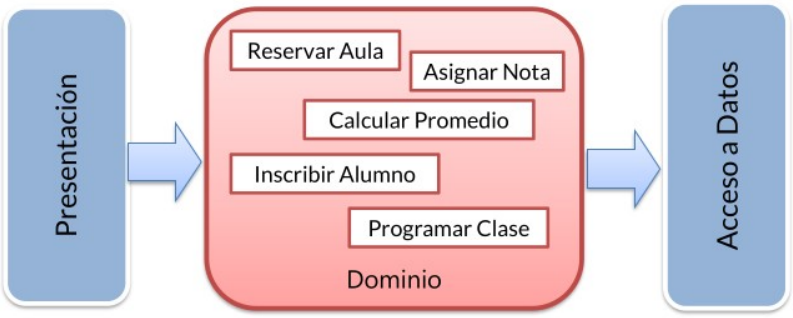
Estilo de asignación de trabajo

ECS Element (Module)		
Segment	**Subsystem**	**Organizational Unit**
Science Data Processing Segment (SDPS)	Client	Science team
	Interoperability	Prime contractor team 1
	Ingest	Prime contractor team 2
	Data Management	Data team
	Data Processing	Data team
	Data Server	Data team
	Planning	Orbital vehicle team
Flight Operations Segment (FOS)	Planning and Scheduling	Orbital vehicle team
	Data Management	Database team
	User Interface	User interface team
. . .	. . .	. . .

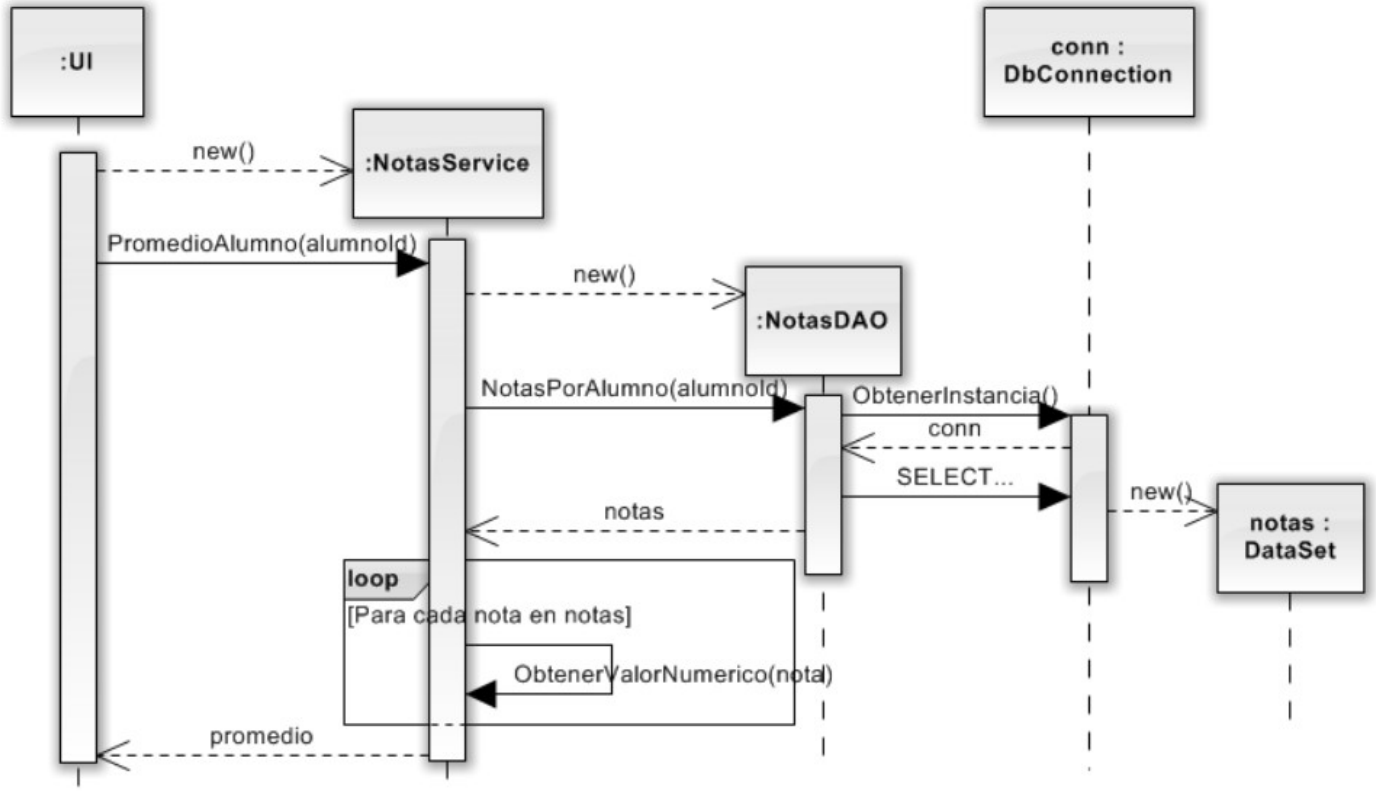
DESCRIPCIÓN	El estilo de asignación de trabajo describe el mapeo de la arquitectura de software a los equipos en la organización de desarrollo
COMPONENTES	• Elemento de software: un módulo. Las propiedades incluyen el conjunto de habilidades requeridas, y la capacidad de esfuerzo y tiempo necesarias • Elemento de ambiente: una unidad organizacional, tales como una persona, un equipo, un departamento, un subcontratado, étc. Las propiedades incluyen el conjunto de habilidades provistas y la capacidad en términos de trabajo y tiempo calendario disponible
RELACIONES	*Asignado-a*: un elemento de software es asignado a una unidad organizacional
RESTRICCIONES	En general, la asignación es irrestringida; en la práctica es usualmente restringida así un módulo es asignado a una unidad organizacional
PARA QUÉ SIRVE	• Muestra las unidades de software más importantes que deben estar presentes para formar un sistema de trabajo y quién lo produce, así como las herramientas y ambientes en los cuales el software es desarrollado • Ayuda con la planificación y administración de las asignaciones de los recursos del equipo, asignando responsabilidades para builds, y explicando la estructura de un proyecto

31. Explicar un patrón de aplicaciones empresariales de la capa de dominio

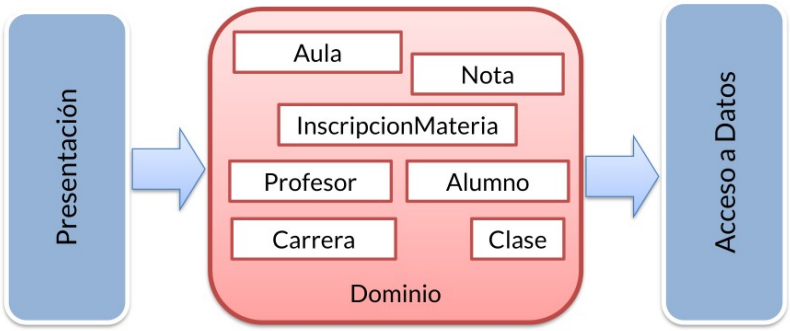
Transaction script



- *Descripción:* Organiza la lógica de negocio por procedimiento, donde cada procedimiento maneja un único pedido desde la capa de presentación. Un Transaction Script se podría resumir en validar los datos de entrada, consultar la base de datos, realizar cálculos y guardar los resultados en la base de datos.
- Ventajas
  - Dada su simplicidad, es natural utilizarlo en aplicaciones que tengan poca lógica de negocios, ya que involucra poco overhead tanto en performance como en curva de aprendizaje
- Limitaciones
  - A medida que la lógica de negocio se torna más compleja, es difícil mantenerlo con un buen diseño
  - Generalmente tiene problemas con la duplicación de código. Puesto que el foco está en resolver una transacción, muchas veces se tiende a duplicar código común.



Domain Model



- *Descripción:* Crea una red de objetos interconectados, donde cada objeto representa un concepto significativo, ya sea tan grande como una corporación o tan pequeño como un ítem de una factura. Domain Model mezcla datos y procesos, tiene atributos multivaluados, tiene una compleja red de asociaciones y usa herencia.
- *Tipos*
  - Simple: luce muy parecido al modelo de bases de datos, normalmente con un objeto de dominio por cada tabla de la base de datos. Puede utilizar el patrón Active Record para el acceso a datos
  - Rico: puede lucir diferente al modelo de base de datos, incorporando herencia, strategies, otros patrones de diseño y redes complejas de pequeños objetos interconectados. Requiere utilizar el patrón Data Mapper para el acceso a datos
- *Uso*
  - Complejidad de comportamiento del sistema: si tenemos reglas de negocio complicadas y cambiantes que involucran validaciones, cálculos, etc., sería preferible la utilización de un Domain Model
  - Comodidad del equipo en el uso de objetos de dominio: aprender a diseñar y usar un Domain Model es un ejercicio significativo que lleva práctica y coaching.
  - Posibilidad de utilizar un Data Mapper para el acceso a datos: en algunas circunstancias no es posible o lleva mucho esfuerzo utilizar un Data Mapper y esto complica la utilización de Domain Model.

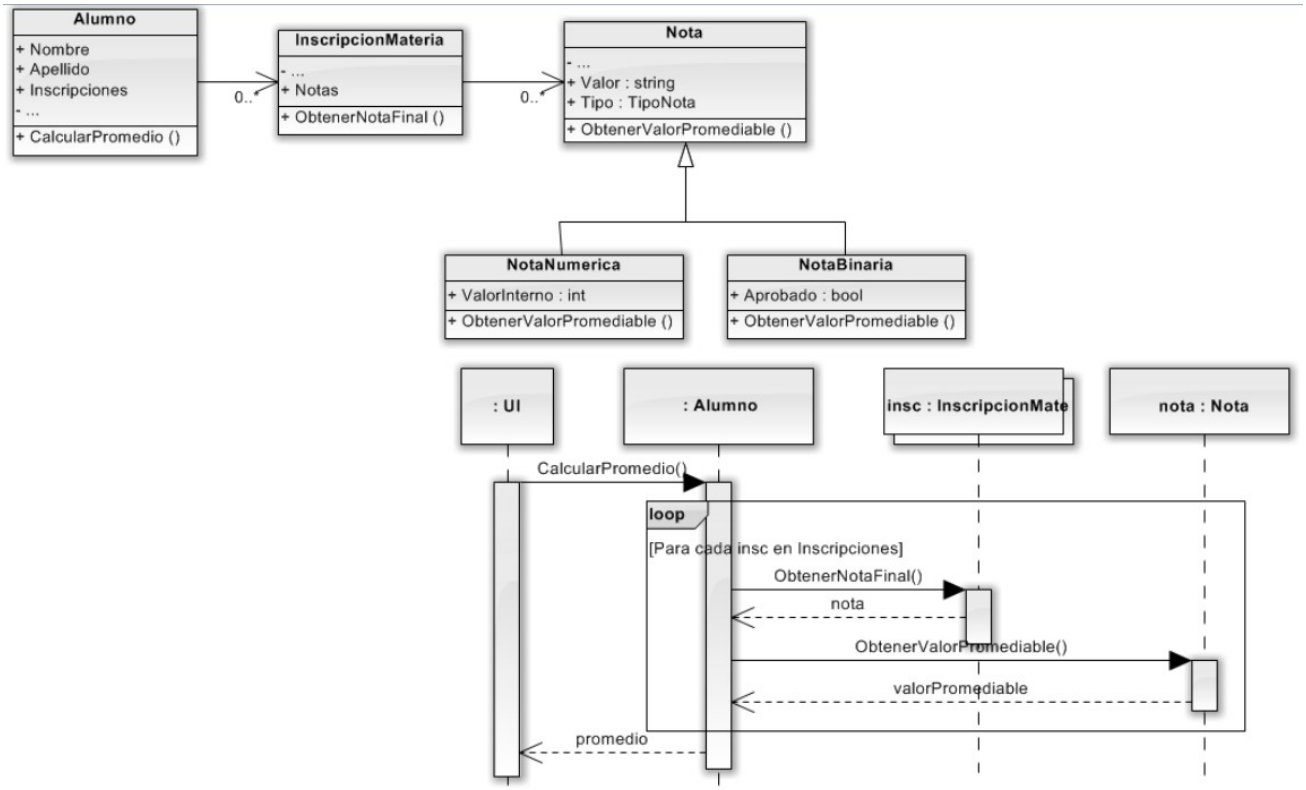
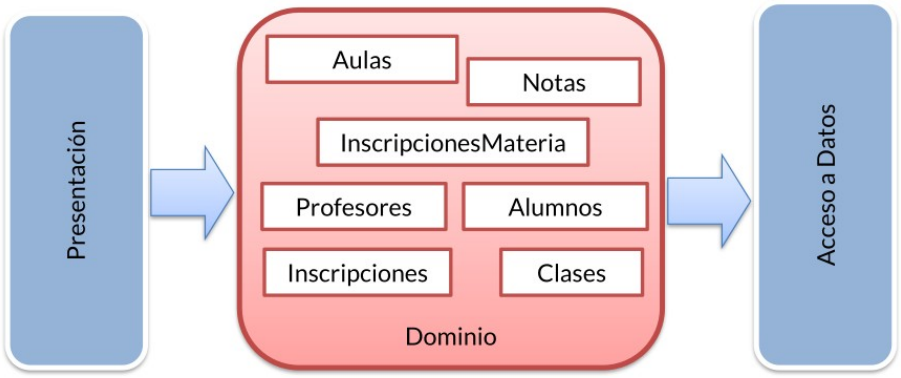
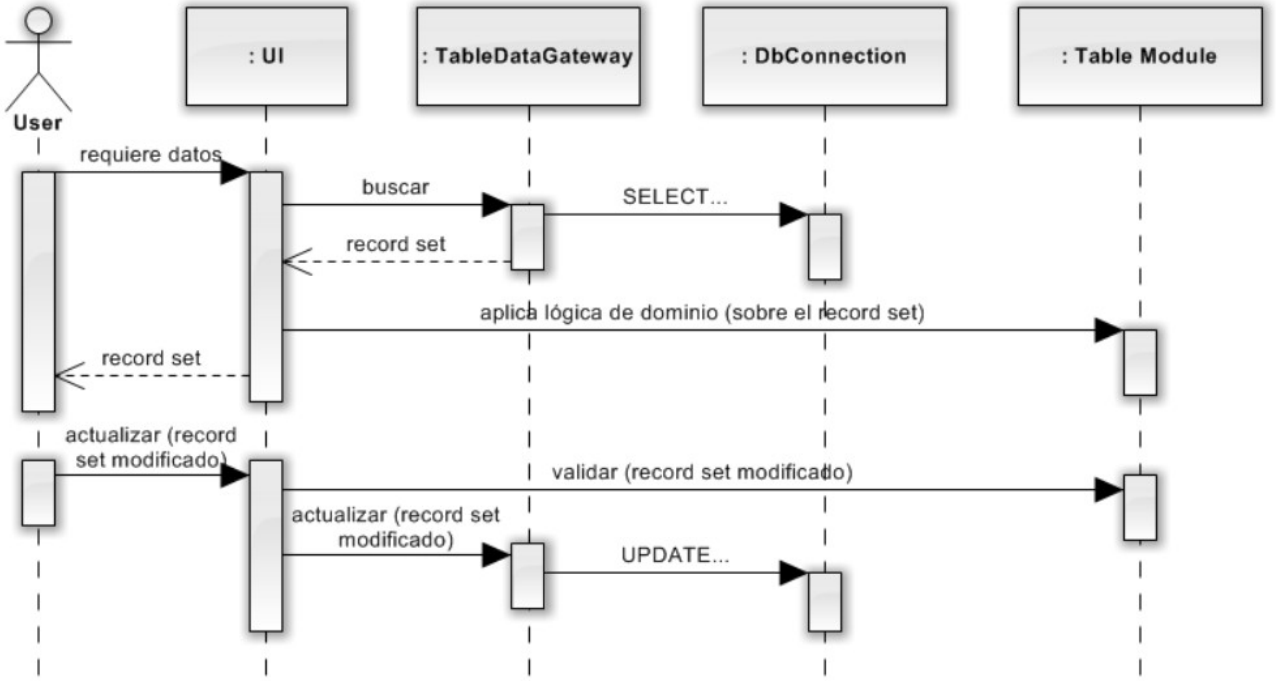


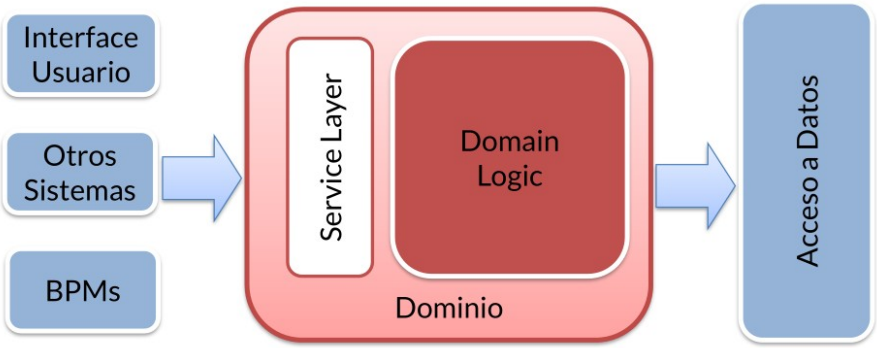
Table Module



- *Descripción:* Se basa en tener una única instancia que maneje la lógica de dominio para todas las filas en una tabla o vista de la base de datos. Un Table Module organiza la lógica de dominio con una clase por tabla de base de datos, y una única instancia de una clase contiene los distintos procedimientos que actuarán sobre los datos. Así se logra también agrupar los datos con el comportamiento que los utiliza
- *Características*
  - Sin identidad: Table Module no tiene noción de identidad de los objetos con los que trabaja. Por ejemplo, para obtener un nombre de un alumno se debe ejecutar: `String nombreAlumno = alumnosModule.obtenerNombre(long alumnoID)`
  - Estructura de datos de respaldo: generalmente se usa un Table Module con una estructura de datos de respaldo que sea orientada a tablas. Normalmente estos datos tabulares resultan de una consulta SQL y se mantiene en un Record Set que imita la tabla en la base de datos.
- *Ventajas*
  - Dado que está basado en la manipulación de datos orientados a tablas, cobra mayor sentido utilizarlo cuando se está accediendo a datos tabulares usando Record Set
  - Trabaja mejor que Domain Model + Active Record cuando otras partes de la aplicación están basadas en una estructura de datos orientada a tablas
- *Desventajas*
  - En casos de alta complejidad de lógica de negocios, Domain Model es una mejor opción
  - Con Table Module se pierde el poder de la orientación a objetos en organizar lógica compleja (no se pueden tener relaciones entre instancias ni polimorfismo)



Service Layer



- *Descripción*: define el límite de una aplicación con una capa de servicios que establecen el conjunto disponible de operaciones y coordina la respuesta de la aplicación en cada operación. Service Layer factoriza la lógica de aplicación en una capa separada, promoviendo los beneficios conocidos de layering y produciendo clases de negocio puras y más reusables de aplicación en aplicación.
- *Razones para proveer una Service Layer*
  - Proveer una API sobre la lógica de negocio
  - Manejar transacciones a través de múltiples recursos: por ejemplo, transacciones distribuidas; y coordinar distintas respuestas para una operación (workflows, lógica de interacción con otros sistemas)
- *Desventajas:*
  - Las clases de dominio son menos reutilizables entre aplicaciones
  - Mezclar ambos tipos de lógica en una misma clase hace más difícil reimplementar la lógica de aplicación en, por ejemplo, una herramienta de workflows.
  - Tratar con distribución de objetos puede ser muy complicado
  - Trabajo extra con Data Transfer Objetcts (DTOs): el costo de transformar la entrada y salida de la Service Layer en DTOs puede ser muy alto, especialmente para dominios complejos.

32. *Explicar un patrón de aplicaciones empresariales de la capa de datos que sirva para modelar el comportamiento objeto-relación*

Unit of Work

- *Descripción:* Mantiene una lista de objetos afectados por una transacción de negocio y coordina la escritura de los cambios y la resolución de problemas de concurrencia. Con cada cambio en el modelo de objetos se podría actualizar la base de datos, sin embargo, esto sería poco eficiente ya que requeriría muchas llamadas a la base de datos, además de que se necesitaría una transacción abierta para la interacción completa, lo cuál no es práctico para transacciones de negocio que implican múltiples pedidos.
- *¿Cómo trabaja?*
  - Cuando se comienza a trabajar con la base de datos, se crea un Unit of Work
  - Unit of Work mantiene registro de todos los objetos creados/modificados/eliminados durante la transacción de negocio
  - Al finalizar la transacción de negocio, Unit of Work decide qué hacer:
    - Abrir una transacción de base de datos
    - Realizar chequeos de concurrencia
    - Escribir cambios a la base de datos en el orden apropiado
    - Cerrar la transacción de base de datos

Identity Map

- *Descripción*: asegura que cada objeto sea cargado solamente una vez, manteniendo cada objeto cargado en un map. Cuando se referencia a un objeto, lo busca a través del map.
- *¿Cómo trabaja?*
  - Mantiene registro de todos los objetos que han sido leídos desde la base de datos en una única transacción de negocio
  - Cuando se quiere obtener un objeto, primero se busca en el map por si ya está. Si no está, se busca en la base de datos y se lo incorpora al map.
  - También sirve como caché de objetos obtenidos en una transacción de negocio.

Lazy load

- *Descripción:* un objeto que no contiene todos los datos que se necesitan, pero sabe cómo obtenerlos. Soluciona el problema de performance debido a que la carga de un objeto provoque cargar una gran cantidad de objetos relacionados
- *¿Cómo trabaja?*

  Interrumpe el proceso de carga de objetos relacionados al objeto que se quiere recuperar, dejando una marca en la estructura de objetos tal que si el datos es necesario, pueda ser cargado solamente cuando sea usado.

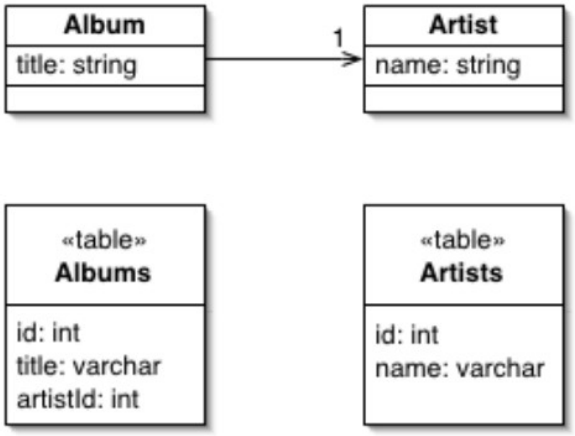
33. *Explicar un patrón de aplicaciones empresariales de la capa de datos que sirva para modelar el mapeo estructural objeto-relación*

Identity Field

Descripción: guarda el campo ID de la base de datos en el objeto para mantener la identidad entre un objeto en memoria y la fila de la base de datos. La forma más simple de Identity Field es un campo que concuerde con el tipo de la clave en la base de datos. Para que funcionen, las claves deben ser únicas e inmutables.

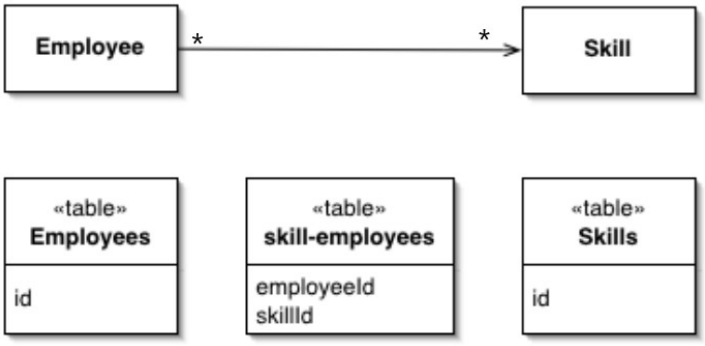
- Meaningful key: por ejemplo, el número de documento de una persona
- Meaningless key: número aleatorio que no sería utilizado por humanos
- Clave simple: solamente se usa un campo de la base de datos
- Clave compuesta: usa más de un campo de la base de datos. Generalmente se utiliza una clase que las representa
- Únicas por tabla
- Únicas por base de datos

Foreign Key Mapping



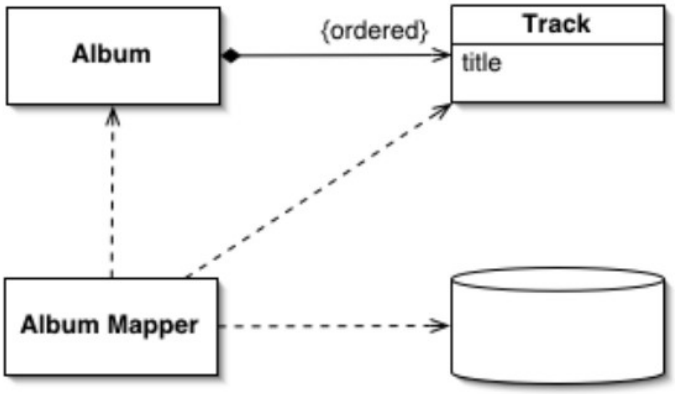
Descripción: mapea una asociación entre objetos a una clave foránea entre tablas.

Association Table Mapping



Descripción: guarda una asociación en una tabla con claves foráneas a las tablas que están relacionadas por la asociación. El caso de las relaciones muchos a muchos no puede solucionarse con un patrón Foreign Key Mapping, en este caso, se recomienda usar el Association Table Mapping.

Dependent Mapping

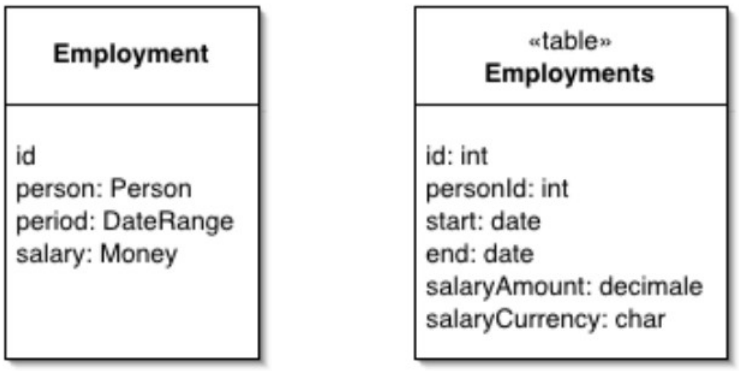


Descripción: Una clase tiene que realizar el mapeo de base de datos para una clase hija.

El objeto dependiente:

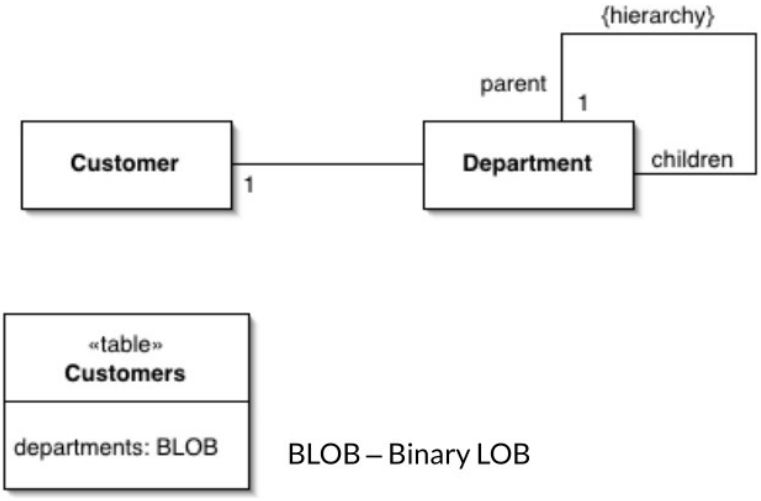
- Sólo tiene un owner
- No puede hacer referencias a él desde otro objeto que no sea su owner
- No tiene un Identity Field
- No es almacenado en el Identity Map
- No puede ser buscado directamente
- Sólo es accedido desde su owner
- Generalmente es un Value Object

Embedded Value



Descripción: Mapea un objeto en varios campos de la tabla de otro objeto. Muchos pequeños objetos tienen sentido en un sistema orientado a objetos, sin embargo, no tienen como tablas en una base de datos. Por ejemplo, objetos relacionados con tipo de moneda y rango de fechas. Se mapean entonces los valores de un objeto (por ejemplo, period) a campos (por ejemplo, start, end) en el registro del propietario del objeto.

Serialized LOB



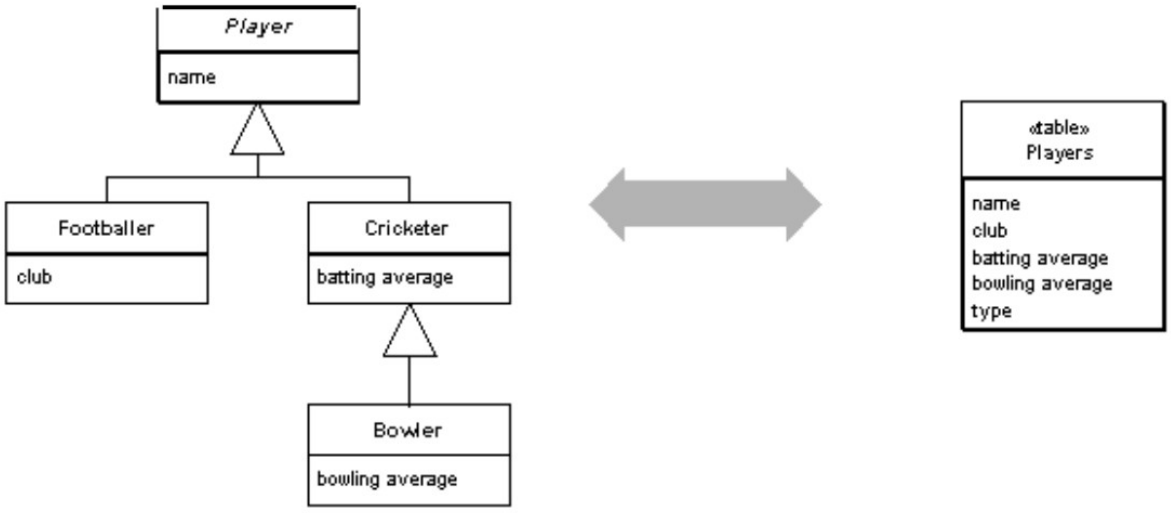
Descripción: Guarda un grafo de objetos serializándolos en un único objeto grande (LOB), el cuál se almacena en un campo de la BD.

Tipos de serialización:

- BLOB: Binary LOB
- CLOB: Textual characters LOB

Se diferencia con Embedded Value en que está orientado a estructuras de objetos más complejas

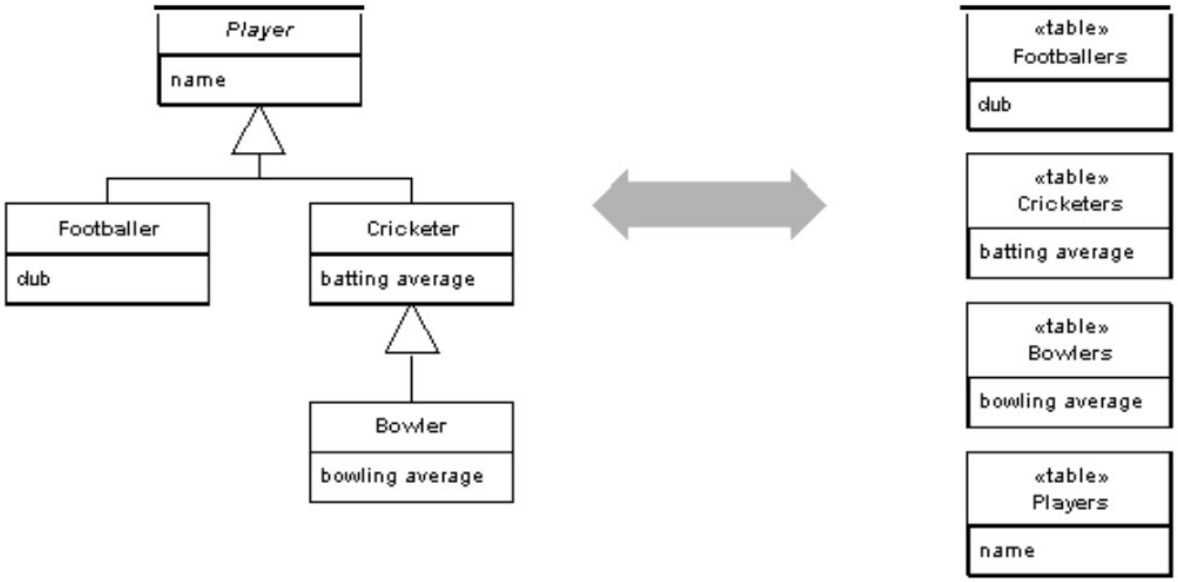
Single Table Inheritance



- *Descripción:* representa una jerarquía de herencia de clases como una única tabla que tiene columnas para todos los campos de las clases de la jerarquía. Existe una tabla que contiene todos los datos para todas las clases de la jerarquía. Cuando se carga un objeto a memoria, se necesita conocer qué clase instancia, por lo que se necesita un campo extra en la tabla que indique qué clase debiera utilizarse para esa fila
- *Ventajas*
  - ✔ Habrá una única tabla por la que preocuparse
  - ✔ No se realizan joins para recuperar información
  - ✔ Cualquier refactoring que eleve/baje campos de la jerarquía no requiere que se cambie la base de datos

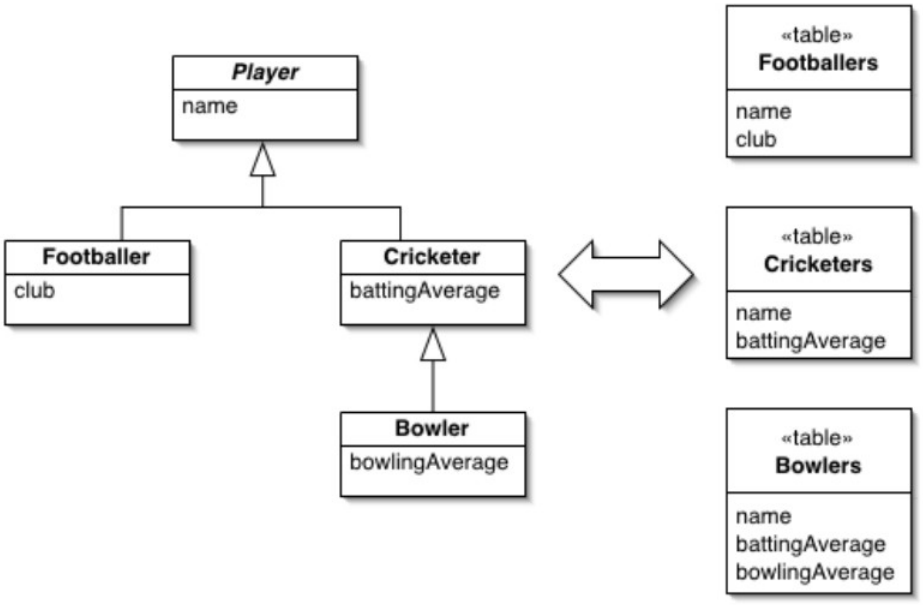
- *Desventajas*
  - ✘ Los nombres de columnas a veces son relevantes y a veces no, lo cual puede confundir a personas que accedan directamente a la base de datos
  - ✘ Las columnas que sólo son utilizadas por algunas subclases conducen a desperdiciar espacio
  - ✘ La tabla podría terminar siendo muy grande, con muchos índices y frecuentes bloqueos, lo cual afectaría a la performance
  - ✘ Sólo se tiene un único namespace para los nombres de los campos, con lo cual se debe asegurar que no existan dos campos en la jerarquía con el mismo nombre

Class Table Inheritance



- *Descripción:* Representa una jerarquía de herencia de clases con una tabla para cada clase
- *¿Cómo trabaja?*
  - Existe una tabla por cada clase en la jerarquía de herencia
  - Los campos en la clase de dominio se mapean directamente a los campos de la tabla correspondiente
  - ¿Cómo enlazar las filas en distintas tablas que representan a un objeto concreto de la jerarquía?
    - Usar un valor de clave primaria común a la jerarquía
    - Cada tabla tiene sus propias claves primarias. Entonces, usar foreign keys en la tabla de la superclase para apuntar al resto de las filas
- *Ventajas*
  - ✔ Todas las columnas son relevantes para cada fila. Por lo tanto, las tablas son fáciles de entender y no desperdician espacio
  - ✔ La relación entre el modelo de dominio y la base de datos es directa
- *Desventajas*
  - ✘ Se necesita acceder a múltiples tablas para cargar un objeto
  - ✘ Cualquier refactoring que eleve/baje campos de la jerarquía causa cambios en la base de datos
  - ✘ Las tablas de los supertipos podrían convertirse en un cuello de botella ya que serían accedidas con mucha más frecuencia que las tablas de las hojas de la jerarquía
  - ✘ La alta normalización puede hacer que sea difícil entender los queries

Concrete Tabla Inheritance

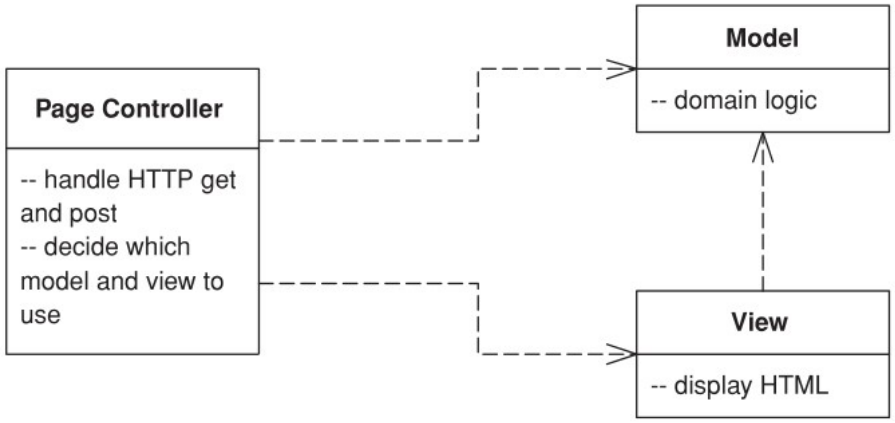


- *Descripción*: representa una jerarquía de herencia de clases con una tabla por clase concreta de la jerarquía.
  - Existe una tabla para cada clase concreta de la jerarquía de herencia
  - Cada tabla contiene las columnas para los campos propios de la clase concreta y para los campos de todos sus ancestros. Por lo tanto, los campos de las superclases son duplicados en las tablas de sus subclases concretas
  - Se necesitan claves que sean únicas para toda la jerarquía
- *Ventajas*
  - ✔ Cada tabla es auto-contenida y no tiene campos no relevantes. Esto es favorable cuando las tablas serán utilizadas también por otros sistemas que no estén usando objetos
  - ✔ No se necesitan joins para leer datos
  - ✔ Cada tabla es accedida solamente cuando la clase correspondiente es accedida
- *Desventajas*
  - ✘ Las claves primarias pueden ser difíciles de manejar
  - ✘ No se pueden forzar relaciones de base de datos a clases abstractas
  - ✘ Cualquier refactoring que eleve/baje campos de la jerarquía causa cambios en la base de datos, aunque no tantos como en Class Table Inheritance
  - ✘ Si cambia un campo de una superclase, se deben modificar todas las tablas que contengan dicho campo
  - ✘ Una búsqueda sobre la superclase fuerza a chequear todas las tablas

34. Explicar un patrón de aplicaciones empresariales de la capa de presentación

(Patterns of Enterprise Application Architecture, M. Fowler, p. 333)

Page Controller



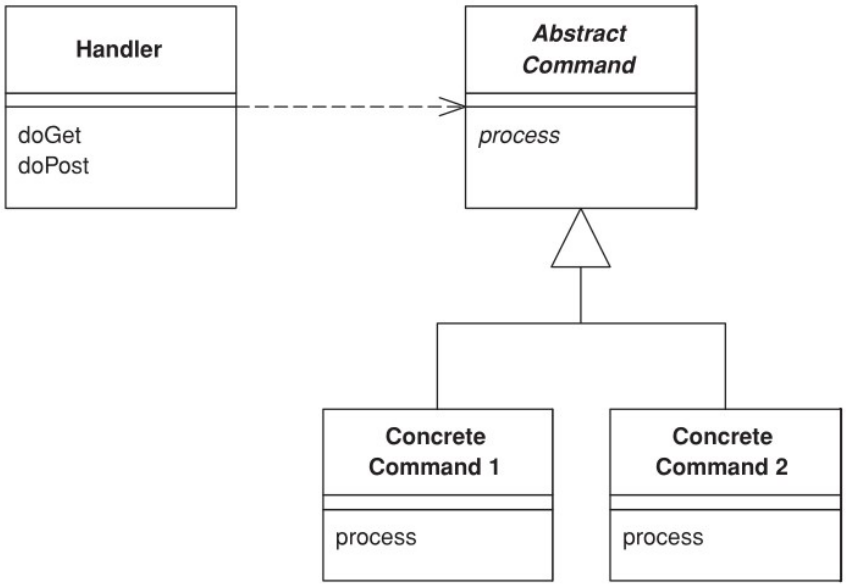
- *Descripción:* un objeto que maneja una request para una página específica o acción sobre un sitio web
- *¿Cómo funciona?*

  La idea básica detrás de Page Controler es tener un módulo en el servidor web que actúe como controlador para cada página en el sitio web. Las responsabilidades básicas de un Page Controller son:

  - Decodificar la URL y extraer cualquier forma de datos para averiguar toda la información para la acción
  - Crear e invocar cualquier objeto del modelo para procesar la información. Toda información relevante desde la request HTML debe ser pasada al modelo así los objetos del modelo no necesitan ninguna conexión con la request HTML
  - Determinar cuál vista debe mostrar la página resultante y reenviarle la información del modelo
- *¿Cuando usarlo?*

  Page Controller funciona particularmente bien en sitios donde la mayoría de la lógica de controlador es simple.

Front Controller



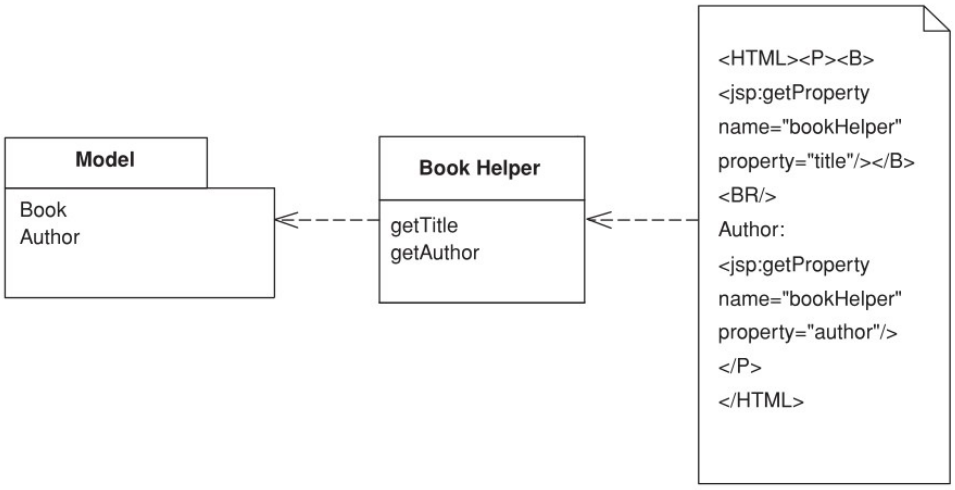
- *Descripción:* un controlador que maneja todos los request para un sitio web. Front Controler consolida todo el manejo de requests canalizando las requests a través de un sólo objeto manejador. Este objeto puede realizar un comportamiento común, el cuál puede ser modificado en tiempo de ejecución con decoradores. Luego el manejador envía objetos de comando para un comportamiento en particular de una request.
- *¿Cómo funciona?*

  Un Front Controller maneja todas las llamadas para un sitio web, y es usualmente estructurado en dos partes: un manejador web y una jerarquía de comandos. El manejador web es el objeto que de hecho recibe las request POST o GET desde el servidor web. Extrae suficiente información de la URL y la request para decidir qué tipo de acción debe ser iniciada y entonces la delega a un comando que realizará la acción
- *¿Cuando usarlo?*

  Una ventaja de un Front Controller es que permite factorizar el código que sería de otra manera duplicado en Page Controller.

  Front Controller tiene un solo controlador, así que es fácil de mejorar su comportamiento en tiempo de ejecución con decoradores. Puedes tener decoradores para autenticación, codificación de caracteres, internacionalización, étc, y agregarlos usando un archivo de configuración o incluso cuando el server está ejecutándose.

Template View



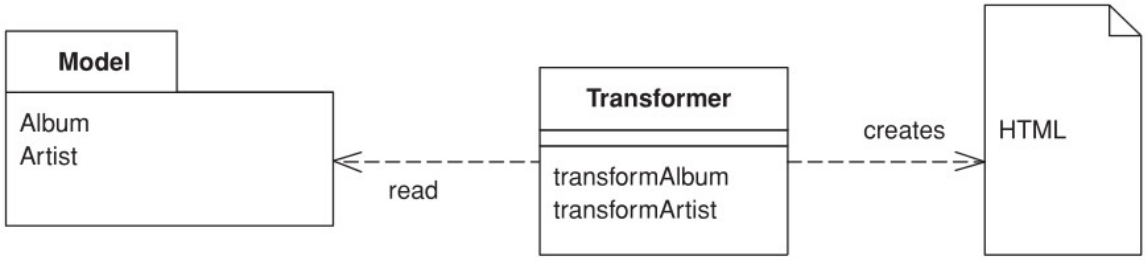
- *Descripción:* renderiza información en un HTML incrustando marcadores en una página HTML. La mejor manera de componer una página web dinámica es hacerla estática pero colocar marcadores que puedan resolverse en llamadas para recolectar información dinámica. Como la parte estática de la página actúa como un template para una respuesta particular, la llamamos Template View
- *¿Cómo funciona?*

  La idea básica de un Template View es incrustar marcadores en la página HTML estática cuando es escrita. Cuando la página es usada para atender una request, los marcadores son remplazados por los resultados de alguna computación, como una consulta a la base de datos.
- *¿Cuando usarlo?*

  La principal fortaleza de Template View es que permite componer el contenido de la página mirando la estructura de la página. En particular ayuda a la idea de un diseñador gráfico diseñando una página con un programador trabajando en el helper.

  Template View tiene dos debilidades significativas. Primero, la implementación común hace muy fácil colocar lógica complicada en la página, haciéndola difícil de mantener, especialmente por no programadores. Debe tenerse buena disciplina para mantener la página simple y orientada a ser mostrada, colocando la lógica en el helper. La segunda debilidad es que Template View es difícil de testear. La mayoría de las implementaciones de Template View son diseñadas para trabajar con un servidor web y son muy difíciles o imposibles de testear.

Transform View



- *Descripción:* una vista que procesa la información de dominio elemento a elemento y los trasforma en HTML.

  Cuando se emiten solicitudes de información a las capas de dominio y datos, se devuelven los datos necesarios que las satisfacen, pero sin el formateo necesario para hacer una página web. Usar Transform View significa pensar en esto como una transformación donde se tiene los datos del modelo como entrada y su HTML como salida.

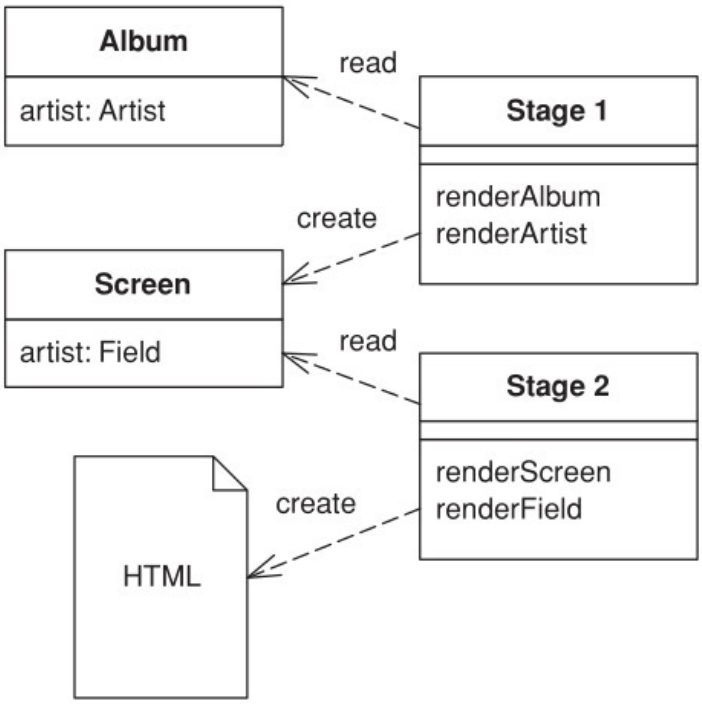
- *¿Cómo funciona?*

  La noción básica de Transform View es escribir un programa que busque información orientada al dominio y la convierta a HTML. El programa recorre la estructura de datos del dominio y, así como reconoce cada forma de datos del dominio, escribe la pieza de HTML particular para él.

- *¿Cuando usarlo?*

  Transform View evita dos de los grandes problemas de Template View. Es fácil mantener la transformación focalizada solamente en el renderizado HTML, de este modo evitar tener demasiada lógica en la vista. También es fácil correr Transform View y capturar las salidas para testing. Esto hace fácil testear la vista y no es necesario un servidor web para correr los tests.

Two Step View



- *Descripción:* convierte los datos de dominio en HTML en dos pasos: primero formando una especia de página lógica, y luego renderizarla a HTML.

  Si se tiene una aplicación web con muchas páginas, usualmente se busca una apariencia y organización consistente para el sitio. Además, se busca que sea fácil hacer un cambio global a la apariencia del sitio. Two Step View se ocupa de esto dividiendo la transformación en dos etapas. La primera transforma los datos del modelo en una presentación lógica sin ningún formato específico; la segunda convierte esa presentación lógica con el formato real necesitado. De esta manera pueden hacerse cambios globales alterando la segunda etapa.

- *¿Cómo funciona?*

  La clave de este patrón es hacer la transformación a HTML en un proceso de dos etapas. La primer etapa ensambla la información en una estructura lógica de pantalla que aún no contiene HTML. La segunda etapa toma esa estructura orientada a la presentación y la renderiza en HTML.

  Esta estructura orientada a la presentación es ensamblada por el código escrito específicamente para cada pantalla. La responsabilidad de la primer etapa es la de acceder al modelo orientado al dominio (por ejemplo, una base de datos), extraer la información relevante para esa pantalla y luego colocar esa información en una estructura orientada a la presentación.

  La segunda etapa transforma esa estructura orientada a la presentación en HTML. De este modo, un sistema con muchas pantallas puede ser renderizado a HTML por una sola segunda etapa así todas las decisiones de formateo de HTML son hechas en un sólo lugar.

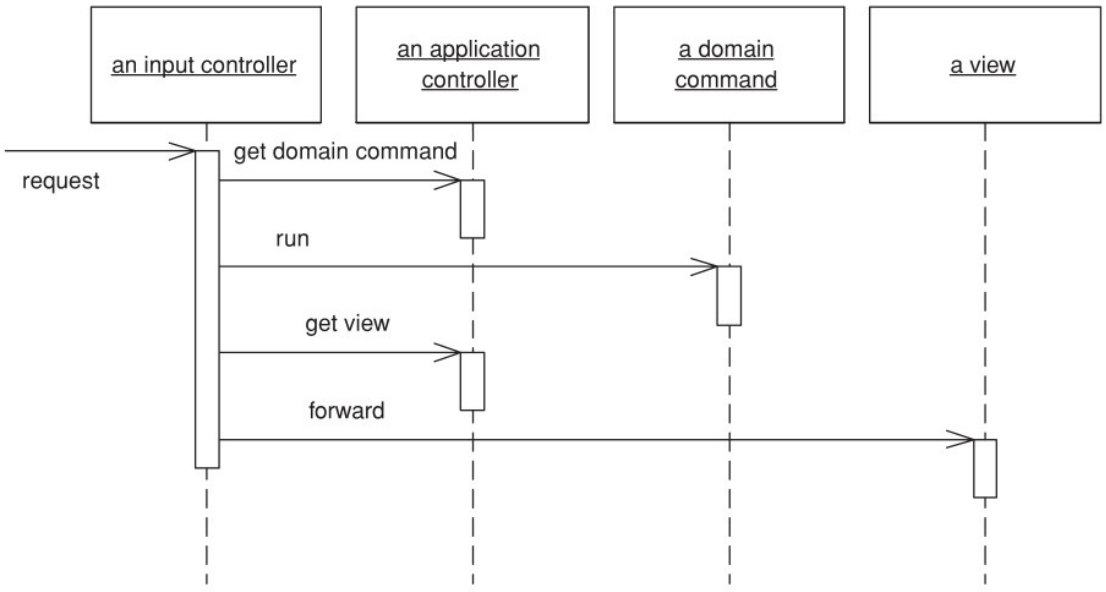
- *¿Cuando usarlo?*

  El valor clave de Two Step View viene de la separación de la primera y segunda etapas, permitiendo hacer cambios globales fácilmente. Esto ayuda a pensar en dos situaciones: aplicaciones web multiapariencia y monoapariencia.

En aplicaciones monoapariencia tienes dos etapas: un módulo de primer etapa por página y un único módulo de segunda etapa para toda la aplicación. Cualquier cambio en la apariencia del sitio en la segunda etapa es mucho más fácil de hacer, ya que los cambios en la segunda etapa afectan a todo el sistema.

Con una aplicación multiapariencia esta ventaja es más notoria porque se tiene una única vista para cada combinación de pantallas y apariencias. De este modo, mientras que 10 pantallas con 3 apariencias requiere 30 módulos en patrones de una sola etapa, usando Two Step View requiere solamente 10 primeras etapas y 3 segundas etapas. Cuando más pantallas y apariencias se tiene, mayor es el ahorro.

Application Controller



- *Descripción:* un punto centralizado para manejar la navegación de la pantalla y el flujo de una aplicación.

  Cuando la aplicación tiene un estilo de interacción wizard, donde el usuario es guiado a través de una serie de pantallas en un orden específico; o se ven pantallas que son mostradas bajo ciertas condiciones; o las elecciones entre diferentes pantallas dependen de entradas anteriores, la aplicación se vuelve compleja y puede llevar a tener código duplicado, ya que varios controladores de diferentes pantallas necesitan saber qué hacer bajo ciertas situaciones. Puede eliminarse esta duplicación colocando toda la lógica del flujo en un Application Controller. Los controladores de entrada entonces le preguntan al Application Controller por los comandos apropiados para la ejecución contra el modelo y la vista correcta a usar dependiendo del contexto de la aplicación.

- *¿Cómo funciona?*

  Un Application Controller tiene dos responsabilidades principales: decidir qué lógica del dominio correr y decidir la vista con la cual mostrar la respuesta. Para lograr esto, típicamente mantiene dos colecciones estructuradas de referencias a clases, una para comandos de dominio para ejecutar en la capa de dominio y una para vistas. Para ambos, el Application Controller necesita una manera de almacenar algo que pueda invocar.

  Una aplicación puede tener múltiples Application Controllers para manejar cada una de sus distintas partes. Esto permite separar lógica compleja en varias clases.

- *¿Cuando usarlo?*

  La fortaleza de un Application Controller viene de definir reglas acerca del orden en el cuál las diferentes páginas deben ser visitadas y vistas dependiendo del estado de los objetos.

  Una buena señal para usar un Application Controller es si te encuentras teniendo que hacer cambios similares en muchos lugares diferentes cuando el flujo de tu aplicación cambia.