



Resumen teórico

⌚ Created	@July 7, 2022 8:51 PM
🕒 Class	Técnicas de Diseño
🕒 Type	Resumen
☑ Completed	<input type="checkbox"/>
🕒 Status	Materia Aprobada

Modelo de Dominio

Teoría

Patrones de análisis o colaboración

Ejercicio: Carga telefónica

Ejercicio: RoboDoc

Arquitectura del Software

Teoría

Patrones de arquitectura y diseño

Criterios generales de calidad de diseño

Patrones

Ejercicio: Sistema Locu

Ejercicio: Punto de venta

Ejercicio: Documentos

Resolución de Parcial

Tema 1 - Running

Tema 2

Paradigmas de Programación

Teoría

Patrones de Diseño

Teoría

Creacionales (factory, factory method, builder)

Extensión de interfaz/objeto

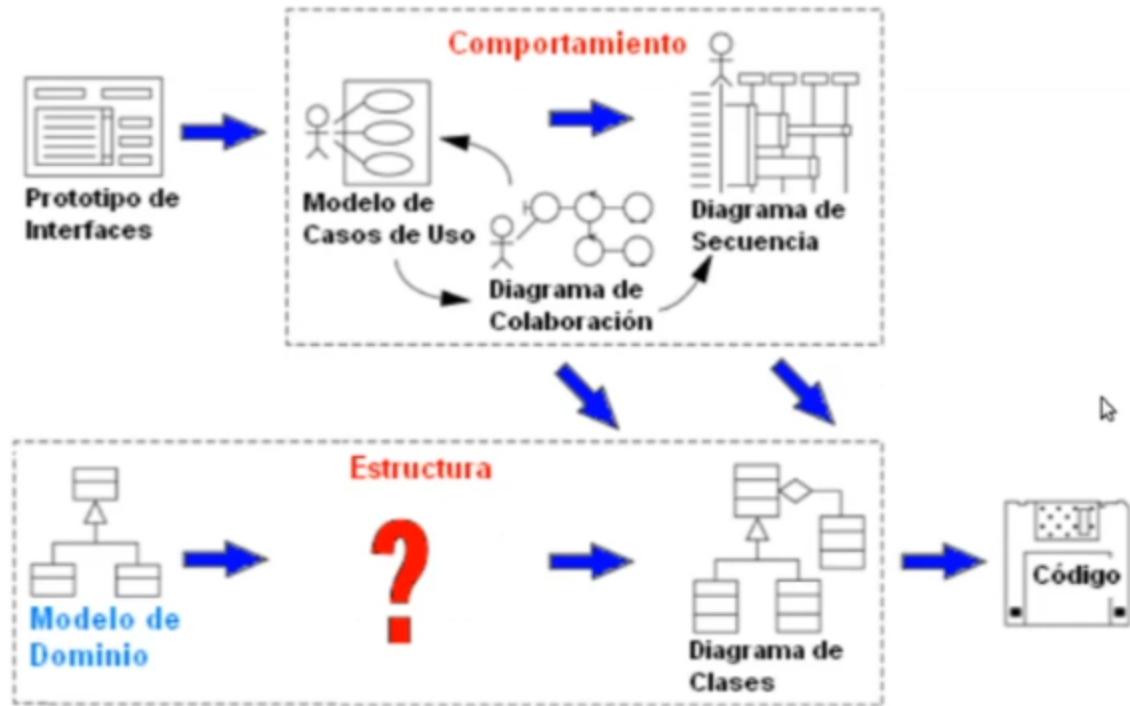
Facade

[Bridge](#)
[Estado](#)
[Estrategia](#)
[Método template](#)
[Decorador](#)
[Visitor](#)
[Adapter](#)
[Composite](#)
[Machete](#)
[Métricas](#)
[Teoría](#)

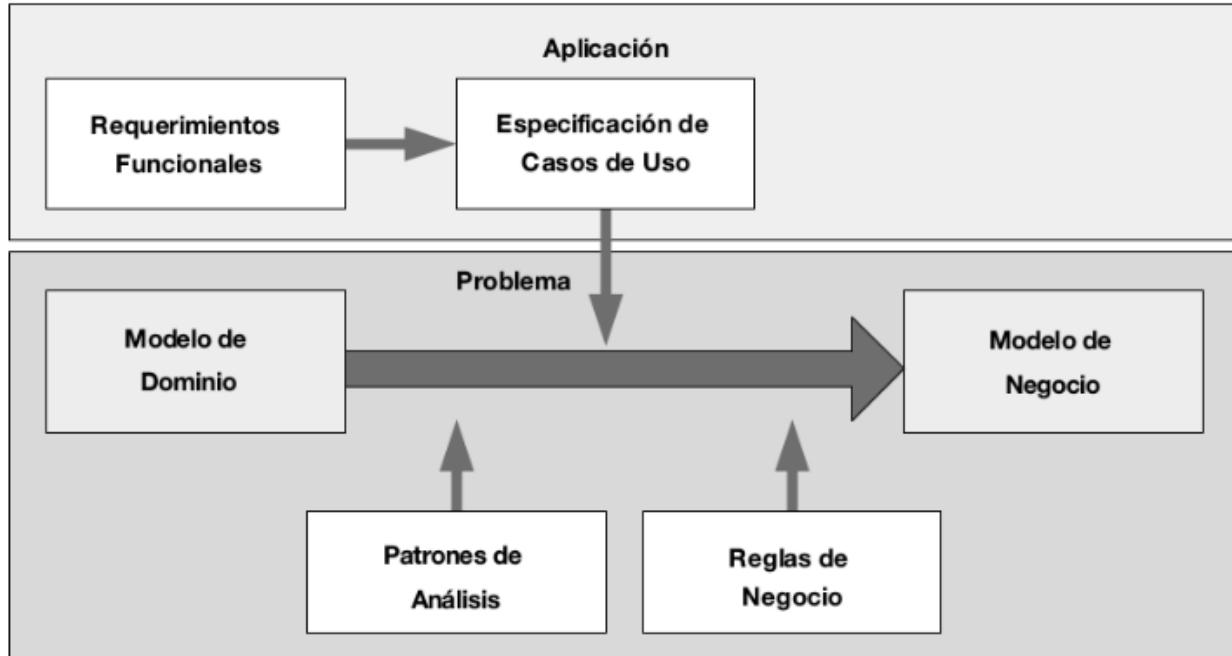
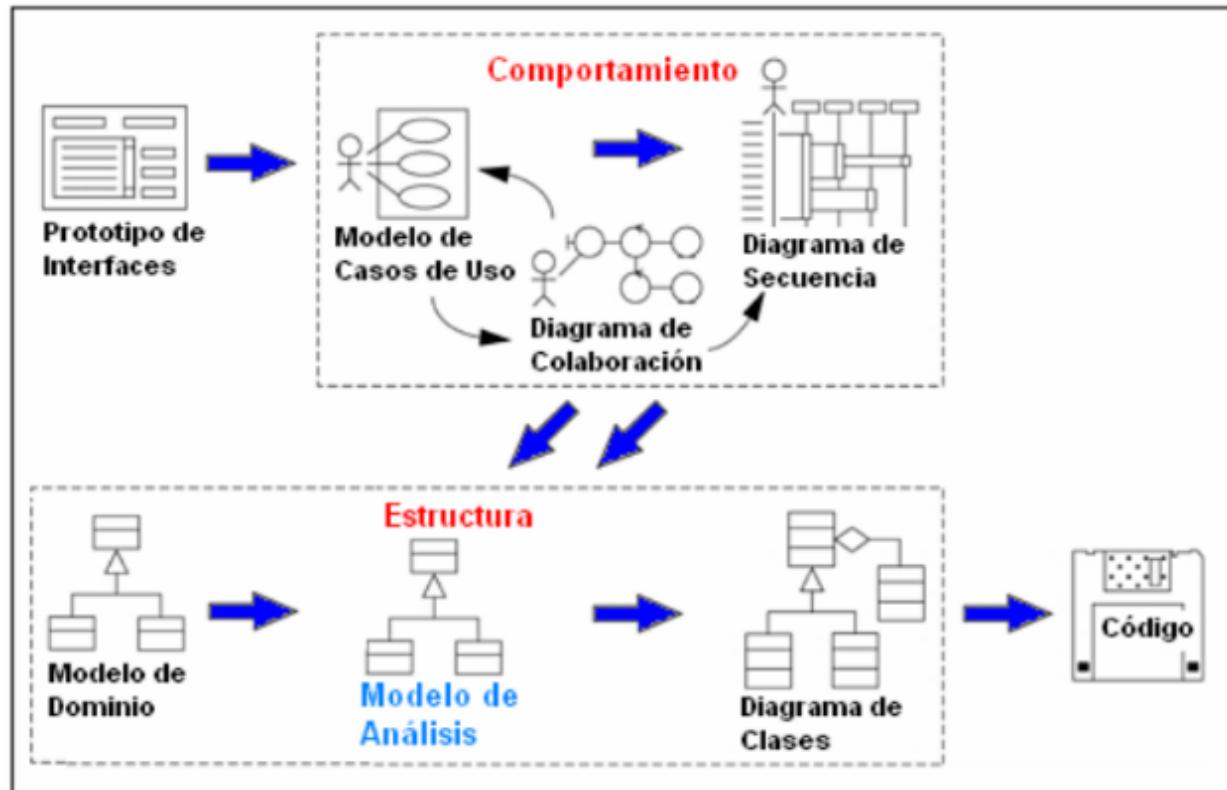
Modelo de Dominio

Teoría

Un caso de uso es una especificación de un pedazo de **funcionalidad** que un sistema tiene que implementar. Cuando especificamos un sistema, lo registramos y damos forma a partir de un modelo de casos de uso. Formando parte del modelo de casos de uso, hay un diagrama que en general es acompañado por prototipos de interfaces y con algún tipo de diagrama que muestre la dinámica. Eventualmente los requerimientos funcionales definen las acciones de un negocio, estas acciones de alguna manera van a tener impacto en objetos materializados en memoria de algun servidor y algunos de ellos terminaran persistidos en alguna base de datos → este modelo de objetos está representando el **modelo del negocio**.



El modelo de dominio esta compuesto por conceptos extraídos del dominio del problema, muchos de ellos van a terminar siendo objetos de nuestro modelo de objetos. Agregando el comportamiento, obtenemos en modelo de análisis, es el modelo de dominio + el comportamiento, o sea las clases con métodos/funciones. El modelo de dominio es fundamental para entender el negocio y sus reglas; el mecanismo que vamos a utilizar es el de Patrones de análisis o colaboración.

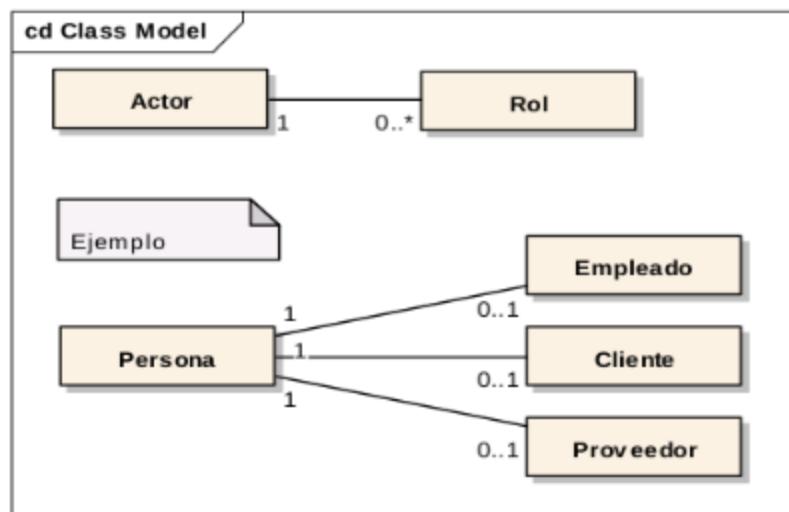


El modelo de diseño tiene el objetivo de implementar una solución al problema planteado en el análisis más las restricciones impuestas por los requerimientos no funcionales; el mecanismo que usaremos es el de patrones de diseño.

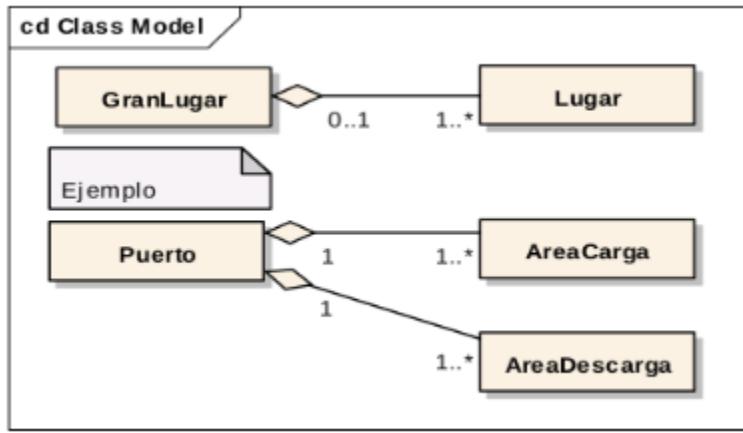
Patrones de análisis o colaboración

Conceptos a buscar	Instancias
Gente	Actor Rol
Lugares	Lugar Gran Lugar
Cosas	Ítem Ítem Específico Ensamble Parte Contenedor Contenido Grupo Miembro
Eventos	Transacciones Transacciones Compuestas Transacciones Cronológicas Line Ítem

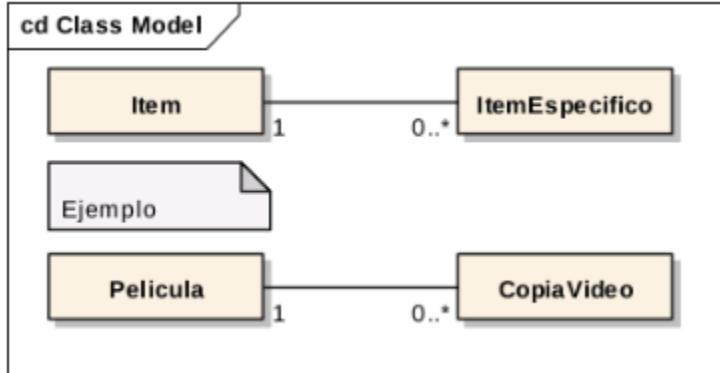
Ejemplos



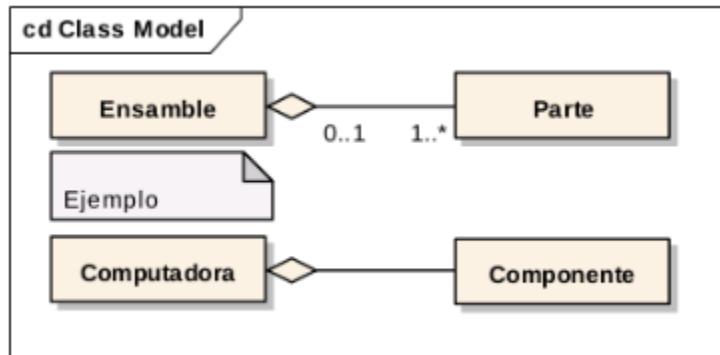
En general se presentan como pares de clases. Se lee como que un actor puede tener uno o muchos roles, pero un rol es asignado solo a un actor.



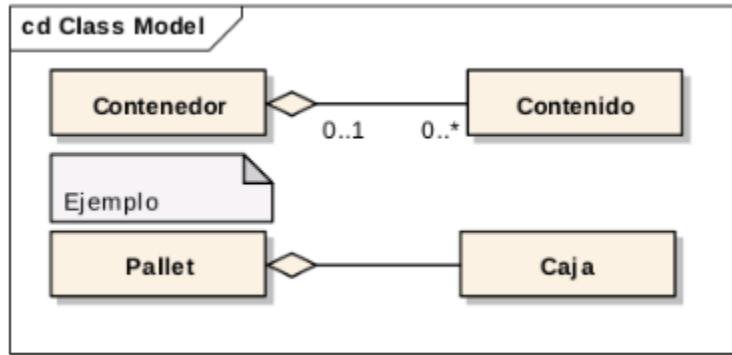
Es una composición de los distintos lugares en un gran lugar. Un lugar puede formar parte de uno o ningún gran lugar.



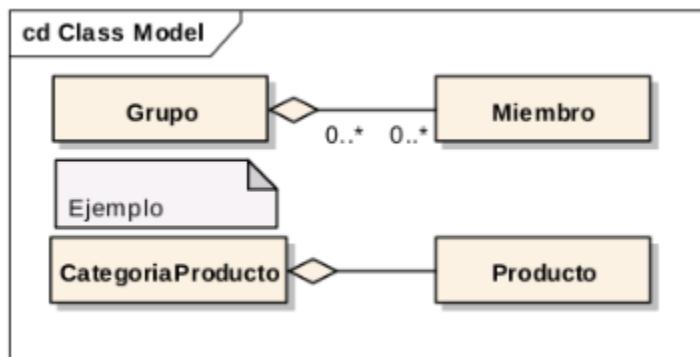
De un item puedo tener ninguno o muchos items específicos, un item específico solo pertenece a un item.



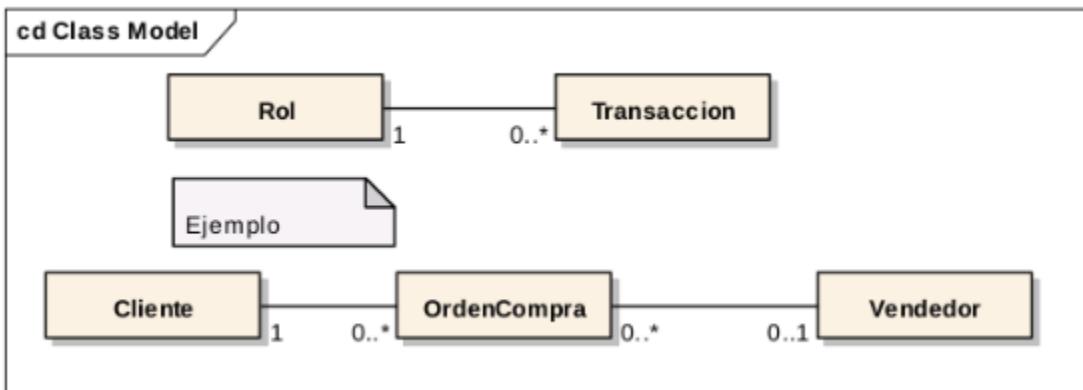
Otra composición.

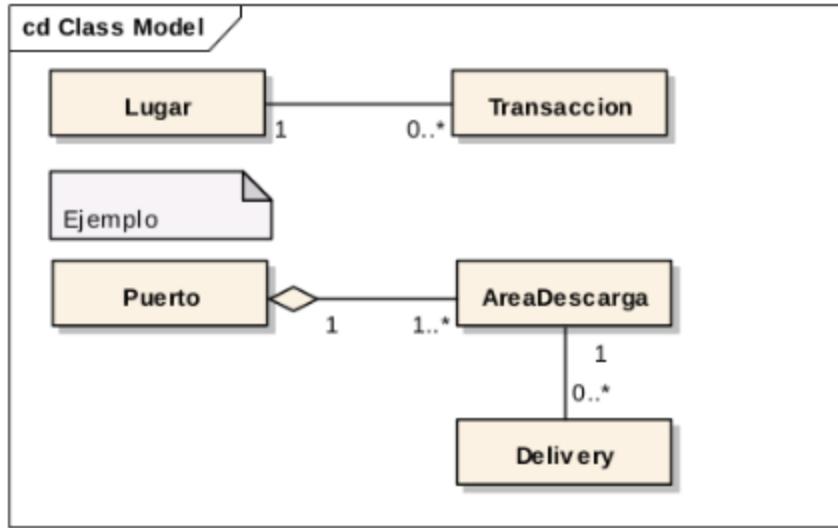


Es una agregación. Un contenedor puede estar vacío o tener muchos contenidos. Un contenido puede estar en ningún contenedor o en uno.

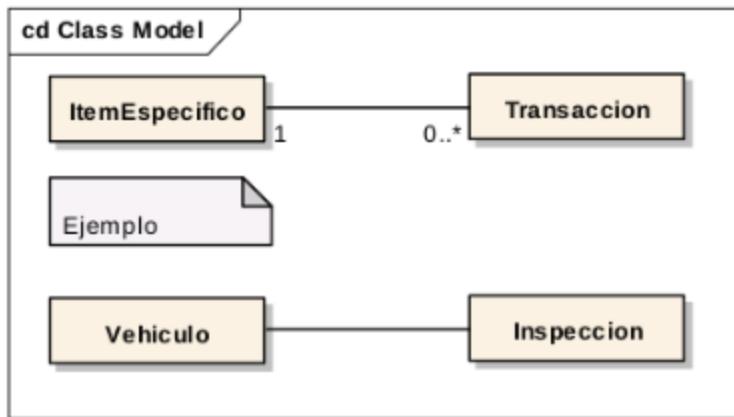


También es una agregación.

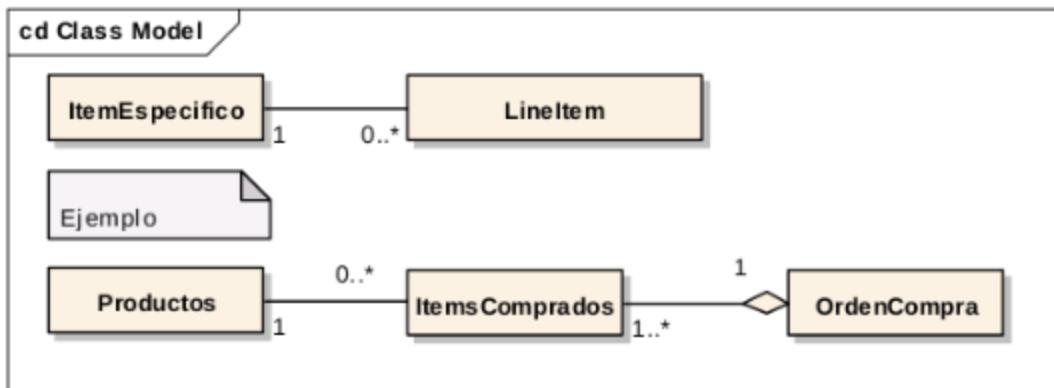




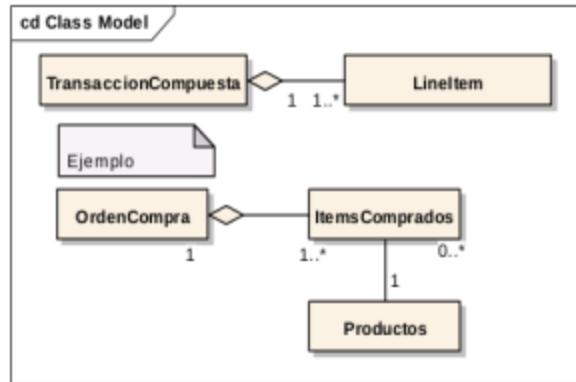
En un lugar se pueden hacer una o muchas transacciones y una transacción se puede hacer en un solo lugar.



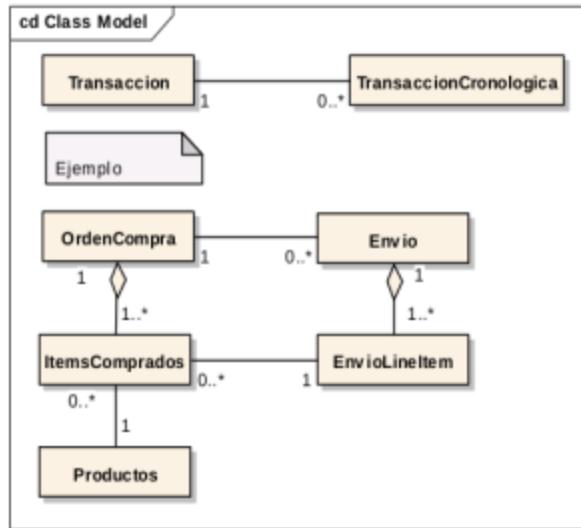
Toda transacción se da sobre un determinado objeto (item específico)



Item específico es el producto y el lineitem los productos comprados.



Por ejemplo un carrito de supermercado con lineitems asociado a una determinada compra.

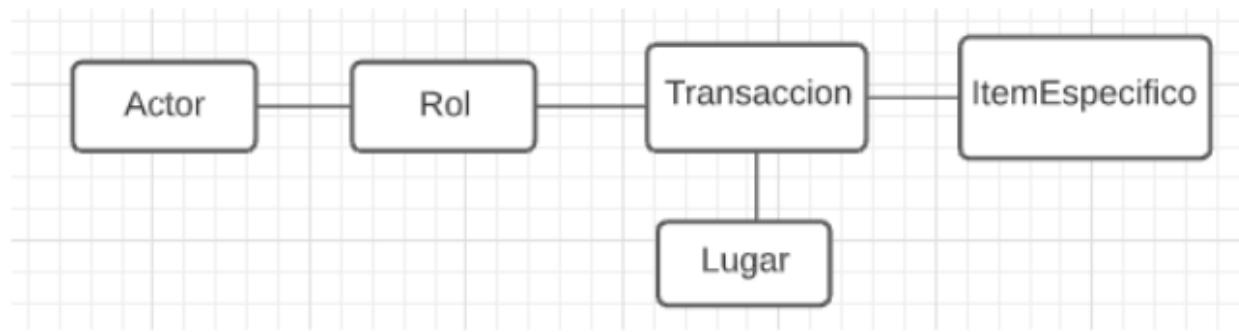


Pasos para su búsqueda

1. Preguntarse quiénes realizan tareas en este dominio/negocio, de esta forma se identifican actores y se les asigna un rol
2. Preguntarse qué tareas realizan, así se asignan transacciones
3. Preguntarse sobre qué objetos de negocio se realiza estas transacciones, así se asignan items específicos

4. Preguntarse qué transacciones deben realizarse antes que otras, así se asignan transacciones cronológicas
5. Preguntarse dónde se realizan las transacciones, así se asigna un lugar a estas transacciones
6. Preguntarse si los Items tienen identificación o no, para identificar si se usa Lines Items o no
7. No confundir transacciones de negocio con transacciones de sistema.

La base que tiene que quedar en todo modelo de dominio es la siguiente:



La forma de colaborar es a partir de la validación de las **reglas de negocio**. Las reglas de negocio son las restricciones que gobiernan las acciones dentro de un dominio determinado. Conducen por dónde pueden evolucionar las transacciones de un determinado negocio. Son el comportamiento que se le asignan a las entidades del modelo de dominio para terminar obteniendo un modelo de análisis, o sea son los métodos de validación de los objetos.

Tipos de reglas

- Tipo: un medicamento puede ser cargado solo en un container refrigerado
- Multiplicidad: un pallet refrigerado puede contener hasta 10 cajas
- Propiedad (validación, comparación): un pago debe registrar un número válido de tarjeta de crédito, la temperatura de un container refrigerado debe ser menor a 0 grados centígrados
- Estado: una orden no debe ser entregada si antes fue cancelada

- Conflicto: un vuelo no puede ser programado en una puerta en un mismo horario que otro vuelo, un producto no puede ser sumado a una orden de compra de un menor de edad si la venta está prohibida a menores.

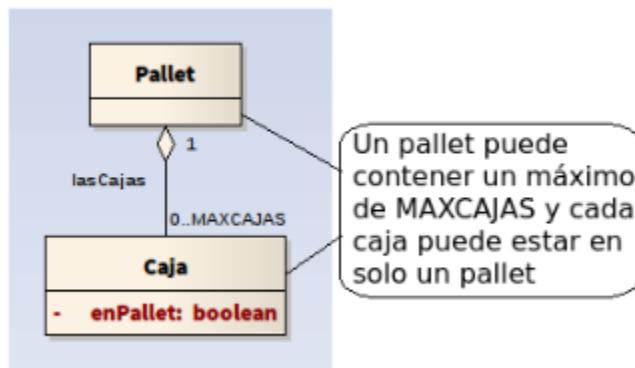
Reglas de colaboración: chequeo de reglas de negocio entre objetos participantes de las relaciones. El cambio de estado, el establecimiento de una relación o la ruptura de la misma requiere el chequeo de una regla de negocio. Este chequeo lo vemos como una colaboración entre objetos

Estrategia de asignación de validación a las clases de los patrones

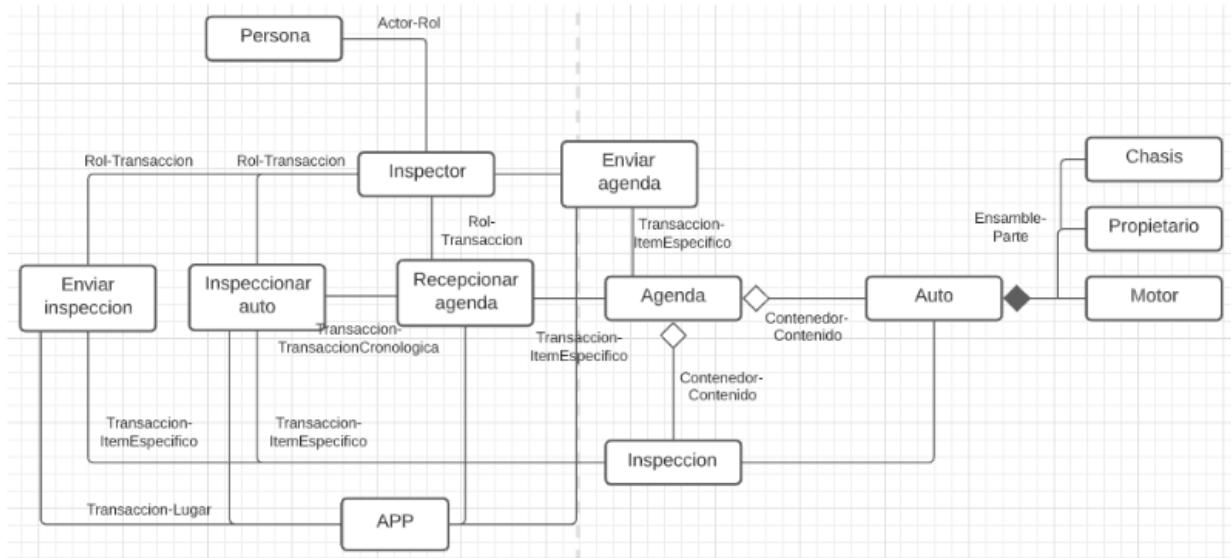
Regla	Patrón de colaboración											
	A-R	GL-L	I-IE	E-P	C-C	G-M	T-R	T-L	T-IE	TC-LI	LI-IE	TC-T
Tipo	R	L	IE	P	C	M	R	L	IE	LI		T
Multiplicidad	R	GL, L	I, IE	E, P	C, C	G, M	T, R	T, L	T, IE	TC, LI	LI, IE	TC, T
Propiedad	R	GL, L	IE	E, P	C, C	G, M	R	L	IE			T
Estado	R	GL, L	I, IE	E, P	C, C	G, M	R	L	IE			T
Conflicto	R	L	IE	P	C	G, M	R	L	IE			T

Por ejemplo, si tenemos dos clases vinculadas a partir de Actor-Rol, cualquier tipo de regla va a ser validada por la clase Rol porque el Actor no participa en la validación, lo hace solo el Rol ⇒ no hay colaboración.

Ejemplo



Ejercicio: Carga telefónica



Ejercicio: RoboDoc

Enunciado

Una empresa de desarrollos electrónicos planea lanzar al mercado el instrumento universal de diagnóstico que ha bautizado con el nombre de “RobotDoc”. Para este proyecto nos ha contratado el desarrollo del software y nos ha pedido una primera impresión.

El aparato será una especie de robot ambulante y tendrá como parte de su arquitectura una computadora donde se ejecutará el software, una carcaza que le dará usabilidad desde el punto de vista ergonómico y un componente electrónico que permite conectarse con otros instrumentos que no poseen medios de comunicación standard como las computadoras (RJ45, USB, etc.). Está orientado a centros de salud que no poseen un sistema integrado de información.

Se pensó a “RobotDoc” como un **recolector de información** de los diferentes consultorios/laboratorios donde se realizan los siguientes estudios: **electroencefalograma, electrocardiograma, resonancia magnética, rayos X y electromiograma**. En estos casos el aparato se conecta con otros instrumentos y **consume datos** tales como señales muestreadas. En el caso de los laboratorios se conecta con las computadoras y consume datos de informes.

El **operador** se desplaza a cada lugar y al tomar los datos **ingresa información del paciente** de manera que al final del recorrido por diferentes áreas del hospital queda agrupada la información por paciente. Es decir que **para cada paciente habrá un conjunto de estudios** con los cuales los diferentes **médicos** que lo atiendan podrán

elaborar un diagnóstico utilizándola.

“RobotDoc” es compartido por varios médicos, los cuales acceden a él y elaboran su diagnóstico con el enfoque de su especialidad. Para este fin “RobotDoc” debe presentar la funcionalidad de “**diagnosticar**” con enfoques preestablecidos para cada especialidad (gastroenterología, cardíaco, clínico, neurológico, etc.). Los diagnósticos no podrán ser realizados si no fue recolectada la información de todos los estudios solicitados.

Aquellos estudios que daten de más de 15 días deberán repetirse, es decir que quedan invalidados para ser utilizados en la elaboración de un diagnóstico, solo serán usados como informativos. Cada paciente tendrá una **historia clínica** conformada por todos los **estudios** realizados y los **diagnósticos** realizados.

La información de los diferentes estudios deben poder relacionarse a partir de un conjunto determinado de algoritmos que aún no fueron entregados. Esta funcionalidad es una especie de asistencia al médico que permiten detectar diferentes fenómenos, medir distintas magnitudes, correlacionar sucesos de diferentes estudios, procesar señales e imágenes.

Los diagnósticos incluirán posibles causas, explicaciones, recomendaciones y prescripciones realizadas por el médico especialista en base a la información recolectada y procesada. Los profesionales para utilizarlos deben registrarse previamente y serán agrupados por el administrador según su especialidad. Las secretarías de las áreas del hospital podrán acceder a los estudios e imprimirlas para ser entregados a los pacientes.

Todos los usuarios deben estar agrupados a efectos de poder acceder al uso. Se cree que la experiencia compartida por profesionales de la salud con diferentes visiones y sugerencias respecto de las interfaces de usuario así como la aparición de nuevos instrumentos a conectar generará al año de uso la producción de un segundo release del software que nos encargan.

Toda la información de “RobotDoc” se almacena en un servidor del hospital, aunque este es el límite de nuestro contrato, es decir no incluye el desarrollo de ninguna parte de este servidor.

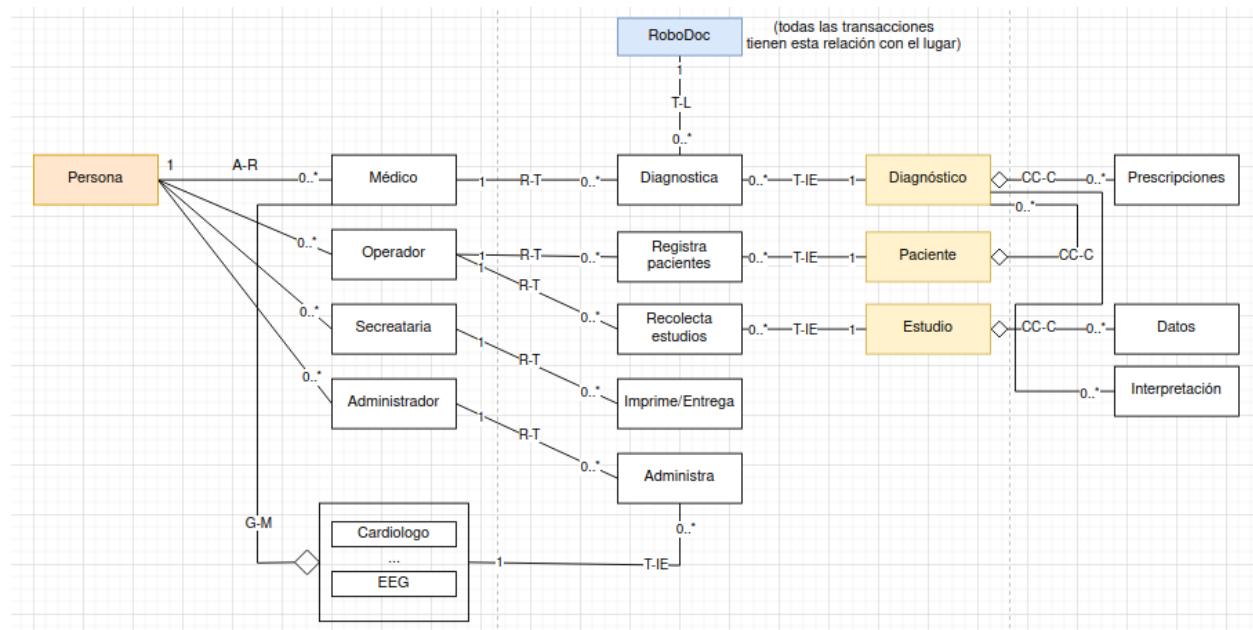
Hemos decidido elaborar, para presentar a la empresa, un Modelo de Dominio que muestre el

funcionamiento de este negocio para validarla y una definición preliminar de la Arquitectura donde se justifique cada una de las partes incluidas.

Un año después ... hay que agregar la funcionalidad de realizar diagnósticos con enfoque de medicina del deporte para lo cual es necesario conectarse con un aparato

de electrocardiogramas de esfuerzo diferente a todos los manejados hasta ahora. También incluiremos los cambios sugeridos a algunas de las interfaces de los diagnósticos por imágenes. Por favor indique en que lugar de la arquitectura definida anteriormente impactarán estos tres cambios.

Modelo



Arquitectura del Software

Teoría

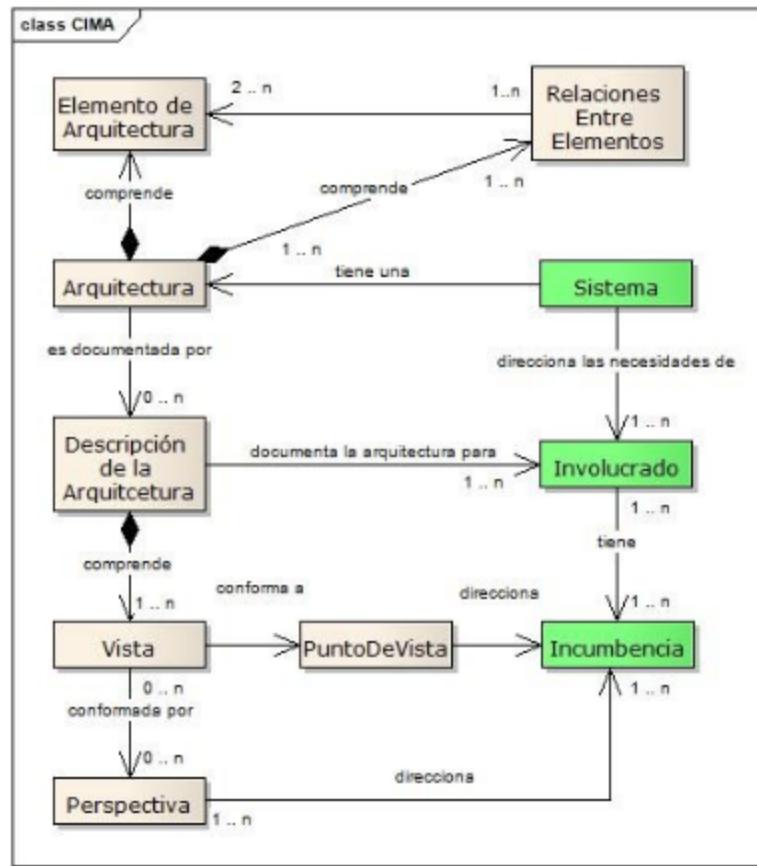
Perspectivas

Son los requerimientos no funcionales

- Seguridad
- Performance y escalabilidad
- Disponibilidad y resiliencia (capacidad de recuperación)
- Evolución
- Accesibilidad
- Internacionalización

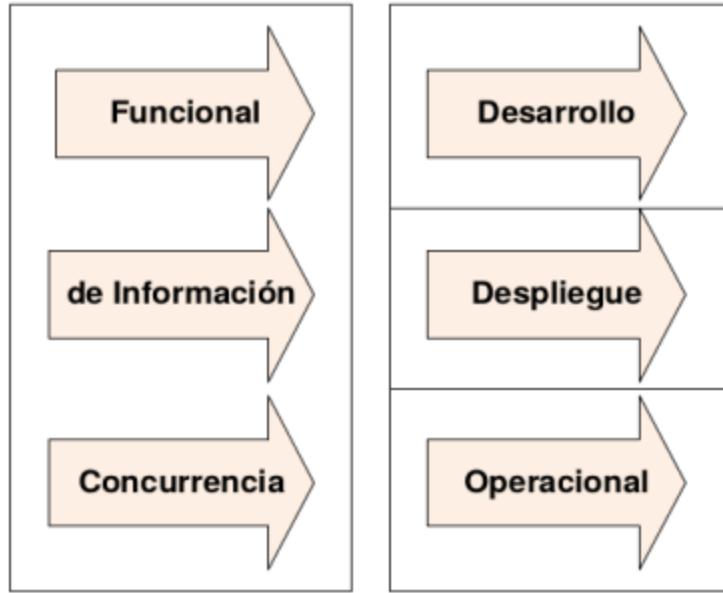
- Locación
- Usabilidad
- Regulación
- Recursos de desarrollo

Elementos



Vistas

Es un conjunto de patrones, plantillas y convenciones para la construcción de un tipo de vista. En él se definen las incumbencias y preocupaciones de los involucrados. En él se reflejan las directrices, principios y modelos para la construcción de sus vistas.



La idea es relacionar las vistas con las perspectivas

Vista **funcional***: Describe los elementos funcionales del sistema, sus responsabilidades, interfaces e interacciones primarias. Conduce la forma de las otras estructuras del sistema tales como la estructura de la información, la estructura de la concurrencia, la estructura del despliegue, y así sucesivamente.

Vista de **información***: Describe la forma en que la arquitectura almacena, manipula, maneja, y distribuye información. El objetivo de este análisis es responder a las grandes preguntas en torno a contenidos, estructura, propiedad, latencia, referencias y la migración de datos.

Vista de **desarrollo***: Describe la arquitectura que soporta el proceso de desarrollo de software.

Esta Vista de Desarrollo comunica los aspectos de la arquitectura de interés para los actores involucrados en la construcción, pruebas, mantenimiento y la mejora del sistema.

Vista de **concurrencia**: Describe la estructura de la concurrencia del sistema y mapea elementos funcionales a las unidades de concurrencia para identificar claramente las partes del sistema que pueden ejecutarse al mismo tiempo y cómo esto se coordina y controla.

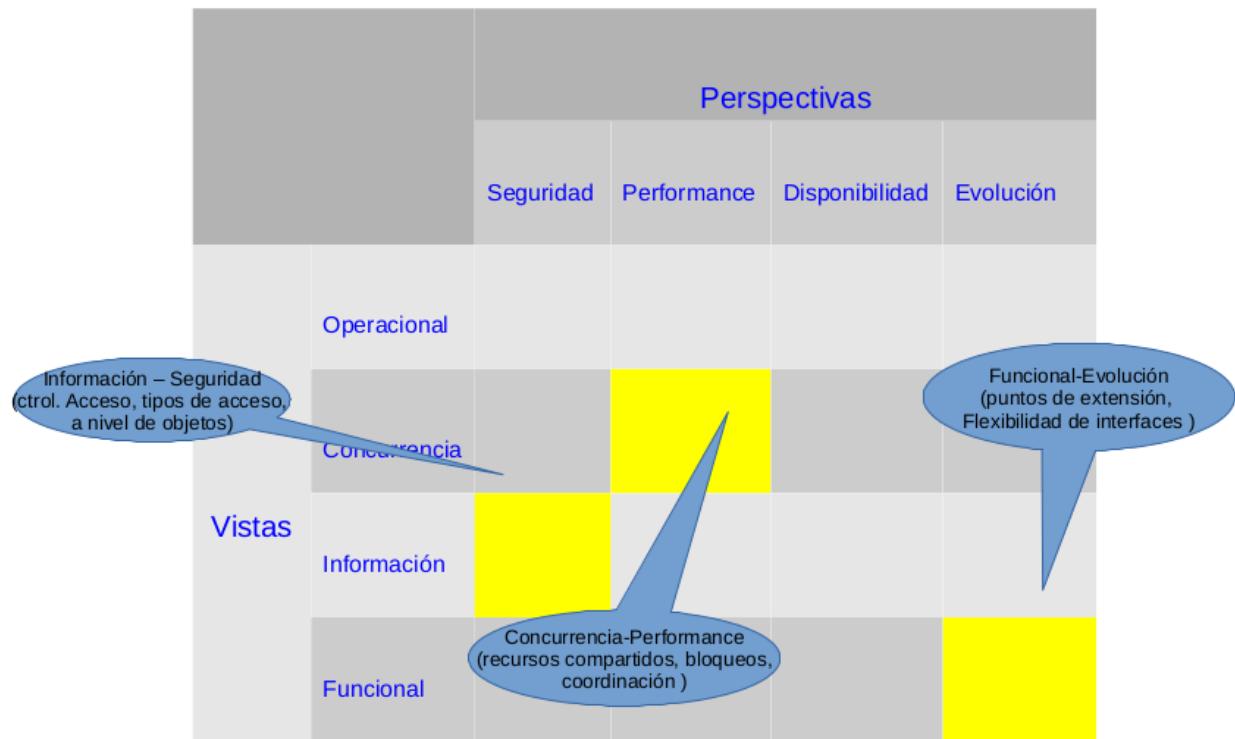
Vista de **despliegue**: Describe el ambiente en el que el sistema se implementará, incluyendo la captura de las dependencias que el sistema tiene en su ambiente de tiempo de ejecución.

Vista **operacional**: Describe cómo el sistema será operado, administrado, y soportado cuando se esté ejecutando en su entorno de producción.

Estas últimas dos son clave en sistemas voluminosos con muchos puntos de falla.

*no pueden faltar.

Ejemplo



La vista de información está relacionada a la seguridad porque tengo que administrar los datos con distintos tipos de seguridad.

La vista de concurrencia impacta en la performance por ejemplo, dependiendo de la estrategia de concurrencia que elija, alguna de ellas va a bloquear el sistema y va a degradar la performance entonces voy a tener que trabajar en balancear eso.

La evolución tiene un impacto sobre la vista funcional, evolucion tiene que ver con la flexibilidad ante los cambios o la posibilidad de hacer reuso y que sean flexibles las distintas partes que se desarrollaron.

Grado de aplicabilidad de las Perspectivas				
Vistas	Seguridad	Performance y Escalabilidad	Disponibilidad y Resiliencia	Evolución
Funcional	media	media	baja	alta
Información	media	media	baja	alta
Concurrencia	baja	alta	media	media
Desarrollo	media	baja	baja	alta
Despliegue	alta	alta	alta	baja
O operacional	media	baja	media	baja

Patrones de arquitectura y diseño

Criterios generales de calidad de diseño

Economía: evitar complejidades innecesarias, flexibilidad innecesaria, enfoque en la usabilidad más que en la reusabilidad

Visibilidad: la especificación en términos de interfaces y herencia o implementación suele ser obscura, poco visible y difícil de entender y navegar. Prefiera composición y no herencia.

Espaciamiento: contribuye a la visibilidad y al reuso.

- Distancia entre responsabilidades funcionales claramente distintas y autónomas conduce a componentes.

- Espaciado entre las perspectivas de uso de un componente conduce a interfaces con rol específico.
- La separación entre grupos de componentes conduce a capas y subsistemas.
- El espacio entre el contrato y la realización conduce a interfaces explícitas e implementaciones separadas.
- Distancia entre puntos comunes y las variabilidades conduce al marco estable de arquitecturas con conceptos enchufables definidos para variabilidades.

Simetría: facilita la comprensión, comunicación, extensión y mantenimiento.

Emergencia: facilita la organización y escalabilidad. Es el comportamiento de auto-organización y junto con el control son a menudo la clave para la escalabilidad, eficiencia y economía en las arquitecturas. Ejemplo: validación de reglas de negocio.

Patrones

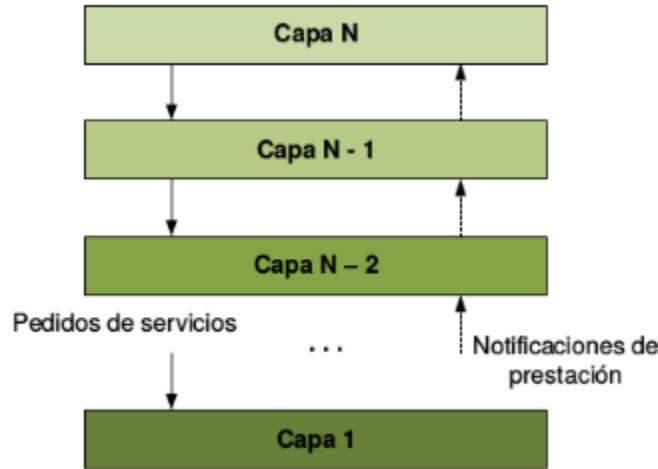
Layer

¿Cuándo usarlo? cuando la arquitectura que se busca definir para el sistema debe soportar diferentes y variados niveles de abstracción. Se requiere cambiabilidad, reuso a partir de la definición de interfaces estables. **Partición lógica.** Cualquier sistema con un negocio que no sea trivial, su composición en capas garantizaría un desarrollo más organizado. El concepto es particionar un negocio que tiene distintos niveles de abstracción. El problema a resolver es que cada una de esas layers hagan posible el reuso a partir de la definición de interfaces estables.

¿Cómo usarlo? Cada abstracción en un layer, cada subproducto deseable de reuso en cada layer. La idea es estratificarlo tal que cada capa superior use una capa inferior.

Ventajas / beneficios: reuso, cambiabilidad, desarrollo simultáneo.

Desventajas: diseño más elaborado, no hay una receta que me diga qué poner en cada capa o cuántas capas poner.



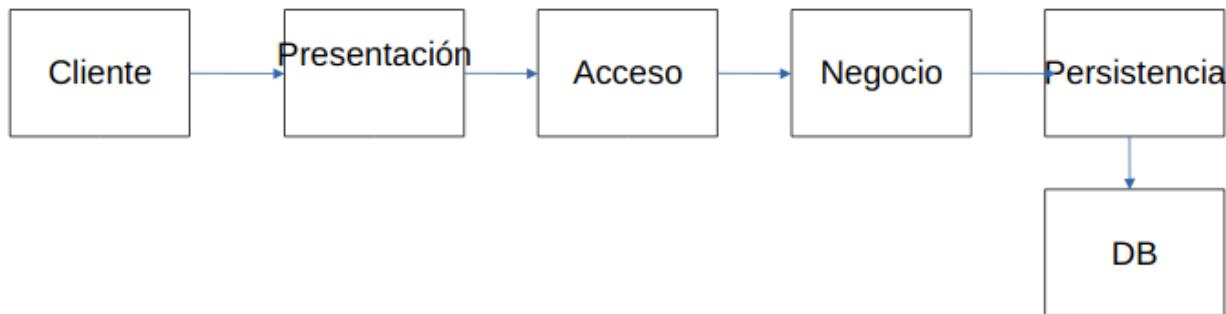
EA

¿Cuándo? Cuando el sistema es cliente servidor y el servidor es remoto. Cuando hay usuarios concurrentes, múltiples interfaces de usuario, reglas de negocio y datos persistentes.

¿Cómo usarlo? Desacoplando las diferentes “tiers” se logra aeparación de incumbencias y reuso.

Ventajas: reuso, cambiabilidad

Desventajas: mayor trabajo



Microkernel

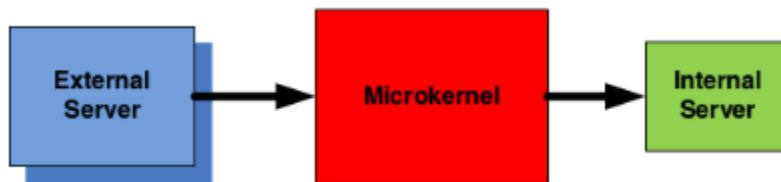
¿Cuándo usarlo? En **sistemas de larga vida** que deben evolucionar ante cambios de distinto tipo y con diferente frecuencia. Ejemplo: un sistema operativo. Un sistema

operativo en general está pensado para durar en el tiempo, sin embargo la tecnología cambia y por ejemplo, podrían aparecer nuevos protocolos de red.

¿Cómo usarlo? Asignar componentes esenciales al microkernel y distribuir los otros componentes entre los servers. En el ejemplo del sistema operativo, incluiríamos la administración de la memoria, procesos y file system en el microkernel, en el internal server se incluiría el manejo de todos los protocolos y los external server incluirían los distintos contextos de ejecución para las aplicaciones de otros sistemas operativos.

Ventajas / beneficios: soporta cambios con menor impacto.

Desventajas: diseño más elaborado.



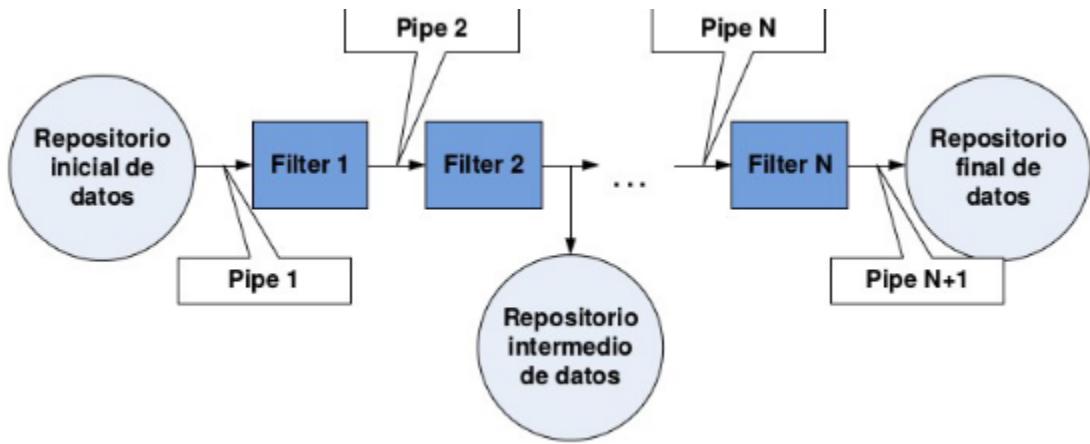
Pipe Filter

¿Cuándo? Cuando es necesario procesar un stream de datos y el procesamiento es factible de descomposición en N procesamientos atómicos. Por ejemplo, en un laboratorio que procesa señales.

¿Cómo usarlo? Distribuir los procesamientos en los filters, definir el formato de datos de los pipes y definir la implementación.

Ventajas / beneficios: reuso, rápido prototipado

Desventajas: no es aconsejable en sistemas críticos por no controlar el estado global, mayor trabajo en el diseño porque se tiene que definir un estandar para los pipes.



MVC

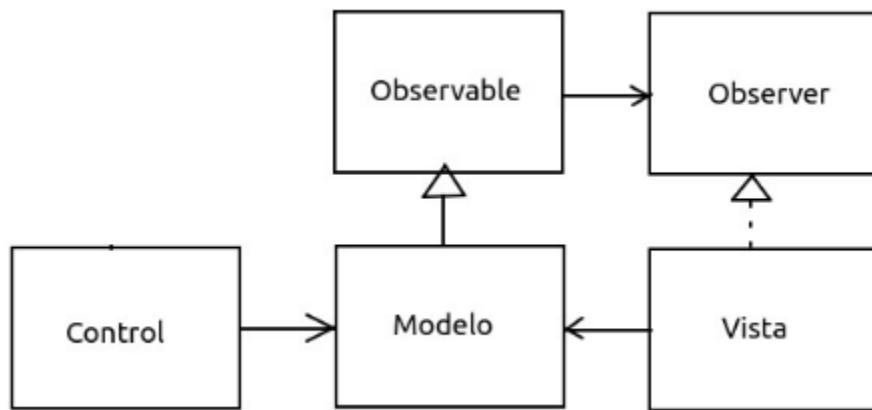
¿Cuando? Cuando es necesario contar con interfaces de usuario múltiples, diferentes para los mismos datos, flexibles y cambiables además de código de negocio transportable.

Que la interfaz conozca al sistema y no el sistema a la interfaz.

¿Cómo usarlo? Las interfaces conocen al modelo, el modelo está desacoplado (transportable). Observer / Observable se ocupan de la propagación de cambios en el modelo.

Ventajas / beneficios: reuso, facilidad de cambios y extensión

Desventajas: cuidado con librerías que ya implementan algo, mayor diseño



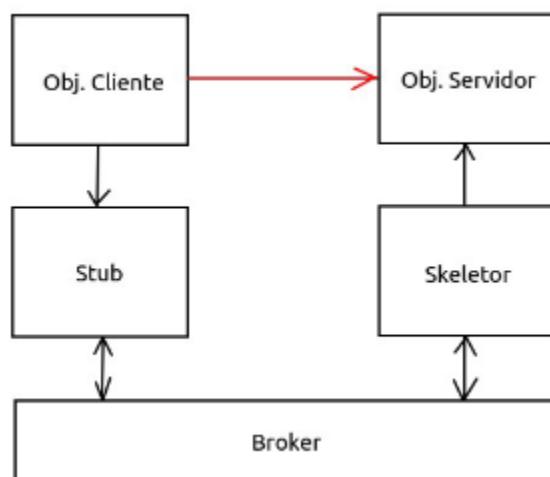
Broker

¿Cuando? Cuando es necesario contar con procesamiento distribuido a partir de objetos distribuidos. Cuando es necesario abstraerse de la tecnología que implementa la comunicación entreprocesos.

¿Cómo usarlo? Siempre que pueda no distribuya sus objetos.

Ventajas / beneficios: cambiabilidad, portabilidad, reuso

Desventajas: baja eficiencia, no tolerante ante fallas



Ejercicio: Sistema Locu

Enunciado

EMPRESA es un organización que se dedica a la venta de equipos para la administración de llamadas telefónicas en el área de telefonía pública. Estos equipos están compuestos de el software para la administración de la totalidad de la funcionalidad requerida por el negocio y además el hardware para interactuar con las líneas de telefonía.

El contexto en el cual deben funcionar estos equipos es el de los conocidos como locutorios. El locutorista compra servicios a la TELCO y los revende, ganando de esa forma una ganacia pactada con la TELCO. El locutorio es una sala con N cabina en cada una de las cuales hay una linea telefonica que puede utilizarse para realizar llamadas. Las N lineas se conectan a la computadora de administración de la sala a través de la interfaz de hardware. Los clientes solicitan realizar llamadas al operador, este indica una cabina por su número y el cliente pasa y realiza una o más llamada. Cuando el cliente finaliza, solicita pagar por el servicio, el operador extrae del sistema

un ticket en el cual aparecen los totales y detalles de los servicios utilizados. A partir de este momento la cabina pasa a estar disponible para ser asignada a otro cliente.

Durante la realización de las llamadas, el cliente dispone información visual en un visor situado en la cabina del ANI discado y el monto gastado hasta ese instante.

Durante la realización de las llamadas, el operador dispone información visual en la terminal de la

computadora del estado de todas y cada una de las cabinas. Debe visualizar el estado, el ANI discado, el monto gastado y el número de llamadas realizadas hasta ese instante, también el tipo, destino y duración de la llamada actua.

El operador previamente a asignar una cabina deberá pasarla del estado bloqueada a habilitada. El operador podrá bloquear cualquiera de las cabinas siempre que no haya una llamada en curso

Los operadores trabajan en turnos de ocho horas. Para administrar el flujo de dinero del turno, se debe llevar un reconto de las llamadas realizadas y de la contabilidad asociada. También se requiere que se haga lo mismo para cada día y mes. Los operadores solo tendrán acceso a su turno y los administradores al resto de la información. Se debe poder ver e imprimir cada registro contable, también deben memorizarse.

A partir de una normativa de la TELCO, el sistema debe comunicarse por lo menos una vez por día con un centro de control y procesamiento a efectos de tomar la información actualizada para tarifar las llamadas. También debe entregar al centro el movimiento del locutorio, todas las llamadas, por esta razón debe memorizarse dicha información. La comunicación con la TELCO debe ser vía modem telefónico y /o algún protocolo sobre TCP/IP (http o FTP).

Los posibles clientes de EMPRESA van desde empresas familiares con un único locutorio de cuatro (4) cabinas hasta Pymes propietarias de varios locutorios de hasta veinte (20) cabinas cada uno.

Los clientes deben poder hacer consultas sobre numeraciones telefónicas para distintos destinos así como precios de llamadas a distintos destinos en diferentes horas del día.

En cada cabina se debe poder cargar un valor límite de dinero, de forma que gastado dicho monto, el sistema termina la llamada en forma automática.

Se debe poder dar de alta clientes, los cuales harán prepagos y al ser asignados ellos a una cabina, el importe gastado se debitara del monto prepagado.

El sistema debe ser lo suficientemente flexible para adaptarse a diferentes ambientes legales, por

ejemplo utilización de impresoras fiscales o impresoras eticketeadoras de alta

velocidad.

El sistema debe ser transportable, pudiendo con el menor costo de adaptación funcionar sobre plataformas tipo Microsoft y / o Linux.

Este sistema tiene dos perspectivas centrales a su funcionamiento. Por un lado la **Performance** debido a su característica de tiempo real, la otra es la **Evolución** debido a la cantidad de mercados diferentes que se desea proveer.

Aplicabilidad a las diferentes Vistas de la perspectiva Performance	
Vista	Aplicabilidad
Funcional	La aplicación de esta perspectiva puede revelar la necesidad de cambios y compromisos a su estructura funcional ideal para lograr requerimientos de rendimiento del sistema (por ejemplo, mediante la consolidación de los elementos del sistema para evitar la comunicación excesiva). Los modelos de este punto de vista también proporcionan información para la creación de modelos de performance. <u>La suma de impresiones fiscales manejadas vía líneas serie hizo necesario desacoplarlas del resto de la aplicación para no degradar la performance.</u>
Información	La vista de información proporciona información útil para los modelos de desempeño, identificando recursos compartidos y los requisitos transaccionales de cada uno. Al aplicar este perspectiva, es posible identificar los aspectos de la vista Información como obstáculos para rendimiento o la escalabilidad. Además, la escalabilidad puede sugerir elementos que podrían ser replicados o distribuidos para facilitar el logro del objetivo. <u>Fue necesario la creación de files con información de los tickets para que sean consumidos por las impresoras fiscales.</u>
Concurrencia	La aplicación de esta perspectiva puede dar lugar a cambios en el diseño de concurrencia debido a la identificación de problemas, tales como la contención excesiva sobre los recursos clave. Alternativamente, teniendo en cuenta el rendimiento y la escalabilidad, pueden surgir elementos de diseño más importantes para cumplir estos requisitos en la vista de Concurrencia . Elementos de la vista de Concurrencia (como mecanismos de comunicación entre procesos) pueden también proporcionar las métricas de calibración para los modelos de rendimiento. <u>Se creó un file índice como punto de contención al acceso a la información de tickets a imprimir.</u>

Desarrollo	Una de las posibles salidas de la aplicación de este punto de vista es un conjunto de directrices relacionadas con el rendimiento y la escalabilidad que se deben seguir durante el desarrollo. Estas directrices probablemente tomarán la forma de hacer y no hacer (por ejemplo, patrones y antipatrones) que se deben seguir cuando se desarrolla el software para evitar problemas de rendimiento y escalabilidad más tarde cuando se despliega. Capturar esta información en la vista de Desarrollo. <u>Fue necesario distribuir parte de la funcionalidad en diferentes módulos.</u>
Despliegue	La vista de despliegue es un insumo crucial para el proceso de examen de desempeño y la escalabilidad. Muchas partes de los modelos de rendimiento del sistema se derivaron de los contenidos de este punto de vista, que también proporciona un número de métricas críticas de calibración. A su vez, la aplicación de esta perspectiva a menudo sugerirá cambios y mejoras en el entorno de despliegue, que le permite soportar necesidades de rendimiento y escalabilidad del sistema. <u>La instalación de los releases con las modificaciones mencionadas en dos nodos a efectos de contar con recursos de hardware para diferentes exigencias.</u>
Operacional	La aplicación de esta perspectiva destaca la necesidad de monitoreo del rendimiento y gestión de capacidades. <u>Desde la perspectiva del servicio técnico se agregaron funcionalidades de prueba en modo simulación de llamadas utilizando el modo de funcionamiento real del hardware y las funciones de interrupción.</u>

Aplicabilidad a las diferentes Vistas de la perspectiva Evolución	
Vista	Aplicabilidad
Funcional	Si la evolución requerida es significativa, la estructura funcional tendrá que volver a ser diseñada. <u>La arquitectura fue refactorizada a efectos de mantener clausurada ante cambios la administración de líneas, cabinas, impresiones y almacenamiento de la información.</u>
Información	Si se necesita evolución del ambiente operacional o de la información, un modelo flexible de información será necesario. <u>Cambió con cada versión la estructura de la información que el sistema recibía del CCP, el modo de facturación y la información registrada para cada llamada.</u>
Concurrencia	Necesidades evolutivas pueden dictar en particular elementos de empaquetado o alguna restricción sobre la estructura de concurrencia. <u>No fue alterada por las diferentes versiones una vez refactorizada la arquitectura.</u>
Desarrollo	Requisitos de la evolución puede tener un impacto significativo en el entorno de desarrollo que debe ser definido (por ejemplo, hacer cumplir las directrices de portabilidad). <u>Fue necesario la creación y administración de diferentes</u>

	<u>repositorios de datos.</u>
Despliegue	Esta perspectiva rara vez tiene un impacto significativo en la vista de despliegue, la evolución del sistema por lo general afecta a las estructuras descritas en otras vistas. <u>El despliegue se vio alterado por la mayor cantidad de componentes a instalar.</u>
Operacional	Esta perspectiva por lo general tiene un menor impacto en la vista operacional. <u>No fue necesario generar cambios.</u>

Ejercicio: Punto de venta

Ejercicio: Documentos

Resolución de Parcial

Cuestiones que vía web no se pueden hacer, o sea que se requiere tener un software residente en computadoras locales → broker que ofrezca objetos distribuidos para que el sistema de un externo se conecte y haga consultas.

Tema 1 - Running

El sistema a desarrollar está orientado a la administración de grupos de entrenamiento aeróbico de corredores denominados en adelante “running”.

Este negocio está compuesto por grupos con uno o más profesores entrenadores y eventualmente algún profesional de apoyo como masajistas, médicos deportólogos, nutricionistas, etc. Funcionan de la siguiente manera: los clientes en adelante corredores se agrupan dos o tres veces por semana en un lugar prefijado al aire libre o gimnasio y de allí arrancan con el entrenamiento que consiste en trotos de diferentes distancias, carreras, ejercicios físicos en grupo o individuales.

Además de estos encuentros, el resto de los días de la semana los corredores entran en forma individual.

Los planes de entrenamiento son elaborados por los profesores de acuerdo a las características del corredor y son distribuidos a los mismos a través de la web. Los

planes son semanales, quicenales y/o mensuales e incluyen las actividades día por día. Los planes son seleccionados de un grupo prearmado y se adaptan a la medida de cada corredor.

Por el servicio de planificación y seguimiento del entrenamiento más cualquiera de los complementarios los corredores pagan una cuota mensual.

En ocasiones los corredores asisten a competiciones (carreras, maratones, etc.) de las cuales son

avisados a través de la web.

Cualquier otro evento que los corredores deban conocer tal como próximo vencimiento de cuotas,

cancelación de entrenamientos, encuentros sociales, etc. serán informados a través de un servicio de mensajería al estilo Whatsapp.

El sistema a desarrollar estará compuesto por un sitio web y una aplicación móvil con la cual los

corredores podrán interactuar con dicho sitio y consultar desde sus teléfonos celulares su plan de

entrenamiento de la fecha e ingresar datos relacionados tales como tiempos y observaciones.

La aplicación será descargada de la web previa identificación del corredor. También éstos podrán

ingresar al sitio previa identificación y consultar la misma información que con la aplicación más

datos extendidos y análisis de sus entrenamientos así como información general del grupo.

Con la aplicación los profesores podrán acceder a la misma información que los corredores sólo que para la totalidad del grupo.

El sitio será utilizado por los profesores para la gestión de corredores en sus aspectos administrativos (alta de nuevos corredores, registración de pagos) y técnicos (armado y análisis de

entrenamientos). El sitio funcionará conectado a una página web en Facebook para la publicación

de información pública del grupo.

Para aquellos corredores que posean algún reloj/dispositivo cronómetro/podómetro/gps el sistema

tomará información del sitio asociado al dispositivo y los mostrará asociado al plan de

entrenamiento del corredor. Para esto ya se cuenta con información del API de dos de dichos sistemas.

En principio los profesionales complementarios podrán acceder al listado de la totalidad los corredores y documentar con un texto corto y libre la práctica complementaria que realicen.

Este sistema es nuevo y se cree que con la experiencia de uso requerirá la incorporación de nuevas funcionalidades. Indique como prepararía su diseño para soportar este requerimiento. También deberemos hacer adaptaciones en la presentación de la información a todos los roles ya

que su utilización sugerirá nuevas formas de uso.

Además es posible que deba cambiarse parte de la información de los alumnos que actualmente se administra para agregar alguna adicional. Es posible también que algún nuevo reporte/vista deba ser elaborado con la información ya disponible.

Se está pensando la incorporación, para una segunda versión, en la interacción con un sistema

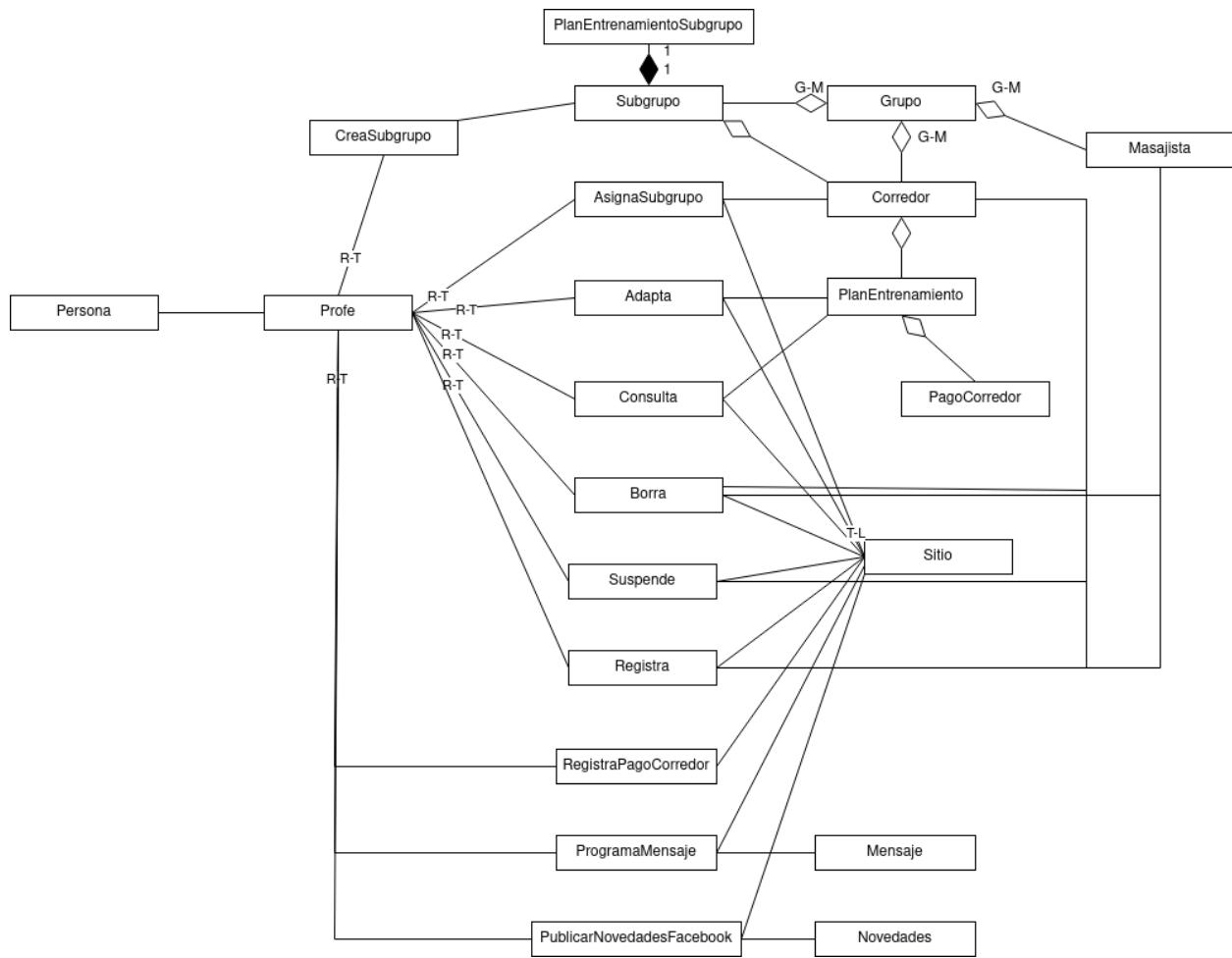
externo de GPS para ofrecer la programación de circuitos con diferentes recorridos. Qué lineamientos y elementos incorporaría en su diseño para facilitar la evolución del sistema a lo largo del tiempo.

Por último haga una evaluación corta acerca de qué tecnología (objetos, funcional) utilizaría en la implementación de cada parte del sistema.

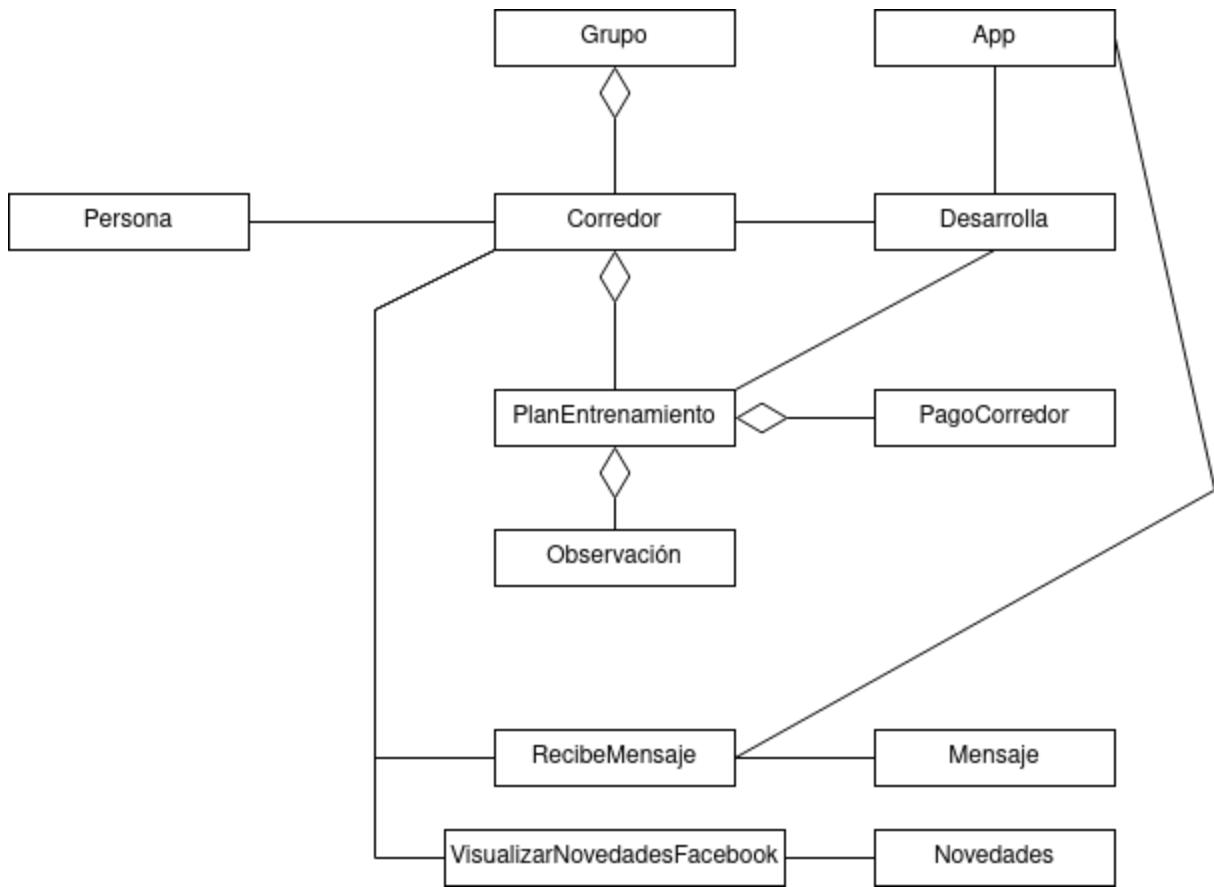
Notar que el sitio está pensado para que sea usado por los entrenadores y la aplicación para ser usada por los corredores.

Es importante que si hay una transacción tiene que haber un ítem asociado a la transacción.

Rol Entrenador/Profesor



Rol corredor



Arquitectura

Vistas	Usabilidad	Evolución	Cambiabilidad/Escalabilidad
Funcional	x	x	x
Información		x	x
Desarrollo	x	x	x
Concurrencia	x		x
Despliegue			x
Operacional	x		x

Usabilidad porque hay roles diferentes y no sabemos qué perfil van a tener los corredores, si algunos nunca usaron una computadora, otros sí..

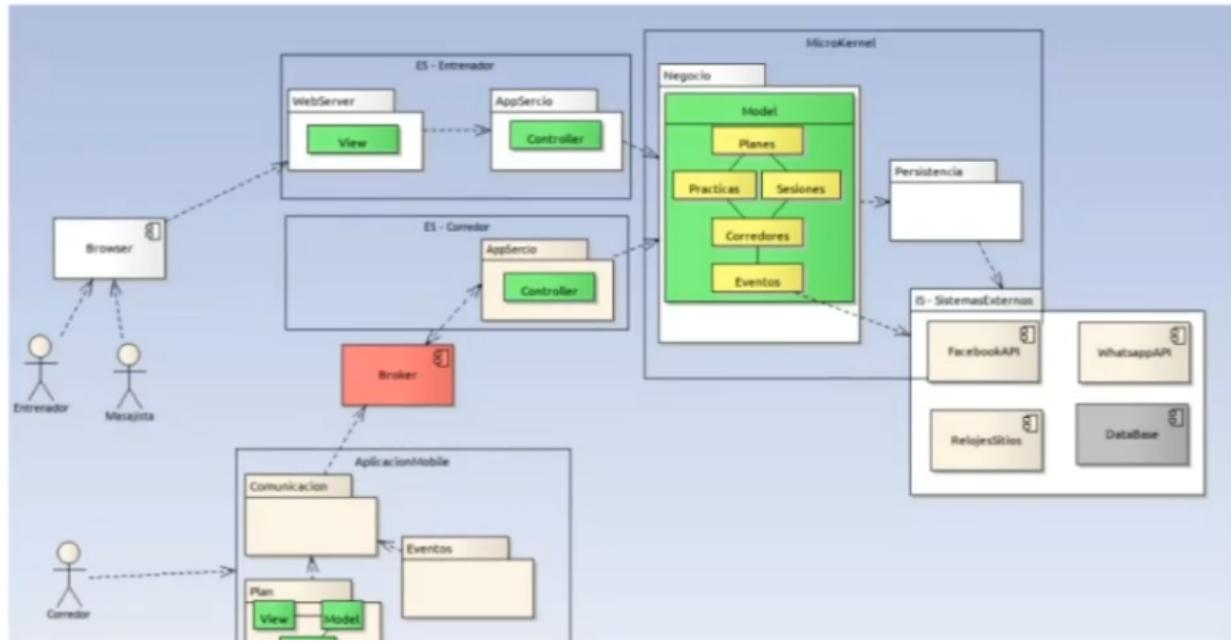
Evolucion porque según dice el texto con el uso van a aprender para qué sirve el sistema y van a aprender nuevas cuestiones ⇒ el sistema debe evolucionar.

Escalabilidad porque dice que eventualmente va a conectarse a distintos tipos de dispositivos.

Importante → los requerimientos no funcionales deben derivarse del problema, no hay que listar cosas que se nos ocurran. No poner disponibilidad porque excede a esta materia.

Arquitectura

Como es una aplicación que tiene multiples usuarios, podria llegar a haber algun tipo de concurrencia, algunas pocas reglas de negocio, conexion con sistemas externos ⇒ entonces se propone una arquitectura de tipo EA que, además, esté de alguna manera complementada con algo que me permita soportar el impacto que van a tener los cambios que se hagan eventualmente. Por ejemplo, cambios en alguna mensajería externa tipo WhatsApp, en distintos relojes que se soportarán, etc. Además, dice que los usuarios van a poder sugerir cambios entonces nos interesaría asentuar el uso de la tecnologua que se use con MVC. Como voy a tener distintos roles y las funcionalidad son diferentes ⇒ Microkernel. El microkernel va a tener toda la logica de negocio estratificada en layers y para el lado de los external servers estara el uso del MVC, como la aplicación web que va a ser utilizada por los profesionales complementarios y entrenadores, y por el lado de internal servers las conexiones con los relojes, gps, base de datos, etc. y por otro lado como hay un aplicación que tiene que hacer uso del sitio, entonces la app también tiene la problematica de flexibilizacion de cambios y va a estar conectada a traves de objetos distribuidos ⇒ broker.



Usabilidad y evolución → MVC/Internal servers

Escalabilidad → Microkernel

Tema 2

El depósito de la empresa XXX recibe productos(items) de sus proveedores y los almacena hasta que reciben una orden de compra. Hay una dársena de descarga donde arriban los items que luego serán almacenados en pallets en distintos tachos. Los operadores se encargan de distribuir los items en los tachos buscando no mezclarlos. Cuando llega una orden de compra deben armar y despachar los pedidos realizando todo el trabajo a mano.

No poseen un sistema para llevar control del stock sino que lo cuentan a mano. Cuando se arma un pedido no tienen certeza de encontrarlo, pero tampoco desean acumular stock.

Se desea entonces: Un sistema que supervise la recepción de productos facilitando a los operadores la descarga y depósito de los items recibidos en tachos con espacio libre a partir de la impresión de una guía de descarga con las coordenadas del lugar en donde realizar la descarga de acuerdo al control de las cantidades del depósito.

Para despachar a sus clientes los items una vez empaquetados son acompañados del remito correspondiente generado en el depósito con los datos de los clientes, que se

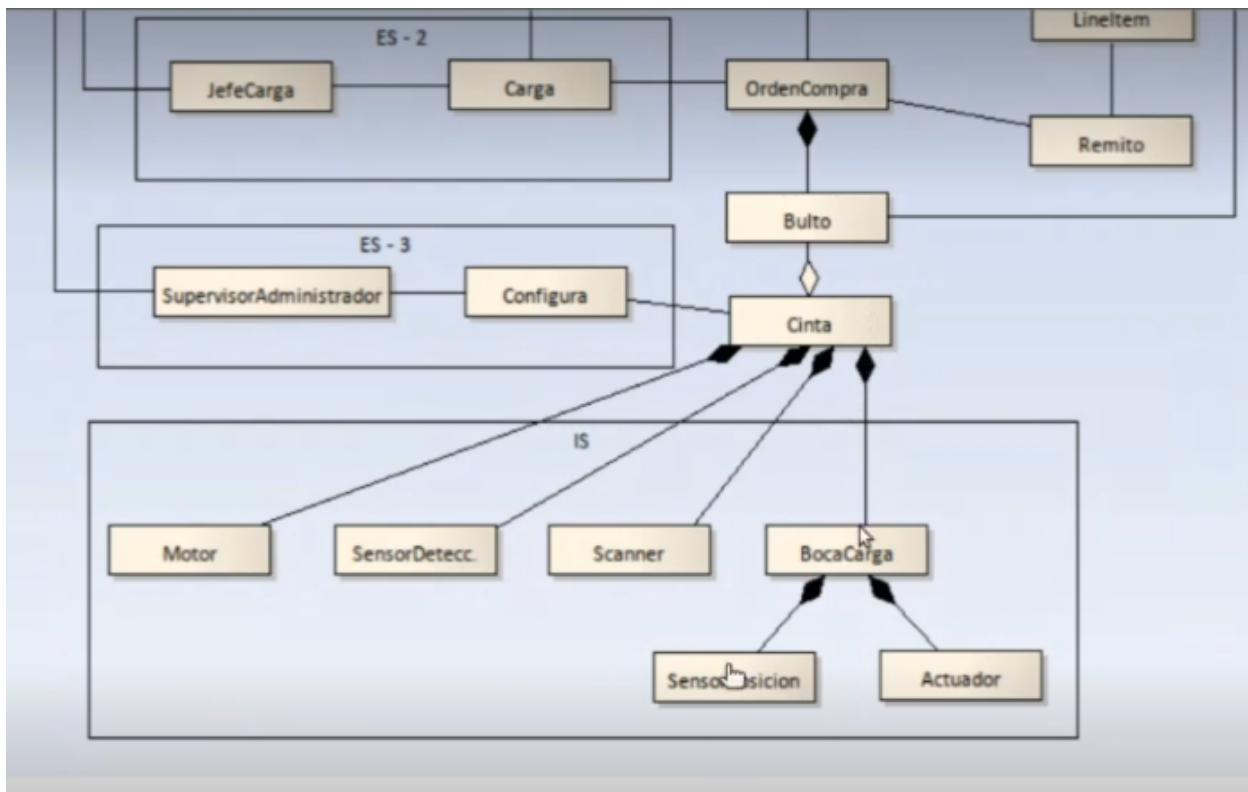
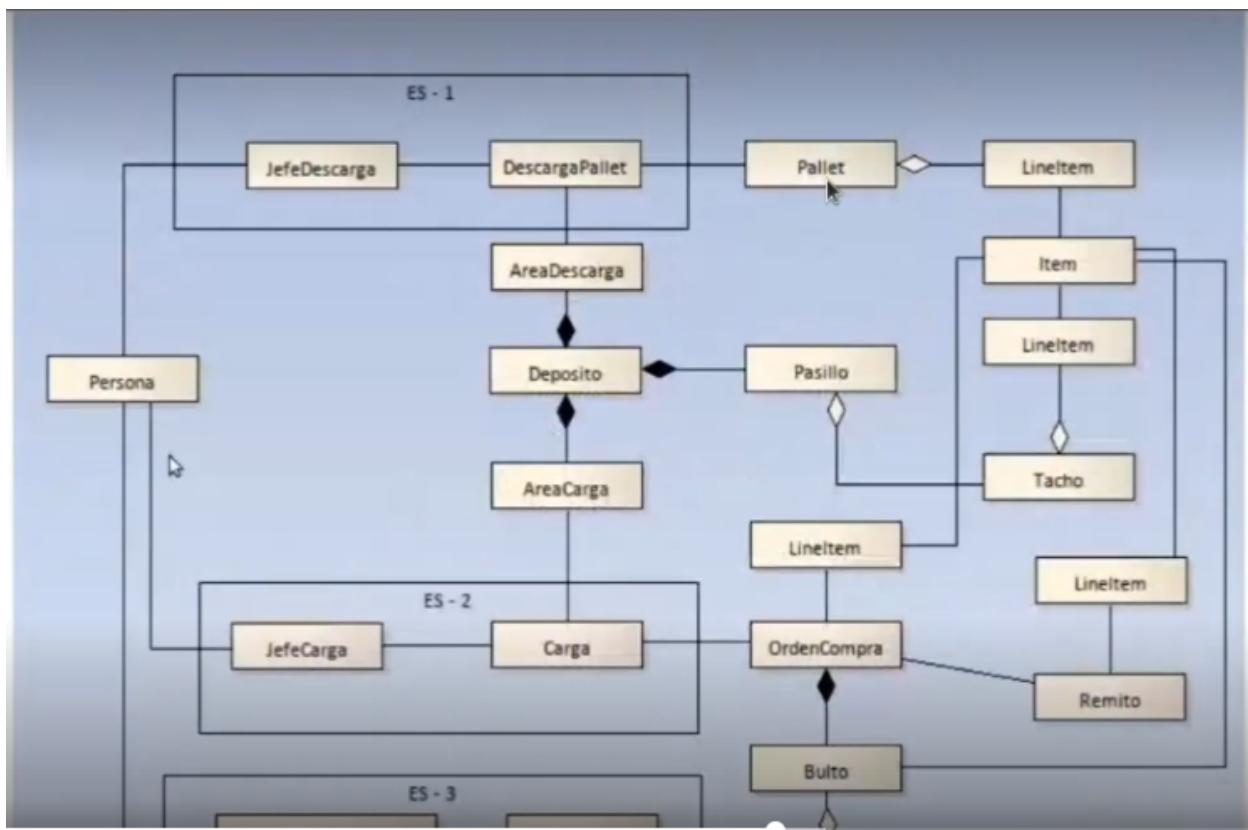
registraron en el sistema de ventas de la empresa.

Para esto se decidió incorporar una cinta transportadora que lleve los pedidos ya armados manualmente a las distintas bocas de expedio donde los camiones los cargan para distribuirlos desde la darsena de carga.

La idea es que al imprimir las guías de armados, las que vienen de las órdenes de compra, se asigne una boca y se le pegue un ticket autoadhesivo al pallet de forma que cuando el operador encargado de armar la orden de compra, empaqueta los items y los deposita en la cinta, este los transporta a la boca predefinida. Los pedidos son distribuidos a través de la cinta. Para que un pedido llegue a una boca, se cuenta con un sensor y un mecanismo de empuje, que al leer el código previamente mencionado empuja en pedido en la boca.

En el futuro pueden haber cambios y agregar un sensor por mecanismo de empuje. El sistema tiene que estar abierto a agregar esta nueva funcionalidad.

El sistema que se debe desarrollar debe contemplar la funcionalidad de la configuración de esta parte del sistema por parte del personal técnico; además debe incluir la funcionalidad de gestionar la descarga de los embarques recibidos por parte de un jefe de recepción y la funcionalidad de gestión de los embarques a despachar por parte de un jefe de despachos.



Vistas	Usabilidad	Evolución	Cambiabilidad/Escalabilidad	Performance
Funcional	X	X	X	X
Información		X	X	X
Desarrollo	X	X	X	X
Concurrencia	X		X	
Despliegue				
Operacional	X		X	X

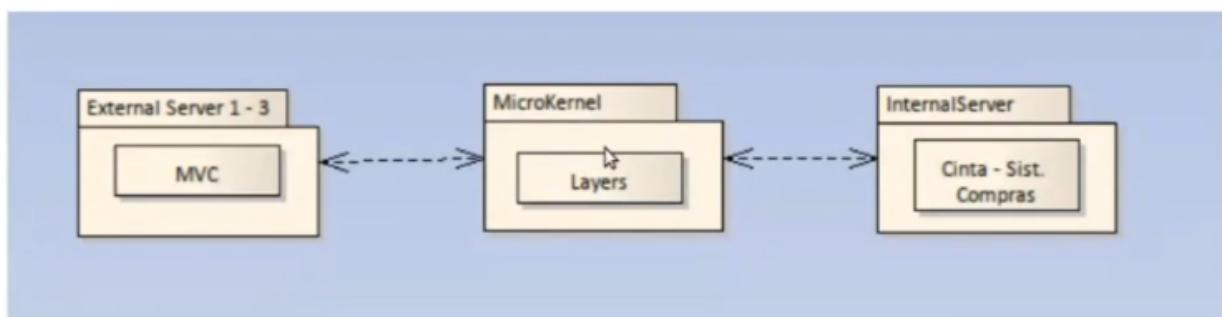
Usabilidad y evolución casi siempre porque en este caso hay operadores que no sabemos su formacion y ademas es un sistema nuevo donde seguro vamos a tener que cambiar algo.

El manejo del sistema de transportacion parece de mediana a alta complejidad y en el futuro va a haber que cambiarlo ⇒ escalabilidad. [pinta microkernel]

Además hay componentes electronicos que van a tener que cumplir con una cuestion de performance porque va a necesitar de tiempo real.

Arquitectura

Microkernel con capas. En el modelo de dominio se pueden ver las capas. Cada MVC va asociado a cada uno de los roles que figuran en el modelo.



Paradigmas de Programación

Teoría

Cada paradigma ofrece un enfoque y un conjunto de mecanismos de razonamiento.

OOP (programación orientada a objetos) = gran cantidad de abstracciones organizadas en jerarquías.

LP (programación lógica) = navegar estructuras simbólicas complejas según un conjunto de reglas lógicas.

FP (programación funcional) = algoritmos complejos resueltos a partir de composición de funciones.

Al conjunto de técnicas de programación y principios de diseño para construir programas es lo que llamamos un modelo de programación; cada lenguaje tiene un Modelo de Programación asociado. El modo de razonamiento lo aporta el paradigma soportado y la sintaxis y semántica lo aporta el lenguaje utilizado.

Por ejemplo, un sistema de software requiere de una plataforma que provea los servicios generales de la arquitectura (protocolos de comunicación, persistencia, armado de las interfaces de presentación, creación y destrucción de objetos, establecimiento de relaciones y ruptura de las mismas entre objetos) y además los escenarios y mecanismos de programación para el procesamiento de datos y validación de reglas de negocio. Por lo tanto el modelo de programación para su implementación requiere que por un lado posea propiedades de abstracción para armar jerarquías, modularidad para ser componible, administración de concurrencia para permitir multiprocesamiento y además la capacidad de resolver algoritmos complejos y de realizar busquedas y validaciones de reglas.

Propiedad	Paradigma		
	OOP (objetos)	FP (funcional)	LP (lógica)
Abstracción	X	X	
Componibilidad	X	X	
Encapsulamiento	X		
Estados	X		
Concurrencia		X	
Declarativo		X	X
Impl. Algoritmos		X	
Validar Reglas			X
Lenguaje	Clojure Scala C++11 Java8 Ruby Common Lisp		
		Mercury	
		Curry	
		Mozart	
		Racket	

Declarative programming is when you say *what* you want, and imperative language is when you say *how* to get what you want. A simple example in Python:

```
# Declarative
small_nums = [x for x in range(20) if x < 5]

# Imperative
small_nums = []
for i in range(20):
    if i < 5:
        small_nums.append(i)
```

Programación declarativa => definiciones => representación del conocimiento externo al sistema +

argumentos => sin estados explícitos

Programación iterativa/imperativa => comandos => representación interna del sistema

=> con

estados explícitos

Concurrencia y estados

Estado = secuencia de valores cronológicos que representan resultados intermedios.

Estado implícito: No necesita soporte de cálculo programado y existe en la mente del programador.

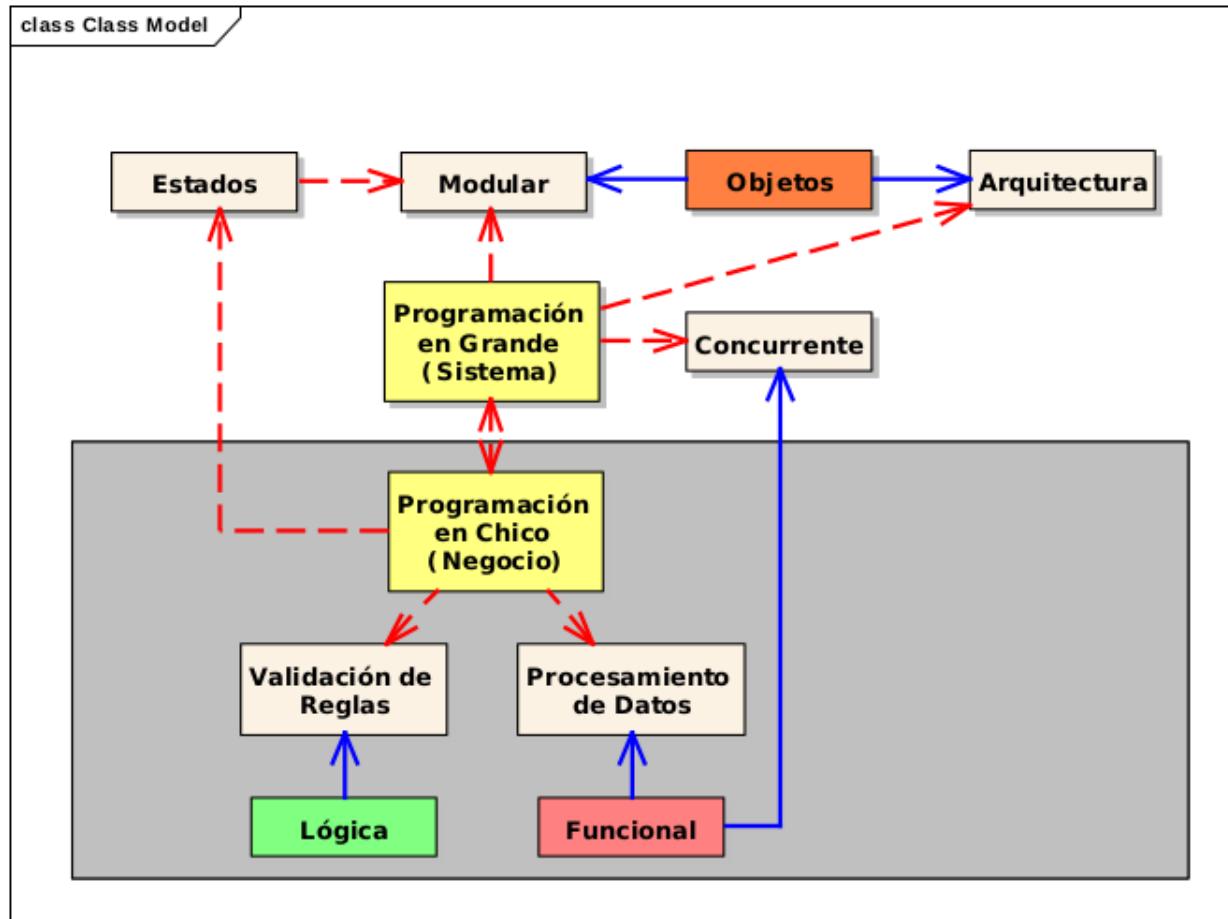
```
fun {Sum Xs A}
  case Xs
  of X|Xr then {Sum Xr A+X}
  nil then A
  end
end
```

Los dos argumentos Xs y A representan los estados implícitos

Estado explícito: necesitan soporte de cálculo programado y existe explícitamente en el modelo

programado por el programador.

La modularidad permite el manejo de estados explícitos. La programación orientada a objetos tiene esta propiedad.



Patrones de Diseño

Teoría

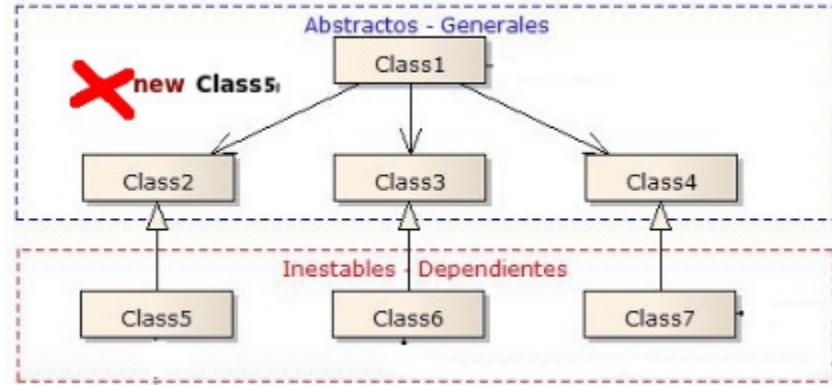
Creacionales (factory, factory method, builder)

¿Cuándo usarlo?

Contexto y problemas presentes en el diseño/código: la arquitectura goza de la inversión de la

cadena de dependencia; en qué lugar del código se deben instanciar los objetos para no

violar el criterio anterior. Clases con múltiples constructores que no expresan su intención, instancias que deben ser construidas con valores predeterminados en sus atributos o relaciones.



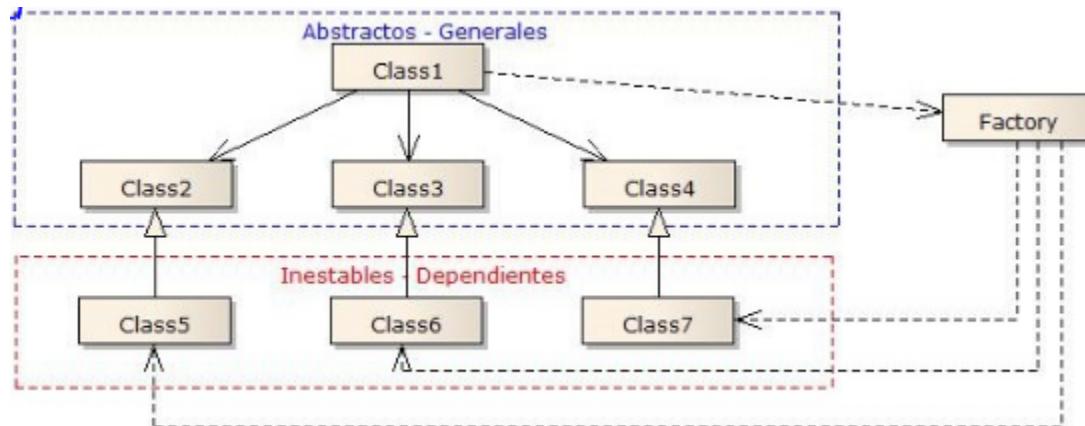
¿Cómo usarlo?

No se puede instanciar objetos utilizando los constructores de objetos en zonas de código clausurado ante cambios porque introducen dependencias. Encapsular el conocimiento y la creación de objetos concretos en métodos o clases que devuelvan objetos concretos detrás de referencias a tipos genéricos.

Consecuencias

Ventajas / beneficios: creación controlada de objetos, código desacoplado al evitar dependencias indeseadas, código más seguro.

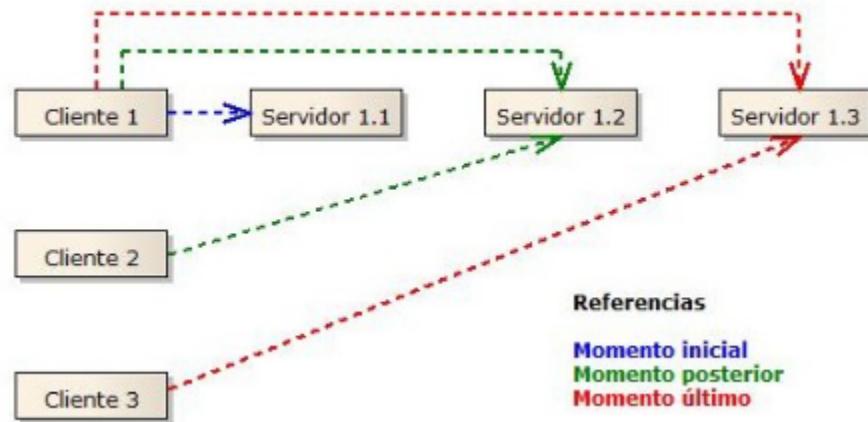
Desventajas: diseño más elaborado.



Extensión de interfaz/objeto

¿Cuando usarlo?

Contexto y problemas presentes en el diseño/código: software con componentes cliente y servidores. Diferentes releases complican el mantenimiento de versiones del servidor que debe evolucionar.



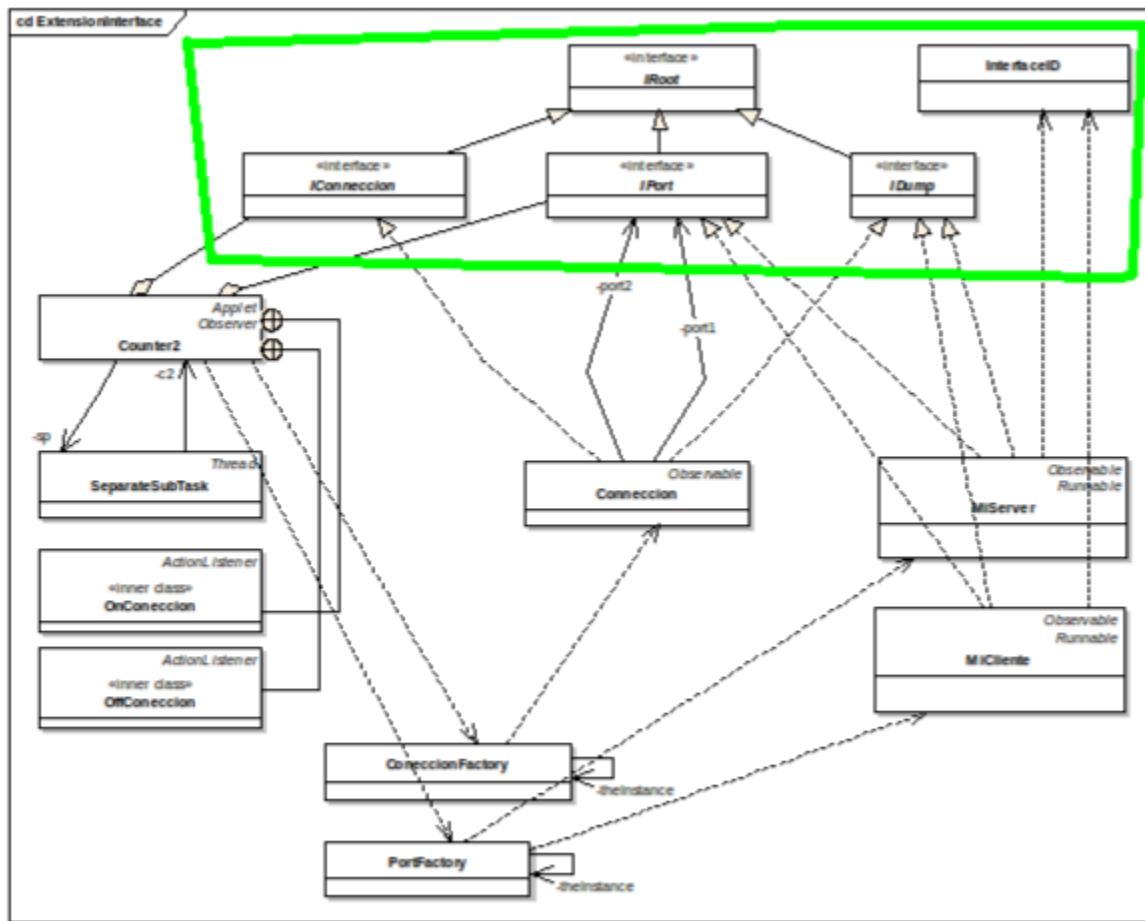
¿Cómo usarlo?

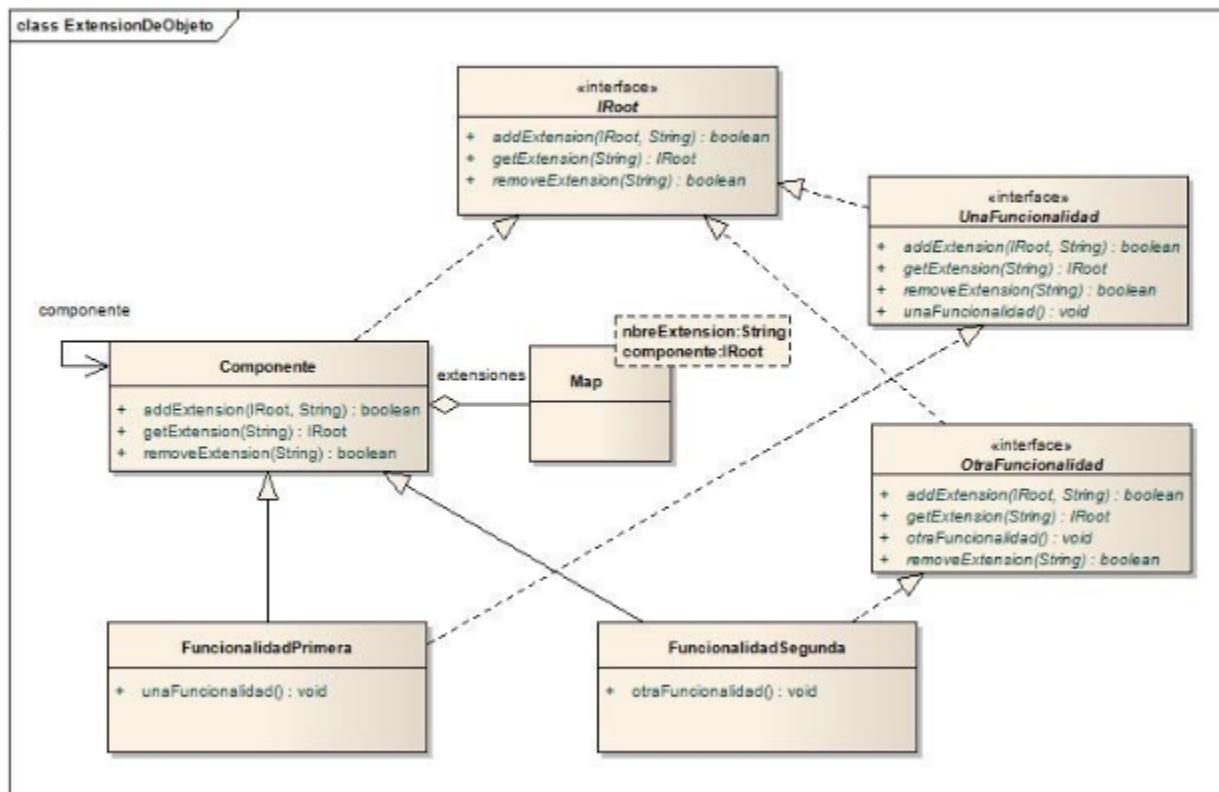
El componente servidor será único para evitar contar con múltiples versiones. Las diferentes versiones del cliente deben ser capaces de funcionar con la única versión adaptada del servidor.

Consecuencias

Ventajas/beneficios: facilita el mantenimiento de componentes al simplificar la evolución del servidor.

Desventajas: diseño más elaborado.

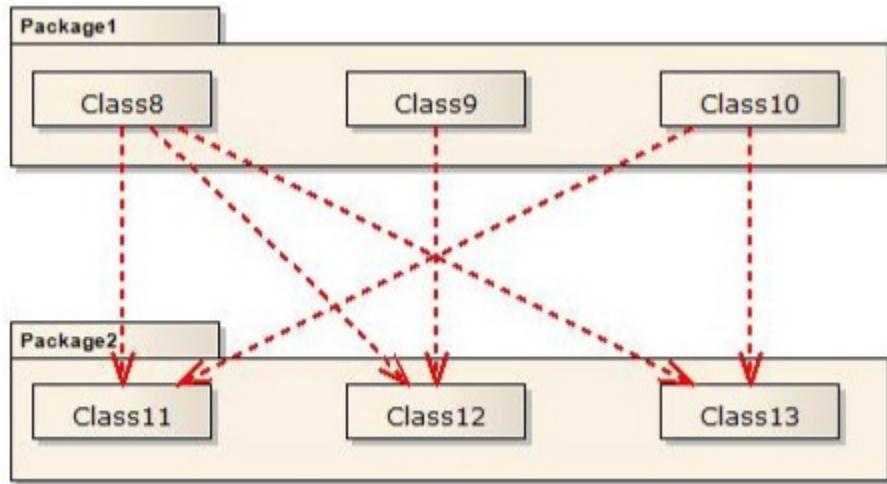




Facade

¿Cuándo usarlo?

Contexto y problemas presentes en el diseño/código: subsistemas que ofrecen múltiples funcionalidades, más de las que se necesitan en el código cliente. Interfaces con sistemas externos con contextos diferentes al del sistema en desarrollo.



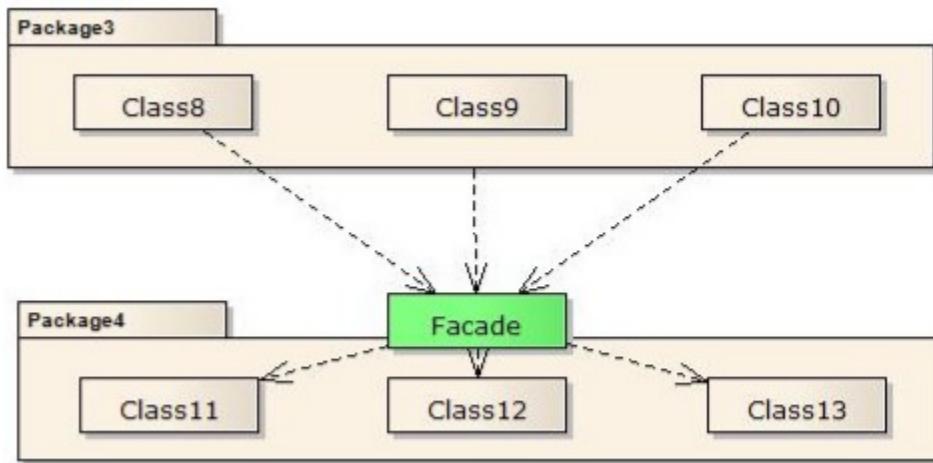
¿Cómo usarlo?

Conjunto de clases que ordenan el acceso según las funcionalidades necesarias en el contexto del código cliente.

Consecuencias

Ventajas/beneficios: facilita el desarrollo y la comprensión del código cliente, evita acoplamientos innecesarios, ordena el acceso a los subsistemas.

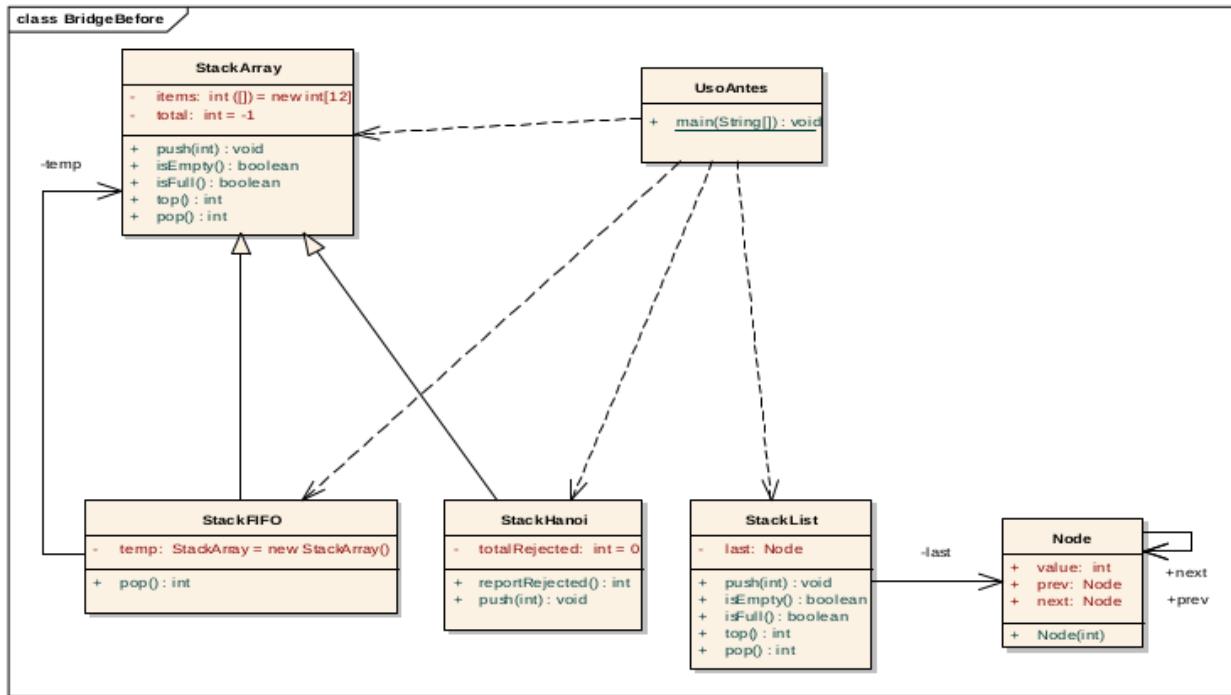
Desventajas: diseño más elaborado.



Bridge

¿Cuándo usarlo?

Contexto y problemas presentes en el diseño/código: abstracciones mezcladas con implementaciones, clases que modelan conceptos ortogonales en la misma jerarquía.



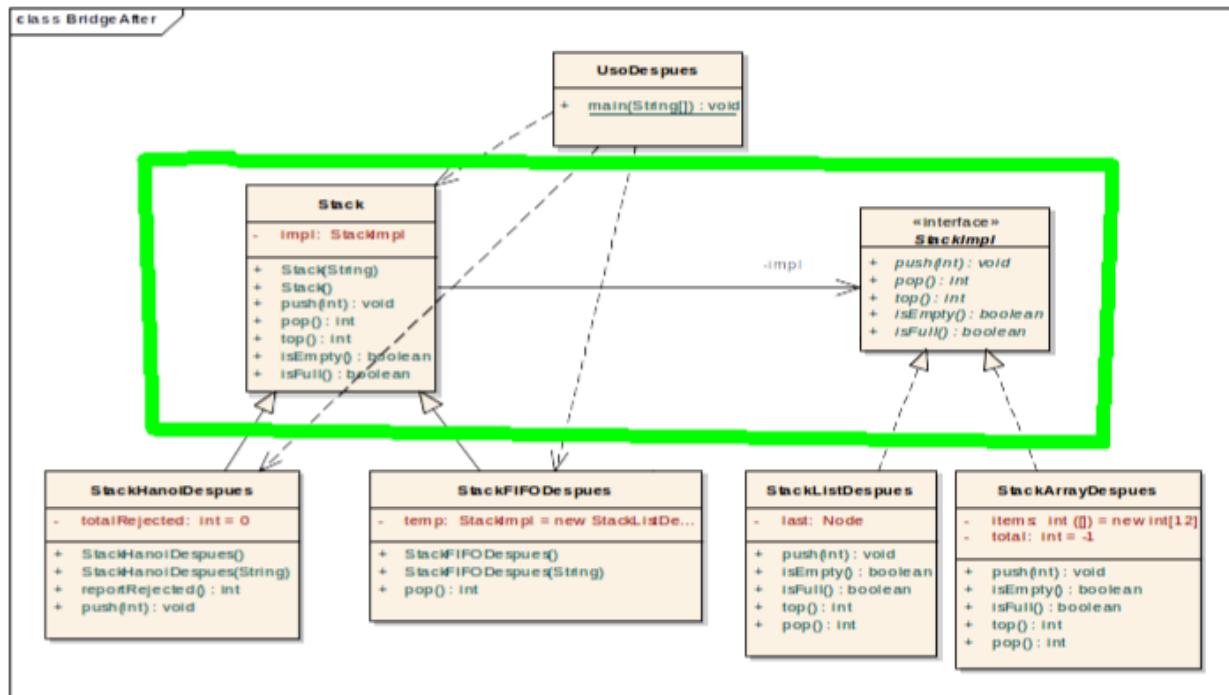
¿Cómo usarlo?

Jerarquías de clases independientes para las abstracciones e implementaciones permiten la evolución por separado.

Consecuencias

Ventajas/beneficios: facilita el desarrollo evolutivo de las abstracciones e implementaciones.

Desventajas: diseño más elaborado



Estado

¿Cuándo usarlo?

Cuando hay código con lógica compleja para administrar la transición de estados de un objeto.

```

class LineaTelefonica {
    private State currentState;
    public static enum State {COLGADO, DESCOLGADO, DISCANDO, HABLANDO};
    public LineaTelefonica() {
        currentState = State.COLGADO;
    }
    public State getCurrentState() {
        return currentState;
    }
    public final void runAll(Iterator<?> inputs) {
        while(inputs.hasNext()) {
            Input i = (Input)inputs.next();
            System.out.println(i);
            Evento ma = (Evento)i;
            if(currentState == LineaTelefonica.State.COLGADO){
                if(ma.equals(Evento.descuelga)){
                    currentState = LineaTelefonica.State.DESCOLGADO;
                    runDescolgado();
                }
                continue;
            }
            if(currentState == LineaTelefonica.State.DESCOLGADO){
                if(ma.equals(Evento.discua)){
                    currentState = LineaTelefonica.State.DISCANDO;
                    runDiscando();
                }
                else if(ma.equals(Evento.cuelga)){
                    currentState = LineaTelefonica.State.COLGADO;
                    runColgado();
                }
                continue;
            }
            if(currentState == LineaTelefonica.State.DISCANDO){
                if(ma.equals(Evento.atienden)){
                    currentState = LineaTelefonica.State.HABLANDO;
                    runHablando();
                }
                else if(ma.equals(Evento.cuelga)){
                    currentState = LineaTelefonica.State.COLGADO;
                    runColgado();
                }
                continue;
            }
            if(currentState == LineaTelefonica.State.HABLANDO){
                if(ma.equals(Evento.cuelga)){
                    currentState = LineaTelefonica.State.COLGADO;
                    runColgado();
                }
            }
        }
    }
}

```

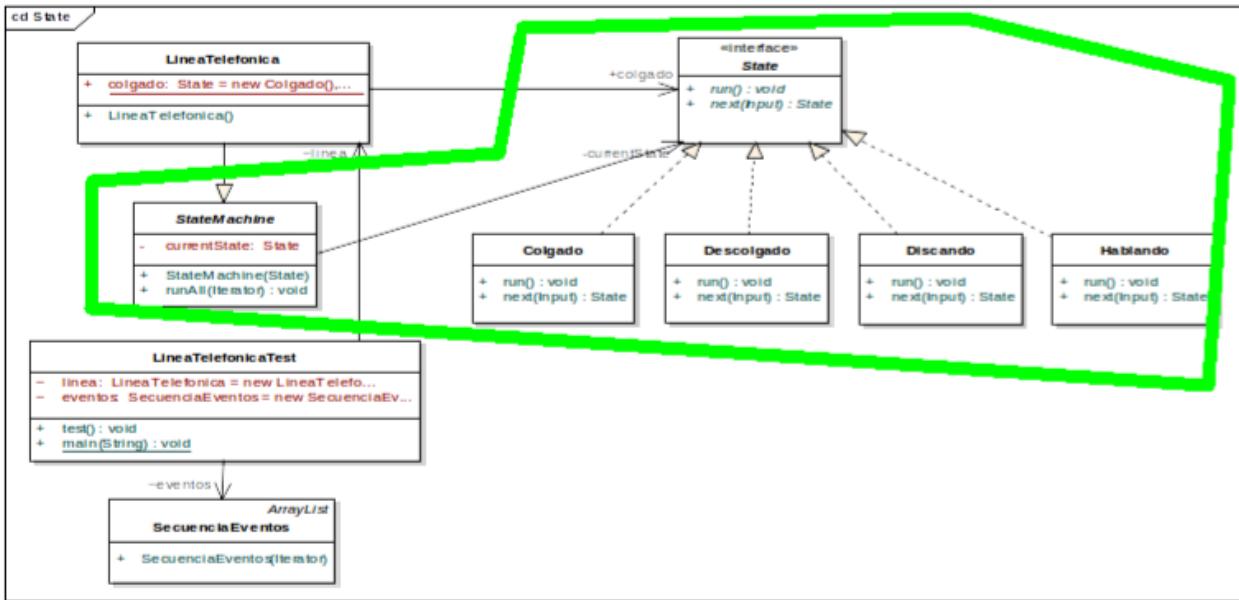
¿Cómo usarlo?

Distribuir la funcionalidad entre los objetos de tipo Estado y la lógica de la transición en clase que funcione como máquina de estados.

Consecuencias

Ventajas/beneficios: aclara la lógica de las transiciones, permite un crecimiento más ordenado.

Desventajas: complica el diseño.



Estrategia

¿Cuándo usarlo?

Cuando hay código con lógica para administrar distintas alternativas de realización de la misma funcionalidad y se desea seleccionarlas en tiempo de ejecución. Encapsula un algoritmo.

```

public void paint(Graphics g)
{
    int xp = calcx(x[0]);      //get first point
    int yp = calcy(y[0]);
    g.setColor(Color.white);   //flood background
    g.fillRect(0,0,getWidth(), getHeight());
    g.setColor(Color.black);

    //draw bounding rectangle
    g.drawRect(xpmin, ypmin, xpmax, ypm);
    g.setColor(color);

    //draw line graph
    for(int i=1; i< x.length; i++)
    {
        int xpl = calcx(x[i]);
        int ypl = calcy(y[i]);
        g.drawLine(xp, yp, xpl, ypl);
        xp = xpl;
        yp = ypl;
    }
}

public void paint(Graphics g)
{
    int xp, yp;

    int ypm = (int)(ypmax * 1.05f);
    g.setColor(Color.white);
    g.fillRect(0,0,getWidth(), getHeight());
    g.setColor(Color.black);
    g.drawRect(xpmin, ypmin, xpmax, ypm - ypm);
    g.setColor(colors[0]);

    for(int i=0; i< x.length; i++)
    {
        g.setColor(colors[i]);
        xp = calcx(x[i]);
        yp = calcy(y[i]);
        g.fillRect(xp, yp, calcx(1.0f), ypm-yp);
    }
}

```

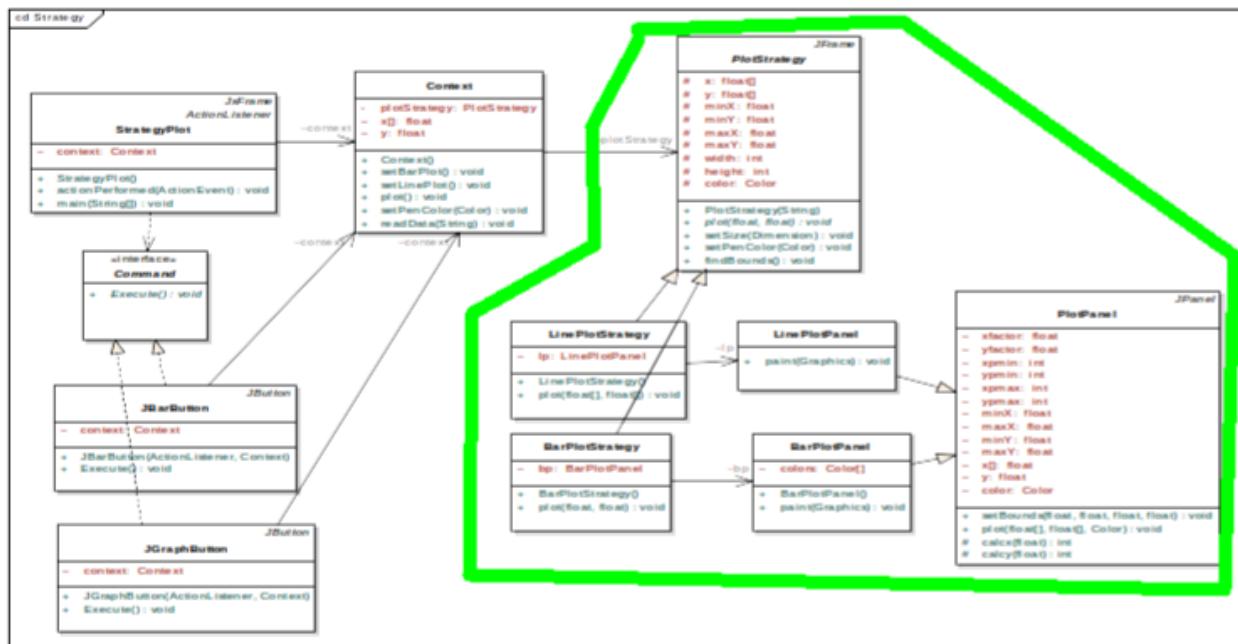
¿Cómo usarlo?

Seleccionar la alternativa desde el contexto.

Consecuencias

Ventajas/beneficios: Aclara los algoritmos al eliminar la lógica de selección, simplifica la clase al mover los algoritmos a diferentes clases, permite el cambio dinámico de algoritmo.

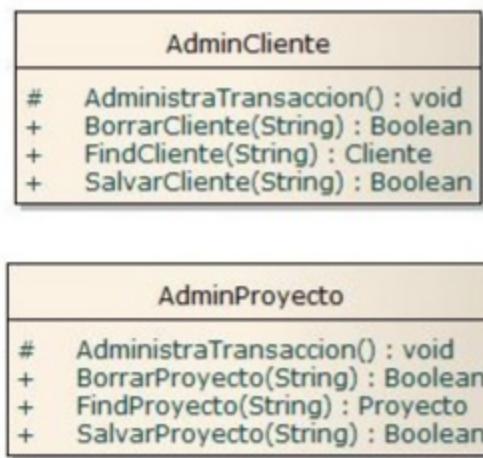
Desventajas: complica el diseño y el pasaje de datos.



Método template

¿Cuando usarlo?

Cuando existe un algoritmo predefinido con uno o más pasos indefinidos.



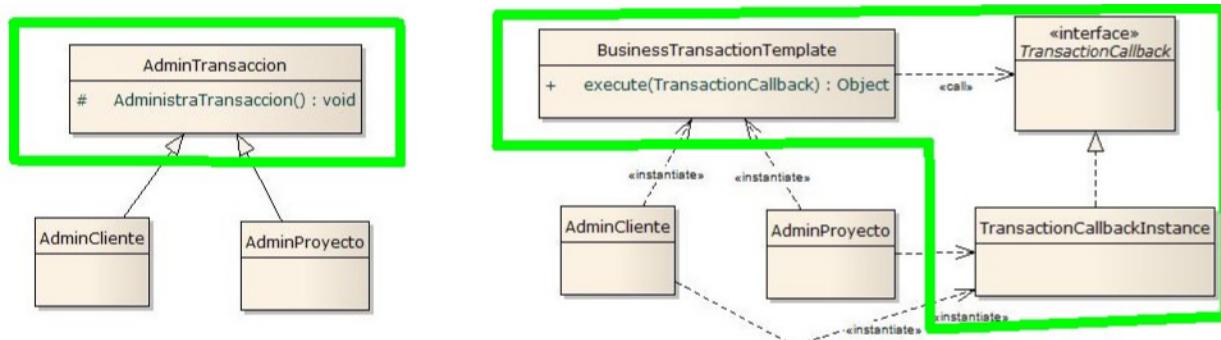
¿Cómo usarlo?

Generalizar en una clase el método del algoritmo pre definido y dejar la definición de los pasos indefinidos para que los definan las subclases.

Consecuencias

Ventajas/beneficios: evitar duplicación de código, documenta el algoritmo, facilita la definición.

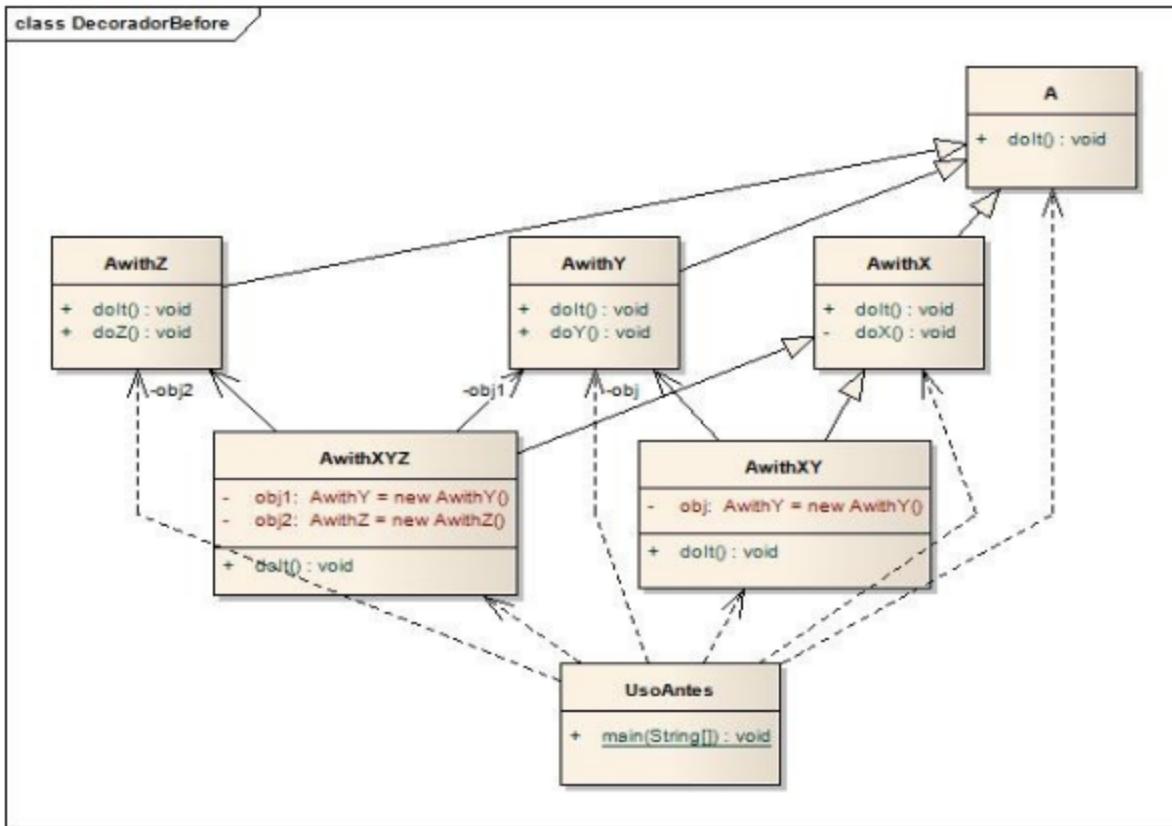
Desventajas: complica el diseño.



Decorador

¿Cuándo usarlo?

Contexto y problemas presentes en el diseño/código: clases a las cuales es necesario agregarles funcionalidad en forma flexible, en tiempo de ejecución.



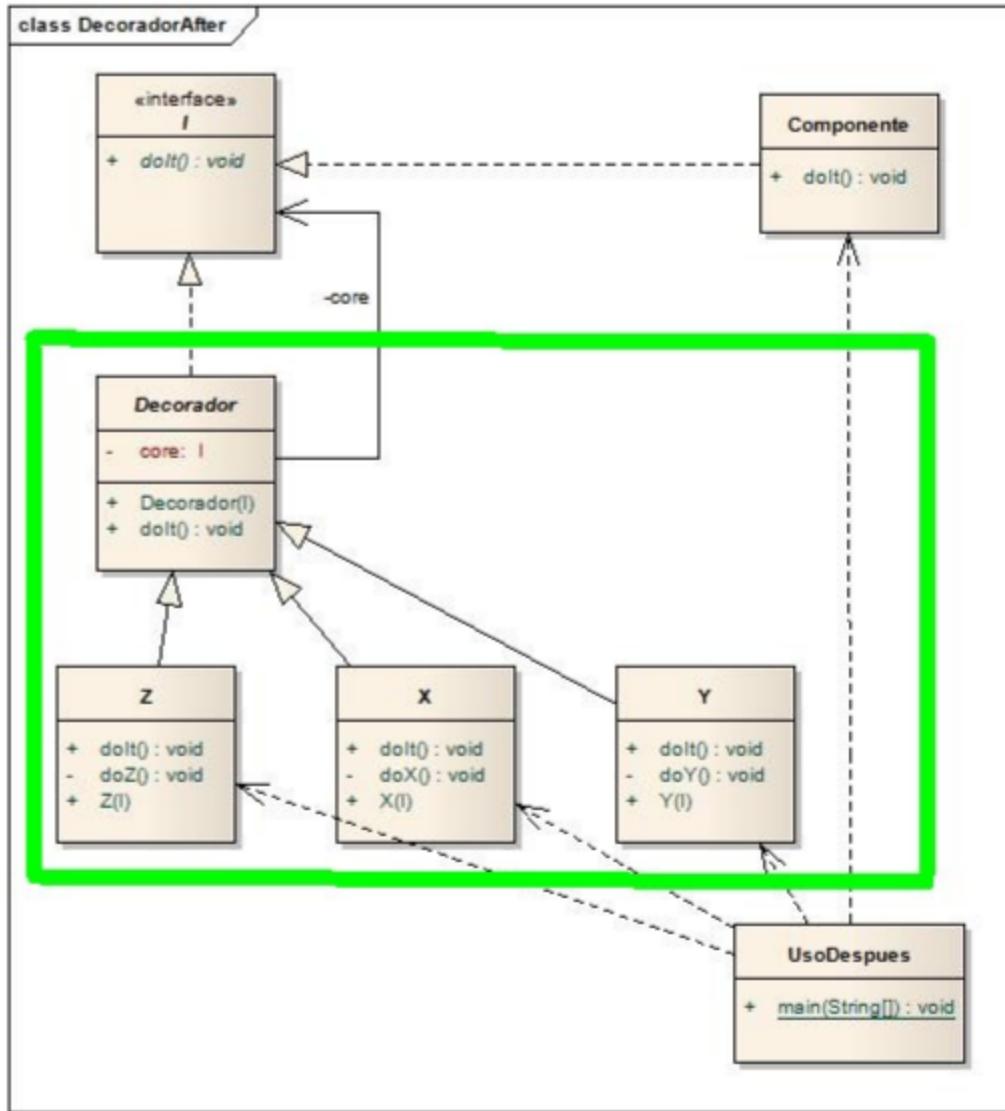
¿Cómo usarlo?

Simplifica las clases y las hace más livianas. Permite agregar la misma funcionalidad más de una vez.

Consecuencias

Ventajas/beneficios: facilita el desarrollo mientras avanza el diseño.

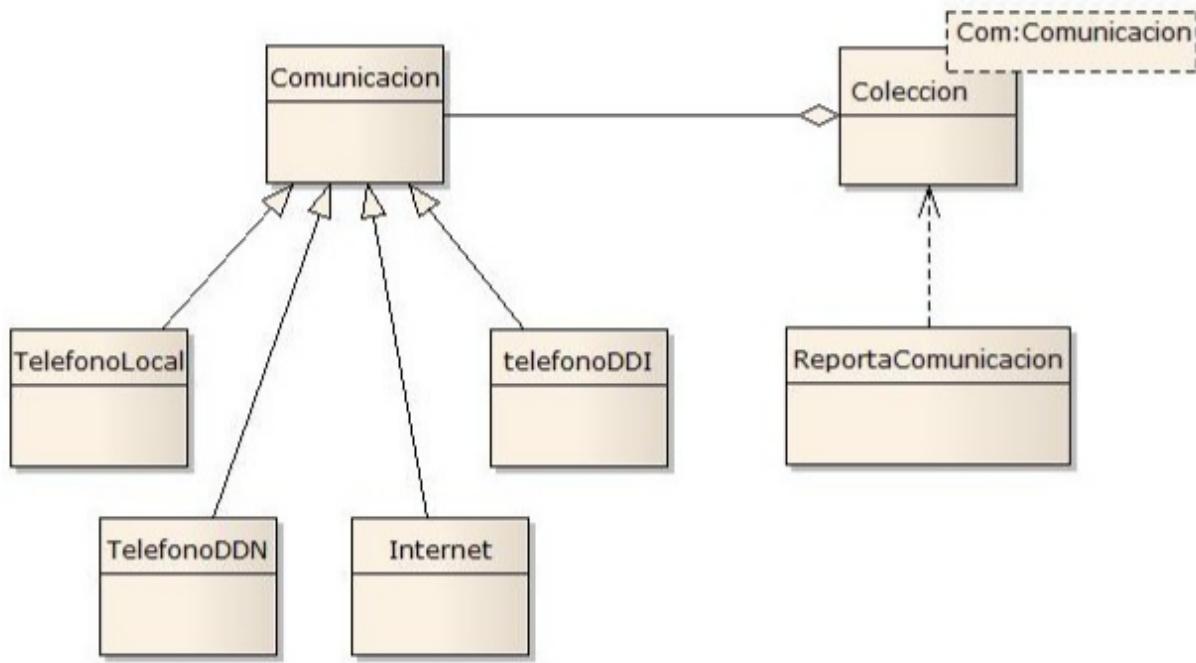
Desventajas: modifica la identidad de los objetos, hace al código difícil de entender y debuguear.



Visitor

¿Cuándo usarlo?

Cuando hay indefiniciones acerca de funcionalidades asignadas a determinadas clases. Cuando es necesario hacer acumulaciones para informar.



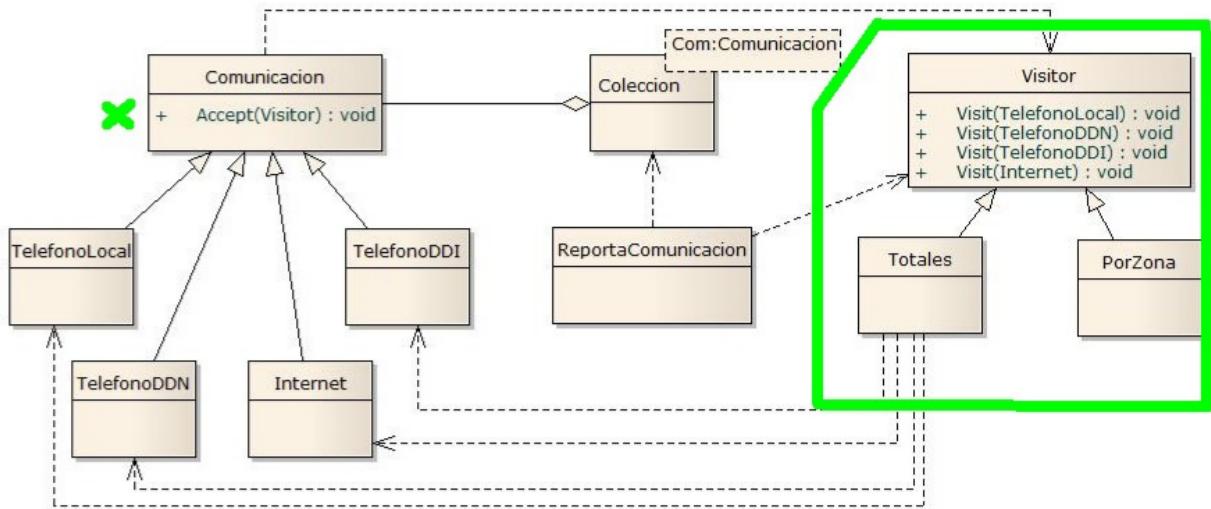
¿Cómo usarlo?

Implementar algoritmos de acumulación sobre jerarquías de clases, visitar clases en una o más jerarquías, evitar type casting al recorrer clases mientras se totalizan índices,

Consecuencias

Ventajas / beneficios: Extender el servicio prestado por clases sin modificarlas.

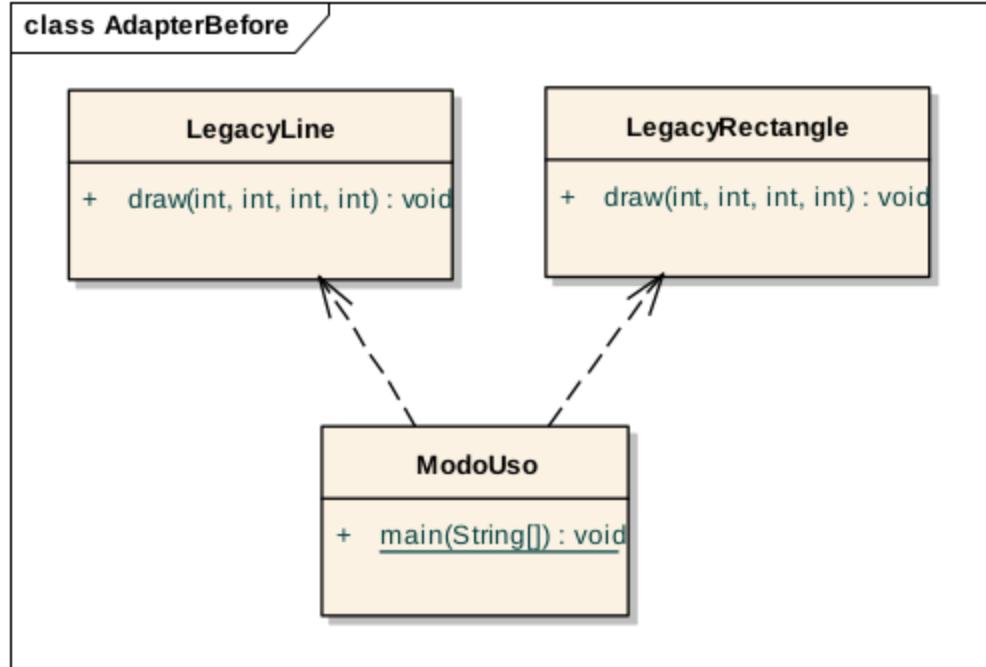
Desventajas: rompe con el encapsulamiento de las clases visitadas y complica el diseño cuando es necesario agregar Visitor extra.



Adapter

¿Cuándo usarlo?

Clases con funcionalidades útiles pero con interfaces diferentes. Cliente con protocolo ya definido es la interfaz a respetar.



¿Cómo usarlo?

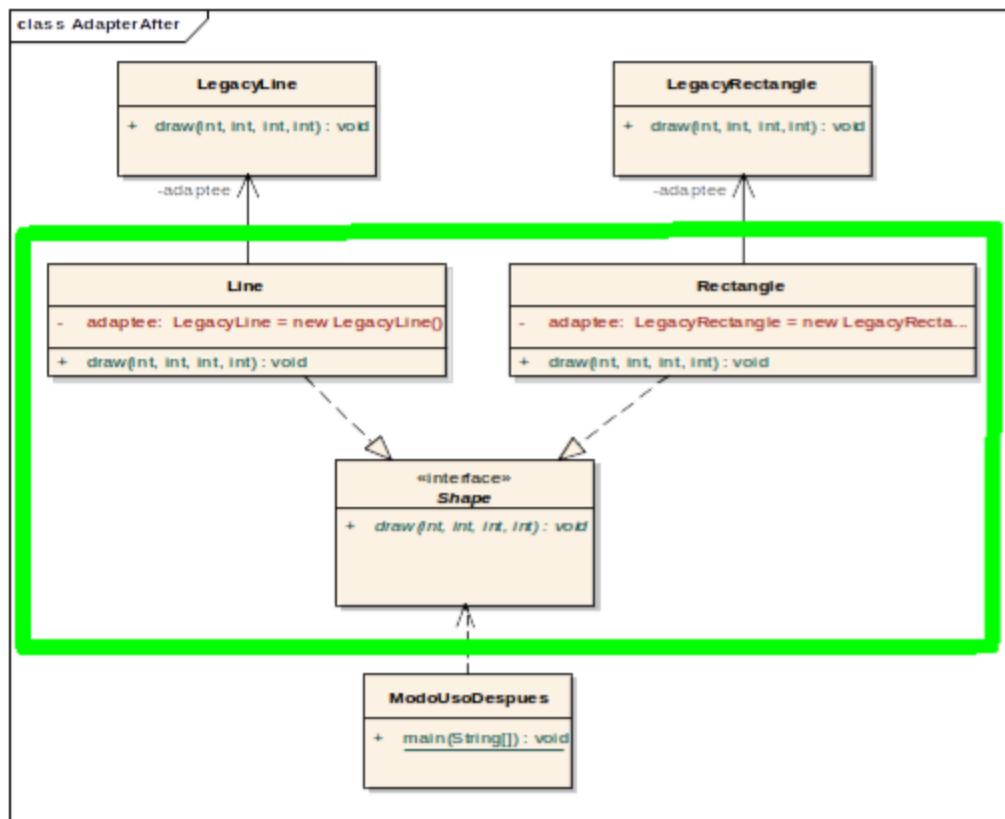
Habilita al código cliente a utilizar funcionalidades de clases con diferentes interfaces

adaptándolas. Simplifica y unifica el código cliente al adaptar la interfaz de las clases que proveen las diferentes funcionalidades.

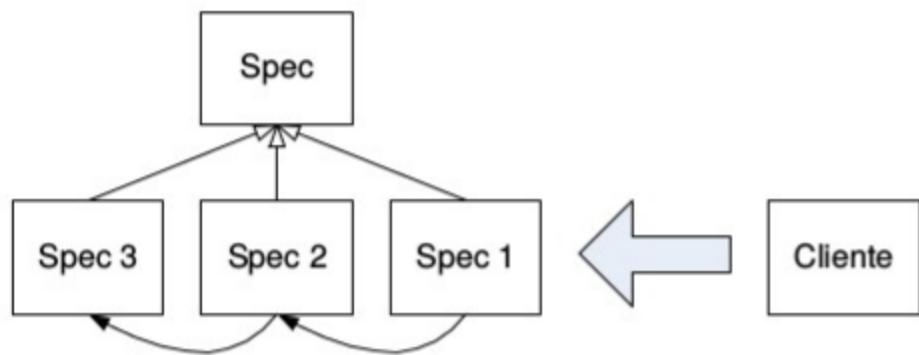
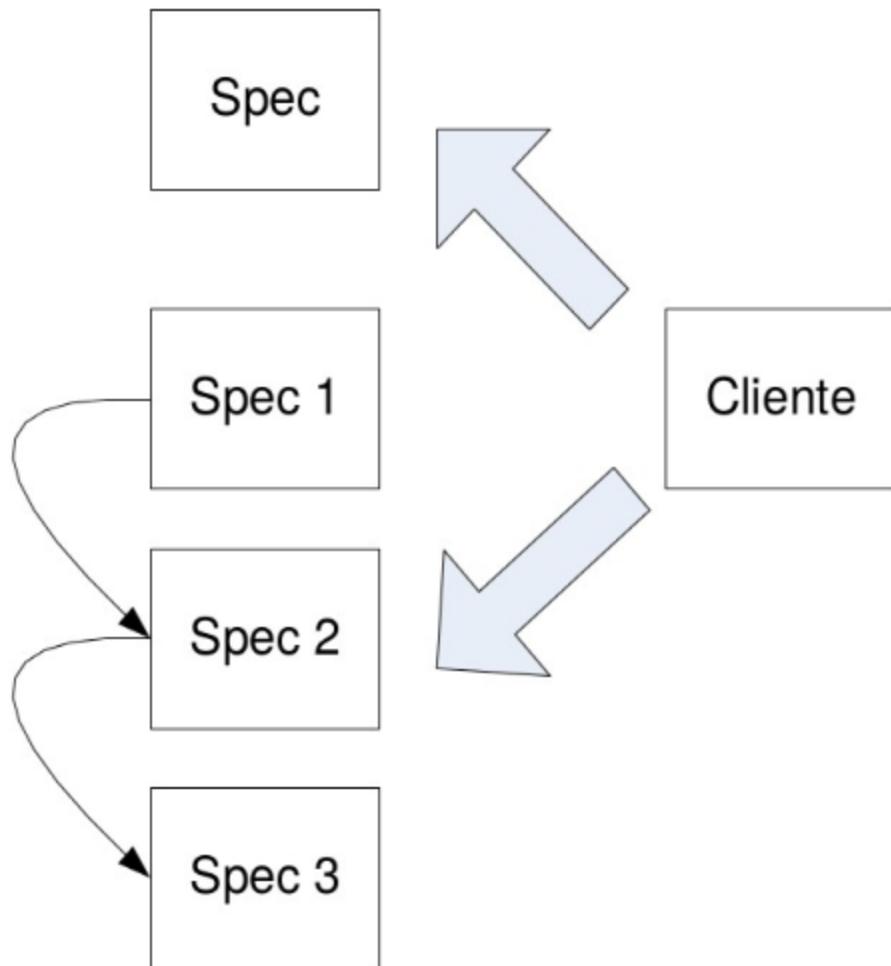
Consecuencias

Ventajas / beneficios: promueve y ordena el reuso.

Desventajas: a veces es difícil adaptar las interfaces y complica el diseño.



Composite



Machete

Categoría de Problemas	Categoría de Patrones		
	Arquitectura	Diseño	Idiom
Cimientos	Layers Pipes- Filters		
Sistemas Distribuidos	Broker		
Sistemas Interactivos	Model View Controller		
Sistemas Adaptables	Microkernel Reflection		
Creación		Abstract Factory Prototype Builder	Singleton Factory Method
Descomposición Estructural		Whole Part Composite	
Organización del Trabajo		Chain of Responsibility Command Mediator Master-Slave	
Control de Acceso		Proxy Facade Iterator	
Variación de Servicios		Bridge Strategy State	Template Method
Extensión de servicios		Decorator Visitor Extensión de Interfaz Extensión de Objeto	
Administración		Memento	
Adaptación		Adapter	
Comunicación		Forwarder-Receiver Client-Dispatcher-Server Publisher-Subscriber	
Manejo de Recursos		Flyweight	Counter Pointer

Métricas

Teoría

¿Para qué sirven las medidas tomadas sobre el diseño y el código?

Permiten:

1. Caracterizar el diseño de sistemas utilizando programación orientada a objetos

2. Evaluar y mejorar el diseño de sistemas
3. Guiar y conducir revisiones de diseño y código como mecanismo de garantía de calidad

Definiciones de medidas

NOP: nro de paquetes o namespaces en el sistema

NOC: nro de clases en el sistema

NOM: nro de métodos x clase

LOC: nro de líneas de código x método

CYCLO: nro de decisiones x líneas de código (Complejidad Ciclomática)

WMC: CYCLO / Metodos x LOC / Metodo x NOM / clase

AMW: CYCLO / Metodo (average method weight)

Métricas de código

Tamaño y Complejidad

Permiten comparaciones independientes del tamaño de los proyectos. Se calculan como la razón entre medidas adyacentes como se muestra en el figura.

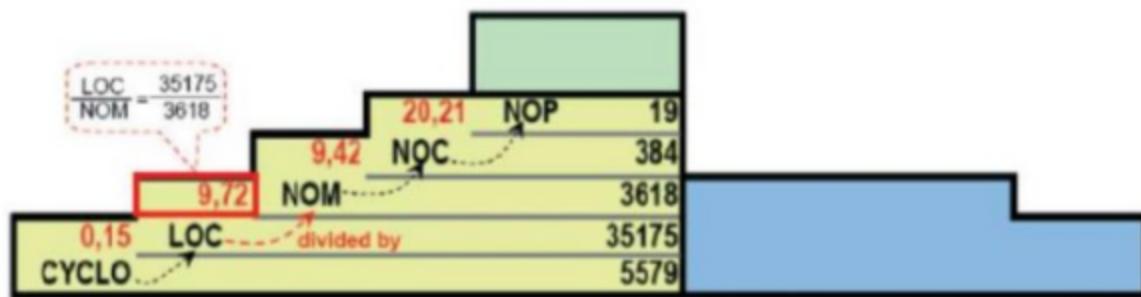


Fig. 3.2. Size and complexity characterization.

Medidas de Acoplamiento

CALLS: nro de invocaciones a cualquier método de una clase desde diferentes métodos de otras

FANOUT: nro de clases dependientes (uso, invocaciones)

Las razones calculadas tienen el siguiente significado:

FANOUT / CALLS: cantidad de invocaciones salientes dividida la cantidad de llamadas

entrantes

CALLS / NOM: cantidad de llamadas entrantes dividida la cantidad de metodos

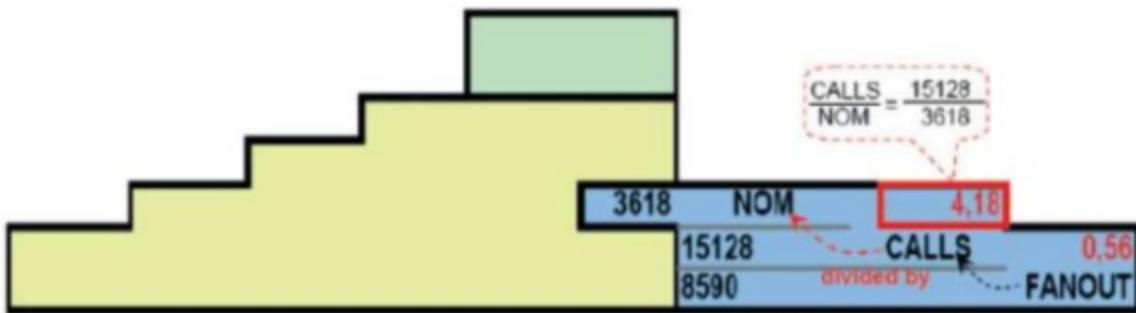


Fig. 3.3. Characterizing a system's coupling.

Medidas de Herencia

ANDC: Average Number of Derived Classes. Solo se contabilizan clases (no interfaces) propias (no de librerías).

$$\text{ANDC} = \text{sumatoria} (\text{nro clases} \times \text{nro herencias por clase}) / \text{NOC}$$

AHH: Average Hierarchy Height

$$\text{AHH} = \text{sumatoria}(\text{nro clases root} \times \text{nro niveles de herencia}) / \text{NOC}$$

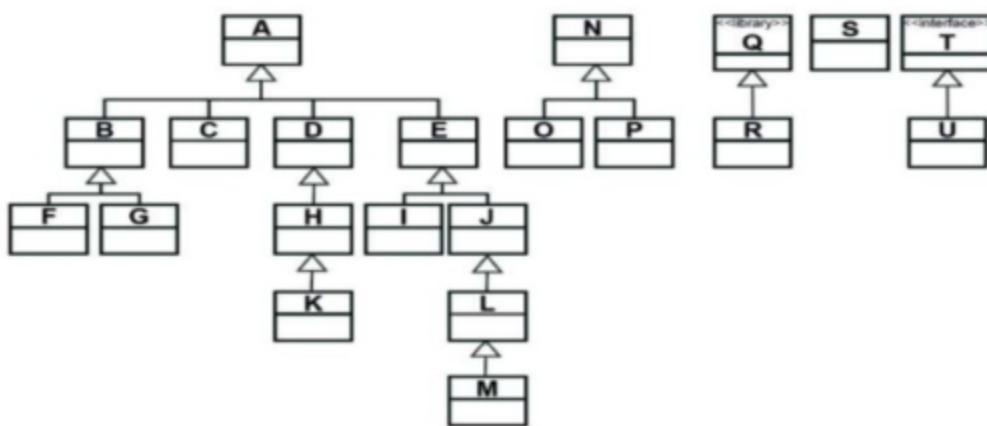
Ejemplo

Nro de clases total = 19, sin contar Q que es de una librería y T que es una interfaz.

$$\text{ANDC} = (11 \text{ clases} \cdot 0 \text{ herencia} + 4 \text{ clases} \cdot 1 \text{ herencia} + 3 \cdot 2 + 1 \cdot 4) / 19 = 0.73$$

Nro. total de root clases 5 (A, N, R, S, U)

$$\text{AHH} = (4 \text{ niveles de herencia (A)} + 1 \text{ (N)} + 0 \text{ (R)} + 0 \text{ (S)} + 0 \text{ (U)}) / 5 = 1$$



Interpretación

Metric	Java			C++		
	Low	Average	High	Low	Average	High
CYCLO/Line of code	0.16	0.20	0.24	0.20	0.25	0.30
LOC/Operation	7	10	13	5	10	16
NOM/Class	4	7	10	4	9	15
NOC /Package	6	17	26	3	19	35
CALLS/Operation	2.01	2.62	3.2	1.17	1.58	2
FANOUT /Call	0.56	0.62	0.68	0.20	0.34	0.48
ANDC	0.25	0.41	0.57	0.19	0.28	0.37
AHH	0.09	0.21	0.32	0.05	0.13	0.21

Table 3.1. Statistical thresholds of 45 Java and 37 C++systems computed for the proportions (ratios) used in this *Overview Pyramid*.

Sumamos colores



Fig. 3.6. Using colors to interpret the *Overview Pyramid*. BLUE means a *low* value; GREEN means an *average* value; RED stands for a *high* value.

Interpretación de la Complejidad

CYCLO / LOC = **0.15** evidencia que la complejidad es baja en relación al umbral.

LOC / NOM = **9.72** evidencia que el tamaño de los métodos es normal en relación al umbral.

NOM / NOC = **9.42** evidencia que el tamaño de las clases es excesivo en relación al umbral.

Interpretación del Acoplamiento

CALLS / NOM : **4.18** evidencia gran acoplamiento

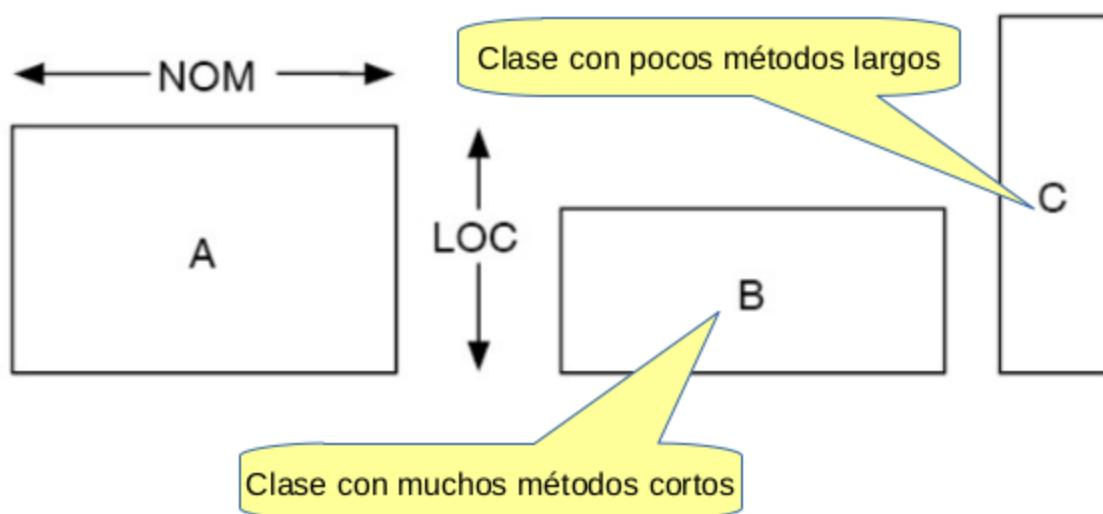
FANOUT / CALL = **0.56** evidencia que el acoplamiento está concentrado a unas pocas clases.

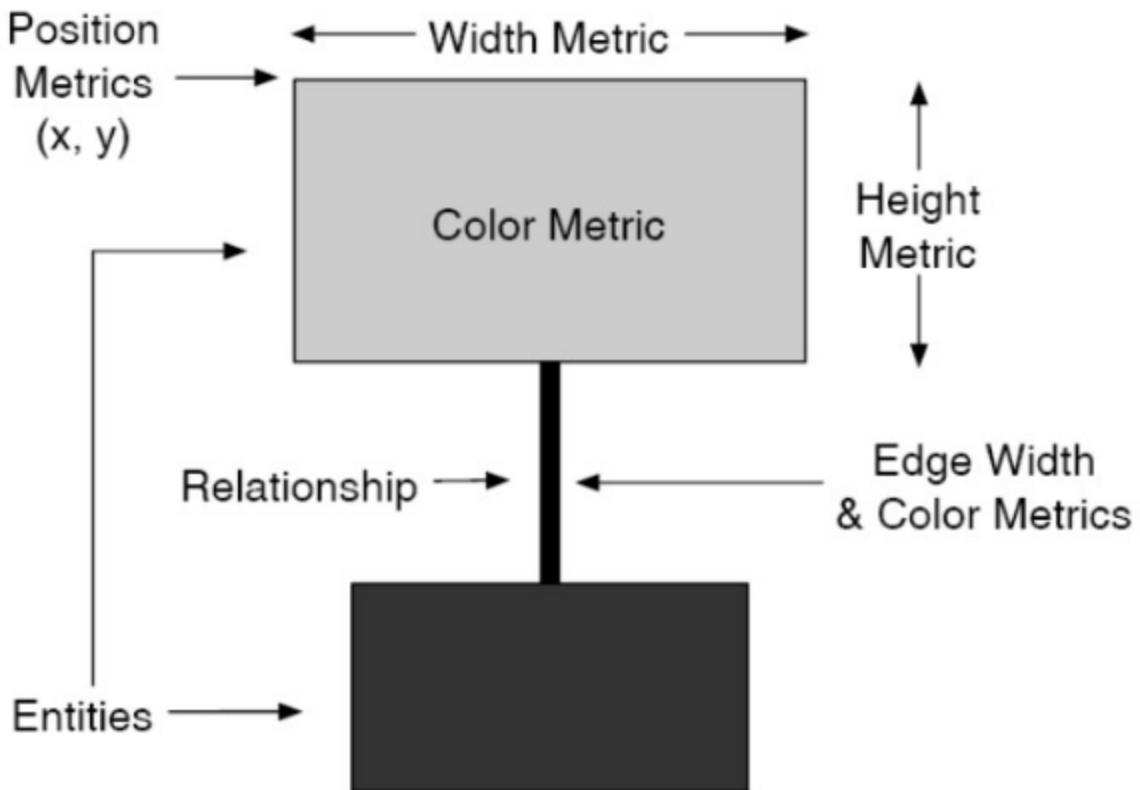
Interpretación de la Herencia

ANDC = **0.31** muestra uso bajo de herencia AHH = **0.12** indica que las jerarquías no son profundas sino horizontales.

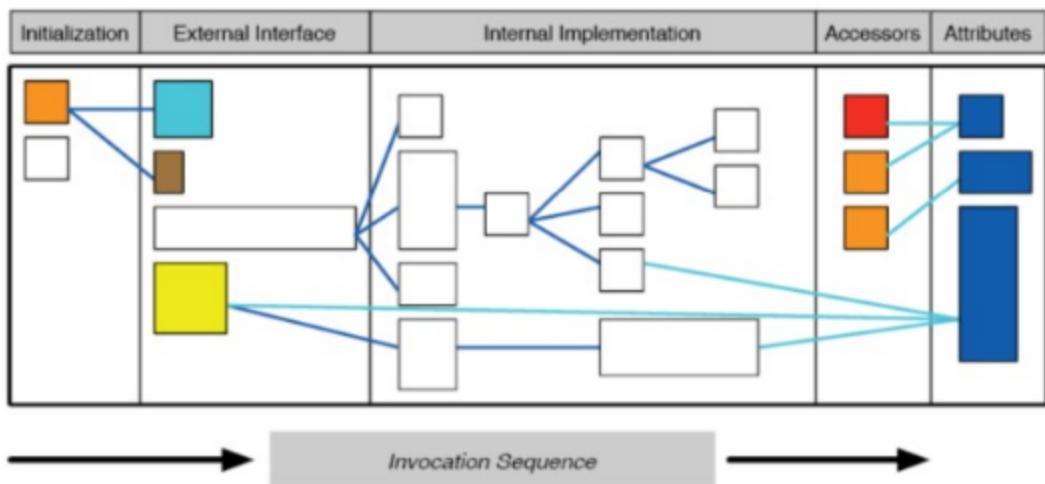
Métricas y visualización estática

Visualización Pomimétrica (estática)





Polimétricas y Class Blueprint



Las capas en que se descompone una clase en un blueprint se muestran en la figura anterior y son:

Inicialización: incluye constructores y métodos de inicialización

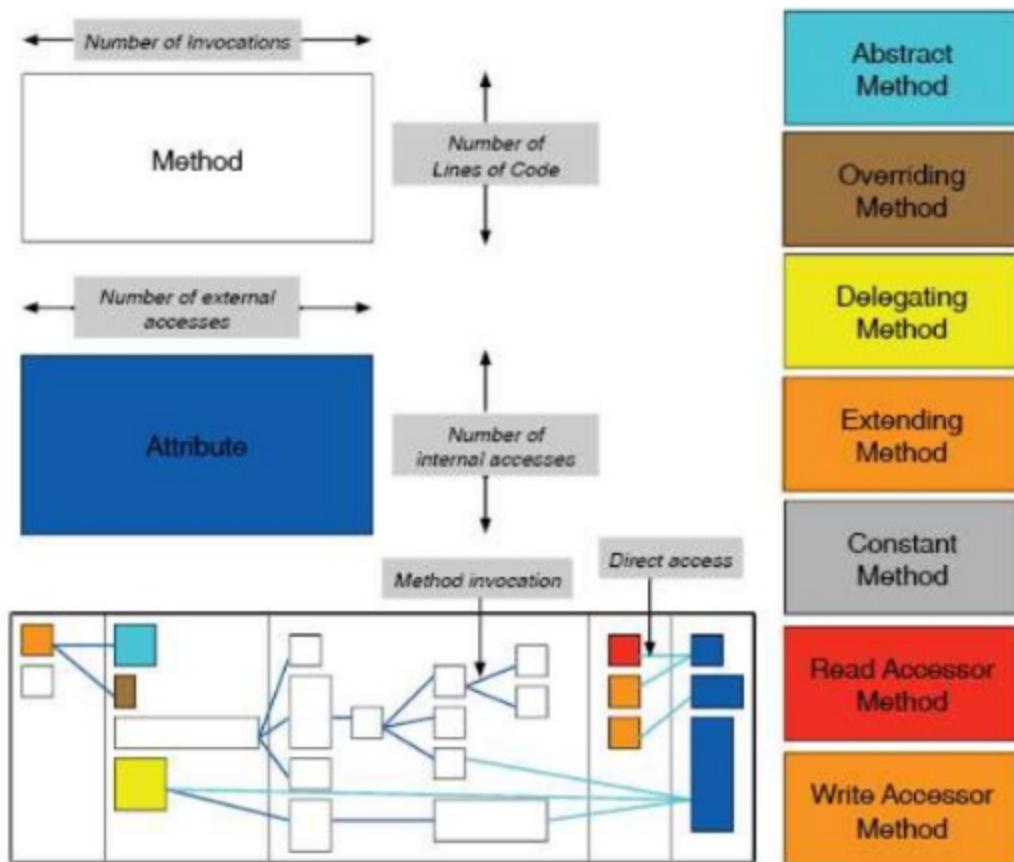
Interfaz externa: incluye los métodos públicos de conducción del negocio

Implementación interna: métodos privados

Accesores: métodos de acceso a atributos

Atributos: atributos

También se indica la invocación de los diferentes métodos mostradas como líneas que conectan dichos métodos.



Overview Pyramid: brinda un resumen construido en base a métricas simples y sirve para caracterizar un sistema en términos de tamaño, complejidad, acoplamiento y herencia.

Polymetric Views: brinda un medio simple y poderoso de visualización de la complejidad

Medidas de falta de armonía

Armonía de tipo Identidad

Armonía entre la clase que modela un concepto y los datos y métodos que operan sobre esos datos.

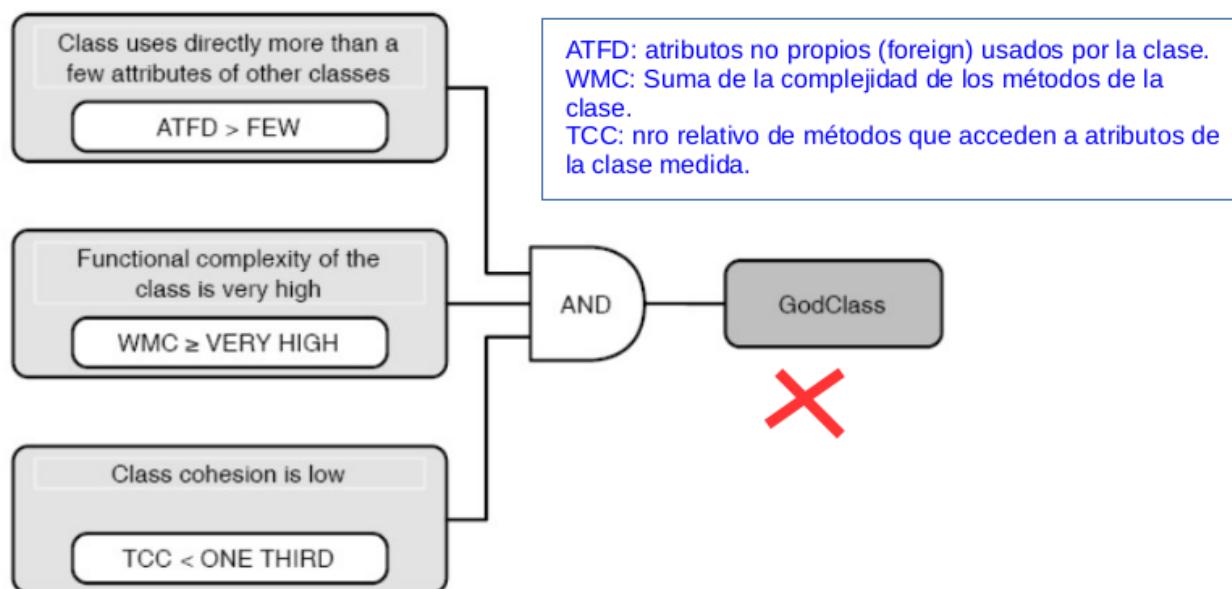
Armonía de tipo Colaboración

Armonía en la forma en que colabora cada clase con el resto a efectos de lograr la funcionalidad pedida.

Armonía de tipo Clasificación

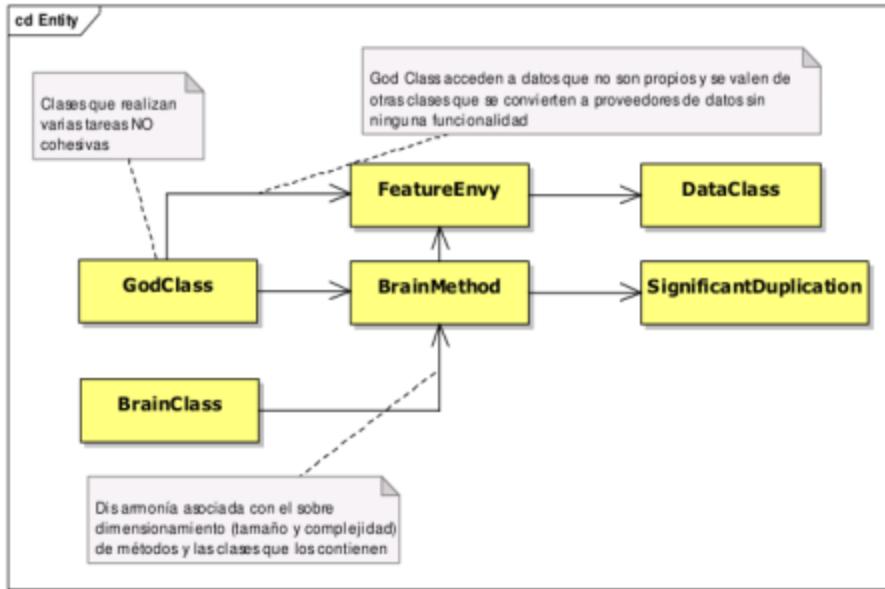
Cómo cada clase hace uso de los métodos heredados y los propios. Cuál es el comportamiento de cada clase en la jerarquía de herencia a la cual pertenece.

Detección de falta de armonía



Tipo Entidad - Criterios de buen diseño

1. Métodos y clases deben tener tamaños no excesivos. Ver métricas simples de la triada.
2. Cada clase posee una interfaz que define su comportamiento al mundo exterior a través del conjunto de servicios que brinda. Criterio de Segregación de interfaces.
3. Los datos y las operaciones colaboran armónicamente en el interior de la clase a la cual pertenecen. Principio de programación orientada a objetos.



Detección

Clases que contienen una alta cantidad de desarmonías en sus métodos tienen prioridad.

Clases que incluyen más de una desarmonía de tipo Entidad tienen prioridad.

Clases que son afectadas por otro tipo de desarmonías son tratadas primero para encontrar también otros aspectos

Soluciones

1. Remover duplicaciones refactorizando (extraer método para intra clases y extraer super class para las inter clases).
2. Remover atributos temporales (aquellos usados en un solo método) y reemplazarlos por variables locales. Esto está orientado a reducir fuentes de baja cohesión.
3. Reducir la utilización de datos no propios, moviendo el método que los usa a la clase propietaria de los datos o mover los datos usados como atributos a esta clase.
4. Extraer método hasta contar con métodos atómicos y cohesivos

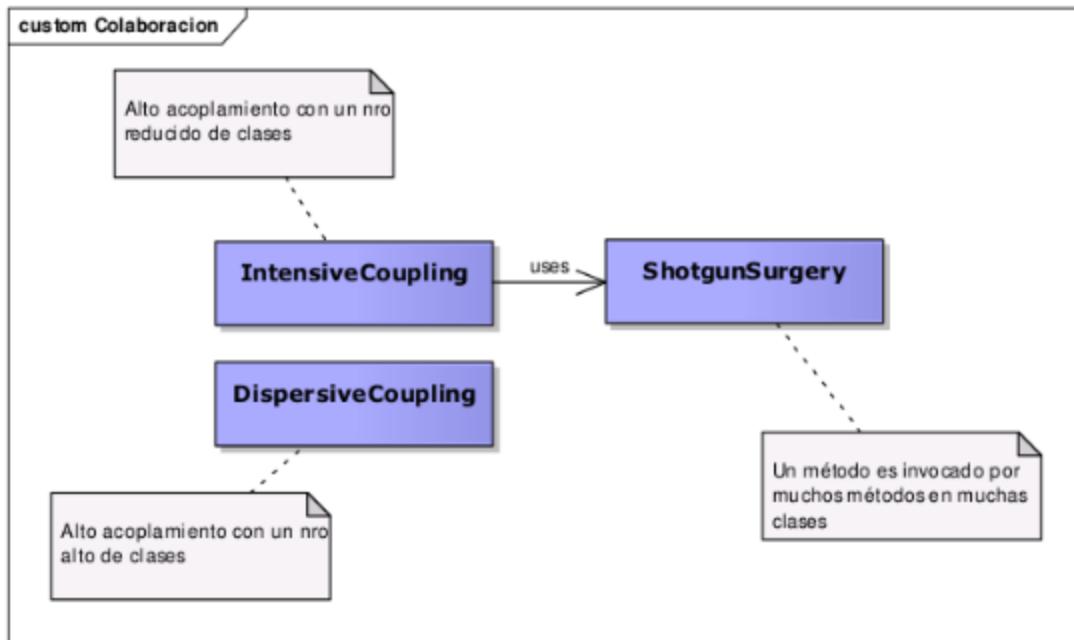
Tipo Colaboración - Criterios de buen diseño

Bajo acoplamiento entre clases: a través de invocación de métodos con extensión (nro limitado de clases), intensidad (unidireccionales) y dispersión (mismo nivel abstracción,

otro nivel, otro package) limitadas.

Excesivo funout -> muy inestable

Excesivo funin -> inmutable



Detección

Para detectar Acoplamiento Intensivo se buscan clases cuyos métodos invocan un grupo de métodos de una única clase.

Para detectar Acoplamiento Dispersivo se buscan clases cuyos métodos invocan métodos de un grupo de clases.

Para detectar ShotgunSurgery se buscan métodos llamados por muchos otros métodos de diferentes clases.

Soluciones

Definir servicios más complejos a partir de métodos cohesivos que implementen el servicio completo y que al invocarlos solo a ellos disminuyan el acoplamiento intensivo.

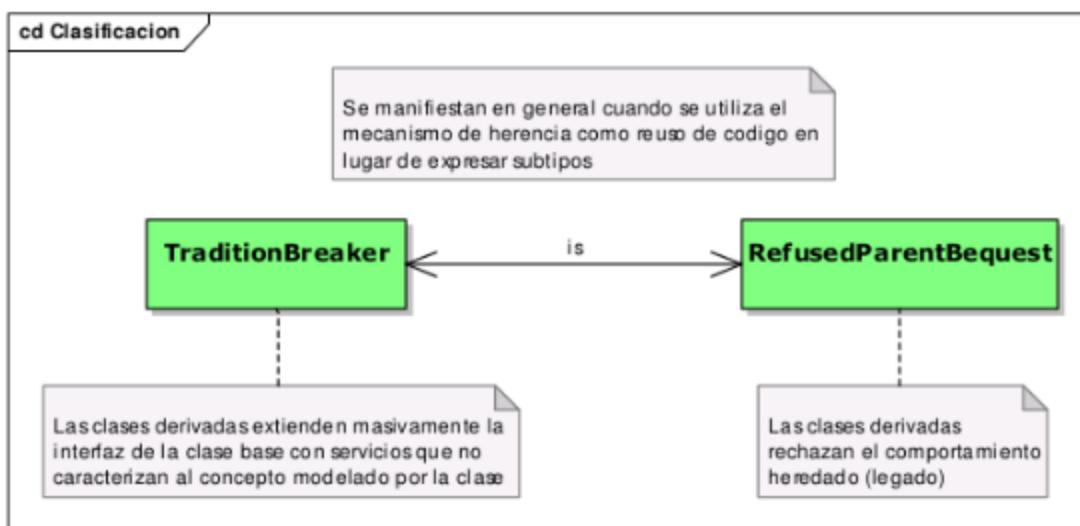
1. Si el caso de acoplamiento dispersivo es un BrainMethod, luego ataque esa disarmonía.
2. Si es posible, mover funcionalidad invocada con el fin de reducir el acoplamiento dispersivo.
3. En los casos de acoplamiento mencionados la estrategia a usar está guiada por "Mover"

comportamiento lo más cerca de los datos” y “Eliminar el código de navegación” ya que es muy frecuente la mala asignación de responsabilidades a las clases en cuestión.

4. Resolver la ShotgunSurgery tiene relación con la Law of Demeter: relacionar objetos a partir del menor conocimiento posible de la estructura de relaciones y comportamiento asociado a las instancias.

Tipo Clasificación - Criterios de buen diseño

1. Las clases deben estar organizadas en jerarquías armónicas, ni demasiado profundas ni demasiado anchas.
2. Cada clase debe estar en armonía con las clases de las cuales deriva. Es decir, extender las interfaces, especializar el comportamiento y decrecer la abstracción.
3. Las clases deben presentar armonía en la colaboración dentro de la jerarquía a la cual pertenecen. Es decir, las dependencias deben ser bottom – up y el comportamiento debe ser especializado (redefinido) más que sumar nuevos para implementar uno ya definido en las clases bases.



Detección

Las clases derivadas no redefinen métodos de su clase base

Las clases además tienen cierto tamaño, medido en número de métodos y complejidad (Refused)

El tamaño de la interfaz pública de la clase hija ha crecido mucho y su tamaño y

complejidad

son altos y la clase base tiene una cierta funcionalidad definida (Breaker)

Soluciones

Hay tres casos de RefusedParentBequest

1. Falsa herencia: las clases no tienen una relación directa de herencia sino con una superbase, luego extraer la clase de la jerarquía.
2. Legado Irrelevante: las clases derivadas no utilizan los métodos protected ni los redefinen luego hacerlos privados en la base.
3. Legado Discriminatorio: la clase base tiene un gran número de clases derivadas y ofrece un legado que tiene sentido solo para algunas y no para otras. La figura muestra una posible solución, que de ser aplicable denota que la clase base fue usada para modelar más de un concepto y no uno único.

Hay cuatro casos de TraditionalBreaker

1. Tradición Irrelevante: excesivos métodos públicos, deberían ser privados o protected.
2. Tradición negada: la clase base no incluye un conjunto de servicios que son implementados en casi todas las hijas.
3. Subclases con Doble Intencionalidad: clases hijas que implementan funcionalidad completamente diferente a la expresada por la interfaz de la base. Se debe extraer esta funcionalidad a otra clase fuera de la jerarquía.
4. Subclases Mal Relacionadas: las interfaces de la hija y la base no tienen nada que ver.

No alcanza con analizar clases aisladas, se deben analizar jerarquías. ¿Cómo se agrupan las clases afectadas y cuáles tienen prioridad?

- Tienen prioridad las jerarquías con mayor número de clases afectadas.
- Las jerarquías con más niveles de profundidad afectados es mayor.

- Las jerarquías con mayor diversidad de desarmonías tienen prioridad.
- Las jerarquías con otro tipo de desarmonías tienen prioridad.