

Patrones de Arquitectura

Layer

Contexto: se utiliza cuando se busca dividir a la aplicación en distintos niveles (layers) de abstracción.

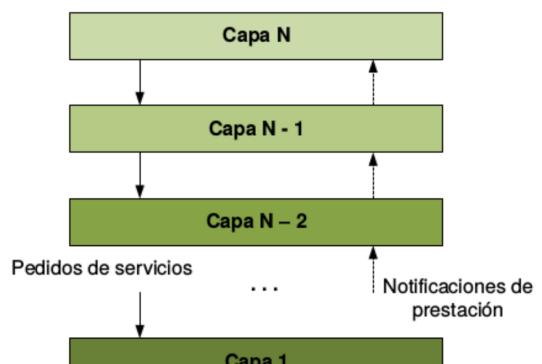
Cada layer tiene como responsabilidad brindar servicios a las layers superiores y delegar las subtareas a las layers inferiores.

Ventajas:

- + Reuso de las capas
- + Cambiabilidad
- + Desarrollo simultáneo
- + Las dependencias son locales
- + Soporta la estandarización

Desventajas:

- + Requiere un diseño más elaborado
- + Baja eficiencia
- + Dificultad para establecer la granularidad de la capa: cuando tenes pocas layers no explotas el patrón, mientras que cuando hay demasiadas se vuelve demasiado complejo, ergo no hay punto medio

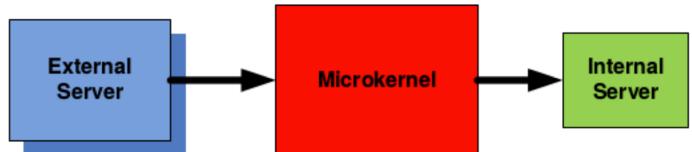


Microkernel

Contexto: se utiliza cuando se quiere desarrollar distintas aplicaciones que utilizan interfaces similares que se basan en la misma funcionalidad central.

Estructura:

- + Microkernel:
 - + Provee mecanismos básicos.
 - + Ofrece facilidades para la comunicación.
 - + Encapsula las dependencias del sistema.
 - + Maneja y controla recursos.
- + Internal Server:
 - + Implementa servicios adicionales.
 - + Encapsula algunas especificaciones del sistema.
- + External Server:
 - + Provee interfaces para los usuarios



El flujo de comunicación es: External Server --> Microkernel --> Internal Server

Ventajas:

- + Escalabilidad / Soporta cambios con menor impacto
- + Portabilidad
- + Flexibilidad y extensibilidad
- + Separación de policies y mecanismos
- + Fiabilidad
- + Transparencia

Desventajas:

- + Mala performance (en comparación con un monolito)
- + Diseño e implementación complejos

Pipes and Filters

Contexto: se utiliza cuando se quiere procesar un stream de datos y el procesamiento es factible de descomposición en N procesamientos atomicos. O sea, para procesar stream de datos.

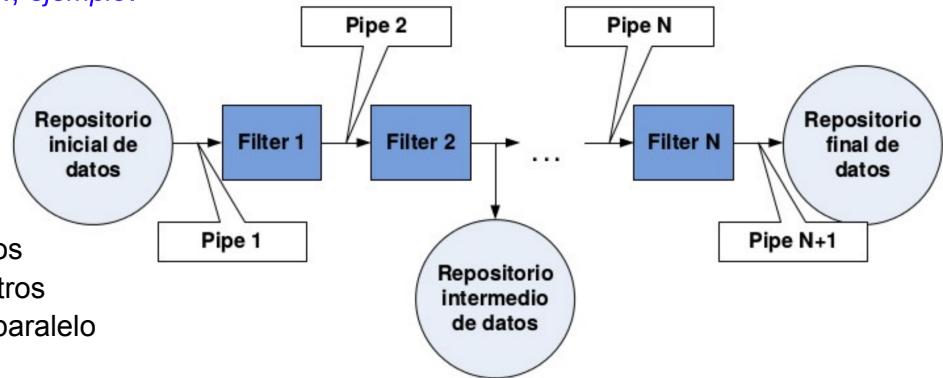
Estructura:

- + Filter:
 - + Obtener la input data
 - + Realiza operaciones sobre la input data
 - + Devuelve la data procesada
 - + Agrega, refina o transforma la input data

+ Pipe:

- + Transfiere data
- + Almacena data
- + Conecta los filtros

1, *ejemplo.*



Ventajas:

- + Reuso de los filtros
- + Rápido tipado de pipelines
- + No se necesitan archivos intermedios
- + Flexibilidad por intercambio entre filtros
- + Eficiencia por el procesamiento en paralelo

Desventajas:

- + Compartir el estado de la información es costoso
- + Sobretransformación de la data
- + Manejo de errores

Model-View-Controller (MVC)

Contexto: divide a una aplicación interactiva en tres componentes. El *Model* contiene las funcionalidades core y la data. *Views* se encarga de mostrar la información al usuario, y los *Controllers* se encargan de manejar los inputs de los usuarios.

Se utiliza para obtener aplicaciones interactivas con una interfaz 'human-computer' flexible.

- **OBS:** Views + Controlles = UI

Estructura:

+ Model:

- + Provee de un core funcional para la aplicación
- + Registra vistas y controladores dependientes
- + Notifica a componentes dependientes sobre cambios en la data

+ View:

- + Crea e inicializa el controller asociado
- + Muestra la información al usuario
- + Implementa el proceso de actualización
- + Recupera la data del modelo

+ Controller:

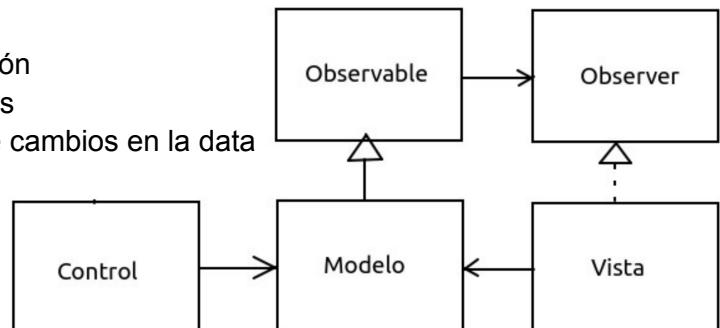
- + Acepta los inputs del usuario como eventos
- + Traduce el evento en un *service request* para el Modelo o una *display request* para la Vista
- + Implementa el proceso de actualización en caso de ser necesario

Ventajas:

- + Reusabilidad y facilidad de cambios y extension
- + Múltiples vistas del mismo modelo
- + Vistas sincronizadas
- + 'Pluggable' vistas y controladores
- + Framework potencial

Desventajas:

- + Mayor complejidad
- + Conexión íntima entre vista y controlador
- + Posibilidad de un número excesivo de actualizaciones
- + Acceso a la data ineficiente en la Vista
- + Dificultades para usar MVC con herramientas modernas de UI



Broker

Contexto: se utiliza para estructurar sistemas distribuidos con componentes desacoplados que interactúan por invocaciones de servicios remotos. Un componente Broker es responsable de coordinar la comunicación, ya sea reenviando una request como también transmitiendo resultados y excepciones.

Estructura:

- + Server: implementa objetos que exponen su funcionalidad a través de una interfaz que consiste de operaciones y atributos.
- + Clients: aplicaciones que acceden a los servicios de al menos un server.
- + Broker: es responsable de la transmisiones de request de los clientes a los servers, como también de las respuestas.
- + Client-side proxies: representan una capa entre los clientes y el broker.
- + Server-side proxies: análogo al caso anterior.
- + Bridges: son componentes adicionales usados para ocultar los detalles de implementación cuando dos brokers se encuentran operando.

Resumiendo, las responsabilidades de cada parte de la estructura son:

Client:

- + Implementa las funcionalidades para el usuario
- + Envía request al server a través del client-side proxy

Server:

- + Implementa los servicios
- + Se registra a sí mismo con el Broker local
- + Envía respuestas y excepciones a los clientes a través de un server-side proxy

Broker:

- + (Un)registers servers
- + Ofrece las APIs
- + Transfiere los mensajes
- + Error recovery
- + Interopera con otros brokers a través de Bridges
- + Localiza servidores

Client-side Proxy:

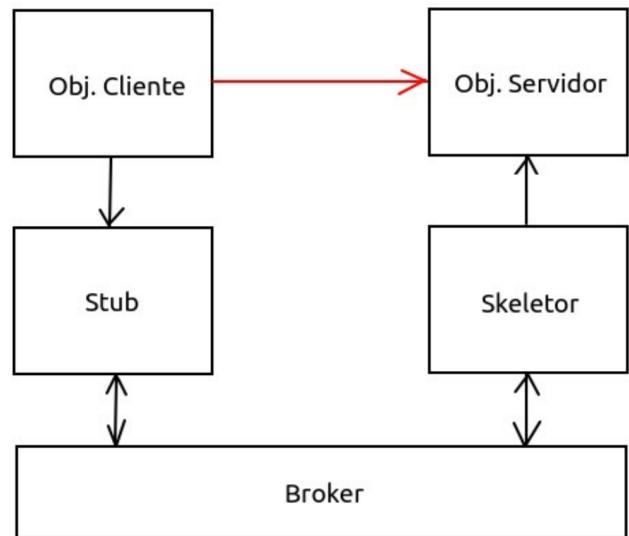
- + Encapsula funcionalidades específicas del sistema
- + Mediador entre el Client y el Broker

Server-side Proxy:

- + Llama a los servicios dentro del server
- + Encapsula funcionalidades específicas del sistema
- + Mediador entre el Server y el Broker

Bridge:

- + Encapsula funcionalidades específicas de la red
- + Mediador entre el Local Broker y el bridge de un Broker remoto



Ventajas:

- + Cambiabilidad
- + Portabilidad
- + Reusabilidad
- + Transparencia de ubicación
- + Cambiabilidad y extensibilidad de los componentes
- + Portabilidad de un sistema Broker
- + Interoperabilidad entre distintos sistemas Broker

Desventajas:

- + Baja eficiencia
- + Poca tolerancia a fallas
- + Dificultad de desarrollo

EA (Enterprise Architecture)

Contexto: Se utiliza cuando el sistema es cliente servidor y el servidor es remoto, también cuando hay usuarios concurrentes, múltiples interfaces de usuario reglas de negocio y datos persistentes.

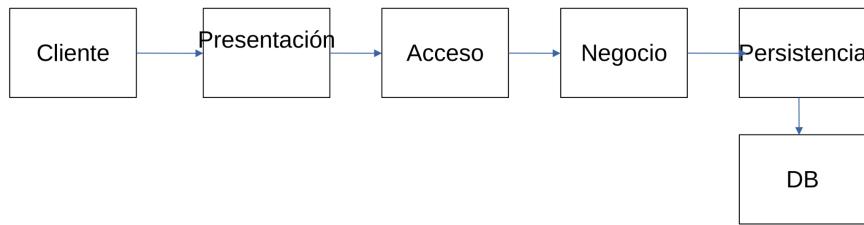
Se utiliza desacoplando las diferentes "tiers" logrando separación de incumbencias y reuso

Ventajas:

- + Reuso
 - + Cambiabilidad

Desventajas:

- + Mayor trabajo



Patrones de Diseño

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
Object	Abstract Factory (87)	Adapter (object) (139)	Chain of Responsibility (223)	
	Builder (97)	Bridge (151)	Command (233)	
	Prototype (117)	Composite (163)	Iterator (257)	
	Singleton (127)	Decorator (175)	Mediator (273)	
		Facade (185)	Memento (283)	
		Flyweight (195)	Observer (293)	
		Proxy (207)	State (305)	
			Strategy (315)	
			Visitor (331)	

Creacionales (factory, factory method, builder)

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.

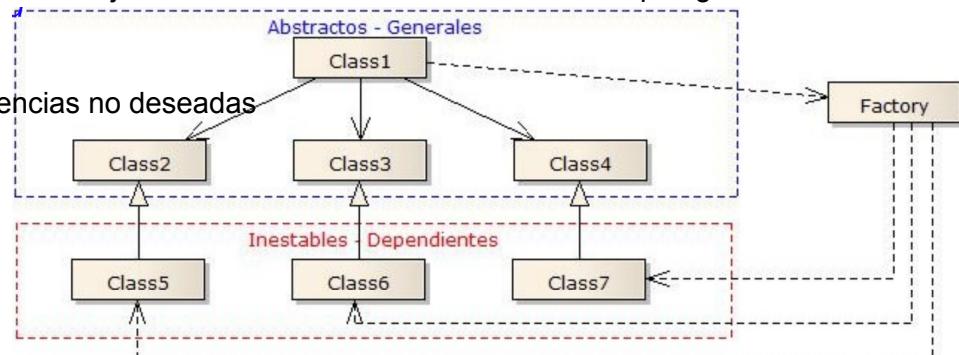
Cómo usarlo: No se puede instanciar objetos utilizando los constructores de objetos en zonas de código clausurado ante cambios porque introducen dependencias. Encapsular el conocimiento y la creación de objetos concretos en métodos o clases que devuelvan objetos concretos detrás de referencias a tipos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

Ventajas:

- + Creación controlada de objetos
 - + Código desacoplado al evitar dependencias no deseadas
 - + Código más seguro

Desventajas:

- + Diseño más elaborado



Extensión de interfaz / objeto

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.

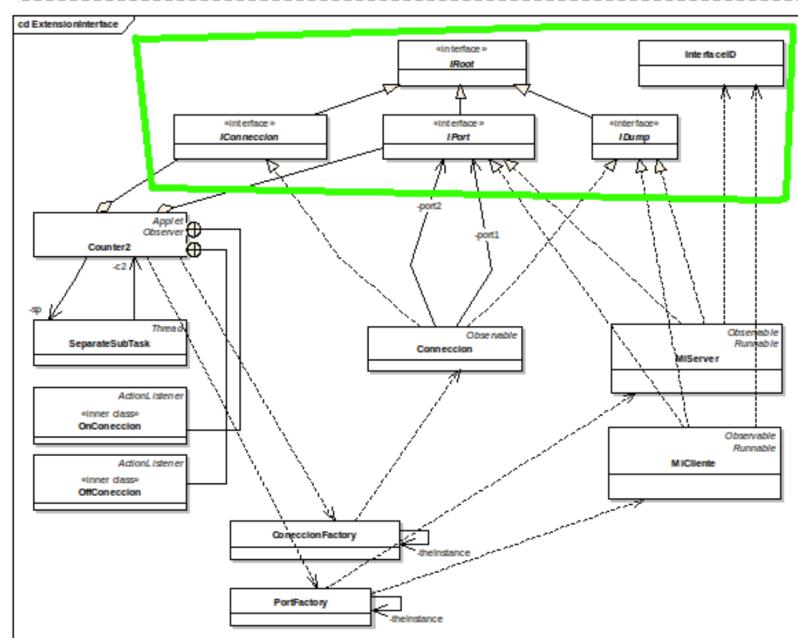
Cómo usarlo: El componente servidor será único para evitar contar con múltiples versiones. Las diferentes versiones del cliente deben ser capaces de funcionar con la única versión adaptada del servidor.

Ventajas:

- + Facilita el mantenimiento de componentes al simplificar la evolución del servidor.

Desventajas:

- + Diseño más elaborado



Facade

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.

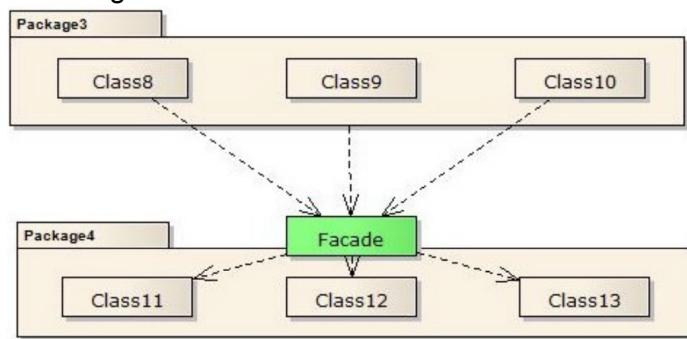
Cómo usarlo: Conjunto de clases que ordenan el acceso según las funcionalidades necesarias en el contexto del código cliente.

Ventajas:

- + Facilita el desarrollo
 - + La comprensión del código cliente
 - + Evita acoplamientos innecesarios
 - + Ordena el acceso a los subsistemas

Desventajas:

- + Diseño más elaborado



Bridge

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

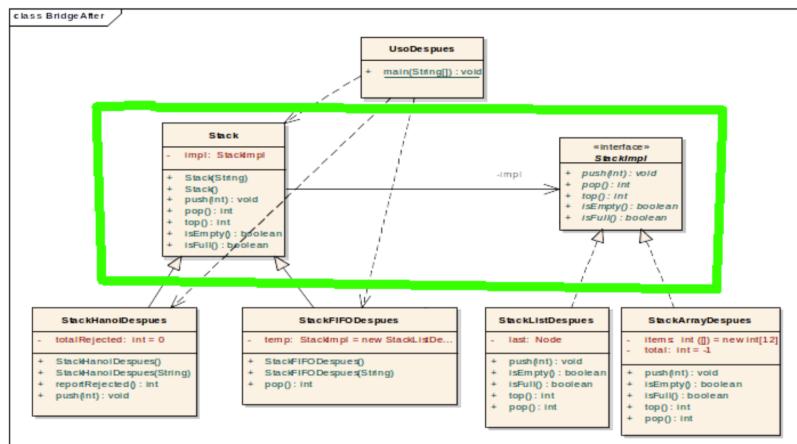
Cómo usarlo: Jerarquías de clases independientes para las abstracciones e implementaciones permiten la evolución por separado.

Ventajas:

- + facilita el desarrollo evolutivo de las abstracciones e implementaciones

Desventajas:

- **+ Diseño más elaborado**



Estado

Contexto y problemas presentes en el diseño / código: Cuando hay código con lógica compleja para administrar la transición de estados de un objeto.

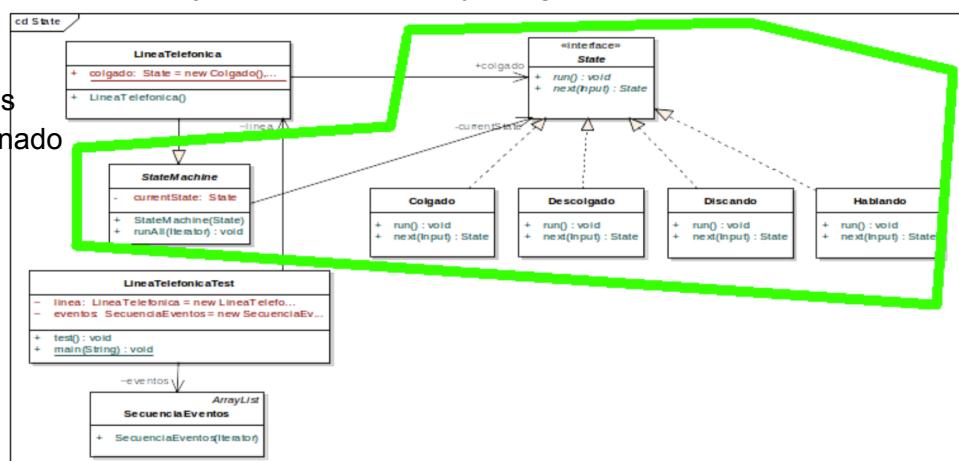
Cómo usarlo: Distribuir la funcionalidad entre los objetos de tipo Estado y la lógica de la transición en clase que funcione como máquina de estados.

Ventajas:

- + Aclara la lógica de las transiciones
 - + Permite un crecimiento más ordenado

Desventajas:

- Complica el diseño



Estrategia

Contexto y problemas presentes en el diseño / código: 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.

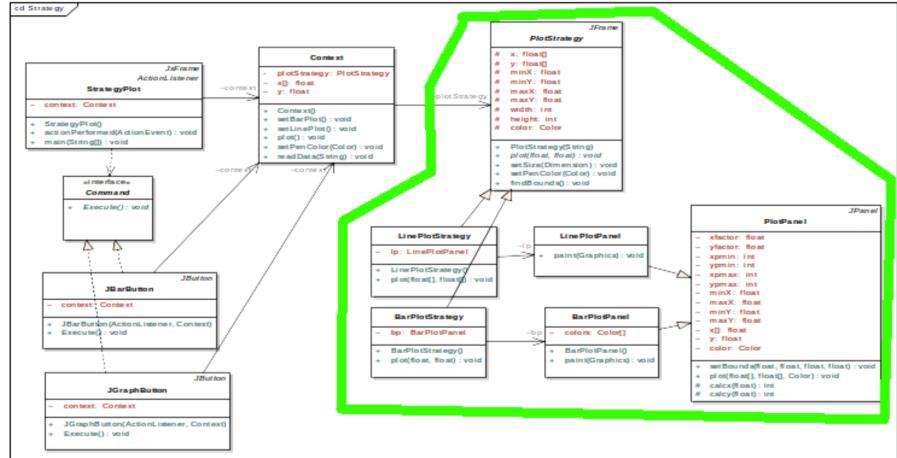
Cómo usarlo: Seleccionar la alternativa desde el contexto.

Ventajas:

- + 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
- + Pasaje de datos



Método template

Contexto y problemas presentes en el diseño / código: Cuando existe un algoritmo predefinido con uno o más pasos indefinidos.

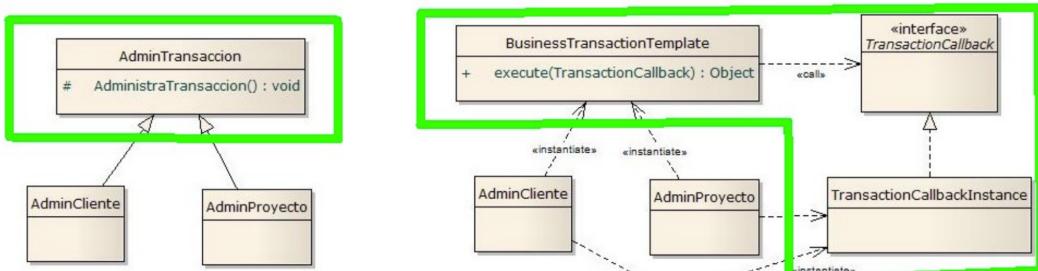
Cómo usarlo: Generalizar en una clase el método del algoritmo pre definido y dejar la definición de los pasos indefinidos para que los definan las subclases.

Ventajas:

- + Evitar duplicación de código
- + Documenta el algoritmo
- + Facilita la definición

Desventajas:

- + Complica el diseño



Decorador

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

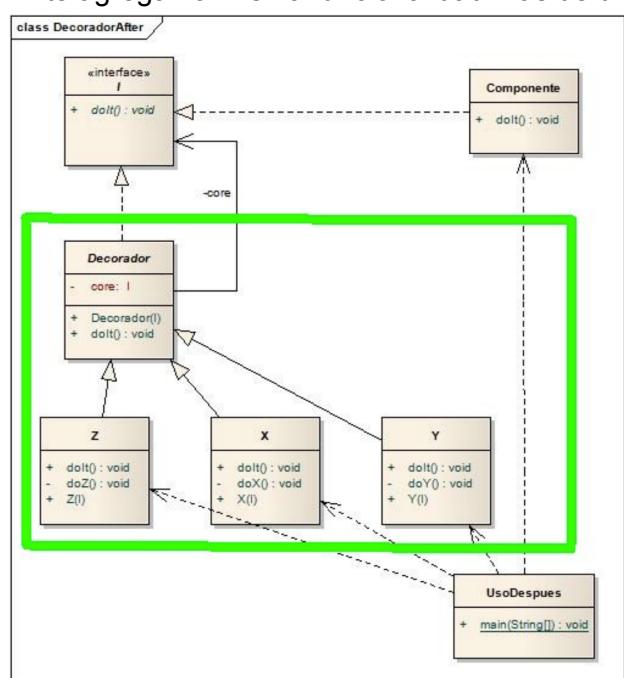
Cómo usarlo: Simplifica las clases y las hace más livianas. Permite agregar la misma funcionalidad más de una vez.

Ventajas:

- + Facilita el desarrollo mientras avanza el diseño

Desventajas:

- + Modifica la identidad de los objetos
- + Hace al código difícil de entender y debuguear



Visitor

Contexto y problemas presentes en el diseño / código: Cuando hay indefiniciones acerca de funcionalidades asignadas a determinadas clases. Cuando es necesario hacer acumulaciones para informar.

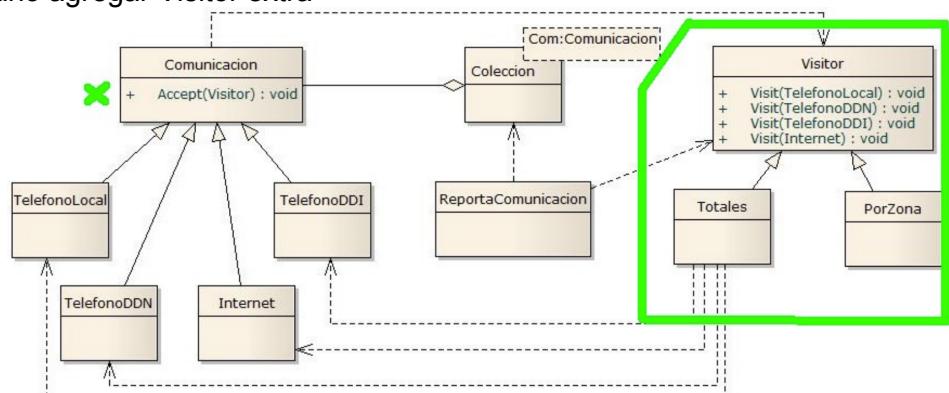
Cómo usarlo: Implementar algoritmos de acumulación sobre jerarquías de clases, visitar clases en una o más jerarquías, evitar type casting al recorrer clases mientras se totalizan índices

Ventajas:

- + Extender el servicio prestado por clases sin modificarlas

Desventajas:

- + Rompe con el encapsulamiento de las clases visitadas
- + Complica el diseño cuando es necesario agregar Visitor extra



Adapter

Contexto y problemas presentes en el diseño / código: Clases con funcionalidades útiles pero con interfaces diferentes. Cliente con protocolo ya definido es la interfaz a respetar

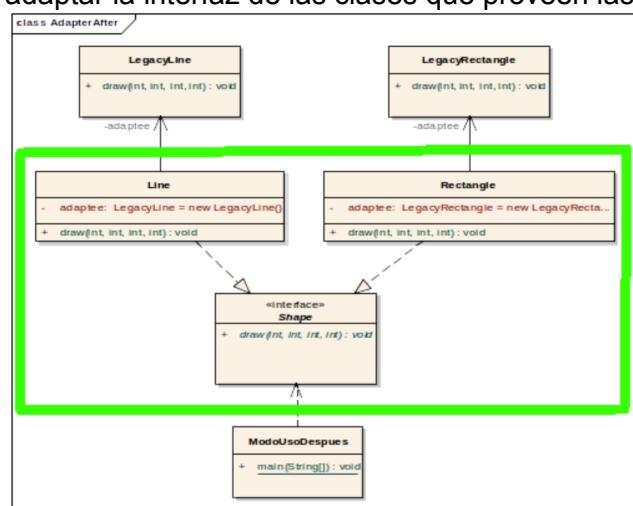
Cómo usarlo: Habilita al código cliente a utilizar funcionalidades de clases con diferentes interfaces adaptándolas. Simplifica y unifica el código cliente al adaptar la interfaz de las clases que proveen las diferentes funcionalidades.

Ventajas:

- + Promueve y ordena el reuso

Desventajas:

- + A veces es difícil adaptar las interfaces
- + Y complica el diseño



Composite

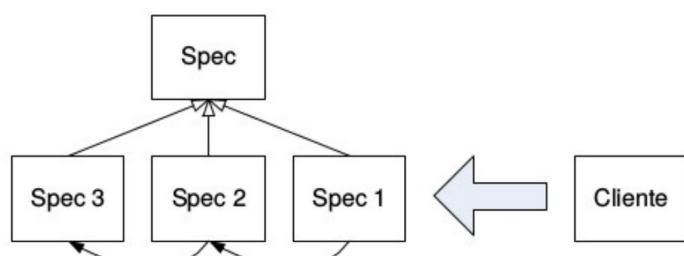
Cómo usarlo: Habilita al código cliente a utilizar funcionalidades de clases con diferentes interfaces adaptándolas. Simplifica y unifica el código cliente al adaptar la interfaz de las clases que proveen las diferentes funcionalidades.

Ventajas:

- + Promueve y ordena el reuso

Desventajas:

- + A veces es difícil adaptar las interfaces
- + Y complica el diseño



Métricas

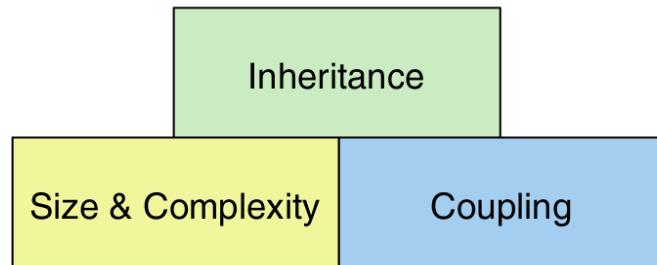


Fig. 3.1. The the three major structural aspects of a system quantified by the *Overview Pyramid*.

Inheritance (Herencia)

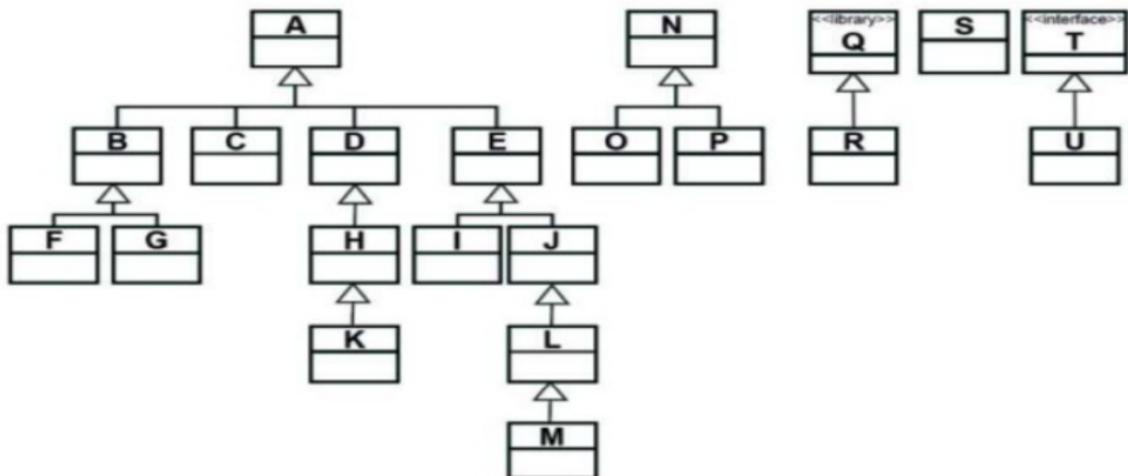
ANDC: Average Number of Derived Classes. Sumatoria(nro clases x nro herencias por clase). Solo se contabilizan clases (no interfaces) propias (no de librerías). Esta medida nos indica el ancho de la herencia del árbol de clases

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

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)

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



AHH: Average Hierarchy Height. Sumatoria(nro clases root x nro niveles de herencia)/NOC. Esta medida nos dice qué tan profundas son las jerarquías de clases. Un valor pequeño indica que la clase es plana

Size & Complexity:

NOP: nro de paquetes o namespaces en el sistema.

NOC: nro de clases en el sistema. Cantidad de clases definidas en el sistema sin contar las library classes.

NOM: nro de métodos x clase. Cuenta el número total de métodos llamados y la cantidad de llamadas únicas de funciones en la clase. La excesiva complejidad puede hacer que el código sea difícil de entender, con mala performance y riesgoso de cambiar.

LOC: nro de líneas de código x método. A mayor número de líneas, más complejo será el código. Si un file se vuelve grande es recomendable separarlo en files más pequeños.

CYCLO: nro de decisiones x líneas de código (Complejidad ciclomática). Computa la cantidad de caminos de ejecución distintos a través de un método o función, lo que también es la cantidad mínima de tests que necesitan alcanzar el 100% de coverage. Valores altos de esta complejidad ciclomática se correlacionan con métodos complejos y función o métodos difíciles de leer.

WMC: CYCLO / Métodos x LOC / Método x LOC / clase

AMW: CYCLO / Método (average method weight)

Coupling (Acoplamiento):

CALLS: nro de invocaciones a cualquier método de una clase desde diferentes métodos de otras. Esta métrica cuenta el número total de llamados a distintas operaciones del proyecto. Ex. si la operación foo() es llamada tres veces por la función f1(), va a contar como una sola llamada. En cambio si es llamada por f1(), f2() y f3() el valor de esta métrica será tres.

FANOUT: es el número de clases llamadas, es decir a cuantos métodos EXTERNOS llama una clase.

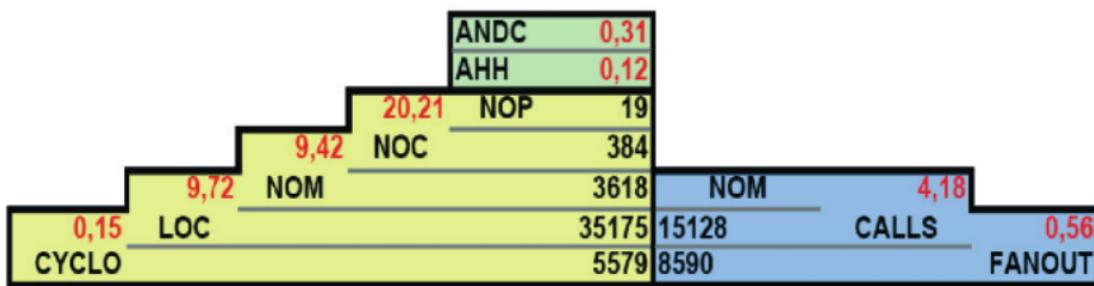


Fig. 3.5. A complete Overview Pyramid.

Ayuda memoria: $0.15 = 5579 / 35175$; $9.72 = 35175 / 3618$;

Computed properties

NOC / NOP: indica si los paquetes tienden a ser de granularidad gruesa o de granularidad fina.

NOM / NOC: indica la calidad del diseño porque revela cómo están distribuidas las operaciones entre clases.

Valores altos pueden indicar que están faltando clases

LOC / NOM: indica qué tan bien está distribuido el código entre las operaciones. Valores altos sugieren que las operaciones de sistema son bastante pesadas.

CYCLO / LOC: indica qué tanta complejidad condicional vamos a encontrar en las operaciones. Ex. 0.2 indica que una nueva rama (condición, if, else) se agrega cada 5 líneas.

CALLS / NOM: indica el nivel de acoplamiento entre operaciones. Valores altos sugieren que hay acoplamiento excesivo entre las operaciones.

FANOUT / CALLS: Esta proporción es un indicador de cuánto involucra el acoplamiento de muchas clases. Ex. 0.5 significa que de 2 llamadas a operaciones se involucra a otra clase.

Medidas de falta de armonía

1. **Identity Harmony** – “How do I define myself?” Every entity in a software system must justify its existence: does it implement a specific concept and how does it do that? Is it doing too many things or nothing at all?
2. **Collaboration Harmony** – “How do I interact with others?” Every entity collaborates with others to fulfill its tasks. Does it do that all on its own, or does it use other entities. How does it use them? Does it use too many?
3. **Classification Harmony** – “How do I define myself with respect to my ancestors and descendants?”. This harmony combines elements of both identity and collaboration harmony in the context of inheritance. For example, does a subclass use all the inherited services, or does it ignore some of them?

Medidas de falta de armonía

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

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.