Criterios SOLID

Son principios que cuando se utilizan en forma combinada, hacen más fácil el desarrollo y el mantenimiento del software. También colaboran en evitar code smells y la facilidad de refactoring en el código.

S. Single responsability -
O. Open closed - Código cancelado ante cambios y abierto a ser extendido
L. Liskov substitution - Principio de sustitución de Liskov (diseño por contrato)
I. Interface segregation - Segregación de interfaces
D. Dependency Inversion - Principio de inversión en cadena de dependencia

Single responsability

Una clase debe tener un único trabajo.

Ver ejemplo 1

Open Closed

Código cancelado antes cambios y abierto a ser extendido

El ejemplo típico es el de un método que tiene varios CASE dependiendo de qué clase esta tratando (calcular el Area de una figura) y si se tiene que agregar una figura nueva hay que modificar el método ese, lo cual no se desea hacer. Para arreglar esto lo que se hace es crear una clase abstracta Figura que oblige a implementar un método calcular área y que lo haga cada clase que implemente Figura, de forma tal que nuestro viejo método solo se tenga que asegurar que el objeto implementa Figura y utilice el método calcular area.

Liskov sustitution

Definición 1: Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento de un programa.

Definición 2: Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

Definición 3: si **S** es un subtipo de **T**, entonces los objetos de tipo **T** en un programa de computadora pueden ser sustituidos por objetos de tipo **S** (es decir, los objetos de tipo **S**

pueden sustituir objetos de tipo **T**), sin alterar ninguna de las propiedades deseables de ese programa (la corrección, la tarea que realiza, etc.)

Segregación de interfaces

Un cliente nunca debe ser obligado a implementar una interface que no usa.

Ejemplo 1:

Si tengo una Good Class que todo el mundo la implementa porque es re completa, muchos de esos clientes van a usar pedazitos nomas y lo demas sobra. Lo que hay que hacer es dividir la interfaz en muchas interfaces distintas y que cada cliente use las que necesita solamente..

Ejemplo 2: Link

Dependency Inversion

Las entidades deben depender de abstracciones no de cosas concretas. Los estados que están en un modulo de nivel superior no deben implementar cosas de un nivel menor, pero si deberían depender de abstracciones.

Generalidad es equivalente a abstracción
Dependencia es equivalente a inestable

Un software es estable cuando el esfuerzo por cambiarlo es bajo.

Hay 2 formulas que se aplican entre paquetes principalmente:

Inestabilidad = Referencias salientes / (Referencias salientes + Referencias entrantes)
Abstracción = Clases abstractas / (Clases abstractas + Clases concretas)

Notas sobre los patrones de diseño.

Patrones de creación:

- Abstract Factory: Plantea una clase abstracta Factory y clases concretas que heredan de ella para crear productos concretos.
- Factory method: Heredas del producto ó ayudado por un template method y definís un método que devuelva la instancia de la clase bien armada.
- Builder: Aca definís una clase abstracta Builder y le creas Builders concretos herendando de ella, cada builder concreto va a tener métodos que van a construir objetos complejos en general.
- Prototype: Creas una clase Prototype que “fabrica” un prototipo y la podes llamar para que te devuelva clones de cada producto. Todos los productos tienen que tener un método clone.
- Singleton

En Abstract Factory en general:

- las Concrete Factories implementan un Singleton.
- las Concrete Factories definen un factory method para cada producto.
- las Concrete Factories pueden ser implementadas usando un Prototype.

En Builder frecuentemente construye un Composite.

La principal diferencia entre Builder y Abstract Factory es que Builder se enfoca en construir un objeto complejo paso por paso, en cambio Abstract Factory pone foco en las familias de productos sin importar si son simples ó compuestos.

En Factory Method en general son llamados por Template Methods.

Diseños que usan un Composite ó Decorator pueden usar un Prototype.

Patrones estructurales

- Adapter: Crea una clase que hereda del objeto que se necesita y contiene al objeto que tiene otra interfaz que necesita ser adaptado.
- Decorator: Es casi un Composite pasa que tiene diferentes propósitos,en el Decorator se crea la clase abstracta Decorator y abajo de ella los ConcreteDecorators que son los que agregan la funcionalidad. Cuando se usa, los ConcreteDecorators se referencian como un Decorator.
- Proxy: Creas una clase que contenga al elemento del cual se quiere controlar el acceso y se puede hacer que herede de la misma interfaz que este también.
    - Cuando es un Virtual Proxy no tiene una referencia al objeto que se esta controlando el acceso.
- Fachada
- Compositive

- Bridge: Desacopla la abstracción de su implementación de forma tal que la abstracción pueda tener diferentes implementaciónes.

Patrones estructurales:

- Strategy
- State
- Chain of responsability
- Observer
- Visitor
- Command

El Proxy se usa para controlar acceso, a diferencia del Adapter que se usa para adaptar un objeto, esto también pasa en comparación con el Decorator que lo que hace es agregar responsabilidades a un objeto.

La diferencia entre el Fachada y el Mediator es que el primero no agrega ninguna funcionalidad en cambio el 2do si.

En la implementación del Strategy hay varias formas de pasarle parametros al método que se ejecuta en la estrategia:

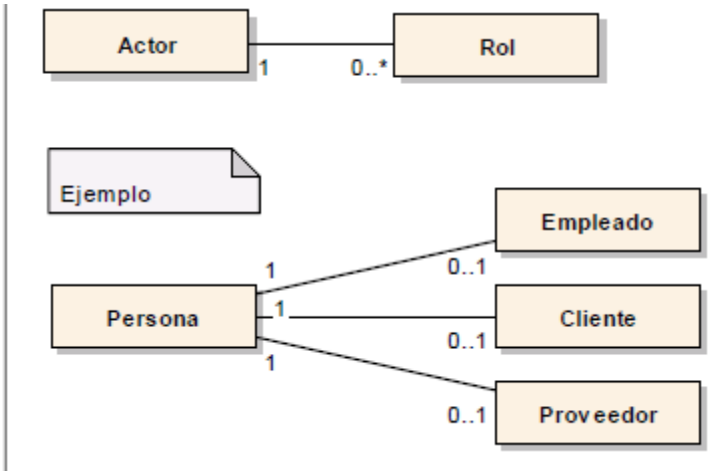
1. Pasarle el mismo Context
2. Pasarle otro objeto Context
3. Reutilizando con generics.

El Visitor usa un Iterator para recorrer los elementos visitables.

El Visitor puede ser utilizado junto al Compositive si se necesita que haya un elemento visitable que a su vez contiene otros elementos visitables también.

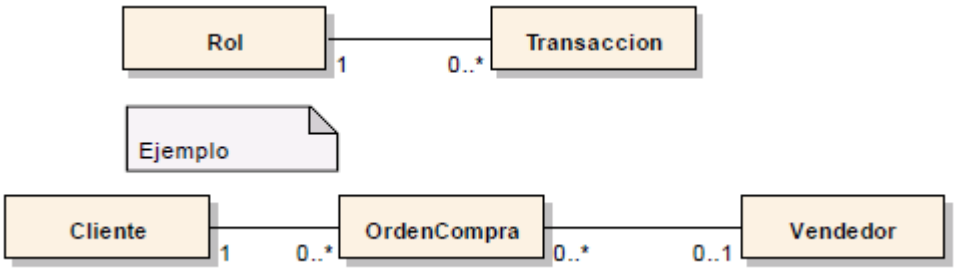
Patrones de análisis

1. Actor-Rol (A-R)

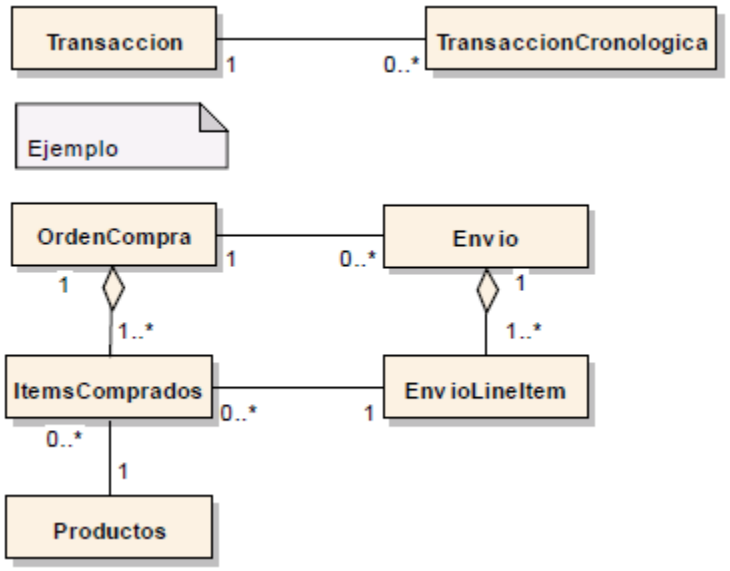


*Reglas de negocio: las tiene el **Rol***

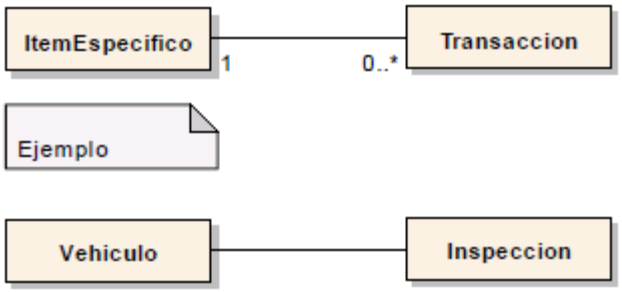
2. Rol-Transacción (R-T)



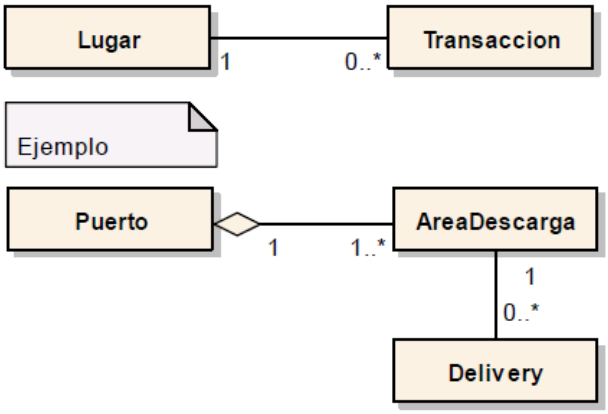
3. Transacción-Transacción Cronológica (T-TC)



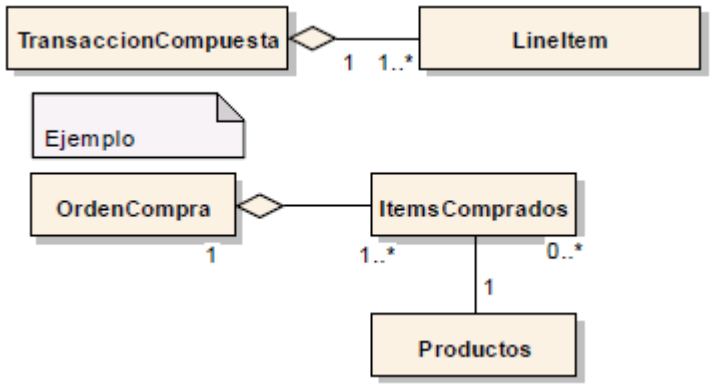
4. Item Especifico-Transacción (T-IE)



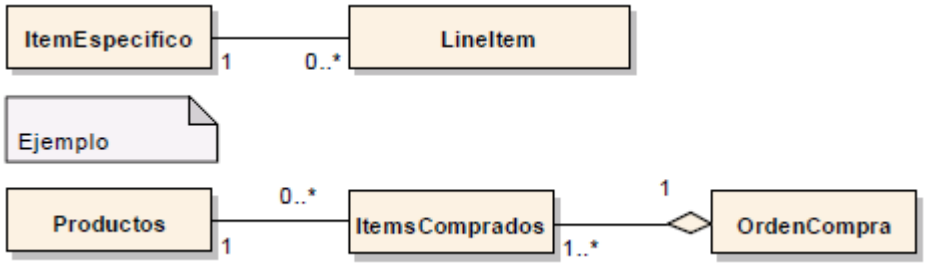
5. Transacción-Lugar (T-L)



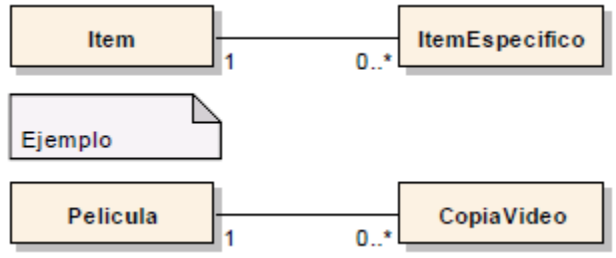
5. Transacción Compuesta-LineItem (T-LI)



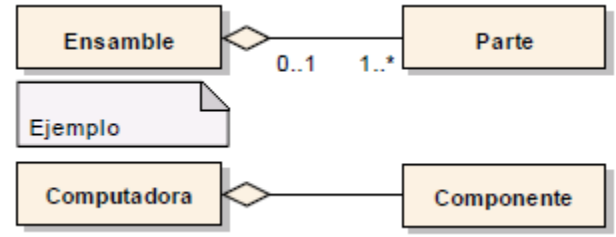
6. Item Especifico - Line Item (IE-LI)



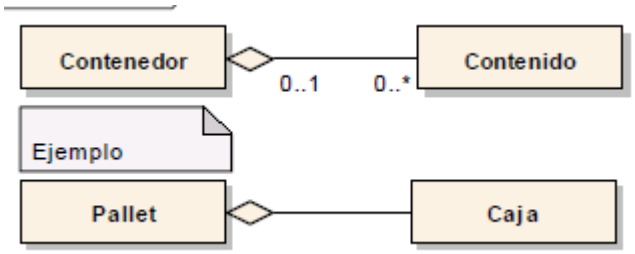
7. Item-Item Especifico (I-IE)



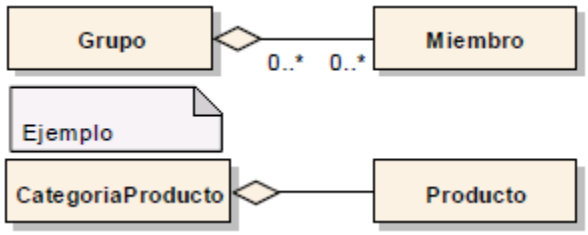
8. Ensamble-Parte (E-P)



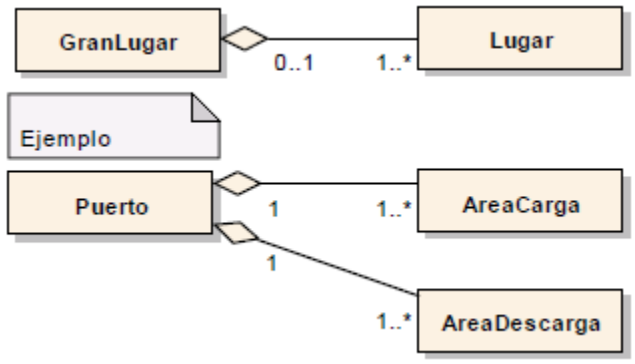
9. Contenedor-contenido (C-C)



10. Grupo-Miembro (G-M)

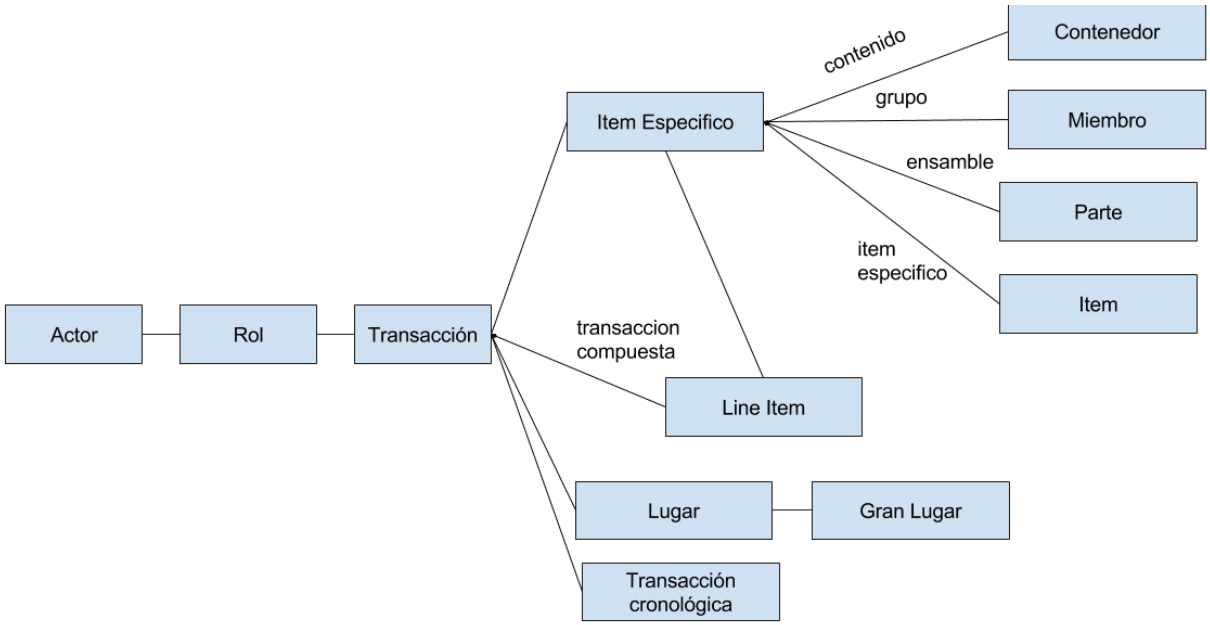


11. Gran lugar-Lugar (GL-L)



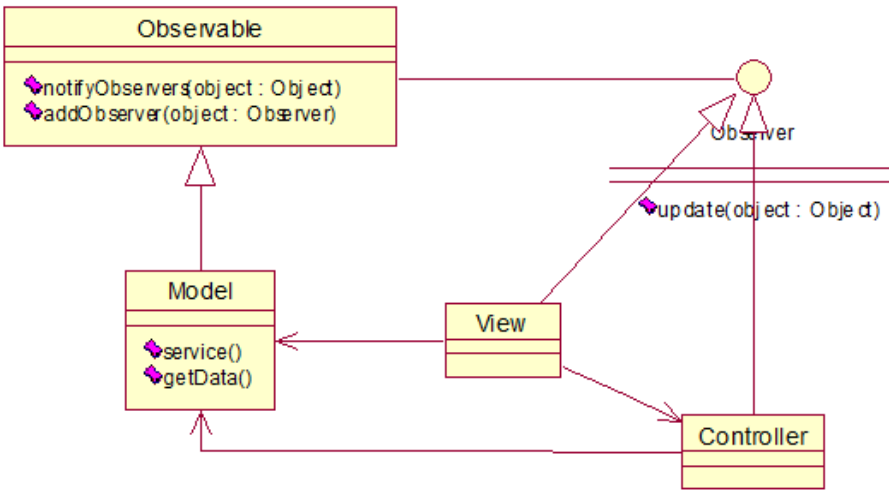
Reglas de negocio:

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


Actor
Rol
Transacción
Item Especifico
contenido
Contenedor
grupo
Miembro
ensamble
Parte
item
especifico
Item
transaccion
compuesta
Line Item
Lugar
Gran Lugar
Transacción
cronológica

Patrones de arquitectura

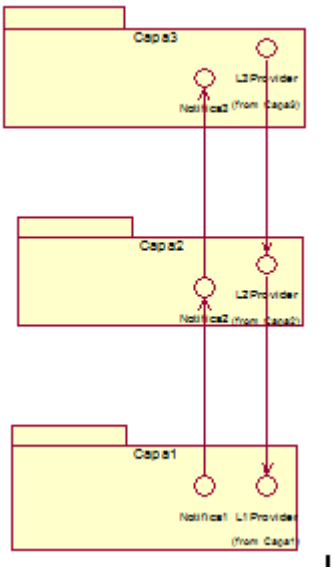
MVC



¿Cuando usarlo?

- Cuando hay usuarios múltiples que requieren que la información (modelo) sea visualizada de distintas formas. (hablo de usuarios que no “saben” de computación y requiere cada uno la información mostrada APB.

Layers



¿Cuando usarlo?

- Cuando es un sistema complejo que posee distintos niveles de abstracción, *si no podemos encontrar niveles de abstracción no usarlo.*

Problema a resolver:

1) Cada capa se comunica solo con su proveedor de servicios.
2) Cada capa informa del servicio que presta
3) Interfaces estables
4) Recurso de las distintas capas

Enterprise Architecture (también lo mencionó como Cliente Servidor)

Browser -> WebService -> Acceso a negocio -> Negocio ->Persistencia -> DB

Cada una de estas “cajas” puede tener un patron Layer adentro.

¿Cuando usarlo?

- 1er indicio: Cuando la aplicación que tenemos que construir es algo business, posee una logica de negocio, almacena información.
- 2do indicio: Cuando hay un cliente y un servidor, por ejemplo una aplicación mobile.
- 3er indicio: cuando hay reglas de negocio, concurrencia.

Contexto:
Sistemas
Multiusuarios
Concurrente
Muchas Vistas
Persistentes
Reglas de negocios

Microkernel

External Services --> Microkernel --> Internal Server
(aplicaciones) Core (drivers)

Sistema o desarrollo de sistemas que requieren adaptarse a distintas tecnologías y que se le pide un conjunto de funcionalidades para construir aplicaciones.

La idea de este patrón es separar la dependencia de la tecnología de las funcionalidades.

Sistemas van a tener larga vida.

¿Cuando usarlo?
Cuando se cumplen estas tres cosas:

- Cuando tenemos bien marcados quienes van a ser los External Services y quienes los Internal Server.
- Cuando los ES solamente pueden ser aplicaciones y en general mas o menos grandes no cualquier pavada.
- Cuando los IS son drivers relacionados con tecnología

Nota: El patron EA se podría transformar en un Microkernel siendo el WebService, Acceso a negocio un ES y la Persistencia y la BD un IS, quedando la capa de negocio como un MicroKernel.

Pipe & Filter

La idea de este patron es usar capas para particionar un string de datos y procesarlo de forma independiente.

¿Cuando usarlo?

- Cuando hay que procesar un string de datos ó similar
- Cuando existe la posibilidad de particionar el procesamiento de la info en procesos atómicos y reutilizables entre si. (como los pipe de linux)
- Conectar procesos atomicos entre si
- Cuando no es un sistema crítico

Broker

Cliente->Proceso <-> Broker <-> Proceso <- Server

El broker es como un mediator.
Distribuir en distintas plataformas objetos.

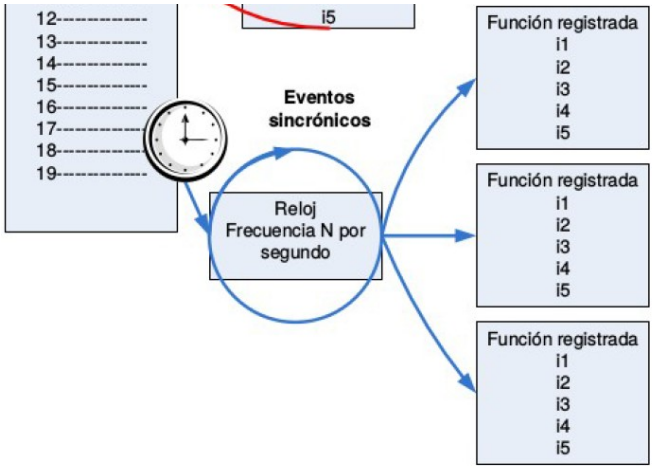
¿Cuando usarlo?

- Tiene que ser un sistema distribuido donde la programación del proceso es independiente de la comunicación y del lenguaje de programación.
- Cuando hay cambios de Server o de Cliente **runtime**.
- Cuando hay transporte de ubicación física de alguno de los componentes

Patrones de tiempo real (Contexto: muestrean eventos)

Pooling

Eventos periódicos



¿Cuando usarlo?

- Sistemas de tiempo real con eventos periódicos

Interrupción

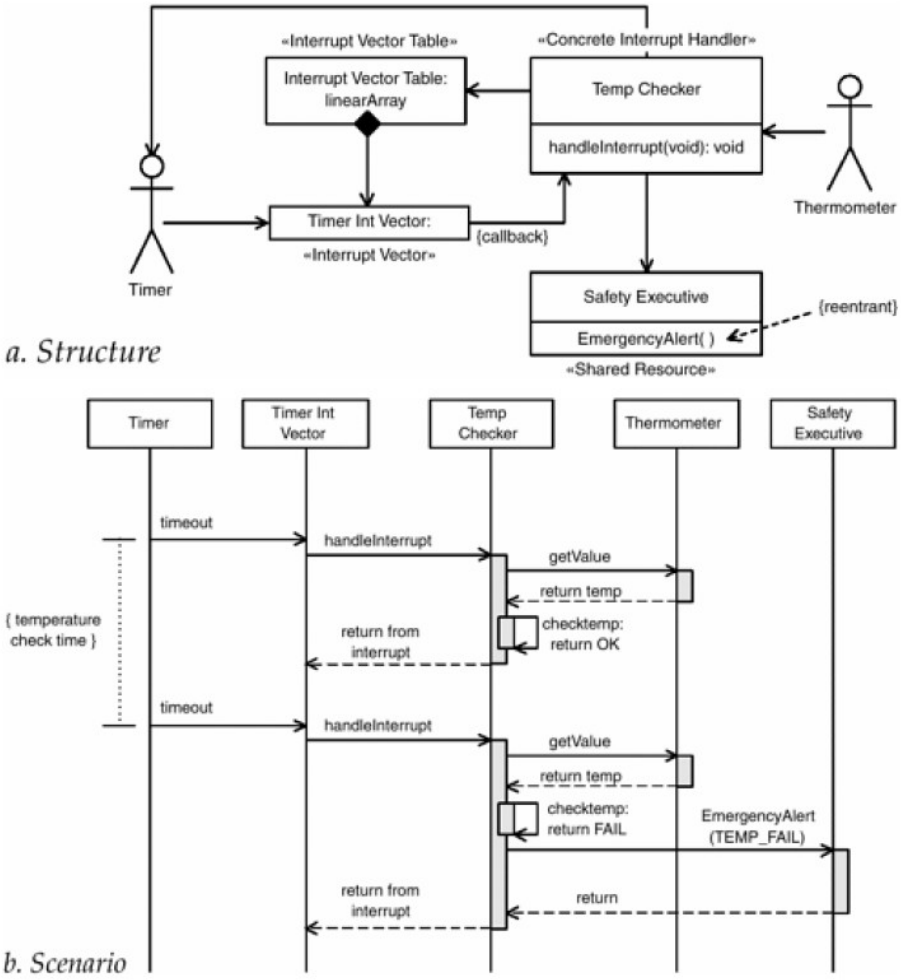
En sistemas de tiempo real, hay que responder a eventos en forma rapida y eficiente, sin importar lo que este ocurriendo o el sistema este haciendo en ese momento.

¿Cuando usarlo?

Cuando son sistemas de tiempo real que hay que responder a eventos asincronicos en forma rapida y eficiente y siempre y cuando estos eventos seas cortos, atomicos y no-interrumpibles.

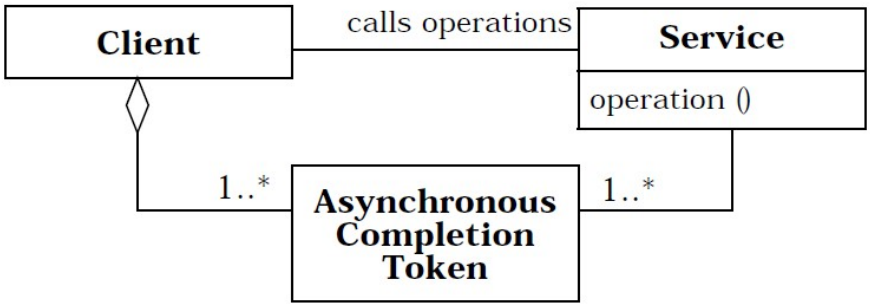
Ejemplo:

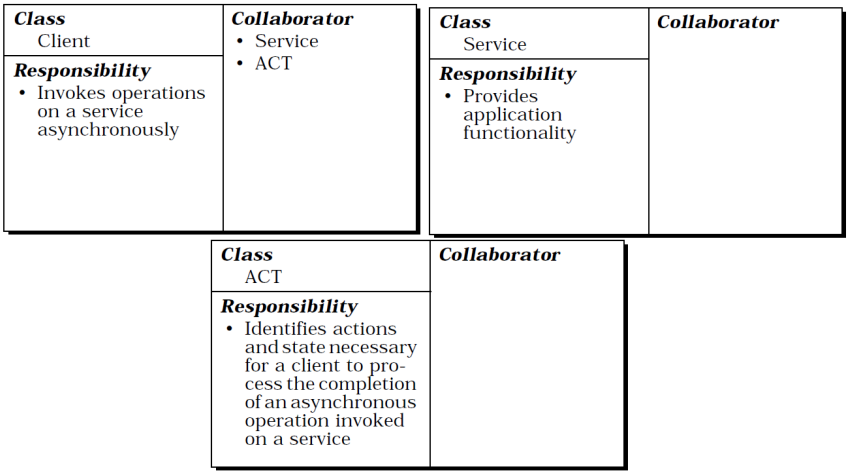
Cada 500 ms la temperatura de un reactor debe ser chequeada. Si la temperatura es menor o igual que un valor predeterinado esta bien, caso contrario tiene que alertar.

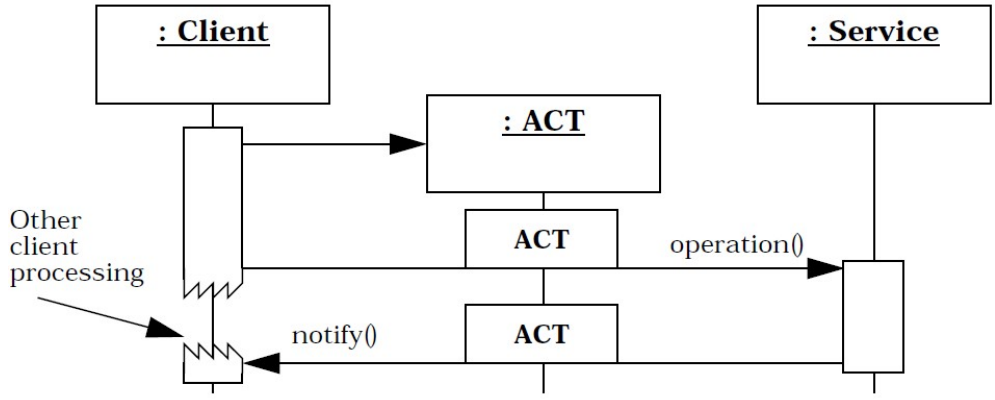


ACT (Asynchronous Completion Token)

Resuelve el problema de realizar peticiones (Cliente) de forma asincronica a un Servidor y recibir respuesta de esas peticiones para que el Cliente procese.







¿Cuando usarlo?

- Cuando tenemos un Cliente que tiene que realizar peticiones a un Servidor de forma asincronica y hacer algo con la respuesta a la peticion. Puede haber muchos servidores distintos de donde recolectar la información.

  Ejemplo:

Una aplicación que se encarga de administrar la información de un paciente de varias máquinas de la habitación. La aplicación debe primero invocar de forma sincrónica con todos los dispositivos de la sala para que estos notifiquen que están prendidos con una respuesta.

Ventajas:

- Simplifica la estructura de datos del cliente porque no necesita mantener complejas estructuras asociadas a cada servidor ya que el ACT tiene todo lo necesario.

Active Object

Desacopla la ejecución del método de la invocación y simplifica el acceso a objetos comunes entre los threads.

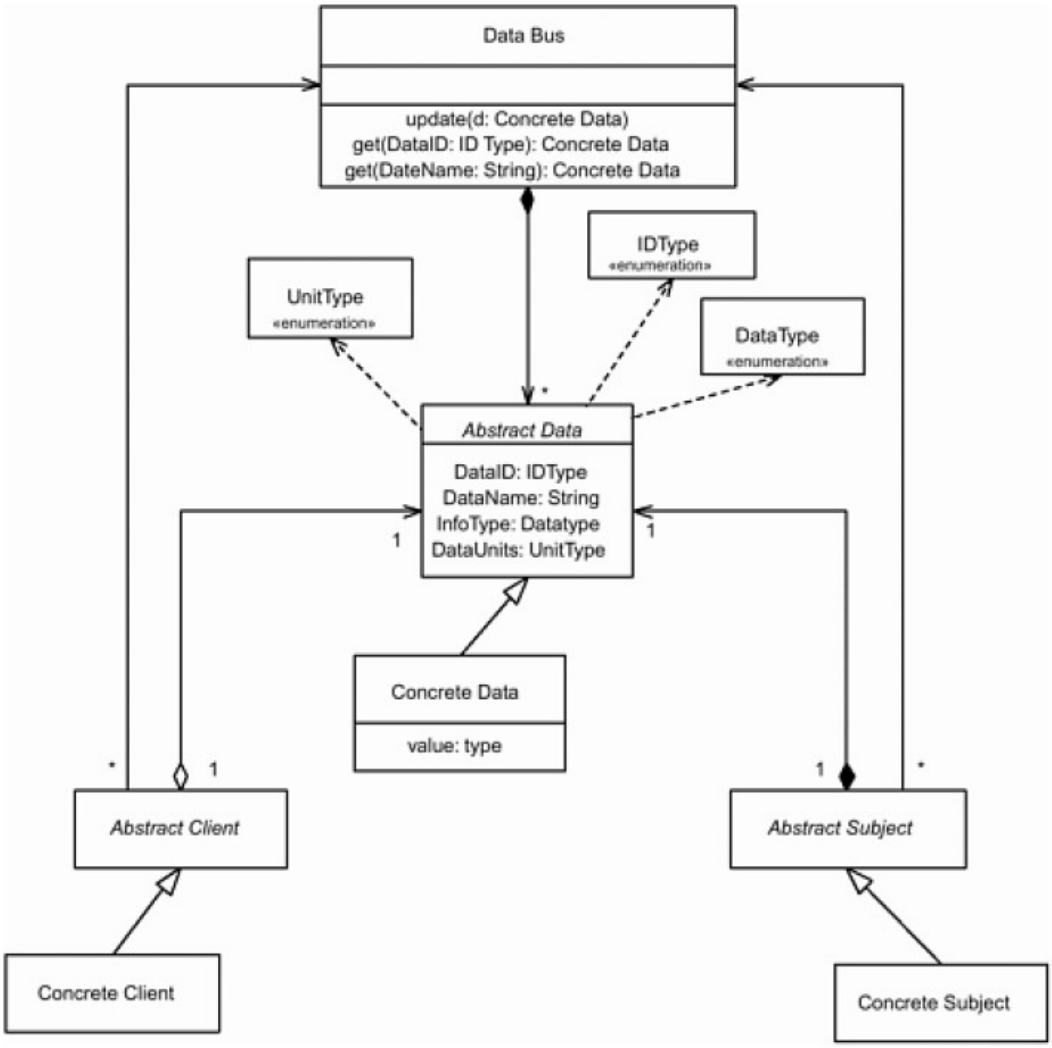
Básicamente el patrón Active Object es un objeto contiene una cola de mensajes (que esta en el hilo principal) y se ejecuta en un único thread propio, se la pasa revisando si hay algo en la cola de mensajes para procesar. Cuando el cliente quiere que el Active Object ejecute una acción lo único que hace es agregar un mensaje a esa cola.

¿Cuando usarlo?

- Cuando tenemos que secuenciar tareas en un ambiente multitarea
- Cuando varios threads de control requieren acceso sincronizado a datos compartidos.
- Cuando la información puede llegar en distinto orden en el que se debe ejecutar los procesos.

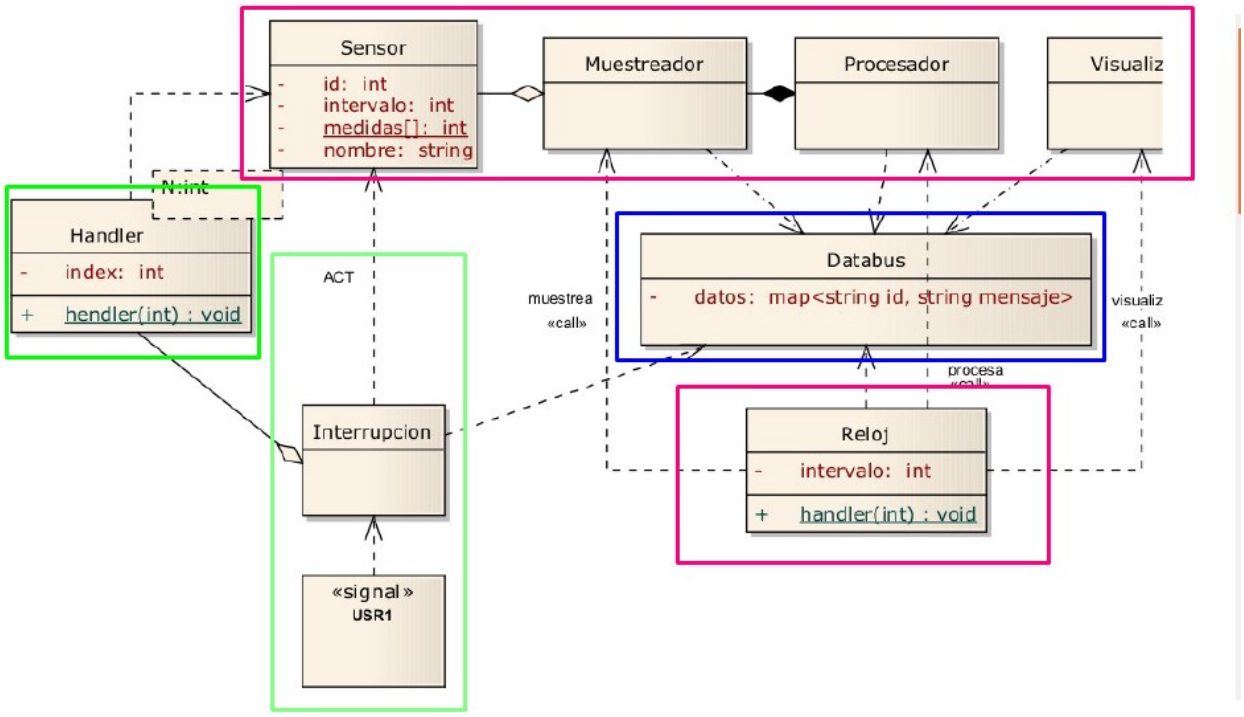
DataBus

Proporciona un bus común al que varios servidores pueden publicar su información y donde varios clientes consumen diversos eventos y datos publicados en el bus.


Data Bus
update(d: Concrete Data)
get(DataID: ID Type): Concrete Data
get(DateName: String): Concrete Data
IDType
«enumeration»
UnitType
«enumeration»
DataType
«enumeration»
Abstract Data
DataID: IDType
DataName: String
InfoType: Datatype
DataUnits: UnitType
Concrete Data
value: type
Abstract Client
Abstract Subject
Concrete Client
Concrete Subject

Todo junto

Ejemplo de Pooling + Interrupción + DataBus + ACT



Rosa = Pooling

El pooling es el que cada X tiempo marcado por el Reloj llama a los objetos Muestreador, Procesador y Visualizador. Estos 3 no son threads, sino el Reloj se encarga de ejecutarlo en forma sincrónica.

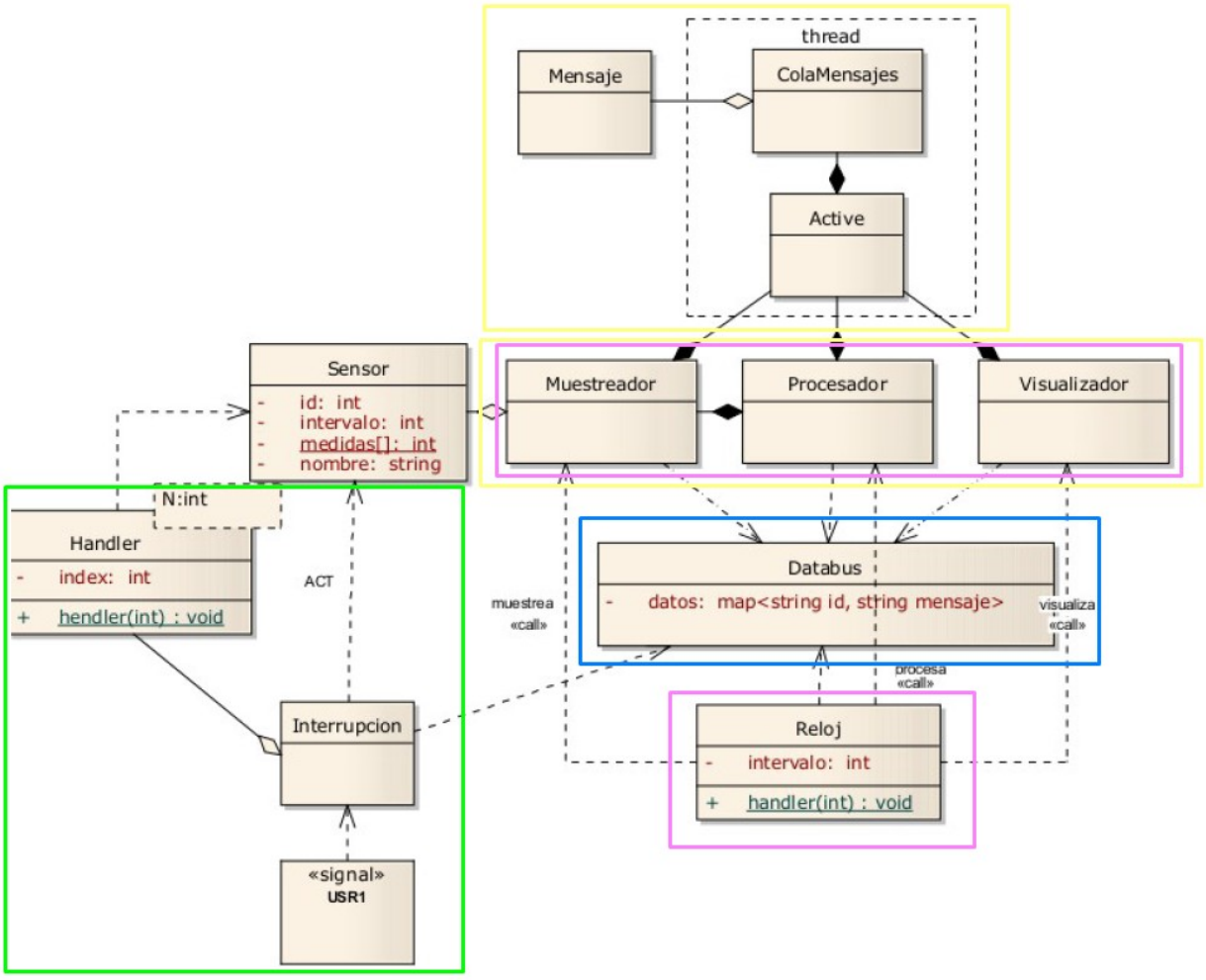
Verde = Interrupt + ACT

Viene la señal de afuera entonces la Interrupción se crea y contiene a un Handler con un identificador único, la interrupción llama al Sensor para que mida pasandole el ACT para que luego lo complete y deja un mensaje en el DataBus para que lo lean los demas.

Azul = Data Bus

Los procesos van dejando mensajes para que los lean otros procesos.

Ejemplo de Active Object + Pooling + Data Bus + Interrupt



La idea es que el Active Object contiene una cola de mensajes, cuando tiene un mensaje que ejecutar entonces va a crear los threads para ejecutar el muestreador, procesador y visualizador, y cada uno de estos va a tener el Active Object para comunicarse entre si con el objetivo de resolver el mensaje que se está procesando.

Metricas

CYCLO: complejidad de un programa.
LOC ó SLOC: Lineas de código, se puede usar para predecir el esfuerzo requerido para desarrollar ó mantener un programa.
NOM: Cantidad de métodos
NOC: Cantidad de clases
CALLS: Cantidad de llamadas
FANOUT: La cantidad de clases en las que una clase “confia” cosas.

ANDC: Promedio entre el número de clases con subclasses y el total de clases. (no cuentan las interfaces). Una clase que no tiene clases que derivan de ella (subclases) entonces tiene ANDC=0.
AHH: Promedio entre el máximo tamaño de subclases de las clase raiz y el total de clases. Una clase raíz es una clase que no deriva de ninguna otra (sin contar interfaces)

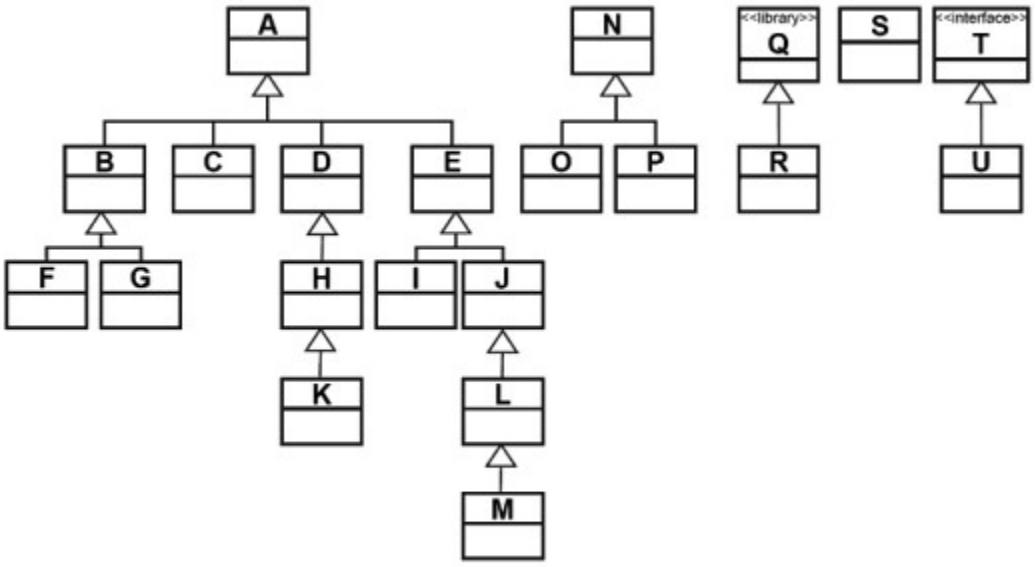


Fig. 3.4. Example to illustrate the computation of inheritance metrics used in the *Overview Pyramid*

1. ANDC — **Average Number of Derived Classes**, i.e., the average number of direct subclasses of a class. All classes defined within the measured system (and only those) are considered. Interfaces (in Java or C#) are not counted. If a class has no derived classes, then the class participates with a value of 0 to ANDC . The metric is a first sign of how extensively abstractions are refined by means of inheritance. We illustrate how ANDC is computed based on the example presented in Fig. 3.4. First, we count the classes: there are 19 classes, as we do not count *Q* because it is a library class; we also do not count *T* because it is an interface. Out of the 19 classes, 11 have no subclasses at all, four classes (i.e., *D*, *H*, *J*, *L*) each have one direct subclass, three classes (i.e., *B*, *E*, *N*) have 2 direct subclasses each, and there is one class (i.e., *A*) with four direct descendants. Thus, ANDC is computed as:

$$ANDC = \frac{11 \cdot 0 + 4 \cdot 1 + 3 \cdot 2 + 1 \cdot 4}{19} = 0.73$$

2. **AHH — Average Hierarchy Height.** The metric is computed as an average of the *Height of the Inheritance Tree* (HIT) among the *root classes* defined within the system. AHH is the average of the maximum path length from a root to its deepest subclasses. A class is a *root* if it is not derived from another class belonging to the analyzed system. Interfaces (in Java or C#) are not counted. Standalone classes (i.e., classes with no base class in the system and no descendants) are considered root classes with a HIT value of 0. The number tells us how deep the class hierarchies are. Low numbers suggests a *flat class hierarchy* structure. In order to illustrate how AHH is computed we revisit the sample system depicted in Fig. 3.4. First, we have to count the *root classes*. Based on the specification of AHH , we identify five root classes: *A*, *N*, *R* (because it is derived from a library class), *S* and *U* (because the implementation of an interface does not make *U* a subclass). For these root classes the values for the HIT metric are: HIT(*A*) = 4, HIT(*N*) = 1, HIT(*R*) = 0, HIT(*S*) = 0, HIT(*U*) = 0. Thus, AHH is computed as:

$$AHH = \frac{4 + 1 + 0 + 0 + 0}{5} = 1$$

Ejercicios:

1.¿Qué métricas utilizaría para caracterizar acoplamiento?

Seleccione una o más de una:
a. NOM
b. CALLS
c. LOC
d. FANOUT
e. WCM
f. ATFD
g. TCC
h. NOA

RTA: a - b - d

2. NOM puede servir para caracterizar "Tamaño".

RESPUESTA CORRECTA: Verdadero

3.

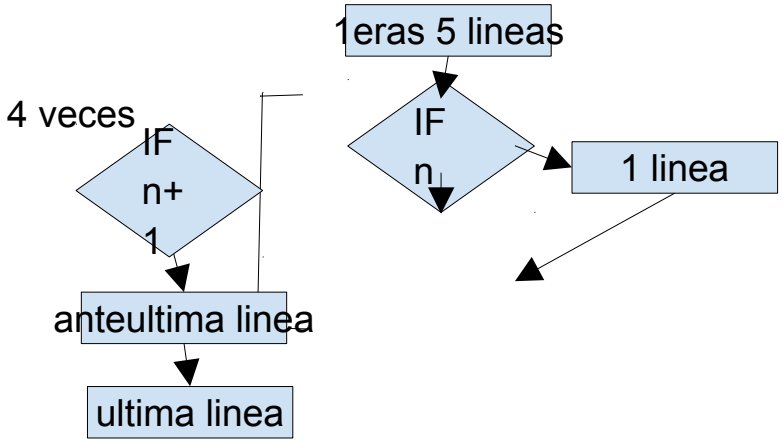
Ejercicio nro 1

Analizar el código del método "long getDirection()" que calcula la dirección a seguir por un navío a partir de las coordenadas actuales (latitud – longitud) y las del punto destino. Determine valores para una métrica que muestre el grado de las falencias presentes. Sugiera, a continuación, cambios a dicho código y muestre cómo los cambios se reflejan como mejoras en la métrica anterior.

```
287
288     public long getDirection(){
289
290         long direction = 0;
291         long latP = previousPosition.getLat();
292         long lonP = previousPosition.getLon();
293         long latC = currentPosition.getLat();
294         long lonC = currentPosition.getLon();
295
296         if(latC>latP && lonP==lonC) direction = MathFP.toFP("0");
297         if(latC>latP && lonC>lonP) direction = MathFP.toFP("45");
298         if(latP==latC && lonC>lonP) direction = MathFP.toFP("90");
299         if(latC<latP && lonC>lonP) direction = MathFP.toFP("135");
300         if(latC<latP && lonP==lonC) direction = MathFP.toFP("180");
301         if(latC<latP && lonC<lonP) direction = MathFP.toFP("225");
302         if(latP==latC && lonC<lonP) direction = MathFP.toFP("270");
303         if(latC>latP && lonC<lonP) direction = MathFP.toFP("315");
304
305         direction = Coordinates.degreesToRadians(direction);
306         return direction;
307     }
```

CYCLO=9
LOC = 15
9/15 = 0,6

FANOUT=2
CALLS=10
2/10=0,2



Cyclomatic complexity = E - N + P
E = number of edges in the flow graph.
N = number of nodes in the flow graph.
P = number of nodes that have exit points

Forma alternativa: The cyclomatic complexity is calculated by adding 1 to the following:

- Number of branches (such as **if**, **while**, and **do**)
- Number of **case** statements in a **switch**

CYCLO = 27 - 27 + 8*2 = 16 ó con el otro calculo sería 17 porque cada if cuenta como 2 ifs.
LOC=15

CYCLO / LOC > 1

Tiene alta esta medida que tiene que ver con el tamaño y la complejidad, ademas el LOC se podria ya por si solo considerar alto.

CALLS=13
FANOUT=3
FANOUT / CLASS = 3 / 13 = 0,23

Este esta OK y tiene que ver con el acoplamiento.