UML

Diagrama Secuencia
- Orden peticiones entre objetos
- Recuadro superior (obj), --- (vida), rectangulo (activacion)

Sincrónico
- Espera mensaje para continuar

Asincrónico
- Continúa ejec. sin mensaje respuesta

Diagrama de estados
- Estados posibles de obj. y transiciones entre ellos
- Cuadros (estados), flechas (transicion), circulos (ini y fin)

Estado compuesto
- Estado que dentro tiene diagrama de estados (Ej: marchas individuales de caja auto)

Estado histórico
- Recuperar estado anterior de conj. de estados (Ej: volver entre marchas individuales)

Diagrama de actividades
- Visualizacion simplificada de operacion o proceso (Ej: actividades posibles de un doc ABM)

Actividades en Paralelo
- Maquina de café: deposito agua, moler café, colocar vaso

Marco responsabilidad
- Segmento de diag. de actividad con un responsable y sus actividades asociadas

Capas y MVC

Arquitectura de software
- Organización fundamental de un sistema y sus componentes
- Relaciones componentes
- Ambiente y principios que orientan su diseño y evolución

Qué es Arquitectura
- Estructura gral. del software alto nivel
- Modo ofrecer integridad conceptual al sistema
- Estilo o combinación para una solución
- Centra requerimientos no funcionales
- Esencial proyecto

Qué no es
- Error (R) con Modelado Orientado a objetos

- Únicamente diseñada y documentada con UML
- Academia (Modelos industria probar, aceptar o refutar) VS Industria (implementa soluciones representando casos de buen uso o mal uso luego)

Estilos Arquitectonicos
- Patrones posibles aplicaciones
- Evaluar ventajas y desventajas conocidas de conjuntos reqs. no funcionales
- Sintetizar estructura soluciones

Capas Normal
- Define niveles y cada interno se aproxima mas a instrucs. maquina

Capas Puro
- Solo comunicacion con capas vecinas (**Portabilidad**)

Modelo 3 capas
- Presentacion (GUI, Input, Validacion) - HTML
- Negocio (Modelo, obtencion datos, recepcion presentacion) - Backend
- Datos (almacenamiento y recuperacion) - BD (sql, nosql)

MVC
- Modelo (Representacion datos, Lógica negocio, Mecanismo de persistencia)
- Vista (GUI, Presentacion Modelo, interaccion usuario)
- Controlador (responde a eventos, peticiones modelo sí recibe pedidos de informacion, medio entre vista y modelo)

MVC pasivo
- No notifica cambios en el modelo
- No hay dependencia modelo y vista

MVC activo
- Notifica cambios a la vista el modelo
- Hay dependencia

Diferencia 3 Capas y MVC
1) Capas no define componentes, y MVC obliga uso componentes implementados como clases
2) Capas puede lenguajes No POO, y MVC sólo para lo lenguajes POO
3) Capas eventos usuario x capa presentacion, MVC primero por el controlador
4) Capas BD acoplada a capa datos, MVC modelo desacoplado BD
5) Capas presentacion comunicacion directa con capa logica (negocio), MVC vistas no comunican directamente con el modelo (intermedio controlador)
6) Capas tiene 3, MVC tiene N capas
7) Capas presentacion única, MVC reutilizacion de vistas para componer otras vistas

Patrón de diseño
- Describe problema conocido dando solución en términos de objetos.

Composición Patrón
- Nombre (+ soluciones y consecuencias)

- Problema (aplicabilidad, explica problema y contexto)
- Solución (plantilla aplicarse a cualquier situación)
- Consecuencias (alternativas de diseño patrón, costos y beneficios)

Clasificación Patrones

- Propósito (qué hace)
    - Creación (de objetos)
    - Estructural (composicion clases u obj)
    - De comportamiento (interaccion obj. y responsabilidades)
- Ámbito (dónde aplico)
    - Clase (relaciones de herencia estatica en tiempo compilacion)
    - Objeto (relaciones dinamicas en tiempo ejecucion)

POO

Diferencia Interfaz y Clase Abstracta

- Estado interno
- Interfaz no tiene métodos definidos, abstracta puede tener comportamiento por default o no.
- Interfaz modela comportamiento, abstracta define futuros objs. con pequeñas diferencias en ciertos métodos.

Diferencia asociacion, composicion y agregacion

- Asociación: existe un atributo de una clase que es de un tipo de otra clase
- Composición: relación fuerte (ciclo de vida de objeto esta relacionado), tiene método de destruccion objeto.
- Agregación: relación débil (ciclo de vida no condicionado por destruccion de otro), existe un atributo en gral. coleccion, metodo agregar a colecc.

Delegación

- Sustitución a herencia
- Caracteriza por reutilización selectiva
- En lugar de heredar todo, se llama delegando cierta funcionalidad a otra clase desde un metodo propio.

Patrones

Observer

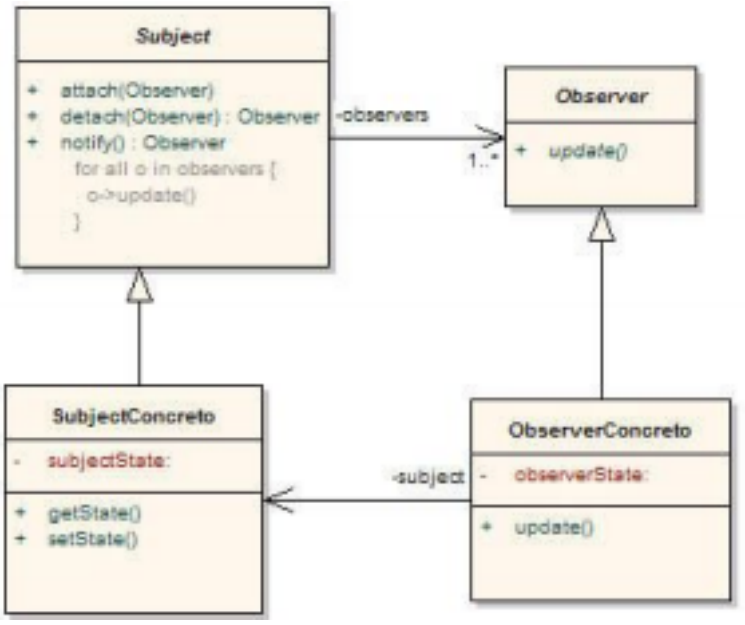
Propósito

- Dependencia 1 a muchos.
- Sí cambia el 1 notifica a los demás
- Al notificarse se actualiza c/u

Motivación

- Reutilización independiente de clases que definen los datos y las representaciones

Estructura



- Sujeto: add/ delete implementadores observer. N observadores. Clase Abs.
- Observer: Interfaz update objs. que deben notificarse
- SujetoConcreto: estado de interés. Hereda Sujeto
- Obs Concreto: referencia a Suj Concreto. Estado consistente con sujetoconcreto

Aplicabilidad

- Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse.
- Notificar N objetos sin conocerlos para evitar acoplamiento.

Ventajas

● Bajo acoplamiento Sujeto y Observador. (solo se conoce la interfaz de observer y se llama actualizar para c/u)
● Capacidad de comunicacion mediante difusion (no se conoce receptores, solo se notifica a los que estan escuchando al sujeto)

Desventaja
● Sin historial de cambios (no se sabe que cambió, hay que comparar la info vieja con la nueva)

Ejemplo
● Velocímetro digital y Analógico de un auto

Diferencia implementación pull y push
● ***Modelo push***: el sujeto envía a los observadores información detallada acerca del cambio, independientemente si se la solicitan o no.
● ***Modelo pull***: el sujeto no envía nada más que la notificación mínima, y los observadores piden después los detalles explícitamente.

Decorator

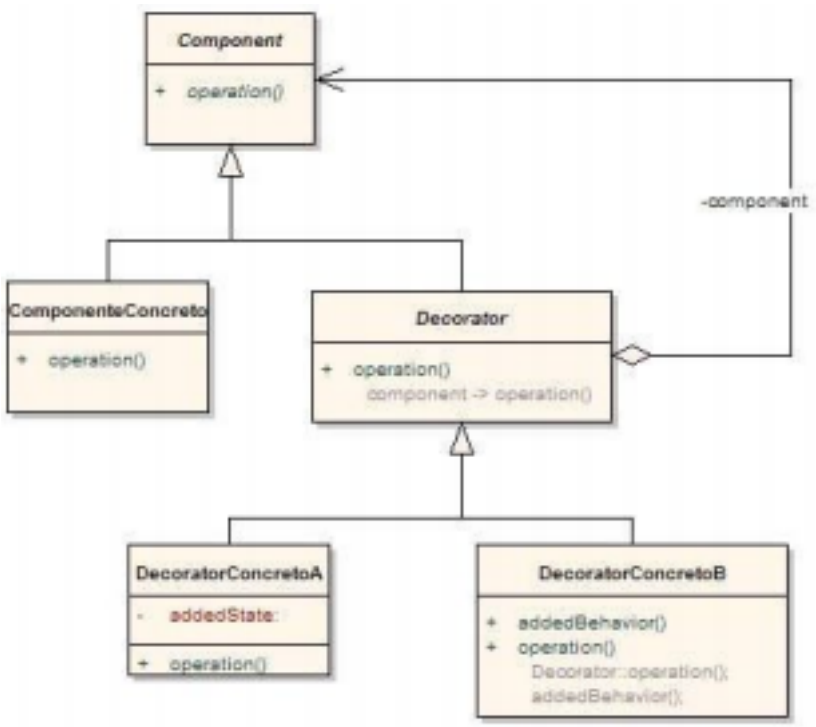
Propósito

- Asigna responsabilidades adicionales a un objeto dinámicamente
- Alternativa flexible a la herencia para extender funcionalidad.

Motivación

- Añadir responsabilidades a obj. individual en vez de a una clase
- Herencia no apropiada porque decora toda instancia igual
- Transparencia del decorador implementando interfaz componente

Estructura



- Componente: Define la interfaz para objetos a los que se puede añadir responsabilidades dinámicamente.
- ComponenteConcreto: Define un objeto al que se pueden añadir responsabilidades adicionales.
- Decorador: Mantiene una referencia a un objeto componente y define una interfaz que se ajusta a la interfaz del componente.
- DecoradorConcreto: Añade responsabilidades al componente

Aplicabilidad

- Añadir responsabilidad dinamicamente sin afectar el objeto
- Estas pueden ser quitadas
- Herencia inviable

Ventajas

- Más Flexibilidad que la herencia estática (agregar responsabilidad dinamicamente o quitar, mediante composicion)
- Evita clases cargadas de funciones en la parte de arriba de la jerarquía. No tener

una super clase adaptable a toda situación, sino componer con objetos decorador simples.

Desventajas

- Muchos objetos pequeños (difícil de mantener)

Ejemplo

- En un seguro agregar addons a una tarifa base, para obtener cobertura adicional

Singleton

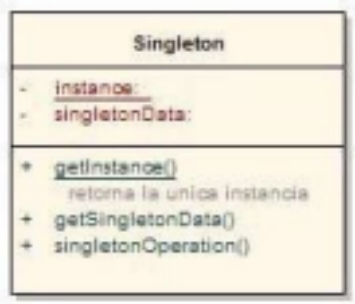
Propósito

- Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

Motivación

- Clase requiere una instancia única y se garantiza eso.
- Responsabilidad de la clase de retornar la instancia unica.

Estructura



Aplicabilidad

- Punto de acceso global a una única instancia de una clase

Ventajas

- Acceso controlado a la única instancia
- Espacio de nombres reducido <> variable global
- Es fácil volver a permitir más instancias ya que sólo se modifica una clase

Desventajas

- No thread safe.

Implementación

- Constructor privado
- Método instancia estático que devuelve el atributo interno instancia ● Se usa lazy load para inicializar cuando se demanda la primera vez

Ejemplos

- Log
- Cola impresión

● Clase configuración

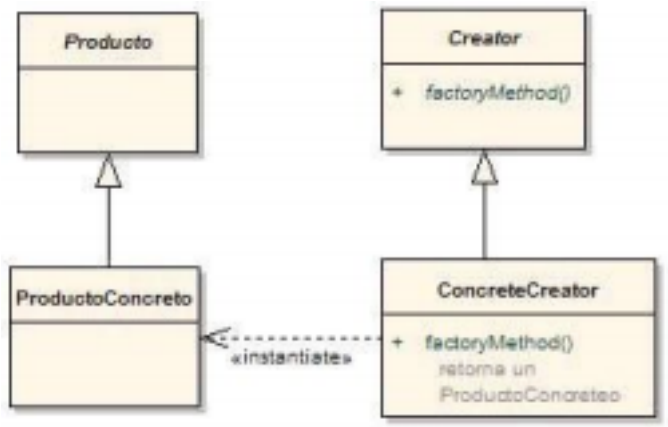
ThreadSafe
● Sí varios hilos concurrentes piden la instancia de la clase pueden llegar a generar multiples instancias. (usar sincronizacion)

Factory

Propósito
● Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar.
● Permite que una clase delegue en sus subclases la creación de objetos

Motivación
● Cuando no se puede predecir que subclase instanciar
● Se pasa un tipo sí se quiere obtener un objeto concreto

Estructura



- Producto: Define la interfaz de los objetos que crea el método de fabricación.
- ProductoConcreto: Implementa la interfaz Producto.
- Creador: Declara el método de fabricación, el cual devuelve un objeto de tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelva un objeto ProductoConcreto. Puede llamar al método de fabricación para crear un objeto Producto.
- CreadorConcreto: Redefine el método de fabricación para devolver una instancia de un ProductoConcreto.

Aplicabilidad
- Una clase no puede prever la clase a crear
- Una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea.

Ventajas
- Flexibilidad
- Encapsulamiento (complejidad de creación)

Desventaja

- Se obliga al cliente a definir subclases de la clase Creador sólo para crear un producto concreto y esto puede no ser apropiado siempre

Implementación
Métodos de fabricación parametrizados:

- Otra variante del patrón permite que los métodos de fabricación creen varios tipos de productos.
  - El método de fabricación recibe un parámetro que identifica el tipo de objeto a crear.

Ejemplos

- Creación de autos
- Productos en apps comerciales

3 Variantes Clase Creador

- **Abstracta**: la clase no proporciona una implementación para el método de fabricación que declara.
  - *Implica que las subclases definan una implementación porque no hay ningún comportamiento predeterminado*.
- **Concreta**: el Creador es una clase concreta y proporciona una implementación predeterminada del método de fabricación.
  - *El Creador debe crear objetos en una operación aparte, para que las subclases puedan cambiar la clase de objetos que instancia su clase padre de ser necesario*.
- También es posible tener una clase **abstracta** que define una implementación predeterminada, pero es poco común.

Facade

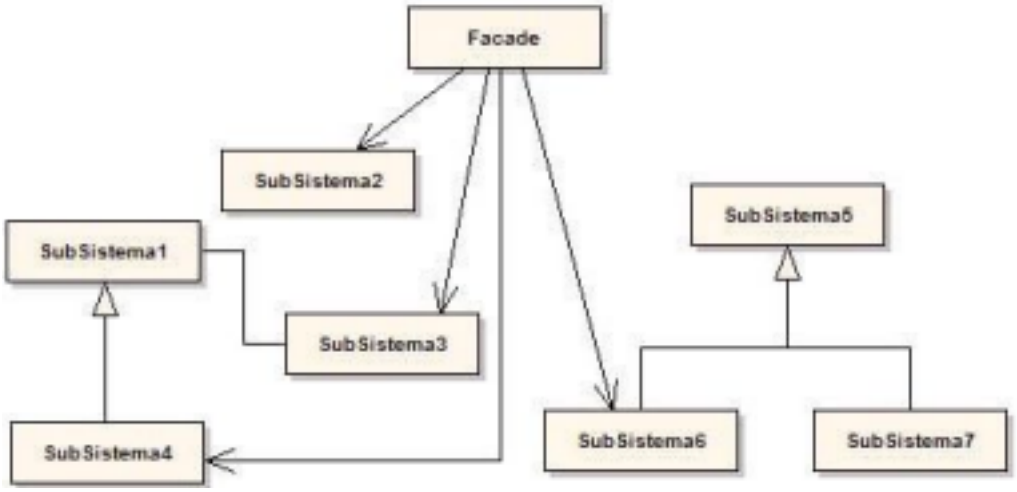
Propósito

- Define interfaz unificada para conjunto de subsistemas
- Interfaz de alto nivel más fácil de usar para el cliente

Motivación

- Reducir complejidad en relaciones entre subsistemas
- Reducir dependencias

Estructura



- Fachada: Sabe qué clases del subsistema son las responsables ante una petición. Delega las peticiones de los clientes en los objetos apropiados del subsistema.
- Clases del subsistema: Implementan la funcionalidad del subsistema. Realizan las labores encomendadas por el objeto fachada. No conocen a la fachada: es decir, no tienen referencias a ella.

Aplicabilidad

- Subsistema evoluciona y crece en cantidad de clases, y este patron hace reutilizable y fácil de personalizar.
- Simplifica la visión de los subsistemas para la mayoría de los clientes
- Se introduce una fachada para desacoplar el subsistema de sus clientes y de otros subsistemas. Aumenta portabilidad.
- Se usa una fachada para definir un punto de entrada en cada nivel del subsistema.

Ventajas

- Abstracción del cliente de los subsistemas
- Bajo acoplamiento (fácil cambiar subsistemas sin afectar clientes)
- Estructurar en capas
- Elimina dependencias
- *No impide que las aplicaciones usen las clases del subsistema en caso de que sea necesario. De este modo se puede elegir entre* ***facilidad de uso*** *(generalidad) y customización.*

Desventajas NO

Implementación

- Recibe peticiones clientes y redirecciona a subsistemas (respeta interfaz entrada y salida - Parámetros)

Ejemplos

- Helpers

Clientes acceden a la fachada?

- Solamente si requieren más personalización

Patrón relacionado

- Singleton: sólo se necesita una fachada. Sólo si no tiene estado interno.

Strategy

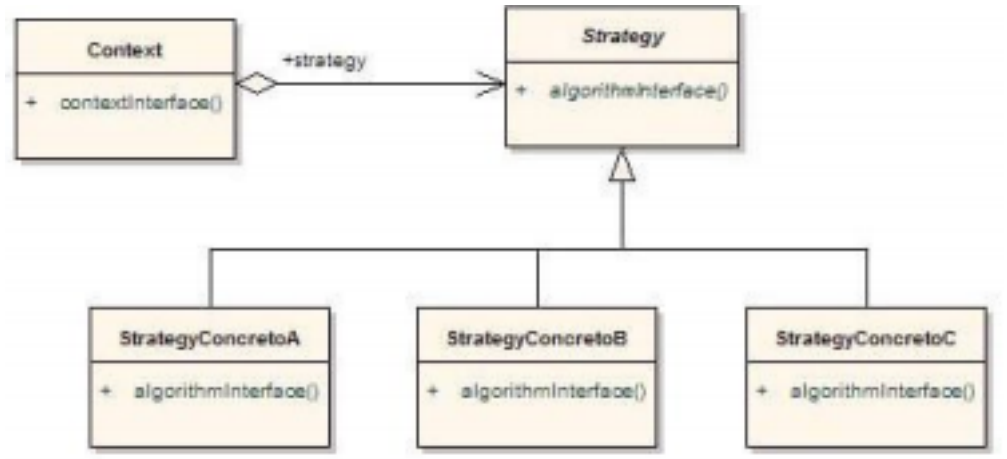
Propósito

- *Define una familia de algoritmos*
- *Encapsula cada uno de ellos y los hace intercambiables.*
- *Permite que un algoritmo varíe independientemente de los clientes que lo usan*

Motivación

- El cliente sí conoce la implementación de un algoritmo complejo se complejiza
- Algoritmos son útiles en determinados momentos
- Difícil cambiar algoritmo sí está acoplado a un cliente

Estructura



- *Estrategia: Declara una interfaz común a todos los algoritmos permitidos. El contexto usa esta interfaz para llamar al algoritmo definido por una EstrategiaConcreta.*
- *EstrategiaConcreta: Implementa el algoritmo usando la interfaz Estrategia.*
- *Contexto: Se configura con un objeto EstrategiaConcreta. Mantiene una referencia a un objeto Estrategia. Puede definir una interfaz que permita a la Estrategia acceder a sus datos.*

Aplicabilidad

- Muchas clases difieren sólo en su comportamiento (estrategia configura eso)
- Variantes de un algoritmo
- Algortimo oculto al cliente
- Se evitan sentencias condicionales para comportamiento, al moverlo a la clase estrategia.

Ventajas

- Familias de algoritmos relacionados (herencia factor comun de funcionalidad)
- Estrategias eliminan sentencias condicionales
- Estrategias distintas implementaciones del mismo comportamiento ● Alternativa a la herencia (sí se heredan objetos contexto se hace dificil de mantener y extender al mezclar algoritmos con el contexto)
- Se puede modificar algoritmo dinamicamente.
- Algoritmo independiente de la clase contextual

Desventajas

- Clientes deben conocer las diferentes estrategias (sólo usar cuando variación de comportamiento sea conocida x clientes)
- Costo de comunicación Estrategia y contexto (datos demás en algunas estrategias concretas)
- Aumenta número de objetos de aplicación

Implementación

- *Hacer opcionales los objetos estrategia*
- *La clase Contexto puede simplificarse en caso de que tenga sentido no tener un objeto Estrategia. El Contexto comprueba si tiene un objeto Estrategia antes de acceder a él. Si existe lo usa normalmente, sino, Contexto realiza el comportamiento predeterminado.*
- *Ventaja: Los clientes no tienen que tratar con los objeto Estrategia a menos que no les sirva el comportamiento predeterminado.*

Ejemplo

- Algoritmos ordenamiento
- Gateways de pago

2 alternativas contexto y estrategia

- *El Contexto puede pasar los datos como parámetros a las operaciones de Estrategia.*

  <u>Ventaja:</u> Esto mantiene a Estrategia y Contexto desacoplados.
  <u>Desventaja:</u> Contexto podría pasar datos a la Estrategia que esta no necesita.
- *Otra técnica consiste en que un contexto se pase a si mismo como argumento, y que la estrategia pida los datos explícitamente al contexto.*

  <u>Ventaja:</u> La estrategia puede pedir exactamente lo que necesita.
  <u>Desventaja:</u> Contexto debe definir una interfaz más elaborada para sus datos, lo que acopla más estrechamente a Estrategia y Contexto.

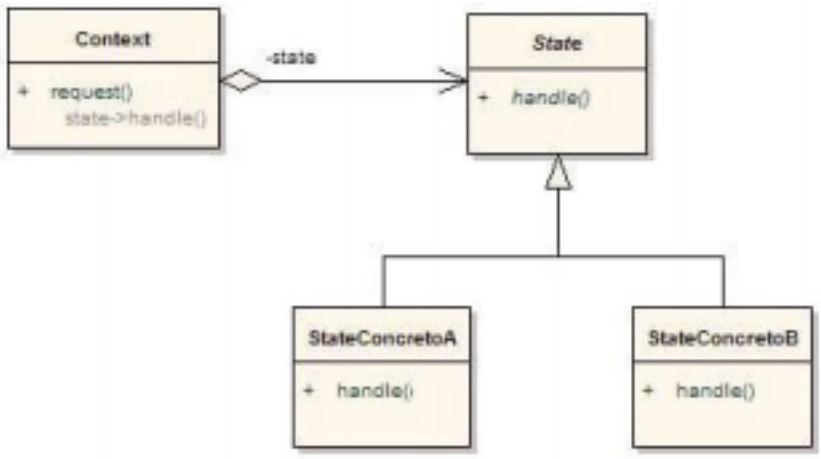
No hay mejor, la necesitad y los requerimientos mostrarán cuál es mejor.

State

Propósito

- *Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.*

Motivación = Estructura



- *<u>Contexto:</u> Define la interfaz de interés para los clientes. Mantiene una instancia de una subclase de EstadoConcreto que define el estado actual.*
- *<u>Estado:</u> Define una interfaz para encapsular el comportamiento asociado con un determinado estado del Contexto.*
- *<u>Subclases de EstadoConcreto:</u> Cada subclase implementa un comportamiento asociado con un estado del Contexto.*

Aplicabilidad

- *El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado.*
- *Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. El Patrón State pone cada rama de la condición en una clase aparte.*

Ventajas

- Localiza el comportamiento dependiente del estado y divide dicho comportamiento en diferentes estados (codigo dependiente del estado esta en estadoconcreto, nuevos estados nuevas subclases)
- Evita uso sentencias condicionales
- Explicita transiciones entre estados (evitas mirar el codigo para saber segun el estado interno en que estado está, se reemplaza por objetos concretos)

Desventajas

- Aumenta número de clases

Implementación

El Patrón State no especifica qué participante define los criterios para las transiciones entre estados. Hay dos enfoques:

- *Los criterios podrían implementarse enteramente en el Contexto. ● Es*

generalmente más flexible y conveniente que sean las propias subclases de Estado quienes especifiquen su estado sucesor y cuándo llevar a cabo la transición. Esto requiere añadir una interfaz al Contexto (métodos) que permita a los objetos Estado modificar el estado actual del Contexto.

- *Ventajas: Descentralizar la lógica de transición facilita modificarla o extenderla definiendo nuevas subclases de Estado.*
- *Desventajas: Una subclase de Estado conocerá al menos a otra, lo que introduce dependencias de implementación entre subclases.*

Ejemplos

- Estados usuarios
- ATM

Active Record

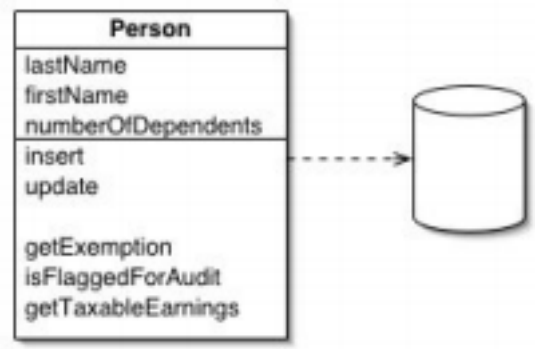
Propósito

- *Un objeto que contiene un registro de una tabla (o vista), encapsula el acceso a la base de datos y agrega lógica de dominio a los datos.*

Motivación

- Persistencia Datos

Estructura



- *<u>ClaseAPersistir:</u> Tiene el comportamiento propio del negocio, además tiene la responsabilidad de persistirse a sí misma en BD.*
- *Es conveniente agregar una <u>Interfaz Persistible</u> que sea común a todos los objetos que deben persistirse en la base de datos.*

Aplicabilidad

- *Recomendable para proyectos pequeños.*
- *Cuando la lógica del negocio es poco compleja.*

Ventajas

- Simple: entender y aplicar

Desventajas

- Esquema isomorfo (coincide obj AR con el de la tabla DB)
- Alto acoplamiento (difícil refactorizar)

Implementación

La estructura de las clases del Active Record debe ser exactamente igual a la estructura de la base de datos: un atributo en la clase por cada columna de la tabla.

La clases que implementan Active Record tienen típicamente métodos para:

- *Construir una instancia del Active Record en base a un resultado SQL.*
- *Construir una nueva instancia para una posterior inserción en una tabla.*
- *Métodos para encapsular las consultas SQL más comunes y devolver objetos Active Record, pueden o no ser estáticos.*
- *Actualizar la base de datos, para hacer updates sobre la tabla asociada al objeto Active Record.*
- *Getters y setters de los campos.*
- *Implementación de lógica de negocio.*

Alternativas a búsquedas del ActiveRecord

- Clases estáticas para consultas SQL

1 a muchos en Active Record

- *Esto es como lo vimos con los ejemplos, como Usuario tiene muchos Mails, se agrega la FK del usuario a la clase mail. La respuesta formal sería agregando la FK a la clase de ordinalidad n. Simplemente eso.*

Muchos a muchos

- *También con ejemplos de clase, si muchos comedores tienen muchos asistentes, lo que se hace en este caso es que cada comedor tiene una lista de asistentes, donde cada asistente tiene el id de ese comedor y como es una relación de muchos a muchos, los asistentes se pueden repetir para distintos comedores.*

Se relacionan clases de persistencia y no persistidas?
Por supuesto, sí hacen a la lógica de negocio
Strategy por ejemplo no se persiste.