Patrones de Diseño

Alejandro Mancilla
@alxmancilla

Agenda del curso

- Patrones ¿qué son y para qué sirven?
- Categorías de Patrones
- Clasificación de Patrones de Diseño
- Patrones de Creación - ¿Cómo creo un objeto?
- Patrones de Estructura - ¿Cómo pueden trabajar una clase y un objeto?
- Patrones de Comportamiento - ¿Cómo interactúo entre objetos?
- Patrones de Arquitectura - ¿Cómo diseño una aplicación?

Clasificación de Patrones de Diseño (GoF)

		Propósito		
		Creación	Estructura	Comportamiento
Alcance	Clase	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Patrones de Diseño de Creación - ¿Cómo creo un objeto?

- Factory Method
  Crea una instancia de varias clases derivadas
- Abstract Factory
  Crea una instancia de varias familias de clases
- Builder
  Separa la construcción de un objeto de su representación
- Prototype
  Una instancia inicializada para ser copiada o clonada
- Singleton
  Una clase de la que sólo puede existir una instancia

Patrones de Estructura
- ¿Cómo pueden trabajar una clase y un objeto?

- Adapter
  Relaciona interfaces de diferentes clases
- Bridge
  Separa la interfaz de un objeto de su implementación
- Composite
  Una estructura de árbol de objetos simples y compuestos
- Decorator
  Añadir responsabilidades a los objetos dinámicamente
- Façade
  Una única clase que representa todo un subsistema
- Flyweight
  Una instancia usada para compartición eficiente
- Proxy
  Un objeto que representa a otro objeto

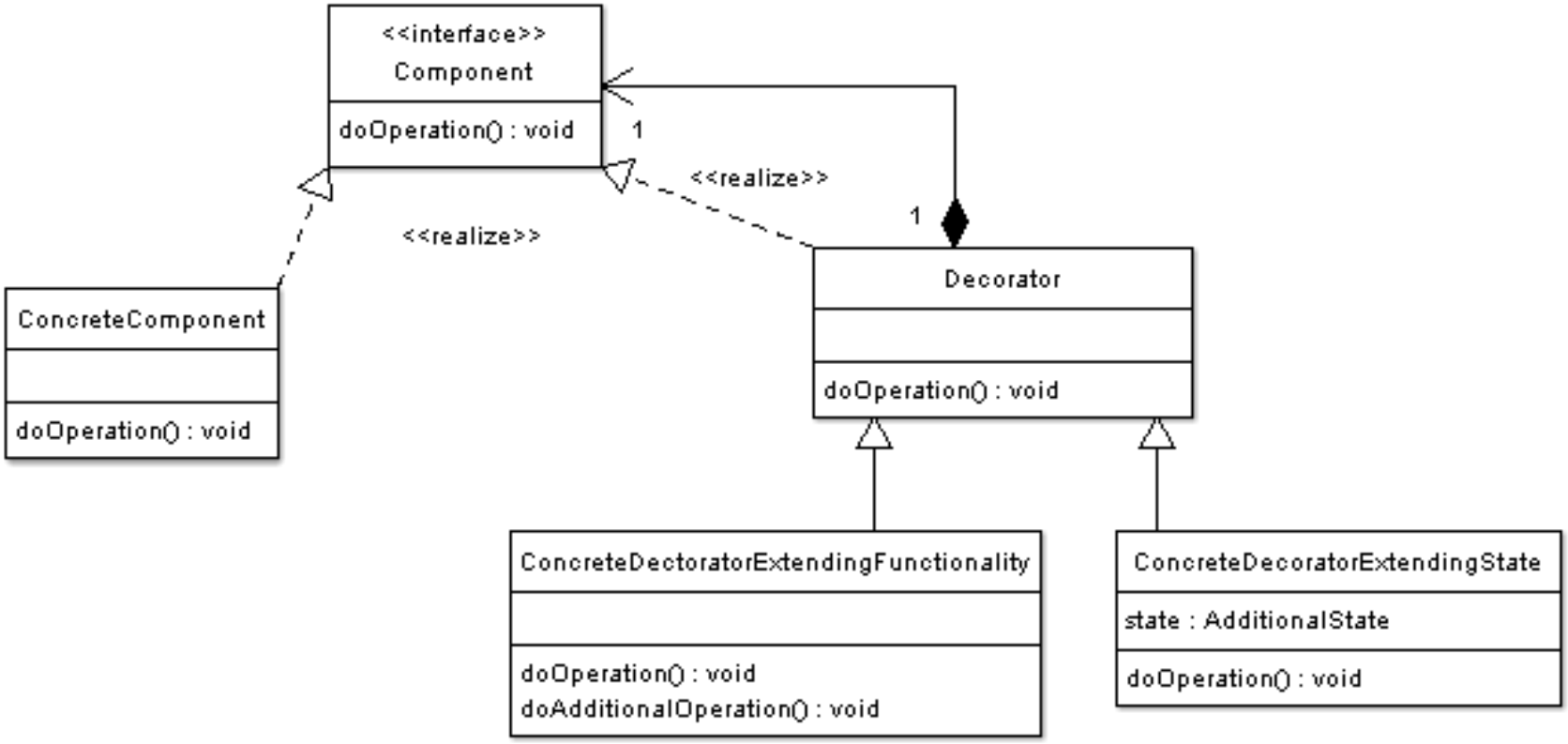
Patrón: Decorator (a.k.a. Wrapper)

- Problema:
  - Agregar responsabilidades a objetos individuales dinámica y transparentemente
- Contexto:
  - A veces queremos añadir responsabilidades a objetos individuales, no para toda una clase
- Solución
  - Adjuntar responsabilidades adicionales a un objeto de forma dinámica.
  - Decorators proporcionan una alternativa flexible a las subclases para ampliar la funcionalidad.

Participantes del patrón: Decorator

- **Component**
  - Interfaz para objetos que pueden tener responsabilidades añadidas dinámicamente
- **ConcreteComponent**
  - Define un objeto al cuál se le pueden añadir responsabilidades adicionales
- **Decorator**
  - Mantiene una referencia al objeto **Component** y define una interfaz que cumple con la interfaz de **Component**
- **Concrete Decorators**
  - Extienden la funcionalidad del **Component** al añadirle estado o comportamiento

Participantes del patrón: Decorator



Ejemplo del Patrón Decorator

Algunas notas sobre Patrones de Estructura

- **AbstractFactory** puede ser usado como una alternativa a **Façade** para esconder clases específicas de una platforma.
- **Adapter** proporciona una interfaz diferente a su objeto.
- **Proxy** proporciona la misma intefaz.
- **Decorator** proporciona una interfaz aumentada.
- Los objetos **Façade** a menudo son **Singleton**, ya que sólo se requiere un objeto **Façade**.

Patrones de Comportamiento - ¿Cómo interactúo entre objetos?

- Chain of responsibility
  Una manera de pasar una petición a una cadena de objetos
- Command
  Encapsular una petición de comandos como un objeto
- Interpreter
  Una manera de incluir elementos del lenguaje en un programa
- Iterator
  Acceso secuencial a los elementos de una colección
- Mediator
  Define una comunicación simplificada entre clases

Patrones de Comportamiento - ¿Cómo interactúo entre objetos?

- Memento
  - Captura y restaura el estado interno de un objeto
- Observer
  - Una forma de notificar cambios a varias clases
- State
  - Altera el comportamiento de un objeto cuando cambia de estado
- Strategy
  - Encapsula un algoritmo dentro de una clase
- Template method
  - Difiere los pasos exactos de un algoritmos a una subclase
- Visitor
  - Define una nueva operación a una clase sin cambiarla

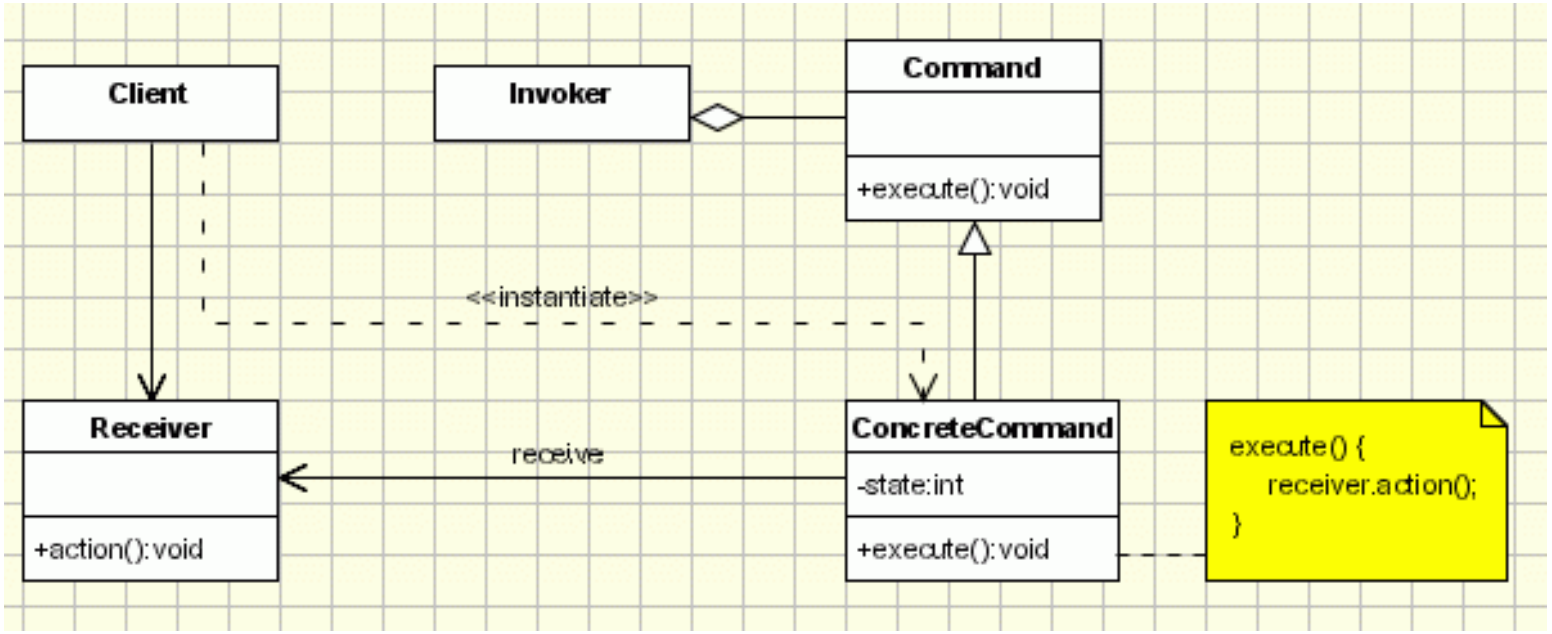
Patrón: Command (a.k.a. Action, Transaction)

- Problema:
  - Especificar, encolar y ejecutar peticiones en diferentes tiempos
  - Callbacks
- Contexto:
  - Emisiión de peticiones a objetos sin saber nada de la operación que se solicite o el receptor de la solicitud
- Solución
  - Encapsular una petición como un objeto
  - Almacenar las peticiones en una cola

Participantes del patrón: Command

- **Command**
  - Declara una interfaz para ejecutar una operación
- **ConcreteCommand**
  - Extiende Command, implementando el método execute al invocar las operaciones correspondientes en el Receiver
- **Invoker**
  - Le pide al Command que ejecute la petición
- **Receiver**
  - Sabe cómo ejecutar las operaciones
- **Client**
  - Crea un objeto ConcreteCommand y le asigna su Receiver

Participantes del patrón: Command



Ejemplo de Command

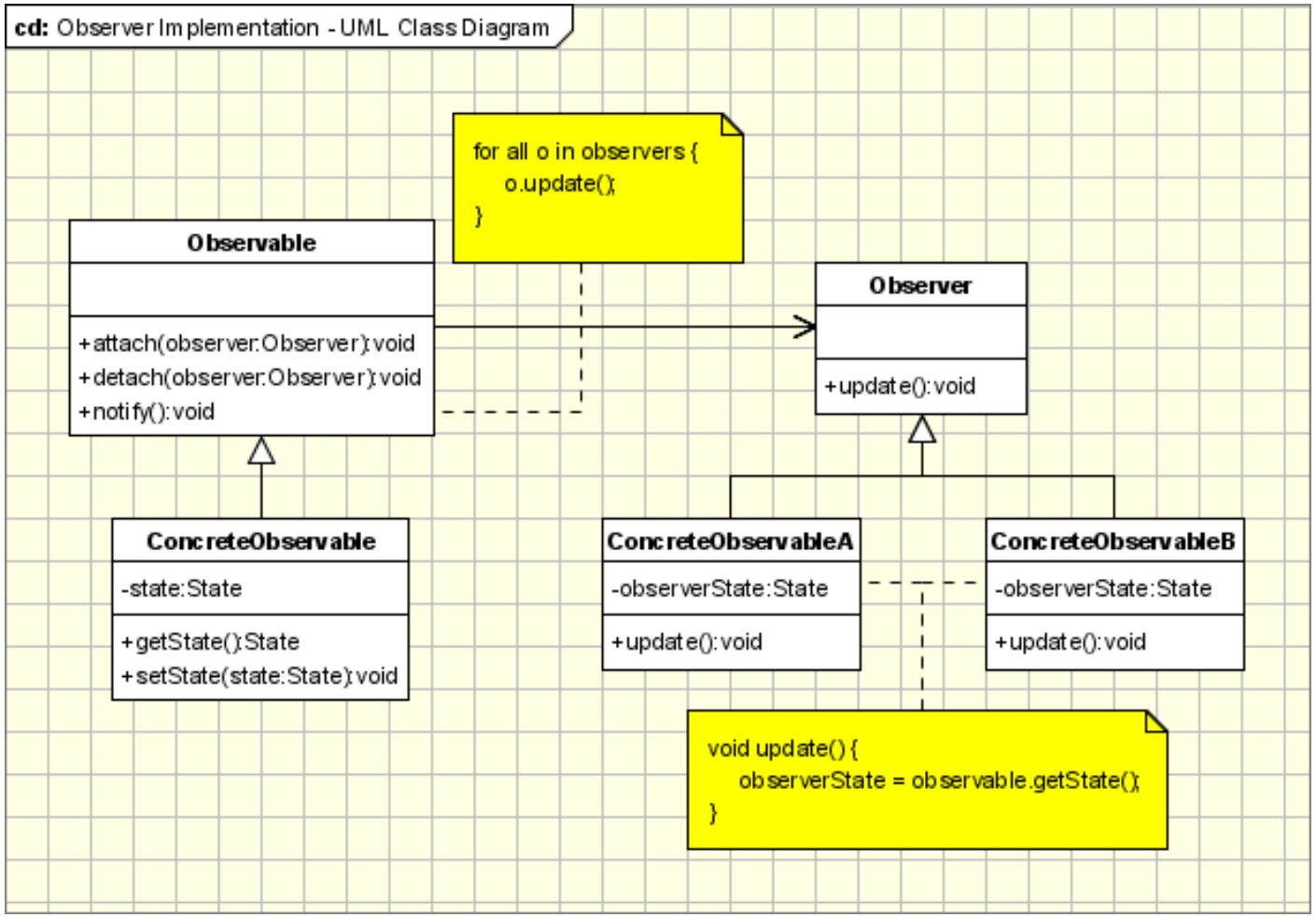
Patrón: Observer (a.k.a. Dependents, Publish-Subscribe)

- Problema:
  - Un cambio en un objeto requiere cambios en otros,y no sabes cuántos objetos necesitan ser cambiados
  - Una abstracción tiene dos aspectos dependientes.
- Contexto:
  - Al particionar un sistema en una colección de clases cooperativas, se requiere mantener la consistencia entre objetos relacionados
- Solución
  - Definir un dependencia uno-a-muchos entre objetos, para que al cambiar un objeto, todos sus dependientes sean notificados automáticamente.

Participantes del patrón: Observer

- **Observable**
  - Declara una interfaz para añadir o remover Observers del cliente
- **ConcreteObservable**
  - Extiende Observable. Mantiene el estado del objeto y cuando cambia, notifica a los Observers ligados
- **Observer**
  - Interfaz que define las operaciones a ser usadas para notificar a este objeto
- **ConcreteObserverA, ConcreteObserver2**
  - Implementaciones concretas de Observer

Participantes del patrón: Observer



Ejemplo de Observer

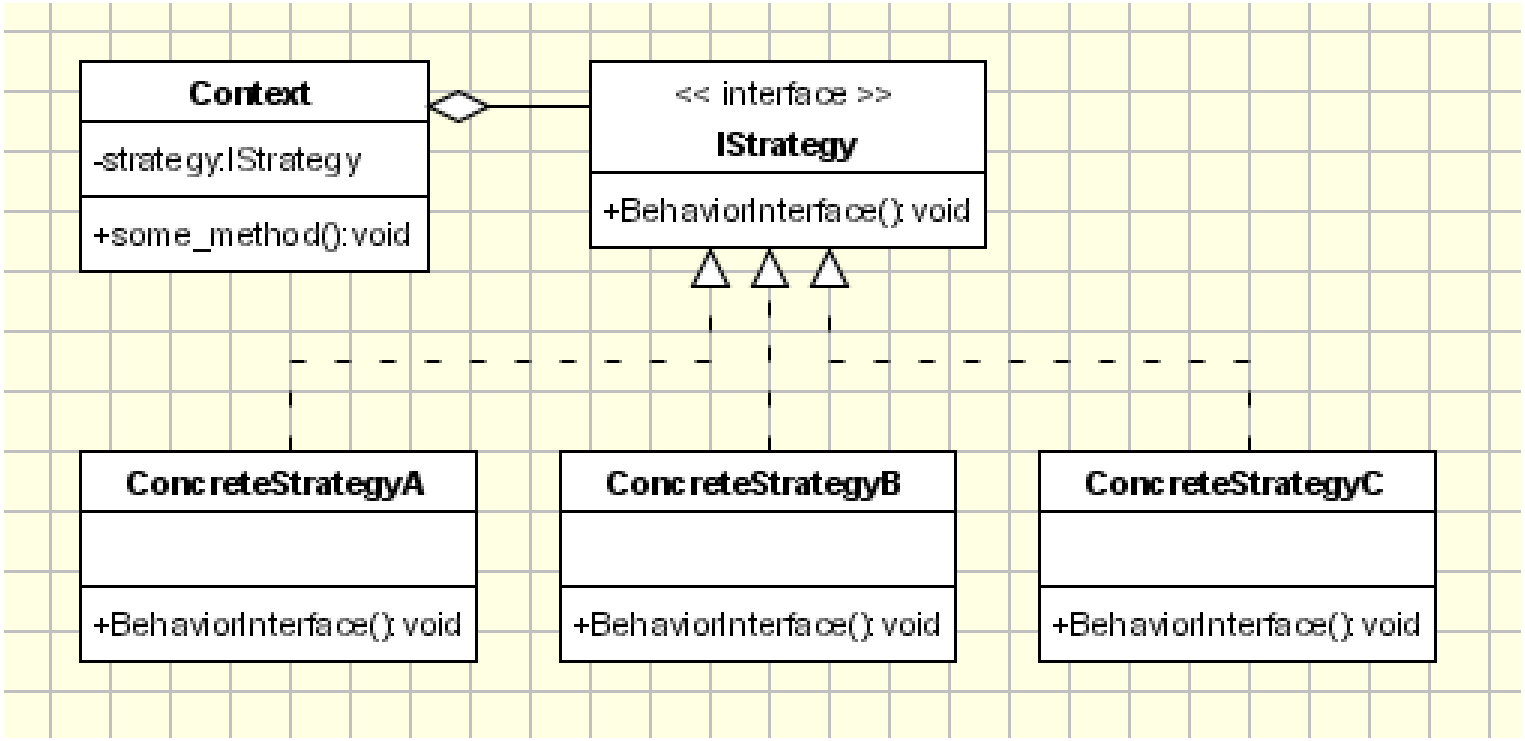
Patrón: Strategy (a.k.a. Policy)

- Problema:
  - Requieres diferentes variantes de un algoritmo
- Contexto:
  - Clases relacionadas sólo difieren en su comportamiento
- Solución
  - Define una familia de algoritmos, los encapsula, y los hace intercambiables.

Participantes del patrón: Strategy

- **Strategy**
  - Declara una interfaz para soportar todos los algoritmos
- **ConcreteStrategy**
  - Extiende a **Strategy**. Cada **ConcreteStrategy** implementa un algoritmo.
- **Context**
  - Mantiene una referencia al objeto **Strategy**
  - Define una interfaz para an interface que deja a Strategy accesar sus datos.

Participantes del patrón: Strategy



Ejemplo de Strategy

Algunas notas sobre Patrones de Comportamiento

- Chain of responsibility puede usar Command para representar peticiones como objetos.
- Objetos State a menudo son Singletons.
- Strategy te permite cambiar la parte interna de un objeto, mientras que Decorator te permite cambiar la piel.
- Strategy para los algoritmos, como Builder es para la creación de objetos.

Patrones de Arquitectura - ¿Cómo diseño una aplicación?

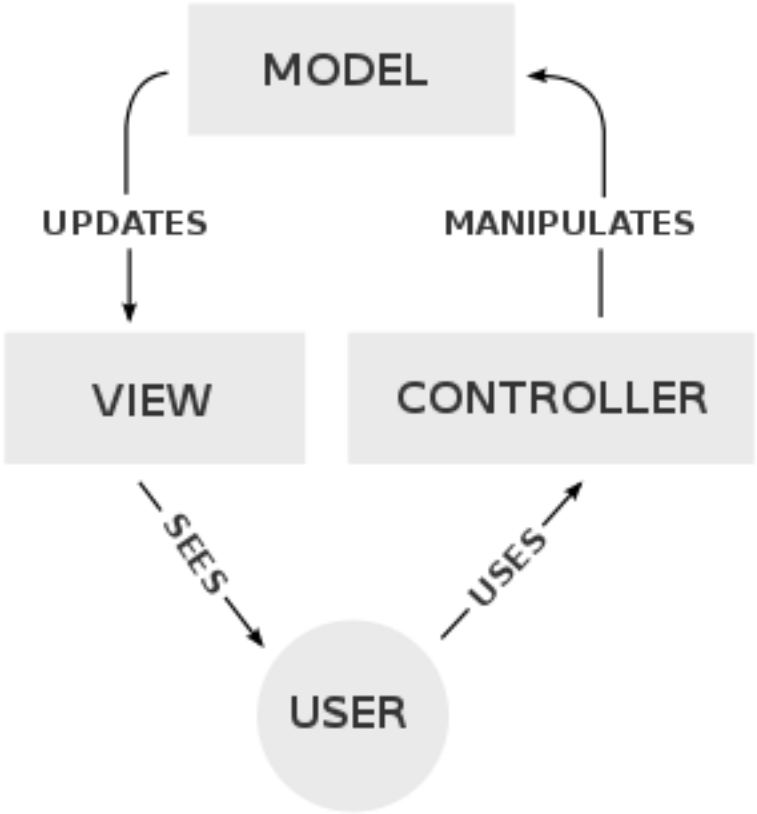
- MVC
  - Model-View-Controller
- EDA
  - Event-Driven Architecture

MVC

- **Model–view–controller** (**MVC**) es un patrón de arquitectura de software para la implementación de interfaces de usuario.
- Divide una aplicación de software dado en tres partes interconectadas.

Participantes de MVC

- Model
- View
- Controller


MODEL
UPDATES
MANIPULATES
VIEW
CONTROLLER
SEES
USES
USER

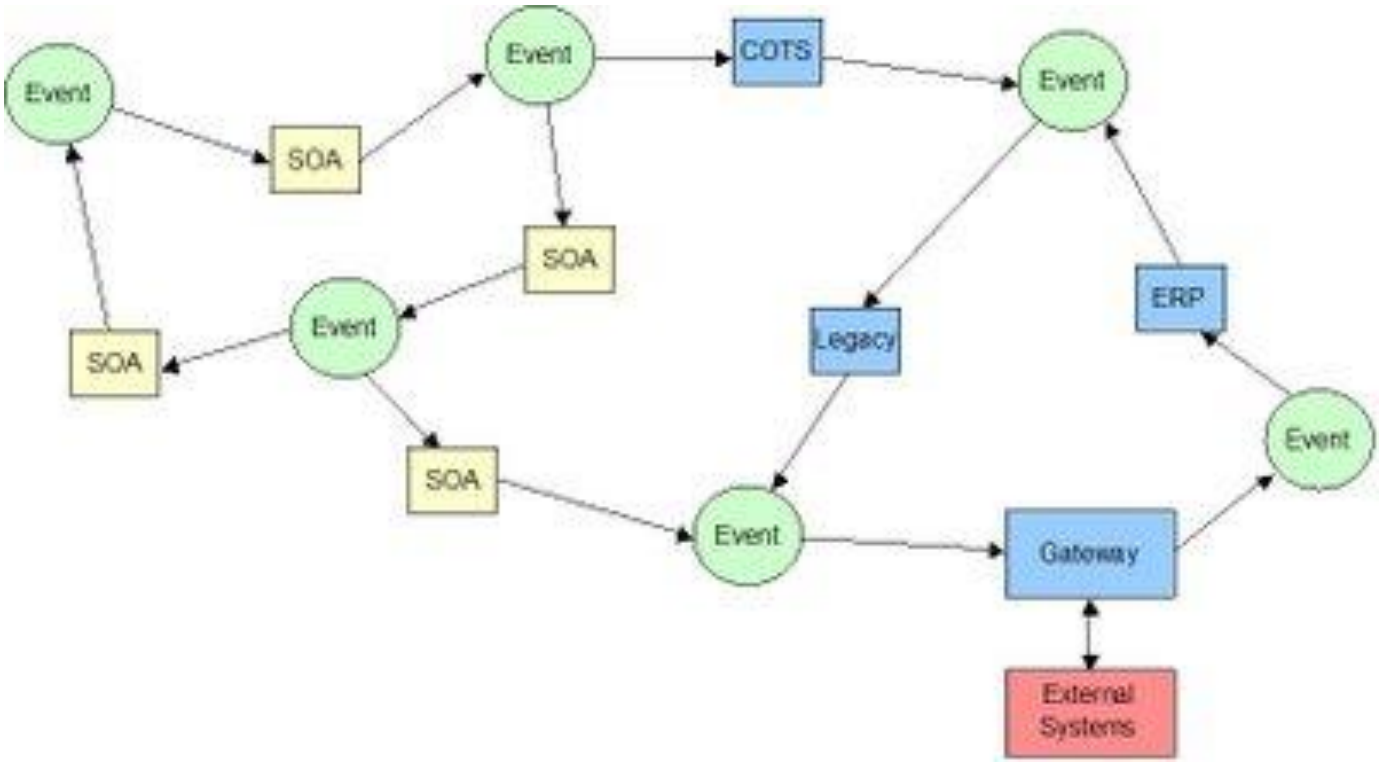
EDA

- **Event–driven Architecture** (**EDA**) es un patrón de arquitectura de software que promueve la producción, la detección, el consumo de, y reacción a los eventos.
- Este patrón se puede aplicar para el diseño e implementación de aplicaciones y sistemas que transmiten eventos entre los componentes de software y servicios débilmente acoplados.

Participantes de EDA

- Event emitters
  - tienen la responsabilidad de detectar, recoger, y transferir los eventos
- Event consumers

- Event channels
  - son conductos en el que los eventos se transmiten de emisores de eventos para los consumidores de eventos.


Event
SOA
Event
COTS
Event
SOA
SOA
Event
ERP
Legacy
SOA
Event
Event
Gateway
External Systems