# DESARROLLO INCREMENTAL E ITERATIVO

Fundación Jala®

innova SANTA CRUZ cainco

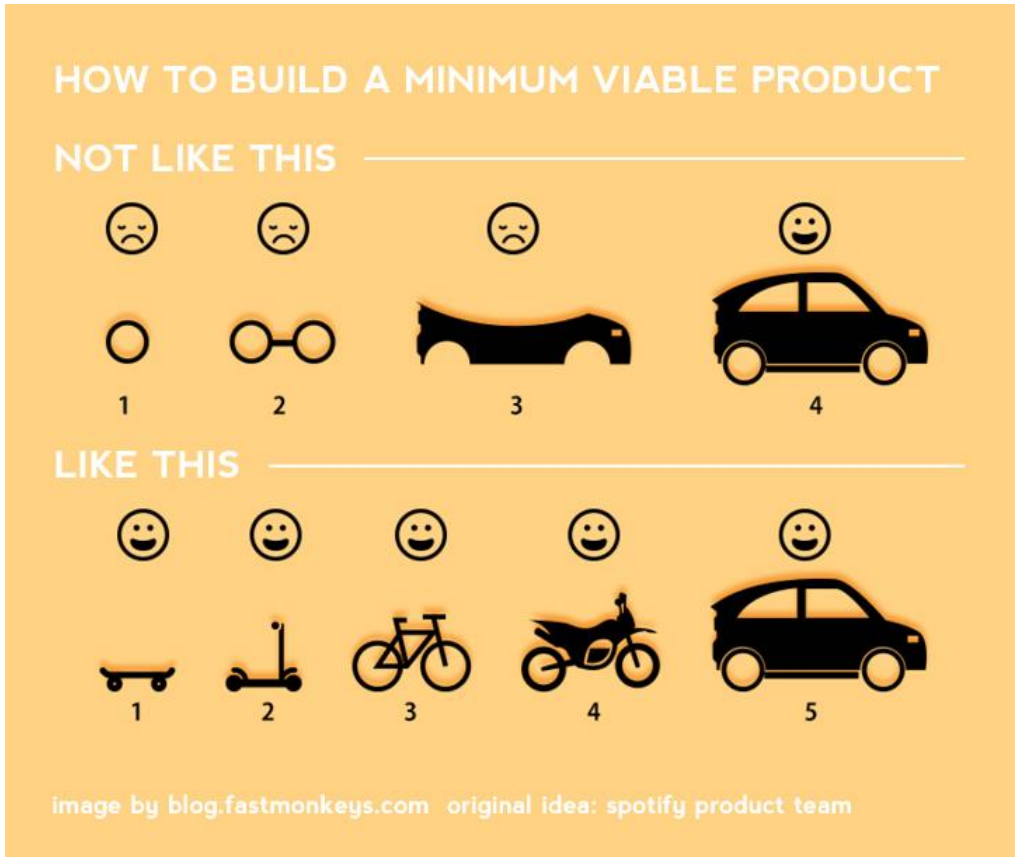
# Agenda

- Qué significa Iterativo e incremental.
- Características del software: acoplamiento, cohesión, encapsulación.
- SOLID
- Ejercicio práctico.
- Deuda técnica.

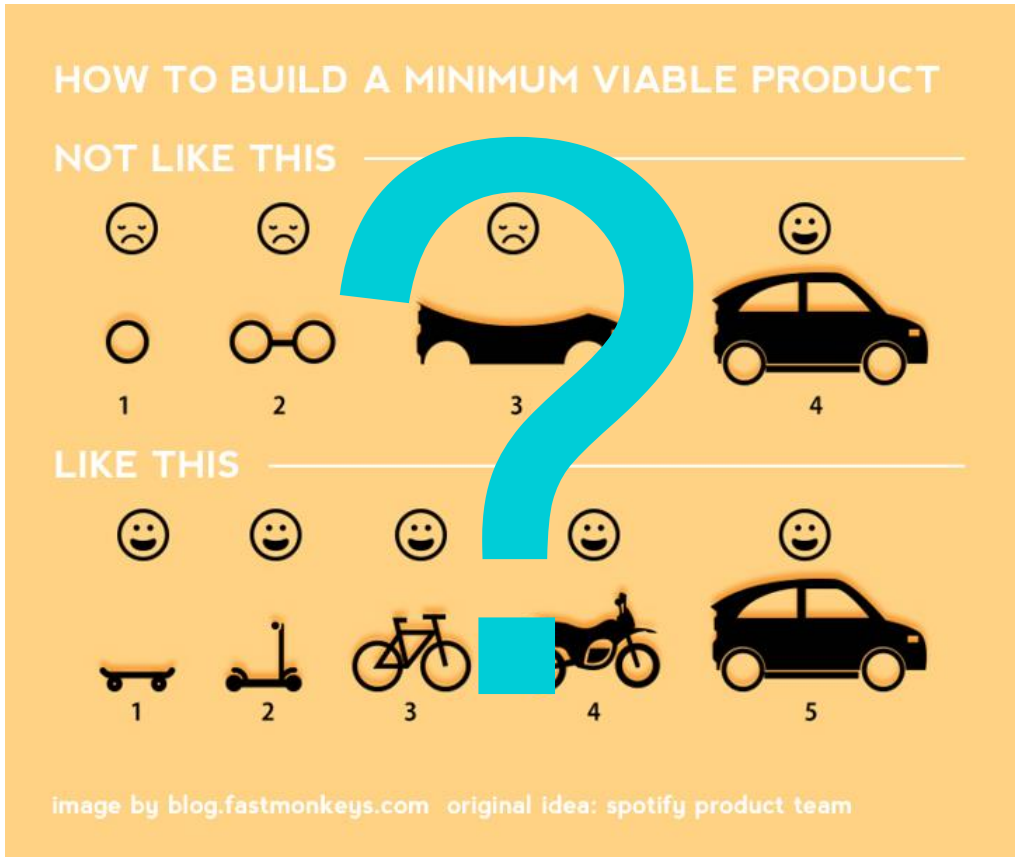
Fundación Jala

innova SANTA CRUZ cainco

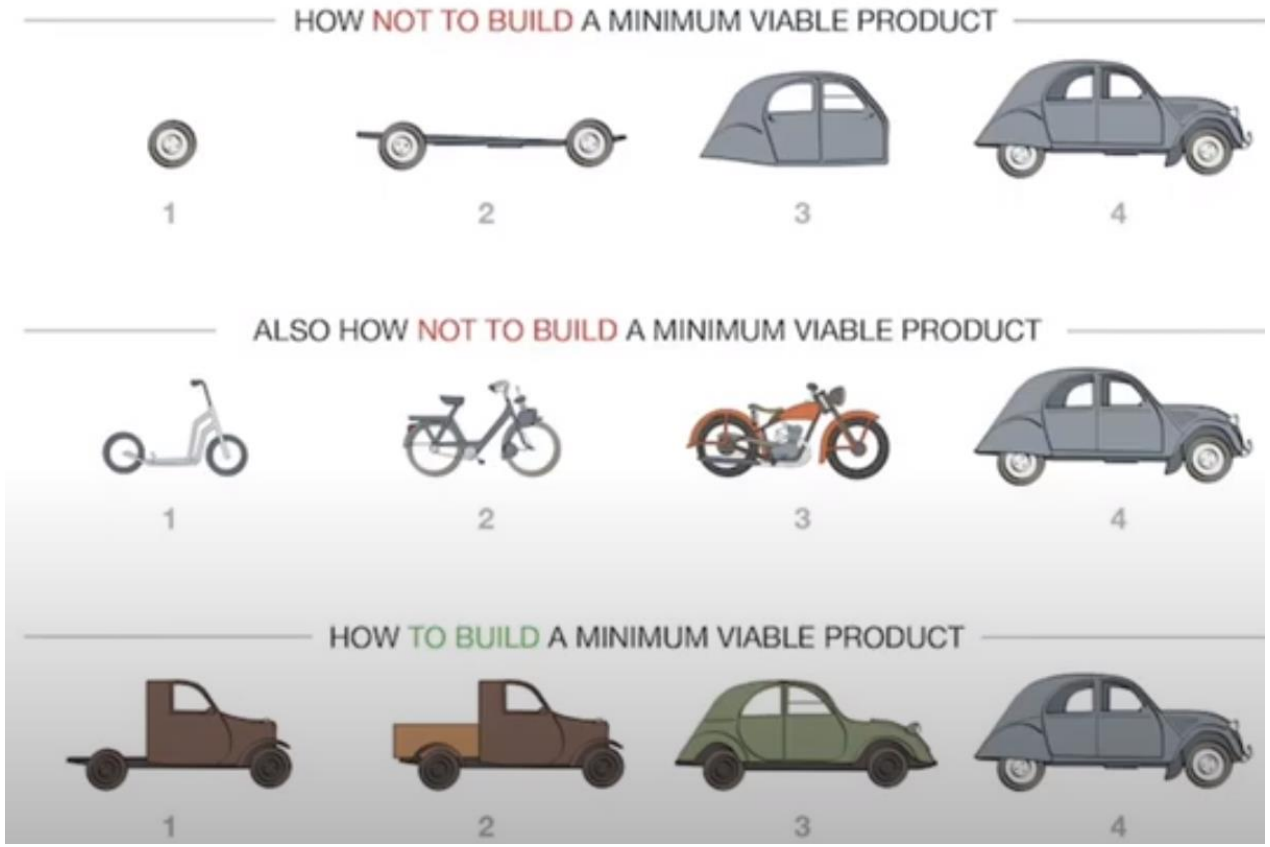
# Iterativo e incremental



# Iterativo e incremental



# Iterativo e incremental




https://blog.mozilla.org/ux/2019/05/the-11-secrets-of-iterative-and-incremental-product-development-a-lightining-talk/

# Iterativo e incremental

- Necesitamos una base sólida: chasis, motor.
- Vamos aumentando (incrementando) nueva funcionalidad.
- La “base” debe poder soportar extensiones.

LO QUE DEBEMOS SABER PARA…

# Construir una base que sea extensible

# Acoplamiento

El acoplamiento o dependencia es el grado en que cada módulo del programa se basa en cada uno de los otros módulos.

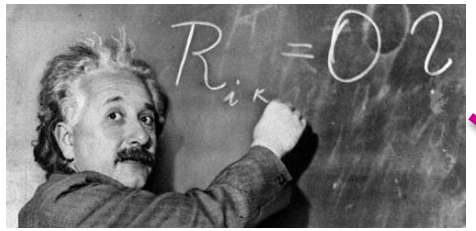
# Acoplamiento

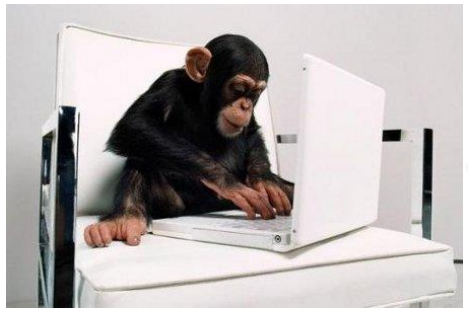


# Cohesión

# Cohesión

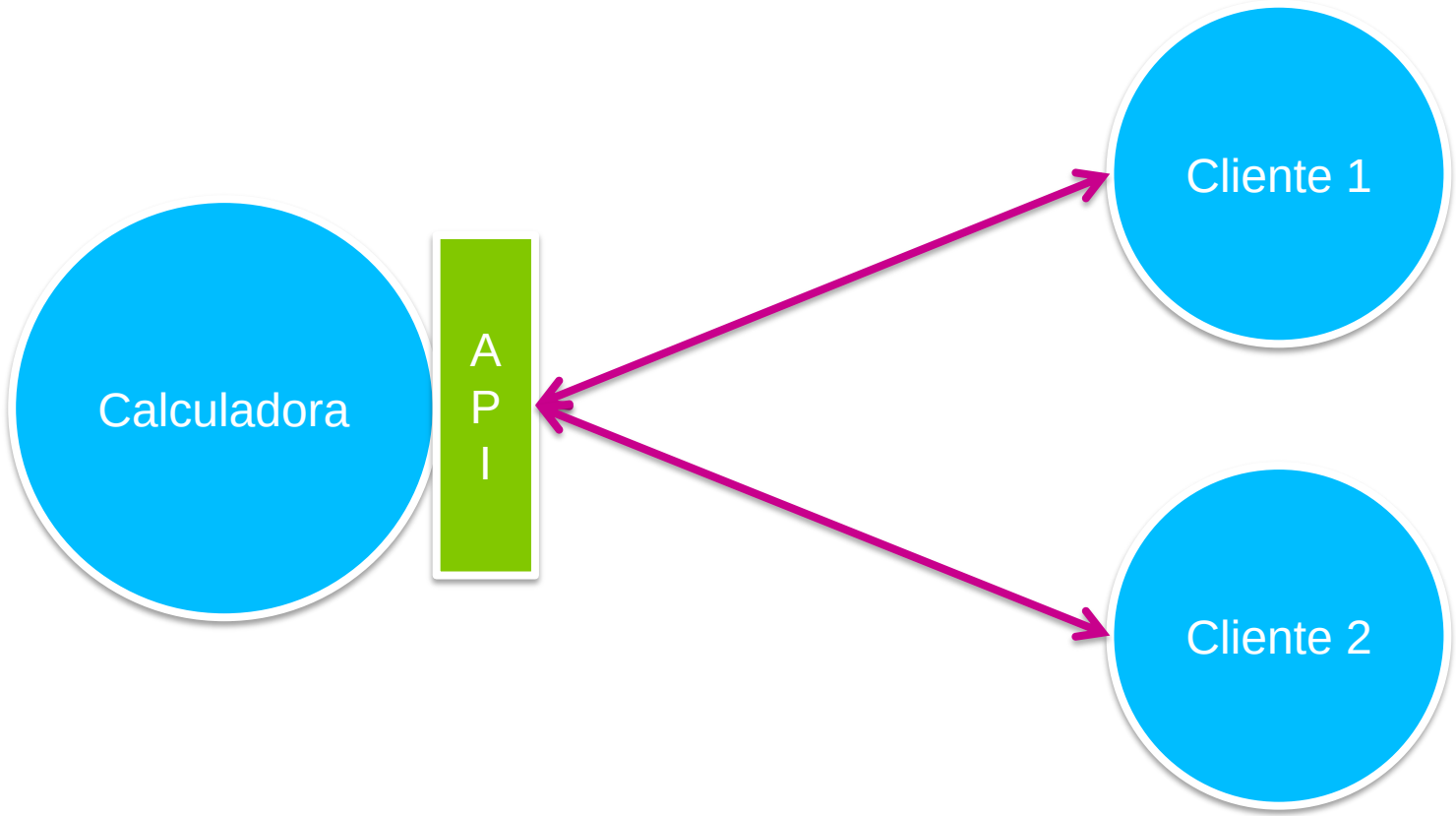
- La cohesión se refiere al grado en que los elementos de un módulo pertenecen juntos.
- La cohesión es una medida de cuán fuertemente relacionadas o enfocadas están las responsabilidades de un solo módulo.






Interface de la calculadora

# Encapsulación




# Encapsulación

- Las interfaces deben representar la fachada del módulo.
- El diseño comienza con la identificación de un conjunto de decisiones de diseño importantes que pueden cambiar durante el desarrollo.
- Cada una de estas decisiones debe encapsularse en un módulo independiente.
- Como resultado de este proceso, los cambios en el sistema de software deben limitarse dentro de los límites de un módulo.

# Encapsulación

# Las características deseadas

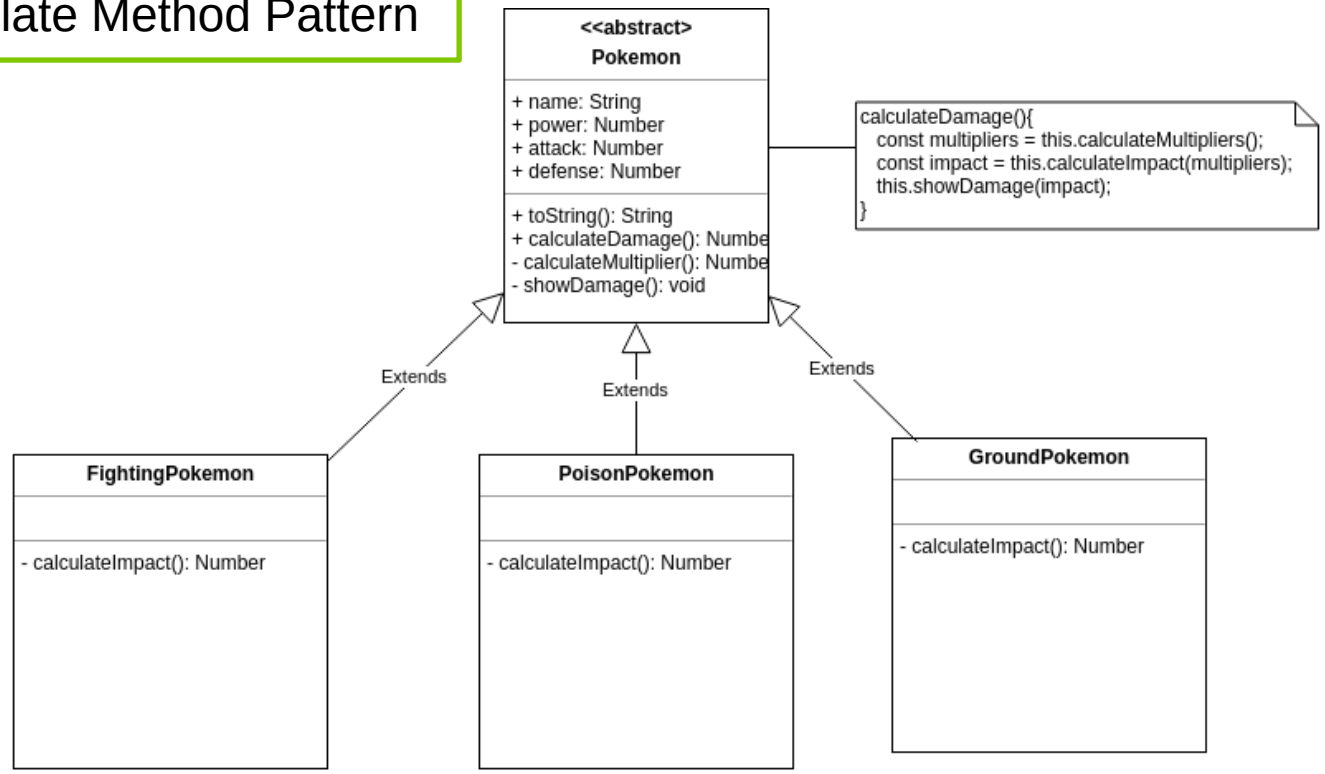
- Alta cohesión.
- Bajo acoplamiento.
- Buena encapsulación.


# Principios útiles


# Principios útiles

# Las buenas recetas cumplen los principios

Template Method Pattern

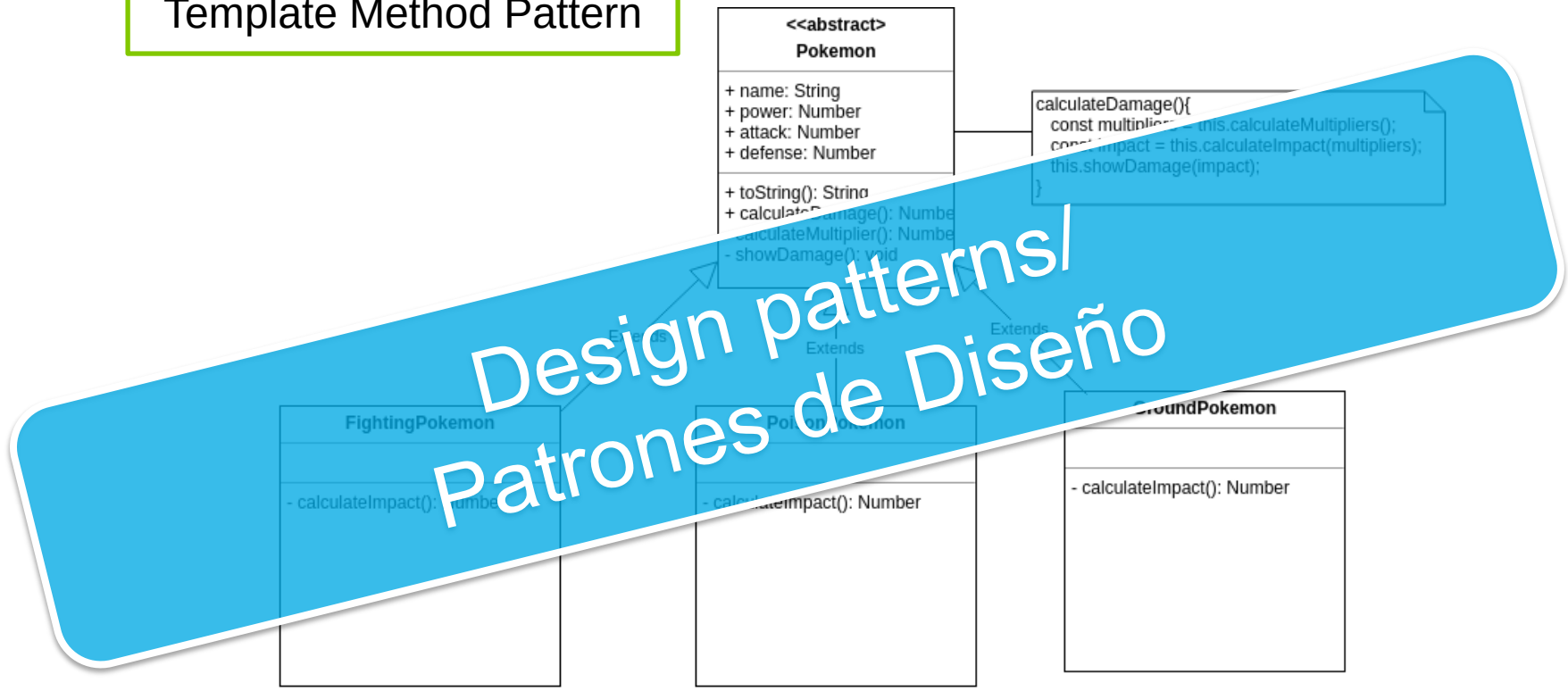



Fundación Jala

innova SANTA CRUZ cainco

# Las buenas recetas cumplen los principios

Template Method Pattern

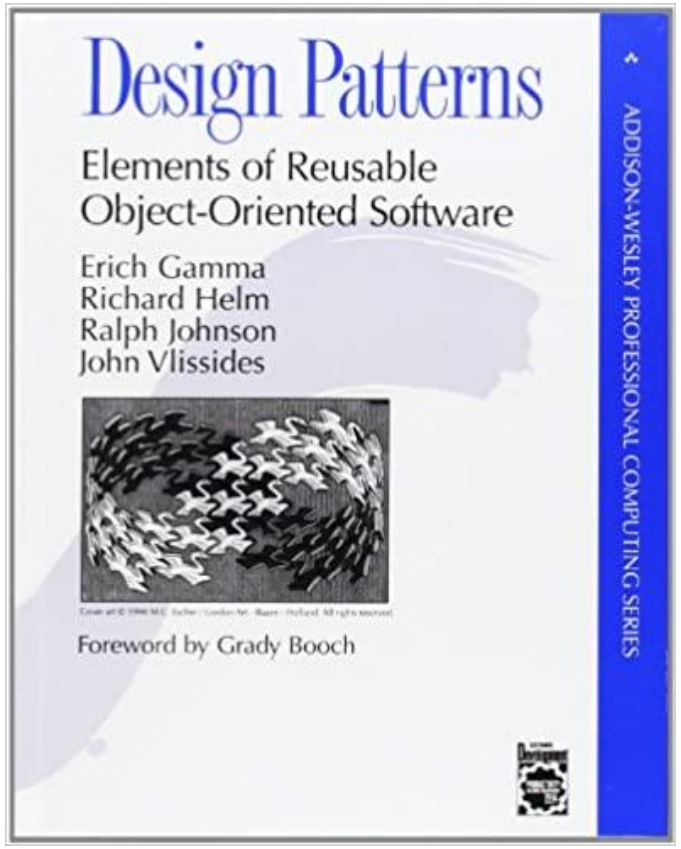



Design patterns/
Patrones de Diseño

Fundación Jala

innova SANTA CRUZ cainco

# Un buen compendio de estas recetas





Fundación Jala

innova SANTA CRUZ cainco

# Cómo te vuelves un campeón de ajedrez?

- Jugando al ajedrez.
- Aprendiendo los principios.
- Estudiando jugadas de los maestros.
- Aplicando las jugadas de los maestros en tus propias partidas.

Fundación Jala

innova SANTA CRUZ cainco

# Cómo te vuelves un campeón de ajedrez?

- Jugando al ajedrez.
- Aprendiendo los princ
- Estudiando jugadas c os.
- Aplicando las jugadas tros en tus propias partidas.


Fundación Jala

innova SANTA CRUZ cainco

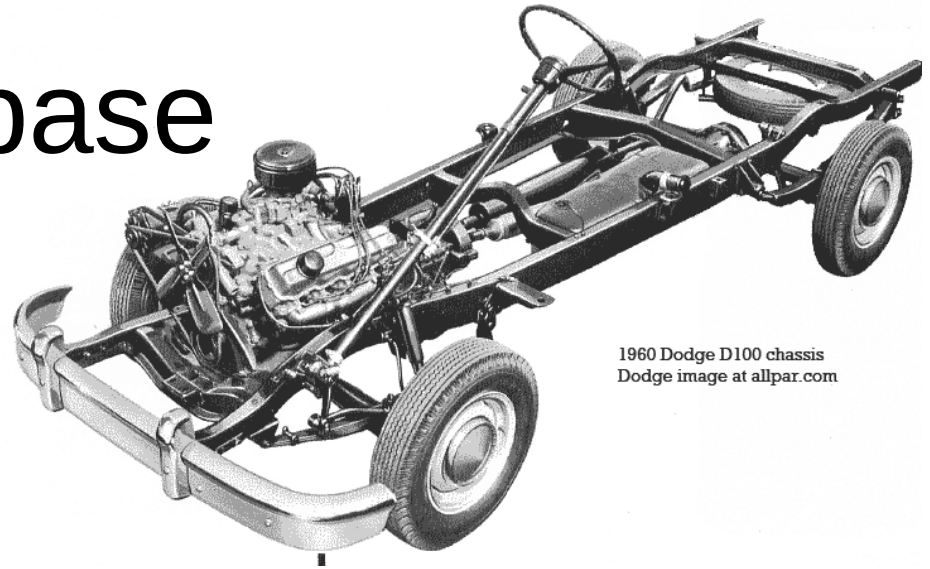
# Cómo te vuelves un diseñador de software?

- Haciendo software (programando).
- Aprendiendo los principios de diseño.
- Estudiando los diseños de los maestros.
- Aplicando las recetas de los maestros en tus propias soluciones.

Fundación Jala

innova SANTA CRUZ cainco

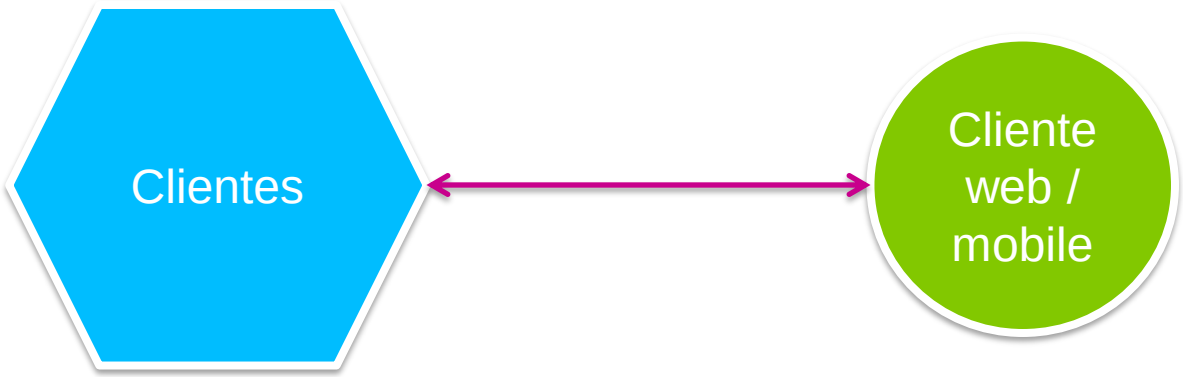
# Construyendo una buena base



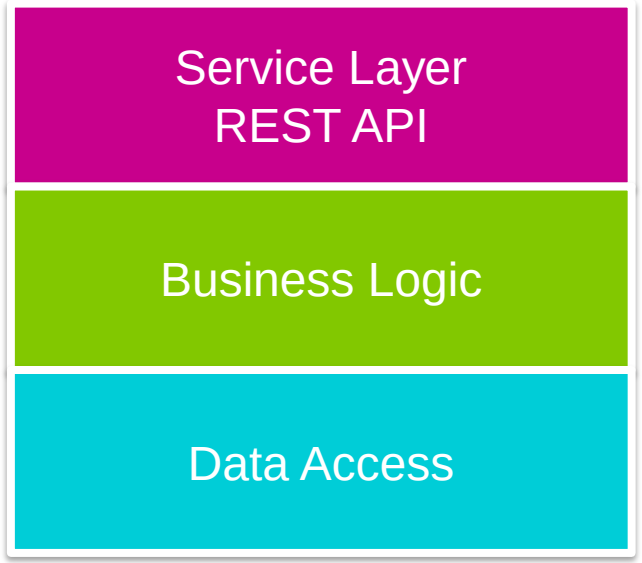
1960 Dodge D100 chassis
Dodge image at allpar.com

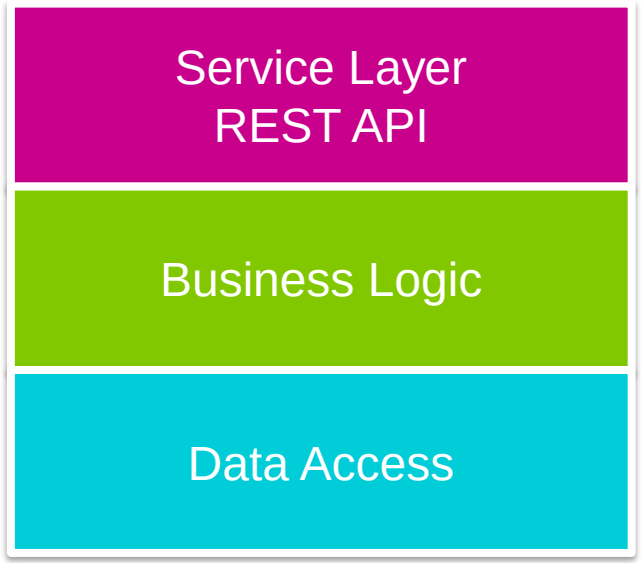
# Funcionalidad

- Es necesario registrar clientes para tener un registro de las personas que utilizan nuestros Servicios.


Clientes
Cliente web / mobile

# Back-End


Service Layer
REST API
Business Logic
Data Access

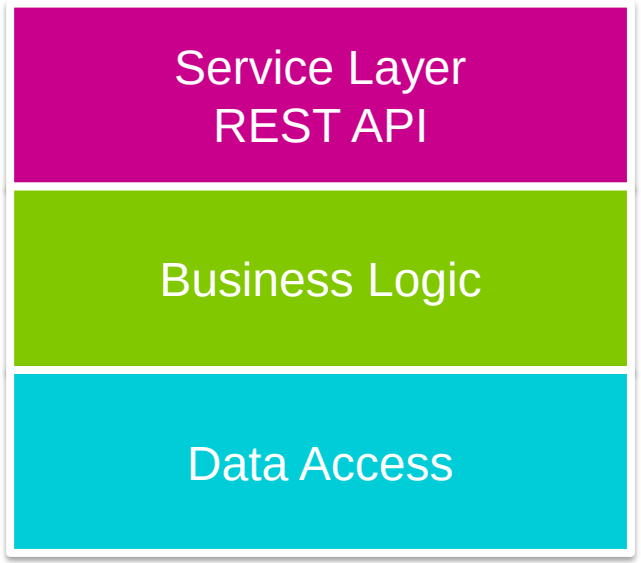



Layered Architecture
https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

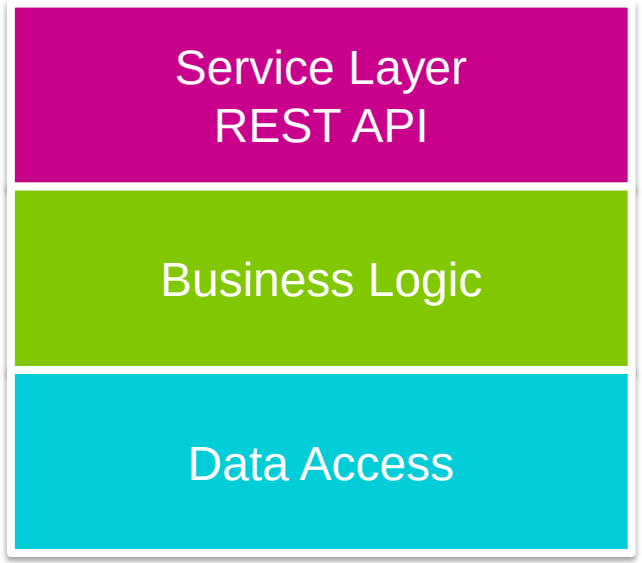
- Autenticación/Autorización
- Reconstrucción de contexto
- Múltiples clientes

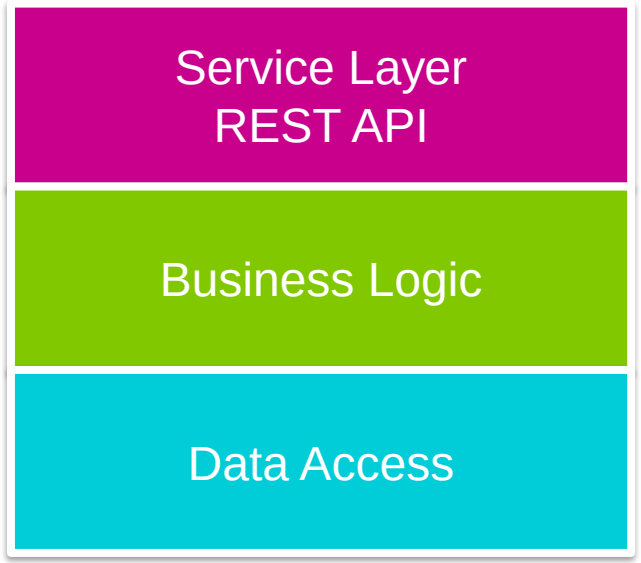
- Funcionalidad, validaciones
- Interacción entre las entidades
- Seguridad, Auditoría (auditing) ?

- Persistencia de datos




Autenticación/Autorización •
Reconstrucción de contexto •
Múltiples clientes •
Funcionalidad •
Interacción entre las entidades •
Seguridad, Auditoría (auditing) •
Persistencia de datos •
Service Layer
REST API
Business Logic
Data Access
Single Responsibility -> High cohesion


Service Layer
REST API
Business Logic
Data Access




**Recurso**: clientes
**Verbos**: POST, GET, PATCH, DELETE

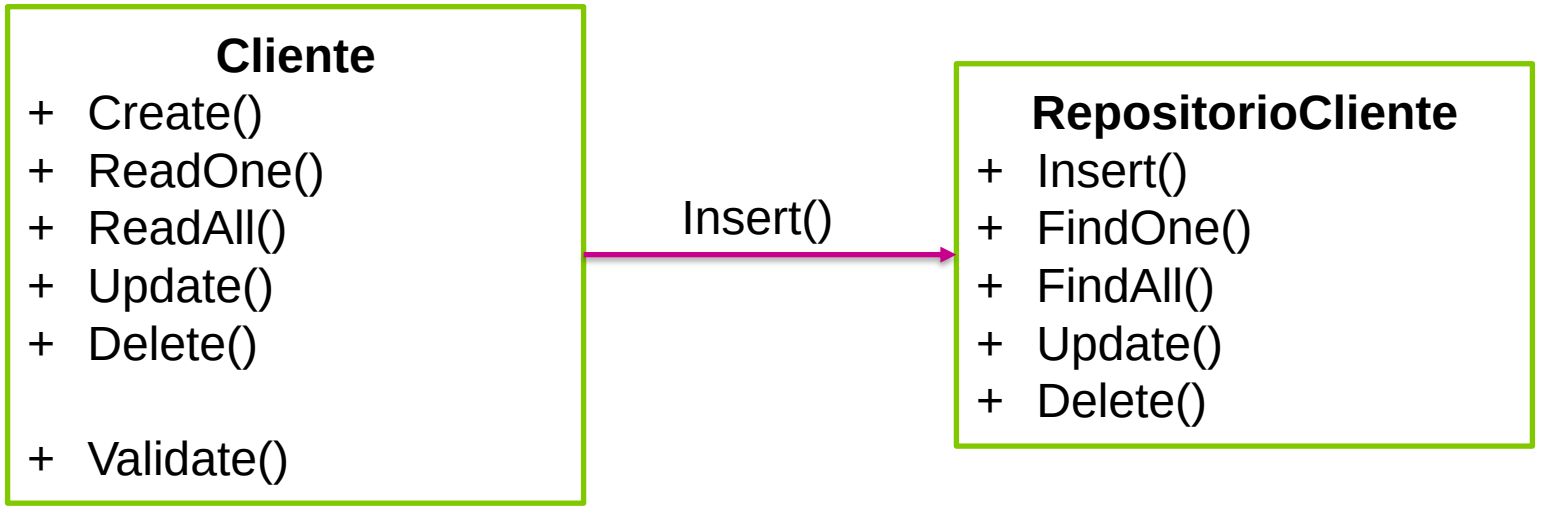
**Objeto**: Cliente
**Métodos**: Create, ReadAll, ReadOne, Update, Delete (CRUD)

**Tabla/Colección**: Cliente(s)

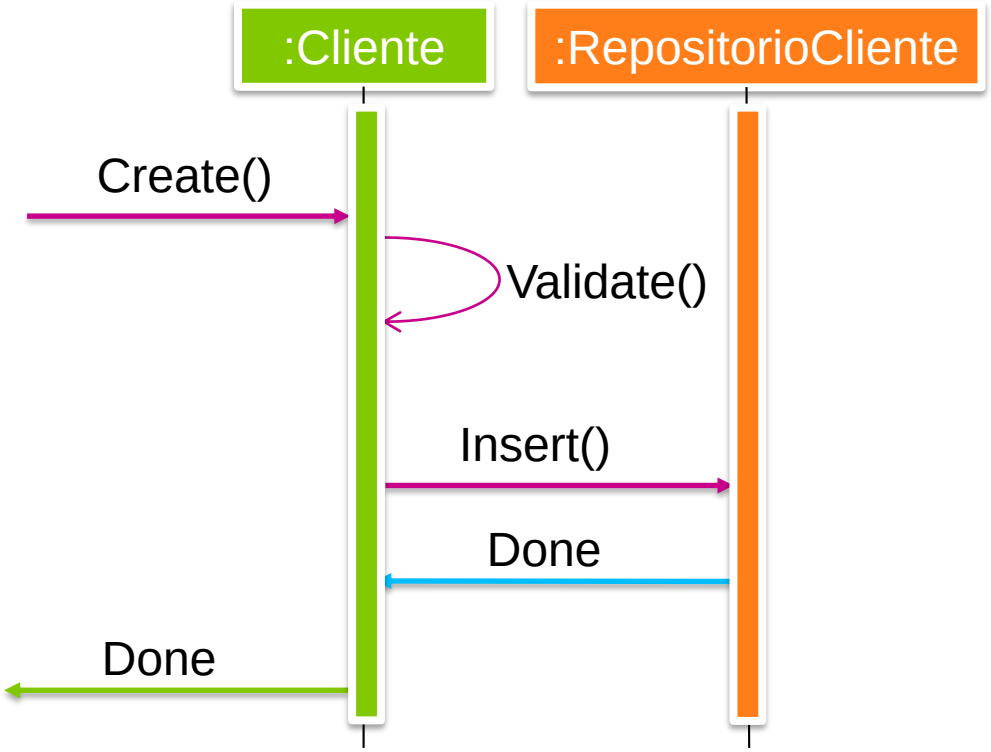
Fundación Jala

innova SANTA CRUZ cainco


# ERD



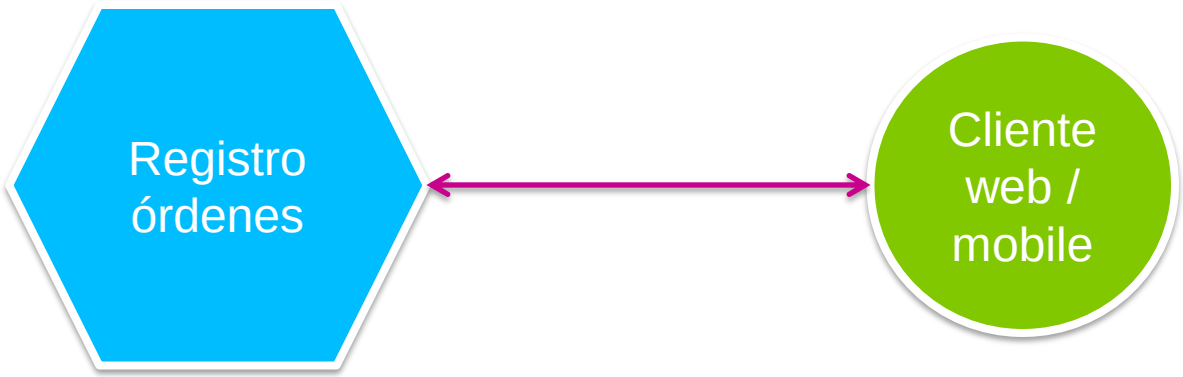

# Static diagram

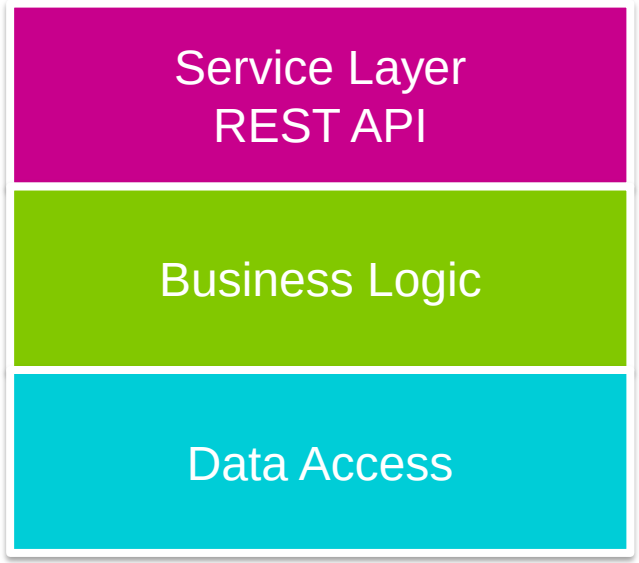



# Sequence diagram

# Funcionalidad

- Es necesario crear una orden que registre el pedido, la orden debe ser asociada a un cliente.

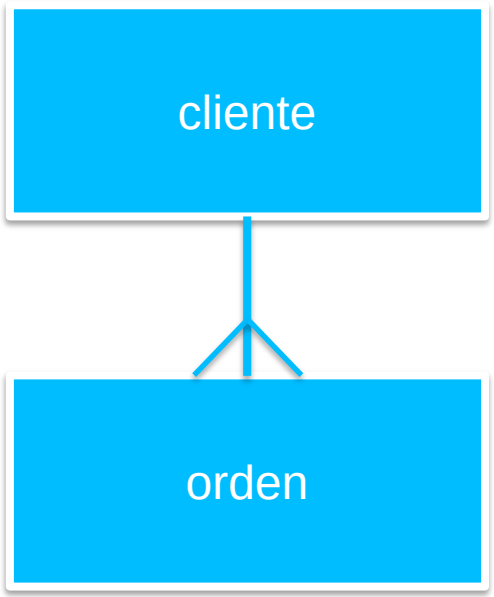

Registro
órdenes
Cliente
web /
mobile



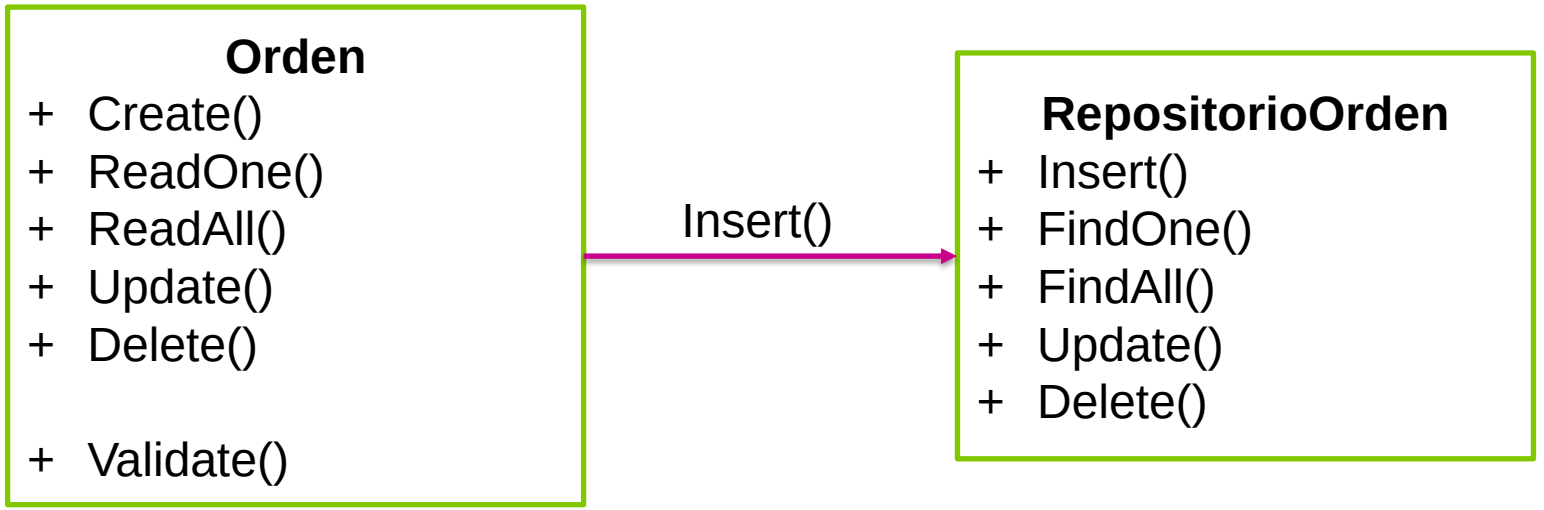

**Recursos**: usuarios, órdenes

**Objetos**: Usuario, Orden

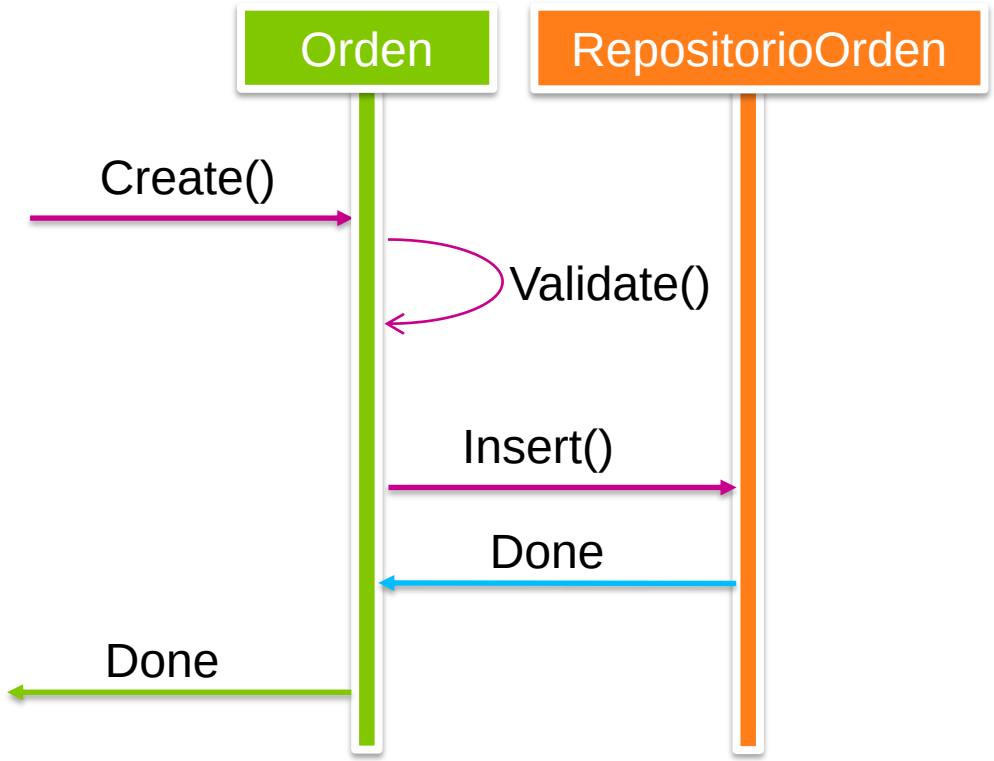
**Tabla/Colección**: Usuario(s), Ordenes




# ERD




# Static diagram

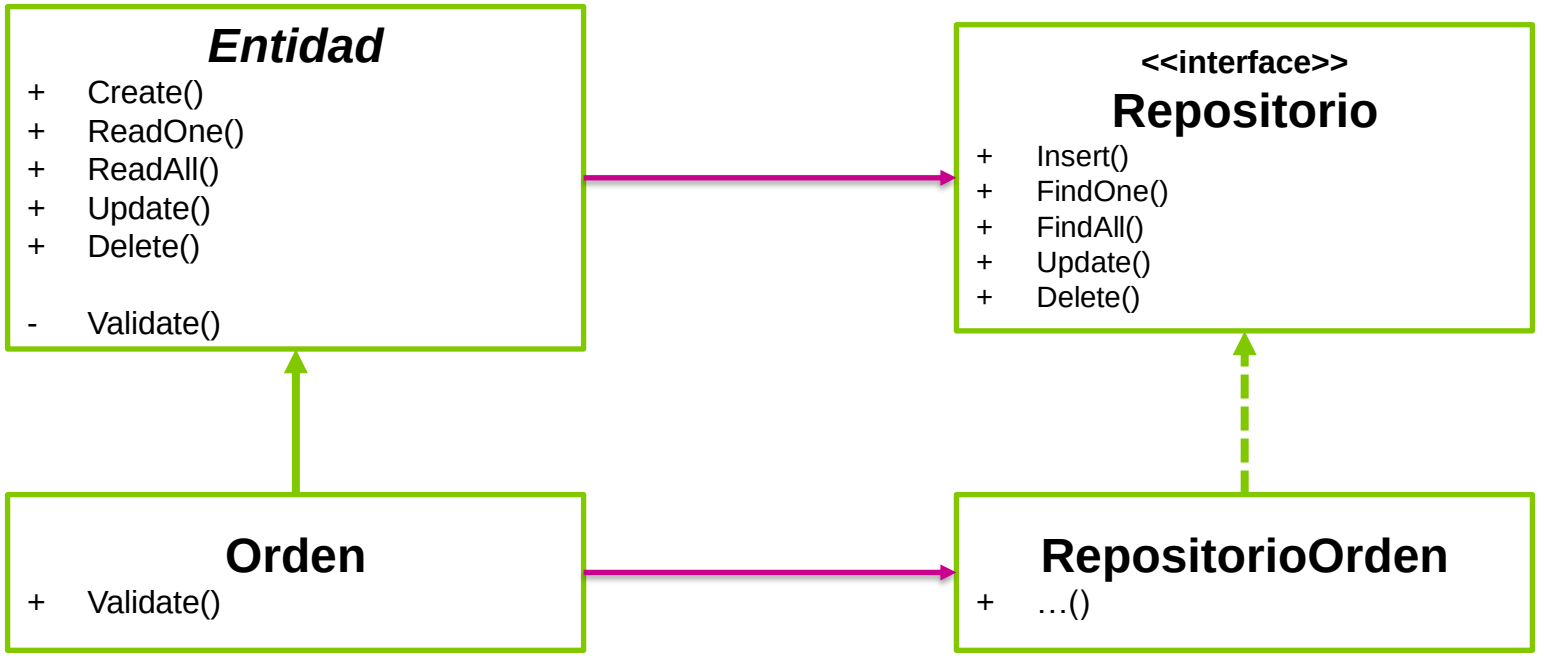



# Sequence diagram

# Lecciones aprendidas

- Las implementaciones para Cliente y Orden se parecen mucho.
- Probablemente el validador cambie, y también el repositorio.
- Es posible hacer algunas abstracciones, siguiendo el principio de Open / Closed.

# Manos a la obra…

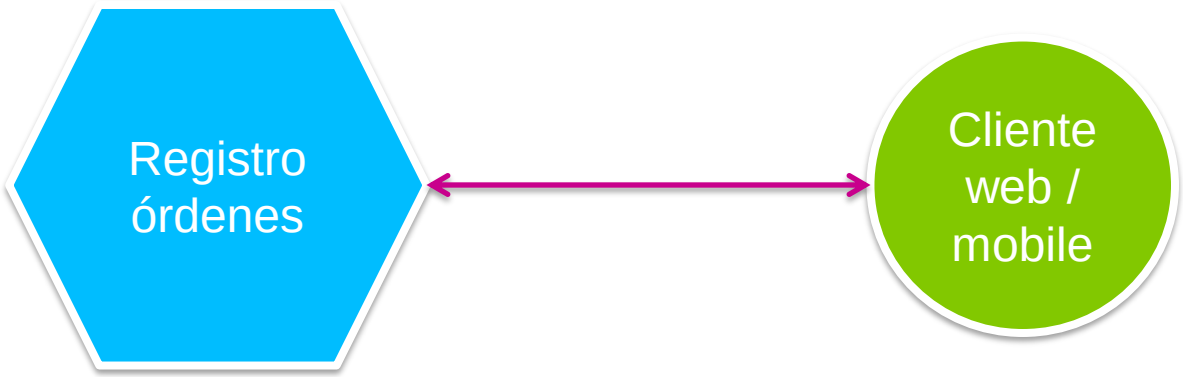


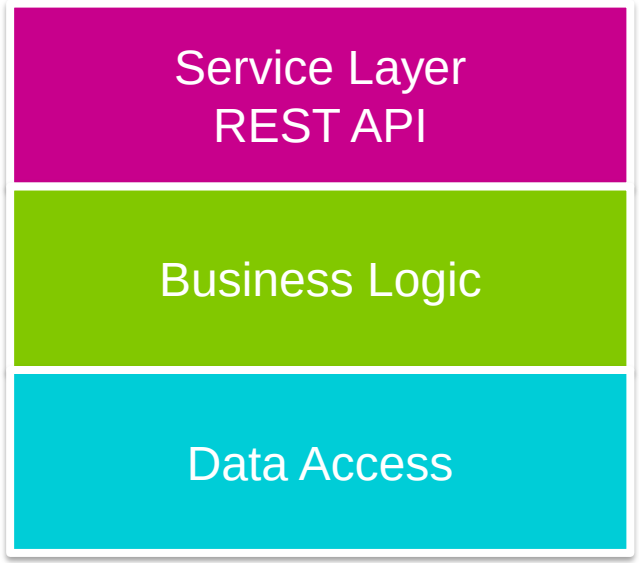

# Static diagram

```
class Entidad {
    constructor() {
        this.repositorio = null;
    }
    create(dto) {
        this.Validate(dto);
        this.repositorio.insert(dto);
    }
    readOne(id);
    readAll(pageInfo);
    update(id, dto);
    delete(id);
    validate(dto);
}

class Orden extends Entidad {
    constructor(repositorio) {
        this.repositorio = repositorio;
    }

    validate(dto) {
        // some logic to validate.
        // if the DTO does not meet the constraints
        throw new Error("Error de validación");
    }
}
```

# Funcionalidad

- Cuando se crea una orden, se crea un ticket para los encargados de los pedidos.


Registro
órdenes
Cliente
web /
mobile

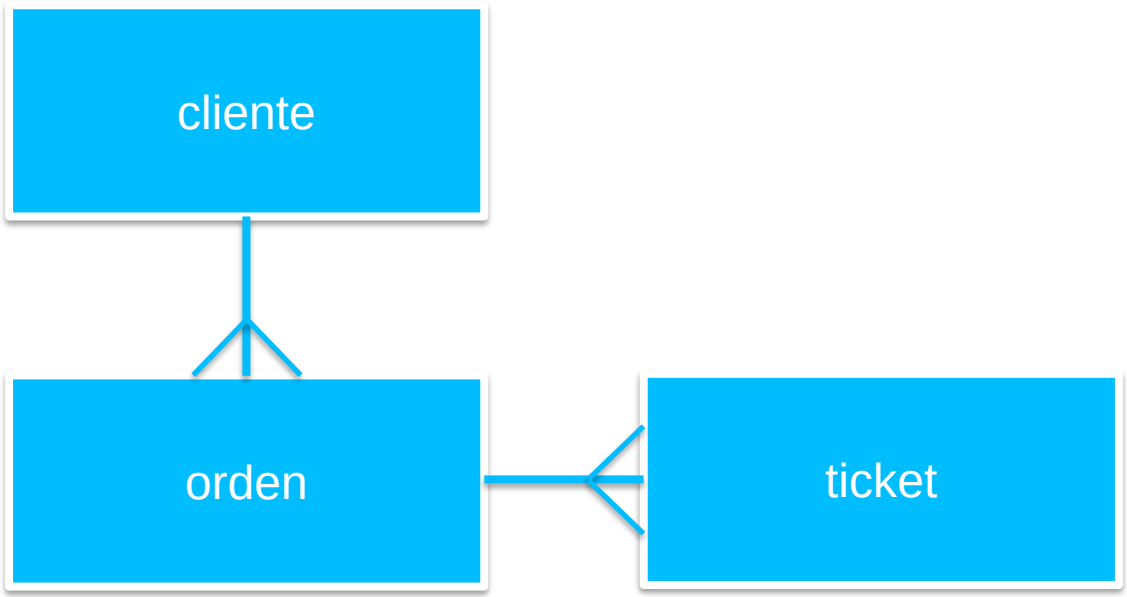



**Recursos**: usuarios, órdenes, tickets

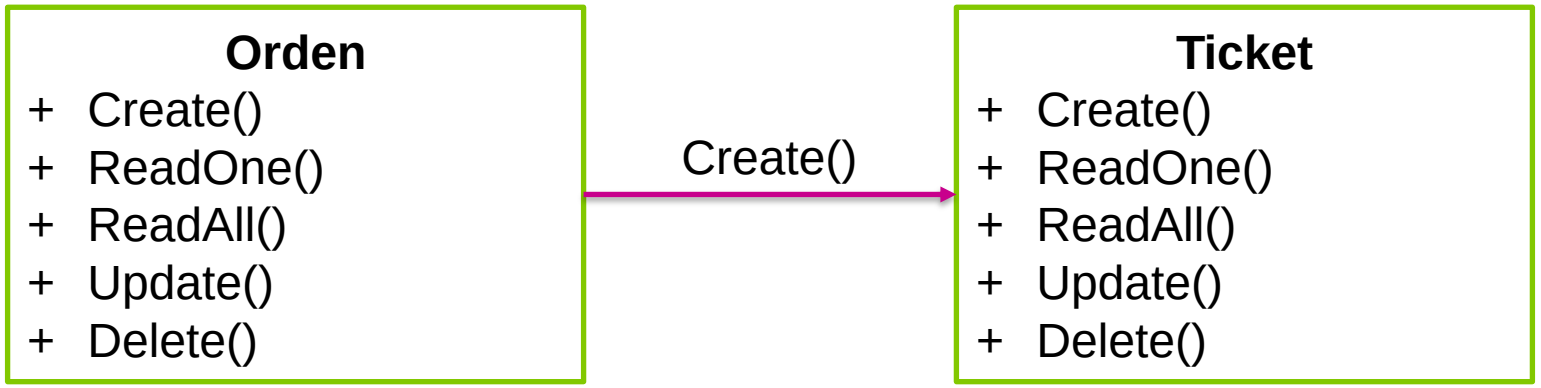
**Objetos**: Usuario, Orden, Ticket

**Tabla/Colección**: Usuario(s), Ordenes, Tickets

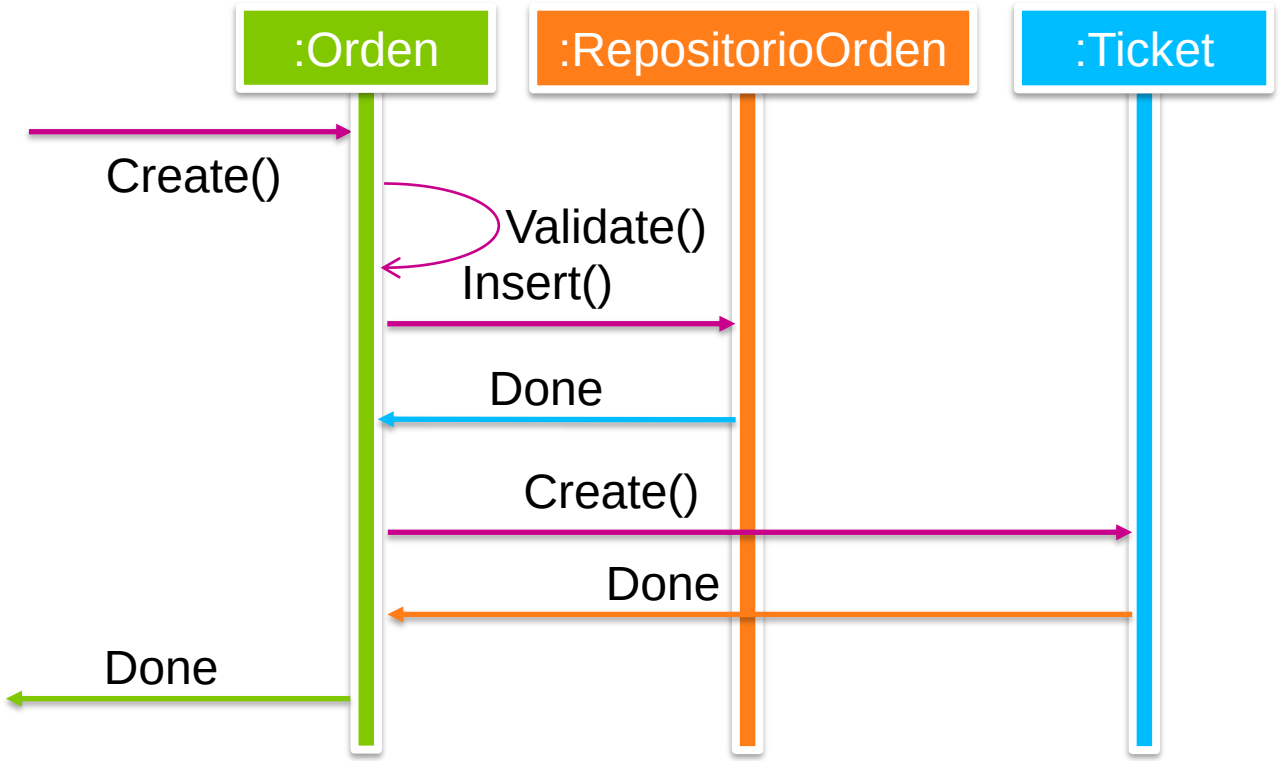
# Lo que un novato haría…




# ERD




# Static diagram




# Sequence diagram

```
class Entidad {
    constructor() {
        this.repositorio = null;
    }
    create(dto) {
        this.Validate(dto);
        this.repositorio.insert(dto);
    }
    // ...
}

class Orden extends Entidad {
    constructor(repositorio, objetoTicket) {
        this.repositorio = repositorio;
        this.objetoTicket = objetoTicket;
    }

    create(dto) {
        super.create(dto);
        const dtoTicket = {};
        this.objetoTicket.create(dtoTicket);
    }

    validate(dto);
}
```


# Principios útiles

# Cuál es el problema?

```
1   class Entidad {
2       constructor() {
3           this.repositorio = null;
4       }
5       create(dto) {
6           this.Validate(dto);
7           this.repositorio.insert(dto);
8       }
9       // ...
10  }
11
12  class Orden extends Entidad {
13      constructor(repositorio, objetoTicket) {
14          this.repositorio = repositorio;
15          this.objetoTicket = objetoTicket;
16      }
17
18      create(dto) {
19          super.create(dto);
20          const dtoTicket = {};
21          this.objetoTicket.create(dtoTicket);
22      }
23
24      validate(dto);
25  }
```

La orden conoce al ticket.


OPEN CLOSED PRINCIPLE
Open Chest Surgery Is Not Needed When Putting On A Coat

# Lecciones aprendidas

- El objeto Orden está fuertemente acoplado al objeto Ticket.
- Esto provoca baja cohesión en la funcionalidad del objeto Orden.
- Que pasaría si en una futura iteración ya no queremos crear el ticket después de la orden?.

# Fundamental theorem of software engineering

"We can solve any problem by introducing an extra level of indirection.“

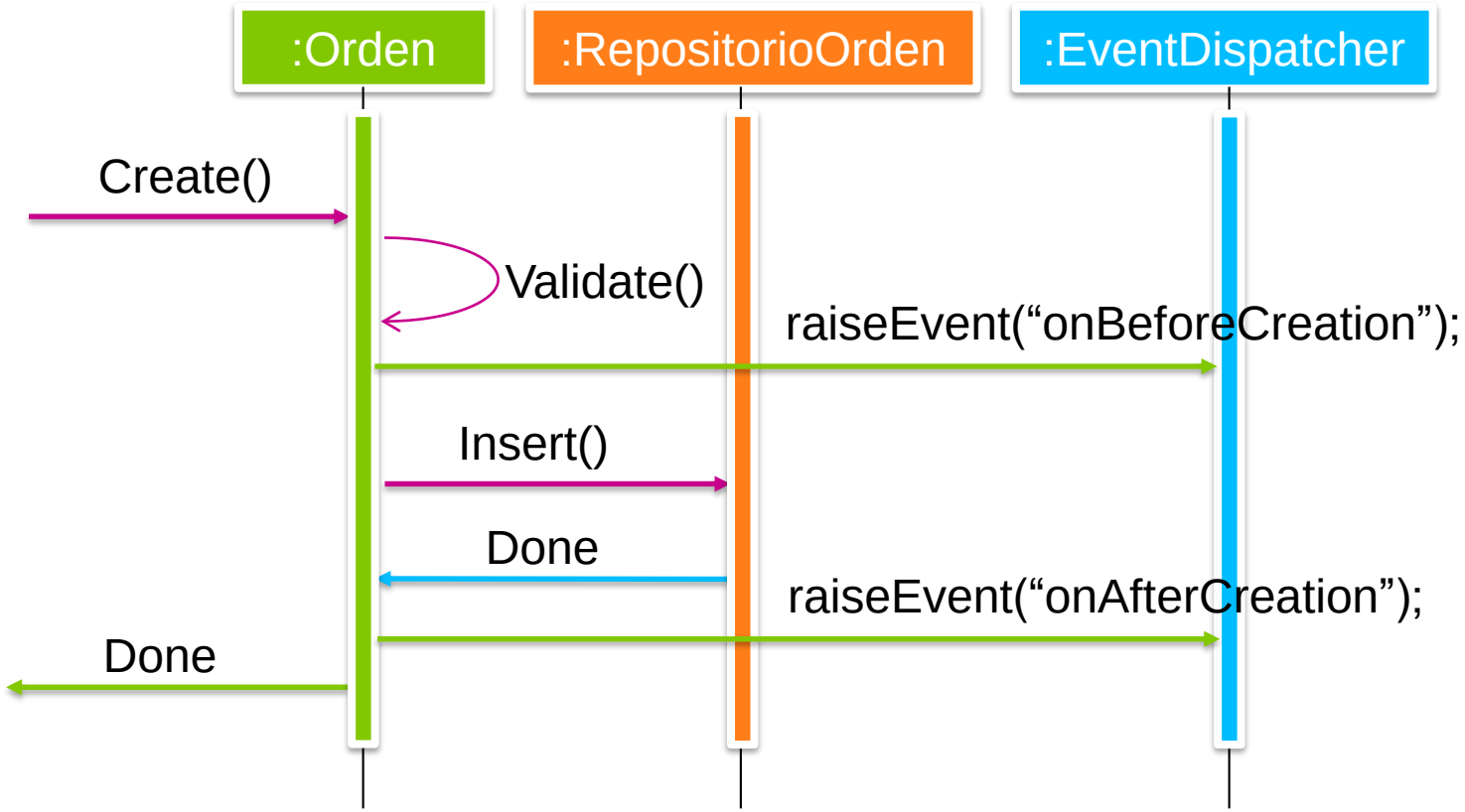
https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering

Fundación Jala

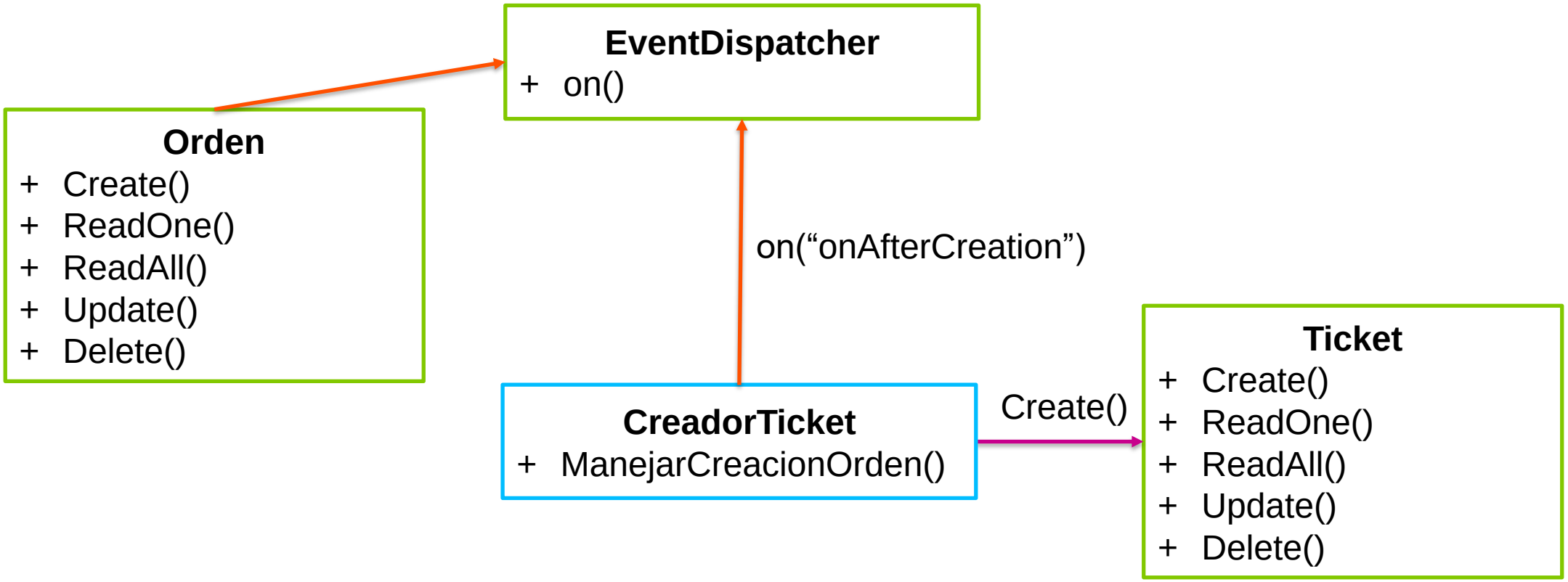
innova SANTA CRUZ cainco

# Estrategia de solución.

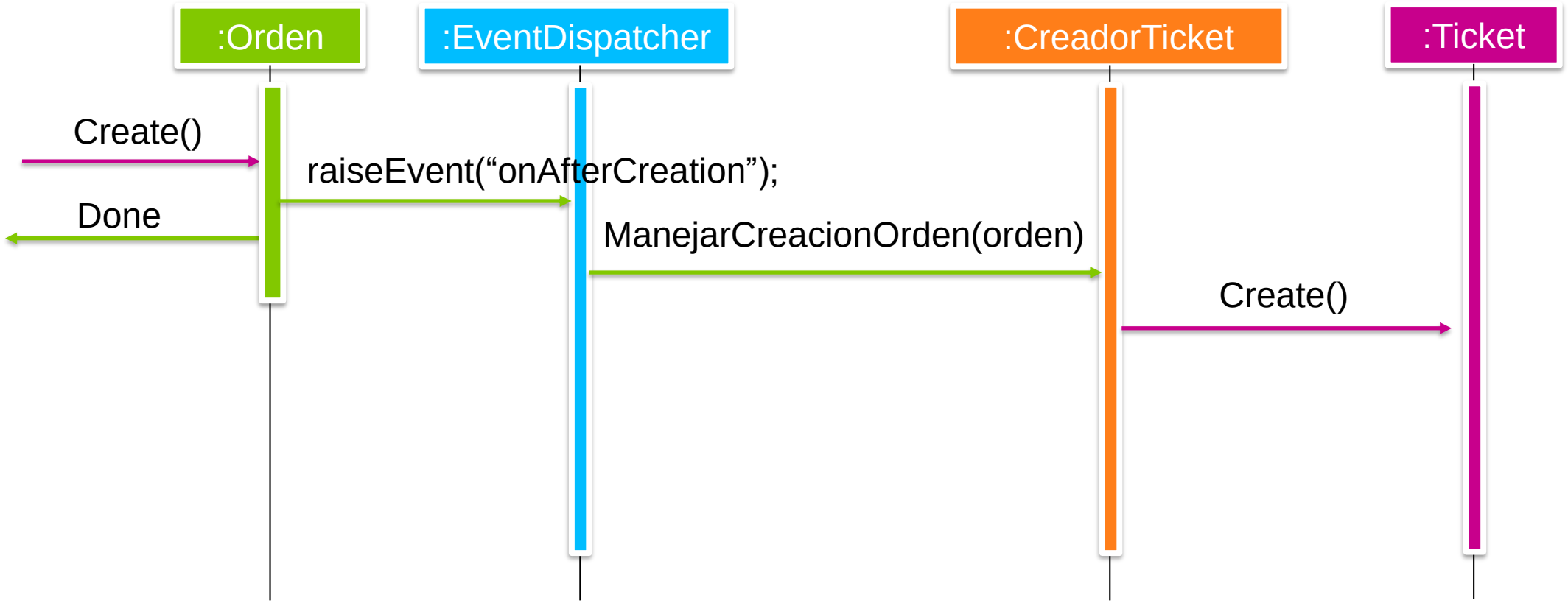
- Necesitamos evitar que el objeto Orden conozca directamente al objeto Ticket.
- Para eso necesitamos una conexión débil que pueda ser establecida o quitada sin modificar los objetos independientes.
- Un mecanismo que nos permite tener este tipo de relaciones son los eventos.




# Pipeline – Creación de Orden



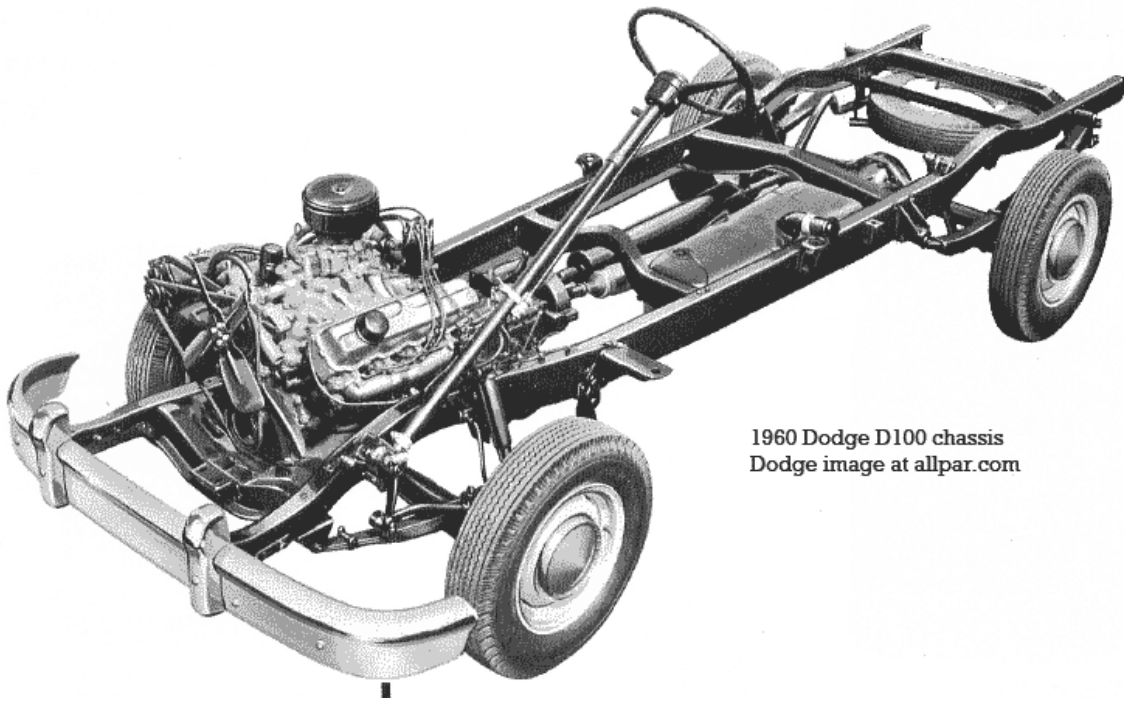

# Static diagram - Eventos

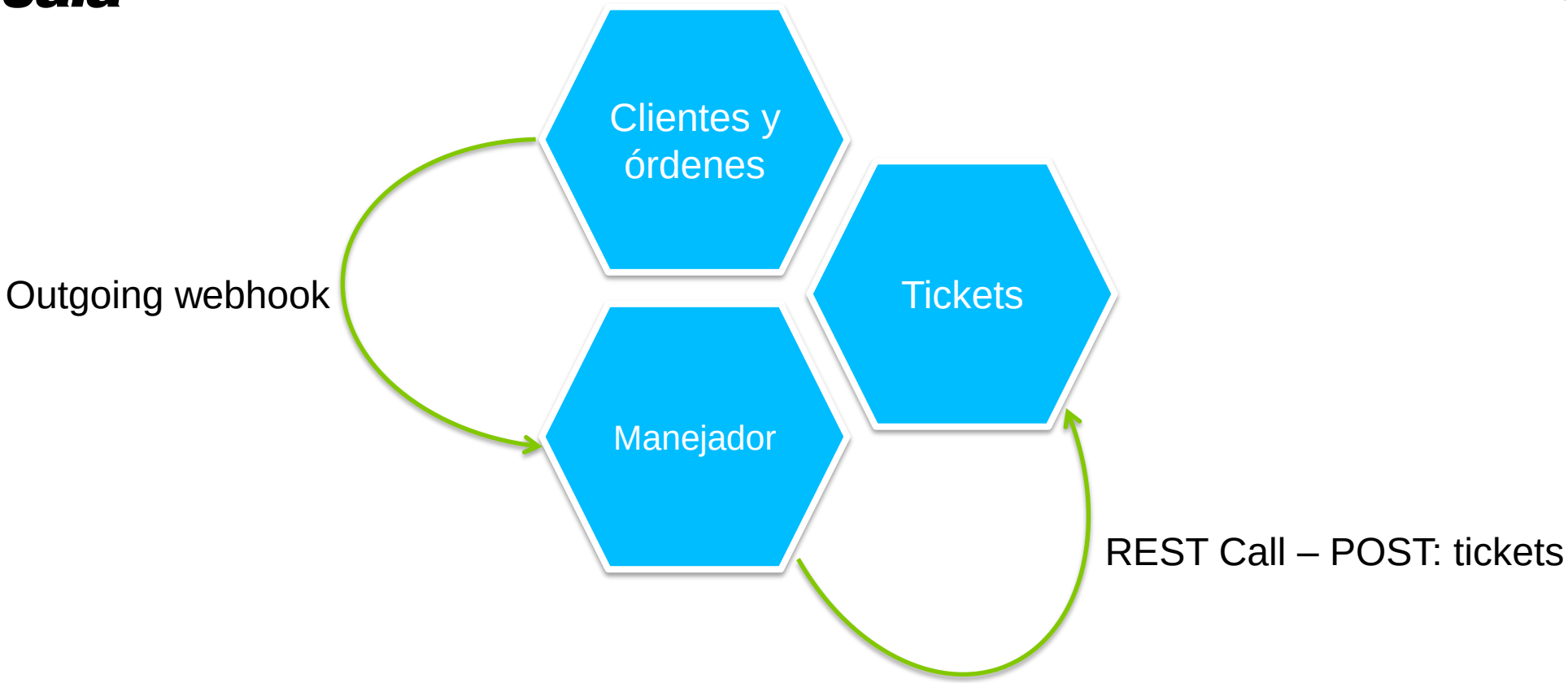



# Pipeline – Creación de Ticket

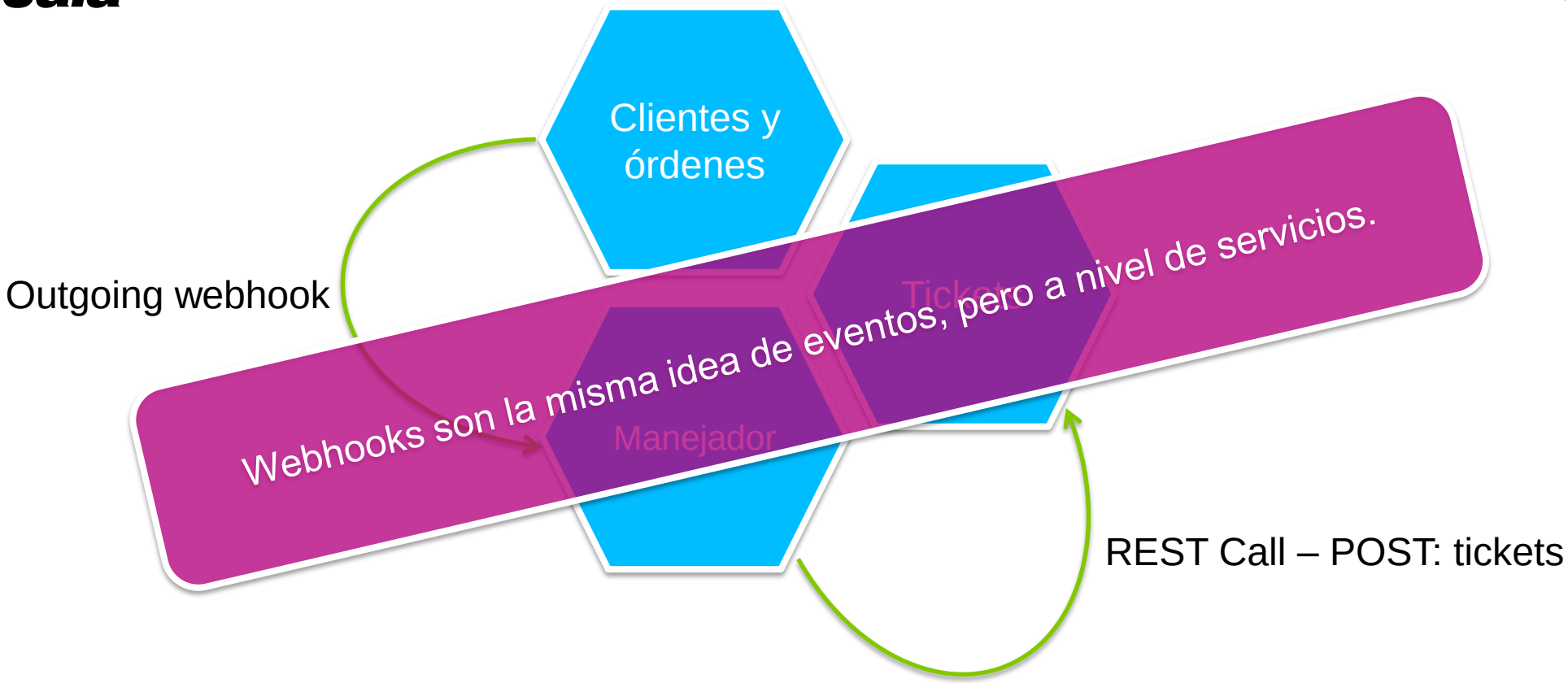
```
const EventEmitter = require('events');
class Entidad extends EventEmitter{
    constructor() {
        this.repositorio = null;
    }
    create(dto) {
        this.Validate(dto);
        this.emit('onBeforeCreation', dto);
        const created = this.repositorio.insert(dto);
        this.emit('onAfterCreation', created)
    }
    // ...
}
class Orden extends Entidad {
    constructor(repositorio) {
        this.repositorio = repositorio;
    }

    validate(dto);
}
class Ticket extends Entidad {
    constructor(repositorio) {
        this.repositorio = repositorio;
    }

    validate(dto);
}
class CreadorTicket {
    constructor(objetoOrden, objetoTicket) {
        this.objetoTicket = objetoTicket;
        objetoOrden.on('onAfterCreation', this.manejarCreacionOrden);
    }
    manejarCreacionOrden(objetoOrden) {
        const ticket = {
            orden: objetoOrden
            //, ...
        };
        this.objetoTicket.create(ticket)
    }
}
```

- La infraestructura de Entidad y Repositorio son la base para la creación de los objetos.
- Los eventos permiten relacionar las entidades sin que se conozcan directamente.






# La misma idea a nivel de servicio

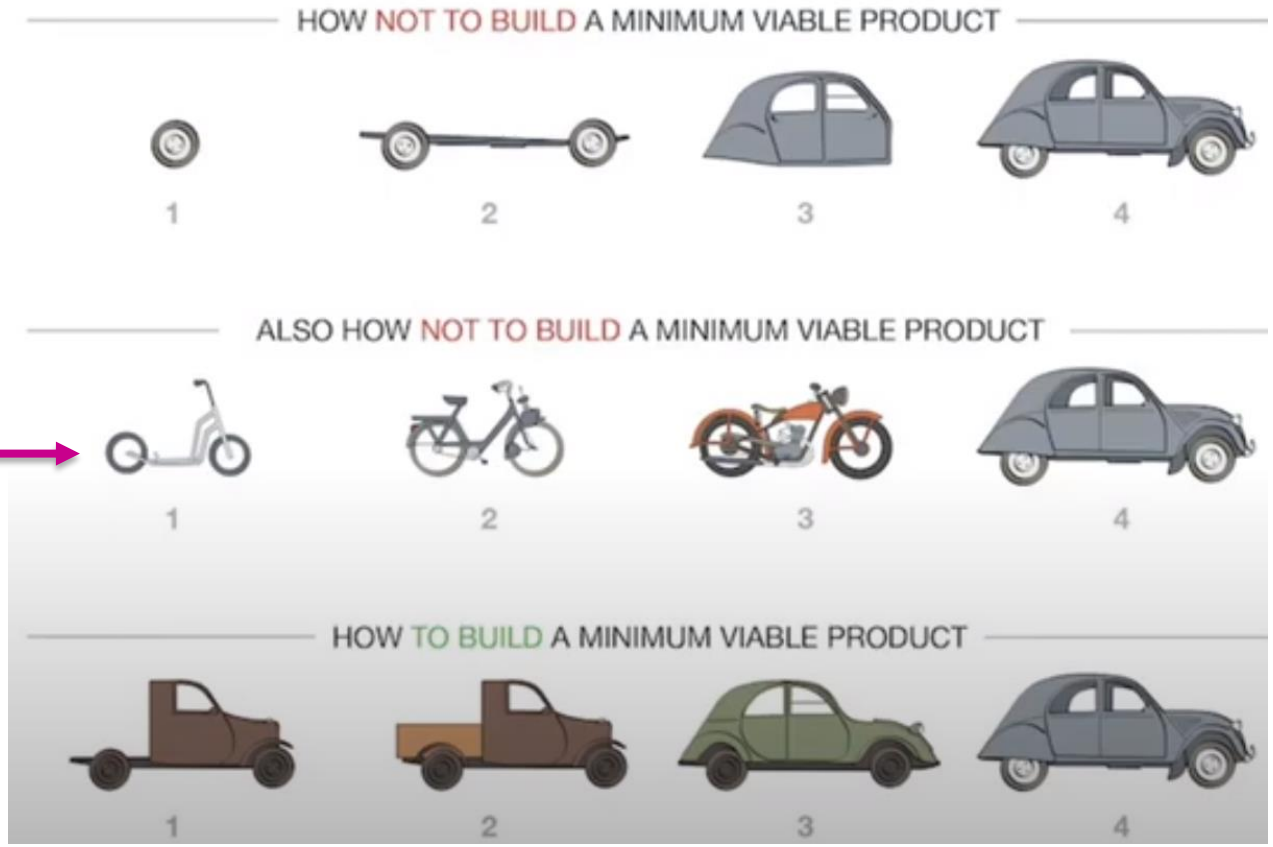



# La misma idea a nivel de servicio

# Deuda técnica

# Iterativo e incremental

Con este enfoque se tienen que hacer muchos cambios entre las iteraciones.



# Deuda técnica

- **La deuda técnica** (también conocida como **deuda de diseño** o **deuda de código**, pero también puede relacionarse con otros esfuerzos técnicos).
- Refleja el costo implícito del retrabajo adicional causado por elegir una solución fácil en lugar de utilizar un enfoque que llevaría más tiempo en su desarrollo e implementación.

# Deuda técnica

- Cuando se acumula mucha deuda técnica, es posible que la entrega de valor se ralentice en etapas posteriores en el desarrollo.
- El PO debe poder equilibrar el trabajo en nuevos PBIs y la resolución de la deuda técnica.

# Resumen

- El desarrollo incremental requiere una base sólida para ir incrementando funcionalidad.
- Es necesario conocer a profundidad los principios de diseño, y estudiar las soluciones ya existentes.
- En todo momento se debe evaluar si nuestro software siguen respetando los principios SOLID, aunque no sea útil en ese momento.
- Se debe evitar tener mucha deuda técnica para poder seguir entregando valor en el producto en las iteraciones futuras.