# Tema 2
# Domain-Driven Design

Modelado y diseño de software con DDD

PDS - Procesos de Desarrollo de Software 2025/26

# Contenidos

1. Objetivo de DDD.
2. Lenguaje ubicuo.
3. Modelado de software con DDD.
4. Eventos de dominio.
5. Caso de estudio.

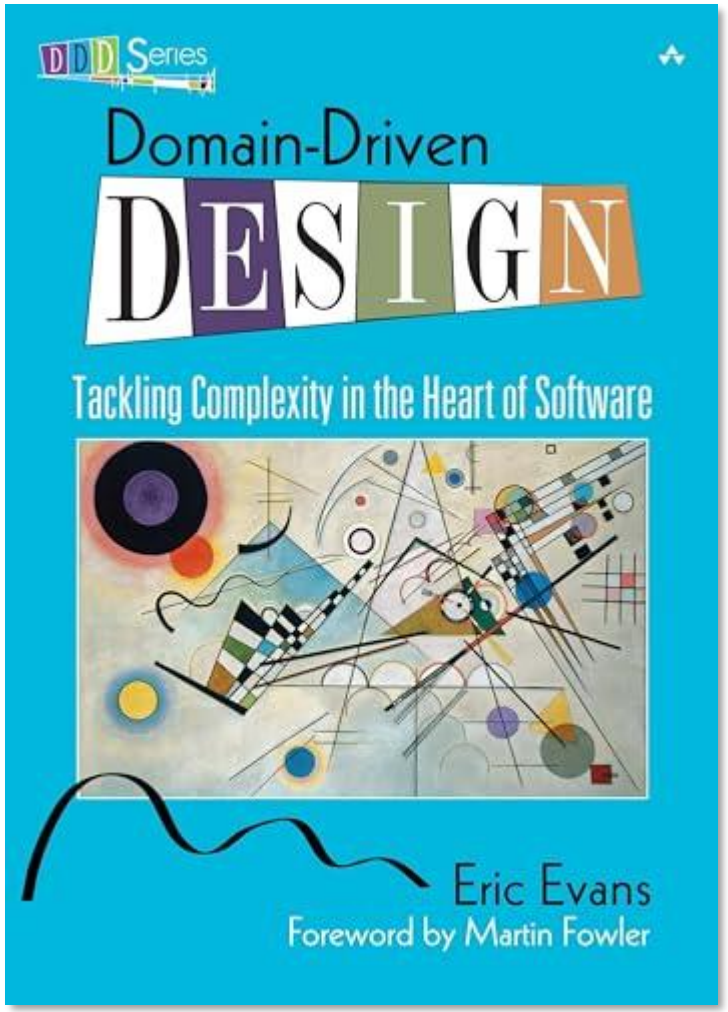
# 1. Objetivo de DDD

¿Qué es y para qué sirve el DDD?

# 1. DDD – ¿Qué es?

- Enfoque para el desarrollo de software creado por Eric Evans.
  - Creado en 2003.

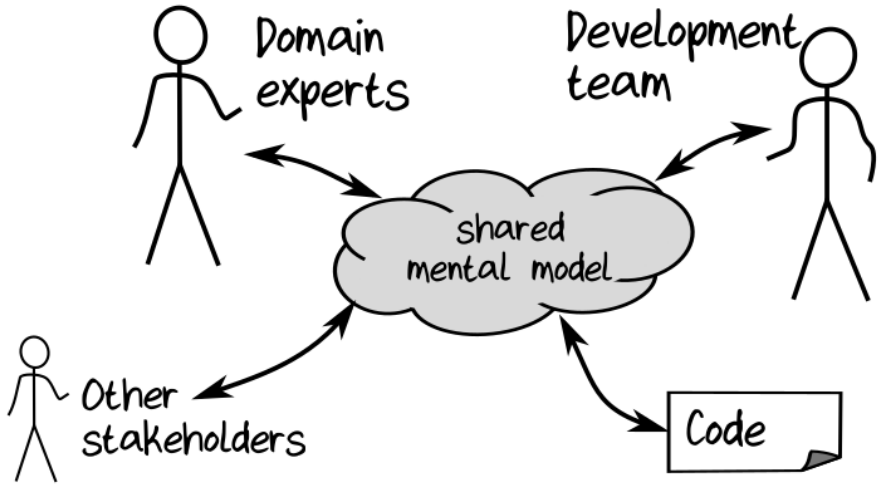
- Dos objetivos básicos:
  - Comprender el dominio del problema.
  - El diseño del software debe reflejar fielmente ese dominio.



# 1. DDD – Objetivo

## Problema

- Brecha entre los expertos del dominio y los desarrolladores.
- Expertos del dominio y desarrolladores *hablan un idioma diferente.*




## Solución

- Modelo compartido entre expertos de dominio y desarrolladores.
- Diseño del software muy cercano al modelo.
- Diseño reflejado directamente en la implementación.

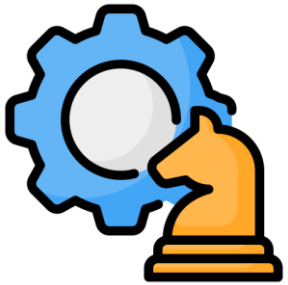
*La “verdad” está en el código*

# 1. DDD – Diseño estratégico vs. táctico

## DDD tiene dos partes

### Diseño estratégico

- Trata de responder ¿qué construir y por qué?
- Estudio del dominio – División en subdominios



### Diseño táctico

- Trata de responder ¿cómo construirlo?
- Patrones de implementación.

* En este curso nos centramos en el Diseño Táctico

# 1. DDD – Dominio - ¿Qué es?

El **dominio** es el área de conocimiento, actividad o negocio específico para el cual se está desarrollando el software.

## Características

- Hace referencia al “qué” del sistema, no al “cómo”
- No dictado por decisiones tecnológicas.
  - Técnicas como “Arquitectura hexagonal” o “Arquitectura limpia” van en esta dirección

# 1. DDD – Ejemplos de dominios


## Juego de fútbol

- Reglas del juego.
- Estados del partido faltas, prórrogas, penalties
- IA, física, cansancio, lesiones


## Plataforma de streaming

- Catálogos y licencias por país.
- Idioma, restricciones por edad.
- Suscripciones, perfiles, recomendaciones.


## Market place

- Millones de productos y proveedores.
- Logística global.
- Campañas, ofertas.
- Políticas y restricciones legales.

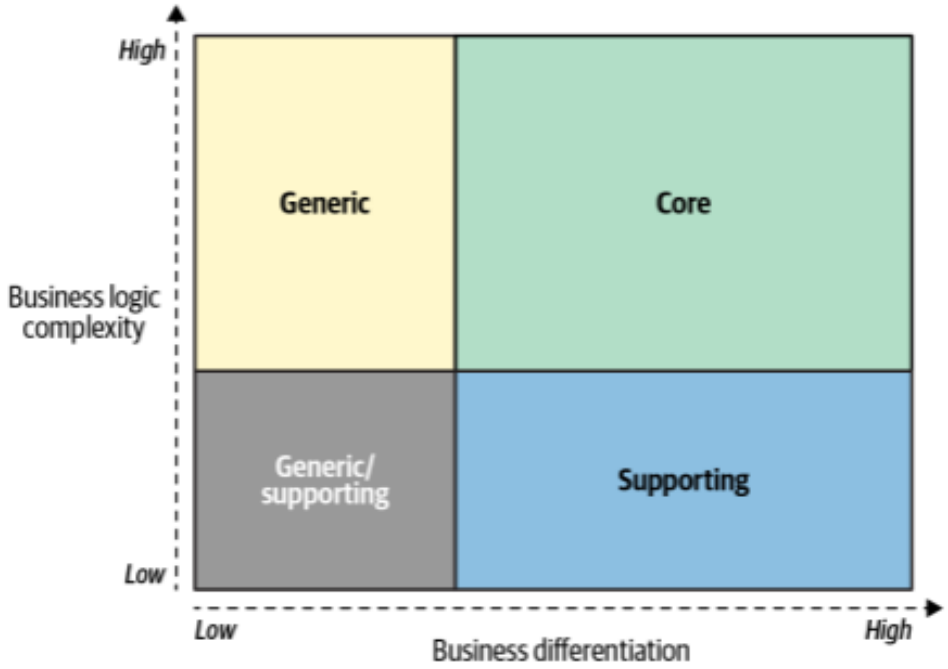
# 1. DDD – Subdominio

El **subdominio** es una parte del negocio que agrupa capacidades y reglas relacionadas.

- Distinguir entre dominios sirve para priorizar esfuerzo y decidir dónde poner el talento.

## Tres tipos:

- Dominio principal (Core domain)
- Subdominio de soporte (Supporting subdomain)
- Subdominio genérico (Generic subdomain)



# 1. DDD – Subdominio – Dominio principal

## Dominio principal (Core Domain)

Área principal. Lo que distingue al negocio.

- Mejor modelado.
- Mejores devs.
- Más cuidado con el lenguaje ubicuo.


Simulación del partido y experiencia de juego


Gestión y consumo de contenido audiovisual personalizado


Gestión dinámica de precios, ofertas y matching entre oferta y demanda

# 1. DDD – Subdominio – Subdominio de soporte

## Dominio de soporte

Necesario para que el dominio principal funcione.

- Modelado suficiente.
- Complejidad controlada.
- Buenas prácticas, sin sobre-ingeniería.


Gestion de ligas, rankings, estadísticas


Moderación, estadísticas, reclamaciones, ….

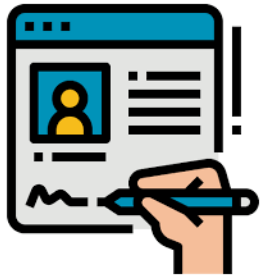

Atención al cliente, gestión de devoluciones

# 1. DDD – Subdominio – Subdominio genérico

## Subdominio genérico

Actividades que son muy comunes.

- No diferencian al negocio; no merece “inventar la rueda”.
- Aquí conviene:
  - reutilizar
  - comprar/usar SaaS
  - librerías maduras



Identidad y acceso

Envíos de emails, push notifications, sms


Logging

Pagos y facturación

# 1. DDD – Subdominios

| Tipo | Ventaja Competitiva | Complejidad | Volatilidad | Implementar | Problema |
|---|---|---|---|---|---|
| Core | Sí | Alta | Alta | Propia | Interesante |
| Genérico | No | Alta | Baja | Reutilizar | Solucionado |
| Soporte | No | Baja | Baja | Propia/Sub. | Obvio |

# 1. DDD – Subdominio – Ejemplo Gigmaster

Gigmaster es una empresa de **venta y distribución de entradas**. Su aplicación móvil analiza las bibliotecas musicales de los usuarios, sus cuentas de servicios de *streaming* y sus perfiles en redes sociales para identificar conciertos cercanos a los que los usuarios podrían estar interesados en asistir.

Los usuarios de Gigmaster son conscientes de su privacidad. Por ello, toda la información personal de los usuarios está cifrada. Además, para garantizar que los gustos de los usuarios no se filtren bajo ninguna circunstancia, el algoritmo de recomendación de la empresa funciona exclusivamente con datos anonimizados.

Para mejorar las recomendaciones de la aplicación, se implementó un nuevo módulo. Este permite a los usuarios registrar conciertos a los que asistieron en el pasado, incluso si las entradas no se compraron a través de Gigmaster.

Ejemplo tomado de “Learning Domain-Driven Design”, pág. 14

# 1. DDD – Dominio – Ejemplo Gigmaster

## Dominio principal

- Recomendar conciertos relevantes usando datos anonimizados sin filtrar gustos.

## Subdominios de soporte

- Venta y distribución de entradas.
- Módulo de conciertos previos.
- Integración con redes sociales, streaming y biblioteca musical.
- Aplicación móvil.

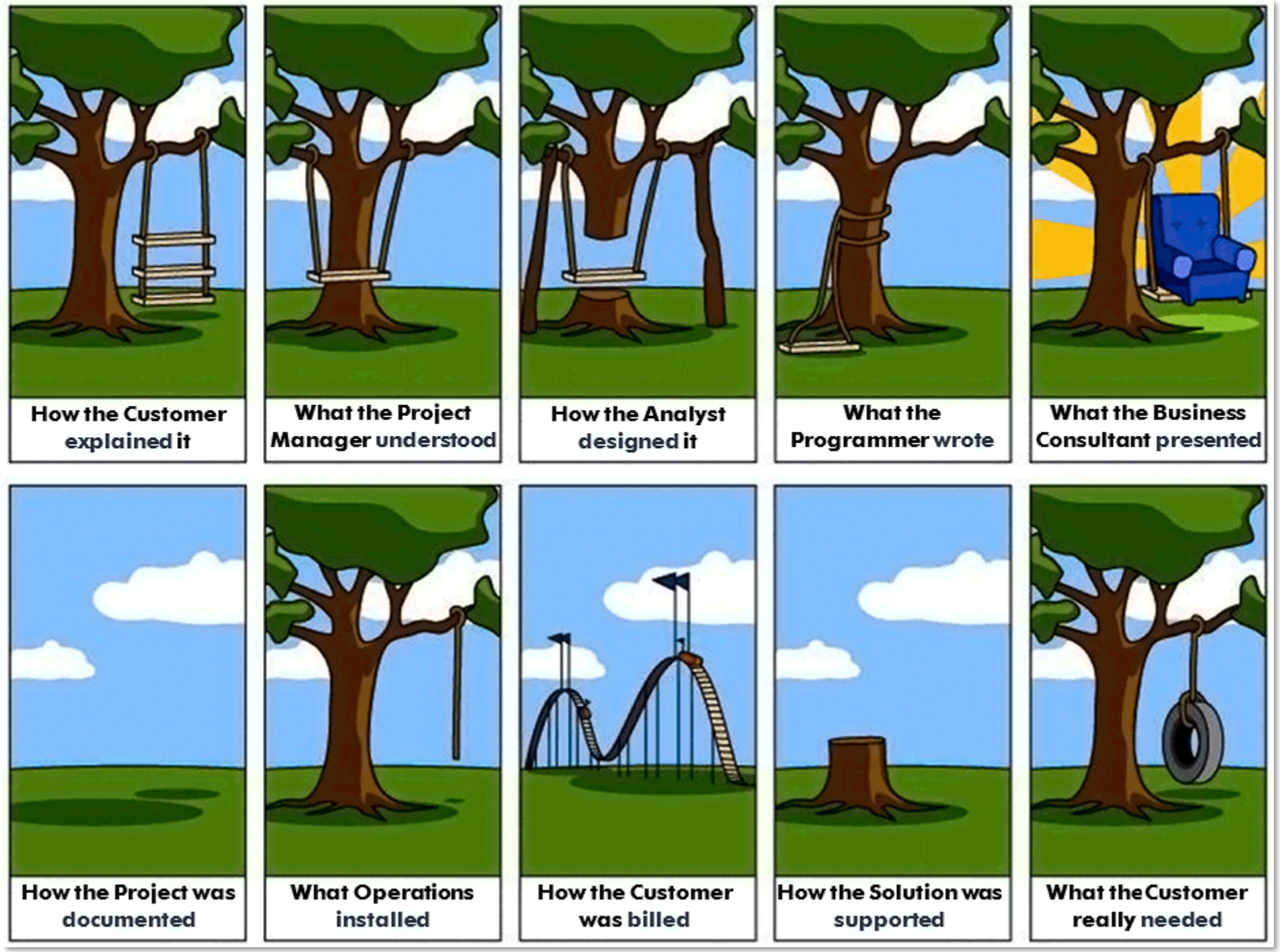
## Subdominios genéricos

- Encriptación
- Autenticación y autorización
- Facturación
- Anonimización de datos.

Ejemplo tomado de “Learning Domain-Driven Design”, pág. 14

# 2. Lenguaje ubicuo

A language to rule them all


How the Customer explained it
What the Project Manager understood
How the Analyst designed it
What the Programmer wrote
What the Business Consultant presented
How the Project was documented
What Operations installed
How the Customer was billed
How the Solution was supported
What the Customer really needed

# 2. Lenguaje ubicuo – Definición

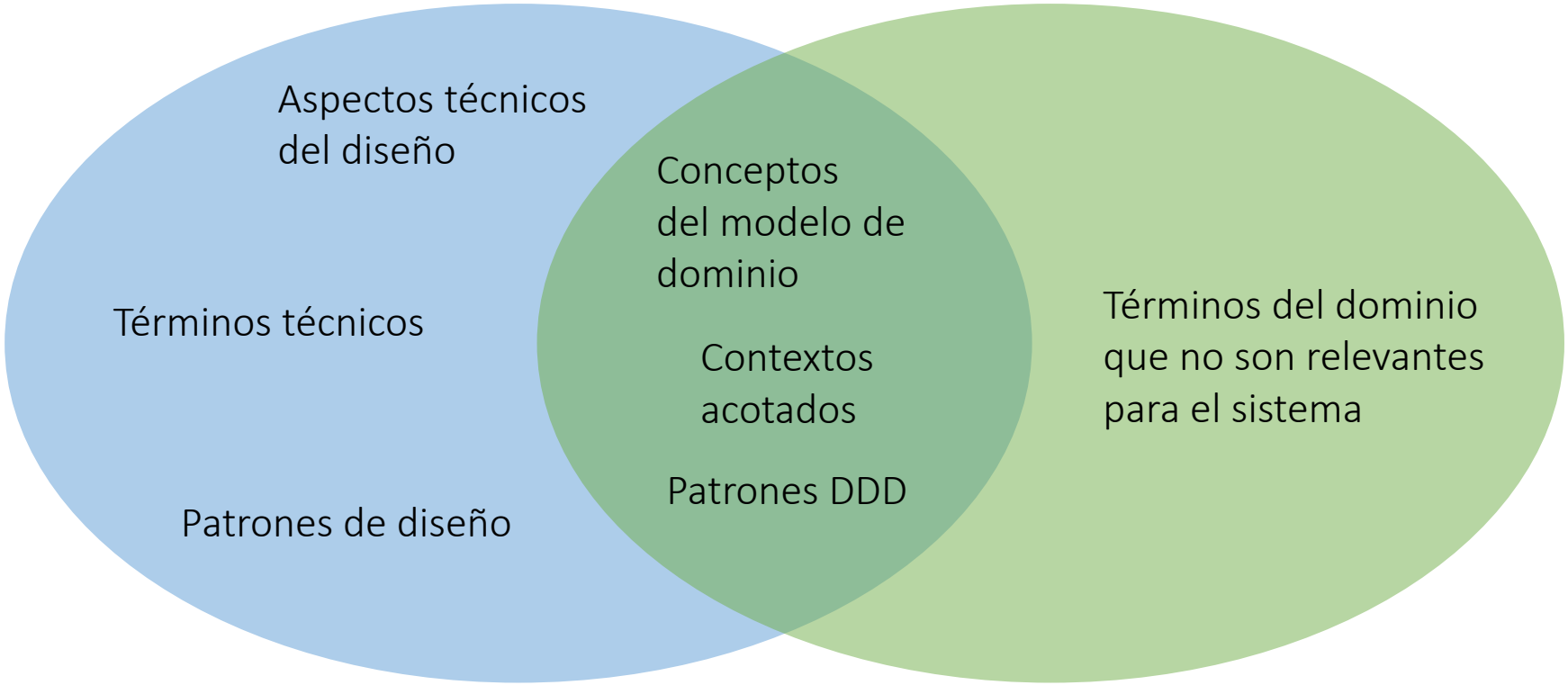
> El **Lenguaje ubicuo** al conjunto de conceptos y vocabulario que se comparte entre todos los **stakeholders** del proyecto.

Define el modelo mental compartido del proyecto.

## Motivación

- Expertos del dominio no entienden el lenguaje técnico.
- Los desarrolladores tienden a *tecnificar* los conceptos.
- El juego del teléfono roto
  hay que traducir de unos a otros los conceptos y se pierde el significado original.

# 2. Lenguaje ubicuo – The sweet spot




El sweet spot es cuando el modelo es lo bastante cercano al negocio para que el experto lo entienda, y lo bastante preciso para que el desarrollador lo implemente.

# 2. Lenguaje ubicuo – ¿Qué incluye?

Representado por un modelo de dominio

## ¿Qué incluye el modelo de dominio?

- Diagramas UML (clases, interacción, objetos)
- Escenarios.
  - Con representaciones visuales especificas del dominio
- Explicaciones textuales.

# 2. Lenguaje ubicuo – ¿Pero qué es un modelo?

Un **modelo** es una simplificación de la realidad, en el sentido de que los detalles irrelevantes se ignoran, pero centrándose en los aspectos que interesan para resolver un problema particular.

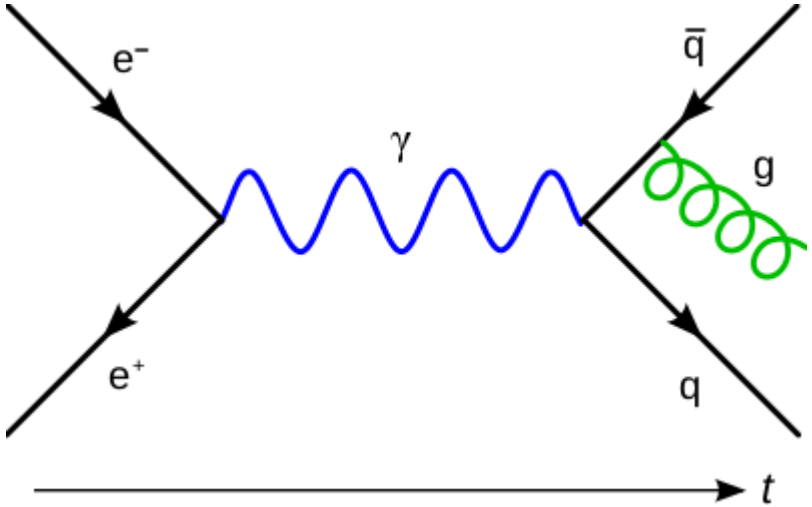
- Dada una tarea, un modelo es útil si:
  - Contiene los detalles que necesitamos.
  - Omite los detalles que son irrelevantes.
- El modelo nos ayudará en ciertas tareas.
  - Pero no será útil para otras.
- Un modelo es una abstracción.


**"All models are wrong, but some are useful"** – George Box

# 2. Lenguaje ubicuo – ¿Pero qué es un modelo?

## **Ejemplo:** Diagrama de Feynmann

- Representación abstracta de la interacción de partículas subatómicas.
- Útil en física de partículas .
- Ayuda a razonar.




*un electrón y un positrón se aniquilan, produciendo un fotón (representado por la onda sinusoidal azul) que se convierte en un par quark-antiquark, después de lo cual el antiquark irradia un gluón representado por la hélice verde).*

* https://es.wikipedia.org/wiki/Diagrama_de_Feynman

# 2. Lenguaje ubicuo – Utilidad de los modelos

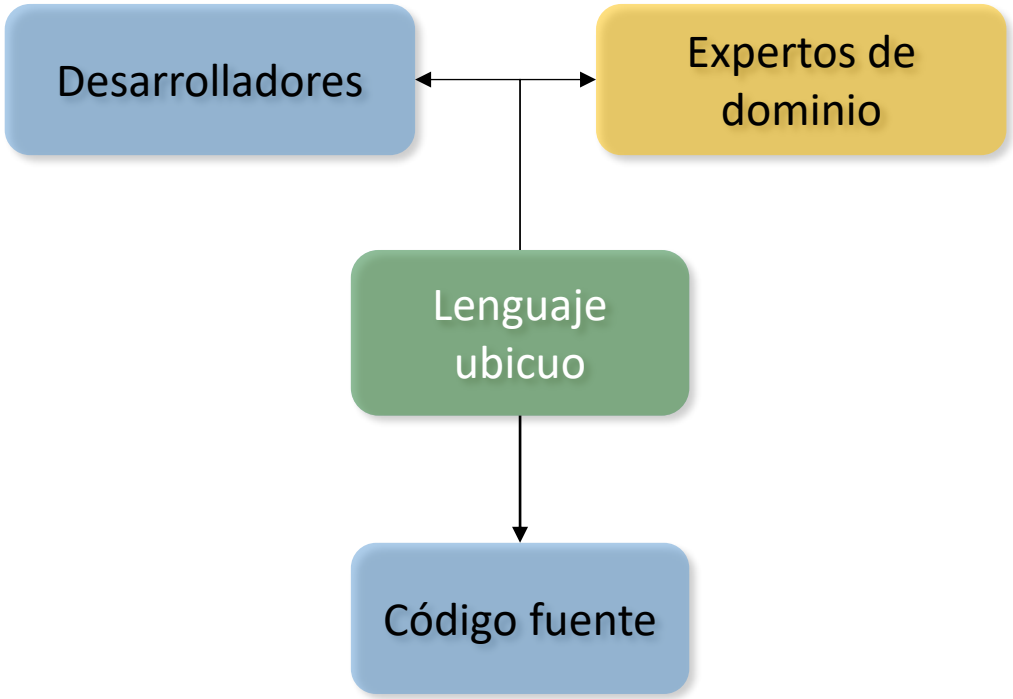
Para describir un sistema se utilizarán varios modelos.

## Propósito de los modelos

- Ayudar a **entender** las características más destacadas del sistema.
- Facilitar la **comunicación** entre los diferentes *stakeholders*.
- Para poder **razonar** acerca de las características del diseño propuesto.
  - ¿Será suficientemente rápido?
  - ¿Cuánto costará?
- Pueden servir como **guía** para la implementación del sistema

# 2. Lenguaje ubicuo – ¿Qué incluye el lenguaje ubicuo?

- Facilita la comunicación entre desarrolladores y expertos de dominio.
- Proporciona nombres para las entidades del código fuente:
  - Clases, métodos, paquetes, APIs, etc.



# 2. Lenguaje ubicuo – Modelado de software

## ¿A qué nivel de abstracción debe estar el modelo de dominio?

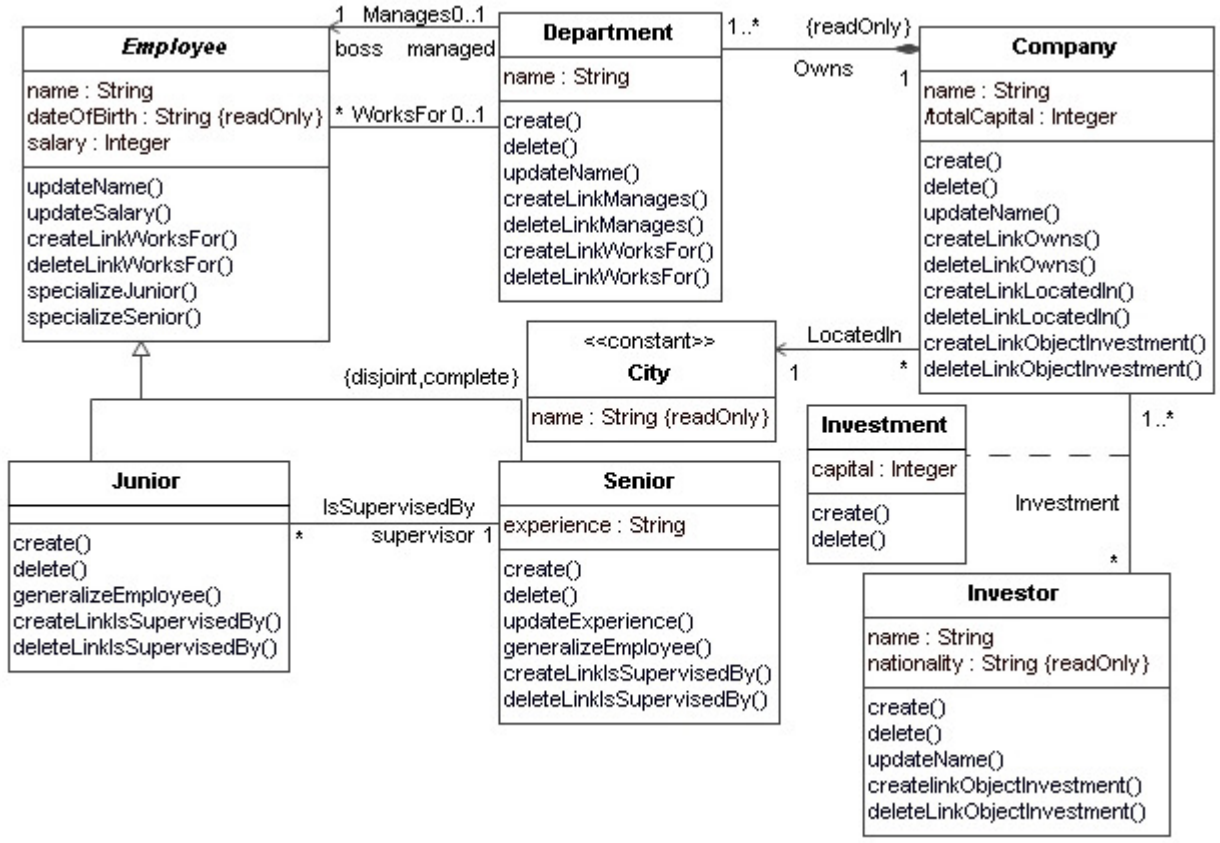
El modelo debe servir tanto para análisis como para diseño

## Lógica

Si el modelo unifica todo y está reflejado directamente en el software entonces no habrá malentendidos.

Cuando en DDD hablamos de modelo no estamos hablando de un modelo conceptual (¡eso es el lenguaje ubicuo!). Por modelo nos referimos a la intersección entre análisis y diseño que está representado en el código.

# 2. Lenguaje ubicuo – ¿Qué no queremos?



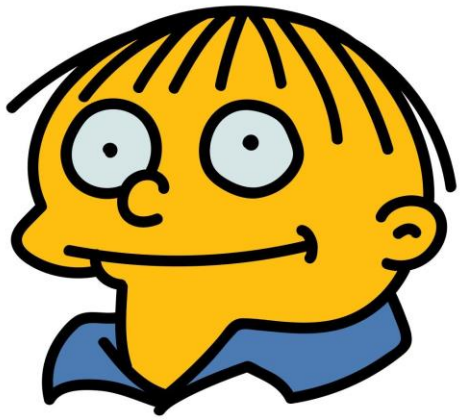

Para esto ya tenemos una sintaxis mucho mejor… ¡el código fuente!

# 2. Lenguaje ubicuo – Modelos ricos vs. anémicos

No todos los modelos representan igual de bien el dominio.

Un modelo anémico:

- Se limitar a transportar datos.
- Tiene únicamente con getters y setters.
- Aporta lo peor de la OO (ej., complejidad en el mapping relacional) sin sus ventajas.



En DDD nos interesa que el modelo refleje el comportamiento del dominio.

# 2. Lenguaje ubicuo – Modelo anémico

```
public class Cuenta {
  private String numero;
  private double saldo;
  ... getters/setters ...
}

public class ServicioCuenta {
  public void retirar(Cuenta cuenta, double importe) {
     checkArgument(importe > 0, "El importe debe ser positivo");
     checkArgument(cuenta.getSaldo() >= importe, "Saldo insuficiente");
     cuenta.setSaldo(cuenta.getSaldo() - importe);
  }
}
```

# 2. Lenguaje ubicuo – Bounded context

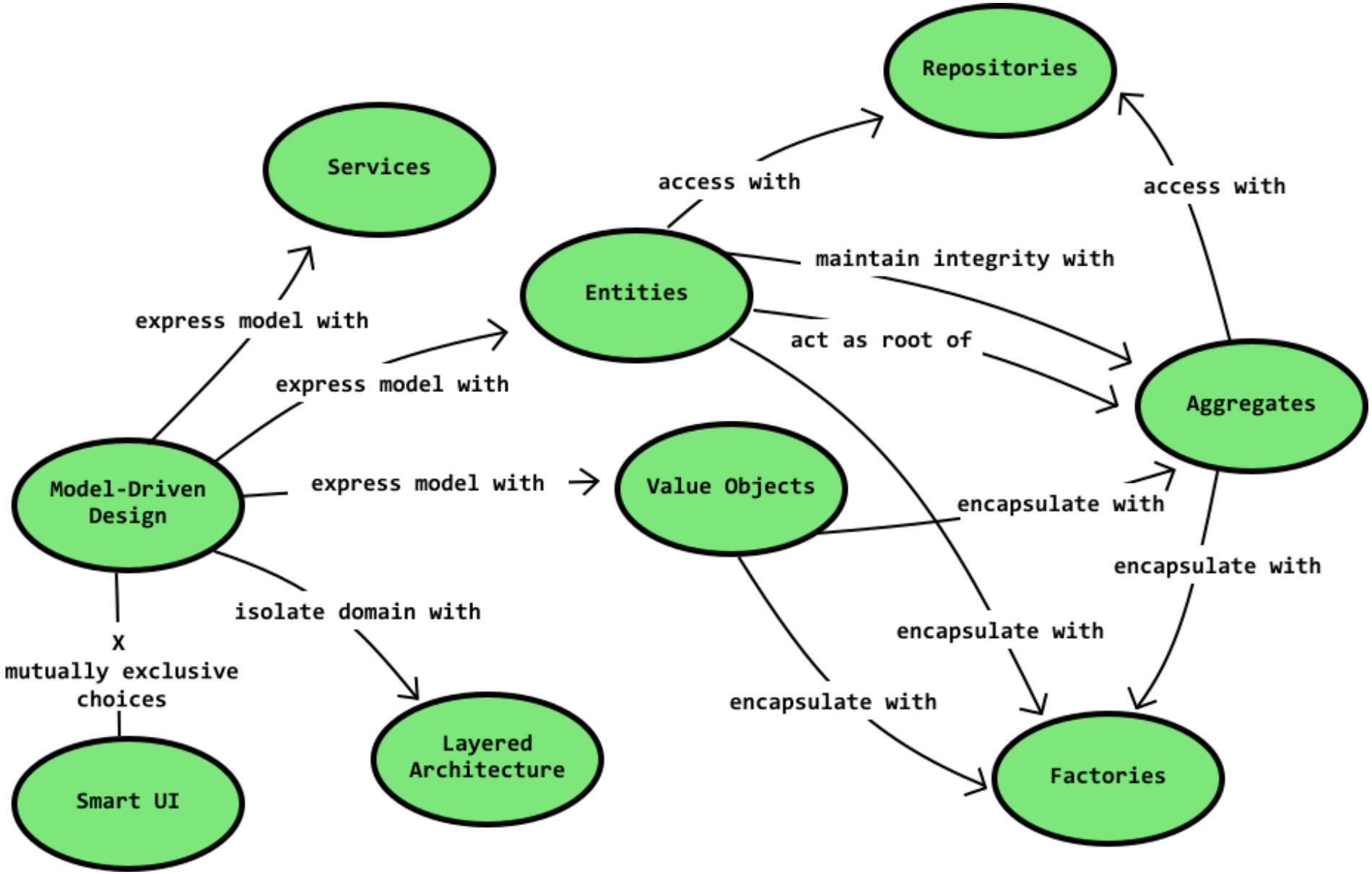
- Un **bounded context** delimita la aplicabilidad del lenguaje ubicuo
  - Los términos del modelo tienen un significado dentro de ese contexto
  - Se diseña el software para ese contexto.
  - Los módulos se comunicarán *mapeando* significados
- Ejemplo – Tienda online
  - Bounded context: Pedidos
    - Entidad **Pedido**
    - Una compra realizada por un cliente
    - Estados: pagada, cancelada
  - Bounded context: Envíos y logística
    - Entidad **Pedido**
    - Una petición para enviar productos
    - Estado: empaquetado, enviado, entregado

* En este curso asumiremos un solo *bounded context* para simplificar

# 3. Modelado de software

Un modelo representado en código

# 3. Modelado de software – Elementos de DDD



# 3. Modelado de software – Entidades

Una **entidad** es un objeto que se define principalmente por su identidad, en lugar de sus atributos o propiedades.

## Ejemplos de entidades:

- Un usuario de una aplicación.
- Una empresa en un CRM.
- Un mensaje en un foro.

**Una entidad se distingue por su identidad, no por sus atributos.**

“Juan López Hernández”
DNI “28520584C”

≠

“Juan López Hernández”
DNI “11320584C”

# 3. Modelado de software – Entidades – Identificadores

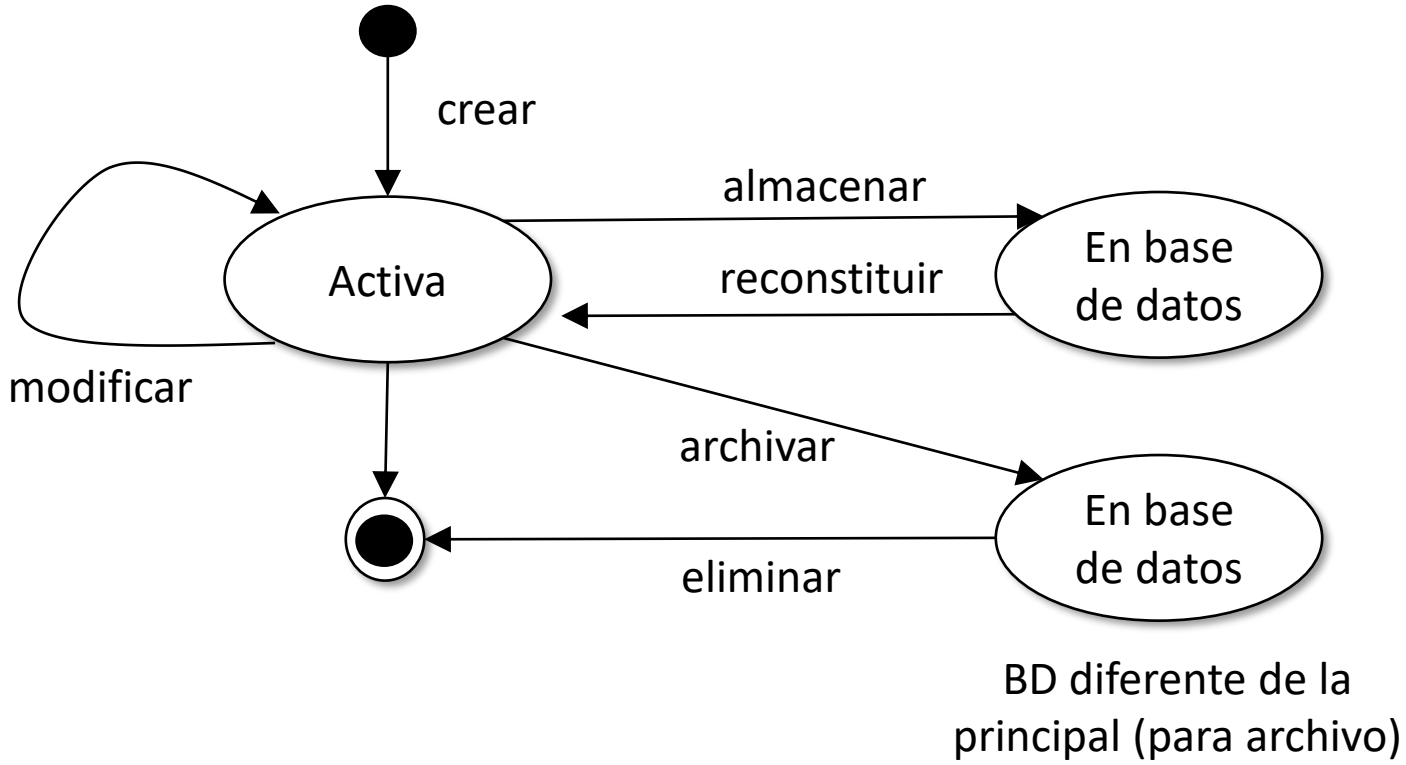
## Identificadores naturales

- DNI.
- Correo electrónico.
- Número de la seguridad social.

## Identificadores sintéticos

- Generados automáticamente con algún algoritmo.
  - UUID.
  - Autogenerado por la BD.
- Interno.
- Visibles para el usuario.
  Número de seguimiento de un pedido.

# 3. Modelado de software – Entidades – Ciclo de vida



# 3. Modelado de software – Objetos valor

## Primitive Obsesión

es un ***code smell*** que aparece cuando se usan tipos primitivos para representar conceptos del dominio que tienen significado y reglas propias.

## Consecuencias:

- Validaciones duplicadas y dispersas.
- Pérdida de significado en el modelo.
- Código más frágil y difícil de mantener.
- Reglas de negocio fuera del dominio.

```
String email vs Email email.
String iban  vs IBAN iban.
double Price vs Money Price.
```

**Si un dato tiene reglas, es un concepto del dominio.**

# 3. Modelado de software – Objetos valor

Para evitar tratar conceptos del dominio como tipos primitivos, usamos objetos valor.

> Un **objeto valor (`value object`)** representa un concepto de dominio sin identidad propia, definido por sus valores y sus reglas.

- Dos objetos valor son iguales si todos sus atributos son iguales.
- Son inmutables.
- Garantizan invariantes del dominio.
  - Los objetos valor incluyen validaciones sobre sus datos.
    Dirección, email, color, …

# 3. Modelado de software – Objetos valor

```
public record Color(int rojo, int verde, int azul) {
   // Validación en el constructor
   public Color {
     if (rojo < 0 || rojo > 255 ||
         verde < 0 || verde > 255 ||
         azul < 0 || azul > 255) {
       throw new IllegalArgumentException(
              "Valores deben estar entre 0 y 255");
     }
   }
}
```

Invariante: un color RGB siempre tiene valores entre 0 y 255.

# 3. Modelado de software – Objetos valor

```
public record Money(BigDecimal amount, Currency currency) {

  public Money {
    if (amount.signum() < 0)
      throw new IllegalArgumentException("Cantidad negativa");
  }

  public Money add(Money other) {
    return new Money(amount.add(other.amount), currency);
  }
}
```

## Inmutabilidad:

El método **`add`** no modifica **`this`**, devuelve un nuevo objeto **`Money`**.

# 3. Modelado de software – Objetos valor

```
public record Email(String value) {
  public Email {
    value = value.trim().toLowerCase();
    if (!value.matches(".+@.+\\..+")) {
      throw new IllegalArgumentException("Email inválido");
    }
  }
}
```

## Validación:

- La validación está dentro del objeto.
- El objeto, o nace válido o no nace.
- El sistema confía en el objeto.

# 3. Modelado de software – Objetos valor

```
public record VerifiedEmail(String value) {

  public VerifiedEmail {
    if (!value.matches(".+@.+\\..+")) {
      throw new IllegalArgumentException("Email inválido");
    }
  }

  public static VerifiedEmail from(Email email) {
    return new VerifiedEmail(email.value());
  }
}
```

Modelado explícito del dominio.

- Evita combinaciones inválidas.
- Las reglas se cumplen por diseño.

# 3. Modelado de software – Entidades - Objetos valor

Las **entidades:**

- Se identifican por su **identidad**.
- **Cambian** a lo largo del tiempo.
- Contienen **lógica de negocio**.
- Protegen **invariantes globales**.

Los **objetos valor:**

- Se definen por su **valor**.
- Son **inmutables**.
- Modelan **conceptos de dominio**.
- Protegen **invariantes locales**.

# 3. Modelado de software – Servicios - Tipos

## Servicios de dominio

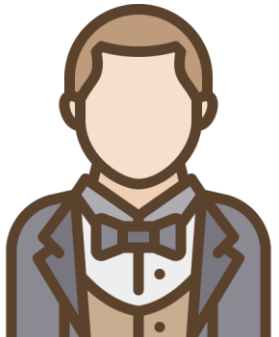
- Modela lógica de negocio.
- Expresa acciones del dominio.
- No depende de frameworks ni infraestructura.

## Servicios de aplicación

- Coordina casos de uso.
- Orquesta entidades y servicios del dominio.
- No contiene lógica de negocio compleja.

## Servicios de infraestructura

- Implementa detalles técnicos.
- Acceso a BD, mensajería y APIs externas.
- Da soporte al resto del sistema.



# 3. Modelado de software – Servicios – Dominio

Un **servicio de dominio** representa operaciones o acciones importantes en el dominio que no encajan naturalmente en una entidad.

- Representan operaciones o acciones del dominio.
- Contiene lógica de negocio, no coordinación técnica.
- Sus parámetros son otros elementos del modelo.
- Son *stateless* (no mantienen estado interno entre invocaciones)
- No es un helper.
  Se usa solo cuando la lógica no pertenece claramente a una entidad.

# 3. Modelado de software – Servicios – Dominio

```
public class ServicioDeContratacion {

  public Contrato contratar (Cliente cliente, Plan plan) {

    if (!cliente.puedeContratar(plan)) {
      throw new ReglaDeNegocio("El cliente no puede
contratar este plan");
    }

    if (!plan.estaDisponible()) {
      throw new ReglaDeNegocio("El plan no está
disponible");
    }

    return new Contrato(cliente, plan);
  }
}
```

## Ejemplo

- Intervienen varias entidades: Cliente, Plan y Contrato.
- La lógica no pertenece solo a Cliente ni solo a Plan
- Modela una acción del dominio: **contratar.**

# 3. Modelado de software – Servicios – Dominio

```
public class CalculadorDePrecioPedido {

  public Money calcularTotal(Pedido pedido) {
    Money total = pedido.subtotal();

    if (pedido.tieneDescuentoPorClienteFrecuente()) {
      total = total.aplicarDescuento(0.10);
    }

    if (pedido.superaCantidadParaEnvioGratis()) {
      total = total.sumar(Envio.GRATIS);
    }

    return total;
  }
}
```

## Ejercicio

¿Qué problemas ves en este servicio?

# 3. Modelado de software – Servicios – Aplicación

Los servicios de aplicación orquestan elementos del modelo, transaccionalidad y otros servicios.
En ciertas arquitecturas, son “use cases”, “controladores”, etc.

Los servicios de aplicación:

- Actúan como un orquestador de casos de uso.
- Reciben un comando de un usuario o sistema externo.
- Coordinan entidades, servicios de dominio y repositorios.
- Controlan transacciones.
- No contienen lógica de negocio compleja.

Ejemplo:
`RegistrarUsuario`, `ConfirmarPago`

Posiblemente más claro usar “UseCase”

# 3. Modelado de software – Servicios – Aplicación

```
public class PrestamoServicioAplicacion {
    private LibroRepository repoLibros;
    private UsuarioRepository repoUsuarios;

    public void prestar(LibroId idLibro, UsuarioId idUsuario) {
      var libro = repoLibros.findbyId(idLibro);
      var usuario = repoUsuarios.findbyId(idUsuario);
      Preconditions.checkNotNull(libro);
      Preconditions.checkNotNull(usuario);
      // Asociar usuario a prestamos. ¿Cómo?
      // Regla de negocio: el usuario no puede tener más
      // de 3 de libros ni estar sancionado.
    }
}
```

# 3. Modelado de software – Servicios – Aplicación

```
public class PrestamoServicioAplicacion {
   private LibroRepository repoLibros;
   private UsuarioRepository repoUsuarios;

   public void prestar(LibroId idLibro, UsuarioId idUsuario) {
     var libro = repoLibros.findbyId(idLibro);
     var usuario = repoUsuarios.findbyId(idUsuario);
     Preconditions.checkNotNull(libro);
     Preconditions.checkNotNull(usuario);
     // Asociar usuario a prestamos. ¿Cómo?
     // Regla de negocio: el usuario no puede tener más
     // de 3 de libros ni estar sancionado.
```

```
Prestamo p = usuario.tomarPrestado(libro);
if (p != null) {
  repoPrestamos.save(p);
}
```

```
if (reglaNegocioOk(libro, usuario) {
  var p = new Prestamo(libro, usuario);
  repoPrestamos.save(p);
}
```

# 3. Modelado de software – Servicios – Aplicación

```
public class PrestamoServicioAplicacion {
   private LibroRepository repoLibros;
   private UsuarioRepository repoUsuarios;

   public void prestar(LibroId idLibro, UsuarioId idUsuario) {
     var libro = repoLibros.findbyId(idLibro);
     var usuario = repoUsuarios.findbyId(idUsuario);
     Preconditions.checkNotNull(libro);
     Preconditions.checkNotNull(usuario);
     // Asociar usuario a prestamos. ¿Cómo?
     // Regla de negocio: el usuario no puede tener más
     // de 3 de libros ni estar sancionado.
```

```
Prestamo p = usuario.tomarPrestado(libro);
if (p != null) {
  repoPrestamos.save(p);
}
```

```
if (reglaNegocioOk(libro, usuario) {
  var p = new Prestamo(libro, usuario);
  repoPrestamos.save(p);
}
```

**Lógica de negocio en la capa de aplicación**

# 3. Modelado de software – Servicios - Infraestructura

Los servicios de infraestructura proporcionan implementaciones técnicas que el dominio y la aplicación necesitan para funcionar, pero no forman parte del negocio.

## Características:

- Implementan detalles técnicos.
- Dependen de tecnologías externas.
- No contienen lógica de negocio.
- Dar soporte a los servicios de aplicación.

## Ejemplos:

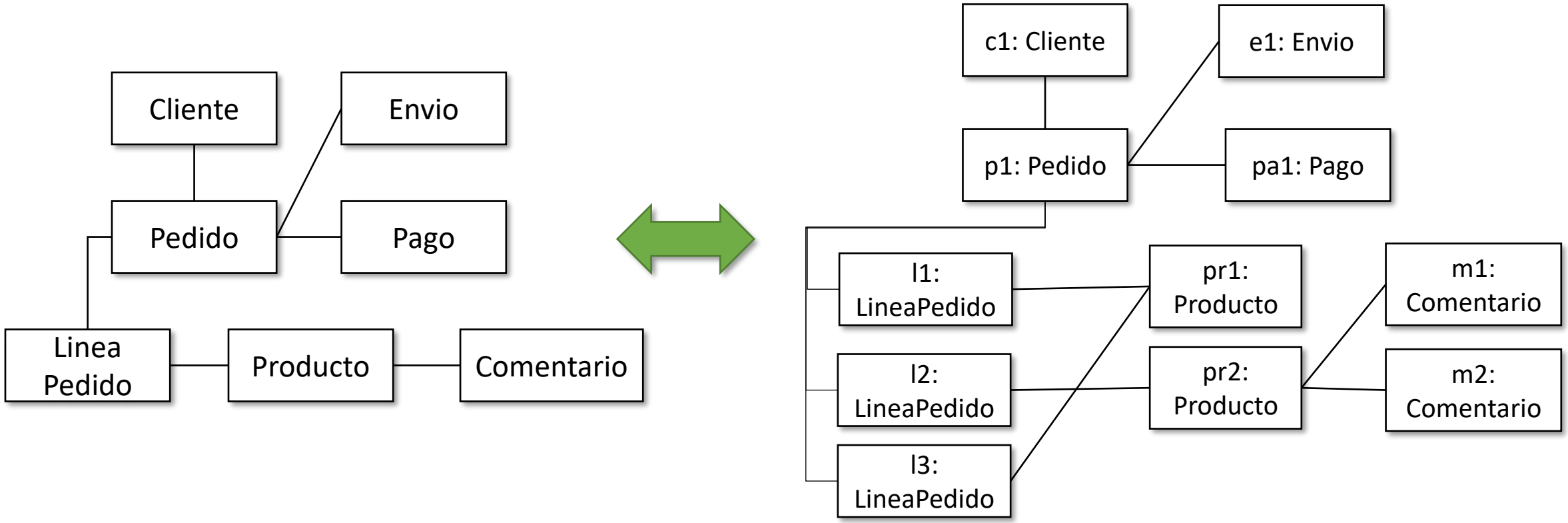
- Envío de e-mails.
- Persistencia.
- Mensajería.
- APIs externas.

# 3. Modelado de software – Agregados

Un modelo OO completo puede contener muchas clases.

No es viable manejar todos los objetos como “una sopa de objetos”

- Mantener la consistencia de todos los objetos es muy complicado.
- Big ball of mud.



# 3. Modelado de software – Agregados

> Un **agregado (aggregate)** es un conjunto de objetos del dominio que se tratan como una unidad de consistencia, y cuya coherencia se protege mediante reglas del dominio.

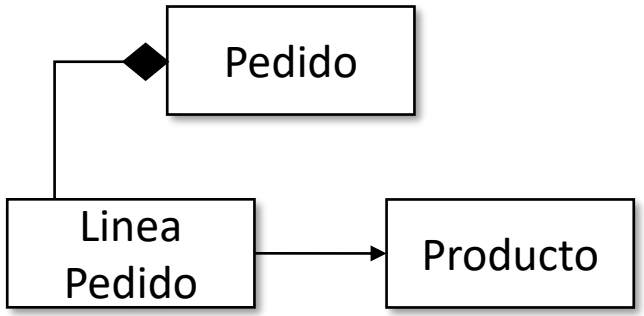
Un agregado:

- Protege invariantes que afectan a varios objetos.
- Tras una operación, el agregado debe estar en un estado válido.
- Garantiza la consistencia del agregado, no fuera.

# 3. Modelado de software – Agregados

## Relación con la composición en UML

En UML, la **composición** (agregación fuerte) indica que el ciclo de vida del objetivo contenido depende del contenedor.



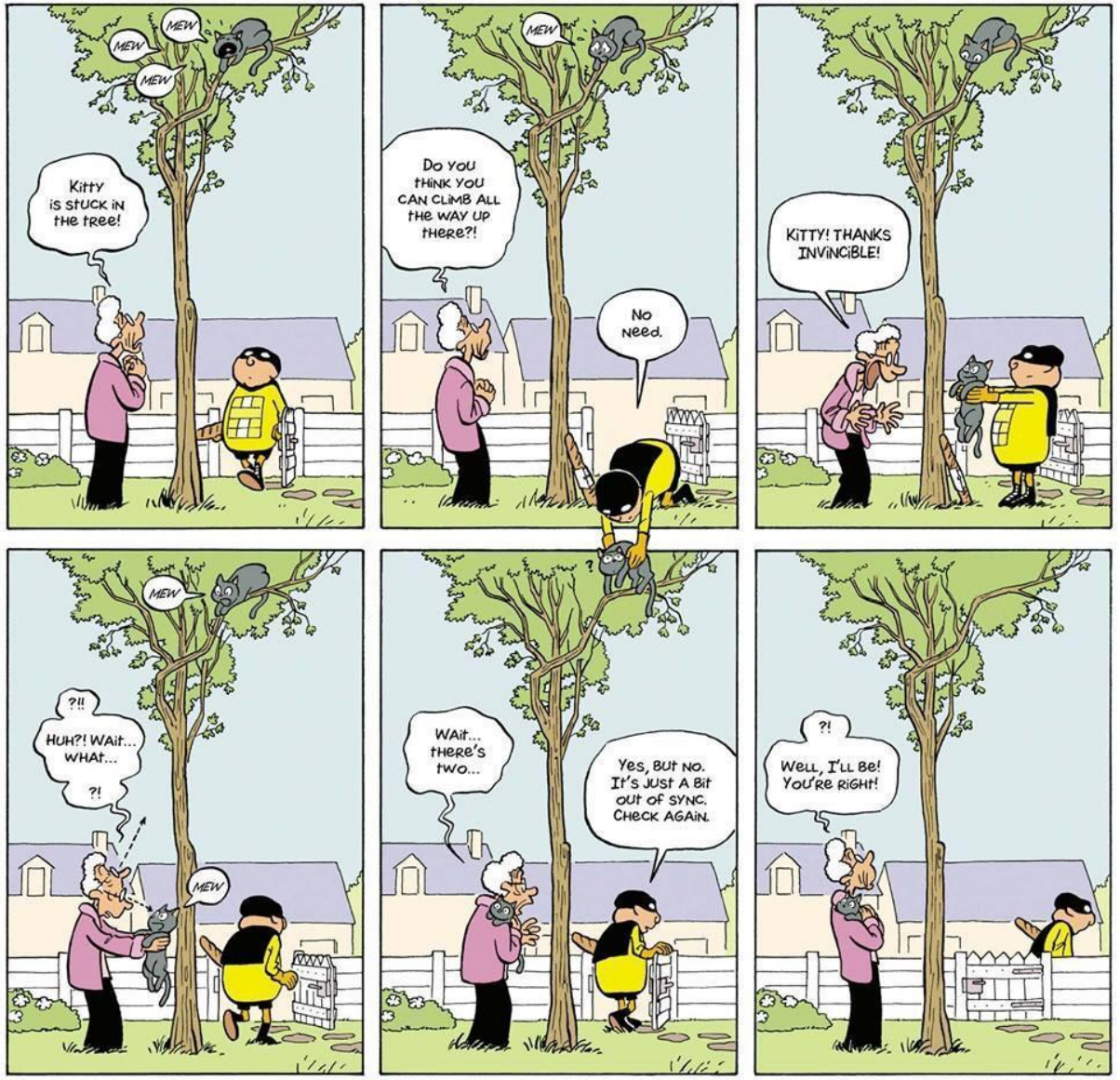

- ¿Si un pedido se borra, qué pasa con las líneas?
- ¿Y con los productos? ¿Y si un producto se borra?

# 3. Modelado de software – Consistencia

En sistemas reales, no todas las reglas del dominio requieren el mismo nivel de consistencia.

- **Consistencia fuerte o transaccional.**
  Las invariantes deben cumplirse al finalizar la operación.
  - Un pedido no puede tener líneas con cantidad negativa.
  - El total del pedido debe coincidir con la suma de sus líneas.
  - Una reserva no puede existir sin cliente.
- **Consistencia eventual.**
  Las invariantes pueden cumplirse con el tiempo, de forma asíncrona.
  - El stock global se actualiza después de confirmar el pedido.
  - El e-mail de confirmación se envía tras crear el pedido.

# 3. Modelado de software – Consistencia eventual



# 3. Modelado de software – Agregados - Reglas

Cada agregado tiene una raíz (**`Aggregate Root`**).

- Es la única entidad que puede ser referenciada desde fuera del agregado.
- Todas las operaciones se realizan a través de la raíz, que define la API del agregado.
- Los invariantes del agregado se garantizan al final de cada operación que afecta al agregado.
- Las invariantes que deben cumplirse “siempre” deben estar dentro del mismo agregado.
- Entre agregados solo existe consistencia eventual.
- Normalmente existe un repositorio por agregado.
- Un agregado solo puede apuntar a la raíz de otro agregado por su ID.

# 3. Modelado de software – Agregados - Ejemplo

## Equipo

- Un equipo no puede tener más de 24 jugadores activos.
- No puede haber dos jugadores con el mismo número.

## Jugador

- No puede estar activo en más de un equipo.
- Solo puede cambiar de número si el equipo lo permite.

## Liga

- Dos equipos no pueden tener el mismo nombre
- La liga puede cerrar inscripciones y ningún equipo puede cambiar.

## Jugadores sin equipo

- Disponible para jugar en un equipo.

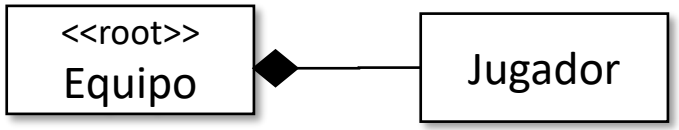
# 3. Modelado de software – Agregados - Ejemplo

## Equipo

- Un equipo no puede tener más de 24 jugadores activos.
- No puede haber dos jugadores con el mismo número.

## Jugador

- No puede estar activo en más de un equipo.
- Solo puede cambiar de número si el equipo lo permite.




## Liga

- Dos equipos no pueden tener el mismo nombre
- La liga puede cerrar inscripciones y ningún equipo puede cambiar.

## Jugadores sin equipo

- Disponible para jugar en un equipo.

# 3. Modelado de software – Agregados - Ejemplo

## Equipo

- Un equipo no puede tener más de 24 jugadores activos.
- No puede haber dos jugadores con el mismo número.

## Jugador

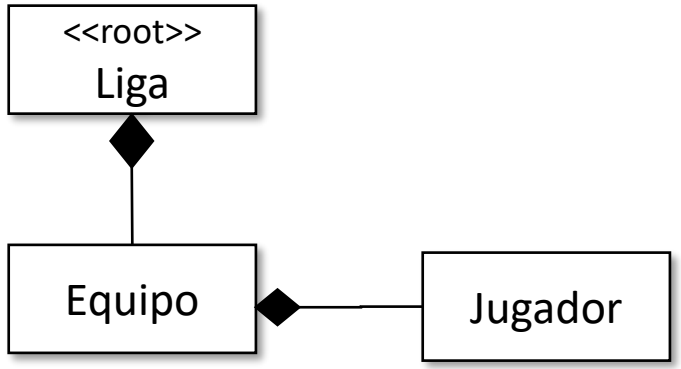
- No puede estar activo en más de un equipo.
- Solo puede cambiar de número si el equipo lo permite.

## Liga

- Dos equipos no pueden tener el mismo nombre
- La liga puede cerrar inscripciones y ningún equipo puede cambiar.

## Jugadores sin equipo

- Disponible para jugar en un equipo.




- Invariantes de la liga tienen que cumplirse fuertemente
- Los equipos no se añadirán a las ligas concurrentemente

# 3. Modelado de software – Agregados - Ejemplo

## Equipo

- Un equipo no puede tener más de 24 jugadores activos.
- No puede haber dos jugadores con el mismo número.

## Jugador

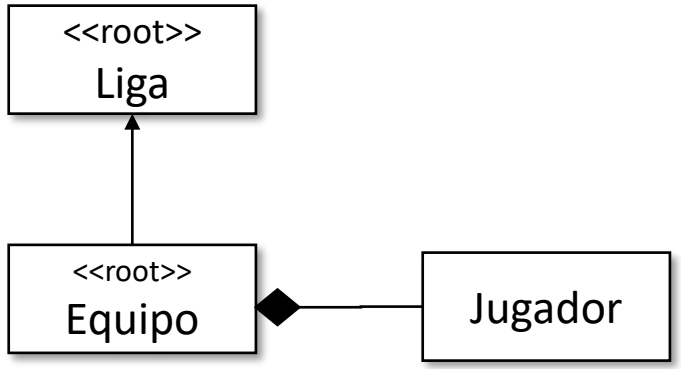
- No puede estar activo en más de un equipo.
- Solo puede cambiar de número si el equipo lo permite.

## Liga

- Dos equipos no pueden tener el mismo nombre
- La liga puede cerrar inscripciones y ningún equipo puede cambiar.

## Jugadores sin equipo

- Disponible para jugar en un equipo.




- Invariantes de la liga pueden relajarse.
- Se requiere concurrencia
- No se quiere agregados grandes (liga cargada con frecuencia)

# 3. Modelado de software – Agregados - Ejemplo

## Equipo

- Un equipo no puede tener más de 24 jugadores activos.
- No puede haber dos jugadores con el mismo número.

## Jugador

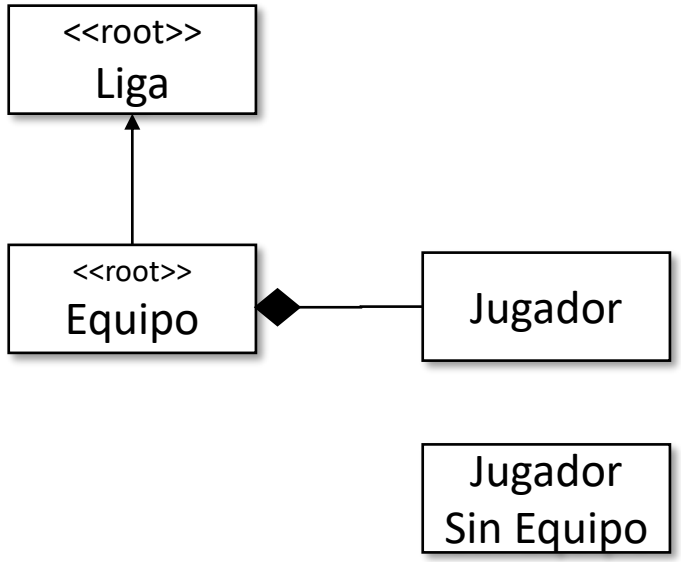
- No puede estar activo en más de un equipo.
- Solo puede cambiar de número si el equipo lo permite.

## Liga

- Dos equipos no pueden tener el mismo nombre
- La liga puede cerrar inscripciones y ningún equipo puede cambiar.

## Jugadores sin equipo

- Disponible para jugar en un equipo.




```
Equipo.fichar(jugadorSinEquipo)
```

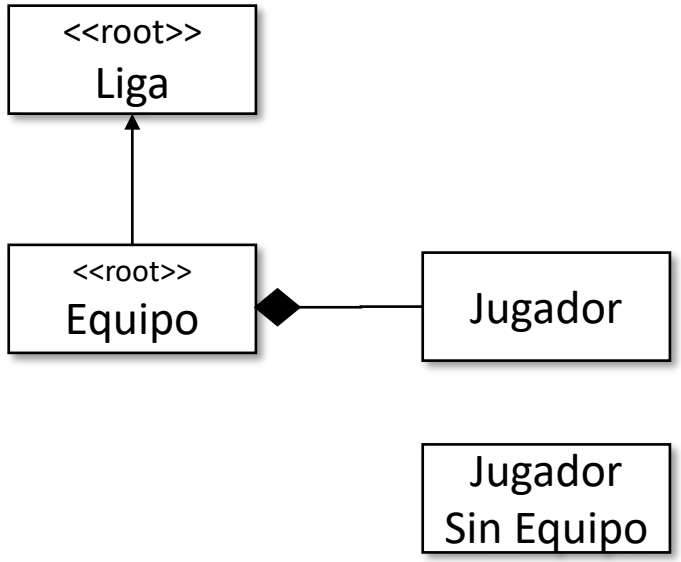
Crea un jugador a partir de los datos del jugador sin equipo (que se puede eliminar)

# 3. Modelado de software – Agregados - Ejemplo

```
public void ficharJugador(JugadorSinEquipoId id,
                          EquipoId equipoId) {
  JugadorSinEquipo libre = libres.findById(id);
  Equipo equipo = equipos.findById(equipoId);

  equipo.fichar(libre);

  repositorioJugadorSinEquipo.delete(libre);
  repositorioEquipos.save(equipo);
}
```



# 3. Modelado de software – Factoría

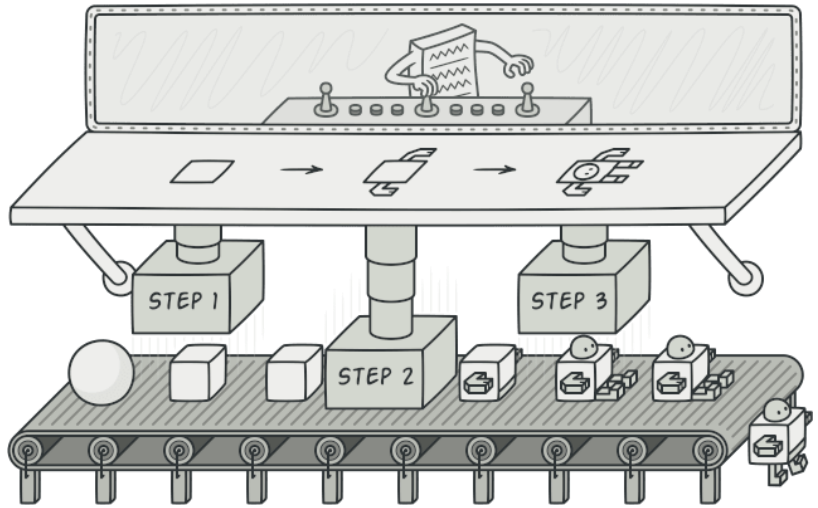
Una **factoría (`factory`)** se encarga de crear instancias de objetos complejos, garantizando que se cumplen todos los invariantes.

- Utilización de los patrones de creación.
  - Método factoría.
  - Factoría abstracta.
  - Builder.
- Relacionado con el patrón GRASP.

# 3. Modelado de software – Patrón Builder

## Problema con los objetos complejos

- Requieren validaciones complejas.
- No se pueden crear “de golpe” con un constructor.
- Los constructores largos son difíciles de leer.
- Permiten estados inválidos.
- Rompen encapsulación.




## Idea del patrón Builder

https://refactoring.guru/design-patterns/builder

Separar la construcción de un objeto de su representación final.

- Guía el proceso de creación paso a paso.
- Valida antes de crear el objeto final
- Garantiza que el objeto nace válido

# 3. Modelado de software – Repositorio

> Un repositorio **(repository)** representa la ficción de una colección de objetos del dominio, proporcionando métodos para recuperarlos y persistirlos, ocultando los detalles técnicos.

- **Un repositorio por cada agregado**
  Se trabaja siempre a nivel de `Aggregate Root`.
- **Mapea** identificadores a objetos del dominio.
  `PedidoId` → `Pedido`
- **Oculta** la persistencia al dominio.
  BD, ORM, SQL, etc. no aparecen en el modelo.

# 4. Eventos de dominio

Comunicar elementos del modelo

# 4. Eventos de dominio – Definición

> Un evento de dominio **(domain event)** representa algo importante que ha ocurrido en el dominio y que otros componentes pueden necesitar conocer.

- Describe hechos pasados, no acciones futuras.
- Normalmente se nombra en pasado `PedidoConfirmado`, `PagoAceptado`, `UsuarioRegistrado`
- Se expresa en lenguaje del dominio.
- Es inmutable.

# 4. Eventos de dominio – ¿Dónde viven?

Los **eventos de dominio nacen dentro de un agregado** cuando se cumple una regla de negocio relevante.

- El `Aggregate Root` decide cuándo lanzar el evento.
- El `Aggregate Root` garantiza que los invariantes se cumplen antes del evento.
- El evento se produce al final de una operación válida.

# 4. Eventos de dominio – ¿Para qué se usan?

Los **eventos de dominio** se usan para:

- Notificar a otros agregados.
- Desencadenar procesos secundarios.
- Integrarse con otros sistemas.
- Desacoplar lógica.

**Ejemplos**:

`PedidoConfirmado` → `EnviarEmail, IniciarEnvío, GenerarFactura,…`

`UsuarioRegistrado` → `CrearPerfil, EnviarBienvenida, …`

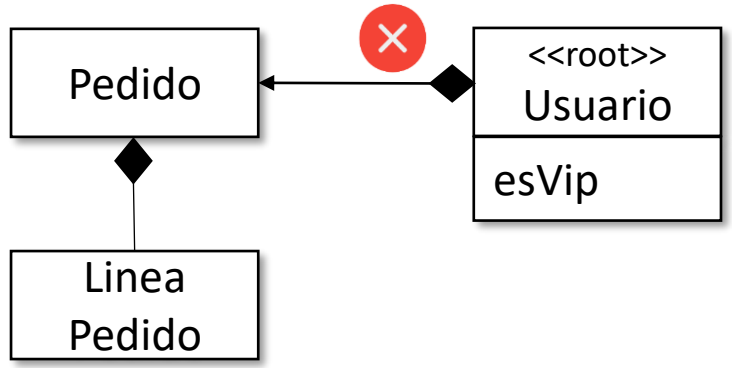
# 4. Eventos de dominio – Ejemplo

## Regla de negocio

*Un pedido con más de 100 ítems convierte al cliente en VIP.*

## Preguntas

1. ¿Es el **`usuario`** un agregado que contiene **`pedidos`**?
   - Un usuario puede tener muchos pedidos.
   - Esto genera agregados muy grandes.
   - Mala escalabilidad y acoplamiento.
2. ¿Implementamos la regla de negocio en un servicio?
   - Implica modificar otro agregado directamente.
   - Regla: no modificar dos agregados en la misma transacción




```
... manejo del pedido...
if (pedido.size() > 100) {
  var u = repoUsuarios.findById(uId);
  u.setVip(true);
  repoUsuarios.save(u);
}
```

**Lo que realmente necesitamos es consistencia eventual.**

No es crítico que la regla de negocio se aplique de forma inmediata

# 4. Eventos de dominio – Ejemplo

## Regla de negocio

*Un pedido con más de 100 ítems convierte al cliente en VIP.*

## ¿Dónde se genera el evento?

- Dentro del agregado Pedido.
- En el momento en el que se cumple la condición.

```
... manejo del pedido...
if (pedido.size() > 100) {
  registrarEvento(new
PedidoConMasDe100Items(idPedido, idCliente));
}
```

## ¿Quién reacciona al evento?

- Un listener externo al agregado.
- Puede modificar otro agregado.

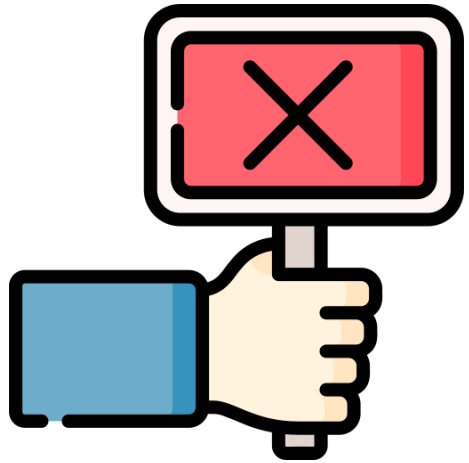
```
on (PedidoConMasDe100Items e) {
    Usuario u = repoUsuarios.findById(e.idCliente());
    u.marcarComoVip();
    repoUsuarios.save(u);
}
```

# 4. Eventos de dominio – ¿Cuándo se usan?

- Ocurre algo **relevante** en el negocio.
  Pedido confirmado, usuario bloqueado, pago rechazado…
- La reacción **no necesita ser inmediata**.
  Consistencia eventual es aceptable.
- La lógica **afecta a otros agregados**.
  Evita referencias directas entre agregados.
- Quieres **desacoplar el modelo**.
  El agregado no sabe quién reaccionará.
- El hecho es algo que el negocio podría decir en voz alta.
  Cuando pasa X, entonces Y.

# 4. Eventos de dominio – ¿Cuándo NO se usan?

- La regla debe cumplirse siempre y de forma inmediata.
  Invariante fuerte → mismo agregado.
- La lógica pertenece claramente a una entidad.
  Mejor un método del agregado.
- Solo quieres reutilizar código.
  Recuerda: Un evento no es un helper.
- Es un detalle técnico.
  Logs, métricas, cache, emails
- Complica innecesariamente el modelo.



# Bibliografía


- Domain-Driven Design: Tackling Complexity in the Heart of Software. 2003. Eric Evans.
- Implementing Domain-Driven Design. 2013. Vernon Vaughn.
- Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. 2021. Vladik Khononov
- DDD. Marco Tulio Valente.
  - https://softengbook.org/articles/ddd