autentia

SOFTWARE DESIGN

GUÍA COMPLETA

GUIA PARA
DIRECTIVOS Y TÉCNICOS

V.3

Software Design

Guía completa

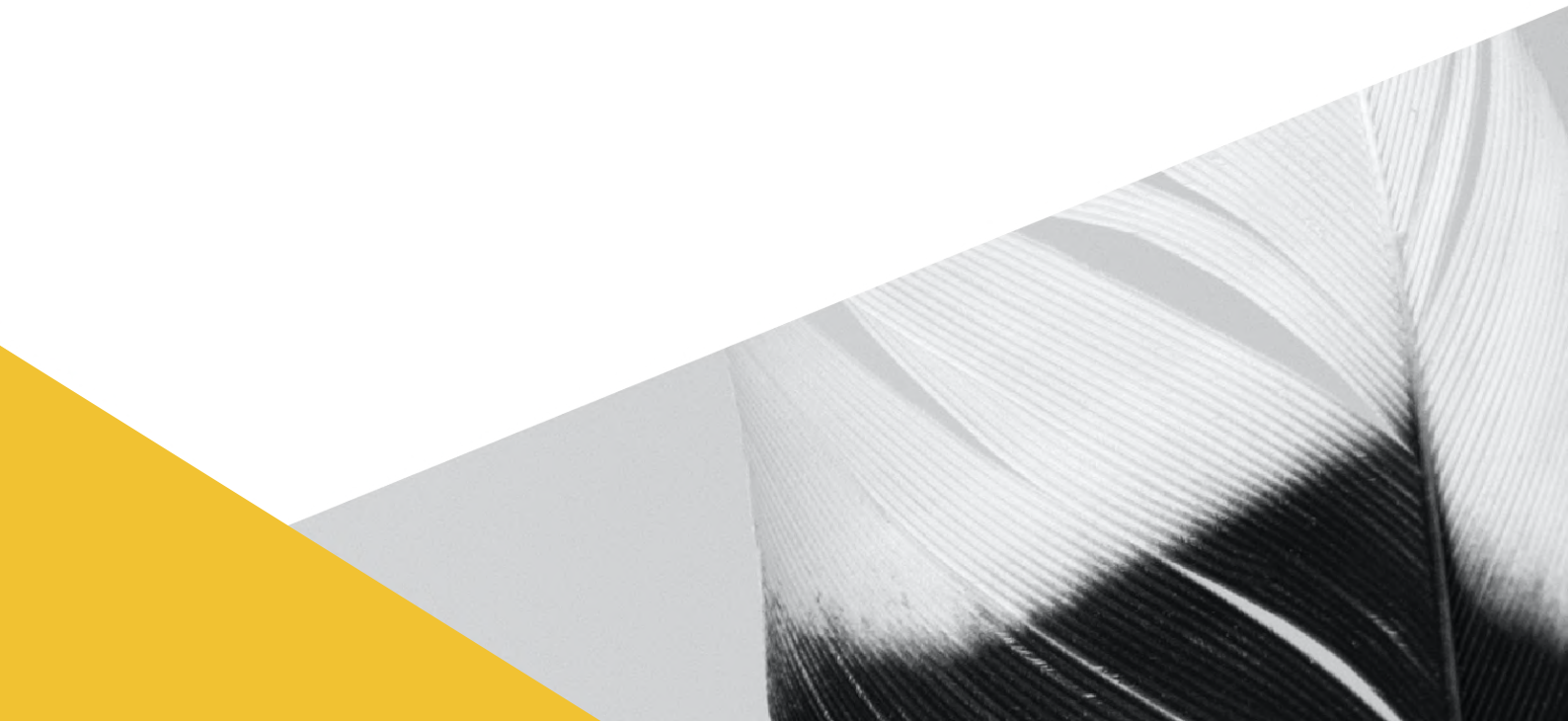
Este documento forma parte de las guías de onboarding de Autentia. Si te apasiona el desarrollo de software de calidad ayúdanos a difundirlas y anímate a unirte al equipo. Este es un documento vivo y puedes encontrar la última versión, así como el resto de partes que completan este documento, en nuestra web.

https://www.autentia.com/libros/

Esta obra está licenciada bajo la licencia Creative Commons Attribution ShareAlike 4.0 International (CC BY-SA 4.0)

En las fases iniciales del desarrollo de un proyecto de software, durante sus primeras versiones, las nuevas funcionalidades fluyen de manera graciosa y natural, casi parecen construirse solas y "todo son vino y rosas". A medida que los proyectos van avanzando y creciendo, de repente, toda esa magia desaparece y comienzan a surgir una serie de problemas y situaciones que muestran que algo no va bien, son síntomas de que nuestros diseños se están degradando y pudriendo.

"Sólo el tiempo demostrará si nuestro software, al enfrentarse a los fuertes vientos del cambio, está asentado sobre roca firme o sobre arena"



Los síntomas más evidentes son:

- **Rigidez:** el software se vuelve difícil de cambiar incluso en tareas sencillas. Las estimaciones son cada vez más abultadas y cada funcionalidad nueva cuesta horrores cuando antes era casi inmediata

- **Fragilidad:** muy relacionado con la rigidez, la fragilidad es la tendencia a que el software se rompa en múltiples sitios cada vez que se hace un cambio, incluso en partes que conceptualmente no tienen relación ninguna las unas con las otras. Cuando subimos una nueva versión que ha costado "sudor y lágrimas", resulta que se rompen cosas que aparentemente no tienen nada que ver con lo que hemos hecho.

- **Inmovilidad o poca reutilización:** cuando resulta imposible reutilizar software de otros proyectos o incluso de otras partes del mismo proyecto. Suele ocurrir porque el módulo que queremos reutilizar tiene una mochila de dependencias demasiado grande como para asumir el esfuerzo y el riesgo de desacoplarlo. Se piden cosas que son prácticamente una copia de otras funcionalidades de las que ya disponemos, sin embargo esto no parece ser una ventaja y cuesta demasiado sacar lo común para reutilizarlo, tendiendose a copiar y duplicar código.

- **Viscosidad:** la viscosidad en el ámbito del diseño se da cuando es más sencillo hacer las cosas mal, hacer la "ñapa", que tratar de

hacerlas por el camino trazado. La viscosidad en el entorno ocurre cuando el ecosistema de desarrollo es lento e ineficiente en el más amplio sentido de la palabra. Por ejemplo, cuando una vez tenemos la funcionalidad terminada, hacerla llegar hasta producción supone una auténtica aventura y se necesitan días (o semanas).

Design Smells autentia

¿Qué son?

Los "olores de diseño" dentro del desarrollo del software son una manera de definir los síntomas que surgen con el avance del proyecto y crecimiento del software que influyen en la degradación del diseño.

CARACTERÍSTICAS

Síntoma	Definición	Consecuencias
Rigidez	Dificultad de realizar cambios en el software, incluso en tareas sencillas.	• Las estimaciones son más abultadas. • Añadir funcionalidades nuevas cuesta mucho cuando antes era más rápido.
Fragilidad	Tendencia a que el software se rompa en múltiples sitios cada vez que se realiza un cambio en él, incluso sin tener relación el cambio con lo que se rompe.	• Subir una nueva versión y que se rompan partes que aparentemente no tienen que ver con lo modificado. • Aparición de mayor cantidad de bugs, aumentando el coste de esfuerzo y tiempo en corregirlos.
Inmovilidad	Incapacidad de reutilizar piezas de código por el gran acoplamiento que tiene, haciendo que este sea inamovible cuando la funcionalidad es prácticamente la misma que la deseada.	• Aumenta la complejidad del diseño y del código al introducir código duplicado. • Complica el entendimiento del código y genera equivocaciones.
Viscosidad	La tendencia de que, en el ámbito del software, realizar la solución buena requiere mayor esfuerzo comparado con la solución mala y rápida y, en el ámbito del entorno de desarrollo, éste es lento e ineficiente.	• Provoca el efecto bola de nieve: cuanto más tarde se corrija esa deuda técnica más difícil y costoso será resolverlo. • Pérdida de tiempo, estrés, confusión, pasotismo, acciones inseguras, etc.

Los principios de desarrollo de software son una serie de **reglas y recomendaciones** específicas que los programadores deben seguir durante el desarrollo si quieren escribir un código limpio, comprensible y fácil de mantener. No hay una varita mágica por medio de la cual se pueda transformar una combinación de variables, clases y funciones en el código ideal, pero hay algunos consejos y sugerencias que pueden ayudar al programador a determinar si está haciendo las cosas bien y tratar de evitar las situaciones que a modo de ejemplo hemos narrado en el apartado anterior, y que seguro, que si llevamos el tiempo suficiente dedicados a esta maravillosa profesión del desarrollo de software, habremos vivido, sino igual, al menos de manera similar.

Software Design

Principios y patrones del desarrollo de software

Índice

Parte 1: Principios y patrones del desarrollo de software

- Principios generales
  - Principios S.O.L.I.D.
    - Single responsibility (SRP)
    - Open/closed (OCP)
    - Liskov substitution (LSP)
    - Interface segregation (ISP)
    - Dependency inversion (DIP)
  - Don’t Repeat Yourself (DRY)
  - Inversion of Control (IoC)
  - You Aren’t Gonna Need It (YAGNI)
  - Keep It Simple, Stupid (KISS)
  - Law of Demeter (LoD)
  - Strive for loosely coupled design between objects that interact
  - Composition over inheritance
  - Encapsulate what varies
  - The four rules of simple design
  - The boy scout rule
  - Last Responsible Moment

- Design Patterns
    - Patrones creacionales
        - Builder
        - Singleton
        - Dependency Injection
        - Service Locator
        - Abstract Factory
        - Factory Method
    - Patrones estructurales
        - Adapter
        - Data Access Object (DAO)
        - Query Object
        - Decorator
        - Bridge
    - Patrones de comportamiento
        - Command
        - Chain of Responsibility
        - Strategy
        - Template Method
        - Interpreter
        - Observer
        - State
        - Visitor
        - Iterator
- Patrones JEE
    - Introducción
    - Intercepting filter
    - Front controller
    - Dispatcher view
    - Business delegate
    - Value object
    - Session facade

        - Composite entity
        - Transfer object assembler
        - Value list handler
        - Service locator
        - Data access object
        - Service activator
- Antipatrones
    - Antipatrones de diseño
        - The Blob
        - Lava Flow
        - Ambiguous Viewpoint
        - Functional Decomposition
        - Poltergeists
        - Spaghetti Code
        - Input Kludge
    - Antipatrones de metodología
        - Golden Hammer
        - Cut-And-Paste Programming
    - Antipatrones de gestión
        - Continuous Obsolescence
        - Mushroom Management
        - Dead End
        - Boat Anchor
        - Walking through a Minefield

Parte 2: Principios y patrones de la arquitectura del software

- Introducción a la Arquitectura de Software
- DDD: Domain-Driven Design
    - Building blocks
        - Entities
        - Value objects
        - Modules
        - Servicios
        - Aggregates

        - Repositorios
        - Factorías
        - Diseño estratégico
        - Introducción
        - Bounded context
        - Subdomains
        - Introducción a las relaciones entre los Bounded Context
        - Shared kernel
        - Customer/Supplier
        - Capa de Anticorrupción
- Command-Query Responsibility Segregation
    - Command
    - Query
- Clean Architecture
    - Capas
        - Capa de dominio
        - Capa de aplicación
        - Capa de infraestructura
    - Hexagonal architecture
        - Bloques de un sistema
        - Puertos y adaptadores
    - Componentes
    - Flujo de control
        - Usando servicios de aplicación.
        - CQRS sin Bus
        - CQRS con Bus
    - Testing
    - Event-driven architecture
        - Introducción
        - Evento de dominio
        - Contexto
    - Event sourcing
    - Consistencia eventual

Bibliografía

Lecciones aprendidas con esta guía

Parte 1

Principios y patrones del desarrollo de software

Principios generales

Principios S.O.L.I.D.

S.O.L.I.D. es un acrónimo mnemónico para cinco principios de diseño destinados a hacer que los diseños de software sean más comprensibles, flexibles y mantenibles. Los principios son un subconjunto de muchos principios promovidos por el ingeniero e instructor de software estadounidense Robert C. Martin. Aunque se aplican a cualquier diseño orientado a objetos, los principios SOLID también pueden formar una filosofía central para metodologías como el desarrollo ágil o el desarrollo de software adaptativo.

Los principios comprendidos en S.O.L.I.D. son:

- **S**: Single responsibility.
- **O**: Open/closed.
- **L**: Liskov substitution.

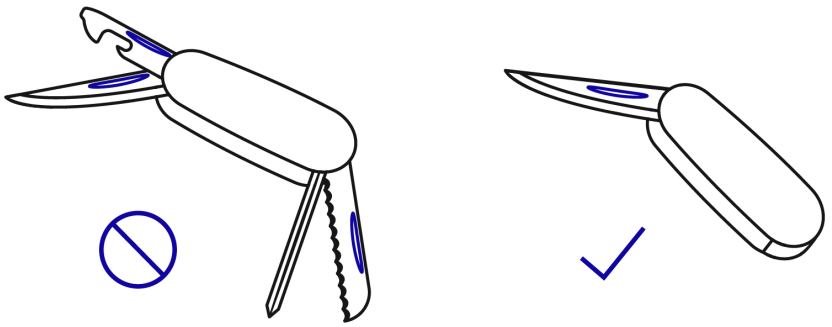
- **I**: Interface segregation.
- **D**: Dependency inversion.

Aplicar estos principios facilitará mucho el trabajo, tanto propio como ajeno (es muy probable que nuestro código lo acaben leyendo muchos otros desarrolladores a lo largo de su ciclo de vida). Algunas de las ventajas de aplicarlo son:

- Facilitar el mantenimiento del código.
- Reducir la complejidad de añadir nuevas funcionalidades.
- Aumentar la reusabilidad de piezas y componentes.
- Mejorar la calidad del código y su comprensión.

Single responsibility (SRP)

El principio de responsabilidad única o single responsibility establece que **un módulo de software debe tener una y solo una razón para cambiar**. Esta razón para cambiar es lo que se entiende por responsabilidad.



“Reúna las cosas que cambian por las mismas razones. Separe las cosas que cambian por diferentes razones.”

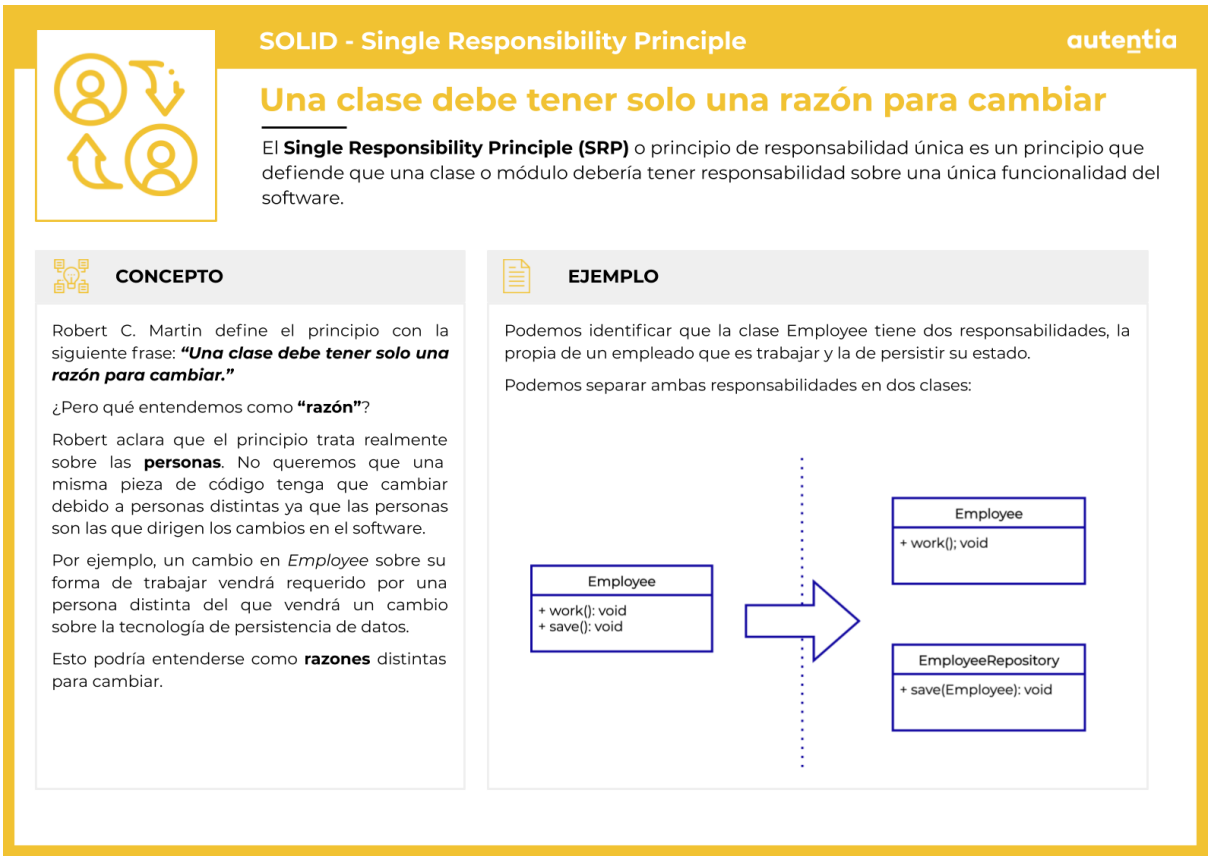
Este principio está estrechamente relacionado con los conceptos de acoplamiento y cohesión. Queremos aumentar la cohesión entre las cosas que cambian por las mismas razones y disminuir el acoplamiento entre las cosas que cambian por diferentes razones. Este principio trata sobre **limitar el impacto de un cambio**.

Si existe más de una razón para cambiar una clase, probablemente tenga más de una responsabilidad. Otro posible "mal olor" es que tenga diferentes comportamientos dependiendo de su estado. Tener más de una responsabilidad también hace que el código sea difícil de leer, testear y mantener. Es decir, hace que el código sea menos flexible.

Entre las ventajas de aplicar este principio encontramos que, si se necesita hacer algún cambio, éste será fácil de detectar ya que estará aislado en una clase claramente definida y comprensible. Minimizando los efectos colaterales en otras clases. Algunos ejemplos que encontramos en la vida real son:

Si cambia la forma en que se compra un artículo, no tendremos que modificar el código responsable de almacenarlo. Si cambia la base de datos, no habrá que arreglar cada pedazo de código donde se utiliza.

Para más detalle, ver el artículo *[S.O.L.I.D.] Single responsibility principle / Principio de Responsabilidad Única* en Adictos al Trabajo.



Open/closed (OCP)

Este principio nos dice que **los módulos de software deben ser abiertos para su extensión pero cerrados para su modificación**. ¿A qué se refiere con esto?

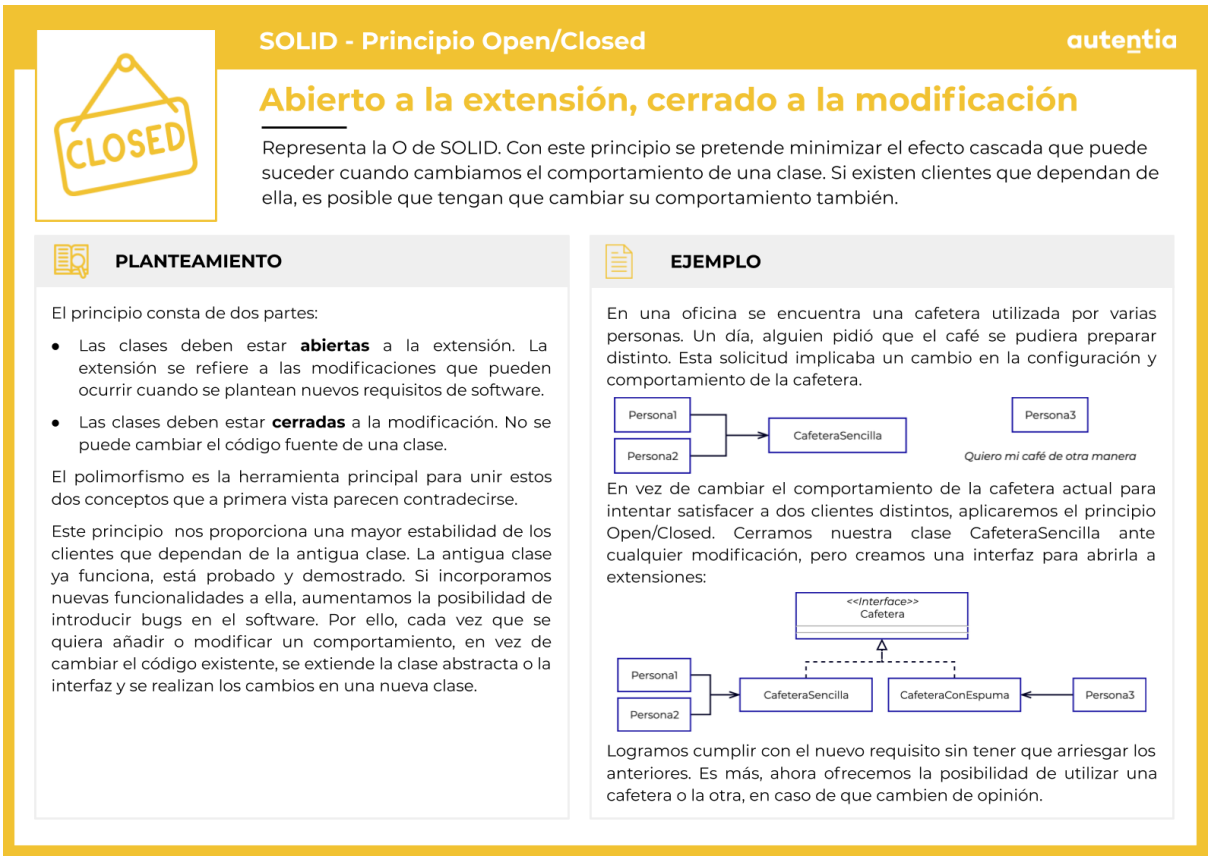
- **Abierto para la extensión**: esto significa que el comportamiento del módulo puede extenderse. A medida que cambian los requisitos de la aplicación, podemos ampliar el módulo con nuevos comportamientos que satisfagan esos cambios. En otras palabras, podemos cambiar lo que hace el módulo.
- **Cerrado por modificación**: un módulo estará cerrado si dispone de una descripción (interface) estable y bien definida. Extender el comportamiento de un módulo no debería afectar al código ya existente en el módulo, es decir, el código original del módulo no

debería verse afectado y tener que modificarse.

Y esa es realmente la esencia de este principio. Debería ser fácil cambiar el comportamiento de un módulo sin cambiar el código fuente de ese módulo. Esto no significa que nunca cambiará el código fuente. Lo que significa es que debemos esforzarnos por lograr que nuestro código esté estructurado de forma que, cuando el comportamiento cambie de la manera esperada, no debamos hacer cambios radicales en todos los módulos del sistema. Idealmente, podremos agregar el nuevo comportamiento, añadiendo código nuevo y cambiando poco o nada del código antiguo.

La forma de implementar este principio en el mundo práctico, es a través del polimorfismo, ya sea por interfaces o clases abstractas.

Para más detalle, ver el artículo *[S.O.L.I.D.] Open-Closed Principle / Principio Abierto-Cerrado* en Adictos al Trabajo.



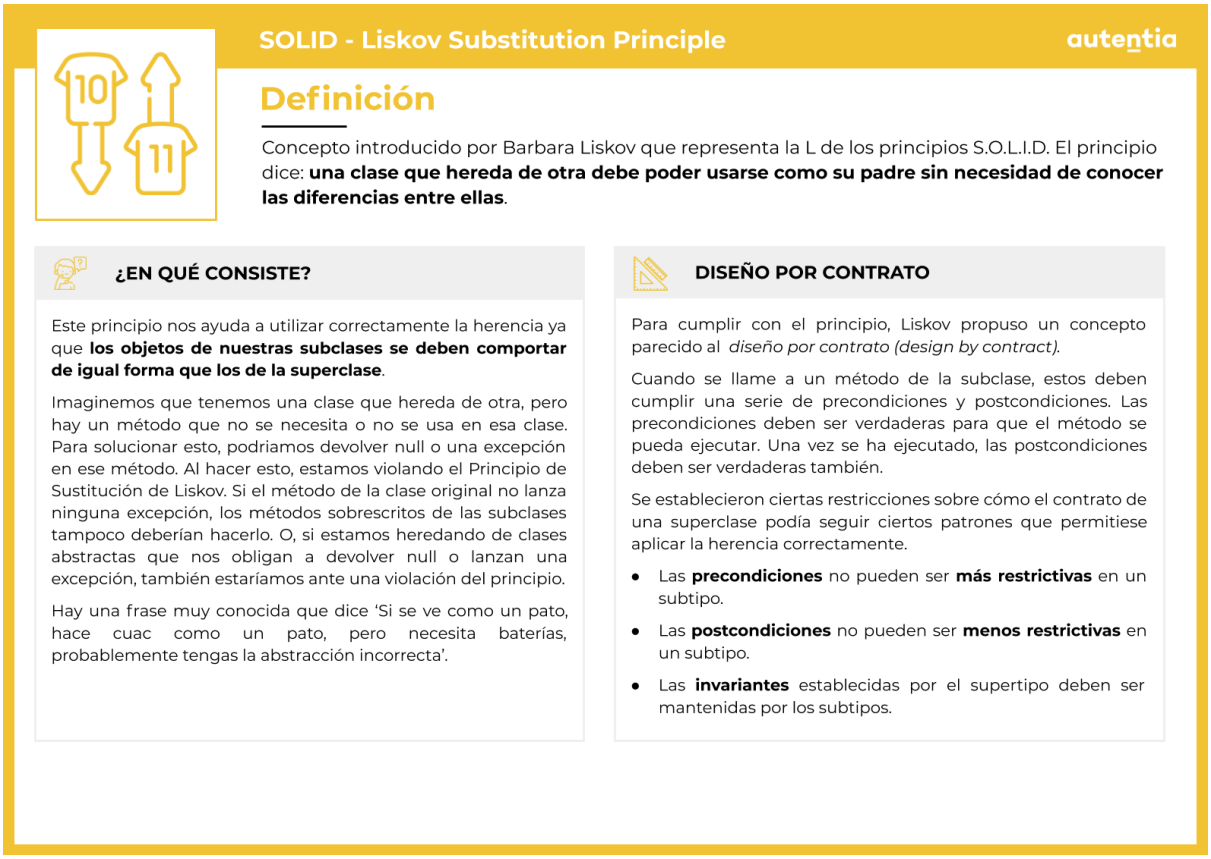
Liskov substitution (LSP)

"Si se ve como un pato, hace cuac como un pato, pero necesita baterías, probablemente tengas la abstracción incorrecta."

La sustitución de Liskov nos dice que los **objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento** del programa. Básicamente, si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido. Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la clase padre.

Este principio nos **ayuda a utilizar la herencia de forma correcta** y nos muestra que no se debe mapear automáticamente el mundo real en un modelo orientado a objetos, ya que no existe una equivalencia unívoca entre ambos modelos.

Para más detalle, ver el artículo *[S.O.L.I.D.] Liskov substitution* en Adictos al Trabajo.



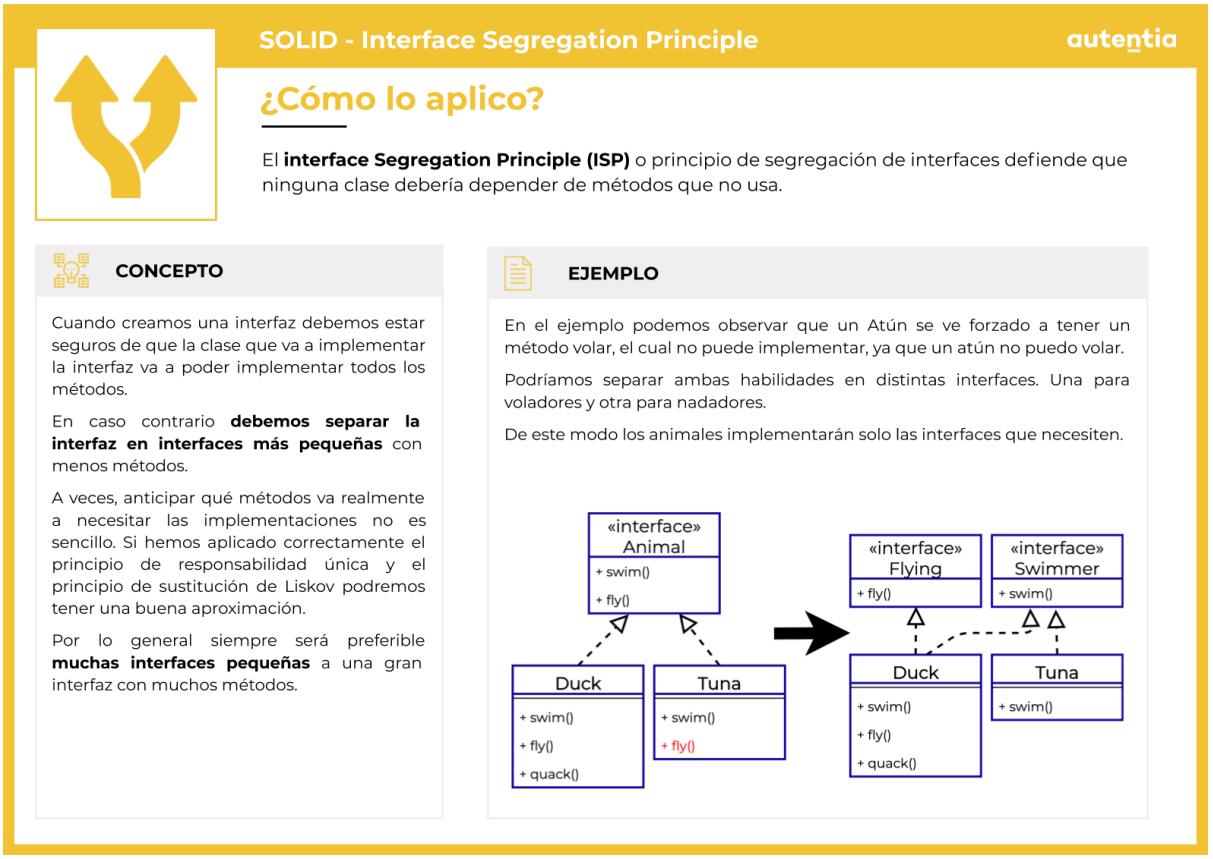
Interface segregation (ISP)

El principio de segregación de interfaces establece que **muchas interfaces cliente específicas son mejores que una interfaz de propósito general**. Cuando los clientes son forzados a utilizar interfaces que no usan por completo, están sujetos a cambios de esa interfaz. Esto al final resulta en un acoplamiento innecesario entre los clientes.

Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz, cuya funcionalidad este cliente no usa pero que otros clientes si, este cliente estará siendo afectado por los cambios que fuercen otros clientes en la clase en cuestión.

Debemos intentar evitar este tipo de acoplamiento cuando sea posible. Esto se consigue separando las interfaces en otras más pequeñas y específicas.

Para más detalle, ver el artículo *[S.O.L.I.D.] Interface Segregation Principle / Principio de segregación de interfaz* en Adictos al Trabajo.



Dependency inversion (DIP)

El principio de inversión de dependencia nos dice que **las entidades de software deben depender de abstracciones, no de implementaciones**. A su vez, los módulos de alto nivel no deberían depender de los de bajo nivel. Ambos deberían depender de abstracciones.

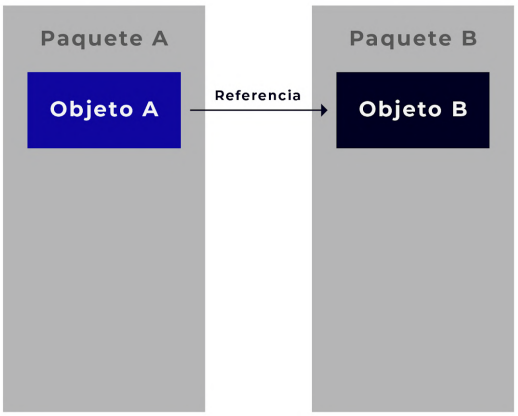


Figura 1

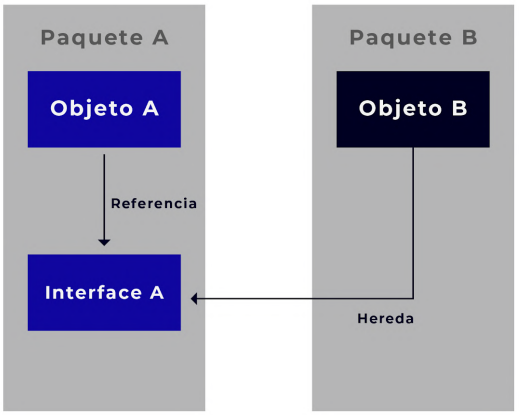


Figura 2

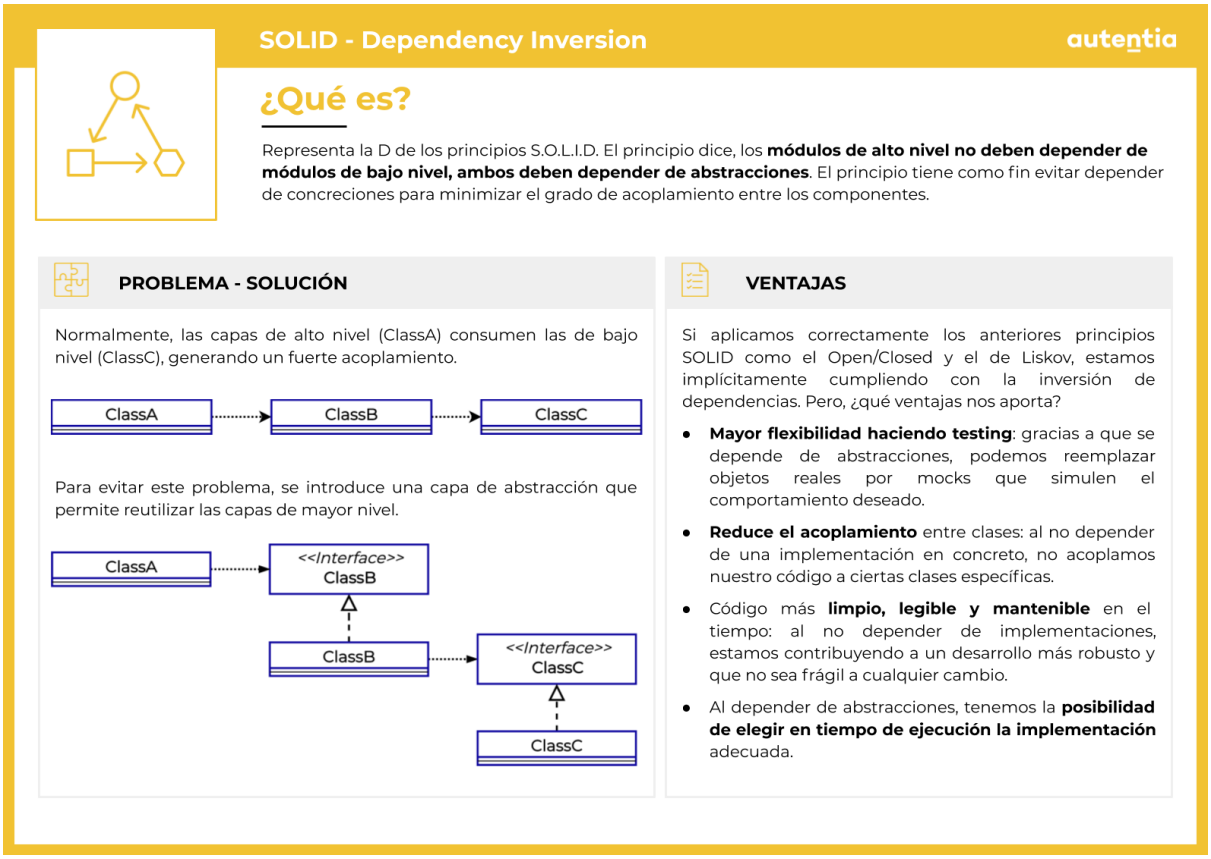
autentia

Mediante este principio ocultamos los detalles de implementación, ganando en flexibilidad. Cuando estamos haciendo tests, podemos reemplazar dependencias reales por objetos mockeados.

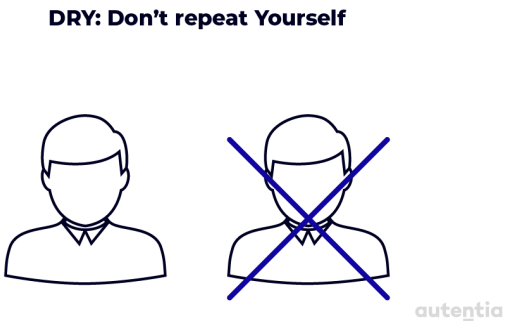
Gracias a esta flexibilidad, vamos a poder sustituir componentes sin que los clientes que los consumen se vean afectados ya que dependen de la abstracción y no de la implementación concreta.

Lo que se pretende es que no existan dependencias directas entre módulos, sino que dependan de abstracciones. De esta forma, nuestros módulos pueden ser más fácilmente reutilizables. Es fundamental que la abstracción se defina en base a las necesidades del módulo o cliente y no en las capacidades de la implementación, de lo contrario, la abstracción estaría bastante acoplada a la implementación teniendo así menos flexibilidad.

Para más detalle, ver el artículo *[S.O.L.I.D.] Dependency inversion principle / Principio de inversión de dependencias* en Adictos al Trabajo.



Don’t Repeat Yourself (DRY)



Su objetivo principal es **evitar la duplicación de lógica**. Cada pieza de funcionalidad debe tener una **identidad única**, **no ambigua** y **representativa** dentro del sistema.

Según este principio toda pieza de funcionalidad nunca debería estar duplicada ya que esta duplicidad incrementa la dificultad en los cambios y

su evolución posterior, puede perjudicar la claridad y crear un espacio para posibles inconsistencias.

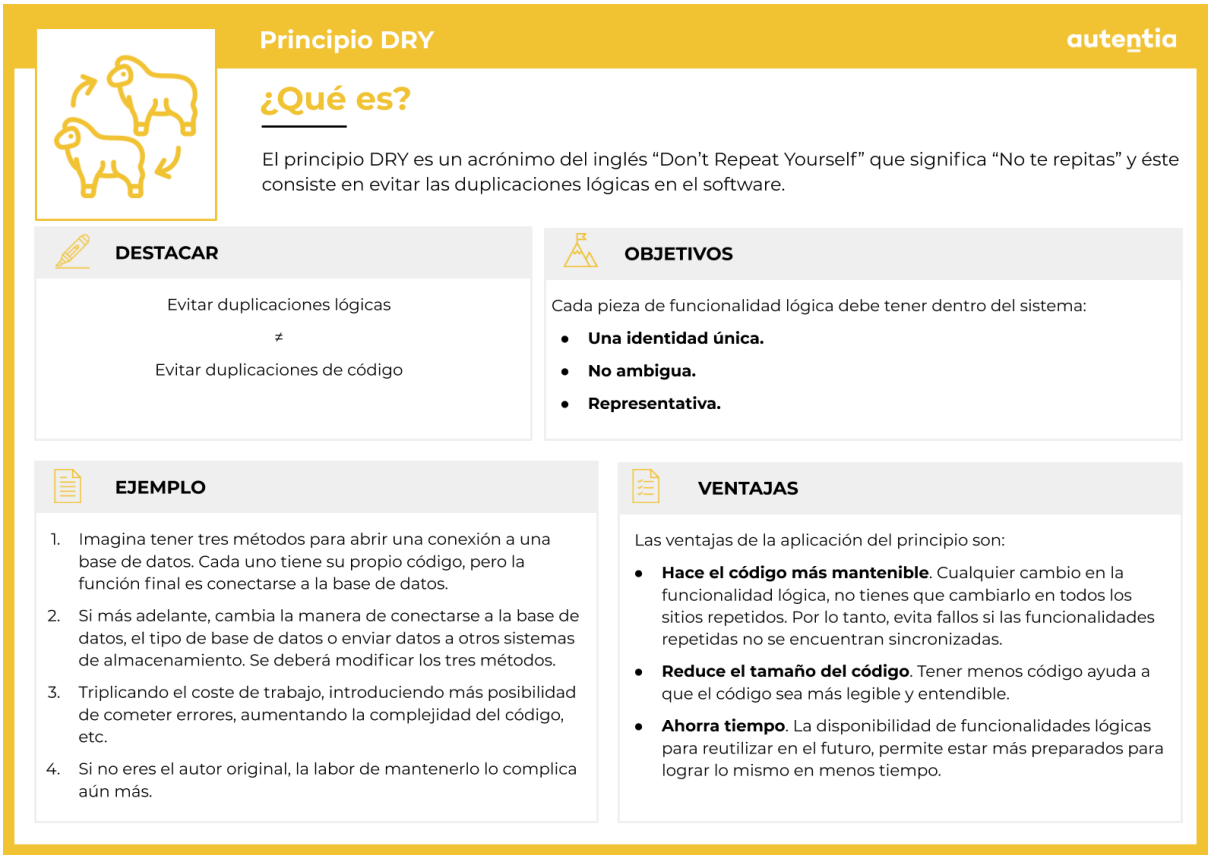
Por pieza de funcionalidad, no nos referimos a código sino a lógica, y más concretamente a función lógica. No es saludable tener tres métodos para abrir una conexión a una base de datos, cada uno con su propio código, si no hay un motivo que así lo justifique. Como se ve en este caso, los tres métodos pueden tener distinto código, pero la función final de los tres sigue siendo conectarse a la base de datos. Si en la evolución de nuestro software se nos solicita cambiar la forma de conectarnos, el tipo de base de datos o incluso enviar los datos a otros sistema de almacenamiento, deberemos modificar los tres métodos, lo que incrementa la cantidad de trabajo ya que debemos escribir el código tres veces y, consecuentemente, probarlo, introduce más posibilidades de cometer errores, aumenta la complejidad del código y más. Si además, no somos los autores originales de todo este código y solo nos toca mantenerlo, todo parece más difícil y laborioso aún.

Cuando el principio DRY se aplica de forma eficiente, los cambios en cualquier parte del proceso requieren cambios en un único lugar. Por el contrario, si algunas partes del proceso están repetidas por varios sitios, los cambios pueden provocar fallos con mayor facilidad si todos los sitios en los que aparece no se encuentran sincronizados.

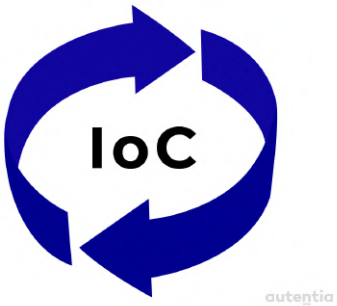
En resumen, ¿por qué es importante?

- **Hace el código más mantenible**. Evitar la repetición de lógica permite que si alguna vez cambia la funcionalidad en cuestión, no lo tengas que hacer en todos los lugares en los que lo repetiste.
- **Reduce el tamaño del código**. Esto lo hace más legible y entendible porque hay menos código.
- **Ahorra tiempo**. Al tener pedazos de lógica disponibles para reutilizarlos, en el futuro, estamos más preparados para lograr lo

mismo en menos tiempo.



Inversion of Control (IoC)



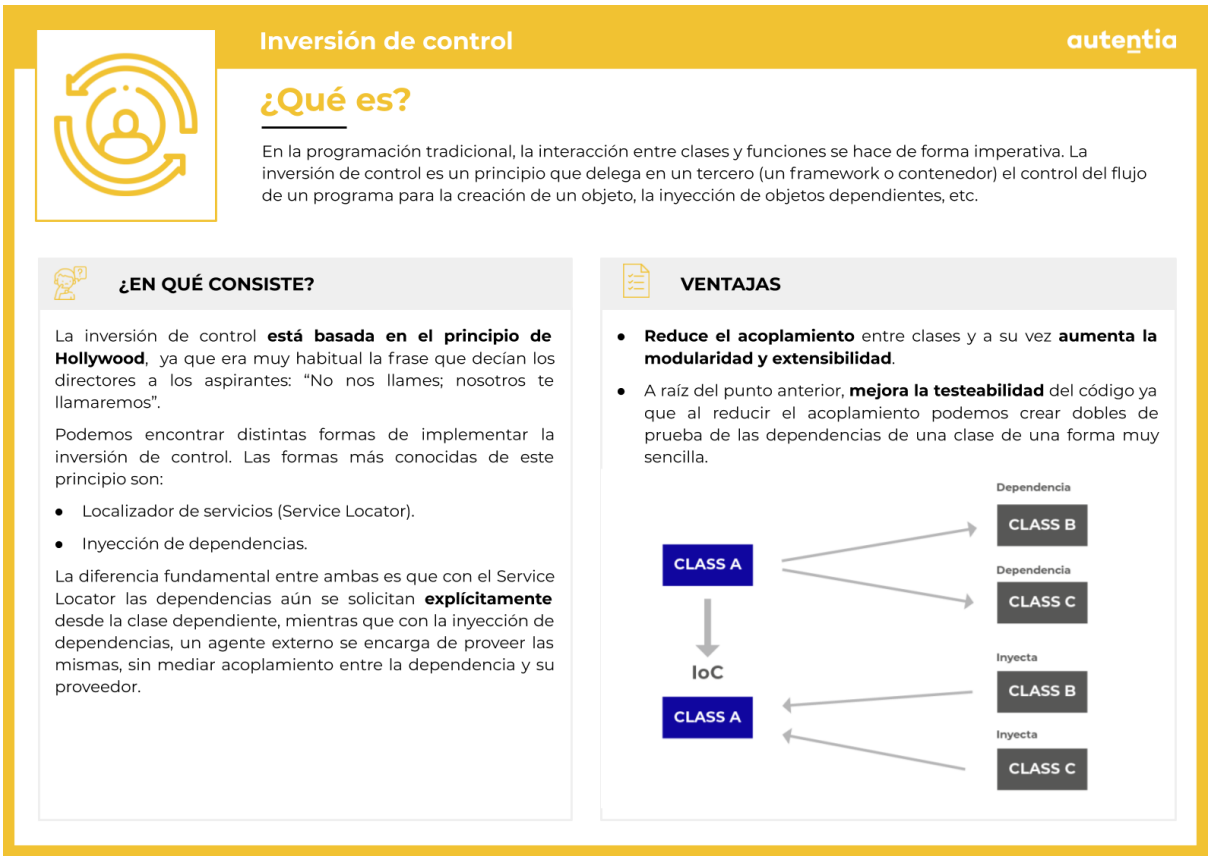
Como su nombre indica, “inversión de control”, se utiliza en el diseño orientado a objetos para **delegar en un tercero diferentes tipos de flujos de control** para lograr un bajo acoplamiento. Esto incluye el control sobre el flujo de una aplicación y el control sobre el flujo de la creación de un objeto o la creación y vinculación de objetos dependientes.

El principio de IoC ayuda a aumentar la modularidad del programa y al diseño de clases con bajo acoplamiento, lo que las hace testeables, mantenibles y extensibles.

Algunos patrones de diseño son implementaciones de este principio:

- Service locator.
- Dependency injection.
- Template method.
- Strategy.
- Abstract Factory.
- Observer.

A este principio se lo conoce también como *Don't call us, we'll call you* (No nos llame, nosotros lo llamamos) o *Hollywood Principle* (Principio de Hollywood).



You Aren't Gonna Need It (YAGNI)

Este principio, que podemos traducir como "No vas a necesitar eso", es un

principio que indica que **no se deben agregar funcionalidades extras** hasta **que no sea necesario**. La tentación de escribir código que no es necesario pero que puede serlo en un futuro, tiene varias desventajas, como el desperdicio del tiempo que se destinaría para la funcionalidad básica (las nuevas características deben ser depuradas, documentadas y soportadas) o que cuando se requieran las nuevas funcionalidades, estas no funcionen correctamente, ya que hasta que no está definido para qué se puede necesitar, es imposible saber qué debe hacer.

Ver el artículo *El principio YAGNI*, en Adictos al Trabajo, para más detalle.

Keep It Simple, Stupid (KISS)

El principio KISS (🤘💋) es un acrónimo que proviene de la frase inglesa “keep it simple, stupid”, que podemos traducir como “mantenlo simple, estúpido”. Se entiende como la necesidad de minimizar los errores tratando de realizar las tareas de forma efectiva y eficiente complicandose lo mínimo posible.

La simplicidad debe ser un objetivo clave tanto en el diseño, como en el desarrollo de la solución y se debe evitar la **complejidad innecesaria**.

Law of Demeter (LoD)

La Ley de Demeter, también conocida como el *Principle of least knowledge* o principio *Don’t talk to strangers* nos dice que un método de un objeto sólo debería interactuar con:

1. Métodos del propio objeto.
2. Los argumentos que recibe.
3. Cualquier objeto creado dentro del método.

4. Cualquier propiedad / campo directo dentro del propio objeto.

La idea principal es que un objeto **no tiene porqué conocer la estructura interna de los objetos con los que colabora**. En otras palabras, lo que se quiere evitar es el código con una estructura similar a la siguiente:

```
object.getX().getY().getZ().doSomething()
```

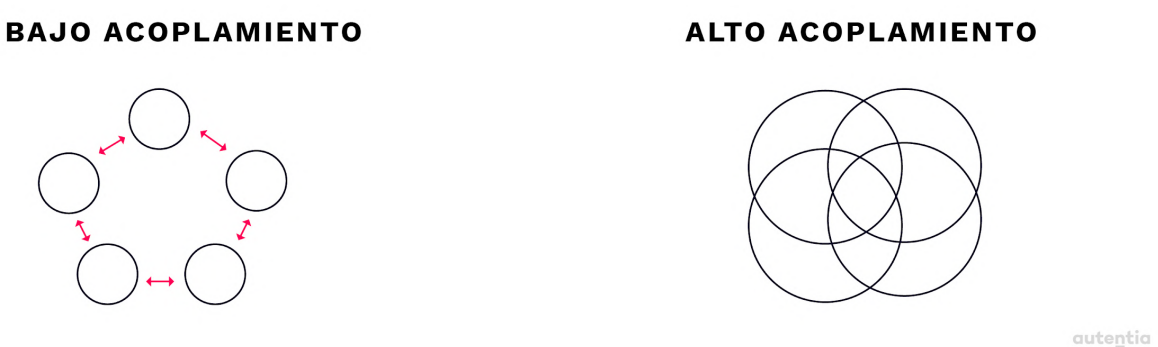
¿Cuál es el problema? Básicamente, la cadena denota un fuerte acoplamiento a la estructura de las clases involucradas en ella, afectándonos cualquier cambio o modificación en las mismas.

Entre las ventajas de aplicar este principio, encontramos:

- El software resultante tiende a ser más fácil de mantener y adaptar, ya que los objetos dependen menos de la estructura interna de otros objetos, lo que reduce el acoplamiento.
- Se vuelve más sencillo reutilizar las clases.
- El código es más fácil de probar.

Para más detalle, ver el artículo *Ley de Demeter* en Adictos al Trabajo.

Strive for loosely coupled design between objects that interact



En español “conseguir un diseño débilmente acoplado entre objetos que interactúan”. El acoplamiento se refiere al grado de conocimiento directo que un elemento tiene de otro.

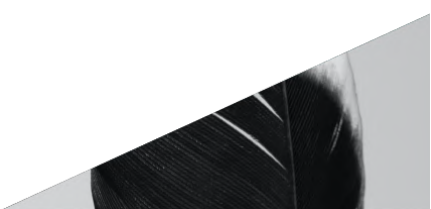
El objetivo es **reducir el riesgo de que un cambio en los objetos con los que interaccionamos provoque cambios en otros objetos**.

Limitar las interconexiones puede ayudar a aislar los problemas cuando las cosas salen mal y simplificar los procedimientos de prueba, mantenimiento y solución de problemas. Esto nos permite construir sistemas flexibles que pueden manejar los cambios porque reducen la dependencia entre múltiples objetos.

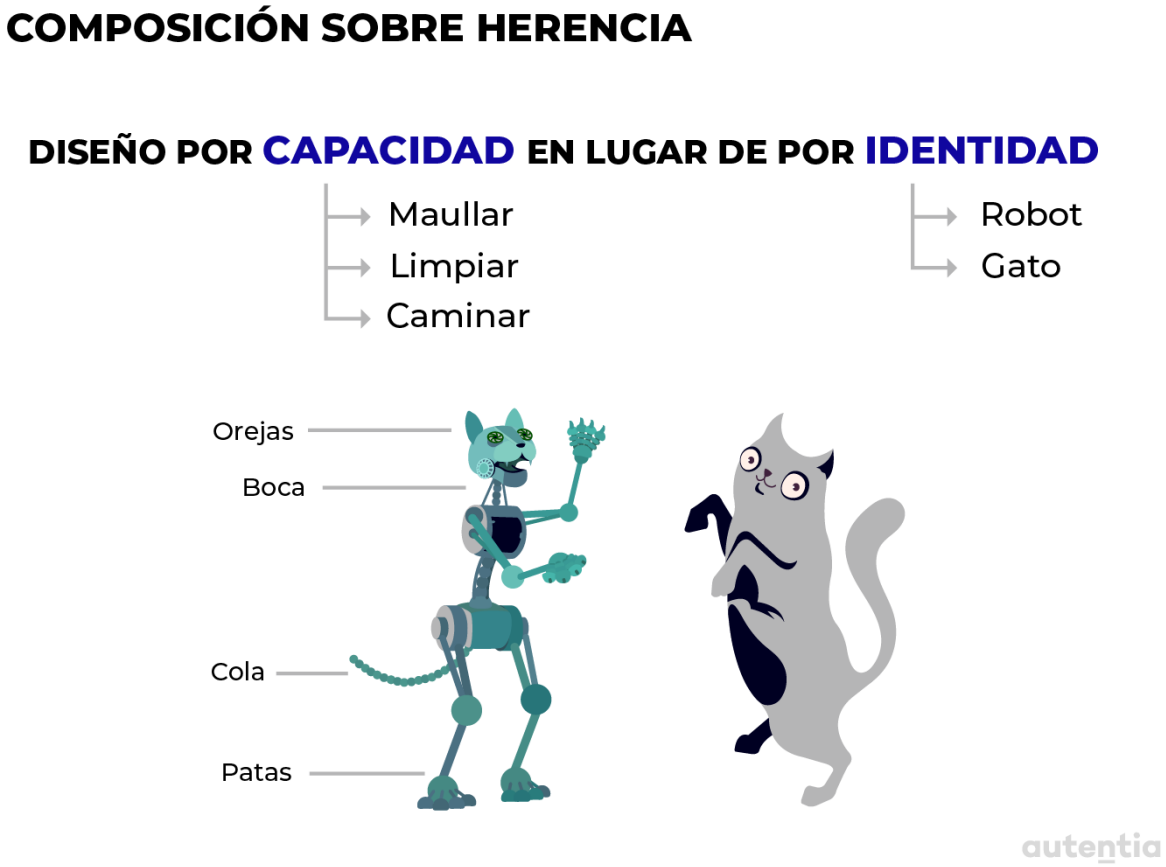
La arquitectura con bajo acoplamiento tiene las siguientes características:

- **Reduce el riesgo** de que un cambio en un elemento pueda provocar cambios en otros elementos.
- **Simplifica las pruebas**, el mantenimiento y la resolución de problemas.
- Los componentes en un sistema débilmente acoplado pueden **reemplazarse** con implementaciones alternativas que brinden los mismos servicios.

Un claro ejemplo de la implementación de este principio es el patrón *Observer*:



Composition over inheritance



El principio de composición sobre la herencia (también conocido como *composite reuse principle*) establece que las clases deben **lograr un comportamiento polimórfico y la reutilización del código mediante la composición** (al contener instancias de otras clases que implementan la funcionalidad deseada), en lugar de a través de la la herencia de una clase base o primaria, siempre que sea posible.

Con la herencia, estructuramos las clases alrededor de lo que son. Con composición, estructuramos las clases basándonos en lo que hacen. Al favorecer la composición sobre la herencia y **pensar en términos de lo que hacen las cosas** en lugar de lo que son, nos liberamos de estructuras de

herencia frágiles y estrechamente acopladas.

El gran problema con la herencia es que tendemos a predecir el futuro, a construir una estructura rígida con un fuerte acoplamiento entre clases padres e hijas en una etapa muy temprana del proyecto y lo más probable, es que cometamos errores de diseño al hacer esto, dado que no podemos predecir el futuro, y cambiar o salir de estas estructuras o taxonomías de herencia es mucho más difícil de lo que parece.

Al favorecer la composición sobre la herencia, dotamos al diseño de una mayor flexibilidad. Es más natural construir clases a partir de varios componentes que tratar de encontrar puntos en común entre ellos y crear un árbol genealógico. Por ejemplo, un pedal acelerador y un volante comparten muy pocos rasgos comunes, sin embargo, ambos son componentes vitales en un automóvil. Lo que pueden hacer y cómo se pueden utilizar para beneficiar al automóvil se define fácilmente. La composición también proporciona un dominio más estable a largo plazo, ya que es menos propenso a las peculiaridades de los miembros. En otras palabras, es mejor componer lo que un objeto puede hacer, verificando que se cumpla la relación *HAS-A* o *TIENE-UN*, que extender lo que es. Esto no significa que nunca se utilice la herencia, se puede implementar siempre que esta sea simple y tenga sentido dentro del modelo y, fundamentalmente, verificando siempre que se cumpla la relación *IS-A* o *ES-UN*.

El diseño inicial se simplifica identificando los comportamientos de los objetos del sistema en interfaces separadas, en lugar de crear una relación jerárquica para distribuir los comportamientos entre las clases a través de la herencia. Este enfoque es más flexible a cambios futuros que de otro modo requerirían una reestructuración completa de las clases de dominio en el modelo de herencia. Además, evita problemas a menudo asociados con cambios relativamente menores en un modelo basado en la herencia

que incluye varias generaciones de clases. Evitando posibles diseños donde se requiera herencia múltiple, ya que muchos lenguajes no la soportan.

Como desventaja, los diseños basados en un enfoque por composición son menos intuitivos.

Encapsulate what varies

Este principio se refiere a que cuando se identifiquen partes de la aplicación que **pueden cambiar**, se deben **aislar** y **encapsular en abstracciones** que permitan realizar el cambio sin afectar a otras partes de la aplicación.

Este principio se apoya en otros dos vistos en apartados anteriores como son *Single responsibility (SRP)* y *Open/closed (OCP)*.

Con la correcta aplicación de este principio se puede obtener dos beneficios fundamentales:

- Cuando una responsabilidad es correctamente acotada en un único módulo, variaciones en los requisitos de esa responsabilidad influyen únicamente en dicho módulo, reduciendo la fragilidad de nuestro sistema y aumentando su reusabilidad.
- La solicitud de nuevos requisitos o nuevos comportamientos se obtiene mediante la incorporación de nuevos elementos en lugar de la modificación de los elementos ya existentes, se reduce la rigidez de nuestro sistema (se vuelve más versátil y flexible) y se reduce también la fragilidad del mismo, ya que el código anterior (y por lo tanto probado) no se modifica.

La mayoría de los patrones de diseño se basan en estos principios. Algunos de estos patrones son:

- Abstract Factory.
- Factory Method.
- Adapter.
- Bridge.
- Decorator.
- Iterator.
- Observer.
- State.
- Strategy.
- Template Method.
- Visitor.

The four rules of simple design

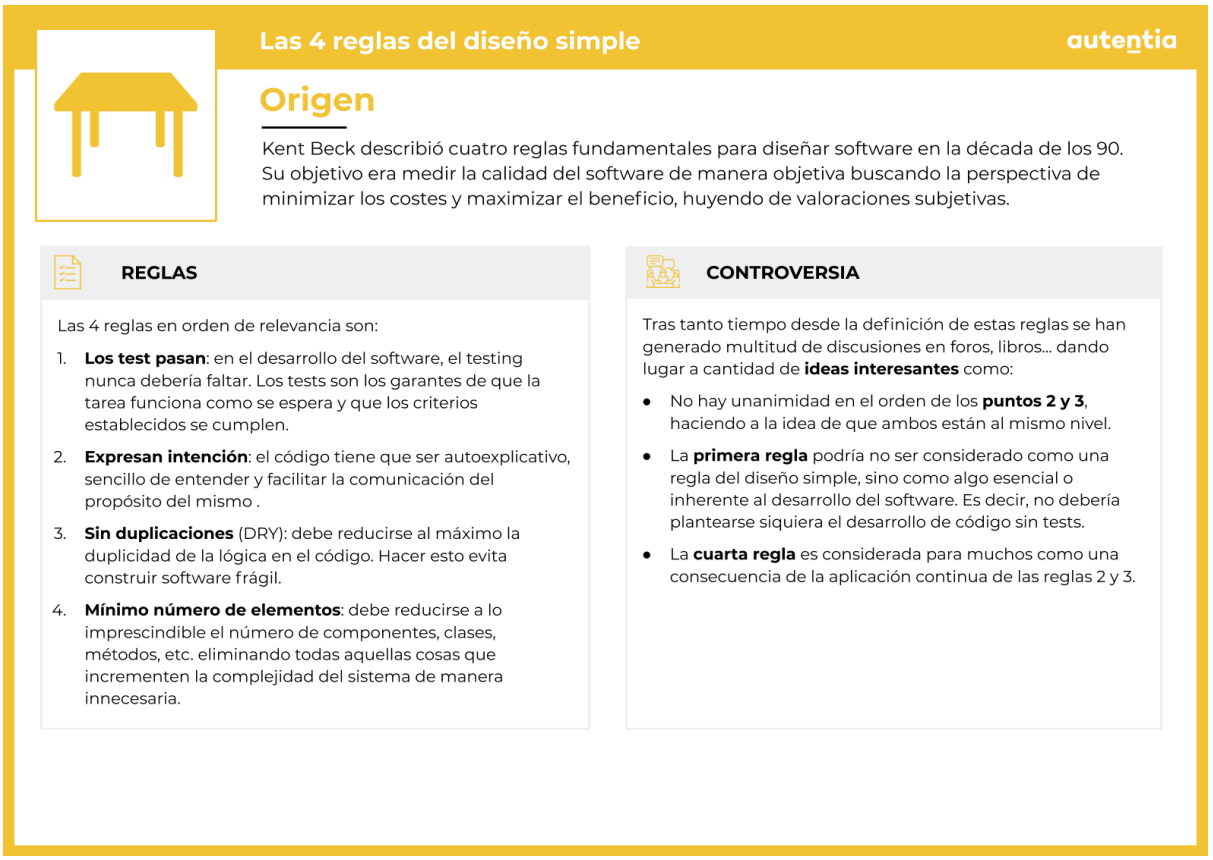
Hay 4 reglas que Kent Beck introdujo en los años 90 sobre los puntos fundamentales que se deben tener en cuenta a la hora de diseñar software, buscando una **manera objetiva** de poder **medir la calidad** de un diseño desde la perspectiva de minimizar los costes y maximizar el beneficio y huyendo de valoraciones subjetivas. Estas cuatro reglas sencillas de recordar están ordenadas por relevancia:

1. **Los tests pasan**: el testing es una pieza que no puede faltar cuando desarrollamos software. El objetivo principal es que cada tarea funcione de la manera esperada y que haya un/unos tests que verifican que estos criterios se cumplen.
2. **Expresan intención**: el código es autoexplicativo, fácil de entender y facilita la comunicación del propósito del mismo.
3. **No hay duplicidades** (DRY): se debe reducir al máximo la duplicidad de la lógica en el código, ya que de no hacerlo así, estaremos construyendo software frágil y cualquier cambio, por muy pequeño que sea, puede “romper” otras partes.

4. **Mínimo número de elementos**: se debe procurar reducir el número de componentes, clases, métodos, etc., a lo imprescindible, eliminando todas aquellas cosas que incrementen la complejidad del sistema de manera innecesaria.

Estas 4 reglas han sido discutidas en gran variedad de libros y foros diversos dando lugar a una buena cantidad de ideas interesantes al respecto, destacando entre otras:

- No hay unanimidad en el orden de los puntos 2 y 3, originando una idea generalizada de que ambos deberían estar en el mismo nivel de importancia.
- El primer punto podría no ser considerado siquiera como un punto del diseño simple, sino como algo esencial y connatural al desarrollo de software. Es decir, ni siquiera se debería plantear la posibilidad de un código sin tests.
- El último punto es considerado para muchos como una consecuencia de la continua aplicación de los puntos 2 y 3.



The boy scout rule

Los Boy Scouts tienen la regla de dejar el campamento más limpio de lo que se lo encontraron y en caso de ensuciarlo, se limpia y se deja lo mejor posible para la siguiente persona que venga. Si se aplica esto al desarrollo de software, se puede decir que si vemos alguna parte del código que se pueda mejorar, independientemente de quién lo haya hecho, debemos hacerlo.

El objetivo principal es mejorar la calidad del código y evitar su deterioro con el fin de ayudar al siguiente desarrollador (o a uno mismo dentro de un tiempo), a cambiar o desarrollar una nueva funcionalidad de una forma más sencilla. Se promueve el trabajo en equipo por encima de la individualidad, ya que no solo es importante la tarea que esa persona ha realizado, sino el proyecto en general y si se ve que algo se puede mejorar, se hace. La idea

es mejorar pequeñas partes de código de manera acotada y segura, ya que tampoco es cuestión de cambiar un módulo entero, sino poco a poco, ir mejorando su calidad.

Para aplicar esta regla, se deben tener claros los principios SOLID.

Para más detalle, ver el artículo *La regla del Boy Scout y la Oxidación del Software* en Adictos al Trabajo.

Last Responsible Moment

El desarrollo de software es una disciplina realmente curiosa. No es extraño, y me atrevería decir que es lo más habitual, encontrarse trabajando ya sobre las funcionalidades de un proyecto incipiente o añadiendo nuevas a otro ya más avanzado, sin tener aún claramente descritos los requisitos. Este principio propone como estrategia para abordar nuestros diseños, diferir nuestras decisiones, especialmente aquellas que se puedan considerar irreversibles, hasta el **último momento posible**. Este momento sería aquel en el que no tomar la decisión, supone un coste mayor que tomarla. Cuanto más tiempo mantengamos nuestras decisiones abiertas, más información iremos acumulando para optar por la decisión más adecuada.

Design Patterns

“Los patrones de diseño son descripciones de objetos y clases conectadas que se personalizan para resolver un problema de diseño general en un contexto particular”.

- Gang of Four

Los patrones de diseño ofrecen **soluciones comunes** a problemas recurrentes de diseño de aplicaciones. En la programación orientada a objetos, los patrones de diseño normalmente están dirigidos a resolver los problemas asociados con la creación e interacción de objetos, en lugar de los problemas a gran escala que afrontan las arquitecturas generales del software. Proporcionan soluciones generalizadas en forma de repeticiones que se pueden aplicar a problemas de la vida real.

Los patrones de diseño son soluciones útiles y probadas para los problemas que inevitablemente aparecen. No solo albergan años de conocimiento y experiencia colectiva, sino que además los patrones de diseño ofrecen un **vocabulario común** entre los desarrolladores y arrojan luz sobre muchos problemas.

Sin embargo, el uso innecesario o excesivo de patrones de diseño puede suponer también una sobre ingeniería, dando como resultado un sistema

excesivamente complejo que lejos de resolver los problemas, los aumenta, dando lugar a un diseño ineficiente, bajo rendimiento y problemas de mantenimiento.

Los patrones de diseño se clasificaron originalmente en tres grupos:

- Creacionales.
- Estructurales.
- De comportamiento.

Con el tiempo, fueron apareciendo nuevos patrones y con ellos, nuevas categorías de problemas que solucionan, por ejemplo, los patrones de concurrencia.

Patrones creacionales

Builder

Este patrón pretende separar la lógica de construcción de su representación. Para ello, define una clase abstracta, Builder, que es la encargada de crear las instancias de los objetos. Los elementos que intervienen son los siguientes:

- Builder: interfaz abstracta que crea los productos.
- Builder concreto: implementación concreta del builder que crea productos de un cierto tipo.
- Director: el encargado de utilizar la clase builder para crear los objetos.

Creacional - Builder

autentia

¿En qué consiste?

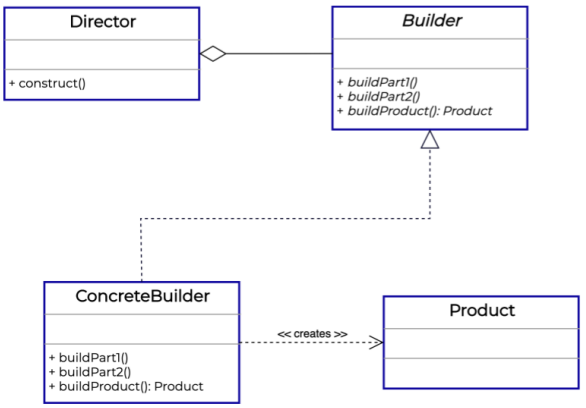
Patrón creacional que permite la **creación de diferentes representaciones de un objeto.** Se utiliza en situaciones en las que el objeto tiene una gran cantidad de atributos en el constructor por lo que la construcción se realiza en un conjunto de pasos.

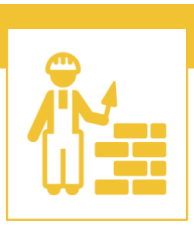
RAZONAMIENTO

Construir un objeto cuyo constructor tiene una larga lista de parámetros, puede llegar a ser tedioso. Sobretodo, porque debemos estar muy pendientes de cuál es el parámetro que va en cada posición. Otro problema es que a veces solo necesitamos construir el objeto con ciertas propiedades ya que las otras no son necesarias. La siguiente línea de código nos muestra un claro ejemplo del problema: **new Product(null, null, null, 'Madrid').** Un objeto Builder soluciona este problema simplificando la construcción y creando un objeto consistente.

También se suele usar este patrón para construir objetos inmutables por lo que la clase del objeto original no debe tener setters. Se tendrá un **'director'** que se encargue de crear siempre los objetos, de este modo, el cliente se abstrae de saber con qué propiedades se está construyendo dicho objeto.

Para el siguiente ejemplo no vamos a tener en cuenta esto, pero debemos conocer otro tipo de variaciones del patrón.





Creacional - Builder

autentia

Implementación

SOLUCIÓN

Builder organiza la construcción del objeto en una serie de pasos, siendo estos pasos básicamente los métodos que hay por cada atributo del objeto.

¿Qué ventajas tiene?

- **Mayor control** a la hora de construir un objeto.
- Se pueden construir objetos **inmutables**.

El cliente (Director) dirige la construcción del objeto User usando el UserBuilder:

```
User user = new UserBuilder().city("Madrid").build();
```

```
public class User {
    private String name;
    private String username;
    private Long age;
    private String city;

    public User(String name, String
username,   Long age, String city) {
        this.name = name;
        this.username = username;
        this.age = age;
        this.city = city;
    }
// getters...
}
```

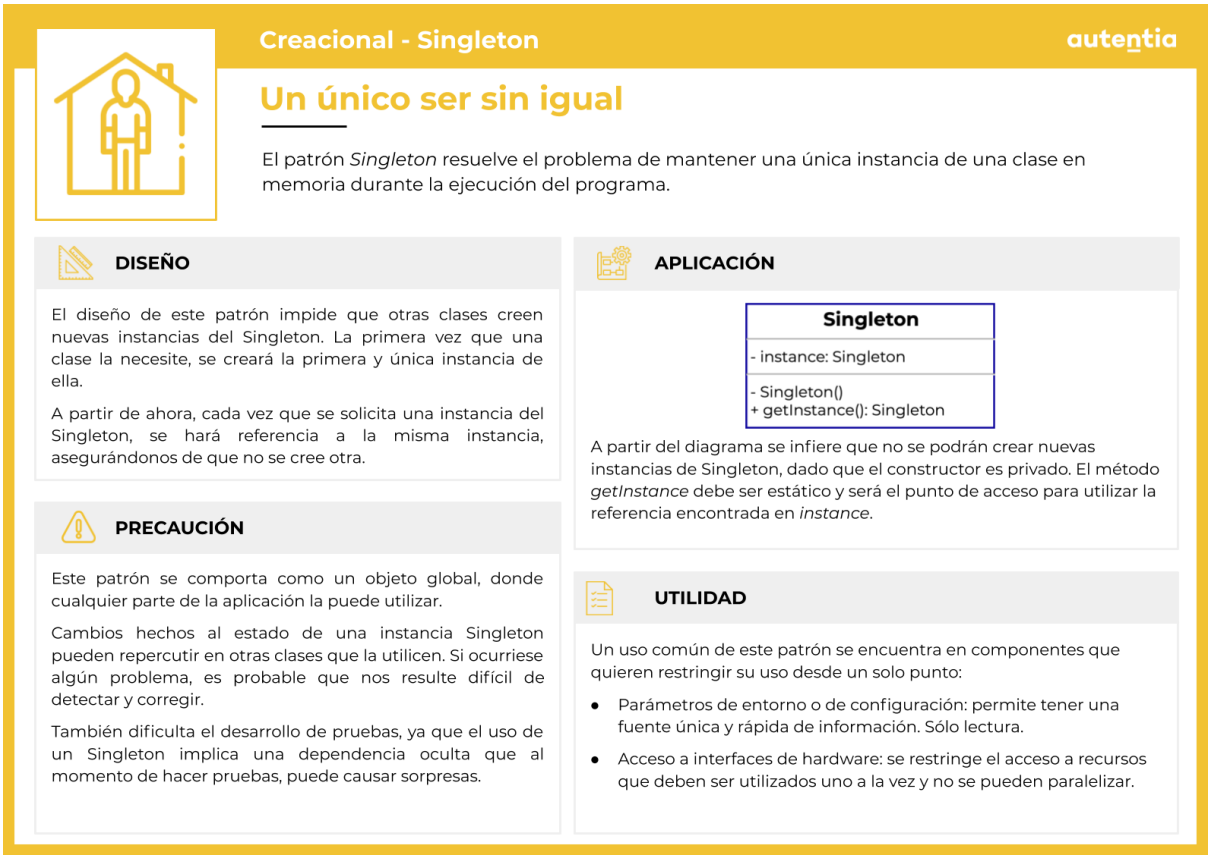
```
public class UserBuilder {
    private String name;
    private String username;
    private Long age;
    private String city;

    public UserBuilder name(String name) {
        this.name = name;
        return this;
    }

    public UserBuilder username(String username) {
        this.username = username;
        return this;
    }

    public UserBuilder age(Long age) {
        this.age = age;
        return this;
    }

    public UserBuilder city(String city) {
        this.city = city;
        return this;
    }

    public User build() {
        return new User(name, username, age, city);
    }
}
```

Singleton

Este patrón consiste en utilizar **una sola instancia** de clase, definiendo así un único punto global de acceso a ella. Dicha instancia es la encargada de la inicialización, creación y acceso a las propiedades de clase.

Este patrón es muy utilizado cuando se quiere controlar el acceso a un único recurso físico (fichero de lectura de uso exclusivo), o haya datos que deban estar disponibles para el resto de objetos de la aplicación (una instancia de log, por ejemplo).

Se define un método de acceso para recuperar la instancia de la clase. Este método también se encargará de crear la instancia en el caso de que se solicite por primera vez. Hay que prestar atención a los problemas que pudiera haber de acceso exclusivo.

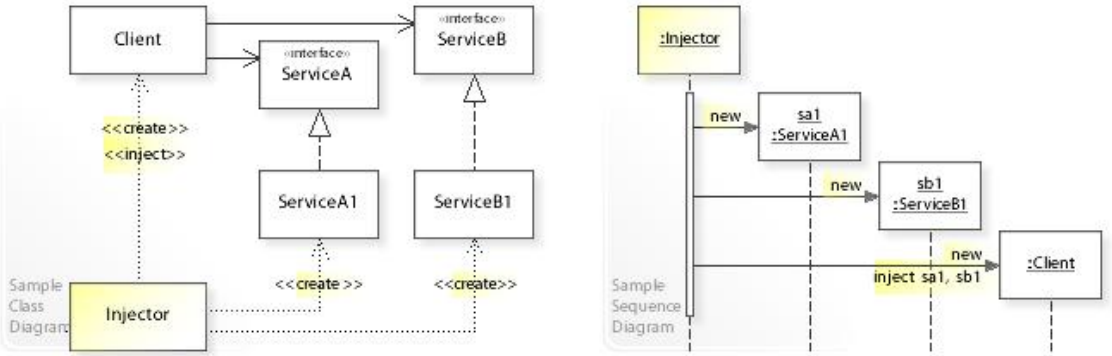


Dependency Injection

Se trata de un patrón de diseño que se encarga de extraer la responsabilidad de la **creación** de instancias de un componente para **delegarla** en otro. Permite que un objeto reciba otros objetos de los que depende, en lugar de ser el propio objeto el que los cree. Estos otros objetos se llaman dependencias. En la típica relación de "uso", el objeto receptor se llama cliente y el objeto pasado (es decir, "inyectado") se llama servicio. El código que pasa el servicio al cliente puede ser de muchos tipos y se llama inyector. En lugar de que el cliente especifique qué servicio usará, el inyector le dice al cliente qué servicio usar. La "inyección" se refiere al paso de una dependencia (un servicio), al objeto (un cliente) que lo usaría.

La inyección de dependencias es una forma de lograr la [inversión de control](#).

El cliente únicamente necesita conocer las interfaces de los servicios sin preocuparse de la implementación real de los mismos



By Vanderjoe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/wiki/File:W3sDesign_Dependency_Injection_Design_Pattern_UML.jpg

Para más detalle, ver el artículo [Patrón de Inyección de dependencias](#) en Adictos al Trabajo.

Service Locator

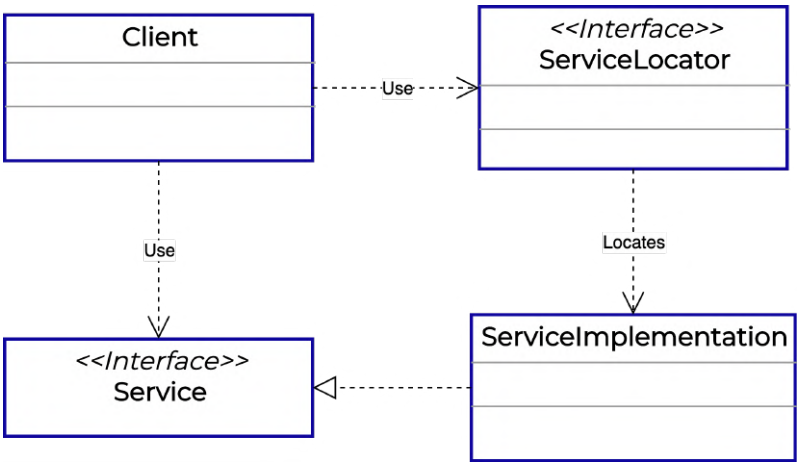
El patrón de localización de servicios es un patrón de diseño utilizado para encapsular los procesos involucrados en la obtención de un servicio con una capa de abstracción fuerte. Este patrón utiliza un registro central conocido como el "Service Locator" que, a demanda, devuelve el componente necesario para realizar una determinada tarea.

Se basa en la creación de una clase, llamada ServiceLocator, que sabe cómo crear las dependencias de otros tipos. A menudo, el localizador de servicios actúa como un depósito para objetos de servicios previamente inicializados. Cuando se requiere uno de estos servicios, se solicita el mismo llamando a un método determinado del ServiceLocator. En algunos casos, el método encargado de la localización de servicios crea instancias de objetos a medida que se necesitan.

Los defensores del patrón dicen que el enfoque simplifica las aplicaciones basadas en componentes, donde todas las dependencias se enumeran limpiamente al comienzo de todo el diseño de la aplicación, lo que hace que la inyección de dependencias tradicional sea una forma más compleja de conectar objetos. Los críticos del patrón argumentan que el software es más difícil de probar.

La principal diferencia frente a la inyección de dependencias es que en este caso hay una solicitud explícita para obtener la dependencia mientras que en la inyección de dependencias la obtención viene ya dada.

Este patrón es otra implementación del principio de inversión de control (IoC).



Abstract Factory

El propósito de Abstract Factory es proporcionar una interfaz para **crear familias** de **objetos** relacionados, sin especificar clases concretas. Normalmente, el cliente crea una implementación concreta de la fábrica abstracta y luego utiliza la interfaz genérica de la misma para crear los objetos concretos. El cliente no sabe (ni le importa) qué objetos concretos obtiene de cada una de estas fábricas internas, ya que utiliza solo las interfaces genéricas de sus productos. Este patrón separa los detalles de la implementación de un conjunto de objetos de su uso general y se basa en la composición del objeto, ya que la creación de objetos se implementa en los métodos expuestos en la interfaz de fábrica.

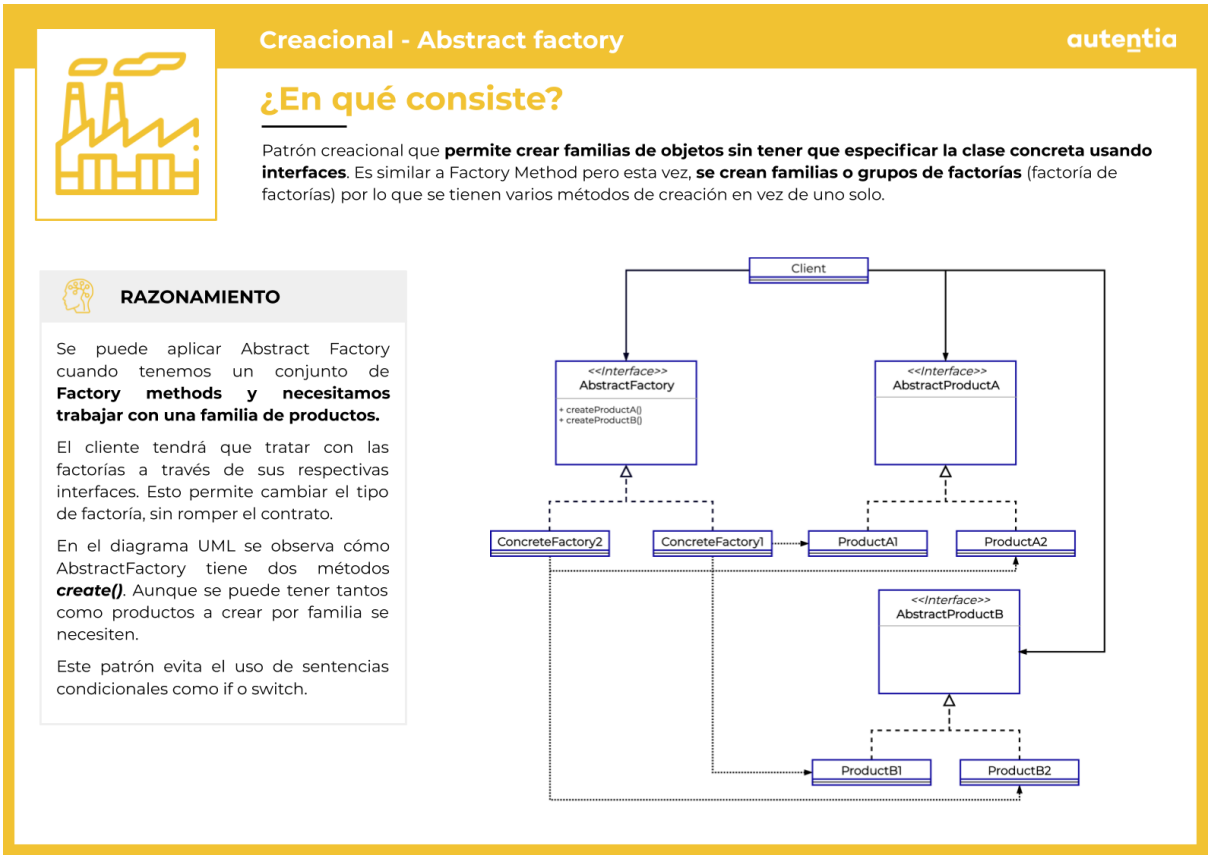
La estructura típica del patrón Abstract Factory es la siguiente:

- Cliente: la clase que llamará a la factoría adecuada ya que necesita crear uno de los objetos que provee la factoría.
- Abstract Factory: es la definición de las interfaces de las factorías. Debe de proveer un método para la obtención de cada objeto que pueda crear.
- Factorías Concretas: estas son las diferentes familias de productos. Provee la instancia concreta del tipo de objeto que se encarga de

crear.

- Producto abstracto: definición de las interfaces para la familia de productos genéricos.
- Producto concreto: implementación de los diferentes productos.

Este patrón es otra implementación del principio de inversión de control (IoC).



Creacional - Abstract factory

autentia

Implementación (1/2)

SOLUCIÓN

Tenemos **dos tipos de 'familia'.** La familia Google y la familia Microsoft. Cada familia tiene el producto Mail y el producto CloudStorage, por lo que aplicando **polimorfismo** podemos crear implementaciones específicas para cada producto.

```
public interface CloudStorage {
   String show();
}

public class GoogleCloudStorage implements CloudStorage {
   @Override
   public String show() {
       return "show Google cloud info";
   }
}

public class MicrosoftCloudStorage implements CloudStorage{
   @Override
   public String show() {
       return "show Microsoft cloud info";
   }
}
```

```
public interface Mail {
   String show();
}

public class GoogleMail implements Mail {
   @Override
   public String show() {
       return "show Google mail info";
   }
}

public class MicrosoftMail implements Mail {
   @Override
   public String show() {
       return "show Microsoft mail info";
   }
}
```

Creacional - Abstract factory

autentia

Implementación (2/2)

SOLUCIÓN

La interfaz **AbstractFactory** declara un conjunto de métodos para **crear los productos** (Mail y CloudStorage), pero no sabemos a qué familia pertenecen, solo queremos que mediante esta abstracción, se puedan crear productos (en general), independientemente de si son de una familia u otra.

Dicho esto, la firma de los métodos debe devolver la interfaz correspondiente, de esta manera, el código no se acopla a una única implementación y el cliente se aísla de los detalles.

Se debe crear una implementación de AbstractFactory **por cada familia.**

¿Qué ventajas tiene?

- **Reduce el acoplamiento**.
- Aplica el **Principio de Responsabilidad Unica.**
- Aplica el **Principio de Abierto a la extensión y cerrado a la modificación.**

```
public interface AbstractFactory {
   Mail createMail();
   CloudStorage createCloudStorage();
}

public class GoogleFactory implements AbstractFactory {
   @Override
   public Mail createMail() {
       return new GoogleMail();
   }

   @Override
   public CloudStorage createCloudStorage() {
       return new GoogleCloudStorage();
   }
}

public class MicrosoftFactory implements AbstractFactory {
   @Override
   public Mail createMail() {
       return new MicrosoftMail();
   }

   @Override
   public CloudStorage createCloudStorage() {
       return new MicrosoftCloudStorage();
   }
}
```

Factory Method

Provee una interfaz o clase abstracta (creator) que permite encapsular la lógica de creación de los objetos en subclases y éstas deciden qué clase instanciar. Los objetos se crean a partir de un método (factory method) y no a través de un constructor como se hace normalmente. Además, los ConcreteCreators devuelven siempre la interfaz (Product), esto permite que el cliente trate a los productos por igual, tengan una implementación u otra.

La estructura típica del patrón Factory method es la siguiente:

- Product: definición de las interfaces para la familia de productos genéricos.
- ConcreteProduct: implementación de los diferentes productos.
- Creator: declara el factory method que se encargará de instanciar nuevos objetos. Es importante que este método devuelva la interfaz Product. Normalmente el Creator suele ser una clase abstracta con cierta lógica de negocio relacionada con los productos a crear. Dependiendo de la instancia de producto que se devuelva, se puede seguir un flujo u otro.
- ConcreteCreator: crea la instancia del producto concreto.

Creacional - Factory method

autentia

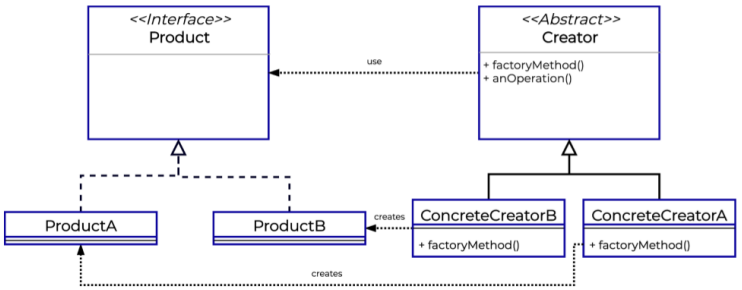
¿En qué consiste?

Patrón creacional que provee **una interfaz o clase abstracta**(creator) que permite encapsular la lógica de creación de los objetos en subclases. Las subclases deciden qué clase instanciar. **Los objetos se crean a partir de un método** y no a través de un constructor como se hace normalmente.

RAZONAMIENTO

Al contrario que Simple Factory, que instancia todos los objetos en la misma clase y no hay subclases, **Factory Method crea una implementación o subclase por cada producto**.

En el ejemplo, tenemos como productos distintos tipos de animales. Todos ellos implementan la interfaz ***Animal*** para que siempre se dependa de una abstracción y nunca de una concreción.



Creacional - Factory method

autentia

Implementación

SOLUCIÓN

AnimalFactory tiene un método **createAnimal (factory method)** y cada implementación se encarga de la creación de sus objetos. Podríamos incluso crear una factoría **DomesticAnimalFactory** que se encargue de crear solo aquellos animales considerados como domésticos.

Todas las factorías implementan la misma interfaz, por lo que gracias al **polimorfismo** podemos crear tantas implementaciones como necesitemos o cambiar la implementación de una factoría sin que la clase que la use se vea afectada.

¿Qué ventajas tiene?

- **Reduce el acoplamiento y encapsula** el código encargado de crear objetos.
- Aplica el **Principio de Responsabilidad Unica.**
- Aplica el **Principio de Abierto a la extensión y cerrado a la modificación.**

Se puede hacer una analogía de este patrón a la programación declarativa. Quiero una instancia del objeto X, pero, cómo se construya por debajo o cómo esté implementado no es mi responsabilidad.

```java
public interface Animal {}

public class Cat implements Animal {}

public class Cocodrile implements Animal {}

public class Dog implements Animal {}

public class Lion implements Animal {}


public interface AnimalFactory {
   Animal createAnimal();
}

public class WildAnimalFactory implements AnimalFactory {
   @Override
   public Animal createAnimal() {
       // creates only wild animals
   }
}

public class RandomAnimalFactory implements AnimalFactory {
   @Override
   public Animal createAnimal() {
       // creates random animals
   }
}
```

Patrones estructurales

Adapter

El libro [GoF] indica que este patrón "proporciona una **interfaz unificada** a un conjunto de interfaces en un subsistema". Head First Design Patterns da la misma explicación y señala que convierte la interfaz de una clase en otra interfaz que los clientes esperan. El adaptador permite que las clases puedan trabajar juntas ya que de otro modo, no podrían debido a tener interfaces incompatibles.

En el libro [GoF] se nos describen dos tipos principales de adaptadores:

- Adaptadores de clase: generalmente usan herencia múltiple o varias interfaces para implementarlo.
- Adaptadores de objetos: realizan las composiciones de objetos para adaptarlos.

Un adaptador se puede considerar como la aplicación del principio de inversión de dependencias (DIP), cuando la clase de alto nivel define su propia interfaz (adaptador) para el módulo de bajo nivel (implementado por una clase adaptada).

Estructurales - Adaptador

autentia

¿Qué es?

El patrón adaptador actúa como un **conector entre dos interfaces que son incompatibles y que no pueden estar conectadas directamente**.

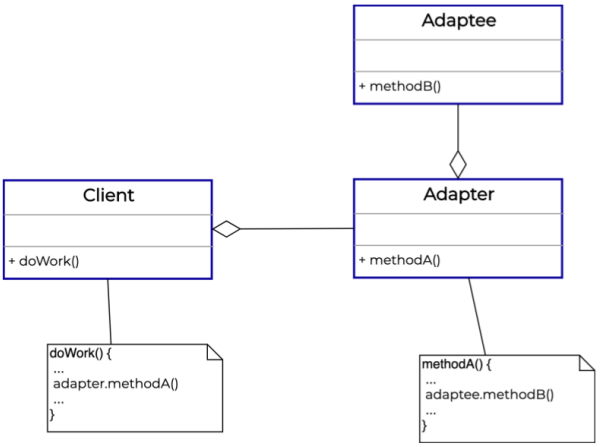
CONCEPTO

Sean dos interfaces A y B incompatibles entre sí existiendo la necesidad de que A llame a B. **Este patrón permite a través de una interfaz adaptador, envolver el contenido de la interfaz B de modo que pueda ser llamada por A**. En el diagrama de la derecha, la clase *Client* interactúa con la clase Adapter para poder utilizar *Adaptee*.

El patrón adaptador es conveniente utilizarlo cuando:

- Un componente tercero ya proporciona una funcionalidad que se puede integrar en nuestro sistema pero es incompatible.
- La aplicación no es compatible con la interfaz que espera consumir el cliente.
- Existe código *legacy* que se quiere utilizar en la aplicación sin tener que hacer ningún cambio sobre él.

Dentro del desarrollo de software este adaptador es también conocido como *wrapper*.



Estructurales - Adaptador

autentia

Implementación

EJEMPLO

Consideremos el escenario en el que tenemos una aplicación diseñada en Estados Unidos que devuelve el precio de un coche en dólares. Queremos utilizar esta funcionalidad dentro de nuestra aplicación pero los precios los tenemos que devolver en euros.

Para ello, vamos a construir una interfaz, *AdapterPricer* que envuelva a la interfaz *Pricer*, que es la que devuelve el precio en dólares.

Las implementaciones de este adaptador tendrán como dependencia la instancia de Pricer que corresponda. Se implementará el método getPrice() de *AdapterPricer* de modo que tras recuperar el precio en dólares de Pricer haga la conversión para devolver el precio en euros.

```
public interface Pricer {
    // Return price in
    double getPrice();
}


public class FerrariPricer implements
Pricer {
    // Return price in $
    double getPrice() {
        return 2000000;
    }
}
```

```
public interface AdapterPricer {
    // Return price in
    double getPrice();
}

public class AdapterPricerImpl implements
AdapterPricer {
    private final Pricer pricer;

    //Constructor

    public double getPrice() {
      ...
      double priceInDolars = pricer.getPrice();
      double priceInEuro =
convertToEuros(priceInDolars);
    }

    private double convertToEuros(double price){
       //Returns price in euros
    }
}
```

```
public class Main {
    public static void main(String[] args) {
      final FerrariPricer ferrariPricer = new FerrariPricer();
      final AdapterPricer ferrariPricerAdapter = new AdapterPricerImpl(ferrariPricer);
      ...
      double ferrariEuroPrice = ferrariPricerAdapter.getPrice();
      // ferrari price in Euros!!
    }
}
```

Data Access Object (DAO)

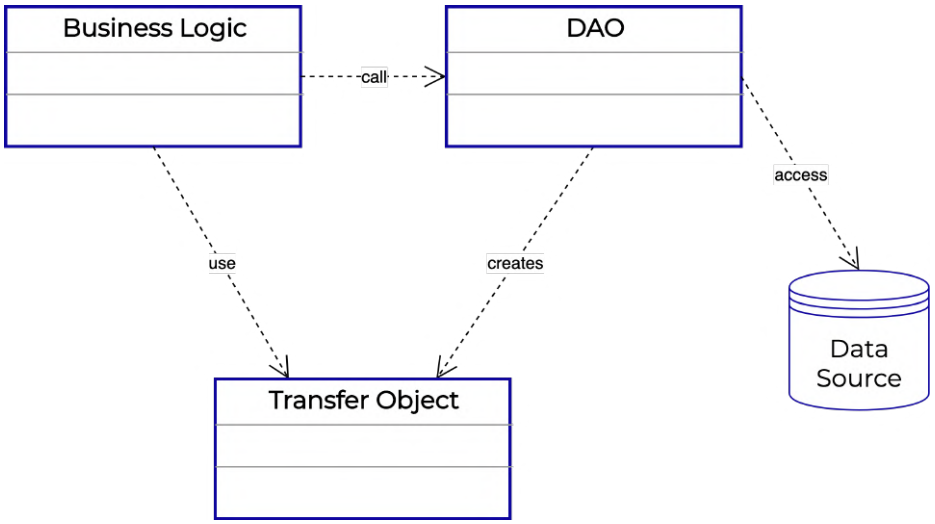
La solución original que propuso el patrón DAO se definió en el libro Core J2EE Patterns: Best Practices and Design Strategies de la siguiente manera:

“Se usa el Data Access Object (DAO) para **abstraer y encapsular todo el acceso a la fuente de datos**. El DAO gestiona la conexión con la fuente de datos para obtener y almacenar datos”.

El problema que se resolvió con la abstracción y la encapsulación de la fuente de datos, fue evitar que la aplicación dependiera de la implementación de la fuente de datos. Esto desacopla la capa de negocio de la fuente de datos.

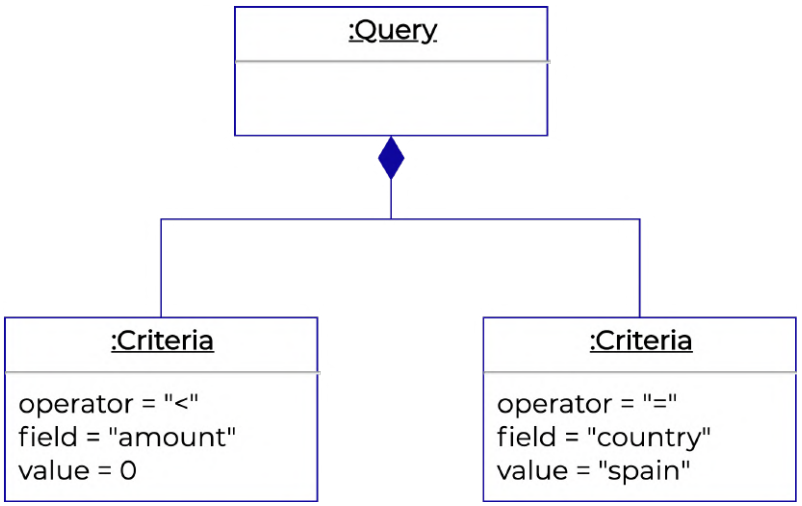
Aunque originariamente permitía protegerse frente a cambios en el motor de base de datos, el patrón DAO sigue siendo un patrón valioso y su solución original, sigue siendo válida. En lugar de protegerse contra el impacto de un cambio improbable en el tipo de fuente de datos, el valor está en su capacidad de prueba y su uso para estructurar el código y mantenerlo limpio de código de acceso a datos.

El patrón DAO encapsula las operaciones de acceso a los datos en una interfaz implementada por una clase en concreto. Si se mockea esta clase, se pueden probar las clases de negocio sin hacer conexiones a la base de datos. La implementación concreta de un DAO utiliza API de bajo nivel para realizar las operaciones de acceso a datos.



Query Object

Este patrón se puede consultar en el libro Patterns of Enterprise Application Architecture. Un Query Object es un **intérprete** [GoF], es decir, una estructura de objetos que puede **formar** una consulta **SQL**. Puede crear esta consulta haciendo referencia a clases y campos en lugar de tablas y columnas. De esta forma, quienes escriben las consultas pueden hacerlo independientemente del esquema de la base de datos y los cambios en el esquema, se pueden localizar en un solo lugar.



Decorator

El propósito de este patrón es el de asignar **responsabilidades adicionales** a un objeto **dinámicamente**, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

La estructura está compuesta por:

- Component: define la interfaz que deben implementar los objetos a los que se les pueden añadir funcionalidades.
- ConcreteComponent: define un objeto al cual se le pueden agregar responsabilidades adicionales. Implementa la interfaz Component.
- Decorator: mantiene una referencia al Component asociado. Implementa la interfaz de la super clase Component, delegando en el Component asociado. El Decorator, en general, añade comportamiento antes o después de un método que ya existe en el Component.
- ConcreteDecorator: añade responsabilidades al Component.

Entre las ventajas de implementar este patrón, podemos encontrar:

- Es más flexible que la herencia.
- Permite añadir y eliminar responsabilidades en tiempo de ejecución.
- Evita la herencia con muchas clases y la herencia múltiple.
- Limita la responsabilidad de los componentes para evitar clases con excesivas responsabilidades en los niveles superiores de la jerarquía.

Estructurales - Decorator

autentia

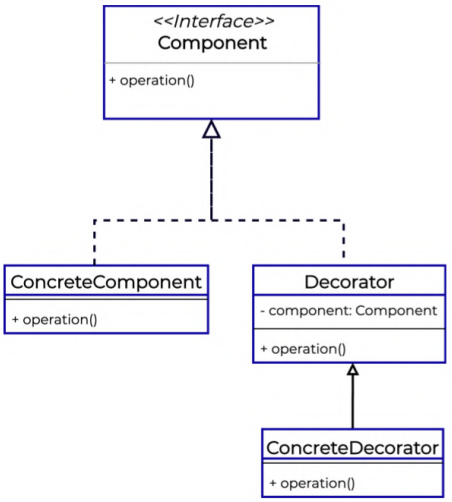
¿En qué consiste?

Patrón que **permite añadir nuevas funcionalidades a un objeto en tiempo de ejecución sin modificar su estructura** y a través de una envoltura (wrapper). El decorador envuelve la clase original sin cambiar la firma de los métodos existentes.

CONCEPTO

Decorator ofrece una alternativa cuando no es posible extender el comportamiento de un objeto a través de la herencia.

Normalmente tenemos una interfaz con varias implementaciones. Para aplicar este patrón, debemos crear una nueva 'implementación' que será nuestro *Decorator*. A partir de esta clase, creamos clases concretas de Decorator con las nuevas funcionalidades que se desean añadir.



Estructurales - Decorator

autentia

Implementación

SOLUCIÓN

Definimos la interfaz *Text* con su implementación *BaseText* que define un comportamiento básico que podrá ser modificado por un decorator. Importante que la clase *TextDecorator* tenga un campo de tipo Text ya que será el que envolvamos para su posterior modificación. *BoldTextDecorador*, *Italic* y *Underline* definen el comportamiento que se va a añadir dinámicamente al componente, en nuestro caso, a *BaseText*. Desde la perspectiva del cliente, los objetos son iguales.

Ventajas de usar este patrón:

- **Añade o elimina funcionalidades de forma flexible** a un objeto en tiempo de ejecución.
- Sigue el **principio Open/Closed**.
- Permite envolver un objeto en varios decorators.

```java
public interface Text {
    void write();
}

public class BaseText implements Text {
    @Override
    public void write() {
        System.out.println("some basic text");
    }
}
```

```java
public class TextDecorator implements Text {
    private Text decoratedText;

    public TextDecorator(Text decoratedText) {
        this.decoratedText = decoratedText;
    }

    @Override
    public void write() {
        decoratedText.write();
    }
}
```

```java
public class BoldTextDecorator extends
TextDecorator {

    public BoldTextDecorator(Text decoratedText) {
        super(decoratedText);
    }

    @Override
    public void write() {
        System.out.println("adding bold style to
text");
        super.write();
    }
}

/*
 Another class for ItalicTextDecorator...
 Another class for UnderlineTextDecorator...
*/
```

```java
public class Main {
    public static void main(String[] args) {
        Text baseText = new BaseText();
        baseText.write();

        Text boldText = new BoldTextDecorator(baseText);
        boldText.write();

        Text italicText = new ItalicTextDecorator(baseText);
        italicText.write();

        //...
    }
}
```

Bridge

Tiene como objetivo **desacoplar** la **abstracción** de la **implementación**. Permite que la abstracción y la implementación se desarrollen de forma independiente a través de un puente (bridge) entre ambas. El código del cliente podrá acceder a la abstracción sin preocuparse por la parte de implementación.

Participantes:

- Abstraction: recibe por parámetro la interfaz que servirá de puente con las implementaciones y es la que se comunicará con el cliente.
- ConcreteAbstraction: puede trabajar con distintas implementaciones a través de la interfaz.
- Implementator: declara una interfaz con sus respectivos métodos que serán los que actúen de enlace entre la abstracción y las implementaciones.
- ConcreteImplementator: contiene código concreto sobre cada implementación.

Entre las ventajas de implementar este patrón podemos encontrar:

- El cliente siempre trabaja con abstracciones y nunca con implementaciones.
- Cumple con el principio Open/Closed ya que se pueden añadir nuevas implementaciones independientes unas de otras.
- Permite reducir el número de subclases que si usaramos herencia pura.

Estructurales - Bridge

autentia

¿En qué consiste?

Es un patrón que busca **desacoplar la abstracción de su implementación**. Permite que la abstracción y la implementación se desarrollen de forma independiente y el código del cliente solo puede acceder a la abstracción sin preocuparse por la parte de implementación. Se puede pensar en una **abstracción con dos capas.**

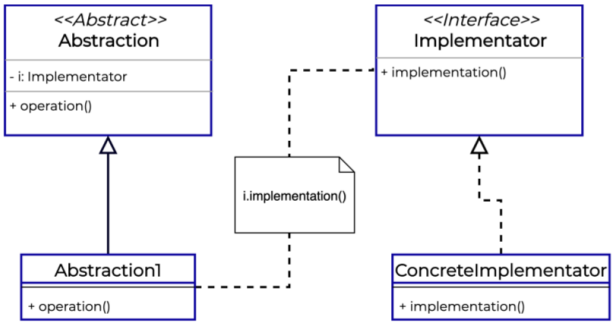
CONCEPTO

Este patrón normalmente define una abstracción y son las implementaciones las que cambian, pero hay casos en los que desde un principio, no estamos seguros del alcance de la abstracción y sabemos que va a ir evolucionando. Para poder **hacer cambios a la abstracción, sin estar rompiendo las implementaciones continuamente,** usamos este patrón.

Este patrón es útil en las siguientes situaciones:

- Queremos que una abstracción y su implementación se definan y extiendan de forma independiente.
- Queremos desacoplar la abstracción y su implementación para poder cambiarla en ejecución.

En la figura se ve la clase Abstraction, que tendrá una referencia a *Implementator*. El método operation() de *Abstraction1* puede trabajar con distintas implementaciones de la interfaz *Implementator*.



Estructurales - Bridge

autentia

Implementación

EJEMPLO

Implementamos la clase abstracta ***RemoteControl*** a la cual podemos inyectar una ***TV***. La ***TV***, en nuestro caso, será el implementator del patrón **Bridge**.

El modo en el que inyectamos el implementator no es relevante para el patrón. Si necesitamos cambiar la implementación en **runtime** podríamos hacerlo mediante un setter en vez de utilizar el constructor.

Tenemos un RemoteControl específico llamado ***ModernRemoteControl*** que tiene la posibilidad adicional de cambiar de canal a la posición siguiente o anterior.

Como ***TV*** podemos tener una digital y otra analógica. De esta forma tenemos **dos jerarquías totalmente independientes** y podemos separar la evolución de ***RemoteControl*** de las implementaciones concretas de ***TV*** para las que se utilizan.

```
public abstract class RemoteControl {

    private final TV implementator;
    private final int currentChannel = 0;

    public RemoteControl(TV implementator) {
        this.implementator = implementator;
    }

    protected final void setChannel(int channel) {
        implementator.setChannel(channel);
        this.currentChannel = channel;
    }

    protected final int getCurrentChannel() {
        return this.currentChannel;
    }
}
```

```
public interface TV {
    setChannel();
}

public class DigitalTV implements TV {
    @Override
    protected final setChannel(int channel) {
        // Sintoniza de forma digital
    }
}

public class AnalogTV implements TV {
    @Override
    protected final setChannel(int channel) {
        // Sintoniza de forma analógica
    }
}
```

```
public class ModernRemoteControl extends RemoteControl {

    public ModernRemoteControl(TV implementator) {
        super(implementator);
    }

    public void previousChannel(int channel) {
        this.setChannel(this.getCurrentChannel() - 1);
    }

    public void nextChannel(int channel) {
        this.setChannel(this.getCurrentChannel() + 1);
    }
}
```

```
public static void main(String[] args) {
    TV implementator = new DigitalTV();
    RemoteControl remoteControl = new ModernRemoteControl(implementator);
    remoteControl.setChannel(5);
}
```

Patrones de comportamiento

Command

El libro The Gang of Four [GoF] indica que se usa el patrón de Comando para "**encapsular** una **solicitud**" como un objeto, permitiendo definir una interfaz común para invocar acciones diversas.

Simplificando, el objetivo de un comando es ejecutar una serie de acciones en su receptor (Receiver). El cliente crea un objeto Command y generalmente, le pasa el Receiver para poder acceder a él. Cuando el Client desea ejecutar el Command, se utiliza el Invoker que almacena el Command y se encarga de iniciar su ejecución en algún momento, invocando al método execute del Command.

Esto permite añadir otras funcionalidades a las acciones como encolamiento, registro, acciones de deshacer o rehacer las operaciones, etc., gracias a **desacoplar** la **solicitud** de una acción de su ejecución.

Comportamiento - Command

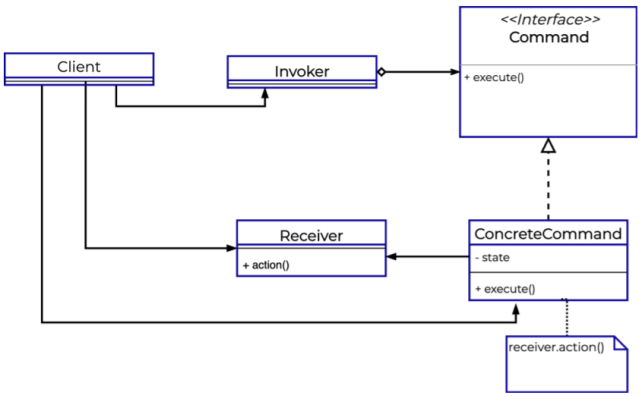
autentia

¿En qué consiste?

Patrón que **permite encapsular una petición en un objeto** que contiene toda la información necesaria para realizar una acción (un comando).

CARACTERÍSTICAS

- **Command:** objeto que contiene toda la información necesaria para realizar una acción específica.
- **Receiver:** objeto que realiza las operaciones cuando se ejecuta el comando.
- **Invoker**: objeto encargado de ejecutar la acción, pero desconoce la implementación del comando. Él únicamente recibe la interfaz y llama al método *execute*, aunque también se encarga de gestionar una cola de comandos (en caso de que se necesite) o de realizar la acción de revertir (si fuera necesario). Invoker actúa como mediador y permite desacoplar a los consumidores del objeto comando.
- **Client:** objeto que controla el proceso de ejecución de los comandos. Instancia los comandos deseados y se los pasa al Invoker.



Comportamiento - Command

autentia

Implementación (1/2)

```
public interface Command {
   void execute();
   void undo();
}
```

SOLUCIÓN

Se han creado solo dos clases como *Receivers* para simplificar el ejemplo, pero se podrían haber creado más como *Television*, *Radio*, *Heater*, *Cooker* etc., cada una con sus respectivas acciones. Las clases command tienen como atributo su respectivo receiver y se encargan de ejecutar y revertir la acciones pertinentes.

Se podría también, tener una clase TurnOffAllDevicesCommand donde a través del constructor nos llegue una lista de electrodomesticos y se ejecute la acción de apagarlos todos.

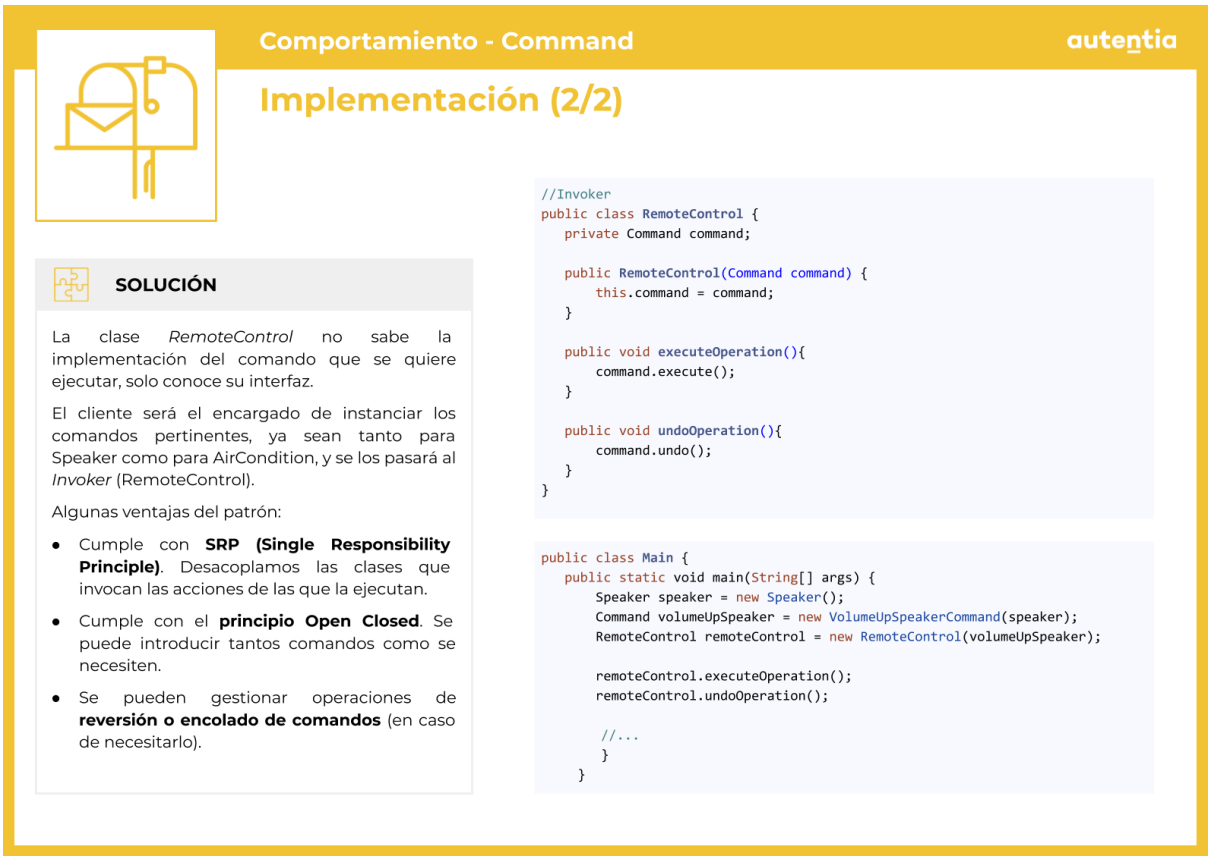
Una pequeña desventaja que se puede observar es el incremento del número de clases por comando.

```
//Receiver
public class Speaker {
   public String turnUpVolume() {
       return "turning up volume...";
   }

   public String turnDownVolume() {
       return "turning down volume...";
   }
}
```

```
//Receiver
public class AirCondition {
   public String on() {
       return "A.C is on";
   }

   public String off() {
       return "A.C is off";
   }
}
```

```
public class VolumeDownSpeakerCommand
implements Command {
   private Speaker speaker;

   //Constructor

   @Override
   public void execute() {
       speaker.turnDownVolume();
   }

   @Override
   public void undo() {
       speaker.turnUpVolume();
   }
}
//Similar para clase AirConditionOnCommand
```

```
public class VolumeUpSpeakerCommand
implements Command {
   private Speaker speaker;

   //Constructor

   @Override
   public void execute() {
       speaker.turnUpVolume();
   }

   @Override
   public void undo() {
       speaker.turnDownVolume();
   }
}
// similar para clase AirConditionOffCommand
```



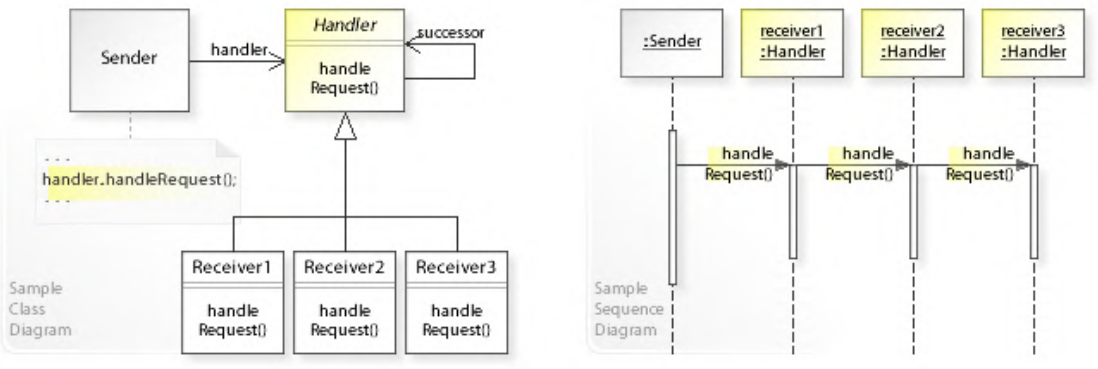
Chain of Responsibility

El libro The Gang of Four [GoF] indica que se usa el patrón "evitar acoplar el remitente de una solicitud a su receptor, al darle a **más de un objeto** la oportunidad de **manejar** la **solicitud**. Se encadenan los objetos receptores y pasa la solicitud a lo largo de la cadena hasta que un receptor lo maneja."

Aquí se procesan una serie de receptores uno por uno (es decir, de forma secuencial). Una fuente iniciará este procesamiento. Con este patrón, constituimos una cadena donde cada uno de los objetos de receptores puede tener cierta lógica para manejar un tipo particular de objeto. Una vez que se realiza el procesamiento, si aún hay algo pendiente, se puede reenviar al siguiente receptor de la cadena.

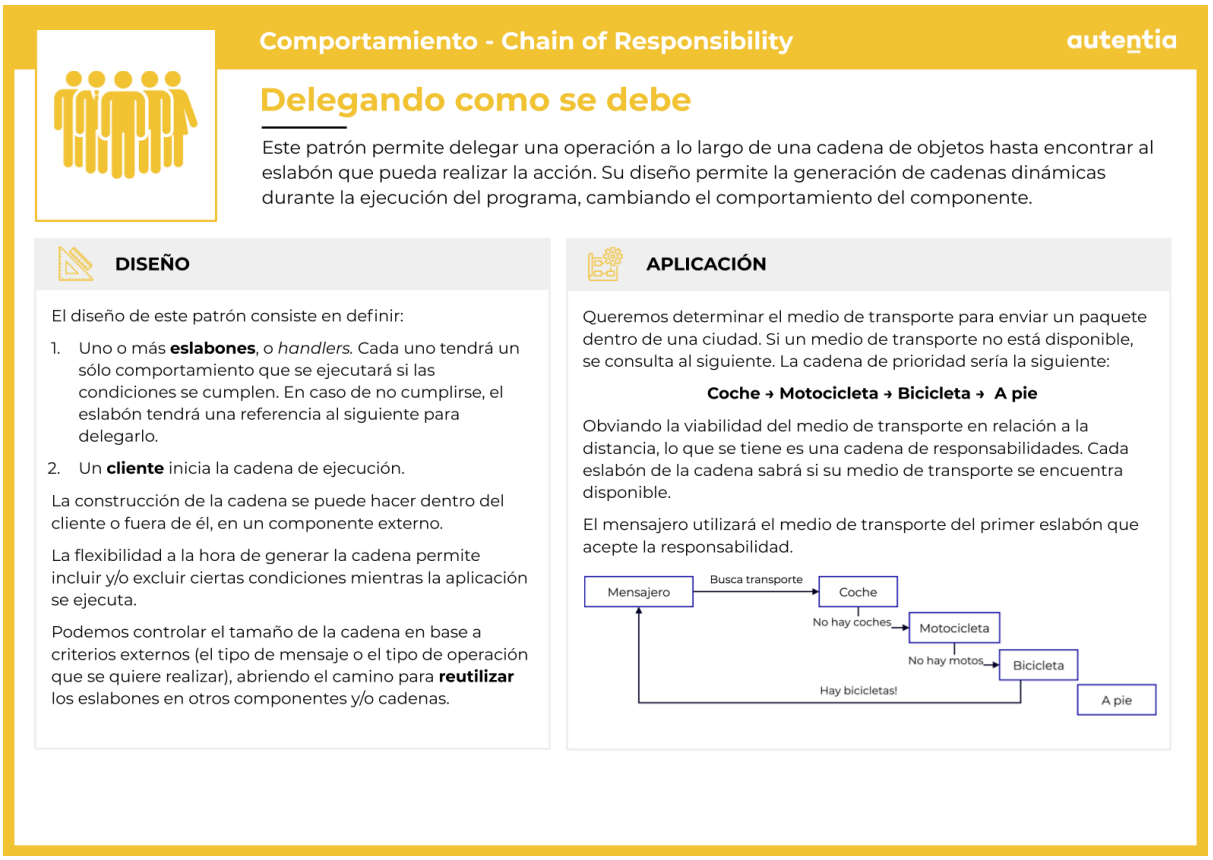
Cabe indicar que este tipo de patrón establece una jerarquía entre los receptores, pues los primeros en la cadena tienen prioridad sobre los

siguientes. Podemos agregar nuevos receptores en cualquier momento al final de una cadena.



By Vanderjoe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=62530468

Este patrón representa una implementación del concepto definido en el principio de responsabilidad única (SRP).



Strategy

El libro The Gang of Four [GoF] indica que se usa este patrón para definir una familia de **algoritmos** que se encapsulan cada uno de forma que sean **intercambiables**. El patrón estrategia permite que el algoritmo varíe independientemente de un cliente a otro.

La clave para aplicar el patrón Strategy es diseñar interfaces para la estrategia y su contexto, que sean lo suficientemente generales como para admitir una variedad de algoritmos. No debería tener que cambiar la estrategia o la interfaz de contexto para admitir un nuevo algoritmo.

Según el patrón de estrategia, los comportamientos de una clase no deben heredarse. En su lugar, deben encapsularse utilizando interfaces. Esto es compatible con el principio Open/Closed (OCP), que propone que las clases deben estar abiertas para la extensión pero cerradas para la modificación.

Para más detalle, ver el artículo Utilizando el patrón Estrategia en Adictos al Trabajo.

Comportamiento - Estrategia autentia

¿Qué es?

El patrón estrategia define una familia de algoritmos, quedando encapsulados y pudiendo intercambiarse entre ellos. Los algoritmos quedan independientes de los clientes que los usan.

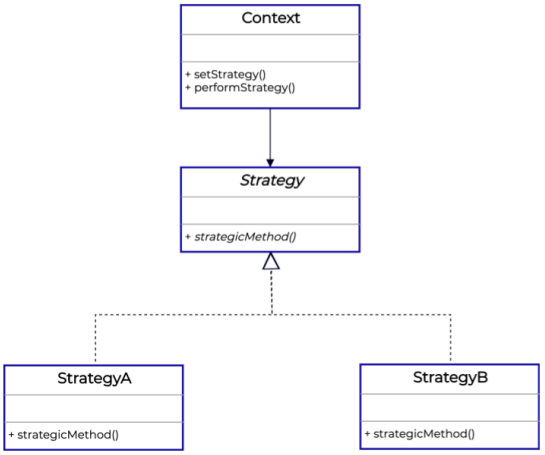
CONCEPTO

Este patrón resulta útil cuando tenemos un problema que se puede resolver de varias formas y queremos la libertad de elegir la forma de hacerlo en ejecución.

Se **define una familia de algoritmos (o estrategias) y el objeto cliente elige entre ellos.** Puede haber cualquier número de estrategias y todas implementan la misma interfaz, por lo que son intercambiables entre sí, incluso en tiempo de ejecución.

De este modo, si hay información del algoritmo que los clientes no deberían saber, ésta va a quedar encapsulada fuera del código del cliente.

Este **patrón es compatible con el principio abierto/cerrado (OCP)**, que propone que las clases deben estar abiertas para extensión y cerradas para modificación. Se añade comportamiento creando nuevas estrategias pero no se modifica el código existente.



Comportamiento - Estrategia autentia

Implementación

EJEMPLO

Queremos encontrar la ruta para llegar de un punto a otro utilizando los algoritmos de búsqueda de caminos más famosos. Mediante el patrón estrategia vamos a crear la interfaz PathfindingStrategy que en el método **find()** va a hacer los cálculos oportunos para encontrar una ruta.

Tenemos dos implementaciones que extienden la interfaz con los algoritmos más reconocidos para resolver este problema: A* y Dijkstra.

La clase de **Contexto es el cliente de la estrategia**. La estrategia se puede cambiar en cualquier punto de la ejecución, gracias a que tiene un setter disponible.

```
interface PathfindingStrategy {
     public void find();
}

public class AStarAlgorithm implements
PathfindingStrategy {
     public void find() {
         System.out.println("Has usado A*");
     }
}

public class DijkstraAlgorithm implements
PathfindingStrategy {
     public void find() {
         System.out.println("Has usado
Dijkstra");
     }
}
```

```
public class Context {
    PathfindingStrategy c;

    public Context(Strategy c){
        this.c = c;
    }

    public void setStrategy(Strategy c)
{
        this.c = c;
    }

    public void performTask(){
        c.find();
    }
}
```

```
public class Main {
    public static void main(String args[]) {
      // Usamos el algoritmo A*
      PathfindingStrategy aStar = new AStarAlgorithm();
      Context context = new Context(aStar);
      context.performTask();
      // Usamos el algoritmo de Dijkstra
      PathfindingStrategy dijsktra = new DijkstraAlgorithm();
      context.setStrategy(dijsktra);
      context.performTask();
      // Volvemos al algoritmo A*
      context.setStrategy(aStar);
      context.performTask();
    }
}
```

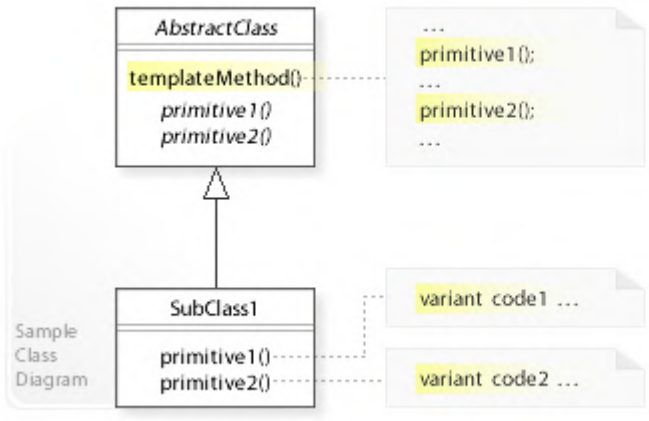
Template Method

El libro The Gang of Four [GoF] indica que se usa este patrón para definir el **esqueleto** de un **algoritmo** en una operación, **delegando** algunos **pasos** a subclases. El método plantilla permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.

En un template method, definimos la estructura mínima o esencial de un algoritmo. Luego diferimos algunas funcionalidades (responsabilidades) a las subclases. Como resultado, podemos redefinir ciertos pasos de un algoritmo manteniendo la estructura clave fija para ese algoritmo.

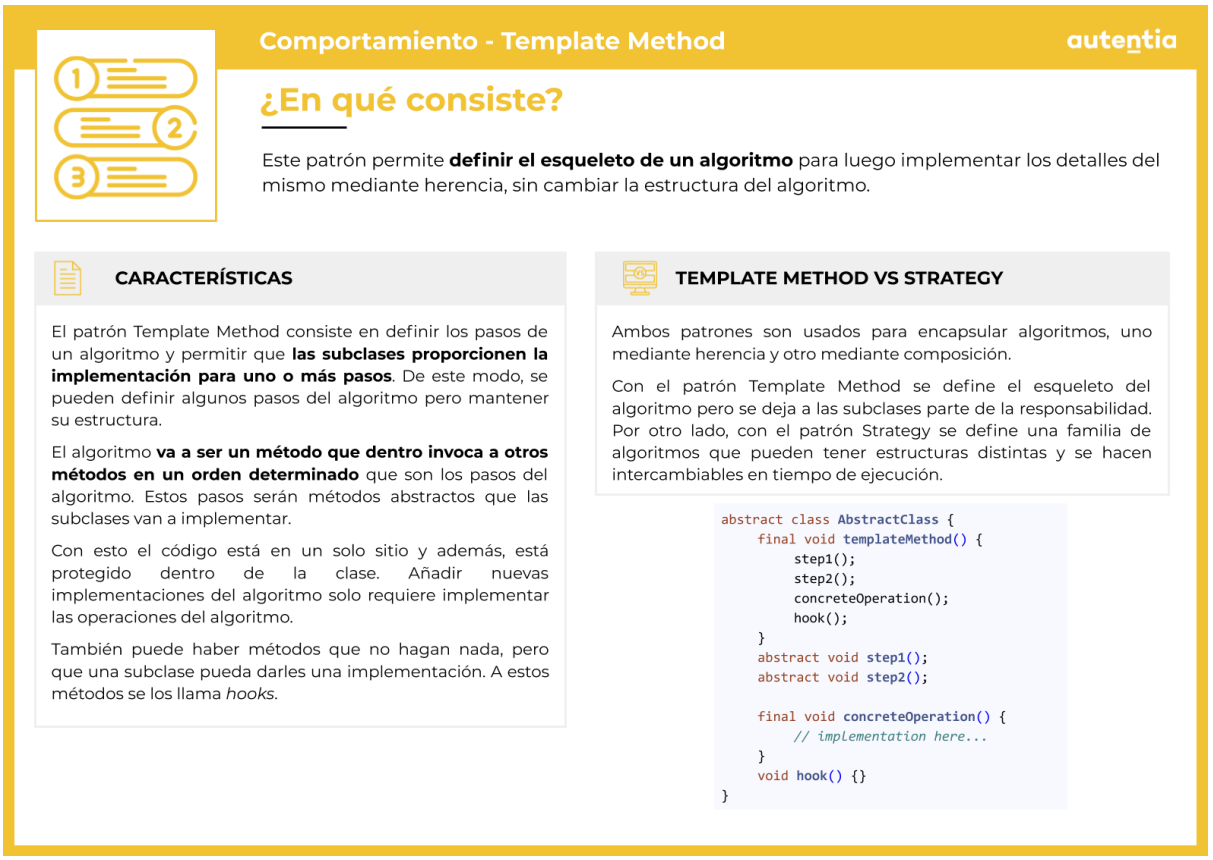
En tiempo de ejecución, el algoritmo representado por el método de plantilla se ejecuta enviando el mensaje de plantilla a una instancia de una de las subclases concretas. A través de la herencia, el método de plantilla en la clase base comienza a ejecutarse delegando parte de los detalles de la implementación en las clases hijas. Este mecanismo garantiza que el algoritmo general siga los mismos pasos cada vez, al tiempo que permite que los detalles de algunos pasos dependan de qué instancia recibió la solicitud original para ejecutar el algoritmo.

Este patrón es un ejemplo de inversión de control porque el código de alto nivel ya no determina qué algoritmos ejecutar, en su lugar, se selecciona un algoritmo de nivel inferior en tiempo de ejecución.



By Vanderjoe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=63155402

Para más detalle, ver el artículo *El patrón de diseño Template Method* en Adictos al Trabajo.



Interpreter

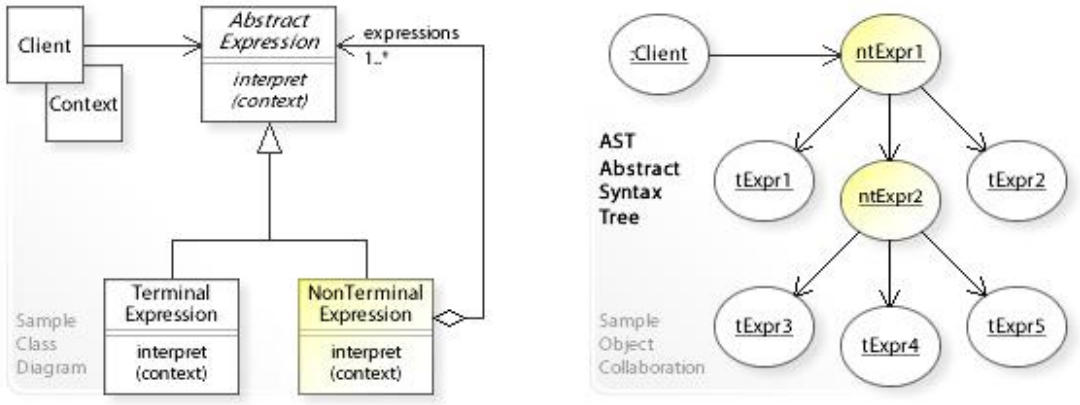
El libro [GoF] lo define de la siguiente manera:

“Dado un lenguaje, define una representación de su **gramática** junto con un **intérprete** que usa dicha representación para interpretar sentencias del lenguaje.”

En otras palabras, el patrón define la gramática de un lenguaje particular de una manera orientada a objetos que puede ser evaluada por el propio intérprete.

Teniendo esto en cuenta, técnicamente podríamos construir nuestra expresión regular personalizada, un intérprete DSL personalizado o podríamos analizar cualquiera de los lenguajes humanos, construir árboles de sintaxis abstracta y luego ejecutar la interpretación.

El patrón intérprete, generalmente debe usarse cuando la gramática es relativamente simple. De lo contrario, podría ser difícil de mantener.



By Vanderjoe - Own work, CC BY-SA 4.0, https://upload.wikimedia.org/wikipedia/commons/3/33/W3sDesign_Interpreter_Design_Pattern_UML.jpg

Ventajas e inconvenientes:

- Facilidad de cambiar y ampliar: puesto que el patrón usa clases para representar las reglas de la gramática, se puede usar la herencia para cambiar o extender ésta.
- Fácil implementación:. las clases que definen los nodos del árbol

sintáctico abstracto tienen implementaciones similares.

- Las gramáticas complejas son difíciles de mantener ya que define, al menos, una clase para cada regla de la gramática. De ahí que las gramáticas que contienen muchas reglas puedan ser difíciles de controlar y mantener.

Para más detalle, ver el artículo Patrón Intérprete en Adictos al Trabajo.

Observer

Este patrón define una **dependencia** entre objetos de forma que cuando un objeto **cambia** su **estado**, todos los objetos que dependen de él son **notificados** y pueden reaccionar si lo desean a una acción.

El objeto de datos, o Subject, provee de métodos para que cualquier objeto Observer pueda suscribirse o cancelar la suscripción, pasando una referencia de sí mismo al Observable o Subject. El Subject mantiene una lista con las referencias a sus Observers para notificarles cada cambio de estado (si procede).

Los Observers a su vez están obligados a implementar los métodos que utiliza el Subject para notificar a sus Observers de los cambios que sufre para que todos ellos tengan la oportunidad de reaccionar a ese cambio de manera que, cuando se produzca un cambio en el Subject, éste pueda recorrer la lista de Observers notificando a cada uno.

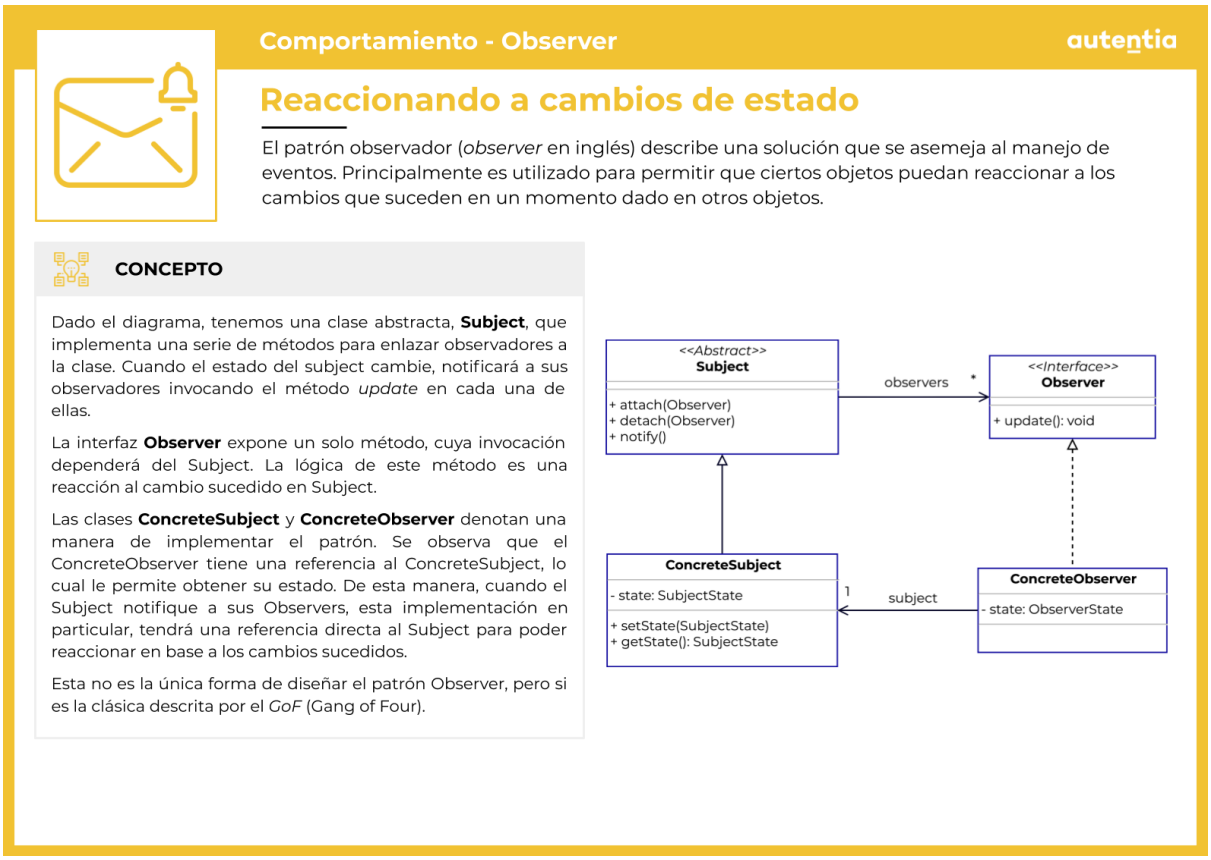
Este patrón es aplicable cuando:

- Una abstracción tiene dos puntos de vista dependientes uno del otro. Encapsular estos puntos de vista en objetos separados permite cambiarlos y reutilizarlos.
- Un cambio en un objeto requiere cambiar otros y no sabemos cuántos objetos van a cambiar.

- Un objeto debería poder notificar a otros sin saber quiénes son.

Entre las ventajas de utilizar este patrón encontramos:

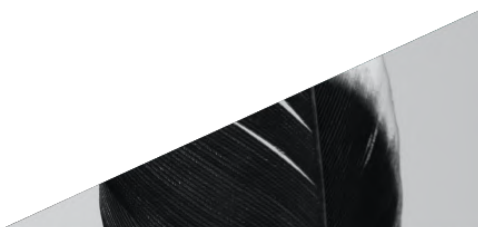
- Podemos modificar el Subject y los Observers de forma independiente al estar desacoplados.
- Nos permite reutilizar tanto Observers como Subjects.
- Respeta el principio Open/Closed, permitiendo añadir Observers sin modificar los Subjects.
- Reduce el acoplamiento entre Subject y Observer, ya que un Subject sólo conoce su lista de Observers a través de un interfaz pero no la clase concreta.
- El Subject se comunica con los Observers mediante broadcast o difusión.

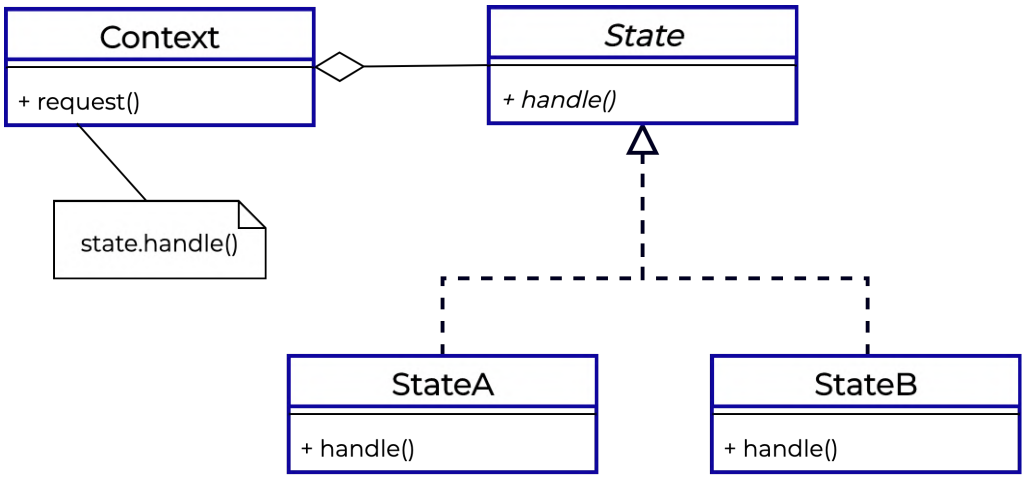


State

Según el libro de [GoF], el patrón State "permite que un objeto **modifique** su **comportamiento** cada vez que **cambie** su **estado** interno. Parecerá que cambia la clase del objeto". Podemos dibujar el comportamiento de nuestro objeto como si se tratase de una "máquina de estados finita".

El patrón estado se usa para encapsular todo el comportamiento variable de un objeto en función de su estado interno. Es una manera más limpia de construir un objeto que cambia su comportamiento en tiempo de ejecución sin recurrir a declaraciones condicionales, respetando el principio Open/Closed y el Single Responsibility Principle, ya que cada estado está representado por una clase que implementa una interfaz. Esto facilita la mantenibilidad del código.





Participantes:

- Context:
    - Define la interfaz que será usada por los clientes.
    - Mantiene una instancia que representa el estado actual del objeto.
- State:
    - Define una interfaz para encapsular el comportamiento asociado con un determinado estado del Context.
- Subclases de State:
    - Cada subclase implementa un comportamiento asociado a los diferentes estados del Context.

Principales ventajas:

- Separa el comportamiento del objeto por estados. Al aislar el comportamiento en estados y convertirlo en clases separadas, nos permite fácilmente añadir nuevos estados y definir las transiciones hacia ellos.

- Hace explícitas las transiciones entre estados. Podemos evitar las transiciones a estados internos inconsistentes, ya que las transiciones son atómicas para el Context.
- Los objetos State pueden compartirse por diferentes contextos. Siempre que estos no tengan estado interno, comportándose como objetos del patrón Flyweight (sin estado intrínseco y con comportamiento).

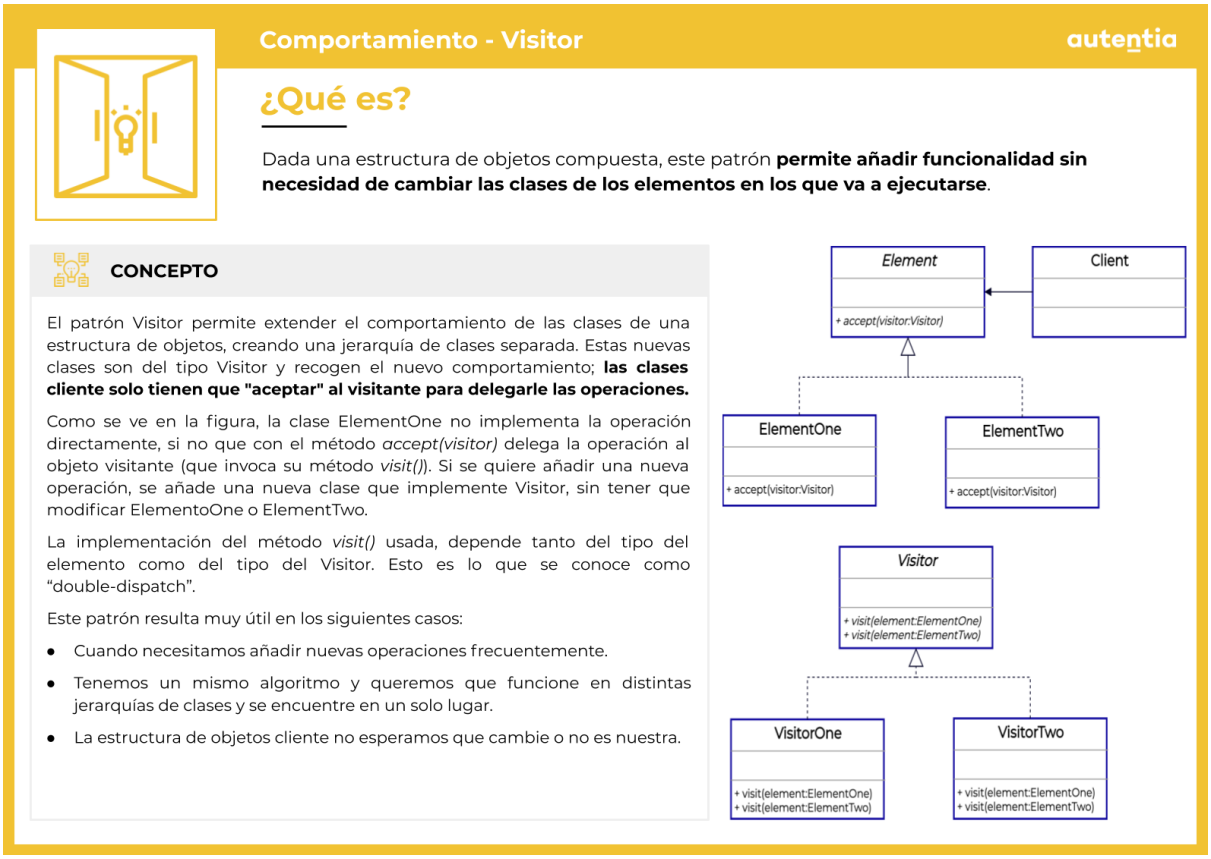
Visitor

Permite **añadir funcionalidad sin** necesidad de **cambiar** las clases de los elementos en los que va a ejecutarse, a través de los denominados Visitors. El patrón sugiere que situemos el comportamiento en una nueva clase llamada Visitor, en vez de integrarlo todo en la clase base. Cada vez que se

necesite añadir un nuevo comportamiento, se hará en una nueva implementación de un Visitor y la clase base solo tiene que aceptar o delegar este comportamiento en el Visitor correspondiente.

Participantes:

- Visitor: interfaz que declara una serie de métodos, normalmente llamados visit y que reciben como parámetro elementos concretos a los que se les va a añadir funcionalidad. Se deben crear tantos métodos visit como clases concretas tengamos por lo que al llamarse igual, se diferenciarán por la firma del método.
- ConcreteVisitor: implementan la interfaz Visitor y definen las funcionalidades que se aplicarán a los elementos o clase base. Aquí es donde se debe definir el comportamiento que debe tener nuestra clase base. En caso de querer hacer alguna modificación, se hará siempre en los ConcreteVisitors y nunca directamente en el elemento. Recordemos que la clase base únicamente delega al Visitor que realice estas modificaciones.
- Element: interfaz que declara un método que acepta los Visitors.
- ConcreteElement: clase base que implementa la interfaz Element y tiene como objetivo redirigir al Visitor concreto que se encargará de añadir el comportamiento específico.



Iterator

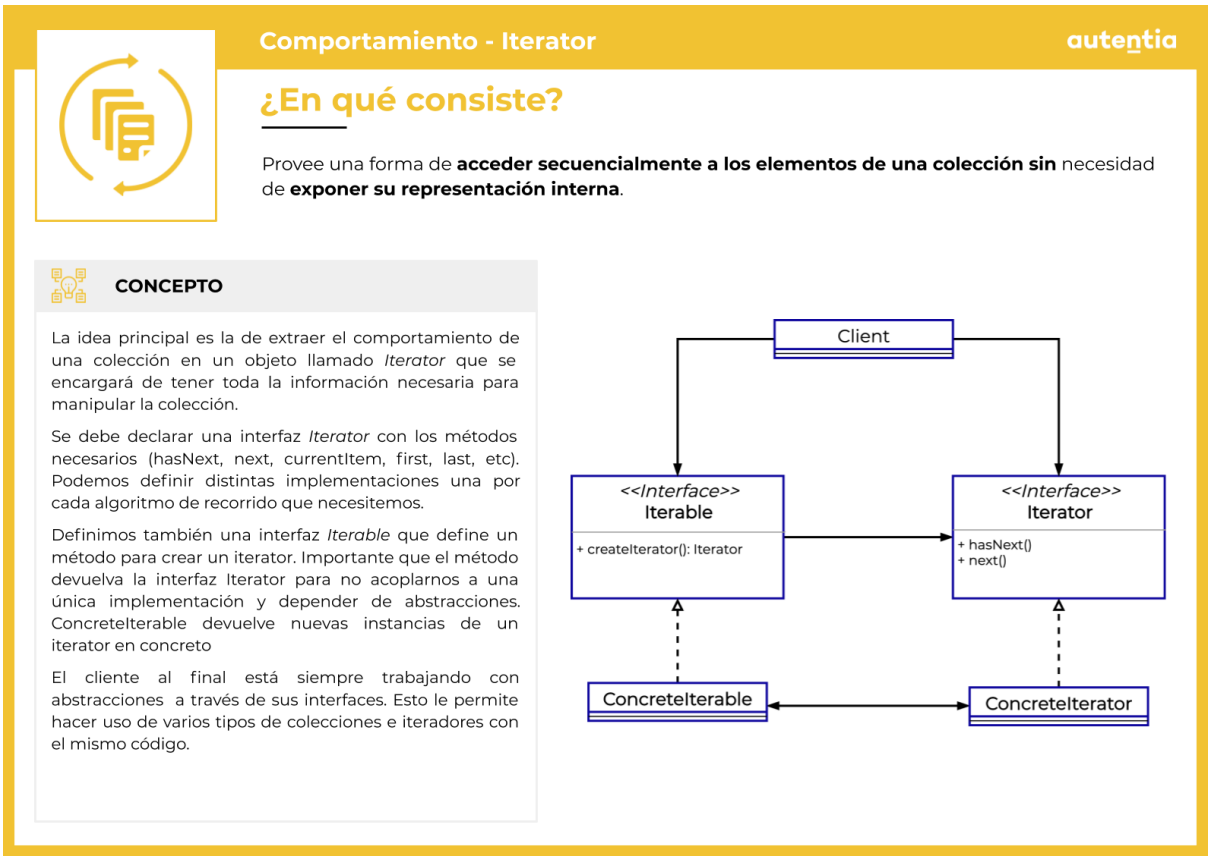
Provee una forma de acceder secuencialmente a los elementos de una colección sin necesidad de exponer su representación interna. El objetivo principal es el de extraer el comportamiento de una colección en un objeto llamado Iterator que se encargará de tener toda la información necesaria para manipularla. El cliente está siempre trabajando con abstracciones a través de sus interfaces. Esto le permite hacer uso de varios tipos de colecciones e iteradores con el mismo código.

Participantes:

- Client: interactúa tanto con Iterator como Iterable a través de sus interfaces para no acoplarse a clases concretas.
- Iterable: declara un método responsable de instanciar el objeto Iterator. Importante que el método devuelva la interfaz para no

acoplarnos a implementaciones.

- ConcreteIterable: implementa la interfaz e instancia el Iterator concreto que iterará sobre una colección específica.
- Iterator: declara los métodos necesarios para recorrer la colección. Puede declarar varios métodos como remove, getFirst, currentItem, size, next, etc.
- ConcreteIterator: implementa los métodos declarados en la interfaz y es responsable de gestionar la posición de la iteración.



Comportamiento - Iterator

autentia

Implementación (1/2)

SOLUCIÓN

Definimos la interfaz Iterator con dos métodos (se podrían añadir más) y su correspondiente implementación. *MessageIterator* debería implementar el algoritmo especifico para recorrer esa colección, además de otra lógica, como tener una propiedad que se encargue de gestionar la posición actual de iterator sobre el array/coleccion/lista, etc.

La interfaz Iterable define un método para instanciar iterator. En esta clase podríamos tener una lista con su respectivo método que vaya añadiendo nuevos mensajes y al instanciar *MessageIterator*, se podría pasar dicha lista por parámetro para que pueda ser recorrida posteriormente.

```
public class Message{
  //...
}

public interface Iterable {
   Iterator createIterator();
}

public class MessageIterable
implements Iterable {
   //...

   @Override
   public Iterator createIterator(){
       return new
MessageIterator(messageList);
   }
}
```

```
public interface Iterator {
   Boolean hasNext();
   Object next();
 }

public class MessageIterator implements Iterator{
    private Message[] messageList
    private int currentPosition;
    //constructor

   @Override
   public Boolean hasNext() {
       if(position >= messageList.length){
           return false
        }else {true}
   }

   @Override
   public Object next() {
       Message nextItem = messageList[currentPosition]
       currentPosition += 1;
       return nextItem;
   }
}
```

Comportamiento - Iterator

autentia

Implementación (2/2)

SOLUCIÓN

NotificationScreen será la clase encargada de imprimir los mensajes correspondientes usando los métodos de la clase iterator.

La clase main simplemente define que tipo de iterable desea usar. En este ejemplo, solo tenemos una implementación (*MessageIterable*) pero en un ejemplo real podríamos tener varios tipos.

```
public class NotificationScreen {

   private Iterable messages;

   public NotificationScreen(Iterable messages) {
       this.messages = messages;
   }

   public void showMessages() {
       Iterator iterator = messages.createIterator();
       while (iterator.hasNext()) {
          System.out.println("next item: " + iterator.next());
           //...
       }
   }
}
```

```
public class Main {
   public static void main(String[] args) {
       Iterable messages = new MessageIterable();
       NotificationScreen notificationScreen = new NotificationScreen(mesagges);
       notificationScreen.showMessages();
   }
}
```

Patrones JEE

Introducción

Java Enterprise Edition, conocida por sus siglas JEE, es un conjunto de tecnologías y especificaciones orientadas al desarrollo de aplicaciones empresariales con Java. El término *empresarial* hace referencia a que esta plataforma fue diseñada para resolver problemas de desarrollo a las empresas. Proporciona una API y un entorno de ejecución para ejecutar aplicaciones en red que se caracterizan por su escalabilidad, fiabilidad y seguridad.

Según las empresas han ido incorporando en sus proyectos el stack Java EE han ido apareciendo problemas de diseño y arquitectura comunes a esta plataforma. Y con ello, han aparecido sus correspondientes **soluciones**.

Los patrones JEE nacen para resolver los problemas que surgen en el desarrollo de aplicaciones a través de Java EE. En este documento abordaremos los patrones más usados en la actualidad, teniendo en cuenta que la tecnología avanza mucho y la plataforma Java EE destaca por ser una de las más usadas en el mercado.

Intercepting filter

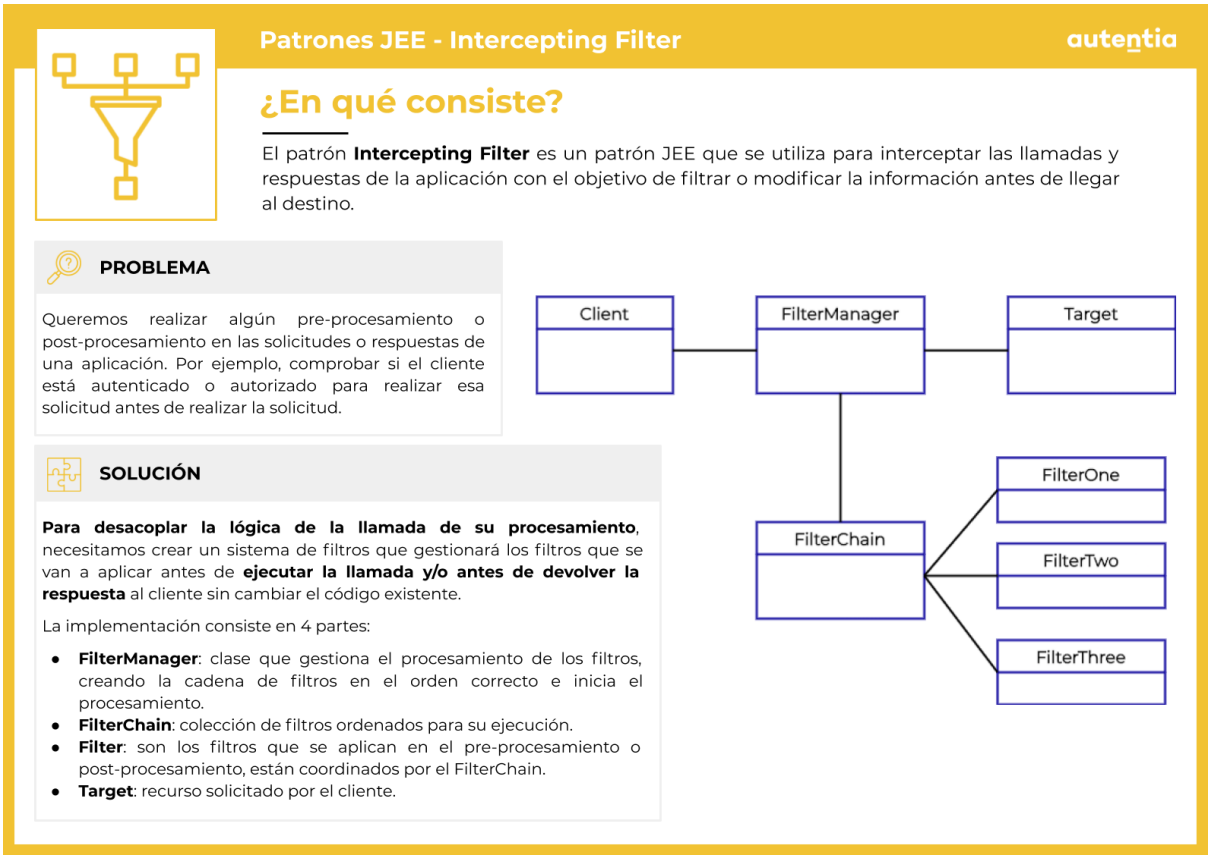
El objetivo principal de este patrón es interceptar y manipular solicitudes y respuestas, antes y después de que estas sean procesadas.

Es común que las aplicaciones tengan requisitos como autenticar el usuario en cada solicitud que se haga a la aplicación. Ese tipo de funcionalidades

se pueden centralizar en una clase por la que pasan todas las solicitudes y respuestas, esa clase suele ser el **FilterManager**. Al tener el código más separado se reducirá el acoplamiento.

El **FilterManager** recibirá la solicitud/respuesta y creará un objeto **FilterChain** que contiene los **filtros** necesarios para manipular esa solicitud/respuesta antes de llegar al destino.

Aplicando el patrón, la funcionalidad de autenticación de antes estaría implementada en un filtro que será el único que tendremos que modificar y afectará a todas las solicitudes.



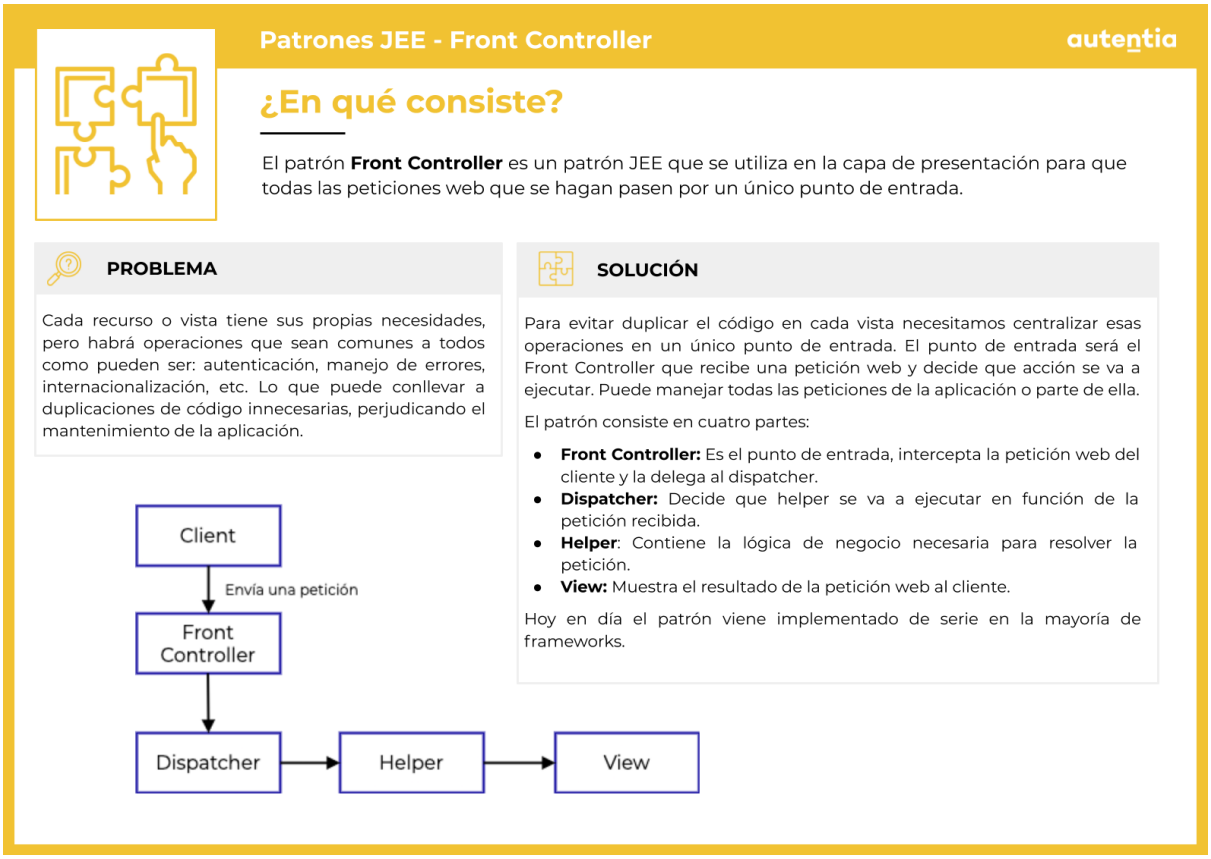
Front controller

Este patrón JEE consiste en utilizar una clase que actúa como intermediaria entre el cliente y el recurso solicitado. La idea es que las

operaciones más comunes, como autenticación, gestión de errores, etc. Se centralizan implementado un controlador único para evitar duplicar el código en cada recurso y facilitar el mantenimiento de la aplicación.

El patrón gestiona todas las peticiones web. Para implementarlo es necesario crear un:

1. **FrontController**: el cliente al solicitar un recurso pasará primero por el FrontController que es el punto de entrada que se encarga de interceptar las peticiones web del cliente y delegarlas al dispatcher.
2. **Dispatcher**: gestiona y coordina las acciones necesarias para resolver las peticiones. Recibe ayuda de un Helper para obtener el resultado y también de la Vista (en inglés View) para mostrar el resultado al cliente.
3. **Helper**: contiene la lógica de negocio para resolver la petición.
4. **View**: muestra el resultado de la petición al cliente.



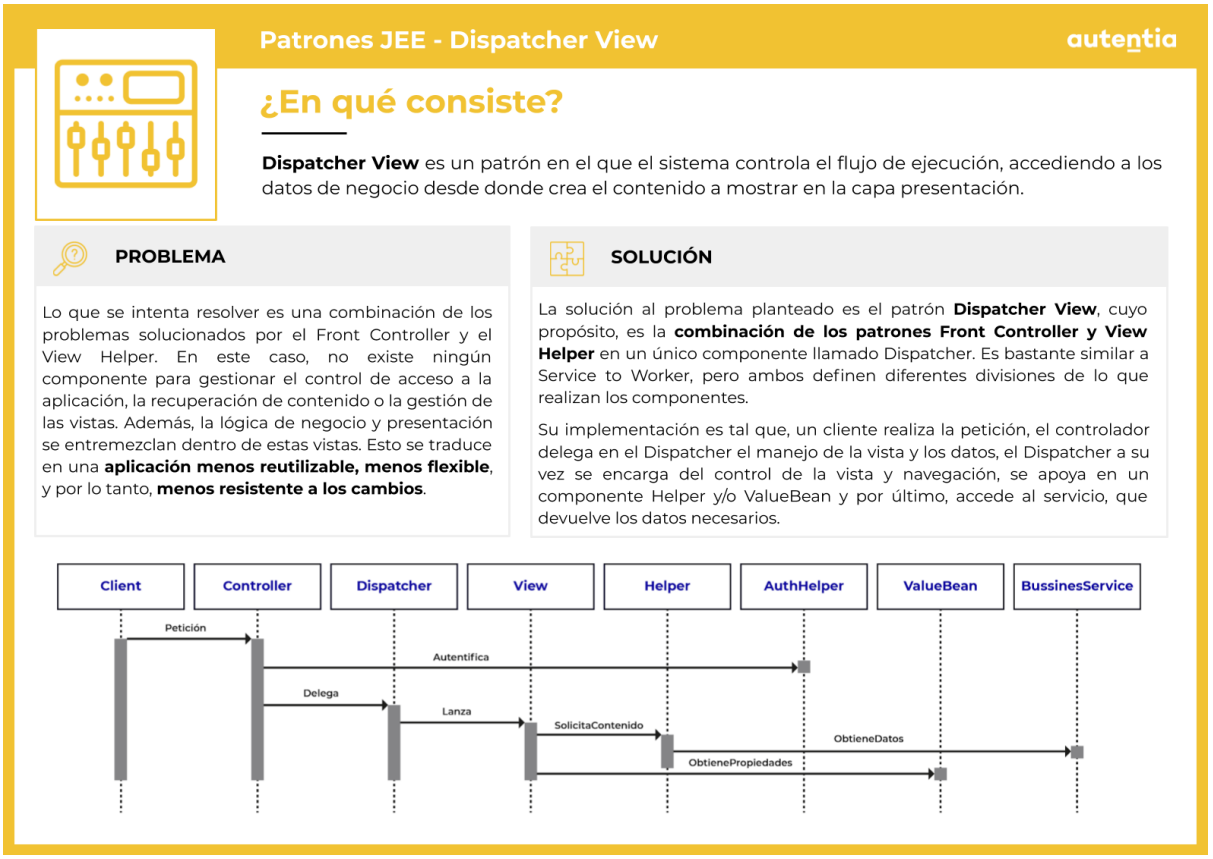
Dispatcher view

Este patrón JEE nos permite que cuando un controlador derive la respuesta a un componente de presentación, separar su parte lógica de la física. Añadir un nivel más entre componentes e intercambiar los elementos físicos de presentación de forma transparente a la lógica de la aplicación.

Esto también facilita la integración entre diferentes aplicaciones y la gestión de errores centralizada.

En este patrón, el controlador delega sobre el *dispatcher* la presentación de algún elemento lógico, luego el *dispatcher* se encargará de resolver la indirección y de esta forma, proporcionar el elemento físico. El *dispatcher*, a su vez se encargará del control de la vista y la navegación.

De esta forma, lo que se intenta resolver es una combinación de los problemas que solucionan con el patrón **Front Controller** y **View Helper**, por tanto, cuando queremos que una vista controle una solicitud y genere una respuesta, debemos usar este patrón.



Business delegate

Este patrón nos permite crear clases que hagan de *proxy* sobre las funciones reales remotas, esto genera más complejidad pero mejora la flexibilidad.

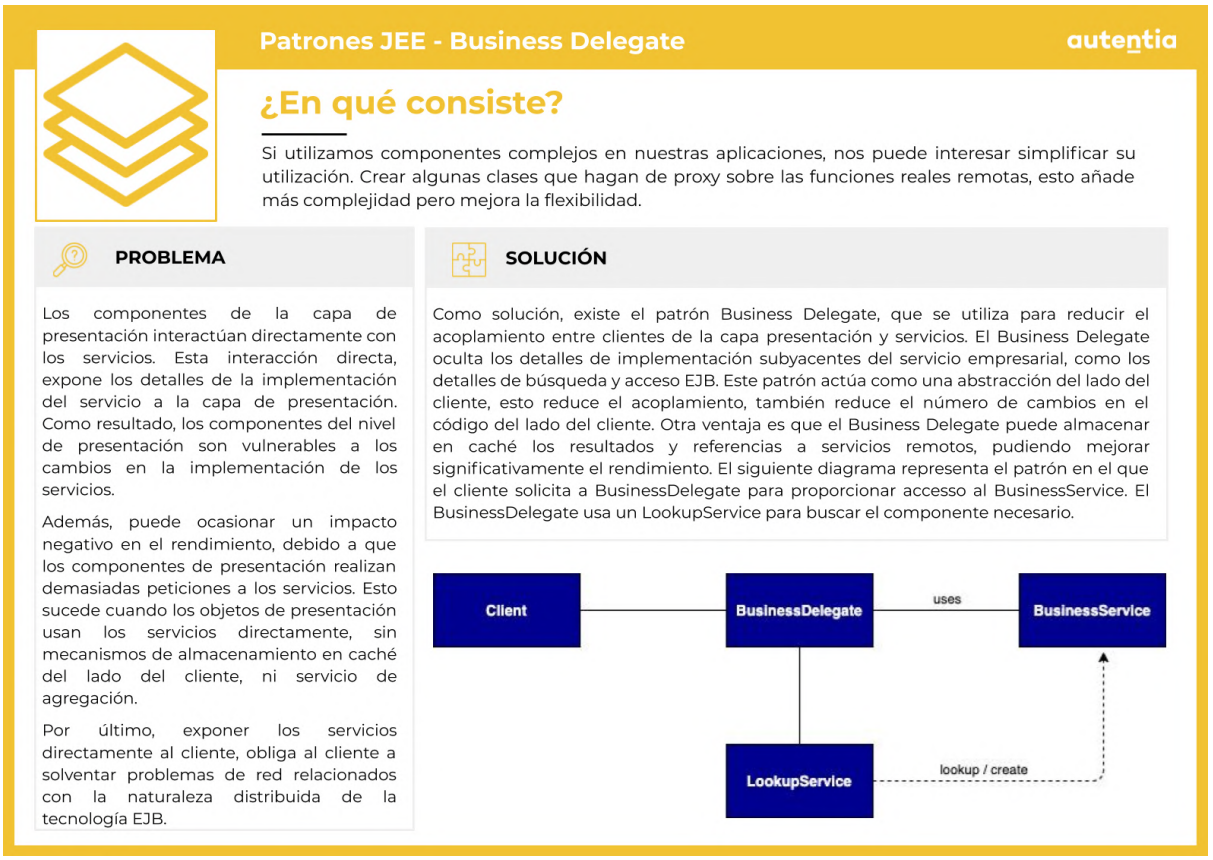
El problema que queremos resolver con este patrón es que los componentes de la capa de presentación interactúan directamente con los

servicios. Esta interacción directa, expone los detalles de la implementación del servicio a la capa de presentación. Como resultado, los componentes del nivel de presentación son vulnerables a los cambios en la implementación de los servicios.

Este patrón nos resulta muy útil cuando queremos ocultar a los clientes la complejidad de la comunicación remota con los componentes o servicios.

Por ejemplo, queremos tener acceso a los componentes de nivel empresarial desde los componentes y clientes a nivel de presentación, o minimizar el acoplamiento entre clientes y servicios, ocultando de esa forma los detalles de la implementación del servicio, como la búsqueda y el acceso, si deseamos ocultar los detalles de creación de servicios o traducir excepciones de red, en excepciones de aplicación o usuario.

De esta forma, usando un **Business Delegate**, que actúa como una abstracción del lado del cliente, podemos encapsular el acceso a un servicio. Este ocultará los detalles de implementación del servicio, así como los mecanismos de búsqueda y acceso al servicio, reduciendo el acoplamiento y el número de cambios en el código del lado del cliente.



Value object

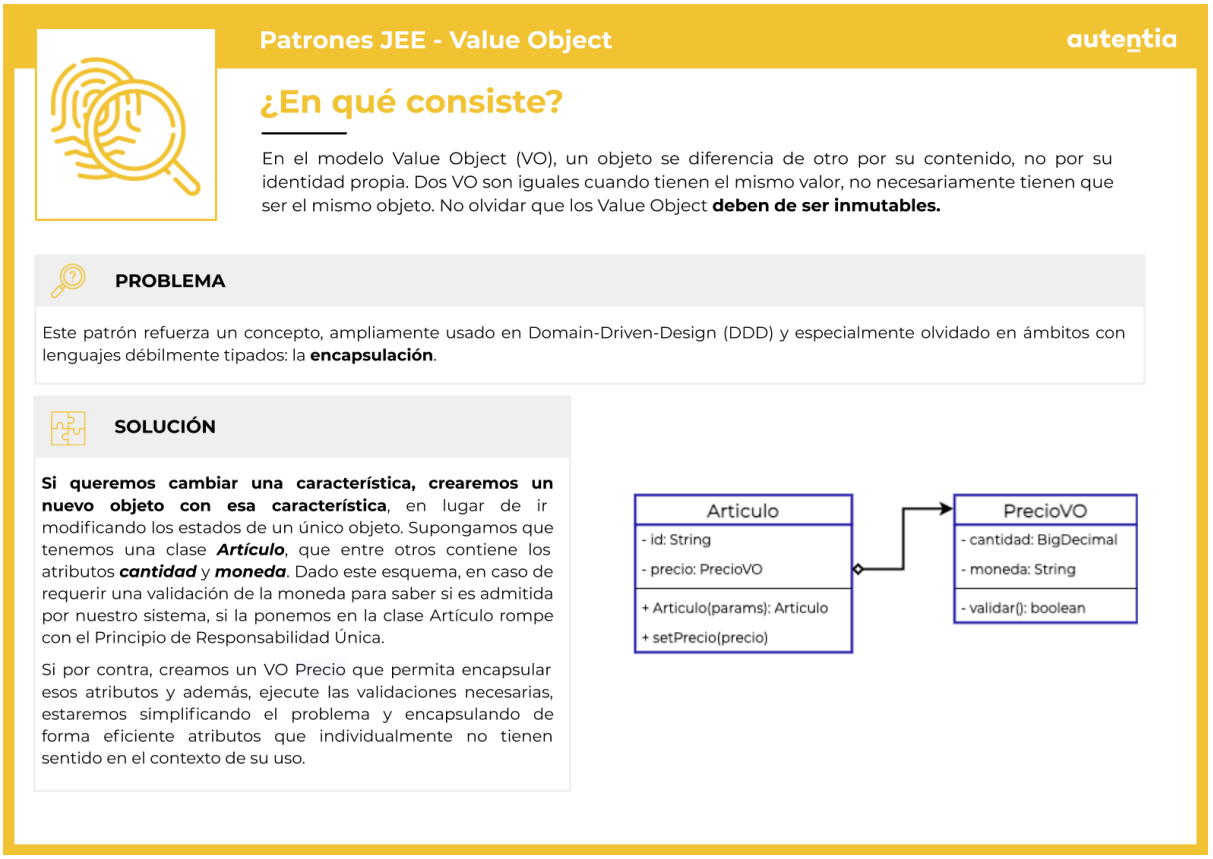
El objeto *Value Object* es un tipo inmutable que se distingue de otro por su contenido. Al contrario de una *Entidad* que tiene identidad propia, los dos objetos *Value object* son iguales si sus propiedades tienen el mismo valor.

Este patrón es ampliamente usado en Domain-Driven-Design (DDD). El patrón *Value Object* refuerza un concepto olvidado de los principios de orientación a objetos, especialmente por aquellos que están acostumbrados a los lenguajes débilmente tipados: la encapsulación.

Los *Value Object* son normalmente objetos inmutables. Dado su limitado tamaño, su construcción no tiene un impacto significativo en el consumo de memoria, por lo que es preferible la creación de una nueva instancia antes que la modificación de una ya existente, evitando así las

consecuencias derivadas.

Hay que distinguir una *Entidad* y un *Value object*. La principal diferencia es que las primeras poseen una identidad, un identificador que las hace únicas de cara a otra instancia de la misma clase. Un *Value object* en cambio no posee identidad, por lo que las comparaciones entre value objects deben hacerse basándose en su contenido, y no un identificador o referencia.



Domain Driven Design (DDD)

autentia

¿Qué es?

El diseño guiado por el dominio es un enfoque para el desarrollo de software con necesidades complejas mediante una profunda conexión entre la implementación y los conceptos del modelo y núcleo del negocio. El término fue acuñado por Eric Evans en su libro *"Domain-Driven Design - Tackling Complexity in the Heart of Software"*.

¿QUÉ PROBLEMA INTENTA RESOLVER DDD?

- Depende del caso o **proyecto** nos podemos encontrar con que la **complejidad** de muchas aplicaciones no está en la parte técnica sino **en la lógica del negocio o dominio.**
- El dilema empieza cuando intentamos **resolver problemas del dominio con tecnología**. Eso provoca que, aunque la aplicación funcione, no haya *nadie capaz de entender realmente cómo lo hace*. Es habitual que surja el anti-patrón *"Modelo del Dominio Anémico"* (en inglés Anemic Domain Model).

¿CÓMO INTENTA RESOLVERLO?

- El **DDD no es una tecnología ni una metodología**, éste *provee una estructura de prácticas y terminologías* para tomar decisiones de diseño que enfoquen y aceleren el manejo de dominios complejos en los proyectos de software.
- **Proporciona una serie de patrones tácticos y estratégicos** que nos ayudan a trabajar con los expertos del dominio modelando el problema y la solución. De este modo, se inicia una colaboración creativa entre técnicos y expertos de dominio para interactuar lo más cercano posible a los conceptos fundamentales del problema.

¿QUÉ CONCEPTOS CLAVE UTILIZA?

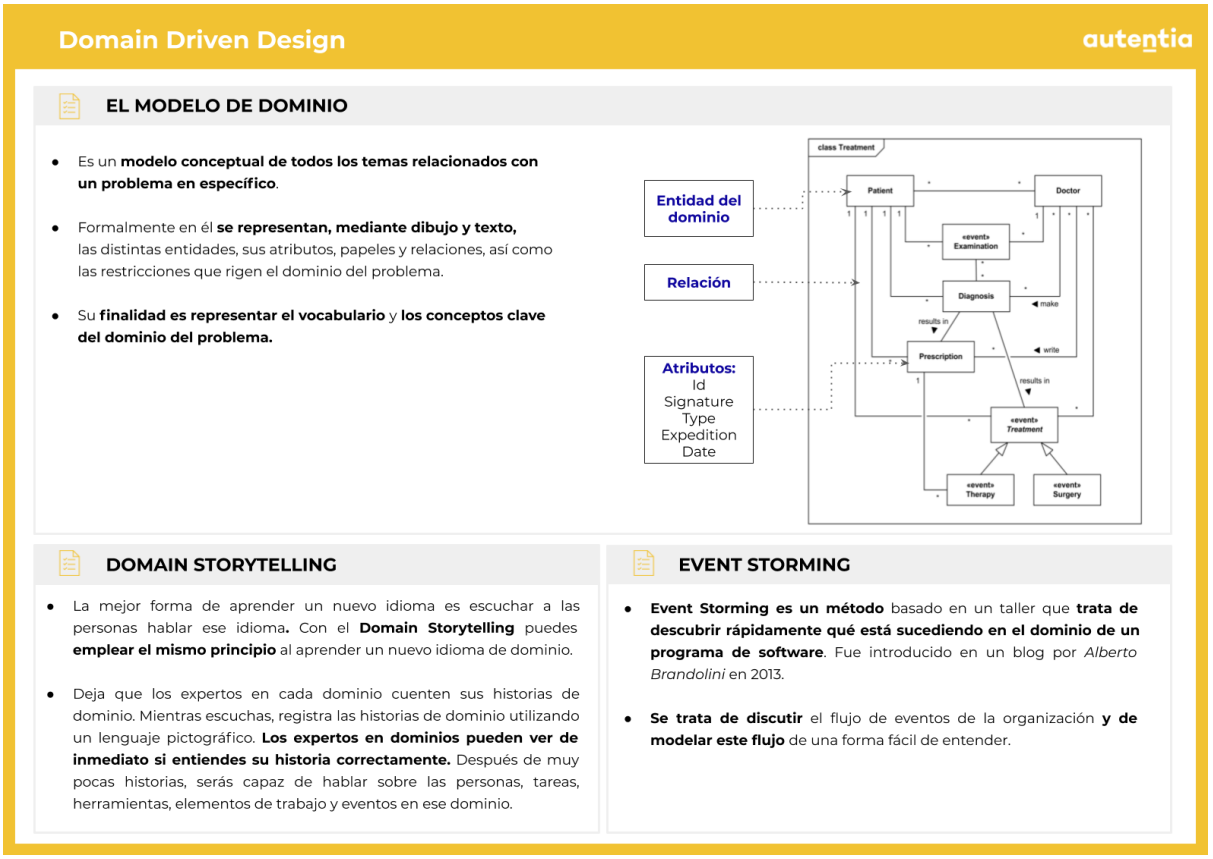
- El **lenguaje ubicuo** (lenguaje común entre los programadores y los usuarios) y el **bounded context** (identificación de los límites de los diferentes dominios/subdominios) principalmente.

CARACTERÍSTICAS DEBE TENER UN PROYECTO PARA QUE SEA INTERESANTE APLICAR DDD

- Tenemos un **dominio complejo**.
- No tenemos ni idea del dominio, pero **sabemos que vamos a tener muchos procesos, HdUs, etc.**
- Se trata de un proyecto con **proyección a varios años vista.**

PRE-REQUISITOS NECESARIOS PARA APLICAR DDD

- El **desarrollo debe ser iterativo.** Esto será necesario para ir refinando el modelo del dominio continuamente a medida que aprendemos más sobre este y avanzamos.
- **Debe existir una estrecha relación entre los desarrolladores y los expertos del dominio.** El conocimiento profundo del dominio es esencial, al igual que la colaboración con los expertos de desarrollo durante la vida del proyecto; esto evitará malos entendidos entre las partes del equipo y ofrecerá la oportunidad de obtener un conocimiento más profundo del dominio.



Session facade

Este patrón de JEE se usa para ocultar la complejidad de diferentes interfaces, solo exponiendo una interfaz como la única entrada.

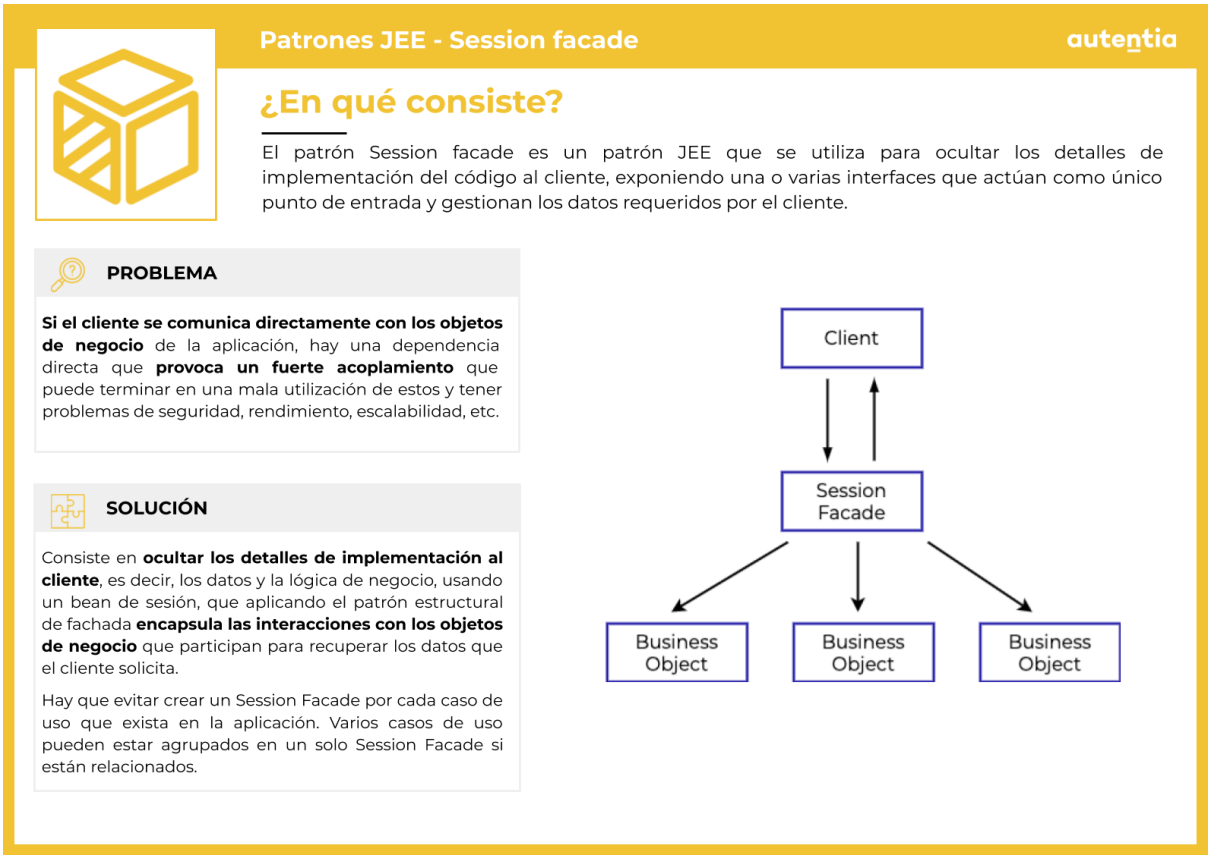
Los clientes de aplicaciones necesitan acceso a los objetos de negocio para cumplir con sus responsabilidades y cumplir con los requisitos de usuarios. Los clientes pueden interactuar directamente con estos objetos. Pero cuando se exponen los objetos de negocio directamente al cliente, el cliente debe comprender y ser responsable de todas las relaciones con esos objetos y debe poder manejar todo el flujo. Sin embargo, la interacción directa entre el cliente y los objetos de negocio crea un acoplamiento estrecho entre los dos, y ese acoplamiento estrecho hace que el cliente

dependa directamente de la implementación de los objetos de negocio. La dependencia directa significa que el cliente debe representar e implementar las interacciones complejas de lógica de negocio. A medida que aumentan los requisitos del cliente, aumenta la complejidad de la interacción entre el cliente y los objetos de negocio.

La solución es usar un bean de session como fachada para encapsular la complejidad de las interacciones entre los objetos de negocio que participan en un flujo y el cliente. El bean de sesión (que representa la fachada de sesión) gestiona las relaciones entre los objetos de negocio. El bean de sesión también gestiona el ciclo de vida de estos participantes al crearlos, localizarlos (buscarlos), modificarlos y eliminarlos según lo requiera el flujo de la aplicación. En una aplicación compleja, Session Facade puede delegar esta gestión a un objeto separado.

Conviene evitar crear la fachada (Session facade) para cada caso de uso. Se pueden agrupar los casos de uso relacionados entre ellos. Por lo tanto, es aconsejable diseñar las fachadas de sesión para agregar un grupo de interacciones relacionadas en una sola fachada de sesión. Esto da como resultado menos complejidad para la aplicación y de este modo podemos aprovechar los beneficios del patrón Session Facade.

En resumen, *Session facade* gestiona los objetos de negocio y proporciona una capa de acceso a los clientes. También abstrae las interacciones entre los objetos de negocio y proporciona una capa de servicio que expone sólo las interfaces necesarias. Y de esta manera, oculta a la vista del cliente las complejas interacciones entre los participantes.



Composite entity

Este patrón JEE nos permite modelar y gestionar un conjunto de objetos relacionados como si fuesen uno solo, un todo. Al reducir la cantidad de objetos mejoramos el mantenimiento. Para entender este patrón primero tenemos que entender la granularidad de objetos:

1. **Coarse-grained object (Objeto de grano grueso)**: es un objeto que contiene datos relacionados. Por ejemplo, un objeto “User” guarda el nombre, apellido, nombre de usuario, contraseña, rol, fecha de contratación, etc.
2. **Fine-grained object (Objeto de grano fino)**: es un objeto que contiene menos datos, son datos concretos de ese dominio. Por ejemplo, un objeto “User” guarda el nombre del usuario y la contraseña, un objeto “Rol” guarda el rol del usuario y cuando ha accedido la última vez,

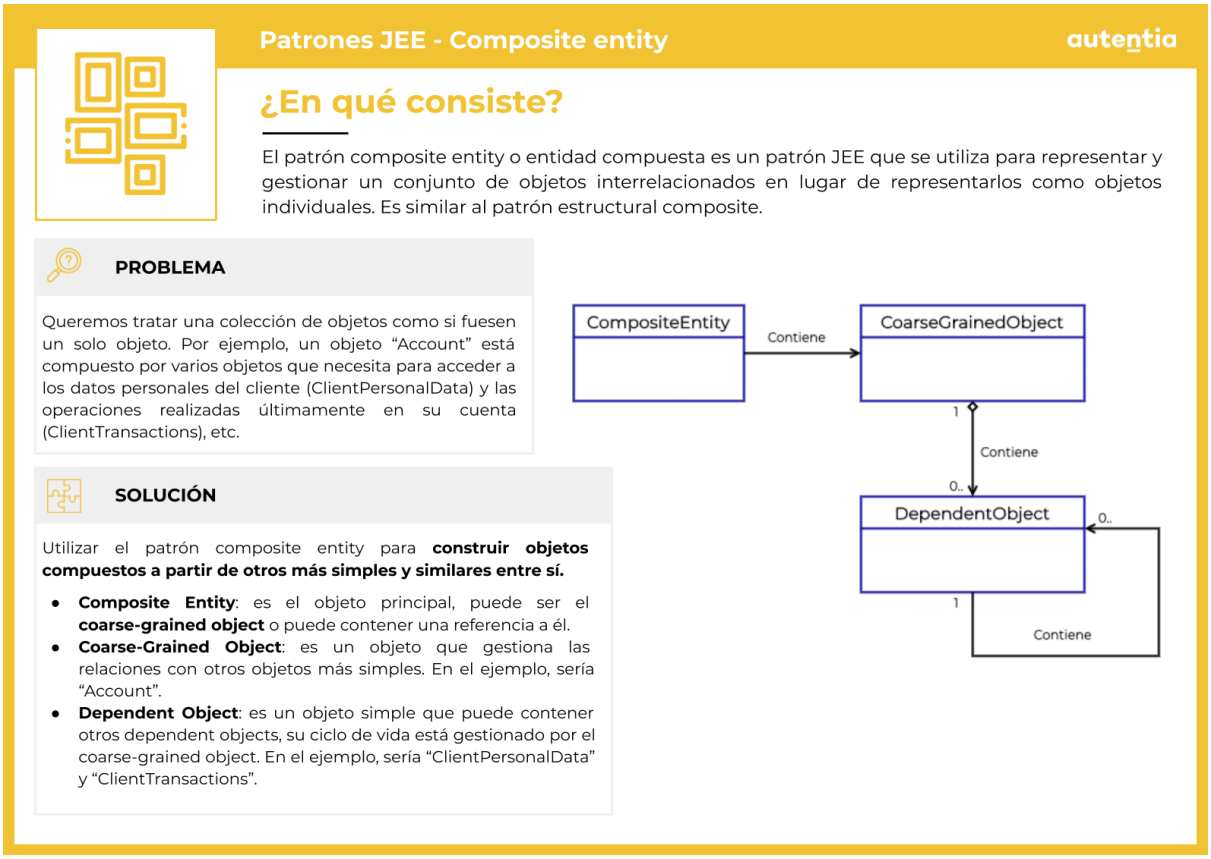
etc. Hay relaciones definidas entre estos objetos.

Composite entity (Entidad compuesta): es la entidad principal que puede ser de grano grueso o puede contener referencias a objetos de grano grueso.

Coarse-grained object (Objeto de grano grueso): contiene los objetos dependientes, tiene su propio ciclo de vida y también gestiona el ciclo de vida de los objetos dependientes.

Dependent object (Objetos dependientes): depende de los objetos de grano grueso y suelen ser objetos de grano fino. Es posible que contenga otros objetos dependientes, lo que puede dar lugar a una estructura en árbol.

Hay varias estrategias para implementar el patrón, la estrategia más popular consiste en que la entidad compuesta tiene la responsabilidad de guardar varios objetos de grano grueso, mientras que estos gestionan sus relaciones con los objetos dependientes. De esta manera, cuando una entidad compuesta se actualiza, todos los objetos dependientes son automáticamente actualizados.

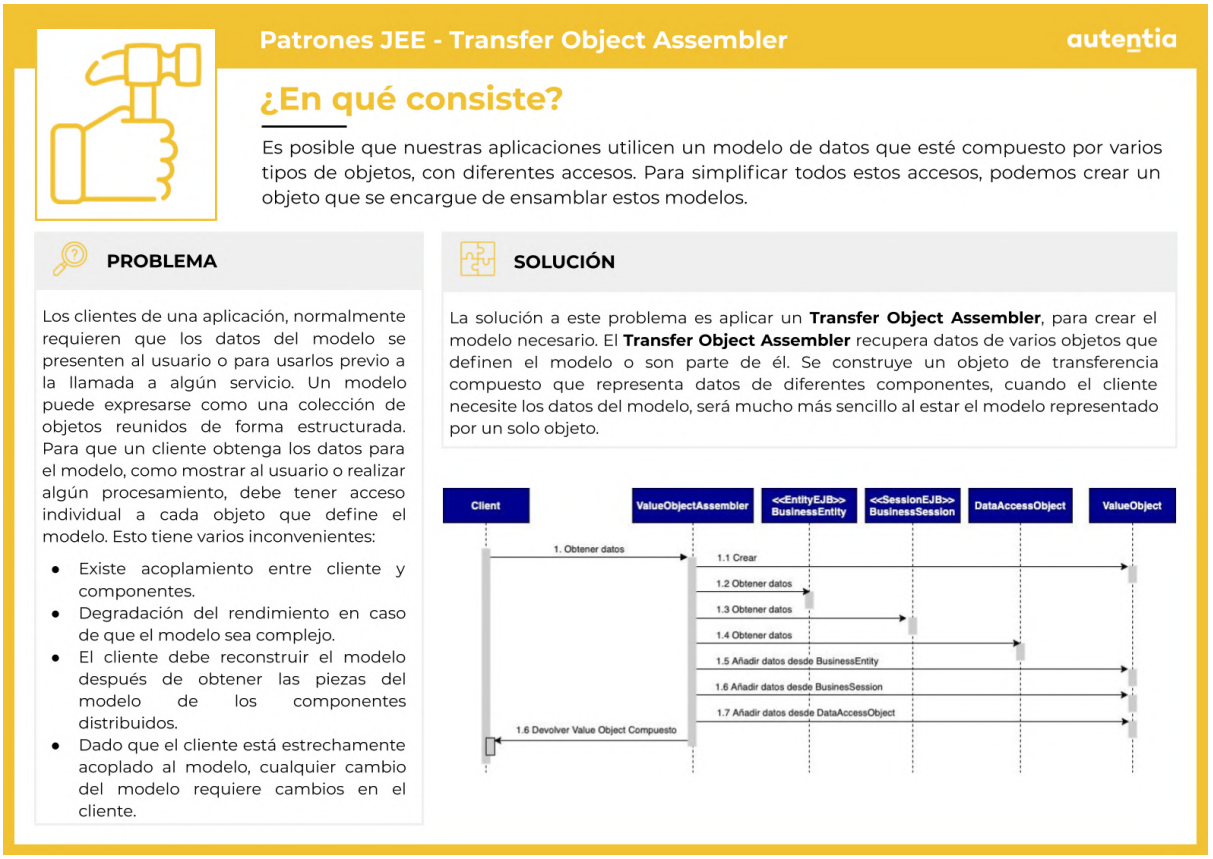


Transfer object assembler

Este patrón JEE nos permite construir un modelo desde diferentes servicios de la aplicación, para poder usarlo desde el lado del cliente.

Transfer Object Assembler nos permite crear un modelo de aplicación como un objeto de transferencia compuesto. Este ensamblador agrega varios objetos desde varios componentes y servicios y una vez terminado, lo devuelve al cliente.

Es útil cuando deseamos encapsular la lógica de negocio de forma centralizada y evitar su implementación en el cliente, crear un modelo complejo para el cliente con fines de presentación, o que los clientes sean independientes de la complejidad de la implementación del modelo y reducir el acoplamiento entre cliente y componentes empresariales.



Value list handler

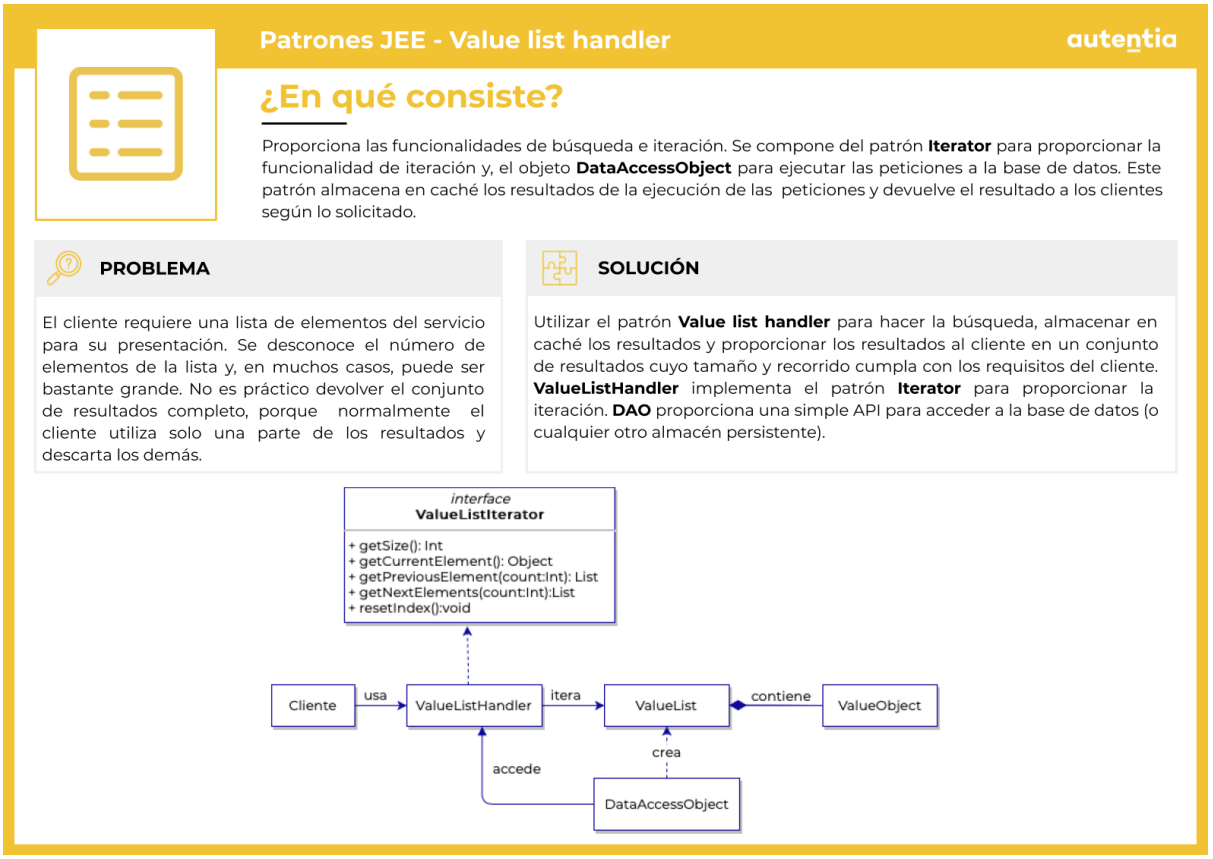
Este patrón JEE es útil cuando queremos recuperar una lista de objetos desde algún servicio, por ejemplo, cuando tenemos una aplicación cliente que quiere recorrer una lista de resultados.

De esta forma, podemos crear un componente, que implemente una interfaz y de esta manera, nos permita obtener los diferentes valores e iterar por ellos. Es recomendable revisar las interfaces estándar de Java para no repetir cualquier comportamiento que ya exista.

Este patrón es útil también cuando queremos evitar la sobrecarga del uso de métodos de búsqueda, se desea implementar casos de uso de solo lectura que no requiera alguna transacción o mantener los resultados de la

búsqueda en el lado del servidor.

En pocas palabras, necesitamos un **List Value Handler** para buscar, almacenar en caché los resultados y permitir que la aplicación cliente itere y seleccione los elementos desde los resultados.



Service locator

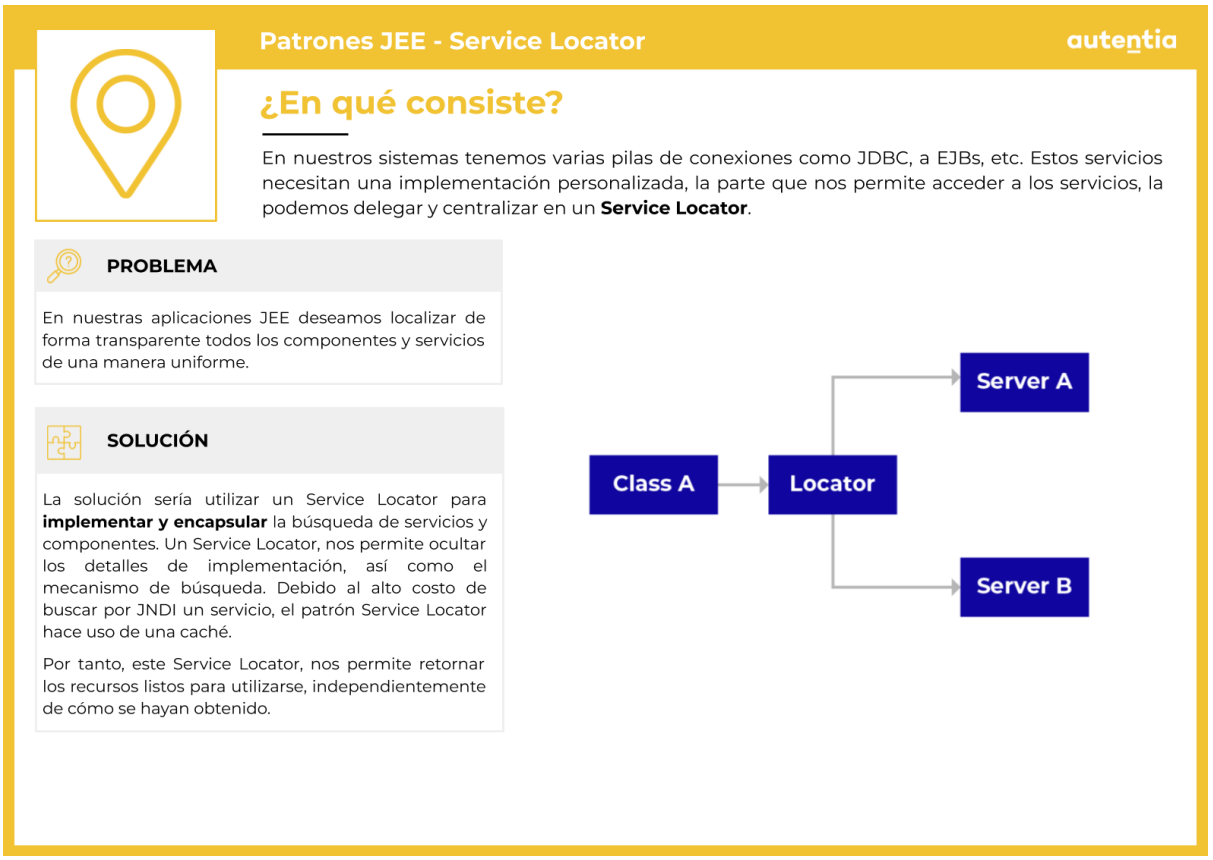
Este patrón nos permite localizar de una forma transparente los servicios y componentes JEE de una manera uniforme.

Cuando en nuestros sistemas queremos acceder a JDBC, servicios asíncronos, etc. Necesitaremos código específico, para cada implementación en particular. Esto último, lo podemos centralizar en un solo componente que nos permita localizar los servicios.

Este localizador de servicios se encargará de devolver los recursos para que sean utilizados, independientemente de cómo los haya obtenido.

Este patrón es útil cuando por ejemplo, deseamos utilizar la API JNDI para buscar y usar componentes, queremos centralizar y/o reutilizar la implementación de mecanismos de búsqueda, necesitamos evitar que el rendimiento se vea afectado por la creación del contexto inicial, etc.

Por tanto, resumiendo, se trata de un localizador de servicios que implementa y encapsula la búsqueda de servicios y componentes. Este se encargaría de ocultar los detalles de implementación del mecanismo de búsqueda y encapsular las dependencias.



Data access object

Este patrón (DAO) se utiliza para separar la lógica de negocio de la lógica para acceder a los datos. El DAO proporciona los métodos necesarios para insertar, actualizar, borrar y consultar la información. Por otra parte, la capa de negocio solo se preocupa por lógica de negocio e interactúa con el DAO para acceder a los datos.

¿Por qué tenemos que separar la capa de negocio de la fuente de datos? Porque podríamos tener diferentes formatos de datos en la nube, en la base de datos local, en otro servidor o en otro sistema. En cada caso para acceder a los datos, tenemos que actuar de otra manera distinta. Si toda la casuística de acceso a los datos está presente en la capa de negocio, estamos acoplando nuestro modelo de negocio a una fuente de datos determinada. No podréis cambiarla en el futuro sin cambiar nuestra lógica de negocio.

Necesitamos crear un DAO para abstraer toda la lógica de acceso a los datos. Así podemos cambiar la fuente de datos fácilmente sin cambiar nada en la capa de negocio.

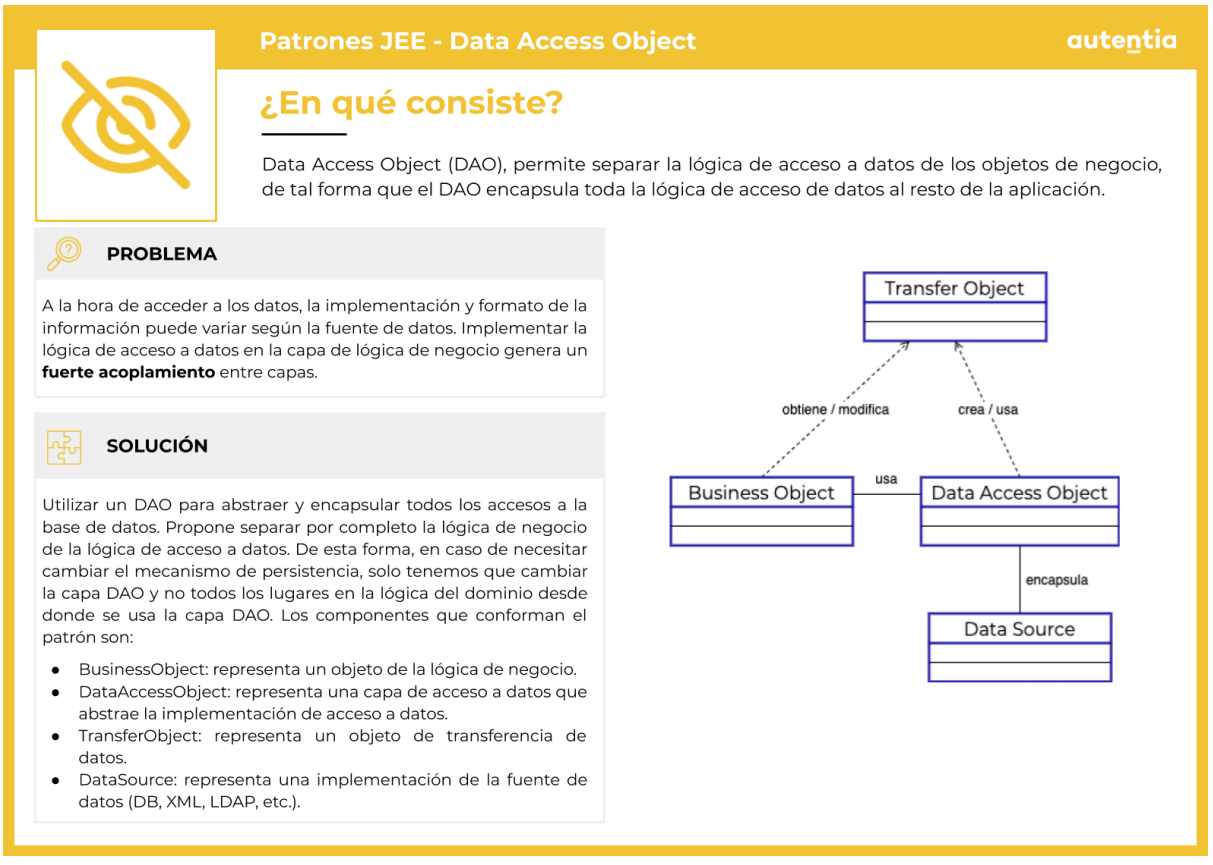
El patrón DAO contiene los siguientes componentes:

- *Business Object*: representa un objeto con la lógica de negocio.
- *Data Access Object*: representa una capa de acceso a datos que oculta la fuente y los detalles técnicos para recuperar los datos.
- *Transfer Object*: este es un objeto plano que implementa el patrón DTO, el cual sirve para transmitir la información entre el DAO y el *Business Object*.
- *DataSource*: representa la fuente de datos, la cual puede ser una

base de datos, los servicios Web, archivos de texto, etc.

El patrón DTO se utiliza para crear un objeto plano (POJO) con una serie de atributos que puedan ser recuperados del servidor con una sola llamada de tal forma que un DTO puede contener información de múltiples fuentes o tablas y concentrarlas en una única clase simple.

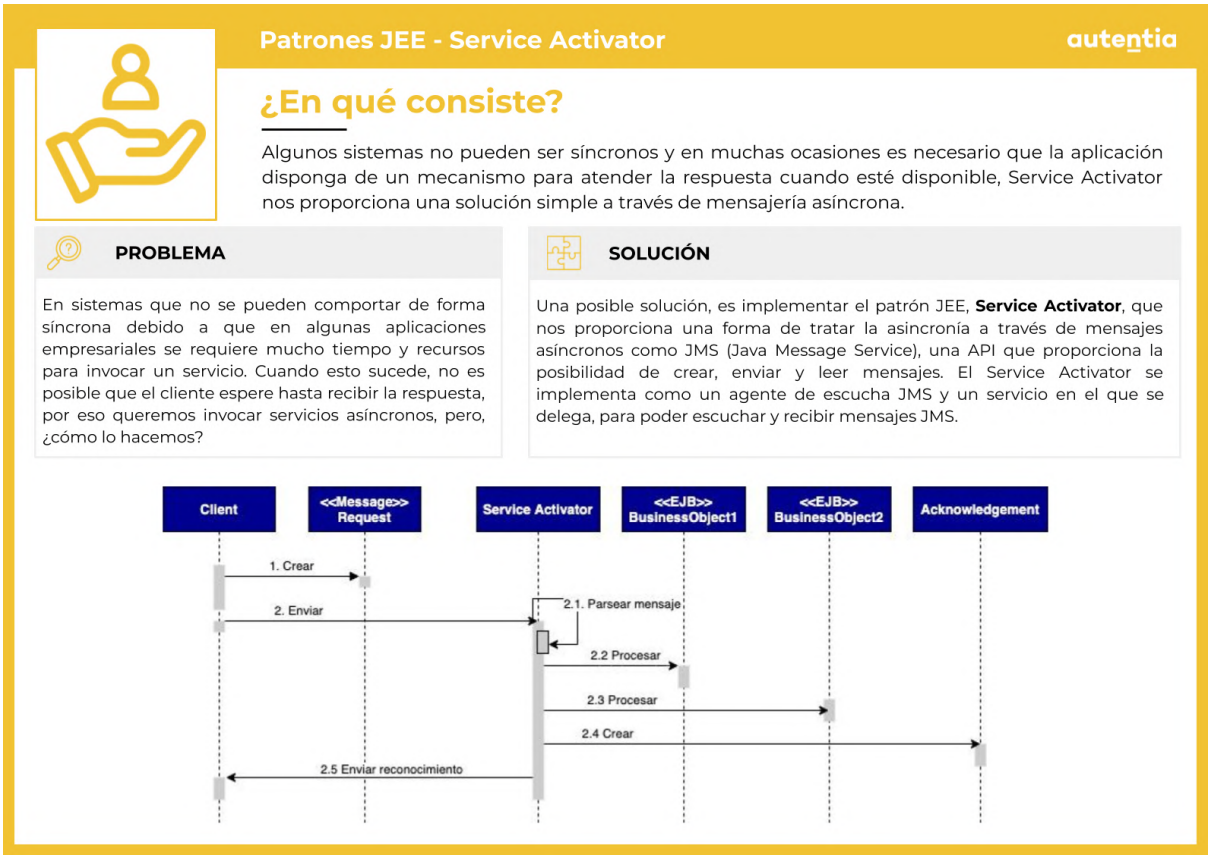
El objeto de negocio *Business Object* solo recupera los Transfer Object de nuestro DAO y no sabe nada del origen de los datos. De este modo, podemos ocultar los detalles de representación real de los datos en la fuente y blindar la separación de responsabilidades en nuestro sistema.



Service activator

Este patrón JEE es el ideal cuando queremos solucionar el inconveniente de realizar peticiones a servicios de forma asíncrona.

En muchas ocasiones una aplicación no se puede comportar de un modo síncrono, de forma que realice una petición y reciba una respuesta inmediata sino que muchas veces necesitamos aplicaciones que envíen una petición y dispongan de un medio para atender adecuadamente la respuesta cuando esté disponible. Por ejemplo, cuando queremos invocar servicios de forma asíncrona o cuando necesitamos realizar una tarea empresarial que se compone lógicamente de varias tareas empresariales. De esta forma, el patrón **Service Activator**, nos permite llamar a uno o varios servicios de forma asíncrona y nos proporciona un modelo simple que podemos implementar a través de servicios de mensajería asíncrona como JMS.



Antipatrones

Si bien los patrones de diseño ofrecen soluciones comunes a problemas recurrentes de diseño de aplicaciones, los **antipatrones** ofrecen ejemplos de **malas** soluciones que se aplican comúnmente a distintos problemas.

Para que un antipatrón sea reconocido como tal debe seguir el siguiente ciclo: en primer lugar, describe un ejemplo de una mala solución; después, analiza las causas que llevaron a llegar a esa mala solución; tras ello, recopila los síntomas que hacen se reconozca que se ha aplicado una mala solución, junto con sus consecuencias, para finalmente, exponer una

solución **refactorizada** que describe cómo pasar de la mala solución a una solución bien diseñada.

La existencia de los antipatrones hace que los programadores cometan menos errores a la hora de desarrollar su código, son una plantilla que identifica un conjunto de errores aplicados y, encima, proveen una solución a los mismos de la manera más óptima.

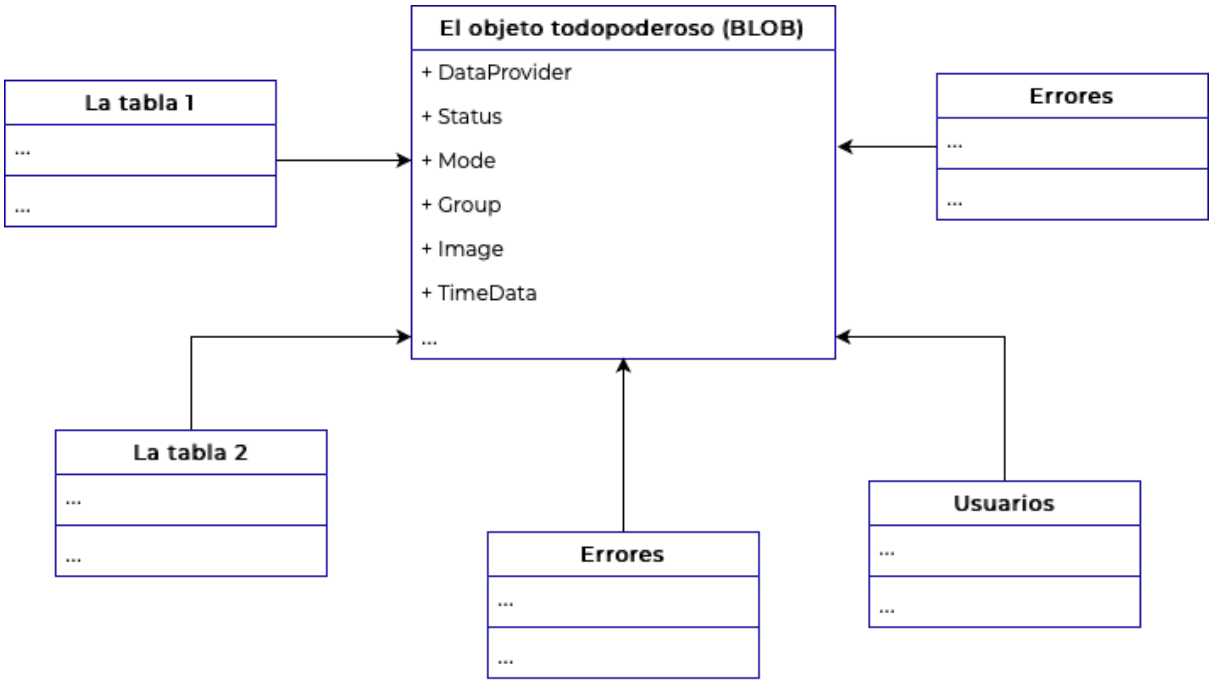
Se pueden encontrar diferentes ejemplos de categorización de antipatrones en función de la naturaleza del problema que presentan. A continuación presentaremos algunos de ellos.

Antipatrones de diseño

Los antipatrones de diseño catalogan un conjunto de problemas comunes a la hora de organizar o implementar el software junto con una serie de recomendaciones a la hora de abordar la refactorización del código para su mejora.

The Blob

Es un antipatrón de diseño que se da cuando se crea un objeto todopoderoso (*Blob*), que monopoliza todos los procedimientos y toda la lógica de la aplicación.



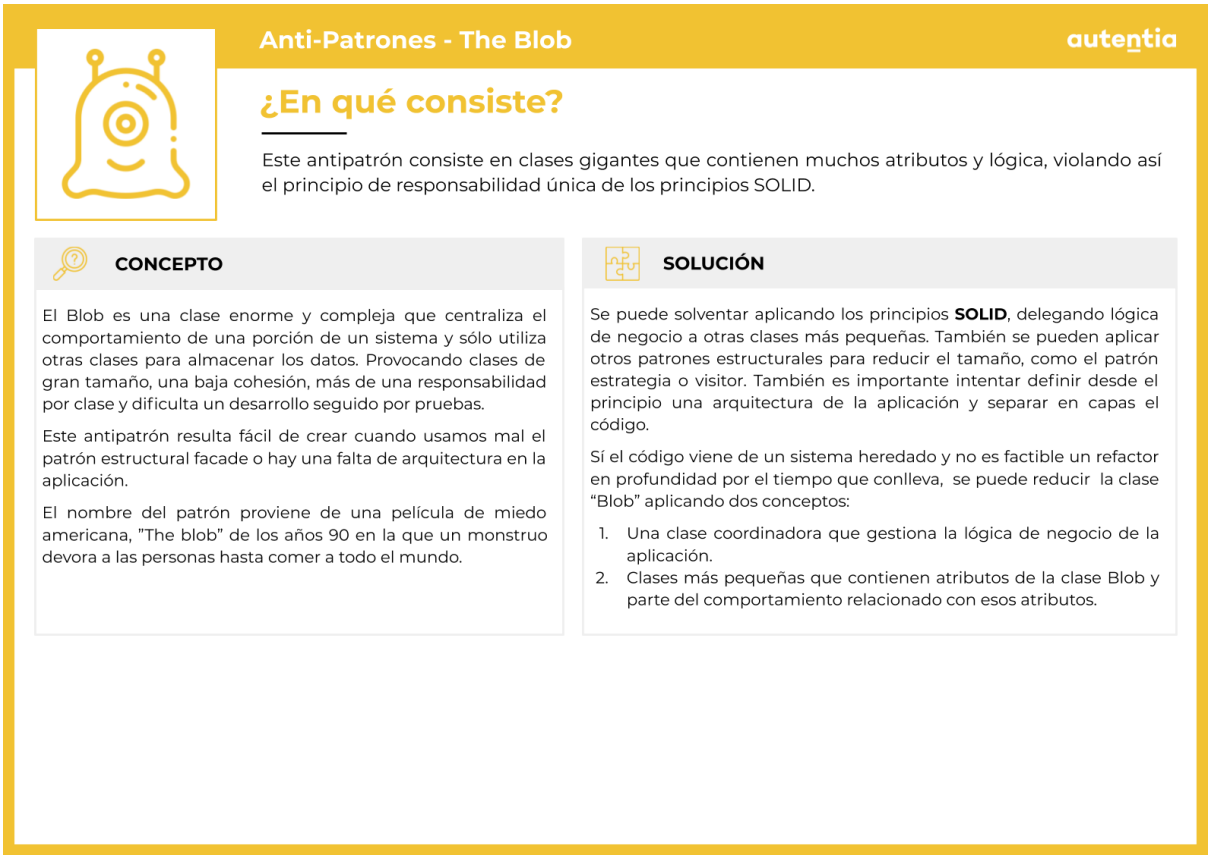
La idea básica de programación estructurada es dividir un gran problema en muchos pequeños problemas (estrategia **Divide y Vencerás**). Una vez que los pequeños problemas han sido resueltos, el gran problema ha sido resuelto también. De este modo, tenemos que planear y dividir nuestra lógica en muchas partes (objetos). Y cada objeto tiene que resolver solo sus propios problemas. Sin embargo, muchas veces por simplificar las cosas y por cualquier otra razón, los programadores tienden a crear solo un mega objeto que contiene toda la lógica. Y los demás objetos sólo contienen los datos. A menudo, el **Blob** es el resultado de un desarrollo iterativo en el que el código de prueba de concepto evoluciona con el tiempo hasta convertirse en un prototipo y, finalmente, en un sistema de producción. La asignación de responsabilidades no se reparte durante la evolución del sistema, de modo que un módulo se vuelve predominante. El Blob suele ir acompañado de código innecesario, lo que dificulta la diferenciación entre la funcionalidad útil de Blob Class y el código que ya no se utiliza (*Lava Flow AntiPattern*).

En general, el Blob es un diseño procedural, aunque puede representarse mediante los objetos e implementarse en lenguajes orientados a objetos.

Un diseño procedural separa el proceso de los datos, mientras que un diseño orientado a objetos fusiona la lógica y los datos creando objetos.

Como ocurre con la mayoría de los antipatrones, la solución implica una refactorización. Una de las claves, es dividir la lógica del Blob, puede ser apropiado mover una parte de lógica a otros objetos de manera que estos objetos sean más capaces y el Blob menos complejo. También se pueden crear nuevos objetos más pequeños para que incorporen una parte del Blob. Otra solución es crear una clase coordinadora que gestione la lógica de negocio de la aplicación.

Para prevenir la aparición de un objeto global todopoderoso (Blob) hay que procurar planear la arquitectura de aplicación antes de empezar a programar. Pensar en el dominio de nuestra aplicación y en las entidades (objetos) que lo componen.



Lava Flow

Este antipatrón se produce cuando el código de nuestra aplicación se desarrolla espontáneamente sin tener una arquitectura establecida. El código crece como lava de manera desordenada y sin ningún tipo de documentación.

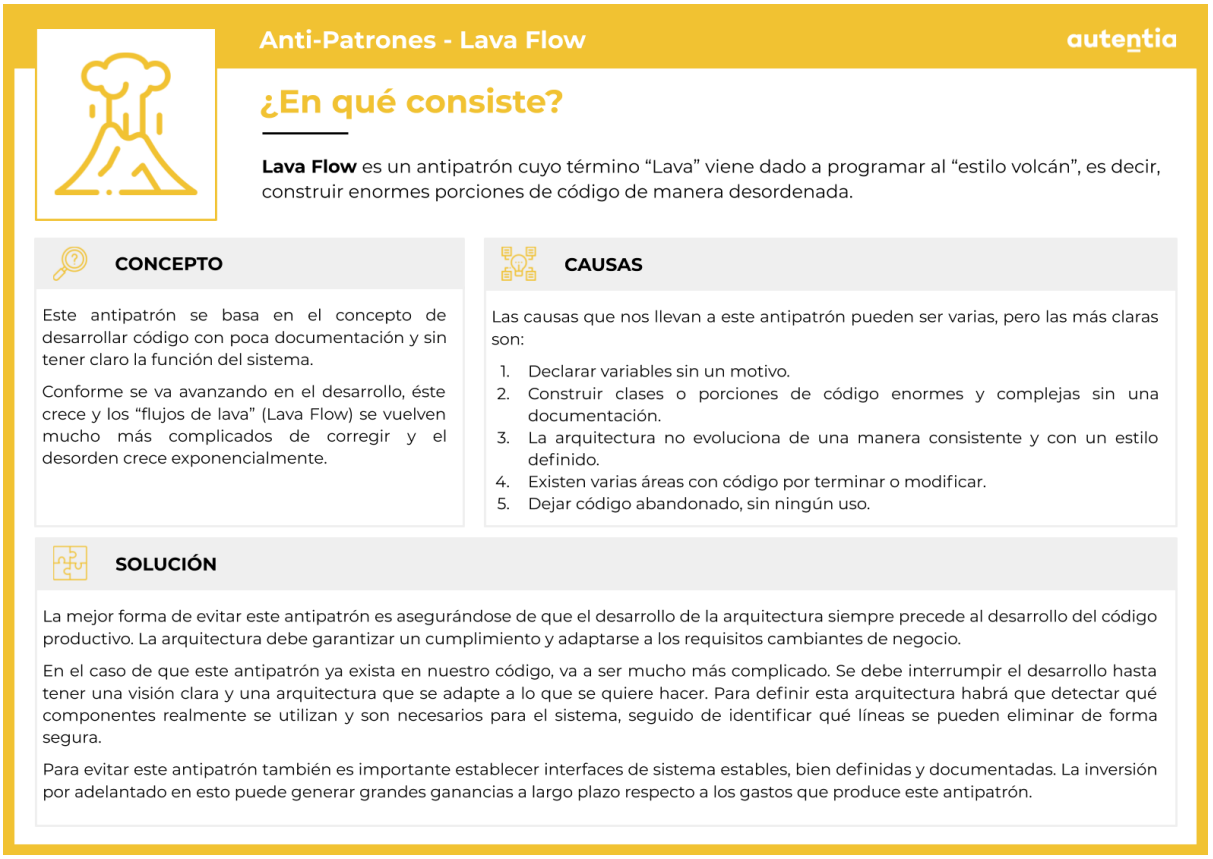
El antipatrón **Lava Flow** se encuentra frecuentemente, en sistemas que empezaron como un prototipo pero terminaron en producción. También podría ser el resultado del desarrollo anterior cuando, mientras estaban en un modo de investigación, los desarrolladores probaron varias formas de conseguir los objetivos, generalmente con prisa por ofrecer algún tipo de demostración, ignorando así las buenas prácticas de programación y sacrificando la documentación.

Aunque podría ser divertido investigar este código desordenado, los desarrolladores prefieren esquivar "el código lava" y dejarlo como está. De este modo se queda para siempre complicando el desarrollo futuro y añadiendo más incertidumbre.

Para prevenir este antipatrón hay que planificar la arquitectura del sistema antes de empezar a programar. La arquitectura tiene que adaptarse a los requisitos cambiantes de negocio.

Si el antipatrón ya existe en el código, hay que investigar todo el sistema y definir claramente las funcionalidades de cada componente. Es mejor parar el desarrollo que seguir añadiendo nuevas funcionalidades generando aún más incertidumbre. Habrá que detectar qué componentes se utilizan dentro del sistema y cuáles son necesarios para su funcionamiento. También hay que definir la nueva arquitectura basándose en los requisitos de negocio actuales. Solo después de todo el proceso de análisis, se puede empezar a refactorizar. Un elemento clave para no caer de nuevo en este antipatrón es

generar la documentación o por lo menos, dejar los comentarios explícitos en el código. La inversión de recursos en planificación y documentación nos ayudará en el futuro y hará disminuir los costes de mantenimiento a largo plazo.



Ambiguous Viewpoint

Es muy frecuente encontrarse con modelos de análisis y diseños orientados a objetos que se presentan sin aclarar el punto de vista representado por el modelo. De forma predeterminada, los modelos reflejan un punto de vista de la implementación que es potencialmente el menos útil. Los puntos de vista mixtos no permiten la separación de las interfaces frente a los detalles de implementación, que son uno de los principales beneficios del paradigma orientado a objetos.

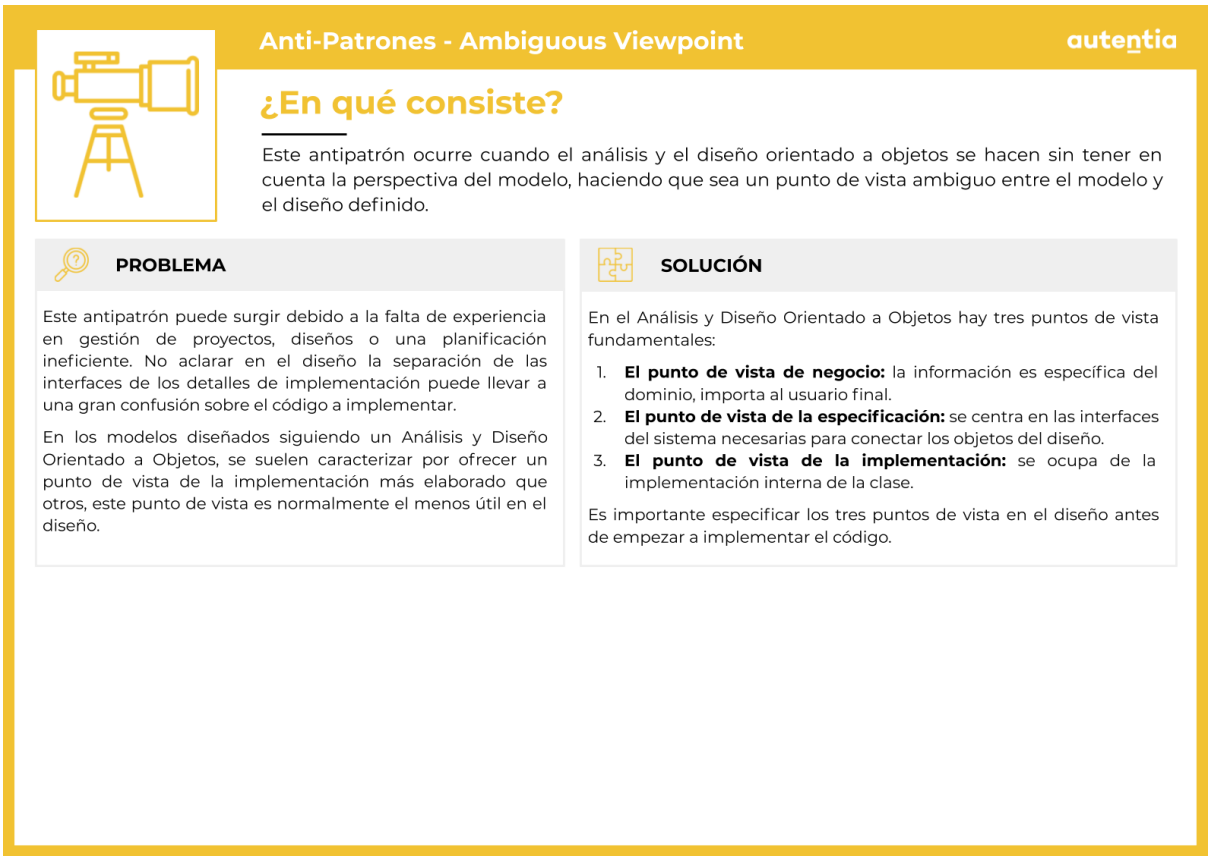
Hay tres puntos de vista fundamentales para los modelos orientados a objetos: el punto de vista del negocio, el punto de vista de la especificación y el punto de vista de la implementación.

El punto de vista del negocio define el modelo de información y procesamiento de usuario. Este es un modelo que los expertos en dominio pueden defender y explicar (modelo de análisis). Los modelos de análisis son los modelos más estables del sistema y más valiosos.

El punto de vista de especificación se centra principalmente en las interfaces de software. Define las abstracciones y el comportamiento del sistema.

El punto de vista de implementación describe los detalles de implementación de los objetos, los cuales son más cambiantes y tienen que reflejar el actual estado del sistema.

Es importante distinguir los tres puntos de vista y no mezclar los modelos.



Functional Decomposition

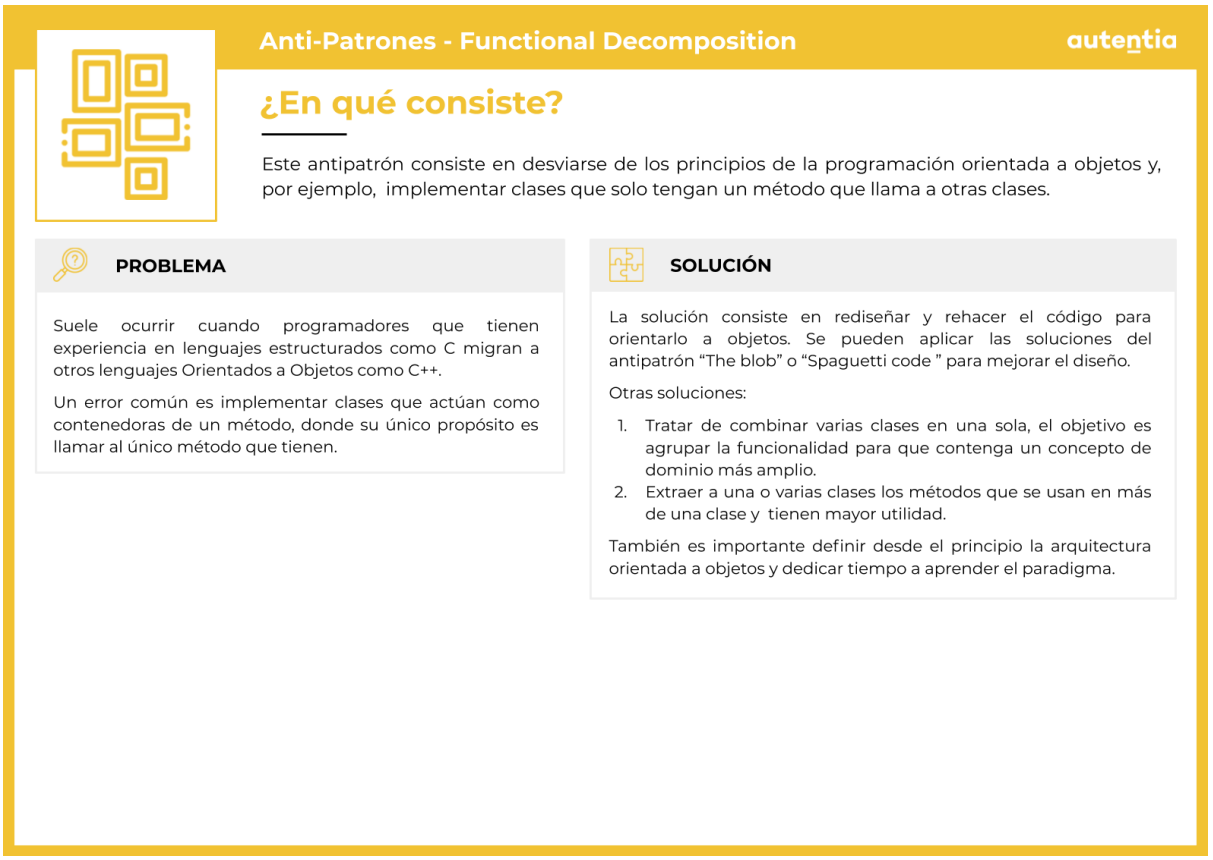
Este antipatrón consiste en aplicar los principios de la programación procedural en un entorno orientado a objetos. Suele ocurrir con los programadores que no tienen bastante experiencia en la programación orientada a objetos. Simplemente dividir el código del sistema en diferentes partes y convertirlas en clases no refleja realmente el modelo de negocio del sistema.

Un síntoma bastante frecuente de este antipatrón es una clase que solo tiene un método, que llama a su vez a otra clase. También si todos los atributos de clase son privados y se usan dentro de la misma clase. Además, también se da cuando no se usan la herencia o el polimorfismo

para construir la jerarquía de clases o no es posible reutilizar el código similar en diferentes partes del sistema.

Para solucionar los problemas que vienen con este antipatrón hay que definir el modelo de análisis para explicar las funcionalidades más importantes del sistema. Esto es esencial para descubrir la motivación de muchas de las construcciones de software en el código existente, que se han perdido con el tiempo. A continuación, hay que formular un modelo de diseño que incorpore las piezas esenciales del sistema existente.

Idealmente, el modelo de diseño explicará el significado de la mayoría de los módulos de software. Es posible que existan los módulos con el propósito desconocido. Lo importante es consolidar la funcionalidad de diferentes clases existentes en una clase que representa el concepto más amplio de dominio. Si una clase no tiene ningún estado se puede convertir en un método y dejarlo accesible para diferentes partes del sistema. Además, hay que investigar el sistema para buscar los subsistemas similares, estos son perfectos candidatos para extraer el código similar y reutilizarlo entre ellos.



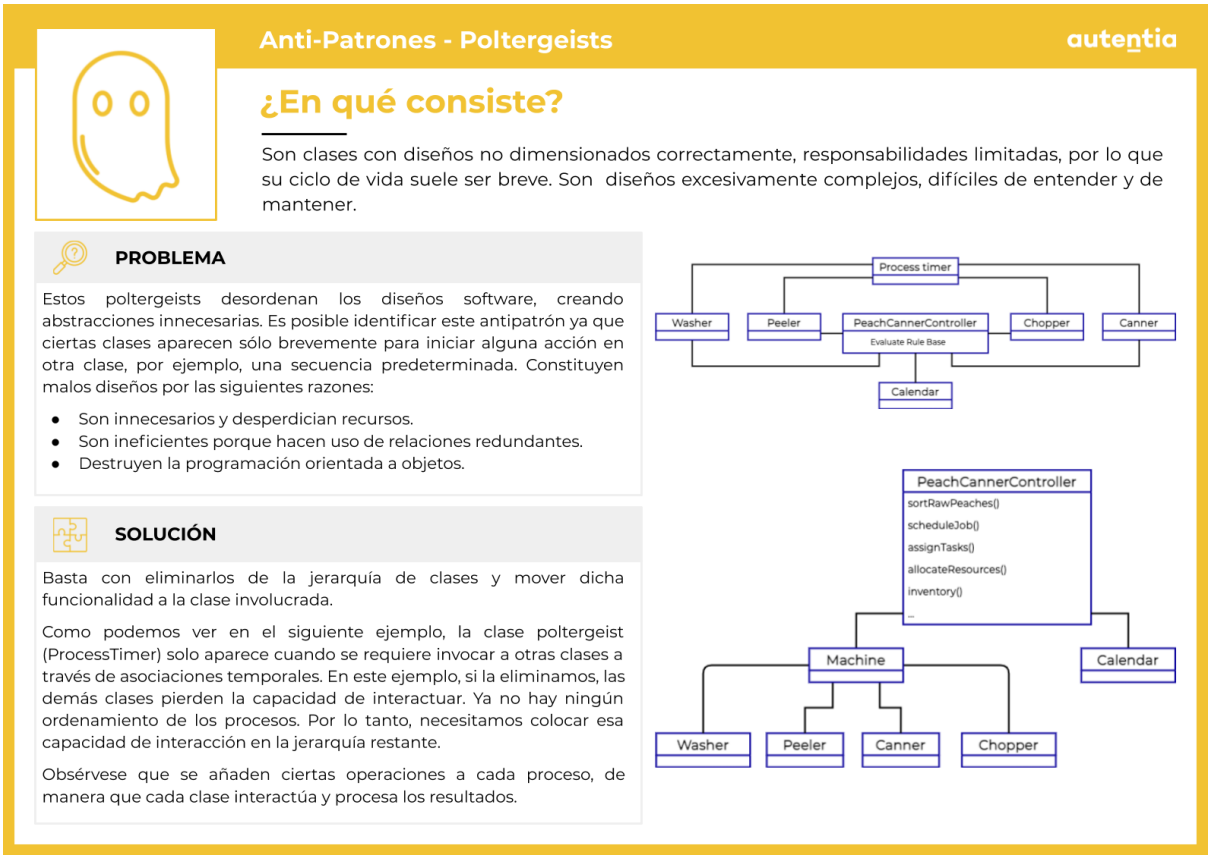
Poltergeists

Este antipatrón se da cuando aparecen las clases con responsabilidades y roles limitados en el sistema. Además su ciclo de vida es bastante breve. Esas clases no tienen ningún estado o solamente tienen una responsabilidad de llamar a otras clases. Los objetos poltergeists desordenan los diseños de software, creando abstracciones innecesarias. Son excesivamente complejos, difíciles de entender y difíciles de mantener. Este antipatrón es típico en los casos en los que los arquitectos de sistemas son nuevos en el diseño orientado a objetos. En los sistemas proyectados por ellos es posible identificar una o más clases con el significado desconocido que aparecen sólo un rato para iniciar alguna acción en otra clase. Los objetos así no tienen ningún valor por sí mismos.

Las clases de poltergeist son un error por tres razones clave: son

innecesarias, por lo que desperdician recursos cada vez que "aparecen". Son ineficientes porque utilizan varias rutas de navegación redundantes. Se interponen en el camino del diseño orientado a objetos adecuado al saturar innecesariamente el modelo de objetos.

Se resuelven los casos de los Poltergeists eliminándolos por completo de la jerarquía de clases. Sin embargo, después de su eliminación, la funcionalidad que fue "proporcionada" por el poltergeist debe ser reemplazada. La clave es mover las acciones de lanzamientos inicialmente encapsuladas en las clases Poltergeist a las clases que ellas invocan.



Spaghetti Code

Este antipatrón es quizá el más famoso de todos, siempre se ha comentado en artículos o libros de una u otra forma. El **Spaghetti Code** se aplica usualmente cuando se codifica en lenguajes de programación no orientados

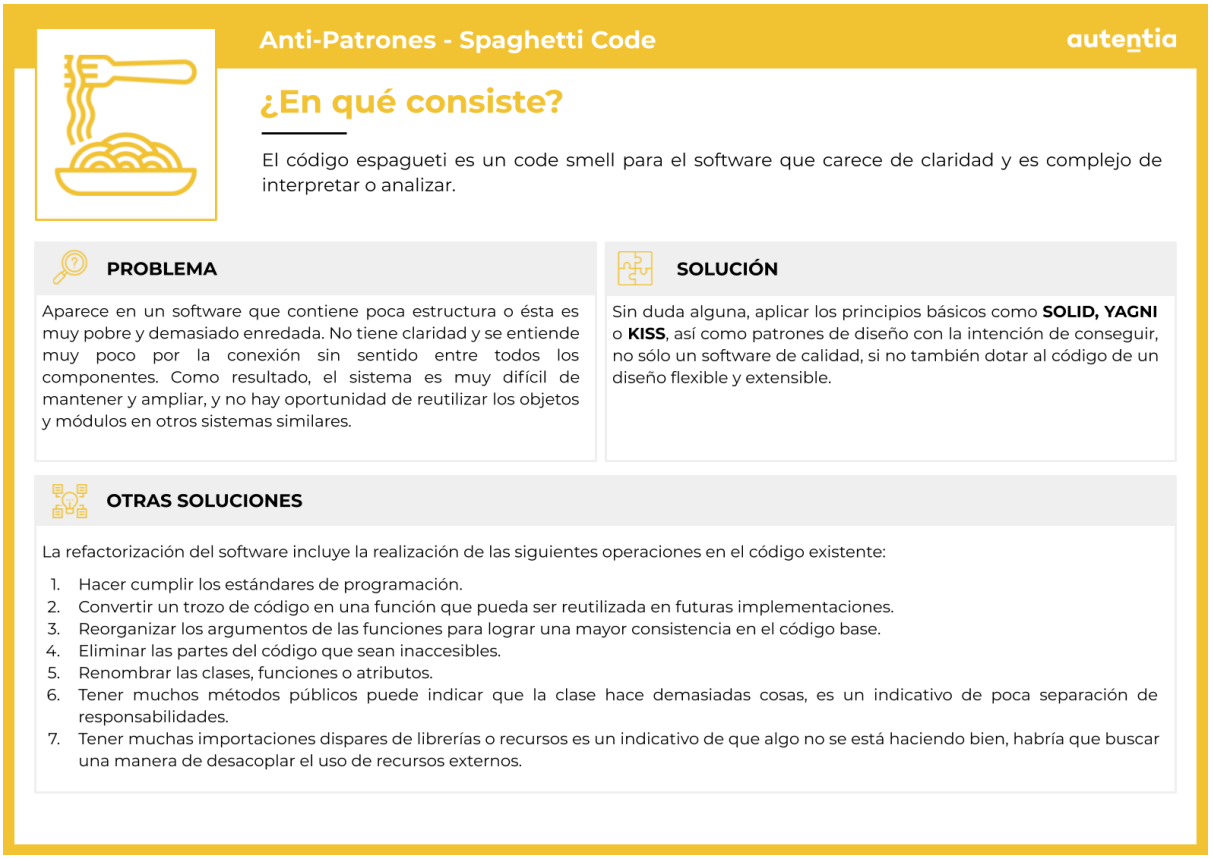
a objetos, aunque también puede existir en lenguajes orientados a objetos cuando no se han dominado completamente los conceptos avanzados de este paradigma.

Este antipatrón se da cuando un trozo de código no está documentado y donde cualquier pequeño cambio o modificación hace tambalear la estructura completa del sistema. Este antipatrón puede confundirse con **Lava Flow**, pero a diferencia de este último donde el principal problema reside en la forma en que el sistema crece, aquí el problema reside en la forma en la que se escribe el código, de forma que existen relaciones mínimas entre objetos, los métodos están muy orientados a los procesos, e inclusive, algunos objetos se llegan a nombrar como procesos. Se pierden beneficios de la programación orientada a objetos y no se utiliza ni la herencia ni el polimorfismo.

En resumen: esto se da cuando un sistema tiene una estructura de control de flujo demasiado compleja e incomprensible.

Algunas de las causas que nos pueden llevar a implementar este antipatrón puede ser la propia inexperiencia del desarrollador con tecnologías orientadas a objetos: no existen revisiones de código o estas son ineficaces, y no hay un proceso de diseño y análisis antes de implementar la solución.

La mejor solución a este antipatrón es la refactorización, es decir, la limpieza de código. Otras soluciones consisten en aplicar buenas prácticas y principios, como **SOLID**, **DRY**, **KISS**, etc. Hay que destacar que antes de implementar estas soluciones conviene prevenir, por tanto, insistir en un proceso de análisis orientado a objetos para crear el modelo del dominio, abordar los requisitos del sistema y la variabilidad a la que nos podemos enfrentar, asegurarse de que los objetos se descomponen a un nivel en el que los programadores son capaces de comprenderlos y realizar una implementación basada en un diseño previo.



Input Kludge

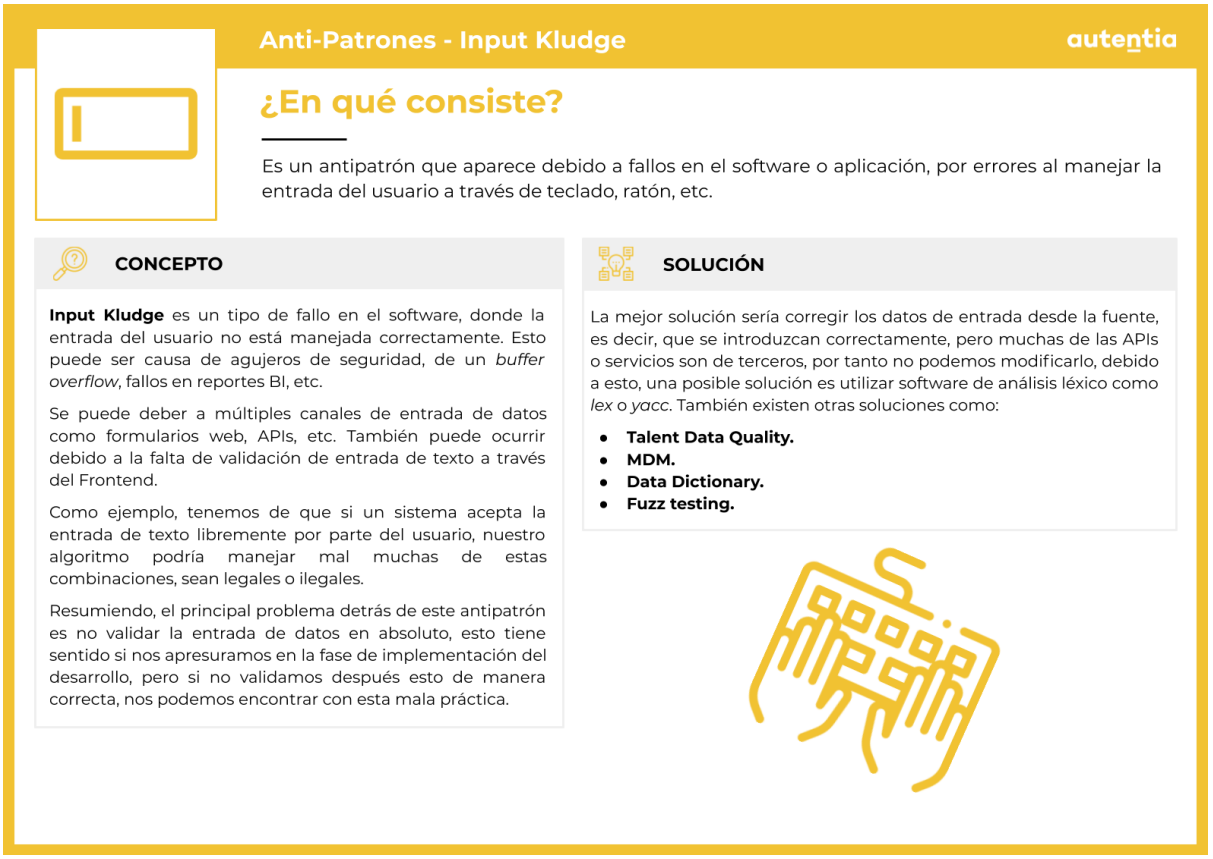
Este antipatrón se da cuando la entrada de información de un sistema o aplicación no se maneja de la manera adecuada. Por ejemplo, si tenemos una aplicación en la que podemos introducir cualquier tipo de dato, sea válido o no, y lo que va a hacer el sistema es manipular esta entrada para que sea correcta siempre mediante algún algoritmo. Además, los datos de entrada pueden llegar a través de múltiples canales como formularios web, APIs, etc. Por tanto, es bastante probable que se den entradas no válidas.

El mal uso de los datos de entrada puede suponer un agujero de seguridad importante en los sistemas, como por ejemplo los desbordamientos de búfer.

Como solución a esto tenemos algoritmos de validación de entrada de datos, que nos permiten únicamente introducir datos que sean válidos y

descartar aquellos que no lo sean. Por ejemplo, si nos encontramos en un formulario web, en el que se pide en un campo del formulario la edad, que sea un componente que solo acepte caracteres numéricos, de esta forma no llegarán entradas inválidas al sistema.

Se recomienda el uso de software que realice el análisis sintáctico como **Lex**, **Yacc** o **GNU Bison**, además de otras más, de esta forma tendremos un control robusto del texto a través de expresiones regulares sin tener en cuenta el contexto del lenguaje.



Antipatrones de metodología

Los antipatrones metodológicos agrupan errores típicos en la forma de abordar nuestro trabajo, vicios adquiridos y técnicas poco cuidadas.

Golden Hammer

Este es uno de los antipatrones más comunes en el mundo del software. Principalmente, es utilizar la misma herramienta para resolver cualquier problema, por ejemplo, utilizamos el mismo *framework* o el mismo lenguaje de programación, independientemente de las necesidades del problema en cuestión. Esto generalmente, sucede cuando la directiva de una empresa o los desarrolladores se sienten cómodos con una tecnología en cuestión y no están dispuestos a aprender algo nuevo. Dado el coste que supone crear una solución específica, los defensores de este antipatrón muchas veces argumentan que el coste es demasiado alto y que es preferible utilizar algo ya implementado o que ya conocen, por lo que a parte de disminuir el coste también disminuye el riesgo.

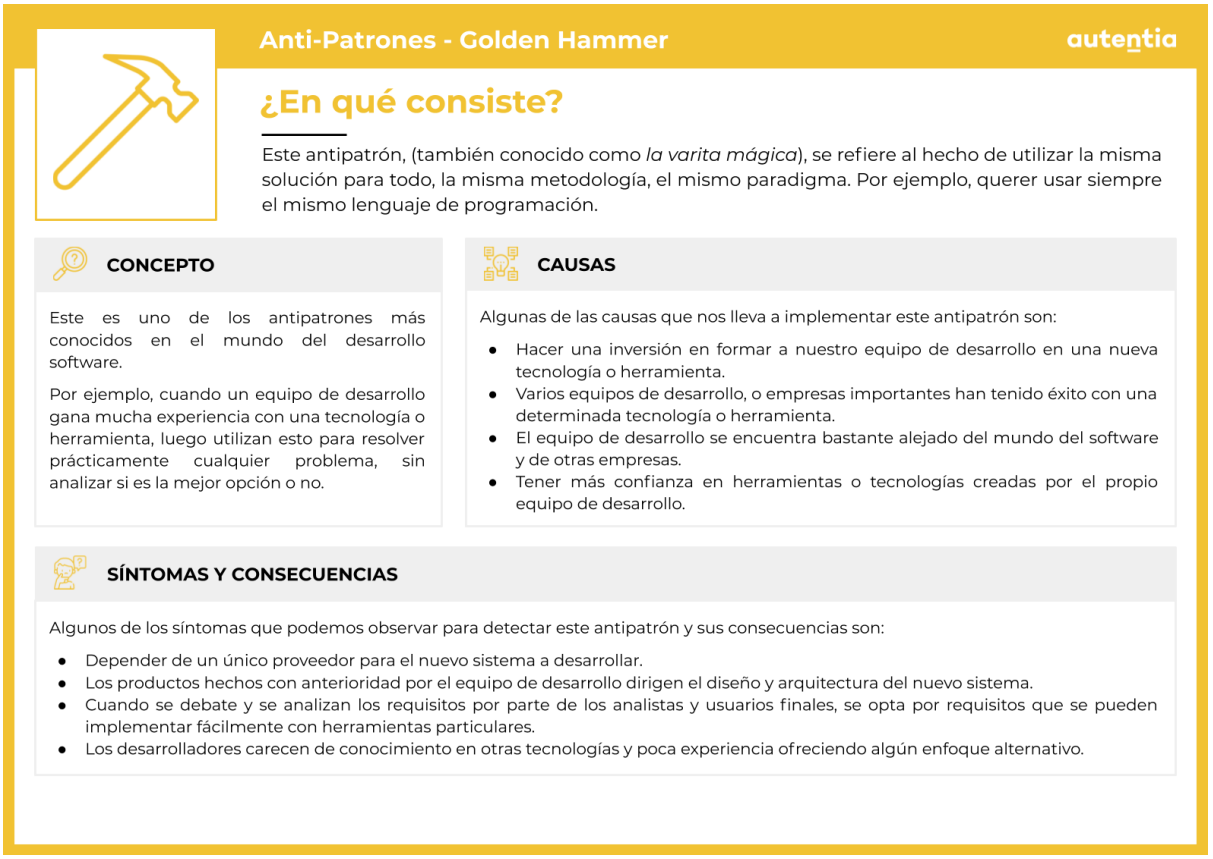
Entre las causas más comunes que nos encontramos para implementar este antipatrón, tenemos:

- Haber tenido éxito con un enfoque en varias ocasiones. Esto nos hace pensar que esta solución puede ser la más correcta en muchos casos.
- Existen grandes inversiones detrás de una herramienta.
- El equipo de desarrolladores solo conoce las herramientas que propone la organización y no está al tanto de nuevas tecnologías.
- Tener más confianza en productos propios que en ajenos.

Para solucionar este antipatrón necesitamos cambiar el pensamiento de la organización, se necesita desarrollar un compromiso por el continuo

aprendizaje y la exploración de nuevas tecnologías. Sin este compromiso, podemos caer en la excesiva dependencia de una tecnología o herramienta. Incluso sin una correcta gestión, los desarrolladores pueden organizarse para establecer redes de información con otros desarrolladores que estén al tanto de nuevas tecnologías.

Otra forma de evitar esto es fomentar la contratación de personas de diferentes áreas, ya que si en nuestra empresa solemos trabajar con bases de datos relacionales, quizá contratar a alguien con experiencia en bases de datos no relacionales nos haga ver un punto de vista diferente y no depender de solo una tecnología.



Cut-And-Paste Programming

Este antipatrón es muy utilizado a la hora de reutilizar código pero crea importantes problemas a la hora del mantenimiento del sistema. Se parte de la idea, de que es más fácil modificar el software existente que crear uno desde cero, esta idea no está mal del todo, pero podemos caer en un uso excesivo de esto.

Este antipatrón es fácil de identificar por la presencia de varias porciones de código similares intercalados en diferentes ficheros a lo largo del sistema o aplicación. Esto es común cuando tenemos gente nueva en el equipo o con poca experiencia, que se fijan en los ejemplos del código ya creado para hacer sus desarrollos. Esto puede crear duplicidad de código, que no es ideal.

Para detectar este antipatrón podemos fijarnos en lo siguiente:

- Tenemos un error igual en varias partes del software a pesar de que se ha corregido localmente.
- Cuando se hace una *code review*, esta se alarga innecesariamente.
- Muchas partes del código se pueden reutilizar sin mucho esfuerzo.
- El software tiene defectos o errores que se replican a lo largo de la aplicación.
- Vemos que las líneas de código de nuestro software aumentan pero la productividad no.
- Es complicado localizar y corregir todas las apariciones de un error en particular.
- Existen costes excesivos en mantenimiento del software.
- Se crean varias correcciones para errores de manera individual, sin tener una solución estándar.
- Las partes del software que se podrían reutilizar no son fáciles de

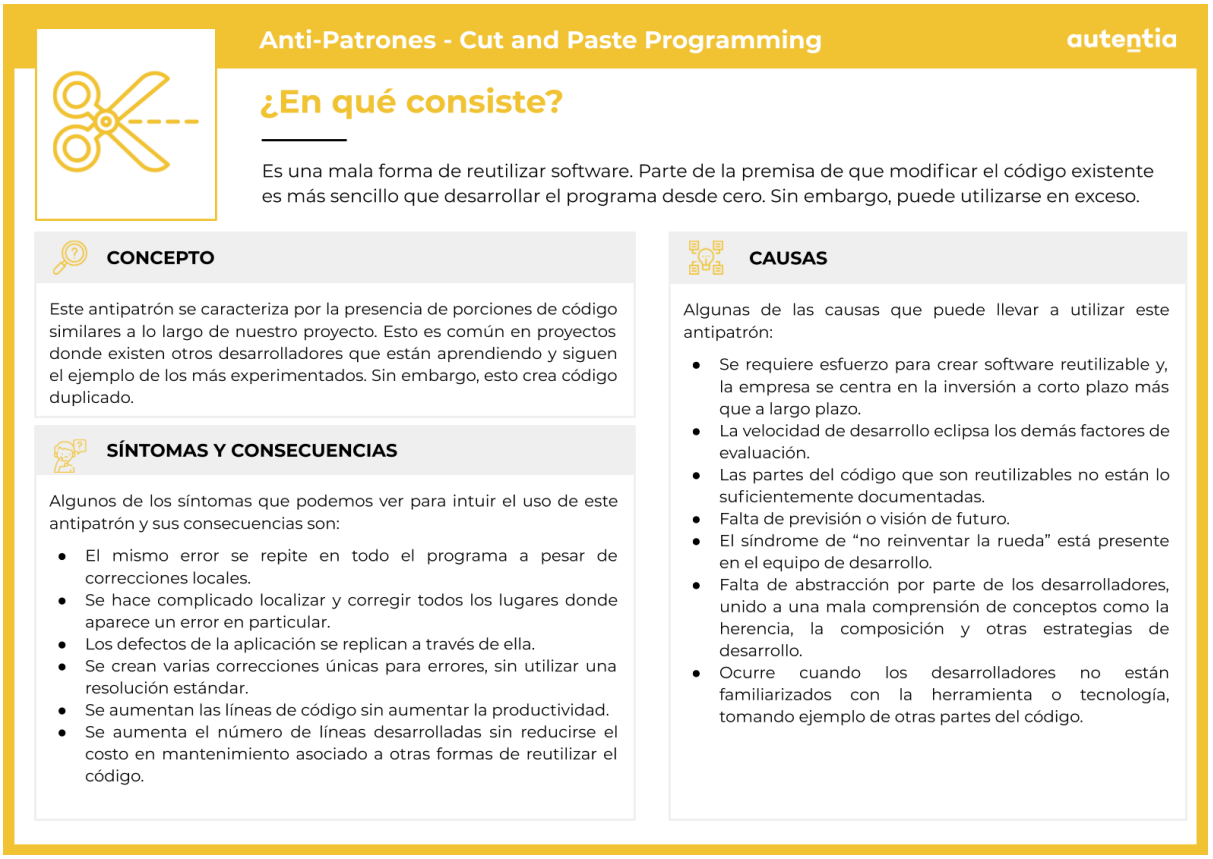
volver a usar.

Existen varias razones por las que se produce este antipatrón, entre ellas tenemos, que se necesita mucho esfuerzo para crear código reutilizable y la empresa prefiere la ganancia a corto plazo que la inversión a largo plazo, que existe falta de abstracción por parte de los desarrolladores, además de falta de comprensión sobre la herencia, la composición, etc. Mucha gente utiliza en exceso el dicho de “no reinventar la rueda” y copian y pegan lo que ya existe, esto es bastante probable que suceda cuando las personas no están muy familiarizadas con las tecnologías que están usando por lo que usan ejemplos de otros compañeros.

Algunas veces, los equipos de desarrollo se enfrentan a muy poco tiempo para resolver problemas, por lo que copiar y pegar podría ser una solución a corto plazo, pero que nos generará muchos gastos en el futuro.

La solución a este antipatrón empieza por tener tests en tu código que te permitan modificarlo asegurándose de no romper nada, luego empieza por encontrar duplicaciones en el código y viendo si se puede extraer la lógica a una sola clase por ejemplo.

Otra parte de la solución es cambiar el pensamiento del equipo de desarrollo, acostumbrarse a no dejar código malo en el software, a intentar aplicar soluciones más complejas pero eficientes, a realizar code reviews, etc. También conlleva un cambio de pensamiento en la organización, si se valora antes la entrega rápida que la calidad, no habrá nada que hacer.



Antipatrones de gestión

Este conjunto de antipatrones abordan problemas organizativos y de planificación a la hora de afrontar el desarrollo de software.

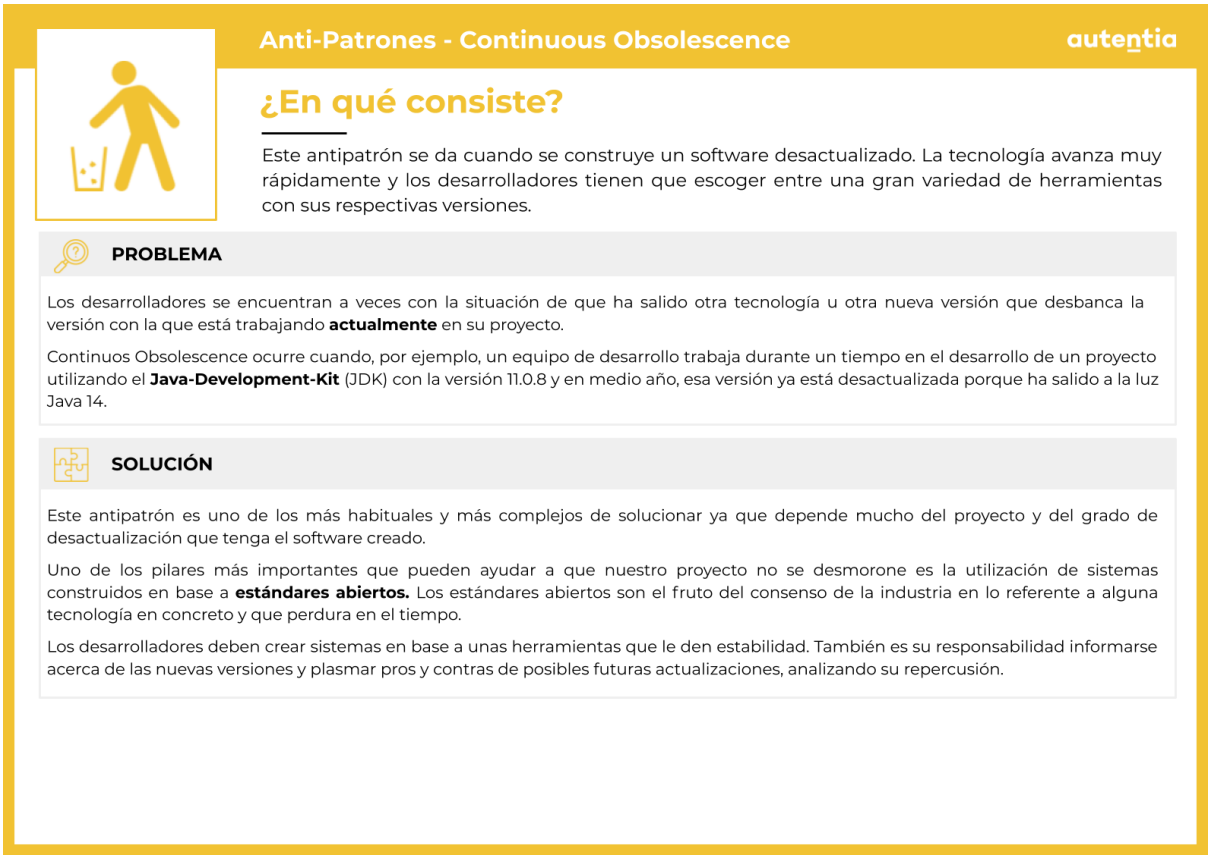
Continuous Obsolescence

Este antipatrón aparece cuando debido al paso del tiempo, estamos obligados a actualizar constantemente los componentes de nuestro sistema y las herramientas de desarrollo. Las tecnologías cambian tan rápidamente que los desarrolladores tienen problemas para mantenerse al

día con las versiones actuales de software y encontrar combinaciones de versiones de productos que funcionen en conjunto. Otra razón es que, lanzando más a menudo las nuevas versiones de software, las empresas tecnológicas obtienen más ingresos.

Cada línea de productos comerciales evoluciona lanzando cada vez más nuevas versiones y la situación se vuelve cada vez más difícil de afrontar para los desarrolladores. El reto de encontrar versiones compatibles de software que funcionan conjuntamente con éxito es muy difícil y al final, podríamos estar en una situación insostenible cuando no podemos implementar la nueva funcionalidad sin actualizar las dependencias, pero al hacerlo, tendremos que rehacer buena parte de nuestro sistema y gastar muchos recursos sin obtener ningún beneficio visible.

La solución a este problema es utilizar software construido en base a estándares abiertos. Los estándares abiertos ayudan a estabilizar el ritmo de las actualizaciones de software y perduran en el tiempo. Hay que elegir las herramientas de desarrollo pensando en el futuro, el soporte que tendremos, sopesando pros y contras de cada posible actualización.



Mushroom Management

Este antipatrón de gestión consiste en mantener a los desarrolladores alejados de los usuarios finales del software, esto implica más riesgo de que finalmente se construya o se diseñe el sistema inadecuado. Muchas veces los requisitos de un sistema, llegan a los desarrolladores a través de intermediarios, sin la participación de estos, dado que se tiene la falsa creencia de que los requisitos son estables y son bien entendidos por los usuarios finales.

Pero, hay varias cosas a tener en cuenta, la primera de ellas es que los requisitos no suelen ser estables, sino al contrario, estos suelen cambiar frecuentemente, por tanto, si aplicamos este antipatrón, estos cambios no serán descubiertos hasta la entrega del producto final. Además, la documentación de los requisitos frecuentemente no es comprendida por

los usuarios finales, y por último cuando los integrantes del equipo de desarrollo no son capaces de entender los requisitos globales y el producto, no podrán comprender las necesidades y por tanto se tomarán decisiones de diseño incorrectas.

Todo esto genera un ambiente de incertidumbre sobre el proyecto.

Como solución a este problema, podemos desarrollar el software basándonos en prototipos y comentarios de los usuarios finales, de esta forma, dado que se evalúa de manera constante la aceptación de las funcionalidades por parte del usuario final, el riesgo a un rechazo será mucho menor. Otra opción puede ser la de incluir alguien que conozca mejor el negocio y se pueda integrar con el equipo de desarrollo, que sería algo así como la figura del *Product Owner* en **Scrum**.

Dead End

Este antipatrón se da cuando el equipo de desarrollo modifica una porción de código o componente reutilizable cuando el proveedor ya no mantiene este componente. En el momento en que los cambios son aplicados, el mantenimiento de este se transfiere a los desarrolladores internos, ya que el código es externo, las funciones y correcciones de errores son más difíciles de integrar, a corto plazo esto puede suponer el progreso y avance en el desarrollo, pero a largo plazo es insostenible cuando se intentan compaginar las nuevas versiones del proveedor, la actualización del sistema y el mantenimiento.

Cuando esta personalización del componente o software de terceros no se puede evitar, lo recomendable es utilizar alguna capa de aislamiento u otras técnicas para separar las dependencias.

Así mismo, también podríamos ponernos en contacto con el proveedor para seguir sus indicaciones y utilizar sus herramientas.

La personalización de componentes o software de terceros, solo debe llevarse a cabo cuando el beneficio es muy grande.

Boat Anchor

Este antipatrón se da cuando se adquiere software o hardware que no ofrece ninguna utilidad en el proyecto. Frecuentemente esta adquisición es costosa, lo que hace que la compra sea aún más perjudicial.

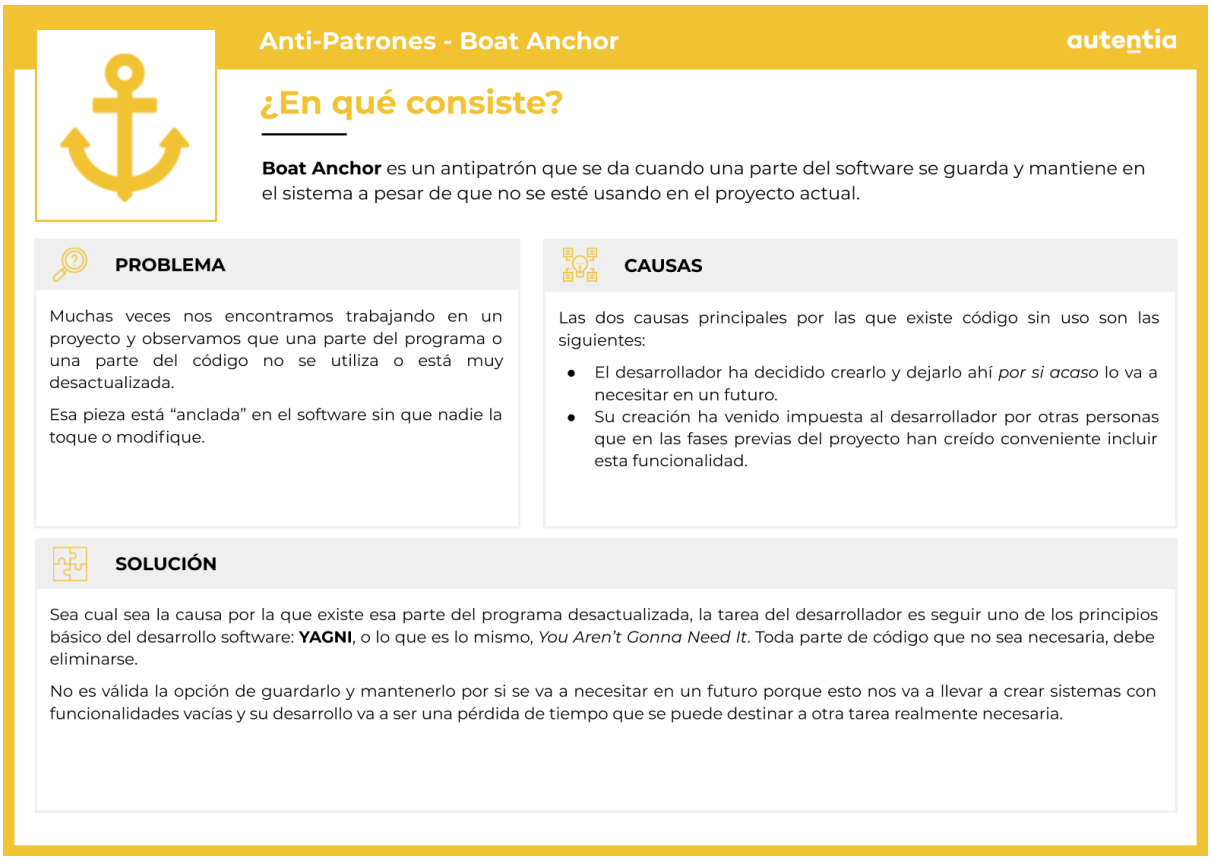
Muchas veces, desde la parte directiva de la organización se realiza una adquisición de software o de hardware porque algún gerente clave está convencido de su utilidad en ese momento.

También sucede que las empresas que venden productos software o hardware, lo hacen a los directivos, sin que alguien con conocimientos más técnicos lo evalúe. Las consecuencias para los gerentes y desarrolladores es que es posible que se tenga que dedicar un esfuerzo mayor a que el

producto funcione.

Tras toda la inversión en cuanto a tiempo y recursos, los desarrolladores se dan cuenta de que dicho producto no sirve en el contexto actual y simplemente se abandona.

Este antipatrón es de los pocos que tiene una solución clara y concisa que se conoce como **YAGNI**: todo aquello que no sea necesario debe eliminarse. Si observas que una parte del software se comporta como un **Boat Anchor**, no te lo pienses y simplemente elimínalo. En cuanto al hardware, resulta más difícil deshacerse de él.



Walking through a Minefield

Si en el software constantemente aparecen nuevos errores, el sistema es inestable y los usuarios no pueden ejecutar sus tareas con normalidad, esto

es el caso de este antipatrón de gestión. Trabajar con el sistema que sufre las consecuencias de este antipatrón es como andar por un campo de minas.

Una de las principales causas es publicar el software sin *testearlo* y usar a los usuarios como *testers*. También los errores de implementación y diseño pueden afectar al funcionamiento del sistema. A veces el sistema no puede cumplir las expectativas del cliente por la mala comunicación previa entre el cliente y el equipo de desarrollo.

Para resolver el problema de mala calidad del código es importante llevar a cabo las buenas prácticas de desarrollo como las pruebas Test-Driven Development, tener un equipo externo de pruebas, etc.

Parte 2

Principios y patrones de la arquitectura del software

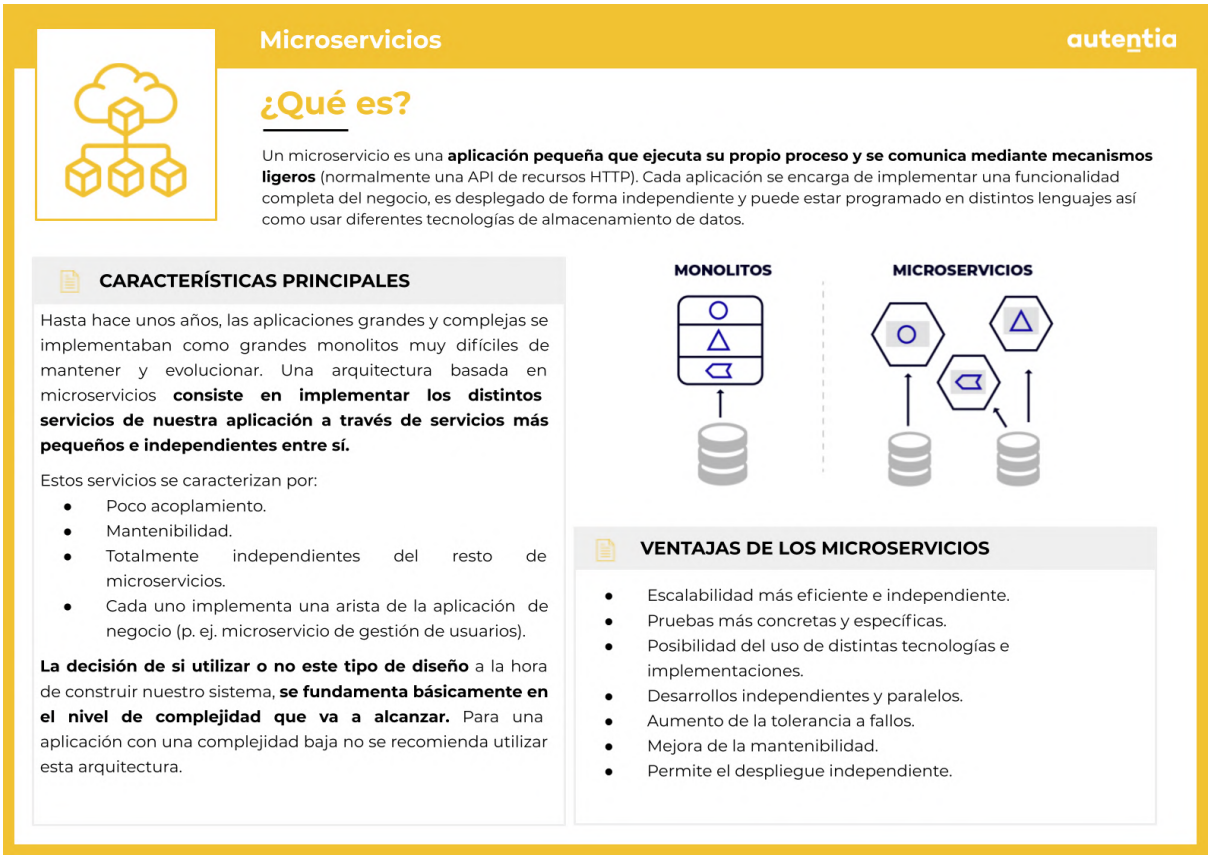
Introducción a la Arquitectura de Software

La arquitectura de software como concepto es aquel conjunto de reglas que se definen para diseñar y estructurar los artefactos y piezas que conforman nuestras aplicaciones. Esto normalmente, se hace porque existen una serie de beneficios interesantes a la hora de homogeneizar un sistema software. También es importante recalcar que la propia arquitectura debe venir definida por el contexto de la aplicación que se pretende desarrollar, y dar respuesta a las necesidades de los equipos que la están implementando.

Así también, hay decisiones arquitectónicas tales como empezar un proyecto con un monolito o iniciarlo con una estructura de microservicios, donde muchas veces, se da por sentado que cualquier cosa sería de mucho más valor iniciarlo como un microservicio u otra arquitectura de moda. ¿Por qué? Puede ser porque es la tendencia, porque es lo que domina el equipo de desarrollo o porque las herramientas más modernas de hoy en día están orientadas a microservicios y, no nos engañemos, eso vende. Sin embargo, hace falta un análisis más profundo de qué es lo que aporta cada modelo o patrón de arquitectura y cuáles son las ventajas y desventajas que tenemos al usar una u otra arquitectura. En el mundo del software no existen las "*balas de plata*", por lo que tendremos que ser conscientes de a qué inconvenientes nos vamos a enfrentar.

Así que desde nuestra experiencia abordaremos los patrones y arquitecturas que más se suelen encontrar los desarrolladores, recordando

siempre que la arquitectura debe ser algo que florezca de forma natural dependiendo de la naturaleza del producto o negocio, para ayudarnos a solucionar los problemas, sin crearnos nuevos. A esta forma de definir la arquitectura en base a las necesidades se le denomina Arquitectura Emergente.



DDD: Domain-Driven Design

Desarrollar software no es algo sencillo. Quizás uno puede considerar que la complejidad está relacionada con la parte más tecnológica del desarrollo como la seguridad, acceso a datos, integración con sistemas de terceros, despliegue... y cierto es, que eso no es sencillo, pero la mayoría de los proyectos, cuando fracasan, no lo hacen por problemas técnicos que no pudieron resolverse, sino por una incorrecta o incompleta comprensión del modelo de negocio o por una deficiencia en su evolución y desarrollo posterior. El propósito del DDD es precisamente proporcionar una serie de herramientas y procedimientos que permitan tomar decisiones de diseño acertadas, poniendo en el centro y origen de ellas el negocio.

Building blocks

En esta sección se definen los bloques que hacen que el modelo y la implementación estén alineados entre sí. Se dividen en dos partes, el modelo expresado como software (*Entities*, *Value Objects*, *Modules* y *Services*) y el ciclo de vida de un objeto de dominio (*Aggregates*, *Factories* y *Repositories*).

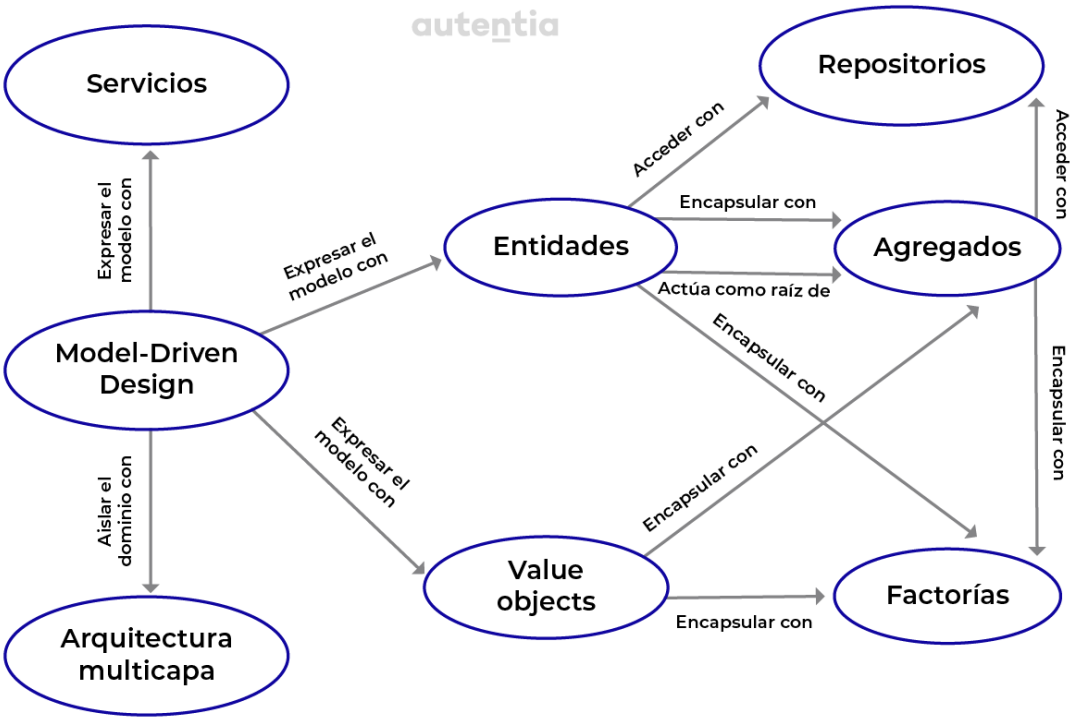
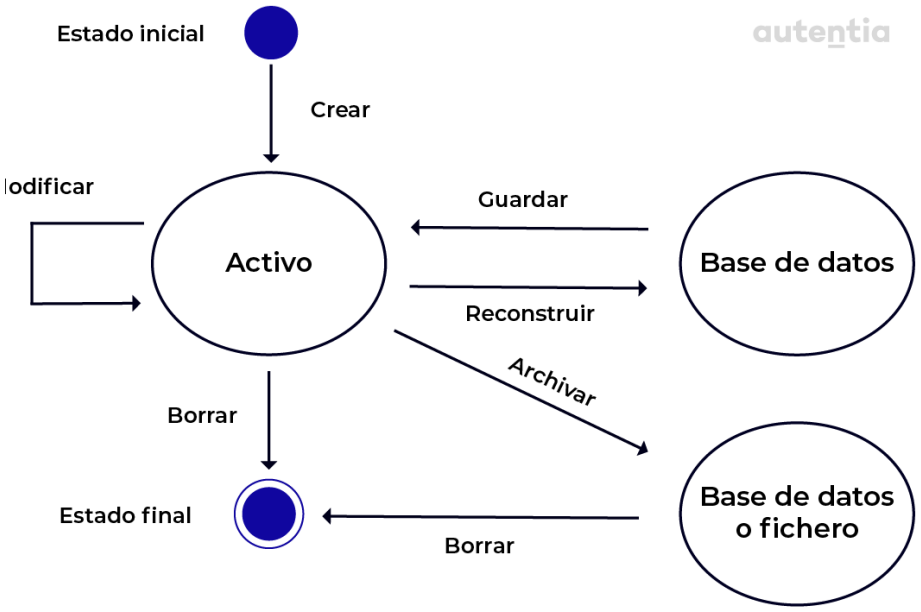


Diagrama de relación entre los building blocks

Entities

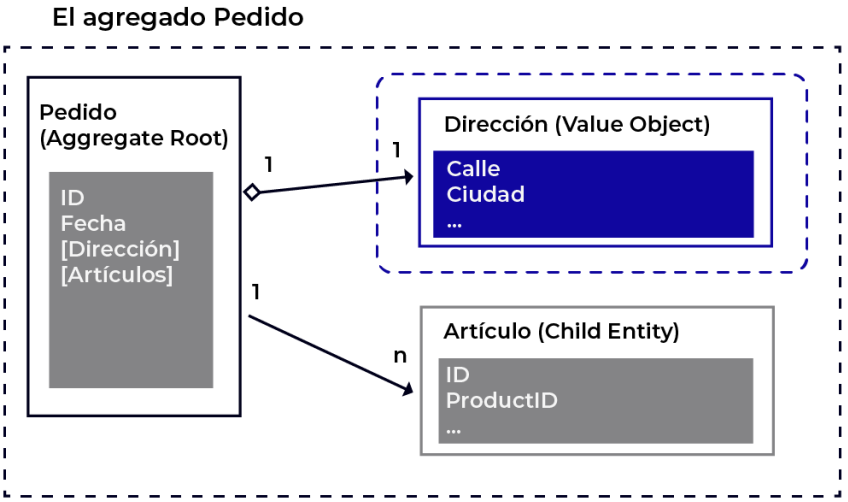
Una entidad en DDD es un objeto dentro de un dominio que puede estar compuesto por varios *value objects* para enriquecer su significado. El concepto de entidad en un dominio es todo aquel objeto que tiene vida propia y tiene un ciclo de vida definido. El objetivo de definir entidades en nuestro dominio es evitar trabajar con tipos primitivos y aportar semántica a nivel de código, y también facilitar el hecho de llevar la lógica de negocio hacia el dominio. Cuando un objeto es más relevante por su identidad que por sus atributos, entonces debemos considerarlo una entidad. En estos objetos es importante conocer qué les hace ser únicos.



Ciclo de vida de una entidad.

Value objects

Un *value object (VO)* es un conjunto de valores, que no tiene identidad propia. Su identidad se basa en su estado (los valores de sus campos). Un *Value Object* se usa para modelar muchos conceptos de nuestro sistema,



autentia

como pueden ser identificadores, fechas o rangos de fechas, precios, pesos, velocidades o incluso títulos, nombres o direcciones. Las entidades, por ejemplo, se componen de *Value Objects* o de otras entidades.

Los *Value Object* se crean para que midan, cuantifiquen o describan un concepto de nuestra capa de dominio.

Los *Value Object* deberían ser siempre objetos inmutables. Dado su limitado tamaño, su construcción no tiene un fuerte impacto en el consumo de memoria, por lo que es preferible la creación de una nueva instancia antes que la modificación de una ya existente, evitando así los efectos secundarios derivados de la modificación de los mismos.

No hay que confundir nunca una entidad con *un Value Object*. La primera siempre tiene una identidad, un identificador que la hace única. Un *Value Object* en cambio, no tiene identidad por lo que las comparaciones entre *value objects* deben hacerse basándose en su contenido y no en un identificador o una referencia.

Modules

Los módulos son agrupaciones de un conjunto de elementos (como clases, interfaces...), ya que juntos dan vida a una funcionalidad y por lo tanto, son elementos que tienen sentido dentro de un dominio o *bounded context*. Por ejemplo, teniendo *Atención al cliente* como *bounded context*, tendremos como posibles módulos *Cliente*, *Producto* y *Ticket*.

Existen una serie de aspectos a tener en cuenta cuando trabajamos con los módulos:

- Tener bien definido el lenguaje ubicuo y respetarlo en el nombramiento de los módulos, de tal manera que cualquier miembro del equipo, al ver el nombre de un módulo, entienda para qué se usa.
- Procurar que los módulos sean lo más pequeños y simples posible, intentando mantener una fuerte cohesión en los mismos.
- Mantener el acoplamiento al mínimo, garantizando la testabilidad del módulo de forma aislada.

Servicios

Los servicios en DDD son acciones y operaciones que no pertenecen directamente ni a una entidad ni a un *Value Object* pero que interactúan con todos ellos, por lo que el objetivo de los servicios es desacoplar las

tareas en distintas capas como infraestructura, aplicación y dominio.

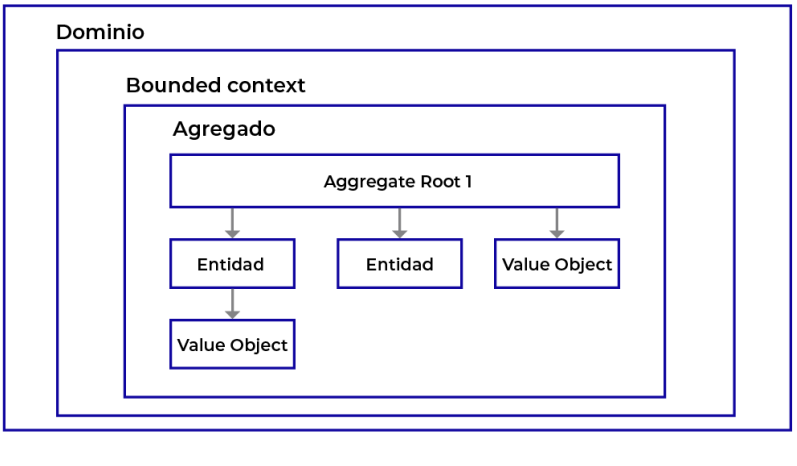
Los servicios se definen por la acción que entregan, no por un concepto de negocio. Por ejemplo, los servicios en la capa de aplicación se pueden mapear como casos de uso donde existen múltiples interacciones con las entidades, *value objects*, agregados, etc. Otro aspecto interesante a tener en cuenta es que los servicios son objetos donde sus operaciones no tienen estado (recomendable). Esto viene a significar que cuando nosotros realicemos una llamada o se lance un evento, el servicio no tenga que depender de llamadas pasadas que se le realizaron.
También hay que destacar que los servicios reciben y devuelven conceptos de negocio, es decir, entidades y *value objects*, y además, sus operaciones no se pueden reflejar nunca en entidades o *value objects*.

Según en la capa que nos encontremos, nuestros servicios tienen ciertas tareas, por lo que pasamos a enumerar los 3 tipos de servicios más conocidos:

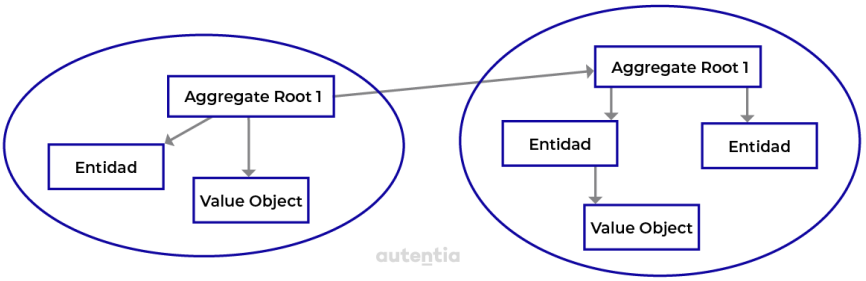
1. Servicios de dominio: su objetivo es orquestar funcionalidades que engloban diferentes entidades y *value objects*, y pueden servir de "fachada" a otros servicios de aplicación o de dominio, todo esto con el objetivo de tener nuestra lógica de negocio encapsulada.
2. Servicios de aplicación: estos servicios tienen una misión similar a los casos de uso, orquestando acciones entre diferentes servicios de negocio y de infraestructura, incorporando gestiones de seguridad y transaccionalidad.
3. Los servicios de infraestructura: estos servicios proveen acceso a recursos externos principalmente a los servicios de aplicación.

Aggregates

Un agregado en DDD (*aggregate*) es una agrupación de *entidades* y *value objects* que conceptualmente pertenecen a un conjunto. Por ejemplo, un pedido y la lista de artículos del pedido son entidades diferentes, pero para construir el modelo de dominio es más útil tratarlos como un conjunto: *un agregado*.



Cada agregado tiene una entidad raíz (*aggregate root*) de la que dependen el resto de *entidades* y *value objects*. Cualquier modificación de un componente interno del agregado debe realizarse a partir de métodos accesibles en dicha entidad raíz manteniendo la integridad y estado del conjunto en memoria. Cualquier referencia de fuera del agregado solo debe ir a la raíz del agregado.



Los agregados pueden relacionarse entre sí únicamente mediante los identificadores de su *entidad raíz*.

Los agregados son el elemento básico de la transferencia de datos: se cargan o se guardan los agregados completos. Las transacciones no deben cruzar las fronteras de los agregados. Los clientes que consumen el agregado no deben conocer los detalles de implementación, ni navegar entre referencias internas. Los atributos, propiedades, elementos y comportamientos internos del agregado no deben ser accesibles por el cliente.

No se recomienda diseñar agregados grandes. Hay que dividirlos, si es posible, para crear unos más pequeños, permitiéndonos así ser más escalables trabajando con transacciones y lógicas más livianas y aisladas. Para interactuar entre diferentes agregados podemos usar los eventos de dominio en sistemas distribuidos.

No siempre es posible separar agregados debido a posibles reglas de negocio que deban ser inmediatas y atómicas (en una misma transacción). Por eso es importante conservar *la consistencia transaccional* dentro de un único agregado (mediante las reglas que se llaman *las invariantes*). *La consistencia transaccional* supone que solo se ejecutan aquellas operaciones que no van a romper la reglas de negocio aplicadas a este agregado. Tenemos que discutir con los expertos de dominio cada caso. Teniendo en cuenta siempre la regla de *generar una única transacción por instancia de agregado.*

Los agregados DDD a veces se confunden con las clases de colección (listas, mapas, etc.). Los agregados de DDD son conceptos de dominio (orden, pedido, lista de reproducción), mientras que las colecciones son genéricas. Un agregado contendrá varias colecciones, junto con campos simples. El término "agregado" es común y se usa en varios contextos diferentes (por ejemplo, UML), en cuyo caso no se refiere al mismo

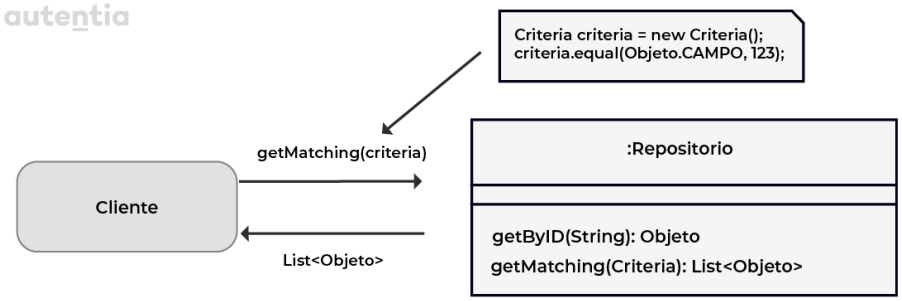
concepto que un agregado DDD.

Repositorios

Podemos encontrar tres formas de conseguir objetos en su ciclo de vida: crear uno nuevo, conseguirlo mediante la asociación con otro objeto que ya tenemos o construirlo obteniendo sus atributos de una base de datos. Para el último, existe el patrón **repositorio**, que se encarga de encapsular los accesos a la base de datos (o al sistema de gestión de información pertinente) para obtener los valores necesarios. Este patrón hace que el cliente del repositorio no tenga que centrarse en cómo recuperar la información ni en la tecnología utilizada para ello, haciendo que no pierda el foco en el dominio.

Un repositorio debe, como mínimo, ofrecer métodos para añadir, recuperar y eliminar objetos de un tipo. Para esto, el repositorio puede utilizar consultas a las bases de datos que inserten y borren datos de la misma. Además, también pueden ofrecer métodos para que el cliente recupere datos en base a criterios específicos, como el identificador, un valor de un atributo del objeto o un rango de valores, y que devuelva un objeto concreto o una colección de objetos que cumplan con el criterio. Aparte de éstas, se podrían añadir otras funciones que devuelvan cálculos hechos sobre la base de datos, como sumas de atributos o el número de objetos que cumplen con un criterio.

Por otro lado, también pueden ofrecer métodos a los que se les pasa una especificación de los criterios por los que se quieren obtener los objetos. Esto hace que la responsabilidad de definir los criterios de consulta sea del cliente, pero sin que éste tenga que tener en cuenta la tecnología que se utiliza para recuperar los datos. Debe ser el repositorio el que se encargue de transformar los criterios del cliente en consultas a la base de datos.



Estos repositorios deben ser interfaces, delegando así la implementación y permitiendo que se pueda sustituir en el cliente por mocks para hacer tests unitarios sin depender de la implementación del repositorio.

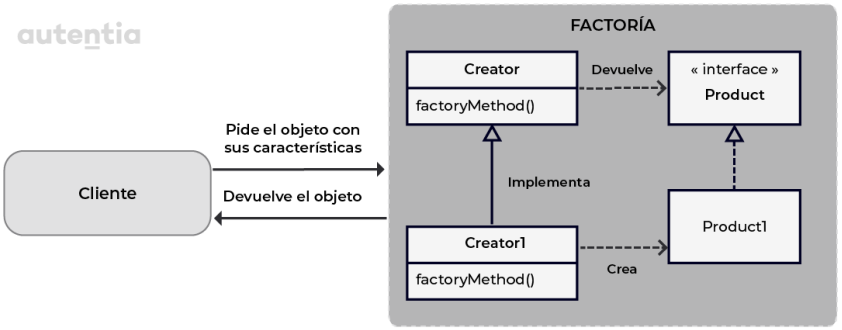
Factorías

Cuando la creación de un objeto o un *agregado* se vuelve excesivamente compleja, el patrón *factoría (factory)* puede ayudar a simplificar el proceso y ocultar la estructura interna de objetos creados.

Un cliente que se encarga de la creación de objetos se vuelve innecesariamente complicado y asume la responsabilidad que no es suya, violando la encapsulación de los objetos de dominio y agregados. Peor aún, si el cliente es parte de la *capa de aplicación*, entonces las responsabilidades se han filtrado fuera de la *capa de dominio*. Este estrecho acoplamiento elimina la mayoría de los beneficios de la abstracción de la capa de dominio.

La creación de objetos complejos es responsabilidad de la capa de dominio que no pertenece a los objetos que expresan el modelo. No tiene significado en el dominio, pero es una necesidad real y vital. Para resolver este problema, tenemos que agregar nuevos objetos al diseño del dominio que no son *entidades*, *value objects* o *servicios*. Es importante tener en cuenta que estamos agregando nuevos elementos que no se corresponden

con nada en el modelo, pero que, sin embargo, son parte de la responsabilidad de la capa de dominio.



Cada lenguaje orientado a objetos proporciona un mecanismo para crear objetos (*constructores,* etc.), pero existe la necesidad de mecanismos de construcción más abstractos que estén desacoplados de los otros objetos. Un elemento del programa cuya responsabilidad es la creación de otros objetos se llama *factoría*.

Los dos requisitos básicos para cualquier buena factoría son:

- Cada método de creación de factoría crea solo un objeto consistente y respetando todas las condiciones (*invariantes*) de clase de este objeto. Si la interfaz permite solicitar un objeto que no se puede crear correctamente, entonces se debe generar una excepción o se debe invocar algún otro mecanismo que asegure que no sea posible un valor de retorno incorrecto.
- La factoría debe abstraerse del tipo de objeto deseado.

Así como la interfaz de un objeto debe encapsular su implementación, permitiendo que un cliente use su comportamiento sin saber cómo funciona, una *factoría* encapsula el conocimiento necesario para crear un objeto complejo o *agregado*. Proporciona una interfaz que refleja los objetivos del cliente y una vista abstracta del objeto creado. Por lo tanto, traslada la responsabilidad de crear instancias de objetos complejos y

agregados a un objeto separado, que en sí mismo puede no tener responsabilidad en el modelo de dominio, pero sigue siendo parte del diseño del dominio. Proporciona una interfaz que encapsula todo el proceso complejo y que no requiere que el cliente haga referencia a las clases concretas de los objetos que se instalan.

Diseño estratégico

Introducción

Una de las mayores dificultades que encontramos en los sistemas es su tamaño y por tanto, su complejidad. Muchas veces estas complicaciones vienen dadas por no saber reconocer la *modularidad* o *individualidad* de las partes en las que se divide para poder escalar en el desarrollo.

El **diseño estratégico** propone para ello, una serie de guías y diseño de decisiones a tener en cuenta para poder hacer un correcto entendimiento y análisis posterior que reduzca la dependencia de las distintas partes del sistema pero sin sacrificar la sinergia de estar interconectadas. Se proponen tres principios:

- **Bounded Context**: se basa en que un sistema tiene que ser correctamente definido en partes bien diferenciadas sin contradicciones ni solapamientos y tiene que haber consistencia entre ellas.
- **Distillation**: saber centrarse correctamente en un área determinada del modelo para poder definirla correctamente dentro de una visión global y poder darle mayor o menor importancia dependiendo del valor que añada al sistema conjunto así como su propia *perspectiva*.
- **Large-Scale Structure**: propone un diseño basado en una amplia estructura que permita aplicar tantos patrones y diseños como sean

necesarios a nuestro sistema. Para ello, se propone la creación de un conjunto de capas o *layers* que se puedan relacionar entre sí y que cada una contengan responsabilidades diferentes en el conjunto global. Con una correcta definición de estas capas se podrá tener un sistema escalable uniformemente.

Bounded context

Uno de los principales términos de DDD es *el bounded context (*contexto limitado*)*. Es una área donde *el dominio* está totalmente definido utilizando su propio lenguaje ubicuo (*ubiquitous language*). ¿Qué queremos decir con "*el dominio*"? El dominio es el problema de negocio que queremos analizar para dar una solución con el software. El dominio puede dividirse en dominios más pequeños (subdominios). Es un concepto divisible. Además, el DDD nos indica diferentes tipos de dominios (*core domain, supporting subdomain, generic domain*), clasificándolos en base a su relevancia para el negocio.

Cualquier negocio tiene sus reglas y sistemas existentes que deben integrarse como parte de la solución. Muchas empresas tienen términos específicos que poseen su propio sentido dentro del contexto de su organización. Además, cada departamento de empresa puede tener sus propios objetivos y métricas. Entonces, ¿cómo podemos definir nuestro dominio? La respuesta es usando el lenguaje ubicuo, que tiene sentido solo dentro del contexto de cada dominio (o *subdominio*).

Por ejemplo, en un negocio donde existen dos departamentos: uno de ventas y otro de soporte, el término *Cliente* significa diferentes cosas. Para el departamento de ventas *un Cliente* hace compras, pedidos, pagos con sus tarjetas, etc. Y para el departamento de soporte, *un Cliente* es el que tiene problemas, preguntas, tickets relacionados con los productos que vende el negocio, etc. Cada departamento tiene su propia visión del término

Cliente y le interesa la información sobre un *Cliente* que está relacionada con su contexto/visión. Al departamento de ventas le interesa saber el historial de compras, su frecuencia, los medios de pago, etc. El departamento de soporte tiene interés en conocer los problemas que ha tenido el cliente, sus contactos, etc. Son dos contextos distintos (*bounded context*), aunque pueden tener la misma información (los contactos de cliente, los datos personales de cliente...).

Por eso es importante emplear un lenguaje distinto (el lenguaje ubicuo) en cada contexto, para poder distinguir los términos de cada contexto. Este lenguaje tiene que ser común y claro para todo el equipo (los desarrolladores, los expertos de dominio, los usuarios finales, etc.). Tanto los desarrolladores como los expertos tienen que usarlo (lenguaje ubicuo*)* dentro del código fuente o en el modelo del dominio. Este lenguaje puede evolucionar con el paso del tiempo, no se define en una sola reunión. Se elabora con la participación de todo el equipo intentando encontrar la mejor manera que describe el dominio/subdominio en este contexto concreto (*bounding context*).

Normalmente, cada *bounding context* representa solo un subdominio*,* separando así diferentes subdominios en diferentes contextos. Pero en el caso de los sistemas heredados o de los sistemas de terceros, es posible que *un bounding context* represente varios dominios. También puede pasar que desde el principio no hayamos separado nuestro dominio de forma correcta, mezclando diferentes subdominios en *un bounding context*. De todas maneras, la mejor manera de crear diferentes *bounding context* es extraerlos basándonos en sus responsabilidades y su comportamiento y separándolos creando un *bounding context* por un subdominio.

Subdomains

Un subdominio es el espacio que delimita un problema, cuya

implementación depende de un *bounded context*. Los distintos subdominios que conforman el dominio están definidos por el lenguaje ubicuo y los *bounded context* ayudan a delimitar la lógica a implementar.

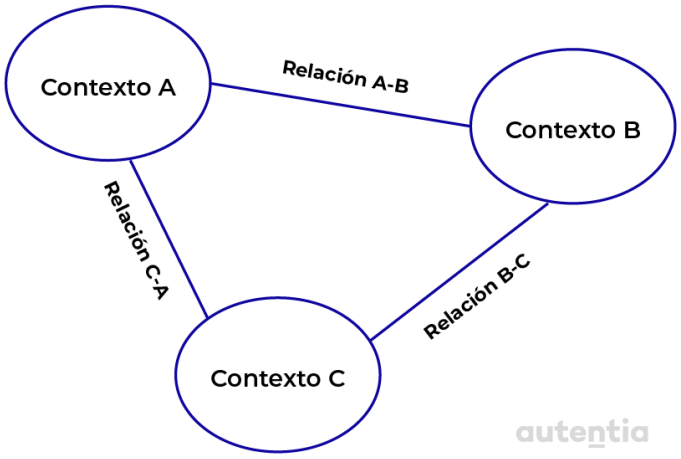
Llegados a este punto, podemos entrar en cierta confusión. ¿Los *bounded context* y los subdominios son lo mismo? La respuesta es sí y no. Cierto es que ambos conceptos atacan a lo mismo, pero tenemos que tener en cuenta que en el DDD existen 2 visiones a tener en cuenta: el espacio del problema y el espacio de la solución, donde los subdominios pertenecen al espacio del problema y los bounded context al espacio de la solución.

autentia

La relación entre ellos es similar a la relación entre el análisis o el qué y el diseño o el cómo: el espacio del problema presenta las razones de por qué se necesita nuestro software y qué pretende solucionar, mientras que el espacio de la solución explica cómo se pretende plantear dicha solución.

Introducción a las relaciones entre los Bounded Context

Cada área del modelo o **Bounded Context** está dispuesto de tecnología, lenguaje de programación y arquitectura mínima para poder funcionar de manera cohesionada dentro del modelo de dominio. El conjunto de las relaciones entre éstos es denominado **Context Map**.



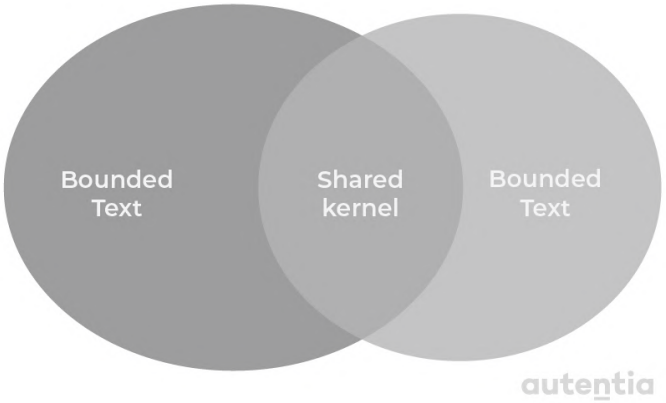
Para que los diferentes Bounded Contexts estén correctamente conectados entre sí y así hacer que el Context Map quede lo menos ambiguo y más transparente posible, se definen una serie de **patrones** con el objetivo de organizar el trabajo de desarrollo y proporcionar un vocabulario para describir la organización existente a nivel global.

Una relación puede no ajustarse completamente a la definición de uno de los patrones. Esto es normal, y el objetivo es que se vaya modificando el diseño para que éste se acerque más al patrón elegido. Además, en algunas ocasiones, una vez identificadas estas relaciones, éstas pueden ser confusas o complejas de entender y para ello, también existen patrones que nos ayudarán a hacer una **reorganización** que nos ayude a tomar decisiones en determinadas circunstancias.

A continuación se van a describir los **patrones organizativos y de integración** más destacables que describen los tipos de relación entre *Bounded Context* en un *Context Map*: **Shared kernel**, **Capa de anticorrupción** y **Customer/Supplier**.

Shared kernel

Shared kernel son los *bounded context* que comparten o reutilizan una parte del dominio, existiendo una interdependencia o un área común entre ellos. No deberían modificarse sin consultas previas entre los equipos afectados.



Si dos equipos trabajan en dos subdominios estrechamente relacionados puede que sea más beneficioso para ellos compartir algún subconjunto del modelo de dominio. Pueden gastar más tiempo creando las capas de traducción y de anticorrupción (una capa adicional entre los diferentes subsistemas que traduce las solicitudes que un subsistema hace al otro subsistema), que si utilizaran una parte del dominio conjuntamente. Por supuesto, esto incluye, junto con este subconjunto del modelo, el subconjunto de código o del diseño de la base de datos asociado con esa parte del modelo. Este subconjunto compartido es especial y no debe cambiarse sin consultar con el otro equipo.

El núcleo compartido no se puede cambiar tan libremente como otras partes del diseño. Cuando se realizan los cambios, todos los tests de ambos equipos deben pasar. Por lo general, los equipos realizan cambios en copias separadas del *shared kernel*, integrándose con el otro equipo a intervalos. Independientemente de cuándo se programe la integración de

código, cuanto antes hablen ambos equipos sobre los cambios, mejor.

Bastante a menudo, *shared kernel es* una parte del *core domain (el principal problema del negocio)* o un conjunto de *generic subdomains*, pero puede ser cualquier parte del dominio que se necesita para ambos equipos. El objetivo es reducir la duplicidad al máximo, haciendo la integración entre dos subsistemas más sencilla.

Customer/Supplier

Habitualmente la funcionalidad del sistema está dividida de tal manera que un subsistema esencialmente provee los datos a otro, mientras que el segundo realiza análisis u otras funciones que no afectan mucho al primero. También es común en estos casos que los dos subsistemas sirvan a los usuarios muy diferentes, con diferentes tareas, donde existen diferentes modelos. Lo que significa que el código no se puede compartir.

Los dos subsistemas, el cliente (*customer*) y el proveedor (*supplier*), donde el componente descendente (*cliente*) usa la salida del componente ascendente (*proveedor*) y todas las dependencias van en una dirección, se separan naturalmente en dos *bounded context* diferentes y esta relación se llama *Customer/Supplier*. El proceso es bastante simple por tener que operar en una sola dirección. Esto es especialmente cierto cuando los dos componentes requieren habilidades diferentes o emplean un conjunto de herramientas diferentes para la implementación. Pero hay problemas que pueden surgir en esta situación, dependiendo de la relación de los dos equipos.

El desarrollo libre del equipo "*proveedor*" puede verse limitado o incluso impedido si el equipo "*cliente*" tiene poder de veto sobre los cambios o si los procedimientos para solicitar cambios son demasiado complicados. El equipo proveedor puede incluso, verse inhibido por la preocupación de romper el sistema de cliente. Mientras tanto, el equipo cliente puede estar indefenso ante los cambios de proveedor.

El cliente necesita cosas del proveedor, pero el proveedor no es responsable de los entregables del cliente. Se necesita un gran esfuerzo para anticipar lo que afectará al otro equipo. Para resolver todos estos problemas se necesitan formalizar las relaciones entre los dos equipos creando una API documentada, un calendario de cambios, planeando nuevas tareas y funcionalidades conjuntamente, etc. Los equipos cliente y proveedor tienen más probabilidades de trabajar bien si los dos equipos trabajan bajo la misma dirección o comparten objetivos.

Capa de Anticorrupción

Los sistemas nuevos casi siempre tienen que integrarse con sistemas heredados que tendrán sus propios modelos. Cuando no se puede usar los patrones como *Shared Kernel* o *Customer/Supplier* la interacción entre los sistemas puede ser complicada. Las capas de traducción pueden ser simples, cuando conectan *Bounded Context* bien diseñados, pero cuando otro sistema está peor diseñado, la capa de traducción puede adoptar un tono más defensivo y complejo.

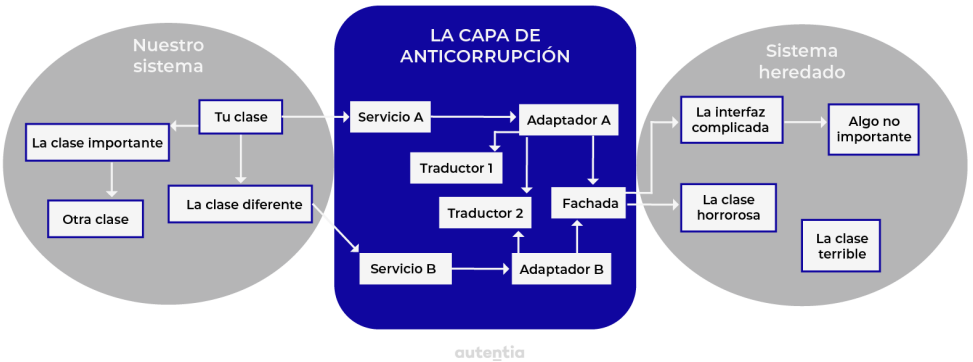
En el caso de interacción con un sistema heredado, a veces tiene sentido implementar una capa que aísla nuestro sistema del sistema antiguo. Esta capa puede traducir las solicitudes que un sistema hace al otro subsistema. Se llama *Capa de Anticorrupción*. Se usa para asegurarse de que el diseño de la aplicación no se vea limitado por dependencias de subsistemas externos. El patrón de la *Capa de Anticorrupción (Anticorruption Layer)* fue

descrito por primera vez en el libro "*Domain-Driven Design*" de Eric Evans (*Diseño basado en dominios*).

Es una capa de aislamiento para proporcionar a los clientes funcionalidad en términos de su propio modelo de dominio. Esta capa se comunica con el otro sistema a través de su interfaz existente, lo que requiere poca o ninguna modificación en el otro sistema. Internamente, la capa traduce las peticiones en ambas direcciones según sea necesario entre los dos modelos.

Una *Capa Anticorrupción* no es un mecanismo para enviar mensajes a otro sistema. Es un mecanismo que traduce objetos conceptuales y acciones de un modelo o protocolo a otro.

Una forma de organizar el diseño de la Capa de Anticorrupción es una combinación de fachadas, adaptadores y traductores, junto con los mecanismos de comunicación que normalmente se necesitan para interactuar entre sistemas.



Una fachada es una interfaz alternativa para un subsistema que simplifica el acceso para el cliente y hace que el subsistema sea más fácil de usar. Como sabemos exactamente qué funcionalidad del otro sistema queremos utilizar, podemos crear una fachada que facilite y agilice el acceso a esas funcionalidades y oculte el resto. La fachada no cambia el modelo del sistema subyacente. Debe crearse estrictamente de acuerdo con el modelo

del otro sistema. De lo contrario, añades la responsabilidad de la traducción a la fachada y creas otro modelo que no pertenece al otro sistema ni a tu propio *bounded context*. La fachada pertenece al *bounded context* del otro sistema. Simplemente, presenta un rostro más amigable especializado para tus necesidades.

Un *adaptador* es un contenedor que permite a un cliente utilizar la interfaz diferente que es entendida por otro sistema. Cuando un cliente envía un mensaje a un adaptador, se convierte en un mensaje semánticamente equivalente. La respuesta se convierte y se devuelve. El principal objetivo de un adaptador es hacer que un objeto envuelto se ajuste a una interfaz estándar que esperan los clientes. Para cada *servicio* que definimos, necesitamos *un adaptador* que soporte la interfaz del *servicio* y sepa cómo realizar solicitudes equivalentes del otro sistema o su *fachada*. El trabajo del adaptador es saber cómo realizar una solicitud.

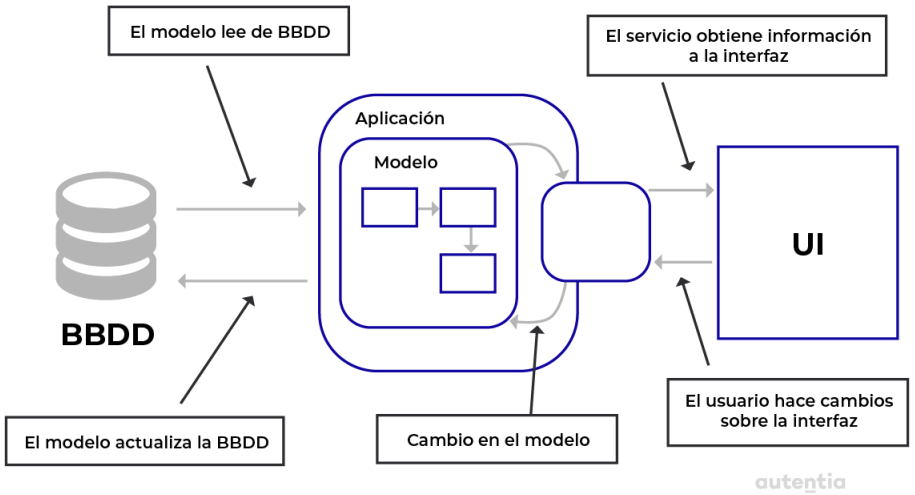
La conversión real de objetos o datos conceptuales es una tarea compleja distinta que se puede colocar en un objeto propio, *el traductor,* haciéndolos mucho más fáciles de entender. *Un traductor* puede ser un objeto ligero instanciado cuando sea necesario. No necesita ningún estado y no necesita ser distribuido, ya que pertenece al adaptador/adaptadores a los que sirve.

Al final *una Capa Anticorrupción* puede convertirse en una pieza compleja de software que requiere mucho esfuerzo para mantenerla. La clave es saber equilibrar los beneficios entre crear una Capa de Anticorrupción entre dos subsistemas y crear toda la funcionalidad de otro sistema de nuevo.

Command-Query Responsibility Segregation

CQRS es un patrón arquitectónico basado en la separación de las operaciones de lectura frente a las de escritura, refiriéndose a ellos como modelos Query y Command respectivamente. Conceptualmente, esto significa separar ambos modelos de datos y que funcionen de manera independiente con el objetivo de aumentar la simplicidad para escalar, leer y escribir y aumentar la seguridad de ambas partes por separado, sin embargo añade una complejidad adicional a nuestro sistema.

Partiendo de un esquema de modelo con base de datos e interfaz de usuario, un diseño comúnmente usado es el siguiente:



Este diseño tiene la particularidad de que pueden existir múltiples representaciones de la información: en el lado de la lectura se puede trabajar con DTOs con distintas formas y la asignación de objetos puede ser complicada; por otro lado, en el lado de la escritura, la lógica de negocio puede llegar a hacer un modelo excesivamente complicado. Además, los rendimientos de cada una de las partes suelen ser muy diferentes.

Por tanto, la solución que nos ofrece el patrón Command-Query Responsibility Segregation es separar los dos modelos de lectura y escritura, usando "**comandos**" para actualizar los datos y consultas o "**queries**" para leer los datos.

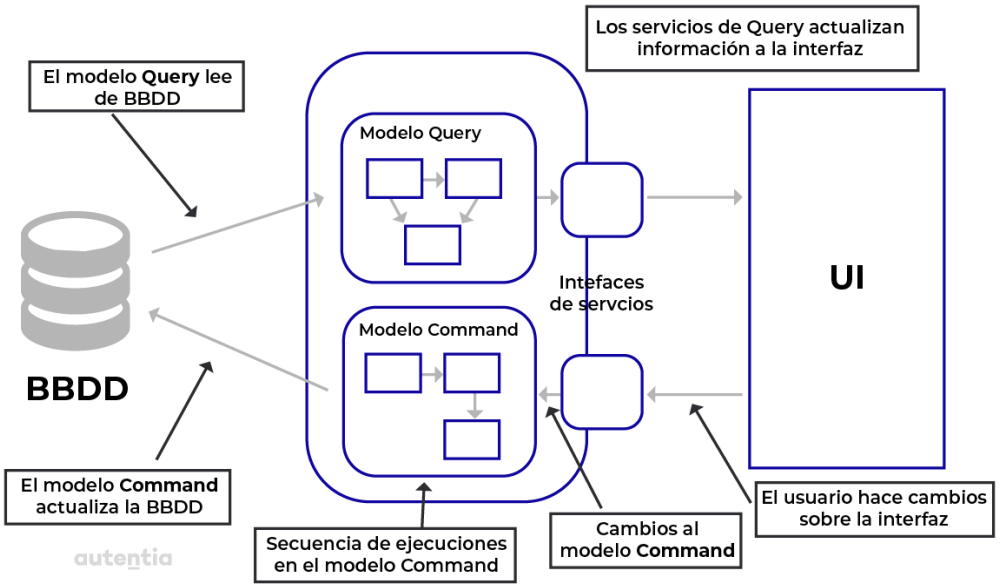
Command

Corresponde con el modelo de escritura sobre datos para persistir la información procedente de la capa de presentación. Debería de ser la única forma de alterar el estado del sistema y puede incluir reglas de validación sobre los datos.

Query

Corresponde con el modelo de lectura de datos devolviendo un resultado sin cambiar el estado del sistema y se encarga de transformar los datos para mostrarlos en la interfaz de usuario de la manera que sea requerida.

Podemos ver como quedaría esta división actualizando el diagrama anterior:



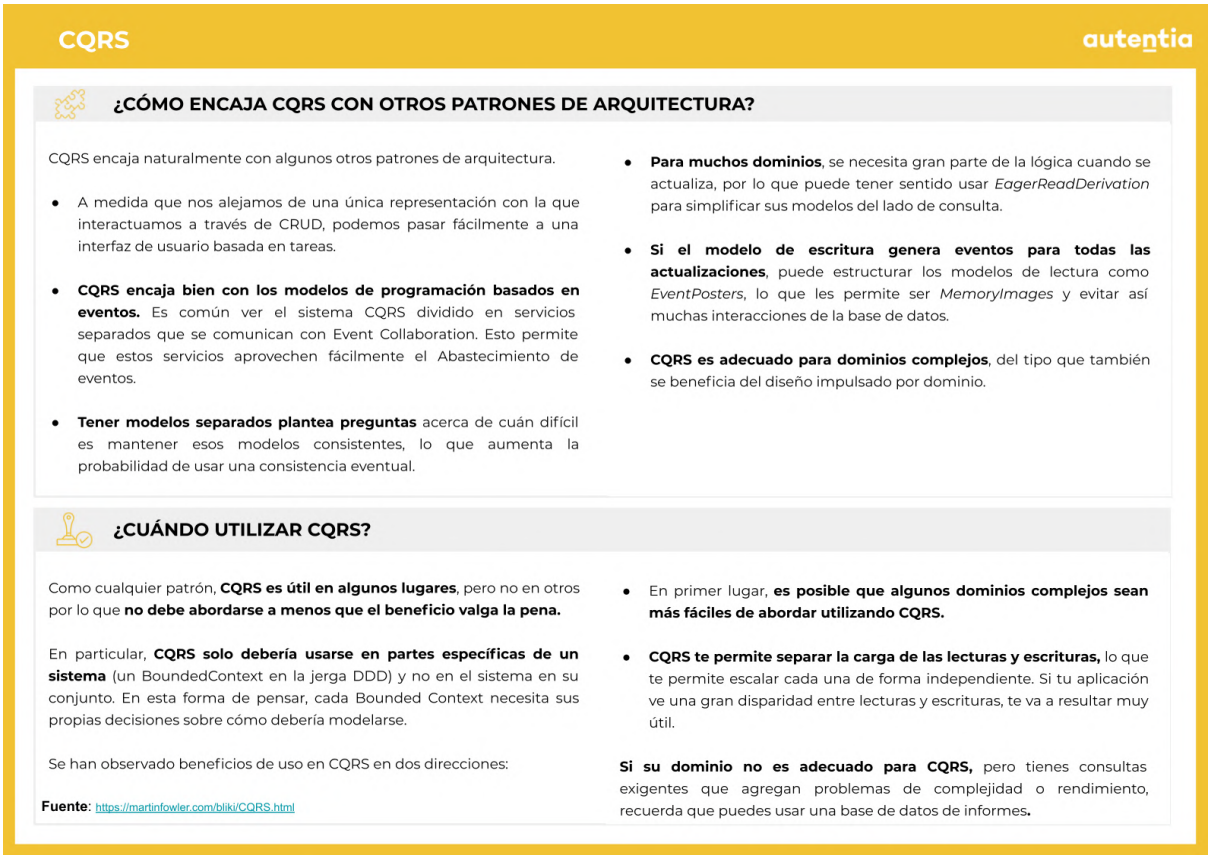
La principal ventaja que nos ofrece esta distinción o *segregación*, es que tenemos una independencia entre los dos modelos, pudiendo escalar cada uno como mejor nos convenga e incluso tener sus ejecuciones en diferentes máquinas. Al ser dos "entidades" diferenciadas, podemos aplicar por separado distintos niveles de seguridad y optimizar el tipo de trabajo ya sea consulta o persistencia en BBDD.

Pero también ofrece unas desventajas en la complejidad que indican que puede no ser una opción recomendable para determinados tipos de sistemas, pues requiere el doble de trabajo mantener ambas partes por separado y añadir nuevas funcionalidades dependiendo de nuevos requerimientos y si no se invierte en ello, el sistema entero se podría

quedar obsoleto continuamente.

El uso de este patrón puede ser beneficioso en casos como:

- Sistemas donde muchos usuarios acceden a los mismos datos y cada uno de ellos necesita realizar algún tipo de proceso con una serie de pasos.
- Situaciones donde hay una clara asimetría entre volumen de operaciones de lectura y escritura, permitiendo su escalado de manera independiente racionalizando los recursos.
- Escenarios en los que se quieren separar los procesos de lectura y de escritura permitiendo evolucionar de manera independiente la UI así como las reglas de negocio.



CQRS autentia

¿CÓMO ENCAJA CQRS CON OTROS PATRONES DE ARQUITECTURA?

CQRS encaja naturalmente con algunos otros patrones de arquitectura.

- A medida que nos alejamos de una única representación con la que interactuamos a través de CRUD, podemos pasar fácilmente a una interfaz de usuario basada en tareas.
- **CQRS encaja bien con los modelos de programación basados en eventos.** Es común ver el sistema CQRS dividido en servicios separados que se comunican con Event Collaboration. Esto permite que estos servicios aprovechen fácilmente el Abastecimiento de eventos.
- **Tener modelos separados plantea preguntas** acerca de cuán difícil es mantener esos modelos consistentes, lo que aumenta la probabilidad de usar una consistencia eventual.
- **Para muchos dominios**, se necesita gran parte de la lógica cuando se actualiza, por lo que puede tener sentido usar *EagerReadDerivation* para simplificar sus modelos del lado de consulta.
- **Si el modelo de escritura genera eventos para todas las actualizaciones**, puede estructurar los modelos de lectura como *EventPosters*, lo que les permite ser *MemoryImages* y evitar así muchas interacciones de la base de datos.
- **CQRS es adecuado para dominios complejos**, del tipo que también se beneficia del diseño impulsado por dominio.

¿CUÁNDO UTILIZAR CQRS?

Como cualquier patrón, **CQRS es útil en algunos lugares**, pero no en otros por lo que **no debe abordarse a menos que el beneficio valga la pena.**

En particular, **CQRS solo debería usarse en partes específicas de un sistema** (un BoundedContext en la jerga DDD) y no en el sistema en su conjunto. En esta forma de pensar, cada Bounded Context necesita sus propias decisiones sobre cómo debería modelarse.

Se han observado beneficios de uso en CQRS en dos direcciones:

Fuente: https://martinfowler.com/bliki/CQRS.html

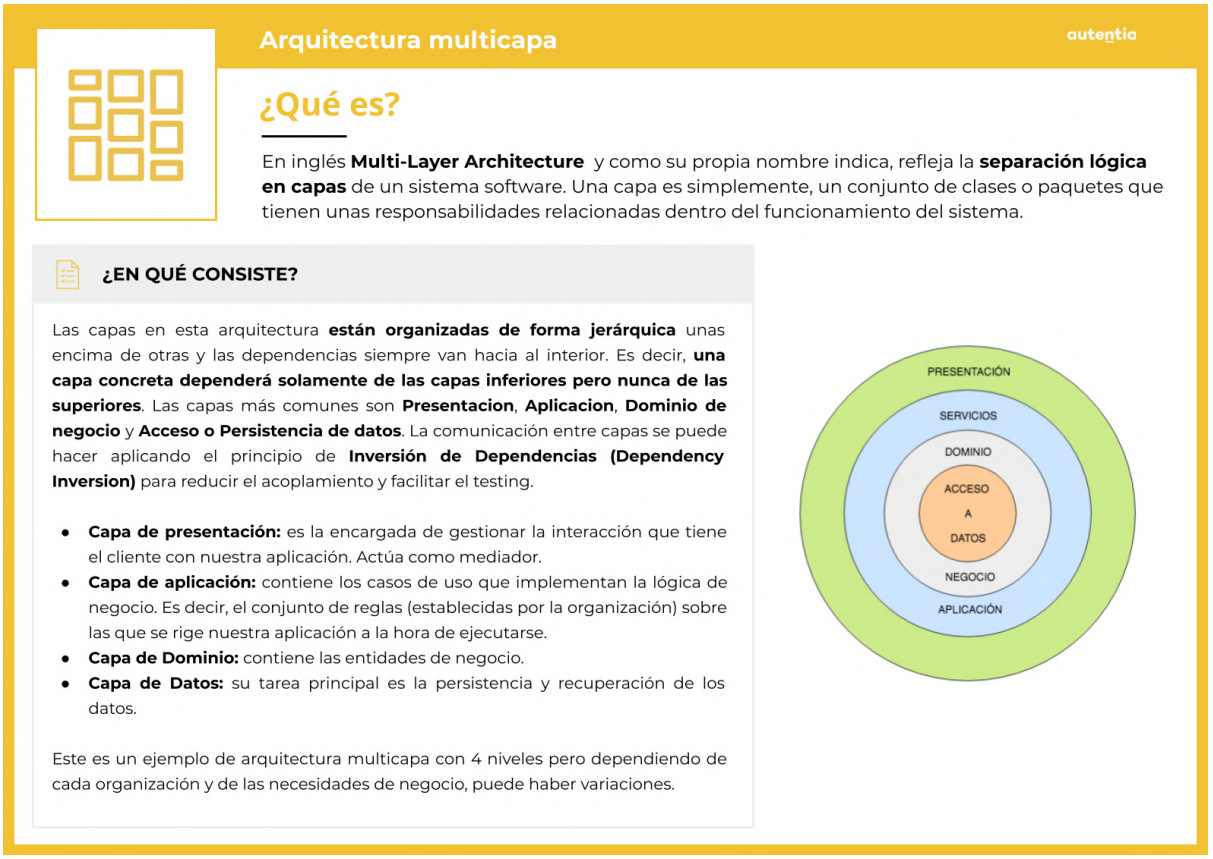
- En primer lugar, **es posible que algunos dominios complejos sean más fáciles de abordar utilizando CQRS.**
- **CQRS te permite separar la carga de las lecturas y escrituras,** lo que te permite escalar cada una de forma independiente. Si tu aplicación ve una gran disparidad entre lecturas y escrituras, te va a resultar muy útil.

Si su dominio no es adecuado para CQRS, pero tienes consultas exigentes que agregan problemas de complejidad o rendimiento, recuerda que puedes usar una base de datos de informes.

Clean Architecture

Las arquitecturas limpias o clean architectures organizan la aplicación en función de responsabilidades y cada una de estas se representa en forma de capa. De esta manera, se obtiene una arquitectura compuesta desde unas capas externas a otras más internas.

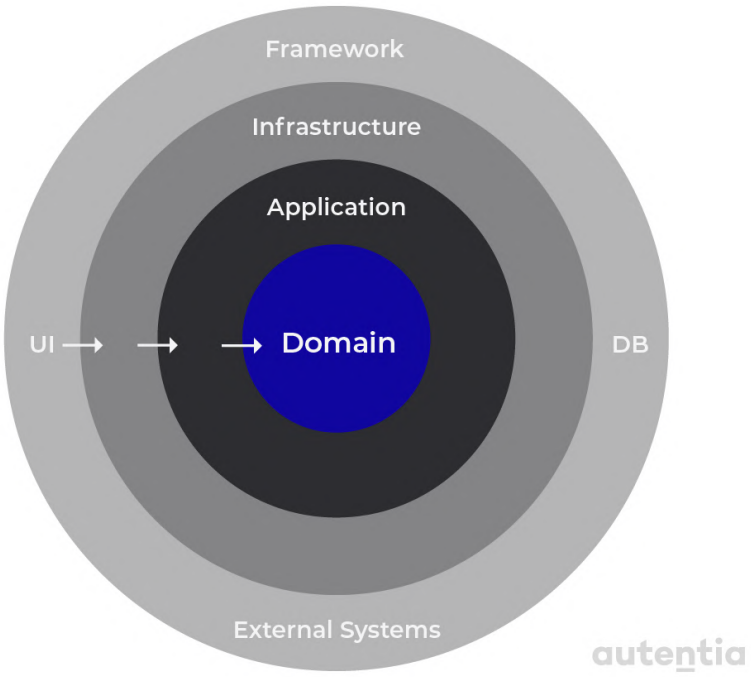
Una de las propiedades más importantes de este tipo de arquitectura es la regla de la dependencia, que dice que una capa interior debe estar totalmente aislada sobre una capa exterior. Por otro lado, las capas exteriores sí pueden conocer los detalles de las capas interiores.



Capas

Para diseñar una arquitectura por capas siguiendo los principios de DDD es necesario partir de la regla de la dependencia, es decir, que un elemento de una capa sólo tiene dependencias con otros elementos de la misma capa o con elementos de las capas más internas a esta. Esto permite que según va creciendo el proyecto sea más fácil añadir y adaptar nuevas piezas de código y se facilite el mantenimiento, ya que teniendo claramente diferenciadas las piezas software, se simplifica bastante el poder corregir futuros errores.

Pero, ¿cuáles son esas capas? Cuando se aplica DDD hay que tener diferenciadas tres capas claramente: la capa de dominio, la capa de aplicación y la capa de infraestructura.



Capa de dominio

La capa de dominio es el core o el corazón de la aplicación. Es la encargada de transformar las principales necesidades de negocio en código. Para ello, se hace uso de los términos de *entities*, *value objects* y *aggregates*, tal y como se han definido anteriormente. Dentro de esta capa podemos hacer una distinción entre **modelo** y **servicios** de dominio. El modelo son aquellos objetos que forman parte del core del dominio y representan unívocamente partes del negocio como pueden ser por ejemplo, las entidades. Cuando la lógica de negocio es compleja y no podemos representarla en el modelo, aparecen los servicios de dominio que representan aquellos servicios que contienen lógica propia del negocio para un dominio en concreto.

Capa de aplicación

La capa de aplicación tiene la responsabilidad de la creación y recuperación de los objetos de dominio para satisfacer las órdenes del usuario. No contiene lógica de negocio pero sí realiza labores de coordinación entre ellos acorde a las necesidades y casos de uso de la aplicación. Dicho en otras palabras, contiene los servicios que conectan la capa de dominio con el mundo exterior.

Capa de infraestructura

La capa de infraestructura provee capacidades técnicas a las capas superiores o internas (según se pinte) como por ejemplo, permitir que las entidades de dominio se persistan en las bases de datos. Debe estar completamente desacoplada de la capa de dominio para que si estamos usando pongamos, una base de datos relacional para la persistencia de los datos, si después queremos cambiar a otro tipo de motor de persistencia,

el cambio no impacte a las capas superiores.

Hexagonal architecture

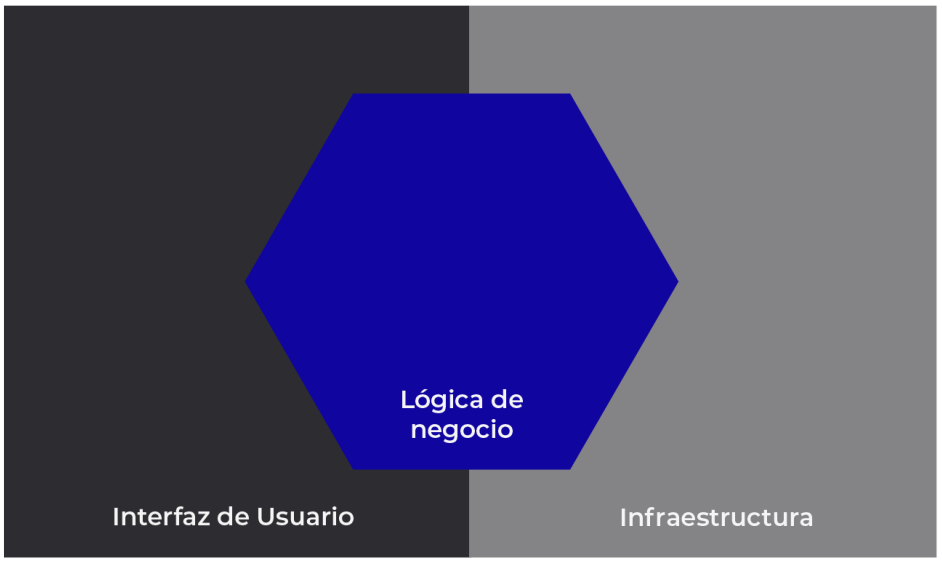
"Create your application to work without either a UI or a database so you can run automated regression-tests against the application, work when the database becomes unavailable, and link applications together without any user involvement."
(Alistair Cockburn)

Desde que se comenzó a hablar de arquitecturas limpias han ido apareciendo a lo largo de los años nuevos conceptos y propuestas que se rigen bajo sus principios, entre los que está *hexagonal architecture* de Alistair Cockburn u *onion architecture* de Jeffrey Palermo.

Bloques de un sistema

Podemos pensar en nuestra aplicación como un gran sistema el cual procuraremos dividir en tres capas diferenciadas como hemos visto anteriormente siguiendo la filosofía de clean architecture. Según el patrón de **arquitectura hexagonal** podemos asignar cada una de las capas a un *bloque de código* respectivamente:

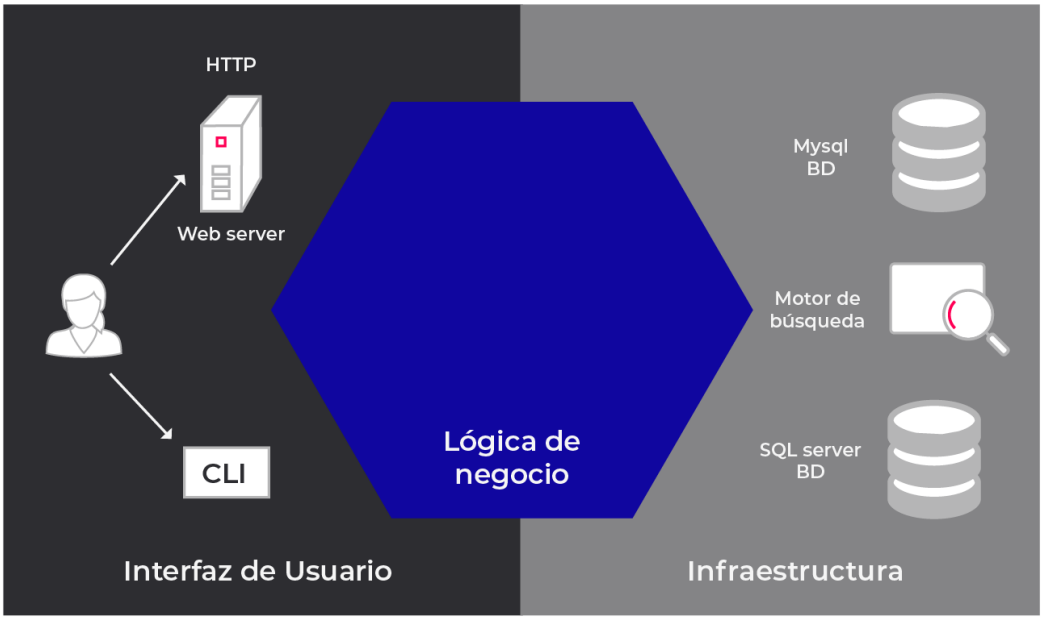
- **Interfaz de usuario**: todo el conjunto de librerías e implementaciones que hacen posible que exista una interfaz de usuario para interactuar con un sistema.
- **Lógica de negocio** o **Núcleo de la aplicación**: es el bloque con el que toma contacto la interfaz de usuario para proporcionarle una respuesta al usuario.
- **Infraestructura**: se encarga de conectar el núcleo de nuestra aplicación con herramientas como bases de datos, servidores, colas, etc.



autentia

Como bien sabemos, la lógica de negocio de nuestra aplicación es la parte más sensible e importante de nuestro sistema y por tanto, el resto de bloques tienen que interactuar con ella de forma adecuada y de manera agnóstica, esto es, la lógica de negocio no tiene que conocer nada acerca de la implementación tanto de la infraestructura como de la interfaz de usuario.

Ambas partes están compuestas de diferentes *herramientas* (motor de base de datos, servidor web, CLI...) y se encargan de interactuar con la lógica de negocio mediante *unidades de código* o **Adaptadores** de los que hablaremos más adelante, así como del **puerto** al que el adaptador se encarga de servir.



autentia

Por supuesto, los adaptadores de los dos bloques no pueden tener el mismo propósito. La diferencia principal es que los del bloque de la interfaz de usuario **dan información a nuestra aplicación** para ejecutar una tarea, mientras que el bloque de infraestructura está siempre **a la espera de recibir información** y procesarla de la manera correspondiente.

Puertos y adaptadores

Como comentábamos en la arquitectura hexagonal, nuestro sistema va a estar conformado por puertos y adaptadores donde todas las entradas y salidas de nuestra aplicación pasan por determinados puntos que aíslan la aplicación de herramientas externas tales como servidores web, motores de bases de datos, etc.

Originalmente, el modelo tradicional de modelado de un sistema acarreaba distintos problemas. En el lado de front-end, se incluían elementos de la lógica de negocio en la interfaz de usuario por ejemplo, (añadiendo una funcionalidad concreta en alguna vista, lo que hacía que no fuera reutilizable en algún otro lugar de la aplicación). En el back-end, el uso de librerías externas a veces influía sobre la lógica de negocio, haciéndose uso de ellas dentro de la propia implementación, creando así un fuerte acoplamiento.

Dominio

Algo muy importante en este patrón es saber delimitar el dominio del sistema, para saber aislarlo correctamente y así crear los puertos y adaptadores correspondientes. El dominio contiene todo lo concerniente a la lógica de negocio y debe de ser independiente y separada de detalles técnicos de implementaciones como frameworks y bases de datos.

Puerto

Un **puerto** es un punto de conexión de entradas y salidas a/desde la aplicación. Esto normalmente, está representado en el código con una *interfaz* y representa una especificación concreta de cómo la herramienta externa puede usar la lógica de negocio, o como ésta hace uso de dicha herramienta externa.

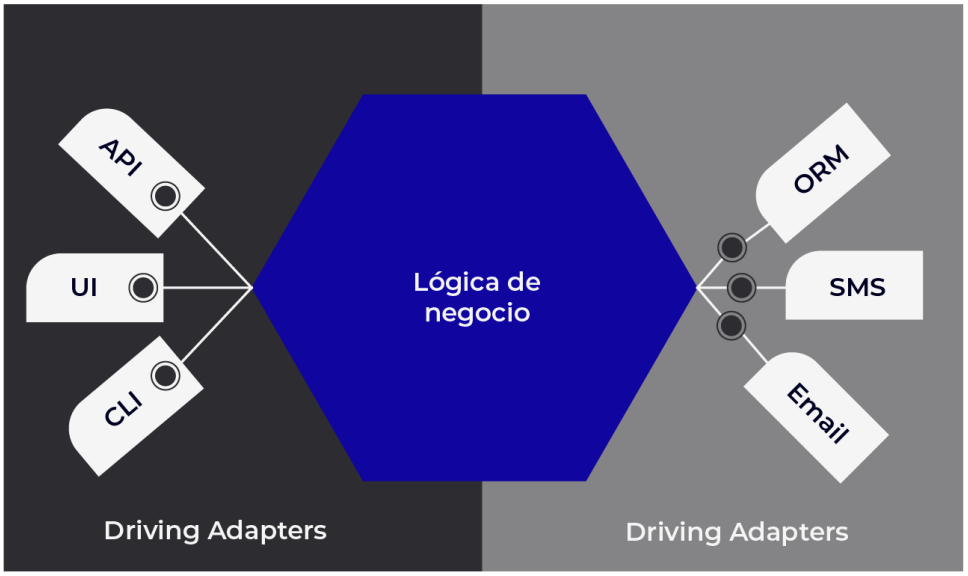
Adaptador

Un **adaptador** es una clase que transforma una interfaz en otra. En nuestro sistema se va a encargar de *adaptarse* a cada uno de los puertos.

Existen dos tipos de adaptadores: **Primarios** o **Driving Adapters** y **Secundarios** o **Driven Adapters**.

- **Driving Adapters**: son los adaptadores que comienzan algún tipo de acción en la aplicación.
- **Driven Adapters**: representan las conexiones con las herramientas de back-end y siempre reaccionan ante una entrada de los adaptadores primarios.

Esto se puede diferenciar además en dos lados distintos donde en cada uno se usan puertos y adaptadores.



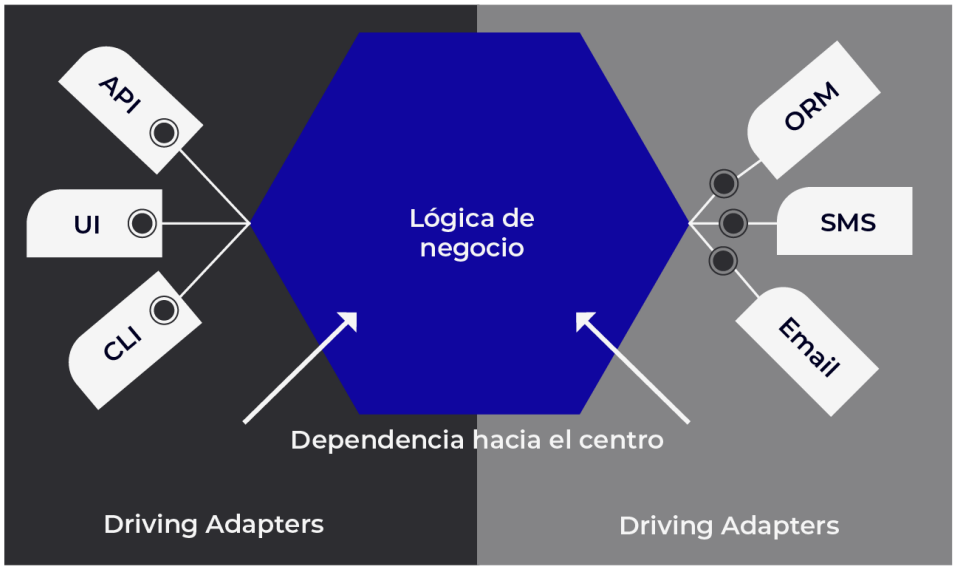
autentia

En el **lado izquierdo**, el adaptador depende del puerto y se le inyecta la implementación concreta del puerto (el adaptador envuelve al puerto). En este lado tanto el puerto como la implementación concreta del adaptador

están **dentro** de la aplicación (por eso en el diagrama anterior, aparece dentro de la representación de las aplicaciones). Los puertos pueden ser creados como **interfaces de servicio** o **interfaces de repositorio** que un controlador usa y la implementación concreta del Servicio o Repositorio es el **adaptador** que es inyectada y utilizada en el controlador.

En el **lado derecho**, el adaptador es la implementación específica del puerto y se inyecta en la lógica de negocio, conociéndolo ésta, únicamente como una interfaz. En este lado, el puerto pertenece a la aplicación pero su implementación se encuentra fuera y está contenida en alguna herramienta externa. Por ejemplo, si necesitamos persistir datos, creamos una interfaz para ello con métodos como *save* o *delete*, dicha interfaz implementará los métodos de un **adaptador** específico requerido como pueden ser las operaciones utilizando una base de datos MySQL. La ventaja de esto es que si queremos cambiar la implementación y hacer uso de otro motor de base de datos como PostgreSQL, únicamente tendremos que **crear su adaptador** con las implementaciones de las operaciones y **sustituir** la inyección del antiguo.

Como bien se puede comprobar en ambas partes, se hace continuamente uso de **Inversion of Control**, ya que la lógica de negocio depende de la **interfaz del puerto** y no de la **implementación del adaptador**; la dirección de las dependencias tendrían dirección hacia el centro o lógica de negocio desde ambos lados.



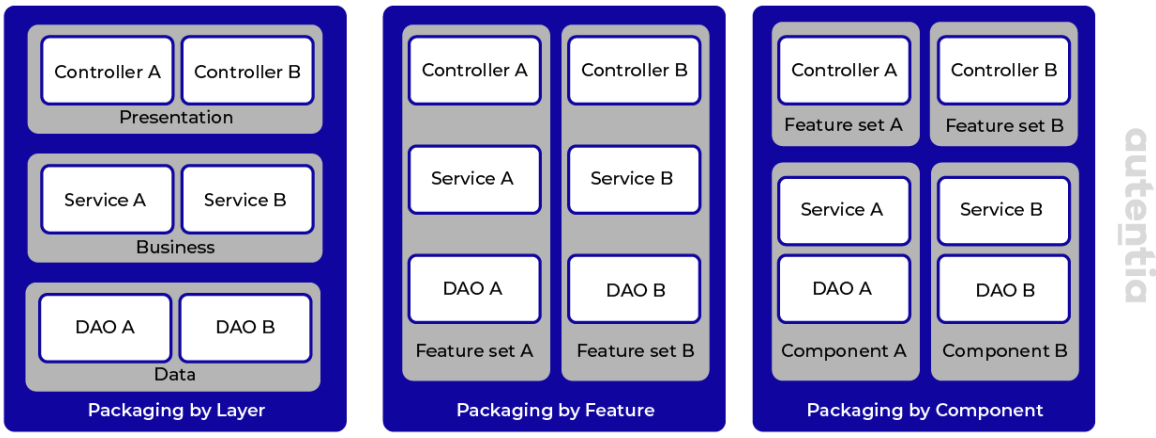
autentia

Para verlo con un ejemplo: tenemos una aplicación que utiliza MySQL para almacenar la información de usuarios. Inicialmente creamos una interfaz llamada *UserRepository* y por tanto, crearemos el adaptador de MySQL que implementará los métodos de esa interfaz: *UserRepositoryMySQLAdapter*. Si por ejemplo, se decide cambiar a otro sistema de persistencia como Cassandra, únicamente habría que crear otro adaptador que implemente también los métodos de *UserRepository* y lo podríamos llamar *UserRepositoryCassandraAdapter*.

Por otro lado, en la entrada de datos al sistema, creamos un **servicio de aplicación** llamado *UserProfileUpdate* donde habrá métodos de implementaciones de las herramientas externas como pueden ser GUI, CLI y API para nuestra aplicación. Cada una tendrá un controlador que usará *UserProfileUpdate* e inyectará la implementación específica del servicio para poder usarlo; por tanto, el **controlador es el adaptador**.

Componentes

Hasta ahora sólo hemos hablado de capas, pero ¿cuál es la manera más adecuada de organizar y agrupar nuestro código? Tal y cómo hemos visto en la arquitectura hexagonal, parece tener sentido agrupar nuestros ficheros en función de la capa a la que pertenecen. Sin embargo, también se ha hecho mucho hincapié, dentro de DDD, en la importancia de mantener separados los elementos en diferentes Bounded Contexts o subdominios. Entonces, ¿cuál es el criterio más adecuado para segregar el código? Podríamos decir que la manera más adecuada sería una combinación de ambas, **agrupadas en componentes.** Podríamos definir un componente como una agrupación lógica de elementos de software que dan solución a alguna necesidad de negocio. Ejemplo de componentes podrían ser Facturación, Reservas, Clientes... Así mismo, estos elementos se organizan o empaquetan internamente en elementos que pertenecen a las diferentes capas: presentación, lógica de negocio e infraestructura (estos nombres variarán en función de la terminología que estemos empleando).



Cuando un componente quiere hacer uso de una funcionalidad que pertenece a otro componente, pero queremos mantenerlos desacoplados ¿cómo debe ocurrir esa llamada entre componentes? Si realmente

queremos mantener los componentes independientes no podremos realizar una llamada directa a algún método del componente, sino que será necesario apoyarse sobre algún mecanismo que medie entre ambos como podría ser un ***Event Dispatcher*** que se encargue de enrutar eventos entre componentes o como mínimo hacer uso del API que exponga.

Flujo de control

En una aplicación, el control fluye desde el usuario hacia la aplicación que ejecuta cierta lógica. Es muy posible que el control se dirija hacia algún recurso de la infraestructura como un motor de persistencia de datos y después retorna desde la aplicación, de nuevo hacia el usuario. Sin embargo, ¿cómo se produce ese flujo? ¿Cómo se relacionan los componentes de cada capa que participan en un flujo? ¿Qué dependencias existen?

El flujo se iniciará mediante algún mecanismo que permita a los usuarios interactuar con nuestro sistema, como puede ser una aplicación Web o una interfaz de línea de comandos (CLI). El flujo llegará a algún controlador (adaptador primario) que necesitará o dependerá de algún puerto primario para invocar un caso de uso de la aplicación. La forma en la que se implementen realmente estos puertos primarios puede variar en función de si hemos decidido usar una capa de servicios de aplicación o estamos usando CQRS, y en este segundo caso, si estamos usando un Bus de comandos para desacoplar la solicitud de la ejecución del comando o no.

Usando servicios de aplicación

Si estamos usando servicios de aplicación, el controlador tendrá una dependencia directa con ellos y serán estos servicios los que contengan la lógica del caso de uso en cuestión. Para resolver el caso de uso interactúan

con elementos de la infraestructura mediante puertos secundarios y también con entidades de negocio (probablemente recuperadas a través de estos puertos mencionados), bien directamente o a través de servicios de negocio que permitan orquestar procesos que engloban a más entidades o que permitan reutilizar esa lógica en casos de uso diferentes. Una vez completado el proceso, es posible que el servicio necesite emitir algún evento indicando que el caso de uso se ha completado, usando alguna dependencia con algún elemento que haga esta labor de emisor de eventos. Será aquí también donde recaiga la gestión de la transaccionalidad de las operaciones. Finalmente, el servicio retornará el flujo al controlador recuperando la información en forma de VO si fuera necesario.

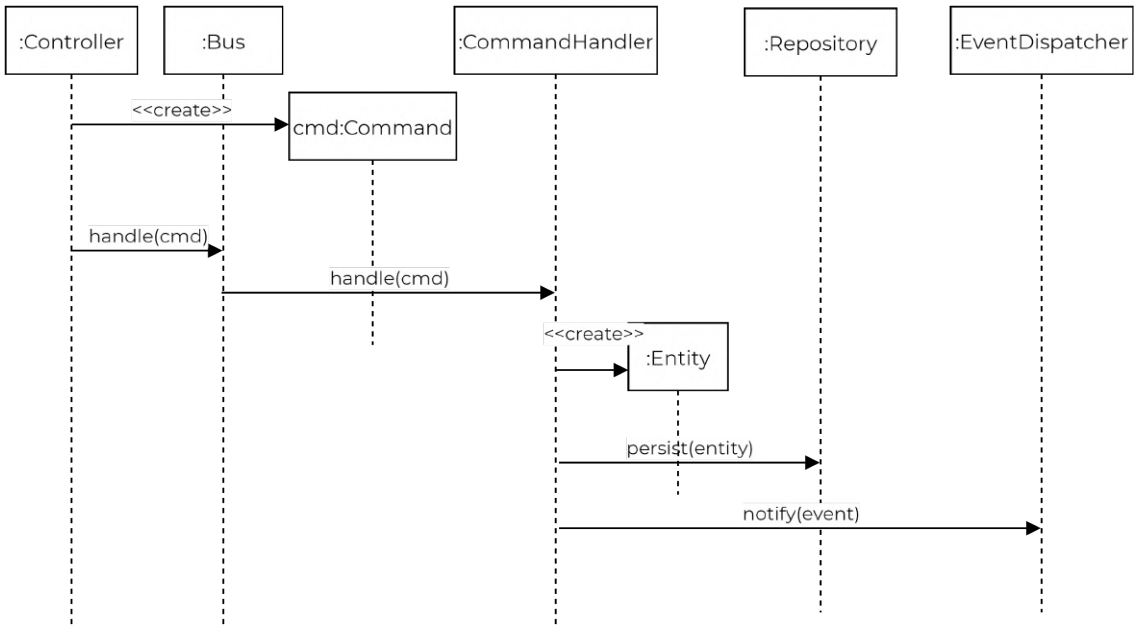
CQRS sin Bus

Si hemos decidido separar el modelo de recuperación de información del modelo de actualización mediante Commands y Queries para las operaciones de actualización, el controlador tendrá una dependencia con un Command. Éste será el responsable de llevar a cabo el caso de uso de una manera similar a lo descrito para los servicios de aplicación en el apartado anterior. Para las operaciones de lectura, el controlador usará un objeto de tipo Query. Éste contendrá la lógica de recuperación de la información necesaria mediante un puerto secundario, retornando dicha información al controlador en forma de VO. Éste se encargará de transmitir hacia la vista la información necesaria, bien directamente retornado el VO o bien adaptando la respuesta en un DTO.

CQRS con Bus

En este caso el controlador dependerá del Bus así como de un Command o Query. Sin embargo, no serán estos los que se encarguen de aplicar la lógica del caso de uso, sino meros transmisores de la información necesaria para llevarlo a cabo. Será el Bus el que se encargue de encontrar un

Command Handler adecuado para cada caso de uso. El *Command Handler* contendrá la lógica del mismo de la misma forma que en el apartado anterior, lo hacía directamente el Command.



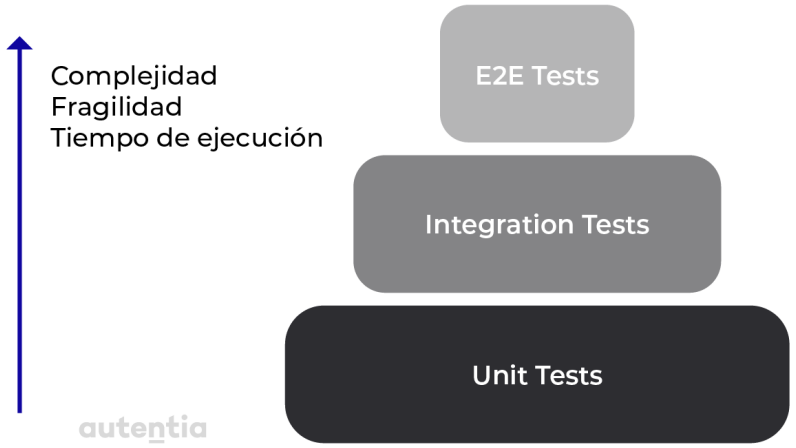
Testing

En un artículo denominado [“Screaming Architecture”], Robert C. Martin exponía la importancia de que las arquitecturas de nuestras aplicaciones deben reflejar de manera clara qué aplicación pretenden sustentar, al igual que el plano de un edificio debe representar qué tipo de edificio se pretende construir. Por eso, en este documento se ha hecho hincapié en que lo más importante de nuestras aplicaciones es la lógica de negocio y los casos de uso que soportan, siendo por tanto éstos la piedra angular sobre la que basar nuestra arquitectura y no los diferentes frameworks, herramientas y demás mecanismos de infraestructura que en última instancia, se utilizan para resolver los detalles técnicos.

Una buena solución arquitectónica debe permitir diferir las decisiones de infraestructura hasta el último momento posible, así como modificarlas sin

impactar en la aplicación. De la misma forma, una buena solución debe permitir testear cada caso de uso sin la necesidad de poner en juego para ello, ningún framework, herramienta, motor de persistencia o demás elementos que no sean los básicos del lenguaje de desarrollo en el que estemos implementando nuestra aplicación.

Basándose en estos criterios, a la hora de plantear cuál debe ser la naturaleza de nuestros tests y el volumen de cada tipo, encontramos que la mayor parte de nuestros tests serán unitarios. Éstos soportan el porcentaje mayoritario de caminos a probar dentro de la lógica y de los casos de uso de nuestra aplicación. Obviamente, nuestras aplicaciones no viven aisladas de la infraestructura y necesitan interactuar con motores de bases de datos, protocolos de comunicación diversos, etc. Necesitaremos por tanto, tests de integración que comprueben que las implementaciones en conjunto funcionan correctamente, pero ya sin la necesidad de incluir en estas pruebas los múltiples caminos relacionados con la variedad de los flujos de un caso de uso. Por último, necesitamos también incluir tests end-to-end de diversa índole: tests de aceptación, de UI, de API, etc. Estos tests son los más fiables pero son muy costosos y frágiles, por lo que hay que saber muy bien cuáles elegir y mantenerlos en un porcentaje razonable.

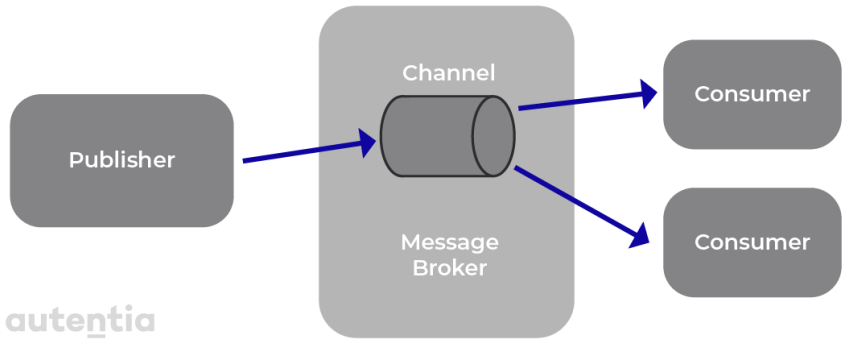


Event-driven architecture

Introducción

La *arquitectura orientada a eventos* (*event-driven architecture o EDA*) es un patrón que promueve el uso de eventos para la comunicación entre componentes independientes y es bastante común en aplicaciones creadas con microservicios. Un *evento* es un cambio asíncrono de estado o una actualización, como un artículo que se coloca en un carrito de la compra en un sitio web de comercio electrónico. Los eventos pueden incluir el estado (el artículo comprado, su precio y una dirección de entrega) o pueden ser identificadores (una notificación de que se envió un pedido).

Las arquitecturas impulsadas por eventos tienen tres componentes clave:



los ***publishers*** *de eventos*, los ***message brokers*** *(routers) de eventos* y los ***consumers*** *de eventos*. Un *publisher* publica un evento en el *message broker*, que filtra y envía los eventos a los consumidores que se hayan registrado para recibir ese tipo concreto de eventos. El servicio que publica los eventos y el servicio que los consume están desacoplados, lo que permite escalarlos, actualizarlos e implementarlos de forma independiente.

A través del ***channel (canal),*** los eventos son transferidos desde un

publicador a los consumidores que estén suscritos a dicho canal.

¿Cuáles son los beneficios de la arquitectura orientada a los eventos?

- **El *escalado y la gestión de errores de forma independiente.*** Al desacoplar los servicios, estos únicamente interactúan con *el message broker* y no se conocen entre ellos. Esto significa que los servicios son interoperables, pero si un servicio falla, el resto seguirá funcionando. El *message broker* actúa como un balanceador que se adapta a los aumentos de carga de trabajo.
- ***El desarrollo ágil***. Ya no es necesario escribir código personalizado para sondear, filtrar y enrutar eventos. *El message broker* se encarga de filtrar y enviar eventos automáticamente a los consumidores. El *message broker* también elimina la necesidad de una gran coordinación entre los servicios del *publisher* y del *consumidor*, lo que acelera su proceso de desarrollo.
- ***Inspeccionar con facilidad***. Un *message broker* actúa como una ubicación centralizada para guardar los logs de tu aplicación y definir políticas de acceso. Estas políticas pueden restringir quién puede publicar y suscribirse a un *message broker* y controlar qué usuarios y recursos tienen permiso para acceder a sus datos. También puede cifrar sus eventos tanto en tránsito como en reposo.
- ***Reducir costes.*** Las arquitecturas impulsadas por eventos están basadas en *push*, por lo que todo sucede bajo demanda cuando el evento se presenta en el *message broker*. De esta manera, no tienes que hacer un sondeo continuo para verificar si hay un evento. Esto significa menos consumo de ancho de banda de red, menos utilización de CPU y otros recursos.
- ***Implementación de patrones de control de flujo.*** Patrones como *backpressure* son más fáciles de implementar cuando tenemos un sistema que nos hace de buffer, en este caso el *message broker*. De

esta manera podemos ajustar la ingesta de eventos a la capacidad de procesamiento del servicio evitando saturarlo.

- ***Evitar penalizaciones por consumidores lentos.*** Si un evento es consumido por más de un servicio y los tiempos de respuesta son dispares, no penalizaremos el rendimiento del más rápido, ya que cada consumidor realiza el proceso de forma independiente.

Las desventajas de las arquitecturas orientadas a los eventos:

- ***Eventos duplicados***. Sin una planificación adecuada, un solo evento puede desencadenar múltiples mensajes duplicados en diferentes servicios, creando brechas en la comunicación.
- ***Complejidad de la nomenclatura.*** Al manejar numerosos publishers y suscriptores, es fácil tener nombres duplicados, especialmente en varios equipos que no se comunican con regularidad.
- ***No existe un orden de flujo de trabajo claro***. Cada paso a lo largo de un flujo de trabajo se activa cuando un servicio recibe una alerta sobre un evento en particular. Si el componente receptor no funciona correctamente, creará errores y provocará problemas de flujo de trabajo en cascada.
- ***Manejo de errores y resolución de problemas***. Una aplicación compleja puede incluir fácilmente cientos de agentes y servicios de mensajes, todos ellos tienen que pasar y recibir eventos constantemente. La arquitectura impulsada por eventos requiere un conjunto completo de herramientas y prácticas de monitoreo que visualicen el flujo de eventos.
- ***Consistencia eventual.*** Al ser procesos asíncronos no tenemos garantía de que en un momento dado el sistema tenga información consistente. Lo que sí se garantiza es que al finalizar el proceso de un determinado evento, el sistema será consistente. Esto provoca que haya que implementar patrones de gestión de la

transaccionalidad para garantizar la consistencia en caso de un error en uno de los pasos del flujo, añadiendo complejidad a la solución. Se trata más en profundidad en el punto de *Consistencia Eventual* de *Event Sourcing*.

- **Eventos en cascada.** Cuando en nuestra aplicación un evento desencadena otro, podemos llegar a la situación A -> B -> C, en donde no es trivial descubrir que el evento C es respuesta al tratamiento de A. Este tipo de arquitecturas conllevan el riesgo inherente de eventos en cascada y la dificultad añadida a la hora de trazar y auditar las acciones.

Evento de dominio

Un **evento de dominio** contiene la información sobre algo que ha sucedido en el dominio que quiere comunicar a otras partes del dominio. Otras partes del dominio pueden reaccionar o no a este evento.

El principal objetivo de un evento de dominio es captar la información que puede provocar cambios en el sistema. Los eventos se pueden guardar para luego utilizarlos como ***Audit log*** *(*el registro de todos los eventos de dominio).

Cada evento de dominio captura información que proviene de su origen. Si queremos usar los logs como auditoría, es importante que estos datos de origen sean **inmutables**. Es decir, una vez que hayas creado el objeto de evento, estos datos de origen no se pueden cambiar. Sin embargo, también hay otro tipo de datos sobre el evento que registra lo que un sistema ha hecho con él: *los* ***datos de procesamiento***.

Los datos de un evento de dominio son datos de origen inmutables que capturan de qué trata el evento y luego, los datos de procesamiento mutables que registran lo que el sistema hace en respuesta a ellos. Aunque

los datos de origen nunca pueden cambiarse, es posible que el sistema necesite hacer un cambio, normalmente porque el evento original fue incorrecto. Se puede corregir utilizando otro evento que se llama e*l* ***evento retroactivo***. Se utiliza para corregir las consecuencias del evento erróneo anterior.

Otra categoría de datos de eventos son los datos almacenados (*caché*) de otros eventos pasados. En este caso, el procesador de eventos resume la información de eventos pasados mientras procesa el evento actual y agrega esos datos resumidos al evento actual para acelerar el procesamiento futuro. Como cualquier caché, es importante señalar el hecho de que estos datos se pueden eliminar y volver a calcular en caso de que se produzcan ajustes.

Los diferentes eventos ocurren por diferentes razones, por lo que es común usar diferentes **tipos de eventos**. El procesador de eventos utilizará el tipo de evento como parte de su mecanismo de envío. El tipo de evento se puede representar mediante subtipos de eventos, un objeto de tipo de evento separado o una combinación de ambos.

Audit log

Un **registro de auditoría** (*audit log*) es la forma más simple, pero también una de las formas más eficaces de rastrear información temporal. La idea es que cada vez que ocurra algo significativo, tienes que crear algún registro que indique lo que sucedió y cuándo sucedió.

Un registro de auditoría puede adoptar muchas formas físicas. La forma más común es un archivo. Sin embargo, una tabla de base de datos también constituye un buen registro de auditoría. Si usas un archivo, necesitas un formato. Si se trata de una estructura simple, el texto delimitado por tabulaciones es simple y eficaz. Las estructuras más

complejas pueden manejarse con XML por ejemplo.

El *audit log* es fácil de crear pero más difícil de leer, especialmente a medida que crece. Se pueden realizar lecturas ocasionales con una simple vista y con herramientas de procesamiento de texto simples. Las tareas más complicadas o repetitivas se pueden automatizar con scripts. Muchos lenguajes de secuencias de comandos se adaptan bien para procesar los archivos de texto. Si usas una tabla de base de datos, puedes crear los scripts de SQL para obtener la información.

Cuando utilices el registro de auditoría, siempre debes considerar guardar tanto las fechas reales como las de registro. Son fáciles de crear y guardar y aunque pueden ser iguales casi siempre, habrá veces que pueden ayudarte a ahorrar mucho tiempo. Es importante recordar que la fecha de registro es siempre la fecha de procesamiento actual.

Evento retroactivo

Como hemos dicho antes, para corregir un evento erróneo se utilizan *los eventos retroactivos.* Existen tres tipos principales de situaciones con las que puedes encontrarte en un sistema orientado a los eventos:

- El **evento *fuera de orden*** es un evento que ha llegado suficientemente tarde para que ya hayas procesado los eventos que deberían haberse procesado después de que se recibió el evento fuera de orden.
- El **evento *rechazado*** es un evento falso y que no deberías haberlo procesado nunca.
- El **evento *incorrecto*** es un evento que contiene información errónea.

Cada una de las tres situaciones requiere que realices diferentes acciones con el flujo de eventos para corregirlas:

- Para eventos fuera de orden, cambiamos el orden de los eventos

retrocediendo al punto temporal donde tuviéramos que recibir el evento, insertamos el evento en cuestión y aplicamos los eventos posteriores.

- Para los eventos rechazados, revertimos el evento y lo marcamos como rechazado. Los eventos marcados como rechazados son ignorados por el procesador de eventos y solo permanecen en los registros de eventos para mantener el historial.
- Para eventos incorrectos, revertimos el evento y lo marcamos como rechazado. Además, insertamos el evento correcto y lo procesamos. Puedes pensar que es como una combinación de un evento rechazado y un evento fuera de orden.

Para poder utilizar los eventos retroactivos que te ayuden a corregir los eventos incorrectos se necesitan dos cosas principales: utilizar el ***Event Sourcing*** dentro de tu sistema y añadir la posibilidad de revertir los eventos de tu modelo o lo que se llaman *los* ***modelos paralelos***. Son dos requisitos bastante importantes que hacen que no todos los sistemas puedan aprovechar este mecanismo de corrección.

Punto temporal

El punto temporal representa **un punto en el tiempo con una precisión determinada**. Los puntos temporales se utilizan prácticamente en todos los sistemas o librerías. El principal problema que puede surgir con ellos es la precisión. No es lo mismo “el 22 de Abril” que “el 22 de Abril 22:33:23.00”. Si un evento pasa el día 22 de Abril nos puede interesar la hora exacta de procesamiento o recepción de dicho evento. Pero puede ser que no. Si las transacciones del sistema requieren precisión diaria, lo importante para nosotros es el día del evento. En este caso nos da igual “el 22 de Abril” o “el 22 de Abril 22:33.23.00”. Hay que intentar que las fechas y los puntos temporales tanto dentro de nuestro sistema como en los sistemas externos de los que dependemos sean **coherentes** y con la misma precisión. Es

bastante difícil comparar dos fechas con diferente precisión. Se requiere establecer las reglas adicionales para poder comparar los puntos temporales con diferentes tipos de precisión.

Ten cuidado con el uso de puntos de tiempo que sean **demasiado precisos** para tus necesidades. Muchas plataformas solo proporcionan un punto de tiempo con una precisión de segundo o mayor. Entonces, ¿cómo represento en cualquier momento el 22 de abril de 2021? Normalmente se utiliza una convención de medianoche ("00:00:00"). Eso puede funcionar en algunas circunstancias, pero los problemas pueden aparecer. Puede haber diferencias de segundos o milisegundos y dos puntos temporales aparentemente iguales pueden ser diferentes para nuestro sistema. Hay que intentar prevenirlo.

Otra cosa que tienes que tener en cuenta con los puntos temporales son las **zonas horarias**. Hay que estar muy seguro si realmente necesitas utilizarlas en tu sistema. Es perfectamente asumible usar los puntos temporales sin zona horaria. Un punto temporal sin una zona horaria es un tiempo local dentro del contexto del sistema. No siempre la información sobre la zona horaria es útil y puede llevar a la confusión a los usuarios de tu sistema.

Contexto

Una de las formas más comunes de pensar en una aplicación es imaginarla como un sistema que reacciona a eventos del mundo exterior. Podemos pensar que la interacción de un sistema con el mundo exterior es un flujo de eventos. Este enfoque es muy interesante sobre todo en el caso de la integración de múltiples sistemas. Mediante el uso de mensajes de eventos, puedes desacoplar fácilmente remitentes y receptores tanto en términos de identidad (transmitimos eventos sin importar quién responda a ellos) como de tiempo (los eventos se pueden poner en cola y reenviar cuando el

receptor está listo para procesarlos). Estas arquitecturas ofrecen una gran escalabilidad y modificabilidad debido a este acoplamiento flexible. Pero el enfoque orientado a los eventos también puede ser útil para diseñar la propia aplicación.

Una aplicación (Eventos internos)

Los eventos se pueden usar tanto para comunicación entre diferentes sistemas, como para comunicación interna entre diferentes partes del sistema. Normalmente **un evento indica algo que ha sucedido** - "un hecho". Se transmite la información sobre este "hecho" en alguna estructura de datos llevando los datos que describen el suceso, los puntos temporales (la hora en que ha ocurrido el evento, la hora de procesamiento del evento, etc.) También puede contener los datos de procesamiento que describen lo que ha hecho el sistema con el evento.

Si dividimos nuestra aplicación **en diferentes componentes (subsistemas)** podemos pensar que la forma más natural de la comunicación entre diferentes componentes es solicitud/respuesta. Un componente pide algo a otro componente mediante una petición REST o invocando un método de este componente. Otro estilo de colaboración es ***Event Collaboration*** *(*colaboración de eventos*)*. En este estilo, nunca un componente le pide a otro que haga nada, sino que cada componente emite un evento cuando algo cambia. Otros componentes escuchan ese evento y actúan si es necesario. El conocido patrón Observer es un ejemplo de colaboración de eventos.

Event Collaboration cambia las responsabilidades de objetos en torno al estado de almacenamiento. Con la colaboración de solicitud/respuesta, nos esforzamos por tener solo un componente que almacene una porción de datos, otros componentes luego le piden a ese componente los datos cuando los necesiten. Con la colaboración de eventos, cada componente almacena todos los datos que necesite y escucha los eventos de actualización para esos datos. Como resultado, tenemos un acoplamiento bastante débil que nos permite agregar nuevos componentes sin necesidad de modificar los componentes existentes. La desventaja de *Event Collaboration* es que es más difícil entender las reglas de la colaboración. Las colaboraciones de solicitud/respuesta se especifican en alguna forma en el código que muestra el flujo completo, **la colaboración de eventos es mucho más implícita**, lo que hace que sea mucho más difícil de depurar cuando sucede algo inesperado.

Múltiples aplicaciones (Integración de sistemas)

Podemos considerar que existen tres tipos de integración entre diferentes aplicaciones: *application integration*, *data integration* y *event integration*:

- ***Application integration*** se consigue a través de REST, SOAP, ESB, etc. Se utiliza para hacer que la funcionalidad en una aplicación se ejecute en respuesta a una solicitud de otra aplicación. Se usa bastante a menudo en las aplicaciones basadas en microservicios. Es menos usada para muchos casos de uso de integración de datos, ya que la mayoría de las integraciones de aplicaciones esperan de forma pasiva ser invocadas por un cliente, en lugar de enviar los datos de forma activa.
- ***Data integration*** consiste en transferir datos de un punto A a un punto B, incluido ETL, transferencia de archivos, procesos batch, etc. Es muy útil para crear informes de BI y otras tareas de movimiento de datos, pero menos adecuada que la integración de aplicaciones, para aplicaciones que comparten la funcionalidad.
- ***Event integration*** se podría considerar un híbrido entre los dos tipos de integración anteriores y en gran medida, obtiene los beneficios de ambos. Cuando una aplicación se suscribe a los eventos de otra aplicación, puede activar una parte del código en respuesta a esos eventos, lo que parece un poco como una API de la integración de aplicaciones. Los eventos que activan esta funcionalidad también llevan consigo una cantidad significativa de datos, lo que parece una integración de datos.

Event integration logra un equilibrio entre los dos modos de integración clásicos. La transformación de las integraciones de aplicaciones tradicionales en una integración de eventos abre más puertas para el análisis, el aprendizaje automático, BI o la sincronización de datos entre

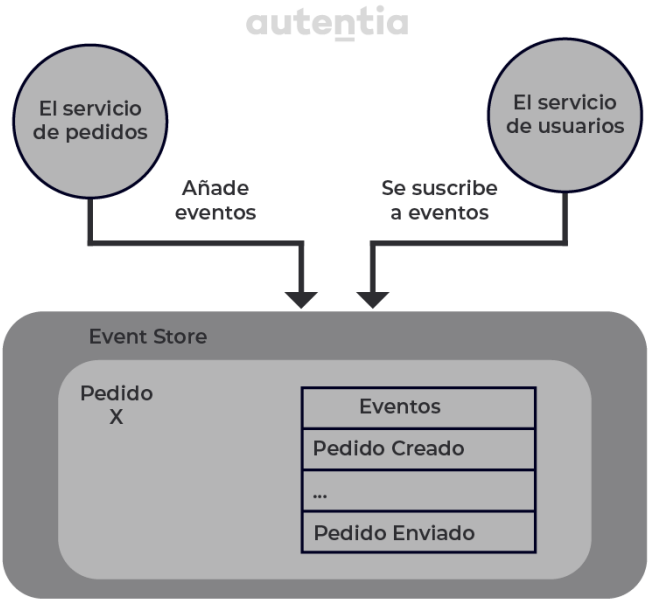
aplicaciones.

Event sourcing

Event sourcing nos provee la atomicidad mediante el **almacenamiento de los eventos de dominio**. En lugar de almacenar el estado actual de una entidad, la aplicación almacena una **secuencia ordenada de eventos** de cambio de estado. La aplicación reconstruye el estado actual de una entidad reproduciendo dicha secuencia. Siempre que cambia el estado de una entidad de negocio, se agrega un nuevo evento a la lista de eventos. Dado que guardar un evento es una sola operación, es siempre atómico.

Por ejemplo, en un sistema tradicional guardamos todos los pedidos dentro de una tabla de pedidos y cada fila de esta tabla representa un pedido. Pero usando *Event sourcing* solo guardaremos los eventos de cambio de estado de un pedido: pedido creado, pedido aprobado, pedido pagado,

pedido cancelado, pedido enviado, etc.



Los eventos persisten en un ***Event Store***, que es una base de datos de eventos. *Event Store* tiene una API para agregar y recuperar los eventos de una entidad. *Event Store* también se comporta como un *Message Broker*. Proporciona una API que permite que los servicios se suscriban a eventos. *Event Store* ofrece todos los eventos a todos los suscriptores interesados. *Event Store* es el principal componente de una arquitectura de microservicios basada en eventos.

Event Sourcing tiene varias ventajas y desventajas. Resuelve uno de los problemas clave en la implementación de una arquitectura impulsada por eventos, la ***coherencia de datos*** y hace posible publicar eventos de manera confiable siempre que cambie el estado. Además, debido a que persiste eventos en lugar de objetos de dominio, evita principalmente el problema de **desajuste** entre los **tipos** de datos relacionales (en BBDD) y los tipos de datos en el programa. *Event sourcing* también proporciona un **registro confiable** de los cambios realizados en una entidad de negocio y hace posible implementar **consultas temporales** que determinan el estado de

una entidad en cualquier momento.

Event Sourcing también tiene algunos inconvenientes. Es un estilo de programación diferente y poco conocido, por lo que hay una curva de aprendizaje pronunciada. *Event Store* solo admite directamente la búsqueda de entidades por clave principal. Se necesita crear vistas materializadas para poder proveer la información necesaria para los clientes.

Estado de la aplicación

La forma más sencilla de pensar en el uso de *Event Sourcing* es **calcular el estado** de una aplicación comenzando desde un estado de aplicación en blanco y luego aplicando los eventos para alcanzar el estado deseado. Es un proceso bastante lento, particularmente si hay muchos eventos.

En muchas aplicaciones, es más común solicitar estados de aplicación recientes, si es así, una alternativa más rápida es almacenar el estado actual de la aplicación y si alguien quiere las características especiales que ofrece *Event Sourcing,* esa capacidad adicional se construye aparte.

Los estados de la aplicación se pueden almacenar en la memoria o en el disco. Dado que el estado de una aplicación se deriva puramente del registro de eventos, puedes almacenarlo en cualquier lugar que desees. Un ejemplo sería inicializar el estado del sistema al comienzo del día a partir de una instantánea nocturna y mantener el estado actual de la aplicación en la memoria. Si el sistema falla, reproducimos el estado desde los eventos de la instantánea nocturna y aplicamos los eventos recibidos este día. Al final de la jornada se puede guardar el estado en una nueva **instantánea**. Se pueden realizar nuevas instantáneas en cualquier momento en paralelo sin tener que apagar la aplicación en ejecución.

El sistema de registro puede ser el registro de eventos o el estado actual de la aplicación. Si el estado actual de la aplicación se mantiene en una

base de datos, es posible que los registros de eventos solo estén allí para auditoría y análisis. Pero los registros de eventos pueden ser el registro principal y se pueden construir bases de datos a partir de ellos cuando sea necesario.

Transaction scripts y modelos de dominio

Se puede considerar que la lógica de la mayoría de las aplicaciones comerciales es una serie de transacciones. Una transacción puede pedir cierta información organizada de una manera particular, otra hará cambios en ella. Cada interacción entre un sistema - cliente y un sistema - servidor contiene una cierta cantidad de lógica. En algunos casos, esto puede ser tan simple como mostrar información recibida desde una base de datos. En otros, puede implicar muchos pasos de validaciones y cálculos.

Un *transaction script* organiza toda esta lógica principalmente como una sola función (procedimiento), realizando llamadas directamente a la base de datos o mediante una capa de wrapper (envoltura) de base de datos. Cada transacción tendrá su propio script de transacción, aunque las subtareas comunes se pueden dividir en subprocedimientos.

El patrón *Transaction Script* funciona bien para aplicaciones pequeñas que no implementan ninguna lógica compleja. Pero cuando se usa en aplicaciones más grandes, *Transaction Script* trae problemas como la duplicación de código entre diferentes procedimientos, mantenimiento complejo, etc. Cuando esto ocurre, debemos tender hacia el uso del patrón *Domain Model*. *Domain Model* crea una serie de objetos interconectados, donde cada objeto representa algo significativo, ya sea tan grande como una empresa o tan pequeño como una sola línea en un formulario.

En el proceso de desarrollo de nuestro sistema también puede surgir otra duda. **¿Dónde colocar la lógica para manejar los eventos?** Una buena forma de pensar en esto es crear dos lógicas diferentes que tengan dos

responsabilidades diferentes. La lógica del dominio de procesamiento es la lógica empresarial que manipula la aplicación. La lógica de selección de procesamiento es la lógica que elige qué parte de la lógica del dominio de procesamiento debe ejecutarse según el evento entrante. Puede combinarlas juntas, este es el enfoque de *Transaction Script*, pero también puede separarlas colocando la lógica de selección de procesamiento en el sistema de procesamiento de eventos, y llamar a un método en el *Domain Model* que contiene la lógica del dominio de procesamiento.

Si no es necesario revertir los eventos, entonces se puede permitir que *Domain Model* ignore el registro de eventos. Pero si es necesario, *Domain Model* necesita almacenar y recuperar el estado anterior, lo que hace que sea inevitable para *Domain Model* conocer el registro de eventos.

Modelos paralelos

Las aplicaciones que utiliza *Event Sourcing* nos permiten recrear cualquier momento de nuestra aplicación gracias al almacenamiento de todos los eventos de nuestro sistema. Así podemos crear *un* ***modelo paralelo***, que es una representación alternativa del estado del sistema en un momento dado tanto en el pasado, como en el futuro.

Event Sourcing captura y almacena cada evento. Así podemos pensar que el estado actual de una aplicación es el resultado de tomar un estado inicial y reproducir eventos correspondientes. Cada secuencia de eventos conduce a un estado diferente. El estado del viernes por la mañana es el resultado de aplicar todos los eventos que ocurrieron antes del viernes, el estado ahora es el resultado de reproducir todos los eventos que alguna vez ocurrieron.

Pensar en un estado como resultado de la aplicación de eventos, establece un enfoque diferente que te permite reproducir **realidades hipotéticas**.

Para construir *un modelo paralelo*, lo primero que debes verificar es si

tienes un sistema que utiliza *Event Sourcing*. Si el sistema se diseñó teniendo en cuenta *Event Sourcing*, eso es lo mejor porque nos facilitará su implementación. En caso contrario necesitaremos adaptarlo para soportar *Event Sourcing*. Lo siguiente que necesitarás es una forma de procesar eventos en un modelo que esté **separado** de la realidad. En general, hay dos formas de hacer esto. Uno es construir un sistema paralelo que es una copia separada de la aplicación con su propia base de datos y entorno. El otro es permitir que un solo sistema pueda cambiar entre modelos paralelos integrados.

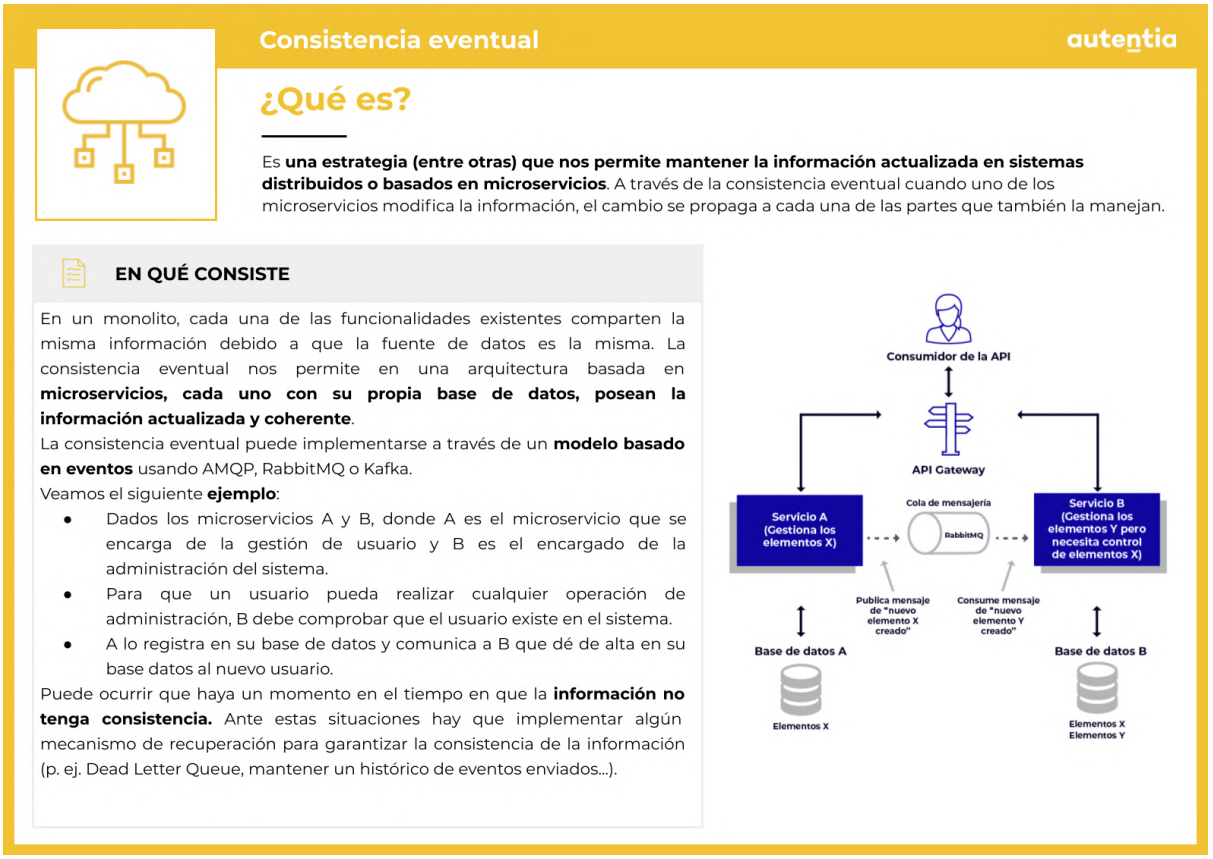
La gran ventaja de un **sistema paralelo** es que no es necesario hacer nada en la aplicación propia para construir un modelo paralelo. Hacemos una copia de la aplicación, cambiamos los recursos, como la base de datos y ya está. Pero es difícil comparar los resultados de diferentes modelos al ser unos sistemas diferentes.

Uno de los beneficios de utilizar un sistema paralelo es que también tiene la capacidad de cambiar el código fuente del sistema paralelo. Con un sistema paralelo, puedes realizar cualquier cambio que quieras en el código fuente de la aplicación y ver las consecuencias en el flujo de eventos.

Si quieres incrustar varios modelos paralelos en una aplicación, deberás asegurarte de que cualquier almacenamiento permanente se pueda cambiar fácilmente entre varias fuentes de datos. Una buena forma de hacerlo es utilizar un repositorio y proporcionar un mecanismo para que la aplicación pueda cambiar de repositorio en tiempo de ejecución. Los repositorios temporales no necesitan ser persistentes, dado que el modelo paralelo tiende a ser temporal, incluso se puede almacenar en la memoria. Otra ventaja de usar repositorios es que puede ser mucho más rápido ya que estás procesando todo en la memoria operativa. Una vez que hayas procesado sus eventos y hayas obtenido la información que necesitas del modelo paralelo, puedes descartar el repositorio.

Consistencia eventual

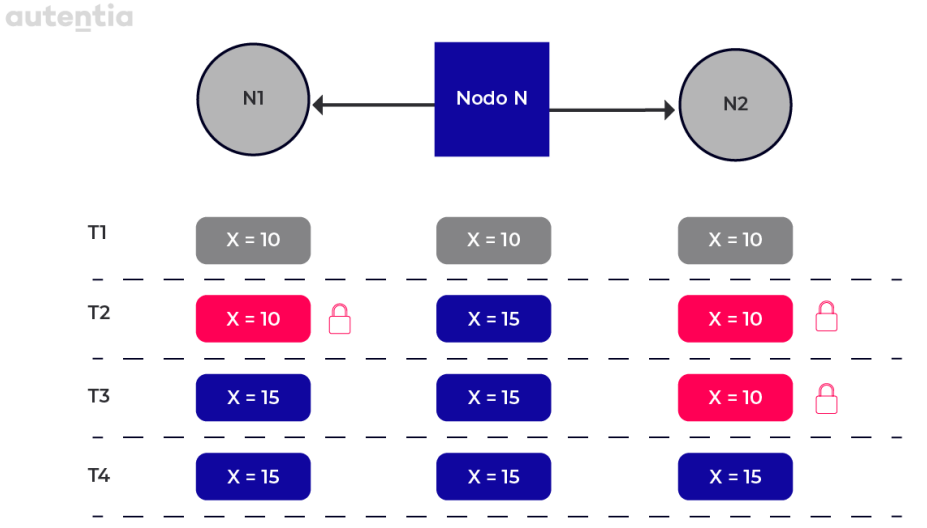
*La **consistencia eventual*** en los sistemas distribuidos consiste en que si no se realizan nuevas actualizaciones al objeto, todos los accesos a este objeto devolverán el último valor actualizado. La consistencia eventual también puede llamarse como *la consistencia optimista*. Es un modelo de *consistencia débil (weak consistency)* que no garantiza que los accesos posteriores al cambio devuelvan el valor actualizado. A diferencia de la *consistencia fuerte (strong consistency)* que sí garantiza que todos los accesos posteriores al cambio devuelven el valor actualizado. La gran diferencia entre un modelo y otro es la accesibilidad. Los sistemas con un modelo de consistencia fuerte tienen que **bloquear** los accesos a otros nodos del sistema hasta que se actualicen todos los nodos del sistema.



Los servicios con la consistencia eventual se clasifican como los servicios que proveen las garantías ***BASE*** (*Basically Available, Soft state, Eventual consistency*), en contraste con las garantías tradicionales ***ACID*** (*Atomicity, Consistency, Isolation y Durability*). Estas son las definiciones aproximadas de cada término de BASE:

- ***Basically Available***: las operaciones básicas de lectura y escritura están disponibles tanto como sea posible (utilizando todos los nodos de un clúster de base de datos), pero sin ningún tipo de garantía de coherencia.
- ***Soft state***: sin garantías de consistencia, después de un tiempo, solo tenemos cierta probabilidad de conocer el estado.
- ***Eventual consistency***: si el sistema está funcionando y esperamos lo suficiente, finalmente podremos saber cuál es el estado de la base de datos.

El periodo de tiempo entre la actualización del objeto y el momento en el que está garantizado que todos los accesos al objeto devuelven el valor actualizado se llama *la* ***ventana de inconsistencia (inconsistency window)***. Si no ocurren errores, el tamaño máximo de la ventana de inconsistencia se puede determinar en función de diferentes factores como las demoras en la comunicación, la carga en el sistema y la cantidad de réplicas involucradas en el esquema de replicación. El sistema más popular que implementa la consistencia eventual es DNS. Las actualizaciones de un nombre de dominio (host) se distribuyen de acuerdo con un patrón configurado y en combinación con cachés controlados por tiempo. Finalmente, todos los clientes verán la actualización.



En cualquier sistema distribuido (por ejemplo, uno de microservicios) habrá múltiples copias de cualquier dato. La cuestión es cómo diseñar la gestión de datos en este sistema. Por ejemplo, en un sistema bancario ¿cómo se gestiona el almacenamiento del saldo de cuenta corriente? Es posible que tenga un valor almacenado en caché. También habrá un valor en la base de datos en los sistemas de back-end del banco. Debemos tomar una decisión sobre cuánto esfuerzo haremos para que el saldo bancario sea consistente. Si decidimos que el número debe ser siempre el mismo en todas partes (por ejemplo, el saldo que se muestra en la página web del banco y el saldo guardado en la base de datos de back-end son siempre idénticos), eso se llama un sistema con *consistencia fuerte*.

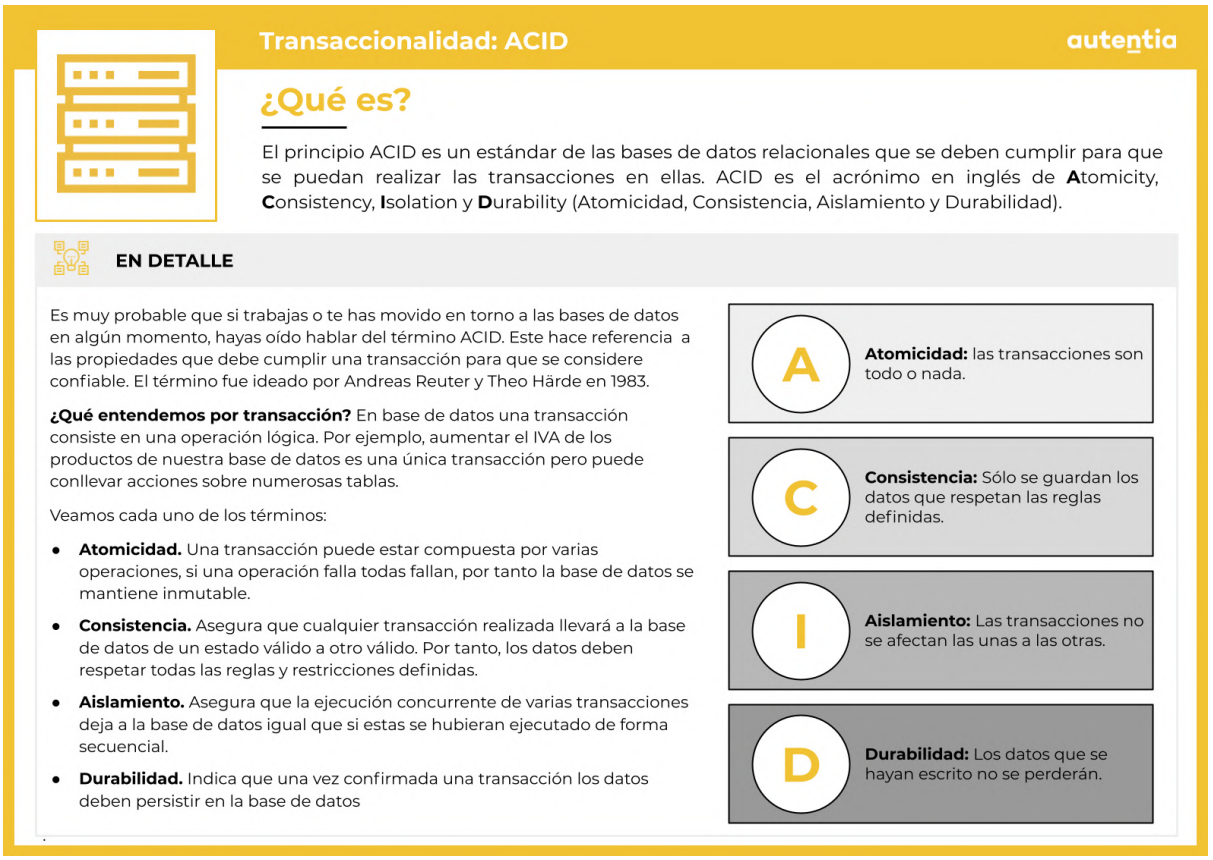
¿Pero qué desventajas puede tener esta solución? La desventaja de este enfoque es que el **rendimiento** de sistemas fuertemente consistentes puede resentirse y que el sistema sea menos útil. ¿Qué pasa si nuestro móvil pierde contacto con el servidor? En un mundo muy "consistente", no podrá mostrarnos el saldo bancario, incluso si sólo se ha actualizado hace 5 minutos y es 90% probable que sea exacto. La mayoría de los usuarios consideran que el requisito de la consistencia fuerte es molestamente exagerado. Es incluso peor en un sistema en el que las operaciones

individuales pueden tardar mucho en completarse. Las transacciones se ponen en cola una detrás de la otra y todo se detiene.

Con los sistemas con la consistencia eventual no pasa esto. Las copias de datos no siempre tienen que ser idénticas siempre que estén diseñadas para volverse coherentes una vez que se hayan procesado todas las operaciones actuales. Si necesitas solo la coherencia final, la aplicación de tu móvil podría mostrarte el último valor de saldo (si está lo suficientemente actualizado como para ser razonablemente confiable) incluso si el servidor de base de datos no está disponible para realizar peticiones en este mismo momento.

Transacciones

Una aplicación monolítica suele tener una única base de datos relacional. Un beneficio clave de usar una base de datos relacional es que su aplicación puede usar transacciones *ACID*.



Como resultado, la aplicación puede simplemente comenzar una transacción, cambiar (insertar, actualizar y eliminar) varias filas y confirmar la transacción.

Desafortunadamente, el acceso a los datos se vuelve mucho más complejo cuando pasamos a una arquitectura de microservicios. Esto se debe a que los datos que posee cada microservicio son privados para ese microservicio y solo se puede acceder a ellos a través de su API. La encapsulación de los datos garantiza que los microservicios estén débilmente acoplados y puedan evolucionar de forma independiente entre sí. Si varios servicios acceden a los mismos datos, las actualizaciones de esquemas requieren actualizaciones coordinadas y requieren mucho tiempo para todos los servicios.

Una arquitectura así tiene muchos beneficios como servicios poco acoplados y un mejor rendimiento y escalabilidad. Sin embargo, presenta

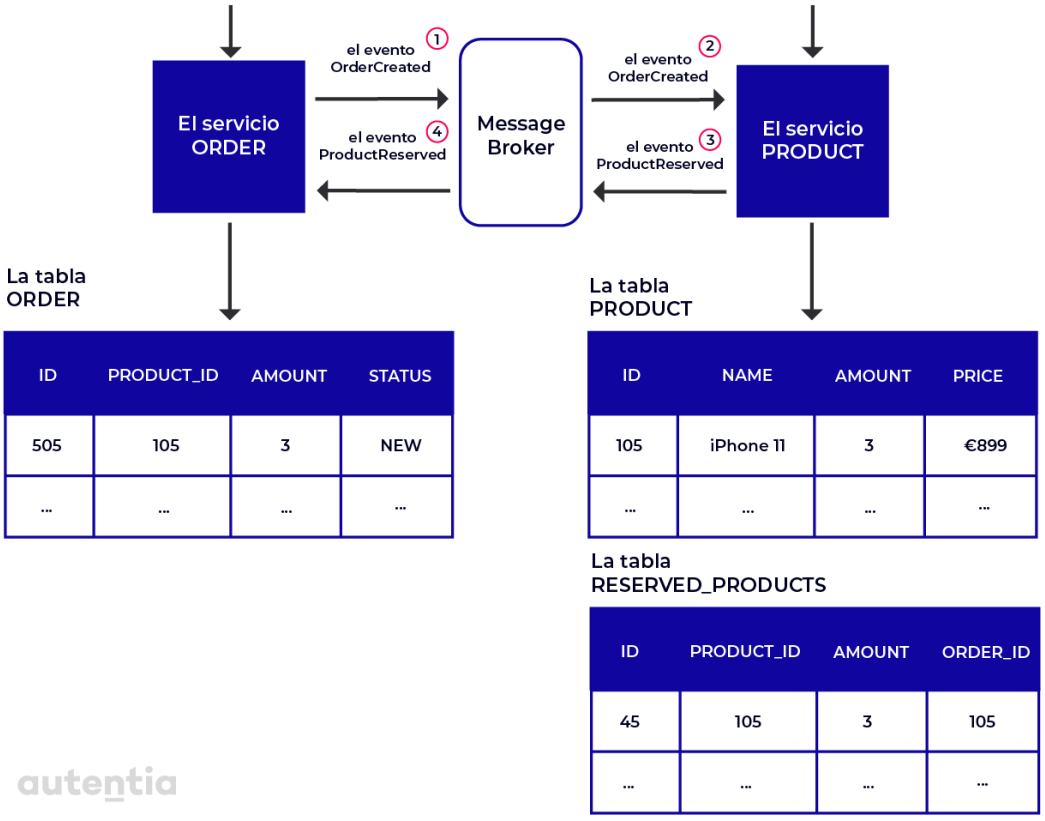
diferentes desafíos a la hora de gestionar los datos distribuidos.

El primer desafío es cómo implementar transacciones que mantengan la **coherencia entre múltiples servicios**. No podemos acceder directamente a la base de datos de otros servicios, pero tenemos que sincronizar los datos entre servicios.

El otro desafío es cómo implementar las peticiones que **obtengan los datos de diferentes servicios**. No siempre los diferentes servicios nos proveen la API que nos da la información necesaria.

Para solucionar estos tipos de problemas se aplican las arquitecturas orientadas a los eventos. En esta arquitectura un microservicio publica un evento cuando sucede algo notable, como cuando actualiza una entidad. Otros microservicios se suscriben a estos eventos. Cuando un microservicio recibe un evento, puede actualizar sus propias entidades, lo que podría dar lugar a que se publiquen más eventos.

De este modo, podemos utilizar los eventos para implementar **transacciones que abarcan varios servicios**. Una transacción consiste normalmente, en una serie de pasos. Cada paso consiste en un microservicio que actualiza una entidad de negocio y que publica un evento que desencadena el siguiente paso:



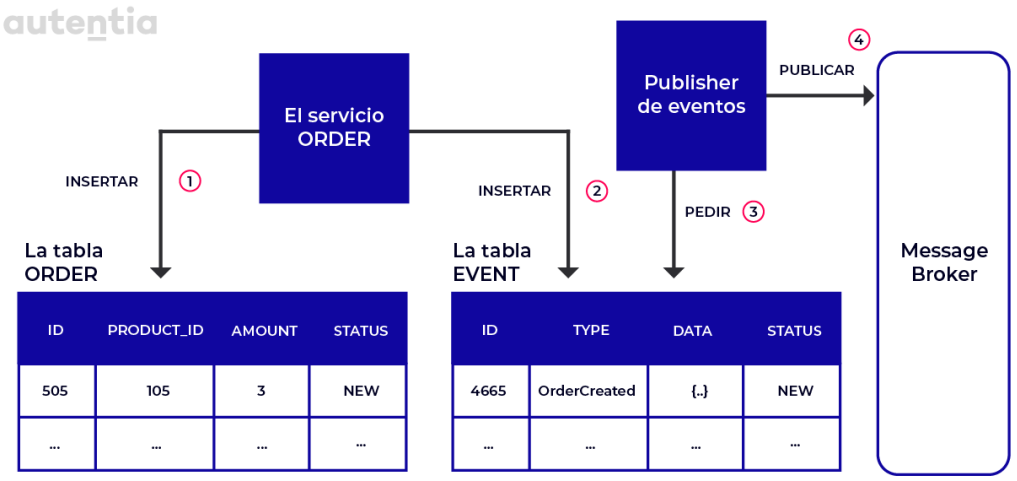
Como podemos observar, los microservicios intercambian eventos a través de un message broker. Siempre que cada servicio actualice automáticamente la base de datos y publique un evento y el message broker garantice que los eventos se entregan al menos una vez, se pueden implementar transacciones que abarquen varios servicios. Es importante señalar que estas no son transacciones *ACID*. Ofrecen una garantía mucho más débil: *BASE* o la *consistencia eventual*.

Una arquitectura orientada a eventos tiene varios beneficios e inconvenientes. Permite la implementación de transacciones que abarcan múltiples servicios y proveen la consistencia eventual. El inconveniente es que el modelo de programación es más complejo ya que requiere un aprendizaje específico. También tienes que implementar mecanismos de compensación para recuperarse después de los errores en el nivel de la aplicación. Además, las aplicaciones deben lidiar con los datos

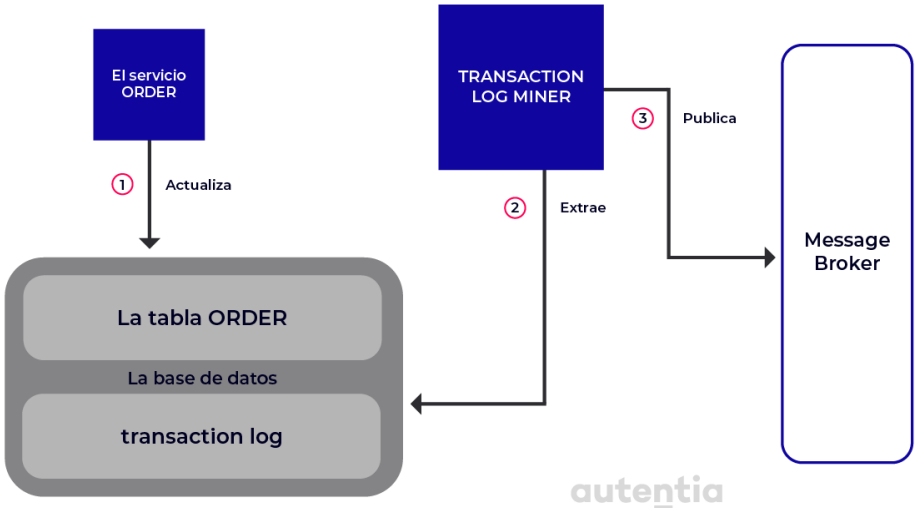
inconsistentes. Otro inconveniente es que los suscriptores deben detectar e ignorar los eventos duplicados.

En una arquitectura orientada a eventos también existe el problema de actualizar atómicamente la base de datos y publicar un evento. Por ejemplo, el servicio de pedidos debe insertar una fila en la tabla de pedidos y publicar un evento de pedido creado. Es fundamental que estas dos operaciones se realicen de forma **atómica**. Si el servicio falla después de actualizar la base de datos pero antes de publicar el evento, el sistema se vuelve **incoherente**. La forma estándar de garantizar la atomicidad es utilizar una transacción distribuida que involucre la base de datos y un message Broker. Pero tienes que elegir entre la disponibilidad y la coherencia.

Una forma de lograr la atomicidad es que la aplicación publique eventos mediante un proceso de varios pasos que tienen solo transacciones locales. El truco consiste en tener una tabla de eventos EVENT, que funciona como una cola de mensajes, en la base de datos que almacena el estado de las entidades comerciales. La aplicación inicia una transacción de base de datos (local), actualiza el estado de las entidades comerciales, inserta un evento en la tabla EVENT y confirma la transacción. Un proceso o subproceso de aplicación independiente consulta la tabla EVENT, publica los eventos en Message Broker y luego usa una transacción local para marcar los eventos como publicados. El siguiente diagrama muestra el diseño.



Otra forma de lograr la atomicidad es que los eventos sean publicados por un hilo o proceso que extrae la transacción del *log* de base de datos. La aplicación actualiza la base de datos, lo que da como resultado que los cambios se registren en el registro de transacciones de la base de datos. El subproceso de *transaction log miner* lee el registro de transacciones y publica eventos en el *message broker*. El siguiente diagrama muestra este proceso.



Este método tiene varios beneficios e inconvenientes. Un beneficio es que garantiza que se publique un evento por cada actualización. La extracción

de registros de transacciones también puede simplificar la aplicación al separar la publicación de eventos de la lógica empresarial de la aplicación. Un inconveniente importante es que el formato del registro de transacciones es único para cada base de datos e incluso puede cambiar entre las versiones de la base de datos. Además, puede resultar difícil crear eventos de alto nivel a partir de las actualizaciones de bajo nivel registradas en el registro de transacciones.

Bibliografía

Estas son las fuentes que hemos consultado y en las que nos hemos basado para la redacción de este material:

- The 4 rules of simple design:

  https://blog.jbrains.ca/permalink/the-four-elements-of-simple-design

  https://blog.thecodewhisperer.com/permalink/putting-an-age-old-battle-to-rest

  Understanding the Four Rules of Simple Design, Corey Haines

- The Clean Code Blog by Robert C. Martin (Uncle Bob):

  https://blog.cleancoder.com

- Head First Design Patterns. Eric Freeman & Elisabeth Freeman.
- Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. [GoF]
- Core J2EE Patterns: Best Practices and Design Strategies. Deepak Alur, John Crupi, Dan Malks.
- Patterns of Enterprise Application Architecture. Martin Fowler con la colaboración de David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee y Randy Stafford.
- AntiPatterns: https://sourcemaking.com/antipatterns
- AntiPatterns Wikipedia: https://en.wikipedia.org/wiki/Anti-pattern
- Service Activator Pattern: https://www.javaguides.net/2018/08/service-activator-pattern-in-java.html

- Core J2EE http://index-of.co.uk/Programming/SUN%20-%20Core%20J2EE%20patterns.pdf
- Java Design Patterns https://java-design-patterns.com/patterns/
- Core J2EE Patterns http://www.corej2eepatterns.com/
- https://josecuellar.net/domain-driven-design-episodio-iii-arquitectura/
- https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/
- https://blog.ploeh.dk/2013/12/03/layers-onions-ports-adapters-its-all-the-same/
- https://martinfowler.com/bliki/CQRS.html
- “Domain-Driven Design: Tackling Complexity in the Heart of Software”, Eric Evans, 2003

Lecciones aprendidas con esta guía

En esta guía hemos visto los principales principios, patrones y antipatrones. Estas prácticas no son más que la experiencia de los profesionales del sector, reunidas en una serie de recetas o buenas prácticas para que el **software crezca y evolucione de una forma sostenible.**

Hemos visto los **malos olores** que puede desprender el código y como profesionales, es nuestra responsabilidad irlos solucionando poco a poco y no ir generando una **deuda técnica** que sea como una bola de nieve.

Hacer de **boy scout** va implícito en nuestras tareas del día a día y es lo que va a permitir que las futuras entregas de valor sean rápidas, de calidad y que no rompan funcionalidad existente.

A menudo se pide permiso a managers, product owners o gente de desarrollo de negocio para acometer estas tareas asumiendo que es su decisión, ya que son tareas que conllevan una inversión de tiempo. Esto es un error de base. **La construcción de software es responsabilidad de los desarrolladores de software,** lo mismo que el estado del software.

Conocer los **patrones creacionales, estructurales y de comportamiento** nos van a proporcionar herramientas válidas para entregar valor constante y de calidad.

Conocer las **buenas prácticas y los principios de diseño** nos va a proporcionar las herramientas necesarias para afrontar nuevos desarrollos con seguridad, sabiendo qué decisiones tomar, cuáles aplazar y cómo construir software sobre pilares sólidos, que cambio tras cambio se mantengan firmes.

Los principales **patrones y antipatrones** nos ayudan a encontrar una solución adecuada y probada por muchos desarrolladores.
Nos ahorran tiempo y favorecen crear una arquitectura mantenible, sostenible y extensible.

Además, los antipatrones pueden darte una visión de malas prácticas y así evitar su uso en tu desarrollo.
Como el código es sólo una parte de la solución, también hemos añadido arquitecturas de referencia que están ampliamente aceptadas hoy en día, de forma que la estructura y organización de nuestro código responda a conceptos como Open-Closed como arquitectura.
Tener presentes las ventajas e inconvenientes de cada una de las arquitecturas expuestas, así como la capa en la que aplican, nos permitirá tomar las mejores decisiones de diseño posibles, así como elegir la combinación de estas que mejor se adapten a nuestro problema.

En Autentia proporcionamos soporte al desarrollo de software y ayudamos a la transformación digital de grandes organizaciones siendo referentes en eficacia y buenas prácticas. Te invito a que te informes sobre los servicios profesionales de Autentia y el soporte que podemos proporcionar para la transformación digital de tu empresa.

¡Conoce más!

Expertos en creación de software de calidad

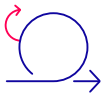
Diseñamos productos digitales y experiencias a medida

SOPORTE A DESARROLLO

Construimos entornos sólidos para los proyectos, trabajando a diario con los equipos de desarrollo.

DISEÑO DE PRODUCTO Y UX

Convertimos tus ideas en productos digitales de valor para los usuarios finales.

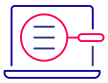


AGILE & CULTURE

Conectamos la estrategia con la ejecución, ayudándote a capitalizar las oportunidades del mercado.

DESARROLLO DE SOFTWARE

Te ayudamos a impulsar tu negocio mediante la construcción de software de calidad que apoye la transformación digital de tu organización.



AUDITORÍA

Analizamos la calidad técnica de tu producto y te ayudamos a recuperar la productividad perdida.

FORMACIÓN

Formamos empresas, con clases impartidas por desarrolladores profesionales en activo.

www.autentia.com

info@autentia.com | T. 91 675 33 06

¡Síguenos en nuestros canales!

Adictos al trabajo | by autentia