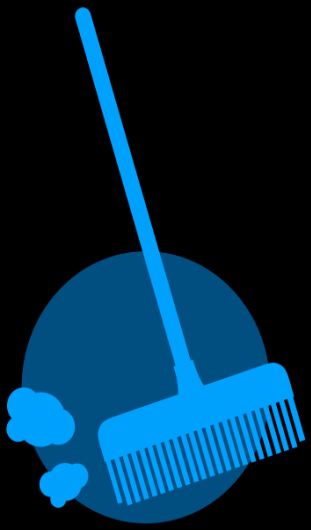

LA GUÍA DEL REFACTOR COTIDIANO

FRAN IGLESIAS

La guía del refactor cotidiano

Mantener tu código en forma es un trabajo para todos los días

Fran Iglesias

Este libro está a la venta en
http://leanpub.com/refactorcotidiano

Esta versión se publicó en 2023-09-20

Éste es un libro de Leanpub. Leanpub anima a los autores y publicadoras con el proceso de publicación. Lean Publishing es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

A Isabel

Índice general

Introducción de la primera edición

En The Talking Bit hemos escrito bastante sobre refactor[1], principalmente porque nos parece que es una de las mejores cosas que podemos hacer con nuestro código, sea nuevo o legacy.

El refactor es una forma de mantenimiento del código que consiste en mejorar su expresividad a través de pequeños cambios que no alteran el comportamiento y que tampoco cambian sustancialmente la implementación.

Una reescritura, por el contrario, suele plantear un cambio brusco de implementación que podría incluso provocar algunos cambios en el comportamiento.

Por otro lado, el refactor puede hacerse de una manera continua e iterativa, interviniendo en el código siempre que se tenga ocasión. Por ejemplo: porque estamos revisándolo a fin de corregir un error o añadir una nueva característica.

Durante el proceso de lectura y análisis podemos encontrarnos con fragmentos de código que no expresan bien un concepto del dominio, que no se entienden fácilmente o que interfieren en la compresión de ese código. Ese momento es ideal para realizar pequeños refactors seguros que, acumulados a lo largo del tiempo, van haciendo que el código evolucione hacia un mejor diseño. Pero sobre todo, hacen que el código refleje cada vez mejor el conocimiento que tenemos.

[1] https://franiglesias.github.io/tag/refactoring/

Como ya hemos mencionado en otras ocasiones, el refactor trata principalmente sobre conocimiento y significado[2]. Es decir, trata sobre que el código exprese cosas y, concretamente, que exprese de la mejor forma posible nuestro conocimiento sobre el dominio en el que trabajamos y cómo estamos resolviendo los problemas que nos plantea.

Por esa razón, se nos ha ocurrido que podría ser buena idea crear una especie de curso o guía para aprender sobre cómo hacer *refactor cotidiano*.

Refactor sin piedad

El refactor cotidiano es otro nombre para una práctica de *Extreme Programming* conocida como *refactor sin piedad*. Dicho en pocas palabras, consiste en refactorizar el código en cualquier momento en que sea posible. Contra lo que podría parecer no se trata de engolfarse dando vueltas al código hasta llegar a un código perfecto, sino de realizar retoques limitados para corregir lo que en ese momento podemos considerar como pequeñas imperfecciones. Por ejemplo:

- Un nombre no consigue expresar correctamente un concepto.
- Una expresión compleja podría descomponerse otras más pequeñas, más fáciles de entender y mantener.
- El cuerpo de un método o función es muy largo y puede ser dividido en secciones autónomas.
- Una expresión condicional es difícil de entender y se puede encapsular como función o método con un nombre expresivo, ocultando sus detalles.

[2] https://franiglesias.github.io/refactor-knowledge-meaning/

- Una estructura condicional presenta demasiada anidación, de modo que extraemos sus partes para organizarlo según niveles de abstracción.
- Un fragmento de código está en lugar inadecuado y lo movemos a donde corresponde.

Con el tiempo, la acumulación de estos pequeños cambios irá mejorando la estructura, expresividad y mantenibilidad del código. Es muy posible que acabe revelando oportunidades de cambio de mayor calado que tal vez impliquen un rediseño.

En cualquier caso, al mejorar la situación del código, mejoran nuestras posibilidades de intervenir en él. Incorporar nuevas prestaciones será más rápido y seguro, lo que nos permite entregar con mayor frecuencia y predictibilidad.

Los momentos del refactor

En su charla *Flows of refactoring*, Martin Fowler explica qué es realmente el refactor y cuándo se tendría que hacer.

Fowler insiste en la idea de que si necesitas un momento especial para refactorizar es que no lo estás haciendo, sino, tal vez, reescribiendo o cambiando el diseño de tu software.

Existen tres momentos principales para el refactor:

- Durante la lectura de código
- Refactor preparatorio para introducir un cambio
- Refactor posterior a un cambio

Lectura de código, los momentos WTF

Pasamos la mayor parte del tiempo leyendo código. Cada vez que queremos introducir un cambio necesitamos leer el código

existente para saber dónde es el mejor lugar para hacerlo o averiguar qué recursos ya están disponibles. También leemos el código simplemente para buscar algún tipo de conocimiento relacionado con el código mismo o el negocio.

En esos momentos podemos encontrarnos con fragmentos que nos hagan lanzar una interjección (WTF!), o que necesitamos leer dos o tres veces para entender lo que ocurre. Momentos en los que vemos algo que, por alguna razón no cuadra. No es que no funcione, es simplemente que hay algo descolocado, sucio, fuera de sitio, desentonando. En una palabra: desordenado.

Por lo general, estos momentos de extrañeza los genera la existencia de una distancia entre nuestro conocimiento del negocio y cómo se expresa en el propio código. Este desfase es lo que Ward Cunningham denominó *deuda técnica*.

La deuda técnica se paga refactorizando el software para que refleje el conocimiento del negocio de la mejor manera posible.

En cierto modo, es imposible evitar que el código tenga un cierto nivel de deuda técnica porque el conocimiento del negocio cambia mucho más rápidamente. De hecho, cuando introducimos cambios en el software y los llevamos a producción, el negocio ya está cambiando. En parte, gracias a las nuevas funcionalidades del software y al impacto que tienen.

La forma de prevenir el crecimiento de la deuda técnica es entregar de forma sostenida pequeños cambios y refactorizar constantemente.

Refactor preparatorio

Al introducir cambios en el código para incorporar nuevas funcionalidades o corregir errores suele ser necesario arreglar

el código existente para hacer posibles esos mismos cambios. Una cita de Kent Beck lo expresa más o menos así:

> For each desired change, make the change easy (warning: this may be hard), then make the easy change

Es decir:

> Primero, haz que el cambio sea fácil (advertencia: puede ser difícil conseguirlo). Luego haz el cambio fácil.

Con frecuencia el código no está preparado para admitir ciertos cambios, por lo que debemos transformarlo primero a fin de que la nueva funcionalidad sea fácil de introducir.

Esto es lo que denominamos refactor preparatorio y debería hacerse siempre antes de introducir los nuevos cambios. En otras palabras: mejoramos la estructura del código asegurándonos de no cambiar su comportamiento. Mezclamos esos cambios y solo entonces procedemos a desarrollar la nueva funcionalidad.

Usando otra metáfora de Kent Beck: usamos dos sombreros: el de refactorizar y el de crear nueva funcionalidad. No podemos llevar los dos sombreros a la vez.

Refactor posterior a un cambio

Tras realizar los cambios necesarios volvemos a encasquetarnos el sombrero de refactor para limpiar el código que acabamos de introducir.

Es posible que nuestra nueva funcionalidad haya introducido algo de duplicación y esto podría revelar la existencia de un

concepto más general que hasta entonces no habíamos descubierto. En otras ocasiones, el código nuevo incrementa la complejidad de algún área y debemos trabajar para simplificarla.

Al añadir código a un sistema de software, aumentamos su entropía o desorden. La entropía de software es un concepto introducido por Ivar Jacobson y otros en el libro *Object-Oriented Software Engineering: A Use Case Driven Approach*. La mejor forma de luchar contra un crecimiento desmesurado de la entropía, es aplicar un refactor continuado, particularmente tras introducir código nuevo.

Cómo refactorizar

La idea del refactor cotidiano es muy simple:

Se trata de realizar pequeños cambios inocuos en nuestro código en cualquier momento que se nos presente la ocasión. Es lo que algunos autores denominan **refactor oportunista**.

Nuestra propuesta concreta es que hagas un refactor muy pequeño cada vez que lo veas necesario, de modo que, en una primera fase:

- solo tocas un archivo.
- los cambios quedan recogidos en un único commit atómico, que contengan solo los cambios debidos a ese refactor.

En una segunda fase:

- los cambios podrían a varios archivos, pero el ámbito es limitado.
- los cambios quedan recogidos en un único commit atómico.

En una tercera fase:

- los cambios podrían suponer introducción de nuevas clases.
- de nuevo, los cambios quedarían recogidos en un único commit.

La guía

Esta guía se compone de una serie de capítulos en los que se exponen diversas orientaciones y principios que seguir a la hora de refactorizar. La idea es explicar ámbitos en los que podrías intervenir en los tres niveles indicados en el apartado anterior.

En muchos casos los refactors propuestos, al menos en el primer nivel o fase no necesitarían tests porque podrían ejecutarse mediante herramientas del IDE.

Con el tiempo, es posible que esos pequeños refactors acumulado día tras día, mejoren la forma y calidad de tu código y te despejen caminos para mejorar su expresividad y arquitectura.

Así que, *¡happy tiding!*

Cuando los comentarios confunden

Nuestro primer capítulo trata sobre los comentarios. Con frecuencia entramos en un código nuevo y nos dirigimos en primer lugar a los comentarios para tratar de entenderlo. Y, sin embargo, muchas veces lo que debería ser una documentación del proyecto no hace más que sumirnos en confusión.

Comentarios y documentación

Los lenguajes de programación incluyen la posibilidad de insertar comentarios en el código como forma de añadir documentación al mismo. Es decir: el objetivo de los comentarios es añadir conocimiento cerca del lugar en el que puede ser necesario.

Los comentarios en el código parecen una buena idea y, probablemente, eran más útiles en otros tiempos, cuando la necesidad de economizar recursos y las limitaciones de los lenguajes de programación no nos permitían escribir un código lo bastante expresivo como para ser capaz de documentarse a sí mismo.

En esta entrega, intentaremos explicar qué comentarios nos sobran y por qué, y cuáles dejar.

¿Por qué deberías eliminar comentarios?

Las principales razones para borrar comentarios son:

Suponen una dificultad añadida para leer el código. En muchos aspectos, los comentarios suponen una narrativa paralela a la del código y nuestro cerebro tiende a enfocarse en una de las dos. Si nos enfocamos en la de los comentarios, no estamos leyendo el código. Si nos enfocamos en la del código: ¿para qué queremos comentarios?

Los comentarios suponen una carga cognitiva. Incluso leyéndolos con el rabillo del ojo, los comentarios pueden suponer una carga cognitiva si, de algún modo, discrepan con lo que el código dice. Esto puede interrumpir tu flujo de lectura hasta que consigues aclarar si ese comentario tiene algún valor o no.

Pueden alargar innecesariamente un bloque de código. Idealmente, deberías poder leer un bloque de código en una sola pantalla. Los comentarios añaden líneas que podrían provocar que tengas que deslizar para ver todo el bloque. Son especialmente problemáticos los que están intercalados con las líneas de código.

Pueden mentir. Con el tiempo, si no se hace mantenimiento de los comentarios, estos acaban siendo mentirosos. Esto ocurre porque los cambios en el código no siempre son reflejados con cambios en los comentarios por lo que llegará un momento en que unos y otro no tengan nada que ver.

Refactor de comentarios

Básico

Simplemente, eliminamos los comentarios que no necesitamos. Es un refactor completamente seguro, ya que no afecta de ningún modo al código.

Reemplazar comentarios por mejores nombres

Eliminamos comentarios obvios, redundantes o innecesarios, cambiando el nombre de los símbolos que tratan de explicar.

Cambiar nombres es un refactor muy seguro, sobre todo con la ayuda de un buen IDE, que puede realizarlo automáticamente, y dentro de ámbitos seguros, como método o variables privadas.

Reemplazar comentarios por nuevas implementaciones

En algunos casos podríamos plantearnos mejorar el diseño de una parte del código porque al reflexionar sobre la necesidad de mantener un comentario nos damos cuenta de que es posible expresar la misma idea en el código.

Este tipo de refactor no encaja en la idea de esta serie sobre refactor cotidiano, pero plantea el modo en que los pequeños refactors del día a día van despejando el camino para refactors e incluso reescrituras de mayor alcance.

Comentarios redundantes

Los comentarios redundantes son aquellos que nos repiten lo que ya dice el código, por lo que podemos eliminarlos.

Por ejemplo:

```
// Class to represent a Book
class Book
{
    //...
}
```

En serio, ¿qué nos aporta este comentario que no esté ya expresado?

```
class Book
{
    //...
}
```

Los lenguajes tipados, que soportan *type hinting* y/o *return typing*, nos ahorran toneladas de comentarios.

```
class Book
{
    /**
    * Creates a book with a title and an author
    *
    * @param Title $title
    * @param Author $author
    * @returns Book
    */
    public static function create(Title $title, A\
uthor $author): Book
    {
        //
```

```
14      }
15  }
```

Los tipos de los parámetros y del objeto devuelto están explícitos en el código, por lo que es redundante que aparezcan como comentarios.

```
1   class Book
2   {
3       /**
4       * Creates a book with a title and an author
5       */
6       public static function create(Title $title, A\
7   uthor $author): Book
8       {
9           //
10      }
11  }
```

Excepciones: este tipo de comentarios tiene su razón de ser cuando no podemos hacer explícitos los tipos.

Puedes eliminar los comentarios redundantes poniendo mejores nombres. Por ejemplo, en este caso en que utilizamos un constructor secundario:

```php
class Book
{
    /**
    * Creates a book with a title and an author
    */
    public static function create(Title $title, A\
uthor $author): Book
    {
        //
    }
}
```

Con un nombre expresivo ya no necesitamos comentario:

```php
class Book
{
    public static function withTitleAndAuthor(Tit\
le $title, Author $author): Book
    {
        //
    }
}
```

O incluso más explícito:

```
class Book
{
    public static function newWithTitleAndAuthor(\
Title $title, Author $author): Book
    {
        //
    }
}
```

Y podemos usar el objeto así, lo cual documenta perfectamente lo que está pasando:

```
$newBook = Book::withTitleAndAuthor($title, $auth\
or);
```

```
$newBook = Book::newWithTitleAndAuthor($title, $a\
uthor);
```

Más excepciones: si lo que estamos desarrollando es una librería que pueda utilizarse en múltiples proyectos, incluso que no sean nuestros, los comentarios que describen lo que hace el código pueden ser necesarios.

Comentarios mentirosos

Los comentarios mentirosos son aquellos que dicen algo distinto que el código. Y, por tanto, deben desaparecer.

¿De dónde vienen los comentarios mentirosos? Sencillamente, ha ocurrido que los comentarios se han quedado olvidados, sin mantenimiento, mientras que el código ha evolucionado. Por

eso, cuando los lees hoy es posible que digan cosas que ya no valen para nada.

Mi experiencia personal con este tipo de comentarios cuando entro a un código nuevo suele ser bastante negativa. Si el comentario y el código difieren te encuentras con el problema de decidir a cuál de los dos hacer caso. Lo cierto es que el código manda porque es lo que realmente se está ejecutando y lo que está produciendo resultados, pero la existencia del comentario genera esa inquietud: ¿por qué el comentario dice una cosa y el código hace otra?

Habitualmente, debería ser suficiente con comprobar si hay diferencias en la fecha en que se añadió el comentario y la fecha en la que se modificó el código. Si esta es posterior, es que tenemos un caso de comentario mentiroso por abandono y lo más adecuado sería borrarlo.

Este hecho debería bastar para que no añadas nuevos comentarios sin una buena razón. Tendemos a ignorar los comentarios triviales, de modo que cuando cambiamos el código nos despreocupamos de mantenerlos actualizados y acaban siendo mentirosos. Así que procuraremos dejar solo aquellos comentarios que nos importen realmente.

Si ya nos hemos librado de los comentarios redundantes, deberíamos contar solo con los que pueden aportar alguna información útil, así que nos toca examinarlos para asegurarnos de que no sean mentirosos. Y serán mentirosos si no nos cuentan lo mismo que cuenta el código.

Puede parecer un poco absurdo, pero al fin y al cabo los comentarios simplemente están ahí y nos les prestamos mucha atención, salvo que sea la primera vez que nos movemos por cierto fragmento de código y tratamos de aprovechar cualquier información que nos parezca útil. Es entonces cuando descubrimos comentarios que pueden oscilar entre lo simplemente desactualizado y lo esperpéntico.

Así que, fuera con ellos. Algunos ejemplos:

To-dos olvidados. Las anotaciones **To do** seguramente hace meses que han dejado de tener sentido. Mienten en tanto que no tenemos ninguna referencia que les aporte significado.

¿De qué otro tipo de servicio estábamos hablando aquí hace tres meses? ¿Será que ya lo hemos cambiado?

```
// @todo we should use another kind of service he\
re

$service = new Service();
$service->execute();
```

Sería diferente si el comentario fuese mucho más preciso y detallado, de tal forma que indique con claridad el ámbito y plazo de la tarea pendiente. Algo así:

```
/**
@todo we should replace this service with the new\
 implementation that support Kafka brokers when i\
nfra team finished the migration from SQS, schedu\
led for 18/09/2023
*/

$service = new Service();
$service->execute();
```

En ese caso, el comentario hace explícitos unos detalles que definen con precisión los motivos, acciones y plazos.

Comentarios olvidados. En algunos casos puede ocurrir que simplemente nos hayamos dejado comentarios olvidados. Por

ejemplo, podríamos haber usado comentarios para definir las líneas básicas de un algoritmo, que es una técnica bien conocida, y ahí se habrían quedado. Todo ello también tiene que desaparecer:

```
public function calculateFee(Request $dataToCalcu\
late)
{
    // Normalize amounts to same currency

    // ... code here

    // Perform initial calculation

    // ... more code here

    // Apply transormation

    // .., more code here
}
```

Comentarios para estructurar código. Claro que puede que el algoritmo sea lo bastante complejo como para que necesitemos describir sus diferentes partes. En este caso, el mejor refactor es extraer esas partes a métodos privados con nombres descriptivos, en lugar de usar comentarios:

```
public function calculateFee(Request $dataToCalcu\
late)
{
    $this->normalizeAmountsToTheSameCurrency($dat\
aToCalculate);
    $initialCalculation = $this->performInitialCa\
lculation($dataToCalculate);
    $transformedResponse = $this->applyTransforma\
tion($initialCalculation);
}
```

De este modo el código está estructurado y documentado.

Comentarios sobre valores válidos. Consideremos este código:

```
// Valid values: started, paused, running, termin\
ated
public function updateStatus(string $newStatus): \
void
{
    $this->checkValidStatus($newStatus);
    $this->status = $newStatus;
}
```

El comentario delimita los valores aceptables para un parámetro, pero no fuerza ninguno de ellos. Eso tenemos que hacerlo mediante una cláusula de guarda. ¿Hay una forma mejor de hacerlo?

Por supuesto: utilizar un *enumerable*.

```php
class Status
{
    private const STARTED = 'started';
    private const PAUSED = 'paused';
    private const RUNNING = 'running';
    private const TERMINATED = 'terminated';

    private $value;

    private function __construct(string $status)
    {
        $this->checkValidStatus($status);
        $this->status = $status;
    }

    public static function fromString(string $sta\
tus): Status
    {
        return new self($status);
    }

    public static function started(): Status
    {
        return new self(self::STARTED);
    }

    //...
}
```

Lo que permite eliminar el comentario, a la vez que tener una

implementación más limpia y coherente:

```
public function updateStatus(Status $newStatus): \
void
{
    $this->status = $newStatus;
}
```

Código comentado

En alguna parte he escuchado o leído algo así como "código comentado: código borrado". El código comentado debería desaparecer. Lo más seguro es que ya nadie se acuerde de por qué estaba ese código ahí, para empezar, y por qué sigue aunque sea escondido en un comentario.

Si es necesario recuperarlo (spoiler: no lo será) siempre nos queda el control de versiones.

Excepciones: a veces se puede usar la técnica de comentar un código para desactivarlo temporalmente. En ese caso, deberíamos explicar esa decisión también en el mismo comentario. Mucho mejor que eso es utilizar alguna técnica de *feature flag*. Existen librerías en todos los lenguajes para gestionar *feature flags*, pero en muchos casos podemos introducer alguna variable que sea fácil de cambiar:

```
if ($useNewCode === true) {
    $this->newCode();
} else {
    $this->oldCode();
}
```

Comentarios que podríamos conservar... o no

Comentarios que explican decisiones

Los buenos comentarios deberían explicar por qué tomamos alguna decisión que no podemos expresar mediante el propio código y que, por su naturaleza, podríamos considerar como independientes de la implementación concreta que el código realiza. Es decir, no deberíamos escribir comentarios que expliquen cómo es el código, que es algo que ya podemos ver, sino que expliquen por qué es así.

Lo normal es que estos comentarios sean pocos, pero relevantes, lo cual los pone en una buena situación para realizar un mantenimiento activo de los mismos.

Obviamente, corremos el riesgo de que los comentarios se hagan obsoletos si olvidamos actualizarlos cuando sea necesario. Por eso la importancia de que no estén "acoplados" a la implementación en código.

Un ejemplo de comentario relevante podría ser este:

```
1  // We apply taxes to conform the procedure stated\
2   in law RD 2018/09
3  public function applyTaxes(Money $totalAmountBefo\
4  reTaxes): Money
5  {
6      //... some code here
7  }
```

Este comentario es completamente independiente del código e indica una información importante que no podríamos expre-

sar con él. Si en un momento dado cambia la legislación y debemos aplicar otra normativa, podemos cambiar el comentario.

Aunque, a decir verdad, podríamos llegar a expresarlo en código. A grandes rasgos:

```
interface Taxes
{
    public function apply(Money $amountBeforeTaxe\
s): Money;
}

class RD201809Taxes implements Taxes
{
    public function apply(Money $amountBeforeTaxe\
s): Money
    {
        // ... some code here
    }
}


class RD201821Taxes implements Taxes
{
    public function apply(Money $amountBeforeTaxe\
s): Money
    {
        // ... some code here
    }
}
```

Dudas razonables

Comentarios para el IDE

En aquellos lenguajes en los que el análisis estático por parte del IDE no pueda interpretar algunas cosas, añadir comentarios en forma de anotaciones puede suponer una ayuda para el IDE. En algunos casos, gracias a eso el IDE nos avisa de problemas potenciales antes de integrar los cambios.

No debería ser una práctica común, pero es un compromiso aceptable. Por ejemplo, en PHP era frecuente indicar el tipo de las propiedades de los objetos y otras variables con comentarios, ya que el lenguaje no permitía hacerlo en código.

```
1  class Status
2  {
3      /** string **/
4      private $value;
5  }
```

Esto se introdujo en la versión 7.4:

```
1  class Status
2  {
3      private string $value;
4  }
```

En otros lenguajes, estas características estaban presentes desde mucho antes.

Las seis preguntas

El framework de las seis preguntas se utiliza en algunas disciplinas para determinar si una fuente proporciona información completa. Estas preguntas se pueden usar para decidir qué comentar en un código y qué no es necesario:

- **¿Cuándo se ha escrito el código?**: Esta información la encontramos fácilmente en el sistema de control de versiones. No hay que añadirla como comentario. El único caso en que se me ocurre que podría ser útil es cuando mudamos un repositorio a un servidor diferente, ya que se puede perder la información.
- **¿Quién ha escrito el código?**: Aplica lo mismo que en la pregunta anterior, es información que nos proporciona el sistema de control de versiones, de una forma mucho más precisa.
- **¿Dónde está el código?**: Aplicado a la *paquetización* del código, básicamente es una información que o bien se declara de forma explícita, o bien el lenguaje se encarga de reportarnos en caso de errores. Por tanto, tampoco parece necesario establecerlo en un comentario.
- **¿Qué hace el código?**: La respuesta corta es que el código ya dice lo que hace, pero con frecuencia eso no queda tan claro porque los nombres están mal escogidos o la estructura del código lleva a confusión. Eso podría llevarnos a plantear la necesidad de indicarlo en un comentario. Pero antes de ellos, lo apropiado sería reflexionar sobre cómo explicitar la intención de ese fragmento de código usando un buen nombre.
- **¿Cómo hace el código lo que hace?**: De nuevo, una vez que sabemos lo que hace un código, el cómo debería ser el código en sí. Ahora bien, hay algunos casos en los que es recomendable añadir comentarios. Uno de esos casos es

el uso de algoritmos bien conocidos, que tienen nombre. En esa situación, es muy buena idea hacerlo explícito. Otro caso podría ser el de documentar distintos pasos en un algoritmo, aunque para ello suele ser mejor extraerlos a sus propios métodos.

- **¿Por qué hace el código lo que hace?**: Finalmente, esta es una pregunta que solo podemos contestar nosotras: las personas responsables de ese conocimiento. Y esa explicación debe aparecer como comentario.

Resumen del capítulo

Los comentarios en el código tienen una utilidad limitada y, con frecuencia, se vuelven mentirosos y no resultan de ayuda para comprender lo que nuestro código hace, pudiendo incluso llevarnos a confusión si les hacemos caso.

Podrían ser directamente eliminados o reemplazados si utilizamos mejores nombres para los símbolos (variables, constantes, clases, métodos, funciones…). En algunos casos, plantearnos eliminar comentarios puede llevarnos a reflexionar sobre nuevas formas de implementar algo.

Por otro lado, los comentarios que sí pueden permanecer suelen referirse a aspectos que no podemos expresar fácilmente con código, como puede ser explicar los motivos para hacer algo de una forma concreta.

Mejorando los nombres

El segundo capítulo de la guía del refactor cotidiano trata de los nombres y cómo mejorarlos. Probablemente en ningún lugar como el código los nombres configuran la realidad. Escribir código implica establecer decenas de nombres cada día, para identificar conceptos y procesos. Una mala elección de nombre puede condicionar nuestra forma de ver un problema de negocio. Un nombre ambiguo puede llevarnos a entrar en un callejón sin salida, ahora o en un futuro no muy lejano. Pero un nombre bien escogido puede ahorrarnos tiempo, dinero y dificultades.

Símbolos con nombres

Un trozo de código debería poder leerse como una especie de narrativa, en la cual cada palabra expresase de forma unívoca un significado. También de forma ubicua y coherente, es decir, que el mismo símbolo debería representar el mismo concepto en todas partes del código.

¿Cuándo refactorizar nombres?

La regla de oro es muy sencilla: cada vez que al leer una línea de código tenemos que pararnos a pensar qué está diciendo lo más probable sea que deberíamos cambiar algún nombre.

Este es un ejemplo de un código en el que nos encontramos con unos cuantos problemas de nombres, algunos son evidentes y otros no tanto:

```
class PriceCalculator {
    var Discount $rate;

    public price(): float
    {
        //...
        $rate = $this->getRate($product);
        $tax = $this->taxRepository->byProduct($pr\
oduct);

        $amount = $product->basePrice();

        $amount = $this->calculate($amount, $rate,\
 $tax);

        $amount = $amount - $this->calculateDiscou\
nt($amount);

        return $amount;
    }

    public discount(float $rate): void
    {
         $this->rate = $rate;
    }

```

```
27      private function calculateDiscount(float $pri\
28  ce): float
29      {
30          $discount = $price * $this->rate;
31
32          return $discount;
33      }
34
35      private function calculate(int $amount, float\
36   $rate, float $tax): float
37      {
38          // some complex calculation
39      }
40  }
```

Por supuesto, en este ejemplo hay algunos errores más aparte de los nombres. Pero hoy solo nos ocuparemos de estos. Vamos por partes.

Nombres demasiado genéricos

Los nombres demasiado genéricos requieren el esfuerzo de interpretar el caso concreto en que se están aplicando. Además, en un plano más práctico, resulta difícil localizar una aparición específica del mismo que tenga el significado deseado.

¿De dónde vienen los nombres demasiado genéricos? Normalmente, vienen de estadios iniciales del código, en los que probablemente bastaba con ese término genérico para designar un concepto. Con el tiempo, ese concepto evoluciona y se ramifica a medida que el conocimiento de negocio avanza, pero el código puede que no lo haya hecho al mismo ritmo,

con lo que llega un momento en que este no es reflejo del conocimiento actual que tenemos del negocio.

Calculate... what? Exactamente, ¿qué estamos calculando aquí? El código no lo refleja. Podría ocurrir, por ejemplo, que `$rate` fuese algún tipo de comisión, `$tax` resulta bastante obvio y `$amount` parece claro que es algo así como el precio de tarifa de algún producto o servicio, sea lo que sea que vende esta empresa. Es muy posible que este método lo que haga sea calcular el precio final para el consumidor del producto. ¿Por qué no declararlo de forma explícita?

```
1  public price(): float
2  {
3      //...
4      $rate = $this->getRate($product);
5      $tax = $this->taxRepository->byProduct($produ\
6  ct);
7
8      $amount = $product->basePrice();
9
10     $amount = $this->calculateFinalConsumerPrice(\
11 $amount, $rate, $tax);
12
13     $amount = $amount - $this->calculateDiscount(\
14 $amount);
15
16     return $amount;
17 }
18
19 private function calculateFinalConsumerPrice(int \
20 $amount, float $rate, float $tax): float
```

```
{
    // some complex calculation
}
```

Vayamos ahora con `$rate`. Tras hablarlo con negocio, hemos llegado a la conclusión de que representa el porcentaje de comisión que corresponde al comercial que ha realizado la venta. Podría pasar a llamarse `$commissionRate`, al igual que el método del cual la obtenemos.

```
public price(): float
{
    //...
    $commissionRate = $this->getCommissionRate($p\
roduct);
    $tax = $this->taxRepository->byProduct($produ\
ct);

    $amount = $product->basePrice();

    $amount = $this->calculateFinalConsumerPrice(\
$amount, $commissionRate, $tax);

    $amount = $amount - $this->calculateDiscount(\
$amount);

    return $amount;
}

private function calculateFinalConsumerPrice(int \
```

```
$amount, float $commissionRate, float $tax): floa\
t
{
    // some complex calculation
}
```

Además, esto era bastante necesario, porque resulta que la clase tiene otro `$rate`, que es una propiedad que, teniendo el mismo nombre, representa algo completamente distinto, como es un descuento. Tanto `$commissionRate` como `$rate` son ratios (proporciones o porcentajes), pero el hecho de que sean el mismo concepto matemático (ratio o proporción), no implica que sean el mismo concepto de negocio. Por supuesto, necesitamos mayor precisión también aquí:

```
var float $discountRate;

//...

public discount(float $discountRate): void
{
    $this->discountRate = $discountRate;
}

private function calculateDiscount(float $price):\
 float
{
    $discount = $price * $this->discountRate;

    return $discount;
}
```

`$tax` puede mejorar también. Pero, ¿qué nos cuesta hacerlo explícito si queremos decir que se trata del IVA?

```
public price(): float
{
    //...
    $commissionRate = $this->getCommissionRate($p\
roduct);
    $vat = $this->taxRepository->byProduct($produ\
ct);

    $amount = $product->basePrice();

    $amount = $this->calculateFinalConsumerPrice(\
$amount, $commissionRate, $vat);

    $amount = $amount - $this->calculateDiscount(\
$amount);

    return $amount;
}

private function calculateFinalConsumerPrice(
    int $amount,
    float $commissionRate,
    float $vat
): float {
    // some complex calculation
}
```

Nombres reutilizados en el mismo *scope*

Nunca se deben reutilizar nombres en el mismo *scope*, o sea, dentro de un mismo bloque, función o clase, para representar cosas distintas porque nos lleva a confusión. Los lenguajes más estrictos son capaces de evitar que introduzcas valores de distinto tipo en la misma variable, mientras que otros no te dejarán reasignarla una vez inicializada, pero existen muchos casos en que esa reasignación es posible, al menos si no cambia el tipo de dato.

En nuestro ejemplo, la variable `$amount` es asignada tres veces y utilizada con significados diferentes.

Voy a eliminar parte del código para que te puedas fijar en algo aparentemente inocente:

```
1  public price(): float
2  {
3      //...
4      $amount = $product->basePrice();
5
6      $amount = $this->calculateFinalConsumerPrice(\
7  $amount, $commissionRate, $vat);
8
9      //...
10
11     return $amount;
12 }
13
14 private function calculateFinalConsumerPrice(
15     int $amount,
16     float $commissionRate,
```

```
    float $vat
): float {
    // some complex calculation
}
```

¿Lo has pillado? El método `calculateFinalConsumerPrice` espera que `$amount` sea un entero, mientras que devuelve un valor de coma flotante que se asigna de nuevo a `$amount`. ¿Qué está pasando aquí?

Pues, por ejemplo, podría estar pasando que el precio que contiene `$product`, esté expresado en céntimos por la razón que sea, mientras que el precio final se va a expresar en euros. De nuevo, el conflicto se puede resolver siendo explícitos sobre lo que la variable realmente contiene o el parámetro exige:

```
public price(): float
{
    //...
    $amountInCents = $product->basePrice();

    $amountInEuros = $this->calculateFinalConsume\
rPrice($amountInCents, $commissionRate, $vat);

    //...

    return $amountInEuros;
}

private function calculateFinalConsumerPrice(
    int $amountInCents,
    float $commissionRate,
```

```
    float $vat
): float {
    // some complex calculation
}
```

Por otro lado, la secuencia de transformaciones que sufre `$amountInEuros` puede transmitir mensajes confusos. Por un lado, refleja que es un precio base que se transforma por diversas razones (comisiones, impuestos, descuentos), pero, por otro lado, podría tener diversos significados de negocio que en un momento dado necesitaríamos discriminar.

En el primer caso, esta representación puede ser mucho más descriptiva de lo que realmente pasa:

```
public price(): float
{
    //...
    $commissionRate = $this->getCommissionRate($p\
roduct);
    $vat = $this->taxRepository->byProduct($produ\
ct);

    $amountInCents = $product->basePrice();

    $amountInEuros = $this->calculateFinalConsume\
rPrice($amountInCents, $commissionRate, $vat);

    $amountInEuros -= $this->calculateDiscount($a\
mountInEuros);

```

```
17      return $amountInEuros;
18  }
```

Por ejemplo, podríamos necesitar discriminar el precio antes y después de impuestos. O el total de la comisión que se lleva el comercial, porque se han convertido en cuestiones importantes del negocio:

```
1   public price(): float
2   {
3       //...
4       $commissionRate = $this->getCommissionRate($p\
5   roduct);
6       $vat = $this->taxRepository->byProduct($produ\
7   ct);
8
9       $amountInCentsBeforeTaxes = $product->basePri\
10  ce();
11
12      $commission = $this->calculateCommission($amo\
13  untInCentsBeforeTaxes, $commissionRate);
14
15      $amountInEuros = $this->calculateFinalConsume\
16  rPrice($amountInCentsBeforeTaxes, $commissionRate\
17  , $vat);
18
19      $amountInEuros -= $this->calculateDiscount($a\
20  mountInEuros);
21
22      return $amountInEuros;
```

```
23  }
```

El refactor va aclarando, por una parte, conceptos de negocio, pero también nos permite descubrir que tenemos problemas más profundos en el código.

Por ejemplo, que nos vendría bien utilizar un **ValueObject** para representar el precio, como `Money`, incluso aunque al final devolvamos un float para no cambiar la interfaz pública:

```
1   public price(): float
2   {
3       //...
4       $commissionRate = $this->getCommissionRate($p\
5   roduct);
6       $vat = $this->taxRepository->byProduct($produ\
7   ct);
8
9       $amountBeforeTaxes = Money::fromCents($produc\
10  t->basePrice(), 'EUR');
11
12      $amount = $amountBeforeTaxes
13          ->addRate($commissionRate)
14          ->addRate($vat);
15
16      $discountedAmount = $amount->substractRate($t\
17  his->discountRate());
18
19      return $discountedAmount->amount();
20  }
```

Tipo de palabra inadecuada

Los símbolos que, de algún modo, contradicen el concepto que representan son más difíciles de procesar, generalmente porque provocan una expectativa que no se cumple y, por tanto, debemos re-evaluar lo que estamos leyendo.

Así, una acción debería representarse siempre mediante un verbo.

Y un concepto, siempre mediante un sustantivo.

A su vez, nunca nos sobran los adjetivos para precisar el significado del sustantivo, por lo que los nombres compuestos nos ayudan a representar con mayor precisión las cosas.

Volvamos al ejemplo. `PriceCalculator` parece un buen nombre. Es un sustantivo, por lo que se deduce que es un actor que hace algo. Veámosla como *interface*:

```
1  interface PriceCalculator {
2      public price(Product $product): float;
3      public discount(float $rate): float;
4  }
```

Obviamente, este refactor es un poco más arriesgado. Vamos a tocar una interfaz pública, pero también es verdad que con los IDE modernos este tipo de cambios es razonablemente seguro.

Vamos por la más evidente. El método `discount` en realidad nos sirve para asignar un descuento aplicable a la siguiente operación `price`. Estamos usando un sustantivo para indicar una acción. La opción más inmediata:

```
interface PriceCalculator {
    public price(Product $product): float;
    public setDiscount(float $rate): float;
}
```

Está mejor, pero también podemos ser más fieles al lenguaje de negocio. De hecho, `set` tiene un significado demasiado genérico y no dice realmente nada:

```
interface PriceCalculator {
    public price(Product $product): float;
    public applyDiscount(float $rate): float;
}
```

En cambio, `applyDiscount` es una clara acción de negocio y no deja muchas dudas en cuanto al significado. Pero todavía podríamos aportar un poco más de precisión, aunque el nombre del parámetro es `$rate`, nunca se sabe cómo se va a utilizar:

```
interface PriceCalculator {
    public price(Product $product): float;
    public applyDiscountRate(float $rate): float;
}
```

Ahora, sí.

¿Y qué decir de `price`? De nuevo, es un sustantivo que representa una acción, por lo que podríamos cambiarlo.

Pero antes… Volvamos un momento a la clase. `PriceCalculator`, ¿es un actor o una acción? A veces tendemos a ver los objetos como representaciones de objetos del *mundo real*. Sin embargo, podemos representar acciones y otros conceptos con objetos

en el código. Esta forma de verlo puede cambiar por completo nuestra manera de hacer las cosas.

Supongamos entonces, que consideramos que `PriceCalculator` no es una *cosa*, sino una *acción*:

```
interface CalculatePrice {
    public price(Product $product): float;
    public applyDiscountRate(float $rate): float;
}
```

Tal y como está ahora, expresar ciertas cosas resulta extraño:

```
$calculatePrice = new CalculatePrice();

$calculatePrice->applyDiscountRate($rate);
$calculatePrice->price($product);
```

Pero podemos imaginarlo de otra forma mucho más fluida:

```
$calculatePrice = new CalculatePrice();

$calculatePrice->applyingDiscountRate($rate);
$calculatePrice->finalForProduct($product);
```

Lo que nos deja con esta interfaz:

```
interface CalculatePrice {
    public finalForProduct(Product $product): flo\
at;
    public applyingDiscountRate(float $rate): flo\
at;
}
```

Números mágicos

En este caso no se trata estrictamente de refactorizar nombres, sino de bautizar elementos que están presentes en nuestro código en forma de valores abstractos que tienen un valor de negocio que no ha sido hecho explícito.

Poniéndoles un nombre, lo hacemos. Antes:

```
$vatAmount = $amountBeforeTaxes * .21;
```

Después:

```
$vatAmount = $amountBeforeTaxes * self::VAT_RATE;
```

Convertir estos valores en constantes con nombre hace que su significado de negocio esté presente, sin tener que preocuparse de interpretarlo. Además, esto los hace reutilizables a lo largo de todo el código, lo que añade un plus de coherencia.

Así que, cada vez que encuentres uno de estos valores, hazte un favor y reemplázalo por una constante. Por ejemplo, los naturalmente ilegibles patrones de expresiones regulares:

```
$isValidNif = preg_match('/^[0-9XYZ]\d{7}[^\dUIOÑ\
]$/', $nif);

// vs

$isValidNif = preg_match(Nif::VALID_NIF_PATTERN, \
$nif);
```

O los patrones de formato para todo tipo de mensajes:

```
$mensaje = sprintf('¿Enviar un mensaje a %s en la\
 dirección %s?', $user->username(), $user->email(\
));

$mensaje = sprintf(self::CONFIRM_SEND_EMAIL_MESSA\
GE, $user->username(), $user->email());
```

Nombres técnicos

Personalmente me gustan poco los nombres técnicos formando parte de los nombres de variables, clases, interfaces, etc. De hecho, creo que en muchas ocasiones condicionan tanto el *naming*, que favorecen la creación de malos nombres.

Ya he hablado del problema de entender que los objetos en programación tienen que ser representaciones de objetos del mundo real. Esa forma de pensar nos lleva a ver todos los objetos como actores que hacen algo, cuando muchas veces son acciones.

En ocasiones, es verdad que tenemos que representar ciertas operaciones técnicas, que no todo va a ser negocio, pero eso no

quiere decir que no hagamos las cosas de una manera elegante. Por ejemplo:

```
interface BookTransformer
{
    public function transformToJson(Book $book): \
string;
    public function transformFromJson(string $boo\
kDto): Book;
}

// vs

interface TransformBook
{
    public function toJson(Book $book): string;
    public function fromJson(string $bookDto): Bo\
ok;
}
```

En cambio, en el dominio me choca ver cosas como:

```
class BookWasPrintedEvent implements DomainEvent
{
}

// vs

class BookWasPrinted implements DomainEvent
```

```
8 {
9 }
```

Ya que el uso del verbo en pasado debería ser suficiente para entender de un vistazo que está hablando de un event (un mensaje que indica que algo ha ocurrido).

Es cierto que incluir algunos *apellidos técnicos* a nuestros nombres puede ayudarnos a localizar cosas en el IDE. Pero hay que recordar que no programamos para un IDE.

Refactor de nombres

En general, gracias a las capacidades de refactor de los IDE o incluso del Buscar/Reemplazar en proyectos, realizar refactors de nombres es bastante seguro.

Variables locales en métodos y funciones. Cambiarlas no supone ningún problema, pues no afectan a nada que ocurra fuera de su ámbito.

Propiedades y métodos privados en clases. Tampoco suponen ningún problema al no afectar a nada externo a la clase.

Interfaces públicas. Aunque es más delicado, los IDE modernos deberían ayudarnos a realizarlos sin mayores problemas. La mayor dificultad me la he encontrado al cambiar nombres de clases, puesto que el IDE aunque localiza y cambia correctamente sus usos, no siempre identifica objetos relacionados, como los tests.

El coste de un mal nombre

Imaginemos un sistema de gestión de bibliotecas que, inicialmente, se creó para gestionar libros. Simplificando muchísimo, aquí tenemos un concepto clave del negocio:

```
class Book
{
    private $id;
    private $title;
    private $author;
    private $editor;
    private $year;
    private $city;
}
```

Con el tiempo la biblioteca pasó a gestionar revistas. Las revistas tienen número, pero tal vez en su momento se pensó que no sería necesario desarrollar una especialización:

```
class Book
{
    private $id;
    private $title;
    private $author;
    private $editor;
    private $year;
    private $city;
    private $issue;
}
```

Y aquí comienza un desastre que solo se detecta mucho tiempo después y que puede suponer una sangría, quizá lenta pero constante, de tiempo, recursos y, en último término, dinero para los equipos y empresas.

La modificación de la clase `Book` hizo que esta pasara a representar dos conceptos distintos, pero quizá se consideró que era una ambigüedad manejable: un compromiso aceptable.

Claro que la biblioteca siguió evolucionando y con el avance tecnológico comenzó a introducir nuevos tipos de objetos, como CD, DVD, libros electrónicos, y un largo etcétera. En este punto, el conocimiento que maneja negocio y su representación en el código se han alejado tanto que el código se ha convertido en una pesadilla: ¿cómo sabemos si `Book` se refiere a un libro físico, a uno electrónico, a una película en DVD, a un juego en CD? Solo lo podemos saber examinando el contenido de cada objeto `Book`. Es decir: el código nos está obligando a pararnos a pensar para entenderlo. Necesitamos refactorizar y reescribir.

Es cierto que, dejando aparte el contenido, todos los objetos culturales conservados en una biblioteca comparten ese carácter de objeto cultural o soporte de contenidos. `CulturalObject` se nos antoja un nombre demasiado forzado, pero `Media` resulta bastante manejable:

```
1  class Media
2  {
3      private $id;
4      private $signature;
5      private $registeredSince;
6      private $status;
7  }
```

De `Media` que representaría a los soportes de contenidos archivados en la biblioteca y que contendría propiedades como un número de registro (el *id*), la signatura topográfica (que nos comunica su ubicación física) y otros detalles relacionados con la actividad de archivo, préstamo, etcétera.

Pero esa clase tendría especializaciones que representan tipos de medios específicos, con sus propiedades y comportamientos propios.

```
1  class Book extends Media
2  {
3  }
4
5  class Review extends Media
6  {
7  }
8
9  class ElectronicBook extends Media
10 {
11 }
12
13 class Movie extends Media
14 {
15 }
```

Podríamos desarrollar más el conocimiento de negocio en el código, añadiendo interfaces. Por ejemplo, la gestión del préstamo:

```
interface Lendable
{
    public function lend(User $user): void;
    public function return(DateTimeInterface $dat\
e): void;
}
```

Pero el resumen es que el hecho de no haber ido reflejando la evolución del conocimiento del negocio en el código nos lleva a tener un sobre-coste en forma de:

- El tiempo y recursos necesarios para actualizar el desarrollo a través de reescrituras.
- El tiempo y recursos necesarios para mantener el software cuando surgen problemas derivados de la mala representación del conocimiento.
- Las pérdidas por no ingresos debidos a la dificultad del software de adaptarse a las necesidades cambiantes del negocio.

Por esto, preocúpate por poner buenos nombres y mantenerlos al día. Va en ello tu salario.

Acondiciona las condicionales

Este capítulo de la *Guía del refactor cotidiano* trata sobre cómo mejorar las estructuras condicionales.

Es bastante obvio que si hay algo que añade complejidad a un software es la **toma de decisiones** y, por tanto, las estructuras condicionales con las que la expresamos.

Estas estructuras pueden introducir dificultades de comprensión debido a varias razones:

- **La complejidad de las expresiones evaluadas**, sobre todo cuando se combinan mediante operadores lógicos tres o más condiciones.
- **La anidación de estructuras condicionales** y la concatenación de condicionales mediante `else`.
- **El desequilibrio entre las ramas** en las que una rama tiene unas pocas líneas frente a la otra que esconde su propia complejidad.

¿Cuándo refactorizar condicionales?

En general, como regla práctica, hay que refactorizar condicionales cuando su lectura no nos deja claro cuál es su significado. Esto se aplica en las dos partes de la estructura:

- **La expresión condicional**: qué define lo que tiene que pasar para que el flujo se dirija por una o por otra rama.
- **Las ramas**: las diferentes acciones que se deben ejecutar en caso de cumplirse o no la condición.

Otras reglas prácticas que podemos aplicar son:

Aplanar niveles de indentación: cuanto menos anidamiento en el código, más fácil de leer es porque indica que no estamos mezclando niveles de abstracción.

Eliminar `else`: en muchos casos, es posible eliminar ramas alternativas, bien directamente, bien encapsulando toda la estructura en un método o función.

La rama corta primero

Si una estructura condicional nos lleva por una rama muy corta en caso de cumplirse y por una muy larga en el caso contrario, se recomienda que la rama corta sea la primera, para evitar que pase desapercibida.

Por ejemplo, este fragmento tan feo:

```
1  if ($selectedPaymentMethod == null) {
2      $logger = Logger::getInstance();
3      $logger->debug("Medio de pago desconocido");
4      if ($order->getDestinationCountry() == Countr\
5  y::FRANCE && $order->id() < 745) {
6          $paymentMethod = PaymentTypes::PAYPAL;
7      }
8  } else {
9      $paymentMethod = $selectedPaymentMethod->getP\
```

```
aymentMethodType()->getPaymentMethodTypeId();
}
```

Podría reescribirse así:

```
if (null !== $selectedPaymentMethod) {
    $paymentMethod = $selectedPaymentMethod->getP\
aymentMethodType()->getPaymentMethodTypeId();
} else {
    $logger = Logger::getInstance();
    $logger->debug("Medio de pago desconocido");
    if ($order->getDestinationCountry() == Countr\
y::FRANCE && $order->id() < 745) {
        $paymentMethod = PaymentTypes::PAYPAL;
    }
}
```

Return early

Si estamos dentro de una función o método y podemos hacer el retorno desde dentro de una rama es preferible hacerlo. Con eso podemos evitar el *else* y hacer que el código vuelva al nivel de indentación anterior, mejor si es el primero, lo que facilitará la lectura.

Imaginemos que tras el código anterior tenemos un `return`, no hace falta que sea inmediatamente después:

```
if (null !== $selectedPaymentMethod) {
    $paymentMethod = $selectedPaymentMethod->getP\
aymentMethodType()->getPaymentMethodTypeId();
} else {
    $logger = Logger::getInstance();
    $logger->debug("Medio de pago desconocido");
    if ($order->getDestinationCountry() == Countr\
y::FRANCE && $order->id() < 745) {
        $paymentMethod = PaymentTypes::PAYPAL;
    }
}

// Some more code to get a value for $paymentMeth\
od

return $paymentMethod;
```

En realidad, en la primera rama ya podríamos volver sin problemas, lo que nos permite eliminar la cláusula `else`, reduciendo la indentación del código.

```
if (null !== $selectedPaymentMethod) {
    $paymentMethod = $selectedPaymentMethod->getP\
aymentMethodType()->getPaymentMethodTypeId();

    return $paymentMethod;
}

$logger = Logger::getInstance();
$logger->debug("Medio de pago desconocido");
```

```
10
11 if ($order->getDestinationCountry() == Country::F\
12 RANCE && $order->id() < 745) {
13     $paymentMethod = PaymentTypes::PAYPAL;
14 }
15
16 // Some more code to get a value for $paymentMeth\
17 od
18
19 return $paymentMethod;
```

Además, no hace falta crear ni poblar una variable temporal, gracias a lo cual podemos devolver directamente la respuesta obtenida, aplicando lo mismo a la condicional que podemos ver al final:

```
1  if (null !== $selectedPaymentMethod) {
2      return $selectedPaymentMethod->getPaymentMeth\
3  odType()->getPaymentMethodTypeId();
4  }
5
6  $logger = Logger::getInstance();
7  $logger->debug("Medio de pago desconocido");
8
9  if ($order->getDestinationCountry() == Country::F\
10 RANCE && $order->id() < 745) {
11     return PaymentTypes::PAYPAL;
12 }
13
14 // Some more code to get a value for $paymentMeth\
```

```
15 od
```

Un uso habitual de esta técnica es la de tratar casos particulares o que sean obvios en los primeros pasos del algoritmo, volviendo al flujo principal cuanto antes, de modo que el algoritmo solo recibe aquellos casos a los que se aplica realmente.

Cláusulas de guarda

En muchas ocasiones, cuando los datos tienen que ser validados antes de operar con ellos, podemos encapsular esas condiciones que dan lugar a excepciones en forma de cláusulas de guarda. Estas cláusulas de guarda, también se conocen como aserciones, o precondiciones. Si los parámetros no las cumplen, el método o función falla lanzando excepciones.

```
1     if ($parameter > 100 || $parameter < 0) {
2         throw new OutOfRangeException(sprintf('Pa\
3 rameter should be between 0 and 100 (inc), %s pro\
4 vided.', $parameter));
5     }
6
7 // further processing
```

Extraemos toda la estructura a un método privado:

```
$this->checkTheParameterIsInRange($parameter);

// further processing

private function checkTheParameterIsInRange(int $\
parameter)
{
    if ($parameter > 100 || $parameter < 0) {
        throw new OutOfRangeException(sprintf('Pa\
rameter should be between 0 and 100 (inc), %s pro\
vided.', $parameter));
    }
}
```

La lógica bajo este tipo de cláusulas es que si no salta ninguna excepción, quiere decir que `$parameter` ha superado todas las validaciones y lo puedes usar con confianza. La ventaja es que las reglas de validación definidas con estas técnicas resultan muy expresivas, ocultando los detalles técnicos en los métodos extraídos.

Una alternativa es usar una librería de aserciones, lo que nos permite hacer lo mismo de una forma más limpia y normalizada. Si la aserción no se cumple, se tirará una excepción:

```
Assert::betweenExclusive($parameter, 0, 100)
```

Una limitación de las aserciones que debemos tener en cuenta es no usarlas para control de flujo. Esto es, las aserciones fallan con una excepción, interrumpiendo la ejecución del programa, que así puede comunicar al módulo llamante una circunstancia que impide continuar. En el caso de necesitar una alternativa

si el parámetro no cumple las condiciones, utilizaremos condicionales.

Aquí lo podemos ver, si el parámetro excede los límites queremos que se ajuste al límite que ha superado:

```
$parameter = $this->checkTheParameterIsInRange($p\
arameter);

// further processing

private function checkTheParameterIsInRange(int $\
parameter)
{
    if ($parameter > 100) {
        return 100;
    }

    if ($parameter < 0) {
        return 0;
    }

    return $parameter;
}
```

Preferir condiciones afirmativas

Diversos estudios han mostrado que las frases afirmativas son más fáciles de entender que las negativas, por lo que siempre que sea posible deberíamos intentar convertir la condición

en afirmativa bien sea invirtiéndola, bien encapsulándola de modo que se exprese de manera afirmativa.

Con frecuencia, además, la sintaxis de la negación puede hacerlas poco visibles:

```
1 if (!$selectedPaymentMethod) {
2     return $selectedPaymentMethod->getPaymentMeth\
3 odType()->getPaymentMethodTypeId();
4 }
```

En uno de los ejemplos anteriores habíamos llegado a la siguiente construcción, que es una condición negada especialmente difícil de leer:

```
1 if (null !== $selectedPaymentMethod) {
2     return $selectedPaymentMethod->getPaymentMeth\
3 odType()->getPaymentMethodTypeId();
4 }
```

Nosotros lo que queremos es devolver el método de pago si es que tenemos uno seleccionado:

```
1 if ($selectedPaymentMethod) {
2     return $selectedPaymentMethod->getPaymentMeth\
3 odType()->getPaymentMethodTypeId();
4 }
```

Una forma alternativa, si la condición es compleja o simplemente difícil de entender tal cual es encapsularla en un método:

```
if ($this->userHasSelectedAPaymentMethod($selecte\
dPaymentMethod)) {
    return $selectedPaymentMethod->getPaymentMeth\
odType()->getPaymentMethodTypeId();
}

function userHasSelectedAPaymentMethod($selectedP\
aymentMethod)
{
    return null !== $selectedPaymentMethod;
}
```

Encapsula expresiones complejas en métodos o funciones

La idea es encapsular expresiones condicionales complejas en funciones o métodos, de modo que su nombre describa el significado de la expresión condicional, manteniendo ocultos los detalles *escabrosos* de la misma. Esto puede hacerse de forma global o por partes.

Justo en el apartado anterior hemos visto un ejemplo de esto mismo, haciendo explícito el significado de una expresión condicional difícil de leer.

Veamos otro caso en el mismo ejemplo, la extraña condicional:

```
if ($order->getDestinationCountry() == Country::F\
RANCE && $order->id() < 745) {
    return PaymentTypes::PAYPAL;
}

// Some more code to get a value for $paymentMeth\
od
```

Podría ser un poco más explicativa encapsulada en un método:

```
if (legacyOrdersWithDestinationFrance($order)) {
    return PaymentTypes::PAYPAL;
}

// Some more code to get a value for $paymentMeth\
od

private function legacyOrdersWithDestinationFranc\
e($order)
{
    return $order->getDestinationCountry() == Cou\
ntry::FRANCE && $order->id() < 745;
}
```

Esto deja el bloque de esta manera:

```
if ($selectedPaymentMethod) {
    return $selectedPaymentMethod->getPaymentMeth\
odType()->getPaymentMethodTypeId();
}

$logger = Logger::getInstance();
$logger->debug("Medio de pago desconocido");

if (legacyOrdersWithDestinationFrance($order)) {
    return PaymentTypes::PAYPAL;
}

// Some more code to get a value for $paymentMeth\
od

private function legacyOrdersWithDestinationFranc\
e($order)
{
    return $order->getDestinationCountry() == Cou\
ntry::FRANCE && $order->id() < 745;
}
```

Del singleton que tenemos por ahí no hablaremos en esta ocasión.

Encapsula ramas en métodos o funciones

Consiste en encapsular todo el bloque de código de cada rama de ejecución en su propio método, de modo que el nombre nos indique qué hace. Esto nos deja las ramas de la estructura condicional al mismo nivel y expresando lo que hacen de manera explícita y global. En los métodos extraídos podemos seguir aplicando refactors progresivos hasta que ya no sea necesario.

Este fragmento de código, que está bastante limpio, podría clarificarse un poco, encapsulando tanto las condiciones como la rama:

```
if ($productStatus == OrderStatuses::PROVIDER_PEN\
DING ||
    $productStatus == OrderStatuses::PENDING ||
    $productStatus == OrderStatuses::WAITING_FOR_\
PAYMENT
) {
    if ($paymentMethod == PaymentTypes::BANK_TRAN\
SFER) {
        return 'pendiente de transferencia';
    }
    if ($paymentMethod == PaymentTypes::PAYPAL ||\
 $paymentMethod == PaymentTypes::CREDIT_CARD) {
        return 'pago a crédito';
    }
    if ($this->paymentMethods->hasSelectedDebitCa\
rd()) {
        return 'pago a débito';
```

```
18      }
19      if (!$this->paymentMethods->requiresAuthoriza\
20  tion()) {
21          return 'pago no requiere autorización';
22      }
23  }
```

Veamos como:

```
1   if ($this->productIsInPendingStatus($productStatu\
2   s)) {
3       return $this->reportForProductInPendingStatus\
4   ($paymentMethod);
5   }
6
7   private function productIsInPendingStatus($produc\
8   tStatus)
9   {
10      return ($productStatus == OrderStatuses::PROV\
11  IDER_PENDING ||
12      $productStatus == OrderStatuses::PENDING ||
13      $productStatus == OrderStatuses::WAITING_FOR_\
14  PAYMENT);
15  }
16
17  private function reportForProductInPendingStatus(\
18  paymentMethod)
19  {
20      if ($paymentMethod == PaymentTypes::BANK_TRAN\
21  SFER) {
```

```
22          return 'pendiente de transferencia';
23      }
24      if ($paymentMethod == PaymentTypes::PAYPAL ||\
25   $paymentMethod == PaymentTypes::CREDIT_CARD) {
26          return 'pago a crédito';
27      }
28      if ($this->paymentMethods->hasSelectedDebitCa\
29  rd()) {
30          return 'pago a débito';
31      }
32      if (!$this->paymentMethods->requiresAuthoriza\
33  tion()) {
34          return 'pago no requiere autorización';
35      }
36  }
```

De ese modo, la complejidad queda oculta en los métodos y el cuerpo principal se entiende fácilmente. Ya es cuestión nuestra si necesitamos seguir el refactor dentro de los métodos privados que acabamos de crear.

Equalize branches

Si hacemos esto en todas las ramas de una condicional o de un switch las dejaremos todas al mismo nivel, lo que facilita su lectura.

Reemplaza `if…elseif` sucesivos con `switch`

En muchos casos, sucesiones de `if` o `if…else` quedarán mejor expresados mediante una estructura `switch`. Por ejemplo, siguiendo con el ejemplo anterior, este método que hemos extraído:

```
1  private function reportForProductInPendingStatus(\
2  paymentMethod)
3  {
4      if ($paymentMethod == PaymentTypes::BANK_TRAN\
5  SFER) {
6          return 'pendiente de transferencia';
7      }
8      if ($paymentMethod == PaymentTypes::PAYPAL ||\
9   $paymentMethod == PaymentTypes::CREDIT_CARD) {
10         return 'pago a crédito';
11     }
12     if ($this->paymentMethods->hasSelectedDebitCa\
13 rd()) {
14         return 'pago a débito';
15     }
16     if (!$this->paymentMethods->requiresAuthoriza\
17 tion()) {
18         return 'pago no requiere autorización';
19     }
20 }
```

Podría convertirse en algo así:

```
private function reportForProductInPendingStatus(\
paymentMethod)
{
    switch $paymentMethod {
        case PaymentTypes::BANK_TRANSFER:
            return 'pendiente de transferencia';
        case PaymentTypes::PAYPAL:
        case PaymentTypes::CREDIT_CARD:
            return 'pago a crédito';
    }

    if ($this->paymentMethods->hasSelectedDebitCa\
rd()) {
        return 'pago a débito';
    }
    if (!$this->paymentMethods->requiresAuthoriza\
tion()) {
        return 'pago no requiere autorización';
    }
}
```

Sustituir if por el operador ternario

A veces, un operador ternario puede ser más legible que una condicional:

```
function selectElement(Criteria $criteria, Desira\
bility $desirability)
{
    $found = false;

    $elements = $this->getElements($criteria);

    foreach($elements as $element) {
        if (!$found && $this->isDesired($element,\
 $desirability)) {
            $result = $element;
            $found = true;
        }
    }
    if (!$found) {
        $result = null;
    }

    return $result;
}
```

Realmente las últimas líneas pueden expresarse en una sola y queda más claro:

```
function selectElement(Criteria $criteria, Desira\
bility $desirability)
{
    $found = false;

    $elements = $this->getElements($criteria);

    foreach($elements as $element) {
        if (!$found && $this->isDesired($element,\
 $desirability)) {
            $result = $element;
            $found = true;
        }
    }

    return $found ? $result : null;
}
```

El operador ternario tiene sus problemas, pero, en general, es una buena solución cuando queremos expresar un cálculo que se resuelve de dos maneras según una condición. Eso sí: nunca anides operadores ternarios porque su lectura entonces se complica enormemente.

Resumen del capítulo

Las expresiones y estructuras condicionales pueden hacer que seguir el flujo de un código sea especialmente difícil, particularmente cuando están anidadas o son muy complejas. Mediante técnicas de extracción podemos simplificarlas, aplanarlas y

hacerlas más expresivas.

Sustituye escalares por objetos

PHP viene de serie con un conjunto de tipos de datos básicos que denominamos escalares: bool, int, float, string…, que utilizamos para representar cosas y operar con ellas. La parte mala es que son tipos genéricos y, a veces, necesitaríamos algo con más significado.

Lo ideal sería poder crear nuestros propios tipos, aptos para el dominio en el que estemos trabajando e incluyendo sus propias restricciones y propiedades. Además, podrían encapsular las operaciones que les sean necesarias. ¿Te suena el concepto? Estamos hablando de **Value Object**.

Los Value Objects son objetos que representan algún concepto importante en el dominio. Ya hemos hablado un montón de veces de ellos en el blog, por lo que simplemente haré un resumen. Puedes encontrar más detalles y ejemplos en este artículo de Dani Tomé[3].

En resumen, los Value Objects:

- Representan conceptos importantes o interesantes del dominio, entendido como el dominio de conocimiento que toca el código que estamos implementando o estudiando.
- Siempre son creados consistentes, de modo que si obtienes una instancia puedes tener la seguridad de que

[3] https://danitome24.github.io/2018-11-19/usando-value-objects-con-php

es válida. De otro modo, no se crean y se lanza una excepción.
- Los objetos nos interesan por su valor, no por su identidad, por lo que tienen que tener alguna forma de chequear esa igualdad.
- Son inmutables, o sea, su valor no puede cambiar durante su ciclo de vida. En caso de que tengan métodos *mutators*, estos devolverán una nueva instancia de la clase con el valor modificado.
- Encapsulan comportamientos. Los buenos Value Objects atraen y encapsulan comportamientos que pueden ser utilizados por el resto del código.

Los Value Objects pueden ser genéricos y reutilizables, como Money, o muy específicos de un dominio.

Refactorizar a Value Objects

Refactorizar a Value Objects puede ser una tarea de bastante calado, ya que implica crear nuevas clases y utilizarlas en diversos puntos del código. Ahora bien, este proceso puede hacerse de forma bastante gradual. Ten en cuenta que:

- Los Value Objects no tienen dependencias, para crearlos solo necesitas tipos escalares u otros Value Objects.
- Los Value Objects se pueden instanciar allí donde los necesites, son *newables*.
- Normalmente, tendrás métodos para convertir los Value Objects a escalares, de modo que puedas utilizar sus valores con código que no puedes modificar.

Los Value Objects aportan varias ventajas:

- Al encapsular su validación tendrás objetos con valores adecuados que puedes usar libremente sin necesidad de validar constantemente.
- Aportarán significado a tu código, siempre sabrás cuando una variable es un precio, un email, una edad, lo que necesites.
- Te permiten abstraerte de cuestiones como formato, precisión, etc.

Propiedades múltiples para un solo concepto

Veamos un objeto típico de cualquier negocio: `Customer` que da lugar a varios ejemplos clásicos de Value Object. Un cliente siempre suele tener un nombre, que acostumbra a ser una combinación de nombre de pila y uno o más apellidos. También tiene una dirección, que es una combinación de unos cuantos datos.

```
1  class Customer
2  {
3      private $id;
4      private $name;
5      private $firstSurname;
6      private $lastSurname;
7      private $street;
8      private $streetNumber;
9      private $floor;
10     private $postalCode;
11     private $city;
12 }
```

El constructor de nuestro `Customer` podría ser muy complicado, y eso que no hemos incluido todos los campos:

```
class Customer
{
    private $id;
    private $name;
    private $firstSurname;
    private $lastSurname;
    private $street;
    private $streetNumber;
    private $floor;
    private $postalCode;
    private $city;

    public function __construct(
        string $id,
        string $name,
        string $firstSurname,
        ?string $lastSurname,
        string $street,
        string $streetNumber,
        string $floor,
        string $postalCode,
        string $city
    )
    {
        $this->id = $id;
        $this->name = $name;
        $this->firstSurname = $firstSurname;
```

```
28          $this->lastSurname = $lastSurname;
29          $this->street = $street;
30          $this->streetNumber = $streetNumber;
31          $this->floor = $floor;
32          $this->postalCode = $postalCode;
33          $this->city = $city;
34      }
35
36      public function fullName(): string
37      {
38          $fullName = $this->name . ' ' . $this->fi\
39  rstSurname;
40
41          if ($this->lastSurname) {
42              $fullName .= ' ' . $this->lastSurname;
43          }
44
45          return $fullName;
46      }
47
48      public function address(): string
49      {
50          $address = $this->street . ', ' . $this->\
51  streetNumber;
52
53          if ($this->floor) {
54              $address .= ' '. $this->floor;
55          }
56
```

```
57          $address .= $this->postalCode. '-'.$this-\
58  >city;
59
60          return $address;
61      }
62  }
```

Solemos decir que las cosas que cambian juntas deben ir juntas, pero eso también implica que las cosas que no cambian juntas deberían estar separadas. En el constructor van todos los detalles mezclados y se hace muy difícil de manejar.

Yo no sé a ti pero a mí esto me pide un builder:

```
1   class CustomerBuilder
2   {
3       private $name;
4       private $firstSurname;
5       private $lastSurname;
6       private $street;
7       private $streetNumber;
8       private $floor;
9       private $postalCode;
10      private $city;
11
12      public function withName(string $name, string\
13   $firstSurname, ?string $lastSurname) : self
14      {
15          $this->name = $name;
16          $this->firstSurname = $firstSurname;
17          $this->lastSurname = $lastSurname;
```

```
18
19          return $this;
20      }
21
22      public function withAddress(string $street, s\
23  tring $streetNumber, string $floor, string $posta\
24  lCode, string $city): self
25      {
26          $this->street = $street;
27          $this->streetNumber = $streetNumber;
28          $this->floor = $floor;
29          $this->postalCode = $postalCode;
30          $this->city = $city;
31
32          return $this;
33      }
34
35      public function build() : Customer
36      {
37          return new Customer(
38              $this->id,
39              $this->name,
40              $this->firstSurname,
41              $this->lastSurname,
42              $this->street,
43              $this->streetNumber,
44              $this->floor,
45              $this->postalCode,
46              $this->city
```

```
47          );
48      }
49  }
```

Builder que podríamos usar así:

```
1  $customerBuilder = new CustomerBuilder();
2
3  $customer = $customerBuilder
4      ->withName('Fran', 'Iglesias', 'Gómez')
5      ->withAddress('Piruleta St', '123', '4', '080\
6  30', 'Barcelona')
7      ->build();
```

Gracias a usar el Builder podemos ver que hay, al menos, dos conceptos: el nombre del cliente y su dirección. De hecho, en la dirección tendríamos también dos conceptos: la localidad y las señas dentro de esa localidad.

Vamos por partes:

Value Object simple

Parece que no, pero manejamos mucha lógica en algo tan simple como un nombre. Veamos por ejemplo:

- En España usamos nombres con dos apellidos, pero en muchos otros países se suele usar un nombre con un único apellido.
- A veces necesitamos usar partes del nombre por separado, como sería el nombre de pila (“Estimada Susana”, “Sr. Pérez”). Otras veces queremos combinarlo de diferentes

formas, como podría ser poner el apellido primero, lo que es útil para listados.
- Y, ¿qué pasa si queremos introducir nueva información relacionada con el nombre? Por ejemplo, el tratamiento (Sr./Sra., Estimado/Estimada, etc.).

El nombre del cliente se puede convertir fácilmente a un Value Object, lo que retirará cualquier lógica de la "gestión" del nombre de la clase `Customer`, contribuyendo al Single Responsibility Principle y proporcionándonos un comportamiento reutilizable.

Así que podemos crear un Value Object sencillo:

```
1  class PersonName
2  {
3      /** @var string */
4      private $name;
5      /** @var string */
6      private $firstSurname;
7      /** @var string */
8      private $lastSurname;
9
10     public function __construct(string $name, str\
11 ing $firstSurname, string $lastSurname)
12     {
13         $this->name = $name;
14         $this->firstSurname = $firstSurname;
15         $this->lastSurname = $lastSurname;
16     }
17 }
```

La validación debe hacerse en el constructor, de modo que solo se puedan instanciar `PersonName` correctos. Supongamos que nuestras reglas son:

- Name y FirstSurname son obligatorios y no pueden ser un string vacío.
- LastSurname es opcional.

Al incluir la validación tendremos el siguiente código:

```
1  class PersonName
2  {
3      /** @var string */
4      private $name;
5      /** @var string */
6      private $firstSurname;
7      /** @var string */
8      private $lastSurname;
9
10     public function __construct(string $name, str\
11 ing $firstSurname, ?string $lastSurname = null)
12     {
13         if ($name === '' || $firstSurname === '')\
14  {
15             throw new InvalidArgumentException('N\
16 ame and First Surname are mandatory');
17         }
18         $this->name = $name;
19         $this->firstSurname = $firstSurname;
20         $this->lastSurname = $lastSurname;
```

```
21      }
22  }
```

Más adelante volveremos sobre este objeto. Ahora vamos a definir varios Value Objects. De momento, solo me voy a concentrar en los constructores, sin añadir ningún comportamiento, ni siquiera el método `equals` ya que quiere centrarme en cómo movernos de usar escalares a estos objetos.

Value Object Compuesto

Para crear el VO `Address` haremos algo parecido y crearemos una clase `Address` para representar las direcciones de los clientes.

Sin embargo, hemos dicho que podríamos crear un Value Object en torno al concepto de localidad o código postal, que incluiría el código postal y la ciudad. Obviamente, esto dependerá de nuestro dominio. En algunos casos no nos hará falta esa granularidad porque simplemente queremos disponer de una dirección postal de nuestros clientes para enviar comunicaciones. Pero en otros casos puede ocurrir que nuestro negocio tenga aspectos que dependan de ese concepto, como un servicio cuya tarifa sea función de la ubicación.

```
1  class Locality
2  {
3      /** @var string */
4      private $postalCode;
5      /** @var string */
6      private $locality;
7
8      public function __construct(string $postalCod\
```

```
9   e, string $locality)
10      {
11          $this->isValidPostalCode($postalCode);
12          $this->isValidLocality($locality);
13
14          $this->postalCode = $postalCode;
15          $this->locality = $locality;
16      }
17
18      private function isValidPostalCode(string $po\
19  stalCode) : void
20      {
21          if (\strlen($postalCode) !== 5 || (int) s\
22  ubstr($postalCode, 0, 2) > 52) {
23              throw new InvalidArgumentException('I\
24  nvalid Postal Code');
25          }
26      }
27
28      private function isValidLocality(string $loca\
29  lity) : void
30      {
31          if ($locality === '') {
32              throw new InvalidArgumentException('L\
33  ocality should have a value');
34          }
35      }
```

La verdad es que podríamos hilar más fino y declarar un VO `PostalCode`:

```
class PostalCode
{
    /** @var string */
    private $postalCode;

    public function __construct(string $postalCod\
e)
    {
        $this->isValidPostalCode($postalCode);

        $this->postalCode = $postalCode;
    }

    private function isValidPostalCode(string $po\
stalCode) : void
    {
        if (\strlen($postalCode) !== 5 || (int) s\
ubstr($postalCode, 0, 2) > 52) {
            throw new InvalidArgumentException('I\
nvalid Postal Code');
        }
    }
}
```

De modo que `Locality` quedaría así:

```
class Locality
{
    /** @var PostalCode */
    private $postalCode;
    /** @var string */
    private $locality;

    public function __construct(PostalCode $posta\
lCode, string $locality)
    {
        $this->isValidLocality($locality);

        $this->postalCode = $postalCode;
        $this->locality = $locality;
    }

    private function isValidLocality(string $loca\
lity) : void
    {
        if ($locality === '') {
            throw new InvalidArgumentException('L\
ocality should have a value');
        }
    }
}
```

En este caso, no consideramos que `PostalCode` sea una dependencia de `Locality`. Estamos hablando de tipos de datos por lo que estos objetos son *newables*, que es una forma de decir que se instancian a medida que se necesitan.

En fin. Volviendo a nuestro problema original de crear un objeto `Address` podríamos adoptar este enfoque:

```
class Address
{
    /** @var string */
    private $street;
    /** @var string */
    private $streetNumber;
    /** @var string */
    private $floor;
    /** @var Locality */
    private $locality;

    public function __construct(string $street, s\
tring $streetNumber, ?string $floor, Locality $lo\
cality)
    {
        if ('' === $street || '' === $streetNumbe\
r) {
            throw new InvalidArgumentException('A\
ddress should include street and number');
        }
        $this->street = $street;
        $this->streetNumber = $streetNumber;
        $this->floor = $floor;
        $this->locality = $locality;
    }
}
```

Puesto que `Locality` es un VO no es necesario validarla. Además, aquí no necesitamos saber cómo se construye por lo que nos daría igual si hemos optado por un diseño u otro de la clase, ya que la vamos a recibir construida y `Address` puede confiar en que funcionará como es debido.

Siempre que un objeto requiere muchos parámetros en su construcción puede ser interesante plantearse si tenemos buenas razones para organizarlos en forma de Value Objects, aplicando el principio de co-variación: si cambian juntos, deberían ir juntos. En este caso, `$street`, `$streetNumber` y `$floor` pueden ir juntos, en forma de `StreetAddress` porque entre los tres componen un concepto útil.

```
1  class StreetAddress
2  {
3      /** @var string */
4      private $street;
5      /** @var string */
6      private $streetNumber;
7      /** @var string */
8      private $floor;
9
10     public function __construct(string $street, s\
11 tring $streetNumber, ?string $floor)
12     {
13         if ('' === $street || '' === $streetNumbe\
14 r) {
15             throw new InvalidArgumentException('A\
16 ddress should include street and number');
17         }
18         $this->street = $street;
```

```
19          $this->streetNumber = $streetNumber;
20          $this->floor = $floor;
21      }
22  }
```

De este modo, Address se hace más simple y ni siquiera tiene que ocuparse de validar nada:

```
1   class Address
2   {
3       /** @var StreetAddress */
4       private $streetAddress;
5       /** @var Locality */
6       private $locality;
7
8       public function __construct(StreetAddress $st\
9   reetAddress, Locality $locality)
10      {
11          $this->streetAddress = $streetAddress;
12          $this->locality = $locality;
13      }
14  }
```

En resumidas cuentas, a medida que reflexionamos sobre los conceptos del dominio podemos percibir la necesidad de trasladar esa reflexión al código de una forma más articulada y precisa. Pero como hemos señalado antes todo depende de las necesidades de nuestro dominio. Lo cierto es que, como veremos a lo largo del artículo, cuanto más articulado tengamos el dominio, vamos a tener más capacidad de maniobra y muchísima más coherencia.

Introduciendo los Value Objects

Volvamos a `Customer`. De momento, el hecho de introducir una serie de Value Objects no afecta para nada al código que tengamos, por lo que podríamos estar creando cada uno de los VO, haciendo *commits*, mezclando en master y desplegando sin afectar de ningún modo a la funcionalidad existente. Simplemente, hemos añadido clases a nuestra base de código y ahí están: esperando a ser utilizadas.

En este caso, tener a `CustomerBuilder` nos viene muy bien pues encapsula la compleja construcción de Customer, aislándola del resto del código. Podremos refactorizar `Customer` sin afectar a nadie. Empezaremos por el nombre:

```
1  class Customer
2  {
3      private $id;
4      /** @var PersonName */
5      private $personName;
6      private $street;
7      private $streetNumber;
8      private $floor;
9      private $postalCode;
10     private $city;
11
12     public function __construct(
13         string $id,
14         PersonName $personName,
15         string $street,
16         string $streetNumber,
17         string $floor,
```

```
18          string $postalCode,
19          string $city
20      ) {
21          $this->id = $id;
22          $this->personName = $personName;
23          $this->street = $street;
24          $this->streetNumber = $streetNumber;
25          $this->floor = $floor;
26          $this->postalCode = $postalCode;
27          $this->city = $city;
28      }
29
30      public function fullName(): string
31      {
32          return $this->personName->fullName();
33      }
34
35      public function address(): string
36      {
37          $address = $this->street . ', ' . $this->\
38  streetNumber;
39
40          if ($this->floor) {
41              $address .= ' '. $this->floor;
42          }
43
44          $address .= $this->postalCode. '-'.$this-\
45  >city;
46
```

```
47          return $address;
48      }
49  }
```

Como podemos ver, para empezar el constructor ya es más simple. Además, el método `fullName` puede delegarse al disponible en el objeto `PersonName`, que se puede ocupar cómodamente de cualquier variante o formato particular que necesitemos a lo largo de la aplicación.

```
1   class PersonName
2   {
3       /** @var string */
4       private $name;
5       /** @var string */
6       private $firstSurname;
7       /** @var string */
8       private $lastSurname;
9
10      public function __construct(string $name, str\
11  ing $firstSurname, ?string $lastSurname = null)
12      {
13          if ($name === '' || $firstSurname === '')\
14   {
15              throw new InvalidArgumentException('N\
16  ame and First Surname are mandatory');
17          }
18          $this->name = $name;
19          $this->firstSurname = $firstSurname;
20          $this->lastSurname = $lastSurname;
```

```
21      }
22
23      public function fullName(): string
24      {
25          $fullName = $this->name . ' ' . $this->fi\
26  rstSurname;
27
28          if ($this->lastSurname) {
29              $fullName .= ' ' . $this->lastSurname;
30          }
31
32          return $fullName;
33      }
34  }
```

Es por esto que decimos que los Value Objects "atraen" comportamientos, ya que cualquier cosa que las clases usuarias necesiten puede encapsularse en el propio VO. Si necesitásemos el nombre en un formato apto para listas podríamos hacer lo siguiente:

```
1   class PersonName implements PersonNameInterface
2   {
3       /** @var string */
4       private $name;
5       /** @var string */
6       private $firstSurname;
7       /** @var string */
8       private $lastSurname;
9
```

```
10      public function __construct(string $name, str\
11  ing $firstSurname, ?string $lastSurname = null)
12      {
13          if ($name === '' || $firstSurname === '')\
14   {
15              throw new InvalidArgumentException('N\
16  ame and First Surname are mandatory');
17          }
18          $this->name = $name;
19          $this->firstSurname = $firstSurname;
20          $this->lastSurname = $lastSurname;
21      }
22
23      public function fullName(): string
24      {
25          return $this->name .' ' . $this->surname(\
26  );
27      }
28
29      public function listName(): string
30      {
31          return $this->surname() . ', ' . $this->n\
32  ame;
33      }
34
35      public function surname(): string
36      {
37          $surname = $this->firstSurname;
38
```

```
39          if ($this->lastSurname) {
40              $surname .= ' ' . $this->lastSurname;
41          }
42
43          return $surname;
44      }
45
46      public function treatment() : string
47      {
48          return $this->name();
49      }
50  }
```

Como tenemos un Builder que encapsula la construcción de `Customer`, lo que hacemos es modificar esa construcción de acuerdo al nuevo diseño:

```
1   class CustomerBuilder
2   {
3       private $personName;
4       private $street;
5       private $streetNumber;
6       private $floor;
7       private $postalCode;
8       private $city;
9
10      public function withName(string $name, string\
11   $firstSurname, ?string $lastSurname) : self
12      {
13          $this->personName = new PersonName($name,\
```

```
14   $firstSurname, $lastSurname);
15
16           return $this;
17       }
18
19       public function withAddress(string $street, s\
20   tring $streetNumber, string $floor, string $posta\
21   lCode, string $city): self
22       {
23           $this->street = $street;
24           $this->streetNumber = $streetNumber;
25           $this->floor = $floor;
26           $this->postalCode = $postalCode;
27           $this->city = $city;
28
29           return $this;
30       }
31
32       public function build() : Customer
33       {
34           return new Customer(
35               $this->id,
36               $this->personName,
37               $this->street,
38               $this->streetNumber,
39               $this->floor,
40               $this->postalCode,
41               $this->city
42           );
```

```
43      }
44  }
```

Fíjate que he dejado el método `withName` tal y como estaba. De esta forma, no cambio la interfaz pública de `CustomerBuilder`, como tampoco cambia la de `Customer` salvo en el constructor, y el código que lo usa no se enterará del cambio. En otras palabras, el ejemplo anterior funcionará exactamente igual:

```
1  $customerBuilder = new CustomerBuilder();
2
3  $customer = $customerBuilder
4      ->withName('Fran', 'Iglesias', 'Gómez')
5      ->withAddress('Piruleta St', '123', '4', '080\
6  30', 'Barcelona')
7      ->build();
```

Por supuesto, haríamos lo mismo con el VO Address. Así queda Customer:

```
1  class Customer
2  {
3      private $id;
4      /** @var PersonName */
5      private $personName;
6      /** @var Address */
7      private $address;
8
9      public function __construct(
10         string $id,
```

```
11          PersonName $personName,
12          Address $address
13      ) {
14          $this->id = $id;
15          $this->personName = $personName;
16          $this->address = $address;
17      }
18
19      public function fullName(): string
20      {
21          return $this->personName->fullName();
22      }
23
24      public function address(): string
25      {
26          return $this->address->full();
27      }
28  }
```

El método `full` en `Address` lo resuelvo mediante un *type casting* a string de sus componentes, que es una manera sencilla de disponer de su valor en un formato escalar estándar:

```
class Address
{
    /** @var StreetAddress */
    private $streetAddress;
    /** @var Locality */
    private $locality;

    public function __construct(StreetAddress $st\
reetAddress, Locality $locality)
    {
        $this->streetAddress = $streetAddress;
        $this->locality = $locality;
    }

    public function full(): string
    {
        return (string)$this->streetAddress . ' '\
 . (string)$this->locality();
    }
}
```

En este caso necesitaremos:

```
class StreetAddress
{
    /** @var string */
    private $street;
    /** @var string */
    private $streetNumber;
    /** @var string */
    private $floor;

    public function __construct(string $street, s\
tring $streetNumber, ?string $floor)
    {
        if ('' === $street || '' === $streetNumbe\
r) {
            throw new InvalidArgumentException('A\
ddress should include street and number');
        }
        $this->street = $street;
        $this->streetNumber = $streetNumber;
        $this->floor = $floor;
    }

    public function __toString(): string
    {
        $fullAddress = $this->street . ' ' . $thi\
s->streetNumber;

        if ($this->floor) {
            $fullAddress .= ', '. $this->floor;
```

```
        }

        return $fullAddress;
    }
}
```

Y también:

```
class PostalCode
{
    /** @var string */
    private $postalCode;

    public function __construct(string $postalCod\
e)
    {
        $this->isValidPostalCode($postalCode);

        $this->postalCode = $postalCode;
    }

    private function isValidPostalCode(string $po\
stalCode) : void
    {
        if (\strlen($postalCode) !== 5 || (int) s\
ubstr($postalCode, 0, 2) > 52) {
            throw new InvalidArgumentException('I\
nvalid Postal Code');
        }
    }
```

```
23
24      public function __toString(): string
25      {
26          return $this->postalCode;
27      }
28  }
```

Así como:

```
1   class Locality
2   {
3       /** @var PostalCode */
4       private $postalCode;
5       /** @var string */
6       private $locality;
7
8       public function __construct(PostalCode $posta\
9   lCode, string $locality)
10      {
11          $this->isValidLocality($locality);
12
13          $this->postalCode = $postalCode;
14          $this->locality = $locality;
15      }
16
17      private function isValidLocality(string $loca\
18  lity) : void
19      {
20          if ($locality === '') {
21              throw new InvalidArgumentException('L\
```

```
ocality should have a value');
        }
    }

    public function __toString(): string
    {
        return (string) $this->postalCode .'-'.$t\
his->locality;
    }
}
```

Del mismo modo que antes, modificaremos `CustomerBuilder` para utilizar los nuevos objetos:

```
class CustomerBuilder
{
    private $personName;
    private $address;

    public function withName(string $name, string\
 $firstSurname, ?string $lastSurname) : self
    {
        $this->personName = new PersonName($name,\
 $firstSurname, $lastSurname);

        return $this;
    }

    public function withAddress(string $street, s\
tring $streetNumber, string $floor, string $posta\
```

```
17 lCode, string $city) : self
18     {
19         $locality = new Locality(new PostalCode($\
20 postalCode), $city);
21         $streetAddress = new StreetAddress($stree\
22 t, $streetNumber, $floor);
23 
24         $this->address = new Address($streetAddre\
25 ss, $locality);
26 
27         return $this;
28     }
29 
30     public function build() : Customer
31     {
32         return new Customer(
33             $this->id,
34             $this->personName,
35             $this->address
36         );
37     }
38 }
```

Y ya está, hemos hecho este cambio sin tener que tocar en ningún lugar más del código. Obviamente, tener un Builder de Customer nos ha facilitado muchos las cosas, lo que nos dice que es bueno tener un único lugar de instanciación de los objetos.

Beneficios

El beneficio más evidente es que las clases importantes del dominio como `Customer`, quedan mucho más compactas. Hemos podido reducir ocho propiedades a dos, que son conceptos relevantes dentro de `Customer`.

Por otro lado, todo lo que tiene que ver con ellos, `Customer` se lo delega. Dicho de otro modo, `Customer` no tiene que saber cómo se da formato a un nombre o a una dirección. Simplemente, cuando se lo piden entrega el nombre o la dirección formateados. Asimismo, cualquier otro objeto que usase `PersonName` o `Address`, lo hará de la misma manera.

Otra cosa interesante es que los cambios que necesitemos en el comportamiento de estas propiedades pueden aplicarse sin tocar el código de la clase, modificando los Value Objects, con lo cual el nuevo comportamiento se extenderá a todas las partes de la aplicación que lo utilicen.

Introduciendo nuevas features a través de los Value Objects

Vamos a ver cómo la nueva situación en la que nos deja el refactor nos facilita la vida en el futuro. Imaginemos que tenemos que añadir una nueva feature en la aplicación.

Es importante tratar bien a los clientes, por lo que nos han pedido incluir una propiedad de género que permita personalizar el tratamiento que utilizamos en las comunicaciones.

¿A quién pertenecería esa propiedad en nuestro modelo? En el diseño inicial tendríamos que ponerla en `Customer`, pero ahora podríamos hacerlo en `PersonName`. Aún mejor: no toquemos nada, o casi nada, de lo que hay.

Supongamos que `Customer` tiene un método `treatment` al que recurrimos para montar emails o cartas:

```
public function treatment(): string
{
    return $this->personName->treatment();
}
```

Primero queremos un nuevo Value Object, que será Gender:

```
class Gender
{
    /** @var string */
    private $gender;

    private const FEMALE = 'female';
    private const MALE = 'male';

    private const VALID_GENDERS = [
        self::FEMALE,
        self::MALE
    ];

    private const TREATMENTS = [
        self::FEMALE => 'Estimada',
        self::MALE => 'Estimado'
    ];

    public function __construct(string $gender)
    {
```

```
21          if (!\in_array(strtolower($gender), self:\
22  :VALID_GENDERS, true)) {
23              throw new \InvalidArgumentException('\
24  Gender should be one of '.implode(', ', self::VAL\
25  ID_GENDERS));
26          }
27
28          $this->gender = strtolower($gender);
29      }
30
31      public function treatment(): string
32      {
33          return self::TREATMENTS[$this->gender];
34      }
35  }
```

Este tipo de Value Object será un Enumerable. Representa conceptos que tienen un número limitado (numerable) de valores. PHP, de momento, no tiene una implementación propia como otros lenguajes, por lo que podemos simularla de este modo.

El siguiente paso es hacer que la clase PersonName implemente una interfaz PersonNameInterface:

```
interface PersonNameInterface
{
    public function fullName() : string;

    public function listName() : string;

    public function surname() : string;

    public function treatment(): string;
}
```

Y hacemos que Customer la utilice. Es un cambio pequeño, que nos permitirá usar implementaciones alternativas:

```
class Customer
{
    private $id;
    /** @var PersonNameInterface */
    private $personName;
    /** @var Address */
    private $address;

    public function __construct(
        string $id,
        PersonNameInterface $personName,
        Address $address
    ) {
        $this->id = $id;
        $this->personName = $personName;
        $this->address = $address;
```

```
17      }
18
19      public function fullName(): string
20      {
21          return $this->personName->fullName();
22      }
23
24      public function address(): string
25      {
26          return $this->address->full();
27      }
28
29      public function treatment(): string
30      {
31          return $this->personName->treatment();
32      }
33  }
```

Ahora, crearemos un tipo de `PersonName` que sepa algo acerca del género del nombre:

```
1   class GenderMarkedPersonName implements PersonNam\
2   eInterface
3   {
4       /** @var Gender */
5       private $gender;
6       /** @var PersonName */
7       private $personName;
8
9       public function __construct(Gender $gender, P\
```

```
10  ersonName $personName)
11      {
12          $this->gender = $gender;
13          $this->personName = $personName;
14      }
15
16      public function fullName(): string
17      {
18          return $this->personName->fullName();
19      }
20
21      public function listName(): string
22      {
23          return $this->personName->listName();
24      }
25
26      public function surname(): string
27      {
28          return $this->personName->surname();
29      }
30
31      public function treatment() : string
32      {
33          return $this->gender->treatment().' '.$th\
34  is->personName->treatment();
35      }
36  }
```

Y ahora, el último cambio, en el Builder usaremos la nueva implementación:

```
class CustomerBuilder
{
    private $personName;
    private $address;
    private $gender;

    public function withName(string $name, string\
 $firstSurname, ?string $lastSurname) : self
    {
        $this->personName = new PersonName($name,\
 $firstSurname, $lastSurname);

        return $this;
    }

    public function withAddress(string $street, s\
tring $streetNumber, string $floor, string $posta\
lCode, string $city) : self
    {
        $locality = new Locality(new PostalCode($\
postalCode), $city);
        $streetAddress = new StreetAddress($stree\
t, $streetNumber, $floor);

        $this->address = new Address($streetAddre\
ss, $locality);

        return $this;
    }
```

```
30
31      public function withGender(string $gender): s\
32  elf
33      {
34          $this->gender = new Gender($gender);
35
36          return $this;
37      }
38
39      public function build() : Customer
40      {
41          $personName = new GenderMarkedPersonName(\
42  $this->gender, $this->personName);
43
44          return new Customer(
45              $this->id,
46              $personName,
47              $this->address
48          );
49      }
50  }
```

Esta modificación ya tiene algo más de calado, pero sigue siendo razonablemente segura. Si te fijas, solo estamos modificando comportamientos del Builder y ahí podemos ser un poco menos estrictos. Teniendo muchísimo rigor, podríamos crear un decorador de `CustomerBuilder` que crease un `Customer` cuyo `PersonNameInterface` fuese un GenderMarkerPersonName, pero ya estaríamos entrando quizá en la sobre-ingeniería.

Fíjate que ahora `Customer` es capaz de usar el tratamiento

adecuado para cada cliente y realmente no lo hemos tenido que tocar. De hecho, hemos añadido la `feature` añadiendo bastante código, pero con un mínimo riesgo de afectación al resto de la aplicación.

Haciendo balance

Veamos algunos números. Empezamos con dos clases: `Customer` y `CustomerBuilder`. ¿Sabes cuántas tenemos ahora? Nada menos que nueve, además de una interfaz. La parte buena es que hemos afectado muy poco al código restante, tan solo en el último paso, cuando hemos tenido que introducir una nueva `feature`. Pero aquí ya no estamos hablando de refactor.

Sin embargo, nuestro dominio tiene ahora muchísima flexibilidad y capacidad de cambio.

Sería bastante fácil, por ejemplo, dar soporte a los múltiples formatos de dirección postal que se usan en todo el mundo, de modo que nuestro negocio está mejor preparado para expandirse internacionalmente, puesto que solo tendríamos que introducir una interfaz y nuevos formatos a medida que los necesitemos, sin tener que cambiar el `core` del dominio. Puede sonar exagerado, pero estos pequeños detalles pueden ser un dolor de cabeza enorme si seguimos el modelo con el que empezamos. Algo tan pequeño puede ser la diferencia entre un código y un negocio que escale fácilmente o no.

Refactoriza a Enumerables

En este capítulo profundizaremos en una idea que ya apuntamos en un capítulo anterior: refactorizar introduciendo enumerables. Los **enumerables** son tipos de datos que cuentan con un número finito de valores posibles.

Supongamos el típico problema de representar los estados de alguna entidad o proceso. Habitualmente usamos un *string* o un número para ello. La primera opción ayuda a que el valor sea legible por humanos, mientras que la representación numérica puede ser más compacta aunque más difícil de entender.

En cualquier caso, el número de estados es limitado y lo podemos contar. El problema es garantizar la consistencia de los valores que utilicemos para representarlos, incluso entre distintos sistemas.

Y aquí es dónde pueden ayudarnos los **Enumerables**.

Los **Enumerables** se modelan como **Value Objects**. Esto quiere decir que un objeto representa un valor y se encarga de mantener su consistencia, disfrutando de todas las ventajas que señalamos en el capítulo anterior[4].

En la práctica, además, podemos hacer que los **Enumerables** nos permitan una representación semántica en el código, aunque internamente transporten valores abstractos, como códigos numéricos, necesarios para la persistencia en base de datos, por ejemplo.

[4] https://franiglesias.github.io/everyday-refactor-4/

De escalar a enumerable

Empecemos con un caso más o menos típico. Tenemos una entidad con una propiedad que puede tener dos valores, como 'activo' y 'cancelado'. Inicialmente la modelamos con un *string*, que es como se va a guardar en base de datos, y confiamos en que lo sabremos manejar sin mayores problemas en el código. ¿Qué podría salir mal?

Para empezar, cuando tenemos una variable o propiedad de tipo *string*, tenemos infinitos valores potenciales de esa variable o propiedad y tan solo queremos usar dos de ellos. Así que, cuando necesitemos usarlo, tendremos que asegurarnos de que solo consideraremos esos dos *strings* concretos. En otras palabras: tendremos que validarlos cada vez.

Habitualmente también haremos alguna normalización, como poner el *string* en minúsculas o mayúsculas, para simplificar el proceso de comparación y asegurarnos una representación coherente.

Pero esto debería hacerse cada vez que vamos a utilizar esta variable o propiedad, o al menos, siempre que sepamos que su origen no es confiable, como el input de usuario o una petición a la API o cualquier fuente de datos que no esté en un estado conocido.

En lugar de esto, deberíamos usar un **Value Object**, como vimos en el capítulo anterior. Realmente, lo único que hace un poco especiales a los **Enumerables** es el hecho de que el número de valores posibles es limitado lo que nos permite usar algunas técnicas interesantes.

Nuestra primera iteración es simple, definimos una clase `Status` que contiene un valor *string*.

```php
<?php
declare(strict_types=1);

namespace App\Domain;

class Status
{
    /** @var string */
    private $status;

    public function __construct(string $status)
    {
        $this->status = $status;
    }

}
```

Personalmente, me gusta añadir un método `__toString` a los VO para poder hacer el *type cast* si lo necesito.

```php
<?php
declare(strict_types=1);

namespace App\Domain;

class Status
{
    /** @var string */
    private $status;

```

```
11      public function __construct(string $status)
12      {
13          $this->status = $status;
14      }
15
16      public function __toString(): string
17      {
18          return $this->status;
19      }
20
21  }
```

Tenemos que definir cuáles son los valores aceptables para este VO, lo cual podemos hacer mediante constantes de clase. Definiremos una para cada valor válido y una extra que será un simple array que los agrupa, lo que nos facilitará la validación.

```
1   <?php
2   declare(strict_types=1);
3
4   namespace App\Domain;
5
6   class Status
7   {
8       public const ACTIVE = 'activo';
9       public const CANCELLED = 'cancelado';
10
11      private const VALID_VALUES = [
12          self::CANCELLED,
13          self::ACTIVE
```

```
14      ];
15
16      /** @var string */
17      private $status;
18
19      public function __construct(string $status)
20      {
21          $this->status = $status;
22      }
23
24      public function __toString(): string
25      {
26          return $this->status;
27      }
28  }
```

En el ejemplo he puesto los valores aceptados en español y los nombres de las constantes en inglés, que es como haremos referencia a ellos en el código por cuestiones de lenguaje de dominio. Esta diferencia podría darse cuando, por ejemplo, necesitamos interactuar con un sistema *legacy* en el que esos valores están representados en español y sería más costoso cambiarlo que adaptarnos.

En cualquier caso, lo que importa es que vamos a tener una representación en código de esa propiedad y los valores concretos son detalles de implementación que en unos casos podremos elegir y en otros no.

Por otro lado, está el tema de las constantes públicas. Es una cuestión de conveniencia, ya que nos puede permitir acceder a los valores estándar en momentos en los que no podemos usar objetos mediante llamada estática.

Nuestro siguiente paso debería ser implementar la validación que nos garantice que podemos instanciar solo valores correctos.

```
<?php
declare(strict_types=1);

namespace App\Domain;

use InvalidArgumentException;

class Status
{
    public const ACTIVE = 'activo';
    public const CANCELLED = 'cancelado';

    private const VALID_VALUES = [
        self::CANCELLED,
        self::ACTIVE
    ];

    /** @var string */
    private $status;

    public function __construct(string $status)
    {
        if (! in_array($status, self::VALID_VALUE\
S, true)) {
            throw new InvalidArgumentException(sp\
rintf('%s is an invalid value for status', $statu\
```

```
s));
        }
        $this->status = $status;
    }

    public function __toString(): string
    {
        return $this->status;
    }
}
```

Como se puede ver, es muy sencillo, ya que simplemente comprobamos si el valor aportado está en la lista de valores admitidos. Pero, como es un *string*, podríamos tener algún problema en caso de que nos pasen el dato con alguna mayúscula. En estos casos, no está de más, realizar una normalización básica. Tampoco se trata de arreglar el input externo, pero sí de prevenir alguno de los errores habituales.

```
<?php
declare(strict_types=1);

namespace App\Domain;

use InvalidArgumentException;

class Status
{
    public const ACTIVE = 'activo';
    public const CANCELLED = 'cancelado';

```

```
13      private const VALID_VALUES = [
14          self::CANCELLED,
15          self::ACTIVE
16      ];
17
18      /** @var string */
19      private $status;
20
21      public function __construct(string $status)
22      {
23          $status = mb_convert_case($status, MB_CAS\
24  E_LOWER);
25
26          if (! in_array($status, self::VALID_VALUE\
27  S, true)) {
28              throw new InvalidArgumentException(sp\
29  rintf('%s is an invalid value for status', $statu\
30  s));
31          }
32
33          $this->status = $status;
34      }
35
36      public function __toString(): string
37      {
38          return $this->status;
39      }
40  }
```

Para terminar con lo básico, necesitamos un método para

comprobar la igualdad, así como un método para obtener su valor escalar si fuese preciso.

```
<?php
declare(strict_types=1);

namespace App\Domain;

use InvalidArgumentException;

class Status
{
    public const ACTIVE = 'activo';
    public const CANCELLED = 'cancelado';

    private const VALID_VALUES = [
        self::CANCELLED,
        self::ACTIVE
    ];

    /** @var string */
    private $status;

    public function __construct(string $status)
    {
        $status = mb_convert_case($status, MB_CAS\
E_LOWER);

        if (! in_array($status, self::VALID_VALUE\
S, true)) {
```

```
28              throw new InvalidArgumentException(sp\
29 rintf('%s is an invalid value for status', $statu\
30 s));
31          }
32          $this->status = $status;
33      }
34 
35      public function value(): string
36      {
37          return $this->status;
38      }
39 
40      public function equals(Status $anotherStatus)\
41 : bool
42      {
43          return $this->status === $anotherStatus->\
44 status;
45      }
46 
47      public function __toString(): string
48      {
49          return $this->value();
50      }
51 
52 }
```

Y, con esto, ya tenemos un Enumerable.

Bonus points

Hay algunas cosas interesantes que podemos hacer con los **enumerables**, a fin de que resulten más cómodos y útiles.

Por ejemplo, podemos querer tener *named constructors* que hagan más explícita la forma de creación.

```
public static function fromString(string $status)\
: Status
{
    return new self($status);
}
```

Puesto que son pocos valores, podríamos permitirnos tener *named constructors* para crear directamente instancias con un valor determinado:

```
<?php
declare(strict_types=1);

namespace App\Domain;

use InvalidArgumentException;
use phpDocumentor\Reflection\Types\Self_;

class Status
{
    public const ACTIVE = 'activo';
    public const CANCELLED = 'cancelado';

```

```
14      private const VALID_VALUES = [
15          self::CANCELLED,
16          self::ACTIVE
17      ];
18
19
20      /** @var string */
21      private $status;
22
23      private function __construct(string $status)
24      {
25          $status = mb_convert_case($status, MB_CAS\
26  E_LOWER);
27
28          if (! in_array($status, self::VALID_VALUE\
29  S, true)) {
30              throw new InvalidArgumentException(sp\
31  rintf('%s is an invalid value for status', $statu\
32  s));
33          }
34          $this->status = $status;
35      }
36
37      public static function fromString(string $sta\
38  tus): Status
39      {
40          return new self($status);
41      }
42
```

```
43      public static function active(): Status
44      {
45          return new self(self::ACTIVE);
46      }
47
48      public static function cancelled(): Status
49      {
50          return new self(self::CANCELLED);
51      }
52
53      public function value(): string
54      {
55          return $this->status;
56      }
57
58      public function equals(Status $anotherStatus)\
59  : bool
60      {
61          return $this->status === $anotherStatus->\
62  status;
63      }
64
65      public function __toString(): string
66      {
67          return $this->value();
68      }
69
70  }
```

Con esto, podemos hacer privado el constructor standard,

usando así la clase:

```
1 $initialStatus = Status::active();
2
3 $newStatus = Status::cancelled();
```

Enumerables y cambios de estado

Supongamos que una cierta propiedad de una entidad se puede modelar con un enumerable de n elementos con la característica de que solo puede cambiar en una cierta secuencia.

Con frecuencia nos encontramos que esta gestión de estados la realiza la entidad. Sin embargo, podemos delegar en el enumerable buena parte de este comportamiento.

A veces esta secuencia es lineal, indicando que la entidad pasa, a lo largo de su ciclo de vida, por los diferentes estados en un orden prefijado. Por ejemplo, un contrato puede pasar por los estados *pre-signed*, *signed*, *extended* and *finalized*, pero siempre lo hará en ese orden, por lo que es necesario comprobar que no es posible asignar a un contrato un estado nuevo que sea "incompatible" con el actual.

Otras veces, el orden de la secuencia puede variar, pero solo se puede pasar de unos estados determinados a otros. Por ejemplo, un post de un blog, puede pasar de *draft* a *ready to review*, pero no directamente a *published*, mientras que desde *ready to review* puede volver a *draft*, si el revisor no lo encuentra adecuado, o avanzar a *published* si está listo para ver la luz.

Como hemos dicho, este tipo de reglas de negocio pueden encapsularse en el propio enumerable simplificando así el código

de la entidad. Hay muchas formas de hacer esto y en casos complejos necesitaremos hacer uso de otros patrones.

Veamos el ejemplo lineal. Supongamos un `ContractStatus` que admite tres estados que se suceden en una única secuencia. Podemos tener un método en el Enumerable para avanzar un paso el estado:

```php
<?php
declare(strict_types=1);

namespace App\Domain;

use DomainException;

class ContractStatus
{
    public const PRESIGNED = 'presigned';
    public const SIGNED = 'signed';
    public const FINALIZED = 'finalized';

    private const VALID_STATUSES = [
        self::PRESIGNED,
        self::SIGNED,
        self::FINALIZED
    ];
    /** @var string */
    private $status;

    public function __construct(string $status)
    {
```

```
24          $status = mb_convert_case($status, MB_CAS\
25  E_LOWER);
26          if (! in_array($status, self::VALID_STATU\
27  SES, true)) {
28              throw new \InvalidArgumentException(s\
29  printf('%s status not valid', $status));
30          }
31          $this->status = $status;
32      }
33
34      public function status(): string
35      {
36          return $this->status;
37      }
38
39      public function forward(): ContractStatus
40      {
41          switch ($this->status) {
42              case self::PRESIGNED:
43                  return new self(self::SIGNED);
44              case self::SIGNED:
45                  return new self(self::FINALIZED);
46          }
47
48          throw new DomainException(
49              sprintf('Can not forward from %s stat\
50  us', $this->status())
51          );
52      }
```

```
53 }
```

Este ejemplo nos permite hacer avanzar el estado de un objeto contrato de este modo. Recuerda que al ser un Value Object el método nos devuelve una nueva instancia de `ContractStatus`.

```
1 try {
2     $this->contractStatus = $this->contractStatus\
3 ->forward();
4 } catch (DomainException $exception) {
5     // Do the right thing to manage exception
6 }
```

Otra situación interesante se produce cuando necesitamos reasignar el estado del contrato de forma directa. Por ejemplo, debido a errores o tal vez por necesidades de sincronización entre distintos sistemas. En esos caso, podríamos tener (o no) reglas de negocio que permitan ciertos cambios y prohiban otros.

Para nuestro ejemplo vamos a imaginar que un contrato puede volver atrás un paso (de *signed* a *pre-signed* y de *finalized* a *signed*) o avanzar un paso, como en el método `forward`.

Esta implementación es bastante tosca, pero creo que representa con claridad la intención. El método `changeTo` nos permite pasarle un nuevo `ContractStatus` y nos lo devuelve si el cambio es válido o bien lanza una excepción.

```php
<?php
declare(strict_types=1);

namespace App\Domain;

use DomainException;

class ContractStatus
{
    public const PRESIGNED = 'presigned';
    public const SIGNED = 'signed';
    public const FINALIZED = 'finalized';

    private const VALID_STATUSES = [
        self::PRESIGNED,
        self::SIGNED,
        self::FINALIZED
    ];
    /** @var string */
    private $status;

    public function __construct(string $status)
    {
        $status = mb_convert_case($status, MB_CAS\
E_LOWER);
        if (! in_array($status, self::VALID_STATU\
SES, true)) {
            throw new \InvalidArgumentException(s\
printf('%s status not valid', $status));
```

```
30          }
31          $this->status = $status;
32      }
33
34      public function status(): string
35      {
36          return $this->status;
37      }
38
39      public function forward(): ContractStatus
40      {
41          switch ($this->status) {
42              case self::PRESIGNED:
43                  return new self(self::SIGNED);
44              case self::SIGNED:
45                  return new self(self::FINALIZED);
46          }
47
48          throw new DomainException(
49              sprintf('Can not forward from %s stat\
50  us', $this->status())
51          );
52      }
53
54      public function changeTo(ContractStatus $newC\
55  ontractStatus): ContractStatus
56      {
57
58          switch ($this->status) {
```

```
59              case self::PRESIGNED:
60                  if ($newContractStatus->status() \
61  !== self::SIGNED) {
62                      $this->failWhenChangeIsNotAll\
63  owed($newContractStatus);
64                  }
65                  break;
66              case self::FINALIZED:
67                  if ($newContractStatus->status() \
68  !== self::SIGNED) {
69                      $this->failWhenChangeIsNotAll\
70  owed($newContractStatus);
71                  }
72                  break;
73              default:
74                  if ($newContractStatus->status() \
75  === self::SIGNED) {
76                      $this->failWhenChangeIsNotAll\
77  owed($newContractStatus);
78                  }
79          }
80
81          return $newContractStatus;
82      }
83
84      private function failWhenChangeIsNotAllowed(C\
85  ontractStatus $newContractStatus): void
86      {
87          throw new DomainException(
```

```
88              sprintf(
89                  'Change form %s to %s is not allo\
90  wed',
91                  (string)$this,
92                  (string)$newContractStatus
93              )
94          );
95      }
96  }
```

En esencia, el método `changeTo` valida que el estado se pueda cambiar teniendo en cuenta el estado actual. La idea de fondo es aplicar el principio `Tell, don't ask`, de modo que no le preguntemos al contrato por su estado, ni a `ContractStatus` por su valor, si no que le decimos que cambie a un nuevo estado si es posible. En caso de fallo, ya tomaremos nosotros las medidas necesarias.

```
1  try {
2      $newStatus = new ContractStatus('finalized');
3
4      $this->contractStatus = $this->contractStatus\
5  ->changeTo($newStatus);
6  } catch (DomainException $exception) {
7      // Do the right thing to manage exception
8  }
```

Enumerables como traductores

¿Y qué ocurre si tenemos que interactuar con distintos sistemas que representan el mismo significado con distintos valores? Podría ocurrir que uno de los sistemas lo hiciese con enteros de modo que necesitamos alguna traducción.

Un enfoque pragmático, cuando las combinaciones de valores/versiones son reducidas, sería incorporar esa capacidad al propio Enumerable, mediante un named constructor específico y un método para obtener esa versión del valor.

```php
<?php
declare(strict_types=1);

namespace App\Domain;

use InvalidArgumentException;

class Status
{
    public const ACTIVE = 'activo';
    public const CANCELLED = 'cancelado';

    private const VALID_VALUES = [
        self::CANCELLED,
        self::ACTIVE
    ];

    private const LEGACY_MAP = [
        101 => self::CANCELLED,
```

```
20          200 => self::ACTIVE
21      ];
22
23      /** @var string */
24      private $status;
25
26      public function __construct(string $status)
27      {
28          $status = mb_convert_case($status, MB_CAS\
29  E_LOWER);
30
31          if (! in_array($status, self::VALID_VALUE\
32  S, true)) {
33              throw new InvalidArgumentException(sp\
34  rintf('%s is an invalid value for status', $statu\
35  s));
36          }
37          $this->status = $status;
38      }
39
40      public static function fromString(string $sta\
41  tus): Status
42      {
43          return new self($status);
44      }
45
46      public static function fromLegacy(int $status\
47  ): Status
48      {
```

```
49          return new self(self::LEGACY_MAP[$status]\
50 );
51      }
52
53      public function toLegacy(): int
54      {
55          return array_flip(self::LEGACY_MAP)[$this\
56 ->status];
57      }
58
59      public function value(): string
60      {
61          return $this->status;
62      }
63
64      public function equals(Status $anotherStatus)\
65 : bool
66      {
67          return $this->status === $anotherStatus->\
68 status;
69      }
70
71      public function __toString(): string
72      {
73          return $this->value();
74      }
75 }
```

Añadiendo rigor al enumerable

Aunque la solución que acabamos de ver resulta práctica en ciertos casos, lo cierto es que no es precisamente rigurosa al mezclar responsabilidades.

Nos hace falta algún tipo de traductor:

```
<?php
declare(strict_types=1);

namespace App\Domain;

class LegacyStatusTransformer
{
    private const LEGACY_MAP = [
        '101' => Status::CANCELLED,
        '200' => Status::ACTIVE
    ];

    public function fromLegacy(string $status): S\
tatus
    {
        return new Status(self::LEGACY_MAP[$statu\
s]);
    }


    public function toLegacy(Status $status): str\
ing
    {
```

```
24          return (string)array_flip(self::LEGACY_MA\
25  P)[$status->value()];
26      }
27  }
```

Aplica Tell, Don’t Ask y la Ley de Demeter

En este capítulo veremos refactors basados en la redistribución de responsabilidades.

Hasta ahora, hemos trabajado refactors muy orientados a mejorar la expresividad del código y a la organización de unidades de código. En este capítulo vamos a trabajar en cómo mejorar las relaciones entre objetos.

Los principios de diseño nos proporcionan criterios útiles tanto para guiarnos en el desarrollo como para evaluar código existente en el que tenemos que intervenir. Vamos a centrarnos en dos principios que son bastante fáciles de aplicar y que mejorarán la inteligibilidad y la posibilidad de testear nuestro código. Se trata de *Tell, Don’t Ask* y la *Ley de Demeter*.

Primero haremos un repaso y luego los veremos en acción.

Tell, don’t ask

La traducción de este enunciado a español sería algo así como “Ordena, no preguntes”. La idea de fondo de este principio es que cuando queremos modificar un objeto basándose su propio estado, no es buena idea preguntarle por su estado (*ask*), hacer el cálculo y cambiar su estado si fuera preciso. En su lugar, lo propio sería encapsular ese proceso en un método del propio objeto y decirle (*tell*) que lo realice él mismo.

Dicho en otras palabras: cada objeto debe ser responsable de su estado.

Veamos un ejemplo bastante absurdo, pero que lo deja claro.

Supongamos que tenemos una clase `Square` que representa un cuadrado y queremos poder calcular su área.

```
1  $square = new Square(20);
2
3  $side = $square->side();
4
5  $area = $side**2;
```

Si aplicamos el principio *Tell, Don't Ask*, el cálculo del área estaría en la clase `Square`:

```
1  $square = new Square(20);
2
3  $area = $square->area();
```

Mejor, ¿no? Veamos por qué.

En el dominio de las figuras geométricas planas, el área o superficie es una propiedad que tienen todas ellas y que, a su vez, depende de otras que son su base y su altura, que en el caso del cuadrado coinciden. La función para determinar el área ocupada por una figura plana depende de cada figura específica.

Posiblemente estés de acuerdo en que al modelar este comportamiento lo pondríamos en la clase de cada figura desde el primer momento, lo que seguramente nos llevaría a una interfaz.

```
interface TwoDimensionalShapeInterface
{
    public function area(): float;
}
```

La primera razón es que toda la información necesaria para calcular el área está en la clase, por lo que tiene todo el sentido del mundo que el conocimiento preciso para calcularla también esté allí. Se aplica el **principio de Cohesión** y el **principio de Encapsulamiento**, manteniendo juntos los datos y las funciones que procesan esos datos.

Una segunda razón es más pragmática: si el conocimiento para calcular el área está en otro lugar, como un servicio, cada vez que necesitemos incorporar una nueva clase al sistema, tendremos que modificar el servicio para añadirle ese conocimiento, rompiendo el **principio Abierto/Cerrado**.

En tercer lugar, el testing se simplifica. Es fácil hacer tests de las clases que representan las figuras. Por otro lado, el testing de las otras clases que las utilizan también se simplifica. Normalmente esas clases usarán el cálculo del área como una utilidad para llevar a cabo sus propias responsabilidades que es lo que queremos saber si hacen correctamente.

Siguiendo el principio ***Tell, Don't Ask*** movemos responsabilidades a los objetos a los que pertenecen.

Ley de Demeter

La **Ley de Demeter**[5] también se conoce como **Principio de menor conocimiento** y, más o menos, dice que un objeto no

[5] El nombre viene del proyecto donde se usó por primera vez.

debería conocer la organización interna de los otros objetos con los que colabora.

Siguiendo la **Ley de Demeter**, como veremos, un método de una clase solo puede usar los objetos a los que conoce. Estos son:

- La propia clase, de la que puede usar todos sus métodos.
- Objetos que son propiedades de esa clase.
- Objetos creados en el mismo método que los usa.
- Objetos pasados como parámetros a ese método.

La finalidad de la **ley de Demeter** es evitar el acoplamiento estrecho entre objetos. Si un método usa un objeto, contenido en otro objeto que ha recibido o creado, implica un conocimiento que va más allá de la interfaz pública del objeto intermedio.

```
1  public function calculatePrice(Product $product, \
2  int $units): float
3  {
4      $discountPct = $product->currentPromotion()->\
5  discountPct();
6      //...
7  }
```

En el ejemplo, el método `calculatePrice` obtiene el descuento aplicable llamando a un método de `Product`, que devuelve otro objeto al cual le preguntamos sobre el descuento.

¿Qué objeto es este y cuál es su interfaz? Podemos suponer que se trata de un objeto `Promotion`, pero eso es algo que sabemos nosotros, no el código. Este conocimiento excesivo nos dice que estamos ante una violación de la **Ley de Demeter**.

Puedes ponerlo así, pero sigue siendo el mismo problema:

```
public function calculatePrice(Product $product, \
int $units): float
{
    $promotions = $product->currentPromotion();
    $discountPct = $promotion->discountPct();
    //...
}
```

Una justificación que se aduce en ocasiones es que puesto que sabes qué clase de objeto devuelve el objeto intermedio, entonces puedes saber su interfaz. Sin embargo, lo que conseguimos es una dependencia oculta de entre dos objetos que no tienen una relación directa. Esto es:

El **objeto A** usa el **objeto B** para obtener y usar el **objeto C**.

El **objeto A** depende del **objeto C**, pero no hay nada en A que nos diga que existe esa dependencia.

Es posible aplicar varias soluciones. La más adecuada depende de varios factores.

Encapsular en nuevos métodos

En algunos casos, se trataría de aplicar el principio ***Tell, Don't Ask***. Esto es. A veces, la responsabilidad de ofrecernos una cierta respuesta encajaría en el objeto intermedio, por lo que podríamos encapsular la cadena de llamadas. Veámoslo con un ejemplo similar:

```
public function calculatePrice(Product $product, \
int $units): float
{
    $basePrice = $product->family()->basePrice();
    $unitPrice = $basePrice + $product->extraPric\
e();
    return $unitPrice * $units;
}
```

En este caso, resulta razonable pensar que la estructura del precio de un producto es algo propio del producto, y los usuarios del objeto no tienen por qué conocerla. En un primer paso, aplicamos la **Ley de Demeter** haciendo que el objeto `Price` sea el que obtiene el precio base, sin que la calculadora tenga que saber de dónde se obtiene.

```
public function calculatePrice(Product $product, \
int $units): float
{
    $basePrice = $product->basePrice();
    $unitPrice = $basePrice + $product->extraPric\
e();
    return $unitPrice * $units;
}
```

En ese caso, `Product` utiliza su colaborador `Family` para obtener el valor que devolver en `basePrice`.

En el segundo paso, aplicamos `Tell, Don't Ask`, ya que realmente estamos pidiéndole cosas a `Product` que puede hacer por sí mismo.

```
public function calculatePrice(Product $product, \
int $units): float
{
    return $product->unitPrice() * $units;
}
```

Reasignación de responsabilidades

El primer ejemplo sobre descuentos es un poco más delicado que el que acabamos de ver:

```
public function calculatePrice(Product $product, \
int $units): float
{
    $discountPct = $product->currentPromotion()->\
discountPct();
    //...
}
```

No está tan claro que las posibles promociones formen parte de la estructura de precios de un producto. Las promociones son seguramente una responsabilidad de Marketing y los productos y precios son de Ventas. Puesto que no queremos tener dos razones para cambiar Producto, lo lógico es que las promociones estén en otra parte.

Dicho de otro modo, no tiene mucho sentido que un producto conozca cuáles son las promociones que se le aplican en el contexto de una campaña de marketing, que son puntuales y limitadas en el tiempo, mientras que la estructura de precio es algo permanente.

Tiene más sentido que la responsabilidad de las promociones esté en otro lugar. Podría ser algo así:

```
public function calculatePrice(Product $product, \
int $units): float
{
    $discountPct = $this->getPromotions->forProdu\
ct($product);
    $price =  $product->unitPrice() * $units;
    return $price - ($price * $discountPct / 100);
}
```

La clase que contiene el método `calculatePrice` tendría un colaborador que le proporciona los descuentos disponibles para un producto.

En resumidas cuentas, el código que incumple la **Ley de Demeter** tratando con objetos que no conoce directamente puede estar revelando problemas más profundos en el diseño y que hay que solucionar. Estos problemas se manifiestan en un acoplamiento fuerte entre objetos que tienen una relación escasa entre sí.

Refactoriza de single return a return early

En el blog ya hemos hablado del patrón clásico Single Exit Point[6] y cómo acabó derivando en *single return*. También algún momento de esta guía de refactor hemos hablado también del *return early*. Ahora vamos a retomarlos conjuntamente porque seguramente nos los encontraremos más de una vez.

Lo primero será saber de qué estamos hablando:

Single return

Se trata de que en cada método o función solo tengamos un único `return`, a pesar de que el código pueda tener diversos caminos que nos permitirían finalizar en otros momentos.

Obviamente, si el método solo tiene un camino posible tendrá un solo `return`.

[6] https://franiglesias.github.io/lidiando-con-el-patron-single-exit-point/

```php
public function isValid(string $luhnCode) : bool
{
    $inverted = strrev($luhnCode);

    $oddAdded = $this->addOddDigits($inverted);
    $evenAdded = $this->addEvenDigits($inverted);

    return ($oddAdded + $evenAdded) % 10 === 0;
}
```

Si el método tiene dos caminos, caben dos posibilidades:

- Uno de los flujos se separa del principal, hace alguna cosa y vuelve de forma natural al tronco para terminar lo que tenga que hacer.

```php
public function forProduct(Client $client, Produc\
t $product)
{
    $contract = new Contract($product);

    if ($client->hasBenefits()) {
        $contract->addBenefits($client->benefits(\
));
    }

    $this->mailer->send($client, $contract);
}
```

- Uno de los flujos se separa para resolver la tarea de una manera alternativa, por lo que podría devolver el resultado una vez obtenido. Sin embargo, si se sigue el patrón *single return*, hay que forzar que el flujo vuelva al principal antes de retornar.

```
1  private function reduceToOneDigit($double) : int
2  {
3      if ($double >= 10) {
4          $double = intdiv($double, 10) + $double %\
5   10;
6      }
7
8      return $double;
9  }
```

Si el método tiene más de dos caminos se dará una combinación de las posibilidades anteriores, es decir, algunas ramas volverán de forma natural al flujo principal y otras podrían retornar por su cuenta.

En principio, la ventaja del *single Return* es poder controlar con facilidad que se devuelve el tipo de respuesta correcta, algo que sería más difícil si tenemos muchos lugares con `return`. Pero la verdad es que explicitando *return types* es algo de lo que ni siquiera tendríamos que preocuparnos.

En cambio, el mayor problema que tiene Single Return es que fuerza la anidación de condicionales y el uso de `else` hasta extremos exagerados, lo que provoca que el código sea especialmente difícil de leer. Lo peor es que eso no se justifica por necesidades del algoritmo, sino por la gestión del flujo para conseguir que solo se pueda retornar en un punto.

El origen de esta práctica parece que podría ser una mala interpretación del patrón *Single Exit Point* de Djkstra, un patrón que era útil en lenguajes que permitían que las llamadas a subrutinas y sus retornos pudieran hacerse a líneas arbitrarias. Su objetivo era asegurar que se entrase a una subrutina en su primera línea y se volviese siempre a la línea siguiente a la llamada.

Early return

El patrón *early return* consiste en salir de una función o método en cuanto sea posible, bien porque se ha detectado un problema (*fail fast*), bien porque se detecta un caso especial que se maneja fuera del algoritmo general o por otro motivo.

Dentro de este patrón encajan cosas como las cláusulas de guarda, que validan los parámetros recibidos y lanzan una excepción si no son correctos.

También se encuentran aquellos casos particulares que necesitan un tratamiento especial, pero que es breve o inmediato.

De este modo, al final nos queda el algoritmo principal.

El principal inconveniente es la posible inconsistencia que pueda darse en los diferentes returns en cuanto al tipo o formato de los datos, algo que se puede controlar fácilmente forzando un *return type*.

Por otra parte, ganamos en legibilidad, ya que mantenemos bajo control el anidamiento de condicionales y los niveles de indentación.

Además, al tratar primero los casos especiales podemos centrar la atención en el algoritmo principal de ese método.

Ejemplo básico

Hace un par de años comencé a practicar un ejercicio para estudiar algoritmos y estructuras de datos, reproduciéndolos en PHP usando metodología TDD. El código visto ahora está un poco pobre, pero me viene bien porque he encontrado varios ejemplos de *single return* y otros puntos de mejora.

En primer lugar, vamos a ver un caso en el que podemos refactorizar un *single return* muy evidente, pero también uno que no lo es tanto:

```
1  <?php
2
3  namespace Dsa\Algorithms\Sorting;
4
5  class QuickSort
6  {
7      public function sort(array $source)
8      {
9          $length = count($source);
10         if ($length > 1) {
11             $pivot = $this->median($source);
12             $equal = $less = $greater = [];
13             for ($i = 0; $i < $length; $i++) {
14                 if ($source[$i] == $pivot) {
15                     $equal[] = $source[$i];
16                 } elseif ($source[$i] < $pivot) {
17                     $less[] = $source[$i];
18                 } else {
19                     $greater[] = $source[$i];
```

```
20                  }
21              }
22              $sorted = array_merge($this->sort($le\
23  ss), $equal, $this->sort($greater));
24          } else {
25              $sorted = $source;
26          }
27
28          return $sorted;
29      }
30
31      private function median($source)
32      {
33          $points = [];
34          for ($i = 0; $i < 3; $i++) {
35              $point = array_splice($source, rand(0\
36  , count($source) - 1), 1);
37              $points[] = array_shift($point);
38          }
39
40          return array_sum($points) - max($points) \
41  - min($points);
42      }
43  }
```

El primer paso es invertir la condicional, para ver la rama más corta en primer lugar:

```
1  public function sort(array $source)
2  {
3      $length = count($source);
4      if ($length <= 1) {
5          $sorted = $source;
6      } else {
7          $pivot = $this->median($source);
8          $equal = $less = $greater = [];
9          for ($i = 0; $i < $length; $i++) {
10             if ($source[$i] == $pivot) {
11                 $equal[] = $source[$i];
12             } elseif ($source[$i] < $pivot) {
13                 $less[] = $source[$i];
14             } else {
15                 $greater[] = $source[$i];
16             }
17         }
18         $sorted = array_merge($this->sort($less),\
19  $equal, $this->sort($greater));
20     }
21
22     return $sorted;
23 }
```

Al invertir las ramas es fácil ver que en caso de que `$length` sea menor o igual que uno podemos retornar sin problema. De hecho, no tiene mucho sentido intentar ordenar una lista de un solo elemento.

Al hacer esto, también podemos eliminar el uso de la variable temporal `$sorted` que es innecesaria.

```php
public function sort(array $source)
{
    $length = count($source);
    if ($length <= 1) {
        return $source;
    }

    $pivot = $this->median($source);
    $equal = $less = $greater = [];
    for ($i = 0; $i < $length; $i++) {
        if ($source[$i] == $pivot) {
            $equal[] = $source[$i];
        } elseif ($source[$i] < $pivot) {
            $less[] = $source[$i];
        } else {
            $greater[] = $source[$i];
        }
    }

    return array_merge($this->sort($less), $equal\
, $this->sort($greater));
}
```

Con este arreglo el código ya mejora mucho su legibilidad gracias a que despejamos el terreno tratando el caso especial y dejando el algoritmo principal limpio.

Pero vamos a ir un paso más allá. El bucle `for` contiene una forma velada de *single return* en forma de estructura `if...else` que voy a intentar explicar.

El algoritmo **quicksort** se basa en hacer pivotar los elementos

de la lista en torno a su mediana, es decir, al valor que estaría exactamente en la posición central de la lista ordenada. Para ello, se calcula la mediana de forma aproximada y se van comparando los números para colocarlos en la mitad que les toca: bien por debajo o bien por encima de la mediana.

Para eso se compara cada número con el valor mediano para ver sucesivamente si es igual, menor o mayor, con lo que se añade a la sub-lista correspondiente y se van ordenando esas sub-listas de forma recursiva.

En este caso las cláusulas `else` tienden a hacer más difícil la lectura y, aunque la semántica es correcta, podemos hacerlo un poco más claro.

Como ya sabrás, podemos forzar la salida de un bucle con `continue`:

```
1  public function sort(array $source)
2  {
3      $length = count($source);
4      if ($length <= 1) {
5          return $source;
6      }
7
8      $pivot = $this->median($source);
9      $equal = $less = $greater = [];
10     for ($i = 0; $i < $length; $i++) {
11         if ($source[$i] == $pivot) {
12             $equal[] = $source[$i];
13             continue;
14         }
15
```

```
16          if ($source[$i] < $pivot) {
17              $less[] = $source[$i];
18              continue;
19          }
20
21          $greater[] = $source[$i];
22      }
23
24      return array_merge($this->sort($less), $equal\
25  , $this->sort($greater));
26  }
```

Y, aunque en este caso concreto no es especialmente necesario, esta disposición hace que la lectura del bucle sea más cómoda. Incluso es más fácil reordenarlo y que exprese mejor lo que hace:

```
1   public function sort(array $source): array
2   {
3       $length = count($source);
4       if ($length <= 1) {
5           return $source;
6       }
7
8       $pivot = $this->median($source);
9       $equal = $less = $greater = [];
10      for ($i = 0; $i < $length; $i++) {
11          if ($source[$i] > $pivot) {
12              $greater[] = $source[$i];
13              continue;
```

```
14          }
15
16          if ($source[$i] < $pivot) {
17              $less[] = $source[$i];
18              continue;
19          }
20
21          $equal[] = $source[$i];
22      }
23
24      return array_merge($this->sort($less), $equal\
25  , $this->sort($greater));
26  }
```

Otro ejemplo

En este caso es un Binary Search Tree, en el que se nota que no tenía muy claro el concepto de *return early* o, al menos, no lo había aplicado hasta sus últimas consecuencias, por lo que el código no mejora apenas:

```
1   <?php
2
3   namespace Dsa\Structures;
4
5   class BinarySearchTree
6   {
7       /**
8        * @var BinarySearchNode
```

```
9         */
10        private $root;
11
12        public function insert($value)
13        {
14            $new = new BinarySearchNode($value);
15            if (! $this->root) {
16                $this->root = $new;
17            } else {
18                $this->insertNew($this->root, $new);
19            }
20        }
21
22        public function insertNew(
23            BinarySearchNode $current,
24            BinarySearchNode $new
25        ) {
26            if ($new->getValue() < $current->getValue\
27    ()) {
28                if (! $current->getLeft()) {
29                    $current->setLeft($new);
30                } else {
31                    $this->insertNew($current->getLef\
32    t(), $new);
33                }
34            } else {
35                if (! $current->getRight()) {
36                    $current->setRight($new);
37                } else {
```

```
38                  $this->insertNew($current->getRig\
39 ht(), $new);
40              }
41          }
42      }
43
44      public function isInTree($value)
45      {
46          if (! $this->root) {
47              return false;
48          }
49
50          return $this->contains($this->root, $valu\
51 e);
52      }
53
54      public function contains(
55          BinarySearchNode $current = null,
56          $value
57      ) {
58          if (! $current) {
59              return false;
60          }
61
62          return $this->findNode($current, $value) \
63 ? true : false;
64      }
65
66      public function getParentValueOf($value)
```

```
67      {
68          if ($value == $this->root->getValue()) {
69              return null;
70          }
71
72          return $this->findParent($this->root, $va\
73  lue)->getValue();
74      }
75
76      private function findParent(
77          BinarySearchNode $current,
78          $value
79      ) {
80          if ($value < $current->getValue()) {
81              if (! $current->getLeft()) {
82                  return null;
83              } elseif ($current->getLeft()->getVal\
84  ue() == $value) {
85                  return $current;
86              } else {
87                  return $this->findParent($current\
88  ->getLeft(), $value);
89              }
90          } else {
91              if (! $current->getRight()) {
92                  return null;
93              } elseif ($current->getRight()->getVa\
94  lue() == $value) {
95                  return $current;
```

```
96              } else {
97                  return $this->findParent($current\
98  ->getRight(), $value);
99              }
100         }
101     }
102
103     private function findNode(
104         BinarySearchNode $current = null,
105         $value
106     ) {
107         if (! $current) {
108             return false;
109         }
110
111         if ($current->getValue() == $value) {
112             return $current;
113         } elseif ($value < $current->getValue()) {
114             return $this->findNode($current->getL\
115  eft(), $value);
116         } else {
117             return $this->findNode($current->getR\
118  ight(), $value);
119         }
120     }
121 }
```

Empecemos mejorando el método `insert`:

```
public function insert($value)
{
    $new = new BinarySearchNode($value);
    if (! $this->root) {
        $this->root = $new;
    } else {
        $this->insertNew($this->root, $new);
    }
}
```

Que podría quedar así:

```
public function insert($value): void
{
    $new = new BinarySearchNode($value);
    if (! $this->root) {
        $this->root = $new;
        return;
    }

    $this->insertNew($this->root, $new);
}
```

Al método insertNew le sobra indentación:

```
public function insertNew(
    BinarySearchNode $current,
    BinarySearchNode $new
) {
    if ($new->getValue() < $current->getValue()) {
        if (! $current->getLeft()) {
            $current->setLeft($new);
        } else {
            $this->insertNew($current->getLeft(),\
 $new);
        }
    } else {
        if (! $current->getRight()) {
            $current->setRight($new);
        } else {
            $this->insertNew($current->getRight()\
, $new);
        }
    }
}
```

Empezamos aplicando el *return early* una vez:

```
public function insertNew(
    BinarySearchNode $current,
    BinarySearchNode $new
) {
    if ($new->getValue() < $current->getValue()) {
        if (! $current->getLeft()) {
            $current->setLeft($new);

            return;
        }

        $this->insertNew($current->getLeft(), $ne\
w);
    } else {
        if (! $current->getRight()) {
            $current->setRight($new);

            return;
        }

        $this->insertNew($current->getRight(), $n\
ew);
    }
}
```

Y una segunda vez:

```
public function insertNew(
    BinarySearchNode $current,
    BinarySearchNode $new
): void {
    if ($new->getValue() < $current->getValue()) {
        if (! $current->getLeft()) {
            $current->setLeft($new);

            return;
        }

        $this->insertNew($current->getLeft(), $ne\
w);
        return;
    }

    if (! $current->getRight()) {
        $current->setRight($new);

        return;
    }

    $this->insertNew($current->getRight(), $new);
}
```

Otro lugar que necesita un arreglo es el método `findParent`. Aquí hemos usado return early, pero no habíamos sabido aprovecharlo:

```
private function findParent(
    BinarySearchNode $current,
    $value
) {
    if ($value < $current->getValue()) {
        if (! $current->getLeft()) {
            return null;
        } elseif ($current->getLeft()->getValue()\
 == $value) {
            return $current;
        } else {
            return $this->findParent($current->ge\
tLeft(), $value);
        }
    } else {
        if (! $current->getRight()) {
            return null;
        } elseif ($current->getRight()->getValue(\
) == $value) {
            return $current;
        } else {
            return $this->findParent($current->ge\
tRight(), $value);
        }
    }
}
```

Al hacerlo, nos queda un código más limpio:

```
private function findParent(
    BinarySearchNode $current,
    $value
) {
    if ($value < $current->getValue()) {
        if (! $current->getLeft()) {
            return null;
        }

        if ($current->getLeft()->getValue() === $\
value) {
            return $current;
        }

        return $this->findParent($current->getLef\
t(), $value);
    }

    if (! $current->getRight()) {
        return null;
    }

    if ($current->getRight()->getValue() === $val\
ue) {
        return $current;
    }

    return $this->findParent($current->getRight()\
, $value);
```

```
30 }
```

Todos estos refactors se pueden hacer sin riesgo con las herramientas de *Intentions* (comando + return) de PHPStorm que nos ofrecen la **inversión de if/else** (*flip*), y la **separación de flujos** (*split workflows*) cuando son posibles. En todo caso, estas clases estaban cubiertas por tests y estos siguen pasando sin ningún problema.

Finalmente, arreglamos `findNode`, que estaba así:

```
1      private function findNode(
2          BinarySearchNode $current = null,
3          $value
4      ) {
5          if (! $current) {
6              return false;
7          }
8
9          if ($current->getValue() == $value) {
10             return $current;
11         } elseif ($value < $current->getValue()) {
12             return $this->findNode($current->getL\
13 eft(), $value);
14         } else {
15             return $this->findNode($current->getR\
16 ight(), $value);
17         }
18     }
```

Y quedará así:

```
private function findNode(
    BinarySearchNode $current = null,
    $value
) {
    if (! $current) {
        return false;
    }

    if ($current->getValue() === $value) {
        return $current;
    }

    if ($value < $current->getValue()) {
        return $this->findNode($current->getLeft(\
), $value);
    }

    return $this->findNode($current->getRight(), \
$value);
}
```

Dónde poner el conocimiento

En anteriores entregas hemos propuesto refactorizar código a Value Objects o aplicar principios como la *ley de Demeter* o *Tell, don't ask* para mover código a un lugar más adecuado. En este capítulo vamos a analizarlo desde un punto de vista más general.

Solemos decir que refactorizar tiene que ver con el **conocimiento** y el **significado**. Fundamentalmente, porque lo que hacemos es aportar significado al código con el objetivo de que este represente de una manera fiel y dinámica el conocimiento cambiante que tenemos del negocio del que nos ocupamos.

En el código de una aplicación tenemos objetos que representan alguna de estas cosas del negocio:

- **Conceptos**, ya sea en forma de entidades o de value objects. Las entidades representan conceptos que nos interesan por su identidad y tienen un ciclo de vida. Los value objects representan conceptos que nos interesan por su valor.
- **Relaciones entre esos conceptos**, que suelen representarse en forma de agregados y que están definidas por las reglas de negocio.
- **Procesos** que hacen interactuar los conceptos conforme a reglas de negocio también.

Uno de los problemas que tenemos que resolver al escribir código y al refactorizarlo es dónde poner el conocimiento y, más exactamente, las reglas de negocio.

Si hay algo que caracteriza al *legacy* es que el conocimiento sobre las reglas de negocio suele estar disperso a lo largo y ancho del código, en los lugares más imprevisibles y representado de las formas más dispares. El efecto de refactorizar este código es, esperamos, llegar a trasladar ese conocimiento al lugar donde mejor nos puede servir.

Principios básicos

Principio de abstracción. Benjamin Pierce formuló el principio de abstracción en su libro *Types and programming languages*[7]:

> Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

DRY. Por su parte, Andy Hunt y David Thomas, en *The Pragmatic Programmer*[8], presentan una versión de este mismo principio que posiblemente te sonará más: **Don't Repeat Yourself:**

[7] https://www.amazon.es/Types-Programming-Languages-MIT-Press/dp/0262162091/ref=sr_1_1?adgrpid=56467442856&hvadid=275405870353&hvdev=c&hvlocphy=1005434&hvnetw=g&hvpos=1t1&hvqmt=e&hvrand=14620334178548921835&hvtargid=kwd-298006847943&keywords=types+and+programming+languages&qid=1555607597&s=gateway&sr=8-1

[8] https://www.amazon.es/s?k=the+pragmatic+programmer&adgrpid=55802357883&hvadid=275519489680&hvdev=c&hvlocphy=1005434&hvnetw=g&hvpos=1t1&hvqmt=e&hvrand=18337966056430542312&hvtargid=kwd-302199567278&tag=hydes-21&ref=pd_sl_63qujbqsqo_e

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

En esencia, la idea que nos interesa recalcar es que cada regla de negocio estará representada en un único lugar y esa representación será la de referencia para todo el código.

Los principios que hemos enunciado se centran en el carácter único de la representación, pero no nos dicen dónde debe residir la misma. Lo cierto es que es un tema complejo, pues es algo que puede admitir varias interpretaciones y puede depender del estado de nuestro conocimiento actual del negocio.

Buscando dónde guardar el conocimiento

En los objetos a los que pertenece

El principio *Tell, don't ask* nos proporciona una primera pista: el conocimiento que afecta solo a un objeto debería estar en el propio objeto. Esto es, en lugar de obtener información de un objeto para operar con ella y tomar una decisión sobre ese objeto, le pedimos que lo haga él mismo y nos entregue el resultado si es preciso.

En ese sentido, los *Value Objects*, de los que hemos hablado tantas veces, son lugares ideales para encapsular conocimiento. Veamos un ejemplo:

Supongamos que en nuestro negocio estamos interesados en ofrecer ciertos productos o ventajas a usuarios cuya cuenta de correo pertenezca a ciertos dominios. Por tanto, el correo

electrónico es un concepto importante del negocio y lo representamos mediante un Value Object:

```
<?php
declare(strict_types=1);

namespace App\Domain;

use InvalidArgumentException;

class Email
{
    /** @var string */
    private $email;

    public function __construct(string $email)
    {
        if (! filter_var($email, FILTER_VALIDATE_\
EMAIL)) {
            throw new InvalidArgumentException(sp\
rintf('%s is not valid email.', $email));
        }

        $this->email = $email;
    }

    public function value(): string
    {
        return $this->email;
    }
```

```
28
29      public function __toString(): string
30      {
31          return $this->value();
32      }
33  }
```

En un determinado servicio verificamos que el dominio del correo electrónico del usuario se encuentra dentro de la lista de dominios beneficiados de este tratamiento especial.

```
1   <?php
2   declare(strict_types=1);
3
4   namespace App\Domain;
5
6   class OfferPromotion
7   {
8       public function applyTo(Order $order)
9       {
10          [, $domain] = explode('@', $order->custom\
11  erEmail());
12
13          if (in_array($domain, $this->getPromotion\
14  Domains->execute(), true)) {
15              $order->applyPromotion($this);
16          }
17      }
18  }
```

El problema aquí es que el servicio tiene que ocuparse de

obtener el dominio de la dirección de correo, cosa que no tendría que ser de su incumbencia. Pero la clase Email nos está pidiendo a gritos convertirse en la experta de calcular la parte del dominio del correo:

```
<?php
declare(strict_types=1);

namespace App\Domain;

use InvalidArgumentException;

class Email
{
    /** @var string */
    private $email;

    public function __construct(string $email)
    {
        if (! filter_var($email, FILTER_VALIDATE_\
EMAIL)) {
            throw new InvalidArgumentException(sp\
rintf('%s is not valid email.', $email));
        }

        $this->email = $email;
    }

    public function value(): string
    {
```

```
26          return $this->email;
27      }
28
29      public function __toString(): string
30      {
31          return $this->value();
32      }
33
34      public function domain(): string
35      {
36          [, $domain] = explode('@', $this->email);
37
38          return $domain;
39      }
40  }
```

Lo que hace más expresivo nuestro servicio:

```
1   <?php
2   declare(strict_types=1);
3
4   namespace App\Domain;
5
6   class OfferPromotion
7   {
8       public function applyTo(Order $order)
9       {
10          $email = $order->customer()->email();
11
12          if (in_array($email->domain(), $this->get\
```

```
13  PromotionDomains->execute(), true)) {
14              $order->applyPromotion($this);
15          }
16      }
17  }
```

Reglas de negocio como Specification

El ejemplo anterior es una primera aproximación a cómo mover el conocimiento. En este caso no se trata tanto de la regla de negocio como de un requisito para poder implementarla.

Podríamos decir que la regla de negocio implica distintos conocimientos. En términos de negocio nuestro ejemplo se enunciaría como "todos los clientes cuyo dominio de correo esté incluido en la lista tienen derecho a tal ventaja cuando realicen tal acción". Técnicamente implica saber sobre usuarios y sus emails, y saber extraer su dominio de correo para saber si está incluido en tal lista.

Desde el punto de vista del negocio la regla relaciona clientes, seleccionados por una característica, con una ventaja que les vamos a otorgar.

Ese conocimiento se puede encapsular en una **Specification**, que no es más que un objeto que puede decidir si otro objeto cumple una serie de condiciones.

```php
<?php
declare(strict_types=1);

namespace App\Domain;

class HasDomainEligibleForPromotion
{
    public $domains = [
        'example.com',
        'example.org'
    ];

    public function isSatisfiedBy(Customer $custo\
mer): bool
    {
        if (in_array($customer->email(), $this->d\
omains, true)) {
            return true;
        }

        return false;
    }
}
```

Ahora el conocimiento de la regla de negocio se encuentra en un solo lugar y lo puedes reutilizar allí donde lo necesites[9]

[9]Una objeción que se puede poner a este código es que instanciamos la Specification. Normalmente lo mejor sería inyectar en el servicio una factoría de Specification para pedirle las que necesitemos y que sea la factoría la que gestione sus posibles dependencias.

```php
<?php
declare(strict_types=1);

namespace App\Domain;

class OfferPromotion
{
    public function applyTo(Order $order)
    {
        $eligibleForPromotion = new HasDomainElig\
ibleForPromotion();

        if ($eligibleForPromotion->isSatisfiedBy(\
$order->customer())) {
            $order->applyPromotion($this);
        }
    }
}
```

No solo eso, sino que incluso nos permite escribir mejor el servicio al expresar las relaciones correctas: en este caso la regla de negocio se basa en una propiedad de los clientes y no de los pedidos, aunque luego se aplique el resultado a los pedidos o al cálculo de su importe.

Sobre el patrón Specification puedes encontrar más información en este artículo[10]

[10] https://franiglesias.github.io/patron-specification-del-dominio-a-la-infraestructura-1/