ALCALDÍA MAYOR
DE BOGOTÁ D.C.
CULTURA RECREACIÓN Y DEPORTE
Instituto Distrital de las Artes

GUIA DE BUENAS PRÁCTICAS PARA EL DESARROLLO DE SOFTWARE

BOGOTÁ | INSTITUTO DISTRITAL DE LAS ARTES IDARTES

HISTORICO DE CAMBIOS		
Versión	**Fecha**	**Cambios Realizados**
01	22/07/2021	Emisión Inicial

Elaboró:	**Revisó:**	**Aprobó:**	**Avaló:**
Hernán Mauricio Rincón Bedoya Contratista Tecnologías de la Información **Cristian Camilo Calderón Tapia** Contratista Tecnologías de la Información **Steven Hernández Ríos** Contratista Tecnologías de la Información **Laura Viviana Cruz Martínez** Contratista Tecnologías de la Información	**Camila Crespo Murillo** Contratista Oficina Asesora de Planeación y Tecnologías de la Información	**Carlos Alfonso Gaitán Sánchez** Jefe de la Oficina Asesora de Planeación y Tecnologías de la Información	**Carlos Alfonso Gaitán Sánchez** Jefe de la Oficina Asesora de Planeación y Tecnologías de la Información

CONTENIDO

1. INTRODUCCIÓN

Este documento es una guía para lograr la adopción de buenas prácticas de desarrollo de software las cuales recogen experiencias a lo largo de los años en la industria del software. Los elementos aquí nombrados aportan a la construcción del documento de arquitectura de solución del dominio de sistemas de información, que componen el marco de referencia de arquitectura empresarial del Mintic.

En el ciclo de vida del desarrollo de software se plantean fases como el análisis, diseño, codificación, pruebas, implementación y mantenimiento. Esta guía busca orientar las fases de diseño en primera instancia, pero se centrará en la etapa de codificación o desarrollo.

A continuación, se presentan los objetivos y alcance de la guía, así mismo se identifican el cómo implementar principios y buenas prácticas para el desarrollo y en qué casos se debe implementar.

2. OBJETIVOS DE LA GUÍA

- Orientar a los equipos de desarrollo durante las fases de diseño y desarrollo de software.
- Mostrar a partir de ejemplos casos de uso de refactorización de código y las ventajas de utilizar buenas prácticas de desarrollo de software.

3. ALCANCE

Los lineamientos aquí definidos aplican a todos los servidores públicos, contratistas y practicantes que tienen un vínculo laboral y/o contractual con el Instituto o personal de otras entidades que intervengan en los procesos de desarrollo de software de la entidad.

4. GLOSARIO

- Ciclo de vida de Desarrollo de Software: Conjunto de actividades de un proyecto de construcción y resultados asociados que generan un producto de software.

- Códigos o Archivos fuentes: Son los documentos que contienen las líneas de código que se ejecutaron en el lenguaje de programación, utilizado para el desarrollo del producto o software.

- POO: Programación orientada a objetos

- Clase: Es una abstracción que representa un elemento que puede ser tangible o intangible pero que posee características o atributos y comportamientos o métodos. Los atributos pueden tener modificadores de acceso generalmente público, privado y protegido, estos determinan el encapsulamiento de la clase.

- Modificadores de acceso: son instrucciones o conjunto de palabras clave propias de cada lenguaje de programación orientado a objetos que permite controlar la visibilidad de clases, estado (propiedades) y funcionalidades (métodos) de una aplicación desde otras partes de esta. Los modificadores más comunes son public, private y protected.

- Objeto: Es una instancia de una clase UML dentro de una aplicación de software, que consta de un estado (el conjunto de atributos), y comportamientos, se pueden entender como pequeñas funcionalidades que hacen parte de la solución total desarrollada.

- APIE: Acrónimo que hace referencia a los principios de diseño orientado a objetos en el orden: Abstracción, Polimorfismo, Inheritance (Herencia) y Encapsulamiento.

- Abstracción: Capacidad de representar la información que es importante para el contexto del problema.

- Herencia: capacidad de construir nuevas clases a partir de clases existentes, la clase hija hereda el estado (atributos) y el comportamiento (métodos) definidos. Generalmente se usa la palabra clave extends para denotar la clase padre. En algunos lenguajes de programación no existe la herencia múltiple, es decir que una clase sólo puede heredar de una clase padre.

- Polimorfismo: Capacidad de un método de devolver valores diferentes dadas ciertas condiciones (parámetros y herencia).

- Polimorfismo por sobrecarga: En el evento en que una clase tiene dos métodos con el mismo nombre y cada método tiene diferente número o tipo de parámetros. El entorno de ejecución que está instanciando la clase puede distinguir cuál método usar.

- Polimorfismo por sobreescritura: Cuando se hereda, desde la clase hija es posible redefinir un método propio de la clase padre. El entorno de ejecución que está instanciando la clase puede distinguir cuál método usar.

- Encapsulamiento: Habilidad de un objeto de decidir qué partes puede exponer a otros objetos, se usa generalmente los modificadores de acceso para tal fin.

- Clase Abstracta: Al igual que las clases, representan una abstracción. No obstante, se enfocan en generalidades de manera que son usadas para iniciar jerarquía de clases, pueden tener estados (atributos) y comportamientos (métodos) y puede implementar dichos métodos (tener contenido en los métodos). A diferencia de una clase convencional con la clase abstracta no es posible instanciar objetos directamente, por lo que esta acción se debe hacer a través de una clase hija o clase concreta que herede de la clase abstracta.

- Interfaz: Al igual que las clases abstractas, representan una abstracción de un elemento que representa el inicio de una jerarquía de clases. A diferencia de la clase abstracta la interfaz no tiene estado (atributos) y el comportamiento (métodos) solo se definen “salvo algunos lenguajes como Java 8 o C# 8 en donde sí se pueden implementar los métodos”, creando un contrato con las clases hijas que lo implementan, pues estas deben de forma obligatoria implementar los métodos. A diferencia de las clases abstractas, una clase puede implementar más de una interfaz, pero dependiendo del lenguaje de programación sólo puede heredar de una sola clase padre.

- Cohesión: es el nivel en que distintos elementos de un sistema permanecen unidos para lograr un mejor resultado que si trabajaran por separado. En otras palabras, es la capacidad de agrupar diferentes unidades de software para generar una unidad mayor.

- Acoplamiento: Se refiere al nivel de interdependencia que tienen dos unidades de software entre sí, entendiendo por unidades de software: clases, subtipos, métodos, módulos, funciones, etc. Si dos unidades de software son completamente independientes la una de la otra, se considera que están desacopladas.

- SOLID: Principios de diseño cuya finalidad es hacer un código más mantenible y limpio.

- SRP: Single Responsibility Principle o principio de Responsabilidad Simple, es la S del acrónimo de los principios SOLID.

- OCP: Open Closed Principle o principio de Abierto Cerrado, es la O del acrónimo de los principios SOLID.

- LSP: Liskov Substitution Principle o principio de sustitución de Liskov, es la L del acrónimo de los principios SOLID.

- ISP: Interface Segregation Principle o principio de segregación de interfaces, es la I del acrónimo de los principios SOLID.

- DIP: Dependency Inversion Principle o principio de inversión de dependencias, es la I del acrónimo de los principios SOLID.

- Requerimiento: Es la concepción de la idea la cual busca generar soluciones a necesidades específicas, el cual se puede manejar mediante una herramienta automatizada de software.

- Software: Son las aplicaciones desarrolladas bajo un lenguaje de programación y un entorno de ejecución que dan solución a un problema o necesidad específica.

- Arquitectura de software: Describe el conjunto de componentes de software que hacen parte de un sistema de información y las relaciones que existen entre ellos. Cada componente de software está descrito en términos de sus características funcionales y no funcionales. Las relaciones se expresan a través de conectores que reflejan el flujo de datos, de control y de sincronización. La arquitectura de software debe describir la manera en que el sistema de información maneja aspectos como seguridad, comunicación entre componentes, formato de los datos, acceso a fuentes de datos, entre otros.

- PHP: (acrónimo recursivo de PHP: Hypertext Preprocessor) es un lenguaje de código abierto muy popular especialmente adecuado para el desarrollo web y que puede ser incrustado en HTML.

5. APLICACIÓN DE PRINCIPIOS DE POO

Los pilares de programación a objetos denominado en ocasiones como APIE por las siglas: Abstracción, Polimorfismo, Inheritance (Herencia) y Encapsulamiento; tienen una intervención en las fases de diseño y de desarrollo de software.

5.1 PILARES DE LA POO

Es importante tener la capacidad de abstraer los componentes que hacen parte del requerimiento funcional para traducirlo a clases. Este ejercicio de modelamiento se suele saltar en ocasiones pensando que el frameworks ya cuenta con la estructura suficiente para solventar la lógica del negocio, siendo esto un grave error.

El desarrollador debe tener claro que la lógica del negocio debe ser vista como un núcleo a un alto nivel, luego debe pasar a dividirse en componentes o paquetes que se comuniquen entre sí de una forma desacoplada. En ese sentido es importante aplicar los pilares:

5.1.1 ABSTRACCIÓN

Se debe realizar una identificación de los elementos que hacen parte de la lógica del negocio.

Se puede partir del ejercicio de la identificación de entidades o tablas para el modelado de base de datos, Cada tabla se convierte en una clase de tipo Entity, en donde se identifica su estado (atributos), sus relaciones de agregación o composición con otras tablas, pero no su comportamiento (métodos).

En este punto hay que mencionar que es mejor dejar que las tablas tipo Entity se encarguen exclusivamente del mapeo de datos con el ORM que disponga el frameworks.

En ese sentido el comportamiento de los elementos debe ir en otras clases. Por ejemplo, dada la clase que representa un contrato muy probablemente tendrá información cómo su valor, fecha de inicio, fecha de fin, contratista, supervisor y obligaciones.

En modelado de base de datos el contratista y el supervisor seguramente serán llaves foráneas para la tabla contrato, mientras que las obligaciones se convierten en otra tabla de uno a muchos por cada contrato.

Para POO, el contratista y el supervisor se modelan como clases separadas y las obligaciones pasan a ser un atributo de tipo lista en la clase de contrato.

¿Pero qué relación tiene la obligación con el contrato, es por agregación o composición?

Recordemos que tanto la composición como la agregación tienen la relación de tipo "tiene un" en este caso un contrato (el todo) tiene unas obligaciones (las partes). Si el ciclo de vida de la parte muere cuando muere el todo, o no tiene sentido que exista si ya no existe el todo, entonces hablamos de composición. de lo contrario hablamos de agregación.

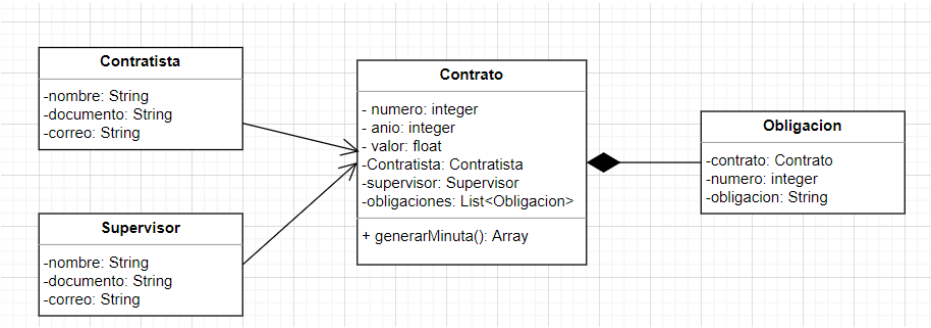

Figura 1: Relaciones básicas de un contrato.

Volviendo al tema del comportamiento o métodos del contrato, podemos ver que han solicitado que ese contrato pueda generar la minuta en formato PDF.

Como dicho comportamiento no hace parte de la lógica de almacenado de datos ni relaciones entre entidades, se debe crear una clase adicional que permita implementar esa lógica separado del guardado de la información.

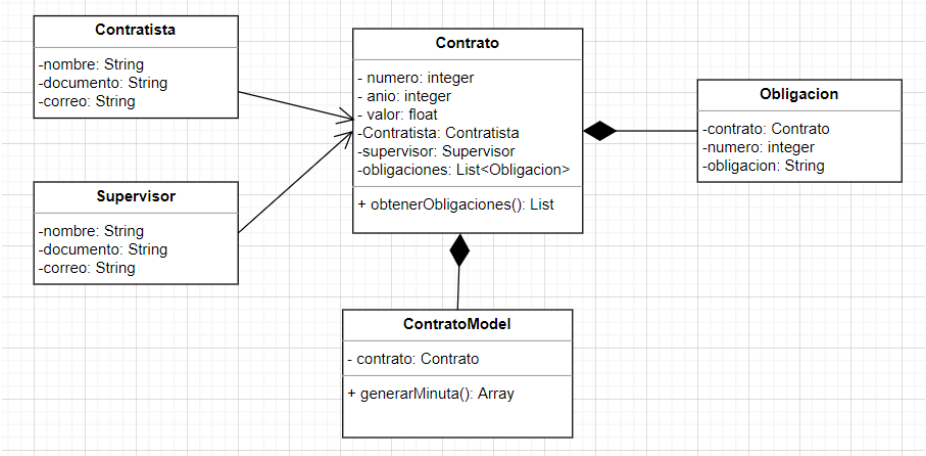

Figura 2: Separación de responsabilidades por medio de la abstracción.

Es de esta manera cómo se inicia un paquete de funcionalidades, donde implícitamente aplicamos el principio de responsabilidad simple de SOLID.

5.1.2 POLIMORFISMO

El polimorfismo es útil cuando tenemos métodos cuyo comportamiento sea variante dependiendo de sus argumentos de entrada.

En los ejemplos anteriores suponga que para crear una obligación de un contrato se requiere: el contrato, el numeral de la obligación y el texto de la obligación.

Luego solicitan que el numeral se tiene que calcular automáticamente y por lo tanto ya no debe ser solicitado, pero el código anterior está enlazado a otra aplicación legado que ya no cuenta con mantenimiento y se requiere mantener la compatibilidad.

A esto se le conoce como polimorfismo por sobrecarga:

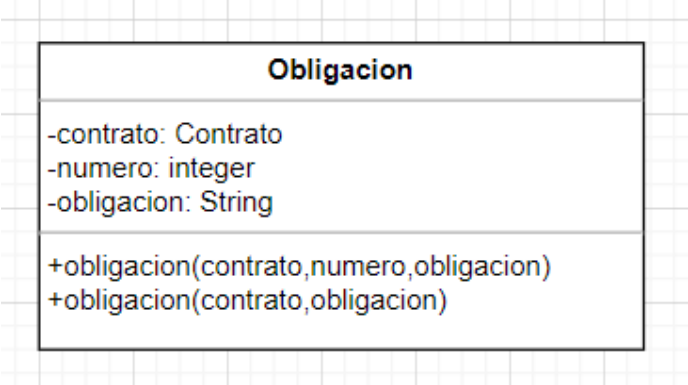

Figura 3: Ejemplo de polimorfismo por sobrecarga

Por otro lado, se puede sobre escribir un comportamiento de la clase padre, desde una clase hijo.

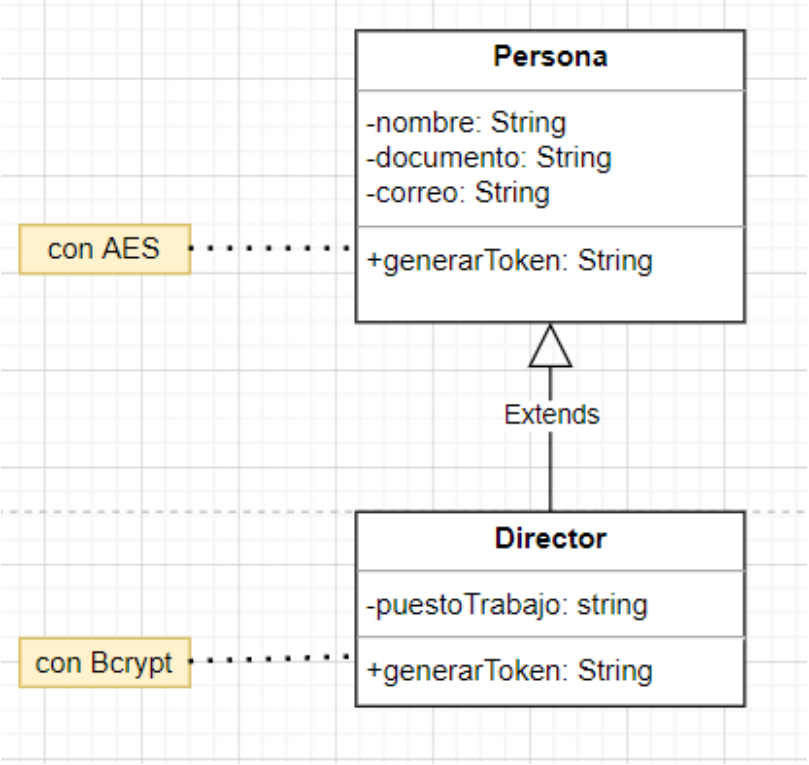

Figura 4: Ejemplo de herencia por sobreescritura.

Con el siguiente código es posible sobreescribir el método de encriptamiento de un token simplemente instanciando a un objeto de la clase hijo, pero haciéndolo de tipo clase padre.

```
Persona $pablo = new Director();
$pablo->generarToken();
```

Figura 5: Código de instancia de una clase hija manteniendo el tipo del padre.

Nota: en versiones inferiores a php 8 no existe el tipo estricto de variables por lo que definir que la variable $pablo sea de tipo persona marcaría error.

A esto se le conoce como polimorfismo por sobreescritura.

5.1.3 HERENCIA

En ocasiones es importante crear una jerarquía de clases que permita abstraer a un nivel general las características de un elemento (clase padre) hasta encontrar características específicas (clase hija).

En el ejemplo anterior se presentó a un contratista y un supervisor asociado a un contrato. Estos dos elementos se caracterizan porque ambos son personas que comparten atributos y métodos, pero pueden tener comportamientos diferentes, por ejemplo, el supervisor genera el acta de inicio, mientras que el contratista la firma:

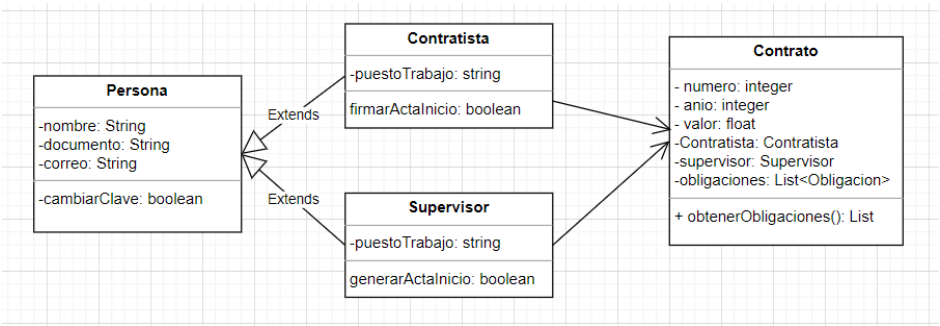

Figura 6 Ejemplo de herencia extendiendo funcionalidades de clases de alto nivel.

Otra forma de generar jerarquía de clases es por medio de una interfaz, por ejemplo imagine que la generación de la minuta del contrato ahora tiene que generarse tanto en PDF como en Excel, la clase encargada de hacer esa actividad era ContratoModel en su método generarMinuta(). Una opción válida sería agregar un switch o if que recibiera por parámetro el formato y simplemente seguir con el proceso, pero podemos pensar en que existe un formato genérico de minuta que luego permite agrupar formatos personalizados.

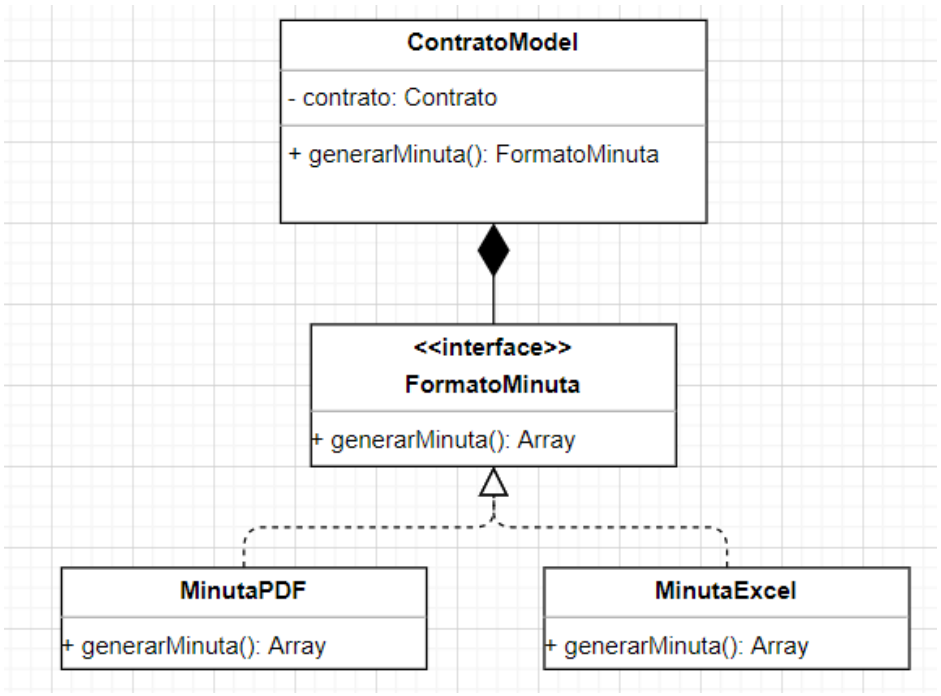

Figura 7 Ejemplo de simular la herencia implementando funcionalidades desde una interfaz.

En este caso la interface FormatoMinuta especifica un comportamiento para generar la minuta, y luego las clases MinutaPDF y MinutaExcel implementan el método de acuerdo a su naturaleza.

5.1.4 ENCAPSULAMIENTO

En los dos pilares mencionados anteriormente se ha venido trabajando implícitamente con encapsulamiento, pues en cada clase los atributos se les asigna un acceso privado (signo menos al inicio) mientras que a los métodos se les da un acceso público (signo más al inicio).

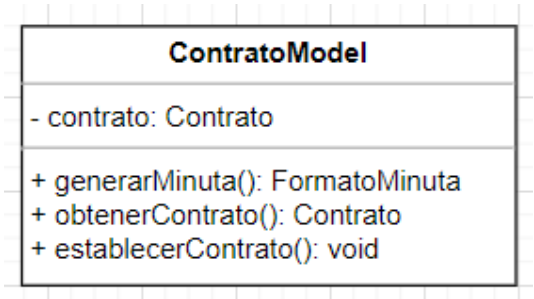

Figura 8 Clase con atributos privados y métodos públicos.

Pero no solamente los modificadores de acceso actúan sobre atributos y métodos, también es posible asignarlos a una clase y de esta forma podríamos por ejemplo dejar visible a una clase solo para un paquete específico o solo para una clase que se relacione directamente con ella.

5.2 APLICACIÓN DE PRINCIPIOS SOLID

SOLID es un acrónimo de 5 principios que acuñó Michael Feathers, basado en principios de diseño condensados por Robert Martin "Tío Bob" que ayudan a organizar el código en componentes, funciones y clases. Además, permite crear un mejor software que sea más fácil de mantener, entender y probar.

Los principios SOLID están estrechamente relacionados con los patrones de diseño y en este sentido persiguen la alta cohesión y el bajo acoplamiento de software.

A continuación, se menciona cada uno de los 5 principios y se acompañará con una frase que permite interiorizar el concepto.

5.2.1 **Single Responsibility:** Una sola razón para cambiar

Este principio plantea que un módulo (entendido como una clase o método) solo debe tener una razón para cambiar. Esto no debe confundirse con la recomendación de que un módulo (clase o método) se debe dedicar a solo una cosa y por tanto tener una sola responsabilidad. A continuación, se explica cómo se distancian estos dos conceptos por medio de un ejemplo:

Dada la tabla llamada tbl_plan que representa un plan distrital de desarrollo. Se dispone entonces una clase llamada Plan para mapear la tabla y una clase llamada PlanRepository para controlar el acceso a datos y la lógica de reportes.

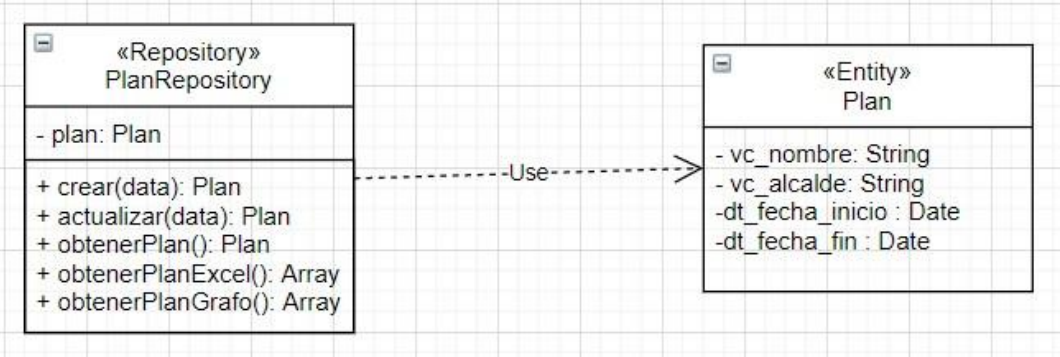

Figura 9 Ejemplo de clase sobrecargada de responsabilidades.

Se pueden apreciar cuatro métodos en la clase de tipo Repositorio. los dos primeros usados para gestionar la creación y actualización del plan de desarrollo y otros dos para generar arreglos que sirven para mostrar al usuario el arbol del plan de desarrollo. Si vamos a la recomendación de que una clase solo debe tener una responsabilidad podríamos pensar que la responsabilidad de PlanRepository es gestionar las interacciones de un Plan.

Imaginemos ahora que se requiere agregar un método para mostrar el plan en formato PDF, y además que se debe permitir borrar un plan. Identificamos dos cambios asociados a dos razones diferentes una de acceso a datos y otra de reporte. Vemos que, aunque cumple con la definición que la clase "solo tiene una responsabilidad", se incumple el principio de "una sola razón para cambiar". En este caso la solución sería:

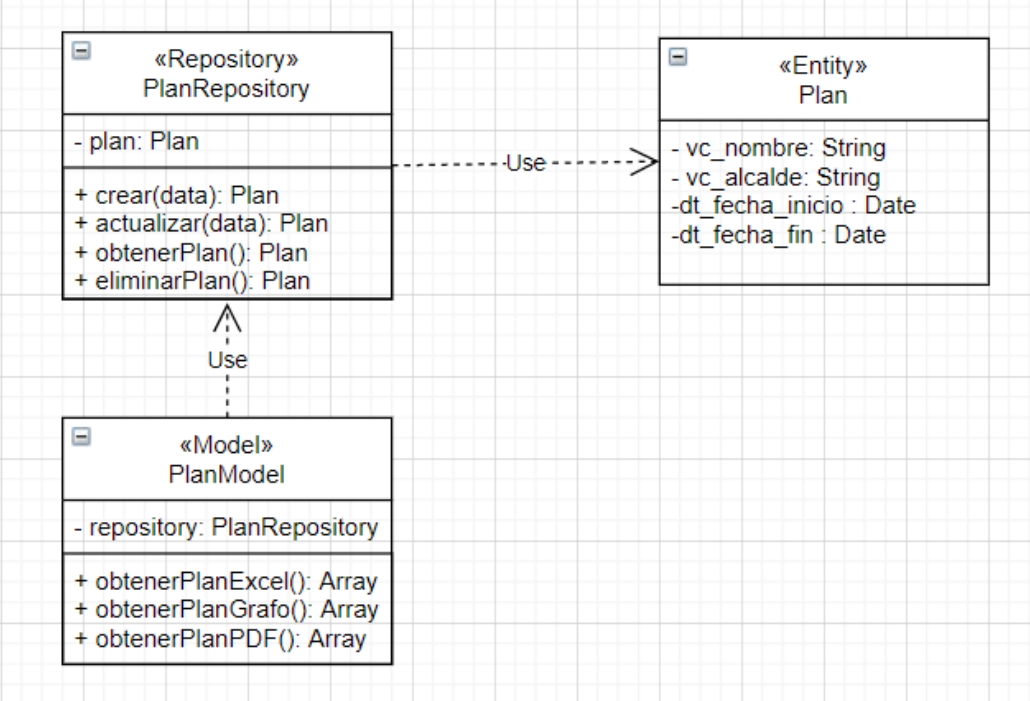

Figura 10 Ejemplo de solución a una clase sobrecargada de responsabilidades.

Ahora observamos una nueva clase llamada PlanModel cuya función es contener la lógica de reportes. Con este ajuste se cumple el principio de de SRP o Single Responsibility Principle.

Tener en cuenta:

- Este principio es muy fácil de solucionar, pero también...
- Es muy fácil de interpretar de manera errónea.
- Esta solución cumple con SRP, pero incumple el principio de Open / Close.
- Algunos patrones que se pueden usar para aplicar este diseño son: Facade y Proxy.

5.2.2 Open / Close: Extender y no modificar

Este principio habla que un módulo debe ser abierto para ser extendido pero cerrado para ser modificado. En simples términos, los cambios que se hacen en las clases deben ser de bajo impacto. Sin embargo, los módulos (conjunto de clases) deben estar diseñados para permitir nuevas funcionalidades sin implicar grandes cambios sobre lo ya construido. A continuación, se explicará con ejemplos cuando aplicar este principio:

Retomando el módulo de Plan, PlanRepository y PlanModel del ejemplo del principio de SRP, imagine que solicitaron que desde una sola vista se pueda elegir en qué tipo de formato ver la estructura del plan de desarrollo. las opciones son Excel, Grafo y PDF. De inmediato una idea que se puede venir a la mente es crear un método que reciba como parámetro el tipo de reporte y que allí se decida cual invocar.

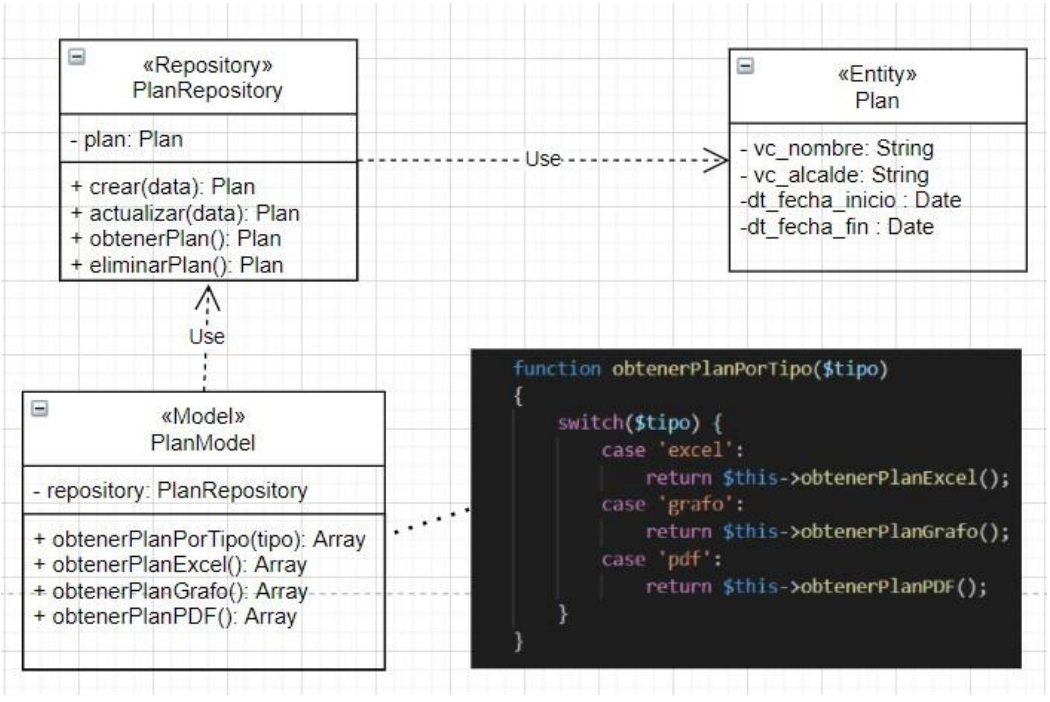

Figura 11 Ejemplo de una implementación que incumple el OCP.

El inconveniente en este caso es que en el evento en que soliciten el plan de desarrollo en un nuevo tipo, se tendría que crear un nuevo método para tal fin, pero además se tendría que agregar un nuevo case al switch del método obtenerPlanPorTipo(). En este momento se incumple el principio de abierto / cerrado pues al introducir una nueva mejora, se expuso el módulo a un cambio que pudo ser evitado.

La solución en este caso es abrir cada tipo de reporte en una clase que implemente de una interfaz y desde la clase PlanModel se direccione a ejecutar el código dependiendo de la vista que el usuario seleccionó:

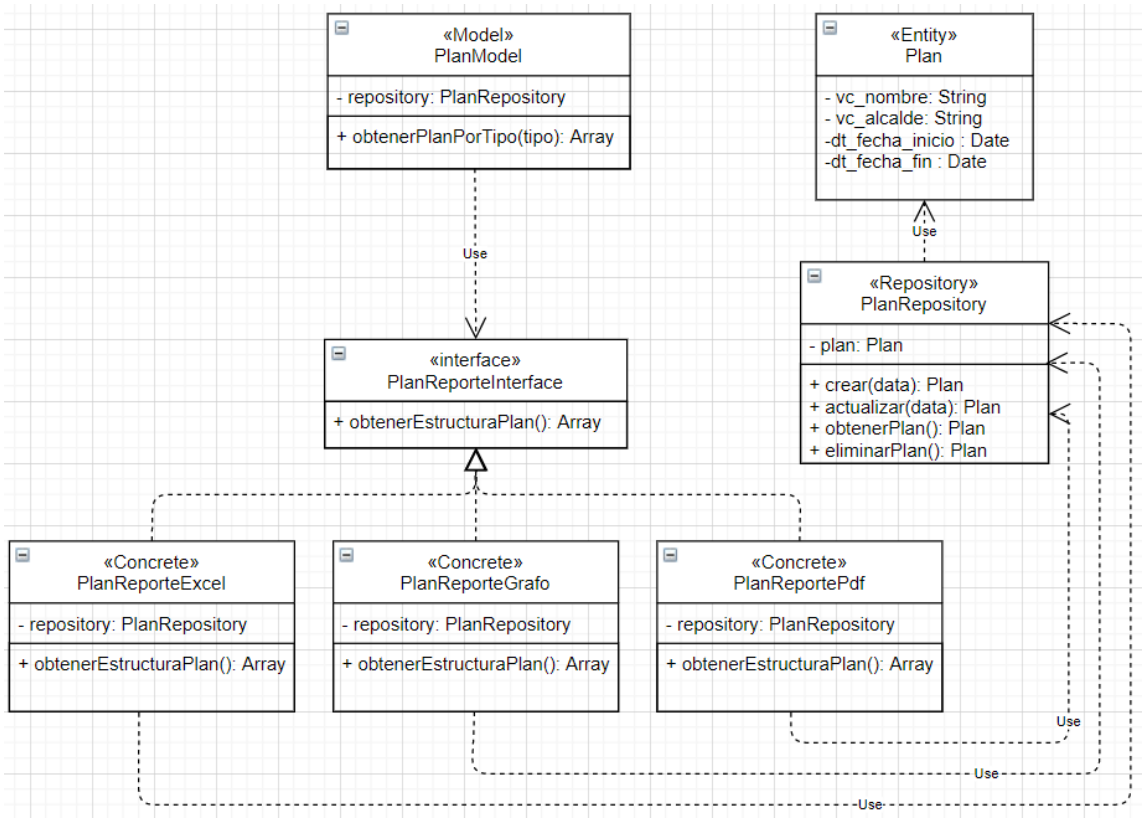

Figura 12 Ejemplo de solución a una implementación que incumple el OCP.

Desde la clase PlanModel se determinará qué clase concreta usar por medio de la vista, para este fin se hace uso de un archivo de configuración del framework de referencia en la entidad (Laravel) para indexar las clases que se deseen crear, y por otro lado en el método obtenerPlanPorTipo() se instancia un objeto de tipo reporte y se ejecuta el método que implementan cada una de las clases:

```php
#archivo de configuracion: /config/modelos.php
return array(
  'plan'=>'App\Modulos\Plan\Plan',
  'excel'=>'App\Modulos\Plan\PlanReporteExcel',
  'grafo'=>'App\Modulos\Plan\PlanReporteGrafo',
  'pdf'=>'App\Modulos\Plan\PlanReportePdf',
);
```

Figura 13 extracto de un archivo de configuración config de Laravel.

Este archivo de configuración es usado como un conjunto de clave y valor, al invocar la llave, se devolverá el valor que en este caso es el nombre de la clase con su namespace. Con el nombre de la clase en un string es posible instanciar un objeto y luego usar el método.

```php
#Clase PlanModel
public function obtenerPlanPorTipo($vista)
{
    $nombreClaseConcreta = config('modelos.'.$vista);
    $planReporteTipo = new $nombreClaseConcreta;
    return $planReporteTipo->obtenerEstructuraPlan();
}
```

Figura 14 Código de la selección de un algoritmo de acuerdo a la decisión del cliente.

De esta forma en el momento en que soliciten un nuevo tipo de reporte por ejemplo en formato JSON, solo se tendría que agregar una nueva clase que implemente de PlanReporteInterface y luego asociarla en el archivo de configuración para pueda ser accedida desde PlanModel. El resto de código no es necesario tocarlo, así podemos decir que el módulo está abierto a extensiones y cerrado a modificaciones.

Tener en cuenta:

- Este principio ayuda a reusar y mantener el código, haciendo más productivo el proceso de desarrollo y disminuyendo posibles errores de acoplamiento.
- Se considera buena práctica hacer siempre los atributos de las clases de tipo privado (para que no queden abiertos a modificaciones) y no usar variables globales (pues desde otro módulo se pueden modificar).
- Algunos patrones que se pueden usar para aplicar este diseño son: Decorator, Strategy.

- El ejemplo usa el patrón Strategy, pero se puede complementar con un Factory que se encargue de crear las instancias de las clases concretas.

5.2.3 Liskov Substitution: De tal padre tal hijo

Se reconoce como LSP, definido por Barbara Liskov en el cual expresa que: si se tiene una clase hija y una clase padre, se debería ser capaz de reemplazar un objeto de la clase hija, por un objeto de la clase padre, y el programa que utiliza la clase padre debería funcionar sin ningún problema. En otras palabras, una clase hija no debe contrariar (cambiar lógica o reglas de negocio de la clase padre) el comportamiento de una clase padre cuando sobre escribe alguno de sus comportamientos.

Por ejemplo, suponga que existe una jerarquía de clases que modelan los tipos de contratación pública como: Selecciona abreviada, Concurso de méritos y Contratación directa. La clase base llamada TipoContratacion tiene un método que realiza una aprobación y por regla de negocio de la entidad debe mantener un número fijo de aprobaciones el cual no debe cambiar para él ningún tipo de contratación.

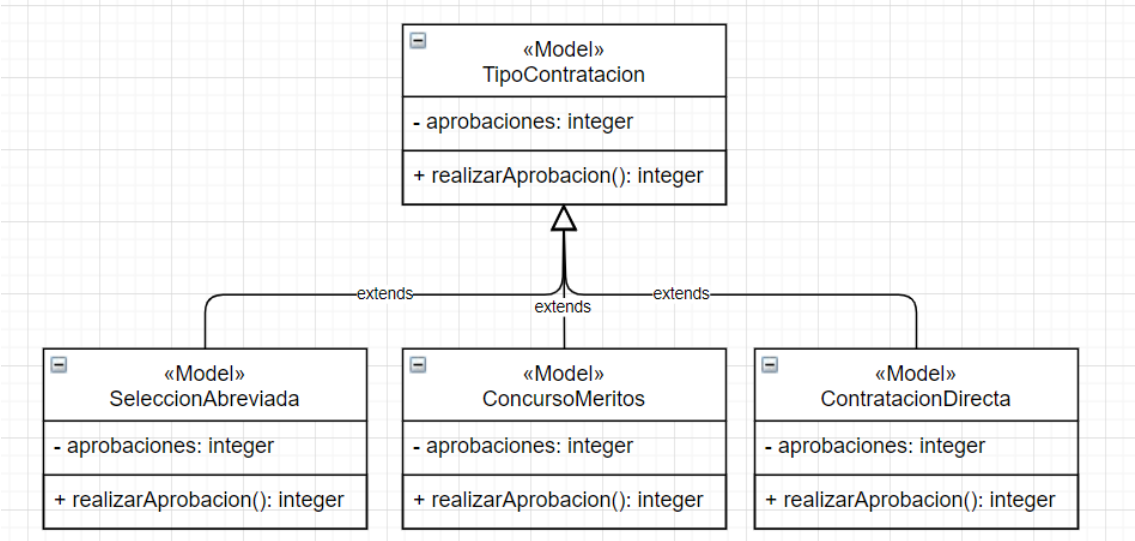

Figura 15 Jerarquía de clases para un tipo de contratación.

por ejemplo, ese número de aprobaciones se configura en un archivo .properties o .env y supongamos que el valor es 3. Este número no solo se usa dentro de las clases de la jerarquía mencionada si no en otro modelo que simula dichas aprobaciones.

Ahora imagine que dentro de una clase hija se modifica el valor de las aprobaciones a un valor diferente que el acordado por lógica de negocio.

```php
<?php

namespace App\Modulos\Contrato\TipoContratacion;


class SeleccionAbreviada extends TipoContratacion
{


    public function realizarAprobacion()
    {
        # Invoca la implementación del método del padre
        parent::realizarAprobacion();
        #y sobreescribe el valor de aprobaciones
        $this->aprobaciones = 4;
    }
}
```

Figura 16 Ejemplo de cómo la Selección Abreviada altera la lógica de 3 aprobaciones para el tipo de contratación.

De acuerdo con el principio cualquier hijo debe conservar el comportamiento (en este caso determinado por lógica de negocio) y en este caso el cambio del valor de 3 a 4 en la variable aprobaciones puede causar inconsistencias en otras clases que llaman al padre y espera que solo se tenga 3 aprobaciones.

```php
<?php

namespace App\Modulos\Contrato\TipoContratacion;


class TipoContratacionModel
{
    const CONSTANTE_APROBACIONES = 3;

    public function tiposContratacion()
    {
        $tipoContratacion = new TipoContratacion(CONSTANTE_APROBACIONES);
        $tipoContratacion->realizarAprobacion(); #aqui aprobaciones == CONSTANTE_APROBACIONES

        $tipoContratacion = new SeleccionAbreviada(CONSTANTE_APROBACIONES);
        $tipoContratacion->realizarAprobacion();  #aqui aprobaciones != CONSTANTE_APROBACIONES
    }
}
```

Figura 17 Implementación del ejemplo donde se rompe el LSP.
Se evidencia que se rompe el principio al sobre escribir el comportamiento de la clase padre desde la clase hijo.

La manera de solucionar este error es simplemente garantizar que las clases hijas no alteran el comportamiento de la clase padre dadas unas reglas de negocio.

Tener en cuenta:

- Cuando se realiza herencia hay que garantizar un comportamiento coherente entre clases padres e hijas.
- Dejar vacío un método de clase padre en una clase hijo es una manera de romper fácilmente el principio LSP.
- Lanzar excepciones desde la clase hija no conocidas en la clase padre es una manera de romper el principio LSP.
- Se debe mantener la compatibilidad entre métodos de clases padres e hijas. Una manera de romper el principio es por ejemplo documentar que el método realizarAprobación es deprecated para la clase ContratacionDirecta.
- Al incumplir con este principio además se incumple con el OCP.
- Cumplir con el principio es garantizar la coherencia de la jerarquía que se modelo.

5.2.4 Interface Segregation: No depender de lo que no se necesita

El principio advierte que no se debe implementar una interfaz que cuyos métodos no se van a utilizar en su totalidad (ejemplo: se implementa el método, pero no contiene código o lanza una excepción). Esto puede causar efectos contraproducentes pues más adelante en la misma o en otra aplicación se puede intentar ingresar a ese comportamiento que reglamenta la interfaz y no obtener lo deseado.

Para aterrizar el concepto con un ejemplo, Imagine que se crea una interfaz para reglamentar que todas las clases de tipo repositorio implementen las acciones CRUD básicas más una de obtener todos los registros y otra de pasar los datos a formato dataTable (plugin de javascript para ver tablas). El diagrama sería algo así:

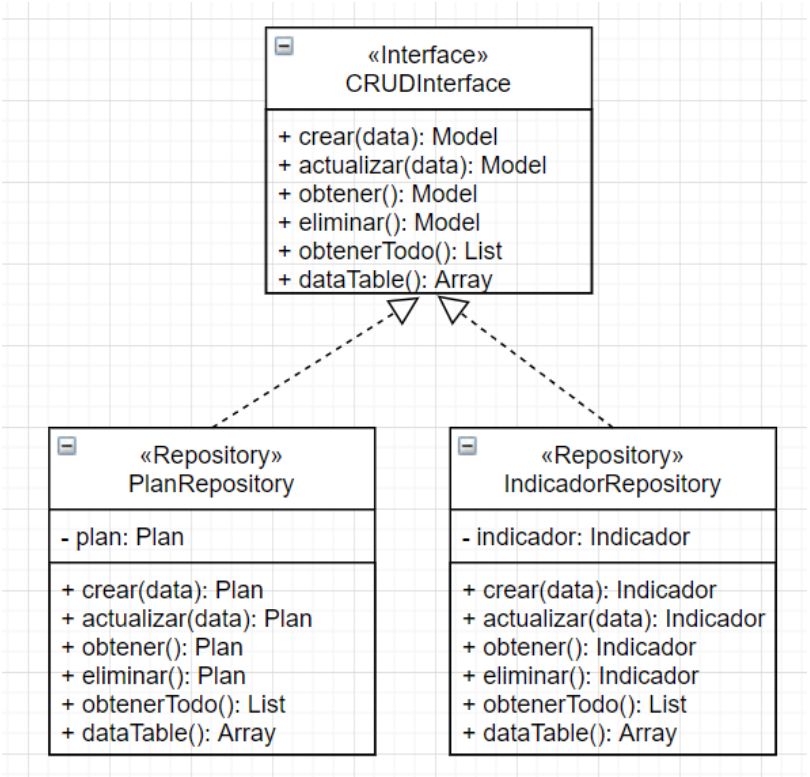

Figura 18 Diagrama de clases Repository que implementan métodos que no usan.

Ahora considere que en la clase PlanRepository no se requiere el método obtenerTodo ni el de dataTable, nuestro código se vería así:

```php
<?php

namespace App\Modulos\Plan\Repository;

use App\Modulos\Plan\Plan;

class PlanRepository implements CrudInterface{

    public function crear($request){
        $plan = new Plan;
        return $this->procesar($plan, $request);
    }

    public function actualizar($request, $id){
        $plan = Plan::find($id);
        return $this->procesar($plan, $request);
    }

    public function procesar($plan, $request)
    {…
    }

    public function obtener($id, $relaciones = []){
        return Plan::with($relaciones)->find($id);
    }

    public function eliminar($id){}

    public function obtenerTodo($relaciones = []){
        return Plan::where('i_estado',1)->get();
    }

    public function dataTable($relaciones = []){}
}
```

Figura 19 Clase Repository que implementan métodos que no usan.

En este caso dichas funciones se declaran por norma de la interfaz, pero no se implementan. Esto a simple vista puede que no genere inconsistencias, pero imagine que su conjunto de clases se ha convertido en un módulo API y que alguien usa el método dataTable de la clase en PlanRepository, en ese momento al artefacto que haga las veces de cliente de ese servicio obtendrá un valor no esperado.

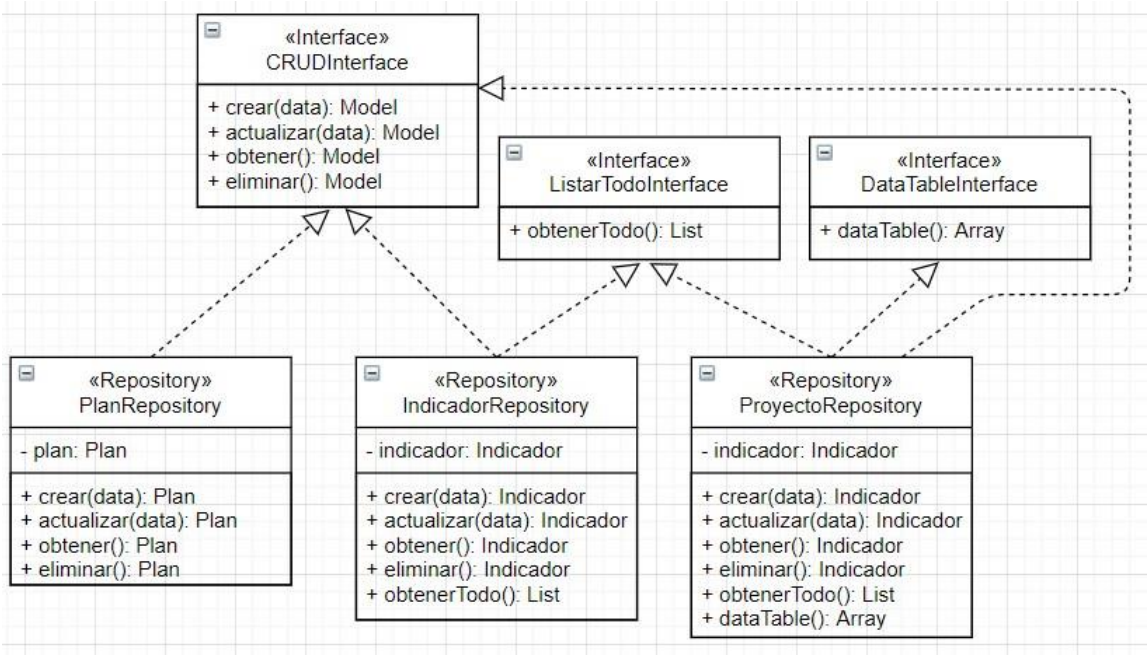

Figura 20 Clases Repository que no implementan métodos innecesarios.

Note que, al aplicar el principio, se crearon nuevas interfaces segregando el comportamiento y se mantuvo la reusabilidad del código pues de izquierda a derecha se aprecia que los repositorios implementan una, dos y tres interfaces de acuerdo con sus necesidades.

Tener en cuenta:

- Las interfaces que se crean no deberían tener métodos que no necesitan las clases que la implementen.
- Las clases concretas no deberían depender de métodos que no utilicen.
- Un diagnóstico inicial para detectar que no se cumple con ISP es validar si SRP tampoco se cumple.
- Tanto las clases abstractas como las interfaces deben ser muy estables, es decir no deben cambiar mucho en el tiempo.

5.2.5 Dependency Inversion: Depender de lo abstracto y no de lo concreto

Este principio habla de dos cosas: los módulos de alto nivel no deberían depender de módulos de bajo nivel, en otras palabras, debe depender de abstracciones. El módulo de alto nivel no debería tener conocimiento explícito de un módulo de bajo nivel.

Considere de alto nivel una clase abstracta o interfaz que inicia una jerarquía, y como bajo nivel a una clase (hija, subtipo, concreta) las cuales son equivalentes o incluso una que no pertenece a una jerarquía (no implementa ni extiende). De nuevo para entender cómo y en donde se aplica este principio, se muestra el siguiente ejemplo:

Retomando el ejercicio del SRP y OCP, en donde se tenía un modelo que por un lado accedía a la base de datos por medio de un repositorio y por otro lado una clase determinaba qué tipo de reporte generar (Excel, Grafo o PDF). Por el momento se hace foco en la relación entre las clases que llaman al repositorio.

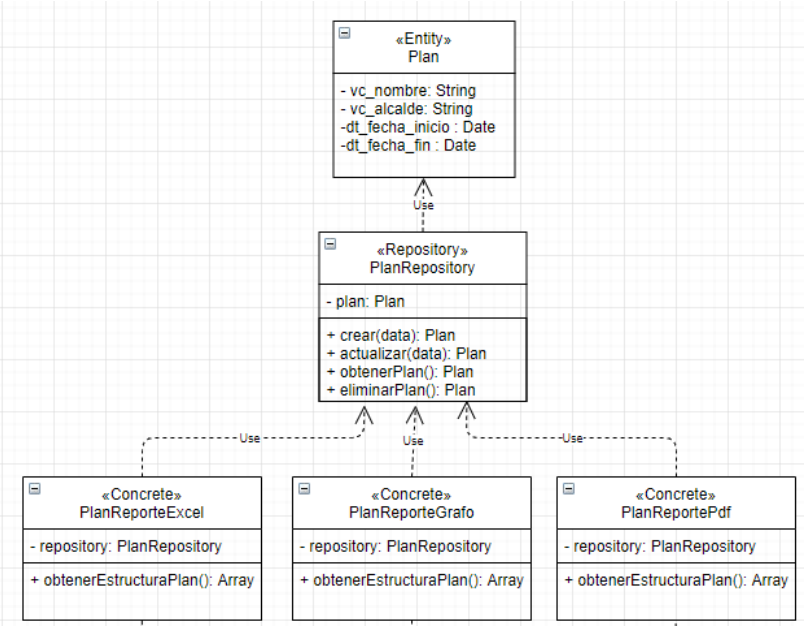

Figura 21 Clases que dependen de una clase concreta.

Suponga que la entidad Plan modela una tabla de una base de datos relacional como MariaDB u Oracle. Pero ahora el requerimiento es migrar a una base no relacional como MongoDB. Esto significa

que se debe crear una clase PlanCollection para representar la colección de mongo que almacenará los datos. Al mismo tiempo, asuma que los métodos en PlanRepository fueron creados para gestionar los datos por SQL y no por el ORM, por lo tanto, también sería necesario crear un repositorio que solvente la interacción con MongoDB.

Como las clases (PlanReporteExcel, PlanReporteGrafo y PlanReportePdf) dependen de una clase de bajo nivel. En el momento en que cambia el requerimiento de base de datos, se afectan las clases directamente. pues se tendría que cambiar el repositorio y en el peor de los casos el nombre de los métodos que acceden a los datos. La solución a este problema es depender de algo de más alto nivel o abstracto como una interfaz o clase abstracta en lugar de una clase concreta. Por lo tanto, se plantea crear una jerarquía para los repositorios, estableciendo una regla para el comportamiento (métodos) de las clases que implementen:

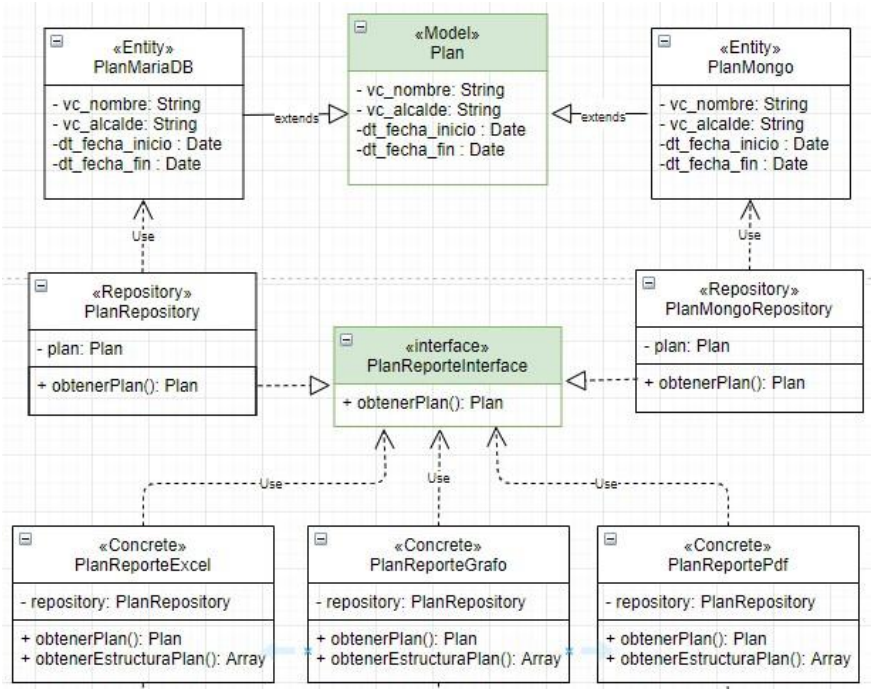

Figura 22 Clases que dependen de una abstracción, es decir una interfaz.

Las clases resaltadas con color verde son la clase e interfaz que abstraen o inician la jerarquía y por lo tanto se considera de alto nivel.

Con este ajuste las clases que usan el repositorio no les importa si los datos vienen de Mongo o de MariaDB porque solo dependen de un método llamado obtenerPlan (establecido por contrato en la interfaz) que le proporciona el insumo para construir su lógica. De igual forma se crean las clases PlanMongo y PlanMariaDB dado a que extienden de Plan que será la clase general con la que las clases de repositorios realmente interactúan.

Note además que se ha aplicado el principio de ISP, pues la interface PlanReporteInterface no solicita implementar métodos CRUD, solo los de consulta, anteriormente en las clases de reporte, aunque no se usaban estaban disponibles cuando usaban directamente la clase PlanRepository inicial.

Ahora bien, pasando del concepto a la práctica en un framework, el cambio de un manejador de base de datos relacional por uno no relacional no es tan traumático si se usan librerías, por ejemplo, en Laravel se puede usar:

composer require jenssegers/mongodb

Luego en las tablas Entity (o Model para laravel) se cambia solo la clase de la cual se extiende:

PlanMariaDB	PlanMongoDB

```php
<?php

namespace App\Modulos\Plan;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;
class Plan extends Model
{
    use SoftDeletes;

    protected $table = 'tbl_plan';
    protected $dates = ['deleted_at'];
    protected $primaryKey= 'i_pk_id';
    protected $fillable = [
        'vc_nombre',
        'vc_alcalde',
        'vc_sector',
        'vc_acuerdo',
        'tx_objetivo',
        'tx_proyecto_estrategico',
        'tx_observaciones',
        'd_fecha_inicio',
        'd_fecha_fin',
        'i_estado',
        'tx_mision',
        'tx_vision',
        'i_fk_id_tipo',
        'i_principal',
        'd_fecha_implementacion' ,
        'i_clonar'

    ];
```

```php
<?php

namespace App\Modulos\Plan;

use Jenssegers\Mongodb\Eloquent\Model as MongoModel;
use Illuminate\Database\Eloquent\SoftDeletes;
class Plan extends MongoModel
{
    use SoftDeletes;

    protected $table = 'tbl_plan';
    protected $dates = ['deleted_at'];
    protected $primaryKey= 'i_pk_id';
    protected $fillable = [
        'vc_nombre',
        'vc_alcalde',
        'vc_sector',
        'vc_acuerdo',
        'tx_objetivo',
        'tx_proyecto_estrategico',
        'tx_observaciones',
        'd_fecha_inicio',
        'd_fecha_fin',
        'i_estado',
        'tx_mision',
        'tx_vision',
        'i_fk_id_tipo',
        'i_principal',
        'd_fecha_implementacion' ,
        'i_clonar'

    ];
```

Figura 23 Ejemplo de una clase entidad de base de datos relacional, vs base de datos no relacional.

Y luego en las clases repositorio si se usa el ORM Eloquent, las consultas no deberían cambiar.

Tener en cuenta:

- Por otro lado, las abstracciones no deben depender de los detalles o clases concretas.
- Los detalles deben depender de las abstracciones.
- Las abstracciones (creación de interfaces) se crean cuando sus implementaciones son volátiles.
- No se debe heredar de clases volátiles.

5.3 CONCEPTOS DE PRINCIPIOS APIE CON SOLID

En las secciones anteriores se explicó cómo usar los principios de POO APIE y luego cómo implementar los principios SOLID. En este apartado se darán recomendaciones acerca de cómo pensar en los principios APIE teniendo en cuenta los conceptos adquiridos con SOLID.

5.3.1 Abstracción: programar sobre abstracción

Es mejor que la lógica del negocio dependa de una interfaz o clase abstracta que de una clase concreta dado que esto permitirá tener una mayor flexibilidad ante un cambio.

5.3.2 Polimorfismo: mismo nombre distintos resultados

Al realizar polimorfismo por sobreescritura es importante que la clase hija que sobre escribe el método de la clase padre, no altere el comportamiento esperado definido por la clase padre. Al hacer esto incumple el principio LSD.

5.3.3 Encapsulación: encapsular lo que varía

Cuando se enfrenta a un problema en el cual parte del código fuente cambia con una frecuencia alta, es importante aislar ese componente en clases con métodos específicos de tal manera que no se encuentren dispersos por toda la aplicación.

En términos generales en las clases se debe dar acceso solo a lo estrictamente necesario y lo demás ocultarlo para las demás clases del código. En la medida en que se proteja lo que no se necesita exponer, se logra evitar un alto acoplamiento lo cual genera dependencias de tipo: “cuando se sube un cambio se altera una funcionalidad de producción”.

5.3.4 Herencia: composición antes que herencia

Siempre que tenga sentido es mejor hacer composición que hacer herencia, la composición es una relación tipo “tiene un”. Al realizar herencia se genera una serie de retos como: evitar que cuando una clase hija no requiera de toda la estructura de la clase padre, no se debería usar la herencia, al heredar la clase padre pierde su encapsulamiento, además al diseñar la clase hija hay que cuidar que no altere el comportamiento de la clase padre (rompe el principio de sustitución de Liskov).

5.4 APLICACIÓN DE PATRONES DE DISEÑO

Se orientará esta guía de aplicación de patrones por medio de la clasificación clásica del libro de patrones de diseño de la pandilla de 4 o GOF (Gang of Four) que hace alusión a los autores del libro. En este libro se dan a conocer 23 principios, no obstante, solo se mencionará los patrones que el equipo de fábrica de software ha trabajado y espera que a medida que haya uso y apropiación de estos los diferentes desarrolladores categoría junior, semi junior y senior aporten a esta sección.

Los patrones aquí mencionados, encajan en el lenguaje de referencia de la entidad que es PHP y en la arquitectura de software que es la hexagonal, ambos planteados en la política de desarrollo de software. Sin embargo, no quiere decir que solo se deban utilizar los patrones aquí mencionados, cómo se indicó anteriormente los patrones describen la solución a un problema recurrente. Por lo que el criterio y la experiencia del desarrollador determina que patrón adaptar en cada caso.

Se mencionan patrones que no hacen parte del conjunto de 23 patrones originales, que se organizará de acuerdo con la clasificación, como se detalla a continuación:

5.4.1 Patrones Creacionales

Contienen técnicas que ayudan a centralizar el proceso de la creación de objetos, a continuación, se mencionan los principales:

5.4.1.1 Simple Factory

Para algunos, Simple Factory no es un patrón en sí mismo, es más un concepto que los desarrolladores necesitan saber antes de saber más sobre el método Factory y el método Abstract Factory. Este patrón ayuda a crear objetos de diferentes tipos en lugar de la instanciación directa de objetos.

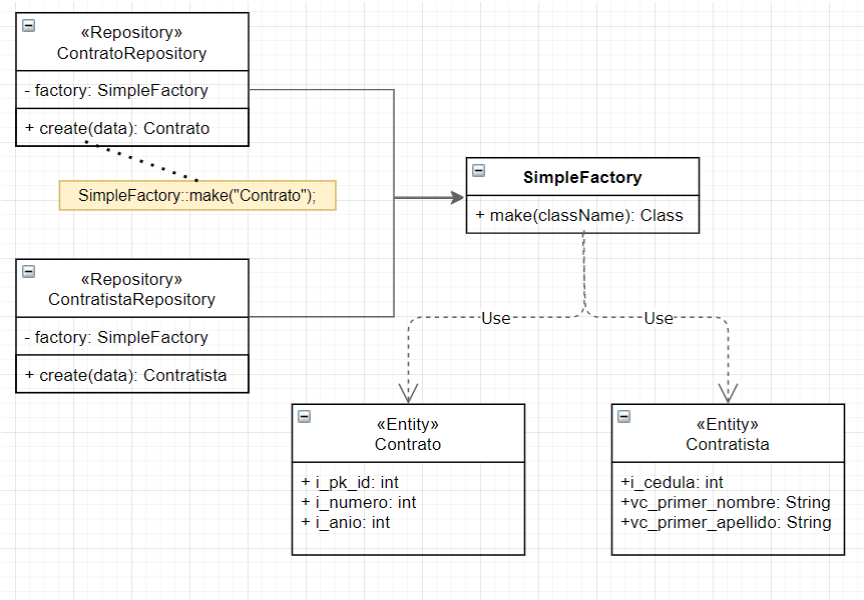

Figura 24 Diagrama de clases del patrón SingleFactory

Esta clase sería tan sencilla como:

```php
<?php

namespace App\Modulos\Factory;
class SimpleFactory
{
    public static function make($class) {
        return new $class();
    }
}
```

Figura 25 contenido del método make del SingleFactory

y su llamado se hace de forma estática como se aprecia a continuación:

```php
use App\Modulos\Factory\SimpleFactory;
use App\Modulos\Estado\Estado;

class ProyectoMetaSeguimientoActividadCuadriculaExcel implements ProyectoMetaSeguimientoCuadriculaInterface
{

    private $repository;
    private $caracteristicaRepository;
    private $datos;

    public function __construct(){
        $this->repository = SimpleFactory::make(config('modelos.desagregado_actividad_repository'));
    }
```

Figura 26 uso del SingleFactory desde una clase.

Importante mencionar que lo que recibe el método make es un string que contiene el nombre de la clase a crear. En este caso se hace uso de los archivos clave - valor de configuración que provee Laravel para organizar el diccionario de clases que se pueden crear desde este patrón.

Finalmente, en Laravel se puede usar una fábrica que viene en la clase App, cómo se aprecia a continuación:

```php
class ProyectoMetaSeguimientoActividadCuadriculaExcel implements ProyectoMetaSeguimientoCuadriculaInterface
{
    private $repository;
    private $caracteristicaRepository;
    private $datos;

    public function __construct(){
        $this->repository = \App::make(config('modelos.desagregado_actividad_repository'),['variable' => 1]);
        //$this->repository = SimpleFactory::make(config('modelos.desagregado_actividad_repository'));
    }
```

Figura 27 uso del Facade de creación de Laravel

Esta clase al igual que Simple Factory tiene un método estático make que puede recibir además un arreglo con los parámetros que se inyectan en el constructor de la clase.

Tener en cuenta:

- Es un buen inicio para quitarle la responsabilidad a las clases que crean objetos cuya creación es simple.
- Para usar este patrón las clases que son usadas por la fábrica deben compartir la forma en que se instancia un nuevo objeto.

5.4.1.2 Singleton

El objetivo de usar este patrón es asegurarse que una clase tiene sólo una instancia y que esta proporciona un acceso global para acceder a ella. Se debe establecer su estado (atributos) en el primer uso.

El problema que soluciona este patrón es: el sistema necesita una y solo una instancia de un objeto. Además, se requiere que el acceso global sea requerido.

Singleton debe considerarse solo si se cumplen los siguientes criterios:

- La propiedad de la instancia única no puede asignarse razonablemente.
- La inicialización perezosa es deseable.
- El acceso global no se proporciona de otra manera.

Si la propiedad de la instancia única, cuándo y cómo se produce la inicialización y el acceso global no son problemas, Singleton no es lo suficientemente interesante.

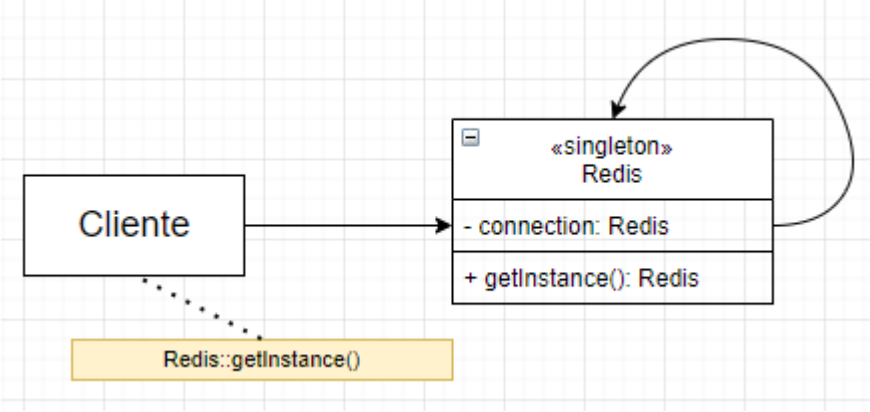

Figura 28 Diagrama de clases del patrón singleton

En la práctica, aunque se puede crear una clase de acuerdo al diagrama anterior, en Laravel se hace uso del Service Container y del Service Provider para implementar inyección de dependencias y singleton.

Para comprender el concepto de Service Container y Provider, imagine que el Container es una bolsa en donde guarda objetos y el Provider le proporciona la manera de identificar los objetos, y de qué cómo acceder a ellos antes de guardarlos en la bolsa del contenedor.

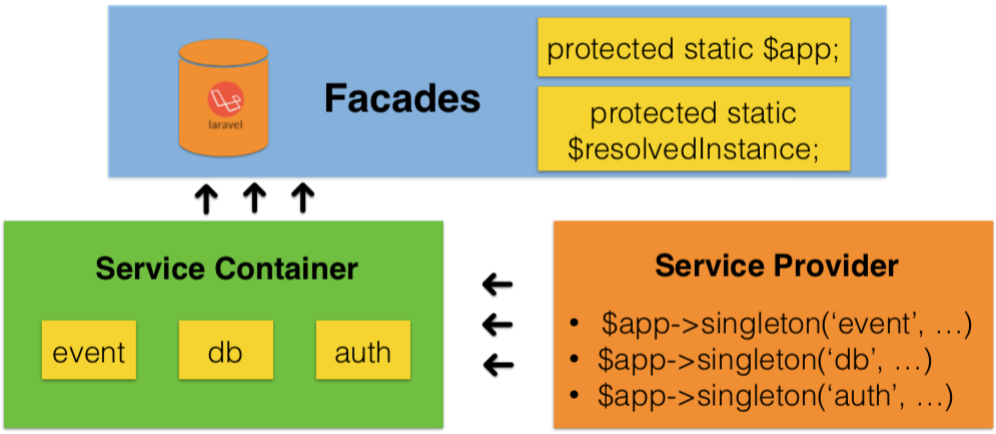

Figura 29 Diagrama de interacción entre Facade, Service Container y Service Provider en Laravel

Existen dos maneras de registrar una clase, por el método bind (crea un espacio en memoria cada que se llama desde una fachada, o singleton el cual se inicializa desde el ingreso al contenedor y luego queda disponible en cualquier parte del framework.

La manera en que laravel hace esto es creando un service provider:

`php artisan make:provider RedisServiceProvider`

El comando anterior creará un archivo, y allí agregando en el método register la instrucción del singleton, el primer parámetro es el nombre de la clase que se instancia (puede ser una interfaz) y luego un closure que retorna la instancia de la clase. Aquí se ha registrado para el Service Container y se le ha dado un nombre.

```php
<?php

namespace App\Providers;

use App\Services\Redis\Redis;
use Illuminate\Support\ServiceProvider;

class RedisServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Redis::class, function ($app) {
            return new Redis(config('Redis'));
        });
    }
}
```

Figura 30 Código de clase Service Provider usando Singleton

No obstante, hace falta registrar esa clase ServiceProvider en el archivo de configuración de Laravel para que prepare desde el arranque la instancia:

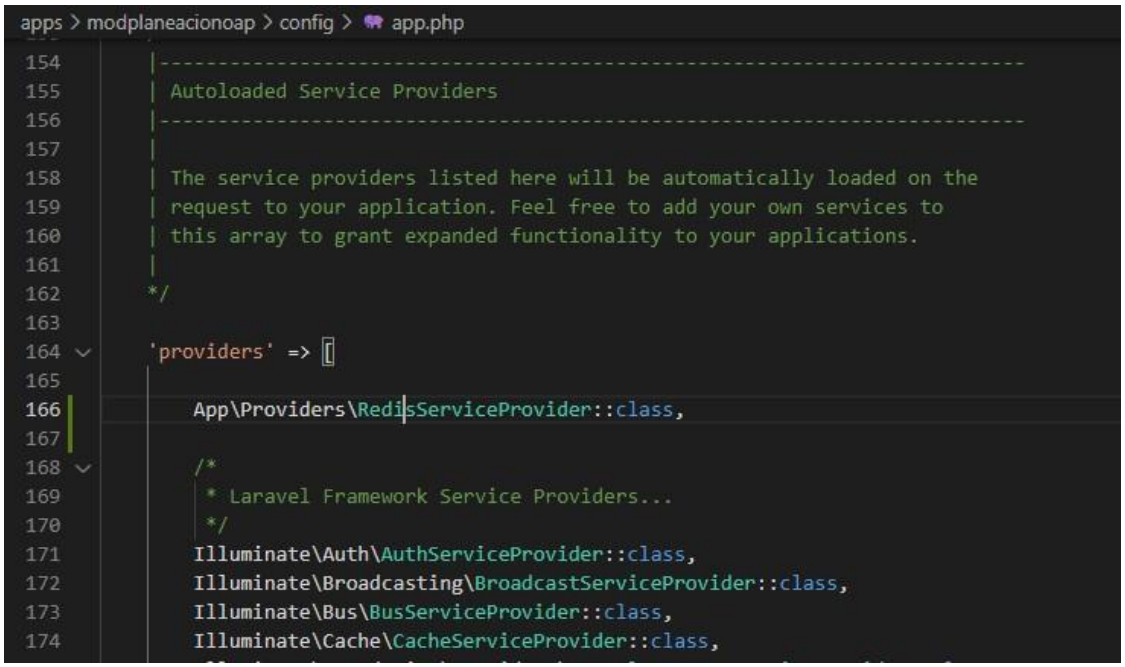

Figura 31 Lugar de registro de Service Provider en el Service Container

La modificación de este archivo es clave pues aquí se encuentra el cargue de todos los ServiceProvider propios del framework.

Finalmente, para acceder a la instancia desde cualquier parte del framework:

```
use App\Services\Redis\Redis;

$redisSingleton = $this->app->make(Redis::class);
# o
$redisSingleton = \App::make(Redis::class);
```

Figura 32 Llamado de un Singleton en Laravel.

Tener en cuenta:

- Suele usarse en los objetos que representan la clase que maneja las conexiones a bases de datos, la clase que maneja el sistema de logs, o en tipos de almacenes de datos como REDIS.
- Una única instancia significa que solo hay un espacio en memoria para el objeto.

5.4.1.3 Inyección de dependencias

La inyección de dependencias, aunque no hace parte del conjunto de patrones GOF, se categoriza en esta sección porque está ligado a la creación de los objetos, aunque no hace esto propiamente.

Cuando una clase A usa otra clase B, lo normal es que en la clase A haya un objeto de tipo B (composición o agregación). Para usar a B, A tuvo que instanciarlo con la típica instrucción de new.

La inyección de dependencias busca que las dependencias lleguen a la clase ya creadas y listas para usar, con el objetivo de quitar responsabilidad de creación sobre los objetos que usa.

En Laravel, la inyección de dependencias se hace de manera automática en los constructores de la clase controlador, aunque cómo se mostró en el ejemplo del singleton, puede fácilmente ser usado en otra parte.

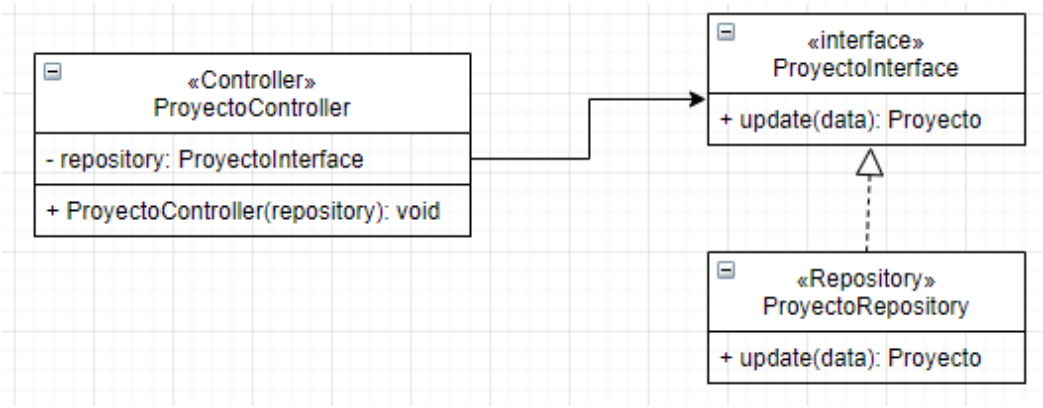

Figura 33 Relación del controlador con un repositorio por medio de una interfaz inyectada .

Bajo este modelo el controlador usa el repositorio sin conocer a detalle ni depender de su implementación, esto es clave para los principios de POO como se mencionó anteriormente,

Ahora en el Service Provider hay que nombrar a la interfaz, pero instanciar a su implementación, en este caso la clase Repositorio.

```php
<?php

namespace App\Modulos\Proyectos\Preformulacion\Providers;

use Illuminate\Support\ServiceProvider;
use App\Modulos\Proyectos\Preformulacion\Repository\ProyectoPreformulacionRepository;

class ProyectoPreformulacionServiceProvider extends ServiceProvider
{

    public function boot()
    {

    }

    public function register()
    {
        $this->app->bind(config('modelos.proyecto_preformulacion_interface'), function($app)
        {
            return new ProyectoPreformulacionRepository;
        });
    }

}
```

Figura 34 Código de un Service Provider que registra un repositorio en el Service Container enlazando la entidad.

Note que el método en el register es bind y no singleton, como en el ejemplo anterior.

```php
<?php

namespace App\Modulos\Proyectos\Preformulacion\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Modulos\Proyectos\Preformulacion\Repository\ProyectoPreformulacionInterface;

class ProyectoPreformulacionController extends Controller
{

    private $proyectoPreformulacionRepository;

    public function __construct(
        ProyectoPreformulacionInterface $proyectoPreformulacionRepository){
        $this->proyectoPreformulacionRepository = $proyectoPreformulacionRepository;
    }
    public function update(Request $request, $id)
    {
        $proyecto = $this->proyectoPreformulacionRepository->actualizar($request,$id);
        return response()->json($proyecto);
    }
}
```

Figura 35 Código de la inyección del repositorio por el constructor del controlador.

Finalmente se aprecia en el constructor del controlador cómo se inyecta el objeto de tipo interfaz que puede ser implementado por cualquier otra clase que cumpla con el contrato de la interfaz.

Tener en cuenta:

- La inyección en los controladores de Laravel se puede hacer directamente en el constructor. Para otras clases es necesario inyectar el objeto por parámetro de un método.
- Es importante siempre que el ServiceProvider dependa de una interfaz, esto permitirá una mejor flexibilidad en el código.

5.4.2 Patrones Estructurales

5.4.2.1 Facade

Este patrón tiene como finalidad ocultar un subsistema detrás de una clase simple. Facade analiza la encapsulación de un subsistema complejo dentro de un único objeto de interfaz. Esto reduce la curva de aprendizaje necesaria para aprovechar con éxito el subsistema. También promueve el desacoplamiento del subsistema de sus potenciales clientes. Por otro lado, si la fachada es el único punto de acceso para el subsistema, limitará las funciones y la flexibilidad que los "usuarios avanzados" puedan necesitar.

El patrón persigue los siguientes objetivos:

- Definir una interfaz de nivel superior que facilite el uso del subsistema.
- Envolver un subsistema complicado con una interfaz más simple

Pasos para crear una fachada.

- Identifique una interfaz unificada más simple para el subsistema o componente.
- Diseñe una clase 'contenedora' que encapsule el subsistema.
- La fachada / envoltura captura la complejidad y las colaboraciones del componente y delega en los métodos apropiados.
- El cliente utiliza (está acoplado) únicamente a la fachada.
- Considere si las fachadas adicionales agregarían valor.

Entrando en la práctica, imagine que la entidad tiene pensado realizar un módulo para gestionar las notificaciones de los sistemas de información. Se notificarán tanto a usuarios registrados como externos, y por el momento hay 3 tipos de notificaciones (correo, SMS y Slack) pero no se descartan otras como push en una futura app.

Aplicando el principio de encapsular lo que varía, se identifica que el tipo de notificación es volátil y para ello se usan el patrón Strategy o estrategia para permitir agregar funcionalidades posteriormente sin alterar el código ya escrito.

Por otro lado, hay que gestionar acceso a base de datos por medio de repositorios para Usuario y Notificación. Todo lo descrito anteriormente es el subsistema. Ahora bien, para efectos de uso por parte de otros módulos, incluso APIS, se simplificará por medio de la fachada NotificacionFacade

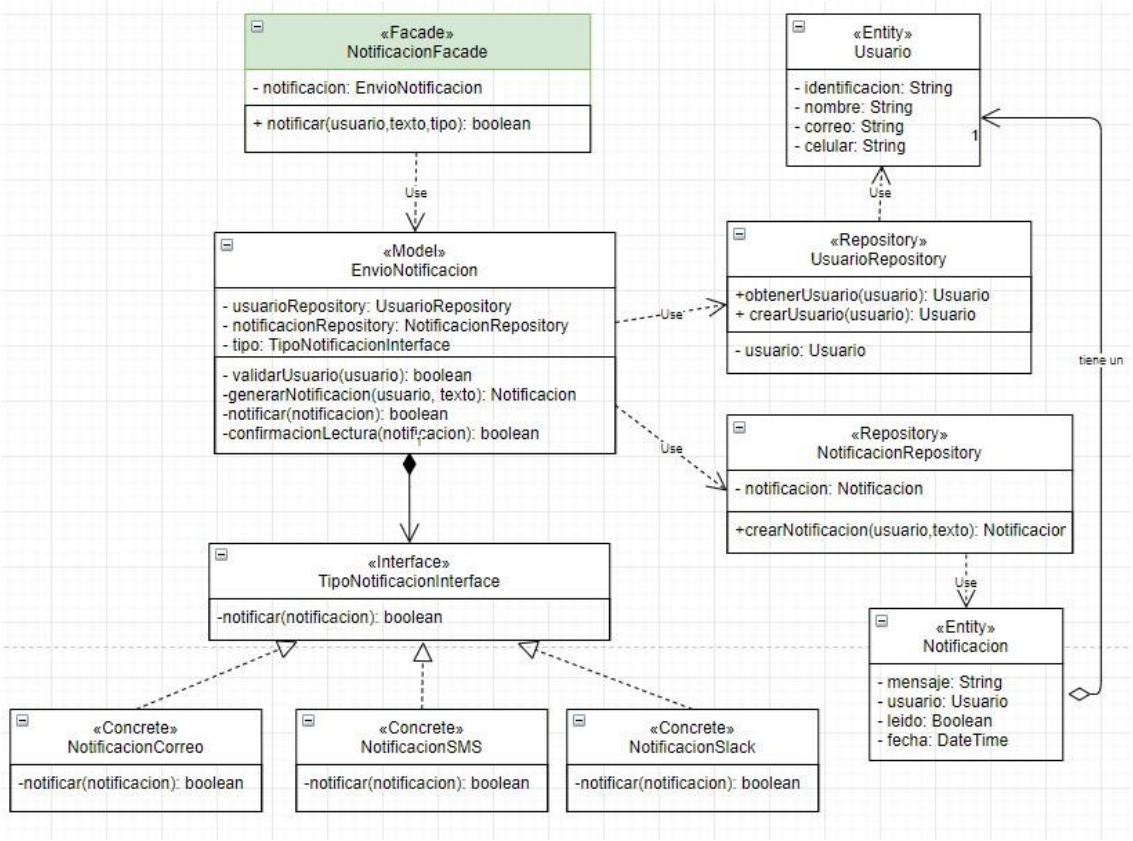

Figura 36 Ejemplo del patrón Facade, encapsulando toda la lógica en una simple clase.

En cuanto a Laravel, se podría decir que hace uso fuertemente de este patrón y como usuario del framework accedemos a sus funcionalidades por medio de fachadas. Por ejemplo, cada vez que retornemos una vista "view", accedemos al almacenamiento "Storage", al usuario autenticado "Auth", a la conexión de base de datos "DB" estamos usando una fachada del framework.

En Laravel las fachadas se registran en el archivo config/app.php como se aprecia en la imagen.

```
# archivo de configuración config/app.php
'aliases' => [
    'App' => Illuminate\Support\Facades\App::class,
    'Artisan' => Illuminate\Support\Facades\Artisan::class,
    'Auth' => Illuminate\Support\Facades\Auth::class,
    'Blade' => Illuminate\Support\Facades\Blade::class,
    'Broadcast' => Illuminate\Support\Facades\Broadcast::class,
    'Bus' => Illuminate\Support\Facades\Bus::class,
    'Cache' => Illuminate\Support\Facades\Cache::class,
    'Config' => Illuminate\Support\Facades\Config::class,
    'Cookie' => Illuminate\Support\Facades\Cookie::class,
    'Crypt' => Illuminate\Support\Facades\Crypt::class,
    'DB' => Illuminate\Support\Facades\DB::class,
    'Eloquent' => Illuminate\Database\Eloquent\Model::class,
```

Figura 37 arreglo de Facades de Laravel en el archivo app.php.

Lo usamos a menudo, solo que los usamos a través de las funciones Helper de Laravel:

```
#Acceso directo al Facade
return View::make('admin.profile', $data);
#Acceso mediante el Helper
return view('admin.profile', $data);
```

Figura 38 Distintas formas de llamar a un Facade en Laravel, por acceso directo a la clase y por Helper.

Tener en cuenta:

- Los objetos Facade son Singleton, dado a que solo se requiere uno para toda la aplicación.
- Facade define una nueva interfaz, mientras que Adapter usa una interfaz antigua. Recuerde que Adapter hace que dos interfaces existentes funcionen juntas en lugar de definir una completamente nueva.

5.4.3 Patrones de Comportamiento

5.4.3.1 Strategy

El patrón estrategia define una familia de algoritmos y los organiza en clases separadas. Permite que se agreguen nuevos algoritmos sin afectar el código existente y al mismo tiempo hacerlos intercambiables en tiempo de ejecución.

El principal problema que resuelve este patrón está relacionado con el principio de abierto / cerrado OCP, pues es muy común que existan funcionalidades en el código que requieran extensión en el futuro, por ejemplo: agregar un nuevo método de pago, agregar un nuevo formato para el reporte, calcular los impuestos dependiendo de una nueva región o país entre otras reglas de negocio. Estas reglas se convierten en condicionales como if o switch que se pueden encontrar en diferentes partes del sitio y al agregar una nueva condición, se debe ir a modificar lo mismo en varias partes.

Cómo se mencionó anteriormente el principio “encapsular lo que varía” hace referencia a detectar aquella parte del código que es volátil o susceptible a cambios y aislarlo o separarlo. Con aislar se hace referencia a pasar esa lógica a una clase puntual, pero esto no es suficiente cuando se requiere ejecutar un algoritmo dependiendo de la decisión del usuario final y es aquí donde entra a jugar el patrón Strategy.

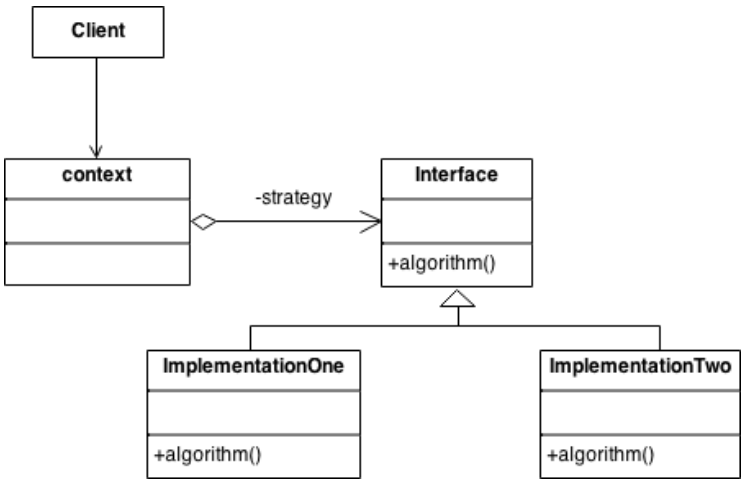

Figura 39 Diagrama de clases inicial del patrón Strategy.

El planteamiento de la solución es:

1. Crear la familia de algoritmos iniciando la jerarquía desde una interfaz.
2. Por cada algoritmo se crea una clase que implementa dicha entidad.
3. Disponer de una clase "contexto" que conozca cual es la estrategia a usar para ejecutarla.

A lo largo de esta guía se ha implementado este patrón por ejemplo en la sección del principio Open / Close OCP. Sin embargo, se compartirá un ejemplo relacionado al módulo de Seguimiento de Proyectos de Pandora en donde se tienen 3 tipos de seguimientos (actividades, estímulos y jurados) que están asociados a las metas de proyecto. El requerimiento es permitir generar desde una sola interfaz cualquiera de los tipos de seguimiento.

Lo que primero se puede venir a la mente es crear una serie de condicionales desde el controlador dependiendo del valor del parámetro que viene desde la vista, como se aprecia a continuación:

```php
public function reporteConsolidadoCuantitativo(Request $request)
{
    $tiposSeguimiento = [];
    if($request->i_vista == "actividades"){
        $tiposSeguimiento = [Estado::CODIGO_META_CUANTITATIVA_ACTIVIDAD, Estado::CODIGO_META_CUANTITATIVA_ASISTENCIA];
        $data = $this->proyectoSeguimientoRepository->generarExcelDesagregadosActividades($request, $tiposSeguimiento);
    }else if($request->i_vista == "estimulos"){
        $tiposSeguimiento = [Estado::CODIGO_META_CUANTITATIVA_ESTIMULO];
        $data = $this->proyectoSeguimientoRepository->generarExcelDesagregadosEstimulos($request, $tiposSeguimiento);
    }else if($request->i_vista == "jurados"){
        $tiposSeguimiento = [Estado::CODIGO_META_CUANTITATIVA_ESTIMULO];
        $data = $this->proyectoSeguimientoRepository->generarExcelDesagregadosJurados($request, $tiposSeguimiento);
    }
    if($data['estado']==='error'){
        $mensaje = [
            'message'=>$data['message'],
            'title'=>'Error al generar documento',
            'type'=>'error'
        ];
        return Redirect::back()->with($mensaje);
    }else{
        $this->consolidadoCuantitativoExport->configurarDocumento();
        $this->consolidadoCuantitativoExport->setDatos($data);
        $this->consolidadoCuantitativoExport->handleExport();
    }
}
```

Figura 40 Programación de una necesidad que aplica varios algoritmos bajo una condición.

El problema es que cada vez que se genera una nueva condición, se va a requerir cambiar el método en el controlador y en cualquier parte del código con esta característica, esto rompe el principio de abierto / cerrado.

Para prevenir estos errores, se puede implementar el patrón estrategia. A continuación, se presenta

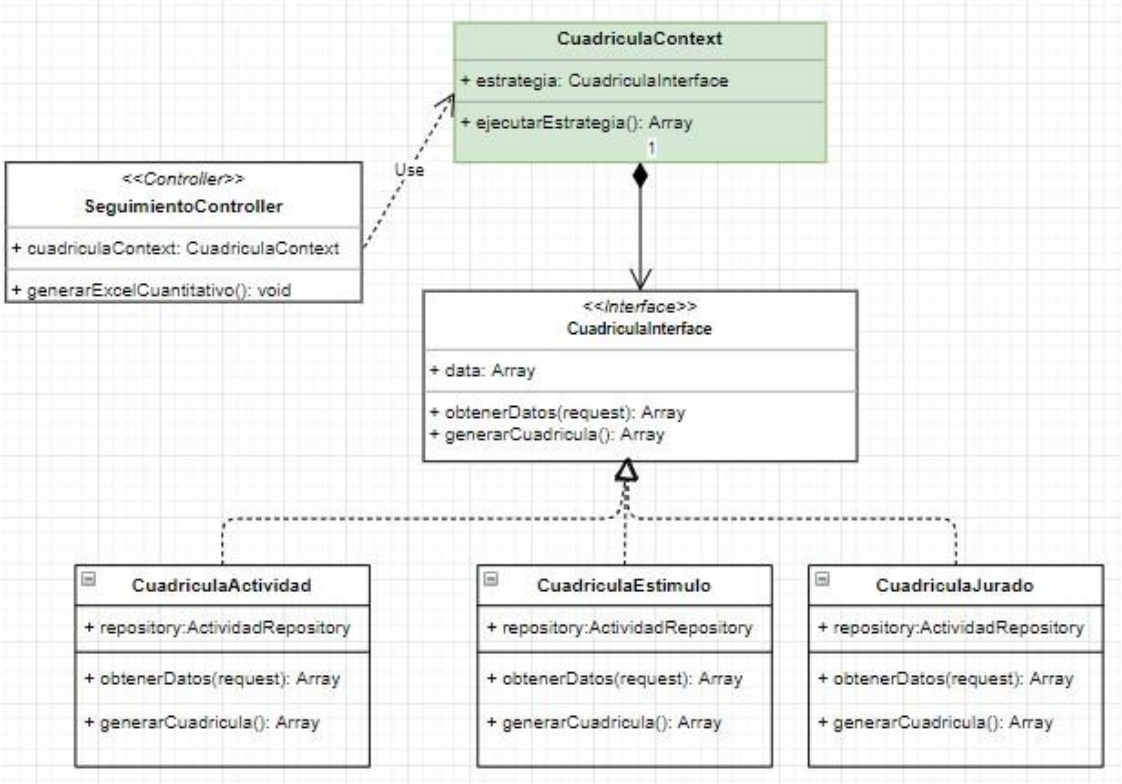

Figura 41 Implementación del patrón estrategia, partiendo del "cliente" Controlador.

Si validamos la lista de chequeo a nivel de código tenemos:

1. Crear la familia de algoritmos iniciando la jerarquía desde una interfaz.

```php
<?php

namespace App\Modulos\Proyectos\Seguimiento\Models\Contracts;

use Illuminate\Support\ServiceProvider;

interface ProyectoMetaSeguimientoCuadriculaInterface
{

    public function obtenerDatos($request);
    public function generarCuadricula();

}

```

Figura 42 Definición de la interfaz que inicia la jerarquía y establece las reglas o métodos que heredan.

2. Por cada algoritmo se crea una clase que implementa dicha entidad.

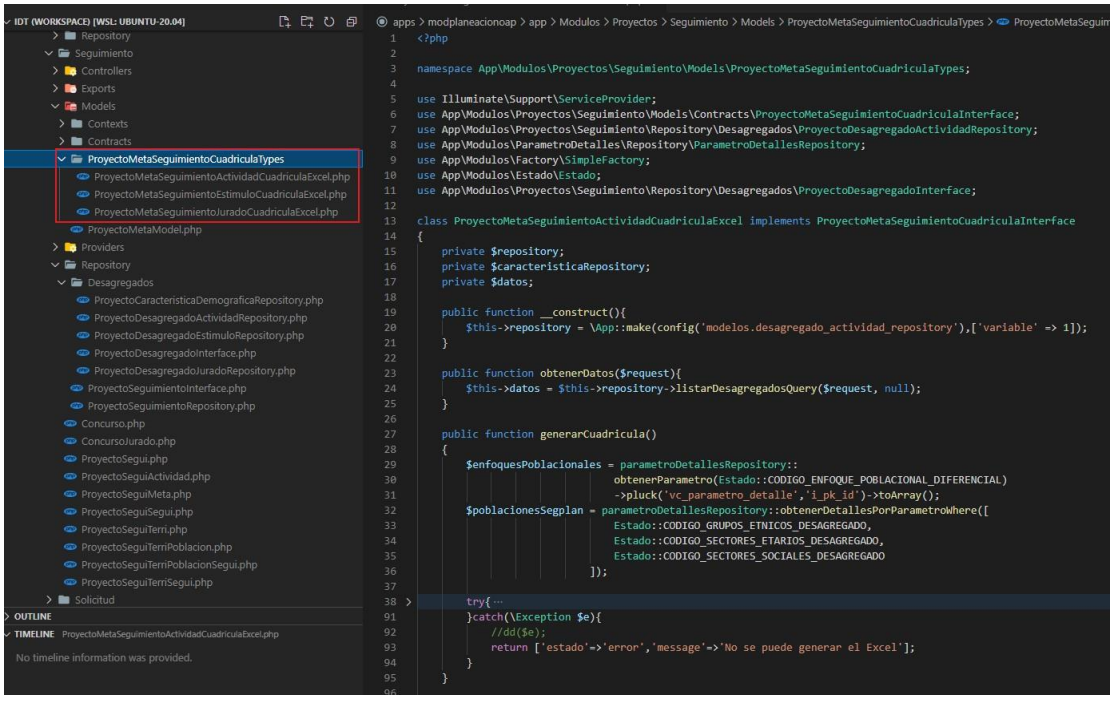

Figura 43 Estructura de clases creadas por cada algoritmo que implementa los métodos de la interfaz .

3. Disponer de una clase “contexto” que conozca cual es la estrategia a usar para ejecutarla.

```php
<?php

namespace App\Modulos\Proyectos\Seguimiento\Models\Contexts;

use Illuminate\Support\ServiceProvider;
use App\Modulos\Proyectos\Seguimiento\Models\Contracts\ProyectoMetaSeguimientoCuadriculaAbstract;

class ProyectoMetaSeguimientoCuadriculaContext
{
    private $estrategia;

    public function asignarEstrategia($estrategia) {
        $this->estrategia = $estrategia;
    }

    public function obtenerEstrategia() {
        return $this->estrategia;
    }

    public function ejecutarEstrategia($request)
    {
        $this->estrategia->obtenerDatos($request);
        return $this->estrategia->generarCuadricula();
    }
}
```

Figura 44 Definición de la clase Contexto que conoce la estrategia y ejecuta el algoritmo.

En este punto se menciona que la estrategia (Clase a ejecutar) viene inicialmente desde el controlador que le inyecta la clase y con esa instancia en el método ejecutarEstrategia ejecuta directamente el método de la clase que contiene el algoritmo. A continuación, se presenta el controlador con el cambio de la implementación de la estratégia y el código anterior se ubica entre comentarios para evidenciar el cambio.

```php
public function reporteConsolidadoCuantitativo(Request $request)
{
    $clase = config('modelos.'.$request->i_vista);
    $this->cuadriculaContext->asignarEstrategia(new $clase);
    $data = $this->cuadriculaContext->ejecutarEstrategia($request);
    /*$tiposSeguimiento = [];
    if($request->i_vista == "actividades"){
        $tiposSeguimiento = [Estado::CODIGO_META_CUANTITATIVA_ACTIVIDAD
        $data = $this->proyectoSeguimientoRepository->generarExcelDesag
    }else if($request->i_vista == "estimulos"){
        $tiposSeguimiento = [Estado::CODIGO_META_CUANTITATIVA_ESTIMULO
        $data = $this->proyectoSeguimientoRepository->generarExcelDesag
    }else if($request->i_vista == "jurados"){
        $tiposSeguimiento = [Estado::CODIGO_META_CUANTITATIVA_ESTIMULO
        $data = $this->proyectoSeguimientoRepository->generarExcelDesag
    }*/

    if($data['estado']==='error'){
        $mensaje = [
            'message'=>$data['message'],
            'title'=>'Error al generar documento',
            'type'=>'error'
        ];
        return Redirect::back()->with($mensaje);
    }else{
        $this->consolidadoCuantitativoExport->configurarDocumento();
        $this->consolidadoCuantitativoExport->setDatos($data);
        $this->consolidadoCuantitativoExport->handleExport();
    }
}
```

Figura 45 Llamado de la clase contexto desde el controlador.

5.4.4 Otros patrones de Diseño

Es importante mencionar que existen otros patrones que solo se mencionan y serán abordados en versiones posteriores de la presente guía. No obstante, se dejan referencias externas desde donde se puede documentar.

Creacionales:

- FactoryMethod: Crea una instancia de varias clases derivadas.
- Object Pool: Evite la adquisición costosa y la liberación de recursos mediante el reciclaje de objetos que ya no están en uso.
- Prototype: Clona o copia una instancia de un objeto.

Estructurales:

- Adapter: Comunica interfaces de diferentes clases.
- Proxy: Un objeto representando a otro objeto.

De comportamiento:

- Observer: Una manera de notificar cambios a un número de clases
- Mediator: Define una comunicación simplificada entre clases
- Memento: Captura y restaura el estado interno de un objeto funcionalidad (ctrl+c, ctrl+z)

6. REFERENCIAS

Freeman, E., Freeman, E., Bates, B., & Sierra, K. (2004). Head First Design Patterns. O'Reilly.

Making, S. (2021). Source Making. Obtenido de Source Making: https://sourcemaking.com/design_patterns

Mintic. (2019). Mintic. Obtenido de https://www.mintic.gov.co/arquitecturati/630/articles-9262_recurso_pdf.pdf

Point, T. (2021). Tutorials Point. Obtenido de Tutorials Point: https://www.tutorialspoint.com/design_pattern/index.htm

TIC, A. C. (2017). Alta Consejería Distrital TIC. Obtenido de https://tic.bogota.gov.co/sites/default/files/archivos-adjuntos/arquitecturaTI.pdf

En el marco de los lineamientos del numeral 6.5 de la "Guía diseño de documentos del sistema integrado de gestión – SIG", se actualiza el código del presente documento para que se articule con la codificación vigente relacionada en la señalada guía y en el Listado Maestro de Documentos. El contenido del documento no cambia.