Universidad Don Bosco
Formando profesionales con calidad salesiana

FACULTAD DE INGENIERÍA

ESCUELA DE COMPUTACIÓN

Asignatura: Ingeniería de Software ISO104

Ciclo Académico: III-2021

Actividad:

Trabajo de Investigación #01

Catedrático:

Ing. Alexander Alberto Sigüenza Campos

Grupo Teórico:

04T

Presentado por:

Apellidos, Nombres	Carnet
Carrillo García, René Alexander	CG152751
Díaz Reyes, Jonathan Omar	DR160475
Hernández López, Manuel Oswaldo	HL180507
Olmedo López, Edwin Josué	OL150100

Soyapango, 18 de junio de 2021

Arquitectura Clean

Clean arquieture es un nombre popularizado por Roberth Cecile Martin conocido como "uncle bob" y se basa en la premisa de estructurar el codigo en capas contiguas.

Las capas que componen clean Architecture son:

ui: La interfaz de usuario propiamente dicha (html, xml)

- presenters: clases que se suscriben a los eventos generados por la interfaz
  - use cases: Evaluacion de reglas de negocios y toma de desiciones
    - Entities: Modelo de datos de la Aplicación, comunicacion con servidor web y cache de datos

Ventajas

Independecia: cada capa tiene su proplo modelo Arquitetonico.

Estructuración: mejor organizacion del codigo, facilitando la busqueda de funcionalidades

Desacoplamiento: Cada capa es independiente de las demes por lo que podria ser remplezada

Facilidad de testeo: podemo realizar test unitarios a cada

Desventajas

Metodologia: todo el equipo de desarrollo debe conocer la metodologia que se esta aplicando

Complejidad: La velocidad de desorrollo al comienzo es menor debido a que hay que establecer esta estructura

Principios Solid

- Single Responsibility Principle

Una clase debería ser responsable de una única cosa. En el momento en que adquiere más responsabilidad para el estar acoplada, algo que no es deseable si se quiere asegurar el mantenimiento de la aplicación. Esto se debe a que el cambio en una de sus responsabilidades puede afectar a la otra y viceversa.

Por supuesto este principio no aplica sólo a las clases, sino también a otros componentes de software así como a los famosos "microservicios"

Por ejemplo, supongamos esta clase:

```
class vehicle {
    constructor (identifier: string) { }
    getvehicleidentifier() { }
    saveVehicle (V: vehicle) { }
    getVehicle (identifier: string): vehicle { }
}
```

Esta clase viola el SRP. ¿Por qué? El principio de responsabilidad única establece que cada clase debería tener una única función. Sin embargo, la clase (vehicle) se encarga tanto de gestionar las propiedades (getVehicleIdentifier) como del almacenamiento

de nuestra aplicación es probable que esto afecte a la forma en que se gestionan las propiedades obligándonos a cambiar la clase "vehicle" así como las clases que la están usando.

Es decir, el SRP establece un alto grado de rigidez, de modo que no se produzca un efecto dominó cada vez que se produzca un cambio.

Una forma de solucionar el ejemplo anterior sería la siguiente:

```
class vehicle {
  Constructor (identifier: string) { }
  getVehicleIdentifier() { }
}

class vehicleDB {
    saveVehicle (v: vehicle) { }
    getVehicle (identifier: string): vehicle { }
}
```

De esta forma la clase "vehicle" solo tiene que validar una sola tarea y delega la tarea de almacenar las características de "vehicle"

* Open-Closed Principle.

Este principio establece que los componentes o clase debe tener en cuenta que debe estar abiertos para extender a partir de ellos, pero cerrados para evitar que se modifiquen.

Con esto protegemos la funcionalidad basica del sistema, es decir esto nos obliga a escribir nuevo codigo cuando queremos agregar nuevas funcionalidades es decir codigo que no se cambie conforme el proyecto avanza

Las principales herramientas para implemetar este principio son herencia y polimorfismo.

Ejemplo:

```
Interface IShape {
    function area(): number;
}

Class Rectangle Implements IShape {
    width: number;
    heigth: number;

    funcion area(){
      return this.width * this.heigth;
    }
}
```

```
Class Triangle implements IShape {
    Width: number;
    Heigth: number;
    función area() {
      return this.Width * this.heigth/2;
    }
}
```

Utilizando polimorfismo hemos creado la clase base IShape que tiene la función area esta solo pregunta que area tiene

```
Class Areacalculator {
  función computeArea(shapes: IShape[]) {
      let area=0;
      for(let shape OF shapes) {
        area+= Shape.area();
      }
      return are;
  }
}
```

Si se quisieran agregar nuevas formas solo faltaria crear nuevas clases que hereden la clase base.

*LisKov substitution principle.

Este principio lo que nos indica es que toda clase que es hija de otra clase debe poder utilizarse como si fuera el mismo padre, nadie que necesite utilizar el padre debe comportarse diferente si interactua con cualquiera de sus descendientes.

Ejemplo:

```
interface IFly {
  function flyL(): void;
}
interface ISwin{
   function swin(): void;
}
interface ICuack {
   function cuack(): void;
}
 class Duck implements Ifly, Iswin, ICuack{
    function flyL() {}
    function swin() {}
    function cuackl() {}
}
Class RubberDuck implements Iswin, ICuack {
   function swin() {
      console.log('le swin');
   }
   function cuack() {
      console.log('le cuack');
} }
```

pe esta manera diferenciamos lo que es un pato real a uno de goma que hacen cosas similares pero no por completo

```
Class DuckProcesser {

    function makeDucksFly(ducks: IFly[]) {
        for(let duck of DUCKS) {
            duck.fly();
        }
    }
}
```

* Interface segregation principle

Se describe como la priorización de la creación de multiples interfaces especificas en lugar de una unica interfaz en general esto facilita tener una arquitectura desacoplada esto conlleva a que las modificaciones y refactorizaciones sean implementadas de una manera sencilla, y que el software no tenga conflictos.

- Dependency Inversion Principle

La dependencia debería recaer sobre abstracciones, no sobre clases concretas

Dicho de otro modo:

- Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.

- Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

En el desarrollo de grandes aplicaciones llega un momento en el que contamos con multitud de módulos. Antes de que se produzcan un fuerte acoplamiento entre los distintos componentes y una gran dependencia de las librerías empleadas es recomendable aplicar este principio.

Por ejemplo, supongamos que tenemos un servicio encargado de recuperar elementos de la clase "vehicle" desde una API externa:

```
class vehicleFetcher {
    constructor() {
        $this->httpService = new HTTPService()
        $this->serializer = new Serializer();
    }
}
```

Esta clase incumple el principio de inversión de dependencia pues "vehicleFetcher" depende de clases de bajo nivel como la que se encarga de realizar llamadas HTTP o serializar objetos, las cuales están instanciadas en el constructor y utilizando posteriormente.

El primer paso para cumplir con este principio será parar a depender de abstracciones en vez de elementos concretos:

```
interface HttpService interface{
    function get();
    function post();
}

class HTTPService implements HttpService interface{
}
```

```
interface serializerInterface {
    function serialize();
    function unserialize();

class serializer implements serializerInterface {
}
```

De este modo estamos aislando una capa de abstracción a nuestra aplicación que nos permite depender tan solo de las interfaces en vez de las implementación concreta (podemos tener distintas librerías que serializan pero existe un contrato con la interfaz que todas deberán implementar).

A continuación, deberemos dejar de crear las clases concretas en el constructor de la clase "vehicle fetcher" de modo que estas sean pasadas desde el constructor:

```
class vehicle fetcher {
    construct (HttpServiceInterface $HttpService,
 serializerInterface $serializer ){
        $this -> http client = $HttpService;
        $this -> serializer = $serializer;
    }
}
```

De modo que si cambiamos la librería encargada de realizar llamadas HTTP no nos veamos obligados a modificar el código de la clase "VehicleFetcher" pues seguirá recibiendo una interfaz que cumple con el contrato establecido.

Patrones de Diseño.

- ¿Porque utilitar patrones de diseño?

Primero que todo es una buena practica de desarrollo de software, esta practica se usa para identificar condiciones de error y problemas en el codigo que pueden no ser evidentes en ese preciso momento.

"Los patrones de diseño te ayudan a estar seguro de la validez de tu codigo, ya que son soluciones que funcionan y han sido problemas por muchos desarrolladores"

Los patrones de diseño mas utilizados se clasifican en 3 categorias:

- Patrones creacionales.
- Patrones estructurados
- Patrones de comportamiento

Patrones Creacionales: Estos te ayudan a tener diversos mecanismos de creacion de objetos, que aumentan la flexibilidad y la reutilizacion del codigo existente de una manera adecuada a la situacion.

- Abstract Factory
- Builder Patterns
- Factory Method
- Prototype
- Singleton.

- **Patrones Estructurales**: Estos te facilitan soluciones estándares frecuentes a las composiciones de clase y las estructuras de objetos.

  - Adapter
  - Bridge
  - Composite
  - Facade
  - Decorador
  - Flyweight
  - Proxy.

- **Patrones de Comportamiento**: Este te ayuda a la comunicación entre objetos de clase. se utiliza para detectar patrones de comunicación ya presentes.

  - Chain of responsability
  - Command
  - Interpreted
  - Iterator
  - Mediador
  - State