

UNIVERSIDAD DON BOSCO.



Facultad de Ingeniería

Materia: Ingeniería de Software

Docente: Ing. Alexander Alberto Siguenza Campos

Tema: trabajo de investigación

Presentado por:

- Mario Josue Beltran Garcia
- Fher Enrique Climaco Escamilla
- German Alexander Meléndez Serrano
- José Samuel Mena Reyes



Hola soy atenea
diosa de la sabidura
y del arte de
la guerra.

Les enseñare
como combatir algunas
cosas que algunos
programadores no toman
en cuenta

"Arquitectura
CLEAN"

Pero, ¿que es la
arquitectura clean?

Para la guerra y para programar se
necesita una estrategia y una organización
impeable, es un conjunto de principios
cuya finalidad principal es ocultar los
detalles de implementación a la lógica
de dominio de la aplicación.

Y esto sirve para mantener aislada la lógica,
consiguiendo tener una lógica mucho más
mantenible y escalable.

Clean tiene divisiones, como una cebolla

- UI: La interfaz del usuario
- Presenters: Son las clases que reaccionan a eventos en el UI
- Use cases: La lógica del negocio
- Entities: Modelo para la comunicación con los datos

La guerra se basa en la superioridad, sobre el enemigo, en la programación existen más bien problemas y no enemigos. Relatemos las ventajas que tiene clean. A continuación le presentare mis armas

Independencia: cada capa tiene su propio paradigma o modelo arquitectónico como si se tratara de una aplicación en sí misma sin afectar al resto de niveles

Estructuración: mejor organización del código, facilitando la búsqueda de funcionalidades y navegación por el mismo.

Desacoplamiento: cada capa es independiente de las demás por lo que podríamos reemplazarla e incluso desarrollar en diferentes tecnologías.

Facilidad de testeo: podremos realizar test unitarios de cada una de las capas y test de integración de las diferentes capas entre sí, pudiendo reemplazarlas por objetos temporales que simulen su comportamiento de forma sencilla.

Ninguna defensa es perfecta (excepto la espantana) y veremos las desventajas de clean.

metodología: todo el equipo de desarrollo debe conocer la metodología que se está aplicando y cada desarrollador debe ser responsable de entender y aplicar las reglas establecidas a medida que está desarrollando.

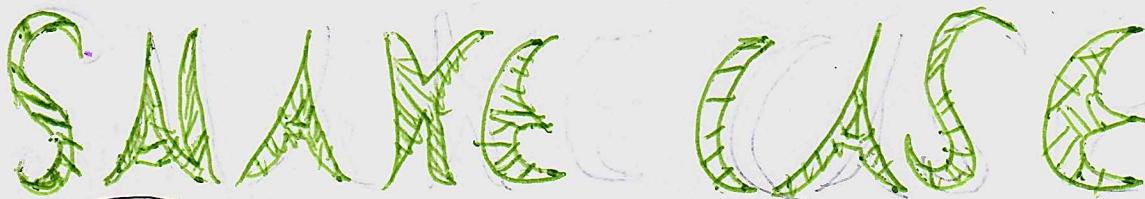
Complejidad: la velocidad de desarrollo al comienzo del proyecto es menor debido a que hay que establecer esta estructura y todo el equipo debe adaptarse a la nueva forma de trabajar, pero poco a poco y a medida que la aplicación va creciendo el mantenimiento y ampliación será más sencillo



ATENEA APPROVED

Debido a la cantidad de pros versus las contras la arquitectura Clean recibe un "atenea approved".

Otra herramienta que siempre se debe aplicar son las convenciones de código en pocas palabras es una forma estandarizada de como ordenar el código y como nombrar las variables



Es la convención que compone las palabras separadas por la barra baja (under score) ().
Basicamente la serpiente es el guion estirado) en lugar de espacios y con la primera letra de cada palabra en minuscula. Por ejemplo lista_de_sepiente

Este tipo de convención se utiliza en nombres de variables y funciones de lenguajes antiguo particularmente asociado con C. Aunque lenguajes como ruby y Python lo han adoptado. Existe una variante donde todas las letras estan en mayusculas y se llama SCREAMING-SNAKE-CASE



El nombre se debe a que las mayúsculas a lo largo de una palabra en Camel Case se asemejan a las jorobas de un camello.

Existen dos tipos de Camel Case

- **UpperCamelCase**, cuando la primera letra de cada palabra es mayúscula. ejemplo: **ListCamel**
- **lowerCamelCase**, igual que la anterior con la excepción de que la primera letra es minúscula. ejemplo: **listCamel**

Esta convención es muy usada, se usa en **#hashTag**, en nombres de empresas como La Liga y para variables en muchos lenguajes de programación PHP, Java, C#

<!-- Comentarios -->

Toda la guerra que se gana, pero no todo se ve algunas batallas se luchan en la sombra, y son iguales o mas importantes que las que todos ven, algunos consejos para ganar las batallas que nadie vera,

- **Complementar al código.** Un buen comentario debe transmitir la intención del programador, el concepto, el motivo, la estrategia, pero nunca a lenguaje natural ni explicar el código que se ha realizado, eso sera tarea del lector.

o Ser claro y comprensible. Escribimos los comentarios para que otros, o nosotros mismos en el futuro, los lean, así que asegúremos de que cuando lo hagan comprendan perfectamente el mensaje que les estemos transmitiendo.

¿Cuántas veces escribiste una nota que al leerla pasado cierto tiempo no eres capaz de entenderla?

o Ser conciso. No hace falta extendernos a explicar hasta el último detalle, cuanto más concisos seamos más fácil será comprenderlo e invertiremos menos tiempo en ello.



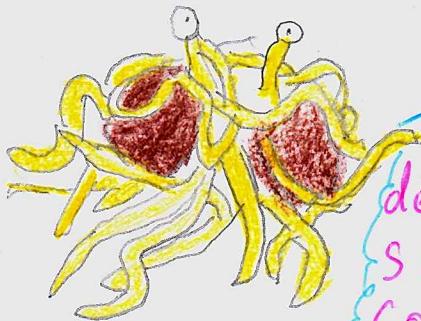
¡Ay, miren!

Miren al Kirby consejero
todo chiquito
todo consejero

C O D I G O
e S P a g u 3 T E i

El orden es fundamental para la victoria, si falta de ella es inaplicable por ello debes aplicar los siguientes consejos.

o Siempre debes estar atento a cualquier por si existe una posibilidad de código similar y jamás repetir código.



Gueridos seguidores, la pasta
debe estar en todo, pero por
subien que no este en su
Codigo.

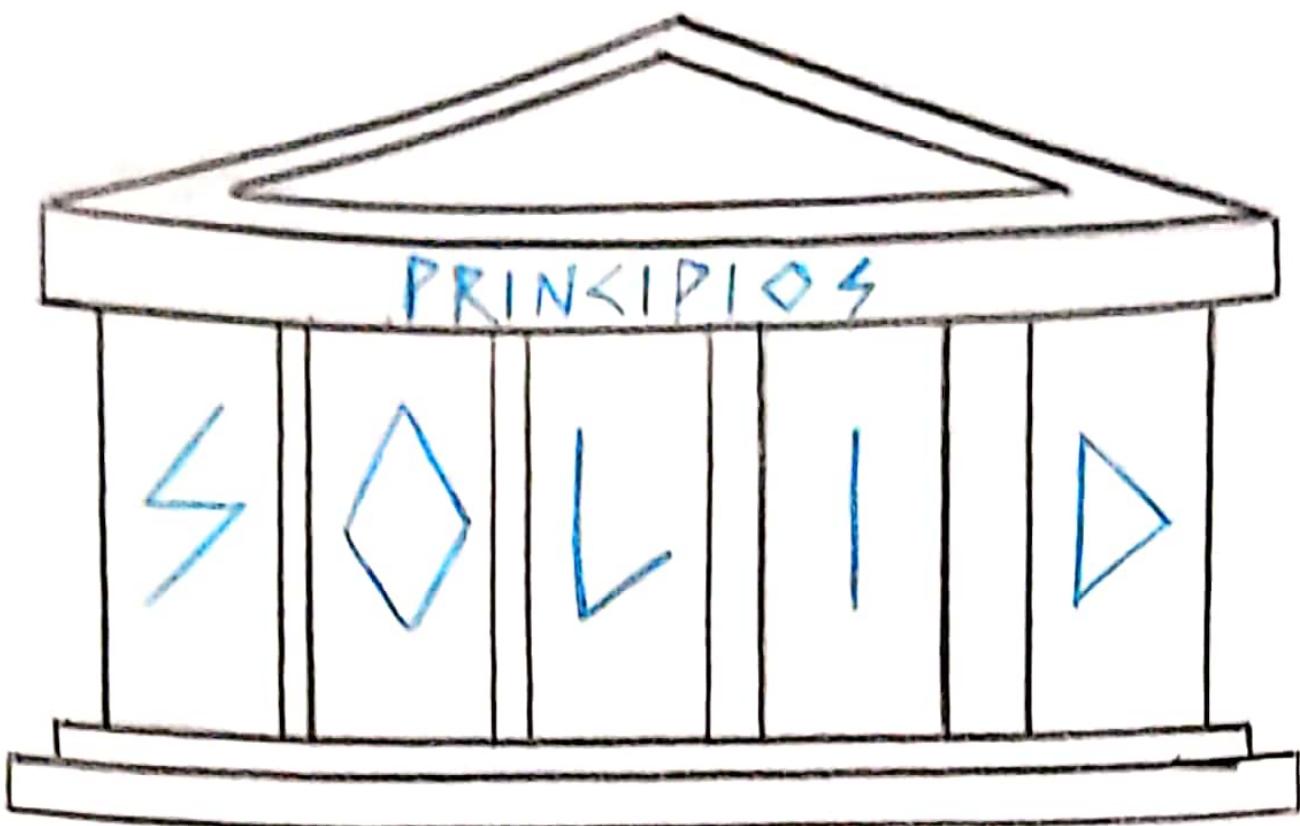
att. Mones vol

- Mantén una estructura lógica y organizada separa tu código
- Divide y vencerás resuelve primero el problema, crea el algoritmo en pequeñas secciones
- Los nombres de tus clases, funciones y variables deben de ser auto descriptivos
- Respeta los estándares de codificación
- Evitar usar más de un lenguaje de programación por fichero,
- Usar frameworks

Espero que todos
mis consejos les
sirvan

att. atenea





Estos principios ayudan a la gestión de dependencias dentro de la utilización del paradigma de programación orientado a objetos. Robert C. Martin creó esta propuesta para mejorar nuestro código tanto para nosotros como para otros programadores que vengan a leerlo. Estos son los siguientes principios:

- Responsabilidad Única
- Abierto/Cerrado
- Substitución de Liskov
- Segregación de Interfaces
- Inversión de Dependencias.



Principio de Responsabilidad Única



Por veces damos responsabilidades a una clase, sin que estas tenga esa responsabilidad de hacerlo.

Forma Incorrecta

```
class OrderService {  
    constructor(private _client:SmtpClient){}  
    add(order:Order){}  
  
    sendCustomerNotification(message:Message){  
    }  
}
```



Forma Correcta



```
class OrderService {  
    constructor(private _mailService:MailService){}  
    add(order:Order){}  
}  
  
class MailService {  
    constructor(private _smtpClient:SmtpClient){}  
    send(message:Message){}  
}
```

- Ahora tenemos otra clase con la responsabilidad de enviar notificaciones. Esto es grandioso, a la hora de estar cambiando el método de notificaciones únicamente tenemos que utilizar esa clase que creamos.

Principio de Abierto / Cerrado

- Consiste en piezas del software que deben estar abiertas para su extensión y cerradas para su modificación.



Forma Incorrecta

```
class NotificationService {
    send(notifications: Array<Notification>) {
        notifications.forEach(notification => {
            if (notification.type === "sms") {
                sendBySMS();
            }
        });
    }
}
```

Aún seguis programando así, que horror.



Forma Correcta

Código Abierto
a la modificación



Código Cerrado
a la modificación

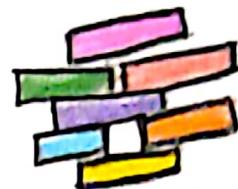
```
interface INotification {
    notify(): void;
}

class NotificationSMSService implements INotification {
    constructor(
        private to: String,
        private subject: String) {}

    notify(): void {}
}

class NotificationService {
    send(notifi: Array<INotification>) {
        notifi.forEach(noti => {
            noti.notify();
        });
    }
}
```

Principio de Sustitución de Liskov.



Las clases subtipos deben ser reemplazables por sus clases padres. Esto quiere decir, las clases hijas que heredan los métodos o atributos, no siempre son utilizadas como corresponde por las clases padres.

Forma incorrecta.

```
class Animal {  
    run(): void {}  
    walk(): void {}  
    hunt(): void {}  
}  
  
class Tiger extends Animal {}  
class Turtle extends Animal {}  
  
run() {  
    throw new Error('');  
}  
hunt() {  
    throw new Error('');  
}
```

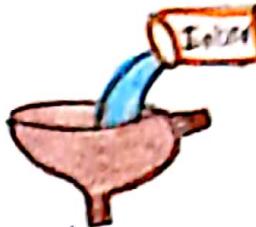
Forma Correcta

Hoy, si hay métodos que corresponden a las clases.



```
interface IHunt {  
    hunt(): void;  
}  
interface IRun {  
    run(): void;  
}  
interface IWALK {  
    walk(): void;  
}  
  
class Tiger implements IHunt, IRun, IWALK {  
    hunt(): void {}  
    run(): void {}  
    walk(): void {}  
}  
  
class Turtle implements IWALK {  
    walk(): void {}  
}
```

Principio de Segregación de Interfaces



Varias interfaces funcionan mejor que una sola.

Es decir que debemos separar los métodos en interfaces más accesibles y manejables para poder implementarlas en una clase.

Forma Incorrecta.

```
interface IRepository<T> {  
    update() {}  
    create() {}  
    get(): void;  
    getAll(): void;  
}  
  
class UserReportRepository implements  
IRepository<UserReport> {  
    get() {}  
    getAll() {}  
    create() { throw new Error(""); }  
}
```

Hay métodos que no se están utilizando dentro de la clase.
No quiero ver más el código.



¡Exquisito! La segregación de la interfaz está bien implementada,



Forma Correcta.

```
interface IReadable<T> {  
    get(): void;  
    getAll(): void;  
}  
  
interface IWriteable<T> {  
    update(): void;  
    create(): void;  
}  
  
class UserReportRepository implements  
IReadable<UserReport> {  
    get() {}  
    getAll() {}  
}  
}
```

Principio de Inversión de Dependencias.



Clases de alto nivel no deben depender de las clases de bajo nivel.

Forma Incorrecta

```
class MailChimpService {  
    send(){}  
}  
  
class OrderService {  
    constructor(private mailchimpService: MailChimpService){}  
    create(){}  
}
```

Este código no tiene escalabilidad.



Esta mejora de código nos ayudará a no depender de una clase de bajo nivel y esto nos facilitará a escalar el código en futuras versiones del software.

Forma Correcta

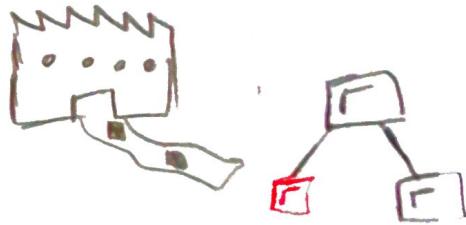
```
interface IMailService {  
    send(): void;  
}  
  
class MailChimpService implements  
IMailService {  
    send(){}  
}  
  
class OrderService {  
    constructor(private mailService:  
IMailService){}  
    create(){}  
}
```

Patrones Creacionales

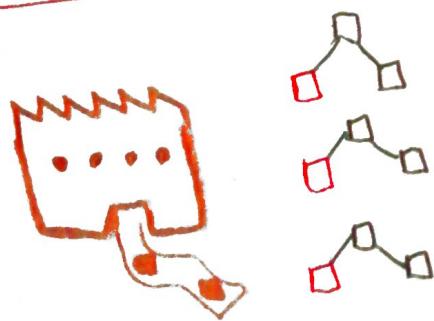
Los patrones creacionales proporcionan varios mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente.

Los patrones creacionales son los siguientes

Factory Method; este patrón proporciona una interfaz para crear objetos en una superclase, pero permite que las subclases alteren el tipo de objetos que se crearán.



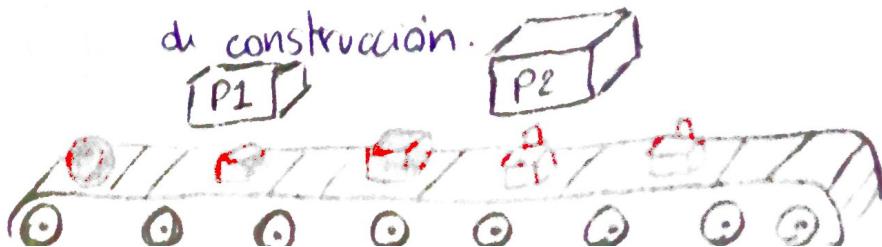
Abstract Factory



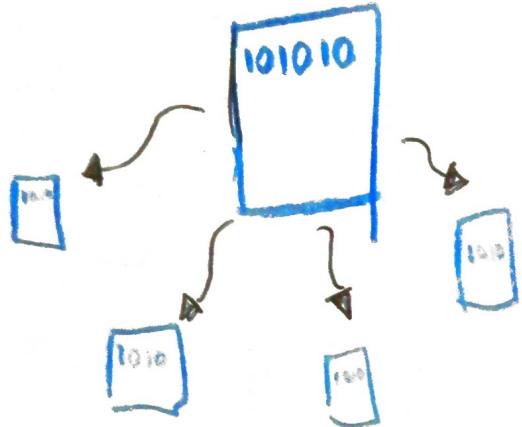
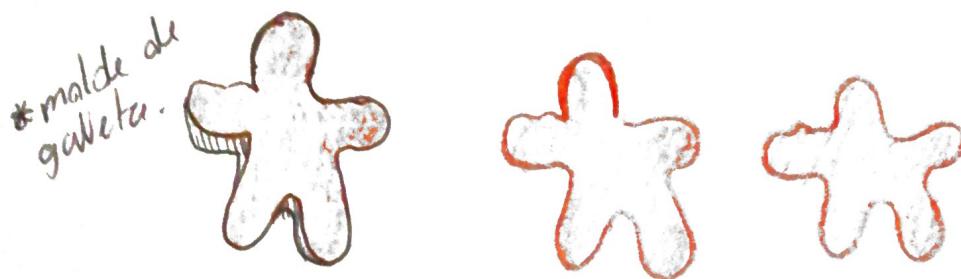
Este patrón nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

Builder

El builder nos permite construir objetos complejos paso a paso. El cual nos ayuda a producir distintas tipos y representaciones de un objeto empleando el mismo código de construcción.



Prototype, permite copiar objetos existentes sin que el código dependa de sus clases.



Singleton, permite asegurarnos de que una clase clase tenga una única instancia, que a su vez proporciona un punto de acceso global a dicha instancia.

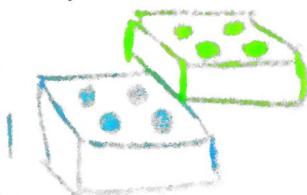
Patrones

Estructurales

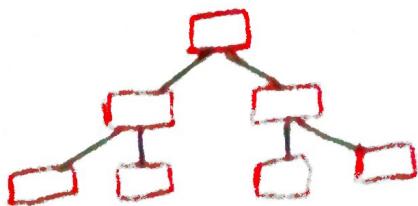
los patrones estructurales expresan cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras.

Los patrones estructurales son los siguientes

Adapter, este permite que trabajen juntas clases con interfaces incompatibles.



Composite, como su nombre indica, nos permite componer objetos en estructuras jerárquicas y trabajar cada nodo como si fueran objetos individuales.



Bridge, nos permite dividir una clase grande o un grupo de clases estrechamente relacionadas, convirtiéndolas en dos jerarquías separadas donde normalmente son abstracción e implementación.



Flyweight, nos permite usar más objetos dentro de la cantidad disponible en memoria RAM.



Decorator, permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especializadas que los contiene.



Facade, proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.



Proxy, permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original.



Patrones de Comportamiento

El patrón de comportamiento se ocupa de la comunicación entre objetos de la clase. Se usan para detectar la presencia de patrones de comunicación ya presentes.

- Chain of responsibility.

Evita acopljar el emisor de una petición a su receptor dando a más de un objeto la posibilidad de responder a una petición.

- Command.

Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud.

- Interpreter.

Se utiliza para evaluar el lenguaje o la expresión al crear una interfaz que indique el contexto para la interpretación.

- Iterator.

Su utilidad es proporcionar acceso secuencial a un número de elementos presentes dentro de un objeto de colección sin realizar intercambio de información relevante.

- Mediator.

Este patrón proporciona una comunicación fácil a través de su clase que permite la comunicación para varias clases.

- Memento

Permite recorrer elementos de una colección sin exponer su representación subjacente.

- Observer

Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda que está siendo observado.

- State

El comportamiento de una clase varía con su estado y está representado por el objeto de contexto.

- Strategy

Permite definir una familia de algoritmos, poner cada uno de ellos en una clase separada y hacer que sus objetos sean intercomunicables.

- Template method

Se usa con componentes que tienen similitud donde se puede implementar una plantilla del código para probar ambos componentes. El código se puede cambiar con pestañas modificaciones.

- Visitor

Definir una nueva operación sin introducir las modificaciones a una estructura de objeto existente.