

**FACULTAD DE INGENIERIA**  
**ESCUELA DE COMPUTACION**  
**Asignatura: Ingeniería de Software**  
**CICLO ACADEMICO: 03-2021**

Título:

Trabajo de investigación #1

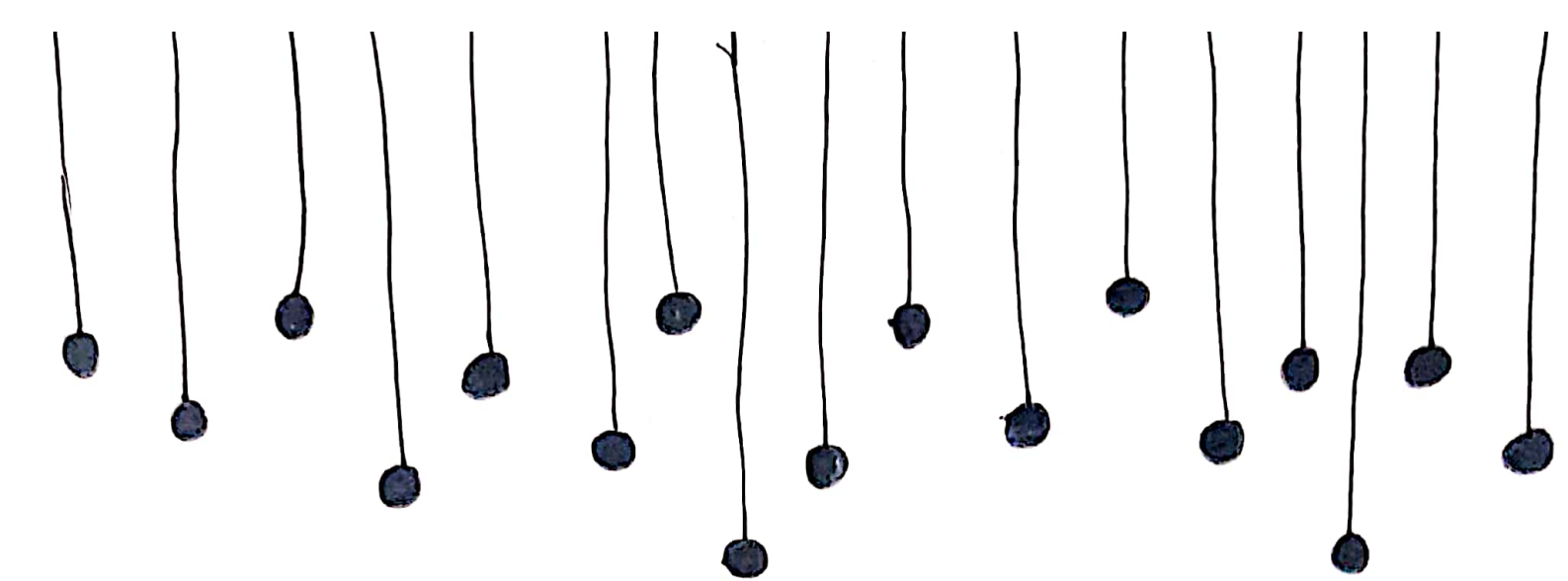
Docente:

Ing. Alexander Alberto Sigüenza Campos.

Presentado por:

<b>Apellidos, Nombres</b>	<b>Carné</b>
Gonzales Castellón, Carlos Armando	GC140244
Chávez García, Josué Alexander	CG172415
Velásquez Vega, Mauricio Ernesto	VV140557
Martínez Sanabria, David Ezequiel	MS180761

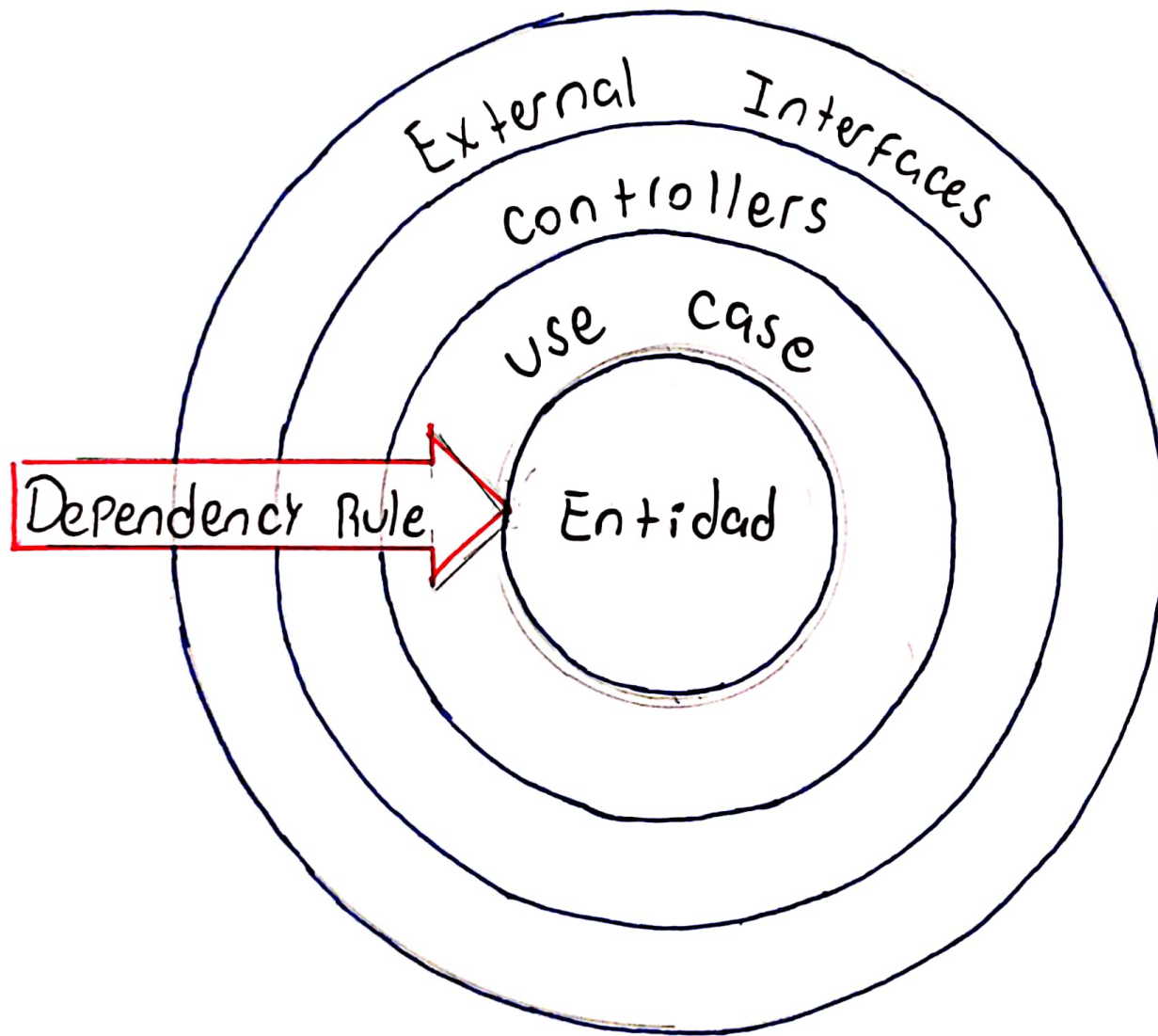
Soyapango, 17 de junio de 2021.



ARQUITECTURA

Clean

# Arquitectura Clean



## ¿Que es la Arquitectura CLEAN?



La arquitectura Clean o Clean architecture es un conjunto de principios cuya finalidad principal es ocultar los detalles de implementación a la lógica de dominio de la aplicación.

De esta manera mantenemos aislada la lógica, consiguiendo tener una lógica mucho más mantenible y escalable en el tiempo.

### La regla de dependencia

La principal característica de Clean architecture frente a otras arquitecturas es la regla de la dependencia.

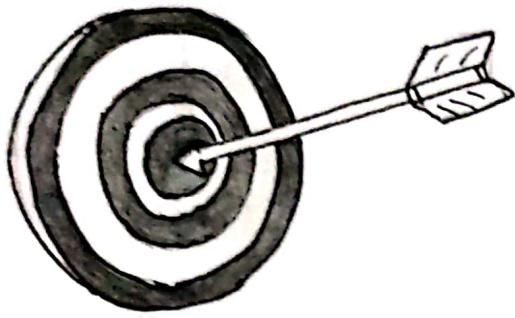
En Clean Architecture, una aplicación se divide en responsabilidades y cada una de estas responsabilidades se representa en forma de capa.

De esta forma tenemos capas exteriores y capas interiores:

- ▶ La capa más exterior representa los detalles de implementación
- ▶ Las capas más interiores representan el dominio incluyendo lógica de aplicaciones y lógica negocio empresarial.

La regla de dependencia nos dice que un círculo interior nunca debe conocer nada sobre un círculo exterior. Sin embargo los círculos exteriores si pueden conocer círculos interiores.

## ¿En qué capas se divide Clean Architecture?



La arquitectura Clean se divide en las siguientes capas:

- ▶ Entidades
- ▶ Casos de uso
- ▶ Adaptadores
- ▶ Frameworks y drivers

También podemos observar estas capas bajo la siguiente agrupación:

- ▶ Dominio → entidades y casos de uso
- ▶ Adaptadores
- ▶ Detalles de implementación → Frameworks y drivers

### ➡ Dominio

Es el corazón de una aplicación y tiene que estar totalmente aislado de cualquier dependencia ajena a la lógica o los datos de negocio.

### ➡ Entidades

Es aquella lógica que existirá aunque no tengamos una aplicación para automatizar los procesos de una compañía.

La lógica y datos de negocio empresarial se representa utilizando las entidades. Las entidades contienen los datos de negocio así como las reglas de negocio empresarial.



## ➡ Casos de uso

Representan la lógica de aplicación, que existe principalmente debido a la automatización de procesos mediante la aplicación y es inherente a cada aplicación.

## ➡ Adaptadores

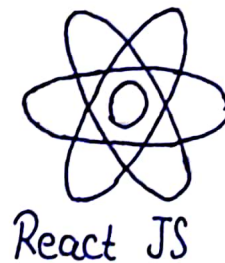
Se encargan de transformar la información como se entiende y es representada en los detalles de implementación o frameworks, drivers a como la entiende el dominio.

Es habitual utilizar en este punto junto con arquitectura Clean diferentes patrones de presentación como MVC, MVP, MVVM o BloC donde estos serían los adaptadores encargados de transformar la información de las vistas a información que necesitan los casos de uso.

## ➡ Detalles de implementación

Son todos aquellos frameworks, librerías que se suelen utilizar en una aplicación para mostrar o almacenar información:

- ~> Librerías de UI
- ~> Librerías de base de datos
- ~> Librerías de red
- ~> Librerías de analítica
- ~> Framework de Android
- ~> Framework de iOS
- ~> React JS
- ~> Vue JS
- ~> etc...



## Ventajas de usar Arquitectura Clean.

Las principales ventajas de usar Arquitectura Limpia son :

- ▶ Independencia de frameworks.
- ▶ Testeable
- ▶ Independiente de la interfaz de usuario
- ▶ Independiente de la base de datos
- ▶ El dominio es la parte más importante
- ▶ Jerarquía funcional





PRINCIPIOS

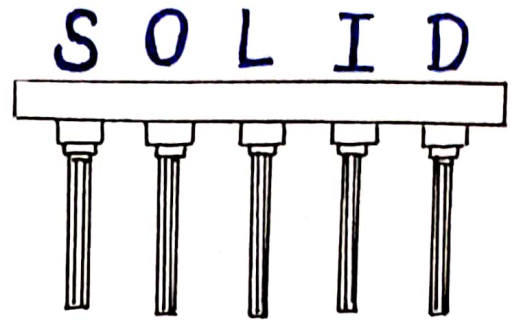
Solid





# ¿Que es el Principio SOLID?

SOLID es el acrónimo que acuñó Michael Feathers, basándose en los Principios de la programación orientada a objetos que Robert C. Martin había recopilado en el año 2000



## Los Principios SOLID

Los 5 Principios solid de diseño de aplicaciones de software:

S - Single Responsibility Principle (SRP)

O - Open/Closed Principle (OCP)

L - Liskov Substitution Principle (LSP)

I - Interface Segregation Principle (ISP)

D - Dependency Inversion Principle (DIP)

Entre los objetivos de tener en cuenta estos 5 principios a la hora de escribir código encontramos

<> Crear un software eficaz que cumpla con su cometido y que sea robusto y estable

<> Escribir un código limpio y flexible ante los cambios, que sea reutilizable y mantenible.

<> Permitir escalabilidad, que acepte ser ampliado con nuevas funcionalidades de manera ágil

## 1. Principio de Responsabilidad Única



Segun este Principio "una clase debería tener una y solo una razón para cambiar". Esto, precisamente "razon para cambiar", lo que se identifica como responsabilidad

El Principio de Responsabilidad Única es el más importante y fundamental de SOLID, muy sencillo de explicar, pero el más difícil de seguir en la práctica.

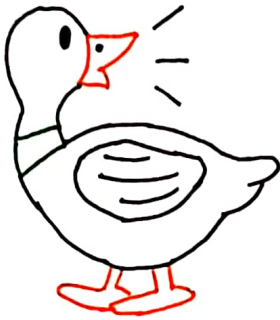
## 2. Principio de Abierto/Cerrado



El segundo Principio de solid lo formulo Bertrand Meyer en 1988 en su libro "Object Oriented Software Construction" y dice: "Deberías ser capaz de extender el comportamiento de una

clase, sin modificarla". En otras palabras: Las clases que usamos deberian estar abiertas para poder extenderse y cerradas para modificarse.

### 3. Principio de Sustitución de Liskov



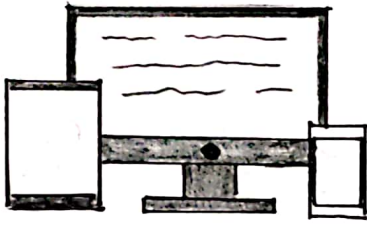
"Si Parece un Pato, hace Cuak como un Pato, Pero necesita baterías, Seguramente has hecho una mala abstracción"

La L de SOLID alude al apellido de quien lo creo, Barbara Liskov, y dice que las clases derivadas deben poder sustituirse por sus clases base.

Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.

Segun Robert C. Martin incumplir el Principio de Liskov (LSP) implica violar también el Principio de open/close.

## 4. Principio de Segregación de la interfaz



En el cuarto principio de SOLID, nos dice lo siguiente: "Haz interfaces que sean específicas para un tipo de cliente", es decir, para una finalidad concreta.

En este sentido, según el Interface Segregation Principle (ISP), es preferible contar con muchas interfaces que definen pocos métodos que tener una interfaz forzada e implementar muchos métodos a los que no se les darán uso alguno.

## 5. Principio de Inversión de Dependencias



El objetivo del Dependency Inversion Principle (DIP) consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases.

Robert C. Martin recomienda:

1. Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
2. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.






P a t r o n e s  
PATRONES

DE

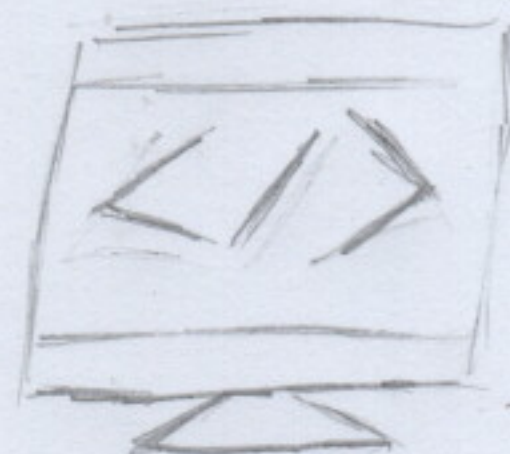
DISEÑO  
*diseño*  
DISEÑO





# P

## Patrones de diseño de Software



### ¿Qué son?

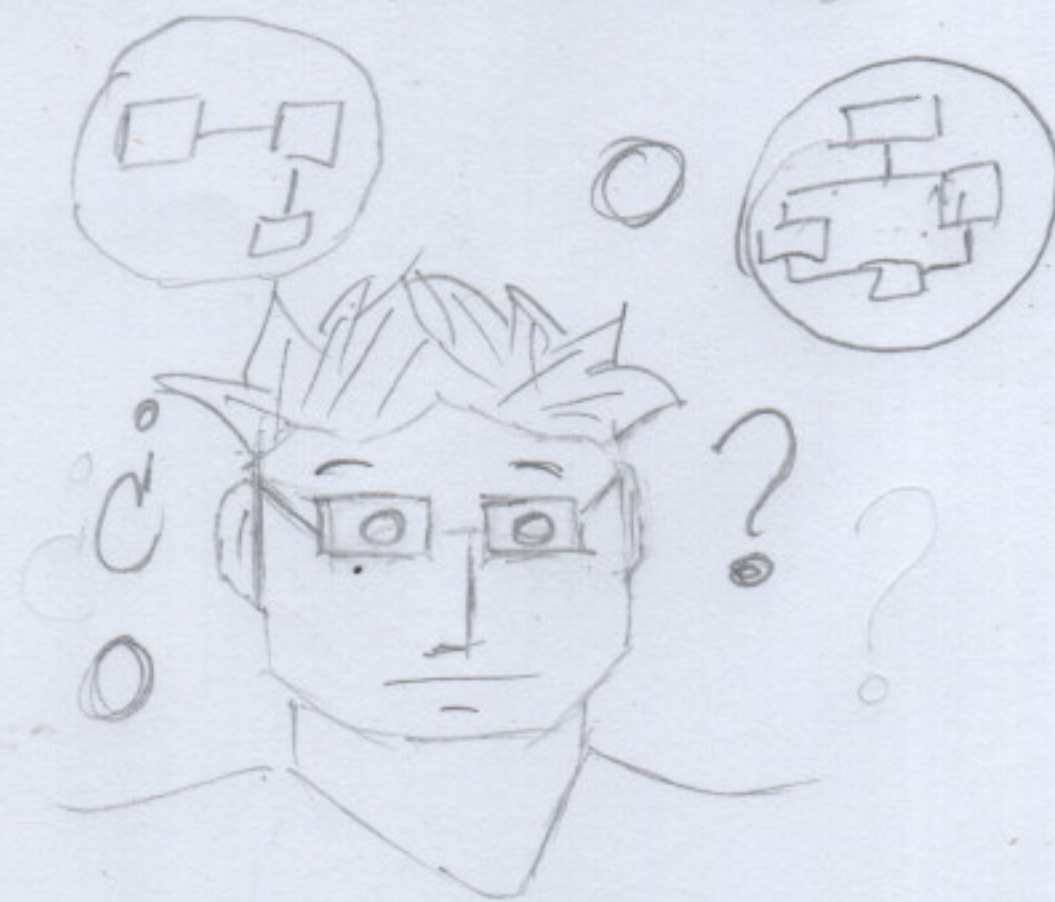
Es una solución general y reutilizable aplicable a diferentes problemas de diseño de software. Se trata de plantillas que identifican problemas en el sistema y proporcionan soluciones apropiadas a problemas generales.

### ¿Por qué usar Patrones de diseño?

Los patrones de diseño son soluciones que funcionan y han sido probados por muchísimos desarrolladores siendo menos propenso a errores.

### Tipos de patrones de diseño

✚ Patrones creacionales  
proporcionan diversos mecanismos de creación de objetos que aumentan la flexibilidad y la reutilización de código existente de una manera adecuada a la situación.



Estos son los patrones creacionales

▸ Abstract Factory.

Crea conjunto o familias de objetos relacionados sin especificar el nombre de la clase

▸ Builder Patterns

permite producir diferentes tipos y representaciones de un objeto utilizando el mismo código de construcción. Se utiliza para la creación etapa por etapa de un objeto complejo combinando objetos simples.



### ► Factory Method

Proporciona una interfaz para crear objetos en una superclase pero permite que las subclases alteren el tipo de objetos que se crearán.

### ► Prototype

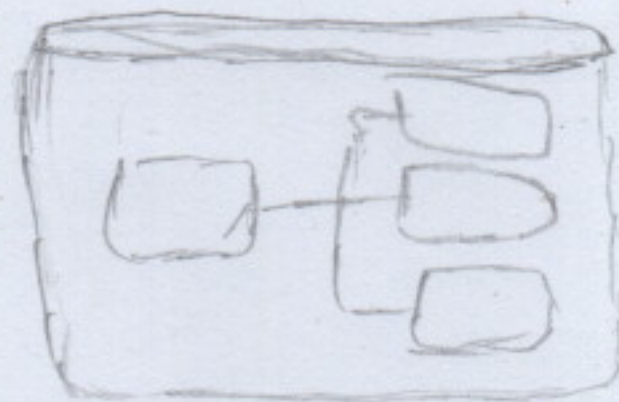
permite copiar objetos existentes sin hacer su código dependa de sus clases.

### ► Singleton

Este patrón restringe la creación de instancias de una clase a un único objeto.

## ⊕ Patrones estructurales

Facilitan soluciones y estándares eficientes con respecto a las composiciones de clase y las estructuras de objetos.



### ► Adapter

Se utiliza para vincular dos interfaces que no son compatibles y utilizan sus funcionalidades.

### ► Decorator

Este patrón restringe la alteración de la estructura del objeto mientras se le agrega una nueva funcionalidad.

### ► Proxy

Se utiliza para crear objetos que pueden representar funciones de otras clases u objetos y la interfaz se utiliza para acceder a estas funcionalidades.

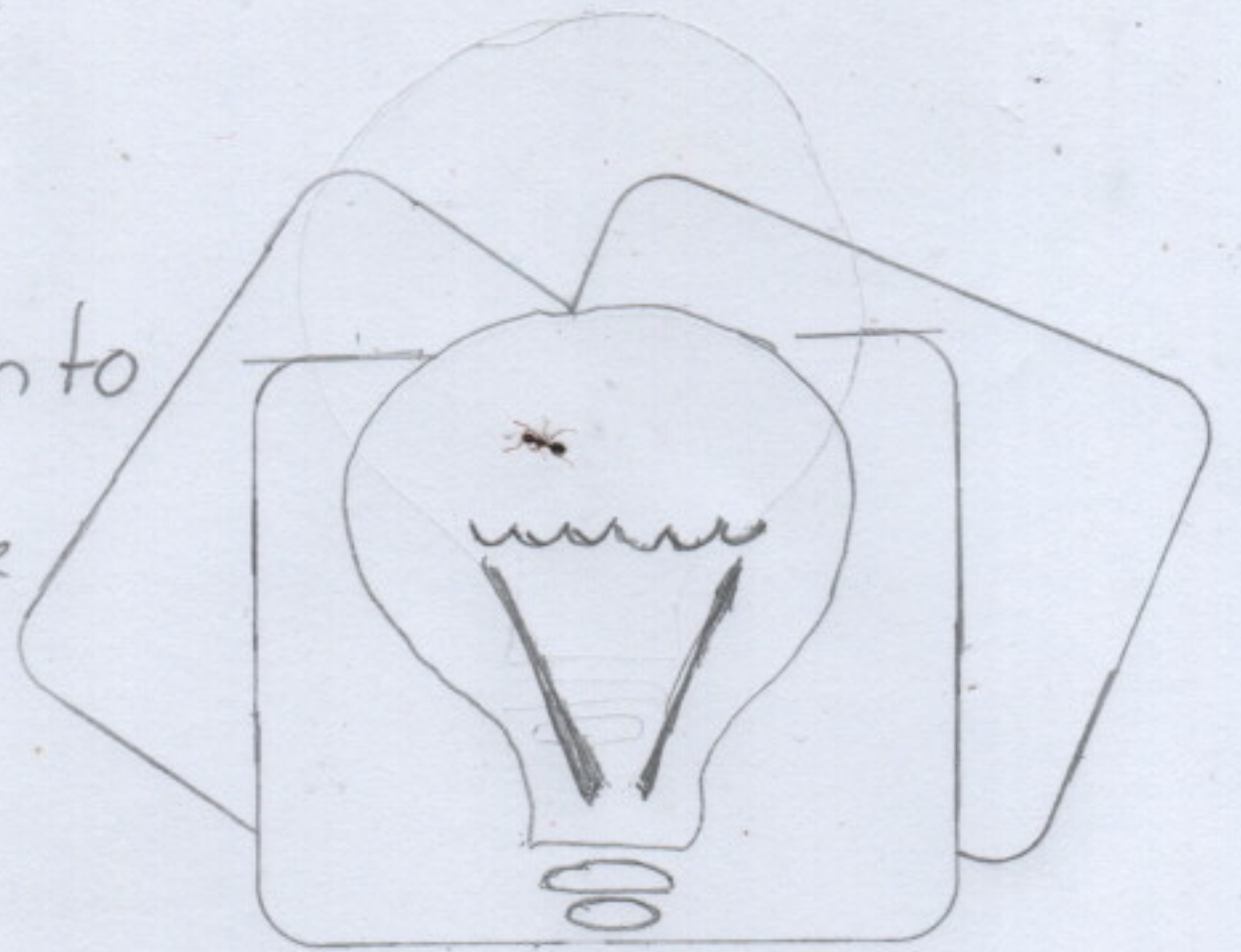
### ► Facade

Proporciona una interfaz simplificada para una biblioteca, un marco o cualquier otro conjunto complejo de clases.



## ⊕ Patrones de Comportamiento

Se ocupa de la comunicación entre objetos de clase.



### ▷ 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.

### ▷ Strategy

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

### ▷ Template method

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