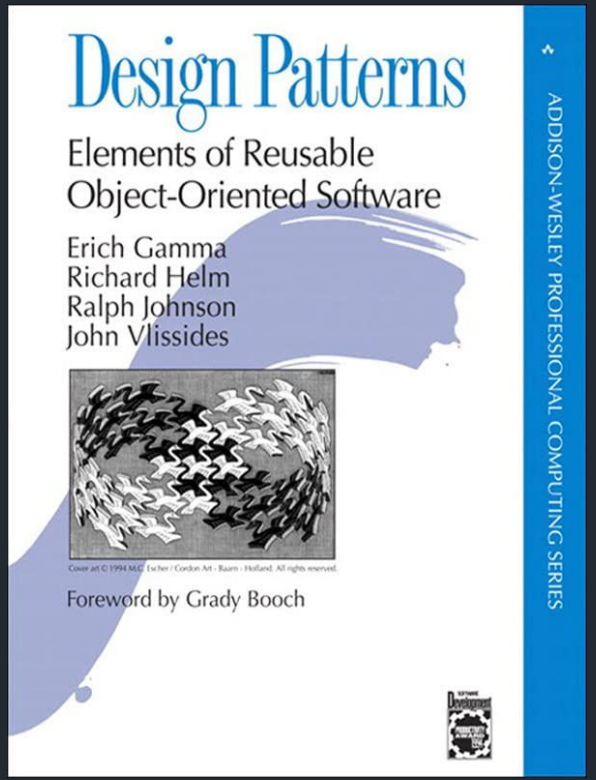
COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

Patrones de diseño

¿Qué son los patrones de diseño?

- Soluciones estandarizadas para problemas comunes en el diseño de software.
- Conceptos a alto nivel de posibles soluciones, varían dependiendo de cada problema.
- Compilados por el ***Gang of Four (GoF)*** en el año 1994 en el siguiente libro.
- Más información.



Ventajas de los patrones

- Conjunto de soluciones comprobadas y refinadas a lo largo de los años de experiencia real en el diseño de software.
- Usados correctamente ayudan a un mayor **mantenimiento** y **legibilidad** del código, **reduciendo** la **deuda técnica**.
- El impacto positivo del uso de patrones en proyectos a gran escala es enorme, ya que la arquitectura se produce antes de la implementación del código, ahorrando muchos problemas a futuro.
- Definen un marco de discusión común entre los desarrolladores en el que encontrar soluciones óptimas que todos puedan comprender y acordar.

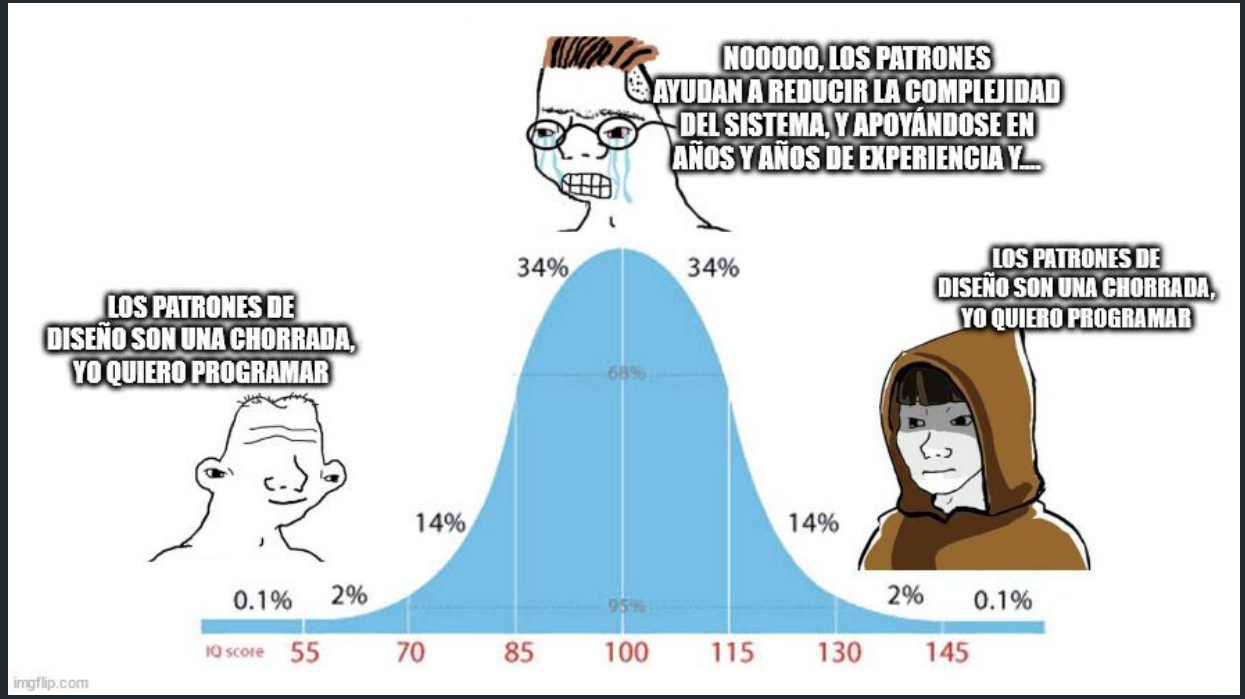
*(Ej: Acordar el uso de un patrón común como **Bridge** en tu arquitectura vs aprender la nueva lógica que se ha inventado un desarrollador de tu equipo).*

Los patrones como herramienta: críticas

- Los patrones son herramientas, no dogmas, por lo que su implementación depende de cada sistema, requisito, y motivaciones externas.
  - *Usar patrones de diseño en una aplicación empresarial o juego que requiera más de 3 años de mantenimiento y mejora* → *BIEN, necesario.*
  - *Usar patrones de diseño en una hack jam de 48h* → *No obligatorio.*
- El uso excesivo de patrones de diseño puede derivar en mucho boilerplate (código provisional) innecesario para funcionalidades simples, lo cual deriva en peor legibilidad y comprensión del sistema en su conjunto.
- *"Si eres un martillo, todo te parece un clavo…"* Aprender patrones de diseño puede derivar en intentar aplicarlos en cualquier parte, cuándo código más simple puede ser suficiente.
- *"Los patrones son soluciones antiguas a problemas que nuevos lenguajes incorporan"*

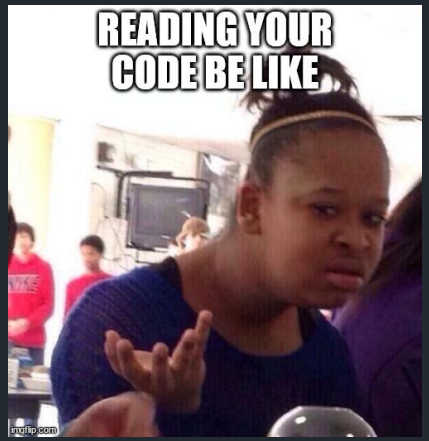
  → Cierto en algunos patrones como Iterator, pero la mayoría siguen solventando problemas de arquitectura del día a día.

Los patrones como herramienta: críticas



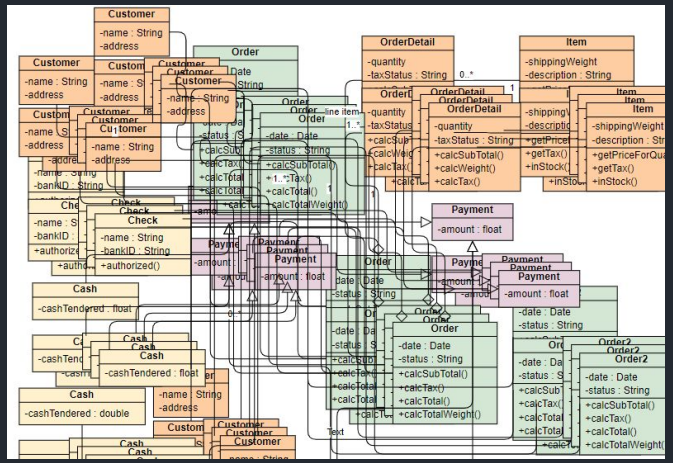
Anti-Patrones

- Un anti-patrón es cómo se conoce coloquialmente a una serie de problemas comunes encontrados en proyectos donde no se ha hecho uso debido de patrones, creando grandes problemas y dolores de cabeza si se sigue trabajando en él.
- Esto aumenta la deuda técnica del proyecto, haciendo cada vez más y más difícil extender la funcionalidad sin romperlo, o incorporar nuevos desarrolladores al mismo.



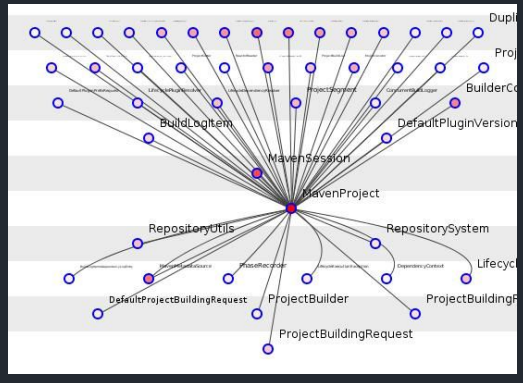
Anti-Patrones: Spaguetti code

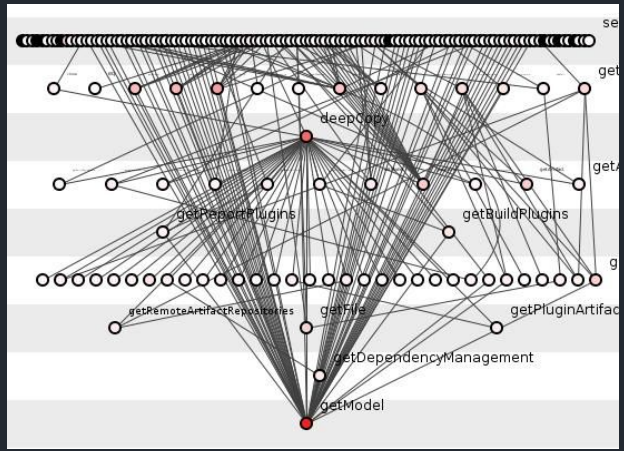
- Código sin estructura alguna, dependencias extrañas, referencias circulares, clases con miles de líneas y decenas de responsabilidades, código que ya no se usa…
- Prácticamente imposible de comprender por gente ajena al proyecto, muy difícil de añadir nueva funcionalidad… Arqueología de código.



Anti-Patrones: God Object

- God Object es un objeto al que se accede desde TODO para TODO.
- God object se produce cuando extendemos la funcionalidad de una clase para acabar teniendo mucha más responsabilidad de la que debería.
- Un God Object reduce la legibilidad del código y viola el principio de única responsabilidad (SRP).
- Desacoplar un God Object de las dependencias que tenga en nuevas clases puede ser muy muy complicado.





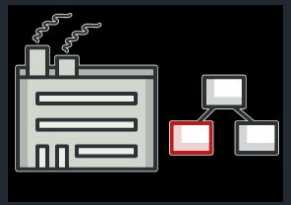
Anti-Patrones: Boat Anchor & Dead code

- Con Boat Anchor nos referimos a cuando partes del código innecesarias se quedan en él durante meses porque "*Algún día lo necesitaré*".
- Si es necesario extender la funcionalidad de la clase, quizá es necesario también heredar esas partes que nunca se han usado.
- Existe el control de versiones, no es necesario dejar código muerto en la aplicación.

Anti-Patrones: Golden hammer

- Golden Hammer hace referencia al uso de una misma lógica/arquitectura una y otra vez a lo largo del código sin pensar si realmente es la mejor solución.
- *"Si el martillo sirve para construir casas, también puedo usarlo para construir móviles y relojes!"*

Tipos de patrones

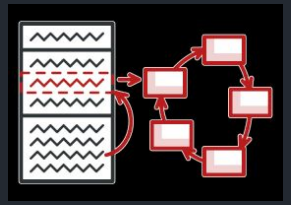


Patrones Creacionales

Proveen mecanismos de creación de objetos

Patrones Estructurales

Métodos de organización de grandes estructuras

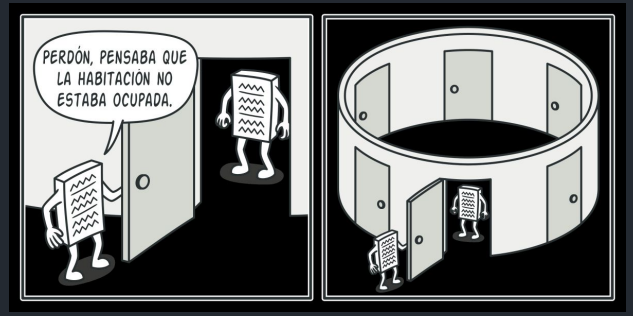


Patrones de Comportamiento

Se encargan de la correcta asignación de responsabilidades entre objetos

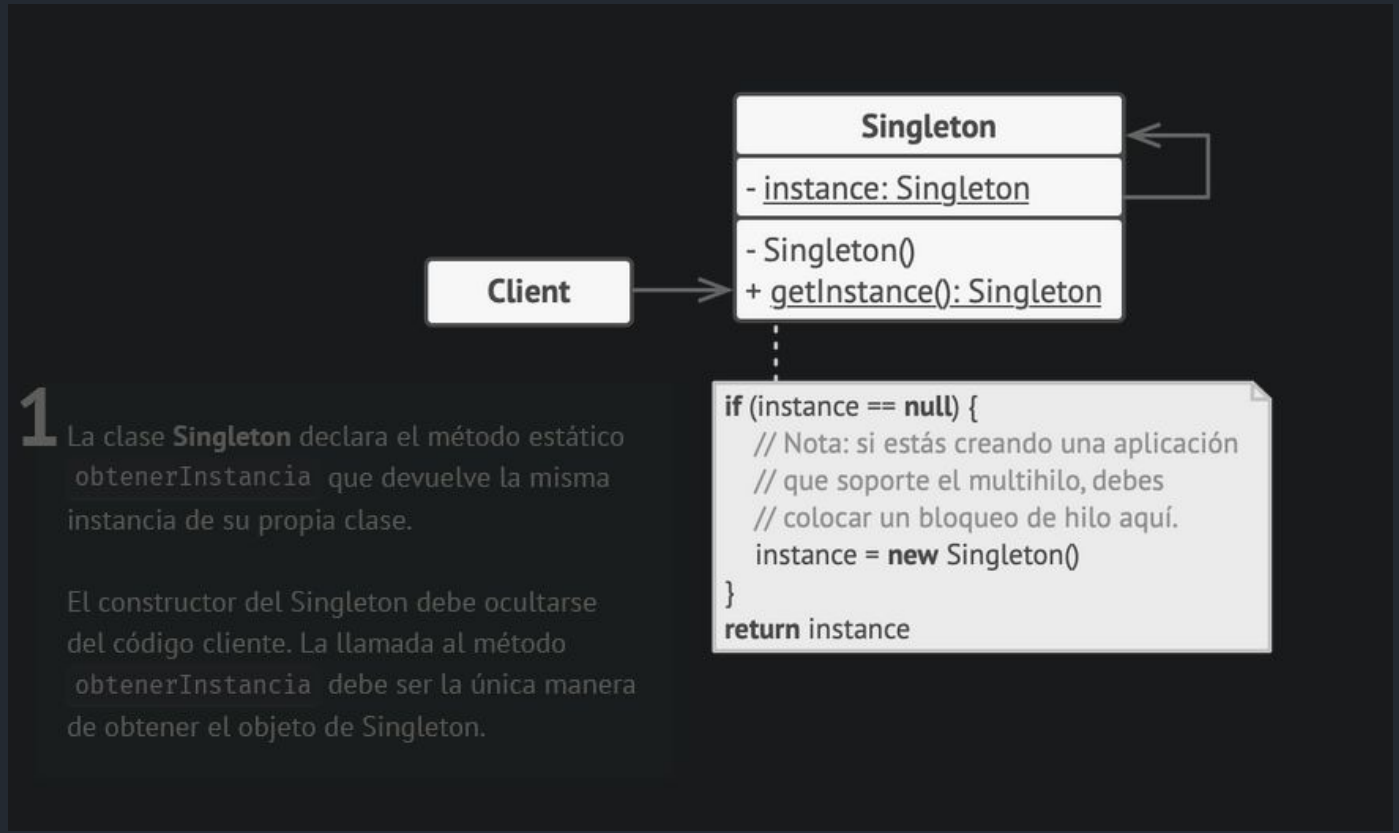
Patrón 1: Singleton (Creacional)

- **Problema**: Queremos acceder a una instancia única de una clase para almacenar o manipular data. (Ej: AudioManager, ScoreManager…)
- **Solución**: Crear una clase con el patrón Singleton:
  - Sólo puede existir una única instancia de dicha clase.
  - Proveer un acceso global a dicha instancia.
- Singleton es un patrón que genera nuevos problemas. No es recomendable en grandes sistemas.



(Por ejemplo, sólo puede existir un gobierno en España)

Singleton - Estructura



Singleton - Código

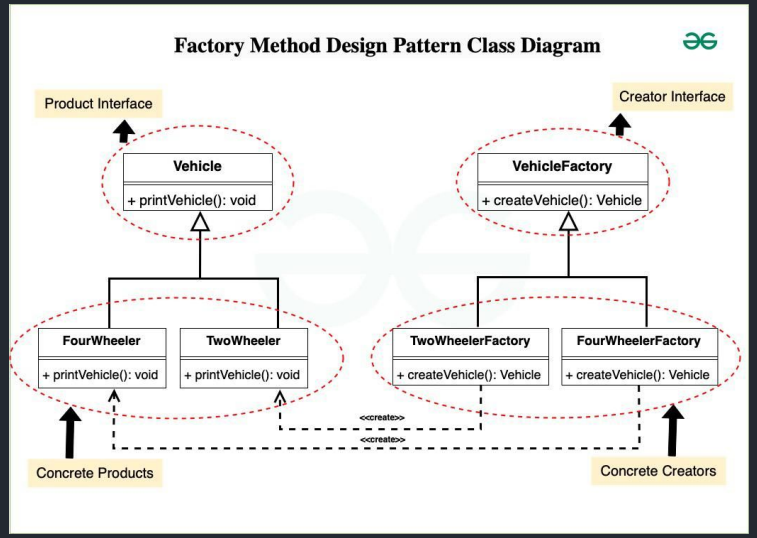
Singleton
- Constructor Privado
- Variable static typo <nombreClases> private
- GetInstance()/Instance() static

```
0 references
static void Main(string[] args)
{
    // ScoreSingleton references an unique instance of its class
    // that can be accessed from everywhere
    ScoreSingleton scoreSingleton1 = ScoreSingleton.GetInstance();
    ScoreSingleton scoreSingleton2 = ScoreSingleton.GetInstance();

    // Prints 1
    Console.WriteLine(scoreSingleton1.AddOneToScoreAndPrintValue());

    // Prints 2
    Console.WriteLine(scoreSingleton2.AddOneToScoreAndPrintValue());

    if (scoreSingleton1 == scoreSingleton2)
    {
        Console.WriteLine("Singleton works, " +
            "both variables contain the same instance.");
    }
    else
    {
        Console.WriteLine("Singleton failed, " +
            "variables contain different instances.");
    }
}
```

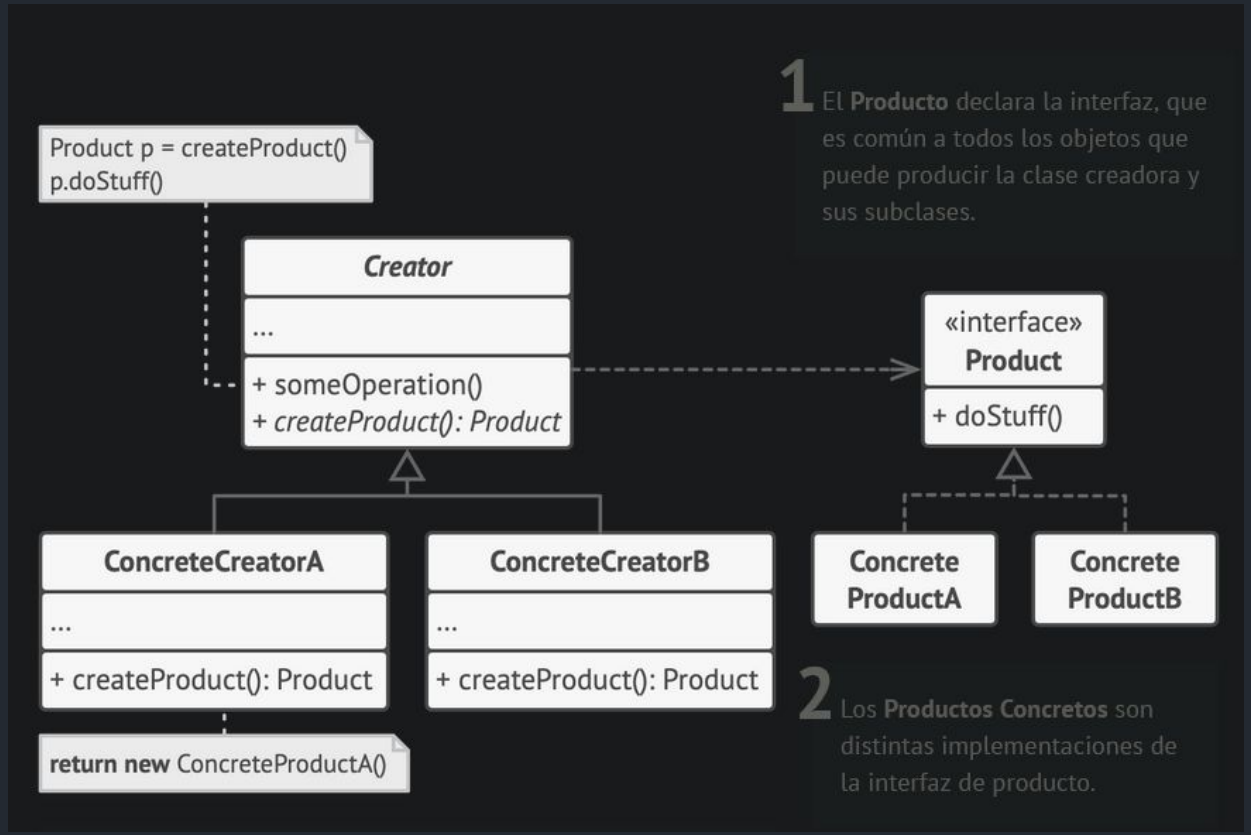
```
8 references
public class ScoreSingleton
{
    // private constructor so it cannot be built
    // somewhere else
    1 reference
    private ScoreSingleton() { }

    // Static reference since we force it to only
    // have one possible instance, itself
    private static ScoreSingleton _instance;

    public int score = 0;

    // Access to Singleton's instance, could be done
    // using properties
    2 references
    public static ScoreSingleton GetInstance()
    {
        if(_instance == null)
        {
            _instance= new ScoreSingleton();
        }
        return _instance;
    }

    2 references
    public int AddOneToScoreAndPrintValue()
    {
        score++;
        return score;
    }
}
```

Patrón 2: Factoría (C)

- Problema: La lógica de nuestros objetos requiere de nuevas subclases.
  El código que creaba y usaba dicha clase base crecerá en longitud y complejidad.
- Solución: Delegar la creación de objetos a una clase factoría base y factorías hijas que devuelvan el objeto requerido, desacoplando la creación de la lógica de uso.



Factoría - Estructura



Factoría - Código

```
0 references
static void Main(string[] args)
{
    TruckFactory truckFactory = new TruckFactory();
    BoatFactory boatFactory = new BoatFactory();

    Vehicle truck = truckFactory.BuildVehicle(weight: 2000);
    Vehicle boat = boatFactory.BuildVehicle(weight: 100000);

    truck.Move();
    boat.Move();
}
```

```
public abstract class VehicleFactory
{
    4 references
    public abstract Vehicle BuildVehicle(float weight);
}

2 references
public class BoatFactory : VehicleFactory
{
    2 references
    public override Vehicle BuildVehicle(float weight)
    {
        return new Boat(weight, speed: 1000);
    }
}

2 references
public class TruckFactory : VehicleFactory
{
    2 references
    public override Vehicle BuildVehicle(float weight)
    {
        return new Truck(weight, companyName: "Toyota");
    }
}
```

Factoría - Código

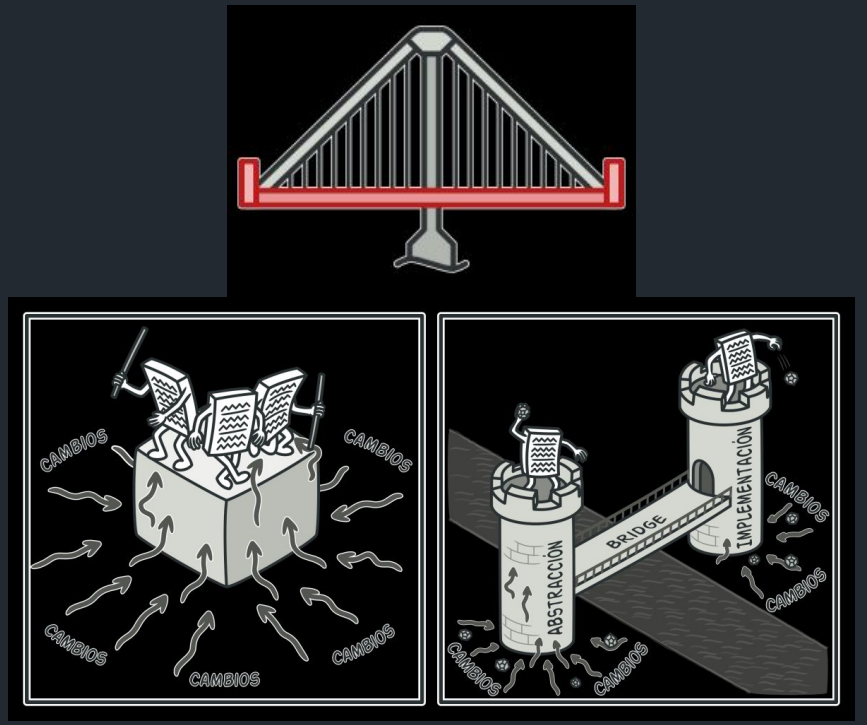
```
10 references
public abstract class Vehicle
{
    0 references
    public string Type { get; }

    1 reference
    public float Weight { get; }

    2 references
    protected Vehicle(float weight)
    {
        this.Weight = weight;
    }
    4 references
    public abstract void Move();
}
```

```
2 references
public class Boat : Vehicle
{
    1 reference
    public float Speed { get; }
    1 reference
    public Boat(float weight, float speed) : base(weight)
    {
        this.Speed = speed;
    }

    3 references
    public override void Move()
    {
        // Move boat's motors
    }
}
```

```
2 references
public class Truck : Vehicle
{
    1 reference
    public string CompanyName { get; }
    1 reference
    public Truck(float weight, string companyName) : base(weight)
    {
        this.CompanyName = companyName;
    }

    3 references
    public override void Move()
    {
        // Move truck's wheels
    }
}
```

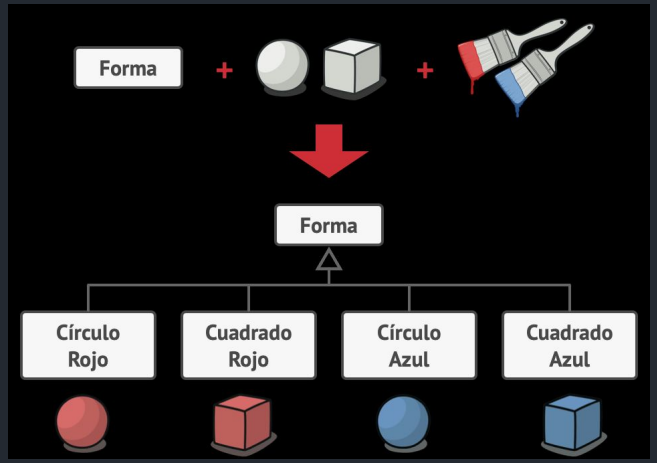
Patrón 3: Bridge (Structural)

- Bridge consiste en separar la **abstracción** de una clase de su **implementación**:
  - La abstracción es la capa de alto nivel que define la entidad y con la que el cliente interactúa. No contiene la lógica de trabajo.
  - La lógica de funcionamiento de la abstracción es ejecutada por la implementación.
  - La abstracción contiene una referencia a la implementación.



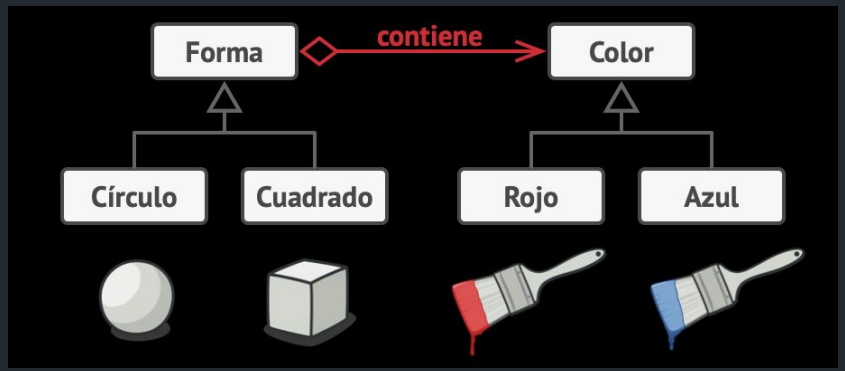
Patrón 3: Bridge

- Problema: Tenemos un sistema con figuras geométricas de diferente forma y color.
  - Este problema se presenta cuando queremos extender nuevas funcionalidades específicas dependiendo de diferentes parámetros.
  - Partiendo de una clase base **Forma**, si implementamos subclases dependiendo de cada forma y color, cada cambio a uno de estos parámetros requerirá de una nueva subclase.

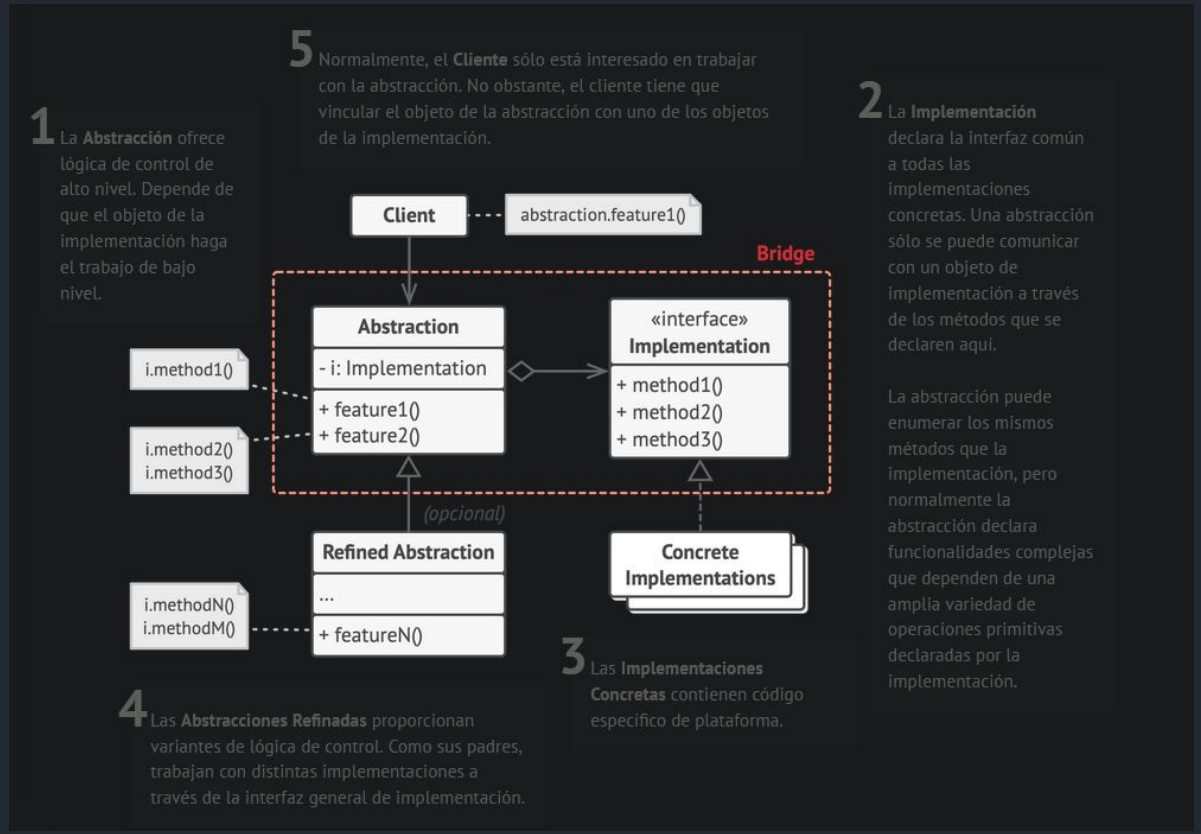


Patrón 3: Bridge

- Solución: Separar la abstracción Forma de sus implementaciones (Color).
  - El patrón Bridge separa la herencia de clases en la composición del objeto.
  - La clase Forma contiene una referencia a la implementación de Color, y delegará su lógica de funcionamiento a ella.
  - Añadir nuevos Colores al sistema no dependerá de crear nuevos subtipos de Forma.



Bridge - Estructura



Bridge - Código

```
0 references
static void Main(string[] args)
{
    // Geometric forms are the abstraction,
    // the ones who will be used by the client
    Square square = new Square();
    Sphere sphere = new Sphere();

    // Color and Texture are implementations, needed for the
    // abstraction to work and called by its methods
    Color red = new Color(255, 0, 0);
    Texture grassTexture = new Texture("Grass");

    // Add ref of implementations to the abstraction
    square.Color = red;
    square.Texture = grassTexture;

    sphere.Color = red;
    sphere.Texture = grassTexture;

    // Interact with the abstraction
    square.Draw();
    sphere.Draw();
}
```

Bridge - Código

```
3 references
public class GeometricForm
{
    // Using default constructor for code simplicity
    2 references
    public Color Color { get; set; }
    2 references
    public Texture Texture { get; set; }
    0 references
    public GeometricForm() { }
}

2 references
public class Square : GeometricForm
{
    1 reference
    public void Draw()
    {
        // Draw square using Color and Texture implementations
        //Color.ApplyColor();
        //Texture.ApplyTexture();
    }
}

2 references
public class Sphere : GeometricForm
{
    1 reference
    public void Draw()
    {
        // Draw sphere using Color and Texture implementations
        //Color.ApplyColor();
        //Texture.ApplyTexture();
    }
}
```

```
public class Color
{
    public int red, green, blue;

    1 reference
    public Color(int red, int green, int blue)
    {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

    0 references
    public void ApplyColor()
    {
        // Implementation's work
    }
}
```

```
4 references
public class Texture
{
    1 reference
    public string Name { get; set; }

    1 reference
    public Texture(string name)
    {
        this.Name = name;
    }

    0 references
    public void ApplyTexture()
    {
        // Implementation's work
    }
}
```