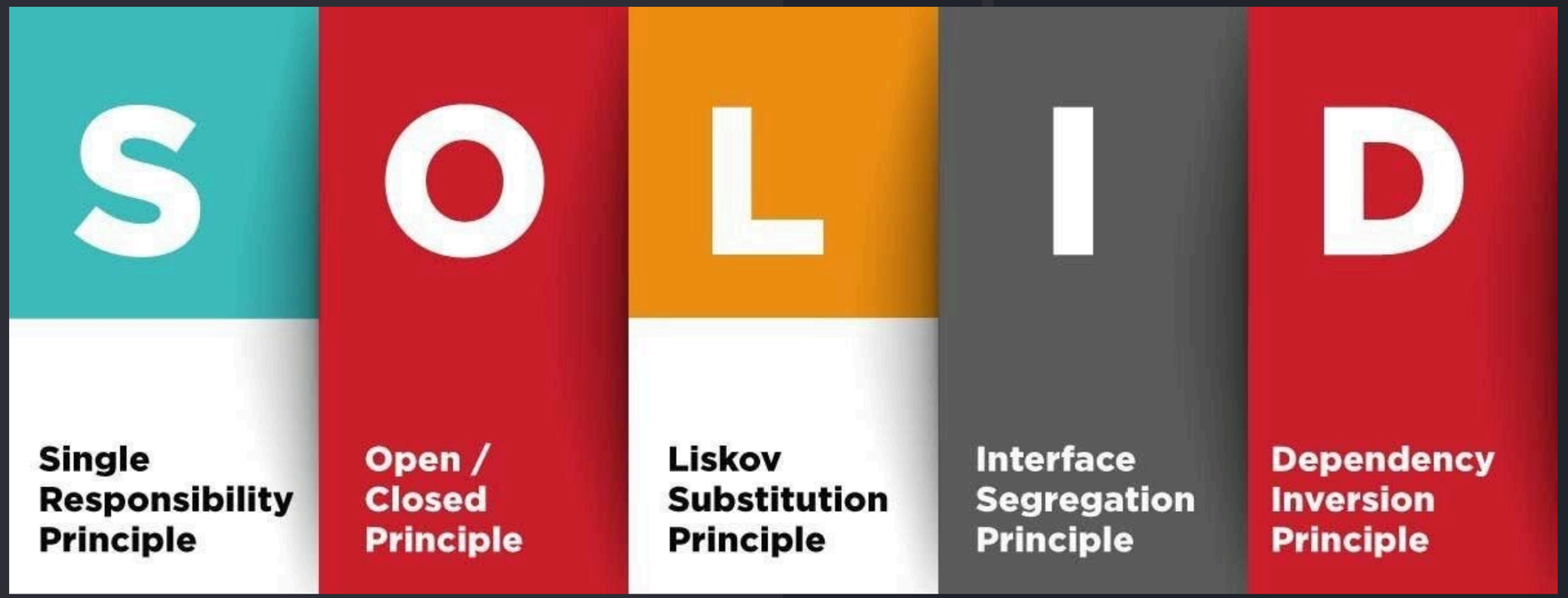
<!--SOLID-->

SOLID - Principio abierto/cerrado

<Por="Raquel Martinez"/>

}

SOLID {


S
O
L
I
D
Single Responsibility Principle
Open / Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

}

Definición{

Un artefacto de software debe estar abierto a ampliaciones pero cerrado a modificaciones.

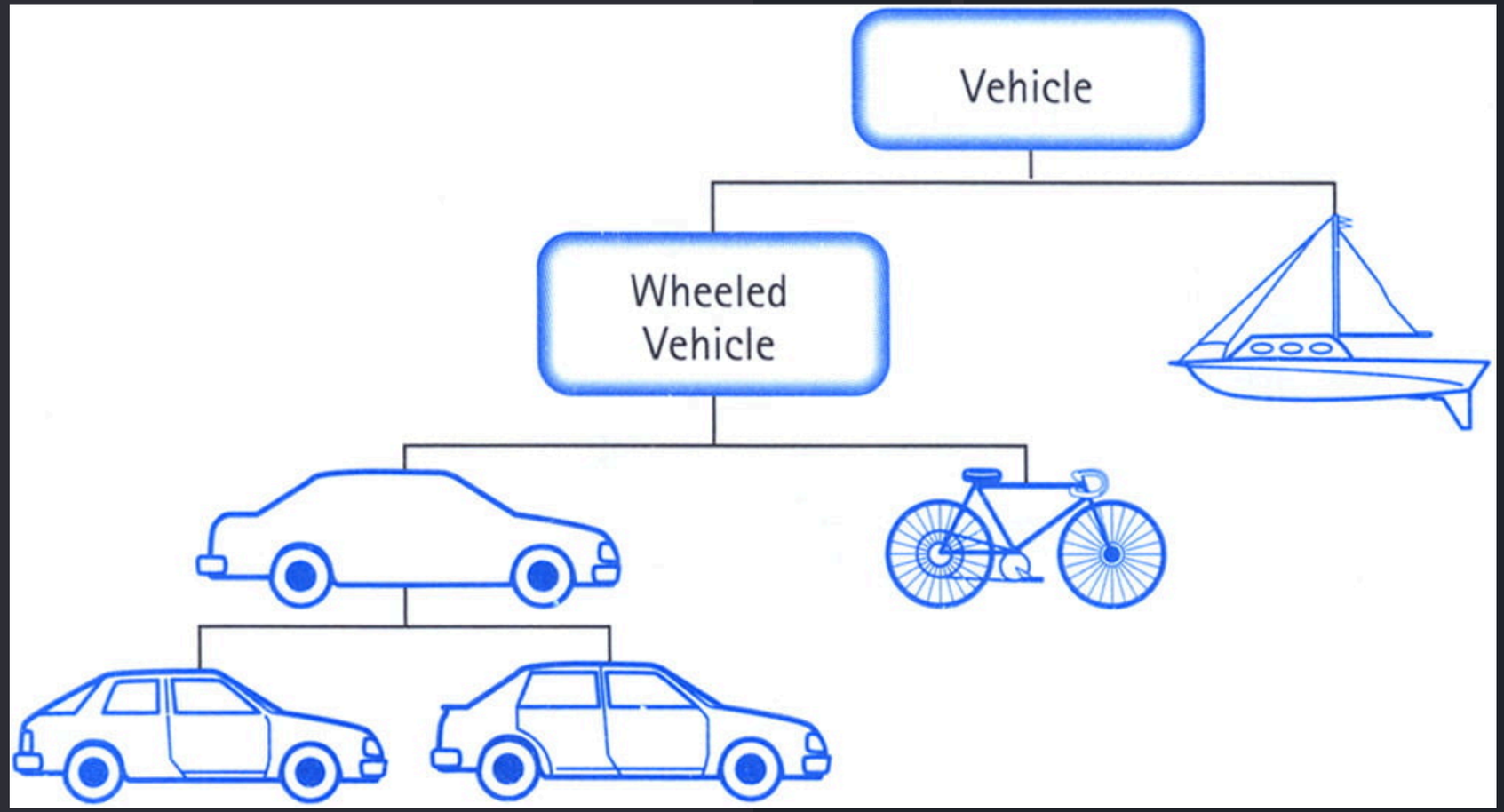
}

Beneficios {

- Facilita el mantenimiento y la evolución del código a largo plazo.
- Mejora la modularidad y la reutilización del código.
- Reduce el riesgo de introducir errores al modificar el código existente.
- Promueve un diseño más flexible y extensible.

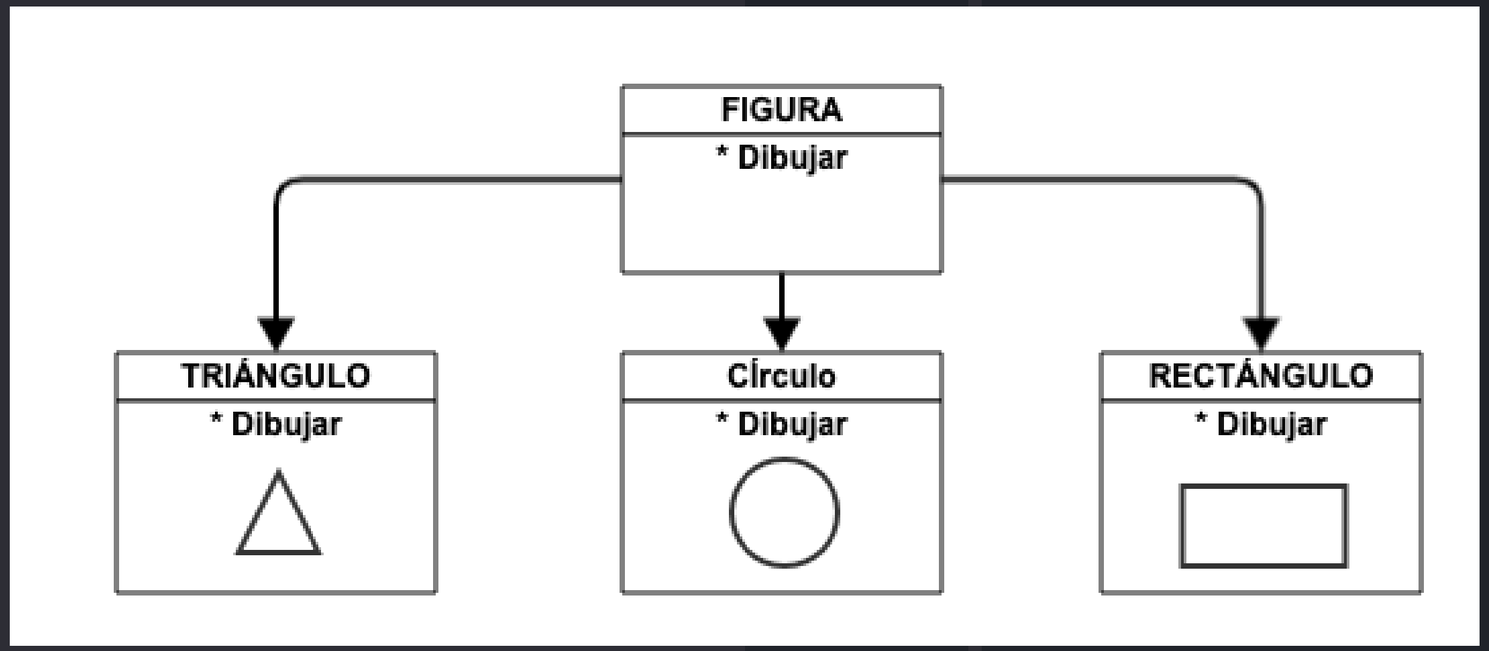
}

Implementación en Python - Herencia{



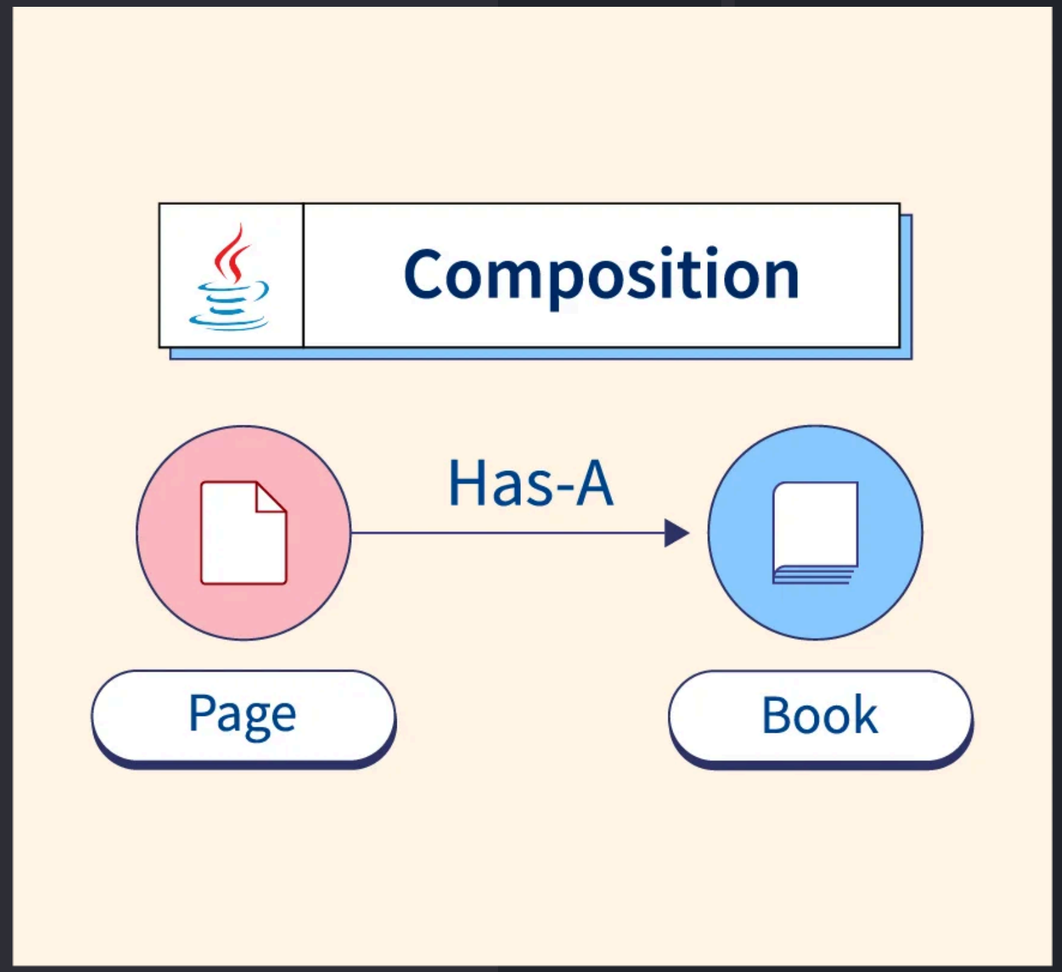
}

Implementación en Python - Polimorfismo {



}

Implementación en Python - Composición{



}

WO - OC {

```python
from math import pi

class Shape:
    def __init__(self, shape_type, **kwargs):
        self.shape_type = shape_type
        if self.shape_type == "rectangle":
            self.width = kwargs["width"]
            self.height = kwargs["height"]
        elif self.shape_type == "circle":
            self.radius = kwargs["radius"]

    def calculate_area(self):
        if self.shape_type == "rectangle":
            return self.width * self.height
        elif self.shape_type == "circle":
            return pi * self.radius**2i
```

}

W - OC {

```python
from abc import ABC, abstractmetho
from math import pi

class Shape(ABC):
    def __init__(self, shape_type):
        self.shape_type = shape_type

    @abstractmethod
    def calculate_area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        super().__init__("circle")
        self.radius = radius

    def calculate_area(self):
        return pi * self.radius**2

class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("rectangle")
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

class Square(Shape):
    def __init__(self, side):
        super().__init__("square")
        self.side = side

    def calculate_area(self):
        return self.side**2d
```

}

Implementación de diferentes algoritmos de clasificación {

```python
from abc import ABC, abstractmethod

class Classifier(ABC):
    @abstractmethod
    def train(self, data, labels):
        pass

    @abstractmethod
    def predict(self, data):
        pass
```

}

Implementación de diferentes algoritmos de clasificación {

```python
class NaiveBayesClassifier(Classifier):
    def train(self, data, labels):
        # Implementación del entrenamiento de Naive Bayes

    def predict(self, data):
        # Implementación de la predicción de Naive Bayes

class NeuralNetworkClassifier(Classifier):
    def train(self, data, labels):
        # Implementación del entrenamiento de Red Neuronal

    def predict(self, data):
        # Implementación de la predicción de Red Neuronal

class SVMClassifier(Classifier):
    def train(self, data, labels):
        # Implementación del entrenamiento de SVM

    def predict(self, data):
        # Implementación de la predicción de SVM
```

}

Implementación de diferentes algoritmos de enrutamiento {

```python
from abc import ABC, abstractmethod

class RoutingAlgorithm(ABC):
    @abstractmethod
    def calculate_routes(self, network_topology):
        pass

    @abstractmethod
    def update_routes(self, network_event):
        pass
```

}

Implementación de diferentes algoritmos de enrutamiento {

```python
class DistanceVectorRouting(RoutingAlgorithm):
    def calculate_routes(self, network_topology):
        # Implementación del cálculo de rutas de vector de distancia

    def update_routes(self, network_event):
        # Implementación de la actualización de rutas de vector de distanci

class LinkStateRouting(RoutingAlgorithm):
    def calculate_routes(self, network_topology):
        # Implementación del cálculo de rutas de estado de enlace

    def update_routes(self, network_event):
        # Implementación de la actualización de rutas de estado de enlace

class PolicyBasedRouting(RoutingAlgorithm):
    def calculate_routes(self, network_topology):
        # Implementación del cálculo de rutas basado en políticas

    def update_routes(self, network_event):
        # Implementación de la actualización de rutas basada en políticas
```

}

<!--SOLID-->

Codifiquemos! {

<Ejemplo/>

}

<!--SOLID-->

Gracias {

<Por="Raquel Martinez/>

}