Flying SOLID-ly:
Python design principles explained with birds

Egon Ferri

Core AIDE

Me: computer vision engineer in immobiliare.it

www.linkedin.com/in/egon-ferri/

@egonferri

LABS immobiliare

PyData ROMA CAPITALE

www.linkedin.com/company/pydata-roma-capitale

@pydataroma @pydataromacommunity

www.meetup.com/pydata-roma-capitale

LABS Immobiliare

What are the SOLID Principles?

What are the SOLID Principles?

***S**oftware **O**bject-oriented **L**ifelines for **I**t-Stays-Maintainable **D**esign*

Coined by Robert C. Martin (“Uncle Bob”), SOLID is a mnemonic for five guidelines that make classes easy to understand, reuse, test, and extend.

As codebases grow, tight coupling, leaky abstractions, and ever-changing requirements turn simple features into tangled bird-nests. SOLID gives us a checklist for untangling that mess.

A class should have only one reason to change – one responsibility.

Single responsibility

```python
# ❌ BAD: one bloated class does data, I/O *and* rendering
class Bird:
    def __init__(self, name):
        self.name = name

    def fly(self):
        print(f"{self.name} flaps its wings!")

    # DB responsibility
    def save_to_db(self, conn):
        conn.execute("INSERT INTO birds VALUES (?)", (self.name,))

    # UI responsibility
    def print_card(self):
        print(f"*** {self.name.upper()} ***")
```

LABS immobiliare

A class should have only one reason to change – one responsibility.

Single responsibility

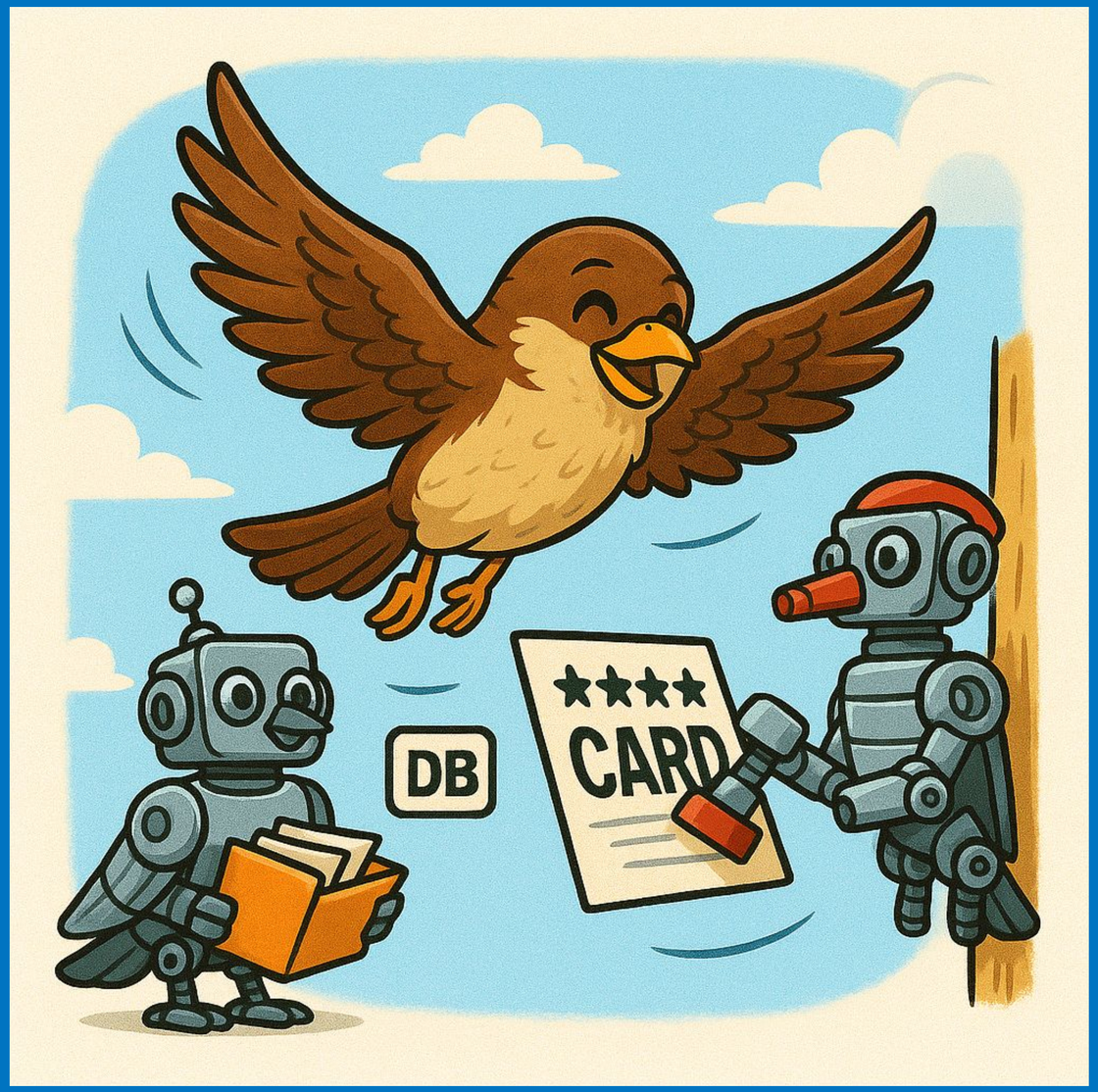
```python
# ✅ GOOD: separate classes, one job each
class Bird:
    def __init__(self, name):
        self.name = name

    def fly(self):
        print(f"{self.name} flaps its wings!")

class BirdRepository:
    def __init__(self, conn):
        self.conn = conn

    def save(self, bird: Bird):
        self.conn.execute("INSERT INTO birds VALUES (?)", (bird.name,))

class BirdCardPrinter:
    def render(self, bird: Bird):
        print(f"*** {bird.name.upper()} ***")
```



LABS Immobiliare

Software entities should be open for extension, closed for modification.

Open/closed

```python
# ❌ BAD: every new bird forces us to edit this function
def flight_speed(bird):
    if isinstance(bird, Sparrow):
        return 12
    elif isinstance(bird, Eagle):
        return 30
    # add elif for every new species…
```

LABS immobiliare

Software entities should be open for extension, closed for modification.

Open/closed

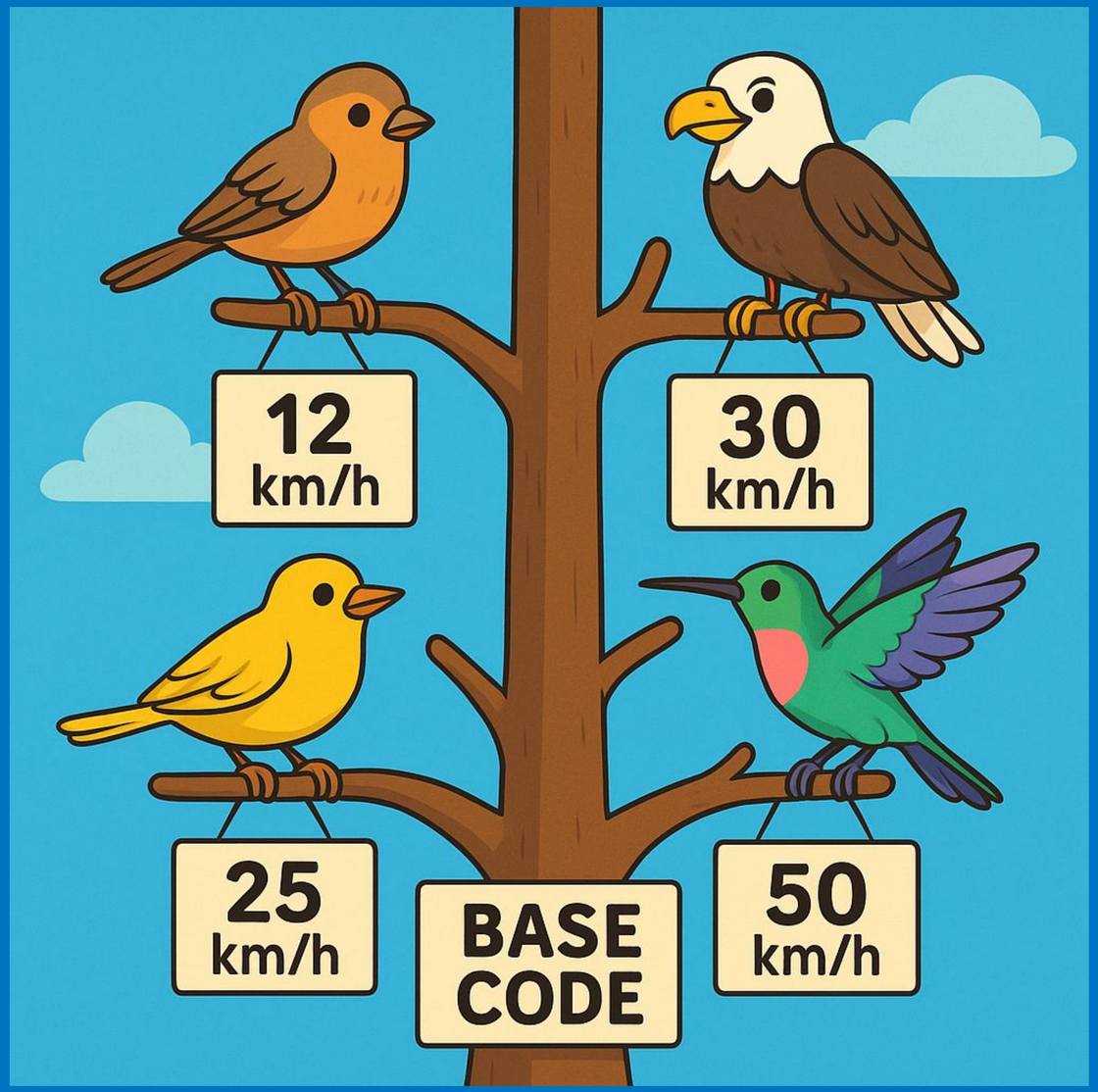
```python
from abc import ABC, abstractmethod

class Bird(ABC):
    @abstractmethod
    def flight_speed(self) -> int: ...

class Sparrow(Bird):
    def flight_speed(self): return 12

class Eagle(Bird):
    def flight_speed(self): return 30

def report_speed(bird: Bird):
    print(bird.flight_speed())
```



LABS
immobiliare

Subtypes must be completely substitutable for their base type without altering correctness.

Liskov Substitution

```python
# ❌ BAD: base class assumes all birds can fly
class Bird:
    def fly(self):
        print("Flying high!")

class Penguin(Bird):
    pass  # inherits fly() it cannot honour
```

Immobiliare LABS

Subtypes must be completely substitutable for their base type without altering correctness.

Liskov Substitution

```python
from abc import ABC, abstractmethod

class Bird(ABC):
    pass

class Flyable(ABC):
    @abstractmethod
    def fly(self): ...

class FlyingBird(Bird, Flyable):
    def fly(self):
        print("Flying high!")

class Penguin(Bird):
    def swim(self):
        print("Gliding through icy water!")
```

Clients shouldn’t be forced to depend on methods they don’t use; keep interfaces small and specific.

Interface Segregation

```python
# ❌ BAD: one giant interface
class Bird(ABC):
    @abstractmethod
    def fly(self): ...
    @abstractmethod
    def swim(self): ...
    @abstractmethod
    def sing(self): ...

class Kiwi(Bird):           # flightless!
    def fly(self): raise NotImplementedError
    def swim(self): raise NotImplementedError
    def sing(self): print("Peep!")
```

LABS Immobiliare

Clients shouldn’t be forced to depend on methods they don’t use; keep interfaces small and specific.

Interface Segregation

```python
class Flyable(ABC):
    @abstractmethod
    def fly(self): ...

class Swimmable(ABC):
    @abstractmethod
    def swim(self): ...

class Singable(ABC):
    @abstractmethod
    def sing(self): ...

class Kiwi(Singable):
    def sing(self): print("Peep!")

class Duck(Flyable, Swimmable, Singable):
    def fly(self): print("Whoosh!")
    def swim(self): print("Paddle!")
    def sing(self): print("Quack!")
```

LABS Immobiliare

High-level modules should depend on abstractions, not on concrete implementations.

Dependency Inversion

```
# ❌ BAD: Feeder locked to one food source
class WormDispenser:
    def dispense(self): return "worm"

class BirdFeeder:
    def __init__(self):
        self.source = WormDispenser()  # concrete dependency

    def feed(self, bird):
        bird.eat(self.source.dispense())
```

LABS Immobiliare

High-level modules should depend on abstractions, not on concrete implementations.

Dependency Inversion

```python
from abc import ABC, abstractmethod

class FoodProvider(ABC):
    @abstractmethod
    def dispense(self) -> str: ...

class WormDispenser(FoodProvider):
    def dispense(self): return "worm"

class SeedDispenser(FoodProvider):
    def dispense(self): return "seed"

class BirdFeeder:
    def __init__(self, provider: FoodProvider):
        self.provider = provider

    def feed(self, bird):
        bird.eat(self.provider.dispense())
```

LABS Immobiliare