Python Biella Group

S.O.L.I.D.

Python Biella Group

Obiettivi

Rendere più leggibile il nostro codice

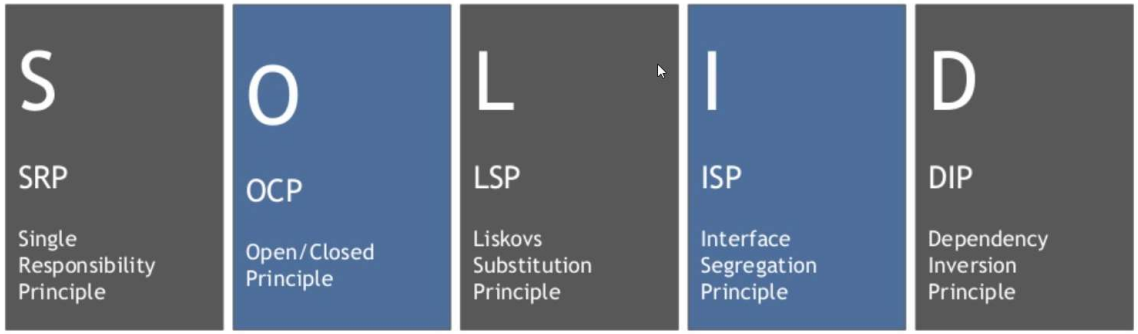
Migliorare conoscenza e utilizzo della programmazione a oggetti

Python Biella Group

S.O.L.I.D.

- Framework per progettare / fare refactoring con codice object-oriented “migliore”
- Serie di principi che, se rispettati, garantiscono
  - Manutenzione più facile / Più estendibilità
  - Comprensibilità / Codice più logico e facile da leggere
  - Stabilità / Codice più duraturo

- SOLID è un acronimo facilmente memorizzabile e sta per:
  - Single Responsibility Principle (SRP)
  - Open/Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)



- Vediamoli separatamente con esempi in Python

Python Biella Group

Single Responsibility Principle (SRP)

"A class should have one and only one reason to change"

- SRP richiede che una classe debba fare un solo lavoro.
- Quindi, se una classe ha più di una responsabilità, diventa accoppiata.
- Una modifica a una responsabilità comporta la modifica dell'altra responsabilità.

Open/Closed Principle (OCP)

*"Software entities should be **open** for extension, but **closed** for modification."*

- Classi, moduli, funzioni: dovrebbero essere aperte per l'estensione, chiuse per la modifica

Liskov Substitution Principle (LSP) / 1

- Se S è un sottotipo di T, gli oggetti di tipo T possono essere sostituiti con oggetti di tipo S
- Per qualsiasi classe, un client dovrebbe essere in grado di utilizzare indistinguibilmente uno qualsiasi dei suoi sottotipi, senza nemmeno accorgersene, e quindi senza compromettere il comportamento previsto in fase di esecuzione. Ciò significa che i client sono completamente isolati e inconsapevoli dei cambiamenti nella gerarchia delle classi.
- Una sottoclasse, figlio o specializzazione di un oggetto o di una classe deve essere «adatta» al suo genitore o superclasse.
- Si tratta di creare gerarchie corrette in modo che le classi derivate da una base siano polimorfiche rispetto all'interfaccia di quella genitore.
- Se tentiamo di estendere una classe con una nuova incompatibile, fallirà e il contratto con il cliente verrà rotto e di conseguenza tale estensione non sarà possibile.

"A subclass should behave in such a way that it will not cause problems when used instead of the superclass."

Python Biella Group

Liskov Substitution Principle (LSP) / 2

Break the Liskov Substitution Principle

Every subclass of Calculator needs to **implement a calculation function that returns a number**. Let's **break the principle** by creating a calculate function that can also raise an error. This example adds a DividerCalculator class (inherits from Calculator) where the overridden calculate function **raises an error when Python tries to divide by zero**.

```
class Calculator():
    def calculate(self, a, b): # returns a number
        return a * b

class DividerCalculator(Calculator):
    def calculate(self, a, b): # returns a number or raises an Error
        return a / b

calculation_results = [
    Calculator().calculate(3, 4),
    Calculator().calculate(5, 7),
    DividerCalculator().calculate(3, 4),
    DividerCalculator().calculate(5, 0) # 0 will cause an Error
]

print(calculation_results)
```

Non c'è modo di correggere questo codice senza:
- Refactoring della gerarchia di classi o
- Includere ogni chiamata di calcolo nel codice try / except

La classe **DividerCalculator** è diversa dalla classe **Calculator** in questo modo:
- La moltiplicazione di due numeri dà sempre come risultato un numero
- La divisione di due numeri genera un numero o un errore

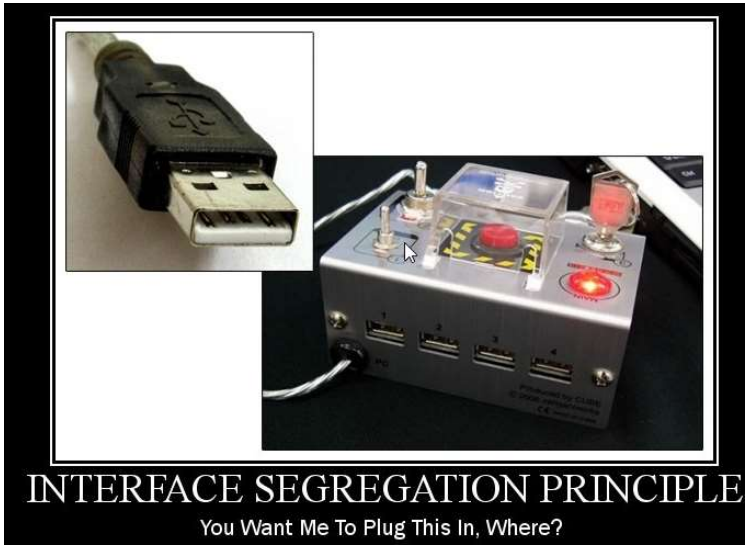
Ciò rende il tipo di risultato diverso e quindi l'interfaccia diversa. Moltiplica e Divide non sono la stessa cosa quando si tratta del tipo di risultato e uno non dovrebbe derivare dall'altro.

SOLUZIONE: aumentare o diminuire i livelli di astrazione nella gerarchia di classe

Python Biella Group

Interface Segregation Principle (ISP)

- Creare interfacce/astrazioni a "grana fine" che siano specifiche del client; evitare interfacce «grasse».
- I client non dovrebbero essere costretti a dipendere da interfacce che non utilizzano.
- Questo principio si occupa degli svantaggi dell'implementazione di grandi interfacce.




```python

class IShape:
    def draw(self):
        raise NotImplementedError

class Circle(IShape):
    def draw(self):
        pass

class Square(IShape):
    def draw(self):
        pass

class Rectangle(IShape):
    def draw(self):
        pass
```

Interface segregation.py hosted with ❤ by GitHub

Anche se in Python non ci sono le «interfacce», il principio si traduce nel trovare la «corretta astrazione»

Esempio classico

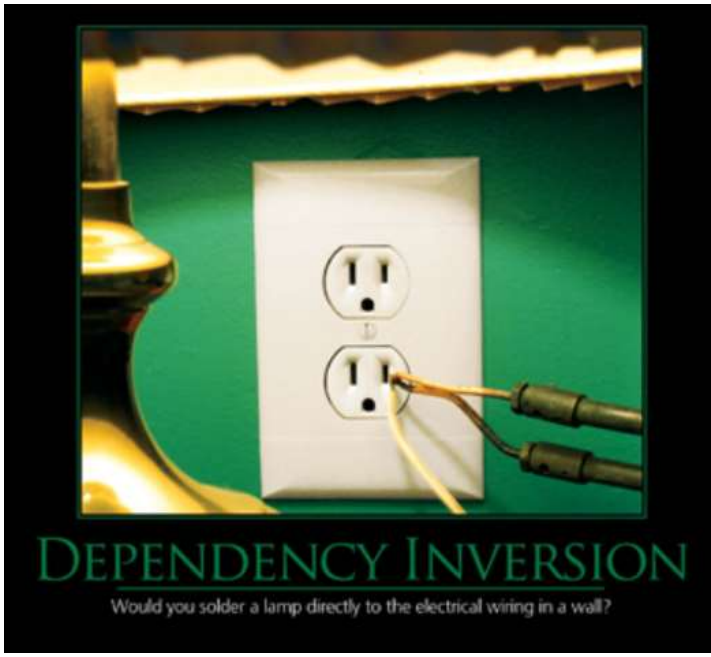
Ricordati che una singola classe può implementare diverse interfacce, se necessario.
Quindi possiamo fornire un'unica implementazione per tutti i metodi comuni tra le interfacce.

Le interfacce segregate ci costringeranno anche a pensare al nostro codice più dal punto di vista del cliente, il che a sua volta porterà a un accoppiamento lento e a facili test.
Quindi, non solo abbiamo migliorato il nostro codice per i nostri clienti, ma abbiamo anche reso più facile per noi stessi capire, testare e implementare.

“Clients should not be forced to depend upon interfaces that they don't use”

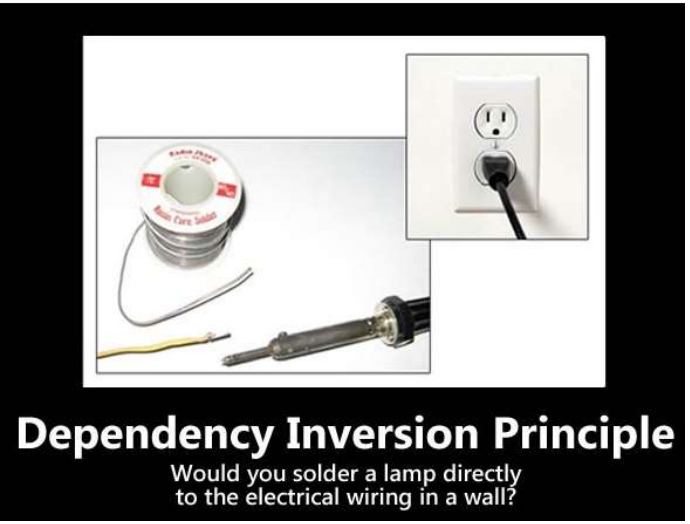
Python Biella Group

Dependency Inversion Principle (DIP)



- La dipendenza dovrebbe essere sulle astrazioni, non sugli oggetti concreti.
- I moduli di alto livello non dovrebbero dipendere dai moduli di basso livello.
- Sia le classi di basso che quelle di alto livello dovrebbero dipendere dalle stesse astrazioni.
- Le astrazioni non dovrebbero dipendere dai dettagli.
- I dettagli dovrebbero dipendere dalle astrazioni.

- Questo è ciò che lega tutto insieme. Tutto ciò che abbiamo fatto con gli altri principi SOLID è stato quello di arrivare a un punto in cui non siamo più dipendenti da un dettaglio



Si arriva un punto nello sviluppo del software in cui la nostra app sarà in gran parte composta da moduli.

Quando ciò accade, dobbiamo chiarire le cose usando l'inserimento delle dipendenze.

Per creare un comportamento specifico puoi utilizzare tecniche come l'ereditarietà o le interfacce/astrazioni.

Esempio: EventStreamer

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

"Abstractions should not depend upon details. Details should depend upon abstractions."

Python Biella Group

Riferimenti e approfondimenti

- SOLID
- https://github.com/heykarimoff/solid.python
- https://codingwithjohan.com/blog
- https://medium.com/@dorela/s-o-l-i-d-principles-explained-in-python-with-examples-3332520b90ff
- https://dev.to/ezzy1337/a-pythonic-guide-to-solid-design-principles-4c8i

- LSP
- https://hackernoon.com/liskov-substitution-principle-a982551d584a

- DIP
- https://stackoverflow.com/questions/26447502/explain-this-motivational-poster-about-dependency-inversion-principle

...and KISS!

Keep It Simply Stupid

- Sito: https://pythonbiella.herokuapp.com/
- GitHub: https://github.com/PythonGroupBiella/MaterialeLezioni
- YouTube: https://www.youtube.com/channel/UCkvQcNjmC_duLhvDxeUPJAg
- Telegram: https://t.me/joinchat/AAAAAFGSWcxhSln_SRhseQ

JOIN US!