SOLID-принципы

Принцип единственной ответственности (The Single Responsibility Principle): у каждого объекта должна быть только одна ответственность. Все поведение этого объекта должно быть направлено на обеспечение этой ответственности и никаких других.

```
# Неправильно
class EventHandler: # Обработчик событий
    def handle_event_1(self, event):
        # Обработчик первого события
        pass

    def handle_event_2(self, event):
        # Обработчик второго события
        pass

    def handle_event_3(self, event):
        # Обработчик третьего события
        pass

    def database_logger(self, event):
        # Метод для записи логов в базу данных
        pass


# Правильно
class  EventHandler: # Обработчик событий

    def handle_event_1(self, event):
        # Обработчик первого события
        pass

    def handle_event_2(self, event):
        # Обработчик второго события
        pass

    def handle_event_3(self, event):
        # Обработчик третьего события
        pass


class DatabaseLogger:

    def database_logger(self, event):
        # Метод для записи логов в базу данных
        pass

```

Принцип открытости/закрытости (The Open Closed Principle): классы должны быть открыты для расширения, но закрыты для изменения. Этот принцип является важным,

потому что внесение изменений в существующие компоненты системы может также привести к непредвиденным изменения в работе самой этой системы. Однако поведение существующих объектов при необходимости можно расширить при помощи создания новых сущностей.

Рассмотрим на примере. Пусть существует класс Robot. У этого класса есть метод brake. Мы хотим создать робота, который при поломке кроме всего прочего включает аварийную сигнализацию alarm. При этом мы не должны переписывать сам класс Robot, а должны создать потомка AlarmRobot, который при вызове break после вызова соответствующего метода родительского класса будет так же вызывать метод alarm.

Принцип подстановки Барбары Лисков (The Liskov Substitution Principle): функции, которые используют базовый тип должны иметь возможность использовать его подтипы не зная об этом.

```
42  # Неправильный код
43  class Parent:
44      def __init__(self, value):
45          self.value = value
46
47      def do_something(self):
48          print("Function was called")
49
50
51  class Child(Parent):
52
53      def do_something(self):
54          super().do_something()
55          self.value = 0
56
57
58  def function(obj: Parent):
59      obj.do_something()
60      if obj.value > 0:
61          print("All correct!")
62      else:
63          print("SOMETHING IS GOING WRONG!")
64
65  # Посмотрим на поведение
66  parent = Parent(5)
67  function(parent)
68  print()
69
70  # Данный код должен работать корректно, если вместо родителя подставить потомка
71  child = Child(5)
72  function(child)
73  print()
```

Принцип разделения интерфейса (The Interface Segregation Principle): клиенты не должны зависеть от методов, которые они не используют.

```
74  # Неправильно
75  class AllScoresCalculator:
76      def calculate_accuracy(self, y_true, y_pred):
77          return sum(int(x == y) for x, y in zip(y_true, y_pred)) / len(y_true)
78
79      def log_loss(self, y_true, y_pred):
80          return sum((x * math.log(y) + (1 - x) * math.log(1 - y))
81                     for x, y in zip(y_true, y_pred)) / len(y_true)
82
83
84  # Правильно
85  class CalculateLosses:
86      def log_loss(self, y_true, y_pred):
87          return sum((x * math.log(y) + (1 - x) * math.log(1 - y))
88                     for x, y in zip(y_true, y_pred)) / len(y_true)
89
90
91  class CalculateMetrics:
92      def calculate_accuracy(self, y_true, y_pred):
93          return sum(int(x == y) for x, y in zip(y_true, y_pred)) / len(y_true)
94
```

Принцип инверсии зависимостей (The Dependency Inversion Principle):

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Приведем пример. Пусть у вас есть базовый класс Distributer, который может отправлять сообщения в различные социальные сети. У этого класса есть несколько реализаций, например VKDistributer и OKDistributer. Согласно принципу инверсии зависимостей, эти реализации не должны зависеть от методов класса Distributer (например VK_send_message и OK_send_message). Вместо этого у класса Destributor должен быть объявлен общий абстрактный метод send_message, который и будет реализован отдельно в каждом из потомков.