Мини-курс по принципам SOLID

Руководство по проектированию кода на Python и C#

Создано для изучения принципов качественного ООП

Содержание

1 Что такое SOLID?

SOLID — это акроним из пяти принципов объектно-ориентированного программирования, предложенных Робертом Мартином. Они помогают создавать гибкий, масштабируемый и поддерживаемый код. Представьте SOLID как правила организации офиса: каждый сотрудник (класс) выполняет свою задачу, не мешая другим, и легко адаптируется к изменениям.

1.1 Зачем нужны SOLID?

- **Читаемость**: Код легче понимать.
- **Гибкость**: Легко добавлять новые функции.
- **Поддержка**: Упрощает исправление ошибок.
- **Масштабируемость**: Код выдерживает рост проекта.

2 Базовая терминология

Ответственность Одна задача, за которую отвечает класс.

Интерфейс Контракт, определяющий поведение.

Абстракция Обобщённое описание сущности.

Наследование Использование свойств базового класса.

Инверсия зависимостей Зависимость от абстракций, а не реализаций.

3 Принципы SOLID

3.1 S: Принцип единственной ответственности

Класс должен иметь только одну причину для изменения, то есть одну ответственность. Это уменьшает связанность и упрощает поддержку.

Аналогия: Работник в офисе должен выполнять одну задачу (например, бухгалтерия), а не всё сразу (бухгалтерия, уборка, продажи).

3.1.1 Неправильный подход (Python)

```python
# Класс делает всё: обработка заказа, печать, отправка
class Order:
    def __init__(self, id, items):
        self.id = id
        self.items = items

    def process_order(self):
        print(f"Обработка заказа {self.id}")
        # Логика обработки

    def print_order(self):
        print(f"Печать заказа {self.id}: {self.items}")

    def send_email(self):
        print(f"Отправка email для заказа {self.id}")
```

Проблемы: Класс отвечает за обработку, печать и отправку email. Изменение одного аспекта (например, формата email) затрагивает весь класс.

3.1.2 Правильный подход (Python)

```python
class Order:
    def __init__(self, id, items):
        self.id = id
        self.items = items

    def process_order(self):
        print(f"Обработка заказа {self.id}")

class OrderPrinter:
    def print_order(self, order):
        print(f"Печать заказа {order.id}: {order.items}")

class EmailSender:
    def send_email(self, order):
        print(f"Отправка email для заказа {order.id}")
```

Преимущества: Каждый класс имеет одну ответственность. Изменения в печати не затрагивают обработку заказа.

3.1.3 Неправильный подход (C#)

```csharp
// Класс выполняет несколько задач
public class Order
{
    public int Id { get; set; }
    public List<string> Items { get; set; }

    public void ProcessOrder()
    {
        Console.WriteLine($"Обработка заказа {Id}");
    }

    public void PrintOrder()
    {
        Console.WriteLine($"Печать заказа {Id}: {string.Join(", ", Items)}");
    }

    public void SendEmail()
    {
        Console.WriteLine($"Отправка email для заказа {Id}");
    }
}
```

Проблемы: Нарушение SRP; класс перегружен задачами.

3.1.4 Правильный подход (C#)

```csharp
public class Order
{
    public int Id { get; set; }
    public List<string> Items { get; set; }

    public void ProcessOrder()
    {
        Console.WriteLine($"Обработка заказа {Id}");
    }
}

public class OrderPrinter
```

```
{
    public void PrintOrder(Order order)
    {
        Console.WriteLine($"Печать заказа {order.Id}: {string.Join(", ",
order.Items)}");
    }
}

public class EmailSender
{
    public void SendEmail(Order order)
    {
        Console.WriteLine($"Отправка email для заказа {order.Id}");
    }
}
```

Преимущества: Разделение задач упрощает тестирование и поддержку.

3.2 O: Принцип открытости/закрытости

Классы должны быть открыты для расширения (добавления функциональности), но закрыты для модификации (изменения исходного кода).

Аналогия: Офисное ПО можно дополнять плагинами, не переписывая программу.

3.2.1 Неправильный подход (Python)

```python
class PaymentProcessor:
    def process_payment(self, payment_type, amount):
        if payment_type == "credit_card":
            print(f"Обработка кредитной карты: {amount}")
        elif payment_type == "paypal":
            print(f"Обработка PayPal: {amount}")
        # Для нового типа нужно менять класс
```

Проблемы: Добавление нового типа оплаты (например, Bitcoin) требует изменения кода класса.

3.2.2 Правильный подход (Python)

```python
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

class CreditCardProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Обработка кредитной карты: {amount}")

class PayPalProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Обработка PayPal: {amount}")

# Новый тип оплаты добавляется без изменения
class BitcoinProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Обработка Bitcoin: {amount}")
```

Преимущества: Новые типы оплаты добавляются через наследование, без изменения базового класса.

3.2.3 Неправильный подход (C#)

```
public class PaymentProcessor
{
    public void ProcessPayment(string paymentType, decimal amount)
    {
        if (paymentType == "CreditCard")
            Console.WriteLine($"Обработка кредитной карты: {amount}");
        else if (paymentType == "PayPal")
            Console.WriteLine($"Обработка PayPal: {amount}");
    }
}
```

Проблемы: Новый тип оплаты требует изменения метода.

3.2.4 Правильный подход (C#)

```
public interface IPaymentProcessor
{
    void ProcessPayment(decimal amount);
}

public class CreditCardProcessor : IPaymentProcessor
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Обработка кредитной карты: {amount}");
    }
}

public class PayPalProcessor : IPaymentProcessor
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Обработка PayPal: {amount}");
    }
}

// Новый тип оплаты
public class BitcoinProcessor : IPaymentProcessor
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Обработка Bitcoin: {amount}");
    }
}
```

Преимущества: Расширение через новые классы, без изменения интерфейса.

3.3 L: Принцип подстановки Лисков

Подтипы должны быть взаимозаменяемы с базовыми типами без изменения поведения программы.

Аналогия: Если у вас есть утка (базовый тип), её можно заменить на резиновую утку (подтип) в ванной, но не на самолёт.

3.3.1 Неправильный подход (Python)

```python
class Bird:
    def fly(self):
        print("Птица летит")

class Ostrich(Bird):
    def fly(self):
        raise Exception("Страус не летает")
```

Проблемы: `Ostrich` нарушает контракт `Bird`, так как не может летать, ломая ожидания.

3.3.2 Правильный подход (Python)

```python
from abc import ABC, abstractmethod

class Bird(ABC):
    @abstractmethod
    def move(self):
        pass

class Sparrow(Bird):
    def move(self):
        print("Воробей летит")

class Ostrich(Bird):
    def move(self):
        print("Страус бежит")
```

Преимущества: `move` подходит всем птицам, обеспечивая взаимозаменяемость.

3.3.3 Неправильный подход (C#)

```csharp
public class Bird
{
    public virtual void Fly()
    {
        Console.WriteLine("Птица летит");
    }
}

public class Ostrich : Bird
{
    public override void Fly()
    {
        throw new Exception("Страус не летает");
    }
}
```

Проблемы: `Ostrich` нарушает поведение `Bird`.

3.3.4 Правильный подход (C#)

```csharp
public interface IBird
{
    void Move();
}

public class Sparrow : IBird
```

```
{
    public void Move()
    {
        Console.WriteLine("Воробей летит");
    }
}

public class Ostrich : IBird
{
    public void Move()
    {
        Console.WriteLine("Страус бежит");
    }
}
```

Преимущества: Интерфейс `IBird` обеспечивает корректное поведение.

3.4 I: Принцип разделения интерфейсов

Клиенты не должны зависеть от интерфейсов, которые они не используют.

Аналогия: Меню в ресторане должно предлагать только блюда, которые хочет клиент, а не весь каталог.

3.4.1 Неправильный подход (Python)

```python
class Worker:
    def work(self):
        pass
    def eat(self):
        pass

class Robot(Worker):
    def work(self):
        print("Робот работает")
    def eat(self):
        raise Exception("Робот не ест")
```

Проблемы: `Robot` вынужден реализовать ненужный метод `eat`.

3.4.2 Правильный подход (Python)

```python
from abc import ABC, abstractmethod

class Workable(ABC):
    @abstractmethod
    def work(self):
        pass

class Eatable(ABC):
    @abstractmethod
    def eat(self):
        pass

class Robot(Workable):
    def work(self):
        print("Робот работает")

class Human(Workable, Eatable):
    def work(self):
        print("Человек работает")
```

```
    def eat(self):
        print("Человек ест")
```

Преимущества: Robot реализует только Workable, избегая ненужных методов.

3.4.3 Неправильный подход (C#)

```
public interface IWorker
{
    void Work();
    void Eat();
}

public class Robot : IWorker
{
    public void Work()
    {
        Console.WriteLine("Робот работает");
    }

    public void Eat()
    {
        throw new NotImplementedException("Робот не ест");
    }
}
```

Проблемы: Robot реализует ненужный метод Eat.

3.4.4 Правильный подход (C#)

```
public interface IWorkable
{
    void Work();
}

public interface IEatable
{
    void Eat();
}

public class Robot : IWorkable
{
    public void Work()
    {
        Console.WriteLine("Робот работает");
    }
}

public class Human : IWorkable, IEatable
{
    public void Work()
    {
        Console.WriteLine("Человек работает");
    }

    public void Eat()
    {
        Console.WriteLine("Человек ест");
    }
}
```

Преимущества: Разделённые интерфейсы упрощают реализацию.

3.5 D: При recitedнцип инверсии зависимостей

Модули высокого уровня не должны зависеть от низкоуровневых; оба должны зависеть от абстракций.

Аналогия: Офисный принтер зависит от стандарта USB, а не от конкретной модели.

3.5.1 Неправильный подход (Python)

```python
class MySQLDatabase:
    def save(self, data):
        print(f"Сохранение {data} в MySQL")

class UserService:
    def __init__(self):
        self.db = MySQLDatabase()  # Прямая зависимость

    def save_user(self, user):
        self.db.save(user)
```

Проблемы: `UserService` зависит от конкретной реализации `MySQLDatabase`, что затрудняет замену базы данных.

3.5.2 Правильный подход (Python)

```python
from abc import ABC, abstractmethod

class Database(ABC):
    @abstractmethod
    def save(self, data):
        pass

class MySQLDatabase(Database):
    def save(self, data):
        print(f"Сохранение {data} в MySQL")

class MongoDatabase(Database):
    def save(self, data):
        print(f"Сохранение {data} в MongoDB")

class UserService:
    def __init__(self, db: Database):
        self.db = db

    def save_user(self, user):
        self.db.save(user)
```

Преимущества: `UserService` зависит от абстракции `Database`, что позволяет менять реализацию.

Неправильный подход (C#)

```csharp
public class MySQLDatabase
{
    public void Save(string data)
    {
        Console.WriteLine($"Сохранение {data} в MySQL");
    }
}

public class UserService
{
```

```
    private MySQLDatabase _db = new MySQLDatabase();

    public void SaveUser(string user)
    {
        _db.Save(user);
    }
}
```

Проблемы: Жёсткая зависимость от `MySQLDatabase`.

Правильный подход (C#)

```
public interface IDatabase
{
    void Save(string data);
}

public class MySQLDatabase : IDatabase
{
    public void Save(string data)
    {
        Console.WriteLine($"Сохранение {data} в MySQL");
    }
}

public class MongoDatabase : IDatabase
{
    public void Save(string data)
    {
        Console.WriteLine($"Сохранение {data} в MongoDB");
    }
}

public class UserService
{
    private readonly IDatabase _db;

    public UserService(IDatabase db)
    {
        _db = db;
    }

    public void SaveUser(string user)
    {
        _db.Save(user);
    }
}
```

Преимущества: Инверсия зависимостей через интерфейс `IDatabase`.

4 Полезные советы

- **Рефакторинг**: Постепенно применяйте SOLID к существующему коду.
- **Тестирование**: SOLID упрощает написание модульных тестов.
- **Паттерны**: Используйте паттерны (например, DI-контейнеры) для реализации SOLID.

5 Справочник принципов SOLID

Принцип	**Описание**	**Пример**
Single Responsibility Principle	Класс имеет одну ответственность.	Разделение `Order`, `OrderPrinter`, `EmailSender`.
Open/Closed Principle	Открыт для расширения, закрыт для изменения.	Интерфейс `PaymentProcessor` с реализациями `CreditCardProcessor`.
Liskov Substitution Principle	Подтипы взаимозаменяемы с базовыми.	`Bird` с методом `move` для `Sparrow` и `Ostrich`.
Interface Segregation Principle	Клиенты не зависят от ненужных интерфейсов.	Разделение `Workable` и `Eatable` для `Robot`.
Dependency Inversion Principle	Зависимость от абстракций.	`UserService` зависит от `IDatabase`, а не `MySQLDatabase`.