УРОК №14. ПРИНЦИПЫ SOLID.

Учебные материалы

Оцени учебный материал:

(Среднее: **5,00**)

1. Почему принципы SOLID важны?

Прежде чем мы углубимся в суть принципов SOLID, расставим все точки над «i». Зачем нам вообще нужны принципы SOLID?

Чтобы ответить на этот вопрос, представьте себя архитектором. Ваша задача — спроектировать прочное и практичное здание, которое будет служить своим жильцам долгие годы. Проект должен быть простым для понимания, рациональным и достаточно прочным, чтобы выдержать испытание временем и изменения в требованиях.

Теперь перенесите этот сценарий в мир разработки программного обеспечения. Разработчики программного обеспечения также являются архитекторами — мы создаем приложения и системы, которые люди используют для производительности, развлечений или решения проблем. Как и в физической архитектуре, качественный дизайн программного обеспечения имеет решающее значение. Он делает систему легкой для понимания, эффективной в работе, адаптивной к изменениям и достаточно надежной, чтобы выдержать испытание временем.

Именно здесь пригодились принципы SOLID.

> **Принципы SOLID**
>
> Это набор руководящих принципов проектирования, которые помогают создавать легко поддерживаемое, масштабируемое и надежное программное обеспечение. Ввел их Роберт Мартин, инженер-программист, обративший внимание на повторяющиеся проблемы в проектах, над которыми работал, и захотел их решить. С годами эти принципы стали ценным набором важных практик, помогающих разработчикам избегать распространенных ошибок в коде.

Роберт С. Мартин, также известный как «Дядя Боб», является легендарной фигурой в области программной инженерии, признанной за его вклад в разработку многих принципов проектирования программного обеспечения и влияние на программное мастерство. Мартин, который работает профессиональным программистом и

консультантом в сфере ПО с середины 1970-х годов, является автором многих влиятельных книг, среди которых «Чистый код», «Чистая архитектура» и «Чистый разработчик». В начале 2000-х годов он представил принципы SOLID — аббревиатуру, которая, по его замыслу, инкапсулирует пять фундаментальных принципов объектно-ориентированного программирования и дизайна. С тех пор эти принципы стали важным стандартом для профессиональных разработчиков программного обеспечения во всем мире, ведь они помогают им создавать системы, которые легко поддерживать и масштабировать со временем и, которыми просто управлять. Принципами SOLID, как и другими наработками Мартина, продолжают руководствоваться новые поколения разработчиков, которые стремятся создавать код не только функциональным, но и хорошо структурированным и надежным.

Эти принципы, при правильном применении, позволяют разработчикам:

- **Уменьшить риск ошибок.** Поскольку принципы SOLID предусматривают обособленную и инкапсулированную разработку, они минимизируют эффект волны, когда изменение в одной части системы вызывает появление ошибок в других частях.
- **Улучшить дизайн системы.** Принципы SOLID помогают разработчикам создавать более понятные и гибкие проекты. Это, в свою очередь, уменьшает сложность системы и облегчает ее обслуживание и совершенствование.
- **Повысить адаптивность.** Поскольку требования к программному обеспечению часто меняются, принципы SOLID помогают разрабатывать ПО, которое бы адаптировалось к новым требованиям, не требуя капитального ремонта.
- **Содействовать командной работе.** Систему, разработанную по принципам SOLID, легче понимать, а значит, с ней легче работать команде. Это уменьшает время обучения для новых членов команды и повышает производительность.
- **Повторно использовать код.** Принципы SOLID ориентируют разработчиков на модульный дизайн, где компоненты являются независимыми и могут быть повторно использованы в разных частях системы или даже в других системах.

Так же, как архитектор не начнет строить без набора правил или рекомендаций, разработчик ПО не должен начинать кодирование без понимания принципов SOLID. Они служат дорожной картой для создания надежного, эффективного и удобного в обслуживании программного обеспечения, которое выдержит испытание временем, как и любое хорошо построенное здание.

SOLID — это аббревиатура, представляющая пять принципов объектно-ориентированного программирования и дизайна. Каждая буква в SOLID соответствует определенному принципу, которые в совокупности являются руководством для написания чистого, масштабируемого и поддерживаемого кода. «S» означает Single Responsibility Principle *(Принцип единой ответственности*), предполагающий, что

класс должен иметь только одну работу или ответственность. «O» означает Open-Closed Principle *(Принцип открытости-закрытости*), предполагающий, что программные объекты должны быть открытыми для расширения, но закрытыми для модификации. «L» означает Liskov Substitution Principle *(Принцип замещения Лисков*), указывающий на то, что подклассы должны быть заменяемыми для своих базовых классов без изменения корректности программы. «I» означает Interface Segregation Principle *(Принцип разделения интерфейсов*), который поощряет использование многих специфических интерфейсов вместо одного интерфейса общего назначения. «D» означает Dependency Inversion Principle *(Принцип инверсии зависимостей*), который продвигает идею о том, что высокоуровневые модули не должны зависеть от низкоуровневых модулей, но оба должны зависеть от абстракций.

Рассмотрим подробно каждый из принципов SOLID поочередно.

2. S-принцип единой ответственности

Принцип единой ответственности *(Single Responsibility Principle*, сокращенно — SRP) соответствует букве «S» в SOLID и формирует основу того, как мы структурируем наши классы и модули в программной системе.

Согласно этому принципу у класса должна быть одна и только одна причина для изменения. Это означает, что класс должен иметь только одну работу или обязанность. Когда мы говорим «причина для изменений», мы имеем в виду то, как определенная функция или аспект системы может развиваться со временем. Эта эволюция может быть связана с изменением требований, исправлением ошибок или добавлением улучшений.

Рассмотрим пример: представим класс в приложении, которое обрабатывает данные о сотрудниках. Согласно SRP, если этот класс отвечает и за расчет зарплаты работника, и за печать его данных, то он имеет более чем одну ответственность. Такой дизайн может привести к проблемам в будущем. Если нужно изменить способ расчета зарплаты, есть риск, что это непреднамеренно повлияет на функциональность печати. Аналогично, изменение макета печати может вызвать неожиданную проблему в расчете заработной платы.

Итак, в соответствии с SRP, эти обязанности следует разделить на разные классы. Один класс мог бы заниматься расчетами заработной платы, а другой — печатью данных о работниках. Таким образом, изменения в одной области с меньшей вероятностью повлияют на другие, что сделает систему более надежной, более легкой для понимания, обслуживания и тестирования.

Принцип единой ответственности способствует созданию более организованной системы, где каждый класс имеет четкую и определенную роль. Эта простота и ясность может облегчить навигацию

и поддержку системы, а также уменьшить вероятность появления ошибок при внесении изменений. Он также может облегчить параллельную работу различных разработчиков или команд, поскольку изменения в одном классе с меньшей вероятностью повлияют на другой класс.

Теория — это хорошо, но лучше увидеть код, чтобы все понять. Каждый принцип мы будем разбирать таким образом: фрагмент кода с проблемами -> решение с использованием принципа. Пожалуйста, прежде чем читать правильное решение, попробуйте понять, что не так с кодом, и только после этого читайте описание проблемы и способ ее решения.

Вот наш код для буквы S. Код находится в файле *S principle wrong way.py*, который прилагается к этому уроку:

```
class User:
    def __init__(self, name, last_name, age):
        self.name = name
        self.last_name = last_name
        self.age = age

    # Getter for the age attribute.
    @property
    def age(self):
        return self._age

    # Setter for the age attribute, also validates the age is in a valid range.
    @age.setter
    def age(self, age):
        if age < 0or age >= 130:
            raise ValueError("Age must be between 0 and 130 ")
        self._age = age

    # Displays user information to the console.
    def display(self):
        print(f"{self.name}{self.last_name}{self.age}")

    # Reads user input from the console to create or update a User object.
    def input(self):
        self.name = input("Input name:")
        self.last_name = input("Input last name:")
        self.age = int(input("Input age:"))

# Create a User object, display its information,
# update it with user input, then display it again.
obj = User("Bill", "Windows", 34)
obj.display()
obj.input()
obj.display()
```

Итак, в нашем примере есть класс User. Внутри этого класса есть свойства: name, last_name, age. Также в этом классе есть 2 метода: display, input. Метод display отображает данные на экран.

Метод input отображает данные с клавиатуры. В другой части кода мы создаем объект User и работаем с ним. Все бы ничего, но класс User нарушает принцип SRP. Давайте обсудим почему.

Приведенный класс User нарушает принцип единой ответственности. Согласно SRP, класс должен иметь только одну причину для изменения, что означает единую ответственность. Однако класс User, о котором идет речь, отвечает за несколько разных сфер.

Во-первых, он управляет состоянием пользователя. Это отображается через свойства name, last_name и age, которые определяют состояние пользователя.

Во-вторых, он участвует в операциях ввода/вывода данных пользователем. Методы ввода и отображения отвечают за получение данных от пользователя из консоли и отображение данных пользователя соответственно.

Эти две обязанности существенно отличаются друг от друга. Например, изменение атрибутов пользователя или способа управления этими атрибутами приведет к изменению класса, который отвечает за управление состоянием пользователя. С другой стороны, изменение формата отображения пользовательских данных или способа их ввода приведет к изменению класса, связанного с операциями ввода/вывода пользователя.

Класс User обрабатывает более чем один тип причин изменений, а следовательно, он выполняет более чем одну обязанность. Это нарушает принцип единой ответственности. Не спешите и подумайте, как улучшить класс User, чтобы он не нарушал принцип SRP.

Вот наше решение для буквы S. Код находится в файле *S principle right way.py*, который прилагается к этому уроку:

```
# User class is responsible for maintaining user's state and validating age.
class User:
    def __init__(self, name, last_name, age):
        self.name = name
        self.last_name = last_name
        self.age = age

    @property
    def age(self):
        return self._age

    # Setter for age includes validation.
    @age.setter
    def age(self, age):
        if age < 0or age >= 130:
            raise ValueError("Age must be between 0 and 130 ")
        self._age = age

# Console class handles user I/O operations.
class Console:
    # Displays user details.
    @staticmethod
```

```
        def display(obj):
            print(f"{obj.name}{obj.last_name}{obj.age}")

        # Reads user input from console and returns a User object.
        @staticmethod
        def input():
            name = input("Input name:")
            last_name = input("Input last name:")
            age = int(input("Input age:"))
            return User(name, last_name, age)

# Create a User, display it, take console input to create a new User,
# then display the new User.
obj = User("Bill", "Windows", 34)
Console.display(obj)
obj = Console.input()
Console.display(obj)
```

Приведенный код помогает понять, как работать с принципом единой ответственности. Код состоит из двух классов, User и Console, каждый из которых имеет свою отдельную область ответственности.

Класс User предназначен для управления и поддержки состояния пользователя, который включает свойства name, last_name и age. Он также предоставляет setter для свойства age, который включает проверку возраста, а именно находится ли он в пределах определенного диапазона. В этом классе единственной ответственностью или причиной для изменений будет все, что влияет на то, как управляется или проверяется состояние пользователя.

С другой стороны, класс Console отвечает исключительно за обработку операций ввода и вывода данных пользователем. Он содержит два статических метода: display, который отображает данные пользователя, и input, который считывает данные пользователя из консоли и возвращает новый объект User. Здесь единственной ответственностью или причиной для изменений будет все, что связано с тем, как пользовательские данные принимаются в качестве входных или отображаются в качестве выходных.

Разделив эти две проблемы на разные классы, мы следуем стандарту SRP. Каждый класс теперь имеет только одну причину для изменений. Это упрощает разработку системы, делая ее более надежной, более простой в обслуживании, понимании и тестировании. Любое изменение или эволюция в процессе управления состоянием пользователя повлияет только на класс User, а любое изменение в обработке ввода/вывода повлияет только на класс Console.

Принцип единой ответственности является чрезвычайно важным правилом для разработки надежного, легкого в обслуживании и понятного программного обеспечения. Принцип SRP предлагает определять классы таким образом, чтобы каждый класс отвечал за одну задачу или функцию, имея только одну причину для изменений.

Рассмотрев пример, мы убедились, что нарушение SRP может привести к появлению классов, которые сложно поддерживать и которые подвержены неожиданным проблемам. Классы с несколькими обязанностями увеличивают риск непредсказуемых последствий при внесении изменений, что делает код более сложным для тестирования, понимания и отладки.

Однако при соблюдении SRP каждый класс становится проще и более сфокусированным. Изменения в одной области системы с меньшей вероятностью будут иметь непредсказуемые последствия для других областей, что обеспечит более стабильную и поддерживаемую кодовую базу.

Разделяя задачи, как в случае с классами User и Console в правильном варианте, мы способствуем модульности и читабельности кода, а также делаем нашу систему более гибкой и приспособленной к будущим изменениям.

2.1. Преимущества применения принципа единой ответственности

- **Снижение сложности.** Когда класс или модуль имеет только одну причину для изменения, его легче понять и отладить. Это уменьшает когнитивную нагрузку на разработчиков, когда они рассматривают фрагмент кода.
- **Повышение слаженности.** Благодаря SRP код становится слаженным. Когда каждый модуль или класс отвечает за одну функцию, он становится более согласованным и, следовательно, более надежным.
- **Улучшенная модульность.** Классы или модули, которые соответствуют принципу SRP, можно легко изолировать и подключить к другим системам или модулям, что повышает модульность программного обеспечения.
- **Более простое обслуживание.** Благодаря SRP, когда нужно внести изменения, мы четко знаем где их вносить. Такая точность упрощает обслуживание и уменьшает вероятность нежелательных побочных эффектов при изменении сферы ответственности.

2.2. Как применять принцип единой ответственности?

В рамках Python есть несколько способов придерживаться SRP:

- **Распределение задач.** Убедитесь, что на простейшем уровне различные функциональные возможности размещены в отдельных функциях или методах. Это позволит каждой функции или методу иметь единую, четкую цель.

- **Организация модулей.** Организуйте свой код с помощью модулей. Если определенная функциональность или задача становится слишком сложной, ее можно разделить на отдельные модули, что гарантирует, что каждый модуль будет соответствовать SRP.
- **Использование классов.** Если необходимо, инкапсулируйте одну ответственность в рамках класса. Таким образом, у нас будет четко определенная ответственность для каждого класса, и у каждого класса будет только одна причина для изменений.
- **Делегируйте обязанности.** Вместо того, чтобы иметь один класс, который выполняет несколько задач, создайте вспомогательные классы или утилиты, которые выполняют конкретные обязанности. Это способствует соблюдению принципа SRP, а также позволяет повторно использовать код.

Изучая принципы SOLID, помните, что это — не жесткие правила, а скорее рекомендации, направленные на повышение качества вашего кода. Применяя эти принципы, включая SRP, всегда учитывайте конкретный контекст и потребности вашего проекта.

3. O-принцип открытости-закрытости

Принцип открытости/закрытости *(Open/Closed Principle, OCP*) соответствует букве «O» в SOLID. Это — фундаментальная концепция объектно-ориентированного проектирования, которая играет решающую роль в обеспечении масштабируемости, сопровождения и адаптации программного обеспечения. Суть принципа заключается в следующем: «Объекты программного обеспечения (классы, модули, функции и т.д.) должны быть открытыми для расширения, но закрытыми для модификации».

Проанализируем это определение подробнее, чтобы действительно его понять.

3.1. Открытый для расширения

Это означает, что поведение модуля может быть расширено. Если требования вашего программного обеспечения меняются, вы должны иметь возможность добавлять новые функции, не затрагивая существующий код. То есть, у вас должна быть возможность добавлять новые функции или поведение, не изменяя существующий код.

3.2. Закрытый для модификации

Это означает, что после того, как модуль был разработан и протестирован, его поведение нельзя изменять. Это не означает, что его нельзя изменить, а скорее, что изменения не следует вносить в существующий код, если только не обнаружена ошибка. Вместо

модификации вы расширяете модуль новыми функциональными возможностями.

3.3. Пример нарушения OCP из реальной жизни

Представьте, что у вас есть кофемашина, которая может приготовить чашку черного кофе. Все его любят, всем он подходит.

Однажды к вам заходит друг и просит сделать ему эспрессо. Ваша кофемашина не готовит эспрессо. Вы модифицируете ее, добавляете несколько ручек и кнопок и теперь она готовит эспрессо. Отлично!

Через неделю другой друг просит капучино. Опять же, ваша кофейная машина не может его приготовить. Итак, вы настраиваете ее снова, добавляете капучинатор и еще несколько настроек. Теперь она также готовит капучино.

Поскольку все больше и больше друзей просят разные виды кофе — латте, мокко, холодный кофе — вы постоянно модифицируете одну и ту же машину, добавляя все больше и больше компонентов и настроек, чтобы удовлетворить все запросы.

3.3.1. В чем было нарушение OCP?

Ваша кофемашина нарушает принцип открытости/закрытости. Почему? Потому что каждый раз, когда появляется новое требование (тип кофе), вы модифицируете существующую машину. Со временем это может привести к созданию очень сложной и хрупкой системы. Что, если в одной из ваших модификаций первоначальная простая настройка кофе перестанет работать? Или настройка эспрессо мешает настройке капучино?

Вместо этого, дизайн, который соответствует принципу OCP, предусматривает базовую кофемашину, функции которой можно расширять. Если вам нужна новая функция (например, вспениватель молока для капучино), вы добавите расширение или новый модуль, не меняя основную, проверенную и работающую функциональность оригинальной кофемашины.

Эта аналогия имеет целью показать, что постоянная модификация системы для добавления новых функций (вместо того, чтобы расширять ее) может привести к созданию сложных, нестабильных систем, от чего и уберегает принцип открытости/закрытости.

3.3.2. Пример нарушения OCP в коде

Начнем с неправильного пути и посмотрим, в чем проблема. Вот наш код для буквы O. Код находится в файле *O principle wrong way.py*,

который прилагается к этому уроку:

```
import io
import os

# The Output class is responsible for managing the output of data
# to different types of destinations.
class Output:
    def __init__(self, data, output_type):
        # output_type determines the destination type for the data.
        self.output_type = output_type
        self.data = data

    # The display method outputs the data to the chosen destination.
    def display(self):
        # If output_type is "console", the data is printed to the console.
        if self.output_type == "console":
            print(f"{self.data}")
        # If output_type is "file", the data is written to
        # a file in the script's directory.
        elif self.output_type == "file":
            # Get the directory that this script is in.
            file_dir = os.path.dirname(os.path.abspath(__file__))
            # Change the working directory to the directory of the script.
            os.chdir(file_dir)
            # Open the output file and write the data to it.
            with open('output.txt', 'w') as f:
                # Write the data to the file.
                f.write(self.data)
        # If output_type is neither "console" nor "file", raise an error.
        else:
            raise ValueError("Wrong type of output")

# Create an Output object with data "some string"
# and output_type "file", then call its display method.
obj = Output("some string", "file")
obj.display()
```

Приведенный код определяет класс Output, который управляет выводом данных в разные места назначения на основе типа output_type. Этот тип определяет, будут ли данные выводиться на консоль или записываться в файл. Код инициализирует объект Output для вывода данных «some string» в файл, а затем вызывает его метод отображения для выполнения операции вывода.

Однако такая конструкция нарушает принцип открытости/закрытости. Причина в том, что каждый раз, когда мы хотим поддерживать новый тип назначения вывода, внутренняя логика класса Output нуждается в модификациях. Например, чтобы добавить новое место назначения вывода, такое как «database» или «cloud», нужно будет скорректировать метод отображения, введя новое условие elif. Такая практика проблематична. Каждое изменение или модификация класса создает потенциальные риски и может нарушить существующие функциональные возможности. Согласно OCP, класс должен быть

открытым для расширения (например, добавление новых направлений вывода), но закрытым для модификации. Дизайн, более приближенный к OCP, предусматривает абстрагирование механизма отображения, возможно, путем использования отдельных классов или стратегий для каждого типа вывода. Это гарантирует, что при добавлении нового типа вывода можно будет создать новый класс или стратегию, не изменяя код существующего класса вывода.

Не спешите и подумайте, как улучшить наш код, чтобы он не нарушал принцип открытости/закрытости.

Вот наше решение для буквы O. Код находится в файле *right way.py*, который прилагается к этому уроку:

```
import io
import os
from abc import ABC, abstractmethod

# Abstract base class for outputs
class Output(ABC):
    def __init__(self, data):
        self.data = data

    # Declare an abstract display method
    @abstractmethod
    def display(self):
        pass

# Console output implementation
class ConsoleOutput(Output):
    def display(self):
        print(f"{self.data}")

# File output implementation
class FileOutput(Output):
    def display(self):
        with open('output.txt', 'w') as f:
            f.write(self.data)

# Create a ConsoleOutput object and display
obj = ConsoleOutput("some string")
obj.display()

# Create a FileOutput object and display
obj2 = FileOutput("another string")
obj2.display()
```

В приведенном коде определен абстрактный базовый класс Output, который представляет общее назначение вывода. Затем он предоставляет две конкретные реализации этого абстрактного класса: ConsoleOutput для вывода данных на консоль и FileOutput для записи данных в файл. Код демонстрирует использование этих конкретных реализаций, инициализируя объект Output для вывода данных «some string» на консоль и другой объект Output для записи «another string» в файл.

Этот код соответствует принципу открытости/закрытости, ведь если в будущем возникнет потребность в поддержке нового типа назначения вывода, вы сможете просто создать новый класс, унаследованный от абстрактного базового класса Output, без необходимости модифицировать существующие классы. Таким образом, система является открытой для расширения (добавление новых типов вывода), но закрытой для модификации. Использование абстрактного базового класса и определение конкретных реализаций для каждого типа вывода обеспечивает четкое разграничение задач и способствует модульному и масштабируемому подходу к проектированию программного обеспечения.

3.4. Преимущества применения принципа открытости/закрытости

- **Уменьшение рисков.** Внесение изменений в существующий код может привести к появлению ошибок. Избегая изменений в протестированном и рабочем коде, вы минимизируете риск появления новых ошибок.
- **Возможность повторного использования.** Код, соответствующий принципу OCP, является более модульным и его легко повторно использовать, благодаря чему ваша кодовая база не будет содержать лишнего.
- **Гибкость.** Программное обеспечение, разрабатываемое с учетом OCP лучше адаптируется к изменениям. Если требования к ПО меняются, вы сможете легко вносить коррективы без значительной переработки.
- **Более простое обслуживание.** По мере роста программного обеспечения его обслуживание может стать более сложным. Если придерживаться OCP, процесс обслуживания становится проще, поскольку новые функции или поведение добавляются без изменения существующего кода.

3.5. Как применить принцип открытости/закрытости?

Хотя мы не углубляемся в код, стоит отметить механизмы для соблюдения принципа открытости/закрытости:

- **Наследование.** Это один из основных способов расширения поведения в объектно-ориентированных языках. Наследуя от базового класса, производный класс может как использовать, так и расширять поведение базового класса.
- **Композиция.** Создание сложных объектов путем комбинирования более простых. Вместо того, чтобы модифицировать существующий объект, вы можете добавить новое поведение путем компоновки объектов.

- **Модель стратегии.** Предполагается определение семейства алгоритмов или поведений и их взаимозаменяемость. Модель стратегии может помочь вам добавлять новые поведения без изменения существующих.

Подводя итог, можем утверждать, что принцип открытости/закрытости подчеркивает важность написания кода, который мог бы расти и изменяться по мере развития потребностей, не требуя внесения изменений в существующий стабильный код. Этот принцип способствует созданию более надежных, адаптивных и поддержи-ваемых программных систем.

4. L-принцип замещения Лисков

Принцип замещения Лисков *(Liskov Substitution Principle, LSP)* соответствует букве «L» в SOLID. Его ввела Барбара Лисков в 1987 году. Этот принцип подчеркивает, что объекты суперкласса должны заменяться объектами подкласса, не влияя на корректность программы. То есть, если определенный объект рассматривается как принадлежащий к определенному классу, любой подкласс этого класса должен вести себя в соответствии с этим ожиданием.

Проще говоря, согласно принципу LSP каждый подкласс должен быть заменяемым для своего родительского класса. Если вы ожидаете от класса определенное поведение или набор поведений, любой из его подклассов должен иметь возможность быть задействованным и выполнить его, не вызывая неожиданных результатов.

Формально, LSP говорит нам, что для того, чтобы программа оставалась корректной, когда объект подкласса заменяется объектом суперкласса, подкласс должен придерживаться поведения и ожиданий суперкласса.

4.1. Пример нарушения LSP из реальной жизни

Представьте, что у вас есть обычные цифровые наручные часы, которые показывают время. Вы решили обновить их до усовершенствованных спортивных часов, которые не только показывают время, но и отслеживают ваш пульс, шаги и другую физическую активность. В обычных часах вы могли просто нажать кнопку, чтобы отрегулировать время. На новых спортивных часах нажатие той же кнопки может начать отслеживать новую активность, а не корректировать время.

4.1.1. В чем было нарушение LSP?

В примере с наручными часами базовые цифровые наручные часы создают прецедент или ожидание для пользователя: когда вы нажимаете определенную кнопку, вы можете настроить время. Новые

спортивные часы, которые являются более продвинутой версией (или «подклассом» с точки зрения ООП) наручных часов, должны учитывать поведение, заданное их предшественником, базовыми часами. Однако он изменяет это поведение, назначая кнопке новую функцию, тем самым нарушая ожидания пользователя.

С точки зрения принципа замещения Лисков, если спортивные часы считаются подклассом базовых наручных часов, то пользователи должны иметь возможность заменить свои базовые наручные часы на спортивные, но без изменения базовых функций, к которым они привыкли, например, в настройке времени. Переопределяя функцию кнопки, спортивные часы нарушают LSP. Это приводит к неожиданному поведению, что может вызвать путаницу или ошибки в работе.

4.1.2. Пример нарушения LSP в коде

Давайте начнем с неправильного пути и посмотрим, в чем проблема. Вот наш код для буквы L. Код находится в файле *L principle wrong way.py*, который прилагается к этому уроку:

```
    @height.setter
    def height(self, value):
        self._height = value

    # Calculate rectangle area
    def calculate_area(self):
    return self._width * self._height

# Square class, inherits from Rectangle
class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)

    @property
    def width(self):
        return self._width
```

```
    total_area = 0
    for obj in rects:
        total_area += obj.calculate_area()
    return total_area

rects = [Rectangle(5,6),Square(10)]
print (calculate total area(rects))
```

Приведенный код определяет две геометрические фигуры: прямоугольник и квадрат. Класс Rectangle предназначен для представления прямоугольника, который определяется его шириной и высотой. Этот класс имеет методы для установления и получения ширины и высоты, а также метод для вычисления площади.
Класс Square наследует класс Rectangle, но с ограничением, что ширина и высота квадрата всегда равны. В результате, когда задается ширина (или высота) квадрата, его высота (или ширина) подстраивается соответственно, благодаря чему сохраняются свойства объекта как квадрата. Для вычисления площади квадрата просто перемножается одна сторона на другую.

Существует также отдельная функция calculate_total_area, которая принимает список прямоугольников (который, благодаря наследованию, может включать также квадраты) и вычисляет суммарную площадь всех фигур в списке. В конце концов, код создает список, состоящий из прямоугольника с размерами 5 на 6 и квадрата с длиной стороны 10. Затем он вычисляет и выводит суммарную площадь этих двух фигур с помощью функции calculate_total_area.

Приведенный код нарушает принцип замещения Лисков из-за поведенческих расхождений, введенных подклассом Square.
Класс Rectangle позволяет независимо изменять ширину и высоту квадрата, а подкласс Square, который наследует Rectangle, объединяет эти два свойства. Изменение ширины квадрата невольно изменяет его высоту, и наоборот. Таким образом, при подстановке объекта Square вместо Rectangle такое взаимосвязанное поведение может привести к неожиданным результатам, что противоречит сути ООП, согласно которой объекты суперкласса должны заменяться объектами подкласса, не нанося ущерба корректности программы.

Не спешите и подумайте, как улучшить наш код, чтобы он не нарушал LSP.

Вот наше решение для буквы L. Код находится в файле *L principleright way.py*, который прилагается к этому уроку:

```
from typing import List
from abc import ABC, abstractmethod

# Define an abstract base class for figures
class Figure(ABC):
    # Abstract method for calculating area
    @abstractmethod
    def calculate_area(self):
```

```
        pass

# Rectangle class, a subtype of Figure
class Rectangle(Figure):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    # Concrete implementation of area calculation for rectangles
    def calculate_area(self):
        return self.width * self.height

# Square class, another subtype of Figure
class Square(Figure):
    def __init__(self, side):
        self.side = side

    # Concrete implementation of area calculation for squares
    def calculate_area(self):
        return self.side * self.side

# Function to calculate the total area of multiple figures
def calculate_total_area(figures:List[Figure]):
    total_area = 0
    for obj in figures:
        total_area += obj.calculate_area()
    return total_area

# Example usage
figures = [Rectangle(5,6),Square(10)]
print (calculate_total_area(figures))
```

Этот код предоставляет структуру для представления геометрических фигур и вычисления их площадей. В основе лежит абстрактный базовый класс Figure, который обязывает своих потомков реализовывать метод calculate_area. От этого базового класса происходят два конкретных типа фигур, Rectangle и Square, каждый из которых обеспечивает свою реализацию для вычисления площади. Функция calculate_total_area принимает список объектов-фигур (прямоугольников, квадратов или любых других будущих типов фигур) и вычисляет их общую площадь.

Код поддерживает принцип замещения Лисков в своем дизайне. В соответствии с LSP подтипы должны заменяться своими базовыми типами без изменений в корректности работы программы. В этом случае классы Rectangle и Square являются подтипами абстрактного базового класса Figure. Независимо от того, работаем мы с прямоугольником или квадратом, функция calculate_total_area может легко обработать каждый объект, полагаясь на полиморфное поведение метода calculate_area. Благодаря этому любой новый тип фигуры, если он соответствует условиям контракта Figure и правильно реализует метод calculate_area, может быть интегрирован без нарушения существующих функциональных возможностей — наглядное проявление LSP в действии.

4.2. Преимущества применения принципа замещения Лисков

- **Предсказуемость.** Принцип гарантирует, что подклассы могут быть заменены на их базовые классы, не провоцируя неожиданное поведение, благодаря чему разработчики могут полагаться на установленные контракты и гарантии базового класса. Такая согласованность позволяет избежать неожиданностей во время выполнения.
- **Поддерживаемость.** Благодаря соблюдению LSP, ментальная модель кода остается нетронутой. Разработчики могут рассчитывать на то, что подклассы будут вести себя в соответствии со свойствами своих родительских классов, что облегчает понимание, отладку и расширение кода без необходимости углубляться в специфику каждого подкласса.
- **Надежность.** Соблюдение принципа LSP гарантирует, что система остается корректной, когда объект подкласса заменяет объект родительского класса. Эта надежность позволяет избежать ошибок, которые могут возникнуть из-за неправильных предположений о поведении подкласса.
- **Гибкость.** LSP продвигает правильную иерархию наследования и полиморфизм. Эта философия проектирования позволяет разработчикам уверенно внедрять новые подклассы, зная, что они будут плавно интегрироваться в существующие системы, использующие родительский класс.
- **Уменьшение технического долга.** Нарушение LSP часто приводит к появлению обновлений (патчей), обходных путей и специального кода со временем, когда возникают проблемы. Соблюдение LSP позволяет избежать этих ловушек, сохраняя кодовую базу чистой и уменьшая будущий технический долг.

4.3. Как применить принцип замещения Лисков?

Избегая определенных фрагментов кода, крайне важно разбираться в основных стратегиях поддержки LSP:

- **Типизируйте контракты.** Убедитесь, что методы подкласса поддерживают инварианты, предпосылки и постусловия методов базового класса, которые они переопределяют. Если метод базового класса ожидает выполнения определенных условий, его переопределенные версии в подклассах не должны требовать больше или обещать меньше.
- **Полиморфизм.** Используйте полиморфизм, который гарантирует, что производные классы действительно заменяют свои базовые классы. Этого можно достичь, когда переопределение методов в подклассах сохраняет семантическое значение базового класса.

- **Согласованность интерфейсов.** Все методы подклассов, которые переопределяют методы базового класса, должны иметь согласованные сигнатуры. Это касается типов параметров, типов возврата и генерируемых исключений.
- **Избегайте специальных проверок.** Если вы пишете код, который проверяет тип или идентичность объекта, чтобы определить его поведение, это красный флаг. Такие специальные проверки часто указывают на нарушение LSP, поскольку ожидается разное поведение для подклассов.

Принцип замещения Лисков подчеркивает важность беспрепятственного замещения подклассами своих базовых классов. Соблюдение LSP не только обеспечивает более плавную интеграцию, но и создает основу для построения надежных программных компонентов.

5. I-принцип разделения интерфейсов

Принцип разделения интерфейсов *(Interface Segregation Principle, ISP*) — один из пяти принципов SOLID. ISP предполагает, что «ни один клиент не должен быть вынужден зависеть от интерфейсов, которыми не пользуется». Это означает, что большие, более сложные интерфейсы следует разбивать на меньшие, более специфические, чтобы клиенту нужно было знать только о тех методах, которые его интересуют. Несоблюдение этого принципа может привести к раздутым интерфейсам, которые имеют больше шансов измениться, тем самым влияя на все классы, которые от них зависят.

5.1. Пример нарушения ISP из реальной жизни

Рассмотрим универсальный пульт дистанционного управления. Этот пульт разработан по принципу «все включено» — с кнопками для телевизора, радио, DVD-плеера, Blu-ray-плеера, системы домашнего кинотеатра и даже старых устройств, таких как VHS-плеер. Если у человека есть только телевизор и DVD-плеер, он будет использовать только часть кнопок, доступных на пульте. Остальные не используются и могут запутать пользователя, ведь приходится искать среди множества ненужных кнопок лишь несколько нужных.

5.1.1. В чем было нарушение ISP?

Универсальный пульт был разработан по принципу «все включено». Вместо создания нескольких отдельных пультов дистанционного управления или настроек с помощью кнопок для отдельных устройств, он объединяет все возможные функциональные возможности в одном интерфейсе. Это приводит к тому, что пользователи, как наш владелец телевизора или DVD, вынуждены сталкиваться со множеством

неиспользуемых кнопок и функций. Они вынуждены зависеть от интерфейсов (кнопок), которые не используют. Универсальный пульт дистанционного управления был бы более удобным для пользователя, если был бы создан с учетом принципа ISP. То есть, если бы позволял пользователям выбирать или настраивать кнопки устройства, которые им нужны, а не нагружать их всеми возможными опциями.

Давайте начнем с неправильного пути и посмотрим, в чем проблема. Вот наш код для буквы I. Код находится в файле *I principle wrongway.py*, который прилагается к этому уроку:

```
from abc import ABC, abstractmethod

# Define an abstract base class (ABC) for Bird with two abstract methods
class Bird(ABC):
    @abstractmethod
    def fly(self):
        pass

    @abstractmethod
    def walk(self):
        pass

# Ostriche is a type of Bird but cannot fly,
# so an exception is raised when fly method is called
class Ostriche(Bird):
    def fly(self):
        # Not an appropriate design,
        # violates Interface Segregation Principle (ISP)
        raise Exception("Ostriche is not flying")

    def walk(self):
        print("Ostriche is walking")

# Eagle is a type of Bird that can both fly and walk
class Eagle(Bird):
    def fly(self):
        print("Eagle is flying")

    def walk(self):
        print("Eagle is walking")

# Create instances and call methods
try:
    obj = Eagle()
    obj.fly()
    obj.walk()
    obj2 = Ostriche()
    obj2.walk()

    # Will raise an exception as ostriches can't fly
    obj2.fly()
except Exception as e:
    # Print the exception message
    print(e)
```

В нашем примере есть класс Bird с двумя абстрактными методами: fly и walk. Метод fly описывает поведение птиц во время

полета, а метод walk описывает поведение птиц при ходьбе. У нас также есть два подкласса класса Bird: Ostriche и Eagle.
Класс Ostriche обеспечивает реализацию обоих методов, но метод fly вызывает исключение, указывающее на то, что страусы *(Ostriche*) не могут летать. В противоположность этому, класс Eagle предлагает простые реализации для обоих методов, выводя на экран сообщения, которые показывают, что орел *(Eagle*) летает и ходит. В последней части кода мы создаем объекты для каждого подкласса и вызываем их методы. Когда мы пытаемся заставить страуса летать, возникает исключение, которое перехватывается, предоставляя пользователю обратную связь. Программный выбор заставить нелетающую птицу, такую как страус, реализовать метод fly, что можно рассматривать как ограничение этой настройки.

В нашем примере класс Bird требует, чтобы любой подкласс имел методы fly и walk. Это заставляет класс Ostriche, подкласс Bird, реализовать метод fly, хотя на самом деле он не применяется к страусам. Такая программа заставляет класс Ostriche зависеть от интерфейса(fly), который он не использует, что является прямым нарушением принципа разделения интерфейсов. Согласно ISP ни один класс не должен быть вынужден реализовывать интерфейсы, которые он не использует. Таким образом, наш код только выиграет от разделения интерфейса Bird, чтобы более адекватно соответствовать различным типам поведения птиц.

Не спешите и подумайте, как улучшить наш код, чтобы он не нарушал требования провайдера.

Вот наше решение для буквы I. Код находится в файле *I principle right way.py*, который прилагается к этому уроку:

```
from abc import ABC, abstractmethod

# Define a Walkable interface with a walk method
class Walkable(ABC):
    @abstractmethod
    def walk(self):
        pass

# Define a Flyable interface with a fly method
class Flyable(ABC):
    @abstractmethod
    def fly(self):
        pass

# Ostriche class implementing the Walkable interface
class Ostriche(Walkable):
    def walk(self):
        print("Ostriche is walking")

# Eagle class implementing both Walkable and Flyable interfaces
class Eagle(Walkable, Flyable):
    def fly(self):
        print("Eagle is flying")
    def walk(self):
```

```
        print("Eagle is walking")

# Creating instances and calling methods
try:
    obj = Eagle()
    # Expected: "Eagle is flying"
    obj.fly()

    # Expected: "Eagle is walking"
    obj.walk()
    obj2 = Ostriche()

    # Expected: "Ostriche is walking"
    obj2.walk()

except Exception as e:
        # Print any exceptions that might occur
        print(e)
```

В нашем примере есть два отдельных интерфейса: Walkable и Flyable, каждый из которых предоставляет контракт на поведение ходьбы и полета соответственно. Класс Ostriche реализует интерфейс Walkable, а следовательно, обязан предоставить метод walk. Соответственно, когда создается экземпляр объекта Ostriche и вызывается его метод walk, он показывает, что страус ходит. Класс Eagle реализует оба интерфейса: Walkable и Flyable. Это означает, что он должен обеспечивать реализацию как методов walk, так и методов fly. Как видим, когда создается объект Eagle и вызываются его методы, он показывает, что орел и ходит, и летает. Код использует разделение интерфейсов для точного представления возможностей различных птиц, не заставляя ни один класс птиц реализовывать поведение, которое им не свойственно.

В нашем примере вместо того, чтобы навязывать монолитный интерфейс для всех классов птиц, код вводит отдельные интерфейсы, Walkable и Flyable. Класс Ostriche, который не летает, реализует только интерфейс Walkable, соблюдая только тот контракт, который имеет отношение к его поведению. Класс Eagle, который может и ходить, и летать, реализует оба интерфейса. Таким образом, код гарантирует, что классы должны реализовывать только те методы, которые непосредственно связаны с их поведением, без лишних обязательств. Такой модульный подход обеспечивает соблюдение принципа разделения интерфейсов, поскольку ни один класс не обременен реализацией ненужных ему интерфейсов. Дизайн предлагает гибкость и более четкое представление возможностей каждой птицы, что полностью соответствует сути ISP.

5.2. Преимущества применения принципа разделения интерфейсов

- **Уменьшение сложности.** Когда интерфейсы разделены и более специфичны, они становятся проще и понятней. Это означает, что

разработчикам не нужно разбираться со сторонними методами, которые не имеют отношения к их непосредственной задаче.
- **Повышенная гибкость.** Более детализированные интерфейсы позволяют системам быть гибче. Классы могут выбирать, какие интерфейсы они хотят реализовать, что обеспечивает более адаптированный и гибкий дизайн.
- **Более высокая сплоченность.** Поскольку интерфейс фокусируется только на определенном наборе поведения, классы, которые его реализуют, более сплочены, а это означает, что они несут единую, четкую ответственность.
- **Более легкая эволюция.** В процессе развития и роста вашего программного обеспечения, ISP гарантирует, что изменения в одной части системы не приведут к возникновению проблем в других, не связанных с ней частях. Это связано с тем, что каждый интерфейс имеет узкую сферу применения, что ограничивает широту его влияния.
- **Улучшенная читабельность.** Благодаря хорошо разделенным интерфейсам разработчики быстрее понимают архитектуру системы, поскольку каждый интерфейс и класс, который его реализует, имеет четкую и ограниченную роль.

5.3. Как применить принцип разделения интерфейсов?

Не углубляясь в код, важно знать самые распространенные приемы в Python для поддержки ISP:

- **Несколько маленьких интерфейсов.** Вместо того, чтобы создавать большой, всеобъемлющий интерфейс, разбейте его на меньшие, более сфокусированные интерфейсы. Каждый из этих интерфейсов должен представлять определенную возможность или набор связанных действий.
- **Классы-миксины.** В Python миксины — это способ обеспечить многократное использование частей функционала. Их можно рассматривать как тип интерфейса с реализованными методами. Используя миксины, вы можете создавать классы, смешивая необходимую им функциональность, не отягощая их несущественными методами.
- **Абстрактные базовые классы.** Абстрактные базовые классы можно использовать для определения интерфейсных контрактов. С помощью декоратора @abstractmethod вы можете гарантировать, что производные классы реализуют конкретные методы.
- **Преимущество композиции над наследованием.** Хотя наследование может помочь с повторным использованием кода, иногда оно может заставить классы наследовать методы, которые им не нужны. Отдавая предпочтение композиции, вы будете уверены, что класс получит только то поведение (методы), которое ему нужно, путем создания меньших, четко определенных объектов или интерфейсов.

Принцип разделения интерфейсов подчеркивает важность того, чтобы класс был вынужден реализовывать только те методы, которые ему действительно нужны. Следуя ISP, вы создаете более чистые, модульные и легкие в обслуживании программные архитектуры.

6. D-принцип инверсии зависимостей

Принцип инверсии зависимостей *(Dependency Inversion Principle, DIP*) соответствует букве «D» в аббревиатуре SOLID. Принцип DIP способствует разделению программных модулей, поскольку высокоуровневые модули не должны зависеть от низкоуровневых модулей. Вместо этого оба должны зависеть от абстракций (то есть интерфейсов или абстрактных классов). Идея состоит в том, чтобы минимизировать прямые зависимости между конкретными классами, что облегчает модификацию, масштабирование и понимание кодовой базы.

6.1. Пример нарушения DIP из реальной жизни

Представьте, что вы управляете предприятием, которое производит электронику. Ваша производственная линия имеет несколько машин, каждая из которых выполняет разные функции: одна машина собирает детали, другая их красит, а третья тестирует готовую продукцию. Для управления этими машинами у вас есть центральная система управления, которая напрямую взаимодействует с каждой из них.

Вы решили обновить один из станков на более новую модель от другого производителя. Вскоре вы понимаете, что ваша центральная система управления была четко запрограммирована на работу с особенностями старого станка. Теперь вам придется вносить существенные изменения в систему управления, чтобы приспособить ее к этой новой машине, что, в свою очередь, может нарушить работу всей линии.

6.2. В чем было нарушение DIP?

В этом примере центральная система управления (модуль высокого уровня) напрямую зависит от модулей низкого уровня, которыми являются отдельные машины. Это нарушает принцип инверсии зависимости, поскольку изменения в модулях низкого уровня (например, замена одной машины на другую) вызывают изменения в модуле высокого уровня. Если бы система управления была разработана для взаимодействия с абстракцией (например, общим «машинным интерфейсом», который должна реализовывать любая машина), замена одной машины на другую не потребовала бы изменений в самой системе управления. Высокоуровневые и низкоуровневые модули будут зависеть от этой абстракции, что делает систему более модульной и более легкой в управлении.

6.3. Пример нарушения DIP в коде

Давайте начнем с неправильного пути и попробуем найти проблему. Вот наш код для буквы D. Код находится в файле *D wrong way.py*, который прилагается к этому уроку:

```
# Class responsible for reading and writing data to a text file
class TextFileOperations:
    def __init__(self,path):
        self.path = path

    # Reads the data from the file
    def read_text_data(self):
        with open(self.path, 'r') as file:
            data = file.read()
        return data

    # Writes data to the file
    def write_text_data(self,data):
        with open(self.path,'w') as file:
            file.write(data)

# Class for operations on the text data
class TextOperations:
    def __init__(self,text_source):
        self.text_source = text_source
        self.data = self.text_source.read_text_data()

    # Searches for a word in the data
    def search_for_word(self, word):
        return word inself.data

    # Counts the occurrences of a word in the data
    def count_occurences(self, word):
        return self.data.count(word)

file = TextFileOperations("D:\\data.txt")

obj = TextOperations(file)
print(f"{obj.search_for_word('more')}")
print(f"{obj.count_occurences('be')}")
```

Код состоит из двух основных
классов: TextFileOperations и TextOperations.
Класс TextFileOperations отвечает за работу с текстовыми файлами, предлагая методы для чтения и записи данных в указанный путь к файлу. Класс TextOperations выполняет операции с текстом, такие как поиск слова и подсчет его вхождений в текстовых данных. Он принимает объект TextFileOperations как аргумент для инициализации своего источника данных. Также наш код демонстрирует создание экземпляров этих классов и использование их методов для поиска слова и подсчета его вхождений в текстовом файле, расположенном по адресу «*D:\\data.txt*».

Код нарушает принцип инверсии зависимостей, поскольку высокоуровневый модуль TextOperations непосредственно зависит от низкоуровневого модуля TextFileOperations. Такая тесная связь означает, что если вы захотите изменить источник получения текстовых данных (например, из базы данных или API вместо текстового файла), вам придется вносить изменения в сам класс TextOperations. Такой дизайн делает систему менее гибкой и более трудной для расширения или модификации. В идеале, и TextOperations, и TextFileOperations должны зависеть от абстракции (например, интерфейса или абстрактного класса), чтобы разъединить их и сделать систему более модульной.

Не спешите и подумайте, как улучшить наш код, чтобы он не нарушал DIP.

Вот наше решение для буквы D. Код находится в файле *D right way.py*, котрый прилагается к этому уроку:

```
from abc import ABC, abstractmethod

# Abstract base class for a data source
class DataSource(ABC):
    def __init__(self,path):
        self.path = path

    # Define interface for reading and writing data
    @abstractmethod
    def read_data(self):
        pass
    @abstractmethod
    def write_data(self):
        pass

# DataSource implementation for text files
class TextDataSource(DataSource):
    def read_data(self):
        with open(self.path, 'r') as file:
            data = file.read()
        return data
    def write_data(self,data):
        with open(self.path,'w') as file:
            file.write(data)

# DataSource implementation for databases
class DbDataSource(DataSource):
    def read_data(self):
        return "data from database"
    def write_data(self,data):
        print(f"write {data} to database")

# Perform text operations using a DataSource
class TextOperations:
    def __init__(self,data_source):
        self.data_source = data_source
        self.data = self.data_source.read_data()

    # Returns True if the word is in the data
```

```
    def search_for_word(self, word):
        return word inself.data

    # Returns the number of occurrences of a word in the data
    def count_occurrences(self, word):
```

В коде определен абстрактный базовый класс DataSource с методами read_data и write_data, который служит общим интерфейсом для источников данных. Два конкретных класса, TextDataSource и DbDataSource, реализуют этот интерфейс для работы с текстовыми файлами и базами данных соответственно. Класс TextOperations выполняет текстовые операции, такие как поиск слов и подсчет вхождений, но теперь работает с объектом DataSource. Это позволяет ему беспрепятственно работать с различными источниками данных, что демонстрируется путем инициализации его экземплярами TextDataSource и DbDataSource.

Этот код придерживается принципа инверсии зависимостей, поскольку высокоуровневый модуль (TextOperations) и низкоуровневые модули (TextDataSource и DbDataSource) зависят от абстракции DataSource. Таким образом, система разделяет модули, что облегчает расширение или модификацию источников данных, не влияя на класс TextOperations. Если вы хотите добавить новый источник данных, вам просто нужно создать новый класс, который реализует интерфейс DataSource, при этом система остается гибкой и поддерживаемой.

6.4. Преимущества применения принципа инверсии зависимостей

Разделение. Соблюдение DIP обеспечивает разделение высокоуровневых и низкоуровневых модулей в коде. Это облегчает внесение изменений в один уровень без влияния на другой и повышает поддерживаемость и гибкость кода.

Легкое расширение. Когда модули зависят от абстракций, а не от конкретных реализаций, добавлять новые функции или изменять существующие становится проще. Вы можете внедрять новые низкоуровневые модули, которые соответствуют существующей абстракции, не изменяя высокоуровневые модули.

Возможность тестирования. Инверсия зависимостей облегчает тестирование системы. Поскольку высокоуровневые модули не привязаны к низкоуровневым, вы можете легко заменить реальные реализации на имитационные объекты для тестирования, улучшая процесс обеспечения качества.

Повторное использование кода. DIP способствует созданию многоразовых абстракций. Это означает, что вы можете повторно использовать высокоуровневые модули в различных проектах и

контекстах, если новые низкоуровневые модули придерживаются установленных абстракций.

Улучшенная читабельность. Благодаря акценту на абстракциях, код естественно становится более понятным, что позволяет новым разработчикам быстрее понять архитектуру системы. Это облегчает процесс адаптации и ускоряет циклы разработки.

6.5. Как применить принцип инверсии зависимостей?

Следуя идее об абстрактности кода, в нашем распоряжении есть несколько способов поддержки принципа DIP:

- **Абстрагирование.** Создание интерфейсов или абстрактных базовых классов является основным механизмом применения DIP. Высокоуровневые и низкоуровневые модули должны зависеть от этих абстракций, а не от конкретных реализаций.
- **Инъекция зависимостей.** Предоставление высокоуровневому модулю его зависимостей вместо того, чтобы позволить ему создавать их. Инъекция зависимостей может быть осуществлена через конструкторы, методы или свойства, таким образом экстернализируя задачу создания и связывания объектов.
- **Агрегация.** Подобно композиции в OCP, агрегация в DIP предусматривает инкапсуляцию связанных функциональных возможностей в пределах одного высокоуровневого модуля. Эти функциональные возможности должны придерживаться общего интерфейса, что позволяет модулю высокого уровня не знать об их конкретной реализации.
- **Паттерн адаптера.** При работе со старым кодом или сторонними библиотеками, которые не придерживаются определенных вами абстракций, можно использовать паттерн адаптера. Этот паттерн позволяет создать новый интерфейс, который отображается на существующий, гарантируя, что ваши высокоуровневые модули взаимодействуют только с определенными вами абстракциями.

Таким образом, принцип инверсии зависимостей предполагает, что как высокоуровневые, так и низкоуровневые модули должны зависеть от абстракций. Следуя этому принципу, программные системы становятся более модульными, их легче тестировать, проще поддерживать и расширять.

7. Подведение итогов по принципам SOLID

Принципы SOLID (единой ответственности, открытости/закрытости, замещения Лисков, разделения интерфейсов и инверсии зависимостей) являются фундаментальными установками, цель которых — улучшить

дизайн, структуру и понимание программной системы. Эти принципы служат основой для создания программного обеспечения, которым легко управлять, который просто поддерживать и расширять. Они ведут вас к написанию модульного кода, в котором компоненты имеют единую, четко определенную роль. Они обеспечивают гибкость, позволяя системам развиваться и адаптироваться со временем без дорогостоящего рефакторинга. Они подчеркивают важность создания кода, который можно легко протестировать, обеспечивая тем самым надежность приложений. Кроме того, они продвигают идею разделения, что облегчает замену компонентов и повышает возможность повторного использования. По сути, освоение принципов SOLID является шагом к освоению самой программной инженерии, поскольку они дают разработчикам необходимые концептуальные инструменты для преодоления сложностей, присущих созданию масштабируемых и надежных программных решений.