## **1. Single Responsibility Principle**

## **(Принцип единственной обязанности)**

Принцип единственной обязанности требует того, чтобы один класс выполнял только одну работу. Таким образом, если у класса есть более одной работы, он становится зависимым. Изменение поведения одной работы класса приводит к изменению в другой.

# Below is Given a class which has two responsibilities

class User:

def \_\_init\_\_(self, name: str):

self.name = name

def get\_name(self) -> str:

pass

def save(self, user: User):

pass

Мы имеем класс User, который ответственен за две работы — свойства пользователя и управление базой данных. Если в приложении будет изменен функционал управления базой данных для пользователя, тогда классы использующие свойства класса User тоже придется доработать и перекомпилировать, чтобы компенсировать новые изменения. Это как домино эффект, уроните одну кость, и она уронит все за ней следом.

Мы же просто разделим класс. Мы создадим ещё один класс, который возьмет на себя одну ответственность — управление базой данных пользователя.

class User:

def \_\_init\_\_(self, name: str):

self.name = name

def get\_name(self):

pass

class UserDB:

def get\_user(self, id) -> User:

pass

def save(self, user: User):

pass

Распространённым решением этой проблемы является применение шаблона проектирования Фасад. User класс был бы фасадом для управления базой данных пользователя и управления свойствами пользователя.

## **2. Open-Closed Principle**

## **(Принцип открытости/закрытости)**

Программные сущности (классы, модули, функции) должно быть открыты для расширения, но не модификации.

Давайте представим, что у вас есть магазин, и вы даете скидку в 20% для ваших любимых покупателей используя класс Discount. Если бы вы решаете удвоить 20-ти процентную скидку для VIP клиентов, вы могли бы изменить класс следующим образом:

class Discount:

def \_\_init\_\_(self, customer, price):

self.customer = customer

self.price = price

def give\_discount(self):

if self.customer == 'fav':

return self.price \* 0.2

if self.customer == 'vip':

return self.price \* 0.4

Но нет, это нарушает OCP. OCP запрещает это. Например, если мы хотим дать новую скидку для другого типа покупателей, то это требует добавления новой логики. Чтобы следовать OCP, мы добавим новый класс, который будет расширять Discount. И в этом новом классе реализуем требуемую логику:

class Discount:

def \_\_init\_\_(self, customer, price):

self.customer = customer

self.price = price

def get\_discount(self):

return self.price \* 0.2

class VIPDiscount(Discount):

def get\_discount(self):

return super().get\_discount() \* 2

Если вы решите дать скидку супер VIP пользователям, то это будет выглядеть так:

class SuperVIPDiscount(VIPDiscount):

def get\_discount(self):

return super().get\_discount() \* 2

Расширяйте, но не модифицируйте.

## 

## 

## **3. Liskov Substitution Principle**

## **(Принцип подстановки Лисков)**

Главная идея, стоящая за Liskov Substitution Principle в том, что для любого класса клиент должен иметь возможность использовать любой подкласс базового класса, не замечая разницы между ними, и следовательно, без каких-либо изменений поведения программы при выполнении. Это означает, что клиент полностью изолирован и не подозревает об изменениях в иерархии классов.

*Более формально: Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T.*

Проще говоря, это значит, что подкласс, дочерний класс должны соответствовать их родительскому классу или супер классу.

class User():

def \_\_init\_\_(self, color, board):

create\_pieces()

self.color = color

self.board = board

def move(self, piece:Piece, position:int):

piece.move(position)

chessmate\_check()

board = ChessBoard()

user\_white = User("white", board)

user\_black = User("black", board)

pieces = user\_white.pieces

horse = helper.getHorse(user\_white, 1)

user.move(horse)

LSP это основа хорошего объектно-ориентированного проектирования программного обеспечения, потому что он следует одному из базовых принципов ООП — полиморфизму. Речь о том, чтобы создавать правильные иерархии, такие, что классы, производные от базового являлись полиморфными для их родителя по отношению к методам их интерфейсов. Ещё интересно отметить, как этот принцип относится к примеру предыдущего принципа. Если мы пытаемся расширить класс новым несовместимым классом, то все сломается. Взаимодействие с клиентом будет нарушено, и как результат, такое расширение будет невозможно (или, для того чтобы сделать это возможным, нам пришлось бы нарушить другой принцип и модифицировать код клиента, который должен быть закрыт для модификации, такое крайне нежелательно и неприемлемо).

Тщательное обдумывание новых классов в соответствии с LSP помогает нам расширять иерархию классов правильно. Также, LSP способствует OCP.

# **4. Interface Segregation Principle**

# **(Принцип разделения интерфейсов)**

Создавайте тонкие интерфейсы, которые ориентированы на клиента. Клиенты не должны зависеть от интерфейсов, которые они не используют. Этот принцип устраняет недостатки реализации больших интерфейсов.

Чтобы полностью проиллюстрировать это, мы возьмем классический пример, потому что он очень показательный и легок для понимания. Классический пример:

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

Еще один приятный трюк заключается в том, что в нашей бизнес-логике отдельный класс может реализовать несколько интерфейсов, если необходимо. Таким образом, мы может предоставить единую реализацию для всех общих методов между интерфейсами. Сегрегированные интерфейсы заставляют нас больше думать о нашем коде с точки зрения клиента, что приведет нас к меньшей зависимости и более легкому тестированию. Таким образом, мы не только сделали наш код лучше для клиента, но также это облегчило нам понимание, тестирование и реализацию кода для нас самих.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## **5. Dependecy Inversion Principle**

## **(Принцип инверсии зависимостей)**

Зависимость должна быть от абстракций, а не от конкретики. Модули верхних уровней не должны зависеть от модулей нижних уровней. Классы и верхних, и нижних уровней должны зависеть от одних и тех же абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Наступает момент в разработке, когда наше приложение в основном состоит из модулей. Когда такое происходит, нам необходимо улучшать код используя внедрение зависимостей. Функционирование компонентов высокого уровня зависит от компонентов низкого уровня. Для создания определенного поведения вы можете использовать наследование или интерфейсы.

class AuthenticationForUser():

def \_\_init\_\_(self, connector:Connector):

self.connection = connector.connect()

def authenticate(self, credentials):

pass

def is\_authenticated(self):

pass

def last\_login(self):

pass

class AnonymousAuth(AuthenticationForUser):

pass

class GithubAuth(AuthenticationForUser):

def last\_login(self):

pass

class FacebookAuth(AuthenticationForUser):

pass

class Permissions()

def \_\_init\_\_(self, auth: AuthenticationForUser)

self.auth = auth

def has\_permissions():

pass

class IsLoggedInPermissions (Permissions):

def last\_login():

return auth.last\_log