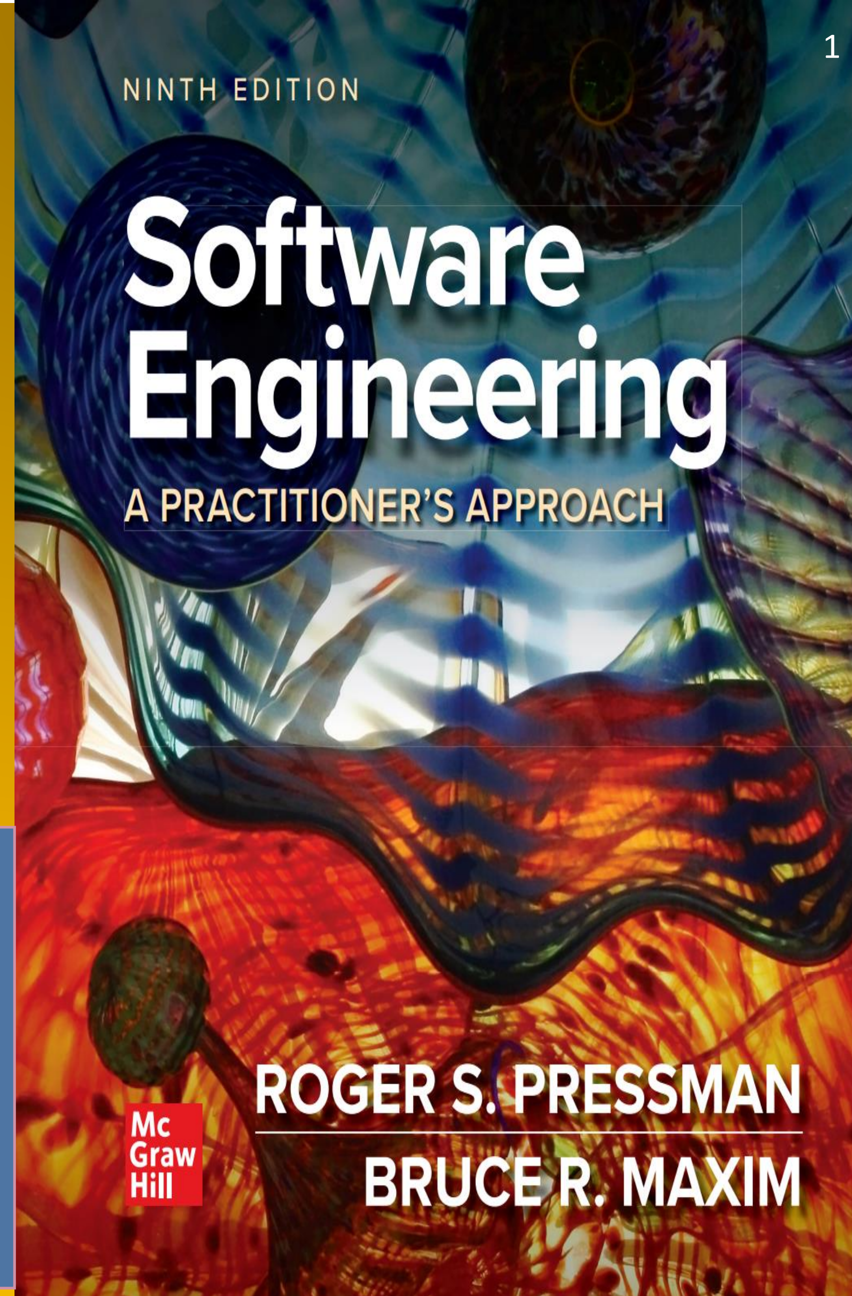
مهندسی نرم افزار ۲

Prof. A. Taghinezhad
University of Tabriz



Website: ataghinezhad.github.io, Email:a0taghinezhad@gmail.com

برنامه نویسی بر اصول Solid

اصول SOLID برای کدنویسی

- اصول SOLID پنج اصل طراحی هستند که بهترین شیوه‌های برنامه‌نویسی شیءگرا را ترویج می‌کنند:
- **اصل مسئولیت واحد :(SRP)**
- Single Responsibility Principle
- **اصل باز/بسته :(OCP)**
- Open/Closed Principle
- **اصل جانشینی لیسکوف :(LSP)**
- Liskov Substitution Principle
- **اصل جداسازی رابط :(ISP)**
- Interface Segregation Principle
- **اصل وارونگی وابستگی :(DIP)**
- Dependency Inversion Principle

اصول SOLID برای کدنویسی

- اصول SOLID پنج اصل طراحی هستند که بهترین شیوه‌های برنامه‌نویسی شیءگرا را ترویج می‌کنند:

1. **اصل مسئولیت واحد (SRP):** یک **کلاس باید فقط یک دلیل برای تغییر** داشته باشد. این امر باعث می‌شود **کلاس‌ها متمرکز** باقی بمانند و احتمال بروز عوارض جانبی ناخواسته هنگام ایجاد تغییرات را کاهش می‌دهد.

- **مثال:** فرض کنید سناریویی دارید که در آن یک کلاس به نام «کارمند» Employeeوجود دارد که هم اطلاعات کارمندان و هم محاسبات حقوق و دستمزد آن‌ها را مدیریت می‌کند.

بدون SRP

```python
class Employee:
    def __init__(self, name, id, salary):
        self.name = name
        self.id = id
        self.salary = salary

    def calculate_salary(self):
        # Payroll calculation logic
        pass

    # Other methods related to employee information
    def update_personal_info(self, new_info):
        pass

    def get_department(self):
        pass
```

- کلاس «کارمند» مسئولیت‌های چندگانه‌ای دارد. این کلاس هم اطلاعات کارمندان (مانند نام، شناسه، بخش) و هم محاسبات حقوق و دستمزد آن‌ها را مدیریت می‌کند.

- اگر در منطق محاسبه حقوق و دستمزد تغییراتی ایجاد شود، ممکن است بر مدیریت اطلاعات کارمندان تأثیر بگذارد و اصل SRPنقض شود.

با SRP

- ما مسئولیت‌های مدیریت اطلاعات کارمندان و انجام محاسبات حقوق و دستمزد را به دو کلاس جداگانه تقسیم کرده‌ایم: «کارمند» و «محاسبه‌گر حقوق و دستمزد»

- کلاس «کارمند» مسئول ذخیره اطلاعات کارمندان است، در حالی که کلاس «محاسبه‌گر حقوق و دستمزد» مسئول محاسبه حقوق بر اساس داده‌های کارمندان است.

```
class Employee:
    def __init__(self, name, id, department):
        self.name = name
        self.id = id
        self.department = department

class PayrollCalculator:
    @staticmethod
    def calculate_salary(employee):
        # Payroll calculation logic
        pass
```

اصول SOLID برای کدنویسی

۲. اصل باز/بسته (OCP): موجودیت‌های نرم‌افزاری (**کلاس‌ها**، ماژول‌ها، توابع) **باید** برای **توسعه** باز باشند، اما برای **اصلاح بسته** باشند. این امر باعث **تشویق قابلیت استفاده مجدد** کد می‌شود و امکان افزودن قابلیت‌های جدید بدون تغییر کد موجود را فراهم می‌سازد.

- **مثال:** فرض کنید سیستمی دارید که انواع مختلف سفارشات را پردازش می‌کند، مانند سفارشات آنلاین، سفارشات تلفنی و سفارشات حضوری

بدون OCP

```python
class OrderProcessor:
    def process_order(self, order):
        if order.type == 'online':
            self.process_online_order(order)
        elif order.type == 'phone':
            self.process_phone_order(order)
        elif order.type == 'in_store':
            self.process_in_store_order(order)

    def process_online_order(self, order):
        print("Processing online order:", order)

    def process_phone_order(self, order):
        print("Processing phone order:", order)

    def process_in_store_order(self, order):
        print("Processing in-store order:", order)
```

- کلاس OrderProcessor مسئول پردازش انواع مختلف سفارشات است.
- اگر نوع جدیدی از سفارش‌ها مانند سفارش‌های ایمیلی معرفی شود، باید کلاس OrderProcessorرا تغییر دهیم که OCPرا نقض می‌کند.

با OCP

```python
from abc import ABC, abstractmethod

class OrderProcessor(ABC):
    @abstractmethod
    def process_order(self, order):
        pass

class OnlineOrderProcessor(OrderProcessor):
    def process_order(self, order):
        print("Processing online order:", order)

class PhoneOrderProcessor(OrderProcessor):
    def process_order(self, order):
        print("Processing phone order:", order)

class InStoreOrderProcessor(OrderProcessor):
    def process_order(self, order):
        print("Processing in-store order:", order)
```

حال اگر نیاز به پردازش نوع جدیدی از سفارش، مانند سفارشات «ایمیل» باشد، می‌توانیم به سادگی یک زیر کلاس جدید از OrderProcessorایجاد کنیم بدون اینکه کد موجود را تغییر دهیم.

اصول SOLID برای کدنویسی

۳- اصل جانشینی لیسکوف (LSP): این اصل بیان می‌کند که اشیاء یک زیرکلاس باید بدون اینکه بر صحت برنامه تأثیر بگذارد، قابل جایگزینی با اشیاء سوپرکلاس آن باشند.

- **مثال:** فرض کنید سناریویی دارید که در آن یک سلسله مراتب کلاسی برای اشکال دارید، از جمله یک کلاس پایه به نام «شکل» Shapeو زیرکلاس‌هایی مانند «مربع» و «دایره»

```python
class Shape:
    def area(self):
        pass

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length * self.side_length

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

def calculate_total_area(shapes):
    total_area = 0
    for shape in shapes:
        total_area += shape.area()
    return total_area

shapes = [Square(5), Circle(3)]
print("Total area:", calculate_total_area(shapes))
```

بدون اصل LSP

، کلاس‌های «مربع» و «دایره» زیرکلاس‌هایی از «شکل» هستند و هر دو متد «مساحت» areaرا برای محاسبه مساحت‌های مربوط به خود بازنویسی می‌کنند.

اگر سعی کنیم از تابع «محاسبه مساحت کل» calculate_total_areaبا لیستی حاوی اشکال از انواع مختلف استفاده کنیم، ممکن است نتیجه درستی به دست نیاید،

زیرا هر زیرکلاس مساحت خود را به روش متفاوتی محاسبه می‌کند.

با اصل LSP

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Square(Shape):
    def __init__(self, side_length):
        self.side_length = side_length

    def area(self):
        return self.side_length * self.side_length

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

def calculate_total_area(shapes):
    total_area = 0
    for shape in shapes:
        total_area += shape.area()
    return total_area

shapes = [Square(5), Circle(3)]
print("Total area:", calculate_total_area(shapes))
```

- در این مثال که از اصل LSPتبعیت می‌کند، کلاس «شکل» را انتزاعی abstractکرده‌ایم و متد «مساحت» را نیز به صورت انتزاعی تعریف کرده‌ایم.
- این کار باعث می‌شود که همه زیرکلاس‌ها مجبور شوند متد «مساحت» را مطابق با اشکال خاص خود پیاده‌سازی کنند نه از متد مساحت والد استفاده کنند..
- حال، هنگامی که از تابع «محاسبه مساحت کل» با لیستی حاوی اشکال از انواع مختلف استفاده می‌کنیم، هر زیرکلاس پیاده‌سازی خاص خود از متد «مساحت» را ارائه می‌دهد و بدین ترتیب رفتار صحیح را تضمین می‌کند.

اصول SOLID برای کدنویسی

۴- اصل جداسازی رابط (ISP): نباید افراد را مجبور کرد به متدهایی که استفاده نمی‌کنند وابسته باشند. این به معنای ایجاد رابط‌های کوچک‌تر و خاص‌تر است که مشتریان می‌توانند به جای یک رابط بزرگ و عمومی به آن‌ها وابسته باشند.

- **مثال:** تصور کنید سناریویی را طراحی می‌کنید که در آن رابط‌هایی برای یک سیستم پردازش سند تعریف می‌کنید که شامل قابلیت‌هایی برای چاپ و اسکن اسناد است.

Interface Segregation Principle بدون اصل جداسازی رابط

```python
class Machine:
    def print_document(self, document):
        pass

    def scan_document(self):
        pass

class AllInOnePrinter(Machine):
    def print_document(self, document):
        print("Printing document:", document)

    def scan_document(self):
        print("Scanning document")
```

- تصور کنید سناریویی را طراحی می‌کنید که در آن رابط‌هایی برای یک سیستم پردازش سند تعریف می‌کنید که شامل قابلیت‌هایی برای چاپ و اسکن اسناد است.
- رابط «دستگاه» شامل هر دو متد «چاپ سند» و «اسکن سند» است.
- ممکن است همه «کاربران» به هر دو قابلیت نیاز نداشته باشند.

با اصل جداسازی رابط

```python
class Printer:
    def print_document(self, document):
        pass

class Scanner:
    def scan_document(self):
        pass


class AllInOneMachine(Printer, Scanner):
    def print_document(self, document):
        print("Printing document:", document)

    def scan_document(self):
        print("Scanning document")
```

- اینجا، رابط «دستگاه» به دو رابط کوچک‌تر تقسیم کرده‌ایم: «چاپگر» و «اسکنر» حال، «کاربران» می‌توانند بسته به نیازهای خاص خود، به صورت جداگانه به رابط «چاپگر» یا «اسکنر» وابسته باشند.

- کلاس «دستگاه همه کاره» هر دو رابط را برای ارائه عملکرد ترکیبی پیاده‌سازی می‌کند، اما «کاربران» می‌توانند فقط از بخش‌هایی که نیاز دارند استفاده کنند.

اصول SOLID برای کدنویسی

- **۵-اصل وارونگی وابستگی :**
  - ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند.
  - هر دو سطح باید به انتزاعیات Abstractions وابسته باشند.
  - انتزاعیات نباید به جزئیات وابسته باشند.
  - جزئیات باید به انتزاعیات وابسته باشند.
- **توضیح:**
- اصل وابستگی معکوس بیان می‌کند که ماژول‌های یک سیستم باید به جای وابستگی به پیاده‌سازی‌های خاص Concrete Implementations، به انتزاعیات (مانند رابطها یا کلاس‌های انتزاعی) وابسته باشند. این کار انعطاف‌پذیری و جداسازی Decouplingرا به ارمغان می‌آورد، زیرا به ماژول‌های سطح بالا اجازه می‌دهد تا تحت تاثیر تغییرات در ماژول‌های سطح پایین قرار نگیرند.

اصول SOLID برای کدنویسی اصل وارونگی وابستگی

- **مثال:**
- فرض کنید سناریوی ساده‌ای دارید که در آن یک ماژول مدیریت کاربر UserManagement برای انجام عملیات CRUD (ایجاد، خواندن، به‌روزرسانی، حذف) روی داده‌های کاربر با یک ماژول پایگاه داده Database تعامل دارد.
- در رویکرد سنتی، ماژول مدیریت کاربر مستقیماً به کلاس پایگاه داده خاص (مثلاً MySQLDatabase) وابسته است. این وابستگی مستقیم باعث ایجاد مشکلات زیر می‌شود:
- **عدم انعطاف‌پذیری:** اگر بعداً بخواهید از یک پایگاه داده متفاوت مانند PostgreSQL استفاده کنید، باید کد ماژول مدیریت کاربر را برای کار با پایگاه داده جدید تغییر دهید.
- **اتصال کامل Tight Coupling:** هر گونه تغییر در کلاس پایگاه داده خاص بر ماژول مدیریت کاربر تأثیر می‌گذارد و آزمایش و نگهداری کد را پیچیده‌تر می‌کند.
- راه کار؟

```python
class UserManagement:
    def __init__(self):
        self.db = Database()

    def add_user(self, user):
        self.db.insert(user)

    def get_user(self, user_id):
        return self.db.select(user_id)
```

اصول SOLID برای کدنویسی اصل وارونگی وابستگی

- **راه حل:**
- با استفاده از اصل وابستگی معکوس، می‌توانیم این مشکلات را برطرف کنیم.
- یک رابط کاربری به نام "دیتابیس " Databaseتعریف می‌کنیم که عملکردهای CRUDرا مشخص می‌کند.
- کلاس‌های خاصی مانند MySQLDatabaseو PostgreSQLDatabaseایجاد می‌کنیم که رابط "دیتابیس" را پیاده‌سازی می‌کنند.
- ماژول مدیریت کاربر به رابط "دیتابیس" وابسته می‌شود و می‌تواند با هر پیاده‌سازی مشخصی از این رابط کار کند.
- با این رویکرد، ماژول مدیریت کاربر از جزئیات پیاده‌سازی پایگاه داده خاص جدا شده است.
  - این کار باعث انعطاف‌پذیری بیشتر کد می‌شود، زیرا می‌توانیم به راحتی از پایگاه‌های داده مختلف بدون تغییر کد ماژول مدیریت کاربر استفاده کنیم.
  - ، جداسازی Decouplingرا بهبود می‌بخشد و آزمایش و نگهداری کد را آسان‌تر می‌کند.

```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def select(self, key):
        pass

    # Other abstract methods for update and delete operations

class UserManagement:
    def __init__(self, db):
        self.db = db

    def add_user(self, user):
        self.db.insert(user)

    def get_user(self, user_id):
        return self.db.select(user_id)
```

پایان