**Optimizing Python Code with SOLID Principles**

[**software design**](https://arjancodes.com/blog/tags/software-design/)

When designing software, it’s important to think about scalability and maintainability as the application grows, so in this blog post I am going to show you how you can improve your code using the SOLID design principles.

**What are the SOLID principles?**

**Single Responsibility Principle (SRP)**

The Single Responsibility Principle advocates for a class or module to have only one reason to change. In simpler terms, it should do one thing and do it well. By adhering to SRP, your code becomes more modular, making it easier to understand and maintain.

**Open-Closed Principle (OCP)**

The Open-Closed Principle states that software entities should be open for extension but closed for modification. This means that you should be able to extend a class’s behavior without modifying it.

**Liskov Substitution Principle (LSP)**

The Liskov Substitution Principle states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program. In other words, a subclass should be able to replace its parent class without breaking the code.

**Interface Segregation Principle (ISP)**

The Interface Segregation Principle states that clients should not be forced to depend on methods they do not use. This means that you should not have to implement methods that you do not need.

**Dependency Inversion Principle (DIP)**

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules, but both should depend on abstractions. This means that you should not have to change your code when you change the implementation of a module.

**Let’s dive in!**

I am going to show you some code that violates the SOLID principles and then refactor it to adhere to them.

**Single Responsibility Principle**

class Order:

def \_\_init\_\_(*self*):

self.items = []

self.quantities = []

self.prices = []

self.status = "open"

def add\_item(*self*, *name*: str, *quantity*: int, *price*: float) -> None:

self.items.append(name)

self.quantities.append(quantity)

self.prices.append(price)

def total\_price(*self*):

total = 0

for quantity, price in zip(self.quantities, self.prices):

total += quantity \* price

return total

def pay(*self*, *payment\_type*: str, *security\_code*):

if payment\_type == "debit":

print("Processing debit payment type")

print(f"Verifying security code: {security\_code}")

self.status = "paid"

elif payment\_type == "credit":

print("Processing credit payment type")

print(f"Verifying security code: {security\_code}")

self.status = "paid"

else:

raise Exception(f"Unknown payment type: {payment\_type}")

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

print(order.total\_price())

order.pay("debit", "0372846")

This code violates the SRP because it is both responsible for managing the order and the payment. This results in our code being highly coupled and makes it harder to understand, maintain, and test.

Let’s refactor this code to adhere to the SRP.

class Order:

def \_\_init\_\_(*self*):

self.items = []

self.quantities = []

self.prices = []

self.status = "open"

def add\_item(*self*, *name*: str, *quantity*: int, *price*: float) -> None:

self.items.append(name)

self.quantities.append(quantity)

self.prices.append(price)

class PaymentProcessor:

def pay(*self*, *order*: Order, *security\_code*: str):

print("Processing payment")

print(f"Verifying security code: {security\_code}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = PaymentProcessor()

processor.pay(order, "0372846")

print(order.status)

By separating our concerns, we are able to add new methods of payment with ease without having to modify the Order class.

*Note:* This code still violates the SRP because the order is responsible for both the prices and the quantities and could be improved by separating these concerns.

**Open-Closed Principle**

We can further improve this code by adhering to the Open-Closed Principle.

If we wish to add a new payment method, we would have to make modifications to the PaymentProcessor class. This violates the Open-Closed Principle, which, as we know, states that software entities should be open for extension but closed for modification.

Let’s rework this code to adhere to the OCP.

from abc import ABC, abstractmethod

class Order:

def \_\_init\_\_(*self*):

self.items = []

self.quantities = []

self.prices = []

self.status = "open"

def add\_item(*self*, *name*: str, *quantity*: int, *price*: float) -> None:

self.items.append(name)

self.quantities.append(quantity)

self.prices.append(price)

class PaymentProcessor(ABC):

@abstractmethod

def pay(*self*, *order*, *security\_code*: str):

...

class CreditCardPaymentProcessor(PaymentProcessor):

def pay(*self*, *order*, *security\_code*: str):

print("Processing credit card payment")

print(f"Verifying security code: {security\_code}")

order.status = "paid"

class DebitCardPaymentProcessor(PaymentProcessor):

def pay(*self*, *order*, *security\_code*: str):

print("Processing debit card payment")

print(f"Verifying security code: {security\_code}")

order.status = "paid"

class PaypalPaymentProcessor(PaymentProcessor):

def pay(*self*, *order*, *security\_code*: str):

print("Processing paypal payment")

print(f"Verifying email: {security\_code}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = CreditCardPaymentProcessor()

processor.pay(order, "0372846")

print(order.status)

Now that the code adheres to the OCP, it’s closed for modification and open for extension because we can add new payment methods without modifying the PaymentProcessor class.

**Liskov Substitution Principle**

In our refactoring, we have introduced a new violation of SOLID principles. Paypal uses email addresses for verification, whereas credit and debit cards use security codes. This means we are abusing the Liskov Substitution Principle because we are using a subclass in a way that is not compatible with its parent class. This is because of the concept of *Design by Contract*, which in this context dictates classes should adhere to the “contract” set out by their interface for consistency and integrity.

Let’s refactor this code to adhere to the LSP.

from abc import ABC, abstractmethod

class Order:

def \_\_init\_\_(*self*):

self.items = []

self.quantities = []

self.prices = []

self.status = "open"

def add\_item(*self*, *name*: str, *quantity*: int, *price*: float) -> None:

self.items.append(name)

self.quantities.append(quantity)

self.prices.append(price)

class PaymentProcessor(ABC):

@abstractmethod

def pay(*self*, *order*):

...

class DebitPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str):

self.security\_code = security\_code

def pay(*self*, *order*):

print("Processing debit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str):

self.security\_code = security\_code

def pay(*self*, *order*):

print("Processing credit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

class PaypalPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *email\_address*: str):

self.email\_address = email\_address

def pay(*self*, *order*):

print("Processing paypal payment")

print(f"Verifying email: {self.email\_address}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = PaypalPaymentProcessor("hi@arjancodes.com")

processor.pay(order)

print(order.status)

Now the code adheres to the LSP because we are using the subclasses in a way that is compatible with their parent class.

**Interface Segregation Principle**

This principle states that clients should not be forced to depend on methods they do not use. This means it’s better to have interfaces that are suited to specific task rather than one general-purpose interface.

I’ll give you an example where we add the ability to send the user an SMS to authenticate their payment.

from abc import ABC, abstractmethod

class Order:

def \_\_init\_\_(*self*):

self.items = []

self.quantities = []

self.prices = []

self.status = "open"

def add\_item(*self*, *name*: str, *quantity*: int, *price*: float) -> None:

self.items.append(name)

self.quantities.append(quantity)

self.prices.append(price)

class PaymentProcessor(ABC):

@abstractmethod

def pay(*self*, *order*):

...

@abstractmethod

def auth\_sms(*self*, *order*, *code*: str):

...

class DebitPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str):

self.security\_code = security\_code

self.authenticated = False

def pay(*self*, *order*):

if not self.authenticated:

raise Exception("Not authenticated")

print("Processing debit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

def auth\_sms(*self*, *order*, *code*: str):

print("Authenticating via SMS")

self.authenticated = True

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str):

self.security\_code = security\_code

def pay(*self*, *order*):

print("Processing credit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

def auth\_sms(*self*, *order*, *code*: str):

raise Exception("Not implemented")

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = DebitPaymentProcessor("0372846")

processor.auth\_sms(order, "12345")

processor.pay(order)

print(order.status)

This code violates the ISP because the CreditPaymentProcessor class is forced to implement the auth\_sms method, even though it does not use it. This not only means we end up writing more code, but it could potentially cause bugs if we forget to implement the method.

Let’s refactor this code to adhere to the ISP.

from abc import ABC, abstractmethod

class Order:

def \_\_init\_\_(*self*):

self.items = []

self.quantities = []

self.prices = []

self.status = "open"

def add\_item(*self*, *name*: str, *quantity*: int, *price*: float) -> None:

self.items.append(name)

self.quantities.append(quantity)

self.prices.append(price)

class PaymentProcessor(ABC):

@abstractmethod

def pay(*self*, *order*: Order):

...

class SmsPaymentProcessor(PaymentProcessor):

@abstractmethod

def auth\_sms(*self*, *order*: Order, *code*: str):

...

class DebitPaymentProcessor(SmsPaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str):

self.security\_code = security\_code

self.authenticated = False

def pay(*self*, *order*):

if not self.authenticated:

raise Exception("Not authenticated")

print("Processing debit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

def auth\_sms(*self*, *order*, *code*: str):

print("Authenticating via SMS")

self.authenticated = True

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str):

self.security\_code = security\_code

def pay(*self*, *order*):

print("Processing credit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

processor = DebitPaymentProcessor("0372846")

processor.auth\_sms(order, "12345")

processor.pay(order)

print(order.status)

Now, we could make this code even better by separating the authorization logic from the payment processor.

from abc import ABC, abstractmethod

class Order:

def \_\_init\_\_(*self*):

self.items = []

self.quantities = []

self.prices = []

self.status = "open"

def add\_item(*self*, *name*: str, *quantity*: int, *price*: float) -> None:

self.items.append(name)

self.quantities.append(quantity)

self.prices.append(price)

class SMSAuthorizer:

def \_\_init\_\_(*self*):

self.authenticated = False

def verify\_code(*self*, *code*: str):

print("Verifying code")

self.authenticated = True

def is\_authenticated(*self*):

return self.authenticated

class PaymentProcessor(ABC):

@abstractmethod

def pay(*self*, *order*: Order):

...

class DebitPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str, *authorizer*: SMSAuthorizer):

self.security\_code = security\_code

self.authorizer = authorizer

def pay(*self*, *order*):

if not self.authorizer.is\_authenticated():

raise Exception("Not authenticated")

print("Processing debit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str):

self.security\_code = security\_code

def pay(*self*, *order*: Order):

print("Processing credit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

authorizer = SMSAuthorizer()

authorizer.verify\_code("12345")

processor = DebitPaymentProcessor("0372846", authorizer)

processor.pay(order)

print(order.status)

**Dependency Inversion Principle**

We could further improve this code by adhering to the Dependency Inversion Principle. The Dependency Inversion Principle states that high-level modules should not depend on low-level modules, but both should depend on abstractions. This means that you should not have to change other sections of your code when you change the implementation of a class.

In practice, this means that our payment processor shouldn’t be concerned with how its payment is validated, whether that be by SMS, a robot check, or an email.

Let’s refactor this code to adhere to the DIP.

from abc import ABC, abstractmethod

class Order:

def \_\_init\_\_(*self*):

self.items = []

self.quantities = []

self.prices = []

self.status = "open"

def add\_item(*self*, *name*: str, *quantity*: int, *price*: float) -> None:

self.items.append(name)

self.quantities.append(quantity)

self.prices.append(price)

class Authorizer(ABC):

@abstractmethod

def is\_authenticated(*self*):

...

class SMSAuthorizer(Authorizer):

def \_\_init\_\_(*self*):

self.authenticated = False

def verify\_code(*self*, *code*: str):

print("Verifying code")

self.authenticated = True

def is\_authenticated(*self*):

return self.authenticated

class NotARobotAuthorizer(Authorizer):

def \_\_init\_\_(*self*):

self.authenticated = False

def ask(*self*):

print("Are you a robot?!!! [┐∵]┘")

self.authenticated = True

def is\_authenticated(*self*):

return self.authenticated

class PaymentProcessor(ABC):

@abstractmethod

def pay(*self*, *order*: Order):

...

class DebitPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str, *authorizer*: Authorizer):

self.security\_code = security\_code

self.authorizer = authorizer

def pay(*self*, *order*: Order):

if not self.authorizer.is\_authenticated():

raise Exception("Not authenticated")

print("Processing debit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

class CreditPaymentProcessor(PaymentProcessor):

def \_\_init\_\_(*self*, *security\_code*: str):

self.security\_code = security\_code

def pay(*self*, *order*: Order):

print("Processing credit payment type")

print(f"Verifying security code: {self.security\_code}")

order.status = "paid"

order = Order()

order.add\_item("Keyboard", 1, 50)

order.add\_item("SSD", 1, 150)

order.add\_item("USB cable", 2, 5)

authorizer = NotARobotAuthorizer()

authorizer.ask()

processor = DebitPaymentProcessor("0372846", authorizer)

processor.pay(order)

print(order.status)

Now the code adheres to the DIP because the high-level module DebitPaymentProcessor does not depend on the low-level class and instead depends on the high-level abstraction Authorizer. This means we can change the implementation of the Authorizer class we use without having to change the DebitPaymentProcessor class.

**Final thoughts**

As you can see, by thoughtfully applying the SOLID design principles, you can improve the maintainability and scalability of your applications. The practical application of these principles can lead to software that is more cohesive, less coupled, and easier to improve over the long term.

Play