# SOLID Design Principles

Through Python Examples

Hiral Amodia (amodia.hiral@gmail.com)
10-8-2020

**Introduction**

An efficient algorithm sets the base of an efficient software application. Once the algorithm is in place, the next most important thing in Software Engineering would be to ensure that the software or the application is designed with best designing and architecting practices. Many researchers and experts have defined various best practices towards efficient designing of software applications. One of the most popular among these are the design principles popular by the acronym – SOLID.

**What are SOLID Design Principles**

One of the most important and most popular are the Design Principles introduced by Robert C Martin (Uncle Bob). Uncle Bob has introduced various Design Principles and among them the most popular are the five principles acronymic as SOLID Design Principles that are primarily focused to Object Oriented Software Designing. These best practices, when considered in designing an object-oriented software application would tend to reduce code complexity, reduce risk of code breaks, improve the communication between different entities and make code more flexible, readable and manageable.

**Robert C Martin (Uncle Bob)**

*(Pic Courtesy: Wikipedia)*

**Uncle Bob's SOLID principles are as below:**

**S** – Single Responsibility Principle

**O** – Open-Closed Principle

**L** – Liskov Substitution Principle

**I** – Interface Segregation Principle

**D** – Dependency Inversion Principle

In this article I am sharing my understanding of Robert C. Martin's SOLID Design Principles along with Python examples.

**NOTE**: The code examples that I have shared are minimalist in nature and written with the only aim of explaining the respective principle. They may not be complete or may not adhere to some other principle or best practice. I request readers to kindly consider this aspect while referring to every code example shared with every principle.

# Single Responsibility Principle

**Single Responsibility Principle**

Single Responsibility Principle states that a class should have only one primary responsibility and should not take other responsibilities. Robert C. Martin explains this as "A class should have only one reason to change".

Eg.

Let's take an example of a Telephone Directory application. We are designing a Telephone Directory and that contains a TelephoneDirectory Class which is supposed to handle primary responsibility of maintaining Telephone Directory entries, i. e Telephone numbers and Names of the entities to which the Telephone Numbers belong. Thus, the operations that this class is expected to perform are adding a new entry (Name and Telephone Number), delete an existing entry, change a Telephone Number assigned to an entity Name, and provide a lookup that returns Telephone Number assigned to a particular entity Name.

Our TelephoneDirectory class could look as below:

```
#Single Responsibility Principle (aka Separation Of Concerns)

class TelephoneDirectory:
  def __init__(self):
    self.telephonedirectory = {}

  def add_entry(self, name, number):
    self.telephonedirectory[name] = number

  def delete_entry(self, name):
    self.telephonedirectory.pop(name)

  def update_entry(self, name, number):
    self.telephonedirectory[name] = number

  def lookup_number(self, name):
    return self.telephonedirectory[name]

  def __str__(self):
    ret_dct = ""
    for key, value in self.telephonedirectory.items():
      ret_dct += f'{key} : {value}\n'
    return ret_dct
```

```python
myTelephoneDirectory = TelephoneDirectory()
myTelephoneDirectory.add_entry("Ravi", 123456)
myTelephoneDirectory.add_entry("Vikas", 678452)
print(myTelephoneDirectory)

myTelephoneDirectory.delete_entry("Ravi")
myTelephoneDirectory.add_entry("Ravi", 123456)
myTelephoneDirectory.update_entry("Vikas", 776589)
print(myTelephoneDirectory.lookup_number("Vikas"))
print(myTelephoneDirectory)
```

```
Ravi : 123456
Vikas : 678452

776589
Vikas : 776589
Ravi : 123456
```

Till now, our TelephoneDirectory class looks good and it has exactly implements the expected features.

Now let’s say that there are two more requirements in the project – Persist the contents of Telephone Directory to a Database and transfer the contents of Telephone Directory to a file.

So, we can add two more methods to the TelephoneDirectory class as below:

```python
#Breaking Single Responsibility Principle (aka Separation Of Concerns)

class TelephoneDirectory:
  def __init__(self):
    self.telephonedirectory = {}

  def add_entry(self, name, number):
    self.telephonedirectory[name] = number

  def delete_entry(self, name):
    self.telephonedirectory.pop(name)

  def update_entry(self, name, number):
    self.telephonedirectory[name] = number

  def lookup_number(self, name):
    return self.telephonedirectory[name]

  def save_to_file(self, file_name, location):
    #code to save the contents of telephonedirectory dictionary to the file
    pass

  def persist_to_database(self, database_details):
    #code to persist the contents of telephonedirectory dictionary to database
    pass

  def __str__(self):
    ret_dct = ""
    for key, value in self.telephonedirectory.items():
      ret_dct += f'{key} : {value}\n'
    return ret_dct
```

Now this is where we broke the Single Responsibility Design Principle. By adding the functionalities of persisting to database and saving to file, we gave additional responsibilities to TelephoneDirectory class which are not its primary responsibility. This class now has additional features that can cause it to change. In future if there are any requirements related to persisting the data then those can cause changes to the TelephoneDirectory class. Thus, TelephoneDirectory is prone to changes due to the reasons that are not its primary responsibility.

Single Responsibility Principle asks us not to add additional responsibilities to a class so that we don't have to modify a class unless there is change to its primary responsibility. We can handle the current situation by having separate classes that would handle database persistence and saving to file. We can pass the TelephoneDirectory object to the objects of those classes and write any additional features in those classes.

This would ensure that TelephoneDirectory class has only one reason to change that is any change in its primary responsibility

```python
class persist_to_database:
  #functionality of the class
  def __init__(self, object_to_persist):
    pass

class save_to_file:
  #functionality of the class
  def __init__(self, object_to_save):
    pass
```

You can find above code examples as downloadable files on GitHub location:
https://github.com/amodiahs/SOLID_Design_Principles/tree/master/Single_Responsibility_Principle

# Open-Closed Principle

**Open-Closed Principle**

Open Closed Principle was first conceptualized by Berterd Meyer in 1988. Robert C. Martin mentioned this as “the most important principle of object-oriented design”.

Open Closed Principle states that “Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”

Following this principle ensures that a class is well defined to do what it is supposed to do. Adding any further features can be done through creating new entities that extend the existing class’s features and add more features to itself. Thus preventing frequent and trivial changes to a well-established low level class.

E.g.

Let’s say we have an application for an apparel store. Among various features in the system, there is also a feature to apply select discount based on type of apparel.

Below example shows one way of implementing this requirement.

In the below example we have a class DiscocuntCalculator that has a property to hold the type of apparel. It has a function that calculates the discount based on the type of apparel and returns the new cost after deducting the discount amount.

```python
#Open - Close Principle - Incorrect implementation
from enum import Enum
class Products(Enum):
  SHIRT = 1
  TSHIRT = 2
  PANT = 3

class DiscountCalculator():
  def __init__(self, product_type, cost):
    self.product_type = product_type
    self.cost = cost

  def get_discounted_price(self):
    if self.product_type == Products.SHIRT:
      return self.cost - (self.cost * 0.10)
    elif self.product_type == Products.TSHIRT:
      return self.cost - (self.cost * 0.15)
    elif self.product_type == Products.PANT:
      return self.cost - (self.cost * 0.25)

dc_Shirt = DiscountCalculator(Products.SHIRT, 100)
print(dc_Shirt.get_discounted_price())

dc_TShirt = DiscountCalculator(Products.TSHIRT, 100)
print(dc_TShirt.get_discounted_price())

dc_Pant = DiscountCalculator(Products.PANT, 100)
print(dc_Pant.get_discounted_price())
```

```
90.0
85.0
75.0
```

This design breaches the Open Closed principle because this class will need modification if a). A new apparel type is to be included and b). If the discount amount for any apparel changes.

This feature can be implemented efficiently as below.

```python
#Open - Close Principle - Correct implementation
from enum import Enum
from abc import ABCMeta, abstractmethod

class DiscountCalculator():

  @abstractmethod
  def get_discounted_price(self):
    pass

class DiscountCalculatorShirt(DiscountCalculator):
  def __init__(self, cost):
    self.cost = cost

  def get_discounted_price(self):
      return self.cost - (self.cost * 0.10)

class DiscountCalculatorTshirt(DiscountCalculator):
  def __init__(self, cost):
    self.cost = cost

  def get_discounted_price(self):
      return self.cost - (self.cost * 0.15)

class DiscountCalculatorPant(DiscountCalculator):
  def __init__(self, cost):
    self.cost = cost

  def get_discounted_price(self):
      return self.cost - (self.cost * 0.25)

dc_Shirt = DiscountCalculatorShirt(100)
print(dc_Shirt.get_discounted_price())

dc_TShirt = DiscountCalculatorTshirt(100)
print(dc_TShirt.get_discounted_price())

dc_Pant = DiscountCalculatorPant(100)
print(dc_Pant.get_discounted_price())
```

```
90.0
85.0
75.0
```

As shown in above example we have now a very simple base class DiscountCalculator that has a single abstract method get_discounted_price. We have created new classes for apparels that extends the base DiscountCalculator class. Hence now every sub class would need to implement the discount part on itself. By doing this we have now removed the previous constraints that required modification to the base class. Now without modifying the base class we can add more apparels as well as we can change discount amount of an individual apparel as needed.

You can find above code examples as downloadable files on GitHub location: [https://github.com/amodiahs/SOLID_Design_Principles/tree/master/Open_Closed_Principle](https://github.com/amodiahs/SOLID_Design_Principles/tree/master/Open_Closed_Principle)

# Liskov Substitution Principle

**Liskov Substitution Principle**

Liskov Substitution Principle was one of the toughest principles for me to understand and I had to look at various examples on the internet to understand it correctly. But I feel that once understood properly, it is one of the easiest principles to adhere to while designing Object Oriented Applications.

Liskov Substitution Principle states that "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."

Liskov Substitution Principle was introduced by **Barbara Liskov** through a subtyping relation called Strong Behavioral Subtyping. This principle states that if a class *Sub* is subtype of a class *Sup*, then in the program, objects of type *Sup* should be easily substituted with objects of type *Sub* without needing to change the program. Uncle Bob included this as one of the top 5 SOLID Design Principles definition.

E.g.

Let's say we have a base class Car that would indicate what is the type of a car. Car class is inherited by sub class PetrolCar. Similarly, the base class Car can be inherited by other classes which can extend the features as desired.

```
# Liskov Substitution Principle
class Car():
  def __init__(self, type):
    self.type = type

class PetrolCar(Car):
  def __init__(self, type):
    self.type = type

car = Car("SUV")
car.properties = {"Color": "Red", "Gear": "Auto", "Capacity": 6}
print(car.properties)

petrol_car = PetrolCar("Sedan")
petrol_car.properties = ("Blue", "Manual", 4)
print(petrol_car.properties)
```

```
{'Color': 'Red', 'Gear': 'Auto', 'Capacity': 6}
('Blue', 'Manual', 4)
```

As we can see here, there is no standard specification to add properties of the Car and it is left to the developers to implement in the way convenient to them. One developer may implement it as Dictionary and another may implement it as a Tuple and thus it can be implemented in multiple ways.

Till here there is no problem. But let's say that there is a requirement to find all Red colored cars. Let's try to write a function that would take all the Cars and try to find out Red cars based on the implementation of object of "Car" Super Class.

```
#Breaking - Liskov Substitution Principle
class Car():
  def __init__(self, type):
    self.type = type

class PetrolCar(Car):
  def __init__(self, type):
    self.type = type

car = Car("SUV")
car.properties = {"Color": "Red", "Gear": "Auto", "Capacity": 6}

petrol_car = PetrolCar("Sedan")
petrol_car.properties = ("Blue", "Manual", 4)

cars = [car, petrol_car]

def find_red_cars(cars):
  red_cars = 0
  for car in cars:
    if car.properties['Color'] == "Red":
      red_cars += 1
  print(f'Number of Red Cars = {red_cars}')

find_red_cars(cars)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-0f0b83b0e35e> in <module>()
     23   print(f'Number of Red Cars = {red_cars}')
     24 
---> 25 find_red_cars(cars)

<ipython-input-5-0f0b83b0e35e> in find_red_cars(cars)
     19   red_cars = 0
     20   for car in cars:
---> 21     if car.properties['Color'] == "Red":
     22       red_cars += 1
     23   print(f'Number of Red Cars = {red_cars}')

TypeError: tuple indices must be integers or slices, not str
```

As we can see here, we are trying to loop through a list of car objects. And here we break the Liskov Substitution principle as we cannot replace Super type Car's objects with objects of Sub type PetrolCar in the function written to find Red cars.

A better way to implement this would be to introduce setter and getter methods in the Super class Car using which we can set and get Car's properties without leaving that implementation to individual developers. This way we just get the properties through a setter method and its implementation remains internal to the Super class.

By doing this we can comply to Liskov's Substitution Principle as shown below:

```
[ ] #Correct implementation - Liskov Substitution Principle
    class car():
      def __init__(self, type):
        self.type = type
        self.car_properties = {}

      def set_properties(self, color, gear, capacity):
        self.car_properties = {"Color": color, "Gear": gear, "Capacity": capacity}

      def get_properties(self):
        return self.car_properties

    class petrol_car(car):
      def __init__(self, type):
        self.type = type
        self.car_properties = {}

    car = car("SUV")
    car.set_properties("Red", "Auto", 6)

    petrol_car = petrol_car("Sedan")
    petrol_car.set_properties("Blue", "Manual", 4)

    cars = [car, petrol_car]

    def find_red_cars(cars):
      red_cars = 0
      for car in cars:
        if car.get_properties()['Color'] == "Red":
          red_cars += 1
      print(f'Number of Red Cars = {red_cars}')

    find_red_cars(cars)
```

```
Number of Red Cars = 1
```

You can find above code examples as downloadable files on GitHub location:
https://github.com/amodiahs/SOLID_Design_Principles/tree/master/Liskov_Substitution_Principle

# Interface Segregation Principle

**Interface Segregation Principle**

Interface Segregation Principle states that “No client should be forced to depend on methods it does not use”.

Interface Segregation Principle was introduced by Robert C Martin while he was consulting for Xerox.

Interface Segregation Principle suggests creating of smaller interfaces known as “role interfaces” instead of a large interface consisting of multiple methods. By segregating the role-based methods into smaller role interfaces, the clients would depend only on the methods that are relevant to it.

E.g.

Let’s say we are designing an application for different communication devices. We identify that a communication device is a device that would have one or many of these features – a) to make calls, b). send SMS and c). browse the Internet. So, we create an interface named CommunicationDevice and add the respective abstract methods for each of these features such that any implementing class would need to implement these methods.

We then create a class SmartPhone using the CommunicationDevice interface and implement the functionalities of the abstract methods. Till here its all fine.

Now say that we want to implement a traditional Landline phone. This also being a communication device, we create a new class LandlinePhone using the same CommunicationDevice interface. This is exactly when we face the problem due to a large CommunicationDevice interface we created. In the class LanlinePhone we implement make_calls() method, but as we also inherit abstract methods send_sms() and browse_internet() we have to provide implementation of these two abstract methods also in the LandlinePhone class even if these are not applicable to this class LandlinePhone. We can either throw exception or just write pass in the implementation, but we still need to provide implementation.

```python
#Interface Substitution Principle - Incorrect Implementation
class CommunicationDevice():
  @abstractmethod
  def make_calls():
    pass

  @abstractmethod
  def send_sms():
    pass

  @abstractmethod
  def browse_internet():
    pass

class SmartPhone(CommunicationDevice):
  def make_calls():
    #implementation
    pass

  def send_sms():
    #implementation
    pass

  def browse_internet():
    #implementation
    pass

  class LandlinePhone(CommunicationDevice):
    def make_calls():
      #implementation
      pass

    def send_sms():
      #just pass or raise exception as this feature is not supported
      pass

    def browse_internet():
      #just pass or raise exception as this feature is not supported
      pass
```

This can be corrected by following Interface Segregation Principle as in below example. Instead of creating a large interface we create smaller role interfaces for each method. The respective classes would only use the related interfaces.

```python
from abc import ABCMeta, abstractmethod

#interface Substitution Principle - Correct Implementation

class CallingDevice():
  @abstractmethod
  def make_calls():
    pass

class MessagingDevice():
  @abstractmethod
  def send_sms():
    pass

class InternetbrowsingDevice():
  @abstractmethod
  def browse_internet():
    pass

class SmartPhone(CallingDevice, MessagingDevice, InternetbrowsingDevice):
  def make_calls():
    #implementation
    pass

  def send_sms():
    #implementation
    pass

  def browse_internet():
    #implementation
    pass

class LandlinePhone(CallingDevice):
  def make_calls():
    #implementation
    pass
```

You can find above code examples as downloadable files on GitHub location:
https://github.com/amodiahs/SOLID_Design_Principles/tree/master/Interface_Segregation_Principle

# Dependency Inversion Principle

**Dependency Inversion Principle**

Dependency Inversion Principle states that:

a). High level module should not depend on low level modules. Both should depend on abstractions b). Abstractions should not depend on details. Details should depend on abstractions.

If your code follows Open-Closed Principle and Liskov Substitution Principle, then it will be implicitly aligned to be compliant to Dependency Inversion Principle also.

By following Open-Closed Principle, you create Interfaces that can be used to provide different high-level implementations. By following Liskov Substitution Principle you ensure that you can replace the low-level class objects with high-level class objects without causing any adverse effect on the application. Thus, by following these two principles you ensure that your high-level classes and low-level classes depend on interfaces. Hence you would implicitly follow Dependency Inversion Principle.

E.g.

As shown in below code we have a class Student which we use to create Student entities and a class TeamMemberships which holds memberships of different students into different teams.

Now we define a high-level class Analysis where we need to find out all students belonging to RED team.

```python
#Dependency Inversion Principle - Incorrect implementation
from enum import Enum
from abc import ABCMeta, abstractmethod

class Teams(Enum):
  BLUE_TEAM = 1
  RED_TEAM = 2
  GREEN_TEAM = 3

class Student:
  def __init__(self, name):
    self.name = name

class TeamMemberships():
  def __init__(self):
    self.team_memberships = []

  def add_team_memberships(self, student, team):
    self.team_memberships.append((student, team))

class Analysis():
  def __init__(self, team_student_memberships):
    memberships = team_student_memberships.team_memberships
    for members in memberships:
      if members[1] == Teams.RED_TEAM:
        print(f'{members[0].name} is in RED team')

student1 = Student('Ravi')
student2 = Student('Archie')
student3 = Student('James')

team_memberships = TeamMemberships()
team_memberships.add_team_memberships(student1, Teams.BLUE_TEAM)
team_memberships.add_team_memberships(student2, Teams.RED_TEAM)
team_memberships.add_team_memberships(student3, Teams.GREEN_TEAM)

Analysis(team_memberships)
```

```
Archie is in RED team
```

As we can see in this implementation, we are directly using team_student_memberships.team_memberships  In our high level class Analysis and we are using the implementation of this list directly in high level class. As of now this is fine but imagine a situation in which we need to change this implementation from list to something else. In that case our high-level class Analysis would break as it is dependent on implementation details of Low level class TeamMemberships.

Now see the below example where we change this implementation and make it comply to Dependency Inversion Principle

```
#Dependency Inversion Principle - Correct implementation
from enum import Enum
from abc import ABCMeta, abstractmethod

class Teams(Enum):
  BLUE_TEAM = 1
  RED_TEAM = 2
  GREEN_TEAM = 3

class TeamMembershipLookup():
  @abstractmethod
  def find_all_students_of_team(self, team):
    pass

class Student:
  def __init__(self, name):
    self.name = name

class TeamMemberships(TeamMembershipLookup):
  def __init__(self):
    self.team_memberships = []

  def add_team_memberships(self, student, team):
    self.team_memberships.append((student, team))

  def find_all_students_of_team(self, team):
    for members in self.team_memberships:
      if members[1] == team:
        yield members[0].name

class Analysis():
  def __init__(self, team_membership_lookup):
    for student in team_membership_lookup.find_all_students_of_team(Teams.RED_TEAM):
      print(f'{student} is in RED team.')

student1 = Student('Ravi')
student2 = Student('Archie')
student3 = Student('James')

team_memberships = TeamMemberships()
team_memberships.add_team_memberships(student1, Teams.BLUE_TEAM)
team_memberships.add_team_memberships(student2, Teams.RED_TEAM)
team_memberships.add_team_memberships(student3, Teams.GREEN_TEAM)

Analysis(team_memberships)
```

```
Archie is in RED team.
```

To comply to Dependency Inversion Principle, we need to ensure that high level class Anslysis should not depend on concrete implementation of low level class TeamMemberships. Instead it should depend on some abstraction.

So, we create an interface TeamMembershipLookup that contains an abstract method find_all_students_of_team which is passed to any class that inherits from this interface. We make our TeamMembership class to inherit from this interface and hence now TeamMembership class needs to provide an implementation of the find_all_students_of_team function. This function then yields the results to any other calling entity. We moved the processing that was done in high-level Analysis class to TeamMemberships class through the interface TeamMembershipLookup.

So, by doing this we have removed dependency of high level class Analysis from low level class TeamMemberships and transferred this dependency to interface TeamMembershipLookup. Now the high-level class doesn't depend on implementation details of low level class. Any changes to the implementation details of low level class doesn't impact the high-level class.

You can find above code examples as downloadable files on GitHub location:
https://github.com/amodiahs/SOLID_Design_Principles/tree/master/Dependency_Inversion_Principle

**Summary**:

| Principle | Concept |
|---|---|
| Single Responsibility Principle | A class should have only one reason to change |
| Open-Closed Principle | Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification |
| Liskov Substitution Principle | Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program |
| Interface Segregation Principle | No client should be forced to depend on methods it does not use |
| Dependency Inversion Principle | a). High level module should not depend on low level modules. Both should depend on abstractions<br>b). Abstractions should not depend on details. Details should depend on abstractions |

**References/Gratitude:**

a). Dmitri Nesteruk for his Udemy course on Design Patterns

b). Jordan Hudgens of DevCamp for his tutorials on YouTube

c). Wikipedia for the amazing informative content

**About Me:**

My Name is **Hiral Amodia** and I am based at Bangalore, India. I am a Software Engineer working in Indian IT Industry for over 16 years now. Currently I am employed as Software Engineering Manager at a leading Indian IT services company. I am passionate about learning new technologies and concepts. I strongly believe that teaching is the best way of learning and that caring is the true way of sharing. So, I keep on writing articles on technology/concepts etc. that I learn.

Feel free to buzz me on my below coordinates if you want to share any feedback or improvement areas that you encounter in my articles.

**My Coordinates as below:**



| | |
|---|---|
| Email | Amodia.hiral@gmail.com |
| LinkedIn | https://www.linkedin.com/in/hiral-amodia/ |
| GitHub | https://github.com/amodiahs |