# Exercise: SOLID

The following problem descriptions **do not require submissions** to the **Judge System**.

## Workers

You are provided with a code on which you have to apply the **DIP** (Dependency Inversion Principle) so that when adding new **worker classes**, the Manager class will work properly.

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| worker = Worker()  manager = Manager()  manager.set\_worker(worker)  manager.manage()  super\_worker = SuperWorker()  try:  manager.set\_worker(super\_worker)  except AssertionError:  print("manager fails to support super\_worker....") | I'm working!!  manager fails to support super\_worker.... |
| **After** | **Result** |
| worker = Worker()  manager = Manager()  manager.set\_worker(worker)  manager.manage()  super\_worker = SuperWorker()  try:  manager.set\_worker(super\_worker)  manager.manage()  except AssertionError:  print("manager fails to support super\_worker....") | I'm working!!  I work very hard!!! |

## Workers - Updated

You are provided with a code on which you have to apply the **ISP** (Interface Segregation Principle) by **splitting** the **Worker** class into two classes (**Workable** and **Eatable**), so the **Robot** class no longer needs to implement the **eat** method

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| manager = Manager()  manager.set\_worker(Worker())  manager.manage()  manager.lunch\_break()  manager.set\_worker(SuperWorker())  manager.manage()  manager.lunch\_break()  manager.set\_worker(Robot())  manager.manage()  manager.lunch\_break() | I'm normal worker. I'm working.  Lunch break....(5 secs)  I'm super worker. I work very hard!  Lunch break....(3 secs)  I'm a robot. I'm working....  I don't need to eat.... |
| **After** | **Result** |
| work\_manager = WorkManager()  break\_manager = BreakManager()  work\_manager.set\_worker(Worker())  break\_manager.set\_worker(Worker())  work\_manager.manage()  break\_manager.lunch\_break()  work\_manager.set\_worker(SuperWorker())  break\_manager.set\_worker(SuperWorker())  work\_manager.manage()  break\_manager.lunch\_break()  work\_manager.set\_worker(Robot())  work\_manager.manage()  try:  break\_manager.set\_worker(Robot())  break\_manager.lunch\_break()  except:  pass | I'm normal worker. I'm working.  Lunch break....(5 secs)  I'm super worker. I work very hard!  Lunch break....(3 secs)  I'm a robot. I'm working.... |

## Prisoner

You are provided with a code containing a class **Prisoner** and a **class Person**. A **prisoner** is a **person**, but since a **prisoner** is **not free** to move an arbitrary distance, the **Person** class can be named **FreePerson**, then the idea that a **Prisoner inherits FreePerson** is **wrong**. Rewrite the code and apply the **LSP** (Liskov Substitution Principle).

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| prisoner = Prisoner()  print("The prisoner trying to walk to north by 10 and east by -3.")  try:  prisoner.walk\_north(10)  prisoner.walk\_east(-3)  except:  pass  print(f"The location of the prison: {prisoner.PRISON\_LOCATION}")  print(f"The current position of the prisoner: {prisoner.position}") | The prisoner trying to walk to north by 10 and east by -3.  The location of the prison: [3, 3]  The current position of the prisoner: [0, 13] |
| **After** | **Result** |
| prisoner = Prisoner()  print("The prisoner trying to walk to north by 10 and east by -3.")  try:  prisoner.walk\_north(10)  prisoner.walk\_east(-3)  except:  pass  print(f"The location of the prison: {prisoner.PRISON\_LOCATION}")  print(f"The current position of the prisoner: {prisoner.position}") | The prisoner trying to walk to north by 10 and east by -3.  The location of the prison: (3, 3)  The current position of the prisoner: (3, 3) |

## Shapes

You are provided with code containing **class Rectangle** and **class AreaCalculator**. Refactor the code using the **Open/Closed Principle** so that the code is open for extension (adding **more shapes**) but closed for modification.

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| shapes = [Rectangle(2, 3), Rectangle(1, 6)]  calculator = AreaCalculator(shapes)  print("The total area is: ", calculator.total\_area) | The total area is: 12 |
| **After** | **Result** |
| shapes = [Rectangle(1, 6), Triangle(2, 3)]  calculator = AreaCalculator(shapes)  print("The total area is: ", calculator.total\_area) | The total area is: 9.0 |

## Emails

You are provided with code containing **class IEmail** and **class Email**. The code does not follow the principle of **single responsibility** (the Email class has **2 responsibilities**). Create a new **class - IContent**, and a class that inherits it called **MyContent** to split the responsibilities.

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| email = Email('IM', 'MyML')  email.set\_sender('qmal')  email.set\_receiver('james')  email.set\_content('Hello, there!')  print(email) | Sender: I'm qmal  Receiver: I'm james  Content:  <myML>  Hello, there!  </myML> |
| **After** | **Result** |
| email = Email('IM')  email.set\_sender('qmal')  email.set\_receiver('james')  content = MyContent('Hello, there!')  email.set\_content(content)  print(email) | Sender: I'm qmal  Receiver: I'm james  Content:  <MyML>Hello, there!</MyML> |

## Media Management System

You are developing a **media management system** that **handles different types of media**: **Books**, **eBooks**, and **Audiobooks**. The **initial design violates** several principles: the **Single Responsibility Principle** (**SRP**), the **Open/Closed Principle** (**OCP**), the **Liskov Substitution Principle** (**LSP**), and the **Interface Segregation Principle** (**ISP**) because it **forces** different media classes to **implement methods irrelevant** to **their specific functionality**. Specifically:

* **Book** is required to implement **listen()** even though it doesn't need this functionality.
* **EBook** is also required to implement **listen()** even though it should not have this method.
* **Audiobook** is required to implement **read()** even though it doesn't need this functionality.

Tasks:

* **Refactor** the design by creating **separate interfaces**: **Borrowable**, **Readable**, and **Listenable**.
* Note that **each book must be borrowed before** it can be **read** or **listened to**.

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| book = Book() book.borrow("user123") book.read() book.listen() *# No effect, but must be present* ebook = EBook() ebook.borrow("user456") ebook.read() ebook.listen() *# No effect, but must be present* audiobook = Audiobook() audiobook.borrow("user789") audiobook.read() *# No effect, but must be present* audiobook.listen() | Book borrowed by user user123.  Reading the book. Progress: 10%  eBook borrowed by user user456. DRM applied.  Reading the eBook. Progress: 20%  Audiobook borrowed by user user789.  Listening to the audiobook. Progress: 15% |
| **After** | **Result** |
| book = Book() book.borrow("user123") book.read()  *try*:  book.listen() *except* AttributeError *as* e:  print(e)  ebook = EBook() ebook.borrow("user456") ebook.read()  *try*:  ebook.listen() *except* AttributeError *as* e:  print(e)  audiobook = Audiobook() audiobook.borrow("user789") audiobook.listen()  *try*:  audiobook.read() *except* AttributeError *as* e:  print(e) | Book borrowed by user user123.  Reading the book. Progress: 10%  'Book' object has no attribute 'listen'  eBook borrowed by user user456. DRM applied.  Reading the eBook. Progress: 20%  'EBook' object has no attribute 'listen'  Audiobook borrowed by user user789.  Listening to the audiobook. Progress: 15%  'Audiobook' object has no attribute 'read' |

## \*Notification System

You are developing a **notification system** that supports different **types of notifications**: **Email**, **SMS**, and **Push notifications**. The initial design has a **single NotificationSender** **class** that **enforces** a common interface for all notification types. This design **violates** the **Interface Segregation Principle** (**ISP**), the **Open/Closed Principle** (**OCP**),the **Liskov Substitution Principle** (**LSP**), and the **Single Responsibility Principle** (**SRP**).

Some **Violation Examples**:

* **EmailSender** and **SMSSender** are required to implement the **send()** method, which is appropriate for them.
* **PushSender** implements **send()** but the functionality is **currently** **under maintenance**, so it should **handle** this scenario **gracefully**.
* The **NotificationService** **class** directly **depends** on **specific notification types**, which **violates** the **Open/Closed Principle** (OCP) by requiring modifications to support new notification types.

Your Tasks:

* Refactor the **NotificationSender** Class:
  + Create a base interface, **NotificationSender**, that defines the common method **send()**.
  + Ensure that **NotificationSender** does **not include methods** that are **not applicable** to all notification types.
* Implement Concrete **Notification** Classes:
  + **EmailSender**: Implements **NotificationSender** and provides functionality for sending email notifications.
  + **SMSSender**: Implements **NotificationSender** and provides functionality for sending SMS notifications.
  + **PushSender**: Implements **NotificationSender** and provides functionality for push notifications when the service is normally working.
  + **Handle** **scenarios** where **any service can be** **under maintenance**. Return feedback that the **service** is **currently unavailable**.
* Refactor the **NotificationService** Class:
* Modify **NotificationService** to depend on the **NotificationSender** abstraction rather than specific notification implementations.
* Ensure that **NotificationService** can handle new types of notifications by accepting any object that implements **NotificationSender**.
* Handle **Maintenance Status** Appropriately:
* If a notification type is **under maintenance**, provide an **appropriate message**.

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| try:  email\_service = NotificationService("email")  email\_service.notify("Hello via email!")  sms\_service = NotificationService("sms")  sms\_service.notify("Hello via SMS!")  push\_service = NotificationService("push")  push\_service.notify("Hello via Push!")  except UnderMaintenanceException as ex:  print(ex) | Sending email with message: Hello via email!  Sending SMS with message: Hello via SMS!  The Push Sender is under maintenance. |
| **After** | **Result** |
| email\_sender = EmailSender()  sms\_sender = SMSSender()  push\_sender = PushSender()  push\_sender.is\_under\_maintenance = True  *"""*  *The status change represents manual toggling done by an admin or a developer when they know a system component is under maintenance.*  *In more complex systems, this would be automated or pulled from an external source.*  *"""*  email\_service = NotificationService(email\_sender)  sms\_service = NotificationService(sms\_sender)  push\_service = NotificationService(push\_sender)  email\_service.notify('Hello via email!')  sms\_service.notify('Hello via SMS!')  push\_service.notify('Hello via Push!') | Sending email with message: Hello via email!  Sending SMS with message: Hello via SMS!  This service is currently under maintenance. |

## \*Payment System

You are building a **simple payment processing system** for an e-commerce application. The current system (the provided code) has a single **Order** **class** that handles **multiple responsibilities**: managing order items, calculating the total price, and processing payments for **different payment methods** like **Credit Card** and **PayPal**. This design **does not adhere** to **key architecture principles**, specifically the **Single Responsibility Principle** (**SRP**), the **Open/Closed Principle** (**OCP**), and the **Dependency Inversion Principle** (**DIP**).

Your Tasks:

* Create a **PaymentMethod** Interface (Abstract Base Class):
  + Define an interface or abstract base class for **processing payments**.
  + This class should have a **method** that **all concrete payment method classes** **must implement**.
    - The method should define the contract for payment processing.
* Create a **PaymentProcessor** class:
  + Define a **class** that accepts a **PaymentMethod** object.
  + Implement a **method** that **processes payments** using the provided **PaymentMethod** object for a given **Order**.
* Implement concrete **payment classes**:
* **CreditCardPayment** class that handles credit card payment processing.
* **PayPalPayment** class that handles PayPal payment processing.
  + Each class should implement the **PaymentMethod** interface.
* Refactor the **Order** class:
* **Remove** any payment processing logic from the **Order** class.
* **Modify** the **Order** class to use the **PaymentProcessor** class to process payments.
* Ensure that the **Order** class depends on the **PaymentMethod** interface rather than specific payment classes, adhering to the **Dependency Inversion Principle** (DIP) by depending on an abstraction.

By the end of this task, **your system should be designed** to **easily accommodate new payment methods** by **implementing new classes** that follow the **PaymentMethod** interface. The **Order** class should **not need modification** when new payment methods are introduced, ensuring adherence to both the **Open/Closed Principle** (OCP) and **Dependency Inversion Principle** (DIP).

### Examples

|  |  |
| --- | --- |
| **Before** | **Result** |
| order\_obj = Order([  ('Apple', 2, 1.0),  ('Banana', 5, 0.5)  ]) order\_obj.process\_payment('credit\_card') | Processing credit card payment for $4.5 |
| **After** | **Result** |
| order\_obj = Order([  ('Apple', 2, 1.0),  ('Banana', 5, 0.5)  ])  credit\_card\_payment = CreditCardPayment() payment\_processor = PaymentProcessor(credit\_card\_payment) payment\_processor.process\_payment(order\_obj) | Processing credit card payment for $4.5 |