Copilot

Sure, let's take a more real-world example of a simple system for managing a library. We'll use these examples to illustrate the SOLID principles.

1. **Single Responsibility Principle (SRP)**: A class should have only one reason to change. Let's say we have a `Book` class. This class should only be responsible for book-related properties and methods.

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages
```

1. **Open-Closed Principle (OCP)**: Classes should be open for extension but closed for modification. Let's say we want to add a method to calculate late fees for a `LibraryItem`. We can create a new class that extends `LibraryItem` instead of modifying it.

```
class LibraryItem:
    def __init__(self, title):
        self.title = title

class Book(LibraryItem):
    def __init__(self, title, author, pages):
        super().__init__(title)
        self.author = author
        self.pages = pages

    def calculate_late_fee(self, days_late):
        return days_late * 0.25  # 25 cents per day late
```

1. **Liskov Substitution Principle (LSP)**: Subtypes must be substitutable for their base types. If we have a method that accepts a `LibraryItem`, it should work correctly if we pass in a `Book` (which is a subtype of `LibraryItem`).

```
def print_item_title(item: LibraryItem):
    print(item.title)

book = Book("Moby Dick", "Herman Melville", 635)
print_item_title(book)  # This should work correctly
```

1. **Interface Segregation Principle (ISP)**: Clients should not be forced to depend on interfaces they do not use. If we have a `LibraryMember` class, it should not have to know about methods related to library administration.

```
class LibraryMember:
    def check_out_item(self, item: LibraryItem):
        pass

class Librarian:
    def check_in_item(self, item: LibraryItem):
        pass

    def calculate_late_fee(self, item: LibraryItem, days_late: int):
        pass
```

1. **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules. Both should depend on abstractions. Instead of having `LibraryMember` directly instantiate a `Book`, we can have `Book` be passed in as a parameter.

```
class LibraryMember:
    def __init__(self, name):
        self.name = name
        self.checked_out_items = []
    def check_out_item(self, item: LibraryItem):
        self.checked_out_items.append(item)
```

These examples are still somewhat simplified, but hopefully they give you a better idea of how you might apply the SOLID principles in a more real-world scenario. Remember, these principles are guidelines, not hard rules. They can help guide your design decisions, but there may be times when it makes sense to deviate from them. The key is to understand the trade-offs involved and make informed decisions. Happy coding! 😊