LAPORAN PRAKTIKUM PEMROGRAMAN BERORIENTASI OBJEK(PBO)

PRAKTIKUM 11

2411102441205

Andi Muh Fitrah Andi Kambe

FAKULTAS SAINS DAN TEKNOLOGI

PROGRAM STUDI S1 TEKNIK INFORMATIKA

UNIVERSITAS MUHAMMADIYAH KALIMANTAN TIMUR

EMAIL : 2411102441205@umkt.ac.id

Latar Belakang

Dalam pengembangan perangkat lunak, pembuatan satu kelas besar yang menangani terlalu banyak tugas (*God Class*) sering kali menjadi kendala. Meskipun praktis di awal, struktur ini merupakan *code smell* yang membuat program menjadi kaku, sulit dikelola, dan rapuh terhadap perubahan.

Masalah biasanya muncul saat perubahan kecil pada satu fungsi justru merusak bagian lain akibat ketergantungan kode yang terlalu kuat. Untuk mengatasinya, diperlukan *refactoring* dengan menerapkan prinsip SOLID, khususnya SRP, OCP, dan DIP. Dengan memecah tanggung jawab kelas menjadi komponen yang lebih spesifik dan menggunakan abstraksi melalui *Dependency Injection*, kita dapat menciptakan kode yang lebih modular dan fleksibel. Praktikum ini bertujuan mentransformasi kode yang kaku tersebut agar sistem lebih mudah dikembangkan di masa depan tanpa mengganggu stabilitas kode yang sudah ada.

Tujuan Praktikum

Melalui praktikum ini, mahasiswa diharapkan dapat:

1. Menganalisis pelanggaran Single Responsibility Principle (SRP) pada kelas monolitik (*God Class*).Mengimplementasikan interaksi antar objek melalui method dan atribut.
2. Menggunakan *Abstraksi* (Abstract Class) untuk memisahkan tanggung jawab dan memenuhi Dependency Inversion Principle (DIP).
3. Membuat desain yang terbuka untuk ekstensi dan tertutup untuk modifikasi (OCP) menggunakan *Dependency Injection*.
4. Menerapkan proses *Refactoring* dari kode yang kaku (rigid) menjadi kode yang fleksibel.

EMAIL : 2411102441205@umkt.ac.id

C. Langkah – Langkah Praktikum.

refactor_solid.py > OrderManager > process_checkout

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass

@dataclass
class Order:
    customer_name: str
    total_price: float
    status: str = "open"

# === KODE BURUK (SEBELUM REFACTOR) ===
class OrderManager: # Melanggar SRP, OCP, DIP
    def process_checkout(self, order: Order, payment_method: str):
        print(f"Memulai checkout untuk {order.customer_name}...")

        # LOGIKA PEMBAYARAN (Pelanggaran OCP/DIP)
        if payment_method == "credit_card":
            # Logika detail implementasi hardcoded di sini
            print("Processing Credit Card...")
        elif payment_method == "bank_transfer":
            # Logika detail implementasi hardcoded di sini
            print("Processing Bank Transfer...")
        else:
            print("Metode tidak valid.")
            return False

        # LOGIKA NOTIFIKASI (Pelanggaran SRP)
        print(f"Mengirim notifikasi ke {order.customer_name}...")
        order.status = "paid"

        return True
```

```python
class INotificationService(ABC):
    """Kontrak: Semua layanan notifikasi harus punya method 'send'."""
    @abstractmethod
    def send(self, order: Order):
        pass

# --- IMPLEMENTASI KONKRIT (Plug-in) ---
class CreditCardProcessor(IPaymentProcessor):
    def process(self, order: Order) -> bool:
        print("Payment: Memproses Kartu Kredit.")
        return True

class EmailNotifier(INotificationService):
    def send(self, order: Order):
        print(f"Notif: Mengirim email konfirmasi ke {order.customer_name}.")

# --- KELAS KOORDINATOR (SRP & DIP) ---
class CheckoutService: # Tanggung jawab tunggal: Mengkoordinasi Checkout
    def __init__(self, payment_processor: IPaymentProcessor, notifier: INotificationService):
        # Dependency Injection (DIP): Bergantung pada Abstraksi, bukan Konkrit
        self.payment_processor = payment_processor
        self.notifier = notifier

    def run_checkout(self, order: Order):
        payment_success = self.payment_processor.process(order) # Delegasi 1

        if payment_success:
            order.status = "paid"
            self.notifier.send(order) # Delegasi 2
            print('Checkout Sukses.')
            return True

        return False
    # --- PROGRAM UTAMA ---
# 1. Setup Dependencies
andi_order = Order("Andi", 500000)
email_service = EmailNotifier()
```

```python
# 1. Inject implementasi Credit Card
cc_processor = CreditCardProcessor()
checkout_cc = CheckoutService(payment_processor=cc_processor, notifier=email_service)
print("--- Skenario 1: Credit Card ---")
checkout_cc.run_checkout(andi_order)

# 2. Pembuktian OCP: Menambah Metode Pembayaran QRIS (Tanpa Mengubah CheckoutService)
class QrisProcessor(IPaymentProcessor):
    def process(self, order: Order) -> bool:
        print("Payment: Memproses QRIS.")
        return True

budi_order = Order("Budi", 100000)
qris_processor = QrisProcessor()

# Inject implementasi QRIS yang baru dibuat
checkout_qris = CheckoutService(payment_processor=qris_processor, notifier=email_service)
print("\n--- Skenario 2: Pembuktian OCP (QRIS) ---")
checkout_qris.run_checkout(budi_order)
```

OUTPUT:

```
PS C:\Users\Asus GK\OneDrive\Documents\PBO_Praktikum\Praktikum 11
>       > & C:/Python/Python312/python.exe "c:/Users/Asus GK/OneDr
ive/Documents/PBO_Praktikum/Praktikum 11/refactor_solid.py"
--- Skenario 1: Credit Card ---
Payment: Memproses Kartu Kredit.
Notif: Mengirim email konfirmasi ke Andi.
Checkout Sukses.

--- Skenario 2: Pembuktian OCP (QRIS) ---
Payment: Memproses QRIS.
Notif: Mengirim email konfirmasi ke Budi.
Checkout Sukses.
PS C:\Users\Asus GK\OneDrive\Documents\PBO_Praktikum\Praktikum 11
>
```

EMAIL : 2411102441205@umkt.ac.id

D. Latihan Mandiri

Studi Kasus – Sistem Pemrosesan Musik Digital

Tugas_Mandiri.py > ...

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass

@dataclass
class Song:
    title: str
    artist: str
    raw_data: str  #Simulasi data audio
    is_processed: bool = False

# === KODE BURUK (SEBELUM REFACTOR) ===
class MusicManager: # Melanggar SRP, OCP, DIP
    def process_and_upload(self, song: Song, effect: str):
        print(f"Mengolah lagu: {song.title} oleh {song.artist}")

        # LOGIKA EFEK (Pelanggaran OCP/DIP)
        if effect == "distorsi":
            print("Menerapkan efek Distorsi Gitar...")
        elif effect == "bass_boost":
            print("Meningkatkan frekuensi Bass...")
        else:
            print("Efek tidak didukung.")
            return False

        # LOGIKA PUBLIKASI (Pelanggaran SRP)
        print(f"Mengunggah '{song.title}' ke Spotify...")
        song.is_processed = True
        return True

# --- REFACTORING: ABSTRAKSI (Kontrak DIP/OCP) ---
class IAudioEffect(ABC):
    """Kontrak: Semua efek audio harus punya method 'apply' yaa."""
    @abstractmethod
    def apply(self, song: Song):
        pass

class IMusicPublisher(ABC):
    """Kontrak: Semua layanan publikasi harus punya method 'upload' yaak."""
    @abstractmethod
    def upload(self, song: Song):
        pass

# --- IMPLEMENTASI KONKRIT (SRP) ---
class BassBoostEffect(IAudioEffect):
    def apply(self, song: Song):
        print(f"Audio Effect: Meningkatkan Bass pada lagu '{song.title}'.")

class SpotifyPublisher(IMusicPublisher):
    def upload(self, song: Song):
        print(f"Publisher: Lagu '{song.title}' berhasil rilis di Spotify.")

# --- KELAS KOORDINATOR (SRP & DIP)
class MusicProductionSystem:
    def __init__(self, effect: IAudioEffect, publisher: IMusicPublisher):
        # Dependency Injection: Bergantung pada Abstraksi
        self.effect = effect
        self.publisher = publisher

    def produce(self, song: Song):
        self.effect.apply(song)
        song.is_processed = True
        self.publisher.upload(song)
        print("Produksi Musik Selesai.")

# --- PROGRAM UTAMA ---
if __name__ == "__main__":
    # 1. Setup Awal
    my_song = Song("Cranberries", "Indie Band", "audio_binary_data")

    # 2. Inject implementasi Bass Boost dan Spotify
    bass_effect = BassBoostEffect()
    spotify_pub = SpotifyPublisher()

    studio = MusicProductionSystem(effect=bass_effect, publisher=spotify_pub)

    print("--- Skenario 1: Produksi Musik Standar ---")
    studio.produce(my_song)
```

EMAIL : 2411102441205@umkt.ac.id

```
78
79      # 3. Pembuktian OCP: Menambah Efek REVERB tanpa mengubah MusicProductionSystem
80      class ReverbEffect(IAudioEffect):
81          def apply(self, song: Song):
82              print(f"Audio Effect: Menambahkan Reverb (Ruang) pada '{song.title}'.")
83
84      new_song = Song("WaterBoys", "Pop Band", "audio_binary_data")
85      reverb = ReverbEffect()
86
87      # Inject efek baru (OCP Terbukti)
88      studio_pro = MusicProductionSystem(effect=reverb, publisher=spotify_pub)
89      print("\n--- Skenario 2: Pembuktian OCP (Efek Reverb Baru) ---")
90      studio_pro.produce(new_song)
```

OUTPUT:

```
PS C:\Users\Asus GK\OneDrive\Documents\PBO_Praktikum\Pr
> & C:/Python/Python312/python.exe "c:/Users/Asus GK/On
eDrive/Documents/PBO_Praktikum/Praktikum 11/Tugas_Mandi
ri.py"
--- Skenario 1: Produksi Musik Standar ---
Audio Effect: Meningkatkan Bass pada lagu 'Cranberries'
.
Publisher: Lagu 'Cranberries' berhasil rilis di Spotify
.
Produksi Musik Selesai.

--- Skenario 2: Pembuktian OCP (Efek Reverb Baru) ---
Audio Effect: Menambahkan Reverb (Ruang) pada 'WaterBoy
s'.
Publisher: Lagu 'WaterBoys' berhasil rilis di Spotify.
Produksi Musik Selesai.
PS C:\Users\Asus GK\OneDrive\Documents\PBO_Praktikum\Pr
aktikum 11>
```

Refleksi SIngkat:

Dalam metode if/else, kelas koordinator dipaksa untuk mengetahui semua detail teknis dari setiap opsi yang ada, jadinya setiap kali ada fitur baru, kita harus membongkar kode lama yang sudah stabil (melanggar OCP). Dengan DI, kelas koordinator tidak perlu lagi memedulikan detail cara kerja sebuah fitur dia hanya cukup bergantung sama kontrak atau abstraksi universalnya.

EMAIL : 2411102441205@umkt.ac.id