OOP, 디자인 패턴, UML, 소프트웨어 공학 개념 정리

1. 객체지향 프로그래밍(OOP)

1.1 객체지향 개념

- **객체(Object)**: 상태(데이터)와 행위(메서드)를 가지는 독립적인 개체
- **클래스(Class)**: 객체를 생성하기 위한 청사진(설계도)
- **추상화(Abstraction)**: 불필요한 정보를 제거하고 본질적인 요소만 표현하는 기법
- **캡슐화(Encapsulation)**: 데이터를 보호하고 직접 접근을 제한하는 기법
- **상속(Inheritance)**: 기존 클래스를 재사용하여 새로운 클래스를 생성하는 기법
- **다형성(Polymorphism)**: 같은 인터페이스를 통해 여러 가지 동작을 수행하는 능력

1.2 객체지향의 장점

- 코드 재사용성 증가
- 유지보수 용이
- 확장성과 유연성 향상
- 보안성 증가 (캡슐화를 통한 접근 제어)

2. 디자인 패턴(Design Patterns)

2.1 디자인 패턴 개요

- 디자인 패턴은 소프트웨어 설계에서 자주 사용되는 문제 해결 방법을 정형화한 것
- **GoF(Gang of Four) 패턴**: 23가지 디자인 패턴을 체계적으로 정리함

2.2 디자인 패턴의 유형

1. 생성 패턴 (Creational Patterns)

1. **싱글톤(Singleton) 패턴**: 특정 클래스의 인스턴스를 하나만 생성하여 사용하는 패턴
2. **팩토리 메서드(Factory Method) 패턴**: 객체 생성을 서브클래스에서 담당하도록 하는 패턴
3. **추상 팩토리(Abstract Factory) 패턴**: 관련 객체들을 생성하는 인터페이스 제공
4. **빌더(Builder) 패턴**: 복잡한 객체 생성을 단계별로 수행
5. **프로토타입(Prototype) 패턴**: 기존 객체를 복제하여 새로운 객체를 생성

생성 패턴 코드 예제

1. **싱글톤(Singleton) 패턴**

```python
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance

singleton1 = Singleton()
singleton2 = Singleton()
print(singleton1 is singleton2)  # True
```

2. 팩토리 메서드(Factory Method) 패턴

```python
from abc import ABC, abstractmethod

class Product(ABC):
    @abstractmethod
    def operation(self):
        pass

class ConcreteProductA(Product):
    def operation(self):
        return "ConcreteProductA 생성"

class ConcreteProductB(Product):
    def operation(self):
        return "ConcreteProductB 생성"

class Creator(ABC):
    @abstractmethod
    def factory_method(self):
        pass

class ConcreteCreatorA(Creator):
    def factory_method(self):
        return ConcreteProductA()

class ConcreteCreatorB(Creator):
    def factory_method(self):
        return ConcreteProductB()

creator = ConcreteCreatorA()
product = creator.factory_method()
print(product.operation())  # "ConcreteProductA 생성"
```

3. 추상 팩토리(Abstract Factory) 패턴

```
class AbstractFactory:
    def create_product(self):
        pass

class ConcreteFactoryA(AbstractFactory):
    def create_product(self):
        return ConcreteProductA()

class ConcreteFactoryB(AbstractFactory):
    def create_product(self):
        return ConcreteProductB()

factory = ConcreteFactoryA()
product = factory.create_product()
print(product.operation())
```

4. 빌더(Builder) 패턴

```
class Product:
    def __init__(self):
        self.parts = []

    def add(self, part):
        self.parts.append(part)

    def show(self):
        return ", ".join(self.parts)

class Builder:
    def build_part(self):
        pass

class ConcreteBuilder(Builder):
    def __init__(self):
        self.product = Product()

    def build_part(self):
        self.product.add("Part A")

    def get_product(self):
        return self.product

builder = ConcreteBuilder()
builder.build_part()
product = builder.get_product()
print(product.show())
```

5. 프로토타입(Prototype) 패턴

```python
import copy

class Prototype:
    def clone(self):
        return copy.deepcopy(self)

class ConcretePrototype(Prototype):
    def __init__(self, value):
        self.value = value

prototype = ConcretePrototype("Hello")
clone = prototype.clone()
print(clone.value)  # "Hello"
```

2. 구조 패턴 (Structural Patterns)

1. **어댑터(Adapter) 패턴**: 인터페이스가 다른 클래스를 변환하여 호환성을 제공
2. **브리지(Bridge) 패턴**: 구현부와 추상화 부를 분리하여 독립적으로 확장 가능
3. **컴포지트(Composite) 패턴**: 트리 구조로 객체를 구성하여 계층 구조 표현
4. **데코레이터(Decorator) 패턴**: 동적으로 기능을 추가할 때 사용
5. **퍼사드(Facade) 패턴**: 복잡한 시스템을 단순한 인터페이스로 제공
6. **플라이웨이트(Flyweight) 패턴**: 메모리 사용을 최소화하기 위해 공유 객체 사용
7. **프록시(Proxy) 패턴**: 접근 제어 및 캐싱을 위해 대리 객체를 사용하는 패턴

구조 패턴 예제

1. **어댑터(Adapter) 패턴**

```python
class Target:
    def request(self):
        return "기본 동작"

class Adaptee:
    def specific_request(self):
        return "특수 동작"

class Adapter(Target):
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def request(self):
        return self.adaptee.specific_request()

adaptee = Adaptee()
adapter = Adapter(adaptee)
print(adapter.request())  # "특수 동작"
```

2. 데코레이터(Decorator) 패턴

```
class Component:
    def operation(self):
        return "기본 동작"

class Decorator(Component):
    def __init__(self, component):
        self.component = component

    def operation(self):
        return f"{self.component.operation()} + 추가 기능"

component = Component()
decorator = Decorator(component)
print(decorator.operation())  # "기본 동작 + 추가 기능"
```

3. 퍼사드(Facade) 패턴

```
class SubsystemA:
    def operation_a(self):
        return "Subsystem A 동작"

class SubsystemB:
    def operation_b(self):
        return "Subsystem B 동작"

class Facade:
    def __init__(self):
        self.subsystem_a = SubsystemA()
        self.subsystem_b = SubsystemB()

    def operation(self):
        return f"{self.subsystem_a.operation_a()} + {self.subsystem_b.operation_b()}"

facade = Facade()
print(facade.operation())  # "Subsystem A 동작 + Subsystem B 동작"
```

4. 프록시(Proxy) 패턴

```
class RealSubject:
    def request(self):
        return "RealSubject 동작"

class Proxy:
    def __init__(self):
        self.real_subject = RealSubject()
```

```
    def request(self):
        return f"Proxy: {self.real_subject.request()}"

proxy = Proxy()
print(proxy.request())  # "Proxy: RealSubject 동작"
```

5. 브리지(Bridge) 패턴

```
class Implementor:
    def operation(self):
        pass

class ConcreteImplementorA(Implementor):
    def operation(self):
        return "구현 A"

class ConcreteImplementorB(Implementor):
    def operation(self):
        return "구현 B"

class Abstraction:
    def __init__(self, implementor):
        self.implementor = implementor

    def operation(self):
        return self.implementor.operation()

bridge_a = Abstraction(ConcreteImplementorA())
print(bridge_a.operation())  # "구현 A"
```

6. 컴포지트(Composite) 패턴

```
class Component:
    def operation(self):
        pass

class Leaf(Component):
    def operation(self):
        return "Leaf 동작"

class Composite(Component):
    def __init__(self):
        self.children = []

    def add(self, component):
        self.children.append(component)
```

```
    def operation(self):
        results = [child.operation() for child in self.children]
        return " + ".join(results)

leaf1 = Leaf()
leaf2 = Leaf()
composite = Composite()
composite.add(leaf1)
composite.add(leaf2)
print(composite.operation())  # "Leaf 동작 + Leaf 동작"
```

7. **플라이웨이트(Flyweight) 패턴**

```
class Flyweight:
    def __init__(self, shared_state):
        self.shared_state = shared_state

    def operation(self, unique_state):
        return f"공유 상태: {self.shared_state}, 고유 상태: {unique_state}"

class FlyweightFactory:
    _flyweights = {}

    @staticmethod
    def get_flyweight(shared_state):
        if shared_state not in FlyweightFactory._flyweights:
            FlyweightFactory._flyweights[shared_state] = Flyweight(shared_state)
        return FlyweightFactory._flyweights[shared_state]

flyweight1 = FlyweightFactory.get_flyweight("State1")
print(flyweight1.operation("Unique1"))  # "공유 상태: State1, 고유 상태: Unique1"
```

3. 행위 패턴 (Behavioral Patterns)

1. **책임 연쇄(Chain of Responsibility) 패턴**: 요청을 여러 객체에 전달하면서 처리할 수 있도록 구성
2. **커맨드(Command) 패턴**: 실행될 기능을 캡슐화하여 호출자와 실행 로직을 분리
3. **인터프리터(Interpreter) 패턴**: 언어의 문법을 해석하여 실행하는 패턴
4. **이터레이터(Iterator) 패턴**: 컬렉션 요소들을 순차적으로 접근하는 방법 제공
5. **미디에이터(Mediator) 패턴**: 객체 간의 직접적인 통신을 방지하고 중재자를 통해 조정
6. **메멘토(Memento) 패턴**: 객체 상태를 저장하고 복원하는 기능 제공
7. **옵서버(Observer) 패턴**: 특정 객체의 상태 변화가 발생하면 연결된 객체들에게 자동으로 알림
8. **상태(State) 패턴**: 객체의 상태에 따라 동작을 변경하는 패턴
9. **전략(Strategy) 패턴**: 실행할 알고리즘을 동적으로 변경할 수 있도록 구성

10. **템플릿 메서드(Template Method) 패턴**: 알고리즘의 구조를 정의하고 일부 단계를 서브클래스에서 구현하도록 하는 패턴
11. **비지터(Visitor) 패턴**: 객체 구조를 변경하지 않고 새로운 연산을 추가하는 패턴

행위 패턴 예제

1. 책임 연쇄(Chain of Responsibility) 패턴

```
class Handler:
    def __init__(self, successor=None):
        self.successor = successor

    def handle_request(self, request):
        if self.successor:
            return self.successor.handle_request(request)
        return f"요청 {request} 처리 불가"

class ConcreteHandlerA(Handler):
    def handle_request(self, request):
        if request == "A":
            return "Handler A가 요청 처리"
        return super().handle_request(request)

class ConcreteHandlerB(Handler):
    def handle_request(self, request):
        if request == "B":
            return "Handler B가 요청 처리"
        return super().handle_request(request)

handler_chain = ConcreteHandlerA(ConcreteHandlerB())
print(handler_chain.handle_request("A"))  # "Handler A가 요청 처리"
print(handler_chain.handle_request("B"))  # "Handler B가 요청 처리"
print(handler_chain.handle_request("C"))  # "요청 C 처리 불가"
```

2. 커맨드(Command) 패턴

```
class Command:
    def execute(self):
        pass

class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        return self.light.turn_on()

class Light:
```

```
    def turn_on(self):
        return "전등 켜짐"

class RemoteControl:
    def __init__(self, command):
        self.command = command

    def press_button(self):
        return self.command.execute()

light = Light()
command = LightOnCommand(light)
remote = RemoteControl(command)
print(remote.press_button())  # "전등 켜짐"
```

3. 옵서버(Observer) 패턴

```
class Observer:
    def update(self, message):
        pass

class ConcreteObserver(Observer):
    def __init__(self, name):
        self.name = name

    def update(self, message):
        print(f"{self.name} received: {message}")

class Subject:
    def __init__(self):
        self.observers = []

    def attach(self, observer):
        self.observers.append(observer)

    def notify(self, message):
        for observer in self.observers:
            observer.update(message)

subject = Subject()
observer1 = ConcreteObserver("Observer1")
observer2 = ConcreteObserver("Observer2")
subject.attach(observer1)
subject.attach(observer2)
subject.notify("Hello, Observers!")
```

4. 상태(State) 패턴

```python
class State:
    def handle(self):
        pass

class ConcreteStateA(State):
    def handle(self):
        return "상태 A 동작"

class ConcreteStateB(State):
    def handle(self):
        return "상태 B 동작"

class Context:
    def __init__(self, state):
        self.state = state

    def set_state(self, state):
        self.state = state

    def request(self):
        return self.state.handle()

context = Context(ConcreteStateA())
print(context.request())  # "상태 A 동작"
context.set_state(ConcreteStateB())
print(context.request())  # "상태 B 동작"
```

5. 전략(Strategy) 패턴

```python
class Strategy:
    def execute(self):
        pass

class ConcreteStrategyA(Strategy):
    def execute(self):
        return "전략 A 실행"

class ConcreteStrategyB(Strategy):
    def execute(self):
        return "전략 B 실행"

class Context:
    def __init__(self, strategy):
        self.strategy = strategy

    def set_strategy(self, strategy):
        self.strategy = strategy
```

```
    def execute_strategy(self):
        return self.strategy.execute()

context = Context(ConcreteStrategyA())
print(context.execute_strategy())  # "전략 A 실행"
context.set_strategy(ConcreteStrategyB())
print(context.execute_strategy())  # "전략 B 실행"
```

3. UML (Unified Modeling Language)

3.1 UML 개요

- 소프트웨어 개발 과정에서 시스템을 시각적으로 표현하는 표준 모델링 언어
- 클래스 다이어그램, 시퀀스 다이어그램, 유스케이스 다이어그램 등 다양한 다이어그램 제공

3.2 주요 UML 다이어그램

1. **클래스 다이어그램(Class Diagram)**: 클래스 간의 관계를 표현
2. **시퀀스 다이어그램(Sequence Diagram)**: 객체 간의 상호작용을 시간 순서대로 표현
3. **유스케이스 다이어그램(Use Case Diagram)**: 사용자의 행위와 시스템 기능을 표현
4. **상태 다이어그램(State Diagram)**: 객체의 상태 변화 흐름을 표현
5. **액티비티 다이어그램(Activity Diagram)**: 작업의 흐름과 프로세스를 표현
6. **컴포넌트 다이어그램(Component Diagram)**: 소프트웨어의 구성 요소와 관계 표현
7. **배치 다이어그램(Deployment Diagram)**: 시스템의 물리적 배포 구조 표현

4. 소프트웨어 공학 개념

4.1 소프트웨어 개발 생명 주기 (SDLC)

1. **요구 분석(Requirement Analysis)**: 고객 요구사항 분석 및 명세화
2. **설계(Design)**: 시스템 구조 및 아키텍처 설계
3. **구현(Implementation)**: 소프트웨어 코드 개발
4. **테스트(Testing)**: 소프트웨어의 오류 및 품질 검증
5. **배포(Deployment)**: 실제 운영 환경에 소프트웨어 배포
6. **유지보수(Maintenance)**: 지속적인 개선 및 오류 수정

4.2 소프트웨어 개발 방법론

1. **폭포수 모델(Waterfall Model)**: 순차적 개발 방식
2. **애자일(Agile) 개발**: 반복적이고 유연한 개발 방식
3. **스파이럴 모델(Spiral Model)**: 위험 분석을 기반으로 한 반복 개발 방식
4. **V-모델(V-Model)**: 테스트와 개발을 동시에 진행하는 방식

4.3 소프트웨어 품질 특성

- **정확성(Correctness)**: 요구사항을 만족하는 정도
- **신뢰성(Reliability)**: 일정 시간 동안 정상적으로 동작하는 능력
- **유지보수성(Maintainability)**: 변경 및 수정이 용이한 정도
- **확장성(Scalability)**: 시스템이 확장될 수 있는 능력
- **재사용성(Reusability)**: 기존 코드를 재사용할 수 있는 정도