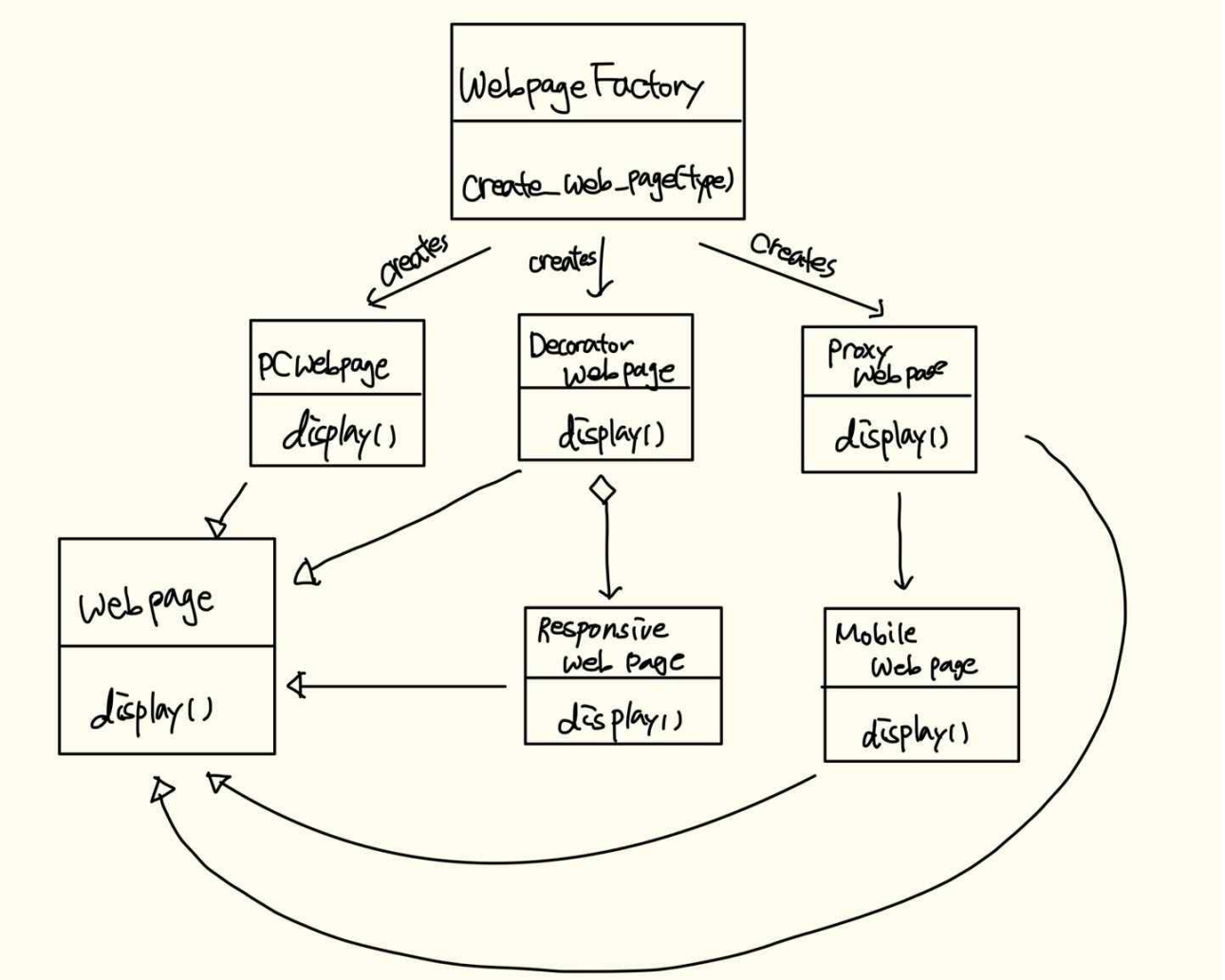
중간고사 대체 레포트

수 강 과 목 : 설계패턴
담 당 교 수 : 전병환 교수님
학 과 : 컴퓨터공학과
학 번 : 202001818
이 름 : 송강규
제 출 일 : 2024.05.03

먼저 제가 작성한 다이어그램은 다음과 같습니다 :



위 다이어그램을 토대로 작성한 코드는 다음과 같습니다:

```
from abc import ABC , abstractmethod
# Factory Pattern
class WebPageFactory :
  def create_web_page (self , type ):
   # 웹 페이지 유형에 따라 객체를 생성하는 팩토리 메서드입니다.
   if type =='PC':
    return PCWebPage ()
   elif type =='Responsive':
    return DecoratorWebPage (ResponsiveWebPage ())
   elif type =='Mobile':
    return ProxyWebPage (MobileWebPage ())
   else :
    raise ValueError ('Unknown Web Page Type')
# 기본 웹 페이지 인터페이스
```

```
class WebPage (ABC ):
  @abstractmethod
  def display (self ):
    # 모든 웹페이지는 display 메서드를 구현해야 합니다.
    pass
# 구체적인 웹 페이지 구현
class PCWebPage (WebPage ):
  def display (self ):
    print ("Displaying PC WebPage")
class ResponsiveWebPage (WebPage ):
  def display (self ):
    print ("Displaying Responsive WebPage")
class MobileWebPage (WebPage ):
  def display (self ):
    print ("Displaying Mobile WebPage")
# Decorator Pattern
class DecoratorWebPage (WebPage ):
  def __init__(self , web_page ):
    self ._web_page =web_page  # 다른 웹 페이지를 감싸는 데코레이터
  def display (self ):
    # 데코레이션 이전의 원래 기능을 호출
    self ._web_page .display()
    # 추가적인 기능 또는 스타일을 적용
    print ("Decorated WebPage")
# Proxy Pattern
class ProxyWebPage (WebPage ):
  def __init__(self , web_page ):
    self ._web_page =web_page  # 대리 실행할 웹 페이지
  def display (self ):
    # 접근 제어 또는 사전 처리
    print ("Proxy before access.")
    self ._web_page .display()
    # 필요시 사후 처리를 수행
# 클라이언트 코드
factory =WebPageFactory ()
web_page_type = ['PC', 'Responsive', 'Mobile']
for wt in web_page_type :
  web_page =factory .create_web_page (wt )
  print (f "--- {wt } WebPage ---")
  web_page .display ()
```

실행결과는 다음과 같습니다:

```
--- PC WebPage ---
Displaying PC WebPage
--- Responsive WebPage ---
Displaying Responsive WebPage
Decorated WebPage
--- Mobile WebPage ---
Proxy before access.
Displaying Mobile WebPage
```

제가 구상한 소프트웨어는 웹 페이지의 분류 및 관리를 위한 소프트웨어를 구현한 것 입니다.
이 코드는 다양한 타임의 웹 페이지 (PC, 반응형, 모바일)를 관리하고, 데코레이터 패턴과 프록시 패턴을 사용하여 웹 페이지에 추가 기능이나 스타일을 동적으로 적용하는 기능을 제공합니다.

제가 사용한 패턴은 팩토리 패턴, 데코레이터 패턴, 프록시 패턴으로 총 3개를 사용하였습니다. 팩토리 패턴이 적용되어야하는 이유는 웹페이지 유형에 따라 다른 객체를 생성해야 할 때, 클라이언트 코드가 구체적인 클래스에 의존하지 않도록 하기 위함입니다. 이를 통해 코드의 유연성을 높이고, 확장성을 개선하며 유지보수를 용이하게 할 수 있게 합니다. 웹 페이지에 대한 접근을 제어하거나, 접근 후에 추가 작업을 해야할 수도 있다고 생각해 웹페이지에 접근하기 전후에 추가적인 작업을 수행할 수 있는 프록시 패턴을 적용했습니다. 마지막으로 데코레이터 패턴을 적용한 이유는 실행 시간에 웹페이지에 특정 스타일을 적용하는 것 처럼 새로운 기능이나 책인을 동적으로 추가해야된다고 생각해 데코레이터 패턴을 사용하였습니다.

```
class WebPageFactory :
  def create_web_page (self , type ):
   # 웹 페이지 유형에 따라 객체를 생성하는 팩토리 메서드입니다.
   if type =='PC':
    return PCWebPage ()
   elif type =='Responsive':
    return DecoratorWebPage (ResponsiveWebPage ())
   elif type =='Mobile':
    return ProxyWebPage (MobileWebPage ())
   else :
    raise ValueError ('Unknown Web Page Type')
```

위의 WebPageFactory 클래스는 웹페이지 유형 (PC, Responsive, Mobile)을 입력 받고 해당 유형에 맞는 웹 페이지 객체를 생성하여 반환합니다. 이 팩토리 메서드를 통해 클라이언트가 구체적인 웹 페이지 클래스에 의존하지 않도록 해줍니다.

```
class ProxyWebPage (WebPage ):
  def __init__(self , web_page ):
   self ._web_page =web_page  # 대리 실행할 웹 페이지
  def display (self ):
   # 접근 제어 또는 사전 처리
   print ("Proxy before access.")
   self ._web_page .display()
   # 필요시 사후 처리를 수행
```

ProxyWebPage 클래스는 웹 페이지 객체를 감싸고, display 메서드를 호출할 때 추가적인 작업을 수행합니다. 이를 통해 원본 객체에 대한 접근을 제어할 수 있습니다

```
class DecoratorWebPage (WebPage ):
  def __init__(self , web_page ):
   self ._web_page =web_page  # 다른 웹 페이지를 감싸는 데코레이터
  def display (self ):
   # 데코레이션 이전의 원래 기능을 호출
   self ._web_page .display()
   # 추가적인 기능 또는 스타일을 적용
   print ("Decorated WebPage")
```

DecoratorWebPage 클래스는 다른 WebPage 객체를 감싸고, 원래 객체의 display 메서드를 호출한 후 추가적인 기능을 제공합니다. 이러한 방식을 통해 기존 객체의 기능을 수정하지 않고도 새로운 기능을 추가할 수 있습니다.

웹 페이지 분류 및 관리하는 소프트웨어를 설계하고 각각의 패턴을 적용하는 과정에서 제가 가장 많이 생각한 것은 소프트웨어가 유지 보수에 용이해야 되고, 쉽게 확장되어야 하며 클라이언트가 가질 수 있는 요구 사항과 변경 사항에 효과적으로 대응할 수 있었으면 좋겠다 라고 생각했습니다. 이를 통해 가장 먼저 코드에 유연성과 확장성을 부여할 수 있는 팩토리 패턴을 사용해야겠다고 생각했고, 프록시 패턴과 데코레이터 패턴을 통해 특정 웹 페이지에 대한 접근 제한과 추가적인 기능 확장이 있으면 견고한 소프트웨어가 될 것이라고 생각했습니다. 마지막으로 다양한 패턴을 사용하는 소프트웨어를 구상하면서 느낀 점은, 패턴을 배우기 전까진 클라이언트라는 개념이 크게 잡혀 있지 않았으므로 코드의 유연성과 기능의 추가 같은 중요한 부분들을 고려하지 않았습니다. 하지만 이러한 개념들과 패턴들을 배우고 코드에 적용하면서 앞으로 프로그램을 코딩해야 할 때 배운 패턴들을 통해 더욱 견고하고 품질을 높일 수 있는 프로그램을 만들 수 있게 되었다고 느꼈습니다.