

객체지향설계 Fall 2018



임성수 교수

Week 14: 주요 디자인 패턴 예제 및 마무리



수업 내용

1. 생성 패턴
2. 구조 패턴
3. 행위 패턴
4. 마무리

공지



향후 진행 계획

14주차 - 12/07 금	주요 디자인 패턴	
추석보충 - 12/10 월	기말고사	
15주차 - 12/17 월	프로젝트 최종 발표	최종 보고서 제출
15주차 - 12/21 금	종강	최종 성적 확인

■ 기말고사

- 일정: 12/10 월 09:00~10:15 / 11:00~12:15 (75분)
- 범위: Lecture 6 ~ Lecture 10 ('상속 및 구성'부터 포함)
- 평가: 25문제 x 4점 = 100점
- [+] 난이도: 중간고사보다 쉬움



향후 진행 계획

■ 프로젝트 최종 발표

- 일정: 12/17 월 09:00~11:00 / 11:00~13:00 (120분/12조)
- 시간: 발표 5~8분 (시간 엄수) + 질문/답변 2분
- 구성: 목적 및 달성 여부 / 주요 기능 및 구현 방법 / 중간 발표 후 수정 사항
- 제출: 12/17 월 08:00까지 슬라이드 메일로 제출 (시간 엄수)

■ 프로젝트 최종 보고서

- 기한: 12/17 월 23:59까지 제출
- 구성: 발표는 주요 내용 요약 / 보고서는 자세한 내용 설명
- 제출: 최종 보고서 (자율 양식) 및 결과물 메일로 제출

■ 최종 성적 확인

- 성적 게시: 12/21 금 09:00까지 (이러닝 공지)
- 성적 문의: 12/21 금 10:00~12:00 / 14:00~17:00 (교수 오피스)



프로젝트 최종 평가 기준

- 발표: 3점
 - 1. 발표: 시간을 지키고 목적 및 결과를 명료하게 설명
 - 2. 적합성: 객체지향 개발 절차에 맞춰 개발했음을 설명
 - 3. 완성도: 목표로 했던 기능 및 구현 결과 설명
 - —[가산점] 학생들이 본인 팀 제외 가장 우수한 팀 3팀 투표, 장점 1줄씩 작성 (미실시)
- 보고서: 3점
 - 1. 목적 및 계획: 문제의 필요성/해결 방안 설명
 - 2. 객체지향 개발: 계획/분석/설계/구현/테스트 과정 설명
 - 3. 구현: 목표로 했던 기능 및 구현 결과 설명
 - [가산점] 팀원 역할 분담 작성, 기여도에 따라 가산점 부여
 - [+] 프로젝트 추진 과정에서의 문제점 및 해결방안, 프로젝트를 통해 배우거나 느낀 점 작성
- 결과물: 4점
 - 1. 구현: 목표로 한 기능의 정상 동작 여부
 - 2. 코드: 각 기능 별 소스코드 평가 (주석 포함)
 - [가산점] GUI 등 부가 기능은 평가 기준에 미포함, 우수한 경우 가산점 부여

GoF 패턴 분류



디자인 패턴

- 자주 발생하는 설계 상의 문제를 해결하기 위한 반복적인 해법
- 코드가 더 견고해지고, 재사용이 용이하고, 설계에 대한 의사소통에 도움

GoF(Gang of Four) 디자인 패턴

- 대표적인 디자인 패턴 23가지
<http://www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf>
- 유형에 따른 분류
 - 생성 패턴: 5가지
 - 구조 패턴: 7가지
 - 행위 패턴: 11가지



1. 생성 패턴

생성 패턴

- 객체를 생성하는 절차를 추상화하는 패턴
- 객체를 생성하는 방법이나 객체의 표현 방법과 시스템을 분리
- 생성 패턴을 이용하면 객체 생성에 대한 유연성 제공

생성(Creational) 패턴	구조(Structural) 패턴	행위(Behavioral) 패턴
<ul style="list-style-type: none">• 추상 팩토리 (Abstract Factory)• 빌더 (Builder)• 팩토리 메서드 (Factory Method)• 프로토타입 (Prototype)• 싱글턴 (Singleton)	<ul style="list-style-type: none">• 어댑터 (Adapter)• 브리지 (Bridge)• 컴퍼지트 (Composite)• 데코레이터 (Decorator)• 퍼사드 (Facade)• 플라이웨이트 (Flyweight)• 프록시 (Proxy)	<ul style="list-style-type: none">• 책임 연쇄 (Chain of Responsibility)• 커맨드 (Command)• 인터프리터 (Interpreter)• 이터레이터 (Iterator)• 미디에이터 (Mediator)• 메멘토 (Memento)• 옵서버 (Observer)• 테이트 (State)• 스트래티지 (Strategy)• 템플릿 메서드 (Template Method)• 비지터 (Visitor)



1. 생성 패턴: 팩토리 메소드 패턴

팩토리 메소드 패턴

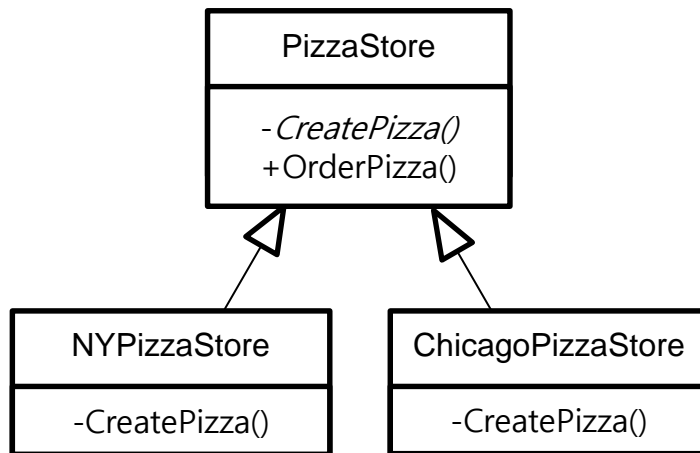
- 의도
 - 객체를 생성하기 위해 인터페이스를 정의
 - 어떤 클래스의 인스턴스를 생성할 것인지에 대한 결정은 파생 클래스가 결정
- 활용
 - 클래스가 자신이 생성해야 하는 객체의 해당 클래스를 예측하기 어려울 때
 - 생성할 객체를 기술하는 책임을 자신의 파생 클래스가 지정하길 바랄 때
 - 어떤 클래스가 객체 생성의 책임을 가지는지에 대한 정보를 줄이고 싶을 때
- 이점
 - 병렬적인 클래스 계통을 연결하는 역할을 담당



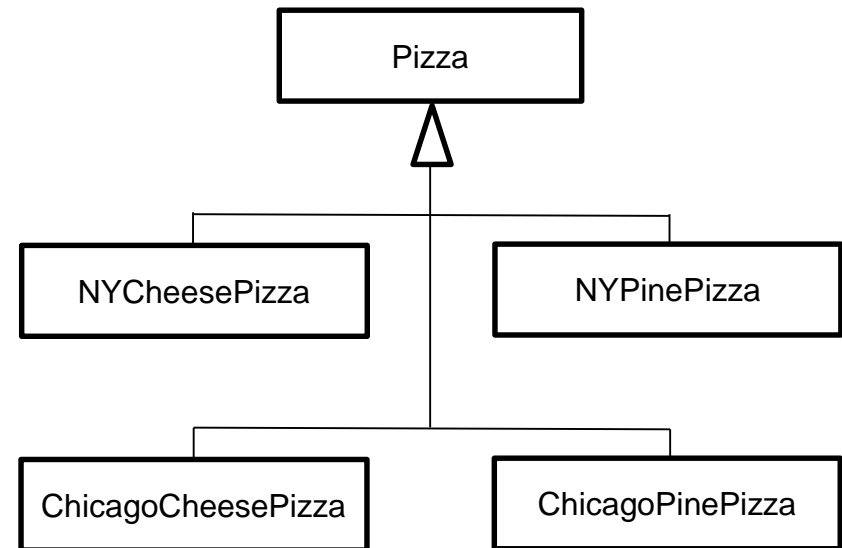
1. 생성 패턴: 팩토리 메소드 패턴

[예제] 팩토리 메소드 패턴

- 상품의 주문 과정은 중앙에서 이뤄지고, 가게에 따라 상품 생성이 달라지는 경우
- 기존 클래스의 OrderPizza()에서 종류를 결정,
파생 클래스에서 객체 생성 후 CreatePizza() 실행



Factory



Product



1. 생성 패턴: 팩토리 메소드 패턴

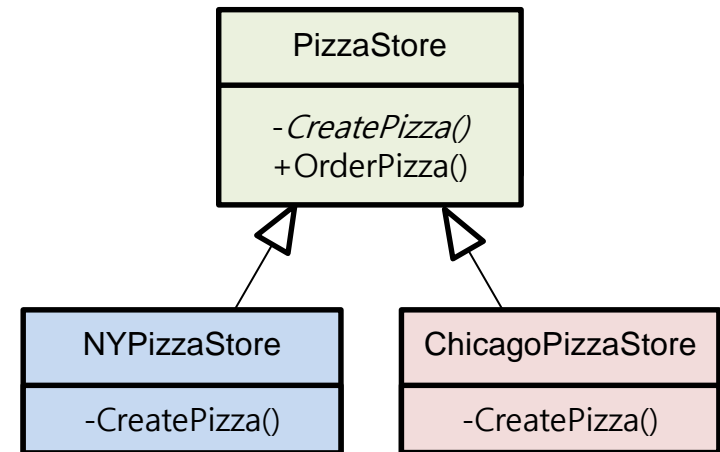
[예제] 팩토리 메소드 패턴

```
// PizzaStore.h
#ifndef PIZZASTORE
#define PIZZASTORE
class Pizza;

class PizzaStore {
private:
    // 피자 생성을 가상 함수로 정의
    virtual Pizza* CreatePizza(const char* pType);
public:
    Pizza* OrderPizza(const char* pType);
};

class NYPizzaStore : public PizzaStore {
private:
    Pizza* CreatePizza(const char* pType);
};

class ChicagoPizzaStore : public PizzaStore {
private:
    Pizza* CreatePizza(const char* pType);
};
#endif
```



Factory



1. 생성 패턴: 팩토리 메소드 패턴

[예제] 팩토리 메소드 패턴

```
// PizzaStore.cpp
#include "pch.h"
#include "Pizza.h"
#include "PizzaStore.h"
#include <iostream>
using namespace std;

// 피자 생성
Pizza* PizzaStore::CreatePizza(const char* pType) {
    Pizza* pizza = NULL;
    return pizza;
}

// 피자 주문
Pizza* PizzaStore::OrderPizza(const char* pType) {
    Pizza* pizza = NULL;
    pizza = CreatePizza(pType);
    return pizza;
}
```

```
// 뉴욕 피자 가게
Pizza* NYPizzaStore::CreatePizza(const char* pType) {
    Pizza* pizza = NULL;

    // 피자 종류
    if (!strcmp(pType, "치즈")) {
        pizza = new NYCheesePizza;
    }
    else if (!strcmp(pType, "파인애플")) {
        pizza = new NYPinePizza;
    }
    else {
        cout << "* 메뉴가 없습니다." << endl;
    }

    return pizza;
}

// 시카고 피자 가게: NY -> Chicago로 바꿔서 정의(생략)
```



1. 생성 패턴: 팩토리 메소드 패턴

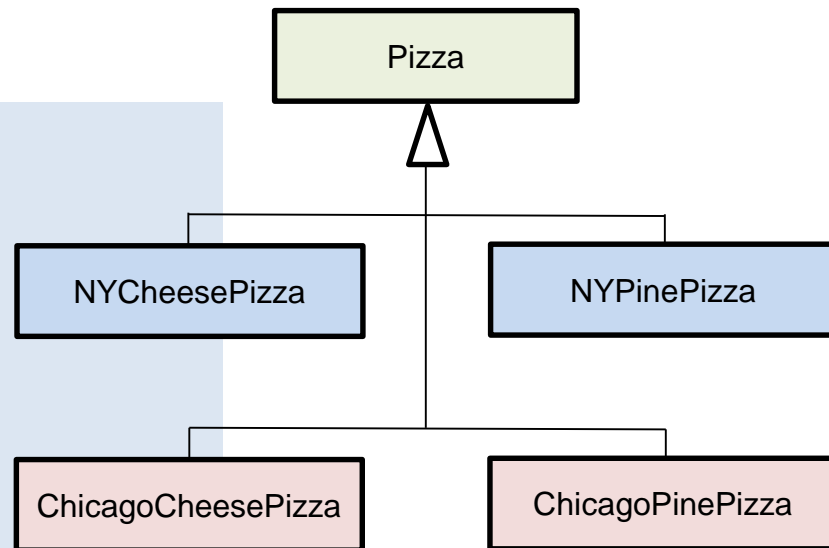
[예제] 팩토리 메소드 패턴

```
// Pizza.h
#ifndef PIZZA
#define PIZZA
#include <iostream>
using namespace std;
```

```
class Pizza {
public:
    virtual void PizzaName(void) {}
};
```

```
// 뉴욕 피자
class NYCheesePizza : public Pizza {
public:
    void PizzaName(void) { cout << "뉴욕 치즈 피자" << endl; }
};
class NYPinePizza : public Pizza {
public:
    void PizzaName(void) { cout << "뉴욕 파인애플 피자" << endl; }
};
```

```
// 시카고 피자 종류: NY -> Chicago로 바꿔서 정의(생략)
```



Product



1. 생성 패턴: 팩토리 메소드 패턴

[예제] 팩토리 메소드 패턴

```
// main.cpp
#include "pch.h"
#include "Pizza.h"
#include "PizzaStore.h"
#include <iostream>
using namespace std;

int main(void) {
    cout << "Factory Method Patten" << endl;

    // 뉴욕 피자 스토어 생성
    PizzaStore* NyPizza = new NYPizzaStore;

    // 생성된 객체에 대해 주문
    Pizza* pizza = NyPizza->OrderPizza("치즈");

    // 피자 이름 출력
    pizza -> PizzaName();

    return 0;
}
```

실행 결과

Factory Method Pattern
뉴욕 치즈 피자

2. 구조 패턴



구조 패턴

- 더 큰 구조를 형성하기 위한 작은 클래스들의 합성
- 상속을 통하여 여러 클래스에서 제공하는 인터페이스를 통합 제공
- 사용자의 편의성과 유연성 높임

생성(Creational) 패턴	구조(Structural) 패턴	행위(Behavioral) 패턴
<ul style="list-style-type: none">• 추상 팩토리 (Abstract Factory)• 빌더 (Builder)• 팩토리 메서드 (Factory Method)• 프로토타입 (Prototype)• 싱글턴 (Singleton)	<ul style="list-style-type: none">• 어댑터 (Adapter)• 브리지 (Bridge)• 컴퍼지트 (Composite)• 데코레이터 (Decorator)• 퍼사드 (Facade)• 플라이웨이트 (Flyweight)• 프록시 (Proxy)	<ul style="list-style-type: none">• 책임 연쇄 (Chain of Responsibility)• 커맨드 (Command)• 인터프리터 (Interpreter)• 이터레이터 (Iterator)• 미디에이터 (Mediator)• 메멘토 (Memento)• 옵서버 (Observer)• 테이트 (State)• 스트래티지 (Strategy)• 템플릿 메서드 (Template Method)• 비지터 (Visitor)



2. 구조 패턴: 컴포지트 패턴

컴포지트 패턴

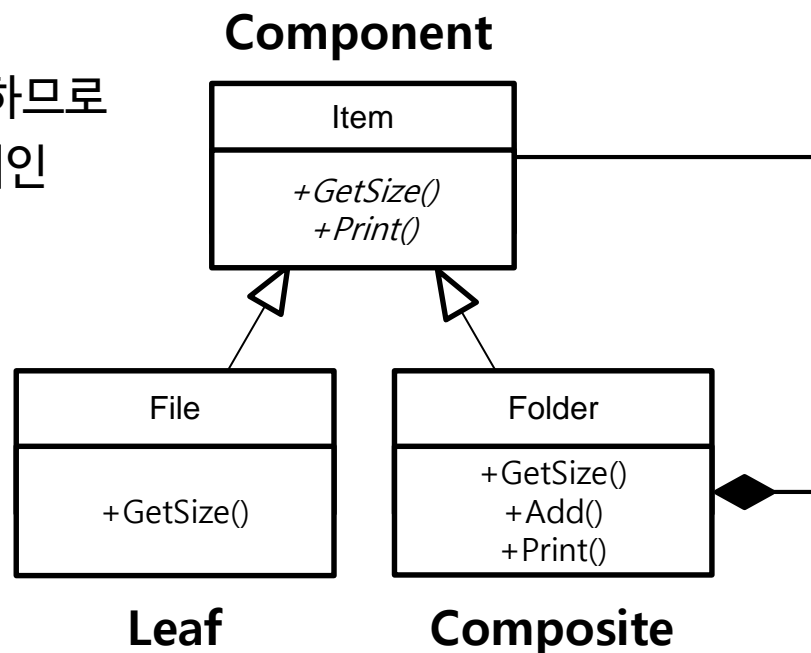
- 의도
 - 부분과 전체의 계층을 표현하기 위해 객체들을 모아서 구성
- 활용
 - 부분-전체의 객체 계통 표현
 - 사용자가 객체의 구성으로 생긴 포함 객체(embedded object)와 개개의 객체의 차이를 알지 않고도 모든 객체를 똑같이 취급 가능
- 이점
 - 사용자 코드의 단순화
 - 새로운 종류의 구성요소 추가 쉬움



2. 구조 패턴: 컴포지트 패턴

[예제] 컴포지트 패턴

- 객체들을 트리 구조로 표현 가능, 폴더에 속한 파일 가능, 파일에 속한 폴더/파일 불가능
- Component: 모든 객체에 대한 인터페이스 정의
- Leaf: 자식이 없는 개별 객체
- Composite: 부분-전체 구조를 다루어야 하므로 포함 객체뿐 아니라 개별 객체인 Leaf와 관련된 기능도 구현



2. 구조 패턴: 컴포지트 패턴



[예제] 컴포지트 패턴

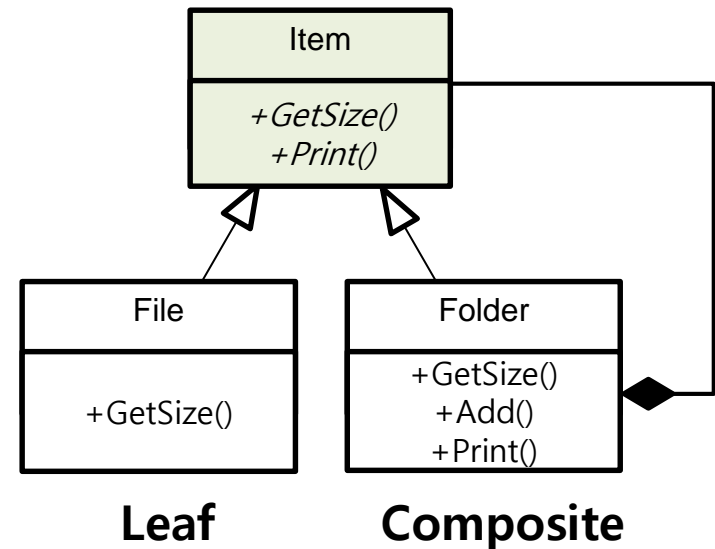
```
#include "pch.h"
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// Item: Base 클래스
class Item {
    string name;
public:
    Item(string s) : name(s) {}

    // 파일, 폴더의 이름을 반환하는 함수
    string GetName() const { return name; }

    // 파일, 폴더의 사이즈
    virtual int GetSize() = 0;

    virtual void print(string prefix) const {
        cout << prefix << GetName() << endl;
    }
};
```



2. 구조 패턴: 컴포지트 패턴



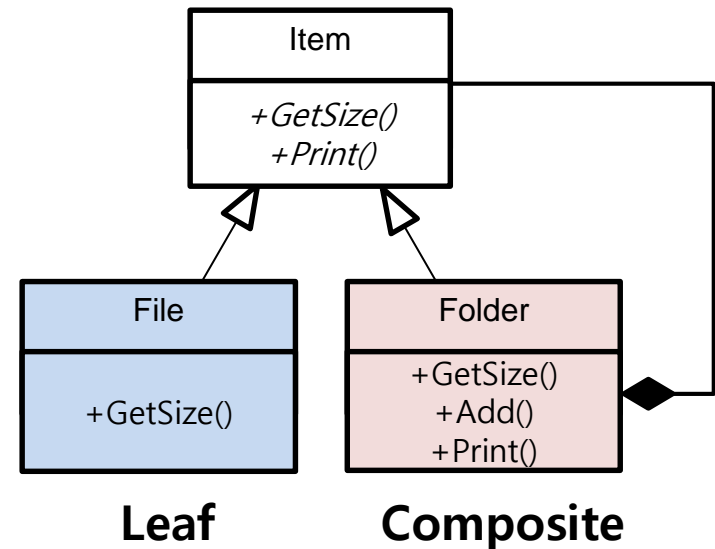
[예제] 컴포지트 패턴

```
class Folder : public Item {
    vector<Item*> items;
public:
    Folder(string n) : Item(n) {}
    void Add(Item* p) { items.push_back(p); }

    virtual int GetSize() {
        int sz = 0;
        int items_size = items.size();
        for (int i = 0; i < items_size; ++i) {
            sz += items[i]->GetSize();
        }
        return sz;
    }

    virtual void print(string prefix) const {
        cout << prefix << GetName() << endl;
        prefix = prefix + string("Wt");
        int items_size = items.size();
        for (int i = 0; i < items_size; ++i) {
            items[i]->print(prefix);
        }
    }
};
```

```
class File : public Item {
    int size;
public:
    File(string n, int s) : Item(n), size(s) {}
    virtual int GetSize() { return size; }
};
```



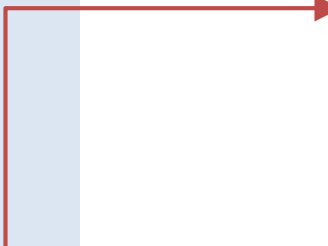


2. 구조 패턴: 컴포지트 패턴

[예제] 컴포지트 패턴

```
void main() {  
    Folder* R = new Folder("ROOT");  
    Folder* A = new Folder("A");  
    Folder* B = new Folder("B");  
  
    R->Add(A);  
    R->Add(B);  
  
    R->Add(new File("a.txt", 10));  
    A->Add(new File("b.txt", 20));  
    B->Add(new File("c.txt", 30));  
    cout << R->GetSize() << endl;  
  
    R->print("");  
}
```

실행 결과



```
60  
ROOT  
    A  
        b.txt  
    B  
        c.txt  
    a.txt
```

3. 행위 패턴



행위 패턴

- 반복적으로 일어나는 객체들의 상호 작용을 패턴화한 것
- 객체들 간의 알고리즘이나 역할 분담에 관련된 것
- 객체 간의 제어 구조보다는 객체들을 어떻게 연결시킬지 관심

생성(Creational) 패턴	구조(Structural) 패턴	행위(Behavioral) 패턴
<ul style="list-style-type: none">• 추상 팩토리 (Abstract Factory)• 빌더 (Builder)• 팩토리 메서드 (Factory Method)• 프로토타입 (Prototype)• 싱글턴 (Singleton)	<ul style="list-style-type: none">• 어댑터 (Adapter)• 브리지 (Bridge)• 컴퍼지트 (Composite)• 데코레이터 (Decorator)• 퍼사드 (Facade)• 플라이웨이트 (Flyweight)• 프록시 (Proxy)	<ul style="list-style-type: none">• 책임 연쇄 (Chain of Responsibility)• 커맨드 (Command)• 인터프리터 (Interpreter)• 이터레이터 (Iterator)• 미디에이터 (Mediator)• 메멘토 (Memento)• 옵서버 (Observer)• 테이트 (State)• 스트래티지 (Strategy)• 템플릿 메서드 (Template Method)• 비지터 (Visitor)



3. 행위 패턴: 스테이트 패턴

스테이트 패턴

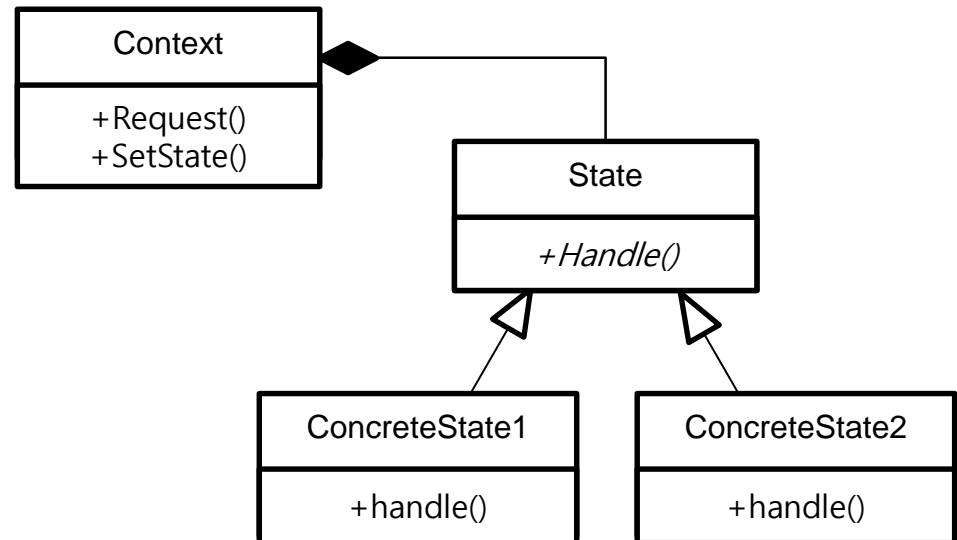
- 의도
 - 객체의 내부 상태에 따라 스스로 행동을 변경할 수 있게 허가
- 활용
 - 동일한 요청이라도 객체의 상태에 따라 행동을 다르게 처리
 - 객체의 상태를 별도로 객체로 정의하여 다양화
- 이점
 - 상태에 따른 행동을 국소화하여 서로 다른 상태에 대한 행동을 객체로 관리
 - 상태 변경을 명확하게 함

3. 행위 패턴: 스테이트 패턴



[예제] 스테이트 패턴

- 상태를 별도의 객체로 관리하여 조건에 따라 결과를 다르게 함 (예: 자판기 등)
- Context: 클라이언트가 필요한 기능(인터페이스) 제공, State에 요청 처리 위임
- State: 모든 상태에 공통된 기능 처리를 정의
- Concrete: 특정 상태의 기능 처리 구현





3. 행위 패턴: 스테이트 패턴

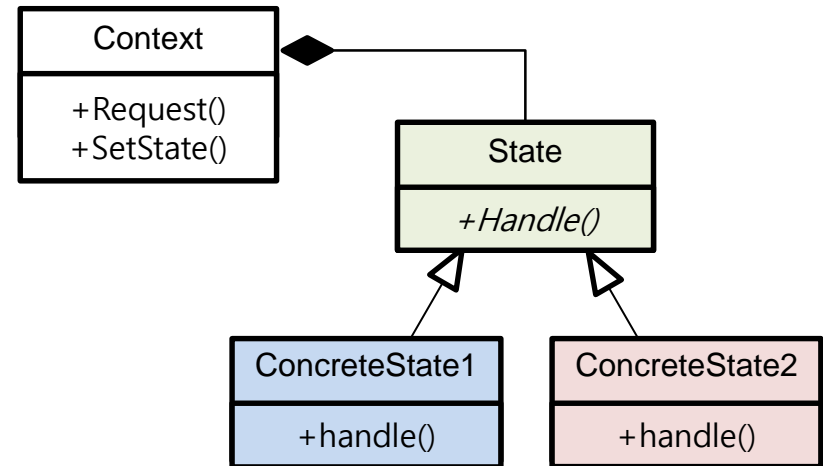
[예제] 스테이트 패턴

```
#include "pch.h"
#include <iostream>
using namespace std;

class State {
public:
    virtual void handle() {};
};

class ConcreteState1 : public State {
public:
    void handle() {
        cout << "콘크리트 1" << endl;
    }
};

class ConcreteState2 : public State {
public:
    void handle() {
        cout << "콘크리트 2" << endl;
    }
};
```



3. 행위 패턴: 스테이트 패턴

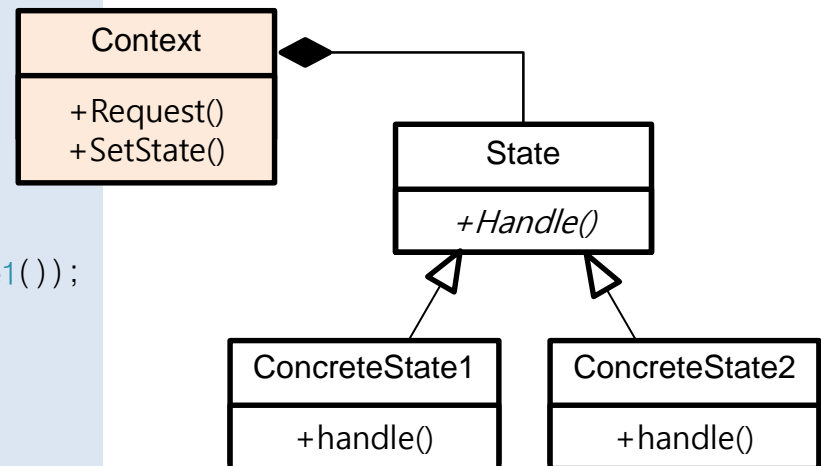


[예제] 스테이트 패턴

```
class Context {  
public:  
    Context(State *state):pState(state) {}  
    void SetState(State *state) {  
        if (NULL != pState) {  
            delete pState;  
            pState = state;  
        }  
    }  
    void Request() {  
        pState->handle();  
    }  
private:  
    State *pState;  
};  
  
int main() {  
    Context *pContext = new Context(new ConcreteState1());  
    pContext->Request();  
    pContext->SetState(new ConcreteState2());  
    pContext->Request();  
    delete pContext;  
    return 0;  
}
```

실행 결과

콘크리트1
콘크리트2



4. 마무리



Lecture 6: Inheritance & Composition (~5문제)

- 상속과 구성: 상속과 구성의 구분
- 생성자: 상속에서의 생성자 표현
- 이름 은닉: 파생 클래스에서 멤버 함수 재정의 가능
- 접근 제어: public / private / protected
- 하위 타입: subtyping, 파생 클래스를 하위 타입으로 활용
- 업캐스팅: upcasting, 파생 클래스에서 기존 클래스로의 형 변환
- 복사 생성자: 객체의 복사가 이뤄질 때 호출되는 생성자

4. 마무리



Lecture 7: Polymorphism & Virtual Functions (~9문제)

- 상속과 포인터/참조자: 상속에서 포인터/참조자의 접근 권한
- 동적 바인딩: 오버라이딩, 가상 함수 사용, 정적/동적 바인딩
- VTABLE: 가상 함수의 동작 원리: 테이블과 포인터 구조
- 순수 가상 함수: 순수 가상 함수를 쓰는 이유, 추상 클래스
- 가상 소멸자: 파생 클래스 소멸자 호출 방식

4. 마무리



Lecture 8: Introduction to Templates (~4문제)

- 함수 템플릿: 템플릿의 정의, 템플릿 함수의 활용 예
- 클래스 템플릿: 템플릿 클래스의 활용 예

Lecture 9~10: Design Patterns (~7문제)

- 디자인 패턴 소개: 디자인 패턴의 역사, 의미, 장점 등
- 디자인 패턴 분류: Lecture 9의 14-18쪽은 객관식 1문제, GoF 유형 구분
- 아키텍처 패턴: Lecture 9의 21-22쪽에 대해 1~2문제, 이후는 생략
- 생성, 구조, 행위 패턴의 예: 싱글턴 패턴, 오늘 수업의 세 가지 패턴 예제

질문 및 답변

