객체지향설계
Fall 2018

임성수 교수

Week 13: 디자인 패턴 소개

수업 내용

1. 디자인 패턴 소개
2. 디자인 패턴 분류
3. 아키텍처 패턴

공지

향후 진행 계획

13주차 - 11/30 금	**디자인 패턴 소개**	
14주차 - 12/03 월	실습	
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

공지

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)

공지

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
- 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
- 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

1. 디자인 패턴 소개

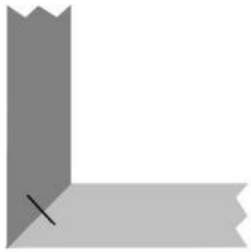
시스템 설계의 재사용

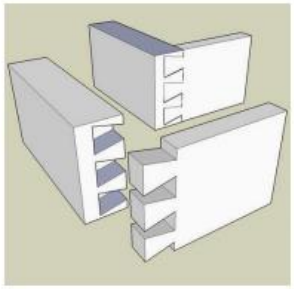
- 다형성: 프로그램 코드의 재사용
- **디자인 패턴**: 시스템 설계 측면에서의 재사용
  - 경험 많은 소프트웨어 엔지니어는 많은 시스템 설계 경험과 해법 보유
- 디자인 패턴의 장점
  - 해법을 모아 **디자인적인 추상성을 주어 패턴화**한다면,
    하나의 시스템 설계 시 공통 언어의 역할 가능
  - **공통의 언어**를 통해 효율적으로 협동 작업을 할 수 있고,
    해법을 익혀 시스템 안정성, 성능을 높일 수 있음
  - 예: ~한 성질을 만족하는 ~를 사용하자 vs. ~패턴을 사용하자

1. 디자인 패턴 소개

패턴 언어 (Pattern Language)

- 1970년대 Christopher Alexander에 의해 연구된 건축 설계 패턴
  - 좋은 건축이라는 것은 객관적인가?
  - 설계가 좋다는 것을 무엇으로 알 수 있나?
  - 좋은 설계에는 있지만 나쁜 설계에 없는 것은?
  - 좋은 설계 또는 나쁜 설계가 되게 하는 요소는?
- 인간이 환경을 **설계하는 데 항상 일정한 언어가 필요**하다는 패러다임
  - 예: 벽돌집의 구조를 어떻게 하면 좋을까?
  - 예: 서랍의 이음새를 어떻게 하면 좋을까?
- 디자인 패턴에 지대한 영향을 끼침





1. 디자인 패턴 소개

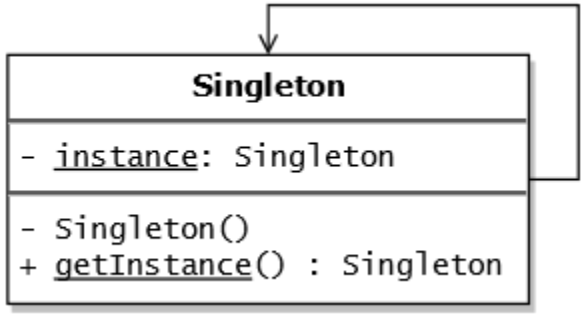
디자인 패턴의 역사

- "Using Pattern Language for OOP," OOPSLA 1987
  - 패턴 언어의 아이디어를 통해 사용자 인터페이스(UI)에 대한 다섯 가지 패턴 제시
  - Window Per Task, Few Panes Per Window, Standard Panes, Short Menus, Nouns and Verbs
- "Design Patterns: Elements of Reusable OO Software," 1995
  - **GoF**(Gang of Four)로 불림
  - **23개의 주요 디자인 패턴** 수록
  - 디자인 패턴의 아이디어를 널리 알린 계기
- 현재까지 수많은 종류의 디자인 패턴이 발표됨
- 패턴의 홍수로 사용이 힘들다는 의견도 있음

1. 디자인 패턴 소개

[예제] 싱글턴 패턴 (Singleton-)

- 타입: 생성 패턴
- 용도
  - 시스템 내부에 1개의 객체만 생성하고 싶은 경우
  - 예: 컴퓨터 자체를 표현한 클래스
  - 예: 시스템 로그 파일
- 사용 방법
  - 생성자를 private으로 선언
  - 정적 멤버 변수, 정적 멤버 함수의 활용
  - 외부에서는 하나의 객체만을 참조 (같은 주소)



1. 디자인 패턴 소개

[예제] 싱글턴 패턴 (Singleton-)

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

class Singleton {
private:
  // Make the constructor private
  Singleton() {}
  // Copy constructor private
  Singleton(const Singleton & obj) {}
  static Singleton* s_instance;
public:
  static Singleton* GetInstance() {
    if (NULL == s_instance){
      s_instance = new Singleton();
    }
    return s_instance;
  }
};
```

생성자 private 선언: 외부에서의 객체 생성 방지

복사 생성자 private 선언: 외부에서의 객체 생성 방지

정적 멤버 변수 선언:
클래스에 속하지만, 객체 별로 할당하지 않고
클래스의 모든 객체가 공유하는 멤버

하나의 객체만 반환하는 정적 멤버 함수:
해당 클래스 객체를 생성하지 않고도
클래스 이름만으로 호출 가능
→ 이 함수를 통해서만 객체를 생성 가능

1. 디자인 패턴 소개

[예제] 싱글턴 패턴 (Singleton-)

```
// Initialize static singleton pointer
Singleton *Singleton::s_instance = 0;

int main() {
  Singleton* s_obj1 = Singleton::GetInstance();
  cout << "Address of object is:" << s_obj1 << endl;

  Singleton* s_obj2 = Singleton::GetInstance();
  cout << "Address of object is:" << s_obj2 << endl;

  return 0;
}
```

정적 멤버 변수 초기화:
클래스 외부에서 초기화

최초 GetInstance가 호출되면 s_instance가 null이므로 new에 의해 객체 생성

다시 GetInstance가 호출되면, s_instanc가 null이 아님
이미 만들어진 싱글턴 객체 s_instance 반환

객체들의 주소가 같음:
동일한 객체임을 알 수 있음

실행 결과

```
Address of object is: 005FF9E8
Address of object is: 005FF9E8
```

1. 디자인 패턴 소개

디자인 패턴 명세

- 패턴은 누구나 배울 수 있도록 아래와 같은 형식으로 정리
  - **이름**: 패턴은 이름과 타입(생성/구조/행위)을 가짐
  - **배경, 문제**: 패턴이 **적용되는 상황 및 다루는 문제** 설명
  - **솔루션**: 패턴의 구조적 설계나 행위적 설계 기술
    클래스/시퀀스 다이어그램으로 패턴에 참여하는 **클래스의 역할/관계** 표현
  - **혜택과 책임**: 패턴을 적용함으로서 얻는 이점과 잠재적 문제점 기술
  - **가이드라인**: 패턴을 적용할 때 유용한 정보 제공
  - **관련 패턴**: 유사/연관 패턴을 적용하여 변형/확장 시 참고

2. 디자인 패턴 분류

디자인 패턴 분류

- 기본 패턴: 별도 패턴으로 분류되지 않는 객체지향 패턴의 관용구
  - 개념 실체 패턴
  - 플레이어 역할 패턴
  - 위임 패턴
  - 계층 구조 패턴
- GoF 디자인 패턴
  - 생성 패턴 (Creational-)
  - 구조 패턴 (Structural-)
  - 행위 패턴 (Behavioral-)

} 다음 시간에 주요 디자인 패턴 및 예제 소개

2. 디자인 패턴 분류

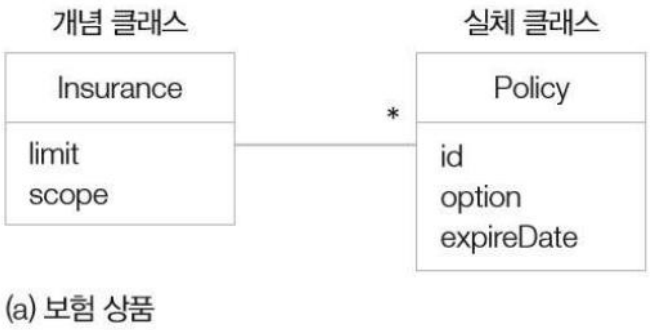
기본 패턴 분류

- 개념 실체 패턴
  - 각 객체의 공통 정보를 공유할 때 사용
- 플레이어 역할 패턴
  - 하나의 클래스에 다양한 역할을 표현하는 패턴
- 위임 패턴
  - 다른 클래스가 가진 특정 오퍼레이션을 활용하여 책임을 위임하는 패턴
- 계층 구조 패턴
  - 계층 관계를 가진 객체들을 묶어 동일한 처리를 하고 싶을 때 사용

2. 디자인 패턴 분류

개념 실체 패턴 (Abstraction Occurrence-)

- 개념: 공유하는 정보를 담은 클래스 (Insurance, BookTitle)
- 실체: 공통된 정보를 가진 멤버들의 모임 (Policy, LibraryItem)
- 중복되는 정보를 저장하지 않게 함
- 하나의 클래스에 모두 저장하거나 상속을 이용하는 것보다 적합



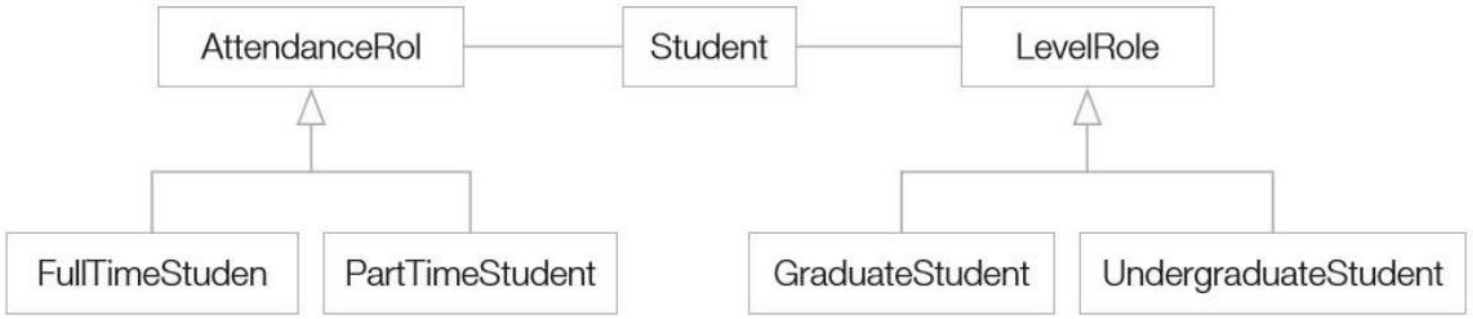
(a) 보험 상품

(b) 도서 목록

2. 디자인 패턴 분류

플레이어 역할 패턴 (Player Role-)

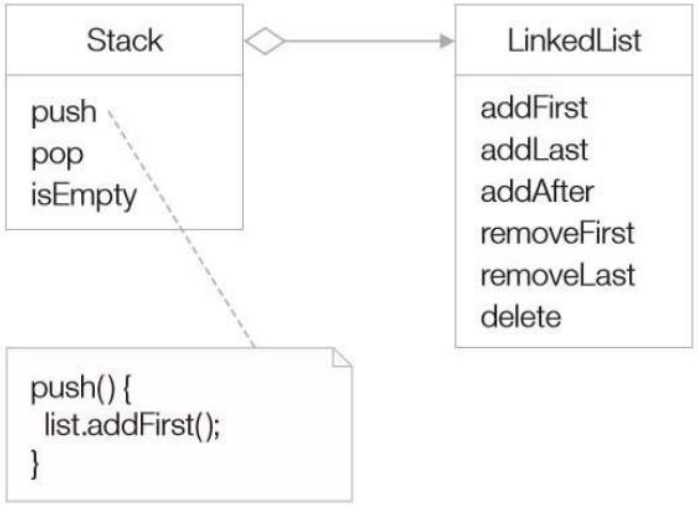
- 플레이어가 환경에 따라 다른 역할을 해야 할 때
- 학생은 등록(전일제/파트타임), 과정(대학원/학부) 등 역할 구분
- 각 역할의 상위클래스와 연관을 맺어서 다른 상태를 가질 수 있게 함



2. 디자인 패턴 분류

위임 패턴 (Delegation-)

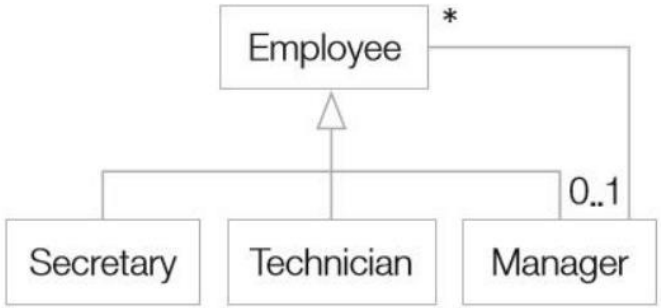
- 다른 클래스의 오퍼레이션에 작업을 요청
- 클래스 안에서 해결하지 못하는 일은 다른 클래스의 오퍼레이션에 위임
- 상속보다 연관을 통해 효과적으로 재사용 (근접한 정보에만 접근)

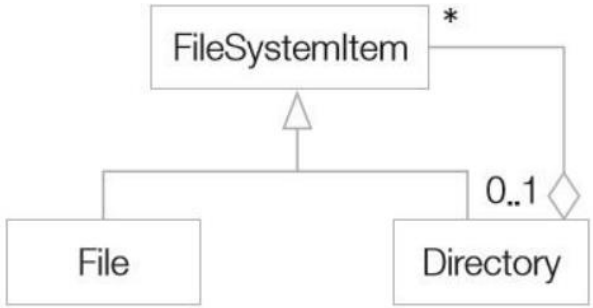


2. 디자인 패턴 분류

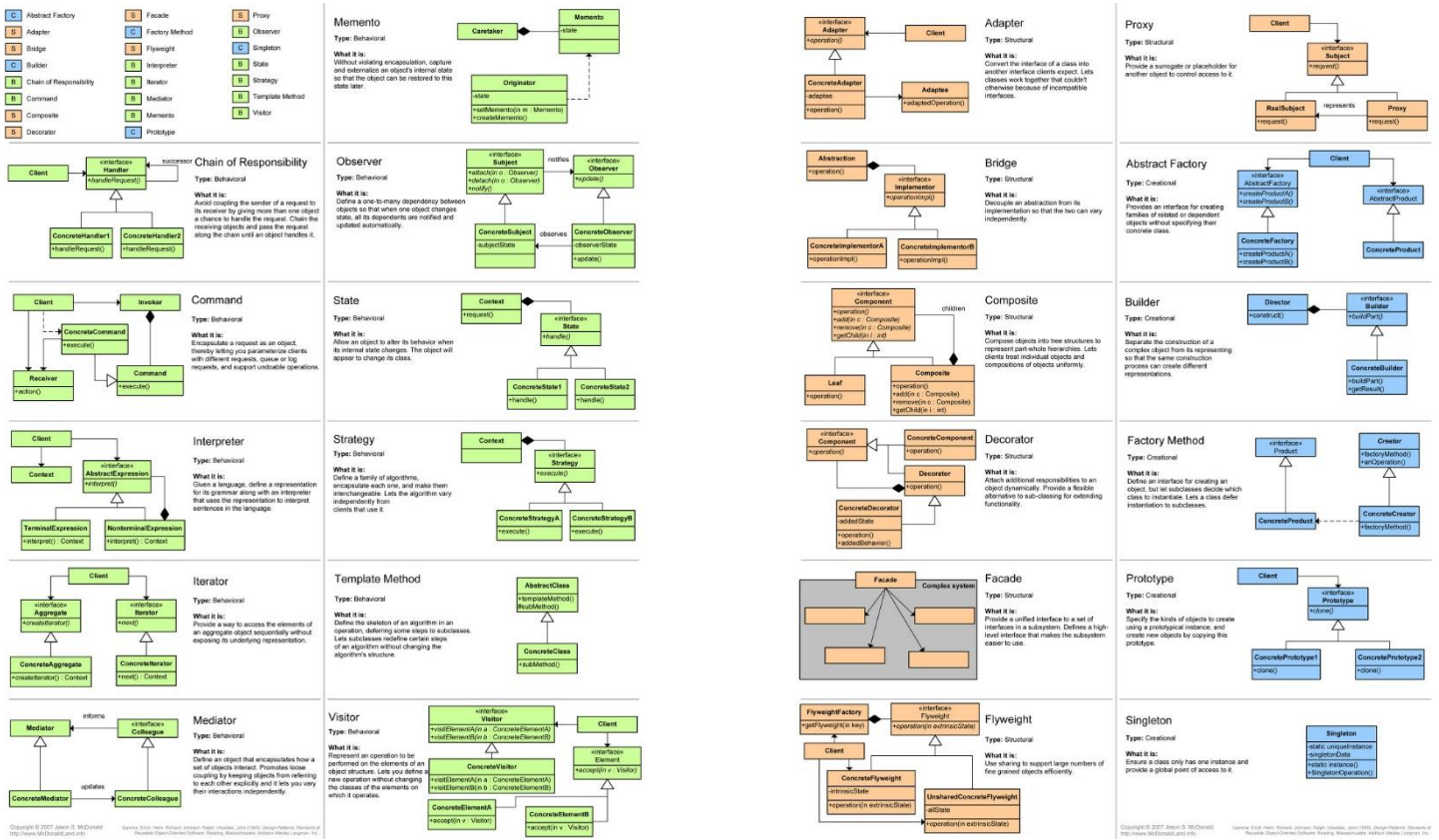
계층 구조 패턴

- 회사의 조직도, 파일 구조, 구문 트리 등 계층 구조를 다룰 때 필요
- 파생된 객체가 기존 클래스와 재귀적 연관 관계 맺기 가능
- 재귀적인 연관 관계를 통한 반복적인 계층 표현 가능





2. 디자인 패턴 분류



GoF 디자인 패턴

2. 디자인 패턴 분류

GoF 디자인 패턴

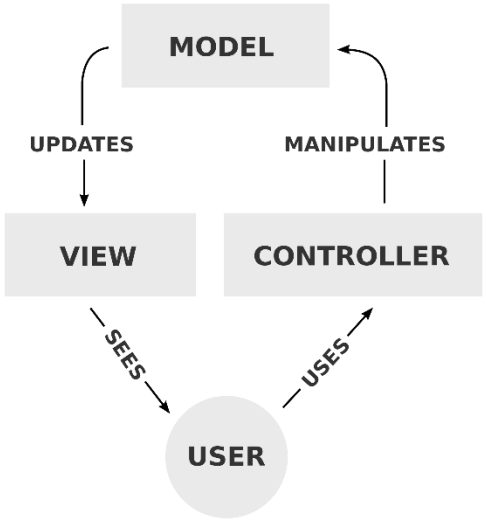
- **생성 패턴**
  - 간단한 블록의 코드로 여러 가지 **다양한 객체를 생성**
  - 여러 가지 객체의 집합을 포함한 어플리케이션을 설계하는데 유용
- **구조패턴**
  - 더 큰 구조를 형성하기 위해 클래스를 **어떻게 구성하고 합성하는지**를 다룸
  - 주로 인터페이스를 상속받아 구현하여 합성
- **행위패턴**
  - 반복적으로 일어나는 **객체들의 상호작용**을 패턴화
  - 객체들 간의 알고리즘이나 역할 분담과 관련

3. 아키텍처 패턴

아키텍처 패턴

- 소프트웨어 개발에 있어서 반복되는 문제들에 대한 해법을 문서화
- **GoF 패턴은 설계**에 중점, **아키텍처 패턴은 시스템 구축**에 중점
- 아키텍처 패턴의 활용
  - 시스템 전반에 관한 문제 해결에 관심
  - 큰 건물처럼 큰 시스템을 구축할 때는 아키텍처가 필요
  - 아키텍처 패턴의 구성은 디자인 패턴들로 이뤄지는 경우가 많음
  - **MVC 패턴**: 하나의 시스템 전체를 Model, View, Controller 세 개의 컴포넌트로 나눠서 각 부분의 변경 영향도를 최소화하기 위한 패턴

3. 아키텍처 패턴



MVC 패턴 (Model-View-Controller)

- 프로그램을 구성하는 요소들을 나누어서 각 컴퍼넌트의 역할을 수행
- Model
  - 데이터, 상태 및 어플리케이션 로직을 포함
  - 특정 출력 표현 방식, 특정 입력 동작의 영향을 받지 않음
- View
  - 모델로부터 제공된 데이터를 표현하는 방법을 제공하는 사용자 인터페이스
- Controller
  - 뷰와 모델 사이에서, 사용자의 입력을 받아서 모델에게 어떤 의미가 있는지 파악

3. 아키텍처 패턴

[예제] 아키텍처 패턴과 디자인 패턴 (skip)

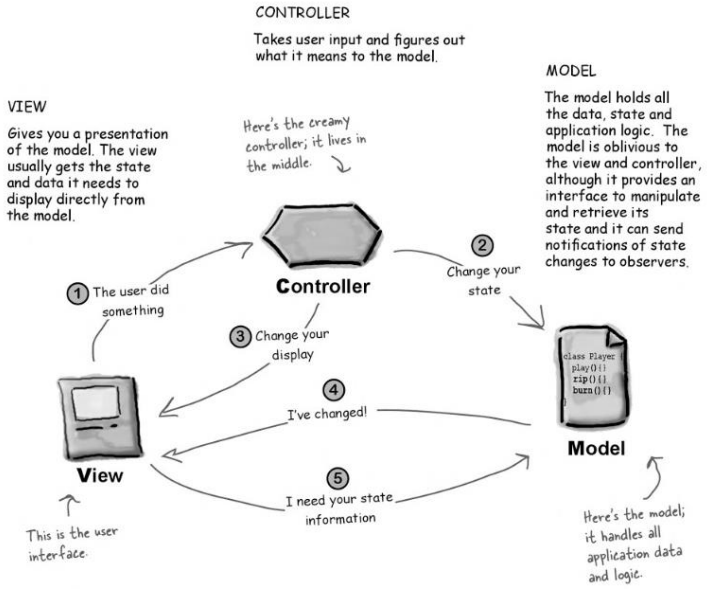
- 출처: Head First Design Patterns, 마이크로소프트웨어

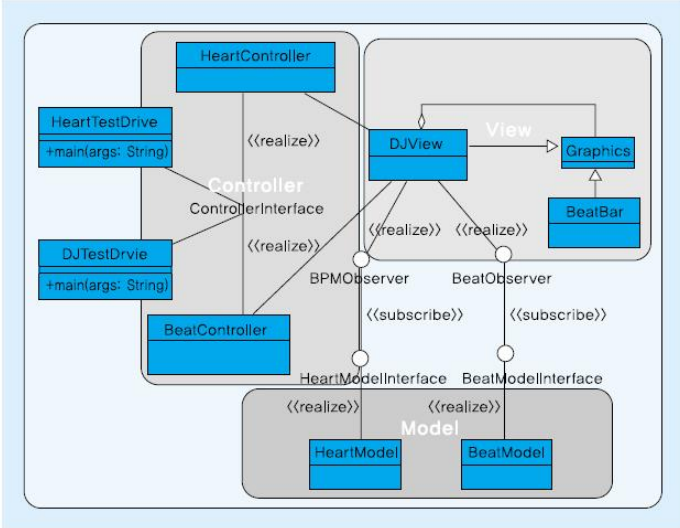
DJ 오디오 시스템

- Model: 아이팟
- View: 스피커
- Controller: 스크래치 디제잉

심장 박동기

- Model: 심장 박동 입력
- View: 심장 박동 수치를 그래프로 표현
- Controller: 측정 대상과 연결되는 입력 단자





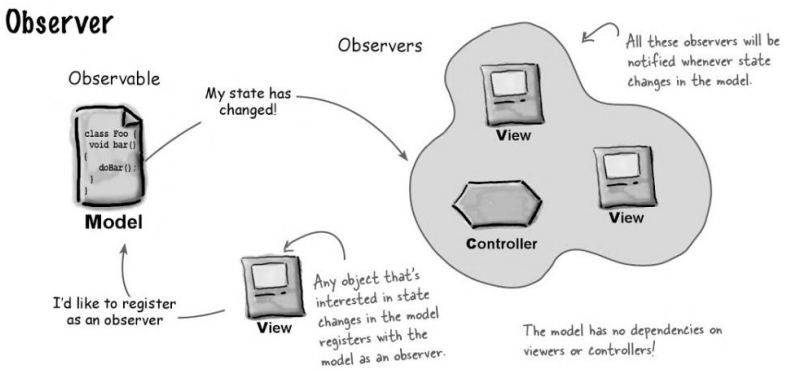
〈그림 7〉 시스템 아키텍처에 적용된 MVC 패턴

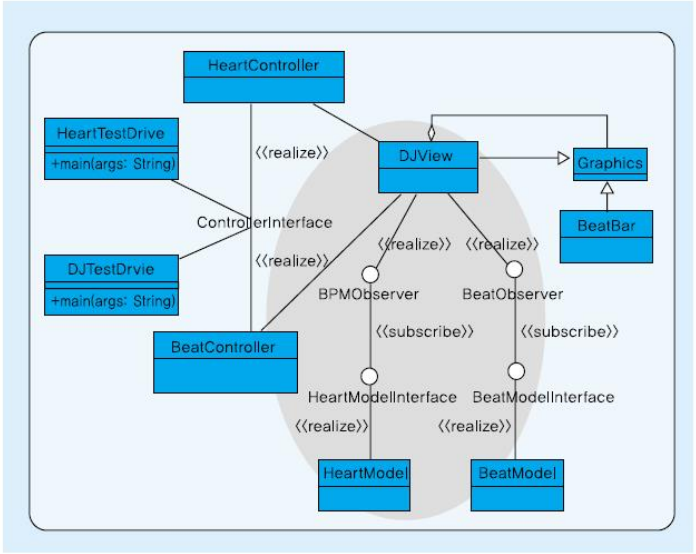
3. 아키텍처 패턴

[예제] 아키텍처 패턴과 디자인 패턴 (skip)

옵저버 패턴 (Observer-)

- 타입: 행위 패턴
- 용도
  - 관찰 대상의 상태가 변했을 때 관찰자에게 통지
  - 상태 변화에 따른 처리를 기술할 때 효과적
- Model과 View를 연결
  - 아이팟에 의해 다음 음이 해석되면 스피커에 자동으로 알려주는 용도

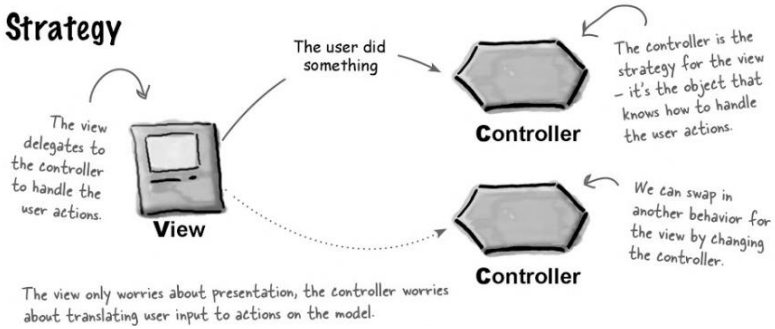




〈그림 8〉모델과 뷰를 연결하는 Observer Pattern

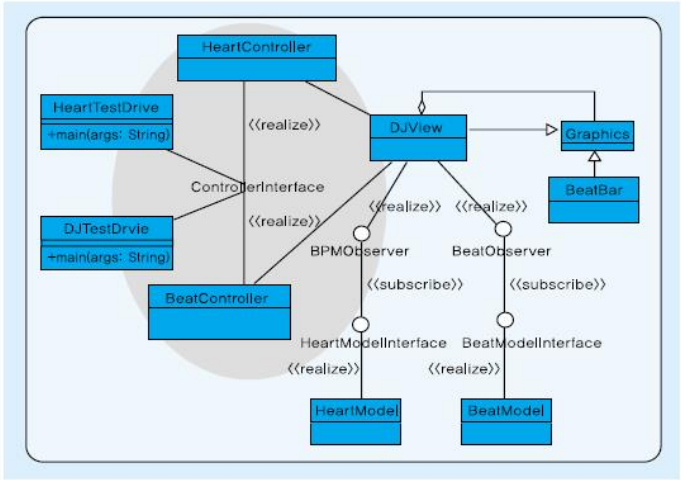
3. 아키텍처 패턴

[예제] 아키텍처 패턴과 디자인 패턴 (skip)



전략 패턴 (Strategy-)

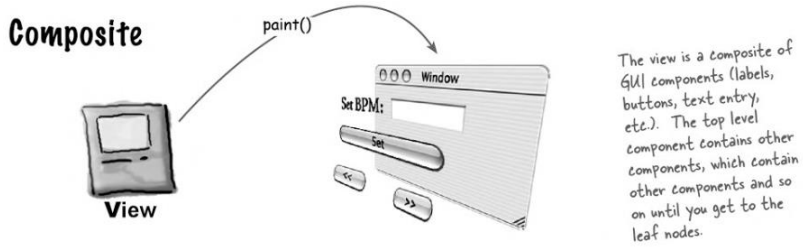
- 타입: 행위 패턴
- 용도
  - 같은 알고리즘들을 각각 캡슐화하여 서로 호환 가능하게 만드는 설계
  - 클라이언트에 따라서 독립적으로 원하는 알고리즘 사용 가능
- Model 컴퍼넌트에 적용
  - 음의 해석을 음악으로 볼 것인지 심장 박동으로 볼 것인지 선택 가능하게 함



〈그림 10〉모델 컴포넌트에 적용된 Strategy Pattern

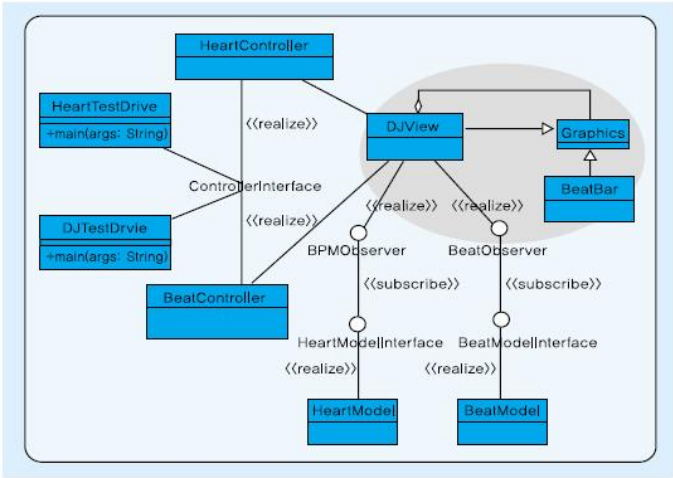
3. 아키텍처 패턴

[예제] 아키텍처 패턴과 디자인 패턴 (skip)



구성 패턴 (Composite-)

- 타입: 구조 패턴
- 용도
  - 집합 속에 포함될 객체와 집합을 가지고 있는 객체 모두가 자신과 동일한 메소드와 데이터를 가지게 함
  - 재귀적 구성: 컨테이너 클래스가 기본 요소 외에 기본 요소를 포함한 컨테이너 자체도 포함 가능
  - 예: 폴더가 파일 외에 폴더도 포함 가능
- View 컴퍼넌트에 적용
  - 사용자 UI를 구성함에 있어 모든 UI 파트를 Graphics라는 유형으로 단일 취급할 수 있게 사용



〈그림 11〉 뷰 컴포넌트에 적용된 Composite Pattern

질문 및 답변