애플리케이션설계

1. 공통 모듈 설계

1) 재사용(Reuse)

* 개념

- 목표 시스템의 개발시간과 비용 절감을 위해 검증된 기능 파악 및 재구성
- 기존의 소프트웨어와 소프트웨어 지식 활용 → 새로운 소프트웨어 구축

* 유형

함수와 객체 사용	- 클래스나 함수 단위로 구현된 소스 코드 재사용
컴포넌트 재사용	- 컴포넌트 단위로 재사용 - 컴포넌트의 인터페이스를 통해 통신
애플리케이션 재사용	- 공통기능을 제공하는 애플리케이션과 기능을 공유하여 재사용

※컴포넌트(Component) : 특정한 기능을 수행하기 위해 독립적으로 개발되어 보급하고 다른 부품과 조립되어 응용시스템을 구축하기 위해 사용되는 소프트웨어 프로그램

2) 공통 모듈(★★★)

* 공통 모듈의 개념

- 모듈의 개념

- 크게 독립된 하나의 소프트웨어 또는 하드웨어 단위를 지칭하는 용어
- 모듈화를 통해 분리된 시스템의 기능들로 서브 프로그램, 서브 루틴, 소프트웨어 내의 단위 프로그램, 작업 단위 등과 같은 의미로 사용

- 모듈의 특징

독립성	- 각각의 모듈은 상대적 독립성을 가짐 - 모듈의 독립성은 결합도와 응집도에 의해 측정
다양한 조합	- 모듈 내부에는 모듈을 하나로 통합하는 수많은 조합이 존재
재사용	- 모듈은 단독으로 컴파일이 가능 + 재사용 가능
영향 최소화	- 독립성이 높은 모듈일수록 수정 시, 다른 모듈에 영향을 거의 미치지 않음

- 공통 모듈의 개념

- 전체 프로그램의 기능 중 특정 기능을 처리할 수 있는 실행 코드
- 자체적으로 컴파일 가능+다른 프로그램에서 재사용 가능

- 여러 기능 및 프로그램에서 공통으로 사용할 수 있는 모듈
  ex> 날짜 처리를 위한 유틸리티 모듈

* 공통 모듈 원칙

정확성 (Correctness)	해당 기능이 실제 시스템 구현 시, 필요한지 아닌지 알 수 있도록 정확히 작성
명확성 (Clarity)	해당 기능에 대해 일관되게 이해되고 한 가지로 해석될 수 있도록 작성
완전성 (Completeness)	시스템이 구현될 때, 필요하고 요구되는 모든 것을 기술
일관성 (Consistency)	공통 기능 간에 상호 충돌이 없도록 작성
추적성 (Traceability)	공통기능에 대한 요구사항 출처와 관련 시스템 등의 유기적 관계에 대한 식별이 가능하도록 작성

* 모듈화

- 개념 : 프로그램이 효율적으로 관리될 수 있도록 시스템을 분해하고 추상화
  > 소프트웨어의 제품 성능 향상
  > 시스템의 수정 및 재사용
  > 용이한 유지 관리

- 기법

<table>
<tr>
<td>루틴
(Routine)</td>
<td>
- 개념 :
소프트웨어에서 특정 동작을 수행하는 일련의 코드로 기능을 가진 명령들의 모임

- 구성 :
<table>
<tr><td>메인 루틴
(Main Routine)</td><td>- 프로그램의 주요 부분이며, 전체의 개략적인 동작 절차를 표시하도록 만들어진 루틴
- 메인루틴은 서브 루틴을 호출</td></tr>
<tr><td>서브 루틴
(Sub Routine)</td><td>메인 루틴에 의해 필요할 때마다 호출되는 루틴</td></tr>
</table>
</td>
</tr>
</table>

- 모듈화 필요성
  - 모듈의 크기가 너무 작을 경우 : 모듈 간 통합비용이 많이 발생
  - 모듈의 크기가 너무 클 경우 : 모듈 당 개발비용이 증가

- 바람직한 모듈 설계 방안
  - 모듈의 독립성과 재사용성을 높이기 위해 결합도를 낮추고 응집도를 높인다.

- 모듈의 복잡도와 중복성을 줄이고 일관성을 유지한다.
- 모듈의 기능은 예측이 가능해야하며 지나치게 제한적이지 않아야 한다.
- 적당한 모듈의 크기를 유지한다 : 비용 문제
- 모듈 간의 효과적 제어를 위해 설계에서 계층적 자료 조직이 제시되어야 한다.
- 유지보수가 용이해야 하고 이식성을 고려해야 한다.

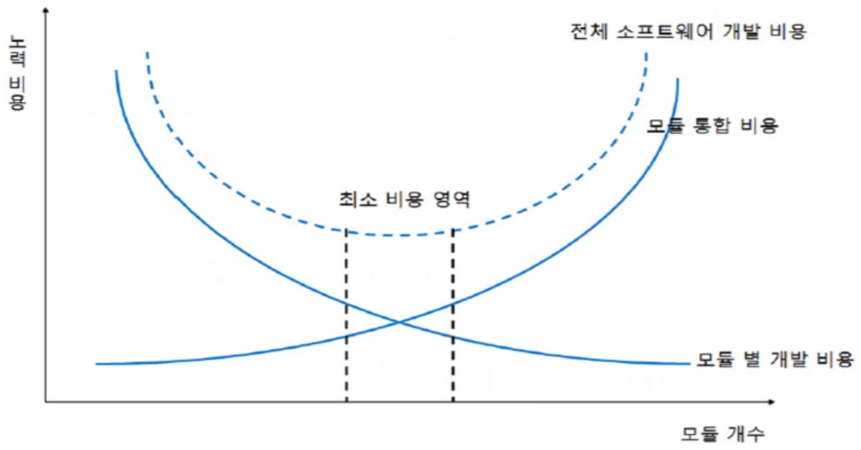


그림. 모듈 개수 및 비용 간 상관관계

* 모듈화 측정 지표

응집도 (Cohesion)	- 모듈의 내부 요소들의 서로 관련되어 있는 정도 - 모듈이 독립적인 기능으로 정의되어 있는 정도
결합도 (Coupling)	- 모듈 간에 상호 의존하는 정도 - 두 모듈 사이의 연관 관계를 맺고 있는 정도

- 좋은 모듈화 : 용도에 맞게 잘 구분된 기능을 가진 모듈들로 세분화
- 개별 모듈은 독립적으로 주어진 역할만 수행, 타 모듈에 의존도가 높지 않아야 한다.

* 모듈화 유형

• 응집도(Cohesion)

☞ 개념

- 모듈의 독립성을 나타내는 개념 : 모듈 내부 구성요소 간, 연관 정도
- 정보 은닉 개념의 확장 개념 > 하나의 모듈은 하나의 기능을 수행

☞ 유형

우연적 응집도 (Coincidental Cohesion)	- 서로 간에 어떤 의미 있는 연관 관계도 없는 기능 요소로 구성될 경우의 응집도 - 서로 다른 상위 모듈에 의해 호출되어 처리상의 연관성이 없는 서로 다른 기능을 수행할 경우의 응집도
논리적 응집도 (Logical Cohesion)	유사한 성격을 갖거나 특정 형태로 분류되는 처리 요소들이 한 모듈에서 처리되는 경우의 응집도

시간적 응집도 (Temporal Cohesion)	연관된 기능이라기보다는 특정 시간에 처리되어야 하는 활동들을 한 모듈에서 처리할 경우의 응집도
절차적 응집도 (Procedural Cohesion)	모듈이 다수의 관련 기능을 가질 때, 모듈 안의 구성요소들이 그 기능을 순차적으로 수행할 경우의 응집도
통신적 응집도 (Communication Cohesion)	동일한 입력과 출력을 사용하여 다른 기능을 수행하는 활동들이 모여있을 경우의 응집도
순차적 응집도 (Sequential Cohesion)	모듈 내에서 한 활동으로부터 나온 출력값을 다른 활동이 사용하는 경우의 응집도
기능적 응집도 (Functional Cohesion)	모듈 내부의 모든 기능이 단일한 목적을 위해 수행되는 경우의 응집도

※ 응집도 : 우연적<논리적<시간적<절차적<통신적<순차적<기능적

- 낮은 응집도 : 하나의 모듈 내부에서 다양한 기능 구현 > 독립성↓
- 높은 응집도 : 단 하나의 기능만을 분리 구현 > 독립성 보장 + 유지보수 용이

- 결합도(Coupling)

☞ 개념

- 외부의 모듈과의 연관도 또는 모듈 간의 상호 의존성을 나타내는 정도
- 소프트웨어 구조에서 모듈 간의 관련성을 측정하는 척도

☞ 유형

- 외부의 모듈과의 연관도 또는 모듈 간의 상호 의존성을 나타내는 정도
- 소프트웨어 구조에서 모듈 간의 관련성을 측정하는 척도

내용 결합도 (Coincidental Cohesion)	- 다른 모듈 내부에 있는 변수나 기능을 다른 모듈에서 사용하는 경우의 결합도 - 하나의 모듈이 직접적으로 다른 모듈의 내용을 참조할 때, 내용적으로 결합되어 있는 경우의 결합도
공통 결합도 (Logical Cohesion)	- 파라미터가 아닌 모듈 밖에 선언되어 있는 전역 변수를 참조하고 갱신하는 식으로 상호작용하는 경우의 결합도 - 공유되는 공통 데이터 영역을 여러 모듈이 사용할 때의 결합도
외부 결합도 (Temporal Cohesion)	- 두 개의 모듈이 외부에서 도입된 데이터 포맷, 통신 프로토콜 또는 디바이스 인터페이스를 공유할 경우의 결합도 - 외부 모듈에서 선언한 데이터를 외부의 다른 모듈에서 참조할 때의 결합도
제어 결합도 (Procedural Cohesion)	- 어떤 모듈이 다른 모듈의 내부 논리 조직을 제어하기 위한 목적으로 제어 신호를 이용하여 통신하는 경우의 결합도 - 하위 모듈에서 상위 모듈로 제어 신호가 이동하여 상위 모듈에게 처리 명령을 부여하는 권리 전도 현상이 발생하는 결합도
스탬프 결합도 (Communication Cohesion)	- 모듈 간의 인터페이스로 배열이나 객체, 구조 등이 전달되는 경우의 결합도 - 두 모듈이 동일한 자료 구조를 조회하는 경우의 결합도 > 자료 구조의 어떠한 변화는 모든 모듈에 영향
자료 결합도 (Sequential Cohesion)	- 모듈 간의 인터페이스로 전달되는 파라미터를 통해서만 모듈 간의 상호작용이 일어나는 경우의 결합도 - 한 모듈의 내용을 변경하더라도 다른 모듈에는 영향을 미치지 않는 상태로

	가장 바람직한 결합도

※ 결합도 : 내용>공통>외부>제어>스탬프>자료

- 효과적 모듈 설계를 위한 유의사항
  - 모듈 간의 결합도를 약화 : 모듈 독립성 향상
  - 모듈의 복잡도와 중복성을 줄이고 일관성을 유지
  - 모듈의 기능은 예측이 가능해야하며 지나치게 제한적이지 않아야 함.
  - 유지보수의 용이
  - 모듈의 수가 증가할 경우
    > 상대적으로 각 모듈의 크기가 작아지고 모듈 간의 상호 교류 증가 → 과부하 현상 발생

- 팬인 / 팬아웃

☞ 개념
- 팬인(Fan-In) : 어떤 모듈을 제어하는 모듈의 수
- 팬아웃(Fan-Out) : 어떤 모듈에 의해 제어되는 모듈의 수

☞ 특징
- 모듈을 계층적으로 분석하기 위해 팬인, 팬아웃을 활용

구분	팬인(Fan-in)	팬아웃(Fan-Out)
모듈 숫자 계산	모듈 자신을 기준으로 들어옴	모듈 자신을 기준으로 나감
고려사항	- 팬인이 높으면 재사용 측면에서 설계가 잘 되었지만 단일 장애점 발생가능 - 팬인이 높으면 관리 비용 및 테스트 비용 증가	- 팬아웃이 높을 경우, 불필요한 모듈 호출 여부 검토 필요 - 팬아웃이 높을 경우, 단순화 여부 검토 필요

- 팬인↑ 팬아웃↓: 시스템 복잡도 최적화

☞ 팬인 및 팬아웃 계산법
- 팬인(Fan-In) : 어떤 모듈을 제어하는 모듈의 수(자신을 기준으로 모듈에 들어옴)
- 팬아웃(Fan-Out) : 어떤 모듈에 의해 제어되는 모듈의 수(자신을 기준으로 모듈에서 나감)

Fan In : 자신을 사용하는 모듈의 수 (A:0, B:1, C:1, D:1, E: 1, F:2, G:1, H:2, I:1, J:1)

Fan Out: 자신이 호출하는 모듈의 수(A:3, B:2, C:2, D:1, E:1, F:1, G:1, H:0, I:0, J:0)

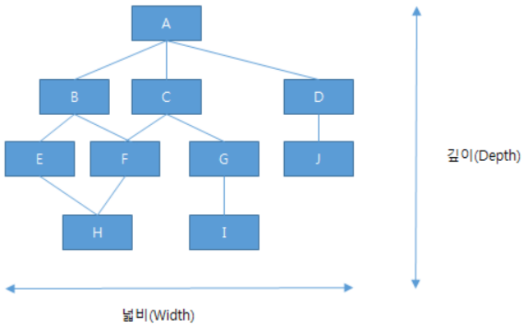


그림. 팬인 및 팬아웃 계산

3) 설계 모델링

* 소프트웨어 설계 유형

- 개념

  - 설계 모델링은 요구사항 분석 단계에서 규명된 필수 기능들의 구체적 구현 방법을 명시하는 기법
  - 소프트웨어에 요구되는 기능과 성능 조건들을 만족하는 소프트웨어의 내부 기능, 구조 및 동적 행위들을 모델링하여 표현, 분석, 검증하는 과정

- 원칙

  - 소프트웨어 설계는 변경이 쉽도록 구조화
  - 하나의 함수 안에 특정 기능을 수행하는데 필요한 자료만 사용하도록 규제
  - 독립적이고 기능적인 특성을 지닌 모듈 단위로 분할 설계
  - 계층적 구조의 보유

- 유형

유형	설명	구성요소
구조 모델링	- 소프트웨어를 구성하는 컴포넌트들의 유형, 인터페이스 내부 설계 구조 및 이들의 상호 연결구조를 모델링 - 시스템의 구성요소들과 이들 사이의 구조적 관계와 특성들을 모델링	프로시저, 데이터 구조, 모듈, 파일 구조
행위 모델링	- 소프트웨어의 구성요소들의 기능들과 이들이 언제, 어떤 순서로 기능을 수행하고 상호 작용하는지를 모델링 - 시스템의 구성요소들이 언제 어떠한 순서로 수행되는가와 같은 동적인 특성들을 모델링	입력 데이터, 출력 데이터, 데이터 흐름, 데이터 변환, 데이터 저장 등

- 프로시저(Procedure) : 프로그램을 기능에 따라 여러 개의 단위로 분해하여 작성 (서브 프로시저+함수 프로시저)
- 모듈(Module) : 프로그램을 구성하는 구성요소의 일부로 관련된 함수나 변수 또는 클래스를 모아 놓은 파일

* 소프트웨어 설계 유형

자료 구조 설계 (Data Structure Design)	요구분석 단계에서 생성된 정보를 바탕으로 소프트웨어를 구현하는데 필요한 자료 구조로 변환하는 과정을 설계
아키텍처 설계 (Architecture Design)	- 예비 설계 또는 상위 수준 설계 - 소프트웨어 시스템의 전체 구조 기술 - 소프트웨어를 구성하는 컴포넌트 간의 관계 정의
인터페이스 설계 (Interface Design)	소프트웨어와 상호 작용하는 컴퓨터 시스템, 사용자 등이 어떻게 통신하는지 설계
프로시저 설계 (Procedure Design)	모듈이 수행할 기능을 절차적 기술로 바꾸는 설계

<table>
<tr><td>협약에 의한 설계
(Design by Contract)</td><td>- 클래스에 대한 여러 가정을 공유하도록 명세한 설계
- 소프트웨어 컴포넌트에 대한 정확한 인터페이스 명세를 위해 다음을 나타냄
<table>
<tr><td>선행조건
(Precondition)</td><td>컴포넌트의 오퍼레이션 사용 전, 참이 되어야 할 조건</td></tr>
<tr><td>결과조건
(Postcondition)</td><td>사용 후 만족되어야 할 조건</td></tr>
<tr><td>불변조건
(Invariant)</td><td>오퍼레이션이 실행되는 동안 항상 만족되어야 할 조건</td></tr>
</table></td></tr>
</table>

※ 상위 설계 : 시스템 수준에서 소프트웨어 구성 컴포넌트들 간의 관계로 구성된 시스템의 전체적 설계

> 자료 구조 설계, 아키텍쳐 설계, 인터페이스 설계, 프로시저 설계, 협약에 의한 설계

하위 설계 : 시스템의 각 구성요소들의 내부 구조, 동적 행위 등을 결정

> 모듈 설계

* 소프트웨어 설계 원리

상향식 설계 (Bottom-Up Design)	- 하위 기능들로부터 시작하여 제일 상위에 있는 기능으로 접근해가는 방식 - 인터페이스가 성립되어 있어야 기능 추가가 용이 - 기존 컴포넌트들을 조합하여 시스템을 개발하는 경우 적합
하향식 설계 (Top-Down Design)	- 소프트웨어 설계 시, 제일 상위에 있는 기능에서 시작하여 기능을 하위 기능들로 분할해 가며 설계하는 방식 - 레벨이 낮은 데이터 구조의 세부 사항은 설계 초기 단계에서 필요 - 통합 검사 시, 인터페이스가 이미 정의되어 있어서 통합이 간단함 - 시스템 명세가 명확한 경우와 모든 것을 새로 개발하는 작업에 적합

* 코드 설계(기출 내용 중심으로 학습)

- 개념

> 데이터의 분류나 조합을 쉽게 하기 위해 사물을 표현하는 코드를 설계하는 기법

- 기능

표준화	정보들 종류, 모양, 크기 등의 일정한 기준에 따라 통일적으로 표현하는 기능
분류	정보들을 동일한 특성을 가진 데이터로 그룹화하여 나누는 기능
식별	다른 것과 구별될 수 있는 기능
배열	일련의 순서로 나열할 수 있는 기능
간소화	정보의 표현을 간소화해서 나타낼 수 있는 기능
연상	정보를 표현하고자 하는 대상의 뜻과 의미가 코드에 내포되게 하는 기능
암호화	정보의 외부 표현을 감추고자 하는 기능
오류 검출	정보 입력이나 관리 시 잘못된 정보를 찾아내는 기능

- 설계 종류

연상 코드 (Mnemonic Code)	코드만 보고 대상을 연상할 수 있도록 명칭 일부를 약호형태로 넣어 구성된 코드 (예> 국가 코드 : KR, US....)
블록 코드 (Block Code)	- 공통성이 있는 것끼리 블록으로 구분 : 각 블록 내에서 일련번호를 부여 - 구분 코드 (예> 전화번호 : 지역번호-국번-일련번호 조합에서 지역번호-국번은 같은 지역끼리 통일)
순차 코드 (Sequence Code)	일정한 기준에 따라 순서대로 일련번호를 부여한 코드 (예> 학교에서의 학생들 번호)
표의 숫자 코드 (Significant Digit Code)	대상 자료의 물리적 수치인 길이, 넓이, 용량 등을 표시한 코드 (예> 20-10-300 : 길이-넓이-용량 조합)
십진 코드 (Decimal Code)	10 진수 형태로 표현한 코드 (예> 상품 바코드)
그룹 분류식 코드 (Group Classification Code)	대상을 기준에 따라 대분류, 중분류, 소분류로 구분하여 번호를 부여 (예> 학번 : 입학 연도-일련번호 조합)

- 코드 설계 절차 :

  코드화 항목 선정→코드화 목적 설정→코드화 대상 확인→코드화 범위 결정→코드 사용 기간 설정→코드화 항목의 특성 분석→코드화 방식 결정→문서화

- 코드 오류 종류 :

사본 오류 (Transcription Error)	- 한 자리를 잘못 표기한 경우 - 필사 오류, 오자 오류라고도 한다.
전위 오류 (Transposition Error)	- 연속된 두 글자가 서로 바뀌어 표기된 경우
생략 오류 (Omission Error)	- 한 글자를 빼먹고 기술한 경우
첨가 오류 (Addition Error)	- 한 글자를 추가되어 기술한 경우
이중 전위 오류 (Double Transposition Error)	- 전위 오류가 중복 발생한 경우

* HIPO(Hierarchy Input Process Output)

- 개념 :

  시스템의 분석 및 설계, 문서화할 때 사용되며 하향식 소프트웨어 개발을 위한 문서화 도구

- 특징

  - 체계적인 문서 관리 가능

- 기호, 도표 등을 사용해 보기가 쉽고 이해 용이
- 기능과 자료의 의존 관계를 동시에 표현
- 변경, 유지보수가 용이
- 시스템의 기능을 고유 모듈들로 분할 : 이들 간의 인터페이스를 계층구조로 표현
  > HIPO 차트

- HIPO 차트 종류

가시적 도표 (Visual Table of Contents)	시스템의 전체적 기능과 흐름을 보여주는 계층 구조도
총체적 도표 (Overview Diagram)	- 입력, 처리, 출력에 대한 정보를 제공하는 도표 - 프로그램을 구성하는 기능을 기술
세부적 도표 (Detail Diagram)	총체적 도표에 표시된 기능을 구성하는 기본 요소들을 상세히 기술하는 도표

4) 소프트웨어 아키텍처(소프트웨어 설계부분에서 가장 큰 틀을 제시하는 부분)

* 개념

- 여러 가지 소프트웨어 구성요소와 그 구성요소가 가진 특성 중에서 외부에 드러나는 특성, 그리고 구성요소 간의 관계를 포함하는 시스템의 구조
- 소프트웨어를 설계하고 전개하기 위한 지침과 원칙

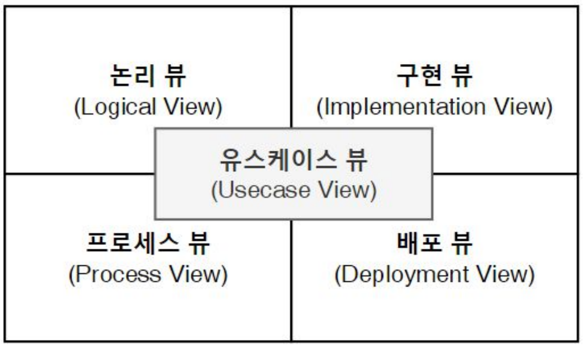
* 필요성

- 아키텍처를 활용하여 주요 이해관계자들 간의 관점 조율을 통해 시스템 최적화
- 아키텍처는 시스템의 비기능적인 요소에 집중해서 만들어지지만 기능적 요소도 고려

* 4+1뷰

- 개념
  - 고객의 요구사항을 정리해 놓은 시나리오를 4개의 관점에서 바라보는 소프트웨어적 접근 방법
  - 4개의 분리된 구조로 구성되는 아키텍처 개념을 제시, 이들 4개 구조가 서로 충돌되지 않는지, 시스템의 요구사항을 충족시키는지를 증명하기 위해 체크 방법으로 유스케이스를 사용

- 구성 요소



4+1 뷰

그림. 4+1 뷰

유스케이스 뷰 (Usecase View)	- 유스케이스 또는 아키텍쳐를 도출하고 설계하며 다른 뷰를 검증하는데 사용되는 뷰 - 외부 행위자에 의해 인식되는 시스템의 기능 요구사항을 보여주는데 초점 - 사용자, 설계자, 개발자, 테스트 관점(all)
논리 뷰 (Logical View)	- 시스템의 기능적 요구사항이 어떻게 제공되는지 설명해주는 뷰 - 설계자, 개발자 관점
프로세스 뷰 (Process View)	- 시스템의 비기능적인 속성으로서 자원의 효율적인 사용, 병행 실행, 비동기, 이벤트 처리 등을 표현한 뷰 - 개발자, 시스템 통합자 관점
구현 뷰 (Implementation View)	- 개발 환경 안에서 정적인 소프트웨어 모듈의 구성을 보여주는 뷰 - 컴포넌트 구조와 의존성을 보여주고 컴포넌트에 관한 부가적 정보 정의
배포 뷰 (Deployment View)	- 컴포넌트가 물리적 아키텍처에 어떻게 배치되는가를 매핑해서 보여주는 뷰 - 물리적 시스템을 구성하고 있는 각 부분들의 분산 형태와 설치에 초점

※ 매핑 : 키(key) 역할을 하는 데이터와 값(value) 역할을 하는 데이터를 짝 지어(=연결 지어) 저장하는 데이터 구조를 말한다.

* 비용 평가 모델

- 개념
  - 아키텍처 접근법이 품질 속성에 미치는 영향을 판단
  - 아키텍처의 적합성을 평가

- 종류

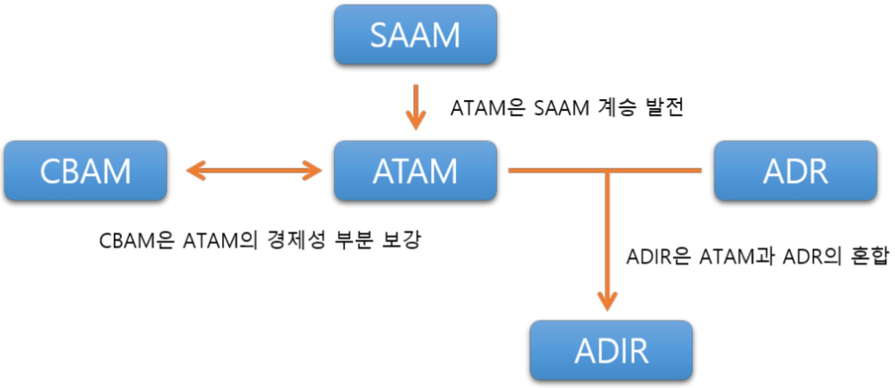


그림. 소프트웨어 아키텍처 비용 평가 모델 관계

SAAM (Software Architecture Analysis Method)	변경 용이성과 기능성에 집중, 평가가 용이하며 경험이 없는 조직에서도 활용 가능한 비용 평가모델

ATAM (Architecture Trade-off Analysis Method)	아키텍쳐 품질 속성을 만족시키는지 판단 및 품질 속성들의 이해 상충 관계까지 평가하는 모델
CBAM (Cost Benefit Analysis Method)	ATAM 바탕의 시스템 아키텍처 분석 중심으로 경제적 의사결정에 대한 요구를 충족하는 비용 평가 모델
ADR (Active Design Review)	소프트웨어 아키텍처 구성요소 간 응집도를 평가하는 모델
ARID (Active Reviews for Intermediate Designs)	전체 아키텍처가 아닌 특정 부분에 대한 품질요소에 집중하는 비용 평가 모델

※ 아키텍쳐 품질 속성 : 아키텍처 비용 평가를 위해 필요한 사항

> 특정 품질에 대한 요구사항을 명세한 내역, 최적의 아키텍처를 선택하기 위한 핵심요소(품질속성)

5) 소프트웨어 아키텍처 패턴(Software Architecture Pattern)

* 개념

- 외부에서 인식할 수 있는 특성이 담긴 소프트웨어의 골격이 되는 기본구조
- 소프트웨어를 설계할 때 참조할 수 있는 전형적인 해결 방식
- 주어진 상황에서의 소프트웨어 아키텍처에서 일반적으로 발생하는 문제점들에 대한 일반화되고 재사용 가능한 솔루션

* 패턴 유형

유형	설명
계층화 패턴 (Layered Pattern)	- 시스템을 계층으로 구분하여 구성 - 각 하위 모듈들은 특정 수준의 추상화 제공 - 각 계층은 다음 상위 계층에 서비스를 제공 - 계층화 패턴은 서로 마주 보는 두 개의 계층 사이에서만 상호작용이 이루어짐
클라이언트-서버 패턴 (Client-Server Pattern)	- 하나의 서버와 다수의 클라이언트로 구성된 패턴 - 사용자가 클라이언트를 통해 서버에 서비스를 요청하면 서버는 클라이언트에게 서비스 제공 - 서버는 계속 클라이언트로부터 요청을 대기
파이프-필터 패턴 (Pipe-Filter Pattern)	- 데이터 스트림을 생성하고 처리하는 시스템에서 사용가능한 단방향 패턴 - 서브 시스템이 입력 데이터를 받아 처리하고, 결과를 다음 서브 시스템으로 넘겨주는 과정을 반복 - 필터 컴포넌트는 재사용성이 좋고, 추가가 쉬움 > 확장이 용이 > 필터 간 데이터 이동에서 데이터 변환 오버헤드 발생
브로커 패턴	- 분리된 컴포넌트들로 이루어진 분산 시스템에서 사용되고, 이 컴포넌트들은 원격

<table>
<tr><td>(Broker Pattern)</td><td>서비스 실행을 통해 상호작용이 가능한 패턴
- 컴포넌트 간의 통신을 조정하는 역할 수행
- 서버는 자신의 기능들(서비스 및 특성)을 브로커(Publish)에 넘겨주며 클라이언트가 브로커에 서비스를 요청하면 브로커는 클라이언트를 자신의 레지스트리에 있는 적합한 서비스로 리다이렉션</td></tr>
<tr><td>모델-뷰-컨트롤러 패턴
(MVC : Model-View-Controller Pattern)</td><td>- 대화형 애플리케이션을모델, 뷰, 컨트롤 뷰 3개의 서브 시스템으로 구조화하는 패턴
<table>
<tr><td>모델
(Model)</td><td>핵심 기능과 데이터 보관</td></tr>
<tr><td>뷰
(View)</td><td>사용자에게 정보 표시
(하나 이상의 뷰가 정의될 수 있음)</td></tr>
<tr><td>컨트롤러
(Controller)</td><td>- 사용자로부터 요청을 입력받아 처리
- 모델과 뷰 사이에서 전달자 역할 수행</td></tr>
</table>
- 각 부분이 별도의 컴포넌트로 분리되어 있어서 서로 영향을 받지 않고 개발 작업 수행 가능
- 컴포넌트를 분리하며 코드의 효율적 재사용을 가능하게 하고, 여러 개의 뷰가 있어야 하는 대화형 애플리케이션 구축에 적합</td></tr>
<tr><td>마스터-슬레이브 패턴
(Master-Slave Pattern)</td><td>- 연산, 통신, 조정을 책임지는 마스터와 제어되고 동기화되는 대상인 슬레이브로 구성되는 패턴
- 일반적으로 실시간 시스템에서 사용</td></tr>
</table>

※ 참고 : https://the-boxer.tistory.com/26

※ 데이터 스트림 : 연결 지향 통신에서, 전송된 정보를 수집하거나 정보를 전송할 때 사용되는 디지털 방식으로 암호화된 일관된 신호의 흐름

※ 오버 헤드 : 어떤 처리를 하기 위해 들어가는 간접적인 처리 시간 · 메모리 등

* 소프트웨어 아키텍처 품질 속성

- 아키텍처 비용 평가를 위해 필요한 사항 :
  > 특정 품질에 대한 요구사항을 명세한 내역
  > 최적의 아키텍처를 선택하기 위한 핵심요소(품질 속성)
- 이해 관계자들의 품질 요구 사항을 반영하여 결정

시스템 품질 속성	가용성(Availability), 변경 용이성(Modifiability), 성능(Performance), 보안성(Security), 사용 편의성(Usability), 시험 용이성(Testabilty)
비즈니스 품질 속성 (시장성을 중심으로 한다.)	시장 적시성(Time to Market), 비용과 이익(Cost and Benefit), 시스템 프로젝트 생명주기(Projected lifetime of the System), 목표 시장(Targeted Market), 신규 발매 일정(Rollout Schedule), 노후 시스템과 통합(Integration with Legacy System)
아키텍처 품질 속성	개념적 무결성(Conceptual Integrity), 정확성과 안정성(Correctness and Completeness), 개발 용이성(Buildabilty)

2. 객체 지향 설계

1) 객체 지향

* 개념

- 실세계의 개체를 속성과 메소드가 결합한 형태의 객체로 표현하는 기법

* 구성 요소

클래스(Class)	- 특정 객체 내에 있는 변수와 메소드를 정의하는 일종의 틀 - 객체 지향 프로그래밍에서 데이터를 추상화하는 단위 - 하나 이상의 유사한 객체들을 묶어서 하나의 공통된 특성을 표현 - 속성은 변수, 행위는 메소드 형태로 선언
객체(Object)	- 물리적, 추상적으로 자신과 다른 것을 식별 가능한 대상 - 클래스에서 정의한 것을 토대로 메모리에 할당 - 객체마다 각각의 상태와 식별성 보유
메소드(Method)	- 클래스로부터 생성된 객체 사용 - 객체가 메시지를 받아 실행해야 할 객체의 구체적 연산 - 전통적 시스템의 함수 또는 프로시저에 해당하는 연산
메시지(Message)	- 객체 간 상호작용을 위한 수단 : 객체에서 객체로 전달 - 객체에게 어떤 행위를 하도록 지시하는 방법
인스턴스(Instance)	- 객체 지향 기법에서 클래스를 통해 만든 실제 실형 각체 - 클래스에 속한 각각의 객체 - 실제로 메모리상에 할당
속성(Property)	- 한 클래스 내에 속한 객체들이 가지고 있는 데이터 값들을 단위별로 정의 - 성질, 분류, 식별, 수량, 현재 상태 등에 대한 표현값

* 객체 지향 기법

캡슐화 (Encapsulation)	- 서로 연관된 데이터와 함수를 함께 묶어 외부와 경계를 만들고 필요한 케이스만을 밖으로 드러내는 기법 - 결합도가 낮아지고 재사용이 용이 - 인터페이스 단순화 - 정보 은닉과 관계가 깊음 - 변경 발생 시, 오류의 파급 효과가 적음
상속성 (Inheritance)	상위 클래스의 속성과 메소드를 하위 클래스에서 재정의 없이 물려받아 사용
다형성 (Polymorphism)	- 하나의 메시지에 대해 각 객체가 가지고 있는 고유 방법으로 응답할 수 있는 능력 - 상속받은 여러 개의 하위 객체들이 다른 형태의 특성을 갖는 객체로 이용될 수 있는 성질

<table>
<tr><td></td><td><table>
<tr><td>오버로딩
(Overloading)</td><td>매개변수의 유형과 개수를 다르게 하여 같은 이름의 메소드를 여러 개 가지는 기법</td></tr>
<tr><td>오버라이딩
(Overriding)</td><td>상위 클래스에서 정의한 일반 메소드의 구현을 하위클래스에서 무시하고 재정의할 수 있는 기법</td></tr>
</table></td></tr>
<tr><td>추상화
(Abstraction)</td><td>- 공통 성질을 추출하여 추상 클래스를 설정하는 기법
- 과정 추상화, 자료 추상화, 제어 추상화가 존재</td></tr>
<tr><td>정보 은닉
(Information Hiding)
>모듈 간의 결합도를 낮춤</td><td>- 코드 내부 데이터와 메소드를 숨기고 공개 인터페이스를 통해서만 접근이 가능하도록 하는 보안 기술
- 필요하지 않는 정보는 접근할 수 없도록 하여 한 모듈 또는 하부 시스템이 다른 모듈의 구현에 영향을 받지 않게 설계(Side-effect 최소화)
- 모듈 내부의 자료 구조와 접근 동작들에만 수정을 국한하지 않기 때문에 요구 사항 등 변화에 따른 수정이 가능
- 모듈 사이의 독립성을 유지
- 설계에서 은닉되어야 할 기본 정보
> IP주소와 같은 물리적 코드, 상세 데이터 구조</td></tr>
<tr><td>관계성
(Relationship)</td><td>- 두 개 이상의 엔터티 형에서 데이터를 참조하는 관계를 나타내는 기법<table>
<tr><td>연관화</td><td>- is-member-of 관계
- 클래스와 객체의 참조 및 이용관계
- 같은 계층에 속하는 클래스들 사이의 상호 의존성을 보여주는 비계측정 관계성을 나타냄</td></tr>
<tr><td>집단화</td><td>- is part of 관계, part-whole 관계
- 서로 관련 있는 여러 개의 객체를 묶어 한 개의 상위 객체를 만드는 특징
- 일반화와 달리 상위 클래스의 성질들이 하위 클래스로 상속화 되지 않음</td></tr>
<tr><td>분류화</td><td>- is-instance-of 관계
- 공통된 속성에 의해 정의된 객체 구성원들의 인스턴스</td></tr>
<tr><td>일반화</td><td>- is-a 관계
- 클래스들 간의 개념적 포함 관계
- 상위 클래스의 특성을 하위 클래스가 상속</td></tr>
<tr><td>특수화</td><td>- is-a 관계
- 상위 클래스의 특성들을 상속받으며 하위 클래스에서 나름의 수정을 가하고 자기 자신의 고유 특성을 갖는 관계</td></tr>
</table></td></tr>
</table>

※ 개체(Entity) : 데이터베이스가 표현하려고 하는 유형, 무형의 정보 대상

* 객체 지향 설계 원칙(SOLID)

단일 책임의 원칙 (Single Responsibility Principle)	- 하나의 클래스는 하나의 목적을 위해 생성 > 클래스가 제공하는 모든 서비스는 하나의 책임을 수행하는데 집중되어 있어야 한다는 원칙 - 객체 지향 프로그래밍의 5원칙 중 나머지 4원칙의 기초 원칙
개방 폐쇄 원칙	소프트웨어의 구성요소(컴포넌트, 클래스, 모듈, 함수)는 확장에는 열려있고

(Open Close Principle)	변경에는 닫혀있어야 한다는 원칙
리스코프 치환의 원칙 (Liskov Substitution Principle)	서브 타입(상속받은 하위 클래스)은 어디서나 자신의 기반 타입(상위 클래스)으로 교체할 수 있어야 한다는 원칙
인터페이스 분리의 원칙 (Interface Segregation Principle)	- 한 클래스는 자신이 사용하지 않는 인터페이스는 구현하지 말아야 한다는 원칙 - 클라이언트가 사용하지 않는 인터페이스 때문에 영향을 받아서는 안 된다는 원칙
의존성 역전의 원칙 (Dependency Inversion Principle)	- 실제 사용 관계는 변하지 않으며, 추상을 매개로 메시지를 주고받음으로써 관계를 최대한 느슨하게 만드는 원칙

※ 의존성 역전의 원칙 : 의존성 관계를 맺을 때, 변화가 없는 것에 의존 관계를 형성

> 추상 클래스 또는 인터페이스를 참고하게 될 경우, 변화에 따른 추가수정X

*** 객체 지향 분석(OOA : Object Oriented Analysis)의 개념**

- 요구사항을 분석하여 요구된 문제와 관련된 모든 클래스(객체), 속성과 연산, 관계 등으로 나누어서 분석하는 기법
- 데이터와 행위를 하나로 묶어 객체를 정의하고 추상화
- 코드 재사용에 의한 프로그램 생산성 향상 및 요구에 따른 시스템의 변경 가능

*** 객체 지향 방법론 종류(←상향식 방식을 기준)**

<table>
<tr><th>종류</th><th>만든이</th><th>설명</th></tr>
<tr><td>OOSE
(Object Oriented Software Engineering)</td><td>야콥슨(Jacobson)</td><td>- 유스케이스에 의한 접근 방법으로 유스케이스를 모든 모델의 근간으로 활용되는 방법론
- 분석→설계→구현 단계로 구성
- 기능적 요구사항 중심의 시스템</td></tr>
<tr><td>OMT
(Object Modeling Technology)</td><td>럼바우(Rumbaugh)</td><td>- 그래픽 표기법을 이용하여 소프트웨어 구성요소를 모델링하는 방법론
- 분석 절차 : 객체 모델링→동적 모델링→기능 모델링
<table>
<tr><td>객체 모델링
(Object Modeling)</td><td>- 정보 모델링이라고도 하며, 시스템에서 요구하는 객체를 찾고 객체 간의 관계를 정의하는 모델링
- 가장 중요하며 선행되어 진행
- 객체 다이어그램을 활용하여 표현</td></tr>
<tr><td>동적 모델링
(Dynamic Modeling)</td><td>- 시간의 흐름에 따라 객체들 사이의 제어흐름, 동작 순서 등의 동적인 행위를 표현하는 모델링
- 상태 다이어그램을 활용하여 표현</td></tr>
<tr><td>기능 모델링
(Functional Modeling)</td><td>- 프로세스들의 자료 흐름을 중심으로 처리 과정을 표현하는 모델링
- 자료 흐름도(DFD)를 활용하여 표현</td></tr>
</table></td></tr>
<tr><td>OOD
(Object Oriented Design)</td><td>부치(Booch)</td><td>- 설계 문서화를 강조하여 다이어그램 중심으로 개발하는 방법론
- 분석과 설계의 분리가 불가능
- 분석하는데 이용된 객체 모델의 설계 시 적용</td></tr>
</table>

+ Coad와 Yourdon 방법론 :
  E-R다이어그램을 사용하여 객체의 행위를 모델링하며 객체 식별, 구조 식별, 주체 정의, 속성 및 관계 정의 ,서비스 정의 등의 과정으로 구성되는 객체 지향 분석 방법

+ Wirfs-Brock 방법론 :
  분석과 설계 간의 구분이 없고 고객 명세서를 평가해서 설계 작업까지 연속적으로 수행하는 분석 방법

1) 디자인 패턴

* 개념

- 소프트웨어 공학의 소프트웨어 설계에서 공통으로 발생하는 문제에 대해 자주 쓰이는 설계 방법을 정리
- 디자인 패턴을 참고하여 개발할 경우, 개발의 효율성과 유지보수성 운용성 등의 품질이 높아지며 프로그램의 최적화에 도움

* 구성 요소

패턴의 이름	디자인 패턴을 부를 때, 사용하는 이름과 디자인 패턴의 유형
문제 및 배경	디자인 패턴이 사용되는 분야 또는 배경, 해결하는 문제를 의미
솔루션	디자인 패턴을 이루는 요소들, 관계, 협동 과정
사례	디자인 패턴의 간단한 적용 사례
결과	디자인 패턴을 사용하면 얻게 되는 이점이나 영향
샘플 코드	디자인 패턴이 적용된 원시 코드

* 유형

생성 패턴 (Creational Pattern)	- 객체 인스턴스 생성에 관여 - 클래스 정의와 객체 생성 방식을 구조화 - 캡슐화를 수행
구조 패턴 (Structural Pattern)	더 큰 구조 형성 목적으로 클래스나 객체의 조합을 다루는 패턴
행위 패턴 (Behavioral Pattern)	클래스나 객체들이 상호 작용하는 방법과 역할 분담을 다루는 패턴

* 종류(패턴의 특징 정도만 파악하고 넘어가기)

- 생성 패턴

Builder	- 복잡한 인스턴스를 조립하여 만드는 구조 > 복합 객체를 생성할 때, 객체를 생성하는 방법(과정)과 객체를 구현(표현)하는 방법을 분리함으로써 동일한 생성 절차에서 서로 다른 표현 결과를 도출 - 생성과 표기를 분리>복잡한 객체 생성
Prototype	- 처음부터 일반적인 원형을 만들어 놓고, 그것을 복사 후 필요한 부분만 수정하여

	사용, 생성할 객체의 원형을 제공하는 인스턴스에서 생성할 객체들의 타입이 결정되도록 설정하며 객체를 생성할 때 갖추어야 할 기본 형태가 있을 때 사용되는 디자인 패턴 - 기존 객체를 복제함으로써 객체를 생성
Factory Method	- 상위 클래스에서 객체를 생성하는 인터페이스를 정의 > 인스턴스를 만드는 방법만 결정 - 하위 클래스에서 인스턴스를 생성하도록 하는 방식 > 데이터의 생성을 책임지고 조작하는 함수들을 오버라이딩 > 인터페이스와 실체 객체를 생성하는 클래스를 분리할 수 있는 특성을 갖는 디자인 패턴 - 생성할 객체의 클래스를 국한하지 않고 객체 생성
Abstract Factory	- 구체적인 클래스에 의존하지 않고 서로 연관되거나 의존적인 객체들의 조합을 만드는 인터페이스를 제공하는 패턴 > 생성된 클래스에서는 사용자에게 인터페이스(API)를 제공 > 구체적인 구현은 Concrete Product 클래스에서 이루어지는 특징을 갖는 디자인 패턴
Singleton	- 전역 변수를 사용하지 않고 객체를 하나만 생성하도록 하며, 생성된 객체를 어디에서든 참조할 수 있도록 하는 디자인 패턴 - 한 클래스에 한 객체만 존재

- 구조 패턴

Bridge	- 기능의 클래스 계층과 구현의 클래스 계층을 연결 - 구현 부에서 추상 계층을 분리하여 추상화된 부분과 실제 구현 부분을 독립적으로 확장 가능 - 구현뿐만 아니라, 추상화된 부분까지 변경해야 하는 경우 활용
Decorator	- 기존에 구현되어 있는 클래스에 필요한 기능을 추가해 나가는 설계패턴 - 기능 확장이 필요할 때, 객체 간 결합을 통해 기능을 동적으로 유연하게 확장할 수 있게 해주어 상속의 대안으로 사용하는 디자인 패턴 - 객체의 결합을 통해 기능을 동적으로 유연하게 확장
Facade	- 복잡한 시스템에 대해 단순한 인터페이스 제공 > 사용자와 시스템 간 또는 여타 시스템과의 결합도를 낮추어 시스템 구조에 대한 파악을 쉽게 하는 패턴 > 오류에 대해서 단위별로 확인 > 단순한 인터페이스를 제공하여 접근성 향상
Flyweight	- 다수의 객체로 생성될 경우, 모두가 갖는 본질적인 요소를 클래스 화하여 공유 > 메모리 절약+클래스 경량화 - 여러 개의 가상 인스턴스를 제공하여 메모리 절감
Proxy	- 실제 객체에 대한 대리 객체로 실제 객체에 대한 접근 이전에 필요한 행동을 취할 수 있게 만들며, 이 점을 이용해 미리 할당하지 않아도 상관없는 것들을 실제 이용할 때 할당하게 하여 메모리 용량을 아낄 수 있으며, 실제 객체를 드러나지 않게 하여 정보은닉의 역할도 수행 - 특정 객체로의 접근을 제어하기 위한 용도
Composite	- 객체들의 관계를 트리구조로 구성 : 부분-전체 계층을 표현하는 패턴 - 사용자가 단일 객체와 복합객체 모두 동일하게 다루도록 함. - 복합 객체와 단일 객체를 동일하게 취급
Adapter	- 기존에 생성된 클래스를 재사용할 수 있도록 중간에서 맞춰주는 역할을 하는 인터페이스를 만드는 패턴 - 상속을 이용하는 클래스 패턴과 위임을 이용하는 인터페이스 패턴의 두 가지 형태로

	사용되는 디자인 패턴

- **행위 패턴**

Mediator	- 객체 지향 설계에서 객체의 수가 너무 많아지면 서로 간 통신을 위해 복잡해져서 객체 지향에서 가장 중요한 느슨한 결합의 특성을 해칠 수 있기 때문에 이를 해결하는 방법으로 중간에 이를 통제하고 지시할 수 있는 역할을 하는 중재자를 두고, 중재자에게 모든 것을 요구하여 통신의 빈도수를 줄여 객체 지향의 목표를 달성하게 해주는 패턴 - 상호작용의 유연한 변경을 지원
Interpreter	- 언어의 다양한 해석, 구체적으로 구문을 나누고 그 분리된 구문의 해석을 맡는 클래스를 각각 작성하여 여러 형태의 언어 구문을 해석할 수 있게 만드는 디자인 패턴 - 문법 자체를 캡슐화하여 사용
Iterator	- 컬렉션 구현 방법을 노출시키지 않으면서 그 집합체 안에 들어있는 모든 항목에 반복자를 사용하여 접근할 수 있는 디자인 패턴
Template Method	- 어떤 작업을 처리하는 일부분을 서브 클래스로 캡슐화해 전체 일을 수행하는 구조는 변경하지 않으면서 특정 단계에서 수행하는 내역을 변경 - 상위 작업의 구조를 바꾸지 않으면서 서브 클래스로 작업의 일부분을 수행
Observer	- 한 객체의 상태가 바뀌면 그 객체에 의존하는 다른 객체들에 연락이 가고 자동으로 내용이 갱신되는 방법으로 일대 다의 의존성을 가지며 상호 작용하는 객체 사이에서는 가능하면 느슨하게 결합하는 디자인 패턴 - 객체의 상태 변화에 따라 다른 객체의 상태도 연동, 일대다 의존
State	- 객체 상태를 캡슐화하여 클래스화 > 그것을 참조하게 하는 방식으로 상태에 따라 다르게 처리할 수 있도록 행위내용을 변경, 변경 시 원시 코드의 수정을 최소화할 수 있고, 유지보수의 편의성도 갖는 디자인 패턴 - 객체의 상태에 따라 행위 내용 변경
Visitor	- 각 클래스 데이터 구조로부터 처리 기능 분리 > 별도의 클래스를 만들어 놓고 해당 클래스의 메서드가 각 클래스를 돌아다니며 특정 작업을 수행하도록 만드는 패턴, 객체의 구조는 변경하지 않으며 기능만 따로 추가하거나 확장할 때 사용하는 디자인 패턴 - 특정 구조를 이루는 복합 객체의 원소 특성에 따라 동작을 수행할 수 있도록 지원
Command	- 실행될 기능을 캡슐화함으로써주어진 여러 기능을 실행할 수 있는 재사용성이 높은 클래스를 설계하는 패턴 - 하나의 추상 클래스에 메서드를 만들어 각 명령이 들어오면 그에 맞는 서브 클래스가 선택되어 실행 - 요구사항을 객체로 캡슐화
Strategy	- 알고리즘 군을 정의하고 같은 알고리즘을 각각 하나의 클래스로 캡슐화한 다음, 필요할 때 서로 교환해서 사용할 수 있게 하는 패턴 - 행위를 클래스로 캡슐화해 동적으로 행위를 자유롭게 바꿀 수 있게 해주는 디자인 패턴
Memento	- 클래스 설계 관점에서 객체의 정보를 저장할 필요가 있을 때 적용하는 디자인 패턴으로 Undo 기능을 개발할 때 사용하는 디자인 패턴 - 객체를 이전 상태로 복구시켜야 하는 경우, 작업취소 요청 가능
Chain Of Responsibility	- 정적으로 어떤 기능에 대한 처리의 연결이 하드코딩 되어 있을 때, 기능 처리의 연결 변경이 불가능한데, 이를 동적으로 연결되어 있는 경우에 따라 다르게 처리될 수 있도록 연결한 디자인 패턴 - 한 요청을 2개 이상의 객체에서 처리

* 장단점

장점	- 요구사항 변경에 따른 소스 코드 변경을 최소화할 수 있게 해줌 - 소프트웨어 코드의 품질을 향상시킬 수 있음 - 설계 변경 요청에 대한 유연한 대처 가능 - 범용적 코딩 스타일 적용 가능 - 개발자 간 원활한 의사소통 가능 - 재사용을 통한 개발 시간 단축 가능 - 소프트웨어 구조 파악 용이 - 객체 지향 설계 및 구현의 생산성을 높이는데 적합 - 소프트웨어의 품질과 생산성 향상
단점	- 객체 지향 설계/구현 위주로 사용 - 초기 투자 비용의 부담