안녕하세요. 메모리사업부 진수언입니다.

저는 현업에서 모바일 낸드 스토리지의 Nand Flash FW를 개발하고 있습니다.

그중에서도 FTL 레이어 개발 부서에서 근무하고 있습니다.

FTL에도 여러 모듈이 있는데 그 중에서 IO 관련 모듈을 주로 개발하고 있습니다.

코스웍에서 느낀점..

현업에서 C로 업무를 하고 있는데, 유지보수에 많은 어려움을 겪고 있다.

고객 요구사항으로 인한 기능적 변경이나, target 성능을 맞추기 위한 다양한 변경이 요구되는 경우가 많은데,

해당 변경이 시스템에 미치는 영향을 예측하는 것이 매우 어렵습니다.

아키텍쳐 어떻게 설계를 해야 이런 문제들로부터 비교적 자유로워질 수 있는지 배울 수 있었다

AA인증과정에 입과하기 전에는 SW 설계나 유지보수에 관한 지식이 많이 부족하여, 현업에서 업무를 하면서

요구사항을 정제하거나, 설계문서를 작성하는 것이 불편하고 귀찮은 일인줄만 알았는데, 코스웍과 인증과제를 통해 해당 작업들이 SW를 개발하고 유지보수하는데 있어서 얼마나 중요한 작업인지 깨닫게 되었습니다.

품질속성 ISO9126관련 그 품질속성이 뭘 의미하는지 + 품질속성 올릴 수 있는 tactic  
(additional 품질속성시나리오, 품질속성시나리오 구성요소 등)

* Performance 품질 속성을 위한 tactic말해봐라
* 가용성 품질 속성을 위한 tactic 말해봐라

SOLID가 뜻하는 5가지 principle

* Lsp가 뭔지 설명해봐라

Design pattern 패턴 싹다정리

* 배치시퀀셜 vs 파이프앤필터
* 미디에이터 vs 퍼사드
* 팩토리패턴은 뭐냐?
* 팩토리 vs abstract의 차이점?
* Micro service의 단점은 뭐냐

Architecture Style 싹다정리

인증과제관련

* 블랙보드
* 클러스터링
* 내가 썼던 태틱이랑 그림들 이해하기
* ATAM 시나리오 선정 방법 + NFR 시나리오에 대해 importance/difficulty 선정 이유
* P\_EK를 어떻게 representation했는가
* Quality in use가 왜 필요한가? SW의 품질이 internal/external quality로 다 표현되는거아니냐
* Deployment diagram에서 dispatcher의 역할이 뭐냐 -> 서버여러개라 워크로드 분배를 위해  
  -> 서버 여러 개면 데이터 싱크 어떻게맞출거냐 -> 브릿지패턴 -> 왜 브릿지패턴?
* Class diagram에서 device에 대한 가변성을 어떻게 나타냈는가?
* HI변경 용이성에 composite pattern 쓴 이유를 설명해라

품질속성 시나리오의 구성요소

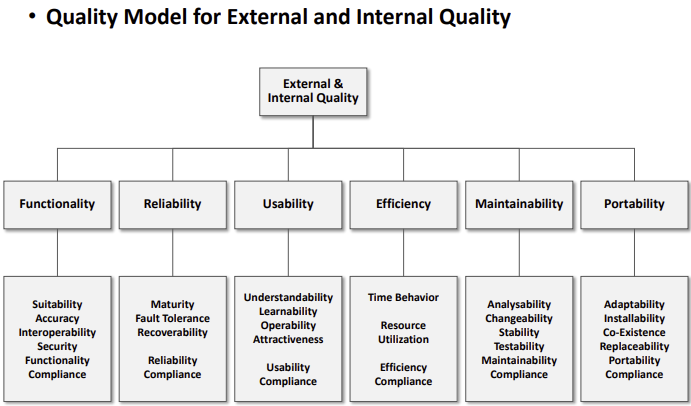
자극 유발원(Source of stimulus) 자극(Stimulus) 환경(Environment) 대상체(Artifact)   
응답(Response) 응답 측정(Response measure)

텍스트이(가) 표시된 사진

자동 생성된 설명

가용성 변경용이성 성능 보안 시험용이성 사용편의성

품질 (Software Quality)



SW Quality

Types of Quality Aspects (품질측면의 유형)

Process Quality : 라이프사이클 프로세스의 품질

Internal quality : 중간 아티팩트의 품질

External quality : 구현 코드의  품질

Quality In Use : 사용자가 평가하는 부분

ISO 9126

|  |  |
| --- | --- |
| Functionality 기능성 | Suitability 적합한 기능 세트를 제공  Accuracy 올바른(합의된) 결과 또는 효과를 제공  Interoperability 지정된 시스템과 상호 작용  Security 공격에 저항 |
| Reliability 신뢰성 | Maturity(성숙도) 실패를 방지 기능  Fault Tolerance(내결함성) 소프트웨어 장애 또는 지정된 인터페이스 침해의 경우에 특정 수준의 성능을 유지하는 능력  Recoverability(복구가능성) 장애 발생 시 데이터를 복구 |
| Usability 사용성 | Understandability 이해하기 쉽고  Learnability 학습할 수 있고  Operability 작동하고 제어  Attractiveness 좋아하는, 매력적인 |
| Efficiency 효율성 | Time Behavior 응답시간 처리시간  Resource Utilization 적절한 리소스를 사용할 수 있는 |
| Maintainability 유지보수성 | Analysability 분석 가능해야 하며(디버거빌리티)  Changeability 변경용이성  Stability 수정으로 인한 예상치 못한 영향을 최소화  Testability 시험용이성 |
| Portability 이식성 | Adaptability 서로 다른 지정된 환경에 대해 적용가능  Installability 지정된 환경에 설치  Co-Existence 리소스를 공유하는 다른 소프트웨어와 공존  Replaceability 지정된 다른 소프트웨어 대신 사용 |

[가용성](https://ko.wikipedia.org/wiki/%EA%B0%80%EC%9A%A9%EC%84%B1)(Availability) 이란?

정상 동작시간 / 전체 사용시간(정상+비정상) 이 높을수록 고 가용성이라 한다.

가용성 태틱(tactic)으로는 3종류가 있음. Prevention, Fault Tolerance, Recoverability

Prevention

|  |  |
| --- | --- |
| • [P1] 2 Phase Commitment Transaction | 트랜잭션 + 롤백 |
| • [P2] Active Redundancy | 여러 컴포넌트에 같은 일을 시킴 |
| • [P3] Passive Redundancy | Standby 컴포넌트가있어서 계속 Update 시켜주다가  Fault 발생시 Activate해서 사용 |
| • [P4] Spare | Cold Spare 가있음  Fault 발생시 Initialize해서 사용 |

Fault Tolerance

|  |  |
| --- | --- |
| • [T1] Ping/Echo |  |
| • [T2] Heartbeat | 주기적으로 살아있음을 알림 |
| • [T3] Voting | 여러개를 돌려서 결과를 비교 |
| • [T4] Process Monitor / Watch Dog | 감시하다가 오류가 나면 초기화 시킴 |
| • [T5] Exceptions | Try Catch 처럼 오류가 감지되면 Handler가 처리함 |

Recoverability

|  |  |
| --- | --- |
| • [R1] Shadowing | Shadow 모드에서 코치고 다시 돌아감 |
| • [R2] State Resynchronization | Redundancy로 고치고 최근 상태로 동기화 |
| • [R3] Rollback with Checkpoint | 체크포인트로 롤백 |

SOLID

|  |  |  |
| --- | --- | --- |
|  | **약어** | **개념** |
| **S** | [SRP](https://ko.wikipedia.org/wiki/%EB%8B%A8%EC%9D%BC_%EC%B1%85%EC%9E%84_%EC%9B%90%EC%B9%99) | **[단일 책임 원칙 (Single responsibility principle)](https://ko.wikipedia.org/wiki/%EB%8B%A8%EC%9D%BC_%EC%B1%85%EC%9E%84_%EC%9B%90%EC%B9%99)**  한 [클래스](https://ko.wikipedia.org/wiki/%ED%81%B4%EB%9E%98%EC%8A%A4_(%EC%BB%B4%ED%93%A8%ED%84%B0_%EA%B3%BC%ED%95%99))는 하나의 책임만 가져야 한다.  모든 클래스는 하나의 책임만 가지며, 클래스는 그 책임을 완전히 캡슐화해야 함을 일컫는다.  다양한 Responsibility를 가져야 된다면 Class를 분리  SRP를 하면 응집도(Cohesion)가 좋아짐 |
| **O** | [OCP](https://ko.wikipedia.org/wiki/%EA%B0%9C%EB%B0%A9-%ED%8F%90%EC%87%84_%EC%9B%90%EC%B9%99) | **[개방-폐쇄 원칙 (Open/closed principle)](https://ko.wikipedia.org/wiki/%EA%B0%9C%EB%B0%A9-%ED%8F%90%EC%87%84_%EC%9B%90%EC%B9%99)**  “소프트웨어 요소(클래스, 모듈, 함수 등등)는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다.”  Open: Software Entities (class, module, function 등)이 확장성 있게 만들어져야 한다  Closed: 그 확장을 위해 과도한 수정이 필요하면 안된다.  (변화에 대해 닫혀있다) |
| **L** | [LSP](https://ko.wikipedia.org/wiki/%EB%A6%AC%EC%8A%A4%EC%BD%94%ED%94%84_%EC%B9%98%ED%99%98_%EC%9B%90%EC%B9%99) | **[리스코프 치환 원칙 (Liskov substitution principle)](https://ko.wikipedia.org/wiki/%EB%A6%AC%EC%8A%A4%EC%BD%94%ED%94%84_%EC%B9%98%ED%99%98_%EC%9B%90%EC%B9%99)**  “프로그램의 [객체](https://ko.wikipedia.org/wiki/%EA%B0%9D%EC%B2%B4)는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다.”.  Sub Type 은 그들의 Base Type을 대체할 수 있어야 한다.  LSP를 위배하는 상속보다는, Composition(합성)을 통해 코드 재사용  Cost가 있는게 아니기 때문에, 적용하지 않을 이유가 없음. |
| **I** | [ISP](https://ko.wikipedia.org/wiki/%EC%9D%B8%ED%84%B0%ED%8E%98%EC%9D%B4%EC%8A%A4_%EB%B6%84%EB%A6%AC_%EC%9B%90%EC%B9%99) | **[인터페이스 분리 원칙 (Interface segregation principle)](https://ko.wikipedia.org/wiki/%EC%9D%B8%ED%84%B0%ED%8E%98%EC%9D%B4%EC%8A%A4_%EB%B6%84%EB%A6%AC_%EC%9B%90%EC%B9%99)**  “특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다.”  클라이언트가 자신이 이용하지 않는 메서드에 의존하지 않아야 한다는 원칙. 인터페이스 분리 원칙은 큰 덩어리의 인터페이스들을 구체적이고 작은 단위들로 분리시킴으로써 클라이언트들이 꼭 필요한 메서드들만 이용할 수 있게 한다. |
| **D** | [DIP](https://ko.wikipedia.org/wiki/%EC%9D%98%EC%A1%B4%EA%B4%80%EA%B3%84_%EC%97%AD%EC%A0%84_%EC%9B%90%EC%B9%99) | **[의존관계 역전 원칙 (Dependency inversion principle)](https://ko.wikipedia.org/wiki/%EC%9D%98%EC%A1%B4%EA%B4%80%EA%B3%84_%EC%97%AD%EC%A0%84_%EC%9B%90%EC%B9%99)**  High Level Module은 Low-level module에 의존성을 가지면 안된다.  둘 다 Abstraction에 대해서 의존성을 가져야 한다.  “추상화에 의존해야지, 구체화에 의존하면 안된다.”[[4]](https://ko.wikipedia.org/wiki/SOLID_(%EA%B0%9D%EC%B2%B4_%EC%A7%80%ED%96%A5_%EC%84%A4%EA%B3%84)#cite_note-martin-design-principles-4) [의존성 주입](https://ko.wikipedia.org/wiki/%EC%9D%98%EC%A1%B4%EC%84%B1_%EC%A3%BC%EC%9E%85)은 이 원칙을 따르는 방법 중 하나다. |

캡슐화(encapsulation): 객체의 속성(data fields)과 행위(메서드, methods)를 하나로 묶고, 실제 구현 내용 일부를 내부에 감추어 은닉한다.

Architecture Style

|  |  |
| --- | --- |
| Batch Sequential | 서브시스템에서 다음 서브시스템으로 순차적으로 데이터를 처리해 나가는 스타일  Data를 조작하는 Component 들로 구성됨.  데이터 처리 시작부터 끝까지 사용자의 개입이 불가능하며  만약 100개의 데이터set에 대한 처리를 해야 하는 경우 모두 끝난 후 다음 단계를 시작  (장) 입출력만 맞으면 컴포넌트 교체가능  (단) 동시성 미지원 = 낮은성능 |
| Pipe-n-Filter | Pipe는 데이터를 전달하고 Filter는 Data를 처리합니다. 데이터 스트림을 처리하는데 적합한 아키텍쳐 스타일입니다. 필터는 지속적으로 이전 파이프에서 데이터를 가져와 처리하고 다음 파이프로 넣습니다.  (장) Concurrency(동시성이 보장되고 병렬처리가 가능하여 성능향상이 가능함.  (장) 필터 교환이나 재조합을 통한 유연성 확보 가능  (단) 데이터 스트림 형태가 고정됨. 동적으로 못바꿈 |
| Layered | 관련있는 기능별 모듈을 그룹화하여 계층화하는 아키텍쳐로 수직적인 구조의 아키텍쳐  (장) Layer 별로 개발하여 유지보수성이 좋고 높은 이식성을 제공합니다.  (장) 하위 레이어의 변화가 상위 레이어에 영향을 미치지 않습니다.  (단) 레이어간 전달로인한 오버헤드로 성능 이슈, 중복코드 발생가능 |
| MVC | 상호작용 어플리케이션을 Model View Controller 세개의 컴포넌트로 구분하는 아키텍처 패턴입니다. Model컴포넌트는 데이터를 Encapsulate / View는 사용자와 상호작용 / Controller는 트렌젝션을 수행. Observer 패턴을 이용하여 View와 Moddel의 데이터 일관성을 보장합니다.  (장) 유지보수성 확장성보장 (단) 간단한 프로그램의 경우 복잡도만 증가시킬 수 있음. Model과 View의 완벽한 분리가 어렵다 |
| Blackboard | Nondeterministic 문제를 해결하는데 유용한 아키텍처 패턴입니다.  Blackboard, KnowledgeSource, Control로 구성되어 있으며, Blackboard에는 풀어야 할 문제들과 partial solution을 관리하고, KnowledgeSource는 문제에 대한 답을 Blackboard에 공유합니다. Control은 지속적으로 Blackboard를 확인하고 적절한 로직을 수행합니다.  (장) knowledge source의 추가 및 삭제가 용이함  (단) 결과값이 매번 바뀔 수 있어 완전한 해결책을 보장하지 못함. |
| Shared Repository | 여러 컴포넌트들이 공통의 데이터에 접근해야 할 때 사용합니다. 직접 접근함  (장) 컨제스쳔 발생할 순 있지만 **데이터 공유가 효율적이다**, 일관성 보장이 쉽다.  (단) 데이터 스키마가 변경되면 접근하는 컴포넌트들이 다 변경되어야 함 |
| Micro Kernel | Core Syste에 핵심적이고 공통적인 기능을 제공하고 각 기능 모듈을PlugIn을 통해 제공  (장) 이클립스처럼 새로운 기능을 유연하게 추가할 수 있음  (단) message passing으로 인한 성능저하? 개발 비용이 많이 듬 |
| Micro Service | 서비스를 잘게 나누어 배포 인터페이스를 통해 마이크로 서비스를 호출하여 사용함.  하나의 큰 애플리케이션을 독립적으로 동작가능한 여러 개의 작은 애플리케이션으로 쪼개어 변경과 조합이 가능하도록 만든 아키텍처를 말합니다.  각각의 애플리케이션들이 느슨하게 연결해서 독립적으로 개발 및 유지보수가 가능하도록 만드는 구조 즉 기능간 분리입니다.  (장) 작은 서비스 단위로 독립적으로 개발 운영 가능, 테스트 및 유지보수 비용 절감  (단) 같은 내용일지라도 서비스별로 따로 관리함으로 정합성 문제 |
| Dispatch | 디스패쳐와 다수의 서버로 구성  높은 품질의 서버로 연결해주고, 연결 이후에는 클라이언트가 직접 서버를 호출  (장) 브로커보다 성능은 좋음  (단) Session이 길어지면 서버 다운가능 |
| Broker | 브로커와 다수의 서버로 구성 / 높은 품질의 서버로 Client의 요청을 위임함.  브로커를 통해 Client와 Server가 통신함  (장) 서버하나가 고장나도 QoS측정하여 다른 서버로 연결해줌  (단) 브로커에 오버해드, SPOF(single point of failure)문제 야기 가능 |

생성(Creational) Design Pattern

|  |  |
| --- | --- |
| Factory Method | 팩토리 패턴은 객체를 만드는 과정에 집중한 패턴  구체적인 객체 생성 과정을 하위 클래스로 옮기는게 목적임  -> 팩토리 인터페이스를 구현하여 객체를 생성하도록 함(Create하는 method를 제공)  (장) OCP원칙을 지킬 수 있음(수정에 열리고 확장에 닫힘)  (단) 타입이 다른 다양한 객체를 사용할 경우에 구현을 다시해야함 |
| Abstract Factory | 추상팩토리 패턴은 팩토리를 사용하는 클라이언트에 집중한 패턴  연관되거나 의존적인 여러종류의 객체를 concrete(구체적) 클래스에 의존하지 않고 많들 수 있게 하는게 목적임  -> 팩토리 객체가 아니라 다른 객체  (장) 콘크리트(구체적) 클래스에 의존하지 않고 다양한 객체 생성 가능  (단) 타입이 아니라 아예 새로운 종류의 제품을 제공하는데에 문제가 있을 수 있다(?) |
| Builder | 같은 구성요소를 가지지만 서로다른 방법으로 구축되는데 적용함  “복잡한 객체”를 생성하는 클래스와 표현하는 클래스를 분리하여,  동일한 절차에서도 서로 다른 표현을 생성하는 방법을 제공한다.  (장) 유동적으로 필드에 값을 세팅하고, 객체를 생성한 후 변경불가 상태로 만들게 됨 🡪 코드가 깔끔해짐  (단) 수정이 필요한 경우 setter 추가해야됨  관리해야 할 클래스가 늘어날 수 있다. |
| Prototype | 생성할 객체들의 타입이 프로토타입인 인스턴스로부터 결정되도록 하며, 인스턴스는 새 객체를 만들기 위해 자신을 복제(clone)하게 된다.  (장) 객체를 생성해주기 위한 별도의 객체 생성 클래스가 필요하지 않습니다. 객체의 각 부분을 조합해서 생성되는 형태에도 적용이 가능합니다.  (단) 생성될 객체들의 자료형인 클래스들이 모두 clone() 메서드를 구현해야 합니다. |
| Singleton | 객체의 인스턴스가 오직 1개만 생성되는 패턴. 단 하나의 인스턴스만 만들도록 보장하고싶을 때 사용  (장) 메모리를 효율적으로 사용 가능, 공유가 쉽고 일관성 보장 쉬움  (단) testability가 떨어짐(전역에서 공유하는 변수가 있으니까)  멀티스레딩 환경이라면 synchronize 문제도 해결해야함(동시접근성) |

구조(Structural) Design Pattern

|  |  |
| --- | --- |
| Adapter | 인터페이스가 맞지 않을 때 사용  🡪 호환성 없는 인터페이스 때문에 같이 못쓰는 애들을 함께 동작할 수 있도록 해줌  (장) 기존 코드를 변경하지 않아도 됨 🡪 재사용성 증가  (단) 구성 요소를 위해 클래스를 증가시켜야 해서 복잡도 증가할 수 있음  Adapter의 경우 상속을 사용하기 때문에 유연하지 못함 |
| Bridge | 큰 클래스 또는 밀접하게 관련된 클래스들의 집합을 두 개의 개별 계층구조​(추상화 및 구현)​로 나눈 후 각각 독립적으로 개발할 수 있도록 함.  (장) 추상화된 부분과 실제 구현 부분을 독립적으로 확장 가능 🡪 확장성증가  구현할 인터페이스에 완전히 결합시키지 않았기 때문에 구현과 추상화된 부분을 분리시킬 수 있습니다.  (단) 디자인이 복잡 |
| Composite | 객체의 계층적 표현이 필요할 때 사용(파일 디렉토리같은곳)  객체들의 관계를 트리 구조로 구성하여 부분-전체 계층을 표현하는 패턴으로,  사용자가 단일 객체와 복합 객체 모두 동일하게 다루도록 한다.  (장) 객체를 동일화시켜 클라이언트를 단순화함  (단) Leaf와 composite의 공통된 operation을 component가 갖고 있어야 하므로 단일 책임 원칙(SRP)에 위배될 수 있다 |
| Decorator | 객체의 책임과 행동을 동적으로 수정해야 할 때 사용  장식의 순서나 개수를 동적으로 바꿀 수 있다 (예시로는 카페에서 커피 레시피->가격)  (장) 장식의 순서나 개수를 동적으로 바꿀 수 있다(런타임에)  (단) 구성 요소를 초기화하는 코드가 복잡 |
| 퍼사드 | 퍼사드패턴은 복잡한 시스템에 대한 엑세스를 제공하기 위해 간단한 통합 인터페이스를 제공하는 패턴임. 서브시스템이 퍼사드의 존재를 모름  (장) 클라이언트와 서브시스템간의 커플링을 줄일 수 있다.  (단) 퍼사드에 기능을 추가하기 어렵다(강제로 동일한 인터페이스 적용해서) |
| 프록시 (대변인, 대리자) | 어떤 객체를 사용하고자 할때, 객체를 직접적으로 참조하는 것이 아닌 해당 객체를 대항하는 객체를 통해 대상 객체에 접근하는 방식을 사용하면 해당 객체가 메모리에 존재하지 않아도 기본적인 정보를 참조하거나 설정할 수 있고, 실제 객체의 기능이 필요한 시점까지 객체의 생성을 미룰 수 있다. |

행위(Behavior) Design Pattern

|  |  |
| --- | --- |
| Strategy | 관련 클래스들이 행위만 다루게 함. (매소드 일일이 바꾸는게 어려우니까)  (장) 전략 추가가 쉽다  (단) 언제 어떤 전략을 사용해야 하는 지 클라이언트가 알아야함 |
| Template method | 작업을 처리하는 일부분을 서브 클래스로 캡슐화하는 패턴  수행하는 구조 전체를 바꾸지 않으면서 특정 단계에서 수행하는 내역을 바꾸는 패턴  (장) 중복 코드를 없애고 하위 클래스에서 특정 로직에만 집중하게 할 수 있음(SRP)  (장) 새로운 비즈니스 로직이 추가되어도 subclass만 수정할 수 있음(OCP)  (단) 클래스 생성이 많아져 관리해야할 클래스가 많아짐 -> 복잡도 증가 |
| Mediator  중재자 | 미디에이터(중제자) 패턴은 중제자가 다수의 객체를 조정해야하는 경우 사용함  중제자 패턴은 서브시스템이 중제자를 앎.  퍼사드처럼 강제적으로 모든 서브시스템에게 동일한 정책을 제공하는게 아니라  각각의 서브 시스템에게 맞는 정책을 적용한다.  (장) 중제자에 기능 추가가 가능하다. 객체간 통신을 위해 서로 직접 참조할 필요가 없다.  (단) 객체의 형태나 통신 로직이 자주 변경되는 경우 유지보수가 어렵다(?) |
| Observer | 관찰 대상의 상태변화를 하나이상의 옵저버가 통지받아야할 때 사용  관찰자가 관찰대상을 등록하는걸 subscribe  관찰대상이 관찰자 여럿한테 알려주는게 publish  (장) 로우커플링 달성하여 의존성을 제거할 수 있다. 유연하다  (단) subscribe하고 사용 안하게되면 메모리 누수 가능함 |
| Iterator | 컬렉션 객체 안에 들어있는 모든 항목에 접근하는 방식이 통일되어 있으면  모든 항목에 일일이 접근하는 작업을 이터레이터가 대신해줄 수 있음.  (장) 다양한 데이터 순회 알고리즘 코드를 분리할 수 있음(SRP)  (장) 코드 수정하지 않고도 새로운 타입의 컬렉션과 이터레이터 추가 가능(OCP)  (단) 컬렉션의 특정 인스턴스에 바로 접근해야 할 경우 비효율적일 수 있다. |
| State | 객체 내부 상태에 따라 행동을 변경할 수 있다  State에 따른 behavior가 정해질 때 사용함  객체 내부 상태에 따라 객체가 스스로 다른 행동을 하도록 할 떄 씀  스트레티지랑 비슷한데 스테이트에 따라 알아서 동작하는게 다름  (장) 스테이트 별 동작이 변경되더라도 삭제 및 수정이 간편함(변경용이성),  (장) if else나 switch case를 적게 쓰게되어 코드가 간결해지고 가독성도 좋아짐  (단) 관리할 클래스 수가 늘어남 |
| Command | 하나의 객체를 통해 여러 객체들에 명령(Command)을 해야 할 때 사용되는 패턴이다.  커멘드 패턴을 사용하면 요청을 캡슐화해서 커멘드 객체가 명령을 해야하는 객체들에 대한 의존성을 느슨하게 만들 수 있다. |
| Memento | 객체를 이전 상태로 되돌릴 수 있는 기능(Undo)을 제공하는 패턴  오리지네이터(originator), 케어테이커(caretaker), 메멘토(memento).  (단) undo를 위해 이전 상태를 저장해야하는데, 비용이 클 수 있다. |