전체 문제

## OOP Concepts

|  |  |
| --- | --- |
| **OOP Concepts** | |
| Procedural Programming  vs.  OO Programing | * Procedural Programming: 프로그램의 순서와 흐름을 먼저 세우고 필요한 자료 구조와 함수를 설계하는 방식 (Fortran, C) * 장점: 프로그램의 흐름이 쉽게 이해 된다. * 단점: 절차가 중심이 됨에 따라 현실세계의 문제를 모델링이 어렵다.   Data관리및 코드의 유지보수가 어렵다.   * OO Programing: 객체(자료구조와 이를 기반으로 한 모듈)들을 먼저 설계하고 실행 순서와 흐름으로 구성하는 방식 – Program의 기능을 객체들간의 communication을 통해 제공함. * 장점: 관련된 data가 객체로 표현됨에 따라 Strong Cohesion을 가짐.   각 객체가 독립적으로 존재 함에 따라 weak coupling 이 된다.   * 단점: 상속구조로 인하여 코드가 복잡해 질 수 있다.   의도하지 않은 data의 공유가 발생할 수도 있다. |
| **Q:** What is the typical drawback of procedural programs such as C? | *1. Data와 Method의 응집성이 떨어져서 SW 규모가 커질수록 유지보수 비용이 많이 든다.* |
| **Q:** Why is *Dynamic Binding* essential in object-oriented programming? | *1. OO Program의 기본 개념인 Polymorphism과 상속이 동작하기 위해서는 Run-time에 실제 동작에 연결되는 것이 반드시 필요함.* |
| Encapsulation | * 객체의 attribute와 method를 하나로 묶고, 실제 구현 내용을 외부에 감추는 것이다. * 객체의 상세 내용을 객체 외부에 철저히 숨기고 단순히 메시지 만으로 객체와의 상호작용을 하게 하는 것이다. * Component들은 black-box로 하고, interface만을 노출한다. * Information Hiding의 수단이 된다. * 클래스간 의존도를 줄이고 클래스 내부의 응집도를 높일 수 있다 |
| Information Hiding   * Visibility | * Attribute나 method의 visibility를 정의하여 외부에서 접근하지 못하게 하는 것. * 객체에 대한 구체적인 information을 노출시키지 않는다. 즉 implementation을 노출 시키지 않고, interface를 노출시키는 것이 한 예가 될 수 있다. * Encapsulation을 통해서 객체의 정보 노출을 막는 것 |
| Encapsulation vs. hiding | 캡슐화는 관련된 요소를 묶어 캡슐 내부와 외부를 구별짓는 장치이고, info hiding은 캡슐 내부요소들에 대한 세부 구현사항을 외부에 숨기는 장치이다. 따라서 캡슐화 했다고 해서 반드시 hiding이 되는 것은 아니다. |
| Inheritance | * 객체간의 관계 중 하나로 super class의 attribute와 method를 sub class가 상속 받아서 반복을 제거하고, 재사용성을 높인다 |
| ★ Polymorphism   * Overloading * Overriding,   Dynamic Binding, Substitutability | * Polymorphism: 하나의 method나 class가 다양한 형태를(Type을) 가질 수 있게 하는 것 * Overloading: 구현이 다른 동일한 이름의 Method를 여러 개 만드는 것. * Overriding: Super class에서 상속받은 것을 재정의하여 사용 * Dynamic Binding: class의 method가 super/sub class의 method중 어떤 것이 호출될 것인지를 runtime시에 결정하는 것. * Substitutability: 선언은 superclass 로 하고, 생성은 subclass 로 하여 유연성을 높인다. 항상 subclass는 superclass의 역할을 할 수 있어야 한다. (LSP)   -> composite/decorate 패턴이 좋은 예. |
| Abstract Class  vs.  Interface | Interface는 모든 method가 abstract method이며, attribute를 가질 수 없다. 이에 비해 abstract class는 attribute를 가질 수 있고, abstract method 뿐 아니라 일반 method도 가질 수 있다.  Abstract Class: Abstract Class와 이를 상속받은 하위클래스가 개념상 포함관계가 존재하는 경우 (확장목적). Superclass로부터 생성되는 Instance가 없게 하기 위한 목적도 있음.  Interface: 개념상 포함관계가 아닌 동작상 공통점만 뽑아놓은 것 |
| **Q:** What is the difference between *Abstract Class* and *Interface*? | 위와 상동. |
| Generic Class/Template | 타입 변수를 받아서 사용한다는 같은 목적을 가지지만,  Generic Class: JAVA. 특정 data 형식과 관련이 없는 작업을 encapsulation할 수 있도록 타입 매개변수를 가지는 class (class 내부에서 사용할 data type을 외부에서 지정하는 기법)  Template: C++. Generic class와 매우 비슷하지만, 조금 다른 특성을 가지고 있다. Java는 down-casting이 가능한 하나의 클래스가 생성되지만, C++은 타입 별로 별도의 클래스들이 생성된다. 따라서 static변수를 공유하지 못하는 등의 차이가 있다. |
| Object vs Class | Object is an instance of class.  클래스(class)는 객체 지향 프로그래밍(OOP)에서 특정 객체를 생성하기 위해 변수와 Method를 정의하는 일종의 틀. |
| Abstraction | 구체적인 것을 관찰자가 관심있는 특성만 가지고 재조합 하는것  프로세스에서, 중요한 디테일과 그렇지 않은 것 구분  아이템에서, 중요한 특성을 관찰, 뽑아내는 것 |

## OO Analysis & Design

|  |  |
| --- | --- |
| **OO Analysis & Design** | |
| **※ 각 Diagram의 구성요소가 어떤 것인지 질문.** | |
| Use Case Diagram   * Relationships, Granularity of Use Cases * Use Case Description/Scenario | System의 기능적 요구사항과 시스템 외의 요소를 Actor와 Use-case로 표현.  Actor + Use Case + Association.  Relationships: Extend/Include/Generalization  Granularity(크기): Use-case가 담당하는 기능 범위  Use Case Description: Actor, Pre/Post Condition, Basic Path, Alternative Path, Exceptional Path 등으로 Use Case를 설명. |
| Class Diagram   * 5 Types of Relationships | 시스템을 구성하는 클래스의 구성 (attribute, method)과 클래스 간의 관계를 표현  Inheritance   * Generalization: 상위 클래스를 상속하는 관계 * Implementation: Interface를 구현하는 관계   Composition: 상대 클래스를 소유하는 관계. 상대 클래스와 life cycle이 동일. 함께 생성, 함께 해제  Aggregation: 상대 클래스와 life cycle이 다름. Reference를 공유하는 관계. 생성할 때 참조 필드 할당.  Association: Method 호출을 통하여 상대 클래스의 참조 할당.  Dependency: 상대 클래스에 대해 알고만 있는 수준. Method의 파라미터로 사용되는 경우 |
| **Q:** Order the strengths of 5 relationship types. Justify the ordering. | 설명:   1. Inheritance는 Life Cycle이 동일. Superclass의 일부 동작은 반드시 포함. 2. Composition은 Life Cycle이 동일. 3. Aggregation은 Life-Cycle은 다르나, 생성시 관계가 설정됨. 4. Association은 해당 클래스의 참조를 지속적으로 유지함. 5. Dependency는 Dependency가 함수 단위로 범위가 제한적임. |
| Sequence Diagram   * Workflow related to Use Case Description * Applying MVC style paradigm in drawing Sequence Diagram | 하나의 Use Case에 대한 객체들 사이의 interaction을 표현한 diagram. Behavior를 보여주는 diagram  Use Case diagram에 명시된 action에 대한 객체들의 interaction이 sequence diagram에 표현되어야 함.  Sequence diagram에 명시된 class가 class diagram에 정의되어 있어야 하고 주고 받는 메시지 (method)도 정의되어 있어야 함  alt: 다른 시나리오 (if~else if ~ else)  opt: optional 선택적 시나리오 (if)  par: parallel로 수행하는 경우.  loop: 반복문 / break: loop를 벗어난 경우  seq: 순서가 엄격하지 않은 경우 / strict: 순서가 엄격한 경우  neg: 절대 발생해선 안되는 경우  critical: atomic한 처리 / par: 병렬 처리  ignore: 특정 메시지 무시 / consider: 특정 메시지 고려  Symbol : Actor, LifeLine, Synchronous message / Asynchronous message / response message, Fragment  Fragment : Alt, opt , loop , break, seq , strict, par, critical, ignore, consider, assert, neg  **Activity vs. Sequence?**  Activity diagram is good at depicting the control flowing from one activity to another, especially good at the logic of conditional structures, loops, concurrency.  Sequence diagram is good at depicting the sequence of messages flowing from one object to another, how their messages/events are exchanged in what time-order.  **Activity: flow of usecases, sequence: interaction between objects in one usecase** |
| State Machine Diagram   * Transition, Event, Guard, Action | Target Machine의 주요 동작이 State의 Transition이 주 Control을 표현하는 경우.  시스템의 상태 전이를 표현하는 diagram.  State에 종속적인 시스템의 behavior를 표현할 때 사용  Orthogonal state: Compatible하고 independent하게 동시에 존재하는 2개 이상의 state  Transition: 시스템의 상태가 변경되는 것을 말함.  Event: 원래 상태에 있을 때 해당 전이를 유발시키는 사건  Guard: 전이 조건. 시스템 전이 시 검토되는 Boolean 식  Action: 전이를 할 때 수행하는 동작  Class 하나당 state machine diagram 하나씩 있어야 함. |
| **Q:** What is *Event* in State Machine Diagram? Give examples. | Event: 원래 상태에 있을 때 해당 전이를 유발시키는 사건  예: 사용자의 입력, 외부 상태의 변화. Time-out 등. |
| Timing Diagram | Sequence diagram과 state machine diagram을 합쳐 놓은 것  Timing을 보여주는 Diagram으로 시간의 변화에 따른 Object의 상태 변화를 표현할 수 있습니다.  시간에 따른 객체의 상태와 그 상태를 변경시키는 메시지들을 보여줌  Timing diagram에서 state 변경을 촉발 시키는 요인이 state machine diagram에 명시되어 있어야 함.  Timing diagram은 객체들 간의 상태 변화를 시간 제약으로 표현한 것이므로 state machine diagram만으로는 대체가 어려움 |
| Activity Diagram   * Action, Activity | 전체 시스템의 action들이 순차, 병행 방식으로 어떻게 수행되는지 workflow를 보여주는 diagram  Behavior를 보여주는 diagram  Use Case diagram에 명시된 action에 대한 workflow가 activity diagram에 표현되어야 함   * Action is a named element which represents a single atomic step within activity * Activity is a parameterized behavior represented as coordinated flow of actions.   Activity includes “action, object, control”  Transition, decision, merge, fork-join |
| **Q:** When would you use Activity Diagram? | The purpose of an activity diagram can be described as −   * Draw the activity flow of a system.   시스템에 필요한 동작의 흐름을 도식화   * Describe the sequence from one activity to another.   각 Activity들의 실행순서를 명확화   * Describe the parallel, branched and concurrent flow of the system.   병렬, 분기, 동시 실행 흐름을 도식화. |
| Component Diagram   * Provided Interface, Required Interface * Pluggable Object | Component Diagram: 컴포넌트간의 연결관계를 표현.  복잡하고 큰 시스템에서 내부 의존성과 구조를 나타낼 수 있습니다.  A provided interface is the one that is either   * realized directly by the component itself, or * realized by one of the classifiers realizing component, or * is provided by a public port of the component. * 객체가 실현하는 역할   A required interface is either   * designated by usage dependency from the component itself, or * designated by usage dependency from one of the classifiers realizing component, or * is required by a public port of the component. * 객체가 자신의 역할 수행을 위해 필요로 하는 다른 객체의 역할   Pluggable Object: 플러그러블 객체 (Pluggable Object): 컴포넌트 라이브러리 패턴을 사용하여 컴포넌트를 추가할 경우 추가한 클래스가 기존 클래스와 약간만 다를 수 있다. 이때 클래스를 새로 추가하지 않고 기존의 클래스에 인자를 사용하여 객체를 생성하도록 하여 다양성을 높일 수 있다. 인자는 상수, 심볼, 클래스 레퍼런스, 인스턴스 변수, 함수 포인터 등이 될 수 있다. |
| **Q:** Why is *Required Interface* essential in designing platform SW? | Required Interface로의 dependency를 통해 모듈 구현이 이루어진다. 추상화된 Required Interface 제공이 DIP를 실현하는 방법이다. (Dependency Inversion Principle) |
| Deployment Diagram   * Node, Artifacts | Deployment Diagram: SW모듈(Artifacts)이 어떤 HW Component(Node)에 존재하는지를 표현.  (Deployment diagram would show what hardware components ("nodes") exist (e.g., a web server, an application server, and a database server), what software components ("artifacts") run on each node.)  실행시점의 시스템의 구조를 모델링하기 위한 다이어그램으로, 하드웨어 구성과 소프트웨어 배치 구성을 표현합니다. 즉 소프트웨어가 구동되는 컴퓨팅 노드와 노드 간의 연결관계를 나타냅니다.  - Node: 시스템을 구성하는 소프트웨어 컴포넌드가 배치(Deploy)되어 수행되는 하드웨어자원을 뜻한다.  - Artifacts: 물리적인 형태의 모든 정보 (모델 파일, 소스 코드, 산출물, 실행 파일)  스테레오타입으로 <<artifact>>로 표기 |
| 병렬 표현 | 시퀀스 다이어그램의 "par" combined fragment(복합적 부분) 를 이용하여 표현가능  Activity 다이어그램의 Fork 를 이용하여 표현가능  Sate 다이어그램의 Fork를 이용하여 표현가능 |
| UML Constructs for Parallel Processing   * Available in UML Diagrams | Behavioral diagram에서 표현 가능 🡪 Sequence, Activity, Timing, State Machine  Sequence diagram의 par  Activity diagram의 fork  State Machine diagram의 fork |
| Consistency   * Among Use Case Model, Class Diagram, & Sequence Diagram * Between State Machine Diagram & Class Diagram * Between Activity Diagram & Use Case Diagram | * Use Case diagram & Class diagram   . Use Case에서 언급되는 data는 class diagram에 정의되어 있어야 한다.  . Functionality가 dataset을 manipulation한다.   * Sequence Diagram & Class Diagram   . Sequence Diagram의 Method가 Class Diagram의 Method/인자로 표현.   * State Machine Diagram & Class Diagram   . State가 Class로 표현되어야 함.   * Activity Diagram & Use Case Diagram   . Use Case들이 activity diagram의 action work flow로 표현되어야 함  Timing & Sequence  State와 해당하는 message들이 존재하는 지 sequence에서 확인가능 |
| **Q:** How do you check the consistency between <X diagram> & <Y diagram>? | 위 내용 상동. |
| UP (Unified Process) | Inception -> Elaboration -> Construction -> Transition  OO software를 점진적이고 반복적으로 개발해나가는 방법론입니다. 초반에 core archictecture를 안정화하는데 초점을 맞추고, 그 이후로 점진적으로 구체화해나갑니다. Inception, Elaboration, Conttruction, Transition 단계를 거치지만, 유의해야 할것은 모든 설계가 끝나고 개발한다는 개념은 아니며 phase마다 중점을 두는 것이 다를뿐, 요구사항 정제, 설계, 개발, 테스트는 모든 phase 에서 일어납니다. |

## Principles of SW Design

|  |  |
| --- | --- |
| **Principles of SW Design** | |
| SOLID   * Set of 5 principles | ※ “왜?” 에 대해서 질문함.  연관된 design pattern 생각  SOLID:   * SRP (Single Responsibility Principle): 하나의 클래스는 하나의 책임만 갖는다.   수정할 이유는 하나여야 한다. High cohesion   * OCP (Open-Close Principle): 변경에는 닫혀있어야 하고 확장에는 열려 있어야 한다.   Starategy, decorator.. inheritance, composition   * LSP (Liskov Substitution Principle): 상위 클래스가 사용되는 곳에 이를 상속받은 하위 클래스로 치환해도 정상 동작이 이루어진다.   지키지 않으면 해당 type을 확인하는 분기가 필요해지고 이는 재사용성을 줄이고 복잡도를 늘려서 예기치 못한 동작을 야기할 수도 있다.   * ISP (Interface Segregation Principle): 특정 client를 위한 interface 여러 개가 범용 interface 하나보다 낫다. SRP의 interface 버전 Clients should not be forced to depend on methods that they do not use.   Notimplementerror 등을 만들게 되는 때가 있음  사용하지 않는 부분들을 덜어냄으로써 cohesion을 높이고 dependency를 낮춘다.   * DIP (Dependency Inversion Principle): Implementation이 아닌 abstraction에 의존하여 프로그래밍해야 한다.   changes are risky, and by depending on a concept instead of on an implementation, you reduce the need for change at call sites. Coupling을 줄인다  LSP:예시로, square는 setwidth 호출 이후 height도 동시에 변경되므로, LSP 위배. |
| **Q:** What are the benefits of applying SOLID? | 모듈화 (Low coupling, High cohesion)를 가능하게 하여 재사용성, 유지보수성을 높여준다. |
| Abstraction | 복잡한 모듈/시스템으로부터 핵심적인 개념이나 기능을 간추려 내는 것을 의미한다. Unnecessary한 detail을 user가 보지 않게 함으로서 복잡성을 줄인다. |
| Refinement | Generic term of computer science that encompasses various approaches for producing correct computer programs and simplifying existing programs to enable their formal verification  정확한 SW를 만들기 위한, 혹은 기존에 존재하는 프로그램을 효율적으로 개선하기 위한 여러 Approach. |
| Separation of Concerns | * 프로그램에 영향을 주는 정보들을 구분하여 분리하고, 그 분리에 따라서 영역을 구분하여 설계하는 방법. Concern은 HW가 될 수도 있고, Class 또는 MVC처럼 layer가 될 수도 있다. * System이 하나의 concern만을 담당하는 부분들로 나누어 져야 한다는 원칙이다. Concern은 명확한 feature를 말하며, 이 concern들 사이에는 overlap이 최대한 없어야 한다. * SRP는 low level관점에서 class나 module이 하나의 responsibility를 가져야 한다는 principle이고, SOC는 high level관점에서 system의 feature를 나누는 것이다 좀 더 명확하게 * Layered 에서 Layer를 나누는 것이 좋은 예가 될 수 있다. |
| **Q:** Give examples of applying Separation of Concerns? | * Internet protocol stack (Layered Architect)   + ISO 7 Layer: Application – Presentation – Session – Transport – Network – Datalink - Physical * 세부 Class로 분리 (Object-Oriented) |
| ★Modularity, Functional Independence   * Cohesion * Coupling | Modularity: High Cohesion & Low Coupling  Cohesion: Indication of the relative functional strength of a module.  Coupling Indication of the relative interdependence among modules. |
| **Q:** What are the differences between *Cohesion & Coupling*? | Cohesion은 class내부의 응집성, Coupling은 class간 의존성  Modularity를 위해서는 cohesion은 높아야 하고 coupling은 낮아야 한다 |
| Information Hiding | 정보를 외부로부터 은닉시키고 이를 이용하기 위한 method를 정의하도록 하여 클래스간 coupling을 낮추고 모듈성을 향상시킬 수 있음. |
| **Q:** What is the *Information Hiding* mechanism in OOP? | Encapsulation supports information hiding by making use of the three access specifiers of a class.   * Public: If a class member is public, it can be used anywhere without the access restrictions. * Private: if a class member is private, it can be used only by the members and friends of class. * Protected: if a class member is protected, it can be used only by the members and friends of class and the members and friends of classes derived from class. |
| ADD (Attribute Driven Design) | 체계적으로 반복적인 분해과정을 통한 SW 설계방식  1) 분해할 모듈 선택 (처음엔 전체 시스템으로 시작, 이후에는 Subsystem, …)  2) architecture driver를 만족하는 style, tactic들을 적용  3) 모듈 분해가 필요 없을 때까지 1)2) 반복 |
| Architecture Driver 도출 | Stakeholder와 concern 사이의 association을 표현하고 그 중에서 우선 순위가 높은 concern을 선택, 선택된 concern을 더 구체화 시키고 정제하여 architecture driver로 선정 |
| Architecture Viewpoint | 특정 system concern을 나타내기 위해 architecture view의 생성, 해석, 사용을 위한 convention을 정의해 놓은 것 |
| GRASP design pattern | GRASP ( General Responsibility Assignment Software Patterns )  : 어떤 Responsibility를 어떤 class/object에게 부여할 것인지를 결정하는 원칙이다. (역할 부여의 원칙)  1) Creator  A 객체와 B 객체의 관계가 다음중 하나라면 A의 생성을 B의 역할로 부여하라  - B 객체가 A 객체를 포함하고 있다.  - B 객체가 A 객체의 정보를 기록하고 있다.  - A 객체가 B 객체의 일부이다.  - B 객체가 A 객체를 긴밀하게 사용하고 있다.  - B 객체가 A 객체의 생성에 필요한 정보를 가지고 있다.  ex) VideoStore객체와 Video객체가 있으면, VideoStore객체는 Video객체를 가지고 있으므로, VideoStore객체가 Video객체를 생성한다.  2) Information Expert  특정 동작을 수행할 수 있는 정보를 가지고 있는 class에 해당 역할을 부여한다.  ex) VideoStore와 Video객체가 있을때, 모든 Video의 제목을 가지고 와야 하는 동작은 VideoStore가 모든 Video정보를 가지고 있으므로 VideoStore에 역할을 부여한다.  3) Controller  UI layer 에서 오는 요청 ( 즉 사용자의 요청 ) 을 처리할 객체를 따로 만들어라. 이를 Controller객체라고 할 수 있다.  만일, subsystem안에 있는 각 객체의 기능을 사용할 때, 직접적으로 각 객체에 접근하게 되면, subsystem과 외부간의 coupling이 증가되고,  subsystem의 수정이 필요한 경우, 외부에 까지 영향이 미친다. sub system을 사용하는 입장에서 보면, 이 controller 객체만 알고 있으면 되므로 사용하기 쉽다.  4) Low Coupling  Low Coupling이란, 특정 element의 변경이 dependent를 가지는 다른 element에 적게 영향을 주는 것을 말한다.  object-oriented syste은 각 객체들과 그들 간의 상호작용을 통해 동작하므로, 각 객체들 사이에 coupling이 존재하지 않을 수는 없다.  이 pattern은 요구사항은 충족시키면서도 각 객체들, sub system간의 coupling을 낮은 수준이 되게끔 design하라는 원칙이다. ( class의 연관관계를 가능하면 하나의 class와만 관계하도록 모아라 )  Low coupling은 각 객체, sub system의 reusability, maintainablity를 높인다.  ex) RentVideo라는 class가 VideoStore와 Video객체와 각각 association관계에 있다면, 이를 VideoStore와만 의존성을 가지도록 한다.  5) High Cohesion  각 객체는 밀접한 연관성을 가진 동작들로만 이루어 져야 한다는 원칙으로,  High Cohesion으로 이루어진 객체는 다른 객체를 참조하는 일이 적을 것이므로 Low coupling과도 연관성이 있다.  6) Polymorphism  객체의 종류에 따라 behaviour이 변경된다면, 객체의 종류를 check하는 조건문을 상요하지 말고, object-oriented의 polymorphism을 사용하라는 원칙이다.  7) Pure Fabrication  Information Expert pattern을 적용시, Low coupling과 High Cohesion의 원칙이 깨진다면, 기능 적인 역할을 한 곳으로 모아라.  예를 들어, 로그 정보를 기록하는 역할을 생각해 보면, 각 정보는 각각의 객체에 있으므로, Information Expert 원칙을 적용하는 것이 맞다.  그러나, 이럴 경우, 모든 객체가 log를 저장하는 database 객체와 dependency를 가지게 되므로 log coupling 의 원칙이 깨지게 된다.  이럴 경우에는 공통적인 기능을 제공하는 역할을 한 곳으로 모아서 가상의 객체, sub system을 만드는 것이 효과적이다.  8) Indirection  두 객체 사이의 직접적인 coupling을 피하고 싶으면, 그 사이에 다른 객체를 사용해라. 여기서 말하는 다른 객체란 주로 interface인 경우가 많다.    9) Protected Variation  변경될 여지가 있는 곳에는 interface를 정의해라 |

## Design Patterns

|  |  |
| --- | --- |
| **Design Patterns** | |
| Design Pattern | 디자인 패턴이란 객체 지향 소프트웨어 설계에서 자주 발생했던 상황에 대한 문제에 대해 반복적으로 재사용할 할 수있는 해법들입니다 |
| Architecture style vs. design pattern | 아키텍쳐 스타일은 비즈니스 품질속성을 반영한 제약사항의 집합이고, 디자인 패턴은 개발관점에서 설계에 대한 문제 해결 방식이다 |
| Principles of Design Patterns | * Interface Separated from Implementation * Substitution with Various Implementations * Open Close Principle (OCP)   확장에는 open되어 있고 변경에는 close되어 있음 |
| **Q:** What are the benefits of OCP? | 기존 구조를 유지하고 기능 추가가 가능하므로 유지보수성이 좋다. |
| **Q:** What are the mechanisms for implementing OCP? | 상속(Inheritance/ Specialization), 위임(Delegation), Dynamic Binding, Abstraction with Interface  Strategy 패턴을 예로 설명할 수 있어야 함 – Context는 Closed, Strategy는 Open. |
| Classification of Design Patterns   * Creation Patterns * Structural Patterns * Behavioral Patterns | * 하단 추가 “Design Pattern 상세” 참고. |
| For each Pattern   * Situation * Structure in Class Diagram * Collaboration * Pros and Cons | 하단 추가 “Design Pattern 상세” 참고. |
| ★ **Q:** When would use <X pattern>? | 하단 추가 “Design Pattern 상세” 참고. |
| **Q:** What is the key benefit of using <X pattern>? | 하단 추가 “Design Pattern 상세” 참고. |
| Common OO Mechanisms found in Design Patterns | * Abstraction with Interface * Inheritance / Specialization * Dynamic Binding * Delegation |
| ★ **Q:** Explain how OO mechanisms are utilized in <X pattern>? | 위 항 내용 참고. |
| Advanced Topics | * Similarity between Patterns * Difference between Patterns * 하단 내용 참고 |
| ★ **Q:** What are the similarity & differences between <X pattern> & <Y pattern>? | 하단 추가 “Design Pattern 상세” 참고 |

## Requirement Engineering

|  |  |
| --- | --- |
| **Requirement Engineering** | |
| Definition of SW Quality | 정의: 비기능적 요구사항이며 SW의 기능에 제약사항을 가하는 것임. 이를 달성하기 위해서 SW architecture가 적용됨 |
| Types of Quality Aspects   * Process Quality * Internal Quality * External Quality * Quality in Use | Process Quality: 방법론, 절차  Internal Quality: 설계 품질  External Quality: 구현 품질  Quality in Use: 사용자 품질 |
| **Q:** Why is *Process Quality* important? | 방법론이 모든 개발 프로세스 및 제품의 품질로 이어지기 때문에 중요함 |
| **Q:** Why is *Quality-in-Use* important? | 제품의 목적과 사용자의 Context에 따른 요구 기대 부합성(예: 효율성, 즐거움) 🡪 최종 품질이므로 중요함. |
| ★ Quality Model of ISO 9126   * 6 Main Factors and Sub-Factors for Each. | **유이 기사 효신**  유지보수성 – 고치기 쉬운가? (분석성, 변경성, 안정성, 시험성)  이식성 – 다른 환경으로 쉽게 바꿀 수 있는가? (환경적응성, 설치용이성, 치환성)  기능성 – 필요한 기능을 제공하는가 (적절성, 정확성, 상호운용성, 보안성)  사용성 – 쓰기 쉬운가? (이해용이성, 운영성, 기호성)  효율성 – 사용되는 자원의 양에 따라 요구된 성능을 만족하는가? (시간효율성, 자원효율성)  신뢰성 – 믿을 만 한가? (성숙성, 결함수용성, 복구용이성)  ISO 9126ì ëí ì´ë¯¸ì§ ê²ìê²°ê³¼  https://mblogthumb-phinf.pstatic.net/MjAxNzExMTBfMTAx/MDAxNTEwMjcxNDY1NTYw.8qKMeEmPj83uf-SwoYmLdxYJY5XCCX3pTGcRZdVHABMg.uwduQiLThOFFsWu1Jm7NL1NMRSnmjYzKF1Ic0SGYa50g.PNG.suresofttech/image.png?type=w800  - 시간 관점의 Efficiency  - Resource 관점의 Efficiency |
| **Q:** What are the quality factors which are important in developing <X type of system>? | TBD |
| Process of Requirement Engineering | 요구사항 추출 🡪 요구사항 분석 🡪 요구사항 명세 🡪 요구사항 검증 의 process를 가지며 요구사항 변경을 대비한 요구사항 변경관리가 전 과정에서 이루어짐 |
| Agile Constructs for Requirement Management | Prototyping과 Iteration.  매 sprint의 산출물인 prototype을 통해 요구사항의 정제가 이루어짐  요구사항 오해에 대한 해결 과정이 있음 |
| **Q:** What are the *Agile* constructs which assist requirement engineering? | 위 항 참조. |
| Requirements for Platform Software | 개발자가 stakeholder이고 여러 system에서 여러 application개발에 사용되어야 하기 때문에 이식성과 유지 보수성이 가장 큰 품질 속성이 될 것임. |
| **Q:** How do you define/refine requirements for *Platform Software*? | 위 항 참조. |
| Non-Functional Requirements (NFRs)   * Architectural Requirements | 비기능 요구사항. 주로 품질 요구사항임. |

## Architecture Styles

|  |  |
| --- | --- |
| **Architecture Styles** | |
| Batch Sequential  <<Data Flow>> | **Situation:**  전체 Data를 순차적으로 처리할 필요가 있을 때 사용.  **Structure with Components and Connectors:**  전체 data의 묶음이 하나의 sub system에서 다른 sub system으로 이동하는 data flow architecture style.  Data의 transformation은 하나의 subsystem에서 처리가 완료된 이후에만 가능.     * Components: Programs and Data stores * Connector: one way pipes that transfer bulk data sets.   **Pros and Cons:**  장점:   * Subsystem간의 simple division. * 하나의 subsystem은 stand-alone program일 수 있다.   단점:   * External Control이 필요함. * Interactive Interface를 제공하지 않음. * No concurrency & Low throughput * High latency |
| Pipe-n-Filter  <<Data Flow>> | **Situation:**  실시간으로 들어오는 Streaming data의 처리할 때 사용됨.  **Structure with Components and Connectors:**  각각의 process step을 filter로 encapsulation되어 있고, 인접한 filter들은 pipe로 연결됨.    Components:  Data stream: stream of bytes, characters or any other type.  Filter: 독립적인 데이터 처리모듈 (transform/process). Active Filter/Passive Filter.  Pipe: Data를 한 Filer로부터 다른 Filter로 이동시키는 모듈  **Pros and Cons:**  장점:   * Concurrency & High Throughput * Reusability: Filter의 재사용 * Modifiability: Pipe의 interface만 변경되지 않으면, Filter 내부의 변경은 자유롭다. * Flexibility: sequential and parallel execution 이 가능   단점:   * Dynamic integration은 어렵다. * Dynamic Configuration은 어렵다. * Filter들 사이의 data transformation이 overhead일 수 있다. |
| Layered  <<Hierarchical>> | **Situation:**  효율적인 개발 및 유지보수를 위하여 하나의 system을 여러 layer로 나누는 architecture style이다. 대부분의 중/대규모 application 개발에 적용되며 각각의 layer은 독립된 R&R을 가진다. 일반적으로 presentation layer, business logic layer, data access layer로 나뉜다.  **Structure with Components and Connectors:**  System은 Higher and Lower Layer로 계층적으로 나뉘어지고 각 Layer는 연관된 class들로 구성되며, 각자 고유의 책임과 역할을 갖는다.  각 Layer는 하위 Layer에 대해서만 Dependency를 갖는다.  Ex) OSI 7-Layer model.  **Pros and Cons:**  장점:   * 각 layer의 독립성이 증가하여, interface의 변경이 없다면, low layer의 변경이 상위 layer에 영향을 미치지 않는다. * Flexibility, interchangeability, reusability - interface에 의해 분리 * Portability: 각 layer는 독립적으로 abstract machine이 될 수 있다   단점:   * Low Runtime Performance. * Layer간 호출 Rule이 지켜지지 않을 경우, Deadlock 등의 문제가 발생한다. * Exception and error handling issue - 특정 layer의 fault는 모든 calling layer에 영향을 미친다. * Layer를 나누기가 어렵고 대다수의 Application은 Layer 구조를 적용하기 힘듦. |
| MVC  <<Interaction-oriented>> | **Situation:**  사용자와의 interaction이 필요한 GUI 기반의 application의 개발을 위한 Architecture Style로 View, Model, Control 3가지 component로 구성되어 있다.  **Structure with Components and Connectors:**  mvcì ëí ì´ë¯¸ì§ ê²ìê²°ê³¼  Components:   * View: 사용자가 보게 되는 화면이며, Model로 부터 결과 값을 받아와서 출력하는 역할을 한다. * Model: Control로부터 전달된 명령을 통해 data를 저장하거나, View에게 결과 결과를 보여주게 한다. * Control: User와 직접적으로 상호 작용하는 부분으로, Model로 명령어를 보낸다. 이를 통해서 model의 상태를 변경 시키게 된다.   동작 sequence는 사용자의 입력이 controller에게 전달되면, controller는 이를 model에게 전달하여 상태를 변경하라고 요청한다. Model은 상태를 변경하고 이를 view에게 알린다. View는 Model에게 상태 변경의 결과를 요청하고 이를 받아 화면에 표시한다. (Controller에서 View에게 상태 변경의 표시를 요청할 수 도 있다.)  **Pros and Cons:**  장점:   * 많은 MVC vendor toolkit이 존재하여 개발이 용이하다 * 한 Data model에 여러 View를 제공할 수 있다. * 다른 모듈의 변경 없이 Interface를 변경할 수 있다.   단점:   * Agent-oriented application에는 적용이 어렵다. * 많은 Model과 Control이 제공되는 경우, Data model 수정 비용이 매우 크다. * Model과 View의 dependency를 완전히 없앨 수 없다. * Test Code작성이 어렵다. (UI 위주의 test code를 작성해야 함) |
| Blackboard  <<Data-centered>> | **Situation:**  가능한 해결 전략이 알려지지 않은 문제의 해결을 위한 Architecture Style이다. 공통 데이터에 대해 협업 가능한 각각의 Program들을 모아서 동작한다.  **Structure with Components and Connectors:**    Components:  - Knowledge source: Black Board로부터 데이터를 가져와서 가공 후, 다시 Black Board에 저장. system이 언제 멈춰야 하는지, final result가 무엇인지를 결정.  - Control: Blackboard의 change를 monitor 하고 다음 action이 무엇인지를 결정.  - Blackboard: 현재 Blackboard상의 solution이 유효한지 check하며, data를 유지**.**  **Pros and Cons:**  장점:   * Scalability: Knowledge Source의 add / update가 쉽다. * Concurrency: 모든 knowledge source는 parallel 하게 서로 독립적으로 동작한다. * Reusability Concurrency   단점:   * Knowledge source와 blackboard의 close dependency로 blackboard 의 변경이 모든 knowledge source에 영향을 준다. * Partial solution or approximate solution이 나올 수 있으므로 언제 process를 종료 할지 결정하기 어렵다. * Data sharing으로 synchronization issue가 있을 수 있다 |
| Shared Repository  <<Data-centered>> | **Situation:**  Data processing을 위한 user interaction을 제공하는 data-centered architecture이다. 서로 communicate하고, 비교적 크고 변경되는 Data를 exchange하는 것이 필요한 여러 Software Component로 구성된 Software System에 적합. DB나 compiler system에 많이 사용된다.  **Structure with Components and Connectors:**     * Components:   + Agent: Repository를 사용/변경. Repository에 직접 접근 함.   + Shared Repository: 데이터 저장소. * Connector: Data Repository를 Communication에 이용.   **Pros and Cons:**  장점:   * Data integrity : Easy to backup and restore * System scalability and reusability of agents: component간 direct communication이 없어서 새로운 component의 추가가 용이하다   단점:   * Reliability, availability 가 중요함 * Data structure, data store 와 agent가 높은 dependency를 가진다. * 분산 system일 경우 network 비용 추가 |
| Micro Kernel  <<Hierarchical>> | **Situation:**  Plug-in architecture로 Core system과 Plug-in Component로 구성된다. Core System의 경우, Plug-in component들의 integrate와 collaborate를 위한 최소한의 기본적인 functionality만을 제공한다. Plug-in component는 system functionality에 중요한 part를 제공한다. Eclipse, Internet Browser에 사용된다.  **Structure with Components and Connectors:**     * Components:   + Microkernel: central service들을 구현한다. System specific한 dependency를 encapsulation한다.   + Internal Server: microkernel이 제공하는 기능을 확장한다.   + External Server: 자신이 제공하는 Application Domain의 기능을 microkernel의 기능일 이용하여 구현한다.   + Adapter: Client와 External Server와의 Interface. Client에게 Portability 제공.   + Client: Service 이용.   **Pros and Cons:**  장점:   * Portability, Flexibility, Separation of policy and mechanism * Distributed Microkernel Benefits   + Scalability, Reliability, Transparency, Performance, Complexity of design and implementation   단점:   * Plug-in들의 communication에 dependency가 고려 되어야 한다. |
| Micro Service  <<Distributed>> | **Situation:**  Application을 상호 독립적이고 최소 규모의 구성 요소로 분할 하여 system을 구성하는 architecture이다. 각각의 service는 별도의 process에서 실행 되며, HTTP등 가벼운 mechanism으로 통신하여 하나의 application을 만든다. 하나의 micro service는 각각 business기능을 담당하고, 독립적으로 배포 가능해야 한다.  **Structure with Components and Connectors:**  https://microservices.io/i/Microservice_Architecture.png  **Pros and Cons:**  장점:   * Extensibility * 개발 및 배포 cycle관리 용이 * 각 service에 최적화된 방법/언어 사용가능 * 타 서비스와의 종속성이 없으므로 서비스 변경이 쉬움   단점:   * 운영 및 test 복잡도 증가 * 서비스 간의 외부 호출로 인한 비용증가 * 배포와 운영을 자동화하지 않으면 기존 방식 보다 더 어려움 |
| Dispatcher  <<Distributed>> | **Situation:**  Load balancer architecture style 이다.  Distributed computing에서 사용되는 middleware architecture로 server farm을 client에게 하나의 server처럼 보이게 한다.  Server는 high scalability와 high availability를 이룰 수 있다.  Availability: Uptime (fault 이후 restart task하는데 걸리는 시간)으로 정의됨. Uptime동안에는 계속 서비스가 제공되어야 하는 Quality로 Dispatcher는 다른 Server에게 Take over 하면서 Availability를 높인다.  Scalability: 서비스하는 client의 수와 상관없이 동일한 response time을 보장해야 되는 quality로 high scalable system은 HW Resource를 추가함으로써 이에 비례하여 throughput이 증가하면 scalability가 좋다고 한다. Dispatcher는 request를 여러 Server에 Distribution함으로써 좋아진다.  DNS based load balancing, TCP/IP server load balancer에 많이 사용된다.  **Structure with Components and Connectors:**     * Components:   + Dispatcher(Load Balancer): based on availability/load, dispatcher decide which server should handle the request and forwards it to the selected server   + Client:   + Server: Server provide information about availability/load   **Pros and Cons:**  장점:   * High scalability * High availability   단점:   * Load balancer에 부하가 걸릴 경우, low fault tolerant |
| Broker  <<Distributed>> | **Situation:**  distributed computing환경에서, Server와 Client간의 통신을 coordinate하기 위한 middleware architecture이다. Distributed software system에서 decoupled된 component들 간의 통신을 coordination하기 위해서 자주 사용된다. Broker의 역할은 Request가 오면 적당한 server에 locating하고, Server에 request를 forwarding하고, client에게 result를 transmit하는 역할을 한다.CORBA, RPC에 사용된다  **Structure with Components and Connectors:**     * Components: * Servers: Register with broker. Present method interfaces to client. * Client: Access server’s method via broker. Use same form to call server’s method. * Broker’s task: * Locating appropriate server * Forwarding Request to server * Transmitting Results and exceptions to client.   **Pros and Cons:**  장점:   * Transparency of Server Implementation and Location. * Changeability and Extensibility * Client에게 Server access를 단순화 시켜줌 * Reusability * Server component의 runtime change * Heterogeneous System들을 같이 운용할 수 있음   단점:   * Proxy overhead * Low fault-tolerance * Difficulty in testing due to the amount of proxies. |
| Process Control  <<Data Flow>> | 데이터 처리를 위한 Cycle이 존재하고, 목표 값이 존재할 때.  Controller: 변화를 계산하고 처리. 목표 값을 관리함.  Process: 데이터를 처리. |
| Master-Slave  <<Hierarchical>> | Load를 다수의 slave들에게 분산하여 병렬처리 하도록 하는 구조  Reliability가 중요한 시스템이 이용  Slave가 독립적이므로 공유되는 상태가 없다 |
| Virtual Machine  <<Hierarchical>> | 기 존재하는 system외에 Virtual Abstraction제공.  장점: Portability.  단잠: low performance. |
| Plug-In  <<Hierarchical>> | Core system과 plugin component로 이루어져 있음  (Core system 🡪 시스템을 동작시킬 수 있는 최소한의 기능으로 이루어짐)  동적으로 plugin 설치로 인해 기능 확장 가능  Downloadable Application  Portability, Flexibility 에 강점 |
| PAC  <<Interaction-oriented>> | Presentation - Abstraction - Control, MVC의 variation  Tree기반, Agent기반으로 이루어진 시스템에서 사용되는 architecture 스타일이며 Agent가 PAC 구조로 되어 있다.  Controller가 다른 agent (또 다른 PAC 구조)와 상호작용하는 역할, MVC에서는 M이 V의 출력내용을 업데이트 하는데 PAC에서는 P와 A가 통신하기 위해서는 C를 거쳐야 함.  MVC에서는 C🡪M🡪V / PAC에서는 C🡪A, C🡪P  Multi-tasking, Multi-viewing에 좋음. / Agent의 재사용성, 확장성  Agent 간 통신에 의한 overhead |
| Non-buffered Event-based Implicit Invocations  <<Implicit Asynchronous Communication>> | Publisher-subscriber간의 이벤트 전달 방법. |
| Buffered Message-Based  <<Implicit Asynchronous Communication>> | Message Queue를 통해 이벤트를 전달.  장점: 익명성, 동시성, 확장성  단점: Message Queue의 용량 이슈. |
| Client-Server  <<Distributed>> | 시스템이 서버와 클라이언트로 구성되어 책임을 분배하는 구조  다수의 클라이언트와 하나의 서버  여러 클라이언트가 생길 수 있을 때  서버의 재사용  무거운 연산을 서버에서 수행해야 할 때  서버 상태가 기능에 영향을 줄 수 있다.  클라이언트가 공통 서버로부터 제공된 공통 기능을 사용할 때  Separation of responsibility  Security 이슈  Server의 availability, reliability 이슈 |
| N-Tier  <<Distributed>> | 시스템이 다수개의 tier로 구분. Tier는 하나의 독립된 computational HW  Middle-tier의 존재로 인하여 Server Client 구조보다 reusability 및 scalability 향상  테스트 어려움.  여러 개의 서버를 적용함으로써 availability, reliability 이슈. 네트워크 트래픽 |
| Peer to Peer  <<Distributed>> | Peer node끼리 통신하는 구조. Controller가 없음  하나의 peer가 서버이자 클라이언트  탈 중앙화된 컴퓨팅 지원, 특정 Node의 장애에 강함, Scalability이 매우 뛰어남  Central point가 존재하지 않아 제어가 어려움  모든 node들의 HW 성능이 동일하지 않음  Security 이슈 (바이러스 전파) |
| Proxy  <<Distributed>> | Client-Server통신 시 Pre-processing/Post-processing이 필요한 경우.  장점: 원본에 원하는 처리 가능. 성능 (Cache)  단점: 잠재적 과잉 처리. Indirection 증가. |
| RESTful  <<Distributed>> | REST: Representational State Transfer.  Internet과 Computer System간의 정보처리 상호 운용성을 제공하기 위한 구조.  Rest-compliant web service는 web resource에 대한 text기반 요청을 허용한다. |
| Active Repository | Shared repository의 변형으로 repository가 능동적으로 객체의 상태를 모니터링 하다가 변화를 알려주는 구조. |
| Publish-Subscribe | Publisher와 Subscriber, Event broker로 구성, Loose coupling, 비동기 동작  State가 바뀌면 그 사실을 통보하고 sub은 그 새로운 state를 갖고올 수도 안 갖고 올 수 도 있다.  Broker에서 filtering이 이루어지고 Asynchronous하게 동작한다는 점에서 observer 패턴과 차이가 있음  최적설계 때 배운 buffered message based 방식이 이거인 듯  중간에 queue, topic 같은 data 저장 공간이 있기 때문에 publish 시점에 data를 받지 못하더라도 나중에 받을 수 있음  Queue / Topic 사이즈가 한계가 있는 게 단점  Observer 와 subject 사이의 연결이 있지만, pub-sub은 그렇지 않음.  Message Broker가 존재 |
| Event-Driven | Event source는 event를 event space로 보내고 event sink는 event space로부터 event를 받아서 처리함. Loose coupling, 비동기 동작  Event sink가 event source에 자신을 등록하는 형태 (event sink = event listener)  최적설계 때 배운 non-buffered event based 방식이 이거인 듯  모든 메시지가 동일한 event space를 통해 전달 되기 때문에 scalability 이슈가 발생할 수 있음  == Event-Bus.  Event processor, event bus, event |
| Sensor Controller Actuator | Process-Control Architecture  Sensor로부터 수신된 이벤트를 Controller가 해석하여 이에 대한 처리를 Actuator에게 전달하는 구조  Embedded 시스템에 많이 사용됨 |
| Understanding Architecture Styles   * Situation * Structure with Components and Connectors * Pros and Cons | 위 각 Style별 내용 참조. |
| **Q:** When would you use <X architecture style>? | 위 항 참조. |
| **Q:** What are the pros and cons of using <X architecture style>? | 위 항 참조. |
| **Q:** Explain the mechanism of <X architecture style> | ※ 메커니즘 설명 방식  1) 구조 언급  2) 각 구조의 역할 설명  3) 실행 순서 언급 |

## Skeleton/Core Architecture

|  |  |
| --- | --- |
| **Skeleton/Core Architecture** | |
| How to evaluate the Applicability of Architect Styles?   * Situation, Pros and Cons | 각 Style의 활용 Situation과 타겟 시스템의 요구사항, 특징을 비교  SRS와 시스템의 Context Model을 비교에 이용  시스템에 활용될 수 있는 경우, 후보 Architecture 스타일로 선정 |
| How to Derive Skeleton Architecture from Style?   * Instantiating each selected Style * Merging Styles | 아래 항 참조. |
| **Q:** How do you merge multiple architecture styles? | 시스템 전체 구조를 가장 크게 좌우할 style을 기반 구조로 선택하고 여기에 다른 style들을 적용하는 방식으로 진행함  Style별 component와 connector 정의  Style별 공통 component와 connector 를 찾아서 이를 이용하여 병합  Style별 공통부가 없다면 application-specific connector로 연결  Style간 충돌 있다면 둘 중 하나만 선택 |
| Defining Interactions among Components in the Architecture | Component간의 interaction은 함수 호출, 데이터 flow, event, 포함관계가 있을 수 있으며 이러한 interaction은 architecture에 영향을 미칠 수 있기 때문에 이를 정의하는 것이 중요함. |
| **Q:** Why we do define *Interactions* among components? | 위 항 참조. |

## Applying Viewpoints

|  |  |
| --- | --- |
| **Applying Viewpoints** | |
| Essential Viewpoints   * Functional * Information * Behavior * Deployment Views | Functional Viewpoint:   * Use Case diagram으로부터 functional component 추출 * Functional component의 interface 추출 * Component간 interaction 정의 * Architecture에 functional component 배치   Information Viewpoint:   * Class diagram으로부터 data component 추출 * Object Relation Mapping 🡪 Data 접근을 위한 method 정의 (CRUD method) * Mapping Inheritance * Architecture에 data component 배치   Behavior Viewpoints:   * Whole system을 모델링하기 위한 activity diagram * (Action, Activity, Fork-Join, Decision Node, Swim-lane, Data Node, Object Node) * Use Case로부터 activity를 추출하여 순차, 병행 방식으로 어떻게 수행되는지 workflow를 명시   Deployment Views   * Deployment diagram 🡪 HW 스펙이 명시되는 diagram * Node 🡪 독립된 computing HW device * Artifacts 🡪 SW |
| Guidelines for each View | 위 항 참조. |
| **Q:** What is the procedure of applying <X viewpoint>? | 위 항 참조. |
| **Q:** View간의 일관성 문제 출제 가능성 있음 | *Functional-Behavior: Functional간의 호출이 Behavior 에 정확하게 표현?*  *Information-Deployment: Information Node가 Deploy에 정확하게 표현?* |
| Behavior vs Functionality | Functionality는 시스템이 “어떤” 기능을 제공하느냐에 대한 것이고 Behavior는 시스템이 제공하는 기능이 “어떻게” 이루어지느냐에 대한 것이다. |

## Applying Tactics for NFR

|  |  |
| --- | --- |
| **Applying Tactics for NFR** | |
| Architecture Design for Conventional NFRs   * Tactics for Common NFRs: Available | Modifiability   * Localize Modification: 기능을 나누고, Module을 일반화 시켜라 * Prevent Ripple Effects : Interface * Defer Binding time : Polymorphism   Efficiency ( Performance )   * Resource Demand: event rate조절 * Resource Management: 병렬 처리, data 다중화 * Resource Arbitration * Caching * Queueing * Polling   Availability   * Fault Detection : Ping & Echo * Fault Recovery: redundancy 사용, state 재 동기화, Rollback * Fault Prevention : process monitor   Security   * Resisting Attacks : authenticate users , authorize users , maintain data confidentiality , maintain integrity , limit exposure , limit access * Detecting Attacks * Recovering from attacks   Usability   * Separate User Interface : MVC * Support User initiative: Cancel , Undo 제공 * Support system initiative   Testability   * Input/Output: Interface와 Implementation을 분리 * Internal Monitoring : built-in monitor |
| Architecture Design for Non-Conventional NFRs   * Tactics to be devised by Architecture |  |
| Tactic for High Maintainability | High cohesion, Low coupling  Localize Modification 🡪 최소한의 모듈 변경만으로 기능 추가, Ripple effect 최소화  Coherence 유지, 모듈 일반화 (interface, abstraction)  정보 은닉, interface 유지, Interaction path 제한, 매개체 (strategy, façade, bridge, proxy 등) 사용 |
| Tactic for High Reliability | Ping & Echo, Heartbeat, Exception (오류감지) 적용  Voting, Active Redundancy, Passive Redundancy,  Spare, State re-synchronization, shadow, checkpoint/rollback 등 |
| Tactic for High Performance | 연산 효율을 높이기 위한 수정  Overhead 감소  Event rate, 샘플링 rate 조정  스케줄링 조정  동시성 확보, 가용 자원 증가, 분산 처리 |
| Tactic for High Efficiency  Time-wise, Resource-wise | Time-wise: SW가 기능을 수행할 때, 적절한 응답시간, 처리효율, 처리 시간을 제공할 수 있는 능력에 영향을 미치는 속성 🡪 연산 효율 높임, overhead 감소, event rate, 샘플링 rate 조정, 분산 처리 등  Resource-wise: 효율적인 자원관리를 통해 성능 향상, SW가 기능을 수행할 때, 적절한 양의 자원을 소모하는 능력에 영향을 미치는 속성 🡪 가용 자원 증가, 동시성 확보 |
| Impact of Tactics on Views |  |
| **Q:** What will be effective tactics for <X conventional NFR>? | ※ Reliability, 성능, Efficiency, Modifiability. 2~3개 정도.  첫 항 참조. |
| **Q:** Justify <X tactic> proposed in your CEP final report? | TBD |

## Architectural Evaluation

|  |  |
| --- | --- |
| **Architectural Evaluation** | |
| ATAM | ATAM: Architecture Tradeoff Analysis Method   * 품질 속성 시나리오에 기반하여 Architecture tradeoff를 분석하고, 이에 대한 위험 요소를 찾아내는 Architecture 평가 방법 * 품질만 보며 비용/시간/인력등은 고려하지 않는다. * 9단계 평가   ①ATAM소개 🡪 ②비즈니스 동인 소개 🡪 ③Architecture 소개 🡪  ④Architecture접근법 식별 🡪 ⑤품질 속성 utility tree작성 🡪 ⑥Architecture접근법 분석🡪  ⑦Brain Storming과 시나리오 우선순위 결정 🡪 ⑧Architecture접근법 분석 🡪  ⑨결과 발표   * 장점: 품질 속성 별로 평가 가능하다. * 단점: COST등은 고려하지 않는다. |
| CBAM | CBAM: Cost Benefit Analysis Method   * Architecture를 기반으로 비용, 이익등을 분석하여 Architecture적 판단을 내리는 경제적 측면의 평가 방법 * 시스템 개발의 ROI (Return On Investment) 수준에서 Architecture 전략을 결정 * 시나리오 개발 🡪 우선순위 결정 🡪 시나리오별 효용 결정 🡪 시나리오 예상값에 따른 아키텍처 전략 수립 🡪 예상 효용 계산 🡪 아키텍처 전략에 따른 전체 이익 계산 🡪 비용을 고려한 ROI 계산 후 순위 조절 |
| SAAM | SAAM: Software Architecture Analysis Methods  시나리오 기반의 아키텍처 평가방법  유지보수성만 평가   * 시나리오 개발 🡪 우선순위 결정 🡪 직접/간접 시나리오 평가 🡪 시나리오 상호작용 평가 |
| Others |  |
| **Q:** What is the procedure of applying <X evaluation> method? | ATAM은 위 내용 참조.  CBAM/SAAM은 TBD. |

## Re-engineering

|  |  |
| --- | --- |
| **Re-engineering** | |
| Refactoring   * Process * Instructions | 정의: 기능 변화 없이, 이해하기 쉽고, 수정하기 쉽도록 내부 SW architecture를 변경하는 작업  Code Smell: TBD  Process:  Instructions   * Composing Methods: 메소드 구성 * Moving Feature between Objects: 객체 간 기능 이동 * Organizing Data: 데이터 정리 * Simplifying Conditional Expression: 조건식 단순화 * Simplifying Method Calls: Method 호출 단순화 * Dealing with Generalization: 일반화 다루기 |
| Re-engineering for Different Paradigms   * From C Programs to Object-Oriented Design | TBD |
| **Q:** How you re-factor <Y artifact> from <X artifact>? | TBD |

Design Pattern 상세 – 패턴 별 설명

## Creational Pattern

### Factory Method Pattern

|  |  |
| --- | --- |
| Description | Object를 생성하는 interface를 정의하고, 어떤 concrete class가 생성될지는 sub class에게 맡기는 design pattern |
| Situation | * 어떤 클래스가 자신이 생성해야 하는 객체의 클래스를 예측할 수 없을 때 * 객체 생성의 책임을 서브클래스 중 하나에 위임하고, 그 정보를 국소화하고 싶을 때. * Use the Factory Method when you want to provide users of your library or framework with a way to extend its internal components |
| Structure in Class Diagram | Product: 생성하는 객체의 Interface를 정의  Concrete Product: Product의 Interface를 실제로 구현.  Creator: Product 타입의 객체를 반환하는 factoryMethod를 선언.  Concrete Creator: factoryMethod를 재정의하여 Concrete Product의 Instance를 반환. |
| Collaboration | Creator는 자신의 서브 클래스를 통해 실제 필요한 factoryMethod를 정의하여 적절한 Concrete Product의 Instance를 반환함. |
| Pros and Cons | 장점:   * DIP : Dependency Inversion Principle * OCP : Open Closed Principle * Factory Method를 사용하여 객체를 생성하므로, Client는 실제로 어떤 객체가 생성되는지 모른다. (Security) * Factory Method의 변경으로 Client에 영향 없이 객체 생성을 변경 할 수 있다. ( Low Coupling )   단점:   * 추가적인 subclass가 필요하다. |
| Key Benefit | 사용자가 정의한 어떤 Concrete Product 클래스와도 동작할 수 있게 함. |
| Rule of thumbs | Factory Method: creation through inheritance. Prototype: creation through delegation.  Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize. |

### Abstract Factory Pattern

|  |  |
| --- | --- |
| Description | 구체적인 Class의 Specification을 모른 상태에서 정의된 Interface를 통해서 관련성 있는 object들의 family set을 생성하는 design pattern이다. 동일한 Look & Feel을 지원해야 하는 User Interface 의 생성에 사용된다 |
| Situation | * 객체가 생성되거나 구성/표현되는 방식과 무관하게 시스템을 독립적으로 만들고자 할 때. * 여러 제품 군 중 하나를 선택해서 시스템을 설정해야 하고 한번 구성한 제품을 다른 것으로 대체할 수 있을 때. |
| Structure in Class Diagram | Abstract Factory: 개념적 제품에 대한 객체를 생성하는 연산으로 Interface를 정의.  Concrete Factory: 구체적인 제품에 대한 객체를 생성.  Abstract Product: 개념적 제품에 대한 Interface를 정의.  Concrete Product: Factory가 생성할 객체를 정의하고 Abstract Product의 Interface를 구현.  Client: Abstract Factory/Abstract Product를 사용. |
| Collaboration | Concrete Factory 클래스의 Instance 한 개가 Runtime에 만들어짐.  Abstract Factory는 필요한 객체 생성 책임을 Concrete Factory에 위임. |
| Pros and Cons | 장점:   * DIP : Dependency Inversion Principle * OCP : Open Closed Principle * Factory Method를 사용하여 객체를 생성하므로, Client는 실제로 어떤 객체가 생성되는지 모른다. (Security) * Factory Method의 변경으로 Client에 영향 없이 객체 생성을 변경 할 수 있다. ( Low Coupling )   단점:   * 추가적인 class가 필요하여 code complexity가 올라간다. |
| Key Benefit | Concrete클래스를 분리하고, 제품군을 쉽게 대체할 수 있도록 함. |
| Rule of thumbs | Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects, Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation.  Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object.  Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype.  Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.  Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.  Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. |

### Builder Pattern

|  |  |
| --- | --- |
| Description | 복잡한 객체 생성 과정을 추상화 하여, 동일한 방법으로 다른 형태의 객체를 생성할 수 있도록 하는 design pattern이다 |
| Situation | * 복합 객체의 생성 과정과 표현 방법을 분리하여 동일한 생성 절차에서 서로 다른 표현 결과를 만들 수 있게 하고 싶을 때. |
| Structure in Class Diagram | Builder: Product객체의 일부 요소들을 생생하기 위한 Interface 정의  Concrete Builder: Builder Interface를 구현.  Director: Part를 합성.  Product: 생성할 복합 객체를 표현. |
| Collaboration | 사용자는 Director 객체를 생성하고, 이 객체를 자신이 원하는 Builder객체로 합성해 나감.  제품의 일부가 구축(built)될 때마다 Director는 Builder에 통보.  Builder는 Director의 요청을 처리하여 제품에 부품을 추가.  사용자는 Builder에서 제품을 검색. |
| Pros and Cons | 장점:   * 객체 생성에 필요한 인자의 의미 파악이 쉽다. * 여러 가지 형태의 생성자가 필요하지 않도록 해준다   단점:   * 추가적인 class가 필요하여 code complexity가 올라간다. |
| Key Benefit | TBD |
| Rule of thumbs | Builder가 composite을 만들 수도 있다. 즉, composite 객체의 생성을 Builder가 담당할 수도 있다. |

### Singleton Pattern

|  |  |
| --- | --- |
| Description | 하나의 객체 생성을 보장해 주는 design pattern이다 |
| Situation | * 클래스의 Instance가 오직 하나여야 함을 보장하고, 잘 정의된 access point가 필요할 때. * Subclass로 확장이 가능해야 하며, 해당 확장이 사용자 코드에 영향을 미치지 않아야 할 때. |
| Structure in Class Diagram | Singleton: Instance()연산을 정의하여, Singleton Instance로 접근할 수 있도록 함. Instance() 연산은 클래스의 static member 함수임. |
| Collaboration | 사용자는 Singleton 클래스에 정의된 Instance()연산을 통해서 singleton instance에 접근. |
| Pros and Cons | 장점:   * 불필요한 instance의 생성을 방지하여 memory 를 절약할 수 있다. * 두 번째 객체 생성 때부터는 속도가 빨라진다   단점:   * Violate SRP: Single Responsibility Principle (Class의 생성 관리, class 원래의 functionality 두 개의 responsibility 가 존재하게 된다. ) * Global variable과 유사한 개념으로 Singleton객체를 사용하는 여러 객체와 coupling이 형성된다. * Multi Thread 환경에서 사용될 경우, synchronization issue가 있다. |
| Key Benefit | TBD |
| Rule of thumbs | Facade objects are often Singletons because only one Facade object is required  State objects are often Singletons.  전역변수처럼 사용하게 되는 것에 주의를 해야함. 여러 class로부터의 coupling이 발생할 수 있음  There should be only one Singleton instance, whereas a Flyweight class can have multiple instances with different intrinsic states.  The Singleton object can be mutable. Flyweight objects are immutable. |

### Prototype Pattern

|  |  |
| --- | --- |
| Description | 원형이 되는 Instance를 사용하여 생성할 객체의 종류를 명시하고, 이렇게 만든 Prototype을 복사해서 새로운 객체를 생성. |
| Situation | * Instance화 할 클래스를 Runtime에 지정할 때 * Factory 클래스를 피하고 싶을 때. * *Use the Prototype pattern when your code shouldn’t depend on the concrete classes of objects that you need to copy.* * *Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects. Somebody could have created these subclasses to be able to create objects with a specific configuration.* |
| Structure in Class Diagram | Prototype: 자신을 복제하는데 필요한 Interface를 정의  Concrete Prototype: Clone연산을 구현.  Client: Prototype에 Clone을 요청하여 새로운 객체를 생성 |
| Collaboration | 사용자는 Prototype에 스스로를 복제하도록 요청. |
| Pros and Cons | 장점:   * Runtime에 새로운 제품을 추가/삭제할 수 있음. * 서브클래스의 수를 줄일 수 있음. * You can get rid of repeated initialization code in favor of cloning pre-built prototypes.   단점:   * Clone()함수의 구현이 어려움. |
| Key Benefit | TBD |
| Rule of thumbs | Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.  Prototypes are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. In this context, Prototype can avoid expensive "creation from scratch", and support cheap cloning of a pre-initialized prototype.  Prototypes are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. In this context, Prototype can avoid expensive "creation from scratch", and support cheap cloning of a pre-initialized prototype. |

## Structural Patterns:

### Adapter Pattern

|  |  |
| --- | --- |
| Description | 이질적인 interface를 가진 component들간의 상호 작용이 가능하도록 중간에서 adaptation을 해주는 design pattern이다. Class Scope의 Adapter 와 Object Scope의 Adapter가 존재하며, Class Scope의 Adapter는 다중 상속을, Object Scope의 Adapter는 delegation을 사용한다. |
| Situation | * 기존 Class를 사용하고 싶은데 Interface가 맞지 않을 때 * 기존 library를 재사용하고 싶은데, 이 라이브러리의 수정이 불가능할 때. |
| Structure in Class Diagram | Class Adapter:    Object Adapter:    Target: 사용자가 사용할 응용 분야에 종속적인 Interface를 정의.  Client: Target을 사용.  Adaptee: 적용할 기존 Interface를 정의.  Adapter: Target Interface에 Adaptee Interface를 적응시킴. |
| Collaboration | Client는 Adapter를 호출하고, Adapter는 이를 수행하기 위해 Adaptee의 메소드를 호출. |
| Pros and Cons | 장점: Client에게 implementation detail을 숨긴다. (Client는 adapter만 본다.)  단점: 추가적인 class가 필요하여 code complexity가 올라간다. |
| Key Benefit | TBD |

### Bridge Pattern

|  |  |
| --- | --- |
| Description | Abstraction계층과 Implementation계층을 분리한 design pattern이다. 두 계층은 runtime에 bound되며, 두 계층을 분리함으로, 서로 독립적으로 확장이 가능하며, client와 implementation이 decoupled 되어서 implementation의 변경이 client에게 영향을 미치지 않는다. |
| Situation | * Abstract계층과 Implementation계층 사이의 Dependency를 제거하고 싶을 때. * Abstract계층과 Implementation계층이 모두 독립적으로 확장되어야 할 때. |
| Structure in Class Diagram | Abstraction: Abstract 개념에 대한 Interface를 제공하고, Implementer에 대한 Reference를 관리.  Refined Abstraction: Abstraction을 확장.  Implementer: 구현 클래스에 대한 Interface 제공.  Concrete Implementer: 실제 구현 클래스. |
| Collaboration | Abstract클래스가 사용자의 요청을 Implementer 객체에 전달한다. |
| Pros and Cons | 장점:   * Platform independent code를 만들 수 있다. * OCP : Open Closed Principle * Client에게 implementation code를 숨긴다   단점:   * 추가적인 class가 필요하여 code complexity가 올라간다. |
| Key Benefit | TBD |
| Rules of thumb | Bridge, State, Strategy (and to some degree Adapter) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn’t just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.  You can use Abstract Factory along with Bridge. This pairing is useful when some abstractions defined by Bridge can only work with specific implementations. In this case, Abstract Factory can encapsulate these relations and hide the complexity from the client code. |

### Composite Pattern

|  |  |
| --- | --- |
| Description | Object의 계층 구조를 위한 design pattern이다. Client는 단일 object, 복합 object의 구별 없이 동일한 interface로 control가능하다. 이로 인해 programmer는 단일 object, 복합 object구별 없이 programming이 가능하며, 동일한 자료 구조를 상속받았으므로 class의 추가가 용이하다. |
| Situation | * 부분-전체의 객체 계통을 표현하고 싶을 때 * 사용자가 복합객체와 개개의 객체 사이의 차이를 모르게 하고 싶을 때. |
| Structure in Class Diagram | Component: 집합 관계에 정의될 모든 객체에 대한 Interface를 정의.  Leaf: 가장 말단의 객체, 즉 자식이 없는 객체. 가장 기본이 되는 객체의 행동을 정의.  Composite: 자식이 있는 구성요소에 대한 행동을 정의.  Client: Interface 사용. |
| Collaboration | 사용자는 Component 클래스 Interface를 사용. 요청 받은 대상이 Leaf 인스턴스이면 자신이 정의한 행동을 수행하고, Composite이면 자식 객체들에게 요청을 위임. 위임 전후에 다른 처리 수행 가능. |
| Pros and Cons | 장점:   * 복잡한 tree structure와 interact하는 client의 code를 간단히 해준다. * 새로운 component의 추가가 간단하다   단점:   * 지나치게 범용적인 Design이 될 수 있다. |
| Key Benefit | TBD |
| Rules of thumb | A Decorator is like a Composite but only has one child component. There’s another significant difference: Decorator adds additional responsibilities to the wrapped object, while Composite just “sums up” its children’s results.  You can use Iterators to traverse Composite trees.  You can use Visitor to execute an operation over an entire Composite tree.  CoR, decorator를 composite tree의 결과를 위해 사용할 수도 있다. |

### Decorator Pattern

|  |  |
| --- | --- |
| Description | Class를 변경 하지 않고, 추가적인 responsibility를 부여 할 수 있는 design pattern으로, 기능 확대를 위한 sub class 를 대체 할 수 있다. OCP가 잘 적용된 pattern이라고 할 수 있다. |
| Situation | * 다른 객체에 영향을 주지 않고 개개의 객체에 새로운 책임을 추가하고 싶을 때. * 상속으로 서브클래스를 계속 만드는 방법이 실질적이지 못할 때. (조합에 한계, 혹은 final) |
| Structure in Class Diagram | Component: 동적으로 추가할 서비스를 가질 가능성이 있는 객체들에 대한 Interface.  Concrete Component: 추가적인 서비스가 실제로 정의되어야 할 필요가 있는 객체.  Decorator: Component객체에 대한 Reference를 관리하면서 Component에 정의된 Interface를 만족하도록 Interface 정의.  Concrete Decorator: Component에 새롭게 추가할 서비스를 실제로 구현하는 클래스. |
| Collaboration | Decorator는 자신의 Component 객체 쪽으로 요청을 전달함. 요청전달 전 및 후에 자신만의 추가 연산을 선택적으로 할 수도 있음. |
| Pros and Cons | 장점:   * 상속보다 Flexible 하다.   단점:   * 많은 작은 class들이 생성된다. |
| Key Benefit | TBD |
| Rules of thumb | Chain of Responsibility and Decorator have very similar class structures. Both patterns rely on recursive composition to pass the execution through a series of objects. However, there are several crucial differences.  The CoR handlers can execute arbitrary operations independently of each other. They can also stop passing the request further at any point. On the other hand, various Decorators can extend the object’s behavior while keeping it consistent with the base interface. In addition, decorators aren’t allowed to break the flow of the request. |

### Flyweight Pattern

|  |  |
| --- | --- |
| Description | 자원을 공유하여 memory를 절약하는 design pattern이다. 자주 사용되는 자원은 한번 생성하여 pool에 넣어 놓고, 다른 component에서 요청시 새로 생성하지 않고 기존에 생성한 자원을 재 사용하도록 한다. |
| Situation | * 응용프로그램이 대량의 객체를 사용해야 할 때 * 객체의 수가 너무 많아져 저장비용이 너무 높아질 때. |
| Structure in Class Diagram | Flyweight: 동작해야 하는 Interface를 정의.  Concrete Flyweight: Interface를 구현하고 저장소를 정의.  Unshared Concrete Flyweight: 모든 서브클래스들이 공유될 필요는 없음.  Flyweight Factory: Flyweight 객체 생성/관리.  Client: Flyweight 객체의 부가적인 상태를 저장. |
| Collaboration | 본질적 상태는 Concrete Flyweight에 저장하고, 부가적인 상태는 사용자가 저장하거나 연산되어야 하는 다른 상태로 관리. |
| Pros and Cons | 장점:   * 메모리 절약.   단점:   * Flyweight검색을 위한 추가적인 CPU Time이 필요하다. * 추가적인 class가 필요하여 code complexity가 올라간다. |
| Key Benefit | TBD |
| Rules of thumb | Flyweight would resemble Singleton if you somehow managed to reduce all shared states of the objects to just one flyweight object. But there are two fundamental differences between these patterns:  There should be only one Singleton instance, whereas a Flyweight class can have multiple instances with different intrinsic states.  The Singleton object can be mutable. Flyweight objects are immutable. |

### Façade Pattern

|  |  |
| --- | --- |
| Description | Subclass들의 interface들을 사용하여 high level의 interface를 제공하는 design pattern이다. Client와 sub system을 decoupled 하여 sub system의 변경이 client에게 영향이 없도록 해준다. 또한 최소 지식의 원칙이 잘 적용된 pattern이다. |
| Situation | * 복잡한 서브시스템에 대한 단순한 Interface 제공이 필요할 때. * 구현 클래스와 사용자 사이에 너무 많은 종속성이 존재할 때. * 서브시스템을 계층화 시킬 때. |
| Structure in Class Diagram | Façade: 단순하고 일관된 통합 Interface를 제공. 사용자의 요청을 해당 서브시스템 객체에 전달.  서브시스템 클래스들: 실제 기능을 구현하는 클래스들. 이들은 Façade에 대한 아무런 정보가 없음. |
| Collaboration | 사용자는 Façade에 정의된 Interface를 이용해 서브시스템을 이용하고, 실제로 서브시스템 객체로 직접 접근하지 않음. |
| Pros and Cons | 장점:   * Client와 sub system의 decoupled.   단점:   * 모든 application class와 coupled된 god object가 될 수 있다 |
| Key Benefit | TBD |

### Proxy Pattern

|  |  |
| --- | --- |
| Description | 다른 object의 접근을 control하기 위해서 대리자 (or placeholder ) 를 두는 design pattern이다.  대리자 객체는 client의 요청을 처리한 후에 client에게 결과를 알려준다.대리자 객체는 client에게 request를 받은 후에 접근 제어 등의 추가적인 작업을 한 후 실제 real subject객체에게 이를 전달한다. Real subject가 request를 후에는 다시 proxy를 통해서 client에게 결과가 전달된다.  Proxy의 종류에 따라서, Remote Proxy, Virtual Proxy, Access(Protection) Proxy, Smart Reference 등이 있다. |
| Situation | * *다른 객체에 대한 Reference를 관리할 필요가 있거나 추가적인 작업을 수행할 필요가 있을 때(?)* * *Client 가 Reference에 대한 관리를 하지 않게 하고 싶을 때.* |
| Structure in Class Diagram | Proxy: 실제로 참조할 대상에 대한 Reference를 관리. Subject와 동일한 Interface를 제공.  3가지 종류가 있음(원격지 Proxy /가상의 Proxy /보호용 Proxy)  Subject: Real Subject와 Proxy에 공통적인 Interface 정의.  Real Subject: 실제 객체. |
| Collaboration | Proxy는 자신이 받은 요청을 Real Subject에 전달.. |
| Pros and Cons | 장점:   * 실제 service와 client를 분리하여 중간에 많은 작업을 할 수 있다.   단점:   * Proxy 로 인하여 response time이 증가 할 수 있다 |
| Key Benefit | TBD |

## Behavioral Patterns:

### Interpreter Pattern

|  |  |
| --- | --- |
| Description | 문법 규칙을 class화 하여, 문장을 해석하는 pattern이다. 이를 위해서는 client와의 구문 약속이 되어 있어야 하며, 구문 약속에 맞게 정의된 language는 interpreter에 의해서 해석이 되어야 한다. 문법 규칙이 class로 정의 되어 있으므로 추가가 쉬우나, 문법 규칙이 많아지는 경우에는 별도의 파서, 컴파일러 등을 사용하는 것이 좋다. |
| Situation | * 해석이 필요한 언어가 존재하거나 추상구문 Tree로서 그 언어의 문장을 표현하려 할 때. |
| Structure in Class Diagram | Abstract Expression: 모든 Node들이 공통적으로 가져야 할 Interpret() 연산을 정의.  Terminal Expression: 문법에 정의한 터미널 기호와 관련된 해석 방법을 구현.  Nonterminal Expression: Nonterminal Expression에 대한 구현.  Context: 번역기에 대한 포괄적인 정보를 제공.  Client: 구문 트리. |
| Collaboration | 사용자는 Terminal/Nonterminal Expression 클래스로 추상 구문 Tree를 만듦.  각 노트에 정의한 Interpret()연산은 해석자의 상태를 저장하거나 그것을 알기 위해서 문맥 정보를 이용. |
| Pros and Cons | 장점:   * 문법 규칙을 class로 구현하므로, 구현하기 쉬우며, 쉽게 추가가 가능하다   단점:   * 문법 규칙이 많아지면 복잡해 지고 무거워 진다. |
| Key Benefit | TBD |

### Chain of Responsibility Pattern

|  |  |
| --- | --- |
| Description | Request를 처리 할 수 있는 기회를 여러 객체에게 부여하여, client와 request 처리 객체를 decoupled한 design pattern이다. 일반적으로 request를 처리 할 수 있는 객체들을 chain ( linked list )등으로 묶고, request가 들어오면 제일 처음의 handler에게 요청한다. 각각의 handler는 자신이 처리 할 수 없는 request는 다음 객체에게 전달한다. |
| Situation | * 하나 이상의 객체가 요청을 처리해야 하고, 그 처리자 중 어떤 것이 선행자(priori)인지 모를 때. * 메시지를 받을 객체를 명시하지 않은 채 여러 객체 중 하나에게 처리를 요청하고 싶을 때. |
| Structure in Class Diagram | Handler: 요청을 처리하는 Interface를 정의하고, 후속 처리자 (Successor)와 연결을 구현.  Concrete Handler: 처리동작에 대한 구현. 자신이 처리할 것이 있으면 하고, 아니면 후속자에 요청.  Client: 요청을 보냄. |
| Collaboration | 사용자는 처리를 요청하고, 이 처리 요청은 실제로 그 요청을 받을 책임이 있는 Concrete Handler 객체를 만날 때까지 연결고리를 따라서 전달됨. |
| Pros and Cons | 장점:   * Handler의 추가 삭제가 용이하다. * Request의 sender와 receiver의 coupling을 줄인다. * SRP, OCP를 잘 따른다   단점:   * Chain을 잘 관리하지 않을 경우, 처리 되지 않는 request가 발생 할 수 있다. * Request가 언제 처리 될지를 가늠하기 어렵다 |
| Key Benefit | TBD |
| Rules of thumb | Chain of Responsibility, Command, Mediator and Observer address various ways of connecting senders and receivers of requests:  Chain of Responsibility passes a request sequentially along a dynamic chain of potential receivers until one of them handles it.  Command establishes unidirectional connections between senders and receivers.  Mediator eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object.  Observer lets receivers dynamically subscribe to and unsubscribe from receiving requests.  Chain of Responsibility is often used in conjunction with Composite. In this case, when a leaf component gets a request, it may pass it through the chain of all of the parent components down to the root of the object tree.  Handlers in Chain of Responsibility can be implemented as Commands. In this case, you can execute a lot of different operations over the same context object, represented by a request.  However, there’s another approach, where the request itself is a Command object. In this case, you can execute the same operation in a series of different contexts linked into a chain. |

### Command Pattern

|  |  |
| --- | --- |
| Description | Request를 encapsulation한 design pattern이다. Request를 encapsulation 할 때, 이를 처리할 수 있는 method, 매개 변수 등을 같이 encapsulation하여, 처리시간 및 logging, undo등이 가능하게 된다. Client는 command를 생성 시, 이를 처리 할 수 있는 receiver를 같이 넣어 주어야 한다. |
| Situation | * Sender와 Receiver의 coupling을 제거하고자 할 때. * 수행할 동작을 객체로 매개변수화 하고자 할 때. * 서로 다른 시간에 요청을 명시하고, 저장하고, 실행하고 싶을 때. * 실행 취소 기능을 지원하고 싶을 때 * 시스템이 고장 났을 때 재적용이 가능하도록 변경과정에 대한 로깅을 지원하고 싶을 때. |
| Structure in Class Diagram | Command: 연산 수행에 필요한 Interface를 선언.  Concrete Command: Receiver 객체와 액션 간의 연결성을 정의. 실제 Execute()를 구현.  Client: Concrete Command 객체를 생성하고 처리 객체로 정의.  Invoker: 명령에 대해 처리할 것을 요청.  Receiver: 요청에 관련된 연산 수행 방법을 알고 있음.  <https://gmlwjd9405.github.io/2018/07/07/command-pattern.html> |
| Collaboration | 사용자는 Concrete Command를 생성하고 이를 수신자로 지정함.  Invoker 는 Concrete Command를 저장하고, execute()를 호출하여 요청을 발생시킴.  Concrete Command는 요청을 처리할 실제 객체에 정의된 연산을 호출. |
| Pros and Cons | 장점:   * Operation을 invoke하는 class와 이를 수행하는 class가 decoupled된다. * undo 를 가능하게 한다. * Execution을 연기 할 수 있다. * OCP   단점:   * 추가적인 class가 필요하여 code complexity가 올라간다. |
| Key Benefit | TBD |
| Rules of thumb | Handlers in Chain of Responsibility can be implemented as Commands. In this case, you can execute a lot of different operations over the same context object, represented by a request.  However, there’s another approach, where the request itself is a Command object. In this case, you can execute the same operation in a series of different contexts linked into a chain.  You can use Command and Memento together when implementing “undo”. In this case, commands are responsible for performing various operations over a target object, while mementos save the state of that object just before a command gets executed.  Prototype can help when you need to save copies of Commands into history.  You can treat Visitor as a powerful version of the Command pattern. Its objects can execute operations over various objects of different classes. |

### Mediator Pattern

|  |  |
| --- | --- |
| Description | Request를 encapsulation한 design pattern이다. Request를 Encapsulation 할 때, 이를 처리할 수 있는 method, 매개 변수 등을 같이 encapsulation하여, 처리시간 및 logging, undo등이 가능하게 된다. Client는 command를 생성 시, 이를 처리 할 수 있는 receiver를 같이 넣어 주어야 한다. |
| Situation | * 여러 객체가 잘 정의된 형태이기는 하지만 복잡한 상호작용을 가질 때. * 한 객체가 다른 객체를 너무 많이 참조해서 재사용하기 힘들 때. * 여러 클래스에 분산된 행동들이 상속 없이 상황에 맞게 수정이 되어야 할 때. |
| Structure in Class Diagram | Mediator: Colleague 객체와 교류하는데 필요한 Interface를 정의  Concrete Mediator: Coordinate 수행. Colleague를 파악하고 관리.  Colleague 클래스들: 자신의 중재자 객체가 무엇인지 파악. |
| Collaboration | Colleague는 Mediator서 요청을 송수신함. Mediator는 필요한 Colleague에 요청을 전달. |
| Pros and Cons | 장점:   * M:N의 관계에서 1:M의 관계로 복잡도를 떨어뜨린다   단점:   * Mediator는 잘못하면 God Class가 될 수 있다. * Mediator는 재사용이 어렵다. |
| Key Benefit | TBD |

### Strategy Pattern

|  |  |
| --- | --- |
| Description | Algorithm 군을 정의하고 각각을 encapsulation 하여 algorithm을 교환 할 수 있는 pattern으로 client와 상관없이 algorithm을 변경 할 수 있다. Inheritance를 사용하는 경우, 상위 class에 정의된 method중에서 다른 algorithm이 필요한 경우, overriding을 하게 되며, 이는 객체의 수가 많아 질수록 burden이다. 또한 interface를 사용할 경우는 코드의 reuse가 사라진다. |
| Situation | * 행동들이 조금씩 다를 뿐 개념적으로 관련된 많은 클래스들이 존재할 때. * 알고리즘의 변경이 필요할 때. * 사용자가 몰라야 하는 데이터를 사용하는 알고리즘이 있을 때. |
| Structure in Class Diagram | Strategy: 제공하는 모든 알고리즘에 대한 공통의 연산들을 Interface로 정의.  Concrete Strategy: 실제 알고리즘을 구현.  Context: Concrete Strategy객체를 통해 구성됨. |
| Collaboration | Strategy 클래스와 Context 클래스는 의사교환을 통해 선택한 알고리즘을 구현 함.  Context 클래스는 사용자 쪽에서 온 요청을 각 전략 객체로 전달. |
| Pros and Cons | 장점:   * 코드의 중복을 줄일 수 있다. * Runtime시 Algorithm의 변경이 가능하다. * Extensibility가 좋다. * Client와 독립되어 있어서 algorithm의 수정이 가능하다. * Program to interface, not an implementation * Favor composition over inheritance * Encapsulation what varies   단점:   * 추가적인 class로 인한 code complexity 증가 * Client는 적당한 algorithm을 선택하기 위해 algorithm간의 차이를 알고 있어야 한다. |
| Key Benefit | TBD |
|  | Decorator lets you change the skin of an object, while Strategy lets you change the guts.  Template Method is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses. Strategy is based on composition: you can alter parts of the object’s behavior by supplying it with different strategies that correspond to that behavior. Template Method works at the class level, so it’s static. Strategy works on the object level, letting you switch behaviors at runtime.  State can be considered as an extension of Strategy. Both patterns are based on composition: they change the behavior of the context by delegating some work to helper objects. Strategy makes these objects completely independent and unaware of each other. However, State doesn’t restrict dependencies between concrete states, letting them alter the state of the context at will. |

### Visitor Pattern

|  |  |
| --- | --- |
| Description | 이미 구조적으로 완성된 class구조에 기능을 추가하고 싶을 때 사용되는 pattern이다. 이를 구현하기 위해서는 사전 작업이 필요하다. 즉, 기능 추가가 필요하다고 생각 되는 class에는 accept함수를 interface로 상속 시키고, visitor를 외부에서 정의하여 accept에 인자로 넣어 줌으로써 기능을 추가한다. |
| Situation | * 객체 구조가 서로 관련은 없지만 많은 종류의 Operation 적용이 필요할 때. * 객체 구조에 정의한 클래스는 거의 변하지 않지만, 전체 구조에 걸쳐 새로운 연산을 추가하고 싶을 때. * 다른 Interface를 가진 클래스가 객체 구조에 포함되어 있으며, 구체 클래스에 따라 달라진 연산을 이들 클래스의 객체에 대해 수행하고자 잘 때 * Use the Visitor when you need to perform an operation on all elements of a complex object structure (for example, an object tree). * Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others. |
| Structure in Class Diagram | Visitor: 객체 구조 내에 있는 각 Concrete Element클래스를 위한 Visit() 연산을 선언.  Concrete Visitor: Visitor에 정의된 Visit() 연산들을 구현.  Element: Visitor를 인자로 받아들이는 Accept 연산을 정의.  Concrete Element: Accept 연산을 구현.  Object Structure: 객체 구조 내의 원소들을 나열. |
| Collaboration | 사용자는 Concrete Visitor 클래스 객체를 생성하고 객체 구조를 따라서 각 원소를 방문하며 순회.  Visitor가 구성 원소들을 방문할 때, 구성 원소는 Visitor의 연산을 호출. |
| Pros and Cons | 장점:   * 복잡한 object structure에 새로운 operation을 간단히 추가할 수 있다. * 관련된 behavior를 하나의 class로 관리 가능하다   단점:   * Element의 hierarchy가 자주 변경된다면 적용이 어렵다. * Element객체가 추가 될 때 마다, visitor의 변경이 필요하다. * 최초 구조 잡기가 어렵다. * Visitor가 element의 private member에 접근할 수 없으므로 기능 구현에 제약이 있을 수 있다. |
| Key Benefit | TBD |
|  | You can use Visitor along with Iterator to traverse a complex data structure and execute some operation over its elements, even if they all have different classes.  You can use Visitor to execute an operation over an entire Composite tree. |

### Memento Pattern

|  |  |
| --- | --- |
| Description | Object의 상태를 기록해 놓았다가 원래 상태로 복원하고자 할 필요가 있을 때 기록해 놓은 것을 사용하는 pattern이다. 실행 취소 기능을 지원하고자 할 때, 많이 사용된다. Client는 Originator의 상태를 저장해야 할 때, originator를 통해서 memento를 생성하고 caretaker에 저장을 한다. 이후, 복원하고자 할 때는 caretaker에서 memento를 가지고 와서 originator의 setMemento 함수를 사용하여 복원한다. |
| Situation | * 어떤 객체의 상태에 대한 스냅샷을 저장한 후 나중에 이 상태로 복구해야 할 때. * 상태를 얻는데 필요한 Interface를 두는 것이 힘들 때 (정보 노출 위험) |
| Structure in Class Diagram | Memento: Originator 객체의 내부 상태를 저장  Originator: Memento를 생성하여 현재 객체의 상태를 저장하고 필요 시 복원함.  Caretaker: Memento의 보관을 책임지는 보관자. |
| Collaboration | Caretaker는 Originator에 Memento를 요청하여 저장.  Memento객체는 수동적이며, Memento를 생성한 Originator만이 상태를 설정하고 읽어올 수 있음. |
| Pros and Cons | 장점:   * Memento는 Client에게 노출되지 않는다   단점:   * Originator이 encapsulation이 약화 된다. ( 상태 정보를 memento와 공유 ) * Memento가 많아 질 수록 메모리 사용이 늘어난다. * Caretaker는 쓸모 없어진 memento의 삭제를 위하여 originator의 life cycle을 알고 있어야 한다. |
| Key Benefit | TBD |
|  | 실제 snapshot을 생성하고 다시 복원하는 것을 외부에서 진행한다면 그 필요한 정보를 노출시킬 수 밖에 없음. |

### Iterator Pattern

|  |  |
| --- | --- |
| Description | 내부 자료 구조에 대한 이해 없이 aggregator의 자료 구조에 접근할 수 있도록 해주는 design pattern이다. ( ConcreteAggregator에서 createIterator에서 concreteIterator를 생성하면서, 자신의 자료 구조를 넘겨 주어 ConcreteIterator에서 이를 이용하여 관련 함수를 구현, 이후 부터는 iterator를 통해서 자료 구조에 접근 가능하다. |
| Situation | * 객체 내부 표현 방식을 모르고도 집합 객체의 각 원소들에게 접근하고 싶을 때 * 집합 객체를 순회하는 다양한 방법을 지원하고 싶을 때 |
| Structure in Class Diagram | Iterator: 원소를 접근하고 순회하는 데 필요한 Interface를 제공.  Concrete Iterator: Iterator를 구현.  Aggregate: Iterator 객체를 생성하는 Interface를 정의.  Concrete Aggregate: Aggregate 구현. |
| Collaboration | Concrete Iterator는 집합 객체 내 현재 객체를 계속 추적하고 다음에 방문할 객체를 결정. |
| Pros and Cons | 장점:   * Aggregator이 코드를 단순화할 수 있다. * SRP   단점:   * 간단한 Collection에서의 사용은 Over이다. |
| Key Benefit | TBD |

### State Pattern

|  |  |
| --- | --- |
| Description | Object의 환경과 behavior를 묶는다. 즉, 현재 state에 따라 다른 behavior를 가능하게 하는 design pattern이다. State 객체의 interface에는 state machine의 모든 action을 위한 method를 가지고 있어야 하며, state machine으로 들어온 명령을 현재 state에게 delegation한다. |
| Situation | * 객체의 행동이 상태에 딸 달라질 수 있고, 객체의 상태에 따라서 런타임에 행동이 바뀌어야 함. * 어떤 연산에 그 객체의 상태에 따라 달라지는 다중 분기 조건 처리가 너무 많이 들어 있을 때. |
| Structure in Class Diagram | Context: 사용자가 관심 있는 Interface를 정의  State: Context의 각 상태 별로 필요한 행동을 캡슐화 하여 Interface로 정의  Concrete State: 상태에 따라 처리되어야 할 실제 행동을 구현. |
| Collaboration | 상태에 따라 다른 요청을 받으면 Context클래스는 현재의 Concrete State객체를 전달.  Context클래스는 실제 연산을 처리할 State객체에 자신을 매개 변수로 전달.  Context클래스는 사용자가 사용할 수 있는 기본 Interface를 제공.  Context, Concrete State는 자기 다음의 상태가 무엇이고, 어떤 환경에서 다음 상태로 가는지 결정할 수 있음. |
| Pros and Cons | 장점:   * OCP * 복잡한 if-statement 가 사라지고, readability가 좋아진다.   단점:   * State의 증가로 class 개수가 증가 할 수 있다. |
| Key Benefit | TBD |

### Template Method Pattern

|  |  |
| --- | --- |
| Description | Algorithm의 step을 정의하고, subclass로 하여금 하나 혹은 그 이상의 step을 재 정의 할 수 있도록 하는 pattern이다. Super class에서는 skeletor algorithm와 일부 step을 정의한다.  또한 Hook Method를 통해서 default 함수를 정의할 수 있다. |
| Situation | * 객체의 행동이 상태에 딸 달라질 수 있고, 객체의 상태에 따라서 런타임에 행동이 바뀌어야 함. * 어떤 연산에 그 객체의 상태에 따라 달라지는 다중 분기 조건 처리가 너무 많이 들어 있을 때. |
| Structure in Class Diagram | Abstract Class: 서브클래스들이 재정의를 통해 구현해야 하는 알고리즘 처리 단계 내의 기본 연산을 정의.  Concrete Class: 기본 연산을 구현. |
| Collaboration | Concrete Class는 Abstract Class를 통하여 알고리즘이 변하지 않는 처리 한계를 구현. |
| Pros and Cons | 장점:   * DIP * 중복 code를 없애고, 유지보수에 유리하다   단점:   * Template method변경 시 모든 자식 class가 영향을 받는다. * 하위 class에서 default step을 skip등을 하게 되면, Liskov Substitution Principle을 위반 할 수 있다 |

### Observer Pattern

|  |  |
| --- | --- |
| Description | 하나 혹은 그 이상의 객체가 system내의 다른 객체의 변화를 알 수 있도록 해 주는 design pattern이다. (One to Many Dependency). Subject와 observer가 loose coupling 되어 있으며, 어떤 객체의 상태 변화가 다른 객체의 동작을 trigger해야 할 때 사용한다. |
| Situation | * 한 객체에 가해진 변경으로 다른 객체를 변경해야 하고, 프로그래머들은 얼마나 많은 객체들이 변경되어야 하는지 몰라도 될 때. * 변화에 대해 통보할 때, 그 변화에 관심 있어 하는 객체들이 누구인지 몰라도 될 때. |
| Structure in Class Diagram | Subject: Observer들을 알고 있는 주체.  Observer: 변화에 관심 있는 객체를 갱신하는데 필요한 Interface를 정의  Concrete Subject: Concrete Observer에게 알려주어야 하는 상태를 저장. 변경을 통보.  Concrete Observer: Concrete Subject에 대한 Reference를 관리. Update함수를 구현. |
| Collaboration | Concrete Subject는 상태 변경 시 Observer에게 통보.  Concrete Observer는 필요한 정보를 Subject에서 질의하여 얻어 옴. |
| Pros and Cons | 장점:   * Subject와 observer가 decoupled되어 있다 * Subscribe / unsubscribe가 가능 * OCP   단점:   * Notification이 Random order로 이루어 진다 |

## 패턴 별 비교

### Mediator vs Façade

* 둘 다 객체 간 중재 역할을 수행하나 mediator는 객체 간의 workflow를 관장하는 것이 목적이고 façade는 sub system interface를 통합한 **하나의** 객체로 제공하여 편리성을 제공하는 목적임

### Strategy vs Template Method

* 알고리즘의 확장이 목적이나, Strategy는 알고리즘 전체가 교환/확장 대상이고, Template Method는 알고리즘의 일부가 교환/확장 대상임.

### Façade vs Adapter

|  |  |  |
| --- | --- | --- |
|  | Façade | Adapter |
| Are there existing classes? | Yes | No |
| Is there an interface we must design to? | No  (새롭게 제안하는 것임) | Yes |
| Is a simper interface needed? | Yes | No (반드시 그렇지는 않음.) |

### Adapter vs Bridge

* Adapter는 기존 인터페이스와 적용하려는 라이브러리의 인터페이스가 다를 때 이를 맞추기 위해 적용.
* Bridge 는 추상과 구현을 분리하기 위한 것.
* Adapter는 어떤 코드에 맞게끔 기존의 코드를 쓰기 위해 그 당시 만들어서 사용하는 경우 사용되고, Bridge는 확장성을 고려하여 미리 예상하여 bridge class를 구현 해 두어 코드를 진행해 나갈 때 사용되어짐.

### Composite vs Decorator vs Proxy

* Composite & Decorator 공통점: 여러 객체를 조직화 하기 위한 재귀적 합성 기법.
* Composite는 클래스를 구조화 하여 어떻게 하면 일관된 인터페이스로 다룰 수 있도록 일관성을 부여할 것인가가 초점.
* Decorator는 상속 없이 객체에 새로운 기능을 부여하기 위한 것에 초점.
* Proxy는 실제 구현체에 대해 indirection을 적용한다는 점에서 Decorator와 비슷하나, 서비스 제공자의 Reference를 직접 관리하는 불편함을 해결하는 것에 초점. 동적으로 어떤 기능을 추가하거나 제거하지 않음.

### Mediator vs Observer

* Mediator는 object간의 Communication을 encapsulate하고, observer는 communication을 observer과 subject에 distribute함.
* Mediator장단점: communication흐름을 이해하기 쉬우나, mediator 재사용 어려움.
* Observer장단점: reusable이 쉬우나, communication흐름을 이해하기 어려움.

### Factory Method (sub-classing) vs Abstract Factory/Builder/Prototype (object composition)

* Factory Method: 변경을 위해서는 Subclass 정의
* Abstract Factory: 한 Factory Object가 여러 개의 class들을 만듦.
* Builder: 여러 개의 Object를 합하여 하나의 Complex Object를 만듦.
* Prototype: factory object가 prototype object를 copy해서 생성. factory와 prototype이 한 object임.
* Factory method 패턴은 객체를 생성하는 method를 상속을 통해 구현하는 것이고 abstract Factory 패턴은 객체를 생성하는 factory class를 composition하여 위임을 통해 객체 생성을 하는 것

### Strategy vs State

* State 패턴은 state에 따른 behavior를 캡슐화 한 것이고 Strategy 패턴은 알고리즘을 캡슐화 한 것

### Builder vs Abstract Factory

* Builder 패턴은 생성 마지막 단계에서 반환
* Abstract Factory 패턴은 만드는 즉시 반환

### COR : Decorator

* COR은 linked list와 같이 수평구조로 책임을 덧붙이고 Decorator는 수직구조로 책임을 덧붙임.
* COR은 하나의 객체에서만 처리가 일어나는데 반해 Decorator는 덧붙이는 기능 모두에서 고유의 기능성이 적용
* Decorator는 기능을 덧붙여나갈 수 있는 구조적인 특성이 패턴의 핵심이라면 COR은 책임을 다음 handler에게 전달하는 행위가 핵심.

### Creation Pattern별 Composition 사용 방법 차이

|  |  |
| --- | --- |
| Pattern | Pattern Aspects that vary |
| Singleton | The sole instance of a class |
| Factory Method | Subclass of object that is instantiated |
| Abstract Factory | Families of product objects |
| Prototype | Class of object that is instantiated |

# CEP 준비

|  |  |
| --- | --- |
| ML Model | Logistic Regression   * 독립변수의 선형결합을 이용한 예측. 결과값이 0과 1 사이. * 단순선형 예측이 예측결과가 설정된 범위 값을 벗어나는 문제를 해결하기 위함.   Random Forest   * 수많은 의사결정 트리가 모여서 생성됨. * Voting을 통해서 최종 결정.   SVM (Support Vector Machine)   * 비선형 분류.   Ensemble   * Aggregation: 집계 * Bagging Approach (Bagging: Bootstrap Aggregation). Bias-variance trade-off를 극복하기 위함.   샘플을 여러 번 뽑아 각 모델을 학습시킴.   * Boosting: 맞추기 어려운 문제를 맞추는데 초점. * Stacking: 서로 다른 모델을 조합. |
| Collaborative Filtering Model | * 다른 사용자의 정보를 활용 * Matric의 데이터가 sparse한 경우가 많고, 이때는 정확도가 떨어짐. * Memory-based, Model-based (ML) 가능. |
| Content-based Recommender | * Item 기반으로 비슷한 Item을 추천. * 효율적이나, Diversity(다양성)이 떨어짐. |
| Knowledge-based Recommender | * Domain Knowledge 활용. * Domain Knowledge를 Machine Learning이 이해할 수 있는 형태로 표현하는 것이 어려움. |
| Context-based Recommender | * 다양한 형태의 Context를 추천에 활용. * Time, Location, Social Data, ETC. * Time-sensitive, Location-based, Social 등 |
| Cold-start Problem | Content-based와 Knowledge-based가 Collaborative보다 조금 더 유리함. |