**1. Object-Oriented Analysis and Design (OOAD)**

**1.1 OOP Concept**

#Encapsulation

code와 data를 하나의 번들로 같이 두는 것. 예를 들면, 함수도 될 수 있고, 클래스도 될 수 있음.

#object

객체지향의 가장 기본적인 것. 관련있는 attribute와 method를 한 바구니 즉, 객체에 넣는 것. (해당하는 바구니의 identity가 있어야 존재의 의미가 있음)

#Information Hiding

field나 method를 다른 객체에 공개하지 않고 숨기는 것. Association을 맺을 때 모두 공개할 필요가 없음. Interface 관점에서 다른 컴퍼넌트가 사용할 수 있는 method나 attribute를 제한할 수도 있고, 안전성 측면에서도 필요함.

#Visibility

Public : 제한없이 access 가능

Package : 같은 패키지 내에서만 access 가능

Protected : 본인과 상속받은 자식들까지만 access 가능

Private : 본인 자신만 access 가능

#Inheritance : 상속, 부모에 대해 성립하는 것은 자식에게도 모두 성립해야 한다. Is-a 관계일 때만 써야한다. ex) 펭귄은 조류다.

#Polymorphism

같은 메시지에 대해 클래스나 인자에 따라 다른 행위를 하는 것.

#Dynamic Binding

컴파일시 해당클래스가 정해지는 것이 아니라 초기바인딩 이후 런타임시 동적으로 할당 되는 것. 부모클래스로 선언하고 자식에 따라 method가 다르게 호출되는 것.

#OOP에서 왜 Dynamic binding이 필수적인지?? (객체지향의 존재의 목적을 재사용 -> 다형성)

상속과 다형성이 주된 개념이고, 런타임에 어떤 method가 호출될지 결정되기 때문에, if else와 같은 statement 분기가 필요 없이 객체에 따라 해당 메소드가 호출됨. 즉, client class의 소스코드 수정도 필요없고, recomplile도 필요 없다는 것.

주로 같은 method명으로 다른 impl을 보유하고 있을 때 수행이 됨.

단점으로는 program의 reliablie을 떨어뜨림 (컴파일에 의한 에러 detection이 되지 않기 때문에)

#Overriding

다형성의 하나. 부모클래스로부터 받은 메소드를 같은 이름, 같은 return, 같은 인자로 새로운 로직을 수행하는 것

#Overloading

하나의 클래스에서 같은 이름의 메소드들을 여러 개 가지는 것.

#Substitutability (치환성)

기반 클래스는 파생 클래스로 대체가능해야 한다.

#Abstract Class, Interface

공통점 : abstract class(추상 클래스)와 interface 는 선언만 있고 구현 내용이 없는 클래스이다.

차이점 : 추상클래스는 말 그대로 클래스임. Interface는 구현하기 전에 명세된 계약 같은 것. 추상클래스에는 abstract가 구현되어 있는 것도 추상화되어있는 method도 있음.

Interace는 구현체 자체가 없음. 명세만 있음

#Generic Class/Template

다룰 객체의 타입을 미리 명시해줌으로써 형변환을 하지 않아도 되게 하는 것.

다형성을 사용해야 하는 경우에는 부모의 타입을 지정해줌으로써 여러 객체를 저장할 수도 있음.   
와일카드카드 : 하나 이상의 타입을 지정할 수 있음(? 사용)

#Procedural programming vs OO programming

둘다 장단점이 있다고 생각함  
procedural programming의 대표적인 언어는 C가 있으며, DATA 위주가 아닌 행위에 포커스를 두고 step by step으로 수행하는 절차에 초점을 두고 프로그래밍 함. 루틴과 서브루틴들로 구성되어 있으며, 어떻게 동작할지 기대와 실제 동작이 매우 직관적임.

OOP의 대표적인 언어는 JAVA가 있겠죠. DATA와 객체 캡슐화에 초점을 두고 개발. 적은 코드로 더 복잡한 동작을 생성할 수 있음. 코드의 재사용이 가능. (모듈화가 되어있으므로)

가장 큰 차이는 어디에 포커스를 두는 개발방법인가인듯 합니다. 프로시져는 데이터를 처리하는 절차에 의존. OOP는 객체, 모듈화에 의존

#Procedural programming의 단점은 무엇인가?

선택의 여지가 없다면?? 당연히 procedural programming을 해야 함.

객체화가 되어있지 않기 때문에 코드가 massive하고 양이 많아지게 되면 유지보수가 어렵다.

모든 함수들이 global variables에 완벽하게 접근이 가능하기 때문에, 새로운 데이터가 추가되면 모든 함수들이 변경되어야 함.

컴퍼넌트 function이나 data structure가 real world의 모델처럼 디자인되지 않기 때문에 design하기가 어렵다.

**1.2 OOAD with UML**#Use Case Diagram

Actor + use case(행위)로 구분되며, 기능요구사항을 나타내기 위해서 작성. Actor는 기능을 요구하는 대상, 외부시스템, 타이머 등 다양하게 올 수 있음. 사용자 요구사항을 추출하고 분석함.  
1) relation

Assosication, Generaliztion, Include(A는 B를 포함함), extend(B는 A에 의해 가끔 호출됨)

2) Granularity of Use cases

You will encounter such decision points of the level of granularity that you need to define when you build any of the UML diagrams.

#Use Case Description / Scenario

비즈니스 트랜잭션을 하기 위한 행위를 sequencial 한 step으로 작성한 것. Title, actors, pre conditions, post conditions, path, use case ID, description, create by, date, priority, frequency of use 등에 정보를 갖고 있음

#Class Diagram

시스템을 구성하는 클래스와 구조를 표현함

객체들의 공통 구조와 동작들을 추상화 함

시스템을 구현할 때 어떤 클래스가 필요한지 그들사이의 관계는 어떻게 되는 지 표현.

클래스 이름. 속성. 오퍼레이션으로 구성

Abstract class는 이탤릭체, 스테레오타입, property 등을 줘서 표시함. (concrete class 가 아님)

진행 정도에 따라 다른 레벨의 class diagram이 나올 수 있음. 그 예로 처음에는 association 이나 inheritance만 있다가 점점 변경될 수 있음.

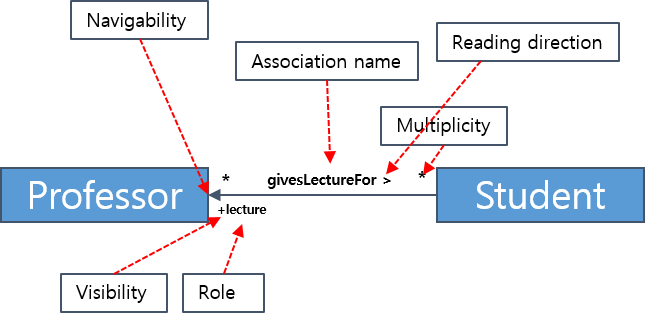
-relation

1) Dependency

A class needs to know about another class to use object of that class

2) Association

Models possible relationship between instances of classes



\*Binary association

\*Unary association

\*n-ary association : More than two partner objects are involved in the relationship

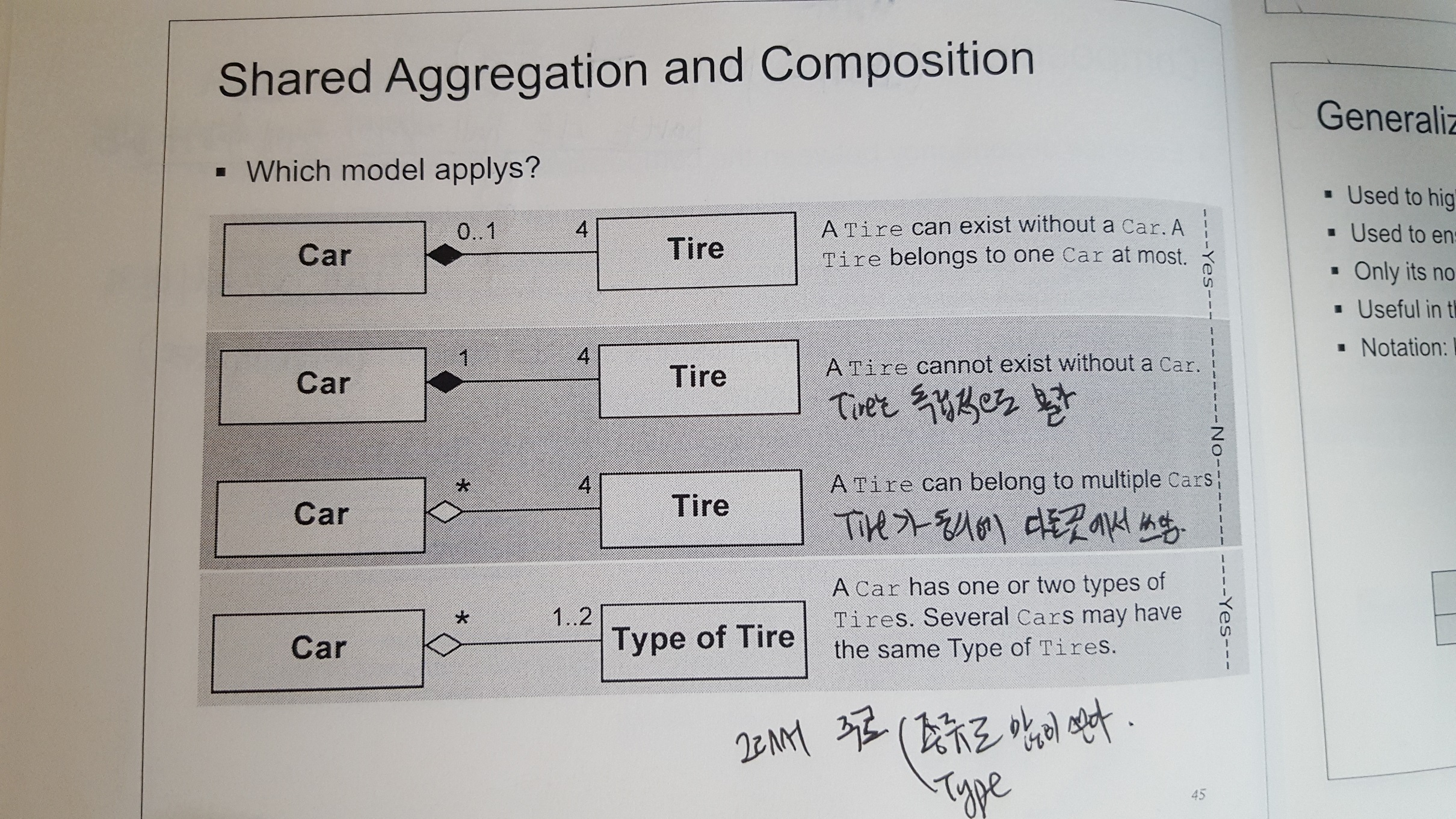
\*Association class : assign attribute to the relationship between classes rather than to a class itself (디폴트로 유니크하게 됨)

3) Aggregation

Whole과 part가 있음. 즉, 파트의 의미가 있음. 하지만 강한 의미는 아니

4) Composition

Whole과 part의 개념이 있음. 하지만, part는 다른 hall 객체에 동시에 속할 수 가 없음



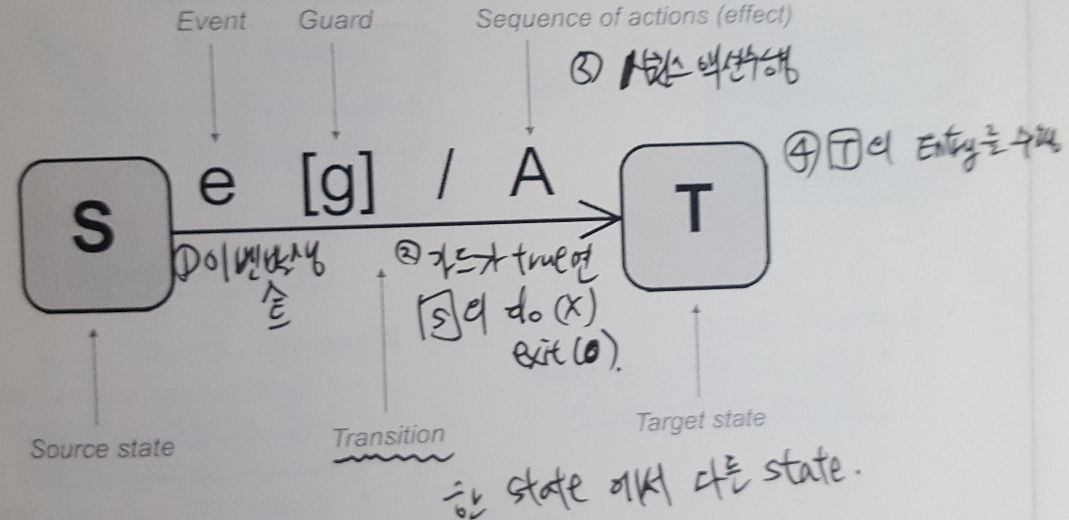
5) Inheritance

서브클래스가 부모 클래스의 특성, assosication, aggregation 등 모든 것을 물려받음

# Sequence Diagram

Behavior를 모델링 하기 위한 다이어그램. 인터렉션, 메시지, 루프 등 오브젝트간의 behavior에 대해 정의 함.

Concrete 한 시나리오를 모델링하기 때문에 1개의 시스템에서 여러 개가 나올 수 있음



sync message : ◀ / async message : ← / response : ---

# State Machine Diagram

Flow chart의 성격으로는 activity diagram이 좋고, 해당은 state 즉, 외부의 자극에 의해 변경되는 모습으로 표현되는 다이어그램

- Transition

Change from one status to another

-Event

Signal, Call, Time, Competition, Change이 될 수 있음. 예를 들면, 오븐에 대해 state machine diagram을 그린다하면, door\_open, door\_close, do\_baking, do\_toasting등등 모두 이벤트가 될 수 있음s

-Guard

조건

-Action

Guard가 성립했을 때 수행하는 action

- pseudo state

Initial state같은 pseudo statue 에서는 멈춰있을 수가 없음

-Decision Node

분기

-Orthogonal State

동시에 2개 이상의 state가 composite 될 수있음

-Fork and Join

병렬처리에 사용. 머물 수 없는 pseudo state

-H  
같은 레벨의 직전의 state 를 기억

- H’

가장 깊은 레벨의 직전의 state를 기억

- Encapsulation 가능

#Activity Diagram

Use case처럼 굵은 단위의 설계. Activity의 sequence를 보여주기 위함. 디테일한 프로세스를 보여줄 때 사용하며, 이에 대한 start point 부터 end point까지의 workflow를 보여줌

즉, use case가 what에 집중한다고 하면, 이것은 how에 집중하는 diagram.

-Action

액티비티의 더 이상 분해할 수 없는 single step을 이야기 한다. Ex) 오퍼레이션의 호출, 시그널의 전송, 객체의 생성이나 소멸 및 단순 계산 등

-Activity

작업의 실행을 의미하는 것으로 일련의 액션들로 구성된 작업 프로세스를 의미. High 레벨의 스트럭쳐로 action들의 순서로 구성됨

즉, activity diagram은 activity를 구성하는 action들을 보여준다.

#Component Diagram

1) 4 + 1 view

Logical view : logical한 상태에서 어떻게 구성되는지? => class diagram, state machine, object diagram 등

Process view : 시스템의 부분에 대해서 일의 절차를 기준으로 어떻게 해석 되는지? => activity diagram

Development view : 모듈과 컴퍼넌트로 적절하게 구성되어 있는지 ? => 개발자 관점에서

Physical view : 배포되는 단위. host간의 네트워크 등 ex)deployment diagram

Use case view : 시스템이 하는 기능

Ex) use case diagram, description

2) Provided Interface / Required Interface

컴퍼넌트는 루즈 커플링이 되어야 함. 그래서 컴퍼넌트는 provided and required interface가 필요.

Provided interface: 다른 컴퍼넌트에게 제공하는 인터페이스

Required interface: 다른 컴퍼넌트에게 요구하는 인터페이스

인터페이스의 방식

Ball and socket

Stereotype notation

Listing component

Dependency를 이용해서 그릴 수도 있다.

3) Pluggable Object

Required interface를 통해 component / system을 pass하는 object. 즉, interface가 있으면 해당 interface를 통해 imple 해서 required에 plugged in 되는 object

4) UML class와 UML component의 차이점

컴퍼넌트는 PORT와 internal structure를 갖고있으며, 오직 interface를 통해서만 access한다. Modular, self-contained, replaceable

5) Composite Structure

Class와 component에 동시에 반영할 수 있는 diagram

- Ports : 인터페이스를 런타임에 몇 개나 제공할지?

-Delegation connectors : interfaces가 어떻게 inter간에 어떻게 되는지?

- Assembly connectors : 다른 컴퍼넌트와의 인터페이스는 어떻게 되는지?

- Black and white-box views

#Deployment Diagram

시스템을 구성하는 HW자원 간의 연결관계 표현하고 HW자원에 대한 SW 컴퍼넌트 배치 상태를 표현.

-Node

직육면체로 표기. SW 컴퍼넌트가 탑재되어 처리되는데 관련된 HW 자원. 특히 computational resource.

-Artifacts

소프트웨어 개발 프로세스나 시스템 배치와 운영에 의해 사용되거나 만들어지는 산출물 (apk, jar등)

-Relationship to Quality-in-Use

런타임 실행환경이 system의 system-in-use에 영향을 줌. 예를 들면, computing ability 가 떨어지면 해당 하는 node를 추가하는 설계를 해야함

**1.3 Principle of OOAD**

# Principles for good design

1) Program to interfaces, not implementations

2) Favor object composition over inheritance

3) Encapsulate

4) 오브젝트 간의 loose couple design

5) SOLID

|  |  |  |
| --- | --- | --- |
| Single responsibility  (SRP) | 하나의 클래스는 하나의 책임만 가져아한다. | DAO 객체 |
| Open close  Principle  (OCP) | 확장에 열려 있고, 변경에 닫혀있어야 한다. | 상속과 어댑터클래스를 통한 클라이언트 접속 |
| The Liskov Substitutio  (LSP) | 자식 타입들은 부모 타입들이 사용되는 곳에 대체될 수 있어야 한다. | 상속을 해야할지 말아야 할지 잘 고려해서 해라 |
| Interface Segregation  (ISP) | 하나의 일반적인 인터페이스보다 여러 개의 인터페이스가 나음. 인터페이스의 크기가 너무 크면 자신이 실제 알아야 되는 부분보다 더 많은 부분을 알아야 함 | 파일입력은inputInterface, 파일 출력은 outputInterface |
| The Dependency Inversion  (DIP) | 클라이언트는 구체 클래스가 아닌 인터페이스나 추상 클래스에 의존해야 함 | 이벤트 드리븐, 콜백 |

소프트웨어 디자인을 이해하기 쉽고, 유연하며, 유지보수하기 쉽게 하기 위한 다섯 가지 원칙. Low coupling, high cohesion, strong encapsulation에 대해서 얘기는 하지만 어떻게 해야할지 막막할 때, 이 원리를 생각하면서 프로그래밍을 해야 한다.

# Modularity

1) Coupling

확장성을 위해 component간의 결합도를 느슨하게 하는 것. 1개의 컴퍼넌트를 바꿔야 할 때 수정해야 될게 5~6개의 컴퍼넌트면 유지보수 및 재사용에 어려움이 있음

2) Cohesion

하나의 클래스가 한 개의 responsibility를 갖고 있을 때. 하나의 함수가 있다는 얘기가 아니고, 그 함수를 변경해야 할 이유가 여러 개 가 있을때, design smell shotgun smell 처럼 responsibility가 여러 곳에 나눠져 있으면, high cohesion이라고 할 수 없음.

# Consistency between Diagrams

1) Use case diagram VS sequence diagram

둘다 behavior 다이어그램이며, use case는 요구사항에 있는 모든 기능적 요구사항을 actor와 use case별로 구분하여 작성하여야 하지만, sequence diagram은 해당 기능을 일부를 추출하여 work flow를 도식화 한 것임.

2) Class diagram VS sequence diagram

클래스 다이어그램은 구조적 다이어그램 이고, sequence diagram은 행위적 다이어그램의 한 종류 입니다. 여기서 sequence diagram의 depth가 어느 정도냐에 따라 다르긴 하겠지만, sequence diagram의 lifeline에 component나 class가 들어가게 될 것이고, 해당 lifeline간 msg가 교환되는 flow에 함수나 파라미터가 사용될 때 클래스다이어그램에 있는 것들이 들어갈 것으로 보임. 시퀀스다이어그램에 나타난 method가 실제 class diagram의 association, dependency등에 잘 녹아져있는지 확인

3) Class diagram VS State Machine diagram

클래스 다이어그램의 클래스가 state machine으로 translation이 잘되었는지 본다. 클래스 뷰는 어떤 메소드와 어떤 속성이 있는지 보여주고, state machine은 그것의 method execution에 대한 order를 보여줌.

- 액션의 argument의 개수가 class의 parameter에 매치 해야함

- operation의 이름이 class의 object와 일치

#UML Constructs for Modeling Parallel Processing

Class diagram: parallel service manager class 밑에 상속으로 여러 개의 class를 표현.

Activity diagram: fork and join으로 병렬처리 표시

Sequence diagram: par fragments로 표현 가능

State machine : orthogonal state로 동시에 수행되는 것을 표현 가능

#UML이 많은 이유??

해당 시스템을 여러 개의 view의 다양한 모습으로 시각화 할 수가 있음 (각자의 역할이 다름)

**2. Prerequisite: Design Patterns**

**2.1 Principles of Design Patterns**

1) 3 Principles of Design Patterns

Interface Separated from Implementation

=> 바로 구현하는 것보다 인터페이스를 이용

Substitution with various implementations

=> 구현 상속 보다는 인터페이스 상속을 사용한 객체의 조합을 선호해라

Open Closed Principle (OCP)

=> white box reuse vs black box reuse

#OCP의 장점?

OCP가 잘 적용이 되면, 기능을 추가하거나 변경해야 할 때 이미 동작하고 있는 원래 코드를 변경하지 않아도, 새로운 코드를 추가함으로써 기능의 추가나 변경이 가능하다. (주로 추상화를 통한 OCP 수행)

#OCP의 mechanism?

Direct access 되어 있는 객체가 있다고 쳐봅시다. Robot 클래스가 있고, work 클래스가 있는데, work 클래스는 움직이고, 자르고, 붙이고를 해야 한다고 햇을 때, 이것이 direct access가 되어있으면?? Work가 하는 일이 변경되거나 추가되면 원래 코드를 변경해야 함. Work1 에서 work2 등등 만들어서 robot 해당하는 것을 하도록 조정해야함. 이러한 direct access를 추상화를 통해서 indirect 시켜주는 것. Work 라는 추상화 클래스를 만들고, robot은 이 추상화클래스의 객체가 뭔지는 모르겠는데, 움직이고 붙이고, 잘라줘 라고 하면?? 코드의 변경 없이 확장이 가능하게 됨. 이러한 추상화와 다형성의 메커니즘을 통해서 OCP가 가능하게 함

2) Common OOP Mechanisms / Constructs used in Design Patterns

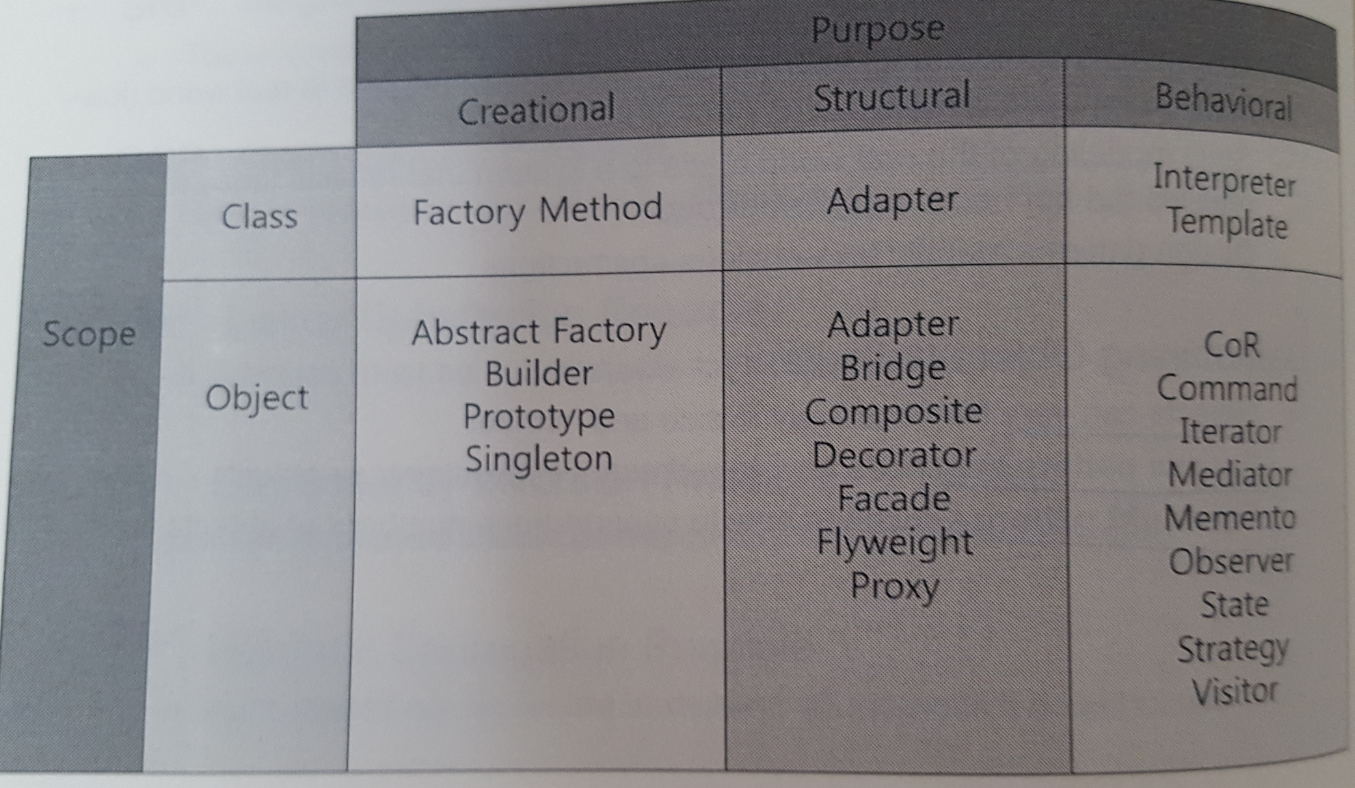
 Abstraction with Interface

 Inheritance / Specialization

 Dynamic Binding

 Delegation

# design pattern



|  |  |
| --- | --- |
| Singleton | 해당 클래스의 인스턴스가 하나만 만들어지고, 어디서든지 그 인스턴스에 접근할 수 있도록(전역 접근) 하기 위한 패턴 |
|  |
| Strategy | 알고리즘군을 정의하고 각각을 캡슐화하여 교환해서 사용할 수 있도록 만든다. 알고리즘을 사용하는 클라이언트와는 독립적으로 알고리즘을 변경할 수 있다.  일반적으로 서브클래스를 만드는 방법을 대신하여 유연성을 극대화하기 위한 용도로 쓰인다.  Ex) calculate 함수에 대해 각 서브클래스(add, sub)이 각자 알고리즘을 구현. Minus class 가 생겨도 Strategy 부분은 변경되지 않는다. |
|  |
| Observer | 한 객체의 상태가 바뀌면 그 객체에 의존하는 다른 객체들한테 연락이 가고, 자동으로 내용이 갱신되는 방식으로 일대다(one-to-many) 의존성을 정의한다.  Ex) currentConditions, staticdisplay, forecastdisplay 가 subject의 registerobserver를 access해서 본인을 등록. 등록된 것은 리스트로 관리. 온도가 바뀔 때 마다 notifyObservers 호출. 거기서 observer들의 update함수를 호출 |
|  |
| Decorator | 기존 상속구조에서 부모 코드를 변경하지 않고, 런타임에 동적으로 기능을 추가하기 위한 pattern  Ex) Beverage beverage2 = new HouseBlend(); beverage2 = new Soy(beverage2); beverage2 = new Mocha(beverage2); 하우스블렌드 커피 객체에 소이와 모카를 추가 |
|  |
| Factory | 객체를 생성하기 위한 인터페이스를 정의하는데, 어떤 클래스의 인스턴스를 만들지는 서브클래스에서 결정하게 만든다. 클래스의 책임 몰아주기가 가능 (제품을 생산하는 부분과 사용하는 부분을 분리)  Ex) 각모양을 만드는 팩토리를 구분. 새로운 모양을 만들어야 되면 팩토리 추가. 동그라미 그리는 부분에 문제가 있으면 rectanglefactory만 수정하면 됨. |
|  |
| Command | 요구 사항을 객체로 캡슐화 할 수 잇으며, 매개변수를 써서 여러 가지 다른 요구 사항을 집어넣을 수 있다. 또한 요청 내역을 큐에 저장하거나 로그로 기록할 수도 잇으며, 작업취소 기능도 지원 가능하다.  Ex) remoteControl(인보커)이 command 객체를 갖고, cmd에서는 추상화 메소드 구성 및 실제 수행은 concrete class 인 lightonCommand에서 에서 수행. RemoteLoader는 client 역할 수행 |
|  |
| Adapter | 한 클래스의 인터페이스를 클라이언트에서 사용하고자 하는 다른 인터페이스로 변환한다.  주로 구현을 이미 해놨는데, 사후에 인터페이스를 해야될 때 함  Ex) adapter가 target을 realization 한다. 그러면 adapter에서 target에서 있는 모든 함수를 구현해야하고, 그 adapter에서 adaptee를 호출하여 함수를 호출시킨다. |
|  |
| Facade | 어떤 서브시스템의 일련의 인터페이스에 대한 통합된 인터페이스를 제공한다. Ex) ShapeManger에서 package1,2,3에 대한 로직을 모두 수행함. 만약 해당 내용이 변경되면 façade만 바꾸면됨. => loose coupling  만약 없었으면 client 1, 2, 3다 바꿔야함 |
|  |
| Template Method | 메서드에서 알고리즘의 골격을 정의한다. 알고리즘의 여러 단계 중 일부는 서브클래스에서 구현할 수 있다.  Ex) 공용으로 사용할 preparerecipte, boilwaster, pourincup은 상위클래스에서 구현 나머지는 abstract 로 서브클래스에서 구현 |
|  |
| Iterator | 컬렉션 구현 방법을 노출시키지 않으면서도 그 잡합체 안에 들어있는 모든 항목에 접근할 수 있게 해주는 방법을 제공한다. 즉, 구현과 interator를 나눠준다.  Ex) Aggregate는 집합체. Iterator 루프관련 인터페이스. BookShelfIterator는 검색을 실행하는 클래스. Book은 책. BookShelf는 책들을 저장하고 있는 배열, 리스트 등 collector |
|  |
| Composite | 객체들을 트리 구조로 구성하여 부분과 전체를 나타내는 계층구조로 만들 수 있다.  Ex) 트리 구조의 패턴, 디렉토리 구조에 주로 사용하여 부모클래스인 component가 composite 클래스의 자식으로 들어가는 recursive 형태 |
|  |
| State | 객체의 내부 상태가 바뀜에 따라서 객체의 행동을 바꿀 수 있다. 마치 객체의 클래스가 바뀌는 것과 같은 결과를 얻을 수 있음. 하나의 객체에 여러가지의 상태가 존재할때 보통 IF문이나 SWITCH 문으로 분기를 해서 결과를 처리한다. 그러나 신규 상태가 존재할때마다 기존의 IF문이나 SWITCH이 다시 수정되어야 한다.  Ex) Light의 on, off의 state에따라 on\_button\_pushed(), off\_button\_pushed()의 처리하는 로직이 달라짐 |
|  |
| Proxy | 어떤 객체에 대한 접근을 제어하기 위한 용도로 대리인이나 대변인에 해당하는 객체를 제공하는 패턴  Ex) main에서 proxy 객체 생성. 해당 객체의 doAction을 수행. proxy에서 realsubject에 access |
|  |
| Bridge | 구현(implementation)으로부터 추상(abstraction) 레이어를 분리하여 이 둘이 서로 독립적으로 변화할 수 있도록 한다  Ex) cTribeHuman = new cHuman(); cTribeHuman.attack();  cTribeElf = new cElf();cTribeElf.attack(); 처럼 사용하여, 실제 구현은 wWeapon의 cSword cBow에서 각자 작성 |
|  |
| Chain of responsibility | Chain of Responsiblity 패턴에서는 문제 해결사들이 한줄로 쫙 서있다가 문제가 들어오면, 자기가 해결할 수 있으면 해결하고, 안 되면 다음 해결사에게 문제를 넘겨버림.  AbstractOrgDimensionValidator가 자신을 aggregation(집합), setNext()함수를 구현하여 다음 객체를 지정. 한 객체씩 돌아가면서 실행하도록 구현, |
|  |
| Visitor | 구조에서 operation을 띄어놓는 패턴. 즉 오퍼레이션을 클래스처럼 집어넣는 패턴. 방문하고 싶은 NODE들을 하나씩 방문하면서 해당에 맞는 operation 내용을 수행. 사용하는 이유는 기존의 class를 새로운 responsibility로 더럽히고 싶지 않기 때문에.  double dispatch (V.E).visit()을 해야하는데, 이게 불가능하므로 두번에 걸쳐서 함. 근데 if else로 하는게 아니라 framework 형식으로 제공 |
| visitor |

**2.5 Issues**

1) Similarity and difference between Two Different Patterns

|  |  |
| --- | --- |
| Bridge vs Adapter | 객체를 입력받아, 입력받은 객체와 매우 관련 높은 새로운 클래스 혹은 인터페이스로 참조 |
| Adapter는 어떠한 클랙스가 기존에 구현되어있는 인퍼테이스와 기능을 동일한데, 함수명이나 paraemter나 등등 뭔가 중간에서 adapt의 역할을 수행해야 하는 component가 필요할 때  Bridge는 기존의 존재하는 객체의 참조와는 관련이 없고, 객체의 범용성을 증가시키기 위해 구현부를 비워놓고, 객체의 생성시에 구현부를 입력 받는 방식  Ex) unit class, weapon class |
| Façade vs mediator | 중간의 매개체에서 통합된 인터페이스를 제공한다 |
| Façade는 별도의 비니지스 로직을 보유하고 있지 않고, 그거 서브클래스들의 high level의 interface를 제공하는 것. 복잡한 서브 시스템 대한 단순한 인터페이스의 제공이 필요할 때 유용. 일반적으로 단 방향  Mediator는 어떤 프로그램이 여러 개로 나눠져있을 때 로직과 계산을 해당 클래스로부터 나누는 것. 자신의 갖고있는 인터페이스를 강제적이지 않은 방식으로 적용한다. 일반적으로 양 방향 ex) 비행기 관제탑 |
| Composite vs  Decorator | 여러 객체를 조직화 하기 위해 재귀적인 합성기법을 사용 |
| 목적이 다름. Composite 패턴은 클래스의 구조화에 초점이 맞춰진 것. 어떻게 하면 관련된 객체들을 하나의 인터페이스로 다룰 수 있도록 일관성을 부여할 것인가  Decorator는 상속 없이 객체에 새로운 서비스를 추가하려는 목적을 갖음. 즉, 서비스 클래스의 조합이 필요할 때 마다 클래스를 생성해야 하는 문제를 해결 |
| Decorator vs  Proxy | 클라이언트에서 동일한 인터페이스로 객체들을 합성할 수 있다. |
| proxy에서는 decorator 처럼 어떤 기능성을 추가했다 지웠다 하는 것이 아니라, 서비스를 제공하는 대상에 대한 참조자를 직접관리하는 불편함을 해결하려는 것임 |
| Template method vs strategy | 둘다 추상화를 위한 패턴이라는 것은 동일 |
| 템플릿 메소드 패턴은 concrete algorithm이 서브클래싱에 의해서 컴파일할 때 결정이 됨. 정적 바인딩이라고 할 수 있음. 반면에 strategy 패턴은 클라이언트에 의해 런타임 시 concrete class가 결정이 됨 |
| State vs strategy | 둘다 현재 상태나 인스턴스에 따라서 dynamic bounding 된다는 것 |
| Strategy 패턴은 외부에서 해당 dynamic bounding 되는 객체가 결정이 됨. 즉 client 객체에서 context 객체가 실행할 오브젝트를 변경 함  State 패턴은 내부에서 해당 dynamic bounding 되는 객체에 대해 결정 함. 즉 상태 자체를 객체 내부에서 결정함. Context 객체의 상태를 상태객체 내에서 변경하여 외부에서 상태를 변경하지 않음 |
| Strategy vs command pattern | 메소드를 Dynamic binding 시키는 부분 |
| Strategy는 객체의 다양한 전략에 대해 설정하는 반면,  command pattern은 command라는 단일행위 자체에 집중하고 그 단일 행위에 대한 사전 / 사후 동적들을 정의. Command들을 캡슐화하고, 매개변수처럼 사용 가능 즉 실행된 기능들을 로깅하거나 undo기능 같은 것 할 때 사용하면 좋다.  Controller를 만들고, command를 interface를 만들고, lightCmd, musicPlayCmd 를 만들어서 그안에 lightCmd, musicPlayCmd안에 onExcute, onUndo 등을 구현한다. |
| Strategy vs visitor | 같은 interface로 다른 implementation을 하는 것은 동일 (추상화 시키는 패턴임에는 동일) |
| Strategy pattern은 1 : Many 의 방식. 한가지에 대해서 여러 개의 multiple operation을 수행하고 싶으면 strategy pattern 수행  Visitor pattern은 Many : Many 의 방식으로 수행. 1개의 obj가 visit하는 obj에 따라 imple이 결정됨 |
| Bridge vs strategy | 객체의 행동방식은 입력 받은 객체에 의해 정해짐 |
| Strategy는 객체의 생성시점이 아닌 동작시점의 행동방식의 변경에 초점이 맞춰져 있음.  Bridge는 생성시점에 초점이 맞춰져 있음. 전략 패턴처럼 런타임에 행동이 변경되는 것이 아니라 객체의 생성시점에 입력으로 받는 행동객체에 따라 브리지 객체의 행동방식이 결정된다. |
| Factory method  Vs abstract factory | Factory에 의해 객체의 생성이 결정된다. |
| Factory method는 template method와 비슷하게 사용하면서 어떤 클래스의 인스턴스를 만들지는 서브클래스나 클라이언트에서 결정하게 함  유닛 H;  H = 배럭.생성(MR); //마린생성  H.어택(); //총쏘기  H = 배럭.생성(MD); //매딕생성  H.어택(); //치료하기  Factory pattern은 인터페이스를 이용하여 나머지 코드들은 동일하게 하고 factory만 다르게하면 수행  Ex) 생산공장 F;  유닛 H;  F = new 배럭();  H = F.생성(); //마린과 매딕 생성  H.어택(); //총쏘기와 치료  F = new 스타포트();  H = F.생성(); //배틀크루저와 레이스의 생성  H.어택(); //레이저와 미사일 |

3) Why some of the design patterns don’t fully follow the SW design principles?

- 디자인 패턴은 "'이러한 상황' 이라면 '이러한 형태의 클래스 디자인'을 하는게 대게 좋더라" 라고 말해주는 것임

#주의해야 하는 이유?

- 표준화된 작업은 여러가지 상황을 고려하며 만들어지기 때문에 대체로 효율성이 떨어지기 마련이다. 또한 패턴간의 관계를 생각하지 못하면 설계를 복잡하게 만드는 요소가 되어버린다.

(사용 목적, 장단점, 강점과 약점에 대한 충분한 이해와 패턴간의 융합에 발생할 수 있는 문제점들을 확실히 예상할 수 있어야 하며 이를 바탕으로 디자인 패턴을 적용해야함)

#주의해야 할점

- 패턴에 대한 설명은 이해도를 높이기 위해 분리된 것처럼 진행되지만, 패턴의 사용에 있어서는 여러 가지 패턴들이 조화롭게 섞여있는 구조를 가지게 된다는 점 => 유기적으로 결합되지 못한다면 디자인패턴은 필요악이 되어버림

- 구동 환경이 다름. PC 어플리케이션이라면 디자인패턴이 가져오는 퍼포먼스에 대한 결점은 큰 문제가 되지 않을 것이다. 하지만 안드로이드 같은 모바일 환경이나 임베디드 환경이라면 디자인패턴은 사치품에 지나지 않는음

- 구동 목적이 고려되어야 함. 구동에 큰 설계가 필요하지 않거나 명확한 클래스구조를 갖고 코드의 변경을 고려하지 않아도 된다면 디자인패턴은 쓸데없는 비용을 늘리는 요소가 됨

**2.6 the most impressive pattern**

Visitor pattern

알고리즘을 객체 구조에서 분리시키는 디자인 패턴. 구조를 수정하지 않고도 실질적인 새로운 동작을 기존의 객체 구조에 추가가 가능! (OCP의 끝판왕). Spring frame웍처럼 객체구조를 무너뜨리지 않고, 새로운 responsibility를 추가 할 수 있음

다른 pattern들과 다른 점은 double dispatch가 된다는 점. 예를 들어, element가 집, 아파트, 사무실 visitor가 팔기, 사기, 가격내기등이 있다고 하면, 각자를 객제와 기능을 나눠서 구현할 수 있음.

장점) 객체나 데이터의 구조는 자주 변하지 않으면서 많은 기능들의 추가를 요구할 때 사용

Ex) 만약 팔기에 대한 기능을 수정한다고 하면? 팔기 class만 바꾸면 됨. 원래는 A의 팔기 B의 팔기 C의 팔기를 다 바꿔야 하는데

처리 나 알고리즘을 데이터구조에서 분리. 데이터의 독립성을 높여줌

단점) 객체가 늘어나거나 줄어들면 모든 visitor를 수정해야 함.(cost가 많이 듬)

**3. Requirement Engineering**

1) Quality Model, such as ISO 9126

# Definition of SW Quality

2가지 측면에서 볼 수 있음

- 기능상의 품질 : 기능요건이나 사양에 기반하여 주어진 설계를 얼마나 잘 충족하고 있는지를 반영

- 비 기능상의 품질 : 올바르게 소프트웨어 올바르게 개발될 수 있는 가늠하는 척도. 비즈니스 가치를 제공하기 위한 갖춰야 할 묵시적인 항목들.

품질을 갖추면?? 내부 구조의 효율화, 대외 이미지, 신뢰도, 마케팅의 효과 등을 얻을 수 있음

# Types of Quality Aspects

- Process Quality : Quality of life-cycle process 개발 프로세스

- Internal Quality : Quality of intermediate Artifacts 설계 산출물 (소프트웨어 실행에 영향에 상관이 없음)

Ex) source 코드 줄수, artifacts 사이즈 등, 가독성, 구조화

- External Quality : Quality of implementation Code 구현 코드

Ex) 소프트웨어의 러닝에 따른 response time, UI는 쉬워? 소프트웨어는 빠르나?

- Quality in Use : 유저가 바라보는 software를 포함환경 환경에 대한 quality

소프트웨어를 운영하는 환경에서 평가 됨

실질적으로 소프트웨어를 평가하는

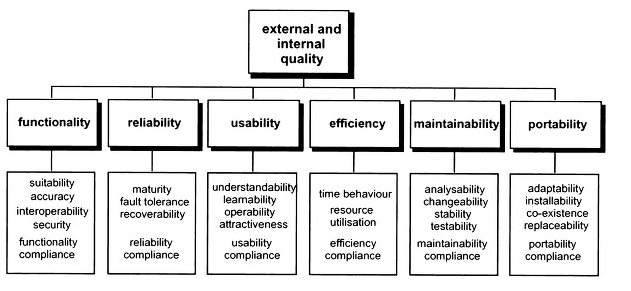
Ex) 환경에서 평가가 된다는 것. End-point 에서 유저가 실제로 느끼는 품질.   
사용편이성, 기호성 등과 같은 사용성(Usability)에 한정된 것이 아닌 전반적인 품질 특성에 의해 영향

을 얘기하고, Process quality => Internal Quality => External Quality => Quality in Use 는 모두 depends on 상위 개념.

즉 Process quality가 잘못되면 다른 품질들도 영향을 미치기 때문에 중요하다.

#Quality Model of ISO 9126

 6 Main Factors and Sub‐Factors for each



2) Requirement Engineering Process / Steps

|  |  |
| --- | --- |
| 타당성검토 | 기술적 검토. 비용. 프로젝트 가능여부 등등 진행해도 될지에 대한 타당성을 검토 |
| 요구사항 추출 | Stake holder간의 대화를 통해 Abstract한 요구사항을 concrete한 요구사항으로 변경 |
| 요구사항 명세 | 모든 참여자가 이해할 수 있는 정형화된 문서로 생성. 커뮤니케이션의 baseline이 됨.  정확성, 완전성.,, 모호성제거, 일관성, 검증성, 가변성, 추적성 |
| 요구사항 협상 | Stake holder들의 mutually satisfactory 가 중요함  Winwin negotiation 방법 사용  1. stake holder들 별로 요구사항을 쭉 나열  2. conflict가 나는 요구사항을 묶음  3. stake holder들 끼리 협상수행  4. 요구사항에 대한 option들을 추출하고 agreement를 정함 |
| 요구사항 분석 | Stake holder들이 요구사항을 명확히 이해하도록 요구사항을 자연어나 다이어그램으로 표현하는 것 |
| 요구사항 검증 | 검증기준 : atomic (각 요구사항은 하나의 traceable요소), unique , feasible, legal, clear, precise, verifiable, abstract (특정설계를 내포하지 않음), |
| 요구사항 변경관리 | 변경식별 및 변경제어, 추적 및 영향분석, 형상관리등의 제반활동  #요구사항 변경을 억제하지 않음 (변경의 필요성을 이해하고 관리 / 변경수용의 딜레마) / 변경 가능성이 있는 요구사항을 기록(Global Req, Volatile Req)  #요구사항 추적성 : high level artifact derive refined artifact, Traceability link should be generated between two artifacts. 해당 req를 고쳤을 때 연관된 아키텍쳐나 코드 테스트케이스 등 연결되어있는 부분을 추적할 수 있어야 함 |

3) Agile Process for Requirement Management

애자일에 도움이 되는 요구공학은????

최근 소프트웨어는 민첩한 개발에 관심을 보이고 있음. 높은 품질의 소프트웨어를 신속하게 제공하기 위해 기존의 전통적인 SW개발방법론과는 다른 민첩한 방법을 사용하는데, 이럴 때 사용하는 게 애자일 방법론임. 요구공학에서 요구사항을 추출하고, 애매하거나 정확하지 않은 요구사항을 정제하고, 이를 명세화하는 작업을 하는데. 원래 애자일은 2주나 4주 등 기간을 기준으로 stake holder들이 모여서, 요구사항을 명확히 하고 해당 스프린트를 진행해 나감. 요구항의 변화에 최적으로 대응 할 수 있음.

- 가짜모델이 아닌 진짜 software를 보고싶다. Plan을 완벽하게 세울생각을 하지말고, 변화를 수용하는 방법으로 개발을 수행하자

- 특징 : Time boxing / Estimation 0 or 1 / Coordination & Collaboration

- 이 컴퍼넌트 개발하는데 얼마나 걸려 ? => 2주동안 개발할 수 있는 게 뭐야?

4) Requirements for Application Software vs Platform Software

플랫폼화된 서비스 ex) 아마존이나 구글의 머신러닝. 이클립스.

즉, n개의 어플리케이션이 모두 호환이 되도록 설계가 되어야 함. 이 점이 그냥 보통의 application과는 다름. 이를 위해 framework, platform, middleware, OS 등에 대해서 고려를 해봐야함.

중요한 요소로는 Applicability of Platform, OCP가 있음. 이런 것들이 반드시 설계단계에서 선행되어야 함. 이게 반영이 안되면 아키텍쳐가 나올 수 가 없음

**4. Subject : Architecture Design**

**4.1 Terms in Software Architecture (용어)**

1) software architecture

타겟 시스템의 기본적인 컨셉이나 속성을 정의하는 것. 시스템의 elements, relationship, design의 principles이나 evolution.

Principle design의 decisions의 집합

2) architecture의 구성

Component + relationship + @ (비 기능적 요구사항)

3) 좋은 AD란 무엇인가?

효과적이게 stakeholder들과 communication 할 수 있는 수단이 되는 document.

방법론을 사용하는 것 + BOK 를 이용하는 것

4) 아키텍쳐 디자인의 4요소

Process : 시멘틱한 workflow 소프트웨어 디자인을 위한

Styles : 디자인 패턴 등

ViewPoints : 각 뷰에 따라 아키텍쳐를 기술

NFR-driven Design : 아키텍쳐는 NFR을 만족하는 tactics와 decision들이 있어야 함

**4.2 Architecture Styles**

|  |  |  |
| --- | --- | --- |
| hieratical | Main-subroutine | 서브루틴을 재사용해야 할 경우  독립적인 서브루틴을 갖고 있을 경우  소프트웨어가 서브루틴으로 계층적으로 빠져야 할 때 |
| 테스크 단위로 decompose 하귀 쉽다 (top-down) |
| 글로벌하게 사용되는 데이터가 있을 경우 취약하다 |
| Master-slave | Support fault tolerance and system reliability  Slave에서 replicated(복제된) 서비스를 제공  병렬 처리와 정확한 컴퓨팅에 사용  ex)빅데이터 |
| 완벽한 신뢰성을 보장 가능  (replication of server) |
| Master-slave의 네트워크속도에 의존적이다 |
| Layered | 시스템이 몇 개의 계층적 레이어(각자 가지가 맡은 역할이 있음)로 나누어져 있어야 할 경우  ex) OSI 7 layer, virtual machine, repository 등 |
| 소프트웨어 개발이 빨라진다. (추상화 가능)  Modifiability, 한 layer는 위,아래 layer에게만 영향을 준다. => reuse 용이 |
| 모든 시스템에 suitable 하게 적용되지 않는다  추상화 하는게 쉽지는 않음  다른 시스템과의 coupling 정도에 따라 성능이 달라질 수 있음 |
| Virtual-machine | Execution 환경에서 실제 application을 분리해야 할 경우사용  Ex) 에뮬레이션, JVM 등 |
| Portability and machine platform independency  Simplicity of software development  Simulator for disaster working model |
| Slow execution of the interpreter due to the interpreter nature  Additional overhead due to the new layer  Related architecture |
| MicroKernel | Adapt to changing system requirements  Plug-in 아키텍쳐의 특징  Ex) 이클립스, 인터넷브라우져 |
| Extensibility (확장성), Flexibility,  독립적임 플로그인 모듈과 코어시스템 |
| 어플리케이션이 플랫폼이 계속 대처해야된다. 새로운 하드웨어와 소프트웨어 발전에  느리고 시스템이 복잡해진다. |
| Async Communication | Non-uffered event-based Implicit Invocation | 메시지 없이 비동기식 호출이 필요할 경우  어플리케이션이 호출이 필요한 컴퍼넌트와 loose coupling을 해야할 경우  Ex) interactive GUI component communication,  Async notification |
| 많은 벤더의 API이용가능 (swing, AWT등)  이벤트 핸들러의 재사용이 가능  병렬 이벤트 처리가 가능 |
| 테스트와 디버그 하기가 어려움 (예측하기가 어려워서)  이벤트 소스와 리스너와의 tight 커플링(queue invocation와 비교시) |
| Buffered Message-based | Async로 주고 받아야되는데 message를 주고 받아야 할 경우  신뢰성이 있는 메시지-oriented-middleware가 필요한 분산처리 시스템일 경우  Ex) rabbitmQ |
| 메시지 프로듀서와 컨슈머간의 원자성 보장  컨슈머와 프로듀서가 동시실행  신뢰성있고 확장성있는 메시지 딜리버리 가능  프로듀서와 컨슈머와의 loose coupling |
| 메시지 큐의 사이즈 제한  시스템이 복잡해짐 |
| Distributed | Client-server | 2개의 다른 프로세스가 커뮤니케이션 네트워크를 통해서 분산처리 해야할 경우 |
| 역할에 따라 분리가 가능 (프리젠테이션, 비즈니스 로직)  서버컴퍼넌트의 재사용이 가능 |
| 시큐리티 복잡성  서버 가용성과 확장성이 필요함 |
| Muti-tiers | 엔터프라이즈 어클리케이션에서 주로 사용 |
| 확장성, 재사용성이 가능 (미들 티어를 추가함으로써) |
| 테스트가 어려움  서버 가용성과 확장성이 필요함 |
| Proxy | 프록시를 통해서 indirectly하게 강제적으로 메소드를 delegate함  일반적으로 공통 인터페이스로 사용  일루젼이나, Preprocessing 이 필요할 경우 |
| 메모리 절약  퍼포먼스 향상(캐쉬 프락시일 경우)  하우스 키핑과 기능성을 분리 |
| 복잡한 기능일시 잠재적인 overkill 존재 |
| Dispatcher | 분산처리에서 확장성과 가용성이 필요할 경우 |
| 가용성, 확장성 확보  효율적인 분산처리가 가능 |
| 비용  Dispatcher가 죽이면 연결되어 있는 서버가 무용지물이됨 |
| Broker | 클라이언트와 서버 사이에 코디네이터 해주는 미들웨어 시스템이 필요할 경우  브로커에 등록하면 찾고 중간에서 연결하나 load balance의 개념은 없음  Ex) corba, |
| 안쓰는 application에 대해서 꺼놓고 있다가 후 키는등 다양한 기능이 있음  Location transparency (브로커가 알아서 할당해주니까 서버 위치를 알 필요가없음)  Changeability and 확장성, 재사용성 |
| 프락시로 인한 오버헤드  Low fault-tolerance  테스트 하기 어려움 |
| SOA | 모든 것을 svc 기반으로 하여 독립적으로 사용하고 싶을 때  Ex) web svc, openAPI svc, rest ful 기반의 웹 서비스  순서 : 등록, SVC를 찾음, 찾았으면 바로 표준화된 언어를 이용하여 파싱 |
| 루즈 커플링 커넥션  Interoperability : 그들의 플랫폼을 고려할 필요없이 서비스를 접근할 수 있음  Reusability : 서비스화 하면 재 사용 쉽게 가능  Scalability : 서비스화 하면 확장도 쉬움 |
| 인터페이스와 프로토콜의 표준화가 없으면 접근 불가 |
| Data flow | Batch Sequential | 한덩어리의 배치 데이터를 끝까지 처리해서 답을 얻어야 하는 경우 |
| 서브시스템의 간단한 디비전  각각의 서브시스템은 stand-alone 프로그램이 가능함 |
| 인터렉티브 인터페이스를 제공하지 않음  Concurrency가 제공되지 않는다.  단계를 넘어갈 때 전체가  High latency (잠재) |
| Pipes & Filter | Stream data를 처리해야할 경우  각각의 프로세싱 스탭을 캡슐화 해야할 경우  Ex) 음성인식, 센서데이터 처리, 뉴럴네트워크 |
| Concurrency : high overall throughput for excessive data processing  Reusability : 필터들의 캡슐화가 가능하며 이를 재사용할 수 있음  Modifiability : 필터들 사이에 루즈 커플링이 가능  Flexibility : 스퀀셜하고 패러럴하게 처리 가능 |
| Limitations : 다이나믹 인터렉션을 할 수 없음  시스템 설계가 어려움  여러 필터로 데이터를 보내야 할 경우 오버헤드가 발생할 수 있음 |
| Process Control | 임베디드 시스템 소프트웨어를 디자인해야 할 경우  전체 시스템을 여러 개로 나누고 이를 연결하는 아키텍쳐가 필요 할 때  루프 + close loop가 존재  Ex) 무인자동차, 무인시스템, 건물 온도조절 시스템 |
| ? |
| ? |
| Data-centered | Repository | 데이터 집중화가 필요한 아키텍쳐 일 때  여러 개의 소프트웨어 컴퍼넌트 들이 커뮤니케이션 해야할 경우 (특히 많은 데이터들을)  Ex) 데이터베이스 |
| 에이전트의 확장성과 재사용성  백업과 restore 하기가 쉬움 |
| Data store의 신뢰성과 가용성이 중요한 이슈  에이전트와 데이터 구조와의 높은 의존성  데이터가 분산되어 있을 경우 옮기는 비용이 비쌈 |
| Blackboards | Raw data를 high level data로 transform 해야 할 경우  독립적인 시스템이 같은 데이터 스트럭쳐를 사용하는데 이를 관리해주는 코디네이터가 필요할 경우  중간에서 컨트롤러가 rule-base로 코디네이팅을 함  Ex) vision, Image recognition, speech recognition |
| 확장성 : knowledge source를 더하거나 업데이트 하기 쉽다  Knowledge source를 병렬로 처리할 수 있음  Agent의 Knowledge source 를 재사용하기 쉬움 |
| Knowledge source와 black의 높은 의존성으로 인해 black board의 구조가 변경되면 모든 agent에 영향이 감 |
| Interaction-Oriented | MVC | 필요해서 그냥 partition 한것은 MVC로 보기 쉽지않다.  View가 변경되도 쉽게 적응할 수 있는 구조에서 사용  (high cohesion) |
| 많은 Vendor 프레임웍을 이용할 수 있음  여러 개의 뷰가 같은 모델과 인터렉션 함  Changeability 뷰  그래픽 기반의 시스템에서 효과적인 아키텍쳐 |
| Agent-oriented-application 에서 적합하지 않음  뷰와 컨트롤러의 구분이 명확하지 않을 때가 많음  어떤 데이터모델에 많은수의 뷰와 컨트롤러가 매칭 되어 있으면 변경되었을 때 비용이 큼 |
| PAC | 어플리케이션이 여러 개의 agents를 갖고, 또한 인터렉티브한 요청이 요구될 때  계층적으로 agent들을 컨트롤 해야할 때 |
| 멀티태스킹과 멀티뷰잉이 가능  에이전트에 대해 재사용성 및 확장이 용이하다  여러 개의 멀티 에이전트를 병렬로 처리할 때 좋다 |
| 시간적 손실 (presentation과 abstraction 사이의 control bridge 때문에)  개발시 복잡도 증가 |

**#GRASP**

**Low coupling : 낮은의존도, 재사용성 증가**

**High Cohension : focus on responsibilities of a class. 같은 일을 하는 애들끼리 묶음**

**Expert : 정보를 가지고 있는 요소가 해당하는 것을 처리하도록**

**Creator : Assign class B가 class A를 품고있을 경우**

**Controller : External event sources와 internal event handler와의 indirect coupling을 위해**

**Don’t talk to Stranger : no reason to be directly aware of each other, then the two classes should not directly interact**

**4.3 Core / Skeleton architecture**

1) Process for Designing Skeleton Architecture

|  |  |
| --- | --- |
| Architecture Requirement Refinement | Target 시스템을 위해 요구사항을 정제  1. Stakeholder 를 identify  2. 기능 요구사항을 정제  \* Typical types of requirement deficiency (완벽하지 않은, 획일성이 없는, 정확하지 않은, 애매모호한, 부적절한)  3. architecture concern을 수집  - stake holder로부터 concern을 수집한 후 NFR 요구사항을 추출한 후 merge 그리고 map을 이용하여 수치화  4. 비 기능 요구사항 정제  - NFR item set을 선택하고, NFR중 실제 accept가 되는 내용을 선별 |
| System context analysis | Architecture decision을 하기 전 target 시스템을 이해하는 것 (도메인 타겟시스템이 명확하지 않을 경우 특히 중요함)  Flow chart, context diagram, block diagram 등을 시스템을 파악하는 것 |
| Skeleton Architecture design | 목적 : initial and high level architecture를 그리는 것  1. functional 요소와 relationship을 identify  2. clustering of functional 요소  3. 후보 아키텍쳐 선정  4. 후보 아키텍쳐 평가  5. 아키텍쳐 선택  6. 만약 여러 개면 merge 한다. |
| View-Specific Architecture Design | 해당 아키텍쳐를 각각의 view로 표현  Functional, information, behavior, deployment 총 4가지 뷰 |
| NFR-Specific Architecture Design | 아키텍쳐를 추가적인 NFR item을 반영하도록 재설계 (다양한 tactic으로 반영될 것임) |
| Architecture Validation | 기능적인면 비기능적인면을 아키텍쳐가 잘 반영하고 있는지 테스트 수행 |

2) how to integrate multiple styles?

#공통의 component와 connector를 찾아라. 간단하게 이름을 비교하기보단 각각의 component와 connector가 갖고 있는 Role을 비교하면서 공통적인 부분을 elicit 해라

#이렇게 공통적인 부분이 없을 경우?

Application-specific connectors를 이용하여 components 들을 merge 해라. Conflict가 났을 경우에는 이를 해결해야 하며, 만약 이러한 conflict가 해결되지 않으면 이 둘은 merge 될수 없다.

Interaction의 이용 : interaction paths among components (function call, invocation, data access, flow of data, event)

3) 왜 merger 를 하는지? Target 시스템은 한 개밖에 없기 때문에 선택된 아키텍쳐들이 seamless하게 merge가 되어야 함. 스타일 중에 완벽한 스타일은 없음

4) 왜 components 사이에 interactor을 정의하는지?

- 일종의 connect 하기 위한 protocol의 역할을 수행

- 외부에서 볼 수 있는 기능적인 declaration

- set of method signatures

**4.4 Architecture Views**

1) 4가지 view

#Functional

-target 시스템의 기능적인 관점의 뷰. Functional elements, responsibility, interfaces, connectors, external entity 관점에서 생각. 다른 뷰들의 동인이자 건물의 뼈대와 같은 초석의 역할을 담당

\* ouput

Use-case

#Information

Architecture의 정보를 어떻게 store, 조종, 관리, 분배 등 주로 데이터 관점의 뷰

\*output

class diagram

ER diagram

#Behavior (Concurrency view)

실행순서의 관점. Runtime에 순서가 어떻게 되는지? 병렬처리인지? 분산처리인지? 등 시스템의 동시에 발생하는 행위의 관점. 예를 들면, process, thread, work flow 등

\*output

Activity diagram

Sequence diagram

State machine diagram

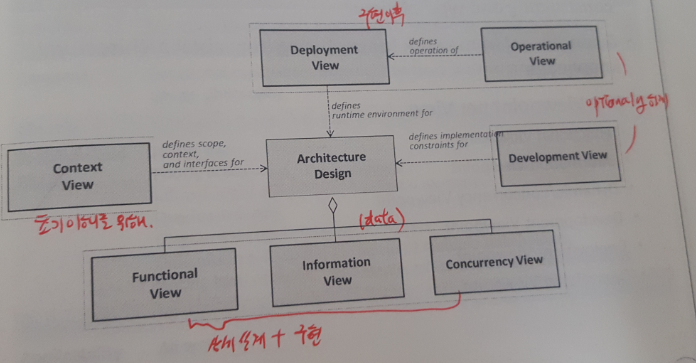
#Deployment views

시스템이 실제 설치 및 배포되는 관점의 뷰를 이야기 하며, 예를 들면 hardware 환경이나 software 환경등이 있을 수 있고, 해당 뷰의 설계는 Quality in use에 영향을 많이 미침

\*output

Deployment diagram

2) Process for Applying each View Points



|  |  |
| --- | --- |
| Functional | Domain knowledge와 use case를 이용하여 기능적 component 찾기 (취급하는 데이터가 같은 것들은 같은 기능적 component로 묶음)  Tier에 따라 function component 정제  Functional component간 interface 추가  View component와의 interface 추가  Data component와의 interface 추가 |
| Information | Class와 entity의 이름 짓기  Data component 기술  Modeling timeliness, distribution or replication of datasets |
| Behavior | Behavior of use-case (use-case 레벨의 workflow)  Behavior of functional component (component 레벨의 workflow)  Behavior of data component (data manipulation) |
| Deployment | 시스템이 deploy될 환경을 기술하기  Artifact, node, communication path 등을 기술 |

**4.5 Architecture Design for NFR (Tactics)**

1) Design for Conventional NFR

일반적인 NFR. 전형적인 cookbooks은 없으나 많은 tactic들이 반영되어 있다.

Security, performance & scalability, availability, evolution, accessibility, location, adaptability, dependability

1. Identify Sensitive Resources: 시스템의 민감한 resource가 어떤 것이 있는지 정의

2. Define the Security Policy :

3. Identify Threats : 위협이 되는 요소들을 identify

4. Design the security impl : 위협을 완화시킬수 있는 방법을 디자인. 후보 tactics들

5. Access the security risks : security risk를 평가

2) Design for Non-Conventional NFR

제품의 innovative한 feature에서 나오는 것

1. identify underlying facts and policies : 이미 잘 알려진 assumptions, consensus, 룰, 규정

2. define desirable quality criteria : 알려져있는 fact를 만족하면 이렇게 되며 좋겠어 하는 희망 (should be)

3. define candidate tactics for the criteria : criteria를 만족할 수 있는 전략

4. evaluate the candidate tactics : cost와 benefit의 관점에서 tactcs에 대한 평가 수행

5. analyze impacts of tactics on views : 해당 tactics가 각 view에 어떠한 영향을 끼치는 분석

6. apply the tactics on architecture views : 아키텍쳐에 tactic 반영

3) Impact of Tactics on Views

|  |  |
| --- | --- |
| Functional | Adding or removing component  Refining functional component with attribute, operations and relationships |
| Information | Refining data components |
| Behavior | Defining / refining algorithms |
| Deployment | Adding and removing nodes  Changing the network environment |

**4.6 Architecture Validation**

\*프레임웍 : 아키텍쳐가 잘 구현되어 있는 코드

\*waterfall 방식이 맞는 system이 있지만, 대부분은 그렇지 않음

\*

1) ATAM (Architecture tradeoff analysis method)

품질속성 시나리오에 기반하여 아키텍쳐 tradeoff를 분석하고 이에 대한 위험요소(Risk)를 찾아내는 아키텍쳐 평가 방법

|  |  |  |
| --- | --- | --- |
| 소개 | ATAM 소개 | 참가자들에게 각 단계별 진행 방식 이해 시키기 위해 아키텍쳐 평가 리더가 ATMA의 정의 및 목적 설명 |
| 비즈니스 동인 소개 | 과제배경 / 기능 요구사항 / 품질속성 / 제약사항 / 용어 |
| 아키텍쳐 소개 | 상위 수준 아키텍쳐 (설명하기 용이한 아키텍쳐 뷰로) 이용하여 시스템의 요구사항과 어떻게 그것이 맵핑이 되었는지 소개. 그러면서 이슈와 리스크도 같이 병행 소개 |
| 조사와 분석 | 아키텍쳐 접근법 식별 | 아키텍쳐 결정사항 추출 => 즉, 품질요구사항 |
| 품질속성 유틸리티  트리 작성 | 품질 속성 시나리오 작성 후 우선순위별로 ordering을 함 |
| 아키테쳐 접근법  분석 | 아키텍쳐 접근법이 품질속성에 적합한지 검사 (민감점, 절충점, 위험, 무위험) |
| 테스트 | 브레인 스토밍과  시나리오 우선순위 결정 | 다양한 stakeholder들을 상대로 브레인스토밍을 진행하면서 시나리오 재정비, utility tree 재정비 |
| 아키텍쳐 접근법  분석 반복 | 반복 |
| 보고 | 결과 보고 |  |

#수행효과

가. 개발자들에게 아키텍쳐 설계 원칙을 공유가능. 그때 아키텍쳐 설계에 대한 논리적인 기준 및 근거 제시 가능

나. 대안 아키텍쳐에 대한 분석 및 고민 유도 및 위험 관리 수행

#단점

ATAM이 너무 시간과 비용. 즉, cost가 고려되지 않아서 향후에 cost의 요인으로 진행이 불가능할 경우가 있음

2) CBAM

아키텍쳐를 기반으로 비용, 이익등을 분석하여 아키텍쳐 판단을 내리는 경제 측면의 평가 방법

|  |  |
| --- | --- |
| 시나리오 수집하기 | ATAM에서 수집한 품질속성 시나리오를 인계받거나 새로운 시나리오를 개발하여 ordering 후 상위 30% 선별 |
| 시나리오 정제하기 | Best, worst, current, desired의 속성을 구분하여 정제 |
| 시나리오 우선순위 | 기대 반응값을 고려하여 우선순위 투표하여 상위 1/2 선별 |
| 시나리오 별 utility  정하기 | 선별한 시나리오에 utility값 정함(best : 100 / worst : 0 / current, desired : stake holder들이 정함) |
| 시나리오 예상값에 따른 아키텍쳐 전략 만들기 | 시나리오 관련된 아키텍쳐 접근법을 결정(ATAM 아키텍쳐 접근법) |
| 예상utility수준 계산 | 예상 utility를 계산 |
| 전체이익 계산 | 모든 시나리오에 대한 이익 계산 |
| ROI 계산하여 순위결정 | 아키텍쳐 접근법을 실현하는데 필요한 비용과 기간을 산정하여 ROI를 계산하고 이를 기준으로 접근법 순위 결정 |

단점 : 절차가 복잡하고, reasoning을 제대로 하지 않으면 시나리오의 우선순위에 너무 큰 영향이 간다.

3) 비교

|  |  |  |
| --- | --- | --- |
| 항목 | ATAM | CBAM |
| 특징 | 품질 속성간 상충관계 | 품질 속성의 경제성 중심 |
| 시점 | 프로젝트 초기단계 | ATAM 수행 이후 |
| 시나리오 | 개별 시나리오 | 여러가지 시나리오 |
| 관심사항 | 아키텍쳐 품질 | 이해관계자의 이익 |
| 장점 | 기존 시스템 분석 가능 | 비용, 일정, 위험 파악 |
| 단점 | 경제성 평가 곤란 | 품질속성 평가 미흡 |

4) others

|  |  |
| --- | --- |
| 코드리뷰 | 개발자가 직접 소스코드를 보면서 코드의 품질을 체크 |
| Cyclomatinc complexity | 하나의 function이 얼마나 복잡한지를 정량적으로 표현 |
| OO Metric | 재사용성, 유지보수성 |
| 구조분석 | 소스의 디렉토리와 소스간의 사용관계 |
| DSM | Dependency 검사 |
| 프로토타입 | 성능평가에 좋음. 아키텍쳐의 핵심 요소를 범위로 잡음 |

**5. Refactoring**

Software의 internal structure를 lower cost로 maintain 및 확장할 수 있도록 변경하는 것

1) Design smell

|  |  |
| --- | --- |
| Rigidity | 시스템을 변경하는 것이 어려움 |
| Fragility | 한쪽의 변화가 다른쪽에 영향을 주는 것 |
| Immobility | 모듈화가 힘듦 |
| Viscosity | 시스템의 룰을 고수하면서 고치는게 너무 어려움 |
| Needless Complexity | 필요이상으로 복잡한 것 |
| Needless Repetition | 필요이상으로 코드중복이 많은 것 |
| Opacity | 복잡한 관계를 설명하기 어려움 |

1) Refactoring Process

Target system 이 있을 때, 단위 별로 refactoring을 수행하고, 해당 단위를 unitTest 이러한 과정을 smell이 없을 때까지 반복

주의) 한번에 refactoring을 모두 수행하고, test 하는 것이 아님.

2) Refactoring Instructions

Extract method, move, encapsulate,

|  |  |  |
| --- | --- | --- |
| Bloaters | Long method | 너무 긴 메소드 |
| Extract Method |
| Large class | 너무 방대한 클래스 |
| Extract Class and Extract Subclass |
| Primitive Obsession | ex) 전화번호에 string 쓰는것. 파싱하는 부분넣어야함. Util에 빼서 넣을 것인지? |
| concept으로 사용할 수 있는 것은 과감하게 abstraction |
| Long parameter List | 이해하기 어려울 정도로 긴 파라미터 |
| Introduce parameter object |
| Object-Orientation Abusers | Switch statements | 복잡한 Switch state 같은 것 |
| Polymorphism을 적용해야 함 (extract method -> move -> replace type code with subclass) |
| Temporary Field | A variable is in the class scope, when it should be in a method |
| Use extract class |
| Refused Bequest | 상속받은 method를 거부하겠다. Ex) 새 – 펭귄 |
| Create a new subclass |
| Alternative Classes with Different Interface | 두개의 다른 클래스가 같은 내용을 하는 것 |
| Rename method to make them identical |
| Change Preventers | Divergent change | 다중인격자. 한가지 클래스가 다양한 이유로 다양한 방법으로 변화해야됨 |
| Identify each cause of change, use Extract class to isolate |
| Shotgun surgery | Responsibility가 여기저기 흩어져 있는 것 |
| Consolidate responsibility in a single class remove redundant classes |
| Parallel inheritance hierarchy | 클래스의 서브 클래스를 만들면 다른 클래스의 서브클래스도 바꿔야 할 때 (1:1 매칭) |
| Merge class hierarchies |
| Dispensable | Duplicate Code | 코드중복 |
| Extract method 로 새로 만들어서 빼라 |
| Lazy Class | 요지보수 하기 어려운 클래스 |
| Components that are near-useless should be given the inline class |
| Data Class | 하는일 없이 데이터만 저장하는 클래스 |
| Move client code closer to the data by moving and extract method |
| Dead code | 사용하지 않는 코드 |
| 지워라 inline 해라 |
| Speculative Generality | 사용하지 않는 클래스 함수 파라미터 |
| 합치거나 지워 |
| Couplers | Feature Envy | 본인이 처리할 일이 아닌데 굳이 데이터를 가져와서 처리하려고 함 heavy use. |
| Move 하던가 extract method로 뽑던가s |
| Inappropriate Intimacy | 부적절한 친밀감 |
| Move 하던가 abstract method로 뽑던가 |
| Message Chains | 클라이언트가 1개 요청했는데, 함수의 꼬리를 물고물어서 계속 호출되서 결과가 나옴 |
| Hide delegate |
| Middle Man | 중계역할만 하는 함수 (일부로 만들기도 함) |
| 지우던가 inline 하던가 |
| Incomplete Library Class | 필요한 라이브러리는 있는데 거기에는 해당하는 원하는 내용이 부족할 때 |
| Use introduce foreign method |

3) Re-engineering for Different Paradigms. (From C programs to OOD)

C => 코드중복이나 구조화되지 않은 스파게티소스. 가시적이지 않은 변수 등 일반적인 것들에 대한 refactoring이 가능

OOD principle을 기반으로 한 refactoring이 가능하다.