

요구사항정의에서 코딩까지

엄진영

홈페이지: <http://www.jinyoung-eom.kr>

저작권 및 배포안내

- ▶ 이 저작물은 참고 문헌을 토대로 “엄진영”이 재 가공하였습니다.
- ▶ 내용의 일부는 참고 문헌의 저작권자에게 있음을 명시합니다.
- ▶ 이 저작물의 인용이나 배포 시 이점을 명시하여 저작권 침해에 따른 불이익이 없도록 하시기 바랍니다.
- ▶ 위 사항을 명시하는 한 자유롭게 배포하실 수 있습니다.

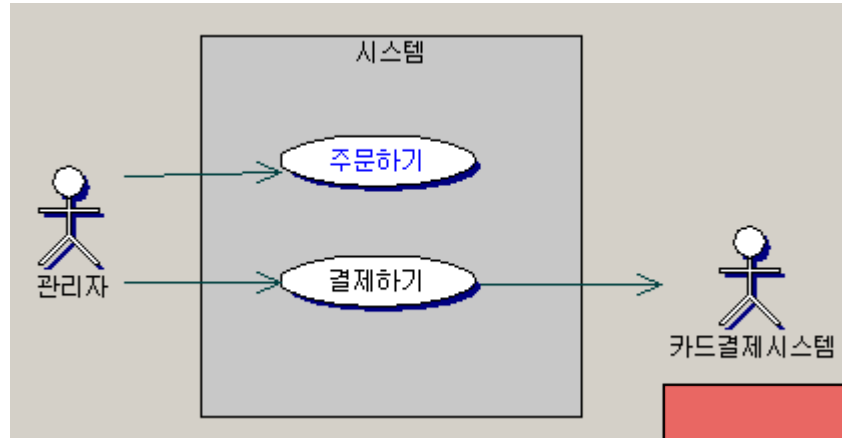
참고문헌

- ▶ Applying UML and Pattern, third edition.
 - [Craig Larman, Addison Wesley Professional, October 20, 2004]
- ▶ IBM's Rational Method Composer
 - Rational Unified Process Document
- ▶ Applying Use Cases 2nd
 - [A Practical Guide] – Addison Wesley
- ▶ Applying Use Case Driven Object Modeling with UML
 - [an annotated E-Commerce Example] – Addison Wesley
- ▶ UML and The Unified Process
 - [Practical OOA/OOD] – Addison Wesley
- ▶ EJB Design Pattern
- ▶ Design Pattern of GOF
- ▶ Rational RUP WEB Document

차례

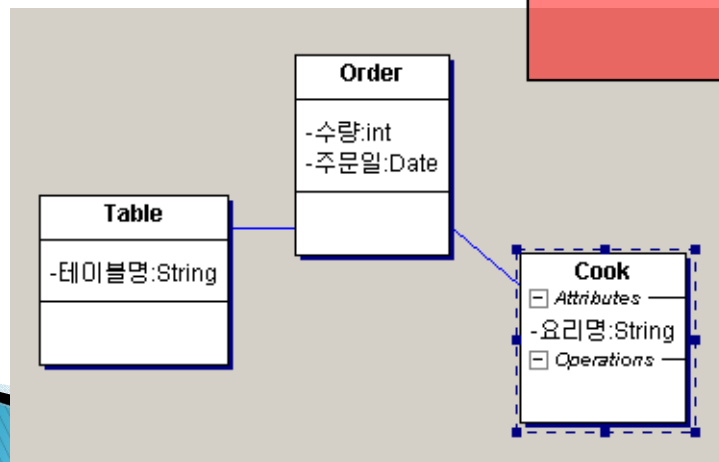
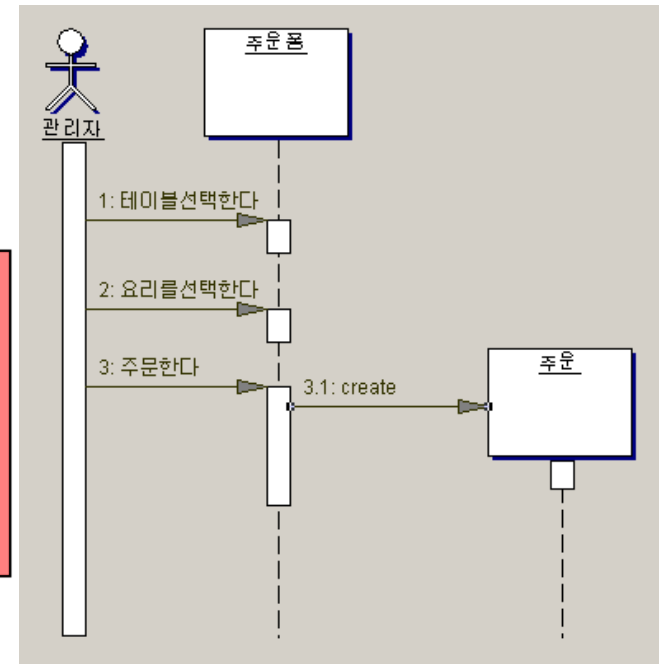
- ▶ 시스템 모델링
- ▶ 개발 프로세스
- ▶ 요구사항 정의
 - Use Case 명세서
- ▶ 분석-도메인 모델
- ▶ Robustness Analysis
- ▶ 설계-Sequence Diagram
- ▶ 패턴의적용 I
- ▶ GRASP 패턴
- ▶ 패턴의적용 II
- ▶ 코드작성

시스템 모델링



[사용의 측면]

[행위적인 측면]



[개념적인 측면]

▶ 사용의 측면

- 사용자의 관점에서 시스템을 사용하는 것을 모델링
- Actor 와 Use Case를 찾고 Use Case Text를 작성

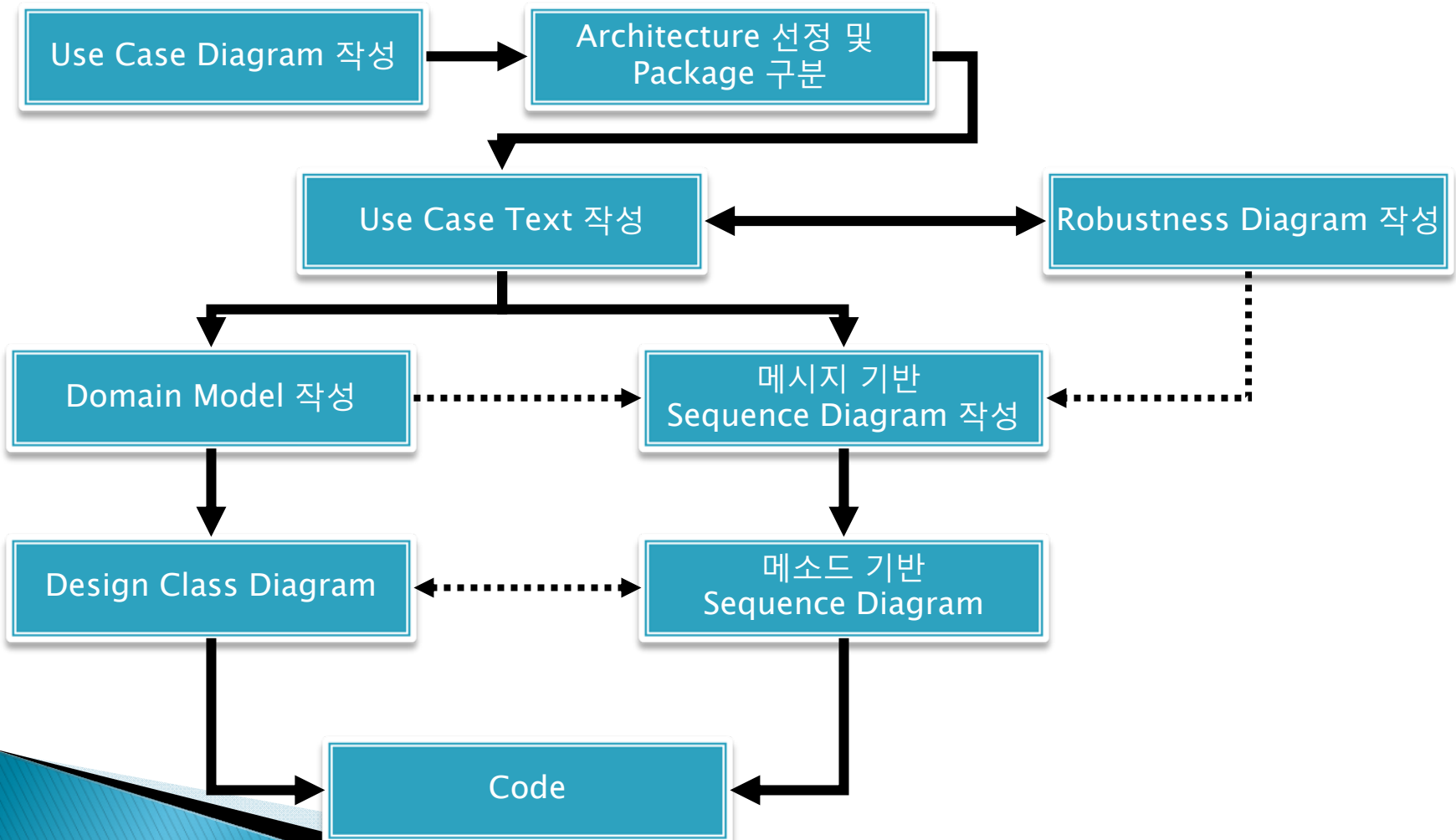
▶ 개념적인 측면

- 현재의 도메인을 개별적인 개념적 클래스들이나 객체들로 분해
- Class Diagram 작성

▶ 행위적인 측면

- 작업을 완성하기 위해 객체들이 서로 협력하는 것을 모델링
- Sequence Diagram 및 Collaboration Diagram을 작성

개발 프로세스



요구사항 정의

▶ Use Case Diagram 작성

- 시스템경계 선정
 - 개발하고 하는 시스템 그 자체
 - 주요 Actor를 정의하면 경계가 분명해짐
- Actor 찾기
 - 주요 Actor – 시스템의 서비스를 이용함으로써 사용자 목표 달성
 - 보조 Actor – 시스템에게 서비스를 제공
- Use Case 찾기
 - 사용자의 목표를 찾기 → 각 목표에 대해 Use Case를 정의
 - 사용자의 목표를 돕는 부 목표(부 기능목표)도 가끔 Use Case로 구분된다.
 - 부 기능이 반복되거나 여러 사용자 목표 Use Case의 전제 조건이 되는 경우.
 - 예) 자신의 신분을 확인시키고 인증 받는다 → Login
 - 업무 프로세스 식별 → 각 EBP (Elementary Business Process)별 Use Case 정의
 - 가시적이고 측정 가능한 가치 있는 비즈니스
 - 부 업무가 다른 Use Case와 중복되어 사용될 때는 별도 Use Case로 분리

▶ 주 Actor 파악 지침

- 정보를 제공하고, 사용하고, 삭제하는 사람은 누구인가?
- 이런 기능들을 사용하는 사람은 누구인가?
- 어떤 요구사항에 대해서 관심을 가지는 사람은 누구인가?
- 조직 내에서 사용될 시스템이 있는 장소는?
- 시스템의 유지보수 및 관리를 하는 사람은 누구인가?
- 시스템의 외부 자원은 무엇인가?
- 이 시스템과 상호 작용할 다른 시스템은 무엇이 있는가?
- 시스템을 시작하거나 종료하는 사람은 누구인가?
- 사용자와 보안을 관리하는 사람은 누구인가?
- 시간 이벤트에 반응해서 시스템이 어떤 것을 처리한다면 시간을 행위자로 볼 것인가?
- 시스템 장애가 발생하면 재 시작시키는 모니터링 프로세스가 존재하는가?
- 시스템이 활동이나 성능을 평가하는 사람은 누구인가?
- 소프트웨어는 어떻게 업데이트하는가? 내보내기(push)나 내려 받기(download) 업데이트 중 어느 것인가?
- 로그를 평가하는 사람은 누구인가? 로그는 원격지에서 추출하는가?

▶ 간과할 수 있는 다른 Actor와 목표 찾기

- 누가 시스템을 시작하고 종료하는가?
- 누가 사용자와 보안을 관리하는가?
- 누가 시스템을 관리하는가?
- 시스템이 시간 이벤트에 반응하여 무엇인가를 하기 때문에 "시간"이 Actor 인가?
- 시스템이 고장 났을 때 다시 시작시키는 모니터링 프로세스가 있는가?
- 누가 시스템 활동과 성능을 평가하는가?
- 소프트웨어 업데이트는 어떻게 처리되는가?
- 누가 로그(log)를 평가하는가? 로그를 원격검색 할 수 있는가?

▶ 행위자 목표 파악 지침

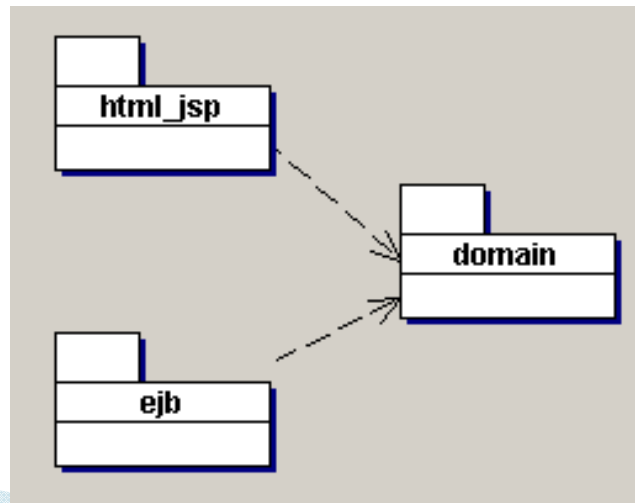
- 행위자는 무엇을 하고자 하는가? 또는 행위자의 목표는 무엇인가? (예: 로그인)
- 해당 목표의 목표는 무엇인가(목표의 계층화)? (예: 사용자인증 → 자료유출 방지)

▶ 쓰임새 식별 지침

- 기본적으로 각 목표에 단위 업무 프로세스(EBP) 하나 수준의 쓰임새 작성한다.
단위 업무 프로세스(EBP)란, 한 사람이 한 시점에 한 장소에서 업무 이벤트 하나에 반응하여, 측정 가능한 업무 가치를 얻고, 데이터를 일관성 있는 상태로 유지하는 작업.
 - 예) 목표: 판매처리하다 → 쓰임새: 판매처리
영어) 목표: Process a sale → 쓰임새: ProcessSale
- 별도의 삽입, 조회, 갱신, 삭제(Create/Retrieve/Update/Delete 의 문자를 따서 CRUD 라고 한다) 목표는 "관리 쓰임새" 하나로 통합한다.
 - 예) '사용자등록', '사용자조회', '사용자갱신', '사용자삭제' → '사용자관리'
- 서로 깊은 연관이 있는 목표들을 하나의 쓰임새로 통합하는 것이 효율적이다.

▶ Architecture 선정 및 Package 구분

- 사용할 아키텍처를 정의 한다
- 각 계층별 패키지를 만든다.
- 예) 웹 기반 C/S 환경에서의 3-tier
 - Html_jsp – domain – ejb
- Use Case들 도 관련 목표나 업무에 따라 패키지로 묶는다.



Use Case 명세서

▶ 간결한(Brief) 형식

- 보통 주요 성공시나리오를 한 문단으로 간결하게 요약

▶ 자유(Casual) 형식

- 비형식적인 문단형태. 다양한 시나리오를 여러 문단에 걸쳐 표현한다.

▶ 갖춘(fully dressed) 형식

- 가장 상세한 형태. 모든 단계와 변형을 상세히 기록한다. 전제조건(precondition)이나 성공보장(success guarantee)과 같은 보조항목들을 둔다.

분석-도메인 모델

▶ 개요

- 관심 있는 도메인에서 개념적 클래스와 실 세계 객체들을 가시적으로 표현
- 도메인 객체 모델, 분석 객체 모델이라 부른다.
- Class Diagram 을 이용하여 표현. 이때, 오퍼레이션은 정의되지 않는다.
 - 도메인 객체들 또는 개념적 클래스들
 - 개념적 클래스들 사이의 연관(association)
 - 개념적 클래스들의 속성(attribute) – 주요속성
- 주목할 만한 추상화, 도메인 어휘, 도메인 내의 정보에 대한 가시적인 사전 (visual dictionary)으로 볼 수 있다.
- 개념적 클래스들을 큰 단위로 찾아내기 보다는 잘게 쪼개어 작은 단위로 찾아내는 것이 더 좋은 방법이다.
- 요구사항에서 필요로 하지 않거나 속성이 없다고 배제하지 말아라.
- 속성이 없는 개념적 클래스, 또는 정보 역할 대신 도메인에서 단순한 행위 역할을 하는 개념적 클래스들을 포함하는 것도 가능하다.
- 초기 도메인 모델에서는 다중성을 고려하지 말라.

- ▶ 개념적 클래스 식별 지침
 - 개념적 클래스 카테고리 목록
 - Use Case에서 명사 또는 명사구

▶ 개념적 클래스 카테고리 목록

- 물리적 혹은 만질 수 있는 객체들
 - 예) Register, Airplane
- 사물에 대한 명세, 설계, 혹은 기술들
 - 예) ProductSpecification, FlightDescription
- 장소
 - 예) Store, Airport
- 트랜잭션
 - 예) Sale, Payment, Reservation
- 트랜잭션 라인아이템
 - 예) SaleLineItem
- 사람의 역할
 - 예) Cashier, Pilot
- 다른 것들을 담는 컨테이너
 - 예) Store, Bin, Airplane

- 컨테이너에 담긴 것들
 - 예) Item, Passenger
- 현 시스템 외부의 컴퓨터나 전자 기계적 시스템
 - 예) CreditPaymentAuthorizationSystem, AirTrafficControl
- 추상명사 개념
 - 예) Hunger, Acrophobia
- 조직
 - 예) SalesDepartment, ObjectAirline
- 이벤트
 - 예) Sale, Payment, Meeting, Flight, Crash, Landing
- 프로세스(보통 개념으로 표시되지 않지만 그럴 수도 있다)
 - 예) SellingAProduct, BookingASeat
- 법칙과 정책
 - 예) RefundPolicy, CancellationPolicy
- 카탈로그
 - 예) ProductCatalog, PartsCatalog

- 재정, 작업, 계약, 법률적 사항의 기록
 - 예) Receipt, Ledger, EmploymentContract, MaintenanceLog
- 재무 도구와 서비스
 - 예) LineOfCredit, Stock
- 매뉴얼, 문서, 참고문헌, 책
 - 예) DailyPriceChangeList, RepairManual

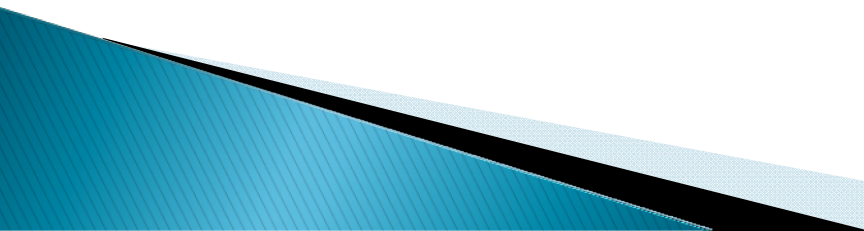
▶ 연관(Association) 찾기

- 개념 및 객체들간의 의미 있고 관심 있는 관계를 표시한다.
- UML에서는 "서로의 인스턴스들이 연결된 둘 이상의 분류자들(classifier)들에 대한 의미적인 관계" 라 정의한다.
- 종류
 - Need-to-know : 어느 일정기간 동안 지속될 필요가 있는 관계에 대한 연관.
 - Comprehension-only : 도메인을 이해하는 데 도움을 주는 연관.
- Need-to-know 연관을 강조하라. 또한, 도메인에 대한 결정적인 이해를 돕는 'comprehension-only' 연관도 포함시켜라.
- 불필요하거나 도출 가능한 연관은 나타내지 않는다.
- 공통 연관 목록으로부터 도출한 연관.
- 개념적 클래스들을 찾는 것이 연관을 찾는 것 보다 더 중요하다. 도메인 모델을 생성하는 데 드는 시간의 대부분을 연관보다는 개념적 클래스를 찾는 데 집중하라.

▶ 공통 연관 목록

- A는 B의 물리적인 한 부분이다.
 - 예) Drawer – Register, Wing – Airplane
- A는 B의 논리적인 한 부분이다.
 - 예) SaleLineItem – Sale, FlightLeg – FlightRoute
- A는 B에 물리적으로 포함된다.
 - 예) Register – Store, Item – Shelf, Passenger – Airplane
- A는 B에 논리적으로 포함된다.
 - 예) ItemDescription – Catalog, Flight – FlightSchedule
- A는 B에 대한 기술이다.
 - 예) ItemDescription – Item, FlightDescription – Flight
- A는 트랜잭션 또는 보고서 B의 라인 아이템이다.
 - 예) SalesLineItem – Sale, MaintenanceJob – MaintenanceLog
- A는 B에 알려진다/기록된다/보고된다/파악된다.
 - 예) Sale – Register, Reservation – FlightManifest

- A는 B의 구성원이다.
 - 예) Cashier – Store, Pilot – Airline
- A는 B의 조직상 하위 단위이다.
 - 예) Department – Store, Maintenance – Airline
- A는 B를 사용하거나 관리한다.
 - 예) Cashier – Register, Pilot – Airplane
- A는 B와 정보를 교환한다.
 - 예) Customer – Cashier, ReservationAgent – Passenger
- A는 트랜잭션 B와 관련된다.
 - 예) Customer – Payment, Passenger – Ticket
- A는 트랜잭션 B와 관련된 트랜잭션이다.
 - 예) Payment – Sale, Reservation – Cancellation
- A는 B의 옆에 있다.
 - 예) SalesLineItem – SalesLineItem, City – City

- A를 B가 소유한다.
 - 예) Register – Store, Plane – Airline
 - A는 B와 관련된 이벤트이다.
 - 예) Sale – Customer, Sale – Store, Departure – Flight
-
- ▶ 도메인 모델에 포함될 필요가 있는 주요 연관
 - A는 B의 물리적 혹은 논리적 부분이다.
 - A는 물리적으로 혹은 논리적으로 B에 포함된다.
 - A는 B에 기록된다.
- 

▶ 속성(attribute) 찾기

- 정보를 기억할 필요가 있는 속성이 무엇인가를 요구사항 (Use Case등)을 통해 결정하고, 도메인 모델에 이 속성들을 포함시킨다.
- 간단한 속성이거나 데이터 타입인 것이 좋다.
 - 가장 일반적인 속성타입: Boolean, Date, Number, String, Time
 - 다른 일반적인 타입: Address, Color, Geometrics (Point, Rectangle), PhoneNumber 등
- 확신이 서지 않을 때는 속성으로 정의하는 것 보다 개별적인 개념적 클래스로 정의하라.
- 속성을 외래 키로 사용하지 말라
 - 개념적 클래스들을 속성이 아닌 연관으로 관련시켜라.
 - 관계에 대한 구현은 설계단계로 미뤄라.
- 도메인 모델은 의사소통 도구이다. 항상 이 사실을 명심하면서 도메인 모델에 무엇을 나타낼 것인가를 선택하는 것이 좋다.

Robustness Analysis

▶ 개요

- Ivar Jacobson의 Objectory method의 일부
- 각 Use Case에 필요한 객체가 무엇인가?
- Design Class Diagram의 기초를 제공한다.
- Sequence Diagram의 기초를 제공한다.
- Use Case에서 Sequence Diagram 으로 나아가는데 도움을 준다.
- 개발자에게 “**What**” 의 관점에서 “**How**” 관점으로 이동을 돕는다.

▶ Robustness 객체들

◦ Boundary 객체

- Actor 가 시스템과 대화할 때 사용한다.
- GUI 객체 – Window, HTML
- 명사로 표현된다.

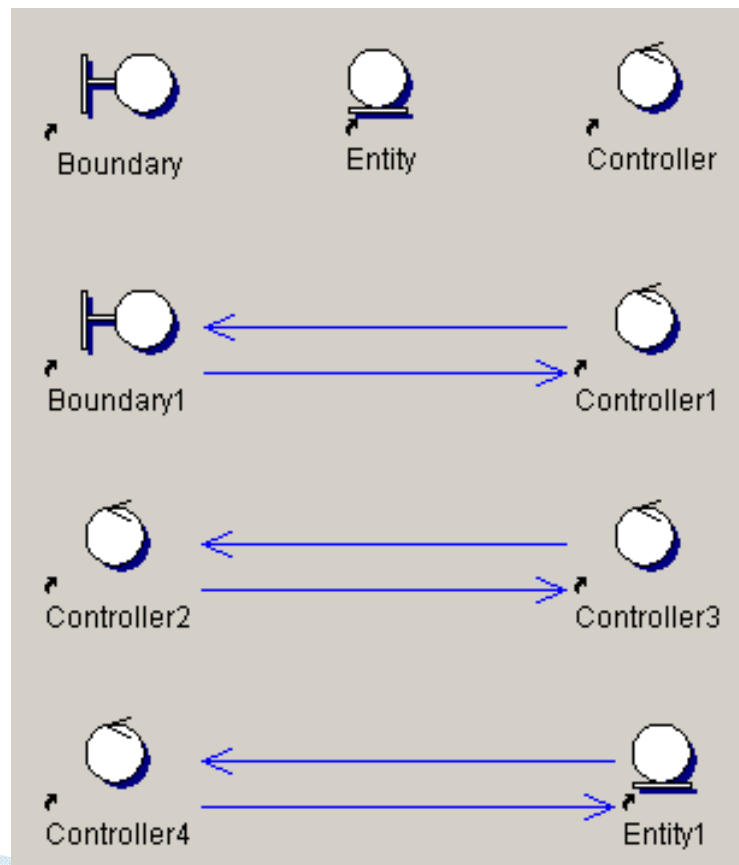
◦ Entity 객체

- 보통 도메인 모델의 객체. 지속적으로 유지되는 정보를 나타내는 객체
- Use Case 가 끝나도 지속되는 객체. 때론 Use Case 내에서 일시적으로 사용되는 객체.
- 명사로 표현된다.

◦ Control 객체

- Boundary 와 Entity 사이에 중계를 해준다.
- Business logic을 표현. 설계 시 “**객체**”로서 구현 되기도 하지만, 대부분 메소드로서 구현 된다. 만약, 웹 기반 프로젝트이면 JSP/Servlet으로 구현된다.
- 동사로서 표현된다.

▶ Robustness Diagram 작성 규칙



▶ Robustness 와 Use Case

- Use Case Text에서 시나리오의 각 단계를 따라가면서 Robustness Diagram을 작성하라
- Boundary를 찾아라
- 찾은 Boundary 를 적용하여 Use Case Text를 갱신하라.
- Use Case Text의 기본흐름, 대안흐름, 예외흐름(선택)을 모두 표현하라.

설계-Sequence Diagram

▶ 개요

- 요구사항을 충족시키기 위해 객체들이 어떻게 협력하는가를 보여준다.
- Design Class Diagram을 작성하는데 이용된다.
 - Sequence Diagram을 작성하면서 각 클래스의 메소드가 식별된다.
 - 각 클래스에 대해 책임이 할당된다.
- 메시지의 순서나 시간적인 순서를 명확하게 보여준다.
- Collaboration Diagram 과 스위칭 가능.
 - 공간에 따라 객체들을 배열하여 메시지를 나열한다.
 - 복잡한 분기, 반복, 동시적인 행동을 표현하는데 더 적합

▶ 작성 규칙

- Use Case Text에서 기본흐름, 대안흐름, 예외흐름(선택)을 복사하여 왼쪽에 붙인다.
- Robustness 분석에서 나온 Boundary 와 Entity 객체를 붙인다.
- Use Case Text 와 Robustness Diagram을 보면서 메시지를 그린다.
- Robustness에서의 Control 은 어떻게 표현하는가?
 - UI 관련 제어
 - Boundary 객체의 메시지로 표현
 - 웹: HTML 링크 또는 JavaScript 메소드로 표현
 - Use Case 관련 제어 (Business Logic)
 - Entity 객체의 메시지로 표현
 - 웹: JSP/Servlet으로 표현, EJB Session Bean
 - 책임할당 패턴을 적용하여 메시지를 표현한다.
 - Use Case 하나나 몇 개를 묶어서 하나의 Control 객체로 표현하기도 한다.
- 설계 패턴이 적용되기 전까지 메소드로 표현하지 말아라

패턴의 적용 I

- ▶ 책임할당 패턴을 적용하여 추가 클래스 생성
- ▶ 다시 Sequence Diagram 개정
 - 책임할당 패턴에 의해 추가된 클래스의 객체 표시
 - 메시지 변경
 - 메시지를 메소드로 변환

GRASP 패턴

▶ 개요

- General Responsibility Assignment Software Patterns 의 약자
- 객체에 책임(Responsibility)을 할당하는 기본원리를 방법론적으로 표현
- Sequence Diagram을 그리는 동안 고려
- 패턴 : 원리나 관용법이 “문제와 해결책” 형식으로 요약 정리되고 이름을 붙인 것.
 - 경험에 의해 사실로 밝혀진 기존의 지식, 관용법, 원리들을 조목 별로 요약 정리한 것.
 - 패턴의 이름은 개념을 구체화하여 기억하기 쉽도록 지어진다.
 - 개발자간 의사소통을 원할 하게 한다.

▶ Information Expert 패턴

- 문제

- 책임을 객체에 할당하는 일반적인 원리는 무엇인가?

- 해결책

- 정보 전문가, 즉 책임을 수행하는데 필요한 정보를 가지고 있는 클래스에게 할당하라.
 - 설계모델에 적합한 클래스가 있는지 찾아본다.
 - 없으면, 도메인 모델에서 찾는다.
 - 없으면, 해당 설계 클래스를 생성하도록 한다.
 - 물론, Architecture를 염두 해 두고 적용한다.

▶ Creator 패턴

- 문제

- 누가 클래스의 새로운 인스턴스를 생성할 책임이 있는가?

- 해결

- 가장 일반적인 활동중의 하나.
 - 다음 사항에 해당하면 B는 A객체들의 Creator이다.
 - B가 A 객체들로 구성된다.
 - B가 A 객체들을 포함한다.
 - B가 A 객체의 인스턴스를 기록한다.
 - B가 A 객체를 가깝게 사용한다.
 - A가 생성될 때 A에게 전달되는 초기화 데이터를 B가 가지고 있다.

▶ Low Coupling 패턴

◦ 문제

- 어떻게 의존성을 적게 하고, 변화의 영향을 적게 하며 재 사용성을 증가시킬 것인가?
- 결합도(coupling)는 한 요소가 다른 요소와 얼마나 연관되었나에 대한 척도이다.
- 높은 결합도
 - 관련된 다른 클래스들이 수정되면 그 클래스도 수정해야 한다.
 - 그 클래스 자체만 갖고 이해하기 어렵다.
 - 의존하고 있는 클래스들이 있으므로 그 클래스 자체만으로 재사용하기 어렵다.

◦ 해결

- 결합도를 낮게 유지하도록 책임을 할당한다.
- 모든 설계 결정에 있어서 염두 해 두어야 할 원리.
- Trade-off

▶ High Cohesion 패턴

◦ 문제

- 어떻게 복잡성을 관리할 수 있는 수준으로 유지할 것인가?
- 응집도(cohesion:기능적 응집도)는 한 요소의 책임들이 얼마나 강력하게 관련되고 집중되어 있는가에 대한 척도이다.
- 낮은 응집도
 - 이해하기 어렵다.
 - 재사용이 어렵다.
 - 유지보수하기 어렵다.
 - 민감하며, 변화의 영향을 지속적으로 받는다.

◦ 해결

- 응집도가 높게 유지되도록 책임을 할당
- 유사기능과 서로 밀접하게 관련된 기능들을 묶어라.

▶ Controller 패턴

◦ 문제

- 누가 입력 시스템 이벤트를 처리할 책임이 있는가?
- 시스템 이벤트는 외부의 액터에 의해 생성되는 이벤트이다.
- Controller는 시스템 이벤트를 받고 처리하는 책임을 수행하는 비 사용자 인터페이스 객체이다. Controller는 시스템 오퍼레이션에 대한 메소드를 정의한다.

◦ 해결

- 시스템 이벤트 메시지를 받거나 다루는 책임을 다음 중 하나의 클래스에 할당
 - 전체 시스템, 장치, 혹은 서브시스템을 나타내는 클래스(façade controller)
 - 시스템 이벤트가 일어나는 Use Case 시나리오를 나타내는 클래스
- 일반적으로 Controller는 수행되어야 할 일을 다른 객체들에게 위임하는 것이 좋다. Controller는 활동을 조정하거나 제어하면, 그 자신은 많은 일을 하지는 않는다.

패턴의 적용 II

- ▶ 각 클래스의 필드에 대해 패턴 적용
- ▶ 각 연관에 대해 패턴적용
 - Association
 - A 객체가 지속적으로 B 객체와 관계를 맺는다.
 - Aggregation
 - A 객체가 B 객체를 포함한다.
 - Composition
 - A 객체가 B 객체를 포함한다. B객체의 생명은 A객체에 의해 좌우된다.
 - Dependency
 - A 객체의 메소드에서 B객체를 사용한다.
- ▶ 각 클래스에 대해 패턴적용

코드작성

- ▶ Design Class Diagram 및 Sequence Diagram을 참조하여 직접 작성
- ▶ 분석/설계 툴에 의해 자동 생성