Object-Oriented Analysis & Design

Software Engineering T3 :

200711429 박상욱
200711439 송인하
200711444 양동은

Konkuk Univ. Computer Engineering

Content :

6. 책임 할당을 위한 기본 설계 패턴

7. 설계 클래스 다이어그램

8. 시스템 설계 패턴

9. 시스템 구현

10. 사용 관계 및 타입 관계

11. 패키지와 개념 모델의 확장

12. 행위 모델링

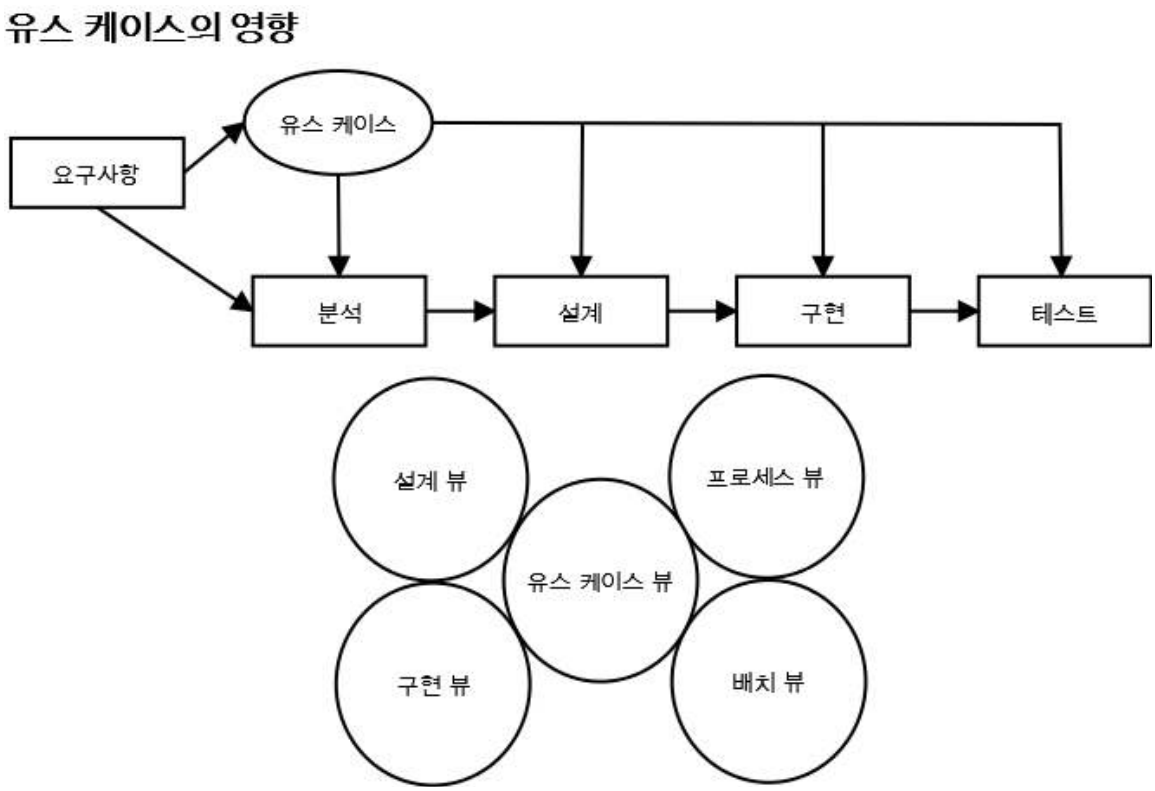
1. Introduction

1.1 UML 사용 프로세스 특성

1.1.1 유스 케이스 위주

UML의 유스 케이스는 시스템의 기능적인 요구사항을 정의 하며 이는 요구사항 분석 이후의 모든 단계에서의 개발을 유도한다. 이는 시스템의 필요한 모든 기능이 실현되는 것을 보장한다. 검증, 테스트를 위해서도 유스 케이스는 구현이 된다. 시스템의 기능에 대한 설명을 포함하기 때문에 시스템 개발의 모든 단계 및 모든 뷰에 영향을 미친다.

유스 케이스는 분선단계에서 시스템의 필요한 기능을 정의하고 있어 사용자와의 의사소통을 위해 사용되며 이는 시스템의 설계 및 구현이 실현되어야 한다는 뜻이다. 또한 테스트 케이스를 위한 기반이 되며 개발 작업을 구성하는 기반이 된다.



1.1.2 아키텍처 중심

UML을 사용하는 프로세스는 아키텍처 중심적이다. 시스템 아키텍처는 매우 중요하기 때문에 프로젝트 초기에 수립을 위해 노력하여야 한다. 모델링 언어의 다양한 뷰에 의하여 반영되며 여러 단계의 반복 과정을 거쳐 개발된다. 그러나 가급적 프로젝트 초기에 기본 아키텍처를 정의하고, prototype을 거쳐 검증하며 개선하는 방식으로 나아가야 한다.

아키텍처는 시스템의 다양한 부분 정의, 이들 간의 관계 와 상호작용, 통신 메커니즘, 규칙 등을 제공하며 기능적인 측면과 비 기능적 측면 모두 설명하여야 한다. 수정할 수 있고, 쉽게 이해하고, 재사용 가능한 시스템 정의가 중요하다.

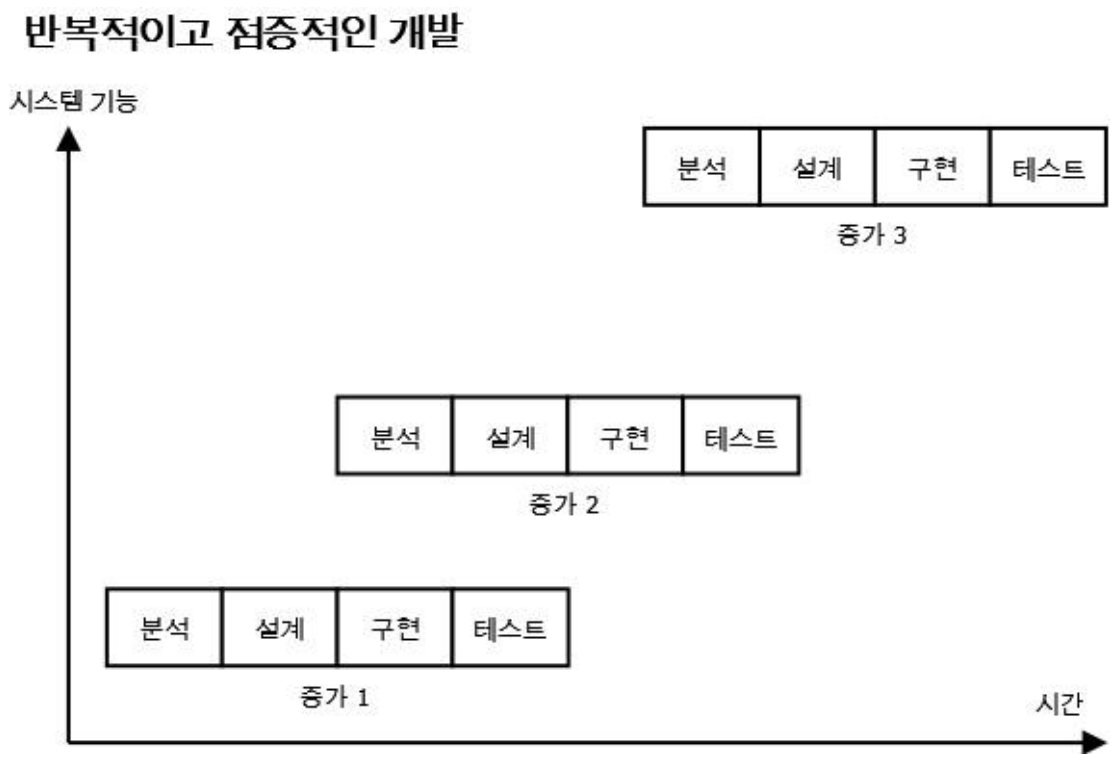
1.1.3 Iteration

모델이나 다이어그램의 모든 세부적인 사항은 한 번에 정의하기 보다는 일련의 반복적인 개발 주기를 통하여 개발하는 것이 바람직하다. 각 개발주기는 새로운 정보, 세부사항을 모델에 추가하며 평가를 거쳐 다음 개발 주기를 위한 입력으로 사용된다. 결국 반복은 최종 산출물을 개선시키는 피드백을 제공한다.

개발 주기에 포함시킬 요소를 결정할 때에는 시스템에 가장 영향을 미치는 요소 위험이 높은 요소 고려가 중요하다. 이러한 시스템의 중요한 문제는 초기 개발주기에 포함되어야 한다.

1.1.4 incremental

개발주기 및 반복적인 개발 주기는 시스템의 increment로 간주할 수 있다. 시스템을 반복적이고 점증적으로 개발할 때 시스템 개발은 기술적 관점, 경제적 관점, 프로세스의 관점에서 평가되고 산출물을 제공한다. 모든 단계는 새로운 기능이나 속성을 추가하여야 하며 이러한 추가 사항은 사전에 계획되어야 하다.



1.2 객체지향 분석과 설계(OOA/D)

1.2.1 책임 할당

OOA/D에서 가장 중요한 능력은 소프트웨어 컴포넌트에 책임을 적절히 할당하는 것이다. 이는 분석과 설계 과정에서 반드시 수행되어야 하며 이후 소프트웨어 재사용성, 유지보수성에 영향을 미친다. 문제 도메인의 적절한 객체 혹은 추상화 식별 또한 중요한 기술이다.

1.2.2 분석과 설계

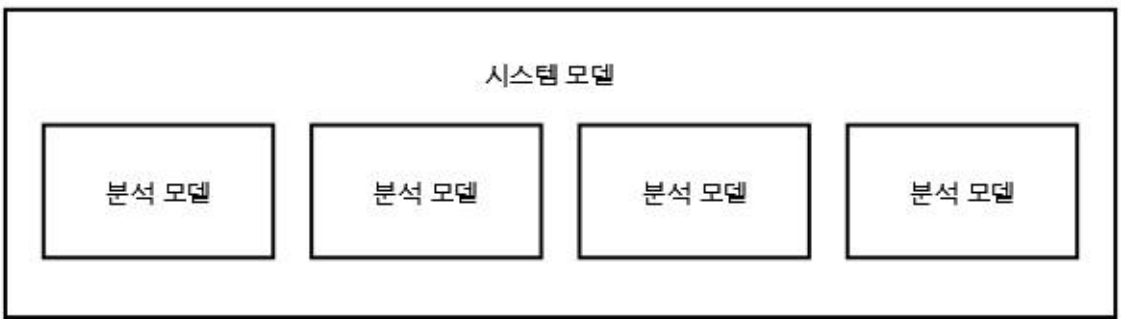
소프트웨어 개발을 위해서는 문제 및 요구사항을 정확히 정의하여야 한다. 분석은 해결책 제시보다는 문제에 대한 조사가 더욱 강조된다. 설계는 시스템의 요구사항을 만족시키는 논리적인 해결책을 강조한다.

OOA/D의 핵심은 문제 도메인과 논리적인 해결책을 객체의 관점에서 고려한다는 것이다. 분석에서는 문제 도메인의 객체 혹은 개념을 식별, 정의한다. 비즈니스 프로세스와 요구사항은 유스 케이스로 표현한다. conceptual model로 도메인의 개념, 애튜리뷰트 및 관계를 표현할 수 있다. 설계는 객체지향 프로그래밍 언어로 구현될 논리적인 소프트웨어 객체를 정의한다. 소프트웨어 객체는 애튜리뷰트와 오퍼레이션을 갖는다.

1.3 UML을 이용하는 객체지향 프로세스

OOA/D는 분석, 설계, 구현, 테스트 단계로 이루어진다. 각각의 단계는 다수의 다이어그램으로 구성되는 각자의 모델을 작성한다.

주요 개발 단계와 모델



1.3.1 요구사항 분석

요구사항 분석은 시스템의 필요한 기능을 설명하기 위하여 유스 케이스, Business Process, 혹은 텍스트 문서를 작성한다. 이는 기술적인 사항에 대한 고려가 아닌 시스템의 외부의 관점에서 분석된다. 대부분 개발자와 사용자간의 토의와 협상으로 이

루어진다.

요구사항을 분석할 때 기능적 사항뿐만 아니라 성능, 신뢰성 등과 같은 비 기능적인 사항과 규모, 기술 환경, 기존 시스템과의 통합 등 과 같은 제약조건도 고려하여야 한다.

요구사항 분석은 개발자와 사용자가 합의한 명세서를 작성한다. entity 개념 모델이나 중요한 용어에 대한 카탈로그 작성도 매우 유용하다.

요구사항 분석에서 사용되는 UML 다이어그램은 Use case diagram, Class diagram, Activity diagram이 있다.

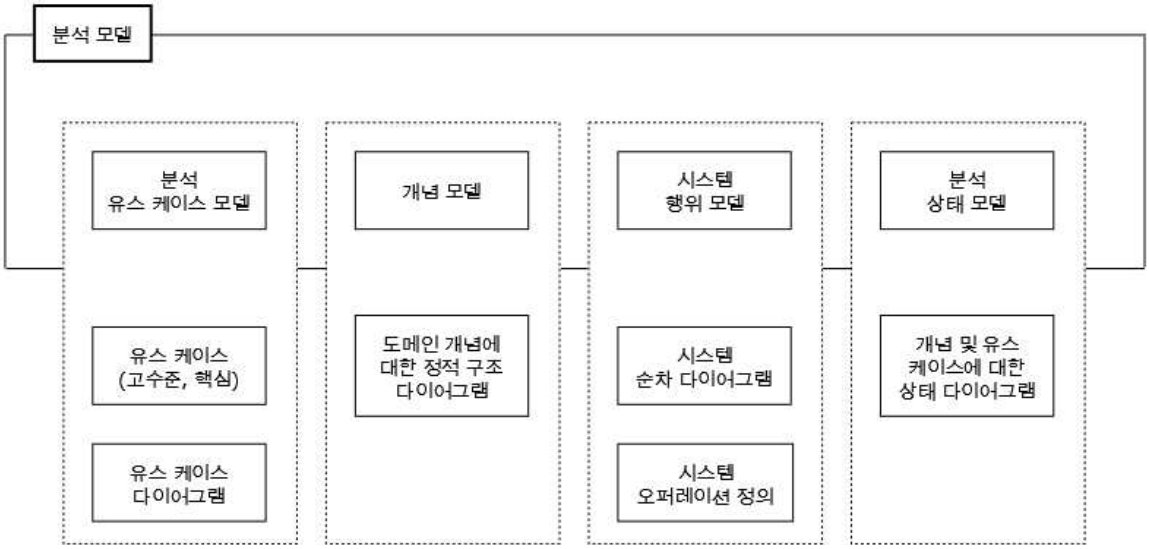
1.3.2 분석

분석 단계는 실세계 entity를 모델링하는 클래스, 객체, 상호작용 등으로 구성되는 문제도메인에 대한 모델을 생성한다. 기술적인 사항이나 구현사항으로부터 자유로워야 하며 문제 도메인을 해결하고자 하는 문제를 정의하는 이상적인 모델만은 다뤄야 한다.

분석 단계의 전형적이 활동
● 요구사항 명세서, 비즈니스 프로세스 모델, 영어 카탈로그, 기존 시스템에 대한 설명, 시스템 사용자와의 인터뷰를 통한 문제 도메인에 대한 지식 확보
● 유스 케이스를 이용한 시스템의 기능 정의
● 후보 클래들을 브레인스토밍 세션에서 찾은 후 부적절한 클래스 제거
● 클래스 간 정적인 관계를 연관, 집단화, 일반화 및 종속 관계를 이용하여 모델링. 이를 문서화하기 위해 Class diagram 사용
● 객체간의 행위와 협동을 Activity diagram, Collaboration diagram, Sequence diagram, State diagram 등을 사용하여 정의
● 모든 다이어그램이 작성되면 시물레이션으로 전체 모델을 검증
● Proto type을 이용하여 기본적 UI 정의 및 사용자와 전반적인 구조 테스트, 토의

분석 문서는 문제 도메인을 설명하는 모델과 필요한 기능을 제공하기 위한 도메인 클래스의 행위로 구성된다. 이는 기술 환경과 기술적인 세부사항을 고려하지 않는 이상적인 시스템을 설명하여야 한다. 분석 단계에서 작성되는 UML 다이어그램은 Class diagram, Sequence diagram, Collaboration diagram, State diagram, Activity diagram 이다. 기술 해결책이 아닌 문제 도메인에 초점을 맞추어야 한다.

분석 모델



1.3.3 설계

설계는 분석 결과에 대한 기술적인 확장 및 적응이다. 클래스 관계 및 협동은 시스템 구현에 초점을 맞춰 새로운 요소로 보완된다. 기술적인 세부사항과 구현 환경에 대한 제약조건을 고려하여야 한다.

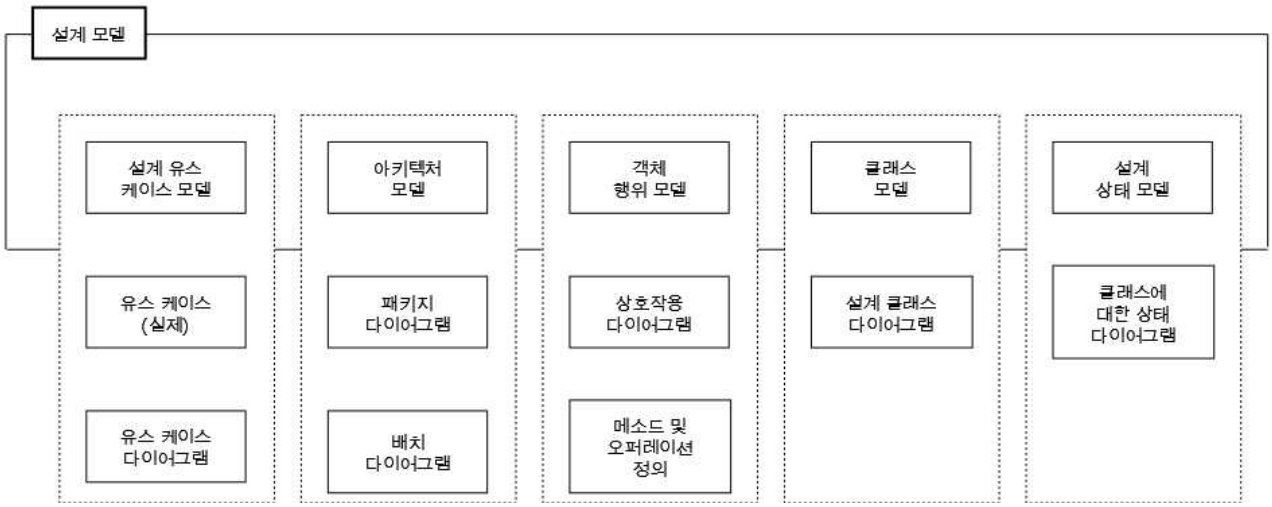
설계 단계에서 전형적인 활동
● 분석 과정의 클래스를 기능적인 패키지로 분할. UI, 데이터베스 처리, 통신 등을 위한 패키지 추가 및 이들의 통신 메커니즘 설정
● 공유 자원처리를 위한 능동 객체, 비동기 메시지, 동기화 기법 사용으로 병행성 소요 식별 및 모델링
● 시스템 출력의 형식 지정
● 아키텍처를 개선, 구현작업을 최소화 시키는 클래스 라이브러리 컴포넌트 구입
● 관계 데이터베이스 사용 시 시스템 클래스를 관계 데이터 모델의 테이블로 매핑
● 시스템 exception과 fault 처리 고려
● Component diagram과 Deployment diagram을 사용하여 소스 코드 컴포넌트에 할당 및 실행 컴포넌트를 노드에 할당

세부적인 설계 활동은 클래스의 구현 애튜리뷰트, 클래스의 상세한 Interface, 오퍼레이션 설명 등 모든 클래스의 명세서를 포함하여야 세부적으로 작성해야 한다. 설계단계에서 작성되는 UML 다이어그램은 Class diagram, Sequence diagram, Collaboration digaram, State diagram, Activity diagram, Component diagram, Deployment diagram이다.

설계 단계에서는 추직성, 인터페이스, 성능, 단순성, 문서화를 기억하여야 한다.

- 추직성 : 원래의 요구사항이 어디에서 유래 되었는가 명백히 하여야 하며, 분석, 설계 및 구현 모델을 각자로부터 분리하여야 한다.
- 인터페이스 : 모든 컴포넌트의 서비스를 쉽게 이해하고 사용할수 있도록 단순, 완전, 일관적인 interface를 개발하여야 한다.
- 성능 : 초기 단계에서 성능을 너무 강조해서는 안 된다.
- 단순성 : 모든 개발자가 이해 가능하도록 산출물은 단순하여야 한다.
- 문서화 : 개발 프로세스에서 모든 사항에 대하여 주석을 사용한다.

설계 모델



1.3.4 구현

구현 단계에서는 실질적 코딩이 이루어진다. 컴포넌트 개발, 컴파일, 링크, 디버깅을 반복 수행한다. 새로운 다이어그램 설계가 아닌 작성한 다이어그램을 수정하거나 상세화 한다.

1.3.5 테스트

테스트 코드의 에러를 찾는 것이며 다수의 테스트 케이스로 구성된다. 목적에 따라 단위 테스팅, 통합 테스팅, 시스템 테스팅, 인스 테스팅 등으로 구분된다.

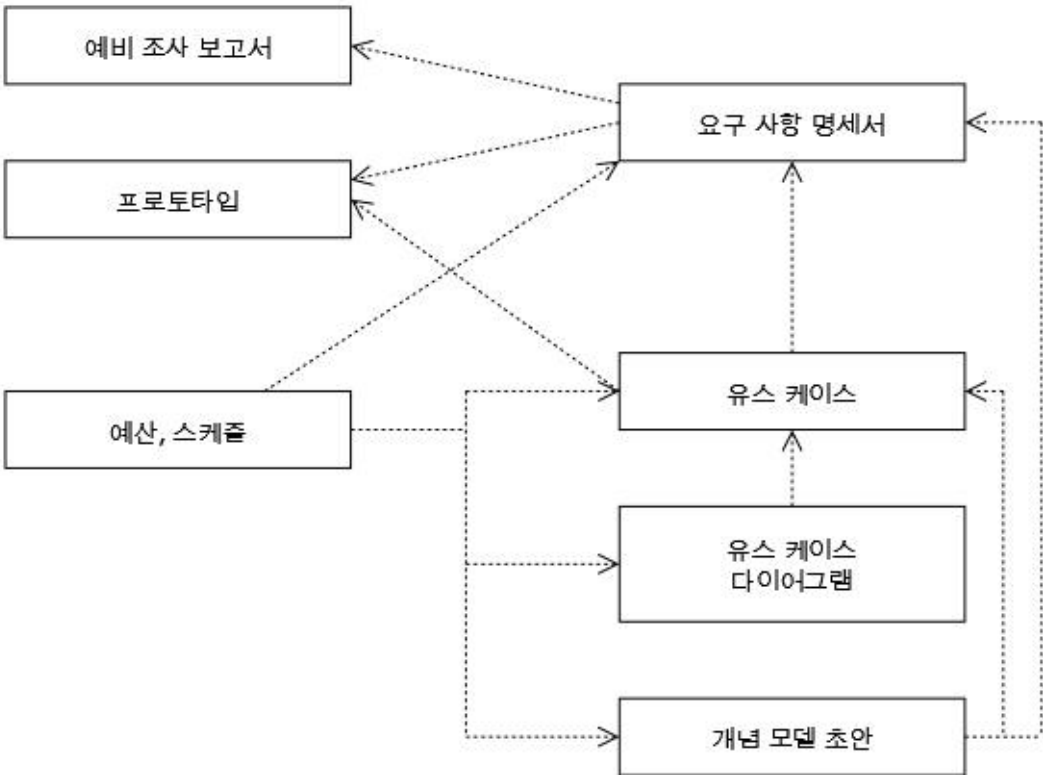
요구사항 분석 및 분석 단계에서 작성한 다이어그램 및 유스 케이스를 기초로 테스팅 한다.

2. 요구사항 분석

시스템은 복잡도를 관리하고 이해하기 위해서 작은 단위인 모델로 분리하여야 하며 이는 시스템의 핵심적인 사항을 정의한다. 소프트웨어 시스템을 개발하는 단계는 우선 문제 도메인의 중요한 사항을 구성하는 모델을 개발하는 것이다.

모델링 과정에서 작성하는 산출물은 상호간에 종속 관계를 가지며 이를 이해하면 모델의 일관성과 추적성을 보장할 수 있다.

분석 단계의 산출물과 관계



2.1 요구사항

요구사항 분석 단계의 기본적인 목적은 필요한 사항의 식별이며 사용자와 개발자가 의사소통을 할 수 있는 형태로 문서화 하는 것이다.

시스템 기능은 시스템이 수행하여야 하는 일을 정의하며 논리적인 그룹으로 식별하여 분류하여야 한다. 반면 시스템 애튜리뷰트는 비 기능적인 시스템의 품질을 의미한다. 이는 시스템의 특성이나 차원으로서 시스템의 기능이 아니며 기능 명세서의 일부분이 되어서는 안 된다. 따라서 별도의 시스템 애튜리뷰트 명세서에 포함시켜야 한다.

시스템 기능은 중요성에 따라 우선순위를 부여할 수 있도록 분류하여야 한다.

기능의 범주	의미
핵심	반드시 수행하여야 하는 기능이며, 이러한 기능이 수행된다는 것을 사용자가 인식하여야 한다.
주요	반드시 수행하여야 하는 기능이지만, 사용자는 인식하지 못한다.
선택	선택적인 기능으로서 이러한 기능의 추가가 다른 기능에 중대한 영향을 미치지 않는다.

2.2 유스 케이스

유스 케이스는 액터 혹은 조직체에서 가치 있는 결과를 제공하기 위한 일련의 이벤트, 활동 및 트랜잭션을 처음부터 끝까지 설명한다. 이는 도메인 프로세스를 완료하기 위하여 시스템의 액터가 수행하는 순차적인 행위를 설명한 서술식 문서이다. 유스 케이스는 시스템을 사용하는 스토리이며, 시스템을 사용하는 케이스로 정형화된 양식을 지정하지 않는다.

고수준 유스 케이스는 적은 수의 문장을 이용하여 프로세스를 매우 간략하게 설명한다. 고수준 유스 케이스는 시스템의 복잡도와 기능을 쉽게 이해하기 위하여 초기의 요구사항 분석 과정이나 프로젝트의 범위를 설정할 때 작성한다.

Expanded 유스 케이스는 고수준 유스 케이스보다 세보적인 사항을 보여주며 프로세스와 요구사항을 보다 깊게 이해하는데 유용하다. Expanded 유스 케이스는 액터와 시스템간의 대화 형식으로 작성된다.

유스 케이스	유스 케이스의 이름을 기록한다.
액터	액터를 열거한다. 유스 케이스를 시작하는 액터를 표시한다.
목적	유스 케이스의 목적을 기록한다.
개요	고수준 유스 케이스 혹은 이와 유사한 요약을 기록한다.
유형	기존, 보조 혹은 선택의 유형이나 핵심 혹은 실제의 유형을 기록한다.
참조	관련된 유스 케이스와 시스템 기능을 기록한다.
이벤트 순서	액터와 시스템간의 상호작용을 대화식으로 기록한다.
대체 이벤트	예외적인 사항이 발생하는 경우의 조치를 기록한다.

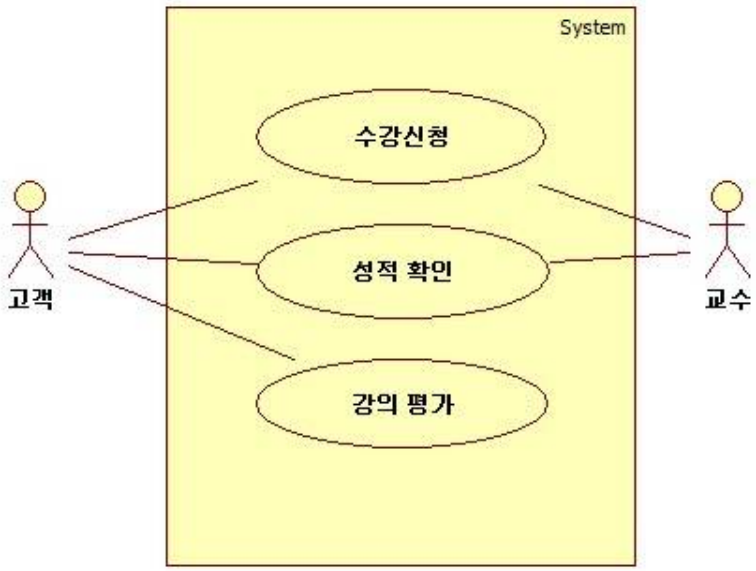
요구사항을 정의하는 과정에서 식별한 모든 시스템 기능은 유스 케이스에 모두 할당하여야 하며 궁극적으로 모든 시스템 기능과 유스 케이스는 구현과 테스팅을 통하여 추적할 수 있어야 한다.

유스 케이스를 식별에는 방법에는 두 가지가 있다. 첫 번째는 액터에 근거를 두고 유스 케이스를 식별한다. 우선 시스템에 관련된 액터를 식별하고 액터가 개시하거나 참여하는 프로세스를 식별한다. 두 번째는 이벤트에 근거를 두고 유스 케이스를 식별한다. 즉 외부 이벤트를 식별하고, 각각의 이벤트를 액터와 시스템에 연관시킨다.

유스 케이스 식별 과정에서 빈번히 발생하는 에러는 개별적인 업무절차, 오퍼레이션

혹은 트랜잭션을 표현하는 것이다. 유스 케이스는 프로세서의 개별적인 절차나 활동이 아니다.

유스 케이스 다이어그램은 일련의 유스 케이스, 액터, 유스 케이스와 액터간의 관계를 표현하며 시스템의 외부 액터를 빨리 식별하고, 이들이 시스템을 사용하는 중요한 방법을 빠르게 이해하기 위한 일종의 Context diagram이다.



2.3 유스 케이스의 유형

유스 케이스는 중요도에 따라 Primary, Secondary, Optional 유스 케이스로 분류한다. 이러한 분류에 따라 개발을 위한 유스 케이스의 우선순위를 부여하게 된다.

- Primary 유스 케이스 : 중요한 공통적인 프로세스를 표현
- Secondary 유스 케이스 : 중요도가 다소 떨어지는 프로세스를 표현
- Optional 유스 케이스 : 구현하지 않을 수도 있는 프로세스를 표현

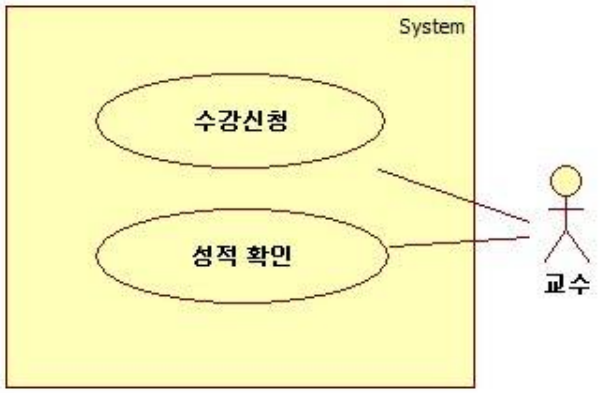
또한 유스 케이스는 기술적인 설계 내역의 포함 여부에 따라 핵심 유스케이스, 실제 유스 케이스로 분류한다.

- 핵심 유스 케이스 : 기술 및 구현 세부사항으로부터 자유로운 형태로 표현하며 초기 요구사항 분석 과정에서 작성
- 실제 유스 케이스 : 현재의 실질적인 설계, 특정한 입출력 기능등 을 고려하여 프로세스 표현하며 개발 주기의 설계 단계에서 작성

2.4 시스템 경계

유스 케이스는 시스템과의 상호 작용을 설명한다. 시스템 경계는 컴퓨터 시스템 혹은 장치의 하드웨어/소프트웨어 경계, 조직체의 부서, 전체 조직체를 포함한다.

시스템의 경계를 정의하는 것은 시스템의 내부 요소와 외부 요소를 식별하고 시스템의 책임을 정의하는 것이기 때문에 매우 중요하다. 시스템의 외부 환경은 액터에 의해서만 표현되지만 소프트웨어 응용이나 장치를 개발하는 경우라면, 하드웨어와 소프트웨어 경계를 시스템 경계로 설정하는 것이 의미가 있다. 그러나 비즈니스 프로세스 리엔지니어링을 수행하는 경우라면, 전체 비즈니스 혹은 전체 조직체를 시스템으로 간주하는 것이 보다 타당하다.



2.5 개발 프로세스에서의 유스 케이스

유스 케이스는 개발 프로세스의 단계에 따라 작성하여야 하는 유형을 고려하여야 한다. 이는 문제의 핵심을 정확히 이해하기 위한 노력이다.

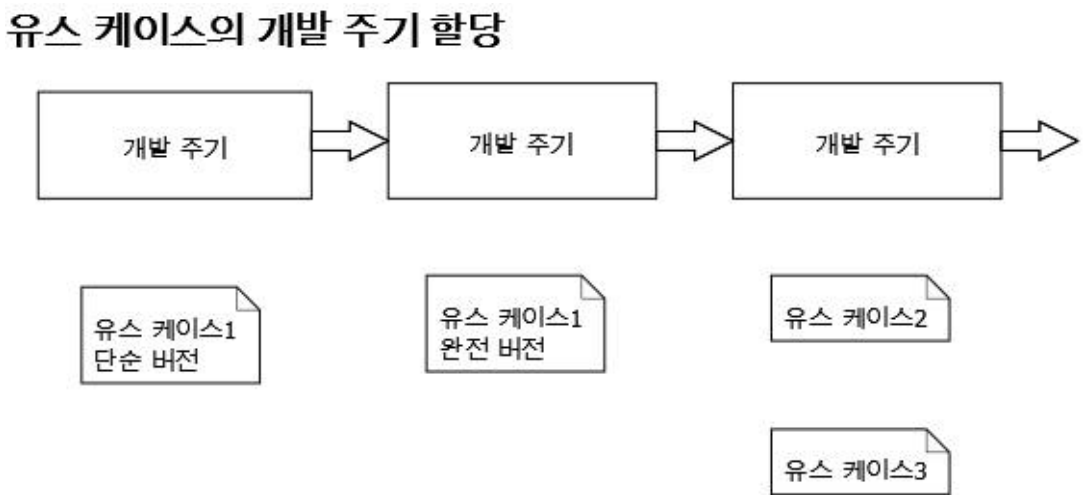
계획 단계에서 유스 케이스 작성 고려사항
① 시스템 기능을 식별한 이후, 시스템의 범위를 정의하고 액터와 유스 케이스를 식별한다.
② 모든 유스 케이스를 고수준 형태로 작성하고 Primary, Secondary, Optional로 분류한다.
③ 유스 케이스 Diagram을 작성한다.
④ 유스 케이스간의 관계성을 식별하고, 이들을 유스 케이스 Digaram으로 표현한다.
⑤ 보다 충실한 이해를 위하여 가장 중요하며 위험요소가 있는 유스 케이스를 expanded essential 유스 케이스 형태로 작성하고, 문제의 특성과 규모 추정
⑥ 실제 유스 케이스는 설계에 관련된 결정사항을 포함하기 때문에 개발 주기의 설계단계까지 지연시켜야 하지만 실제 유스 케이스의 작성이 문제를 이해하는데 도움이 되거나 고객이 원할 경우 초기의 요구사항 단계에서도 작성할 수 있다.
⑦ 유스 케이스의 우선순위를 결정한다.

반복적 개발 단계에서 유스 케이스 작성 고려사항
① 분석 단계 : 현재 다루고 있는 유스 케이스에 대하여 expanded essential 유스 케이스를 작성한다.
② 설계 단계 : 현재 다루고 있는 유스 케이스에 대하여 실제 유스 케이스를 작성한다.

2.6 유스 케이스의 우선순위

반복적인 개발 단계의 각 개발주기는 유스 케이스 요구사항에 따라 구성된다. 즉, 각

각의 개발 주기는 하나 혹은 그 이상의 유스 케이스를 구현한다.

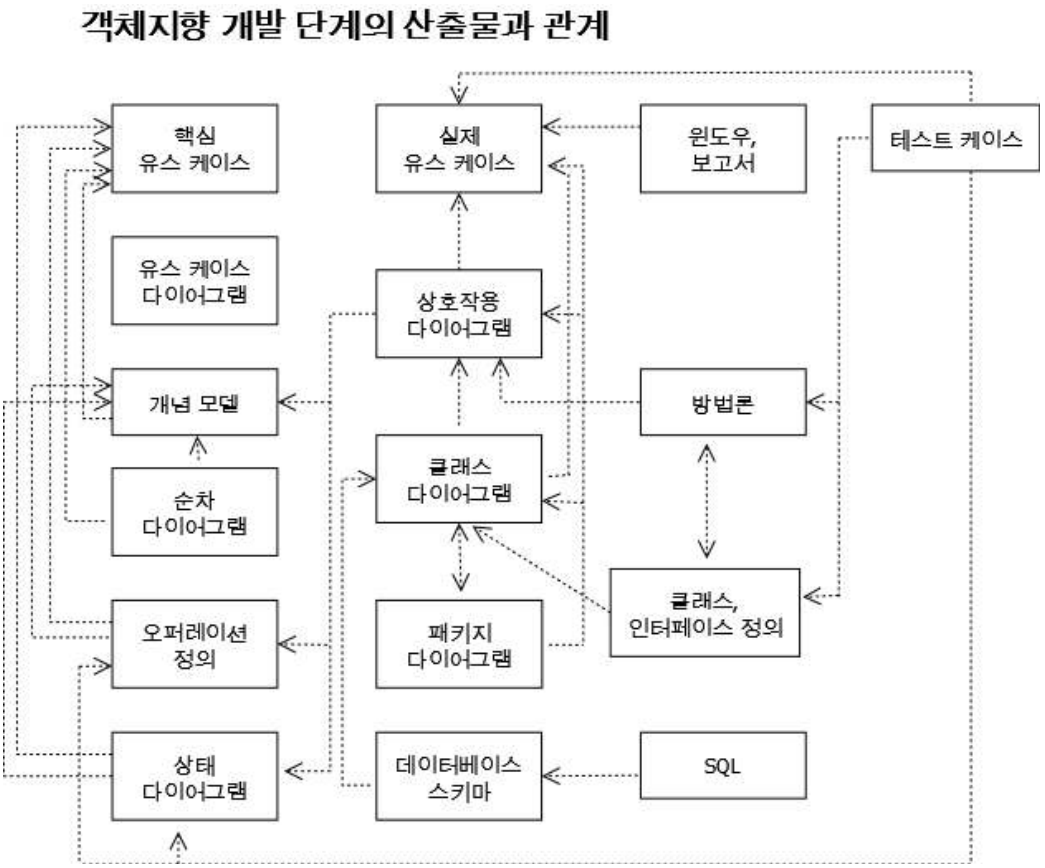


우선순위가 높은 유스 케이스를 개발 주기의 초기에 다루기 위하여 유스 케이스에 대한 우선순위를 부여하는 것이 좋다.

유스 케이스 우선순위 할당
• 아키텍처 설계에 중대한 영향을 미친다.
• 적은 노력으로 설계에 관련된 중요한 정보나 개념을 확보할 수 있다.
• 위험이 높거나, 시간에 민감하고, 혹은 복잡한 기능을 포함한다.
• 많은 연구를 필요로 하거나, 새로이 개발된 기술을 활용한다.
• 중요한 비즈니스 프로세스를 표현한다.
• 개발 효과를 높이거나 비용을 감소시키는 직접적인 개선을 제공한다.

3. 개념 모델 구축

개념 모델(conceptual model)은 문제 도메인의 의미있는 개념들을 설명한다. 개념은 객체, 대상 혹은 아이디어이다. UML은 개념 모델을 설명하기 위하여 Static structure diagram을 사용한다. 개념 모델은 소프트웨어 컴포넌트가 아닌 현실 세계의 대상을 표현한다.



3.1 개념 모델

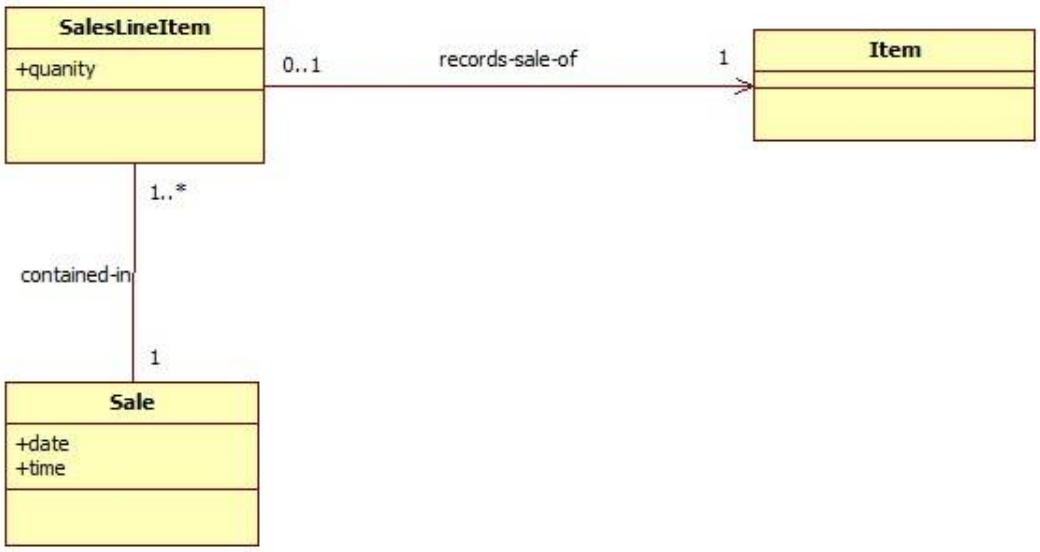
분석 과정에서 가장 객체지향적인 요소는 문제 도메인을 개별적인 개념 혹은 객체로 분할하는 것이다. 개념 모델은 문제 도메인에 존재하는 개념을 표현한다. UML에서 개념 모델은 오퍼레이션을 정의 하지 않는 Static structure diagram으로 나타낸다. 개념모델은 개념, 개념간의 연관 관계, 개념의 애튜리뷰트를 제시한다.

개념모델은 소프트웨어 설계가 아닌 실세계의 문제 도메인에 존재하는 개념을 설명한다. 따라서 데이터베이스 등과 같은 산출물이나 메소드 등과 같은 설계 내역은 개념 모델에 표현하지 않는다.

분해는 문제 도메인을 관리할 수 있는 단위로 분할하여 복잡도를 관리하는 방법이며 객체지향 분석에서는 개념 혹은 객체가 분해의 초점이 된다. 문제 도메인에서 세부적인 개념을 식별하여 개념 모델로서 문서화 하여야 한다.

개념 모델 작성을 위한 절차
① 개념 식별 리스트(Concept Category List) 와 명사구 식별 방법을 사용하여 현재 고려중인 요구 사항에 관련된 후보 개념들을 식별한다.
② 개념 모델에 개념들을 표현한다.
③ 표현하여야 할 필요가 있는 정보를 기록하기 위하여 개념간의 연관 관계를 추가한다.
④ 표현하여야 할 필요가 있는 정보를 기록하기 위하여 각각의 개념에 애튜리뷰트를 추가한다.

부분적 개념 모델



3.2 개념 식별

새로이 식별되는 개념들은 모두 개념 모델에 추가하여야 한다. 애튜리뷰트가 없는 개념도 유효하며, 정보의 역할이 아닌 도메인에서의 행위 역할만을 갖는 개념도 존재한다. 개념 식별에는 두 가지 방법이 있다. 첫 번째는 개념 식별 리스트를 사용한 식별이다.

개념 범주	
• 물리적 객체 혹은 실존 객체	• 추상적 명사 개념
• 명세서, 설계서	• 조직체
• 장소	• 이벤트
• 트랜젝션	• 프로세스
• 트랜젝션 구성 항목	• 규칙 및 정책
• 사람의 역할	• 카탈로그
• 사물에 대한 컨테이너	• 재무 등의 기록
• 컨테이너 내부의 사물	• 매뉴얼, 도서
• 시스템 외부의 기타 시스템	

두 번째는 문제 도메인을 설명하는 텍스트 문구에서 명사와 명사구를 식별하여 개념을 식별하고, 이들을 개념 혹은 애튜리뷰트의 후보로 간주하는 것이다. 명사나 명사구를 검토할 Expended 유스 케이스를 사용하는 것이 좋다.

개념을 개념이 아닌 애튜리뷰트로 표현하는 것은 개념 모델을 작성할 때 가장 일반적으로 발생하는 실수이다. 실세계에서 숫자나 텍스트로 생각되지 않는 요소들은 애튜리뷰트가 아닌 개념으로 정의하여야 한다.

3.3 개념과 연관 관계

유스 케이스의 정보 요구사항을 충족시키기 위하여 개념간의 연관 관계를 식별하여야 한다. 연관 관계는 개념 모델을 이해하는데 도움이 된다.

연관 관계 포함 특성
• 개념간의 관계에 대한 지식을 일정 기간 동안 보존할 필요가 있는 연관 관계
• 연관 식별 리스트(Associations Category List)로부터 유도된 연관관계

UML 다이어그램에서는 연관 관계의 읽는 방향을 표시하기 위해 작은 삼각형을 사용한다.

3.4 연관 식별

개념간의 연관 관계는 연관 식별 리스트를 사용하여 식별할 수 있다.

연관 범주	
• A는 B의 물리적인 부분이다.	• A는 B의 부서이다.
• A는 B의 논리적인 부분이다.	• A는 B를 사용하거나 관리한다.
• A는 B에 물리적으로 포함된다.	• A는 B와 통신한다.
• A는 B에 논리적으로 포함된다.	• A는 트랜잭션 B와 관련된다.
• A는 B에 대한 설명이다.	• A는 트랜잭션 B와 관련된 트랜잭션이다.
• A는 트랜잭션 혹은 보고서 B의 세부 항목이다.	• A는 B 다음이다.
• A는 B에 기록/보고/저장된다.	• A는 B에 의해 소유된다.
• A는 B의 멤버이다.	

다음 범주에 속하는 연관관계들은 shb은 우선순위를 부여받는다.

- A는 B의 논리적인 부분이다
- A는 B에 논리적으로 포함된다
- A는 B에 기록된다

연관 관계 식별 주의사항
• 연관 관계에 대한 지식을 일정 기간 동안 보존할 필요가 있는 연관 관계에 초점을 맞춘다.
• 연관 관계를 식별하는 것보다 개념을 식별하는 것이 보다 중요하다.
• 많은 수의 연관 관계는 개념 모델을 설명하기보다는 혼란을 초래하는 경향이 있다. 많은 연관 관계를 식별하기 위한 노력은 그 효과에 비해 시간을 낭비하는 요인이 된다.
• 중복적인 연관 관계나 다른 연관 관계로부터 유도할 수 있는 연관 관계는 생략하여야 한다.

3.5 개념과 애튜리뷰트

유스 케이스의 정보 요구사항을 충족시키기 위하여 개념과 애튜리뷰트를 식별하여야 한다. 대부분의 애튜리뷰트는 Primitive Data type으로 인식된다.

개념모델의 애튜리뷰트는 부울, 날짜, 숫자, 텍스트 및 시간 등과 같은 Simple 애튜리뷰트 와 각자의 고유한 주체성을 구분하는 것이 의미가 없는 값, 즉 value object인 Pure data 이어야 한다.

애튜리뷰트는 요구사항 명세서, 현재 고려중인 유스 케이스, 기타 관련 문서들을 참조함으로써 쉽게 식별할 수 있다.

3.6 용어사전(Glossary)

Glossary 혹은 모델 사전은 문제 도메인에 존재하는 용어를 정의하는 간단한 문서이다. 이는 의사소통을 향상시키고, 용어의 오 해에 따른 위험을 감소시키기 위하여 사용한다. 계획 단계에 작성되며 추후 계속 보강하여야 한다. 도메인 규칙이나 비즈니스 규칙, 제약조건 등을 기록할 수도 있다. 용어사전을 위한 공식적인 양식이 정의되어 있지는 않다.

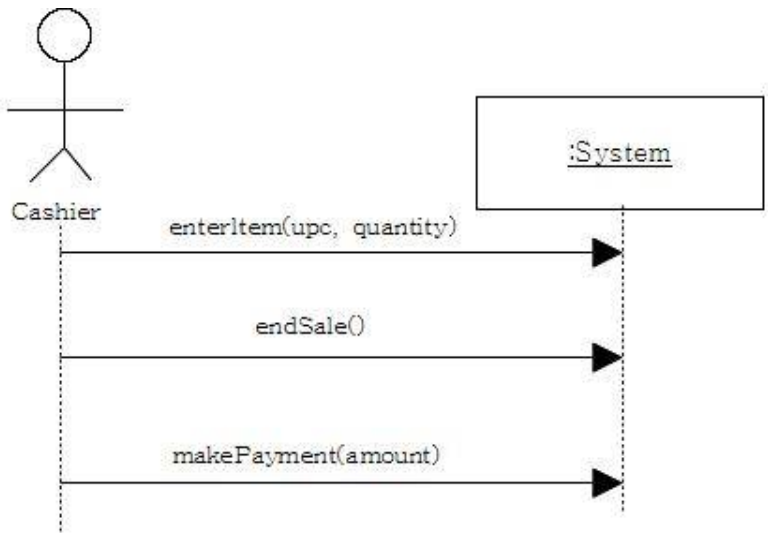
4. 시스템 행위

소프트웨어 시스템에 대한 논리적인 설계 이전에 시스템 행위를 조사하고 정의하여야 한다. 이는 무엇을 하여야 하는가에 대한 설명이며 수행 방법에 대해서는 포함시키지 않는다. System sequence diagram으로 표현할 수 있으며 액터로부터 시스템에게 전달되는 이벤트를 보여준다. 시스템이 무엇을 하여야 하는가를 조사하는 것이므로 분석 단계에서 작성한다.

4.1 시스템 순차 다이어그램

유스 케이스는 액터와 소프트웨어 시스템간의 상호작용을 표현한다. 액터는 시스템에 대한 이벤트를 발생시키고 그 응답으로 오퍼레이션을 일으킨다. 액터에 의한 시스템 오퍼레이션은 분리하여 표현하는 것이 바람직하다

순차 다이어그램



System sequence diagram은 Use case의 특정한 시나리오에 대하여 액터가 발생하는 이벤트 및 이벤트 순서를 보여준다. System sequence diagram는 액터와 시스템간의 시스템 경계를 지나는 이벤트에 초점을 맞추며 유스 케이스의 이벤트 순서를 반영하여야 한다.

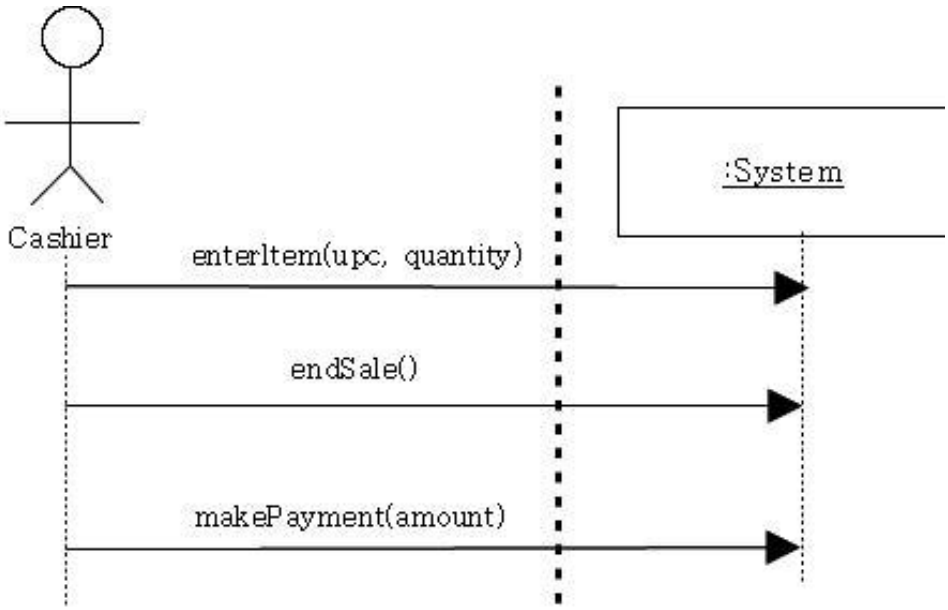
System sequence diagram 작성 절차
① 시스템을 블랙 박스로 표현하는 선을 그린다.
② 시스템에 직접적으로 작용하는 각각의 액터를 식별한다. 각각의 액터에 대해서도 선을 그린다.
③ 유스 케이스의 이벤트 순서 내용을 참조하여 각각의 액터가 발생시키는 시스템 이벤트를 식별한다. 이러한 이벤트를 다이어그램에 표시한다.
④ 유스 케이스의 내용을 다이어그램의 왼쪽에 기록할 수 있다.

4.2 시스템 이벤트 및 시스템 오퍼레이션

시스템 이벤트는 액터가 발생하는 외부적인 입력 이벤트이며 이에 대응되는 오퍼레이션을 실행시킨다. 이벤트와 오퍼레이션의 이름은 동일하다.

시스템 이벤트 식별을 위해 시스템 경계를 명확히 설정하는 것이 필요하다. 소프트웨어 개발의 경우 일반적으로 시스템 경계는 소프트웨어 자체가 된다.

시스템 경계와 시스템 이벤트



4.3 시스템 오퍼레이션 정의

시스템 Operaion 정의는 오퍼레이션을 통하여 시스템의 행위를 정의하는 것이다. 이는 시스템 오퍼레이션이 시스템에 미치는 영향을 설명한다. UML은 오퍼레이션에 대한 사전조건(pre-condition)과 사후조건(post-condition)을 정의함으로써 시스템 오퍼레이션을 정의한다.

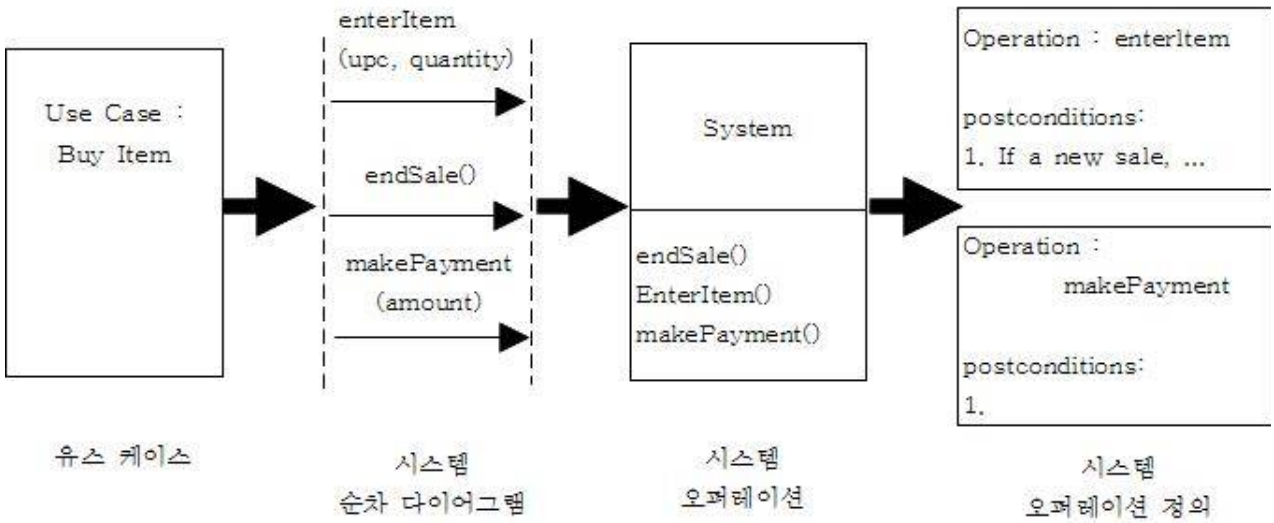
모든 항목을 기록할 필요는 없지만 오퍼레이션의 책임과 사후조건은 반드시 기록하는 것이 좋다.

시스템 오퍼레이션 정의 양식	
항목	**의미**
이름	• 시스템 오퍼레이션의 이름과 매개변수를 기록한다.
책임	• 시스템 오퍼레이션이 반드시 충족시켜야 하는 책임을 간략히 기록한다.
유형	• 유형의 이름을 기록한다. (개념, 소프트웨어 클래스, 인터페이스)
참조	• 관련 유스 케이스와 시스템 기능을 기록한다.
주석	• 설계 주의사항, 알고리즘 등을 기록한다.
예외조건	• 예외적인 사항을 기록한다.
출력	• 시스템 외부로 보내는 메시지나 레코드 등과 같은 출력을 기록한다.
사전조건	• 시스템 오퍼레이션이 실행되기 이전에 시스템의 상태에 대한 가정사항을 기록한다.
사후조건	• 시스템 오퍼레이션이 종료된 이후에 시스템의 상태를 기록한다.

시스템 오퍼레이션 정의 절차
① 시스템 순차 다이어그램으로부터 시스템 오퍼레이션을 식별한다.
② 각각의 시스템 오퍼레이션에 대하여, 시스템 오퍼레이션 정의를 기록한다.
③ 시스템 오퍼레이션의 목적을 간략히 기록하는 책임 항목을 기록하는 것으로 시작한다.
④ 그후 개념 모델의 객체에 발생한 상태 변화를 기록하기 위하여 사후조건 항목을 기록한다.
⑤ 사후조건을 기록할 때 다음과 같은 사항을 고려한다. - 객체 인스턴스의 생성과 소멸 - 애트리뷰트 변경 - 연관 관계의 형성과 소멸

유스 케이스는 시스템 이벤트와 시스템 순차 다이어그램을 식별할 수 있도록 하며, 이를 통하여 시스템 오퍼레이션을 식별할 수 있다.

시스템 오퍼레이션 정의와 기타 산출물간의 관계



4.4 사후조건 및 사전조건

시스템 오퍼레이션 정의에서 가장 중요한 것은 책임이며 사후조건이다. 사후조건은 오퍼레이션의 실행 결과로 시스템이 어떻게 변화했는지를 설명하므로 오퍼레이션이 종료되었을 때의 시스템 상태를 표명한다. 이를 표현하기 위한 양식의 제한은 없으나 능동태보다는 수동태로 표현하는 것이 바람직하다.

사후조건은 개념 모델의 범위 내에서 표현된다. 개념 모델에 존재하는 새로운 인스턴스가 생성되고, 개념 모델에 존재하는 연관 관계 중에서 새로운 연관 관계가 생성된다.

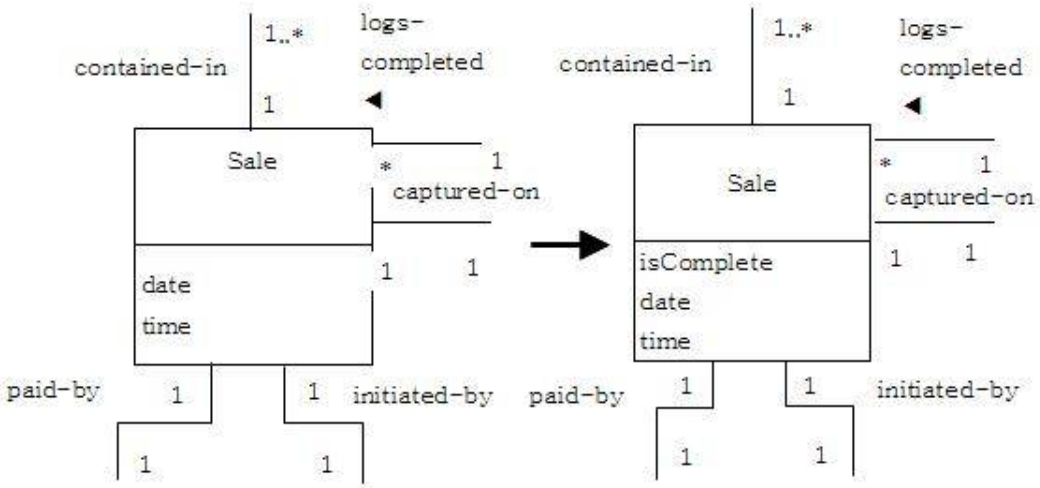
시스템 설계 단계에서 시스템 오퍼레이션에 대한 추가적인 정의들을 식별할 수 있으며 이후 반복적인 개발 주기의 계획 및 조사 단계에 입력되어 질수 있다.

사후조건은 오퍼레이션 시작 시점에서의 시스템의 상태를 정의한다.

4.5 개념 모델의 변경

개념 모델에 표현되어 있는 않은 추가적인 정보가 있을 수 있으며 이 경우 개념 모델의 애트리뷰트 혹은 오퍼레이션을 변경하여 모델에 반영하면 된다.

개념 모델의 변경



5. 시스템 설계와 상호작용

시스템 분석 단계는 시스템에 관련된 요구사항, 개념 및 오퍼레이션을 이해하는 단계이다. 분석 단계에서의 유스 케이스, 개념 모델, 시스템 순차 다이어그램, 시스템 오퍼레이션 정의와 같은 산출물이 작성되면 설계 단계로 진행할 수 있다. 시스템 설계과정에서는 객체지향 패러다임에 근거한 논리적인 해결책이 제시되며 객체들이 요구사항을 충족시키기 위하여 어떻게 상호작용 하는지를 보여주는 상호작용 다이어그램을 작성하는 것이다.

상호작용 다이어그램을 작성하면 소프트웨어로 구현하는 클래스와 인터페이스를 정의하기 위하여 설계 클래스 다이어그램(Design Class diagram)을 작성할 수 있다.

우수한 상호작용 다이어그램을 작성하기 위해서는 객체에 대한 책임할당(responsibility assign-ment) 원리를 이해하고, 설계 패턴(design pattern)을 사용하는 방법을 이해하는 것이 필요하다.

5.1 실제 유스 케이스

실제 유스 케이스는 유스 케이스를 어떻게 구현할 것인지에 대한 실질적인 설계 사항을 보여주는 것으로 입출력 기술과 유스 케이스의 전반적인 구현사항을 제시한다.

실제 유스 케이스를 작성하는 것은 설계 단계의 첫 번째 작업이지만 항상 작성하여야 하는 것은 아니다.

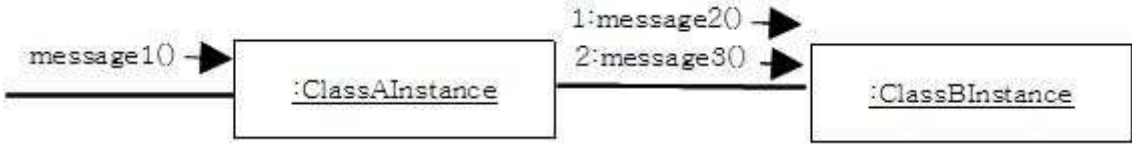
5.2 상호작용 다이어그램

상호작용 다이어그램은 소프트웨어 객체가 주어진 태스크를 수행하기 위하여 메시지를 통해 어떻게 상요작용 하는지를 보여주며 설계 단계에서 작성한다.

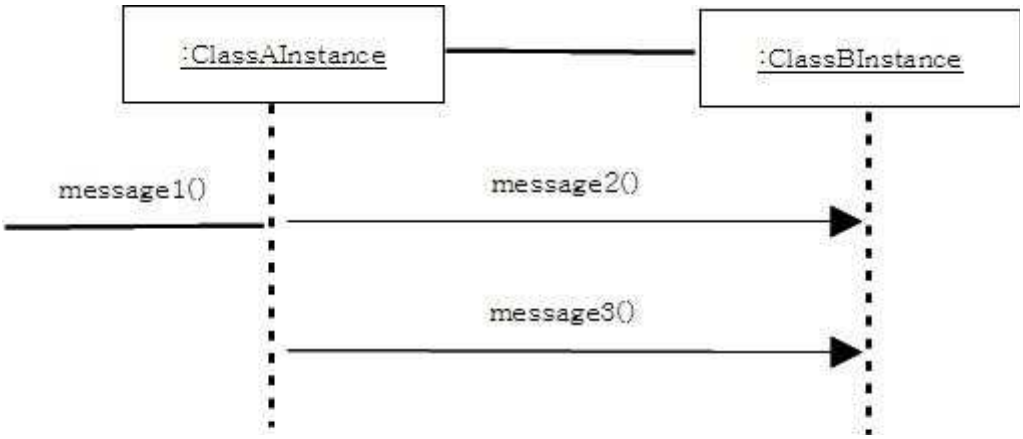
상호작용 다이어그램이 의존하는 산출물
• 개념 모델 : 개념 모델의 개념에 대응되는 소프트웨어 클래스를 정의한다.
• 시스템 오퍼레이션 : 상호작용 다이어그램이 충족시켜야 하는 책임과 사후조건을 식별한다.
• 실제 혹은 핵심 유스 케이스 : 시스템 오퍼레이션의 정의 이외에 상호작용 다이어그램이 충족시켜야 하는 태스크에 대한 정보를 수집한다.

상호작용 다이어그램은 클래스 모델에 존재하는 인스턴스간의 메시지 상호작용을 표시한다. 이러한 상호작용의 시작점은 시스템 오퍼레이션 정의에서 식별된 사후조건을 충족시키는 것이다. UML은 협동 다이어그램과 순차 다이어그램 두 가지 유형의 상호작용 다이어그램을 정의한다.

협동 다이어그램



순차 다이어그램



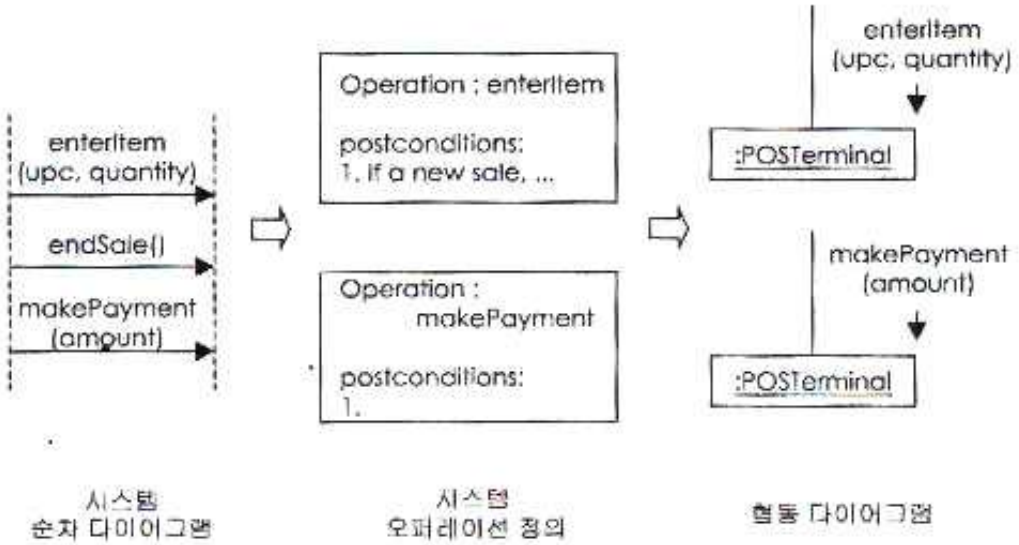
상호작용 다이어그램은 객체지향 분석·설계 과정에서 작성하는 가장 중요한 산출물 중의 하나이다.

5.3 협동 다이어그램의 작성

협동 다이어그램 작성 절차
① 현재의 개발 주기에서 개발 중인 각각의 시스템 오퍼레이션에 대하여 별도의 다이어그램을 작성한다.
② 각각의 시스템 오퍼레이션 메시지를 시작 메시지로 하여 다이어그램을 작성한다.
③ 다이어그램이 복잡해지면 보다 작은 다이어그램으로 분할한다.
④ 시스템 오퍼레이션 정의의 책임과 사후조건, 유스 케이스의 설명을 시작점으로 하여 태스크를 충족시키기 위한 객체의 상호작용을 설계한다. 우수한 설계를 위하여 여러 가지 설계 패넡과 설계 원리를 적용한다.

협동 다이어그램과 산출물간의 관계
• 유스 케이스는 시스템 순차 다이어그램에 명시적으로 표현되는 시스템 이벤트를 제공한다.
• 시스템 오퍼레이션의 초기 영향은 시스템 오퍼레이션 정의에서 제공된다.
• 시스템 오퍼레이션은 상호작용 다이어그램을 시작하는 메시지를 나타낸다.

협동 다이어그램과 산출물간의 관계



6. 책임 할당을 위한 기본 설계 패턴

시스템 분석 단계는 시스템에 관련된 요구사항, 개념 및 오퍼레이션을 이해하는 단계이다. 분석 단계에서의 유스 케이스, 개념 모델, 시스템 순차 다이어그램, 시스템 오퍼레이션 정의와 같은 산출물이 작성되면 설계 단계로 진행할 수 있다. 시스템 설계과정에서는 객체지향 패러다임에 근거한 논리적인 해결책이 제시되며 객체들이 요구사항을 충족시키기 위하여 어떻게 상호작용 하는지를 보여주는 상호작용 다이어그램을 작성하는 것이다.

상호작용 다이어그램을 작성하면 소프트웨어로 구현하는 클래스와 인터페이스를 정의하기 위하여 설계 클래스 다이어그램(Design Class diagram)을 작성할 수 있다.

우수한 상호작용 다이어그램을 작성하기 위해서는 객체에 대한 책임할당(responsibility assign-ment) 원리를 이해하고, 설계 패턴(design pattern)을 사용하는 방법을 이해하는 것이 필요하다.

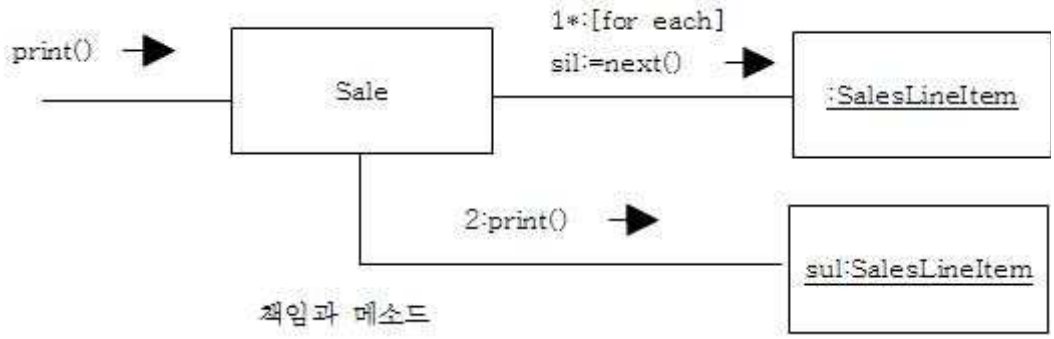
6.1 책임과 메소드

책임은 객체의 의무와 관련되며 수행(doing)책임과 지식(knowing) 책임으로 구분된다.

객체의 수행 책임에는 객체 스스로 무엇을 수행하는 것, 다른 객체의 행동을 시작시키는 것, 다른 객체의 행동을 제어하고 조정하는 것 등이 포함된다.

책임은 객체지향 설계 동안에 객체에 할당된다. 클래스 혹은 메소드로의 책임 변환은 책임의 크기에 의하여 영향을 받는다. 책임은 혼자서 실행되거나 혹은 다른 메소드 및 객체와 협동하는 메소드를 사용하여 구현된다.

책임과 메소드



상호작용 다이어그램은 객체에 대한 책임 할당을 보여준다. 책임 할당은 어느 메시지가 어느 객체로 보내지는가에 의하여 반영되며 책임 할당을 위한 설계 패턴은 책임을 어디에 할당할 것인지에 대한 지침을 제공해준다.

6.2 패턴

패턴은 새로운 환경에 적용할 수 있도록 과거의 경험으로부터 얻은 문제/해결책 쌍

(Pair)을 정리하고, 이에 이름을 부여한 것이다. 이러한 패턴은 문제의 범주에서 객체에 책임을 어떻게 할당할 것인지에 대한 지침을 제공한다. 패턴은 기존의 지식이나 원리를 문서화 한 것이다.

6.3 GRASP 패턴

GRASP 패턴은 객체에 책임을 할당하기 위한 기본적인 원리를 설명한다. GRASP 패턴은 크게 5가지 기본원리로 구성되어 있다.

6.3.1 Information Expert

역할을 수행할 수 있는 정보를 가지고 있는 객체에 역할을 부여한다. 객체지향의 기본 원리이며 객체간의 상호작용을 정의할 때, 클래스에 책임을 할당하기 위한 결정을 하여야 한다.

패턴 이름	Expert
문제	• 객체지향 설계에서 책임을 할당할 때 고려하여야 하는 가장 기본적인 원리는 무엇인가?
해결책	• 정보 전문가 즉, 책임을 충족시키는데 필요한 정보를 갖고 있는 클래스에게 책임을 할당한다.

책임 할당은 상호작용 다이어그램을 작성하는 과정에서 빈번히 발생한다.

부분적인 협동 다이어그램

6.3.2 Creator

객체의 생성은 생성되는 객체의 컨텍스트를 알고 있는 다른 객체가 있다면, 컨텍스트를 알고 있는 객체에 부여한다. A 객체와 B 객체의 관계의 관계가 다음 중 하나라면 A의 생성을 B의 역할로 부여한다.

- 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.

6.3.3 Low Coupling

객체들간, 서브 시스템들간의 상호의존도가 낮게 역할을 부여한다. Object-Oriented 시스템은 각 객체들과 그들 간의 상호작용을 통하여 요구사항을 충족시키는 것을 기본으로 한다. 그러므로, 각 객체들 사이에 Coupling이 존재하지 않을 수는 없다. 이 패턴은 요구사항은 충족시키면서도 각 객체들, 각 서브시스템 간의 Coupling를 낮은 수준으로 유지하는 방향으로 디자인하라고 말하고 있다. Low Coupling은 각 객체, 서브시스템의 재 사용성을 높이고, 시스템 관리에 편하게 한다.

6.3.4 High Cohesion

각 객체가 밀접하게 연관된 역할들만 가지도록 역할을 부여한다. 이 패턴은 Low Coupling 패턴과 동전의 양면을 이루는 것으로, 한 객체, 한 서브시스템이 자기 자신이 부여받은 역할만을 수행하도록 짜임새 있게 구성되어 있다면, 자신이 부여 받은 역할을 충족시키기 위해 다른 객체나 시스템을 참조하는 일이 적을 것이고, 그것이 곧 Low Coupling이기 때문이다.

6.3.5 Controller

시스템 이벤트(사용자의 요청)를 처리할 객체를 만든다. 시스템, 서브시스템으로 들어오는 외부 요청을 처리하는 객체를 만들어 사용한다. 만약 어떤 서브시스템안에 있는 각 객체의 기능을 사용할 때, 직접적으로 각 객체에 접근하게 된다면 서브시스템과 외부간의 Coupling이 증가되고, 서브시스템의 어떤 객체를 수정할 경우, 외부에 주는 충격이 크게 된다. 서브시스템을 사용하는 입장에서 보면, 이 Controller 객체만 알고 있으면 되므로 사용하기 쉽다.

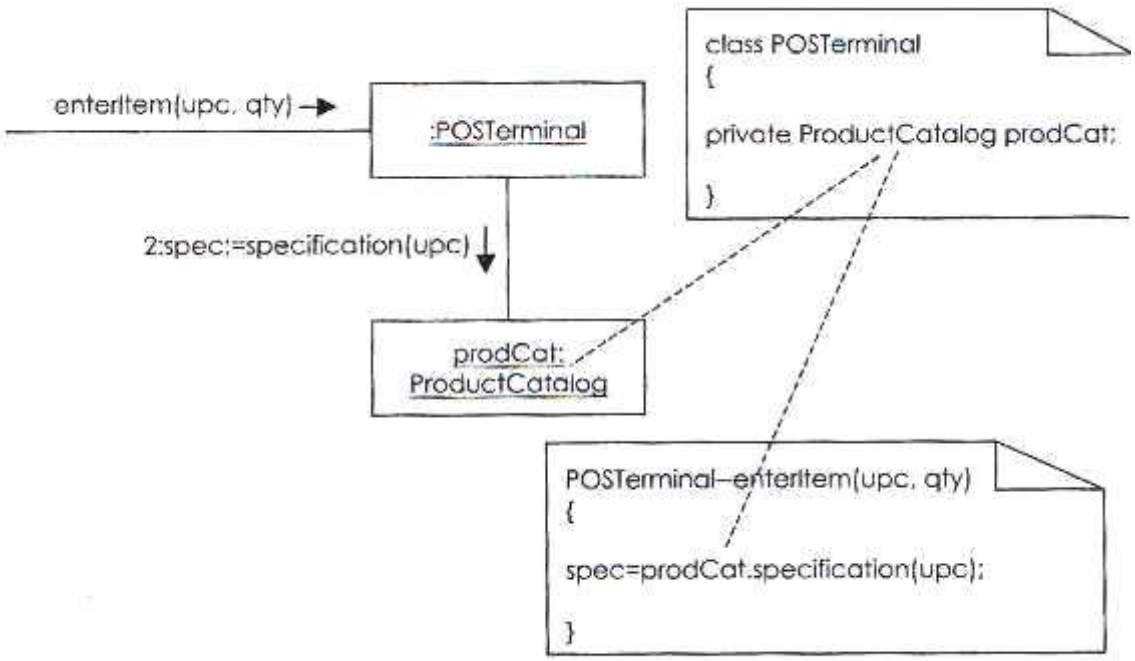
6.4 가시성

가시성은 다른 객체를 참조하거나 사용할 수 있는 객체의 능력이다. 시스템 이벤트를 위하여 작성된 협동 다이어그램은 객체간의 메시지를 보여준다. 송신 객체는 수신 객체에게 메시지를 보내기 위하여 반드시 수신 객체에게 가시적이어야 한다.

객체 A에서 객체 B로의 가시성의 네가지 형식
• 애트리뷰트 가시성 : 객체 B가 객체 A의 애트리뷰트이다.
• 매개변수 가시성 : 객체 B가 객체 A의 메소드에 대한 매개변수이다.
• 로컬 선언 가시성 : 객체 B가 객체 A의 메소드에서 로컬 객체로 선언된다.
• 글로벌 가시성 : 객체 B가 글로벌 가시성을 갖는다.

6.4.1 애트리뷰트 가시성

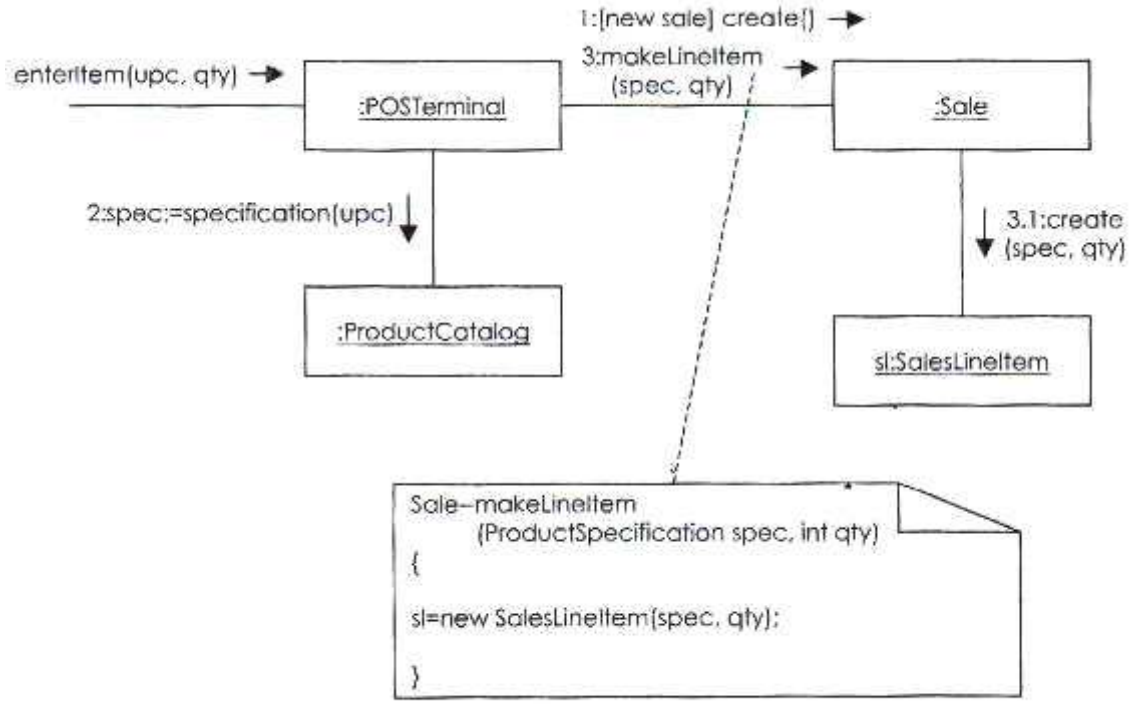
객체 A에서 객체 B로의 애트리뷰트 가시성은 객체 B가 객체 A의 애트리뷰트일 때 존재한다.



애트리뷰트 가시성은 객체 A가 객체 B가 존재하는 한 지속되기 때문에 상대적으로 영구적인 가시성이다.

6.4.2 매개변수 가시성

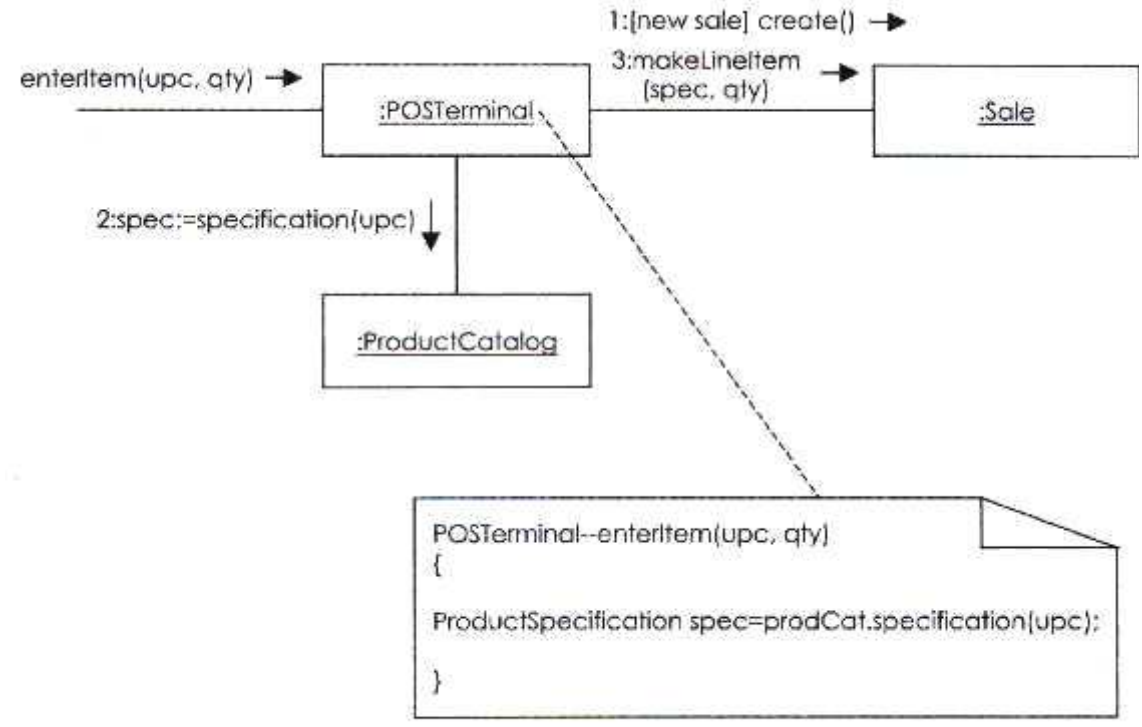
객체 A에서 객체 B로의 매개변수 가시성은 객체 B가 객체 A의 메소드에 대한 매개변수로서 전달될 때 존재한다. 매개변수 가시성은 메소드의 범위 내에서만 지속되기 때문에 상대적으로 임시적인 가시성이다. 매개변수 가시성은 애트리뷰트 가시성으로 변환할 수도 있다.



6.4.3 로컬 선언 가시성

객체 A에서 객체 B로의 로컬 선언 가시성은 객체 B가 객체 A의 메소드 내에서 로컬 객체로 선언될 때 존재한다. 로컬 선언 가시성은 메소드의 범위 내에서만 지속되기 때문에 상대적으로 임시적인 가시성이다.

로컬 선언 가시성은 새로운 로컬 인스턴스를 생성하고 로컬 변수를 할당하는 형태와 메소드 호출의 반환 객체(return object)를 로컬 변수에 할당하는 두가지 형태가 있다.



6.4.4 글로벌 가시성

객체 A에서 객체 B로의 글로벌 가시성은 객체 B가 객체 A에게 글로벌일 때 존재한다. 글로벌 가시성은 객체 A와 객체 B가 존재하는 한 지속되기 때문에 상대적으로 영구적인 가시성이다. 이를 이루기 위한 가장 확실한 방법은 글로벌 변수에 인스턴스를 할당하는 것이다.

7. 설계 클래스 다이어그램

상호작용 다이어그램의 작성이 완료되면 소프트웨어 해결책에 참여하는 소프트웨어 클래스와 인터페이스의 명세서를 작성할 수 있다. 개념 모델과 달리 설계 클래스 다이어그램은 현실 세계의 개념이 아닌 소프트웨어 엔티티를 위한 정의를 보여준다.

설계 클래스 다이어그램은 시스템 설계 단계에서 정의되며, 상호작용 다이어그램, 개념모델 다이어그램 산출물에 의존한다.

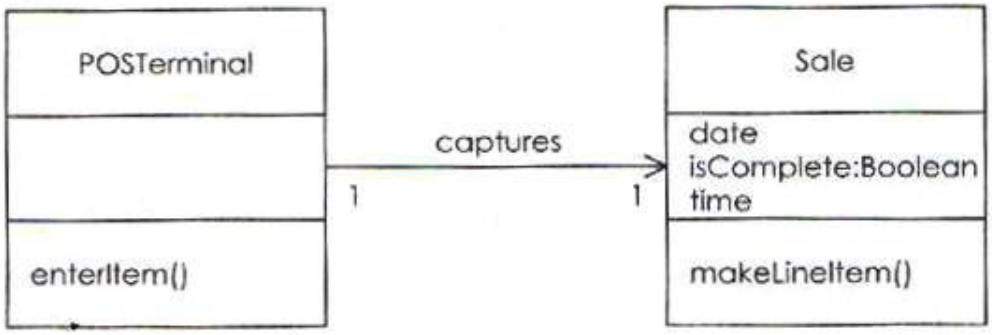
7.1 설계 클래스 다이어그램의 정의

설계 클래스 다이어그램은 응용 시스템의 소프트웨어 클래스와 인터페이스에 대한 명세서를 설명한다.

설계 클래스 다이어그램 정보
• 클래스, 연관 관계 및 애트리뷰트
• 인터페이스, 오퍼레이션 상수(constant)
• 메소드
• 애트리뷰트 타입 정보
• 네비게이션
• 종속 관계

설계 클래스 다이어그램 작성 절차
① 소프트웨어 해결책에 참여하는 모든 클래스를 식별한다. 클래스의 식별은 상호작용 다이어그램을 분석하여 수행한다.
② 식별한 클래스를 클래스 다이어그램에 표현한다.
③ 개념 모델의 관련 개념으로부터 클래스의 애트리뷰를 식별한다.
④ 상호작용 다이어그램을 분석하여 메소드 이름을 추가한다.
⑤ 애트리뷰트와 메소드에 타입 정보를 추가한다
⑥ 요구되는 애트리뷰트 가시성을 지원하기 위한 연관 관계를 추가한다.
⑦ 애트리뷰트 가시성의 방향을 표시하기 위하여 연관 관계에 네비게이션 화살표를 추가한다.
⑧ 비애트리뷰트(non-attribute) 가시성을 표시하기 위하여 종속 관계를 추가한다.

설계 클래스 다이어그램


POSTerminal
enterItem()
captures
1
1
Sale
date
isComplete:Boolean
time
makeLineItem()

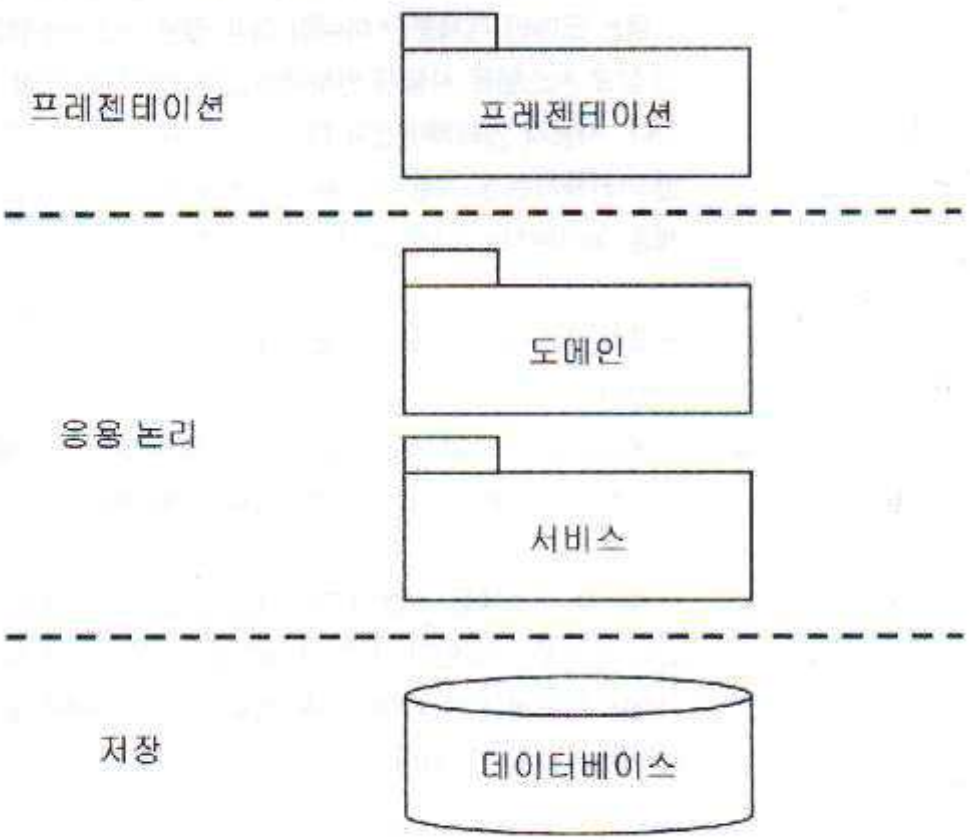
8. 시스템 설계 패턴

도메인 객체는 시스템의 핵심적인 개념과 행위를 정의한다. 그러나 대부분의 시스템은 도메인 객체뿐만 아니라 다른 많은 서브시스템을 포함하고 있다. 사용자 인터페이스와 데이터 저장을 포함하는 정보 시스템에 대한 일반적인 아키텍처는 3-계층 아키텍처(three-tier architecture)로 알려져 있다. 3-계층 아키텍처는 프레젠테이션, 응용 논리, 저장으로 구성된다. 객체지향 시스템을 위한 아키텍처는 전통적인 3-계층 아키텍처가 제시하는 책임의 분리를 수용한다. 이러한 책임은 각각의 소프트웨어 객체에게 할당된다. 그러나 객체지향 설계에서 응용 논리 계층은 도메인 객체, 서비스 객체로 더욱 세분화 하여 분할할 수 있다.

8.1 패키지

UML은 일련의 모델링 요소 혹은 서브시스템 그룹을 표시하기 위하여 패키지 메커니즘을 제공한다. 전체 시스템은 시스템 패키지라고 하는 최고 수준의 단일 패키지 범위 내에 있는 것으로 간주한다. UML 표기법을 이용하여 시스템 아키텍처를 표현하기 위해서는 패키지를 사용 할 수 있다.

아키텍처 패키지 다이어그램



8.2 Facade 패턴

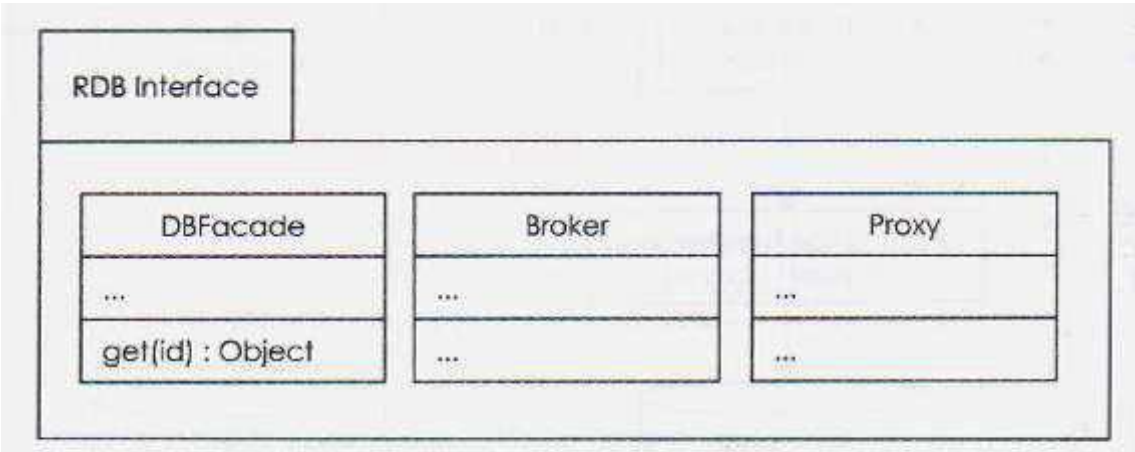
다른 컴포넌트 그룹에 대한 공통적인 인터페이스나 이질적인 일련의 인터페이스를 제공하는 클래스를 정의할 때 Facade 패턴을 사용한다.

Facade 패턴은 다음과 같은 질문에 대한 답을 얻을 수 있다.

- 시스템 컴포넌트들은 서비스 패키지 내의 클래스와 어떻게 인터페이스를 하여야 하는가?
- 객체와 데이터베이스 레코드간의 변환을 위한 서비스는 누가 제공하는가?

패턴 이름	Facade
문제	• 서브시스템과 같은 이질적인 일련의 인터페이스에 대한 공통적인 통합 인터페이스가 필요하다. 무엇을 하여야 하는가?
해결책	• 인터페이스를 통합하는 하나의 클래스를 정의하여 서브시스템과 협동하는 책임을 할당한다.

Facade 패턴의 적용



8.3 Model-View Separation 패턴

일반적으로 윈도우 객체와 다른 컴포넌트간에는 직접적인 결합이 없도록 하는 것이 바람직하다. 이는 비 윈도우 컴포넌트들은 새로운 응용에서 재사용될 수 있기 때문이다. 이것이 Model-view Separation 패턴의 기본적인 원리이다. 모델은 도메인 객체를 의미하며, 뷰는 프레젠테이션 객체를 의미한다. 모델 객체가 뷰 객체와 직접적으로 결합되지 않도록 한다.

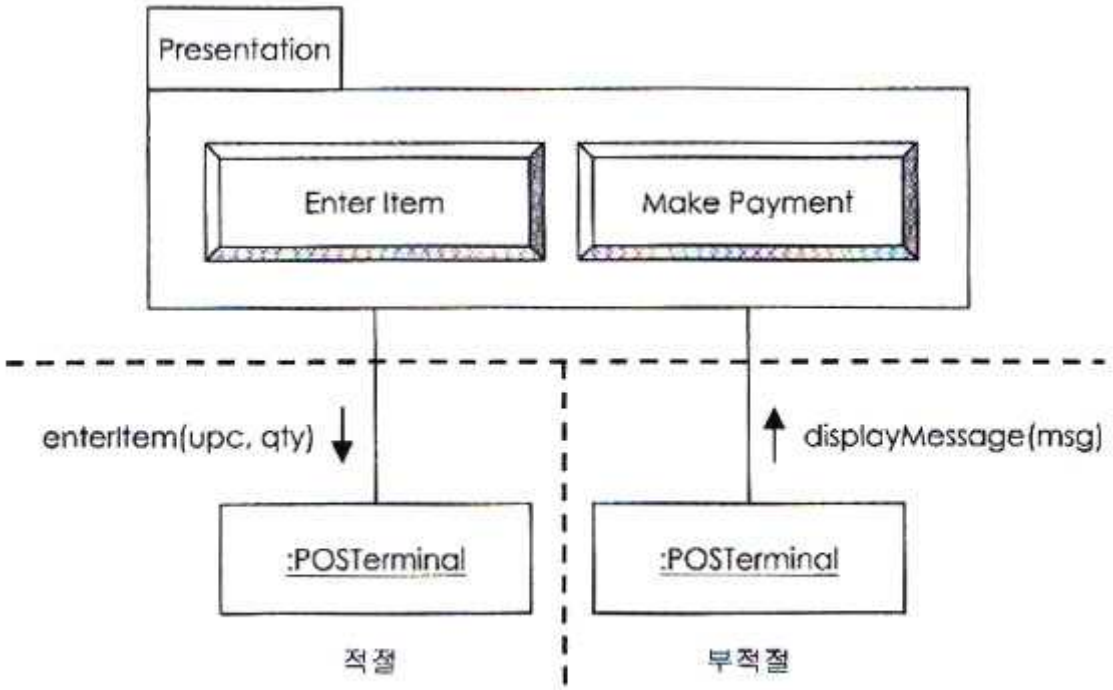
Model-View Separation 패턴은 다음과 같은 질문에 대한 답을 얻을 수 있다.

- 프레젠테이션 계층에 대하여 다른 패키지들은 어떠한 유형의 가시성을 가져야 하는가?
- 비윈도우(non-window) 클래스들은 윈도우와 어떻게 통신하여야 하는가?

패턴 이름	Model-View Separation
문제	• 도메인 객체의 재사용성을 증가시키고 인터페이스의 변경이 도메인 객체에 미치는 영향을 최소화하기 위하여 도메인 객체와 윈도우를 결합시키지 않는 것이 바람직하다. 무엇을 하여야 하는가?
해결책	• 도메인 클래스가 윈도우 클래스에 대한 가시성을 갖지 않거나 직접적인 결합이 이루어지지 않도록 도메인 클래스를 정의한다. 그리고 응용 데이터와 기능이 도메인 클래스에서 유지되도록 한다.

Model-View Separation 패턴의 장점
• 도메인 프로세스에 초점을 맞추는 응집력있는 모델 정의를 지원한다.
• 모델과 사용자 인터페이스 계층의 분리 개발을 허용한다.
• 인터페이스 요구사항 변경이 도메인 계층에 미치는 영향을 최소화한다.
• 도메인 계층에 영향을 미치지 않으면서도 기존의 도메인 계층에 새로운 뷰를 쉽게 연결할 수 있도록 한다.
• 동일한 모델 객체에 대한 다양한 뷰를 동시적으로 허용한다.
• 사용자 인터페이스 계층과 독립적인 모델 계층의 실행을 허용한다.
• 새로운 사용자 인터페이스 프레임워크에 대한 모델 계층의 이식을 용이하게 한다.

Model-View Separation 패턴



8.4 Publish-Subscribe 패턴

도메인 객체가 윈도우와 간접적으로 통신해야 할 필요가 발생한다. Publish-Subscribe 패턴 혹은 Observer 패턴은 도메인 객체와 윈도우간의 간접 통신에 대한 해결책을 제공한다.

Publish-Subscribe 패턴은 다음과 같은 질문에 대한 답을 얻을 수 있다.

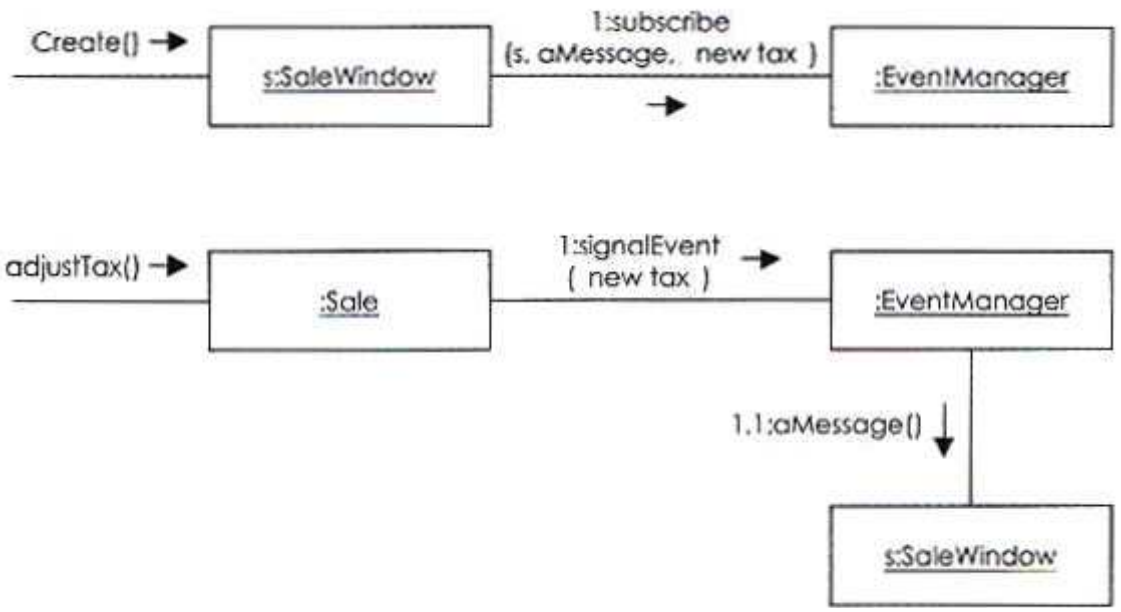
• 윈도우는 표시할 정보를 어떻게 얻을 수 있는가?

패턴 이름	Publish-Subscribe
문제	• 시스템 이벤트의 발표자(Publisher) 내에서 상태 변화가 발생하고, 이벤트의 구독자(subscribe)인 다른 객체가 이러한 이벤트에 관심을 갖거나 종속된다. 그러나 발표자는 구독자에 대한 직접적인 가시성을 가져서는 안 된다. 무엇을 하여야 하는가?
해결책	• 발표자가 구독자에게 간접적으로 통보할 수 있도록 이벤트 통보 시스템 (event notification system)을 정의한다.

Publish-Subscribe 아키텍처는 이벤트 통보와 간접 통신을 위한 일반적인 케머니즘을 제공한다. 이는 이벤트 통보 기반의 설계(event notification-based design) 라는 새로운 객체지향 시스템 설계 방법을 제시한다.

이벤트 통보 기반의 설계 효과
• 송신 객체와 수신 객체간의 직접 결합이 필요하지 않다.
• 단일 이벤트를 다수의 구독자에게 브로드캐스트할 수 있다.
• 이벤트에 대한 반은은 Callback 객체로 일반화될 수 있다.
• 각각의 Callback을 독자적인 스레드에서 실행시킴으로써 병행성을 쉽게 제공할 수 있다.

Publish-Subscribe 패턴



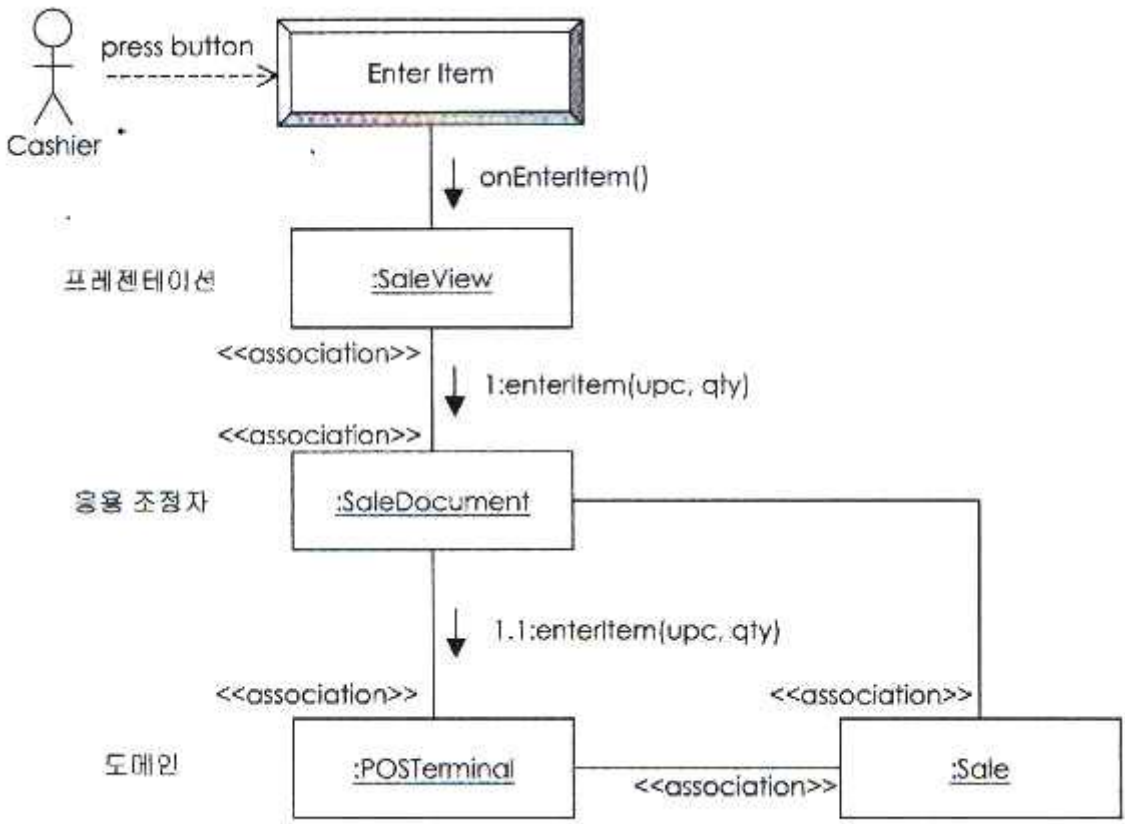
8.5 응용 조정자

응용 조정자(application coordinator)는 인터페이스와 도메인 계층간의 조정을 책임지는 클래스이다.

응용 조정자의 기본적인 책임
• 도메인 객체와 인터페이스간의 정보를 매핑시킨다.
• 인터페이스로부터의 이벤트에 응답한다.
• 도메인 객체로부터의 정보를 표시하는 윈도우를 연다.
• 트랜잭션을 관리한다.

응용 조정자의 다중 뷰 책임
• 응용 조정자 인스턴스로부터의 정보를 동시에 표시하는 다중 윈도우를 지원한다.
• 정보가 변경되거나 윈도우가 새로운 정보로 변경되었을 때 종속 윈도우에게 통보한다.

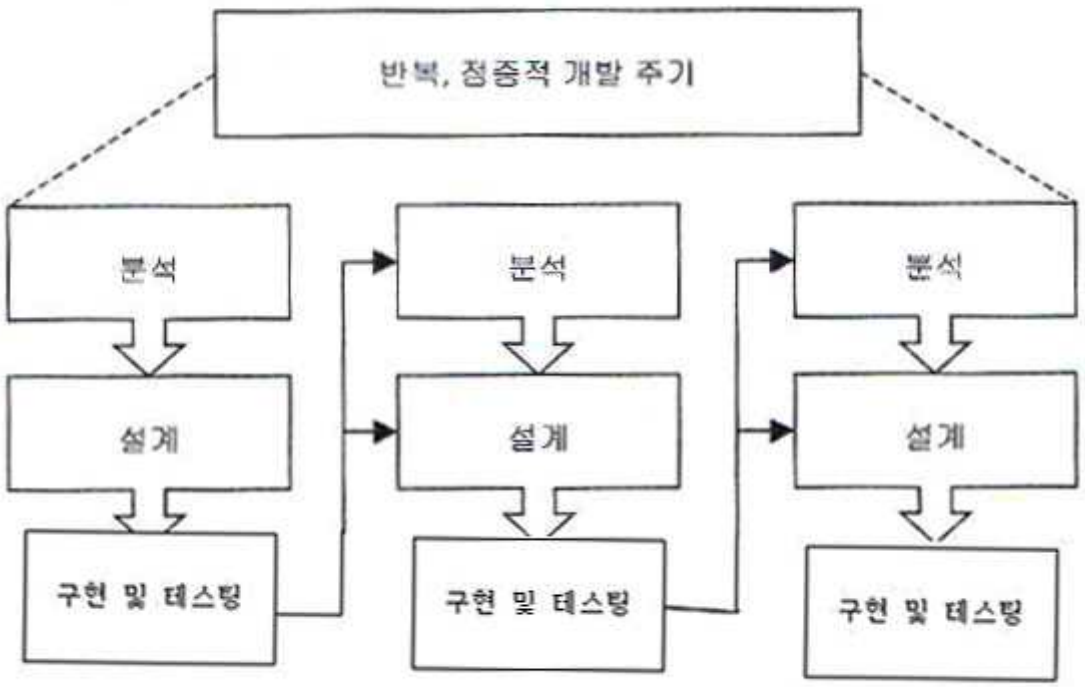
뷰 아키텍처 계창근의 협동과 가시성



9. 시스템 구현

상호작용 다이어그램과 설계 클래스 다이어그램이 완성되었으면, 도메인 객체에 대한 코드를 생성할 수 있는 정도의 충분한 세부사항이 확보되어 있는 상태이다. 코드의 생성은 객체지향 분석 및 설계의 부분은 아니다. 객체지향 분석·설계와 객체지향 프로그래밍은 요구사항에서 코드에 이르기까지 완전한 길잡이를 제공한다.

하지만 설계 과정의 산출물이 불완전하기 때문에 프로그래밍과 테스팅 동안에 구체적인 문제점들을 식별하여 해결하여야 한다.



9.1 클래스 정의 생성

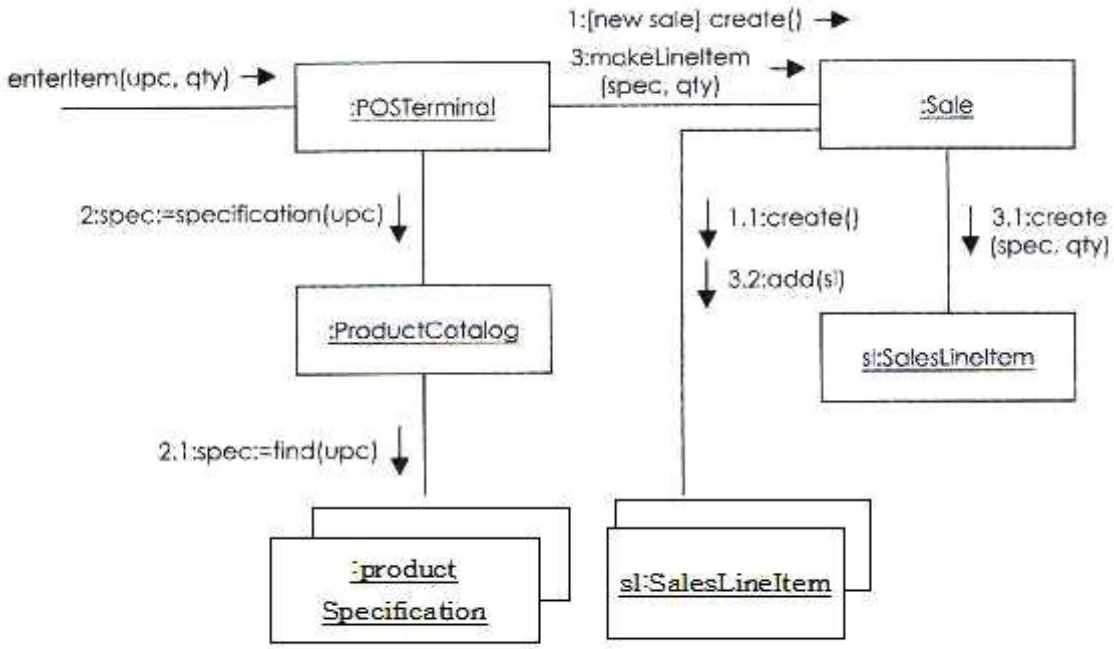
설계 클래스 다이어그램은 최소한 클래스의 이름, 수퍼클래스, 메소드 시그너처, 클래스의 단순 애트리뷰트 등을 서술한다. 이는 기본적인 클래스 정의에 충분한 정보이다.

클래스의 참조 애트리뷰트는 클래스 다이어그램의 연관 관계와 네비게이션에 의하여 식별된다. 연관 관계의 역할 이름은 역할을 식별하는 이름이다. 만약 클래스 다이어그램에서 역할 이름이 사용되었다면, 코드 생성 동안에 참조 애트리뷰트의 이름으로서 역할 이름을 사용한다.

9.2 메소드 생성

협동 다이어그램은 메소드 호출에 응답하여 보내진 메소드를 보여준다. 이러한 메시지의 순서는 메소드 정의의 일련의 문장으로 전환된다.

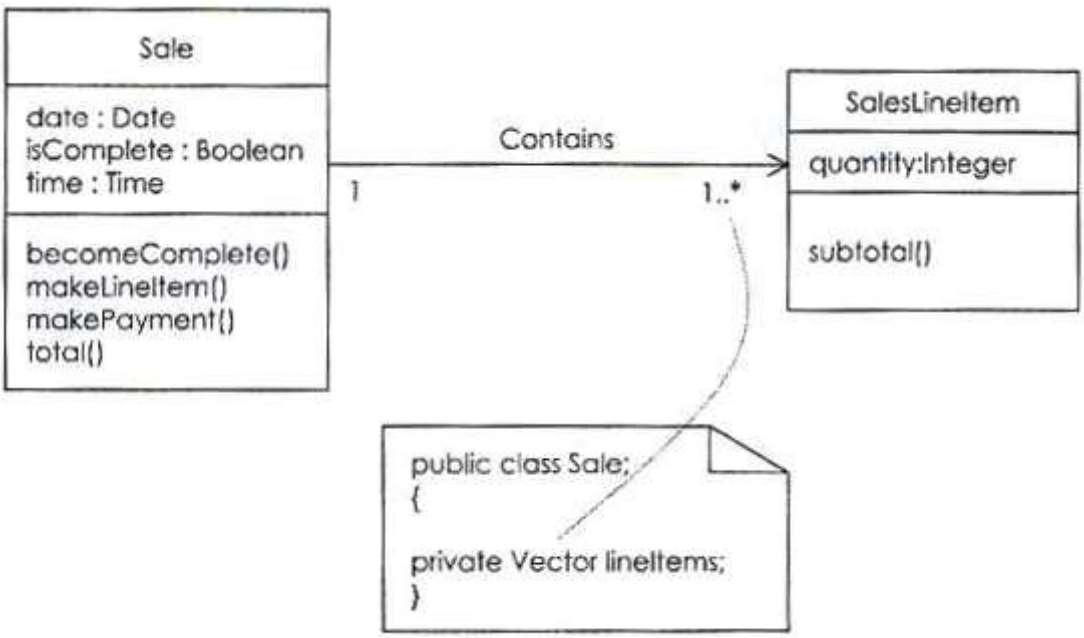
협동 다이어그램



9.3 컨테이너/컬렉션 클래스

객체는 종종 다른 객체의 그룹에 대한 가시성을 유지할 필요가 있다. 이러한 필요성은 클래스 다이어그램의 다중성 값에 의하여 발생한다.

컨테이너 추가



9.4 구현 순서

클래스는 결합도가 낮은 클래스부터 시작하여 구현하는 것이 바람직하다. 즉, 완전한 단위 테스팅이 가능하도록 구현하는 것이 좋다.

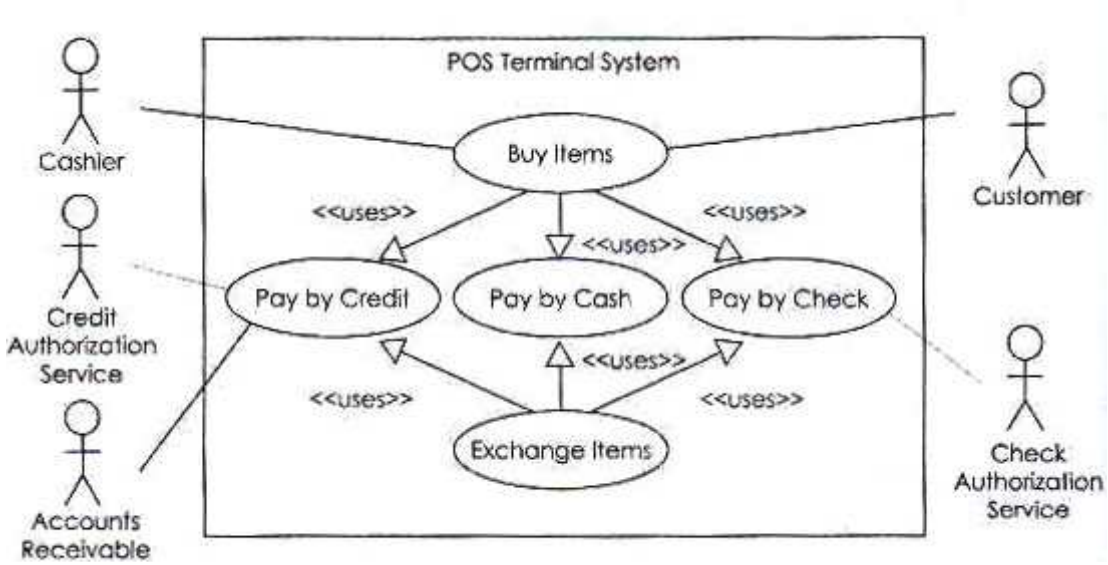
10. 사용 관계 및 타입 관계

반복적인 객체지향 분석 및 설계에서는 개발 주기가 진행됨에 따라 동일한 유스 케이스라 하더라도 점차 많은 기능이 추가되어 최종적인 유스 케이스에 이르게 된다.
유스 케이스의 단계 혹은 분기에 의한 행위들이 다른 유스 케이스에서도 중복되거나 복잡하고 길 경우 이들을 별도의 유스 케이스로 분리하는 것이 이해가 용이하다.

10.1 유스 케이스 다이어그램과 사용 관계

하나의 유스 케이스가 다른 유스 케이스의 행위를 시작시키거나 포함하는 경우, 첫 번째 유스 케이스가 두 번째 유스 케이스를 사용한다고 하며 유스 케이스간에는 사용 관계(uses relationship)가 존재한다고 한다.
UML에서 사용 관계는 스트레오 타입<<uses>>를 갖는 일반화로 표현한다.

사용 관계에 의한 유스 케이스의 연결

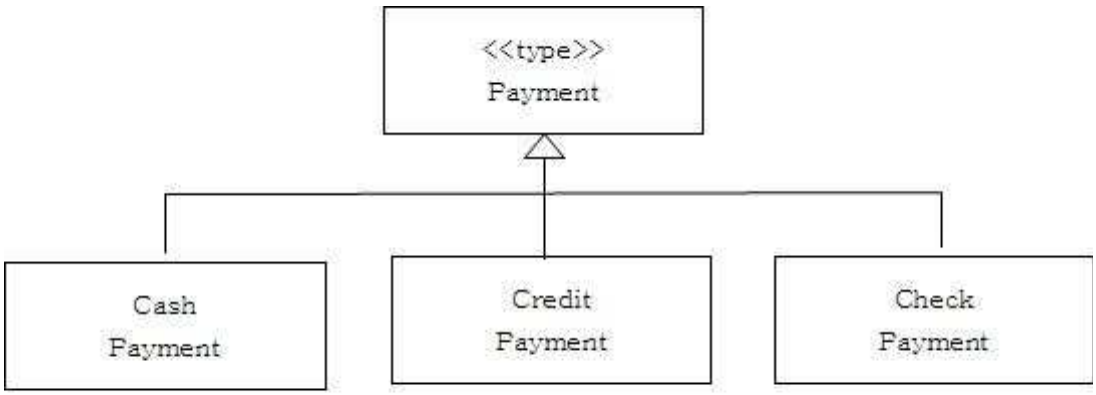


유스 케이스가 사용 관계에 있을 때 유스 케이스 문서는 이러한 관계를 표현하여야 한다. 따라서 다른 유스 케이스의 행위를 사용하는 유스 케이스는 유스 케이스 내용에서 “시작하다(initiate)"라는 용어를 사용하여 사용 관계를 표시하여야 한다.

10.2 일반화

일반화는 개념간의 공통성을 식별하고 수포타입과 서브타입 관계를 식별하는 활동이다. 타입 계층에서 수퍼타입은 보다 일반적인 개념을 표현하며, 서브타입은 보다 세부적인 개념을 표현한다. 타입 관계를 이용하면 표현을 경제적으로 할 수 있으며, 이해가 용이하고, 중복 정보의 표현을 감소시킬 수 있다.

타입 계층



10.3 수퍼타입과 서브타입

수퍼타입 정의는 서브타입 정의보다 일반적이며 포괄적이다. 또한 서브타입 세트의 모든 멤버는 자신이 속해 있는 수퍼타입 세트의 멤버이다. 타입 계층을 정의할 때 수퍼타입에 대한 모든 사항은 서브타입에도 적용된다. 즉, 수퍼타입의 애트리뷰트와 연관 관계가 모두 서브타입에 적용된다. 또한 수퍼타입과 서브타입간에는 is-a 관계가 성립된다.

타입 분할은 타입을 분리된 별도의 서브타입으로 구분한다.

서브타입 정의의 일반적인 경우
• 서브타입이 추가적인 애트리뷰트를 갖는다.
• 서브타입이 추가적인 연관 관계를 갖는다.
• 서브타입 개념이 수퍼타입이나 다른 서브타입과는 다른 방식으로 운영되거나, 처리되거나, 조작된다.
• 서브타입 개념이 수퍼타입이나 다른 서브타입과는 다른 방식으로 행동하는 대상을 표현한다.

수퍼타입의 일반화는 일반적으로 예상되는 서브타입간에 어떠한 공통점이 식별될 때 수행한다.

수퍼타입을 정의하는 경우
• 예상되는 서브타입들이 유사한 개념에 대한 변형을 표현한다.
• 모든 서브타입이 수퍼타입으로 추출하여 표현할 수 있는 동일한 애트리뷰트를 갖는다.
• 모든 서브타입이 수퍼타입으로 추출하여 표현할 수 있는 동일한 연관 관계를 갖는다.
• 서브타입 세트의 모든 멤버가 자신이 속할 수퍼타입 세트의 멤버가 된다.

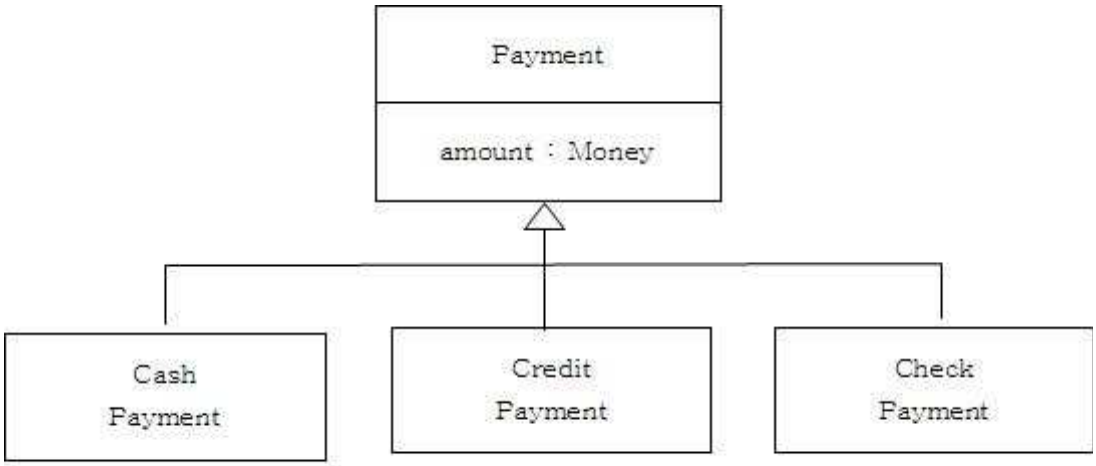
10.4 추상 타입

추상타입(abstract type)은 문제 도메인의 규칙을 명확하게 설명해 주기 때문에 이들을 개념 모델에서 식별하는 것이 유용하다. 타입 T의 모든 멤버가 서브타입의 멤버가 된다면, 타입 T는 추상 타입이라고 한다.

UML에서 추상타입은 이탤릭체로 표현한다. 추상 타입이 설계 단계에서 클래스로 구

현된다면, 이를 추상 클래스라고 한다. 추상 클래스는 클래스를 위한 인스턴스를 생성하지 않는다. 추상 메소드는 추상 클래스에서 선언되는 메소드로서 구현되지 않는다.

추상 타입

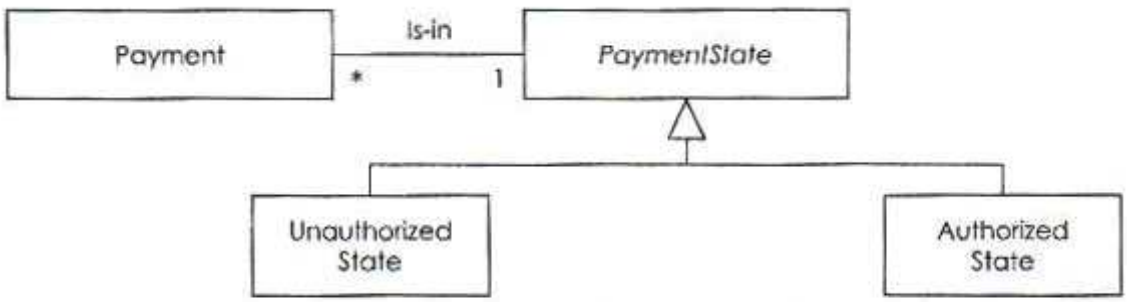


10.5 상태 변화의 모델링

개념의 상태가 변화하는 경우의 모델링을 수행하기 위해서는 기준을 적용하는 것이 바람직하다.

개념 상태 변화에 따른 모델링 수행 기준
• 개념 X의 상태를 X의 서브타입으로 모델링해서는 안 된다.
• 그 대신에, 상태 계층(state hierarchy)을 정의하고 상태를 X에 연관시킨다.
• 혹은 개념의 상태를 개념 모델에 표시하지 말고 상태 다이어그램에 표시한다.

상태 변화 모델링



11. 패키지와 개념 모델의 확장

모델링 요소를 패키지로 구성하는 것은 세부적인 요소들을 보다 커다란 추상화로 묶을 수 있기 때문에 고수준의 뷰를 제공하고 단순한 그룹핑으로 모델을 볼 수 있도록 해준다. 전체 시스템 아키텍처는 수직적인 계층과 수평적인 분할로 이루어진다. 개념 모델의 패키지는 도메인 계층의 객체에 대한 분할로 간주할 수 있다.

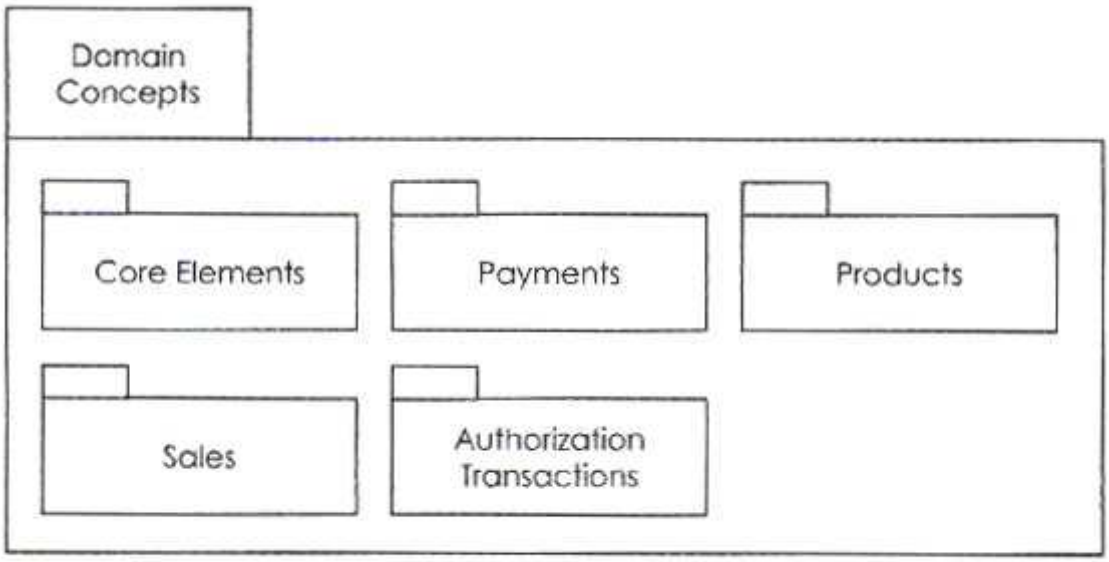
11.1 개념 모델의 분할

개념 모델을 패키지로 분할하기 위해서는 기준에 따라 요소를 배치하여야 한다.

개념 모델 분할을 위한 기준
• 동일한 주제 영역 즉, 밀접하게 관련된 개념 혹은 목적들을 함께 위치시킨다.
• 타입 계층에 함께 존재하는 요소들은 함께 위치시킨다.
• 동일한 유스 케이스에 참여하는 요소들은 함께 위치시킨다.
• 연관 관계로 강력하게 연결되어 있는 요소들은 함께 위치시킨다.

개념 모델의 모든 요소들은 도메인 개념이라는 패키지를 최상위의 패키지로 하여 관련시키는 것이 유용하다.

도메인 개념 패키지

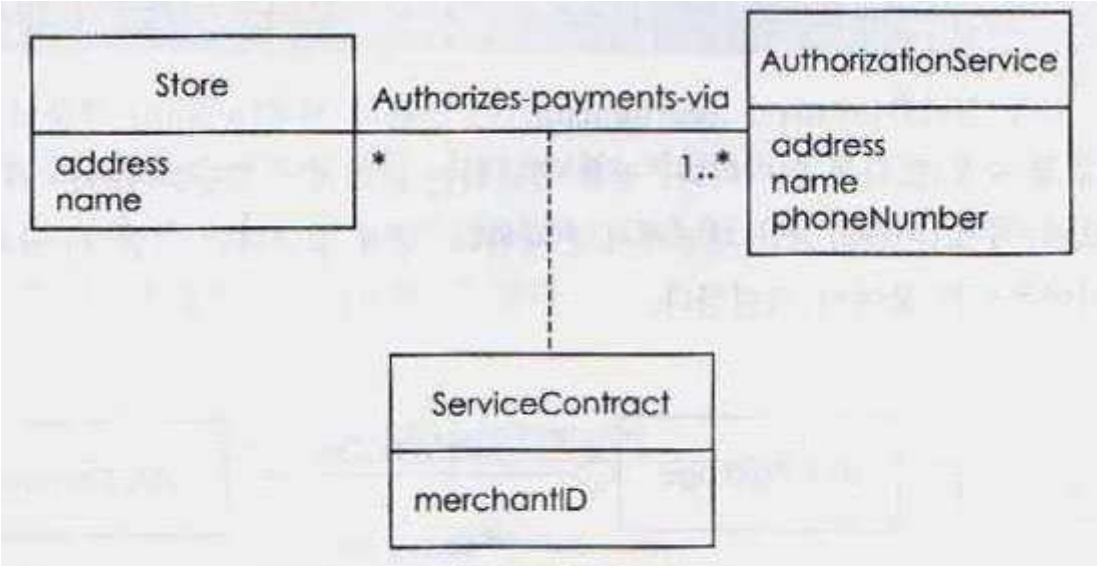


11.2 연관 클래스

개념 모델에서 만약 타입 T가 동일한 유형의 애트리뷰트 A에 대하여 많은 값을 동시에 가질 수 있다면, 애트리뷰트 A를 타입 T에 위치시켜서는 안 된다. 애트리뷰트 A는 타입 T와 연관되는 다른 타입에 위치시켜야 한다.

연관 클래스(associative class) 혹은 연관 타입(associative type)은 연관 관계 자체에 애트리뷰트를 부여할 수 있도록 한다.

연관 클래스



개념 모델에서 연관 클래스의 사용이 유용한 경우
• 애트리뷰트가 연관 관계에 관련되어 있다.
• 연관 클래스의 인스턴스가 연관 관계에 대하여 수명 종속 관계(life-time dependency)를 갖는다. 즉, 연관 타입의 수명이 연관 관계의 종속된다.
• 두 개념간에 다-대-다(many-to-many) 연관이 존재하고, 연관 관계 자체에 관련된 정보가 존재한다.
• 연관 관계에 참여하는 두 객체간에 단지 하나의 연관 클래스 인스턴스만이 존재한다.

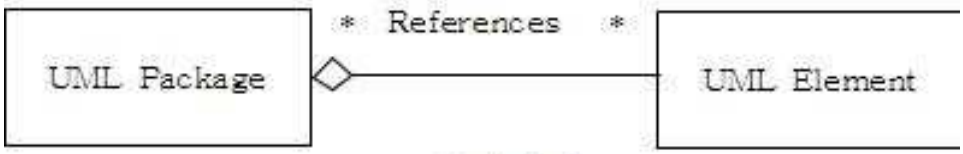
11.3 집단화 및 합성

집단화(aggregation)는 연관의 특수한 유형으로서 whole-part 관계를 모델링하기 위하여 사용된다. 합성(composition)은 연관의 전체(whole) 부분의 다중성이 최대 일이라는 의미이다. 즉, 전체 부분이 부품(part) 부분을 독점적으로 소유하며, 트리 구조의 부품 계층이 존재한다는 의미이다. 합성은 연관의 전체 부분에 흑색 다이어몬드를 붙여서 표현한다.

합성

공유 집단화(shared aggregation)는 연관의 전체(whole) 부분의 다중성이 일보다 클 수 있다는 의미이다. 공유 집단화는 물리적인 집단화에서는 거의 존재하지 않으며, 주로 비물리적 개념에서 존재한다. 공유 집단화는 흰색 다이아몬드를 붙여서 표현한다.

공유 집단화

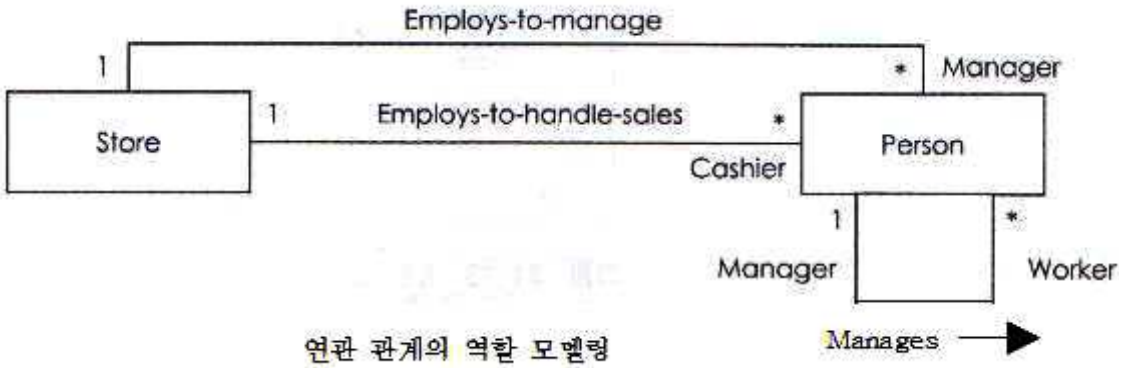


집단화 표현을 하는 경우
• whole-part 관계에서, 부품의 수명이 전체의 수명에 의하여 제한된다. 즉, 전체와 부품간에 생성-소멸 종속 관계(create-delete dependency)가 존재한다.
• 명백한 whole-part의 물리적 혹은 논리적인 조립 관계가 존재한다.
• 전체 부분의 일부 특성이 부품에게 전달된다.
• 전체 부분에 적용되는 오퍼레이션이 부품에게 전달된다.

11.4 연관의 역할 표현

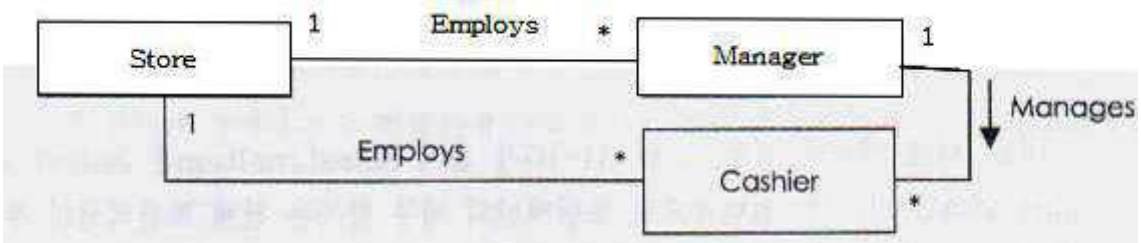
개념 모델에서 현실 세계의 역할, 특히 인간의 역할은 다양한 방법으로 모델링될 수 있다.

연관 관계의 역할 모델링



연관 관계의 역할 모델링

개념으로서의 역할 모델링

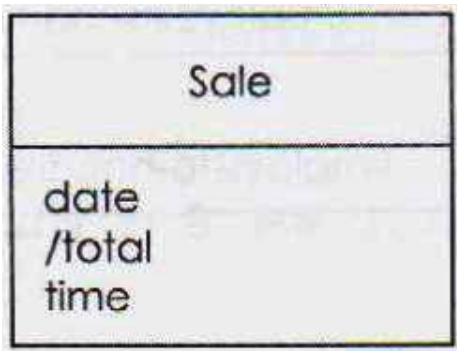


두 가지의 모델링 방법은 모두 각자의 장점을 갖는다. 연관 관계의 역할 모델링은 동일한 사람이 여러 가지의 연관 관계에서 다양한 역할을 갖는 혈실을 정확하게 모델링 할 수 있다는 장점이 있다. 반면 개념적으로의 역할 모델링은 각각의 역할에 대한 고유한 애트리뷰트, 연관 관계 및 부수적인 세멘틱스를 추가할 수 있는 융통성과 용이함을 제공한다.

11.5 유도 요소

유도 요소(derived element)는 다른 요소로부터 결정될 수 있다. 애트리뷰트와 연관 관계는 가장 일반적인 유도 요소이다. UML에서 유도 요소는 요소의 이름 앞에 "/" 기호를 붙여서 표기한다.

유도 요소



12. 행위 모델링

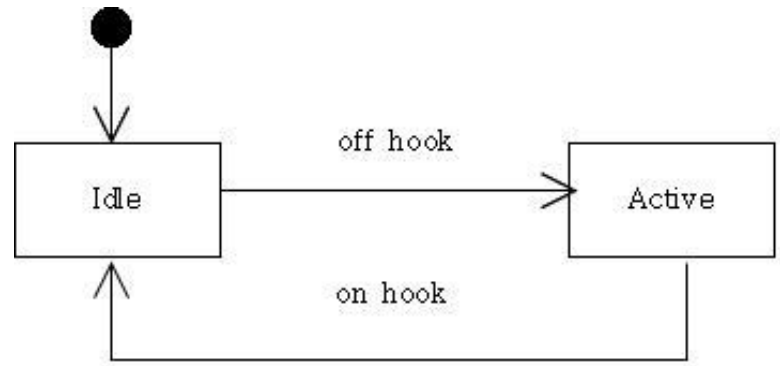
UML은 이벤트와 객체의 상태를 설명하기 위하여 상태 다이어그램을 사용한다. 일반적으로 상태 다이어그램의 사용은 유스 케이스의 시스템 이벤트를 보이기 위한 것이다. 그러나 상태 다이어그램은 다른 모델링 타입에도 적용될 수 있다.

이벤트는 고려할 만한 가치가 있는 어떠한 사건의 발생을 의미한다. 상태는 어느 한 시점에서 혹은 이벤트간의 시간 가격에서 객체의 조건을 나타낸다. 변환은 이벤트가 발생했을 때 객체가 이전 상태에서 다음 상태로 이동하는 두 개의 상태간의 관계이다.

12.1 상태 다이어그램

UML의 상태 다이어그램은 이벤트와 객체의 상태, 그리고 이벤트에 반응하는 객체의 행위를 보여준다. 변환은 이벤트를 이름으로 갖는 화살표로 표시된다.

상태 다이어그램



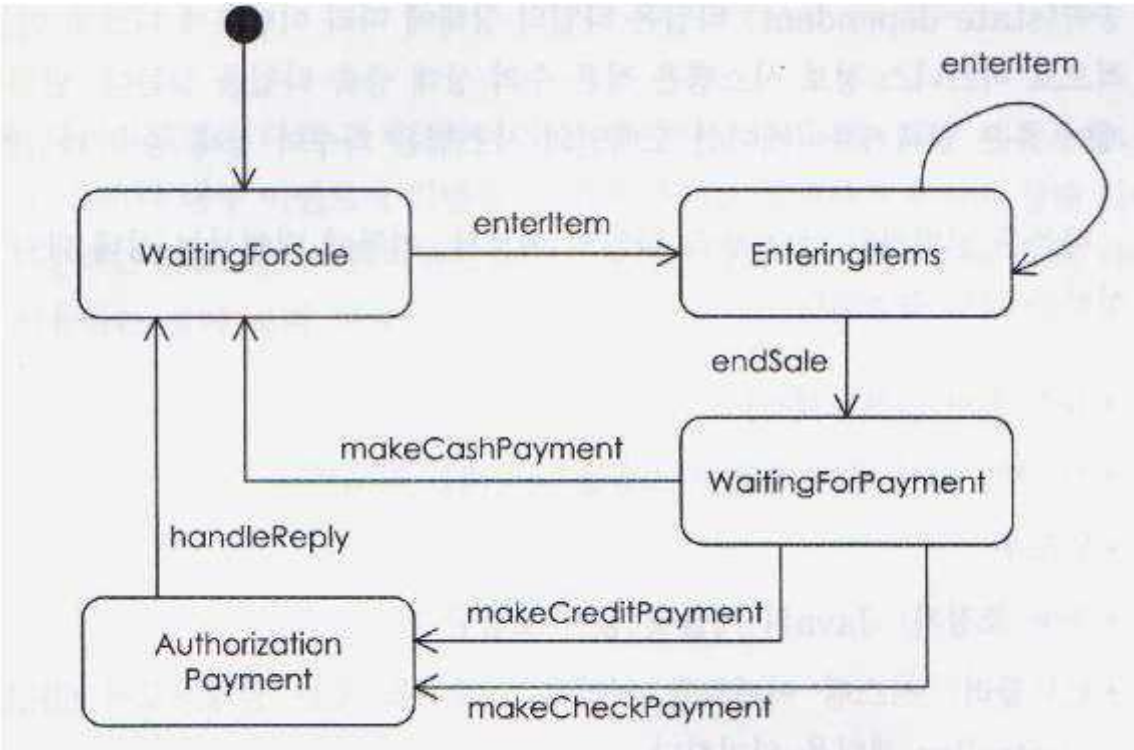
상태 다이어그램은 객체의 수명주기를 보여준다. 즉, 객체가 어떠한 이벤트를 경험하면, 이벤트간에 어떠한 상태를 갖고, 어떠한 변환이 일어나는지 등을 보여준다. 상태 다이어그램은 소프트웨어 클래스, 타입(개념), 유스 케이스 같은 UML요소에 적용될 수 있다.

전체 시스템은 문제 도메인에 존재하는 타입, 개념 혹은 일련의 시스템으로 표현될 수 있기 때문에 전체 시스템 역시 독자적인 상태 다이어그램을 가질 수 있다.

12.2 유스 케이스 상태 다이어그램

상태 다이어그램은 유스 케이스의 범위에서 시스템에 의해 읜식됙 처리되는 외부 시스템 이벤트의 올바른 순서를 표현하는 것이다.

유스 케이스 상태 다이어그램



유스 케이스 내의 모든 시스템 이벤트와 그 순서를 표현하는 상태 다이어그램을 유스 케이스 상태 다이어그램이라고 한다. 시스템 설계자들은 이러한 유스 케이스 상태 다이어그램을 이용하여 시스템 이벤트간의 올바른 순서가 보장되는 설계를 할 수 있게 된다.

시스템 상태 다이어그램은 모든 유스 케이스 상태 다이어그램의 합집합(union)이라고 할 수 있다.

12.3 상태 종속 타입 및 상태 독립 타입

객체가 이벤트에 대하여 항상 동일하게 반응한다면, 객체는 해당 이벤트에 대하여 상태 독립적(state-independent)이라고 할 수 있다. 이와 같이 모든 이벤트에 대하여 항상 동일하게 반응하는 타입을 상태 독립 타입이라고 한다. 복잡한 행위를 갖는 상태 독립 타입에 대해서는 상태 다이어그램을 작성하는 것이 바람직하다. 반면에 상태 종속(state-dependent) 타입은 타입의 상태에 따라 이벤트에 다르게 반응한다.

상태 종속 타입의 예
• 유스 케이스(프로세스)
• 시스템 : 전체 응용 혹은 시스템을 나타내는 타입
• 윈도우
• 응용 조정자 : Java 애플릿 포함
• 컨트롤러 : 시스템 이벤트를 처리하는 책임을 갖는 클래스로서 GRASP 패턴의 Controller 패턴을 의미
• 트랜잭션 : 이벤트에 대한 트랜잭션의 반응은 종종 트랜잭션 수명주기에서의 현재 상태에 종속
• 장치
• 변경자(mutator): 타입 혹은 역할을 변경하는 타입

12.4 이벤트 타입

이벤트는 외부 이벤트(external event), 내부 이벤트(internal event), 시간 이벤트(temporal event)로 구분된다.

이벤트의 구분
• 외부 이벤트 : 시스템 이벤트라고도 하며, 시스템 범위의 외부로부터 발생하는 이벤트이다. 시스템 순차 다이어그램은 외부 이벤트를 설명한다. 외부 이벤트는 이에 반응하는 시스템 오퍼레이션의 호출을 유발한다.
• 내부 이벤트 : 시스템 범위의 내부에서 발생하는 이벤트이다. 소프트웨어에서 내부 이벤트는 다른 내부 객체로부터 보내진 메시지 혹은 신호를 통하여 오퍼레이션이 호출될 때 발생한다. 협동 다이어그램의 메시지는 내부 이벤트를 설명한다.
• 시간 이벤트 : 특정 날짜 혹은 시간이나 시간의 전달에 의하여 발생하는 이벤트이다. 소프트웨어에서 시간 이벤트는 실시간 혹은 타임 시뮬레이션에 의하여 유도된다.

실제 다이어그램은 다른 객체로부터 받은 메시지를 나타내는 내부 이벤트를 표현할 수 있다. 그러나 내부 이벤트에 기반을 둔 객체 행위를 설계하기 위하여 상태 다이어그램을 사용한는 것보다는 외부 이벤트와 시간 이벤트를 표현하기 위하여 상태 다이어그램을 사용하는 것이 보다 바람직하다.

UML이 무엇이며 왜 중요한가?

UML은 소프트웨어 시스템이나 업무 모델링 그리고 기타 비 소프트웨어 시스템 등을 나타내는 가공물을 구체화하고, 시각화하고, 구축하고, 문서화하기 위해 만들어진 언어이다.

UML은 Rational Software와 그 협력회사에 의해 개발되었다.

업무 처리과정에서 그 업무의 범위와 규모가 커짐에 따른 시스템의 복잡성을 처리할 필요성을 느끼게 되었는데, 특히 물리적인 시스템의 분산, 동시성, 반복성, 보안, 결점 보완, 시스템들의 부하에 대한 균등화와 같은 반복해서 발생하는 구조적 문제에 대한 프로세스가 필요하게 되었으며다. 또한 웹의 발전에 따라 시스템을 만들기는 쉬워졌으나 이러한 구조적 문제는 더욱 악화되었기에 이러한 모든 필요성에 의해 UML은 만들어졌다.

모델링의 중요성

시스템의 복잡성이 증가함에 따라, 강력한 소프트웨어 시스템을 만들기 위해 구축하고 개선하기에 앞서 모델을 만드는 것이 건물을 만들기 위한 청사진을 만드는 것과 같이 핵심적인 요소가 되었다. 잘 만들어진 모델은 프로젝트 팀 간의 통신수단으로써, 구조적인 문제를 해결하기 위한 수단으로써 핵심적인 것이다.

모델링 언어가 반드시 포함해야 하는 것

모델 요소 (Model elements) : 기본적 모델링 개념과 의미
표기 (Notation) : 모델 요소의 시각적인 그림
가이드 라인 (Guide Line) : 관용적인 사용 방법

시스템의 복잡성이 증가함에 따라 객체 지향 시스템과 컴포넌트 기반 시스템을 구축하기 위한 시각적 모델링 언어를 선택하는 것이 필연적이다.

UML의 목적

. 사용자에게 즉시 사용가능하고 직관적인 시각적 모델링 언어를 제공함으로써 사용자는 의미있는 모델들을 개발하고 서로 교환할 수 있어야 한다.
. 핵심적인 개념을 확장할 수 있는 확장성과 특수화 방법을 제공한다.
. 특정 개발 프로세스와 언어에 종속되지 않아야 한다.

. 모델링 언어를 이해하기 위한 공식적인 기초를 제공한다.

. 협동(Collaboration), 프레임 워크, 패턴, 컴포넌트 같은 고수준의 개발 개념을 제공한다.

OMG(Object Management Group)-UML의 범위

. UML은 Booch, OMT(Object Modeling Technique), OOSE(Object Oriented Software Engineering)의 개념을 융화시켜 만들었기에 일반적이고 넓게 사용될 수 있다

. UML은 기존의 방법론들을 가지고 있는 어떠한 작업에도 적합한 방편을 제공한다.

. UML은 표준적인 방법론에 역점을 두지 않고, 표준적인 모델링 언어에 역점을 두었다.

UML의 범위 외부

UML은 모든 것을 포함하는 언어가 아닌, 단순하고 표준화된 모델링의 제공을 목표로 하고 있기에 산업계 전반에 걸쳐 존재하는 다양한 시스템의 디자인에 사용될 수 있는 유연성을 제공한다.

프로그래밍 언어

UML은 비쥬얼 모델링 언어지 비쥬얼 프로그래밍 언어가 아니다. 하나 어떤 의미에서는 시각적이고 의미적인 비쥬얼적 모델링 언어가 제공하는 모든 지원을 가지고 있다. UML은 실제 코드로의 지향을 위해 사용될 수 있다.

UML은 객체 언어와 밀접하게 묶여 사용이 가능하고 이는 최고의 결과를 낼 수 있다.

툴(Tools)

UML은 툴들의 상호 운용성을 위한 툴 인터페이스, 저장소, 실행시간 모델등과 같은 방법을 제공하는 것이 아니라, 의미적 메타 모델(Meta-model)을 제공한다.

UML의 주요 다이어그램

유스케이스 다이어그램	시스템의 기능과 유저를 표현한다
클래스 다이어그램	시스템의 정적인 구조를 표현한다
객체 다이어그램	시스템 어느 시점에서의(스냅샷의) 정적인 구조를 표현한다
상호작용 다이어그램 (시퀀스다이어그램, 콜레보레이션 다이어그램)	객체 간의 상호작용(메세지 교환)을 표현한다
스테이차트 다이어그램	객체의 상태추이를 표현한다
액티비티 다이어그램	처리나 업무의 흐름을 표현한다
컴포넌트 다이어그램	컴포너트 간 의존관계를 표현한다
배치도	시스템의 물리적인 구성을 표현한다
패키지 다이어그램	패키지 간 의존관계를 표현한다

각 설계공정에서 사용할 수 있는 UML

클래스 다이어그램이란

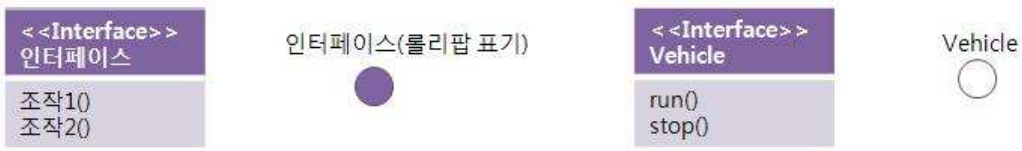
클래스 다이어그램은 "클래스"라고 하는 객체지향 설계단위를 이용하여 시스템의 정적인 구조(모델)를 표현한 것이다. 클래스 다이어그램은 분석, 설계, 구현 등 다양한 상황에서 그 사용목적에 맞게 입도를 조절하여 기술 할 수 있다.

클래스

- 객체의 구조를 표현·확인
- 속성(변수)의 사양을 표현·확인 함
- 조작(메소드)의 사양을 표현·확인 함

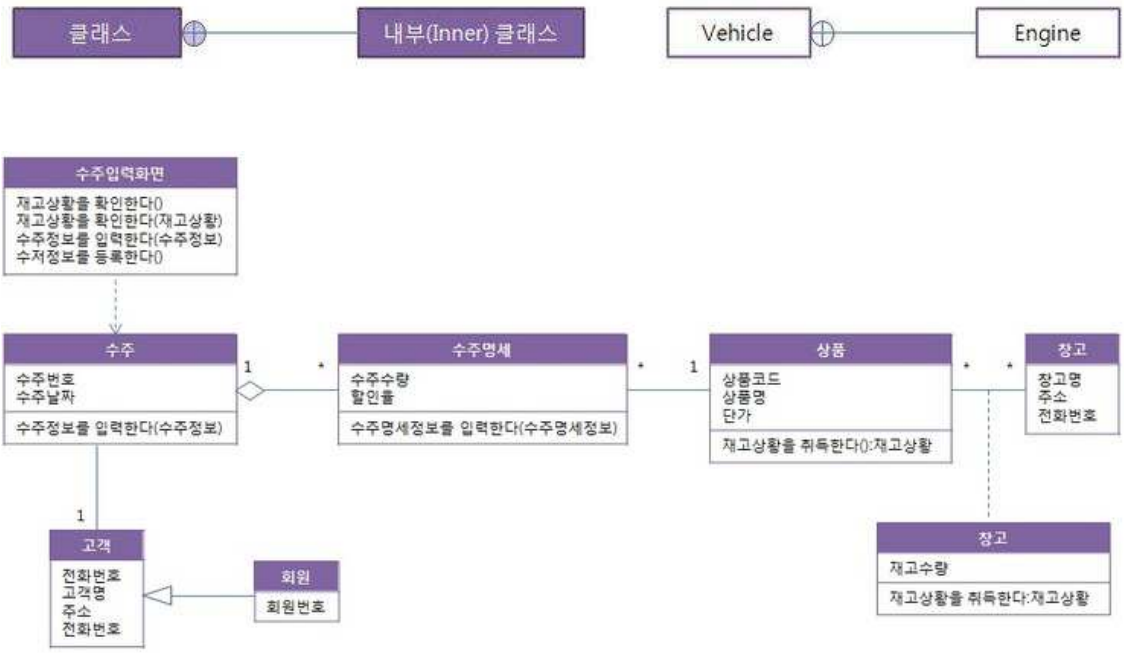
인터페이스

- 외부에 공개하는 대체 가능한 클래스의 사양을 표현·확인 함



내부 클래스(Inner Class)

- 클래스 안에서 선언된 클래스를 표현·확인 함
- 내부 클래스를 확인 함

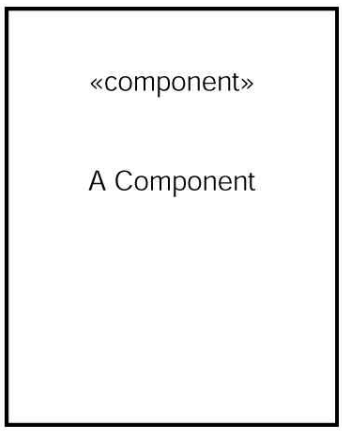


컴포넌트 다이어그램

컴포넌트는 다른 컴포넌트가 접근할 수 있도록 인터페이스를 제공한다. 접근하고 있는 컴포넌트에서는 필수의 인터페이스를 사용한다.

UML 1.X에서는 컴포넌트 아이콘을 사각형의 왼쪽에 작은 사각형 두개를 붙임으로써 나타냈다. UML2.0에서는 상단에 <<componet>>키워드를 가진 사각형으로 나타내며, 오른쪽 상단에 1.X의 컴포넌트 아이콘을 축소하여 포함시키는 것을 추천한다. 인공물(Artifact)의 아이콘은 <<Arifact>>키워드를 상단에 가진 사각형의 모양이다. 오른쪽 상단에는 노트기호를 추가할 수도 있다.

인터페이스를 나타내는 방법에는 두가지가 있다. 첫번째는 인터페이스의 정보를 가지고 있는 사각형을 텅빈 삼각형 머리를 한 점선으로 컴포넌트에 연결하는 방법이고, 두번째는 작은 원을 실선으로 컴포넌트에 연결하는 방법이다. UML2.0에서는 인터페에스가 컴포넌트에 의해 제공되며, 다른 컴포넌트에는 필수적으로 필요하다는 것을 공과 소켓 표기법으로도 표현할수도 있다. 공의 모양은 이미 알고 있듯이 작은 원 모양이고, 소켓은 다른 컴포넌트에 실선으로 연결된 작은 반원(열려있는)모양이다. 공은 제공되는 인터페이스를 의미하고, 소켓은 필수의 인터페이스를 의미한다.

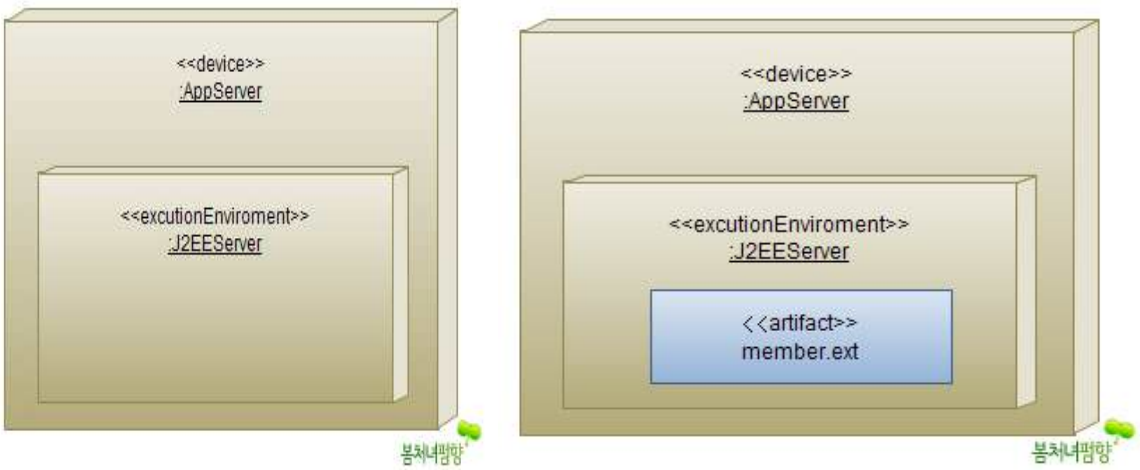


[그림 1.9] 버전 2.0에서의 UML 컴포넌트 다이어그램

배치도

시스템을 구성하는 컴퓨터, 프린터, 모니터 등등 하드웨어의 구성을 표현하는 것이 배치도인이다. 배치도에 그려지는 것들을 노드라고 한다.이 노드는 하드웨어 뿐 아니라 OS 등의 소프트웨어(실행환경)도 표현가능하다. 하드웨어 환경을 그린 노드는 <<device>>를 붙여주고, 실행환경을 그린 노드에는 <<executionEnvironment>>를 붙여준다.

이 실행환경 안에 있는 파일을 표현할 때 직접 파일 이름을 적어주면 된다. 파일은 성과물이라 하여 <<artifact>>라는 스테레오타입을 적어준다.



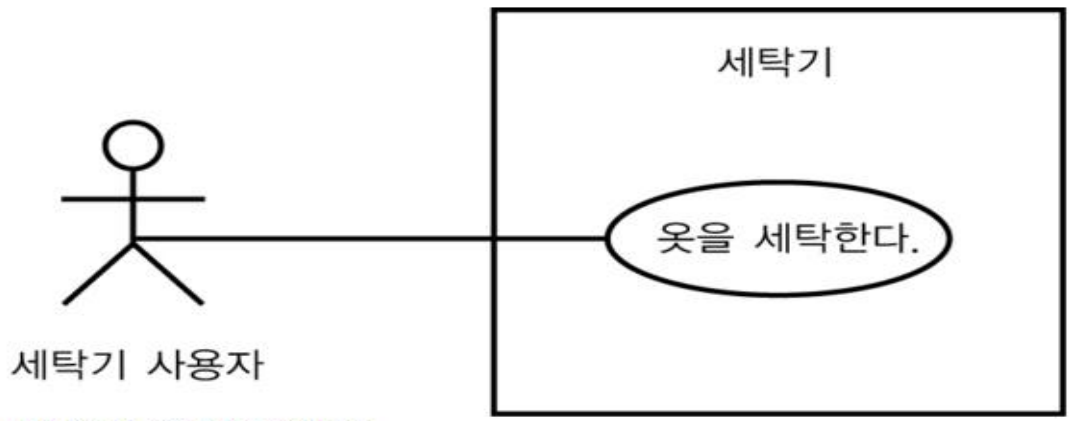
패키지 다이어그램

패키지는 Modeling의 다른 요소(class, interface, component, node, use case)들을 그룹으로 구성하는 범용 메커니즘이다.

다른 package와 구분하기 위한 문자열의 이름을 갖고 있다. class와 마찬가지로 꼬리표 값으로 장식하거나, 상세한 내용을 표현하기 위하여 추가 칸을 사용한다.

Use case 다이어그램

유스 케이스(use case)는 사용자의 입장에서 본 시스템의 행동을 일컫는다. 시스템 개발자에게 이 use case라는 것은 무척 값진 도구이다. 왜냐면, 사용자가 원하는 시스템 사항을 얻어내는데 유용하게 쓰이기 때문이다. 목표가 사람들이 사용할 수 있는 시스템을 만드는 것이라면 더욱 중요하다.



[그림 1.3] UML 유스 케이스 다이어그램

세탁기 사용자를 나타내는 막대 인간 그림을 actor 라고 한다. 타원은 use case를 나타낸다. actor (use case와 대화를 시작하는 개체) - 사람 혹은 다른 시스템

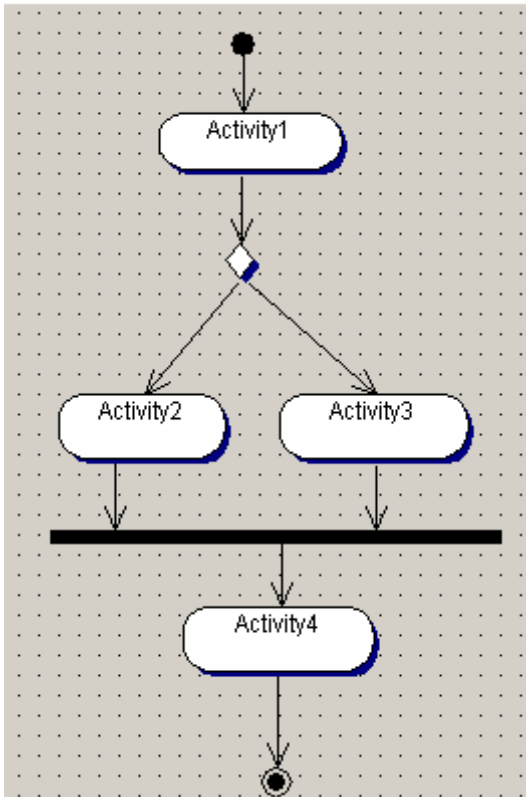
use case가 시스템을 의미하는 사각형 내에 있고, actor는 사각형 바깥에 있다.

Activity 다이어그램

Activity Diagram은 시스템에서 activity와 activity 간의 제어의 흐름을 보여주는 workflow를 나타내는 흐름도이다.

Activity diagram은 순차적인 제어의 흐름 뿐 아니라, 병렬적으로 수행되는 활동과 분기가 이루는 대안들에 대해서도 표현해준다. use case 다이어그램을 작성한 후 하나의 use case 내에서의 흐름을 표현해주는 activity 다이어그램을 작성할 수 있다.

나중에는 특정 operation 내에서의 workflow 표현을 위해 activity 다이어그램을 사용할 것이다. Activity 다이어그램은 activity, transition, decision point와 synchronization bar 등으로 구성된다.



상호작용 개념도

하나의 프로그램은 무엇이 되었든 메시지를 서로 주거니 받거니 하면서 움직이게 된다.

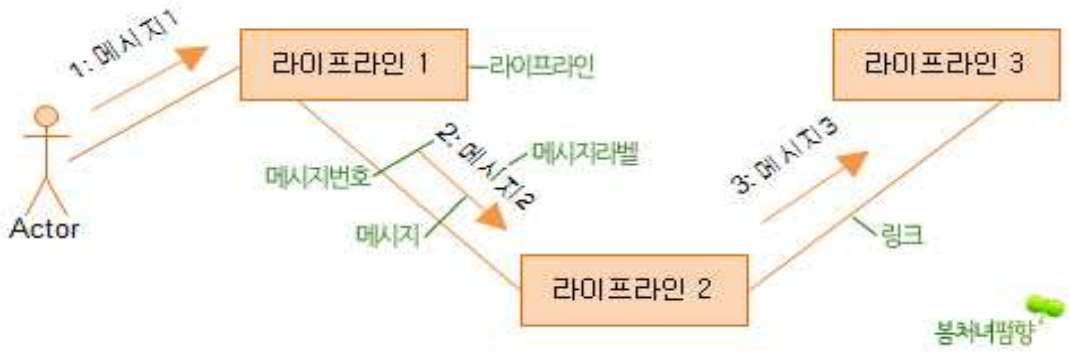
이 메시지의 주고 받음을 표현하는 다이어그램이 상호작용도이다. 프로그래머가 상호작용도를 보고 코딩을 하게 될 때, 메시지를 주고 받는 주체 즉, 클래스의 object를 만들어 그 object의 조작을 불러오면서 코딩하게 된다.

UML의 상호작용도로는 Sequence 다이어그램, Communication 다이어그램이 있다.

Communication 다이어그램

Communication 다이어그램은 sequence 다이어그램과 같이 상호작용을 나타내는 메시지의 흐름을 나타내는 다이어그램이다. 하지만 이 둘의 차이는 sequence 다이어그램은 위에서 아래로 시간에 따라서 표현하는 반면, communication 다이어그램은 상호작용에 참가하는 참가자를 중심으로 표현한다. 따라서, **메시지가 어떤 순서로 흐르는지 번호를 꼭 써줘야 한다.**

Life line들을 갖고 이들 간의 메시지의 흐름을 나타내며, 이 Life line 들의 연결은 메시지가 아닌 링크로 연결을 해준다. 모든 life line을 모두 링크로 연결해준 뒤, 다시 화살표로 메시지를 그려준다. 이때 메시지는 어떻게 움직이는지 메시지 번호가 필수다.



Sequence 다이어그램

직사각형은 각각의 인스턴스를 나타낸다.

:A 와 같이 콜론 뒤에 클래스명을 표기하고 밑줄을 표기합니다.

각각의 A 클래스 인스턴스, B 클래스의 인스턴스, C 클래스의 인스턴스를 나타낸다.

인스턴스에 이름이 필요한 경우 a:A와 같이 콜론앞에 인스턴스 이름을 표기한다.

이 다이어그램은 위쪽은 과거, 아래쪽은 미래로써, 시간은 위에서 아래로 흐른다.

Life line 은 인스턴스가 존재하는 동안만 존재한다.

화살표 -▶ 는 메소드의 호출을 의미하고, 반대로 ◀- 는 메소드의 리턴을 의미한다.

메소트 리턴은 생략이 가능하다.