유스케이스 재구성을 통한 서비스 식별

Services Identification based on Use Case Recomposition

김유경(Yukyong Kim)*

초 록

서비스 지향 아키텍처는 느슨하게 연결되고 상호 호환 가능한 서비스들의 결합을 통해 어플리케이션을 구현하는 기술이다. 서비스는 적절한 입도를 갖는 구현된 비즈니스 함수로 정의할 수 있고, 잘 구성된 인터페이스를 통해 외부에 노출된다. 서비스 모델링 단계에서 서비스의 입도(granularity)가 너무 작아지면, 서비스의 재사용성, 유연성이 낮아진다. 이런 서비스 입도의 문제로 인해 도메인 분석 모델로부터 적절한 추상화 레벨을 갖는 서비스를 식별하고 정의하는 것은 매우 중요한 일이다. 본 논문에서는 도메인 분석 모델인 유스케이스 모델로부터 서비스를 식별하기 위한 절차를 제안한다. 유스케이스와 유스케이스 기술서(description)로부터 태스크 트리를 생성하고, 태스크 트리의 분할과 결합을 통해 유스케이스를 재구성한다. 이렇게 재구성된 유스케이스들로부터 서비스를 식별하고, 명세할 수 있다. 본 논문에서 제안하는 방법은 이미 널리 사용되는 UML 유스케이스 모델을 사용하므로 다양한 플랫폼과 도메인에서 서비스 모델링을 위한 개발 시간과 노력을 최소화 할 수 있을 것으로 기대된다.

ABSTRACT

Service-Oriented Architecture is a style of information systems that enables the creation of applications that are built by combining loosely coupled and interoperable services. A service is an implementation of business functionality with proper granularity and invoked with well-defined interface. In service modeling, when the granularity of a service is finer, the reusability and flexibility of the service is lower. For solving this problem concerns with the service granularity, it is critical to identify and define coarse-grained services from the domain analysis model. In this paper, we define the process to identify services from the Use Case model elicited from domain analysis. A task tree is derived from Use Cases and their descriptions, and Use Cases are reconstructed by the composition and decomposition of the task tree. Reconstructed Use Cases are defined and specified as services. Because our method is based on the widely used UML Use Case models, it can be helpful to minimize time and cost for developing services in various platforms and domains.

키워드 : 서비스지향 아키텍처, 서비스 모델링, 유스케이스 모델, 입도, 재구성
Service-Oriented Architecture, Service Modeling, Usecase Model, Granularity, Recomposition

이 논문은 2005년 교육인적자원부의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(KRF-2005-214-D00342).

* 한양대학교 전자컴퓨터공학부 컴퓨터공학전공 연구교수

1. 서 론

서비스 지향 아키텍처(SOA, Service-Oriented Architecture)는 재사용성, 느슨한 결합, 플랫폼에 독립적인 기술 구현 등의 특징으로 인해 내 외부 환경 변화에 능동적으로 대응할 수 있는 기술로 광범위하게 자리 잡아 가고 있다. 이는 협업의 증가로 인한 이기종 시스템 간의 호환, 고객의 요구에 맞는 다양하고 새로운 상품의 개발 등과 같은 경영 환경의 변화에 따라 생산성과 유연성을 극대화 할 수 있는 새로운 IT 전략 요구를 잘 반영하고 있기 때문이다. 서비스는 일반적으로 소프트웨어로 구현된 비즈니스 함수이며, 잘 구성된 인터페이스를 통해 외부에 노출된다. 또한 여러 개의 서비스들이 상호 연동하여 하나의 상위 수준의 비즈니스 트랜잭션을 형성하게 된다. 이런 서비스들의 조합은 비즈니스 협업을 확장하기 위한 중요한 수단이 되고 있다[1].

서비스 모델링은 서비스 식별(identification), 명세(specification) 및 구현(realization)을 정의하는 과정이다. SOA 기반의 시스템이 효율적이고 성공적으로 자리 잡기 위해서는 올바른 서비스 구성이 필수적이며, 이를 위한 서비스 모델링 과정이 시스템 전체의 운명을 흔들 수 있는 가장 중요한 단계이며 시작점이 된다[2]. 또한 서비스 모델링 단계에서 서비스의 범위를 너무 작게(fine-grained) 잡으면, 서비스의 재사용성, 유연성이 낮아진다. 따라서 이런 서비스 입도(granularity)의 문제로 인해 도메인 분석 모델로부터 적절한 추상화 레벨을 갖는 서비스를 식별하고 정의하는 과정은 매우 중요하다.

본 논문에서는 도메인 분석 모델인 유스케이스 모델로부터 서비스를 식별하기 위한 절차를 제안한다. 유스케이스 모델 재구성을 통해 서비스를 정의할 수 있도록 단계별로 기술한다. 본 논문의 목적은 설계 및 개발 단계에 필요한 적절한 추상화 수준을 갖는 서비스(coarse-grained services)를 정의하기 위해 도메인 모델로부터 서비스를 식별하는 절차를 명시하는 것이다. 제안된 식별 절차는 다양한 플랫폼과 도메인에서 서비스 개발 시간과 노력을 최소화 할 수 있을 것으로 기대된다.

본 논문의 구성은 다음과 같다. 제 2장에서는 SOA 및 유스케이스에 대한 개념을 간단히 소개하고, 서비스 모델링에 대한 기존 연구들을 살펴본다. 제 3장에서는 본 논문에서 제안하고 있는 서비스 모델링 절차에 대해 설명한다. 제 4장은 사례연구의 결과를 보여주고 제 5장에서 결론을 맺는다.

2. 관련 연구

2.1 SOA와 UML 유스케이스 모델

사용자들이 서비스를 제공 받을 때 어떤 제약도 없는 것이 가장 이상적인 분산 컴퓨팅 환경이다. 즉 프로그래밍 언어 및 특정 플랫폼에 상관없이 서비스가 제공되어야 하고, 제공되는 서비스의 유지보수가 용이해야 한다. 이런 조건을 만족시키기 위해서는 표준화된 기술요소가 필요하며, 이로 인해 대두된 개념이 SOA이다. SOA는 서비스 요청자, 서비스 공급자, 그리고 서비스 레지스트리(re-

gistry)의 3가지 역할을 포함하고 있다. 이런 구조를 통해서 SOA는 서비스의 발견과 동적 바인딩이라는 개념을 지원한다[3]. 또한 SOA에서 사용하는 서비스는 컴포넌트에서와 마찬가지로 독립적인 모듈이다. 각각의 서비스는 독립적으로 개발, 유지, 관리되며, 서로의 작동 자체에 큰 영향을 미치지 않는다. 이런 점은 특정 서비스를 수정했을 때 발생하는 파급효과를 최소화 시켜 결합도를 낮추는 중요한 특징이다. 각 서비스는 자신을 호출할 수 있는 인터페이스를 제공하고 있으며, 호출이 이루어지는 프로토콜과 호출 메시지의 포맷만 이해 할 수 있다면, 서로 다른 플랫폼 위에서 개발, 운영되는 서비스끼리도 통신이 가능한 플랫폼에 독립적인 상호운용성을 제공한다.

도메인 모델링은 시스템에서 수행되어야 할 작업들을 전체적인 관점에서 이해할 수 있도록 목표 도메인(target domain)에 관련된 정보들을 상세히 분석하는 과정이다. 수립된 요구사항을 분석하여 목표 도메인에서 시스템과 관련 된 이해관계자를 식별하고, 목표 도메인의 이름, 설명, 비즈니스 영역 등의 정보와 프로세스를 정의하게 된다. 이들 프로세스와 정보들을 정제하여 유스케이스 모델을 작성한다.

유스케이스는 시스템이 사용자에 의해서 어떠한 형태로 사용되는지를 기술하는 UML의 표준 표기법이다. 유스케이스 다이어그램은 유스케이스의 구성요소인 액터와 유스케이스 그리고 그들의 전체적인 관계를 그림의 형태로 보여주기 때문에 이를 통해 시스템의 전체적인 개요와 구성을 쉽게 알 수 있다. 유스케이스 다이어그램에 등장하는 구성요소의 세부적인 내용은 유스케이스 기술서(description)에 작성한다[4].

유스케이스 모델에 정의된 유스케이스들 사이의 관계는 일반화 관계와 연관 관계로서 ≪ *extend* ≫ 및 ≪ *include* ≫가 있다. ≪ *extend* ≫는 베이스 유스케이스(base usecase)가 명시된 제약사항(constraints)을 만족하게 되는 경우, 확장된 유스케이스(extending usecase)에 정의된 행위들이 베이스 유스케이스의 확장 지점(extension point)에 삽입되어 수행되는 관계이다. 확장 지점은 유스케이스 기호(symbol)의 타원 안에 텍스트 문자열로 표현된다. ≪ *include* ≫는 한 유스케이스가 또 다른 유스케이스에 정의된 행위들을 포함하는 관계이다. 확장된 유스케이스가 조건에 의해 선택적으로 실행되는 관계라고 본다면, 포함된 유스케이스(included usecase)는 선택적인 사항이 아니라, 베이스 유스케이스가 정확하게 실행되기 위해 꼭 필요한 유스케이스이다. 즉, 베이스 유스케이스는 포함된 유스케이스의 실행 결과에 따라 실행 흐름이 달라지는 의존적인 관계를 말한다.

2.2 서비스 모델링에 대한 기존 연구

서비스 기반 시스템 구축은 MDA(Model-Driven Architecture) 기반 개발 프로세스를 적용하여 다양한 플랫폼에서도 호환 가능하도록 설계되어야 하며, 웹서비스 기술 등을 적용함으로써 서로 다른 플랫폼의 응용 간에 통신을 가능하게 한다. 즉, MDA 기술과 SOA 및 웹서비스 기술의 통합을 통해 시스템의 도메인별 핵심 컴포넌트를 제공하는 과정이 서비스 모델링 프로세스라고 할 수 있

다[5].

기존의 서비스 모델링 방법은 컴포넌트 기반으로 서비스를 식별하여 정의하였다. 그러나 이런 방법에 따라 식별한 단위 서비스는 입도가 너무 낮다는 문제를 안고 있다[6]. 서비스는 비즈니스 작업 흐름을 반영하는 단위가 되어야 하므로, 프로세스를 반영하는 유스케이스 모델을 기반으로 서비스 모델링이 이루어져야 한다. 서비스 모델링을 위한 기존의 방법들은 대부분 객체지향 방법이나 컴포넌트 기반 방법을 기반으로 하기 때문에 클래스나 컴포넌트를 중심으로 서비스 모델링을 하고 있다. 현재 알려진 서비스 모델링을 지원하는 방법론으로는, Erl이 정의한 SOA 방법론[7], SOUP(Service Oriented Unified Process)[8], IBM의 SOMA(Service-Oriented Modeling and Architecture)[9]가 있다. 국내에서는 삼성 SDS의 서비스지향 개발 방법론[2]이 있다.

Thomas Erl의 SOA 방법론은 서비스 기반의 분석 설계 방법을 제시하면서, 서비스 모델링의 12단계를 포함하고 있다. 이들 절차는 특별히 응용 계층(application layer), 비즈니스 계층(business layer) 그리고 조직화 서비스 계층(orchestration service layer)을 구성하는 SOA의 모델링을 위한 단계로 정의하고 있다. SOUP은 RUP와 XP의 특징만을 모아서 만든 방법론이다. 소프트웨어 개발단계를 여섯 부분으로 나누고, 각 단계에서 수행해야 되는 활동들을 정의한다. SOUP의 프로세스는 SOA 배치(deployment)와 배치 후의 SOA 관리(management)의 두 부분으로 나누어진다. SOA 배치부분은 RUP에 기반을 두고 정의되었고, SOA 관리부분은 XP에 기반을 두고 있다. SOMA는 SOA를 지원하는 모델링, 분석 설계 기술과 활동을 포함하며, SOA이 각 계층 의 요소를 정의하고 있다. 하향식과 기존자산의 비즈니스 중심 상향식 접근법을 통합하는 방법으로서, 서비스 식별, 명세, 실현의 세 단계로 구성된다. 서비스 식별은 비즈니스 도메인을 기능적으로 서브시스템으로 분해하는 하향식과 이미 가지고 있는 시스템의 분석을 통해 비즈니스 프로세스에서 수용 가능한 기능을 가진 후보 서비스를 식별하는 상향식이 결합된 방법을 사용한다. 그리고 상향식이나 하향식으로 식별되어지지 않은 다른 서비스를 찾아내는 목표서비스 모델링 기법을 사용한다.

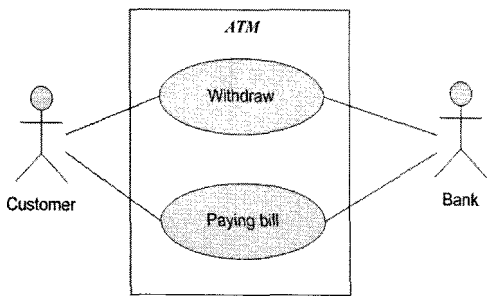
위의 방법론들이 제시한 서비스 모델링 단계의 문제점은 서비스 식별 및 정의 단계가 정확하게 명시되지 않았다는 것이다. 서술적인 가이드라인만으로는 서비스 모델링을 수행하는 과정을 객관적으로 적용하기 어렵고, 식별 방법에 대한 기술적인 지원이 없기 때문에 개인의 경험과 지식에 의존해야 한다는 문제를 안고 있다. 게다가 기존의 클래스 다이어그램과 같은 객체지향 모델을 기반으로 서비스를 추출하고 있다. SOA 지원 개발 방법론의 경우엔 이전에 객체지향 개발이나 컴포넌트 기반 개발과는 다르게 시스템 개발의 단위가 변화되었으며, 따라서 클래스나 컴포넌트보다는 좀 더 추상화된 개념인 서비스 정의에 대한 고려가 이루어져야 한다. 이를 위해서는 클래스가 아닌 유스케이스와 같이 좀 더 입도가 큰 대상으로부터 서비스를 유도하는 것이 적절하다. 본 논문에서는 유스케이스를 재구성하여 서비스를 추출하도록 정의하였다. 또한 접근 방

법에 대한 절차를 명확하게 제시하고자 시도하였다.

3. 유스케이스 모델의 재구성

3.1 태스크 트리

태스크 트리는 유스케이스 시나리오에 나타나는 작업 흐름을 표현하기 위해 사용한다. 각 유스케이스는 하나이상의 시나리오를 제공하며, 이들 시나리오는 시스템이 특정 비즈니스 목표 또는 기능을 수행하기 위해 사용자인 액터(actor)와 어떻게 상호작용해야만 하는지를 알려준다. 태스크는 시나리오를 구성하는 단위가 된다. 태스크 트리와 재구성 규칙을 설명하기 위해 <그림 1>과 같은 간단한 ATM 예제를 사용하기로 한다.



〈그림 1〉 ATM을 위한 유스케이스 모델

유스케이스 시나리오는 일반적으로 형식을 갖추지 않은 자연언어(natural language)로 작성된다. <그림 2>는 유스케이스 'Paying bill'의 시나리오 일부이다.

태스크 트리는 유스케이스의 작업흐름을 기술해 놓은 시나리오(Main Success Scenario)로부터 만들어진다. 태스크는 사용자가 이해할 수 있는 최소의 작업 단위이므로, 시나리오에 기술된 문장을 단문 형태로 구분한다. 주어가 생략되어 있는 문장은 주어를 찾아내어 작업흐름상의 주체가 분명히 나타나도록 기술한다. 이렇게 구분된 문장들에 일련번호를 사용하여 태스크를 정의한다. 서로 연관된 태스크들을 모아서 하나의 서브트리로 구성하고, 태스크들을 잘 표현할 수 있는 레이블로 T1, T2와 같이 붙여준다. 구분된 태스크들은 <그림 3>와 같이 들여쓰기를 통해 트리 형태로 작업의 구분이 쉽게 이루어질 수 있도록 한다. Scope나 Primary actor와 같이 주요 작업 흐름과 관련이 없는 부분은 표현하지 않았다.

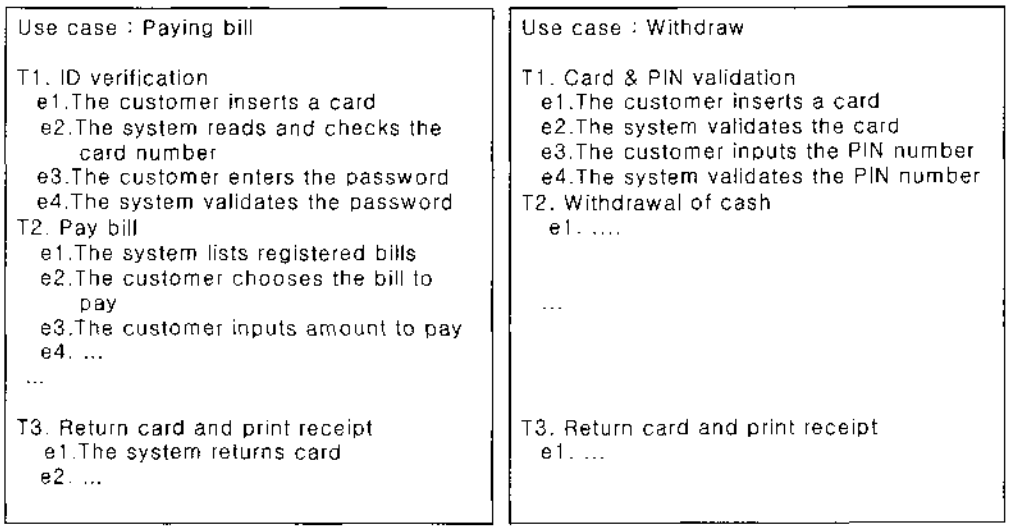
```
Usecase : Paying bill
 Scope : "System" means the Automated
         Teller Machine System
 Level : User goal
 Goal : A User wants to pay the bill
 Primary Actor : Customer, Bank

 Main Success Scenario :
1. The customer inserts a card.
2. The system reads and validates the card.
3. The customer enters the PIN number.
4. The system lists registered bill.
5. The customer choose the bill to pay.
6. The system requires to input the amount
   of the bill.
   ...
```

〈그림 2〉 "Paying bill" 유스케이스

3.2 동등(equivalence) 개념

재구성은 구조적 변환을 의미하며, 최초의 의미(semantics)가 유지(preserved)된다는 것

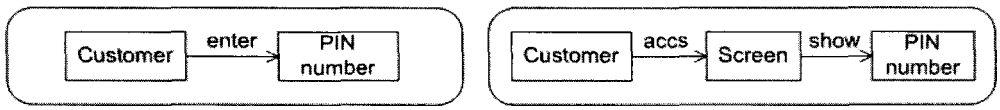


〈그림 3〉 태스크 트리

을 보장해야 한다. 본 논문에서 정의하는 유스케이스 재구성은 유스케이스가 갖고 있는 시나리오를 태스크 단위로 분해하여, 태스크를 재배치하는 과정으로, 태스크가 새롭게 생성되거나 삭제되는 변화 없이 이루어지게 된다. 따라서 유스케이스의 행위에는 변화가 없는 의미가 유지되는 변환이라고 할 수 있다. 유스케이스 재구성은 UML 2.0 스펙에 기술된 유스케이스 모델을 기반으로 하며, 유스케이스가 갖는 행위의 재배치로서 결과적으로 유스케이스들 사이의 관계에 대한 구조적 변경이 수반된다.

재구성이 수행되기 전과 후의 모델이 같은 의미를 가지고 있어야 하므로, 도메인 모델에 대해 변환 전후의 다이어그램이 동등한(equivalent) 관계를 가져야 한다. 일반적인 동등 개념은 두 다이어그램이 같은 의미적 요소를 가지는 경우를 말한다. 그러나 동등 개념은 주변 요소들을 포함하는 경우 유연하게 적용되지 않는다[10]. 예를 들면, <그림 4>에서, 같은 의미를 갖지만 다른 형태로 표현되는 경우를 볼 수 있다. 이때 (b)에 존재하는 또 다른 요소인 'Screen'으로 인해, (a)와 (b)가 동등하다고 정의하기 어렵다. 이들 모델들의 'Customer'와 'Account'의 관계를 고려해 볼 때 직관적으로는 동등하다고 할 수 있다. 그러나 일반적인 동등 개념에서는 서로 다른 구성요소들을 가지고 있기 때문에 동등하다고 정의하지 않는다.

따라서 본 논문에서는 자연어를 사용해 기



(a) (b)

〈그림 4〉 동등 개념

술된 유스케이스 시나리오를 기반을 하므로, 좀 더 유연한 의미의 동등 개념을 사용하고자 한다. 이를 위해, 주요 모델 요소들의 집합 Σ을 정의하고, Σ에 존재하지 않는 모델 요소들은 주변 요소로 간주한다. 예를 들면, 앞의 <그림 4>에서 'Customer'와 'PIN number' 만을 Σ에 포함된다고 가정한다. 그러면 (a)와 (b)가 갖는 요소들에 대해서 Σ 상에서 같은 의미를 갖게 되고, 두 그림을 동등하다고 할 수 있다. Σ 상에 존재하지 않는 'Screen'이나 'accs', 'show' 등과 같은 요소들은 주변 요소로 간주한다. 또 다른 경우로, Σ에 존재하지만 양쪽에 모두 존재하지 않아 비교할 수 없는 요소들이 있다. 앞의 예에서 Σ에 'enter'가 포함되어 있다고 하자. <그림 4>의 (a)에는 'enter'가 존재하지만, (b)에는 존재하지 않기 때문에 두 그림을 비교할 수 없게 된다. 그렇지만 실제로는 'accs'와 'show'를 결합하여 'enter'로 표현할 수 있다.

따라서 재구성이 이루어지는 동안 다른 요소에 의해 대치될 수 있는 상황이 발생할 가능성이 존재하므로, 이를 위해 대응 함수를 고려하고자 한다. 한 모델에 존재하는 하나의 요소를 다른 모델에 있는 요소들로 어떻게 대응시키는지를 보여주는 함수 $f(v)$를 다음과 같이 정의한다.

정의 1 : 대응 함수 $f(v)$는 요소들의 이름 v와 표현식 exp에 대해 다음과 같이 정의된다.

$$f(v) : v \rightarrow exp$$

예를 들면, $\bullet$을 두 항목의 조합이라고 한다면, $enter \rightarrow accs \bullet show$와 같은 항목을 포함하는 함수를 정의할 수 있다. 'enter'가 (b)에 나타난 두 주변 요소의 조합으로 정의된 것이다.

3.3 재구성 규칙

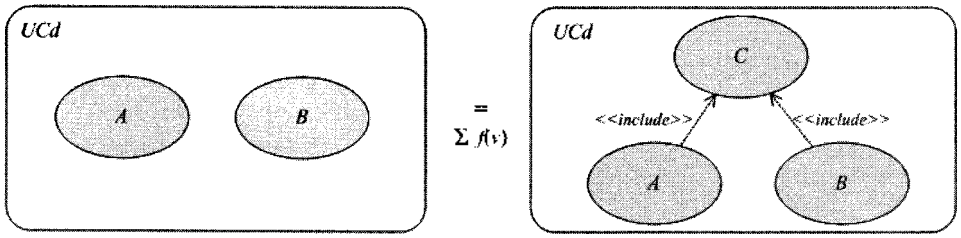
앞에서 정의한 동등 개념을 사용하여 유스케이스 모델의 재구성 규칙을 정의한다. 재구성을 위해 모든 유스케이스의 모델 요소들은 Σ에 선언되어 있고, 재구성 후에 존재하는 모든 모델 요소들도 정의 1에 의해 Σ에 선언되어 있다고 가정하자. 또한 UML 유스케이스 다이어그램은 같은 이름을 갖는 서로 다른 유스케이스들을 포함할 수 없다고 가정한다. 규칙 표현에 사용된 UCd는 모델을 구성하는 모든 유스케이스, 유스케이스들 사이의 관계를 포함하는 것으로 정의한다.

규칙 1. 분할(decomposition)

한 유스케이스가 다른 유스케이스에 비해 입도가 큰 경우, 여러 개의 유스케이스로 분할할 수 있다. 이러한 분할을 통해 유스케이스들 사이의 유사한 추상화 수준을 유지할 수 있다. 분할은 유스케이스들 사이의 태스크들이 공통성이 있는지 없는지에 따라 2가지 경우로 구분해 볼 수 있다.

규칙 1.1 일반화(generalization)

일반화는 유스케이스들이 갖고 있는 공통의 태스크들을 모아서, 새로운 유스케이스로 생성시키는 경우이다. 각 유스케이스에서 공통으로 수행되는 태스크들은 선택적으로 수행되는 것이 아니므로, 새로운 유스케이스는 $\ll include \gg$관계로 연결된다. 다음은 OCL (Object Constraint Language)로 기술된 일



〈그림 5〉 일반화에 의한 재구성

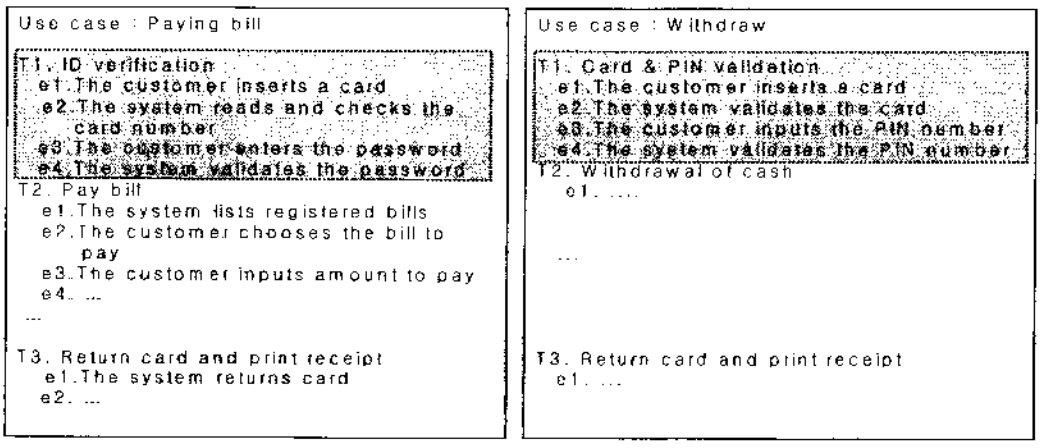
〈표 1〉 일반화 분할 규칙에 대한 OCL 식

```
Context UCd
    def : usecases : Set(UseCase) = self.ownedClassifier  >
        select(oclTypeOf(UseCase)) -> collect(c|c.oclAsType(UseCase))

Context UCd :: gen_decomposition(A, B : UseCase)
    pre : usecases  > exists(self.name = A) and
        usecases -> exists(self.name = B)
        def : TA : Set(Task) = TA.allInstances()  >
            collect(t|A -> intersection(B))
        not TA -> isEmpty()
    post : let C : UseCase = usecases -> self.oclIsNew().name = C in
        C -> oclAsType(UseCase) -> collect(TA.allInstances)
        A -> collect(A - TA) and B -> collect(B - TA)
```

반화 분할 규칙이다.

예제 <그림 3>에서, Σ = {customer, system, ID, card, password, bill}이라고 하자. 그러면 대응함수 $f(v)$에 의해, $id \rightarrow card \cdot PIN$

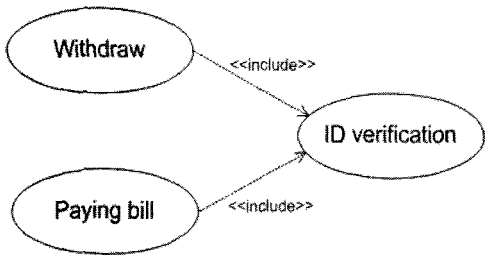


(a) (b)

〈그림 6〉 공통 태스크

number 또는 *password*→*PIN number*를 정의하면, <그림 6>(a)와 <그림 6>(b)에서 점선으로 구분된 태스크들은 동등하다고 할 수 있다.

따라서 이들을 공통의 태스크로 추출하여, 새로운 유스케이스 'ID verification'을 정의하게 된다. <그림 1>의 유스케이스 모델에 대해, 일반화 규칙을 이용해 재구성한 경우가 아래 <그림 7>이다.



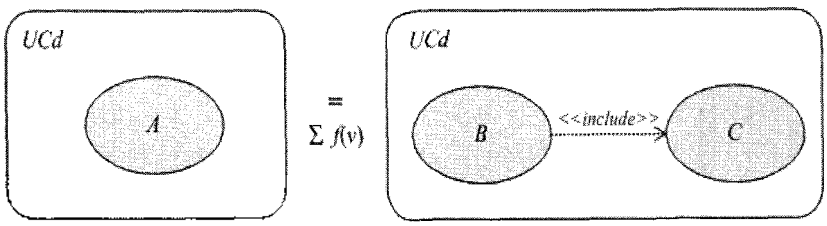
〈그림 7〉 일반화에 의한 재구성 예제

규칙 1.2 독립적 분할

한 유스케이스가 나머지 다른 유스케이스들에 비해 높은 입도를 갖는다면, 추상화 수준을 맞추기 위해 두 개 이상의 입도가 낮은 유스케이스로 분할 할 수 있다. 하나의 유스케이스를 새로운 여러 개 유스케이스로 생성시키는 경우이다. 각 유스케이스들 사이에는 태스크 흐름에 맞추어 ≪*include*≫관계로 연결된다.

규칙 2. 결합(composition)

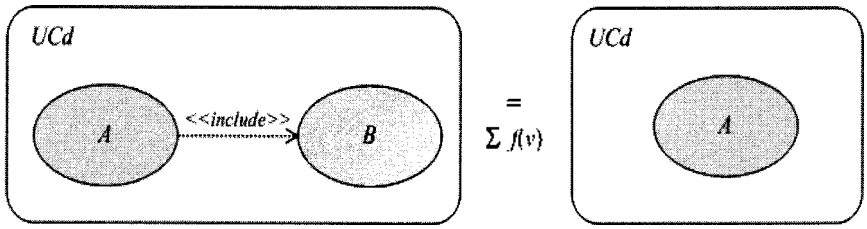
한 유스케이스가 다른 유스케이스에 비해 입도가 낮다면, 다른 유스케이스에 포함시키는 것이 더 좋은 방법이다. 결합을 통해 유스케이스들 사이의 유사한 추상화 수준을 유지할 수 있을 것이다. 결합은 유스케이스들 사이의 관계에 따라 3가지 경우로 구분해 볼



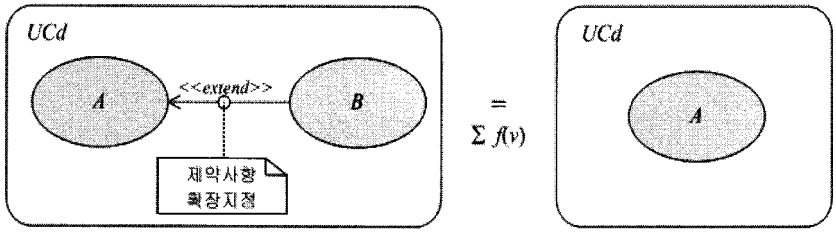
〈그림 8〉 독립적 분할에 의한 재구성

〈표 2〉 독립적 분할 규칙에 대한 OCL 식

```
Context UCd :: inde_decomposition(A : UseCase)
    def : P, Q : Set(Classifier) = P.notEmpty and Q.notEmpty
    pre : usecases -> exists(self.name = A) and
          A.allInstances -> include(P -> union(Q))and
          (P -> intersection(Q)) -> isEmpty()
    post : usecases -> self.oclIsNew().name = B and
          usecases -> self.oclIsNew().name = C
          self.oclAsType(UseCase) -> collect(P.allInstances) and
          self.oclAsType(UseCase)  > collect(Q.allInstances)
```



〈그림 9〉 ≪ *include* ≫관계의 결합



〈그림 10〉 ≪ *extend* ≫관계의 결합

수 있다.

규칙 2.1 연관 관계의 결합

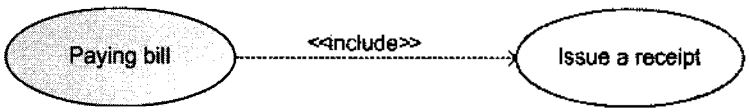
두 유스케이스가 ≪ *include* ≫또는 ≪ *extend* ≫ 연결되어 있고, 유스케이스의 입도가 작으면, 유스케이스를 베이스 유스케이스와 결합시켜 하나의 유스케이스로 재구성한다.

<그림 11>과 같은 관계를 갖는 유스케이스들이 존재한다고 하자.

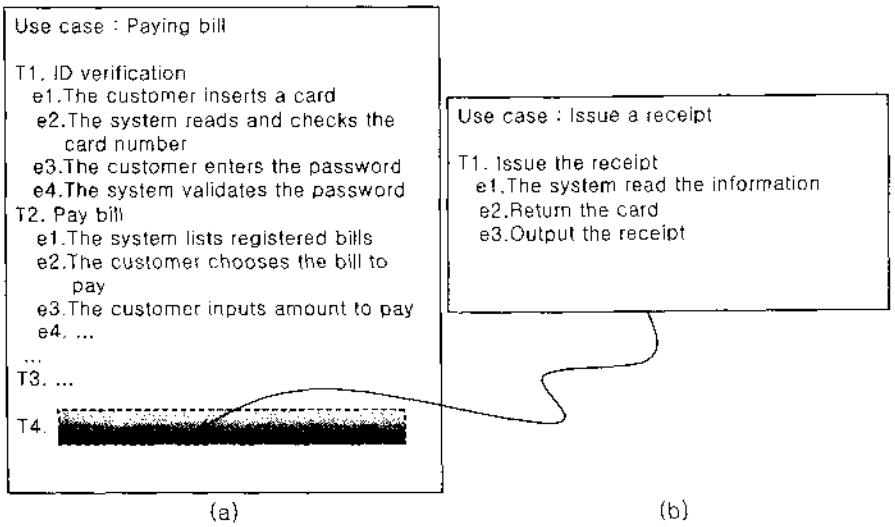
이들에 대한 시나리오가 <그림 12>와 같

〈표 3〉 연관 관계 결합 규칙에 대한 OCL 식

```
Context UCd :: re_composition(A, B : UseCase)
    let connection(u1 : UseCase) : Boolean =
        u1.connection -> exists(as : UseCase | u1.hasInclude(as) or
        u1.hasExtend(as))
    pre : usecases -> exists(self.name = A) and
        usecases -> exists(self.name = B)
        A -> connection(B)
    post : A = self.oclAsType(UseCase)
                    -> asSet(self) -> union(asSet(B))
```



〈그림 11〉 ≪ *include* ≫관계의 유스케이스

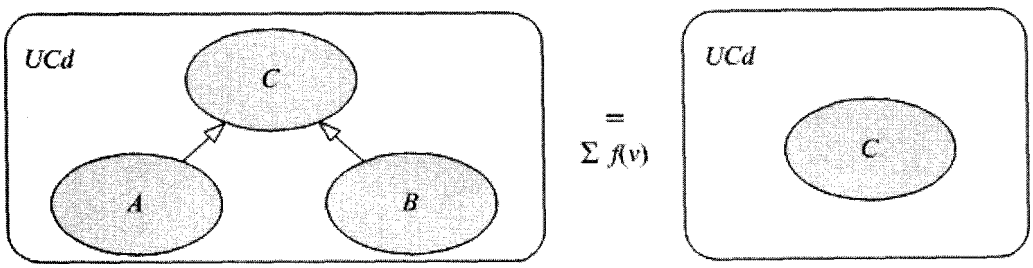


〈그림 12〉 유스케이스 결합 예제

다고 한다면, 유스케이스 'Issue a receipt'의 경우 하나의 출력 함수(function)만으로 구현이 가능하므로 입도가 너무 작은 경우에 해당한다. 따라서 베이스 유스케이스인 'Paying bill'과 결합시킬 수 있다. 결합 과정은 <그림 12>에서와 같이, 베이스 유스케이스의 작업 흐름에 맞추어, 포함된 유스케이스를 삽입할 수 있다. ≪*extend*≫관계에 있는 유스케이스들의 결합은 확장지점을 베이스 유스케이스가 가지고 있으므로, 베이스 유스케이스의 작업 흐름을 고려하지 않아도 된다.

규칙 2.2 상속 관계로 정의된 유스케이스들의 결합

UML이 객체지향 모델링 언어이기 때문에, 유스케이스들 사이의 상속 개념을 포함하고 있다. 그러나 서비스들은 느슨하게 연결된(loosely coupled) 구조를 가지고 있으므로, 상속과 같이 강하게 결합된(tightly coupled) 구조적 개념을 지원하지 않는다. 따라서 유스케이스들 사이의 상속 관계는 서비스 기반 개발에 알맞은 구조로 재구성되어야 한다. 상속은 재사용을 위한 개념이므로, 상속 계층 구조상에 존재하는 유스케이스들의 입도는 다



〈그림 13〉 상속 관계를 갖는 경우의 결합

른 유스케이스들에 비해 작은 경우가 대부분이다. 따라서 상속 관계의 유스케이스들은 결합에 의해 일단 하나의 유스케이스로 재구성되어야 한다.

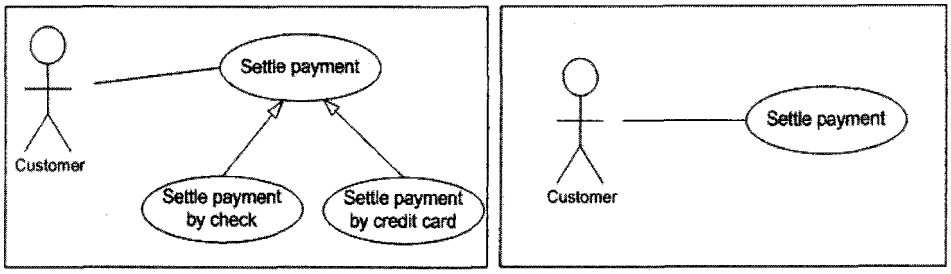
<그림 14>(a)에 존재하는 두 유스케이스 'Settle payment by check'와 'Settle payment by credit card'는 상위 유스케이스인 'Settle payment'와 상속 관계를 갖고 있다. 이들은 <그림 14>(b)와 같이 결합에 의해 재구성할 수 있다. 상속 관계의 결합은 태스크 트리 상에서 <그림 12>와 같이 한 태스크의 하위 트리를 구성하게 되는 형태가 된다.

규칙 2.3 독립적으로 정의된 유스케이스들의 결합

유스케이스들의 입도가 다른 유스케이스들

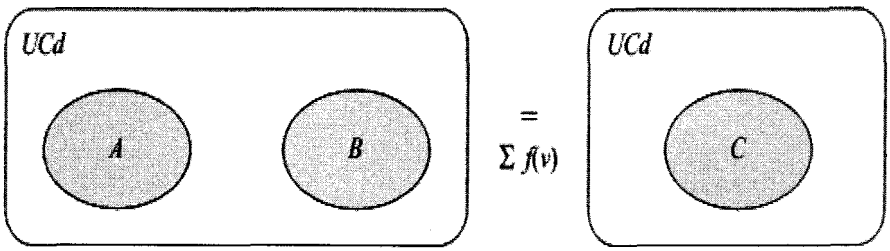
〈표 4〉 상속 관계 결합 규칙에 대한 OCL 식

```
Context UCd :: inh_composition(A, B, C : UseCase)
    let childUC : UseCase = self.child.oclAsType(UseCase)
    pre : usecases  > exists(self.name = A) and
          usecases -> exists(self.name = B) and
          usecases -> exists(self.name = C)
          not C.childUC -> size() = 0
    post : C = self.oclAsType(UseCase) -> asSet(self) ->
                          union(asSet(A) -> union(asSet(B)))
```



(a) 결합 전 (b) 결합 후

〈그림 14〉 상속 관계의 결합



〈그림 15〉 독립 유스케이스들의 결합에 의한 재구성

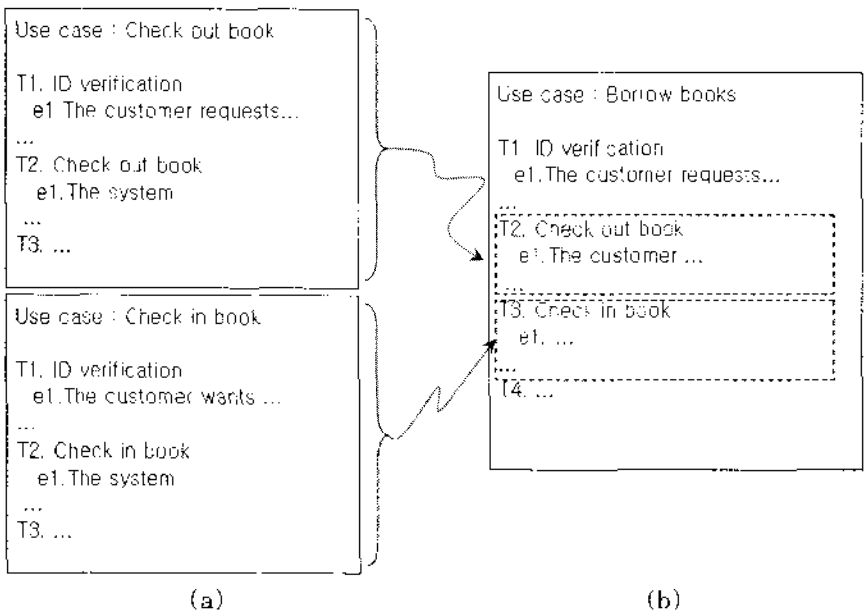
보다 작을 때, 서로 독립적으로 정의되었지만, 연관시킬 수 있는 유스케이스들은 하나의 유스케이스로 결합시킨다.

독립적으로 정의된 각 유스케이스의 태스크 트리들은 <그림 16>과 같이 작업흐름의 순서를 고려하여, 결합시켜준다. 즉, <그림 16>(a)에 있는 두 유스케이스들이 그 유스케이스 다이어그램 내에 존재하는 다른 유스케이스들보다 입도가 작다면, <그림 16>(b)와 같이 결합에 의해 재구성할 수 있다.

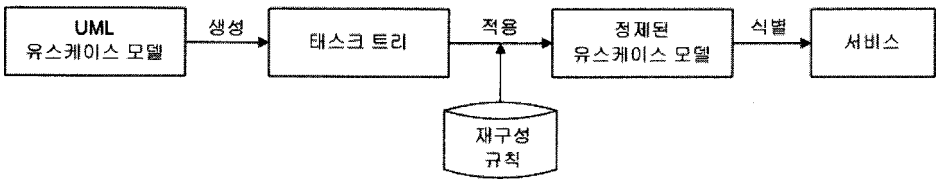
유스케이스 재구성은 서비스로 정의될 유스케이스들이 갖는 입도의 수준을 맞추어주기 위한 작업이다. 서비스 입도가 작은 서비스들은 재사용성이 낮고 서비스가 제공할 수 있는 기능의 유연성이 줄어들게 된다. 또한, 입도가 작은 서비스들은 시비스 제공자와 서비스 사용자 사이의 많은 네트워크 트래픽을 유발하게 되므로, 결합과 같은 재구성을 통해 서비스들이 적절한 입도를 갖추도록 하는 것이 필요하다.

〈표 5〉 독립 결합 규칙에 대한 OCL 식

```
Context UCd :: inde_composition(A : UseCase)
    pre : usecases  > exists(self.name = A) and
          usecases -> exists(self.name = B)
          not A -> connection(B)
    post : usecases -> self.oclIsNew().name = C and
          C = self.oclAsType(UseCase) -> asSet(self) ->
                              union(asSet(A)  > union(asSet(B)))
```



〈그림 16〉 독립된 유스케이스들의 결합



〈그림 17〉 서비스 식별 절차의 블록 다이어그램

3.4 서비스 식별 절차

서비스의 관점에서 보면, 시스템은 서비스들과 서비스들 사이의 상호작용의 집합으로 정의할 수 있다. 서비스는 여러 시스템에서 재사용할 수 있고, 따라서 여러 컴퓨팅 시스템들에 의해 구현되고 실행된다. 서비스는 객체나 컴포넌트보다 입도가 큰(coarse) 단위이

```
<?xml version="1.0" encoding="UTF-8"?>                    서비스 BorrowBooks에 대한정의
<definitions name="BorrowBooks"
                  targetNamespace="http://www.ecerami.com/wsdl/BorrowBooks.wsdl"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:tns="http://www.ecerami.com/wsdl/BorrowBooks.wsdl"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <message name="IDverificationRequest">
     <part name="ID" type="xsd:string"/>
 </message>
 <message name="IDverificationResponse">
     <part name="status" type="xsd:boolean"/>                          각 연산에 대한
 </message>                                                            입/출력 매개변수
     .....
 <portType name="BorrowBooks_PortType">
     <operation name="IDverification">
            <input message="tns:IDverificationRequest"/>
            <output message="tns:IDverificationResponse"/>
     </operation>
     <operation name="CheckOut">
             <input message="tns:..."/>
             <output message="tns:..."/>
     </operation>
     <operation name="CheckIn">
             <input message="tns:..."/>
             <output message="tns:..."/>
     </operation>
 </portType>
 <binding name="BorrowBooks_Binding" type="tns:BorrowBooks_PortType">
     <soap:binding style="rpc" transport="http://schemas...../>
     <operation name="IDverification">
              <soap:operation soapAction=""/>
                     <input>
                       <soap:body
                          encodingStyle="http://....."
                          namespace="urn:examples:borrowbooks"
                          use="encoded"/>                              각 연산에 대한
                     </input>                                          SOAP메시지정의
                     <output>
                        <soap:body
                           encodingStyle="http://....."
                           namespace="urn:examples:borrowbooks"
                           use="encoded"/>
                     </output>
      </operation>
          .....
 </binding>
 <service name="BorrowBooks">
      <documentation>WSDL file for borrow books service</documentation>
      <port binding="tns:BorrowBooks_Binding" name="BorrowBooks_PortType">
            <soap:address
                     location="http://localhost:8080/soap/.../>
       </port>
 </service>
</definitions>
```

〈그림 18〉 WSDL 예제

기 때문에, 비즈니스 프로세스와 시스템의 기능(functionality)을 잘 표현하고 있는 유스케이스에서 서비스 모델링을 시작하게 되는 것이 더 좋은 식별 방법이 된다[11]. 본 논문에서는 <그림 17>과 같은 서비스 식별 절차를 제안한다.

우선 UML 유스케이스 모델로부터 태스크 트리를 생성한다. UML 유스케이스 모델은 유스케이스 다이어그램과 유스케이스 기술서(description)으로 이루어져있다. 유스케이스 기술서는 사용자가 이해하는 최소의 작업단위인 태스크로 분할될 수 있다. 즉, 각 태스크는 유스케이스가 수행하는 시나리오의 부분으로서, 일련의 상호작용인 다이얼로그(dialogues)를 표현한다. 태스크 트리는 유스케이스 시나리오를 태스크 단위로 분할시킨 것이다. 생성된 태스크 트리에 재구성 규칙을 반복적으로 적용하여, 재구성된 유스케이스 모델을 만든다. 재구성된 유스케이스 모델을 기반으로, 하나의 유스케이스는 하나의 서비스로 식별한다. 각 유스케이스에 대한 태스크 트리를 이용하면, WSDL 서비스를 명세를 생성할 수 있다. <그림 18>은 <그림 16>(b)에 있는 유스케이스 "Borrow books"에 대한 WSDL 명세이다.

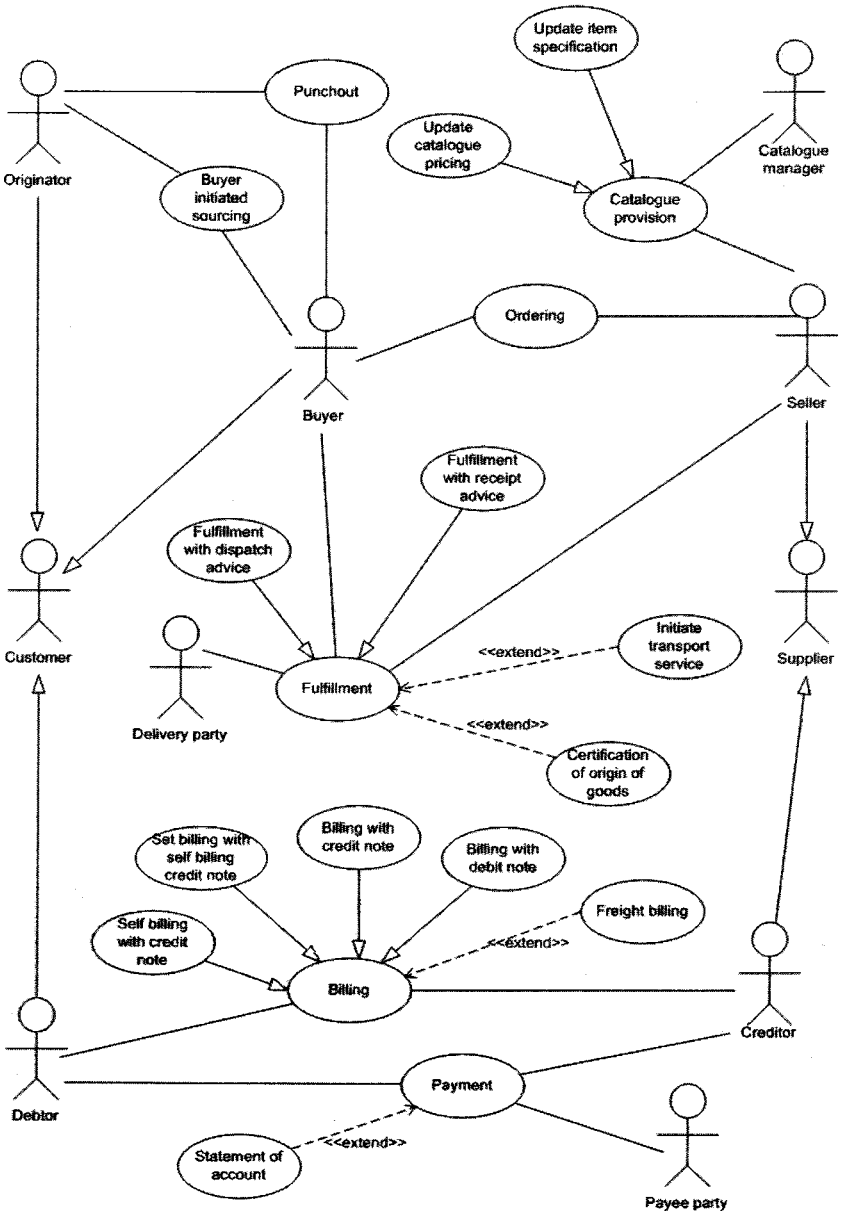
식별된 서비스들은 서비스가 갖는 연산의 개수나 다른 서비스와의 메시지 호출 관계 등을 평가하여, 최적화하는 과정을 거쳐야 할 것이다. 이런 정제 과정은 각 서비스의 역할과 기능을 사용자에게 좀 더 분명하게 하기 위해 필요하다. 서비스들 사이의 상호작용을 분석함으로써 서비스들 사이의 우선순위를 분명하게 하는 것이다. 필요하다면, 기존의 서비스들을 수정하거나 제거할 수 있고, 새로운 서비스를 생성 할 수도 있을 것이다. 이런 반복적인 정제 과정을 거쳐 최종적인 서비스들의 리스트가 생성된다.

4. 사례연구

본 논문에서 제시한 방법을 평가하기 위하여, 간단한 송장발부를 포함한 주문처리 시스템(Order-to-invoice business system)에 적용해 보았다. <그림 19>는 주문처리 시스템의 유스케이스 다이어그램이다. 액터들 사이의 관계는 본 논문의 범위와 상관이 없으므로, 재구성하지 않았다.

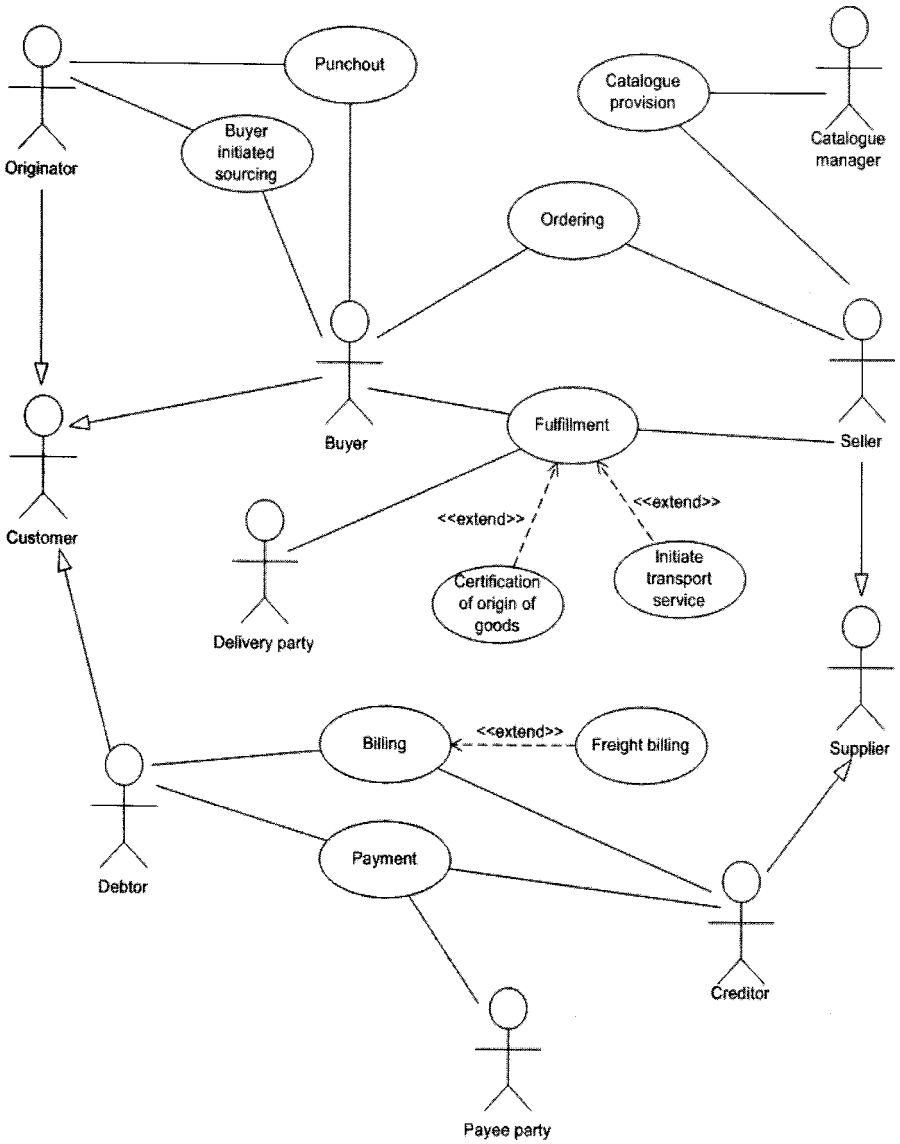
먼저 "Catalogue provision," "Fulfillment" 그리고 "Billing"이 가지고 있는 상속 관계를 제거한다. 규칙 2.2의 상속 관계에 있는 유스케이스들 사이의 결합에 의해, 상속 관계를 가지고 있는 세 개의 유스케이스들 사이의 상속 계층 구조가 제거되었다. 또한 유스케이스 "Payment"와 "Statement of account"는 규칙 2.1에 의해 ≪*extend*≫관계에 있는 두 유스케이스를 결합시켰다. 이때, 새로운 유스케이스의 이름보다 "Payment"의 이름을 그대로 유지하는 것이 유스케이스의 기능을 보다 잘 표현할 수 있으므로, 그대로 명명하였다. 나머지 유스케이스들 사이의 ≪*extend*≫관계는 입도의 수준을 맞추기 위해 그대로 관계를 유지시켰다. 식별된 서비스들의 아키텍처 및 WSDL을 이용한 명세 작성은 제외하고, 순수하게 유스케이스를 재구성해서 서비스를 유도해 내고 정제하는 과정까지를 수행하였다. <그림 20>은 제시된 절차에 따라 재구성한 유스케이스 다이어그램이다.

재구성 과정에서 가장 문제가 되었던 부분은 유스케이스들 사이의 입도를 비교하는 것이었다. 유스케이스에 대한 입도를 평가하기 위한 지침이 마련되지 않았기 때문에, 추상화 수준을 맞추기 위해 분할과 결합을 결정하기 위한 전략이 필요했다. 본 사례 연구에서의 분할과 결합에 대한 의사 결정은 유스케이스 시나리오의 태스크 수를 비교하는 것으로 이



〈그림 19〉 유스케이스 다이어그램

루어졌다. 태스크들이 비교적 간결한 단문 형태로 기술이 되어 있었으므로, 태스크의 수를 비교하여 태스크가 다른 유스케이스에 비해 10개 이상 많은 경우 분할을, 반대로 10개 이상 적다면, 결합을 하도록 결정하였다. 그러나 재시된 재구성 규칙을 적용하기 위해서는 보다 객관적인 입도를 평가할 수 있는 방법이 필요하며, 입도의 평가 결과를 통해 선택적으로 결합과 분할을 선택할 수 있는 방법이 요구된다.



〈그림 20〉 재구성된 유스케이스 다이어그램

5. 결론 및 향후 연구과제

서비스 기반 개발 생명주기(service-oriented development lifecycle)를 놓고 보면, 대부분의 프로젝트들은 모델링 단계에서 요구사항들로부터 서비스 컴포넌트를 정의하는 문제에서 어려움에 직면하게 된다. 이것은 모델링 단계에서 클래스나 컴포넌트를 기반으로 서비스 컴포넌트를 정의함으로써 야기되는 문제이다. 이는 서비스가 객체나 컴포넌트와는 다른 입도를 가지고 있고, 비즈니스 프로세스들을 반영해야 하는 특징을 가지기 때문이다. 따라서 서비스의 개념에 적절한 체계적인 접근방법이 필요하다. 본 논문에서는 이 문제를 해결하기 위해 유스케이스 모델을 기반으로 재구성 개념을 적용하였다. 유스케이스 모델은 비즈니스 프로세스를 잘 반영하는 도메인 모델이다[12]. 따라서 유스케이스 모델로부터 서비스를 생성하면, 객체모델이나 컴포넌트 모델로부터 서비스를 식별하는 것보다 훨씬 더 빨리 서비스 생성이 가능해진다. 또한 서비스의 입도 문제는 서비스 기반 시스템 개발에서 매우 중요한 부분으로, 유스케이스 모델을 사용하여 보다 추상화된 서비스들을 정의할 수 있다.

또한 제안된 서비스 식별 방법은 유스케이스 모델 기반으로, 여러 유형의 문제에 대한 정제된 관점을 표현할 수 있는 다양한 모델들과 연결되어 정의될 수 있으므로, 다양한 플랫폼과 도메인에 적용하여 서비스 개발 시간과 노력을 최소화 할 수 있을 것이다. 이는 MDA 개념을 서비스 기반 개발에 접목할 수 있는 기반을 제공하는 것이다.

유스케이스 재구성은 서비스 입도를 조절할 수 있고, 서비스들 사이에 유사한 추상화 수준을 유지할 수 있도록 도와준다. 본 논문에서는 유스케이스 재구성의 개념을 도입하여, 서비스 모델링의 중요한 이슈가 되는 서비스 식별의 문제를 해결하고자 하였다. 유스케이스 명세서의 비형식화 문제를 트리 구조를 사용한 태스크 모델을 작성함으로써 해소하려고 시도하였다.

그러나 사례 연구를 통해 제시된 방법을 적용하는 과정에서, 유스케이스 모델로부터 태스크 모델을 작성하는 가이드라인과 유스케이스 재구성 전략을 결정하기 위한 구체적인 입도의 평가 방법이 부족하다. 또한 태스크 트리를 형식화하거나, 유스케이스 명세서를 분석하기 위해 그래프를 사용하는 등의 보완이 이루어져야 한다. 태스크의 수가 많아지고, 유스케이스의 개수가 많아진다면, 일일이 태스크트리를 분석하기에는 시간이 많이 필요한 작업이 된다. 따라서 보다 많은 복잡한 경우에 대해 적용될 수 있도록 유스케이스 명세서의 표현방법 및 의미적 구성을 통한 자동화가 이루어져야 할 것이다. 이와 함께 더 많은 재구성 규칙들이 정의되어야 하며, 좀 더 규모가 큰 시스템에 적용한 사례연구를 통하여, 제시된 모델링 프로세스의 실질적인 유효성 검증이 이루어져야 할 것이다.

참 고 문 헌

[1] 김동수, 배혜림, "협업적 웹서비스 표준과 기업간 협업", 한국정보과학회지, 제22권, 제10호, 2004, pp. 26-31.

[2] 한상우, 박선희, 노재호, "Service-Oriented Architecture 적용을 위한 서비스 식별 기법", 한국정보과학회지, 제24권, 제11호, 2006, pp. 27-31.

[3] 전자상거래 표준화통합 포럼 W3C 한국사무국, W3C 차세대 웹기술백서, 2004.

[4] OMG, "OMG Unified Modeling Language Specification ver. 2.0", part 10 : Activities, 2004.

[5] 김행곤, "e-비즈니스 응용 시스템을 위한 컴포넌트 개발에 관한 연구", 한국정보처리학회논문지, 제11-D권, 제5호, 2004, pp. 1095-1105.

[6] Kim, Y., Doh, K. G., "The Service Modeling Process based on Use Case Refactoring," Lecture Notes in Computer Science, Vol. 4439, 2007, pp. 108-120.

[7] Thomas Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice-Hall, NJ, 2005, pp. 398-422,

[8] Kunal Mittal, "Service Oriented Unified Process(SOUP)," IBM Journal, 2005.

[9] Ali Arsanjani, "Service-Oriented Modeling and Architecture : How to identify, specify, and realize services for your SOA," IBM developerWorks, 2004.

[10] Tiago M., Rohit G., and Paulo B., "Formal Refactoring for UML Class Diagrams," OOPSLA 2005, ACM press, 2005, pp. 208-209.

[11] 김유경, 윤홍란, "SOA를 위한 서비스지향 개발 프로세스", 한국전자거래학회지, 제12권, 제2호, 2007, pp. 75-93.

[12] 유철중, 정소영, "요구사항 기술서로부터 유스케이스 다이어그램의 추출기법", 한국정보처리학회 논문지, 제9-D권, 제4호, 2002, pp. 639-650.

저 자 소 개

김유경 (E-mail : yukyong@hanyang.ac.kr)
숙명여자대학교 졸업
숙명여자대학교 컴퓨터과학과 (박사)
University of California Davis, Post-Doc.
현재 한양대학교 전자컴퓨터공학부 컴퓨터공학전공 연구교수
관심분야 웹서비스, SOA, MDA, S/W 품질평가, 시맨틱웹