안드로이드 기반 모바일 서비스 어플리케이션의 아키텍처

숭실대학교 ▌김수동* · 라현정**

1. 서 론

최근 들어 iPhone이나 안드로이드 장비와 같은 모바일 장비의 활용이 본격화 되고 있다. 모바일 디바이스는 기본적으로 휴대폰으로서의 전화 기능을 제공함은 물론, 다양한 소프트웨어 어플리케이션을 설치 운영할 수 있는 컴퓨터의 기능도 제공한다. 따라서, 모바일 디바이스는 퍼스널 컴퓨팅 용도 뿐만 아니라, 기업의 엔터프라이즈 어플리케이션의 클라이언트 단말로도 사용될 것으로 예상된다. 또한, 모바일 디바이스는 이동성(Mobility)을 제공하고, 위치 파악, 가속 측정등 주변 상황을 인지하는 기능도 제공하므로, 다양한 어플리케이션 개발과 응용을 가능케 한다.

그러나, 모바일 디바이스는 물리적인 작은 크기로 인하여, 컴퓨팅 파워와 메모리, 화면 크기, 배터리 수명 등의 자원(Resource) 측면에서 제약점이 있다. 따라서, 복잡한 기능의 어플리케이션은 이 디바이스 상에서 설치와 운영이 어렵다.

이러한 단점을 극복하고 모바일 디바이스의 활용을 최대화하기 위해 서비스 기반의 모바일 어플리케이션(Service-based Mobile Application, SMA)이 각광을 받고 있다[1]. SMA는 일부 혹은 상당 부분의 기능이 서비스의 형태로 구현되어 서버 측에서 설치 운영되고, 사용자는 모바일 디바이스에 설치된 클라이언트 어플리케이션은 이 서비스를 호출하여 사용한다. 이렇게 두 개의 노드에서 목표 어플리케이션을 실행함으로 해서, 모바일 디바이스가 가지는 자원 제약성의 문제는 상당히 해소된다. SMA에서 사용되는 서비스는 Software-as-a-Service(SaaS)와 Component-as-a-Service(CaaS)의 두 가지 형태가 모두 가능하다[2].

SMA의 아키텍처는 두개의 노드에 서로 협력(Collaboration)하는 모델-뷰-컨트롤(Model-View-Control, MVC) 방식의 두개의 프로그램으로 구성된다. 이런 아키텍처의 설계는 두개의 노드에 기능분할, 복제된 객체들의 상태의 일관성 유지, 네트웍상의 원격 서비스 호출 등을 고려하여 설계해야 하므로, 전통적인 MVC 설계 기법으로는 충분하지 않다.

본 논문에서는 이러한 SMA의 아키텍처를 효과적으로 체계적으로 설계하기 위한 핵심 개념, 구성 및 설계 지침을 제안 설명한다. 즉, 소프트웨어 아키텍처 설계의 기본 원칙을 준수하면서, 모바일 어플리케이션의 고유 특징을 고려한 SMA 아키텍처를 설명한다. 제안된 아키텍처는 성능, 확장성, 가용성, 자원 활용률 등 다양한 품질 요구사항을 만족한다.

논문의 구성은 3장에서 모바일 디바이스의 특징을 설명하며, 4장에서 독립형 모바일 어플리케이션과 SMA 아키텍처를 비교한다. 5장에서는 SMA에 사용되는 Balanced MVC 아키텍처를 정의하고, 관련된 기술적 핵심 개념을 소개하고, 설계 지침을 제공한다. 6 장에서는 모바일 디바이스의 한 표준으로 자리잡고 있는 안드로이드로 SMA를 상세 설계하고 구현하는 지침을 제공한다. 제시된 기법을 적용하면 보다 효과적으로 고품질의 모바일 어플리케이션을 개발할 수 있다.

2. 관련 연구

본 장에서는 서비스 기반의 모바일 어플리케이션 분야의 관련 연구를 정리한다. Tergujeff의 연구에서는 적용 가능한 기술과 프로그래밍 인터페이스, 지원 가능한 장비들의 조사를 기반으로 모바일 디바이스를 위한 SOA를 제안한다[3]. 연구에서는 JSR 172를 기반으로 하는 아키텍처 설명을 하고 있다.

Natchetoi의 연구에서는 J2ME환경에서 실행 가능한 비즈니스 어플리케이션을 위한 경량 서비스 기반 아키텍처를 제안한다[4]. 연구에서 모바일 디바이스의 중요한 특징을 다루는 설계 방법을 제안에 초점을 두고 있다. 모바일 디바이스의 특징으로는 적은 량의 데이터 전송 및 저장, 능동적 데이터 적재, 보안 등이 있다.

* 종신회원
** 학생회원

Ennai의 연구에서는 서비스지향 프레임워크를 제안한다. 프레임워크는 서비스기반 경량 어플리케이션의 배치와 사용자 컨텍스트에 적절한 서비스와 장치의 적용, 능동적 서비스 호출과 같은 핵심적인 구조적 고려사항을 만족하고 있다[5]. 제안하고 있는 아키텍처는 네 개의 컴포넌트로 구성되어 있고, 최종 사용자, 모바일 SOA계층, 가상 서비스, 모바일 디바이스 플랫폼이 이에 해당 한다. 각 컴포넌트는 동적인 서비스 발견 및 바인딩, 상황인지 서비스 분배, 비동기적 푸시 서비스 호출을 위해 서로 상호작용 한다.

Wang[6]과 Thanf[7]의 연구에서는 모바일 컴퓨팅의 특징적인 요구사항을 언급하며 모바일 서비스를 지원하는 서비스지향 프레임워크를 제안하고 있다. 이들 연구는 구조 설계적인 측면 보다 모바일 서비스를 서비스 사용자게 전달하는 방법에 초점을 두고 있다.

기존의 연구들을 요약하면, 주로 모바일 어플리케이션을 위한 서비스 지향 아키텍처의 중요성과 기초개념에 주로 다루고 있으며, 개괄적인 아키텍처 설계 지침을 설명하는데 그치고 있다.

3. 모바일 디바이스의 특징

모바일 디바이스는 무선 인터넷에 접속되는 휴대용 장치로 정의된다. 이름에서도 알 수 있듯이 모바일 디바이스는 무선 인터넷의 접속성과 휴대성을 강조한 디바이스이다. 모바일 디바이스의 특징은 다음과 같이 정리할 수 있다[1,5,8].

· 무선 인터넷 기반 접속

모바일 디바이스는 높은 수준의 무선 인터넷 접속성을 강조한다. 모바일 디바이스는 CDMA나 GSM 같은 핸드폰 통신 프로토콜 표준과 Wi-Fi 같은 무선 인터넷 표준 프로토콜을 이용한다. 그러나, 무선 인터넷의 일시적인 장애 가능성으로 인해 어플리케이션의 오류가 발생할 수 있어, 어플리케이션 설계시 고려되어야 한다.

· 자원 제약

모바일 디바이스는 다양한 자원 제약성을 가진다. 모바일 디바이스 어플리케이션은 제한된 자원을 효율적으로 활용하도록 설계되어야 한다.

· 휴대성

모바일 디바이스의 장점은 사용자가 이동 중에도 기능성을 제공할 수 있다는 점이다. 이것은 사용자가 모바일 디바이스를 가지고 자유롭게 움직일 수 있다는 것을 의미한다. 사용자가 한 장소에서 다른 장소로 이동한다면, 프로토콜, 시간대, 그리고 심지어 사용자 디바이스에 설치된 어플리케이션에 기능성을 제공하고 있는 서비스 제공자까지도 변경될 수 있다.

이러한 특징들을 고려하여 모바일 어플리케이션과 그 아키텍처가 설계되어야 한다.

4. 독립형과 서비스 기반 모바일 어플리케이션

본 장에서는 모바일 어플리케이션의 두 가지 형태인 독립형 모바일 어플리케이션과 서비스 기반 모바일 어플리케이션에 대하여 알아본다.

4.1 독립형 모바일 어플리케이션

독립형(Standalone) 모바일 어플리케이션은 잠재적인 사용자가 필요로 하는 전체 기능이 모바일 디바이스에 설치, 실행된다. 그림 1은 독립형 모바일 어플리케이션의 전체 구조를 보여준다. 사용자는 모바일 디바이스에 어플리케이션을 설치하고, 해당 어플리케이션은 모바일 디바이스의 자원만을 이용하여 기능을 수행한다.

이는 현재 널리 사용되고 있는 어플리케이션 형태로서, iPhone이나 안드로이드폰에 설치되어 있는 계산기(Calculator) 프로그램이나 메모장(Notepad) 프로그램이 이에 속한다.

모바일 디바이스는 CPU, 메모리 등의 제한된 자원을 가지기 때문에, 엔터프라이즈 어플리케이션과 같이 복잡한 기능을 수행하는 어플리케이션을 실행시키는 것은 어렵다. 그러나, 모바일 디바이스에 설치되어 있는 어플리케이션을 사용하기 때문에 휴대성이 좋고, 네트워크를 이용하여 기능을 수행하지 않으므로 비교적 빠른 시간 내에 응답을 받을 수 있게 된다.

4.2 서비스 기반 모바일 어플리케이션

또 다른 형태인 SMA는 모바일 디바이스의 장점을 부각시키고 단점을 보완하기 위하여 제안된 새로운 형태이다. SMA는 사용자가 필요로 하는 기능의 일부는 서버 측에 배포하고, 모바일 디바이스에 설치된 클라이언트 어플리케이션과의 네트워크를 통한 상호작용을 통하여 전체 기능을 실행한다. 그림 2는 서비스 기반 모바일 어플리케이션의 전체 구조를 보여준다.

사용자가 필요로 하는 전체 기능 및 데이터베이스는 모바일 디바이스에 설치된 클라이언트 측과 서비스 제

그림 1 독립형 모바일 어플리케이션의 구조

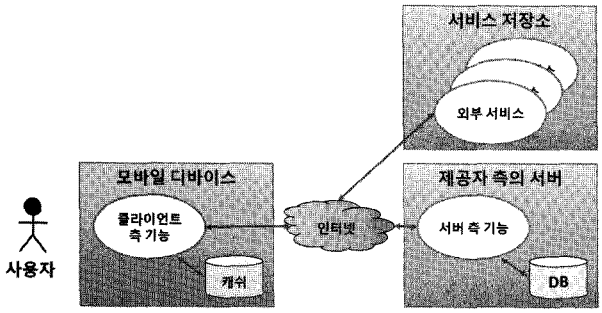

그림 2 서비스 기반 모바일 어플리케이션의 구조

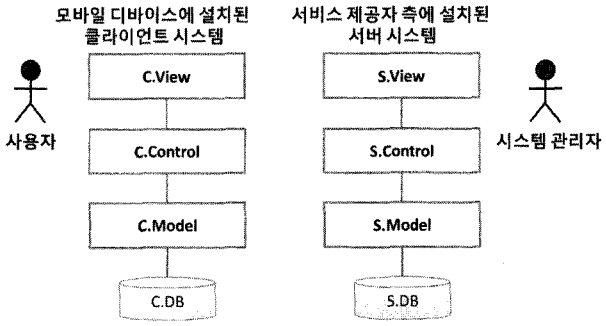

그림 3 Balance MVC의 주요 구성 요소

공자 측에 설치된 서버 측에 분리되어 있다. 전체 어플리케이션 기능 중 비교적 적은 자원을 필요로 하는 간단한 기능은 클라이언트 측에서 실행이 되며, 복잡한 계산 및 데이터 조작을 요구하는 기능은 서버 측에서 실행이 된다. 게다가, 공통적이고 재사용 가능한 기능성은 서비스(Service) 형태로 제공이 되며, 서비스 지향 아키텍처와 클라우드 컴퓨팅에서 언급하는 서비스가 이에 해당된다. 서비스는 모바일 어플리케이션 제공자가 직접 개발하여 제공하거나, 구글의 Map 서비스와 같이 제 3의 서비스 제공자로부터 구독하여 사용될 수 있다.

SMA는 다양하고 풍부한 네트워크를 이용하여 독립형 모바일 어플리케이션의 기능 제약성을 보완할 수 있다. 즉, 성능 좋은 서버에서 제공되는 서비스를 사용함으로써, 모바일 디바이스의 특징인 부족한 컴퓨팅 자원을 확장해서 복잡한 기능의 어플리케이션을 사용할 수 있게 된다. 그러나, 기능 중 일부는 서비스 측에서 실행되기 때문에, 네트워크 안정성이 보장이 되어야만 사용자가 기능 호출에 대한 응답을 받을 수 있다. 그리고, 모든 기능 호출은 반드시 네트워크 통신을 필요로 하기 때문에 독립형 모바일 어플리케이션만큼 빠른 시간 내에 응답을 받지는 못한다.

5. 서비스 기반 모바일 어플리케이션의 아키텍처 기법

본 장에서는 서비스 기반 모바일 어플리케이션의 아키텍처 구성 요소 및 설계 기법을 제안한다.

5.1 Balanced MVC 아키텍처의 구성요소

MVC는 객체지향 시스템 등에 흔히 적용되는 아키텍처 스타일이다[9,10]. SMA는 클라이언트 프로그램과 서버 프로그램이 별도로 존재하기 때문에, 기존의 MVC 아키텍처와는 다른 형태로 구성된다. 즉, 그림 3과 같이 각 프로그램에 MVC 3개의 계층을 별도로 배치할 수 있게 된다. 이를 설명하기 위하여, 본 논문에서는 다음의 용어를 사용한다.

- 클라이언트 어플리케이션을 위한 *C.Model, C.View, C.Control*
- 서비스 시스템은 위한 *M.Model, M.View, M.Control*

*C.View*는 SMA의 사용자를 위한 사용자 인터페이스를 제공하는 계층이다. *C.Control*은 클라이언트 시스템에서 실행되는 비즈니스 프로세스 로직을 실행하며, *C.View*가 기능을 호출할 수 있도록 인터페이스를 제공합니다. 즉, 사용자에게 노출되는 모든 기능은 *C.Control*에 정의되어 있으며, *C.Control*은 *C.Model*에 속한 클래스의 메소드를 호출함으로써 사용자에게 결과를 전송할 수 있게 된다. *C.Model*은 클라이언트 데이터베이스(*C.DB*)를 접근하여 클라이언트 데이터를 관리한다. 모바일 디바이스의 자원이 극히 제한되어 있는 경우, *C.DB*는 영속적인 데이터를 관리하는 데이터베이스가 아니라, 일정 세션 동안만 데이터를 유지하는 캐쉬 형태가 될 수 있다.

*S.View*는 사용자가 아니라 서비스 제공자를 위한 뷰 계층으로, 서버 시스템 관리자를 위해서 선택적으로 존재한다. 만약 시스템 관리를 위한 사용자 인터페이스가 필요하지 않는다면, *S.View*는 구현되지 않는다. *S.Control*은 여러 사용자에 의해 재사용될 수 있는 공통 비즈니스 로직을 구현하고 있으며, *S.Model*에 있는 클래스의 외부 메소드를 호출하여 기능을 수행한다. *S.Model*은 여러 사용자들의 영속적인 데이터를 관리하는 엔티티 클래스를 포함하며, *C.Model*에 속한 대부분의 클래스들이 *S.Model*에도 속한다. *C.Model*에 속한 클래스는 특정 사용자만을 위한 데이터를 관리하지만, *S.Model*에 속한 클래스는 여러 사용자들을 위한 데이터를 제 2의 저장소인 별도의 데이터베이스에 관리한다. 그러므로, *S.DB*에 속하는 일부의 테이블만이 *C.DB*에 존재하게 된다.

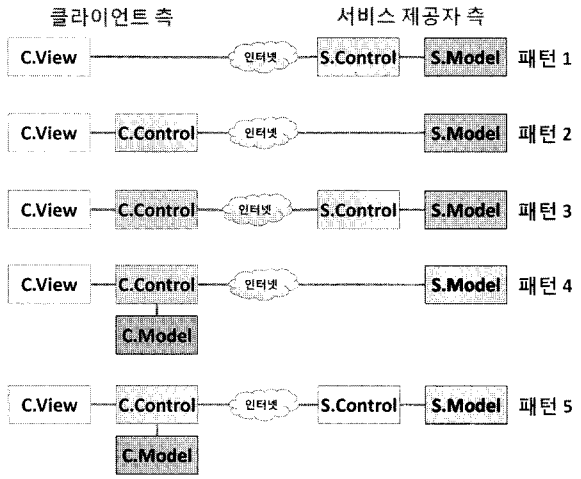

그림 4 MVC 아키텍처의 다섯 가지 패턴

Balanced MVC 아키텍처는 전체 어플리케이션 기능을 클라이언트 측과 서버 측 모두에 위치시키기 때문에, 최적화된 기능 분리에 대해 결정 하는 것은 매우 어려운 일이다. 그림 4는 Balanced MVC 아키텍처 상에서 발생 가능한 형태의 기능 분할을 보여준다. 일반적으로 SMA는 시스템 관리자가 아니라 모바일 디바이스를 소유한 사용자를 위한 기능을 수행하기 때문에, 그림 3의 *S.View*는 고려하지 않았다. 주요 차이점은 비즈니스 로직을 담당하는 컨트롤러의 기능 분할과 *C.Model*의 유무에 있다.

패턴 1은 *S.Control*와 *S.Model*가 존재하는 경우에 해당한다. 클라이언트 측에는 사용자와의 상호작용을 담당하는 *C.View*만 존재하고, *C.View*를 통한 기능 호출은 네트워크를 통하여 서버 측의 *S.Control*와 *S.Model*에 의해서 수행된다. 이 패턴은 자원 제약성을 가지는 모바일 디바이스에 씬 클라이언트 어플리케이션을 설치해야 한다는 조건을 잘 만족시키고 있다. 그러나, 비록 간단한 기능일 지라도, 사용자의 모든 기능 호출은 네트워크를 통해야 한다.

패턴 2는 *C.Control*와 *S.Model*이 존재하는 경우이다. 사용자는 *C.View*를 통하여 비즈니스 로직을 가지는 *C.Contrl*의 기능을 로컬에서 호출하고, *C.Control*은 네트워크를 통하여 *S.Model*에 접근하여 기능을 수행한다. 이 패턴은 *C.Contrl*과 *S.Model* 간의 의존도가 다소 낮어 적은 수의 상호작용을 필요로 하는 경우에 적합하다.

패턴 3은 컨트롤러에 해당하는 *C.Control*과 *S.Control*이 모두 존재하며, 이 외에 *S.Model*이 존재하는 경우이다. 사용자가 *C.View*를 통해 호출한 기능은 *C.Control*와 *S.Control* 간의 원격 상호작용과 영속적 데이터 접근을 위한 *S.Control*과 *S.Model*의 로컬 상호작용을 통해 수행된다. 이 패턴은 SMA의 전체 비즈니스 프로세스 로직을 *C.Contrl*과 *S.Control*에 분할할 수 있고, *C.Control*의 기능은 *C.View*와 잦은 상호작용을 필요로 하고, *S.Control*의 기능은 *S.Model*과 빈번하게 상호작용하는 경우에 적합하다. *C.Control*과 *S.Control*이 비즈니스 로직을 동시에 수행할 수 있으므로, 컨트롤러 계층에서의 높은 병렬 효과를 기대할 수 있다.

패턴 4는 모델에 해당하는 *C.Model*과 *S.Model*이 모두 존재하며, 이 외에 *C.Control*이 존재하는 경우이다. *C.View*를 통해 호출된 *C.Control*의 기능은 *C.Model*과 *S.Model*와의 상호작용을 통하여 수행을 완료한다. 이 패턴은 사용자는 *C.Control*와의 높은 상호작용을 필요로 하고, *C.Control*은 *C.Model*과 *S.Model*을 모두 접근해야 하는 경우에 적합하다. *C.Model*과 *S.Model*에서 동시에 데이터 처리를 할 수 있으므로, 모델 계층에서의 높은 병렬 효과를 기대할 수 있다.

패턴 5는 컨트롤러와 모델 계층이 모두 양측에 분할되어 있는 경우로서, 패턴 3과 패턴 4가 혼합된 형태이다. 이 패턴은 *C.Model*과 *C.Control* 간의 의존도와 *S.Model*과 *S.Control* 간의 의존도가 높고, *C.Control*와 *S.Control* 간의 의존도는 낮은 경우에 적합하다.

각 패턴은 표 1과 같이 각각의 장점과 단점을 가지고 있으므로, 기능 수행 비용, 통신 비용, 병렬 효과 등을 고려하여 적절한 경우에 적절한 패턴을 선택해서 사용해야 한다.

상황에 따라 적절한 패턴을 선택하는 방법은 5.3절에서 살펴본다.

표 1 각 패턴의 장/단점

패턴	클라이언트 측 기능 수행 비용	네트워크 오버헤드	병렬효과
1	낮음	낮음	낮음
2	중간	높음	낮음
3	중간	낮음	높음
4	높음	높음	중간
5	높음	낮음	높음

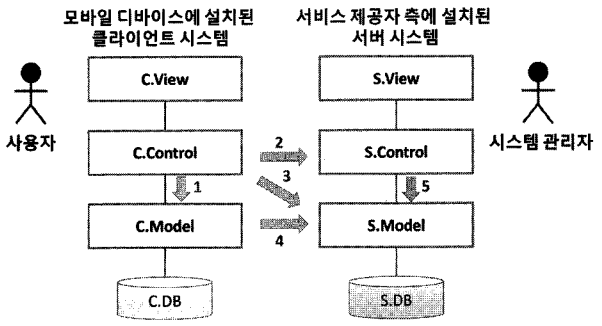

그림 5 Balanced MVC에서의 상호작용경로

5.2 Balanced MVC의 상호연동 경로

Balanced MVC 아키텍처는 클라이언트와 서버 노드가 세 개의 계층을 가지고 있으므로, 이 계층들간은 다양한 상호연동 경로들이 있다. MVC 아키텍처의 기본 원칙을 고려해 보면, 그림 5와 같이 5개의 상호연동 경로를 도출할 수 있다.

그림에서 경로 1은 클라이언트 노드의 컨트롤 계층인 *C.Control* 같은 노드의 모델 계층인 *C.Model* 계층의 객체들의 메소드를 실행하므로, 어떤 통신과 관련된 비용이 들지 않아 효율적이다. 그러나, *C.Model*에 필요한 객체들이 얼마나 복제되어 있는지가 고려되어야 한다.

경로 2는 *C.Control*이 서버 노드의 컨트롤 계층인 *S.Control*에게 메시지를 보내 필요한 기능을 실행하는 방식이다. *S.Control*이 다양한 클라이언드 어플리케이션의 필요로 하는 공통적이고 재사용 가능한 기능을 가지고 있을 때 효과적이다. 이 경우, *S.Control*이 필요로 하는 객체들은 *S.Model*에 있어야 한다.

경로 3은 *C.Control*이 서버 노드에 있는 *S.Control*을 경유하지 않고, *S.Model* 객체들과 직접 연동하는 경우이다. 이는 클라이언트 프로그램이 서버의 엔티티 성격의 객체를 효율적으로 처리해야 하는 경우에 효과적이다. 단, 이 경우, *S.Model*은 양쪽 노드에서 사용 가능하도록 로컬 및 리모드 인터페이스를 제공해야 한다.

경로 4는 클라이언트 노드에 복제되어 있는 객체들의 상태를 서버 노드에 있는 원본 객체들의 상태와 동기화할 때 사용되는 경로이다. 모바일 어플리케이션의 효율성을 고려할 때, 서버 노드의 객체들을 복제해서 사용하면 여러 장점들이 있기 때문에 동기화는 복제로 인해 발생하는 상태 불일치를 일치화하기 위해 불가피한 기능이다.

경로 5 는 MVC의 전형적인 컨트롤과 모델 계층간의 연동이다.

이와 같이 살펴본데로, Balanced MVC에서는 전통적인 MVC에서 고려되지 않았던, 다양한 상호연동 경로들이 존재하고, 어떤 경로를 어떤 기능 수행에 사용할 것인지에 대한 설계가 이루어 져야 한다.

5.3 Balanced MVC 아키텍처 설계 지침

Balanced MVC 아키텍처 설계를 위해서 요구사항 명세서 작성, 기능 모델링, 객체 모델링이 반드시 선행되어야 한다. 즉, Balanced MVC 아키텍처 설계를 위해 요구사항 명세서, 유즈케이스 모델, 객체 모델이 반드시 있어야 한다. 이 모든 입력물들을 작성하기 위하여, 기존의 객체 지향 분석/설계 방법[11]을 다음과 같이 확장하여야 한다.

요구사항 명세서: SMA를 위한 요구사항은 기존의 요구공학 기법을 이용하여 작성할 수 있다. 그러나, 다음과 같이 SMA 특징을 반영한 요구사항을 주로 수집해야 한다.

- 클라이언트 어플리케이션과 서버 시스템 간의 기능 분할을 위한 개략적 수준의 요구사항
- 외부 서비스 (예, Google의Map 서비스) 사용에 대한 요구사항
- 모바일 디바이스의 장점인 이동성 및 컨텍스트 감지 능력을 기반으로 하는 요구사항

유즈케이스 모델 (유즈케이스 다이어그램과 유즈케이스 명세서): 기존의 다이어그램과 크게 차이가 나지 않지만, SMA를 위한 유즈케이스 다이어그램에는 외부 서비스를 위한 액터와 외부 서비스 간의 상호작용을 반드시 표현해야 한다.

SMA는 기존 독립형 어플리케이션과는 달리 다음과 같이 4가지 종류의 상호작용을 포함한다.

- 상호작용 #1) 액터와 클라이언트 시스템 간의 상호작용
- 상호작용 #2) 클라이언트 시스템과 서버 시스템 간의 상호작용
- 상호작용 #3) 클라이언트 시스템과 외부 서비스 간의 상호작용
- 상호작용 #4) 서버 시스템과 외부 서비스 간의 상호작용

이렇듯, SMA는 액터, 클라이언트 어플리케이션, 서버 시스템, 외부 서비스 간의 다소 복잡한 상호작용 경로를 가지고 있으므로, 기존의 2-column 유즈케이스 명세서는 액터와 클라이언트 어플리케이션, 클라이언트 어플리케이션과 서버 시스템, 클라이언트 시스템과 외부 서비스 간의 상호작용을 모두 효과적으로 표현하기 어렵다. 그러므로, 모든 종류의 상호작용을 위해, 표 2와 같은 4-column 유즈케이스 명세서를 제안한다.

SMA 시스템의 '클라이언트 어플리케이션' 열에는 최종 사용자인 액터와의 서버 시스템 또는 액터와 외부 서비스 간의 상호작용을 조정하는 역할을 역할을 한다. 액터와의 모든 상호작용은 클라이언트 어플리케이션을 통해야 한다.

SMA 시스템의 '서버 시스템' 열에는 서버 측에서 수행되는 기능을 나열하고, 이 기능들은 대부분 복잡한 계산을 하여 많은 자원을 소모한다.

'서비스' 열에는 SMA 요구사항 명세서에 기술하였던 SMA와 상호작용하는 외부 서비스 이름을 대괄호와

표 2 확장된 유즈 케이스 명세서

액터	SMA 시스템		«external» 서비스
	클라이언트 어플리케이션	서버 시스템	
Enter…			
	Verify info. Display msg "Enter pin #."		
Enter a pin #.			
	Compute …		
		Retrieve data. Computer…	
	Display…		
Enter…			
	Compute…		
			[Google Map] Find location…
…			

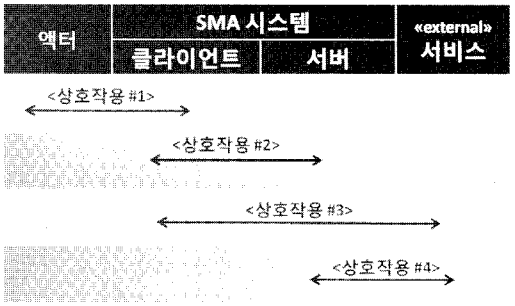

그림 6 유즈케이스 명세서에 표현된 상호작용

함께 기술한다(예, [Google Map]). 이와 같은 방식은 하나의 유즈케이스에서 여러 개의 외부 서비스를 호출할 경우에 유용하다.

4-column 유즈케이스 명세서를 사용하면 그림 6에 표현한 것과 같이, SMA에서 발생가능한 4가지 종류의 상호작용을 모두 효과적으로 표현할 수 있게 된다.

객체 모델: SMA를 위한 객체 모델은 SMA요구사항 명세서에 명세한 기능 분할에 따라 클라이언트 측과 서버 측을 위한 최대 두 개의 객체 모델을 그려야 한다는 점을 제외하고는 기존 모델과 차이점은 없다.

Balanced MVC와 기존 산출물 간의 관계: 지금까지 산출물, 특히 유즈케이스 명세서와 객체 모델은 Balanced MVC 아키텍처를 설계하는데 매우 중요하게 사용된다. 그림 7은 MVC 아키텍처와 이전 산출물 간의 상관 관계를 보여준다.

Balanced MVC 아키텍처 설계: 기존의 산출물을 이용하여 Balanced MVC 아키텍처를 설계하기 위해서는 3단계의 스텝을 거쳐야 한다.

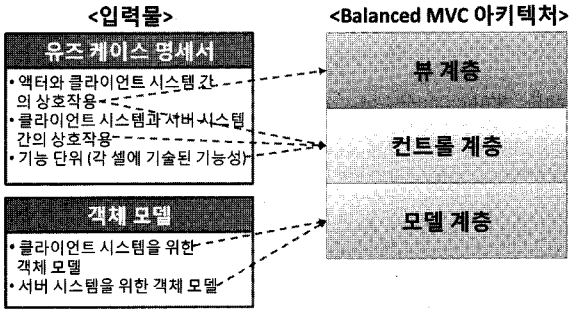

그림 7 Balanced MVC 아키텍처와 입력물 간의 관계

- 스텝 1. 적절한 패턴 선택
- 스텝 2. 각 계층에 포함되는 컴포넌트를 유도하여 선택한 패턴 구체화
- 스텝 3. 각 컴포넌트의 오퍼레이션 정의

스텝 1: 개발하고자 하는 타켓 어플리케이션에 적절한 패턴을 선택하기 위해서는 요구사항 명세서, 객체 모델이 필요하며, 클라이언트 어플리케이션과 서버 시스템 간의 통신비용(Communication Cost)을 줄이는 것을 주 목적으로 한다. 그림 8은 통신 비용을 결정하는데 중요한 역할을 하는 의존성(Dependency)을 고려하여 적절한 패턴을 선택하는 프로세스를 보여준다.

객체 모델을 참조하여 클라이언트 측을 위한 별도의 객체 모델이 있는지를 확인하는 것으로 시작한다. 그림 4에서 보면, 패턴 1, 패턴 2, 패턴 3과 패턴 4, 패턴 5를 분별하는 가장 큰 차이점이 *C.Model*의 유무이기 때문이다.

*C.Model*의 유무를 판단한 후에는 계층 간의 의존성을 분석하여 하나의 패턴을 선택한다. 계층 간의 의존성은 *Depends(A, B)*라는 함수를 이용하여 결정할 수 있으며, 이 함수는 A와 B 간의 예측 가능한 의존성 정도를 반환한다. 이 값은 유즈 케이스 명세서에 기술된 액터, 클라이언트 시스템, 서버 시스템, 외부 서비스

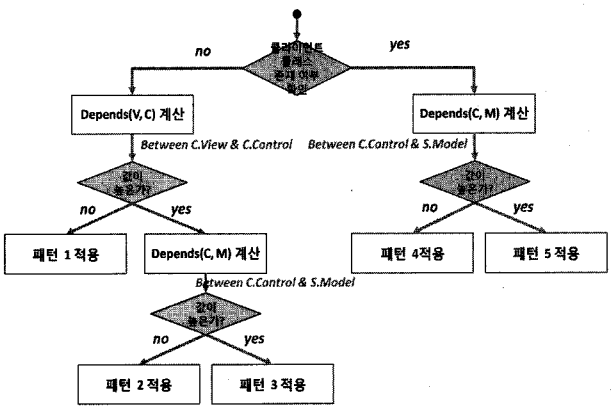

그림 8 적절한 패턴 결정 프로세스

표 3 Depends( )를 이용한 각 패턴의 특징

패턴	패턴의 의미
1	· *Depends(C.View, S.Control)* : minimal · *Depends(S.Control, S.Model)* : maximal
2	· *Depends(C.View, C.Control)* : maximal · *Depends(C.Control, S.Model)* : minimal
3	· *Depends(C.View, C.Control)* : maximal · *Depends(C.Control, S.Control)* : minimal · *Depends(S.Control, S.Model)* :maximal
4	· *Depends(C.View, C.Control)* : maximal · *Depends(C.Control, C.Model)* : maximal · *Depends(C.Control, S.Model)* : minimal
5	· *Depends(C.View, C.Control)* : maximal · *Depends(C.Control, C.Model)* : maximal · *Depends(C.Control, S.Control)* : minimal · *Depends(S.Control, S.Model)* : maximal

간의 메시지 교환 횟수 또는 데이터 교환 횟수를 참고하여 결정된다.

Depends ( ) 함수를 기반으로 한 각 패턴의 특성은 표 3과 같이 요약할 수 있다. 그러므로, Depends( ) 값을 유즈케이스 명세서와 객체 모델을 기반으로 판단하여, 적절한 패턴을 선택할 수 있게 된다.

스텝2: 이전 산출물인 유즈케이스 모델과 객체 모델을 이용하여 각 계층에 포함되는 클래스를 도출한다. 이전 산출물과 각 계층에 포함될 클래스 간의 관계는 그림 7에서 개략적으로 나타나있다.

***C.View*에 포함되는 클래스:** 아키텍처 설계 시점에서는 *C.View*에 포함되는 정확한 클래스를 도출하는 것은 어렵지만, 유즈케이스 명세서에 기술된 액터와 클라이언트 시스템 간의 상호작용을 참조하여 도출하도록 한다. *C.View*에 속하는 클래스들의 상세한 구조는 추후 사용자 인터페이스 설계 과정에서 구체적으로 결정된다.

***C.Model*과 *S.Model*에 포함되는 클래스:** 객체 모델에 포함된 클래스는 대부분 영속적인 객체를 나타내기 때문에, *C.Model*과 *S.Model*의 클래스로 매핑된다.

***C.Control*과 *S.Control*에 포함되는 클래스:** *C.Contrl*과 *S.Contrl*에 포함되는 클래스를 도출하는 것은 유즈케이스 모델에 명시된 기능 그룹으로 시작한다. 기능 그룹은 대부분 유사한 기능을 수행하는 유즈케이스들의 집합으로 구성되며, 대부분 서브시스템, 사용자 권한 등에 따라 결정된다.

먼저, 유즈케이스 모델을 확인하여 기능 그룹을 확인하고, 효율성을 고려하여 기능 그룹들을 재정의한다.

기능 그룹은 유사한 기능들을 수행하는 유즈케이스들로 구성되어 있기 때문에, 이들로부터 일반적으로 하나의 컨트롤러 클래스가 도출된다.

스텝3: 이 단계에서는 유즈케이스 명세서를 기반으로 *C.Control*과 *S.Control*에 속한 클래스들의 오퍼레이션을 도출한다.

*C.Contrl*과 *S.Control*에 포함된 클래스들과 매핑되는 기능 그룹을 먼저 결정한다. 그리고, 기능 그룹에 포함되는 유즈케이스들을 확인한다.

유즈케이스 명세서의 액터와 클라이언트 시스템 간의 상호작용, 클라이언트 시스템과 서버 시스템 간의 상호작용을 분석함으로써, *C.Control*과 *S.Control*의 클래스에 속하는 오퍼레이션들을 정의할 수 있다.

6. 안드로이드 특성을 고려한 아키텍처 설계 및 구현 기법

안드로이드는 전통적인 플랫폼과 상당히 다른 아키텍처적인 차이점을 가지고 있다[12-14]. 본 절에서는 안드로이드의 특성을 고려한, 아키텍처 설계 및 구현 이슈와 해결 기법을 제시한다.

6.1 컴포넌트의 생명주기 관리

안드로이드의 각 컴포넌트는 생명주기가 정의되어 있고, 현재의 상태에 따라 호출되는 메소드가 지정되어 있다. 이는 전통적인 객체지향 방식의 객체의 가용성 부분과 다른 점이다.

즉, 안드로이드의 4개 컴포넌트 중 Acitivity와 Service만 생명주기를 가지고 있으며 각 상태(State)에서 가용한 기능만이 실행되도록 상세 설계 시 반영한다.

6.2 컴포넌트의 실행

안드로이드는 컴포넌트 클래스의 함수 호출이 아닌 Intent를 통해서만 컴포넌트를 시작할 수 있다. Intent는 시스템이 어떠한 행동을 해야 하는지를 지정한 것이다. 따라서 Intent에는 action, category, data 등 컴포넌트 시작에 필요한 것들을 지정할 수 있다. 다음의 코드는 Intent를 초기화 하고, 이를 이용해 서비스를 실행하는 부분을 보여준다.

Activity 의 시작

```
Intent intent =
   new Intent(this, LoginView.class);
startActivity(intent);
```

Service 의 시작

```
Intent i = new Intent(this, com.MMS.So
   cket.ClientService.class);
bindService(i, conn, Context.BIND_AUTO
   _CREATE);
```

6.3 컴포넌트 간의 데이터 전달

안드로이드는 컴포넌트를 시작함에 있어서 Intent를

사용하기 때문에 메소드 호출시의 매개변수 값을 전달할 수 있는 방법이 없다. 이 경우 안드로이드는 변수 값을 공유하는 방법을 세 가지로 지정하고 있다.

Bundle: Intent의 putExtra메소드를 이용하면 bundle을 이용하여 시작하는 컴포넌트에 값을 전달할 수 있다. 하지만 기본적인 데이터 타입만을 지원한다.

Shared Preference: Preference에 값을 저장하고 다른 곳에서도 그 값을 불러 쓸 수 있는 기능이다.

Content provider: 데이터베이스나 파일, 네트워크를 이용해 컴포넌트간 데이터를 공유하는 방법이다. 이것은 하나의 컴포넌트로 따로 컴포넌트를 구현해야 하는 불편함이 있지만 데이터는 계속 지속된다.

6.4 외부 서비스의 사용

안드로이드 어플리케이션에서 서비스 저장소에 등재된 외부 서비스를 이용하기 위해서 네트워크 접근이 필요하다. 자바의 socket을 활용하던지, RESTful 서비스 방식을 이용할 수 있다. 다음 코드는 socket을 이용하여 서비스를 호출하는 예제이다.

Socket connection 사용방법:

```
private static Socket connectServer (int
   port){

   InetAddress serverAddr;
   Socket socket = null;
   try {
     serverAddr =
       InetAddress.getByName(ServerIP);
     socket = new Socket(serverAddr,
       port);
   } catch (UnknownHostException e) {
     e.printStackTrace();
   }catch (IOException e) {
     e.printStackTrace();
   }
   return socket;
}
```

6.5 배경에 있는 Activity의 실행

실행중인 activity가 다른 activity의 실행에 의해서 상태가 변화하여 background에 있을 때 호출할 수 없다. 안드로이드는 실행하는 activity를 Activity manager가 스택을 이용하여 관리한다. 이것을 이용하여 이전에 실행했던 activity를 호출할 수 있다. 또는 service에서 호출 시 callback 등록하여 activity를 다시 foreground로 호출 할 수 있다.

6.6 안드로이드의 라이브러리 기능 사용

안드로이드의 여러 기능(Location, Sensing, etc)을 사용함에 있어서 모든 부분에 Context 정보가 필요하다. Context는 Activity, Service 등 주요 컴포넌트의 base class이다. 따라서 안드로이드의 기능을 이용할 때는 컴포넌트 내에서 사용하거나 컴포넌트의 객체를 전달받아 사용해야 한다.

7. 사례 연구

본 장에서는 SMA를 위한 Balanced MVC 아키텍처 설계 기법의 적용성을 보여주기 위하여, Mobile Mate Service(MMS)를 도메인으로 사례 연구를 수행한다. MMS는 모바일 디바이스을 보유한 사용자의 위치정보를 이용하여 소셜 네트워킹 기능을 지원하는 어플리케이션이다.

MMS는 크게 4가지 종류의 기능 그룹 요구사항을 포함한다.

- 멤버 관리: 멤버 가입, 멤버 정보 수정, 검색 및 삭제
- 프로파일 관리: 멤버 개인의 취미 등의 프로파일 생성, 수정 및 검색
- 그룹 관리: 그룹 생성 및 삭제, 친구 초대, 메시지 전달, 친구 위치 확인 등
- 기타 기능: 로그인, 로그아웃, 도움말

유즈케이스 다이어그램 작성시에 각 유즈케이스마다 기능 그룹을 명시하기 위하여 두 글자로 이루어진 식별자를 부여하였다. 예를 들어, 멤버 관리와 관련된 유즈케이스에는 식별자MM, 그룹 관리와 관련된 유즈케이스에는 식별자GM을 부여하였다.

MMS 요구사항에는 클라이언트 측과 서버 측의 기능 분할에 대한 요구사항이 포함되어 있으므로, 객체 모델링 시에 클라이언트 측을 위한 객체 모델과 서버 측을 위한 객체 모델을 별도로 작성하였다.

객체 모델이 클라이언트 측과 서버 측으로 분리되어 있기 때문에, 그림 8의 프로세스를 따라 패턴 4 또는 패턴 5 중 하나가 Balanced MVC 아키텍처 후보가 된다.

그리고, 유즈케이스 명세서 상의 기술된 클라이언트 시스템과 서버 시스템 간의 메시지 상호작용과 데이터 상호전달 횟수를 분석하여, *C.Control*와 *S.Model* 간의 의존도가 높은 것으로 판단되었다. 그러므로, *C.Control*과 *S.Model*이 직접적으로 상호작용하는 횟수를 줄이기 위하여, *C.Control*과 의존도가 낮은 부분을 *S.Control*로 분리하고, *S.Control*이 *S.Model*과 주로 상호작용을 하도록 패턴 5를 MMS의 Balanced MVC 아키텍처로 선정하였다. 그림 9는 패턴 5를 적용하여 설계된 MMS의 Balanced MVC 아키텍처를 보여준다.

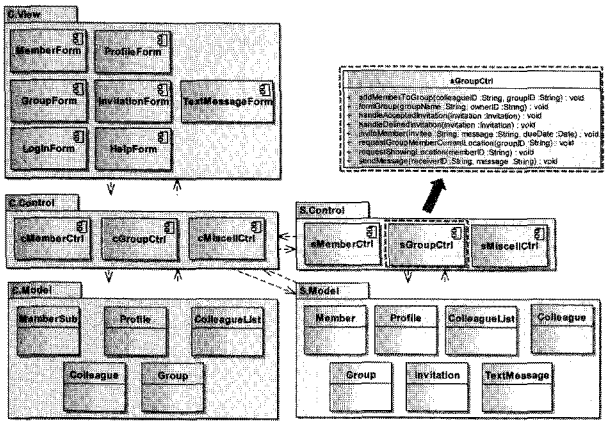

그림 9 Balanced MVC 아키텍처가 적용된 결과

클라이언트측에는 *C.Model*, *C.Control*, *C.View*가 존재하며, 서버측에는 *S.Control*과 *S.Model*만이 존재한다. 서버 측과 클라이언트 측 간의 대부분의 상호작용은 *C.Control*과 *S.Control* 사이에서 이루어지며, *C.Control*과 *S.Model* 간의 상호작용도 존재한다. *C.Control*과 *S.Model* 간의 상호작용은 클라이언트 측에서 서버 측에 저장된 멤버의 프로파일(Profile) 정보를 가져오는 과정에서 발생한다. 별도의 가공없이 멤버의 프로파일 정보를 가져오면 되기 때문에, *S.Control*을 거쳐 성능을 저하시키는 것을 해결하기 위하여 직접 *S.Model*에 속하는 클래스인 Profile을 접근할 수 있도록 설계하였다.

MMS에 적절한 패턴을 선택한 후에, *C.Control*과 *S.Control*에 속할 클래스를 도출한다. 유즈케이스 다이어그램 작성시에 4개의 기능 그룹을 이용하였지만, 그 그룹중 멤버 관리 그룹과 프로파일 관리 그룹은 상당히 밀접한 관계가 있기 때문에 하나의 그룹으로 재정의하였다. 그리하여, 멤버 관리를 위한 컨트롤러 클래스(*cMemberCtrl*과 *sMemberCtrl*), 그룹 관리를 위한 컨트롤러 클래스(*cGroupCtrl*과 *sGroupCtrl*), 기타 기능들을 위한 컨트롤러 클래스(*cMiscellCtrl*과 *sMiscellCtrl*)를 클라이언트 측과 서버 측에 각각 배치하였다.

마지막으로, 각 컨트롤러 클래스에 포함되는 오퍼레이션을 유즈케이스 명세서를 이용하여 도출한다. 그림 9를 보면, *sGroupCtrl*을 위한 오퍼레이션 목록들이 나열되어 있으며, 이는 *sGroupCtrl*과 관련된 그룹 관리 유즈케이스에 대한 명세서를 분석하여 도출하였다.

본 사례 연구를 통하여 설계된 MMS의 Balanced MVC 아키텍처를 분석하면 다음과 같은 결과를 도출할 수 있다.

서버 측에는 모바일 디바이스를 통해 MMS에 접근하는 모든 사용자의 정보가 저장되어 있으며, 다수 사용자 간의 소셜 네트워킹을 위한 기능을 제공한다. 그리고, 여러 사용자의 위치 정보를 검색하는 등의 다소 복잡한 계산을 필요로 하는 기능을 수행한다. 이 기능들은 *sGroupCtrl*의 오퍼레이션으로 설계된다.

모바일 디바이스에 배포된 어플리케이션에는 특정 개인과 관련된 정보와 다소 간단한 계산을 필요로 하는 기능만 가지고 있어 많은 자원을 필요로 하지 않는다. 그러므로, 모바일 디바이스 상에서 다소 복잡한 기능을 많은 자원을 소모하지 않고 사용할 수 있게 된다.

클라이언트 측과 서버 측의 통신 비용을 절감하기 위하여, *C.Control*의 클래스가 *S.Model*에 속한 Profile 클래스를 직접 접근하는 것을 허용하였다.

8. 맺는말

모바일 디바이스는 퍼스널 컴퓨팅 용도뿐만 아니라, 기업의 엔터프라이즈 어플리케이션의 클라이언트 단말로도 사용될 것으로 예상된다. 또한, 상황 인지 장비로서의 기능도 우수하다. 그러나, 모바일 디바이스는 다양한 자원 제약성을 가지고 있다. 따라서, 복잡한 기능의 어플리케이션은 이 디바이스 상에서 설치와 운영이 어렵다.

본 논문에서는 이러한 단점을 극복하고 모바일 디바이스의 활용을 최대화하기 위해 서비스 기반의 모바일 어플리케이션(Service-based Mobile Application, SMA)을 설명하였다. SMA는 일부 혹은 상당 부분의 기능이 서비스의 형태로 구현되어 서버 측에서 설치 운영되고, 사용자는 모바일 디바이스에 설치된 클라이언트 어플리케이션은 이 서비스를 호출하여 사용한다. 이렇게 두 개의 노드에서 목표 어플리케이션을 실행함으로 해서, 모바일 디바이스가 가지는 자원 제약성의 문제는 상당히 해소된다.

본 논문에서는 이러한 SMA에 효과적으로 적용될 수 있는 Balanced MVC 아키텍처를 제안하였다. 즉, 이 아키텍처의 핵심 개념, 구성 및 설계지침을 설명하였고, 효과적인 아키텍처 설계 지침도 제공하였다. 또한, 안드로이드에서 이 아키텍처를 구현할 때 필요한, 상세 설계 및 구현 지침도 제시하였다. 제시된 기법을 적용하면 보다 효과적으로 고품질의 모바일 어플리케이션을 개발할 수 있다.

Acknowledgement

본 연구는 방위사업청과 국방과학연구소의 지원으로 수행되었습니다 (UD060048AD).

참고문헌

[1] König-Ries, B. and Jena, F., "Challenges in Mobile

Application Development," *it-Information Technology*, Vol. 52, No. 2, pp. 69-71, 2009.

[2] Gillett, F.E., "Future View: New Tech Ecosystems of Cloud, Cloud Services, and Cloud Computing," *Forrester Research Paper*, 2008.

[3] Tergujeff, R., Haajanen, J., Leppanen, J., and Toivonen, S., "Mobile SOA: Service Orientation on Lightweight Mobile Devices," *In Proceedings of 2007 IEEE International Conference on Web Services (ICWS 2007)*, pp. 1224-1225, 2007.

[4] Natchetoi, Y., Kaufman, V., and Shapiro, A., "Service-Oriented Architecture for Mobile Applications," *In Proceedings of the 1st international workshop on Software architectures and mobility (SAM '08)*, pp. 27-32, 2008.

[5] Ennai A and Bose S, "MobileSOA: A Service Oriented Web 2.0 Framework for Context-Aware, Lightweight and Flexible Mobile Applications," *In Proceedings of the 2009 12th Enterprise Distributed Object Computing Conference Workshop (EDOCW 2008)*, pp. 348-382, 2008.

[6] Wang Q and Deters R, "SOA's Last Mile - Connecting Smartphones to the Service Cloud," *In Proceedings of 2009 IEEE International Conference on Cloud Computing (CLOUD 2009)*, pp. 80-87, 2009.

[7] Thanh, D. and Jorstad, I., "A Service-Oriented Architecture Framework for Mobile Services," *In Proceedings of the Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference / E-Learning on Telecommunications Workshop (AICT/SAPIR/ELETE'05)*, pp. 65-70, 2005.

[8] Forman, G.H and Zahorjan, J, "The Challenges of Mobile Computing," Computer, Vol. 27, No. 4, pp. 38-47, 1994.

[9] Clements, P., et al., *Documenting Software Architectures Views and Beyond*, Addison-Wesley, 2003.

[10] Rozanski, N. and Woods, E., *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, Addison Wesley, 2005.

[11] Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual*, Second Edition, Addison Wesley, 2005.

[12] Android Developers, http://developer.android.com/index.html.

[13] Burnette, E., *Hello, Android Introducing Google's Mobile Development Platform*, The Pragmatic Bookshelf, 2008 .

[14] Hashimi, S.Y. and Komatineni, S., *Pro Android*, Apress, 2009.

약 력

김 수 동

1984 Northeast Missouri State University 전산학 학사
1988/1991 The University of Iowa 전산학 석사/박사
1991～1993 한국통신 연구개발단 선임연구원
1994～1995 현대전자 소프트웨어연구소 책임연구원
1995～현재 숭실대학교 컴퓨터학부 교수.
관심분야 : 서비스 지향 아키텍처(SOA), 클라우드 컴퓨팅(Cloud Computing), 모바일 서비스(Mobile Service), 객체지향 S/W공학, 컴포넌트 기반 개발 (CBD), 소프트웨어 아키텍처
E-mail : sdkim777@gmail.com

라 현 정

2003 경희대학교 우주과학과 이학사
2006 숭실대학교 컴퓨터학과 공학석사
2006～현재 숭실대학교 컴퓨터학과 박사수료
관심분야 : 서비스 지향 아키텍처(SOA), 클라우드 컴퓨팅(Cloud Computing), 모바일 서비스(Mobile Service)
E-mail : hjla@otlab.ssu.ac.kr