모바일 어플리케이션 영역의 클라이언트 개발에 적용 가능한 소프트웨어 아키텍처 패턴 설계*

조호진° 양진석
포항공과대학교 소프트웨어 공학 연구실
NUCL23@postech.ac.kr, badukee@chol.com

Designing of Software Architecture Patterns for Developing Client in Mobile Application Domain

Hojin Cho° Jin-Seok Yang
POSTECH Software Engineering Laboratory

요 약

모바일 어플리케이션 영역에서의 클라이언트 어플리케이션 개발 시, 지금까지는 소프트웨어 공학적인 접근 방법이 없이 빠른 시장 출시를 위해 급급한 개발에 초점을 두었다. 실제로 쉽게 적용이 가능한 방법과 다양한 품질 속성을 만족할 수 있는 방법이 존재함에도 불구하고 시도하지 않고 있다. 본 논문에서는 실제로 적용 가능한 접근 방법으로 소프트웨어 아키텍처 패턴을 제안하며, 이를 이용할 때 나타날 수 있는 단점을 보완하는 방법을 제시한다.

1. 서 론

최근에는 모바일 기기를 사용하지 않는 사람이 거의 없을 정도로 사용자 수가 많고 모바일 어플리케이션의 사용 역시 증가하고 있다.

이러한 클라이언트 측의 모바일 어플리케이션들의 공통된 특징으로 빠른 시장 출시(time-to-market)를 만족해야 하며, 어플리케이션의 크기가 작다는 점을 들 수 있다. 또한 품질 속성으로는 보안 적인 측면을 강화하거나 제한된 자원을 이용해야 한다는 점이 있다.

모바일 개발 업체에서 실제로 개발하는 상황, 환경 그리고 결과물을 살펴보면, 대부분 클라이언트 측의 모바일 어플리케이션에서 코드 수준의 재사용에 머물러 있다. 또한 컨텐츠의 경우 상당히 비슷한 종류를 개발하고 있다. 이러한 문제를 해결하게 되면 앞 단락에서 설명한 특징들을 대부분 만족시킬 수 있음에도 불구하고 지금까지는 별다른 연구 및 결과물이 나오지 않고 있다.

본 논문에서는 이러한 문제점을 해결하기 위한 방법으로 소프트웨어 아키텍처 패턴을 이용한 방법을 제안하며 이를 통해서 위에서 설명한 품질 속성 및 모바일 어플리케이션 영역의 특징들을 만족할 수 있는 패턴을 설명한다. 2장에서는 모바일 어플리케이션 영역에 대해서 설명하며, 3장에서는 소프트웨어 아키텍처 패턴을 위한 템플릿과 실제 두 가지의 아키텍처 패턴을 제안한다. 4장에서는 이러한 패턴을 적용하기 위한 방법을 제안하며 5장에서 향후 연구 및 결론을 서술한다.

2. 모바일 어플리케이션 영역

모바일 어플리케이션 영역에서는 어플리케이션이 구동되는 환경에 따라 크게 서버와 클라이언트로 분류된다. 대부분의 모바일 어플리케이션 영역의 아키텍처는 서버와 클라이언트로 구분하여 설명을 한다.[1] 하지만, 단순히 클라이언트 영역의 아키텍처를 레이어 아키텍처 스타일로 구분하고 있어 모바일 어플리케이션 개발에 아키텍처를 적용해서 개발하기에는 부족하다. 실제 모바일 게임 영역의 모바일 어플리케이션을 대상으로 역공학(Reverse Engineering)을 적용하여 분석해 보면, 제품의 전반적인 부분에 걸쳐 구조화를 시킬 수 있는 패턴을 볼 수 있다. 이는 특정한 문제를 해결할 수 있는 디자인 패턴[2]의 수준을 넘어 소프트웨어 아키텍처 관점에서의

* 본 연구는 정보통신부 및 정보통신연구진흥원의 IT신성장동력 핵심기술개발사업의 일환으로 수행하였음. [2007-S032-01, 다중 플랫폼 지원 모바일 응용 S/W 개발 환경 기술 개발]

패턴이며, 어플리케이션의 전반적인 부분에 걸쳐 적용 가능한 것이라 할 수 있다. 본 논문에서는 소프트웨어 아키텍처 패턴이라는 용어를 [3]의 정의를 따른다.

또한 본 논문에서는 아키텍처 패턴을 도출하기 위한 대상 영역으로 모바일 어플리케이션 영역 중에서도 리치 클라이언트(rich client 또는 fat client)를 대상으로 하며 그 중에서도 최근 많은 사용자를 확보하고 있는 모바일 게임을 대상으로 한다. 모바일 게임에도 종류가 많은데 [1]에 제시된 "접속 유지형", "필요시 접속형", 그리고 "단일형" 어플리케이션 중, 필요시 접속형(partially connected)을 대상으로 한다. 모바일 게임 어플리케이션을 대상으로 한 이유는 모바일 어플리케이션 영역가운데 가장 짧은 시장 출시 기간과 소프트웨어 생명주기를 가지고 있으므로 아키텍처 패턴을 적용하기 좋은 영역이기 때문이다.

클라이언트 측의 어플리케이션들은 서버 측의 어플리케이션에 비해서 크기가 작고 짧은 주기의 시장 출시(time-to-market)을 가지고 있다. 더구나 클라이언트 측의 하드웨어와 플랫폼은 서버 측 보다 사용가능한 자원이 더 제한되어 있다. 또한 모바일 어플리케이션 영역에서는 개발 시에 사용할 수 있는 프로그래밍 언어가 플랫폼에 종속적인 C, Java 등으로 제한적이기 때문에 이러한 점도 제약사항이 된다. 이러한 제약사항 외에 아키텍처 패턴을 도출하기 위해 모바일 게임 영역의 주요한 특성들을 요약하면 아래와 같다.

1) 비슷한 구조의 어플리케이션 개발
   - 모바일 게임들은 스토리만 달라질 뿐 메뉴와 컨트롤러 등의 구조가 유사하다.
2) 빠른 시장 출시
   - 장르별로 대부분 비슷한 게임들이 많으므로 먼저 출시하는 곳이 시장을 선점한다.
3) 코드 수준의 재사용
   - 코드 조각이나 라이브러리 수준에서의 재사용만이 사용되고 있다.

위에서 나열된 특징들은, 코드 수준의 재사용을 극복한다면 생산성이 높아짐과 동시에 대부분 해결이 가능한 것들이라고 할 수 있다. 코드 수준의 재사용을 극복하면서 높은 적용성을 갖기 위해서 아키텍처 패턴을 제안한다.

하지만, 소프트웨어 아키텍처 패턴을 적용할 때 주의해야 할 점이 있다. 예를 들어, 성능과 관련하여 클래스의 개수가 적을수록 좋다는 비기능적인 요구사항이 있을 수 있고 이를 만족시키는 방법으로 하나의 클래스에 모든 기능을 구현할 수도 있다. 즉, 재사용과 성능 등의 두 가지를 모두 만족시키기는 어렵다. 따라서 두 가지를 상충하여 적정선에서 만족시킬 수 있도록 모바일 어플리케이션의 소프트웨어 아키텍처 패턴을 정의하도록 해야 한다. 이러한 성능과 관련한 적용 방법에 대한 내용은 5장의 향후 연구 및 결론에서 살펴보도록 한다.

3. 소프트웨어 아키텍처 패턴

본 논문에서 말하는 소프트웨어 아키텍처 패턴은 소프트웨어 아키텍처 스타일[4]과 유사하지만, 많이 추상화되어있지는 않고 실제 개발 시 바로 적용할 있으며 실제 구현과 밀접한 관련이 있다는 점에서 다르다. 이러한 아키텍처 패턴이 가져야 할 품질 속성은 아래와 같다.

표 1. 아키텍처 패턴이 가져야 할 품질 속성

품질 속성	설명
적용성 (Adaptability)	실제 개발에 적용이 가능해야 한다.
재사용성 (Reusability)	코드 수준을 극복하여 전반적인 구조에 대해서 재사용이 가능해야 한다.
변경가능성 (Modifiability)	변경 가능한 부분에서 변화를 수용할 수 있어야 한다.
성능 (Performance)	실제로 사용될 수 있도록 적절한 성능이 뒷받침 되어야 한다.

이러한 품질 속성을 만족 시키면서 실제 개발에 바로 적용이 가능하다면, 아키텍처 수준에서의 재사용이 실현되고 결론적으로 코드 수준에서 보다 생산성이 향상될 것을 기대할 수 있다. 또한 아키텍처 패턴에서 컴포넌트를 통한 모듈화는 낮은 결합도과 높은 응집성을 구현하는 개체가 되며, 개별 컴포넌트의 지정된 인터페이스는 패턴에서 지정되므로 동시(concurrent) 개발과 개별 컴포넌트의 전문적인 구현을 기대할 수 있고 유지보수성 측면에서도 효과를 볼 수 있다.

아키텍처 패턴의 구조를 시각적으로 표현하기 위한 표기법을 [그림 1]과 같이 제안한다.

표기법은 기본적으로 [5]의 "Notation for Basic Components and Interfaces"와 같으며, 차이점 이라면 본 논문에서 제시하는 컴포넌트는 자신이 반드시 가져야 하는 객체들을 표현해야 한다는 점이다. 이는 본 논문에서 제안하는 소프트웨어 아키텍처 패턴을 구현에 바로

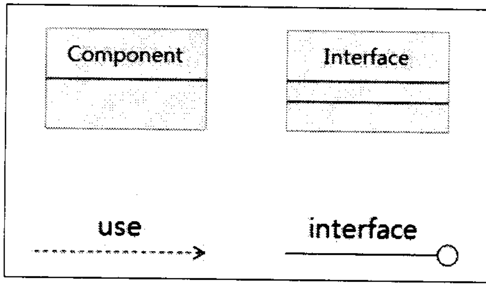


그림 1. 아키텍처 패턴을 위한 표기법

적용할 수 있도록 하며, 각 컴포넌트를 실제로 구현하는 클래스가 어떤 것인지 알 수 있으므로 필요한 정보들만 볼 수 있고, 클래스 간의 관계를 다 보지 않아도 되므로 컴포넌트로의 범위 한정을 통해 개발 시의 복잡성을 줄일 수 있다. 개별 컴포넌트를 설명하기 위해서는 컴포넌트를 구현하는 클래스들을 UML 클래스 다이어그램을 이용하여 표현할 수 있다.

[1]에 따르면, 모바일 어플리케이션의 클라이언트 측에서는 씬 클라이언트(thin client)와 리치 혹은 팻 클라이언트(rich/fat client)의 두 가지로 아키텍처를 구분하고 있다. 하지만 모바일 게임 영역에서는, 대부분 리치클라이언트의 3계층 모델처럼 모델-뷰-컨트롤러(MVC) 형식으로 표현할 수 있다. 그 이유는 게임을 컨트롤 하는 부분, 게임의 모델 정보, 화면 표시 부분으로 항상 나눌 수 있기 때문이다. 따라서 본 논문에서는 이러한 기본 MVC 구조 위에서 게임 영역에 적합한 아키텍처 패턴을 도출하는 것을 목적으로 한다.

다양한 패턴들을 효과적으로 분류 및 관리하기 위해서는 표준 형식이나 템플릿 등이 필요하며 본 논문에서는 아래 항목들을 제안한다.

표 2. 아키텍처 패턴을 위한 템플릿

항목	설명
이름 (Name)	아키텍처 패턴의 이름
문맥 (Context)	이 아키텍처가 사용되는 환경
해결 (Solution)	컨텍스트에 어떻게 적용되는지
제약사항 (Constraints)	패턴에서 반드시 지켜져야 하는 제약사항
변화 (Variant)	패턴에서 변화 가능한 부분
관련된 패턴 (Related pattern)	이 패턴과 관련된 다른 패턴들

위 형식을 사용하여 다음 절에서부터 개별 패턴을 설명하도록 한다.

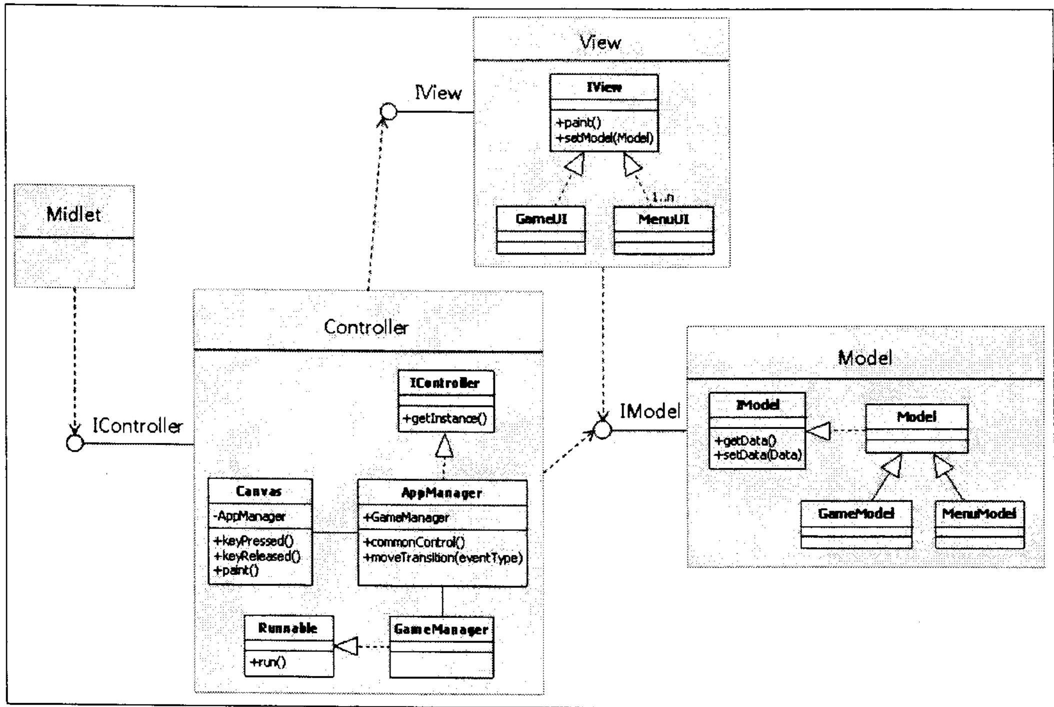
3.1 컨트롤러 집중형 패턴

컨트롤러 집중형 패턴은 기본적인 MVC 구조에서 컨트롤러에 대부분의 책임을 부여하여 컨트롤러가 대부분 일을 수행하는 것이다. 어플리케이션이 구동되는데 필요한 정보를 이를 제어하는 컨트롤러에 모아서, 클래스 등이 로딩되는 시간을 효과적으로 줄여 성능 향상을 기대할 수 있는 패턴이다. 이 때, MVC의 모델과 뷰는 수동적인 개체이며, 컨트롤러에서 모델의 정보를 변경하게 되고, 뷰는 컨트롤러의 지시를 받아서 화면을 업데이트 하게 된다. 이는 관련된 패턴들 중 옵저버 패턴을 이용하면 능동적으로 수행하도록 변경할 수도 있다. 이 패턴을 표준 형식으로 정리하면 아래와 같다.

표 3. 컨트롤러 집중형 패턴

항목	설명
이름 (Name)	컨트롤러 집중형 패턴
문맥 (Context)	모델과 뷰의 개수가 적어서 별도의 컴포넌트로 관리하지 않아도 되며, 로딩 될 수 있는 클래스 개수를 줄여 하나의 컴포넌트 내에 구현할 수 있을 때.
해결 (Solution)	모델과 뷰는 수동적인 개체로 만들고 컨트롤러에서 모델을 편집하고 뷰를 이용하여 화면을 업데이트 하도록 한다.
제약사항 (Constraints)	컨트롤러는 AppManager를 갖는다. 모델에서는 다른 컴포넌트를 참조할 수 없다.
변화 (Variant)	모델이 수동적이기 때문에 자료 객체로 보고 컨트롤러에서 포함할 수 있다.
관련된 패턴 (Related pattern)	옵저버 패턴, 팩토리 패턴, MVC 패턴

컨트롤러 집중형 패턴의 전체 구조는 [그림 2]에 나타나 있다. [그림 2]에서 볼 수 있듯이 컨트롤러 컴포넌트에서 대부분의 주요한 매니저 클래스들을 갖고 있는 것을 확인할 수 있다. 이는 어플리케이션을 구동하는데 필요한 정보는 모두 컨트롤러에 집중되어 있다는 것을 의



[그림 2] 컨트롤러 집중형 아키텍처 패턴

미한다. 뷰는 수동적인 개체이지만, 컨트롤러가 화면을 그리는 메소드를 호출하면 모델의 정보를 참조하여 화면에 출력하게 된다. 컴포넌트 내에 포함되는 GameUI와 MenuUI는 IView 인터페이스를 구현해야 하며, 해당 인터페이스는 paint()와 setModel() 메소드를 구현해야 한다.

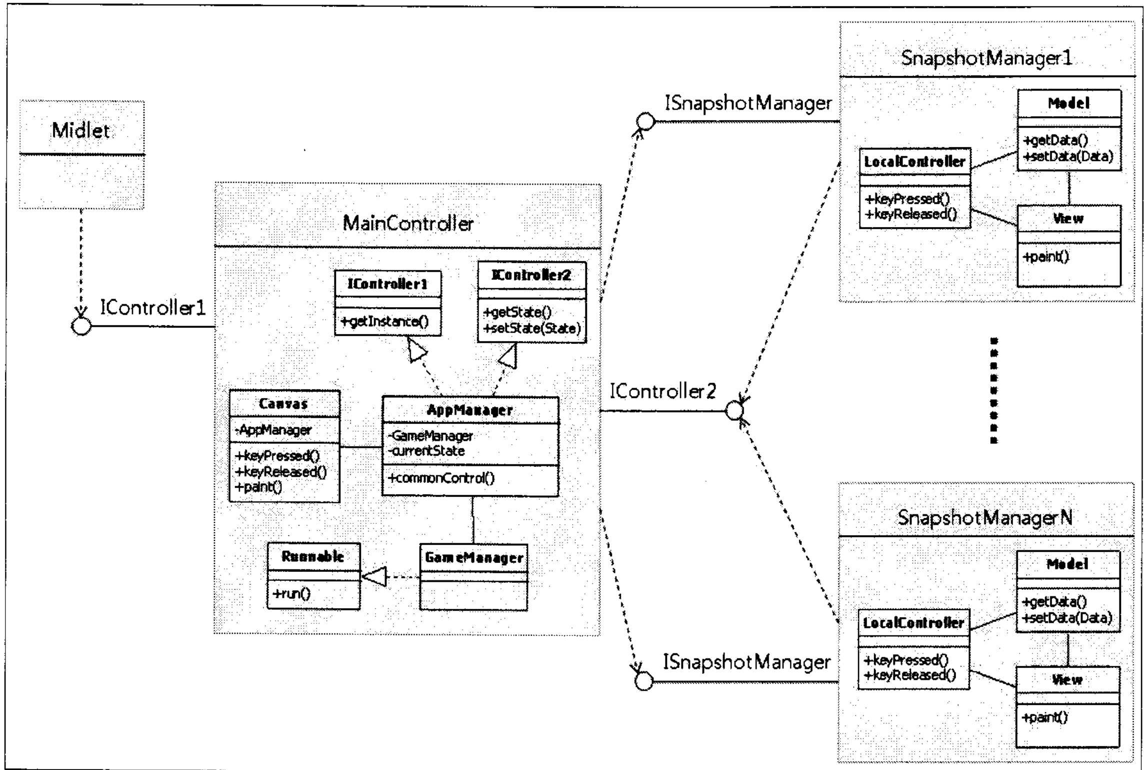
개별 컴포넌트 내에는 UML 클래스 다이어그램을 표현할 수 있고, 해당 클래스들이 컴포넌트를 구현하게 된다. 클래스들 간의 관계에서 필요한 부분에는 허용 가능한 요소 개수(cardinality)를 표현할 수 있다. 그리고 아키텍처 패턴을 표현할 때, 그림 2와 같이 클래스 다이어그램을 이용하여 컴포넌트를 표현할 수도 있지만 속성(attribute)처럼 표현하는 방법도 허용한다.

3.2 화면별 분산 제어 패턴

화면별 분산 제어 패턴은, 3.1에서 설명한 컨트롤러 집중형 패턴과 달리 컨트롤러가 하던 책임을 각 화면(사용자 인터페이스)으로 위임한 것이다. 이는 실제 모바일 어플리케이션 개발에서의 특성인 "화면 중심의 개발"을 지원하는 패턴이다. 화면 중심의 개발을 위해서는 화면 간에 인터페이스가 분명해야 하며, 각 화면이 하나의 컴

표 4. 화면별 분산 제어 패턴

항목	설명
이름 (Name)	화면별 분산 제어 패턴
문맥 (Context)	모바일 어플리케이션에서 사용되는 화면이 많아서 개발시간을 줄이고 싶을 때 혹은 비슷한 어플리케이션 개발 시 화면(사용자 인터페이스)의 코드수준에서의 재사용을 극복하려 할 때.
해결 (Solution)	화면 별 매니저 컴포넌트를 두고 컴포넌트 내부에서는 MVC구조 등을 이용하여 미리 정의된 인터페이스를 통해 통신한다.
제약사항 (Constraints)	화면 별로 사용되는 모델 정보와 이를 제어하는 컨트롤러가 같이 존재한다.
변화 (Variant)	컴포넌트 내부의 구조가 MVC외에 모델을 관리할 수 있는 유사한 구조로 변화 가능하다.
관련된 패턴 (Related pattern)	MVC 패턴



[그림 3] 화면별 분산 제어 패턴

포넌트가 되어 동시(concurrent) 개발을 가능하게 한다. 전체적인 구조는 [4]에서 소개된 블랙 보드(Black board) 스타일과 유사하며 Snapshot Manager 컴포넌트가 동시에 수행될 수 없다는 점이 다르다.

화면별 분산 제어 패턴의 전체 구조는 [그림 3]과 같다. 화면별 컴포넌트의 범위가 인터페이스에 의해 정해지게 되므로 동시 개발이 가능하게 된다. 3.1절에서 설명하였던 패턴과 반대로 클래스 개수가 많아지기 때문에 각 컴포넌트 별 클래스 구조는 변할 수 있다.

화면이 컴포넌트 하나로 구현되기 때문에, 모바일 어플리케이션 개발 시 공통적인 메뉴 화면이나 다른 버전의 같은 게임에서 컴포넌트 재사용이 가능하다. 이처럼 컴포넌트 재사용성을 높일 수 있고, 새로운 컴포넌트 개발 시에도 동시 개발이 가능하기 때문에 생산성 향상을 기대할 수 있는 패턴이다.

4. 아키텍처 패턴의 적용 방법 제안

그 동안 모바일 어플리케이션 영역에서도 소프트웨어 공학적인 접근법을 이용하여 생산성을 높일 수 있음에도 불구하고 이를 적용하지 않고 있다. 그 이유는 주로 소프트웨어 공학적인 방법을 적용하면 성능이 느려진다는 생각 때문이다. 하지만, 실제로 본 논문에서의 아키텍처 패턴처럼 구조를 모듈화 하여 작성하더라도 성능에서 느려지지 않게 하는 방법을 생각해 볼 수 있다.

모바일 게임 영역에서는 클래스 개수가 많아지면 이에 따라 개발된 제품의 속도가 느려진다고 한다. 따라서 모듈화 된 구조를 합쳐서 개수를 줄일 수 있어야 하는데, 두 가지 방법을 생각해 볼 수 있다. 첫째는 프로그래밍 언어 혹은 매크로 언어의 지원을 생각해 볼 수 있다. AOP (Aspect-Oriented Programming)[5]에서는 결합점(Join point)을 정의하여 충고(advice)에서 교차점(pointcut)을 통해 해당 부분의 코드 실행을 변경하거나 코드를 삽입할 수 있다. 하지만, 하나의 관점(aspect)이 하나의 정적 클래스처럼 컴파일 되므로 이를 모바일 어플리케이션 영역에서 사용한다면 일반 객체지향 프로그래밍과 비교하여 속도면에서 향상을 기대하기 힘들다. 따라서 AOP의 아이디어만 인용하여, 해당 관점의 충고 코드를 호출하도록 삽입하는 부분을, 아키텍처 패턴의 객체들을 모두 인라인으로 삽입하여 컴파일하는 방법을

생각해 볼 수 있다.

이 방법이 가능한 이유는, 해당 클래스들이 사용하는 자료 명이나 메소드 이름이 명확히 구분되도록 패턴을 정의하였기 때문이다. 즉, 실제 구현할 때는 따로 구현하고 컴파일 직전에 직조(weave)하여 하나로 합치는 방법을 생각해 볼 수 있다.

주의할 점으로는 반드시 필요한 구조대로 작성해야 하고, 이들이 합쳐질 부분을 정확히 명시하여 필요한 부분에 알맞은 코드가 삽입되도록 해야 한다는 점이다. 예를 들어, 모바일 어플리케이션에서 컨트롤러는 상태 관리 쓰레드와 키 이벤트 핸들링과 같이 여러 가지 일을 담당하게 되는데, 이 때 키 이벤트를 핸들링하는 모듈을 따로 작성하여 컴파일 직전에 컨트롤러 모듈에 삽입되도록 할 수 있을 것이다.

두 번째는, 모바일 어플리케이션 영역에서 패턴을 이용한 개발 시 지금까지 개발해오던 것처럼 소수의 클래스가 실제로 존재하면서, 개발 툴의 도움을 얻어 이를 구조화하여 편집할 수 있도록 하는 것이다. 구조화하여 편집이 가능한 이유는 기존에 분류한 아키텍처 패턴에 따라 구조화 할 수 있기 때문이다. 즉, 하나의 긴 클래스를 기능 및 목적에 맞도록 부분 편집 및 부분 재사용이 가능하도록 툴에서 지원하는 방법이라 할 수 있다.

위에서 소개한 방법들의 공통된 아이디어는 모듈화 하여 개발한 어플리케이션을 실제 구동할 때, 모듈이 아닌 것처럼 하기 위함이다. 즉, 클래스 로딩 시간이나 상호간에 메소드 호출 시간 등을 줄이면 아키텍처 패턴을 효율적으로 사용할 수 있을 것이다.

5. 향후 연구 및 결론

본 논문에서는 모바일 게임 영역에서의 소프트웨어 아키텍처 패턴만을 다루었다. 클라이언트 측에서 게임 이외에 비즈니스 어플리케이션 또는 모바일 어플리케이션 서버 측에서 사용가능한 아키텍처 패턴에 관해서도 생각해 볼 수 있다. 또한 이렇게 도출한 아키텍처 패턴을 실제로 쉽게 사용할 수 있도록 개발 툴에서 패턴 중심의 개발이 가능하도록 지원할 수 있어야 한다.

이렇듯 모바일 어플리케이션 영역별로 적용 가능한 소프트웨어 아키텍처 패턴을 분류해 놓는다면, 동일 영역의 다른 제품 개발 시에 소프트웨어 아키텍처 수준에서의 재사용이 가능할 것이고 이는 곧 생산성을 향상시켜 빠른 시장 출시(Fast Time-To-Market)를 만족시킬 수 있을 것이다. 또한 개발 시에 동시 개발이 가능하며 각 부분별로 전문적인 개발이 가능해 지기 때문에 이러한 점도 생산성 향상에 도움이 된다. 그리고 구조화 되어서 높은 응집성을 가지고 인터페이스를 통해 낮은 결합성을 가지는 모듈들로 전체 구조가 관리됨으로써 유지보수 측면에서도 이득을 볼 수 있으리라 기대된다.

참고 문헌

[1] Heather Schneider, Valentino Lee and Robbie Schell, Introduction to Mobile Application Architectures: Mobile Application Architectures, Reading, Prentice Hall, April 2004.

[2] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Reading, Addison-Wesley, August 1995.

[3] Architectural pattern, http://en.wikipedia.org/wiki/Architectural_pattern_%28computer_science%29

[3] David Garlan and Mary Shaw, An Introduction to Software Architecture, School of Computer Science, Carnegie Mellon University, Pittsburgh, January 1994.

[4] Desmond F. D'Souza and Alan C. Wills, Objects, Components: The Catalysis(SM) Approach, and Frameworks with UML, P452, Addison-Wesley, October 1998.

[5] Aspect-Oriented Programming, http://en.wikipedia.org/wiki/Aspect-oriented_programming