Architecture-Based Measuring of Software Extensibility

Jungho, Kim
School of Engineering Information and Communication University
E-mail : kimjh@skcc.com

Sungwon Kang
School of Engineering Information and Communication University
E-mail : kangsw@icu.ac.kr

요 약

시스템을 결정하는 품질 요소에는 여러 가지가 있으나 그 중에서도 유지보수성(Maintainability)이 높은 시스템을 만드는 것은 쉬운 일이 아니다. 또한 유지보수성이 높은 시스템인지 여부를 시스템 개발이 완료 전에 측정하는 것도 어려운 문제이다. 모든 품질 요소가 그렇겠지만 시스템을 구성하는 소프트웨어 아키텍처 수준에서 시스템의 품질을 명확히 측정해내지 못하고 시스템 개발 이후에 측정한다면 품질 향상을 위한 시스템 수정은 생각할 수도 없는 문제가 될 것이다. 이 논문은 유지보수성 중에서도 가장 중요한 기능확장성(Extensibility), 즉 기존 시스템에서 신규 서비스를 추가하려 할 때 기존의 서비스에 영향을 미치지 않으면서 비용 및 기간을 최소화하여 새로운 기능을 추가할 수 있는 품질 속성을 소프트웨어 아키텍처 수준에서 비교하는 방법을 정의하여 제시한다.

본 논문에서는 소프트웨어 아키텍처 중에서도 기능확장성에 가장 큰 영향을 미치는 모듈 뷰를 기준으로 기능확장성을 측정할 것이나 모듈 내부의 변경을 통한 기능확장성은 고려하지 않고 새로운 모듈의 추가로 인한 기능확장성을 고려하였다. 본 논문은 기 수립된 소프트웨어 아키텍처의 모듈 뷰가 가지는 고유한 기능확장성을 측정하는 함수를 제시하고 여기에 새로운 모듈이 추가될 때 변화한 소프트웨어 아키텍처의 기능확장 effort 함수를 제시한다. 이 두 함수를 통해 두 개의 대안 소프트웨어 아키텍처 모듈 뷰에서 어떤 것이 더 기능확장성이 있는지 판단할 수 있게 한다.

우리는 이를 검증하기 위해서 기능확장성이 좋다고 제시되고 있는 디자인 패턴(계층패턴과 Broker 패턴)을 통해 본 논문의 이론을 정립하고 그 효용성을 입증하였다. 따라서 소프트웨어 아키텍처 수준에서 기능확장성이 얼마나 가능한지 서로 비교 가능한 수치를 제시하였고 이 수치를 통해 실제 개발이 완료되기 전 시점에 시스템의 기능확장성을 명백히 측정할 수 있게 하여 시스템 기능 확장에 들어가는 많은 비용을 줄일 수 있다는 것을 보여준다.

1. 서론

소프트웨어 아키텍처는 소프트웨어 시스템의 동적, 정적 구조를 설명한다. [5] 이러한 구조를 기준으로 여러 가지 품질 속성을 모델링할 수 있으며 시스템으로 개발될 수 있다. 그러나 시스템의 품질 요소가 좋은 아키텍처를 모델링하는 것은 쉬운 일이 아니고 품질 요소가 좋다는 것을 아키텍처 수준에서 정확히 측정하는 것도 쉬운 일이 아니다. 소프트웨어 시스템의 품질을 측정하는 가장 일반적인 방법은 시나리오 기반의 아키텍처 검증 방법인 ATAM을 사용할 수 있다[3].

소프트웨어 주기에서 보면 소프트웨어 개발 이후의 유지보수 비용이 50% 이상이라는 사실은 널리 알려진 것이다. 이런 비용이 발생하는 것은 시스템 개발 시에는 고려하지 못했던 변화가 시스템 개발 이후에 필요하였기 때문이고 이를 효과적으로 적용할 수 있는 아키텍처가 부족하였기 때문이다. 또한 최근의 소프트웨어 개발의 핵심적인 쟁점이 되는 것은 신규 서비스(기능)을 얼마나 빠르게 추가할 수 있는가 하는 것이다. 이런 이슈가 중요한 이유는 최근의 전사 소프트웨어 개발 프로젝트에서 고객이 급변하는 경영환경에 효과적이고 빠르게 적응하는 시스템을 원하기 때문이다. 예를 들어 유명한 운동선수가 무릎 부상으로 경기에 결장하게 되면 관절 부상과 관련된 보험이 많이 팔리게 된다. 이 시장은 갑자기 열리면서 급하게 사라져 버린다. 이 때 가장 먼저 보험을 설계하여 출시하는 보험사가 이 시장을 점유하게 되는 것이다. 이처럼 빠른 변화에 민감하고 신속하게 적응할 수 있는 소프트웨어 시스템이 필수적이고 이런 기능을 가능하게 하는 소프트웨어 아키텍처가 필수적인 것이다.

소프트웨어 시스템의 변경용이성(Modifiability) 만을 위한 시나리오 기반의 측정 방법이 연구되었다. 본 논문에서는 소프트웨어 아키텍처 수준[2] 에서 기능확장성 (Extensibility)을 시나리오 기반이 아니고 수치를 기반으로 한 측정 방법을 제시하고자 한다.

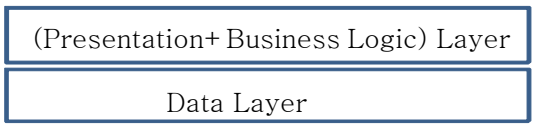
2.아키텍처 수준에서 기능확장성 판단 요인

2장에서는 기능확장성이 좋다고 판단되는 계층 패턴과 브로커 패턴을 소개하여 왜 이러한 패턴을 사용하면 기능확장성을 높일 수 있는지 그 이유를 알아본다[1].

2.1 Layer 패턴 아키텍처와 기능확장성의 관계

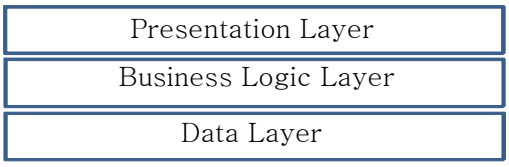
일반적으로 기능확장성을 좋게 만드는 아키텍처 스타일은 계층 스타일이다. 여기서 가장 간단한 계층 스타일을 적용해보자. 다음은 클라이언트/서버 구조에서 서버에서 클라이언트 인터페이스와 비즈니스 로직을 구분하여 기능확장성을 높인 기본적인 아키텍처의 비교 예제이다.

예제 1.



인터페이스/서비스 계층은 클라이언트에서 호출되어 처리되는 기능과 비즈니스 로직을 모두 포함하는 계층이고 데이터는 요청되는 서비스가 사용하는 데이터를 담고 있는 계층이다.

예제 2.



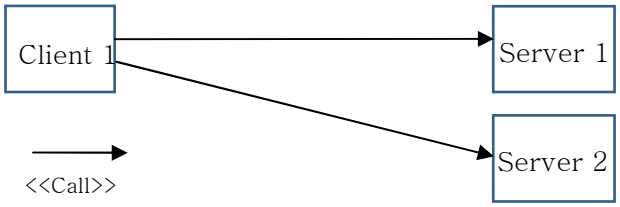
인터페이스 계층은 클라이언트와 호출되어 처리되는 기능을 보여준다. 또한 서비스 계층은 서비스 로직을 담고 있는 계층이고 데이터는 이 서비스 계층이 사용하는 데이터를 담고 있는 계층이다. 위의 예제 2를 보면 separate of concern의 원칙에 따라 계층을 구분할 경우 수정이나 추가될 부분에 대한 크기(size)가 줄어들어 기능확장성을 높일 수 있다는 것을 알 수 있다.

2.2 Broker 패턴과 기능확장성의 관계

너무 많은 분해로 인해 아키텍처 구성요소의 사이즈를 최대한 적게 만드는 것이 오히려 기능확장성에 해가 되는 예도 있다. 예를 들어 작은 서비스 단위로 서버의 컴포넌트를 구성한 시스템이 있다고 하자. 이 시스템은 다양한 사용자의 서비스 요구사항을 작은 서버 사이드의 컴포넌트 조합으로 제공할 수 있다. 그러나 이런 서버 컴포넌트가 많아짐으로 인해서 더욱 많은 서버 연결이 필요하게 될 것이다. 또한 복잡한 시스템일 경우 사용자가 필요한 컴포넌트가 다양한 환경에 존재할 수 있으므로 이런 컴포넌트를 사용하려는 클라이언트의 어플리케이션은 오히려 더 복잡한 개발 절차를 거쳐야 서비스를 추가할 수 있을 것이다. 이런 문제를 해결하기 위해서 등장한 것이 브로커 패턴이다.

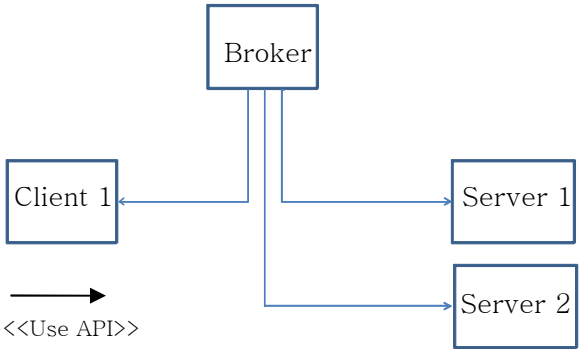
브로커 패턴을 사용하지 않은 상태로 기능이 확장되는 형태의 아키텍처는 예제 3과 같다.

예제 3.



이 상태에서 클라이언트 어플리케이션(Client)이 추가되고 서버 사이드의 서비스 컴포넌트(Server)가 추가될 경우에 아키텍처는 다음과 같이 변화하게 된다.

예제 4.



위의 예제 4와 같이 서버 사이드의 서비스 컴포넌트의 사이즈는 줄었지만 신규 클라이언트 어플리케이션을 추가하려고 하면 서버 사이드의 서비스 컴포넌트 속성과 API를 파악하여 호출하여야 한다. 서비스 컴포넌트의 운영환경이 다를 경우(OS가 다르거나 API 방식이 다를 경우)에는 클라이언트 어플리케이션을 추가하기가 더욱 힘들어진다.

이와 같은 경우에 초기 개발 비용이 추가되더라도 다음과 같은 브로커 아키텍처 패턴을 사용하게 되면 서버 사이드의 서비스 컴포넌트를 연결하기 위해 알아야 하는 많은 정보를 위임할 수 있게 된다. 하지만 새로운 클라이언트 어플리케이션이나 새로운 서버 사이드의 서비스 컴포넌트가 추가되더라도

브로커에 등록만 하게 되면 쉽게 이 기능들을 사용할 수 있게 되는 것을 알 수 있다.

3.아키텍처 수준에서의 기능확장성 측정 함수

위의 예제에서 보듯이 추가될 모듈의 크기뿐 아니라 추가되는 아키텍처 요소가 발생하는 관계의 복잡성(interaction complexity)이 중요한 요소가 됨을 알 수 있다. 그러므로 기능확장성을 아키텍처 수준에서 판단할 수 있는 요인은 아키텍처 구성요소의 사이즈와 구성요소간의 관계의 복잡성이다. 또한 사이즈와 연결의 복잡성 사이에도 경우에 따라 가중치가 부여됨을 알 수 있다. 따라서 이들의 가중치는 기존의 규모 산정(Function Point)에 사용하였던 가중치를 참조하여 사용하기로 한다[7].

본 논문에서는 기 수립된 고유의 소프트웨어 아키텍처 기능확장성을 측정하는 함수(Ext)를 제공하고 여기에 새로운 모듈(기능)을 추가할 때 발생하는 기능확장 effort를 계산하는 함수 Ext_effort)를 제공한다.

3.1 고유의 기능확정성 측정 함수

기능확장성 함수는 기 수립된 S/W 아키텍처의 고유한 확장 용이성을 측정하는 것이다. 위에서 살펴보았듯이 모듈 구조가 세분화되면 될 수록 변경에 영향을 받는 size도 줄고 추가되는 부분도 줄어들어 기능확장을 높이는 장점이 있다. 따라서 이것의 인자는 기능확장성과 비례관계임을 알 수 있다. 또한 위에서 말했듯이 모듈간의 연관관계가 복잡하면 새로운 모듈을 추가할 때 상대적으로 많은 부분에 대한 고려해야 하고, 따라서 기능확장성과 반비례 관계가 되는 것을 알 수 있다. 따라서 S/W 아키텍처 고유의 기능확장성은 모듈간의 decomposition level과 decompose된 모듈과의 연관관계를 나타내는 relation의 복잡성의 함수로 결정된다. 여기에서 모듈이 하나이고 연관관계가 없는 아키텍처는 기능확장성 함수에 대입하지 안기로 한다. 위와 같은 전제를 기준으로 소프트웨어 아키텍처가 가지는 고유의 기능확장성 함수는 다음과 같다.

$$Ext(SA) = \text{decom} / \sum \beta\, r$$

SA : 소프트웨어 아키텍처
decom: 기존 아키텍처에서 분할된 모듈 갯수
r : 소프트웨어 아키텍처가 가지는 고유한 relation $\beta\, r$: relation이 가지는 복잡도 (FP 기준)

3.2 기능확장 effort 측정 함수

기능확장 effort 함수는 기 수립된 S/W 아키텍처에 새로운 기능을 추가할 때 들어가는 effort를 측정하는 것이다. 신규로 추가되는 모듈의 사이즈는 작을 수록 기능확장 effort가 줄어드는 관계를 가진다.(반비례 관계) 또한 신규로 추가될 모듈간의 연관관계가 복잡하면 복잡할 수록 기능확장 effort가 많아지는 관계가 있다. (비례 관계) 기능확장 effort 함수는 신규로 추가될 모듈의 size와 신규로 추가될 모듈과의 연관관계를 나타내는 relation의 복잡성의 함수로 결정된다. 따라서 소프트웨어 아키텍처에 새로운 기능을 추가할 때 발생하는 effort를 측정하는 기능확장 effort 함수는 다음과 같다.

$$Ext\_effort(SA, N) = (\alpha * size(N)) + (\sum \beta\, r(N))$$

SA : 소프트웨어 아키텍처
N : 신규로 추가될 기능,
α : size의 가중치, $size(N)$: 신규로 추가될 서비스 모듈의 크기, 즉 신규 서비스를 구현할 때 소요되는 effort(LOC 기준)
$r(N)$: 서비스 N 모듈이 신규 추가될 때 발생하는 relation (FP 기준) $\beta\, r$: relation을 구현하는데 소요되는 effort (FP 기준)

4. 검증

3장에서 제시된 두개의 함수를 검증하기 위해 브로커 패턴의 예제에 적용해본다. 브로커 패턴을 사용할 경우 신규 클라이언트와 서버를 추가할 때 수월하다는 것은 이미 다 알려진 사실이다. 따라서 본 논문에서 제시하는 2개의 함수를 브로커 패턴에 적용하여 그 효용성을 검증하겠다.

예제 3에 고유한 기능확장성을 적용한 결과는 다음과 같다.

$Ext\ (SA1) = decom/\sum\beta r$

$= 3 / (4+4) = 3/8 = 0.375$

(decom : 3 modules, relation : 2 uses (EO of FP))

반면에 예제 4인 브로커 패턴에 적용한 결과는 다음과 같다.

$Ext\ (SA2) = decom/\sum\beta r$

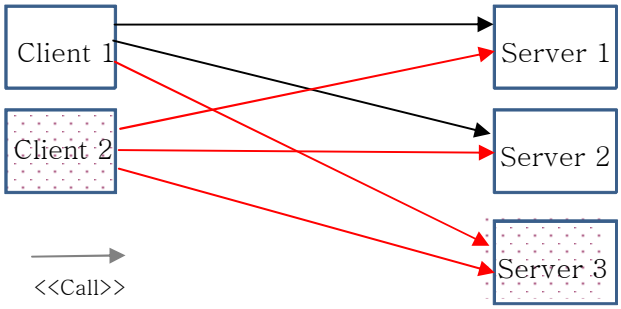
$= 4 / (4+4+4) = 4/12 = 0.333$

(decom: 4 modules, relation : 3 uses (EO of FP)

이 결과로 보면 예제 4인 브로커 패턴을 사용한 경우보다 그것을 사용하지 않은 예제 3인 경우가 고유의 기능확장성이 높은 것으로 나타났다.

다음으로 예제 3과 예제 4에서 기능이 확장된 형태의 아키텍처를 기준으로 기능확장 effort 함수를 측정해 보겠다. 예제 3의 아키텍처에서 신규 클라이언트와 서버가 추가된 아키텍처는 다음 예제 5와 같다. 이 경우에 발생하는 기능확장 effort를 측정한 값은 다음과 같다.

예제 5.



$Ext\_effort(SA2, Client\ 2) + Ext\_effort(SA2, Server\ 2)$

$= (\alpha * size(Client2)) + (\sum\beta r(Client2)) +$

$(\alpha * size(Server2)) +$

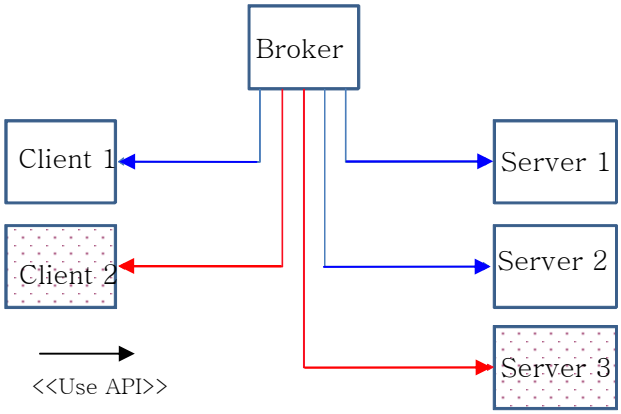
$(\sum\beta r(Server2))$

$= ((1*10) + 4 + 4) + ((1*10) + 4 + 4) = 36$

(size: LOC, weight: 1(basic) relation : 4 uses)

예제 6은 예제 4의 브로커 패턴에서 신규 클라이언트와 서버가 추가된 형태의 아키텍처를 표현한 것이다. 이것의 기능확장 effort 함수로 측정한 값은 다음과 같다

예제 6.



$Ext\_effort(SA1, Client\ 2) + Ext\_effort(SA1, Server\ 2)$

$= (\alpha * size(Client2)) + (\sum\beta r(Client2)) +$

$(\alpha * size(Server2)) + (\sum\beta r(Server2))$

$= ((1*10) + 4) + ((1*10) + 4) = 28$

(size: LOC, weight : 1 relation : 2 uses)

위의 결과를 보면 브로커 패턴을 사용하는 소프트웨어 아키텍처의 모듈 뷰가 고유한 기능확장성은 그렇지 않은 것에 비해 떨어지지만 새로운 기능을 추가할 때 발생하는 effort는 브로커 패턴을 사용한 소프트웨어 아키텍처가 그렇지 않은 것에 비해 높다는 것을 알 수 있다.

5. 결론

데이빗 파나스의 논문에서는 소프트웨어의 확장성을 위해서 ‘ 사용’ 관계의 사용과 가상 머신을 이용한 계층뷰를 사용하라고 주장하였다[6]. 그 이후에 많은 논문이 이를 뒷받침하는 이론을 만들어 연구를 하였다. 최근에는 객체 내부에 대한 coupling, cohesion을 통한 객체 지향 시스템의 변경 용이성을 측정한 논문[9]들에서 소프트웨어 구조를 기준으로 변경용이성을 판단하는 논문이 등장하고 있다[8].

이 논문은 소프트웨어 아키텍처를 수립한 직후, 두 개의 대안 아키텍처를 비교하여 어떤 소프트웨어 아키텍처가 기능확장성 측면에서 우수한지 판단할 수 있도록 한다. 기존의 기능확장성이 추상적인 시나리오에 의존하여 주장되었거나 모듈의 내부 변화에 의존하였지만 본 논문에서는 기존의 아키텍처에서 추가되는 서비스(기능)을 중심으로 기능확장성을 수치적으로 제시할 수 있는 방법을 연구하였다.

현재 개발된 기능확장성 측정 함수들은 보다 많은 예제를 통해 검증, 수정되어야 할 것이다. 따라서 MVC 패턴등의 이미 증명된 패턴들을 적용하여 이 함수의 정확성을 검증할 것이다.

따라서 소프트웨어 아키텍처 측면에서 기능확장성을 명확히 측정하여 소프트웨어 개발 전에 기능확장성이 우수한 소프트웨어 아키텍처를 설계할 수 있도록 할 것이다.

[참고 문헌]

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern Oriented Software Architecture*, John Wiley & Sons, 1996.

[2] Paul Clements, Len Bass, David Garlan,, *Document Software Architecture: Views and Beyond,* Addison Wesley Longman, 2003.

[3] Rick Kazman, Mark Klein, Paul Clements, *ATAM: Method for Architecture Evaluation* TECHNICAL REPORT, CMU/SEI-2000-TR-004, ESC-TR-2000-004, 2004.

[4] Bass, L., Clements, P., Kazman, R., *Software Architecture in Practice 2nd Edition.* Addison Wesley Longman, 2003.

[5] Parnas, D. L., *Designing software for ease of extension and contraction* IEEE Trans. Software. Eng. SE-5(2), 128-138 (Mar. 1979).

[6] J. Brian Dreger, *Function Point Analysis*, Prentice Hall, 1989.

[7] Amnon H. Eden, Tom Mens, *Measuring Software Flexibility*. IEE Software, Vol. 153, No. 3, pp. 113- 126, Jun. 2006,

[8] Chae H. S., Kwon Y. R., Bae D., *A cohesion Measure for Object-Oriented Classes,* Software-Practice & Experience, V.30 n.12, p.1405-1431, Oct. 2000.