매트릭을 이용한 아키텍쳐 안정성 평가

이현주⁰ 박찬진 강유훈 김택수 우치수
서울대학교 소프트웨어공학 연구실
{gradys⁰,cjpark, rmaker, dolicoli, wuchisu}@selab.snu.ac.kr

Evaluating Software Architectural Stability by Metrics

Hyunjoo Lee⁰ ChanJin Park Yoohoon Kang Taeksu Kim Chisu Wu
School of Computer Science and Engineering, Seoul National University

요 약

아키텍쳐(Architecture)는 프로젝트 초기에 결정된 설계 결정사항을 기재해 놓은 산출물이고 프로젝트 관련자(Stakeholder)간 의사소통의 수단이다. 아키텍쳐가 안정되어야 향후 진화(Evolution) 과정 중에 시스템을 이해하고 예측, 관리하는 것이 쉬워지고 또한 기본 구조를 변경하지 않고 여러 가지 기능을 추가할 수 있다. 아키텍쳐 안정성(Architecture Stability)이란 진화과정 중에 발생하는 변경들을 견디는 정도를 말하고 변경은 적응변경(Adaptive Changes), 교정변경(Corrective Changes), 완전변경(Perfective Changes)을 포함한다. 그런데 진화 과정 동안에 변경들은 필연적으로 발생하게 되고 그로 인해 결정된 아키텍쳐는 본래의 모습을 그대로 유지할 수 없게 된다. 따라서 진화 과정시 아키텍쳐의 안정성을 측정하고 향후 변경에 대한 대비가 필요하다. 본 논문은 변경의 크기, 변경의 횟수, 변경의 종류와 아키텍쳐 불안정성의 관계를 버전별로 연구하고 그와 관련된 매트릭을 제안한다. 매트릭을 실제 프로젝트(Ant,JDT)에 적용하고 측정된 결과를 통해 아키텍쳐 안정성을 살펴봄으로써, 향후 안정성을 고려하여 아키텍쳐를 관리하고 개발하는데 도움을 줄 수 있을 것이다.

1.서론

아키텍쳐가 불안정하면 아키텍쳐가 생각했던 것과는 다르게 진화될 가능성이 있으며 심한 경우 시스템 구조의 붕괴를 가져올 수 있다. 그래서 이를 막기 위해서는 아키텍쳐의 안정성을 예측하는 것이 중요하다. 특히 시스템의 목적이 오랫동안 지속되는 것이라면 미래 변화에 대해 수용할 수 있게 진화되어야 하기 때문에 안정성은 아키텍쳐를 평가하는데 있어서 중요한 품질속성이 될 것이다.[3] 소프트웨어 아키텍쳐 안정성을 평가하는 목적은 진화의 영향(Impact Of Evolution)을 이해하기 위함이고 진화의 영향을 이해하는 것은 진화 과정시 발생한 변경의 영향을 평가하는 것이다.[2]

본 논문은 변경의 크기, 변경의 횟수, 변경의 종류와 관련된 요소를 파악하고 그와 관련된 매트릭을 제안한다. 매트릭은 소프트웨어 아키텍쳐의 안정성을 측정하는 것을 목표로 한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구를 살펴보고 3장에서는 본 논문에서 아키텍쳐와 아키텍쳐 안정성을 정의한다. 4장에서는 매트릭을 제안하고 5장에서는 매트릭을 적용한 결과를 설명한다. 마지막으로 6장에서는 결론과 향후 연구를 기술한다.

2. 관련 연구

Bahoon[3]은 아키텍쳐 안정성을 측정하기 위해서 변경을 안정성에 영향을 주는 변수로 고려하지만 주식 가격을 변경의 값으로 보고 경제적인 측면을 아키텍쳐에 적용한다. 구조적인 측면에서 안정성을 측정한 논문[4,6]으로 Bansiya[6]는 안정성을 예측하기 위해서 안정성과 관련된 9가지의 객체 지향 디자인 매트릭 값을 측정한 후 이전 버전의 매트릭 값과 현재 버전의 매트릭 값의 합을 빼서 변경의 확장성(Extend Of Change)의 값을 구한다. 이 값은 아키텍쳐 구조의 상대적인 안정성을 나타낸다. Grosser[4]는 Bansiya[6]의 논문처럼 객체 지향 매트릭 중에 응집도(Cohesion), 결합도(Coupling), 상속(Inheritance), 복잡도(Complexity)와 관련된 매트릭을 사용하지만 사례 기반 추론(Case-Based Reasoning)방법을 사용하여 유사한 데이터 집합으로부터 다른 집합의 안정성을 예측하는 방법을 사용한 점이 다르다. 이 두 논문은 매트릭을 사용하여 아키텍쳐의 안정성을 측정하지만 기존에 존재하는 디자인 매트릭을 적용하였고 본 논문은 변경에 관련된 매트릭을 모듈 레벨에서 제안한다.

3. 정의

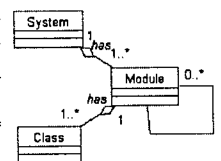
3.1 아키텍쳐

본 논문에서 아키텍쳐에 대한 정의는 아래와 같다. 이 정의는 소프트웨어 아키텍쳐의 구조적인 관점을 강조한 것이다.

소프트웨어 아키텍쳐는 모듈과 모듈 사이를 연결해주는 관계로 구성된 전체적인 구조로 시스템 진화 과정시 가이드 역할을 해주는 상위레벨의 개념이다.

3.2 모듈과 모듈 관계

모듈(Module)은 서로 연관된 클래스들의 집합으로 논리적인 개념이다. 본 논문에서 시스템은 복수개의 모듈과 모듈들의 관계로 구성된 객체지향 시스템으로 한정한다. 자바 시스템에서는 클래스들의 모임인 패키지를 모듈로 간주할 수 있다. 그림1은 UML 클래스 다이어그램으로 나타낸 구조적인 관계이다.



[그림1] 시스템 구조

그림 2는 모듈과 모듈간의 정의를 보여준다. 모듈과 모듈간의 관계(Module Dependency)(1)는 논리적인 개념으로 모듈의 하위요소인 클래스와 클래스간의 관계(Class Dependency)로 부터 유추된다. 클래스와 클래스간의 관계(2)는 연관(Association)과 상속(Inheritance) 그리고 매소드 호출(Method Call)로 정의한다. 연관과 상속은 구조적 관계이고 매소드 호출은 행위적

Module Dependency (MD) (1)

$\{ < M_1, M_2 > \mid < M_1, C_1 > \in Has \wedge$
$< M_2, C_2 > \in Has \wedge < C_1, C_2 > \in CD, M_1 \in Module$
$, M_2 \in Module, C_1 \in Class, C_2 \in Class \}$
$A\ Has\ B = A\ contain\ B, Module = A\ set\ of\ modules, Class = A\ set\ of\ classes$

Class Dependency (CD) (2)

$\{ < C_1, C_2 > \mid < C_1, C_2 > \in Assoc \vee$
$< C_1, C_2 > \in Inher \vee < C_1, C_2 > \in MethodCall \}$
$A\ \ Assoc\ \ B = A\ is\ associated\ with\ B$
$A\ \ Inher\ B = B\ is\ inherited\ from\ A$
$A\ \ Method\ Call\ B = A\ \ call\ B$

[그림2] 모듈 관계 정의

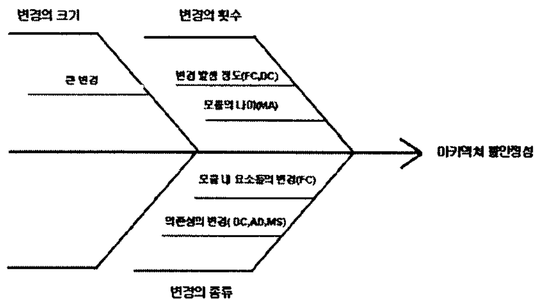
인 관계로 두 가지 관점에서 클래스와 클래스 간의 관계를 정리하였다.

3.3 아키텍쳐 안정성
Jazayeri[9] 는 아키텍쳐 안정성은 시스템 개발과정이 아키텍쳐의 변화 없이 얼마나 잘 전개되는지를 측정하는 것이라고 정의하고 Bahsoon[3]은 아키텍쳐 안정성이란 아키텍쳐 구조는 변하지 않고 투자자들의 요구사항과 환경으로 인해 전개되는 변화를 융통성 있게 견뎌낼 수 있는 아키텍쳐의 정도라고 한다. 이 논문에서는 아키텍쳐 안정성을 아래와 같이 정의하였다.

아키텍쳐 안정성이란 아키텍쳐가 소프트웨어 진화 과정 동안에 구조를 변경하지 않고 변화에 융통성 있게 견뎌내는 정도를 말한다. 변경은 적응 변경, 교정 변경, 완전 변경을 포함한다.

진화 관점의 생명주기는 초기 개발(Initial Development), 진화(Evolution), 서비스(Servicing), 정지(Phaseout), 종료(Close down)단계로 구성된다.[1] 소프트웨어의 진화 과정은 초기 개발 단계 이후 버전업(Version-Up) 되는 진화 단계를 말하고 아키텍쳐의 변경은 이 진화 과정에서 발생한다.

4. 아키텍쳐 안정성 매트릭
그림 3에서는 아키텍쳐 불안정성의 원인을 변경의 크기, 변경의 횟수, 변경의 종류로 분류하였다. 모듈과 모듈간의 변경이 크고 빈번하게 발생할 경우에 아키텍쳐가 영향을 받을 것이라는 가정에서 아래와 같이 분류하였다.



[그림3] 아키텍쳐 불안정성의 원인 (Fish Bone Diagram)

그림3에 분류된 내용을 바탕으로 표1과 같이 아키텍쳐 안정성을 측정하는 매트릭을 제안한다.

[표1] 아키텍쳐 안정성 매트릭

Metric Name	Description ($_m$:module , $_s$:System)
Module Age(MA)	$MA_m = AVE(CCD_m)$ AVE: average CCD: class created date
Frequent Change (FC)	$FC_m(I) = AC_m(I) + DC_m(I) + CIC_m(I)$ AC: The number of added Class DC: The number of deleted Class CIC: The number of class internal change I: Interval between release
Module Stability (MS)	$MS_m = \frac{Ce_m}{Ca_m + Ce_m}$ Ca(Afferent Couplings): The number of classes outside this module that depend upon classes within this module Ce(Efferent Couplings): The number of classes inside this module that depend upon classes outside this module
Acyclic Dependency (AD)	$AD_s = \frac{CD_s}{TD_s}$ CD(Cyclic Dependency): The number of Cyclic Dependency TD(Total Dependency): The number of Total Dependency
Dependency Change (DC)	$DC_m(I) = NDC_m(I)$ NDC: the number of Dependency change I: Interval between release

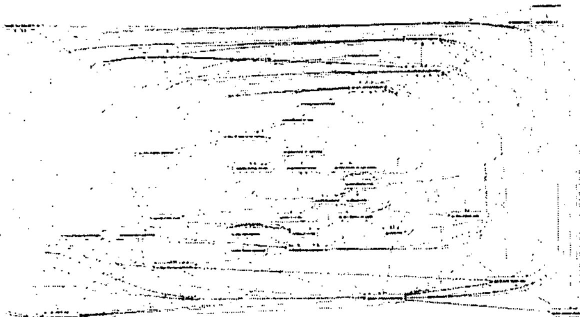
4.1 모듈 나이(Module Age)
모듈 내부 요소(Class)들의 생성 날짜의 평균을 계산한다. 소프트웨어 단위(Unit)의 나이를 정의할 때 라인의 평균을 사용한 매트릭이 있다[5].이 논문에 따르면 나이가 높을수록 빈번하게 변경이 발생했으며 이것은 곧 코드가 낡았다(Code Decay)는 증거라고 설명한다. 본 논문에서는 모듈의 나이가 크고 변경이 빈번하게 일어날수록 안정성은 안 좋다고 가정한다.

4.2 빈번한 변경(Frequent Change)
특정 간격 사이에 변경이 자주 일어나는 모듈을 계산한다. 변경이 빈번하게 발생한 모듈은 향후에도 변경이 가능할 가능성이 크다는 가정 하에 자주 변경된 모듈은 안정성이 안 좋을 수 있다.

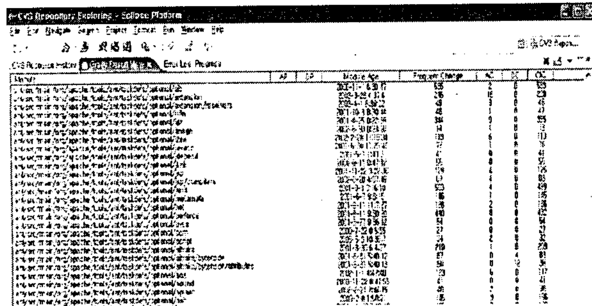
4.3 모듈 안정성(Module Stability)
한 모듈의 안정성을 계산해 준다. Martin[8]은 표1의 Ca와 Ce를 사용하여 안정성을 정의하였다 이 논문에 따라 모듈 안정성의 값이 0에 가까울수록 안정성이 좋고 1에 가까울수록 불안정하다고 가정한다.

4.4 비순환식 의존성(Acyclic Dependency)
전체 모듈 의존성 중에 순환을 이루는 의존성의 비율을 측정해 준다. 의존성이 순환적이면 결과적으로 모든 모듈을 알아야 하기 때문에 순환을 이루는 의존성이 많을수록 안정성도 안 좋아진다고 가정한다. 패키지 디자인 원칙 중에 하나인 비순환식 의존성 원칙(Acyclic Dependency Principle)을 적용한 매트릭이다.[8]

4.5 의존성 변경(Dependency change)
특정 간격사이에 모듈 관계의 변경 횟수를 계산한다. 의존성이 자주 변경되는 모듈은 안정성이 안 좋을 것이라고 가정한다.



[그림 4] 의존성 다이아그램



[그림 5] 안정성 적용 도구

5. 사례 연구

apache Ant와 ecplise JDT에 소스 부분을 추출하여 아키텍쳐 안정성 측정 매트릭을 적용해 보았다. 그림 4는 JDT 2.0 의 의존성 다이아그램 이다. 구현한 툴을 통해 그림과 같이 JDT 2.0의 의존성 관계를 확인할 수 있다. 그림 5는 Ant 1.4와 Ant1.6.2간 모듈 나이와 변경 횟수의 값을 확인한 것이다. 사례연구는 모듈과 패키지를 1:1 대응하여 측정하였다. 그런데 측정자가 원하면 여러 개의 패키지(Tree-Cut하여 원하는 레벨의 패키지를 선정 가능)를 하나의 모듈로 대응시킬 수 있다. 단, 모듈과 모듈 사이에 교집합이 없도록 한다.

우선 표2를 살펴보면, Ant 시스템 매트릭(System-Metric) 적용 결과와 JDT 시스템 매트릭 적용 결과를 비교해 볼 수 있다. JDT 경우는 버전별 AD 매트릭 값이 Ant 값보다 크다. 즉 순환을 이루는 의존성관계는 JDT가 Ant보다 많다고 볼 수 있다. 그리고 JDT가 Ant보다 FC와 DC 매트릭의 평균값이 크기 때문에 JDT에 모듈 내의 변경과 모듈과 모듈간의 변경 횟수가 Ant 보다 많다고 볼 수 있다.

결론적으로 말하자면, 표2의 결과에서 볼 수 있듯이 Ant의 아키텍쳐가 JDT 보다 더 안정적이라고 말할 수 있다. 실제로 JDT 경우는 하나의 Release와 다음번 Release사이에 생성된 버전이 많았고 패키지 구조와 패키지와 패키지 간에 의존성 변경이 많았다. 특히 Ant는 패키지 삭제와 의존성이 삭제된 패키지가 없는 반면에 JDT는 패키지 버전별 2-3개의 패키지가 삭제되었고 의존성이 삭제된 모듈도 많았다.

표3은 Ant와 JDT Release에서 모든 매트릭이 평균 이상으로 안정성이 안 좋게 나온 모듈을 정리한 결과이다. 매트릭에서 불안정하다고 판단되는 모듈이 Ant 보다 JDT가 많았다.

6. 결론

본 논문은 아키텍쳐 안정성의 원인을 분류하고 그것을 바탕으로 아키텍쳐 매트릭을 생성한 후, Ant와 JDT에 적용하고 그 결과를 설명하였다. 기존에 아키텍쳐의 안정성을 측정하는 논문

[표2] Ant와 JDT 결과 비교

((SYSTEM- METRICs))-ANT

release / metric	Ant 1.2	Ant 1.4	Ant 1.6.2
[illegible]	199	374	668
[illegible]	84748	141040	195530
[illegible]	980	1736	2929
[illegible]	1754	3086	5293
[illegible]	15	26	40
[illegible]	33	59	91
[illegible]	2.0	2.0	2.0
[illegible]	6.1	3.4	2.2
[illegible]		2000-11-15 8:21	2001-11-23 22:50
[illegible]		38.7	236.4
[illegible]		0.9	1.2

((SYSTEM- METRICs))-JDT

release / metric	Jdt 1.4.6	Jdt 2	Jdt 3
[illegible]	794	869	99
[illegible]	453321	468443	50864
[illegible]	3685	4058	5267
[illegible]	8593	9554	1091
[illegible]	37	37	41
[illegible]	199	183	21
[illegible]	18	20	2
[illegible]	9.0	7.7	9.1
[illegible]		2001-06-18 9:19	2002-03-07 9:5
[illegible]		171.8	360.
[illegible]		2.9	4

[표3] Ant와 JDT 모듈별 결과

((MODULE- METRICs))

metric / release	FC	MA	MS	DC
[illegible]	209	2000-06-30 2:48	0.5	10
[illegible]	1255	2001-08-03 5:59	0.7	14
[illegible]	54	2000-08-27 9:50	1.0	1
[illegible]	284	2001-10-17 8:37	0.5	2
[illegible]	546	2001-05-07 7:54	0.9	3
[illegible]	1724	2001-10-09 10:50	0.9	6
[illegible]	821	2001-05-23 1:46	1.0	4
[illegible]	1344	2001-05-19 10:08	0.9	4
[illegible]	2790	2001-10-06 6:08	0.9	4
[illegible]	447	2001-10-09 2:28	0.9	4

들에서는 기존의 디자인 매트릭을 그대로 활용하고 클래스 레벨에서 안정성을 측정하였지만 본 논문은 변경의 크기와 횟수 및 종류와 아키텍쳐 불안정성의 관계를 버전별로 연구하고 변경에 관련된 매트릭을 모듈 레벨에서 제안하였다. 모듈 레벨에서 발생하는 변경의 결과를 수치적으로 확인할 수 있고 모듈 구조를 개선하고자 할 경우에 도움을 줄 수 있을 것이다. 하지만 향후 매트릭의 정확성을 확보하기 위한 노력이 더 필요하고 아키텍쳐 안정성과 제시한 모든 매트릭과의 연관성에 대한 연구가 필요하다. 향후에는 이를 보안하기 위한 연구를 진행할 계획이다.

참고문헌

[1] V.T. Rajlich, K. H.. Bennett, "A Staged Model for the Software Life Cycle" *IEEE Software*, 33(7), pp. 66-71, 2000.

[2] R. Bahsoon , W. Emmerich, "Evaluating software architectures: development, stability and evolution" *Proceedings of ACS/IEEE Int. Conf. on Computer Systems and Applications*, 2003.

[3] R. Bahsoon, W. Emmerich,"Evaluating Software Architectures for Stability: A Real Options Approach" *Proceedings of the 25 th Int. Conf. on Software Engineering*, 2003.

[4] D. Grosser, H. A. Sahraoui, P. Valtchev, "An Analogy-Based Approach for Predicting Design Stability of Java Classes" *IEEE METRICS*, pp. 252-262, 2003.

[5] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus," Does code decay? assessing the evidence from change management data" *IEEE Transactions on Software Engineering*, pp 1-12, 2001.

[6] J. Bansiya,"Evaluating framework architecture structural stability" *ACM Computing Surveys (CSUR)*, 2000.

[7] M. Fayad, "Accomplishing software stability" *Communications of the ACM*, Vol. 45, No. 1,2002.

[8] R.C. Martin," Stability" *C++ Report*, 1997.

[9] M.Jazayeri," On Architectural Stability and Evolution" *Lecture Notes in Computer Science*, 2002.