효율적인 코드 인스펙션을 위한 규칙 수립에 대한 연구: A 社 정보시스템 사례

경태원*, 김상국**
*경희대학교 일반대학원 산업공학과
**경희대학교 테크노공학대학 산업공학전공 교수
twkyung@khu.ac.kr, sangkkim@khu.ac.kr

A Study on Criteria Establishment for Efficient Code Inspection

Tae-Won Kyung*, Sang-Kuk Kim**
*Dept. of Industrial Engineering, Kyung-Hee University
**College of Advanced Technology, Kyung-Hee University

요 약

본 연구에서는 소프트웨어의 완성도와 품질을 높이기 위한 방법들 중 하나인 인스펙션과 기존 검토 기법들을 정리 하였다. 그리고 코드 인스펙션(Code Inspection)을 위한 규칙들을 수립하고 실제 프로젝트에 적용함으로써 그 효율성과 효과성을 검증하였다. 본 연구를 통해 다음과 같은 결과를 얻었다. 첫째, 소프트웨어 검토 방법들에 대한 이론적 내용을 정리하였다. 둘째, 코드 인스펙션을 위한 규칙을 수립하고 사례를 통해 성과를 증명하였다. 셋째, 코드 인스펙션을 통해 정량적 데이터 관리를 수행하였다.

1. 서론

소프트웨어 개발에서 품질은 중요한 문제이다. 만들어진 소프트웨어의 품질에 따라 프로젝트의 성패가 좌우하게 된다. 결함의 존재 여부나 그 양은 품질에 많은 영향을 미치게 된다. 이때 결함을 발견하고 수정하는 작업에는 상당한 시간과 비용이 소요된다. 특히 테스트나 유지보수 단계에서 결함을 검출하고 수정하는 비용과 시간은 개발 단계 초기에 수행하는 것보다 10~100 배에 이르게 된다[3]. 이러한 이유 때문에 결함을 조기에 발견하고자 하는 시도가 끊이지 않았다. 이러한 결함 발견을 위한 대표적인 방법 중 하나가 인스펙션(Inspection)이다. 인스펙션은 1976 년 Michael Fagan 에 의해 제안되었으며[4], 인스펙션을 통해 모든 결함의 60~90% 정도를 찾을 수 있고, 인스펙션으로부터 피드백을 받음으로써 프로그래머는 같은 실수를 피할 수 있다고 주장하였다[5]. 따라서 본 연구에서는 소프트웨어의 완성도와 품질을 높이기 위한 방법들 중 하나인 인스펙션과 기존의 검토 기법들을 정리 하고자 한다. 그리고 코드 인스펙션(Code Inspection)을 위한 규칙들을 수립하고 실제 프로젝트에 적용함으로써 그 효율성과 효과성을 검증하고자 한다. 본 연구에서는 A 社의 정보시스템 구축 프로젝트를 대상으로 코드 인스펙션의 효율적 수행을 위한 규칙들을 수립하고 적용하였다.

2. 문헌 연구

2.1 소프트웨어 결함 검출 기법

2.1.1 테스트(Test)

IEEE 에서는 테스트를 “시스템이 특정 요구사항을 만족하는지 검증하고 예상했던 결과와 실제 결과의 차이를 수동 또는 자동화된 방식을 통해 식별하는 실험 또는 평가하는 프로세스이다” 라고 정의하였다[7].

테스트는 크게 세 가지 단계로 나누어진다. 즉, 단위 테스트(Unit test), 통합 테스트(Integration test), 인수 테스트(Acceptance test) 이다. 단위 테스트는 대부분 모듈을 구현한 프로그래머가 실시한다. 단위 테스트의 주요 목적은 모듈을 정확하게 구현하였는가, 예정한 기능을 제대로 발휘하는가를 점검한다. 통합 테스트는 전체 시스템을 이루는 모듈을 모아 통합적으로 시험하는 것을 말한다. 시스템이 요구된 기능을 제대로 수행하는가를 점검하고 모듈 사이의 인터페이스를 시험하는 것이 주목적이다. 인수 테스트는 완성된 제품에 대한 시험이다. 인수 테스트의 목적은 시스템이 사용될 준비가 다 되었다고 드러내 보이는 것이다[1].

2.1.2 동료검토(Peer Review)

동료검토는 일반적으로 형식 없이 검토가 필요할 때마다 동료들이 작업 산출물을 검토하는 방법을 의미한다. 정해진 검토 인원이나 제한시간, 검토리더는 없으며 동료와 의견을 나누듯이 검토를 진행한다. 동료검토의 장점은 자유로운 의견을 통해 다양한 아이디어를 얻을 수 있고, 검토 과정을 통하여 지식이 동료들에게 전파되는 것이다. 하지만 검토 리더가 정해져 있지 않아 정상적인 검토가 진행되지 않거나 일반 회의처럼 다수의 의견이나 힘있는 자의 의견에 전체의 흐름이 동조될 수 있다는 단점이 있다[2].

2.1.3 워크스루(Walkthrough)

Fewster and Graham 는 워크스루를 “가상의 입력에 대하여 원시코드의 수행을 문장 하나씩 짚어보는 작업”으로 정의하였다[6]. 또한 “시스템의 형태나 프로그램 개발 사항에 대해 개발 동료들로 하여금 조기에 오류를 확인할 수 있도록 하는 검토회의”라고 정의하고 있다. 워크스루는 일반적으로 프로그램의 소스코드에 대해 실시한다. 검토자들은 소스코드를 한 줄씩 읽어나가면서 발견한 이슈를 제기한다. 워크

스루는 데이터 정의부분, 매뉴얼, 명세 등 프로그램이 아닌 부분도 검토 대상이 된다.

2.2 인스펙션(Inspection)

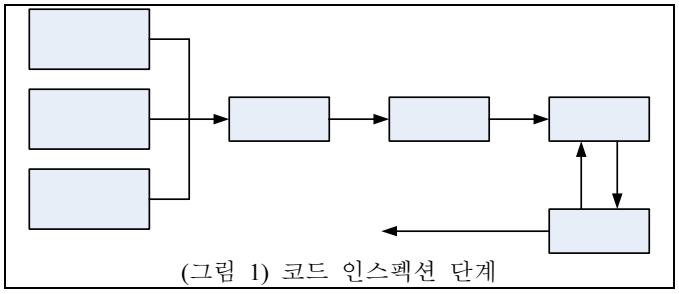
ISO 8402:1995 에서는 인스펙션(Inspection)을 “표준이나 명세서에 대한 편차와 에러를 포함한 결함을 발견하고 식별하기 위해 작업 산출물에 대해 수행하는 검사”라고 정의하고 있다[8]. 인스펙션은 소프트웨어의 품질과 생산성을 높일 수 있는 방법으로 1976 년 Michael Fagan 에 의해 제안되었다[4]. 현재까지 품질 개선과 비용 절감을 위한 대표적인 기법 중 하나로 사용되고 있다. Fagan 인스펙션은 여러 사람이 모여 소프트웨어에 관련된 문서를 체계적으로 검사하는 방법으로, 이 방법의 효용성은 이미 여러 연구와 적용에 의해서 입증되었다[9].

3. 효율적인 소프트웨어 개발을 위한 코드 인스펙션 규칙 수립

본 절에서는 A 社의 정보시스템 구축 프로젝트를 대상으로 코드 인스펙션을 위한 규칙들을 수립하고자 한다. 인스펙션 규칙 수립을 위해 참고한 자료의 구체적 명칭과 출처는 저작권 및 기관의 보안 유지를 위해 밝히지 않는다. 또한 수립된 코드 인스펙션 규칙은 본 연구에 초점이 맞춰진 것으로써 일반화 시키기 위해서는 약간의 조정이 필요함을 밝힌다.

3.1 코드 인스펙션 단계

본 연구에서는 일반적인 인스펙션 프로세스를 바탕으로 코드 인스펙션 과정을 수립하였다. (그림 1)은 코드 인스펙션 수행을 위한 과정을 보여주고 있다. 특히, 코드 인스펙션을 위한 규칙을 수립함으로써 인스펙션 활동의 효율과 성과를 높이고자 한다.



(그림 1) 코드 인스펙션 단계

3.2 코드 인스펙션을 위한 규칙 수립

코드 인스펙션 규칙 수립을 위해 다음과 같은 자료를 참고하였다.

- 발주사 프로그램 작성가이드
- 컨설팅사 개발환경 및 코딩가이드
- 개발사 품질보증팀 기준

프로젝트에 참여한 이해당사자들과 세 개 기관의 소프트웨어 개발 가이드라인을 바탕으로 본 연구에 적용하기 위한 코드 인스펙션 규칙들을 수립하였다. 또한 수립된 코드 인스펙션 규칙들은 유사한 성격을 갖는 항목별로 분류하여 8 개 카테고리 30 개 항목으로 정리하였다.

<표 3>은 코드 인스펙션 규칙들과 각 규칙들에 대한 내용을 정리한 것이다.

4. 코드 인스펙션 규칙 적용 사례 분석

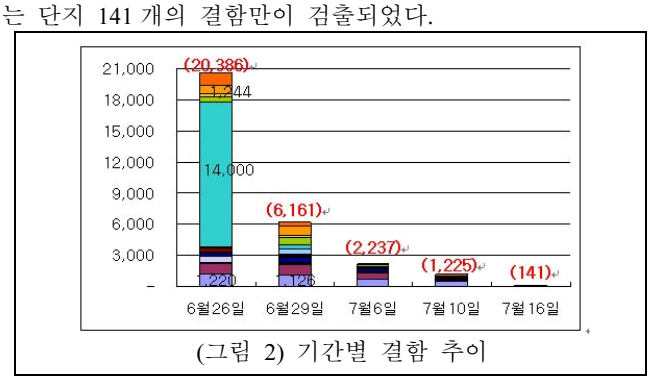
본 연구에서는 A 社 정보시스템 구축 프로젝트를 대상으로 코드 인스펙션을 수행하였다. 코드 인스펙션 참여자는 개발자, QA 팀 담당자, 그리고 프로젝트 매니저 이다. 코드 인스펙션은 사전에 수립된 규칙(표 3)을 바탕으로 수행하였다.

4.1 코드 인스펙션 결과 분석

코드 인스펙션은 공통 업무를 포함한 총 6 개 업무를 대상으로 수행하였다.

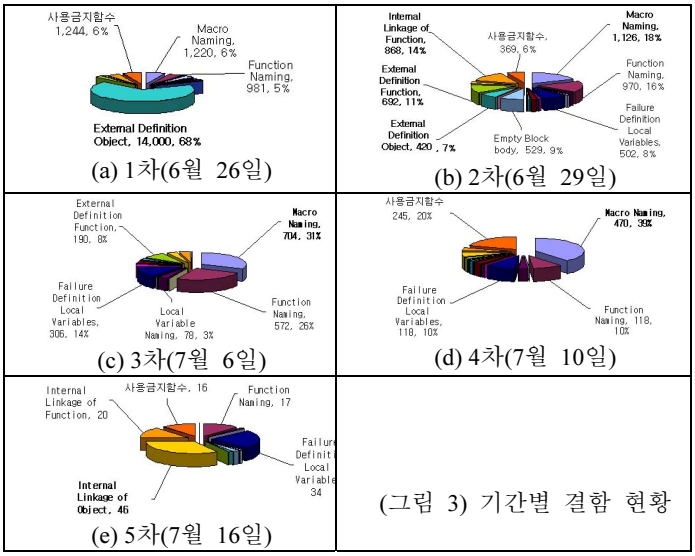
코드 인스펙션 활동은 5 회에 걸쳐 수행하였으며, (그림 2)는 코드 인스펙션을 통해 검출된 결함의 수를 기간별로 보여주고 있다.

(그림 2)에서 보는 바와 같이 1 차 코드 인스펙션을 수행한 결과 20,386 개의 결함이 발견되었다. 그러나 2 차 코드 인스펙션을 수행한 후에는 30% 정도 감소한 6,161 개의 결함이 발견되었다. 그리고 5 차 코드 인스펙션을 수행한 후에는 단지 141 개의 결함만이 검출되었다.



(그림 2) 기간별 결함 추이

아래 (그림 3)에서는 코드 인스펙션 수행 횟수에 따른 결함의 수와 종류들을 보여주고 있다.



(a) 1차(6월 26일) (b) 2차(6월 29일) (c) 3차(7월 6일) (d) 4차(7월 10일) (e) 5차(7월 16일)

(그림 3) 기간별 결함 현황

(그림 3)의 (a)는 1 차 코드 인스펙션을 수행한 결과이다. 총 20,386 개의 결함이 발견되었으며, 그 중 ‘External Definition Object’ 규칙에 위배된 결함이 14,000 개가 검출되었다. 이것은 동일한 이름의 전역 변수가 서로 다른 파일에서 중복사용 되었기 때문이다. 이러한 경우, 변수 선언 시 정적(static) 변수로 선언하거나 중복되지 않게 선언함으로써 결함 발생을 피할 수 있다. (그림 3)의 (b)는 2 차 코드 인스펙션을 수행한 결과로써, 1 차 코드 인스펙션을 수행했을 때보다 결함의 수를 약 30% 정도 줄었음을 확인하였다. 하지만 ‘Macro Naming’ 규칙에 위배된 항목 1,126 건이 검출되었다. 매크로

<표 3> 코드 인스펙션을 위한 검사항목

구분	NO	검사항목	내용
가독성 및 유지보수성 관련 규칙(8)	1	Macro Naming	매크로 이름은 모두 대문자 및 (_) 사용
	2	Function Naming	이름의 첫 글자는 영문 소문자로 시작하며, 그 이후의 단어의 첫 글자는 대문자로 시작 단어를 구분하는데 밑줄(_) 사용, Function 이름은 계층적 구조가 나타나도록 명명
	3	Enum Constants Naming	Enum Constants 는 정의된 규칙(모두 대문자)에 맞게 기술
	4	Global Variable Naming	전역변수의 첫 글자는 소문자, 그 이후는 숫자, _, 영문자(대소문자) 포함
	5	Local Variable Naming	첫 글자는 소문자로 시작, 그 이후는 숫자, _(under bar), 영문자(대소문자)
	6	File Naming	파일 이름의 첫 글자는 영문대문자, 소문자사용, 확장자(C,c,H,h)는 대소문자 구분하지 않음
	7	#define or #undef within a block	블록안에 #define 이나 #undef 가 존재하지 않아야 함
	8	File Comments	첫 라인은 /* 로 시작하여야 함(헤더주석)
Dead Code 관련 규칙(2)	9	Failure Definition Local Variables	사용하지 않은 지역변수(파라미터 포함)는 제거되어야 함
	10	Non-Null statements	무의미한 문장은 제거, 모든 널(Null)이 아닌 문장들은 둘 중 하나여야 함
잠재적 에러 관련 규칙(8)	11	Default in Switch	Switch Statement 의 마지막 절은 default 로 명시해야 함
	12	Floating Point Comparison	부동소수점(float, double)의 비교연산은 금지해야 함.
	13	Uninitialized Pointer	모든 포인터는 초기화 되어야 함.
	14	Variable Initialization	모든 변수는 초기화 되어야 함.
	15	Null pointer Assignment	모든 대입문은 유효한 값을 가져야 함.
	16	Assignments in boolean expression	조건문에서 대입연산을 배제(true or false) 함.
	17	Braces of loop body	For 문에서 Braces{} 사용을 권장함.
	18	Three expressions of a for statement	For 문의 세 수식들은 단지 루프제어에만 관계해야 함.
제어 에러규칙(4)	19	Unreachable Code	접근할 수 없는 Code 는 제거 해야 함
	20	Goto Statement	비구조화 GOTO 문은 사용하지 않아야 함.
	21	Empty Block body	빈 Block 문장들(if, for, while, do 의 body)의 기술을 지양함.
	22	Loop Counter type	Loop Counter 는 Signed value 를 사용해야 함.
성능 저하 관련 규칙(1)	23	Debug Statement	Console(printf())로 표현되는 문장의 사용을 지양함.
저장관리 에러규칙(1)	24	Dynamic heap Memory	Dynamic Heap Memory 할당을 최소화(함수 calloc, malloc, realloc, free 의 사용을 최소화)
인터페이스(5)	25	Number of Arguments and Parameters	함수에서 정해진 인자(argument)들의 개수는 파라미터들의 개수와 일치해야 함.
	26	External Definition object	Static 이 아닌 object(글로벌 변수)는 중복되어서는 안됨.
	27	External Definition function	Static 이 아닌 함수명은 중복되어서는 안됨.
	28	Internal linkage of object	정적 저장 분류 지정자 static 은 내부 연결을 가지는 객체 또는 함수들의 선언 및 정의 시에만 사용해야 하며 그 이외의 경우에는 반드시 extern 을 명시하여야 함.
	29	Internal linkage of function	
정보보안 규칙(1)	30	사용금지 함수 준수	39 개 함수로 세부사항은 금지함수 참조(발주사 프로그램 작성가이드 참조)

의 이름은 모두 대문자를 사용한다는 규칙에 따라 재작업(Rework)을 수행하였다. 3 차와 4 차 코드 인스펙션 활동을 통해 결함을 지속적으로 제거하였으며, 5 차 코드 인스펙션 수행 후에는 141 개의 결함만이 검출되었다. 대표적인 결함으로는 'Internal Linkage of Object' 규칙을 위반한 항목이 46 개, 'Failure Definition Local Variables' 규칙을 위반한 항목이 34 개, 'Internal Linkage of Function' 규칙을 위반한 항목이 20 개 이다. 'Internal Linkage of Object' 와 'Internal Linkage of Function' 관련 결함을 예방하기 위해서는 내부 연결을 가지는 개체나 함수를 선언할 때 'static' 키워드를 사용하고, 그 이외의 경우에는 반드시 'extern' 키워드를 사용함으로써 결함을 피할 수 있다. 또한 이러한 항목들을 점검할 때 해당 헤더 파일이 정상적으로 헤더파일 디렉토리에 포함되어 있는지 여부를 확인해야 한다. 'Failure Definition Local Variables' 규칙 위반으로 발생된 결함은 사용하지 않는 지역변수를 제거함으로써 예방할 수 있다. 사용하지 않는 지역변수들은 메소드(Method)가 실행될 때 메모리 및 실행 속도에 악영향을 줄 수 있기 때문에 반드시 제거해야 한다.

이처럼 본 연구에서는 3 주 동안 5 회에 걸쳐 코드 인스펙션을 수행하였으며, <표 4>는 코드인스펙션 활동의 대상이 된 업무별 전체 코드의 수, 파일 수, 매크로의 수, 함수의 수, 그리고 변수의 수를 정리한 것이다.

<표 4> 시스템 통계

업무명	Lines of Code	File	Macro	Function	Variable
공통 업무	47,061	187	403	485	219
A 업무	14,624	46	156	113	46
B 업무	87,850	206	784	840	429
C 업무	207,427	628	1,717	2,084	628
D 업무	15,736	46	174	142	85
E 업무	30,869	82	272	299	83
계	403,567	1,195	3,506	7,469	8,959

5 회에 걸쳐 6 개 업무 1,195 개 파일에 대해 수행한 코드 인스펙션 결과 결함이 포함된 파일은 하나도 발생하지 않았다. <표 5>에는 업무별 코드 인스펙션을 수행한 파일들과 결함이 포함된 파일의 수를 정리하였다.

<표 5> 파일 분석 결과

업무명	Successful files	Erroneous Files
공통 업무	187	0
A 업무	46	0
B 업무	206	0
C 업무	628	0
D 업무	46	0
E 업무	82	0
계	1,195	0

현재 정보시스템 개발이 종료된 것이 아니기 때문에 개발 과정에서 오류나 결함이 발생할 수 있다. 따라서 산출물에 대한 지속적인 코드 인스펙션 활동과 모니터링 활동이 필요할 것으로 본다.

5. 결론 및 향후 연구방향

인스펙션은 그 효과가 널리 알려져 있음에도 불구하고 국내에서 체계적으로 수행된 사례는 거의 없다. 따라서 본 연구에서는 인스펙션과 기존의 검토방법들을 정리함으로써 소프트웨어 개발에 관련된 이해당사들에게 인스펙션의 이론적 배경을 제공하고, 인스펙션의 중요성과 필요성을 전달하고자 하였다. 또한 코드 인스펙션을 위한 규칙을 수립하고 실제 프로젝트에 적용함으로써 그 성과를 분석하였다.

본 연구는 다음과 같은 측면에서 의의를 찾을 수 있다.

첫째, 소프트웨어 검토 방법들에 대한 이론적 내용을 정리한 점이다.

인스펙션은 소프트웨어의 품질을 높이기 위한 검토 방법들 중 하나로써 그 정의와 목적, 그리고 수행 방법에 있어 기존 방법들과 분명한 차이가 있다. 하지만 대부분의 개발자나 관리자들이 기존 방법인 동료검토, 워크스루, 그리고 테스트와 혼동을 하거나 명확히 구분을 하지 못하고 있다. 따라서 본 연구에서는 문헌자료를 바탕으로 인스펙션과 기존 검토 방법들을 정리하였다.

둘째, 코드 인스펙션을 위한 규칙을 수립하고 사례를 통해 성과를 증명하였다.

본 연구에서는 A 社의 정보시스템 개발 프로젝트를 대상으로 코드 인스펙션을 위한 규칙들을 수립하였다. 물론 특정 프로젝트를 대상으로 수립된 규칙이기 때문에 일반화시키거나 표준화 시키기에는 다소 무리가 있다. 하지만 본 연구에서 수립한 규칙들은 소프트웨어 개발 시 암묵적으로 지켜온 방법들을 기초로 수립된 것으로써 충분한 가치가 있다고 판단된다.

셋째, 코드 인스펙션을 통해 정량적 데이터 관리를 수행하였다.

코드 인스펙션 대상을 각 업무별, 기능별, 세부 항목에 따라 정량적으로 파악하고 결함의 규모와 빈도수를 분석하였다. 그리고 결함율 0%를 목표로 코드 인스펙션과 재작업 과정을 반복함으로써 코드의 완성도를 높이고 소프트웨어 개발의 효율과 품질을 높이기 위해 노력하였다.

이처럼 본 연구에서는 코드 인스펙션 규칙들을 수립하고, A 社의 정보시스템 구축 프로젝트에 적용하여 그 성과를 분석하였다. 이러한 노력은 향후 소프트웨어의 원활한 개발과 유지보수 그리고 프로젝트의 품질을 높이는데 긍정적인 영향을 줄 것으로 기대된다. 하지만 인스펙션은 기존에 수행하던 활동이 아니며 초기에 개발자들이 평균적으로 더 많은 작업시간을 필요로 하기 때문에 인스펙션을 공식적인 개발활동의 일부로 인정받을 수 있는 환경조성이 매우 중요하다. 이를 위해서는 개발자뿐만 아니라 경영진 그리고 고객의 합의와 참여를 이끌어 내기 위한 대책 마련이 필요하다. 또한 소프트웨어 인스펙션의 지속적인 적용과 연구를 바탕으로 체계적인 소프트웨어 품질향상의 기반을 구축해야 할 것이다.

참고문헌

[1] 최은만, "소프트웨어 공학", 정익사, 2006.

[2] CMU/SEI, "The Capability Maturity Model: Guides for Improving the Software Process", Addison Wesley, 1994.

[3] Davis, A., "Software Requirements: Analysis and Specification", Prentice-Hall, 1990, p. 20.

[4] Fagan, M. E., "Design and Code Inspections to Reduce Errors in Programming Development", IBM Systems, Vol. 15, No. 3, 1976.

[5] Fagan, M. E., "Advances in Software Inspections", IEEE Transactions in Software Engineering, Vol. 12, No. 7, 1986, pp. 744~751

[6] Fewster, M. and D. Graham, "Software Test Automation, Effective Use of Test Execution Tools", Addison-Wesley, 1999.

[7] IEEE, "IEEE Standard Glossary of Software Engineering Terms", IEEE Society Press, 1983.

[8] ISO 8402:1995, "Quality Management and Quality Assurance. Vocabulary", 1995.

[9] Kelly, J. C., J. S. Sherif and J. Hops, "An Analysis of Defect Densities Found During. Software Inspections", Journal of Systems Software, Vol 17, 1992.