Static 테스트를 통한 소프트웨어 결함기회 감소에 대한 연구

이동현○ 이주병 박은철
LIG 넥스원주식회사, Maritime연구센터
dhlee00@lignex1.com, erinjlee@lignex1.com, park.eunchul@lignex1.com

A Study on Reduction of Software Fault by Using Static Test

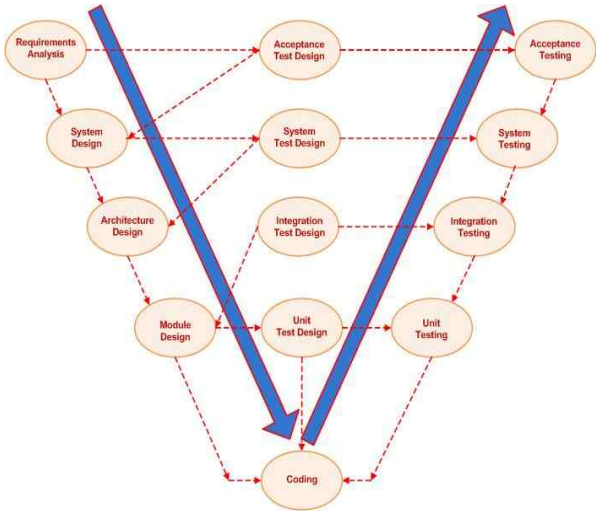
Donghyun Lee○, Joobyeong Lee, Eunchul Park
Maritime R&D Lab, LIG Nex1 Co.,Ltd

요 약

소프트웨어 개발의 복잡도가 증가하고 개발규모가 대형화 됨에 따라 소프트웨어 결함의 발생요인도 증가하고 있다. 소프트웨어의 결함을 줄이고 우수한 품질의 소프트웨어 개발을 위해 소프트웨어 테스트 기법인 static 테스트를 사용한다. 본 논문에서는 테스트 tool 을 이용하여 규모가 큰 소프트웨어 개발과정에 static 테스트를 적용하였으며, 적용결과를 통해 static 테스트가 소프트웨어 품질과 결함기회 감소에 미치는 영향을 분석한다.

1. 서 론

프로세서의 성능향상과 이에 따른 요구사항의 증가로 인해 소프트웨어 개발은 이전에 비해 복잡성이 높아졌으며 많은 인원이 투입되고 있다. 이는 소프트웨어의 결함 발생률을 높이는 요인으로, 소프트웨어의 품질을 높이고 결함을 제거하려는 시도가 지속적으로 진행되고 있다. 그 중 하나가 소프트웨어 테스트이며 높은 신뢰도를 요구하는 자동차, 항공, 국방분야의 소프트웨어 개발에 주로 사용된다.



[그림 1] 소프트웨어 개발의 V모델

소프트웨어 테스트란 소프트웨어의 결함을 발견하기 위해 사용하는 방법으로 코드를 검사하거나 프로그램 실행을 통해 결함을 검출하는 방법이다. 소프트웨어 테스트는 [그림 1]과 같이 V모델로 표현되는 일반적인 소프트웨어 개발 및 검증 단계(verification & validation)에서, 소프트웨어 단위시험에 주로 사용된다.[1][2]

프로젝트단위의 소프트웨어 개발 과정에서 소프트웨어 테스트는 40%의 내외의 비중을 차지할 정도로 커서, 효율적인 소프트웨어 테스트 수행이 소프트웨어의 신뢰도를 결정하는 중요한 요소가 된다. 최근에는 테스트에 들어가는 비용을 감소시키고 오류검출의 효율을 높이기 위해 자동화된 소프트웨어 테스팅 tool들을 사용하고 있다.[3] 소프트웨어 테스팅 tool을 프로젝트에 적용한 NASA에서는 tool을 사용하지 않은 이전 프로젝트에 비해서 결함율이 낮아지고 수정에 소요된 시간도 줄어들었다는 보고서를 발표하는 등 소프트웨어 테스트에서 tool 사용의 이점이 강조되고 있다.[4]

2. 개 요

소프트웨어 테스트는 크게 static 테스트와 dynamic 테스트 두가지로 나뉘는데, static 테스트는 실제로 수행되는 프로그램 없이 소스코드 혹은 오브젝트코드 만으로 테스트를 수행하는 것인 반면, dynamic 테스트는 실제 혹은 가상 프로세서 상에서 프로그램을 구동시키면서 테스트를 수행하는 것이다.[5] 본 논문에서는 테스트 방법 중 static 테스트를 적용했을 경우 결함감소에 미치는 영향만을 분석한다.

static 테스트를 지원하는 tool로는 Microsoft.Net에 사용되는 FxCop와 StyleCop, java에 사용되는 FindBugs, PMD, Jlint, C/C++에 사용되는 CodeSonar, HP Code Advisor, Mygcc, Splint, PolySpace Verifier 등이 있다.

본 논문에서는 static테스트를 소프트웨어 개발 과정에

적용하며, 앞서 기술한 tool중 GRAMMATECH사의 CodeSonar를 사용하여 소프트웨어 테스트의 효과를 분석한다.

소프트웨어 테스트의 대상은 10명의 소프트웨어 개발자가 투입되어 3년간 개발되는 소프트웨어 프로젝트이다.

2.1. 시험조건

CodeSonar에서 제공하는 static 테스트의 검사항목은 50여가지이다. 하지만 모든 검사항목을 테스트 하는 것은 많은 오검출로 인해 오검출 선별 시간이 더 많이 소요될 수 있으므로, 프로젝트의 영향도를 고려하여 검사항목들 중에서 결함이 발생하였을 경우 시스템에 치명적인 오류를 일으킬 수 있는 항목들로만 24종을 선정하였다. 선정된 검사항목들은 위험도에 따라 다시 A type, B type, C type으로 분류하였으며, A type은 고장으로 이어지는 고위험의 결함, B type은 경우에 따라 고장으로 이어지는 다소 위험한 결함, C type은 고장으로 반드시 이어지는 것은 아니지만 개발자의 확인이 필요한 항목으로 구분하였다. A type의 결함은 반드시 수정이 필요한 항목들이며, B type은 가능한 수정이 필요한 항목들이다. 선정된 24종의 결함검출 항목들은 3차에 걸친 소프트웨어 테스트에서 동일하게 적용하며 그 내용은 [표 1]과 같다.

[표 1] 결함 검출항목

결함 검출내용 (24종)	중요도
Buffer Overrun	*A*
Buffer Underrun	*B*
Cast Alters Value	*C*
Dangerous Function Cast	*C*
delete Object Created by new[]	*A*
Division By Zero	*B*
Double Free	*A*
Empty if Statement	*B*
Return Pointer To Local	*A*
Leak	*A*
memcpy Length Unreasonable	*A*
memcpy With Overlapping Regions	*A*
Missing Return Statement	*A*
Null Pointer Dereference	*A*
Null Test After Dereference	*C*
Redundant Condition	*B*
strncpy Does Not Null Terminate	*C*
Type Overrun	*A*
Type Underrun	*B*
Uninitialized Variable	*A*
Unreachable Code	*C*
Unused Value	*C*
Use After Free	*A*
Useless Assignment	*C*

3. 소프트웨어 테스트 계획

소프트웨어 테스트를 적용한 프로젝트의 개발계획은 [그림 2]와 같으며, 서론에서 소개한 V 모델을 바탕으로 작성 되었다. 소프트웨어 테스트는 static 테스트가 가능한 단계인 'SW 코딩 ' 단계부터 'CSCIⓐ 시험' 단계까지 수행하며, 이 기간은 총 14개월이 소요된다. 소프트웨어 테스트는 이 두 단계의 기간동안 총 3회에 걸쳐 실시하며, 1차 테스트는 'SW 코딩'단계가 시작된 후 10개월째, 2차 테스트는 12개월째, 3차 테스트는 14개월째에 각각 수행한다.

[그림 2] 프로젝트 개발계획

3.1 테스팅을 위한 형상항목 및 특이사항

소프트웨어 대상 CSCI는 [표2] 의 CSCI 항목과 같이 1차 테스트에서는 8종류(a,b,c,d,e,f,g,h), 2차 테스트에서는 8종류(a,b,c,d,e,f,g+h), 3차 테스트에서는 18종류(a,b,d,e,i,f,g+h,k,l,m,n,o.p,q,r,s,t,u)의 CSCI로 소프트웨어 테스트를 수행한다. 1차 테스트에서 수행된 CSCI 중에서 CSCI g 와 CSCI h는 형상항목의 통합으로 인해 2차, 3차 테스트에서는 CSCI g+h 한 항목으로 테스트를 진행한다. 또한 CSCI d 는 3차 테스트에서 설계 변경으로 인해 코드수가 오히려 감소하였지만 결함이 발생하지 않아 코드라인 증가 항목은 0으로 설정하였다.

본 논문의 결과분석에서 기술한 결함건수는 CodeSonar에서 검출한 모든 결함의 수이며 수정건수는 검출된 결함 중 오검출을 제외한 실제 수정이 필요한 결함의 수이다. 또한 결함건수는 이전에 오검출로 판단한 항목도 다음 시험에서 다시 결함으로 검출하게 되므로 해당하는 결함은 결함건수에서 제외한다.

테스트 결과 분석의 주요항목인 결함밀도는 1000라인당 결함발생률로써 (수정건수 / 코드라인수) * 1000으로 계산하며, 단위는 KLOC(kilo lines of code) 이다.

ⓐ CSCI: Computer Software Configuration Item, 소프트웨어형상항목

4. 결과분석

소프트웨어 테스트결과는 CSCI별 분석과 인원별 분석 두가지로 나누어 진행 한다. CSCI별 분석에서는 1,2,3차 테스트에서 CSCI별로 발생한 결함밀도의 변화를, 인원별 분석에서는 1,2,3차 테스트를 수행하면서 인원별로 발생한 결함밀도의 변화를 분석하며, 이 두가지 분석 결과를 통해 소프트웨어 테스트가 소프트웨어 품질에 미치는 영향을 추적한다.

4.1 CSCI별 결함밀도 분석

1,2,3차의 테스트를 수행한 결과는 [표2]와 같으며 CSCI별 결함밀도를 분석하여 소프트웨어 테스트가 CSCI에 미치는 영향을 분석한다.

[표 2] CodeSonar 테스트 결과

CSCI	1차 테스트				2차 테스트				3차 테스트			
	결함 건수	수정	결함 밀도	코드 라인수	결함 건수	수정	결함 밀도	코드 라인수	결함 건수	수정	결함 밀도	코드 라인수
a	1	1	0.06	15,463	1	0	0.00	26,320	13	1	0.45	29,071
b	29	10	1.03	9,708	26	3	0.13	23,610	30	0	0.93	32,400
c	516	78	3.52	22,184	487	121	1.74	69,561				
d	28	15	1.17	12,872	111	22	0.58	37,749	36	0	1.52	23,616
e	22	7	11.24	623	15	2	1.75	1,145	6	1	4.22	1,423
i					17	9	1.86	4,833	40	3	5.10	7,838
f	43	22	1.31	16,855	26	3	0.17	17,767	32	0	1.76	18,182
g	25	7	0.69	10,109	39	2	0.06	31,230	86	2	2.65	32,471
h	39	9	0.91	9,844								
k									8	4	2.67	2,995
l									6	1	3.98	1,506
m									7	0	2.07	3,378
n									125	71	57.23	2,184
o									22	7	0.95	23,049
p									109	39	4.02	27,093
q									3	0	0.14	20,908
r									2	0	0.09	21,296
s									23	14	0.82	27,983
t									32	21	1.10	29,106
u									95	57	15.72	6,043
	703	149	1.53	97,658	722	162	0.76	212,215	675	221	2.17	310,542

이 시험결과에서 테스트 진행 차수별 결함밀도는 1차 테스트에서 1.53KLOC, 2차 테스트에서 0.76KLOC, 3차 테스트에서 2.17KLOC이다. 진행 차수와 결함밀도간에 상관관계를 일치시키기 위해 결함밀도 분석을 [표 3]과 같이 각 테스트별로 새로 작성된 코드만을 비교할 수 있도록 증가한 코드라인에 대비한 결함밀도로 재정리한다.

[표 3]에 제시된 증가코드 대비 결함밀도 분석결과 테스트를 수행함에 따라 결함밀도는 1차, 2차, 3차에 걸쳐 1.53KLOC, 1.41KLOC, 1.21KLOC로 꾸준히 감소하였다.

[표 3] 증가코드 대비 결함밀도

CSCI	1차 테스트		2차 테스트		3차 테스트	
	증가분 결함밀도	증가코드 라인수	증가분 결함밀도	증가코드 라인수	증가분 결함밀도	증가코드 라인수
a	0.06	15,463	0	10857	0.36	2751
b	1.03	9,708	0.22	13902	0	8790
c	3.52	22,184	2.55	47377		
d	1.17	12,872	0.88	24877	0	0
e	11.24	623	3.83	522	3.6	278
i			1.86	4833	1	3005
f	1.31	16,855	3.29	912	0	415
g	0.69	10,109	0.18	11277	1.61	1241
h	0.91	9,844				
k					1.34	2995
l					0.66	1506
m					0	3378
n					32.51	2184
o					0.3	23049
p					1.44	27093
q					0	20908
r					0	21296
s					0.5	27983
t					0.72	29106
u					9.43	6043
	1.53	97,658	1.41	114,557	1.21	170,480

또한 3차에서 테스트를 수행한 CSCI들 중 2차에서도 테스트를 진행했던 CSCI들만으로 증가분 결함밀도를 계산해보면 0.42KLOC가 나온다. 이를 통해 소프트웨어 테스트를 계속 수행할 수록 코드의 결함밀도가 줄어드는 것을 확인하였다.

4.2 인원별 결함밀도 분석

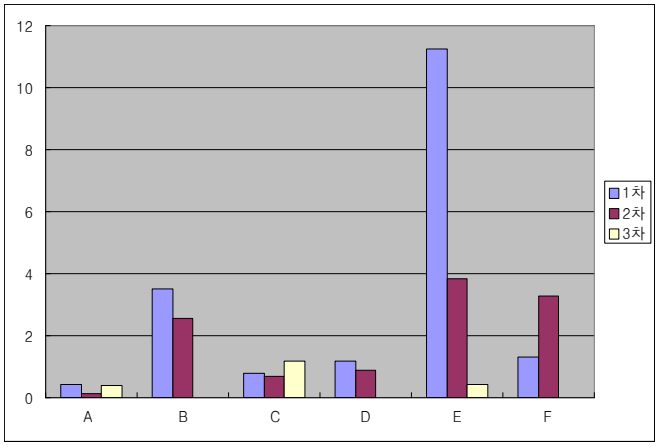
각 CSCI별 작성자와 진행차수는 [표4]와 같으며 개발자별 결함밀도를 분석하여 소프트웨어 테스트가 개발자에게 미치는 영향을 분석한다.

1차, 2차 테스트에서 모두 테스트를 진행한 CSCI를 작성한 개발자는 A,B,C,D,E,F 6명이다. 이들의 차수별 결함밀도를 비교해보면 [그림 3]과 같다.

A, C, F의 경우 결함밀도가 일부 증가했지만 A의 경우는 꾸준히 낮은 수준의 결함밀도를 유지하고 있으며, C와 F는 추가 개발한 코드라인이 많지 않아 정확한 비교대상이 될 수 없다는 점을 감안한다면 테스트 경험이 결함밀도를 낮추는데 일정부분 기여를 했음을 알 수 있다.

[표 4] CSCI별 작성자

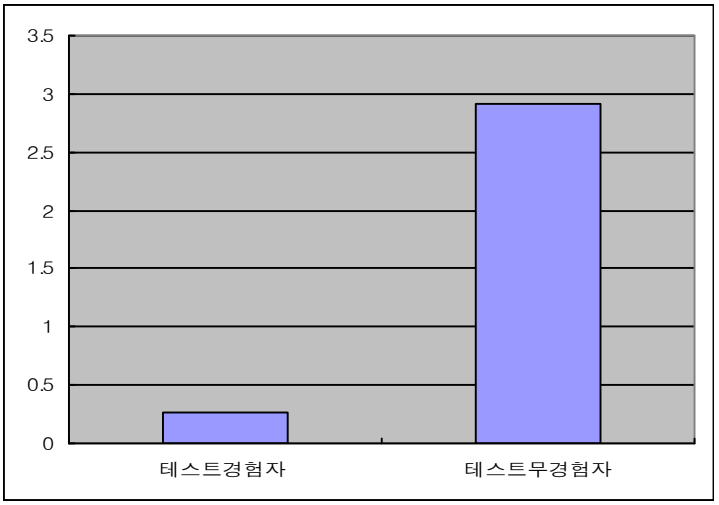
CSCI	작성자	진행차수
a	A	1,2,3
b	A	1,2,3
c	B	1,2
d	D	1,2,3
e	E	1,2,3
i	C	2,3
f	F	1,2,3
g	C	1,2,3
h	C	1,2,3
k	E	3
l	E	3
m	E	3
n	K	3
o	E	3
p	H	3
q	B	3
r	B	3
s	A	3
t	G	3
u	I	3



[그림 3] 개발자별 테스트 결과

3차에서 테스트를 진행한 CSCI들 중 1,2차 테스트에 참여한 개발자가 3차에서 처음 작성한 CSCI들과 새로 참여한 개발자들이 작성한 CSCI들의 결함밀도 비교를 통해 테스트 경험과 결함밀도 사이의 연관을 추적 할 수 있다. [그림 4]는 3차 테스트에 처음 참여한 개발자들이 작성한 CSCI n,p,t,u 와 1,2차 테스트 경험이 있는 개발자들이 3차 테스트에서 처음 작성한 CSCI k,i,m,o,q,r,s 사이의 결함밀도를 비교한 그림이다.

이 6명이 3차 테스트에서 처음 작성한 CSCI들의 결함밀도는 0.26KLOC이고, 처음 테스트를 진행한 G,H,I,K 4명이 작성한 CSCI들의 결함밀도는 2.92KLOC 였다. 즉, 기존에 테스트를 진행했던 개발자들이 작성한 CSCI의 결함밀도가 상대적으로 많이 낮았다. 이는 테스트를 진행했던 개발자들은 자신이 작성한 결함들을 수정하는 과정을 통해 자신이 만든 결함들을 인식하고 다른 CSCI 작성시 같은 결함들을 회피하고 있음을 나타낸다.



[그림 4] 테스트 경험자와 무경험자의 결함밀도

또한 [표 5]와 같이 1차, 2차 테스트를 진행하였던 개발자들이 작성한 CSCI에서는 A급 결함이 하나도 나오지 않았다는 점은, 반복적인 결함 수정과정을 통해 고위험의 결함을 인식하고 고위험의 결함이 발생하지 않도록 주의를 기울이는 것으로 볼 수 있다.

[표 5] CSCI별 결함등급

CSCI	작성자	결함등급			결함등급			결함등급		
		A급	B급	C급	A급	B급	C급	A급	B급	C급
a	A	1	0	0	0	0	0	0	0	1
b	A	2	5	3	1	0	2	0	0	0
c	B	13	23	42	9	52	60			
d	D	2	4	9	2	10	10	0	0	0
e	E	0	5	2	0	0	2	0	1	0
i	C				0	4	5	0	2	1
f	F	3	7	12	0	1	2	0	0	0
g	C	1	1	5	0	2	0	0	0	2
h	C	0	2	7						
k	E							1	1	2
l	E							0	0	1
m	E							0	0	0
n	K							21	17	33
o	E							1	2	4
p	H							7	19	13
q	B							0	0	0
r	B							0	0	0
s	A							2	5	7
t	G							5	10	6
u	I							11	9	37
		22	47	80	12	69	81	48	66	107

5. 결 론

소프트웨어 테스트를 반복해서 수행할수록 소프트웨어의 결함밀도가 떨어지는 것과 3차에서만 테스트를 진행했던 CSCI들의 결함밀도가 계속 테스트를 진행해왔던 CSCI들의 결함밀도보다 훨씬 높다는 것을 확인하였으며, 이는 반복되는 소프트웨어 테스트 수행이 소프트웨어 결함률을 감소시키는 것으로 분석 할 수 있다.

또한 반복해서 테스트를 수행한 개발자들이 작성한 CSCI들의 결함밀도가 낮다는 결과는 소프트웨어 테스트 수행이 개발자들의 소프트웨어 개발 능력 향상에 도움이 된다는 것으로 분석된다.

이 두가지 분석을 바탕으로, 소프트웨어 테스트를 수행함으로써 소프트웨어의 결함률이 낮은 우수한 품질의 소프트웨어 개발을 기대할 수 있다.

참고문헌

[1] 이범우, 소프트웨어 테스팅의 진화과정, 소프트웨어 테스팅 전문 저널 - STEN Journal Vol. II, January 2005, pp. 8-11

[2] 서광익, 임베디드 소프트웨어 테스트를 위한 Mulitple V-모델과 적용, 소프트웨어 테스팅 전문 저널 - STEN Journal Vol. II, January 2005, pp. 34-38

[3] Roger S.Pressman, Sfotware Engineering A Practioner's approach, Third Edition, 1992

[4] Technology Infusion of CodeSonar into the Space Network Ground Segment, NASA, Goddard Space Flight Center, September, 2008

[5] Ivo Gomes, An overview of the Static Code Analysis approach in Software Development, Faculdade de Engenharia da Universidade do Porto, 2009