소프트웨어 재사용에 따른 생산성 향상의 분석

강 현 미, 박 만 곤
부경대학교 자연과학대학 전자계산학과
장 화 식
제주관광전문대학 사무자동화과

요 약

새로운 소프트웨어를 개발할 때에 이미 개발되어진 소프트웨어를 재사용함으로써 얻을 수 있는 장점은 많다. 그중에서 이미 검증된 소프트웨어를 사용함으로써, 오류가 감소됨에 따라 고품질의 소프트웨어를 생산할 수 있고, 소프트웨어의 개발비용을 절감시킬 수 있고, 개발인원을 감소시킬 수 있으며, 소프트웨어 개발기간을 단축시켜 생산성을 향상시킬 수 있다. 본 논문에서는 소프트웨어를 재사용함으로써 소프트웨어 개발시에 사용되는 비용과 생산성을 상대 비교해서, 소프트웨어 개발에 소요되는 비용과 생산성의 관계를 나타내는 새로운 모형과 예를 제시하고 있다. 또한 소프트웨어 재사용시에 개발기간과 총인월에 대한 생산성을 COCOMO 모형을 사용해 규명하고 있다.

1. 서 론

소프트웨어 개발비용이 너무 높다고 인식되어 가는 만큼 소프트웨어 기능의 부품화 연구가 활발히 진행되어 오고 있다. 소프트웨어 재사용은 소프트웨어를 개발하거나 유지보수를 할 때 기존에 사용된 부품(component)을 일부 수정하거나 또는 완전히 다시 사용할 수 있는 개념을 의미한다. 이렇게 소프트웨어를 재사용함으로써 얻을 수 있는 이점으로는 오류가 감소함으로써 고품질(highest quality)의 소프트웨어를 생산할 수 있고, 소프트웨어 개발기간을 단축시켜 생산성(productivity)을 향상시킬 수 있으며, 소프트웨어 개발비용을 절감시킬 수 있고 개발 인원을 감소시킬 수 있다[1][2].

소프트웨어 재사용은 방법론적으로는 패턴재사용(pattern reuse)방법과 빌딩블럭 재사용(reuse of building blocks)방법으로 구분될 수 있다[2]. 패턴재사용 방법은 목적 시스템을 생성하기 위해 수행되는 부분을 포함한다. 이것은 제너레이터(generator)를 이용한 패턴(pattern)의 재사용이라고 볼 수 있다. 이 방안은 목표시 되는 소프트웨어에 대한 일반적인 모형(model)을 만들어 놓고 거기에 필요한 매개변수를 적용하여 필요에 따라 소프트웨어를 생성해내는 방법이다. 이러한 패턴재사용 모형은 응용생성기(application generator)안에서는 코드형태로 존재하며 변환시스템(transformation system)에서는 변환규칙 형태로 존재한다. 이 분야에는 언어기반생성기(language-based generators), 응용생성기(application generator), 변환시스템(transformation system)을 통한 연구가 진행되고 있다.

반면에 빌딩블럭재사용 방법은 새로운 목적 시스템을 생성하기 위해 결합되는, 포괄적이고 재사용 가능한 소프트웨어 부품을 포함하는 소프트웨어 라이브러리(library)를 만드는 방법을 말한다. 이것은 부가적인 재작업없이 소프트웨어와 기존의 소프트웨어 라이브러리를 통합하기 위해 사용될 수 있다. 또한 유용한 부품들을 수집하고 알맞게 분류하여 라이브러리에 저장한 뒤 필요에

따라 식별하여 사용하는 것이며 부품들 사이에 인터페이스가 분명히 정의되어 있을수록 재사용이 용이하다. 이 방안의 실용화를 위해서 응용부품 라이브러리(application component library), 구조와 구성원리(organization and composition principles) 등에 관한 연구가 진행중이다. 전형적인 시스템으로는 서브루틴 라이브러리(subroutine library), UNIX의 파이프(pipe)기능 및 객체지향언어 등이 있다.

소프트웨어 부품을 재사용하기 위해서는 기존에 개발된 재사용가능한 부품을 효율적으로 분류하여 라이브러리에 구조적으로 저장하는 방법, 사용자의 요구사항을 만족하는 부품을 신속하고 정확하게 검색하는 방법, 사용자의 요구사항에 따라 검색한 부품을 이해하기 위한 표현 방법, 새로운 소프트웨어 시스템에 결합시키는 방법 및 이들 기법을 평가하기 위한 기술 등이 개발되어져야 한다.

소프트웨어 생산성이 중요한 이유는 소프트웨어 위기(software crisis) 현상을 극복하기 위함인데, 소프트웨어 개발 단계에서 생산성을 높일 수 있는 가장 좋은 방법은 기존에 개발된 소프트웨어를 재사용하는 것이다[3]. 또한 소프트웨어를 재사용함으로써 소프트웨어 개발에 드는 비용이 현저히 감소하게 된다.

본 논문에서는 소프트웨어 개발에 사용되는 비용과 생산성을 상대 비교해서 소프트웨어 개발에 드는 비용과 생산성의 관계를 새로운 모형과 예제로써 재조명하고자 한다. 제2장에서는 소프트웨어 생산성 메트릭스에 대해 기술하였고, 제3장에서는 기존에 연구되었던 소프트웨어 생산성에 대한 상대비용과 생산성의 모형을 검토하고, 제4장에서는 제3장에서 제시되었던 모형을 기반으로 NASA[4]에서 측정된 자료를 중심으로 상대비용과 상대생산성에 대한 새로운 모형을 제시하고 예를 통해 생산성을 분석하였으며, 제5장에서는 COCOMO 모형을 이용해 총인월과 개발기간을 고려한 생산성 분석을 하였으며, 제6장에서는 결론으로 구성되었다.

2. 소프트웨어 생산성 메트릭스

본 절에서는 생산성 측정(productivity measurement)을 위한 대표적인 메트릭스(metrics) 중 라인수 메트릭스, Halstead의 메트릭스, 소프트웨어 복잡도를 측정하는 McCabe의 메트릭스 그리고 프로젝트의 규모를 평가해 보는 Albrecht의 기능점수(function point) 메트릭스에 대해 알아보기로 한다.

2.1 라인수 메트릭스

동일한 기능과 성능을 갖는 소프트웨어를 각각의 프로그래밍 언어를 이용하여 구현했을 때 소요되는 인월 즉 인건비는 저급언어보다는 고급언어일수록 낮다는 것은 확실하다. 그렇지만 구현결과를 라인수로 따져보고 이를 소요 인월로 나누어 보면 오히려 고급언어 사용자의 생산성은 저하되고 라인당 비용도 높은 것으로 나타난다. 이러한 결과로 인월당 라인수를 생산성 척도로 이용하는 것은 문제가 된다고 지적되고 있다.

2.2 소프트웨어 과학

프로그래밍 언어의 이질성을 극복하는 회로의 시도는 프로그램의 기능과 데이터를 분류시켜 분석해 볼 M. Halstead의 연구결과이다[5]. 소프트웨어의 논리량(logical volume)을 과학적으로 측정하고 있다는 점에서 소프트웨어 과학(software science)이라고 한다.

소프트웨어 과학은 프로그램을 오퍼레이터(operator)와 오퍼랜드(operand)라고 하는 독립된 원소(automic particle)들의 집합으로 정의한다. 오퍼레이터란 더하기, 빼기, 이동(move), 비교(compare), 읽기(read), 쓰기(write), 마치기(terminate) 등의 동사(verb)격인 명령어 자체를 뜻하며, 오퍼랜드란 데이터로서 숫자, 혹은 주기억장치 내의 주소(address) 및 변수(variable) 등 명사(noun)격인 명령의 대상을 뜻한다. 즉,

프로그래밍 언어와 무관하게 동사격인 오퍼레이터와 명사격인 오퍼랜드의 숫자를 셈하여 프로그램에 내재된 논리의 규모를 측정하는 것이다. 소프트웨어 과학의 기본 메트릭스는 다음과 같다.

n1 : 오퍼레이터의 수(동사의 수)
n2 : 오퍼랜드의 수(명사의 수)
N1 : 오퍼레이터가 사용된 총 수
N2 : 오퍼랜드가 사용된 총 수

위의 네가지 메트릭스를 이용한 주요한 공식은 아래와 같다.

프로그램의 단어(n) : n1 + n2
프로그램의 길이(N) : N1 + N2
프로그램의 부피(V)
: $(N1 + N2)\log_2(n1 + n2)$

프로그램의 길이란 단어수와 정비례하며, 부피란 프로그램 단어의 수에 따라 급속히 증가한다. 소프트웨어 과학은 명사에만 치중하고 형용사와 부사를 무시하고 있는 단점이 있다.

2.3 소프트웨어 복잡도

소프트웨어의 복잡도는 문제의 복잡도, 프로그램 구조의 복잡도로 세분된다.

(1) 문제의 복잡도

주어진 문제에 대한 인간의 인지도(perception)는 전문가와 비전문가의 사이에는 차이가 있다. 주어진 문제의 복잡도를 해소시키려면 경험이 중요하다. 전체의 흐름을 읽고 경험에 비추어 패턴(pattern) 인식 감각이 있어야 한다. 따라서 주어진 문제에 대한 이해방법은 경험과 지식에 의존하는 것이다.

(2) 프로그램 구조의 복잡도

프로그램 구조의 복잡도를 측정하는 대표적인 메트릭스는 T. McCabe의 순환복잡도(cyclomatic complexity)이다[6][7][8]. 이 메트릭스는 프로그램 수행의 경로(execution flow)를 프로그램 라인을 표시하는 노드(node)와 수행경로를 표시하는 화살표(arrow)의 그래프(graph)로 나타낸 후 다음과 같은 공식을 적용시키고 있다.

복잡도(G) = e - n + 2

여기에서, e는 간선(edge)들의 수(화살표의 수)이고 n은 노드(node)들의 수이다. 또한 T. McCabe는 복잡도에 따르는 소프트웨어 품질을 다음의 <표 1>과 같이 평가하고 있다.

<표 1> 소프트웨어 복잡도의 분류 및 평가

복잡도(G)	평 가 결 과
G≤5	매우 간단한 프로그램
6≤G≤10	매우 구조적이며 안정된 프로그램
G≥20	문제 자체가 매우 복잡하거나 구조가 필요 이상으로 복잡한 프로그램
G≥50	매우 비구조적이며 불안정한 프로그램

복잡도 메트릭스는 결과를 평가해 줄 뿐 개발과정에서의 유용한 도구는 되지 못하고 있으며, 구조의 복잡도에 치우쳐서 데이터 복잡도는 무시하고 있는 단점이 있다.

2.4 소프트웨어 기능 측정

A. J. Albrecht[9]의 기능점수(function point) 메트릭스는 소프트웨어 생산성을 평가하는데 가장 유용한 모형으로 평가받고 있다. 데이터의 복잡도와 프로그램의 구조적 연관성을 복합적으로 고려함은 물론 프로그래밍 언어와도 독립성을 갖고 있기 때문이다. 기능점수 메트릭스는 입력, 출력, 문의(inquiries), 파일, 그리고 외부와의 인터페이스(interface)들을 집합시켜 이를 소프트웨어가 제공해야 하는 하나의 기능점수로 예측하는 것이다. 기능점수에 대한 공

식은 아래와 같다.

기능점수 = [입력수×4] + [출력수×5]
+ [문의수×4] + [파일수×10]
+ [인터페이스수×7]

3. 상대비용과 상대생산성에 대한 기존의 연구

Bruce Barnes, Thomas Durek, John Gaffney, Arthur Pyster[10]는 상대비용 및 상대 생산성을 다음과 같이 정의하고 있다.

3.1 상대비용(Relative Cost)

Bruce Barnes, Thomas Durek, John Gaffney, Arthur Pyster는 상대비용을 정의하기 위하여 다음과 같은 가정을 두고 있다.

가정) 1. b는 재사용가능한 부분에 소요되는 상대비용이다.

2. (1-R)는 프로젝트내에서 새롭게 개발되는 코드의 비율이다.

3. 새로운 코드를 개발하는데 소요되는 상대비용은 1이다.

이와같은 가정을 두고 상대비용을 산출하면 다음과 같다.

$$RC = (1-R)1 + Rb = (b-1)R + 1. \quad \cdots\cdots(1)$$

3.2 상대생산성(Relative Productivity)

식(1)을 이용하여 상대생산성을 산출하면 다음과 같다.

$$RP = \frac{1}{RC} = \frac{1}{(b-1)R + 1}, \quad \cdots\cdots(2)$$

여기에서 재사용 가능한 부분에 소요되는 상대비용이 0이라 가정하면 상대생산성은 더욱 간단한 형태로 나타난다.

$$RP = \frac{1}{1-R}. \quad \cdots\cdots(3)$$

4. 소프트웨어 재사용에 대한 생산성 향상 분석

본 논문에서는 상대비용 및 상대생산성의 분석을 위해 <표 2>의 NASA에서 측정된 자료[4]를 활용하였다.

<표 2> 프로젝트 시기 구분, 프로젝트 크기 구분 및 개별 모듈의 재사용 구분 자료

프로젝트시기	프로젝트크기	프로젝트번호	재사용 모듈의 비율				모듈의 수
			완전히 새롭게 개발	많은 수정	조금 수정	완전한 재사용	
초기	대형	P1	90.07		6.51	3.42	292
		P2	68.47	8.24	6.25	17.05	352
		P3	63.76	4.62	24.47	7.35	585
		P4	18.12	6.12	7.29	68.47	425
		P5	67.92	1.10	5.01	25.98	639
		P6	66.51	12.56	12.87	8.06	645
		P7	67.06	6.80	12.90	13.25	853
		P8	98.56	0.48		0.96	418
	소형	P9	60.87		13.04	26.09	23
		P10	92.86		2.38	4.76	42
		P11	54.46	6.93	14.85	23.76	101
		P12	91.15	2.65		6.19	113
		P13	79.73	5.41	12.16	2.70	74
후기	대형	P14	48.51	8.01	16.48	27.00	437
		P15	55.41	5.18	15.09	24.32	444
		P16	75.59	0.39		24.02	254
		P17	76.57	3.53	15.37	4.53	397
		P18	91.37	2.40	6.24		417
	소형	P19	59.39	0.61	16.36	23.64	165
		P20	66.20			33.80	71
		P21	27.27			72.73	22
		P22	87.40			12.60	246
		P23	100.0				28
		P24	77.89		2.11	20.00	95
		P25	38.00	2.00	32.00	28.00	50
전체			68.93	4.53	12.30	20.81	

<표 2>의 자료 구성을 보면 다음과 같다.

프로젝트 정보

1. 프로젝트의 크기는 두가지 형태로 구분하고 있다.
   대형(≥ 20,000 LOC)
   소형(< 20,000 LOC)

2. 프로젝트 개발 시기의 구분은 다음과 같다.
   초기(1979년 9월 이전)
   후기(1979년 9월 이후)

모듈정보

1. 모듈의 형태는 다음과 같다.
   대형(≥ 140 LOC)
   소형(< 140 LOC)

2. 재사용된 모듈의 구분은 다음과 같다.

   수정없이 완전한 재사용
   조금 수정(< 25% 수정)
   많이 수정(≥ 25% 수정)
   완전히 새롭게 개발

<표 2>의 자료를 중심으로 식(1)을 이용하여 상대비용을 산출하면 다음과 같다.

$$RC = R_0 b_0 + R_1 b_1 + R_2 b_2 + R_3 b_3. \quad \cdots\cdots(4)$$

(식 4)를 이용하여 상대생산성을 산출하면 아래와 같다.

$$RP = \frac{1}{R_0 b_0 + R_1 b_1 + R_2 b_2 + R_3 b_3}. \quad \cdots\cdots(5)$$

여기에서, 전체 코드라인 수와 전체 개발비용에 대해서

R_0 는 완전히 새롭게 개발하는 코드의 비율이다.

R_1 는 많이 수정되는 코드의 비율이다.

R_2 는 조금 수정되는 코드의 비율이다.

R_3 는 완전히 재사용하는 코드의 비율이다.

b_0 는 소프트웨어를 완전히 새롭게 개발하는데 소요되는 상대비용이다

b_1 는 소프트웨어를 많이 수정하는데 소요되는 상대비용이다.

b_2 는 소프트웨어를 조금 수정하는데 소요되는 상대비용이다.

b_3 는 소프트웨어를 완전히 재사용하는데 소요되는 상대비용이다.

이상의 정의를 기반으로 <표 2>를 이용해 상대비용과 상대생산성을 산출하기 위해 다음과 같은 가정을 둔다.

가정) 1. 소프트웨어를 새롭게 개발하는데 소요되는 상대비용은 1이다.

2. 소프트웨어를 완전히 재사용하는 부분에 소요되는 상대비용은 0이다.

3. $b_i \geq b_{i+1}$, i = 0, 1, 2, 3.

식(4)를 이용하여 <표 1>의 자료를 중심으로 상대비용을 재구성하면 다음 <표 3>와 같다. <표 3>은 소프트웨어 재사용의 초기단계와 후기단계에 대한 대형과 소형의 프로젝트를 평균한 내용으로 구성되어 있다.

위에서 사용한 가정들을 이용하여 식(4), (5)를 변형하면 아래와 같다.

$$RC = R_0 + R_1 b_1 + R_2 b_2 . \quad \cdots\cdots(6)$$

$$RP = \frac{1}{R_0 + R_1 b_1 + R_2 b_2} \cdot \quad \cdots\cdots(7)$$

<표 3>과 식 (6), (7)을 이용하여 상대비용 b_1, b_2를 적용하였을 때 계산된 결과는 <표 4>과 같다.

<표 3> 상대비용 산출결과

초기	대형	$RC = 67.56b_0 + 5.70b_1 + 9.41b_2 + 18.07b_3$
	소형	$RC = 75.81b_0 + 3.00b_1 + 8.49b_2 + 12.70b_3$
	평균	$RC = 70.73b_0 + 4.22b_1 + 9.06b_2 + 16.00b_3$
후기	대형	$RC = 69.49b_0 + 3.90b_1 + 10.64b_2 + 15.97b_3$
	소형	$RC = 65.16b_0 + 0.37b_1 + 7.21b_2 + 27.25b_3$
	평균	$RC = 66.97b_0 + 3.16b_1 + 14.81b_2 + 27.06b_3$
전체	평균	$RC = 68.93b_0 + 3.08b_1 + 8.86b_2 + 19.15b_3$

4.1 초기의 생산성과 후기의 생산성 분석

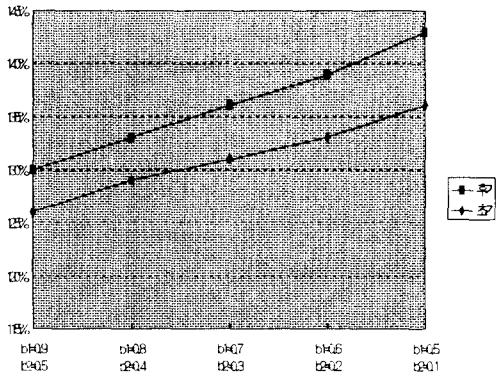
소프트웨어 재사용의 생산성을 초기단계와 후기단계로 비교해 보았을 때 그 결과는 아래 <표 5>와 같다.

<표 5> 초기의 생산성과 후기의 생산성

b_1	b_2	초기의 생산성(%)	후기의 생산성(%)
0.9	0.5	126	130
0.8	0.4	129	133
0.7	0.3	131	136
0.6	0.2	133	139
0.5	0.1	136	143

<표 4> b_1과 b_2를 적용한 상대비용 및 상대생산성의 예

b_1, b_2	프로젝트날짜	프로젝트크기	새롭게 개발	많은 수정	조금 수정	상대비용 (%)	상대생산성 (%)
b_1=0.9 b_2=0.5	초기	대형	67.56	5.13	4.71	77.40	129
		소형	75.81	2.70	4.25	82.76	121
		평균	70.73	3.80	4.53	79.06	126
	후기	대형	69.49	3.51	5.32	78.32	128
		소형	65.16	0.33	3.61	69.10	145
		평균	66.97	2.84	7.41	77.22	130
	전체		68.93	2.77	4.43	76.13	131
b_1=0.8 b_2=0.4	초기	대형	67.56	4.56	3.76	75.88	132
		소형	75.81	2.40	3.40	81.61	123
		평균	70.73	3.38	3.62	77.73	129
	후기	대형	69.49	3.12	4.26	76.87	130
		소형	65.16	0.30	2.88	68.34	146
		평균	66.97	2.53	5.92	75.42	133
	전체		68.93	2.46	3.54	74.94	133
b_1=0.7 b_2=0.3	초기	대형	67.56	3.99	2.82	74.37	134
		소형	75.81	2.10	2.55	80.46	124
		평균	70.73	2.95	2.72	76.40	131
	후기	대형	69.49	2.73	3.19	75.41	133
		소형	65.16	0.26	2.16	67.58	148
		평균	66.97	2.21	4.44	73.63	136
	전체		68.93	2.16	2.66	73.74	136
b_1=0.6 b_2=0.2	초기	대형	67.56	3.42	1.88	72.86	137
		소형	75.81	1.80	1.70	79.31	126
		평균	70.73	2.53	1.81	75.07	133
	후기	대형	69.49	2.34	2.13	73.96	135
		소형	65.16	0.22	1.44	66.82	150
		평균	66.97	1.90	2.96	71.83	139
	전체		68.93	1.85	1.77	72.55	138
b_1=0.5 b_2=0.1	초기	대형	67.56	2.85	0.94	71.35	140
		소형	75.81	1.50	0.85	78.16	128
		평균	70.73	2.11	0.91	73.75	136
	후기	대형	69.49	1.95	1.06	72.50	138
		소형	65.16	0.19	0.72	66.07	151
		평균	66.97	1.58	1.48	70.03	143
	전체		68.93	1.54	0.89	71.36	140



<그림 1> 초기의 생산성과 후기의 생산성 비교

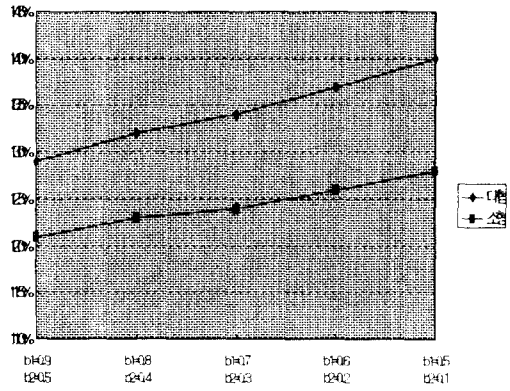
<표 5>와 <그림 1>의 결과를 보면 초기단계(1979년 9월 이전)의 생산성보다 후기단계(1979년 이후)의 생산성이 높음을 알 수 있는데, 이는 소트웨어 위기현상을 극복하기 위한 하나의 방편으로서 소프트웨어 재사용이 많이 이루어졌음을 나타내고 있다.

4.2 초기의 대형과 소형 생산성

초기단계에서 프로젝트의 크기가 큰 (≥20,000) 소프트웨어와 프로젝트의 크기가 작은(<20,000) 소프트웨어의 생산성을 비교해 보면 아래 <표 6>와 같다.

<표 6> 초기의 대형의 생산성 및 소형의 생산성

b_1	b_2	초기 대형의 생산성(%)	초기 소형의 생산성(%)
0.9	0.5	129	121
0.8	0.4	132	123
0.7	0.3	134	124
0.6	0.2	137	126
0.5	0.1	140	128



<그림 2> 초기의 대형의 생산성 및 소형의 생산성의 비교

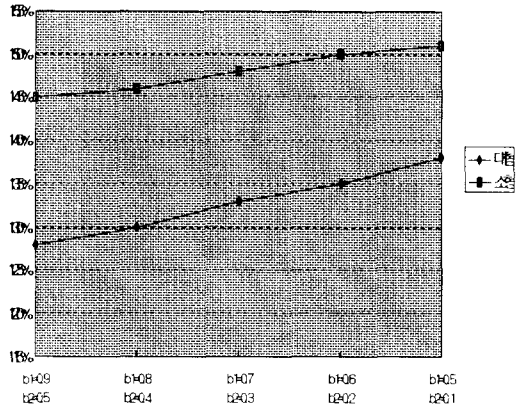
<표 6>과 <그림 2>의 결과를 보면 초기단계(1979년 9월 이전)의 대형 생산성이 소형 생산성보다 높음을 확인할 수 있다. 즉 초기단계의 소프트웨어 재사용은 크기가 큰 프로젝트가 크기가 작은 프로젝트보다 소프트웨어 재사용이 많이 이루어졌음을 알 수 있다.

4.3 후기의 대형과 소형의 생산성

후기단계에서 프로젝트의 크기가 큰 (≥20,000) 소프트웨어와 프로젝트의 크기가 작은(<20,000) 소프트웨어의 생산성을 비교해 보면 아래 <표 7>과 같다.

<표 7> 후기의 대형의 생산성 및 소형의 생산성

b_1	b_2	후기 대형의 생산성(%)	후기 소형의 생산성(%)
0.9	0.5	128	145
0.8	0.4	130	146
0.7	0.3	133	148
0.6	0.2	135	150
0.5	0.1	138	151



<그림 3> 후기의 대형과 소형의 생산성 비교

<표 7>와 <그림 3>의 후기단계 생산성을 보면 크기가 큰 소프트웨어의 생산성이 크기가 작은 소프트웨어의 생산성보다 낮음을 알 수 있다.

이상의 결과를 정리하면 다음과 같다. 소프트웨어 재사용은 1979년 9월 이후가 좀 더 많이 활성화되었으며, 최근에는 큰 프로젝트보다는 작은 프로젝트가 재사용이 많이 되었다고 할 수 있다. 또한 어떤 프로젝트 내에서 완전히 새롭게 개발되는 부분이 작을수록 그리고 소프트웨어를 수정하는 비용이 적게 들수록 소프트웨어 개발에 소요되는 비용은 감소된다. 또한 전체 프로젝트 내에서 수정하는 부분이 많은 비중을 차지하고 있지 않으므로, 수정하는 부분이 생산성에 미치는 영향은 그다지 크지 않음을 알 수 있었다. 즉, 다시 말하면 소프트웨어를 개발할 때에 완전히 재사용하는 부분이 많을수록 소프트웨어 개발에 소요되는 비용은 감소되고 생산성도 향상시킴을 알 수 있다.

5. 총인월과 개발기간을 고려한 생산성 분석

본 절에서는 소프트웨어를 재사용함으로써 인월(Man-Month)과 개발기간이 생산성 향상에 미치는 영향을 수학적인 방법을 이용해 논의하고자 한다.

총인월 및 개발기간을 추정하기 위한 수학적인 방법으로는 Boehm의 COCOMO 모형, Putnam의 생명주기 예측모형 그리고 Albrecht의 기능점수모형으로 집약된다[7]. 본 논문에서는 Boehm의 COCOMO 모형을 중심으로 개발기간 및 인월을 산출함으로써 소프트웨어 재사용과 생산성 관계를 조명하고자 한다. COCOMO 모형의 기본 공식은 다음 <표 8>과 같다.

<표 8> COCOMO 모형의 기본 공식

유형	기 본 공 식	
	총인월(MM)	개발기간(월)
유기형	$2.4\times[KDSI]^{1.05}$	$2.5\times[MM]^{0.38}$
준 분리형	$3.0\times[KDSI]^{1.12}$	$2.5\times[MM]^{0.35}$
내장형	$3.6\times[KDSI]^{1.20}$	$2.5\times[MM]^{0.32}$

여기에서 개발기간 및 총인월을 산출하기 위해 다음 <표 9>와 같은 기본 예를 두기로 한다.

<표 9> 재사용에 따른 신규 코드수 및 재사용 코드수

분 류	신규 제작코드	재사용 코드라인수	계
재사용 없슴	10,000	0	10,000
25% 재사용	7,500	2,500	10,000
50% 재사용	5,000	5,000	10,000
75% 재사용	2,500	7,500	10,000

<표 9>를 COCOMO 모형에 적용시키면 총인월과 개발기간은 다음의 <표 10>과 같다.

<표 10> 소프트웨어 유형별 총인월 및 개발기간

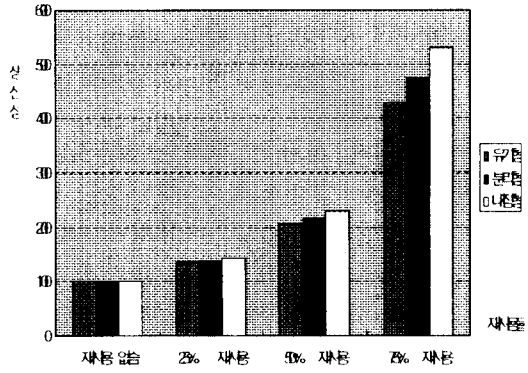
소프트웨어 유형 \ 재사용율		재사용 없슴	25% 재사용	50% 재사용	75% 재사용
유기형	총인월 (MM)	26.93	19.91	13.01	6.28
	개발기간 (월)	8.74	7.79	6.63	5.03
준분리형	총인월 (MM)	39.55	28.65	18.20	8.37
	개발기간 (월)	9.06	8.09	6.90	5.26
내장형	총인월 (MM)	57.06	40.40	24.84	10.81
	개발기간 (월)	9.12	8.17	6.99	5.36

<표 10>의 결과를 이용하여 소프트웨어 유형별 생산성을 분석하면 다음의 <표 11>, <그림 4>, <그림 5>와 같다.

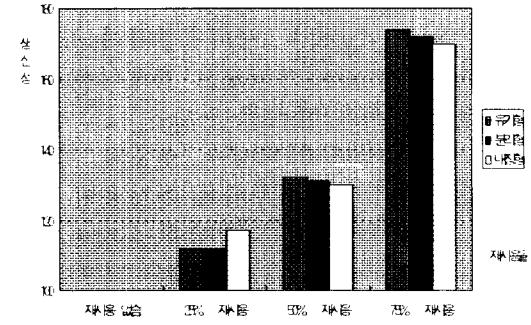
<표 11> 소프트웨어 유형별 생산성 분석(%)

소프트웨어 유형 \ 재사용율		재사용 없슴	25% 재사용	50% 재사용	75% 재사용
유기형	총인월	100	135	207	429
	개발기간	100	112	132	174
준분리형	총인월	100	138	217	473
	개발기간	100	112	131	172
내장형	총인월	100	141	230	528
	개발기간	100	117	130	170

<표 11>, <그림 4>, <그림 5>의 결과를 통하여 총인월의 경우 재사용율이 높음에 따라 생산성이 대폭 향상되었음을 확인할 수 있고, 또한 개발기간도 생산성이 많이 향상되었음을 알 수 있다. 따라서 소프트웨어 개발시에 기존에 개발된 소프트웨어를 재사용함으로써 개발기간 그리고 총인월이 생산성에 미치는 영향은 크다는 것을 확인할 수 있다.



<그림 4> 총인월과 재사용율에 따른 생산성



<그림 5> 개발기간과 재사용율에 따른 생산성

6. 결론

소프트웨어를 개발시에 소프트웨어 개발팀의 생산성을 향상시키고 소프트웨어 개발 비용을 감소시키기 위한 방법중의 하나가 한 번 사용한 소프트웨어 컴포넌트를 재사용하는 것이다.

소프트웨어 재사용을 통해 소프트웨어 생산성을 향상시키는 방법이 유망함에도 불구하고 크게 현실화 되어 있지는 못하다. 그 이유로는 재사용되는 코드가 자신이 설계가 한 것이 아니므로 수정하는 것이 쉽지 않다. 또한 재사용의 적용이 현실화 되기까지는 반드시 대두되는 것이 표준화 문제인데 부품의 분류, 재사용도 평가, 인덱스 탐색, 부품연결 등이 아직까지는 미흡한 상태이다.

이러한 장애요인을 해소해기 위해서는 객체지향적인 방법론의 활용, 재사용 소프트웨어 라이브러리의 구축, CASE를 활용해야 할 것이다.

본 논문에서 사용된 자료는 NASA에서 측정된 자료를 중심으로 전개되었는데 상대비용을 기반으로 상대 생산성을 비교해 보았다. 소프트웨어 재사용의 초기와 후기의 생산성을 비교해 보면, 후기에 들어 생산성이 높음을 알 수 있었고, 규모가 큰 소프트웨어와 규모가 작은 소프트웨어를 비교해 보았을 때는 초기에는 큰 소프트웨어가 생산성이 높았는 반면, 후기에는 작은 소프트웨어가 생산성이 훨씬 높음을 알 수 있었다. 또한 새롭게 개발되는 비율이 적고 상대비용이 낮을수록 상대생산성이 높음을 확인할 수 있었다. 그러므로 상대비용과 상대생산성은 서로 반비례 관계를 갖고 있다. 이는 소프트웨어 개발시에 새롭게 개발되는 비율이 적고 소프트웨어 재사용이 많이 이루어지면 생산성이 향상됨을 의미한다.

그리고 Boehm의 COCOMO 모형을 이용하여 소프트웨어 개발시 기존에 개발된 소프트웨어를 재사용함으로써 개발기간과 총인월이 생산성에 많은 영향을 미친다는 것을 확인했다. 또한 역으로 소프트웨어를 재사용함으로써 개발기간과 개발에 투입되는 인원이 줄어듬을 알 수 있다.

참고문헌

[1] Arnold S.P. and S. I. Stepoway, "The Reuse System : Cataloging and Retrieval of Reusable Software", Proceeding of COMPCONS'87, 1987, pp. 376-379.

[2] Biggerstaff, T. and C. Richter,"Reusability Framework, Assessment, and Direction", IEEE Software, Mar. 1987, pp. 41-49.

[3] B. W. Boehm, M. Penedo, A. Pyster, E. D. Stuckle, R. D. Williams, "An Environment for Improving Software Productivity", Computer, June 1984.

[4] Rechard W. Selby, "Empirically Analyzing Software Reuse in a Production Environment", Tutorial : Software Reuse : Emerging Technology, 1988, pp. 176-189.

[5] M. H. Halstead, Elements of Software Science, Elsevier North-Holland, 1977.

[6] T. J. McCabe, "A Complexity Measure", IEEE Transaction on Software Engineering, Dec. 1976.

[7] Pressman, R. S. Software Engineering, 3rd Edition, McGraw-Hill, 1992.

[8] Shooman, M. L., Software Engineering, McGraw-Hill, 1983.

[9] A. J. Albrecht, "Measuring Application Development Productivity", Proceeding of the Joint IBM/SHARE/GUIDE Application Development Symposium, Oct. 1979.

[10] Bruce Barnes, Thomas Durek, John Gaffney, Arthur Pyster, "A Framework and Economic Foundation for Software Reuse", Proceeding of the workshop on Software Reusability and Maintainability, Oct. 1987.