CBD 프로젝트에서의 MDD/MDA 기술 적용을 통한 개발생산성 향상 사례

최정일
LG CNS, SW아키텍처 센터

A Case Study of SW Development Productivity Improvement by MDD/MDA Technology Application in CBD Project

Choi, Jeongil
SW Architecture Center, LG CNS
E-mail : jichoi@lgcns.com

요 약

1948년 폰노이만이 최초의 메모리에 저장된 프로그램을 이용하는 현대식 컴퓨터인 IBM SSEC를 발명한 이후, 거의 50여 년 동안 소프트웨어를 만드는 기술은 끊임없이 변화해 왔다. 최근 대두되고 있는 MDD/MDA 기술은 모델 및 코드 간의 자동변환을 통한 소프트웨어 개발생산성을 향상시키기 위한 기술이다. 본 논문에서는 CBD 방법론으로 수행된 SI 프로젝트에서 MDD/MDA 기술 적용으로 인한 생산성 향상 효과를 기능점수(Function Point) 관점에서 분석하여 제시한다. 또한 본 사례의 프로젝트에서 적용한 MDD/MDA 적용 방법을 소개한다.

1. 서론

소프트웨어 개발에 있어서 추상화(abstraction)와 자동화(automation)라는 개념이 결합되기 시작하고 그 두 개념의 결합이 가져오는 무수한 잠재력들이 하나 둘씩 드러나면서 model-driven development (MDD)라고 통칭되는 새로운 모델링 기술이 대두되고 있다. MDD의 가장 큰 특징은 소프트웨어 개발의 전 과정에서 모델을 가장 중요한 산출물(primary artifact)로 여긴다는 점이다. 이는 곧 컴파일러가 특정 컴퓨팅 기계의 복잡성으로부터 개발자들을 자유롭게 했던 것처럼, 소프트웨어 모델이 프로그램 코드가 가지는 복잡성으로부터 벗어날 수 있다는 가능성을 의미한다. MDD는 모델로부터 프로그램 코드를 자동(혹은 부분적으로 자동) 생성을 가능하게 하는 기술에 대한 청사진을 제시하는 개념이다. 오늘날 MDD 개념을 사용한 코드의 자동 생성 수준은 간단한 스켈레톤 코드의 생성으로부터 컴파일러가 그랬던 것처럼 완벽히 실행 가능한 코드의 생성에 이르기까지 천차만별이다. 자동화 수준과 모델의 정확도는 상호간에 영향을 주고 받으면서 지속적으로 개선될 것이며, 이러한 과정을 통하여 MDD라는 개념이 소프트웨어 세계

에 끼치는 영향은 컴파일러가 가져왔던 파급효과를 능가하게 될 가능성이 크다. 사실, 소프트웨어 개발에 있어서 MDD라는 개념의 적용은 새로운 것이 아니며 과거에도 수없이 시도되어 왔다. 그럼에도 불구하고 MDD가 최근 들어서 관심의 초점이 되는 이유는 과거와는 비교할 수 없을 정도로 MDD를 성공적으로 적용시키기 위한 기반 기술이 발전되고 준비되어 있다는 것이다. 이러한 기반기술의 성숙은 MDD에 관한 표준의 제정을 가능하게 했으며, 이로 인하여 관련 도구의 통합이나 사용성 측면에서 급격한 발전을 초래하게 되었다. 이러한 기반기술의 선두에 서 있는 것이 바로 MDA(Model Driven Architecture)이다.

MDA는 MDD 개념을 실제 구현 가능하도록 하는 기반기술로서 2002년 OMG에서 공식적으로 발표되었다. OMG에서 제안한 MDA의 궁극적인 목표는 소프트웨어 개발의 전 과정에서 필요한 모델링 기술에 대한 표준을 제정하는 것이다. 소프트웨어 개발자는 이러한 모델 표준에 근거하여 특정 개발 도구에 종속되지 않으면서 소프트웨어 개발 과정에서 생성되는 모든 모델을 해당 모델 표준에 근거하여 통합하고 제어할 수 있게 된다. MDA의 기본을 이루고 있는 사상은 소프트웨어 자체에 대한 명세와 소프트웨어가 플랫폼을 사용하는 방법에 대한 명세를 분리하려는 데서 출발하고 있다. 이를 위하여 OMG에서는 소프트웨어 모델을 추상화 수준에 따라 CIM(Computer Independent Model), PIM(Platform Independent Model), PSM(Platform Specific Model), Code Model의 네 가지 유형으로 분리하고 있으며, 모델간의 Mapping을 통한 변환(Transformation)을 위한 기반을 만들어 가고 있다.

본 논문에서는 이러한 MDD/MDA 기술을 적용한 SI 프로젝트에서의 개발 생산성을 기능점수(Function Point) 단위로 측정하여 비교 분석함으로써 MDD/MDA 기술을 적용으로 인한 개발 생산성 향상 효과를 살펴본다. 또한 CBD 방법론으로 수행된 SI 프로젝트에서의 MDD/MDA 기술 적용 방법을 소개한다.

2. 본론

2.1 사례 프로젝트 개요

본 논문에서 제시하는 사례는 물류 업무를 산업 도메인으로 하는 D사의 물류통합시스템 구축 프로젝트이다. 프로젝트에 대한 개요는 다음 표와 같다. 개발범위는 D사의 핵심 프로세스인 물류 업무 프로세스와 지원 프로세스인 회계 프로세스를 포함한다.

항목	설명
프로젝트 명	물류통합시스템 구축프로젝트
기간	2005년5월9일 ~2006년4월15일
업무도메인	물류
서브시스템	물류시스템, 회계시스템
개발 방법론	WAY4U-CBD(LG CNS CBD방법론)
모델링언어	UML 1.4
모델링도구	IBM Rational Rose
구현 플랫폼	J2EE

표 1. 사례 프로젝트 개요

2.2 개발생산성 향상 효과 측정

본 논문에서는 개발 규모를 기능점수(Function Point)로 산출하여 개발 생산성을 측정하였으며, 이를 위하여 정보통신부에서 고시한 [소프트웨어 대가기준(제2005-22호)]과 LG CNS의 견적모델을 활용하였다. 다음은 데이터 및 트랜잭션 기능점수로 산출된 사례 프로젝트의 개발 규모이다.

구분	데이터 기능		
	총계	내부논리파일	외부연계파일
물류	3017.0	3017.0	0
회계	1023.2	980.0	43.2
계	4040.2	3997.0	43.2

표 2. 데이터 기능점수

위 데이터 기능점수에서 물류시스템의 외부연계파일에 대한 기능점수는 해당 연계를 EDI 솔루션

패키지를 사용하여 구현하였으므로 본 논문에서는 제외하였다. 한편, 트랜잭션 기능점수는 다음 표와 같이 산출되었다.

구분	트랜잭션 기능			
	총계	외부입력	외부출력	외부조회
물류	7853.6	3578.4	371	3904.2
회계	2976.3	1392.8	451	1132.5
계	10829.9	4971.2	822	5036.7

표 3. 트랜잭션 기능점수

트랜잭션 기능점수의 기능 유형별로 평균 가중치를 살펴보면 다음 표와 같다. 다음 표의 우측에 표현된 기준 가중치는 [소프트웨어대가기준(제2005-22호)]에서 제시한 평균 복잡도 가중치 값이다. 물류 및 회계 시스템 모두가 기준 가중치보다 높으며, 회계 시스템보다 물류 시스템이 평균 가중치가 높게 나타나고 있다. 이는 물류 및 회계 시스템이 일반적인 시스템보다 복잡도가 높으며, 회계 시스템보다 물류 시스템의 복잡도가 더 높음을 의미한다.

구분		총계	물류	회계	기준
외부 입력	기능	1052	711	341	
	FP	4971.2	3578.4	1392.8	
	가중치	4.73	5.03	4.08	3.9
외부 출력	기능	131	52	79	
	FP	822	371	451	
	가중치	6.27	7.13	5.71	5.0
외부 조회	기능	1075	827	248	
	FP	5297	3904.2	1392.8	
	가중치	4.93	4.72	4.57	3.7
총계	기능	2258	1590	668	
	FP	10829.9	7853.6	2976.3	
	가중치	4.80	4.99	4.46	

표 4. 트랜잭션 기능별 평균 가중치

위에서 산출된 데이터 기능점수와 트랜잭션 기능점수를 합한 총 개발규모는 다음 표와 같다. 아래 표 중 보정총계는 [소프트웨어대가기준(제2005-22호)]에서 제시한 규모보정계수 (0.108 * LOGe(기능점수) + 0.2229)를 사용하였다. 규모보정계수를 적용한 총 개발규모는 회계 시스템이 4474 FP, 물류 시스템이 13334.9 FP로서 총계가 17808.9 FP로 나타났다.

구분	기능			
	데이터	트랜잭션	총계	보정총계
물류	3017.0	7853.6	10870.6	13334.9
회계	1023.2	2976.3	3999.5	4474.0
계	4040.2	10829.9	14863.1	17808.9

표 5. 총 개발 규모 (Function Point)

본 논문에서는 개발 생산성 측정 지수로 FP/MM를 사용하였다. 사례 프로젝트에서 측정된 개발 생산성 지수는 다음과 같다. 참고로 사례 프로젝트에서의 1MM 당 근무시간은 228시간이다.

구분	총 기능점수 (FP)	총 투입공수 (MM)	개발 생산성 (FP/MM)
물류	13334.9	355.1	37.55
회계	4474.0	80.3	55.72
총계	17808.9	435.4	40.90

표 6. 사례 프로젝트의 개발 생산성

[표4]에서 나타난 것처럼, 트랜잭션 기능의 평균 가중치가 상대적으로 높은 물류 시스템의 개발 생산성이 회계 시스템의 개발 생산성 보다 더 낮은 측정치를 보이고 있다. 이는 시스템의 평균 복잡도와 개발 생산성 간에 역의 상관관계가 있음을 의미한다. 즉, 평균 복잡도가 높은 시스템일수록 개발 생산성은 낮다는 것을 본 사례가 실증적으로 보여주고 있다.

한편, 사례 프로젝트와 국내 및 국제 프로젝트의 평균 개발 생산성을 비교하면 다음 표와 같다. 국내 프로젝트의 평균 개발 생산성은 한국소프트웨어기술진흥협회(KOSTA)의 2005년 통계 자료이며, 국제 프로젝트의 평균 개발 생산성은 ISBSG(International Software Benchmarking Standards Group)의 통계 자료이다. 개발 시간의 편차를 고려하기 위해 1MM 당 시간은 200시간을 기준으로 하여 개발 생산성 지수를 재조정 하였다. 개발 생

산성은 사례 프로젝트가 35.88 FP/MM, 국내 프로젝트 평균이 17.89 FP/MM, 국외 프로젝트 평균이 17.12 FP/MM으로 나타나고 있다. 사례 프로젝트의 개발 생산성이 국내 프로젝트 평균 개발 생산성 대비 100.6 %, 국외 프로젝트 평균 개발 생산성 대비 109.6%의 생산성 향상률을 보이고 있다.

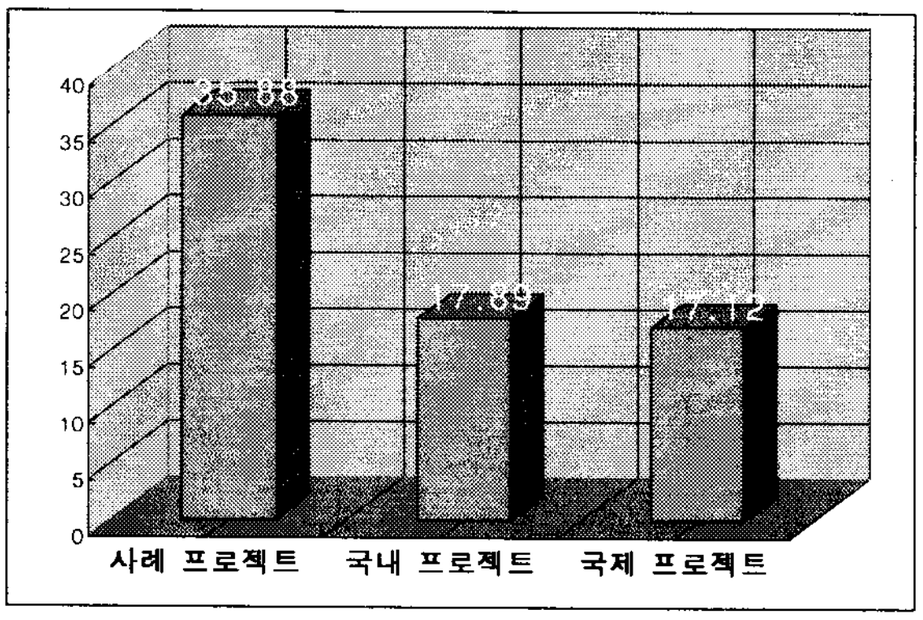

표 7. 개발 생산성 비교

이러한 생산성 향상 효과는 본 사례 프로젝트에 MDD/MDA 기술을 적용하여 모델 및 코드를 자동 생성함으로써 얻어진 것으로 추론해 볼 수 있다. 다음 그림은 사례 프로젝트에 적용된 계층 아키텍처 (Layer Architecture)이다.

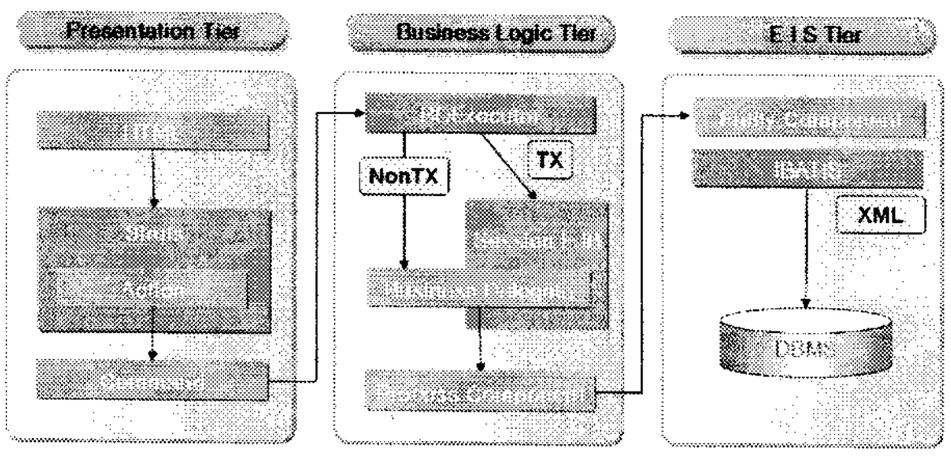

그림 1. 사례 프로젝트의 계층아키텍처

위 그림에 표현된 각 계층들 중 코드 자동 생성의 대상이 된 계층은 Command, Business Delegate, Process Component, Entity Component 계층이다.

코드 생성률은 자동으로 생성된 코드 라인 수를 구현이 완료된 최종 코드의 총 라인 수로 나누어 산출하였다. 다음 표는 코드 자동 생성의 대상이 되는 각 계층(Layer)별 코드 생성률을 보여주고 있다.

구분	총계 (%)	물류 (%)	회계 (%)
Command	85.01	84.71	86.18
Business Delegate	97.69	98.97	93.50
Process Component	37.95	34.03	70.67
Entity Component	90.97	90.27	95.92
총계	70.89	68.18	85.40

표 8. 코드 생성률

회계 시스템의 코드 생성률이 85.40 %로 물류 시스템의 68.18%보다 높은 것을 알 수 있다. 이러한 코드 생성률의 차이는 [표6]에 나타난 두 시스템간의 개발 생산성의 차이와 정의 상관관계가 있음을 의미한다. 즉, 코드 생성률이 높을수록 개발 생산성도 높아진다는 것을 본 사례가 실증적으로 보여주고 있다.

계층별로 코드 생성률을 살펴보면, Process Component의 코드 생성률이 다른 계층에 비해 상당히 낮게 나타남을 볼 수 있다. 이는 대부분의 업무규칙이 Process Component에서 구현되며, 이러한 업무규칙을 구현하는 코드는 개발자에 의해 직접 구현되어야 하기 때문이다. Entity Component의 경우에는 DBMS에 접근하여 해당 정보를 생성, 수정, 삭제, 조회하는 서비스를 주로 제공하므로 코드 생성률이 상대적으로 높게 나타났다. 다음 표는 업무규칙과 코드 생성률과의 상관관계를 보여주고 있다.

구분	Process Component		Entity Component	
	코드 생성률	제어문 개수	코드 생성률	제어문 개수
물류	34.03	4729	90.27	641
회계	70.67	356	95.92	65
총계	37.95	5085	90.97	706

표 9. 업무규칙과 코드 생성률과의 상관관계

업무규칙은 일반적으로 제어문을 사용하여 구현하게 되므로 코드에 구현된 업무규칙의 양을 측정하기 위해서 구현이 완료된 코드에서 제어문의 개수를 산출하였다. 위의 표에서 볼 수 있는 것처럼 제어문의 개수가 많을수록 코드 생성률은 현저히 떨어지는 것을 볼 수 있다.

2.3 MDD/MDA 기술 적용 방법

본 사례 프로젝트에서는 MDD의 기본 개념과 OMG에서 발표한 MDA 가이드를 프로젝트 환경에 적합하게 변경하여 적용하였다. 다음 그림은 사례 프로젝트에서 적용한 MDD/MDA 기술 적용 방법에 대한 개념도이다.

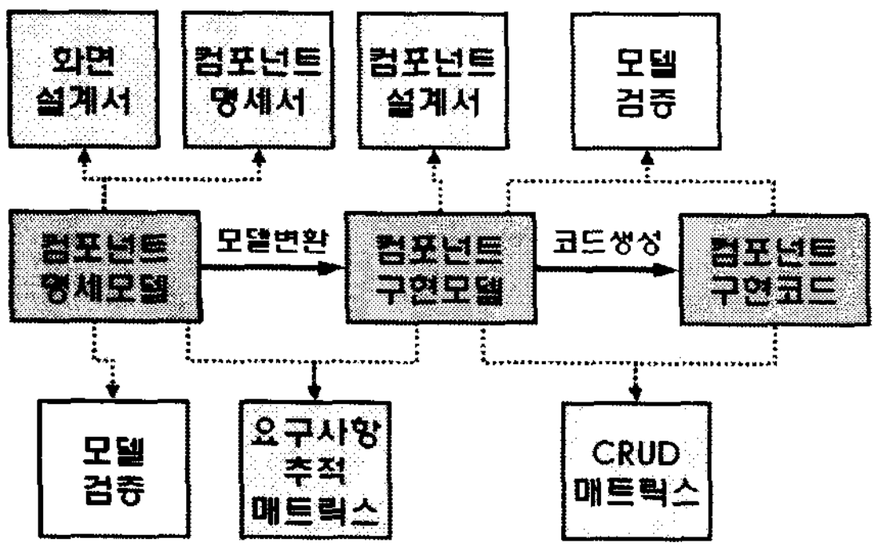

그림 2. MDD/MDA 기술 적용 방법

위 그림에서 볼 수 있는 것처럼, 사례 프로젝트에서는 모델변환, 코드생성, 모델검증, 산출물 생성 등의 네 가지 측면에서 MDD/MDA 기술을 적용하였다. 본 논문에서는 사례 프로젝트에 적용된 모델 변환 방법을 주로 살펴보도록 하겠다.

다음 그림은 사례 프로젝트에 적용된 모델 변환 방법을 보여주고 있다. 컴포넌트 명세 모델 및 컴포넌트 구현 모델에 대한 메타모델을 먼저 정의한 후, 이들 메타 모델을 분석하여 메타모델간의 변환규칙을 정의하였다. 모델 변환규칙은 실행 가능한 형태의 모델변환 컴포넌트로 구현하였으며, 개발 언어는 Rose Script를 사용하였다. 컴포넌트 명세 메타 모델을 준수하여 컴포넌트 명세 모델을 설계한 후에, 이 명세 모델을 입력으로 하여 모델 변환 컴포넌트를 실행하면 컴포넌트 구현 모델을 생성할 수 있다.

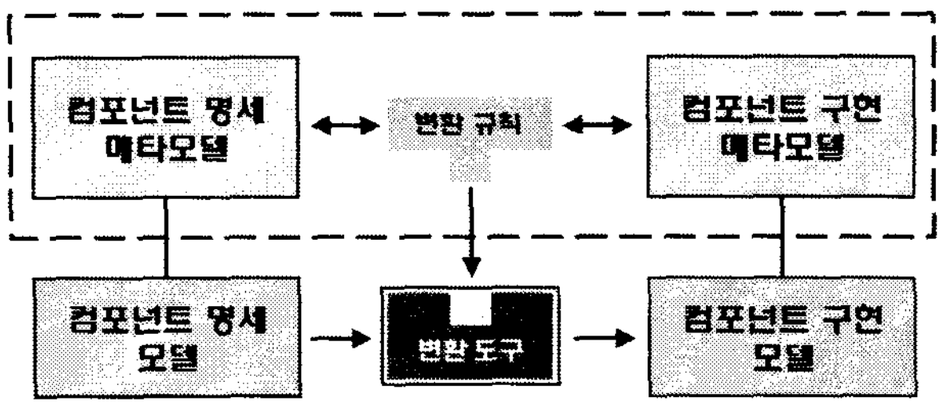

그림 3. 모델 변환 방법

컴포넌트 명세 메타모델은 컴포넌트 명세 모델을 설명하는 메타모델로서 본 사례 프로젝트에서는 UML Profile을 이용하여 컴포넌트 명세 메타모델을 다음 그림과 같이 정의하였다.

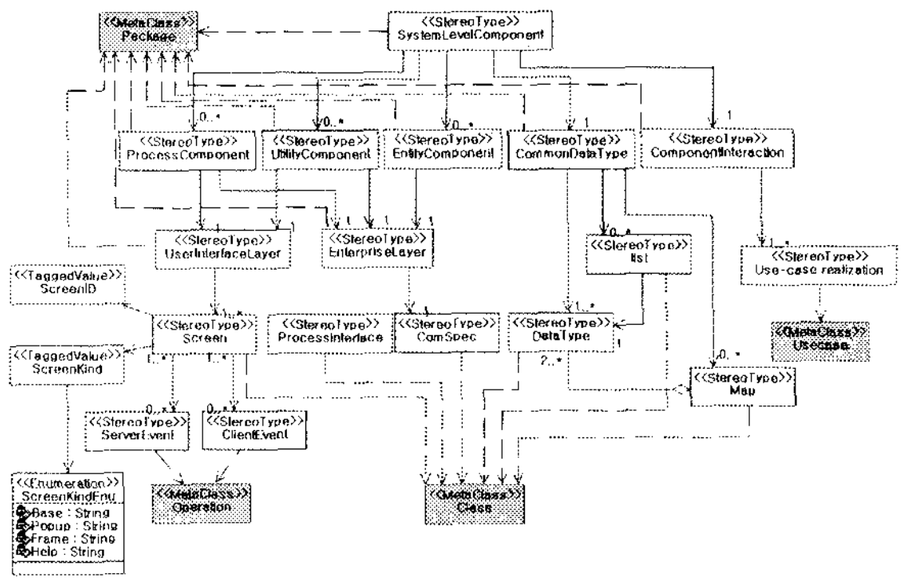

그림 4. 컴포넌트 명세 메타 모델

컴포넌트 구현 메타모델은 컴포넌트 구현 모델을 설명하는 메타모델로서 본 사례 프로젝트의 SW 아키텍처를 고려하여 컴포넌트 구현 메타 모델을 다음 그림과 같이 정의하였다.

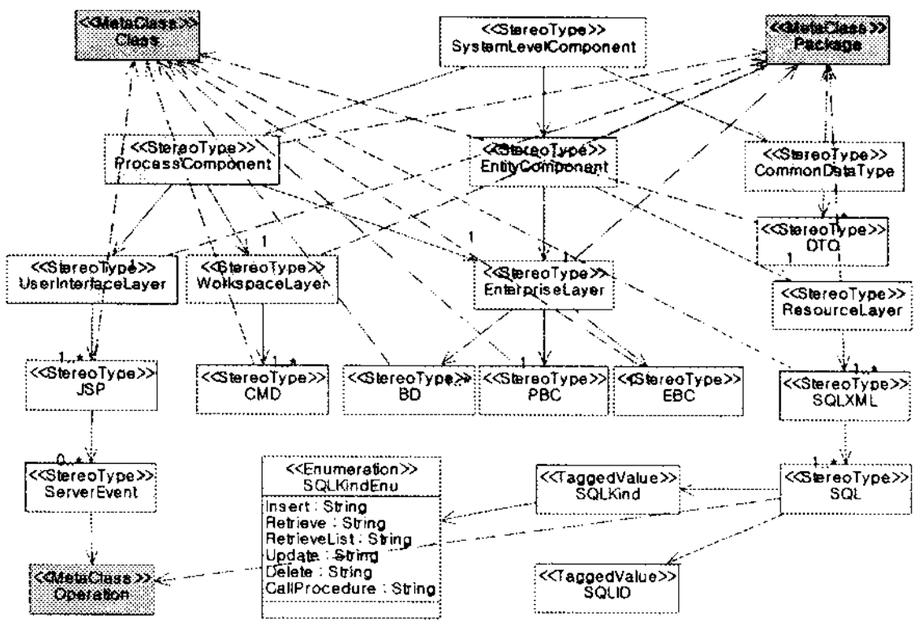

그림 5. 컴포넌트 구현 메타 모델

위의 컴포넌트 명세/구현 메타 모델을 분석하여 두 메타모델간의 변환규칙을 정의한다. 다음 그림은 프로세스 컴포넌트(Process Component)에 대한 명세 메타모델과 구현 메타모델간의 변환규칙을 보여주고 있다.

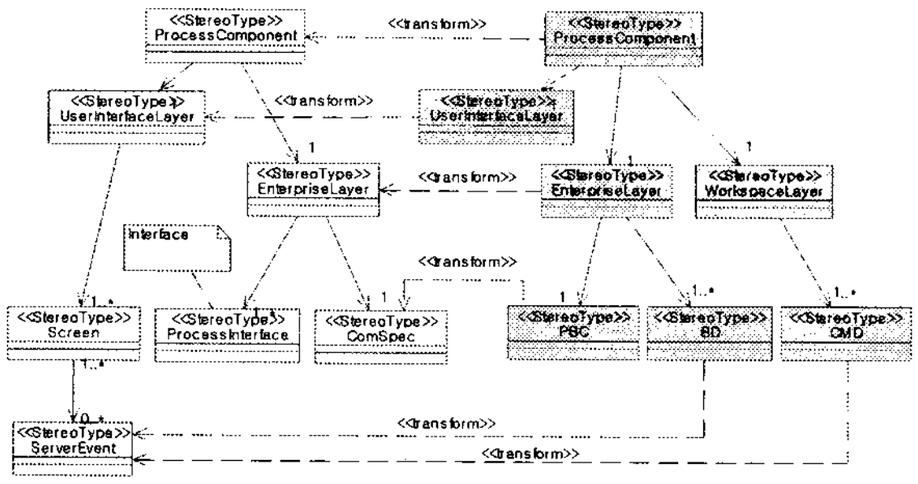

그림 6. 모델 변환 규칙

모델 변환규칙은 실행이 가능한 컴포넌트로 구현되어야 한다. 본 사례 프로젝트에서는 Rose Script를 이용하여 모델 변환규칙을 구현하였다. 다음 그림은 컴포넌트 명세모델을 입력으로 모델 변환 규칙을 실행하여 컴포넌트 구현모델을 생성하는 그림이다.

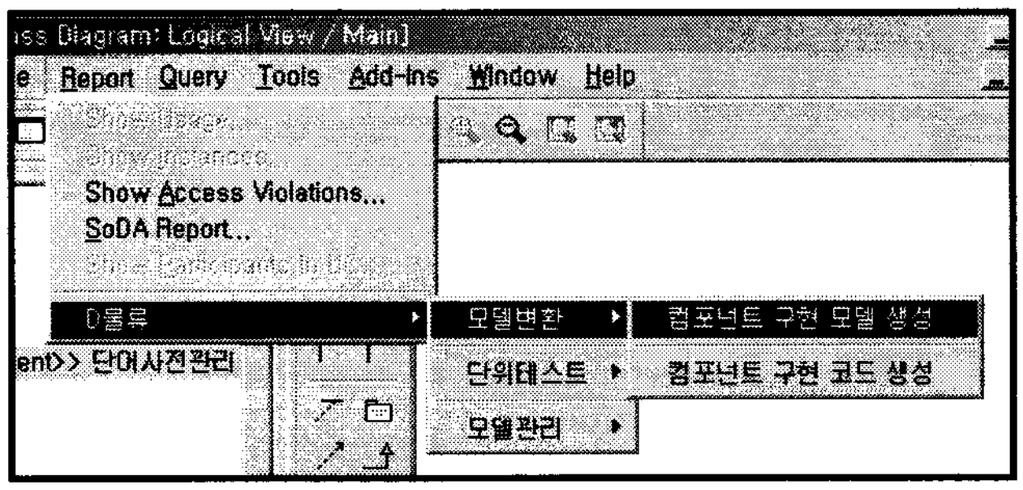

그림 7. 모델 변환규칙의 구현 및 실행

한편, 사례 프로젝트에서 적용한 컴포넌트 구현 모델로부터 코드 생성 방법은 다음 그림과 같다. 컴포넌트 구현 메타모델과 코딩 표준 및 코딩 템플릿을 분석하여 변환규칙을 정의하고 이를 실행 가능한 컴포넌트로 구현한 다음, 컴포넌트 구현모델을 입력으로 변환 컴포넌트를 실행하여 코드를 생성하였다.

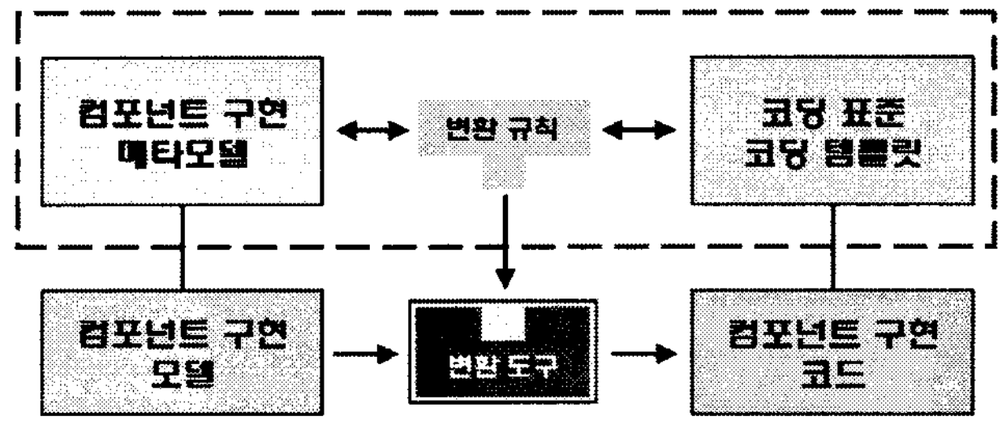

그림 8. 코드 생성 방법

또한, 컴포넌트 명세 및 구현 모델에 포함된 정보들을 읽어 화면설계서, 컴포넌트 명세서, 컴포넌트 설계서, 요구사항추적매트릭스, CRUD매트릭스 등의 산출물을 자동으로 생성하였으며, 메타모델을 기준으로 모델검증 규칙을 정의한 후, 이를 검사하는 모델검증 컴포넌트를 개발하여 모델검증을 자동화하였다.

3. 결론

MDD/MDA 기술은 소프트웨어의 개발을 자동화하는 기술로서, 본 사례 프로젝트에서는 모델변환, 코드생성, 산출물 생성, 모델검증 등의 네 분야에

서 이 기술을 적용하였다. 그 결과 개발 생산성 측면에서 국내 및 국제 프로젝트의 평균 개발 생산성과 비교하여 100% 이상의 개발 생산성 향상 효과를 가져왔다. 또한 코드와 산출물을 모델로부터 자동 생성함으로써 설계와 코드 및 산출물간의 일관성을 유지하기가 용이해졌고 모델검증의 자동화를 통하여 소프트웨어의 설계 품질을 보증하는 활동을 체계적으로 수행하는 것이 가능해졌다.

다만, 사례 프로젝트에서 적용한 모델링 언어(UML 1.4)에서는 업무규칙을 모델로 표현할 수 있는 정도에 한계가 있어서 이와 관련된 코드는 수작업으로 진행할 수 밖에 없었으며, 이로 인하여 개발 생산성을 떨어뜨리는 효과를 가져왔다. 앞으로 MDD/MDA 기술의 발전에 따라 지속적인 연구가 필요한 부분이라 하겠다.

[참고문헌]

[1] 소프트웨어사업대가의 기준 (정보통신부 고시 제2005-22호): 정보통신부, 2005

[2] 정보시스템 구축의 생산성관리 및 향상 방안: 김현수 교수, KCSC SW 생산성 지표 개발 작업반, 2005

[3] 기능점수(FP)에 의한 간편한 소프트웨어 측정방법 검토: 황인수, SDS, 2002

[4] MDA Guide Version 1.0.1: Joaquin Miller and Jishnu Mukerji, OMG, 2003

[5] Model Transformation: The Heart and Soul of Model-Driven Software Development: Shane Sendall,, Wojtek Kozaczynski, IEEE Software, 2003

[6] Model-Driven Development: A Metamodeling Foundation: Colin Atkinson, Thomas Kühne, IEEE Software, 2003

[7] Model Driven Process Engineering: Erwan Breton, Jean Bézivin, Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC.01), 2001

[8] MDA Distilled: Principles of Model-Driven Architecture: Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise, Addison Wesley, 2004

[9] MDA Explained: The Model Driven Architecture™: Practice and Promise: Anneke Kleppe, Jos Warmer, Wim Bast, Addison Wesley, 2003

[10] An MDA Manifesto: Grady Booch, Alan Brown, Sridhar Iyengar, James Rumbaugh, Bran Selic, MDA Journal, 2004