코드 생성을 위한 UML 확장

민현석
고려대학교 컴퓨터정보통신학과 정형기법연구실
e-mail : hsmin@korea.ac.kr

UML Extension for Code Generation

Hyunseok Min
Dept. of Computer Science, Korea University

요 약

OMG 가 시작한 MDA(Model Driven Architecture) 는 소프트웨어 개발자들사이에 빠르게 전파되고 있다. UML 은 OMG 에 의해 MDA 를 위한 언어로 선택되었는데, UML 은 PIM(Platform Independent Model)에서 PSM(Platform Specific Model)을 생성하기에는 충분하지 않다. 하지만, 이러한 PIM-PSM 변환을 가능한한 자동화할수 있는데 이 논문은 자동 코드 생성을 위해 UML 의 확장 방법인 Stereotype 과 Tagged-Value 에 대해 논하게 된다. 또한, Aspect-Oriented 접근을 위해서 AOP 로 확장된 UML 에서 비 AOP 언어로 코드 생성이 가능하게 되는 새로운 방법도 제안을 한다.

1. 서론

UML 은 소프트웨어 모델링에 있어서 실질적인 표준이다. UML 은 작은 내장형 시스템에 쓰이는 프로그램에서 거대한 프로젝트에 이르기까지 전 산업 분야에서 쓰여지고 있다. UML 은 애초에 소프트웨어 디자인을 위해서 만들어졌고 프로그래머들간의 대화가 쉽게 되도록 고안 되었다. 하지만 제대로 된 MDA 지원이 없이는 모델과 소스 코드를 동기화 시키기가 매우 어렵다. 그래서 인기있는 CASE(Computer Aided Software Engineering) 툴들은 UML 모델로부터의 소스 코드 생성을 지원하고 있다. 이는 모델과 코드의 동기화를 견고 하게 하며 그 결과 프로그래밍을 좀 더 효율적으로 할 수 있게 한다. 그러나 UML 그 자체는 모든 상세한 구현을 하는 데에 있어서 충분하지도 않고 가능하지도 않다. 하지만, 가능한한 좀 더 많은 자동 코드 생성을 하도록 개선시키는 것은 가능하며 이는 반복되는 코딩 작업에서 프로그래머들의 짐을 덜어줄 수 있게 된다. 이는 robust 한 구현을 도와줄 뿐만 아니라 소프트웨어 생산성에도 중요하다. 이 논문의 관심은 이를 이루는 방법들이다.

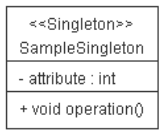
2. UML 에서 소스 코드로의 변환

UML 의 13 개 그림가운데서 소스 코드의 생성에 쓰여질 수 있는 그림은 대략 5 개가 된다. 우선 Class Diagram, Object Diagram, Structure Diagram 들이 있는데 이들은 클래스와 그 인스턴스의 정적인 구조를 보여주는데 사용된다. 이들은 소스 코드 생성에 직접적으로 이용될 수 있고 대부분의 CASE 툴들은 이러한 기능들을 가지고 있다. 다른 2 개의 그림들인 Statechart 와 Activity Diagram 은 근본적으로는 같은 것이라고 할 수 있는데 객체의 행위를 설명하는데 쓰이게 된다. 많은 상업용 CASE 툴들이 이 2 개의 그림으로부터 소스 코드를 생성해 낼 수 있다. 이는 특히 Statechart 가 수학적으로 잘 정의가 되어있기 때문인데 상업용 툴들은 이 그림들 에서의 코드 생성을 그들 툴의 장점으로 내새우고 있다.

이 논문에서는 행위를 나타내는 코드 즉, Statechart 와 Activity Diagram 은 자동 코드 생성에 고려하지 않는다.

2.1. Stereotype 기반의 코드 생성

첫번째 예는 잘 알려진 디자인 패턴인 Singleton pattern 이다. 이 패턴은 패턴을 아는 대부분의 프로그래머들에게 잘 알려져 있으며 매우 간단한다. 프로그램이 살아있는 동안 단 하나의 객체만 가지게 되는 클래스는 (그림 1)과 같이 <<Singleton>> Stereotype 을 가질수 있다.



(그림 1) Singleton stereotype 을 가지는 클래스

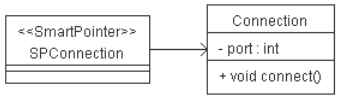
이제 이 모델에서의 코드 생성은 자동적으로 private 생성자와 하나뿐인 객체를 얻기 위한 static method 를 포함하게 된다. 간단한 Java 의 구현을 (그림 2)에 보인다.

그런데, 사용자는 툴벤더에 의해 제공된 기본 구현을 바꾸고 싶을수도 있다. 이를 위해 툴은 기본 구현을 보여주고 이를 바꿀수 있는 방법을 제공해야 한다.

```
//SampleSingleton.java/////////////////////////////////////

class SampleSingleton
{
        // static method for getting a one and only instance of t
        public static SampleSingleton instance() {
                if(sInstance == null)
                        sInstance = new SampleSingleton();
                return sInstance;
        }

        private SampleSingleton() {

        }

        static SampleSingleton* sInstance = null;

        private int attribute;

        public void operation() {
                // implementation here
        }
}
```

(그림 2) Stereotype 에서 생성된 코드

두번째 예는 Smart Pointer 이다. Smart Pointer 클래스는 자동으로 생성되지 않는다면 손으로 일일이 이 클래스들을 작성하는 것은 매우 빠르게 부담으로 작용하게 된다. 물론 C++의 auto_ptr<> 와 같이 일반적으로 사용할 수 있는 라이브러리도 있으나 템플리트의 사용이 모든 개발 환경에서 잘 지원되지도 않으며 그 작동을 사용자가 바꿀수도 없다. 그래서 SmartPointer 를 Stereotype 으로 사용하여 Smart Pointer 클래스를 자동으로 생성할 수 있다면 반복되는 지루한 프로그래밍 작업을 쉽게 줄일 수 있다.



(그림 3) SmartPointer class diagram

```
//SPConnection.h/////////////////////////////////////
#ifndef _SPCONNECTION_H
#define _SPCONNECTION_H

class SPConnection
{
public:
        explicit SPConnection(Connection* p);
        ~SPConnection();
        Connection* operator->();
        Connection& operator*();

private:
        Connection* itsConnection;
};

#endif //_SPCONNECTION_H
```

(그림 4) Stereotype 에서 생성된 헤더 파일

(그림 3)은 <<SmartPointer>> Stereotype 을 가지면서 다른 클래스로 association 을 가지는 클래스를 보여준다. 이 association 은 Smart Pointer 로 wrap 할 raw pointer 를 가리킨다. (그림 4)는 이 UML Stereotype 으로 생성된 코드를 보인다.

Java 언어의 경우는 직접적인 메모리 해제 방법이 없기 때문에 스마트 포인터 자체가 무의미하다. 그래서 UML 모델에서 Stereotype 이 <<SmartPointer>> 인 클래스는 코드 생성을 안하는 것이 더 이치에 맞을 뿐만 아니라 스마트 포인터를 변수로 가지는 클래스는 실제 raw pointer 를 변수로 가지도록 코드 생성이 되는 것이 바람직하다. UML 을 여러가지 프로그래밍 언어로 사용을 하기 위해서는 이런 언어간의 차이를 잘 인식하고 거기에 맞는 코드 생성을 하는 것이 중요하다.

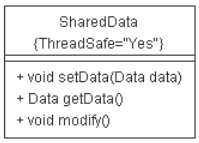
이것들은 Stereotype 이 어떻게 자동 코드 생성에 사용될 수 있는지에 대한 작은 예들이다. CASE 툴들이 이러한 잘 알려진 idiom 들을 정의하고 기본 구현을 제공하여 사용자는 단지 Stereotype 을 선택하여 코드를 생성시킬 수 있도록 하는 것이 중요하다.

디자인 패턴은 잘 알려져 있으므로 이러한 예들은 더 많이 나올수 있다. 예를 들어 <<Visitor>> 라는 Stereotype 은 visit 라는 함수를 자동으로 더하고 <<Visitable>> Stereotype 은 accept 함수를 자동으로 더할수 있다. 이러한 함수들은 상속을 받는 함수에도 자동으로 더하여져야 한다. 이러한 사용 방식은 그 외에도 많다. Stereotype <<Observer>>는 update 함수를 자동으로 더하고 <<Notify>> Stereotype 은 변수로 vector<Observer*> 또는 ArrayList<Observer> 타입을 가지면서 여기에 필요한 함수를 추가할 수 있다. 만약 각각의 디자인 패턴들에 대한 naming convention 이 모든 프로그래머들 사이에 약속된다면 이러한 자동 코드 생성은 더 쉬어지게 될 것이다. Stereotype <<Factory>> 도 또다른 예제가 될 수 있는데, <<Factory>> 구현은 이 클래스들간의 상속 관계가 필요해지고 각 클래스들의 creation 함수가 필요하여 질거여서 이름 규칙이 개발자들간에 합의가 되어야 할 것이다.

2.2 Tagged value

어떤 Property 들은 Stereotype 으로 하기 부적절한 경우도 있다. Thread-safe 한 클래스를 고려해보자. C++ 에서라면 이는 Mutex 나 Semaphore 를 이용하여 모든 함수를 lock/unlock 으로 보호하게 될 것이다. 그런 클래스에 Stereotype 으로 <<ThreadSafe>>이나 <<Guarded>>를 주는 것도 가능한데 이 경우는 클래스가 가지는 많은 특징중에 하나라고 할 수 있다. <<ThreadSafe>>를 Stereotype 으로 만드는게 문제는 아니지만 모든 것을 Stereotype 으로 만들어 더하는 것은 오히려 그 클래스의 특징을 희석시키고 이해를 어렵게 할 여지가 있다. 이런 경우에는 UML 의 확장 방법중의 하나인 Tag 를 이용할 수 있다. (실제로 Tag 는 Stereotype 을 포함한다고 할수도 있다.) Tag 는 Value 와 쌍을 이루게 되는데(그래서 Tagged-Value 로 불린다.) 이 경우에는 ThreadSafe 이 Tag 로 추가되는 것이 더

바람직하다. (그림 5)에는 Tagged-Value 가 추가된 클래스를 보이며 (그림 6)에는 거기서 생성된 코드를 보여준다.



(그림 5) ThreadSafe Tag

```
// SharedData.cpp
#include "SharedData.h"

void SharedData::setData(Data data) {
        // implementation here
        mutex.lock();

        mutex.unlock();
}
Data SharedData::getData() {
        // implementation here
        mutex.lock();

        mutex.unlock();
}
void SharedData::modify() {
        // implementation here
        mutex.lock();

        mutex.unlock();
}
```

(그림 6) Tagged Value 에서 생성된 코드

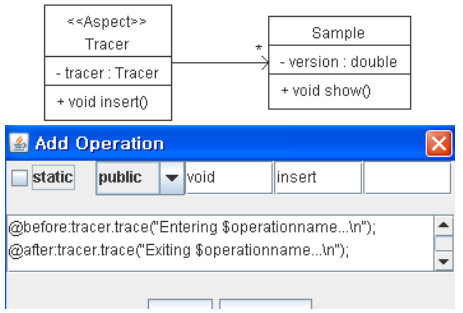
다른 예제로는 Value-Copy-Protection 을 둘수 있다. 이건 Java 와 같이 모든 객체가 레퍼런스로만 참조되는 언어에는 적용되지 않지만 C++의 객체는 함수로의 전달시 비싼 복사 비용이 드는 객체 자체로의 전달이 가능하다. 일반적으로는 거의 대부분의 경우 이런 식의 카피는 잘 짜인 코드가 아니어서 Copy-by-value 는 기본적으로 금지가 되어야 한다. Copy-by-value 는 약간의 특별한 케이스 (std::string 과 같은) 에만 허용되어야 한다. 이러한 복사를 막는 간단한 방법은 복사 생성자와 대입 연산자를 private 으로 선언하는 것이다. 이 경우 사용을 안 하려는 것이므로 특별한 구현 자체도 필요없다. 대부분의 클래스가 이러한 보호가 필요하므로 UML 에서 자동으로 이러한 코드가 생성되면 좋을 것이다. 또한, 이 보호가 필요한 클래스가 많으므로 이를 Stereotype 으로 정하기도 부적절하다. 이 경우는 그래서 NoValueCopy 라는 Tag 를 하나 설정하여 그 값을 Yes/No 로 주는 방법이 적합하다. 보통 Tagged-Value 는 UML 다이어그램에 꼭 표시가 될 필요는 없는데, 코드 생성에 사용된다면 사용자의 이해를 돕기 위해 그림에 보이는 것도 좋을 것이다. 또한, 코드 생성에 상관 없는 Tag 들과 구별을 주는 것도 가능하다.

2.3. AOP 를 위한 UML 확장

마지막으로 중요하게 취급될 수 있는 것은 Aspect 이다. Aspect-Oriented Programming 은 10 여년전에 나왔으나 학계의 관심과 흥미에 비해 실제 산업계에서는 그리 크게 쓰여지고 있지는 않다. 이는 AOP 가 현재의 프로그래밍 언어에서 사용될 수도 있으나 그 복잡성 때문에 연구 방향이 새로운 프로그래밍 언어를 사용하는 쪽으로 향해있기 때문이다. 학계에는 이미 AOP 를 UML 로 표기하는 것에 대한 많은 논문이 나와있다. 이들은 일반적으로 AOP 를 UML 로 그리는 방법을 얘기하고 있는데 OOP 에 기반한 UML 은 AOP 를 그냥은 나타낼 수가 없으므로 앞에서 언급한 Stereotype 이나 Tagged-Value 를 이용한 새로운 Profile 을 이용하게 된다. [5], [6]들이 AOP 를 위한 UML extension 을 언급하고 있다.

AOP 가 현재의 프로그래머들에게 쉽게 사용되려고 한다면 Aspect 를 UML 로 표기는 하되 그 생성되는 언어는 프로그래머가 현재 매일 쓰고 있는 언어로 하는 것이 더 효율적일 것이다. 프로그래머들은 UML 다이어그램에 이해하기 쉬운 방법으로 AOP 를 정의된 확장 방법으로 기술한다. 그 다음 CASE 툴이 이를 프로그래머들이 편하게 여길 수 있는 C++이나 Java 와 같은 언어로 변환한다. 이러한 방법은 AOP 를 쉽게 사용할 수 있으면서도 실제 프로그램 소스를 보는 것과 디버깅은 기존에 익숙해 있는 방법으로 할 수 있게 된다.

첫번째 예로 들 Aspect 는 친숙한 Trace/Logging 기능이다. 디버깅을 위해 모든 함수들이 함수 이름을 남기는 entry/exit 트레이스를 남긴다고 하자. 이것은 꽤나 지루한 작업이고 새로운 함수가 추가될 때마다 계속 유지하기는 쉽지 않은 작업이다. 이러한 목적을 위해서 새로운 Stereotype 인 <<Aspect>>가 쓰여질 수 있다. (그림 7)에 <<Aspect>> 를 가지는 Tracer 클래스와 여기에서 연관 관계를 가지는 Sample 클래스가 보인다. 그런데, 이 연관 관계가 일반적인 클래스들간의 관계와는 다른데, 여기서는 일단 * multiplicity 를 가지는 것으로 표현 되어 있다.



(그림 7) Tracer Aspect class 와 weaving function

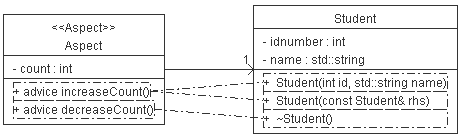
Tracer 클래스는 멤버 함수를 가지고 있는데 이는 crosscutting 이어서 함수의 구현에 before/after 를 붙일 수 있다. 이 crosscutting 함수의 이름은 중요하지 않고 구현에는 나중에 바뀌게 되는 $operationname 이라는 키워드를 사용하게 된다. 두 클래스간의 관계는

multiplicity *로 표현되는데 이는 이 Aspect 가 클래스 단위로 즉, 모든 함수에 적용이 된다는 뜻이다. (그림 8)을 보면 모든 trace 문이 함수의 시작과 끝에 추가가 된다. 이러한 방법으로 Sample 클래스의 경우는 새로 함수가 추가되어도 자동으로 트레이스문이 추가가 될 것이다.

```
// Sample.cpp
#include "Sample.h"

void Sample::show() {
        // implementation here
        tracer.trace("Entering Sample::show...₩n");
        int a = 0;
        int b = 0;
        //.....
        tracer.trace("Exiting Sample::show...₩n");
}
```

(그림 8) Trace 가 추가된 소스 코드

그러나 어떤 Aspect 는 단지 몇 개의 함수에만 weaving 할 필요가 있다. 이경우 association 이 아까와 같이 multiplicity *를 가지는 것이 아니라 직접적으로 관심이 있는 함수들로 매핑을 해도 된다. (그림 9)에 이런 관계를 보인다.



(그림 9) 개별 함수로 weaving 되는 Aspect 함수

이러한 매핑은 임의의 함수들로 확장될 수 있다. AspectJ 와 같은 AOP 언어는 정규표현식과 같은 방법으로 일종의 name-matching 방법을 사용하고 있는데 이는 사용이 어렵지는 않지만 약간 복잡한 경우라면 사용하기가 쉽지 않다. 그러나 UML 식의 접근이라면 이럴 필요가 없다. Weaving 하는 함수들은 직접적으로 매핑을 하여 어느 함수에건 적용될 수 있다. 물론 매핑을 하여야 할 함수의 개수가 많아지면 이것 역시 문제점을 가지게 되어 2 가지방법을 제안한다. 우선 이미 사용되고 있는 정규표현식이다. 툴벤더들은 이를 이용할 수 있는데 이러한 정규표현식은 다시 Tagged-Value 로 사용할 수도 있다. 또하나의 방법은 함수들에 Stereotype 을 이용하는 것이다. 앞의 예와 같이 count 변수를 하나 증가시켜야 하는 함수는 Stereotype 으로 <<count_inc>>와 같은 이름을 주고, Aspect 클래스에도 마찬가지로 count 변수를 증가시킬 weaving 함수에 <<count_inc>>를 Stereotype 으로 준다. 이렇게 하면 나중에 매핑시 같은 Stereotype 을 가진 함수끼리 자동으로 매핑을 하면 된다. 이 방법은 여러 가지 변형된 방법이 가능한데, Aspect 클래스의 함수가 Stereotype 을 가지는 대신에 이 함수에서의 association 이 Stereotype 을 가지고 같은 Stereotype 을 가지는 함수로 매핑하는 것도 가능하다. 또는 weaving 되는 클래스의 함수들 중 같은 Stereotype 을 가진 함수들만 그룹을 만들어서 이 그룹으로 직접 매핑이 되게 하여도 된다.

3. 결론

생산성은 항상 소프트웨어 엔지니어링의 중심 과제가 되어왔다. 현재는 많은 CASE 툴들이 어느 정도 생산성 향상을 도와주고 있는데 특히 코드 생성에 있어서는 개선의 여지가 있다. UML 은 산업계에서 널리 이해되고 사용되고 있으며 여기에서의 코드 생성 또한 이미 사용되고 있다. 잘 알려진 디자인 패턴들과 스마트 포인터와 같은 이디엄들은 Stereotype 이나 Tagged-Value 로 코드 생성에 직접적으로 이용될 수 있다. CASE 툴 벤더들은 이들을 profile 로 정리하여 제공을 해주되 기본 구현과 사용자 지정 방법 역시 제공하여야 한다.

AOP 는 코드 생성에 이용될 수 있는 또다른 분야이다. 이 논문에서는 특히 UML 에만 존재하는 AOP 를 AOP 언어 대신 쓸것을 제안한다. 이를 위해서는 AOP 를 위한 UML 확장이 필요해지며(물론 profile 로) 표준 작업 역시 중요하다.

기본적으로 소프트웨어 엔지니어링은 좀 더 자동화가 되어야 하며 이 논문에 나오는 확장 방법은 그 목적에 유용하게 쓰일 수 있을 것이다.

참고문헌

[1] Jernej Kovse, Theo H?rder - Generic XMI-Based UML Model Transformations Proceedings of the 8th International Conference on Object-Oriented Information Systems, OOIS, Montpellier, France, September 2-5, 2002

[2] Weerasak Witthawaskul, Ralph Johnson - An Object Oriented Model Transformer Framework based on Stereotypes 3rd Workshop in Software Model Engineering (WiSME 2004)

[3] Gy¨orgy Csert´an G´abor Huszerl Istv´an Majzik Zsigmond Pap Andr´as Pataricza D´aniel Varr´o - VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models Ph.D. thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2003.

[4] Jing Dong, Sheng Yang, Kang Zhang - A Model Transformation Approach for Design Pattern Evolutions 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06) pp. 80-92

[5] Omar Aldawud, Tzilla Elrad, Atef Bader - A UML Profile for Aspect Oriented Modeling OOPSLA 2001 workshop on Aspect Oriented Programming

[6] Mark Basch, Arturo Sanchez - Incorporating Aspects into the UML In Proceedings of Third International Workshop on Aspect-Oriented Modeling, March 2003