미래 소프트웨어 개발기술: Aspect-Oriented Programming과 Subject-Oriented Programming

LG전자 이준상

1. 서 론

현대의 소프트웨어 개발기술은 가히 폭발적이라 할 수 있을 만큼 매우 빠른 속도로 변화 발전하고 있다. 하루가 다르게 소프트웨어기반 기술에 대한 새로운 개념 및 용어가 생겨나고 있는 추세이고, 현재까지 가장 최신기술 중 하나로서 많은 관심을 모았던 객체지향 기술과 컴포넌트기반 기술 [1] 등은 더 이상 최신기술이라고 여겨질 수 없을 만큼 주위에서 흔히 접할 수 있게 되었다. 지금은 많은 개발자가 소프트웨어 설계시에 Unified Modeling Languages(UML) [2]를 사용하고 있으며, UML에 대한 깊은 이해가 없더라도 소프트웨어의 기본적이 구조나 행위를 기술하는 용도로 널리 이용되고 있다. 객체지향 기술은 처음 제안된 이후부터 지속적인 발전을 거듭해 왔으며, 공학의 최적화된 형태의 기술이라고 일컬어질 수 있는 컴포넌트기반 기술에까지 이르게 되었다. 여기서, 컴포넌트기반 기술은 객체지향 기술과 완전히 독립적으로 기원했다는 주장도 일리가 있지만 현재의 잘 다듬어진 컴포넌트기반 기술을 볼 때, 컴포넌트기반 기술이 상당부분 객체지향 기술로부터 발전해 왔다는 것은 쉽게 인지할 수 있는 부분이다. 현재까지도 가장 널리 사용되고 있는 객체지향 프로그래밍 언어중 하나인 자바 언어가 버전 1.1 이후부터는 컴포넌트 개념인 빈즈(Beans) 모델과 기본 통신 기법으로서 이벤트 위임 모델(delegation-based event model)이 표준으로 포함되었다는 사실로도 이를 확인할 수 있겠다.

본 논문에서는 적어도 향후 10년간 소프트웨어기반 기술에 영향을 줄 수 있는 소프트웨어 핵심 기술로써 관점지향 프로그래밍 (Aspect-Oriented Programming)과 주체지향 프로그래밍 (Subject-Oriented Programming)에 대해 소개할 것이다. 물론, 이 두 기술은 기존의 객체지향 개념과 완전히 다른 독립 기술로서의 역할보다는 철저한 보완 기술로서 미래의 프로그래밍 언어, 설계 언어, 개발 방법론 등에 모두 영향을 주며 상당 기간 발전해 나갈 것으로 기대되며, 최신 UML 버전 2.0에 포함된 Model-driven Architecture (MDA)나 효과적인 재사용성을 제공할 수 있는 product-line engineering 기술 등에 폭 넓게 영향을 줄 수 있는 유용한 보완 기술로써 주목 받고 있다.

소프트웨어공학은 여타 다른 공학의 연구 분야와는 많이 차별될 수 있는 특성을 가진 제품인 소프트웨어를 개발 대상으로 하고 있다. 여기서 Brooks가 소개한 소프트웨어가 지니는 필수적인 특성 (essential properties)을 살펴 볼 필요가 있다[3]. 첫째로, 소프트웨어는 이미 자체적으로 기계나 전자회로에 비해 보다 훨씬 복잡한 비선형적 상태를 다루고 있음에도 불구하고 동일 구성요소의 반복적인 결합 방식으로 제품 규모의 증가에 대해 적절히 관리 및 대처할 수 있다든지 수학이나 물리학처럼 근본적인 복잡도를 제거할 수 있는 추상 모델을 만들어 사용할 수 없다. 둘째로, 소프트웨어는 항상 같은 한 사람이 작성할 수 있는 규모나 쓰임새를 갖고 있는 것이 아니기 때문에 다수의 사람이 분담하여 다양한 형태의 인터페이스를 갖는 소프트웨어를 개발하게 된다. 때문에 규모가 큰 소프트웨어일수록 소프트웨어 모듈 사이의 일치성이 매우 중요한 이슈가 될 수 있다. 셋째로, 소프트웨어는 기계나 전자회로에 비해서 변경 가능성 및 용이성을 큰 장점으로 하고 있다. 그렇기 때문에 기존에 개발된 소프트웨어 제품은 쓰임새의 변화에 따라서 지속적으로 변경될 수 있는 필수적인 특성을 갖는다. 마지막으로, 소프트웨어는 완전하게 시각

화될 수 없다는 것이다. 현재까지, 소프트웨어 설계나 프로그램의 일부 관점을 시각화하는 연구가 많지만, 소프트웨어 전체에 대해 추상화된 시각화 연구가 존재하지 않음을 알 수 있다. 본질적으로, 소프트웨어는 기하공간 위에서 추상화된 형태로 구현 및 표현되지 않는다. 위에서 언급한 4가지 필수적 특성(essential properties)을 하나라도 제외하는 것은 소프트웨어로서의 쓰임새나 장점을 상당부분 포기해야 하기 때문에 필수적인 특성으로서 취급된다. 따라서 소프트웨어공학 연구는 위의 4가지 필수적 특성을 줄이려는 시도보다는 현장에서 문제시되는 비본질적 특성 (accidental properties)을 보다 효과적으로 극복하기 위한 연구 방향에 대부분의 노력을 기울이고 있는 것이다. 보다 추상화된 방식으로 소프트웨어 구조 및 행위를 기술하기 위한 진보된 개념을 제공하기 위한 프로그래밍 언어나 소프트웨어 설계 언어에 대한 연구 흐름이 대표적인 예라 할 수 있겠다.

현재까지의 발전해 온 프로그래밍 언어의 모습을 관찰하여 보면 소프트웨어 개발에 있어서 비본질적 특성으로서 소프트웨어의 근본적인 복잡도를 가중시키는 요소를 발견하고 이를 세련되고 수학적인 방법으로 추상화시키려는 연구의 흐름으로 해석할 수 있겠다. 소위, 높은 규격화 정도 (modularity)를 제공하는 소프트웨어를 개발함으로써 기능이해성 (understandability)과 유지보수성 (maintainability) 등을 높이 향상시키는데 주된 초점을 두는 연구 방향이라 하겠다. 본 논문에서는 다른 특수한 기능 (예를 들어 병렬성, 분산성, 고신뢰성, 실시간성, 최적화, 고성능 등) 에 목적을 둔 프로그래밍 언어보다는 이해성과 유지보수성을 강조한 보다 규격화된 소프트웨어를 개발할 수 있는 범용 프로그래밍 언어 관련 기술에 주된 초점을 맞추고 있다.

2. 프로그래밍 언어의 발전

컴퓨터는 프로그램을 통해 여러 가지 하드웨어 레지스터, 메모리 연산이나 외부 입출력 작업을 통해 복잡한 자기 상태를 유지/관리해 가면서 미리 계획된 순서의 일련 작업을 수행하게 된다. 가장 저수준의 프로그래밍 언어는 컴퓨터 하드웨어가 개발될 때 함께 제공되는 기계어가 되며, 이는 컴퓨터의 주 기억장치 상에서 그대로 로딩되어 수행될 수 있는 이진코드 형태로 표현되기 때문에 프로그래머는 수행 명령어 (instruction codes)에 대한 정보를 그대로 이해하고 사용하기 매우 어려운 특징이 있다.

이보다 조금 진보된 형태의 프로그래밍 언어로 어셈블리 언어를 생각해 볼 수 있다. 이는 프로그래머로 하여금 컴퓨터가 지원하는 수행 명령어를 인간이 이해하기 쉬운 형태의 명령어 구조, 연산자, 피연산자, 변수 등을 사용할 수 있도록 지원한다. 어셈블리 언어는 기계어와 같은 정보 수준의 프로그래밍 언어지만 인간이 읽고 쓰기 힘들다는 비본질적 특성을 제거 개선한 프로그래밍 기법이라 할 수 있겠다.

어셈블리 언어는 각 명령어에 대해선 그 의미를 이해하기 용이하나 프로그램의 전체 흐름이나 복잡한 수식 계산에 있어서는 프로그래머의 의도를 쉽게 옮기기에는 개념적 차이가 너무 크다. 이를 해결하기 위하여 제안된 고급 프로그래밍 언어 (high-level language)는 제어구조 추상화를 통해 인간의 사고와 매우 유사한 형태의 순차, 분기, 반복의 3가지 기본 제어 구조 패턴만을 이용하여 프로그램을 작성할 수 있도록 지원하며 복잡한 수식을 계산할 때 프로그래머는 인간에 친숙한 형태의 수식을 직접 기술하고, 이를 컴파일러가 실제 기계 상에서 수행 가능한 일련의 스택머신 기반의 수행 명령어들로 자동 번역해 줌으로써 소프트웨어 생산성 및 품질에 많은 도움을 주게 되었다. 결론적으로 고급 프로그래밍 언어는 인간이 불필요하게 수행해야 하는 비본질적 소프트웨어의 복잡성을 감소시키고 있는 것이다.

이후 객체지향 프로그래밍 언어가 제안되면서 기본적으로 3가지 중요한 추상화 개념을 제공하였다 [4]. 첫째, 복잡한 프로그램의 자료형과 이를 조작하는 인지적 수행 코드를 서로 묶어 정의하게 함으로써 이른바 자료 추상화 (data abstraction) 기법을 제공한다. 규모가 크고 복잡한 프로그램일수록 자료의 개수나 구조가 복잡한 형태를 띠게 된다. 실제로 자료의 값을 설정 및 변경하는 것은 고급 프로그래밍 언어에서 함수 혹은 프로시저 단위가 될 것이다. 고급 프로그래밍 언어로 작성된 프로그램 내에서의 자료들은 프로그래머가 의도한 기능성을 직접 표현하기 위한 방법이 아니라 대부분 간접적인 구현 수단으로써 사용되기 때문에 불필요하게 높은 복잡도를 야기시키는 문제를 갖고 있었다. 자료 추상화는 프로그래

머로 하여금 복잡한 자료들의 구체적 이해 없이 이들을 다룰 수 있는 표준화된 프로그램 프로시저만을 정의, 이해, 사용하게 함으로써 비본질적인 특성으로서의 복잡도를 크게 감소시키게 되었다. 둘째, 상위 추상화 (super abstraction) 기법은 일반화 관계로 구성된 클래스 계층구조를 통하여 보다 구체화된 클래스의 특성을 상황에 따라 일반화된 상태로 추상화하여 다룰 수 있는 기법이다. 예를 들어, MS 윈도즈 GUI 프로그램 작성을 위해 제공되는 MFC의 클래스 상속 계층 구조에서 대부분의 클래스들은 CWnd라는 기본 윈도즈 클래스를 상속받아 구체화되어 있다. 때문에, CWnd 클래스는 윈도즈의 구성요소로서 공통적으로 갖는 일반화된 속성과 행위를 모두 정의하고 있으며 만약, 한 프로그램을 구성하는 요소로서의 다양한 윈도즈 클래스들을 형태별로 구분할 필요가 없다면 상위 클래스인 CWnd 클래스가 정의하는 보다 추상화된 행위 수준 (예: 구성 요소의 글자색, 바탕색, 크기 등을 변경) 에서만 객체를 추상화하여 다룰 수 있을 것이다. 셋째, 객체지향 프로그래밍 언어는 은유 추상화 (meta abstraction)를 제공한다. 은유는 흔히 말하는 "A는 B다" 식의 정의 및 표현을 의미하며 여기서 B는 A의 한 인스턴스가 된다. 즉, B는 복잡한 현실을 반영하는 실 객체를 의미하며 이에 대한 특성 전부를 정의/ 표현하기보다는 B를 적절한 수준으로 모형화한 개념인 클래스인 A 수준에서 응용에 필요한 객체를 정의하여 사용할 수 있도록 하는 것이다. 상위 추상화 기법이 객체지향의 상속 관계로부터 발생한다면, 은유 추상화는 클래스-객체, 메타클래스-클래스와 같이 인스턴스 관계로부터 발생하게 된다. 객체지향 프로그래밍 언어가 대표적으로 제공하는 자료 추상화, 상위 추상화, 은유 추상화 기법들은 모두 비본질적으로 발생할 수 있는 소프트웨어의 복잡도 증가의 문제 요인에 대해 적절히 대처할 수 있게 한다.

객체지향 소프트웨어 개발 기법은 처음 제안될 당시의 특성으로부터 지속적으로 개선 및 보안되어 왔다. 객체지향 기법은 복잡한 현실 세계를 컴퓨터로 시뮬레이션 하기에 적합한 기술 형태로서 제안되었다. 즉, 프로그래머가 바라본 '복잡한 현실 세계를 정의하기 위한 신의 방식'을 조금 흉내내려는 것이라 할 수 있겠다. 시간이 지남에 따라, 객체지향 기법은 처음 제안될 당시 기대되었던 만큼의 이점을 제공하지 못한다는 문제점 지적과 함께 보안 연구가 많이 제안되었는데, 이들 중 대표적인 문제점 중의 하나가 바로 클래스 재사용성 및 유지보수 측면의 한계이다. 객체지향 프로그램을 구성하는 클래스들이 다른 프로그램 구성요소로 사용될 경우나, 지속적인 요구사항의 변경으로 유지보수가 빈번하게 일어나게 될 때 비본질적인 복잡성은 많은 문제를 야기시킬 수 있다. 클래스는 클래스 자체의 정의뿐만 아니라 함께 연동하는 다른 클래스들과 관련된 행위를 혼합된 형태(tangled)로 갖고 있기 때문에 클래스 단위의 재사용성만으로는 불필요한 행위를 갖는 클래스를 코드 단위에서 변경하거나 여러 클래스 사이에 필요한 협업 행위를 코드 수준에서 추출해야 하는 등의 복잡한 작업을 필요로 하게 된다. 이렇듯, 클래스의 상호 연동 관점의 행위 요소에 대해서도 규격화 및 추상화가 필요하다는 주장이 지속적으로 제기되기 시작했으며 이러한 연구들을 통칭하여 협업기반 개발 방법 (collaboration-based developments) [5,6] 이라 한다. 이중에서 개념적으로 가장 잘 정의되어 있으며 기존의 객체지향 기법과 잘 부합되는 기술로서 주체지향 개발 기법 (subject-oriented programming) [7] 이 존재한다.

지금까지 범용 프로그래밍 언어의 발전은 주로 소프트웨어의 구조적 특성을 향상시킴으로써 규격화된 특성을 보다 증가시키는 방향으로 전개되어 왔다고 할 수 있다. 즉, 소프트웨어의 개발 및 유지보수 시에 발생될 수 되는 비본질적 복잡성을 가능한 완화하기 위한 목적이었다. 하지만, 잘 규격화되고 이해도가 높은 프로그램 구조일수록 수행 속도에 있어서 어느 정도의 희생을 요구하게 되고, 반대로 수행 속도를 높이기 위한 프로그램 구조는 규격화 정도를 상당 부분 포기해야 하는 trade-off 관계를 갖게 된다. 때문에 수행 속도와 같은 비-기능적 특성 (non-functionality)을 지원하기 위해서는 개발 초기부터 프로그램의 규격화 정도를 포기하거나 프로그램의 기능적 특성(functionality)을 개발한 후에 추가적으로 조율(tuning) 하는 방식을 사용해 왔다. 이러한 방식은 소프트웨어의 높은 재사용성을 목적으로 하여 보다 규격화된 기능성의 구현과 특정 응용 프로그램의 비-기능적 목적을 모두 만족시키기에는 한계가 있었다. 이를 해결하기 위해 관점지향 프로그래밍 (aspect-oriented programming) [8,9] 기법은 이 두 가

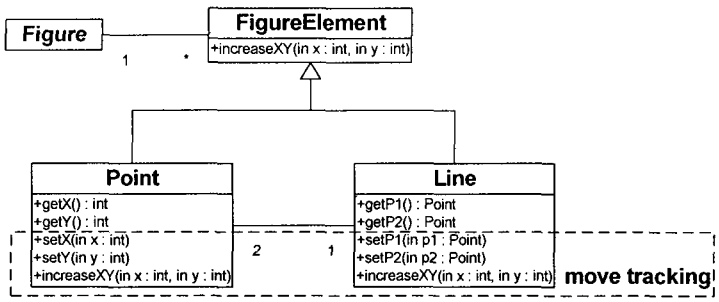

그림 1 간단한 그림 편집기 예제

지 주요 목적을 모두 만족시켜 주기 위한 방편으로 제안되었으며 현재 객체지향 기법과 더불어 지속적인 연구가 진행 중인 상태이다.

3. 관점지향 프로그래밍 (aspect-oriented programming)

3.1 개요

흔히 AOP라고 불리는 관점지향 프로그래밍 기법[8,9]은 1997년 제록스(Xerox)의 팔로알토(The Palo Alto Research Center) 연구소에 있는 Gregor Kiczales의 주도하에 연구가 진행되어 오고 있다. 여기서 관점(aspect)은 기존의 프로그램 모듈이 추상화 및 인캡슐레이션(encapsulation) 개념을 제공할 수 없는 이른바 cross-cutting concern을 의미한다. 관점지향 프로그래밍은 기존의 프로그래밍 언어가 제공하는 모듈 개념으로는 충분히 규격화 할 수 없었던 기능성 및 비-기능성에 대해 규격화 방법을 제공하는 것을 목적으로 하며, 프로그램 내에 존재하는 cross-cutting concerns 들이 관점 모듈의 형태로 설계/구현된다.

그림 1은 그림을 구성하는 요소로써 점과 선 객체와 그들 사이의 관계성을 클래스 다이어그램의 형태로 모델링하고 있다([9]에서 빌려온 예제). 여기서 점과 선으로 구성된 그림을 구성하고 위치를 변경 위한 기능성은 클래스 다이어그램 내에 잘 규격화되어 구현될 수 있지만 위치 이동에 대한 트랙킹 기능은 유사한 행위가 그림 구성 요소마다 중복되어 구현되어야 하는 문제가 있다. 모든 그림이 1개 이상의 점으로 구성되어 있기 때문에 그림의 움직임을 트랙킹 하는 모든 기능을 Point 클래스에 정의하는 방법도 있지만, 이 경우는 실제 그림의 형태나 성격에 따라 다른 움직임-트랙킹 기능을 수행하는 행위를 정의할 수 없다. 따라서 기존의 객체지향 프로그래밍 언어로는 유사한 행위를 여러 클래스에 중복하여 정의할 수 밖에 없으며, 관점지향 프로그래밍에서는 이러한 cross-cutting concern을 관점 (aspect)라고 한다. 이와 같은 cross-cutting concerns 은 여러 클래스에 유사한 기능이 중복되어 정의되어야 하거나 클래스 구조를 이용하여 규격성 (modularity)을 제공할 수 없는 기능성 및 비-기능성 단위를 의미한다.

3.2 결합점 모델(Join Point Model)

현재 대표적인 관점지향 프로그래밍 언어로는 AspectJ [9] 가 널리 사용되고 있다. 관점지향 프로그래밍은 기존 모듈에 대한 규격화 개념을 지원하는 동시에 다양한 cross-cutting concerns에 대해 규격화 기법을 지원해야 한다. 이에 따라 AspectJ는 가장 직관적인 방법으로써 결합점 모델 (join point model)에 기반한 관점-규격화 기법을 제공한다. 즉, cross-cutting concerns을 정의하는 독립된 구조의 관점 모듈이 존재하고 각 관점 모듈은 이미 정의되어 있는 기능성에 대한 모듈 구조상에서 어떠한 위치, 시간, 상황에 의존하여 최종적으로 원하는 행위를 갖는 프로그램으로 변형시킬 것인가에 대하여 기술하는 것이다.

표 1은 AspectJ가 제공하는 결합점 모델에 대해 설명하고 있다. 간단히 정리하면, 관점 모듈은 기본 결합점을 표현할 수 있는 기본 단절점 지정자 (primitive pointcut designator) 들로 구성된 사용자-정의 단절점을 기술하여 메쏘드가 호출될 때, 수행될

표 1 AspectJ의 결합점 모델

결합점 종류	설 명
method call, constructor call	메쏘드나 생성자가 호출되면 결합점은 호출 객체가 되나, 정적 메쏘드의 경우는 결합점이 없다.
method call reception, constructor reception	객체가 메쏘드 호출을 받는 경우 결합점은 호출되는 객체로 제어흐름이 넘어온 후, 해당 메쏘드 실행 전이 된다.
method execution, constructor execution	메쏘드나 생성자 수행이 요청된 경우.
field get	객체, 클래스, 인터페이스의 속성 값이 읽혔을 경우.
field set	객체, 클래스, 인터페이스의 속성 값이 변경됐을 경우.
exception handler execution	예외처리가 수행되는 경우.
class initialization	정적 클래스의 초기화가 수행 되는 경우.
object initialization	객체의 동적 초기화가 수행되는 경우.

때, 다른 메쏘드를 호출할 때, 객체의 속성을 읽거나 변경할 때, 예외상황 발생 시, 객체나 클래스가 정적 변수 형태로 초기화되는 시점에서 실매개 변수 형태로 전달되는 값에 기반 하여 프로그램의 행위를 변경시킨다. AspectJ에서는 aspect라는 지시어 (keyword)를 이용하여 관점 모듈을 정의하며 내부적으로는 단절점 구문 (pointcut statement) 을 이용하여 필요한 결합점의 집합을 정의하고, 각 결합점 집합에서 공통으로 적용될 수 있는 프로그램 변형 코드를 정의한다. 아래의 예는 그림 1에서 제시된 클래스 모델에서 그림의 움직임을 트랙킹하기 위해 필요한 사용자-정의 단절점을 정의한 것이다. 지정된 클래스의 특정 메쏘드가 호출 될 때 결합점 구성 요소로서의 피호출 객체 집합을 논리합 '||' 연산자를 이용하여 AspectJ 문법에 맞게 정의한 것이다.

```
pointcut moves():
receptions(void FigureElement.incrXY(int, int)) ||
receptions(void Line.setP1(Point)) ||
receptions(void Line.setP2(Point)) ||
recpetions(void Point.setX(int)) ||
receptions(void Point.setY(int));
```

또한, *advice* 구문 (5종류: before, after, around, after returning, after throwing)을 이용하여 어떤 단절점이 정의하는 구성 결합점 집합에 공통적으로 추가되어야 하는 코드의 적용 시점을 조절할 수 있다. 예를 들어 "*after(): moves() { flag = true }*"의 문장은 *moves()*라는 사용자-정의 단절점이 지정하는 결합점 집합의 각 시점에서 *flag* 변수를 *true* 값으로 설정하는 행위를 추가함을 의미한다. 관점 모듈 내에 "*before(): moves() { flag = false }*"의 정의도 있다면 *moves()*에서 지정하는 결합점으로서의 메쏘드 집합에 대해 수행 전/후의 상태를 기록함으로써 움직임을 트랙킹 하기 코드를 중복해서 적용하지 않을 수 있다. *flag* 변수는 피호출 객체가 원래 갖고 있는 변수가 사용될 수도 있지만, 여기서는 해당 관점 모듈에 응용상의 목적으로 추가 정의된 변수가 사용되는 것이 보다 바람직하다. *Line* 객체의 위치를 변경할 수 있는 메쏘드의 호출이 두개의 구성 *Point* 클래스 위치를 변경할 수 있는 메쏘드 호출을 유발하는 구조로 정의되어 있기 때문에 관련 메쏘드 호출시에 무조건적으로 트랙킹 코드를 추가하면 중복된 트랙킹 작업이 수행 될 수 있는 오류를 범할 수 있다.

3.3 관점지향 언어로서의 AspectJ

지금까지 설명한 내용은 모두 AspectJ에서 제공하는 동적인 cross-cutting에 해당하는 부분이었다. 즉, 프로그램이 수행되는 중에 잘 정의된 결합점 상

에서 동적으로 추가되어야 하는 코드를 관점 모듈에 기술하는 것이다. 이외에도 정적인 cross-cutting 을 지원하는데, 이는 이미 정의된 클래스에 새로운 메쏘드나 속성을 추가하거나 변경하는 것을 가능하게 한다. AspectJ는 관점 모듈 내에서 정적/동적 cross-cutting 사항을 기술하도록 지원함으로써 관점 지향 프로그래밍의 기본 구조를 형성할 수 있도록 한다.

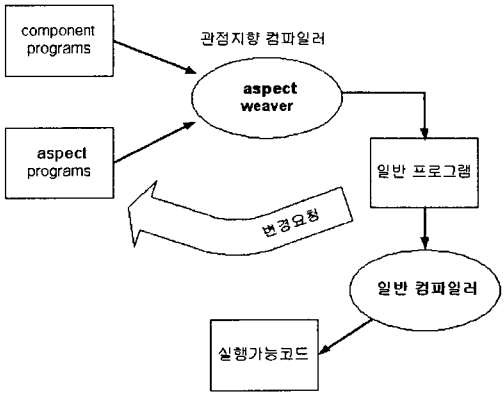

그림 2 관점지향 프로그래밍의 기본 구조 및 절차

AspectJ의 관점지향 프로그램은 제시된 결합점 모델을 통해 cross-cutting concern에 대한 구현 사항이 실제 실행 가능한 프로그램 형태로 변환된다. 그림 2는 관점지향 프로그래밍 언어가 처리 방법이 기본적으로 취하고 있는 구조 및 절차를 묘사하고 있다. 모듈은 기존의 기능성을 규격화하는 컴포넌트 모듈과 cross-cutting concern을 규격화하는 관점 모듈로 분리되어 구성된다. 이는 관점지향 언어 컴파일러에 의해 두 가지 행위가 결합된 형태의 일반 프로그램으로 변역 된다(AspectJ의 경우 일반 자바 프로그램의 형태로 변환해 준다). 프로그램의 작성, 이해, 변경 작업은 모두 관점지향 컴파일러를 거치기 이전 단계에서 수행되므로 기존 컴포넌트 모듈의 규격화와 cross-cutting concern을 정의하는 관점 모듈의 규격화를 모두 지원해 줄 수 있는 것이다.

4. 주체지향 프로그래밍 (subject-oriented programming)

객체지향 프로그래밍은 일반적으로 캡슐화(encapsulation), 다형성(polymorphism), 상속(inheritance) 이렇게 세 가지 대표적 특성으로 표현된다. 하지만, 지금까지의 객체지향 프로그래밍은 단지 객체로서의 식별성(identity)만을 강조해 왔기 때문에 다양한 응용 프로그램 측면에서 객체를 바라보는 주체에 따라 달라질 수 있는 주체적 관점(subjective views)에 대한 중요성에 대해서는 간과해 왔었다. 주체지향 프로그래밍 [7]은 하나의 객체가 고정된 특정 응용 프로그램 내에서만 의미 있는 것이 아니라 재사용이나 유지보수 과정을 거치면서 다양한 응용 프로그램 상에서 사용될 수 있고, 이에 따라 바라보는 주체에 대해 같은 객체가 다른 식별성을 갖게 될 수 있다는 개념이다.

쉬운 예를 들어보면, 객체지향 관점에서 나무(Tree)라는 객체를 생각해보자. 나무는 실세계에서 다양하게 다른 객체들과 관계를 맺고 상호 작용 할 수 있다. 그러나 나무 객체와 관계를 맺는 주체적 대상에 따라 다른 특성을 가진 존재로서 식별될 수 있다. 나무의 과실을 먹거나 위에 둥지를 틀수 있는 대상으로서의 객체로 생각하는 새라는 주체가 존재한다. 이와 동시에 나무를 베어 팔아서 생계를 유지하는 나무꾼 주체의 경우는 나무의 품질이나 베고 가공하는데 필요한 시간 등을 중요 특성으로 바라보고 있을 것이다. 또한 나무를 개인의 재산으로써 책정하고 적절한 세금을 부과하기 위한 대상으로 바라보는 주체로서 과세액-사정자(tax accessor)가 존재한다. 이렇듯, 같은 대상 객체인 나무에 대해 서로의 관점에 따라 다른 특성과 행위를 갖는 객체로서 취급할 수 있도록 하는 것이 주체지향 프로그래밍의 기본 개념이다.

따라서 주체지향 프로그래밍에서는 객체의 본연적 특성(intrinsic properties)과 비본질적 특성(extrinsic properties)에 대한 분리를 강조한다. 예를 들어서, 나무라는 객체를 정의할 때 변하지 않고 필수적으로 지니고 있는 본질적인 특성(예: 나무종류, 길이, 무게, 부피, 나이)을 객체의 기본 특성으로 정의하고, 바라보는 주체의 관점 따라 달라 질 수 있는 응용 특성들은 모두 비본질적 특성으로 분리 정의하여 사용할 수 있도록 지원 한다. 응용 프로그램에서 필요한 주체적 특성들은 상황에 따라 본질적 특성만을 갖는 객체와 결합하여 사용할 수 있는 것이다.

주체지향 프로그래밍에서 주체는 개개의 주체 집합을 의미하기 보다는 특정 응용 프로그램에 의해 관

심있게 보여지는 세계에 대한 이해, 형태, 모습 등을 잘 반영하기 위한 각 객체가 소유하는 부가적인 상태나 행위에 대한 집합이다. 주체는 클래스가 아니며 단지 어떤 응용영역 상으로 새로운 클래스들에 도입되는 정의에 대한 역할을 담당 하는 것이다. 이에 따라 주체는 특정 응용 프로그램의 관점에서 필요한 다양한 클래스의 추가적인 상태나 행위 일부분을 일반적 방법으로 기술하게 된다.

다형성 측면에서도 주체지향 프로그래밍을 설명할 수 있다. 객체지향에서는 객체의 실제 클래스 종류에 따라 같은 형태의 메시지 수신이 다른 행위를 유발시키는 것을 의미한다면, 주체지향 프로그래밍에서는 객체가 실제 같은 클래스 종류로 정의되어 있고 같은 형태의 메시지를 받는다고 해도 실제로 어떤 주체 관점(subjective views) 상에서 발생하느냐에 따라 전혀 다른 행위를 유발할 수 있게 된다.

결론적으로, 주체지향 프로그래밍을 통하여 개발된 응용 프로그램은 객체의 본질적 특성과 비본질적 특성들에 대하여 캡슐화(encapsulation) 개념을 지원함으로써 상황에 따라 비본질적 특성들을 모듈 수준에서 적절히 추가 및 삭제하여 사용할 수 있는 것이다. 이러한 특성은, 개발된 객체가 특정 응용프로그램에 종속적인 특성을 많이 완화시킴으로써 객체 및 주체에 대한 속성 및 행위에 대한 재사용성을 크게 증가시킬 수 있는 기반 환경을 제공한다. 이렇듯, 주체지향 프로그래밍은 객체지향 프로그래밍 내에서 클래스의 기본 행위와 다른 클래스들과의 협업 행위에 대한 분리, 즉 캡슐화 기법을 제공하자는 관점에서 협업기반 설계기법(collaboration-based design)[5] 을 추구하는 다른 많은 연구들과 성격을 함께 하고 있다. 현재 주체지향 프로그래밍은 미국 IBM 연구소의 선도 하에 다양한 측면으로 응용 발전하고 있으며 현재에는 관점지향 프로그래밍 개념과 연계하여 한층 진화된 형태인 다차원 관심사 분리(multi-dimensional separation of concerns)의 개념으로 일반화 되었다[10,11].

5. 결 론

현대 소프트웨어 개발 기술은 상당 부분 객체지향 기술의 모습을 유지하고 있다. 최근 가장 각광을 받고 있는 컴포넌트기반 소프트웨어 개발 기술도 결국은 '다차원 관심사 분리'라는 공통된 개념 하에 발전하고 있다고 볼 수 있겠다. 또 하나의 핵심 개념인 소프트웨어 아키텍쳐 [12]는 컴포넌트기반 소프트웨어 개발 기술을 현실화하기 위해 필요한 가장 중요한 개념이며 근본적으로 다중 관점(multiple views)에 대한 관심사의 분리 개념을 지원해야 한다. 이는 기본적으로는 총체적 구조(gross structure)와 전체적 흐름성 제어(global control flow) 측면에 대한 관심사의 분리 개념을 제공함으로써 컴포넌트를 독립적으로 개발하고 컴포넌트 상호 의존성을 조립 시점으로 연장시킬 수 있게 된다.

UML의 경우, 유즈케이스(usecase)를 구현하는 분석 단계에서 객체의 역할에 기반한 객체상호 간의 상호협력 행위관계를 유즈케이스 단위로 추출/통합함으로써 주체지향 프로그래밍이나 협업기반 설계의 기본 개념을 상당부분 지원하고 있으나, UML의 설계 단계 이후부터는 유즈케이스 별로 객체의 관계 및 협업 행위들을 캡슐화 할 수 있는 실용화 할 수 있는 기법이 아직 제공되지 않기 때문에 소프트웨어 생명주기 전반에 걸쳐 주체지향 프로그래밍 기법을 적용하기에는 너무 이른 상황이다. 하지만 점차적으로 UML을 이용한 소프트웨어 설계 단계에서 주체지향 및 관점 지향 프로그래밍 개념을 도입하는 연구가 늘어나는 상황이며 [13,14] 또한 구현 단계에서의 지원 연구도 활발히 진행되고 있는 추세이다[9,11,15,16]. 머지않아 UML을 이용하여 소프트웨어 생명주기 전반에 대해 다차원 관심사의 분리 개념을 제공할 수 있는 매우 진보된 형태의 미래 소프트웨어 개발 기술이 자리 잡게 될 것으로 전망된다.

참고문헌

[1] A.W. Brown and K.C. Wallnau, "Engineering of component-based systems," in Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, pp. 7-15, IEEE Computer Society Press, 1996.

[2] OMG Unified Modeling Language Specification ver. 1.4, Rational Software Corporation, 2001.

[3] F. P. Brooks, Jr, "No Silver Bullet: Essence and Accidents of Software Engineering,"

IEEE Computer, vol. 20, no. 4, pp. 10-19, April 1987.

[4] P. Wegner, OOPS Messenger, A Quarterly Publication of the SIGPLAN, vol. 1, no. 1, August 1990.

[5] Y. Smaragdakis. Implementing large-scale object-oriented components, Ph.D. thesis, University of Texas at Austin, 1999.

[6] M. VanHilst and D. Notkin. "Using role components to implement collaboration-based designs," in proceedings of ACM Conference on Object-Oriented Systems, Languages, and Applications, 359-369, 1996.

[7] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects), ACM SIGPLAN Notices, vol. 28, no. 10, 411-428, 1993.

[8] G. Kiczales and et al. Aspect-oriented programming, in proceedings of European Conference for Object-Oriented Programming, LNCS 1241, 220-243, 1997.

[9] G. Kiczales and et al, "An Overview of AspectJ (0.8)," in proceedings of European Conference for Object-Oriented Programming, 2001.

[10] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. "N Degrees of Separation: Multi-Dimensional Separation of Concerns." Proceedings of the International Conference on Software Engineering (ICSE'99), May, 1999.

[11] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach." in Proceedings of the Symposium on Software Architectures and Component Technology, 2000.

[12] D. Garlan and D. Perry. Introduction to the special issue on software architecture. IEEE Transactions on Software Engineering, vol. 21, no. 4, April 1995.

[13] J.S. Lee and D.H Bae, "An Approach to Specifying Concurrent, Distributed, and Autonomous Object Behaviors Using a High-Level Meta-Object Protocol," IEICE Transactions on Communications, IEICE/IEEE Joint Special Issue on Autonomous Decentralized Systems, vol. 83-B, no. 5, May 2000, pp. 999-1012.

[14] Siobhan Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. in Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, 12-19, May 2001.

[15] J.S. Lee and D.H Bae, "An Enhanced Role Model for Alleviating the Role-binding Anomaly," Software-Practice and Experience, John & Wiley and Sons, vol. 32, issue 14, pp. 1317-1344, Nov. 2002.

[16] J.S. Lee and D.H Bae, "An Aspect-Oriented Framework for Developing Component-based Software with the Collaboration-based Architectural Style," Information and Software Technology, Elsevier, to appear in a 2003 issue.

이 준 상

1997 동국대학교 컴퓨터공학과(공학사)
1999 한국과학기술원 전산과학 전공(공학석사)
2003 한국과학기술원 전산학과 전공(공학박사)
2003 LG전자 디지털 멀티미디어 연구소, Diznet 그룹 선임 연구원
관심분야 : 소프트웨어공학, 객체지향/컴포넌트기반 기술, computational reflection.
E-mail : windyscar@lge.com