20-1-R03

새로운 프로그래밍 패러다임 - 관점지향 프로그래밍
(New Programming Paradigm - Aspect Oriented Programming)

금 영 욱
성결대학교 컴퓨터공학부
ywkeum@sungkyul.edu

요 약

기존의 프로그래밍 방법론으로 여러 모듈에 속성상 걸치게 되는 횡단 관심사(cross-cutting)를 모듈화할 수 없다. 횡단 관심사의 비모듈화로 인하여 코드 혼합과 코드 산재의 문제가 제기되어 소프트웨어 개발에 어려움이 가중되었다. 횡단 관심사의 모듈화를 해결하기 위한 많은 연구가 진행되었으며 여러 해결책이 제시되었다. 그 중에 가장 주목을 받는 것이 관점지향 프로그래밍이며 프로그래밍 발전사에서 객체지향의 다음 단계를 이어가는 새로운 방법론으로 부각되었다. 관점지향 프로그래밍은 새로운 모듈화 단위인 애스펙트(aspect)를 도입하여 독립적으로 횡단 관심사를 모듈화하며 나중에 직조(weaving) 과정을 통하여 다른 관심사들과 횡단 관심사를 통합하여 완성된 소프트웨어를 제작한다. 이 논문에서 관점지향에서 도입한 새로운 개념과 구체적으로 이를 구현한 언어인 AspectJ에 관해 논의한다. 또한 동적인 직조를 함으로 프로그램 수행 중에 애스펙트가 적용되게 하는 JAsCo에 관하여 소개한다.

1. 서 론

소프트웨어 시스템을 제작하는 최선의 방법은 관심사를 분리(separation of concerns)하는 것이다. 관심사는 소프트웨어 시스템의 목적을 달성하기 위해 처리해야 하는 구체적인 요구사항이나 고려사항을 의미한다. 관심사를 분리하는 최선의 방법은 1972년 David Parnas가 제시한 것처럼 모듈을 생성하여 서로 다른 모듈간에 독립성을 유지하는 모듈화이다[17]. 기존의 절차적 프로그래밍은 함수 추상화를 도입하였고 객체지향 프로그래밍은 객체 추상화를 도입하여 모듈화에 이바지하였다.

기존의 객체지향 프로그래밍 등은 각 모듈에서 수행해야 하는 기본적이고 대표적인 관심사인 핵심 관심사를 모듈화하는데 매우 효율적이지만, 여러 모듈에 걸치는 속성을 지닌 횡단 관심사(cross-cutting concern)를 모듈화할 수 없다. 아파치 톰캣을 조사해 본 결과 XML 파싱이나 URL 패턴 매칭과 같은 관심사는 한 개 내지 두 개의 모듈로 구현이 되지만, 로깅 또는 세션 종료 등과 같은 관심사는 그 구현이 많은 모듈에 걸쳐있게 된다. 이것은 설계자의 잘못이 아니고 객체지향적인 방법이 속성상 여러 모듈에 걸치는 횡단 관심사를 모듈화할 수 없

는 기술적인 한계를 가지고 있기 때문이다.

모듈화할 수 없는 이런 10%의 코드가 전체 유지 보수의 90%의 어려움을 주고 있다는 보고가 있을 정도로 문제의 심각성이 제기되고 있다[16]. 또한 소프트웨어 시스템이 복잡도가 점점 더 해 가기 때문에 이런 문제가 해결되지 않을 경우 소프트웨어 업계는 심각한 위기의식을 느끼고 있다[6].

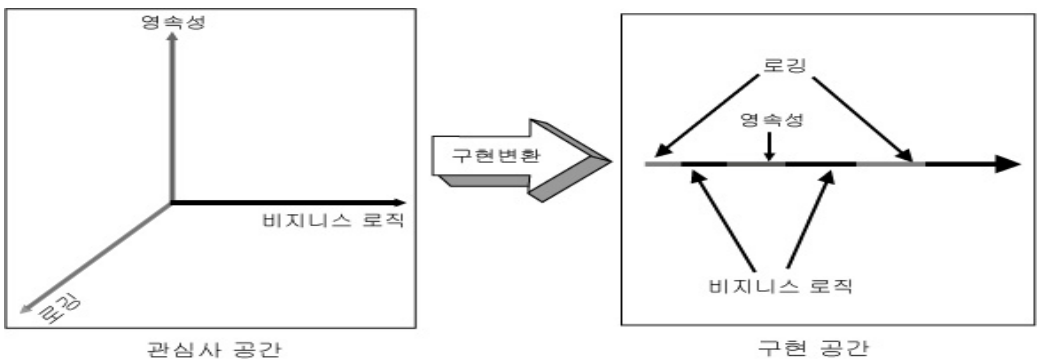
이런 요구사항에 부응하여 여러 해결책이 제시되었는데 그 중에서 관점지향 기술[2-4]은 가장 주목을 받는 기술이며 객체지향을 이어가는 다음 단계의 방법론으로 부각되었다[1]. 제록스의 팔로 알토 연구소의 연구원이었던 Gregor Kiczales가 1996년도에 Aspect-Oriented Programming(AOP)라는 용어를 만들었으며 미국의 국방고등연구소(Defence Advanced Research Projects Agency)의 후원을 받아 노스이스턴 대학의 Cristina Ropes 등과 함께 관점지향 프로그래밍을 구현한 대표적인 언어인 AspectJ를 제작하여 관점지향 프로그래밍의 확산에 기여하였다. 관점지향 프로그래밍은 횡단관심사를 독립적으로 모듈화하여 나중에 직조(합성) 과정을 통하여 한 개의 시스템으로 조립하는 새로운 방법론이다. 이 논문은 관점지향 프로그래밍과 이를 지원하는 정적인 언어 AspectJ와 동적인 프레임워크인 JAsCo를 논의한다.

이 논문의 구성은 다음과 같다. 2장에서 기존의 프로그래밍 방법론의 문제점을 기술하고 3장에서 관점지향 프로그래밍 방법론을 논한다. 4장에서 이를 구현한 대표적인 언어인 AspectJ를 소개하고 5장에서 동적으로 구현한 JAsCo를 소개한다. 6장에서 적용분야를 기술하고 7장에서 결론을 맺는다.

2. 기존의 방법론의 문제점

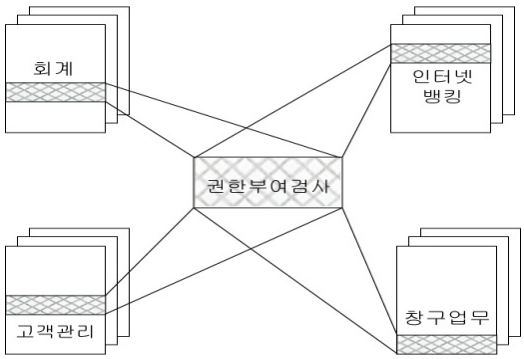
기존의 방법론을 사용하면 핵심 관심사의 모듈화에는 효율적이나 횡단 관심사를 모듈화할 수 없다. 설계 시점에는 횡단 관심사도 독립적인 관심사로 분리할 수 있으나 구현 방법의 제약성으로 인해 구현 시점에는 한 개의 모듈에 횡단 관심사가 혼합하여 들어가게 된다. (그림 1)에서 핵심 관심사인 비즈니스 로직과 횡단 관심사인 영속성과 로깅이 설계 시점에는 독립적인 관심사로 분리가 되지만 구현 시점에는 한 개의 모듈에 혼합하여 들어가는 것을 보여준다.

기존의 방법론의 제약으로 인하여 2가지 현상이 발생한다. (그림 1)에 나타난 것처럼 한 개의 모듈에 여러 개의 서로 다른 관심사가 구현되는 현상을 코드 혼합(code tangling)



(그림 1) 관심사의 변환

이라고 한다. 또한 여러 개의 모듈에 동일한 관심사가 구현되어 들어가는 코드 산재(code scattering) 현상이 생긴다. (그림 2)가 코드 산재 현상을 보여주고 있는데 권한부여(authorization) 검사 코드가 4개의 다른 모듈에 전부 들어가 있는 것을 보여주고 있다.



(그림 2) 코드 산재

3. 관점지향 프로그래밍

관점지향 기술은 횡단 관심사를 모듈화하기 위해 객체지향 기술과 절차적 프로그래밍 기초 위에 새로운 개념과 구조물을 확장한 것이다. 관점지향 기술을 사용하여도 핵심 관심사는 기존의 방법론으로 구현한다. 단지 횡단 관심사를 관점지향 기술로 구현할 뿐이다.

애스펙트(aspect)는 두 가지 의미를 가지고 있다. 요구 분석 시점의 애스펙트는 횡단 관심사(cross-cutting concern)의 의미가 있고 구조물(construct)로 사용될 때는 클래스(class)와 같이 횡단 관심사를 구현하는 한 개의 모듈의 의미를 가지고 있다.

관점지향 프로그래밍을 사용하여 프로그램을 개발하는 간단한 과정은 다음과 같다.

3.1 관심사 분해(decomposition)

요구사항을 분해하여 핵심 관심사와 횡단 관심사를 찾아내고 이들을 분리해 낸다. 예를 들어, 비즈니스 로직, 영속성, 로깅 등의 관심사가 있을 때 이 중에서 비즈니스 로직만이 핵심 관심사이고 나머지는 시스템 전체에서 사용되는 관심사로 다른 모듈들에서도 필요하기 때문에 횡단 관심사로 분류한다.

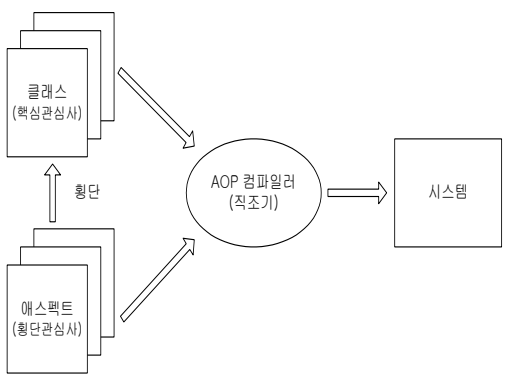
3.2 관심사 구현(implementation)

이 단계에서 각 관심사(핵심 또는 횡단 관심사 포함)를 독립적으로 구현한다. 개발자들은 비즈니스 로직, 영속성, 로깅 등을 독립적으로 개발한다. 핵심관심사인 비즈니스 로직은 클래스로 모듈화하고 횡단 관심사인 영속성, 로깅 등은 애스펙트로 모듈화한다.

3.3 관심사 합성(recomposition)

횡단 관심사(영속성, 로깅 등)가 독립적으로 구현되어 있기 때문에 이를 합성하는 과정이 필요하다. 합성을 하는 과정을 직조(weaving)라 하며 이 직조를 담당하는 소프트웨어를 직조기(weaver)라고 한다.

(그림 3)이 AspectJ 컴파일러에 의한 관심사 합성을 보여주고 있다.



(그림 3) AspectJ 컴파일러에 의한 애스펙트 직조

직조는 소스 코드 수준, 바이트 코드 수준 또는 실행 시점에 수행할 수 있다. AspectJ는 바이트 코드에 직조하고, JAsCo는 실행 시점에 직조한다.

4. AspectJ

AspectJ[2-4, 8-9]는 Java 언어에 관점 지향적인 개념을 추가하여 확장한 대표적인 범용의 언어이다. 모든 유효한 Java 프로그램은 유효한 AspectJ 프로그램이 된다. AspectJ 컴파일러는 Java 바이트 코드 명세를 따르므로 Java 가상기계는 AspectJ가 생성하는 클래스 파일을 실행한다.

AspectJ에서 횡단 모듈을 구현하는데 동적 횡단과 정적 횡단이 있다. 동적 횡단은 핵심 모듈이 실행될 때 새로운 횡단 모듈을 실행하거나 또는 기존의 코드를 새로운 코드로 치환하여 실행함으로 시스템에 횡단 모듈을 구현한다. 4.2절에 동적 횡단의 예를 보인다. 정적 횡단은 클래스, 인터페이스 또는 애스펙트와 같은 정적 구조에 변경을 직조하는 것이다. 정적 횡단의 예로, 기존 클래스나 인터페이스에 새로운 멤버 필드나 메서드의 추가, 또는 컴파일 시점에 어떤 특정한 코드에 대한 경고와 오류 등이 있다.

4.1. AspectJ의 횡단 요소

AspectJ에서 Java 언어를 확장하여 만든 구조물을 횡단요소라고 한다. 이 횡단 요소들이 횡단 관심사를 구현하는데 기초 요소로 사용된다. 횡단 요소를 차례대로 설명한다.

4.1.1. 결합점(join point)

결합점은 실제로는 새로운 구조물은 아니고, 프로그램 실행 과정에서 구별 가능한 지점으로 직조에 의해 횡단 코드가 들어오는 지점이다. 가능한 모든 결합점이 AspectJ에서 허용되는 것이 아니고, 허용되는 결합점은 메서드 결합점, 생성자 결합점, 변수 참조 결합점, 예외처리 실행 결합점, 클래스 초기화 결합점, 객체 초기화 결합점, 객체 초기화 이전 결합점, 그리고 충고 실행 결합점이다. 예를 들어, 지역 변수나 제어문 등은 결합점으로 사용할 수 없다.

4.1.2. 교차점(pointcut)

교차점은 결합점들을 지정하고 결합점의 환경정보(context)를 수집하는 구조물이다. 예를 들어, 교차점은 메서드 호출을 지정하고 메서드가 호출될 때의 대상 객체와 매개변수 등과 같은 환경정보를 얻을 수 있다.

AspectJ에서 제공하는 교차점의 종류로 결합점 유형에 따른 교차점, 제어 흐름 교차점, 어휘 구조에 근거한 교차점, 실행 객체 교차점, 매개변수 교차점, 조건 검사 교차점 등이 있다. (표 1)에서 결합점 유형에 따른 교차점을 보여준다.

(표 1) 결합점 유형에 따른 교차점

결합점 유형	교차점
메서드 실행	execution
메서드 호출	call
생성자 실행	execution
생성자 호출	call
클래스 초기화	staticinitialization
필드 읽기 참조	get
필드 쓰기 참조	set
예외처리실행	handler
객체 초기화	initialization
객체 초기화 이전	preinitialization
충고 실행	adviceexecution

4.1.3. 충고(advice)

충고는 교차점에서 지정한 결합점에서 실행되는 코드이다. 충고의 내용은 메서드의 내용과 유사하여 결합점에 도달할 때 실행되는 로직을 포함하고 있다. 충고는 결합점의 실행 이전에(before), 결합점의 실행 이후에(after), 또는 결합점을 대체하여(around) 실행될 수 있다.

이후 충고는 3가지 유형이 있다. 결합점의 정상적 종료 이후의 충고, 예외가 발생된 비정상적 종료 후의 충고, 그리고 예외 발생 여부와 관계없이 적용되는 충고가 있다. 대체 충고에서 결합점에 원래 있던 기존 코드의 실행을 변경할 수 있는데, 즉 기존 코드를 다른 코드로 대체하거나, 또는 아예 기존 코드가 실행되지 않게 할 수도 있다.

교차점과 충고는 동적 횡단 규칙을 형성한다. 교차점은 결합점을 선택하고 충고는 결합점에서 실행되는 행위를 제공하여 동적 횡단을 완성한다.

4.1.4. 타입간의 선언(inter-type declaration)

타입간의 선언은 소프트웨어 시스템의 클래스, 인터페이스, 그리고 애스펙트에 변화를 도입하는 정적 횡단 명령이다. 클래스의 상속구조를 변경할 수도 있고 클래스에 새로운 메서드나 필드를 추가할 수 있다.

4.1.5. 컴파일 시점 선언

컴파일 시점 선언은 정적 횡단 명령으로, 컴파일 시점에 특정한 코드의 유형을 만나면 컴파일러가 경고나 오류 메시지를 발생할 수 있게 한다. 예를 들어, 시스템의 어떤 부분에 Persistence 클래스에 있는 save() 메서드를 호출하면 컴파일러가 경고 또는 오류 메시지를 출력할 수 있다.

4.1.6. 애스펙트(aspect)

클래스가 Java에서 모듈화하는 단위인 것처럼 애스펙트는 AspectJ에서 모듈화하는 단위이다. 애스펙트는 동적 및 정적 횡단과 관련된 모든 구조물을 포함한다. 즉 앞에서 설명한 교차점, 충고, 타입간의 선언, 컴파일 시점 선언 등이 모두 애스펙트에 정의된다.

4.1.7. 고급 기능

복잡한 시스템을 위하여 다음과 같은 고급 기능이 제공된다.

- 결합점에 대한 정적 또는 동적 정보를 얻기 위한 반사(reflection) 기능
- 여러 개의 애스펙트가 한 개의 결합점에 적용될 때 애스펙트 우선순위 기능
- 애스펙트 인스턴스를 가상기계별, 객체별, 제어 흐름별로 연관시키는 기능
- 예외 처리 횡단 관심사를 모듈화하는데 사용하는 예외 연화 기능(softening)
- 애스펙트에 특권을 부여하는 기능

4.2. AspectJ의 구현 예

이 절에서 간단한 예를 구현한다. (예제 1)에서 메시지를 출력하는 2개의 메서드를 가진 클래스를 생성하자. Message 클래스는 메서드 두 개(수신자를 지정하지 않고 보내는 메서드와 특정인에게 메시지를 보내는 메서드)를 가지고 있다.

(예제 1) Message.java

```
public class Message {
   public static void deliver(String msg) {
      System.out.println(msg);
    }
   public static void deliver(String person,
       String msg) {
      System.out.println(person + ",," + msg);
    }
}
```

다음에 Message 클래스의 기능을 이용하는 Test 클래스를 (예제 2)에 작성한다.

(예제 2) Test.java

```
public class Test {
  public static void main(String[] args) {
    Message.deliver
    (“AspectJ를 배우고 싶으세요?”);
    Message.deliver
    (“길동”, “재미있어요?”);
    }
}
```

Message와 Test 클래스를 함께 컴파일하고 (ajc 명령 사용) Test 프로그램을 실행하면 다음과 같은 출력을 얻는다.

(예제 2)의 실행 결과

```
>ajc Message.java Test.java
>java Test
AspectJ를 배우고 싶으세요?
길동, 재미있어요?
```

Message 클래스에 있는 코드를 전혀 변경하지 않고, 단지 아래의 애스펙트를 추가함으로 클래스의 기능을 향상시킬 수 있다. (예제 3)에 있는 것처럼 MannersAspcet를 구현하자. 이 애스펙트는 “안녕하세요!”로 먼저 인사하고 그 후에 모든 메시지를 전달한다.

(예제 3) MannersAspect.java

```
public aspect MannersAspect { //①
   pointcut deliverMessage()     //②
     :call(* Message.deliver(..));
   before() : daliverMessage() { //③
        System.out.print(“안녕하세요!”);
   }
}
```

이 애스펙트의 코드를 살펴보자.

① 애스펙트의 선언은 클래스 선언과 유사하다.

② 애스펙트에 정의된 교차점 deliverMessage()는 Message에 있는 메서드 deliver()에 대한 모든 호출을 포착한다. 교차점에 있는 와일드카드 “*”는 반환형에 관계없이, 또한 괄호 안에 있는 와일드카드 “..”는 매개변수의 종류와 개수에 관계없이, 모든 deliver() 메서드를 포착한다.

③ 충고를 정의한다. before()를 사용하면 Message.deliver()가 실행되기 전에 충고가 실행된다. 충고에서 메시지 “안녕하세요!”를 줄을 바꾸지 않고 출력한다.

이제 클래스 파일을 애스펙트와 함께 컴파일하자. AspectJ 컴파일러인 ajc가 (예제 3)에 있는 애스펙트를 직조(weaving)한 클래스 파일을 생성하려면 모든 입력 파일을 함께 ajc로 컴파일해야 한다. 컴파일하고 실행하면 다음과 같은 출력을 보게 된다.

MannersAspect를 포함한 실행결과

```
>ajc Message.java MannersAspect.java
     Test.java
>java Test
안녕하세요! AspectJ를 배우고 싶으세요?
안녕하세요! 길동, 재미있어요?
```

이번에는 한국말 특성을 살려 인사하는 애스펙트를 만든다. (예제 4)에서, 사람 이름 뒤에 “님”을 부쳐 인사하고 메시지를 전달한다.

(예제 4) KoreanSalutationAspect.java

```
public aspect KoreanSalutationAspect {
  pointcut sayToPerson(String person) //①
    :call(* Message.deliver(String, String))
       args(person, String);
  void around(String person);
  sayToPerson(person) {       //②
    proceed(person + “님”); //③
  }
}
```

이 애스펙트의 코드를 살펴보자.

① 교차점 sayToPerson은 두 개의 매개변수를 취하는 Messge.deliver()를 호출하는 결합점을 포착한다. 사람 이름 뒤에 “님”을 붙이려면 매개변수 person을 사용해야 한다. args()의 첫 번째 매개변수인 person은 메서드 deliver()의 첫 번째 매개변수가 person으로 들어오는 것을 지정한다. sayToPerson()의 매개변수 person은 deliver()의 첫 번째 매개변수와 일치하여야 한다.

② 출력에서 사람 이름 뒤에 “님”을 부치려면 deliver() 메서드의 매개변수를 고쳐서 실행해야 한다. before() 충고를 사용하면 deliver()가 실행되기 전에 충고의 코드가 실행되고, deliver()가 정작 실행될 때 매개변수가 변하지 않으므로 before() 충고를 사용할 수 없다. 충고를 받는 메서드의 매개변수를 수정하기 위해 around() 충고를 사용한다. around() 충고는 메서드의 기존 환경을 변경하여 실행할 수 있다.

③ AspectJ 키워드인 proceed()는 포착된 결합점을 실행하는 명령이다. 메서드 deliver()의 원래 매개변수 person을 sayToPerson(person)으로 포착하여 proceed() 내에서 “님”을 뒤에 추가하여 proceed()에 넘겨준다. 결과는 변경된 매개변수를 가지고 Message.deliver()를 실행하게 된다.

클래스를 MannersAspect와 KoreanSalutation-Aspect와 함께 컴파일하고 Test 클래스를 실행하면 다음과 같이 출력이 된다.

KoreanSalutation을 포함한 실행결과

```
>ajc Message.java MannersAspect.java
     KoreanSalutationAspect Test.java
>java Test

안녕하세요! AspectJ를 배우고 싶으세요?
안녕하세요! 길동님, 재미있어요?
```

4.3. AspectJ의 구조물의 변환

AspectJ 컴파일러가 컴파일하여 나온 결과물은 Java 가상기계에서 실행되어야 하기 때문에 컴파일된 바이트 코드는 Java의 구조물로 전부 변환되어야 한다. AspectJ 구조물이 Java의 어떤 구조물로 변환되는 것을 (표 2)에서 보여주고 있다.

(표 2) AspectJ 구조물의 변환

AspectJ 구조물	Java의 구조물
애스펙트	클래스
충고	한 개 이상의 메서드
결합점	변환될 필요가 없음
타입간 선언	대상 클래스에 필드, 메서드 도입
컴파일 경고, 오류	바이트 코드에 변화가 없음

앞의 예가 실제 AspectJ 컴파일러에 의해 컴파일되고 직조 되었을 때 코드가 어떻게 변화하는지를 살펴보자. (예제 3)에 있는 애스펙트는 다음과 같은 클래스로 변환될 수 있다.

(변환된 코드 1) MannersAspect.java

```
public class MannersAspect {
  public static MannersAspect asInstance;
  public final void before0$ajc {
    System.out.print("안녕하세요");
   }
  static {
    MannersAspect asInstance
     = new MannersAspect();
   }
}
```

(예제 1)에 있는 클래스는 아래와 같이 변환될 수 있다.

(변환된 코드 2) Message.java

```
public class Message {
  public static void deliver(String msg) {
    MannersAspect.asInstance.before0$ajc();
    System.out.println(msg);
  }
  public static void deliver(String person,
      String msg) {
    MannersAspect.asInstance.before0$ajc();
    System.out.println(person + "," + msg);
  }
}
```

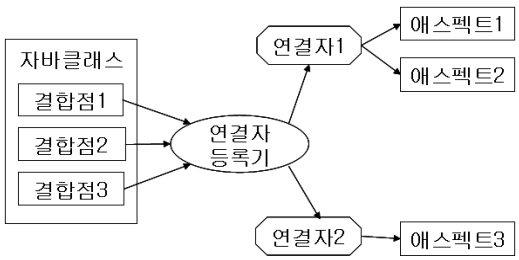
문장 MannersAspect.asInstance.before0$ajc(); 가 System.out.println(msg); 앞에 삽입되어 있는 것을 확인할 수 있다. 그러나 이것은 이해를 돕기 위하여 단지 예로서 제시된 것뿐이지 실제 코드가 위와 동일하게 만들어 지는 것은 아니며 더구나 소스 코드가 변환되는 것이 아니라 바이트 코드가 변환된다.

5. JAsCo

AspectJ와 같은 정적인 컴파일러는 컴파일 시점에 모든 것이 고정되기 때문에 메모리 등의 제약이 있는 소형의 디바이스나 분산 플랫폼에는 적당하지 않다. 반면에 동적인 AOP 프레임워크는 직조가 프로그램 수행 중에 일어나기 때문에 많은 유연성을 제공한다.

이 장에서 동적인 프레임워크의 하나인 JAsCo[13-14]를 소개한다. JAsCo의 개념과 대부분의 구조물은 AspectJ와 동일하다. 따라서 큰 차이를 보이고 있는 것이 런타임 구조이므로 이를 설명하기로 한다. 런타임 구조는 (그림 4)에 표시되어 있다.

실제 JAsCo는 분산 한경을 지원하기 위한 AWED 시스템[18]으로 확장되어 분산 시스템의 관점지향 지원을 위한 새로운 구조물이 추가되었고 런타임 구조도 변경이 되었으나 이에 대한 논의는 지면상의 문제로 생략한다.



(그림 4) JAsCo 런타임 구조

5.1. 연결자(connector)

작성된 애스펙트가 컴파일 시점에 클래스에 직조되지 않기 때문에 JAsCo는 연결자를 도입하였다. 이 연결자는, 결합점을 가진 클래스가 수행 중에, 애스펙트를 활성화하고 클래스와 연결되어 애스펙트의 충고가 실제로 수행될 수 있도록 배치하는 역할을 한다.

5.2. 연결자 등록기(connector registry)

JAsCo의 런타임 구조의 핵심 부분은 연결자 등록기이다. 연결자 등록기가 등록된 연결자와 애스펙트를 관리한다. 새로운 연결자가 시스템 수행 중에 로드되거나 또는 기존의 연결자가 제거될 시에, 연결자 등록기는 통보를 받고 등록된 연결자와 애스펙트를 데이터베이스에 자동으로 수정한다.

5.3. 핫스왑(HotSwap)

핫스왑은 애스펙트가 적용(직조)되어야 하는 결합점에 동적으로 트랩(trap)을 설치

한다. 새로운 애스펙트가 배치되면, 관련된 결합점은 실행 중에 트랩을 가진 결합점과 즉시 교체된다. 만약에 애스펙트가 제거되고 어떤 애스펙트도 직조될 필요가 없으면, 이번에는 원래 바이트 코드가 다시 설치된다.

5.4. 쥬타(Jutta)

쥬타는 JIT 컴파일러로, 애스펙트가 수행될 수 있도록 원래 결합점에 트랩이 설치된 대체 결합점 코드를 매우 효율적으로 생성해 준다.

핫스왑과 쥬타에 근거한 JAsCo 런타임 직조기는 동적인 기능을 유지하면서 또한 AspectJ와 같이 정적으로 컴파일되는 언어와 성능이 거의 비슷하다[13].

6. 관점지향 프로그래밍의 적용

관점지향 프로그래밍의 적용분야[1,4-7]는 아주 작은 시스템으로부터 대형 시스템에 이르기까지 매우 넓고 다양하다. 자원이 부족한 작은 시스템에서는 환경에 따라 변할 수 있는 동적인 시스템을 요구하기 때문에 관점지향 프로그래밍의 직조를 사용하여 꼭 필요한 기능만 선택하여 다양한 시스템을 제작해서 실행할 수 있다. 또한 동적인 직조기를 사용하여 필요한 시점에 필요한 기능을 직조하여 사용하고 필요없는 시점에 제거하는 방법으로 자원의 부족에 효율적으로 대체할 수 있다.

대형 시스템에서는 IBM의 Websphere의 적용 사례[1,6]에서 보듯이, 대용량의 횡단 관심사인 EJB 컴포넌트를 성공적으로 리팩토링하였다. 이는 개발과 유지보수가 더욱 어려워지는 복잡한 소프트웨어 시스템을 관심사 별로 독립적으로 설계 및 구현하여 개발과 유지 보수에 노력이 현저하게 줄어들고, 또한 고객에게 제품을 납품할 때 고객이 필요한 기능만 선택하여 직조할 수 있어 고객의 다양한 요구에 부응할 수 있다. IBM의 성공적인 사례는 관점지향 프로그램의 혜택이 약속이 아니고 현실이란 것을 입증하고 있다[1].

[4]에서 구체적으로 개발하여 제시된 관점지향 프로그래밍의 적용 가능한 분야는 로깅, 트레이싱, 프로파일링, 정책 준수(policy enforcement), 자원 풀링(pooling), 설계 패턴, 이디엄, 스레드 안전, 인증, 권한부여, 트랜잭션 관리, 비즈니스 규칙이며 그 외에 성능[7], 미들웨어 리팩토링[5,6] 등 다양하다.

7. 결론

우리는 이 논문에서 관점지향 프로그래밍의 개념과 구조물 그리고 이를 구현한 대표적인 정적인 언어인 AspectJ와 이를 동적으로 구현한 JAsCo를 논의하였다.

관점지향 프로그래밍은 기존의 프로그래밍 방법론에서 해결하지 못하였던 횡단 관심사의 모듈화를 하기 위해 프로그램에 여러 구조물을 도입하였고 나중에 별도로 구현된 관심사를 직조를 통해 완성된 소프트웨어 시스템을 제작하는 새로운 방법론의 시작을 열었으며 많은 분야에 적용되어 가고 있다. 관점지향 프로그래밍을 사용하면 고도의 모듈화를 이루며, 시스템의 개선이 용이하고, 과잉 설계를 피하며, 코드 재사용을 확대하는 등 많은 혜택을 얻을 수 있다.

관점 지향 프로그래밍은 객체지향 다음 단계의 프로그래밍 언어의 발전 단계로, 이미 많은 연구가 관점지향 학회[15]를 중심으로 진행되고 있으며 앞으로 더욱 발전이 기대된다. 현재 관점지향 모델링[10-12]에도 많은 관심과 연구가 이루어지고 있다.

참 고 문 헌

[1] Daniel Sabbah, "Aspects - from Promise to Reality," 2004 AOSD conference, 2004, pp.1-2

[2] Gregor Kiczales and et la, "Aspect-Oriented Programming," ACM Computing Surveys, Volume 28, Issue 4es, Dec., 1996.

[3] Tizila Elrad and et la, "Aspect-Oriented Programming," Communications of the ACM, Vol..44, No.10, Oct., 2001, pp.29-32.

[4] Ramnivas Laddad, AspectJ in Action, Manning, 2003
(번역본 금영욱, AspectJ in Action, 도서출판 그린 2005)

[5] Charles Zhang and Hans-Arno Jacobsen, "Refactoring Middleware with Aspects," IEEE Transactions and Paralllel and Distributed System, Vol. 14, No. 11, Nov. 2003, pp. 1058-1073

[6] Adrian Colyer and Andrew Clement, "Large-scale AOSD for Middleware," 2004 AOSD conference, 2004, pp.56-65

[7] Erik Putrycz and Guy Bernard, "Using Aspect-Oriented Programming to build a portable load balancing service," Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW'02), 2002

[8] AspectJ, http://eclipse.org/aspectj/

[9] AJDT, http://www.eclipse.org/ajdt/

[10] Omar Aldawud, Tzilla Elrad, and Atef Bader, "UML Profile for Aspect-Oriented Software Development" Workshop on Aspect-Oriented Modeling with UML, 2003 (http://lglwww.epfl.ch/workshops/aosd2003/papers/AldawudAOSD_UML_Profile.pdf)

[11] Ivar Jacobson and Pan-Wei, Aspect-oriented Software Development with Use Cases, Addison Wesley, 2004 Dec.

[12] Siobhan Clarke, Elisa Baniassad, Aspect-Oriented Analysis and Design The Theme Approach, Addison Wesley, 2005 March. framework. In Acta Informatica, 34(9) p653~665, August 1997.

[13] W. Vanderperrn and D. Suvee, JAsCo dynamic AOP through HotSwap and Jutta DAW:Dynamic Aspect Workshop, p.120~134, Mar 2004

[14] JAsCo, JAsCo web site. http://ssel.vub.ac.be/jasco/

[15] AOSD web site http://aosd.net/

[16] Julie Waterhouse and Mik Kersten, "Aspect-Oriented Programming with AspectJ," oopsla 2004

[17] David Parnas, "On the criteria to be used in Decomposing Systems into Modules," Communications of the ACM, 15(12): 1053-58, Dec. 1972

[18] L. Navarro, M. Sudholt, W. Vanderperrn and D. Suvee, "Explicitly distributed AOP using AWED," 2006 AOSD conference, pp.51-62

금 영 욱

1974년~1978년 서울대학교수학과 (학사)

1990년~1992년 서강대학교 전산학과 (석사)

1993년~1997년 서강대학교 전산학과 (박사)

1982년~1993년 IBM 시스템즈 엔지니어

1997년~현재 성결대학교 컴퓨터공학부 부교수

관심분야 : 병렬/분산처리, 프로그래밍 언어, 컴포넌트 소프트웨어 개발