DOI: 10.3745/KIPSTD.2009.16-D.1.93

특성 지향의 제품계열공학을 위한 애스팩트 구현 패턴

이 관 우[†]

요 약

특성 지향 제품계열공학은 특성 관점에서 제품계열의 핵심자산을 개발하고 이를 활용하여 제품을 개발하는 접근방법으로서, 이를 위한 첫 번째 단계는 하나의 특성을 하나의 모듈화된 단위로 구현하는 것이다. 관점 지향 프로그래밍은 특성 구현의 모듈화를 향상시키기 위한 효과적인 메커니즘을 제공한다. 하지만, 특성이 일반적으로 서로 독립적이지 않기 때문에 어떤 특성 구현 모듈의 변화는 다른 특성 구현 모듈에 변화를 일으키거나 원하지 않는 부작용을 야기시킬 수도 있다. 뿐만 아니라, 하나의 특성이 제품에 결합되는 시점이 컴파일 시점에서부터 로드 시점, 실행 시점에 이르기까지 다양할 수 있으므로, 특성이 언제 제품에 결합하느냐에 따라 다르게 구현되어야 할지도 모른다. 따라서, 본 논문에서는 각 특성 구현 모듈이 다른 모듈과 독립적이 되도록 하기 위해서, 특성 구현 모듈로부터 특성 의존성 및 특성 결합 시점을 효과적으로 분리시킬 수 있는 애스팩트 패턴을 제안한다. 이러한 패턴들은 특성 구현 모듈이 특성의 선택에 따라서 다른 모듈에 영향을 주지 않고 유연하게 합성될 수 있도록 한다. 이와 같은 접근 방법을 예시하고 평가하기 위해 공학용 계산기 제품계열을 사용한다.

키워드 : 소프트웨어 제품계열 공학, 특성 지향 분석, 구현 패턴, 관점 지향 프로그래밍

Aspectual Implementation Patterns for Feature-Oriented Product Line Engineering

Kwanwoo Lee[†]

ABSTRACT

Modular implementation of a feature is a first step toward feature-oriented product line engineering, which develops and then utilizes core assets to configure products in terms of features. Aspect-oriented programming provides effective mechanisms for improving the modularity of feature implementations. However, as features in general are not independent of each other, changes in the implementation of one feature may cause changes to or side effects in the implementation of other features. Moreover, since the time at which a feature is incorporated into products, called feature binding time, may be various from compile time through load time to run time, a feature may have to be implemented differently depending on when the feature is bound into a product. To make each feature implementation module as independent as possible, this paper proposes aspectual implementation patterns that can effectively separate feature dependencies as well as feature binding time from feature implementation modules. These patterns enable flexible composition of feature implementation modules without affecting other modules according to feature selection. The approaches are demonstrated and evaluated based on a product line of scientific calculator applications.

Keywords : Product Line Engineering, Feature-Oriented Analysis, Implementation Patterns, Aspect-Oriented Programming

1. 서 론

제품계열공학(product line engineering)[5]은 소프트웨어 제품을 개별적으로 개발하기 보다는 유사한 제품 개발에 재사용될 수 있는 공통의 핵심자산(아키텍처 혹은 구현 모듈)을 미리 개발한 후에, 이를 체계적이고 계획적으로 재사용함으로써, 다양한 고객의 요구에 맞는 제품을 저비용으로 적기에 생산하는 것을 목표로 한다. 이러한 목표를 달성하기 위해서는 제품계열(product line) 범위내의 소프트웨어 제품 간의 공통성과 가변성을 분석하고, 이에 대한 이해를 바탕으로 다양한 제품 환경에 쉽게 재사용될 수 있고, 적응될 수 있는 핵심자산을 개발하는 것이 필수적이다.

지금까지 많은 제품계열공학 방법들[6, 9, 11]은 *특성 지향의 접근방법 (feature-oriented approach)*을 사용해 오고 있다. 즉, 제품 간의 공통성과 가변성을 *특성 (feature)* 관점에서 분석하고, 이를 바탕으로 공통 특성은 재사용 가능한 형태의 자산으로 개발하고, 제품에 따라 선택될 수 있는 가변 특성은 쉽게 구성될 수 있거나 적응될 수 있는 형태의 자산으로 개발한다. 하지만, 이와 같은 형태의 핵심자산을 개발하

※ 본 연구는 2008년도 한성대학교 교내연구비 지원과제임.
† 정 회 원 : 한성대학교 정보시스템공학과 조교수
논문접수 : 2008년 11월 7일
심사완료 : 2008년 12월 3일

는 것은 다음과 같은 세 가지 요인으로 인해 간단하지 않다.

- *특성 구현의 횡단성*: 일반적으로 특성의 단위와 구현 모듈의 단위와는 항상 일치하지 않는다. 즉, 하나의 특성이 여러 모듈에 걸쳐 구현될 수도 있다. 예를 들어, 시스템의 실행 로그를 기록하는 특성을 구현한다면, 시스템 내의 대다수 모듈들이 실행 로그를 저장하기 위한 코드를 포함해야 할 것이다. 특히, 이와 같이 여러 모듈에 구현이 걸치는 횡단성을 가지는 특성이 제품에 따라서 선택 혹은 선택되지 않을 가변성을 가진다면, 특성 선택에 따른 핵심자산의 선택 및 적응 절차가 매우 복잡할 것이다.
- *특성 간의 의존성*: 일반적으로 특성들은 서로 독립적이기 보다는 다른 특성과 의존 관계를 가진다. 만약, 이러한 의존관계를 구현한 코드가 특성을 구현한 모듈에 포함되어 있다면, 가변특성의 선택에 따라, 그 특성과 관련된 의존관계를 구현한 코드를 포함한 모듈도 영향을 받게 된다. 즉, 한 특성의 변화(선택 혹은 비선택)가 다른 특성을 구현한 모듈에 영향을 주게 된다. 이것이 의미하는 바는 한 가지 요인(가변특성)의 변화가 핵심자산의 여러 부분에 영향을 줄 수 있어서, 핵심자산의 적응을 어렵게 만들 수 있게 된다는 것이다.
- *가변특성의 제품 결합 시점*: 특성의 제품 결합이란 특성과 관련된 핵심자산의 일부가 특정 제품을 위해 포함되거나 적응되어서, 그 특성에서 정의된 능력(capability)을 제품이 제공할 수 있음을 의미한다. 가변특성의 경우, 제품에 결합되는 것(핵심자산 일부의 선택과 적응)이 대개 제품을 개발하는 시점(예, 프로그램 컴파일시점)에 이루어지지만, 제품의 유연성(flexibility)을 위해서 제품 실행 시점까지 이를 연기할 수도 있다. 언제 가변특성이 제품에 결합되느냐에 따라서 이를 구현하는 방식이 달라질 수 있다. 특히, 어떠한 특성의 제품 결합 시점이 제품에 따라 가변성을 가진다면, 그 특성을 구현한 모듈이 제품 결합 시점의 가변성에 따라서 변할 수가 있으므로, 특성 구현 모듈의 재사용성을 떨어뜨리게 된다.

특성 구현의 횡단성 문제는 최근 들어 등장한 관점 지향 프로그래밍(AOP:Aspect-Oriented Programming) [13]을 이용하면 효과적으로 해결할 수 있다. 즉, 특성 구현이 여러 구현 모듈에 걸쳐 구현되어야 하는 경우에, 이를 애스팩트(Aspect)라고 불리는 독립적인 모듈로 캡슐화를 시킬 수 있다. AOP를 제품계열공학에 적용하는 많은 연구들[1, 7, 12, 16]은 특성 구현의 횡단성 문제를 효과적으로 해결하는 데 공헌하였다.

하지만 애스팩트로 모듈화된 특성 구현 모듈이 특성 의존성 및 특성 결합 시점 정보를 포함하고 있다면, 이들의 가변성이 특성 구현 모듈에 영향을 주게 되어, 특성 구현 모듈의 재사용성 및 적응성을 떨어뜨리게 된다. 따라서, 본 논문에서는 AOP의 강력한 메커니즘을 이용하여, 제품계열의 분석 단계에서 찾아진 특성 간의 의존성 및 제품 결합 시점 등의 정보를 특성 구현 모듈로부터 효과적으로 분리시키기 위한 방법으로서 애스팩트 구현 패턴을 제안한다. 특히, 대표적인 AOP 언어인 AspectJ [2]를 바탕으로 애스팩트 구현 패턴을 설명한다.

본 논문의 구성은 다음과 같다. 2장에서는 AOP를 이용하여 특성 구현을 모듈화하는 방법을 설명하고, 이를 확장하기 위해서 필요한 특성 간의 의존성과 특성 결합 시점의 개념에 대해서 간단히 설명한다. 3장에서는 AOP 메커니즘을 활용한 특성 의존성 및 특성 결합 시점의 애스팩트 구현 패턴을 제안한다. 4장에서는 정의된 패턴을 공학용 계산기 제품계열에 적용한 사례 연구를 기술한다. 5장에서는 기존 연구와 본 연구와의 차이에 대해서 비교분석을 하고, 6장에서는 본 연구에 대한 토의와 향후 연구 과제에 대해 언급하면서 결론을 내린다.

2. 기반 연구

2.1 AOP를 이용한 특성 구현의 모듈화

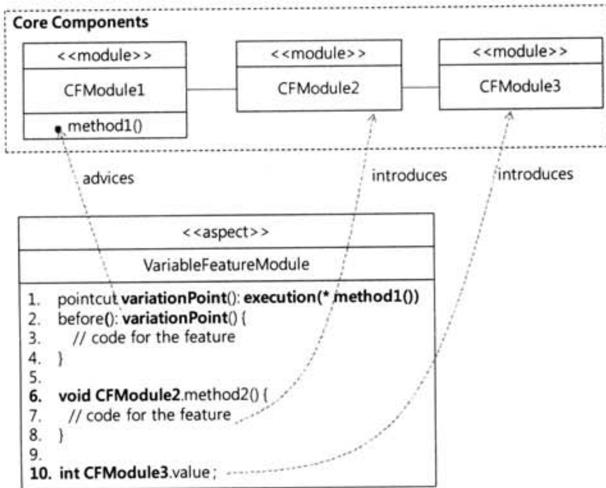
특성이란, 다양한 이해당사자(stakeholder) 관점에서 보여지는 제품간의 구별되는 혹은 두드러진 특징으로 정의된다 [10]. 특성모델[10]은 제품계열 내의 제품간의 공통성과 가변성을 특성 관점에서 표현한다. 제품계열 내의 모든 제품 간에 공통인 요소는 *공통특성*(common feature)으로 표현되고, 제품 간에 구별되는 요소는 *가변특성* (variable feature)으로 표현된다.

가변특성은 제품에 따라 포함될 수도 있고 포함되지 않을 수도 있는 특성이므로, 모든 제품에 항상 포함되는 공통 모듈과 명확히 분리 시킬 필요가 있다. 만약 가변특성의 구현이 하나의 모듈로 한정될 수 있다면, 공통 모듈과의 분리가 어렵지 않겠지만, 가변특성을 구현하기 위해서 기존의 공통 모듈들을 확장 혹은 수정해야 한다면, 가변특성을 공통 모듈로부터 분리시키는 것이 간단하지 않다.

AspectJ는 공통 모듈로부터 가변특성의 구현 모듈을 명확히 분리시키기에 효과적인 메커니즘으로 *포인트컷*(pointcut), *어드바이스* (advice), *인터타입 선언*(intertype declaration)을 제공하고 있다. *포인트컷*은 가변특성에 의해서 확장 및 수정이 이루어져야 할 필요가 있는 공통 모듈의 특정 위치를 정의할 때 사용될 수 있고, *어드바이스*는 *포이트컷*이 지정한 위치에 가변특성의 구현 코드를 삽입 및 대체할 때 사용될 수 있다. *인터타입 선언*은 공통 모듈의 새로운 메소드나 맴버 변수를 추가할 때 사용될 수 있다.

(그림 1)은 기존 공통 모듈들의 확장 및 수정을 요구하는 가변특성의 구현을 공통 모듈로부터 명확히 분리시키기 위해 애스팩트를 이용한 방법을 나타낸다. (그림 1)의 <<module>> 스테레오 타입은 <<class>> 혹은 <<aspect>> 타입의 일반화를 나타낸다.

VariableFeatureModule 애스팩트는 공통 모듈 CFModule1을 확장하기 위해서, 먼저 포인트컷 명세(1 줄)를 통해서 공통 모듈의 변경 위치(method1()의 수행 지점)를 정의하고,



(그림 1) 가변 특성의 애스팩트 구현

정의된 변경 위치 앞에 어드바이스 코드를 삽입한다. 어드바이스 코드의 삽입을 변경 위치 뒷부분에 삽입하기 위해서는 *before* 대신에 *after* 키워드를 사용할 수 있고, 변경 위치의 메소드를 어드바이스 코드로 대체하기 위해서는 *around* 키워드를 사용할 수 있다. 한편, 공통 모듈 CFModule2에 새로운 메소드 method2()를 추가하고 CFModule3에 새로운 int 형 멤버 변수 value를 추가하기 위해서 인터타입 선언을 사용한다.

이와 같이 공통특성을 구현한 모듈로부터 가변특성을 명확히 분리시킬 수 있지만, 가변특성을 구현한 모듈이 다른 가변특성과의 의존성을 구현한 코드를 포함하고 있다면, 다른 가변특성의 선택 여부에 따라서 영향을 받게 된다. 뿐만 아니라, 가변특성이 제품과 결합되는 시점이 제품에 따라 변한다면, 가변특성을 구현한 모듈이 요구되는 시점에 제품과 결합될 수 있도록 다른 방식으로 구현되어야 할 것이다.

본 논문에서는 특성 구현 모듈로부터 특성 간의 의존성 및 제품 결합 시점에 대한 정보를 명확히 분리시킴으로써, 특성 구현 모듈의 재사용성 및 적응성을 높일 수 있는 애스팩트 구현 패턴을 제안한다. 따라서, 이어지는 절에서는 본 논문에서 제안할 애스팩트 구현 패턴을 이해하는데 필요한 기반 개념으로 특성 의존성과 제품 결합 시점에 대해서 설명한다.

2.2 특성 의존성

특성들은 일반적으로 서로 독립적이지 않다. 특성들 사이에 다양한 의존 관계가 존재하지만, 본 논문에서는 [15]에서 정의된 기능 특성 간의 수행의존 (operational dependency) 관계에 초점을 둔다. 수행의존 관계란 시스템의 실행 중에 한 특성의 수행이 다른 특성의 수행에 직·간접적으로 영향을 미치는 관계를 의미한다. 이러한 수행의존 관계에는 변경(Modification)과 활성화(Activation) 관계들이 있다.

특성 간의 변경 의존성은 한 특성의 기능이 다른 특성의 기능에 의해서 확장 및 수정되는 것을 의미한다. 예를 들면, 공학용 계산기에는 history 특성 (이전에 계산된 수식들을 리스트에 저장하고 리스트를 탐색하는 기능)과 base 특성 (수식의 기저를 변환시키는 기능)이 있다. base 특성은 실행 시에 수식의 기저를 변경시키므로, history 특성이 관리하는 수식들도 변경된 기저를 저장할 수 있도록 확장되어야 한다. 즉, history 특성은 그의 기능이 base 특성에 의해서 확장되는 변경 의존성을 가진다.

기능 특성 간의 활성화 의존성은 한 특성의 활성화가 다른 특성의 활성화에 영향을 받는 경우에 정의된다. 이는 크게 배타 활성화(excluded activation), 종속 활성화(subordinate activation), 동시 활성화 (concurrent activation), 순차 활성화 (sequential activation) 로 구분된다.

두 기능 특성 간의 배타 활성화는 한 특성의 기능이 다른 특성의 기능 수행을 제한하는 경우에 해당된다. 예를 들면, 공학용 계산기에서 mode 특성(계산기의 설정 모드를 변경하는 기능)은 history 특성의 활성화를 제한한다. 그 이유는 두 특성이 모두 디스플레이 패널을 공유하고 있기 때문이다.

종속 활성화는 한 특성의 활성화가 다른 특성이 활성화되어 있는 경우에만 활성화 될 수 있다는 의존성을 나타낸다. 예를 들면, logicalOp 특성 (논리 연산 기능)은 이진 수에 대해서 적용되므로, binaryBase특성(수식의 기저를 이진으로 설정)이 활성화된 상태에서는 활성화 될 수 있다.

동시 활성화는 한 특성이 다른 특성과 함께 활성화 되어야 하는 경우에 정의되고, 순차 활성화는 한 특성이 수행을 마친 후에 다른 특성이 활성화 되어야 하는 경우에 정의된다. 예를 들면, memory 특성(수식의 계산 결과를 메모리에 저장)은 evaluation 특성 (수식 계산)이 수행을 마친 후에 활성화되어야 하는 의존관계를 가진다.

이와 같은 특성 간의 의존성으로 인해, 특성의 선택 여부가 다른 특성의 행위에 영향을 줄 수 있다. 만약 이러한 특성 간의 상호작용을 구현한 코드가 특성을 구현한 모듈에 내재되어 있다면, 가변특성의 선택여부에 따라서 그와 관련된 상호작용이 포함된 구현 모듈들이 영향을 받게 된다. 따라서 특성 구현 모듈의 독립성을 위해서는 특성 간의 의존성으로 인한 상호작용을 특성 구현 모듈로부터 명확히 분리시키는 것이 효과적이다. 3장에서는 이를 위한 구체적인 패턴에 대해서 설명한다.

2.3 특성 결합 시점

가변특성은 사용자 요구에 따라 선택되고 제품에 결합된다. 특성이 제품에 결합된다는 의미는 그 제품이 특성에서 정의된 기능을 제공할 수 있는 상태라는 것을 나타낸다. 특성이 제품에 결합되는 시점을 크게 구분하면, 컴파일-시점(compile-time), 로드-시점 (load-time), 실행-시점 (run-time) 등으로 나뉘어진다. 컴파일-시점 결합은 구현된 특성이 프로그램 컴파일 시점 혹은 선처리 시점에 다른 특성과 결합되는 것을 말한다. 즉, 특성 결합 혹은 해지를 위해서는 새로이 코드를 컴파일을 해야 한다. 특성의 로드-시점 결합은 특성 구현 코드가 결합되어야 할 제품의 다른 모듈이 메모리에 로딩될 때, 그 모듈에 결합되는 것을 의미한다. 마지막으로 실행-시점 결합은 프로그램의 실행 중에 구현된 특성이 제품의 다른 모듈과 결합되는 것을 의미한다.

특성의 제품 결합은 특성 구현 모듈이 제품의 다른 구현 모듈을 사용하거나 변경시킬 수 있도록 구현 모듈 간의 연결 관계를 설정하고, 특성 구현 모듈이 사용 가능한 상태가 되도록 설정하는 과정이라고 볼 수 있다. 만약, 특성 구현 모듈이 제품과 연결이 이루어졌다 하더라도 사용 가능하지 않다면 특성의 제품 결합이 이루어지지 않은 것으로 간주된다. 따라서, 특성이 언제 제품과 결합되는 지는 제품과의 연결이 언제 이루어지고, 가용성이 언제 결정되느냐에 따라 다양하게 나타날 수 있다. 가령, 실행 시점에 결합하는 특성을 구현할 때, 제품과의 연결은 프로그램 컴파일-시점에 이루어지더라도 가용성이 실행-시점에 결정되도록 구현될 수도 있고, 제품과의 연결과 가용성이 모두 실행 시점에 이루어지도록 구현될 수도 있다.

요약하면, 가변특성의 제품결합이 언제 어떻게 이루어지는가에 따라서 가변특성이 구현되는 방식이 달라지게 된다. 따라서 특성 결합 시점 분석은 각 가변특성이 언제 제품과 연결되고 언제 가용하게 되는 지를 분석하게 되고, 이러한 분석 결과를 바탕으로 각 가변특성은 요구되는 시점에 제품과 결합될 수 있도록 구현되어야 한다.

3. 핵심자산의 애스팩트 구현 패턴

본 논문에서는 가변특성의 선택에 따라 핵심자산의 구성 및 적응이 용이하도록 하기 위해서 다음의 두 가지 접근 방법을 사용한다.

- *가변특성간의 의존성 분리*: 가변특성 구현 모듈의 독립성을 높여, 보다 유연한 가변특성 구현 모듈의 조합을 가능하게 하기 위해서 가변특성 간의 의존성을 특성 구현 모듈로부터 분리시킨다
- *다양한 특성 결합 시점 지원*: 가변특성을 구현한 모듈이 원하는 시점에 제품과 결합할 수 있도록 특성 결합과 관련된 코드를 분리한다.

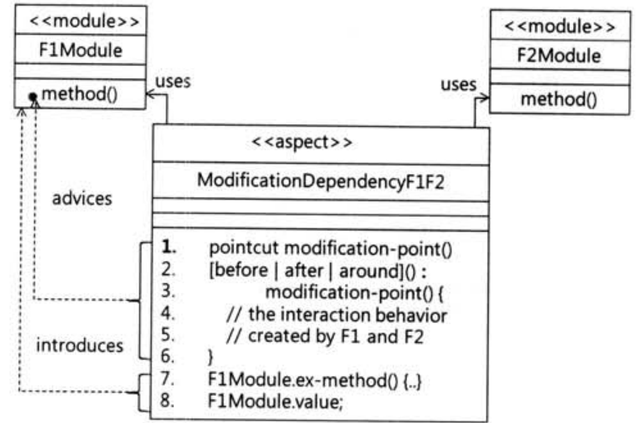
이어지는 절에서는 위의 두 가지 접근 방법에 대한 애스팩트 구현 패턴을 제안한다.

3.1 특성 간의 의존성 분리

가변특성 선택에 따라 가변특성을 구현한 모듈을 핵심자산 내의 다른 모듈에 영향을 주지 않고 구성시키기 위해서는 가변특성 구현 모듈 간에 의존성이 없어야 한다. 다음은 앞에서 설명한 특성 간의 의존성 종류 별로, 이를 특성 구현 모듈로부터 명확히 분리시키기 위한 애스팩트 구현 패턴을 차례대로 설명한다.

3.1.1 변경 의존성 분리

두 특성 F1과 F2에 대해서, F1이 F2에 변경 의존성을 가진다고 가정하자. 즉, F1의 행위가 F2에 의해서 확장 혹은 변경되는 관계에 있음을 나타낸다. 이 경우에 F1을 구현한 모듈은 F1의 핵심 기능을 구현한 부분과 F2에 의해 변경되



(그림 2) 변경 의존성의 구현 패턴

는 부분으로 구분될 수 있다. F2에 의해서 변경되는 부분은 F2의 선택여부에 따라서 변할 수 있는 부분이므로, F1의 핵심 기능을 구현한 부분과 명확히 분리될 필요가 있다.

(그림 2)에서 보는 바와 같이 F1Module과 F2Module 모듈들은 각각 특성 F1과 F2의 핵심 기능을 구현한 모듈이고, ModificationDependencyF1F2 애스팩트는 F1과 F2사이의 변경의존성을 애스팩트로 구현한 것을 나타낸다. (그림 1)의 가변특성 구현 패턴과 유사하게, ModificationDependencyF1F2 애스팩트는 포인트컷과 어드바이스 메커니즘(1-6 줄)을 이용하여 F1Module을 변경하거나, 인터타입 선언 메커니즘(7-8 줄)을 이용하여 F1Module을 확장한다. 또한 ModificationDependencyF1F2 애스팩트는 F1Module을 변경하거나 확장하기 위한 코드를 정의할 때, F1Module과 F2Module을 이용할 수 있다. 이는 <<aspect>> 타입과 <<module>> 타입 사이의 uses 관계로 표현되었다.

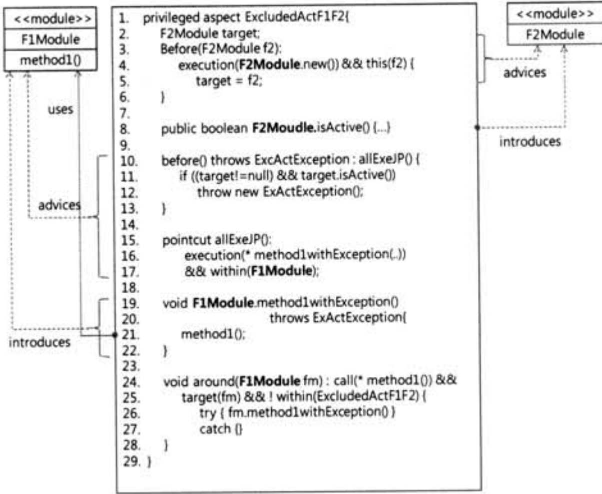
이와 같이 F1과 F2사이의 변경의존성이 F1과 F2를 구현한 모듈(F1Module1, F1Module2)들로부터 분리되어 독립적인 애스팩트로 구현될 수 있으므로, F1Module1은 특성 F2의 선택여부와 상관없이 그대로 재사용될 수 있고, ModificationDependencyF1F2 애스팩트는 F1과 F2 특성 모두가 선택되었을 때만, 제품에 포함되면 되고, 그렇지 않은 경우에는 포함되어서는 안된다.

3.1.2 활성화 의존성 분리

특성 간의 활성화 의존성은 활성화와 관련된 두 특성 간의 상호작용이라고 볼 수 있다. 이러한 특성 간의 상호작용은 특성 구현 모듈로부터 분리된 독립적인 애스팩트로 정의할 수 있다. 다음은 2.2절에서 언급한 각 의존성이 어떻게 독립적인 애스팩트로 분리될 수 있는 지를 설명한다.

두 특성 F1과 F2에 대해서, F1이 F2에 배타 활성화 의존성을 갖고, F1과 F2를 구현한 특성 모듈 타입이 각각 F1Module 과 F2Module이라고 가정하자. (그림 3)의 애스팩트 ExcludedActF1F2는 F1과 F2 사이의 배타 활성화 의존성을 구현한 예를 나타낸다.

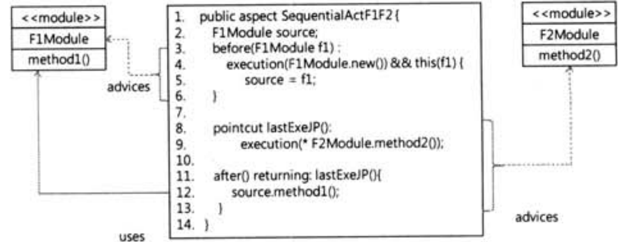
F1이 F2에 배타 활성화 의존성을 가지므로, F1Module모듈 인스턴스(instance)의 수행은 F2Module모듈 인스턴스의



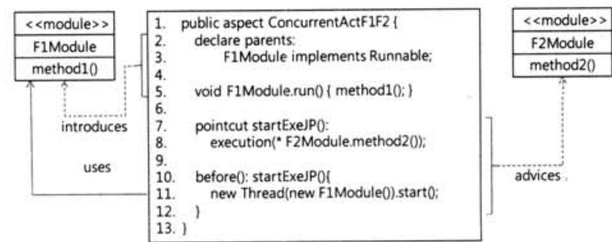
(그림 3) 배타 활성화 의존성 애스팩트

```
1.     public abstract aspect SubordinateActF1F2 {
2-9.     // the same as the ExcludedActF1F2
10.        before() throws SubActException: allExeJP() {
11.          if ((target==null) || ! target.isActive())
12.            throw new SubActException();
13.        }
14-28.   // the same as the ExcludedActF1F2
29.    }
```

(그림 4) 종속 활성화 의존성 애스팩트



(그림 5) 순차 활성화 의존성 애스팩트



(그림 6) 동시 활성화 의존성 애스팩트

실행여부에 따라서 결정된다. F2Module모듈 인스턴스의 실행여부를 판단하기 위해서, F2Module에 isActive() 메소드를 확장하고 (8 줄), 이를 바탕으로 F1Module모듈 인스턴스의 수행이 결정된다. ExcludedActF1F2의 10-13 줄에서 보는 바와 같이, F1Module모듈 인스턴스의 모든 메소드 수행 시작 지점(포인트컷 allExeJP에서 정의)에서 F2Module모듈 인스턴스의 isActive 메소드의 결과에 따라서 수행을 계속하던가, 예외 ExActException 을 발생시킴으로써 메소드의 수행을 중단시킨다. 주지할 점으로, F1Module모듈 인스턴스의 수행 시작 지점에서 F2Module모듈 인스턴스를 접근하기 위해서, F2Module모듈 타입의 인스턴스 변수 target을 선언(2줄)하고, 이 변수의 값을 F2Module모듈 인스턴스가 생성되는 시점에 초기화한다 (3-6 줄). 이 target 변수를 통해서 F1Module모듈 인스턴스의 수행 시작 지점에서 F2Module모듈 인스턴스를 접근할 수 있다.

한편, F1Module모듈 인스턴스의 모든 메소드 수행 시작 지점을 정의하는 포인트컷 allExeJP (15-17 줄)을 살펴보면, F1Module 모듈 타입에 정의된method1() 대신에 method1WithException()을 지정한 것을 알 수 있다. method1WithException() (19-22 줄)은 method1() 수행 중에 예외(Exception) 을 발생시킬 수 있도록 method1()의 메소드 시그니처를 확장시키는 역할을 하는 것으로, 실질적으로는 method1()을 수행하게 된다. 이는 현재 AspectJ에서 기존 메소드의 시그니처를 확장할 수 있는 직접적인 메커니즘을 제공하지 않기 때문에 우회적으로 이를 구현하기 위한 패턴이다.

끝으로, 프로그램의 다른 부분에서 F1Module의 method1()을 호출하는 모든 부분을 method1WithException()을 호출하는 것으로 변경시킴으로써(24-28 줄), 배타 활성화 의존성의 구현이 완성이 된다.

종속 활성화 의존성은 배타 활성화 의존성과 매우 유사하다. (그림 4)에서 보는 바와 같이 10-13 줄을 제외하곤 다른 부분은 배타 활성화 의존성을 구현한 ExcludedActF1F2과 동일하다. 즉, 두 특성 F1과 F2 사이의 종속 활성화란 F1의 활성화가 F2가 활성화가 된 경우에만 가능하므로, F1Module 모듈 인스턴스의 모든 메소드 수행 시작 지점에서 F2Module 모듈 인스턴스의 활성화 여부를 판단하여 활성화 되어 있지 않으면 예외 SubActException을 발생시키고, 활성화 되어 있다면 정상적인 수행을 하면 된다.

순차 활성화 의존성은 한 특성이 수행을 마친 후에 바로 다른 특성이 수행되어야 함을 의미한다. 따라서, (그림 5)의 SequentialActF1F2는 F2Module 모듈 인스턴스가 수행을 마친 후에 F1Module 모듈 인스턴스를 수행시키는 의존성을 구현한 예이다. 먼저, F2Module 모듈 인스턴스의 마지막 메소드 수행 지점을 포인트컷 lastExeJP (8-9줄)에 정의하고, 이 수행 지점이 끝난 후에 F1Module 모듈 인스턴스를 시작시킨다(11-13줄). 이때, F1Module 모듈 인스턴스를 접근하기 위해서 source라는 변수를 사용하였는데, 이는 F1Module 모듈 인스턴스가 생성될 때, 초기화된 변수이다(3-6 줄).

동시 활성화 의존성은 한 특성이 활성화 될 때, 다른 한 특성도 같이 활성화 되어야 함을 의미한다. 두 특성이 동시에 수행될 수 있도록 하기 위해서는 서로 다른 쓰레드(thread)나 프로세스에서 수행될 수 있어야 한다. (그림 6)의 ConcurrentActF1F2 애스팩트는 두 특성 F1과 F2를 구현한 F1Module과 F2Module 모듈 인스턴스들을 동시에 수행시키기 위해 쓰레드를 이용한 방법을 나타낸다.

F2Module 모듈 인스턴스가 수행되는 지점은 포인트컷 startExeJP (7-8 줄)에 정의되고, 이 지점 전에 F1Module 모듈 인스턴스를 새로운 쓰레드로 수행시킨다 (11 줄). F1Module을

새로운 쓰레드로 수행시키기 위해서, F1Module이 Runnable 인터페이스를 구현하도록 확장시키고 (2-3줄), Runnable 인터페이스의 run 메소드를 구현 (5줄) 한다.

이상과 같이 특성 간의 다양한 의존성은 특성 구현 모듈로부터 분리되어 독립적인 모듈로 구현될 수 있다. 따라서 가변특성 간에 존재하는 의존성은 그와 관련된 모든 가변특성이 존재할 때만 필요하므로, 이를 구현한 애스팩트 모듈은 가변특성을 구현한 모듈이 제품에 포함된 경우에만 같이 구성되고, 그렇지 않은 경우에는 제품 구성에서 삭제하면 된다.

3.2 특성 결합 구현

특성 구현 모듈을 제품과 결합시키기 위해서는 특성 구현 모듈과 제품의 다른 모듈과 연결 관계를 설정하고, 특성 구현 모듈이 사용 가능한 상태로 설정되어야 한다. 특성 구현 모듈의 연결과 가용성이 언제 결정되느냐에 따라 다음과 같이 여섯 가지로 특성 결합 구현 기법이 구분될 수 있다.

1. 컴파일-시점 특성 결합: 컴파일-시점 연결, 컴파일-시점 가용
2. 로드-시점 특성 결합: 컴파일-시점 연결, 로드-시점 가용
3. 로드-시점 특성 결합: 로드-시점 연결, 로드-시점 가용
4. 실행-시점 특성 결합: 컴파일-시점 연결, 실행-시점 가용
5. 실행-시점 특성 결합: 로드-시점 연결, 실행-시점 가용
6. 실행-시점 특성 결합: 실행-시점 연결, 실행-시점 가용

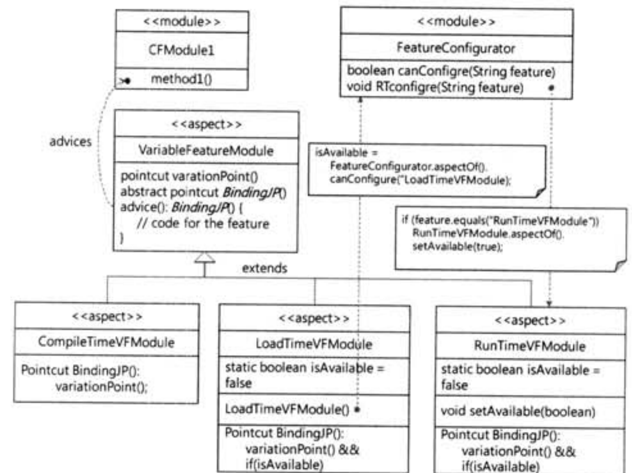
특성 구현 모듈의 연결과 가용성이 같은 시점에 일어나는 첫 번째와 세 번째의 경우에는 AspectJ의 컴파일-시점 직조(weaving)와 로드-시점 직조 메커니즘을 직접 이용하면 된다. 하지만, 여섯 번째의 경우는 AspectJ에서 애스팩트의 실행-시점 직조 메커니즘을 직접 제공하지 않으므로, 이를 가능하게 하기 위한 패턴을 제안한다.

한편, 가용성의 결정이 연결이 된 이후에 일어나는 두 번째, 네 번째, 다섯 번째의 경우는 유사한 패턴으로 정의될 수 있다. 따라서 본 절에서는 위의 두 가지 경우에 대해서 구체적인 패턴을 설명한다.

3.2.1 컴파일 시점 연결/ 로드 및 실행 시점 가용

AspectJ는 컴파일-시점 직조를 통해서 애스팩트 모듈과 다른 모듈들과의 연결이 정적으로 확립되더라도, 애스팩트 모듈들이 실질적으로 사용되기 직전에 동적 로딩(dynamic loading) 방식을 통해서 메모리에 로딩되고 초기화된다. 따라서, 컴파일 시점에 연결이 된 애스팩트 모듈을 그 이후에 사용 가능하게 하기 위해서는 애스팩트 생성 시(로드 시점) 혹은 제품 실행 시에 연결된 애스팩트 모듈을 사용 가능하게 허용하면 된다.

(그림 7)의 VariableFeatureModule은 가변특성을 애스팩트로 구현한 것으로, 공통특성을 구현한 CFModule1의 지정된 위치에서 결합된다. VariableFeatureModule의 포인트컷 variationPoint는 VariableFeatureModule에 의한CFModule1의 가변 지점을 명세한 것이다. 한편, VariableFeatureModule이 다양한 시점에 사용 가능하게 하기 위해서, 가용성 결정을



(그림 7) 특성 결합 시점의 다양성

하위 애스팩트인 CompileTimeVFModule, LoadTimeVFModule, RunTimeVFModule 에서 내리도록 하였다. 이를 위해서 추상 포인트컷 BindingJP를 정의하고, 하위 애스팩트에서 가용성 결정을 위한 구체적인 포인트컷 명세를 BindingJP로서 재정의하게 한다.

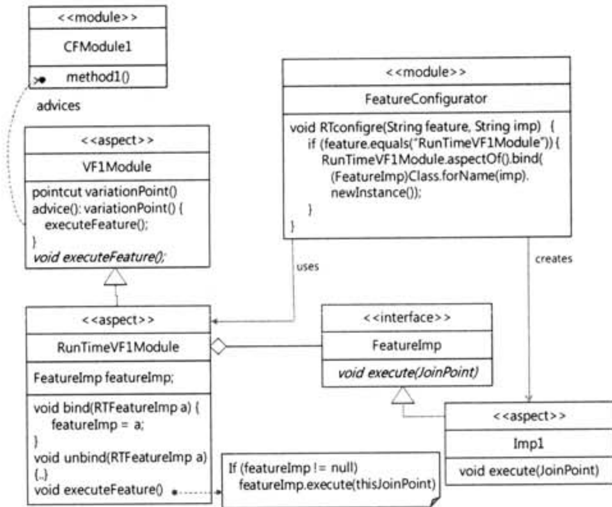
CompileTimeVFModule은 포인트컷 BindingJP를 VariableFeatureModule의 포인트컷 variationPoint로 재정의함으로써, 컴파일 시점에 연결과 가용성의 결정이 동시에 이루어지게 하였다. 반면에 LoadTimeVFModule과 RunTimeVFModule은 포인트컷 BindingJP를 포인트컷 variationPoint와 if(isAvailable)의 논리 곱으로 정의함으로써, isAvailabe의 값이 참인지 거짓인지에 따라서 특성 구현 모듈의 가용성이 결정되도록 하였다. 다만 LoadTimeVFModule과 RunTimeVFModule은 isAvailabe 값이 언제 결정되느냐가 다르다. LoadTimeVFModule은 이 애스팩트가 생성시에 FeatureConfigurator의 canConfigure 메소드(파라미터로 받은 특성이 제품을 위해서 선택되었으면 참을 그렇지 않으면 거짓을 반환)를 통해서 isAvailable 값이 결정된다. 반면에, RunTimeVFModule은 제품 실행 시에 FeatureConfiguration의 RTconfigure 메소드를 통해서 isAvailable 값이 결정된다.

이와 같이 컴파일 시점에 정적으로 연결된 특성 구현 모듈의 코드를 로드 시점 및 실행 시점에 가용하게 만드는 부분을 가변특성 구현 모듈(예, VariableFeatureModule)로 부터 분리 시킴으로써, 가변특성의 제품결합 시점에 대한 다양성을 핵심자산의 변경 없이 제공할 수 있다.

3.2.2 실행 시점 포함/ 실행 시점 가용

AspectJ는 실행 시점에 애스팩트 모듈과 클래스 모듈과의 직조 메커니즘을 제공하지 않는다. AspectJ 대신에 Prose[17]와 같은 실행시점 직조 메커니즘을 제공하는 환경을 이용할 수도 있지만, 이 절에서는 AspectJ를 이용하여 실행 시점에 애스팩트 모듈을 다른 모듈과 직조할 수 있는 패턴을 소개한다.

(그림 8)에서 보는 바와 같이, VF1Module은 가변특성을 구현한 모듈로서 CFModule1의 지정된 위치(포인트컷 variationPoint에 정의)에 어드바이스 코드를 삽입하게 된다. 하지만 삽입될 어드바이스 코드가 실행 중에 FeatureConfigurator에 의해서



(그림 8) 실행 시점 연결 및 가용

VF1Module에 동적으로 로딩되어 결합되게 하기 위해서 다음과 같은 방식을 따른다. 먼저 삽입될 어드바이스 코드는 FeatureImp 인터페이스를 따르는 독립 모듈로 구현한다. 독립 모듈로 구현된 어드바이스 코드를 VF1Module에 삽입/삭제하기 위해서, VF1Module로부터 확장된 RunTimeVF1Module에 bind/unbind 메소드를 정의한다. FeatureConfigurator는 RTconfigure 메소드를 통해서 첫 번째 파라미터로 주어진 특성 구현 모듈(RunTimeVF1Module)에 두 번째 파라미터에 해당되는 구현 코드(Imp1)를 동적으로 로드하여 결합시킨다.

이와 같은 방식의 단점은 실행 시점에 결합될 특성 구현 모듈을 위해 필요한 모듈들(VF1Module, RunTimeVF1Module)을 컴파일 시점 혹은 로드 시점에 제품에 미리 포함시켜야 한다는 것이다. 하지만 일단 결합이 된 이후에는 수행 오버헤드 없이 작동될 수 있다.

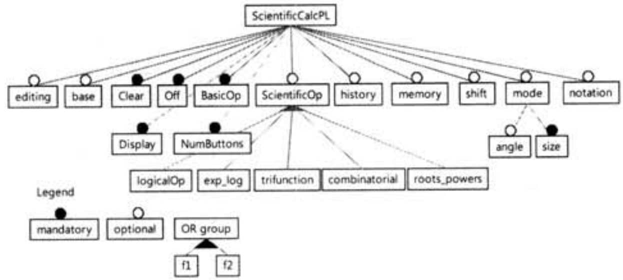
4. 사례 연구

본 논문에서 제안된 특성 구현 패턴의 적용가능성을 검증하기 위해서, 공학용 계산기 프로그램에 대한 사례연구를 수행하였다.

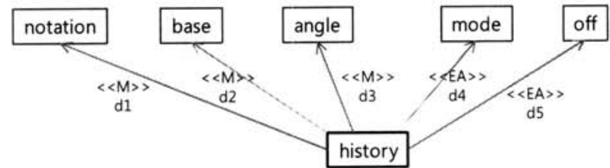
4.1 특성 지향 분석

특성 지향 제품계열 공학 방법은 특성 관점에서 제품계열을 분석하는 것으로 시작한다. 앞서 설명했듯이, 특성 관점의 제품계열 분석에는 가변성 분석, 의존성 분석, 결합 시점 분석 등이 있다. 다음은 공학용 계산기 프로그램에 대한 특성 관점의 제품계열 분석을 수행한 결과이다.

(그림 9)는 공학용 계산기 제품계열의 공통성과 가변성을 분석하여 모델링한 결과를 보여준다. 공학용 계산기 제품계열의 모든 제품에 공통인 특성(Clear, Off, BasicOp, Display, NumButtons, size)은 필수 특성(mandatory feature)로 표현하고, 제품에 따라 선택적인 가변 특성(editing, base, ScientificOp, history, memory, shift, mode, angle, notation)은 선택 특성



(그림 9) 가변성 분석 모델 (특성 모델)



(그림 10) history 특성과 관련된 특성 간의 수행 의존성

(optional feature)으로 표현하였다. 또한 제품에 따라 최소 하나 이상 선택되어야 하는 특성 집합(logicalOp, exp_log, trifunction, combinatorial, roots_powers)을 OR 그룹 특성(OR group feature)으로 모델링 하였다.

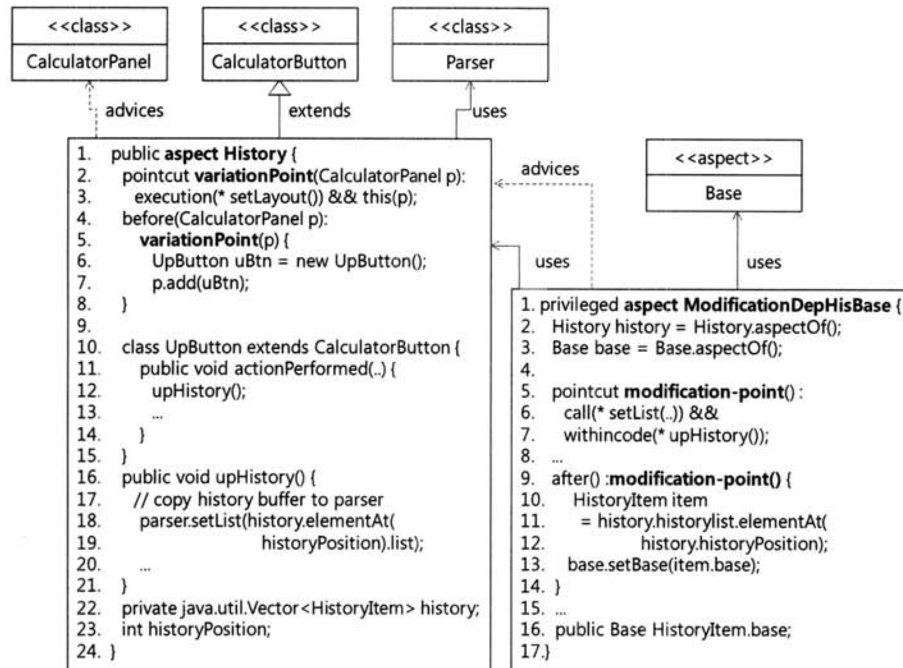
한편, (그림 9)에 나타난 특성들은 서로 독립적이기 보다는 특성 간에 수행의존 관계를 가지고 있다. (그림 10)은 history 특성과 관련된 의존 관계만 요약한 것이다. (그림 10)에서 보는 바와 같이, history 특성(이전에 계산된 수식을 리스트로 관리하며, 리스트에서 선택된 수식을 현재 수식으로 지정하는 기능)은 notation(수식 및 계산 결과의 표현 방식을 변경하는 기능), base(수식의 기저를 변경하는 기능), angle(각도 단위를 변경하는 기능) 특성에 대해서 변경 의존성을 가진다. 가령, base 특성은 현재 수식의 기저를 변경시킬 수 있으므로, base 특성이 history 특성과 같은 제품에 선택된다면, 변경된 수식의 기저 정보도 관리할 수 있도록 history 특성이 확장되어야 한다. 따라서, history 특성은 base 특성에 대해서 변경 의존성을 가진다고 할 수 있다.

또한, history 특성은 mode 및 off 특성에 대해서 배타 활성화 의존성을 가진다. 가령, mode 특성 과 history 특성이 공통의 공유자원(display panel)을 동시에 사용할 수 없기 때문에 한 순간에 하나의 특성만이 활성화 되게 하기 위해서 history 특성이 mode 특성에 대해 배타 의존성을 갖게 된 것이다.

특성 결합 시점 분석은 특성을 언제 어떻게 제품에 결합시킬 지를 분석하는 것이다. (그림 11)은 각 특성 별로 언제 다른 특성 구현 모듈과 연결이 이루어지고, 언제 사용 가능하게 되는 지에 대한 결정을 내린 결과를 보여준다. 또한, 특성과 관련된 의존성도 특성의 제품 결합 시점에 따라 결정이 된다. 예를 들면, history 특성이 실행시점에 연결과 가용성이 결정되므로, 이와 관련된 의존성 (d1~d5)도 모두 실행 시에 연결과 가용성이 결정되어야 한다. 이와 같은 결과는 마케팅 분석결과를 바탕으로 내려진 것으로, 본 논문에서는 이러한 결정이 어떻게 내려졌는지에 초점을 두기 보다

특성	history	notation	base	sngle	mode	off	d1	d2	d3	d4	d5
연결	실행	컴파일	컴파일	로드	로드	컴파일	실행	실행	실행	실행	실행
가용성	실행	실행	로드	실행	로드	컴파일	실행	실행	실행	실행	실행

(그림 11) 특성 및 의존성 결합 시점 분석



(그림 12) history 특성 및 의존성 분리

는 이러한 결정이 핵심 자산 개발에 어떻게 영향을 주는 지에 초점을 두어 설명한다.

4.2 애스팩트 패턴을 이용한 구현

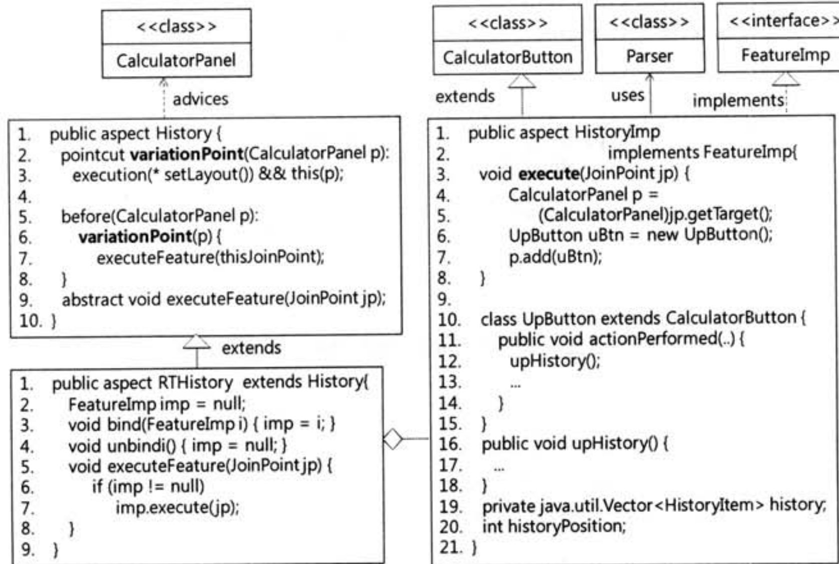
이 절에서는 앞에서 분석된 특성 관점의 분석 결과(가변성, 의존성, 결합시점)가 3장에서 제안된 애스팩트 구현 패턴을 이용하여 어떻게 핵심자산으로 구현되는 가를 설명한다. 특별히, 설명의 간결함을 위해서 history 특성 관점에서 가변특성의 분리, 가변 의존성의 분리, 결합 시점 정보의 분리를 설명한다.

history 특성은 사용자가 계산기 사용자 인터페이스의 위-화살표 혹은 아래-화살표 버튼을 누를 때마다 과거의 입력된 수식을 화면에 보여주면서, 화면에 나타난 수식을 현재 입력된 수식으로 설정하는 기능을 가진다. 이 특성을 구현하기 위해서는 먼저 사용자 인터페이스에 화살표 버튼을 추가해야 하고, 버튼이 눌러졌을 때, 현재 입력된 수식을 리스트에 저장하고, 리스트에 저장된 수식을 계산기의 현재 수식으로 설정하는 일을 버튼의 액션리스너(ActionListener)로 정의해야 한다.

(그림 12)의 왼쪽 부분에서 보는 바와 같이, history 특성을 구현하는 History 애스팩트는 위-화살표 버튼를 나타내는 UpButton을 내부 클래스로 정의하고(10-15줄), 이 버튼의 눌러졌을 때 수행될 행동을 upHistory() 함수로 정의하였다 (16-21줄). 그리고, 이 버튼을 계산기 화면에 추가하기 위해서는 공통 모듈인 CalculatorPanel 클래스에 이와 관련된 코드를 삽입해야 하기 때문에, variationPoint 포인트컷을 이용해서 삽입될 위치를 정의하고(2-3줄), 지정된 위치에 버튼을 생성하고 이를 화면에 추가하는 코드를 삽입하기 위해서 before어드바이스를 이용하였다(4-8줄).

앞서 설명했듯이, history 특성은 base 특성에 대해서 변경 의존성을 지닌다. 이것의 의미는 history 특성 구현 모듈이 base 특성에 의해서 확장 및 변경되는 부분이 있다는 것을 의미한다. base 특성도 가변특성이므로, 이와 관련된 변경 의존성이 History 애스팩트에 포함되어 있다면 History 애스팩트가 base 특성의 선택에 따라서 변경되어야 한다. 따라서, (그림 12)의 오른쪽 부분은 이 변경 의존성을 History 애스팩트와는 분리된 ModificationDepHisBase 애스팩트로 구현된 것을 보여준다. 3.1.1절의 변경 의존성 구현 패턴에 따라서, History 애스팩트의 변경 지점을 modification-point 포인트컷으로 정의하고(5-7 줄), 이 지점에서 추가 될 코드를 after 어드바이스로 정의하였다(9-14 줄).

이상과 같이, 가변특성 및 가변 의존성을 독립 애스팩트 모듈로 구현함으로써, 가변특성 선택에 따른 핵심자산의 적응을 해당 구현 모듈의 선택 혹은 비선택으로 간단히 처리



(그림 13) history 특성의 실행 시점 결합

할 수 있게 된다.

(그림 11)에서 보는 바와 같이 history 특성은 다른 특성 구현 모듈과의 연결 및 history 특성 구현 모듈의 사용 가능성이 모두 실행시간에 결정되어야 한다. 하지만 그림 12의 History 애스팩트는 AspectJ의 직조 메커니즘을 이용하면 컴파일 시점 혹은 로드 시점에 연결 및 가용성 결정이 이루어지는 것을 지원하지만, 실행 시간에 이러한 결정이 이루어지는 경우를 지원하지는 못한다. (그림 13)은 실행 시간에 연결 및 가용성 결정이 이루어질 수 있도록 (그림 12)의 History 애스팩트를 (그림 8)의 패턴을 이용하여 리팩토링한 것이다.

History 애스팩트는 history 특성의 구현 코드를 삽입할 CalculatorPanel 클래스의 가변 위치를 varitionPoint 포인트컷을 이용하여 정의하지만(2-3줄), 실제로 삽입할 코드를 어드바이스 부분(5-8줄)에 정의하는 대신에, 삽입될 코드를 정의할 수 있는 훅-메소드executeFeature()를 정의하였다. History 애스팩트의 하위 애스팩트인 RTHistory 애스팩트는 실행시점에 특성 구현 코드를 History 애스팩트에 결합 혹은 해제시키기 위한 두 가지 메소드 bind()와 unbind()를 제공하고, 결합된 특성 구현 코드를 실행 시킨다(5-8줄). 특성 구현 코드인 HistoryImp는 FeatureImp 인터페이스를 구현한 모듈로서, 실행 시점에 FeatureConfigurator((그림 8) 참조)를 통해서 동적으로 선택되고 로드 된 것이다.

이와 같은 패턴을 이용하면, History와 RTHistory 애스팩트는 늦어도 CalculatorPanel클래스 로드 시점에 결합이 이루어져야 하지만, HistoryImp는 실행시점에 사용자에 의해서 history 특성이 선택된 시점에 제품에 결합이 되게 된다.

한편, history 특성을 실행 시점에 제품과 결합하게 한다면, history 특성과 관련된 의존성도 실행 시점에 제품과 결합되도록 구현되어야 한다. 이것도 (그림 13)에서 예시된 방법과 동일하게 HistoryImp의 변경 지점을 modification-point 포인트컷으로 정의하고, 이 지점에 추가될 코드를 직접 어드바이스에 정의하는 대신에 훅-메소드를 통해서 동적으로 결합될 수 있도록 만들면 된다.

4.3 사례 연구 결과 및 토의

본 절에서는 사례 연구의 수행결과를 바탕으로 제안된 방법을 핵심자산의 유연성, 복잡도 및 도구 지원, 한계점 관점에서 평가하고 토의한다.

4.3.1 핵심자산의 유연성

사례연구를 통해 구현된 공학용 계산기 제품계열은 총 13개의 가변특성으로 구성되어 있다. 이론적으로 계산하면, 총 2^{13}개의 계산기 제품을 구성할 수 있다.

이와 같이 다양한 계산기 제품을 가변특성의 선택에 따라 쉽게 구성하기 위해서는 가변특성의 선택에 따라 해당되는 특성 구현 모듈의 포함과 삭제가 간단히 이루어지도록 핵심자산이 구현되어야 한다. 본 논문에서는 각각의 가변특성을 구현한 모듈이 다른 가변특성을 구현한 모듈에 독립적이 되도록 하기 위해서, 가변특성 간의 의존성을 가변특성 구현 모듈로부터 명확히 분리시키는 방법을 사용하였다. 4.2 절에서 예시한 바와 같이 history 특성을 구현한 History 애스팩트 모듈은 history 특성의 핵심 기능만을 구현하였고, history 특성과 의존성은 독립 모듈로 분리시켰다.

또한, 특성 간의 의존성뿐만 아니라, 특성의 제품 결합 시점에 대한 정보도 특성 구현 모듈로부터 분리시킴으로써, 핵심자산의 유연성을 한층 더 높였다. 따라서, 가변특성의 제품결합 시점이 제품에 따라 다른 경우에도 특성 구현 모듈의 변화 없이 이러한 가변성을 쉽게 수용할 수 있다. 예를 들면, (그림 13)에서는 history 특성이 실행시점에 결합되는 방식을 나타내었는데, 이를 로드 시점이나 컴파일 시점에 결합되는 방식으로 교체하기 위해서는 RTHistory 애스팩트 대신에 로드 시점이나 컴파일 시점에 결합되는 방식을 지원하는 LTHistory 애스팩트나 CTHistory 애스팩트로 대체하면 된다((그림 14)참조).

```
1.  public aspect LTHistory  extends History{
2.     FeatureImp imp = null;
3.
4.     LTHistory() {
5.        if (FeatureConfigure.
6.              canConfigure("History"))
7.              imp = HistoryImp.aspectOf();
8.     }
9.     void executeFeature(JoinPoint jp) {
10.        if (imp != null)
11.            imp.execute(jp);
12.    }
13. }
```

```
1.  public aspect CTHistory  extends History{
2.    void executeFeature(JoinPoint jp) {
3.         HistoryImp.aspectOf();.execute(jp);
4.     }
5.  }
```

(그림 14) history 특성의 로드 시점 및 컴파일 시점 제품 결합 지원

4.3.2 복잡도 및 도구 지원

사례연구를 통해 예시된 공학용 계산기 제품계열은 총 13개의 가변특성과 6개의 필수특성으로 구성되었고, 특성 간의 의존성의 수는 총 56개로서 변경 의존성, 배타 의존성, 종속 의존성, 순차 의존성의 개수가 각각 17, 18, 18, 3 개였다.

구현된 공학용 계산기 제품계열의 핵심자산은 총 6.5K LOC(lines of code)의 크기로 공통 모듈, 가변특성 구현 모듈, 의존성 구현 모듈로 구분된다. 공통 모듈은2.5K LOC 크기로52 개의 클래스로 구성되었으며, 가변특성 구현 모듈은 3.1K LOC의 크기로 95개의 클래스와 99개의 애스팩트로 구성되었다 (여기에는 특성 결합 시점 구현과 관련된 모듈도 포함되었음). 또한, 의존성 구현 모듈은 0.9K의 크기로 3개의 클래스와 56개의 애스팩트로 구성되었다.

이 결과가 의미하는 바는 간단하지 않은 중간 크기의 제품계열을 구현했을 때 조차도 핵심자산으로 구현된 클래스 및 애스팩트 모듈의 수가 적지 않다는 것이다. 만약 제품계열의 크기가 커지게 되면, 더 많은 수의 모듈을 관리해야 할 것이다. 이러한 복잡도 문제를 해결하기 위해서는 자동화된 도구의 지원이 필요할 것이다. 즉, 특성 분석 정보 (가변성, 의존성, 결합시점 정보)와 구현 모듈과의 명시적인 대응을 바탕으로 특성 선택에 따라 해당되는 구현 모듈의 선택과 배제를 자동화할 수 있는 도구가 필요하다.

본 논문에서는 특성 지향의 제품계열 공학을 효과적으로 지원하기 위한 핵심자산의 구현 패턴에 초점을 두어 설명하므로, 지원 도구에 대한 이슈는 다루지 않지만, 제안된 패턴을 이용하여 개발된 핵심자산을 바탕으로 효율적인 제품 생산을 위해서는 자동화된 지원도구가 필수적인 요소이다.

4.3.3 한계점

본 논문의 사례연구를 통해서, 특성간의 의존성 및 결합시점 정보를 모두 쉽게 분리해 낼 수 있음을 보였지만, 본 논문에서 제안된 애스팩트 구현 패턴이 가지고 있는 한계는 다음과 같다.

- *핵심자산의 재사용성 및 적응성의 초점*: 본 논문에서 제안한 애스팩트 구현 패턴은 특성의 선택에 따라 핵심자산이 쉽게 재사용되고 적응될 수 있도록 하는데 목표를 두었다. 만약, 핵심자산의 재사용성 및 적응성 대신에 신뢰성 및 성능을 더 중요하게 생각한다면, 제안된 애스팩트 패턴이 적합하지 않은 경우가 발생할 수도 있다.
- *기능 특성간의 의존성에만 초점*: 본 논문에서는 기능 특성 관점에서 제품계열의 특성 분석하고 이들 간의 수행 의존성에만 초점을 두었다. 제품 특성은 기능적인 특성도 있지만 성능 및 보안 등 비기능적인 특성도 포함될 수 있다. 따라서 비기능 특성 간에 혹은 기능 특성과 비기능 특성 간에는 다른 종류의 의존성 및 상호작용이 존재할 수 있다.
- *실행시점 직조 메커니즘*: AspectJ는 컴파일 시점 및 로드 시점 직조 메커니즘은 제공하지만, 실행 시점 직조 메커니즘은 제공하기 않고 있다. 따라서 본 논문에서 제안된 실행 시점 결합 패턴은 순수한 실행 시점 직조 메커니즘 이라기 보다는 이를 모사한 기법이라는 점에서 약간의 오버헤드가 존재한다. 가령, 그림 8의 VF1Module과 RunTimeVF1Module은 실질적인 특성 구현 모듈인 Impl1을 CFModule1과 결합시키기 위한 중간자 역할을 하며, 실행 시점 전인, 로드 시점이나 컴파일 시점에 CFModule1과 결합되어야 한다. 즉, 실행 시점에 CFModule1과 결합될 Impl1을 위해 그 전에 관련된 코드가 CFModule1과 결합되어야 하는 오버헤드가 발생한다.

5. 관련 연구

가변특성의 구현을 AOP를 이용하여 모듈화 한다는 아이디어는 Griss[8] 에 의해 특성 기반, 애스팩트 지향의 제품계열공학이란 개념적인 프레임워크로 처음으로 제안되었지만, 구체적인 방법 및 사례연구는 제시하지 못하였다. 그 이후로, AOP 언어 메커니즘을 이용하여 가변 특성의 횡단성 문제를 해결하기 위한 노력으로 다수의 연구[1, 12, 16, 17] 들이 있어 왔다. 한편, AOP와 유사하지만 단계적 상세화(stepwise refinement) 기법을 바탕으로 정의된 특성 지향 프로그래밍(Feature-Oriented Programming)을 통해 특성의 횡단성 문제를 해결하는 연구 [3]도 있다. 하지만, 이러한 연구들은 특성 구현의 모듈화에 초점을 두었지만, 특성 간의 의존성 및 특성 결합 시점의 다양성 문제를 고려하지는 않았다.

한편, 특성 간의 의존성을 명시적으로는 언급하지 않았지만, 특성 구현 모듈이 기본모듈(base module)과 유도모듈(derivative module)의 두 부분으로 구분된다고 이해한 연구

들이 있어왔다. Godil et al.[7], Lui et al.[12] 등은 AOP 기법을 이용하여 기존 레거시 코드를 특성 관점에서 리팩토링하는 과정에서 하나의 특성을 구현하는 모듈이 크게 특성의 핵심 기능을 구현한 기본모듈과, 다른 특성과의 상호작용을 통해서 유도된 모듈로 구성됨을 파악했다. 유도 모듈은 본 논문에서 설명한 특성 간의 의존성을 구현한 부분에 해당되지만, 이들은 특성 간의 의존성과 유도 모듈과의 관계를 명시적으로 설명하지 않았다. 본 논문이 가지는 공헌은 특성 간의 의존성을 사전에 미리 분석하고 분석된 의존성을 특성 구현 모듈로부터 명확히 분리시킬 수 있는 애스팩트 구현 패턴을 제안한 것이다.

한편, 특성 간의 의존성을 바탕으로 재사용 가능한 컴포넌트 개발에 반영한 연구로는 [15]이 있었다. 하지만 이는 예측된 가변성을 캡슐화하는 객체 지향 컴포넌트 설계 방식을 택하였기 때문에, 제품계열의 범위가 확장되어 미리 예측되지 못한 가변성 고려하는 경우에는 핵심자산의 변경에 커다란 영향 줄 수가 있고, 캡슐화를 위해 사용한 간접화(indirection)가 불필요한 오버헤드를 나타낼 수 있다.

특성 결합 시점의 분석을 통해 동적으로 재구성 가능한 컴포넌트의 설계에 대한 연구로 [4, 14]이 있었다. [14]는 객체 지향 컴포넌트 설계 방식을 택하였기 때문에, 가변성의 캡슐화를 위해 사용한 간접화(indirection)가 불필요한 오버헤드를 나타낼 수 있다. [4]는 애스팩트 지향 패턴을 제안하였지만, 본 논문에서 제안한 패턴과 비교해 보면, 실행 시점 가용성을 위한 패턴과 유사하고, 실행시점에 연결 및 가용성을 지원하는 패턴은 제공하지 않고 있다.

7. 결 론

특성 지향 제품계열공학의 목표는 제품계열의 분석에서부터 제품계열의 구현에 이르기까지 제품계열공학의 전 과정이 특성 관점에서 일관적으로 이루어지도록 하는 것이다. 본 논문에서는 제품계열공학의 전 과정을 다루기보다는 제품계열의 핵심자산 구현에 있어서, 특성 관점에서 분석된 정보 단위를 핵심자산의 구현 단위로 쉽게 대응시킬 수 있는 패턴 개발에 초점을 두었다. 즉, 제품계열의 분석 결과인 가변특성, 가변특성 간의 의존성, 가변특성의 결합시점 등의 정보를 독립적인 구현 모듈로 개발하기 위해서 AOP의 기법을 이용한 애스팩트 구현 패턴을 제안하였다.

제안된 패턴을 검증하기 위해서 AspectJ를 이용하여 간단하지 않은 공학용 계산기 제품계열의 개발에 적용하였다. 비록 본 논문에서는 AOP 언어 중에 AspectJ를 활용하였지만, AspectJ 보다 더 진보된 메커니즘을 제공하는 프로그래밍 언어 및 개발환경을 이용한다면 제안된 패턴을 보다 향상 시킬 수 있을 것이다. 가령, Prose[17]는 실행 시간 직조 기능을 제공하는 AOP 언어이다. 이를 활용하면 본 논문에서 제한된 실행 시점 연결 및 가용성을 결정하는 패턴이 가지는 약간의 오버헤드를 제거할 수 있을 것이다.

본 논문에서는 특성 지향 제품계열 구현을 위해 고려되어야 하는 여러 이슈를 애스팩트 개념을 활용하여 적용하였지만, 궁극적으로는 AOP 언어 대신에 특성 관점의 개념을 직접적인 프로그래밍 언어 메커니즘으로 제공하는 특성 지향 프로그래밍으로 발전시키기 위한 사전 연구로서 가치가 있다.

참 고 문 헌

[1] V. Alves, P. Matos Jr., L. Cole, P. Borba, G. Ramalho, "Extracting and Evolving Mobile Games Product Lines," *SPLC* 2005, pp.70-81

[2] AspectJ Team, "AspectJ Project," http://www.eclipse.org/aspectj/.

[3] D. Batory, "Feature-Oriented Programming and the AHEAD tool suite," *ICSE 2004*, pp.702-703, 2004.

[4] V. Chakravarthy, J. Regehr, E. Eide, "Edicts: Implementing Features with Flexible Binding Times", AOSD'08, pp. 108-119, 2008.

[5] P. Clements, L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

[6] K. Czarnecki, U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[7] I. Godil and H. A. Jacobsen, "Horizontal Decomposition of Prevalyer," CASCON 2005, pp.83-100.

[8] M. L. Griss, "Implementing Product-Line Features by Composing Aspects," SPLC 2000, pp. 271-288.

[9] M. Griss, J. Favaro, M. d'Alessandro, "Integrating Feature Modeling with the RSEB," ICSR 2009, Victoria, Canada, 1998, pp.76-85

[10] K. C. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," *Technical Report CMU/SEI-90-TR-21*, Software Engineering Institute, Carnegie Mellon University, 1990.

[11] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh, M., "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures," *Annals of Software Engineering*, 5, 1998, pp.143-168.

[12] C. Kastner, S. Apel, and D. Batory, "A Case Study Implementing Features Using AspectJ," *SPLC 2007*, pp. 223-232.

[13] G. Kiczales, J Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, "Aspect-Oriented Programming," ECOOP 1997, 1997.

[14] J. Lee and K. C. Kang, "Feature Binding Analysis for Product Line Component Development", PFE-5, 2003.

[15] K. Lee and K. C. Kang, "Feature Dependency Analysis for Product Line Component Design," *Lecture Notes in Computer Science*, Vol. LNCS 3107, 2004, pp.69-85.

[16] J. Liu, D. Batory, and C. Lengauer, "Feature-Oriented Refactoring of Legacy Applications," *ICSE 2006*, pp.112-121.

[17] Prose, http://prose.ethz.ch/

이 관 우

e-mail : kwlee@hansung.ac.kr

1994년 포항공과대학교 전자계산학과(학사)

1996년 포항공과대학교 컴퓨터공학과 (공학석사)

2003년 포항공과대학교 컴퓨터공학과 (공학박사)

2003년~현 재 한성대학교 정보시스템공학과 조교수

관심분야 : 소프트웨어 제품계열 (Software Product Line), 관점지향 프로그래밍(Aspect-Oriented Programming), 소프트웨어 아키텍처