층위구조 아키텍처의 복구 및 일치성 검사를 위한 프로그램 분석 방법

(A Program Analysis Technique for Recovery of Layered Architecture and Conformance Checking)

박 찬 진 † 홍 의 석 †† 강 유 훈 † 우 치 수 †††
(Chanjin Park) (Euyseok Hong) (Yoohoon Kang) (Chisu Wu)

요 약 층위 구조 아키텍처는 프로그램을 일반성에 따라 분할하는 모듈 구성의 방법이다. 본 논문은 객체지향 프로그램으로부터 층위 구조 아키텍처를 복구하고 아키텍처 문서와의 일치성을 검사하는 방법을 제시한다. 객체지향 프로그램에서의 층위구조 스타일 규칙을 기술하기 위해, 모듈 간 사용 관계에 기반한 모듈들의 부분 순서 집합을 구성하며, 재정의 관계를 통해 모듈 간 층위 관계를 정의한다. 또한, 층위 관계의 의미를 설계 패턴에서의 예를 통해 설명한다. 프로그램으로부터 층위 구조 아키텍처를 복구하기 위한 절차를 기술하며, 복구를 위한 메타 모델을 제시한다. 이를 기반으로 공개 소스 프로젝트를 통해 개발된 소프트웨어의 소스코드들로부터 층위 구조 아키텍처를 복구하고, 복구된 아키텍처로부터 발견된 층위 관계의 의미와 아키텍처 문서와의 불일치 부분들에 대해 논의한다. 검사를 통해 아키텍처 문서와 일치하지 않는 부분들이 발견하였고, 이를 조사한 결과 이들이 층위 구조 아키텍처에서 허용 가능한 예외로 여겨지지만 아키텍트가 이들 부분에 대한 변경을 주의 깊게 관리할 필요가 있다는 것을 지적하였다.

키워드 : 층위구조 아키텍처, 소프트웨어 아키텍처 복구, 소프트웨어 역공학

Abstract Layered Architecture is a kind of module decomposition techniques, which decomposes a program by generality. This paper proposes a layer based method for recovering layered architecture from object-oriented program and checking conformance against architectural document. To specify the rules for layered style in object-oriented program, we define a partially ordered set on modules by module use relationship and module layer relationship by module override relationship. The meaning of module layer relationship is explained with an example from design patterns. Steps to recover layered architecture from program are described and a metamodel for the recovery is proposed. Architecture recovery is performed on source codes from open-source software project, and the implication of parts that do not conform to its architectural document is discussed. As a result of checking, it is pointed out that, although the parts are considered allowable exceptions of layered architecture, their modifications should be controlled carefully.

Key words : Layered Architecture, Software Architecture Recovery, Reverse Engineering

1. 서 론

규모가 큰 소프트웨어 시스템을 개발할 때, 가장 중요한 결정 사항 중 하나는 시스템을 관리 가능한 단위들로 분할하는 방법이다. 여러 아키텍처 스타일들이 시스템을 연결된 컴포넌트들로 분할하기 위해 널리 사용되고 있다. 이러한 스타일들 가운데 층위구조 스타일은 기능의 일반성에 따라 시스템을 분할하며, 분할된 부분들을 단 방향의 사용 관계를 사용하여 연결한다[1]. 층위구조 아키텍처에서는 아래쪽에 배치된 층위가 보다 일반적이고 재사용 가능하며, 보다 위쪽에 배치된 층위는 응용프로그램 구체적이다. 이 아키텍처의 중요한 속성은 단지 위에서 아래로의 사용 관계만이 허용된다는 점이다.

층위구조 스타일로 분할된 시스템은 일반성의 기준에 따라 계층적으로 구성되어 있기 때문에 시스템을 이해

† 학생회원 : 서울대학교 컴퓨터공학부
cjpark@selab.snu.ac.kr
rmaker@selab.snu.ac.kr
†† 정 회 원 : 성신여자대학교 컴퓨터정보학부 교수
hes@sungshin.ac.kr
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
wuchisu@selab.snu.ac.kr
논문접수 : 2005년 1월 26일
심사완료 : 2005년 7월 1일

하기가 용이하다. 또한, 여러 연구들을 통해 변경용이성, 재사용성 및 이식성과 같은 여러 장점들을 가진다는 것이 알려져 있다[1-3]. 하지만, 개발 과정에서 작성되는 설계 산출물이 개발 초기에 결정된 층위구조 스타일을 보존하지 않는다면, 층위구조 스타일이 시스템에 이러한 장점들을 이용하기 어렵다. 더욱이, 초기 아키텍처가 개발 중이거나 개발된 프로그램과 불일치한 경우, 향후 유지보수와 같은 프로그램 이해를 필요로 하는 작업을 어렵게 한다. 아키텍처와 프로그램 간의 일치성을 보장하기 위해서는 시간과 비용이 많이 들기 때문에, 산출물로부터의 아키텍처의 복구 및 아키텍처 일치성 검사를 위한 도구가 필수적이다.

층위 구조 아키텍처는 오랜 역사를 가지고 있으나[4-6], 객체 지향 프로그램에서의 층위 구조를 다루지 않고 있다. 객체 지향 프로그램에서 층위구조 아키텍처는 일종의 모듈 아키텍처로서, 클래스들을 포함하고 있는 패키지들의 부분 순서 집합으로 볼 수 있다. 아키텍트는 개발 과정의 아키텍처의 수립 단계에서 이러한 패키지들의 구성을 결정한다. 이 구성을 통해, 형상 관리 및 프로그램 개발 순서 결정 등의 작업을 수행할 수 있다.

본 논문은 층위구조 아키텍처를 복구하기 위해, 사용 관계에 기반한 모듈들의 부분 순서 집합을 구성한다. 또한, 모듈 재정의 관계에 기반한 층위 관계라는 새로운 타입의 관계를 정의한다. 층위 관계는 층위 간 하향 사용관계를 유지하는 상향 콜백 관계를 표현하기 위해 사용되는 사용 관계의 특수한 경우이다. 이 관계는 소스코드로부터 추출될 수 있으며, 아키텍처 문서와의 비교를 통해 아키텍트의 결정사항이 설계 산출물에 잘 반영되어 있는 지를 확인할 수 있다. 부분 순서화된 모듈 간의 사용 관계와 층위 관계로부터 모듈의 층위 순서 규칙을 정의할 수 있으며, 이에 기반하여 층위 구조 아키텍처를 복구한다. 이러한 접근 방식이 효과적인 지를 보이기 위해, 객체지향 응용프로그램 설계 환경인 Argo-UML[7]의 소스코드로부터 층위 관계를 추출하고, Argo-UML의 아키텍처 문서와 비교하였다.

본 논문의 2장에서는 아키텍처 복구와 관련된 기존 연구에 대해 기술하고, 3장에서 모듈들의 부분 순서 집합과 모듈 간 층위 관계를 명확하게 정의하고 설계 패턴의 예를 통해 층위 관계의 의미에 대해 설명한다. 4장에서는 객체지향 프로그램으로부터 층위 구조 아키텍처를 복구하기 위한 절차들과 아키텍처에 대한 메타모델을 제시한다. 5장에서 ArgoUML 프로그램으로부터 층위 구조 아키텍처를 복구하고, 프로그램의 아키텍처에 대한 일치성을 검사한다. 또한, 추출된 층위 관계의 의미와 아키텍처 문서와 불일치한 관계에 대해 논의한다. 6장은 층위구조 아키텍처 복구 방법 적용에 관련된 토론을 다루며, 결론 및 향후 연구 주제에 대해서는 7장에서 다룬다.

2. 관련 연구

소프트웨어 아키텍처 복구는 구현된 프로그램으로부터 시스템을 설명할 수 있는 큰 그림을 얻는 과정이며, 기존 시스템으로부터 제품 라인을 구성하거나, 최신의 기술과 환경에 맞도록 시스템을 현대화하기 위한 기반 기술이다. 많은 아키텍처 복구 사례들, 기법들, 방법론 및 도구들이 제안되어 왔다[8-11]. 본 논문에서는 층위구조 스타일 규칙을 객체지향 프로그램의 모듈들 간의 사용 관계와 층위 관계를 통해 구체화하며, 특히, 객체지향 소프트웨어 개발에서 약한 결합도를 가지며, 확장성을 부여하는 설계 전략으로 사용되는 새로운 타입의 아키텍처 관계를 제안하고, 이 관계의 의미를 소스코드로부터 복구된 층위구조 아키텍처 상에서 논구하고 아키텍처 문서와의 일치성을 검사한다.

구현 시스템과 일치하는 아키텍처를 얻기 위한 반복적 과정을 제시하는 대표적 복구 모델로 소프트웨어 투영 모델[12-14]이 있다. 관찰자가 기대하는 아키텍처를 기술하며, 아키텍처 상의 모듈과 프로그램 요소 간의 맵핑 관계를 정의하고, 구현 코드 분석을 통해 예상 모듈 관계와 프로그램 요소들로부터 추출된 관계를 비교하여 일치성과 불일치성을 보여준다. 예상 아키텍처에 기술된 관계가 구현 상에 나타나지 않은 경우(Absence)와 예상 아키텍처에 기술되지 않은 관계가 나타난 경우(Divergence)를 반영하는 새로운 예상 아키텍처 모델을 제시하는 과정을 반복적으로 수행함으로써 구현과 일치(Convergence)하는 아키텍처를 구한다. 기존 투영 모델 관련 연구 사례들은 절차적 언어를 대상으로 하므로, 호출에 의한 함수 집합 간의 의존 관계를 아키텍처 관계로 사용하고 있으며, 복구하고자 하는 아키텍처에 대한 규칙은 관찰자의 결정에 따른다. 본 논문에서는 아키텍처 구성 규칙으로 층위구조 스타일을 사용하며, 층위 관계는 객체지향 프로그램으로부터 아키텍처를 복구할 때 중요하게 다루어질 필요가 있음을 보인다.

[15-17]에서는 아키텍처 복구 과정의 절차 및 가이드라인을 제시하고 있다. [15]는 제품 라인의 구성, 기존 시스템에 대한 평가, 시스템 현대화와 같은 복구된 아키텍처의 응용 환경에 따른 품질 요소 주도 아키텍처 복구 과정을 제시하고 있다. 이 연구에서는 아키텍처 복구뿐만 아니라, 복구된 아키텍처를 재사용성, 변경용이성 및 성능과 같은 품질 요소를 기준으로 평가, 분석하는 복구 모델을 기술하고 있다. [16]은 조직에 아키텍처 복구를 적용하는 경험으로부터 도출된 공통적인 문제와 해결책들에 대한 아키텍처 복구 패턴들을 제안하고 있

다. [17]은 아키텍처의 설계 및 복구, 유지보수를 포함하는 아키텍처의 라이프사이클에 걸쳐 개발자에 도움을 주는 개발환경 ART를 제안하고 있다. 이들 방법론들이 복구 대상 아키텍처의 특성을 고려하지 않은 일반적인 복구의 절차를 다루고 있는 반면, 본 논문에서는 프로그램으로부터 층위 구조 아키텍처를 복구하기 위한 절차를 기술한다. 층위구조 아키텍처 복구를 위해 소스모델 추출, 모듈 맵핑, 모듈 층위화, 층위 구조 아키텍처 복구 및 아키텍처 문서와의 일치성 검사의 절차를 따른다.

3. 모듈들의 부분 순서 집합과 층위 관계 정의

이 장에서는 객체지향 프로그램에서의 층위 구조 스타일 규칙을 구체화하기 위해, 모듈들 간의 사용 관계와 재정의 관계를 정의한다. 모듈들 간의 사용 관계로부터 모듈들의 부분 순서 집합을 구성하며, 재정의 관계를 통해 모듈 간 층위 관계를 정의한다. 객체지향 프로그램에서 각 층위는 클래스들을 내부에 가지고 있는 여러 패키지들로 구성된다. 상 층위의 클래스들은 하 층위의 클래스들을 사용할 수 있으나, 그 반대의 사용은 허용되지 않는다.

콜백은 하향식 단 방향 커플링을 보존하면서 상향식 커뮤니케이션을 위해 사용되며, 하 층위는 상 층위가 하 층위에 수행 시에 등록한 함수를 간접적으로 사용한다[2,18]. 이러한 콜백은 수행 시에 제어의 흐름을 역전시키며, 하 층위에서 상 층위로의 호출이 이루어진다. 이러한 특성 때문에 콜백은 층위 구조 아키텍처에서 업콜(up-call)로 불린다[19].

객체지향 프로그램에서 콜백은 동적 바인딩을 통해 구현된다. 상 층위에 있는 클래스가 하나 이상의 하 층위 클래스들을 상속하고, 이 들의 메소드를 재정의한다면, 상 층위에서 이 클래스의 객체를 생성하고 레퍼런스를 하 층위에 넘겨주면 하 층위는 수행 시에 상 층위 타입의 객체에 대해 재정의된 메소드를 호출할 수 있다. 콜백 관계는 층위 간에 중요한 커뮤니케이션 경로를 보여주기 때문에 층위구조 아키텍처에서 이 관계가 명시적으로 표현되어야 만 한다. 혼란을 피하기 위해 본 논문에서는 상 층위는 구체 층위, 하 층위는 추상 층위로 부르기로 한다.

3.1 사용 관계에 의한 모듈 부분 순서 집합과 층위 관계

모듈 간의 관계는 모듈에 속한 클래스들 간의 관계로부터 유도되므로, 먼저 클래스들 간의 관계를 정의한다. 클래스 들 간의 관계는 의존, 상속 및 연관 관계로 분류할 수 있다. 논문에서 이들 클래스 간 관계들을 재정의 관계와 사용 관계의 두 관계로 재 분류한다. 클래스 간 재정의 관계는 일종의 상속 관계로서, 자식 클래스가 부모 클래스의 하나 이상의 동일한 시그내처를 가진 메소드를 재정의하는 관계이다. 클래스 간 사용 관계는 하나의 클래스가 다른 클래스를 메소드의 인자나 클래스의 속성 또는 부모 클래스 등 어떤 형태로든 참조하는 것을 의미한다.

정의 1 (시스템, 모듈, 클래스)

객체지향 시스템은 클래스들의 집합 C로 구성되어 있다. 모듈은 시스템을 분할하는 클래스들의 집합이다.

$S = \{M_1, M_2, \cdots, M_n\}$

$M_i, M_j \subseteq C, \cup M_i = C, M_i \cap M_j = \emptyset$ *if* $i \neq j, 1 \leq i,j \leq n$.

정의 2 (클래스 들 간 사용관계)

각 클래스 c, $d \in C$에 대해, c에서 d를 속성, 인자 및 로컬 변수를 선언하기 위해 사용하거나, 부모클래스를 기술하기 위해 사용하면 use(c, d)이다.

정의 3 (모듈 간의 사용 관계)

모든 모듈 M_i, M_j에 대해,

use-direct(M_i, M_j) iff $\exists c \in M_i, \exists d \in M_j \cdot$ *use(c, d) or* $M_i = M_j$

use(M_i, M_j) iff use-direct(M_i, M_j) or ($\exists M' \subseteq C \cdot$ *use-direct(M_i, M') and use(M', M_j)).*

정리 1 (모듈 간 사용 관계의 이행성)

모든 모듈 M_i, M_j, M_k에 대해, use(M_i, M_j)이고 use(M_j, M_k)이면, use(M_i, M_k)이다.

증명) use(M_i, M_j)이면 use-direct($M_{i,l}$, $M_{i,l+1}$), $0 \leq l < m$을 만족하는 유한한 체인 ($M_{i,0} = M_i$, $M_{i,1}$, $\cdots$, $M_{i,m} = M_j$)이 존재한다. 마찬가지로, use(M_j, M_k)에 대해서도, ($M_{j,0} = M_j$, $M_{j,1}$, $\cdots$, $M_{j,n} = M_k$)가 존재한다. use-direct(M_j, M_j)이므로, ($M_{i,0} = M_i$, $M_{i,1}$, $\cdots$, $M_{i,m} = M_j = M_{j,0}$, $M_{j,1}$, $\cdots$, $M_{j,n} = M_k$)인 체인을 구성할 수 있고, 따라서, use(M_i, M_j)이고 use(M_j, M_k)이면, use(M_i, M_k)이다.

두 모듈 간의 실제 사용 관계(use-direct)는 포함된 클래스들 간의 사용관계로부터 유도되며, 모듈은 자신을 사용하는 것으로 정의한다. [1]에서는, 클래스 간 사용 관계 *use(c, d)*는 c가 자신의 요구를 만족시키기 위해서 올바르게 동작하는 d에 의존하는 관계로 정의하고 있다. 모듈 간의 사용 관계를 두 모듈 간의 실제 사용 관계(use-direct)에 기반해서 재귀적으로 정의하며, 코드 상의 실제 사용 관계라기 보다는 설계 관점의 모듈 간 의존성을 표현한다.

클래스들 간에 순환을 이루는 사용관계가 가능하지만, 이는 각각의 클래스를 서로 독립적으로 변경하기 어렵게 만든다. 특히, 모듈 간에 사용 관계가 순환을 이루면, 둘 다를 완성할 때까지 어느 것도 사용될 수 없다. 다시 말해, 모듈들을 구현하기 위한 작업의 우선 순위를 결정하기 어려워진다. 층위 구조 아키텍처는 모듈들을 단방

향 사용 관계로 구성하므로, 모듈들 간 순환적 사용관계는 층위의 순서를 정할 수 없게 한다. 본 논문에서는 두 모듈이 순환적 사용 관계를 가지게 되면, 모듈 내 클래스 간 사용 관계 삭제를 통해 순환 관계를 해소하거나, 순환 관계를 가지는 모듈들을 하나의 모듈로 병합한다. 삭제와 병합을 통해 만들어지는 새로운 모듈 분할은 부분 순서 집합이 된다. 이 사용 관계에 의한 부분 순서 집합은 층위 구조 아키텍처가 가지는 층위들 간의 순서화(층위화)를 위해 사용된다.

정의 4 (모듈 재분할을 위한 사상)

모듈들의 집합 S를 새로운 모듈들의 집합 $\dot{S}$으로 맵핑하는 사상 f를 정의한다.

$f : S \rightarrow \dot{S}$

$f(M) = \cup Mc$, *where,* $Mc \in S$ *and* $use(M, Mc) \wedge use(Mc, M)$.

이 사상 f를 통해 새로운 재분할된 시스템 $\dot{S}$은 다음과 같다.

$\dot{S} = \{f(M_1), f(M_2), \cdots, f(M_n)\} = \{\dot{M}_1, \dot{M}_2, \cdots, \dot{M}_m\}$

정리 2 (재분할된 모듈 간 사용 관계 정의에 의한 부분 순서 집합)

다음과 같이 정의된 재분할된 모듈 간의 사용 관계는 부분 순서 집합을 구성한다.

$use(\dot{M}_i, \dot{M}_j)$ *iff* $\exists M_i, M_j \in S$ *s.t.* $f(M_i) = \dot{M}_i$, $f(M_j) = \dot{M}_j$, $use(M_i, M_j)$.

증명) *(재귀성)* $f(M_i) = \dot{M}_i$*인 M_i에 대해,* $use(M_i, M_i)$*이므로* $use(\dot{M}_i, \dot{M}_i)$*이다.*

(반대칭성) $use(\dot{M}_i, \dot{M}_j)$*이고* $use(\dot{M}_j, \dot{M}_k)$*이면, 재분할된 모듈 간 사용관계 정의에 의해,* $f(M_i) = \dot{M}_i$, $f(M_j) = \dot{M}_j$, $use(M_i, M_j)$*인 $M_i, M_j \in S$가 존재하고,* $f(M_j') = \dot{M}_j$, $f(M_i') = \dot{M}_i$, $use(M_j', M_i')$*인 $M_j', M_i' \in S$가 존재한다.*

$f(M_i) = f(M_i') = \dot{M}_i$*이고,* $f(M_j) = f(M_j') = \dot{M}_j$*이므로, 사상 f의 성질에 의해* $use(M_i, M_i')$*이고* $use(M_j', M_j)$*이다.*

따라서, S 상의 모듈 사용관계의 이행성에 따라, $use(M_j, M_j') \wedge use(M_j', M_i) \Rightarrow use(M_j, M_i)$*이 성립된다.*

$use(M_i, M_j)$*이고* $use(M_j, M_i)$*이므로,* $f(M_i) = f(M_j)$ *즉,* $\dot{M}_i = \dot{M}_j$.

(이행성) $use(\dot{M}_i, \dot{M}_j)$*이고* $use(\dot{M}_j, \dot{M}_k)$*이면, 재분할된 모듈 간 사용관계 정의에 의해,* $f(M_i) = \dot{M}_i$, $f(M_j) = \dot{M}_j$, $use(M_i, M_j)$*인 $M_i, M_j \in S$가 존재하고,* $f(M_j') = \dot{M}_j$, $f(M_k) = \dot{M}_k$, $use(M_j', M_k)$*인 $M_j', Mk \in S$가 존재한다.*

$f(M_j) = f(M_j') = \dot{M}_j$*이므로, 사상 f의 성질에 의해* $use(M_j, M_j')$*이다.*

S 상의 모듈 사용관계의 이행성에 따라, $use(M_i, M_j) \wedge use(M_j, M_j') \wedge use(M_j', M_k) \Rightarrow use(M_i, M_k)$ *따라서, 재분할된 모듈 간 사용 관계의 정의에 따라,* $use(\dot{M}_i, \dot{M}_k)$*이다.*

층위 간 상향 콜백을 명시적으로 표현하기 위해 모듈 간 층위 관계를 정의한다. 클래스 간 재정의 관계로부터 모듈 간 층위 관계를 정의한다. 클래스 간 재정의 관계는 부모 클래스와 자식 클래스들 간에 모두 정의된 메소드들 사이의 관계에서 유도된다. 재정의 관계가 성립하기 위해서는 상속 관계를 가져야 하므로, 클래스 간 재정의 관계는 사용 관계의 특수한 경우라고 볼 수 있다. 클래스 간 재정의 관계는 모듈 간 재정의 관계를 정의하기 위해 확장된다.

정의 6 (클래스 간 재정의 관계)

각 클래스 $c, d \in C$에 대해, 클래스 c가 클래스 d를 상속하고, 부모 클래스 d의 하나 이상의 메소드들을 재정의하면 $override(c, d)$*이다.*

override의 정의에 의해, $use(c, d)$ *if* $override(c, d)$*이다.*

정의 7 (모듈 간 재정의 관계)

각 모듈 M_i, M_j에 대해,

$override(M_i, M_j)$ *iff* $\exists c \in M_i, c' \in M_j, M_i \neq M_j \cdot override(c, c')$

재분할된 모듈 $\dot{M}_i, \dot{M}_j$ 간의 재정의 관계는 다음과 같이 정의된다:

$override(\dot{M}_i, \dot{M}_j)$ *iff* $override(M_i, M_j)$, $f(M_i) = \dot{M}_i$, $f(M_j) = \dot{M}_j$, $\dot{M}_i \neq \dot{M}_j$.

$\dot{M}_i = \dot{M}_j$*인 경우, 즉 모듈 내의 재정의 관계는 자신으로의 사용관계(use)로 간주한다.*

정의 8 (모듈 간 층위 관계)

각 모듈 M_i, M2에 대해

$layer(M_i, M_j)$ *iff* $override(M_i, M_j) \wedge \sim use(M_j, M_i)$.

$layer(\dot{M}_i, \dot{M}_j)$ *iff* $override(\dot{M}_i, \dot{M}_j)$.

모듈 간 층위 관계는 재정의 관계를 가지나 역방향 사용 관계를 가지지 않는 모듈들 간의 관계로 정의한다. 재분할된 모듈 간 관계는 부분 순서 집합을 이루므로, 층위 관계 정의에서 재정의 관계가 있을 때 역방향 사용 관계가 성립할 수 없다. 두 집합 간에 층위 관계가 성립한다는 것은 추상 층위 상의 클래스가 구체 층위 상의 클래스를 객체 지향의 중요한 요소 중의 하나인 동적 바인딩을 통해 간접적으로 사용하게 되는 것을 의미한다. 정의에서 $\dot{M}_j$는 $\dot{M}_i$에 대해 추상 층위를 구성할 수 있고, 메소드 재정의에 의한 콜백을 통해 $\dot{M}_i$을 사용한다.

모듈 간의 층위 구분은 사용 관계에 기반한 모듈들의 부분 순서 집합에 기반하며, 특정 사용 관계가 층위 관

계라면 이는 프로그램 상에 표현된 중요한 아키텍처 결정 사항을 나타내는 것으로 층위구조 아키텍처에서 명시적으로 표현되어야 한다. 본 논문에서 층위 관계는 모듈 간 관계로부터 물리적 층위 구분을 결정할 수 있는 관계를 의미한다. 모듈 간의 사용 관계가 층위 구분 결정에 사용될 수 있으나 임의적인 구분이 가능하므로, 사용 관계를 통한 층위 구분을 정당화하기 위해서는 층위 구분과 아키텍처 결정사항과의 연관성을 따져야 한다.

정의 9 (사용 관계와 층위 관계에 기반한 모듈의 순서함수 특성)

각 모듈 Mi, Mj에 대해,

$$rank(\dot{M}) : \dot{S} \rightarrow Int$$

$$rank(\dot{M}_i) \geq rank(\dot{M}_j) \text{ if } use(\dot{M}_i, \dot{M}_j)$$

$$rank(\dot{M}_i) > rank(\dot{M}_j) \text{ if } layer(\dot{M}_i, \dot{M}_j).$$

이러한 성질을 만족하는 rank 함수를 정의하면, 프로그램의 모듈 구성을 단방향 사용 관계를 만족시키는 순서화된 층위 집합으로 볼 수 있으며, 동일한 rank 값을 가지는 모듈들이 하나의 층위를 구성한다. 물리적 층위 구분[1]은 정의 9의 층위 관계에 의한 모듈 순서 함수에 의한 제한 조건을 지키는 구분이다. 이 후 아키텍처 복구 과정에서는 관찰자의 프로그램 지식과 아키텍처 문서 상의 층위 정보에 의존하는 개념적 층위 구분을 사용한다. 개념적 층위 구분은 프로그램으로부터 추출된 물리적 층위 구분과 아키텍처 정보를 기반으로 관찰자에 의해 결정되는 층위화 결과이다.

3.2 층위 관계를 통한 층위화

그림 1은 층위 관계를 통한 층위화 개념을 보여주고 있다. B' 클래스를 통해 B에 접근함으로써, {A, B'}와 {B} 간에 층위를 부여할 수 있다. 여기에서, 추상 층위의 클래스 A는 B'를 사용하며, 구체 층위의 클래스 B는 B'의 메소드들을 재정의 하고 있다. 구체 층위에 있는 클래스들은 추상 층위의 클래스들을 사용할 수 있으나, 그 반대는 허용되지 않는다. 이는 구체 층위가 추상 층위 위에 구현되기 때문이다. 점선 화살표는 추상 층위가 구체층위를 콜백을 통해 사용하는 방식을 보여준다. 구체 층위 상의 클래스들은 추상 층위의 어떤 클래스에 의해서도 직접 사용되지 않으므로, 구체 층위 상의 클래스의 변경은 추상 층위로 전파되지 않는다. 더욱이, 구체 층위가 추상 층위로의 직접적인 사용관계를 전혀 가지지 않고, 추상 층위와 층위 관계로만 연결되어 있다면, 추상 층위 역시 구체 층위에 무관하게 변경될 수 있다. 이는 구체 층위의 클래스들이 추상 층위의 구현에 의존하는 것이 아니라, 상속하는 클래스의 이름과 재정의된 메소드의 시그내처에만 의존하기 때문이다.

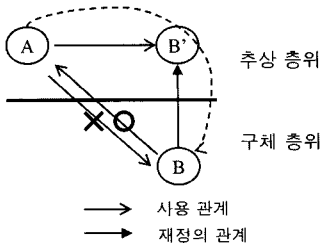

그림 1 층위 관계를 통한 층위화

3.3 층위 관계와 설계 패턴

이 절에서는 층위 관계의 보존이 설계 패턴 사용의 장점을 살리기 위한 중요한 조건이 됨을 논구한다. 층위 관계는 객체지향 프로그램에서 보다 약한 결합도를 가지며 확장성이 좋은 설계를 위해 사용되며, 객체지향 프레임워크 및 설계 패턴에서 공통적으로 발견되는 구조이다. 객체지향 프레임워크는 메소드 재정의를 통해 다양한 응용 프로그램이 프레임워크의 행위를 확장한다. 따라서, 프레임워크에 기반한 응용 프로그램은 두 개의 층위를 가진 아키텍처를 가졌다고 할 수 있다.

설계 패턴은 설계 과정에서 자주 발생되는 문제들에 대한 해결책들을 기술하며, 설계자가 이러한 설계 기법들을 재사용할 수 있게 해준다. 대부분의 패턴들이 재정의 관계를 사용하기 때문에, 층위 관계는 여러 설계 패턴들에서 공통적으로 나타난다. 그림 2는 Observer 설계 패턴의 클래스 구조를 보여주고 있다[20]. 설계 패턴 기술은 구조 및 행위 관점 모두를 가지고 있지만, 층위 구조는 클래스 구조에 초점을 맞추고 있다.

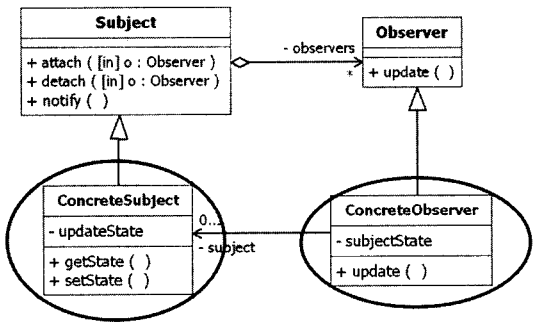

그림 2 Observer 설계 패턴의 예

Observer 설계 패턴은 Subject의 상태가 변경되었을 때, 이 Subject에 등록된 모든 Observer 객체들에게 update 메시지를 브로드캐스팅하기 위해 사용된다. 그림 3은 {Subject, Observer}와 {ConcreteSubject, ConcreteObserver} 사이의 층위 관계를 보여준다.

1) 물리적 층위 구분은 순위함수로부터 유일하게 결정되지 않는다.

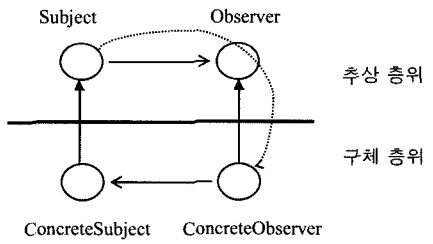

그림 3 Observer 설계 패턴의 층위 그래프

그림 2의 클래스 다이어그램은 두 층위를 가진 층위 그래프로 단순화될 수 있다: {Subject, Observer}와 {ConcreteSubject, ConcreteObserver}. 이 층위화는 추상 층위의 클래스들이 구체 층위의 클래스들을 사용해서는 안 된다는 제한 조건을 부여한다. 만약, Subject가 ConcreteObserver를 사용하고 있다면, 새로운 ConcreteObserver를 위 구조에 추가시킬 때, Subject가 이 새로운 클래스를 위해 변경될 수 있기 때문에 Observer 패턴을 사용하는 장점이 사라진다. 즉, 층위 구조를 위반하는 사용 관계는 Observer 패턴의 확장성을 해칠 수 있으며, 층위 관계가 아키텍처에서 사라지게 된다. 층위 관계 검사를 통해 설계 패턴들이 개발 과정에서 유용한 구조를 잘 유지하고 있는 지 확인할 수 있다.

4. 아키텍처 복구 절차 및 층위 구조 아키텍처 메타 모델

이 장에서는 이전 장에서 정의한 층위 관계를 바탕으로 객체지향 프로그램으로부터 층위구조 아키텍처를 복구하기 위한 절차를 기술한다. 또한, 객체지향 프로그램의 구성요소와 층위구조 아키텍처 사이의 맵핑 관계를 기술하는 메타모델을 제시한다. 제시된 아키텍처 복구 절차와 메타모델에 기반한 아키텍처 복구 수행은 5장에서 다룬다.

4.1 층위 구조 아키텍처 복구의 절차

층위 구조 아키텍처를 객체지향 프로그램으로부터 복구하기 위해서 소스모델 추출, 모듈 맵핑, 모듈 층위화, 층위 구조 아키텍처 복구 및 아키텍처 문서와의 일치성 검사를 수행한다.

소스모델의 추출

소스 모델 추출은 프로그램 소스코드로부터 아키텍처 복구에 필요한 정보를 수집하는 과정이다. 본 논문에서 소스 모델은 클래스들과 이들 간의 사용 관계 및 층위 관계로 구성된다. 클래스 간 사용 관계는 메소드 호출 및 필드 액세스와 같은 객체 상호 연동 관계와 컴파일 의존성에 기반한 타입 의존 관계가 사용될 수 있다. 전자가 실행 시 객체 간 사용 관계를 의미하는 반면, 후자는 한 클래스가 컴파일 되기 위해 필요한 모든 클래스로의 의존 관계를 의미하므로 전자를 포함하는 보다 광범위한 관계이다. 본 논문에서는 타입 의존 관계를 사용 관계로 사용하며, 이를 기반으로 유도되는 모듈 간 사용 관계는 컴파일 의존성을 보여줄 수 있다.

모듈 맵핑

추출된 소스 모델이 아키텍처로 사용하기에 너무 세분화된 클래스 수준의 관계이므로, 설계 관점의 클래스 집합 즉, 모듈을 정의하고 클래스들을 모듈로 맵핑한다. 투영 모델[12]에서는 관찰자의 기대 아키텍처의 구성 요소로 프로그램의 구현 함수들을 맵핑한다. 본 논문에서는 객체지향 프로그램의 소스 모델 상의 패키지 혹은 네임 스페이스를 하나의 아키텍처 모듈로 정의함으로써, 프로그램의 패키지 구조를 반영하는 초기 아키텍처를 정의한다. 패키지는 프로그램의 개발 초기에 작업 단위의 분할 또는 프로그램 관리 차원에서 정의되며, 프로그램 코드에 표현된 중요한 아키텍처 정보이다. 패키지 간의 계층 구조는 패키지 간의 포함 관계를 표현하지만, 본 논문에서는 이 계층 구조는 고려하지 않으며, 클래스가 직접 속하는 패키지를 하나의 모듈로 정의한다. 클래스 간의 사용 관계 및 재정의 관계로부터 모듈 관계를 정의한다.

모듈 층위화

모듈 맵핑을 통한 초기 아키텍처는 여러 모듈 간 순환 관계를 가지고 있을 수 있고, 모듈 간의 순서를 정하기 어렵다. 모듈 층위화 과정에서 관찰자는 아키텍처 문서 및 프로그램 지식에 기반하여 모듈 관계의 삭제 또는 모듈 병합을 통해 모듈 간의 부분 순서 집합을 구성한다. 이 부분 순서 집합은 모듈 간의 실제 사용 관계(use-direct)에 기반하여 재귀적으로 정의된 사용 관계로부터 유도된다. 모듈 관계의 삭제는 관계를 이루는 클래스 간 관계의 분석을 통해 이루어진다. 보다 응용 프로그램 구체적인 기능을 구현하는 모듈에서 일반적인 기능을 수행하는 모듈로의 사용 관계는 보존하고, 반대의 경우는 삭제하는 것을 원칙으로 한다. 모듈의 일반성 정도를 비교하기 어려운 순환 관계의 경우에는 모듈 병합을 수행한다.

층위 구조 아키텍처 복구 및 아키텍처 문서와의 일치성 검사

모듈 층위화를 통해 얻은 순서화된 모듈들 간에 층위 관계를 검토하여, 물리적 층위 구조를 구성한다. 물리적 층위 구조는 이전 장에 정의된 순위함수를 만족시키면서 모듈 들을 한 줄로 배치하여 얻어지는 구조이다. 물리적 층위 구조는 프로그램 상의 사용 관계 및 층위 관계 분석을 통해 유도된 구조이므로, 아키텍처 문서 혹은 관찰자의 프로그램에 대한 지식에 기반하여 각 모듈들이 속하는 층위를 결정하는 과정을 필요로 한다. 층위구

조 스타일 사용을 개발 초기에 결정하였다면, 모듈이 속하는 층위 정보를 사용하여 모듈을 배치 한 후 이 배치가 물리적 층위 구조와 일치하는 지를 검사할 수 있다.

4.2 모듈 층위화 과정에서의 모듈 병합 및 관계 삭제

층위화 과정은 모듈 간의 실제 사용 관계로부터 순환 관계를 해소하여 부분 순서집합을 얻는 과정이다. 순환 관계에 있는 모듈들을 층위 구조 스타일에 맞추기 위해서 두 가지 접근방식이 있다. 하나는 두 패키지를 병합하여 하나의 패키지로 간주하는 것이며, 나머지 하나는 패키지 간 관계를 삭제하고 이 관계를 구성하는 클래스들을 위반 경우로 다루는 방법이다.

모듈의 일반성 정도에 따라, 보다 일반적인 모듈로의 사용관계를 보존하는 방향으로 층위화를 수행한다. 모듈 간의 층위 관계는 추상 층위와 구체 층위를 나누며, 추상 층위의 모듈이 구체 층위의 모듈 보다 일반적인 기능을 구현한다고 볼 수 있다. 따라서, 사용 관계의 순환이 있을 때 층위 관계를 보존하는 방향으로 관계 삭제를 수행한다.

모듈 간 실제 사용 관계의 순환, $<M_1, M_2, \cdots, M_n = M_1>$,이 있을 때, 특정 M_i와 M_j 간 사용 관계 만이 재정의 관계인 경우, 나머지 사용 관계들 중에 하나를 삭제하여 모듈들 간의 사용 관계 순서를 정의할 수 있다. M_i가 M_j를 재정의하고, 관계 삭제를 통해 M_j가 M_i를 사용하지 않으므로, M_i와 M_j간에 층위 관계를 구성하게 된다. 순환에 둘 이상의 재정의 관계가 참여하는 경우에는 관찰자의 프로그램 지식에 의존하여 관계 삭제를 수행하거나 모듈 병합을 수행한다.

삭제된 관계들은 이후 복구된 아키텍처가 설명하지 못하는 예외이므로, 위반 경우로 관리한다. 만약 복구된 아키텍처에 기반하여 프로그램을 재구조화하는 경우에는 이러한 관계들을 리팩토링[21]과 같은 방법을 통하여 아키텍처에 일치하도록 수정하여야 한다. 본 논문에서는 아키텍처의 복구에 초점을 두며, 복구를 통한 프로그램 재구조화는 다루지 않는다.

4.3 층위 구조 아키텍처의 개념적 메타 모델

그림 4는 객체지향 프로그램으로부터 층위 구조 아키텍처를 복구하기 위해 사용되는 메타모델을 보여주고 있다. 아키텍처 컴포넌트(LayerComponent)는 하나 이상의 클래스들로 구성되며, 프로그램 상의 클래스들을 분할한다. 아키텍처 연결자(LayerConnector)는 클래스 간의 메소드 재정의 관계로부터 유도되는 모듈 간 층위 재정의 관계(LayerOverride), 클래스 간 타입 의존 관계로부터 유도되는 모듈 간 사용 관계(LayerUse)로 구성된다. 층위 관계(LayerRelationship)는 정의 8에 따라 층위 간 사용 관계와 재정의 관계로부터 유도된다: 재정의 관계를 가지며, 재정의 관계의 역방향 사용관계를 가지지 않는다.

그림 4의 좌측 사각형은 소스 모델 추출 과정을 통해 프로그램으로부터 추출되는 클래스 간 재정의 관계와 사용 관계를 표현한다. 클래스 간 관계(ClassDependency)는 재정의 관계(OverrideRelationship)와 타입 의존 관계(TypeDependency)로 구성된다. 재정의 관계는 3장에서 정의한 바와 같이 클래스 간 상속과 메소드 재정의 관계로부터 도출된다. 타입 의존 관계는 하나의 클래스가 컴파일 되기 위해 필요한 모든 클래스들로의 관계를 의미한다. 메소드 호출, 객체 생성 및 필드 액세스 관계를 클래스 간 사용 관계로 정의할 수 있으나, 타입 의존 관계가 보다 포괄적이고 보수적인 사용관계이다.

5. 소스코드로부터 층위 추출 및 아키텍처 일치성 검사

층위들 간의 층위 관계는 재사용 가능한 프레임워크 층위에 기반하여 다양한 응용프로그램 층위를 가능하게 하기 위해, 그리고 설계자 혹은 아키텍트의 의도에 따라 설정된 것이므로 개발과정에서 보존되어야 한다. 하지만, 이러한 층위 관계가 개발이 진행되면서 개발자들의

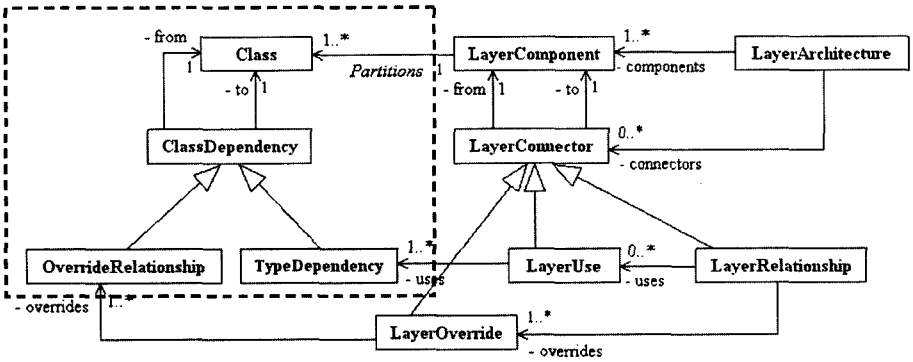

그림 4 층위구조 아키텍처 메타모델

오해나 부주의에 의해 사라질 수 있다. 추상 충위에 있는 클래스가 구체 충위의 클래스를 사용하게 되면 두 충위들은 하나의 충위로 합쳐진다. 충위가 보존되지 않으며, 아키텍트에 의해 설정된 구조가 사라질 수 있고, 충위 사용의 장점이 없어진다.

아키텍처 문서 상의 충위가 소스 코드에서 보존되는지 여부를 검사하기 위한 프로토타입 도구를 구현하였다. 이 프로토타입은 자바 파서 생성기인 JavaCC[22]를 사용하여 클래스와 이 들 간의 관계들에 대한 정보를 수집한 다음, 이 들을 패키지 들 간의 관계로 표현한다. 공개 소스 소프트웨어의 소스 코드에서 추출된 사용 관계와 재정의 관계들을 아키텍처 문서에서 기술된 아키텍처 관계들과 비교하였다.

5.1 소스 모델 추출 및 모듈 맵핑

객체지향 설계 및 구현 산출물이 충위구조 아키텍처에 맞는 지를 보이기 위해 공개 소스 프로젝트인 ArgoUML의 소스코드를 분석하였다. ArgoUML 개발자를 위한 쿡 북에 따르면 프로그램의 아키텍처는 다음 그림 5와 같은 충위구조 스타일을 따른다.

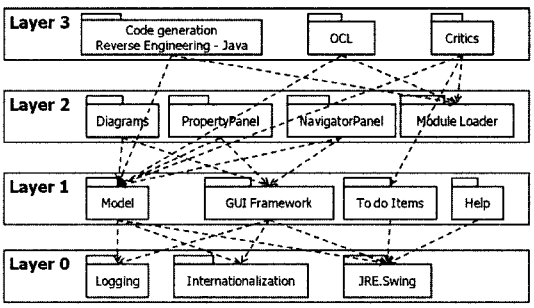

그림 5 ArgoUML의 충위구조 아키텍처

그림 5에 나오는 아키텍처는 소프트웨어의 기능을 표현하는 각 컴포넌트들과 컴포넌트 간의 사용관계들로 구성되어 있다. 쿡 북에 따르면, 각 컴포넌트는 저장될 하나의 디렉토리를 가지며, 하위 컴포넌트들은 이 디렉토리의 하위 디렉토리에 놓이게 된다. 쿡 북은 다음과 같은 충위구조 아키텍처에 대한 규칙을 말하고 있다: "ArgoUML은 아래에서 위로 작성된다. 상위 수준의 컴포넌트들은 하위 수준의 컴포넌트들에 의존하며 다른 방식으로 사용하지 않는다. 각 컴포넌트는 같은 충위 상의 다른 컴포넌트도 사용하지 않는다[7]." 하지만, 검사의 결과 ArgoUML의 구현은 이러한 아키텍처와 정확하게 일치하지는 않았다.

Java 프로그램으로부터 클래스 간 사용 관계 및 재정의 관계를 추출하기 위해, 각 클래스의 선언 정보를 JavaCC를 사용하여 추출하였다: 패키지 선언 정보, 클래스의 상속 정보, 필드 선언 타입, 메소드의 인자 및 리턴 타입, 메소드 본체 내의 로컬 변수 타입 및 메소드 시그내처. JavaCC는 Java 파일을 파싱하여 AST (Abstract Syntax Tree)를 구성하며, 각 AST 노드들을 Visitor 패턴[20]을 통해 탐색할 수 있다. 타입의 선언 정보 만을 처리하며, 프로그램의 수행 흐름에 무관하다. 본 논문에서는 인터페이스 및 중첩(nested) 클래스 모두 별도의 일반 클래스로 취급하였다. 각 클래스의 패키지 선언 정보를 통해 초기 모듈 구조와 모듈-클래스 맵핑 관계를 정의한다. 클래스에서 변수 선언을 위해 사용되는 타입 정보로부터 클래스 간 사용 관계(use-direct)를 정의한다. 상속 정보는 extends와 implements를 통해 상속하는 클래스 및 인터페이스 타입을 추출한다. 이 상속 관계는 이 후 메소드 시그내처 정보를 통해 클래스 간 메소드 재정의 관계(override)를 추출하는 데 사용된다.

ArgoUML은 89개의 자바 패키지들과 1,207개의 클래스들로 이루어져 있다. 충위 관계를 보다 구체적으로 살펴보기 위해 이들 패키지 중에 ArgoUML의 역공학 관련 기능을 구현하는 패키지들을 조사하였다. 이름 표기의 복잡성을 피하기 위해 패키지의 이름을 구성하는 org.argouml 접두어는 생략한다. reveng.java 패키지 내의 클래스들이 컴파일 되기 위해 필요한 모든 클래스들에 대한 정보를 추출하고, 하나의 자바 패키지를 충위 아키텍처의 컴포넌트로 맵핑 하였다. 역공학 패키지의 충위 구조를 파악하기 위해 사용된 패키지들의 수는 31개이며, 클래스들의 수는 총 124개이다. 충위 구조 아키텍처 표현을 단순화하기 위해 이들 패키지들 중에서 역공학과 무관한 패키지들과 이벤트 및 사용자 인터페이스 관련 구현 상세 패키지들은 생략하였다. 그림 6은 역공학과 관련된 12개 패키지들의 사용관계와 재정의 관계를 Graphviz[23]를 사용하여 보여주고 있다. 화살표는 사용관계를 표시하며, 원으로 표시된 화살표는 재정의 관계를 표시한다.

그림에서 패키지 노드에 매겨진 원 안의 숫자는 ArgoUML 아키텍처 문서 상에 매겨진 충위 번호이다. 번호가 낮을수록 하위 컴포넌트이며, 일반적 충위로 정의된 패키지이다. 하지만, 이 그래프는 패키지 간에 많은 순환 관계를 가지고 있다. 패키지 간의 순환 관계는 프로그램의 구성 기능들 간의 의존성을 이해하기 어렵게 하고, 패키지들의 개발 순서 및 테스팅 순서 결정을 어렵게 할 수 있다. 특히, 프로그램에서 공유되는 공통 기능을 구현하는 추상 충위인 kernel, model.uml 패키지가 구체 충위의 패키지들을 사용하고 있는 관계는 이들 충위에서의 변경이 전체 프로그램의 변경으로 이어질 수 있는 위험성을 내포하고 있다.

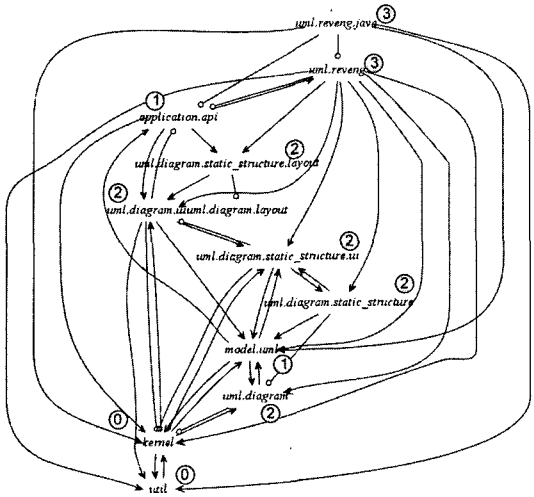

그림 6 역공학 관련 패키지들 간의 사용관계 및 재정의 관계 그래프

5.2 패키지 층위화

그림 7은 패키지 층위화를 거친 후의 그래프를 보여준다. 패키지 병합과 패키지 관계 삭제를 통해 역공학과 관련된 패키지들을 층위 구조 아키텍처로 표현하였다. 그림 상의 점선 구분은 아키텍처 문서 상에 기술된 각 패키지의 소속 층위를 나타내고 있다. L0는 커널과 공통 유틸리티를 구현하며, L1은 모델 정보와 프로그램 공통 API를 정의하며, L2는 정적 구조 다이어그램과 다이어그램 일반에 관련된 기능을, L3은 일반 역공학 지원 기능과 자바 역공학을 구현하고 있음을 패키지의 이름을 통해 알 수 있다.

uml.diagram.ui 패키지와 uml.diagram.static_structure.ui 패키지 간의 사용 관계는 여러 클래스들이 서로의 패키지 내 클래스들을 빈번하게 사용하므로, 하나의 패키지로 병합하였다. 또한, 아키텍처 문서 상에 기술된 층위 정보를 통해 추상 층위에서 구체 층위로의 사용 관계를 없애는 방향으로 패키지 간 관계 삭제를 수행하였다. 삭제된 관계와 위반 클래스는 표 1에 기술되어 있다.

그림 상에서 보이는 가로 막대들은 패키지 간 층위 구분을 나타내고 있다. 두 패키지는 재정의 관계를 가지며, 이 관계의 역방향 사용관계를 가지지 않으므로, 층위 관계에 있다. 소프트웨어를 보다 확장 가능하게 하기 위한 아키텍트의 결정 사항들을 추출된 층위 구분을 통해 알 수 있다. 예를 들어, A는 uml.reveng와 uml.reveng.java 사이의 층위 구분을 나타낸다. uml.reveng 패키지가 역공학 대상 파일들을 선택하는 대화 상자와 같은 보다 일반적인 역공학 행위를 구현하고, uml.reveng.java 패키지는 역공학 대상 언어들 중 자바 관련 파싱 행위를 구현하여, reveng 패키지에 정의된 추상 행위에 참여하는 관계를 보여주고 있다. 이 층위 구분으로부터 향후 다른 구현 언어들을 대상으로 하는 역공학 과정을 위한 재사용 구조를 알 수 있다.

또, B로 표시된 uml.diagram.static_structure와 uml.diagram 패키지 간의 층위 구분은 추상 층위를 이루는 uml.diagram 패키지가 UML의 각 다이어그램을 그리기 위한 클래스들 - 시퀀스 다이어그램, 행동 다이어그램, 상태 다이어그램 등 - 의 부모클래스로 사용되는 UMLMutableGraphModel 클래스를 구현하고, diagram.static_structure 패키지는 역공학된 자바 구현의 정적 구조를 보여주는 클래스 다이어그램에 해당하는 Class-

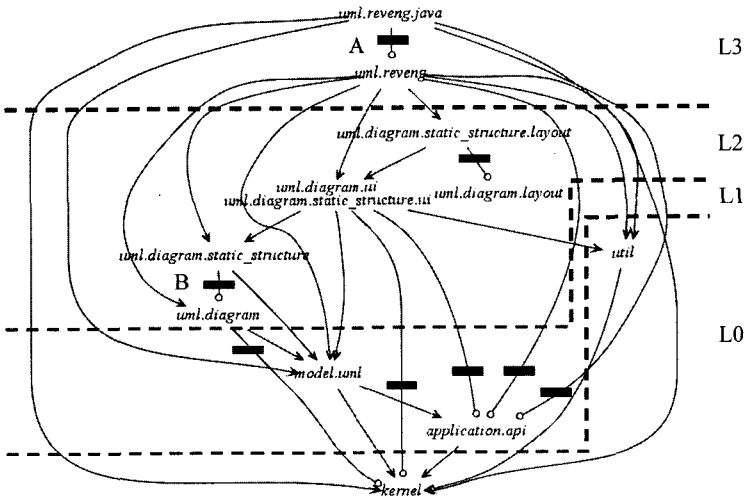

그림 7 층위화를 거친 층위 구조 그래프

표 1 Observer 설계 패턴의 층위 그래프

사용 패키지	피사용 패키지	위반 클래스
kernel	uml.diagram.ui	kernel.Project
	uml.diagram.static_structure.ui	kernel.Project
	model.uml	kernel.Project kernel.XmlFilePersister kernel.ZargoFilePersister
	util	kernel.Project kernel.ZargoFilePersister
	uml.diagram	kernel.Project
application.api	uml.reveng	application.api.PluggableImport
	uml.diagram.static_structure.layout	application.api.PluggableImport
	uml.diagram.ui	application.api.PluggableImport
model.uml	uml.diagram.static_structure.ui	model.uml.CoreFactory model.uml.UmlHelper model.uml.CoreHelper
	uml.diagram	model.uml.CoreFactory
uml.digram.static_structure	uml.digram.static_structure.ui	uml.digram.static_structure. ClassDiagramGraphModel

DiagramGraphModel 클래스를 구현하고 있다.

이 외에도, 다이어그램 상의 모델요소들의 레이아웃에 관련된 uml.diagram.layout, 프로젝트에 임포트되는 소스 파일로부터 모델 요소를 추출하기 위한 인터페이스를 정의하는 application.api, 프로젝트에 포함되는 멤버 다이어그램들에 대한 기본 구현을 가지는 kernel과 같은 패키지들이 추상 층위로 사용되고 있다.

5.3 일치성 검사

아래 표는 패키지 관계 삭제에 사용된 위반 클래스들에 대한 정보를 열거하고 있다. 이 클래스들은 결과된 층위 구조 아키텍처 상에서 추상 층위로부터 구체 층위를 사용하는 것으로, 층위 구조 아키텍처의 제한 조건을 위반하고 있다. 층위 구조 아키텍처의 규칙을 위반하는 클래스는 총 8개 클래스로 파악되었다.

이 들 클래스의 구현을 수작업을 통해 조사한 결과, 대부분의 클래스들이 층위 구조 아키텍처의 규칙에서 허용될 수 있는 예외이거나 위반의 정도가 미약함을 알 수 있었다. kernel 패키지의 Project 클래스는 ArgoUML 도구의 사용자가 현재 작업 중인 UML 모델 프로젝트의 자료구조를 구현하고 있다. 따라서, 프로젝트에 포함되는 여러 다이어그램들과 UML 모델 정보를 다루기 위해 kernel의 구체 층위의 클래스들을 사용하고 있다. 이 클래스는 전역 자료 구조와 같은 일종의 리포지토리 클래스로서, 구체 클래스들의 기능에 의존한다기 보다는 이들로부터 생성된 객체들을 속성으로 저장하고, get/set/remove와 같은 오퍼레이션들을 정의하고 있다. 하지만, 아키텍트는 이 리포지토리 클래스에 대한 변경에 대해 관리하여야 하고, 그렇지 않으면 전체 아키텍처를 해칠 수 있다. 단순 저장 목적이 아니라, 이 클래스는 각 구체 클래스들의 오퍼레이션들에 의존하게 되고, 구체 클래스의 변경이 이 클래스를 통해 다른 구체 클래스들로 파급될 가능성이 있다.

application.api 패키지의 PluggableImport 클래스는 역공학된 자바 파일들을 파싱하여 다이어그램과 UML 모델 정보로 표현하기 위해 필요한 오퍼레이션들을 정의하는 인터페이스 클래스이다. 이 클래스는 구체 층위 클래스를 오퍼레이션 정의 시 인자 타입으로 사용하고 있다. 따라서, 클래스의 이름 만을 사용하고 있으므로, 층위 구조 아키텍처의 위반으로 보기 어렵다. 또한, kernel의 파일 저장 관련 클래스들, model.uml의 팩토리 및 도움 클래스들 및 정적 구조 다이어그램의 그래프 모델 클래스는 각각 구체 층위의 한 클래스에 대해서만 일부 기능을 의존하고 있다. 따라서, 규칙의 위반 정도가 크지 않다고 볼 수 있다.

본 논문에서는 아키텍처 간의 사용 관계를 클래스 간의 타입 의존 관계의 유무를 통해 유도하기 때문에, 사용 관계의 정도를 고려하지 않고 있다. 따라서, 위반의 정도를 정량적으로 측정하지 않기 때문에, 층위 구조를 위반하는 클래스들을 수작업으로 파악하였다. 층위 구조 아키텍처의 규칙의 예외들을 정의하고, 위반 정도 정의를 통해 일치성을 자동 검사하기 위해서는 여러 프로그램으로부터의 층위 구조 아키텍처 복구에 대한 경험적인 연구가 수행될 필요가 있다.

6. 토 론

6.1 모듈 뷰 아키텍처와 실행 뷰 아키텍처

본 논문에서 복구 대상 아키텍처는 모듈 뷰 아키텍처로서 프로그램 구성요소의 분할 관점에 초점을 맞추고 있으며, 프로그램의 정적인 측면을 다루고 있다. 프로세스 간 연결 구조와 같은 실행 관점의 동적 측면 아키텍

처를 복구하지 않는다.

잘 알려진 3 계층(tier) 응용은 N계층 클라이언트-서버 아키텍처 중 하나로, 프로그램을 표현/비즈니스 로직 계층/데이타 서버의 세 계층으로 나누어 보여준다는 의미에서 층위 구조 아키텍처 스타일과 유사점을 가지고 있다. 하지만, 이 아키텍처는 역할에 따른 프로세스의 하드웨어 배치 관점과 프로세스 간 통신과 같은 실행 관점을 보여주는 혼성 아키텍처[1]라는 점에서 모듈 관점의 층위구조 아키텍처와 다르다.

실행 뷰 아키텍처는 프로세스 또는 쓰레드와 같은 컴포넌트와 프로세스 간 통신과 같은 커넥터를 사용하기 때문에, 모듈 뷰 아키텍처와 동일한 차원에서 다루기 어렵다. 모듈 관점의 층위구조 아키텍처와 실행 관점의 N계층 아키텍처는 서로 다른 기준으로 시스템을 나누기 때문에, 복구된 각각의 아키텍처의 구성요소들 간, 관계들 간 맵핑 관계를 정의하는 과정을 통해 결합될 수 있을 것이다. 이 들 다른 뷰 간의 연결 관계를 통해 정의되는 다차원 아키텍처는 다양한 관심사(concern)들에 따라 시스템을 여러 뷰들을 통해 바라볼 수 있게 하여 시스템을 이해하는 데 큰 도움을 줄 수 있다.

6.2 층위 구조 아키텍처의 자동 복구

본 논문에서 제시하고 있는 복구 방법에서는 관찰자의 프로그램에 대한 지식 및 아키텍처 문서 상의 층위 구조 정보가 중요한 역할을 수행한다. 3장에서 제시한 사용 관계와 층위 관계에 대한 정의들은 층위 구조 스타일의 규칙을 객체지향 프로그램에서 구체화 시키고 있으나, 프로그램이 가지는 층위의 적정 수 및 모듈이 속하는 층위를 자동으로 결정하지 않는다. 층위 구분을 위해 관찰자는 순환 관계에 있는 모듈들을 관계 삭제를 통해 순서화를 할 것인지 혹은 병합을 통해 순서화를 할 것인지를 결정해야 하며, 관계 삭제인 경우 어떤 관계를 삭제할 것인지를 선택해야 한다. 또한, 부분 순서화된 모듈들을 층위로 맵핑 하는 작업이 필요하다. 본 논문에서는 모듈의 소속 층위를 결정하기 위해 아키텍처 문서를 사용하였다.

모듈 들 간의 순서 관계에 기반하여 층위 구조 아키텍처를 자동으로 복구하기 위해서는 이러한 관찰자의 입력을 대신할 수 있는 전략이 필요하다. 층위구조 아키텍처의 자동 복구는 층위 구조의 제한 조건을 지키면서 모듈 간의 층위 구분을 찾는 문제라고 할 수 있다. 많은 모듈 클러스터링 연구들[24,25]이 기반하고 있는 모듈 응집도(cohesion)와 모듈 간 연관도(association)는 자동 복구를 위한 전략으로 사용될 수 있다.

7. 결론 및 향후 연구

본 논문은 클래스들간의 관계에 기반한 층위구조 아키텍처에서의 층위 관계 개념을 명확히 하였다. 더욱이, 이 관계의 의미를 객체지향 프레임워크 기반 응용 프로그램 관점에서, 설계 패턴 예를 통해 설명하였다. 이를 통해 층위 관계가 보다 약한 결합도를 가지며 보다 확장성이 좋은 설계 구조를 부여하는 데 사용되고 있음을 설명하였다. 프로그램 코드로부터 사용 관계와 층위 관계를 추출하여 구현 산출물이 아키텍처 규칙에 일치하는 지를 검사하는 프로토타입 도구를 구현하였고, ArgoUML 프로젝트의 구현 산출물과 아키텍처 문서를 통해 도구의 유용성을 보였다. 검사의 결과로, 추출된 층위 구조들이 보다 확장성 좋고 재사용이 용이한 설계를 위해 사용되고 있음을 확인하였다. 더욱이, 아키텍처 문서와 일치하지 않는 부분들이 발견되었고 분석의 결과 이들이 허용 가능한 예외로 여겨지지만 아키텍트가 이들 부분에 대한 변경을 주의 깊게 관리할 필요가 있다는 것을 지적하였다.

향후 작업으로 층위 간 연동을 표현하는 데 아키텍처의 동적 행위 측면을 포함시킬 예정이다. 현재에는 층위 구조 아키텍처를 기술하기 위해 사용 관계 및 상속 관계와 같은 클래스들의 정적인 측면 만이 사용되고 있다. 콜백 관계는 상향 사용을 위해서 구체 층위로부터 생성된 객체를 추상 층위로 등록하는 행위를 요구한다. 또한, 타입 의존 관계 대신에 호출 및 필드 액세스와 같은 사용 관계를 정의할 수 있다. 이러한 동적인 측면은 아키텍처의 구조뿐만 아니라 아키텍처의 행위를 이해하는 데 유용할 것으로 생각된다.

참 고 문 헌

[1] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J., *Documenting Software Architecture*, Addison Wesley, 2003.

[2] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., *Pattern Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons, 1996.

[3] Shaw, M. and Garlan, D., *Software Architecture*, Prentice Hall, 1996.

[4] Griswold, W. G. and Notkin, D., "Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool," *IEEE Trans. on Software Engineering*, Vol. 21, No. 4, 1995.

[5] van der Linden, F. J. and Muller, J. K., "Creating Architectures with Building Blocks," *IEEE Software*, Nov., 1995.

[6] Parnas, D. L., "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, Vol. 15, Issues 12, Dec., 1972.

[7] ArgoUML . http://argouml.tigris.org.

[8] Riva, C., "Reverse Architecting: an Industrial Ex-

perience Report," *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, 2000.
[9] Bowman, I. T., Holt, R. C., and Brewster, N. V., "Linux as a Case Study: Its Extracted Software Architecture," *Proceedings of the 21st International Conference on Software Engineering*, pp.555-563, 1999.
[10] Stoermer, C. and O'Brien, L., "MAP - Mining Architectures for Product Line Evaluations," *Proceedings of the 2nd Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001.
[11] Seacord, R. C., Plakosh, D. and Lewis, G. A., *Modernizing Legacy Systems*, Addison Wesley, 2003.
[12] Murphy, G. C., Notkin, D., and Sullivan, K. J., "Software reflexion models : bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364-380, 2001.
[13] Tran, J. B., Godfrey, M. W., Lee, E. H. S., and Holt, R. C., "Architectural Repair of Open Source Software," *Proceedings of the 8th International Workshop on Program Comprehension (IWPC'00)*, pp. 48-59, 2000.
[14] Koschke, R. and Simon, D., "Hierarchical Reflexion Models," *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, 2003.
[15] Stoermer, C., O'Brien, L., and Verhoef, C., "Moving Towards Quality Attribute Driven Software Architecture Reconstruction," *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, 2003.
[16] Stoermer, C., O'Brien, L., and Verhoef, C., "Practice Patterns for Architecture Reconstruction," *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, 2002.
[17] Riva, C., Selonen, P., Systa, T., Tuovinen, A., Xu, J., and Yang, Y., "Establishing a Software Architecting Environment," *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, 2004.
[18] Szypersky, C., *Component Software*, 2nd Ed., Addison-Wesley, 2002.
[19] Clark, D. D., "The structuring of systems using upcalls," *Proceedings of the 10th ACM symposium on Operating systems principles*, 1985.
[20] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object Oriented Software*, Addison Wesley, 1994.
[21] Fowler, M., *Refactoring, Improving the Design of Existing Code*, Addison Wesley, 1999.
[22] Java Compiler Compiler™ (JavaCC) The Java Parser Generator. https://javacc.dev.java.net.
[23] Graphviz - open source graph drawing software. http://www.research.att.com/sw/tools/graphviz.
[24] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., and Gansner, E. R., "Using Automatic Clustering to Produce High-Level System Organizations of Source Code," *Proceedings of the 6th International Workshop on Program Comprehension (IWPC'98)*, 1998.
[25] Shokoufandeh, A., Mancoridis, S., and Maycock, M., "Applying Spectral Methods to Software Clustering," *Proceedings of the 9th Working Conference in Reverse Engineering (WCRE'02)*, 2002.

박 찬 진

1994년 2월 서울대학교 계산통계학과 학사. 1994년 3월~1998년 2월 (주)LG 소프트웨어 주임연구원. 2000년 2월 서울대학교 전산학과 석사. 2000년 3월~현재 서울대학교 전기컴퓨터공학부 박사과정 관심분야는 소프트웨어 아키텍처 복구 및 평가, 소프트웨어 설계 및 공학 설계, 소프트웨어 역공학

홍 의 석

1992년 2월 서울대학교 계산통계학과 학사. 1994년 2월 서울대학교 전산학과 석사. 1999년 2월 서울대학교 전산학과 박사. 1999년 3월~2002년 2월 안양대학교 디지털 미디어 학부 교수. 2002년 3월~현재 성신여대 컴퓨터정보학부 교수. 관심분야는 소프트웨어품질예측모델, 웹기반 컴포넌트응용기술

강 유 훈

2000년 2월 서울대학교 공과대학 컴퓨터공학과 학사. 2002년 2월 서울대학교 공과대학 전기컴퓨터공학부 석사. 2002년 3월~현재 서울대학교 전기컴퓨터공학부 박사과정. 관심분야는 소프트웨어 아키텍처, 소프트웨어 디자인 및 디자인 패턴, 역공학, 프로그램 분석

우 치 수

1972년 서울대학교 응용수학과 학사. 1977년 서울대학교 전산학 석사. 1975년~1982년 울산대학교 전산학 부교수. 1982년 서울대학교 전산학 박사. 1978년 영국 Loughborough 대학 연구원. 1982년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 소프트웨어 공학, 프로그래밍 언어