프레임워크 가변부위 시험을 위한 객체 구조 패턴의 분류 및 추출 방법

(A Classification and Extraction Method of Object Structure Patterns for Framework Hotspot Testing)

김 장 래 † 전 태 웅 ††
(Jangrae Kim) (Taewoong Jeon)

요 약 객체지향 프레임워크는 개조, 합성이 용이한 클래스들로 분해될 수 있는 유연한 아키텍처를 제공함으로써 컴포넌트 기반의 효율적인 소프트웨어 개발을 지원한다. 프레임워크는 다수의 응용 소프트웨어의 개발에 반복적으로 재사용되므로 철저한 시험이 요구될 뿐만 아니라 재사용 시 확장된 프레임워크에 대해서도 추가적인 시험이 필요하다. 이를 위해서는 테스트 대상이 실행 가능한 형태로 제공되어야 하는데 그 구성 가능한 형태가 극히 다양할 뿐만 아니라 재사용될 때의 모든 형태를 예측하여 테스트하는 것은 현실적으로 불가능하므로, 재사용될 때마다 재구성되는 객체들의 구성 가능한 형태들을 동일한 특성을 갖는 유한 개의 그룹들로 분류하고, 각 그룹에서 시험 대상 실행 환경을 선정하여 시험하면 효과적인 시험이 가능하다. 본 논문에서는 재사용 시 다양한 형태의 객체 구조들로 개조, 확장될 수 있는 프레임워크의 가변부위에 대해 객체 구성의 동일한 특성을 갖는 구조적 테스트 패턴들을 조직적으로 추출하는 방법과 각 패턴들로부터 시험 대상 객체 클러스터 즉, 테스트 대상 인스턴스를 선정하는 방법을 제안한다. 이 방법은 불필요한 테스트 케이스의 선정을 피하고, 테스트 대상 실행 환경의 체계적인 구축을 위해 사용될 수 있다.

키워드 : 프레임워크 테스트, 객체지향 테스트, 테스트 패턴

Abstract An object-oriented framework supports efficient component-based software development by providing a flexible architecture that can be decomposed into easily modifiable and composable classes. Object-oriented frameworks require thorough testing as they are intended to be reused repeatedly in developing numerous applications. Furthermore, additional testing is needed each time the framework is modified and extended for reuse. To test a framework, it must be instantiated into a complete, executable system. It is, however, practically impossible to test a framework exhaustively against all kinds of framework instantiations, as possible systems into which a framework can be configured are infinitely diverse. If we can classify possible configurations of a framework into a finite number of groups so that all configurations of a group have the same structural or behavioral characteristics, we can effectively cover all significant test cases for the framework testing by choosing a representative configuration from each group. This paper proposes a systematic method of classifying object structures of a framework hotspot and extracting structural test patterns from them. This paper also presents how we can select an instance of object structure from each extracted test pattern for use in the frameworks hotspot testing. This method is useful for selection of optimal test cases and systematic construction of executable test target.

Key words : Framework Hotspot Test, Object-Oriented Test, Test Pattern

· 본 연구는 한국과학재단 목적기초연구(과제번호 : R01-1999-00238) 지원으로 수행되었음.
† 비 회 원 : LG전자 핵심망연구소 핵심망 S/W 연구원
toguru@lge.com
†† 종신회원 : 고려대학교 전산학과 교수
jeon@korea.ac.kr
논문접수 : 2001년 4월 3일
심사완료 : 2002년 5월 20일

1. 서 론

소프트웨어 프레임워크는 특정 응용 도메인에 속한 문제들의 해결에 공통적으로 사용될 수 있는 컴포넌트 유형들과 이들로 구성된 시스템 아키텍처를 일반화하여 부분적으로 구현한 시스템이다. 프레임워크 기반의 응용 소

프트웨어 시스템은 프레임워크에 규정된 시스템 구성 요소들 사이의 결합 방식과 상호 작용 규칙들에 맞게 컴포넌트들을 생성, 개조, 대체, 추가하여 프레임워크를 확장(extension), 구체화(instantiation)함으로써 만들어진다[1]. 소프트웨어를 응용 도메인 별로 재사용성이 높은 컴포넌트 라이브러리와 프레임워크들로 구축하여 이들을 다수의 응용 소프트웨어의 개발에 활용할 수 있으면 매우 효율적인 소프트웨어 개발이 가능하다. 이러한 컴포넌트/프레임워크 기반의 소프트웨어 개발 방식에서는 재사용되는 컴포넌트 라이브러리와 프레임워크들에 매우 높은 신뢰성이 요구된다. 따라서 이들에 대한 철저한 시험이 필요하나, 기존의 클래스 단위의 시험에 효과적인 테스트 방법[2, 3, 4]들 만으로는 프레임워크의 효과적인 테스트를 어렵게 하는 여러 가지 문제들을 지니고 있다. 개조와 확장이 빈번하게 일어나는 프레임워크를 효율적으로 시험하기 위해서는 변경된 프레임워크의 점진 결함들(progressive faults)과 회귀 결함들(regression faults)에 대한 점증적인 (재)시험(incremental testing)이 요구된다. 그런데 프레임워크는 (재)사용되는 형태와 상황(context)이 다양하고 확장 부위가 프레임워크의 제어를 받게 되는 경우가 많아서 재사용 시 개조, 합성되는 클래스들 사이의 상호 작용들과 변경에 따른 파급 효과를 사전에 명시하거나 예측하기가 쉽지 않다.

특히 프레임워크가 제공하는 시스템 수준의 아키텍쳐가 재사용될 때 발생할 수 있는 불일치 유형들은 다양하다[5]. 프레임워크의 이러한 성격은 프레임워크에 대한 테스트 케이스의 생성과 테스트 실행 환경의 구축을 어렵게 한다.

프레임워크를 시험하기 위해서는 테스트 대상이 실행 가능한 형태로 제공되어야 하는데 그 구성 가능한 형태가 극히 다양할 뿐만 아니라 재사용될 때의 모든 형태를 예측하여 테스트하는 것은 현실적으로 불가능하므로 임의의 형태나 빈번히 사용될 것으로 예측되는 형태를 구성하여 시험하는 것이 일반적이다. 그러나 이러한 방법은 시험자의 직관력에 의존적이거나 특정 형태의 재사용 가능성에 대해 시험이 편중되므로 비효율적이고, 프레임워크의 확장 및 구체화 과정에서 발생할 수 있는 결함들을 발견하지 못할 가능성이 높다. 따라서 프레임워크가 재사용될 때마다 재구성되는 객체들의 구성 가능한 형태들을 동일한 특성을 갖는 유한 개의 그룹들로 분류하고, 각 그룹에서 시험 대상 실행 환경을 선정하여 시험하면 효과적인 시험이 가능하다.

본 논문에서는 객체지향 프레임워크의 효율적인 시험을 위해 프레임워크가 재사용될 때마다 변경이 발생하는 부위에 대한 테스트 모델로부터 시험 전략(test strategy)을 적용하여 변경 가능 영역을 종류별로 분류하고, 이로부터 인스턴스를 선정하여 테스트 대상의 실행 환경을 구성하는 방법을 제안한다.

2. 프레임워크의 고정부위와 가변부위

프레임워크는 소프트웨어의 뼈대를 형성하는 아키텍쳐를 구현한 것이다. 또한 서브 시스템의 설계이며 추상 클래스들과 구체 클래스들, 그들 간의 인터페이스로 구성된다[6]. 소프트웨어 프레임워크는 자체의 포괄적인 아키텍쳐 즉, 빌딩 블록들간의 합성 및 상호작용에 대한 규칙과 제약 조건들은 계약에 의한 설계(design by contract) 원리[7, 8]에 의해 잘 정의 되어 있다. 프레임워크의 가변부위(hot spot)들에 해당되는 부분은 쉽게 개조될 수 있도록 일반화되어 유연성을 갖고 있고, 고정부위(frozen spot)들은 미리 공통적 요구사항에 대한 정의를 구체적으로 구현한 부분으로써 변경이 불가능하다.

객체지향 프레임워크는 고정부위와 가변부위가 클래스 또는 메쏘드 단위로 분리되어 설계된다. 가변부위와 접속된 고정부위에 해당하는 클래스(메쏘드)를 템플릿 클래스(메쏘드)라고 부르고 가변부위에 해당하는 클래스(메쏘드)를 후크 클래스(메쏘드)라고 부른다[9]. 템플릿 클래스(메쏘드)와 후크 클래스(메쏘드)가 합성되는 형태는 여러 가지 기준 및 관점에 따라 분류할 수 있는데 Pree[9]는 두 가지 기준을 사용하여 분류하였다.

1) 템플릿 클래스 객체에 합성되는 후크 클래스 객체의 개수 (객체 참조 변수의 복수 허용 여부)

2) 상호 합성 관계인 템플릿 클래스와 후크 클래스 사이의 상속 관계 여부

위의 기준들에 의해 템플릿 클래스(메쏘드)와 후크 클래스(메쏘드)가 합성되는 형태는 그림 1과 같이 7가지 메타 패턴으로 분류될 수 있다.

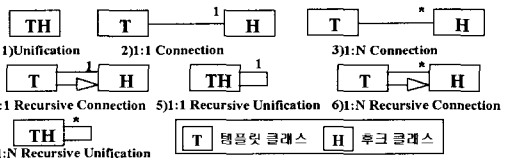


그림 1 템플릿-후크 클래스 합성 패턴

객체지향 프레임워크는 이러한 합성 패턴에 따라 상호 연결된 템플릿 클래스와 후크 클래스들 사이에 허용 가능한 다양한 구조적 관계와 상호 작용들을 통하여 프레임워크의 고정 기능 요소와 가변 기능 요소가 결합된

프레임워크의 기능을 수행한다.

3. 프레임워크의 테스트 패턴

3.1 객체지향 소프트웨어 테스트

앞서 언급된 바와 같이 객체지향 소프트웨어는 구조적 소프트웨어에 비해 시험하기 어려운 구성 요소들로 이루어져 있다. 기존의 구조적 소프트웨어의 시험을 위해 사용하던 방법들을 객체지향 소프트웨어의 시험에 적용, 메쏘드가 호출될 수 있는 순서들의 조합을 메쏘드 호출 순서 명세(method sequence specification)로 정의하고 이로부터 유도된 상태 전이 다이어그램을 이용하여 테스트 케이스와 오러클(test oracle)을 생성하는 방법[2], 클래스 멤버 함수들 간의 주고 받는 자료의 흐름을 분석해 자료 흐름도를 작성하고 이를 기반으로 테스트 케이스를 생성하는 방법[3] 등이 연구되었다. 또, 상태 전이 다이어그램을 이용한 정형 명세로부터 테스트 시나리오를 생성하는 방법[10]과 컬렉션 클래스들(collection classes)을 시험하기 위해 상태 전이 다이어그램을 모델링한 테스트 그래프를 이용하여 테스트 케이스와 테스트 오러클 등을 생성하는 방법[11] 등 객체지향 소프트웨어의 시험에 관한 연구가 각지에서 진행되고 있으나, 객체지향 소프트웨어의 분석 및 설계에 관한 연구에 비하면 미비하다 할 수 있다. 특히 객체지향 프레임워크의 테스트를 위한 테스트 대상 인스턴스 생성 방법에 관한 연구는 더욱 그러하다.

3.2 테스트 패턴 추출 방법

테스트 데이타 분류 방법(TTF : Test Template Framework)은 소프트웨어의 기능 단위(functional unit)인 오퍼레이션을 테스트하기 위해 입력 데이타 영역을 분할 기준에 의거, 세분화하여 시험 데이타를 선정하기 위해 Stocks 와 Carrington[12, 13]에 의해 고안되었다. 이러한 분류는 특정 입력 데이타가 분할 영역을 대표하는 값이 될 때까지 반복된다.

본 논문에서는 객체지향 프레임워크의 가변부위에 해당하는 클래스 구조로부터 생성 가능한 무수히 많은 형태의 객체 구조를 연결 형태별로 분류하고 분류된 패턴으로부터 시험 대상 객체 클러스터를 인스턴스로 선정하기 위해 테스트 데이타 분류 방법을 이용한다. Stocks와 Carrington은 Z[14] 명세 언어를 이용하여 테스트 데이타 분류 방법을 정형화 하였으나, 이는 객체지향 소프트웨어를 명세하기에 복잡하므로 UML[15]과 Object-Z[16] 명세 언어를 이용하면 테스트 대상 모델과 테스트 패턴 추출 과정 등을 효과적으로 정형 명세할 수 있다.

본 논문의 테스트 대상이 되는 CUT(Class cluster Under Test)는 프레임워크의 가변부위를 형성하며, 클래스들 간의 합성 패턴에 따라 다양한 객체 구조로 생성될 수 있다. CUT 모델은 다양한 정보를 포함하고 있는데, 그 중에서 테스트에 필요한 부분만을 추출하거나 테스트에 필요한 정보를 추가해서 테스트 모델을 구성한다[17].

테스트 모델은 테스트 대상이 되는 클래스들의 객체를 생성하고 그 객체들에 대한 테스트 케이스를 구성하기에 충분한 정보를 보유하고 있어야 한다. 테스트 모델의 클래스 클러스터 구조로부터 생성될 수 있는 객체 클러스터의 형태는 매우 다양하며 이러한 객체 클러스터 구조들의 집합은 분할 기준(partition criteria, heuristic)에 의해 분할 영역(Test Template)으로 분류될 수 있다.

각각의 분할 영역은 상위 영역에 대한 제약 조건(constraint)의 형태로 표현되며,

$$TT_{CHILD} \triangleq [\, TT_{PARENT} \mid constraints \,]$$

상위 영역을 하위 분할 영역으로 분할하는 분할 함수 TTH(Test Template Hierarchy)는 계층적인 TTF 분할 구조에서 상위의 구성 가능한 객체 구조(VOS : Valid Object Structure)영역과 분할기준의 쌍을 하위 영역들의 집합으로 대응시키는 함수이다.

$[\, VOS \,]$

$[\, HEURISTIC \,]$

$$TTH : \mathbb{P}\, VOS \times HEURISTIC \rightarrow \mathbb{P}\,(\mathbb{P}\, VOS)$$

분할 함수에 의해 상위 영역으로부터 세분화된 하위 영역 각각의 임의의 인스턴스는 서로 중복되지 않으며(mutual exclusion condition), 모든 하위 영역들로부터 생성 가능한 인스턴스들의 집합은 다시 상위 영역의 인스턴스들과 같다(covering condition).

$$\begin{aligned} &\text{heuristic} : HEURISTIC \\ &\forall vos : \mathbb{P}VOS \bullet (\forall T1, T2 : TTH\,(vos, heuristic) \\ &\qquad \mid T1 \neq T2 \bullet T1 \cap T2 = \varnothing) \\ &\forall vos : \mathbb{P}VOS \bullet (\forall v : vos \\ &\qquad \bullet (\exists T : TTH\,(vos, heuristic) \bullet v \in T)) \end{aligned}$$

이러한 분류는 객체 구조의 확장이 더 이상 지속될 수 없거나, 확장 유형이 상위 단계에서 나타났던 형태와 일치할 때까지 즉, 가능한 모든 형태의 객체 구조 확장이 표현될 때까지 반복된다. 더 이상 세분되지 않는 분할 영역들을 테스트 패턴으로 정의한다.

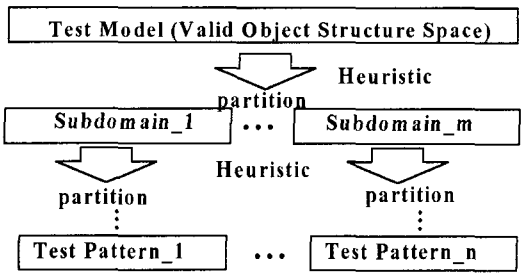


그림 2 객체 클러스터 구조의 분류

그림 2는 테스트 모델에 분류기준을 적용하여 테스트 패턴을 추출하는 과정을 표현한다. 테스트 모델로부터 추출된 테스트 패턴들은 해당 테스트 모델로부터 생성 가능한 모든 객체 클러스터 구조를 세분화한 결과이며, 각각의 테스트 패턴은 분할 기준에 대해 동일한 특성을 갖는 동치 집합(equivalence class)이다. 특정 테스트 패턴으로부터 생성된 임의의 객체 클러스터 구조들은 모두 해당 테스트 패턴의 특성을 보유하고 있으므로, 그 테스트 패턴을 대표하는 테스트 대상 실행 환경이다. 테스트 인스턴스는 분할 함수에 테스트 패턴과 인스턴스 선택 기준을 적용하여 선정한다. 예를 들어, 각각의 테스트 패턴에서 임의로 두 개의 인스턴스를 선정해서 테스트 대상으로 사용하고자 할 때 다음과 같을 수 있다.

| random_2 : HEURISTIC
| TestPattern : P VOS
TTH(TestPattern, random_2) = { Instance : PTestPattern |
#Instance = 2 ∧ (∃ i_1, i_2 : Instance • $i_1 \neq i_2$) }

4. 합성 패턴의 구조적 테스트 패턴 추출

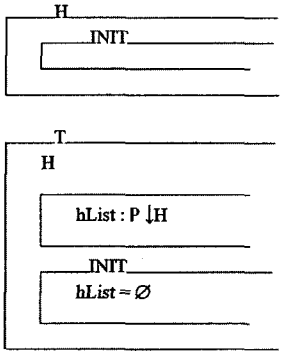
객체지향 프레임워크의 고정부위와 가변부위의 연결부위는 템플릿 클래스(메쏘드)와 후크 클래스(메쏘드)의 연결 형태가 되며 매우 복잡하고 다양한 구조를 이룰 수 있다. 그러나, 이들의 연결 구조는 2장에서 기술된 바와 같이 크게 7개의 합성 패턴으로 분류될 수 있으므로 이 합성 패턴들로부터 3.2절의 방법을 이용하여 테스트 패턴을 추출하고 테스트 패턴의 인스턴스를 선정하면, 각 합성 패턴들의 다양한 테스트 실행환경의 표본을 추출할 수 있다.

2장에서 분류한 7개의 합성 패턴들 간의 관계를 살펴보면, *Unification*과 *1:1 Recursive Unification*의 합성 패턴에서 생성되는 객체 구조들은 1:N Recursive Unification 합성 패턴에서 생성 가능한 객체 구조들의 부분 집합(subset)임을 알 수 있다. 또 1:1 Connection과 1:1 Recursive Connection의 합성 패턴에서 생성되는 객체 구조들은 각각 1:N Connection과 1:N Recursive Connection의 합성 패턴에서 생성 가능한 객체 구조들의 부분 집합이다. 즉, 1:N Connection, 1:N Recursive Connection, 1:N Recursive Unification 3개의 합성 패턴의 생성 가능한 객체 구조들이 나머지 패턴들의 생성 가능한 객체 구조들을 포함하게 된다. 그런데, 1:N Connection은 1:N Recursive Connection의 템플릿-후크 클래스간 상속관계가 없는 단순한 형태이며, 또한 *1:N Recursive Unification*도 템플릿, 후크 메쏘드가 하나의 클래스에 포함된 형태로써, 템플릿 클래스가 후크 클래스를 상속하여 템플릿 클래스안에 템플릿, 후크 메쏘드를 모두 포함하는 1:N Recursive Connection의 단순한 형태임을 알 수 있다.

따라서, 본 장에서는 프레임워크 가변부위를 형성할 수 있는 다양한 합성 패턴 가운데 1:N Recursive Connection(*NRC*)으로 합성되는 경우가 가장 복잡하고, 다른 일부 합성 패턴을 포함하고 있으므로 이에 대해 테스트 모델을 정형화하고 이로부터 테스트 패턴의 추출, 테스트 대상 인스턴스 선정 과정 등을 기술함으로써 다른 모든 합성 패턴들의 테스트 패턴 추출, 인스턴스 선정 과정을 대신한다.

4.1 1:N Recursive Connection 합성 패턴의 테스트 모델

1:N Recursive Connection의 합성 패턴은 템플릿 클래스 객체가 그 객체에 1:N으로 연결된 후크 클래스의 서브클래스의 인스턴스들 이다. 템플릿 클래스 객체에 연결된 복수의 후크 클래스 객체들이 다시 각각 템플릿 클래스 객체로서 또 다른 복수의 후크 클래스 객체들에 연결되는 과정을 반복할 수 있다. 후크 클래스와 템플릿 클래스를 Object-Z 스키마로 표현하면 다음과 같다.



위의 후크 클래스와 템플릿 클래스의 객체들로 이루어진 객체 클러스터는 템플릿 클래스의 인스턴스 변수 hList를 통해 후크 클래스 또는 그 서브 클래스 객체들을 참조하여 참조 관계의 트리 구조를 형성하게 된다. 그러나 트리 구조에서 특정 객체가 자신을 참조하거나, 하위의 객체가 다시 상위 객체를 참조하는 재귀 참조(recursive reference)와 참조 관계의 순환(circular reference)은 허용되지 않는다. 이러한 제약 조건을 표현하기 위해 directReference 함수와 allReference 함수를 정의할 수 있다. directReference 함수는 특정 템플릿 클래스 객체를 그 객체가 직접적으로 참조하는 템플릿 클래스 객체들에 대응시키며, allReference 함수는 특정 템플릿 클래스 객체를 그 객체가 직/간접적으로 참조하는 모든 템플릿 클래스 객체들에 대응시킨다.

$$\begin{aligned}&\text{directReference} : T \rightarrow P \downarrow T\\&\text{allReference} \quad : T \rightarrow P \downarrow T\\&\hline\\&\forall t : T \bullet\\&\quad \text{directReference}(t) = t.\ \text{hList}\\&\quad \text{allReference}(t) = \text{directReference}(t) \cup\\&\quad (\cup \{\ dr : \text{directReference}(t) \bullet \text{allReference}(dr)\ \})\end{aligned}$$

두 함수를 이용하여 객체 클러스터의 구성 요소와 제약 조건을 표현하면 다음과 같다.

$$\begin{aligned}&\text{ObjectCluster\_NRC}\\&\hline\\&\quad \text{TObject} : P\ T\\&\quad \text{HObject} : P\ H\\&\quad \text{TObject}_1 : \text{TObject}\\&\quad\hline\\&\quad \forall t : \text{TObject} \bullet t \notin \text{allReference}(t)\\&\quad \forall t : \text{TObject} \mid t \neq \text{TObject}_1\\&\qquad\qquad \bullet t \in \text{allReference}(\text{TObject}_1)\\&\quad \forall t : \text{TObject} \mid t \neq \text{TObject}_1\\&\qquad\qquad \bullet \text{TObject}_1 \notin \text{allReference}(t)\\&\hline\end{aligned}$$

이때 $TObject_1$은 객체 트리 구조의 최상위(root) 노드에 위치한 객체로서 $TObject_1$ 객체를 참조하는 다른 템플릿 클래스 객체는 존재하지 않으며 다른 모든 객체들은 $TObject_1$에 의해 직/간접적으로 참조된다.

4.2 1:N Recursive Connection 합성 패턴의 테스트 패턴 추출

위의 스키마들은 1:N Recursive Connection의 합성 패턴의 테스트 모델이 된다. 그런데 위의 객체들 구조에서 H 타입의 객체가 T 타입의 객체 없이 단독으로 존재할 수 없으므로, 테스트 모델로부터 구성 가능한 객체 구조는 다음과 같이 정의할 수 있다.

$$\text{VOS} \cong [\ \text{ObjectCluster\_NRC} \mid \#\text{HObject} > 0 \Rightarrow \#\text{TObject} > 0\ ]$$

테스트 패턴을 추출하기 위한 분할의 첫번째 단계는 VOS 영역을 분할하며 이때 사용되는 분할 기준으로는 T 클래스의 H 클래스에 대한 인스턴스 변수 hList의 cardinality가 0 또는 그 이상으로 선언되어 있으므로 hList에 연결된 객체의 개수에 기반한 분할(cardinality based partitioning)을 적용할 수 있다.

$$\begin{aligned}&[\ \text{HEURISTIC}]\\&\mid\ \text{cardinality} : \text{HEURISTIC}\end{aligned}$$

따라서 VOS를 분류한 첫 번째 단계의 하위 영역들 스키마와 TTH함수는 다음과 같다.

($TObject_1$: 첫 번째 단계의 분할에서 생성된 T 타입의 객체)

$$\text{TTH}_{\text{NRC}}(\text{VOS, cardinality}) = \{TT_1, TT_2, TT_3\}$$

$$TT_1 \cong [\ \text{VOS} \mid \#\{\ \text{TObject}_1.\text{hList}\ \} = 0\ ]$$

$$TT_2 \cong [\ \text{VOS} \mid \#\{\ \text{TObject}_1.\text{hList}\ \} = 1\ ]$$

$$TT_3 \cong [\ \text{VOS} \mid \#\{\ \text{TObject}_1.\text{hList}\ \} > 1\ ]$$

이것을 객체 다이어그램으로 표현다면 그림 3과 같다.

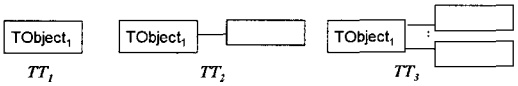


그림 3 NRC VOS의 cardinality에 의한 분할 객체 다이어그램

TTH_{NRC}에 의해 VOS로부터 세분화된 분할영역 TT_1, TT_2, TT_3 각각의 임의의 인스턴스는 서로 중복되지 않으며(mutual exclusion condition) 모든 분할 영역들로부터 생성된 인스턴스들의 집합은 다시 VOS와 일치(covering condition)한다.

$$\begin{aligned}&\forall vos : \text{PVOS} \cdot (\forall T1, T2 : \text{TTH}_{\text{NRC}}\\&\qquad (vos, \text{cardinality}) \mid T1 \neq T2 \cdot T1 \cap T2 = \varnothing)\\&\forall vos : \text{PVOS} \cdot (\forall v : vos \cdot\\&\qquad (\exists T : \text{TTH}_{\text{NRC}}(vos, \text{cardinality}) \cdot v \in T))\end{aligned}$$

위의 첫 번째 단계에서 도출된 세 개의 영역들은 다시 다음 단계에서 세분화될 수 있다. 두 번째 단계에서는 참조변수 $TObject_1.hList$에 할당된 참조 대상 객체의 타입에 의한 분할(type based partitioning) 기준을 적용할 수 있다. 그 이유는 $TObject_1$의 hList에 복수개의 T 타입 객체가 할당 되면 전체적인 객체 클러스터 구조가 트리의 형태로 구성되며, 0 또는 1 개의 T 타입 객체가 할당 되면 객체 클러스터 구조가 리스트(list)의 형태로 구성되기 때문이다. 따라서, 첫번째 단계에서 분할된 3개의 템플릿에 대해 $TObject_1.hList$에 T 타입의

객체가 몇 개 할당되어 있는가(cardinality_T)에 따라 분할하며 그 결과는 다음과 같다.

| cardinality_T : HEURISTIC

TTH_{NRC} (TT_1, cardinality_T) = {TT_1}

TTH_{NRC} (TT_2, cardinality_T) = {$TT_{2.1}$, $TT_{2.2}$}

$TT_{2.1} \cong$ [TT_2 | #{t : TObject | t ∈ $TObject_1$.hList } = 1]

$TT_{2.2} \cong$ [TT_2 | #{t : TObject | t ∈ $TObject_1$.hList } = 0]

TTH_{NRC} (TT_3, cardinality_T) = {$TT_{3.1}$, $TT_{3.2}$, $TT_{3.3}$}

$TT_{3.1} \cong$ [TT_3 | #{t : TObject | t ∈ $TObject_1$.hList } >1]

$TT_{3.2} \cong$ [TT_3 | #{t : TObject | t ∈ $TObject_1$.hList } = 1]

$TT_{3.3} \cong$ [TT_3 | #{t : TObject | t ∈ $TObject_1$.hList } = 0]

위에서 TT_1은 $TObject_1$.hList가 공집합으로 어떤 타입의 객체도 할당될 수 없으므로 더 이상 분할되지 않는 영역임을 알 수 있다. 이와 같이 더 이상 분할되지 않거나, 영역이 분할될 때 객체 클러스터 구조 확장 형태가 반복적으로 나타나면 테스트 패턴으로 정의하고 더 이상 분할하지 않는다. 다음 단계의 분할은 $TObject_1$ 객체의 참조 변수 hList에 할당된 H 타입 객체의 개수에 의해 분할 될 수 있다.

| cardinality_H : HEURISTIC

TTH_{NRC} ($TT_{2.1}$, cardinality_H) = { $TT_{2.1}$}

TTH_{NRC} ($TT_{2.2}$, cardinality_H) = { $TT_{2.2}$}

TTH_{NRC}($TT_{3.1}$,cardinality_H) = {$TT_{3.1.1}$, $TT_{3.1.2}$, $TT_{3.1.3}$}

$TT_{3.1.1} \cong$ [$TT_{3.1}$ | #{h : HObject | h ∈ $TObject_1$.hList } >1]

$TT_{3.1.2} \cong$ [$TT_{3.1}$ | #{ h : HObject | h ∈ $TObject_1$.hList } = 1]

$TT_{3.1.3} \cong$ [$TT_{3.1}$ | #{ h : HObject | h ∈ $TObject_1$.hList } = 0]

TTH_{NRC} ($TT_{3.2}$, cardinality_H) = {$TT_{3.2.1}$, $TT_{3.2.2}$ }

$TT_{3.2.1} \cong$ [$TT_{3.2}$ | #{h : HObject | h ∈ $TObject_1$.hList } = 1]

$TT_{3.2.2} \cong$ [$TT_{3.2}$ | #{ h : HObject | h ∈ $TObject_1$.hList } >1]

TTH_{NRC} ($TT_{3.3}$, cardinality_H) = { $TT_{3.3}$}

$TT_{3.1}$과 $TT_{3.2}$는 다음 단계에서 계속 분할되나, $TT_{2.1}$은 분할될 때 객체 클러스터 구조 확장 형태가 VOS가 분할될 때와 같고 $TT_{2.2}$, $TT_{3.3}$은 더 이상 분할되지 않으므로 테스트 패턴이다. 또 $TT_{3.2}$는 $TObject_1$.hList에 할당된 객체 수가 1 보다 크고 그 중 T 타입의 객체 수는 1이므로 $TObject_1$.hList에 할당된 H 타입의 객체 수가 0인 경우는 존재하지 않기 때문에 위와 같이 두 개의 하위 영역으로 분할된다.

이번 단계에서 도출된 5개의 테스트 템플릿 $TT_{3.1.1}$, $TT_{3.1.2}$, $TT_{3.1.3}$, $TT_{3.2.1}$, $TT_{3.2.2}$은 다음 단계에서의 분할 과정 시 객체 구조 확장 형태가 이전 단계의 형태와 같으므로 더 이상 분할하지 않는다. 따라서 VOS의 분할 결과 모두 9개의 테스트 패턴이 추출되었다.

$TT_1 \cong [ObjectCluster\_NRC \mid \#TObject_1.hList = 0]$

$TT_{2.1} \cong [ObjectCluster\_NRC \mid \#TObject_1.hList = 1 \wedge (\exists_1 t : TObject \mid t \in TObject_1.hList)]$

$TT_{2.2} \cong [ObjectCluster\_NRC \mid \#TObject_1.hList = 1 \wedge (\exists_1 h : HObject \mid h \in TObject_1.hList)]$

$TT_{3.3} \cong [ObjectCluster\_NRC \mid \#TObject = 1 \wedge \#HObject > 1 \wedge (\forall h : HObject \mid h \in TObject_1.hList)]$

$TT_{3.1.1} \cong [ObjectCluster\_NRC \mid \#\{h : HObject \mid h \in TObject_1.hList\} > 1 \wedge \#\{t : TObject \mid t \in TObject_1.hList\} > 1]$

$TT_{3.1.2} \cong [ObjectCluster\_NRC \mid \#\{h : HObject \mid h \in TObject_1.hList\} = 1 \wedge \#\{t : TObject \mid t \in TObject_1.hList\} > 1]$

$TT_{3.1.3} \cong [ObjectCluster\_NRC \mid \#\{h : HObject \mid h \in TObject_1.hList\} = 0 \wedge \#\{t : TObject \mid t \in TObject_1.hList\} > 1]$

$TT_{3.2.1} \cong [ObjectCluster\_NRC \mid \#\{h : HObject \mid h \in TObject_1.hList\} = 1 \wedge \#\{t : TObject \mid t \in TObject_1.hList\} = 1]$

$TT_{3.2.2} \cong [ObjectCluster\_NRC \mid \#\{h : HObject \mid h \in TObject_1.hList\} > 1 \wedge \#\{t : TObject \mid t \in TObject_1.hList\} = 1]$

그림 4는 각 단계의 분할 과정과 테스트 템플릿들을 객체 다이어그램으로 표현한 것이며, 어두운 사각형으로 표시된 부분이 테스트 패턴의 객체 다이어그램이다.

4.3 1:N Recursive Connection 합성 패턴의 테스트 인스턴스 선정

실제 테스트를 수행하기 위해 필요한 것은 테스트 대상의 인스턴스 이다. 앞 절에서 도출된 테스트 패턴들로

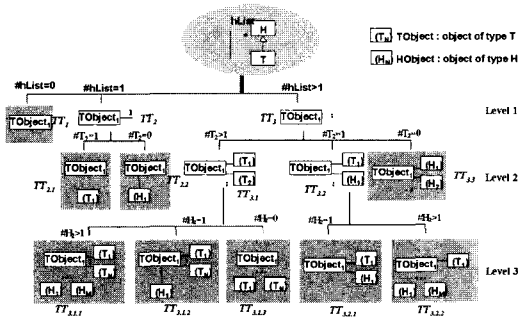


그림 4 1:N Recursive Connection 테스트 모델의 구조적 테스트 패턴 추출

부터 테스트 패턴의 인스턴스를 선정하는 과정 역시 분할 기준(인스턴스 선택 기준)을 적용한 TTH 함수를 이용하여 이루어진다. 각각의 테스트 패턴으로부터 생성 가능한 객체 클러스터 구조 즉, 테스트 패턴 인스턴스의 개수는 무한히 많을 수 있다. 그러나 특정 테스트 패턴에서 선정된 임의의 테스트 패턴 인스턴스는 해당 테스트 패턴의 특성을 대표하므로 각각의 테스트 패턴으로부터 하나, 또는 그 이상의 인스턴스를 생성하면 객체지향 프레임워크의 가변 부위를 테스트하기 위한 객체 연결 구조로 사용될 수 있다.

적용 가능한 다양한 분할 기준 가운데 최소 규모의 객체 구조를 생성해서 테스트 하기 위한 최소값(minimum) 분할 기준을 적용할 수 있다.

| minimum : HEURISTIC

최소값 분할 기준은 해당 테스트 패턴으로부터 생성 가능한 다양한 객체 구조들 가운데 테스트 패턴의 제약 조건(constraints)을 만족하는 가장 적은 개수의 객체로 이루어 지는 객체 구조를 선정하기 위한 분할 기준이다. 테스트 패턴으로부터 생성 가능한 임의의 객체 구조 인스턴스는 해당 테스트 패턴의 모든 가능한 객체 구조 인스턴스과 동일한 제약 조건(테스트 패턴을 추출하기 위해 적용되었던 분할 기준들)을 만족하는 동치 집합의 원소이므로 최소값 분할 기준으로 선정된 인스턴스 역시 그 패턴을 대표하는 표본이다.

예를 들어, $TT_{3.1.1}$에 최소값 분할 기준을 적용하면, 특정 템플릿 객체(*TObject*$_1$)의 참조 변수 hList가 나머지 모든 객체들을 참조하고 있으며 hList에 의해 참조되는 H 타입과 T 타입의 객체 수가 각각 1 보다 커야 하므로 최소값 2를 선택하여 선정한 인스턴스는 $TImin_{3.1.1}$ (Test Instance of $TT_{3.1.1}$ using minimum heuristic)이며 분할 함수는 다음과 같다.

TTH_{NRC} ($TT_{3.1.1}$, minimum) = { $TImin_{3.1.1}$ }

$TImin_{3.1.1}$
- $TT_{3.1.1}$
- t1, t2 : TObject
- h1, h2 : HObject

- $TObject_1$.hList = {t1, t2, h1, h2}
- t1.hList = {}
- t2.hList = {}

이와 같은 방법으로 나머지 테스트 패턴들에 최소값 분할 기준을 적용하여 인스턴스를 선정하면 다음과 같다.

$TImin_1$
- TT_1

- $TObject_1$.hList = {}

$TImin_{2.1}$
- $TT_{2.1}$
- t1 : TObject

- $TObject_1$.hList = {t1}
- t1.hList = {}

$TImin_{2.2}$
- $TT_{2.2}$
- h1 : HObject

- $TObject_1$.hList = { h1}

$TImin_{3.3}$
- $TT_{3.3}$
- h1, h2 : HObject

- $TObject_1$.hList = { h1, h2}

$TImin_{3.1.2}$
- $TT_{3.1.2}$
- t1, t2 : TObject
- h1 : HObject

- $TObject_1$.hList = {t1, t2, h1}
- t1.hList = {}
- t2.hList = {}

$TImin_{3.1.3}$
- $TT_{3.1.3}$
- t1, t2 : TObject

- $TObject_1$.hList = {t1, t2}

t1.hList = {}
t2.hList = {}

$TImin_{3.2.1}$
$TT_{3.2.1}$
t1 : TObject
h1 : HObject

$TObject_1$.hList = {t1, h1}
t1.hList = {}

$TImin_{3.2.2}$
$TT_{3.2.2}$
t1 : TObject
h1, h2 : HObject

$TObject_1$.hList = {t1, h1, h2}
t1.hList = {}

4.4 테스트 인스턴스 검증

테스트 패턴으로부터 인스턴스를 선정하여 생성하면 테스트 대상의 실행 환경이 구성된다. 그러나 그에 앞서 인스턴스의 정확성 검증이 선행되어야 한다. 본 절에서는 인스턴스가 해당 테스트 패턴의 영역에 속하는가에 대한 증명을 보여준다.

테스트 패턴 TT_1, $TT_{2.2}$ 들로부터 선정할 수 있는 테스트 인스턴스는 유일한 반면 테스트 패턴 $TT_{2.1}$, $TT_{3.3}$, $TT_{3.1.1}$, $TT_{3.1.2}$, $TT_{3.1.3}$, $TT_{3.2.1}$, $TT_{3.2.2}$ 등은 무한히 많은 형태의 테스트 인스턴스를 선정할 수 있다. 그러므로 최소값 분할 기준을 적용하여 선정한 인스턴스가 해당 테스트 패턴의 제약 조건을 만족하는가에 대한 검증이 필요하다.

테스트 패턴 $TT_{3.1.1}$의 영역 중에 인스턴스 $TImin_{3.1.1}$이 존재함을 보이기 위해 다음을 증명한다.

$\vdash \exists\ TT_{3.1.1} \cdot TImin_{3.1.1}$
$\Leftrightarrow$
(∃ TObject : P T ;
  HObject : P H
| (∃ $TObject_1$: TObject • #{h : HObject |
        h ∈ $TObject_1$.hList } > 1
  ∧ #{t : TObject | t ∈ $TObject_1$.hList } > 1)
• (t1, t2 : TObject ; h1, h2 : HObject
    | $TObject_1$.hList = {t1, t2, h1, h2}
      ∧ t1.hList = {}
      ∧ t2.hList = {}))
$\Leftrightarrow$
(∃ TObject : P T;
  HObject : P H
  $TObject_1$, t1, t2 : TObject
  h1, h2 : HObject
| (#{h : HObject | h ∈ $TObject_1$.hList } > 1
  ∧ #{t : TObject | t ∈ $TObject_1$.hList } > 1)
• ($TObject_1$.hList = {t1, t2, h1, h2}
    ∧ t1.hList = {}
    ∧ t2.hList = {}))

Existential Quantifier "∃"를 소거하기 위해 "***one-point rule***"을 사용할 수 있다. One-point rule은 전제 조건(predicate)의 변수에 특정 값을 직접 대입해서 전제 조건을 만족하는 표현식(expression)이 존재함을 보임으로써 Existential Quantifier를 소거하는 방법이다. One-point rule과 표현식 $TObject_1$.hList = {t1, t2, h1, h2}를 이용하여 위의 논리식을 다시 표현하면 다음과 같다.

| (#{h : HObject | h ∈ $TObject_1$.hList } > 1
  ∧ #{t : TObject | t ∈ $TObject_1$.hList } > 1)
  • ($TObject_1$.hList = {t1, t2, h1, h2}
      ∧ t1.hList = {}
      ∧ t2.hList = {}))
  ∧ {t : TObject | t ∈ $TObject_1$.hList } = { t1, t2 }

이때, (#{ h1, h2 } = 2 > 1 (#{ t1, t2 } = 2 > 1) 이므로,

$TImin_{3.1.1}$ [{ $TObject_1$, t1, t2 } / TObject, { h1, h2 } / HObject]는 참 값을 갖게 된다.

따라서

$\exists\ TT_{3.1.1} \cdot TImin_{3.1.1}$

이 증명 되었으며, 테스트 인스턴스 $TImin_{3.1.1}$ 이 테스트 템플릿 $TT_{3.1.1}$의 영역에 속하는 값임을 알 수 있다. 나머지 8개의 테스트 인스턴스들도 이와 같은 방법으로 해당 테스트 패턴의 영역에 속함을 증명할 수 있다.

5. 구조적 테스트 패턴 추출 적용 예

앞 장에서는 일반적인 합성 패턴 가운데 비교적 복잡한 1:N Recursive Connection 합성 구조의 테스트 패턴 추출에 대해 살펴 보았다. 본 장에서는 구조적 테스트 패턴 추출 방법을 객체지향 설계 패턴 중 하나인 Object Adapter 패턴[18]의 연결 구조에 적용한 예를 보여준다.

Object Adapter 설계 패턴의 연결 구조를 살펴보면 템플릿 클래스인 T 클래스 객체가 하나 이상의 Target 서

브 클래스 객체를 인스턴스 변수 target을 통해 참조한다. Target 클래스와 Adaptee 클래스 사이의 인터페이스 역할을 하기위해 Adapter가 Target 클래스로부터 속성을 상속 받아 Adaptee를 참조하는 형태로 삽입된다. 따라서 Adaptee 객체와 Adapter 객체는 1:1로 대응된다. 템플릿 메쏘드 tMethod()에서 Target의 후크 메쏘드 Request()를 호출하면 Adaptee 객체의 Specific Request()로 메쏘드 호출이 위임된다. 이런 연결 구조를 갖는 Object Adapter 패턴 클래스 다이어그램은 그림 5과 같다.

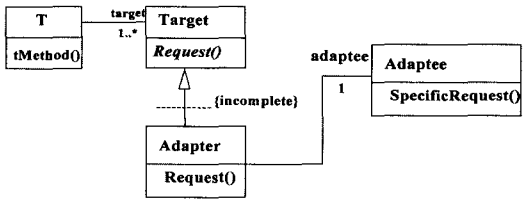
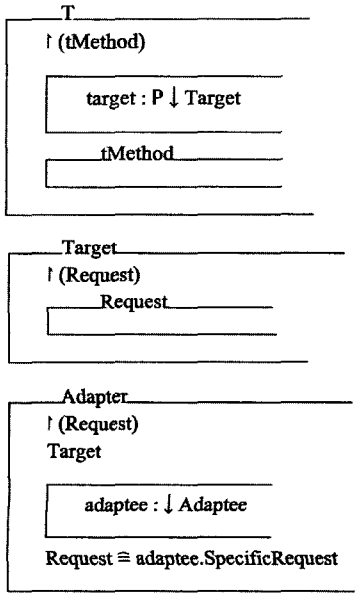
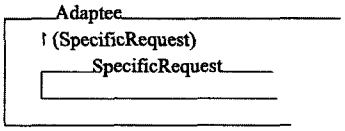


그림 5 Object Adapter 패턴 구조의 클래스 다이어그램

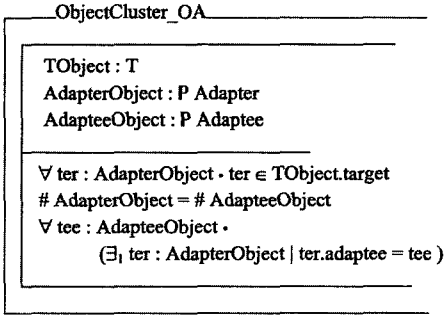
이때 템플릿 클래스인 T 클래스와 후크 클래스인 Target (또는 서브 클래스)의 합성 관계는 Pree[9]의 합성 패턴 가운데 1:N Connection의 형태이다. 이러한 형태로 이루어진 프레임워크의 가변부위를 테스트 하고자 할 때, 테스트 대상의 실행 환경을 선정하고 구성하기 위해 구조적 테스트 패턴 추출 방법을 적용할 수 있다.

Object-Z 스키마를 이용하여 Object Adapter 패턴 클래스들의 테스트 모델을 정형화하면 다음과 같다.





Target 클래스를 추상 클래스로 가정하면 Object Adapter(OA) 구조 객체들로 이루어진 객체 클러스터 스키마는 다음과 같다.



모든 Adapter 객체는 T 객체의 참조 변수에 의해 참조되고, 모든 Adaptee 객체는 각각 Adapter 객체의 참조 변수 adaptee와 1:1의 대응 관계를 갖게 된다. 위의 테스트 모델로부터 생성 가능한 객체 클러스터(VOS: Valid Object Structure)의 구조는 다음과 같고,

VOS≙[ObjectCluster_OA | # TObject.target > 0]

객체들을 생성했을 때 존재하는 Adaptee 객체의 개수에 의해 Adapter 객체의 개수가 결정되므로 이 객체의 개수에 따라 서로 다른 특성을 갖는 하위 영역으로 분할 될 수 있다. 그런데 Adaptee 객체의 개수는 Adapter 객체의 개수와 같으며, T 객체의 참조 변수 target의 cardinality와도 같다. 따라서 target에 할당된 Adapter 객체의 개수에 의한 cardinality_target을 분할 기준으로 적용한다.

| cardinality_target : HEURISTIC

T 클래스의 참조 변수 target에 할당될 수 있는 객체의 수가 1 또는 그 이상이라는 제약 조건(target에 할당되는 객체의 수가 1보다 작으면 Adapter 및 Adaptee 객체가 존재하지 않게 되므로 Object Adapter Pattern의 제약 조건에 위배된다.)이 있으므로, VOS를 분할하는 첫번째 단계의 분할에서는 객체 수가 1인 경우와 그 이상인 경우로 분할한다.

TTH_{OA} (VOS, cardinality_target) = {TT_1, TT_2}

TT_1 ≙ [VOS | # TObject.target = 1]

TT_2 ≙ [VOS | # TObject.target > 1]

TT_1은 객체 참조 변수 target의 cardinality가 1로

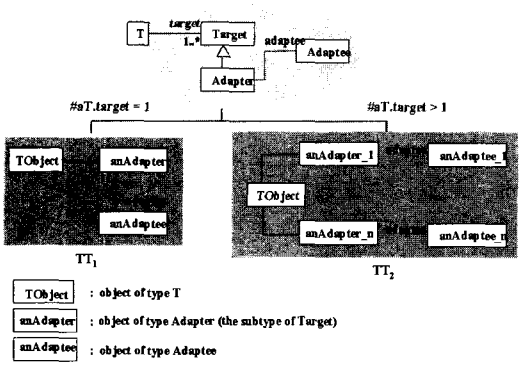


그림 6 Object Adapter CUT의 테스트 패턴

고정 되므로, 객체의 수 및 그 연결 구조가 고정되고, TT_2는 target의 cardinality가 1 이상 무한대 이므로 더 이상 경계값이 존재하지 않으며 TT_2에 의해 구성 가능한 객체 구조들은 동일한 특성을 갖는다고 할 수 있다.

따라서, TT_1, TT_2는 모두 더 이상 분할되지 않으므로 테스트 패턴이다. Object Adapter 테스트 모델은 한 번의 heuristic 적용에 의한 분할로 2개의 테스트 패턴(TT_1, TT_2)이 추출되었으며 도식화해서 표현하면 그림 6과 같다.

앞서 언급된 바와 같이 각각의 테스트 패턴에 최소값 분할 기준(TT_1 : #aT.target = 1, TT_2 : #aT.tartget = 2) 또는 다양한 인스턴스 선정 기준을 적용하면 1:N Recursive Connection의 경우와 마찬가지로 간단히 Object Adapter 구조의 프레임워크 가변부위에 대한 테스트 대상 인스턴스를 선정할 수 있으며, 선정된 인스턴스가 해당 테스트 패턴의 제약 조건들을 만족하는지 증명($\vdash \exists\ TT_1 \cdot TTmin_1$, $\vdash \exists\ TT_2 \cdot TTmin_2$) 할 수도 있다.

6. 결론 및 향후 연구 방향

객체지향 프레임워크는 다수의 응용 소프트웨어의 개발에 반복적으로 재사용될 목적으로 만들어지므로 높은 신뢰성이 필수적인 요구 사항이다. 높은 신뢰성을 위해서는 철저한 시험을 거쳐야 하며 특히, 재사용 시 변경 없이 그대로 사용하는 고정부위보다 재사용할 때마다 컴포넌트들을 생성, 개조, 대체, 추가해야 하는 가변부위에 대한 다양한 환경에서의 시험이 이루어 져야 한다. 그런데 프레임워크가 재사용되는 형태, 상황 및 응용 사례들은 매우 다양할 뿐만 아니라 이를 사전에 명시하고 예측하기가 쉽지 않으므로, 시험을 위한 시험 대상의 실행 환경도 다양하게 구성되어야 한다.

본 논문에서는 객체지향 프레임워크의 가변부위를 시험하기 위해 시험 대상을 정형 명세하고, 시험 대상의 구성 가능한 객체 구조들을 시험 목적에 따른 기준을 적용하여 동일한 특성을 갖는 부분 집합 즉, 테스트 패턴으로 분류하였다. 본 논문에서 제안한 방법은 테스트 대상 범위를 프레임워크의 가변부위로 압축하여 집중적으로 시험함으로써, 프레임워크가 새로운 환경에서 재사용되기 위해 개조, 구체화 될 때 가변부위에서 많이 발생하는 회귀결함을 미연에 방지하는데 기여할 수 있다. 또 다양한 시험 전략에 의한 분할 기준 적용을 가능하게 하며, 이렇게 추출된 테스트 패턴은 시험 데이타와 시나리오 선정의 기준이 된다. 특정 테스트 패턴에서 선정된 임의의 인스턴스는 해당 테스트 패턴을 대표하는 테스트 대상 실행 환경으로 실제 사용 형태를 예측하기 어려운 프레임워크 가변 부위에 대한 체계적인 시험에 효과적으로 사용될 수 있다.

기존에 진행되어 왔던 객체지향 프레임워크의 테스트에 관한 연구는 대부분이 테스트 케이스의 선정, 테스트 데이타 선정 및 생성에 주안점을 두고 있으나, 테스트 대상 인스턴스 분류 기준 및 선정 방법의 제시는 미비하다 할 수 있다. 따라서 본 논문은 수많은 테스트 대상 인스턴스들에 명확한 분류 기준을 적용한 분류 방법과 분류된 그룹으로부터 특정 인스턴스를 선정하는 이론적 접근 방법을 제시한다. 따라서, 본 논문에서 제시하는 방법은 기존에 연구되어온 테스트 프레임워크의 다양한 접근 방법 및 다양한 분야와 병행하여 테스트 효율을 향상시키기 위해 제안되었다.

테스트 모델과 각 단계별 분할 기준을 입력으로 받아서 테스트 패턴을 추출하는 테스트 패턴 생성기(Test Pattern Generator), 테스트 패턴과 인스턴스 선택 기준을 입력 받아서 각 패턴에 대한 인스턴스를 선정하는 테스트 인스턴스 선택기(Test Instance Selector), 인스턴스 명세로부터 실행 가능한 형태의 객체 클러스터 구조를 생성시켜주는 테스트 인스턴스 생성기(Test Instance Builder) 등의 구성으로 자동화가 이루어지면 독립적인 컴포넌트로서 시험 전용 소프트웨어(Tester Software)의 부품으로 삽입되어 사용될 수 있다. 이를 위해서 테스트 모델로부터 테스트 패턴을 효율적 또는, 자동적으로 추출하기 위한 각 단계별 과정의 일반화와 테스트 인스턴스 명세를 이용하여 실제 객체 구조를 자동 생성하는 방법이 연구되어야 하겠다.

참 고 문 헌

[1] M. E. Fayad, D. C. Schmidt, and R. E. Johnson, Building Application Frameworks, John Wiley & Sons, Inc., 1999

[2] S. Kirani and W. T. Tasi, "Method Sequence Spedification and Verification of Classes," Journal of Object-Oriented Programming, October 1994, pp. 28-38.

[3] G. Kim and C. Wu, "A Class Testing Technique Based on Data Bindings," Proceedings of the 1996 Asia-Pacific Software Engineering Conference, 1996, pp.104-109.

[4] A. S. Parrish, R. B. Borie and D. W. Cordes, "Automated Flow Graph-Based Testing of Object-Oriented Software Modules," Journal of Systems and Software, Volume 23, 1993, pp. 95-109.

[5] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch or Why its hard to build systems out of existing parts," Proceedings of 17th Int'l Conference on Software Engineering, Apr. 1995, pp. 179-185.

[6] Wirfs-Brock R.J. and Johnson R.E. "Surveying current research in object-oriented design," Communications of the ACM, 33(9), 1990.

[7] R. Helm, I. M. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," Proceedings of OOPSLA '90, Ottawa, Canada, 1990.

[8] B. Meyer, "Design by Contract," Advances in Object-Oriented Software Engineering, Prentice Hall, 1992, pp. 1-50.

[9] W. Pree, Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995.

[10] K. H. Chang, S-S. Liao, S. B. Seidman and R. Chapman, "Testing object-oriented programs: from formal specification to test scenario generation," Journal of System and Software, Volume 42, 1998, pp. 141-151.

[11] D. Hoffman and P. Strooper, "ClassBench: A Framework for Automated Class Testing," Software-Practice and Experience, May 1997, pp. 573-597.

[12] P. A. Stocks and D. A. Carrington, "Test templates: a specification-based testing framework," Proceedings of the 15th international conference on Software Engineering, May 17-21, 1993, Baltimore, MD, USA, pp. 405-414.

[13] P. A. Stocks and D. A. Carrington, "Test templates framework: A specification-based testing case study," Proceedings of Int'l Symposium on Software Testing and Analysis(ISSTA), June 1993, pp. 11-18.

[14] J. B. Wordsworth, Software Development with Z, Addison-Wesley, 1992.

[15] J. Rumbaugh, I. Jacobson and G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1998.

[16] G. Smith, The Object-Z Specification Language, Kluwer Academic, 1999.

[17] Robert V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 2000.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

김 장 래

1992년 ~1999년 고려대학교 전산학과 학사. 1999년 ~ 2001년 고려대학교 전산학과 석사. 2001년 ~ 현재 LG전자 핵심망 연구소 S/W 1Gr. 주임연구원

전 태 웅

1977년 ~ 1981년 서울대학교 계산통계학과 학사. 1981년 ~ 1983년 서울대학교 계산통계학과 석사. 1987년 ~ 1992년 Illinois Institute of Technology 전산과학 박사. 1983년 ~ 1987년 금성통신 연구소 주임연구원. 1992년 ~ 1995년 LG 산전 연구소 책임연구원. 1995년 ~ 현재 고려대학교 전산학과 부교수. 관심분야는 소프트웨어 테스팅, 소프트웨어 아키텍터, 객체지향 프레임워크, 실시간 소프트웨어 공학.