2000 년 한국정보처리학회 추계 학술발표논문집 제 6 권 제 2 호

컴포넌트 일반성 향상 기법

김철진, 김수동
숭실대학교 컴퓨터학과
e-mail : cjkim@selab.soongsil.ac.kr sdkim@computing.soongsil.ac.kr

An Improvement Technique of Component Generalization

Chul-Jin Kim, Soo-Dong Kim
Dept. of Computing, Soongsil University

요 약

소프트웨어를 개발하는데 미리 구현된 블록을 사용하여 소프트웨어 개발 비용과 시간을 단축할 수 있다. 이와 같이 미리 구현된 블록을 컴포넌트(Component)라고 하며 컴포넌트는 실행 단위로 개발자에게 인터페이스만을 제공하여 내부 상세한 부분을 숨기므로 쉽고 빠르게 어플리케이션을 개발할 수 있다. 그러나 인터페이스 만을 이용하여 시스템을 개발하는 컴포넌트는 범용적으로 많은 도메인에 사용될 수 있도록 컴포넌트를 개발해야 한다. 어플리케이션 개발자는 완전히 내부를 볼 수 없는 블랙 박스(Black Box) 형태의 컴포넌트를 원하며 개발 도메인의 특성에 맞게 속성 및 워크플로우(Workflow)의 변경을 원하기 때문에 워크플로우를 커스터마이즈(Customize)할 수 있는 기법이 제공되어야 한다. 이러한 커스터마이즈 기법에 따라 컴포넌트의 일반성이 좌우될 수 있다. 본 논문에서는 컴포넌트의 일반성을 향상시킬 수 있는 워크플로우 커스터마이즈 기법을 제시한다. 기존에 워크플로우를 변경한다는 것은 컴포넌트 내부를 개발자가 이해하고 코드 수준에서 수정해야 하는 화이트 박스(White Box)이지만, 본 논문에서는 워크플로우의 변경을 화이트 박스가 아니라 블랙 박스 형태로 컴포넌트 인터페이스 만을 이용해 커스터마이즈 할 수 있는 기법을 제시하며 이러한 기법을 통해 일반성을 향상 시킬 수 있도록 한다.

1. 서론

컴포넌트는 소프트웨어 개발의 산업화(Industrialization)를 의미하며 모든 소프트웨어의 기능 모듈을 컴포넌트로 부품화하여 소프트웨어의 대량 생산을 가능하게 할 수 있음을 의미한다[1]. 그러나 이러한 컴포넌트가 범용성을 가지고 있을지라도 특정 도메인에서 재사용될 때 도메인의 특성에 맞게 변경하기 위한 메커니즘이 필요하다. 기존의 커스터마이즈는 단지 컴포넌트의 속성 만을 변경했으며 주로 시각적 컴포넌트(Visual Component)에 집중된다.

컴포넌트를 그래픽 사용자 인터페이스 뿐만 아니라 비즈니스 측면에서 재사용하기 위해서는 속성의 변경과 컴포넌트 내부의 워크플로우를 변경하는 것이 가능해야 한다[2]. 본 논문에서는 컴포넌트의 일반성을 향상시키기 위해 컴포넌트 내부를 변경하지 않고 외부 인터페이스만을 가지고 변경하기 위한 워크플로우 커스터마이즈 기법을 제안한다.

2. 컴포넌트 모델

기존에 우리는 4th GL 인 비쥬얼베이직이나 델파이와 같은 언어에서 컴포넌트에 대해 많이 들어 왔지만 이러한 컴포넌트는 시각적인 컴포넌트인 블랙박스 형태의 컴포넌트로 컴포넌트에 대해 정의를 내리기에는 매우 제한적이다. 컴포넌트의 여러 정의들을 통해 공통적으로 컴포넌트가 의미하는 바를 일반적인 특성들을 중심으로 정의한다.

컴포넌트가 지니는 공통적인 특징은 모듈성, 인터페이스, 아키텍쳐, 컴포지션, 동적 바인딩, 커스터마이지 등으로 요약할 수 있다.

컴포넌트가 모듈성이 강하다는 것은 일반성이 강하다는 의미와 같으며 일반적인 컴포넌트는 여러 도메인에서 컴포넌트의 속성을 많이 변경하지 않고 하드웨어처럼 플러그 엔 플레이(Plug & Play)하여 소프트웨어를 개발할 수 있음을 의미한다.

기존의 라이브러리가 어플리케이션에 의해 사용될

때, 호출이 단지 어플리케이션에서 라이브러리로 단방향으로 이루어 지지만, 컴포넌트에서의 호출은 동적 바인딩에 의해 양방향으로 이루어 진다. 컴포넌트는 실행 시에 인터페이스를 통해 동적으로 어플리케이션에 연결될 수 있으며 다른 컴포넌트와도 동적으로 연결된다. 라이브러리가 개발 시스템에서 재사용될 때 재컴파일(Recompile)되어야 하는 반면 컴포넌트는 실행 시에 개발 시스템에 재컴파일 없이 동적으로 바인딩 되어 사용될 수 있다.

컴포넌트는 블랙박스 형태의 재사용 단위로 개발 시스템에서 컴포넌트를 사용하기 위해 컴포넌트 외부 인터페이스 만을 알면 되며 내부 구현 모듈을 인식할 필요가 없다. 개발 시스템 뿐만 아니라 컴포넌트 사이에도 인터페이스를 이용한다. 인터페이스는 컴포넌트에 의해 제공되는 서비스를 정의한다. 즉 인터페이스는 컴포넌트 내의 하나 이상의 클래스들에 의해 논리적으로 구현된다.

컴포넌트는 다른 컴포넌트나 프레임워크와 상호작용할 수 있도록 하기 위해 미리 정의된 아키텍쳐에 맞게 설계되고 구현되어야만 한다. 컴포넌트 기반 아키텍쳐는 컴포넌트를 첨가하거나 대치하여 기능적인 향상을 이룰 수 있도록 프레임워크 형태로 제공된다. CORBA 나 DCOM 과 같은 컴포넌트 아키텍쳐는 프레임워크 형태로 컴포넌트가 운영될 수 있도록 컴포넌트를 조정하거나 대신 작업을 처리해 준다. 컴포넌트는 운영될 수 있는 컴포넌트 아키텍쳐에 맞게 설계가 되어야 하며 해당 아키텍쳐에 의해 내부적으로 트랜잭션이나 컴포넌트의 상태 관리, 보완 관리 등과 같은 기능을 대신 처리해 준다. 컴포넌트는 자신의 아키텍쳐가 아닌 다른 컴포넌트 아키텍쳐에서는 운영되지 않는다. 그러나 컴포넌트가 아키텍쳐에 종속적인 특징을 가지고 있지만 CORBA 나 DCOM 과 같은 컴포넌트 모델에서는 서로의 컴포넌트 간에 상호 운영성을 위한 스펙을 발표하고 있다. 이와 같이 컴포넌트가 해당 아키텍쳐에 종속적인 특징을 가지고 설계되지만 다른 아키텍쳐에서도 상호 운영될 수 있도록 하는 추세이다[3,4,5].

플러그 앤 플레이 할 수 있는 컴포넌트는 인터페이스를 통해 컴포넌트를 조립한다. 이러한 조합은 다중의 컴포넌트들 간에 이루어 지는 행위를 설계하는 것이 필요하며 실행 시에 조립될 수 있어야 한다. 현재 소프트웨어 시장에서 컴포넌트 컴포지션을 위한 도구로 Visual Java (SUN), Visual Age (IBM)등이 있으나 아직까지 컴포넌트의 동적인 생성, 삭제 그리고 적절한 연결을 하지 못하며 시멘틱 수준(Semantic level)에서 타당성 체크를 제대로 해주지 못하고 있다[6].

개발자는 컴포넌트를 사용하여 어플리케이션을 개발할 때 컴포넌트의 속성을 변경하여 개발하려는 시스템에 컴포넌트를 적절하게 끼워 넣을 수 있다. 이러한 커스터마이즈는 일반적으로 컴포넌트 내의 인터페이스를 통해 데이터에 접근한다[7].

컴포넌트 모델은 컴포넌트의 기본적인 아키텍쳐, 컴포넌트의 인터페이스, 그리고 컴포넌트와 컨테이너 간의 상호작용을 위한 메커니즘을 정의한다. 이와 같이 컴포넌트 모델은 재사용할 수 있는 컴포넌트를 지원하기 위한 환경을 정의한다[8].

컴포넌트는 컨테이너 안에서 실행되며 컨테이너는 하나 이상의 컴포넌트를 위한 어플리케이션 컨텍스트를 제공한다. 또한 컴포넌트를 관리하고 제어하기 위한 서비스를 제공한다. 보통 클라이언트 컴포넌트들은 시각적인 컨테이너 안에서 실행하며 서버 컴포넌트들은 TP(Transaction Processing) 모니터, 웹 서버, 데이터베이스 시스템과 같은 서비스를 제공하는 비시각적인 컨테이너 안에서 실행된다[9].

3. 컴포넌트 일반성 관계

컴포넌트는 컴포넌트의 크기와 일반성과의 관계를 고려할 수 있다. 컴포넌트의 크기가 커지면 적용될 수 있는 범위가 좁아지며 따라서 일반성이 떨어진다. 이와 같이 컴포넌트가 가질 수 있는 여러 특성들간의 트레이드 오프(Trade Off)가 존재하며 이러한 특성들간의 트레이드 오프를 고려하여 적용 도메인의 범위에 맞게 컴포넌트를 선택할 수 있다.

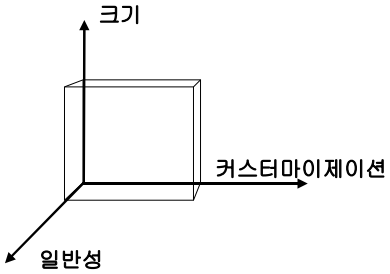


그림 1. 컴포넌트 일반성 관계 (A)

그림 1은 컴포넌트의 일반성, 크기 그리고 커스터마니즈와의 관계를 나타낸다. 컴포넌트의 크기가 커질수록 변경시켜야 할 부분이 많아지며 커스터마이즈 할 수 있는 기능이 많을수록 일반적이라기 보다는 종속적일 가능성이 높아진다.

산업계에서는 작은 컴포넌트 보다는 대용량의 컴포넌트를 원한다. 그러나 이러한 컴포넌트는 범용적으로 사용될 수 있는 일반성은 떨어진다. 일반성을 조금이라도 증가시키기 위해서는 어플리케이션 개발자들이 컴포넌트를 범용적으로 사용할 수 있도록 향상된 커스터마이즈 기능을 제공해야 한다.

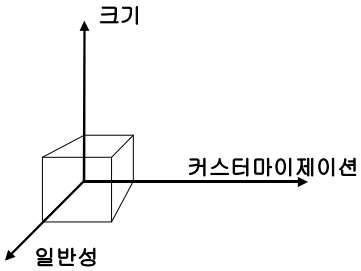


그림 2. 컴포넌트 일반성 관계 (B)

그림 2의 컴포넌트 일반성 관계는 컴포넌트의 일반성이 향상되려면 컴포넌트의 변경도 작아야 하며 크

기 또한 작은 컴포넌트이어야 함을 의미 한다. 이러한 컴포넌트는 주로 시각적인 컴포넌트들이라고 할 수 있다. 즉, 비시각적인 컴포넌트,비즈니스 컴포넌트는 여러 도메인에 사용될 수 있도록 커스터마이즈가 많아야 하며 따라서 일반성을 떨어진다[4,5].

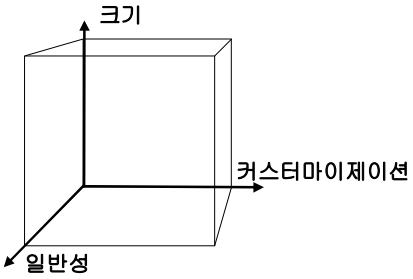


그림 3. 요구 컴포넌트 일반성 관계

그림 3은 컴포넌트의 사용자들이 원하는 형태의 컴포넌트라고 할 수 있다. 소형의 컴포넌트가 아니라 대형의 비즈니스 로직을 가진 컴포넌트로서 다양한 도메인에 쉽게 커스터마이즈 될 수 있는 컴포넌트라고 할 수 있다. 이러한 컴포넌트를 제공하기 위해서는 컴포넌트 개발자들이 컴포넌트의 내부에 유연성을 제공할 수 있도록 해야 한다.

4. 컴포넌트 일반화 기법

컴포넌트 사용자는 단지 컴포넌트의 인터페이스 만을 이용하면 컴포넌트의 워크플로우를 변경할 수 있다. 컴포넌트의 일반성을 향상시킬 수 있는 워크플로우 커스터마이즈 기법으로 3 가지 기법을 정의하고 각각의 기법에 대해 예를 통해 상세하게 설명한다.

첫번째 기법은 컴포넌트의 인터페이스 만을 통해 워크플로우를 변경하는 가장 간단한 기법으로 해당 컴포넌트의 인터페이스에 워크플로우를 좌우하는 클래스 명을 전달하는 기법이다. 두 번째 기법은 첫번째 기법에서 선택된 워크플로우가 흐르는 도중에 선택되지 않은 다른 워크플로우의 중간으로 흘러가도록 하는 기법으로 각각의 워크플로우에 대한 전체적인 흐름을 개발자가 숙지해야 한다. 그러나 다양한 많은 흐름을 제공하므로 컴포넌트의 일반성을 첫번째 기법보다 더욱 향상시킨다고 할 수 있다. 세 번째 기법은 첫번째 기법과 두 번째 기법에서 컴포넌트 내의 워크플로우 만을 이용하여 흐름을 제어하는 것과는 달리 개발자에 의해 새로운 형태의 워크플로우를 동적으로 첨가할 수 있는 기법이다. 물론 첨가할 수 있는 흐름에 대한 클래스는 일반화된 형태인 추상화 클래스를 따라야만 한다. 즉 컴포넌트 내부에서 정의 된 추상 클래스를 상속 받은 클래스 만을 입력하여 흐름을 변경할 수 있는 기법이다. 따라서 세 번째 기법이 가장 일반성을 향상시키며 컴포넌트를 개발하는 개발자에게도 부담을 덜어 준다.

워크플로우 커스터마이즈 기법에 대한 정의를 바탕으로 각각의 기법을 통해 어떻게 일반성이 향상되는지 알아 본다.

첫번째 워크플로우 커스터마이즈 기법은 허용 가능한 워크플로우를 인터페이스를 통해 선택하여 커스터마이즈하는 기법이다.

컴포넌트 인터페이스의 매개변수에 워크플로우를 변경하기를 원하는 클래스 명을 명시해 주어야 한다. 컴포넌트 사용자는 컴포넌트 문서를 통해 어떤 클래스들이 변경이 가능한지를 참조한다.

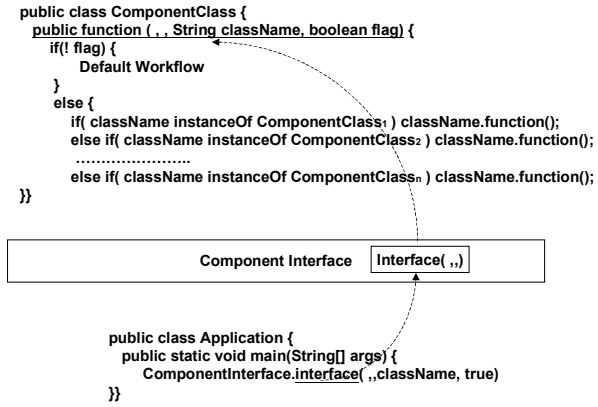


그림 4. 허용 가능한 워크플로우 선택

그림 4은 컴포넌트 인터페이스를 통해 어플리케이션에서 컴포넌트의 워크플로우를 변경하는 과정을 보여 준다. 어플리케이션은 단지 컴포넌트의 인터페이스를 호출하면 컴포넌트 인터페이스에서 내부적으로 변경을 원하는 클래스로 흐름을 변경해 준다. 변경을 원하지 않을 경우는 flag 에 'false'를 입력하여 기본 워크플로우를 이용하도록 한다. 어플리케이션에서 컴포넌트의 인터페이스에 원하는 클래스 명을 입력하면 그 입력된 클래스에 의해 흐름이 변경이 된다.

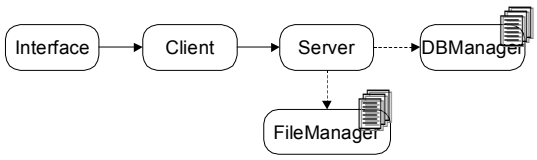


그림 5. 선택 가능한 워크플로우 클래스

인터페이스를 통해 입력할 수 있는 변경 가능한 클래스 명은 그림 5와 같이 여러 워크플로우의 첫 번째 클래스 명인 'FileManager'나 'DBManager'를 입력한다. 즉 인터페이스를 통해 두 클래스의 흐름을 좌우하는 클래스명을 입력하면 개발자가 원하는 흐름으로 흘러가도록 할 수 있다. 물론 그 내부의 함수명은 같도록 추상클래스에서 정의된 함수를 사용한다고 정의한다.

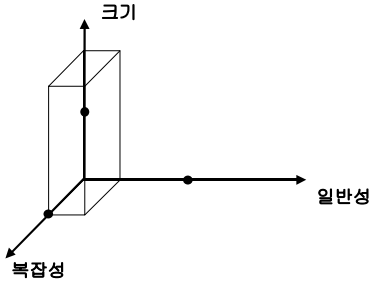


그림 6. 워크플로우 선택 기법의 일반성 관계

그림 6은 첫 번째 기법인 워크플로우 선택 기법에서 일반성과 복잡성, 크기에 관한 관계를 나타낸다. 이 기법은 컴포넌트의 복잡성은 컴포넌트 사용자에게 다소 약화시켜 주지만 일반성 면에서 여러 도메인에 적용하기 에는 다소 약하다. 그러나 컴포넌트의 크기 측면에서는 대규모의 컴포넌트를 구현하는데는 적합하다.

두 번째 워크플로우 커스터마이즈 기법은 허용 가능한 워크플로우의 순차적인 흐름을 변경하는 기법이다. 그림 7과 같이 컴포넌트 내부 워크플로우는 인터페이스를 통해 워크플로우의 흐름 중에 다른 워크플로우로 변경할 수 있다. 이러한 변경은 개발자에게 많은 복잡성을 제공하지만 다양한 변경 가능한 흐름을 지원할 수 있다. 그러나 이러한 허용 가능한 워크플로우에서의 변경은 컴포넌트 내부를 개발자에게 어느 정도 공개해야만 하므로 블랙 박스가 아니라 화이트 박스 형태로 컴포넌트를 제공해야 한다. 개발자는 컴포넌트의 문서를 통해 워크플로우 중에 변경 가능한 흐름을 파악하고 개발해야 한다.

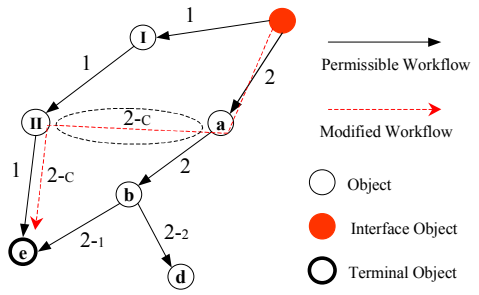


그림 7. 허용 가능한 워크플로우의 변경

그림 7과 같이 허용 가능한 흐름 '1', '2' , '2_{-1}','2_{-2}' 가 있다고 가정할 때, 컴포넌트 사용자가 컴포넌트의 문서를 통해 흐름 '2'의 'a' 객체에서 흐름 '1'의 'II' 객체로 전달되기를 원할 경우 이 기법은 이러한 객체들을 컴포넌트의 인터페이스를 통해 전달한다. 이렇게 전달된 객체들은 컴포넌트 내부의 흐름을 결정한다.

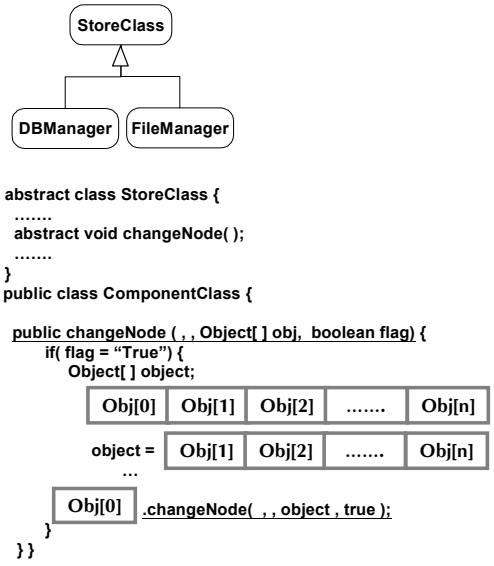


그림 8. 워크플로우 변경 기법

그림 8는 추상 클래스 'StoreClass'와 하위 클래스 'DBManager'이며 'DBManager' 클래스로 허용된 워크플로우를 변경하는 방법을 보여준다. 'DBManager' 클래스의 'changeNode(Object[] object)'는 변경을 원하는 일련의 객체 배열을 넘겨 받는다. 이런 객체들이 컴포넌트 워크플로우의 흐름이다. 이 워크플로우의 순서는 'DBManager'에 해당하는 처리를 수행하고 ('searchData(id,pw)'), 다음 객체(object[0])로 다음 워크플로우를 결정하는 객체 배열을 전달한다. 이러한 객체들은 허용 가능한 워크플로우 사이의 흐름을 변경 가능하게 한다.

```
public class ComponentClass {
  public function ( , , Object[ ] object,  boolean flag) {
     if( flag = "True") {
        Object[] obj;
        for( int i = 1; i < object.length() ; i++) {
              obj[i-1] = object[i];
        }
        object[0].changeNode( obj )      }}}
```

그림 9. 워크플로우 변경 클래스

그림 9는 어플리케이션에서 컴포넌트를 사용할 경우, 'ComponentClass'의 'function()'을 사용해 원하는 일련의 객체들을 입력할 수 있다. 배열 내의 객체들은 자신 객체의 'changeNode()'를 수행하며, 그 함수 내에 객체 자체의 특수한 기능을 수행한다. 기능을 처리한 후에 나머지 객체들을 새로운 객체 배열로 만들어 다음 객체로 전달한다. 이러한 워크플로우를 나타내는 객체 배열을 컴포넌트 인터페이스에 입력하기 위해 컴포넌트 내부의 객체간의 구조와 기본 흐름을 컴포넌트 사용자가 잘 이해하고 있어야 한다.

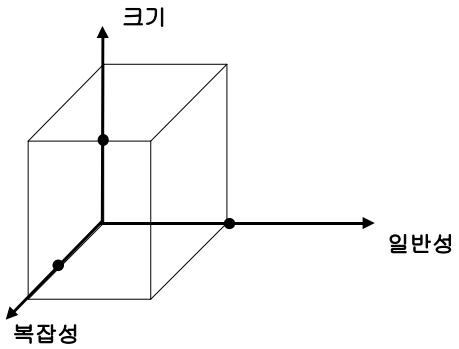


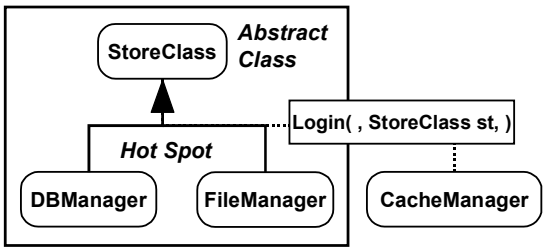
그림 10. 워크플로우 변경 기법의 일반성 관계

그림 10과 같이 이 기법은 첫 번째 기법에 비해 복잡성은 증가 하지만 첫 번째 기법에서 제공하는 허용 가능한 흐름들 과의 조합을 통해 다양한 흐름을 제공할 수 있으므로 일반성이 첫 번째 기법 보다 좋다고 할 수 있다. 그러나 이러한 변경은 특정 비즈니스 로직 부분에 적용하기 보다는 트랜잭션 서비스, 데이터베이스 연결 등과 같이 수평적인 서비스를 제공하는 컴포넌트에 적용하는 적절할 것이다.

세 번째 워크플로우 커스터마이즈 기법은 동적으로 워크플로우 클래스를 입력하거나 새롭게 생성한 클래스를 입력하여 워크플로우를 변경하는 기법이다. 워

크플로우의 변경이 많은 부분(Hot Spot)의 클래스들은 추상클래스(Abstract Class)로 추출하여 동적으로 워크플로우를 변경할 수 있도록 한다. 인터페이스는 추상클래스 타입으로 객체를 받아 동적으로 클래스를 변경할 수 있다.

그림 11와 같이 변경이 가능한 부분에 대해 추상클래스로 만들고 그 추상클래스 내부에 있는 함수는 추상함수로 선언하여 하위 클래스에서 반드시 정의하도록 한다. 이렇게 하위 클래스에서 반드시 정의해야 하는 함수는 하위 클래스들 간에 공통적으로 일치해야 하며 일정한 패턴으로 컴포넌트 개발자가 정의해야 한다. 이렇게 정의된 클래스들은 컴포넌트 내부에서 인터페이스 통해 접근할 수 있도록 한다. 어플리케이션 개발자에 의해 접근할 경우 인터페이스로 원하는 클래스를 입력하면 된다.




```
abstract class StoreClass {
  abstract void searchData(String id, String pw );
}
public class CacheManager extends StoreClass {
  public void searchData(String id, String pw ){ ... }
}
public class ComponentClass {
  public void addChangeWorkflow(
      StoreClass storeclass, String  id, String pw) {
      StoreClass.searchData(id, pw);
  }
}
```

OO Substitution

그림 11. 동적 워크플로우 변경

컴포넌트 인터페이스는 각각의 클래스들을 동적으로 입력 받기 위해 입력 매개변수의 타입을 추상클래스로 선언한다. 추상클래스로 선언하므로 어떤 클래스가 입력되든지 객체지향의 대치성(Substitution)에 의해 입력된 객체를 사용할 수 있다. 기존에 형성된 워크플로우의 고정된 클래스를 입력할 수 있을 뿐만 아니라 새롭게 생성된 클래스도 입력하여 워크플로우를 변경할 수 있다.

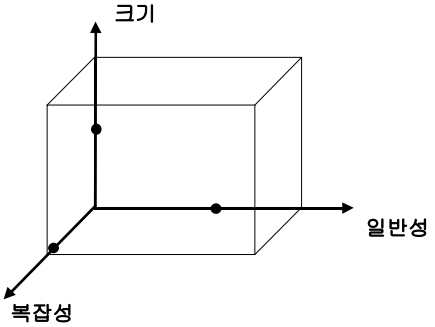


그림 12. 동적 워크플로우 변경 기법의 일반성 관계

그림 12와 같이 세 번째 기법은 컴포넌트 사용자 , 즉 어플리케이션 개발자가 컴포넌트 외부에서 원하는 컴포넌트 클래스를 생성하여 컴포넌트 내부로 입력시키므로 다양한 도메인에 사용될 수 있으므로 일반성이 더욱 향상된다. 또한 컴포넌트 내부가 첫 번째 기법과 두 번째 기법에 비해 복잡하지 않으므로 최적의 컴포넌트 워크플로우 커스터마이제이션 기법이라고 할 수 있다.

본 논문의 컴포넌트 커스터마이즈 기법을 적용하기 위해 그림 13와 같은 컴포넌트 모델을 통해 컴포넌트를 개발한다. 본 컴포넌트 모델은 자바빈즈 컴포넌트 모델에 기반하며 내부에 여러 컴포넌트 클래스들로 구성된다. 각각의 클래스들은 컴포넌트 개발자에 의해 개발되며 클래스들을 명시한 적하목록(Manifest) 파일 통해 패키지화 된다. 컴포넌트 내부의 클래스들을 접근하기 위해 컴포넌트 인터페이스를 이용하며 일반 데이터에 대한 인터페이스와 내부 워크플로우를 커스터마이즈하기 위한 인터페이스로 구성된다.

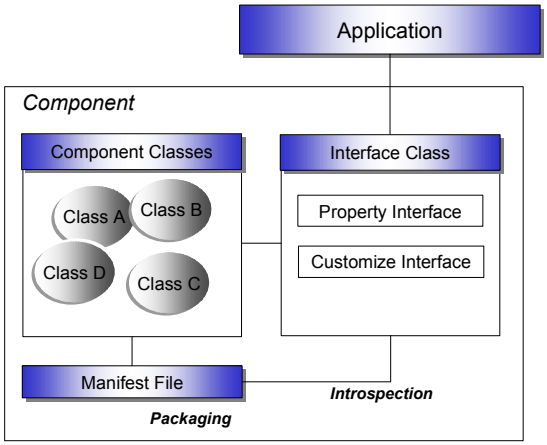


그림 13. 컴포넌트 모델

컴포넌트 사용자가 어플리케이션을 개발할 경우 컴포넌트 인터페이스를 사용하며 컴포넌트 인터페이스를 표현하는 방법은 그림 14와 같이 나타낼 수 있다.

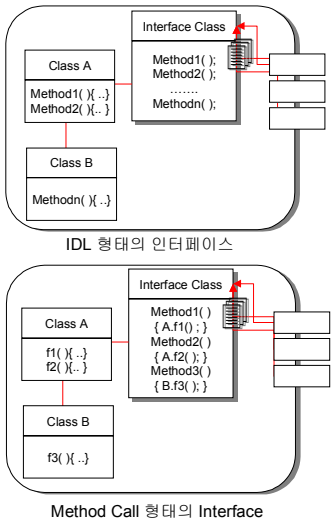


그림 14. 컴포넌트 인터페이싱 기법

IDL 형태의 인터페이스 기법은 인터페이스 클래스의 함수들은 구현하지 않고 IDL(Interface Definition Language)형태로 함수의 선언만 하며 내부 클래스들에서 IDL 로 정의된 함수들을 구현한다. Method Call 형태의 인터페이스 기법은 인터페이스 클래스에서 컴포넌트 내부 클래스들의 함수를 직접 호출하는 기법이다. 이 때 인터페이스 클래스는 IDL 형태로 함수가 선언된 되어 있는 클래스가 아니라 객체를 생성할 수 있는 컨크리트(Concrete) 클래스이다. 본 논문에서 이용하는 컴포넌트 인터페이스 방법은 두 번째 방법을 이용하여 컴포넌트 내부에 컨크리트 클래스 포함시킨다.

5. 결론 및 향후 연구 과제

지금까지 비즈니스 측면에서 컴포넌트의 일반성을 향상시킬 수 있는 컴포넌트 워크플로우 커스터마이제이션 기법을 제시했다. 워크플로우 커스터마이즈 기법은 기존의 제한된 범위의 커스터마이즈 기법을 확장시켰으며 컴포넌트를 하위 수준에서의 재사용이 아니라 비즈니스 측면에서 재사용될 수 있도록 한다. 컴포넌트의 워크플로우를 커스터마이즈 하기 위한 기법으로 단순하게 객체 명 만을 전달하는 흐름선택 방식과 동적으로 객체를 추가하여 컴포넌트 내부 객체들 간의 흐름을 변경하는 동적 변경 방식이 있다. 이러한 방식의 커스터마이즈 기법은 완전하게 블랙 박스 형태로 컴포넌트 워크플로우를 쉽게 커스터마이즈할 수 있으며 비즈니스 측면에서 적용 범위를 확장 시킨다.

표 1. 일반성 향상 기법 비교

일반성 향상기법 / 요소	*Select Workflow*	*Modify Workflow*	*Adding Workflow*
일반성	General	High	Excellent
복잡성 (컴포넌트 사용자)	Simple	Complex	Simple
복잡성 (컴포넌트 개발자)	Simple	Complex	Simple
적용 도메인	Extensive	Limited	Extensive

표 1은 일반성을 향상시키기 위한 워크플로우 커스터마이제이션 기법들 간의 일반성과 복잡성을 비교한 것이다. 첫 번째 기법인 워크플로우 선택 기법은 복잡성이나 적용 도메인 측면에서는 단순하고 다양한 범위에서 사용될 수 있지만 아주 범용적으로 넓은 범위에 사용되기에는 한계가 있다. 반면 두 번째 기법은 일반성은 좋지만 컴포넌트 개발자나 사용자 모두에게 높은 복잡도로 인해 부담을 줄 수 있다. 따라서 이러한 두 기법들의 일반성과 복잡성을 모두 보완해 줄 수 있는 기법으로 세 번째 기법인 동적인 워크플로우 추가 기법이다. 이 기법은 외부에서 워크플로우를 제시할 수 있는 클래스를 컴포넌트 내부로 전달할 수 있으므로 일반성이 더욱 향상될 수 있으며 컴포넌트 개발자나 컴포넌트 사용자 모두에게 복잡성을 감소시켜 쉽게 개발할 수 있도록 한다.

향후 연구과제는 여러 컴포넌트들을 결합하여 대용량의 복합 컴포넌트를 구현할 경우 컴폰넌트 간의 커스터마이즈 기법을 연구한다[10]. 컴포넌트 간에 원시 데이터만 전달되는 것이 아니라 객체가 전달되므로 컴포넌트 간에 각각의 컴포넌트에서 정의한 객체를 다른 컴포넌트에서 전달 받을 때 적절할 커스터마이즈 기법의 연구가 필요하다. 또한 좀더 범용적으로 사용될 수 있는 컴포넌트 개발 기법과 커스터마이즈 기법의 연구가 필요하며 향후 ASP(Application Service Provider) 사업의 발전과 더불어 컴포넌트 유통에 사용될 수 있는 컴포넌트 커스터마이즈 기법을 연구한다.

참고문헌

[1] Alan W. Brown, Kurt C. Wallnau, The Current State of CBSE, IEEE Software, Sep/Oct 1998.

[2] Robert Orfali, Dan Harkey, Client/Server Programming with Java and CORBA, 2nd ed., John Wiley & Sons, Inc., 1998.

[3] Eduardo Pelegri-Llopart, Laurence P.G. Cable, 'How to be a Good Bean', Sun Microsystems, Inc., Sep. 1997.

[4] Jun Han, 'Characterization of Components', International Workshop on Component-Based Software Engineering 1998.

[5] Mikio Aoyama, 'New Age of Software Development : How Component-Based Software Engineering Changes the Way of Software Development', International Workshop on Component-Based Software Engineering 1998.

[6] Digre T., "Business Object Component Architecture," *IEEE Software*, pp.60-69, September 1998.

[7] Mary C., Kathy W., Alison H., The Java Tutorial Continued, Addion-Wesley, 1999.

[8] Klaus Bergner, Andreas Rausch, Marc Sihling, 'Componentware – The Big Picture', International Workshop on Component-Based Software Engineering 1998.

[9] Brown A. W. and Wallnau K. C., "The Current State of CBSE," *IEEE Software,* pp.37-46, Sept./Oct. 1998.

[10] Short K., *Component Based Development and Object Modeling,* Sterling Software, 1997.