컴포넌트 정적/동적 커스터마이제이션 기법
(The Static and Dynamic Customization Technique of Component)

김 철 진 † 김 수 동 ††
(Chul Jin Kim) (Soo Dong Kim)

요 약 컴포넌트 기반 어플리케이션 개발(CBD: Component Based Development)은 Time-To-Market을 위한 필수적인 기법이며 컴포넌트를 이용해 다양한 도메인의 어플리케이션을 개발하기 위해서는 재사용성이 높은 컴포넌트가 제공되어야만 한다. 컴포넌트의 재사용성을 높이려면 다양한 도메인의 요구사항을 분석하여 개발해야 한다. 그러나 개발하려는 컴포넌트에 해당하는 다양한 도메인의 모든 요구사항을 분석해 컴포넌트 내에 포함한다는 것은 많은 부하를 주며, 또한 여러 도메인의 공통 기능을 가진 일반적인 컴포넌트 만 제공하는 것은 개발자가 개발해야 하는 다른 영역이 존재하므로 Time-To-Market을 이루기가 쉽지 않다. 이와 같이, 컴포넌트 개발(CD : Component Development) 시점에 다양한 도메인의 요구사항 분석을 통해 일반적인 컴포넌트를 개발하는 것이 재사용성이 높다고는 볼 수 없으며 이러한 컴포넌트는 공통적인 기능을 가지고 있는 컴포넌트 일뿐이며 공통 영역 외의 영역은 또 다른 개발의 부담을 준다.

따라서 본 논문에서는 공통 컴포넌트를 포함하여 특정 영역의 컴포넌트를 재사용하기 위한 컴포넌트 커스터마이제이션 기법을 제안한다. 컴포넌트의 데이터 속성(Attribute), 기능(Behavior), 그리고 메시지 흐름(Message Flow)에 대한 변경 가능성을 제공하여 컴포넌트의 재사용성을 높일 수 있다. 본 커스터마이제이션 기법은 개발된 컴포넌트들을 통합하거나 컴포넌트 내에 새로운 기능을 제공할 수 있도록 컴포넌트 내의 메시지 흐름을 변경할 수 있다. 또한 컴포넌트 내에 존재하는 클래스를 다른 클래스로 교환하거나 통합된 컴포넌트를 다른 기능의 컴포넌트로 교환할 수 있는 기법을 제공하여 다양한 도메인의 요구 사항을 수용할 수 있도록 한다. 이와 같이 본 커스터마이제이션 기법은 공통 기능의 컴포넌트 뿐만 아니라 특정 영역의 컴포넌트에 대한 재사용성도 확보할 수 있다.

키워드 : 컴포넌트, 컴포넌트 인터페이스,가변성, 커스터마이제이션, 행위, 메시지 흐름

Abstract The CBD (Component Based Development) is a requisite technique for the Time-To-Market, and a highly reusable component should be provided to develop a variety of domain applications with the use of components. To increase the reusability of components, they should be developed by analyzing requirements of many different kinds of domains. However, to analyze requirements of a variety of domains related to the components to be developed and to include them inside the components will give burden to developers. Also, providing only general components that have common facilities for the several domains is not easy to accomplish the time-to-market since there are other domains that the developers have to develop. As such, developing common component through the analysis of several domains at the time of the CD (Component Development) does not always guarantee high reusability of the component, but gives burden to developers to develop another development since such components have common functions.

Considering this, this paper proposes the component customization technique to reuse common components as well as special components. The reusability of the component can be increased by providing changeability of the attribute, behavior and message flow of the component. This customization technique can change the message flow to integrate developed components or to provide new functions within the component. Also, provides a technique to replace the class existing within the component with other class or to exchange the integrated component with the component having a different function so

† 학생회원 : 숭실대학교 컴퓨터학과
cjkim@selab.soongsil.ac.kr
†† 종신회원 : 숭실대학교 컴퓨터학과 교수
sdkim@computing.soongsil.ac.kr
논문접수 : 2002년 1월 7일
심사완료 : 2002년 6월 27일

that requirements from a variety of domains may be satisfied. As such, this technique can accept the requirements of several domains. As such, this customization technique is not only the component with a common function, but it also secures reusability components in the special domain.

Key words : Component, Component Interface, Variability, Customization, Behavior, Message Flow

1. 서 론

1.1 연구 배경

컴포넌트는 이제 생소한 말이 아니면 소프트웨어 개발의 필수적인 요소로 여겨지고 있다. 객체지향 기법이 소프트웨어 개발을 혁신적으로 향상시켜주지 못했기 때문에 80년대 말부터 소개되었던 개념인 컴포넌트 기법을 수용하게 되었다[1]. 그 이유 중에 하나는 객체라는 단위가 개발자들에게 제공되는 범위가 너무나 작기 때문에 개발의 부담을 그렇게 많이 감소시켜 주지 못했기 때문일 것이다. 컴포넌트는 업무의 흐름을 가지고 있으며 객체들을 조합한 기능 단위로 제공되므로 개발의 부담을 현저하게 감소시켜 준다. 컴포넌트 개념은 소프트웨어 개발을 조립의 개념으로 발전시켰으며 외부에서 개발된 컴포넌트도 조립하여 쉽게 소프트웨어를 개발할 수 있도록 하는 구조가 가능하게 되었다. 조립된 소프트웨어는 도메인의 업무 범위에 맞게 다른 컴포넌트로 교체하거나 컴포넌트를 추가하여 도메인의 요구사항을 빠르고 쉽게 충족시킬 수 있다[2,3,4].

컴포넌트는 빠른 시간 내에 어플리케이션을 개발하기 위한 개발 블록(Building Block)으로 컴포넌트 이용자(Component User)들에게 컴포넌트 인터페이스(Component Interface)와 컴포넌트 스펙(Component Specification)을 제공한다. 컴포넌트 인터페이스를 이용해 다양한 도메인의 요구 사항을 충족시키기 위해서는 컴포넌트 내부에 다양성을 제공해야 한다. 그러나 이러한 다양성을 제공하기 위한 가변성을 설계하기가 어려우며 설계된 가변성을 적용하기 위한 기법들이 존재하지 않기 때문에 가변성 부분에 대해 컴포넌트를 공개하여 내부를 변경하는 실정이다. 따라서 본 논문에서는 컴포넌트에 다양성을 제공하기 위해 다양한 도메인을 분석하여 컴포넌트를 설계하기 위한 기법보다는 다양한 도메인의 요구사항을 수용할 수 있는 장치를 제공하기 위한 컴포넌트 커스터마이제이션 기법을 제안한다. 본 논문에서 제안하는 커스터마이제이션의 범위는 컴포넌트 내부의 속성, 행위, 그리고 메시지 흐름에 해당하는 커스터마이제이션 기법을 제안한다.

2. 관련 연구

2.1 컴포넌트 모델

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[5]. 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[6,7].

컴포넌트는 블랙박스 형태의 재사용 단위로 개발 시스템에서 컴포넌트를 사용하기 위해 컴포넌트 인터페이스 만을 알면 되며 내부 구현 모듈을 인식할 필요가 없다. 인터페이스는 컴포넌트에 의해 제공되는 기능을 정의하며 컴포넌트 내의 하나 이상의 클래스들에 의해 논리적으로 구현된다. 컴포넌트 인터페이스는 외부에 기능을 제공하는 역할뿐만 아니라 컴포넌트를 관리하고 제어하는 역할을 한다. 주로, 컴포넌트 사용자에 의해 컴포넌트를 조합하거나 커스터마이제이션하기 위한 기능을 제공할 수 있다. 제공되는 서비스에 한정하여 인터페이스를 정의하면 다양한 도메인에 적용될 때 상이한 요구사항을 충족시키기 위해 컴포넌트 내부를 변경해야하는 문제가 발생할 수 있다.

2.2 CBD 방법론

CBD(Component Based Development) 방법론들은 커스터마이제이션의 필요성에 대해 언급하고 있으며 개념적인 수준에서의 접근 방법을 제시하고 있다. 커스터마이제이션에 대해 언급하고 있는 방법론으로 Catalysis와 Componentware는 컴포넌트의 가변성(Variation)을 정의하여 커스터마이제이션에 대해 언급하고 있다.

Catalysis의 프로세스는 요구사항 분석, 시스템 스펙, 아키텍쳐 설계, 그리고 컴포넌트 내부 설계 단계로 구성되며 설계부터 소스 코드에 이르는 전과정의 추적성(Traceability)과 정확성(Precision)을 보장하고 재사용 측면에서는 소스 코드뿐만 아니라 설계 산출물까지 재사용할 수 있도록 한다[4].

Catalysis에서는 컴포넌트를 변경하기 위한 방법으로 "Plug-Point"를 정의하여 여러 도메인에서 공통적인 부분으로 각각 특성화된 로직을 플러그 인(Plug-In)할 수 있는 부분을 정의한다.

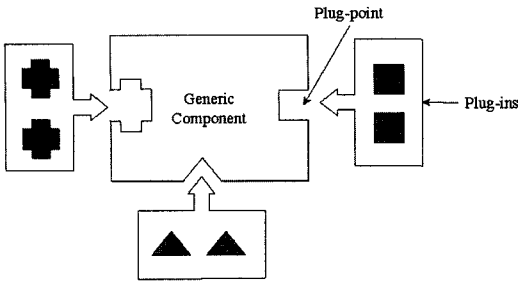


그림 1 Plug-Points

그림 1와 같이 일반 컴포넌트(Generic Component)는 "Plug-Point"를 정의하고 적용되는 도메인의 요구 사항에 맞게 적절한 "Plug-Ins"에 해당하는 모듈이나 컴포넌트를 선택하여 플러그 인 하여 일반 컴포넌트을 변경할 수 있다. Catalysis는 "Required Interface"를 정의하는 절차를 포함하고 있는데 이 인터페이스가 "Plug-Point"를 제공하는 역할을 하며 커스터마이제이션을 위한 가변성을 정의하기 위한 요소가 될 수 있다. 일반 컴포넌트에서 서비스로 제공되는 인터페이스는 "Provided Interface"로 정의한다.

Componentware의 프로세스는 분석, 비즈니스 설계, 기술 설계, 스펙, 그리고 구현 단계로 구성되어 있다. Componetware에서는 이러한 단계를 조합하여 다양한 형태의 프로세스를 구성할 수 있도록 프로세스 패턴들을 제공한다[8]. Componentware는 커스터마이제이션을 위해 Catalysis의 "Required Interface"와 유사한 "Import Interface"를 정의하고 있다. 이 인터페이스도 컴포넌트 외부에 기능을 요구하는 인터페이스로 다양한 요구사항을 충족시킬 수 있도록 할 수 있다.

앞의 CBD 방법론인 Catalysis나 Componentware는 커스터마이제이션을 위한 상세한 절차나 기법보다는 개념적인 적용 방법에 대해서 언급하고 있다. 따라서 본 연구에서 컴포넌트 커스터마이제이션에 대한 구체적인 설계 기법이나 절차에 대해 고려해 본다.

3. 커스터마이제이션 기법(Customization Technique)

본 논문에서 제안하는 컴포넌트의 커스터마이제이션은 3가지 측면에서 고려할 수 있다. 컴포넌트의 속성, 기능, 그리고 메시지 흐름으로 나눌 수 있다. 컴포넌트 속성 커스터마이제이션 기법은 컴포넌트 내에 존재하는 데이터 속성에 대한 변경을 위한 기법이며, 컴포넌트 기능 커스터마이제이션 기법은 컴포넌트에서 제공되는 기능에 대한 변경이나 추가를 위한 기법이다. 컴포넌트 메시지 흐름 커스터마이제이션 기법은 컴포넌트 내의 객체들 간의 메시지 흐름에 대한 동적인 변경이나 컴포넌트들 간의 메시지 흐름에 대한 동적인 변경을 제공하기 위한 기법이다.

3.1 속성 커스터마이제이션(Attribute Customization)

컴포넌트 속성 커스터마이제이션 기법은 컴포넌트 내에 존재하는 일반 비즈니스 객체나 데이터베이스에 관련된 객체의 속성을 변경하기 기법이다. 본 기법은 컴포넌트 변경 시에 컴포넌트 내의 속성을 변경할 경우 관련된 다른 객체들과 일관되게 변경되도록 하기 위한 커스터마이제이션 기법이다.

그림 2에서와 같이 컴포넌트와 관련된 데이터 속성은 프리젠테이션 계층(Presentation Layer), 프로세스 계층(Process Layer), 데이터 계층(Data Layer)에 걸쳐 존재하며 서로 연관관계를 가지고 있다.

본 논문에서 프리젠테이션 계층은 웹 기반이며 웹 화면을 개발하기 위해 JSP Tag Library를 이용하는 것을 전제로 한다. 프리젠테이션 계층에 속성 데이터를 표현하기 위한 태그(Tag) 객체('Object Tag')는 프로세스 계층의 메타 정보('Object.XML')를 이용하여 화면에 표현할 수 있다. 이러한 메타 정보는 컴포넌트('Entity Bean')의 속성을 나타내는 데이터 객체('Data Object')에 대한 메타 정보이다. 프로세스 계층의 컴포넌트인 'Entity Bean'은 데이터 속성을 나타내기 위해 'Data Object'를 상속받는다. 프로세스 계층의 'Data Object'는 데이터 계층의 데이터베이스 스키마 메타 정보('Schema.XML')를 이용하여 속성을 변경할 수 있으며 반대의 경우도 가능하다.

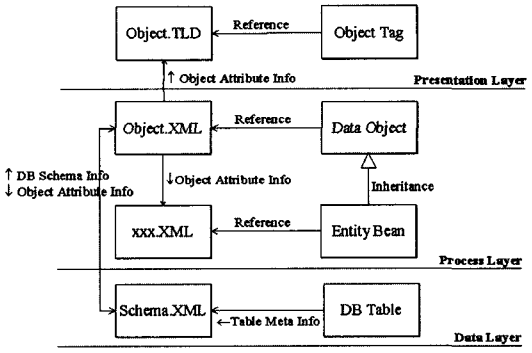


그림 2 속성 커스터마이제이션 구조

이와 같이 각각의 계층은 동일한 데이터 속성을 중복해서 관리하고 있으므로 중복 관리되는 데이터 속성을

하나로 관리하거나 변경 시 각 계층의 데이터 속성에 대한 설정 정보를 동일하게 적용되도록 하여 일관되게 변경하도록 할 수 있다. 따라서 각 계층의 데이터 속성을 나타내는 설정 정보를 일관되게 적용되도록 하기 위해서는 설정 정보를 가지고 있지 않은 계층에는 설정 정보를 생성하도록 해야 한다.

프리젠테이션 계층은 JSP Tag Library를 이용할 경우 '.TLD' 라는 설정 파일을 가지고 있으며, 데이터베이스의 스키마 정보는 데이터베이스의 테이블 메타 정보를 이용해서 쉽게 스키마 정보를 얻을 수 있다. 프로세스 계층의 데이터 객체는 객체를 통해 속성 명, 속성 타입을 이용해 간단하게 설정 XML 파일을 생성할 수 있다. EJB의 엔티티 빈의 경우는 개발 시 데이터 속성 정보를 XML로 관리하고 있다. 이러한 각 계층의 설정 정보를 통해 변경 시 모든 설정 정보를 변경하고 설정과 관련된 각 계층의 객체를 배치로 변경할 수 있다.

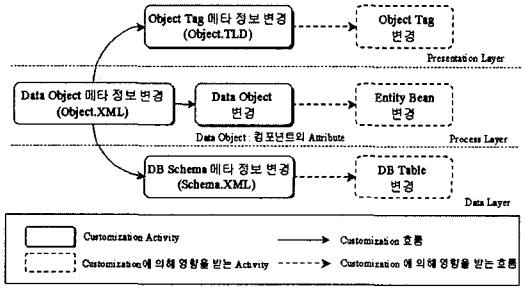


그림 3 속성 커스터마이제이션 프로세스

그림 3과 같이 컴포넌트의 속성 커스터마이제이션 프로세스는 데이터 속성('Data Object')의 메타 정보('Object.XML')를 변경하는 것으로 시작한다. 변경된 메타 정보는 프리젠테이션 계층의 관련된 부분과 데이터 속성을 일치시키기 위해 태그 메타 정보('Object.TLD', JSP Tag Library에서 Tag의 설정 정보를 관리하기 위한 XML)와 일치 시킨다. 화면을 표현하기 위한 'Object Tag'는 변경된 태그 메타 정보를 참조하여 화면에 변경된 속성을 표현한다. 프로세스 계층에서는 변경된 객체 메타 정보('Object.XML')를 이용해 데이터 객체('Data Object')를 파싱하여 변경하며, 변경된 'Data Object'는 컴포넌트인 'Entity Bean'의 속성에 해당하는 객체로서 'Entity Bean'이 'Data Object'를 상속 받아 자동으로 컴포넌트의 속성을 변경한다. 데이터 계층도 객체 메타 정보('Object.XML')와 스키마 메타 정보('Schema.XML')를 동기화하여 데이터베이스 스키마를 변경한다.

앞의 속성 커스터마이즈 프로세스는 속성 커스터마이제이션 구조(그림 2)를 기반으로 각 계층의 속성을 변경할 때 서로 연관된 부분들을 동기화하여 변경할 수 있도록 설정 정보를 변경하는 절차를 나타낸다. 본 논문에서는 설정 정보간의 동기화를 위해 파싱하여 변경하는 기법은 다루고 있지 않다.

```
import java.io.Serializable;

public class TRFQ extends TagSupport implements Serializable {

  public int      rfqId;
  public String   rfqNo;
  public String   title;
```

코드 1 데이터 객체

코드 1은 속성만을 가지고 있는 데이터 객체로서 프리젠테이션 계층이나 프로세스 계층에서 공통적으로 가져갈 수 있는 객체로서 속성 변경의 대상이 될 수 있다. 따라서 이 데이터 객체에 대한 메타정보를 변경하면 프리젠테이션 계층의 태그 메타 정보, 프로세스 계층의 데이터 객체, 그리고 데이터 계층의 스키마 메타 정보가 모두 변경된다.

```
public class CRqERfqBean extends TRFQ implements EntityBean
{
  TRFQ rfq = null;

  public CRqERfqPK ejbCreate(TRFQ rfq) throws CreateException,RemoteException
  {
    try
    {
      setRFQ(rfq);
    }
    catch(Exception e)
    {
      throw new CreateException("*** Create Exception : "+e.getMessage());
    }
    return null;
}
```

코드 2 계층의 컴포넌트

코드 2는 코드 1의 데이터 객체를 상속 받아 엔티티 빈 컴포넌트가 속성에 대한 정보를 전혀 가지고 있지 않으며 데이터 객체가 변경되면 자동으로 컴포넌트에 반영된다. 컴포넌트 내에 존재하는 기능은 모두 데이터 객체에 구현되어 있으며 컴포넌트는 단지 인터페이스(Interface) 역할만 하게 된다. 이렇게 데이터 객체에 데이터와 데이터 관련 함수까지 포함하고 있으므로 변경 시 한곳에 집중해서 변경되므로 다른 계층에 자동으로 반영될 수 있다.

이와 같이 속성 커스터마이제이션은 각 계층의 데이터 속성 정보들의 동기화를 통해 커스터마이제이션의

표 1 기능 커스터마이제이션의 분류

컴포넌트 종류 / 커스터마이제이션 종류	일반 객체	Entity Bean		Session Bean
		CMP	BMP	
새로운 기능 추가	함수추가 (상속)함수추가	함수추가 (상속)함수추가	함수추가 (상속)함수추가	함수추가 (상속)함수추가
기존 기능의 변경	(상속)Overriding	(상속)Overriding	(상속)Overriding, SQL은 XML로분리	(상속)Overriding

부담을 최소화할 수 있으며 효율적이고 신속하게 속성 변경을 할 수 있는 기법이다.

3.2 기능 커스터마이제이션(Bahavior Customization)

기능 커스터마이제이션 기법은 상속의 개념을 이용하여 기능을 변경하거나 추가한다. 기능 커스터마이제이션도 3계층에 반영될 수 있으며 프리젠테이션 계층은 일반 객체, 프로세스 계층은 세션 빈과 일반 객체, 그리고 데이터 계층은 엔티티 빈과 일반 객체에 반영될 수 있다.

표 1에서와 같이 새로운 기능 추가에 대해서는 모든 종류의 객체에 대해서 기존 클래스에 추가하거나 상속을 받아 서브 클래스에 함수를 추가한다. 기존에 존재하는 기능의 변경은 상속을 받아 오버라이딩(Overriding)하여 기능을 변경한다. 이렇게 상속을 이용하여 변경하거나 추가하므로 기본 컴포넌트는 변경되지 않고 블랙박스(Black Box)를 유지할 수 있다.

일반 객체는 모두 상속을 통해 함수를 추가하거나 변경하며 EJB의 세션 빈도 일반 객체와 동일하게 추가하거나 변경한다. 엔티티 빈의 CMP(Container Managed Persistent) 엔티티 빈도 일반 객체나 세션 빈 처럼 변경하면 된다. 그러나 BMP(Bean Managed Persistent) 엔티티 빈은 내부 로직이 대부분은 SQL 문장이기 때문에 SQL 스트링에 해당하는 문장을 XML로 분리하여 동적으로 기능을 변경할 수 있도록 할 수 있다.

상속을 통한 기능의 변경 기법은 그림 4와 같이 컴포넌트로부터 기능을 분리한 'Logic Object'를 상위 클래스로 일반화 시켜 처리를 단일화할 수 있다.

그림 4는 기능을 변경하기 위한 커스터마이제이션 구조로서 변경 가능한 기능들에 대해 단일화하여 변경할 수 있는 기법을 나타낸다. 컴포넌트('Component')로부터 기능을 분리하여 상위 'Logic Object'로 올리면 변경의 지점을 단일화 할 수 있으며 'Value Object'를 상속 받으므로 데이터 속성에 대한 처리도 관리할 수 있다. 이런 식으로 데이터와 기능을 컴포넌트로 분리하므로 모델 기반(Model-Driven Architecture)으로 설계가 가능하며 다른 컴포넌트 아키텍처에도 적용 가능하다[9,10].

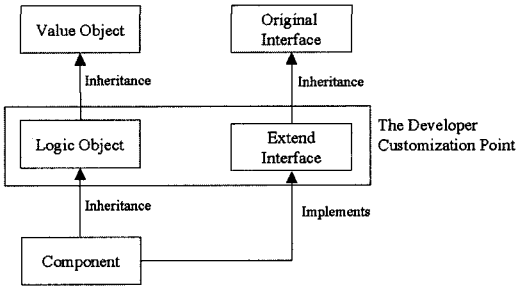


그림 4 기능 커스터마이제이션 구조

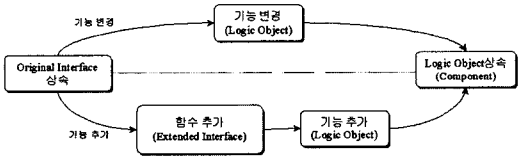
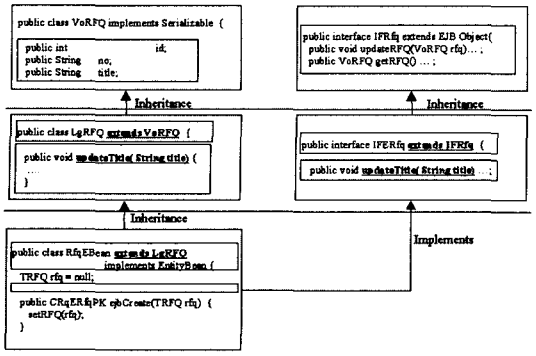


그림 5 컴포넌트 기능 커스터마이제이션 프로세스

이러한 기능 커스터마이제이션의 구조를 기반으로 기능을 변경하는 절차는 그림 5와 같다. 기능을 변경하거나 기능을 추가하기 위한 절차는 기존 컴포넌트 인터페이스('Original Interface')의 기능을 변경하거나 상속을 받아 확장 인터페이스에 기능을 추가한다. 기능 변경은 컴포넌트로부터 로직이 분리된 'Logic Object'의 함수를 변경하며 변경된 'Logic Object'는 컴포넌트에 적용될 수 있도록 'Component'가 'Logic Object'를 상속 받는다. 기능 추가는 확장된 'Extended Interface'에 함수를 추가하며 기능 로직은 'Logic Object'에 추가한다. 기능 변경과 마찬가지로 추가된 기능에 대해 'Component'에 적용하기 위해 'Logic Object'를 상속한다.

코드 3은 EJB 컴포넌트의 기능에 대해 일반화 시킨 코드 사례를 나타낸다. 'RfqEBean' 컴포넌트는 기능을 가지고 있는 'LgRFQ' 클래스로부터 상속을 받으며, 'RfqEBean' 컴포넌트는 'LgRFQ'의 구체화된 기능을 가지고 있는 함수를 호출하는 인터페이스 역할을 한다. 따라서 모든 기능에 대한 변경이나 추가는 'LgRFQ' 클래



코드 3 EJB 컴포넌트의 기능 일반화

스에 변경하거나 추가한다. 변경의 경우에는 'Rfq EBean' 컴포넌트에 어떠한 변경도 할 필요가 없다.

그러나 인터페이스의 시그니처(Signature)가 변경될 경우에는 인터페이스와 컴포넌트의 함수를 변경해야 한다. 기본적으로 변경되지 않는 함수에 대해서는 'IFRfq'에 넣어두고 추가되거나 변경될 가능성이 있는 함수에 대해서는 확장한 인터페이스인 'IFERfq'에 넣어두어 인터페이스의 변경의 단일화를 이룰 수 있다.

함수 커스터마이제이션 기법은 함수의 기능 로직을 변경하는 것이기 때문에 정적으로 함수 내부를 변경해야 하며 상속을 통해 기존 함수에 대한 추가와 변경을 한다. 상속을 통한 변경은 기존 함수를 유지하면서 변경할 수 있으며 이런 식의 커스터마이제이션은 컴포넌트 버전 관리에 유용하게 이용될 수 있다.

3.3 메시지 흐름 커스터마이제이션(Message Flow Customization)

메시지 흐름 커스터마이제시션 기법은 기존의 컴포넌트 간의 통합을 통한 메시지 흐름이 높게 결합되어 있으며 정적으로 코드화 되어 있기 때문에 변경하기에 많은 부하가 걸리며 재사용성도 많이 떨어 진다.

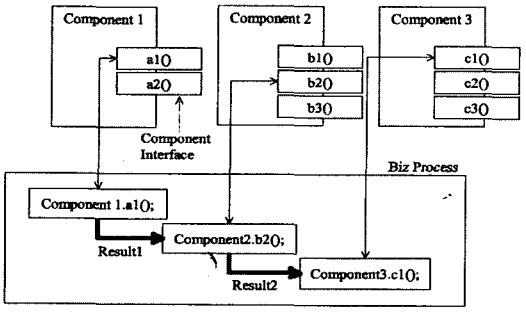


그림 6 기존 컴포넌트의 메시지 흐름

그림 6와 같이 비즈니스 어플리케이션을 개발하기 위해 컴포넌트을 사용하여 프로세스를 생성할 경우 컴포넌트들 간의 메시지 흐름이 정적으로 강하게 결합되어 있다. 이와 같이 강하게 결합되어 있는 비즈니스 프로세스 중에 하나의 컴포넌트 서비스를 삭제하거나 추가할 경우 프로세스 내의 다른 컴포넌트에 많은 영향을 준다. 그림 6에서 비즈니스 프로세스는 'Component1', 'Component2', 'Componnet3'를 호출하고 있으며 각각의 컴포넌트의 처리 결과를 다음 컴포넌트의 입력 파라메터로 이용하고 있다. 만약 'Component2' 가 다른 컴포넌트로 대치 되거나 삭제되는 경우 'Component1'은 어떤 컴포넌트에 결과 값을 전달해야 할지, 그리고 'Component3'는 어떤 컴포넌트의 결과 값을 입력으로 받아야 할지 변경 사항에 대해 고려사항이 복잡하며 다양하다. 이와 같은 변경의 부하는 컴포넌트를 이용한 새로운 제품을 개발하기 위한 Time-To-Market을 이루지 못하는 원인이 된다.

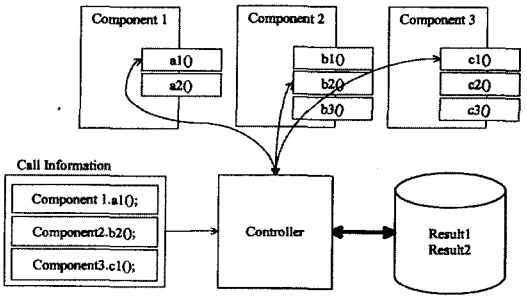


그림 7 동적 컴포넌트 메시지 흐름

강한 결합도를 줄이기 위해 본 논문에서는 그림 7와 같이 모든 컴포넌트의 호출 제어 역할을 하는 'Controller'가 호출 정보를 참조해 컴포넌트를 호출하고 결과 값은 레지스트리에 저장하여 참조할 수 있도록 한다. 메시지의 흐름은 단지 호출 정보 만을 변경하면 되며 다른 컴포넌트의 메시지 흐름에 전혀 영향을 주지 않는다. 이러한 기법을 본 논문에서는 글루 메커니즘을 이용해 동적으로 변경이 가능하도록 한다.

■ 글루 메커니즘(Glue Mechanism)

본 논문에서 제안하는 컴포넌트 메시지 흐름 커스터마이제이션 기법의 기반 기술은 아교(Glue) 개념을 적용한 글루 메커니즘이다. 그림 8 에서와 같이 'Glue'라는 중간 매개체를 통해 컴포넌트들은 서로 호출하고 호출될 수 있다. 호출하고 호출되는 컴포넌트들은 서로의 호출 정보를 가지고 있지 않으며 단지 'Glue'가 호출 정

보를 가지고 동적으로 흐름을 제어한다. 이런 식의 동적인 메시지의 흐름 제어를 통해 컴포넌트 개발의 효율성을 증가 시키며 컴포넌트 팀 단위의 프로젝트가 가능하게 할 수 있다.

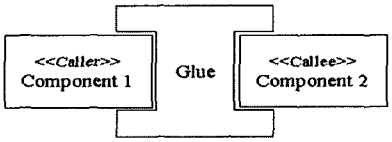


그림 8 Glue Mechanism의 개념

그림 8에서와 같이 호출하는 'Component 1'은 호출되는 'Component 2'의 인터페이스 정보를 전혀 가지고 있지 않으며 'Component 2'가 다른 컴포넌트로 교체되더라도 'Component 1'에 전혀 영향을 미치지 않는다. 단지 'Glue'에 존재하는 컴포넌트 호출 정보에 컴포넌트의 교체 정보만 수정하면 'Component 1'은 교체된 다른 컴포넌트를 호출하도록 한다.

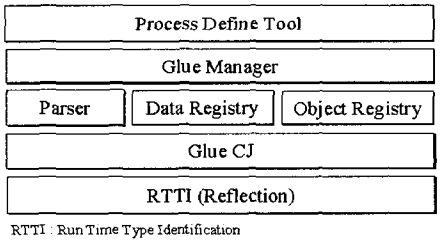


그림 9 Glue Mechanism Architecture

글루 메커니즘의 아키텍처는 그림 9에서 보는 것과 같이 동적으로 컴포넌트 호출할 수 있도록 동적 타입 정의 기능인 리플렉션(Reflection)을 기반으로 하고 있으며[11,12] 'GlueCJ'는 이러한 리플렉션 기능을 이용한다. 어플리케이션에서 정의된 프로세스를 이용하기 위한 제어 역할을 하는 것은 'GlueManager'가 담당한다. 프로세스 정보를 담고 있는 XML의 생성은 직접 프로세스 정보를 코딩하여 'GlueManager'로 전달할 수 있으며 'Process Define Tool'을 통해 XML을 자동 생성하여 'GlueManager'쪽으로 전달할 수도 있다. 'Glue Manager'는 입력 받은 프로세스 XML정보를 'Parser'에 의해 해석하여 'GlueCJ'로 전달한다.

그림 10과 같이 'GlueCJ'는 'GlueManager'로부터 프로세스에 관련된 XML정보를 얻어 컴포넌트의 기능을 처리한 후 결과를 'GlueManager'로 전달한다.

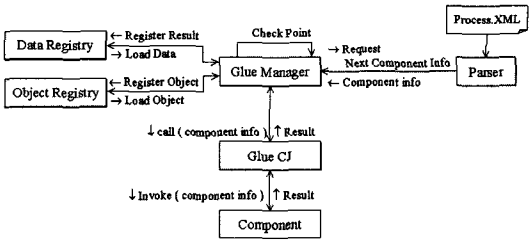


그림 10 Glue Mechanism Flow Architecture

'GlueManager'로부터 XML 정보를 얻은 'GlueCJ'는 컴포넌트를 호출하기 위해 전달하기 위한 다음과 같은 Interface를 활용한다.

```
public Object call(String componentName, String
methodName, Object[] methodParams)
```

XML로부터 해석된 정보 중에 컴포넌트 명과 함수 명을 얻을 수 있으며 함수의 파라메터 데이터는 'Data Registry'로부터 읽어와 객체 배열을 형성하여 'GlueCJ'의 'call()' 함수를 호출한다. 컴포넌트 호출 처리 결과는 'Data Registry'에 저장되며 또한 'Object Registry'는 한번 호출된 컴포넌트의 객체를 캐싱하기 위해 이용된다.

```
public Object call(String componentName, String methodName, Object[] methodParams)
{
  Class _class                = null;
  Class _retType              = null;
  Method _method              = null;
  Object _object              = null;
  Class[] _methodParamsType   = null;
  Object result               = null;
  try
  {
    _class                = Class.forName(componentName);  ①
    _methodParamsType     = getClass(methodParams);        ②
    _method               = _class.getMethod(methodName, methodParamsType ); ③
    _object               = _class.newInstance(); ④
    result                = _method.invoke(_object, methodParams); ⑤
  }
  catch(Exception e)
  {
    e.printStackTrace();
  }
  return result;
}
```

코드 4 GlueCJ의 Call 함수

코드 4는 'GlueCJ'의 'call()'함수로서 컴포넌트의 정보를 받아 리플렉션 기능을 통해 해당 컴포넌트를 호출한다. 컴포넌트 명을 통해 컴포넌트 클래스를 얻으며(①) 이 클래스를 통해 객체를 생성한다(④). 입력된 함수 파라메터 데이터에 대한 타입(type)을 얻은(②) 다음 함수 명과 함수 파라메터를 통해 함수 객체를 생성한다(③). 생성된 함수 객체와 컴포넌트 객체를 이용하여 컴포넌트 서비스를 호출한다(⑤). 이러한 컴포넌트 호출 메커니즘은 프로세스 정보를 담고 있는 XML를 읽어 반복적으로 처리된다.

'GlueManager'는 처리된 결과를 다른 컴포넌트에서 이용될 수 있도록 'Data Registry'에 저장한다. 'Data Registry'는 'Glue Manager' 객체와 라이프 타임(Life Time)이 같기 때문에 프로세스 마다 독자적인 'Data Registry'를 가지고 있다. 아키텍쳐에서 'Object Registry'는 'GlueCJ'를 통해 호출된 객체를 다시 생성하지 않고 'Object Registry'에 등록하여 재 요청 시 호출하여 사용한다.

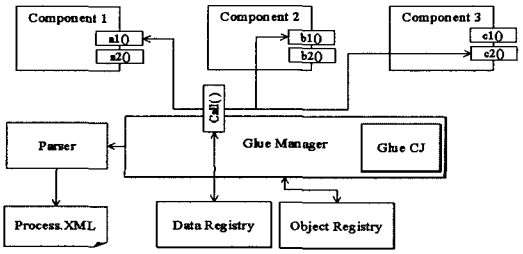


그림 11 Glue Manager의 Call Mechanism

글루 메커니즘의 컴포넌트 호출과 처리 메커니즘은 그림 11에서 보는 바와 같이 모든 호출하는 컴포넌트가 글루 메커니즘의 엔진인 'GlueManager'의 'call()'을 호출한다. 'GlueManager'는 컴포넌트의 흐름 정보를 가지고 있는 XML 정보를 읽어 컴포넌트를 반복적으로 호출한다. 처리 결과는 'Data Registry'에 저장되며 다른 컴포넌트 호출 시 입력으로 필요한 데이터를 읽어와 이용한다.

■ 커스터마이제이션 프로세스

메시지 흐름 커스터마이제이션 기법은 앞에서 정의한 'Glue' 메커니즘을 이용해 동적으로 실행할 수 있으며 동적으로 변경할 수 있다. 'Glue' 메커니즘을 통해 메시지 흐름을 제어하기 위해 메시지 흐름 정보를 XML로 정의한다. 따라서 메시지 흐름에 대한 커스터마이제이션은 설정된 XML 정보만을 변경하면 된다.

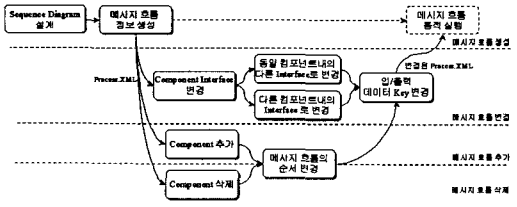


그림 12 메시지 흐름 커스터마이제이션 프로세스

그림 12는 메시지 흐름 커스터마이제이션의 프로세스를 나타내며 메시지 흐름에 대한 변경, 추가, 삭제의 과정을 보여준다.

메시지 흐름 생성은 설계된 순차도(Sequence Diagram)의 메시지 정보를 이용하여 메시지 흐름을 생성한다. 그림 13과 같이 순차도의 메시지 정보와 메시지 흐름 정보를 나타내는 'Process.XML'과의 맵핑 정보를 보여 준다.

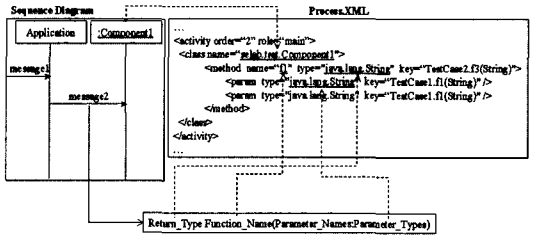


그림 13 순차도를 통한 메시지 흐름 정보 생성

생성되는 메시지 흐름 정보는 순차도의 메시지 정보 중에서 필요한 데이터인 컴포넌트명, 함수명, 결과 타입, 입력 파라메터 타입 만을 맵핑시키며 또한 부수적으로 필요한 정보는 추가하여 생성한다. 예를 들면 메시지 정보 중에 "Parameter_Names"는 메시지를 호출할 때 "Parameter_Types"를 통해 함수를 호출하기 때문에 파라메터 명은 필요하지 않으므로 메시지 흐름 정보에 포함시키지 않는다. 또한 호출 되는 메시지의 처리 결과값을 레지스트리에 저장하도록 하기 위해 키를 생성해서 메시지 흐름 정보에 추가한다.

메시지 흐름 변경은 메시지 흐름 정보를 나타내는 'Process.XML'에서 기존에 설정된 인터페이스를 다른 인터페이스로 변경하는 것이다. 인터페이스에 대한 변경 과정에서 입출력 데이터의 키에 대한 변경을 하여 전후 메시지 흐름에서 전달되는 입력값과 결과값을 받고 넘길 수 있도록 키를 변경한다.

메시지 흐름 추가는 새로운 컴포넌트의 인터페이스를 추가하는 것으로 특정 위치에 컴포넌트 인터페이스를 추가하며 추가된 메시지 흐름 정보를 통해 이후의 메시지 흐름의 순서를 변경한다. 변경된 메시지 흐름에서 전달되는 입출력 데이터의 키값도 변경한다.

메시지 흐름 삭제는 기존에 존재하는 컴포넌트의 인터페이스를 삭제하는 것으로 그 이후의 메시지 흐름의 순서를 변경한다.

■ 메시지 흐름 설정

XML내에는 컴포넌트들 간의 상세한 메시지 흐름 정보들을 포함하고 있으며 동적인 메시지 흐름 커스터마이제이션이 이 흐름 정보를 담고 있는XML을 통해서 가능하다. XML에 포함되는 정보는 다음과 같다.

표 2 XML Process 정보

Information		Tag	Attribute	Type	Description
Component	Component Name	<class>	Name	String	Full Package Name
Component Interface	Interface Name	<method>	Name	String	
	Return Type	<method>	Type	String	Full Package Name
	Saving Key	<method>	Key	String	To Data Registry
Parameter (Multiple)	Parameter Type	<param>	Type	String	Full Package Name
	Loading Key	<param>	Key	String	From Data Registry

```
<process>
  .....
  <activity order="3" role="main">
    <class name="selab.test.TestCase2">
      <method name="f1" type="java.lang.String" key="TestCase2.f3(String)">
        <param type="java.lang.String" key="TestCase1.f1(String)" />
      </method>
    </class>
  </activity>
  .....
</process>
```

코드 5 XML Process Format

코드 5은 프로세스 정보를 저장한 XML 파일이며 '<Activity>' 태그 내에 호출하려고 하는 컴포넌트의 정보가 설정된다. 각각의 '<Activity>'는 'Order' 속성을 가지고 있으며 다음 컴포넌트로 진행하기 위해 속성 데이터가 된다. 각각의 '<Activity>' 내의 태그에 대한 설명은 표 2와 같다.

위의 표에서와 같이 컴포넌트를 호출하기 위한 정보는 컴포넌트 명, 인터페이스 관련 정보, 파라메터 관련 정보로 구성된다. 인터페이스의 'Saving Key'나 인터페이스의 'Loading Key'는 'Data Registry'로 처리 데이터를 저장하거나 호출하기 위해 사용되는 키이다.

컴포넌트 간에 전달되는 데이터의 형태는 표 3과 같이 입력 데이터의 타입과 결과 타입의 조합으로 구성된다. 글루 메커니즘 내부로 전달되는 데이터의 타입은 객체로 인식되며 원시 타입(Primitive Type)은 모두 객체 타입으로 전환하여 전달해야 만 한다.

표 3 Parameter와 Return Type의 조합

Parameter	Return
None	None
None	One Object
One Object	None
One Object	One Object
Object Array	None
None	Object Array
Object Array	One Object
One Object	Object Array
Object Array	Object Array

■ 커스터마이제이션 범위

본 논문에서 제안하는 메시지 흐름 커스터마이제이션 기법의 범위는 컴포넌트 내의 객체들 간의 메시지 흐름 변경과 컴포넌트들 간에 메시지 변경으로 나눈다.

• 컴포넌트 내의 핫 스팟(Hot Spot)

그림 14와 같이 컴포넌트 내에 존재하는 객체들 간의 메시지 흐름 중에 잦은 변경의 가능성이 있는 지점(Hot Spot)을 본 논문에서 제안하는 커스터마이제이션 기법을 적용하므로 동적으로 변경이 가능하다.

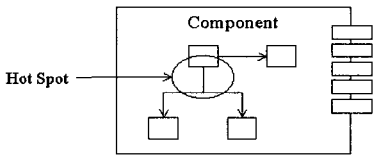


그림 14 컴포넌트 내의 흐름 변경

컴포넌트 내의 객체들은 강하게 결합되어 있으며 컴포넌트 내의 모든 클래스 구조를 본 커스터마이제이션 기법으로 적용하지는 않는다. 단지 동적인 변경을 요구하는 경우에만 적용한다.

• 컴포넌트 통합(Integration)

컴포넌트 통합은 그림 15 에서 보는 것과 같이 컴포넌트들 간에 메시지를 호출하여 통합하는 형태이며 이러한 정적인 컴포넌트 인터페이스 호출을 동적으로 변경할 수 있도록 한다.

컴포넌트 통합의 의미는 호출되는 컴포넌트의 인터페이스를 다른 인터페이스로 변경하거나 호출되는 컴포넌

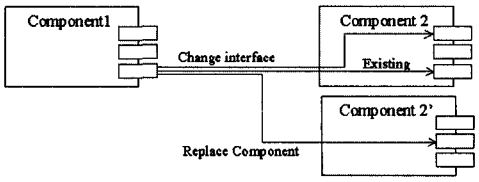


그림 15 컴포넌트들 간의 흐름 변경

트를 다른 컴포넌트로 교체하는 커스터마이제이션을 의미한다.

■ 글루 메커니즘의 커스터마이제이션 형태

글루 메커니즘을 통해 컴포넌트를 커스터마이즈 하는 형태는 컴포넌트 인터페이스 변경, 컴포넌트 삭제, 그리고 컴포넌트 교환으로 구분된다.

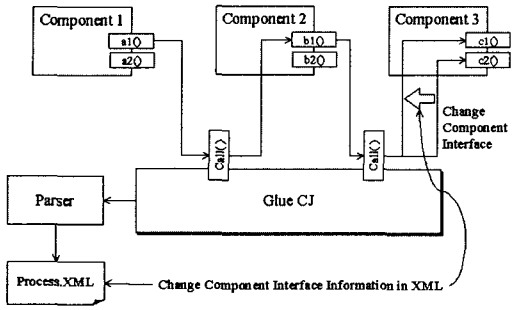


그림 16 Component Interface 변경

그림 16은 컴포넌트 인터페이스를 변경하는 메커니즘으로 호출되는 컴포넌트의 오버로딩(Overloading) 된 다른 인터페이스로 변경하는 과정을 보여주고 있다. 변경 전의 흐름은 'Component2'의 'b1()' 인터페이스가 'Component3'의 인터페이스 'c2()'를 호출하기 위해 글루 메커니즘의 'GlueCJ'의 'call()'을 호출하고 있다. 컴포넌트의 흐름을 변경하기 위해 기존 흐름인 'b1()'에서 'c2()'로의 흐름을 'b1()'에서 'c1()'으로 'Process.XML' 정보만을 변경하면 동적으로 컴포넌트의 흐름을 변경할 수 있다.

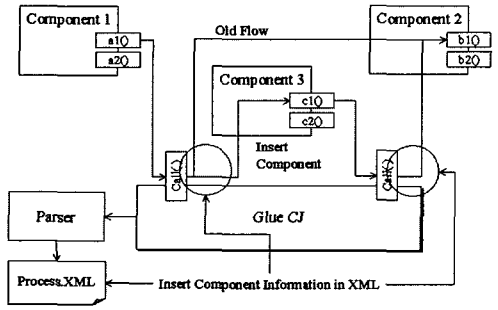


그림 17 Component 추가

컴포넌트 인터페이스의 변경 메커니즘과 유사하게 컴포넌트의 추가 메커니즘도 'Process.XML' 정보를 변경하여 동적으로 컴포넌트를 추가할 수 있다. 그림 17와 같이 'Component 1'에서 'Component 2'로 연결되어 있는 사이에 'Component 3'를 쉽게 추가할 수 있다. 그러나 'Component 3'의 호출하는 인터페이스는 'Component 2'를 호출하기 위해 필요한 입력 데이터를 만족시켜야 한다.

컴포넌트의 삭제 메커니즘은 기존 흐름에서 컴포넌트를 삭제하는 경우로서 다른 예외 사항들이 발생할 수 있다. 단순하게 'Process.XML'에서 삭제하려는 컴포넌트 정보만을 삭제하면 되는 것이 아니라, 삭제 컴포넌트 전후에 관련된 데이터가 존재하기 때문에 이러한 데이터들에 대한 처리가 필요하다. 그렇지 않으며 컴포넌트의 삭제와 동시에 그 프로세스는 쓸모 없게 된다.

4. 사례 연구

본 논문에서 제시한 컴포넌트 커스터마이제이션 기법을 통해 컴포넌트 간의 프로세스 커스터마이즈 하는 과정이 효율적이고 신속하게 변경되는지 검증하도록 한다. 본 사례 연구에서 이용되는 도메인은 *RFQ*(Request For Quotation)로 견적 요청하는 예제를 이용하도록 한다. 견적 요청 과정은 다음과 같다.

견적서 헤더 작성 → 견적 상품 선택 → 견적 요청 가능 업체 선택 → 견적서 발송

업무 성격이나 견적 항목에 따라 견적 가능 업체를 선정하지 않고 견적서를 작성하는 경우도 있으므로 이러한 부분이 컴포넌트의 핫 스팟(Hot Spot)으로 커스터마이제이션의 대상이 될 수 있다.

4.1 프로세스 생성

RFQ에 대한 XML을 생성하기 위해 XML 프로세스 생성 도구를 이용해 프로세스를 생성한다. 이러한 생성 도구 내부도 글루 메커니즘을 이용하고 있다. 프로세스에 해당하는 컴포넌트 명을 입력하면 해당 컴포넌트에 해당하는 인터페이스가 제시되며 그림 18와 같이 해당 컴포넌트의 인터페이스가 선택된다. 제시되는 인터페이스 해당 컴포넌트의 'public' 인터페이스 만 제시된다.

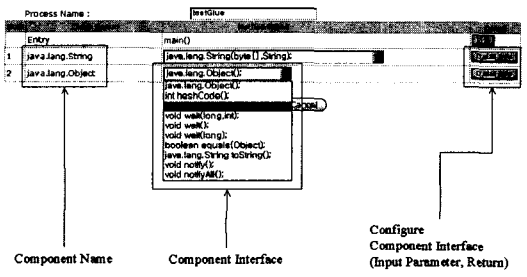


그림 18 컴포넌트 인터페이스 선택

그림 18에서 컴포넌트의 인터페이스를 선택하면 해당 인터페이스에 해당하는 입력 파라메터와 결과 값을 설정할 수 있도록 그림 19과 같이 설정 도구가 제시된다. 현재 컴포넌트의 함수에서 처리된 결과가 다음 컴포넌트에서 이용될 수 있도록 키(key)를 지정해 놓는다. 그러면 해당 컴포넌트의 함수를 처리한 후 결과 값과 키가 'Data Registry'에 저장되며 프로세스의 다른 컴포넌트에서 이용할 수 있다.

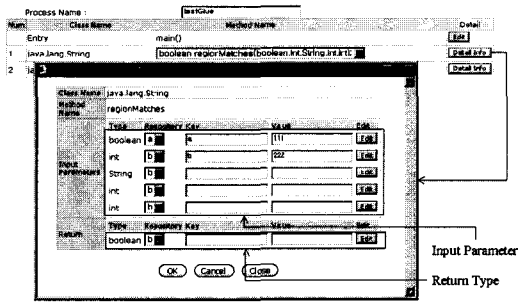


그림 19 컴포넌트 데이터 키 입력

RFQ 프로세스를 생성하기 위해 그림 18와 그림 19 과정을 반복하여 동적으로 컴포넌트를 설정한다. 이러한 과정을 거쳐 생성된 XML은 코드 6과 같으며 위의 도구를 이용하지 않고 에디터를 통해 직접 코딩하여 작성할 수도 있다.

```
<?xml version="1.0"?>
<process>
  <activity order="1" role="main">
    <class name="selab.test.RFQHeader">
      <method name="setHeader" type="java.lang.String" key="void">
        <param type="java.lang.String" key="HeaderData" />
      </method>
    </class>
  </activity>
  <activity order="2" role="main">
    <class name="selab.test.RFQItem">
      <method name="setItems" type="java.lang.String" key="void">
        <param type="java.lang.String" key="ItemData" />
      </method>
    </class>
  </activity>
  <activity order="3" role="main">
    <class name="selab.test.RFQSupplier">
      <method name="setSupplier" type="java.lang.String" key="void">
        <param type="java.lang.String" key="SupplierData" />
      </method>
    </class>
  </activity>
  <activity order="4" role="main">
    <class name="selab.test.RFQSender">
      <method name="send" type="java.lang.String" key="void">
        <param type="java.lang.String" key="HeaderData" />
        <param type="java.lang.String" key="ItemData" />
        <param type="java.lang.String" key="SupplierData" />
      </method>
    </class>
  </activity>
</process>
```

RFQ Header
RFQ Item
RFQ Supplier
RFQ Sender

코드 6 RFQ Process XML

코드 6의 RFQ 프로세스 정보는 4개의 <Activity>로 구성되며 각각의 <Activity>는 순차적인 'Order'를 가지고 있다. 각각의 <Activity>는 <class>와 <method>, <param>의 조합으로 구성되며 <class>와 <method>는 반드시 하나이어야 하며 <param>은 다중으로 구성된다. <param>은 현재 컴포넌트의 <method>를 수행하기 위해 필요한 입력 데이터로서 다른 컴포넌트의 <method> 처리 결과값을 키로 갖거나 아니면 외부에서 입력을 받아야 한다. 현재 RFQ 프로세스의 모든 <param> 키인 'HeaderData', 'ItemData', Supplier Data'를 처리 결과로 반환하는 <method>가 없으므로 외부에서 받아야 한다.

```
import java.util.*;
import java.io.*;
import selab.glue.*;

public class TestGlueCJ{
        public TestGlueCJ()  {}

        public static void main(String[] args) {
                /* Glue Mechanism Class */
                GlueManager gm = new GlueManager();

                /* Entry Point Data(Object Array) */
                Hashtable entryData     = new Hashtable();

                /* Dummy Data for the Entry data of the Process */
                /* Entry Data is the Object Array */
                entryData.add("HeaderData", "001");
                entryData.add("ItemData", "Computer");
                entryData.add("SupplierData", "Samsung");

                /* Result is the Object */
                /* 'Component.XML' contains the information of the process */
                gm. inputEntryData(entryData);
                Object result = gm.doProcess("RFQ-Process.XML");
}}
```

코드 7 RFQ Process이용 어플리케이션

생성된 RFQ 프로세스는 코드 7과 같이 어플리케이션에서 XML을 호출하여 프로세스를 진행할 수 있도록 입력된다. 'Glue Manager'로 XML를 넘기기 전에 엔트리 데이터에 해당하는 데이터를 입력해야 하며 입력하는 데이터를 프로세스에서 요구하는 키와 매핑하여 전달해야 한다.

'Glue Manager'에서는 'RFQ-Process.XML'와 입력 엔트리 데이터를 이용하여 컴포넌트의 기능을 처리한 후 처리 결과를 객체로 반환하게 된다. 반환되는 데이터의 형태는 일반 객체부터 컬렉션(Collection)객체까지 처리할 수 있다.

처리 결과는 그림 20과 같이 RFQ 헤더 작성부터 RFQ 발송까지 프로세스에서 정의된 대로 진행되는 것을 볼 수 있다.

커스터마이제이션 과정은 단지 'RFQ-Process.XML'만을 수정하여 커스터마이제이션 할 수 있다. 따라서 위

```
C:\glue>java TestGlueCJ

 Parameter Type : java.lang.String
** Check Point : Activity Order 1

 Parameter Type : java.lang.String
** Check Point : Activity Order 2

Parameter Type : java.lang.String
 ** Check Point : Activity Order 3

Parameter Type : java.lang.String
Parameter Type : java.lang.String
Parameter Type : java.lang.String
 ** Check Point : Activity Order 4

>> Process:RFQHeader.setHeader(String)->
          RFQItem.setItem(String)->
          RFQSupplier.setSupplier(String) ->
          RFQSender.send(String,String,String)
```

그림 20 RFQ Process 처리 결과

에서 RFQ의 핫 스팟이 공급자 선정 부분이 변동 가능하므로 그 컴포넌트를 XML에서 삭제하도록 한다.

```
<?xml version="1.0"?>
<process>
  <activity order="1" role="main">
    <class name="selab.test.RFQHeader">
      <method name="setHeader" type="java.lang.String" key="void">
        <param type="java.lang.String" key="HeaderData" />
      </method>
    </class>
  </activity>
  <activity order="2" role="main">
    <class name="selab.test.RFQItem">
      <method name="setItems" type="java.lang.String" key="void">
        <param type="java.lang.String" key="ItemData" />
      </method>
    </class>
  </activity>
  <activity order="3" role="main">
    <class name="selab.test.RFQSupplier">
      <method name="setSupplier" type="java.lang.String" key="void">
        <param type="java.lang.String" key="SupplierData" />
      </method>
    </class>
  </activity>
  <activity order="3" role="main">
    <class name="selab.test.RFQSender">
      <method name="send" type="java.lang.String" key="void">
        <param type="java.lang.String" key="HeaderData" />
        <param type="java.lang.String" key="ItemData" />
        <param type="java.lang.String" key="SupplierData" />
      </method>
    </class>
  </activity>
</process>
```

RFQ Header
RFQ Item
Delete
RFQ Supplier
Change '4' to '3'
RFQ Sender
Delete

코드 8 변경된 RFQ Process XML

```
C:\glue>java TestGlueCJ

 Parameter Type : java.lang.String
** Check Point : Activity Order 1

 Parameter Type : java.lang.String
** Check Point : Activity Order 2

Parameter Type : java.lang.String
Parameter Type : java.lang.String
 ** Check Point : Activity Order 3

>> Process:RFQHeader.setHeader(String)->
          RFQItem.setItem(String)->
          RFQSender.send(String,String)
```

그림 21 변경된 RFQ Process 처리 결과

RFQ 커스터마이제이션은 코드 8과 같이 공급자를 선정하는 프로세스를 삭제하기 위해 'Order'가 '3'인 '<Activity>' 태그를 모두 삭제하며 다음 '<Activity>'의 'Order'를 '4'에서 '3'으로 수정한다. 또한 견적서 발송 컴포넌트에서는 공급자 정보가 없이 발송하는 오버로딩 된 함수를 선택하며 공급자 정보를 삭제한다.

위의 그림에서와 같이 공급자 선정 컴포넌트가 제외된 RFQ 프로세스의 변경된 처리 결과를 볼 수 있다. 지금까지의 과정과 같이 컴포넌트 수준에서 컴포넌트 인터페이스의 변경이나 컴포넌트 변경이 동적으로 쉽고 안전하게 처리될 수 있다.

5. 평 가

본 논문에서 제안한 커스터마이제이션 기법이 다른 컴포넌트 개발 방법론에서 제안한 기법들과 어떤 차이가 있는지 비교 평가한다

표 4 커스터마이제이션 기법 비교

Factors			This Method	Catalysis	Component-ware
특징	Black-Box Design ?		F	N	N
	Dynamic Customization ?		F	N	N
	Component Coupling 축소 ?		P	N	N
기능	Attribute	속성 변경	N	N	N
	Behavior	Interface 변경	F	N	N
		Interface 추가	F	P	P
		Component내의 Class 교체	F	P	P
	Message Flow	메시지 호출 변경	F	N	N
		메시지 호출 추가	F	N	N
		메시지 호출 삭제	P	N	N
[F] Full [N] Not Support [P] Partial					

표 4에서와 같이 Catalysis나 Componentware는 "Required Interface"나 "Import Interface"를 이용해 인터페이스를 변경하거나 컴포넌트내의 클래스를 교체하기 위한 개념적인 방법을 제시하고 있다. 본 논문에서는 기존 컴포넌트 개발 방법론에서 언급하지 않고 있는 행위나 메시지 흐름에 대한 구체적인 커스터마이제이션 기법을 제안하고 있다.

표 4와 같이 기존 커스터마이제이션 기법에 대한 연구가 미약하기 때문에, 또 다른 측면에서 본 논문에서 제시한 컴포넌트 커스터마이제이션 기법과 일반적인 소스 코드 수준에서의 변경하는 방식과 어떠한 차이가 있는지 평가한다. 평가 요소로는 컴포넌트나 컴포넌트들

간의 프로세스를 변경하기 위해 접근하는 접근 포인트(AP, Access Point)와 커스터마이제이션 비용(Customization Cost)으로 평가한다. 접근 포인트는 속성, 함수, 메시지 흐름을 변경하기 위해 접근해야 하는 파일 수를 기준으로 하며 커스터마이제이션 비용은 속성, 함수, 메시지 흐름을 변경하기 위해 소요되는 시간(CT, Customization Time)을 기준으로 한다.

[정의1] $AP = \sum_{i=1}^{n}(AP_i \times W_i)$,

$i_{1-3} = \{attribute, method, flow\}$,

W(weight)=flow>method>attribute

정의 1은 커스터마이제이션의 접근 포인트로서 속성, 함수, 메시지 흐름을 변경하기 위해 요구되는 접근의 회수로 평가가 된다. 가중치는 커스터마이즈 하기 위해 복잡성의 정도를 나타내는 것으로 복잡성이 강할수록 높은 가중치를 주므로 커스터마이즈하기가 어려워짐을 나타낸다.

[정의2] $CT = \sum_{i=1}^{n}(CT_i \times W_i)$,

$i_{1-3} = \{attribute, method, flow\}$,

W(weight)=flow>method>attribute

정의 2는 컴포넌트의 변경 시간을 평가하기 위한 메트릭으로 정적 변경 또는 동적 변경 인지에 따라 변경 시간의 결정된다. 동적 변경은 결국 코드 수준에서 하는 것이기 때문에 컴파일 과정까지 거치므로 동적으로 변경하는 것보다 많은 변경 시간이 걸린다고 할 수 있다. 함수의 변경 시간은 4가지 형태로 구분하여 평가 될 수 있다.

[정의3] $CT_{method} = \sum_{j=1}^{m}(CT_i \times W_i)$,

$j_{1-4} = \{Session, CMP, BMP, GeneralClass\}$

정의 3은 함수의 변경 시간을 측정하기 위한 메트릭으로서 세션 빈(Session Bean), 엔티티 빈(Entity Bean)의 CMP(Container Managed Persistent)와 BMP(Bean Managed Persistent), 그리고 일반 클래스(General Class)의 함수를 변경하는데 소요되는 시간을 측정하기 위한 메트릭이다.

[정의4] Eval=AP+CT

정의 4는 접근 포인트와 커스터마이제이션 시간을 통해 기존의 커스터마이제이션 방법과 본 커스터마이제이션 기법을 비교할 수 있다.

표 5은 변경의 접근 포인트를 기존 방식과 본 논문에서 제안한 방식과 비교했을 때 나온 결과로서 평가 결과는 다름과 같다[정의.1].

표 5 Access Point 평가

Method / Factors	Existing Method		This Method	
	Evaluation	Weight	Evaluation	Weight
Attribute	3	1	1	1
	Model, View, Controller		Object (Controller)	
Method Behavior	1	2	1	2
	Modify method body		Overloading (Inheritance)	
Message Flow	1	3	1	2.5
	Modify method body		Modify XML	

기존 방식은 '8'이 나왔으며 본 기법에서는 '5.5'으로 본 기법이 컴포넌트 변경 시 접근하는 부분이 줄어드는 것을 알 수 있다.

표 6 Customization Time 평가

Factors / Method		Existing Method		This Method	
		Evaluation	Weight	Evaluation	Weight
Attribute		Static (1)	1	Static (1)	1
Method Behavior	Session	Static (1)	2	Static (1)	2
	CMP Entity	Static (1)	2	Static (1)	1
	BMP Entity	Static (1)	2	Dynamic (0.5)	1
	General Class	Static (1)	2	Static (1)	2
Message Flow		Static (1)	3	Dynamic (0.5)	2.5

표 6은 컴포넌트나 컴포넌트들 간의 프로세스를 변경 시 어느 정도의 소요 시간이 걸리는 지를 평가했으며 결과는 다음과 같다[정의 2, 정의3].

$CT_{ExistingMethod} = (1\times1)+\{(1\times2)+(1\times2)+(1\times2)+(1\times2)+\}+(1\times3)=12$

$CT_{ThisMethod} = (1\times1)+\{(1\times2)+(1\times1)+(0.5\times1)+(1\times2)+\}+(0.5\times2.5)=6.75$

기존 방식은 '12'가 나왔으며 본 기법을 이용할 경우 '6.75'로 본 기법을 통해 동적으로 변경할 경우 많은 부하를 줄임을 알 수 있다.

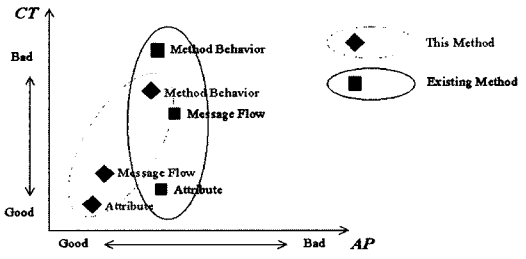


그림 22 평가 분석

$Eval_{EstingMethod} = 8 + 12$
$Eval_{ThisMethod} = 5.5 + 6.75$

전체적으로 접근 포인트와 변경 시간을 통해 기존 방식은 20(=8+12)이 나왔으며 본 기법은 12.75(=5.5+6.75)로 36%의 작업 부하를 줄일 수 있음을 평가할 수 있다. 특히 그림 22에서 보는 것과 같이 본 기법은 속성과 메시지 흐름의 변경에 많은 효과를 줄 수 있다.

6. 결 론

지금까지 본 논문에서는 컴포넌트를 이용하여 소프트웨어를 개발할 때 도메인의 업무 특성에 맞게 컴포넌트를 변경하기 위한 기법을 제안했다. 컴포넌트의 데이터 속성이나 기능을 변경하기 위한 정적 기법과 컴포넌트 내의 객체들 간의 메시지 흐름이나 컴포넌트들 간의 인터페이스를 통한 메시지 흐름을 동적으로 커스터마이즈 하기 위한 기법을 제안하였다. 기존 컴포넌트를 이용해 소프트웨어를 개발할 때의 단점인 컴포넌트들 간의 강한 결합도를 최소화하도록 컴포넌트를 포함한 프로세스를 쉽게 커스터마이즈 하기 위한 기법을 살펴보았다.

향후 연구과제는 컴포넌트들을 통해 개발된 프로세스를 하나의 컴포넌트로 고려하여 프로세스들 간에 커스터마이즈하기 위한 방법을 연구해 본다.

참고 문헌

[1] Chul Jin Kim, Soo Dong Kim, "A Component Workflow Customization Technique," Journal of KISS: Software and Applications, Volume 27, Number 5, May 2000.

[2] Kang K: Issues in Component-Based Software Engineering, 1999 International Workshop on Component-Based Software Engineering

[3] Szyperski C., Component Software: Beyond Object-Oriented Programming, Addison Wesley Longman, Reading, Mass., 1998

[4] D'souza D. F. and Wills A. C., Objects, Components, and Components with UML, Addison-Wesley, 1998.

[5] Desmond Francis D'Souza, Alan Cameron Wills, *Objects, component, and frameworks with UML* : the Catalysis approach, Addison Wesley Longman, Inc., 1999.

[6] 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.

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

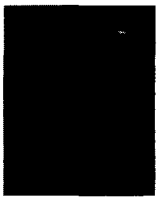
[8] Rausch A. *"Software Evolution in COMPONENTWARE Using Requirements/Assurances Contracts,"* Proceedings of the 22th International Conference on Software Engineering, 06/2000

[9] Richard Soley and the OMG Staff Strategy Group, White Paper Draft 3.2-November 27,2000 "Model Driven Architecture".

[10] Desmond Dsouza, Kinetium, at www.kinetium.com, "Model-Driven Architecture and Integration".

[11] JavaWorld webzine, http://www.javaworld.com

[12] The Third JavaOne Conference, http://www. javasoft.com/javaone



김 철 진

1996년 경기대학교 전산학 학사. 1998년 숭실대학교 전산학 석사. 1998년 ~ 현재 숭실대학교 컴퓨터 학과 박사과정. 관심분야는 객체지향 방법론, 객체지향 프레임워크, 분산 컴퓨팅, 컴포넌트 개발 방법론, 컴포넌트 커스터마이제이션

김 수 동

1984년 미조리 주립 대학교 전산학과 졸업(학사). 1988년 The University of Iowa(전산학 석사). 1991년 The University of Iowa(전산학 박사). 1991년 ~ 1993년 한국통신 연구 개발단 선임연구원. 1994년 현대 전자 소프트웨어 연구소 책임 연구원. 1995년 ~ 현재 숭실대학교 컴퓨터학부 부교수. 관심 분야 객체지향 방법론, 분산 객체 컴퓨팅, 컴포넌트 개발 방법론