# 컴포넌트 행위 커스터마이제이션 기법
## (The Customization Techniques of Component Behavior)

김 철 진 † 정 승 재 †† 김 수 동 †††
(Chul Jin Kim) (Seung Jae Jung) (Soo Dong Kim)

**요 약** 다양한 도메인의 요구사항을 만족시켜 주기 위한 비즈니스 컴포넌트(Business Component)는 다양성을 제공할 수 있도록 개발되어야 한다. 그러나 컴포넌트 개발 시에 다양한 요구사항을 분석하여 개발되더라도 컴포넌트가 이용될 때 예상하지 못한 요구 사항들이 발생하기 때문에 요구 사항들을 완전하게 만족시켜 주기 위한 컴포넌트의 개발은 쉽지 않다. 이와 같은 이유 때문에 컴포넌트가 블랙박스가 아닌 화이트 박스로 제공되므로 컴포넌트를 인터페이스에 의해 변경하는 것이 아니라 직접 코드를 변경하는 문제가 발생한다. 따라서 컴포넌트를 이용한 Time-To-Market을 이루기가 쉽지 않으며 컴포넌트의 재사용성도 떨어진다.

본 논문에서는 컴포넌트의 변경 가능한 부분을 분석하여 다양한 요구 사항을 만족시킬 수 있는 커스터마이제이션 기법을 제안한다. 컴포넌트의 초기 가변성은 컴포넌트 개발(CD : Component Development) 과정에서 설계되며 가변성 적용을 위해 커스터마이제이션 기법을 이용한다. 가변성이 적용된 컴포넌트를 이용하여 어플리케이션을 개발하는 과정에서 가변성이 재 설계될 수 있으며 이러한 과정을 통해 컴포넌트의 가변성은 진화하고 컴포넌트의 일반성은 향상될 수 있다. 본 논문에서 제시하는 커스터마이제이션 기법은 컴포넌트가 재설계될 때 기존 컴포넌트는 전혀 변경하지 않고 확장하여 컴포넌트를 변경할 수 있다.

**키워드** : 컴포넌트, 컴포넌트 인터페이스, 커스터마이제이션, 행위

***Abstract*** The business component for satisfying a variety of domain requirements should be developed to provide a variety. But, although components are developed by analyzing the variety of requirements when they are developed, developing components that satisfy all requirements is not easy since unexpected requirements occur as it is used components. For this reason, components are not provided as black boxes, and there by components are not modified in the interface only but the source codes are directly modified. Accordingly, a Time-To-Market by the use of components is not easy and a reusability of the components also decreases.

This study proposes a customization technique that can be satisfied requirements of many different kinds of domains by analyzing variable spots of components. The initial variability of components is designed in the component development phase, and a customization technique is used to apply the variability. The variability can be redesigned during the development of application by using the components to which the variability is applied. Through this process, a variability of components evolves and a generality of the components can be improved. The proposing customization technique in this study can change the component to extend without changing the existing component when it is redesigned.

**Key words** : Component, Component Interface, Variability, Customization, Behavior

† 학생회원 : 숭실대학교 컴퓨터학과
cjkim@selab.soongsil.ac.kr
†† 비 회 원 : 이네트 기술연구소
chandler@enet.co.kr
††† 종신회원 : 숭실대학교 컴퓨터학과 교수
chungcw@islab.kaist.ac.kr
논문접수 : 2002년 9월 27일
심사완료 : 2003년 2월 10일

## 1. 서 론

컴포넌트는 이제 생소한 말이 아니며 소프트웨어 개발의 필수적인 요소로 여겨지고 있다. 객체지향 기법이 소프트웨어 개발을 혁신적으로 향상시켜주지 못했기 때문에 80년대 말부터 소개 되었던 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. 컴포넌트는 소프트웨어 개발을 조립의 개념으로 발전시켰으며 외부에서 개발된 컴포넌트를 조립하여 쉽게 소프트웨어를 개발할 수 있는 구조를 제공하였다. 조립하는 소프트웨어는 도메인의 업무 범위에 맞는 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[5,6,7]. 그러나 이러한 다양성을 제공하기 위한 가변성을 설계하기가 어려우며 설계된 가변성을 적용하기 위한 기법들이 존재하지 않기 때문에 컴포넌트 내부를 변경해야 한다. 따라서 본 논문에서는 컴포넌트에 다양성을 제공하기 위해 다양한 도메인을 분석하여 컴포넌트를 설계하기 위한 기법보다는 다양한 도메인의 요구사항을 수용할 수 있는 장치를 제공하기 위한 커스터마이제이션 기법을 제안한다. 본 논문에서는 컴포넌트의 행위에 해당하는 커스터마이제이션 기법을 제안한다.

본 논문의 구성은 다음과 같다. 2장은 컴포넌트의 가변성이나 CBD방법론들이 커스터마이제이션 설계를 어떻게 제시하고 있는지 알아보며 커스터마이제이션 설계를 정형적 표현으로 나타내기 위한 Z언어와 LOTOS에 대해 알아본다. 3장은 커스터마이제이션을 설계하는 전체적인 프로세스에 대해 알아보며, 본 논문의 핵심인 4장에서는 컴포넌트 행위 커스터마이제이션 기법과 이러한 기법을 통해 어떻게 커스터마이제이션 되는지 사례를 통해 알아본다. 5장에서는 본 논문의 기법과 다른 기법들을 비교 평가한다.

## 2. 관련 연구

### 2.1 컴포넌트 가변성

컴포넌트 모델은 컴포넌트의 기본적인 아키텍처, 컴포넌트의 인터페이스, 그리고 컴포넌트와 컨테이너 간의 상호작용을 위한 메커니즘을 정의한다. 이와 같이 컴포넌트 모델은 재사용할 수 있는 컴포넌트를 지원하기 위한 환경을 정의한다[8]. 컴포넌트는 다른 컴포넌트나 프레임워크와 상호 작용하기 위해 미리 정의된 아키텍처를 따라 설계되고 구현되어야만 한다. 컴포넌트 기반 아키텍처는 컴포넌트를 첨가하거나 대치하여 기능적인 향상을 이룰 수 있도록 프레임워크 형태로 제공된다[9,10]. 카네기 멜론 대학의 기술 보고서에서 컴포넌트 참조 모델은 외부에서 개발된 컴포넌트가 컴포넌트 모델의 규약을 따르기만 하면 쉽게 플러그인(Plug-In)하여 컴포넌트 프레임워크의 서비스를 이용할 수 있음을 나타낸다[3]. 컴포넌트 프레임워크는 일반적인 운영체제의 기능을 제공하기 보다는 트랜잭션 서비스나 영속성 서비스 등을 제공하는 시스템을 말한다. 예를 들면, EJB 아키텍처에서 EJB 서버를 컴포넌트 프레임워크라고 할 수 있다. 컴포넌트 모델은 컴포넌트가 컴포넌트 프레임워크을 이용할 수 있도록 제공하는 구조를 말하는 것으로 EJB아키텍처의 EJB 컨테이너에 해당한다. 컴포넌트는 컴포넌트 모델에 디플로이(Deploy) 되기 위해서 컴포넌트 타입이나 컴포넌트 계약(Contract)에 따라 개발되어야 한다[11,12].

컴포넌트 모델의 규약에 맞게 개발된 컴포넌트는 인터페이스를 통해 이용될 수 있으며 다양한 도메인에 적용하기 위한 일반화된 인터페이스를 제공해야 한다. 기본적으로 컴포넌트는 블랙박스 형태의 재사용 단위로 개발 시스템에서 컴포넌트를 사용하기 위해 컴포넌트 인터페이스만을 인식해야 하며 내부 구현 모듈을 인식할 필요가 없어야 한다. 그러기 위해서는 컴포넌트 인터페이스는 비즈니스 기능을 제공할 뿐만 아니라 컴포넌트를 관리하고 제어하는 기능도 제공해야 한다. 관리 목적으로 제공되는 기능은 컴포넌트를 조합하거나 커스터마이제이션하기 위한 기능들이다. 비즈니스 기능에 한정하여 인터페이스를 정의하면 다양한 도메인에 적용될 때 상이한 요구사항을 충족시키기 위해 컴포넌트 내부를 변경해야 하는 문제가 발생할 수 있다. 이와 같이 컴포넌트 내부를 변경하지 않고 인터페이스 수준에서 컴포넌트를 변경하기 위한 기법이 필요하다.

### 2.2 커스터마이제이션 기법

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

Desmond F. D'Souza에 의해서 제안된 Catalysis는 1991년에 OMT의 정형화로서 시작 되었으며 객체나 프레임워크와 함께 컴포넌트 기반 개발을 위한 차세대

방법론으로 출현하게 되었다[8, 13].

Catalysis는 모델 간에 일관성 규칙을 잘 정의하고 있으며 복잡한 시스템을 설계하는데 다양한 뷰(View)를 제공하기 위한 기법을 정의하고 있다. 추가적으로 Catalysis는 다른 방법론과는 다르게 프레임워크를 사용하여 컴포넌트를 개발할 수 있는 개발 프로세스를 정의한다[14,15].

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[16]. Catalysis는 프로세스 패턴(Process Pattern)을 정의하여 다양한 프로젝트에 유연하게 프로세스를 구성할 수 있도록 지원한다.

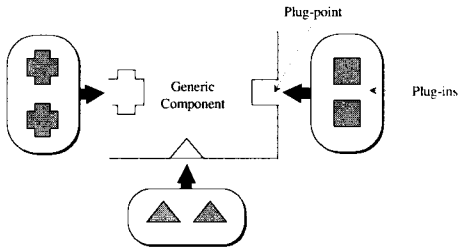



그림 1 Plug-Points

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

그림 1와 같이 일반 컴포넌트(Generic Component)는 "Plug-Point"를 정의하고 적용되는 도메인의 요구 사항에 맞는 "Plug-Ins"(모듈이나 컴포넌트)를 플러그 인하여 일반 컴포넌트을 변경할 수 있다. "Plug-Point"는 특정 도메인의 어플리케이션에 사용되기 위해 가변적으로 적용될 수 있도록 하기 때문에 "Variation-Point"라고도 한다. 어플리케이션에 적용되기 전에 일반 컴포넌트는 미리 만들어진 기본 모듈을 적용할 수 있다.

Catalysis는 "Required Interface"를 정의하는 절차를 포함하고 있다. 이 인터페이스는 "Plug-Point"를 제공하는 역할을 하며 커스터마이제이션을 위한 가변성 요소가 될 수 있다. 일반 컴포넌트에서 서비스로 제공되는 인터페이스를 "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[16]. Componentware는 커스터마이제이션을 위해 Catalysis의 "Required Interface"와 유사한 "Import Interface"를 정의한다. 이 인터페이스도 컴포넌트 외부의 다양한 요구사항을 충족시킬 수 있도록 한다. Componentware는 일반적으로 컴포넌트에서 제공되는 인터페이스를 "Export Interface"로 정의하며 외부 컴포넌트에 제공되는 인터페이스를 "Import Interface"로 정의한다[17]. 이와 같이 "Export Interface"와 "Import Interface" 모두는 컴포넌트 사이와 연결을 가능하게 하며 가변성을 제공할 수 있다.

앞에서 언급한 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 고려한다.

커스터마이제이션 기법에 관련된 연구로서 본 논문과 비교되는 논문은 "Using Component Composition for Self-customizable Systems"[18]와 "Component Interface Pattern" [19]가 있다. 그러나 두 논문은 모두 복합 컴포넌트를 개발할 때 커스터마이제이션을 하기 위한 기법들을 다루고 있기 때문에 컴포넌트 내부의 행위에 대한 커스터마이제이션 기법은 부분적으로 다루고 있다. [18]의 논문은 컴포넌트들 간에 예상하지 못한 변경을 하기 위해 요구되는 인터페이스에 대한 정의를 하여 동적으로 커스터마이제이션이 되도록 하기 위한 기법이다. [19] 기법도 또한 컴포넌트를 조합하기 위해 컴포넌트의 인터페이스에 제공 인터페이스와 요구 인터페이스 뿐만 아니라 구조적 설명(Structural Description)과 행위적 명세(Behavior Specification)을 통해 표현한다. 구조적 설명은 컴포넌트 인터페이스의 서비스 접근 포인트인 채널(Communication Channel)를 이용하여 표현하며 행위적 명세는 패트리 넷(Petri net)을 이용하여 컴포넌트들 간의 연결과 협업을 명세한다. [18]과 [19]의 연구는 모두 복합 컴포넌트를 동적으로 조합하기 위한 명세 기법 및 패턴을 제시하고 있으며 단일 컴포넌트에 대한 내부 행위 커스터마이제이션에 기법은 부분적으로 제시하고 있다.

### 2.3 Z Notation

정형 명세(Formal Specification)는 시스템이 갖는 속성을 명확하게 표현하기 위한 수단이며 시스템의 기능이 어떻게 수행되는지를 나타내기 보다는 어떤 기능들

이 제공되는지를 나타낸다. 이러한 정형 명세는 시스템을 개발하는 프로세스에서 대량의 상세한 프로그램 코드로부터 복잡한 정보를 얻을 필요 없이 시스템이 무엇을 해야 하는지를 명확하게 제시해 줄 수 있기 때문에 유용하게 사용된다[20,21].

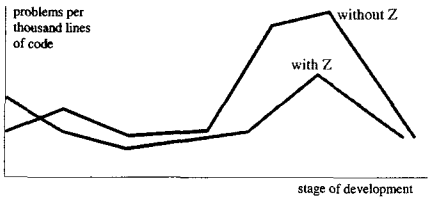



그림 2 Qualitative Results [20]

그림 2와 같이 개발 단계에 정형적 표현을 사용하여 발생할 수 있는 문제점들을 많이 줄일 수 있음을 알 수 있다. 초기 단계는 정형적 표현으로 인한 문제점들이 기본 방식 보다 더 발생할 수 있지만 개발 프로세스가 진행됨에 따라 초기에 명확한 설계로 인해 문제 발생률이 현저하게 감소한다.

이와 같이 소프트웨어를 명세하고 설계하기 위한 표기법으로 Z표기법이 있으며 데이타의 구조와 시스템의 상태, 그리고 속성을 정형적으로 표현하기 위해 사용할 수 있다. 또한 설계의 의도나 개발 단계의 검증을 위해 사용될 수 있다[21]. Z표기법에서 제공하는 중요한 요소는 수학적 언어(Mathematical Language), 스키마 언어(Schema Language), 그리고 정제 이론(Theory of Refinement)이다.

수학적 언어는 집합의 기본적인 기능과 관계를 나타내기 위한 기능들로 구성된다.

Z언어의 두 번째 요소인 스키마 언어는 명세를 작은 단위로 분리하기 위해 방법으로 수학적인 기술을 구조화 시키고 구성하기 위한 방법이다.

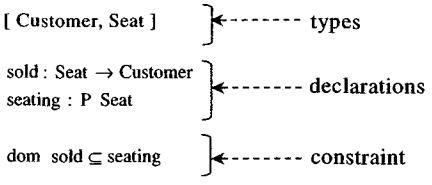



그림 3 Declaration과 Constraint

수학적 기술이란 선언(Declaration)과 제약(Constraint)를 이용하여 표현하는 것으로 그림 3과 같이 선언은 기능이나 데이타의 타입에 대한 선언을 하는 것이며 제약은 선언한 기능이나 데이타에 대한 조건을 나타낸다. 스키마는 이러한 수학적인 기술 요소인 선언과 제약을 이용하여 구조적으로 표현하기 위한 언어이다.

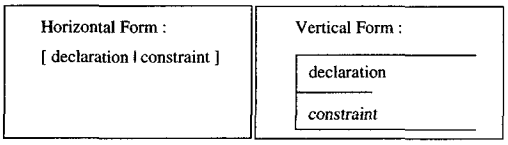



그림 4 Schema Form

스키마의 표현은 그림 4와 같이 수평적 폼과 수직적 폼으로 표현할 수 있다. 수평적 폼은 선언과 제약을 파이프라인('|')으로 구분하여 표현하며 수직적 폼은 박스 내에 수평 라인을 기준으로 상단은 선언을, 하단은 제약을 표현한다. 이와 같이 Z언어는 스키마를 이용하여 명세를 표현하며 스키마는 기능을 작은 단위로 분리하는데 유용하게 이용된다.

Z언어는 명세 수준에서의 정의를 구현 수준으로 설계하기 위한 기술인 정제(Refinement)를 정의하고 있다. 기본 아이디어는 명세 수준에서 나타내는 추상적인 데이타를 구현 수준에서는 구체적인 데이타 구조로 표현하는 것이다. 이러한 구체적인 데이타 구조를 이용하여 구체적인 기능도 유도할 수 있다. 정제 기술로는 데이타 정제(Data Refinement), 오퍼레이션 정제(Operation Refinement), 알고리즘 정제(Algorithm Refinement), 직접 정제(Direct Refinement), 연기 정제(Deferred Refinement) 가 있다.

Z언어의 표기 중에 본 논문에서 자주 사용되는 표기는 다음과 같다.

표 1 Z 언어의 표기

| 표 기 | 설 명 |
|---|---|
| seq X | 집합 X에 속하는 엘리먼트(Element)들의 순서화된 집합 |
| P ∨ Q | Disjunction "P or Q or both" |
| FS | Set of finite subsets of S |

## 2.4 LOTOS

LOTOS는 분산, 병렬 시스템에 적용 가능한 OSI (Open System Interconnection) 아키텍쳐의 정형적 표현을 위해 개발된 ISO(International Standardization Organization) 표준 명세 언어이다. LOTOS는 단일 표기법을 사용해 소프트웨어 시스템의 구조적, 행위적 관

표 2 Operators of LOTOS

| 오퍼레이터 | 표 기 | 설 명 |
|---|---|---|
| Action prefix | ; | a; C는 a의 실행 후에 C의 행위가 처리되는 것을 의미 |
| Enabling | >> | C>>D는 C의 처리가 성공적으로 처리되었을 때만 D가 처리되는 것을 의미 |
| Disabling | [> | C[>D는 C의 처리가 성공적으로 처리되지 않았을 때 D가 처리되는 것을 의미 |
| Choice | [] | C[]D는 C또는 D가 처리되는 것을 의미 |
| Input | ? | 프로세스 정의에서 gates를 통한 입력 |
| Output | ! | 프로세스 정의에서 gate를 통한 출력 |
| Guard | []-> | [C]->는 C조건이 true일 때 다음에 정의된 행위를 처리 |
| Comment | (* *) | (* Comment *) 주석 |

점을 표현하는데 문제가 없다. 또한 객체 지향 시스템의 모델링과 설계 프로세스를 표현하는데 LOTOS의 적합성 연구가 진행되어 왔다[22,23].

LOTOS는 프로세스를 명세하기 위해 그림 5와 같이 LOTOS 프로세스 정의 문법을 이용한다. "PROCESS", "WHERE", "ENDPROC"는 프로세스를 정의 하는 기본 구조가 되며 프로세스의 선언은 프로세스 명("<process-name>"), 시스템 외부로부터의 데이타를 받는 게이트("[<gates>]"), 파라메터("(<params>)"), 그리고 결과값("<funct>:=")으로 구성된다. "PROCESS"와 "WHERE" 사이에 현 프로세스를 나타내기 위한 행위를 명세 하며, "WHERE"와 "ENDPROC"사이에는 현 프로세스 내에서 지역적으로 사용할 필요가 있는 데이타 타입이나 프로세스를 정의한다.

```
PROCESS <process-name> [<gates>] (<params>) : <funct> :=
    <behavior>
WHERE
    <local definitions>
ENDPROC
```

그림 5 Definition of a LOTOS Process

프로세스를 정의하기 위해 LOTOS에서 정의하는 오퍼레이터는 표 2와 같다.

본 논문에서는 제안된 가변성 설계 기법의 절차를 LOTOS를 이용해 정의하며 LOTOS 프로세스 정의에서 이용되는 타입은 Z 명세 언어에서 정의된 스키마를 이용하여 표현한다. 부분적으로 LOTOS에서 표기하기 어려운 부분은 본 논문에서 정의하여 표현한다.

## 3. 커스터마이제이션 프로세스

가변성은 컴포넌트 개발 단계와 컴포넌트 기반 어플리케이션 개발 단계에 설계될 수 있다. 이와 같이 컴포넌트 개발 단계에서는 초기 가변성이 설계되며 컴포넌트를 이용하여 어플리케이션을 개발하는 단계에서는 초기에 설계된 가변성에 대해 발전된 가변성을 설계할 수 있다.

컴포넌트 개발 단계에 설계된 가변성은 어플리케이션 개발 시 커스터마이제이션 기법을 적용하여 변경될 수 있다. 제공된 컴포넌트의 기능들 중에 가변성으로 제공된 기능 외에 다른 비가변성 기능들이 어플리케이션 개발 시에 변경을 요구할 수 있으며 이러한 예외적인 변경 사항을 반영할 수 있도록 커스터마이제이션 기법을 적용할 수 있다.

기존 초기 가변성으로 설계된 기능 외의 기능들이 "Overriding"이나 "Overloading", 또는 "Message Flow"의 형태로 변경을 요구하는 경우에 추가적으로 가변성을 설계한다. 변경 사항은 표 3과 같이 정의할 수 있다.

표 3 컴포넌트 변경 사항

| 변경 사항 | 설 명 |
|---|---|
| 기능 추가 | 새로운 Operation 추가 |
| Overriding | 기존 Operation의 Signature는 변경하지 않고 다른 기능으로 변경 |
| Overloading | 기존 Operation의 함수 명은 동일하고 Parameter와 Return Type 을 변경하여 새로운 기능으로 추가 |
| Message Flow 변경 | 컴포넌트 내의 메시지 흐름에 대한 변경 |

표 3에서의 컴포넌트 변경 사항 중에 "Overriding"과 "Message Flow" 변경은 기존 기능에 대한 변경 사항이기 때문에 커스터마이제이션 기법을 적용하여 제공되어야 할 가변성 범위이다. 그러나 "기능 추가"나 "Overloading"은 기존 기능이 아니라 새로운 기능에 대한 추가이므로 가변성의 범위가 되지 않는다. 결과적으로 "기능 추가"나 "Overloading"은 컴포넌트 개발 시에 기능을 고려하지 않고 설계되었기 때문에 도메인에 적용할 때 부족한 기능으로 판단되어 추가하는 결과가 발생하므로 컴포넌트 설계가 완전하지 않았음을 나타낸다. -

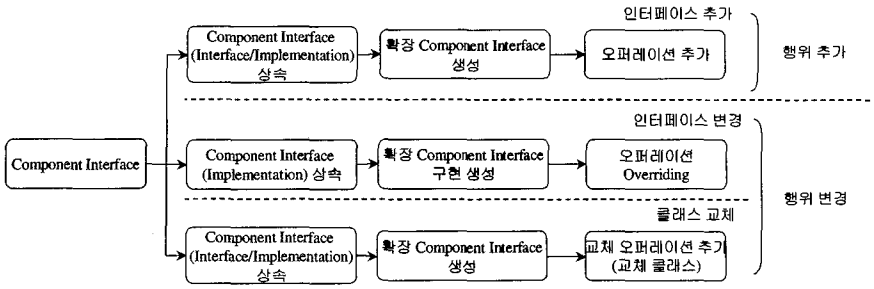



그림 6 행위 커스터마이제이션 프로세스

본 논문에서는 "기능 추가"나 "Overloading"이 가변성의 범위는 아니지만 도메인에 적용되면서 컴포넌트의 일반성을 향상시키는 요소로 고려하여 초기 컴포넌트의 기능으로 추가할 수 있다. 따라서 추가된 기능도 변경 가능성을 판단하여 커스터마이제이션 기법을 제안한다.

### 3.1 행위 커스터마이제이션 프로세스

컴포넌트 행위에 대한 커스터마이제이션은 행위 추가와 행위 변경으로 크게 구분하며 행위 변경은 인터페이스의 변경과 클래스 교체로 구분한다. 각각의 개략적인 프로세스는 그림 6과 같다.

행위 추가는 기존 컴포넌트 인터페이스를 상속 받아 확장 컴포넌트 인터페이스를 생성한다. 이렇게 생성된 확장 컴포넌트 인터페이스는 새로운 기능을 추가할 수 있는 인터페이스가 된다. 또한 확장된 컴포넌트 인터페이스는 기존 컴포넌트 인터페이스를 상속 받으므로 기존 컴포넌트에서 제공되는 기능을 모두 사용할 수 있다.

행위 변경 중에 인터페이스 변경은 기존 컴포넌트 인터페이스 중에 일부 구현 부분만 상속 받으며 기존 컴포넌트 인터페이스의 스펙은 변경하지 않는다. 행위를 변경하기 위해 확장된 컴포넌트 인터페이스의 구현 함수 중에 변경하고자 하는 함수를 오버라이딩하여 변경한다.

행위 변경 중에 클래스 교체는 컴포넌트 내에 존재하는 클래스를 교체하여 컴포넌트 내의 기능을 변경한다. 기존 컴포넌트 인터페이스를 상속 받아 새로운 확장 컴포넌트 인터페이스를 생성하며 클래스를 교체하기 위한 교체 오퍼레이션을 추가한다.

## 4. 커스터마이제이션 기법

커스터마이제이션 기법은 앞에서 제시된 커스터마이제이션 프로세스에 따라 컴포넌트 개발 단계와 컴포넌트를 이용하여 어플리케이션을 개발 단계에 모두 적용될 수 있다. 이러한 커스터마이제이션의 기법에 대해 명확성을 줄 수 있도록 Z 언어를 이용하여 컴포넌트를 정의하며 기법의 절차를 LOTOS를 통해 정의한다.

### 4.1 Foundation

컴포넌트는 그림 7에서와 같이 컴포넌트를 구성하는 클래스와 컴포넌트 기능을 외부에 제공하기 위한 컴포넌트 인터페이스로 구성된다. 컴포넌트 인터페이스는 기능성을 분리하기 위해 한 개 이상의 인터페이스로 구성될 수 있다[8]. 인터페이스는 자바의 인터페이스나 클래스가 될 수 있으며 컴포넌트 내의 클래스들의 기능을 호출하기 위한 역할을 한다. 컴포넌트 인터페이스는 클래스의 기능을 호출하기 위한 호출 함수로 구성된다.

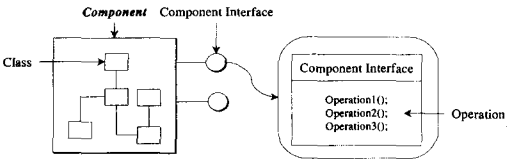



그림 7 컴포넌트 구성

컴포넌트 인터페이스 기법으로는 그림 8의 (a)와 같이 자바의 인터페이스 형태로 제공하는 퍼사드 인터페이스(Facade-Interface) 패턴과 일반 클래스로 컴포넌트 내부의 클래스를 호출하는 그림 8의 (b)와 같은 퍼사드 클래스(Facade-Class) 패턴으로 구성될 수 있다. (b)와 같은 기법은 일반적으로 사용될 수 있는 기법이며 (a)의 기법은 인터페이스에 대해 현재의 구현 클래스 대신 다르게 구현하여 제공할 수 있는 팩토리(Factory)의 구조가 될 수 있다.

가변성이란 다양한 도메인에 컴포넌트를 적용 시 변

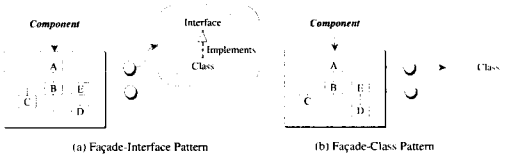



그림 8 컴포넌트 인터페이스 패턴

경 가능성이 있는 기능을 말하며 컴포넌트 인터페이스의 기능 단위나 컴포넌트 내의 클래스 단위가 될 수 있다[3]. 이러한 컴포넌트 가변성에 대해 어플리케이션에서 이용할 때 대응할 수 있도록 커스터마이제이션 기법을 제공해 주어야 한다. 커스터마를 도메인의 요구 사항에 맞도록 컴포넌트를 변경하는 것을 말한다.

초기 가변성의 설계는 요구 인터페이스(Required Interface)를 대상으로 설계하며 이 초기 가변성 설계를 이용하여 커스터마이제이션 기법을 적용한다. 요구 인터페이스는 컴포넌트에서 컴포넌트 외부로 요구하는 기능을 말하며 컴포넌트를 사용하는 측면에서 인터페이스에 맞게 구현하여 컴포넌트 내부로 넣어주어야 하는 기능을 말한다. 따라서 컴포넌트를 적용하는 외부에서 각각의 요구 사항에 맞게 다양하게 구현하여 넣어도 컴포넌트 내부는 전혀 변경되지 않는다. 요구 인터페이스와는 반대로 컴포넌트에서 제공되는 인터페이스를 제공 인터페이스(Provided Interface)라고 한다.

### 4.2 행위 커스터마이제이션 기법

앞에서 정의한 행위 가변성(Behavior Variability)은 컴포넌트 인터페이스 오퍼레이션의 추가, 변경 그리고 컴포넌트 내의 클래스 변경을 의미한다. 컴포넌트 인터페이스 오퍼레이션의 변경은 컴포넌트 내의 클래스들을 호출하는 기능에 대한 변경을 의미하며 컴포넌트 내의 클래스 오퍼레이션의 변경은 컴포넌트의 기능을 변경하는 것으로 클래스 교체를 통해 변경될 수 있다. 본 논문에서 컴포넌트 인터페이스는 앞에서 정의한 컴포넌트 인터페이스 패턴 중에 패서드 인터페이스(Facade-Interface) 패턴을 사용한다. 즉 이 패턴은 컴포넌트 인터페이스가 오퍼레이션 선언과 구현을 모두 갖고 있는 경우를 말하며 오퍼레이션 선언에 대한 구현이 컴포넌트 내부로 들어 갈 수도 있다.

4.2.1 컴포넌트 인터페이스 명세

행위 커스터마이제이션을 위한 컴포넌트 인터페이스는 그림 9과 같이 컴포넌트 인터페이스("Component Interface")를 구성하는 인터페이스("component_interface")와 구현 클래스("component_interface_impl")로 구성된다(Facade-Interface 패턴). "component_interface_impl"는 컴포넌트 내의 클래스("A,B,C,D,E")의 오퍼레이션을 호출하고 있으며, 변경 시 컴포넌트 인터페이스의 이러한 호출 정보를 변경한다.

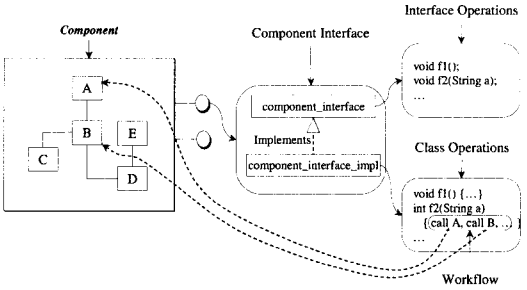



그림 9 컴포넌트 인터페이스 메타 모델

컴포넌트 인터페이스의 명세는 Z 언어를 통해 표현하며 그림 4의 스키마 폼 중에서 수직적 폼을 이용하여 표현한다. 컴포넌트 인터페이스를 명세하기 전에 기본 스키마들을 정의한다.

[STRING, TYPE, LOGIC]

기본 타입으로 문자열인 "STRING", 데이타 타입을 나타내는 "TYPE", 그리고 기능을 나타내는 "LOGIC"을 정의한다.

PARAMETER

type : TYPE
name : STRING

정의 1 파라메터 명세

정의 1은 오퍼레이션의 입력 데이타를 나타내는 파라메터의 정의로서 파라메터의 데이타 타입과 문자열인 파라메터 명으로 정의한다.

VISIBILITY ::= private | public | protected

"VISIBILITY"는 오퍼레이션이나 객체 속성의 가시성을 나타내기 위해 것으로 "private", "public", "protected" 중에 하나로 정의될 수 있다.

OPERATION_SIGNATURE

visibility : VISIBILITY
returnType : TYPE
operationN ame : STRING
parameters : seq PARAMETER

$\forall p1, p2 : ran\ parameters \bullet p1.name \neq p2.name \Rightarrow p1 \neq p2$

정의 2 오퍼레이션의 시그니처 명세

정의 2는 오퍼레이션의 시그니처를 나타내는 스키마로서 가시성, 결과 데이타 타입, 그리고 오퍼레이션 명, 그리고 파라메터들로 구성된다. 파라메터들의 명은 서로 같지 않고 유일해야 한다.

OPERATION

signature : OPERATION_ SIGNATURE
body : LOGIC

정의 3 객체의 오퍼레이션 명세

정의 3은 오퍼레이션을 나타내는 스키마로서 정의 2에서 정의한 시그니처와 시그니처의 기능을 담고 있는 "LOGIC"으로 구성된다. "LOGIC"은 컴포넌트 내의 클래스 오퍼레이션들을 호출하는 메시지 흐름을 의미한다.

ATTRIBUTE

visibility : VISIBILITY
type : TYPE
name : STRING

정의 4 객체의 속성 명세

정의 4는 객체의 속성 명세로서 가시성과 타입, 그리고 속성 명으로 구성된다. 속성은 객체를 구성하는 데이타 멤버를 의미한다.

CLASS

name : STRING
attributes : F ATTRIBUTE
operations : F OPERATION

정의 5 클래스 명세

정의 5는 클래스 명세를 나타내는 스키마로서 클래스 명과 속성(정의 4)들, 오퍼레이션(정의 3)들로 정의한다.

INTERFACE

name : STRING
operation_signatures : F OPERATION_SIGNATURE

정의 6 인터페이스 명세

정의 6은 오퍼레이션의 정의 부분이 없는 시그니처만을 가진 오퍼레이션으로 구성된 인터페이스에 대한 명세이다.

WORKFLOW

operations : seq OPERATION

정의 7 메시지 흐름 명세

정의 7의 메시지 흐름 명세는 순차적인 오퍼레이션들의 흐름을 명세 한다. 이 메시지 흐름은 컴포넌트 내의 클래스 오퍼레이션의 순차적인 흐름을 나타내가 위해 사용된다.

OPERATION I WORKFLOW ∧ WORKFLOW ⊆ LOGIC

메시지 흐름 스키마인 "WORKFLOW"는 컴포넌트 내부의 클래스 오퍼레이션을 포함하며 앞에서 정의한 "LOGIC" 타입의 부분 집합이 된다.

가변성 설계 기법을 정의할 때 객체 간의 관계를 나타내기 위해 구현 관계 ("IMPLEMENTS_ASSOCIATION")와 상속 관계("INHERITANCE_ASSOCIATION")를 정의한다.

IMPLEMENTS_ASSOCIATION

super: INTERFACE
sub: CLASS
operation: OPERATION
operation_signature: OPERATION_SIGNATURE

∀os : operation_signature,o : operation • super.os=sub.o.signature

정의 8 구현 관계 명세

정의 8은 인터페이스의 오퍼레이션들을 구현하는 클래스와의 관계를 정의하는 스키마로서 상위("super")는 인터페이스를 나타내고 하위("sub")는 이 인터페이스를 구현하는 클래스를 나타낸다. 인터페이스의 시스니처와 클래스 오퍼레이션의 시그니처는 반듯이 같아야 한다.

INHERITANCE_ASSOCIATION

super : CLASS I INTERFACE
sub : CLASS I INTERFACE
operation_ signature : OPERATION_ SIGNATURE
operation : OPERATION

∀os : operation_ signature, o : operation •
(super.os ≠ sub.os ∨ super.o.si gnature ≠ sub.o.sign ature) ∧
(super.os ⊆ sub.os ∨ super.o ⊆ sub.o)

정의 9 상속 관계 명세

정의 9는 상속 관계를 나타내는 스키마로서 상위는

COMPONENT_INTERFACE_FACTORY

interface : INTERFACE
classes : F CLASS
operation : OPERATION
operation_ signature : OPERATION_ SIGNATURE

∀os : operation_ signature, o : operation • interface. os = classes.o. signature

정의 10 컴포넌트 인터페이스 팩토리

클래스이거나 인터페이스가 될 수 있으며 하위는 상위의 타입과 동일해야 한다. 상속 관계에서는 상위에 있는 시그니처는 하위의 시그니처와 동일하지 않아야 한다. 그리고 상위의 시그니처나 오퍼레이션은 하위의 인터페이스나 클래스에 포함된다.

정의 10은 컴포넌트 인터페이스의 구현 클래스들을 관리하기 위한 팩토리에 관한 명세로서 하나의 인터페이스에 대해 구현된 여러 클래스들을 갖는다.

```
COMPONENT_INTERFACE
component_ interface :INTERFACE
component_ interface_ impl :CLASS
implements :IMPLEMENTS _ASSOCIATI ON
internal_c lass :CLASS
workflow :WORKFLOW
operation :OPERATION
-----------------------------
implements .super = component_ interface
implements .sub = component_ interface_ impl

∃o : operation • (internal_ class.o ∈ component_ interface_ impl.opera tions ) ∨
(workflow ∈ component_ interface_ impl.opera tions )
```

정의 11 컴포넌트 인터페이스 명세

컴포넌트 인터페이스 명세는 인터페이스인 "component_interface"와 이 인터페이스를 구현하는 클래스인 "component_interface_impl"로 구성되며 이 인터페이스와 클래스 간의 관계는 "implements"로 정의한다. 컴포넌트 인터페이스 클래스의 오퍼레이션은 컴포넌트 내부의 클래스 오퍼레이션이나 메시지 흐름(워크플로우)을 포함한다.

컴포넌트 인터페이스에 대한 명세는 3가지 행위 커스터마이제이션 기법을 제안하는데 기반이 되며 행위 커스터마이제이션 기법들이 컴포넌트의 블랙 박스를 유지하며 변경될 수 있도록 한다.

4.2.2 행위 추가 기법(인터페이스 추가)

행위 추가 커스터마이제이션 기법은 그림 10와 같이 기존 컴포넌트 인터페이스를 상속 받아 확장 컴포넌트 인터페이스를 설계하는 기법이다. 확장 컴포넌트 인터페이스는 기존 컴포넌트 인터페이스의 인터페이스("component_interface")와 클래스("component_interface_impl")를 상속 받으며 확장 컴포넌트 인터페이스의 인터페이스("extend_component_interface")와 클래스("extend_component_interface _impl")에 새로운 기능을 추가한다. 이와 같은 설계 기법은 주로 인터페이스 API를 새로 추가하거나 기존 함수에 대해 오버로딩(Overloading)해야 하는 경우에 적용되는 기법이다.

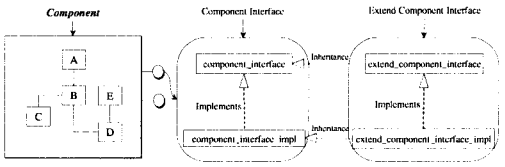



그림 10 인터페이스 추가 설계 메타 모델

본 기법은 기존 컴포넌트의 영속성을 유지하면서 컴포넌트의 기능을 추가할 수 있는 기법으로 기존 컴포넌트 인터페이스는 전혀 변경하지 않고 컴포넌트의 기능을 추가한다.

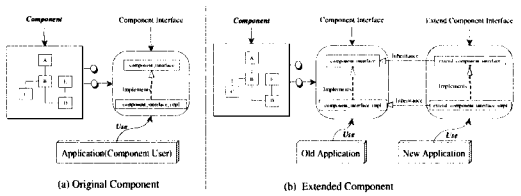



그림 11 컴포넌트 인터페이스의 사용

기능이 추가된 컴포넌트를 컴포넌트 사용자(Component User)에 의해 사용될 때 기존 컴포넌트 인터페이스("Component Interface")는 사용하지 않으며, 그림 11의 (b)와 같이 확장된 컴포넌트 인터페이스("Extend Component Interface")를 사용 한다. 확장된 컴포넌트 인터페이스는 추가된 기능을 포함해서 기존 컴포넌트 인터페이스("Component Interface")를 상속 받고 있으므로 기존 기능도 사용할 수 있다.

행위 추가 커스터마이제이션 기법의 프로세스는 정의 12과 같이 정의한다. 본 논문에서는 커스터마이제이션 기법의 절차를 표현하기 위해 LOTOS를 이용한다. LOTOS의 프로세스 정의를 이용하여 표현하며 부분적으로 Z 언어에서 정의한 명세를 프로세스의 타입으로 이용한다.

```
PROCESS Add_Behavior_Design_Process[interface, extend_interface, extend_class]
  :exit :=

  (* "interface" is the existing component interface, which is the input-gate *)
  (* "extend_interface" is the "extend_component_interface", which is the output-gate *)
  (* "extend_class" is the "extend_component_interface_impl", which is the output-gate *)

  Inherit_Component_Interface[interface]
  >> accept extend_component_interface : INTERFACE
        In Inherit_Component_Class[interface] [extend_component_interface]
  >> accept extend_component_interface : INTERFACE,
            extend_component_interface_impl : CLASS
        In Design_Operation_Of_Extend_Class[interface]
            [extend_component_interface, extend_component_interface_impl]
  >> accept extend_component_interface : INTERFACE,
            extend_component_interface_impl : CLASS
      In
    extend_interface ! extend_component_interface
    : extend_class ! extend_component_interface_impl
  : exit
ENDPROC
```

정의 12 행위 추가 커스터마이제이션 프로세스

행위 추가 커스터마이제이션 프로세스는 기존 컴포넌트 인터페이스를 이용하여 추가 설계를 진행하며 최종적으로 설계된 확장 컴포넌트 인터페이스를 제공한다. 본 프로세스에서는 기존 컴포넌트 인터페이스를 입력으로 받기 위해 "interface" 게이트를 이용하며 확장 인터페이스를 출력으로 제공하기 위해 "extend_interface", "extend_class" 게이트를 이용한다. 행위 추가 가변성 설계 프로세스는 컴포넌트 인터페이스 상속("Inherit_Component_Interface"), 컴포넌트 인터페이스 구현 클래스 상속("Inherit_Component_Class"), 확장 클래스의 오퍼레이션 설계("Design_Operation_Of_Extend_Class")로 진행된다. 각각의 프로세스에서 처리된 결과는 다음 프로세스의 입력 파라메터로 이용될 수 있도록 "Enabling(>>)" 오퍼레이터와 "accept <params> in <process-definition>"를 이용한다. "Enabling(>>)" 오퍼레이터는 전 행위가 반드시 성공해야만 다음 행위를 할 수 있으므로 전 프로세스가 성공적으로 설계가 된 후 처리 결과가 다음 프로세스로 전달될 수 있다.

행위 추가 커스터마이제이션 프로세스내에 포함된 서브 프로세스들에 대해 각각 정의한다.

● 컴포넌트 인터페이스 상속

본 프로세스는 확장 인터페이스를 정의하기 위해 기존 컴포넌트 인터페이스를 상속 받는 단계이다. 확장 인터페이스는 기존 컴포넌트 인터페이스의 모든 시그니처를 포함하므로 컴포넌트를 사용할 때 확장된 인터페이스 만을 접근하여 컴포넌트의 모든 기능을 사용할 수 있다.

```
(* Inherit "component_interface" *)
PROCESS Inherit_Component_Interface[interface]
   :exit(extend_component_interface) :=

    interface ? component_IF : COMPONENT_INTERFACE
  : type inheritance is INHERITANCE_ASSOCIATION endtype
  : type extend_component_interface is INTERFACE endtype
  : type operation_signature is OPERATION_SIGNATURE endtype

  : inheritance.super := component_IF.component_interface
  : inheritance.sub := extend_component_interface

  : ({ isExist(component_IF.component_interface, operation_signature)= false ]->
       : extend_component_interface.operation_signature := operation_signature
       : exit (extend_component_interface) )
    []
    ([ isExist(component_IF.component_interface, operation_signature)= true ]->
       [> abort )
  : exit(extend_component_interface)
WHERE
   PROCESS isExist(component_interface : INTERFACE,
                   operation_signature:OPERATION_SIGNATURE)
      :exit(bool) :=
      ([(operation_signature ∈ component_interface)=true]->
         : exit(false) )
      []
      ([(operation_signature ∈ component_interface)=true]->
         : exit(true) )
   ENDPROC
ENDPROC
```

정의 13 컴포넌트 인터페이스 상속 프로세스

정의 13와 같이 오퍼레이션이 추가된 확장 인터페이스는 기존의 컴포넌트 인터페이스를 상속 받기 위해 타입이 "INHERITANCE_ASSOCIATION"을 이용하여 상속 관계를 나타낸다.

추가되는 오퍼레이션("operation_signature")은 외부로부터 컴포넌트의 인터페이스("component_IF")를 입력으로 받아 기존 컴포넌트 인터페이스("component_interface")에 존재하는지 비교하여 추가한다. 추가하려는 오퍼레이션이 기존 컴포넌트의 인터페이스에 존재하는지 비교하기 위해 "WHERE" 절에 "isExist"라는 서브 프로세스를 정의한다.

● 컴포넌트 인터페이스의 구현 클래스 상속

본 프로세스는 확장 클래스가 컴포넌트의 기능을 이용하기 위해 기존 컴포넌트 인터페이스의 클래스를 상속 받는다. 확장 클래스는 기존 컴포넌트의 기능을 모두 포함하기 때문에 이 확장 클래스만 접근하면 컴포넌트의 모든 기능을 사용할 수 있다.

```
(* Inherit "component_interface_impl" *)
PROCESS Inherit_Component_Class[interface]
   (extend_component_interface : INTERFACE)
   :exit(extend_component_interface_impl) :=

    interface ? component_IF : COMPONENT_INTERFACE
  : type inheritance is INHERITANCE_ASSOCIATION endtype
  : type extend_component_interface_impl is CLASS endtype

  : inheritance.super := component_IF.component_interface_impl
  : inheritance.sub := extend_component_interface_impl

  : extend_component_interface_impl.operation.signature := extend_component_interface.signature
  : exit(extend_component_interface_impl)

ENDPROC
```

정의 14 컴포넌트 인터페이스 구현 클래스 상속 프로세스(추가)

정의 14와 같이 기능을 추가하기 위한 확장 클래스를 정의하고 입력 게이트로부터 받은 컴포넌트의 클래스("component_IF.component_interface_impl")를 상속 받는다. 정의된 확장 클래스는 앞의 프로세스로부터 확장 인터페이스를 입력 받으며 확장 클래스에 추가되는 오퍼레이션은 확장 인터페이스에 선언된 시그니처를 이용한다.

● 확장 클래스의 오퍼레이션 설계(추가)

본 프로세스는 확장 클래스에 추가되는 오퍼레이션의

```
(* Design the operation of "extend_component_interface_impl" *)
PROCESS Design_Operation_Of_Extend_Class[interface]
   (extend_component_interface : INTERFACE, extend_component_interface_impl : CLASS )
   :exit( extend_component_interface, extend_component_interface_impl ) :=

    interface ? component_IF : COMPONENT_INTERFACE
  : type implement is IMPLEMENT_ASSOCIATION endtype
  : type workflow is WORKFLOW endtype
  : type internal_class is CLASS endtype
  : type operation is OPERATION endtype

  : implement.super := extend_component_interface
  : implement.sub := extend_component_interface_impl

  : ( [ operation ∈ internal_class.operation ]->
      : operation := internal_class.operation )
  []
    ( [ operation ∈ workflow ]->
      : operation := workflow
  []
    ( [ operation ∈ component_IF.component_interface_impl.operation ]->
      : operation := component_IF.component_interface.operation_signatures )

  : extend_component_interface_impl.operation.body := operation
  : exit( extend_component_interface, extend_component_interface_impl )

ENDPROC
```

정의 15 확장 클래스의 추가 오퍼레이션 설계 프로세스

기능을 설계하는 단계로서 컴포넌트 인터페이스에 추가가 가능한 기능을 식별하여 설계한다.

정의 15와 같이 확장 클래스의 추가 오퍼레이션을 설계하기 위해서는 확장 인터페이스와 확장 클래스를 인플리먼트 관계(Implement association)로 설정하며 확장 클래스의 오퍼레이션 바디(body)에 새로운 기능을 추가한다. 추가되는 오퍼레이션의 종류는 다음과 같다.

표 4 추가 오퍼레이션 종류

| | |
|---|---|
| 컴포넌트 내의 클래스 호출 | 컴포넌트 내의 하나의 클래스 오퍼레이션 호출 |
| | 컴포넌트 내의 다중의 클래스 오퍼레이션을 통한 메시지 흐름 호출 |
| 기존 컴포넌트 인터페이스의 오퍼레이션 호출 | |

표 4에서 같이 추가되는 오퍼레이션의 종류는 컴포넌트 내부의 클래스 호출과 기존 컴포넌트 인터페이스를 이용하여 새로운 기능을 생성하는 두 가지 경우로 볼 수 있다. 컴포넌트 내의 클래스 호출은 임의의 하나의 클래스 오퍼레이션을 호출하거나 여러 클래스들의 오퍼레이션을 호출하는 경우로 구분할 수 있다. 확장 컴포넌트 인터페이스의 구현 클래스는 이러한 조건에 맞는 오퍼레이션을 추가한다.

4.2.3 행위 변경 기법(인터페이스 변경)

행위 변경 커스터마이제이션 기법은 그림 12와 같이 기존 컴포넌트 인터페이스를 상속하여 확장 클래스를 설계한다. 컴포넌트 인터페이스("Component Interface")에 있는 인터페이스("component_interface")는 상속 받지 않고 구현 클래스("component_interface_impl")만 상속을 받아 설계한다. 확장된 인터페이스 클래스("extend_component_interface_impl")는 기존 컴포넌트 인터페이스의 동일한 인터페이스("component_interface")를 사용하기 때문에 따로 인터페이스를 확장하여 인플리먼트 하지 않으며 변경하려는 함수를 재 정의하여 기존 함수를 변경한다.

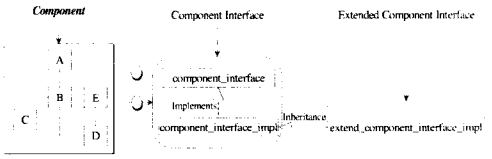



그림 12 인터페이스 변경 설계 메타 모델

동일한 시그니처에 대해 기능을 재 정의하는 이러한 설계는 어플리케이션에서 컴포넌트를 사용할 때 원하는 기능을 포함한 구현 클래스 중에 선택하도록 설계해야 한다.

이러한 기법은 행위 추가 커스터마이제이션 기법에서 컴포넌트 사용자가 확장 컴포넌트 인터페이스를 사용해야 하는 것과는 다르게 컴포넌트 사용자가 선택적으로 컴포넌트 인터페이스를 사용할 수 있다.

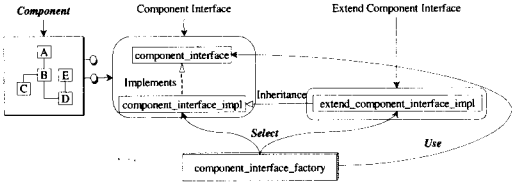



그림 13 컴포넌트 인터페이스 구현 클래스 선택

기존 구현 클래스("component_interface_impl")나 확장된 구현 클래스("extend_component_interface_impl")를 사용할 수 있도록 팩토리("component_interface_factory")를 제공하여 컴포넌트 사용자들이 선택적으로 사용할 수 있도록 한다. 서로 다른 구현 클래스를 통해 컴포넌트 내부의 기능을 호출하더라도 인터페이스는 동일하기 때문에 컴포넌트의 다른 부분(컴포넌트 내부, 어플리케이션)에는 전혀 영향을 주지 않고 변경할 수 있다.

행위 변경 커스터마이제이션 프로세스는 정의 16과 같이 정의한다.

```
PROCESS Change_Behavior_Design_Process
  [interface, extend_class]
  :exit :=

  (* "interface" is the "component_interface", which is the input-gate *)
  (* "extend_class" is the "extend_component_interface_impl", which is the output-gate *)

  Inherit_Component_Class[interface]
  >> accept extend_component_interface_impl : CLASS
       In Redefine_Operation[interface](extend_component_interface_impl)
  >> accept extend_component_interface_impl : CLASS
       In Register_Extend_Class[interface](extend_component_interface_impl)
  >> accept extend_component_interface_impl : CLASS
       In extend_class ! extend_component_interface_impl
  : exit
ENDPROC
```

정의 16 행위 변경 커스터마이제이션 프로세스

행위 변경 커스터마이제이션 프로세스는 기존 컴포넌트 인터페이스를 이용하여 변경 설계를 진행하며 최종적으로 설계된 확장 클래스를 제공한다. 본 프로세스에서는 기존 컴포넌트 인터페이스를 입력으로 받기 위해 "interface" 게이트를 이용하며 확장 클래스를 출력으로 제공하기 위해 "extend_class" 게이트를 이용한다. 행위 변경 커스터마이제이션 프로세스는 컴포넌트 인터페이스 구현 클래스 상속("Inherit_Component_Class"), 컴포넌트 인터페이스의 오퍼레이션 재정의("Redefine_Op-

eration"), 확장 클래스 등록("Register_Extend_Class")으로 설계가 진행된다.

행위 변경 커스터마이제이션 프로세스 내에 포함된 서브 프로세스들에 대해 각각 정의한다.

● 컴포넌트 인터페이스 구현 클래스 상속

본 프로세스는 행위 추가 커스터마이제이션 프로세스의 컴포넌트 인터페이스 구현 클래스 상속 프로세스(정의 14)와 유사하며 기존 기능을 변경하는 기법이기 때문에 인터페이스를 확장하지 않는다. 확장 클래스는 기존 컴포넌트의 클래스를 상속 받고 있기 때문에 기존 컴포넌트 인터페이스와 인플리먼트 관계를 갖는다.

```
(+ inherit "component_interface_impl" +)
PROCESS Inherit_Component_Class[interface]
  :exit(extend_component_interface_impl) :=

    interface ? component_IF : COMPONENT_INTERFACE
  : type inheritance is INHERITANCE_ASSOCIATION endtype
  : type extend_component_interface_impl is CLASS endtype

  : inheritance.super := component_IF.component_interface_impl
  : inheritance.sub := extend_component_interface_impl

  (+ : extend_component_interface_impl.operation.signature := extend_component_interface.signature
     행위 추가 커스터마이제이션 프로세스에 존재하며 행위 변경에서는 확장 인터페이스가 없음. +)
  : exit(extend_component_interface_impl)

ENDPROC
```

정의 17 컴포넌트 인터페이스 구현 클래스 상속 프로세스(변경)

정의 17과 같이 행위 추가 커스터마이제이션 프로세스와 유사하며 확장 클래스의 오퍼레이션 시그니처를 따로 정의하지 않고 상위 클래스의 오퍼레이션을 그대로 이용한다.

● 컴포넌트 인터페이스 구현 클래스의 오퍼레이션 재정의

본 프로세스는 기존 기능을 재정의하는 단계이다. 상속 받은 확장 클래스는 변경을 요구하는 기능을 재정의하여 기능을 변경한다.

```
(+ Redefine the operation of the "component_interface_impl" +)
PROCESS Redefine_Operation[interface]
   ( component_interface_impl:CLASS)
   :exit(extend_component_interface_impl) :=

   interface ? component_IF : COMPONENT_INTERFACE
  : type workflow is WORKFLOW endtype
  : type internal_class is CLASS endtype
  : type operation is OPERATION endtype

  : ( [ operation ∈ internal_class.operation ]->
      : operation := internal_class.operation )
  []
    ( [ operation ∈ workflow ]->
      : operation := workflow
  []
    ( [ operation ∈ component_IF.component_interface_impl.operation ]->
      : operation := component_IF.component_interface.operation_signatures )

  : extend_component_interface_impl.operation := component_interface_impl.operation ⊕ operation
  : exit(extend_component_interface_impl)
ENDPROC
```

정의 18 오퍼레이션 재 정의 프로세스

정의 18과 같이 기존 컴포넌트의 기능을 변경하기 위한 프로세스로서 기존 컴포넌트의 오퍼레이션을 새로 정의된 오퍼레이션으로 오버라이딩(A)하여 재 정의한다. 새로 정의된 오퍼레이션은 컴포넌트 내의 메시지 흐름이나 특정 클래스의 오퍼레이션, 또는 기존 컴포넌트 인터페이스의 오퍼레이션을 호출하는 기능 중에 하나로 정의된다.

● 확장 클래스 등록

확장된 클래스는 동일한 컴포넌트 인터페이스를 통해 제공될 수 있도록 컴포넌트 인터페이스 팩토리에 등록한다. 팩토리를 이용해 변경된 기능을 사용하기 위해서는 "extend_component_interface_impl"을 접근하여 사용하며 재정의되기 전의 기능을 사용하기 위해서는 "component_interface_impl"을 접근하여 사용할 수 있다.

```
(+ Register the "extend_component_interface_impl" +)
PROCESS Register_Extend_Class[interface]
   ( extend_component_interface_impl:CLASS)
   :exit(extend_component_interface_impl) :=

   interface ? component_IF : COMPONENT_INTERFACE
  : type factory is COMPONENT_INTERFACE_FACTORY endtype
  : factory.interface := component_IF.component_interface
  : factory.classes := extend_component_interface_impl
  : exit(extend_component_interface_impl)
ENDPROC
```

정의 19 확장 클래스 등록 프로세스

정의 19은 컴포넌트 인터페이스를 이용해 확장한 클래스를 팩토리에 저장하는 프로세스를 나타낸다. 등록은 컴포넌트 인터페이스와 확장 클래스를 팩토리에 저장하며 동일 컴포넌트 인터페이스에 대해 여러 확장 클래스가 등록된다. 확장한 클래스들은 동일한 인터페이스를 통해 이용되므로 클래스를 선택적으로 이용할 수 있다.

4.2.4 행위 변경(클래스 교체)

클래스 교체 커스터마이제이션 기법은 행위 변경의 기법으로 그림 14과 같이 컴포넌트 인터페이스에 교체 클래스를 입력 받는 오퍼레이션을 정의하여 설계한다. 이 교체 오퍼레이션은 변경된 클래스를 입력 받아 컴포넌트 내부의 해당 클래스와 교체되어 기능을 변경한다.

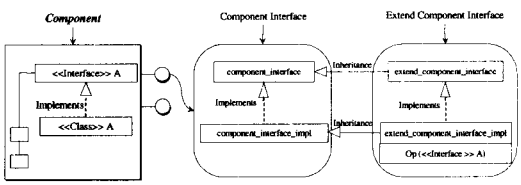



그림 14 클래스 교체 설계 메타 모델

클래스 커스터마이제이션 기법은 교체 오퍼레이션을 추가하는 과정에서는 행위 추가 설계 기법과 동일하며

추가된 교체 오퍼레이션의 기능이 기존 기능을 변경하는 관점에서는 행위 변경 커스터마이제이션 기법과 유사하다.

클래스 교체 커스터마이제이션 프로세스는 정의 20와 같이 정의한다.

```
PROCESS Replace_Class_Design_Process[replace_interface, interface, extend_interface, extend_class]
  :exit :=

(* "replace_interface" is the interface for the replace class, which is the input -gate *)
(* "interface" is the "component_interface", which is the input -gate *)
(* "extend_interface" is the "extend_component_interface", which is the output -gate *)
(* "extend_class" is the "extend_component_interface_impl", which is the output-gate *)

    Implement_Replace_Class[replace_interface]
  : Inherit_Component_Interface[interface]
>> accept extend_component_interface_impl : CLASS
       In  Inherit_Component_Class[interface] (extend_component_interface_impl)
>> accept extend_component_interface : INTERFACE,
          extend_component_interface_impl : CLASS
       In  Design_Replace_Operation_Of_Extend_Class[interface]
           (extend_component_interface, extend_component_interface_impl)
>> accept extend_component_interface : INTERFACE,
          extend_component_interface_impl : CLASS
       In
      extend_interface ! extend_component_interface
      extend_class ! extend_component_interface_impl
  : exit
ENDPROC
```

정의 20 클래스 교체 커스터마이제이션 프로세스

클래스 교체 커스터마이제이션 프로세스는 기존 컴포넌트 인터페이스를 이용하여 클래스 교체 설계를 진행하며 최종적으로 설계된 확장 인터페이스를 컴포넌트 사용자에게 제공한다. 본 프로세스에서는 기존 컴포넌트 인터페이스를 입력으로 받기 위해 "interface", 게이트를 이용하며 확장 인터페이스를 출력으로 제공하기 위해 "extend_interface", "extend_class" 게이트를 이용한다. 교체하기 위한 클래스를 설계하기 위해 "replace_interface" 게이트를 통해 교체 클래스의 인터페이스를 제공한다. 클래스 교체 커스터마이제이션 프로세스는 교체 클래스 구현("Implement_Replace_Class")하는 프로세스로 시작되며 컴포넌트 인터페이스 상속("Inherit_Component_Interface"), 컴포넌트 인터페이스 구현 클래스 상속("Inherit_Component_Class"), 확장 클래스의 교체 오퍼레이션 설계("Design_Replace_Operation_Of_Extend_Class")로 진행된다.

클래스 교체 커스터마이제이션 프로세스 내에 포함된 서브 프로세스들에 대해 다음과 같이 정의한다.

● 교체 클래스 설계

컴포넌트 내의 클래스를 다른 클래스로 교체하기 위해서는 교체 클래스의 인터페이스를 이용해 클래스를 정의한다. 교체하기 위한 클래스는 컴포넌트 외부에서 정의되며 교체 오퍼레이션의 입력 데이타로 활용된다. 정의 21와 같이 입력 게이트로 교체하기 위한 클래스의 인터페이스를 입력 받아 교체 클래스를 구현한다. 교체 클래스는 교체 인터페이스의 스펙을 따라 설계되며 설계된 교체 클래스는 교체 오퍼레이션의 입력

```
(* Implement the replace class *)
PROCESS Implement_Replace_Class[replace_interface]
  :exit :=

    type implement is IMPLEMENT_ASSOCIATION endtype
  : type replace_class is CLASS endtype

  : replace_interface ? internal_replace_class_interface : INTERFACE
  : implement.supes := internal_replace_class_interface
(* Implement the class with the internal_replace_class_interface *)
  : implement.sub := replace_class
  : replace_class ! outer_implement_class
  : exit
ENDPROC
```

정의 21 교체 클래스 설계 프로세스

파라메터로 입력되어 클래스를 교체한다. 교체 클래스를 설계하기 위한 전제 조건은 컴포넌트 내부에서 핫스팟(Hot Spot) 부분에 해당하는 교체 클래스는 반드시 인플리먼트 관계로 설계가 되도록 인터페이스를 제공해야 한다.

● 컴포넌트 인터페이스 상속

클래스 교체를 위한 컴포넌트 인터페이스 상속 프로세스는 행위 추가 커스터마이제이션 프로세스의 정의 13와 동일하다. 확장 인터페이스는 기존 컴포넌트의 인터페이스를 상속 받으며 기존 컴포넌트 인터페이스의 모든 시그니처를 포함한다.

● 컴포넌트 인터페이스 구현 클래스 상속

클래스 교체를 위한 컴포넌트 인터페이스 구현 클래스 상속 프로세스는 행위 추가 커스터마이제이션 프로세스의 정의 14과 동일하다. 확장 클래스는 기존 컴포넌트의 기능을 이용하기 위해 컴포넌트의 인터페이스 클래스를 상속 받으며 이 확장 클래스만 접근하면 기존 컴포넌트의 모든 기능을 사용할 수 있다.

● 확장 클래스의 오퍼레이션 설계(교체)

확장 인터페이스에 정의된 교체 오퍼레이션을 설계하기 위해 파라메터인 교체 클래스를 정의하여 추가 한다.

```
(* Design the replace operation of "extend_component_interface_impl" *)
PROCESS Design_Replace_Operation_Of_Extend_Class[class]
  (extend_component_interface : INTERFACE, extend_component_interface_impl : CLASS)
  :exit (extend_component_interface, extend_component_interface_impl) :=

    class ? replace_class : CLASS
  : type implement is IMPLEMENT_ASSOCIATION endtype
  : type replace_operation is OPERATION endtype

  : implement.super := extend_component_interface
  : implement.sub := extend_component_interface_impl

(* 교체 클래스를 입력 parameter로 하는 교체 오퍼레이션 정의 *)
  : replace_operation.signature.parameters.type := CLASS
  : replace_operation.signature.parameters.name := replace_class.name

(* 교체 operation을 확장 컴포넌트 인터페이스("extend_component_interface_impl") 에 추가 *)
  : extend_component_interface_impl.operation := replace_operation

  : exit (extend_component_interface, extend_component_interface_impl)
ENDPROC
```

정의 22 확장 클래스 정의 프로세스

교체 오퍼레이션의 정의를 통한 확장 클래스는 정의 22와 같이 정의한다. 교체 오퍼레이션은 "class" 게이트로 입력받은 교체 클래스를 파라메터로하여 오퍼레이션을 정의하며 이 오퍼레이션은 확장 클래스의 교체 오퍼

래이션(확장 인터페이스의 시그니처)으로 추가된다.

4.2.5 커스터마이제이션 기법의 조합

지금까지 제시된 3가지 컴포넌트 행위 커스터마이제이션 기법의 프로세스를 이용해 다양한 형태의 커스터마이제이션 기법을 조합할 수 있다.

표 5 컴포넌트 행위 커스터마이제이션 기법의 조합

| 순서 | 프로세스 | 추가 | 변경 | 교체 |
|---|---|---|---|---|
| 1 | 교체 클래스 설계 | | | O |
| 2 | 컴포넌트 인터페이스 상속 | O | | O |
| 3 | 컴포넌트 인터페이스 구현 클래스 상속(추가) | O | | O |
| 4 | 컴포넌트 인터페이스 구현 클래스 상속(변경) | | O | |
| 5 | 컴포넌트 인터페이스 구현 클래스의 오퍼레이션 재 정의 | | O | |
| 6 | 확장 클래스의 오퍼레이션 설계(추가) | O | | |
| 7 | 확장 클래스의 오퍼레이션 설계(교체) | | | O |
| 8 | 확장 클래스 등록 | | O | |

표 5는 각 기법에서 필요로 하는 모든 프로세스를 나타내며 각각의 기법에서 필요로 하는 프로세스들이 다른 기법과 중복하여 사용되고 있다. 2개 이상의 기법을 조합하여 새로운 행위 커스터마이제이션 기법을 구성할 수 있다.

$NT = \{\forall P \mid P_i I\ T_j\ ,\ 1 \le i \le n\ ,\ 1 \le j \le m\}$

NT : 새로운 커스터마이제이션 기법의 프로세스 조합,

T : 기존 커스터마이제이션 기법($T_1$: 추가, $T_2$: 변경, $T_3$: 교체),

P : 표 5의 커스터마이제이션 프로세스,

n : 표 5의 전체 프로세스 수(8),

m : 조합하려는 커스터마이제이션 기법의 수(2..3)

위와 같이 해당 기법에 속하는 프로세스들을 조합하여 새로운 커스터마이제이션 기법을 생성할 수 있으며 조합할 때 각각의 설계 기법에서 존재하는 절차는 모두 포함하여 새로운 기법을 생성한다. 각각의 프로세스들은 독립적으로 이용될 수 있도록 입력과 출력을 정의하고 있으며 프로세스의 조합만으로 쉽게 복합 커스터마이제이션 기법을 구성할 수 있다.

"추가/변경", "변경/교체", "추가/교체", "추가/변경/교체" 기법은 3가지 기본 커스터마이제이션 기법을 이용해 쉽게 조합될 수 있는 기법이다.

## 4.3 커스터마이제이션 프로세스 검증

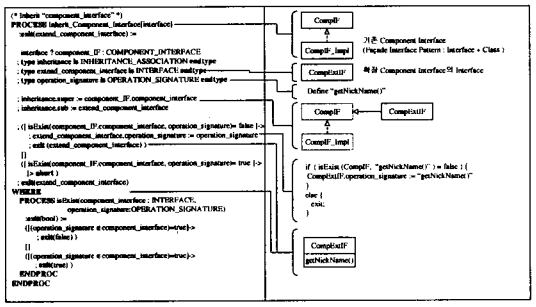
지금까지 커스터마이제이션 기법의 절차적인 모호성을 줄이고 설명으로 표현하기에는 부족한 부분들에 명확성을 주기 위해 정형적으로 표현하였다. 본 절에서는 이러한 목적으로 정의된 커스터마이제이션 프로세스의 타당성과 명확성을 검증하기 위해 실 사례의 커스터마이제이션 과정을 통해 검증하도록 한다. 프로세스 커스터마이제이션 기법 중에 행위 추가 커스터마이제이션 기법을 통해 검증한다.

4.3.1 행위 추가 커스터마이제이션

행위 추가 커스터마이제이션 프로세스는 정의 12에서 다음과 같이 정의하고 있다.

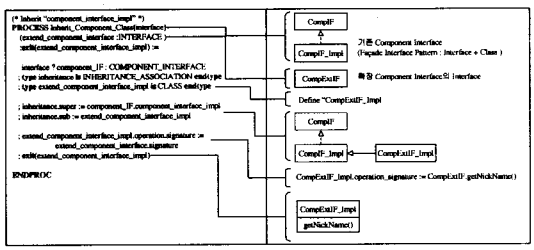
컴포넌트 인터페이스 상속R컴포넌트 인터페이스 구현 클래스 상속R확장 클래스의 오퍼래이션 설계

첫번째 단계로 컴포넌트 인터페이스 상속에 대한 검증은 검증 1과 같이 각각의 절차에 대해 구체적인 사례를 들어 어떻게 적용되는지 나타내고 있다.




검증 1 컴포넌트 인터페이스 상속 검증

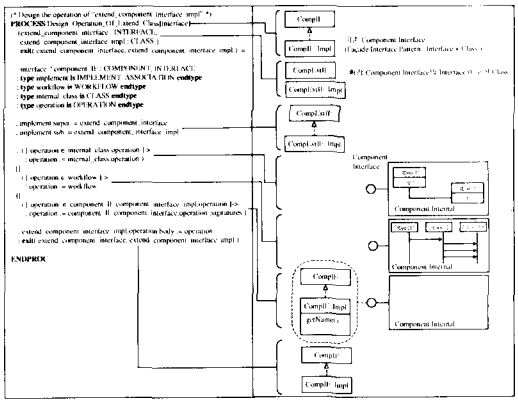
컴포넌트 인터페이스를 상속 받기 위해 기존 컴포넌트 인터페이스("CompIF", CompIF_Impl)를 입력 받으며 컴포넌트 인터페이스에 "getNickName()"이라는 새로운 인터페이스 함수를 추가하기 위해 새로운 인터페이스를 확장한다. 확장된 인터페이스에 새로운 함수를 추가하기 위해 기존 컴포넌트 인터페이스에 존재하는지




검증 2 컴포넌트 인터페이스 구현 클래스 상속 검증

비교하여 추가한다. 본 단계에서는 "CompExtIF"라는 확장 컴포넌트 인터페이스가 산출물로 나온다.

검증 2는 확장 인터페이스에 대해 기존 컴포넌트 인터페이스를 이용하여 확장 인터페이스의 구현 클래스를 설계한다. 본 단계에서는 "CompExtIF_Impl"이라는 확장 컴포넌트의 구현 클래스가 산출물로 나온다.



검증 3 확장 클래스의 오퍼레이션 설계 검증

검증 3은 설계된 확장 컴포넌트 인터페이스의 구현 클래스 내에 추가된 함수의 내부 기능을 설계하는 것으로 컴포넌트 내의 클래스 함수, 워크플로우, 그리고 기존 컴포넌트 인터페이스의 함수를 이용하여 추가 함수의 기능을 설계할 수 있음을 보이고 있다. 본 단계에서의 산출물은 확장 컴포넌트 인터페이스(확장 인터페이스, 확장 인터페이스의 구현 클래스)이다.

그림 15는 행위 추가 커스터마이제이션 프로세스를 통해 최종적으로 확장된 컴포넌트 인터페이스 설계 모델이다. 확장된 인터페이스 "CompExtIF"는 새로운 기능인 "getNickName()"이 추가되었으며 구현 클래스 "Comp ExtIF_Impl"은 컴포넌트 내의 기능을 사용하기 위해 기존

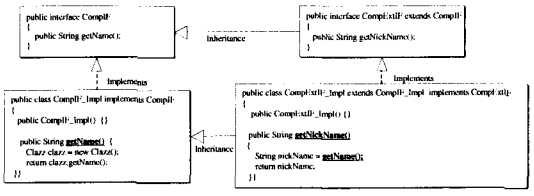


그림 15 기능 추가를 위한 확장 컴포넌트 인터페이스 모델

컴포넌트 인터페이스의 구현 클래스 "CompIF_Impl"을 상속 받았다. 이와 같은 기법을 이용하여 기존 컴포넌트는 전혀 변경을 가하지 않고 확장이 가능함을 보여주고 있다.

지금까지 행위 추가 커스터마이제이션 기법에 대한 정형화된 프로세스의 타당성과 명확성을 검증하기 위해 실 사례를 들어 절차적으로 어떻게 적용되는 지를 살펴보았으며 이러한 단위 프로세스는 독립적으로 입력과 출력을 가지고 있으므로 쉽게 조합되어 새로운 프로세스를 형성할 수 있는 장점을 갖는다.

## 5. 평 가

본 논문에서 제안한 커스터마제이션 기법이 다른 컴포넌트 개발 방법론이나 기법에서 제안한 기법들과 어떤 차이가 있는지 비교 평가한다.

표 6에서와 같이 개발 방법론과 다른 기법에서도 커스터마이제이션 설계를 위한 기법들을 제안하고 있지만 본 논문의 기법과는 다르게 상세 설계 기법을 제안하는 것이 아니라 부분적으로 제안하고 있거나 [18]과 [19] 논문의 연구에서처럼 복합 컴포넌트를 구성할 때의 커스터마이제이션 설계 기법을 제안하고 있다[19]. 연구는 컴포넌트 내의 행위를 추가하기 위한 인터페이스 추가의 기능을 제공하고 있으나 인터페이스의 변경은 컴포넌트 내의 행위를 변경하기 위한 기법이 아니라 복합 컴포넌트들 간의 연결을 위한 변경을 의미한다. 이와 같

표 6 커스터마이제이션 기법 특징 비교

| Factors | | This Method | Catalysis | Component -ware | [18] Method | [19] Method |
|---|---|---|---|---|---|---|
| 가변성 추출? | | N | P | P | N | N |
| 커스터마이제이션 설계? | | S(단일) | P | P | S(복합) | S(복합) |
| Black-Box Design? | | S(단일) | N | N | S(복합) | S(복합) |
| Component Coupling 축소? | | S | S | S | S | S |
| 컴포넌트 행위 커스터 마이제이션 | Interface 변경 | S | N | N | N | S(복합) |
| | Interface 추가 | S | N | N | N | S |
| | Component내의 Class 교체 | S | N | N | N | N |

[S] Support [N] Not Support [P] Partial Support [단일] 단일 컴포넌트 [복합] 복합 컴포넌트

이 본 논문에서 제안하는 기법과 같은 컴포넌트 내의 행위를 커스터마이즈하기 위한 연구가 부분적으로만 이루어 지고 있음을 알 수 있다.

본 논문에서는 가변성 추출이나 설계 기법은 다루고 있지 않으며, 설계된 가변성을 기반으로 기존 컴포넌트를 재설계할 때 커스터마이즈하기 위한 기법을 제안하고 있다.

## 6. 결 론

지금까지 본 논문에서는 컴포넌트의 행위 커스터마이제이션 기법을 제안했으며 컴포넌트의 인터페이스를 확장하는 기법으로 3가지 기법을 제안했다. 각각의 설계 절차를 정형적으로 정의하여 설계 상의 모호성을 줄일 수 있도록 하였으며 또한 정형적인 정의를 통해 다양한 형태의 커스터마이제이션 기법이 조합될 수 있는 기반을 제공한다. 이러한 커스터마이제이션 기법은 컴포넌트를 개발할 때 컴포넌트의 응집력을 높이고 결합력을 낮추게 하여 컴포넌트를 블랙 박스로 이용할 수 있도록 해준다. 또한 모든 도메인 요구 사항을 완전하게 충족할 수 있는 컴포넌트를 개발하는 것은 어려움이 많으므로 본 기법을 통해 다양한 요구 사항을 수용할 수 있도록 하여 컴포넌트의 일반성을 더욱 향상시킬 수 있다.

향후 연구과제는 컴포넌트 인터페이스의 변경을 확대할 수 있도록 컴포넌트 내부의 메시지 흐름을 변경할 수 있는 메시지 흐름 커스터마이제이션 기법과 컴포넌트를 통합하여 컴포넌트들 간의 메시지 흐름을 변경할 수 있는 기법을 연구한다.

## 참고문헌


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

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

[3] Felix Bachman, Technical Report, CMU/SEI-2000-TR-008, ESC-TR-2000-007.

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

[5] Jon Hopkins, "Component Primer", Communication of the ACM Vol. 43, No.10 , October 2000.

[6] Weiss D. M., "Commonality Analysis: A Systematic Process for Defining Families," Second International Workshop on Development and Evolution of Software Architectures for Product Families, February 1998.

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

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

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

[10] 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.

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

[12] Sun Microsystems Inc., "Enterprise JavaBeans Specifications", http://www.javasoft.com.

[13] J. Rumbaugh et. al., Object-Oriented Modeling and Design, Prentice Hall, 1991.

[14] Sterling Software Inc., "The CBD Standard Version 2.1", Sterling Software, July 1998.

[15] Short K., "Component Based Development and Object Modeling", Sterling Software, Technical Handbook Version 1.0, February 1997.

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

[17] Klaus Bergner, Andreas Rausch, Marc Sihling, "Componentware-The Big Picture", Institut fur Informatik,Technische Universitat Munchen, http://www.sei.cmu.edu/cbs/icse98/papers/p6.html.

[18] I. Sora, P. Verbaeten, and Y. Berbers, "Using Component Composition for Self-Customizable Systems", Workshop on Component-Based Software Engineering, ECBS 2002, April 8-11, 2002, Lund, Sweden.

[19] R. P. e Silva, et al., "Component Interface Pattern", Procs. Pattern Languages of Program, 1999.

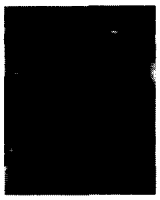
[20] J.M. Spivey, "The Z Notation : A Reference Manual", Programming Research Group, University Of Oxford, Second Edition, 1998.

[21] Graeme Paul Smith, "An Object-Oriented Approach To Formal Specification", The Department of Computer Science University of Queensland, 1992

[22] Jim Davies and Jim Woodcock, http://softeng.comlab.ox.ac.uk/usingz/, Using Z

[23] Department of Computer Science, University of Manchester, Technical Report Series UMCS-01-06-1, "A Proposal for a LOTOS-Based Semantics for UML".

[24] Kenneth J. Turner, International Workshop on Comparing Systems Specification Techniques, pages 83-98, University of Nantes, France, March 1998, "The Invoicing Case Study In (E-)Lotos".



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

정 승 재
2000년 숭실대학교 컴퓨터학과 학사 2002년 숭실대학교 컴퓨터공학 석사 2002년~현재 이네트 기술 연구소 선임연구원. 관심분야는 CBD(Component Based Development), 컴포넌트 개발 방법론

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