데이터 타입 무결성 컴포넌트 모델 : 외부화된 데이터 가변성 처리 기법

(Data Type-Tolerant Component Model: A Method to Process Variability of Externalized Data)

임 윤 선 [†] 김 명 [††] 정 승 남 [†††] 정 안 모 [††††]
(Yoonsun Lim) (Myung Kim) (Seongnam Jeong) (Anmo Jeong)

요 약 다계층 구조로 설계된 현대의 분산 애플리케이션 아키텍처에서 비즈니스 엔티티는 모든 서비스 로직 컴포넌트들을 관통하는 일종의 횡단관심사(Crosscutting Concerns)이다. 그러므로 비즈니스 엔티티가 변화하면 이와 관련된 서비스 컴포넌트들은 비록 애플리케이션 프레임워크의 공통적인 기능을 구현한 서비스 로직 컴포넌트라 할지라도 새로운 비즈니스 엔티티를 다룰 수 있도록 수정되어야 한다. 본 논문에서는 비즈니스 엔티티, 즉 외부화(externalized)된 데이터에 대한 가변성(variability)을 처리하는 DTT 컴포넌트 모델(Data Type-Tolerant Component Model)을 제시한다. DTT 컴포넌트 모델은 SCDT(Self-Contained Data Type)와 가변점(Variation Point) 인터페이스를 통해 프로덕트 라인의 데이터 가변성을 구현 수준에서 구체적으로 표현하고, 서비스 컴포넌트 코드 수정대신 비즈니스 엔티티와 SCDT간 타입 변환을 지원하는 데이터 타입 컨버터를 도입함으로써 애플리케이션 엔지니어링 효율을 향상시킨다. 서비스 컴포넌트가 외부화된 비즈니스 엔티티를 직접 다루지 않고 SCDT만을 다루게 함으로써 데이터와 함수의 커플링을 다시 컴포넌트 수준에서 로컬화했다는 점이 DTT 컴포넌트 모델의 의의라 할 수 있다.

키워드 : 소프트웨어 프로덕트 라인, 가변성, DTT 컴포넌트 모델, 엔터프라이즈 애플리케이션

Abstract Business entities with which most service components interact are kind of cross-cutting concerns in a multi-layered distributed application architecture. When business entities are modified, service components related to them should also be modified, even though they implement common functions of the application framework. This paper proposes what we call DTT (Data Type-Tolerant) component model to process the variability of business entities, or externalized data, which feature modern application architectures. The DTT component model expresses the data variability of product lines at the implementation level by means of SCDTs (Self-Contained Data Types) and variation point interfaces. The model improves the efficiency of application engineering through data type converters which support type conversion between SCDTs and business entities of particular applications. The value of this model lies in that data and functions are coupled locally in each component again by allowing service components to deal with SCDTs only instead of externalized business entities.

Key words : Software Product Line, Variability, DTT Component Model, Enterprise Application

· 이 논문은 2007년도 정부재원(교육인적자원부 학술연구조성사업비)으로 한국학술진흥재단의 지원을 받아 연구되었음(KRF-2007-313-D00654)

† 학생회원 : 이화여자대학교 컴퓨터공학과
lys96@ewhain.net
†† 종신회원 : 이화여자대학교 컴퓨터공학과 교수
mkim@ewha.ac.kr
††† 정 회 원 : 리버넥스 이사
hellojsn@libernex.com
†††† 정 회 원 : 리버넥스 대표
amjeong@libernex.com

논문접수 : 2008년 12월 19일
심사완료 : 2009년 3월 14일

Copyright©2009 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회논문지: 소프트웨어 및 응용 제36권 제5호(2009.5)

1. 서 론

소프트웨어 프로덕트 라인 공학(PLE: Product Line Engineering)에서 가변성(variability) 처리 효율은 애플리케이션 개발 생산성과 직결되는 중요한 문제이다. 가변성을 다루는 기존의 PLE 연구들[1-4]은 데이터와 함수가 캡슐화된 객체 지향 기술을 기반으로 하였기 때문에 아키텍처상의 기능적 가변성이나 로컬화된 데이터 가변성을 처리하는 수준에 머물러 있다. 그러나 단지 기능적 컴포넌트들 선택하거나 추가하는 방식만으로는 현대의 대규모 엔터프라이즈 애플리케이션 아키텍처에서 특징적으로 나타나는 비즈니스 엔티티, 즉 외부화(externalized)된 데이터에 대한 가변성을 효율적으로 처리하기 어렵다. 인사관리 애플리케이션에서 '사원'이라는 비즈니스 엔티티를 구성하는 속성이 기업마다 조금씩 다른 경우에서 보듯이, 일반적으로 엔터프라이즈 애플리케이션에서는 기능 가변성보다 데이터 가변성이 높다.

하드웨어와 네트워크 성능이 비약적으로 발전하고 위치 투명성을 지원하는 컴포넌트 기술이 등장하면서 현대의 엔터프라이즈 애플리케이션들은 다계층 구조로 설계되고 분산환경에서 운영된다. 엔터프라이즈 애플리케이션 구현 기술로 확고하게 자리잡은 자바나 닷넷의 참조 모델에서는 비즈니스 로직 계층과 데이터 계층을 분리하고 있다[5,6]. 그러나 이렇게 데이터 계층의 비즈니스 엔티티를 비즈니스 로직 계층의 서비스 컴포넌트로부터 분리시킴으로써 대규모 애플리케이션 개발을 용이하게 하고 영속성(persistence) 메커니즘 등의 환경변화에 유연히 대처할 수 있게 된 반면, 객체지향 기술에서 로컬화되었던 데이터는 다시 외부화되었다. 따라서 프로덕트 라인이 엔터프라이즈 애플리케이션 개발에서 효율적으로 재사용되려면, 서비스 컴포넌트에서 비즈니스 엔티티 가변성을 효율적으로 처리하는 연구가 필요하다.

기존 PLE 분야의 연구들[1-4]은 도메인의 분석, 설계 방법과 산출물들을 명세서에 표현하는 방법들에 한정되어 있고, 그것을 효과적으로 구현하여 코드화하는 방법은 거의 다루고 있지 않다. 도메인 엔지니어링의 분석과 설계에서 사용되는 UML 등의 모델링 언어는 너무 추상성이 높아 엔터프라이즈 애플리케이션 구현 기술인 닷넷이나 자바기술의 코드 생성에 필요한 상세함을 표현할 수 없기 때문에 설계와 구현간에 필연적으로 괴리가 발생한다[7]. 프로덕트 라인은 결국 구현기술에 의해 구축되어 재사용되어야 의미를 가지므로 그 명세가 구현 기술과 잘 접목되어야 할 뿐만 아니라 구축에 필요한 모든 요소와 과정을 구체적으로 다루어야 한다.

본 논문에서는 외부화된 데이터인 비즈니스 엔티티와 서비스 컴포넌트간의 의존 관계를 완화하는 방법을 제시함으로써 현대의 엔터프라이즈 애플리케이션 개발에서 효율적으로 재사용될 수 있는 프로덕트 라인을 구축할 수 있게 하는 방법을 제시한다. 프로덕트 라인을 구축하고 사용함에 있어 고객사마다 기능적 가변성 못지않게 비즈니스 엔티티의 가변성을 효율적으로 해결하는 것이 절실한 문제이기 때문이다. 또한 비즈니스 엔티티 가변성의 표현이 닷넷이나 자바 등의 컴포넌트 기술과 정확히 접목되게 하고, 도메인 컴포넌트 구현과 가변치(variant) 구현 단계를 잘 정립된 패턴으로 정리하여 필요한 코드 생성을 개발 도구에 의해 자동화할 수 있게 하였다.

본 논문은 다음과 같이 구성된다. 2장에서 관련연구들을 살펴보고, 3장에서는 본 논문에서 제안한 DTT 컴포넌트 모델의 구성 요소 및 상호작용에 대해 설명한다. 4장에서는 DTT 컴포넌트 모델의 구현 예를 실제 코드를 통해 살펴보고, 5장에서 DTT 컴포넌트 모델 구성 요소들의 코드를 자동 생성하는 개발 도구를 소개한다. 6장에서는 연구 결과에 대해 고찰해 보고 7장에서 결론을 맺는다.

2. 관련 연구

최근 대규모 분산 엔터프라이즈 애플리케이션 아키텍처에서 채택되고 있는 비즈니스 엔티티는 이들이 관통하는 다양한 서비스 컴포넌트 관점에서 일종의 횡단관심사(Crosscutting Concerns)이다. 즉 특정 애플리케이션에서 비즈니스 엔티티가 변화하면 애플리케이션 프레임워크를 구성하는 공통 서비스 컴포넌트가 수정되어야 한다. 기존 연구들[8,9]에서는 직접적으로 변화하는 객체를 수용하는 가변성 문제만 다룰 뿐, 비즈니스 엔티티와 같이 직접적인 객체의 변화와 더불어 이에 따른 다른 객체의 이차적인 변화를 수용하는 가변성을 다루고 있지 않다. 이에 본 장에서는 엔터프라이즈 분산 애플리케이션 모델의 특징 및 기존의 가변성 처리 연구에 대해 살펴본다.

2.1 엔터프라이즈 애플리케이션 참조 모델

현대의 대규모 엔터프라이즈 분산 애플리케이션들은 대부분 자바(Java Enterprise Edition), COM+, 닷넷 등 컴포넌트 기술 기반으로 개발된다. 자바의 엔터프라이즈 애플리케이션 참조 모델에서는 비즈니스 주체가 유지관리해야 하는 주요한 데이터 엔티티를 엔티티 빈으로, 엔티티에 대해 비즈니스 로직을 처리하는 서비스 컴포넌트를 세션 빈으로 표현하고 있다[5]. 마이크로소프트 Patterns & Practices의 엔터프라이즈 애플리케이션 참조 모델에서는 비즈니스 엔티티를 다음과 같이 설명하고 있다[6].

비즈니스 엔티티 컴포넌트: 대부분의 애플리케이션들

은 애플리케이션을 구성하는 컴포넌트들이 서로 주고받는 데이터들을 필요로 한다. 예를 들어 소매 애플리케이션에서는 사용자에게 상품 리스트를 보여주기 위해서 데이터 액세스 컴포넌트로부터 사용자 인터페이스 컴포넌트들로 '상품의 리스트'가 전달되어야 한다. 이러한 데이터들은 '상품'이나 '주문'과 같은 실제 비즈니스 엔티티들을 표현하는데 사용된다. 애플리케이션 내부에서 사용되는 비즈니스 엔티티들은 객체지향 기술에서는 커스텀 클래스들로 구현된다.

도메인의 공통성(commonality)를 지원하는 공통 서비스 컴포넌트 세트를 구축하기 위해서는 외부화된 데이터, 즉 비즈니스 엔티티가 바뀌어도 서비스 컴포넌트를 수정 없이 재사용할 수 있게 해주는 새로운 기법이 필요하다.

2.2 아키텍처 기반 가변성 처리

가변성을 모델링하는 방법으로 패턴을 사용하는 연구[10], UML 확장 기법을 이용하는 연구[11]가 있다. 또 정보 은닉, 상속, 가변점을 사용하는 연구[2]도 있다. 이들은 모두 객체지향의 특성을 기반으로 한 연구들로서 아키텍처 수준에서 컴포넌트들을 선택 혹은 대체하는 형태의 기능적 가변성이나 로컬화된 데이터의 가변성만 다루고 있다. 그러나 이들은 아키텍처 수준에서 가변점(Variation Point)에 대응하는 직접적인 가변치들을 모델링의 요소로 나타낼 뿐 애플리케이션 아키텍처에 새로 도입되는 가변치들에 의해서 영향 받는 연관된 요소에서의 이차적인 가변성을 표현하지 못한다. 현대의 엔터프라이즈 애플리케이션에서 재사용될 수 있는 프로덕트 라인을 구축하기 위해서는 기업마다 변화하는 비즈니스 엔티티에 대한 가변성 처리와 더불어 외부화된 데이터의 변화가 서비스 컴포넌트들에 미치는 이차적인 가변성을 효과적으로 처리할 수 있는 연구가 필요하다.

2.3 Component & Connector 기반 가변성 처리

재사용할 컴포넌트들을 통합하여 컴포넌트 인프라스트럭처 혹은 수직적 프레임워크(Vertical Framework)를 만드는데 있어 커넥터의 사용을 강조하는 연구로는 [12,4,13]이 있다. [12]은 컴포넌트들을 커넥터들로 재구성하여 여러 형태의 컴포넌트 인프라스트럭처를 만들 수 있음을 강조하지만 참조 모델이나 커넥터의 구성요소 및 타입에 대한 명확한 언급이 없다. [4]는 한발 더 나아가 커넥터를 데이터 변환, 인터페이스 어댑터, 기능성 변환, 워크플로우 핸들러 타입으로 세분화하고 있지만 구현 기술과의 정합성을 구체적으로 기술하지 않고 있다. [13]은 커넥터를 구현 수준에서 표현하는 방법과 구현 절차 및 런타임 프레임워크를 상세히 표현하였다. 그러나 커넥터는 컴포넌트 통합시에 Required 인터페이스와 Provided 인터페이스 사이에 발생하는 가변성 즉 매개변수의 구문적, 의미론적 불일치를 중재하거나 기능을 보완하는데 사용된다. 따라서 커넥터는 비즈니스 엔티티의 변화에 따른 서비스 컴포넌트의 가변성을 처리하는 데는 적합하지 않다.

3. DTT(Data Type-Tolerant) 컴포넌트 모델

본 장에서는 소프트웨어 프로덕트 라인에서 도메인의 공통 서비스 컴포넌트가 특정 애플리케이션의 비즈니스 엔티티에 종속되지 않는 DTT(Data Type-Tolerant) 컴포넌트 모델을 제시한다. DTT 컴포넌트 모델은 자체 데이터 타입(SCDT: Self-Contained Data Type)을 서비스 컴포넌트에 내장하는 방법과, 이와 같이 개발한 서비스 컴포넌트를 애플리케이션 개발에 재사용할 때, 애플리케이션의 비즈니스 엔티티를 서비스 컴포넌트의 SCDT로 변환하는 데이터 타입 컨버터를 구현하는 방법을 제시한다. 또한 위와 같은 방식으로 개발된 애플리케이션의 실행시 프레임워크가 의존성 주입(Dependency Injection) 정보 파일을 참조하여 구성 요소들의 객체를 생성하고, 데이터 타입 컨버터 객체를 서비스 컴포넌트의 SCDT에 주입함으로써 재사용되는 공통 서비스 컴포넌트들이 새롭게 정의된 비즈니스 엔티티들을 에러 없이 다룰 수 있게 하는 방법을 제시한다.

3.1 DTT 컴포넌트 구성 요소

그림 1은 DTT 컴포넌트 모델 구조이다. DTT 컴포넌트 모델은 (1) 도메인 엔지니어링에서 공통성을 식별하여 미리 개발한 서비스 컴포넌트, (2) 특정 애플리케이션의 비즈니스 엔티티, (3) 실행시 비즈니스 엔티티를 서비스 컴포넌트의 SCDT로 변환하는 데이터 타입 컨버터 컴포넌트, (4) 의존성 주입 정보 파일을 참조하여 각 구성 요소의 객체를 생성하고 데이터 타입 컨버터 객체를 서비스 컴포넌트의 SCDT에 주입하는 런타임 프레임워크로 구성되어 있다.

루트 데이터 타입 / 비즈니스 엔티티

루트데이터 타입은 도메인 공통 서비스 컴포넌트 개발자와 애플리케이션 개발자가 공통으로 사용하는, 모든 비즈니스 엔티티 타입이 상속해야 하는 기본 데이터 타입이다. 공통 서비스 컴포넌트는 실제 연동하게 될 애플리케이션의 비즈니스 엔티티가 존재하지 않는 상태에서 미리 개발되는 것이므로, 구성 메소드의 매개 변수나 반환 값을 루트 데이터 타입으로 선언함으로써 차후 애플리케이션의 비즈니스 엔티티를 수용할 수 있게 된다. 이 루트 데이터 타입은 명칭만 가지는 형식으로 정의되고, 모든 개발자들에게 기본 클래스 라이브러리로 제공된다.

도메인 공통 서비스 컴포넌트 구조

공통 서비스 컴포넌트는 컴포넌트 기능을 구현한 서비스 객체 외에, 차후 연동하게 될 애플리케이션의 비즈

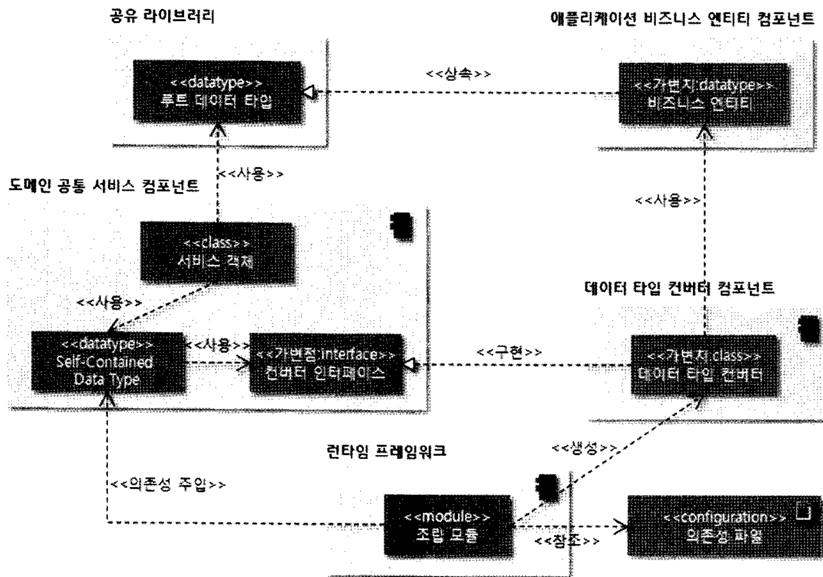

그림 1 DTT 컴포넌트 모델

니스 엔티티 대신 서비스 객체 코딩에 사용하기 위해 정의된 SCDT와, 특정 비즈니스 엔티티 타입을 컴포넌트의 SCDT로 변환하기 위해 구현할 데이터 타입 컨버터 객체의 모태인 데이터 타입 컨버터 가변점 인터페이스를 추가로 내장하는 구조를 갖는다.

데이터 타입 컨버터(Data Type Converter) 컴포넌트 구조

데이터 타입 컨버터 컴포넌트는 애플리케이션 개발시 생성되어, 실행시 비즈니스 엔티티를 서비스 컴포넌트의 SCDT로 타입을 변환해준다. 데이터 타입 컨버터 객체는 SCDT의 각 속성에 대하여 Get, Set 메소드를 구현하며, 서비스 객체가 SCDT의 속성에 접근할 때마다 해당 메소드가 호출된다. 각 메소드에서는 SCDT 속성에 대응되는 비즈니스 엔티티의 속성값을 읽어 SCDT 속성값으로 변환하는 로직을 실행한다.

런타임 프레임워크

애플리케이션 개발시 서비스 컴포넌트, 비즈니스 엔티티, 데이터 타입 컨버터 컴포넌트에 대해 상호 의존 관계를 나타낸 파일을 생성하고, 이들을 그림 1의 런타임 프레임워크에 배치하여 실행하면, 런타임 프레임워크는 상기 의존성 정보 파일에 따라 컴포넌트들의 객체를 생성하고 데이터 타입 컨버터 객체의 레퍼런스를 서비스 컴포넌트의 SCDT에 주입해준다. 이를 통해 비즈니스 엔티티의 속성들은 SCDT 속성들로 변환되며, 이 상태에서 서비스 메소드는 SCDT를 사용하여 필요한 작업을 수행한다.

3.2 DTT 컴포넌트 모델 구성 요소의 상호 작용

그림 2는 본 논문에서 제안한 구조에 따라 개발된 애플리케이션 실행시 각 구성 요소가 클라이언트의 서비스 호출을 처리하는 상호작용을 나타낸 순차도이다. 그림 1의 런타임 프레임워크에서 객체 생성과 객체간 의존성 주입을 실행하고 클라이언트의 서비스 호출을 처리하는 절차를 그림 2의 각 단계별로 설명하면 다음과 같다.

1. 사용자 인터페이스 등 외부의 클라이언트가 런타임 프레임워크에 배치된 서비스 객체를 요구한다.
2. 런타임 프레임워크가 의존성 주입 정보 파일을 참조하여 해당 서비스 객체와 그에 연관된 데이터 타입 컨버터 객체를 생성한다.
3. 생성된 데이터 타입 컨버터 객체의 레퍼런스를 각 컨버터 객체에 대응하는 SCDT의 의존성 주입용 속성에 주입한 후, 서비스를 요청한 클라이언트에 해당 서비스 객체를 반환한다.
4. 클라이언트가 애플리케이션의 비즈니스 엔티티의 객체를 생성한다.
5. 클라이언트가 넘겨받은 컴포넌트 객체의 서비스 메소드를 호출하면서 비즈니스 엔티티의 객체 레퍼런스를 매개변수로 넘겨주면, 호출된 서비스 메소드가 클라이언트로부터 매개변수로 받은 비즈니스 엔티티 객체를 SCDT 객체로 변환하는 명령문을 수행한다.
6. 호출된 서비스 메소드가 자신의 서비스 수행을 위해 SCDT 객체의 특정 속성을 액세스하면, 액세스된 속성의 액세서가 데이터 타입 컨버터 객체의 해당 속성과 관련된 타입 변환 메소드를 호출하면서 비즈니스

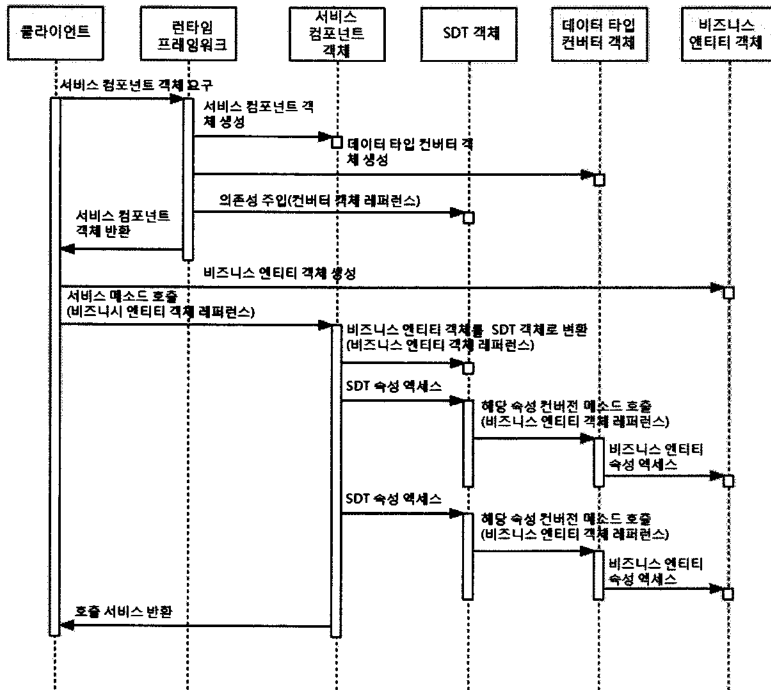

그림 2 DTT 컴포넌트 모델 구성 요소 상호작용 순차도

엔티티 객체의 레퍼런스를 넘겨주고, 호출된 타입 변환 메소드는 비즈니스 엔티티 객체의 속성을 그에 대응하는 SCDT 속성으로 변환하여 반환한다.

7. 클라이언트가 호출한 서비스 메소드는 필요한 만큼 SCDT 객체의 속성을 액세스하면서 자신의 서비스 수행을 완료하고 클라이언트에 서비스 수행 결과를 반환한다.

4. DTT 컴포넌트 모델의 구현 예

본 장에서는 가상의 호텔관리 도메인에서 애플리케이션마다 바뀔 수 있는 고객 비즈니스 엔티티의 가변성을 DTT 컴포넌트 모델을 이용해 처리하는 C# 프로그램 구현 예를 설명한다. 먼저 본 장에서 다루게 될 호텔 관리 프레임워크의 공통 요구사항과 가변 요구사항을 정의한 후, DTT모델에 따라 공통 서비스 컴포넌트와 이 컴포넌트의 SCDT, 데이터 타입 컨버터를 사용하여 앞서 정의된 요구사항을 구현한 예를 설명한다. 구현 예는 DTT 컴포넌트 모델의 개념을 이해하는 데 직접적으로 관련되는 최소한의 코드만을 다룬다.

공통 요구사항

- 고객 비즈니스 엔티티 객체와 메시지를 매개변수로 받아 고객의 이메일 주소로 메시지를 전송하는 기능을 제공하여야 한다.
- 고객 비즈니스 엔티티 객체와 새로운 이메일 주소를 매개변수로 받아 고객의 이메일 주소를 갱신하는 기능을 제공하여야 한다.
- 고객 비즈니스 엔티티는 반드시 고객의 고유 번호와 이메일 주소 필드를 포함하여야 한다.

가변 요구사항

- 고객 비즈니스 엔티티가 고객의 고유 번호와 이메일 주소 필드는 반드시 포함해야 하지만, 그 타입과 필드명은 애플리케이션마다 달라질 수 있다. 고유 번호와 이메일 주소를 제외한 나머지 필드들은 애플리케이션마다 자유롭게 변경될 수 있어야 한다.
- 애플리케이션에 따라 고객 비즈니스 엔티티 필드들이 바뀌어도 서비스 컴포넌트들은 코드 수정 없이 변경된 비즈니스 엔티티 객체를 처리할 수 있어야 한다.

4.1 비즈니스 엔티티

그림 3은 호텔 애플리케이션에서 사용하는 비즈니스 엔티티인 HotelCustomer 클래스의 코드를 보여준다. 이 클래스에 부가된 커스텀 애트리뷰트 "[ClassType(ClassType.EntityClass)]"는 HotelCustomer가 애플리케이션의 가변치인 비즈니스 엔티티임을 표현한다. 이 코드에는 비즈니스 엔티티 HotelCustomer가 루트 데이터 타입인 Entity를 상속하고 있음이 나타나 있다. 이 비즈니스 엔티티에는 CustId 등 6개의 속성이 선언되어 있다.

```
[ClassType(ClassType.EntityClass)]
public class HotelCustomer : Entity
{
    public string CustId;
    public string Name;
    public string PostCode;
    public string Address;
    public string Email;
    public DateTime LastVisit;
}
```

그림 3 고객 비즈니스 엔티티 코드 예

4.2 서비스 컴포넌트

그림 4는 애플리케이션에서 재사용하는 도메인 공통 서비스 컴포넌트 코드 중 서비스 객체 CustomerMgt 부분을 보여준다. 이 서비스 객체는 애플리케이션의 비즈니스 엔티티와 메시지를 매개변수로 받아 고객의 이메일 주소로 메시지를 전송하는 NotifyCustomer 메소드와 비즈니스 엔티티 객체와 새로운 이메일 주소를 매개변수로 받아 고객의 이메일 주소를 갱신하는 UpdateEmailAddress 메소드를 포함한다.

```
public class CustomerMgt
{
    [DependencyInjection]
    public IRq RIRq;

    [ServiceMethod]
    [RequiredMethod("IRq", "RqNotifyCustomer", 1)]
    [SDTUsed("TopCustomer.MyCustomer")]
    [MethodDescription("고객에게 이메일로 정보를 발송하는 서비스")]
    [ParameterDescription("customer", "고객 개인 정보")]
    [ParameterDescription("message", "고객에게 메일로 알려줄 메시지")]
    public void NotifyCustomer(Entity customer, string message)
    {
        MyCustomer cust = (MyCustomer)customer;
        string mailaddr = cust.CustomerEMail;

        // Send `message` to `mailaddr`

    }

    [ServiceMethod]
    [RequiredMethod("IRq", "RqUpdateEmailAddress", 1)]
    [SDTUsed("TopCustomer.MyCustomer")]
    [MethodDescription("고객의 이메일 주소 변경 서비스")]
    [ParameterDescription("customer", "고객의 개인 정보")]
    [ParameterDescription("newEmailAddr", "변경할 이메일 주소")]
    public bool UpdateEmailAddress(Entity customer, string newEmailAddr)
    {
        MyCustomer cust = (MyCustomer)customer;
        cust.CustomerEMail = newEmailAddr;

        // Update customer information in datastore with cust.CustomerNumber

    }
}
```

그림 4 서비스 컴포넌트 코드 예

여기서 서비스 객체의 메소드는 애플리케이션의 비즈니스 엔티티 HotelCustomer보다 먼저 구현되므로 매개변수 타입으로 HotelCustomer를 사용할 수 없고, 대신 모든 비즈니스 엔티티가 상속하는 루트 데이터 타입인 Entity 타입 매개변수를 사용한다.

이 두 메소드는 모두 매개변수 customer로 받은 비즈니스 엔티티 객체를 SCDT인 MyCustomer로 변환하는 코드를 첫 명령문으로 포함하고 있는데, 이 명령문을 작성하기 전에 SCDT와 데이터 타입 컨버터 인터페이스 정의가 선행되어야 한다.

그림 5는 서비스 컴포넌트 코드 중 SCDT 클래스 MyCustomer 부분을 보여준다. 코드 상단에 나타나는

```
[ClassType(ClassType.SelfDefinedTypeClass)]
[Description("고객 정보")]
public class MyCustomer
{
    [Description("고객의 고유 번호")]
    public string CustomerNumber
    {
        get { return TypeConverter.GetCustomerNumber(entity); }
        set { TypeConverter.SetCustomerNumber(entity, value); }
    }

    [Description("고객의 이메일주소")]
    public string CustomerEMail
    {
        get { return TypeConverter.GetCustomerEMail(entity); }
        set { TypeConverter.SetCustomerEMail(entity, value); }
    }

    [DependencyInjection()]
    public static IMyCustomerConverter TypeConverter;
    private Entity entity;

    public MyCustomer(Entity entity)
    {
        this.entity = entity;
    }

    public static explicit operator MyCustomer(Entity entity)
    {
        return new MyCustomer(entity);
    }
}
```

그림 5 SCDT 코드 예

커스텀 애트리뷰트 [ClassType(ClassType.SelfDefinedTypeClass)]은 MyCustomer가 이 서비스 컴포넌트의 SCDT임을 표현한다. SCDT에는 우선 이 타입과 연동할 애플리케이션의 비즈니스 엔티티 객체의 레퍼런스를 유지하기 위한 필드 변수 bizEntity와 실행 시 런타임 프레임워크로부터 데이터 타입 컨버터 객체의 레퍼런스를 유지할 필드 변수 typeConverter가 선언되어 있다. 또 런타임 프레임워크로부터 데이터 타입 컨버터 객체의 레퍼런스를 주입받는 통로로서 속성 TypeConverter가 정의되어 있다. 이어서 이 SCDT 객체가 생성될 때 루트 데이터 타입을 매개변수로 갖도록 규정하는 생성자와 차후 연동할 비즈니스 엔티티 객체를 SCDT인 MyCustomer 객체로 변환하는 컨버전 오퍼레이터가 정의되어 있다. 또 서비스 객체의 메소드들에서 사용하는 SCDT의 속성들이 정의되어 있는데, 본 예에서 연동하게 될 애플리케이션 비즈니스 엔티티 HotelCustomer의 속성들과 개수, 이름, 타입 등이 상이함을 알 수 있다. SCDT의 각 속성의 Get/Set 액세서에는 주입된 데이터 타입 컨버터 객체의 해당 메소드를 호출하는 명령문이 있다.

그림 6은 서비스 컴포넌트의 코드 중 데이터 타입 컨버터 인터페이스 IMyCustomerConverter 부분을 보여준다. 코드 상단에 나타나는 커스텀 애트리뷰트 [InterfaceType(InterfaceType. EntityConverterInterface)]은 IMyCustomerConverter가 가변점이라는 것을 표현한다. 컨버터 인터페이스는 서비스 객체의 서비스 메소드에서 사용하는 SCDT의 각 속성을 그에 대응하는 비즈니스 엔티티 속성으로 변환하는 Get/Set 메소드들로 구성된다.

```
[InterfaceType(InterfaceType.EntityConverterInterface)]
public interface IMyCustomerConverter
{
    string GetCustomerNumber(Entity entity);

    void SetCustomerNumber(Entity entity, string newValue);

    string GetCustomerEMail(Entity entity);

    void SetCustomerEMail(Entity entity, string newValue);
}
```

그림 6 DTT 인터페이스 코드 예

4.3 데이터 타입 컨버터 컴포넌트

그림 7은 비즈니스 엔티티 HotelCustomer의 속성들을 SCDT MyCustomer의 속성들로 변환하는데이터 타입 컨버터 MyCustomerConverter의 코드를 보여준다. 코드 상단에 나타나는 커스텀 애트리뷰트 [ClassType(ClassType.EntityConverterClass)]는 MyCustomer-Converter가 애플리케이션의 가변치임을 표현한다. 데이터 타입 컨버터는 서비스 컴포넌트에 정의되어 있는 데이터 타입 컨버터 인터페이스 IMyCustomerConverter를 상속받아 메소드들을 구현한다. 이 메소드들을 구현할 때 애플리케이션 개발자는 해당 메소드가 다루는 SCDT 속성에 대응하는 비즈니스 엔티티의 속성을 식별하여 매핑 코드를 작성한다. 이 데이터 타입 컨버터 샘플 코드는 비즈니스 로직을 구현한 서비스 컴포넌트가 SCDT 필드 CustomerNumber, CustomerEmail을 액세스할 때마다 고객 비즈니스 엔티티의 필드 CustId와 Email값을 읽거나 변경할 수 있도록 해준다.

```
[ClassType(ClassType.EntityConverterClass)]
[TargetEntity("HotelCustomer.dll", "HotelCustomer.HotelCustomer")]
public class MyCustomerConverter : TopCustomer.IMyCustomerConverter
{
    [TargetProperty("CustId")]
    public string GetCustomerNumber(Entity entity)
    {
        string rVal = 0;
        HotelCustomer.HotelCustomer obj = (HotelCustomer.HotelCustomer)entity;

        rVal = obj.CustId;

        return rVal;
    }

    [TargetProperty("CustId")]
    public void SetCustomerNumber(Entity entity, string newValue)
    {
        HotelCustomer.HotelCustomer obj = (HotelCustomer.HotelCustomer)entity;

        obj.CustId = newValue;
    }

    [TargetProperty("Email")]
    public string GetCustomerEMail(Entity entity)
    {
        string rVal = "";
        HotelCustomer.HotelCustomer obj = (HotelCustomer.HotelCustomer)entity;

        rVal = obj.Email;

        return rVal;
    }

    [TargetProperty("Email")]
    public void SetCustomerEMail(Entity entity, string newValue)
    {
        HotelCustomer.HotelCustomer obj = (HotelCustomer.HotelCustomer)entity;

        obj.Email = newValue;
    }
}
```

그림 7 데이터 타입 컨버터 구현 코드 예

4.2절에서 구현한 공통 서비스 컴포넌트를 포함하는 호텔 관리 프레임워크를 이용하여 새로운 호텔의 애플리케이션을 개발할 때, 호텔 측에서 정수 타입의 Customer_ID 필드를 고객 고유번호로 사용한다고 가정하자. 이 경우, 데이터 타입 컨버터를 그림 8과 같이 수정하면 서비스 컴포넌트는 코드 변경 없이 동작한다. 일반적으로 비즈니스 엔티티는 비즈니스 로직 계층과 프리젠테이션 계층의 여러 컴포넌트들을 관통하지만, DTT 모델에 따라 개발된 애플리케이션 프레임워크에서는 데이터 컨버터만 수정함으로써 데이터 가변성을 처리한다.

```
[ClassType(ClassType.EntityConverterClass)]
[TargetEntity("HotelCustomer.dll", "HotelCustomer.HotelCustomer")]
public class MyCustomerConverter : TopCustomer.IMyCustomerConverter
{
    [TargetProperty("Customer_ID")]
    public string GetCustomerNumber(Entity entity)
    {
        string rVal = 0;
        HotelCustomer.HotelCustomer obj = (HotelCustomer.HotelCustomer)entity;

        int nId = obj.Customer_ID;
        rVal = Convert.ToString(nId);

        return rVal;
    }

    [TargetProperty("Customer_ID")]
    public void SetCustomerNumber(Entity entity, string newValue)
    {
        HotelCustomer.HotelCustomer obj = (HotelCustomer.HotelCustomer)entity;

        int nId = Convert.ToInt32(newValue);
        obj.Customer_ID = nId;
    }

    [TargetProperty("Email")]
    public string GetCustomerEMail(Entity entity)[...]

    [TargetProperty("Email")]
    public void SetCustomerEMail(Entity entity, string newValue)[...]
}
```

그림 8 수정된 데이터 타입 컨버터 구현 코드 예

위 코드 샘플들에 나타난 바와 같이 비즈니스 엔티티, SCDT와 컨버터 인터페이스를 포함하는 서비스 컴포넌트, 데이터 타입 컨버터 메소드 구현 코드 등은 규칙 기반으로 작성되어 있고, 커스텀 애트리뷰트 주석을 사용하여 메타데이터를 표현하고 있으므로, 개발 도구를 사용하여 대부분을 자동으로 생성할 수 있다.

5. DTT 컴포넌트 모델 개발 도구

5.1 도메인 공통 서비스 컴포넌트 개발 도구

본 연구팀은 도메인 공통 서비스 컴포넌트 개발을 지원하는 자동화 도구를 구현하였다. 서비스 컴포넌트 개발 도구는 그림 9와 같은 SCDT 생성 마법사를 제공한다. 서비스 컴포넌트 개발자가 비즈니스 엔티티 대신 사용할 SCDT의 속성 타입과 이름을 입력하면, 개발 도구는 SCDT 코드 생성 알고리즘에 따라 그림 5의 SCDT 코드와 그림 6의 컨버터 인터페이스 코드를 자동 생성한다.

다음은 SCDT 코드 생성 알고리즘이다.

1. 개발중인 서비스 컴포넌트 안에 SCDT 클래스를 만들고 비즈니스 엔티티 객체를 수용하기 위한 루트 타입의 필드 변수를 선언한다.

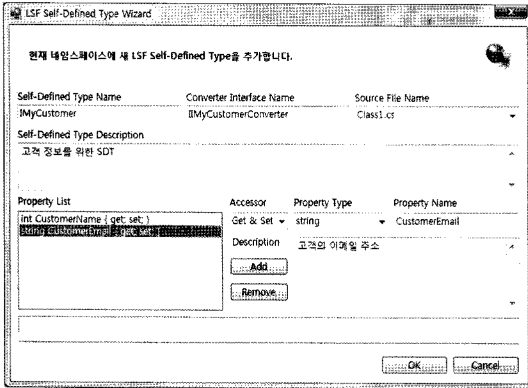

그림 9 SCDT 생성 마법사

2. SCDT 클래스에 루트 타입의 매개변수를 갖는 생성자를 정의하고, 동 클래스에 루트 타입을 상속한 비즈니스 엔티티 객체를 앞서 정의된 생성자를 사용하여 SCDT 객체로 변환하는 컨버전 오퍼레이터를 구현한다.
3. SCDT 클래스에 개발자로부터 입력받은 속성들의 타입과 이름을 이용하여 속성들을 정의하고 각 속성별로 Get 액세서와 Set 액세서를 선언한다.
4. 개발중인 컴포넌트 안에 앞서 선언된 Get 액세서 및 Set 액세서에서 차후 연동할 애플리케이션의 비즈니스 엔티티의 속성을 SCDT 속성으로 변환하는 데 사용할 메소드들로 구성된 컨버터 인터페이스를 정의한다.
5. 런타임 프레임워크의 의존성 주입 기능을 사용하기 위하여 SCDT 클래스에 애플리케이션의 데이터 타입 컨버터 객체의 레퍼런스를 수용할 변수로서 컨버터 인터페이스 타입의 필드와 속성을 선언한다.
6. 마지막으로 앞서 선언된 컨버터 객체 레퍼런스 필드의 변수를 사용하여 SCDT의 각 속성별로 Get 액세서와 Set 액세서를 완성하여 SCDT 정의를 완료한다.

5.2 데이터 타입 컨버터 개발 도구

컨버터 개발 도구는 그림 10과 같은 데이터 타입 컨버터 생성 마법사를 제공한다. 애플리케이션 개발자가 서비스 컴포넌트 SCDT와 대응되는 비즈니스 엔티티를 선택하고, SCDT의 속성과 비즈니스 엔티티의 속성을 맵핑하면, 개발 도구는 컨버터 코드 생성 알고리즘에 따라 그림 7의 데이터 타입 컨버터 코드를 생성한다.

다음은 데이터 타입 컨버터 코드 생성 알고리즘이다.

1. 서비스 컴포넌트에 정의된 SCDT 중 개발자가 선택한 SCDT와 대응되는 컨버터 인터페이스를 식별한다.
2. 식별된 컨버터 인터페이스를 구현하는 클래스를 생성한다.
3. 식별된 컨버터 인터페이스의 메소드 중 구현되지 않은 메소드 하나를 선택한다.
4. SCDT의 속성과 비즈니스 엔티티의 속성 중 선택된 메소드가 변환, 연동해야 하는 속성 쌍을 식별하고, 컨버터 클래스에 식별된 속성들의 값을 변환하는 코드를 작성하여 앞서 선택된 메소드의 구현을 완성한다.

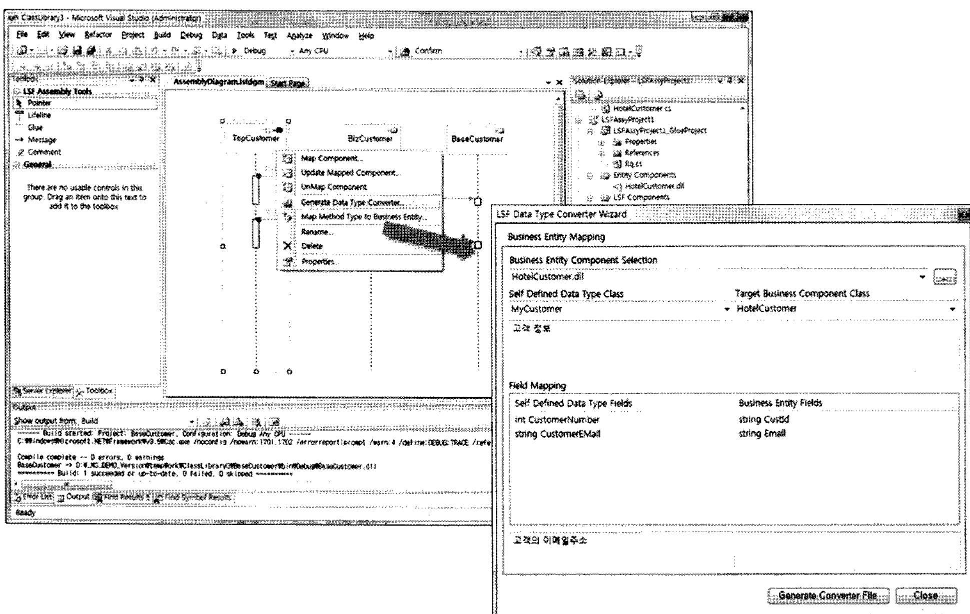

그림 10 데이터 타입 컨버터 생성 마법사

5. 식별된 컨버터 인터페이스의 메소드 중 구현되지 않은 메소드가 남아 있는 경우 모든 메소드가 구현될 때까지 메소드 구현 과정을 반복하여 선택된 컨버터 인터페이스를 구현하는 컨버터 클래스를 완성한다.

6. 고찰(Discussion)

본 논문에서는 엔터프라이즈 애플리케이션마다 달라지는 비즈니스 엔티티로 인한 가변성을 효율적으로 처리하는 DTT 컴포넌트 모델을 제안하였다. 또 DTT 컴포넌트 모델에 따르는 서비스 컴포넌트와 데이터 타입 컨버터 코드를 자동 생성해주는 개발 도구, 구성 요소들을 연동시켜주는 런타임 프레임워크를 구현하고 실험적 애플리케이션 개발을 통해 다음과 같은 결과를 얻었다.

비즈니스 엔티티 가변성 처리에 대한 실용적 구현 기술

기존의 PLE 연구들이 주로 기능적 가변성 처리에 대한 추상적 설계 수준에 머무른 반면 본 연구에서는 서비스 컴포넌트로부터 외부화된 비즈니스 엔티티 가변성 처리에 대해 구현 수준의 구체적인 방법을 제안하였다. DTT 컴포넌트 모델은 도메인의 공통 서비스 기능을 미리 구현할 수 있게 해주는 SCDT와 애플리케이션마다 달라지는 비즈니스 엔티티의 가변성을 다룰 수 있게 해주는 가변점으로서의 인터페이스 및 애플리케이션 구현시에 가변점 인터페이스를 상속하여 구현하는 데이터 타입 컨버터 등 필요한 모든 구성 요소를 구현 수준에서 표현할 수 있음을 확인하였다. 또한 SCDT, 가변점 인터페이스, 데이터 타입 컨버터 등을 자동으로 생성하는 절차를 구현한 개발 도구를 이용해 모든 구성 요소의 코드를 자동 생성할 수 있음을 확인하였다. 의존성 주입 기능을 수행하는 런타임 프레임워크가 구현된 서비스 컴포넌트, 데이터 타입 컨버터, 비즈니스 엔티티들을 인스턴스화하고 동적으로 결합하여 가변성 처리를 효율적으로 수행하는 환경을 지원하는 것을 확인하였다.

DTT 컴포넌트 모델의 제약사항

본 논문에서 제안한 방법이 모든 비즈니스 엔티티들의 가변성을 언제나 해결할 수 있는 것은 아니다. 서비스 컴포넌트 내에 정의된 SCDT의 모든 속성들이 새로운 애플리케이션의 비즈니스 엔티티의 속성과 매핑될 수 있는 경우에만 가변성을 처리한다. 즉 SCDT 속성들 중 하나라도 새로운 비즈니스 엔티티 속성과 의미론적으로 매핑될 수 없고, 구현된 서비스 메소드가 그 속성을 사용한다면 해당 메소드는 새로운 애플리케이션에서 사용될 수 없다.

DTT 컴포넌트 모델의 정당성

본 논문의 DTT 컴포넌트 모델은 도메인의 공통 서비스를 구현하는 서비스 컴포넌트가 애플리케이션 비즈니스 엔티티의 가변성을 처리하는 방법을 제안한 것이다. 특정 메소드를 구현하고자 할 때 그 메소드가 처리하는 데이터가 미리 정해져야만 하는 점을 감안하면, SCDT로 변환될 수 없는 비즈니스 엔티티에 대한 기능은 공통 기능으로 미리 식별될 수 없다는 것을 의미하므로 본 논문의 주장이 정당하다고 할 수 있다. 만약 어떤 기능이 비즈니스 엔티티의 가변성을 처리할 수 없다면 그 기능은 미리 식별할 수 있는 것이 아니며 기능 가변성을 통해 처리되어야 한다.

DTT 컴포넌트 모델의 성능 부담

DTT 컴포넌트 모델로 구현된 서비스 메소드에서 비즈니스 엔티티의 속성을 접근할 때마다 데이터 타입 컨버터 메소드의 로컬 호출을 동반한다. 그러나 다계층 분산환경에서 운영되는 엔터프라이즈 애플리케이션에서 필연적으로 수행될 수 밖에 없는 원격 호출은 로컬 호출에 비해 100~1000배 느리다. 따라서 데이터 타입 컨버터에 의한 성능 부담은 시스템의 전체 성능에 극히 미미한 영향을 줄 뿐이다.

DTT 컴포넌트 모델의 의의

함수 언어 시대의 함수들은 데이터와 분리되어 개발되므로 데이터가 바뀌면 코드 수정 없이 재사용할 수 없었다. 이에 비해 객체 지향 기술이 등장하면서는 데이터와 함수가 캡슐화된 객체가 애플리케이션의 구성 단위가 되었다. 객체에서는 데이터의 접근을 제어할 수 있고 데이터와 함수의 커플링이 클래스 수준에서 로컬화되므로 클래스의 독립적인 재사용이 유리했다. 그러나 규모가 커지고 분산화된 오늘날의 엔터프라이즈 애플리케이션에서는, 구현 효율성과 운영의 유연성을 지원하기 위해 다계층 구조로 설계되면서 데이터 즉 비즈니스 엔티티가 서비스 컴포넌트로부터 외부화되었다. 이에 따라 비즈니스 엔티티에 대한 의존성을 가질 수 밖에 없는 서비스 컴포넌트는 비즈니스 엔티티가 바뀔 때마다 소스 코드를 수정해야만 재사용이 가능하게 되었다. DTT 컴포넌트 모델은 서비스 컴포넌트가 외부화된 비즈니스 엔티티를 직접 다루지 않고 SCDT만을 다루게 함으로써 데이터와 함수의 커플링을 컴포넌트 수준에서 로컬화시키고, 서비스 컴포넌트들 소스 코드 수정 없이 컴포넌트를 재사용할 수 있게 한다.

7. 결 론

본 논문에서는 프로덕트 라인에서 데이터 가변성을 효율적으로 처리하는 DTT 컴포넌트 모델을 제안하였다. DTT 컴포넌트 모델은 서비스 컴포넌트가 외부화된 비즈니스 엔티티를 직접 다루지 않고 서비스 컴포넌트 내에 정의된 SCDT만을 다루게 함으로써 비즈니스 엔티티와 서비스 컴포넌트간의 의존 관계를 완화시켰다. 이는 데이터 가변성이 필연적인 대규모 엔터프라이즈

애플리케이션에서 효율적으로 재사용될 수 있는 프로덕트 라인 구축을 가능하게 한다.

본 논문에서는 또한 닷넷이나 자바 등의 컴포넌트 기술로 구현 가능한 구체적인 표현 기법과 코드 생성 알고리즘을 고안하였다. DTT 컴포넌트 모델의 각 구성요소 코드를 생성해주는 개발 도구와 런타임 프레임워크를 구현하였고, 실험적 애플리케이션 개발을 통해 실효성을 검증하고 그 결과를 고찰하였다.

향후 본 연구팀은 비즈니스 엔티티와 서비스 컴포넌트 SCDT에서 서로 매핑되는 속성들을 자동 식별하고 변환코드를 생성하는 연구를 추진하고 있다. 대규모 엔터프라이즈 애플리케이션에서 취급하는 비즈니스 엔티티는 그 종류와 개수가 매우 많기 때문에, 데이터 가변성 처리의 자동화는 애플리케이션 개발 생산성을 획기적으로 개선하는 효과를 줄 것이다.

참 고 문 헌

[1] Atkinson, C., et al., "Component-Based Product Line Engineering with UML," Addison Wesley, 2002.

[2] Hassan Gomaa and Diana L Webber, "Modeling adaptive and evolvable software product lines using the variation point model," Proceedings of the 37th Annual Hawaii International Conference on System Sciences, p. 10, 2004.

[3] 문미경, 염근혁, "소프트웨어 프로덕트 라인에서 가변성 분석을 통한 도메인 아키텍처 개발 방법", 정보과학회논문지 : 소프트웨어 및 응용 제34권 제4호, 2007. 4.

[4] 허진선, 김수동, "컴포넌트 프레임워크의 실용적 참조 모델", 정보과학회논문지 : 소프트웨어 및 응용 제33권 제6호, 2006. 6.

[5] John Cheesman and John Daniels, "UML Components: A Simple Process for Specifying Component-Based Software," Addison Wesley, 2001.

[6] "Patterns & Practices, Application Architecture for .NET: Designing Applications and Services," Microsoft Corporation, 2002.

[7] 김명호, "마이크로소프트 컴포넌트 기술의 발전과 동향", 정보과학회지: 제24권 제11호, 2006. 12.

[8] Joseph W. Yoder, Ralph E. Johnson, "The Adaptive Object-Model Architectural Style," Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, pp. 3-27, August 2002.

[9] H. Gomaa, D. Webber, "Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model," In Proceedings of the 37th Annual Hawaii International Conference on System Sciences, HICSS'04, pp. 1-10, IEEE Computer Society Press, January 2004.

[10] Barry Keepence and Mike Mannion, "Using patterns to model variability in product families," IEEE Software, Vol.16, Issue: 4, pp. 102-108, 1999.

[11] Matthias Clauß, "Generic Modeling using UML extensions for variability," OOPSLA 2001, Workshop on Domain Specific Visual Languages, 2001.

[12] Alan C. Wills, "Components and Connectors: Catalysis Techniques for Designing Component Infrastructures," In Component-Based Software Engineering: Putting the Pieces together, pp. 307-320. Addison Wesley, 2001.

[13] Yoonsun Lim, Myung Kim, Seungnam Jeong and Anmo Jeong, "A Reuse-Based Software Development Method," Proceedings of International Conference on Convergence and Hybrid Information Technology, Daejeon, Korea, August 2008.

임 윤 선

1987년 중앙대학교 수학과 학사. 2003년 이화여자대학교 컴퓨터학과 석사. 2004년~현재 이화여자대학교 컴퓨터학과 박사과정. 관심분야는 온톨로지, 소프트웨어 재사용, 프로덕트라인 공학, 지식공학 등

김 명

1981년 이화여자대학교 수학과 학사. 1983년 서울대학교 계산통계학과 석사. 1993년 캘리포니아 주립대학교(산타바바라) 컴퓨터학과 박사. 1993년~1994년 캘리포니아 주립대학교(산타바바라) 컴퓨터학과 Postdoc, 강사. 1995년~현재 이화여자대학교 컴퓨터학과 교수. 관심분야는 온톨로지, 고성능 컴퓨팅, 소프트웨어 재사용, 지식공학, 스트림 데이터처리 등

정 승 남

1984년 한양대 영어영문(학사). 1986년 한양대 영어영문(석사). 1992년~2000년 (주)아리스트 선임연구원. 2006년~현재 리버넥스 이사. 관심분야는 소프트웨어 재사용, 소프트웨어 공학, 데이터베이스, 프로그래밍 언어 등

정 안 모

1981년 서울대 물리교육(학사). 1984년~1992년 삼성전자 선임연구원. 1992년~2004년 (주)아리스트 대표이사. 2000년~2004년 이화여대 컴퓨터학과 겸임교수 2004년~현재 리버넥스 대표. 관심분야는 프로그래밍/컴포넌트 모델, 온톨로지, 프로그래밍 언어, 프로덕트라인 공학 등