http://dx.doi.org/10.3745/KIPSTD.2012.19D.2.187

서비스 지향 컴퓨팅을 위한 GoF 디자인 패턴 적용 기법

김 문 권[†] · 라 현 정[††] · 김 수 동[†††]

요 약

대표적인 재사용 패러다임 중 하나인 서비스 지향 컴퓨팅 (Service-Oriented Computing, SOC)는 독립적으로 실행가능하며 외부 인터페이스를 통해서만 접근 가능한 서비스를 재사용 단위로 사용한다. SOC는 서비스 지향 아키텍처 개념과 클라우드 컴퓨팅의 개념을 통칭하는 용어이다. 서비스는 서비스 제공자에게는 높은 재사용성으로 인해 수익을 내도록 하며, 서비스 소비자에게는 서비스를 재사용하여 보다 빠른 시간 내에 적은 노력으로 애플리케이션을 개발할 수 있는 경제성과 생산성을 제공한다. 디자인 패턴 (Design Patterns)는 객체 지향 소프트웨어 설계 시에 자주 발생하는 문제들을 해결하기 위한 범용적이며 재사용 가능한 방법들이며, Open/Closed 원칙을 이용하여, 가변성 및 여러 설계 이슈를 보다 쉽게 처리할 수 있는 설계 구조를 제안한다. 그러나 객체지향 패러다임의 객체와 SOC의 서비스는 구별되는 차이점을 가지고 있어, 기존의 디자인 패턴을 그대로 SOC에 적용하는 것은 어렵다. 서비스 제공자의 입장에서는 서비스 소비자마다의 가변적인 기능을 허용하며, 서비스의 고유 특징을 반영하는 서비스를 설계하고, 서비스 소비자 입장에서는 서비스가 제공하는 기능을 목적에 변경하여 빠른 시간 내에 목표 애플리케이션을 개발하도록 디자인 패턴이 SOC에 맞게 특화되어야 한다. 그러므로 본 논문에서는 서비스 제공자가 재사용성을 비롯한 서비스 고유의 특징을 반영하도록 서비스를 설계하고, 서비스 소비자는 제공되는 서비스를 목적에 맞게 특화하여 목표 애플리케이션을 개발하기 위해, SOC의 특성을 고려하여 특화된 디자인 패턴을 제안한다.

키워드 : 서비스 지향 컴퓨팅, 디자인 패턴, 재사용성, 서비스 특징, 가변성, 불일치 문제

Methods to Apply GoF Design Patterns in Service-Oriented Computing

Moon Kwon Kim[†] · Hyun Jung La[††] · Soo Dong Kim[†††]

ABSTRACT

As a representative reuse paradigm, the theme of service-oriented Paradigm (SOC) is largely centered on publishing and subscribing reusable services. Here, SOC is the term including service oriented architecture and cloud computing. Service providers can produce high profits with reusable services, and service consumers can develop their applications with less time and effort by reusing the services. Design Patterns (DP) is a set of reusable methods to resolve commonly occurring design problems and to provide design structures to deal with the problems by following open/close princples. However, since DPs are mainly proposed for building object-oriented systems and there are distinguishable differences between object-oriented paradigm and SOC, it is challenging to apply the DPs to SOC design problems. Hence, DPs need to be customized by considering the two aspects; for service providers to design services which are highly reusable and reflect their unique characteristics and for service consumers to develop their target applications by reusing and customizing services as soon as possible. Therefore, we propose a set of DPs that are customized to SOC. With the proposed DPs, we believe that service provider can effectively develop highly reusable services, and service consumers can efficiently adapt services for their applications.

Keywords : Service Oriented Computing, Design Pattern, Reusability, Service Unique Characteristics, Variability, Mismatch Problem

1. 서 론

널리 사용되는 재사용 패러다임 중 하나인 서비스 지향 컴퓨팅 (Service-Oriented Computing, SOC)는 독립적으로 실행가능하며, 외부 인터페이스를 통해서만 접근 가능한 서비스를 재사용 단위로 한다[1][2]. 본 논문에서 SOC는 서비스 지향 아키텍처 개념과 클라우드 컴퓨팅을 포함하여, 사용자의 컴퓨터에서 모든 기능이 실행되는 것이 아니라, 외부에 설치되어 있는 재사용 가능한 다양한 서비스를 이용하

※ 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업 지원을 받아 수행된 연구임(No.2011-0002534).
† 준 회 원 : 숭실대학교 컴퓨터학과 석사과정
†† 정 회 원 : 숭실대학교 모바일 서비스 소프트웨어공학센터 연구교수(교신저자)
††† 종신회원 : 숭실대학교 컴퓨터학부 교수
논문접수 : 2012년 1월 20일
수 정 일 : 1차 2012년 3월 30일
심사완료 : 2012년 4월 11일

여 기능을 수행하는 개념을 일컫는다. SOC에서 서비스 제공자는 잠재적인 여러 서비스 사용자를 위해 범용적이고 재사용 가능한 서비스를 개발한다. 서비스 소비자는 서비스 저장소에서 적절한 서비스를 검색하고 발견하며, 해당 서비스를 구독하는 방식으로 목표 애플리케이션을 개발한다. 여러 형재의 서비스 종류가 있지만 본 논문에서는 소프트웨어 기능을 서비스 형태로 제공하는 Component-as-a-Service (CaaS) [2], Software-as-a-Service (SaaS) [2]만 고려한다.

서비스는 다양한 서비스 소비자들에 의해 재사용되는 필수적인 기능 단위이므로, 가능한 많은 소비자들이 재사용할 수 있도록 공통적인 기능을 최대한 제공해야 한다. 그리고, 가능한 모든 소비자들의 다른 요구사항들을 고려하여 서비스를 설계해야 한다. 서비스 제공자가 서비스 설계 시 잠재적인 서비스 소비자를 알지 못하므로, 이런 재사용성이 높은 서비스를 설계하는 것은 어렵다. 이 외에도, 서비스가 자체적으로 가지는 느슨한 결합, 제한된 관리성 등의 고유한 특징을 반영하도록 설계되어야 한다. 그리고, 서비스 소비자는 이미 제공되는 서비스는 자기 목적에 맞게 수정하는 것이 불가능하므로, 어느 정도 서비스를 자기 목적에 맞게 일부를 수정하는 것을 가능하게 하면 서비스 재사용성은 더욱 높아질 것이다. 즉, 서비스 제공자의 입장에서는 서비스 소비자마다의 가변적인 기능을 허용할 수 있도록 하며, 서비스 자체의 특징을 반영하는 서비스를 설계하는 것이 중요하며, 서비스 소비자 입장에서는 서비스가 제공하는 기능을 목적에 변경하여 빠른 시간 내에 목표 애플리케이션을 개발하는 것이 중요하다.

디자인 패턴 (Design Patterns)은 소프트웨어 설계 시에 자주 발생하는 문제들을 해결하기 위한 범용적이며 재사용 가능한 방법들이며, 객체와 객체들 간의 상호작용 및 연관관계를 보여주는 구조로 명세 한다[3]. 대표적인 디자인 패턴 중 하나인 GoF (Gang of Four) 패턴은 총 23개의 패턴으로 구성되며 각 패턴들의 특징 및 목적에 따라 생성 패턴, 구조 패턴, 행위 패턴으로 그룹화 된다. 생성 패턴은 객체의 생성, 구성, 표현시에 자주 발생하는 절차를 추상화하는 재사용 가능한 범용적인 패러다임으로 Abstract Factory 패턴을 포함하여 5개 패턴들로 구성된다. 구조 패턴은 소프트웨어의 구조를 정하고 구축함에 있어, 클래스 및 객체 구성의 절차를 범용적으로 적용 가능하도록 하는 7개의 패턴들로 구성되며 Adapter 패턴이 포함된다. 이 패턴은 독립적으로 구축된 클래스들을 구조화 함에 있어 특히 효율적이다. 행동 패턴은 객체들간의 관계 및 소통 절차를 재사용 가능하도록 정의하며 Strategy 패턴을 포함하여 11개의 패턴들로 구성된다. 이 패턴은 특히 복잡한 제어흐름을 가지는 소프트웨어 구축에 유용하다. GoF 패턴은 Open/Closed 원칙을 이용하여, 가변성 및 여러 설계 이슈를 보다 쉽게 처리할 수 있는 설계 구조를 제안한다. 그러나, 기존의 디자인 패턴을 SOC에 적용하는 것은 다음의 이유로 쉽지 않다.

- 기존 디자인 패턴은 객체 지향 패러다임에 맞게 제안된 것이므로, SOC 고유의 특징인 블랙박스 형태의 서비스, 느슨한 결합도 등의 특징 [1][2][4]을 효과적으로 반영하지 않는다.
- 기존 디자인 패턴은 재사용 관점에서 서비스 소비자와 서비스 제공자의 구별된 역할을 고려하지 않는다.

디자인 패턴은 객체를 기반으로 하고 있지만 서비스 역시 구현 관점으로 보면 객체들의 집합으로 구성되기 때문에, SOC의 고유한 특징을 반영하도록 수정한다면 디자인 패턴의 장점을 SOC에 접목할 수 있을 것이다. 그러므로, 본 논문에서는 서비스 제공자가 재사용성을 포함한 서비스 고유한 특징을 반영한 서비스를 설계하고, 서비스 소비자는 제공되는 서비스를 목적에 맞게 특화하여 목표 애플리케이션을 개발하기 위해, SOC의 특성을 고려하여 특화된 디자인 패턴을 제안한다. 3장에서 객체 지향 패러다임과 SOC의 차이점을 구성 단위 관점과 프로세스 관점에서 비교한다. 분석된 차이점을 기반으로 SOC에서 적용 가능한 디자인 패턴을 4장에서 도출한다. 도출된 디자인 패턴 중 각 유형별 대표적으로 사용되는 Abstract Factory 패턴, Adapter 패턴, Strategy 패턴에 대한 SOC에 특화된 명세를 정의한다. 마지막으로 6장에서는 각 패턴을 실제 서비스 개발 및 서비스 기반 애플리케이션에 적용한 결과를 기술하여, 제안된 패턴의 적용 가능성을 확인한다.

SOC 의 목적에 맞게 수정된 디자인 패턴을 이용하면, 서비스 제공자는 높은 재사용성을 비롯한 서비스 고유한 특징을 모두 반영하는 서비스를 효과적으로 설계할 수 있게 되고, 서비스 소비자는 효율적으로 서비스를 이용하여 목표 애플리케이션을 개발할 수 있게 된다.

2. 관련 연구

Erl의 연구에서 서비스 관리, 설계, 구성을 위한 다른 종류의 패턴들을 소개한다[6]. 이 연구에서 소개하는 패턴들은 객체지향 패턴과 아키텍처 패턴에서 유도, 확장된다. 그리고 각 패턴을 적용 가능한 문제, 해결책, 적용 방법, 다른 패턴들과의 관계와 함께 명세한다. 이 연구에서는 주로 서비스 구현에 유용한 패턴, 서비스 보안에 유용한 패턴 등을 소개하며 다양한 설계 문제를 다루고 있지만 상세 설계 없이 패턴들을 소개하고 있으며 사례 연구가 자세하게 다루어 지지 않고 있는 한계가 있다. Zdun의 연구에서는 GoF 패턴들뿐 아니라 원격, 메시지, Networked and Concurrent Object, 소프트웨어 아키텍처 패턴과 같은 전통적인 소프트웨어 패턴들을 적용하여 서비스를 설계하는 접근법을 제안한다[7]. 이 연구에서는 서비스 탐색 패턴, 서비스 계약 기술을 위한 인터페이스 기술 패턴, 서비스의 다양한 가변점을 다루기 위한 패턴, 서비스 통합을 위한 패턴들을 설명한다. 이 연구는 SOC에서 비즈니스 프로세스에 대한 문제들을 해결하는 패턴들을 소개하고 있지만 전통적 패턴들의 적용 지침이 개선될 필요가 있다. Topaloglu의 연구에서는 웹 서비스의 가변성을 디자인 패턴을 이용하여 표현하는 방법을 제안한다[8]. 먼저, 단일 서비스에서 발생 가능한 가변점을 WSDL 인터페이스의 명세 요소, 프로토콜 종류, 웹 서비스의 기능으로

분류하고, 이 각각을 Decorator, Adapter, Strategy 패턴으로 명세하였다. 그리고, 복합 서비스의 가변점은 포함되는 서비스의 수, 서비스 호출 순서로 분류하고, Composite, Iterator, Chain of Responsibility 패턴을 이용하였다. 이 연구는 가변성 설계 보다는 가변성 명세에 초점을 맞추었다. Milanovic의 연구에서는 서비스 기반 애플리케이션을 디자인 패턴을 이용하여 개발하는 방법을 소개한다[9]. 이 연구에서는 먼저 클라이언트와 서버 사이의 결합도를 줄이기 위한 Proxy 패턴의 세 가지 변경사항과 웹 서비스 구성에 접근하기 위한 Façade 패턴의 두 가지 변경 사항을 설명한다. 그리고 보안, 서비스 메시지 복잡도, 성능에 관련된 문제들을 해결하기 위한 새로운 패턴들을 소개한다. 이 연구는 패턴을 적용하는 지침이 상세하지 않고 Proxy 패턴과 Façade 패턴만을 주로 활용하고 있는 한계가 있다. Mauro의 연구에서는 Service-oriented Device Architecture (SODA)에 대한 일곱 가지 설계 문제를 정의하고, 이 문제들을 해결하기 위해 Erl의 연구[6]에서 제안하는 SOC 패턴들을 분석하였다[10]. 그리고 기존 패턴들이 다루지 않는 문제들을 위한 일곱 가지 새로운 디자인 패턴들을 제시하였다. 그러나 제시하고 있는 패턴들은 SODA만을 위해 정의된 것이고, 이 패턴을 적용하는 상세한 지침이 부족하다. Thu의 연구에서는 GoF 패턴들을 적용하여 서비스들을 통합하는 패턴들을 소개한다[11]. 먼저 서비스들을 통합하는데 발생하는 세 가지 문제를 분류하고, 이러한 문제들을 해결하기 Adapter, Abstract Factory, Singleton 패턴을 사용하였다. 그러나 제안하고 있는 패턴들은 서비스 통합에 대한 문제만을 다루고 있고, 패턴을 적용하는 상세한 지침이 부족하다는 한계가 있다. Arcelli의 연구에서는 레거시 애플리케이션을 SOC로 이전할 때 효과적으로 적용 될 수 있는 패턴들을 제안한다[12]. 먼저 SOC 이주를 위한 12 가지의 디자인 패턴, Façade, Mediator, Singleton, Abstract Factory, Bridge, Decorator, Adapter, Proxy, Command, Chain of Responsibility, State, Observer을 기술한다. 그리고 각 패턴을 일반적 용법을 설명하는 Intent와 이주 목적을 고려한 필수적인 측면을 설명하는 Relevance와 함께 설명한다. 서비스 이주 시에 어떻게 패턴을 효과적으로 적용 할 것인가에 대한 적용 지침이 상세하지 않다.

이렇게 기존 연구들에서 제안하고 있는 패턴들은 가변성 설계가 미흡하고, 서비스가 가지는 고유한 특성을 고려하지 않고 정의된 것이 대부분이다. 그리고, SOC에 적용하기 위한 상세한 설계, 적용 지침, 사례 연구가 부족하다. 본 연구에서는 SOC에서 발생 할 수 있는 가변점을 찾고 가변성을 고려하여 SOC의 특성 및 목적에 맞게 기존 GoF 패턴을 수정하고 수정된 패턴의 적용 지침 및 적용 사례를 제시한다.

3. 객체지향 컴퓨팅과 서비스 지향 컴퓨팅의 비교

객체지향 패러다임과 SOC 차이점으로 인해, 기존의 디자인 패턴을 수정 없이 SOC에 적용하는 것이 어렵다. 이 장에서는 객체 지향 패러다임과 SOC를 1) 빌딩 블록 관점과 2) 개발 프로세스 관점에서 유사점 및 차이점을 비교 분석한다. 분석된 결과는 GoF 패턴의 효과적인 SOC 적용 지침 개발에 사용된다.

3.1 객체와 서비스 비교

빌딩 블록 관점에서는 각 패러다임의 주요 빌딩 블록인 “객체”와 “서비스”를 비교한다. 객체지향 패러다임과 SOC에 관한 대표 문헌[1][2][4][13]을 조사 및 분석하여, 두 빌딩 블록의 차이점을 <표 1>과 같이 정리한다.

〈표 1〉 객체와 서비스 간의 차이점

	객체	서비스
재사용성 범위	단일 목표 시스템 내	여러 애플리케이션들
기능 크기	상대적으로 작은 크기의 기능성 제공	상대적으로 큰 크기의 기능성 제공
인터페이스	구현과 인터페이스가 완전히 분리되지 않음	구현과 인터페이스가 완전히 분리됨
바인딩 영속성	하드 바인딩	소프트 바인딩
획득 방법	구매를 통한 획득	구독으로 통한 서비스 획득

재사용성 범위 관점에서 보면, 객체는 단일 애플리케이션을 위해 설계되는 기능 단위이지만, 서비스는 여러 애플리케이션에서 공통적으로 사용되는 기능을 제공하기 위해 설계되는 단위이다. 예를 들어, 학사 관리 애플리케이션만을 위해, Student, Lecture, Course 등의 객체를 설계한다. 반면에, 구글 Map 서비스는 특정 애플리케이션이 아니라 다수의 애플리케이션을 위해 설계한 범용적인 기능을 제공한다.

기능 크기 관점에서 보면, 객체는 비교적 작은 단위의 기능성을 제공하지만, 서비스는 대개 서비스 소비자에게 독립적인 기능을 제공하기 때문에 객체보다는 상대적으로 큰 크기의 기능성을 제공한다. 인터페이스 관점에서 보면, 객체지향 패러다임에서는 구현과 인터페이스가 명확하게 분리되지 않는다. 그러나, 서비스는 명확하게 서비스 구현체와 서비스 인터페이스가 분리되어 있고, 서비스 소비자는 오로지 서비스 인터페이스를 통해서만 기능을 호출할 수 있다.

바인딩 영속성 관점에서 보면, 애플리케이션을 컴파일하면 객체는 해당 애플리케이션에 영속적으로 바인딩된다. 그러나, 서비스는 애플리케이션에 영속적으로 바인딩되는 것이 아니므로, 서비스 인터페이스에 대한 어느 구현체도 사용할 수 있다. 획득 방법 관점에서 보면, 객체 지향 애플리케이션은 패키지 형태로 구매를 통해 획득 가능하므로 영구적으로 소유 가능하다. 그러나, 서비스는 목적에 맞는 서비스를 구독하여 사용하는 형태이기 때문에 영구적으로 소유하는 형태는 아니다.

이 외에 객체 지향 방식에서 나타나지 않지만, 서비스 또는 서비스 기반 애플리케이션 개발 시 고려해야 하는 다음과 같은 서비스 고유의 특징이 있다[1][2][4].

• 느슨하게 결합되는 서비스 (Loosely-Coupled Nature)

SOC에서 서비스 제공자와 서비스 소비자 간의 사전 지식없이 서비스 제공자는 서비스를 배포하고, 서비스 구매자는 서비스를 구독할 수 있다. 그러므로 서비스 제공자와 구매자는 서로 느슨하게 연결되어 있고, 이들간의 관계는 정적으로 고정된 것이 아니라 동적으로 변경 가능하다.

• 서비스 진화 (Evolvability of Services)

서비스 저장소에 이미 배포된 서비스는 서비스 소비자의 동의 없이도 진화할 수 있다. 이런 서비스의 특성 때문에 SOC에서 서비스가 동적으로 발견되어야 한다는 요구사항이 생긴다.

• 제한적인 관리성 (Limited Manageability)

서비스는 블랙박스 형태로 개발되기 때문에, 서비스 소비자는 서비스의 내부에 접근 할 수 없다. 그러므로 서비스를 모니터링하고 관리하는 것은 기존의 시스템을 관리하는 것보다 더욱 어렵다.

• 부분 일치와 적응성 (Partial Matching and Adaptability)

서비스는 공통적인 기능을 제공하는 단위이다. 그러나 서비스 소비자 간에도 이미 배포된 서비스에 대해 요구하는 것이 약간 다르므로, 서비스를 동적으로 적응시켜 제공하는 것이 필요하다.

• 멀티 테넌시 (Multi Tenacy)

SOC에서 제공되는 여러 서비스 종류 중 Software-as-a-Service는 동시에 여러 소비자가 서비스가 제공하는 기능을 동시에 사용할 수 있다는 특징을 가지고 있다. 그러므로, SOC 서비스는 현재 기능을 사용하고 있는 각 소비자별로 각각의 상태를 관리할 수 있도록 설계되어야 한다.

이렇듯, 여러 관점에서 서비스와 객체는 다른 특성을 가지고 있으므로, 객체를 기반으로 하는 디자인 패턴은 SOC의 특징과 서비스의 특징을 반영하여 수정이 되어야 한다.

3.2 개발 프로세스 관점 비교

객체지향 패러다임은 단일 애플리케이션 개발을 목적으로 하기 때문에, 애플리케이션 개발자만이 참여자로 관여한다. 반면에, SOC에서는 서비스를 개발하는 서비스 제공자와 서비스를 이용하여 서비스 기반 애플리케이션을 개발하는 서비스 소비자의 두 가지 다른 목적을 가진 참여자가 존재한다.

객체지향 개발 프로세스는 일반적으로, 요구사항 명세서 정의, 요구사항 분석, 애플리케이션 설계, 애플리케이션 구현, 테스팅의 단계를 거쳐 객체 지향 애플리케이션을 개발하는 절차를 포함한다.

SOC에서 표준화된 서비스 및 서비스 기반 애플리케이션 프로세스는 없다. 그러므로, SOC 관련 여러 문헌 [1][14][15][16][17][18]들을 조사하여, (그림 1)과 같은 서비스 개발 프로세스와 서비스 기반 애플리케이션 프로세스를 정리할 수 있다.

서비스 제공자 입장에서는 재사용성 높은 서비스를 개발하기 위하여 서비스 분석 단계 (P.P2)에 서비스로 제공될

(그림 1) SOC에서 두 가지 참여자를 고려한 개발 프로세스

기능 중 가변성이 있는지를 분석한다. 가변성은 공통적인 기능 내에 존재하는 여러 서비스 소비자들 간의 다른 기능을 의미하고, 가변점과 가변치로 표현된다[19]. <표 2>는 SOC에서의 가변성 종류를 제안한 기존 문헌들을 조사한 결과를 보여준다. 이들의 연구를 종합하면, 재사용성이 높은 서비스를 설계하기 위해서, 서비스 제공자는 동일한 기능을 다른 인터페이스로 제공하는 인터페이스 가변성, 복합 서비스의 경우 서비스 간의 실행 경로에 차이가 있는 워크플로우 가변성, 복합 서비스에 참여하는 서비스 구현체가 다른 조합 가변성, 단일 서비스의 구현 로직이 다른 로직 가변성을 고려해야 한다.

이런 분석된 결과를 이용해서, 서비스를 설계한다 (P.P3). 재사용성이 높은 서비스를 설계하기 위해서, 이미 분석된 가변성들을 서비스 소비자 또는 서비스 제공자가 보다 쉽게 수정할 수 있도록 디자인 패턴을 이용한다. 그러므로 SOC에 특화된 디자인 패턴은 이렇게 SOC만의 가변성을 만족시킬 수 있도록 확장 및 수정되어야 한다.

〈표 2〉 기존 문헌에서 정의된 가변성 종류

연구	제안된 가변성 종류
Topaloglu 연구[8]	서비스 인터페이스 가변성 (파라미터, 프로토콜), 서비스 구현 가변성, 조합 가변성, Workflow 가변성
Jegadeesan 연구[20]	서비스 인터페이스 가변성, 서비스 구현 가변성
Chang 연구[21]	워크플로우 가변성, 조합 (Composition) 가변성, 로직 가변성, 인터페이스 가변성
Kim 연구[22]	인터페이스 가변성, 웹 서비스 Flow 가변성, WSLA 가변성

서비스 소비자 입장에서는 서비스를 재사용하여 보다 빠르게 목표 애플리케이션을 개발하기 위하여 서비스 기반 애플리케이션 분석 단계 (C.P2)에 목표 애플리케이션의 요구사항 중 일부 기능과 일치하는 서비스가 있는지를 검색 및 식별하고, 검색된 서비스가 요구사항을 완전히 만족하는지 또는 일부만 만족하는지를 분석한다. 그리고 서비스 기반 애플리케이션 설계 단계 (C.P3)에 불일치가 있는 서비스를 목표 애플리케이션에 사용하기 위해 불일치 문제를 해결하고, 서비스와 애플리케이션 특화된 기능을 통합 설계한다. [23]의 연구에서 불일치 종류를 크게 인터페이스 불일치, 기능 불일치, QoS 불일치로 분류하였으며, QoS 불일치는 설계 수준이 아닌 아키텍처 설계 수준에서 해결되어야 하는 문제이므로, 본 논문에서 고려하지 않는다. 서비스 불일치 문제를 해결하여 기존 서비스를 재사용하기 위해, 제공된 서비스의 일부를 목표 애플리케이션에 맞게 수정해야 하고, 이 역시 디자인 패턴을 이용하여 가능해진다. 그러므로 SOC에 특화된 디자인 패턴은 기존의 구현체를 직접적으로 수정하기 않고, 일부를 수정하는 것을 가능하게 하기 위해 확장 및 수정되어야 한다.

객체 지향 패러다임과 SOC를 빌딩 블록 관점과 개발 프로세스 관점에서 비교하면 많은 차이점이 있다. 이런 차이점과 기존 디자인 패턴 적용 여부 관계를 요약하면, 서비스 제공자는 높은 재사용성 보장을 포함한 여러 서비스 고유의 특징을 반영하기 위하여 디자인 패턴을 적용하고, 서비스 소비자는 서비스 불일치 문제와 동적 바인딩을 해결하기 위하여 디자인 패턴을 적용할 수 있다. 그리고, SOC 디자인 패턴은 객체가 아닌 서비스의 주요 구성 요소인 서비스 인터페이스와 서비스 구현체로 구성되어야 한다. 그러므로, (그림 2)에 기술한 기준들을 기반으로 기존 디자인 패턴으로부터 SOC에 특화된 목적을 가지는 디자인 패턴으로 수정 및 확장해야 한다.

4. 서비스 지향 컴퓨팅에 적용 가능한 디자인 패턴

3장에서 구별한 SOC와 객체지향 패러다임의 차이점을 바탕으로, 본 장에서는 SOC에 디자인 패턴을 적용하는 이유와 각 GoF 디자인 패턴의 적용 가능한 상황을 비교하여, SOC에서 적용할 수 있는 GoF 디자인 패턴을 식별한다.

4.1 생성 패턴에 대한 SOC 적용성

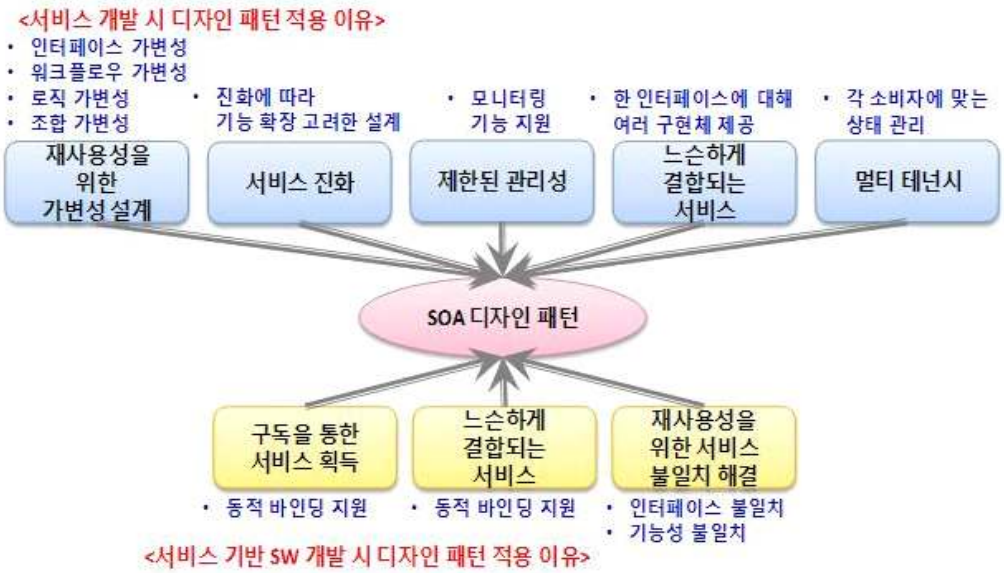
생성 패턴은 객체 생성 과정을 주로 다루며, 서비스 제공자 입장에서 단일 또는 복합 서비스를 구성하기 위한 여러 개의 관련된 구현체를 생성할 때 주로 사용될 수 있다. 다음은 각 패턴에 대한 적용 가능성 유무와 그에 대한 이유이다.

- Abstract Factory 패턴 : 서비스 제공자 측에 적용 가능함

Abstract Factory 패턴은 서로 관련 있는 객체들을 생성하기 위해 사용되며, SOC에서 단일 서비스 인터페이스에 대한 구현체들을 생성 또는 복합 서비스에 포함되는 여러 단일 서비스를 생성하는 상황과 일치한다. SOC에서는 서비스 인터페이스와 서비스 구현체가 분리되어 있기 때문에, 하나의 기능을 제공하기 위하여 여러 클래스들이 서비스 구현체를 구성할 수 있다. 이 패턴을 이용하면, 서비스 구현체 구성에 대한 부분과 인터페이스가 분리되므로, 단일 서비스에 대한 여러 구현체를 제공하거나, 복합 서비스에 포함되는 여러 단일 서비스 집합을 제공하면 여러 소비자에게 비슷하지만 조금씩 다른 기능을 제공할 수 있게 된다. 그리고, 서비스 진화로 인해 서비스 구현체가 변경될 경우에도 인터페이스 변경 없이 보다 쉽게 구현체를 변경할 수 있다.

- Builder 패턴 : 서비스 제공자 측에 적용 가능함

Builder 패턴은 동일한 생성 프로세스를 이용하여 다양한 형태의 복합 객체를 생성하기 위하여 사용되며, Abstract



(그림 2) SOC를 위한 디자인 패턴 도출에 적용되는 기준

Factory 패턴과 유사하게 단일 서비스 인터페이스에 대한 여러 구현체 또는 복합 서비스에 대한 여러 개의 단일 서비스 집합을 제공하는 상황과 일치한다. 이 패턴을 이용하여, 한 인터페이스에 대한 여러 기능을 제공하는 서비스를 실현할 수 있게 되며, 서비스 진화로 인한 서비스 구현체 변경이 쉬워진다.

- Factory Method 패턴 : 서비스 제공자 측에 적용 가능함

Factory Method 패턴은 객체 생성을 위한 인터페이스를 제공하여, 실제 객체를 생성하는 부분과 인터페이스를 구분하기 위해 사용되며, SOC에서 동일한 인터페이스로 비슷하지만 다른 기능을 하는 객체를 생성하여 이를 동적 바인딩하는 상황과 일치한다. 즉, 위의 패턴들과 유사하게 한 인터페이스에 대한 다른 기능을 제공할 경우에 잘 적용되지만, Factory Method 패턴은 여러 객체들로 구성된 서비스 구현체를 생성하는 것이 아니라 한 객체로 이루어진 구현체를 생성한다.

- Prototype패턴과 Singleton 패턴 : 서비스 제공자와 서비스 소비자 측에 모두 적용되지 않음

Prototype 패턴은 이미 생성된 객체를 생성해서 새로운 객체를 생성하기 위해 사용하고, Singleton 패턴은 해당 클래스에서 한 인스턴스만 생성하기 위해 사용한다. 이 두 패턴 모두 서비스의 특징을 반영하기 위해 직접적으로 사용되지 않으므로, SOC에서 적용 가능한 패턴으로 선정하지 않는다.

4.2 구조 패턴에 대한 SOC 적용성

구조 패턴은 큰 단위의 구조체를 클래스 단위로 구성하는 방법을 다루며, 서비스 제공자 입장에서는 서비스의 특징을 만족시키도록 서비스 구현체를 구성하는 목적으로, 서비스 소비자는 서비스를 재사용하여 목표 애플리케이션을 구성하기 위해 사용된다. 다음은 각 패턴에 대한 적용 가능성 유무와 그에 대한 이유이다.

- Adapter 패턴: 서비스 제공자와 서비스 소비자 측에 모두 적용 가능함

[서비스 제공자 측] Adapter 패턴은 클래스의 인터페이스를 사용자가 원하는 형태로 변환하는데 사용되며, 이는 SOC의 대표적인 가변성 유형 중 하나인 인터페이스 가변성은 어댑터 패턴의 적용 가능 상황과 잘 일치한다. 그러므로, 여러 소비자가 기대하는 다양한 형태의 서비스 인터페이스를 고려하여 서비스를 설계하는데 어댑터 패턴 구조를 이용하면, 많은 소비자가 서비스를 재사용할 수 있게 된다. 그리고, 서비스 진화성을 반영하여 서비스의 인터페이스는 유지한 채 서비스의 구현체는 쉽게 교체할 수 있다. 이를 위해 서비스의 구현체는 반드시 해당 서비스 인터페이스를 구현해야 하는데, 일부 서비스 구현체의 많은 부분이 수정되어 서비스 인터페이스를 준수하지 못하는 경우가 발생한다. 이 경우에도 서비스 구현체의 인터페이스를 기존에 등록된 서비스 인터페이스로 변환하기 위해 어댑터 패턴이 적용될 수 있다.

[서비스 소비자 측] 서비스의 기능성이 만족하다 하더라도, 서비스 인터페이스가 일치하는 경우가 많다 (인터페이스 불일치 문제). 게다가 서비스 소비자 측에서 기존의 인터페이스를 변경할 수 없는 상황이라면, 전혀 해당 서비스를 사용할 수 없게 된다. 이 경우는 어댑터 패턴의 적용 가능한 상황과 일치하며, 어댑터 패턴으로 인터페이스 불일치 문제를 쉽게 해결할 수 있게 된다.

- Bridge 패턴: 서비스 제공자와 서비스 소비자 측에 모두 적용 가능함

Bridge패턴은 인터페이스와 구현체의 구조를 분리하여, 하나의 인터페이스가 여러 개의 구현체를 가지는 것을 가능하게 하고, 하나의 구현체가 여러 개의 인터페이스를 구현하는 것도 가능하게 한다.

[서비스 제공자 측] 서비스는 근본적으로 잠재적인 여러 소비자들을 고려하여 재사용성을 보장하도록 설계되어야 한다. 그러나, 설계시에 모든 소비자들을 파악하고 이를 고려한 가변성을 설계하는 것은 매우 어렵다. 그러므로, 잠재적인 서비스 소비자의 가변치를 허용할 수 있도록 서비스가 유연하게 설계되어야 하며, 이는 Bridge 패턴의 적용가능한 상황과 잘 일치한다. 그러므로, 잠재적인 소비자들의 기능 가변치를 보장하게 하기 위해, 동일한 인터페이스로 다양한 구현체를 제공하는데 사용된다.

[서비스 소비자 측] 서비스 기반 애플리케이션은 동일한 기능을 위해 여러 서비스를 구독 및 호출하여, 소비자에게 각자에 맞는 다른 기능을 제공할 수 있다. 인터페이스와 해당 서비스를 호출하는 부분이 분리되면, 이런 상황을 보다 쉽게 구현할 수 있게 되며 이것이 Bridge 패턴의 적용 상황과 일치한다. 이는 서비스 기반의 모바일 애플리케이션에 더욱 잘 발생하는 상황이다. 그러므로, 동일한 인터페이스로 사용자의 현재 상황에 맞는 적절한 서비스를 호출하는데 사용된다.

- Composite 패턴: 서비스 제공자와 서비스 소비자 측에 모두 적용되지 않음

Composite 패턴은 여러 객체들을 이용하여 하나의 복합 객체를 구성하고, 동일한 인터페이스로 복합 객체 또는 이에 포함된 단일 객체의 기능을 호출하는데 사용된다. SOC에서 동일한 인터페이스를 가진 여러 서비스 구현체를 재귀적으로 호출하는 경우가 희박하므로, 적용 가능한 패턴으로 선정하지 않는다.

- Decorator 패턴: 서비스 제공자와 서비스 소비자 측에 모두 적용 가능함

Decorator 패턴은 한 객체에 동적으로 기능을 확장하기 위해 사용되며, 상속 관계보다 보다 유연하게 기능을 확장할 수 있다. 이는 SOC에서 서비스 인터페이스에 해당한 구현체에 동적으로 기능을 확장하기 위해서 사용될 수 있으며, 각 적용 가능한 상황은 다음과 같이 다르다.

[서비스 제공자 측] 서비스는 제한된 관리성을 가지고 있으므로, 현재 서비스 상태를 파악하기 위해 모니터링을 할 필요성이 있다. 이는 서비스가 제공하는 고유의 기능 외의 추가적인 기능으로 하나의 서비스 구현체로 구현할 경우, 서비스 구현이 다소 복잡해진다. 그러므로, 서비스가 제공하

는 기능 외에 모니터링 등의 서비스 관리 목적의 기능을 추가할 때 Decorator 패턴을 적용할 수 있다.

[서비스 소비자 측] 서비스 재사용으로 발생하는 여러 불일치 문제 중, 서비스가 제공하는 기능 외의 추가적인 기능을 필요로 할 때, Decorator 패턴을 적용할 수 있다. 이 패턴을 이용하면 서비스 구현체를 직접적으로 수정하지 않고, 필요하지만 서비스가 제공하지 않는 기능을 추가할 수 있다.

- Façade 패턴과 Flyweight 패턴: 서비스 제공자와 서비스 소비자 측에 모두 적용되지 않음

Façade 패턴은 서브 시스템에 포함된 모든 인터페이스들에 대한 단일 인터페이스를 제공하기 위해 사용되며, Flyweight 패턴은 여러 개의 유사한 객체 생성 및 관리 비용을 절감하기 위해 유사한 객체들이 사용하는 데이터를 공유하기 위해 사용된다. 이 두 가지 패턴의 적용 가능한 상황은 SOC에서 발생하는 문제와 일치하지 않으므로, 적용 가능한 패턴으로 선정하지 않는다.

- Proxy 패턴: 서비스 제공자와 서비스 소비자 측에 모두 적용 가능함

Proxy 패턴은 실제 객체가 일을 수행하기 이전에 부수적인 기능을 실행해야 하는 경우에 사용되며, 다른 장소에 위치하는 객체를 호출하는데 사용하는 Remote Proxy, 자원 소모가 많은 객체를 생성하는데 사용하는 Virtual Proxy, 목표 객체에 대한 접근을 제어하는 Protection Proxy, 실제 객체 이전에 여러 가지 기능을 수행하는 Smar Reference가 있다. 이 중, Remote Proxy와 Smart Reference가 서비스 제공자와 서비스 소비자 측에 다음과 같이 사용된다.

[서비스 제공자 측] 가변성이 설계된 서비스는 여러 가변점을 포함하고 있으므로, 각 소비자가 해당 서비스를 사용하기 이전에 가변치를 설정해야 한다. 즉, 실제 서비스 기능을 호출하기 이전에 가변치 설정이 이루어져야 하며, 이는 Proxy 패턴의 적용 가능한 상황과 일치한다.

[서비스 소비자 측] 서비스 소비자는 목적에 맞게 동일한 기능에 대해 다양한 서비스를 구독할 수 있으며, 구독된 서비스는 현재 상황을 판단하여 적절한 서비스로 바인딩시켜야 한다. 즉, 서비스를 구독 및 호출하기 이전에 현재 상황을 판단하여 적절한 서비스를 바인딩하는 과정을 거쳐야 하며, 이는 Remote Proxy 역할과 유사하다.

4.3 행위 패턴에 대한 SOC 적용성

행위 패턴은 좀 더 복잡한 기능을 제공하기 위해 객체들이 어떻게 상호작용하는지에 대한 방법을 주로 다루며, 서비스 제공자 입장에서는 서비스의 특징을 만족시키기 위한 클래스 간 상호작용 설계 목적으로, 서비스 소비자는 서비스를 재사용한 목표 애플리케이션의 실시간 흐름을 설계하기 위해 주로 사용된다. 다음은 각 패턴에 대한 적용 가능성 유무와 그에 대한 이유이다.

- Chain of Responsibility 패턴: 서비스 제공자와 서비스 소비자 측에 모두 적용 가능함

Chain of Responsibility 패턴은 여러 객체들이 체인으로 구성되고, 체인으로 구성된 각 객체들은 자신이 처리할 수 있는 요청만 처리하고 그렇지 않은 요청은 다른 객체에게 전달하기 위해 사용된다.

[서비스 제공자 측] Chain of Responsibility 패턴의 적용 가능 상황은 서비스 진화에 따라 서비스 제공자가 여러 버전의 서비스 구현체를 가지고 있고, 서비스 소비자는 자신이 필요로 하는 버전의 기능을 사용하게 하는 것과 유사하다. 이로 이냉, 새로운 버전의 서비스 구현체가 쉽게 사용될 수 있다.

[서비스 소비자 측] 서비스 소비자는 자신의 요구사항에 맞게 제공되는 서비스 기능을 추가하거나 비활성해야 하는 경우가 있다. 즉, 서비스가 제공하는 기능을 서비스 자체를 수정하지 않고, 목적에 맞게 수정해야 하는 '기능 불일치'를 해결해야 하며, 이는 Chain of Responsibility 패턴이 해결할 수 있는 설계 문제와 유사하다.

- Command 패턴, Interpreter 패턴, Iterator 패턴: 서비스 제공자와 서비스 소비자 측에 모두 적용되지 않음

Command 패턴은 명령어 자체를 캡슐화하여 관리하는데 사용하며, Interpreter 패턴은 언어, 문장, 구문 등을 해석해야 하는 상황에 적용된다. 그리고, Iterator 패턴은 내부 구현에 상관없이 자료의 집합체를 접근할 때 적용된다. 이 세 패턴의 적용 가능한 상황은 SOC에서 발생하지 않으므로, 적용 가능한 패턴으로 선정하지 않는다.

- Mediator 패턴: 서비스 제공자 측에 적용 가능함

Mediator 패턴은 여러 참여객체들 간의 상호작용을 중재하여 각 참여객체의 결합도를 줄이기 위해 사용되며, 이는 SOC에서 복합 서비스에 포함되는 여러 단일 서비스들의 워크플로우를 제어하고자 하는 상황과 일치한다. Mediator 패턴을 사용하면 각 단일 서비스 간의 결합도를 줄이고, 상호작용 흐름을 다르게 함으로써 워크플로우 가변성을 가지는 서비스를 제공할 수 있다.

- Memento 패턴: 서비스 제공자 측에 적용 가능함

Memento 패턴은 객체의 내부 상태를 캡슐화 원칙을 위배하지 않고 노출시키기 위해 사용되며, 주로 undo/redo 기능을 구현하는데 적용된다. SOC에서 제공되는 SaaS는 멀티테넌시를 보장해야 하는데, 이를 위해서는 각 사용자 세션별 현재 상태를 분리하여 관리해야 한다. 이 상황은 Memento 패턴의 적용 가능한 상황과 일치한다.

- Observer 패턴: 서비스 제공자 측에 적용 가능함

Observer 패턴은 객체의 상태 변화가 있을 때마다 이를 관련 객체에 전달하여 내용을 자동으로 갱신하기 위해 사용되며, 이는 SOC에서 서비스 진화가 발생할 때마다 이를 구독하는 서비스 소비자들이 진화된 서비스를 갱신해야 하는 상황과 일치한다. 그러므로, 동적으로 진화된 서비스를 소비자에게 통지하기 위해서 이 패턴을 적용할 수 있다.

- State 패턴: 소비자 측에 적용 가능함

State 패턴은 객체의 상태 변화에 따라 다른 기능을 제공하기 위하여 사용되며, 이는 서비스 소비자의 현재 상태에 따라 적절한 서비스를 호출해야 하는 상황과 일치한다. 서비스 소비자는 동일한 기능을 위해 여러 서비스를 구독할

수 있으며, 각 서비스는 비슷하지만 다른 기능을 제공한다. 서비스 소비자는 상황에 맞게 적절한 서비스를 호출해야 하며, 이를 위해 State 패턴을 사용할 수 있다.

- Strategy 패턴과 Template Method 패턴: 서비스 제공자 측에 적용 가능함

Strategy 패턴과 Template Method 패턴은 하나의 기능이 여러 다른 알고리즘을 제공해야 하는 경우에 사용되며, 이는 각 서비스 소비자들의 다른 요구사항을 반영하여 다른 알고리즘을 기능을 수행하는 서비스의 로직 가변성의 상황과 일치한다. 동일한 기능에 대하여 다양한 로직 및 알고리즘을 구현하기 위해 이 두 가지 패턴을 적용할 수 있다.

- Visitor 패턴: 서비스 제공자와 서비스 소비자 측에 모두 적용되지 않음

Visitor 패턴은 데이터 부분과 기능 처리 부분을 분리하여, 복잡한 구조체를 구성하는 요소별로 기능을 처리하기 위해 사용된다. 이 패턴의 적용 가능한 상황은 SOC에서 발생하지 않으므로, 적용 가능한 패턴으로 선정하지 않는다.

4.4 요약

<표 3>은 SOC에서 적용 가능한 패턴을 추출한 결과를 보여준다. 표에서 기술된 내용은 SOC에 적용 가능한 경우에 해당하는 이유이며, 빈 칸은 적용되지 않은 패턴임을 나타낸다.

5. SOC에 디자인 패턴 적용 지침

도출된 디자인 패턴 중 각 유형별 대표적으로 사용되는 Abstract Factory 패턴, Adapter 패턴, Strategy 패턴에 대한 SOC에 특화된 명세를 정의한다.

〈표 3〉 SOC를 위한 DP 도출 결과

분류	디자인 패턴 이름	적용 여부	
		서비스 개발의 경우	서비스 기반 SW 개발의 경우
생성 패턴	Abstract Factory	복합 서비스에 포함되는 여러 단일 서비스를 생성하기 위해서 적용 가능함.	
	Builder	복합 서비스에 포함되는 여러 단일 서비스를 생성하기 위해서 적용 가능함. 이 때 단일 서비스 생성 프로세스이 재사용됨.	
	Factory Method	해당 기능 호출에 대해 적절한 서비스 구현체를 생성하여 이를 동적 바인딩하는데 적용 가능함.	
	Prototype		
	Singleton		
구조 패턴	Adapter	서비스 진화로 인해 서비스 구현이 서비스 인터페이스와 일치하지 않은 경우 적용 가능함.	서비스 인터페이스와 서비스 소비자 측에서 필요한 인터페이스 간의 불일치 발생하는 경우 적용 가능함.
	Bridge	잠재적 소비자의 알려져 있지 않은 서비스 구현, 즉 기능 가변치를 허용하는 경우를 가능하게 하기 위해 적용 가능함.	상황에 따라 동일한 기능에 대해 다른 구현체를 가지는 서비스를 바인딩하여 호출하는 경우를 가능하게 하기 위해 적용할 수 있음.
	Composite		
	Decorator	서비스가 제공하는 기능 외에 제한된 관리성을 보완하기 위한 모니터링 기능 등을 추가할 때 적용 가능함.	서비스가 제공하는 기능 외의 기능을 추가할 때 적용 가능함.
	Façade		
	Flyweight		
	Proxy	실제 서비스가 실행되기 이전에 서비스 소비자별 가변치를 결정하는 경우에 Smart Reference 적용 가능함.	상황에 따라 다양한 서비스를 동적으로 바인딩하여 호출하고자 하는 경우에 Remote Proxy가 적용 가능함.

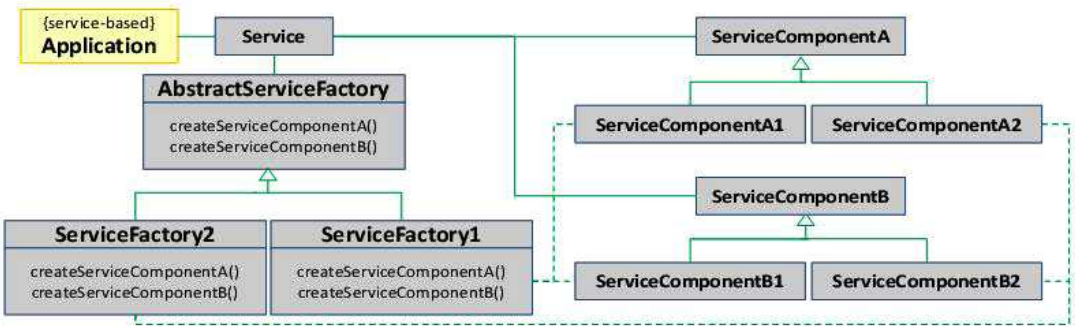
행위 패턴	Chain of Responsibility	서비스 진화에 따라 동일한 서비스에 대한 여러 버전을 제공하고자 할 때 적용 가능함.	서비스가 제공하는 기능과 소비자 요구사항이 불일치하여, 불필요한 기능을 비활성화 시키거나 필요한 기능을 추가하기 위한 방법으로 적용 가능함.
	Command		
	Interpreter		
	Iterator		
	Mediator	복합 서비스의 경우 다양한 워크플로우를 구현한 서비스를 제공하고자 할 때 적용 가능함.	
	Memento	SaaS 서비스의 경우 여러 사용자를 위한 Undo/redo 기능을 제공할 때 사용 가능함.	
	Observer	동적으로 진화된 서비스를 소비자에게 통지해야 하는 경우에 적용 가능함.	
	State		상태에 따라 다양한 서비스를 동적으로 바인딩하여 호출하고자 하는 경우에 적용 가능함.
	Strategy	동일한 기능에 대해 다양한 로직/알고리즘을 수행하는 로직 가변성을 실현하기 위해 적용 가능함.	
	Template Method	동일한 기능에 대해 다양한 로직/알고리즘을 수행하는 로직 가변성을 실현하기 위해 적용 가능함.	
	Visitor		

5.1 Abstract Factory 적용 지침

개요: Abstract Factory 패턴은 서비스 소비자가 호출하는 단일 서비스에 대한 서비스 구현체가 여러 개의 객체로 구성되거나, 복합 서비스에 대한 여러 단일 서비스를 구성해야 할 때 유용하다. 즉, 이 패턴은 생성되는 서비스 구현체의 집합이 가변점일 경우에 유용하다.

SOC에서의 적용 상황: 서비스 소비자의 의도, 디바이스 종류 또는 상태, 컨텍스트, 네트워크 대역폭 등에 따라 서비스 구현체의 구성요소에 가변성이 발생할 수 있다. 서비스 소비자의 가장 중요하게 생각하는 품질에 따라 서비스 구현체의 구성요소가 변할 수 있다. 만약 서비스 소비자가 신뢰성을 가장 중요하게 생각하면 성능이 다소 떨어지더라도 신뢰성이 좋은 구성요소로 이루어진 서비스를 사용하길 원할 것이다. 또한 서비스 소비자의 디바이스가 스마트폰일 경우 제공이 불가능한 서비스 구성요소가 있을 수 있거나 스마트폰만을 위한 서비스 구성요소가 있을 수 있다. 그리고, 서비스 소비자가 사용하는 네트워크의 대역폭에 따라 서비스 구성요소가 변할 수 있다. 즉, 네트워크 대역폭이 작은 경우 서비스는 경량화된 구성요소를 지는 것이 좋다. 이러한 여러 상황에서 Abstract Factory 패턴을 적용함으로써 서비스 소비자에 따라 적절한 구성요소를 지니는 서비스를 제공 할 수 있다.

정적 구조: SOC에서의 Abstract Factory 패턴의 구조는 (그림 3)과 같다.



(그림 3) SOC에서의 Abstract Factory 패턴

SOC에서의 Abstract Factory 패턴의 주요 요소와 그 요소들의 명세를 <표 4>에서 정리한다.

〈표 4〉 SOC를 위한 Abstract Factory 패턴의 주요 요소

주요 요소	명세
Application	특정 기능성을 제공하는 서비스를 사용하는 서비스 기반 애플리케이션이다.
Service	서비스 기반 애플리케이션이 사용하는 단일 또는 복합 서비스이다.
Abstract Service Factory	서비스 구성요소를 생성하기 위한 인터페이스를 제공하는 클래스이다. 서비스가 단일 서비스일 경우 클래스가 구성요소가 될 수 있고, 복합 서비스일 경우 다른 서비스가 구성요소가 될 수 있다.
Serivce Factory#	*AbstractServiceFactory*의 인터페이스를 실현하는 클래스이다. #에 따라 생성하는 구성요소가 다르다.
Service ComponentA, Service ComponentB	서비스 구성요소의 인터페이스이다.
Service ComponentA#, Service ComponentB#	*ServiceComponentA* 또는 *Service ComponentB*의 인터페이스를 실현하는 클래스이다. #에 따라 구현체가 다르다.

동적 구조: 서비스 기반 애플리케이션이 서비스를 호출한다. 요청된 호출에 대해 적절한 서비스는 여러 개의 객체로 이루어져 구현되어 있으므로, 이 요청은 Abstract ServiceFactory로 전달되어 서비스 소비자의 요구사항 또는 디바이스 종류 또는 상태, 컨텍스트 등에 맞게 적절한 서비스 구현체들을 생성한다. 예를 들어, 현재 소비자를 고려하여 ServiceFactory2가 실행이 되면, ServiceComponentA2와 ServiceComponentB2로 구성된 서비스 구현체들이 생성되어 기능을 제공하게 된다.

5.2 Adapter 패턴 적용 지침

개요: Adapter 패턴은 재사용하는 주체의 인터페이스를 재사용할 대상의 인터페이스로 변환할 때 사용한다. 즉, 이 패턴은 재사용 대상의 인터페이스와 재사용하는 주체의 인터페이스가 불일치할 때 유용하다.

SOC에서의 적용 상황: SOC에서의 적용 상황을 두 가지 관점인 서비스 소비자에 적용되는 상황과 서비스 제공자에 적용되는 상황으로 나누어 설명한다.

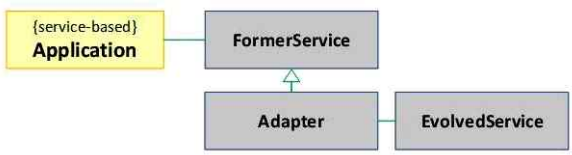
『다른 요구사항에서 발생하는 서비스 인터페이스 가변성』: 서비스 제공자는 공통적인 기능성을 인터페이스와 함께 제공하지만 서비스 소비자가 그 공통적 기능의 서비스를 사용 할 때 요구하는 인터페이스가 다를 수 있다. 즉, 서비스 인터페이스에 가변성이 존재 할 수 있고 이는 서비스 제공자와 서비스 소비자 간의 인터페이스 불일치가 발생한다. 이 때, 서비스 소비자는 Adapter 패턴을 적용하여 서비스 인터페이스와 불일치하는 자신의 인터페이스를 서비스 인터페이스와 일치되도록 변환하여 서비스를 제공 받을 수 있다.

『서비스의 진화로부터 발생하는 서비스 인터페이스 가변성』: 서비스는 기능을 추가하거나 수정함으로써 자주 발전한다. 이 두 경우 중 기능을 수정함으로써 이전에 존재했던 인터페이스가 변경될 수 있다. 이 경우 서비스 소비자는 다시금 그 변경된 서비스의 인터페이스에 일치되도록 그들의 인터페이스를 수정해야 한다. 대개 서비스는 많은 수의 서비스 소비자들에게 제공되고 있으므로 전체적으로 봤을 때 개개의 서비스 소비자 인터페이스 수정은 상당한 비용을 초래한다. 이러한 상황을 피하기 위해 서비스 제공자는 수정 전의 인터페이스와 수정 후의 인터페이스를 Adapter 패턴을 적용함으로써 적응시킬 수 있다. (그림 4)와 같이, 인터페이스를 변경하면, 서비스 소비자들은 서비스 진화 여부를 고려하지 않아도 된다.

(그림 4) SOC에서 Adapter 패턴이 적용 가능한 상황

네 개의 애플리케이션이 있고 그들은 인터페이스 A를 통해 서비스를 사용하고 있다. 이 서비스는 진화로 인해 인터페이스 A가 인터페이스 A'로 변경 되었다. 이 경우에 서비스 제공자는 애플리케이션 개개의 수정을 피하기 위해 Adapter 패턴을 적용하여 인터페이스 A를 인터페이스 A'로 맞추어 줄 수 있다.

정적 구조: 서비스 제공자를 위한 Adapter 패턴의 구조는 (그림 5)와 같다. 서비스 기반 애플리케이션은 서비스 진화로 인해 서비스 제공자가 더이상 제공하지 않는 수정전 서비스를 사용하고 있다. 그리고 *Adapter*는 수정전 서비스가 받는 요청을 수정된 서비스로 넘겨 줌으로써 서비스와 애플리케이션 간의 인터페이스 불일치를 해결한다.



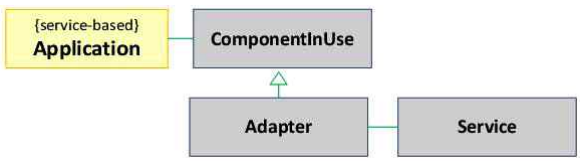
(그림 5) 서비스 제공자를 위한 Adapter 패턴

Adapter 패턴의 주요 요소와 그 요소들의 명세를 <표 5>에서 정리한다.

〈표 5〉 서비스 제공자를 위한 Adapter 패턴의 주요 요소

주요 요소	명세
Application	특정 기능성을 제공하는 서비스를 사용하는 서비스 기반 애플리케이션이다.
Former Service	서비스 진화 이전의 서비스이다. *Application*이 특정 기능성을 요청하는 대상이다.
Evolved Service	서비스 진화 이후의 서비스이다. *Adapter*에 의해서 인터페이스 불일치를 해결 후에 요청된다.
Adapter	*FormerService*의 인터페이스와 *Evolved Service*의 인터페이스 간의 불일치를 해결하기 위한 구현체다. 즉, *FormerService*의 인터페이스가 *Adapter*에 의해 *EvolvedService*의 인터페이스로 변환된다.

서비스 소비자를 위한 SOC에서의 Adapter 패턴 구조는 (그림 6)과 같다. 서비스 기반 애플리케이션인 *Application*은 특정 기능성을 제공하는 컴포넌트인 *ComponentInUse*를 사용하고 있다. 이 컴포넌트의 기능성을 서비스의 기능성으로 대체하여 사용하려 할 때, 서비스 소비자는 *Adapter*를 적용하여 컴포넌트의 인터페이스를 서비스의 인터페이스로 변환할 수 있다. *Adapter*는 이 변환을 위한 구현체다. 이 때, 공통적 부분은 기능성이고 가변적 부분은 인터페이스이다.



(그림 6) 서비스 소비자를 위한 Adapter 패턴

Adapter 패턴의 주요 요소와 그 요소들의 명세를 <표 6>에서 정리한다.

〈표 6〉 서비스 소비자를 위한 Adapter 패턴의 주요 요소

주요 요소	명세
Application	특정 기능성을 제공하는 서비스를 사용하는 서비스 기반 애플리케이션이다.
Component InUse	애플리케이션에서 사용되었던 특정 기능성을 제공하는 컴포넌트이다.
Service	*ComponentInUse*를 대체할 수 있는 서비스이다. 이 서비스는 인터페이스 불일치를 해결한 Adapter로 부터 호출 된다.
Adapter	*ComponentInUse*의 인터페이스를 서비스의 인터페이스로 변환하기 위한 구현체이다.

동적 구조: 애플리케이션이 특정 컴포넌트 혹은 서비스의 인터페이스를 통해 ComponentInUse에서 제공되는 기능을 사용하고 있다. 해당 컴포넌트가 진화되어 새로운 서비스 인터페이스로 제공되면, *Adapter*가 그 컴포넌트 혹은 서비스로의 요청을 새로운 서비스 혹은 진화한 서비스 (Service)로 그 요청을 넘긴다. 그 컴포넌트와 서비스로의 인터페이스가 새로운 서비스와 진화한 서비스의 인터페이스로 각각 변환되고, 진화된 서비스를 호출할 수 있게 된다.

5.3 Strategy 패턴 적용 지침

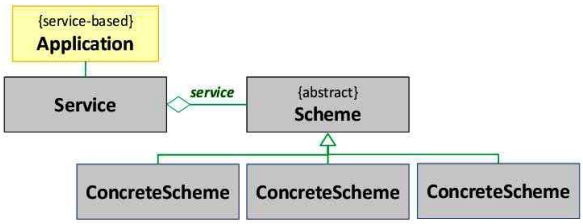
개요: Strategy 패턴은 알고리즘 집단을 관리하고 그 알고리즘들을 실행 중에 상호 교환 가능하게 만든다. 이 패턴은 어떠한 기능성을 위해 많은 수의 구현체들이 존재 할 때 이들을 관리하는데 유용하다.

SOC에서 적용 가능한 상황: Strategy 패턴이 SOC에서 적용되는 두 가지 상황은 다음과 같다.

『알고리즘 가변성이 존재하는 서비스』: 서비스가 서비스 소비자에 따라 알고리즘에 가변성이 존재 할 경우 각 알고리즘을 Strategy 요소로 구현 할 수 있다. 서로 다른 알고리즘의 개수는 Strategy 요소의 개수가 되며, 실행 중에 가변적인 Strategy 요소가 서비스에 적용된다.

『컨텍스트로부터 발생하는 가변성이 존재하는 서비스』: 서비스를 호출하는 애플리케이션은 호출 당시의 위치, 시간대, 디바이스와 사용자의 상태, 서비스의 품질 등의 컨텍스트가 서로 다르다. 그러므로 컨텍스트에 따라 호출 당시의 상황이 추론 될 수 있다 [24]. 그리고 특정 상황에 따라 서비스는 실행시에 알고리즘을 결정 할 수 있다. 상황에 따라 적절한 알고리즘을 선택하는 것의 장점은 현재 상황에 따라 서비스를 개인화 시킬 수 있다는 것이다.

정적 구조: SOC에서 Strategy 패턴의 구조는 (그림 7)과 같다. 여기서, 공통적 부분은 서비스, 즉 공통적 기능성이며 가변적 부분은 Scheme, 즉 공통적 기능성을 위한 알고리즘이다.



(그림 7) SOC에서의 Strategy 패턴

Strategy 패턴의 주요 요소와 그 요소들의 명세를 <표 7>에서 정리한다.

동적 구조: 애플리케이션은 공통적 기능성을 위해 서비스를 호출한다. 그리고 서비스는 애플리케이션에 따라 *Scheme*을 특정 *ConcreteScheme*에 참조시킨다. 이후 서비스는 *Scheme*의 메소드를 실행하며 동적 결합에 의해 실제로는 특정 알고리즘을 실현하고 있는 *ConcreteScheme*의 메

〈표 7〉 SOC를 위한 Strategy Adapter 패턴의 주요 요소

주요 요소	명세
Application	특정 기능성을 제공하는 서비스를 사용하는 서비스 기반 애플리케이션이다.
Service	여러 애플리케이션에게 공통적 기능성을 제공하기 위한 서비스를 의미한다.
Scheme	서비스가 제공하는 기능성을 의미한다.
Concrete Scheme	*Scheme*의 구현체를 의미하며 특정 알고리즘을 실현한다. 이 요소는 알고리즘 가변점의 가변치들을 제공한다.

소드가 실행된다. 이는 애플리케이션에 따라 호출되는 메소드가 달라 질 수 있음을 의미한다. 즉, 서비스는 애플리케이션에 따라 다른 알고리즘을 제공한다.

애플리케이션의 컨텍스트가 변하게 되면 그 애플리케이션의 호출 당시의 상황이 변하게 된다. 이 때, 특정 컨텍스트 따라 특정 *ConcreteScheme*이 동적 결합되며 서비스는 애플리케이션에게 참조하고 있는 *ConcreteScheme*의 기능성을 제공한다. 즉, 컨텍스트에 따라 서비스가 다른 알고리즘을 제공 할 수 있다.

6. SOC에 디자인 패턴 적용 사례

본 장에서 5장에서 제시한 적용 지침을 기반으로 사례 연구를 수행한 결과를 보여준다.

6.1 생성 패턴의 적용 사례: Abstract Factory 패턴

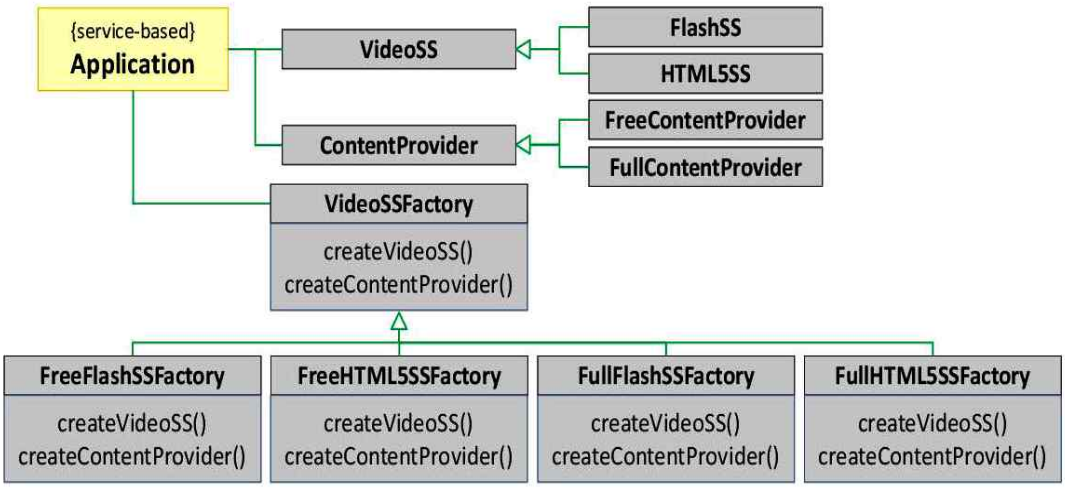
상황: *Video Streamming Service(VideoSS)*는 Flash 또는 HTML5 기반으로 다양한 콘텐츠를 제공해 주는 비디오 스트리밍 서비스이다. 서비스 소비자에 따라 Flash 기반의 무료 또는 유료 비디오 스트리밍 서비스, HTML5 기반의 무료 또는 유료 비디오 스트리밍 서비스로 네 가지 서비스 형태가 있다. 이 때, Abstract Factory 패턴을 적용하여 이 네 가지 서비스 형태들 중 서비스 소비자에게 적합한 서비스 생성하여 제공할 수 있다.

정적 구조: 서비스 제공자는 (그림 8)과 같이 Abstract Factory 패턴을 적용하여 여러가지 형태의 서비스 중에서 서비스 소비자에게 적절한 서비스를 제공할 수 있다. *VideoSS*는 비디오 스트리밍 서비스를 위한 인터페이스를 제공하며 이 인터페이스를 실현화하여 *FlashSS*가 Flash 기반의 서비스를, *HTML5SS*가 HTML5 기반의 서비스를 제공한다. *ContentProvider*는 콘텐츠를 제공하기 위한 인터페이스이며 이 인터페이스를 실현화하여 *FreeContentProvider*가 무료의 제한된 콘텐츠를, *FullContentProvider*가 모든 콘텐츠를 제공한다. 또한 *VideoSSFactory*가 이들 서비스를 생성하기 위한 인터페이스를 제공한다.

스트리밍 서비스와 콘텐츠 제공자의 조합으로 Flash 기반의 무료 비디오 스트리밍 서비스 (*FreeFlashSSFactory*가 생성), HTML5 기반의 무료 비디오 스트리밍 서비스 (*FreeHTML5SSFactory*가 생성), Flash 기반의 유료 비디오 스트리밍 서비스 (*FullFlashSSFactory*가 생성), HTML5 기반의 유료 스트리밍 서비스 (*FullHTML5SSFactory*가 생성)로 네 가지 서비스 형태가 존재한다.

동적 구조: 서비스 소비자가 Flash 또는 HTML5 기반의 서비스를 요청하면 이 소비자가 무료 사용자인지 유료 사용자인지와 어떤 기반의 서비스를 요청했는지를 판단하여 네 가지 형태의 서비스, *FreeFlashSSFactory, FreeHTML5SSFactory, FullFlashSSFactory, FullHTML5SSFactory* 중 하나로 서비스를 생성하여 서비스 소비자에게 제공한다.

결과: Abstract Factory 패턴이 적용된 서비스를 사용하는 서비스 소비자는 자신이 무료 사용자인지 유료 사용자인지에 대한 고려할 필요 없이 서비스를 호출하여 절절한 서비스를 제공 받을 수 있다. 그리고, 서비스 제공자가 또 다른 서비스 형태를 추가하여도 기존의 서비스에는 영향을 주지 않아 확장성이 좋다.

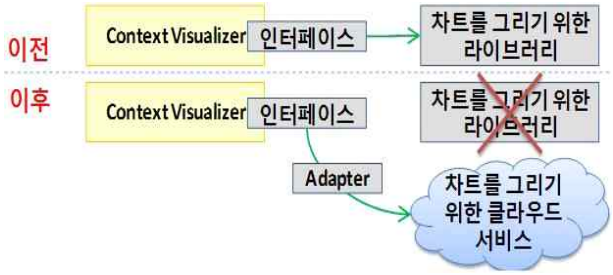


(그림 8) 여러 형태의 서비스를 제공하기 위한 Abstract Factory 패턴을 적용한 클래스 다이어그램

6.2 구조 패턴의 적용 사례: Adapter 패턴

6.2.1 클라우드 서비스의 인터페이스 적응

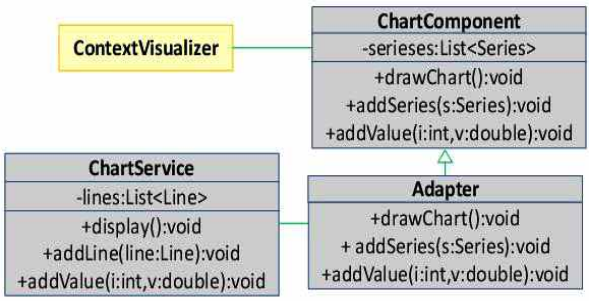
상황: *Context Visualizer*는 모바일 컨텍스트의 값을 측정하고 이를 여러 형태로 가시화 하는 도구이다. 이 도구는 차트를 그리기 위한 오픈소스 라이브러리를 사용하고 있다. 이 때, 이 도구의 차트를 그리는 기능을 클라우드 서비스로 대체하고자 한다. 그러나 이 도구가 이미 사용하고 있는 오픈소스 라이브러리의 인터페이스와 클라우드 서비스의 인터페이스는 (그림 9)에서 처럼 서로 일치하지 않는다.



(그림 9) 애플리케이션과 서비스 간의 인터페이스 불일치 상황

이 불일치를 간편하게 해결하기 위해 Adapter 패턴을 적용 할 수 있다.

정적 구조: 서비스 소비자를 위한 Adapter 패턴을 이용하여 서비스 소비자는 (그림 10)과 같은 클래스 다이어그램을 유도한다. ChartComponent는 이 도구가 이전에 사용하던 차트를 그리기 위한 컴포넌트이며 ChartService는 이 도구가 앞으로 사용할 차트를 그리기 위한 서비스이다.



(그림 10) 서비스 소비자를 위한 Adapter 패턴 적용한 클래스 다이어그램

Adapter는 <표 8>과 같이 ChartComponent의 하위 클래스이며 ChartComponent의 메소드를 오버라이드 함으로써 ChartService와의 인터페이스 불일치를 해결한다.

21줄에서 차트를 그리기 위한 서비스를 호출하기 위해 'chartService' 에 서비스 인스턴스를 할당한다. 이후 37줄과 43줄에서 인터페이스 불일치를 해결하기 위해 차트를 그리기 위한 컴포넌트의 drawChart()와 addSeries()을 각각 차트를 그리기 위한 서비스의 display()와 addLine()으로 변환한다. 이 때, addSeries에서 addLine()으로의 변환은 메소드의 이름 뿐만 아니라 메소드의 인자 또한 변환한다.

<표 8> 인터페이스 적응을 위한 Adapter 클래스의 코드 일부

```
1. public class Adapter extends ChartComponent {
2.       …
13.      public Adapter() {
14.        …
21.        chartService = new ChartService();
22.      }
23.      public void drawChart() {
26.        …
37. chartService.display(); // ChartComponent의
 drawChart()를 ChartService의 display()로 변환
38.      }
39.      public void addValue(int i, double v) {
40.         chartService.addValue(i, v);
41.      }
42.      public void addSeries(Series s) {
43.           chartService.addLine(new Line (s.get
          Name()));
    // ChartComponent의 addSeries()를 ChartService의
    addLine()으로 변환
44.    }
45. }
```

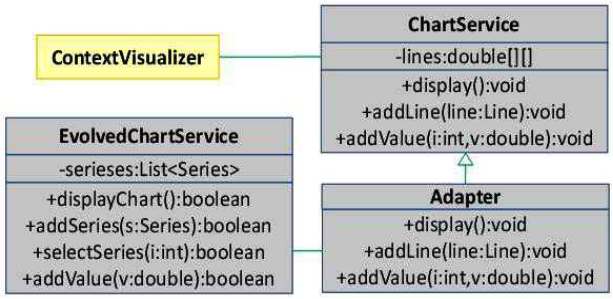
동적 구조: *ContextVisualizer*는 차트를 그릴 것을 *Adapter*에게 요청한다. 그리고 *Adapter*는 그 요청을 인터페이스 불일치를 해결하며 *ChartService*에게 재요청한다.

결과: *ContextVisualizer*에서 *Adapter*에 의해 최소한의 수정으로 *ChartComponent*가 *ChartService*로 대체된다. *ChartComponent* 대신에 서비스를 호출하기 위해 해야하는 수정작업은 *Adapter* 구축과 *ChartComponent*의 인스턴스를 *Adapter*의 인스턴스로 교체하는 것이다.

6.2.2 서비스 버전 제어를 위한 인터페이스 적응

상황: *Context Visualizer*는 차트 서비스를 사용하고 있고 그 서비스가 진화하여, 기존의 서비스 인터페이스와 진화 이후의 서비스 인터페이스에 변화가 생겼다. 이 때, 서비스 제공자가 Adapter 패턴을 적용하여 서비스 소비자들이 애플리케이션의 수정 없이 계속 서비스를 사용하게 할 수 있다.

정적 구조: 서비스 제공자를 위한 Adapter 패턴의 명세에 의해 (그림 11)과 같은 클래스 다이어그램을 도출한다.



(그림 11) 서비스 제공자를 위한 Adapter 패턴 적용한 클래스 다이어그램

Adapter는 *ChartService*의 하위 클래스이며 *Evolved ChartService*의 인스턴스를 가지고 있다. *Adapter*는 오버라이드된 *ChartService*의 메소드에서 해당 메소드의 기능을 수행하는 *EvolvedChartService*의 메소드를 호출하여 *ChartService*와 *EvolvedChartService*간의 인터페이스 불일치를 <표 9>와 같이 해결한다. 32줄와 같이, *ChartService*의 메소드인 addValue()는 *EvolvedChartService*의 selectSeries()와 addValue()에 일치된다.

〈표 9〉 버전 제어를 위해 적용한 Adapter 클래스의 코드 일부

```
1.  public class Adapter extends ChartService {
2.     private EvolvedChartService evolvedChartService =
        new EvolvedChartService();
32.   addValue(int i, double v) {
      // ChartService의 메소드를 오버라이드 함.
33.       evolvedChartService.selectSeries(i);
          // EvolvedChartService의 인스턴스로 addValue와
34.       evolvedChartService.addValue(v);
          // 일치하는 기능을 수행함.
35.   }
36.    …
48.  }
```

동적 구조: *Context Visualizer*는 차트를 그리기 위한 서비스인 *ChartService*를 호출하고 Adapter가 그 호출을 진화된 차트 서비스인 *EvolvedChartService*로 인터페이스 불일치를 해결하여 넘긴다.

결과: 서비스 제공자가 Adapter 패턴을 적용 함으로써 서비스 소비자는 자신의 애플리케이션을 전혀 수정하지 않고 최신의 서비스를 호출 할 수 있다. 그리고 서비스 제공자는 서비스 소비자들에게 높은 유지보수성과 신뢰성을 확보하며 자신의 서비스를 제공 할 수 있다.

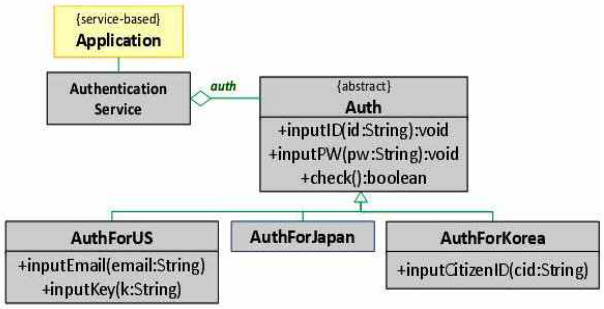
6.3 행위 패턴의 적용 사례: Strategy 패턴

상황: *Authentication Service*는 다국적 사용자의 기능성 사용 권한을 검증하는 인증 서비스다. 인증 절차가 국가별로 정책에 따라 서로 다를 수 있다. 이는 가변성이 인증 절차에 있음을 의미한다. 이 때, 하나의 인터페이스로 서비스를 효과적으로 제공하기 위해 Strategy 패턴을 적용 할 수 있다.

정적 구조: Strategy 패턴을 적용 함으로써 (그림 12)와 같은 클래스 다이어그램을 도출하였다. 세 인증 방법, *AuthForUS, AuthForJapan, AuthForKorea*가 각각 미국, 일본, 한국의 인증 절차를 위해 존재한다.

*Auth*는 인증 서비스를 위한 추상 클래스이며 *Auth ForUS, AuthForJapan, AuthForKorea*가 이를 상속하여 서로 다른 인증 절차 알고리즘의 구현체가 된다.

동적 구조: 애플리케이션이 인증 서비스를 호출하면 애플리케이션이 어느 국가에서 호출하고 있는지 판단한 후



(그림 12) 서비스 소비자를 위해 다른 인증 요청을 처리하기 위한 구조

적절한 *Auth*의 하위 클래스를 동적으로 인증 서비스에 결합한다.

결과: 다국적 서비스 소비자들은 그들의 국가 정책을 신경쓰지 않고 동일한 인터페이스로 인증 서비스를 사용하여 그들의 국가 정책에 맞는 인증 절차를 제공 받을 수 있다. 그리고 서비스 제공자는 여러 인증 절차를 한 인터페이스를 통해 제공할 수 있다. 그리고 서비스 제공자는 인증 서비스에 새로운 인증 절차를 많은 수정 없이 쉽게 추가 할 수 있다.

7. 결 론

SOC의 기본 단위인 서비스는 다양한 서비스 소비자들에 의해 재사용되는 필수적인 기능 단위이므로, 가능한 많은 소비자들이 재사용할 수 있도록 공통적인 기능을 최대한 제공해야 하며, 서비스가 자체적으로 가지는 느슨한 결합, 제한된 관리성 등의 고유한 특징을 반영하도록 설계되어야 한다. 그리고 서비스 소비자는 자신의 목적에 맞게 서비스의 일부를 수정하여 목표 애플리케이션을 개발한다면 서비스 재사용성 이점을 이용하게 된다. 디자인 패턴 (Design Patterns)는 소프트웨어 설계 시에 자주 발생하는 문제들을 해결하기 위한 범용적이며 재사용 가능한 방법들이며, 가변성 및 여러 설계 이슈를 보다 쉽게 처리할 수 있는 설계 구조를 제안한다. 기존 객체와는 구별되는 SOC 서비스의 고유한 특징을 반영하지 않았고, 재사용 관점에서 서비스 소비자와 서비스 제공자의 구별된 역할을 고려하지 않았으므로, 기존의 디자인 패턴들을 SOC 서비스에 그대로 적용하는 것은 어렵다.

본 논문에서는 서비스 제공자가 재사용성을 포함한 서비스 고유한 특징을 반영한 서비스를 설계하고, 서비스 소비자는 제공되는 서비스를 목적에 맞게 특화하여 목표 애플리케이션을 개발하기 위해, SOC의 특성을 고려하여 특화된 디자인 패턴을 제안하였다. 먼저, 3장에서 객체 지향 패러다임과 SOC의 차이점을 구성 단위 관점과 프로세스 관점에서 비교하여, 논문의 동기를 확고히 하고, SOC에 맞는 디자인 패턴을 수정시에 고려해야 하는 기준을 정의하였다. 4장에서는 분석된 차이점을 기반으로 SOC에서 적용 가능한 디자

인 패턴을 4장에서 도출하여, 3개의 생성 패턴, 4개의 구조 패턴, 7개의 행위 패턴이 SOC에 적용될 수 있음을 보였다. 그리고, 도출된 디자인 패턴 중 각 유형별 대표적으로 사용되는 Abstract Factory 패턴, Adapter 패턴, Strategy 패턴에 대한 SOC에 특화된 명세를 정의하고, 이에 대한 사례 연구를 6장에서 수행하였다.

SOC의 목적에 맞게 수정된 디자인 패턴을 이용하면, 서비스 제공자는 높은 재사용성을 비롯한 서비스 고유한 특징을 모두 반영하는 서비스를 효과적으로 설계할 수 있게 되고, 서비스 소비자는 효율적으로 서비스를 이용하여 목표 애플리케이션을 개발할 수 있게 된다.

참 고 문 헌

[1] Erl, T., Service-Oriented Architecture (SOA): Concepts, Technology, and Design, Prentice Hall, 2005.

[2] Gillett, F.E., "Future View: New Tech Ecosystems of Cloud, Cloud Services, and Cloud Computing," Forrester Research Paper, 2008.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995.

[4] Mell, P. and Grance, T.., "The NIST Definition of Cloud Computing (Draft): Recommendations of the National Institute of Standards and Technology," *National Institute of Standards and Technology Special Publication 800-145 (Draft)*, January, 2011.

[5] McNatt, W.B. and Bieman, J.M., "Coupling of design patterns: common practices and their benefits," *Proc. 25th Annual Conference on Computer Software and Applications (COMPSAC 2001)*, IEEE, pp. 574-579, Oct., 2001.

[6] Erl, T., *SOA Design Patterns*, Prentice Hall, 2009.

[7] Zdun, U., Hentrich, C., and van der Aalst, W.M.P., "A Survey of Patterns for Service-Oriented Architecture," *International Journal of Internet Protocol Technology,* Vol.1, No.3, pp.132-143, 2006.

[8] Topaloglu, N.U. and Capilla, R., "Modeling the Variability of Web Services from a Pattern Point of View," *In Proceedings of the 2004 European Conference on Web Services (ECOWS 2004), Lecture Notes in Computer Science 3250*, Springer-Verlag Berlin, pp.128-138, September, 2004.

[9] Milanovic, N., "Service Engineering Design Patterns," *Proc. Second IEEE International Symposium on Service-Oriented System Engineering (SOSE 2006)*, IEEE, pp.19-26, May, 2006.

[10] Mauro, C., Leimeister, J.M., and Krcmar, H., "Service Oriented Device Integration - An Analysis of SOA Design Patterns," *Proc. 43rd Hawaii International Conference on System Sciences (HICSS 2010)*, IEEE, pp.1-10, Jan., 2010.

[11] Thu, T.D. and Tran, H.T.B., "Composite Design Patterns to Integrate Available Services," *Proc. 2008 IEEE International Symposium on Service-Oriented System Engineering (SOSE 2008)*, IEEE, pp.19-24, Dec., 2008.

[12] Arcelli, F., Tosi, C., and Zanoni, M., "Can Design Pattern Detection be Useful for Legacy System Migration towards SOA?" *Proc. 2nd international workshop on Systems development in SOA environments (SDSOA 2008) in conjunction with ICSE 2008*, ACM, pp.63-68, May, 2008.

[13] Kim, S.D., Software Reusability. *In*: Wah, B.W. (Eds.), *Wiley Encyclopedia of Computer Science and Engineering*, Vol.4. Wiley-Interscience, pp.2679-2689, 2009.

[14] Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., and Holley, K., "SOMA: A method for developing service-oriented solutions," *IBM Systems Journal*, Vol.47, No.3, pp.377-396, 2008.

[15] Erradi, A., Kulkarni, N., and Maheshwari, P., "Service Design Process for Reusable Services: Financial Services Case Study," *Proc. 5th International Conference on Service-Oriented Computing (ICSOC 2007), Lecture Notes in Computer Science 4749*, Springer-Verlag Berlin, pp.606-617, Sep., 2007.

[16] Papazoglou, M.P. and van den Heuvel, W.J., "Service-Oriented Design and Development Methodology," *International Journal of Web Engineering and Technology (IJWET)*, Vol.2, No.4, pp 412-442, 2006.

[17] Chang, S.H. and Kim, S.D., "A Service-Oriented Analysis and Design Approach to Developing Adaptable Services," *Proc. IEEE International Conference on Services Computing (SCC 2007, IEEE)*, pp.713-714, July, 2007.

[18] La, H.J., Her, J.S., Oh, S.H., and Kim, S.D., "A Practical Approach to Developing Applications with Reusable Services," *Studies in Computational Intelligence*, Vol.253, pp.95-106, December, 2009.

[19] Kim, S.D., Her, J.S., and Chang, S.H., "A Theoretical Foundation of Variability in Component-based Development," *Information and Software Technology (IST)*, Vol.47, No.10, pp.663-673, July, 2005.

[20] Jegadeesan, H. abd Balasubramaniam, S., "A Method to Support Variability of Enterprise Services on the Cloud," *In Proceedings of 2009 IEEE International Conference on Cloud Computing (CLOUD 2009)*, IEEE, pp.117-124, September, 2009.

[21] Chang, S.D. and Kim, S.D., "A Variability Modeling Method for Adaptable Services in Service-Oriented Computing," *In Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, IEEE, pp.261-268, September, 2007.

[22] Kim, Y.Y. and Doh, K.G., "Adaptable Web Services Modeling using Variability Analysis," *In Proceedings of the 2008 Third International Conference on Convergence and Hybrid Information Technology (ICCIT 2008)*, Vol. 1, IEEE, pp.700-705, December, 2008.

[23] La, H.J. and Kim, S.D., "Static and dynamic adaptations for service-based systems," *Information and Software Technology*, Vol.53, pp.1275-1296, 2011.

[24] Sheng, Q.Z., Benatallah, B., and Maamar, Z., "User-Centric Services Provisioning in Wireless Environments," *Communications of the ACM*, Vol.51, No.11, pp.130-135, 2008.

김 문 권

e-mail : mkdmkk@gmail.com

2012년 숭실대학교 컴퓨터학부(공학사)

2012년～현 재 숭실대학교 컴퓨터학과 석사과정

관심분야 : 모바일 컴퓨팅(Mobile Computing), 컨텍스트 인지 컴퓨팅(Context-Aware Computing), 유비쿼터스 컴퓨팅(Ubiquitous Computing)

라 현 정

e-mail : hjla80@gmail.com

2003년 경희대학교 우주과학과(이학사)

2006년 숭실대학교 컴퓨터학과(공학석사)

2011년 숭실대학교 컴퓨터학과(공학박사)

2011년～현 재 숭실대학교 모바일 서비스 소프트웨어공학센터 연구교수

관심분야 : 서비스 지향 컴퓨팅(Service Oriented Computing), 클라우드 컴퓨팅(Cloud Computing), 모바일 서비스(Mobile Service)

김 수 동

e-mail : sdkim777@gmail.com

1984년 Northeast Missouri State University 전산학(학사)

1988년/1991년 The University of Iowa 전산학(석사/박사)

1991년～1993년 한국통신 연구개발단 선임연구원

1994년～1995년 현대전자 소프트웨어연구소 책임연구원

1995년 9월～현 재 숭실대학교 컴퓨터학부 교수

관심분야 : 서비스 지향 아키텍처(SOA), 클라우드 컴퓨팅(Cloud Computing), 모바일 서비스(Mobile Service), 객체지향 S/W공학, 컴포넌트 기반 개발 (CBD), 소프트웨어 아키텍처(Software Architecture)