컴포넌트 기반의 레거시프로그램 통합을 위한 Adapter와 Facade 패턴의 적용기법

이호성°
고려대학교 소프트웨어공학과
hslee0915°@korea.ac.kr

Techniques of Adapter and Facade design pattern for synthesis of legacy program in Component Base Development

Hosung Lee°
Dept. of Software Engineering, Korea University

요 약

컴포넌트기반 개발에 있어서 레거시프로그램의 재활용은 사업 기간과 범위 그리고 효율성이라는 부분에서 상당히 중요한 요소이다. 레거시프로그램을 재활용하기 위해 레거시 프로그램을 Wrapping하는 프로그램이 필요하며, 이를 위한 연구들이 활발히 진행되고 있다. 본 논문에서는 그 중 Adapter 패턴을 분석하여 상속과 위임의 장단점을 제시하고 상속 방식을 이용하는 패턴에 대하여 심층적으로 분석한다. 이를 바탕으로 레거시 프로그램을 통합함에 있어 개별적 Wrapper 구성을 통해 Adapter의 크기를 최소화하고 유지보수에 편리하도록 지역화 하며, 단점인 인터페이스의 복잡성을 해결하기 위해 Facade 패턴을 활용하여 문제를 해결하는 방법을 제안한다.

1. 서 론

응용프로그램 개발에 있어 통합을 목적으로 프로그램을 개발하는 것은 힘든 일이다. 따라서 각각의 독립된 응용프로그램을 구축하고 이를 모으는 방향으로 응용프로그램 개발이 진행되고 있다. 그러나 이러한 작업은 비즈니스 환경의 변화에 따라 수정하고자 할 경우 상당한 어려움을 겪게 한다. 컴포넌트 기반 개발은 이러한 환경을 개선하기 위한 대안으로써 제안되고 있다.

컴포넌트 기반 개발에서 초기에 코드를 재사용하거나 이미 만들어진 컴포넌트(예를 들어, 상용 컴포넌트 구매)를 사용하여 관련된 기능을 하나로 묶음으로써, 요구사항 변경을 지역화하고 유지보수 문제를 쉽게 처리할 수 있다[1]. 그러나 이러한 재사용을 위해 제안된 기존의 방법론들은 새로운 시스템의 재사용성 확보에 치중할 뿐, 실질적으로 기존의 레거시 시스템의 자원을 활용하여 새로운 시스템으로 재개발하는 방법에 대한 기법을 제시하는 이론이나 재사용 가능한 컴포넌트를 추출하는 방법, 유사성 있는 컴포넌트를 통합하는 방법 등은 명확히 제시하지 못하고 있다[2].

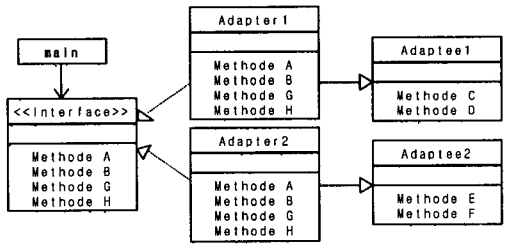
컴포넌트 기반 프로그램에서 과거프로그램에 대한 wrapping의 방식으로 많은 디자인 패턴들이 활용되고 있으며, 그 중 대표적인 방식 중의 하나가 Adapter를 이용하여 기존 클래스를 수정하는 방식이다. Adapter의 사용은 클래스에 의한 Adapter 패턴의 클래스(상속을 사용하는 방법)와 인스턴스에 의한 Adapter 패턴 클래스(위임을 사용하는 방법)가 있다. 상속을 사용하는 방법은 인터페이스를 사용하며 위임을 사용하는 방법은 클래스의 인스턴스를 생성하는 방식이다.[3]

따라서 본 논문에서는 Adapter 방식의 적용에 있어서 상속 과 위임의 차이를 분석하고 개별 Wrapper의 구성과 통합 Wrapper의 구성에 따른 장단점을 비교한다. 또한 개별 Wrapper 구성방식의 단점인 인터페이스 수의 증가에 따른 복잡성을 해결하기 위해 Facade 방식을 이용하여 복잡한 wrapper들을 손쉽게 사용할 수 있는 방법을 제안한다.

2. 관련연구

2.1 클래스에 의한 Adapter 패턴의 클래스(상속)

클래스에 의한 Adapter 패턴의 클래스는 기존의 클래스(Adaptee)를 재사용하기 위해 인터페이스를 이용하여 Adapter를 구성한다. 즉 사용자는 인터페이스를 이용하여 시스템에 접근하며 어댑터는 기존의 클래스를 확장하고 사용자가 접근하는 인터페이스를 구현 하여 기존 프로그램과 새로운 인터페이스를 연결하여 준다.



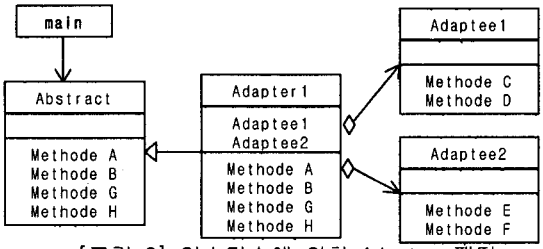
[그림 1] 클래스에 의한 Adapter 패턴

그림 1에서 보는 바와 같이 이러한 Adapter 패턴은

Adaptee를 상속받아 이를 활용하였다. [3] 이 방식은 하나의 Adaptee에 하나의 Adapter 만을 대응시켜 레거시 프로그램을 모듈화 할 수 있다. 반면에 프로그램이 확대될수록 Adapter의 수 및 크기가 증가하게 되고 이는 사용자가 오히려 사용하는데 어려움을 일으킬 수 있다.

2.2 인스턴스에 의한 Adapter 패턴의 클래스(위임)

기존의 클래스(Adaptee)를 재사용하기 위해 추상클래를 이용하여 Adapter를 구성한다. 즉 사용자는 추상클래스를 이용하여 시스템에 접근하며 어댑터는 기존의 클래스를 aggregation association으로 하여 인스턴스를 생성하고 사용자가 접근하는 추상클래스를 extends 하여 기존 프로그램과 새로운 추상클래스를 연결하여 준다.



[그림 2] 인스턴스에 의한 Adapter 패턴

그림 2에서 보는 바와 같이 이러한 Adapter 패턴은 인스턴스를 생성하여 Adapter를 구성하였다. [3] 이러한 구성은 레거시 프로그램의 개수 증가하여도 인스턴스를 추가하여 시스템을 간단하게 구축할 수 있다는 장점이 있다. 반면에 프로그램이 Adaptee의 개수가 증가함에 따라 Adapter의 크기가 커지게 된다. 또한 충분히 테스트가 이루어진 후에 사용하는 것이 바람직하다.

클래스 어댑터와 인스턴스 어댑터는 각기 다른 장단점에 따라 둘 중에 하나를 선택해서 설계한다. 클래스 어댑터는 임시객체의 도입을 피하며, 부가적인 포인터없이 어댑티 시킬 수 있다. 개발자는 서브클래스에 의해 어댑티의 몇가지 행동을 번복할 수 있다. 어댑티를 구체적인 클래스 어댑터에 대한 수영에 의해 Target을 적용시킬 때까지 크래스 어댑터는 클래스와 서브클래스를 적용시키지 않는다.

이에 반해서, 객체 어댑터는 많은 어댑티 즉, 어댑티와 그것에 대한 서브클래스들을 가지고 단일 어댑터화 시킨다. 어댑터는 한번에 모든 어댑티들에 대해 독단적인 기능성을 더할 수 있다. 그렇지만, 어댑티의 행위를 무효로 하기는 어렵다. 이것은 서브클래스화된 어댑티와 어댑티보다는 오히려 어댑티의 서브클래스에 적용시켜 만들어진 어댑터를 요구하기 때문이다[5].

클래스 어댑터의 경우 클래스 구성 규칙을 따르지 않은 Adaptee에 대하여 사용자가 원하는 새로운 클래스를 생성하며, 인스턴스 어댑터는 레거시 프로그램을 새로운 클래스에서 call하는 방식을 사용한다[6].

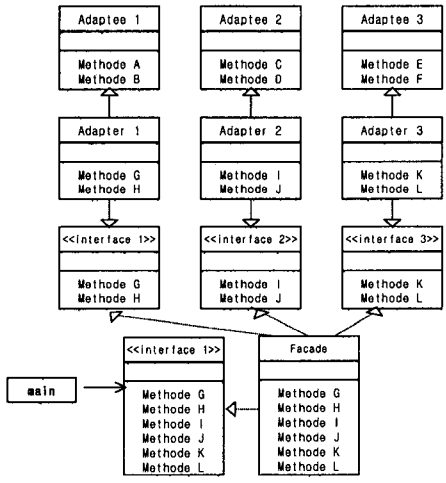
3. 클래스에 의한 Adapter 패턴을 이용한 시스템 최적화

위에서 살펴보았듯이 Adapter 패턴에는 두가지 형태의 구현방안이 존재하며, 각각의 장단점을 가지고 있다. 본 논문에서는 그중 Adaptee를 Overiding 할 수 있는 클래스에 의한 Adapter 패턴의 상속을 이용하여 개별 시스템을 Wrapping하고 이의 문제점인 Interface의 복잡성에 대한 해결방안을 제시함으로써, 시스템내의 Adapter의 크기를 줄이고, Interface의 조합을 손쉽게 접근할 수 있는 방안을 제시하고자 한다.

3.1 다수의 레거시 클래스 통합 인터페이스 구성

본 논문에서는 우선, 상속의 장점인 레거시 프로그램의 자유로운 수정과, Adapter의 크기가 증가하는 것을 방지하기 위해 상속을 이용하여 레거시 프로그램을 독립적으로 Wrapping하는 것이다.

즉, 각각의 레거시 프로그램에 대하여 일대일로 Adapter를 대응하고 이에 대한 인터페이스를 생성하는 것이다.



[그림 3] Adapter와 Facade를 이용한 클래스 통합

그림 3에서 보는 바와 같이 각각의 Adaptee에 대한 Adapter를 만들고 개별 인터페이스를 통해서 실행되도록 구현하였다. 즉, 통합하려는 Adaptee의 수가 증가하여도 Adapter의 크기가 증가하는 것을 방지하는 효과가 있다. 또한 하나의 Adaptee가 변경되어도 이러한 내용이 다른 Adaptee에 영향을 주지 않는다는 장점이 있어 문제점을 지역화 할 수 있다. 그러나 이러한 설계의 문제점은 다수의 Interface가 생성되어 사용자가 이를 통합하여 사용하고자 할 경우 복잡하다는 문제점이 있다.

[표 1] 개별구성과 통합구성의 비교

구분	장점	단점
개별	Adapter의 크기 최소화 문제를 지역화	Interface의 수 증가
통합	Adaptee의 추가 편리	Adapter의 수 및 크기가 증가

표 1는 Adapter의 개별 구성과 통합구성의 장단점을

비교하였다. 보는바와 같이 개별적 Adapter의 구성은 인터페이스의 개수가 많아져 복잡하다는 단점이 있다.

3.2 다수의 인터페이스를 통합하는 Facade 클래스

많은 클래스들을 제어하기 위해서는 '창구'를 준비해두는 것이 좋다. 그렇게 함으로써 많은 클래스들을 개별적으로 제어하지 않아도 '창구'에게 요구만 하면 일이 끝나도록 할 수 있다. Facade 패턴은 복잡하게 얽혀있는 것을 정리해서 높은 레벨의 인터페이스를 제공한다. Facade의 역할은 시스템의 외부에는 간단한 인터페이스를 보여주면서, 시스템의 안쪽에 있는 각 클래스의 역할이나 의존관계를 생각해서 올바른 순서로 클래스를 이용하는 역할을 한다.[4]

그림 3에서 보는 바와 같이 많은 인터페이스들이 존재하는 레거시 프로그램을 손쉽게 사용하기 위해 Facade 클래스를 두고 활용함으로써 개별구성 방식이 가지고 있는 단점을 해결할 수 있음을 보여준다.

4. 성능평가

우선 N개의 Adaptee를 대상으로 개별구성과 통합구성에 대한 연구를 가정하였다. 여기서 특정 Adaptee A.class 에 대한 Adapter를 문자에서 숫자로 값을 받는 형태로 수정하고자 할 경우 수정에 들어가는 스텝수 및 영향을 미치는 Adaptee 수를 비교하였다.

이러한 경우 개별 구성은 Adaptee들에 대응하는 각각의 Adapter를 구성하여 일대일 대응하는 인터페이스를 생성한다. 반면, 통합구성의 경우 각각의 Adapter를 구성하여 이를 통합하는 인터페이스를 생성한다. 여기서 통합 구성된 인터페이스는 하나의 Adapter가 추가됨에 따라 Adapter를 정의하는 메소드의 정의가 추가적으로 필요하며, 따라서 인터페이스의 크기가 증가하고 이는 다른 Adapter에 새로운 정의를 위한 수정을 필요로 한다. 따라서 각각의 새로운 Adaptee를 정의하기 위해 기존의 Adapter가 크기가 증가한다. 또한 하나의 Adapter가 메소드 정의가 변경되면 이는 다른 Adapter들에도 재정의를 필요로 하여 영향을 미친다. 반면 개별 구성의 경우 각각의 수정된 Adapter는 해당 인터페이스 및 Facade에만 영향을 준다.

어댑터 메소드 수정 시 스텝 수 : MC

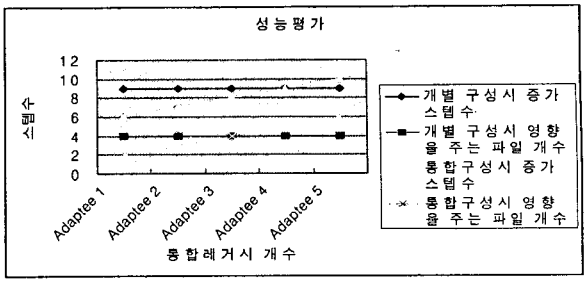
Adaptee의 개수 : N , Facade 패턴 구현을 위한 class 수 : SC

통합 구성의 어댑터 메소드 수정 시 수정 스텝수 : TMS

개별 구성의 어댑터 메소드 수정 시 수정 스텝수 : IMS

$$TMS = MC + N$$

$$IMS = MC + SC$$



[그림 4] 개별Wrapping과 통합Wrapping의 Adapter 수정 시 영향을 미치는 파일 수 및 스텝수 비교

그림4는 개별 구성과 통합구성에서 메소드의 재정의에 따른 수정 스텝수 및 영향을 미치는 파일수를 보여준다.

즉, 통합구성은 래거시 소프트웨어가 증가함에 따라 스텝수 및 영향을 미치는 파일수가 증가하지만, 개별 구성은 그 영향을 미치는 스텝수와 파일의 개수가 일정하여 개수가 증가함에 따라 그 효과가 점점 커진다.

5. 결론

과거 프로그램에 대한 재사용을 위한 방안은 다양하다. 그중 본 논문에서는 레거시 프로그램의 재사용성을 위해 많이 사용되는Adapter 패턴을 중점적으로 분석하였다. 개별 Wrapping의 경우 Adapter의 크기를 최소화 하고, 수정 시 이에 미치는 영향을 지역화 할 수 있다는 장점이 있다. 반면, 다수의 인터페이스는 사용자가 사용하는데 있어 혼돈을 일으키는 원인이 되기도 하는데 이를 해결하기 위해 또 하나의 디자인 패턴인 Facade 패턴을 제안하였다.

향후에는 본 논문에서 제안한 내용 외에 다른 디자인 패턴과 비교 분석하여 가장 효과적인 레거시 프로그램 Wrapper개발방안에 대하여 제안할 것이다.

6. 참고문헌

[1] Katharine Whitehead,"What are components?." , *Component-based Development, addison wesley*,first edition,18~34 ,2002

[2] Lawrence Wilkes, "Creating Component from Legacy Applications", CBDI Forum Journal, December 1998

[3] Hiroshi Yuki, "Adapter Pattern.", *JAVA GENGO DE MANABU DESIGN PATTERN NYUMON, Soft bank .INC,* Japaness edition, 54~64, 2001

[4] Hiroshi Yuki, "Facade Pattern.", *JAVA GENGO DE MANABU DESIGN PATTERN NYUMON, Soft bank .INC,* Japaness edition, 268 ~ 279, 2001

[5] 권민주, "소프트웨어 재사용을 위한 설계패턴들에 관한연구" , 한국정보처리학회 논문지 제6권 제2호,1999

[6] James W. Cooper, "The Adapter Pattern",*The Design Patterns Java Companion, addison wesley,* first edition, 81~89, 1998