韓國IT서비스學會誌
第8卷 第3號
2009年 9月

재사용 가능한 객체 식별을 위한 Two-Pass 추상화 원칙 제안*

고형호** · 김능회*** · 이동현*** · 인 호***

Two-Pass Abstraction Principle for Identifying Reusable Object*

Hyungho Ko** · Neunghoe Kim*** · Donghyun Lee*** · Hoh Peter In***

■ Abstract ■

As the software development cycles is getting shorter, the software reusability is emphasized accordingly. Specifically, the design reusability is being recognized as one of the most important factor to increase the software quality and productivity and make the maintenance cost down. Two essential abilities are needed to improve the design reusability. One is the identification of the reusable objects, and the other is the organization of the relationships among the objects. However, the existing methods using such as a grammatical analysis, a scenario matching and a unit of design problems(design pattern) have not been proposed proper principles to identify the reusable objects on the basis of the abstraction which is the core of the object-oriented concept. In this paper, we will offer the Two-Pass abstraction principle based in the abstraction concept.

Keyword : Object-Oriented, Abstraction, Design Patterns

논문투고일 : 2009년 07월 17일 논문수정완료일 : 2009년 09월 14일 논문게재확정일 : 2009년 09월 16일

* 본 연구는 "지식경제부 및 정보통신산업진흥원의 대학 IT연구센터 지원사업(NIPA-2009-(C1090-0903-0004))"과 "정보통신산업진흥원의 SW공학 요소기술 개발과 전문인력 양성사업"의 지원으로 수행되었음. 질문이 있을 시에 교신 저자인 인호(hoh_in@korea.ac.kr)로 연락을 주시기 바랍니다.
이 논문은 2009 한국컴퓨터종합학술대회에서 '디자인 패턴 참여 객체 도출을 위한 Two-Pass 추상화'의 제목으로 발표된 논문을 확장한 것임.

** 고려대학교 컴퓨터정보통신대학교 소프트웨어공학과
*** 고려대학교 정보통신대학 컴퓨터 · 전파통신공학과

1. 서 론

소프트웨어 개발주기가 점점 짧아짐에 따라, 소프트웨어 품질과 생산성을 높이고 유지보수 비용을 절감하기 위해 소프트웨어 재사용성이 강조된다[13].

재사용성(Reusability)이란 소프트웨어 컴포넌트가 최초 개발된 이후 새로운 응용 분야에서 반복적으로 사용되는 능력으로 정의된다[4]. 특히 재사용성은 소프트웨어 전체 영역을 구현 중심으로 반복 사용하는 능력으로 정의되는 이식성(Portability)과 구별되며, 전체가 아닌 부분의 개념인 컴포넌트를 설계 중심으로 반복 사용하는 것에 초점을 맞춘다. 즉, 재사용성은 설계에 관한 문제(Design consideration)이고 이식성은 구현에 따른 문제(Implementation consideration)인 것이다[1].

이처럼 설계의 재사용성은 소프트웨어 재사용 전체 고려 대상 중 핵심이 된다. 설계 원칙에 충실한 설계라 할지라도 재사용성이 고려되지 않았다면 해당 컴포넌트들을 미래에 새로운 응용 분야에 반복 사용하리라 기대할 수 없다. 재사용성을 위한 일반적인 설계 지침은 모델의 사용(Use of Model), 계층구조의 설계(Layered Architecture), 접속관계 고려(Interface Consideration) 및 효율성의 선택(Efficiency Trade-Offs) 등이 있다[4].

설계의 재사용성을 높이기 위해 재사용할 수 있는 객체(Reusable Object) 식별과 이들 간의 관계 구성은 필수적인 요소가 된다. 이를 위한 많은 연구들 중에서 GoF(Gang of Four ; Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)에 의해 발표된 "디자인 패턴"이 가장 널리 사용되고 있다[5].

디자인 패턴은 소프트웨어 디자인 단계에서 공통적으로 발생하는 설계 문제에 대해, 재사용할 수 있는 객체와 이들 간의 관계로 구성된 해법(Solution)을 제시한다[5, 14, 15]. 또한 디자인 패턴은 재사용할 수 있는 객체 식별을 객체 단위가 아닌, 설계 문제 단위로 객체 식별 방법을 제시하고 있다.

디자인 패턴은 설계에 대한 높은 재사용성과 설계 문제 단위의 객체 식별 및 그들간의 관계 구성 때문에 그 우수성을 인정받고 있으나 실무에서 적극적으로 사용되고 있지 못하고 있는 실정이다. 그 대표적인 원인은 디자인 패턴 적용 절차에서 확인할 수 있다. 패턴 적용 절차는 다음과 같다. 첫째, 요구사항으로부터 설계 문제를 도출하여 이에 적합한 패턴을 찾는다. 둘째, 해당 패턴에서 요구하는 참여 객체(Participants)를 도출한다. 셋째, 그 참여 객체들 사이의 관계를 구성한다. 여기서, 첫째, 설계 문제로부터 적합한 패턴을 찾는 방법은 디자인 패턴 카탈로그(Catalog) 및 선정 프로세스[2] 등 다양한 방법이 제시되고 있다. 그리고 셋째, 식별된 참여 객체들 사이의 관계를 구성하는 방법은 객체지향 설계원칙의 적용이 제시되고 있다[6, 9, 10, 12]. 그러나 둘째, 디자인 패턴에서 제시하는 재사용할 수 있는 객체 식별을 위한 많은 연구 결과가 제시되고 있으나 객체지향 핵심인 추상화를 이용한 방법은 제시되지 않고 있다[7, 11]. 이는 디자인 패턴만이 아닌 기존 언어 문법적 객체 식별과 시나리오 기반 객체 식별등 다양한 객체 식별 방법들이 갖고 있는 공통적인 문제점이다. 이에 본 논문에서는 객체지향 설계의 핵심인 추상화를 기반으로 하여 재사용할 수 있는 객체를 식별하는 "Two-Pass 추상화 원칙"을 제안한다.

본 논문은 다음과 같이 구성된다. 제 2장에서는 객체지향 설계원칙을 소개하고, 제 3장에서는 관련 연구로서 재사용을 높일 수 있는 객체 식별에 관한 기존 연구 분석 결과(Kruger)와 디자인 패턴이 제대로 적용되지 못하고 있는 이유를 살펴본다. 제 4장에서는 본 논문에서 제안하는 Two-Pass 추상화 원칙 정의 및 적용 절차를 확인하고, 이 원칙으로부터 디자인 패턴 참여 객체를 식별한다. 제 5장에서는 JHotDraw 프로그램의 요구사항을 통해 Two-Pass 추상화 원칙의 적용 방법을 설명하고, 제 6장에서는 결론과 향후 연구에 대해 설명한다.

2. 객체지향 설계원칙

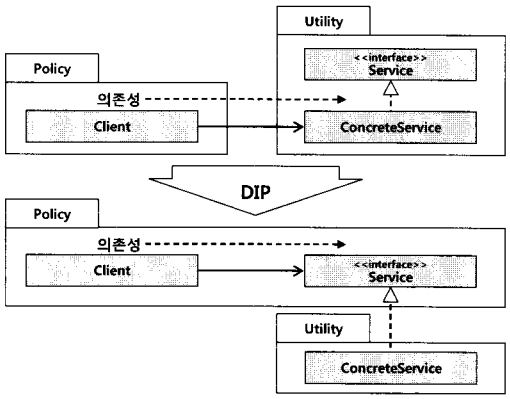
요구사항에서 식별된 객체를 설계 문제의 의도에 따라, 재사용성이 높은 객체 관계로 구성하는 문제를 효과적으로 해결하기 위하여 많은 학자들은 다음과 같이 4가지 객체지향 설계원칙을 제시했다[6, 9, 10, 12].

2.1 Dependency Inversion Principle

Dependency Inversion Principle는 "클라이언트는 구체 클래스가 아닌 인터페이스(추상 클래스)에 의존해야 한다"는 것을 의미한다.

[그림 1]에서 Dependency Inversion Principle 적용 전에 상위 수준의 모듈(Client)은 하위 수준의 모듈(ConcreteService)에 의존하므로 상위 수준의 모듈은 하위 수준의 모듈에 대한 변경의 영향을 받는다. 또한 상위 수준의 모듈이 포함된 라이브러리는 하위 수준의 모듈이 포함된 라이브러리에 대한 변경의 영향을 받는다.

이러한 문제를 해결하기 위해 Dependency Inversion Principle을 적용하면 상위 수준의 모듈은 인터페이스에 의존하므로 상위 수준의 모듈은 하위 수준의 모듈에 대한 변경의 영향을 받지 않는다. 또한 상위 수준의 모듈이 포함된 라이브러리는 하위 수준의 모듈이 포함된 라이브러리에 대한 변경의 영향을 받지 않는다.

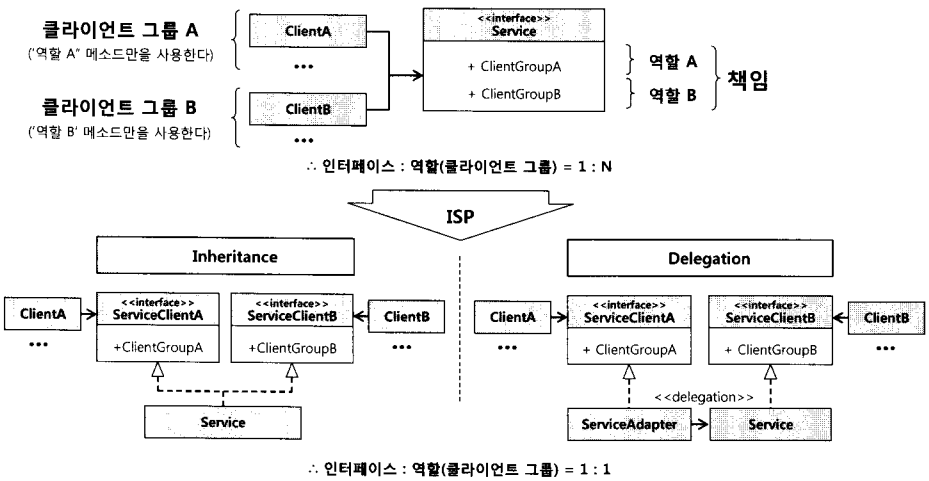


[그림 1] Dependency Inversion Principle

2.2 Interface Segregation Principle

Interface Segregation Principle는 "클라이언트는 특화된 여러 개의 인터페이스가 하나의 범용 인터페이스보다 낫다"는 것을 의미한다.

[그림 2]에서 Interface Segregation Principle 적용 전은 클라이언트 그룹(A, B)는 자신이 사용하지 않는 인터페이스 메소드(Service)에 의존한다.



[그림 2] Interface Segregation Principle

또한 자신이 사용하지 않는 메소드의 변화에 영향을 받는다. 인터페이스와 클라이언트 그룹과의 관계는 "1 : N"으로 구성된다. Interface Segregation Principle 적용 후는 클라이언트 그룹은 자신이 사용하지 않는 메소드에 의존하지 않는다. 또한 자신이 사용하지 않는 메소드의 변화에 영향을 받지 않는다. 인터페이스와 클라이언트 그룹 간의 관계가 "1 : 1"로 구성되기 때문이다.

2.3 Substitution Principle

Substitution Principle은 "기반 클래스는 파생 클래스로 대체할 수 있어야 한다"는 것을 의미한다.

[그림 3]에서 Liskov[12]에 의해 제안된 Substitution Principle 적용 전은 파생 클래스에서 기반 클래스의 모든 메소드를 지원(메소드가 던지는 예외까지 포함)하지 않으면 'IS-A' 관계가 성립하지 않는다. 또한 다형성은 'IS-A' 관계를 기반으로 하므로 안정적인 다형성 획득을 힘들게 한다(잠재적인 Open-Closed Principle 위반).

Substitution Principle 적용 후는 파생 클래스는 기반 클래스의 모든 메소드를 지원한다(올바른 'IS-A' 관계 구성). 또한 올바른 'IS-A' 관계를 기반으로 안정적인 다형성 획득을 할 수 있다(Open-Closed Principle 준수).

2.4 Open-Closed Principle

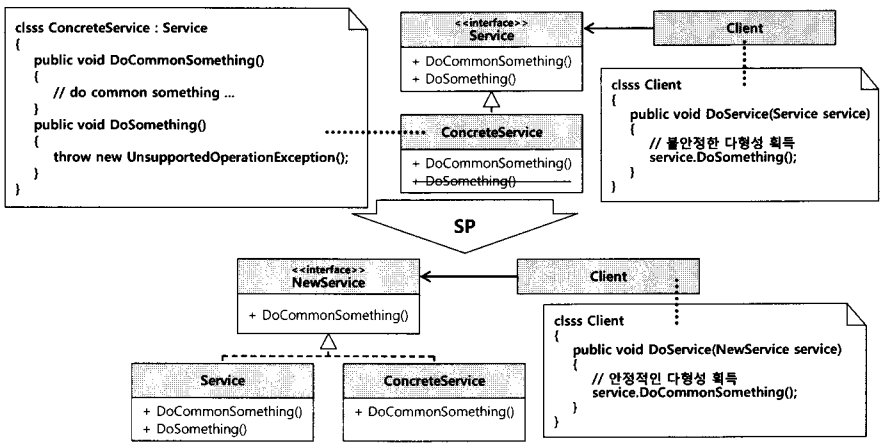
Open-Closed Principle는 "확장에는 열려 있어야 하고, 변경에는 닫혀 있어야 한다"는 것을 의미한다.

[그림 4]에서 Open-Closed Principle 적용 전은 모듈(Service)는 구체 클래스(ConcreteServiceA/B)에 의존하므로 구체 클래스의 내부 변경에 영향을 받는다. 또한 새롭게 추가(확장)되는 구체 클래스를 소스 코드 수정 없이는 인지할 수 없다. Open-Closed Principle 적용 후는 모듈(Service)는 추상화(ServiceImpl)에 의존하므로 구체 클래스(ConcreteServiceA/B)의 내부 변경에 영향을 받지 않는다. 또한 새롭게 추가(확장)되는 구체 클래스를 소스 코드 수정 없이 인지할 수 있다.

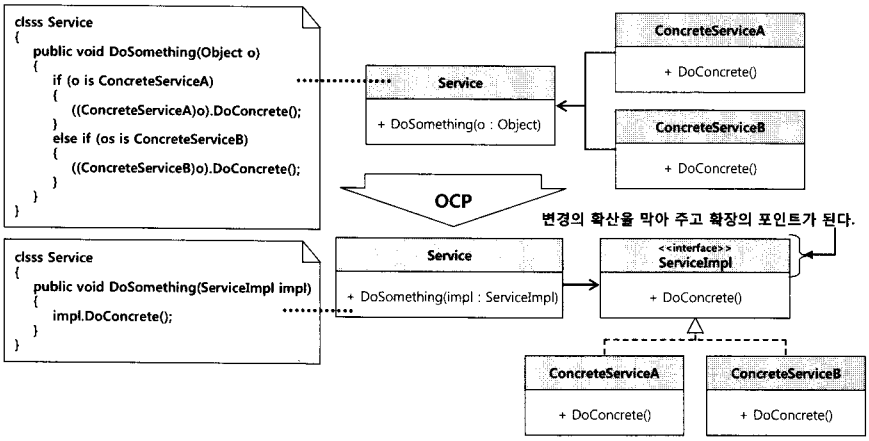
3. 관련 연구

3.1 Krueger의 추상화 원칙

Krueger는 그의 연구에서 "추상화(Abstraction)는 모든 재사용 메커니즘의 기반을 이루고 있다"고 했다[3]. Krueger가 제시한 추상화 원칙은 Specification 상위 층과 Realization 하위 층으로 구성된다[3, 8]. 추상화 Specification 층은 Realization 층

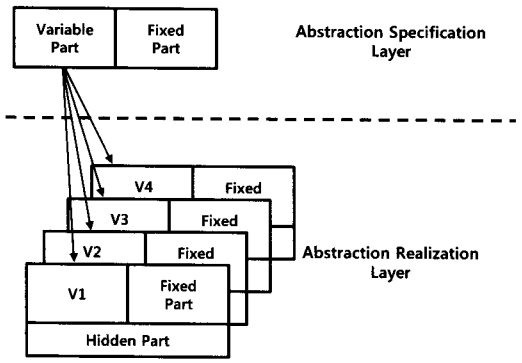


[그림 3] Substitution Principle



[그림 4] Open-Closed Principle

의 변화로 정의된다. 각 층의 세부 형식은 [그림 5]와 같다.



[그림 5] Krueger의 추상화 원칙[3]

추상화 Specification은 Variable Part와 Fixed Part로 구성되며, 추상화 Realization은 Variable Part와 Fixed Part 그리고 Hidden Part로 구성된다. Hidden Part는 추상화 Realization에서만 표시된다. Variable Part는 변화하는(Variant) 성질을 갖고, Fixed Part는 변화하지 않는(Invariant) 성질을 갖는다.

추상화 Specification의 Fixed Part의 내용은 추상화 Realization의 Fixed Part로 변경 없이 실체화된다. 그러나 Specification의 Variable Part는 목적에 따라 Realization의 Variable Part 또는 Fixed Part 및 Hidden Part로 실체화 된다.

예를 들어, Stack을 추상화한다고 하자. 먼저 추상화 Specification 요소를 찾아야 한다. Stack LIFO (Last-In-First-Out) 연산은 변화하지 않는 성질이기 때문에 Fixed Part가 된다. 그러나 Stack 연산 대상인 원소(Element)는 다양하게 변화될 수 있기 때문에 Variable Part가 된다. 그리고 Stack의 데이터 최대 입력 깊이는 변화될 수 있기 때문에 Variable Part가 된다. 만약 데이터 최대 입력 깊이가 추상화 Realization으로 실체화 되면서 Fixed Part로 이동한다면 변경될 수 없는 성질이 되어 사용자는 Stack의 데이터 최대 입력 깊이를 조정할 수 없게 된다. 그러나 Variable Part로 실체화된다면 사용자는 직접 그 깊이를 다양하게(10, 1000, 무제한) 변경시킬 수 있게 된다. 또한 Hidden Part로 실체화된다면 데이터 최대 입력 깊이는 사용자 관심 대상에서 완전히 제외된다.

Krueger의 추상화 원칙에 따라 Stack은 사용자 요구 사항에 따라 최대 깊이를 조정할 수 있는 객체와 최대 깊이가 고정된것 그리고 최대 깊이를 제공하지 않는 구체화된(Concrete) 객체를 얻을 수 있다. 그러나 요구사항에 따라 변화하는 Variable Part를 수용할 수 있는 실체화된 Part(인터페이스

가 여기에 해당) 식별 방안을 제시하지 않는다.

3.2 디자인 패턴

3.2.1 디자인 패턴 소개

디자인 패턴은 재사용할 수 있는 객체 식별을 객체 단위가 아닌 설계 문제 단위로 객체 식별 방법을 제시한다.

크리스토퍼 알렉산더(Christopher Alexander)가 건축학에서 도입한 "패턴(Pattern)" 개념은 소프트웨어 공학에도 영향을 미쳤고, 많은 연구들 중에서 GoF에 의해 성공적인 프로젝트를 대상으로 반복적으로 발생하는 설계 문제를 재사용성이 높은 해법을 발굴하여 패턴 단위로 정의하여 발표하였다. 디자인 패턴은 문제와 해법을 중심으로 패턴의 이름, 의도, 동기들로 구성되어 있다. 특히, 참여 객체에는 디자인 패턴을 구성하는 객체의 역할을 정의한다. 정의된 역할을 기준으로 패턴을 구성하는 객체를 식별하게 된다.

디자인 패턴은 설계에 대한 높은 재사용성 때문에 그 우수성을 인정받고 있으나 제대로 적용되지 못하고 있는 실정이다. 대표적인 원인은 2가지로 정의될 수 있다.

3.2.2 디자인 패턴 적용의 문제점

문제점 1 : 디자인 패턴에 대한 충분한 사전 지식 필요

디자인 패턴은 공통 어휘를 제공하여 설계의 분명한 의도를 전달하여 설계자들 사이의 의사 교환 도구로 사용되어 설계를 단순화 시킬 수 있다. 또한 <표 1> 등을 제시하여 설계 문제의 목적과 방법에 따라 적합한 패턴을 식별하여 이를 적용할 수 있도록 제공하고 있다.

그러나 디자인 패턴은 일반화 수준이 높은 영역에 대해서만 제공되므로 특화된 문제 도메인에 대해서는 직관적인 해결책을 제시하지 못한다. 물론 이러한 문제들은 디자인 패턴의 조합으로 어느 정도 해결할 수 있지만, 디자인 패턴에 대한 충분한 사전 지식과 이들을 적절히 조합할 수 있는 능력 및 경험을 필요로 하므로 쉽게 접근하기 어렵다. 또한 이런 접근법은 도메인 문제를 해결하기 위한 본질적 접근에서 벗어나는 현상을 초래할 수 있다.

〈표 1〉 GoF 디자인 패턴 카탈로그[5]

		목적		
		생성적	구조적	행위적
범위	클래스	Factory M.	Adapter(C.)	Interpreter
				Template M.
	객체	Abstract F.	Adapter(O.)	Chain of R.
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

문제점 2 : 패턴을 구성하는 명확한 객체 식별 방법 필요

디자인 패턴은 "참여 객체"에 객체 식별을 위해 역할을 정의하여 제시한다. 그러나 각각의 설계 문제들은 서로 연관되어 있어 디자인 패턴들 또한 조합된다. 특정 디자인 패턴의 참여 객체는 다른 디자인 패턴의 참여 객체가 될 수 있다. 이는 하나의 객체가 설계 의도에 따라 다양한 역할로 정의될 수 있음을 의미한다. 이러한 복합적인 객체 역할로 인하여 단순한 객체의 역할만으로 객체를 식별하는 것을 더욱 어렵게 한다.

4. Two-Pass 추상화 원칙 제안

4.1 Two-Pass 추상화 원칙

본 장에서는 Two-Pass 추상화 원칙을 정의하고, 이 원칙으로부터 재사용할 수 있는 객체를 식별하는 방법을 설명한다.

Two-Pass 추상화 원칙은 Krueger의 추상화 원

칙을 확장한 것으로 재사용할 수 있는 객체 식별을 목적으로 하며, 추상화 Specification 층에서 Realization 층으로 실체화 시키는 First Pass와 변화를 일으키는 원인(Cause)을 기준으로 재추상화(Reabstraction)하는 Second Pass 과정으로 정의된다.

객체는 클래스의 특징에 따라 구분될 수 있다. 특히 Krueger의 연구 결과에 따라 재사용성을 위해 클래스의 변화 성질을 기준으로 구분하면, 구체적인 구현을 통해 설계 의도에 따라 변화하는 성질을 갖는 구체 클래스(Concrete Class, 구현이 있는 Class가 여기에 해당)와 구체적인 구현 없이 객체들간의 메시지 전달을 목적으로 변화하지 않는 성질을 갖는 인터페이스(Interface, 구현이 없는 Interface와 Abstract Class가 여기에 해당)로 구분된다. Krueger가 제시한 추상화 원칙은 아직 변화하지 않는 성질을 갖는 인터페이스에 해당하는 Part가 존재하지 않는다.

본 논문에서는 변화를 수용하기 위한 Part를 제안한다. 변화를 일으키는 원인을 중심으로 재추상화(Reabstraction)를 수행하여 변화를 수용할 수 있는 Cause Part를 추가하였다. Cause Part는 구체적인 구현 없이 변화를 수용하는 것이 목적이기 때문에 인터페이스로 식별될 수 있다. Fixed Part는 변화를 구현하기 위한 구체적인 역할에 해당되므로 구체 클래스의 메소드로 도출될 수 있다. 또한 Specification에 변화를 수행시키는 주체를 식별하기 위해 Subject Part를 추가하였다.

Subject Part와 Cause Part가 추가된 추상화 원칙은 [그림 6]과 같이 Two-Pass 추상화를 수행하여 실체화 한다

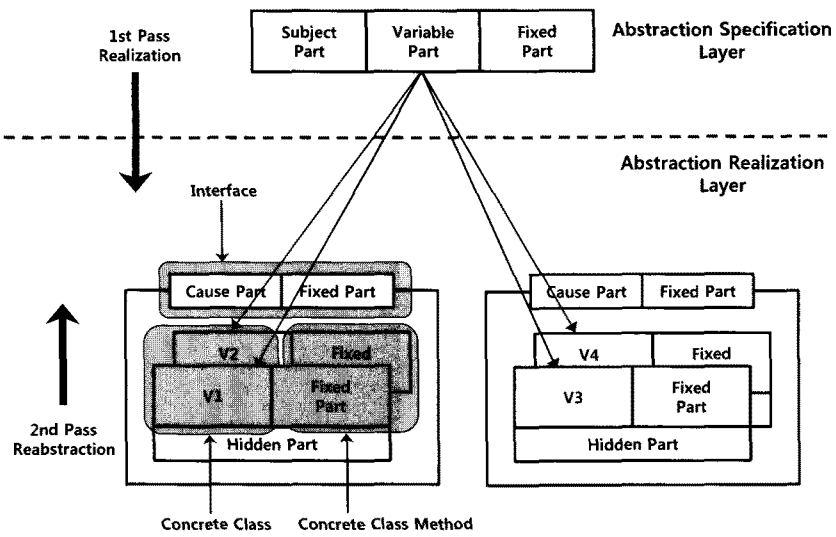
- First Pass

대상(Subject)의 행위를 중심으로 추상화 Specification에서 추상화 Realization으로 실체화 시킨다.

- Second Pass

변화를 일으키는 원인을 기준으로 재추상화(Reabstraction)를 수행한다. 재추상화는 변화의 원인이 Specification의 Variable Part가 될 때까지 반복적으로 수행한다.

재추상화는 Bottom-up 방식으로 구체 클래스(Concrete Class)로부터 Variable Part와 Fixed Part를 재분석하여 Abstraction Hierarchy를 재구성하고, 더불어 필요에 따라 First Pass의 시작이 되는 최상위 클래스를 수정하는 일련의 작업을 말한다.



[그림 6] Two-Pass 추상화 원칙

4.2 Two-Pass 추상화 원칙과 객체지향 설계 원칙

Two-Pass 추상화 원칙과 객체지향 설계 원칙을 설계 문제에 적용하여 재사용 가능한 설계를 얻을 수 있다. 이는 재사용할 수 있는 객체를 기본으로 한 객체들 간의 관계를 구성하기 때문이다. 이를 위한 구체적인 적용 절차는 아래와 같다.

첫째, 추상화 단위로 설계 문제를 식별한다.
둘째, 식별된 설계 문제로부터 Two-Pass 추상화 원칙을 적용한다.
셋째, Two-Pass 추상화 원칙 결과(인터페이스와 구체 클래스)에 객체지향 설계 원칙을 적용한다.
넷째, 재사용 가능한 설계 결과를 얻는다.

위 절차는 객체지향 핵심인 추상화를 중심으로 재사용 가능한 객체를 식별하고 설계 의도에 따라 객체지향 설계원칙을 적용하기 때문에 디자인 패턴으로까지 유도될 수 있다. 또한 디자인 패턴의 조합된 다양한 형태로 까지 유도될 수 있다. 이는 디자인 패턴에 대한 사전 학습과 이를 적용하기 위한 경험적 지식을 최소화 시킬 수 있는 중요한 역할도 담당할 수 있다.

4.3 Two-Pass 추상화 원칙과 디자인 패턴

GoF의 23가지 설계 문제를 대상으로 Two-Pass 추상화 원칙을 적용하여 <표 2>와 같이 해당 패턴의 참여 객체를 식별 결과를 얻을 수 있다(디자인 패턴 참여 객체는 각 패턴에 정의된 참여 객체 이름으로 표시하였다).

<표 2>와 같이 Two-Pass 추상화 원칙을 통해 얻은 객체들은 설계 문제 의도에 따라 Façade와 Memento 그리고 Singleton 패턴을 제외한 모든 패턴의 참여 객체로 도출될 수 있다. 도출된 객체를 대상으로 객체지향 설계원칙을 적용하면 디자인 패턴에 해당하는 설계 결과를 얻게 된다. 디자인 패턴의 참여 객체는 재사용 가능한 객체이기 때문에 Two-Pass 추상화 원칙으로 도출될 수 있는 것이다.

〈표 2〉 Two-Pass 추상화 원칙 적용 결과

Pattern		Subject Part	Cause Part	Variable Part
Abstract Factory			Abstract Factory	Concrete Factory
Adap.	Class	Adaptee	Adapter	Target
	Obj.	Adaptee	Adapter	Target
Bridge			Abstraction	Refined Abstraction
			Implementer	Concrete Implementer
Builder		Director	Builder	Concrete Builder
Chain R.			Handler	Concrete Handler
Command			Command	Concrete Command
Composite			Component	Composite, Leaf
Decorator			Component / Decorator	Concrete Decorator
Façade				
Factory M.			Creator	Concrete Creator
Flyweight			Flyweight	Concrete Flyweight
Interpreter			Abstract Expression	Terminal Expression, NonTerminal Expression
Iterator			Iterator	Concrete Iterator
Mediator			Mediator	Concrete Mediator
Memento				
Observer			Subject	Concrete S.
			Observer	Concrete O.
Prototype			Prototype	Concrete P.
Proxy			Subject	Real Subject, Proxy
Singleton				
Proxy			Subject	Real Subject, Proxy
Strategy		Context	Strategy	Concrete Strategy
Template Method			Abstract Class	Concrete Class
Visitor			Visitor	Concrete Visitor
			Element	Concrete Element

5. 사례 연구

본 연구에서 제안된 디자인 패턴 참여 객체 도출을 위한 Two-Pass 추상화 원칙의 적용 가능성을 확인하기 위하여 디자인 패턴의 GoF 중 한명인 Erich Gamma가 개발한 도형 편집기 프로그램 JHotDraw[16]의 요구사항 중 "도형 그리기" 설계에 이 원칙을 적용하였다.

5.1 요구사항

위 요구사항으로부터 Two-Pass 추상화 원칙을 적용하기 위해서 Specification을 찾아야 한다.

> *JHotDraw는 도형 편집기 프로그램이다. 이 프로그램은 다양한 도형을 그린다. 도형은 타원, 사각형 및 조합된 도형으로 분류된다. 그리고 도형은 상황에 따라 배경과 경계선을 그릴 수 있다.*

5.2 Two-Pass 추상화 원칙

5.2.1 First Pass

요구사항으로부터 "도형"은 다양한 타입을 갖는 변화되는 성질임을 확인할 수 있다. 그리고 "그리기"는 변화하지 않는 성질임을 확인할 수 있다. 그리고 도형 그리기를 수행 주체는 View이다. 이를 기준으로 하여 First Pass 실체화 과정을 수행하여 Realization의 Variable Part 항목을 도출하면 다음과 같다. "조합된 도형, 타원, 사각형, 배경 그리고 경계선"이다. Fixed Part는 변경 없이 Realization으로 실체화 한다.

5.2.2 Second Pass

Realization의 Variable Part를 대상으로 Second Pass 재추상화 과정을 수행한다. 조합된 도형과 타원 그리고 사각형의 변화 원인은 도형의 타입이다. 그리고 배경과 경계선의 변화 원인은 도형 그리기 기능의 확장이다. 이를 근거하여 Cause Part 항목을 도출한다. 타입과 확장의 변화 원인은 Specification의 Variable Part가 된다. 그러므로 더 이상 Second Pass 작업을 반복하지 않는다. Two-Pass 추상화 결과는 [그림 7]과 같다.

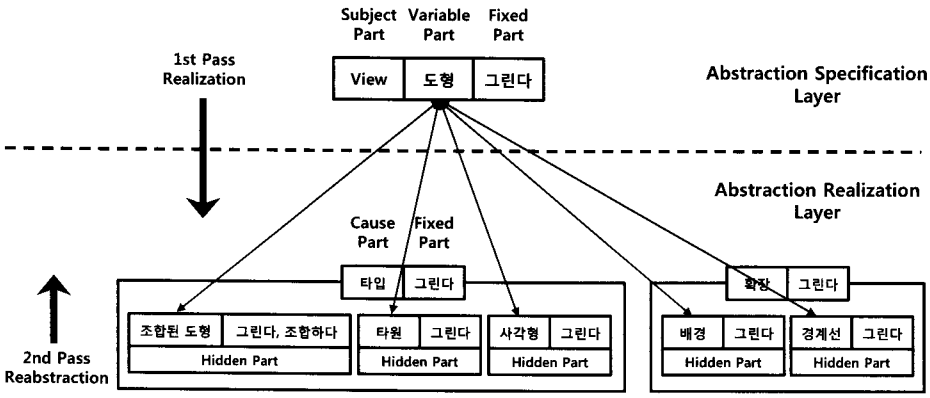
5.3 객체지향 설계원칙

Two-Pass 추상화 원칙을 통해 얻은 객체들을 대상으로 설계 문제의 의도에 따라 객체지향 설계 원칙을 적용하면 [그림 8]과 같다.

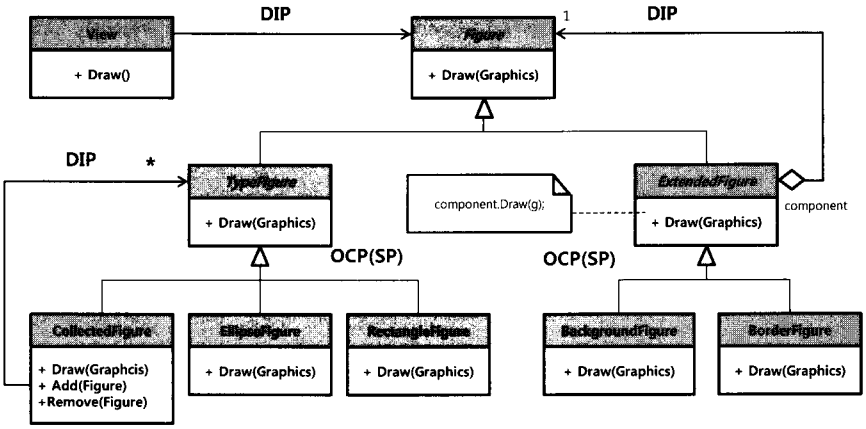
5.3.1 DIP(Dependency Inversion Principle) 원칙 적용 결과

- View

View는 인터페이스 역할을 수행하고 있는 Fi-



[그림 7] Two-Pass 추상화 원칙 결과



[그림 8] 객체지향 설계원칙 적용

gure에 의존하기 때문에 다양한 도형 및 그리기 기능 확장으로 인한 변화의 영향을 받지 않는다.

• CollectedFigure

CollectedFigure는 인터페이스 역할을 수행하고 있는 TypeFigure에 의존하기 때문에 다양한 도형 확장으로 인한 변화의 영향 없이 조합된 도형을 그릴 수 있다.

• ExtendedFigure

ExtendedFigure는 인터페이스 역할을 수행하는 Figure에 의존하기 때문에 다양한 도형 그리기 기능 확장으로 인한 변화의 영향 없이 도형 그리기 기능을 확장할 수 있다.

5.3.2 SP(Substitution Principle)과 OCP (Open-Closed Principle) 원칙 적용 결과

• Figure

Figure의 파생 클래스들은 Figure와 대체할 수 있기 때문에 안정적인 다형성을 얻는다.

• TypeFigure

TypeFigure의 파생 클래스들은 TypeFigure와 대체할 수 있기 때문에 안정적인 다형성을 이용하여 설계 변경 없이 새로운 도형 클래스를 추가할 수 있다.

• ExtendedFigure

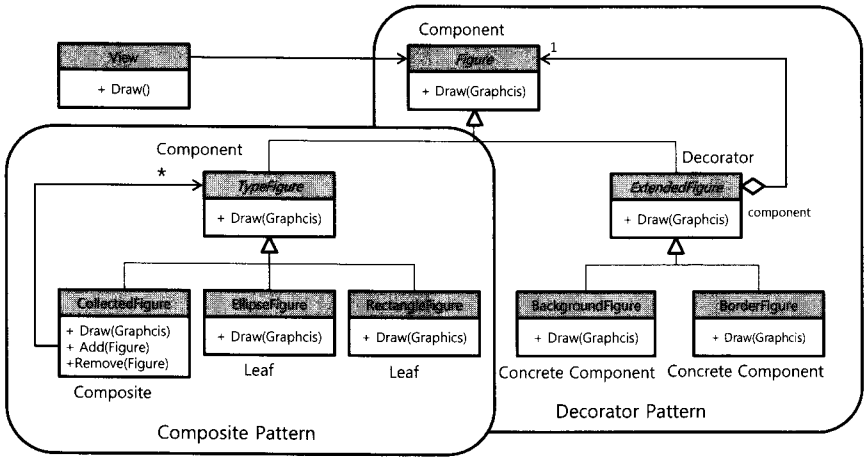
ExtendedFigure의 파생 클래스들은 Extended Figure와 대체할 수 있기 때문에 안정적인 다형성을 이용하여 설계 변경 없이 새로운 도형 그리기 클래스를 추가할 수 있다.

5.4 설계 결과

Two-Pass 추상화 원칙을 통해 얻은 객체들을 대상으로 객체지향 설계원칙을 적용하여 [그림 9]와 같이 Composite 패턴과 Decorator 패턴으로 설계된 결과를 얻을 수 있다.

6. 결 론

소프트웨어의 품질과 생산성을 높이고 유지보수 비용을 절감하기 위해 처음부터 재사용 가능한 설계를 수행하는 것이 중요시되고 있는데, 이로 인해 재사용할 수 있는 객체(Reusable Object) 식별과 이들 간의 관계 구성은 필수적인 요소가 되었다. 언어 문법적 객체 식별과 시나리오 기반 객체 식별 그리고 설계 문제 단위의 객체 식별(디자인



[그림 9] 디자인 패턴

패턴) 등 다양한 연구 결과가 제시되고 있으나 객체지향 핵심인 추상화를 이용한 방법은 제시되지 않고 있다.

본 논문에서는 이러한 문제점을 해결하기 위하여 객체지향의 핵심 원리인 추상화(Abstraction)를 기반으로 하여 재사용 가능한 객체를 식별하는 원리적 접근법을 제시하였다. JHotDraw 프로그램 요구사항을 통해 디자인 패턴에 대한 충분한 사전 지식과 경험이 없는 객체지향 설계 초보자도 Two-Pass 추상화 원칙과 객체지향 설계원칙만 사용하여 재사용성이 높은 설계 결과를 얻을 수 있음을 확인하였다.

향후 Krueger의 추상화 원칙과 Two pass 추상화 원칙을 비교할 수 있는 Easiness, Effectiveness와 같은 정량화 가능한 평가 지표와 모델을 제안할 것이다.

참 고 문 헌

[1] 윤창섭, "소프트웨어 재사용과 설계에 관한 고찰", 「한국군사운영분석학회지」, 제15권, 제1호 (1989), pp.1-13.

[2] 최진명, 류성열, "패턴 기반 소프트웨어 개발을 위한 효과적인 패턴 선정 프로세스", 「정보과학회논문지」, 제32권, 제5호(2005), pp.346-356.

[3] Charles, W. Krueger, *Software reuse*, ACM Comput, Surv., Vol.24, No.2(1992), pp.131-183.

[4] Christin Ausnit, Christine Braun, Sterling Eanes, John Goodenough, Richard Simpson, *Ada Reusability Guideline*, ESD-TR-85-142, SoftTech Inc., 1985.

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[6] Robert C. Martin, Micah Martin, *Agile Principles, Patterns, and Practices in C#*, Prentice Hall, 2006.

[7] Dae-Kyoo Kim, Robert France, Sudipto Ghosh, Eunjee Song, "A Role-Based Metamodeling Approach to Specifying Design Patterns", *Computer Society*, (2003), pp.1-6.

[8] Elmar Juergens, Markus Pizka, "Variability Models Must Not be Invariant!", Variability Modeling of Software-intensive Systems, 2007.

[9] Garzas J. and M. Piattini, "From the OO Design Principles to the Formal Understanding

of the OO Design Patterns", OOPSLA, 2001.

[10] Garzas J. and M. Piattini, "Analyzability and Changeability in Design Patterns", OOPSLA, (2002), pp.33-43.

[11] Ghizlane El Boussaidi and Hafedh Mili, "A model-driven framework for representing and applying design patterns", Computer Society, 2007.

[12] Liskov, Barbara, "Data Abstraction and Hierarchy", SIGPLAN Notices, 1988.

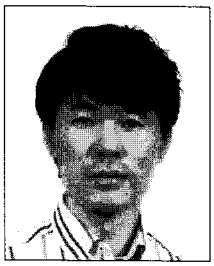
[13] Robert G. Lanergan and Charles A. Grasso, "Software Engineering with Reusable Design and Code", *IEEE Transactions on Software Engineering*, Vol.SE-10, No.5(1984), pp.498-501.

[14] Fowler, Martin, "Writing Software Patterns", 2006.

[15] Gabriel, Dick., "A Pattern Definition", http://hillside.net/patterns/definition.html, 2007.

[16] JHotDraw, http://www.jhotdraw.org/.

◈ 저 자 소 개 ◈



고 형 호 (hhko@korea.ac.kr)

한국방송통신대학교 컴퓨터학과 학사, 고려대학교 컴퓨터정보통신대학원 소프트웨어공학과 석사학위를 취득하였으며, 현재 미래로시스템에서 QFD와 TRIZ 소프트웨어 개발팀장과 아키텍트로 재직 중이다. 주요 관심분야는 소프트웨어 아키텍처, 디자인 패턴, 요구공학 등이다.

김 능 회 (nunghoi@korea.ac.kr)

성결대학교 컴퓨터및정보통신공학과 학사, 고려대학교 컴퓨터학과 석사학위를 취득하였으며, 현재 고려대학교 컴퓨터·전파통신공학 박사과정에 재학 중이다. 주요 관심분야로는 요구공학, 소프트웨어공학, 임베디드 소프트웨어 등이다.

이 동 현 (tellmehenry@korea.ac.kr)

고려대학교 전자공학과에서 학사, 컴퓨터학과에서 석사학위를 취득하였으며, 현재 동 대원 컴퓨터·전파통신공학과에서 박사과정으로 재학 중이다. 주요 관심분야는 요구공학, 소프트웨어공학, 임베디드 소프트웨어 등이다.

인 호 (hoh_in@korea.ac.kr)

고려대학교에서 전산학 전공으로 학사 및 석사, University of Southern California(USC)에서 Computer Science 전공으로 박사학위를 취득하였다. Texas A&M University에서 조교수를 역임하였으며, 현재 고려대학교 정보통신대학 컴퓨터전파통신공학 부교수로 재직 중이다. 주요 연구관심분야로는 요구공학, 소프트웨어공학, 임베디드 소프트웨어 등이다.