The Journal of The Institute of Internet, Broadcasting and Communication (IIBC)
Vol. 17, No. 1, pp.89-97, Feb. 28, 2017. pISSN 2289-0238, eISSN 2289-0246

https://doi.org/10.7236/JIIBC.2017.17.1.89
JIIBC 2017-1-12

안드로이드 디바이스 최적화를 위한 GOF 디자인 패턴 적용 방법에 대한 연구

A Study on the Application Method of GOF Design Pattern for Optimizing Android Devices

정우철*, 전문석**, 최도현***

Woo-Cheol Jung*, Mun-Seok Jeon, Do-Hyeon Choi*****

요 약 최근 사물인터넷(IoT), 웨어러블 등 PC이외에 휴대용 디바이스를 대상으로 OOP(Object-Oriented Programming)와 함께 GoF(Gang of Four)의 디자인 패턴 등 다양한 객체지향 방법론 설계를 기반으로 소프트웨어를 개발하고 있다. 그러나 잘못된 어플리케이션 설계는 운영체제 속도 저하, 메모리 점유율과 배터리 사용량 증가 등 문제를 발생시킬 수 있기 때문에 저 사양 디바이스에서의 프로그래밍 최적화의 중요도가 높다. 본 논문에서는 안드로이드(Android) 운영체제를 기반으로 Strategy패턴, State패턴, Observer패턴 등 최적화된 디자인 패턴 적용 방법을 제안한다. 테스트 결과 제안하는 패턴 선별 기법이 저 사양 디바이스에 최적화된 디자인 패턴을 선별할 수 있다는 것을 확인하였다.

Abstract Recent Internet of Things(IoT), and in addition to wearable PC, such as software development methodologies based on a variety of object-oriented design and design patterns of GoF(Gang of Four) with OOP(Object-Oriented Programming) intended for portable devices. However, incorrect application design specification is that the higher the importance of the optimization of the program on the device because it can cause problems such as decreased operating speed, increase the memory occupancy and battery usage. In this paper, we propose an optimized design pattern based on the method of application, such as Android (Android) OS Strategy Pattern, State Pattern, Observer pattern. Test results show that the proposed scheme selection patterns can be selected to optimize the design pattern in the device that specification.

Key Words : Android, GoF, Design Pattern, OOP, Refactorying

Ⅰ. 서 론

현재 모바일 디바이스 시장은 애플 아이폰(iPhone)과 구글 안드로이드 플랫폼을 중심으로 다양한 소프트웨어가 연구 및 개발되고 있는 실정이다. 스마트폰 소프트웨어 개발 열풍은 하드웨어의 성능향상과 생산단가 하락으로 인해 더욱더 가속화 되고 있다. IoT와 같은 디바이스 시장에서는 관련 소프트웨어를 개발할 수 있는 환경이 조성되어 관심이 증대되고 있다[1]. 일반적으로 소프트웨어의 개발은 기획단계, 개발단계, 인도단계를 거쳐서 완성이 된다. 소프트웨어를 효율적으로 개발하기 위해서는 건물을 공사할 때 설계도면을 그리는 것에 해당하는 기

*정회원, 숭실대학교 컴퓨터학과
**정회원, 숭실대학교 컴퓨터학과
***정회원, 숭실대학교 컴퓨터학과
접수일자 2016년 11월 11일, 수정완료 2017년 1월 17일
게재확정일자 2017년 2월 3일

Received: 11 November, 2016 / Revised: 17 January, 2017 / Accepted: 3 February, 2017
***Corresponding Author: cdhgod0@ssu.ac.kr
Dept of Computer Engineering Soongsil University, Korea

획단계에서의 설계 활동이 매우 중요하다. 모바일 환경의 디자인 패턴은 그동안 다양한 소프트웨어 개발 방법론이 개발되어 활용되었음에도 불구하고 대규모 프로젝트의 설계와 유지보수를 고려하여 최적화된 설계 방법에 대해서는 연구가 부족한 실정이다[2].

본 논문에서는 기존의 OOP를 이용한 설계 방법론에서 GoF의 디자인 패턴 중 Strategy, State, Observer 패턴을 선별하여 모바일 성능 효율성에 중점을 둔 디자인 패턴 적용 기법을 제안한다.

본 논문은 모두 5장으로 구성된다. 2장에서는 소프트웨어 설계방법의 전반적인 기술 개요와 OOP 기반 디자인 패턴을 설명하고 3장에서는 안드로이드 환경에서 GoF의 디자인 패턴을 이용한 패턴 추출 기법을 제안하였다. 4장에서는 구현 및 테스트하고 5장은 결론과 향후 연구 방향을 제시하였다.

II. 관련연구

1. OOP 기반 디자인 패턴

디자인 패턴은 개발자들이 유용하다고 생각하는 클래스 내에 객체들 간의 역할을 정의하고 상호작용 방법 등 문제에 대한 해결방법, 결과와 장단점을 제시한다[3]. Christopher Alexander는 “각 패턴은 개발 환경 내에서 반복적으로 발생하는 문제점을 기술하고, 그 다음에 문제에 대한 핵심적인 해법을 기술한다.”라고 하였다[4]. 1987년 Ward Cunningham과 Kent Beck은 Alexander의 연구의 발전된 개념으로 OOPSLA(Object-Oriented Programming. Systems. Languages & Applications)을 발표하였고, 이후 Erich Gamma1이 1993년 설계패턴을 체계화하여 Object-Oriented Design Pattern을 발표하였다[5][6].

디자인 패턴을 활용하면 일반적인 소프트웨어 개발에서 나타나는 복잡한 문제를 구조화 하여 설계 할 수 있다[7]. 요구사항을 분석하여 프로그램을 객체지향 방법론으로 디자인하는 것은 쉽지 않다. 특히, 재사용성이나 확장성을 고려않은 설계의 경우 변동 사항이나 추가 요구 사항을 처리할 때 다시 디자인해야 하는 최악의 경우가 발생한다. 따라서 자신이 설계하여 어느 정도 검증된 디자인 구조를 새로 개발할 솔루션에 재사용하여 적용하는 것이 일반적이다.

디자인 패턴을 적용하는 것의 장점은 첫째, 요구사항의 분석과 설계를 명확하게 할 수 있고, 소스 코드에 대한 리팩토링(Refactorying)이 가능하게 하여 프로그램 코드상의 모호성과 중복을 제거하여 소프트웨어를 쉽고 유연하게 확장할 수 있다. 둘째, 객체지향 분석과 설계 시 공통된 모델링 언어로 UML이 사용되는 것과 같이 클래스를 공통된 설계 언어로써 사용될 수 있다. 셋째, 패턴을 이용한 리팩토링 방법을 통해 객체지향 언어로 이미 작성되어 있는 레거시(Legacy) 시스템을 효과적으로 재공학(Reengineering)할 수 있다[8].

2. GOF의 디자인 패턴

GoF의 디자인 패턴은 총 23개의 패턴으로 구성되며 각 패턴들의 특징 및 목적에 따라 생성 패턴, 구조 패턴, 행위 패턴으로 그룹화 된다[9]. GoF는 디자인 패턴을 목적에 따라 생성패턴, 구조패턴, 행위패턴으로 분류한다[10]. 다음은 일반적으로 활용도가 높은 GoF의 디자인 패턴 10개를 나타낸다.

① 추상 팩토리(Abstract Factory) : 구체적인 클래스에 의존하지 않고 서로 연관되거나 의존적인 객체들의 조합을 만드는 인터페이스 제공
② 팩토리 메서드(Factory method) : 객체생성 처리를 서브클래스로 분리하여 처리하도록 캡슐화
③ 싱글턴(Singleton) : 전역변수를 사용하지 않고 하나의 객체만 생성하며 어디에서든 참조
④ 컴퍼지트(Composite) : 여러 객체들로 구성된 복합 객체와 단일 객체를 구별 없이 제어
⑤ 데커레이터(Decorator) : 객체의 결합을 통해 기능을 동적으로 유연하게 확장
⑥ 옵서버(Observer) : 객체 상태 변화에 따라 다른 객체의 상태도 연동되도록 일대다 관계를 구성
⑦ 스테이트(State) : 객체 상태에 따라 객체의 행위 내용을 변경 가능
⑧ 스트래티지(Strategy) : 행위를 클래스로 캡슐화 하여 동적으로 행위를 변경 가능
⑨ 템플릿 메서드(Template Method) : 특정 작업의 일부분을 서브클래스로 캡슐화 하여 전체구조는 변경하지 않으면서 특정 작업을 변경 가능
⑩ 커맨드(Command) : 실행할 기능을 캡슐화 하여 주어진 여러 기능을 실행할 수 있는 재사용성 제공

3. 소프트웨어 개발과 리팩토링

소프트웨어 개발 비용에는 유지보수, 요구분석, 명세, 설계, 개발, 시험 및 검증, 통합 등의 요소가 존재한다. 그 중에서 유지보수 비용이 전체 개발 비용의 70% 수준이기 때문에 체계적 관리가 필요하다[11]. 소프트웨어의 규모가 커질수록 유지보수가 차지하는 비중이 커져 점점 더 어려워지게 된다. 리팩토링은 프로그램의 행위(behavior)를 보전하면서 프로그램의 가독성, 구조, 성능, 유지보수성, 추상성 등을 향상시키는 방법이다. 소프트웨어 개발자는 리팩토링을 사용하여 프로그램을 이해하기 쉽고 변화를 포용할 수 있도록 프로그램을 단계적으로 개선 가능하다[12].

리팩토링은 디자인 패턴과는 다르게 소프트웨어를 개발하는데 있어서 구현 완료된 코드를 수정하여 이해하기 쉽고 잘 구조화된 소프트웨어를 만드는 것이 목적이다. 즉 전체 구조적인 부분과 구현 코드를 단계적으로 개선하여 완성해 가는 것이다. 그림 1과 같이 프로그램 설계 단계에서 디자인 패턴과 리팩토링을 사용할 경우 프로그램의 개발 및 업그레이드의 유연성을 가진다[13].

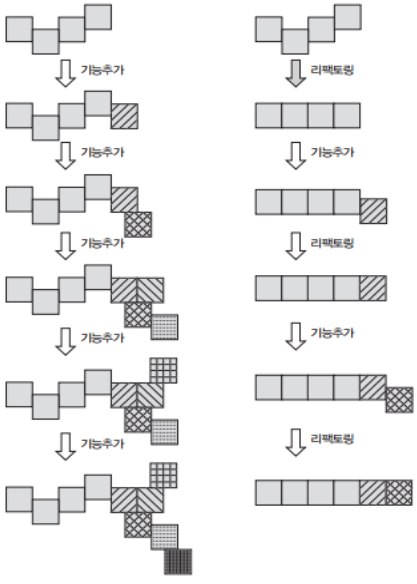

그림 1. 기능추가 반복과 리팩토링의 적용에 따른 변화
Fig. 1. Functional Repetition and Changes due to Applying Refactoring

리팩토링의 저자 M. Fowler는 Code Smell을 소스코드를 이해하기 어렵게 하고 소프트웨어에 잠재적인 문제를 야기할 수 있는 요인으로 정의하였다. 그리고 그 Code Smell을 제거하기 위해 리팩토링을 수행해야 한다고 주장하였다. 또한 Code Smell 중에서 메서드의 내부가 너무 길 경우에 발생하는 Long Method를 리팩토링 하는 기법으로 메서드 추출(Extract Method)을 제안하였다[14]. 기존 여러 메서드를 하나의 메서드로 추출하여 Code Smell를 제거한다.

4. OOP 지향 설계 기법

Jacobson의 OOSE(Object Oriented Software Engineering)는 Use Case모델을 중심으로 한 객체 지향 분석 및 설계 방법이다. Use Case란 시스템을 구성하는 단위기능이며, 모든 쓰임새의 집합은 결국 시스템의 완전한 기능을 설명하게 된다[15]. 객체지향 프레임워크는 어플리케이션 구축을 위한 구현 골격을 제공하기 위해 협력하는 추상 클래스(Abstract Class)와 실제 클래스(Concrete Class)들의 집합이다[16]. 최근 객체지향 설계는 UML이라는 다이어그램 도구를 이용하는데 UML 기반의 객체지향 설계 절차는 다음과 같다[17].

① Use Case 다이어그램을 작성한다 — 각 쓰임새의 시나리오를 작성한다.
② 객체를 식별하여 클래스 다이어그램을 작성한다.
③ 순차 다이어그램을 작성한다. : 메서드 식별
④ 메서드의 호출을 연계하여 상세 클래스 다이어그램을 작성한다.

OOP은 객체 지향적으로 소프트웨어를 설계하기 위한 접근법이다. 3단계로 이루어진 요구사항 분석, 컴포넌트 식별, 디자인 패턴 선정 이후 설계모델을 작성하게 된다. 분석단계는 어플리케이션 요구사항을 분석한 후 식별된 컴포넌트의 기능성을 고려하여 설계 패턴들을 선정하게 된다. 그림 2는 분석단계부터 설계단계까지 전체 OOP 흐름도를 나타낸다.

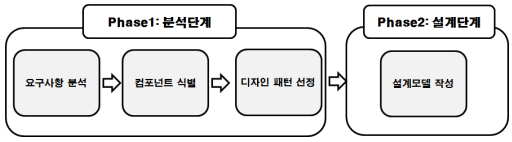

그림 2. 분석단계부터 설계단계까지 전체 OOP 흐름도
Fig. 2. Overall OOP flow from Analysis phase to Design phase

본 논문에서는 OOP를 기반의 접근법을 위해 선정된 디자인 패턴과 최적화된 클래스 다이어그램을 재설계하였고, UML로 작성하였다.

Ⅲ. GoF의 디자인 패턴 적용 기법

1. 제안하는 디자인 패턴의 전체 흐름도

그림 3은 디자인 패턴 적용을 위한 전체 알고리즘 흐름도를 나타낸다.

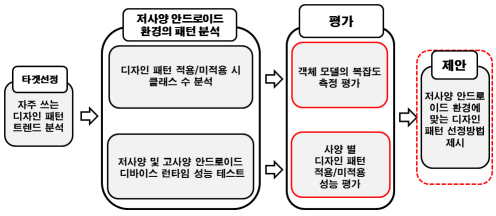

그림 3. 제안 기법의 전체 알고리즘 흐름도
Fig. 3. Overall Algorithm flow of Proposed Method

총 23개 GoF 디자인 패턴에서 통계적으로 구글 검색에서 사용자의 관심도가 높은 디자인 패턴 대상을 선정하였다. 그림 4는 구글에서 디자인 패턴에 대한 트렌드를 분석한 결과이다. 그림 5는 검색빈도와 패턴 중요도에 따른 선정 기준을 나타낸다.

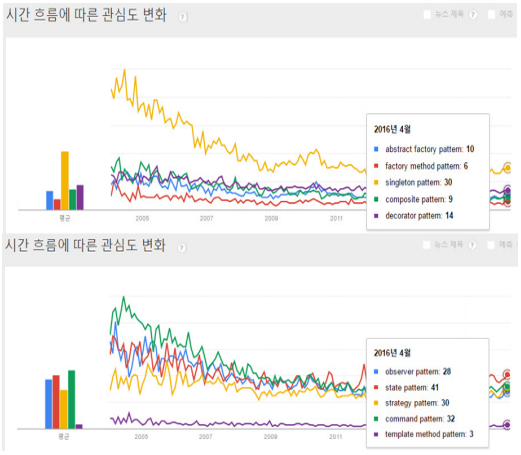

그림 4. 구글 트렌드(디자인 패턴) 키워드 분석 결과
Fig. 4. Google Trends (Design Pattern) Keyword Analysis Results

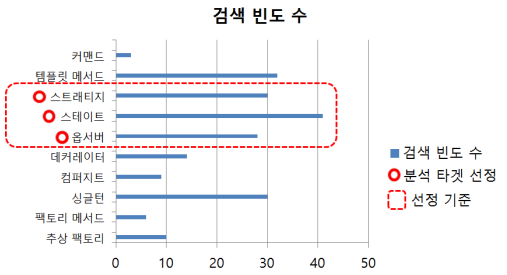

그림 5. 검색빈도와 패턴 중요도에 따른 선정
Fig. 5. Selection according to Search frequency and Pattern Importance

분석결과 빈도수가 높은 Strategy, State, Observer 패턴이 선정되었다.

2. 디자인 패턴 분석 및 최적화

선정된 Strategy, State, Observer 패턴은 안드로이드 환경에 적절하게 최적화되어 적용되어야 한다. 이후 적용 유무에 따른 성능변화를 분석하여 개별 평가를 진행한다.

가. Strategy 패턴

Strategy 패턴은 목적을 달성하기 위해 비즈니스 규칙이나 알고리즘 등의 전략을 쉽게 바꿀 수 있는 디자인 패턴이다. 예를 들어 로봇의 기능제어, 기능변경, 물건 등을 판매할 때 일반세일과 특가 세일 등 가격정책을 변경하고자 할 때 유용하다. 그림 6은 OOP의 방법으로 로봇 클래스를 설계한 것이다.

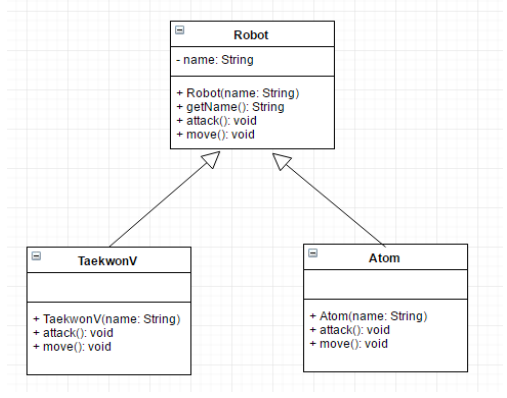

그림 6. Strategy 패턴 적용 전 로봇 클래스
Fig. 6. Robot class before Applying Strategy Pattern

Robot 클래스의 move 메서드는 날 수 없는 로봇과 날을 수 있는 로봇으로 구분할 수 있다. attack메서드는 근거리 펀치 공격과 원거리 미사일 공격으로 성격을 달리 할 수 있다. 변화가 쉬운 부분은 Moving Strategy와 Attack Strategy 인터페이스로 캡슐화를 하는 것이 변화를 쉽게 처리할 수 있게 된다. 그림 7은 리팩터링을 적용하여 Strategy 패턴으로 변경 설계된 결과를 나타낸다.

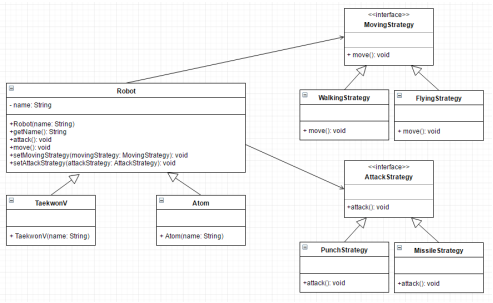

그림 7. Strategy 패턴 적용 후 로봇 클래스
Fig. 7. Robot class after Applying Strategy Pattern

나. State 패턴

State 패턴은 어떤 행위를 수행할 때 상태에 행위를 수행하도록 위임한다. 이를 위해 시스템의 각 상태를 클래스로 분리해 표현하고, 각 클래스에서 수행하는 행위들을 메서드로 구현한다. 그리고 이러한 상태들을 외부로부터 캡슐화하기 위해 인터페이스를 만들어 시스템의 각 상태를 나타내는 클래스로 실체화한다. 그림 8은 디자인 패턴 적용 전으로 형광등 스위치 상태를 On/Off로 변경하는 클래스를 설계한 결과를 나타낸다.

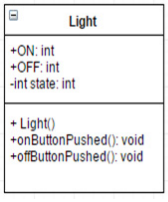

그림 8. State 패턴 적용 전 스위치 클래스
Fig. 8. Switch class before State Pattern Applied

설계된 클래스를 코드를 구현하면 On/Off의 상태별 하드코드와 함께 이루어져 있어 나쁜 객체지향 설계법이며 스위치의 상태변화가 추가될 경우 대처하기 어렵게 된다. 예를 들어 전원 스위치는 On/Off 외에도 무드등 혹은 등의 밝기를 조절할 수 있는 시제품들이 많이 나와 있다. 버튼 하나만을 눌러 상태를 변경하고자 할 경우 State 패턴을 활용하여 전등의 상태를 쉽게 변경할 수 있다. 그림 9는 리팩토링을 적용하여 State 패턴으로 변경 설계된 결과를 나타낸다.

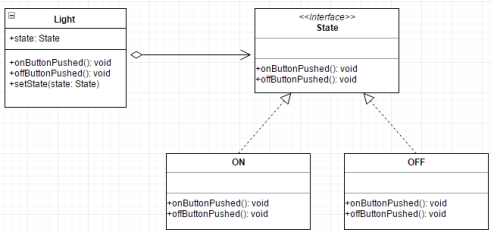

그림 9. State 패턴 적용 후 스위치 클래스
Fig. 9. Switch class after Applying State Pattern

Light 클래스에서 State 클래스에 작업을 위임하여 변화에 유용하면서도 객체지향적인 설계로 변경되었다. 또한 상태 진입을 각자 상태에서 처리하므로 if문이나 switch문을 사용해 상태 변화를 나타낼 필요가 없다.

다. Observer 패턴

Observer 패턴은 데이터의 변경이 발생했을 경우 상대 클래스나 객체에 의존하지 않으면서 데이터 변경을 통보하고자 할 때 유용하다. 예를 들어 탐색기 기능이나 성적 출력, 연료량이 부족하면 통보하는 등의 여러 형태에 적용할 수 있다. 그림 10은 디자인 패턴 적용 전으로 성적표를 출력하는 클래스를 설계한 결과를 나타낸다.

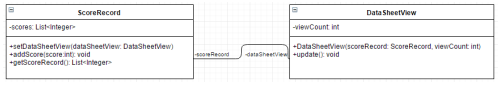

그림 10. Observer 패턴 적용 전 성적표 클래스
Fig. 10. Report card before Applying Observer Pattern

설계된 클래스를 코드를 구현하면 성적을 목록이 아닌 최소/최대값을 출력하는 등 소스코드의 많은 변화가 필요하게 된다. 성적의 최소값과 최대값에 해당하는 MinMaxView 클래스를 추가하여 보여주고 싶을 경우에도 ScoreRecord 클래스를 그대로 재사용 하려면 변화하는 부분을 미리 식별할 수 있다. 그림 11은 리팩토링을 적용하여 Observer 패턴으로 변경 설계한 결과를 나타낸다.

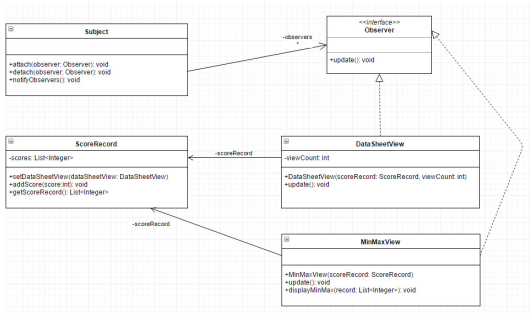

그림 11. Observer 패턴 적용 후 성적표 클래스
Fig .11. Observer Pattern applied Report Card Class

변경 결과 성적변경에 관심이 있는 대상 객체를 관리하는 기능을 구현하는 Subject라는 클래스를 정의한 것을 확인할 수 있다. Subject 클래스는 attach와 detach메서드로 다음과 같이 성적 변경에 관심이 있는 대상 객체를 추가하거나 제거할 수 있는 유연성을 제공한다.

Ⅳ. 테스트 및 평가

1. 시스템 구현 환경

Strategy, State, Observer 패턴을 사용하여 성능을 측정할 수 있는 테스트 시스템을 구축하고, 패턴 별 성능을 테스트 및 평가한다. 표 1, 2는 본 논문에 사용된 시스템 구현 환경과 테스트 환경을 나타낸다.

표 1. 시스템 구현 환경
Table 1. System Implementation Environment

구분	구성요소	세부내용
개발 환경	CPU	Intel 2.4GHz Core2 Duo P8600
	RAM	8GB 1077MHz DDR3
	OS	Windows 10
	Language	Java
	Development Kit	JDK 1.6
	Environment	Android Developer Tools(Eclipse)

표 2. 테스트 환경
Table 2. System Implementation Environment

구분	구성요소	세부내용
테스팅환경	Target	Android 4.4.2 - API Level 19
	Screen	480x800 (HDPI)
	CPU	ARM (armeabi-v7a)
	RAM	256MB
	Internal memory	224MB
	VM Heap	32MB
	Debug Tool	DDMS(Dalvik Debug Monitor Server)

2. 최적화된 디자인 패턴 알고리즘 구현

그림 12는 본 논문에서 제안한 GoF의 디자인 패턴 적용 기법을 증명하기 위한 알고리즘 흐름도를 나타낸다.

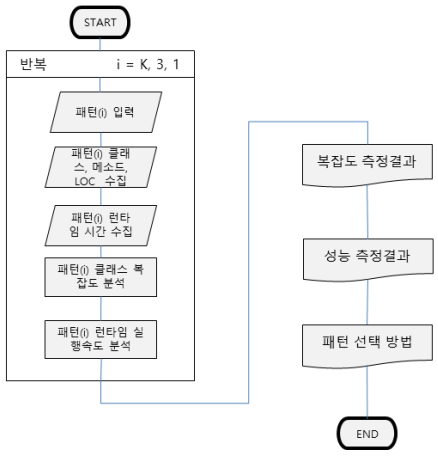

그림 12. 패턴분석을 위한 알고리즘 흐름도
Fig. 12. Algorithm flow Chart for Pattern Analysis

패턴 (i)에 해당하는 것은 GoF의 디자인 패턴 중 선별한 Strategy, State, Observer 패턴을 의미하며 i = k, 3, 1은 각각 패턴을 1회씩 총 3회 수행하여 클래스 개수, 메서드의 개수, LOC 정보를 수집한다. 또한 구현한 코드를 실행한 후 실행 시작시간부터 실행종료 시간까지 걸린 시간을 측정하여 해당 패턴의 수행시간, 클래스의 오브젝트 크기와 실행 속도를 분석한다. 이후 분석 결과인 오브젝트 크기 측정결과와 성능 측정결과를 그래프로 비교 분석하였다. 마지막으로 저 사양 안드로이드 환경에서 성능 최적화를 위한 디자인 패턴을 선택하는 방법을 제시하는 절차로 구성된다. 그림 13은 각 클래스 구현화면을 나타낸다.

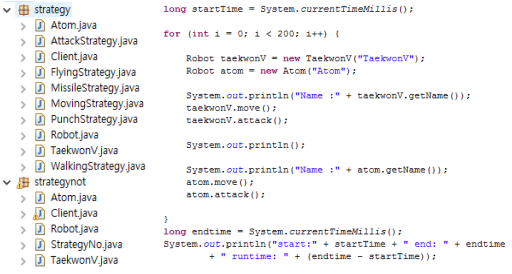

그림 13. 클래스 구현화면
Fig. 13. Class Implementation Screen

3. 평가 방법

성능평가를 위하여 평가지표로는 Strategy, State, Observer 미적용 클래스들과 디자인 패턴 적용 클래스간의 수행 시간을 간단히 프로파일링(profiling)하였다. 측정 방법으로 시간 복잡도(Time Complexity)를 직접 측정하였다. 시간 복잡도는 프로그램을 실행시켜 완료하는데 걸리는 시간을 의미한다. 테스트용 코드의 실행 전 시작시간(Start Time)과 작업이 끝난 후 종료시간(End Time)을 측정하여 종료시간에서 시작시간의 차를 구하면 해당 로직의 수행 시간이 되므로 성능측정 결과인 ET(Execution Time)을 알 수 있다. 수식 (1)은 수행 시간의 계산방법을 나타낸다.

$$ET = End\ Time - Start\ Time \quad (1)$$

또 하나의 평가방법으로 OFS(Object File Size)를 알아내는 것이다. 성능을 측정하는 2가지 요소는 시간과 공간이다. CPU 자원의 사용시간인 PE(Processor Equivalent)와 메모리의 사용량을 알 수 있는 OFS를 분석하여 해당 로직의 성능을 평가한다.

4. 성능평가

성능평가에서는 두 요소인 ET와 OFS, PE를 측정하여 각각의 패턴들에 대한 성능을 비교분석 한다.

가. Execution Time

표 3과 그림 14는 Strategy, State, Observer 패턴의 미적용 로직과 디자인 패턴이 적용된 로직의 최장시간, 최소시간, 평균시간을 비교한 결과를 나타낸다.

표 3. 실행 시간 테스트 결과
Table 3. Execution Time Test Results

구분	테스트 수	최장시간 (ms)	최소 시간 (ms)	평균 시간 (ms)
Strategy 미적용	1000	5628	5325	5501
Strategy 적용	1000	4384	4021	4169
State 미적용	1000	4381	3992	4123
State 적용	1000	4199	3788	3962
Observer 미적용	1000	1452	1135	1322
Observer 적용	1000	1518	1155	1369

Strategy와 State 패턴은 디자인 패턴을 적용하면 속도가 향상되었으며 Observer 패턴은 디자인 패턴을 적용하면 속도가 더 느려짐을 알 수 있다. 따라서 성능 향상을 위해서는 Strategy와 State 패턴을 적용하는 것이 유용하며 Observer 패턴은 성능이 중요하다면 적용하지 않는 방법이 유용할 것이다.

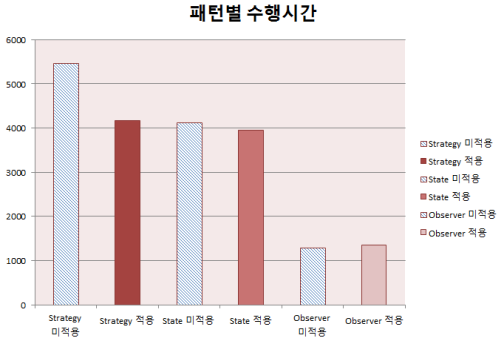

그림 14. 실행 시간 테스트 결과(차트)
Fig. 14. Execution Time Test Results(Chart)

나. Object File Size

표 4와 그림 15는 Strategy, State, Observer 패턴의 적용과 미적용에 따른 오브젝트 파일 크기를 비교한 표이다.

표 4. OFS 테스트 결과
Table 4. OFS Test Results

패턴	OFS (byte)
Strategy 미적용	16739
Strategy 적용	14562
State 미적용	21998
State 적용	23001
Observer 미적용	9815
Observer 적용	8982

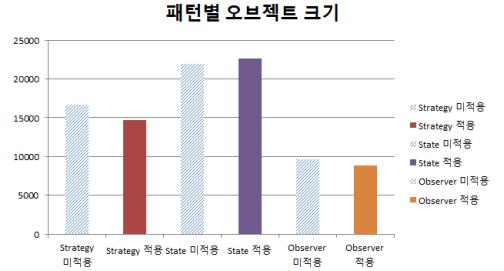

그림 15. OFS 테스트 결과(차트)
Fig. 15. OFS Test Results(Chart)

Strategy와 Observer 패턴은 디자인 패턴을 적용하면 OFS가 감소하고 State 패턴은 OFS가 증가함을 알 수 있다. 종합적인 분석 결과 Strategy 패턴을 사용하면 속도가 증가하고 OFS가 감소하여 최대한 적용하는 것이 효율적이고, State 패턴을 적용하면 속도는 증가하나 OFS도 증가하므로 저사양의 메모리는 공간적인 부분을 충분히 고려해야 할 것으로 분석되었다.

다. Performance Estimation

표 5는 Execution Time과 Object File Size 테스트를 수행한 결과를 바탕으로 최적화된 디자인 패턴 적용 기법을 나타낸다. 성능을 중점으로 소프트웨어 설계할 경우 Strategy와 State 패턴이 적절하고, 저 사양 메모리를 중점으로 설계 할 경우 Strategy와 Observer 패턴을 사용하는 것이 가장 적절한 방법임을 알 수 있다.

표 5. 디자인 패턴 별 종합 비교 분석 결과
Table 5. Comprehensive Analysis of Design Patterns

구분	ET	OFS	저 사양 속도 적용	저 사양 메모리적용
Strategy 적용	증가	감소	적용	적용
State 적용	증가	증가	적용	비적용
Observer 적용	감소	감소	비적용	적용

종합적인 비교 분석 결과 기존의 패턴 적용에 대한 인식과 다른 것을 확인하였다. 일반적으로 대부분의 디자인 패턴을 적용하면 유지보수성은 높아지지만 성능저하가 발생할 것이라고 생각한다. 하지만 실제 테스트 결과 패턴 분류에 따라 성능이 저하되는 경우도 있고 높아지는 부분도 있음을 확인할 수 있다.

V. 결 론

본 논문에서는 효율적인 디자인 패턴 적용을 위해 첫째, OOP의 방식으로 클래스를 설계하며 둘째, 사용자의 관심도가 높은 디자인 패턴을 기반으로 리팩토링하여 이를 최적화 시켰다. 셋째, OOP와 디자인패턴이 적용된 클래스와 메서드 개수, LOC(Line Of Code), 런타임 시간, 오브젝트 크기와 성능을 측정하여 이를 비교 분석하였다. 성능분석 결과 OOP와 디자인패턴 중 성능이 효율적이고 오브젝트 크기가 낮은 패턴을 혼합하여 사용하는 것이 가장 효율적이라는 것을 확인하였다.

본 논문의 연구 결과는 기존의 패턴을 보던 관점과 달리 패턴을 적용할 경우 시스템의 성능변화를 연구한 것에서 의의를 찾을 수 있다. 향후 연구로는 Strategy, State, Observer 패턴 이외에 GoF의 23가지의 모든 패턴들에 대해서 테스트를 확장하고, IoT 장비 등과 같은 제한적인 환경에서 성능을 분석할 계획이다.

References

[1] Park Minwoo, “Education of Domestic Programming and Future of The Software Industry”, Digieco Report Issue&Trend, 2014.

[2] Kim Unyong, Choe Yeonggeun, “Special Issue: Software Quality : Pattern-Oriented Software Development Process using Incremental Composition for Design Patterns”, Korea Information Processing Society, Vol.10, No.5, pp.763-772, 2003.

[3] Kang Yunsung, Lee Junhwan, Cho Hanjin, “Design and Implementation of .NET Remoting Common Framework Applied Design Pattern”, Korea Contents Association, Vol.11, No.3, pp.36-47, 2011.

[4] C.Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I.Fiksdahl-King, S. Angel, “A Pattern Language”, Oxford University Press, NewYork, 1997.

[5] Lee Jangwoo, Lee Minkyu, “Design and Implementation for Applying User-Definable Pattern with UML Modeling Tools”, Korea Information Science Society Fall Conference Proceeding, Vol.31, No.2, pp.310-312, 2004.

[6] Shin Woochang, “Study on Formal Software Design Patterns”, Institute of Industrial Technology Seokyeong University, Vol.17, No.0, pp.71-83, 2006.

[7] Kim Taeho, Cheon Hyeonjae, Lee Hongchul, “Development of Secure Entrance System using AOP and Design Pattern”, Korea Academia-Industrial Cooperation Society, Vol.11, No.3, pp.943-950, 2010.

[8] Choi Jinmyung, Rhew Sungyul, “An Effective Pattern Selection Process for Developing of Pattern Based Software”, Korea Institute of Information

Science and Engineering, Vol.32, No.5, pp.346-356, 2005.

[9] Kim Moonkwon, "Methods to Apply GoF Design Patterns in Service-Oriented Computing", Korea Information Processing Society Transactions. Part D, Vol.19, No.2, pp.187-202, 2012.

[10] E. Gamma, R.Johnson, J. Vlissides, "Design Pattern: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995

[11] C. Zhao, J. Kong, K. Zhang, "Program Behavior Discovery and Verification: A Graph Grammar Approach", IEEE Transactions on Software Engineering, Vol.36, Issue.3, pp.431-448, 2010. DOI: https://doi.org/10.1109/TSE.2010.3

[12] Jung Youngae, "Comparative Analysis of Determination of Method Location between Classes", Korea Contents Association, Vol.6, No.12, pp.80-88, 2006.

[13] Son Hyunseung, Kim Wooyeol, Ahn Hongyoung, Kum Youngchul, "Applying Design Pattern & Refactoring on Implementing RTOS for the Small Educational Multi-Joint Robot", The Journal of The Institute of Internet, Broadcasting and Communication(JIIBC), Vol.9, No.3, pp.217-224, 2009.

[14] M. Fowler, "Refactoring: Improving the Design of Existing Code", Addison Wesley, 1999.

[15] Ivar Jacobson, "Object Oriented Software Engineering : A Use Case Driven", ACM Press, 1992.

[16] Cho Eunsook, Kim Soodong, Rhew Sungyul, "UML-based Object-Oriented Framework Modeling Techniques", Korean Institute of Information Science and Engineering Transactions Part B Vol.26, No.4, pp.533-545, 1999.

[17] Kung Sanghwan, "Smart Design for App", Korea Digital Policy & Management, Vol.10, No.6, pp.269-274, 2012.

저자 소개

정 우 철(정회원)

- 2008년 2월 : 동서울대학교 컴퓨터소프트웨어학과 졸업
- 2010년 8월 : 숭실대학교 컴퓨터학과 석사 졸업
- 2016년 3월 : 숭실대학교 컴퓨터학과 박사 수료

<주관심분야 : Mobile, Network Security, Big data, Deep Learning>

전 문 석(정회원)

- 1981년 : 숭실대학교 전산학과 학사
- 1986년 : University of Maryland 전산학과 석사
- 1989년 : University of Maryland 전산학과 박사
- 1989년 ~ 1991년 : New Mexico State University Physical Science Lab 책임연구원
- 1991년 ~ 현재 : 숭실대학교 컴퓨터학과 정교수

<주관심분야 : 정보보안, PKI, 전자여권, 암호학>

최 도 현(정회원)

- 2008년 2월 : 동서울대학교 컴퓨터소프트웨어학과 졸업
- 2010년 8월 : 숭실대학교 컴퓨터학과 석사
- 2016년 3월 : 숭실대학교 컴퓨터학과 박사

<주관심분야 : Mobile, Network Security, PKI, Virtualization>