행위 모델링 및 조합을 위한 객체지향성

김희언
세종대학교 컴퓨터공학과
hiun@divtag.sejong.edu

Object-orientation for Behavior Modeling and Composition

Hiun Kim
Computer Science Department, Sejong University

요 약

현대 소프트웨어가 가지는 기능의 수와 복잡성은 지속적으로 증가하고 있으며 이를 정확하게 모델링하고 높은 모듈성을 가지도록 구현하는 것이 중요한 요소이다. 에스펙트 지향 프로그래밍은 횡단적 관심사를 에스펙트 객체에 지역화시킴으로써 모듈성을 구현하였지만, 에스펙트 객체 자체의 유연성이 낮아 고차적인 재사용에는 제한적이었다. 본 연구에서는 횡단적 관심사들의 고차적인 재사용성을 확보하기 위해 객체지향성을 소프트웨어 행위의 모델링 및 조합에 적용하는 방법을 제시하며, 행위에 계층적 관계를 부여하는 접근을 통해 추상적인 부모 행위와 구체적인 자식 행위라는 개념을 제안하고, 공통된 횡단적 관심사를 부모 행위에 지역화시켜서 재사용성을 향상시켰다. 이를 유연하게 지원하기 위해 자가조합성이라는 개념을 도입, 다른 행위와의 조합과 함께, 행위 스스로 개선을 통해 고차적인 재사용성을 확보하는 방법을 제안하였다. 우리는 본 연구에서 AOP와 비교하여 웹서비스를 대상으로 본 기법의 효과성을 검증하였으며 행위가 객체와 분리된 존재로 자체적인 계층적 관계를 가짐으로써 더욱 정확하게 모델링되고 쉽게 조합되어 소프트웨어의 모듈화에 기여할수 있다는 것을 제시하였다. 본 연구의 궁극적인 목적은 객체지향프로그래밍의 동기를 따라서 실제 세상의 행위를 프로그래밍 레벨에서 모델링하는 것의 중요성을 자가조합프로그래밍이라는 방법을 통해 제시하며, 그 필요성을 연구자들에게 제시하는 데 있다.

1 서론

현대 소프트웨어는 지속적으로 그 기능(feature)[1]의 수와 기능의 수준이 증가하고 있으며, 결과적으로 하나의 소프트웨어 안에서도 기능간의 가변성(variability)이 증가하였다. 이러한 가변성을 관리하고 지속 가능하게 소프트웨어를 발전시킬 수 있는 요소들로 재사용성(reusability), 유연성(flexibility), 이해성(comprehension)등 이 있으며 [2],. 이러한 요소들을 충족시키기 위해서는 소프트웨어 기능의 모듈성(modularity)은 중요한 요소이다[3]. 프로그래밍 레벨에서 주요한 모듈화 방법인 에스펙트지향프로그래밍(Aspect-oriented Programming; AOP)[4]은 소프트웨어의 기능들과 뭉쳐있고(tangled) 따라서 전역적으로 산포되어(scattered) 있는 공통된 기능인 횡단적 관심사(cross-cutting concern) 모듈화시켜 재사용성과 이해성을 부여하였으나, 유연한 고차적 재사용성의 어려움으로 인해 횡단적 관심사가 일부분만 달라져도 이를 반영하기 위한 직접적 방법이 제한적이라는 것이다. 상속을 통해 이를 해결하고자 하는 방법은 존재하나[5, 6], 직접적인 문법적 지원의 필요성은 여전히 존재한다. 우리는 AOP을 비롯한 기존 방법들과 다르게, 객체를 중심의 사고가 아닌, 독립적인, 객체의 행위를 중심으로 한 사고를 통해 가변성의 문제를 바라보았으며, 소프트웨어의 내부적 행위들에게 관계를 부여하여 유연한 고차적 재사용을 통한 가변성의 관리를 하고자 한다.

1) 본 연구에서 기능(feature)은 Kang et al. [1]의 정의에 따라, '사용자적 관점에서 주요하고 구분될 수 있는 소프트웨어의 품질과 특성 그리고 성질'의 의미로 사용된다. 이와 대조적으로 행위(behavior)는 '소프트웨어의 관점에서 기능이 구현되기 위한 작업'으로 사용되며, 여러 행위가 하나의 기능을 만들 수도 있다.

1.1 현대소프트웨어의 행위적 유사성

우리는 높은 수준의 행위를 하는 현대 소프트웨어를 개발하면서 다양한 가변성을 마주하였다. 현대 소프트웨어를 모듈화하기 어려운 이유는 기능을 구현하는 행위들의 공통성(commonalities)이 적어지고 가변성이 늘어났기 때문이다. 우리는 가변적인 행위들이 공통적이지 않지만 유사(similar)하다는 점을 확인할 수 있었고 이를 행위적 유사성(behavioral similarity)이라 특정하였다. 대표적 예로는 네트워크 연관 소프트웨어가 있다. 시스템 소프트웨어(i.e. OS)와 다르게 이러한 소프트웨어들은 *상대적*으로 많은 종류의 그러나 적은 복잡성을 가진 모듈들의 집합으로 이루어져 있다(i.e. API 서버). 따라서 이러한 모듈들은 네트워크 연관 소프트웨어에서 큰 비중을 차지하며, 이들을 구성하는 횡단적 관심사인 인증이나 캐싱, 데이터 검증 등 하위 행위들이 모듈 간에 비슷한 패턴으로 실행되는 행위적 유사성을 보인다. 또한, 서비스 지향 아키텍처(Service-oriented Architecture; SOA)[7]등 네트워크를 기반으로 느슨하게 연결되어 동작하는 소프트웨어들은 기능의 더 많은 부분을 네트워크상의 다른 소프트웨어에게 의존하게 되므로[3] 이들의 정확한 작동을 위해 더 많은 - 행위적 유사성을 가진 횡단적 관심사들이 비슷한 패턴으로 사용될 것이다. 따라서 네트워크 연관성의 증가와 함께, 행위적 유사성은 가변성과 더불어 현대 소프트웨어의 주요한 성질로 자리 잡게 될 것이다.

1.2 애스펙트지향 접근법

현대 소프트웨어의 가변성을 관리하기 위해, AOP는 하나의 모듈을 핵심 관심사(core concern)와 횡단적 관심사로 분해(de-

compose)하고, 소프트웨어의 모듈 곳곳에 산포되어있는 횡단적 관심사를 함수와 해당 관심사의 연결 지점(pointcut)정보를 가진 에스펙트 객체로 모듈화시켜서, 이를 전후처리 등의 접합점(join point)에 조합하여 사용하는 방법을 제안하였다[8]. 웹서비스를 예로 들면, 핵심관심사는 *글쓰기*나 *댓글쓰기*와 같은 사용자가 사용하는 기능이 되며, 횡단적 관심사는 인증이나 검증과 같이 핵심 관심사의 정확하고 원활한 수행을 도와주는 내부적인 기능들이 된다. 이러한 모듈화는 프로그램의 자가 조작을 위한 방법인 메타프로그래밍(metaprogramming)을 통해서도 가능하나, AOP가 기여한것은 쉽고, 안전하고, 관리가능한 모듈화를 위한 직접적인 어의(direct semantical)를 제공한 것이다[9].

1.3 객체지향성의 본질

객체지향프로그래밍(Object-oriented Programming; OOP)[10]이 제안한 객체지향성(Object-orientation)은 컴퓨터가 실제 세계의 사물(thing)을 모델링하여 궁극적으로 그들 간의 교류(interaction)를 쉽게 표기하기 위한 프로그래밍 및 설계의 방법(paradigm)을 직접적인 어의를 통해 제공하여 준다. AOP는 이러한 객체들의 행위 사이에 횡단적으로 존재하는 관심사를 캡쳐하여 에스펙트 객체에 지역화(localisation)시킴으로써, 높은 모듈성을 부여해 주는 방법이다. 하지만 AOP는 여전히 객체 중심적 사고를 기반으로 횡단적 관심사를 모듈화 시킨 것이기 때문에 *객체의 재사용*은 가능하지만, 어드바이즈함수[11]와 같은 함수, *즉 행위의 재사용*은 어렵다. 1.1장에서 현대 소프트웨어의 행위는 복잡하기 때문에 하나의 행위가 여러 개의 하위 행위로도 구성될 수 있다고 하였는데, 단순히 행위를 재사용하는 것을 넘어서 유사한 패턴으로 사용되는 행위들의 집합을 재사용할 필요성이 대두되었다. 본 연구는 객체지향성의 핵심 아이디어인 객체의 원형인 클레스와 그 실제(realisation)인 인스턴스라는 개념들을 살펴보고, 이에 계층적 관계(hierarchical relationship)를 부여하여, 객체의 상속과 개선(refinement)을 통한 구체화(specification)가 소프트웨어의 복잡성을 감소시켜 준 것과 같이, 동일한 계층적 관계를 소프트웨어의 행위에도 적용시키고자 한다. 객체지향성을 행위에 적용시키기 위해, 우리는 행위가 가져야 할 속성을 자가조합성(Self-composability)이라고 명명하고 자가조합적 관점을 기반으로 프로그래밍 및 설계 그 방법으로 자가조합프로그래밍(Self-composable Programming)을 소개한다. 자가조합프로그래밍은 OOP의 클레스에 해당하는 *추상 행위*를 생성하고, 이를 상속 및 개선하여 구체화와 실제화를 할 수 있는 *구체행위*를 제안한다. 자가조합프로그래밍은 이러한 생성, 상속, 개선의 과정을 직접적인 어의로 제공하기 위한 2가지 기능인 자가조합성(self-composability)과 다단계 상속(multi-level inheritance)을 프로그래밍 레벨에서 제공한다.

1.4 자가조합적 접근법

자가조합프로그래밍은 AOP와는 다르게, 행위에 관계를 부여하여 횡단적 관심사의 모듈화를 시키는 방법을 취한다. 객체지향프로그래밍을 처음 선보인 언어들이 세상에 있는 객체들의 교류를 모델링 하기 위해 접근한 것처럼, 자가조합프로그래밍은 행위를 객체에 의존된 객체지향적인 관점으로 보지 않고, 독립적 존재로 취급한다. 따라서 객체를 상속하는 것처럼 행위를 상속할 수 있으며, 결과적으로 행위에 계층적 관계가 부여되어 추상적인 행위(부모 행위)를 상속하고, 개선하여 구체적인 행위(자식 행위)를 만드는 것이 가능해진다. 이때, 부모 행위에 횡단적 관심사가 모듈화되며, 자식 행위는 이를 자가조합하여 개선해나가면서 최종적으로 원하는 기능을 만들어간다. 즉, 부모 행위는 해당 작업에 대한 도메인 특정(domain-specific)한 재사용 가능한 패턴[12]을 만들어준다고 볼 수 있다. 본 논문의 나머지 부분은 자가조합성의 개념과 자가조합프로그래밍의 구현체인 Self-js[13]를 소개하며, 요구사항 분석 및 설계 단계에서 이러한 행위들의 관계 모델링을 위한 자가조합적 도메인 분석을 웹서비스를 예로 소개한다. 또한, 행위의 자가조합을 위한 5가지의 메서드를 이용한 직접적인 개선(explicit refinement)방법과 이를 고도화 시킬 Traits[14]등의 방안들을 소개한다. 마지막으로 웹서비스를 대상으로 자가조합 프로그래밍방법을 평가하고, 결과 및 개선점 그리고 한계점과 미래 연구 방향을 소개한다.

2 자가조합개념

2.1 조합의 의미

자가조합프로그래밍에서 조합(compose)은 행위 조합을 의미하며, UNIX 철학[15]과 비슷하게, 더욱 고수준의 문제를 해결하기 위해, 이미 조합된 행위가 더 높은 차원의 행위의 일부분으로 조합될 수 있는 고차적인 조합성도 포함한다. 메시징 서비스에서 메시지를 보내는 행위는, 메시지를 보내는 핵심 관심사와 데이터를 검증하고 저장하는 횡단적 관심사로 이루어져 있다. 만약 우리가 (a)파일을 공유하고 (b)자동으로 메시지를 보내는 기능을 만들고자 한다면, a행위와 와 b행위를 조합하여 (c)파일 공유하기라는 고차원의 행위로 바뀔 수 있다. 현대소프트웨어는 다양한 차원의 행위들을 필요로 하며, 이러한 고차원적 조합성을 직접적인 문법으로 지원하는 것은 AOP등 기존 방법들과 비교하여 효과적인 모듈성을 제공해준다.

2.2 행위의 자가조합성

행위의 자가추가 : 행위를 만들기 위해서 프로그래머는 저수준의 행위(하위 행위)를 조합하여 만들 수 있는 요소이다. 메시지 보내기는 인증, 로깅, 검증 등의 하위 행위로써 조합될 수 있으며, 이러한 조합을 통해 상속을 위한 클래스, 즉 추상 행위를 만들 수 있다.

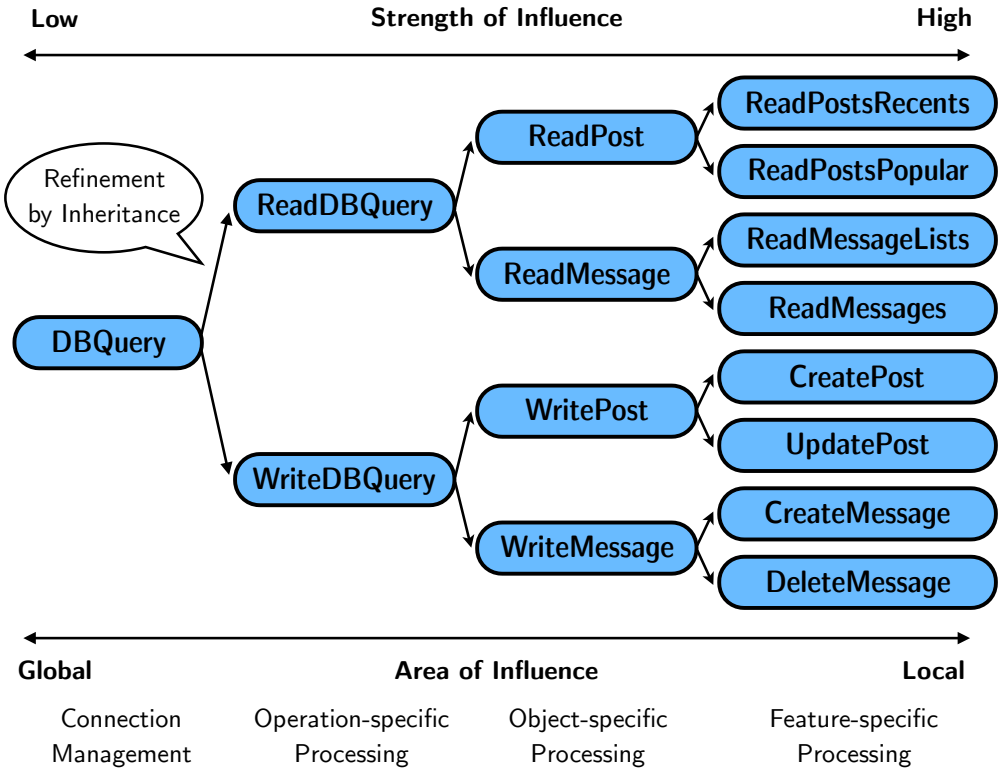


그림 1: 웹서비스 기능들을 구성하는 행위의 다단계 상속

행위의 자가업데이트 : 자가업데이트는 행위의 일부분을 업데이트하는 것으로, 각 행위가 하위 행위들로 구성되어진 점을 활용하여 일부의 하위 행위를 업데이트하는 것을 말한다. 물론 하위 행위 역시 또 다른 하-하위 행위들로 구현될 수 있으므로, 더욱 디테일한 업데이트도 가능하다. (e.g. 메시징 서비스에서 특정 모듈에 새로운 인증방법이 필요한 경우 인증에 관한 모듈을 업데이트하는 경우)

행위의 자가제거 : 특정한 하위 행위를 제거하는 것으로, 자가업데이트와 동일하게 적용 가능하다. (e.g. 공통적으로 적용되는 인증 기능을 없애서 퍼블릭한 API를 만드는 경우)

행위의 자가조작 : 행위의 반복 등 다양한 조작을 가할 수 있는 자유로운 조작 모드이다.

2.3 행위의 다단계 상속

자가조합된 행위는 조합과 개선 그리고 실행의 흐름을 가지고 소프트웨어 안에서 사용된다. 자가조합성이 개선을 위해 사용되는 것이라면, 다단계 상속은 개선할 대상을 만들어주는 역할을 한다. OOP에서 상속은 개선을 통한 객체의 구체화를 목적으로 하는 것처럼, 자가조합프로그래밍은 상속은 개선을 통한 행위의 구체화를 의미한다. 데이터베이스를 이용한 데이터 접근 모듈을 예로 들면, 그림 1이 나타내는 것처럼, 각 행위는 다른 행위와 관계를 가진다. 행위들은 같은 수준에서의 관계도 가지지만, 공통된 횡단적 관심사를 가진 상위의 추상 행위 및 자신을 상속받아 더욱 구체화된 하위의 구체행위와도 관계를 가진다. 본 예제에서는 웹서비스의 기능의 핵심 관심사인 데이터베이스 작업을 위한 4단계의 횡단적 관심사의 도메인을 정의하였다. 연결 관리(connection management)를 시작으로 작업형태에 관련 처리(operation-specific processing), 작업 대상인 객체관련 처리(object-specific processing), 최종 기능관련 처리(feature-specific processing)순으로 횡단적 관심사의 근본성(fundamentality)을 척도로 계층적으로 정의되었다. 가장 추상적인 부모 행위인 DBQuery는 모든 행위에 필요한 횡단적관심사를 모듈화하며, ReadDBQuery및 WriteDBQuery는 읽기와 쓰기라는 작업에 대한 모듈화를 진행한다. 그다음 ReadPost, ReadMessage등은 Post와 Message라는 객체에 대한 처리를 모듈화시키며 마지막 열의 ReadPostRecents 및 ReadPostPopular는 최종 기능의 횡당관심사에 대한 모듈화를 진행한다. 추상적인 행위일수록 글로벌한 영향력을 끼치지만, 그 강도는 낮고, 구체적인 행위일수록 강력하지만, 지역적인 영향력을 끼친다. 화살표가 의미하는 것은 각 행위가 보다 추상적이고 범용적인 행위를 상속받고, 해당 횡단적 관심사 도메인에 필요한 하위 행위를 넣어 개선되는 것을 의미한다(e.g. DBQuery는 일반적인 데이터베이스 모듈을 상속받아 작업 형태에 특정인 읽기 쓰기모듈로 개선시키는 이러한 작업).

앞서 언급한 행위의 계층성은 실제 조직의 의사결정 구조와 비

슷한 점이 많은데, 조직의 근본적인 책임을 지는 최고 책임자, 본 예제에서는 연결을 관리하는 (DBQuery)는 실제 일을 하는, 기능 관련 말단 조직원(ReadPostRecents)들에게 영향을 끼치기는 하지만 그들 개개인에 대한 영향력의 중요도 작다. 하지만, 그 말단 조직원의 직속 책임자(ReadPost)는 그 영향력이 지역적이지만 그 아래의 조직원들에게는 강력하다. 따라서 최종행위는 다양한 상위 행위들의 *어드바이스*의 영향을 받으면서 정의되며, 이러한 어드바이스가 최종 행위를 변경할 수 있다[11]. 행위의 계층적 관계를 아키텍처에 적용하거나 프로그래밍 레벨에서 이용하면 분산된 주체들이 협업하여 기능을 만드는 SOA 및 기타 대규모 시스템에서의 행위를 보다 정확하게 모델링 할 수 있다고 판단된다.

2.4 자가조합적 도메인 분석

자가조합 도메인 분석은 그림 1과 같이 요구사항 분석 단계에서 기능들의 수준을 기반으로 핵심적인 횡단 관심사의 도메인을 정의하고, 도메인을 기반으로 행위의 상속 단계를 설정한 후 소프트웨어를 자가 조합적으로 설계하는 것을 의미한다.

3 Self-js : 프로토타입 구현체

3.1 개요

Self-js[13]는 자가조합성의 구현체이며, 실제 소프트웨어 개발을 위해 행위의 자가조합성과, 다단계 상속을 지원하는 행위를 생성할 수 있는 구현체이다. 우리는 이 컨셉을 구현하기 위한 매개체(medium)로 OOP를 선택하였다[2]. OOP는 행위 스스로 자가조합을 진행하기에 적합한 메서드(method) 표기법을 지원하고, 상속을 지원한다. 우리는 구현체의 언어로 OOP와 함수형 프로그래밍을 지원하는 JavaScript를 선택하였다.

3.2 디자인 및 구현

Self-js는 JavaScript라이브러리로써 함수 생성자와 비슷하게 사용할 수 있다. Self-js를 사용하여 생성한 행위객체 안에는 하위 행위들이 직렬화되어 들어가 있는 배열과 그것을 조작하기 위한 표 1과 같은 메서드들을 가지고 있다. 각 메서드들은 공통적으로 1개의 원시적(primitive)함수나 하위행위로 포함될 또 다른 행위 객체를 인자로 받는다. 따라서 최종 행위적으로 개선된 행위의 실행은 한 최초 인자와 함께 최초 하위 행위를 실행시키고, 해당 하위 행위의 결과값을 다음 행위의 인자로 하여 실행을 반복한다는 형태로 진행된다. 각 하위행위의 파라미터의 경우, 기본적으로 범용성을 가지도록 만들 수도 있으나, 차 후 소개할 map메서드를 사용하여, 하위행위와 인터페이스를 분리할 수도 있다. 이러한 점에 기인하며, 각각의 하위행위가 독립적으로 사용될 수 있으므로, 행위들이 원자 적이고(atomic) 보편적인 빌딩 블록(universal building block)으로써, 여러 형태로 사용될 수 있다.

2) OOP가 대중화된 절차지향형 프로그래밍 위에 만들어진 것처럼, 우리는 자가조합프로그래밍을 OOP를 기반으로 만들었다.

표 1: Self-js의 메소드 목록

메소드 이름	설명
Behavior#add	지정된 행위에 입력된 함수 또는 행위를 덧붙인다(append).
Behavior#Behavior#before	지정된 하위 행위 앞에(before) 입력된 함수 또는 행위를 삽입한다.
Behavior#Behavior#after	지정된 하위 행위 다음에(after) 입력된 함수 또는 행위를 삽입한다.
Behavior#Behavior#update	지정된 하위 행위를 입력된 함수 또는 행위로 업데이트한다.
Behavior#Behavior#delete	지정된 하위 행위를 삭제한다.
Behavior#Behavior#map	지정된 하위 행위를 입력된 함수 또는 행위의 인자로 받아 조작한다(manipulate).

[a] 모든 메서드는 인자로 1개의 원시적(primitive) 함수나 Self-js에 의해서 생성된 행위 객체를 받는다.

4 Self-js의 사용

본 항목에서는 에서는 Self-js의 사용 개요를 코드레벨에서 다루며, 행위의 상호용운용, 개선, 상속에 대한 Self-js의 내부 수행 메커니즘에 대해 다룬다.

4.1 Self-js 한눈에 보기

Listing 1에서는 자가조합행위의 전체적인 라이프사이클을 보여준다.

4.2 행위의 생성

```
1  var Behavior = require('self');
2
3  var DBQuery = new Behavior();
4
5  DBQuery.add(auth);
6  DBQuery.add(validate);
7  DBQuery.add(monit);
```

Listing 2: 자가조합형 행위의 생성

Listing 2는 Self-js를 이용한 행위의 생성에 대해 다룬다. 먼저 Self-js는 범용 JavaScript 런타임인Node.js[16]의 require문을 통해 불러와 사용되어지며, behavior라는 변수에 할당하였다. behavior 객체는 생성자로써 데이터와 메서드 2개의 부분으로 이루어진 DBQuery 인스턴스를 생성한다. 데이터에는 네이티브 함수나 또 다른 Behavior 인스턴스들이 들어갈 수 있으며, 본 예제에서는 .add 메서드를 실행하여 횡단적관심사 인증(auth), 검증

```
/* CONSTRUCTION PART */

// define self-js
var Behavior = require('self');

// initialising behavior
var DBQuery = new Behavior();

// adds some sub-behaviors
DBQuery.add(auth); // authentication checker
DBQuery.add(validate); // data validation
DBQuery.add(monit); // monitoring


/* REFINEMENT PART */

// inherit DBQuery to operation-specific,
    WriteDBQuery
var WriteDBQuery = new DBQuery();

// add some sub-behaviors (refinements)
WriteDBQuery.add(writeBack);

// update specified sub-behavior to new sub-
    behavior
WriteDBQuery.monitoring.update(cacheMonit);

// add sub-behavior in specified location
WriteDBQuery.validate.before(beforeValidate);
WriteDBQuery.validate.after(afterValidate);

// manipulating sub-behavior
WriteDBQuery.validate.map(() => {
return (validate) => {
  validateWrapper(validate);
}
});
//delete sub-behavior
WriteDBQuery.beforeValidate.delete();


/* ADDITIONAL REFINEMENT */

// inherit WriteDBQuery to object-specific
    query
var CreatePost = new WriteDBQuery();
var CreateMessage = new WriteDBQuery();

CreatePost.add(createUserSQLExec);
CreateMessage(createMsgSQLExec);

//additional modification
CreatePost.auth.update(2factorAuth);
CreateMessage.auth.before(geographicalBlock);
```

Listing 1: 자가조합형행위의 라이프사이클

(validate), 모니터링(monit)과 같은 횡단적 관심사를 추가하였다.

4.3 행위의 상속

```
/* Operation-specific Processing */
var ReadDBQuery = new DBQuery();
var WriteDBQuery = new DBQuery();

// ...some refinement


/* Object-specific Processing */
var ReadPost = new ReadDBQuery();
var ReadMessage = new ReadDBQuery();
var WritePosts = new WriteDBQuery();
var WriteMessage = new WriteDBQuery();

// ...some refinement


/* Feature-specific Processing */
var ReadPostsRecents = new ReadPosts();
var ReadPostsPopular = new ReadPosts();
var ReadMessageLists = new ReadMessage();
var ReadMessages = new ReadMessage();
var CreatePost = new WritePost();
var UpdatePost = new WritePost();
var CreateMessage = new WriteMessage();
var DeleteMessage = new WriteMessage();
```

Listing 3: 자가조합형행위의 다단계 상속

Listing 3는 new 키워드를 사용한 다단계 상속에 대해서 다룬다. 본 Listing에서는 3단계 상속을 통해 각 행위가 여러 횡단적 관심사를 개선을 통해 지역화하며. 상속해나간다. 상속 시 내부적으로는 새로운 Behavior 인스턴스를 생성한 뒤, 데이터와 메서드들을 연결시키고(JavaScript에서는 Prototype 체인을 연결)하고 최종적으로 만들어진 인스턴스를 반환하는 방법으로 진행된다.

4.4 행위의 개선

```
var WriteDBQuery = new DBQuery();

WriteDBQuery.add(writeBack);
WriteDBQuery.monitoring.update(cacheMonit);
WriteDBQuery.validate.before(beforeValidate);
WriteDBQuery.validate.after(afterValidate);
WriteDBQuery.validate.map(() => {
return (validate) => {
  validateWrapper(validate);
```

```
10  }
11  });
12  WriteDBQuery.beforeValidate.delete();
13
14  var CreatePost = new WriteDBQuery();
15  var CreateMessage = new WriteDBQuery();
16
17  CreatePost.add(createUserSQLExec);
18  CreateMessage(createMsgSQLExec);
19  CreatePost.auth.update(2factorAuth)
20  CreateMessage.auth.before(geographicalBlock)
```

Listing 4: 자가조합형행위의 개선

Listing 4에서는 Listing 3에서 생성한 DBQuery 개선하여 WriteDBQuery를 만들고, 이를 다시 개선하여 CreatePost와 CreateMessage를 만든다. 개선은 자가조합프로그래밍이 지원하는 다양한 메서드를 통해 직접적으로 진행된다. 추가적인 하위행위들은 add메서드를 통해 덧붙여질 수 있으며, 특정 행위 전후에는before와 after메서드를 사용함으로써 상대위치에 추가할 수 있다. map 메서드는 하위행위들을 새로운 함수영역 안에서 조작할 수 있다. 이 경우, validateWrapper함수의 인자로 입력되었다. Self-js의 각 메서드는 개선의 편의를 위해 하위 행위를 객체의 속성으로 노출시키며, 노출된 행위안에 각각의 개선 메서드들이 작동된다. 하위행위의 이름을 키로 실제 해당 행위가 들어있는 배열에서의 위치를 특정할 수 있으므로, 이 후부터는 일반 배열에 조작을 가하는 것과 동일하게 작동된다.

4.5 기타개선방법

자가조합프로그래밍이 객체지향형 개선을 사용하기 때문에, traits와 같은 기타 OOP 기술을 사용하여 자가조합성을 지원할 수도 있다. 직접적인 개선은 강력하지만 큰 규모 개선의 경우 위험성과 이해성의 측면에서 좋지 않은 면도 존재한다. traits는 대상에 독립적인 행위의 집합으로써[14], 보다 고수준의 간접적인(implicit) 개선을 가능하게 한다.

4.6 행위의 실행

```
1  CreateMessage.exec(Handler);
```

Listing 5: 자가조합형행위의 실행

조합된 최종 행위는 exec메서드를 통해 실행되며, 최종결과물 역시 윈시적인 JavaScript 객체이기 때문에 다른 언어적 지원 없이 쉽게 독립적인 모듈로써 사용될 수 있다.

5 평가 및 분석

5.1 목적

본 섹션에서는 Self-js를 이용한 프로그래밍 및 설계의 모듈성의 평가 및 분석에 대해서 다룬다. 프로그래밍기법은 정량적인 효과만으로 유효성의 검증이 끝나는 것이 아니라 이해성이나 설계에도 영향을 끼치기 때문에 실증적(empirical) 자료가 부족한 새로운 기법의 경우 그 효과를 평가하기가 어렵다. 따라서 본 평가에서는 Self-js를 적용하여 개선된 점들을 기반으로, Self-js의 어떤 점이 모듈성의 측면에서 효율성을 가져다주는지 특정하는 것을 목표로 진행한다. 본 평가에서는 웹 서비스의 핵심 모듈들을 자가조합프로그래밍 및 AOP를 이용하여 고수준에서 구현함으로써[3] 모듈화 정도를 측정하고, 이를 통해 얻은 추세함수를 바탕으로 회귀분석을 통해 프로그램이 확장에 따른 코드량(SLOC)의 증가를 예측, 재사용성을 분석한다. 본 평가의 목적은 Self-js의 기본적인 모듈화 강점인, 재사용의 매계체인 애스펙트 자체를 얼마나 고차적인 재사용을 할 수 있는지를 측정하는 것이 목적이다.

5.2 첫번째 분석 : 새 기능당 요구되는 코드량

첫 번째 분석에서는 User와 Post에 대한 데이터베이스의 Read 작업을 하는 행위를 가진 모듈을 구현하였다. 총 4단계의 상속을 거치며 표 2에 있는 최종 기능 8개를 개선해나간다. 최종기능들의 횡단적 관심사는 인증 기능에 의해 가변성을 가지며, 이를 제외한 부분은 공통성을 가진다. 자가조합적 도메인 분석을 통해 본 기능들의 횡단적 관심사를 작업별, 객체별, 기능별, 형태별(Post에만 적용)의 4단계로 나누었으며 이 구조도는 그림 2 와 같다. 우리는 부록 AOP와 자가조합프로그래밍으로 각 기능을 구현하였고, 각 프로그램의 소스코드와 jsAspect[17]라이브러리 기반의 JavaScript AOP용 레퍼 함수의 소스코드는 부록 A, B, C에 있다. 각 기능별 소요된 SLOC를 산출하기 위해, 우리는 SLOC를 2가지 목적으로 분류하였다.

첫 번째로 최종 기능을 구현하기 위한, 황단적관심사들을 통합해주는 모듈에 사용된 SLOC를 측정하였다. AOP의 경우 에스펙트 객체를 생성하는 부분이 될 것이고, 자가조합프로그래밍의 경우 1 3차까지의 행위 생성 및 개선하는 부분이 될 것이다. 두 번째로 최종적으로 만들어진 횡단적 관심사들의 집합을 핵심 관심사와 조합하는 코드가 된다.

5.2.1 AOP 기반 구현체의 분석

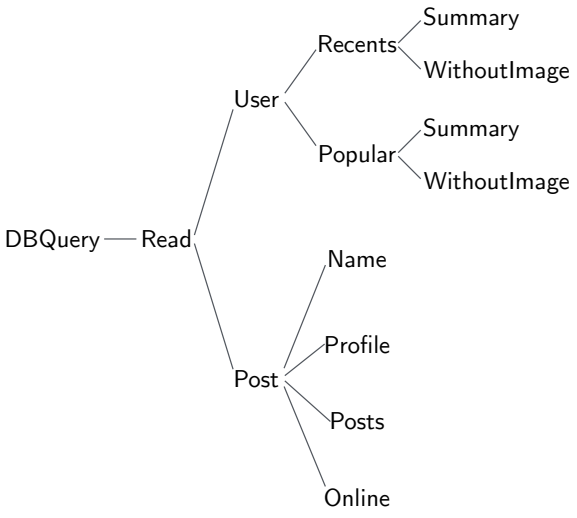
AOP에서 에스펙트 객체를 생성을 위한 내용이 되는 코드는 부록 A의 라인 1~5, 8~12, 21~27, 29~36 총 26줄(표 3의 a)이며, 그 중 실제 횡단적 관심사에 대한 코드는 2-5, 9-11, 22-26, 30-35

3) 본 평가에서 요구되는 기능이 간단한 관계로 AOP에서는 before 어드바이스만을 사용하였다. 또한 양 기법 모두 내부 구현을 생략한 고수준에서의 구현을 가정하였다.

표 2: 첫번째 분석용 소프트웨어의 기능 목록

기능명	설명
User.getName	인증 후 지정된 사용자 이름 받기
User.getProfile	인증 후 지정된 사용자 프로필 받기
User.getPosts	인증 후 사용자의 포스트들 받기
User.getOnline	인증 없이 사용자의 접속 여부 받기
Post.getRecentSummary	인증 후 최신 포스트 요약 받기
Post.getRecentsWithoutImage	인증 후 최신 포스트 글만 받기
Post.getPopularSummary	인증 없이 인기 포스트 요약 받기
Post.getPopularWithoutImage	인증 없이 인기 포스트 글만 받기

그림 2: 첫번째 분석용 소프트웨어의 행위 관계도



*4단계 상속을 거친 행위의 구조도는 위와 같다. 최종 기능의 이름은 왼편부터 끝 노드까지의 이름을 붙혀서 확인할 수 있다.

표 3: 하나의 기능을 구현하는데 필요한 평균 SLOC 비교

	AOP	자가조합
재사용 매계체 선언에 사용된 SLOC(a)	26	14
횡단적 관심사의 SLOC(b)	18	6
기능의 갯수	8	
하나의 기능 구현시 횡단적 관심사의 평균 SLOC (b/8)	2.25	0.75

18줄(표 3의 b)이다. 에스펙트 객체 생성을 위한 코드를 보면, 이들의 하위 행위인 logging, auth, cacheLookup, userIdValidation 등의 행위들이 ReadUser, ReadUserWithAuth, ReadPost, ReadPostWithoutAuth 모듈에 유사한 패턴으로 산포되어 있는 것을 알 수 있으며, auth에 의한 가변성 때문에 Aspect가 이를 효과적으로 재사용하지 못하여 이러한 하위 행위가 중복되게 호출되는 것을 확인할 수 있다.

5.2.2 자가조합 기반 구현체의 분석

자가조합프로그래밍은 프로그램의 이해성에 영향을 주지 않으면서도 중복호출에 의한 코드의 재사용성 하락을 피하기위해, 평평한(flatten) 조합을 하는 AOP와는 다르게, 계층적이고 보다 고수준의 조합을 보여준다. 횡단적 관심사를 통합해주는 User에 관한 1~2차 까지의 행위 생성 및 개선이 라인3, 5, 7, 14총 4줄이며, Post에 관한 행위 생성 및 개선이 라인15, 19 총 2줄로 총 6줄(표 3의 b)이다. 또한, 최종 단계의 개선에 사용되는 코드는 라인 9~12, 16~17, 20~21 총 8줄이며, 이 2개를 합하여 재사용 매계체 선언에 사용된 SLOC는 14줄(표 3의 a)이라 할 수 있다.

5.2.3 결과 분석

표 3에서 하나의 기능을 구현하는 데 필요한 횡단적 관심사의 평균 SLOC는 AOP의 경우 2.25으로 나타났고, 자가조합프로그래밍의 경우 0.75으로 나타났다. 즉 AOP를 사용하면 하나의 기능당 2.25개의 횡단적 관심사가 명시적으로 사용되는 반면, 자가조합프로그래밍의 경우 0.75개만 사용되는 것으로 상당히 고무적인 결과이다. 추가적으로 OOP가 데이터의 캡슐화(encapsulation)를 통해, 정보 감춤(information hiding)를 하는 것처럼 자가조합프로그래밍은 행위를 구성하는 하위 행위들의 캡슐화를 통해 정보 감춤을 하게 된다. 이의 장점과 단점 역시 이해성인데, 고수준의 사용이 가능한 점이 있지만, 이러한 개선 작업이 산포될 경우 프로그래머가 디테일한 동작을 알기 어렵다는 것이다. 또한 현재의 직접적인 개선은 행위의 정확성(correctness)를 깨트릴 수도 있으므로, IDE지원이나 문서화등의 외적인 방법과 함께, 간접적인 개선 방법과 이를 지역화시키는 방법에 대한 추가 연구가 필요하다. 마지막으로 OOP 역시 객체 간 협업(Object-oriented Collaboration)에서 협업의 내용과 단계가 늘어남에 따라 프로그래머가 소프트웨어의 동작을 직접적으로 정의하거나 확인할 방법이 없다는 점이 대두하였고, 이러한 협업의 과정을 보다 명료하게 만들기 위해, OOP 환경에서 데이터와 인터렉션을 분리하고, 둘을 연결하는 컨텍스트를 만드는 DCI architecture가 제안되었다[18]. 행위와 하위행위의 조합 역시 이러한 컨텍스트상에서 조합이 가능하다면 사용자가 행위의 내부를 파악하는데 도움이 될 것이라고 판단되며, 이러한 아키텍처의 필요성을 확인할 수 있었다.

표 4: 자가조합프로그래밍의 행위 상속 단계 별 SLOC 필요량

행위 상속 단계 (차수)	부모행위의 갯수	자식행위의 갯수	개선용 SLOC	전체 SLOC
1차	1	2	5	10
2차	2	5	5	50
3차	5	10	5	250
4차		*(추정치)*		1,250
5차		*(추정치)*		6,249

표 5: AOP의 행위 상속 단계 별 SLOC 필요량

행위 상속 단계 (차수)	부모행위의 갯수	자식행위의 갯수	개선용 SLOC	전체 SLOC
1차	1	2	10	20
2차	2	5	15	150
3차	5	10	20	1,000
4차		*(추정치)*		7,211
5차		*(추정치)*		50,988

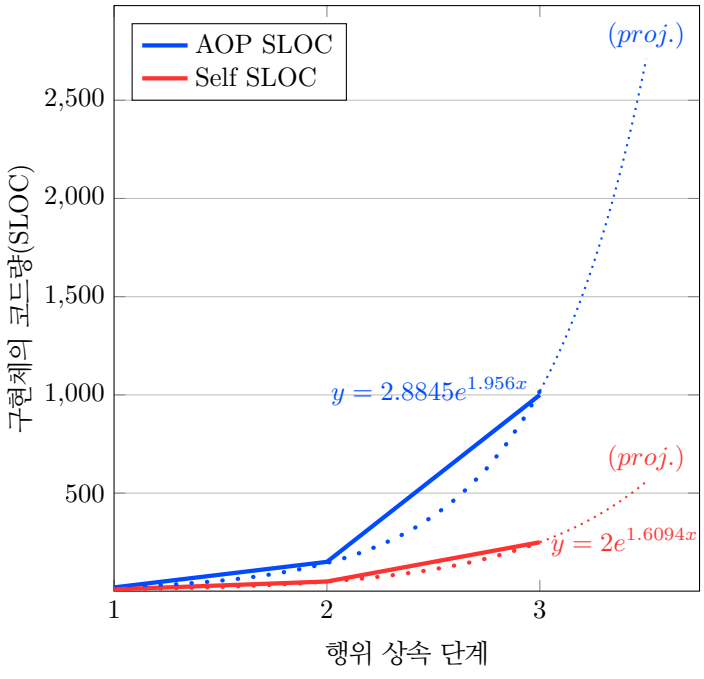


그림 3: 행위의 상속 수준에 따른 코드량의 증가 예측

5.3 두번째분석 : 새 기능당 재사용되는 코드량 추정

두 번째 분석은 AOP및 자가조합프로그래밍을 이용하여 가변성을 가진 기능이 고도화시키면서 어느 정도의 SLOC가 필요한지 측정하는 방법이다. 본 분석의 목적은 첫 번째 분석을 보다 대규모에서 시뮬레이션하여 행위의 개선 단계별로 어느정도의 SLOC 재사용되는지 측정하는 것이다 첫 번째 분석의 규모를 키워 가변성을 가진 소프트웨어의 기능의 수가 고도화되어감에 따라 추가적으로 필요한 소스코드의 양을 예측하여 재사용성을 측정하는 방법이다. AOP에서도 상속을 통해 에스펙트 객체를 재사용하는 방법이 있으나 직접적으로 고차적 재사용을 지원하지 않는다[5, 6]. 본 분석에서는 가상의 웹서비스의 행위를 3단계의 상속을 통해 구체화 시킨다, 따라서 각 단계에 사용되는 SLOC는 부모 행위의 수와 자식 행위의 수 그리고 행위별 사용된 소스코드량 3개 항목의 곱으로 산출된다. 우리는 본 실험에서, 상속을 통해 각각 2, 5, 10개의 자식 행위를 생성하고, 각 상속 시마다 5라인의 코드를 추가하여 요구사항 만족을 위한 개선을 진행한다. AOP의 경우 1차적 재사용만 가능하기 때문에 첫 번째 분석의 부록 A와 같이 기존 에스펙트와 행위적으로 유사한 함수의 실행패턴을 가진 새로운 에스펙트를 만든다. 결과적으로 표 5과 같은 SLOC의 증가량을 보이게 된다. 부록 C의 자가조합프로그래밍의 경우 행위자가 직접(explicit) 하게 실제 바뀐 부분만을 조작함으로써 프로그램 수정의 유연성을 높여 결과적으로 최소한의 비용으로 가변성을 달성하게 한다. 표 5상에서 개선에 필요한 소스코드가 5라인으로 유지되는 것이 이를 증명한다.

5.4 결과 및 개선점

표 4, 5를 기반으로 추세함수를 도출한 결과, 그림 3와 같이 소프트웨어의 행위 수준의 증가에 따른 코드 수의 증가를 효과적으로 억제(suppress)시켜주는것을 알 수 있다. 즉 자가조합프로그래밍은 소프트웨어의 가변성을 효과적으로 관리하여, AOP등의 기존 방법으로는 지수 배로 증가하는 SLOC을 실제 필요한 부분만 조작할 수 있도록 지원하는 것을 확인할 수 있었다.

6 관련 연구

AOP[4]를 비롯하여 GenVoca[19], subject-oriented programming[20], adaptive plug-and-play components[21], role components[22] 등 관심사 분리를 통해 소프트웨어의 가변성에 대응하고자 다양한 기술들이 존재한다. 자가조합프로그래밍이 모듈화에 있어서 5의 결과처럼 기존 방법들과 비교하여 나은 성능을 보여주었지만, 이는 극히 성능의 비교가 목적이 아닌, 효율성을 가지는 부분을 코드 레벨에서 확인하기 위함이었다. 자가조합프로그래밍이 기존 방법과 다른 부분은 코드 생성기나 IDE및 툴지원 없이 코드 레벨에서 OOP의 아이디어를 활용하여 행위들의 관계를 유연하게 모델링 하는 방법을 제공한 것이다.

자가조합프로그래밍은 소프트웨어의 행위를 객체에서 분리하여 생각하는 관점을 제공함으로써, 객체에 독립적으로 행위만의 계층구조를 만들 수 있는 생각의 틀(framework of thinking)을 제공하였다. 이를 프로그래머들에게 자신들이 개발할 소프트웨어의 기능을 자가조합적 도메인 분석을 통해 모델링 할 수 있는 방법을 제안하였다. 자가조합프로그래밍의 최종목적은 라이브러리 또는 프로그래밍언어의 내장 기능으로 동작하며, 가변성을 가진 모듈을 쉽게 만드는 것이며, 따라서 필요에 따라 OOP는 물론, AOP와도 상호보완적으로 사용 될 수 있다. AOP가 프로그래머에게 애스팩트적 사고(aspectual tinking)를 제안하여 AOP구현체의 사용과 별개로 소프트웨어를 설계하는데 하나의 방향을 제시한 것처럼[23], 자가조합프로그래밍 역시 프로그래머들이 소프트웨어의 행위를 바라보는 새로운 관점을 제시하고자 하는 것이 목적이다.

7 논의

7.1 구현형태

현재 자가조합프로그래밍은 라이브러리 형태로 제공되나, 문법 및 처리 측면에서 언어에 내장시킨다면 더욱 효율적인 처리가 가능하다.

7.2 한계점 및 미래 연구 방향

현재 자가조합프로그래밍이 가지는 주요한 한계점은 OOP의 한계점과도 흐름을 같이한다, 다단계 상속과 여러번의 개선을 통해 횡단적관심사를 지역화시킬수 있었지만, 그 개선과 상속과정이 산포될수 있다는 문제가 있다. 이에 의해 최종 사용자는 행위의 내부를 정확하게 파악하는데 어려움을 겪을수 있다. 따라서 개선과 상속과정 자체를 더 표현력있게 만드는 방법이 연구되어야 하며, 이를 위해 OOP의 traits나 mixin[24]등 간접적 조합 방법들을 행위에 적용시키는것이 필요하다. 또한 이러한 작업을 효과적으로 진행하기위한, MVC와 같은 설계 패턴(architectural pattern)이 만들어 질 필요가 있다. MVC는 프로그래미어의 생각 모델(mental model)과 컴퓨터의 데이터 모델을 분리하는 방법으로[25], 개선과 상속의 산포를 방지하기 위해 이러한 분리를 목적으로 DCI[18]와 같은 설계 패턴이 만들어질 필요가 있다. 앞의 실험결과가 보여주듯이, 자가조합프로그래밍은 대규모의 가변성을 가진 소프트웨어에 유용하나, 이러한 시스템을 자가조합하게 리펙토링(refactoring)하는것은 비용 문제가 크다. 따라서 기존 시스템을 자가조합하게 만들어줄 수 있는 프로그램 변환기법이나 런타임에서 동작하는 레퍼나 슬라이싱 방법들이 필요하다. 표기법의 측면에서는 도메인 특정한 행위 생성성 및 표기법(e.g. .add 대신, .addValidation)을 제공함으로써 범용 표기법에 대비해 처리 및 전달상의 용이성을 높일 수 있다. 또한, 자가조합프로그래밍을 패키지 관리 시스템과 연계시켜, 고수준의 기능을 빠른 프로토타이핑을 목적으로 하는 시스템 개발도 가능할 것이다. 이러한 다양한 응용은 자가조합프로그래밍이 기존에 통용되고 있는 OOP를 기반으로 만들어져서 다른 기법들과 다른 장점이지만 동시에 OO의 표현적 한계를 가지고 있다. 따라서 본 연구가 제시한 소프트웨어의 가변성 문제를 근본적으로 해결하기 위해서는 행위를 모델링 위한 전용 표기법 및 이의 정확성을 검증하기 위한 방법들이 개발되어야 한다. 또한, 행위 간의 조합을 원활하게 하려면 행위로부터 생성된 기능 교류 문제(feature interaction problem) 역시 해결되어야 한다[26]. 앞서 1장에서 언급한것처럼 본 연구의 최종 목적은 객체지향프로그래밍의 방식(fashion)과 방법(paradigm)을 행위의 모델링과 조합에 이용하는것이 아니다. 본 연구의 궁극적인 목적은 행위가 객체와 분리된 존재로써, 독립적인 관계를 가질수 있다는 사실을 바탕으로[14], 행위를 모델링하기위한 새로운 방법들의 필요성 및 방법을 제시하여 연구자(researcher)및 전문가(experienced practitioner)들에게 새로운 관점(new point of view)을 제공하는 것이다. 따라서 중장기적으로 우리는 행위의 유연한 모델링과 조합을 위해, 행위의 계층성을 지원하기 위해 본 연구에서 제안된 2가지 속성인 자가조합성과 다중상속성의 개념을 기반으로, 이를 지원하는 OOP-독립적인 새로운 언어적 표기법을 개발하고, 상속이나 형질과 같이 행위라는 순차성과 그 관계를, 실세 세상의 행위라는것의 본질(nature)을 다양한 관점에서 파악하여 프로그래밍 레벨에서 더욱 잘 모델링할수 있는 방법에 대한 연구를 진행할 예정이다.

8 결론

본 연구에서는 소프트웨어의 모듈화를 위해 행위의 계층성이라는 개념을 제시하고, 이를 모델링 및 조합을 하기위한 자가조합성과 다중상속성이라는 개념 제안하였다. 이를 프로그래밍에서 지원하기위해 객체지향성을 이용한 자가조합프로그래밍을 제안하였다. 우리는 현대 소프트웨어의 기능을 행위의 집합이라고 정의하고, 사용되는 행위들의 가변성의 증가에 따른 기존 모듈화 기법의 한계를 고차적인 재사용성 지원의 문제라고 규정하였으며, 고차적인 재사용성을 지원하기 위해 상속 및 자가조합이 가능한 행위를 생성하는 자가조합프로그래밍을 제안하고 자가조합형 도메인 분석을 통해 요구사항분석의 일환으로 자가조합적인 설계방법을 제시하였다. 우리는 자가조합프로그래밍의 웹서비스를 대상으로 AOP와 비교하여 평가 및 분석을 실시하여 평가 항목을 통해서 어떤 부분이 기존 방법인 AOP와 비교하여 효율성을 주는지 도출할 수 있었다. 본 연구는 자가조합프로그래밍이라는 실질적인 프로그래밍 방법을 제시함과 동시에, 객체에서 행위를 독립시키고, 행위들에게 자체적인 계층적인 관계를 부여하여 모듈화를 진행함으로써 행위모델링의 중요성을 강조하며, 이를 통한 실질적인 프로그래밍 기법을 제공함과 동시에 연구자와 전문가에게 새로운 관점을 제공해주는 것이 목표이다. 따라서 OOP 독립적인 추가

적인 새로운 표기법 및 행위라는 것이 가지는 본질을 프로그래밍 레벨에서 더욱 잘 표현하는 방법의 중요성을 강조하였다. 우리는 본 연구를 통해 객체지향프로그래밍이 50여 년 전에 실제 사물을 모델링 한 것처럼, 실제 세상의 행위를 정확하게 모델링하고, 쉽게 제어하는 것에 대한 중요성과 가치는 제시하였고, 자가조합프로그래밍이 행위를 모델링 하는 매계체로써 사용될 수 있음을 제시하였다.

참고 문헌

[1] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," tech. rep., Technical Report CMU/SEI-90- TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[2] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[3] M. Shaw, "Modularity for the modern world: summary of invited keynote," in *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD*, pp. 1–6, 2011.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*, pp. 220–242. Springer Berlin Heidelberg, 1997.

[5] S. Hanenberg and R. Unland, "Concerning aop and inheritance," in *Aspektorientierung-Workshop der GI-Fachgruppe*, vol. 2, pp. 3–4, 2001.

[6] S. Hanenberg and R. Unland, "Using and reusing aspects in aspectj," in *Workshop on Advanced Separation of Concerns, OOPSLA*, 2001.

[7] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Pearson Education India, 2005.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *European Conference on Object-Oriented Programming*, pp. 327–354, Springer, 2001.

[9] G. Kiczales, "It's not metaprogramming," *Software Development Magazine*, 2004.

[10] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "Some features of the simula 67 language," in *Proceedings of the Second Conference on Applications of Simulations*, pp. 29–31, Winter Simulation Conference, 1968.

[11] W. Teitelman, "Pilot: a step toward man-computer symbiosis," tech. rep., PhD thesis, September 1966.

[12] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *ECOOP'93 - Object-Oriented Programming, 7th European Conference*, pp. 406–431, 1993.

[13] "Self-js Webpage." https://github.com/hiun/self-js.

[14] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable units of behaviour," in *European Conference on Object-Oriented Programming*, pp. 248–274, Springer, 2003.

[15] M. D. McIlroy, E. N. Pinson, and B. A. Tague, "Unix time-sharing system: Foreword," *The Bell System Technical Journal*, vol. 57, pp. 1899–1904, July 1978.

[16] "Node.js Webpage." https://nodejs.org.

[17] "jsAspect Webpage." https://git.io/vDmTh.

[18] T. Reenskaug and J. O. Coplien, "The DCI architecture: A new vision of object-oriented programming," *An article starting a new blog:(14pp) http://www. artima. com/articles/dci_vision. html*, 2009.

[19] D. Batory and S. O'malley, "The design and implementation of hierarchical software systems with reusable components," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 4, pp. 355–398, 1992.

[20] W. Harrison and H. Ossher, *Subject-oriented programming: a critique of pure objects*, vol. 28. ACM, 1993.

[21] M. Mezini and K. Lieberherr, "Adaptive plug-and-play components for evolutionary software development," in *ACM Sigplan Notices*, vol. 33, pp. 97–116, ACM, 1998.

[22] M. VanHilst and D. Notkin, "Using role components in implement collaboration-based designs," *ACM SIGPLAN Notices*, vol. 31, no. 10, pp. 359–369, 1996.

[23] G. Kiczales, "Once more, from the top," *Software Development Magazine*, 2005.

[24] G. Bracha and W. Cook, "Mixin-based inheritance," *ACM Sigplan Notices*, vol. 25, no. 10, pp. 303–311, 1990.

[25] T. Reenskaug, "The model-view-controller (mvc) its past and present," *University of Oslo Draft*, 2003.

[26] S. Apel and C. Kästner, "An overview of feature-oriented software development.," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

부록 A AOP 기반 구현체의 소스코드

```
var ReadUser = createAspect(function () {
  logging(data);
  auth(data);
  cacheLookup(data);
  userIdValidation(data);
});

var ReadUserWithoutAuth = createAspect(function
    () {
  logging(data);
  cacheLookup(data);
  userIdValidation(data);
});

var User = {
   getName: applyAspect(ReadUser,
       readUserNameQuery),
   getProfile: applyAspect(ReadUser,
       readUserProfileQuery),
   getPosts: applyAspect(ReadUser,
       readUserPosts),
   getOnline: applyAspect(ReadUserWithoutAuth,
       readUserOnline)
};

ReadPost = createAspect(function () {
  logging(data);
  cacheLookup(data);
  postNumberValidation(data);
  rangeValidation(data);
  ReadRecentsSummaryQuery(data);
});

ReadPostWithoutAuth = createAspect(function ()
    {
  logging(data);
  auth(data);
  cacheLookup(data);
  postNumberValidation(data);
  rangeValidation(data);
  ReadRecentsSummaryQuery(data);
});

var Post = {
  getRecentSummary: applyAspect(ReadPost,
      readPostRecentsSummary)
  getRecentsWithoutImage: applyAspect(ReadPost,
      readPostRecentsWithoutImage)
  getPopularSummary: applyAspect(
      ReadPostWithoutAuth,
      readPostPopularSummary)
  getPopularWithoutImage: applyAspect(
      ReadPostWithoutAuth,
      readPostPopularSummary)
}
```

부록 B AOP Helper 의 소스코드

```
function createAspect (beforeFunc, afterFunc) {
  return new jsAspect.Aspect(new jsAspect.
      Advice.Before(beforeFunc, afterFunc);
}

function applyAspect (aspect, func) {
  //wrapper for applying aspect to function,
      instead of object.
  var obj = {
    method: func
  };
  aspect.applyTo(obj);
  return obj[method];
}
```

부록 C 자가조합프로그래밍 기반 구현체의 소스코드

```
var DBQuery = new Behavior().add(logging);

var DBQueryRead = new DBQuery().add(auth).add(
    cacheLookup);

var DBQueryReadUser = new DBQueryRead().add(
    userIdValidation);

```

```
7  var User = {
8  getName: new DBQueryReadUser().add(
       readUserNameQuery),
9  getProfile: new DBQueryReadUser().add(
       readUserProfileQuery),
10 getPosts: new DBQueryReadUser().add(
       readUserNameQuery),
11 getOnline: new DBQueryReadUser().add(
       readUserOnline).auth.delete()
12 };
13
14 var DBQueryReadPost = new DBQueryRead().add(
       postNumberValidation).add(rangeValidation);
15 var DBQueryReadPostRecents = new
       DBQueryReadPost().add(ReadRecentsQuery);
16 var DBQueryReadPostPopular = new
       DBQueryReadPost().add(ReadPopularQuery);
17
18 var Post = {
19 getRecentSummary: new DBQueryReadPostRecents.
       ReadRecentsQuery.update(
       ReadRecentsSummaryQuery),
20 getRecentsWithoutImage: new
       DBQueryReadPostRecents.ReadRecentsQuery.
       update(ReadRecentsSummaryWithoutImageQuery)
       ,
21 getPopularSummary: new DBQueryReadPostPopular.
       ReadPopularQuery.update(
       ReadPopularSummaryQuery).auth.delete(),
22 getPopularWithoutImage: new
       DBQueryReadPostPopular.ReadPopularQuery.
       update(ReadPopularWithoutImageQuery).auth.
       delete()
23 };
```