CRM 시스템의 아키텍처 설계 비교:
외부 변화가 비즈니스 로직에 미치는 영향 분석

이상희 O, 김규회, 신성현, 정현정, 조동필, 정설영(교신저자), 고재도(교신저자)*
경북대학교, *쿼타랩

lsh277604@gmail.comO, k546kh@gmail.com, sin6708k@gmail.com, wgwjh05169@gmail.com, ehdvlf001@gmail.com, snowflower@knu.ac.kr, haibane84@gmail.com*

Comparative Analysis of Architecture Designs in CRM Systems: Assessing the Impact of External Changes on Business Logic

Sanghee Lee O, Kyuhoi Kim, Seonghyeon Shin, Hyeonjeong Jeong, Dongpil Jo
Seolyeong Jeong, Jaedo Ko*
Kyungpook National University *Quota Lab, Inc

요 약

본 논문은 같은 기능을 계층형 아키텍처와 헥사고날 아키텍처를 적용해 구현하였을 때 각 아키텍처가 외부 변화, 특히 데이터베이스 구조 변경과 같은 외부 요인에 어떻게 반응하는지 분석한다. 계층형 아키텍처는 각 계층 간의 강한 결합으로 인해 외부 변화가 비즈니스 로직에 직접적인 영향을 미칠 수 있는 반면, 헥사고날 아키텍처는 이러한 영향을 최소화한다. 또 계층형 아키텍처와 헥사고날 아키텍처가 장기간 운영되는 시스템에 어떠한 장단점을 가지고 있으며, 각각의 아키텍처가 적합한 환경과 상황을 제시한다. 이를 통해 개발자들이 서비스의 특성에 맞는 아키텍처 선택에 도움을 받을 수 있도록 한다.

Ⅰ. 서 론

CRM (Customer Relationship Management) 시스템은 기업이 고객 관계를 관리하고 분석하는 데 사용하는 기술로, 고객 만족도를 향상시키고, 매출을 증가시키는 데 중요한 역할을 한다. CRM 시스템은 마케팅, 영업, 고객 서비스 등 다양한 부문에서 활용되며, 효율적인 고객 정보의 활용을 통해 개인화된 서비스를 제공하고, 고객의 충성도를 높이는 데 기여한다. 이러한 CRM 시스템의 개발에 있어서, 소프트웨어 아키텍처는 유지보수성, 확장성 및 재사용성에 직접적인 영향을 미친다.

계층형 아키텍처는 가장 널리 사용되는 소프트웨어 아키텍처 중 하나로, 각 계층이 특정한 역할과 책임을 가지며, 상위 계층이 하위 계층에만 의존하는 일방향의 의존성 구조를 가진다.

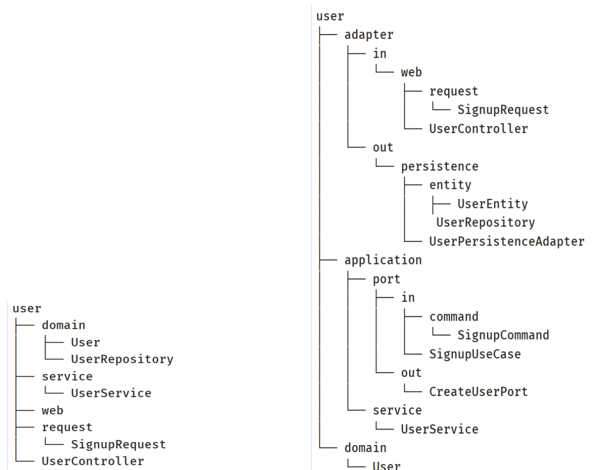
반면, 헥사고날 아키텍처는 애플리케이션의 코어 로직을 중심으로 내부와 외부를 명확히 분리한다. 이 아키텍처는 애플리케이션의 핵심 로직을 ‘내부’로, 사용자 인터페이스나 데이터베이스 등 외부 시스템과의 연동을 ‘포트’를 통해 ‘어댑터’가 매개하는 형식으로 구성된다.

본 논문에서는 CRM 의 같은 기능을 계층형 아키텍처와 헥사고날 아키텍처로 각각 구현했을 때 외부 변경이 비즈니스 로직에 미치는 영향을 비교한다. [1][2]

Ⅱ. 본론

계층형 아키텍처와 헥사고날 아키텍처는 [그림 1]과 같이 패키지를 구성한다. 헥사고날 아키텍처는 도메인 모델을 담은 domain 패키지와 포트와 유스케이스가 담긴 application 패키지, 데이터베이스와 같이 애플리케이션 코어와 상호작용을 하는 다양한 어댑터가 담긴 adapter 패키지로 구성한다.

이러한 구조로 원하는 구성 요소를 빠르게 파악할 수 있고, 변경에 용이하며, 패키지와 클래스의 접근 제어를 통해 아키텍처에서 벗어나는 것을 방지할 수 있다.



[그림 1] 계층형 아키텍처(좌)와 헥사고날 아키텍처(우)의 패키지 구조

다음은 본 논문에서 구현한 회원 데이터베이스에 firstName, lastName attribute 가 하나의 name attribute 로 통합되는 변경이 발생하였을 경우, 계층형 아키텍처와 헥사고날 아키텍처의 비즈니스 로직 변화이다.

```
package user.adapter.out.persistence.entity;

@Getter
@Entity @Table(name = "user")
public class UserEntity implements UserState {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private String email;
    private String firstName; ──┐
    private String lastName;  ──┴──> private String name;
}
```

[그림 2] 헥사고날 아키텍처의 데이터베이스 스키마 변화

```
package user.adapter.out.persistence;

public class UserPersistenceAdapter implements CreateUserPort {
        ...
        public void createUser(UserState userState) {
            UserEntity userEntity = new UserEntity(
                null,
                    userState.getUsername(),
                      userState.getPassword(),
                      userState.getEmail(),
                      userState.getFirstName(), ─┐    "%s %s".formatted(
                      userState.getLastName()   ─┴─>       userState.getFirstN이e(),
                );                                       userState.getLastName()
                                                      )
            userRepository.save(userEntity);
        }
}
```

[그림 3] 헥사고날 아키텍처의 Outgoing Port 변화

```
package user.application.service;

@Service
public class UserService implements CreateUserUseCase {
        ...
    public void signup(SignupCommand command) {
        User user = new User(
                null,
                user.getUsername(),
                user.getPassword(),
                user.getEmail(),
                user.getFirstName(),
                user.getLastName()
        );
        createUserPort.createUser(user);
    }
}
```

[그림 4] 헥사고날 아키텍처의 비즈니스 로직 변화

```
package user.domain;

@Getter
public class User implements UserState {

    private final UserId id;
    private final String username;
    private final String password;
    private final String email;
    private final String firstName;
    private final String lastName;
}
```

[그림 5] 헥사고날 아키텍처의 Domain Entity 변화

헥사고날 아키텍처에서는 변경 요구에 따라, [그림 2]와 같이 데이터베이스 스키마의 수정이 필요하더라도 [그림 4]와 [그림 5]처럼 도메인 모델은 그대로 유지된다.

반면 [그림 6]과 [그림 7]을 보면 계층형 아키텍처의 경우 데이터베이스의변경을 도메인 모델에 반영하기 위해, User 클래스에서 firstName 과 last Name 필드를 제거하고 새로운 name 필드를 추가하는 변경이 필요하다.

```
package user.domain;

@Entity @Table(name = "user")
public class User {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private String email;
    private String firstName; ──┐
    private String lastName;  ──┴──> private String name;
}
```

[그림 6] 계층형 아키텍처의 User Entity 변화

```
package user.service;

@Service
public class UserService {
        ...
    public void signup(String username, String password,
                       String email, String firstName, String lastName) {
        User user = new User(
                username,
                passwordEncoder.encode(password),
                email,
                firstName, ──┐
                lastName   ──┴──> "%s %s".formatted(firstName, lastName)
              );
        userRepository.save(user);
    }
}
```

[그림 7] 계층형 아키텍처의 비즈니스 로직 변화

헥사고날 아키텍처에서 도메인 모델이 변경되지 않았다는 것은 어댑터의 변경으로 외부 변경을 처리할 수 있음을 의미한다. 따라서 [그림 3]과 같이 데이터 접근을 담당하는 포트에서 firstName 과 last Name 을 입력 받아 이를 name 으로 합침으로써 도메인 엔티티에는 변화를 주지 않고도 데이터베이스의 변경을 반영한다.

이러한 변경은 도메인 로직에도 직접적인 영향을 미칠 수 있어, 관련 비즈니스 로직도 수정되어야 한다.

이는 단일 책임 원칙과 개방-폐쇄 원칙, 인터페이스 분리 원칙을 위반하며 이는 시스템의 유지보수성과 확장성, 유연성이 저하될 수 있다.

III. 결론

본 논문은 계층형 아키텍처와 헥사고날 아키텍처를 비교하여 외부 변경이 비즈니스 로직에 미치는 영향을 분석했다. 계층형 아키텍처는 결합도가 높아 외부 변화 시 도메인 로직이 영향을 받는 반면, 헥사고날 아키텍처는 비즈니스 로직과 인프라를 분리해 외부 변경에도 로직이 안정적이다. 헥사고날 아키텍처는 확장성과 유지보수가 우수하나 초기 개발 비용이 증가하는 단점이 있다. 그럼에도 불구하고 지속적 유지보수와 확장 필요성이 있는 서비스에는 이 아키텍처 적용이 적합하다고 판단된다.

ACKNOWLEDGMENT

"본 연구는 과학기술정보통신부 및 정보통신기획평가원의 SW 중심대학사업의 연구결과로 수행되었음"(2021-0- 01082)

참 고 문 헌

[1] Richards, Mark. Software architecture patterns. Vol. 4. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Incorporated, 2015.

[2] Martin, Robert C. "Clean architecture." (2017).