테크놀로지 리더를 위한
Media & IT(Information Technology)

# #6. 애플리케이션과 미디어

강자원
컴퓨터시스템응용기술사
/ KBS MNC(Media Network Center)팀


연재 목차



애플리케이션이란, 기술, 시스템 및 제품 등을 사용하는 것을 말한다. 애플리케이션이란 애플리케이션 프로그램, 즉 응용프로그램의 줄임 말이다. 응용프로그램은 사용자 또는 어떤 경우에는 다른 응용프로그램에, 특정한 기능을 직접 수행하도록 설계된 프로그램이다. 지난 호에서 SW 개발방법론에 대한 이야기를 했다면, 이번은 그 방법론으로 완성된 결과에 대한 이야기를 하려 한다. 개발방법론이 어떠하든 간에 결국 사용자가 직접 접하는 것은 애플리케이션이다. 잘된 방법론을 사용해 개발했다 하더라도 결과물이 사용자에게 외면당하면 아무런 의미가 없다. 그래서 IT는 범위도 넓고 각각의 분야가 톱니바퀴 맞물리듯 아귀가 잘 맞아야 한다.

## 내 주변의 흔한 애플리케이션 개발자들 이야기

이상한 나라의 앨리스에서 체셔 캣(이상한 나라에 등장하는 고양이)은 길을 묻는 앨리스에게 목적지가 어디냐고 반문한다. 앨리스는 앨리스답게 어디든 상관없다고 대답한다. 그러자 체셔 캣은 어딜 가든 상관없다면 어떻게 가든 상관없다고 말한다.

66 소프트웨어 개발자의 목적지는 명확하다 - 특정 애플리케이션을 개발하는 것! 99

앨리스처럼 소프트웨어 개발자도 '어딘가(somewhere)'에 가려고 한다. 그러나 개발자가 방향을 신중히 고민해야 한다는 것이며, 어떻게 갈지도 깊게 고민해야 한다는 점이다. "제대로 일하는 개발자는 궁극적인 목표가 최대한 많은 결과물을 내는 것이 아니라 오히려 그 반대라는 점을 알 것이다. 결과물보다 더 중요한 건 결과물의 영향이기 때문이다"

코드는 줄이고, 영향력을 극대화하는 것. 더더욱이 미디어와 같은 대용량의 데이터들을 요구하고 처리하는 애플리케이션이라면 성능에 대해 고민하지 않을 수 없다. 특별히 메모리, CPU, 스토리지, 네트워크 등 그 어느 것 하나 인프라의 성능을 최대로 요구하지 않을 수 없는 데이터들을 처리하기 때문이다. 그래서 코드가 간소화되거나 메모리 사용에 대한 계획 등을 고려한 개발이 아니면 성능에 대한 최적화를 기대하기 어렵다. 그런데, 개발자가 '양 대신 질'을 실천하기란 어렵다. 특히, 미디어 앱 개발 프로젝트의 경우는 프로젝트의 요구사항이 명확하지 않은 경우가 많아 중간에 기능을 수정하거나 추가하는 일이 비일비재하다. 그럼에도 불구하고 납기일은 꼭 준수해야 하기 때문이다.

## 애플리케이션의 코드가 인프라 성능에 미치는 영향

사용자 관점과 애플리케이션을 운영하는 기업에서 미디어 애플리케이션은 [그림 1]과 같은 성능을 보장해야 잘 만들어진 애플리케이션이라 할 수 있다.

| 사용자 관점 | 기업 관점 |
|---|---|
| 쉬운 사용성 | 높은 투자 수익률 |
| 아름다운 디자인 | 실시간 분석 환경 |
| 높은 보안성 | 무중단 서비스 |
| 빠른 응답 속도 | 벤더 독립성 |
| 장애 없는 서비스 | 확장 가능한 인프라 |
| | 데이터 보안 |
| | 법률 규정 준수 |

그림 1. 사용자 관점/기업 관점에서의 애플리케이션의 품질 기준

사용자들에게 친숙한 스트리밍 서비스를 제공하는 애플리케이션을 상상해 보자. 미디어 애플리케이션이 좋은 품질을 유지하려면, 대부분의 성능개선 목표가 응답시간 단축인 경우가 가장 많다. 응답속도에 영향을 주는 것은 많은 인자들이 있다. 그 인자들은 네트워크 회선속도, 스트리밍 서버의 인프라 성능(CPU, 메모리 등), 데이터베이스의 SQL 처리형태 등이 있다. 그래서 애플리케이션은 설계할 때부터 성능 효율성을 고려해야 한다. 효율적인 애플리케이션 디자인의 가장 중요한 측면에 대해 몇 가지 이야기해 보자.

### 비동기 프로그래밍을 통해 응답시간 단축

호출자가 응답을 받는 시간은 밀리초에서 분까지 다양할 수 있다. 이 시간 동안 응답이 다시 오거나 예외가 발생할 때까지 프로세스에서 스레드를 유지한다. 응답 대기 시간 동안 다른 요청을 처리할 수 없으므로 스레드를 보유하는 것은 비효율적이다.

### 요청을 큐에 넣고 일괄 처리하여 더 빠르게 처리

스케일링 성능이 뛰어난 큐는 호출자와 처리 서비스 사이에 있는 스토리지 버퍼다. 큐는 요청을 받고, 버퍼에 저장하고, 큐에 대기 중인 데이터의 안정적인 배달 및 관리를 위한 서비스를 제공하기 위해 요청을 큐에 넣는다. 큐를 활용하면 유연한 디자인이 가능하다.

### 데이터 압축을 사용하여 최적화

압축 전략을 사용하여 웹 페이지 또는 API 응답을 압축하고 번들로 묶는 것이다. 압축은 페이지 또는 API에서 반환된 데이터를 브라우저 또는 클라이언트 앱으로 축소한다. 클라이언트에 반환된 데이터를 압축하면 네트워크 트래픽이 최적화되고 애플리케이션이 가속화될 수 있다.

### 세션 선호도를 사용하여 성능 향상

세션 선호도를 사용하도록 설정하면 애플리케이션에 대한 후속 요청은 첫 번째 요청을 처리한 동일한 서버로 전달된다. 세션 선호도를 사용하도록 설정하지 않으면 부하 분산 규칙에 따라 후속 요청이 사용 가능한 다음 서버로 전달된다.

### 통합 요구사항에 맞게 백그라운드 작업 실행

많은 유형의 애플리케이션에는 사용자 인터페이스와 독립적으로 실행되는 백그라운드 작업이 필요하다. 이러한 작업의 예로 배치 작업, 집약적인 처리 작업, 워크플로우 등의 장기 실행 프로세스가 있다. 백그라운드 작업은 사용자 상호 작용 없이 실행할 수 있다.

애플리케이션을 어떻게 디자인하느냐에 따라 성능에 영향을 미칠 수 있지만, 설계를 잘했으면 개발단계의 품질에 대해서도 고민하지 않을 수 없다. 구현에 있어 로직이 비효율적이거나, 불필요한 메모리를 많이 점유하고 반환하지 않는 구조로 코딩된다면 아무리 애플리케이션의 설계가 좋다 한들 무슨 소용이 있겠는가.

## 엔지니어들이여, 코드품질 향상에 집중해라!!

개발 프로젝트를 하면서 늘 겪었던 문제는 바로 '버그'인 것 같다. 서비스를 오픈하고 테스트 시나리오에 없던 상황이 벌어지면 처음에는 '버그를 일단 찾아야지'라는 생각으로 개발자와 열심히 소스코드 분석을 하고 논의를 하지만 늘 비슷한 상황이 반복되는 것을 느낄 때가 많다. 상황이 반복되다 보면, 이 문제를 해결하기 위한 근본적인 방안을 찾게 된다. 최고의 방안은 아마 많은 개발자들이 이미 알고 있을 것이다. 그것은 바로 "애초부터 버그가 없게 만드는 것이다!!" 하지만 실상은 버그가 없게 만들려고 열심히 설계하고, 테스트하고, 문제가 되는 부분을 리팩토링하지만, 시간이 지나 코드가 변경되거나, 잘못된 설계 또는 리팩토링으로 인해 다시 스멀스멀 버그란 놈은 언제나 나타나게 마련이다. 설계, 테스트, 리팩토링 모두 프로젝트를 담당하는 개발자의 능력, 스킬, 시야에 따라 아주 좋은 품질을 가진 코드가 생성되기도 하고, 그렇지 못한 코드가 생성되기도 하는 것 같다.

그래서, 필자는 프로젝트를 수행할 때 코드 품질을 위해 다음과 같은 개선 기법들을 활용하고 있는지 확인한다. 코드품질을 위해 이미 개발 현장에서 사용하고 있는 기법과 프로젝트 매니저에게 코드품질 개선을 위해 요구할 수 있는 방법들에 대해 알아보자.


### 코드 컨벤션(Code Convention)

읽고, 관리하기 쉬운 코드를 작성하기 위한 일종의 코딩 스타일 규약이다. 쉽게 말하자면, 변수, 함수, 클래스, 메소드 등 이름을 짓는 방식을 동일하게 하는 것을 말한다. 이런 규칙은 성능에 영향을 주거나 오류를 발생시키는 잠재적 위험 요소를 줄여준다. 특히 규모가 큰 프로젝트일수록 유지보수 비용을 줄이는 데 도움이 된다. 컨벤션을 따르지 않는다면, 다른 개발자와 함께 협업하거나 회사에서 일할 때 '상식이 없는 사람' 취급을 받을 수도 있다. 이건 완전 기초 상식!!

### 코드 리뷰(Code Review)

작성된 코드에 대해 다른 사람의 리뷰를 통해 오류를 미리 검출하고 수정하는 것을 이야기한다. 코드 리뷰를 하여 버그를 조기에 발견하고 해결한다. 당장 데드라인이 급하면 코드 리뷰할 시간이 없는 건 당연지사. 따라서, 코드 리뷰는 팀 또는 조직이 코드의 품질에 대해 신경 쓰고 있는지를 알 수 있는 대표적인 수단이며, 많은 개발자들이 '개발 문화가 좋은 곳'의 요소 중 대표적으로 코드 리뷰 진행 여부를 꼽는다.

### CI 시스템(Continuous Integration System)

개발자를 위한 자동화 프로세스를 의미한다. 정해진 스케줄에 따라 SVN에서 최신 소스 파일을 받아, 자동으로 빌드를 수행하고, 실행 가능한 제품을 지속해서 생산해주는 시스템을 이야기한다. 보통 CI 시스템에는 자동 빌드 이외에 자동 테스트, 코드 분석, 이메일 알림 등의 부가 기능들을 제공한다.

### 단위 테스트(Unit Test)

응용 프로그램에서 테스트 가능한 가장 작은 소프트웨어를 실행하여 예상대로 동작하는지 확인하는 테스트이다. 일반적으로 각 함수 단위로 테스트 코드를 작성하여, 작성한 함수에 대한 오류를 검출하여 수정한다. 함수가 수행해야 할 동작에 대해 테스트 코드를 작성하기 때문에, 나중에 코드 리팩토링 또는 기능 개발을 했을 때, 이 단위 테스트를 통과하지 못한다면, 문제가 있는 코드로 생각할 수 있다.

### 코드 커버리지(Code Coverage)

테스트 코드가 프로덕션 코드를 얼마나 실행했는지를 백분율로 나타내는 지표이다. 즉, 테스트 시 소스의 각 라인이 몇 번 실행 됐는지 분석하는 것을 이야기한다. 이를 통해 전체 코드의 범위를 파악하며, 테스트의 취약한 부분을 발견하여 보완할 수 있게 한다.

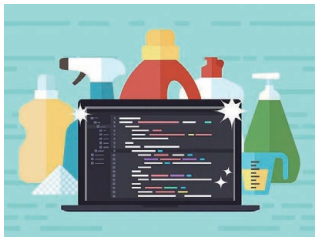
### 정적 분석

정적 분석 도구를 통해 소스 코드 또는 바이너리를 분석하여 잠재적인 결함을 찾아내는 것을 말한다. 문법부터 암묵적인 형 변환, 성능 등 컴파일러에서 제공하는 정보보다 더 많은 정보를 확인할 수 있다. 이를 통해 컴파일러에서 검출하기 어려운 예외 상황에 대해 검출이 가능하지만, 이름 그대로 정적 분석기이기 때문에 런타임 환경에서 동적으로 처리되는 부분에 대해서는 검출이 정확하지 않다.

### 중복 코드 제거

소스 내에서 같은 코드를 제거하여, 추후 유지보수 시 발생할 수 있는 오류를 방지한다. 예를 들어 동일한 코드가 있는 경우, 코드의 위치를 인지하지 못한 상태에서 한쪽의 코드만 수정하는 경우 동일한 다른 코드는 오류를 그대로 가지고 있는 경우가 발생할 수 있다. 동일한 코드가 반복된다면 즉시 리팩토링을 통해, 하나의 함수에서 처리할 수 있게 하는 것이 좋다.

## 처음 만들 때부터, 결함이 없게 만들면 되잖아! 클린코드 (Clean Code)



애초에 결함 없는 응용프로그램을 만들면 되잖아. 안 그래? 클린코드가 중요하다. 클린코드란, 말 그대로 코드를 깨끗하게 짜는 것을 말한다. 그러니까 코드가 군더더기 없이 깔끔하며 기능을 수행하는 코드를 최대한 작은 라인으로 구현했다는 의미로 보면 된다. 최근 '백종원의 골목식당'이라는 프로그램을 보면 음식 메뉴를 줄이고 경쟁력을 키워주는 솔루션이 많이 등장하는데 그 의미와 비슷한 맥락이다.

무엇보다!

실무에서 클린코드의 의미는 유지보수 시간(코드 파악, 디버깅...)의 단축이다. 서비스를 운영하면서 소스코드를 들여다볼 일이 있을 것이다. 그때, 관련된 로직부분을 찾아야 하는데 남이 짠 코드가 술술 읽혀야 좋은 코드인 것이다. 즉, 클린코드는 코드를 처음 보는 사람도 동작을 직관적으로 파악할 수 있도록 해야 한다. 모두가 이해하기 쉽도록 작성된 코드여야 한다. 왜 이렇게 가독성이 중요할까? 일반적으로 기존 코드를 변경하고자 할 때, 해석하는 시간과 수정하는 비율이 10:1이라고 한다. 예를 들면 코드를 변경하기 위해서 걸리는 전체 시간이 10시간이라고 하면, 사전에 코드를 분석하는 시간이 9시간 이상 걸린다는 말로 이해하면 쉬울 것이다. 부분의 결함은 기존 코드를 수정하는 동안에 발생한다고 하니 이해하기 쉬운 코드야말로 오류의 위험성을 최소화하는 셈이다. 따라서 이해하기 쉽게 코드를 작성하는 것이 가장 중요하다.

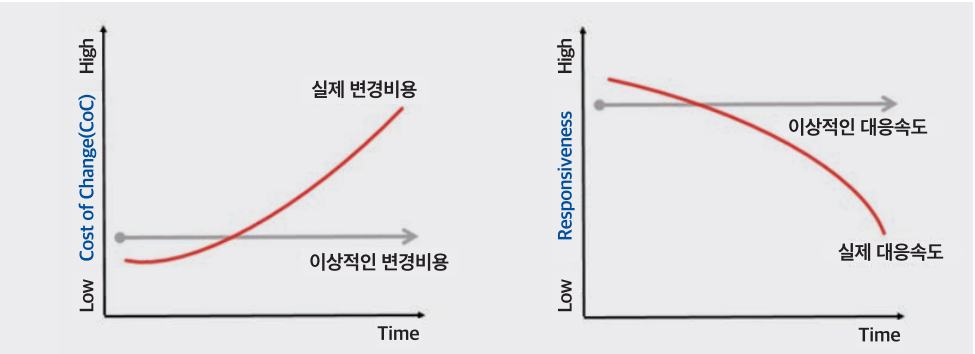



그림 2. 변경 비용과 대응 속도에 대한 이상치와 실제 프로젝트에서 발생하는 수치 비교 / 출처 : Clean Code Sheet

그러면, 클린코드는 어떻게 작성해야 하는가?
다음과 같은 원칙들이 있다. SOLID 원칙이라고도 한다.

**SRP** S
Single
Responsibility
Principle

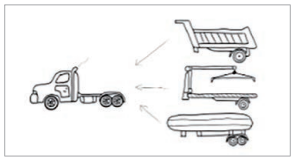
### SRP(Single Responsibility Principle) : 단일 책임 원칙

객체는 오직 하나의 책임을 가져야만 한다. (오직 하나의 변경 이유를 가져야 한다.) 즉, 1개의 함수는 1개의 역할만을 수행하자!

**OCP** O
Open
Closed
Principle

### OCP(Open/Closed Principle) : 개방 폐쇄 원칙

OCP 원칙의 의미는 새로운 기능의 추가가 일어났을 때에는 기존 코드의 수정 없이 추가가 되어야하고, 내부 매커니즘이 변경되어야 할 때는 외부의 코드 변화가 없어야 한다는 것이다.



쉽게 말하자면, 트럭이라는 운송수단과 뒤에 달리는 기구를 분리/결합하는 구조를 만들어 새로운 목적에 해당하는 도구를 만들어야 할 때 트럭 전체를 다시 만들지 않고서 뒤에 달리는 장치만 새롭게 만들면 된다는 뜻이다.

**LSP** L
Lidkov
Substitution
Principle

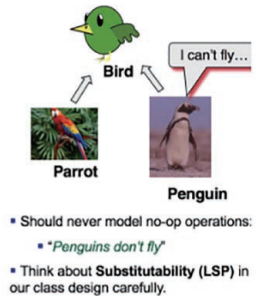
### LSP(Lidkov Substitution Principle) : 리스코프 치환 원칙

프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다는 설계를 참고 한다. 즉, 다형성을 유지할 수 있는 코드를 작성하라는 것이다. 부모의 속성을 자식이 가질 수 없는 코드를 만들지 말라는 것이다. 펭귄을 지우든지 fly()라는 기능을 앵무새에만 두든지 선택해야 한다.

```
class Penguin extends Bird
{
    public void fly()
    {
        // override and print
        System.out.println ("Penguins don't fly");
    }
}
class PlayPet
{
    PlayPet()
    {
        Bird mypet;
        mypet = new Parrot();
        mypet.fly ();  // my pet "is-a" bird, so can fly()

        mypet = new Penguin();
        mypet.fly ();  // BAD if it is a penguin!
    }

    public void PlayWithBird (Bird bird)
    {
        bird.fly(); // OK if it is a parrot
        // BAD if it is a penguin!
    }
```




**ISP** I
Interface
Segregation
Principle

### ISP(Interface Segregation Principle) : 인터페이스 분리원칙

사용자가 필요하지 않은 것들에 의존하지 않게 되도록 인터페이스를 유지하라, 인터페이스를 최대한 작은 단위로 유지 하는 것이 중요하다.

**DIP** D
Dependency
Invension
Principle

### DIP(Dependency Invension Principle) : 의존관계 역전 원칙

"프로그래머는 추상화에 의존하고 구체화에 의존하면 안 된다."

"전기를 이용하기 위해서는 플러그만 꽂으면 된다. (추상화)."
플러그에서 "실제 전기배선이 어떻게 되는지는 사용자는 알 필요가 없다. (구체화)"

이렇게 클린코드에 대한 원칙을 간략하게 알아봤다. 실제로 '코드스멜'이 있을 때, 소스코드 분석을 하려고 보면 정말로 분석하는 데만 엄청난 시간이 걸리는 경험이 다들 있을 것이다.

## 마치며

지금까지 애플리케이션을 만들 때, 주의 깊게 봐야 할 점들에 대해 이야기를 풀어보았다. IT 프로젝트를 할 때 꼭 챙겨야 하는 부분들이다. 실제로 프로젝트는 SI 업체에 맡겨놓고 많은 엔지니어가 통합테스트에만 중점을 두고 기능 구현이 되는지 여부만 확인하는 경우가 대다수이다. 좀 더 꼼꼼하게 테스트 포인트들을 챙겨서 향후 운영과 유지보수까지 신경 쓰는 넓은 시야와 다양한 관점을 지닌 엔지니어로 성장하길 바란다.