

오픈소스 기반 프로젝트 관리

박종규

목차

1. 오픈소스 소프트웨어
 - 1.1. 오픈소스 소프트웨어란
 - 1.2. 오픈소스 소프트웨어의 역사
 - 1.3. 소프트웨어 라이선스
 - 1.4. 오픈소스 커뮤니티
2. 소프트웨어 공학의 이해
 - 2.1. 소프트웨어 공학이란?
 - 2.2. 소프트웨어 개발 계획
 - 2.3. 소프트웨어 개발
 - 2.4. 소프트웨어 관리와 데브옵스
 - 2.5 소프트웨어 공학과 오픈소스
3. 리눅스 시스템
 - 3.1. 리눅스의 역사
 - 3.2. 가상머신에 리눅스 설치하기
 - 3.3. 실습을 위한 리눅스 기초
 - 3.4. 리눅스 네트워크 설정
 - 3.5. 자동화 도구를 사용한 가상머신 설치
4. 개발 협업을 위한 도구, git
 - 4.1. git 기초 사용법
 - 4.2. 버전 관리
 - 4.3. 브랜치 관리
 - 4.4. 원격 저장소
5. Node.js
 - 5.1. Node.js란?
 - 5.2. Node.js를 활용한 웹서비스 실습
6. 가상화
 - 6.1. 가상화와 도커(Docker)
 - 6.2. 도커 기초 사용법
 - 6.3. 도커를 활용한 서비스
7. 깃헙(github)을 이용한 프로젝트 관리
 - 7.1. 깃헙(github)과 깃랩(gitlab)
 - 7.2. 깃헙을 이용한 프로젝트 관리

들어가며

바야흐로 소프트웨어의 시대이다. 이 책을 쓰는 2021년 현재 전 세계 기업 순위 10순위 내의 기업 중 7개의 기업이 IT 관련 기업이며 우리의 생활 또한 많은 부분이 소프트웨어의 영역으로 넘어가서 사람들을 편하게 해주고 있다. 소프트웨어는 해가 갈수록 성장하고 있고 이제 소프트웨어 서비스가 없는 사회는 상상하기가 힘들어졌다.

당연히 소프트웨어 자체의 규모도 점점 커지면서 예전에는 로컬 컴퓨터에 설치하여 소프트웨어 하나를 유저 한명이 사용하던 방식에서, 이제는 네트워크를 통해 프로그램이 서비스화 되면서 소프트웨어 하나를 수백, 수천 명이 동시에 사용하고 있다. 그러다 보니 거대화 된 소프트웨어를 개발하기 위해 개발방법론도 다양하게 변화하였는데, 여기서 오픈소스의 역할이 매우 중요해졌다.

오픈소스는 단순히 공개된 소스코드라는 개념 뿐만 아니라 공개된 프로젝트를 함께 개발하는 개발방법론까지 포함하게 되었다. 특히 대형 소프트웨어를 개발하기 위해 팀 단위로 협력하여 개발하는 경우 팀 커뮤니케이션이 더없이 중요해졌는데 오픈소스의 개발방법론이 여기서 해답을 제시하고 있다. 즉, 오픈소스는 이제 개발자들에게 선택이 아니라 필수가 되었다.

본 책의 주요 대상은 IT 관련 학과 재학생들이며 특히 책의 전반에 걸쳐 IT 관련 용어가 많이 나오므로 어느 정도 프로그래밍을 해본 2~3학년 이상의 학생이 보기에 좋겠다. 책의 전반부는 오픈소스에 대한 굵직한 이야기를 풀어놓고 후반부는 오픈소스를 이용한 프로젝트 관리 실습을 한다. 다만 실습파트를 보고 단순히 프로젝트 관리하는 것을 따라하는 게 아니라 오픈소스가 어떻게 소프트웨어 개발에 도움이 되는지 독자 스스로 고민을 해보아야 한다.

이 책은 다른 오픈소스 관련 서적이나 교재들처럼 독자에게 오픈소스 커뮤니티에 기여하라고 하지 않는다. 오픈소스 기여는 생각 외로 매우 힘든 활동이고 현업에서 종사하며 오픈소스를 십 수년간 다뤄온 개발자들에게도 쉽지 않은 일이다. 다만 [오픈소스 = 무료] 또는 [오픈소스를 사용한다 = 기여한다] 라는 프레임을 깨고 우선 오픈소스를 직접 손쉽게 사용할 수 있는 것에 목표를 두었다.

그리고 본 책을 독파한 것만으로 오픈소스를 다 이해할 수 있을거라곤 생각하지 않는다. 다만 독자들이(IT 관련 학과 재학생일 경우) 졸업을 하고 IT 기업에 취직을 하면 대부분 맨 처음 맞닥뜨리는 것이 오픈소스와 관련된 일일 것이다. 그때 약간이나마 오픈소스에 대한 이해와 공감을 가진 상태에서 주어진 일을 잘 해결해나가길 바란다.

1

오픈소스 소프트웨어

IT와 관련된 분야에 있으면 한 번쯤 오픈소스라는 단어를 들어보았을 수 있다. 그러나 오픈소스가 무엇인냐고 물어보았을 때, 이를 한마디로 설명하기는 쉽지 않다. 단어 의미 그대로 소스코드를 공개하였다는 것일까? 그러면 오픈소스로 개발한다는 것은 공개적으로 개발한다는 의미일까? 첫 번째 장에서는 이 책에서 가장 중요한 핵심인 오픈소스의 의미에 대해서 간략히 알아보도록 한다.

우리가 흔히 오픈소스라고 부르는 오픈소스 소프트웨어(OpenSource Software, OSS)는 기원부터 거슬러 올라가면 사실상 컴퓨터의 역사와 함께한다. 그러나 오래된 역사치고는 그 인식이 상대적으로 약한 것은 부인할 수 없다. 지금도 많은 사람들이 다양한 분야에서 오픈소스를 사용하고 있지만 그것이 오픈소스인 줄도 모르는 경우가 많다. 또한 오픈소스라는 단어를 들어보았다고 해도 제대로 의미를 모르거나 단어에서 뜻을 유추하여 무료 프로그램 정도로만 이해하기도 한다.

대표적으로 우리가 지금 사용하고 있는 유명한 오픈소스는 무엇이 있을까? 필자가 가장 먼저 떠오르는 것은 안드로이드 스마트폰의 운영체제(OS)인 안드로이드(Android)이다. 여기서 조금 더 IT 기술에 관심이 있는 사람은 ‘구글(google) 안드로이드’라고 회사명까지 같이 떠올릴 수도 있겠다. 이 책을 쓰고 있는 2021년 현재 스마트폰의 OS는 Apple사의 iOS와 구글의 안드로이드로 양분되어 있다고 해도 과언이 아니다. 즉, 대부분의 사람들이 알고 있고 또는 쓰고 있는 이 스마트폰의 안드로이드가 바로 오픈소스이다. 안드로이드와 비교할 만큼 유명하지는 않지만 또 다른 오픈소스의 예로는 OBS Studio라는 소프트웨어가 있다. OBS Studio는 사용이 무료이며 녹화나 개인 화면 송출 등의 기능이 편리해 요즘 같은 개인방송 시대에 많은 개인 방송인들이 사용하는 소프트웨어이다. 필자도 비대면 교육시 동영상 제작용으로 많이 사용하였다. 안드로이드와 OBS Studio의 공통점이라고 하면 둘 다 무료로 사용할 수 있다는 점일 것이다. 그러나 무료라는 것만이 오픈소스의 특징은 아니다.

사실 오픈소스라는 단어는 여러 가지 의미를 내포하고 있기 때문에, 오픈소스를 한마디로 정의하기는 쉽지 않다. 특히 오래된 역사와 함께 그 의미가 점점 커지면서 변해왔기 때문에 오픈소스가 가지는 의미는 더욱 많아졌다. 오픈소스는 본질적으로는 **소스코드를 공개하여 누구나 확인이 가능하고 수정하여 사용할 수 있는 코드**를 의미한다. 다만 소스코드를 무작정 공개했다고 하여 그것을 오픈소스라고 할 수는 없으며 공개 이유에 오픈소스의 철학이 담겨야 한다. 여기서 오픈소스 이야기를 할 때 자주 회자되는 명언을 하나 소개하고자 한다.

“누군가가 나의 양초에서 불을 옮겨갔다고 해서 그것이 내 양초를 어둡게 하지는 않는다.
(He who lights his taper at mine, receives light without darkening me.) - Thomas Jefferson”

미국의 정치가였던 토마스 제퍼슨(Thomas Jefferson, 1743~1826)은 무려 200년 전에 오픈소스 정신의 기반이 되는 말을 했다.¹⁾ 물론 당시에는 컴퓨터가 없었으므로 오픈소스 소프트웨어를 대상으로 한 말은 아니었고, 후에 오픈소스의 이념에 찬동하는 사람들에 의해 재조명된 이 발언이 오픈소스의 철학에 완전히 맞아떨어지기에 차용하는 것일테다. 양초의 불을 옮겨서 빛나는 양초가 늘어나면 세상이 더욱 밝아지는 것처럼, 그는 가치있는 생각은 자유롭게 퍼져나가며 더욱 개선되어야 하고, 또 이 가치있는 생각으로 세상을 변화시켜야 한다고 이야기했다. 그리고 이 말은 오픈소스의 공유 가치에 정확히 부합한다.



오픈소스의 가치는 나눔으로써 더 커지는 것이다. 참고로 본 책에서 사용하는 이미지들도 무료 이미지 공유 사이트나 위키의 자료들이다. (<https://pixcove.com>)

소스코드를 공개한다는 것은 토마스 제퍼슨의 발언처럼 누구나 내 양초에서 불을 옮겨갈 수 있게 하는 것과 같다. 그리고 그 소스코드를 통해 더 나은 프로그램을 만들고 또 그것으로 세상을 바꾸길 희망하는 것이다.

오픈소스의 개념은 위와 같이 소스코드의 공개와 공유로부터 출발한다. 비록 이 장의 제목이 ‘오픈소스 소프트웨어란?’ 이지만 이 장에서만 오픈소스를 설명하기에는 한참 부족하며 본 책의 전반에 걸쳐 천천히 그리고 반복적으로 오픈소스에 대한 이야기를 하려고 한다. 독자들은 우선 ‘소스코드를 공개해서 대체 어떤 이득이 있길래 다들 힘들게 만든 코드를 공개하는 걸까?’ 와 같은 의구심을 계속 가지면서 본 책을 읽어주길 바란다.

1) 토마스 제퍼슨은 미국의 3대 대통령이며 우리가 흔히 행운의 화폐라고 부르는 미국 2달러 지폐 초상화의 인물이다.

Note : 안드로이드 오픈소스 프로젝트(Android Open Source Project, AOSP)

사실 엄밀히 말하면 ‘안드로이드’는 오픈소스가 아니라 구글의 저작권이 있는 상표이다. 정확히는 안드로이드 오픈소스 프로젝트(Android Open Source Project), 줄여서 AOSP라고 부르는 이 프로젝트가 오픈소스이며 우리가 흔히 안드로이드라고 부르는 것들은 이 AOSP를 사용하여 개발을 한 다음 구글이 제시하는 테스트들을 거치고 승인을 받아야만 한다. 그래서 구글의 테스트를 받지 않거나 통과하지 못하면 안드로이드의 기능들은 그대로 사용할 수 있으나 ‘안드로이드’라는 이름은 쓰지 못한다.

쉽게 예를 들자면 AOSP는 어떠한 요리의 공개된 레시피이며 이 레시피로 만들 수 있는 요리가 바로 안드로이드이다. 그리고 이 공개된 레시피를 사용해서 요리를 조금씩 바꾸는 것은 괜찮지만 그렇게 만들어진 요리가 레시피를 공개한 사람의 인정을 받아야 되는 것이라고 생각하면 된다. 레시피를 공개한 사람 입장에서는 자신은 김치 레시피를 공개했는데 햄버거를 가져오면 곤란하지 않겠는가?

AOSP는 오픈소스라서 누구든지 소스코드를 확인하여 수정하고 제안할 수 있지만 이 소스코드를 공개하고 관리하는 구글이 이를 승인해주어야 한다. 물론 그 제안²⁾이 합리적이고 안드로이드를 보다 나은 것으로 만든다면 구글이 마다할 이유는 없다.

참고로 2019년에는 미국과 중국의 경제 경쟁의 일환으로 미국이 중국의 화웨이사에 안드로이드를 쓰지 못하게 하는 제재를 가했는데, 오픈소스인 AOSP를 통제하겠다는 것이 아니라 ‘안드로이드’라는 상표를 쓰지 못하게 제재한 것으로 보면 된다. 이에 화웨이는 안드로이드를 포기하고 ‘홍명2.0 (Harmony2.0)’이라는 자체 OS를 개발하여 자사의 스마트 가전제품들에 적용하겠다고 한 상태이다.

2) 이 제안을 오픈소스 커뮤니티에서는 PR(Pull Request)이라고 부르며 PR이 승인되어 소스코드에 반영이 되면 그 오픈소스에 기여(Contribution) 했다고 한다. 대형 오픈소스 프로젝트에 올린 PR이 승인되는 것은 그만큼 어렵지만 승인을 받았다면 해당 오픈소스에 기여했다는 것을 큰 영광으로 여기는 개발자가 많다.

앞서 오픈소스 소프트웨어의 역사는 컴퓨터의 역사와 함께 한다고 하였다. 오픈소스는 자연스럽게 생긴 것이 아니라 초기 컴퓨터의 발전 시기에 개발자들 간의 소프트웨어 공유 이념이 맞부딪히면서 생긴 용어로서 생각보다 복잡한 사연을 가지고 있다. 그러나 기원을 알면 그 이유에 대해서도 한 걸음 더 다가갈 수 있다고 생각하며 본 장에서는 오픈소스 소프트웨어의 시작과 현 주소를 살펴보도록 한다.

혹시 영화 “이미테이션 게임(The Imitation Game, 2014)”을 보았는가? 제2차 세계대전을 배경으로 하는 이 영화는 컴퓨터의 발전에 지대한 영향을 미친 앨런 튜링(Alan Mathison Turing, 1912~1954)이 초창기 형태의 컴퓨터(계산기)를 만드는 내용의 영화이다. 앨런 튜링이 고안한 튜링 머신(Turing Machine)은 현대 컴퓨터 과학에서 없어서는 안 될 중요한 업적이며 이에 대부분이 잘 알고 있는 노벨상, 수학의 필즈상과 같이 컴퓨터 과학계에서는 튜링상을 제정하여 그를 기린다.

갑자기 영화 이야기를 한 이유는 초창기 형태의 컴퓨터에 대해서 이야기하고 싶어서이다. 최초의 컴퓨터에 대해서는 약간 논란이 있을 수 있지만 1946년 펜실베이니아 대학에서 제작된 ‘에니악’이라는 것이 대중적인 견해이다. 당연히 지금과는 모습이 많이 다르고 크기도 거대하여 한번 가동을 하면 그 지역의 가로등이 나갈 정도였다고 한다. 영화 이미테이션 게임에서 나온 계산기도 컴퓨터라기보다는 어떤 공장에나 있을 듯한 거대한 모습이다. 우리가 알고 있는 형태의 개인용 컴퓨터가 나오기까지는 이로부터 대략 30년 정도의 시간이 더 지난 후이다. 현재 컴퓨터에서 가장 많이 쓰이고 있는 x86 아키텍처는 IBM의 IBM 5150 모델부터 시작하였고 이 시기에 ps/2나 ATA 같은 컴퓨터 하드웨어의 표준도 많이 만들어졌다.

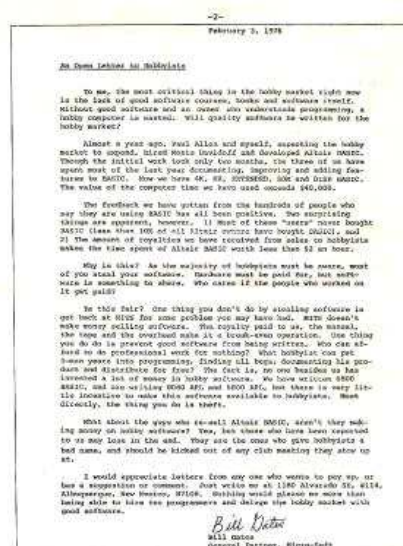


1977년 출시된 애플의 Apple II(좌)와 1981년 출시된 IBM의 IBM 5150(우)

우리가 잘 알고 있는 고급 컴퓨터 언어인 C Programming Language는 1972년에 개발된 언어이다. 물론 C언어가 최초의 프로그래밍 언어는 아니며 그 이전에도 B 언어나 포트란과 같은 컴퓨터 언어는 있었다. 현재와 비슷한 형태의 컴퓨터와 고급 프로그래밍 언어가 대략 1970년~1980년 전후로 발전하고 성행했다는 것만 알면 된다. 이전의 컴퓨터는 너무 비싸고 거대하여 개인이 사용하기에는 무리가 있었다. 대부분 국가시설이나 커다란 기업의 전유물이던 컴퓨터가 개인 사용자에게 보급되고, 소프트웨어를 보다 쉽게 개발할 수 있는 고급 프로그래밍 언어들이 나오게 되자 이제 개발자들은 소프트웨어를 마음껏 만들어 내게 되었다.

필자도 당시 시대의 사람이 아니기에 남겨진 문서를 보거나 필자를 가르친 교수님들에게 들을 수 밖에 없었던 이야기지만, 이 시기의 개발자들은 소프트웨어를 하나 만들면 공동체나 주변의 모든 개발자들과 이를 공유했다고 한다. 필자가 어릴 때 어머니가 집에서 김치를 담거나 부침개라도 구우면 옆집에 나눠주고 하던 모습과 비슷한 것 같다. 유료로 판매하는 패키지 소프트웨어가 없었던 것은 아니지만 워낙에 컴퓨터의 가격이 비싸다 보니 소프트웨어는 하드웨어(컴퓨터)를 구매하면 따라오는 번들과 같은 인식이 강했다. 따라서 소프트웨어에 대한 저작권 등이 정립이 안 된 당시에는 소프트웨어 복사가 지극히 당연했을지도 모른다.

비슷한 시기인 1975년, 우리가 잘 알고 있는 마이크로소프트(Microsoft)사의 공동설립자인 빌 게이츠는 Altair BASIC을 출시하여 판매하였다. 소프트웨어의 라이선스 개념을 통해 상용 소프트웨어를 만들기 시작한 것이다. 그러나 개발자들이 Altair BASIC과 그 외 다른 상용 소프트웨어들을 무분별하게 불법 복제하자 빌 게이츠는 불법 복제에 반대하는 내용의 편지를 써서 대중(익명의 개발자들)에게 공개하기도 하였다. 시간이 흐르며 소프트웨어는 점차 저작권을 인정받기 시작했고, 마이크로소프트는 MS-DOS와 Microsoft Windows와 같은 운영체제를 개발하며 글로벌 소프트웨어 회사로 성장하기 시작했지만, 소프트웨어 자체는 폐쇄적으로 변하고 자본과 기업의 전유물이 되어갔다.



빌 게이츠가 불법 복제에 반대하며
작성한 편지

GNU 프로젝트와 자유 소프트웨어의 등장

1970년 초반 AT&T 벨 연구소의 켄 톰슨(Kenneth Lane Thompson, 1943~)과 데니스 리치³⁾(Dennis Ritchie, 1941~2011)가 개발한 유닉스(Unix)는 당대 최고의 운영체제라 해도 과언이 아니었다. 심지어 오늘날까지도 유닉스의 영향을 받은 많은 운영체제들이 사용되고 있으니 그 영향력을 가히 짐작할 만하다.⁴⁾

MIT 연구실의 연구원이던 리처드 스톨만⁵⁾(Richard Matthew Stallman, 1953~)은 점차 폐쇄적이고 상업화되는 소프트웨어 산업으로 인해 그가 속한 해커 공동체들이 점차 변해가는 것에 회의감을 느꼈다. 본래 해커 공동체에서는 소프트웨어를 서로 공유하고 개발하며 발전해나갔으나 소프트웨어가 상업적으로 변하면서 공유하던 소프트웨어들마저 독점 소프트웨어로 바뀌어버렸다. 이에 리처드 스톨만은 자신이 해커 공동체에서 활동하던 초기의 소프트웨어 공유 정신이 부활하길 바라며 1983년 자유 소프트웨어 프로젝트인 GNU 프로젝트를 제안하였고, 1985년에는 GNU 선언문을 통해 GNU 프로젝트의 취지를 대중에게 널리 알리고 공개하며 참여를 호소하였다. 또한 같은 해 자유 소프트웨어 재단(Free Software Foundation, FSF)을 설립하여 자유 소프트웨어와 GNU 프로젝트를 지원하기 시작했다.

GNU는 “GNU's Not Unix” 라는 문장의 약자에서 따온 이름이며, 유닉스와 마찬가지로 운영체제 중 하나이다.⁶⁾ 그런데 GNU가 그 이름에서 굳이 유닉스를 언급한 점은 왜일까? GNU는 GNU 프로젝트를 통해 개발되며 유닉스의 기능들을 그대로 복제하려고 하였다. 이에 유닉스에 포함된 소프트웨어인 어셈블러, 컴파일러, 디버거, 에디터 등 다양한 소프트웨어를 새로이 개발하는데 성공하였으며 또한 이 소프트웨어들은 자유 소프트웨어로써 공개되었다. 즉, 유닉스와 기능은 동일하지만 유닉스는 아닌 운영체제를 만든 것이다.⁷⁾ 당시 유닉스는 상용화를 위해 만든 소프트웨어는 아니었다고는 하지만 실제로 어마어마한 가격에 팔리는 소프트웨어였고 GNU 프로젝트를 통해 소프트웨어 공유 정신을 발휘하기에 좋은 대상이었다고 생각한다.

다음은 자유 소프트웨어 재단에서 발행한 자유 소프트웨어의 정의와 네 가지 자유이다. 먼저 자유 소프트웨어의 정의는 두 가지 관점을 가지고 있다. 단어가 가지는 내포와 뉘앙스가 상당히 중요한 문장이므로 번역문과 원문을 병기한다.

3) 데니스 리치는 C 언어의 개발자로도 유명하다. 켄 톰슨이 만든 B 언어를 감수하였고 이후 같이 C 언어를 개발한 것으로 알려져 있다.

4) 대표적으로 Apple사의 macOS가 있다. 다만 개인용 컴퓨터에서의 유닉스는 이제 macOS가 거의 유일한 것으로 보이며 대부분은 서버용 OS로 쓰인다. *macOS는 유닉스 계열(unix-like)이 아니라 유닉스이다.

5) vi와 더불어 전통적인 에디터인 Emacs와 GNU 컴파일러 컬렉션인 GCC, 그리고 디버거인 GDB를 만들기도 했다.

6) GNU 프로젝트를 제안한 리처드 스톨만은 GNU를 “그누”로 발음하자고 한다.

7) 커널에 해당하는 GNU 허드는 아직 미완성이며 이후 리눅스 커널이 개발되어 GNU/Linux가 되었다.

우리 이름에서 “자유(free)” 라는 단어는 가격을 의미하는 것이 아니라 자유를 의미한다. 첫 번째로 당신과 당신의 이웃이 사용할 수 있도록 프로그램을 복제하고 재배포할 자유이다. 두 번째로 프로그램이 당신을 제어하는 게 아니라 당신이 프로그램을 제어할 수 있도록 수정할 수 있는 자유이며 이를 위해 소스코드는 반드시 당신에게 제공되어야 한다.

The word “free” in our name does not refer to price; it refers to freedom. First, the freedom to copy a program and redistribute it to your neighbors, so that they can use it as well as you. Second, the freedom to change a program, so that you can control it instead of it controlling you; for this, the source code must be made available to you.

자유 소프트웨어 재단의 설립 이후 이 “자유” 에 대해서는 조금씩 변경이 되어왔고 현재는 소프트웨어에 아래 네 가지 자유의 여부를 판단하여 자유 소프트웨어인지 정의한다.

- 프로그램을 어떠한 목적을 위해서도 실행할 수 있는 자유 (자유 0).
- 프로그램의 작동 원리를 연구하고 이를 자신의 필요에 맞게 변경시킬 수 있는 자유 (자유1). 이러한 자유를 위해서는 소스 코드에 대한 접근이 선행되어야 합니다.
- 이웃을 돕기 위해서 프로그램을 복제하고 배포할 수 있는 자유 (자유2).
- 프로그램을 향상시키고 이를 공동체 전체의 이익을 위해서 다시 환원시킬 수 있는 자유 (자유 3). 이러한 자유를 위해서는 소스 코드에 대한 접근이 선행되어야 합니다.

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

위와 같이 자유 소프트웨어임을 보장하는 라이선스가 바로 GNU 일반 공중 사용 허가서 (GNU General Public License, GNU GPL)이다. GPL에 대해서는 다음 장의 소프트웨어 라이선스에서 상세히 다루도록 하겠다.

오픈소스 소프트웨어의 시작

자유 소프트웨어의 등장은 상업적으로 변해가던 소프트웨어 사회와 개발자들에게 경각심을 주기에 충분했다. 그러나 자유 소프트웨어의 사상에 동조하는 개발자들은 상대적으로 소수였고, 특히 기업들은 소프트웨어로 돈을 벌고 있는데 이러한 소프트웨어를 공유하자고 하는 자유 소프트웨어에 경계심을 가질 수밖에 없었다. 특히 영어권에서는 이 ‘자유(free)’라는 단어가 자기가 마음먹은 대로 할 수 있는 것을 의미하는 자유(liberty) 뿐만 아니라 무료(no charge)라는 의미도 가지고 있기에 더욱 오해를 불러일으켰다. 그리고 소스코드의 공유라는 자유 소프트웨어의 사상에는 동조하지만 GNU 선언문을 통해 발표한 그 자유에 대해서는 반감을 가지는 사람들도 적잖이 있었는데, 소프트웨어의 자유를 이야기하면서 막상 여러 가지 조항이나 규칙들로 개발자들의 자유를 억압하기 때문이었다.

자유 소프트웨어가 나오고 대략 15년의 시간이 흐른 후 1998년 2월, 캘리포니아의 팔로 알토에서 열린 한 세션에서 ‘오픈소스’라는 단어가 만들어졌다. 이는 넷스케이프가 그들의 웹 브라우저 소스코드를 공개하고 난 직후였는데 당시 많은 인기를 끌고 있던 넷스케이프가 소스코드를 공개한 것은 대중들에게도 개발자들에게도 큰 관심을 끌기에 충분했다.⁸⁾ 이에 이 세션에 참석했던 사람들은 자유 소프트웨어 운동을 확대하고 그 가치를 알리기 위해서는 자유 소프트웨어를 조금 더 유연하게 바꿔야 할 필요를 느꼈다. 따라서 ‘자유 소프트웨어’와는 구분되는 새로운 단어가 필요했고 크리스틴 피터슨(Christine Peterson)이 제안한 오픈소스라는 단어로 의견이 모아졌다. 그리고 당시 세션에 참석했던 에릭 레이먼드(Eric Raymond)는 브루스 페런스(Bruce Perens)와 함께 오픈소스 활동을 유지하고 지원하는 ‘Open Source Initiative’를 설립하였다. OSI의 첫 작업은 자유 소프트웨어 재단과 마찬가지로 오픈소스를 정의하는 초안을 작성하고 이에 해당하는 OSI 라이선스 소프트웨어들을 구분하는 일이었다. OSI가 정의한 오픈소스의 정의(Open Source Definition, OSD)는 2007년 마지막 수정 버전에서 총 10가지가 있으며 오픈소스 소프트웨어를 배포할 때에는 각 조건을 준수할 것을 요구한다. 또한 OSI의 검토를 거쳐 승인을 받은 라이선스를 적용하여야지만 정식으로 오픈소스 소프트웨어로 인정받는다.

팀 오라일리⁹⁾(Tim O’Reilly)는 오픈소스 서밋(OpenSource Summit)이라는 행사를 열며 오픈소스라는 단어의 보급에 힘썼다. 이 행사에는 팀 오라일리와 에릭 레이먼드 외에도 리누스 토발즈¹⁰⁾(Linus Torvalds), 귀도 반 로섬¹¹⁾(Guido van Rossum), 래리 월¹²⁾(Larry Wall), 폴 빅시¹³⁾(Paul Vixie) 등 우리가 잘 알고 있는 유명한 개발자들도 참여했다. 이 행사를 통해 오픈소스는 다시 한번 조명을 받았고 많은 개발자들과 기관들의 지지를 받으며 오픈소스 소프트웨어의 시작을 알렸다.

8) 1990년대 초기 웹브라우저 시장에서는 90%에 육박하는 점유율을 달성하였다.

9) 컴퓨터 관련 서적들을 출판하는 O'REILLY 출판사의 대표

10) 리눅스 커널의 개발자

11) 파이썬(Python) 프로그래밍 언어의 개발자. 이 오픈소스 서밋 행사에 대해 귀도가 작성한 수기 형식의 보고서가 있는데 재미있으니 한 번 보았으면 한다.

<https://web.archive.org/web/20131229053622/http://linuxgazette.net/issue28/rosum.html>

12) 펄(Perl) 프로그래밍 언어의 개발자

13) DNS 프로토콜과 시스템, 그리고 안티 스팸메일 시스템의 개발자

아래는 OSI가 정의한 10개의 OSD이며, 내용이 길어 원문은 생략하고 이해를 도울 수 있도록 약간의 번안을 하였다. 그리고 ‘배포’라는 단어가 자주 나오는데, 여기서 이 단어는 동사일 때도 있고 우리가 흔히 ‘배포판’이라고 하는 명사의 의미도 있으므로 감안하고 보길 바란다. 그리고 아래 내용만으로는 이해가 가지 않을 때에는 OSI가 직접 주석을 달아놓은 문서가 있으므로 참고하길 바란다.¹⁴⁾

오픈소스 정의 (The Open Source Definition)

1. 무료 재배포

라이선스는 소프트웨어를 여러 프로그램이 포함된 통합 소프트웨어의 구성 요소로써 제3자에게 판매하거나 양도하는 것을 제한하지 않는다. 라이선스는 판매에 대한 로열티나 수수료를 요구하지 않는다.

2. 소스코드

프로그램은 반드시 소스코드를 포함해야 하고 소스코드와 컴파일된 형태로도 배포할 수 있어야 한다. 어떤 제품을 소스코드 없이 배포 하였다면, 그것은 반드시 합리적인 재생산 비용 이하로 잘 알려진 방법을 통해 소스코드를 얻을 수 있어야 하며, 가급적이면 인터넷을 통해 무료로 다운로드 받을 수 있어야 한다. 소스코드는 반드시 프로그래머가 프로그램을 수정할 때 선호하는 형식이어야 한다. 의도적으로 혼란스럽게 작성된 소스코드는 허용하지 않는다. 전처리기나 번역기의 출력과 같은 중간 단계의 형식은 허용하지 않는다.

3. 파생된 저작물

라이선스는 수정과 파생된 저작물을 무조건 허용하고 반드시 그것들을 원본 소프트웨어와 동일한 조건으로 배포할 수 있어야 한다.

4. 작성자 소스코드의 무결성

라이선스는 빌드할 때 프로그램을 수정할 목적으로 소스코드와 함께 “패치 파일”의 배포를 허용하는 경우에만 수정된 형태로 배포하는 것을 제한할 수도 있다. 라이선스는 반드시 수정된 소스코드로 빌드된 소프트웨어의 배포를 명시적으로 허용해야 한다. 라이선스는 파생된 저작물에 원본 소프트웨어와 다른 이름이나 버전 번호가 필요할 수도 있다.

5. 개인 또는 집단에 대한 차별 금지

라이선스는 어떤 사람이나 집단을 절대로 차별하면 안 된다.

6. 특정 분야에 대한 차별 금지

라이선스는 어떤 특정 분야의 누군가가 프로그램을 사용하는 것을 절대로 제한할 수 없다. 예를 들어 프로그램을 비즈니스에서 사용하거나 유전자 연구를 위해 사용하는 것을 제한할 수 없다.

7. 라이선스 배포

14) 오픈소스 정의 주석 버전 문서. <https://opensource.org/osd-annotated>

프로그램에 첨부한 권리는 해당 당사자가 추가 라이선스를 실행할 필요 없이 재배포한 모든 프로그램에게 반드시 적용되어야 한다.

8. 라이선스는 반드시 제품에만 국한하지 않음

프로그램에 첨부한 권리는 프로그램이 어떤 특정 소프트웨어 배포의 일부가 되는 것과 연관되어서는 안 된다. 만약 배포판에서 추출한 프로그램을 프로그램의 라이선스 조건하에 사용하거나 배포하는 경우 프로그램을 재배포하는 모든 이는 원본 소프트웨어 배포판과 연관하여 부여된 것과 동일한 권리를 가져야 한다.

9. 라이선스는 반드시 다른 소프트웨어를 제한해서는 안 됨

라이선스는 라이선스가 있는 소프트웨어와 함께 배포된 다른 소프트웨어를 절대로 제한해서는 안 된다. 예를 들어 라이선스는 동일한 매체에 배포된 다른 프로그램들이 오픈소스 소프트웨어이어야만 한다고 주장하면 안 된다.

10. 라이선스는 반드시 기술 중립적이어야 함

개별 기술이나 인터페이스의 스타일에 맞춰 라이선스를 제공할 수는 없다.

last modified 2007-03-22

Note : 웹 브라우저의 변화

넷스케이프는 1994년 출시된 웹 브라우저로 처음에는 폭발적인 인기를 끌었으나 그 인기가 오래 가지는 못 했다. 큰 이유 중 하나는 바로 마이크로소프트의 인터넷 익스플로러에게 밀려서인데, 이 당시의 마이크로소프트는 소프트웨어 독점 정책을 강하게 밀고 나갈 때이다. 마이크로소프트는 자신들이 판매하는 OS인 윈도우즈와 마이크로소프트 오피스에 인터넷 익스플로러와 그 외 여러 소프트웨어들을 번들로 포함하여 판매하며 자사 제품들의 시장 점유율을 높였다. 익스플로러에 점점 웹 브라우저 시장 점유율을 빼앗기던 넷스케이프는 1998년에 오픈소스로 소스를 공개하였고 그 이후 웹 브라우저 시장은 한동안 인터넷 익스플로러가 차지하게 된다. 다만 인터넷 익스플로러는 시장을 독점하면서도 웹 표준을 잘 지키지 않았고 특히 한국에서는 액티브X의 폐단으로 인해 많은 사용자들의 지탄을 받았다. 그러나 이 또한 2008년에 등장한 구글의 크롬 브라우저에 의해 점유율 변동이 생겼고 웹2.0이나 웹3.0의 등장, 그리고 컴퓨터 보급의 확대, 인터넷 사용자의 증가 등 다양한 이유로 인해 웹 브라우저 시장에도 조금은 다양성이 생겼다. 그리고 크롬은 현재는 약 70% 이상의 점유율을 차지하고 있으나 출시 당시에 비해 현재는 프로세스가 메모리를 많이 차지하며 무거워졌다는 평가를 받고 있으며 이를 대체하려는 사용자들의 요구에 맞게 다양한 웹 브라우저가 나오고 있다. 전통이 있는 오페라, 모질라 파이어폭스, 사파리 등의 브라우저는 여전히 건재하며 국내에는 네이버가 오픈소스인 크로미움을 사용하여 제작한 웨일(Whale) 브라우저 등이 있다.

자유 소프트웨어와 오픈소스 소프트웨어의 차이

앞서 컴퓨터 발전의 역사와 함께 자유 소프트웨어의 등장 배경 그리고 오픈소스 소프트웨어의 기원에 대해 알아보았다. 그러면 이제 이런 질문이 나올 수 있다.

“자유 소프트웨어와 오픈소스 소프트웨어의 차이는 무엇인가?”

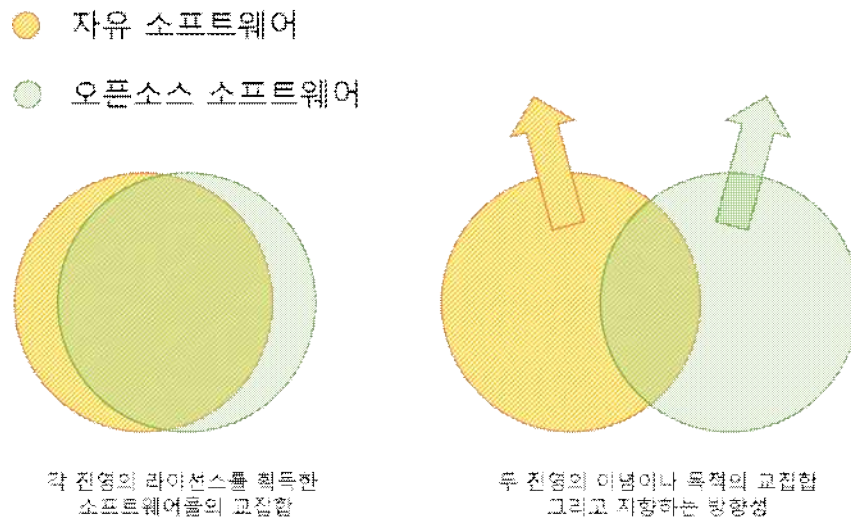
사실 이 질문은 매우 중요하지만 질문을 받는 입장에서는 조금 곱끄러운 질문이기도 하다. 명료하게 대답을 하기 힘들기 때문이다. 그러면 왜 명료하게 대답을 하기 힘든지에 대해서 이야기를 할 차례이다. 들어가기에 앞서 이 부분은 개인적인 관점에 따라 차이가 발생할 수 있는 부분으로 논쟁의 여지가 존재한다. 이것을 감안하고 본 책 뿐만 아니라 다양한 매체에서 여러 의견들을 확인하기를 권한다.

오픈소스 소프트웨어가 자유 소프트웨어에서 출발하긴 했지만 기본적으로 소스코드의 공유라는 점은 동일하다. 사실 자유 소프트웨어와 오픈소스 소프트웨어는 차이점이라기보다는 유사점이 더 많다고 보는 것이 맞다. 대부분의 사람들 또한 이 둘을 심각하게 구별하지 않으며 ‘자유-오픈소스 소프트웨어(Free and OpenSource Software, F/OSS, FOSS)’라고 함께 묶어 부르는 경우도 흔하다. 실제로 소프트웨어 관점에서는 자유 소프트웨어의 ‘자유’도 가지고 있으면서 오픈소스 소프트웨어의 OSI 라이선스 승인을 받은 소프트웨어가 많다. 즉, 하나의 소프트웨어가 자유 소프트웨어이기도 하고 오픈소스 소프트웨어이기도 한 것이다. 다만 자유 소프트웨어는 오픈소스 소프트웨어인 경우가 많지만 오픈소스는 자유 소프트웨어가 아닌 경우가 종종 있다. 애초에 오픈소스 소프트웨어가 자유 소프트웨어에서 갈라져 나온 이유는 자유 소프트웨어의 규정이 너무 완고했기 때문이다. 초기 오픈소스 소프트웨어 지지자들은 소프트웨어 공유의 취지에는 공감하나 거의 윤리적 강령이나 도덕적 규범에 가까웠던 자유 소프트웨어의 규정이 기업이나 개발자들을 힘들게 한다고 생각했다. 따라서 오픈소스 소프트웨어는 이러한 부분을 탈피하고 최대한 실용적인 목적으로 소스코드를 공유해야하는 근거를 마련하기 시작했다. 즉 소스코드를 공개하고 공유함으로써 생길 수 있는 일들이 기업이나 개발자들에게 도움이 된다는 것을 알리기 시작한 것이다.

다시 한번 정리하자. 자유 소프트웨어도 소스코드를 공유해야 하는 이유를 언급하긴 했지만 그 이유는 (그 소프트웨어를 사용할) 너와 네 이웃들을 위해서였다. 즉 윤리적이고 도덕적인 이유였으며 특히 자유 소프트웨어 라이선스인 GPL의 소스코드를 일부라도 사용하면 전체 소프트웨어의 소스코드를 공개해야 했기에 기업 입장에서는 사실상 적용하기 힘든 규범이었다.

그러나 오픈소스 소프트웨어는 이러한 제약을 완화하여 오픈소스로 공개된 소스코드를 사용하여 소프트웨어를 만들더라도 그 소스코드를 공개하게끔 강요하지는 않았다. 즉, 기업이나 개발자들이 오픈소스로 공개된 소스코드를 활용하여 2차 창작을 하고 상업적 이용을 하는 것을 제한하지 않았기 때문에 기업 입장에서는 자유 소프트웨어보다는 선호할 수 밖에 없었다. 다만 오픈소스를 가져다가 사용하는 기업이나 개발자 뿐만 아니라 자신의 소스코드를 오픈소스로

공개하는 기업이나 개발자들에게도 소스코드를 공개했을 때 이득이 있어야 하는데, 이 이득에 대한 모델로 오픈소스는 공개된 소프트웨어가 좀 더 나은 소프트웨어가 될 수 있도록 개발방법론을 제시하였다. 이 개발방법론에 대해서는 뒤에서 다시 이야기하도록 하자.



이렇게 자유 소프트웨어와 오픈소스 소프트웨어를 비교하니 마치 자유 소프트웨어는 쓰기 힘들고 규정을 강요하기 때문에 나쁘고 안 좋은 것처럼 보일까 걱정이다. 게다가 오픈소스 소프트웨어는 사용에 있어 사용자를 그렇게 많이 귀찮게 하지 않기 때문에 상대적으로 자유 소프트웨어에 반발감을 느낄 수도 있다. 하지만 자유 소프트웨어와 오픈소스 소프트웨어는 앞서도 말했듯이 유사한 점이 더 많으며 두 진영이 서로 적대 관계에 있는 것도 결코 아니다. 필자는 그저 두 소프트웨어 공유 이념이 발생한 철학적 이유와 그 방향성만 조금 다른 것이라고 생각한다. 이렇듯 자유 소프트웨어와 오픈소스 소프트웨어는 유사한 점은 많고, 이념이나 목적은 사람마다 관점의 차이로 인해 다르게 볼 수 있기 때문에 사실 둘을 명확하게 구별하는 것은 어렵다. 또한 이 책에서도 앞으로 자유 소프트웨어와 오픈소스 소프트웨어의 구별이 필요한 경우가 아니면 그냥 오픈소스라고 통틀어 지칭한다.

다만 필자는 오픈소스 소프트웨어의 괄목할만한 성장세는 역시나 그 방향성에 있다고 생각하는데, 오픈소스의 공유 정신이나 개발방법론은 이제 소프트웨어를 넘어서서 다양한 문화와 기술 등 사회 전반에 걸쳐 서서히 녹아들고 있다고 생각한다.

가까운 데서 한번 예를 찾아보자. 최근 요식업으로 성공하고 방송가에서도 이름을 알리고 있는 모 사업가는 자신의 식당에서 판매하는 요리의 레시피를 TV 방송이나 인터넷 개인 방송으로 가감 없이 공개하고 있다. 또한 농업인들을 돕자며 특정 식재료의 요리 레시피를 공개하며 식재료 소비의 촉진을 유도한다. 그는 왜 레시피를 공개하고 남을 배려하는 이타심을 발휘하는 것일까? 그러한 행동이 그에게 도움이 되는 것일까? 마찬가지로 오픈소스 소프트웨어의 공개 행위는 과연 우리에게 어떤 도움이 되는 걸까? 한번 고민을 해보았으면 한다.

오픈소스의 성장

오픈소스(자유 소프트웨어를 포함하여)는 등장 이후 차츰 소프트웨어 생태계에서 영향력을 펼쳐갔다. 앞에서는 일반인들도 알만한 오픈소스를 고르다 보니 안드로이드 정도의 예만 들었지만 우리가 지금도 사용하고 있는 오픈소스는 무수히 많다. 특히 IT 관련학과라면 한 번쯤은 들어보았을만한 오픈소스들을 한번 열거해본다.¹⁵⁾



먼저 오픈소스의 대명사라고 해도 과언이 아닌 리눅스가 있다. 아직 리눅스를 사용해보지 않은 독자들도 있을테지만 우리가 지금 실제로 사용하고 있는 서비스(인터넷 쇼핑몰, 배달 서비스, 스트리밍 서비스, 온라인 게임 등등)들은 상당수 리눅스 서버에서 실행되고 있다. 그리고 구글의 웹브라우저인 크롬(Chrome)과 네이버의 웨일(Whale) 브라우저도 오픈소스인 크로미움(Chromium)으로 만들어졌다. 이러한 웹 브라우저로 인터넷을 사용할 때 사용자에게 정보를 제공하는 웹 서버인 아파치 서버, 톰캣(Tomcat), NGINX, Node.js도 오픈소스이다. 페이스북과 구글이 만들고 공개한 리액트(React)와 플러터(Flutter)는 웹과 안드로이드 앱, iOS 앱에 동일한 사용자 서비스를 만들기 위한 최고의 선택이다. 최근의 대규모 클라우드 서비스를 가능하게 해주는 것은 오픈소스인 도커(Docker)와 쿠버네티스(Kubernetes)이며 구글이 제공하는 딥러닝 소프트웨어인 텐서플로우(TensorFlow)도 오픈소스이다.

위에서 언급한 오픈소스들은 당연히 소프트웨어 개발사에서 사용할 뿐만 아니라 해당 영역에서는 대체가 불가능할 정도의 소프트웨어들이다. 그럼에도 불구하고 또 언제 이들을 대체할 오픈소스가 나올지는 장담할 수 없다. 그만큼 오픈소스의 생태계가 빠르고 변화무쌍하기 때문이다.

오픈소스 시장의 점유율이나 사용율을 언급하는 것은 큰 의미가 없어 보인다. 조사기관마다 조사한 오픈소스의 분야와 바로미터가 달라서 결과가 상이했기 때문이다. 그만큼 오픈소스의 분야가 넓고 다양하며 빠르게 변하는 것이 기준을 잡기 어려운 이유이다. 이에 현재 소프트웨어 개발을 하기 위해서 거의 필수라고 여겨지는 오픈소스들을 언급하였으며 아마 현직 개발자들 중 해당 분야에서 일하는 사람들은 상기 오픈소스가 없다고 하면 많이 괴로워할 것이다. 이것만으로도 오픈소스의 위상과 오픈소스가 얼마나 성장했는지 체감할 수 있었으면 좋겠다.

15) 책을 쓰고 있는 지금은 2021년이며 이미 수 십 년간 사용되고 있는 오픈소스도 있고 비교적 최근의 것도 있다.

소프트웨어 라이선스는 소프트웨어와 그 제작자를 주체적으로 인정하고 권리를 지켜주기 위해 나왔다. 소프트웨어를 구매하여 소유권을 가져도 이 라이선스의 범위 내에서 사용하여야 하며 특히 소스 코드가 자유롭게 공유되는 오픈소스에서는 이 라이선스를 반드시 지켜야 한다. 이 장에서는 소프트웨어 라이선스의 종류와 그 내용에 대해서 알아본다.

최근에는 정말 많은 것들이 온라인을 통해 이루어지면서 사실상 인터넷 없이 생활하는 것을 상상하기는 힘들다. 온라인 쇼핑물을 통해 물건을 구매하고 음식을 주문하며 직장이나 학교 생활에서 일이나 학업도 온라인을 통해 이루어지는 경우가 많다. 수없이 많은 예시가 있겠지만 기본적으로 이러한 서비스는 재화의 교환이다. 직접적으로 내 돈을 지불하고 물건을 구매하는 경우도 있고 마케팅의 일환으로 내가 시간이나 그 외 무언가를 투자하면 가치 있는 것을 얻을 수 있다. 이런 재화의 교환은 아주 오랜 옛날부터 인간이 문명을 이루면서 그 방법만 조금씩 바뀌었을 뿐이지 여전히 사회를 구축하는 한 형태이다. 예를 들어 재화의 교환에 문제가 생겨 분쟁이 생기고, 이를 막기 위해 법이 생기며, 법을 제정하고 집행하는 계급이 생겨났다. 물론 이는 근거가 빈약한 예시일 뿐이며 닭이 먼저냐 달걀이 먼저냐 하는 문제로 볼 수도 있지만, 재화의 소유권은 옛날부터 아주 중요한 문제였다. 그러나 실체를 가진 물건 중 주고 받을 수 있는 크기의 물건들은 소유권의 이전이 그나마 쉬웠지만 논밭과 같은 토지나 건물의 경우는 소유권을 증명하기가 쉽지 않다. 아마 이와 관련하여 우리나라에 전해오는 우화로는 봉이 김선달이 대동강의 물을 제 것 마냥 팔아서 이익을 취했다는 이야기를 다들 알고 있을 것이다. 때문에 동서양을 막론하고 사람이 사유할 수 있는 실체에다가 재산의 의미를 부여하고 동산이나 부동산 등으로 구분하여 관리하기 시작했다. 여기에는 문학 작품이나 예술 작품의 창작에 대한 권리도 포함되었는데, 저작권은 창작물을 만든 사람에 대한 보호 권리로 각 국가마다 조금씩 다른 규칙을 가지고 있으며 국제법으로는 베른 협약¹⁶⁾에 바탕을 두고 있다. 우리나라에서도 저작권은 지식재산권에 포함되어 법률에 의해 보호 받고 있다.

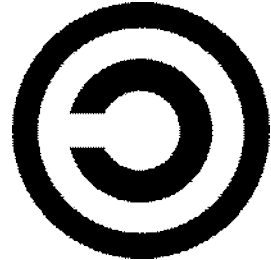
저작권은 영어로 카피라이트(Copyright)라고 하며 문학이나 예술 작품 등 창작물에 대해 저작자가 가지는 권리이다. 일반적으로 만든이의 권리 보호를 위해 창작물의 복제, 공연, 전시, 방송, 전송 등의 이용을 제한할 수 있다. 만약 내가 어떤 물건의 디자인이 마음에 들어서 그 물건을 구입한 다음 동일한 디자인으로 똑같은 물건을 만들어서 팔면 어떻게 될까? 물건을 구매했으니 그 물건의 저작권이 나에게 있지 않을까? 당연히 답은 아니다. 구매한 그 물건에 대한 소유권은 비록 나한테 있을지라도 물건의 저작권은 별도로므로 만약 진짜로 똑같은 물건을 만들고 싶다면 제작자에게 저작료를 지불하고 허가를 구하던가 저작권을 구매하는 방법 등이 있을 것이다. 참고로 지적 재산권¹⁷⁾ 중 저작권은 표현에 대한 권리를 보호하며 특허는 기술이

16) 1886년 스위스 베른에서 체결된 저작권에 관련한 최초의 국제 협약

17) 지적 재산권(Intellectual Property Rights): 지식 재산권이라고도 하며 보통 줄여서 IP라고 많이 부른다. 지식, 정보, 기술, 표현 등등 보호해야 할 무형의 재산에 대한 권리를 말한다.

나 아이디어를 보호한다.

앞서 자유 소프트웨어 재단을 설립한 리처드 스톨만은 소프트웨어의 사유화 및 독점에 반대하며 저작권과 조금 다른 개념인 카피레프트(Copyleft)를 주장했다. 카피레프트는 저작자가 자신의 창작물을 공개하여 다른 사람이 이를 사용할 때, 저작자가 창작물에 포함한 권리를 이행해야 한다는 것이다. 중요한 점은 카피레프트는 저작권을 반대하는 것이 아니라, 오히려 저작권을 인정하면서 창작물을 공유하고 이를 보호하는 것이다. 리처드 스톨만은 저작권으로 창작물을 보호하기만 하며 공유하지 않고 사유화 하는 것을 경계한 것이다. 오히려 어떠한 소프트웨어가 카피레프트라고 하여 저작권을 무시하며 마음대로 사용하거나 수정하여 재배포할 시에는 해당 소프트웨어의 권리에 따라 제재를 받을 수 있으니 주의해서 보아야 한다.



카피레프트의 로고

다시 정리하면 카피라이트와 카피레프트의 차이점은 카피라이트는 독점¹⁸⁾을 위한 권리 보호이며 카피레프트는 공유를 위한 권리 보호라고 생각하면 될 것 같다. 참고로 카피레프트의 로고는 카피라이트의 로고인 ©를 뒤집어 놓은 모양이다.

GNU General Public License

자유 소프트웨어 재단에서는 GNU 프로젝트에 사용할 카피레프트 라이선스로 GNU General Public License, 줄여서 GNU GPL 또는 GPL이라고 부르는 라이선스를 만들었다. 자유 소프트웨어 재단이 인정한 자유 소프트웨어는 GPL을 가지고 있으며 이 자유 소프트웨어는 판매하거나 무료로 제공할 수는 있지만, 만약 소스코드를 수정하여 새로운 프로그램을 만들어 재배포한다면 그 프로그램도 카피레프트 라이선스를 가져야 한다. GPL은 알려진 카피레프트 라이선스 중에서 가장 대표적이며 1989년 처음 발표된 버전을 GPLv1로 부르며 1991년 GPLv2를 발표하였고 현재는 2007년 발표된 GPLv3가 가장 최신이다. GPL은 상당히 강력한 규칙을 가지고 있으며 이 때문에 사용하려면 잘 확인을 하고 사용을 해야하며 아래 다섯가지 의무를 라이선스의 내용으로 포함하여야 한다. 내용을 보면 자유 소프트웨어 재단이 말하는 자유와 비슷한 것을 알 수 있다.

1. 프로그램을 어떠한 목적으로든지 사용할 수 있다.
2. 프로그램을 판매하거나 배포시 소스코드도 함께 공개해야 한다.
3. 프로그램의 소스코드를 용도에 따라 변경할 수 있다.
4. 변경된 프로그램 역시 프로그램의 소스코드를 반드시 공개해야 한다.
5. 변경된 프로그램 역시 반드시 똑같은 GPL 라이선스를 취해야 한다.

18) 여기서 독점은 시장지배구조만을 의미하는 것뿐만 아니라 창작물을 공유하여 사용하지 않는 것을 의미한다.

GPL이 의무하는 이 다섯가지 조항들은 보다시피 상용 소프트웨어 기업에서 선뜻 받아들이기 어려워 보이는 내용이 많다. 앞서도 말했지만 이러한 이유로 기업이 사용하기 쉽도록 오픈 소스 소프트웨어가 새로이 나오기도 하였다. 실제로 GPL은 소프트웨어 라이선스 중 그 규칙이 까다로운 편에 속하며 자유 소프트웨어 재단에서도 이 때문에 규칙을 조금 완화한 LGPL(GNU Lesser General Public License)을 새로이 내기도 했다. LGPL은 소스코드를 라이브러리로 사용하는 것은 소스코드를 공개하지 않아도 된다.

그럼 GPL 소프트웨어를 사용한다는 것은 무엇을 말하는 걸까? 먼저 사용(use)에 대해서 분간을 해야 하는데, 우리가 소프트웨어 라이선스 장에서 계속해서 이야기하는 것은 소스코드의 수정 및 재배포에 관한 것이다. 즉 소프트웨어의 사용과 소프트웨어 소스코드의 사용을 분리해서 봐야 하며, 소프트웨어를 사용하는 것은 해당 소프트웨어를 사용하여 어떤 출력물을 얻는 것이고, 소스코드를 사용하는 것은 해당 소스코드를 수정하여 새로운 소프트웨어를 만드는 것이다. 따라서 GPL 소프트웨어를 사용한다고 하면 GNU의 라이선스를 가진 소프트웨어를 사용하는 것도 되지만, 라이선스와 관련하여 이야기하면 GNU 라이선스를 가진 소프트웨어의 소스코드를 사용하는 것으로 보는 게 좀 더 일반적인 접근이다.¹⁹⁾ 또한 라이선스를 사용한다는 표현 또한 해당 라이선스를 보유한 소프트웨어의 소스코드를 사용한다고 보면 된다.

그럼 소스코드가 아닌 GPL 소프트웨어를 사용하면 GPL이 어떻게 적용될까? 예를 들면, 우리가 가장 흔하게 접하는 컴파일러 중 GCC²⁰⁾는 리처드 스톨만이 직접 만든 컴파일러이며 당연히 GPL을 가진 자유 소프트웨어이다. 그럼 GCC를 사용하여 만든 프로그램은 전부 GPL 소프트웨어인걸까? GPL은 GPL 소프트웨어를 사용하여 나온 출력물은 파생 저작물로 보지 않는다. 그러나 만약 GCC에 파이썬 컴파일러를 추가하고 싶어서 GCC의 소스코드를 수정하여 개선을 했다면, 이는 파생 저작물로 간주되어 GPL 소프트웨어가 된다. 다만 이렇게 수정한 프로그램을 만든 개인이 혼자 사용하거나, 또는 만든 회사 내부에서만 사용한다면 공개할 필요는 없다. 그러나 이를 외부에 재배포 하거나 판매를 한다면 반드시 소스코드까지 공개해야한다. 정리하자면 배포나 판매를 위해서 GPL 소프트웨어의 소스코드를 수정하여 파생 저작물을 만드는 것이 GPL 소프트웨어의 라이선스가 발현되는 사용 범위이다.

본 책에서도 ‘오픈소스를 사용한다’ 라거나 ‘오픈소스로 개발한다’ 라는 표현은 소프트웨어를 직접 사용하는 것이 아닌 오픈소스로 공개된 소프트웨어의 소스코드를 활용하여 새로운 소프트웨어를 개발한다는 표현으로 이해를 하면 된다.

19) 종종 ‘GPL 라이선스’라고 표현하는데 GPL에 이미 라이선스라는 의미가 있으므로 중복되는 표현이다. 그러나 국내 뿐만 아니라 해외에서도 자주 ‘GPL License’라고 표현하는 편이다.

20) 원래 GCC는 ‘GNU C Compiler’였는데 이후 C 언어 외에도 C++, Java 등 다양한 컴파일러를 지원하면서 이름을 ‘GNU Compiler Collection’으로 바꾸었다. 여전히 약자는 GCC를 유지한다.

대표적인 오픈소스 소프트웨어 라이선스

소프트웨어 라이선스의 종류는 대단히 많지만 조건이나 규칙에 따라 몇가지 갈래로 구분된다. 우선 GPL은 카피레프트 라이선스로써 GPL을 사용하는 파생 저작물에도 GPL을 적용하였다. 이에 GPL은 자유-오픈소스 소프트웨어 라이선스이지만 카피레프트로 인해 반환 의무(Reciprocal)를 가진 라이선스로 구분하며 이와 반대로 카피레프트의 성격을 가지지 않거나 최소화 하면서도 오픈소스로써의 성격을 보장하는 라이선스를 퍼미시브(Permissive) 라이선스로 본다. 이에 속한 대표적인 라이선스로는 BSD 라이선스, 아파치 라이선스, MIT 라이선스가 있다. 카피레프트가 적용되지 않기 때문에 독점적 저작권을 발휘할 수 있으며 또 오픈소스로써의 혜택을 볼 수 있기 때문에 대부분의 기업들이 선호하는 라이선스이다.

BSD 라이선스는 버클리 대학에서 나온 라이선스로 누구나 수정 가능하고 제한없이 배포할 수 있다. 그저 저작자의 이름과 BSD 라이선스의 내용을 같이 배포하면 된다. 카피레프트가 아니므로 BSD 라이선스 소프트웨어를 수정하여도 BSD 라이선스를 적용할 필요가 없고 소스코드의 공개 의무가 없으며 상업적 이용에도 제한을 받지 않는다. 공공기관에서 나온 라이선스 답게 공공의 영역으로 반환하고자 하는 의지가 담겨있다고 볼 수 있다. 추가적으로 선택 가능한 조항으로 인해 GPL과 완벽하게 호환되지는 않는다.



소프트웨어 라이선스의 종류와 구분 (출처 : 공개SW 포털)

아파치 라이선스는 아파치 소프트웨어 재단에서 만든 라이선스로 BSD 라이선스와 마찬가지로 사용과 수정 및 재배포 등이 자유롭다. 재배포시에도 수정한 소스코드를 반드시 포함해야 하는 것은 아니지만 아파치 라이선스는 반드시 포함해서 이 소프트웨어가 아파치 소프트웨어임을 알려야 한다. 특이점은 특허의 사용 또한 자유롭기 때문에 만약 특허와 관련된 소프트웨어를 만들고자 한다면 이 점을 알아두어야 한다. GPLv3과 호환된다.

마지막으로 MIT 라이선스는 BSD 라이선스를 기초로 만든 라이선스로써 거의 제약이 없는 아주 약한 라이선스이다. 거의 모든 것이 가능하며 제약이 없고 그저 MIT 라이선스라는 것을 고지하면 될 뿐이다.

아래 표는 GPL과 LGPL 그리고 BSD, Apache, MIT 라이선스의 주요 특징만 추려서 비교한 표이다. LGPL의 경우 대부분 GPL과 의무 사항이 비슷하며 조건을 완화하기 위해 몇 가지 내용이 더 추가되었다.

License	의무 사항	카피레프트	공개 여부	제약 강도
GNU GPL	<ul style="list-style-type: none"> • 배포시 프로그램과 GPL 라이선스 제공 • 파일 수정 사실과 날짜 명기 • 원본 및 파생 저작물을 GPL로 배포 • 원본 및 파생 저작물에 대한 소스코드 제공 	예	예	강함
GNU LGPL	<ul style="list-style-type: none"> • 응용 프로그램을 배포할 경우 LGPL 라이브러리를 사용하고 있다는 사실을 명시 • 사용자가 라이브러리를 수정해도 응용 프로그램을 사용할 수 있도록 허용 	예	예	다소 강함
BSD	<ul style="list-style-type: none"> • 재배포시 저작권 표시 • 준수 조건과 보증 부인에 대한 내용을 고지 사항에 포함할 것 	아니오	아니오	약함
Apache	<ul style="list-style-type: none"> • 배포시 Apache 라이선스 제공 • 수정된 파일에 대해 수정사항 안내 • 특허에 대한 고지사항을 포함 	아니오	아니오	약함
MIT	<ul style="list-style-type: none"> • MIT 라이선스 문구를 복제본에 포함 	아니오	아니오	약함

라이선스의 급증

소프트웨어 라이선스는 현재까지 공개된 것만 해도 수없이 많다. OSI는 자신들의 정의에 맞는 라이선스라면 검토 후 OSI 인증 라이선스라고 승인을 해주는데 이미 OSI 승인 라이선스 수도 100여개가 넘는다. 각 기관이나 단체마다 MIT나 Apache처럼 자신들의 이름을 딴 라이선스를 가지고 싶겠지만 이미 라이선스의 종류가 너무 많아서 이제는 그 수를 늘리기보다 상호 호환성을 확장한다거나 내용의 모호함을 정리하는 것에 더 집중하는 형편이다. 예를 들면 소프트웨어를 개발하는데 A 라이선스를 가진 소프트웨어와 B 라이선스를 가진 소프트웨어 두 가지를 사용했다고 하면, 새로운 소프트웨어에는 A와 B 둘 다 명시해야 한다. 이는 라이선스의 종류가 많을수록 최종 라이선스의 복잡함을 가져오므로 가능하면 호환되는 라이선스의 소프트웨어를 사용하여 간결하게 하는 편이 좋다. 특히 두 개 이상의 소프트웨어 라이선스가 만약 서로 상반된 내용을 가지고 있다면 해당 소프트웨어들은 함께 사용할 수 없다.

Note : 재미있는 비어웨어 라이선스

비어웨어(Beer-ware) 라이선스는 꽤나 재미있는 소프트웨어 라이선스 중 하나이다. 이 라이선스의 사용 조건은 바로 이 소프트웨어를 사용한 사람이 나중에 저작자와 만난다면 맥주를 한잔 사는 것이다. (결국 사용에 거의 제약이 없다는 의미이다.) 놀랍게도 이 라이선스는 GPL과 호환되는 비공식 라이선스로 인정된다. 아래는 이 유쾌한 라이선스의 전문이다.

```
/*
 * -----
 * "THE BEER-WARE LICENSE" (Revision 42):
 * <phk@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
 * can do whatever you want with this stuff.  If we meet some day, and you think
 * this stuff is worth it, you can buy me a beer in return.  Poul-Henning Kamp
 * -----
 */

/*
 * -----
 * "THE BEER-WARE LICENSE" (Revision 42):
 * <phk@FreeBSD.ORG>가 이 파일을 작성함.  이 알리를 유지하는 한 당신은
 * 당신이 원하는 모든 작업을 할 수 있다.  만약 언젠가 우리가 만나서, 당신이 생각하기에
 * 이것이 가치 있다고 생각한다면, 당신은 나에게 맥주 한 잔을 사주면 된다.  폴 해닝 캠프
 * -----
 */
```

GNU GPL의 사용에 대한 문답

GPL은 제약이 많고 사람마다 해석에 대한 이해가 다르기 때문에 사용하기 까다로운 것으로 알려져있다. 아래에는 GPL의 사용에 대해 몇 가지 알려진 질문과 답변을 정리해보았다.

자유 소프트웨어란 GPL을 따르는 소프트웨어를 의미하는 것인가?

- 아니다. GPL 이외에도 많은 종류의 자유 소프트웨어 라이선스가 있으므로 이러한 라이선스들을 따르거나 또는 사용자에게 우리가 말하는 자유를 줄 수 있으면 된다.

두 개의 라이선스가 호환된다는 의미는 무엇인가?

- 두 개의 프로그램을 사용해서 새로운 프로그램을 만드는데 이 프로그램들의 라이선스가 이를 허용 한다면 이는 호환되는 것이다. 그러나 두 개의 라이선스가 동시에 만족되지 않으면 호환되지 않는 것이다.

어떤 라이선스가 GPL과 호환된다는 의미는 무엇인가?

- GPL로 배포된 코드와 그렇지 않은 코드를 사용해서 새로운 프로그램을 만들 수 있으며 또한 이 새로운 프로그램이 GPL이 되는 것을 의미한다.

GPL로 배포한 후에 이를 철회할 수 있는가?

- 이미 공공에 배포하여 많은 사람들이 권리를 가졌기 때문에 철회할 수 없다.

만약 라이브러리가 LGPL이 아닌 GPL이면 이 라이브러리를 사용하는 프로그램은 LGPL인가 GPL인가?

- GPL이다.

GPL 소프트웨어를 독점 소프트웨어 안에 통합시킬 수 있는가?

- GPL 소프트웨어와 통합된 소프트웨어는 GPL 소프트웨어가 확장된 것으로 본다. 즉, 더 이상 독점 소프트웨어라고 할 수 없다. 그러나 방법은 있는데 두 소프트웨어를 분리하여 GPL 소프트웨어와 독립적으로 실행하고, 통신을 통해 GPL 소프트웨어의 결과물을 사용하는 형태가 되어야 한다.

GPL을 법률적으로 강제할 수 있는가?

- GPL은 저작권에 관련된 라이선스이므로 저작자에게 권한이 있다.

만약 프로그램이 리눅스에서만 사용할 수 있는 프로그램이라면 이 프로그램은 GPL인가?

- 비록 리눅스가 GPL 소프트웨어이지만 리눅스에서만 실행할 수 있는 프로그램이라고 하여 GPL인 것은 아니다.

오픈소스의 중요한 요소 중 하나로 오픈소스 커뮤니티를 빼놓을 수 없다. 오픈소스 커뮤니티라고 하여 단순히 오픈소스에 대한 의견을 주고 받는 온라인 공간이 아니라 오픈소스가 오픈소스로서 존재할 수 있도록 생태계를 제공하는 것이 오픈소스 커뮤니티이다. 이 장에서는 대표적인 오픈소스 커뮤니티에 대해서 간략히 알아본다.

오픈소스가 등장한 시기였던 1990년~2000년 무렵에는 우리나라도 컴퓨터가 가정집에 조금씩 보급되고 ADSL과 같은 컴퓨터 네트워크 인프라가 갖춰지는 등 점차 소프트웨어를 접할 수 있는 환경이 마련되었다. 1998년에 미국의 게임 회사 블리자드가 스타크래프트를 발매하고, 또 우리나라에 PC방이 생기기 시작한 무렵이니 대충 어떤 시기인지 짐작해 볼 만하다. 그리고 정말 많은 종류의 소프트웨어 또한 생겼다가 사라지곤 했는데, 이 시기에 등장해서 아직까지 사용되고 있는 소프트웨어가 고프플레이어나 알집과 같은 것들이다.²¹⁾ 이때에는 이런 소프트웨어를 유틸리티 프로그램이라 하여 한데 모아두어 관리하는 웹 사이트도 많았다. 이런 사이트에 접속하면 여러 가지 유틸리티 프로그램을 다운로드 받아 설치하여 사용할 수 있었는데, 관심있게 보았다면 소스코드와 설치파일을 함께 제공하는 것을 볼 수 있었을지도 모르겠다. 아마 소스코드와 함께 배포되었다면 필히 그 소프트웨어가 오픈소스였기 때문일 것이다.

Source distributions:

Platform	Files
Unix/Linux Source (has \n line feeds)	cmake-3.21.1.tar.gz
Windows Source (has \r\n line feeds)	cmake-3.21.1.zip

Binary distributions:

Platform	Files
Windows x64 Installer: Installer tool has changed. Uninstall CMake 3.4 or lower first!	cmake-3.21.1-windows-x86_64.msi
Windows x64 ZIP	cmake-3.21.1-windows-x86_64.zip
Windows i386 Installer: Installer tool has changed. Uninstall CMake 3.4 or lower first!	cmake-3.21.1-windows-i386.msi
Windows i386 ZIP	cmake-3.21.1-windows-i386.zip
macOS 10.13 or later	cmake-3.21.1-macos-universal.dmg
	cmake-3.21.1-macos-universal.tar.gz
macOS 10.10 or later	cmake-3.21.1-macos10.10-universal.dmg
	cmake-3.21.1-macos10.10-universal.tar.gz
Linux x86_64	cmake-3.21.1-linux-x86_64.sh
	cmake-3.21.1-linux-x86_64.tar.gz
Linux aarch64	cmake-3.21.1-linux-aarch64.sh
	cmake-3.21.1-linux-aarch64.tar.gz

오픈소스인 CMake의 다운로드 페이지 화면. 소스 배포판(위)과 설치 배포판(아래)을 각각 제공하는 것을 볼 수 있다.

21) 이외에도 당시에는 무료인 프리웨어로 나왔다가 이후 상업화하여 서비스하고 있는 소프트웨어들이 많다.

당시에 오픈소스를 배포하던 방식은 일반적으로 해당 오픈소스의 홈페이지를 만들어서 다운로드 받을 수 있도록 하는 것이었는데, 사용자들은 자신이 원하는 오픈소스를 찾는 것이 그다지 쉬운 편은 아니었다. 그도 그럴 것이 지금처럼 검색 포털이 잘 동작했던 것도 아니고, 어떤 오픈소스 소프트웨어들이 있는지도 모르는 상태에서 원하는 오픈소스를 찾기는 힘들었다. 또한 어찌어찌 오픈소스를 찾았다 하더라도, 오픈소스에 이해가 어려운 부분이 있어 저작자에게 연락을 하면 답변을 받는데 오래 걸리거나 아예 받지 못할 수도 있었다. 오픈소스는 공유 되었지만 그 방법은 아직 미흡할 수밖에 없었다.

그러다가 소스포지(SourceForge)와 같은 웹 사이트가 생기기 시작했다. 소스포지는 초기 오픈소스 사용자들의 공유와 커뮤니케이션에 대한 갈증을 해소시켜 주었다. 웹에는 수많은 오픈소스들이 등록되었고, 오픈소스에 대해 궁금한 것이 있다면 저작자에게 직접 메일로 연락하는 것이 아니라 토론장을 형성해 관심있는 사람들이 함께 이야기하고 서로 도와주기 시작했다. 오픈소스를 배포하는 홈페이지가 있는 경우에는 소스포지에다가 배포 페이지의 링크를 연결하여 누구나 찾기 쉽게 하였다. 오픈소스 개발자들은 한결 쉽고 편하게 오픈소스를 공유하고 정보를 얻을 수 있었다.

깃헙의 등장

2008년에는 소스포지의 뒤를 이어 깃헙(github)이 등장하였다. 깃헙은 GPL 소프트웨어인 git을 사용하여 오픈소스들을 공유하였다. git은 분산 버전 관리 시스템으로 소스코드의 개발시 동시에 여러 개발자들이 하나의 프로그램을 개발하는데 도움을 준다. git에 대해서는 제4장에서 다시 소개한다.

현재 깃헙에는 세상의 거의 모든 오픈소스가 올라와 있다고 해도 과언이 아닐 정도이며 개인용 저장소도 제공하므로 오픈소스와는 관계없이 개인 프로젝트나 학교 과제 같은 것을 올리기도 한다. 사실 이런 것은 그저 깃헙이 제공하는 여러 가지 서비스 중 하나일 뿐이다.

오픈소스 커뮤니티로서의 깃헙은 우선 견고한 입지를 들 수 있다. 등장 이후 지속적으로 성장을 거듭한 깃헙은 2020년에는 총 5600만명 이상의 개발자가 깃헙에 등록하였고, 6000만개 이상의 신규 저장소가 생성되었다. 그리고 1억9천만개 이상의 풀리퀘스트(Pull-Request)가 승인되어 오픈소스에 기여하였다.²²⁾ 이렇듯 수많은 오픈소스와 사용자들로 인해 오픈소스에 대한 대부분의 정보는 깃헙에서 찾을 수 있게 되었다. 그리고 git을 이용하여 협업이 가능한 웹 서비스를 제공하고 있다. 이 협업은 바로 위에서 언급한 풀리퀘스트와 리뷰라는 개념으로 진행되며 이제는 오픈소스의 개발 모델로 자리를 잡았다. 오픈소스 개발 모델도 제2장에서 다시 보도록 한다.

22) 깃헙은 매년 옥토버스(octoverse.github.com)라는 사이트에서 한 해 동안의 의미있는 통계자료를 공개한다.

Note : 마이크로소프트와 깃헙 그리고 오픈소스

2018년 6월, 전 세계 개발자들을 놀라게 할 만한 뉴스가 나왔다. 바로 마이크로소프트가 깃헙을 인수하겠다고 발표한 것이다. 사실 깃헙은 그 규모를 유지하기 위한 지속적인 투자로 계속 적자를 보고 있었으며, 사실상 글로벌 대기업의 인수합병을 원하는 상황이었다. 그럼에도 불구하고 마이크로소프트의 인수 사실은 놀라운 일이었는데, 왜냐하면 그동안 마이크로소프트는 오픈소스와는 거리가 멀고 독점 소프트웨어로 성공한 기업이라는 이미지가 강했기 때문이었다. 실제로 마이크로소프트는 창립 초기부터 윈도우와 오피스 제품군의 시장 점유까지 독점적인 라이선스를 행사하며 기업을 성장시켰다. 그 뿐만 아니라 마이크로소프트의 무분별한 소프트웨어 개발로 인해 오픈소스 진영이나 표준화 단체와의 마찰도 많이 있었다. 예를 들면 인터넷 익스플로러는 HTML 표준을 지키지 않아 호환이 되지 않는 웹 사이트가 즐비하였다. 또한 마이크로소프트의 대표적인 개발도구인 비주얼 스튜디오에서 사용하는 컴파일러인 VSC 또한 C 언어 표준을 지키지 않은 사례가 있고 마이크로소프트 오피스의 문서 파일들은 표준 문서 포맷(Open Document Format, ODF)을 사용하기보다 .xls, .ppt와 같은 MS 오피스 전용 포맷을 고수하였다. 이러한 마이크로소프트의 지난 행보들은 오픈소스 커뮤니티의 총본산인 깃헙을 인수한다고 했을 때 많은 개발자들로 하여금 걱정을 자아내게 할 수밖에 없었던 것이다.

그러나 막상 깃헙을 인수하고 나자 개발자들의 걱정은 완전히 사라졌다. 깃헙은 이전과 같이 매끄럽게 운영되었으며, 무료로 제공하는 비공개 저장소는 무제한이 되었다. 깃헙은 마이크로소프트의 자회사가 되어 적자 고민에서 벗어났고, 거기다 마이크로소프트 스스로가 현재 전세계에서 깃헙을 가장 많이 사용하는 기업이 되었다. 인수 후에도 등록되는 신규 오픈소스 프로젝트의 수는 폭발적이며 전세계 대부분의 개발자들이 깃헙의 계정을 가지고 있게 되었다.

이러한 성과는 빌 게이츠, 스티브 발머에 이어 마이크로소프트의 3대 CEO인 사티아 나델라의 성향 덕분이라는 분석이 많다. 사티아 나델라는 패키지 소프트웨어 중심이었던 마이크로소프트를 클라우드 서비스와 오픈소스 중심 체제로 바꾸어 놓았다. 심지어 영원히 대립할 줄 알았던 리눅스에 대한 서비스를 시작하였으며 많은 자사의 소프트웨어들을 오픈소스로 공개하기 시작했다. 소프트웨어를 오픈소스로 공개하면서 사내 문화도 오픈소스 개발에 맞물려 개방적이고 협력적인 방향으로 바뀌었으며 기업 경영 지표도 한시적이나마 시가 총액 글로벌 1위를 달성하는 등 우수한 실적을 내었다.

사티아 나델라 취임 이후 마이크로소프트는 회사 자체가 완전히 바뀌었다고 평가 받으며 현재는 오픈소스에 친화적인 기업으로 그 성장을 거듭하고 있다. 마이크로소프트의 사례에 비추어 보아도 소스코드를 공개하는 오픈소스가 기업 경영에 어떤 영향을 미치는지는 생각해보아야 할 문제이다.

2

소프트웨어 공학의 이해

소프트웨어 공학은 소프트웨어의 개발부터 테스트, 운영, 유지보수 등 소프트웨어와 관계된 일련의 프로세스를 체계적으로 정리한 학문이다. 소프트웨어의 개발에 정답은 없지만 소프트웨어 공학에서 제시하는 다양한 모델이나 개발 방법론들은 소프트웨어 개발에 효율적인 방법을 알려준다. 특히 오픈소스가 소프트웨어 개발에 큰 비중을 차지한 이후부터 많은 기업들이 오픈소스 개발 모델을 채용하면서 오픈소스와 소프트웨어 공학은 개발자가 알아두어서 결코 나쁘지 않은 선택이 되었다.

과학(Science)과 공학(Engineering)의 차이는 무엇일까? 이에 대해서는 이미 여러 가지 해석이 있고 또 상황이나 분야, 관점에 따라 다를 수 있다. 예컨대 과학은 자원을 들여서 기술을 만들고, 공학은 기술을 통해 자원을 만든다는 해석도 있다. 또 다른 해석으로는 과학은 자연의 현상을 발견하는 것이고, 공학은 사람의 행위에서 발생한 현상을 발견하는 것이라고도 한다. 컴퓨터 분야의 학문으로 예를 들면, 컴퓨터 과학은 계산 이론, 알고리즘, 인공지능과 같이 수학적 지식을 요구하는 경우가 많고 컴퓨터 공학은 회로 구조, 반도체 설계, 운영체제, 네트워크 등 컴퓨터 시스템의 설계를 위한 이론이 많다.²³⁾

그러면 소프트웨어 공학은 무엇일까? 아주 쉽게 표현하자면 소프트웨어를 잘 만드는 방법을 가르치는 학문이다. 소프트웨어 공학을 통해 소프트웨어를 만드는 것이 아니라 만드는 방법을 배운다는 것에 주목해야 한다. 물론 소프트웨어 공학을 모르더라도 개발을 할 수 있지만 소프트웨어 공학의 많은 내용들은 선배 개발자들이 현장에서 직접 겪은 경험들의 집합이므로 이론으로나마 알면 많은 도움이 될 것이다. 그리고 소프트웨어 공학에서 배우는 것들이 항상 정답은 아니지만 많은 개발자들이 경험한 다양한 상황에서 최선의 선택을 할 수 있도록 도와주는 것이라는 것을 명심하자.

소프트웨어 공학에서는 크게 아래와 같은 분야들을 다룬다.²⁴⁾

- **소프트웨어 요구사항** : 무엇을 위한 소프트웨어인지 그 기능이나 성능, 규격, 제약사항 등 여러 가지 요구사항을 체계적으로 추출하고 분석하는 분야이다. 소프트웨어 요구사항은 소프트웨어 설계와 함께 그 양이 많고 다양하며 표준과 관련된 부분이 많다.
- **소프트웨어 설계** : 소프트웨어의 요구사항과 정의에 따라 사용자 관점, 시스템 관점, 소프트웨어 관점, 개발자 관점 등 여러 가지 방법으로 설계하는 분야이다. 보통 UML(Unified Modeling Language)이라 부르는 언어로 유스케이스 다이어그램이나 클래스

23) 엄밀히 따지면 컴퓨터 과학은 이학 계열로 볼 수 있으나 국내에서는 크게 구분하지 않는 편이다. 대학의 경우 2000년대 초반 공과 대학에 지원을 많이 하였기에 공대로 통합된 경우도 많다.

24) SWEBOOK executive editors, Alain Abran, James W. Moore ; editors, Pierre Bourque, Robert Dupuis. (2004). Pierre Bourque and Robert Dupuis, 편집. 《Guide to the Software Engineering Body of Knowledge - 2004 Version》. IEEE Computer Society. 1-1쪽. ISBN 0-7695-2330-7.

다이어그램, 시퀀스 다이어그램 등 다양한 다이어그램으로 표현을 한다.

- **소프트웨어 개발** : 실제 프로그래밍 언어와 개발 도구 등을 사용하여 소프트웨어를 개발하는 분야이다.
- **소프트웨어 테스트** : 소프트웨어의 기능들이 제대로 동작하는지, 버그는 없는지, 동작 성능은 어떠한지 등등 시스템의 결함을 찾는 과정이며 사실상 모든 결함을 찾기는 불가능하기 때문에 많은 테스트를 통해 소프트웨어의 완성도를 높인다.
- **소프트웨어 유지 보수** : 소프트웨어의 개발을 완료한 후 일으키는 문제에 대한 수정이나 시스템의 향상을 위한 작업이다.
- **소프트웨어 형상 관리** : 소프트웨어의 소스코드 및 버전, 배포판, 문서 등 소프트웨어 관련 자료에 대한 체계적인 관리 방법이다. svn이나 git과 같은 소프트웨어가 대표적인 형상관리나 버전관리 시스템이다.
- **소프트웨어 프로젝트 관리** : 하나 이상의 소프트웨어 개발을 포함하는 프로젝트의 업무 전반을 말하며, 좁은 범위로는 프로젝트 관리 도구를 사용하여 소스코드와 라이브러리, 빌드 파일을 관리하는 것을 의미한다.
- **소프트웨어 개발론** : 소프트웨어를 개발하는 방법에 대한 분야이며 대표적으로 전통적인 폭포수 모델과 비교적 최근의 애자일 모델이 있다. 소프트웨어의 개발을 위한 시작 단계부터 설계, 개발, 테스트, 유지보수 등 소프트웨어 개발 전주기에 대해서 다룬다.
- **소프트웨어 공학 도구** : 소프트웨어를 설계하고, 개발하고, 관리하고 테스트하는 등 소프트웨어 전주기에 걸쳐 도움을 주는 다양한 툴과 그 방법에 대해서 다룬다. 예를 들면 동시에 서버에 100명이 접속하는 것을 테스트 하고 싶을 때, 실제로 100명이 접속하는 것이 아니라 가상으로 접속한 것처럼 테스트하고 그 결과를 확인하는 도구와 그 도구가 동작하는 원리에 대한 내용 등이다.
- **소프트웨어 품질** : 소프트웨어가 더 향상될 수 있도록 사용자 경험, 성능, 인프라, 마케팅 등등 전반에 걸쳐 소프트웨어 품질을 검증하는 분야이다. 소프트웨어 테스트와 종종 비교되지만 엄연히 다른 분야이다.

소프트웨어 공학은 그 내용이 매우 많고 다양하지만 본 장에서는 몇몇 핵심적인 부분들만 다루며 특히 이 장의 마지막에서 설명할 오픈소스와 관련 있는 부분들을 짚고 넘어가도록 한다. 소프트웨어 공학을 배우고 업무에 임한다면 효율적으로 프로젝트를 관리하고 함께 일하는 사람들과 동일한 개발 프로세스를 공유하게 해준다는 점에서 매우 유용한 지식이다.

대부분의 개발자들이 규모가 작은 소프트웨어를 만들 때에는 특별히 계획까지 세우진 않을 것이다. 우선 자리에 앉아서, 코딩을 시작하고, 버그를 잡으며, 이를 반복한다. 그러나 소프트웨어의 규모가 조금씩 커지고, 또 혼자가 아니라 다른 사람과 함께 개발을 해야 한다면 반드시 소프트웨어 개발 계획이 필요하다. 소프트웨어 개발 계획은 함께 개발하는 사람들이 모두 같은 내용을 공유하고 바른 방향으로 개발할 수 있도록 도와준다. 그렇지 않으면 각자가 개발한 코드 조각들이 서로 맞지 않아 시간만 버리게 될 테니까. 이 장에서는 소프트웨어 개발을 하기 전에 해야 할 내용에 대해 알아본다.

혹시 엘리베이터가 어떻게 동작하는지 고민을 해본 적이 있는가? 어느 정도 층수가 있는 고층 건물에 살거나 매일 다니는 학교, 직장엔 엘리베이터가 있다면 거의 매일 엘리베이터를 사용하고 있을 수도 있다. 물론 어쩌다 한번 엘리베이터에 대해 고민을 해본 적은 있지만 대부분은 ‘갑자기 엘리베이터가 멈추면 어떡하지’와 같은 생각일 것이며 그 동작 알고리즘에 대해 깊이 있게 고민해 본 적은 별로 없을 것이다. 그럼 여기서 함께 엘리베이터에 대해 고민을 해보자. 여러분은 이번 장에서 필자와 함께 엘리베이터를 설계해 볼 것이다.

여러분은 한 엘리베이터 회사에 취직을 했다. 그리고 처음으로 일을 맡은 일이 새로 짓는 20층 건물에 들어갈 엘리베이터의 소프트웨어를 개발하는 것이다. 소프트웨어를 어디에 어떻게 설치하는지는 고민하지 말자. 여러분은 소프트웨어만 만들면 된다. 자 머릿속으로 살짝 어떻게 개발하면 좋을지 떠올려보길 바란다. 맨 처음 작성할 코드는 무엇인가?

바로 무언가 떠오른 사람도 있을 수 있고 어디부터 어떻게 해야 할지 막막한 사람도 있을 수 있다. 만약 막막함을 느꼈다면 그건 우리가 만들어야 할 엘리베이터에 대한 정보가 너무 적기 때문이다. 우리가 현재 엘리베이터에 대해 알고 있는 정보는 그저 20층 건물에서 사용한다는 것밖에 없다. 그리고 무언가 바로 떠오른 사람은 우리가 일반적으로 20층 규모의 건물에서 볼 수 있는 엘리베이터의 특징들을 생각했기 때문이다.²⁵⁾ 하지만 우리가 만들어야 하는 게 반드시 그런 일반적인 엘리베이터라고 단정할 수는 없다. 그럼 우리가 엘리베이터 개발을 시작하기 전에 알아야 할 정보는 무엇일까?

먼저 개발을 시작하는 우리 스스로에 대한 분석이 필요하다. 개발에 투입되는 인원은 몇 명이며 누구인지, 명색이 엘리베이터 회사인데 개발에 참고할 수 있는 기존의 엘리베이터 소스코드나 가이드라인은 없는지 등을 확인한다. 회사 내부에 대한 분석을 마쳤으면 이제 우리가 개발하려는 엘리베이터에 대해 알아볼 차례이다.

25) 보통 이런 것을 ‘고정관념’이라고 얘기하기도 한다. 그동안 경험을 통해 축적한 엘리베이터에 대한 고정관념을 사용하여 대상이 가지고 있는 특성들을 빠르게 떠올리는 것이다. 이처럼 고정관념이 항상 나쁜 것은 아니며 사회를 구성하고 인지하는 방법으로 중요한 역할을 한다. 다만 고정관념은 인지 능력의 활용적인 측면에서는 효율적이긴 하지만 잘 알다시피 창의적인 면을 막는다는 단점이 있다.

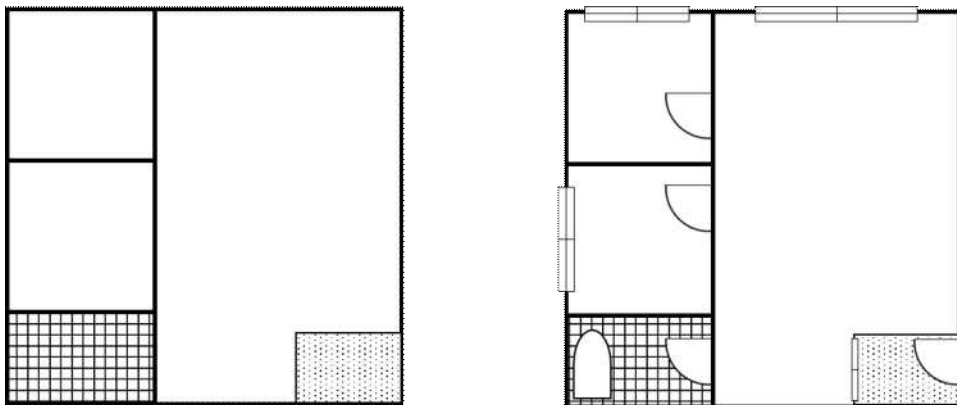
요구사항 분석

요구사항(Requirement)은 시스템이 제대로 동작하기 위해서 필요한 모든 항목들을 말한다. 이러한 요구사항은 기능이 될 수도 있고 성능이 될 수도 있으며 이 요구사항이 자세할수록 우리가 개발해야 하는 시스템이 명확해진다. 앞서 우리는 엘리베이터를 만들기 전에 수집한 정보가 20층 건물에 설치한다는 것밖에 없었다. 승강기는 1개인가 2개인가? 아니면 그 이상인가? 그리고 나머지 정보는 어디서 어떻게 모아야 할까?

일반적으로 요구사항은 시스템을 구매하는 고객이나 사용자로부터 수집을 한다. 고객은 개발자에게 자신이 원하는 것, 즉 요구사항을 전달하며 개발을 부탁하는데 사실 이 과정이 매우 순탄치가 않다. 이해를 돕기 위해 소프트웨어와는 조금 다른 예를 들어 보자. 방 두 개, 화장실, 거실을 포함하여 작은 집을 만들고 싶은 고객이 있다. 고객은 대략적인 방의 크기와 위치를 건축사에게 알려주면서 집을 만들어 달라고 한다. 그러나 실제로 건물을 만들어야 하는 건축사 입장에서는 정보가 너무 부족하다. 그러면 이제 고객에게 역으로 질문을 한다.

“여기 방에서 방문은 어디에 달까요? 혹시 창문은 필요가 없으신가요? 아, 깜박하신 건가요? 그럼 어디에 어느 크기로 달면 좋을까요? 그리고 건물이 들어올 도로의 위치를 보니 화장실이 이 위치에 있으면 나중에 하수 배관에 문제가 생길 수 있습니다. 이렇게 바꿔보시는 건 어떤가요?”

이렇게 건축사는 고객과의 대화를 통해 자신의 지식과 경험을 살려 고객이 미처 생각하지 못한 요구사항들을 이끌어 내준다. 그리고 이 요구사항들을 토대로 대략적인 설계도를 그려서 고객이 생각하는 바와 일치하는지 확인하고, 다시 요구사항을 검토하고 설계도를 그리는 작업을 반복한다. 이러한 작업을 요구사항 유도 또는 요구사항 수집 등으로 부른다.



방 두 개, 화장실과 거실이 있는 집을 만든다고 할 때 어느 쪽 설계도를 참고해야 팀원들 간에 문제가 없을까?

안타깝게도 요구사항을 이끌어내기 위한 고객과의 대화가 항상 쉬운 것은 아니다. 기본적으로 하나의 시스템을 둘러싼 여러 이해관계자들의 생각이 다르기 때문인데, 이 의견차를 좁히는 것이 어렵다. 또한 이해관계자들을 모두 만족시키는 요구사항을 찾는 것도 쉬운 일은 아니다. 따라서 각 이해관계자의 위치와 역할, 상황에 따라 특성을 파악하고 적절한 요구사항을 이끌어내는 것이 중요하다.

아래에 이해관계자의 종류와 각자 어떤 부분을 중요하게 생각하는지에 대해 정리해본다. 아래 표에 있는 것만이 이해관계자의 모든 종류가 아니며 프로젝트에 따라 아래보다 더 많은 유형의 이해관계자가 참여할 수도 있고, 더 적게 참여할 수도 있다.

구분	이해관계자	특징
내부 (수행사)	경영자	<ul style="list-style-type: none"> • 사측 자원(시간, 인건비 등)의 절감 노력 • 추후 사업의 지속 또는 연계 사업을 위해 고객과의 원만한 관계 유지 노력
	프로젝트 매니저(PM)	<ul style="list-style-type: none"> • 개발범위에 따른 프로젝트 난이도 조절 • 프로젝트 이해관계자들의 의견 조율 및 이슈 관리
	개발자	<ul style="list-style-type: none"> • 담당 기능 개발 수행 및 이슈 보고
외부	협력사	<ul style="list-style-type: none"> • 협의에 따른 담당 분야 수행
	공공기관(협력)	<ul style="list-style-type: none"> • 수행사 요청 협조
고객	법인 및 단체	<ul style="list-style-type: none"> • 요구사항 전달 • 프로젝트 완성도 피드백 • 지급 비용 절감 노력
	공공기관(고객)	<ul style="list-style-type: none"> • 요구사항 전달 • 규제나 민원 등에 민감
	고객사 소속 사용자	<ul style="list-style-type: none"> • 고객사가 수익이 아니라 자사의 직원들을 위해 시스템 개발을 의뢰하는 경우 • 요구사항 수집을 위해 인터뷰 필요
	일반 개인 사용자	<ul style="list-style-type: none"> • 고객사가 수익을 위해 개발을 의뢰한 프로그램을 사용하는 사람 • 개발하기 전에 요구사항 수집이 힘들

고객 중에서 개발 수행자에게 개발 비용을 지급하는 고객은 직접적으로 만날 수 있기 때문에 요구사항을 수집하는 것이 그나마 수월하다. 그러나 기업 고객이 자사의 직원들을 위한 프로그램, 예를 들면 사내 게시판과 같은 프로그램을 개발한다고 하면 고객사의 직원들을 대상으로 인터뷰를 하면서 사내 게시판에 원하는 기능들을 수집해야 할 수도 있다. 개발사가 직접 제공할 서비스라면 서비스를 사용할 사람들이 고객이 된다. 이때에는 개발 전에 요구사항을 완벽히 수집하는 것이 어려우므로 개발 완료 후에도 지속적으로 피드백을 받아 시스템에 반영을 해야 한다.²⁶⁾

26) 물론 이런 경우는 시장조사를 통해 충분히 요구사항을 수집하고 개발을 해야 한다.

이제 소프트웨어 요구사항의 종류에 대해서 알아보자. 주의할 것이 있는데 요구사항을 구분하는 방법이나 종류는 다양하며 이 모든 것을 전부 적용할 필요는 없다. 프로젝트의 규모에 맞춰서 적당한 방법을 찾는 것이 필요하다. 요구사항을 찾는 것은 시험 정답을 제출하는 것이 아니라 시스템을 잘 만들기 위해 고객과 개발자가 함께 시스템을 이해하는 과정이다.

요구사항	내용
기능	<ul style="list-style-type: none"> 사용자가 직접 사용할 수 있는 기능에 대한 내용 (예시 : 회원 가입을 하지 않고 사용할 수 있음)
시스템 기능	<ul style="list-style-type: none"> 시스템이 갖춰야 할 기능에 대한 내용 (예시 : 네트워크를 지원해야 함)
시스템 성능	<ul style="list-style-type: none"> 시스템 동시 접속, 전송 속도 등 시스템의 성능과 관련한 내용
사용자 인터페이스	<ul style="list-style-type: none"> 사용자 인터페이스에 대한 내용
시스템 인터페이스	<ul style="list-style-type: none"> 시스템 인터페이스(시스템 간의 연동이나 접속)에 대한 내용
테스트	<ul style="list-style-type: none"> 테스트 방법, 종류, 결과 등에 대한 내용
유지 보수	<ul style="list-style-type: none"> 시스템의 정상적인 유지를 위해 필요한 내용
운영 관리	<ul style="list-style-type: none"> 시스템의 운영을 위해 필요한 인력이나 문제 해결 방법에 대한 내용
시스템 운영	<ul style="list-style-type: none"> 시스템의 운영과 관리를 위해 필요한 내용
투입 인력	<ul style="list-style-type: none"> 개발 기간 동안 투입 인력 및 유지 보수 인력에 대한 내용
보안	<ul style="list-style-type: none"> 시스템(소프트웨어 및 하드웨어)의 보안을 위해 필요한 내용
품질 관리	<ul style="list-style-type: none"> 시스템의 목표 품질이나 품질 평가 방법 등에 대한 내용
제약 사항	<ul style="list-style-type: none"> 시스템과 관련한 기술, 표준, 특허, 법률, 규제 등에 대한 내용
사업 관리	<ul style="list-style-type: none"> 프로젝트 수행을 위해 사업 관리 측면에서 요구하는 내용 (예시 : 개발 기간 동안 3회의 중간 보고 등)
사업 지원	<ul style="list-style-type: none"> 프로젝트 수행을 위해 요구하는 내용 (예시 : 프로젝트 관리자는 공학박사 이상, 개발 기간 동안 파견 등)

대부분의 경우는 기능 요구사항이 가장 많을 것이다. 소프트웨어를 통해 할 수 있는 기능들에 대한 요구사항으로 작은 규모의 프로젝트는 기능 요구사항만으로도 정리가 되기도 한다. 우리가 함께 설계하던 엘리베이터의 예를 들면 ‘엘리베이터 내부의 숫자 버튼을 누르면 해당 층으로 이동한다.’, ‘문 열림 버튼을 누르고 있는 동안은 엘리베이터 문이 닫히지 않는다.’ 등을 기능 요구사항으로 볼 수 있겠다. 프로젝트의 규모가 커질수록 요구사항의 범위와 내용도 많아지게 되며 이러한 요구사항들을 요구사항 명세서에 정리해야 한다. 요구사항 명세서는 시스템의 요구사항들을 정리해놓은 문서로, 국제 표준 기관인 IEEE²⁷⁾에서는 소프트웨어 요구 명세(Software Requirements Specification, SRS)라는 이름으로 요구사항 명세서의 구조를 정리해 두었다.

27) 전기전자기술자협회(Institute of Electrical and Electronics Engineers, IEEE)는 미국에 위치한 전기전자공학에 대한 국제 조직이며 다양한 표준이나 연구 정책을 제안하고 유지한다.

그럼 이제 앞서 20층이라는 정보밖에 없었던 엘리베이터의 요구사항을 한번 분석해보자. 이해를 돕기 위해 만드는 간략한 요구사항이지만, 의도적으로 기존의 엘리베이터와는 다르게 만드는 것이니 참고하길 바란다.²⁸⁾

분류	요구사항	내용
기능	호출	<ul style="list-style-type: none"> 다른 층에 있는 승강기를 사용자가 있는 층으로 오게 하기 위해서 문 앞에 설치된 마이크에 “이리 오너라”라고 말해야 한다. 이미 다른 층으로 움직이고 있는 승강기를 호출하면, 승강기는 현재 위치에서 목적지와 호출지의 거리를 비교하여 가까운 곳으로 먼저 간다.
	이동	<ul style="list-style-type: none"> 다른 층으로 이동하기 위해 승강기 내부에 있는 마이크에 “xx층으로 가자”라고 말해야 한다. 1층~20층 이외의 층을 말하면 흥겨운 음악이 나와야 한다.
인터페이스	사용자 인터페이스	<ul style="list-style-type: none"> 승강기 호출과 이동을 위해 마이크를 사용한다.
성능	이동 속도	<ul style="list-style-type: none"> 승강기는 1개 층을 움직이는데 1분 이상이 걸려야 한다.
제약 사항	법률 준수	<ul style="list-style-type: none"> 시스템의 모든 내용은 승강기 안전 관리법을 준수해야 한다.

여러분은 고객에게 엘리베이터에 대한 정보를 더 얻기 위해 미팅을 요청했고, 고객은 위와 같은 요구사항을 요청했다. 어떨까? 일반적이진 않지만 일단 엘리베이터로써 기능은 할 것 같다. 백번 양보해서 여러분은 이 요구사항을 들어주기로 했고 요구사항의 모호함을 없애기 위해 고객에게 몇 가지를 더 질문을 한다.

“마이크의 인식 범위는 어디까지인가? 음성 인식이 계속 실패하면 어떻게 해야 하는가? 승강기가 움직이는 중간에 계속 다른 사람이 호출하여 아예 목적지에 도착하지 못해도 괜찮은가? 흥겨운 음악이란 어떤 음악인가?”

여러분은 위의 내용보다 질문할 것이 더 많을지도 모르겠다. 사실 그래야 한다. 위의 요구사항은 이해도 안 갈뿐더러 요구사항 유도과 분석을 통해 여러분은 아직 만들지도 않은 소프트웨어에 대해 거의 완벽하게 파악을 할 수 있어야 한다. 그리고 고객이 원하는 것과 우리가 이해한 것을 일치시켜서 개발을 완료하고 난 다음 그 둘의 차이가 거의 없도록 하는 것이 요구사항을 분석하는 이유이다. 예를 들어 위의 요구사항에서 흥겨운 음악에 대한 정의가 명확하지 않으면, 개발자가 임의로 넣은 음악에 대해 고객은 흥겨운 음악이 아니라고 할 수 있다. 따라서 해석하기에 따라 달라지는 모호한 부분들은 질문과 회의를 통해 최대한 없애야 한다.

28) 그리고 소프트웨어 요구 명세라는 표준이 있긴 하지만 반드시 따를 필요는 없다. 고객과 개발자가 서로 잘 이해할 수 있도록 간결하고 가독성이 좋으며 필요한 내용이 다 들어있기만 하면 무엇이든 괜찮다.

Note : 사용자 경험과 요구사항

앞서 본문에서 예시로 든 엘리베이터의 요구사항은 우리가 알고 있는 것과는 달리 대단히 이질적이다. 그럼 반대로 우리가 익히 알고 있는 엘리베이터의 요구사항을 한번 보자.

“이동을 위해서는 엘리베이터 내부에 있는 숫자를 누르면 해당 층으로 이동한다.”

위의 요구사항은 어쩌면 우리가 너무 뻔하다고 할 정도로 잘 알고 있는 내용이다. 만약 고객이 일반적인 엘리베이터의 설계를 요구한다고 한다면 이런 내용도 요구사항 명세서에 들어가야 할까? 또 다른 예로 일반적인 소프트웨어 개발에서 ‘저장’기능을 구현한다고 할 때, “저장 버튼은 디스켓 모양 아이콘으로 나타낸다.”라는 내용이 필요할까?

일단 이에 대한 명확한 정답은 없다. 우리는 각자의 경험을 통해 이러한 내용들을 어느 정도 알고 있다. 하지만 모든 사람들이 동일한 경험을 하고, 동일한 내용을 알고 있으리란 법은 없다. 따라서 이러한 부분은 서로가 알고 있는 부분이 얼마나 유사하냐에 따라 달라질 수 있는 내용이다. 예를 들어 유럽의 정말 오래된 호텔은 내부 엘리베이터가 나무로 되어있고 레버로 움직이기도 한다. 그리고 요즘 젊은 세대들은 플로피 디스켓을 본 적이 없어 저장 버튼으로 쓰이는 그림이 무엇인지, 왜 사용하는지 모른다고 한다. 따라서 고객과 개발자가 서로 알고 있는 영역이 비슷하다면 어느 정도 생략하고 갈 수 있지만, 만약 경험의 차이가 크다면 사소한 내용도 요구사항으로 나타내 주는 게 서로 도움이 된다.

사용자 경험을 의미하는 단어로 UX(User Experience)라는 단어가 있다. 사용자가 어떠한 제품이나 시스템을 사용하면서 얻게 되는 경험을 의미하는데 엘리베이터의 예를 들면 숫자 버튼을 누르면 해당 층으로 이동한다는 것이 UX에 해당한다. 만약 숫자 버튼을 누를 때마다 엘리베이터가 움직이지는 않고 음악이 나온다고 하면 일반적인 엘리베이터와 UX가 다르기 때문에 이 엘리베이터를 사용하는 사람은 혼란에 빠질 수 밖에 없다.

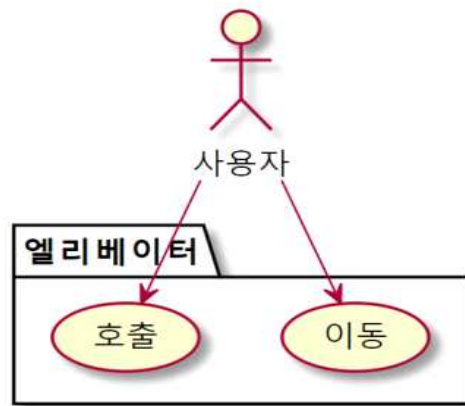
좋은 UX의 가치를 전부 헤아리기는 힘들지만, 사용자에게는 쉽게 제품에 적응하고 사용할 수 있도록 해주고, 다른 사용자와 공유할 수 있는 경험을 제공해 준다. 기업에게는 혁신을 위해 투자한 개발 기술력이 브랜드 가치를 상승시키고, 견고한 제품 사용성은 제품 상담과 운영 관리비를 절감시켜 준다. 따라서 사용자에게 최대한 친숙하면서도, 부분적으로는 혁신을 갖추는 UX를 개발하기 위해 지금도 수많은 기업에서는 UX 전담팀을 꾸려서 운영하고 있다.

UML 다이어그램 작성

UML은 Unified Modeling Language의 약자로 소프트웨어 개발을 위해 필요한 설계를 표현할 수 있는 모델링 언어이다. UML을 통해 설계를 하면 요구사항 명세서에서 문장으로 주고 받던 내용보다 간결하고 또 명확하게 나타낼 수 있어서 협업을 하는 개발자들과 의견을 공유할 수 있는 효과적인 수단이다.²⁹⁾ 건축 설계도에 평면도, 단면도, 조감도 등 다양한 설계도가 있는 것처럼 UML 다이어그램도 여러 종류가 있는데, 여기서는 다이어그램을 작성하는 방법까지 다루지는 않고 다이어그램의 종류를 살펴본 후 앞의 엘리베이터 예제에 대입해보겠다.³⁰⁾

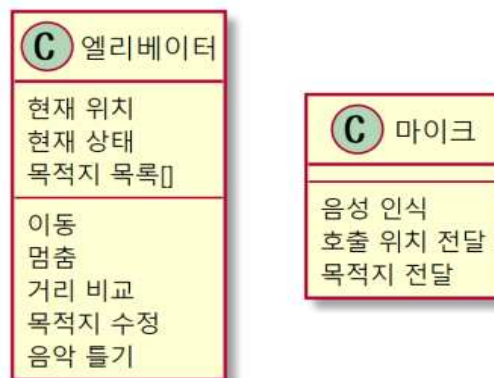
유스케이스(usecase) 다이어그램

사용자(actor)의 입장에서 행동(action)할 수 있는 기능들을 나타냄



클래스 다이어그램

시스템에서 클래스로 표현할 수 있는 속성과 행동들을 묶어서 표현함

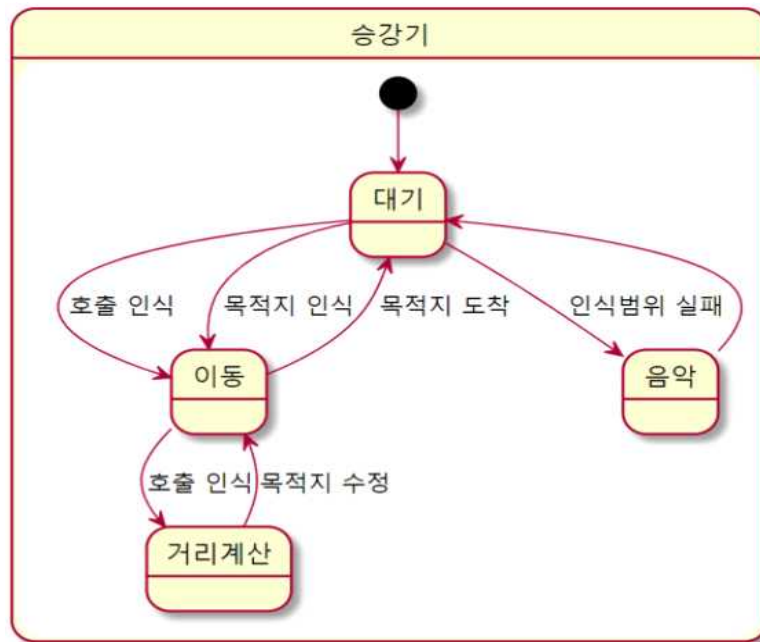


29) UML은 분명히 개발자들이 설계 내용을 공유할 수 있는 매우 정확하고 효과적인 수단이다. 그러나 최근에는 프로젝트를 수행하면서 너무 많은 문서 작업이 비효율적이라고 하여 종종 생략하기도 한다.

30) 본 책의 UML 다이어그램은 PlantUML이라는 오픈소스를 사용하여 그렸다. <https://plantuml.com/ko/>

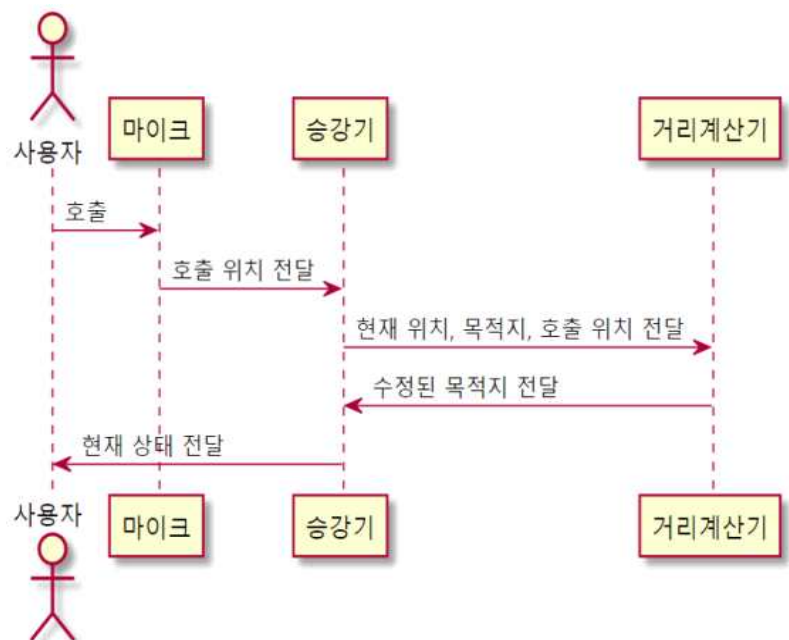
상태 다이어그램

시간의 흐름에 따라 객체가 가지는 상태의 변화를 나타냄



시퀀스(sequence) 다이어그램

객체들이 서로 주고 받는 메시지를 시간의 흐름에 따라 나타냄

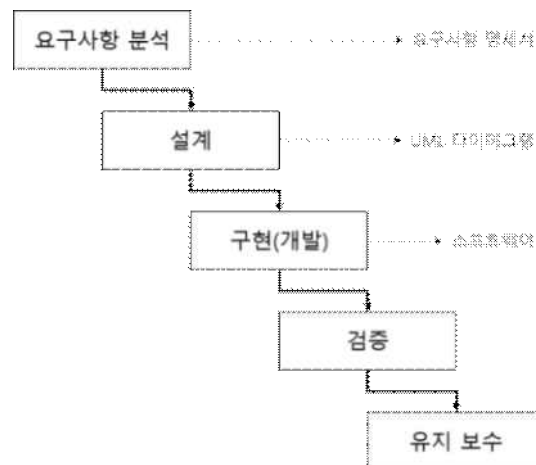


소프트웨어 개발 계획을 마쳤으면 이제 본격적으로 소프트웨어를 개발할 차례이다. 소프트웨어 개발 방법론은 소프트웨어를 개발할 때 필요한 일련의 과정들을 체계적으로 정리하여 개발자들이 프로세스를 따라가면서 프로젝트를 수행하고 함께 협업할 수 있는 방법을 제공한다. 본 장에서는 대표적인 개발방법론인 폭포수 모델과 스크럼 모델을 살펴본다.

소프트웨어 개발 계획을 마쳤으면 이제 본격적인 개발에 들어갈 차례다. 개발 계획에서 요구사항 분석과 소프트웨어의 설계도에 해당하는 UML 다이어그램 문서들을 다 만들었으면 개발자들은 이제 이를 보고 하나씩 역할을 나누어 구현만 하면 된다. 그리고 각자가 구현한 부분들을 가져와서 통합한 다음 테스트를 하여 소프트웨어를 완성하면 된다. 이론상 완벽하지 않은가? 비단 소프트웨어 개발만이 아니라 우리가 다른 사람과 협력하여 일을 할 때 대부분은 위와 같은 방법대로 순차적으로 일을 진행할 것이다.

폭포수 모델

소프트웨어 공학에서는 이와 같은 개발 방법론을 폭포수(Waterfall) 모델이라고 부른다. 마치 개발의 흐름이 폭포수가 흐르는 것처럼 진행된다고 폭포수 모델이다. 폭포수 모델은 앞서 우리가 본 것과 같이 요구사항 분석을 하고 이 요구사항으로부터 소프트웨어의 설계를 한다. 설계 방법은 다양하며 우리는 앞에서 UML 다이어그램을 통해 소프트웨어를 설계한 것을 보았다. 이후 설계도대로 실제 소프트웨어를 개발한다. 마지막으로 테스트를 통해 소프트웨어를 검증하여 개발을 완료한 다음 유지보수 단계로 넘어간다.



폭포수 모델의 구조

폭포수 모델의 특징은 물컵의 물이 다 차야지만 넘쳐서 그 아래로 흐르는 것처럼, 먼저 위의 작업이 완전히 끝나야 아래의 단계로 넘어간다는 것이다. 또한 물이 위로 다시 흐르지 않듯이 이전의 작업으로 돌아가지 않는다. 그러므로 각 단계의 작업에 돌입했을 때 해당 단계에서 진행할 수 있는 내용들은 전부 다 확정하고 다음 단계로 넘어가야 한다. 예를 들면 요구사항 분석을 완전히 다 끝내지 않은 상태에서는 설계를 할 수 없다.³¹⁾

폭포수 모델은 일반적으로 사람이 문제 해결을 위해 행하는 단계를 소프트웨어에 적용하였기 때문에 이해가 쉽다. 또 작업이 시간의 순서에 따라 진행되므로 스케줄 관리에 용이하고 작업의 결과물들을 단계별로 관리하기 편하다. 폭포수 모델은 소프트웨어 역사에서 비교적 초기에 등장한 아주 전통적인 개발 방법론으로, 만약 프로젝트 규모가 그리 크지 않다면 지금도 사용하기 좋은 개발 방법론이다. 특히 요구사항이 명확할 때 폭포수 모델은 큰 효율성을 보장한다. 그러나 폭포수 모델은 단점도 많은 개발 방법론이다. 이유는,

1. 현 단계의 상태를 완전히 파악할 수 있는 사람이 적음
 - 예를 들어 프로젝트 매니저(PM)가 고객과 오랜 시간에 걸쳐 요구사항 분석을 하였다. 그러나 PM의 능력이 아주 뛰어나지 않은 이상 놓친 요구사항이 있을 수 있다. 그러나 이를 인지하지 못하고 다음 단계로 넘어가서 개발을 하다가 다시 요구사항 단계로 돌아와야 할 수도 있다.
2. 고객은 반드시 요구사항 분석 단계가 끝난 다음에도 추가 요구사항을 전달함
 - 고객은 전문가가 아니기 때문에 개발이 진행되며 점차 결과물의 모습을 실제로 보면서 자신이 무엇을 원하는지 확실히 알게 된다. 예를 들어 처음엔 빨간색을 요구했는데 막상 보니 마음에 안 들어서 파란색으로 바꿔 달라고 할 수도 있다.
3. 폭포수 모델은 각 단계마다 많은 문서를 요구함
 - 폭포수 모델은 각 단계마다 수없이 많은 문서를 요구한다. 요구사항 명세서, 설계서, 중간 보고서, 결과 보고서, 테스트 보고서, 매뉴얼 등 각 단계별로 문서작업까지 끝나야 완료로 본다. 이러한 문서 작업들이 실제 소프트웨어 개발에 지장을 많이 준다.
4. 약간의 변화에 큰 영향을 받음
 - 만약 지금 폭포수 모델의 검증단계에 와있는 상황에서 갑자기 고객이 추가 요구사항을 전달했다. 비록 작은 요구사항이지만 이로 인해 설계, 개발, 검증 단계까지 모든 단계가 다 영향을 받고, 각 단계의 결과물들도 모두 수정이 되어야 한다.
5. 유지 보수의 어려움
 - 약간의 변화에 큰 영향을 받기 때문에 개발이 완료된 다음 유지 보수를 위해 업데이트를 하거나 추가 기능을 구현하게 되면 마찬가지로 모든 단계를 거치면서 작업을 해야 한다. 만약 소스코드에 스파게티 코드라도 있으면 엄청난 규모의 작업으로 변질 수 있다.

보다시피 폭포수 모델의 큰 단점은 변경점으로 인해 고스란히 했던 일을 다시 하면서 개발자들의 업무량이 늘어나는 것이다. 물론 무작정 나쁘기만 한 개발 방법론은 아니지만 최근에는 대체할 수 있는 개발 방법론이 많이 있으니 개발 단계에서 방법론을 선택할 때 프로젝트의 수행 환경을 잘 고려해야 한다.

31) 실제로 요구사항 분석을 마치지 않고 설계를 할 수 없는 건 아니지만, 요구사항이 바뀌면 설계를 다시 해야 한다.

스크럼 모델

소프트웨어 공학에서 폭포수 모델과 종종 자주 비교되는 모델이 애자일(Agile) 개발 방법론이다. 애자일의 영어 단어는 민첩한, 날렵한이란 뜻을 가진 형용사이며 이름이 가진 것처럼 빠른 개발을 위해 사용하는 모델이다. 이 애자일 개발 방법론 중에서도 대표적인 것이 스크럼(Scrum) 모델과 칸반(Kanban) 모델이다. 우리는 여기서 스크럼 모델에 대해서 알아본다.

스크럼 모델은 폭포수 모델과 비교하면 조금 복잡한데, 스프린트라는 개발 주기를 정해두고 이 스프린트를 반복해가며 개발을 진행한다. 먼저 용어에 대한 설명을 하고 그 다음에 스크럼 모델의 진행 방법을 설명한다. 스크럼 모델의 개발 프로세스 그림을 참고하여 설명을 보길 바란다.

스크럼 모델 핵심 용어

백로그 :

소프트웨어 개발을 위해 필요한 요구사항 목록

스프린트 :

스크럼 모델에서 개발을 진행하는 최소 주기. 한 번의 스프린트에 1개의 목표를 달성해야 하며 스프린트의 결과물은 항상 실행 가능한 제품의 형태로 나와야 한다.

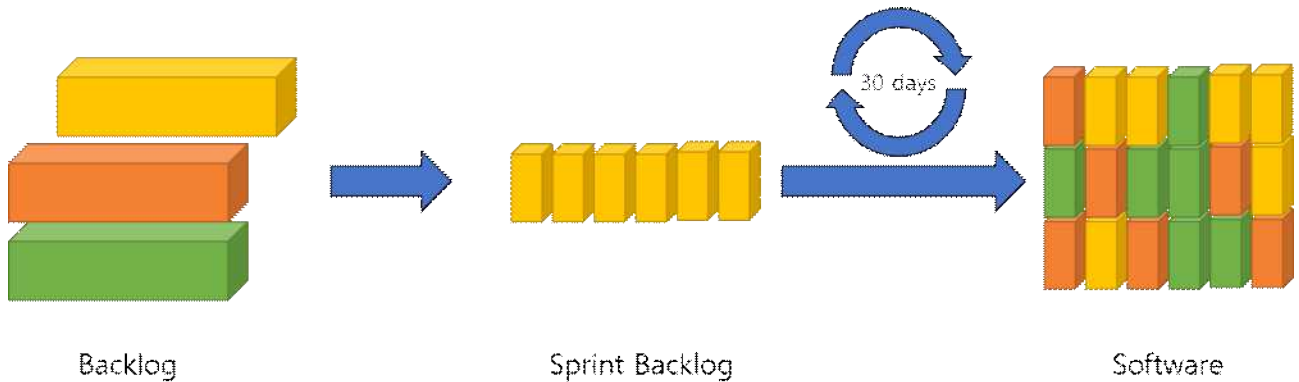
스프린트 백로그 :

스프린트 목표를 달성하기 위한 요구사항 목록

스크럼 모델 진행 방법

1. 개발할 목표 소프트웨어에 대해 팀 내에서 충분한 회의를 거친다.
2. 모든 팀원들이 함께 목표 소프트웨어에 포함할 요구사항들을 백로그에 전부 기록한다.
3. 백로그에 충분히 요구사항을 수집했다면, 이제 요구사항들의 우선순위를 정한다.
4. 스프린트의 목표를 정하고, 이 목표에 맞는 요구사항들을 백로그에서 추려내서 스프린트 백로그에 기록한다.
5. 스프린트를 시작한다. 스프린트는 30일 단위이며, 이 기간 동안 스프린트 백로그에 있는 요구사항들을 다 구현해야 한다.³²⁾
6. 한 스프린트가 끝나면 스프린트 개발 결과물을 정리하고 스프린트 리뷰를 한다.
7. 한 스프린트를 진행하는 중간에 다음 스프린트 진행을 위한 준비를 하고 있어야 한다.
8. 스프린트를 반복하면서 조금씩 백로그에 있는 요구사항들을 하나하나 구현하며 소프트웨어를 완성한다.

32) 스프린트의 기간은 상황에 따라 조정하며 실행 가능한 결과물을 만들 수 있을 정도의 기간을 둔다.



스크럼 모델의 개발 프로세스

스크럼 모델은 매 스프린트마다 실행 가능한 결과물들을 생산하다 보니 개발자들 스스로가 눈에 보이는 결과물을 통해 개선점과 추가 기능들을 제안하게 된다. 따라서 스프린트 리뷰를 통해 추가할 기능들을 또 백로그에 기록하고 다음 스프린트에 기존의 백로그와 개선점을 더해 가며 소프트웨어를 완성해 나간다. 그러므로 스프린트의 결과물은 반드시 실행이 가능해야 하며, 개발이 덜 끝나서 실행이 안 되는 상태라면 스프린트를 종료하지 못한다. 그렇기 때문에 스크럼 모델은 팀의 협력과 커뮤니케이션이 대단히 중요한 개발 모델이다.³³⁾

스크럼 모델을 진행하며 반드시 지켜야 할 중요한 내용이 있다. 기본적으로 팀원들 서로가 서로에 대한 존중이 뒷받침되어야 한다. 팀장의 강압이나 명령보다는 팀의 자율성을 통해 아이디어를 마음껏 제안하고 거기에서 개선점을 찾는 창의력을 발산할 수 있어야 한다. 또한 팀은 비슷한 구성원들보다 다양한 성격의 사람들로 구성되어 여러 환경에 기민하게 대응할 수 있는 것을 권장한다. 팀원 간의 커뮤니케이션에서 갈등과 갈등 해결을 통한 긴장감 조성은 건강한 팀 문화에 도움이 된다. 이러한 팀 커뮤니케이션을 통해 팀원들이 동일한 목표 의식을 가지고 팀 단위로 유기적으로 문제를 해결하는 것이 스크럼 모델에서 지향하는 바다.

폭포수 모델과 비교하면 우선 개발을 수행하는 조직의 구성부터 차이가 있다. 폭포수 모델은 관리자급 직원에 의해 요구사항 분석과 설계가 이루어지고, 이후 개발자들은 이 설계대로 구현과 테스트를 수행한다. 개발 방법과 조직 구성이 수직적인 형태를 지닌다. 그런데 스크럼(애자일) 모델은 팀 단위로 조직되며 팀원 간의 수평적인 구조를 요구한다. 이는 개발 뿐만 아니라 기업의 조직 문화와도 관계가 있는 것으로, 기업이 수직적인 조직 문화와 수동적 참여 태도를 벗어나지 못하면 스크럼 모델을 적용하기가 쉽지 않을 것이다.

위와 같은 이유로 애자일 개발 방법론을 사용하는 기업에서는 각 팀을 관리하는 팀장 또는 중간 관리자의 역할이 매우 중요해졌다. 각 팀은 의사소통이 원활히 가능한 5명 내지 10명 정도의 인원수로 구성되는 것이 보통이며 기업마다 상황에 따라 조금씩 다를 수 있다. 팀장은 팀원들의 커뮤니케이션을 활성화하기 위해 부단히 노력해야 하며 팀원 개개인의 특성을 잘 파악

33) 엄밀히 말하면 팀의 협력과 커뮤니케이션이 중요하지 않은 모델이 어디 있겠느냐만, 스크럼을 포함하여 애자일 개발 방법론들은 팀원의 자발적인 협력 없이는 정상적인 수행조차 힘들다.

하고 있어야 한다. 만약 팀장이 팀원의 역량을 제대로 파악하지 못한다면 스프린트의 계획을 제대로 세울 수 없다. 팀원을 과소평가했다면 스프린트가 너무 일찍 끝날 것이며 반대로 과대평가했다면 기간 안에 스프린트를 종료하지 못하고 야근을 하게 될 것이다. 또한 사회적 분위기도 예전과 다르게 상명하복식의 업무 처리가 아니라 수평적인 기업 문화를 선호하니, 개인의 특성을 존중하되 그 개인의 역량을 끌어내는 역할을 팀장들이 맡게 되었다. 그러므로 팀장들은 개인의 개발 업무 능력 뿐만 아니라 다른 사람과의 커뮤니케이션 능력도 매우 중요하며 팀의 스케줄 관리나 업무 조율 등 다양한 능력을 요구하게 되었다. 특히 개발 능력은 출중해도 커뮤니케이션 능력이 부족한 경우가 많으므로 이 책을 읽는 여러분이 아직 학교를 다니는 중이라면 다양한 조직 생활과 팀 프로젝트를 수행하여 다양한 사람을 대하는 법을 익히길 바란다.

Note : 학교에서의 팀 프로젝트

학교를 다니다 보면 친구들과 함께, 또는 모르는 사람들과 함께 팀 프로젝트를 수행하는 경우가 있다. 대다수의 경우 모르는 사람들과 함께하는 팀 프로젝트를 좋아하지 않는데, 그 이유가 남에게 피해를 주거나, 피해를 받거나, 또는 귀찮다는 이유 등이다. 이는 각 팀원 간의 특성이 다르기 때문인데, 서로 실력도 다르고 의견도 다르며 이 프로젝트를 대하는 자세 또한 다 다르기 때문이다. 만약 팀원 간의 불화로 인해 갈등이 심해지면 그때부터 팀 프로젝트의 의미가 사라지게 된다.

이러한 문제는 해결 방법에 정답이 없다. 정답은 아니지만 가장 간단한 방법이 있는데 무엇인지 아는가? 한 명이 팀장을 맡고, 그 팀장이 시키는 것에 모든 팀원이 불복 없이 따르는 것이다. 가장 간단하고 굳건한 지휘 체계가 필요한 군대가 사용하는 방식이다. 이에 모든 팀원들이 수용한다면 어찌 보면 그것이 가장 나은 방법이 될 수 있다. 하지만 여러분의 팀장이 항상 올바른 결정을 내릴 수 있다고 생각하는가? 그렇지 않기 때문에 팀원들은 팀장의 명령에 제동을 걸게 되고 대부분 이런 식의 팀 운영을 원하지 않을 것이다.

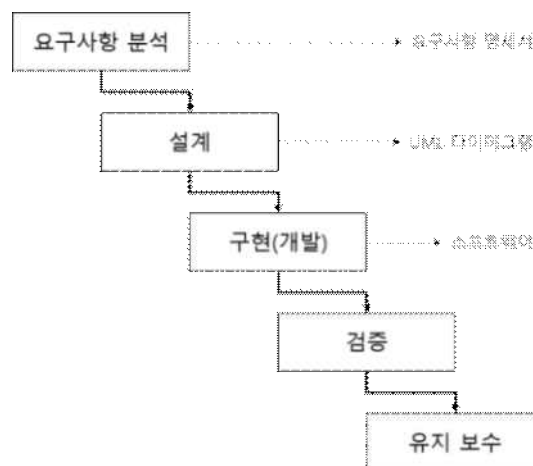
여러분은 팀 프로젝트를 통해 3~5명으로 이루어진 팀에서 함께 소통하는 방법을 배워야 한다. 소통이란 건 '같은 언어로 말을 하고 듣는 것'만 의미하는 것이 절대 아니다. 여러분이 반드시 팀장일 필요는 없지만 팀원으로서도 함께 팀이 처한 문제를 해결 해나가야 한다. 누군가 실력이 부족하면 이끌어줘야 하고, 시간이 부족하면 무언가를 포기할 줄도 알아야 한다. 갈등을 해소하는 것도 대화의 한 축이며, 여러분의 따뜻한 인사 한마디가 흩어지던 의견을 한데 모으는 강력한 힘을 가질 수도 있다는 걸 알아야 한다. 만약 팀 운영 상황이 좋지 않아서 '이번엔 팀 운이 없다.' 라고 생각할 수도 있겠지만 우리가 친구를 만나든 연애를 하든 동료와 일을 하든 항상 사람 관계는 어렵고 정답이 없다. 마음이 잘 맞는 이와 소통을 당연시하지 말고 감사히 여기며 그렇지 않은 이와 소통도 조금씩 노력해보자.

기업에서 원하는 커뮤니케이션 능력은 한순간에 생기는 것이 아니다. 게다가 이론 공부처럼 혼자서 책을 펴고 할 수 있는 것 또한 아니다. 이러한 소통 능력은 팀 프로젝트 뿐만 아니라 친구들과의 활동, 동호회, 봉사 활동, 가족과의 대화 등 다양한 사람 관계를 통해 조금씩 축적되는 인생의 습관이니 항상 주변 사람들과 좋은 관계를 맺고 유지하며 지낼 수 있기를 바란다.

소프트웨어 관리는 소프트웨어 개발 못지않게 중요한 부분이지만, 초보 개발자들이 개발보다 중요성을 낮게 보고 등한시하는 경우가 많다. 그러나 이제는 소프트웨어의 제공 방식이 패키지 중심에서 서비스 중심으로 바뀌면서 소프트웨어의 관리는 개발만큼 중요해졌다. 이번 장에서는 소프트웨어 관리와 관리 방법의 중심에 선 데브옵스에 대해서 알아본다.

소프트웨어 관리는 소프트웨어 공학만큼 넓고 많은 의미를 포함하고 있다. 소프트웨어를 둘러싼 여러 당사자³⁴⁾ 즉 개발자, 운영자, 판매자, 소비자 등 각자가 소프트웨어와 관계된 입장에 따라 소프트웨어 관리 방식도 달라질 것이며 어떤 소프트웨어인지, 사용 목적, 대상, 규모 등 다양한 요건에 따라서도 달라질 것이다. 따라서 본 장에서는 소프트웨어를 제공하는 입장(개발자나 운영자, 또는 그 회사)에서 소프트웨어의 생명주기와 관계된 소프트웨어 관리가 무엇인지 알아보려고 한다.

먼저 생명주기(Life Cycle)라는 것은 무엇인가가 만들어지고 나서 사라지기까지의 과정을 말하는 것이다. 그 과정 속에는 여러 가지 사건들이 있을 텐데 커다란 변화가 있을 때마다 우리는 단계를 구분하여 생명주기를 일반화한다. 소프트웨어 생명주기는 우리가 앞서 봤던 폭포수 모델이나 스크럼 모델의 그것과 같다. 아래 폭포수 모델의 그림을 다시 보자. 우리는 앞에서 요구사항분석부터 구현(개발)까지 집중해서 보았다. 그리고 아직 학교에서는 대부분 이정도까지 배우는 것이 대다수이다.



폭포수 모델의 구조

34) 앞서 요구사항 분석에서도 본 것처럼 하나의 현상에 입장이 다른 여러 참가자들을 보통 이해 관계자, 이해 당사자라고 하며 영어로는 stakeholder라고 한다. 경우에 따라 개발자들도 많이 사용하는 용어이다.

예를 들어 프로그래밍 수업에서 실습 과제가 나왔다고 하자. 먼저 과제의 내용을 이해하는 것이 요구사항 분석으로 볼 수 있겠다. 여기서 과제의 내용을 제대로 이해 못 하면 제대로 된 과제를 만들 수 있을 리가 없다. 두 번째로 설계이다. 과제의 규모에 따라서 설계를 생략하고 바로 개발을 하는 경우도 많다. 만약 조별 과제 정도의 규모라면 분명히 여러분은 역할을 나누어 설계를 한 다음 개발을 시작할 것이다. 그리고 개발과 검증은 보통 같이 진행한 후 과제를 제출하고 마감한다.

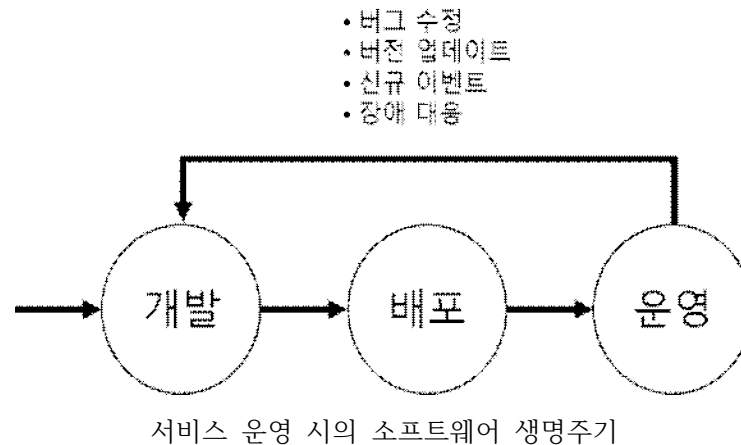
경우에 따라 다르겠지만 정말 많은 경우에 학교에서 실습을 통해 경험하고 있는 소프트웨어 생명주기에는 개발이나 구현 쪽에 조금 더 많은 비중이 쏠리고 요구사항 분석이나 설계, 그리고 검증과 유지보수는 비중이 떨어지거나 아예 꺾어보지 못하는 경우도 많다. 그러다 보니 많은 학생들이 유지보수 단계를 간과하는 경우가 많은데 실제 우리가 개발하고 서비스하는 소프트웨어는 개발도 중요하지만 견고한 소프트웨어의 개발을 위해서는 생명주기의 모든 단계가 전부 중요하다. 특히 현대의 소프트웨어 생태계에서는 유지보수의 단계가 매우 중요해졌는데 그 이유는 소프트웨어의 배포 방식이 패키지에서 서비스로 변화하였기 때문이다.

최근 대부분의 소프트웨어들은 이제 패키지 배포가 아닌 서비스 배포로 그 이데올로기가 바뀌었다. 패키지와 서비스는 게임으로 비유하자면 싱글게임과 온라인 멀티게임으로 비유할 수 있다. 싱글게임은 패키지를 판매할 때 회사에게 수익이 발생한다. 따라서 안 좋게 말하면 게임을 판매하고 난 후에는 회사가 해당 게임의 버그를 고친다거나 추가 기능을 만들어서 업데이트하는 것은 수익과 관계없는 추가 업무이므로 회사 입장에서는 손해이다.³⁵⁾ 그러나 온라인 멀티게임은 배포 후부터가 회사가 수익을 발생시킬 수 있는 구조이다. 각 회사마다 정책은 다르겠지만 월 단위의 구독 형태도 있을 수 있고 게임 내 재화를 판매할 수도 있다. 따라서 온라인 멀티게임은 사용자가 끊임없이 게임을 즐길 수 있게 해줘야지 회사가 지속적인 수익을 얻을 수 있다. 따라서 게임에 버그가 발생하면 사용자가 줄어들기 전에 빨리 수정을 해야 한다.

패키지 배포와 서비스의 차이는 바로 여기서 발생한다. 소프트웨어 생명주기의 유지보수에 해당하는 부분에서 둘 다 개발 이후 버그나 장애가 발생하면 문제를 고치는 것은 똑같지만, 서비스에서는 ‘운영’이라는 개념이 들어간다. 이 운영에는 단순히 버그를 고치고 사용자에게 정보만 제공하는 것뿐만 아니라, 사용자가 늘어나면 서비스가 중단되지 않게 로드 관리도 해야 하고 새로운 버전이 나오면 시스템이 돌아가고 있는 상황에서 라이브 업데이트를 해야 하는 경우도 있다. 게임 소프트웨어의 문제가 아니라 서비스를 제공하는 네트워크나 서버의 장애와 같은 문제도 관리해야 한다. 어찌 보면 소프트웨어 생명주기의 개발 단계보다 유지보수 단계부터의 개발이 훨씬 더 어렵다. 왜냐하면 사용자 수 폭등으로 갑자기 늘어난 부하를 처리하는 경험은 직접 그 정도 규모의 서비스를 운영해보기 전까지는 알기 어렵고, 그동안 운영을 하며 쌓인 사용자 데이터들을 유지한 채로 작업하는 것이 리스크가 더 크기 때문이다.

따라서 소프트웨어 생명주기는 개발 완료 후 테스트, 그리고 배포로 끝나는 것이 아니라 이제는 서비스가 종료되기 전까지 끝나지 않고 반복되는 개발 프로세스를 가지게 되었다.

35) 패키지와 서비스의 비교를 단순화하기 위한 것으로 사후 관리에 따른 브랜드 이미지 제고나 마케팅과 같은 거시적인 부분은 생략한다.



위의 그림과 같이 소프트웨어를 개발하고 배포한 후에는 운영을 하며 새로운 개발을 하기 시작한다. 이러한 개발에는 기존의 버그 수정도 있을 것이고 새로운 기능이 추가되면 버전 업데이트도 한다. 배달의민족이나 그 외 서비스 같은 경우는 할인 이벤트 같은 것이 매주 바뀌는 것을 볼 수 있을 것이다. 카카오톡과 같은 메신저 서비스는 연말연시와 같이 특정 시간대에 사용자가 몰려도 장애가 발생하지 않도록 부하 분산 작업을 할 것이다. 이렇게 소프트웨어의 생명주기 자체가 길어지게 되면서 소프트웨어의 관리도 자연스럽게 중요하게 되었고 이를 위한 여러 가지 방법들도 생기게 되었다.

Note : 다른 의미에서의 소프트웨어 관리

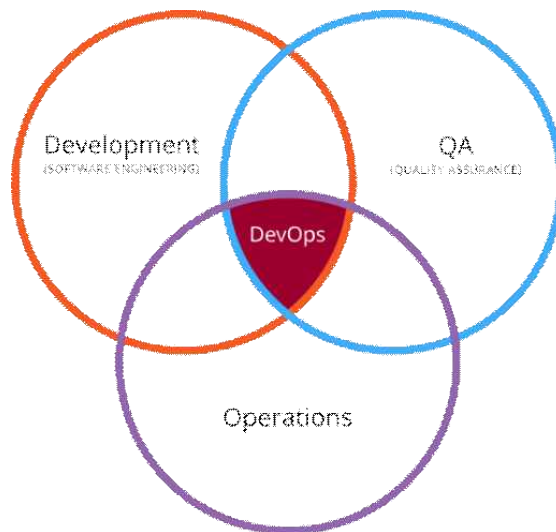
소프트웨어 관리는 그 범위가 넓은 만큼 사람들마다 소프트웨어 관리를 다른 의미로 이해할 수도 있다. 본 장에서는 소프트웨어 생명주기에 집중하여 본 것이며 좀 더 내용이 깊은 소프트웨어 공학 관련 도서를 보면 테스트나 품질보증(Quality Assurance, QA), 형상관리, 버전관리, 모니터링 등 소프트웨어 관리에 관한 다양한 내용이 있다. 이 중 테스트와 품질보증, 그리고 형상관리와 버전관리는 비슷하면서 조금씩 다르다. 특히 우리나라는 테스트와 품질보증을 구분하지 않는 경우가 흔한데 실제로는 품질보증팀은 매우 전문적인 능력을 요구하는 분야이다.

테스트는 소프트웨어의 기능이 정상적으로 동작하고 예상치 못한 버그 등을 검증하는 작업을 수행한다. 기능이나 버그를 잡으면 품질이 올라가니 품질보증도 비슷한 영역이라고 생각할 수 있지만 품질보증은 좀 더 넓은 범위에서 생각해야 하는 분야이다. 예를 들어 휴대폰을 만드는 회사에서 휴대폰 배터리의 용량이 타사의 제품에 비해 작다면 품질보증팀은 휴대폰에서 에너지를 절약할 수 있는 기본 앱이 없는지를 찾고, 또 충전기의 충전 속도를 빠르게 할 수 있는 방법 등을 찾아서 개발팀에 전달한다. 이러한 부분이 없다고 하더라도 제품은 정상적으로 동작하지만, 만약 실제로 특정 앱의 소스코드를 수정하여 에너지를 절약하는 방법을 찾았다고 하면 이 휴대폰의 품질은 올라갈 것이다. 이렇듯 소프트웨어를 관리하는 방법은 다양하게 존재한다.

데브옵스(DevOps)

데브옵스라는 개념 자체는 생긴 지 어느 정도 시간이 흘렀으나 가면 갈수록 그 중요성이 부각되고 있다. 데브옵스는 개발(Development)의 Dev와 운영(Operations)의 Ops가 합쳐진 용어로 개발자와 운영자 간의 소통과 협업을 강조하는 조직 문화를 의미한다. 다시 한번 의미를 반복하자면 데브옵스는 하나의 문화이다. 문화라는 것을 강조하는 이유는 아직 많은 사람들이 데브옵스에 대해 잘 모르는 경우가 많고 또 제대로 이해하지 못하면 데브옵스라는 용어의 오남용으로 인해 오해가 커지기 때문이다.

앞서 말한 대로 최근의 소프트웨어 이데올로기가 패키지에서 서비스로 변화하면서 소프트웨어는 개발하고 배포하는 것으로 끝나는 것이 아니라 운영을 함께 해야 한다는 것을 보았다. 그러면서 자연스럽게 개발팀과 운영팀의 업무 범위가 겹치는 것이 많아졌고 이러한 현상은 새로운 조직 문화를 요구하게 되었는데 그것이 바로 데브옵스이다.³⁶⁾



개발, 운영, 품질보증의 세 조직으로 구성된 데브옵스 벤다이어그램.
반드시 개발과 운영만 교집합을 이루는 것은 아니다.

그러나 말로 설명한 것처럼 데브옵스는 결코 단순하거나 쉬운 것이 아니다. 초기에 데브옵스 개념이 조금씩 알려지기 시작했을 때부터 일찌감치 데브옵스 도입을 시도한 소프트웨어 회사들은 실패를 많이 경험했는데 그 이유가 바로 ‘조직’과 ‘문화’의 차이를 이해하지 못한 데에서 기인한다.

초기의 소프트웨어 기업들은 개발과 운영의 융화를 위해 이 두 ‘조직’을 합쳐버리는 선택을 한다. 즉, 개발팀과 운영팀을 한 팀으로 묶고 개발과 운영을 이 팀에서 책임을 지게 하는 것이었다. 아직 회사 생활을 해보지 않은 독자라면 이것이 무슨 문제인지 잘 이해가 안 될 수

36) 개발팀의 역할은 ‘소프트웨어를 만드는 것’이고 운영팀의 역할은 ‘소프트웨어가 잘 동작하게 하는 것’이다.

도 있다. ‘서로 잘 대화하여 역할을 정하고 원래 자기가 맡았던 일을 잘 하면 되는 것이 아닌가?’ 하고 생각할 수도 있다. 회사 조직이 수평적 구조를 가지고 있는 경우면 그나마 낫지만 대부분의 경우 회사 내에서 기존의 개발팀이나 운영팀의 균형이 다를 수 있다. 운영팀의 팀장이 개발팀의 팀장보다 서열이 높거나 하는 경우인데, 합쳐진 조직에서의 정치싸움이나 책임회피로 인해 실제로는 개발자가 운영을 해야 하는 경우도 생기고 반대로 운영자가 개발을 공부해야 하는 경우도 생겼다. 그리고 같은 팀이라고 해도 팀원과의 커뮤니케이션이 쉬운 것도 아니다. 그러다 보니 안타깝게도 많은 경우에 각 구성원들이 서로 책임을 회피하면서 개발도 운영도 제대로 이루어지지 않을뿐더러 내부 불화를 키우는 경우가 많았다.

초기의 데브옵스 도입 실패에서 교훈을 얻은 소프트웨어 기업들은 차근차근 기업 내 조직들의 융화에 대해 고민하기 시작했고 업무를 위해 소통하는 방식에 대해 집중하기 시작했다. 이 같은 노력은 조직의 구성 방식이나 업무의 진행 방식 뿐만 아니라 기업의 체제나 인사 등에도 영향을 주며 각 회사마다 고유의 문화로 자리 잡게 되었다. 그리고 이를 잘 이룩한 회사들은 이제 역으로 이런 기업 문화를 바탕으로 하여 두 조직 간의 부드러운 커뮤니티 환경을 제공하였다. 이런 분위기에서 개발팀과 운영팀이 서로 커뮤니케이션을 하며 개발과 운영을 상호 협력해야지 데브옵스의 성공적 도입이라고 볼 수 있다.

앞서 데브옵스 도입에 실패했던 경우와 성공한 경우의 차이점은 데브옵스를 위한 조직을 억지로 구성하려 한 것과 조직 문화 자체를 바꾸어 데브옵스를 할 수 있는 환경을 만들어준 것이다. 사실 기업 문화를 바꾼다는 것은 절대 쉬운 일이 아니며 하루 아침에 되는 것도 아니다. 지금 건설한 위상을 뽐내고 있는 IT 기업들은 그만큼 시간과 노력을 들여서 조직의 문화를 개선해왔고 그 결실을 맺는다고 볼 수 있다. 기업 또한 이 과정에서 건강한 기업 문화를 위해서는 기업 구성원들의 학력이나 학점만이 중요한 것이 아니라 커뮤니케이션 능력이나 문제 해결 능력 등의 자질 또한 중요한 것을 인지하고 인사에 많이 반영을 하는 중이다.

데브옵스의 목적

데브옵스가 무엇인지는 알았는데 그럼 데브옵스가 필요한 이유는 무엇일까? 기업 입장에서 그저 개발팀과 운영팀이 서로 소통을 잘하며 열심히 일을 잘 해주기만을 바라는 것일까? 그렇지 않은 것이다. 사실 데브옵스 구성은 조직 문화가 가장 중요하고 기본이지만 자동화 도구와 시스템도 갖춰져야지 완성된다.³⁷⁾ 데브옵스를 위한 자동화 도구들은 운영팀의 요구를 빠르게 수용하여 개발에 반영할 수 있도록 해주고 배포를 자동으로 도와줘서 전체 개발과 운영의 속도를 가속화 해준다. 즉, 데브옵스를 도입하는 목적 자체는 소프트웨어의 개발과 관리를 빠르고 효율적으로 하기 위해서이다. 뒤에서는 어떤 방식과 자동화 도구를 사용하여 소프트웨어의 개발을 돕고 효율적으로 관리하는지 보도록 하겠다.

37) 안타깝게도 이런 데브옵스를 위한 자동화 도구만을 갖추고 자기들은 데브옵스를 적용했다는 기업들도 더러 있다. 앞서 본 것처럼 데브옵스를 위해서는 먼저 소통을 위한 기업 문화가 받쳐주어야 하고 그 이후 업무 체계와 시스템, 조직, 마지막으로 자동화 도구들이 갖춰져야지 제대로 된 데브옵스 문화가 도입되었다고 볼 수 있다.

빌드 자동화, 그리고 지속적 통합과 배포 (CI/CD)

개발자가 소스코드를 작성한 후 완성된 소프트웨어가 나오기까지 과정을 알아보자. 먼저 컴파일(compile)이라는 과정을 거친다. 그 후 빌드(build)를 하고 빌드를 성공하면 소프트웨어를 배포(deploy)하여 사용자들이 사용할 수 있게 한다. 간단하게 아래에 소프트웨어 개발을 번역서 출간 과정에 빗대어서 표현하였다.

소프트웨어 개발 과정	번역서 출간 과정
1. 소스코드 작성	1. 영어로 내용 작성
2. 컴파일	2. 번역가가 한국어로 번역
3. 빌드	3. 번역한 내용을 책으로 편집
4. 배포	4. 서점에 진열
5. 고객이 실행	5. 독자가 읽음

소스코드는 일반적으로 사람이 이해할 수 있는 언어로 작성되기 때문에 컴파일 과정을 거쳐 컴퓨터가 이해할 수 있는 내용으로 번역을 한다. 그리고 이 컴파일 된 내용을 빌드하여 실행 가능한 프로그램 형태로 만든다.³⁸⁾ 이후 완성된 소프트웨어를 배포하여 사용자가 실행할 수 있도록 하면 소프트웨어의 작성부터 배포까지 끝이 난다.

자, 그럼 빌드 자동화는 무엇일까? 위에서 보면 빌드는 컴파일된 내용을 실행 가능한 프로그램을 만드는 것이라고 한다. 그런데 나는 이미 개발도구를 사용해서 소스코드를 작성한 다음 빌드 버튼을 눌러서 프로그램으로 만들고 있었다. 여기서 더 이상 무엇을 자동화 한다는 것일까? 그리고 지속적 통합과 배포는 대체 무슨 말인가?

이해를 돕기 위해 한가지 예를 들어보겠다. 만약 여러분이 번역서를 출간하여 독자들에게 판매를 하였다. 그러다가 어떤 독자가 잘못된 번역에 대해서 제보를 해왔다. 이 실수는 너무나 치명적이라서 원작자의 의도가 완전히 달라졌고, 이대로 두었다가는 원작자의 명예가 훼손될 것 같다. 그러면 여러분은 어떻게 해야하겠는가?

먼저 서점의 번역서는 판매를 중지하고, 이미 판매된 도서에 대해서는 회수 조치를 취해야 할 것이다. 그리고 잘못된 번역을 고치고 새로 편집하여 새로운 판본을 만들어야 한다. 이미 구매했던 독자와 서점에는 새 판본을 전달하는 것으로 이 아찔했던 문제를 해결하고 마무리 짓는다. 다행히 문제는 해결했지만 그 과정이 대단히 귀찮고 번거롭지 않은가?

38) 단순히 컴파일과 빌드를 합쳐서 빌드라고 하기도 한다. 즉, 빌드는 실행 가능한 소프트웨어의 형태를 만드는 것을 의미한다.

그럼 이걸 어떤가? 여러분은 책에 마법을 걸 수 있다. 따라서 여러분은 번역서를 만든 다음 이 번역서들에다가 마법을 걸었다. 그리고 이 마법 걸린 책들을 서점에 배포하고 독자들에게 판매를 했다. 이번에도 독자가 치명적인 번역 실수를 제보해왔다. 하지만 여러분은 걱정하지 않는다. 왜냐하면 이미 책에다가 마법을 걸어놨기 때문이다. 여러분은 이 마법의 책 중 하나를 골라서 잘못된 부분을 찾아 수정하니, 다른 마법의 책들도 갑자기 글씨가 날아다니면서 내용이 바뀌어 잘못된 번역 부분이 감쪽같이 수정되었다. 실제로 이러면 얼마나 편한가? 책은 다시 회수할 필요가 없고, 한군데만 수정하면 나머지도 다 자동으로 수정된다. 정말 멋진 마법이다.

다시 소프트웨어로 돌아오자. 여러분이 개발한 소프트웨어를 배포하여 서비스하고 있다. 그러나 버그는 언제든지 발생할 수 있고 여러분은 소스코드를 다시 고쳐야 한다. 만약 이 버그가 치명적이라면 버그를 고치기 전까지 서비스를 중지해야 할 수도 있다. 반드시 버그가 아니라도 서비스는 계속 개선될 수 있고 그때마다 소스코드를 수정하고 빌드와 배포하는 과정을 반복해야 한다. 아직 끝이 아니다. 본격적인 대규모 서비스를 개발한다면 많은 동료 개발자들과 함께 개발하게 된다. 예를 들어 10명의 개발자가 있다고 하면 10명이 자신의 소스코드를 조금씩 수정한다고 하면 최대 10번의 빌드와 배포가 반복된다. 그러다보니 소프트웨어 관리에도 마법의 책이 필요해졌다. 그것이 바로 빌드 자동화와 지속적 통합/배포이다.

빌드 자동화

빌드 자동화는 소스코드에 수정된 내용이 있으면 자동으로 빌드를 시작하여 테스트하여 소프트웨어를 항상 사용가능한 상태로 유지하는 것을 의미한다. 만약 테스트가 실패하면 이 수정된 내용은 소프트웨어에 반영되지 않는다. 개발 중인 프로젝트에 빌드 자동화를 세팅해두면 개발자는 이후부터는 소스코드 작성과 수정에만 집중할 수 있다.

지속적 통합(Continuous Integration)

CI는 협업 환경에서 빌드 자동화라고 생각하면 쉽다. 위에서 말한 것처럼 개발자가 10명일 때, 이 10명은 각자 자기의 소스코드를 수정하는데 만약 이 CI 환경이 갖춰져 있지 않다면 어떻게 해야하겠는가? 원본 소스코드에 개발자 10명이 자기가 작업한 내용을 가져와서 반영한 다음 일일이 빌드가 되는지 테스트하고 실행해볼 것이다. 그리고 테스트가 성공하면 수정내용을 반영하고 아니면 다시 가져가서 수정을 할 것이다. 그러나 CI 환경이 제공되면 개발자는 자기 자리에서 그저 수정한 소스코드만 제출하면 된다. 빌드 자동화를 통해 수정된 소스코드를 빌드하고 테스트하여 통과하면 원본에 통합될 것이다.

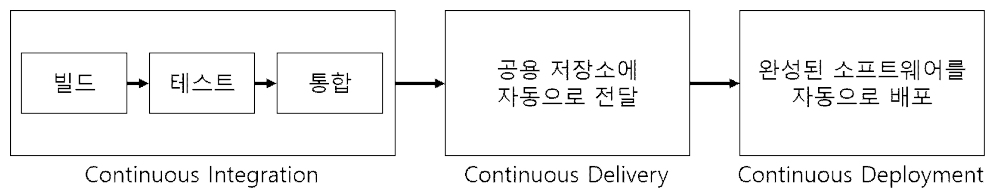
지속적 배포(Continuous Deployment)

CD는 CI를 통해 수정된 소프트웨어를 소비자가 사용할 수 있는 위치까지 자동으로 배포하는 것을 의미한다. 종종 CD는 지속적 배포에 앞서 지속적 전달(Continuous Delivery)을 포함하기도 한다. 개발자가 소스코드를 수정한 이후부터 모든 과정이 자동으로 진행되어 현재 사용중인 소프트웨어가 버전업 되는 것이다.

보통 CI/CD라고 하면 빌드 자동화의 뜻도 내포하고 있다. 또한 대화의 문맥이나 상황에 따라서 CI/CD가 그 자체로 데브옵스를 의미할 때도 있다.

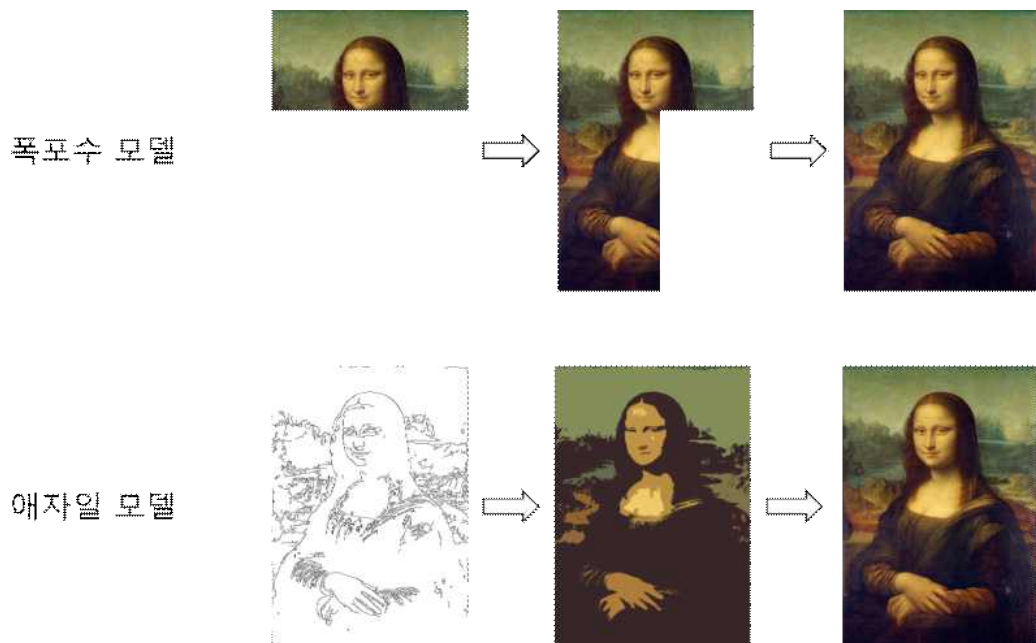
CI/CD 파이프라인

CI/CD 파이프라인은 CI/CD의 과정을 통틀어 부르는 용어이다. 대화의 문맥에 따라서 그냥 CI/CD라고만 하기도 하고, 파이프라인이라고 부르기도 한다.³⁹⁾ CI/CD 파이프라인은 서비스 배포의 단계를 자동화하여 효율적으로 개발하고 소프트웨어를 빠르고 짧은 주기로 소비자에게 배포하기 위한 방법이다.



CI/CD 파이프라인

이와 같은 CI/CD 파이프라인은 개발과 운영의 범위를 모두 포함하고 또한 소스코드의 수정이 빠르게 반영되므로 애자일 개발 방법론에 적합하다. 아래의 그림을 소스코드로 생각해보자. 부분부분 완전하게 완성해나가는 폭포수 모델과 달리 애자일 모델은 완성도는 부족하지만 전체를 그리고, 그리고 이를 자주 반복적으로 수정하고 통합하여 눈으로 확인 가능한 수준까지 배포를 한다.



폭포수 모델과 애자일 모델이 완성을 향해 가는 방법의 차이

39) 컴퓨터 과학에서는 정말 다양한 분야에서 파이프라인이라는 용어가 사용된다. 뜻을 잘 이해하고 사용하도록 하자.

소프트웨어 관리와 데브옵스 그리고 파이프라인

본 장에서는 소프트웨어 관리의 중요성에 대해서 이야기하고 데브옵스가 무엇인지, 그리고 CI/CD 파이프라인을 구축했을 때의 장점에 대해서 알아보았다. 그럼 이제 이것들을 한데 다 묶어서 이야기해보자.

서비스 중심의 소프트웨어 배포 방식에서 소프트웨어의 관리는 서비스를 종료하기 전까지 끊임없이 지속되어야 한다. 이에 고객들에게 빠르고 안정적인 서비스를 지속하기 위해서는 개발과 운영이 긴밀히 협업을 해야하며 이에 따라 데브옵스 문화가 생겨났다. 데브옵스는 개발팀과 운영팀의 소통을 중시하는 조직 문화이며 이를 지원하기 위한 여러 자동화 도구가 생겼다.⁴⁰⁾ CI/CD 파이프라인은 빌드부터 배포까지의 자동화 단계를 의미하며 애자일 개발 방법론을 지원하는데 최적화 되어 있다.

잘 구축된 데브옵스 환경은 앞서 말한 마법의 책처럼 각자가 자신의 일만 잘 하면 그 모든 것들을 자동으로 소프트웨어에 반영하여 새로운 버전으로 배포한다.⁴¹⁾ 결국 이 모든 것은 소프트웨어를 효과적으로 잘 관리하기 위해서 나온 방법들이다.

그리고 이런 협업 방식은 오픈소스 커뮤니티에서 오픈소스 프로젝트를 개발하고 관리하는 방식과 매우 흡사하다. 오픈소스는 수많은 사람들이 개발에 참여하며 프로젝트의 로드맵을 잡고 그에 맞춰 개발을 진행해 나간다.⁴²⁾ 이 또한 애자일 개발 방법론의 그것과 유사하며 오픈소스 프로젝트 관리자들은 참여자들과 함께 소통하면서 프로젝트의 완성도를 높여 나간다.

이미 많은 개발자들이 소프트웨어를 개발하고 관리하는 동안 알게 모르게 많은 부분에서 소프트웨어 공학의 영역을 활용하거나 오픈소스를 사용하고 있다. 따라서 이제는 오픈소스와 오픈소스 생태계는 소프트웨어 개발에 있어서 떼려야 뗄 수 없는 관계가 되었다. 이 책을 읽는 여러분도 오픈소스를 마냥 어려워 하기보다는 천천히 그리고 자주 오픈소스를 접하며 익숙해 지기를 바란다.

40) 개발팀과 운영팀의 소통을 지원하는 이슈 트래커나 문서 도구도 다양하지만 여기서는 빌드 자동화 도구만을 살펴 보았다.

41) 2015년 아마존의 AWS Partner TechShift의 사례 발표에 따르면 전 세계의 아마존 개발자들이 아마존 서비스를 빌드하고 배포한 횟수가 1년에 5천만번이라고 한다. 이는 아마존 내부의 데브옵스 파이프라인을 통해 모든 것이 자동화 되어 있어 가능한 일이다.

42) 심지어 서로 얼굴도 모르는 수많은 사람들이 함께 소통하며 개발을 한다.

Note : 소프트웨어 개발 회사에서 데브옵스는 필수인가?

소프트웨어 개발 회사에서 데브옵스를 운영하는 것은 필수일까? 우선 대답부터 하면 ‘아니오’이다.

데브옵스를 통해 개발팀과 운영팀이 상시 소통하면서 협업을 하면 분명 장점은 많다. 운영을 하면서 발생한 이슈나 새로운 버전과 기능, 또는 버그를 빠르게 개발팀에 전달하여 피드백을 받을 수 있다. 개발팀은 손쉽게 자기가 맡은 부분의 개발만 진행하며 버전업한 소프트웨어를 빌드할 수 있다. 소프트웨어는 자동으로 배포되므로 빠른 주기로 개발을 하고 고객에게 새로운 기능이 적용된 소프트웨어를 지속적으로 제공할 수 있다.

그러나 회사마다 다 상황이 다르고 현실적으로 어려운 점들이 있기 때문에 무조건적으로 데브옵스 환경을 갖추는 것만이 정답은 아니다. 데브옵스 자체가 효율적인 개발과 운영을 위해서인데, 억지로 데브옵스를 구축해서 효율이 더 떨어진다면 구축하지 않는 것이 더 낫다는 것이다.

아래는 필자가 생각하는 데브옵스를 구축하는데 어려운 이유들이다. 만약 회사가 아래와 같은 점들을 많이 가지고 있다면 억지로 데브옵스를 구축하는 것보다 기존의 방식에서 자동화 도구들을 추가하는 정도로만 개선하는 것이 나을 수 있다고 생각한다.⁴³⁾

1. 수직적인 회사 분위기

- 수직적인 회사 분위기는 자유로운 토론 분위기와 커뮤니티를 조성하는데 도움이 되지 않는다. 이는 조직 구성원들을 수동적으로 만들며 이는 소통과 협업을 중시하는 데브옵스와는 맞지 않는다.

2. 기존 소프트웨어 개발 방법과의 차이

- 데브옵스를 위한 자동화 도구들은 분명히 편하지만 인프라를 구축하는 것은 어느정도 난이도가 높고 비용이 든다. 만약 새로운 인프라를 적용하는 것이 힘들고 구성원들도 적응을 못하면 ‘지금은 바쁘니까 일단 이번엔 예전 방식대로 개발하고, 다음에 다 같이 공부해서 새로 적용하자’와 같이 타협을 할 수 있다. 그러나 한번 이렇게 타협하게 되면 앞으로도 새로운 방식을 적용하기는 힘들 것이다.

3. 역할의 모호함

- 규모가 작은 회사일수록 개발자가 운영까지 동시에 도맡는 경우가 많다. 이 같은 경우는 어차피 혼자서 하던 작업이므로 굳이 업무를 나눌 필요가 없다. 자신에게 필요한 자동화 도구만 찾아서 적용하면 더 쾌적한 환경을 구축할 수 있을 것이다.

4. 폭포수 모델이 필요한 경우

- 회사 내의 서비스나 프로젝트가 아니라 외부 고객의 프로젝트(용역이나 외주 등)를 수행하는 경우, 고객의 요구사항에 맞춰서 폭포수 모델로 진행해야 하는 경우가 있다. 이 같은 경우는 기존의 개발 방법이 더 나을 수 있다.

43) 데브옵스 환경이 아니더라도 빌드 자동화 도구나 CI/CD 파이프 라인 등은 얼마든지 활용할 수 있다. 데브옵스와 자동화 도구를 헷갈려서는 안된다. 데브옵스 환경이라고 얘기할 수 있으려면 이러한 자동화 인프라에 소통을 통해 협업하고 개발/운영하는 문화가 존재해야 한다.

오픈소스 소프트웨어에서 찾아볼 수 있는 소프트웨어 공학은 어떤 것이 있을까? 이번 장에서는 소프트웨어 공학에서 오픈소스가 가지는 역할을 알아보도록 한다. 그리고 오픈소스 소프트웨어 프로젝트에서 사용하는 개발 방법들을 알아보도록 하자.

소프트웨어 공학은 소프트웨어를 잘 만들기 위한 방법을 가르친다고 하였다. 그리고 앞서 소개한 소프트웨어 관리와 데브옵스의 경우도 소프트웨어를 잘 개발하고 관리하기 위함이었다. 그러다보니 최신 소프트웨어 공학서에는 데브옵스나 CI/CD 파이프라인을 소개하는 경우도 종종 보인다. 그리고 오픈소스의 개발 방법을 소프트웨어 공학에 접목하여 쓴 책이 있는데, 바로 OSI의 설립자 중 한명인 에릭 레이먼드가 쓴 “성당과 시장(The Cathedral and the Bazaar)”이라는 책이다. 여기서는 오픈소스의 공개 방식에서 두가지를 이야기 하고 있는데, 하나는 성당 모델이고 다른 하나는 시장 모델이다. 성당은 건설할 때 설계도가 필요하고 시장은 사람들이 많이 다니는 곳에 자연스럽게 형성된다.⁴⁴⁾ 이 책은 이를 소프트웨어에 비유한 것이다.

성당 모델 : 개발을 할 때에는 비공개이며 출시하며 공개

시장 모델 : 개발을 할 때부터 공개된 상태

레이먼드는 ‘보는 눈이 많으면 버그도 쉽게 잡을 수 있다.’⁴⁵⁾라고 말한다. 즉, 소스코드를 공개해서 개발을 하면 수많은 사람들이 함께 개발하는 셈이므로 훨씬 빠르고 완성도가 높은 소프트웨어를 만들 수 있다고 말한다.⁴⁶⁾ 그렇다고 소프트웨어를 공개만 해놓는 것으로 끝나진 않을 것이다. 오픈소스 프로젝트도 좋은 소프트웨어를 만들기 위해서는 오픈소스를 위한 개발 방법론이 필요했다. 그리고 이는 오픈소스 커뮤니티가 점차 성장해가면서 구체적인 모습을 보이기 시작한다.

앞서 소개한 것처럼 대표적인 공개 오픈소스 커뮤니티인 깃헙에는 수많은 오픈소스 프로젝트가 등록되어 있고 이를 사용하거나 또는 개발하기 위해 많은 개발자들이 함께하고 있다. 따라서 깃헙에서 오픈소스 프로젝트가 진행되는 것을 보면 오픈소스의 유형이나 개발 방법들을 잘 알 수 있다. 즉, 오픈소스 커뮤니티에서 자연스럽게 오픈소스 소프트웨어를 잘 만들기 위한 방법이 제시되고 있고 사람들은 그에 따라 오픈소스를 개발하고 있다. 레이먼드가 여기까지 예상하고 한 말인지는 모르겠지만, 필자가 보기에 마치 정말로 “시장(Bazaar)” 같다.

44) Bazaar는 Market과는 조금 다른 느낌으로, 천막이나 가건물, 행상들로 이루어진 흡사 옛날 전통시장과 같은 형태로 이해하는 게 좋다.

45) 그는 이 이론을 리눅스 개발자인 리누스 토발즈(Linus Torvalds)의 이름을 따서 리누스 법칙(Linus's Law)이라고 이름 지었다. 실제 이 문구는 ‘given enough eyeballs, all bugs are shallow’이다.

46) 많은 오픈소스 지지자들이 이야기하는 소스코드를 공개해서 개발을 해야 하는 이유이다.

오픈소스 프로젝트 개발

처음 오픈소스 프로젝트를 시작한다면 어디서부터 어떻게 시작해야할까? 성당 모델처럼 전부 다 만든 다음 공개하는 것이 나올까? 아니면 시장 모델처럼 아예 시작부터 공개를 하는 것이 나올까? 시작부터 공개한다면 그 시작은 어디서부터인가? 다양한 오픈소스가 있는 만큼 오픈소스 프로젝트를 시작하는 방법도 다양하다. 아래는 대부분의 오픈소스 프로젝트가 시작하는 방법들이다.

- 1) 프로젝트의 필요성을 느낀 개인이 공개적으로 프로젝트의 개발 의도를 밝히는 방법
- 2) 아주 간단하지만 동작은 하는 베이스 코드를 공개하는 방법
- 3) 프로젝트를 거의 완성하고서 공개하는 방법
- 4) 다른 프로젝트를 포크(fork)하는 방법

1)번의 경우 공개된 소스코드 없이 개발 방향성만 제시하며 함께할 개발자를 모으는 방법이다. 이 같은 경우는 2)번의 베이스 코드를 공개하며 시작하는 방법보다는 좋지 않은 방법이라고 한다.⁴⁷⁾ 4)번의 경우는 다른 프로젝트를 포크(fork)하며 시작하는 방법인데 여기에 대해서는 신중한 접근이 필요하다. 일반적으로 오픈소스 프로젝트를 시작하기 전에 이미 유사한 프로젝트가 있는지 조사가 필요하다. 그리고 새로운 프로젝트를 시작하는 것과 기존의 프로젝트에 기여하는 것 중 무엇이 더 나은지를 판단해야 한다. 만약 기존의 오픈소스 프로젝트가 오랜 기간 동안 개발과 참여가 정체되어 있다고 한다면 포크하여 새로운 프로젝트를 시작하는 것이 좋을 것이다. 그렇지 않고 많은 개발자들이 활발히 참여하고 있는 프로젝트를 포크한다면 그것은 오픈소스에 기여하는 개발자들을 양분하거나 새로운 프로젝트에는 개발자를 모으지 못할 수도 있다.

Note : 포크(Fork)

포크는 어떤 소프트웨어의 소스코드를 복사하여 새로운 소프트웨어를 만드는 것을 말한다. 사전적 의미로는 ‘갈라지다’ 라는 뜻을 가지고 있다. 즉, 기존의 A라는 오픈소스 프로젝트가 있었다고 하면 이를 포크하면 A'라는 새로운 오픈소스 프로젝트가 생기는 것이다. 두 프로젝트는 독립적이며 서로에게 영향을 주지 않는다. 대표적인 포크 사례로는 레드햇 리눅스에서 포크한 CentOS나 MySQL에서 포크한 MariaDB 등이 있다.

또한 최근에는 블록체인의 암호화폐에서 심심찮게 포크가 이루어지는데, 블록체인의 경우 한번 배포되면 수정이 불가능하기 때문에 포크를 통해 기능을 개선하거나 문제점을 보완하기도 한다. 대표적으로 비트코인에서 비트코인 캐시나 비트코인 골드가 나오거나, 이더리움에서 이더리움 클래식으로 포크되어 나온 사례가 있다.

47) 에릭 레이먼드는 그의 저서 “성당과 시장”에서 그간 자기가 오픈소스 프로젝트를 수행하며 관찰하였을 때, 개발 의도만 발표하는 것보다는 조금이라도 실행 가능한 소스코드를 공개하여 시작한 프로젝트들이 더 나왔다고 한다.

오픈소스 개발을 위한 도구의 변화

오픈소스의 개념이 나온 이후 다양한 도구들을 오픈소스 개발에 사용해 왔다. 그리고 이러한 개발 도구들 또한 오픈소스로 나온 것이 많으며 함께 발전해왔다. 몇가지 대표적인 개발 도구들을 한번 살펴보자.

오픈소스 프로젝트를 개발하기 위해서는 소통과 협업이 필수적이었다. 그러나 대부분의 경우는 물리적인 거리가 떨어져 있었으므로 대화를 위한 채널이 필요했다. 옛날에는 이 역할을 전자 메일이 담당하였다. 메일링 리스트는 개발에 참여하는 기여자들에게 효과적으로 내용을 전달할 수 있었으며 오프라인이나 서신을 주고 받는 방법보다는 훨씬 빨랐다. 그리고 컴퓨터와 네트워크 기술도 점점 발전하였고 비동기적인 전자 메일 외에 IRC와 같이 실시간 커뮤니케이션을 지원하는 인스턴트 메시지를 사용하게 되었다. 그리고 현재에 이르러서는 깃헙과 같은 커뮤니티도 생기면서 이제 오픈소스를 개발할 때 사용할 수 있는 커뮤니케이션 채널은 대단히 많이 늘어났다.

버전 관리 시스템은 오픈소스를 개발할 때 필수적인 협업 개발 도구이다. 이는 뒤에서 다시 자세히 다루도록 한다. 버전 관리 시스템도 초기에는 CVS(Concurrent Versions System)을 활용하다가 그 이후 SVN(Subversion, SVN)을 많이 활용하였다. 지금에 이르러서는 대부분의 버전 관리는 git을 사용하고 있다.

이슈 트래커는 프로젝트 개발을 하며 발생하는 다양한 문제들을 추적하게 해준다. 이슈 트래커를 사용하지 않을 때에는 소스코드의 라인을 알려주거나 주석으로 표시하여 찾았다. 그러나 이슈 트래커가 나온 이후에는 클릭 몇 번으로 문제가 발생한 소스코드를 바로 찾을 수 있게 되었다. 이러한 이슈 트래커는 Mozilla의 BugZilla나 Mantis, Redmine 등이 대표적이었다. 최근에는 Atlassian 사의 JIRA가 수준 높은 전사적인 프로젝트 관리와 이슈 트래킹을 제공하고 있다.

그 외에는 테스트를 자동화 해주는 툴이나 디버깅을 위한 도구 등도 있다. 이제는 빌드 자동화와 더불어 CI/CD의 개념도 생겨났다.

이러한 개발 도구들의 발전과 변화는 오픈소스 개발 방식도 다양하게 변화해 왔다는 것을 잘 보여준다. 물론 앞으로도 계속 다양한 방법들과 도구들이 새로 나타날테니 언제나 변화에 준비하고 있어야 할 것이다.

3

리눅스 시스템

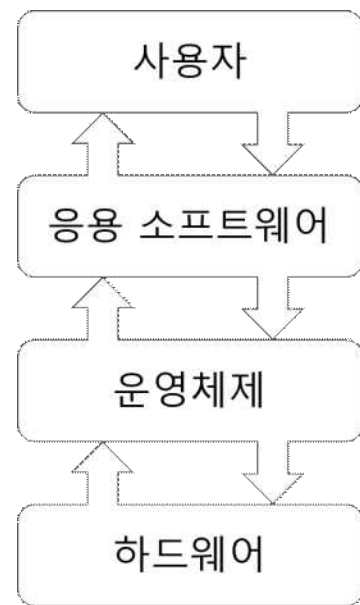


오픈소스 소프트웨어와 리눅스는 떼려야 뗄 수 없는 관계이다. 리눅스가 바로 오픈소스의 대표적인 소프트웨어이기 때문이다. 본 장에서는 리눅스가 처음 나오게 된 배경과 오픈소스로서의 리눅스에 대해 이야기해 보고자 한다.

운영체제(Operating System)는 컴퓨터에서 가장 중요한 소프트웨어로써 하드웨어와 응용 소프트웨어를 관리해주는 역할을 한다. 우리가 일반적으로 컴퓨터라고 부르는 기계는 범용(general purpose)의 성격을 띄고 있는데, 이는 운영체제가 하드웨어의 자원을 적재적소로 관리해주며 다양한 응용 소프트웨어를 이해할 수 있기 때문이다. 덕분에 우리는 한 대의 컴퓨터로 한글, 워드와 같은 문서작업이나 인터넷 브라우저를 통해 웹 서핑이나 유튜브와 같은 방송을 보고 또 게임을 설치하여 즐기는 등 목적에 맞는 소프트웨어를 설치하여 여러 가지 방법으로 사용할 수 있다.

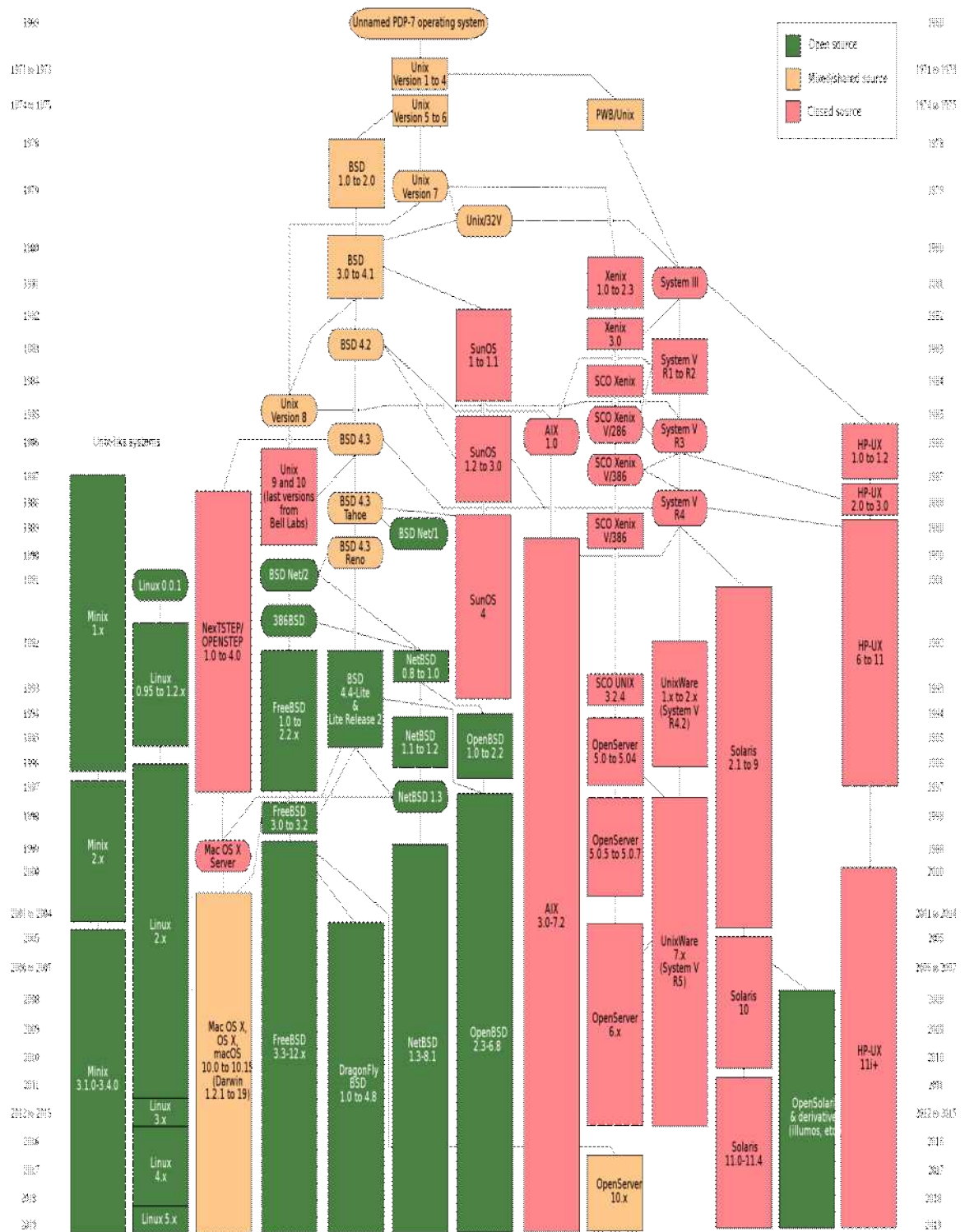
반대로 운영체제가 없는 경우의 예로 에어컨의 리모컨(remote control)을 생각해보자. 이 리모컨의 하드웨어에는 운영체제 없이 에어컨의 소프트웨어가 바로 설치되어 있으며 원격으로 에어컨을 동작시킬 수 있다.⁴⁸⁾ 그러나 이 리모컨으로 TV를 동작시킬 수는 없다. 리모컨의 하드웨어에 TV 소프트웨어를 설치했다 하더라도 리모컨의 하드웨어와 TV 소프트웨어는 서로를 이해하지 못하기 때문이다. 그러나 리모컨과 비슷한 크기의 스마트폰은 어떤가? 요새는 가전기기 제조사들이 스마트폰용 앱도 내고 있어서 에어컨 앱과 TV 앱을 설치하면 스마트폰만으로 두 기기를 제어할 수 있다. 스마트폰이 두 기기의 앱, 즉 소프트웨어를 둘 다 설치하고 사용할 수 있는 이유는 안드로이드나 iOS 같은 운영체제가 있기 때문이다. 스마트폰의 운영체제가 에어컨 앱을 쓸 때는 에어컨 리모컨처럼, TV 앱을 쓸 때는 TV 리모컨처럼 스마트폰 하드웨어를 관리하므로 가능한 일이다.

이처럼 컴퓨터를 범용적인 기계로 만들기 위해서는 운영체제의 역할이 절대적이며 리눅스(Linux)는 마이크로소프트의 Windows, Apple의 macOS와 더불어 현대 컴퓨터 산업에서 가장 중요한 운영체제 중 하나이다. 특히나 리눅스는 가장 대표적인 오픈소스로서 오픈소스를 이야기할 때 빠질 수 없는 소프트웨어이기도 하다.



운영체제의 역할

48) 보통 이렇게 하드웨어에 운영체제 없이 바로 설치되는 소프트웨어를 펌웨어(Firmware)라고 부른다.



유닉스 계열 운영체제의 종류와 역사49)

49) 이미지 출처 : 위키백과 “유닉스 계열”, (wikipedia.org)

리눅스는 앞서 오픈소스의 역사에서도 자주 언급되었던 리누스 토발즈(Linus Torvalds)가 개발한 운영체제 커널(Kernel)로 시작하였다. 커널은 운영체제의 구성 요소 중 하나이며, 컴퓨터 하드웨어와 소프트웨어를 이어주는 핵심 인터페이스라고 생각하면 쉽다. 리눅스 커널은 처음에는 토발즈가 취미 삼아 개발했던 것으로 알려져있지만, 지금은 없어서는 안 될 가장 중요한 소프트웨어 중 하나이다.

커널이 아니라 지금 우리가 알고 있는 운영체제로써의 리눅스는 조금 재미있는 사연을 가지고 있다. 1983년, 리처드 스톨만이 자유 소프트웨어로 구성된 유닉스를 만들기 위해 GNU(GNU is Not Unix) 프로젝트를 시작했다. 그리고 GNU 프로젝트를 통해 운영체제를 구성하는 많은 소프트웨어들을 만들었다. 시스템 라이브러리, 컴파일러(gcc), 텍스트 에디터(vi), 셸(bash) 등 지금도 리눅스와 많은 운영체제에서 사용하고 있는 수많은 소프트웨어들이 이 GNU 프로젝트를 통해 개발되었다. 그러나 막상 GNU 프로젝트를 통해서 커널만은 만들지 못하고 있었는데, 1991년 리눅스 커널이 나온 것이다. 토발즈가 만든 리눅스 커널은 유닉스 계열이었고 자유 소프트웨어였기 때문에 많은 사람들이 리눅스 커널 + GNU 프로젝트 소프트웨어의 조합으로 운영체제를 만들어서 쓰기 시작했다. 이는 급속도로 퍼져나가기 시작했고 이후 이 운영체제를 리눅스(Linux)라고 부르게 되었다.⁵⁰⁾

리눅스 배포판

리눅스 배포판(Linux Distribution)은 리눅스 커널과 GNU 소프트웨어, 그리고 다양한 자유 소프트웨어로 구성된 운영체제이다. 초기 리눅스는 커널과 GNU 소프트웨어들이 모여있는 이미지로 배포되다가 설치와 설정 등이 복잡하다보니 회사나 커뮤니티 단위로 이를 단순하게 하기 위해 여러 가지 배포판들이 발생하게 되었다. 이 배포판에는 각 회사나 커뮤니티가 포함한 소프트웨어들이 기본적으로 들어가면서 리눅스 배포판마다 조금씩 차이가 나게 되었다. 따라서 배포판에 따라 기본적으로 설치되어 있는 소프트웨어랑 경로가 조금씩 다르므로 리눅스를 공부할 때에는 자신이 사용하는 배포판이 어떤 것인지 확인하고 사용해야한다. 물론 초기 설정만 다르며 호환만 된다면 다른 배포판이 사용하는 소프트웨어를 설치하여 사용해도 문제는 없다.

대표적인 배포판으로는 RedHat사의 RHEL, CentOS, Fedora 등이 있고 또 Debian, Ubuntu 등이 유명하다. RedHat사의 세가지 배포판(RedHat, CentOS, Fedora)은 각각의 이유로 차이를 두고 있는데, RHEL은 정식으로 RedHat Enterprise Linux이다. 이 배포판은 매우 안정적이며 RedHat이 직접 기술 지원을 하므로 유료이다. 그리고 CentOS는 RHEL에서 갈라져 나온 배포판으로 RHEL과 대단히 유사하지만 안정성이 조금 떨어진다. 그 이유는 RedHat사가 CentOS에서 먼저 소프트웨어들을 테스트하고 사용자들의 평가를 받아 RHEL에 적용하기 때문이다. 즉, 유료 배포판의 안정성을 위해 CentOS를 무료로 지원하고 있다. Fedora는 개인 사용자들을 대상으로 만든 조금 더 가벼운 느낌의 리눅스 배포판이다.

50) 이런 역사로 인해 자유소프트웨어재단(FSF)에서는 그냥 리눅스라고 부르지 않고 반드시 GNU/Linux라고 부르고 있다.

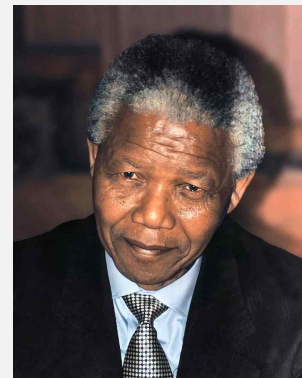
종류	소개	장점	단점
RHEL	RedHat에서 유료 기술지원을 하는 기업용 리눅스	<ul style="list-style-type: none"> 안정성이 높음 최신 기술 지원 	<ul style="list-style-type: none"> 유료
CentOS	RedHat 계열, RHEL과 호환되는 무료 기업용 리눅스	<ul style="list-style-type: none"> 안정성이 높음 RHEL과 유사 	<ul style="list-style-type: none"> RHEL과 같은 유료 지원을 받을 수 없음
Fedora	RedHat 계열, 개인 사용자용	<ul style="list-style-type: none"> RHEL 기술 탑재 	<ul style="list-style-type: none"> 안정성이 낮음
Debian	데비안 계열, 현재 가난 널리 사용되고 있는 배포판 중 하나	<ul style="list-style-type: none"> 안정성이 높음 패키지가 많음 	<ul style="list-style-type: none"> 패키지가 오래됨
Ubuntu	데비안 계열, GNU의 공식 후원을 받는 유일한 배포판으로 안정성이 좋음	<ul style="list-style-type: none"> 사용이 쉬움 	<ul style="list-style-type: none"> 낮은 버전업
Mint	우분투 기반, 쉽고 간편한 UI 지원	<ul style="list-style-type: none"> 사용이 쉬움 	<ul style="list-style-type: none"> 낮은 버전업
SUSE	해외에서 많이 사용하며 서버로써 높은 안정성을 보임	<ul style="list-style-type: none"> 안정성이 높음 	<ul style="list-style-type: none"> 저장소가 작음

데비안(Debian)은 데비안 프로젝트에서 개발한 운영체제로 오래된 만큼 안정성이 높고 지원하는 패키지도 대단히 많다. 데비안 계열은 패키지를 설치할 때 APT라는 패키지 매니저를 사용하여 소프트웨어를 설치하고 업데이트 할 수 있어 사용자에게 큰 매력으로 다가온다. 데비안 계열인 우분투(Ubuntu)는 개인 사용자에게 가장 인기있는 리눅스 배포판이며 강력한 패키지 관리 툴과 쉽고 깔끔한 인터페이스로 많은 인기를 누리고 있다. 본 책에서도 실습 파트에서 우분투 배포판을 사용한다.

Note : 우분투(Ubuntu)의 어원

우분투는 남아프리카의 반투어에서 유래된 말로, 사람들간의 관계와 헌신에 중점을 둔 윤리 사상이다. 우분투의 개발사인 캐노니컬은 “타인을 향한 인간애” 또는 “내가 있으니 내가 있다”라는 의미로 이 단어를 리눅스 이름으로 채택했다. 세계 최초의 흑인 대통령인 넬슨 만델라는 우분투에 대해 아래와 같이 이야기하였다. 이는 오픈소스의 철학과도 딱 맞는 단어가 아닌가 한다.

“옛날에 우리가 어렸을 적에 여행자가 우리 마을에 들르곤 합니다. 여행자는 음식이나 물을 달라고 할 필요가 없습니다. 들르기만 하면 사람들이 밥상에 음식을 차려주기 때문입니다. 이것은 우분투의 한 측면이고, 다양한 측면이 있을 것입니다. 우분투는 사람들이 자신을 위해 일하지 말라는 것이 아닙니다. 중요한 점은, 그렇게 하는 것이 여러분 주변의 공동체가 더 나아지게 하기 위해서 그 일을 하느냐는 것입니다. 이런 것들이 인생에서 가장 중요한 것들이고, 만일 여러분이 그런 일을 한다면, 다른 사람들이 고마워 할 아주 중요한 일을 한 것입니다.”



3-2

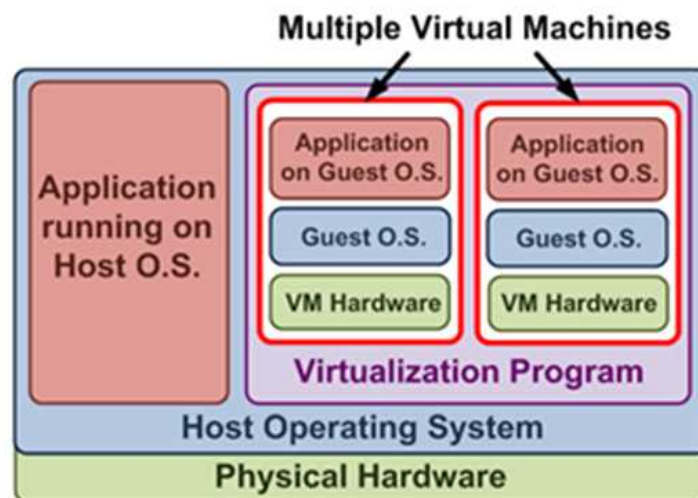
가상머신에 리눅스 설치하기

본 장에서는 실제로 리눅스를 사용해보기 위해 가상머신을 설치하고 그 위에 우분투를 설치하고자 한다. 그리고 네트워크로 서로 통신하며 실제 개발환경과 유사하게 구성하여 실습을 진행한다. 마지막으로 Vagrant 툴을 사용하여 한번에 여러개의 머신들을 자동으로 설치하는 것을 테스트 해본다.

이번 장부터는 오픈소스를 실제로 사용해보기 위해 다양한 실습들을 진행한다. 그에 앞서 실습을 진행할 리눅스 환경부터 구축하고자 한다. 리눅스는 윈도우즈와 같은 운영체제이기 때문에 실제로 리눅스를 사용하려면 대부분의 독자들이 지금 사용하고 있는 윈도우즈 컴퓨터 외에도 한 대 이상의 컴퓨터가 더 필요하다. 그러나 실습을 위해 컴퓨터를 새로 살 수도 없는 노릇이므로 대부분의 경우 리눅스 실습은 가상머신으로 진행하는 경우가 많다.

가상머신(Virtual Machine)

가상머신은 물리적인 컴퓨터가 아니라 소프트웨어로 만들어진 가상의 컴퓨터라고 생각하면 된다. 소프트웨어로 가상의 CPU, 메모리 등의 하드웨어를 만들고 그 위에 운영체제를 설치하여 마치 하나의 컴퓨터처럼 사용하는 것이다. 가상머신은 소프트웨어로 만드는 컴퓨터이므로 얼마든지 복사가 가능하고 따라서 원래 물리 컴퓨터의 자원만 허락한다면 얼마든지 가상의 컴퓨터를 만들어 낼 수 있다는 것이다. 즉, 1대의 물리 컴퓨터로 마치 2대, 3대 이상의 컴퓨터를 사용하는 것 같은 효과를 볼 수 있다. 다만 이 가상의 머신들은 원래 물리 컴퓨터의 자원을 나누어 가져가므로 가상머신에 줄 수 있는 자원의 한계는 존재한다.



가상머신의 구조

이러한 가상머신은 실제 소프트웨어 개발 회사에서도 다양한 이유로 사용이 되는데, 가상머신의 장점으로는 아래와 같은 것들을 들 수 있다.

- 물리적 하드웨어 구축 및 유지 비용을 절감
- 대규모 시스템을 구축할 때 복제해서 사용하면 되므로 편리함
- 가상의 시스템이 고장나거나 문제가 발생했을 시 복구가 수월함
- 기종이 다른 시스템(하드웨어) 위에서도 호환성을 보장함

먼저 물리적 하드웨어 구축 비용의 절감은 우리가 이번에 실습을 하면서 가상머신을 사용하는 이유와 같다. 개발을 하면서도 네트워크를 확인하거나 여러대의 개발 장비가 필요한 경우가 많은데 이때마다 컴퓨터를 사는 것은 비효율적이다. 이 때에는 가상머신을 사용해서 개발을 한다.

실제 대규모 시스템을 구축할 때에도 가상머신을 많이 활용한다. 이 때에는 매번 컴퓨터에 직접 구축하는 것이 아니라 가상머신 한 대에 구축을 하고 이후 설정값만 조금 변경하여 복사를 하면 되므로 대규모 시스템 구축이 편해진다. 이에 대한 실습도 뒤에서 해보도록 한다.

가상의 시스템이 고장났을 때에는 여차하면 가상머신을 지워버리고 다른 머신을 복사해와도 된다. 가상머신 시스템의 복구를 가장 손쉽게 하는 방법이다.

마지막으로 우리가 사용하는 대부분의 소프트웨어는 운영체제에 의존적이다. 따라서 종종 어떤 프로그램을 설치할 때, 윈도우용과 리눅스용, 맥용 프로그램이 구분된 것을 본 적이 있었을 것이다. 운영체제가 다르기 때문에 똑같은 소스코드를 컴파일해도 리눅스에서 컴파일 한 결과와 윈도우에서 컴파일한 결과가 다르다. 따라서 소프트웨어로 만들고 난 이후에도 호환이 되지 않는다. 그런데 가상머신을 쓰면 다른 시스템에서도 호환성을 보장할 수 있다. 대표적인 것이 자바버추얼머신(Java Virtual Machine, JVM)이다. 자바는 리눅스에서 컴파일을 하든 윈도우에서 컴파일을 하든 동일한 결과가 나온다. 이유는 자바는 JVM이라 불리는 가상머신 위에서 소프트웨어를 실행하기 때문이다. 따라서 자바는 한번만 개발하면 어떤 플랫폼이나 운영체제에서도 동일한 사용환경을 제공할 수 있기 때문에 뛰어난 호환성으로 많은 인기를 얻었다. 물론 이 가상머신들은 리눅스용 가상머신이나 윈도우용 가상머신 등 각 플랫폼 별로 다 존재한다. JVM이 운영체제별로 따로 개발된 대신 그 위에서 실행되는 자바는 JVM만 있으면 실행할 수 있게끔 한 것이다. 따라서 가상머신은 서로 기종이 다른 시스템이나 완전히 다른 하드웨어 위에서도 호환성을 제공할 수 있다.

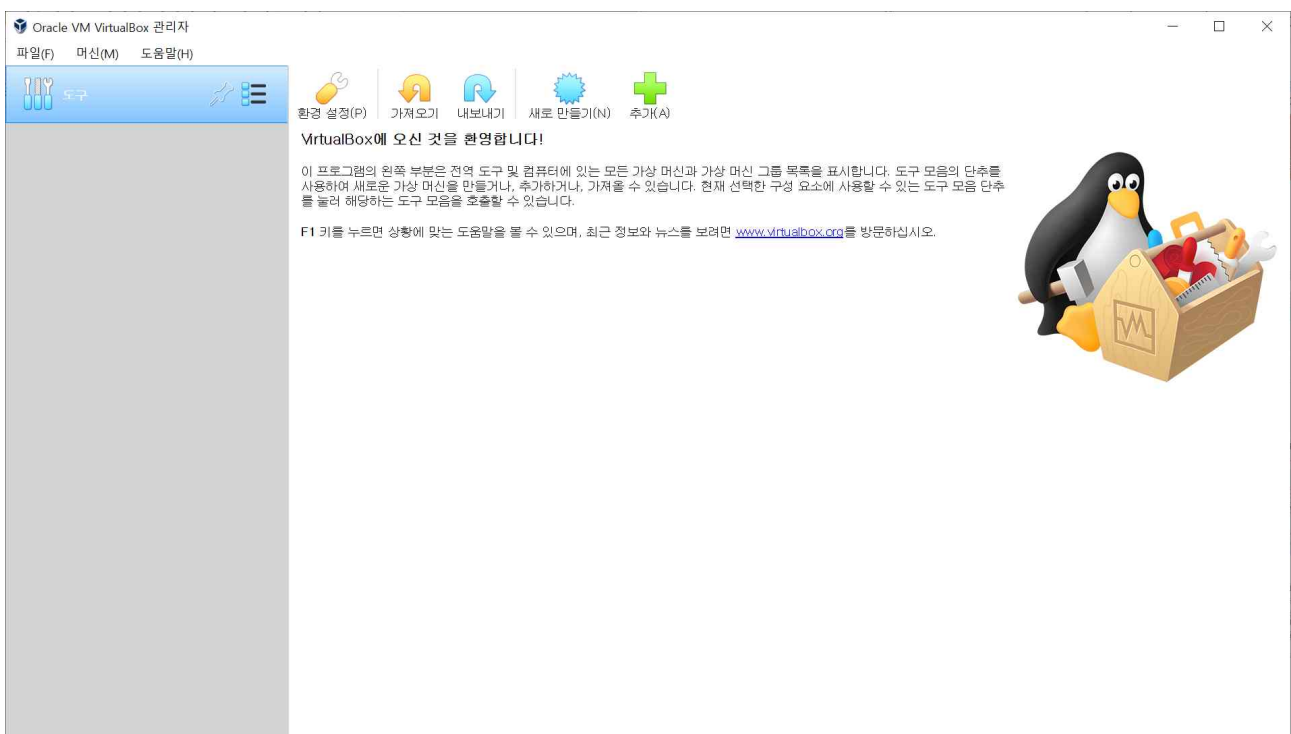
버추얼 박스 설치

버추얼 박스(Virtual Box)는 Innotek이라는 회사에서 개발한 가상머신 소프트웨어이다. 이후 자바로 유명한 Sun Microsystems에 인수되었다가 Sun Microsystems가 Oracle에 인수되면서 이제 Oracle VM VirtualBox로 배포되고 있다. 버추얼 박스 또한 GPL로 공개된 오픈소스이다.

우선 버추얼 박스를 설치하기 위해 공식 홈페이지(virtualbox.org)에서 설치파일을 다운로드 받자. 버추얼 박스 뿐만 아니라 이 책에서 진행하는 실습에서 설치해서 사용하는 소프트웨어들마다 버전이 있을텐데, 필자가 이 책을 쓸 때와 독자들이 볼 때에는 시간적 차이가 있으므로 버전이 다를 수 있다. 우리가 진행하는 대부분의 실습에서는 메이저 버전이 다르지 않은 이상 큰 차이는 나지 않으므로 적당히 최신 버전을 다운로드 받아서 사용하도록 하자.

버추얼 박스를 다운로드 받아서 설치하고 실행하면 아래와 같은 화면을 볼 수 있다. 설치과정에서 딱히 주의할 점은 없으며 일반적인 소프트웨어를 설치하는 것과 같이 진행하면 된다. 만약 기존에 버추얼 박스를 사용했던 사람이라면 이후 진행하다가 버추얼 박스의 설정 문제로 실습 예제가 제대로 되지 않는 경우에는 지웠다가 다시 설치하는 것도 한 방법이다.

한가지 주의할 점은 이 실습은 최소 2대 이상의 가상머신을 실행하므로 노트북과 같이 자원이 작은 컴퓨터에서 본 실습을 진행하면 자원 부족으로 제대로 실행이 되지 않을 수 있으니 참고하길 바란다.

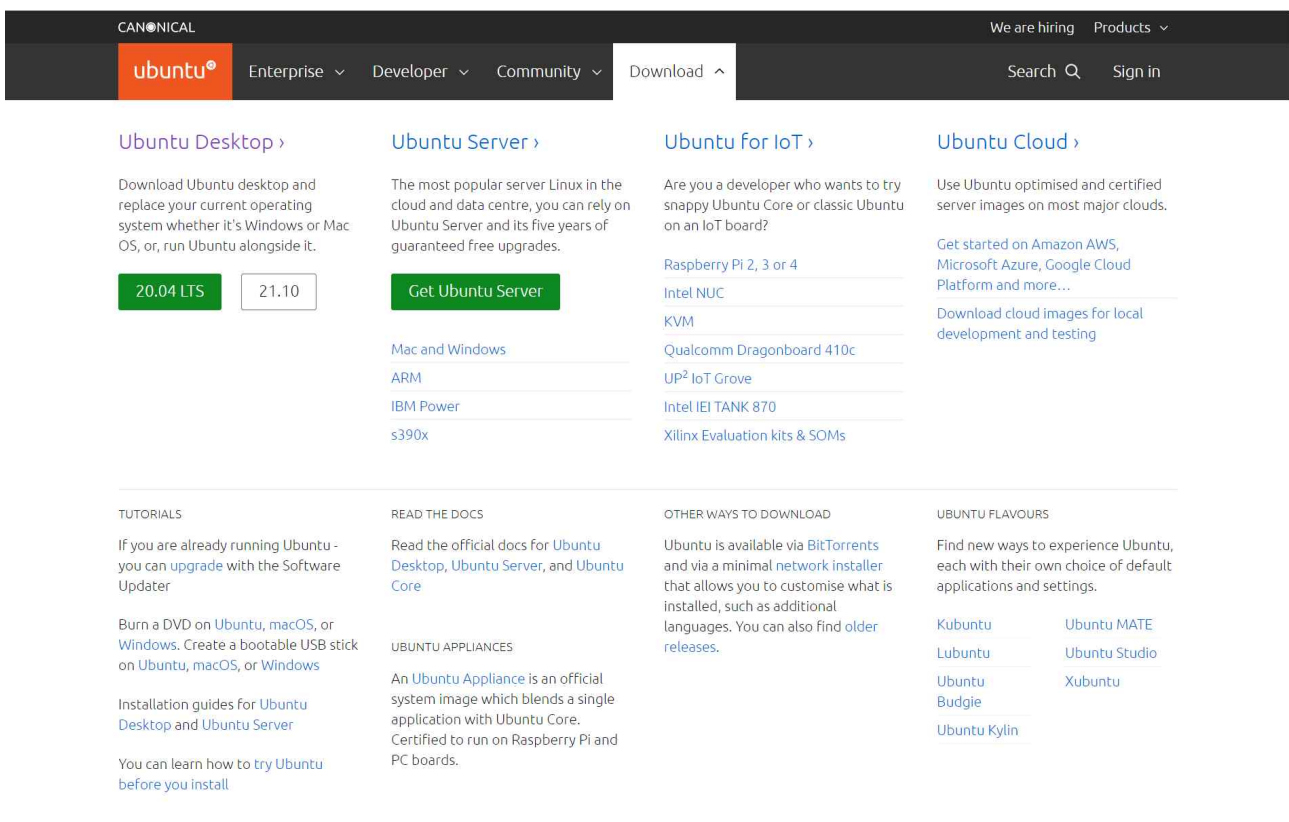


버추얼 박스 실행 화면

우분투 설치

버추얼 박스를 설치했다면 이제 우분투 리눅스 배포판을 다운로드 받아서 가상머신에 설치해보자. 우선 우분투 홈페이지(ubuntu.com)에서 우분투 최신버전을 다운로드 받자. 버전은 실습을 진행하는 시기에 따라서 다를 수도 있지만 적어도 본 실습에서는 크게 중요하지는 않다. Ubuntu Desktop을 다운로드 받으면 되는데 LTS와 그렇지 않은 것이 있을 것이다. LTS는 Long Term Service의 약자로써 더 오랜기간 동안 지원하는 버전이라는 의미이다.⁵¹⁾ 여기서는 21.10 버전을 선택하여 다운로드 받았다.

만약 이전 버전의 우분투를 다운로드 받고 싶다면 older releases 메뉴를 선택하여 원하는 버전을 찾아서 다운로드 받으면 된다. 그리고 최근의 PC들은 대부분 64bit 운영체제를 사용하지만 혹여 자신이 32bit 운영체제를 사용 중이라면 우분투도 32bit로 다운로드 받아서 사용해야 한다.

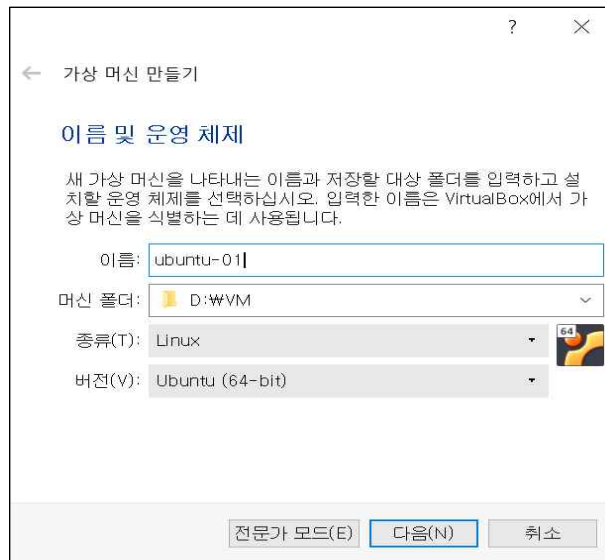


The screenshot shows the Ubuntu website's download page. The header includes the Canonical logo and navigation links for Enterprise, Developer, Community, and Download. The main content area is divided into four columns: Ubuntu Desktop, Ubuntu Server, Ubuntu for IoT, and Ubuntu Cloud. Each column has a brief description and a 'Get' button. The Ubuntu Desktop section shows two versions: 20.04 LTS and 21.10. Below the main content, there are four columns of links for Tutorials, Read the Docs, Other ways to download, and Ubuntu Flavors.

우분투 홈페이지의 다운로드 페이지에서 Ubuntu Desktop을 다운로드 받자

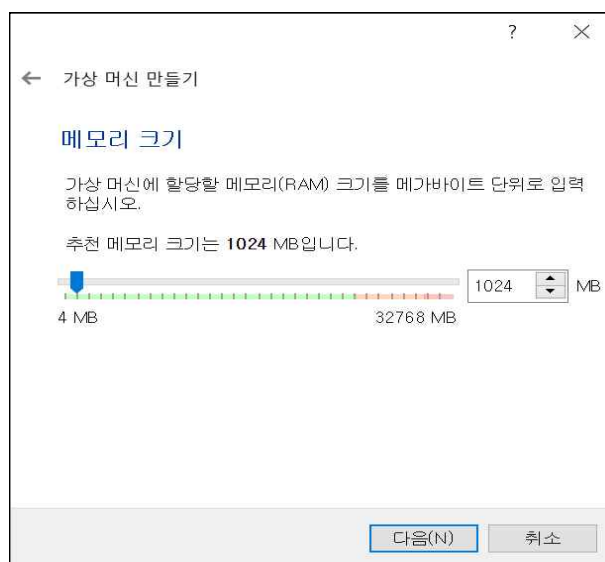
51) 우분투는 6개월에 한 번씩 버전이 오른다. 이는 우분투의 정책이지만 너무 잦은 버전업으로 인해 사용하는 사람들이 혼란을 느끼기도 한다. 이에 우분투에서 제공하는 것이 LTS 버전인데, LTS 버전들은 2년을 유지하면서 해당 버전에 대한 지원을 해준다.

우분투를 다운로드 받았다면 새로운 가상머신을 만들고 그 가상머신에 우분투를 설치하도록 하자. 먼저 버추얼 박스의 메인화면에서 ‘새로 만들기’ 버튼을 찾아서 누르면 아래와 같은 화면이 나온다.



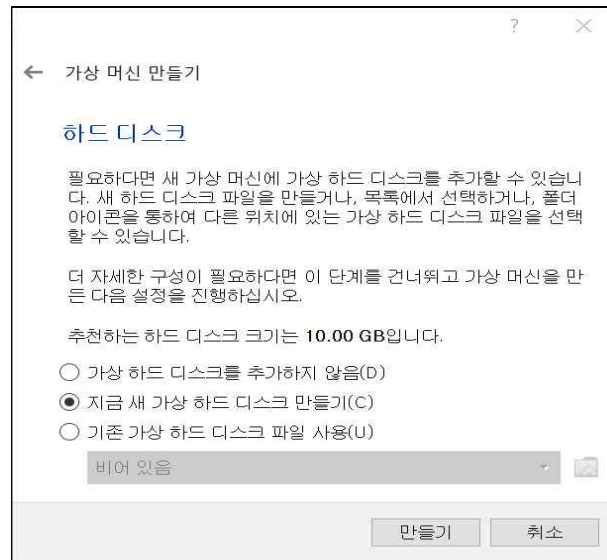
이름에는 ‘ubuntu-01’ 이라고 해주었다. 이후 ubuntu-02와 ubuntu-03도 만들 예정이다. 머신 폴더는 가상머신이 저장될 위치이다. 경우에 따라서 10GB까지 나갈 수 있으니 용량이 충분한 디스크에다가 설정을 해주자. 종류는 Linux, 버전은 Ubuntu(64-bit)를 선택해준다. 버전의 경우 자신의 컴퓨터가 32bit 운영체제를 사용한다면 여기서도 32bit를 선택해주어야 한다.⁵²⁾

그 이후에는 이 가상머신이 사용할 메모리를 설정해준다. 물리 컴퓨터의 크기 내에서 선택할 수 있으며 최소 1024MB는 할당해야 본 실습이 원활하게 진행된다.

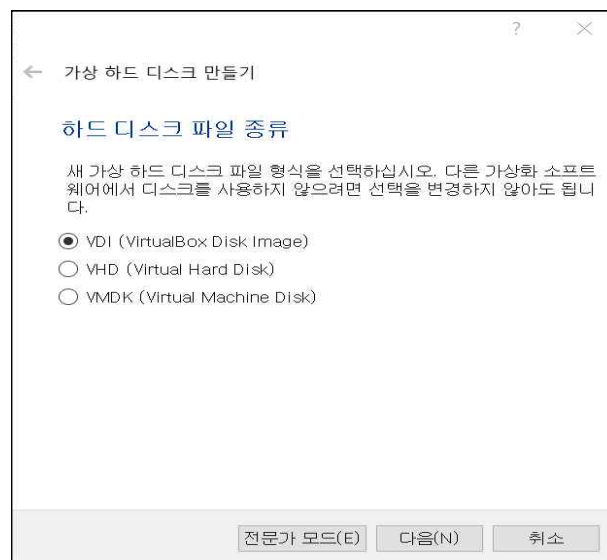


52) 자신의 PC가 몇 bit 운영체제를 사용하는지 모르겠다면 확인을 한 후 선택하도록 한다. 확인 방법은 어렵지 않지만 책에서 전부 설명하기는 분량이 많으므로 직접 인터넷에서 찾아보도록 하자.

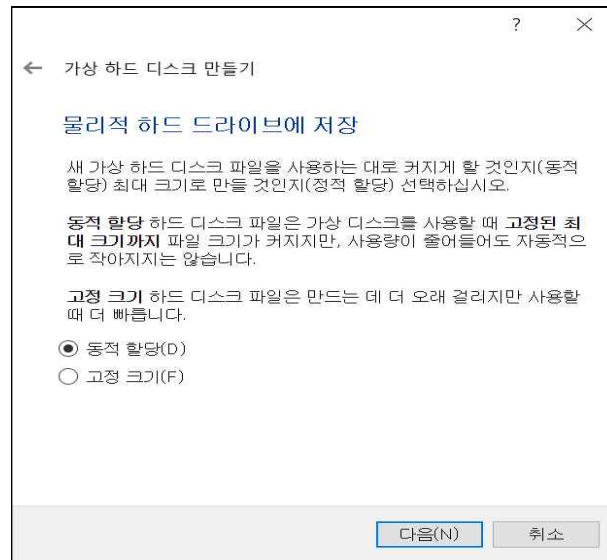
그 다음은 가상머신에서 사용할 디스크의 크기를 지정해준다. 우분투의 이미지는 3GB이고 우리 실습에서 그렇게 많은 용량을 사용하지 않으므로 기본 설정으로 진행해준다. 용량이 너무 작은 경우에도 제대로 실행되지 않으니 적당히 할당을 해주자.



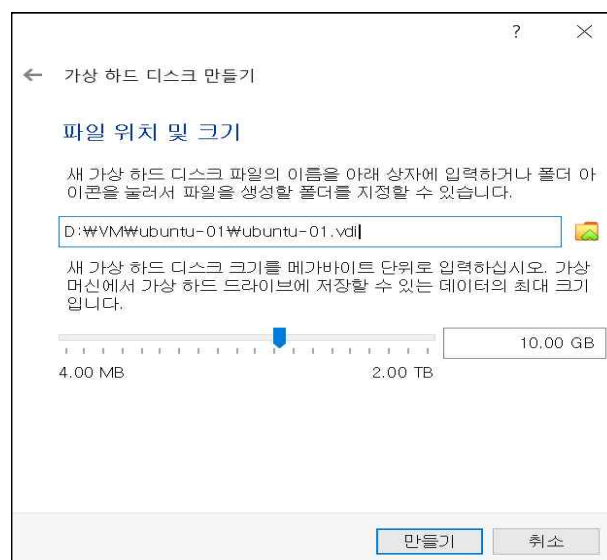
가상 하드 디스크의 종류는 가상머신 파일의 포맷을 정하는 것인데 버추얼 박스 외에도 다른 가상머신 소프트웨어에서도 이 가상머신 이미지를 사용할 때 사용하는 메뉴이다. 기본 선택인 VDI(Virtual Disk Image)에 두고 진행하도록 하자.



동적할당은 가상머신을 사용하면서 용량이 늘어날 때마다 물리 컴퓨터에서 가상머신 파일의 크기도 커지는 것이고, 고정크기는 미리 정해둔 10GB까지 만들어둔 다음 그 안에서 자유롭게 사용하는 것이다. 옛날에는 고정크기가 제공하는 속도의 이점이 분명했지만 최근에는 개인적으로 사용할 때에는 큰 체감은 되지 않는다. 기본 선택인 동적 할당으로 두고 진행한다.



마지막으로 가상머신 이미지의 저장 경로와 크기를 다시 한번 확인한다. 여기까지 진행하면 새로운 가상머신의 준비는 끝난다.

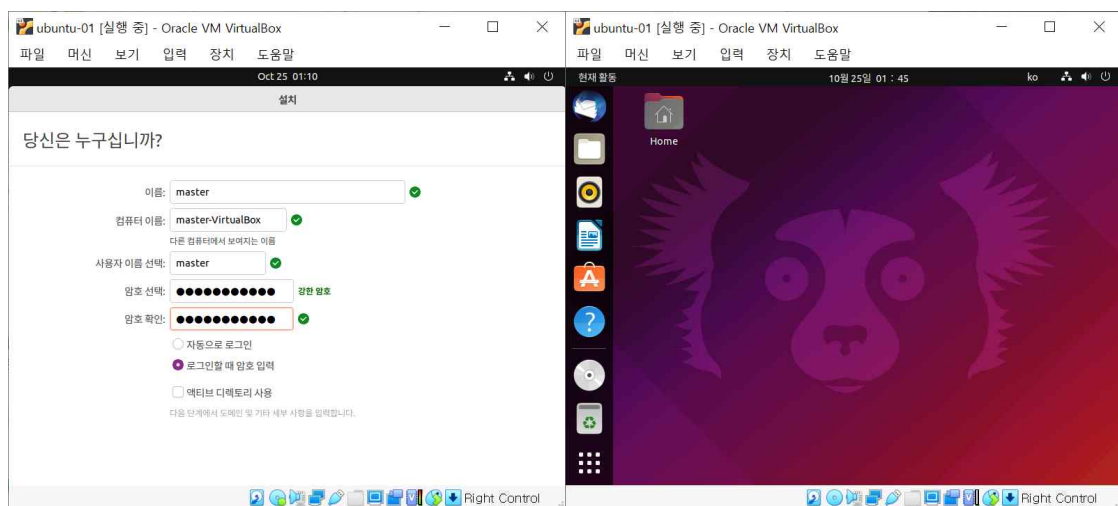
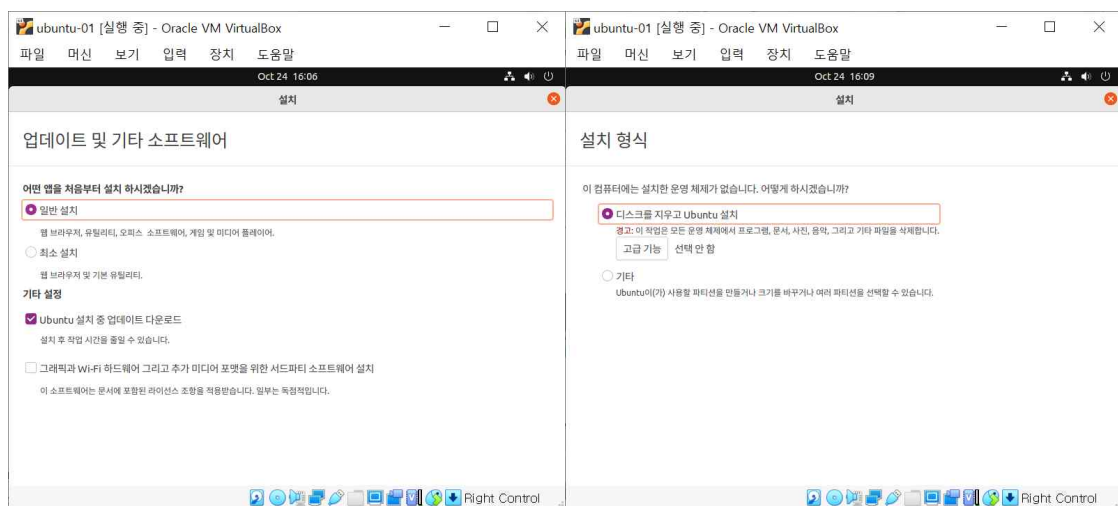
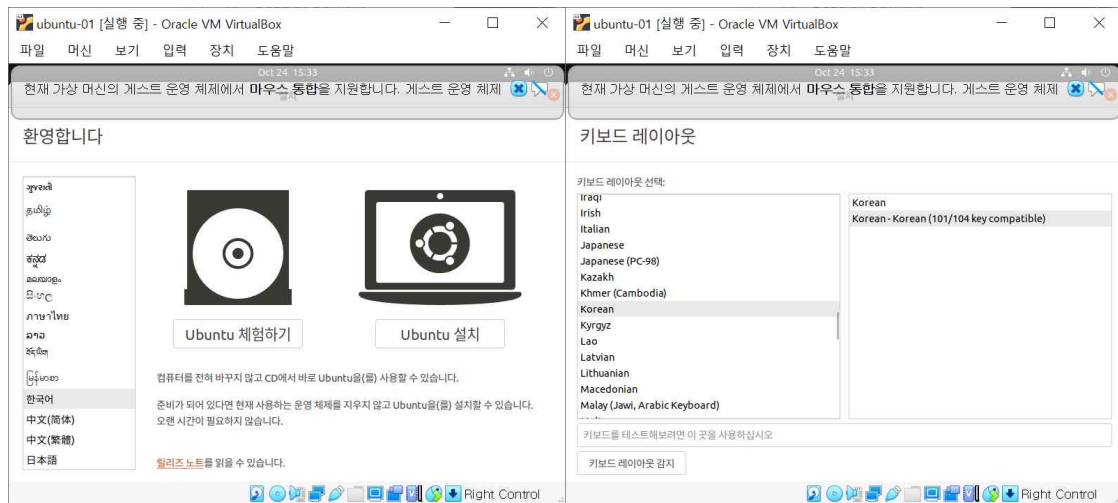


준비가 끝나면 버추얼 박스의 홈에 아까 우리가 설정한 ‘ubuntu-01’ 이라는 이름의 가상머신이 보인다. 소프트웨어로 만든 가상의 컴퓨터가 준비된 것이다. 그러나 가상머신의 이름이 ubuntu일 뿐 아직 운영체제는 설치하지 않았다. 이 가상머신에 우리가 아까 준비한 우분투를 설치해보도록 하자.



ubuntu-01을 선택했을 때 우측에 이 가상머신의 하드웨어 정보들이 나온다. 여기서 [저장소]를 보면 [IDE 세컨더리 마스터: 비어있음] 이라고 되어있을 것이다. 여기를 선택하면 그림과 같이 ‘Choose a disk file...’ 이라는 메뉴가 나온다. 이 메뉴를 선택하면 가상머신에 설치 CD를 넣은 것과 같이 된다. 이후 이 가상머신을 시작하면 우분투 설치를 시작한다.

예전의 리눅스는 설치도 복잡하고 어려워서 많은 공부를 필요로 했지만 지금은 리눅스도 설치가 매우 쉽고 편해졌다. 우분투 설치를 시작하면 한국어 설정만 제외하면 대부분은 기본 설정으로 두고 설치를 진행하면 될 정도이다. 다음은 우분투 설치시 나오는 화면들을 순서대로 정리해둔 것이니 참고만 하길 바란다.



가장 마지막에는 이 우분투를 사용할 사용자의 정보를 입력한다. 자신이 사용할 사용자 계정 이름과 암호를 입력하고 꼭 기억해두자. 설치한 버전에 따라 마지막 화면은 조금씩 다를 수 있다.

처음 리눅스를 사용하면 색다른 인터페이스에 사용이 어려운 것이 사실이다. 본 장에서는 이후의 실습을 위한 리눅스의 기초들을 다룬다. 자신이 아는 내용이면 이 장은 넘어가도 좋다.

커맨드라인 인터페이스 CLI

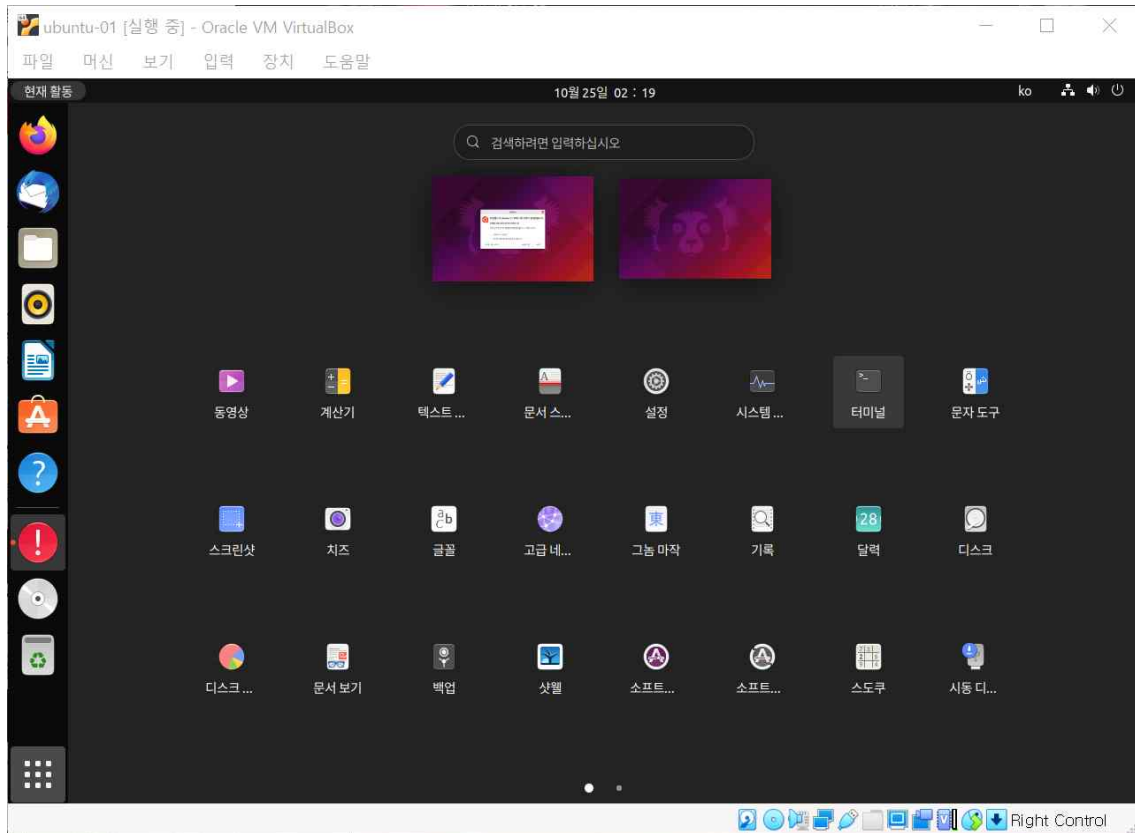
최근에는 리눅스도 GUI(Graphical User Interface)를 많이 사용하지만 기본적으로는 CLI(Command Line Interface) 환경이었다. 그러다 보니 처음 리눅스를 사용하는 사람들은 운영 체제를 설치하고 난 다음 검은 화면에 글자만 몇줄 나오니 컴퓨터를 어떻게 사용해야할지 어려워하는 경우가 많았다. CLI는 그 이름과 같이 명령어(command)를 입력하여 사용하기 때문에 리눅스 명령어를 알아야만 사용할 수 있다는 점도 진입장벽을 높이는 이유기도 하다.

우리가 설치한 우분투도 처음에는 윈도우즈와 비슷해 보인다. 본 책의 실습에서는 대부분 터미널이라고 부르는 프로그램을 통해 CLI 환경을 사용하기 때문에 기본적인 명령어를 익히도록 하겠다.

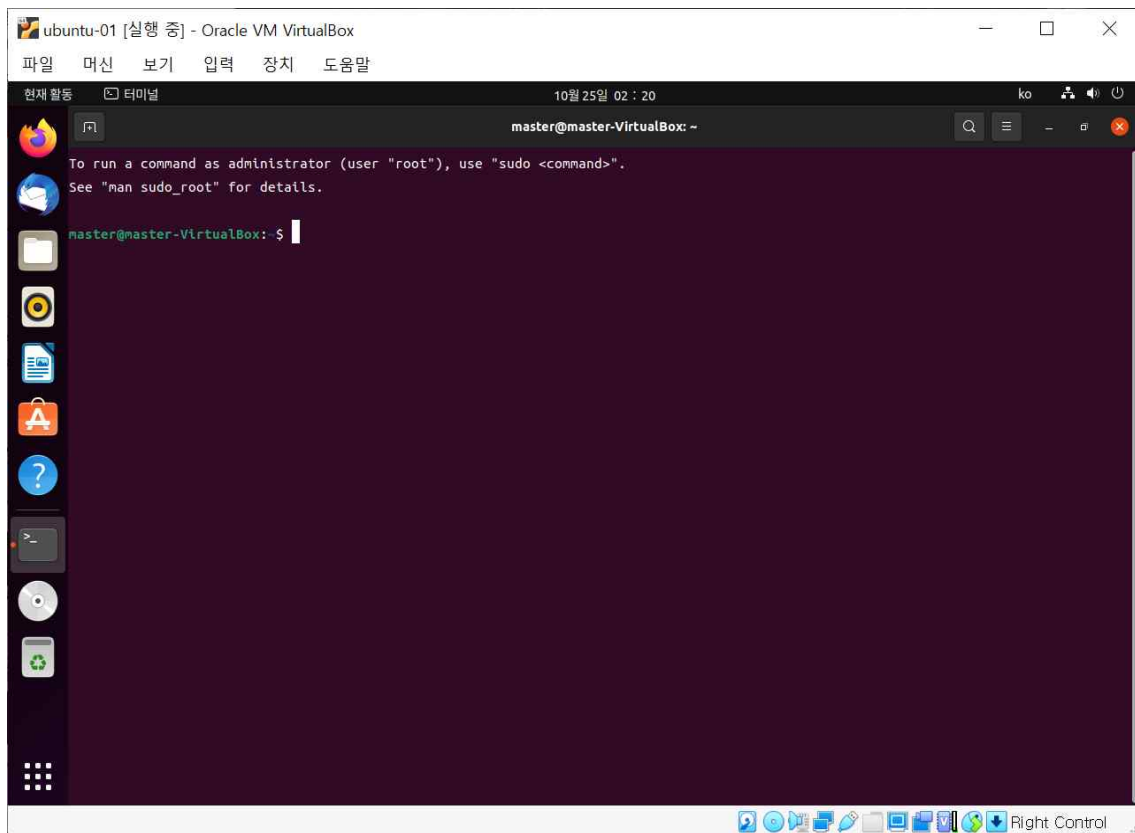
먼저 우분투의 메뉴에서 터미널을 찾아서 실행해보자. GUI에서 terminal 메뉴를 찾아서 실행하면 텍스트를 입력할 수 있는 명령창이 실행된다. 여기서 한번 몇가지 명령어를 실행해보자. 터미널 실행창에서 아래와 같이 “whoami”, “pwd”, “date” 를 한번씩 입력해본다. 이때, 쌍따옴표는 입력하지 않는다. whoami는 내가 누구인지 묻는 명령으로 사용자 계정명을 알려준다. pwd는 현재 작업 경로를 묻는 것으로 현재 경로가 /home/master라고 알려준다. date는 현재 시각을 알려준다. 아래는 명령어 실행과 그 결과이다.

```
bash
```

```
master@master-VirtualBox:~$ whoami
master
master@master-VirtualBox:~$ pwd
/home/master
master@master-VirtualBox:~$ date
2021. 05. 15. (토) 02:35:50 KST
```



우분투 GUI에서 제공하는 전체 메뉴



터미널 실행 모습

기본 명령어

리눅스의 기본 명령어에 대해서 한번 알아보자. 마우스를 이동하고 클릭과 실행으로 사용했던 윈도우즈와 달리 리눅스는 일일이 명령어를 입력해야 하므로 이 명령어들을 외우고 있어야 한다. 따라서 처음 시작하면 어렵게 느껴지기 마련이다. 그러나 실제 개발을 하게 되면 대부분의 회사에서는 리눅스를 기본으로 사용하므로 천천히 익숙해지는 것을 추천한다. 표에 있는 기본 명령어들은 리눅스 배포판이 달라도 대부분 동일하게 사용하는 명령어들이다. 표에 있는 명령어들을 직접 한번씩 실행해보기를 권한다. 사용법과 예시가 같은 경우에는 합쳐서 표기하였다.

- 디렉터리 관련 명령⁵³⁾

명령어	내용	사용법 / 예시	옵션
pwd	현재 작업 경로 출력	\$ pwd	
cd	{경로}로 디렉터리 이동	\$ cd {경로}	
		\$ cd /home/my	
ls	현재 디렉터리 내 파일 출력	\$ ls -l -a	-l, -a
mkdir	새 디렉터리 생성	\$ mkdir {디렉터리 이름}	
		\$ mkdir my_dir	
rmdir	디렉터리 삭제	\$ rmdir {디렉터리 이름}	
		\$ rmdir my_dir	

- 시스템 종료 관련 명령

명령어	내용	사용법 / 예시	옵션
shutdown	옵션에 따라 시스템 종료나 재시작	\$ shutdown	-hH, -r
halt	시스템 종료	\$ halt	
poweroff	시스템 종료	\$ poweroff	-l, -a
reboot	시스템 재시작	\$ reboot	
init	런레벨 설정. 옵션에 따라 시스템 종료 또는 재시작	\$ init	0,1,2,3 ,4,5,6
		\$ init 0	

53) 파일들을 한데 보관하고 있는 계층을 표현할 때, 윈도우즈는 폴더(folder)라고 하고 리눅스는 디렉터리(directory)라고 한다. 그리고 기술적으로도 둘은 다른 구조를 가지고 있다.

- 파일 관련 명령

명령어	내용	사용법 / 예시	옵션
touch	파일의 최근 업데이트 일자 변경	\$ touch {파일이름}	
		\$ touch hello	
cp	파일 복사	\$ cp {복사할 파일} {새 이름}	
		\$ cp orgFile cpFile	
mv	파일 이동	\$ mv {파일이름} {움길위치}	
		\$ mv hello ../my/hello	
rm	파일 삭제	\$ rm {파일이름}	-r, -f 등
		\$ rm myFile	
file	파일 타입 확인	\$ file {파일이름}	
		\$ file myFile	
find	조건에 맞는 특정 파일 찾기	\$ find {찾을위치} {찾을이름}	
		\$ find /home "myfile"	

rm 명령어는 리눅스에서 가장 주의해야할 명령어 중 하나이다. 실제로 큰 회사에서도 rm 명령어를 잘못 사용하여 고객정보를 전부 날려버려 뉴스에도 나온 경우가 한두번이 아니다.

- 출력 관련 명령

명령어	내용	사용법 / 예시	옵션
echo	명령어 다음 내용을 출력	\$ echo {내용}	
		\$ echo hello	
cat	파일 내용 출력	\$ cat {파일이름}	
		\$ cat myFile	
more	출력 결과를 나눠서 출력	\$ more {파일명}	
		\$ more myFile	
grep	파일에서 해당 내용을 찾아서 출력	\$ grep {찾을내용} {파일이름}	
		\$ grep "hello" myFile	
>, >>	리다이렉션, 덮어쓰기 / 추가	\$ {출력 명령어} > {파일명}	
		\$ ls > myFile	
man	매뉴얼 출력	\$ man {명령어 이름}	
		\$ man echo	
clear	터미널 화면 정리	\$ clear	

리눅스 디렉터리 구조

윈도우즈에서는 [내컴퓨터] 아래에 [C:/]나 [D:/] 같이 디스크별로 볼륨명이 붙고, 그 다음 폴더들이 위치한다. 윈도우즈에서 설치하는 프로그램 중 대부분은 [Program Files] 같은 폴더에 설치된다.

리눅스도 이와 같이 기본적인 디렉터리 구조를 가지는데, 리눅스는 디렉터리마다 정해진 역할들이 있다. 처음에는 리눅스에서 무언가를 찾으려고 할 때 이 디렉터리 구조를 몰라서 내용을 찾기 힘들어 한다. 리눅스는 파일시스템 계층구조의 표준(Filesystem Hierachy Standard, FHS)을 따르고 있다. 따라서 이후에라도 리눅스에서 무언가를 설치하거나 정보를 찾을 때, 이 표준을 참고하여 찾으면 원하는 결과를 빠르게 찾을 수 있을 것이다.

디렉터리	설명
/	기본 계층 모든 파일 시스템 계층의 기본인 루트 디렉토리
/bin	모든 사용자를 위해 단일 사용자 모드에서 사용 가능해야 하는 명령어 바이너리
/boot	부트 로더파일
/dev	필요한 장치
/etc	설정파일, 바이너리는 안됨
/etc/opt	/opt/에 대한 설정 파일
/etc/X11	X 윈도 시스템의 설정 파일, 버전 11
/etc/sgml	SGML 설정 파일
/etc/xml	XML 설정 파일
/home	저장된 파일, 개인 설정, 기타 등을 포함한 사용자의 홈 디렉토리
/lib	/bin/과 /sbin/에 있는 바이너리에 필요한 라이브러리
/media	CD-ROM과 같은 이동식 미디어의 마운트 지점
/mnt	임시로 마운트된파일 시스템
/opt	선택 가능한 응용 소프트웨어 패키지
/proc	커널과 프로세스 상태를 문서화한 가상 파일 시스템
/root	루트 사용자의 홈 디렉토리
/sbin	필수 시스템 바이너리
/srv	시스템에서 제공되는 사이트 특정 데이터.
/tmp	임시 파일

디렉터리	설명
/usr	읽기 전용 사용자 데이터가 있는 보조 계층 구조.
/usr/bin	모든 사용자의(단일 사용자 모드에서 필요하지 않은) 비 필수 명령어 바이너리
/usr/include	표준 include 파일
/usr/lib	/usr/bin/과 /usr/sbin/에 있는 바이너리를 위한 라이브러리
/usr/sbin	비 필수 시스템 바이너리
/usr/share	아키텍처에 독립적인(공유) 데이터
/usr/src	소스 코드
/usr/X11R6	X 윈도 시스템
/usr/local	로컬 데이터의 3차 계층
/var	변하기 쉬운 파일
/var/cache	애플리케이션 캐시 데이터. 이런 데이터는 시간이 걸리는 입출력이나 계산의 결과로 로컬에서 발생
/var/lib	상태 정보. 프로그램의 실행 중에 수정되는 영구적인 데이터
/var/lock	잠금 파일. 현재 사용중인 자원을 추적하는 파일
/var/log	로그 파일
/var/mail	사용자의 사서함
/var/run	마지막 부트 때부터 작동하는 시스템에 대한 정보
/var/spool	처리를 기다리는 작업 스푼
/var/spool/mail	예전에 사용했으나 현재에는 사용되지 않는 사용자 사서함 위치
/var/tmp	재부팅 해도 보존되는 임시 파일.

특히 리눅스에서 여러 소프트웨어를 설치하고 설정을 바꾸거나 할 때, 소프트웨어가 어디에 설치되어있는지 모르는 경우가 많다. 항상 그런 것은 아니지만 소프트웨어는 /var 아래에 설치되는 경우가 많고 설정 파일은 /etc 아래에 많다. 특정 소프트웨어를 찾고 싶을 때 \$ where {소프트웨어 이름}으로 검색하면 나오는 경우도 있다.

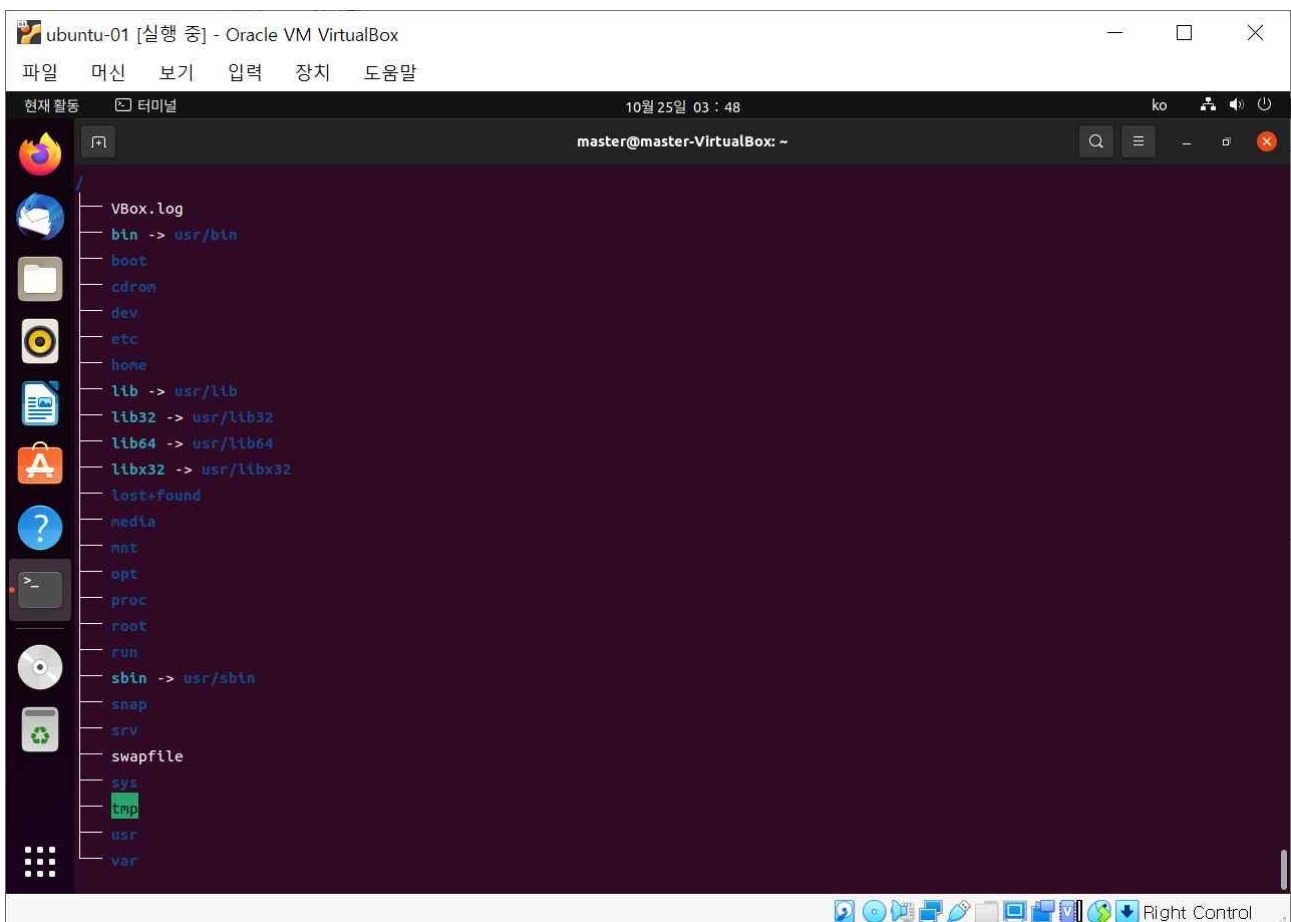
리눅스 패키지 설치

우분투에서는 소프트웨어를 설치할 때 APT(Advanced Packaging Tool)라는 도구를 주로 사용한다. 우분투가 속한 데비안 계열 리눅스에서 주로 사용하는 소프트웨어 관리 및 설치 도구이다. 내가 사용하고 싶은 소프트웨어가 있다면 APT 도구를 사용해서 쉽게 설치하고 사용할 수 있다. 한번 아래와 같이 tree라는 소프트웨어를 설치해보자. sudo라는 명령어는 관리자 권한을 위한 것으로 시스템이 변경되는 명령어를 실행할 경우 필요한 명령어이다.

```
bash
```

```
master@master-VirtualBox:~$ sudo apt install tree
```

```
master@master-VirtualBox:~$ tree / -L 1
```



apt 명령어로 설치한 tree 명령어의 실행 화면

위와 같이 tree 명령어를 실행하면 루트 경로부터 깊이 1만큼의 디렉터리만 탐색하여 출력해준다. 앞 장의 리눅스 디렉터리 구조와 비교해봐도 좋겠다. 이와 같이 apt라는 명령어를 통해서 tree라는 소프트웨어를 설치해서 곧바로 사용할 수 있는 것을 확인할 수 있다.

위와 같이 APT는 apt라는 명령어도 사용할 순 있지만 보통 apt-get이라는 명령어로 많이 활용한다. 발음은 [에이피티 겿]이나 [애프트 겿]이라고 부른다.

- apt-get 명령

명령어	내용
sudo apt-get update	apt 저장소의 최신정보를 받아온다.
sudo apt-get upgrade	apt로 설치된 패키지들을 apt 저장소의 최신버전으로 업그레이드 한다.
sudo apt-get upgrade {package name}	특정 패키지만 최신버전으로 업그레이드한다.
sudo apt-get install {package name}	apt 저장소에 있는 패키지라면 다운로드 받아서 자동으로 설치한다.
sudo apt-get remove {package name}	특정 패키지를 삭제한다.
sudo apt-get purge {package name}	특정 패키지와 관련된 설정 파일들도 전부 삭제한다.
sudo apt-get autoremove	사용되지 않는 패키지들을 삭제한다.

처음 apt-get 명령어를 사용하면 익숙하지 않을 수 있다. 그리고 편리함과 동시에 apt 저장소에 대체 얼마나 많은 패키지가 있는지 궁금할 수도 있다. 우리가 리눅스에서 사용하려는 소프트웨어들은 대부분 apt 저장소에 있다고 해도 과언이 아니다.⁵⁴⁾

우리는 윈도우즈를 사용하면서 소프트웨어를 설치할 때, 대부분 소프트웨어를 제공하는 웹사이트를 찾아가서 파일을 다운로드 받고 설치하는 것에 익숙해있다. 그러나 리눅스는 대부분 패키지 매니저를 통해 내가 설치하려는 패키지의 이름만 입력하여 설치를 한다. 익숙하지 않고 해도 우리는 이와 같은 방식을 이미 사용하고 있다. 바로 휴대폰의 앱 마켓이다.

앱 마켓은 구글이나 애플이 관리하는 많은 앱들을 쉽게 찾을 수 있고 바로 다운로드 받아서 설치하고 사용한다. 패키지 매니저가 바로 이와 같은 기능을 제공한다고 보면 된다. 아마 패키지 관리도구에 익숙해지면 윈도우즈에서 파일을 검색해서 설치하는 방식은 귀찮아질지 모른다.

54) 데비안 계열은 apt를 사용하고 레드햇은 자체 패키지 매니저인 RPM(Redhat Packaging Manager)을 사용한다. 그 외에도 yum이나 dpkg 등 많은 패키지 매니저와 저장소가 존재한다.

리눅스 텍스트 에디터

리눅스의 대표적인 텍스트 에디터는 vi라는 도구이다. 아직 CLI 기반의 사용환경일 때 vi는 대단히 강력한 텍스트 에디터 도구였다. 그러나 마우스가 컴퓨터의 기본 입력 인터페이스가 되고 텍스트 에디터도 많은 기능을 제공하면서 일반 사용자들은 GUI 기반의 에디터를 더 많이 사용하게 되었다. 그러나 여전히 vi는 개발자들이 기본적으로 알아야 하는 에디터로 인식되고 있다. 그 이유는 실제 개발 회사에서는 서버 환경으로 유닉스나 리눅스를 아직까지 많이 사용하고 있는데, 이 서버들은 보통 GUI를 쓰지 않는다. 즉, 명령어로만 작업을 처리해야하므로 이 서버에서의 작업은 vi가 필수라고 할 수 있다. 비록 책에서 vi의 사용법과 중요성을 다 전달하기는 힘들지만 독자들이 개별적으로 vi는 직접 사용해보면서 많이 익숙해지길 바란다. vi의 사용법을 아는 것만으로도 나중에 회사생활을 시작할 때 엄청 많은 시간을 절약시켜 줄 것이다. vi의 사용이 어렵다면 우분투 GUI의 메뉴에 있는 텍스트 에디터를 사용하길 바란다.

Note : vi 학습을 위한 vim-adventures

VIM은 Vi IMproved의 약자로 향상된 vi를 의미한다. 현재 리눅스도 vi를 사용하더라도 vim 기능을 사용할 수 있다. 그러나 vi는 사용방법이 어려워 처음에 익숙해지려면 상당히 많은 시간이 필요한데, 이를 게임으로 만들어서 학습을 도와주는 사이트가 있어서 소개한다.

VIM Adventures는 캐릭터를 이동하여 목표를 달성하면서 vi의 기능을 자연스럽게 익힐 수 있도록 도와주는 사이트이다. 주소는 <http://vim-adventures.com>으로 아직 vi가 익숙하지 않은 사람이라면 접속해서 한번 게임을 즐겨보길 바란다.



3-4

리눅스 네트워크 설정

우분투 네트워크 설정

실제 개발환경과 유사하게 만들기 위해 가상머신을 총 3대를 만들겠다.⁵⁵⁾ 방금 만든 ubuntu-01에 이어 ubuntu-02와 ubuntu-03도 만들어주자. 가상머신의 장점을 확인해보고 싶다면 처음부터 설치하는 것이 아니라 기존 가상머신을 복제하는 방법도 연습해보길 바란다. 이는 독자가 직접 해보는 것을 권한다.



3개의 가상머신을 설치한 직후의 모습

가상머신끼리 네트워크 연결을 하려면 가상머신 내의 리눅스에도 몇가지 유틸리티 소프트웨어를 설치해야하고 가상머신의 설정도 변경을 해야한다.

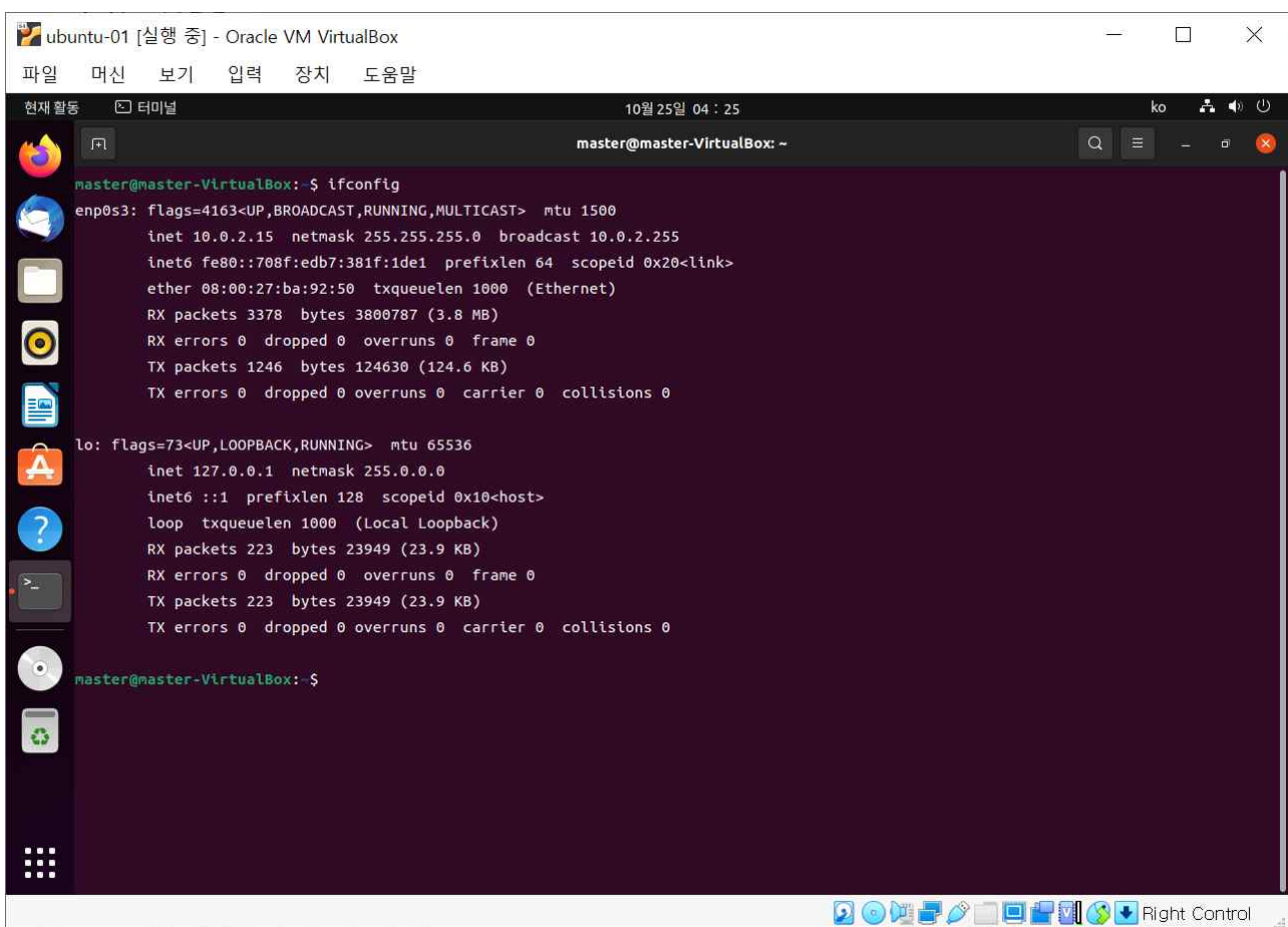
55) 가상머신의 자원 소모가 크므로 자신의 실습 환경에 따라 적절히 2대만 만들어도 좋다.

ubuntu-01에다가 아래와 같은 소프트웨어를 설치하자. apt-get을 사용하면 된다. 그 후 ifconfig 명령어를 실행해보면 현재 ubuntu-01의 네트워크 상태가 나온다. 이 네트워크 상태는 각자가 다 다르므로 아래 설명과 자신의 네트워크 상태를 비교해보면서 보길 바란다.

```
bash
```

```
master@master-VirtualBox:~$ sudo apt-get install net-tools
```

```
master@master-VirtualBox:~$ ifconfig
```



```
master@master-VirtualBox:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.0.2.15 netmask 255.255.255.0 broadcast 10.0.2.255
    inet6 fe80::708f:edb7:381f:1de1 prefixlen 64 scopeid 0x20<link>
    ether 08:00:27:ba:92:50 txqueuelen 1000 (Ethernet)
    RX packets 3378 bytes 3800787 (3.8 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1246 bytes 124630 (124.6 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 223 bytes 23949 (23.9 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 223 bytes 23949 (23.9 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

master@master-VirtualBox:~$
```

위의 네트워크 상태를 보면 enp0s3 인터페이스에 inet 정보가 10.0.2.15인 것을 볼 수 있다. 이것이 쉽게 말해서 가상머신의 IP 주소이다. 아마 처음 우분투를 설치하고 별다른 설정을 하지 않았다면 자동으로 할당되어 있을 것이다. 이후에 이 IP 주소를 우리가 직접 바꿔야 한다.

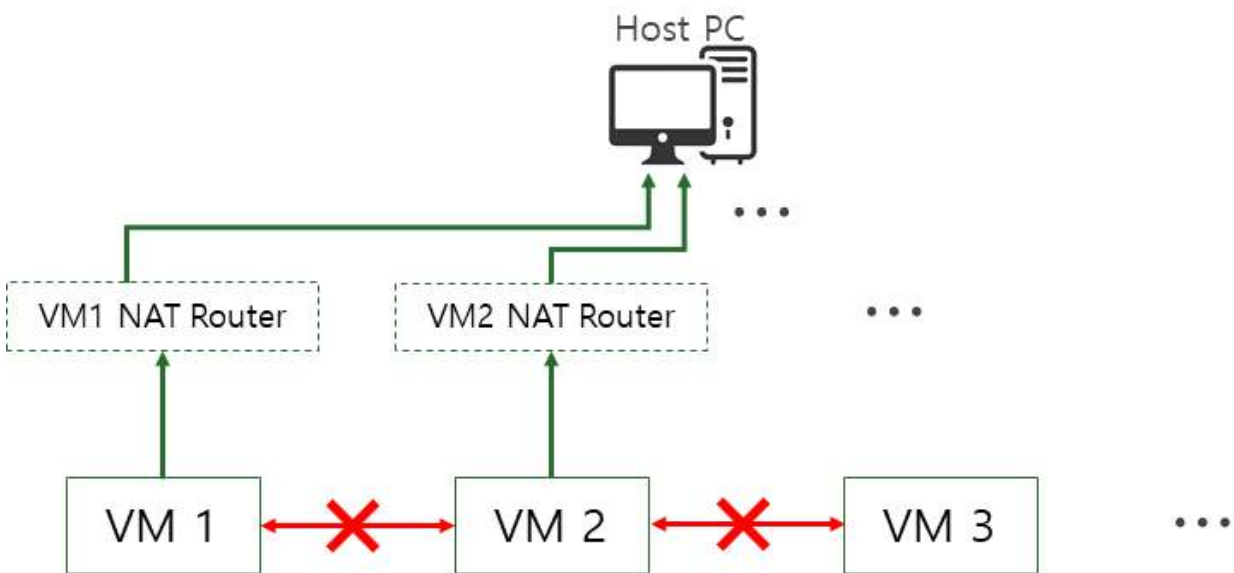
가상머신 네트워크

가상머신의 네트워크는 가상화된 네트워크를 사용하여 Host PC와 Guest PC를 연결하는 것이다. 가상머신이 소프트웨어로 하드웨어를 구현한 것인 만큼 네트워크를 위해 필요한 네트워크 장비도 가상화되어 구현되어 있다. 그래서 우리가 실제로 사용하는 물리 컴퓨터가 가상머신들을 연결해주는 네트워크 장비처럼 취급이 된다. 이 가상 네트워크는 여러 모드를 가지고 있는데 하나씩 소개를 해본다.

- NAT
- NAT Network
- Bridged Adapter
- Internal Network
- Host-Only Adapter

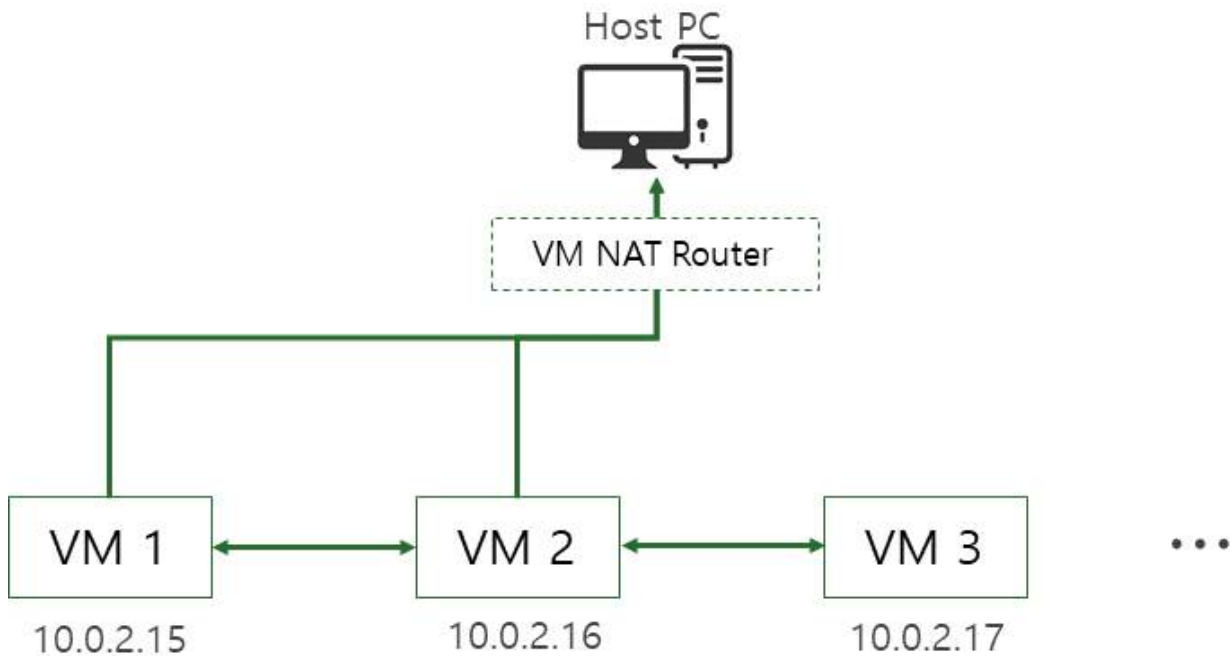
NAT(네트워크 주소 변환)

NAT는 사설 네트워크(private network)에 속한 여러 PC들이 하나의 공인IP 주소를 사용하여 인터넷을 하기 위한 용도이다. 이를 각 가상머신마다 NAT Router를 두는 방식으로 통신하는데, Router가 다른 가상머신에 대해서 알 수 없으므로 가상머신 간의 네트워크는 안된다.



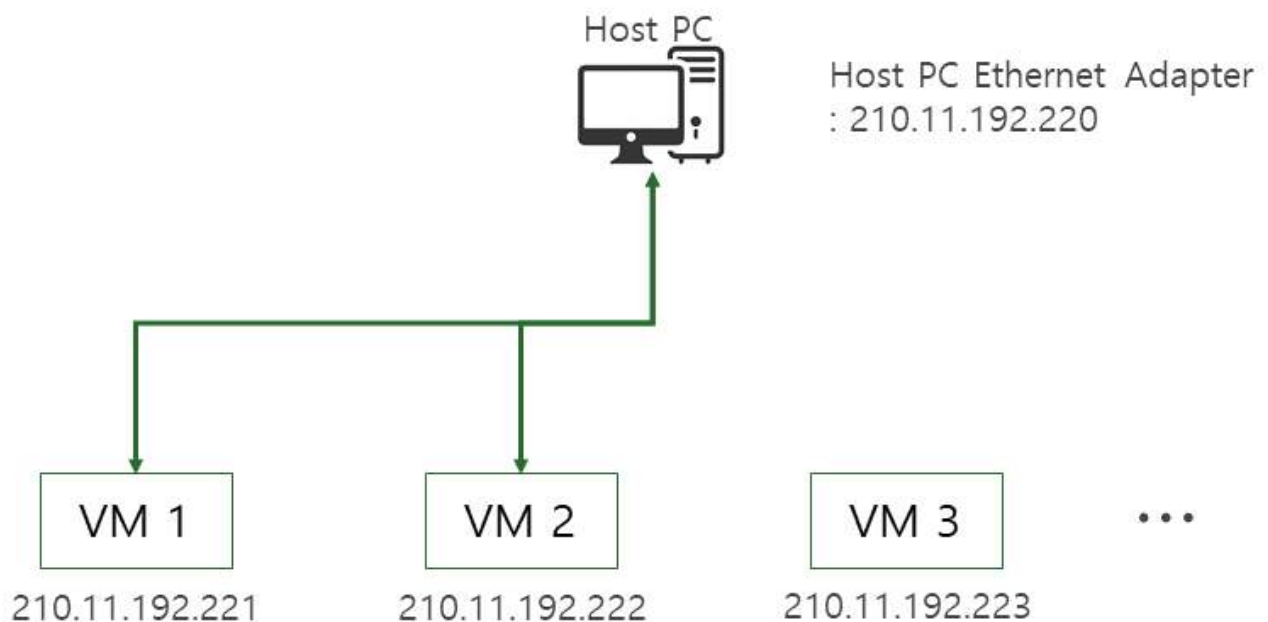
NAT Network

NAT Network는 NAT에서 Router의 위치를 바꾸어 가상머신 간의 통신도 가능하게 한 것이다.



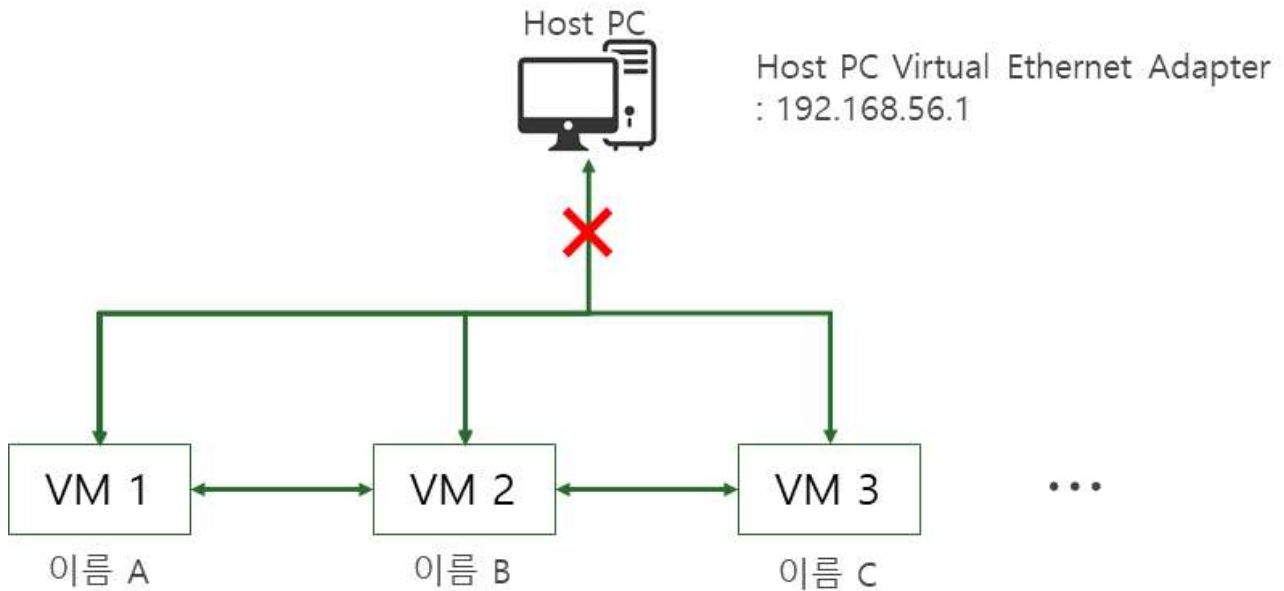
Bridged Adapter

브릿지 방식은 Host PC를 브릿지로 삼아 가상머신이 호스트 PC와 동등한 수준의 네트워크를 가능하게 한다. 각 게스트 PC마다 호스트 PC와 동등한 수준의 실제 IP를 할당해서 사용한다.



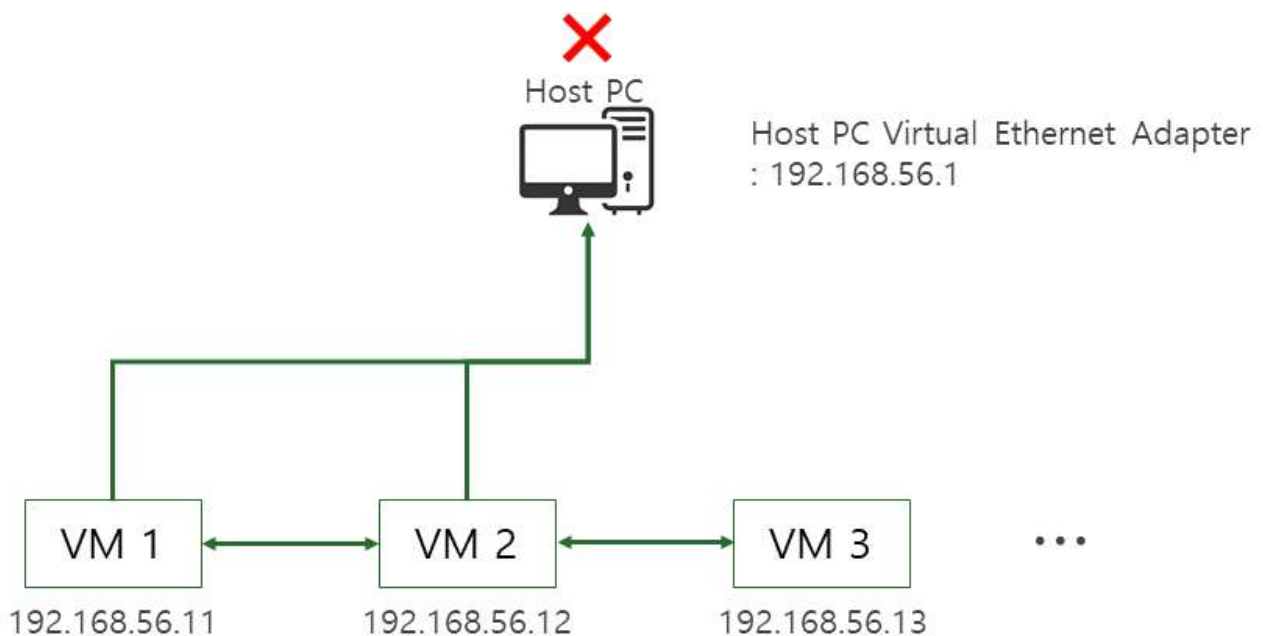
Internal Network

호스트 PC와 독립적으로 내부 네트워크를 만든다. 이 때 IP주소는 별로 중요하지 않으며 이름으로 가상머신 간의 통신이 가능하다.



Host-Only Adapter

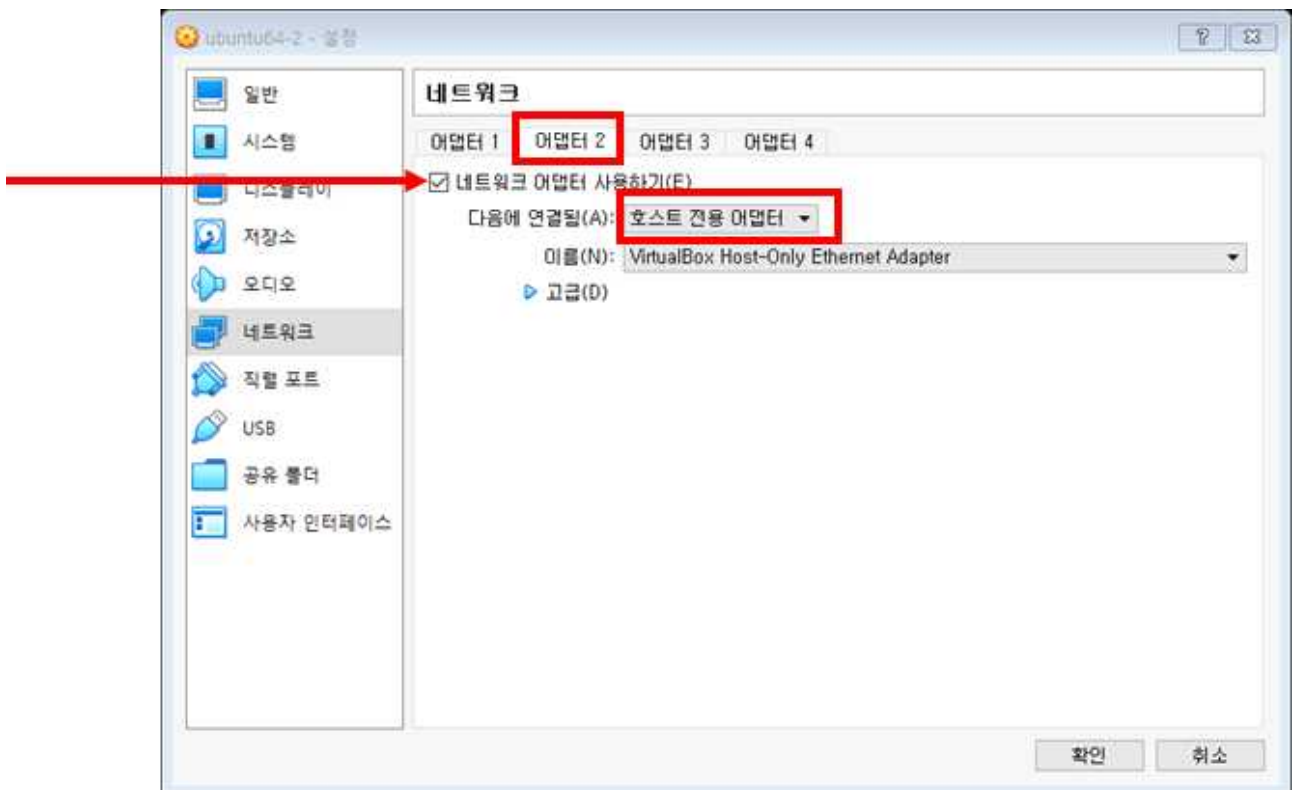
외부 네트워크와는 단절되지만 Host PC를 브릿지처럼 사용하여 내부 통신을 가능하게 한다.



가상머신 네트워크 구성

가상머신 네트워크는 위에서 설명한 네트워크 모드 중에 하나를 선택해서 필요한 경우 맞게 사용한다. 여기서는 호스트 전용 어댑터를 사용하는 방법을 살펴보고 이후에는 자동화 도구를 통해 더욱 편하게 네트워크 설정을 할 것이다.

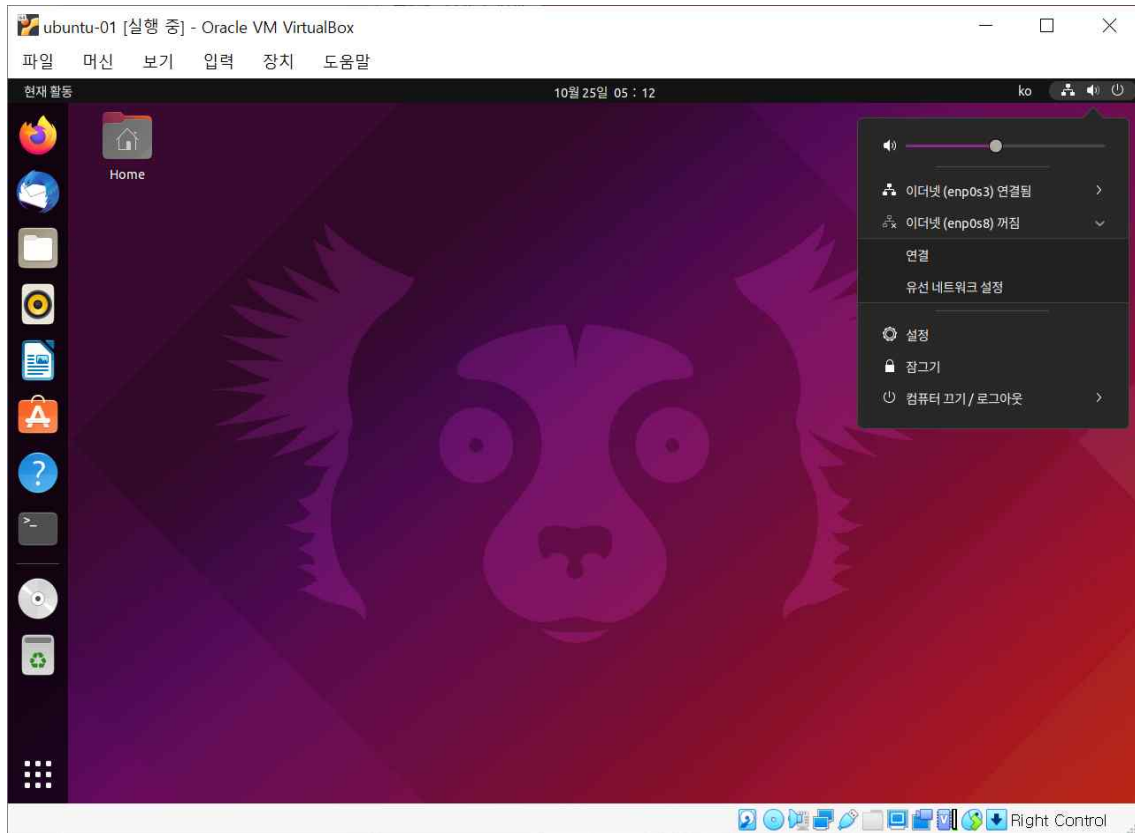
먼저 버추얼 박스의 설정을 변경해야 한다. 버추얼 박스의 가상머신 목록에서 설정을 누르면 가상머신의 하드웨어 설정을 변경할 수 있다. 여기서 네트워크 설정을 보자.



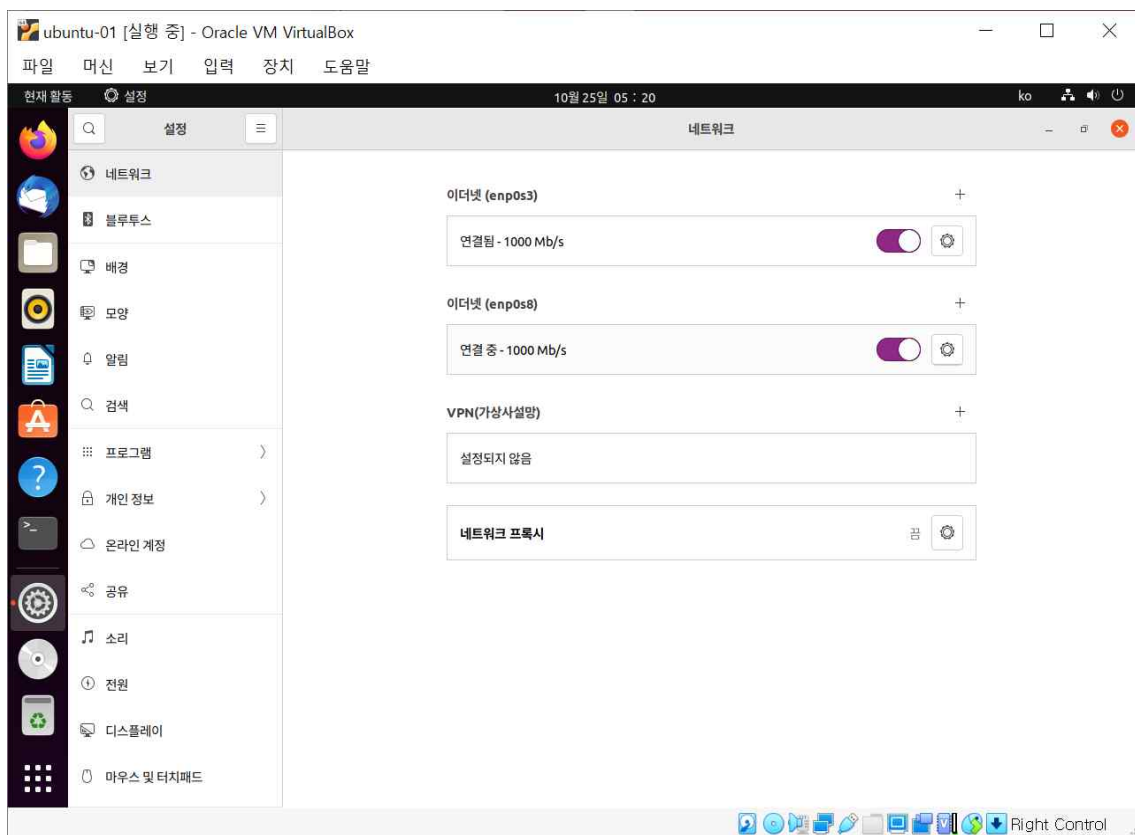
어댑터 1은 기존의 가상머신이 사용하는 설정이다. 가상머신은 이 어댑터를 통해서 인터넷을 사용하고 있다. 우리는 여기서 어댑터 2를 새로 추가하여 가상머신끼리 통신하는 네트워크를 구성할 것이다. 이 설정은 가상머신을 종료한 후에 설정해야한다.

[어댑터 2 선택] -> [네트워크 어댑터 사용하기] -> [호스트 전용 어댑터 선택]

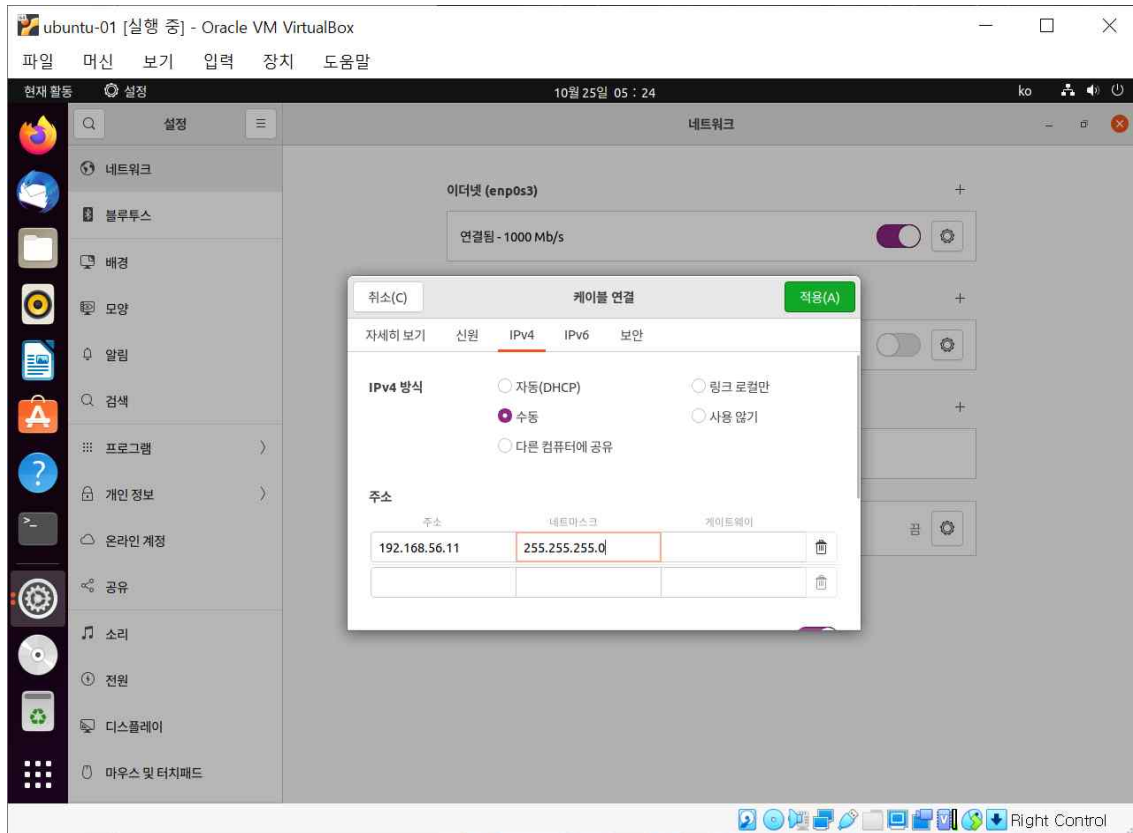
가상머신의 설정이 끝났으면 이제 리눅스에서 추가한 어댑터의 IP 주소를 변경할 차례이다. 가상머신을 다시 켜서 리눅스의 네트워크를 보면 아까와 다르게 enp0s8이라는 인터페이스가 하나 더 추가된 것을 알 수 있다. 이 추가된 인터페이스에서 유선네트워크 설정을 들어가자. 여기서 IP 주소를 물리 컴퓨터의 가상 네트워크 주소와 같은 대역으로 바꿔주면 된다. 여기서는 192.168.56.11로 주소를 설정한다.



추가한 어댑터 enp0s8의 유선네트워크 설정



enp0s8의 톱니바퀴 모양을 눌러서 설정으로 들어가자



IPv4 방식의 설정을 수동으로 바꿔주고 주소와 넷마스크를 각각 위 그림과 같이 바꾸어 준다. 이와 같은 방식으로 ubuntu-02의 주소는 192.168.56.12, ubuntu-03의 주소는 192.168.56.13으로 바꾸어주면 된다.⁵⁶⁾

구분	IP주소
물리 컴퓨터	192.168.56.1 (가상 router 개념)
ubuntu-01	192.168.56.11
ubuntu-02	192.168.56.12
ubuntu-03	192.168.56.13

이제 각 가상머신들은 위의 표와 같은 IP 주소를 갖게 되었고 서로 통신할 때에는 위 IP 주소로 통신하면 된다. 이후에는 이 네트워크 설정을 사용해 서버와 클라이언트 시스템을 구성하고 테스트 할 것이다.

56) 만약 ubuntu-01 가상머신에서 위와 같은 설정들을 다 한 후에 ubuntu-02와 ubuntu-03을 복제해서 생성했다면 MAC 주소의 복사에 주의해야 한다. 컴퓨터 네트워크에서 통신을 하는 방법으로 네트워크의 고유한 주소를 사용하는데 이를 MAC 주소라고 부른다. 만약 설정을 다 한 후에 가상머신을 복사하면서 MAC 주소까지 복사를 했다고 하면 정상적인 네트워크가 안 될 수도 있다.