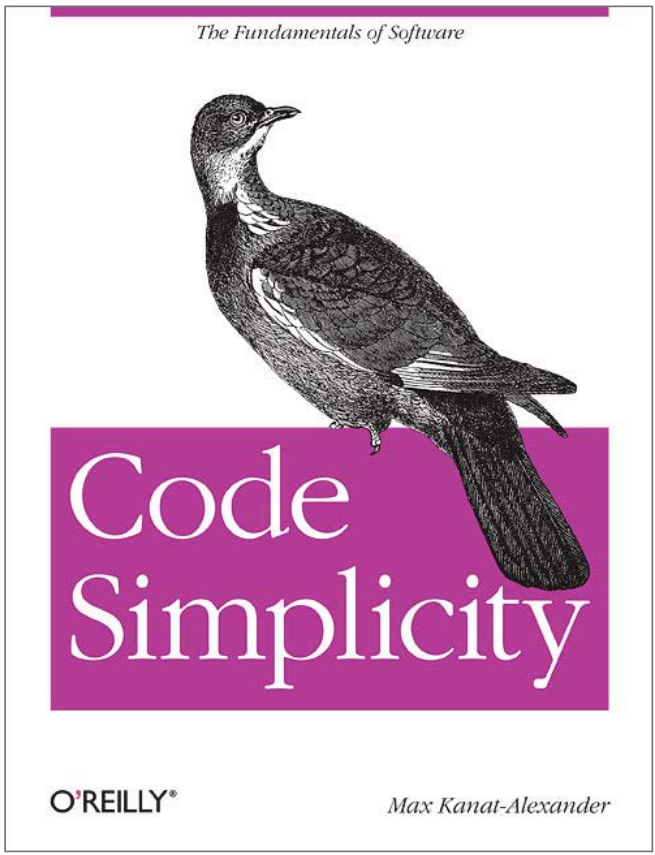

Hanbit eBook
Realtime 10
소프트웨어 생명 연장을 위한 원칙
Code
Simplicity
맥스 카넷-알렉산더 지음 / 신정안 옮김
O'REILLY®
한빛미디어
Hanbit Media, Inc.



이 도서는 O'REILLY의
Code Simplicity의
번역서입니다.

소프트웨어 생명 연장을 위한 원칙

Code Simplicity

지은이_**맥스 카넬-알렉산더**
맥스 카넬-알렉산더는 오픈소스 프로젝트인 버그질라의 수석 아키텍트이자 구글 소프트웨어 엔지니어다. 8살 때부터 컴퓨터를 고쳤으며, 14살에는 소프트웨어를 작성하였다. 현재 그는 http://www.fedorafaq.org를 운영한다.

옮긴이_**신정안**
신정안은 기타 치는 프로그래머다. SK C&C에서 근무하며, 자사 프레임워크인 NEXCORE J2EE Framework 기술 확산 및 보안 플랫폼 개발 업무를 맡고 있다.

소프트웨어 생명 연장을 위한 원칙 **Code Simplicity**

초판발행 2012년 12월 3일

지은이 맥스 카넷-알렉산더 / **옮긴이** 신정안 / **펴낸이** 김태헌
펴낸곳 한빛미디어 (주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-7914-991-3 15560 / **정가** 9,900원

책임편집 배용석 / **기획** 김창수 / **편집** 김창수, 이순옥
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 이순옥
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 박주훈, 정민하

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanb.co.kr / **이메일** ask@hanb.co.kr

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanb.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 서문

오래전에 엉망인 코드가 많은 것으로 유명한 오픈 소스 프로젝트인 '버그질라'에 프로그래머로 참여할 기회가 있었다. 시스템이 너무 복잡하다 보니 소프트웨어 개발에 있어 '회귀 불능 지점'으로 판단되는 지경까지 이르게 되었다. 코드를 수정하기가 너무 어려워서 새로운 기능을 추가하는 일은 진행이 매우 더뎠다. 프로그래머 대부분은 불만을 품은 채 백기를 들고 프로젝트를 떠났다. 자원 봉사자였던 이들이 굳이 나쁜 코드를 고치면서 고생할 필요가 없었던 것이다.

이런 상황을 약 6년 동안 버텨낸 덕에 버그질라의 사용자는 수백만으로 늘어났다. 버그질라는 오픈 소스의 주요 프로젝트 중 하나가 됐고, 거의 모든 주요 오픈 소스 프로젝트가 버그질라를 버그 트래킹 시스템으로 사용했다. 파이어폭스를 만든 모질라 등 몇몇 회사는 회사 내의 모든 소소한 업무 하나하나를 추적하기 위해서 사용하기도 했다. 버그질라가 하나의 프로젝트로서 종료되었다면 작게는 오픈 소스 개발, 크게는 소프트웨어 산업 전체에 큰 타격을 입혔을 것이다.

버그질라가 살아남아야 하는 이유는 분명했다. 그러나 어떻게 살아남을 수 있었을까? 일반적으로 소프트웨어 개발의 생명주기에서는 이런 상황이면 소프트웨어를 다시 만들라고 한다. 그러나 이미 프로그래머가 많이 빠져버려서 다시 개발할 인력이 없었다. 코드의 유지보수도 버거운 상태였다.

그래서 일정 부분은 필요에 의해서, 그리고 무엇보다 전체 시스템을 버리고 똑같은 시스템을 다시 작성하는 것을 싫어하는 이상주의로 인해 시스템을 새로 개발하지 않고 코드의 버그를 수정하자는 운동에 참여하였다. 나와 함께 소규모 프로그래머 그룹은 시스템의 아키텍처를 조금씩 재구성했고, 몇 달마다 조금씩 개선된 새로운 버전을 내놓았다. 이 작업을 하면서 새로운 기능도 추가하였으며, 코드도 엉망인 상태에서 벗어나 점점 더 좋아지기 시작했다.

이 방식은 통했다! 우리가 해낸 것이다! 엉망이던 프로젝트의 코드를 3년 동안 수정해, 초기에 참여한 프로그래머의 1/4밖에 안 되는 인력으로 2배 많은 기능을 추가했다. 파트타임 자원 봉사자 팀으로 예산과 마케팅도 없이, 수많은 프로그래머를 보유하고 있고 매출이 수백만 달러나 되는 경쟁자를 제치고 이 분야의 상위 제품으로 현재까지도 남아있다.

어떻게 이런 일을 해낼 수 있었을까? 나는 버그질라 프로젝트에 참여하기 전 수년 동안, 단순히 프로그래머를 관리하는 새로운 방법이 아니라 소프트웨어 개발 철학의 기초를 만들고 있었다. 이 근본 원리는 모든 프로젝트 및 프로그래밍 언어에 적용할 수 있는 것으로, 프로그래머가 처한 어떤 상황도 풀어나갈 수 있는 보편적 원리였다. 내가 볼 때 문제는 소프트웨어 세계에 대한 의견만 있을 뿐 충분한 사실fact이 없다는 것이다. 소프트웨어 개발에 있어 가장 일반적이고 기본이 되는 사실이 무엇인지 알아낸다면, 다른 많은 문제는 자연스럽게 해결될 것이다. 버그질라 프로젝트는 이런 사실을 알아내는 첫 시험대였고, 이를 통해 많은 사실을 밝혀낼 수 있었다. 또한, 이 덕분에 믿기 어려울 정도로 코드의 질이 향상되었고 성공할 수 있었다.

물론 테스트가 성공적이었다고 해도 겨우 하나의 프로젝트에서 테스트해본 것으로 어떤 생각이 사실이라고 말하기는 어렵다. 그래서 이 사실에 관한 좋은 아이디어가 떠오르면 개인적인 프로젝트를 새로 만들고 이들이 만들어 내는 차이점이 무엇인지, 잘 적용되는지를 확인했다. 그런 후 조직이 처한 상황과 소프트웨어의 개발 진행 과정에 대해 프로그래머들과 인터뷰도 하였다. 내가 밝혀낸 사실과 상반되는 예가 있는지 알고 싶었던 것인데, 결국 발견하지 못했다. 그 반대로 알아낸 사실을 이용하면 프로그래머의 얘기를 다 듣지 않아도 거의 모든 소프트웨어 개발 이야기의 마지막을 쉽게 예측할 수 있다는 점을 알게 되었다.

그러나 이것만으로는 충분하지 않았다. 그래서 여러 프로젝트에 적용해보고, 그 프로젝트에서도 버그질라와 같은 결과가 나오는지를 확인하였다. 프로그래머의 경험에서 반례를 찾아낼 수 있는지 확인하려고 수천 명의 프로그래머에게 이 아이디어를 전달했지만, 그 어떤 반례도 찾지 못했다. 또한, 소프트웨어 개발과 관련된 수많은 실험 결과도 살펴보았다. 흔히 오류로 귀결되는 실험의 결론을 보려는 것이 아니라 실험에서 추적한 데이터가 이 아이디어를 뒷받침할 수 있는지 알아보기 위해서였는데 완벽하게 입증되었다. 그리고 소프트웨어의 개발 이력과 유명 소프트웨어 프로젝트의 이력을 연구해 이 아이디어와 일치한다는 사실을 확인하였다. 따라서 내가 아는 한에서는 이 책에서 이야기하는 것을 반박하는 데이터나 경험은 소프트웨어 개발의 역사, 그 어디에도 없다.

물론 이 책에 있는 내용이 완벽하다는 것은 아니며, 단지 이 방법론이 유용하다는 것이다. 책에 담겨있는 아이디어를 적용한 프로젝트가 개선되었다는 사실을 최선을 다해 증명했다. 이제 어느 정도 정형화되었고, 더 정교한 여러 실험을 통해 이 아이디어를 강력한 과학으로 만들고 싶다. 하지만 지금 이 아이디어를 여러분에게 소개해도 손색이 없을 만큼 실용적이라고 생각한다. 이런 아이디어가 소프트웨어 개발에 있어 보편적 법칙이라 믿는다. 이것들은 우리가 사는 우주의 자연법칙을 나타내며, 모든 소프트웨어 프로젝트를 간단하고, 합당하며, 성공적으로 이끌어갈 잠재 능력을 갖췄다.

아이디어의 가장 특이한 점은 매우 간단하다는 것이다. 사실, 이 아이디어를 읽다 보면 바보 같을 정도로 너무 간단하다고 생각할 수도 있다. 아이디어의 대부분은 사람들이 전혀 몰랐던 것이 아니라 법칙이라는 점을 몰랐던 것이다. 소프트웨어 개발에 있어 올바른 의사결정을 위한 기본으로서, 그리고 최고의 모든 실례로부터

알아낸 확고한 사실로서 이 아이디어를 생각하면 아이디어의 참된 가치를 알게 될 것이다.

이 책의 모든 내용을 알고 있더라도 다음과 같이 생각해보자. "선배 프로그래머의 어려웠던 경험을 똑같이 겪을 필요 없이, 신입 프로그래머가 이 개념을 모두 배운다면 어떻게 될까?" 지금 이 순간에도 몇몇 회사는 미숙함에서 비롯된 나쁜 사례의 끊임없는 혼란 속으로 많은 신입 프로그래머를 내몰고 있다. 신입 프로그래머가 여러 번 안 좋은 사례를 경험해야만 소프트웨어 엔지니어링의 기본 실무를 익힐 수 있는 것일까? 내가 이 책에서 말하고자 하는 게 바로 그거다. 모든 프로그래머에게 소프트웨어의 가장 중요한 부분을 이해하는 기회가 되었으면 좋겠다. 이 책에서 첫 번째로 소개하는 사실은 다음과 같은데, 가장 최근에 발견한 것 중 하나다.

능력 없는 프로그래머와 능력 있는 프로그래머의 차이는 이해의 정도다.

말하자면, 능력 없는 프로그래머는 무엇을 해야 할지 정확히 이해하지 못하는 반면, 능력 있는 프로그래머는 무엇을 해야 하는지 이해했다는 이야기다. 믿거나 말거나 이는 매우 간단한 문제다. 이해한 만큼 지금 하는 일을 더 잘해낼 수 있다. 다른 분야와 같이 프로그래밍에서도 똑같다. 그렇지만 소프트웨어를 만드는 일은 이해가 모든 것인 순수한 정신 활동이므로 프로그래밍에서 더욱 중요하다.

이 책에서 설명하는 내용이 모든 문제를 바로 해결해주는 것은 아니며, 특정 상황에서 정확하게 무엇을 하라고 말해줄 수도 없다. 대신에 소프트웨어 개발에 대해 생각해볼 수 있는 새로운 방법을 제시한다. 이 방법을 여러분이 처한 상황에 맞게 어떻게 사용하는가는 각자의 몫이다. 오직 여러분만이 정확하고 구체적인 의사결정을 하기 위해서 소프트웨어로 무엇을 진행해야 하는지 알 수 있다. 이 책에는

의사를 결정할 수 있게 이끌어줄 일반적인 원리가 담겨 있다.

프로그래머가 아니더라도 이 책을 다음과 같이 유용하게 사용할 수 있다.

- 소프트웨어 조직을 위한 교육자료로 사용할 수 있다.
- 소프트웨어 엔지니어가 어떤 일을 하고 싶어하는지, 소프트웨어를 특정 방법에 따라 개발해야 하는지를 더 효과적으로 이해하는 데 사용할 수 있다.
- 소프트웨어 엔지니어와 의사결정을 하거나 소프트웨어 엔지니어와 효과적으로 대화하는 데 사용할 수 있다.

프로그래밍 경험이 많지 않거나 책을 읽는 데 소질이 없더라도 소프트웨어 관련 분야에서 일하는 사람이라면 이 책을 읽어야 하며 이해해야 한다. 기술적 이해도가 높다면 이 책에 나오는 개념 일부를 보다 쉽게 이해할 수 있겠지만 대부분의 개념은 프로그래밍 경험이 없어도 이해할 수 있다.

사실 이 책은 소프트웨어 개발을 다루지만 프로그램 코드는 거의 없다. 책에서 제안하는 원리들이 프로그래밍 언어와 상관없이 모든 프로젝트에 적용되어야 하기 때문이다. 특정 언어를 알아야만 프로그래밍 언어에 적용되는 내용을 이해할 수 있는 것은 아니다. 하지만, 각 원리를 쉽게 이해할 수 있도록 이 책 전반에 걸쳐 실제 사례와 유사한 내용을 사용했다.

무엇보다도 이 책은 여러분과 여러분의 소프트웨어에 도움이 될 것이며, 소프트웨어 개발 분야에 이성, 질서, 단순함을 실현시켜 줄 것이다.

정의, 사실, 규칙, 법칙

이 책에는 소프트웨어 개발에 관련된 정의, 사실, 규칙, 법칙이 나열되어 있는데 중요함을 나타내기 위해 들여쓰기와 굵은 글씨체로 표시하였다.

- **정의**는 어떤 말이나 사물이 무엇이며, 이것을 어떻게 사용할지를 말한다.
- **사실**은 어떤 내용이 참인 문장이다. 정보의 참인 부분이 사실이다.
- **규칙**은 참된 충고를 말하는 문장이다. 특정한 것을 수용하고 의사결정을 할 수 있게 해주지만, 미래를 예측하거나 그 밖의 사실을 알아내는 데 반드시 도움이 되는 것은 아니다.
- **법칙**은 항상 참인 사실이며 다양한 지식을 포함하고 있다. 다른 중요한 사실을 알아내는 데 도움이 되며, 미래에 어떤 일들이 일어날지를 예측하는 데 도움이 된다.

이 중에서 법칙이 가장 중요하다. 이 책에서는 명시적으로 법칙이라 표시하고 있으므로 무엇이 법칙인지를 알게 될 것이다. 어떤 정보가 어떤 범주 안에 들어가야 하는지 모호하면 부록 B를 참조한다. 이 책에서 설명하는 중요한 내용과 이 내용이 법칙, 규칙, 정의, 오래된 엄연한 사실 중 어디에 속하는지 정리해 두었다.

역자 서문

처음 『Code Simplicity』라는 책 제목을 접했을 때 '저자가 말하고자 하는 것이 무엇일까?' 의문을 가졌다. 책을 읽은 후 '다 알고 있는 내용인데?'라는 생각이 들었다. 이 책은 우리가 이미 알고는 있지만 실행에 옮기지 못하고 있는 부분을 다시 상기시켜주고, 애매한 부분, 즉 말로 표현하기 어려웠던 부분을 정의, 사실, 규칙, 법칙화하여 제시한다. 또한 객체지향 프로그래밍, 모듈화, 프레임워크 등의 기술과 용어가 나오게 된 배경을 생각해 볼 기회가 될 것이다.

이 책을 읽는 독자는 "현재 운영 중인 시스템이나 진행하고 있는 프로그램을 얼마나 오래 유지할 수 있을까?"라고 고민할 것이다. 프로그램이 등장하고 나서 지금까지 끊임없이 반복되는 이 질문에 대해 여러 기술적인, 방법론적인 접근을 통해 답을 찾고자 노력해왔다. 이 책의 저자는 오픈 소스 프로젝트인 버그질라를 진행하면서 똑같은 질문을 스스로 하고 그 답을 찾는 과정에서 이 책을 집필했다. 그리고 꼭 기억해야 하는 부분을 하나의 문장으로 표현하려 했다. 세상에 보이지 않지만 존재하는 진리가 있듯 프로그램의 세계에도 꼭 따라야만 하는 진리가 있으며, 믿고 따라야 한다고 강조한다.

이 책의 마지막 부록에 있는 정의, 사실, 규칙, 법칙 리스트를 업무 중 쉽게 볼 수 있는 위치에 놓고 지속적으로 보길 바란다. 시간이 흐른 후 본인이 만든 코드가 자신뿐만 아니라 다른 프로그래머도 보기 좋은 코드구나 생각하게 될 것이다.

2012.11. 정자동 사무실에서 신정안

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook 입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내시는 선배, 전문가, 고수분에게는 보다 쉽게 집필하실 기회가 되리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 준비 중이며, 조만간 선보일 예정입니다.

2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정한 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나, 저자(역자)와 독자가 소통하면서 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여, DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT기기에서 자유롭게 활용하실 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편리하게 전자책을 보실 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

1 | 들어가며

소프트웨어는 '컴퓨터에 전달하는 명령어'라고 배워왔다. 물론 맞는 말이다. 하지만 소프트웨어 개발 분야만큼 명령어와 그 결과의 관계가 밀접한 분야는 없다. 다른 분야에서는 명령(어)을 작성하고 다른 사람에게 전달한 후 명령(어)이 제대로 전달되었는지 확인하려면 오랜 시간 기다려야 하는 일도 있다. 그러나 코드를 작성할 때 우리와 컴퓨터 사이에는 중간 단계가 없다. 컴퓨터는 명령어가 지시한 대로 정확히 결과를 낼 것이다. 최종 결과의 질은 전적으로 컴퓨터의 질, 아이디어의 질, 그리고 코드의 질에 따라 달라진다.

이 세 가지 중에서 코드의 질은 요즘 프로젝트가 직면한 가장 큰 문제다. 그러므로 이 책 대부분에서 코드의 질 향상에 초점을 맞출 것이다. 여러 아이디어와 컴퓨터에 대해서도 다루겠지만 컴퓨터에 전달할 명령어의 구조와 질을 향상시키는 데 주로 초점을 맞출 것이다.

더 나은 결과를 위해 우리가 하는 일에 초점을 맞춰야 한다는 점을 기억해야 한다. 이 책은 결코 나쁜 결과를 허용하지 않을 것이다. 코드를 개선하는 데 집중하는 이유는 결과를 개선하려면 꼭 풀어야 할 중요한 숙제가 바로 코드의 개선이기 때문이다.

1.1 왜 단순화인가?

집에서 사용하는 가전제품이 컴퓨터처럼 자주 오동작하면 바로 반품했을 것이다. 하지만 요즘은 자연스럽게 "소프트웨어는 당연히 버그가 있어!"라고 생각하게 되었다. 시스템이라는 것은 시간이 지날수록 커지고 유지할 수 없게 되어, 결국 버리고 다시 작성해야 하는 불안정한 흉물이 된다는 것을 당연하게 여긴다.

이를 어쩔 수 없다고 말하는 것은 핑계다. 모든 소프트웨어가 수용할 수 있는 보편적 법칙을 통해 불안정성, 비대함, 그 밖의 코드 관련 문제가 생기지 않게 할 수 있다.

이런 문제가 생기는 주된 원인은 복잡함에 있다.

개발 초기의 소프트웨어 시스템은 작고 유지하기 쉽다. 그러나 항상 시간이 지날수록 커진다. 일반적인 소프트웨어 시스템은 모든 코드를 이해할 수 있는 사람이 없을 정도로 비대해진다. 좋고 나쁘고를 떠나서 이것이 현실이다. 실질적인 소프트웨어 시스템은 모두 복잡할 수밖에 없다. 사람들이 시스템으로 작업할 때 바라는 것이 하나 있다면 부분적으로 단순화하는 것이다. 그래서 각 부분을 볼 때는 어떻게 사용하는지, 어떤 기능이 있는지 이해할 수 있기를 바란다. 프로그래밍은 본질적으로 복잡함을 줄여 단순화하는 작업이다.

프로그래머가 자신이 작성한 코드를 단순화하지 않으면 코드는 점점 이해하기 어려워진다. 결국 디버깅과 수정 그리고 새로운 기능을 추가하기도 어려워진다. 시스템 전반에 걸쳐 복잡해지기 시작하면 시스템을 유지하기 어렵다. 오늘날 소프트웨어 개발의 문제점은 바로 프로그래머가 시스템에서 복잡함을 없애지 않고 오히려 더하는 데 있다.

뛰어난 프로그래머는 다른 프로그래머가 사용하고 이해하기 쉽게 프로그램을 최대한 단순하게 만든다.

여/기/서/잠/깐 단순화에 대한 오해

가끔, 단순함을 "프로그램은 코드가 많으면 안 되며, 고급 기술을 사용하면 안 된다"라고 오해한다. 그러나 사실은 그렇지 않다. 코드가 많으면 실제로 시스템이 더욱 단순해질 수 있다. 좀 더 많이 작성했다면 좀 더 많이 읽으면 된다. 또한, 고급 기술은 학습시간이 필요하지만 코드를 좀 더 단순하게 만들 수 있다.

원하는 기능을 빨리 코딩하는 것보다 단순하게 코딩하는 데 시간이 더 오래 걸린다고 이야기하는 사람도 있다. 하지만 이 말을 증명할 만한 데이터는 어디에도 없다. 만만찮은 소프트웨어를 개발하려면 적게는 몇 주, 몇 달이 걸린다. 오늘 프로그램이 좀 더 복잡해졌다면 내일은 작업이 더욱 더뎌질 것이다. 복잡함이 지름길이라고 생각할 수도 있지만, 단순하게 코딩하면 오히려 작업이 빠르게 처리된다고 많은 연구가 말한다.

1.2 소프트웨어 설계

이 책의 많은 부분에서 소프트웨어 설계 및 코드의 구조를 계획하는 과정을 다룰 것이다.

> **NOTE_** 이 책에서 말하는 '설계'는 '소프트웨어 설계'를 의미한다. 다시 말해서 시각 디자인, 사용자 인터페이스 디자인 같은 시각적인 디자인을 의미하는 것이 아니다.

코딩하는 중에도 의사결정이 이뤄지기도 하지만 소프트웨어는 항상 많은 설계 단계를 거친다. 프로그래머 팀 내의 모든 사람은 설계에 참여한다. 리더 프로그래머는 프로그램의 구조 전반에 걸친 설계를 책임지고, 중급 프로그래머는 큰 부분의 개발 설계를 책임진다. 초급 프로그래머는 파일 하나의 코드 일부만을 담당하더라도 자신의 개발 부분에 대한 설계를 책임진다. 코드 한 줄에도 많은 설계가 들어간다.

소프트웨어를 작성하는 모두가 설계자다.

팀 내 모든 프로그래머는 자신이 작성한 코드가 제대로 설계됐는지를 책임져야 한다. 프로젝트를 위해 코딩하는 그 어떤 누구도 직위를 막론하고 설계를 무시하면 안 된다.

그렇다고 설계가 민주주의 방식으로 이뤄진다는 말은 아니다. 위원회가 설계하면 더욱 나쁜 설계가 될 수 있으며, 설계가 더욱 복잡해질 수도 있다.[01] 모든 프로그래머가 자신의 영역에서 제대로 설계할 수 있게 의사결정 권한이 있어야 한다. 취약하거나 애매하게 의사결정이 이뤄진다면 하위 설계자에 대한 거부권을

01 "낙타는 위원회가 고안한 말이다(a camel is a horse designed by a committee)"처럼 여러 사람이 모여서 간단한 일을 놓고 논의하면 엉뚱한 결과를 낳을 수 있다는 뜻이다. _ 옮긴이주

가진 중급 프로그래머나 리더 프로그래머가 수정해야 한다.[02] 그렇지 않으면 코드 설계의 책임은 실제로 코드를 작업한 사람에게로 넘어간다.

일반적으로 프로그래머는 좋은 아이디어를 떠올리는 똑똑한 사람이므로 설계자는 항상 그들의 제안이나 피드백을 경청해야 한다. 의사결정은 모든 데이터를 고려한 후에 다수 사람이 아닌 반드시 개인이 해야 한다.

02 결정된 사항을 수정해야 하는 사람은, 수정된 내용을 다른 프로그래머에게 가르쳐줘야 한다. 이를 통해서 어떻게, 왜 수정된 내용이 더 좋은지를 설득해야 한다. 이렇게 함으로써, 시간이 지날수록 수정하는 횟수가 줄어들어야 한다. 하지만 몇몇 프로그래머는 학습하지 않는다. 몇 달 심지어 몇 년 동안 교육을 했는데도 나쁜 의사결정을 한다면, 이 프로그래머는 팀에서 나가야 한다. 그러나 대부분 프로그래머는 빠르고 현명하게 의견을 수렴하므로 이런 걱정을 할 일은 드물다.

2 | 소프트웨어의 목적

소프트웨어 개발 법칙을 이야기하기 전에, 소프트웨어 개발 법칙을 이용하여 달성해야 할 목적을 이해해야 한다. 소프트웨어 개발 법칙이 목적 달성에 제대로 이용될 수 있는지를 판단하는 기준은 무엇일까? 목적을 먼저 생각해본 후에야 소프트웨어 개발 법칙이 목적을 이루는 데 제대로 이용될 수 있는지 말할 수 있을 것이다.

따라서 우리에게 필요한 것은 소프트웨어의 목적을 표현하는 문장이다. 프로그래머 개인의 목적도, 조직이 프로그래머를 고용한 이유도 아닌, 소프트웨어 전체가 갖는 실질적인 목적을 표현하는 문장이 필요하다. 그런 후에야 소프트웨어 개발 법칙과 규칙이 소프트웨어의 목적을 이루는 데 도움이 되는지 확인할 수 있다.

모든 소프트웨어의 목적은 다음과 같이 한 문장으로 표현할 수 있다.

소프트웨어의 목적은 사람들에게 도움을 주는 것이다.[01]

소프트웨어마다 각기 고유한 목적이 있다. 예를 들어, 워드 프로세서는 문서 작업하는 데 도움을 주고 웹 브라우저는 웹 페이지 보는 것을 도와준다.

특정 그룹에 속한 사람만을 위한 소프트웨어도 있다. 예를 들어, 회계 소프트웨어는 회계사를 돕는 소프트웨어다. 이런 소프트웨어는 특정 그룹에 속한 사람을 대상으로 한다.

동물이나 식물에 도움이 되는 소프트웨어의 목적은 무엇일까? 사실 이 소프트웨어는 동물이나 식물을 관리하는 사람을 돕는 게 목적일 것이다.

---

01 소프트웨어의 목적에 대한 사실은 법칙보다 중요하다. 영어에서 사실을 간단히 설명할 수 있는 단어는 없다. 법칙의 기준에 정확히 맞지는 않지만(예, 미래를 예측할 수 없다), 아마도 '상위 법칙'이라고 부를 수도 있을 것이다. 단순함에 대한 이해를 돕기 위해 이 책의 마지막 부분에 법칙으로써 사실을 나열해 두었다.

여기서 짚고 넘어가야 할 점은 소프트웨어는 절대 무생물체를 돕는 것이 아니라는 점이다. 소프트웨어는 컴퓨터를 돕는 것이 아니라 컴퓨터를 사용하는 사람을 돕는다. 라이브러리를 작성하는 것조차도 다른 프로그래머를 도우려고 작성하는 것이지 절대 컴퓨터를 위해 작성하는 게 아니다.

그렇다면 '돕는다'의 의미는 무엇일까? 여러 가지 측면에서 주관적인 의미를 갖는다(한 사람을 돕는 것이 다른 어떤 것을 돕는 것을 의미하지는 않는다. 그러나 이 단어는 사전적인 정의가 있으므로 단어 자체가 의미하는 것을 전적으로 개인에 따라 달라진다고 말하기 어렵다. 미국 영어 사전Webster's New World Dictionary에 '돕는다'는 다음과 같이 정의돼 있다.

사람이 무엇을 하기 쉽게 만드는 것; 지원하다; 보조하다; 노동을 공유 또는 쉽게 하다.

스케줄을 정하고, 책을 쓰고, 다이어트를 계획하는 등 도움을 줄 수 있는 것들이 많다. 어떤 도움을 줄 것인가 결정하는 것은 자신의 몫이지만, 그 목적은 항상 도움을 주는 데에 있다.

소프트웨어의 목적은 '돈 벌기'이거나 '얼마나 똑똑한지 보여주기'가 아니다. 이런 목적을 갖고 프로그램하는 사람은 소프트웨어의 목적을 무시하는 것이며, 문제를 일으킬 것이다. 돈 벌기나 얼마나 똑똑한지 보여주기 등은 자기 자신을 위한 방법이지만 다른 사람을 돕는 방법은 아니다. 이런 목적을 갖고 소프트웨어를 설계하면, 사람들이 하고자 하는 것을 도우려는 진정성을 가지고 만든 소프트웨어보다 질이 낮은 소프트웨어를 만들게 될 것이다.[02]

02 '돈 벌기'는 개인이나 조직의 목적 중 하나다. 돈을 버는 것이 나쁜 것은 아니지만 소프트웨어 목적 자체가 돈이 되어서는 안 된다. 어떤 경우는 얼마나 많은 돈을 벌었는지가 얼마나 많은 사람을 도울 수 있는가와 직접적으로 연관되기도 한다. 사실 소프트웨어 산업의 수입을 결정짓는 두 가지 요소는 '조직의 사업 능력(관리, 경영, 마케팅, 판매)'과 '얼마나 사람을 돕고 있는가'이다.

다른 사람을 도울 생각이 없는 사람은 질 낮은 소프트웨어를 만들게 된다. 즉, 이런 소프트웨어는 사람들에게 도움이 안 된다. 오랜 시간 프로그래머들을 관찰한 결과 다음과 같이 확언할 수 있다. "좋은 소프트웨어를 만들 수 있는 잠재능력은 다른 이를 돕겠다는 생각을 얼마나 많이 하느냐에 달렸다."

결국, 소프트웨어에 대한 의사결정을 할 때 기준이 되는 것은 사람을 어떻게 도울 수 있는가에 해당한다. 도움의 정도는 다르다. 많이 도울 수도 거의 돕지 못할 수도 있으며, 많은 사람을 또는 소수를 도울 수도 있다. 이런 기준으로 우선순위를 정할 수 있다. 어떤 부분이 사람을 가장 많이 도울 수 있을까? 어떤 부분이 우선순위가 가장 높은가? "어떻게 우선순위를 정할 것인가"에 대해 더 알아야 할 것들이 있지만 "사용자를 얼마나 많이 돕는가?"가 소프트웨어 시스템의 변화를 이끌어 낼 수 있는 가장 기본이 되는 질문이다.

일반적으로, '사람을 돕는다'라는 목적이 소프트웨어를 설계할 때 가장 명심해야 할 중요한 부분이며, 이런 목적이 소프트웨어 설계 법칙을 만들고 이해할 수 있도록 해줄 것이다.

2.1 실제 애플리케이션

현업에서 소프트웨어의 목적을 어떻게 프로젝트에 적용할 것인가? 프로그래머가 사용할 텍스트 편집기를 만든다고 해보자. 가장 먼저 해야 할 일은 소프트웨어의 목적을 결정하는 것이다. 목적을 간단하게 정하는 것이 가장 중요하다.

이 프로젝트의 목적은 "프로그래머가 텍스트를 편집하는 데 도움을 주는 것이다."라고 정해졌다고 해보자. 이것보다 더 구체적이면 더욱 더 도움이 될 수도 있으나 같이 일하는 그룹이 구체적인 목적에 동의할 수 없다면, 적어도 위와 같은 간단한 한 가지를 찾아야 한다.

이제 목적은 정해졌으니 다음으로 요구사항을 모두 살펴보자. 각 항목에 대해 "프로그래머가 텍스트를 편집하는 데 어떤 도움이 되는가?"라고 스스로 물어볼 수 있다. 이에 대한 대답이 "도움이 안 된다"면 바로 리스트에서 이 항목을 제거한다. 남아있는 항목에 대해서도 짧은 문장으로 답을 적어보자. 예를 들어, 자주 사용하는 기능을 키보드 단축키로 추가해 달라는 요청 항목에 대해 "모든 문장을 직접 칠 필요 없이 프로그래머가 빠르게 작업할 수 있으므로 도움이 된다."라고 답할 수 있다 (실제 상황에 맞지 않는다면 이 모든 것을 다 적을 필요는 없다. 그냥 답에 대해 생각하는 것만으로도 충분하다).

이 질문에 답해야 하는 이유는 다음과 같다.

- 기능 설명 또는 기능 구현 등 막연했던 것을 해결할 수 있다. 예를 들어, 키보드 단축키에 대한 답은 결국 "텍스트 편집기는 빨라야 한다."이다. 즉 이것이 사용자가 추구하는 가치다.

- 팀이 기능의 가치에 동의할 수 있게 해준다. 몇몇 사람은 키보드 단축키 기능을 좋아하지 않을 수 있지만, 단축키 기능에 대한 대답은 이 기능이 왜 가치가 있는지를 설명한다면 그 점에 모두 동의할 수 있어야 한다. 사실, 키보드 단축키 기능을 사용하지 않고 빠르게 편집하는 더 좋은 방법을 아는 프로그래머도 있을 것이다. 좋다. 이 방법이 더 좋은 아이디어를 생각나게 할 수 있다면 단축키 기능 대신에 그것을 구현해야 한다. 답은 정말 필요한 것을 말해주는 것이지 사용자가 원한다고 생각한 것을 말해주는 것이 아니다.

- 질문에 답하면서 몇몇 항목이 다른 것에 비해 상대적으로 중요하다는 것을 확인하게 해준다. 이 절차를 통해 프로젝트 리더는 우선순위를 정하는 데 도움을 받을 수 있다.

- 최악의 경우 너무 많은 요청 항목을 수용해 텍스트 편집기가 너무 커져 버렸을 때, 어떤 항목을 제거해야 하는지 결정하는 데 도움이 될 것이다.

또한 버그 리스트를 만들어 검토한 후 이전과 반대되는 질문을 할 수 있다. "이 버그는 프로그래머의 텍스트 편집을 어떻게 방해하는가?"라는 항목의 답이 명확하다면 이 항목은 실제로 프로그래밍할 필요가 없다. 예를 들어, 파일을 저장할 때 프로그램이 멈춘다면 이 버그가 왜 나쁜지 굳이 설명할 필요가 없다. 일상 업무에서 소프트웨어의 목적을 적용하는 방법은 다양하다. 지금까지 이야기한 것은 예의 일부분에 불과하다.

3 | 소프트웨어의 미래

소프트웨어 설계자가 직면한 근본적인 질문은 “소프트웨어에 대한 의사결정을 어떻게 할 것인가?”다. 여러 갈림길에서 어떤 것이 최선일까? 어떤 의사결정이 절대적으로 옳은지 그른지를 묻는 것이 아니다. 단지, 여러 선택 사항 중 지금 선택한 것이 다른 것보다 좋은가를 알고 싶은 것이다. 우선순위를 결정하는 데 있어서 모든 가능성 중 가장 좋은 결정을 해야 한다. 예를 들어, 설계자는 다음과 같이 생각할 수 있다. “오늘 처리해야 할 항목이 100개 있다. 하지만 현재 2개를 겨우 처리할 수 있는 인력만 있다. 그럼 어떤 것을 먼저 처리해야 할까?”

3.1 소프트웨어 설계 방정식

위 질문을 포함해서 실제 소프트웨어 설계의 특성에 대한 질문은 다음 방정식으로 답할 수 있다.

$$D = \frac{V}{E}$$

D : 변경에 대한 바람직함의 정도. 무언가 하기를 얼마나 원하는가?

V : 변경에 대한 가치. 이 변화는 얼마나 가치가 있는가? 일반적으로, 다양한 방법으로 가치를 매길 수 있지만 “사용자에게 얼마나 도움이 되는가?”란 질문으로 이 가치를 결정할 수 있다.

E : 변경에 필요한 노력. 변경하는 데 필요한 일의 양이다.

이 방정식은 다음과 같이 표현할 수 있다.

변경에 대한 바람직함의 정도는 변경의 가치에 비례하고, 변경에 필요한 노력에 반비례한다.

변경이 절대적으로 옳거나 그르다는 것이 아니다. 여러 선택 사항의 우선순위를 어떻게 매겨야 하는지 방법을 알려주려는 것이다. 적은 노력을 통해 많은 가치를 얻을 수 있는 변경이 더 좋다.

"변경하지 않고 똑같이 유지하면 어떨까?"라고 질문할지 모르지만, 이 방정식에 이미 답이 나와있다. "유지하면서 얻을 수 있는 가치는 무엇인가? 유지하려면 얼마나 노력해야 하는가?"를 스스로 물어보고, 이를 변경 전후의 가치 변화와 변경을 위한 노력과 비교해보자.

3.1.1 가치

방정식에서 '가치'가 의미하는 것은 무엇일까? 가치를 간단히 정의하면 다음과 같다.

어디서든 누구에게나 변경이 도움을 줄 수 있는 정도

사용자는 도움을 줘야 하는 가장 중요한 사람이다. 그러나 자신을 경제적으로 지원하려고 기능을 작성하는 것도 가치의 한 가지 형태다. 자기 자신에게 가치가 있는 것이다. 사실, 변경이 가치를 갖는 방법은 여러 가지다. 여기에서는 일부 예만 들었다.

가끔은 변경에 대한 가치를 실질적이며 정확한 수치로 말하기 어렵다. 예를 들어, 소프트웨어가 사람의 몸무게를 줄이는 데 도움이 되었다고 해보자. 누군가의 몸무게를 줄이는 데 도움을 줬다고 해서 그 정확한 가치를 어떻게 측정할 것인가? 답은 "할 수 없다"이다. 그러나 소프트웨어의 일부 기능은 몸무게를 줄이는 데 '많은' 도움이 되었고, 그 외 몇몇 기능은 몸무게를 줄이는 데 '전혀' 도움이 되지 않았다는 사실은 알 수 있다. 그렇기 때문에, 여전히 가치를 통해서 변경의 등급을 매길 수 있다.

프로그래머의 경험과 사람들에게 가장 도움이 되는 것이 무엇인지에 대한 연구를 통해 변경의 가치를 이해할 수 있다.

가치의 가능성과 잠재적 가치

가치는 실제로 두 가지로 구성된다. 가치의 가능성(변경을 통해 사용자에게 도움이 될 가능성이 얼마나 되는가?)과 잠재적 가치(변경을 통해 사람들에게 얼마나 많은 도움이 될까?)다.

간단한 예로 살펴보자.

- 확률이 수백만 분의 일이라 해도 누군가를 살릴 수 있다면 매우 가치 있는 일이다. 이는 잠재적 가치가 높지만 가치의 가능성이 낮은 경우다. 다른 예로 시각 장애우를 위해 숫자 입력 기능을 추가하는 스프레드시트 프로그램을 생각해보자. 시각 장애우의 수는 적지만 이 기능이 없이는 시각 장애우들은 스프레드시트 프로그램을 전혀 사용할 수 없다. 다시 말해, 이 기능은 소수 사용자에게만 적용되더라도(가치의 가능성이 낮음) 잠재적 가치가 높기 때문에 가치가 있다.

- 사용자를 100% 웃게 하는 기능이 있다면 이것도 가치 있는 일이다. 잠재적 가치가 낮지만(사람을 웃게 하는) 많은 사람에게 영향을 주기 때문에 가치의 가능성은 높다.

- 반면에, 소수의 사람만 웃게 할 수 있고, 확률이 수백만 분의 일밖에 안 되는 기능을 구현하는 일은 전혀 가치가 없다. 잠재적 가치와 가치의 가능성 모두 낮다.

그러므로 가치를 생각할 때는 다음 사항을 고려해야 한다.

- 이 변경 사항이 얼마나 많은 사용자에게 가치가 있는가?

- 이 기능이 사용자에게 가치 있을 가능성은 얼마나 되는가? 다시 말해, 이 기능이 얼마나 자주 가치를 갖는가?

- 언제 가치가 있으며, 얼마나 가치 있는가?

손해의 균형

일부 변경은 도움을 주기도 하지만 손해를 입히기도 한다. 예를 들어, 소프트웨어 속 광고가 프로그래머 지원 차원에서 도움이 되지만 몇몇 사용자는 싫어할 수도 있다.

변경의 가치를 계산하는 일에는 얼마나 피해를 줄 수 있는가도 포함되어야 하며, 가치와 대등하게 균형을 맞춰야 한다.

사용자의 가치

개발한 기능을 사용하는 사람이 없으면 직접적인 가치가 없다. 가치가 없는 기능은 사용자가 찾을 수 없는 기능, 사용하기 너무 어려운 기능, 쉽게 말해 어떤 누구에게도 도움이 안 되는 기능이다. 미래에는 가치가 있을 수도 있지만 지금 당장은 가치가 없다.

다시 말하자면 소프트웨어를 적절한 시기에 배포해 가치를 가질 수 있도록 해야 한다는 의미다. 가치를 위해 장시간에 걸쳐 소프트웨어를 변경해야 한다면 어떤 가치도 가질 수 없다. 사람들이 필요로 할 때 배포되지 않았기 때문이다. 변경을 결정할 때는 소프트웨어 배포 시점을 고려해야 한다.

3.1.2 노력

노력은 가치보다는 수치로 표현하는 것이 쉽다. 일반적으로 '몇 사람이 몇 시간 동안'과 같이 노력을 수치로 표현할 수 있다. 노력을 수치로 표현하는 예로 '연당 100명'은 한 사람이 100년 동안, 100명이 1년 동안, 50명이 2년 동안 등으로 작업할 양을 표현할 수 있다.

노력을 수치로 표현할 수 있지만 특정 상황에서는 수치로 계산하기가 매우 까다롭고 불가능할 때도 있다. 변경에는 예측하기 어려운 보이지 않는 비용도 포함된다. 예를 들어, 변경으로 생기는 버그를 수정하려면 추가 비용이 필요하다. 그러나 경험이

많은 프로그래머는 정확한 수치를 모르더라도 필요한 노력이 어느 정도인지를 가늠해 변경에 대한 우선순위를 매길 수 있다.

변경에 대한 노력을 고려할 때는 단순히 프로그래밍하는 시간뿐만 아니라 이에 수반되는 모든 노력을 고려하는 것이 중요하다. 연구 조사하는 기간은 얼마나 필요하며, 프로그래머들 간에 얼마나 많은 의사소통을 해야 하는가? 변경을 위해 얼마 동안 고민해야 하는가? 등을 고려해야 한다.

간단히 말해, 변경과 연관된 모든 시간이 노력에 대한 비용이다.

3.1.3 유지

방정식은 의외로 간단하다. 그러나 시간이라는 중요한 요소가 빠져 있다. 변경 사항을 구현해야 하며, 계속해서 유지해야 한다. 모든 변경 사항은 유지가 필요하다. 세금 처리 관련 프로그램을 개발할 때는 당연히 매년 바뀌는 세금 법규를 프로그램에 반영해야 한다. 그러나 내년에 코드를 테스트해 정상적으로 동작하는지를 확인하려는 비용임에도 불구하고, 이 변경은 직접적으로는 장기적인 유지비용을 갖지 않을 것처럼 보인다.

우리는 현재와 미래 가치를 모두 고려해야 한다. 시스템의 변경 사항을 구현하면 현재 사용자에게 도움이 될 것이고, 미래의 사용자에게도 도움이 될 것이다. 즉, 미래의 사용자 수에 영향을 미치기 때문에 시스템의 변경 사항은 얼마나 많은 사람을 도울 수 있는가를 결정한다.

시간이 지나면서 가치가 변하는 기능도 있다. 예를 들어, 2011년 세금 법규를 적용한 세금 프로그램은 2011~2012년 동안에는 가치가 있지만, 2013년에는 가치가 없을 것이다. 시간이 지나면서 그 가치를 잃어가는 기능이 있는 반면, 시간이 지날수록 가치가 높아지는 기능도 있다.

현실적으로 보면 노력에는 '구현을 위한 노력'과 '유지를 위한 노력'이 모두 포함

되며, 가치는 현재의 가치와 미래 가치가 포함된다. 그럼 다음과 같은 방정식이 나온다.

$$E = E_i + E_m$$

$$V = V_n + V_f$$

E_i : 구현을 위한 노력

E_m : 유지를 위한 노력

V_n : 현재 가치

V_f : 미래 가치

3.1.4 전체 방정식

위 내용이 모두 담긴 전체 방정식은 다음과 같다.

$$D = \frac{V_n + V_f}{E_i + E_m}$$

방정식을 풀이하면 다음과 같다.

변경의 바람직함의 정도는 현재와 미래에 대한 가치와 정비례하며, 구현과 유지에 필요한 노력에 반비례한다.

이것이 바로 소프트웨어 설계의 기본 법칙이다. 이 법칙에 대해 더 알아보자.

3.1.5 방정식 간소화

'미래 가치'와 '유지를 위한 노력'은 시간과 관련 있다. 이것을 실제 세계에 적용할 때 방정식은 흥미로운 점을 보여준다.

이를 증명하기 위해 가치와 노력을 위한 방정식을 푸는 데 돈을 사용할 수 있다고 가정해보자. '가치'는 변경을 통해 얼마나 많은 돈을 벌 수 있는지로 측정할 것이다. '노력'은 변경 사항을 구현하려고 사용되는 비용이 얼마인지로 측정할 것이다.

실제로는 이처럼 방정식을 사용하면 안 되지만 예로 간단하게 살펴보자.

다음과 같은 방정식을 통해 변경에 필요한 비용을 계산하였다.

$$D = \frac{\$10{,}000 + \$1000/\text{day}}{\$1{,}000 + \$100/\text{day}}$$

이 변경은 변경 사항을 구현하는 데 $1,000의 비용이 들고(구현을 위한 노력, 분모의 왼쪽), $10,000를 번다(현재 가치, 분자의 왼쪽). 이후 매일 $1,000를 벌 수 있으며(미래 가치, 분자의 오른쪽), 유지하는 데 $100 비용이 든다(유지를 위한 노력, 분모의 오른쪽).

10일 후 축적된 미래 가치는 $10,000($1,000/일×10일)이며, 유지를 위한 노력은 $1,000($100/일×10일)이다. 10일만에 원래의 현재 가치와 구현을 위한 노력과 같아진다.

100일 후 미래 가치는 $100,000가 되며, 유지를 위한 노력은 $10,000가 된다.

1,000일 후 미래 가치는 $1,000,000가 되며, 유지를 위한 노력은 $100,000가 된다. 이 시점에서 보면 원래 '현재 가치'와 구현에 드는 비용은 매우 작아 보인다. 시간이 지날수록 중요성이 줄어들다가 결국에는 중요성이 전혀 없어진다. 따라서 시간이 지나면서 다음과 같은 방정식으로 줄여볼 수 있다.[01]

$$D = \frac{V_f}{E_m}$$

01 수학자들을 위한 추가 코멘트 : 수학적 개념을 이해한다면 시간이 무한으로 갈 때 방정식의 극한값을 분석한다는 사실을 알 것이다. 일반적으로, 정적 방정식이 아닌 한계가 있는 무한급수인 것처럼 소프트웨어 설계 방정식을 생각해야 한다. 그러나 단순화하려고 정적인 방정식으로 표현했다.

사실, 소프트웨어 설계를 위한 거의 모든 의사결정은 변경에 대한 미래 가치 대비 유지를 위한 노력을 측정하는 것으로 바꿔 간단히 말할 수 있다. 현재의 가치와 구현을 위한 노력이 의사결정에 있어서 매우 중요한 부분을 차지하는 때도 있지만, 이런 경우는 매우 드물다. 일반적으로 소프트웨어 시스템은 오랜 기간 유지해야 하므로 현재의 가치와 구현에 드는 노력은 장기간에 걸친 미래 가치와 유지를 위한 노력과 비교해보면 매우 적어진다.

3.1.6 무엇을 원하고, 무엇을 원하지 않는가?

이 장에서 언급된 주된 내용은 유지에 드는 노력이 미래 가치를 넘어서는 상황을 피하려는 것이다. 예를 들어, 변경 사항을 구현할 때 5일에 걸친 노력과 가치에 대한 비용이 다음과 같다고 가정해보자.

일수	노력	가치
1	$10	$1,000
2	$100	$100
3	$1,000	$10
4	$10,000	$1
5	$100,000	$0.10
합계	$111,110	$1111.10

절대로 이렇게 해서는 안 되는 최악의 변경이다. 이 비율로 진행한다면 시스템은 결국 유지할 수 없게 된다. 무한대로 비용이 들어가고 가치는 $0가 될 것이다. 가치보다 유지에 드는 노력이 더 빨리 증가하면 처음에는 괜찮아 보여도 결국은 곤란한 상황이 된다.

일수	노력	가치
1	$1,000	$1,000
2	$2,000	$2,000
3	$4,000	$3,000
4	$8,000	$4,000
합계	$15,000	$10,000

성공을 보장하는 이상적인 방안은 유지에 드는 노력을 계속 줄여 결과적으로 0이 되도록 시스템을 설계하는 것이다. 노력을 0으로 할 수만 있다면 미래 가치가 크든 작든 문제가 되지 않을 것이다. 이 부분에 대해서는 걱정할 필요가 없다. 다음 표는 바람직한 상황을 보여준다.

일수	노력	가치
1	$1,000	$0
2	$100	$10
3	$10	$100
4	$0	$1,000
5	$0	$10,000
합계	$1,110	$11,110

일수	노력	가치
1	$20	$10
2	$10	$10
3	$5	$10
4	$1	$10
5	$0	$10
합계	$36	$50

변경이 더 높은 미래 가치를 가져다 주리라 기대할 것이다. 모든 의사결정을 시간이 지날수록 유지비용을 0으로 만들 수 있다면 미래의 위험한 상황으로 몰고 가지는 않을 것이다.

이론적으로 미래 가치가 유지에 드는 노력보다 높다면 변경은 여전히 가치 있는 것이다. 그러므로 미래 가치를 유지에 드는 노력보다 높게 유지한다면 유지에 드는 노력과 미래 가치 모두 증가하도록 변경할 수 있다.

이런 변경은 나쁘지 않다. 그러나 구현에 드는 노력이 더 크더라도 유지에 드는 노력을 줄일 수 있는 변경이 더 가치 있다. 유지에 드는 노력이 줄어든다면 실제로 시간이 지날수록 변경은 더욱 가치 있게 된다. 다른 어떤 대안보다 좋은 선택이다.

유지에 드는 노력을 줄이도록 시스템을 설계하려면 구현하는 데 상당히 많은 노력을 들여야 하는 경우도 있다(꽤 많은 설계 작업과 계획이 필요하다). 그러나 구현에 드는 노력은 의사결정에서는 사소한 문제며 거의 무시되어야 한다. 요약하면 다음과 같다.

유지에 드는 노력을 줄이는 것이 구현에 드는 노력을 줄이는 것보다 더 중요하다.

이것은 소프트웨어 설계에 대해 꼭 알아야 할 것 중 하나다.

무엇이 유지하는 데 노력하게 하는가? 시간이 지날수록 유지에 드는 노력을 줄이려면 시스템을 어떻게 설계해야 할 것인가? 이 책의 나머지에서 주로 이 부분에 대해 다룰 것이지만, 먼저 미래에 대해 좀 더 확인해보자.

3.2 설계의 질

한 사람만을 위한 소프트웨어를 개발하기는 쉽다. 하지만 앞으로 수십 년 동안 많은 사람에게 도움이 되는 소프트웨어를 개발하기는 이보다 훨씬 어렵다. 그런데 개발에 드는 노력 대부분을 어디에 쏟아야 할까? 그리고 사용자의 대부분은 소프트웨어를 언제 사용하는가? 지금 바로인가, 아니면 몇십 년 안에 사용하게 될 것인가?

이 질문에 대한 대답은 개발이 좀 더 많이 진행된 곳, 도와야 하는 더 많은 사용자가 있는 곳, 그리고 현재보다는 미래에 있을 것이다. 소프트웨어는 미래의 경쟁 속에서 살아남아야 하며, 지속해서 유지되어야 한다. 그리고 유지에 드는 노력과 사용자의 수가 증가할 것이다.

미래가 있다는 사실을 무시하고 그저 '지금 동작하도록' 만들면 이 소프트웨어를 미래까지 유지하기 어렵다. 소프트웨어를 유지하기 어렵다면 지속해서 사람들에게 도움을 주기 힘들다(소프트웨어 설계의 목적). 새로운 기능을 추가할 수 없고 문제를 해결할 수 없다면 결과적으로 '나쁜 소프트웨어'가 되고 만다. 사용자에게 도움이 안 되며 버그투성이인 소프트웨어가 될 것이다.

이것은 다음과 같은 규칙을 말한다.

설계의 질적 수준은 시스템이 사람들에게 얼마나 오랫동안 도움을 줄 수 있는가에 비례한다.

잠깐 사용할 소프트웨어를 만든다면 설계에 많은 노력을 기울일 필요가 없다. 그런데 기껏해야 6개월 정도 사용할 거라 생각했는데 10년 이상 사용될지도 모른다면,

설계에 상당히 많은 노력을 기울여야 한다. 소프트웨어를 얼마나 오래 사용할지 잘 모르겠거든 가능한 한 오래 사용할 것이라 생각하고 소프트웨어를 설계하자. 어느 한 가지에 얽매이지 말고 유연하게 유지하며, 절대 바뀔 리 없는 의사결정을 하지 말고 설계에 최대한 많이 집중하자.

3.3 예측할 수 없는 결과

소프트웨어를 설계할 때 미래는 중요한 사항이다. 모든 엔지니어링에서 염두에 두어야 할 중요한 점은 다음과 같다.

미래에는 당신이 모르는 뭔가가 있다.

이를 소프트웨어 설계에 적용해 볼 때 미래에 대해서는 대부분 알 수가 없다.

프로그래머가 범하는 가장 일반적이고 심각한 문제는 사실상 알 수가 없는 미래에 대해 예측한다는 것이다.

예를 들어, 망가진 플로피 디스크를 수리하는 소프트웨어를 1985년에 개발했다고 가정해보자. 이 소프트웨어는 전적으로 플로피 디스크의 동작 방식에 의존하므로 플로피 디스크만 고칠 수 있다. 지금은 플로피 디스크를 사용하지 않아 이 소프트웨어는 퇴물이 되었다. 그 당시 프로그래머는 "플로피 디스크가 영원히 사용될 것이다"라고 예측했던 것이다.

가까운 미래는 예측할 수 있지만 먼 미래는 거의 알 수 없다. 설계에 대한 의사결정은 장기간에 걸쳐 영향을 끼치므로 가까운 미래보다는 먼 미래가 더 중요하다.

> **NOTE_** 미래를 예측하지 말고 대신에 알려진 최근 정보를 기반으로 설계에 대한 모든 의사결정을 하는 것이 가장 안전하다.

이 장에서 설명하는 것과는 정반대로 들리겠지만 그렇지 않다. 미래는 의사결정을 하는 데 고려해야 할 가장 중요한 사항이다. 그러나 미래의 변화를 허용한 설계와 미래를 예측하고 진행된 설계에는 큰 차이점이 있다.

먹는 것과 굶어 죽는 것 중 하나를 선택해야 한다고 가정해보자. 먹는 것이 더 좋은 선택임을 알고 있으므로 선택을 위해 미래를 예측할 필요가 없다. 이는 먹는 것을 선택하는 것이 지금 당장 사는 길이며, 사는 것이 죽는 것보다는 더 나은 미래이기 때문이다. 미래는 중요하며 의사결정에서 미래를 고려해야 하므로 좀 더 나은 미래를 위해 먹는 것을 선택한다. 그러나 이 선택에서는 미래를 예측할 필요가 없다("내일 내 아이의 삶을 보호해야 하므로 나는 지금 먹어야 해"라고 특정한 이유를 말할 필요가 없는 것이다). 내일 무슨 일이 발생하든 굶어 죽는 것보다 지금 먹는 것이 더 나은 내일이 될 수 있기 때문이다.

이와 비슷하게 소프트웨어 설계에서 미래에 정확히 어떤 일이 벌어질지 예측하지 않아도 지금 알고 있는 정보를 기반으로 좀 더 나은 미래를 위한 의사결정을 할 수 있다(유지를 위한 노력을 줄이고, 가치를 올린다).

때때로 곧 닥칠 미래에 무슨 일이 벌어질지 정확히 알 수 있다. 그리고 이 예측을 기반으로 의사결정을 할 수 있다. 그러나 이를 위해서는 미래에 대한 확신이 있어야 하며, 머지않아 예측이 현실로 나타나야 한다. 아무리 현명해도 먼 미래를 정확히 예측할 방법은 없다.

개발이 아닌 다른 분야에서 예를 살펴보자. CD는 1979년에 음악을 듣기 위해 카세트테이프를 교체할 목적으로 설계되었다. 20년 후에 같은 크기와 모양으로 DVD가 만들어져 제조사들이 컴퓨터에 CD/DVD 드라이버를 만들 수 있게 될지를 누가 알았겠는가? 그리고 CD를 CD-ROM 드라이브에서 읽을 때는 생각했던 회전 속도보다 50배 정도 빠른 속도로 회전하는 문제가 발생할지 누가 상상이나 했겠는가?

이것이 바로 소프트웨어 개발 등 엔지니어링 분야에도 '처리 원칙'이 있어야 하는 이유다. 처리 원칙을 잘 따르면 미래에 어떤 일이 일어나든 상관없이 잘 유지된다. 설계자로서 따라야 할 소프트웨어 설계의 법칙과 규칙이 바로 처리 원칙이다.

미래가 있다는 것을 기억하는 것이 중요하다. 그러나 미래를 예측해야 한다는 의미가 아니라 이 책에서 말하는 법칙과 규칙에 따라 의사결정을 해야 한다는 의미다. 그 미래가 무엇을 이끌어 내든 좋은 소프트웨어의 미래로 향하게 할 수 있기 때문이다.

특정 법칙이나 규칙이 미래에 독자에게 어떻게 도움이 될지 모두 예측하긴 어렵다. 하지만 분명히 도움을 줄 것이며 실제 작업에 적용할 수 있다는 점에 기뻐할 것이다.

이 장에서 본 법칙, 규칙, 사실에 동의하지 않더라도 좋다. 하지만 자신만의 결론을 꼭 내길 바란다. 그러나 이 법칙, 규칙, 사실을 따르지 않는다면 예측할 수 없는 미래 어딘가에서 거대한 문제와 맞닥뜨릴 것이다.