

당신의 소프트웨어 프로젝트가 **망하는 이유**

2판

박창욱

당신에게 이 책을 던진다.

머리말

당신의 소프트웨어 프로젝트가 망하는 이유는 멍청한 당신이 추진하기 때문이다. 이것이 이 책의 결론이다. 알았다면 이 책을 더 이상 읽을 필요가 없다. 덮어라.

수십 년간 IT¹ 업계에 있었던 나는 은퇴한 후 음식이 싸고 맛있는 동네에 가서 살자는 계획에 따라 베트남으로 이사했다. 그리고 일년내내 쏟아져 나오는 각종 열대과일과 새콤달콤한 베트남 요리를 배 터지게 먹으며 하루 종일 집안에 처박혀 낮잠을 자거나 인터넷 폐인 짓을 하며 살았다. 게으른 나에게는 바로 이것이 천국의 삶이었다. 그러나 이런 생활에도 시나브로 권태가 찾아와 뭔가 색다른 일거리를 만들어 보고자 여러 가지를 꾸몄는데, 그중 하나가 이 책을 쓰는 것이었다. 심심하면 아무 때나 쓸 수 있고 책을 쓰지 않아도 아무도 뭐라고 하는 사람이 없으니 늘어질 대로 늘어진 나의 베트남 생활에 적격인 일거리였다.

¹ 정보기술(Information Technology)의 약자다. 멍청한 당신도 아는 단어이므로, 더 이상의 자세한 설명은 생략한다.

나는 가끔 생각나는 대로 글을 쓰긴 했지만, 이 책의 출간 일정을 정해 놓지도 않았고 얼마나 진도가 나갔는지 신경을 쓰지도 않았다. 그러던 어느 날 출판을 해도 될 정도의 적당한 분량이 되었다는 것을 발견하고는 새로운 내용을 그만 쓰기로 하고 기존 내용을 다듬기 시작했다. 그러나 이 작업도 하염없이 늘어졌고, 나는 이 늘어짐을 즐겼다. 아무것도 하고 싶지 않다는 마음과 너무나도 심심하다는 마음이 내 안에서 서로 싸웠다. 그러다 어느덧 시간은 흐르고 또 흘러 출판 단계까지 왔다.

이 책을 처음 기획할 때는 소프트웨어 프로젝트를 성공적으로 완료하는 방법에 대해 글을 쓰겠다는 생각이었다. 소프트웨어공학을 전공하는 사람을 대상으로 하는 것이 아니라, 일반인을 대상으로 쉽게 이해할 수 있도록 글을 쓰는 것이 목표였다. 그런데 글을 쓰면 쓸수록 책의 내용이 공학에서 경영학으로 넘어갔다. 그만큼 소프트웨어 프로젝트를 성공으로 이끄는 것에는 공학적인 요소만이 아니라, 여러 다른 요소들이 복합적으로 작용한다는 것을 의미했다. 한편 책을 쓸수록 책에 적혀 있는 대로 따라한다고 성공이 보장되는 것이 아니라는 것을 깨닫고, 반대로 실패하는 이

유를 적기 시작했다. 성공하려면 실패하는 조건들을 모두 피하거나 극복해야 하는데, 말은 쉽지만 실제로는 쉽지 않아서 대부분 망한다. 쓰다 보니 망하는 이야기가 대부분을 차지해서 책 제목을 ‘망하는 이유’로 바꿨다.

이 책에서는 소프트웨어 프로젝트를 망하게 하는 여러 요인을 소프트웨어 개발 수명 주기의 기획과 요구사항, 설계, 구현, 운영의 각 단계로 나눠 살펴본다. 그리고 외주 개발 편에서는 외주 개발로 프로젝트를 진행할 때 발생하는 여러 요인을 살펴본다. 이를 근거로 이 책을 분류하면 소프트웨어 프로젝트의 위험관리에 특화된 책이다.

이 책을 쓰면서 깨달은 사실이 하나 있다. 나와 당신의 관계는 단순한 수주자와 발주자의 관계가 아니라, 애증의 관계였다는 사실이다. 나는 당신에게 그동안 하고 싶었던 말이 너무나 많았다는 사실을 이 책을 쓰면서 느꼈다. 그랬기에 이 책을 쓸 수 있었고, 진심으로 당신에게 감사한다. 당신이 없었다면 이 책은 세상에 나오지 못했다. 그냥 예상 하는 말이 아니라 정말이다. 그토록 나는 내 말을 듣지 않는 답답한 당신에게 짜증이 났다. 하지만 나는 당신의 주머니에 있는 돈을 꺼내기 위해서 어떤 방식이든 당신과 협

력해야만 했다. 그 결과 나는 지금 당신이 준 돈으로 콜라를 마시며 이 글을 쓴다. 고맙다.

이 책을 읽은 후 당신이 할 일은 내가 만든 소프트웨어를 당신이 처음 보았을 때와 똑같이 나에게 많은 수정 요구사항을 나열하는 것이다. 나는 당신의 그 앞뒤가 맞지도 않는 요구사항을 들으며 살던 시절이 가끔 그립다. 그래서 다시 한 번 당신의 요구사항을 듣고자 이 책을 썼다. 이 책은 당신의 요구사항에 따라 조금씩 변화할 것이다. 하지만 언제나 그랬듯이, 당신의 그 황당한 요구사항과는 별개로, 나는 내 마음대로 방향을 정해서 진행하겠다. 언젠가는 이 책이 당신의 터무니없는 요구사항 중 하나인 다음 주 복권 당첨 번호를 알려줄 수 있기를 나 또한 희망한다.

사이공 강의 동물을 바라보며 나는 오늘도 똥을 짐다.

박창욱

목차

머리말	3
목차	7
일반	12
당신은 누구인가?	13
나는 누구인가?	15
소프트웨어 프로젝트란 무엇인가?	17
망하는 것이 정상	19
표준화된 개발 방법론	22
소프트웨어 개발 수명 주기	25
폭포수와 애자일	29
애자일은 월급제, 외주는 폭포수	35
기획	42
기획이란 무엇인가?	43
망할 기획	46
타당성 조사	53
추정	57
ROI 함정	63

당신이 창업하면 망하는 이유	68
멍청하지만 사기를 당하고 싶지 않다	73
산 넘어 산의 또 다른 의미	76
초지일관 vs 조변석개	80
기획자. 99% 확률로 멍청이	84
컨설턴트. 99% 확률로 사기꾼	87
요즘 쓸만한 개발자 찾기 힘들어	92
개발은 최후의 수단	97

요구사항 103

요구사항이란 무엇인가?	104
요구사항 명세서 작성법	107
구체적 요구사항	109
스펙 말장난	112
기획자 vs 개발자. 어린이와 어른	117
예산과 기간, 기능이 이미 정해져 있다	123
최소의 예산으로 최소의 기능만을 개발	125
찔끔찔끔 투자하기	128

설계 131

설계란 무엇인가?	132
-----------	-----

설계 공간 탐색	136
연구와 개발은 같지만 다르다	140
아는 것이 없거나 권한이 없거나	145
당신의 눈에는 UI만 보인다	147
엔지니어 vs 디자이너. 철천지원수	149
총소유비용은 비밀	154
코꿰기	158
모든 문제를 해결할 신기술	163
프로토타입	166

구현	169
구현이란 무엇인가?	170
상세설계와 구현	173
진행률 99.8%	177
개발 상황을 엑셀로 정리해?	181
프로젝트 관리 소프트웨어	184
복불전문 마구잡이 개발자	188
개발자 번아웃	195
개발자를 추가해서 속도를 올려?	199
프로젝트 관리 삼각형	203

코드를 테스트하는 코드를 테스트	206
테스트 다 했어?	210
눈에 보이지 않는 품질	214
운영	220
운영이란 무엇인가?	221
출시하면 지옥이다	225
잘 실행되도록 유지	229
어제까지 잘되다가 오늘 갑자기 안되는데	232
안되면 되게, 되면 더 잘되게 보수	236
요구사항 vs 구현. 누구 잘못?	244
안돼요! 몰라요! 제가요?	247
백업	254
기술빚	260
리팩터링	266
외주 개발	272
외주 개발이란 무엇인가?	273
외주 관리를 못하면 자체 개발도 못한다	275
상주하며 개발하라고?	279
이거 해본 적 있어?	288

믿고 맡길 수 있는 사람	293
발주는 사장, 리베이트는 팀장	296
갑을병정	300
책임능력	308
보증보험	313
외주 개발 절차는 소송 준비 절차	317
소스코드 훔치기	327
코드 난독화	333
부록	342
남들이 말하는 망하는 이유	343
당신의 어록	351
종이책과 글꼴	355
꼬리말 - 2판	358
참고문헌	362

일반

당신은 누구인가?

이 책에서 지칭하는 당신이라는 사람은 미래에 소프트웨어 개발 프로젝트를 진행하려는 사람이거나, 소프트웨어 관련회사를 창업하려고 준비하는 사람이거나, 개발팀을 꾸리거나 외주로 소프트웨어 개발을 추진하려는 사람이거나, 비슷한 것들을 현재 추진하는 사람이거나, 과거에 이미 추진했다가 망한 경력이 있는 사람이다. 그리고 당신은 한 명의 프리랜서 개발자일 수도 있고, 몇 명이 모인 작은 외주 개발회사의 직원일 수도 있고, 스타트업의 사장일 수도 있고, 대형 SI 회사¹의 직원일 수도 있다. 과거에 나와 함께 일을 했던 동료일 수도 있고, 나를 개발자로 고용했던 관리자일 수도 있고, 나에게 컨설팅을 받았던 기획자일 수도 있다.

당신이 한때 프로젝트를 추진했던 경력자라면, 이 책을 읽으며 알게 된 사실을 프로젝트를 추진하던 그때도 알았더라면 정말 좋았을 것이라고 땅을 치며 후회할 수도 있다. 안타깝게도 과거를 되돌릴 방법은 없지만, 이와 비슷한

¹ SI는 System Integration의 약자다. 시스템 통합 회사라고도 한다. 하드웨어를 포함한 소프트웨어의 개발과 이와 관련된 각종 유지보수 등을 전문으로 하는 회사다.

사건이 미래에 되풀이되는 때를 대비할 수는 있다. 그래서 과거의 어디서부터 어떻게 잘못되었는지를 분석하고 기억한다면, 미래에 있을 비슷한 상황에서 과거와는 다른 선택을 해서, 과거와는 다른 미래를 만난다. 이것이 우리가 역사를 배우는 이유이기도 하다. 당신은 과거에 실패했지만, 미래에는 같은 실수를 반복하지 않기를 바란다.

당신이 인생에서 처음으로 소프트웨어 프로젝트를 시작하는데 이 책을 읽는다면, 당신에게는 큰 행운이다. 후회할 사건을 미리 방지할 기회가 왔으니 말이다. 하지만 과신하지 말아라. 당신에게는 낯선 세계로 첫발을 내딛는 것 이겠지만, 이 세계에서 수십 년을 팀굴었던 나의 눈에는 이렇게 보인다.

오늘도 불나방이 불을 향해 날아가는구나!

나는 당신이 불을 향해 날아가지 않기를 바라지만, 이미 당신의 귀에는 아무것도 들리지 않고, 당신의 눈에는 너무나 아름다운 성공의 불만 보인다. 그렇다. 그래야 당신답다.

나는 누구인가?

나는 당신의 외주를 받아 개발하던 개발자이기도 했고, 당신의 회사에 있던 개발팀의 일원이기도 했고, 당신이 고용했던 컨설턴트이기도 했고, 당신의 망해가는 프로젝트를 옆에서 지켜보던 친구이기도 했고, 당신에게 소프트웨어 공학을 가르쳤던 교수이기도 했다.

나는 지난 수십 년간 크고 작은 여러 프로젝트에서 다양한 역할을 맡았고, 갖가지 흥망성쇠를 지켜보았다. 나 역시도 많은 프로젝트를 성공적으로 끝내려고 노력했던 사람이었다. 프로젝트가 망하면 망한 이유를 분석해서 다음에는 절대로 이런 일이 반복되지 않게 만들고 싶었다. 하지만 프로젝트가 망하는 이유는 너무 많아서 아무리 반복을 피해도 늘 새로운 이유가 튀어나와서 망했다. 그 결과 모든 프로젝트에는 저마다의 망하는 이유가 있다는 것을 깨우쳤다. 그래서 이를 모두 피해서 성공하기는 마치 폭우 속에서 젓지 않고 마라톤을 완주하기처럼 비현실적이라고 생각했다. 생각이 여기에 다다르자, 노력하던 나는 사라졌고, 냉소적인 나만 남았다. 이젠 안 봐도 뻔하다. 오늘도 어디선가 프

로젝트가 망한다.¹ 망한 프로젝트를 보면 마치 톨스토이의 소설에 나오는 문장처럼² 다음과 같이 말하고 싶다.

**모든 흥한 프로젝트는 서로 비슷하지만,
망한 프로젝트는 저마다의 이유로 망한다.**

망할 이유를 모두 제거하면 성공한다. ‘안나 카레니나의 법칙’이다 (Diamond, 1997/2017). 가끔은 망하는 요소를 주렁주렁 매단 채로 성공한 프로젝트도 있지만, 망할 이유가 없는데 망한 프로젝트는 단 하나도 없었고 앞으로도 없다. 한때 나는 성공의 환상에 사로잡힌 멍청한 당신을 설득 하려고도 했지만, 당신과 실랑이하는 것마저 피곤해서 은퇴했고, 지금 이렇게 당신에게 그때 다 하지 못했던 이야기를 쓴다.

¹ “Somewhere today, a project is failing” (DeMarco & Lister, 2013, p. 3).

² ‘안나 카레니나’의 도입부. 모든 행복한 가정은 서로 비슷하지만, 불행한 가정은 저마다의 이유로 불행하다 (Толстой, 1877).

소프트웨어 프로젝트란 무엇인가?

소프트웨어 프로젝트는 소프트웨어를 개발하고 운영하는 것을 목표로 추진하는 프로젝트다. 이를 성공적으로 추진 하려면 공학의 관점에서 바라보는 소프트웨어공학의 지식도 필요하고, 경영학의 관점에서 바라보는 프로젝트의 관리 지식도 필요하다. 하지만 당신은 성공하겠다는 욕심만 있을 뿐, 관련 지식이 부족하여 프로젝트가 망한다.

어떻게 하면 소프트웨어를 잘 만들 수 있을지를 연구하는 학문이 소프트웨어공학이다 (Butterfield et al., 2016; Farley, 2021, pp. 3-9). 이 학문 자체는 컴퓨터와 관련된 공학의 한 분야지만, 소프트웨어 프로젝트가 망하는 이유를 찾는 것은 공학이 아니라 사회과학의 한 분야 같기도 하다. 소프트웨어 프로젝트가 망하는 데에는 공학적인 이유보다는 공학 외적인 요소가 더 많이 작용하기 때문이고, 소프트웨어 프로젝트를 관리하는 일은 일반적인 프로젝트 관리의 일종이고 경영학의 한 분야이기 때문이다.

소프트웨어 프로젝트의 관리에 관련된 서적은 이 분야의 원조라고 불리는 The mythical man-month (Brooks,

1975/1995)부터 경전이라 불리는 SWEBOK Guide (Bourque & Fairley, 2014)와 PMBOK Guide (Project Management Institute, 2021) 까지 많은 책이 시중에 나와 있고, 지난 수십 년간 출판된 논문들도 이루 헤아릴 수 없이 많다. 그러나 당신은 지식을 원한다고 주장하면서도 책을 읽을 시간은 없다고 주장할 것이니 더 이상 설명하지 않겠다.

다시 한 번 말한다. SWEBOK과 PMBOK에 적혀 있는 모든 내용을 이해하기 전까지 소프트웨어 프로젝트에 열씸도 하지 말아라. 당신이 소프트웨어 프로젝트를 추진하는 것은 초등학생이 달 탐사 프로젝트를 추진하는 것과 같다. 성공할 가능성이 아예 없고, 시작부터 망한다. 왜 망하는지 설명해달라고 하지도 말아라. 당신 같은 멍청이에게는 하나하나 설명하기조차 짜증이 난다.¹ 초등학생에게는 ‘나중에 커서 어른이 되면 알 수 있다’라고 타이르기라도 하겠지만, 이미 어른이 되었는데도 여전히 멍청한 당신에게는 이것도 통하지 않는다. 당신은 망해야 정신을 차린다. 아니, 정신을 차릴 때까지 계속 망해야 한다.

¹ 바로 이 책이 왜 망하는지 당신에게 설명하는 책이다. 수만 명에게 똑같은 대답을 여러 번 하기 귀찮아서 이 책을 썼다.

망하는 것이 정상

당신이 거액의 당첨금을 지급하는 복권을 하나 샀다고 가정해 보자. 당신은 그 복권에 당첨되길 간절히 원할 것이다. 나도 마음으로는 당신을 응원하지만, 객관적인 사실을 하나 말해주겠다. 확률적으로 보았을 때 당신이 복권에 당첨되지 않는 것이 대부분이고, 일반적이고, 예상했던 일이고, 흔한 일이고, 보통이고, 정상이다. 반대로 당신이 복권에 당첨되는 것은 매우 특별하고, 일어나기 힘든 일이고, 희귀하고, 이례적이고, 예외적이고, 비정상이다. 당신에게 좋은 일이 일어나는 것이 정상이고, 당신에게 좋지 않은 일이 일어나는 것이 비정상이라고 생각하면 큰 착각이다. 발생할 확률이 높은 일이 일어나는 것이 정상이고, 발생할 확률이 낮은 일이 일어나는 것이 비정상이다. 그러므로 당신이 복권에 당첨되지 못하는 것이 정상이다. 중요하니 다시 한번 말한다.

당신에게 좋은 일이 일어나는 게 정상이 아니라,
발생할 확률이 높은 일이 일어나는 게 정상이다.

이제 본론인 소프트웨어 프로젝트에 관해서 이야기 하자. 일반적으로 소프트웨어 프로젝트의 성공 확률은 절반도 되지 않는다. 성공 확률에 대한 조사는 여러 가지가 존재하고 조사마다 서로 의견이 다르지만, 여러 의견을 종합해 볼 때, 대략 30% 이하만이 원래 계획했던 대로 진행되고, 나머지는 프로젝트 자체를 중단하거나, 중단하지는 않았지만 초기의 계획과는 다른 결말에 이른다 (Fox & Patterson, 2021, p. 14). 관점에 따라서 프로젝트를 아예 중단한 것만 망했다고 볼 수도 있고, 처음에 계획했던 대로 일이 순조롭게 진행된 것만 성공했다고 보고 그렇지 않은 모든 것을 망했다고 볼 수도 있고, 비용과 기간이 늘어나고 내용이 바뀌어도 적당한 수준에서 타협하여 성공했다고 볼 수도 있기에, 성공을 판단하는 것도 주관적이다 (Glass, 2005). 이것은 마치 1년 안에 백만 명 모집하기 프로젝트를 시작했는데 98만 명만 모집했다면, 성공으로 봐야 하는지 실패로 봐야 하는지 애매한 것과 같다. 초기에 계획한 그대로 진행된 것만 성공이라고 엄격하게 정의한다면, 프로젝트 대부분은 실패한다는 것이 많은 조사들의 결론이다. 많이 봐줘서 상당한 사양 변경과 추가 비용으로 초심을 잃어버린 채 마무리된 프로젝트까지 성공으로 쳐준다고 해도, 성공 확률은 절반

이 되지 못한다. 그렇기에 소프트웨어 프로젝트를 시작하면 망하는 것이 대부분이고, 일반적이고, 예상했던 일이고, 흔한 일이고, 보통이고, 정상적인 결말이다.

다만 소프트웨어 프로젝트는 복권과 다른 점이 있는데, 프로젝트를 진행하면서 당신이 무엇을 선택하는가에 따라 미래가 바뀐다는 점이다. 그래서 당신의 역할이 매우 중요하고, 당신이 매 순간 올바른 선택을 한다면 프로젝트를 망하지 않게 할 수 있다. 그런데 당신은 명청해서 매순간 올바른 선택을 할 수 없으므로 프로젝트가 망하는 것은 분명하지만, 그나마 어떻게 해야 망하는 것을 최소화할 수 있을지 궁금할 것이다. 그렇다면 이어서 표준화된 개발 방법론부터 이야기를 시작한다.

표준화된 개발 방법론

소프트웨어의 개발을 당신이 인류 최초로 시도하는 것이 아니다. 수많은 사람이 수많은 개발을 이미 경험했다. 그중 성공한 것도 있고 실패한 것도 있다. 그래서 사람들은 어떻게 하면 망하지 않게 할지 고민하며 각자의 개발 방법론을 만들기 시작했다. 하지만 여기에도 사람마다 의견 차이가 있어, 저마다 자기의 개발 방법론이 최고라고 주장하며 서로 싸운다. 그런 싸움의 결과, 많은 개발 방법론 중 인기를 끌며 명맥을 유지해 나가는 것도 있고, 아예 자취를 감춰버린 것도 있다 (Selby, 2007). 나는 이런 방법론 중에서 오랜 시간 동안 살아남은 표준화된 개발 방법론을 개발의 정석이라고 부르고 싶다. 법에 따라 강제로 정석을 따라야 하는 것도 아니고, 정석대로 하면 반드시 성공하는 것도 아니다. 하지만 망할 확률을 낮추는 방법은 정석을 따르는 것이고 그것이 정석의 존재 이유다.

인간은 무지로 인해 많은 실수를 저지르며, 그런 실수를 되풀이하지 않으려고 교육이 존재한다. 하지만 교육 자체를 무시할 정도로 무식한 인간이 존재한다면, 토론이나 설득 따위는 필요 없고, 그저 가두어 놓고 이해할 때까

지 강제로 가르치거나, 그냥 내버려 두는 방법밖에는 할 수 있는 것이 없다. 여러 회사를 방문하면 크건 작건 그 회사만의 독특한 개발 문화를 발견할 수 있고, 때로는 관리자들이 자기 회사만의 개발 문화라며 자랑스럽게 세미나장에서 발표할 때도 있는데, 나의 눈에는 이것이 전혀 좋게 보이지 않고, 외부와 단절되어 살아가는 원시인 마을의 독특한 습성처럼 보인다. 게다가 표준화된 개발 방법론을 무시하고 자신들만의 독창적인 방법론이라며 추진하다가 망하기를 여러 번 반복한 이후에, 그것을 개인과 집단이 성장해 나가는 과정이라며 미화하는 사람들도 있다. 그런 사람을 볼 때마다 가두어 놓고 강제로 가르치고 싶은 마음이 든 적이 여러 번이다.

표준화된 절차를 따르는 것은 구습을 그대로 답습하는 것과 같을 수도 있고 다를 수도 있다. 이러한 표준화된 절차가 어떻게 해서 나오게 되었는지 이해하지 못한 상태에서 그저 따라만 한다면, 그것은 옳고 그름을 분별하지 않고 답습하는 것에 불과하다. 자신이 아직 경험하지 못한 문제점이지만, 과거에 같은 문제점을 겪은 사람들이 여러 시행착오 끝에 만든 해결책이라는 관점에서 표준화된 절차를

바라본다면, 이를 이해하기 좀 더 쉬울 것이다. 그리고 해결책인 개발 방법론은 하나가 아니라 여러 개다. 방법론마다 장단점이 있으니, 충분히 이해한 다음에 선택하고 프로젝트를 추진할 것을 권장한다.

당신은 표준화된 절차를 알고 싶지도 않고, 알 필요도 없다고 생각하며, 당신의 방식으로 추진하겠다고 주장할 수 있다. 당신의 프로젝트이므로 당신의 마음대로 추진할 자유가 당신에게 있고, 나는 당신의 선택을 존중한다. 누구나 방법론을 만들 수 있으므로, 당신도 당신만의 방법론을 만들 수 있다 (Glass, 2002, pp. 164-165). 당신의 방법론이 소프트웨어 분야에 혁신을 이룰 수도 있다. 하지만 과거에 이렇게 주장하며 프로젝트를 추진했던 거의 모든 사람들이 망했다는 사실만은 당신에게 꼭 알려주고 싶다.¹

¹ “한 가지 분명하게 확인할 수 있는 사실은, 실패했던 거의 모든 프로젝트에서 관리를 담당했던 자들이 개발 방법론에 대한 이해가 천박하고 조야했다는 점이다. 거의 모든 실패한 프로젝트의 공통점이다” (이호종, 2011, 페이지: 79).

소프트웨어 개발 수명 주기

소프트웨어 개발 수명 주기(Software Development Life Cycle; SDLC)라는 용어¹가 있다 (Everett & McLeod, 2007, pp. 29-58). 당신에게는 생소한 용어일 것이다. 소프트웨어를 개발할 때 그 과정들을 나열한 것이다. 더욱 상세하게 나누 기도 하지만 ‘기획 → 설계 → 구현 → 운영’의 순으로 대략 진행되고, 이것을 계속 반복하는 것이 소프트웨어 개발 수명 주기다. 좁은 의미로 설계와 구현만 소프트웨어 개발이라 할 때도 있지만, 보통 소프트웨어 개발이라 하면 SDLC 전체를 말한다.

마찬가지로, 좁은 의미로 설계하고 구현하는 사람을 소프트웨어 개발자라 하지만, 넓은 의미로 개발자는 SDLC에 관련된 모든 사람을 말한다. 즉, 기획자, 설계자, 코더, 테스터, 운영자, 관리자, 디자이너, 기술 문서 작성자, 보안 담당자, 백업 담당자, 간식 담당자 등등도 모두 개발자다. 이렇게 보면 당신도 뭘 하든지 상관없이 개발자다.

¹ 소프트웨어에 추가로 하드웨어를 포함하는 개념인 ‘시스템 개발 수명 주기 (Systems Development Life Cycle)’라는 단어가 더 포괄적인 표현이다 (Satzinger et al., 2012, pp. 5-6).

기획 단계에서는 어떤 기능이 있는 제품을 만들지 시장조사를 하고 회의하고 이를 토대로 제품이나 서비스에 필요한 요구사항 명세서를 작성한다. 설계 단계에서는 요구사항 명세서를 분석해서 어떻게 하면 효율적으로 이 조건들을 만족하는 제품을 만들 수 있을지 여러 설계 후보를 비교해서 최적의 설계를 선택하고, 그 결과인 설계 문서를 작성한다. 구현 단계에서는 설계 문서에 따라 여러 하드웨어를 제작하고, 프로그래밍 언어로 코드를 작성하고, 이것이 설계와 일치하는지 테스트를 수행한다. 운영 단계에서는 사용자에게 시스템을 알려주고, 상담에 응해주고, 고장 없이 잘 동작하도록 유지보수, 관리, 감시, 백업, 장애 대응 등의 업무를 수행하고, 이 과정에서 발견한 문제점들을 기록하고 해결한다.

이런 방식으로 소프트웨어의 개발을 진행한다는 것을 당신도 이미 알거나, 처음 듣는다 해도 순순히 받아들인다. 하지만 이것을 계속 반복한다는 것을 모르거나 받아들이길 거부한다. 당신은 한번 개발하면 계속 사용할 수 있을 것이라 착각한다. 하지만 인간의 욕망은 ‘앉으면 눕고 싶고, 누우면 자고 싶다’는 속담처럼 끝이 없기에, 구현을 마치고

운영을 시작하자마자 뭔가를 뜯어고치고 싶고, 이에 따라 바로 다음 버전의 기획을 시작한다. 그리고 이 새로운 기획에 따라 새로운 소프트웨어를 설계하고 구현하고 운영한다. 그리고 이것을 반복한다. 다시 말하면 인간의 욕망이 존재하는 한 소프트웨어의 개발을 멈출 수 없다.

이런 특징으로 인해 소프트웨어 개발은 시작이 있고 끝이 있는 직선적 세계관이 아니라, 끝이 곧 시작이 되는 돌고 도는 윤회적 세계관이다. 그래서 용어에도 ‘수명 주기’라는 단어가 들어갔다. 현재 소프트웨어 개발을 한 번에 끝낼 수 있고 주기를 돌 필요가 없다고 주장하는 전문가는 아무도 없다. 단지 이 주기를 어떻게 도는 것이 가장 효과적인가를 놓고 다양한 개발 방법론을 각자 주장할 뿐이다. 중요한 개념이니 다시 한 번 강조한다.

**소프트웨어 개발은
한 번에 끝나지 않고 계속 반복된다.**

영화를 보면 악당이 과학자들을 가두어 놓고 인류가 파멸할 엄청나게 위험한 것을 개발하는 장면이 흔히 등장한다. 이런 영화에 빠지지 않는 것은 개발이 완료되자마자

악당이 개발 관련자들을 모두 제거하는 장면이다. 이건 악당이 잔인한 것이 아니라 명청한 것이다. 그 어떤 개발도 이런 식으로 똑딱 완료할 수 없고, 어찌어찌 완료했다 해도 관련자 모두를 제거하고 나면 운영할 방법이 없다. 악당이 스스로 유지 보수와 차기 버전 개발 등을 차단하는 짓이니 최악의 자충수다. 영화에서 악당이 하는 짓은 웃고 넘어갈 수 있지만, 현실에서 당신이 이렇게 생각한다면 웃고 넘어갈 수 없는 문제다. 영화에서의 악당처럼 현실에서의 당신도 망하는 결말에 도달한다.

폭포수와 애자일

앞에서 언급한 소프트웨어 개발 수명 주기의 한 주기를 어떻게 진행할 것인지에 따라 여러 개발 방법론이 존재한다. 이들을 크게 분류하면, 처음부터 완벽하고도 철저하게 계획하여 단 한 번에 제대로 끝내면 주기를 반복할 필요조차 없다는 엄격하면서도 앞뒤가 꽉 막힌 고집불통 이론인 폭포수 방법론(waterfall model), 이와 정반대로 아무런 사전 계획 없이 무작정 주기를 뻥뻥이 돌자는 화끈하면서도 될 대로 되라는 식의 자유분방한 이론인 애자일 방법론(agile model), 그 밖에 이 두 가지 극단적인 방법론의 중간에 있는 반복적이고 점진적인 모델(Iterative and Incremental Development model; IID), 나선형 모델(spiral model), V모델(V model) 등의 여러 방법론이 존재한다. 애자일 방법론도 세부적으로 극단적 프로그래밍(Extreme Programming; XP), 스크럼(scrum), 칸반(kanban) 등의 여러 모델이 있지만, 뒤에서 상세히 설명

하기로 하고, 지금은 간략히 이들 모두를 애자일이라 칭하기로 한다.¹

결론부터 말하자면 당신의 모든 프로젝트에 적용할 최고의 방법론은 없고,² 당신이 어떤 방법론을 선택해도 망한다는 사실은 변치 않는다. 그 이유는 명청한 당신이 추진하기 때문이라고 머리말에서 밝혔다. 방법론마다 장단점이 있어 이를 잘 살펴보고 선택해야 하고, 방법론을 선택한 다음에는 그 방법론에 맞춰서 행동해야 할 것과 행동하지 말아야 할 것이 있는데, 당신은 이를 이해하고 따를 능력도 없고 의지도 없어서 망한다. 이 책에서는 개별 방법론의 장단점을 모두 비교하기보다는 한 주기의 길이가 서로 극단적인 폭포수와 애자일 두 개를 간략하게 비교한다.

¹ IID와 나선형 모델, V 모델은 모두 폭포수 방법론의 가지치기 방법론이라며 깎아내리는 사람들도 있다. 반대로 최초의 애자일 방법론인 ‘극단적인 프로그래밍(Extreme Programming; XP)’ 방법론 (Beck, 1999)도 IID 방법론의 반복 주기를 극단적으로 짧게 하자는 방법론이므로, XP 방법론이 IID 방법론이라는 주장도 있다 (Sommerville, 2015, p. 77). 방법론 종교 전쟁이다.

² “As I have said, there is no universal process model that is right for all kinds of software development” (Sommerville, 2015, p. 46).

폭포수 방법론은 개발 수명 주기의 한 주기가 가장 긴 방법론이다. 폭포에서 한 번 떨어진 물은 다시 위로 돌아갈 수 없듯이 각 단계에서 완벽하게 한 다음, 다음 단계로 넘어가는 방식이다 ([Sommerville, 2015, p. 47](#)). 기획 단계에서 완벽하게 기획하고, 이 완벽한 기획에 따라 완벽하게 설계하고, 완벽한 설계에 따라 완벽하게 구현하고, 완벽하게 운영하면 모든 게 완벽하다는 이론이다. 구현하다가 문제가 생겼다는 이유로 설계를 고치는 게 허용되지 않고, 설계가 어렵다는 이유로 기획을 수정하지 않는다. 행동하기 전에 충분히 생각하는 방식이기에 미래에 발생할 문제들을 먼저 예상하고 제거할 수 있다는 점이 장점이다. 반대로 불완전한 인간은 미래에 발생할 모든 일을 내다보며 각 단계를 완벽하게 끝낼 수 없고, 오직 전지전능한 신만이 할 수 있다는 점이 단점이다.

애자일 방법론은 폭포수 방법론과 정반대다. 계획이 없는 게 계획이다. 게다가 제대로 된 문서도 없다. 농담이 아니다 ([Fox & Patterson, 2021, pp. 5-17; Sommerville, 2015, pp. 73-75](#)). 계획도 없고 문서도 없고 손에 잡히는 대로 일하는 게 애자일 방법론의 규칙이다. 언제나 최우선 당면 과제를 쳐

리하는 데에 집중하는 방법론이다. 20세기에 이런 방식으로 개발한다고 하면 한심한 놈들이라며 세상이 비웃었다. 그런데 21세기에 와서는 소프트웨어공학 교재에 당당히 등장하는 주목받는 이론이 되었고, 반대로 폭포수 방법론으로 개발한다고 하면 비웃음을 당하는 세상이 되었다. 왜냐하면 애자일 방법론은 폭포수 방법론에 있는 문제점을 많이 해결해 주었기 때문이다. 애자일 방법론의 최대 장점은 개발 수명 주기의 한 주기가 극단적으로 짧다는 점으로, 변화하는 고객의 요구사항에 따라 변화된 제품을 고객에게 가장 먼저 제공할 수 있다. 고객 만족의 관점에서 보면 가려운 곳을 바로바로 긁어주니 최고의 방법론이다.

여기까지 보면 애자일 방법론이 만능처럼 보인다. 그러나 애자일 방법론에 최대의 단점이 있으니, 그것은 바로 장기계획이 없다는 점이다.¹ 고객이 버튼을 빨간색으로 칠해달라면 빨간색으로 칠하고, 다음 날 고객이 파란색으로 칠해달라면 다시 파란색으로 칠하는 것이 애자일 방법론이

¹ 애자일과 애자일이 아닌 방법론을 구별하는 방법이 계획의 존재 여부다. 애자일 방법론이 등장한 이후에는 애자일이 아닌 방법론을 뭉뚱그려 ‘계획 주도 개발(Plan Driven Development; PDD)’이나 ‘계획과 문서가 있는 방법론(Plan and Document Methodology)’이라 부른다.

다. 심지어 고객이 도로 빨간색으로 칠해달라면 또다시 빨간색으로 칠하는 방식이다. 문제가 보이는가? 그렇다. 고객의 변덕이 가장 큰 문제고, 이에 따른 인력과 시간의 낭비가 애자일 방법론의 가장 큰 단점이다.

이와 반대로 폭포수 방법론에서는 고객과 충분한 회의를 해서 버튼을 어떤 색으로 할지를 사전에 정한다. 그리고 절대로 마음을 바꾸지 않겠다는 고객의 협서를 받아낸 다음에¹ 개발에 착수하는 방식이다. 그렇기에 개발에 착수한 이후에는 고객이 아무리 색을 바꿔 달라고 해도 들은 척도 하지 않고 사전에 약속한 대로만 개발한다. 개발이 계획대로 진행되기에 개발력의 낭비가 없다. 하지만 고객의 관점에서 보면, 자신이 한때 원하기 했지만, 더 이상 원하지 않는 소프트웨어의 개발이 진행된다는 것이 문제다.

다시 한번 강조하지만, 당신의 프로젝트에 적합한 개발 방법론은 정해진 것이 없다. 고민하는 당신에게 누군가 찾아와서 요즘 뜨는 아주 좋은 새로운 개발 방법론이라며 추천한다면, 그것은 거짓말이니 들을 필요도 없다. 모든 면

¹ 서로 협의한 개발 범위를 고객이 최종 승인하는 행위다. 이 이후에는 개발을 완료할 때까지 추가로 요구하거나 변경할 권리가 고객에게 없다.

에서 다 좋은 방법론은 없다. 그러니 나에게 하나를 골라달라고 하지 말아라. 그래도 프로젝트를 추진하려면 이 중의 하나로 정해야 하니 제발 골라달라고 나에게 부탁한다면 이렇게 대답하겠다. 세상이 아무리 바뀌어도 처음에 생각한 대로 밀고 나가는 초지일관 고집불통에게는 폭포수 방법론이 어울리고, 반대로 세상의 변화에 민감하게 반응하며 늘 최적의 생존 전략을 생각하는 조변석개 미꾸라지에게는 애자일 방법론이 어울린다. 이제 당신이 어떤 사람인지 스스로 생각해서 선택해라.

하지만 당신은 명청해서 개발 방법론을 이해하지도 못하니, 어떤 개발 방법론을 선택해도 그대로 따라 하지 못하고, 당신만의 ‘주먹구구 방법론’¹으로 변질된다. 이와 반대로 방법론을 무작정 따라야 하고, 조금이라도 어긋나면 이단으로 여기는 놈인 ‘방법론 맹신자(true believer)’도 프로젝트를 망치는 또 다른 부류다 (DeMarco et al., 2008, pp. 30-31). 그래서 당신의 프로젝트가 망한다.

¹ Ad hoc, chaotic process (McConnell, 2006, pp. 41-42).

애자일은 월급제, 외주는 폭포수

소프트웨어를 개발하는데 애자일 방식으로 하자고 주장하는 사람들의 바탕이 되는 철학인 ‘애자일 소프트웨어 개발 선언문’ (Beck et al., 2001)을 보자.

우리는 소프트웨어를 개발하고, 또 다른 사람의 개발을 도와주면서 소프트웨어 개발의 더 나은 방법들을 찾아가고 있다. 이 작업을 통해 우리는 다음을 가치 있게 여기게 되었다.

공정과 도구보다 개인과 상호작용을
포괄적인 문서보다 작동하는 소프트웨어를
계약 협상보다 고객과의 협력을
계획을 따르기보다 변화에 대응하기를

가치 있게 여긴다. 이 말은, 왼쪽에 있는 것들도 가치가 있지만, 우리는 오른쪽에 있는 것들에 더 높은 가치를 둈다는 것이다.

당신이 이 철학에 동의하거나 말거나 그건 당신의 자유이므로 나는 간섭하지 않겠다. 하지만 선언문의 왼쪽에 있는 것과 오른쪽에 있는 것은 서로 반대되는 것이므로 양

립할 수 없다는 사실에는 동의해라. 애자일로 개발하면 오른쪽에 있는 가치를 얻기 위해 왼쪽에 있는 가치를 희생해야 하고, 애자일이 아닌 방법으로 개발하면 왼쪽에 있는 가치를 얻기 위해 오른쪽에 있는 가치를 희생해야 한다. 다시 한 번 강조한다. 왼쪽 오른쪽 둘 다를 가질 수 없다.

애자일 방법론은 무작정 시작하고 끝없이 개발하는 방식이다. 개발계획이 정해져 있다 하더라도 변화에 대응하는 것을 더 큰 가치로 여기기에 기존의 계획을 무시하고 매일 수정한다. 그래서 개발비도 사전에 책정할 수 없다. 그러므로 애자일 방법론으로 외주 개발을 진행한다면, ‘이 프로젝트 완료까지 비용은 얼마’라는 식으로 계약하는 것은 앞뒤가 맞지 않고, ‘이 프로젝트를 진행하는 동안 한 달에 얼마’라는 식으로 계약해야 한다.

현재 우리나라에서 사용하는 소프트웨어 하도급 표준 계약서¹는 폭포수 모델을 기반으로 한다. 계약하는 시점에 업무의 범위를 미리 지정하고, 지정된 그 업무를 지정된 기간 내에 수행하고, 제대로 수행했는지 검사 후 계약 관계

¹ 여러 정부 부처와 공공기관, 민간 단체에서 서로 다른 것을 제시하기에 표준이 여러 개라고 말할 수도 있고, 표준이 존재하지 않는다고 말할 수도 있다.

를 종료하는 모델이다. 요구사항, 설계와 구현, 검수의 절차로 이어지는 전형적인 폭포수 방법론에 기반한 계약서다.¹ 계약할 때는 계약을 이행해야 하는 대상을 명확하게 지정해야 한다. 아니면 싸움이 난다 (細川義洋, 2013/2014). 그러므로 만약 애자일 방법론에 따라 외주 개발을 하면서 이 계약서대로 계약하려면, 명확하게 지정된 대상에 대해서만 계약할 수 있으니, 스크럼²에서는 스프린트마다 계약서를 작성해야 하고, 칸반³이라면 작업 지시서마다, TDD⁴라면 테스트케이스마다 계약서를 작성해야 하니, 매일 계약서만 쓰다 날이 샐 것이다. 이쯤 되면 소프트웨어 개발이 아니라 계약서 작성 놀이라고 불러야 한다.

¹ 어떤 정신나간 표준 계약서는 기간과 금액을 정해서 계약부터 하고, 요구사항 명세서를 나중에 만들도록 지시한다. 백지 계약을 표준 계약이라 한다. 이놈들이 미쳤는지, 이놈들을 미쳤다고 하는 내가 미쳤는지 잘 모르겠다. ‘외주 개발 절차는 소송 준비 절차’ 꼭지를 참조해라.

² Scrum. 무엇을 할 지 모여서 회의를 한 다음, 일주일 정도 각자 정해진 일을 하는 것을 반복하는 방식. 각자 정해진 일을 하는 기간을 스프린트라 한다.

³ Kanban. 작업 지시서를 벽에 붙여 놓고 아무나 그 할 일을 가져가서 수행하는 방식이다.

⁴ Test Driven Development. 테스트케이스를 먼저 작성한 다음, 그 테스트케이스를 통과하도록 개발하는 방식이다.

또한 애자일 방법론은 기획팀과 개발팀, 기타 관련자들 전체가 뜰뜰 뭉쳐서 협력하는 것을 전제로 한다. 이런 유토피아 같은 상황에서는 애자일이 아니라 오자일이라도 성공한다. 외주로 개발한다는 것은 서로 협력할 생각이 전혀 없는 발주자와 수주자가 만나서 호랑이와 사자가 만난 것처럼 서로 으르렁거리며 계약을 협상하는 것부터 시작한다. 그리고 서로 주고받을 것에 대한 범위를 정하는 계약서를 작성하고, 이 계획된 절차에서 조금이라도 벗어나면 서로 소송하는 것이 일반적이다.¹ 애자일 철학과 전혀 맞지 않는 상황이다. 따라서 철저한 계획에 따라 실행하는 방식인 폭포수 방법론을 적용하는 것이 합당하다.

개발 범위를 정하고 예산을 정한 후에 애자일로 개발하는 것은 애자일의 기본 철학을 무시하는 태도다. 앞으로 어떻게 될지 모르는 상황에서 계속 갈 데까지 가 보자는 것이 애자일의 기본 철학이다. 따라서 애자일로 개발할 때는 개발 범위와 예산을 미리 알 수 없다는 것이 전제조건이므로, 개발자의 투입 시간에 따라서 예산을 집행하는 것이 타당하다. 그리고 애자일 방법론으로 개발하는 개발자의 처

¹ ‘외주 개발 절차는 소송 준비 절차’ 꼭지를 참조해라.

지에서는 내일 회사에 출근하면 어떤 일을 하게 될지 모르는 상황이다. 이것이 애자일 방법론에서는 정상이다. 목표가 없는 상황이기에, 얼마나 왔는지, 어디로 가는지도 모른 채 그저 열심히 개발하는 것이 개발자의 역할이다. 그렇기에 투입한 시간에 따라 임금을 지급하는 월급제가 타당하다.

무엇을 만들어야 하는지 명확하지 않은 상태지만, 일단 시작해서 하루라도 빨리 뭔가를 보여줘야 하는 상황인데, 요구사항 자체도 상황에 따라 계속 변하고, 변하는 요구사항에 즉각 대응해야 하고, 팀 구성이 열 명 이내의 소규모라면 애자일 방법론이 딱 들어맞는 환경이다. 말하고 보니 매일 오락가락하면서 성질도 급한 당신에게 어울리는 방법론이다. 그래서 당신이 애자일을 신봉할 수도 있다. 하지만 애자일로 개발을 시작하면 개발팀이 뭔가를 열심히 하는 것 같기는 한데, 언제 완성될지 알 수 없고, 꼬박꼬박 계산되는 그들의 월급으로 인해 비용은 당신의 예산을 초과한 지 한참이 지났지만, 이제 와서 되돌리기도 어렵다. 이 모든 것의 원인은 끝없이 변하는 당신의 요구사항이지만, 당신이 요구사항을 변경했다고 스스로 인정할 리는 없

고, 단지 세세한 부분에서 기존의 모호했던 요구사항을 구체화하기만 했다고 주장할 것이다. 그러므로 도대체 이 프로젝트가 왜 이 지경이 되었는지 당신은 이해할 수 없다. 안타깝지만 그래서 당신의 프로젝트가 망한다.

당신이 애자일 방법론에 대해 여기까지 충분히 이해해서 월급제를 시행했다고 해도 더 큰 문제가 또 발생한다. 다른 분야도 비슷하겠지만, 잘난 개발자는 못난 개발자보다 대략 열 배 정도 잘한다 (DeMarco & Lister, 2013, p. 44; Glass, 2002, pp. 14-15; Prechelt, 2019). 이 상태에서 모든 개발자에게 똑같은 임금을 지급하는 것은 바보 같은 정책임을 당신도 이해하리라고 본다. 각자의 역량에 따라 차등 지급함이 타당한데, 문제는 당신이 그들의 역량을 제대로 평가하지 못하면서 차등 지급할 때 생긴다. 당신은 전문가가 효율적으로 빠르게 일처리를 하면 원래 쉬운 일처럼 보여서 돈을 덜 주고, 초보자가 오랫동안 헤매다가 겨우 해내면 수고했다며 돈을 더 주는 경향이 있다 (Ariely & Kreisler, 2017, pp. 131-148). 게다가 인사를 잘한다는 점, 항상 웃는 얼굴이라는 점, 제 시간에 출근한다는 점, 복장이 단정하다는 점, 매일 야근한다는 점, 젓가락질을 잘한다는 점 등등 개발 성과와 상관없

는 요소로 개발자를 평가한다. 그 결과 당신에게 해준 것에 비해 받은 것이 적은 고급 개발자들이 떠나고, 당신에게 해준 것에 비해 받은 것이 많은 쓰레기 개발자만 당신 곁에 남아 꼬리를 흔든다.¹ 그러면 그 이후의 진행을 더 볼 필요도 없이 망하는 외길이다. 직접 개발할 능력이 없기에 다른 사람에게 개발시키려는 당신에게 개발자의 역량을 평가하는 능력²까지 요구하는 것은 가혹할 수 있다. 하지만 이것이 현실이다. 그래서 당신의 프로젝트가 망한다.

¹ 당신을 추앙하는 사람들에게 에워싸이면, 당신은 왕 노릇 하며 한세월을 보내게 되고, 프로젝트는 결다리가 된다. 하지만 당신이 행복하다면 OK.

² 개발자의 역량을 어떻게 평가해야 할지 인류는 아직 답을 찾지 못했다. 현재는 어떻게 평가해도 문제가 있다. 미래에는 답을 찾길 바란다. 아니, 미래에는 개발자가 모두 인공지능으로 대체되어 답을 찾을 필요조차 없길 바란다.

부록

남들이 말하는 망하는 이유

소프트웨어 프로젝트가 망하는 이유에 대해서 지금까지 내 생각만 말했다. 이것이 나만의 망상이자 편향된 의견이라고 당신이 무시할 수도 있으니, 다른 사람들의 의견도 여기에 적는다.

첫 번째 자료다. ‘왜 소프트웨어가 실패하나?’라는 제목의 글 (Charette, 2005)이다. 여기에서는 소프트웨어가 실패하는 여러 요인을 다음과 같이 나열했다.

비현실적인 목표, 빗나간 비용 예측, 엉뚱하게 정의된 요구사항, 답답한 프로젝트 상태 보고, 위험관리 부재, 고객과 개발자 간 소통 부재, 실험적인 기술도입, 프로젝트 복잡도 처리능력 부재, 엉망진창 개발 방법, 한심한 프로젝트 관리, 이해관계자들의 정치, 시장의 압박

구구절절 옳은 말씀이다. 그리고 이렇게 나열한 것 중 한두 가지로 인해 망하는 것이 아니라, 여러 가지 복합적인 상호작용으로 인해 망한다고 저자는 말한다. 이렇게 나열하고 보니 프로젝트가 망하는 이유를 다루는 학문은 경영학일 수도 있겠다는 생각이 듈다. 소프트웨어 프로젝트를

관리하는 것은 소프트웨어에 관한 것이기도 하지만, 일반적인 프로젝트 관리에 관한 것이기도 하기 때문이다. ‘무식하면 용감하다’라는 속담처럼, 잘 모를 때는 프로젝트를 쉽게 보고 덤벼드는 경향이 있지만, 이 모든 것을 다 극복해야만 소프트웨어 프로젝트가 성공한다는 것을 이해한 다음에는 프로젝트를 시작하기 두려운 것이 당연하다.

두 번째 자료다. ‘소프트웨어 프로젝트가 실패하는 14가지 흔한 이유’라는 제목의 기사다 ([Forbes Technology Council, 2020](#)). 이 기사에서는 소프트웨어 프로젝트가 망하는 이유로 다음의 14 가지를 나열했다.

업무의 필요성을 이해하지 못함, 우선순위 결정에 합의를 못함, 실행 전략이 불분명함, 고객을 생각하지 않음, 모호한 요구 사항, 만병통치약을 기대, 따로 노는 개발과 영업, 경직된 개발 팀, 교통 정리와 상세 계획이 없음, 애매한 역할 분담, 너무 딱 맞는 것을 기대, 경험 부족, 끝없는 설계변경, 사공이 많음, 가시적인 것에만 집착

첫 번째 자료가 공학적인 측면에서 접근했다면 두 번째 자료는 좀 더 경영학적인 측면에서 접근했다. 세부적인 면에서 보면 두 자료의 내용이 서로 다르지만, 큰 틀에서 보

면 서로 비슷하다. 이 기사는 한 사람이 쓴 글이 아니라 여러 사람이 작성한 글이라 서로 반대인 것처럼 보이는 사항이 있다. 그 예로, 고객의 요구는 이미 변경되었는데 과거의 요구사항대로 만드는 경직된 개발팀 때문에 망한다는 주장도 있고, 반대로 고객의 요구에 따라 끝없이 설계를 변경해서 망한다는 주장도 있다. 둘 다 옳은 말이다. ‘초지일관 vs 조변석개’ 꼭지에서 이 딜레마를 다루었다.

세 번째 자료다. ‘프로젝트 실패의 일곱 가지 이유’라는 제목의 글 (Discenza & Forman, 2007)이고 설문조사를 통한 연구 결과물이다.

비즈니스 가치가 아닌 세부적인 기술에 집중, 측정된 결과에 대한 책임소재가 불분명, 일관된 중간 점검 절차가 없음, 기획과 실행에 대한 일관된 방법론이 없음, 처음부터 고객이 참여해 프로젝트를 같이 만들어 가는 과정이 없음, 인재들에게 동기부여를 하는 데 실패, 팀 구성원에게 생산성을 높일 도구나 기술을 제공하지 않음

설문조사를 통한 연구 결과이므로 여러 프로젝트에서 가장 흔히 나타나는 망하는 이유다. 반대로 말하면 이 일곱 가지만 피해도 높은 확률로 성공할 수 있다. 그러나 당

신은 이 중 단 하나도 피할 능력이 없고 의지도 없어서 당신의 프로젝트가 망한다. 아니, 이보다 당신이 이 내용을 이해할 수나 있을지가 나는 더 궁금하다.

네 번째 자료다. ‘소프트웨어 스펙의 모든 것’이라는 책의 1장 1절에 있는 내용이다 (김익환 & 전규현, 2021, 페이지: 26-27).

이외에도 실패 원인은 끝도 없이 많은데 이를 크게 나누면 스펙, 팀, 관리, 고객, 기술 등으로 분류할 수 있다. 내가 이중에서 가장 중요하게 생각하는 것은 ‘스펙’이다.

저자가 말한 것처럼 실패 원인은 끝도 많다는 것에도 동의하고, 스펙이 가장 중요하다는 것에도 동의한다. 그러나 스펙의 중요성을 무시한 사람도 당신이고, 스펙을 제대로 만들지 않은 사람도 당신이고, 스펙을 제대로 만들라고 지시하지 않은 사람도 당신이고, 스펙이 제대로 되어 있는지 확인하지 않은 사람도 당신이고, 스펙을 아무 때나 바꾼 사람도 당신이고, 스펙과 실제 개발이 따로 놀게 내버려 둔 사람도 당신이다. 그래서 프로젝트의 실패 원인은 바로 당신이다.

다섯 번째 자료다. ‘시스템을 잘 만들게 하는 기술’이라는 책의 머리말에 있는 내용이다 (白川克 & 浜本佳史, 2021/2023, 페이지: XI).

사실 ‘시스템을 만드는 사람’의 문제로 이런 실패가 일어나는 경우는 드물다. 대부분은 ‘시스템을 만들게 하는 사람’, 즉, 지금 이 책을 손에 들고 있는 여러분 같은 사람이 그 원인이다.

이 책의 저자도 ‘시스템을 만들게 하는 사람’, 바로 당신이 원인이라고 지목한다.¹ 나는 나와 뜻을 함께하는 사람이 있어 외롭지 않고 기쁘다. 물론 당신도 당신 같은 사람이 전 세계에 있어 최소한 외롭지는 않다.

여섯 번째 자료다. ChatGPT (OpenAI, 2024)에게 ‘소프트웨어 프로젝트가 망하는 이유를 신나게 알려줘’라고 요청했을 때의 응답이다.

¹ 다짜고짜 ‘시스템’이라고 하면 무엇을 말하는지 모호하므로, 명확하게 ‘IT 시스템’이라고 쓰는 것이 좋고, 이 책의 내용으로 보면 ‘IT 시스템 프로젝트’가 제일 어울린다. 저자가 지칭한 ‘만드는 사람’은 ‘개발자, 또는 수주자’라는 뜻이고, ‘만들게 하는 사람’은 ‘기획자, 또는 발주자’라는 뜻이다. 이 저자의 책은 외주 개발을 전제로 이야기를 진행하므로, 이를 종합하면 ‘시스템을 만들게 하는 사람’은 ‘IT 시스템 프로젝트 발주자’다. 책의 제목도 ‘IT 시스템 프로젝트 발주 기술’이 더 어울린다.

자, 이건 좀 더 친숙한 언어로 소프트웨어 프로젝트의 실패 이유를 신나게 알려줄게!

- 1. 요구 사항이 계속 바뀌어!:** “어머, 그건 참 멋있겠다!”라고 생각한 요구 사항이나 기능이 나중에 변경되면 어떡해? 유저들은 내가 뭐를 만들고 있는지 정말로 알고 있는 건가?
- 2. 팀원들끼리 소통이 안 돼!:** 야, 너가 뭐 하고 있니? 이 프로젝트는 왜 이렇게 지체돼? 만약 팀원들이 서로 얘기를 안 하면, 프로젝트는 점점 더 망가져 버릴 거야!
- 3. 코드가 엉망이야!:** 이 코드는 뭔가? 히힛, 나는 내가 한 부분만 알아도 돼! 이러다가 나중에는 ‘코드 지옥’에 갇히게 될 거야!
- 4. 테스트가 부족해!:** 이거 돌아가길 바랄게! 테스트를 거의 안 하고 코드를 썼다가는, 나중에 대형 참사가 일어날 수 있어!
- 5. 느리게 움직여!:** 세상은 계속 변하고 있어! 만약 프로젝트가 아직도 과거에 머물러 있다면, 우리는 결국 그 뒤를 따라가지 못할 거야!
- 6. 누가 결정을 내야 돼?:** 우리가 모두 함께 일하는 건 좋지만, 누가 이 프로젝트를 이끌고 있나? 결정권이 없으면, 우리는 미로 속에서 방황할 거야!

ChatGPT는 소프트웨어가 망하는 이유를 이렇게 나열한 다음, 이런 위험을 피하는 방법도 이어서 다음과 같이 알려줬다.

소프트웨어 프로젝트가 망할 수 있는 여러 가지 방법이 있지만, 다행히도 우리는 이런 위험들을 피할 수 있어! 제대로된 계획과 소통, 품질 관리, 그리고 유연한 대처 방식이 있으면, 우리는 실패 대신 성공을 만날 수 있어! 함께 신나게 도전해 보자구!



실패 대신 성공을 만날 조건은 망하는 이유와 반대라는 사실을 ChatGPT도 안다. 제대로 된 계획으로 요구사항이 계속 바뀌는 것을 막고, 소통으로 팀원들끼리 소통이 안 되는 것을 막고, 품질 관리로 엉망인 코드와 부족한 테스트를 막고, 유연한 대처 방식으로 느리게 움직이는 것을 막으라는 말이다. 뻔한 말이지만 이 말을 믿고 ‘실패 대신 성공을 만나려고 신나게 도전’하는 바보짓을 하지 말아라. 당신이 도전하면 언제나 실패를 만난다. 내가 보장한다.

¹ ‘제대로된’의 정확한 띄어쓰기는 ‘제대로 된’이지만, 응답을 수정하지 않고 그대로 적었다. 마찬가지로 ‘ (로켓 이모지)’와 비표준어인 ‘보자구’도 응답에 나온 그대로 적었다.

이 외에도 다른 책을 보거나 인터넷을 검색하면 소프트웨어 프로젝트가 망하는 여러 가지 이유의 목록을 쉽게 찾을 수 있다. 몇개를 찾다 보면 다들 비슷하다는 것을 알게 될 것이다. 미래에 더욱 발전된 인공지능에게 ‘인간이 추진하는 소프트웨어 프로젝트가 망하는 이유’를 물어봐도 비슷한 답변을 할 것이다. 이런 이유 중에서 소프트웨어 프로젝트에만 나타나는 일부 이유를 제외하면, 소프트웨어가 아닌 일반적인 프로젝트가 망하는 이유에도 적용될 수 있다. 그러므로 조금 더 부풀려 말하면, 이런 이유는 당신의 모든 사업이 망하는 이유다.¹

¹ ‘당신이 창업하면 망하는 이유’ 꼭지에서 자세하게 설명한다.

당신의 어록

이 책의 곳곳에 당신이 했던 명언이 많이 있다. 그 밖에 당신이 나에게 했던 말 중 내 기억에 남은 멋진 말을 여기에 적는다. 뭐가 문제인지 아는 사람에게는 설명할 필요가 없고, 뭐가 문제인지 모르는 사람에게는 설명하기조차 귀찮으니, 설명은 없다.

이거 진짜 좋은 사업이니까 같이 하자.

이번에 진짜 획기적인 아이디어가 떠올랐는데, 미래를 알려주는 사업이야. 내가 많은 사람에게 물어봤는데, 다들 제품이나 오기만 하면 가격과 상관없이 무조건 사겠다는 거야. 이건 무조건 성공하는 사업이고, 떼돈 버는 사업이야. 같이 하자. 절대 다른 사람들에게 말하지 말고.

그건 내가 누구보다도 잘 알지.

내가 이 업계에만 삼십 년 있었어. 사람들이 뭘 원하는지는 내가 누구보다도 잘 알지. 이 프로젝트는 무조건 성공할 거야. 나만 믿으라고.

이거 개발하려면 돈이 얼마나 들어?

미래를 알려주는 소프트웨어를 개발하려면 돈이 얼마나 들어? 한달 안으로 개발하려면 개발자 몇 명 필요해?

간단해 보이는데 뭐가 그렇게 어렵다고 그래?

내가 아주 복잡한 걸 원하는 게 아니라, 그냥 미래에 어떤 일이 일어날지 알려주기만 하면 되는 거야. 간단해 보이는데 뭐가 그렇게 어렵다고 그래?

안 된다고만 하지 말고 되게 해줘.

미래를 알려주는 기능인데, 다들 해주지는 않으면서 안 된다고만 해. 안되는 이유도 말 안해주고. 도대체 왜 그래? 하기 싫어서? 귀찮아서? 하여간 이거 좀 되게 해줘.

다른 회사는 할 수 있다는데?

다들 미래를 알려주는 기능을 못 만든다고 했는데, 오늘 서울역 앞에서 노숙자한테 물어보니까 선금 백만 원만 주면 바로 해준다는데?

사소한 문제가 있는데, 이것만 좀 해결해줘.

미래를 알려주는 소프트웨어를 다음 달 출시해. 가격도 싸고, 사용하기도 편하고, 홍보도 잘해서 사전구매자도 엄청나게 많아. 그런데 딱 하나, 예측이 자꾸 틀리는 사소한 문제가 있는데, 이것만 좀 해결해줘.

제품만 있으면 영업은 자신 있어.

경쟁사보다 성능이 뛰어나고, 기능도 많고, 사용하기 편리한 멋진 제품을 개발하기만 해. 그럼 영업은 자신 있어. 최소한 백만 개 판다. 내가 보장하지.

해 달라는 것만 달랑 해 오면 어쩌라는 거야?

내가 물에서 건져 달랬다고 달랑 건져만 주면 어쩌라는 거야? 내 보따리는? 갈아입을 옷은? 헤어드라이어는커녕 수건도 준비 안 했어? 이런 건 말하지 않아도 기본이지. 내가 이런 것까지 하나하나 다 말해 줘야 하나?

이거 언제쯤 제대로 쓸 수 있어?

개발을 시작한 지 벌써 오 년이 지났고, 최근 삼 년 동안 계속 쓰면서 문제를 찾는 중인데, 지난달에도 또 새로운 문제가 나왔어. 이거 언제쯤 제대로 쓸 수 있어?

지금은 급하니까 우선 시작해.

지금은 급하니까 지도는 나중에 찾고 우선 남쪽으로 출발해. 서울이 남쪽에 있는 건 확실하니까.

좋은 건 알겠는데 지금은 급하니까 다음에 하지.

지금 서울까지 가는데 하도 걸어서 다리가 너무 아파. 나 좀 업어줄 수 있어? 뭐라고? 버스? 그게 뭔데? 그런데 그거 타려면 이미 지나온 길을 5 킬로미터 거꾸로 돌아간 다음 거기서 내일 아침까지 멍하니 기다려야 한다고? 좋은 건 알겠는데 지금은 급하니까 다음에 하지. 최소한 열흘은 더 걸어가야 하니 업어주기 싫으면 다리나 좀 주물러 주고 가.

종이책과 글꼴

나는 이 책을 전자책으로도 만들었고 종이책으로도 만들었다. 그리고 종이책에는 상대적으로 매우 높은 가격을 책정했다. 화면으로 책을 읽는 것은 의미가 없으며 책이란 무릇 종이에 인쇄된 상태로 보아야 제맛이라고 생각하는 당신에게 청구하는 일종의 징벌적 가격이면서, 동시에 당신이 이렇게 비싼 책을 책꽂이에 꽂아 놓을 만큼 이 책을 사랑한다는 것을 누군가에게 보여주거나, 아무나 가질 수 없는 것을 소유했다는 자부심을 주는 고급화 전략이다. 이 종이책을 갖는 것은 당신이 매우 명청하다는 것을 의미하거나, 진정으로 종이책을 사랑하는 사람임을 증명하거나, 돈이 넘쳐 나 쓸데가 없어 고민하는 사람임을 보여주거나, 내가 생각하지 못한 또 다른 이유다. 무엇이든지 당신의 목적에 부합하기를 바란다.

당신이 이 책을 서재에 꽂아 놓고 자랑거리로 삼는다면, 그러한 당신이 존재한다는 것이 또한 나의 자랑이다. 당신의 책에 저자 친필 서명도 해주고 함께 기념 촬영도 하고 싶다. 이것이 진정한 상생이고 협력이다.

화면을 통해 이 책을 보는 사람은 마음대로 확대와 축소를 할 수 있으므로, 글자의 크기에 대해 신경 쓸 일이 없다. 단지 종이에 인쇄된 상태로 보기를 좋아하는 사람은 대부분 당신처럼 나이를 먹어서 눈이 침침해져 작은 글자가 잘 안 보이는 경향이 있기에, 그런 당신을 배려해서 종이책의 본문 글꼴 크기를 16 포인트(point)¹로 했다. 읽는 데에 불편함이 없길 바란다.

나는 SIL OFL², 또는 이와 비슷한 공개 정책에 따라 배포된 글꼴로만 이 책을 만들려고 노력했다.³ 이 책의 대부분을 차지하는 글꼴은 ‘노토(Noto)’ 글꼴이다. 최대한 많은 글자를 표현하려는 그들의 노력을 인정하는 뜻으로 사용했다고 거창하게 포장하고 싶지 않고, 단지 글꼴을 구매하는 데에 돈을 쓰기 싫은 것이 이유다. 좀 더 정확히 말하면, 공짜 글꼴보다도 못한 영망진창인 글꼴을 돈까지 내서 쓰는 걸 내 자존심이 허락하지 않았다.

¹ 출판 업계에서 사용하는 단위. 16 포인트는 글자의 높이가 약 5.6 mm다.

² SIL International 공개 글꼴 저작권(Open Font License; OFL). 누구나 자유롭게 글꼴을 사용하고, 배포, 수정할 수 있는 저작권이다.

https://en.wikipedia.org/wiki/SIL_Open_Font_License

³ 1판을 출판할 때는 성공했으나 2판을 출판할 때는 실패해서 못내 아쉽다.

나는 이 책의 본문의 글꼴을 정할 때 ‘부리(serif)¹’가 있는 글꼴과 부리가 없는 글꼴 사이에서 끝없이 갈등했다. 나는 부리가 왜 존재하는지 잘 이해하지 못하는 부리 무용론자다. 바꿔 말하면 나는 아무 기능이 없는 쓸데없는 것을 왜 달아놓는지 이해하지 못하는 엔지니어다. 하지만 본문과 제목의 글꼴을 다르게 하고 싶은 욕망이 있었는데, 부리 없는 글꼴들은 서로 비슷하게 생겼기에, 제목과 본문의 글꼴 둘 중 하나는 부리가 있는 글꼴을 골라야만 했다. 그런데 부리가 있는 글꼴이 본문에 쓰이면 책이 보기 싫은 글꼴로 가득 차서 싫었고, 제목에 쓰이면 보기 싫은 글꼴이 큼지막하게 보여 더더욱 싫었다. 하여간 이런 고민의 결과는 지금 보는 이 상태다. 앞으로는 또 어떻게 마음이 바뀌어 어떻게 변경할지 나도 모른다. 그러다 문득 이런 고민은 내가 그토록 저주하던 디자이너들이나 하던 짓임을 깨닫고는 자괴감이 들었다.²

¹ 글자에 붙이는 장식용 작은 선. <https://en.wikipedia.org/wiki/Serif>

² ‘엔지니어 vs 디자이너. 철천지원수’ 꼭지를 참조해라.

꼬리말 - 2판

나는 2017년부터 이 책을 썼다. 2023년 초에 이 책이 베이퍼웨어¹임을 깨닫고, 일정 우선주의를 선택해, 출판일을 2023년 6월 1일로 고정했다. 출판일에는 백지라도 출판한다는 계획이었다. 일정을 맞추려면 뭔가를 희생해야 했다.² 그래서 구현, 운영, 외주 개발 편을 통째로 걷어냈다. 책의 내용이 절반으로 줄었지만, 그 덕택에 할 일도 절반이 되어, 일정에 맞춰 출판할 수 있었다. 출판도 프로젝트이므로 ‘첫 번째 버전에 너무 공들이지 말라’라는 나의 조언에 따랐다.³

1판을 인쇄했고 예상대로 많은 오류가 있었다. ‘첫 번째 버전의 결과에 실망하지 말라’라는 나의 조언에 따라 실망하지 않았다. 오타나 문법적 오류는 참을 수 있었지만, 그중 ‘참고문헌에 같은 책이 두 번 포함된 오류’는 참을 수 없는 오류라, 즉시 수정해서 1판 2쇄를 찍었다.

¹ Vaporware. 곧 나온다고 말하지만, 차일피일 미루며 나오지 않는 물건이다.

² ‘프로젝트 관리 삼각형’ 꼭지를 참조해라.

³ ‘산 넘어 산의 또 다른 의미’ 꼭지를 참조해라.

2판을 쓸 때는 1판에 빠진 편들을 채우고, 품질이 만족스러우면 출판한다는 품질 우선주의를 선택했다.¹ 예상 출판일은 2023년 12월 1일이었다. 하지만 이날까지 품질이 기대에 미치지 못해, 출판일을 2024년 3월 1일로 변경했다. 그리고 이날에도 품질이 기대에 미치지 못해, 다시 한 번 출판일을 2024년 6월 1일로 변경했다. 그동안 품질이 많이 올라가기도 했고, 오류가 없는 책을 출판하는 것은 불가능하므로,² 더 이상 출판일을 미루지 않기로 했다.

2판을 출판하는 지금, 품질이 떨어져 2판에 포함되지 못한 내용이 약 300페이지어치 있다. 이 중에는 바로 포함해도 괜찮은 품질의 꼭지도 있고, 아무런 맥락 없이 당신에 대한 욕으로만 가득 찬 한심한 꼭지도 있다. 2판을 읽다 보면 가끔 ‘어떤 꼭지를 참조해라’라고 적혀 있지만 그 꼭지가 이 책 어디에도 없을 때가 있다. 이것이 품질이 떨어져 빠진 꼭지의 흔적이다. 잘 다듬고 순화해서 3판에 추가하겠다. 하지만 3판을 언제 출판할지는 나도 모른다. 2판에서 출판 프로젝트를 마칠 수도 있고, 1년에 한 번씩 주기적

¹ 품질이 의미하는 것이 너무 많아 뭐라고 딱 정의할 수 없다. ‘눈에 보이지 않는 품질’ 꼭지를 참조해라.

² ‘테스트 다 했어?’ 꼭지를 참조해라.

으로 출판하는 방식으로¹ 3판, 4판이 계속 나올 수도 있다. 내가 당신에게 할 말이 바닥나면 이 책에 더 이상 추가할 내용이 없겠지만, 지금의 기세로는 10판까지도 너끈하다. 하지만 이 책을 쓰는 일보다 훨씬 더 즐거운 다른 일이 많이 생겨, 이 책의 존재조차 잊고 사는 삶이 내가 가장 바라는 미래다.

나는 이 책을 출판하려고 2019년 출판사를 차렸다. 멀쩡한 출판사라면 이따위 책을 출판할 리 없기도 하고, 이 책에 출판사의 입김이 작용하는 것도 싫었고, 심심해서 하나 만들었다. 이 출판사는 뭘 해보려는 의지도 없고, 일하는 직원도 없고, 이렇다 할 실적도 없고, 고작 이따위 책이나 출판한다. 그러니 당연히 망한다. 하지만 창업의 목적이었던 이 책의 출판을 두 번이나 달성했으니 이제 폐업해도 된다. 이렇게 폐업하면 성공인지 실패인지 나도 잘 모르겠다.²

¹ 주기적인 출시(periodic release). 사전에 주기를 정하고, 이에 맞춰 계속 출시하는 방식이다. 소프트웨어 업계에서 사용자에게 업데이트 서비스를 할 때 주로 사용한다.

² ‘당신이 창업하면 망하는 이유’ 꼭지를 참조해라.

이 책에서 지금까지 발견된 오류를 확인하거나, 새로운 오류를 발견했거나, 3판이 나왔는지 궁금하면 이 책의 맨 마지막 페이지에 있는 DOI¹ URL을 이용해라. 만약 정보를 찾을 수 없다면, ‘안돼요! 몰라요! 제가요?’라고 투덜 대지 말고, 스스로 안되는 이유를 알아내고 잘 해결하길 바란다.² 이 책의 정보가 사라졌고, 어디에서 찾을 수 있는지 아무런 단서조차 없다면, 당신과 나는 서로 다른 시대에 산다. 모쪼록 잘 지내길 바란다.

¹ 디지털 객체 식별자(Digital Object Identifier).

² ‘안돼요! 몰라요! 제가요?’ 꼭지를 참조해라.

참고문헌

- Adžić, G. (2011). *Specification by example: How successful teams deliver the right software*. Manning.
- Akerlof, G. (1970). The market for lemons: Quality uncertainty and the market mechanism. *Quarterly Journal of Economics*, 84(3), 488-500. <https://doi.org/10.2307/1879431>
- American Psychological Association. (2020). *Publication manual of the American Psychological Association* (7th ed.).
<https://doi.org/10.1037/0000165-000>
- Appelo, J. (2010). *Management 3.0: Leasing agile developers, developing agile leaders*. Addison-Wesley.
- Ariely, D., & Kreisler, J. (2017). *Dollars and sense: How we misthink money and how to spend smarter*. HarperCollins.
- Aurum, A., & Wohlin, C. (2005). *Engineering and managing software requirements*. Springer. <https://doi.org/10.1007/3-540-28244-0>
- Bass, L., Clements, P., & Kazman, R. (2021). *Software architecture in practice* (4th ed.). Addison-Wesley.
- Beck, K. (1999). *Extreme programming explained: Embrace change*. Addison-Wesley.
- Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for agile software development*. Agile Manifesto. <https://agilemanifesto.org/>
- Beinerts, L. (2014, 3 24). *The expert (short comedy sketch)* [Video]. YouTube. <https://www.youtube.com/watch?v=BKorP55Aqvg>
- Bentley, J. (1985). Programming pearls: Bumper-sticker computer science. *Communications of the ACM*, 28(9), 896-901.
<https://doi.org/10.1145/4284.315122>
- Boehm, B. (2006). A view of 20th and 21st century software engineering. *Proceedings of the 28th International Conference on Software Engineering*, (pp. 12-29).
<https://doi.org/10.1145/1134285.1134288>
- Bourque, P., & Fairley, R. E. (Eds.). (2014). *Guide to the software engineering body of knowledge (SWEBOK®): version 3.0* (3rd ed.). IEEE Computer Society Press.

- Brooks, F. P. (1995). *The mythical man-month: Essays on software engineering* (20th anniversary ed.). Addison-Wesley. (Original work published 1975)
- Brooks, F. P. (1987). No silver bullet - Essense and accidents of software engineering. *IEEE Computer*, 20(4), 10-19.
<https://doi.org/10.1109/MC.1987.1663532>.
- Brown, W. J., Malveau, R. C., McCormick, H. W., & Mowbray, T. J. (1998). *AntiPatterns: Refactoring software, architectures, and projects in crisis*. Jone Wiley & Sons.
- Brumby, D., Janssen, C., & Mark, G. (2019). How do interruptions affect productivity? In C. Sadowski, & T. Zimmermann (Eds.), *Rethinking productivity in software engineering* (pp. 85-107). Apress. https://doi.org/10.1007/978-1-4842-4221-6_9
- Butterfield, A., Ngondi, G. E., & Kerr, A. (Eds.). (2016). *A dictionary of computer science* (7th ed.). Oxford University Press.
<https://doi.org/10.1093/acref/9780199688975.001.0001>
- Cadle, J., & Yeates, D. (2008). *Project management for information systems* (5th ed.). Pearson Education.
- Carr, E. H. (1961). *What is history?* Macmillan.
- Cervantes, H., & Kazman, R. (2016). *Designing software architectures: A practical approach*. Addison-Wesley.
- Charette, R. N. (2005). Why software fails [Software failure]. *IEEE Spectrum*, 42(9), pp. 42-49.
<https://doi.org/10.1109/MSPEC.2005.1502528>
- Chemuturi, M. (2012). *Requirements engineering and management for software development projects*. Springer.
<https://doi.org/10.1007/978-1-4614-5377-2>
- Christensen, C. M. (1997). *The innovator's dilemma: When new technologies cause great firms to fail*. Harvard Business School Press.
- Comuzzi, M., Grefen, P., & Meroni, G. (2023). *Blockchain for business: IT principles into practice*. Routledge.
- Dale, E. (1969). *Audiovisual methods in teaching* (3rd ed.). Dryden Press.
- Davis, A. M. (2005). *Just enough requirements management: Where software development meets marketing*. Dorset House.
- DeMarco, T., & Lister, T. (2013). *Peopleware: Productive projects and teams* (3rd ed.). Addison-Wesley.

- DeMarco, T., Hruschka, P., Lister, T., McMenamin, S., Robertson, J., & Robertson, S. (2008). *Adrenaline junkies and template zombies: Understanding patterns of project behavior* (2nd ed.). Dorset House.
- Deming, E. W. (2018). *Out of the crisis* (Reissue ed.). The MIT Press. (Original work published 1982)
- Deming, E. W. (2018). *The new economics for industry, government, education* (3rd ed.). The MIT Press.
<https://doi.org/10.7551/mitpress/11458.001.0001>
- Denne, M., & Cleland-Huang, J. (2004). The incremental funding method: Data-driven software development. *IEEE Software*, 21(3), 39-47. <https://doi.org/10.1109/MS.2004.1293071>
- Diamond, J. (2017). *Guns, germs, and steel: The fates of human societies* (20th anniversary ed.). W. W. Norton & Company. (Original work published 1997)
- Dimitrov, D. (2019). *Software project estimation: Intelligent forecasting, project control, and client relationship management*. Apress. <https://doi.org/10.1007/978-1-4842-5025-9>
- Discenza, R., & Forman, J. B. (2007). Seven causes of project failure: How to recognize them and how to initiate project recovery. *PMI Global Congress* (p. 15). Project Management Institute.
- Eilam, E. (2005). *Reversing: Secrets of reverse engineering*. Wiley Publishing.
- Ekström, D. (2022). *How to think about software development: About software, software development and software developers*. Dan Ekström.
- Erder, M., & Pureur, P. (2016). *Continuous architecture: Sustainable architecture in an agile and cloud-centric world*. Morgan Kaufmann.
- Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, 2(3), 17-23. <https://doi.org/10.1109/6294.846201>
- Everett, G. D., & McLeod, R. (2007). *Software testing: Testing across the entire software development life cycle*. John Wiley & Sons.
- Faella, M. (2020). *Seriously good software: Code that works, survives, and wins*. Manning.
- Fairbanks, G. (2010). *Just enough software architecture: A risk-driven architecture*. Marshall & Brainerd.
- Fang, W., Chen, W., Zhang, W., Pei, J., Gao, W., & Wang, G. (2020). Digital signature scheme for information non-repudiation in

blockchain: A state of the art review. *EURASIP Journal on Wireless Communications and Networking*, 2020:56.

<https://doi.org/10.1186/s13638-020-01665-w>

Farley, D. (2021). *Modern software engineering: Doing what works to build better software faster*. Addison-Wesley.

Feathers, M. C. (2004). *Working effectively with legacy code*. Prentice Hall.

Feinleib, D. (2011). *Why startups fail: And how yours can succeed*. Apress. <https://doi.org/10.1007/978-1-4302-4141-6>

Fenton, N., & Bieman, J. (2014). *Software metrics: A rigorous and practical approach* (3rd ed.). CRC Press.

Firesmith, D. C. (2014). *Common system and software testing pitfalls: How to prevent and mitigate them: descriptions, symptoms, consequences, causes, and recommendations*. Addison-Wesley.

Forbes Technology Council. (2020). *14 common reasons software projects fail (and how to avoid them)*. <https://www.forbes.com>

Ford, N., Parsons, R., & Kua, P. (2017). *Building evolutionary architectures*. O'Reilly Media.

Ford, N., Richards, M., Sadalage, P., & Dehghani, Z. (2021). *Software architecture: The hard parts*. O'Reilly Media.

Foster, E. C., & Towle, B. A. (2021). *Software engineering: A methodological approach* (2nd ed.). CRC Press.

Fournier, C. (2017). *The manager's path: A guide for tech leaders navigating growth and change*. O'Reilly Media.

Fowler, M. (2018). *Refactoring: Improving the design of existing code* (2nd ed.). Addison-Wesley.

Fox, A., & Patterson, D. (2021). *Engineering software as a service: An agile approach using cloud computing* (2.0b7th ed.). Pogo Press.

Glass, R. L. (2002). *Facts and fallacies of software engineering*. Addison-Wesley.

Glass, R. L. (2005). IT failure rates - 70% or 10-15%? *IEEE Software*, 22(3), 112,110-111. <https://doi.org/10.1109/MS.2005.66>

Gobeli, D. H., Koenig, H. F., & Bechinger, I. (1998). Managing conflict in software development teams: A multilevel analysis. *Journal of Product Innovation Management*, 15(5), 423-435.
<https://doi.org/10.1111/1540-5885.1550423>

Gray, J. (2012). *Men are from Mars, Women are from Venus: The classic guide to understanding the opposite sex* (20th anniversary ed.). Quill. (Original work published 1992)

- Greever, T. (2020). *Articulating design decisions: Communicate with stakeholders, keep your sanity, and deliver the best user experience* (2nd ed.). O'Reilly Media.
- Heagney, J. (2016). *Fundamentals of project management* (5th ed.). American Management Association.
- Heričko, T., & Šumak, B. (2023). Exploring maintainability index variants for software maintainability measurement in object-oriented systems. *Applied Sciences*, 13(5), 2972.
<https://doi.org/10.3390/app13052972>
- Hodges, J. L. (2019). *Software engineering from scratch*. Apress.
<https://doi.org/10.1007/978-1-4842-5206-2>
- Hofstadter, D. R. (1999). *Gödel, Escher, Bach: An eternal golden braid* (20th anniversary ed.). Basic Books. (Original work published 1979)
- Hohpe, G. (2020). *The software architect elevator: Redefining the architect's role in the digital enterprise*. O'Reilly Media.
- Humble, J., & Farley, D. (2010). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Addison-Wesley.
- Humble, J., Molesky, J., & O'Reilly, B. (2015). *Lean enterprise: How high performance organizations innovate at scale*. O'Reilly Media.
- Hunt, A. (2008). *Pragmatic thinking and learning: Refactor your wetware*. Pragmatic Bookshelf.
- ISO/IEC/IEEE. (2017). *ISO/IEC/IEEE 24765:2017 Systems and software engineering - Vocabulary*.
<https://doi.org/10.1109/IEEEESTD.2017.8016712>
- ISO/IEC/IEEE. (2018). *ISO/IEC/IEEE 29148:2018 Systems and software engineering - Life cycle processes - Requirements engineering*.
<https://doi.org/10.1109/IEEEESTD.2018.8559686>
- ISO/IEC/IEEE. (2022). *ISO/IEC/IEEE 14764:2022 Software engineering - Software life cycle processes - Maintenance*.
<https://doi.org/10.1109/IEEEESTD.2022.9690131>
- Jerome, B., & Kazman, R. (2005). Surveying the solitudes: An investigation into the relationships between human computer interaction and software engineering in practice. In A. Seffah, J. Gulliksen, & M. Desmarais (Eds.), *Human-centered software engineering — Integrating usability in the software development lifecycle* (pp. 59-70). Springer. https://doi.org/10.1007/1-4020-4113-6_4

- Jones, C. (2009). *Software engineering best practices: Lessons from successful projects in the top companies*. McGraw-Hill.
- Jordan, A. (2013). *Risk management for project driven organizations: A strategic guide to portfolio, program and PMO success*. J. Ross Publishing.
- Jorgensen, P. C., & DeVries, B. (2021). *Software testing: A craftsman's approach* (5th ed.). CRC Press.
- Kamsties, E. (2005). Understanding ambiguity in requirements engineering. In A. Aurum, & C. Wohlin (Eds.), *Engineering and managing software requirements* (pp. 245-266). Springer.
https://doi.org/10.1007/3-540-28244-0_11
- Khan, A. A., & Le, D.-N. (Eds.). (2022). *Evolving software processes: Trends and future directions*. Wiley-Scrivener Publishing.
- Koch, A. S. (2004). *Agile software development: Evaluating the methods for your organization*. Artech House.
- Kruchten, P., Nord, R., & Ozkaya, I. (2019). *Managing technical debt: Reducing friction in software development*. Addison-Wesley.
- Laplante, P. A., & Kassab, M. H. (2022). *Requirements engineering for software and systems* (4th ed.). CRC Press.
<https://doi.org/10.1201/9781003129509>
- Le, C. D. (2010). *Life of a software engineer in Vietnam*. CreateSpace Independent Publishing Platform.
- Lelek, T., & Skeet, J. (2022). *Software mistakes and tradeoffs: How to make good programming decisions*. Manning.
- Love, A. B. (2017). *IT project management: A geek's guide to leadership (Best practices in portfolio, program, and project management)*. CRC Press.
- Malamidis, G. (2009). The ROI variable. In R. Monson-Haefel (Ed.), *97 things every software architect should know* (pp. 128-129). O'Reilly Media.
- Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Addison-Wesley.
- McConnell, S. (1997). *Software project survival guide*. Microsoft Press.
- McConnell, S. (1998). Problem programmers. *IEEE Software*, 15(2), 128,127,126. <https://doi.org/10.1109/52.663801>
- McConnell, S. (2004). *Code complete* (2nd ed.). Microsoft Press.
- McConnell, S. (2006). *Software estimation: Demystifying the black art*. Microsoft Press.

- Measey, P. (2015). *Agile foundations: Principles, practices and frameworks*. BCS.
- Medinilla, Á. (2014). *Agile kaizen: Managing continuous improvement far beyond retrospectives*. Springer. <https://doi.org/10.1007/978-3-642-54991-5>
- Michaels, P. (2022). *Software architecture by example: Using C# and .NET*. Apress. <https://doi.org/10.1007/978-1-4842-7990-8>
- Mills, H. D. (1988). *Software productivity*. Dorset House. (Original work published 1983)
- Nygard, M. T. (2018). *Release it!: Design and deploy production-ready software* (2nd ed.). Pragmatic Bookshelf.
- O'Toole, G. (2019, 5 21). *The opposite of love is not hate, but indifference*. Quote Investigator®.
<https://quoteinvestigator.com/2019/05/21/indifference/>
- Ogheneovo, E. E. (2014). On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 2(14), 1.
<https://doi.org/10.4236/jcc.2014.214001>
- Oncken, W. (2000). *Monkey business: Are you controlling events or are events controlling you?* Executive Excellence Publishing.
- OpenAI. (2024). ChatGPT (Version 3.5) [Large language model].
<https://chatgpt.com>
- O'Regan, G. (2023). *Mathematical foundations of software engineering: A practical guide to essentials*. Springer.
<https://doi.org/10.1007/978-3-031-26212-8>
- Pavicevic, T., Tomasevic, D., Bucaioni, A., & Ciccozzi, F. (2023). Conflicts between UX designers, front-end and back-end software developers: Good or bad for productivity? In L. Shahram (Ed.), *ITNG 2023 20th International Conference on Information Technology - New Generations* (pp. 161-171). Springer. https://doi.org/10.1007/978-3-031-28332-1_19
- Petzold, C. (2022). *Code: The hidden language of computer hardware and software* (2nd ed.). Microsoft Press.
- Pezzè, M., & Young, M. (2008). *Software testing and analysis: Process, principles and techniques*. John Wiley & Sons.
- Pfleeger, S. L., & Atlee, J. M. (2009). *Software engineering: Theory and practice* (4th ed.). Prentice Hall.
- Pirie, M. (2015). *How to win every argument: The use and abuse of logic* (2nd ed.). Bloomsbury Publishing.

- Platt, D. S. (2007). *Why software sucks ... and what you can do about it.* Addison-Wesley.
- Poller, H. (2010). *Bewältigte Vergangenheit. Das 20. Jahrhundert, erlebt, erlitten, gestaltet.* Ozlog.
- Poppendieck, M., & Poppendieck, T. (2013). *The lean mindset: Ask the right questions.* Addison-Wesley.
- Prechelt, L. (2019). The mythical 10x programmer. In C. Sadowski, & T. Zimmermann (Eds.), *Rethinking productivity in software engineering* (pp. 3-11). Apress. https://doi.org/10.1007/978-1-4842-4221-6_1
- Pressman, R. S., & Maxim, B. R. (2019). *Software engineering: A practitioner's approach* (9th ed.). McGraw-Hill Education.
- Project Management Institute. (2021). *A guide to the project management body of knowledge (PMBOK Guide)* (7th ed.).
- Puzo, M. (2022). *The godfather* (Deluxe ed.). G.P. Putnam's Sons. (Original work published 1969)
- Richards, M., & Ford, N. (2020). *Fundamentals of software architecture: An engineering approach.* O'Reilly Media.
- Robertson, S., & Robertson, J. (2012). *Mastering the requirements process: Getting requirements right* (3rd ed.). Addison-Wesley.
- Rozanski, N., & Woods, E. (2011). *Software systems architecture: Working with stakeholders using viewpoints and perspectives* (2nd ed.). Addison-Wesley.
- Ruggiero, P., & Heckathorn, M. A. (2012). *Data backup options.* US-CERT.
- Russell, B. (1998). The triumph of stupidity. In B. Russell, & H. Ruja (Ed.), *Morals and others: American essays 1931-1935* (Vol. II, pp. 27-28). Routledge. (Original work published 1933)
- Sadowski, C., & Zimmermann, T. (Eds.). (2019). *Rethinking productivity in software engineering.* Apress. <https://doi.org/10.1007/978-1-4842-4221-6>
- Satzinger, J. W., Jackson, R. B., & Burd, S. D. (2012). *Systems analysis and design in a changing world* (6th ed.). Course Technology.
- Schopenhauer, A. (2023). *Eristische Dialektik: Die Kunst, Recht zu behalten: Kunst des Streitens, Kunst des Disputierens.* DigiCat. (Original work published 1831)
- Selby, W. R. (Ed.). (2007). *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research.* Wiley-IEEE Computer Society.

- Simon, B. (2022). *Software architecture for developers*. Leanpub.
<https://leanpub.com/software-architecture-for-developers>
- Sokal, A., & Bricmont, J. (1998). *Fashionable nonsense: Postmodern intellectuals' abuse of science*. Picador.
- Sommerville, I. (2015). *Software engineering* (10th ed.). Pearson Education.
- Spolsky, J. (2004). *Joel on software: And on diverse and occasionally related matters that will prove of interest to software developers, designers, and managers, and to those who, whether by good fortune or ill luck, work with them in some capacity*. Apress. <https://doi.org/10.1007/978-1-4302-0753-5>
- Stanier, J. (2020). *Becoming an effective software engineering manager: How to be the leader your development team needs*. Pragmatic Bookshelf.
- Stellman, A., & Greene, J. (2005). *Applied software project management*. O'Reilly Media.
- Stephens, R. (2015). *Beginning software engineering*. John Wiley & Sons.
- Taleb, N. N. (2007). *The black swan: The impact of the highly improbable*. Random House.
- Thomas, D., & Hunt, A. (2019). *The pragmatic programmer: Your journey to mastery* (2nd ed.). Addison-Wesley.
- Tidwell, J., Brewer, C., & Valencia, A. (2020). *Designing interfaces: Patterns for effective interacting design* (3rd ed.). O'Reilly Media.
- Tockey, S. (2004). *Return on software: Maximizing the return on your software investment*. Addison-Wesley.
- Tornhill, A. (2018). *Software design X-rays: Fix technical debt with behavioral code analysis*. Pragmatic Bookshelf.
- Vance, S. (2014). *Quality code: Software testing principles, practices, and patterns*. Addison-Wesley.
- Wagner, S., & Murphy-Hill, E. (2019). Factors that influence productivity: A checklist. In C. Sadowski, & T. Zimmermann (Eds.), *Rethinking productivity in software engineering* (pp. 69-84). Apress. https://doi.org/10.1007/978-1-4842-4221-6_8
- Whittaker, J., Arbon, J., & Carollo, J. (2012). *How Google tests software*. Addison-Wesley.
- Wiegers, K., & Beatty, J. (2013). *Software requirements* (3rd ed.). Microsoft Press.

- Wiegers, K., & Hokanson, C. (2023). *Software requirements essentials: Core practice for successful business analysis*. Addison-Wesley.
- Wysocki, R. K. (2019). *Effective project management: Traditional, agile, extreme, hybrid* (8th ed.). John Wiley & Sons.
- Πλάτων. (ca. 399-387 B.C.E). *Απολογία Σωκράτους*.
- Толстой, Л. Н. (1877). *Анна Каренина*.
- 과학기술정보통신부 - 정보통신산업진흥원. (2021). *소프트웨어사업 요구사항 분석 적용 가이드*.
- 국가기술자격법. (2022). 제14조 국가기술자격 취득자에 대한 우대. 법률 제18925호. 2022년 6월 10일 일부개정.
- 국립국어원. (날짜 정보 없음). 성공. 표준국어대사전.
- <https://stdict.korean.go.kr/search/searchView.do?searchKeyword=%EC%84%B1%EA%B3%B5>에서 검색된 날짜: 2022년 7월 22일
- 김익환. (2010). 글로벌 소프트웨어를 꿈꾸다. 한빛미디어.
- 김익환. (2014). 글로벌 소프트웨어를 말하다, 지혜. 한빛미디어.
- 김익환, & 전규현. (2010). *소프트웨어 개발의 모든 것: 경영자에서 개발자까지 소프트웨어 회사에서 반드시 알아야 할 핵심 노하우* (개정판). 페가수스.
- 김익환, & 전규현. (2021). *소프트웨어 스펙의 모든 것: 프로젝트를 성공으로 이끄는 소프트웨어 스펙 작성법*. 한빛미디어.
- 김중철, & 김수지. (2021). 오늘도 개발자가 안 된다고 말했다. 디지털북스.
- 김창준. (2018). 함께 자라기: 애자일로 가는 길. 인사이트.
- 나피엠. (2010). *PM의 변: 능력 없는 프로젝트 관리자의 변명*. 비팬북스.
- 대한성서공회. (2017). 공동번역 성서(2017 개정판).
- 박창욱. (2024). *당신의 소프트웨어 프로젝트가 망하는 이유* (2판). 와이출판.
- <https://doi.org/10.23258/979-11-967623-5-3>
- 법인세법 시행규칙. (2018). [별표 6] 업종별자산의 기준내용연수 및 내용연수 범위표 (제15조 제3항 관련). 기획재정부령 제 671호. 2018년 3월 21일 개정.
- 신승환. (2009). 겹손한 개발자가 만든 거만한 소프트웨어. 인사이트.

- 신승환. (2012). *대한민국 소프트웨어, 리스트트: 위기를 넘어 도약으로*. 위키북스.
- 신진현. (2015). *프로젝트 관리자를 위한 프로젝트 관리 이야기*. 한빛미디어.
- 이세민, & 김남준. (2019). *더 팬찮은 개발자가 되기 위한 프로젝트 더보기: 내 기술에 깊이를 더하는 프로젝트의 생성과 소멸*. 프리렉.
- 이호종. (2011). *거꾸로 배우는 소프트웨어 개발*. 로드북.
- 임백준. (2016). *임백준의 대살개문: 대한민국을 살리는 개발자 문화*. 한빛미디어.
- 조달청. (2021). 내용연수. 조달청 고시 제 2021-41호.
- 조대협. (2015). *소프트웨어 개발과 테스트: 조대협의 서버 사이드*. 프리렉.
- 최범균. (2023). *육각형 개발자: 시니어 개발자로 성장하기 위한 10가지 핵심 역량*. 한빛미디어.
- 山本啓二. (2007). *아키텍트 이야기*. (이지연, 역자) 인사이트. (원본 출판 2004년)
- 清水亮. (2016). *화성에서 온 프로그래머, 금성에서 온 기획자*. (지정우, 역자) 한빛미디어. (원본 출판 2015년)
- 白川克, & 浜本佳史. (2023). *시스템을 잘 만들게 하는 기술*. (김모세, 역자) 위키북스. (원본 출판 2021년)
- 細川義洋. (2014). *왜 시스템 개발만 하면 싸워 달까?: 49가지 분쟁 사례로 배우는 프로젝트 관리 비법*. (최미정, 역자) 위키북스. (원본 출판 2013년)

내가 당신에게 할 말은 아직도 많다.

제목: 당신의 소프트웨어 프로젝트가 망하는 이유

판: 2판

지은이: 박창욱

발행인: 박창욱

발행일: 2024년 6월 1일

발행처: 와이출판

주소: (03643) 서울 서대문구 송죽길 19-3 와이원

메일: pub@ycomputing.co.kr

신고번호: 제2019-000067호

본문언어: 한국어

크기: 가로 182mm × 세로 257mm

페이지수: 374 페이지

정가: 500,000원 (종이책)

5,000원 (전자책)

ISBN: 979-11-967623-4-6 (13000) (종이책)

979-11-967623-5-3 (15000) (전자책)

DOI: <https://doi.org/10.23258/979-11-967623-5-3>