어떻게 공부할까? 프로그래머를 위한 공부론

김창준

이 글은 공부하는 방법과 과정에 관한 글입니다. 이 글은 제가 공부한 성공/실패 경험을 기본 토대로 했고, 지난 몇 년간 주변에서 저보다 먼저 공부한 사람들의 경험을 관찰, 분석한 것에 제가 다시 직접 실험한 것과 그밖에 오랫동안 꾸준히 모아온 자료들을 더했습니다. '만약 다시 공부한다면' 저는 이렇게 공부할 것입니다.
부디 독자 제현께서 이 글을 씨앗으로 삼아 자신만의 나무를 키우고 거기서 열매를 얻고, 또 그 열매의 씨앗이 다시 누군가에게 전해질 수 있다면 더 이상 바랄 것이 없겠습니다.
이 글은 특정 주제들의 학습/교수법에 대한 문제점과 제가 경험한 좋은 공부법을 소개하는 식으로 구성됐습니다. 여기에 선택된 과목은 리팩토링, 알고리즘 · 자료구조, 디자인패턴, 익스트림 프로그래밍(Extreme Programming 혹은 XP) 네 가지입니다.
이 네 가지가 선택된 이유는 필자가 관심있게 공부했던 것이기 때문만은 아니고, 모든 프로그래머에게 어떻게든 널리 도움이 될만한 교양과목이라 생각하여 선택한 것입니다. 그런데 이 네 가지의 순서가 겉보기와는 달리 어떤 단계적 발전을 함의하는 것은 아닙니다. 수신(修身)이 끝나면 더 이상 수신은 하지 않고 제가(齊家)를 한다는 것이 어불성설인 것과 같습니다.
원래는 글 후미에 일반론으로서의 공부 패턴들을 쓰려고 했습니다. 하지만 지면의 제약도 있고, 독자 스스로 이 글에서 그런 패턴을 추출하는 것도 의미가 있을 것이기에 생략했습니다. 그런 일반론이 여기 저기 숨어있기 때문에 알고리즘 공부에 나온 방법 대부분이 리팩토링 공부에도 적용할 수 있고, 적용되어야 한다는 점을 꼭 기억해 주셨으면 합니다. 다음에 기회가 닿는다면 제가 평소 사용하는 (컴퓨터) 공부패턴들을 소개하겠습니다.

1. 알고리즘 · 자료구조 학습에서의 문제

우리는 알고리즘 카탈로그를 배웁니다. 이미 그러한 해법이 존재하고, 그것이 최고이며, 따라서 그것을 달달 외우고 이해해야 합니다. 좀 똑똑한 친구들은 종종 "이야 이거 정말 기가 막힌 해법이군!"하고 감탄할지도 모릅니다. 대부분의 나머지 학생들은 그 해법을 이해하려고 머리를 쥐어짜고 한참을 씨름한 후에야 어렴풋이 왜 이 해법이 그 문제를 해결하는지 납득하게 됩니다.

그리고는 그 '증명'은 책 속에 덮어두고 까맣게 사라져버립니다. 앞으로는 그냥 '사용'하면 되는 것입니다. 더 많은 대다수의 학생은 이 과정이 무의미하다는 것을 알기 때문에 왜 이 해법이 이 문제를 문제없이 해결하는지의 증명은 간단히 건너뜁니다.

이런 학생들은 이미 주어진 알고리즘을 사용하는 일종의 객관식 혹은 문제 출제자가 존재하는 시험장 상황에서는 뛰어난 성적을 보일 것임은 자명합니다. 하지만 스스로가 문제와 해답을 모두 만들어내야 하는 상황이라면, 또는 해답이 존재하지 않을 가능성이 있는 상황이라면, 혹은 최적해를 구하는 것이 불가능에 가깝다면, 혹은 알고리즘을 완전히 새로 고안해내야 하거나 기존 알고리즘을 변형해야 하는 상황이라면 어떨까요?

교육은 물고기를 잡는 방법을 가르쳐야 합니다. 어떤 알고리즘을 배운다면 그 알고리즘을 고안해낸 사람이 어떤 사고 과정을 거쳐 그 해법에 도달했는지를 구경할 수 있어야 하고, 학생은 각자 스스로만의 해법을 차근차근 '구성'(construct)할 수 있어야 합니다(이를 교육철학에서 구성주의라고 합니다. 교육철학자 삐아제(Jean Piaget)의 제자이자, 마빈 민스키와 함께 MIT 미디어랩의 선구자인 세이머 페퍼트 박사가 주창했습니다). 전문가가 하는 것을 배우지 말고, 그들이 어떻게 전문가가 되었는지를 배우고 흉내 내야 합니다.

결국은 소크라테스적인 대화법입니다. 해답을 가르쳐 주지 않으면서도 초등학교 학생이 자신이 가진 지식만으로 스스로 퀵소트를 유도할 수 있도록 옆에서 도와줄 수 있습니까? 이것이 우리 스스로와 교사들에게 물어야 할 질문입니다.

왜 우리는 학교에서 '프로그래밍을 하는 과정'이나 '디자인 과정'(소프트웨어 공학에서 말하는 개발 프로세스가 아니라 몇 시간이나 몇 십 분 단위의, 개인적인 차원의 사고 과정 등을 일컫습니다)을 명시적으로 배운 적이 없을까요? 왜 해답에 이르는 과정을 가르쳐주는 사람이 없나요? 우리가 보는 것은 모조리 이미 훌륭히 완성된, 종적 상태의 결과물로서의 프로그램뿐입니다. 어느 날 문득 하늘에서 완성된 프로그램이 뚝 떨어지는 경우는 없는데 말입니다.

교수가 어떤 알고리즘 문제의 해답을 가르칠 때, "교수님, 교수님께서는 어떤 사고 과정을 거쳐, 그리고 어떤 디자인 과정과 프로그래밍 과정을 거쳐서 그 프로그램을 만드셨습니까?"하고 물어봅시다. 만약 여기에 어떤 체계적인 답변도 할 수 없는 분이라면 그 분은 자신의 사고에 대해 '사고'해 본 적이 없거나 문제 해결에 어떤 효율적 체계를 갖추지 못한 분이며, 따라서 아직 남을 가르칠 준비가 되어있지 않은 분일 것입니다. 만약 정말 그렇다면 우리는 어떻게 해야 할까요?

2. 자료구조와 알고리즘 공부

제가 생각건대, 교육적인 목적에서는 자료구조나 알고리즘을 처음 공부할 때 우선은 특정 언어로 구현된 것을 보지 않는 것이 좋을 때가 많습니다. 대신 말로 된 설명이나 의사코드(pseudo-code) 등으로 그 개념까지만 이해하는 것이죠. 그 아이디어를 절차형(C, 어셈블리어)이나 함수형(LISP, Scheme, Haskell), 객체지향(자바, 스몰토크) 언어 등으로 직접 구현해 보는 겁니다. 그 다음에는 다른 사람이나 다른 책의 코드와 비교합니다. 이 경험을 애초에 박탈당한 사람은 귀중한 배움과 깨달음의 기회를 잃은 셈입니다.

만약 여러 사람이 함께 공부한다면 각자 동일한 아이디어를 같은 언어로 혹은 다른 언어로 어떻게 다르게 표현했는지를 서로 비교해 보면 배우는 것이 무척 많습니다.

우리가 자료구조나 알고리즘을 공부하는 이유는, 특정 '실세계의 문제'를 어떠한 '수학적 아이디어'로 매핑시켜 해결할 수 있는지, 그것이 효율적인지, 또 이를 컴퓨터에 어떻게 효율적으로 구현할 수 있는지 따지고, 그것을 실제로 구현하기 위해서입니다. 따라서 이 과정에 있어 실세계의 문제를 수학 문제로, 그리고 수학적 개념을 프로그래밍 언어로 효율적으로 표현해내는 것은 아주 중요한 능력이 됩니다.

알고리즘 공부에서 중요한 것 개별 알고리즘의 목록을 이해, 암기하며 익히는 것도 중요하지만 더 중요한 것은 다음 네 가지입니다.

1. 알고리즘을 스스로 생각해낼 수 있는 능력
2. 다른 알고리즘과 효율을 비교할 수 있는 능력
3. 알고리즘을 컴퓨터와 다른 사람이 이해할 수 있는 언어로 표현해낼 수 있는 능력
4. 이것의 정상작동(correctness) 여부를 검증해 내는 능력

첫 번째가 제대로 훈련되지 못한 사람은 알고리즘 목록의 스테레오 타입에만 길들여져 있어서 모든 문제를 자신이 아는 알고리즘 목록에 끼워 맞추려고 합니다. 디자인패턴을 잘못 공부한 사람과 비슷합니다. 이런 사람들은 마치 과거에 수학 정석만 수십 번 공부해 문제를 하나 던져주기만 하면, 생각해보지도 않고 자신이 풀었던 문제들의 패턴 중 가장 비슷한 것 하나를 기계적 · 무의식적으로 풀어제끼는 문제풀이기계와 비슷합니다. 그들에게 도중에 물어보십시오. "너 지금 무슨 문제 풀고 있는 거니?" 열심히 연습장에 뭔가 풀어나가고는 있지만 그들은 자신이 뭘 풀고 있는지도 제대로 인식하지 못 하는 경우가 많습니다.

머리가 푸는 게 아니고 손이 푸는 것이죠. 이렇게 되면 도구에 종속되는 '망치의 오류'에 빠지기 쉽습니다. 새로운 알고리즘을 고안해야 하는 상황에서도 기존 알고리즘에 계속 매달릴 뿐입니다. 알고리즘을 새로 고안해 내건 혹은 기존의 것을 조합하건 스스로 생각해 내는 훈련이 필요합니다.

두 번째가 제대로 훈련되지 못한 사람은 일일이 구현해 보고 실험해 봐야만 알고리즘 간의 효율을 비교할 수 있습니다. 특히 자신이 가진 카탈로그를 벗어난 알고리즘을 만나면 이 문제가 생깁니다. 이건 상당한 대가를 치르게 합니다.

세 번째가 제대로 훈련되지 못한 사람은, 문제를 보면 "아, 이건 이렇게 저렇게 해결하면 됩니다"하는 말은 곧잘 할 수 있지만 막상 컴퓨터 앞에 앉혀 놓으면 아무 것도 하지 못합니다. 심지어 자신이 생각해낸 그 구체적 알고리즘을 남에게 설명해 줄 수는 있지만, 그걸 '컴퓨터에게' 설명하는 데는 실패합니다. 뭔가 생각해낼 수 있다는 것과 그걸 컴퓨터가 이해할 수 있게 설명할 수 있다는 것은 다른 차원의 능력을 필요로 합니다.

네 번째가 제대로 훈련되지 못한 사람은, 알고리즘을 특정 언어로 구현해도, 그것이 옳다는 확신을 할 수 없습니다. 임시변통(ad hoc)의 아슬아슬한 코드가 되거나 이것저것 덧붙인 누더기 코드가 되기 쉽습니다. 이걸 피하려면 두 가지 훈련이 필요합니다.

하나는 수학적 · 논리학적 증명의 훈련이고, 다른 하나는 테스트 훈련입니다. 전자가 이론적이라면 후자는 실용적인 면이 강합니다. 양자는 상보적인 관계입니다. 특수한 경우들을 개별적으로 테스트해서는 검증해야 할 것이 너무 많고, 또 모든 경우에 대해 확신할 수 없습니다. 테스트가 버그의 부재를 보장할 수는 없습니다. 하지만 수학적 증명을 통하면 그것이 가능합니다. 또, 어떤 경우에는 수학적 증명을 굳이 할 필요 없이 단순히 테스트 케이스 몇 개만으로도 충분히 안정성이 보장되는 경우가 있습니다. 이럴 때는 그냥 테스트만으로 만족할 수 있습니다.

3. 실질적이고 구체적인 문제를 함께 다루라

자료구조와 알고리즘 공부를 할 때에는 가능하면 실질적이고 구체적인 실세계의 문제를 함께 다루는 것이 큰 도움이 됩니다. 모든 학습에 있어 이는 똑같이 적용됩니다. 인류의 지성사를 봐도, 구상(concrete) 다음에 추상(abstract)이 옵니다. 인간 개체 하나의 성장을 봐도 그러합니다. 'be-동사 더하기 to-부정사'가 예정으로 해석될 수 있다는 룰만 외우는 것보다 다양한 예문을 실제 문맥 속에서 여러 번 보는 것

이 훨씬 나을 것은 자명합니다. 알고리즘과 자료구조를 공부할 때 여러 친구들과 함께 연습문제(특히 우리가 경험하는 실세계의 대상들과 관련이 있는 것)를 풀어보기도 하고, ACM의 ICPC(International Collegiate Programming Contest: 세계 대학생 프로그래밍 경진 대회) 등의 프로그래밍 경진 대회 문제 중 해당 알고리즘 · 자료구조가 사용될 수 있는 문제를 같이 풀어보는 것도 아주 좋습니다. 이게 가능하려면 "이 알고리즘이 쓰이는 문제는 이거다"하고 가이드를 해줄 사람이 있으면 좋겠죠. 이것은 그 구체적 알고리즘 · 자료구조를 훈련하는 것이고, 이와 동시에 어떤 알고리즘을 써야할지 선택, 조합하는 것과 새로운 알고리즘을 만들어내는 훈련도 무척 중요합니다.

4. 알고리즘 디자인 과정의 중요성

알고리즘을 좀더 수월하게, 또 잘 만들려면 알고리즘 디자인 과정에 대해 생각해 봐야 합니다. 그냥 밑도 끝도 없이 문제를 쳐다본다고 해서 알고리즘이 튀어나오진 않습니다. 체계적이고 효율적인 접근법을 사용해야 합니다. 대표적인 것으로 다익스트라(E. W. Dijkstra)와 워스(N. Wirth)의 '조금씩 개선하기'(Stepwise Refinement)가 있습니다. 워스의 「Program Development by Stepwise Refinement」(1971, CACM 14.4, http://www.acm.org/classics/dec95)를 꼭 읽어보길 바랍니다. 여기 소개된 조금씩 개선하기는 구조적 프로그래밍에서 핵심적 역할을 했습니다(구조적 프로그래밍을 'goto 문 제거' 정도로 생각하면 안 됩니다). 다익스트라의 「Stepwise Program Construction」(Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982,http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD227.PDF도 추천합니다.

알고리즘 검증은 알고리즘 디자인과 함께 갑니다. 새로운 알고리즘을 고안할 때 검증해 가면서 디자인하기 때문입니다. 물론 가장 큰 역할은 고안이 끝났을 때의 검증입니다. 알고리즘 검증에는 루프 불변식(loop invariant) 같은 것이 아주 유용합니다. 아주 막강한 무기입니다. 익혀 두면 두고두고 가치를 발휘할 것입니다. 맨버(Udi Manber)의 알고리즘 서적(『Introduction to Algorithms: A Creative Approach』)이 알고리즘 검증과 디자인이 함께 진행해 가는 예로 자주 추천됩니다. 많은 계발을 얻을 것입니다. 고전으로는 다익스트라의 『A Discipline of Programming』과 그라이스(Gries)의 『The Science of Programming』이 있습니다. 특히 전자를 추천합니다. 프로그래밍에 대한 관을 뒤흔들어 놓을 것입니다.

5. 알고리즘과 패러다임

알고리즘을 공부하면 큰 줄기들을 알아야 합니다. 개별 테크닉도 중요하지만 '패러다임'이라고 할만한 것들을 알아야 합니다. 이것에 대해서는 튜링상을 수상한 로버트 플로이드(Robert Floyd)의 튜링상 수상 강연(The Paradigms of Programming, 1978)을 추천합니다. 패러다임을 알아야 알고리즘을 상황에 맞게 마음대로 변통할 수 있습니다. 그리고 자신만의 분류법을 만들어야 합니다. 구체적인 문제들을 케이스 바이 케이스로 여럿 접하는 동안 그냥 지나쳐 버리면 개별자는 영원히 개별자로 남을 뿐입니다. 비슷한 문제들을 서로 묶어서 일반화해야 합니다.

이런 패러다임을 발견하려면 '다시 하기'가 아주 좋습니다. 다른 것들과 마찬가지로, 이 다시 하기는 알고리즘에서만이 아니고 모든 공부에 적용할 수 있습니다. 같은 것을 다시 해보는 것에서 우리는 얼마나 많은 것을 배울 수 있을까요. 대단히 많습니다. 왜 동일한 문제를 여러 번 풀고, 왜 같은 내용의 세미나에 또 다시 참석하고, 같은 프로그램을 거듭 작성할까요? 훨씬 더 많이 배울 수 있기 때문입니다. 화술 교육에서는 같은 주제에 대해 한 번 말해본 연사와 두 번 말해본 연사는 천지 차이가 있다고 말합니다. 같은 일에 대해 두 번의 기회가 주어지면 두 번째에는 첫 번째보다 잘 할 기회가 있습니다. 게다가 첫 번째 경험했던 것을 '터널을 벗어나서' 다소 객관적으로 볼 수 있게 됩니다. 왜 자신이 저번에 이걸 잘 못 했고, 저걸 잘 했는지 알게 되고, 어떻게 하면 그걸 더 잘할 수 있을는지 깨닫게 됩니다. 저는 똑같은 문제를 여러 번 풀더라도 매번 조금씩 다른 해답을 얻습니다. 그러면서 정말 엄청나게 많은 것을 배웁니다. '비슷한 문제'를 모두 풀 능력이 생깁니다.

제가 개인적으로 존경하는 전산학자 로버트 플로이드(Robert W. Floyd)는 1978년도 튜링상 강연에서 다음과 같은 말을 합니다.

> 제가 어려운 알고리즘을 디자인하는 경험을 생각해 볼 때, 제 능력을 넓히는 데 가장 도움이 되는 특정한 테크닉이 있습니다. 어려운 문제를 푼 후에, 저는 그것을 처음부터 완전히 새로 풉니다. 좀 전에 얻은 해법의 통찰(insight)만을 유지하면서 말이죠. 해법이 제가 희망하는 만큼 명료하고 직접적인 것이 될 때까지 반복합니다. 그런 다음, 비슷한 문제들을 공략할 어떤 일반적인 룰을 찾습니다. 아까 주어진 문제를 아예 처음부터 최고로 효율적인 방향에서 접근하도록 이끌어줬을 그런 룰을 찾는 거죠. 많은 경우에 그런 룰은 불변의 가치가 있습니다. … 포트란의 룰은 몇 시간 내에 배울 수 있습니다. 하지만 관련 패러다임은 훨씬 더 많은 시간이 걸립니다. 배우거나(learn) 배운 것을 잊거나(unlearn) 하는 데 모두.

수학자와 프로그래머를 포함한 모든 문제 해결자들의 고전이 된 죠지 폴리야(George Polya)의 『How to Solve it』에는 이런 말이 있습니다:

> 심지어는 꽤나 훌륭한 학생들도, 문제의 해법을 얻고 논증을 깨끗하게 적은 다음에는 책을 덮어버리고 뭔가 다른 것을 찾는다. 그렇게 하는 동안 그들은 그 노력의 중요하고도 교육적인 측면을 잃어버리게 된다. … 훌륭한 선생은 어떠한 문제이건 간에 완전히 바닥이 드러나는 경우는 없다는 관점을 스스로 이해하고 또 학생들에게 명심시켜야 한다.

저는 ACM의 ICPC 문제 중에 어떤 문제를 이제까지 열 번도 넘게 풀었습니다. 대부분 짝 프로그래밍이나 세미나를 통해 프로그래밍 시연을 했던 것인데, 제 세미나에 여러 번 참석한 분이 농담조로 웃으며 물었습니다. "신기해요. 창준 씨는 그 문제를 풀 때마다 다른 프로그램을 짜는 것 같아요. 설마 준비를 안 해 와서 그냥 내키는 대로 하는 건 아니죠?" 저는 카오스 시스템과 비슷하게 초기치 민감도가 프로그래밍에도 작용하는 것 같다고 대답했습니다. 저 스스로 다른 해법을 시도하고 싶은 마음이 있으면 출발이 조금 다르고, 또 거기서 나오는 진행 방향도 다르게 됩니다. 그런데 중요한 것은 이렇게 같은 문제를 매번 다르게 풀면서 배우는 것이 엄청나게 많다는 점입니다. 저는 매번, 전보다 개선할 것을 찾아내게 되고, 또 새로운 것을 배웁니다. 마치 마르지 않는 샘물처럼 계속 생각할 거리를 준다는 점이 참 놀랍습니다.

알고리즘 개론 교재로는 CLR(Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest)을 추천합니다. 이와 함께 혹은 이 책을 읽기 전에 존 벤틀리(Jon Bentley)의 『Programming Pearls』도 강력 추천합니다. 세계적인 짱짱한 프로그래머와 전산학자들이 함께 꼽은 위대한 책 목록에서 몇 손가락 안에 드는 책입니다. 아직 이 책을 본 적이 없는 사람은 축하합니다. 아마 몇 주간은 감동 속에 하루하루를 보내게 될 겁니다.

6. 리팩토링 학습에서의 문제

먼저, 본지 2001년 11월호에 제가 썼던 마틴 파울러의 책을 추천하는 글을 인용하겠습니다.

> OOP를 하건 안 하건 프로그래밍이란 업을 하는 사람이라면 이 책은 자신의 공력을 서너 단계 레벨업시켜줄 수 있다. 자질구레한 술기를 익히는 것이 아니고 기감과 내공을 증강하는 것이다.
>
> 혹자는 DP 이전에 RF를 봐야 한다고도 한다. 이 말이 어느 정도 일리가 있는 것이, 효과적인 학습은 문제의식이 선행해야 하기 때문이다. DP는 거시적 차원에서 해결안을 모아놓은 것이다. RF를 보고 나쁜 냄새(Bad Smell)를 맡을 수 있는 후각을 발달시켜야 한다. RF의 목록을 모두 외우는 것은 큰 의미가 없다. 그것보다 냄새나는 코드를 느낄 수 있는 감수성을 키우는 것이 더 중요하다. 필자는 일주일에 한 가지씩 나쁜 냄새를 정해놓고 그 기간 동안에는 자신이 접하는 모든 코드에서 그 냄새만이라도 확실히 맡도록 집중하는 방법을 권한다. 일명 일취집중후각법. 패턴 개념을 만든 건축가 크리스토퍼 알렉산더나 GoF의 랄프 존슨은 좋은 디자인이란 나쁜 것이 없는 상태라고 한다. 무색 무미 무취의 무위(無爲)적 자연(自然) 코드가 되는 그 날을 위해 오늘도 우리는 리팩토링이라는 유위(有爲)를 익힌다.

주변에서 리팩토링을 잘못 공부하는 경우를 종종 접합니다. 어떤 의미에서 잘못 공부한다고 할까요? '실체화'가 문제입니다. 리팩토링은 도구이고 방편일 뿐인데, 그것에 매달리는 것은 달은 보지 않고 손을 보는 것과 같습니다. 저는 리팩토링 책이 또 하나의 (이미 그 병폐가 많이 드러난) GoF 책이 되는 현상이 매우 걱정됩니다.

7. 리팩토링 공부

사람들이 일반적으로 생각하는 바와는 달리 리팩토링 학습에 있어 어떤 리팩토링이 있는지, 구체적으로 무엇인지 등의 리팩토링 목록에 대한 지식과 각각에 대한 메카닉스(Mechanics: 해당 리팩토링을 해나가는 구체적 단계)는 오히려 덜 중요할 수 있습니다. 더 기본적이고 유용한 것은 코드 냄새(Code Smell)와 짧은 테스트-코드 싸이클입니다. 이것만 제대로 되면 리팩토링 책의 모든 리팩토링을 스스로 구성해낼 수 있으며, 다른 종류의 리팩토링까지 직접 발견해낼 수 있습니다.

그 책에서는 테스트의 중요성이 충분히 강조되지 않았습니다. 하지만 테스트 코드 없는 리팩토링은 안전벨트 없는 자동차 경주와 같습니다. 그리고 테스트 코드가 리팩토링의 방향을 결정하기도 합니다. 양자는 음과 양처럼 서로 엮여 있습니다. 이런 의미에서 리팩토링은 TDD(Test Driven Development)와 함께 수련하는 것이 좋습니다. 훨씬 더 빨리, 훨씬 더 많은 것을 배울 수 있을 겁니다.

리팩토링을 공부할 때는 우선 코드 냄새의 종류를 알고, 왜 그것이 나쁜 냄새가 될 수 있는지 이해하고(그게 불가하다면 리팩토링 공부를 미뤄야 합니다) 거기에 동의할 수 있어야 합니다. 그런 다음, 대

충 어떤 종류의 리팩토링이 가능한지 죽 훑어봅니다. 그 중 몇 개는 메카닉스를 따라가면서 실험해 봅니다. 이제는 책을 덮습니다. 그리고 실제 코드를 접하고, 만약 거기에서 냄새를 맡는다면 이 냄새를 없애기 위해 어떻게 해야 할지 스스로 고민합니다. 리팩토링 책의 목록은 일단 무시하십시오. 그 냄새를 없애는 것이 목표이지, 어떤 리팩토링을 여기에 '써먹는 것'이 목표가 되어선 안 됩니다. 이 때, 반드시 테스트 코드가 있어야 합니다. 그래야 '좋은' 리팩토링을 할 수 있습니다. 책을 떠나 있으면서도 책에서 떠나지 않는 방법입니다.

리팩토링을 하기 전에 초록색 불(테스트가 모두 통과)이 들어온 시점에서 코드를 일부 고치면 빨간 불(테스트가 하나라도 실패)로 바뀔 겁니다. 이게 다시 초록색 불이 될 때까지 최소한도의 시간이 걸리도록 하십시오. 현 초록색에서 다음 초록색까지 최소한의 시간을 소비하도록 하면서 코드와 테스팅을 오가게 되면 자기도 모르는 사이에 훌륭한 리팩토링이 자발공으로 터져 나옵니다. 여기서 목적지는 우선은 OAOO(Once And Only Once: 모든 것은 오로지 한 번만 말해야 한다)를 지키는 쪽으로 합니다. 그러면 OAOO와 짧은 테스트-코드 싸이클을 지키는 사이 어느새 탁월한 디자인이 튀어나옵니다. 참고로 저는 '모래시계 프로그래밍'이란 걸 가끔 합니다. 모래시계나 알람 프로그램으로 테스트-코드 사이클의 시간을 재는 것입니다. 그래서 가급적이면 한 사이클이 3분 이내(대부분의 모래시계는 단위가 3분입니다)에 끝나도록 노력합니다. 여기서 성공을 하건 실패를 하건 많은 걸 얻습니다.

리팩토링 수련법 제가 고안, 사용한 몇 가지 리팩토링 수련법을 알려드립니다.

1. 일취집중후각법: 앞에 소개한 본지 2001년 11월호에서 인용된 글 참조
2. 주석 최소화: 주석을 최소화하되 코드의 가독성이 떨어지지 않도록(혹은 오히려 올라가도록) 노력합니다. 이렇게 하면 자동으로 리팩토링이 이뤄지는 경우가 많습니다.
3. OAOO 따르기: OAOO 법칙을 가능하면 최대한, 엄격하게 따르려고 합니다. 역시 자동으로 좋은 리팩토링이 이뤄집니다. 여기서 디자인패턴이 창발하기도 합니다. GoF 책을 한번도 보지 못한 사람이 디자인패턴을 자유자재로 부리는 경우를 보게 됩니다.
4. 디미터 법칙(Law of Demeter) 따르기: 디미터 법칙을 가능하면 지키려고 합니다. 어떤 리팩토링이 저절로 이뤄지거나 혹은 필요 없어지는가요?
5. 짝(Pair) 리팩토링: 함께 리팩토링합니다. 혼자 하는 것보다 더 빨리, 더 많은 걸 배우게 됩니다. 특히, 각자 작성했던 코드를 함께 리팩토링하고, 제3자의 코드를 함께 리팩토링해 봅니다. 사람이 많다면 다른 짝이 리팩토링한 것과 서로 비교하고 토론합니다.
6. '무엇'과 '어떻게'를 분리: 어떻게에서 무엇을 분리해 내도록 합니다. 어떤 리팩토링이 창발합니까?

여기서 번, 짝 리팩토링은 아주 효과적인 방법입니다. 저는 이것을 협동적 학습(Collaborative Learning)이라고 부릅니다. 상대가 나보다 더 많이 아는 경우만이 아니고, 서로 아는 것이 비슷해도 많은 양의 학습이 발생합니다. 특히, 전문가와 함께 짝 프로그래밍을 하면 무서울 만큼 빠른 학습을 경험할 수 있습니다. 저와 짝 프로그래밍을 한 사람이 학습하는 속도에서 경이감을 느낀 적이 한두 번이 아닙니다. 문화는 사회적으로 학습되는 것입니다. 우리가 지식이라고 하는 것의 상당 부분은 문화처럼 암묵적인 지식(Tacit Knowledge)입니다. 전문가가 문제가 생겼을 때 어떻게 사고하고, 어떤 과정을 거쳐 접근하며, 어떻게 디버깅하고, 키보드를 어떤 식으로 누르는지, 사고 도구로 무엇을 사용하는지, 일 계획과 관리를 어떻게 하는지, 동료와 어떻게 대화하는지 등은 성문화되어 있지 않습니다. 그러나 이런 것들은 아주 중요합니다. 프로페셔널의 하루 일과의 대부분을 이루고 있기 때문입니다. 이런 것들은 전문가 바로 옆에서 조금씩 일을 도와주면서 배워야 합니다. 도제 살이(Apprenticeship)입니다. 진정한 전문가는 모든 동작이 우아합니다. 마치 춤을 추는 것 같습니다. 이 모습을 바로 곁에서 지켜보게 되면, 주니어는 한마디로 엄청난 충격을 받습니다. 그리고 스펀지처럼 빨아들이게 됩니다. 도대체 이 경험을 책이나 공장화한 학교가 어떻게 대신하겠습니까. 이와 관련해서는 레이브와 웽거(Jean Lave, Etienne Wenger)의 『Situated Learning : Legitimate Peripheral Participation』을 강력 추천합니다. 이 글을 보는 모든 교육 종사자들이 꼭 읽어봤으면 합니다. 이 협동적 학습은 두 사람만이 아니고 그룹 단위로도 가능합니다. 스터디에 이용하면 재미와 유익함 일석이조를 얻습니다.

이 외에, 특히(어쩌면 가장) 중요한 것은 일취집중후각법 등을 이용해 자신의 코드 후각의 민감도를 높이는 것입니다. 코드 후각의 메타포 말고도 유사한 메타포가 몇 가지 더 있습니다. 켄트 벡은 코드의 소리를 들으라고 하고, 저는 코드를 향해 대화하라고 합니다. 코드의 소리를 듣는 것은 코드가 원하는 것에 귀를 기울이는 것을 말합니다. 코드는 단순해지려는 욕망이 있습니다. 그걸 이뤄주는 것이 프로그래머입니다. 그리고 짝 프로그래밍을 할 때 두 사람의 대화를 코드에 남기도록 합니다. 주석이 아닙니다. 왜 이것이 중요한가는 본지 2001년 12월호 「허실문답 XP 강화」를 참고하기 바랍니다.

기학으로 우리 사상사에 큰 획을 그은 철학자요, '서울서 책만 사다 망한 사람'으로 이름을 날릴 정도로 엄청난 지식욕을 과시하던 조선시대 사상가 혜강 최한기는 그의 저술 『신기통』(神氣通)에서 '눈에 통하는 법(目通), 귀에 통하는 법(耳通), 코에 통하는 법(鼻通)' 등을 이야기하고 있습니다. 어떻게 하면 우리는 코드에 도통할 수 있을까요? 리팩토링을 공부하거나 혹은 했던 사람들에게 많은 영감과 메타

포를 주는 책으로 일독을 권합니다. 필자가 기회가 닿는다면 프로그래밍을 혜강의 사상적 측면에서 조망한 글을 써보고 싶습니다.

앞서의 것들이 어느 정도 이뤄지고, 리팩토링에 대한 감이 오게 되면 그 때 비로소 리팩토링 책을 하나 하나 파헤치고 또 거기서 제대로 된 비판을 할 수 있게 됩니다.

8. 디자인패턴 학습에서의 문제

잡지에 연재되거나 서적으로 출간된 혹은 세미나에서 진행되었던 디자인패턴 '강의'를 몇 가지 접했습니다. 훌륭한 강의도 많았지만 그렇지 못한 것도 있었습니다. 몇 가지 문제점을 지적하겠습니다.

- 패턴을 지나치게 실체화, 정형화해 설명한다.
- 컨텍스트와 문제 상황에 대한 설명이 없거나 부족하다. 결과적으로 문제를 해결하기 위해 패턴이 도입된 것이 아니라 패턴을 써먹기 위해 패턴이 도입된 느낌을 준다.
- 문제의식을 먼저 형성하게 하지 않고 답을 먼저 보여준 뒤 그걸 어디에 써먹을지 가르친다. 왜 이걸 쓰는 게 좋은지는 일언반구 언급이 없다. 독자는 '어린아이가 망치를 들고 있는 오류'에 빠질 것이다.
- 패턴이 어떻게 생성되었는지 그 과정을 보여주지 못한다. 즉, 스스로 패턴을 만들어내는 데 도움이 전혀 되지 않는다.
- 해당 패턴이 현실적으로 가장 자주 쓰이는 맥락을 보여주지 못한다. 대부분 장난감 문제(Toy Problem)에서 끝난다.

그런 패턴 강의를 하는 분들이 알렉산더(Christopher Alexander, 패턴언어 창시자)의 저작을 충실히 읽어봤다면 이런 병폐는 없을 것이라 생각합니다. 알렉산더의 저작을 접해보지 못 하고서 패턴을 가르치는 사람은 성경을 읽어보지 않은 전도사와 같을 것입니다. 알렉산더가 『The Timeless Way of Building』의 마지막에서 무슨 말을 하는가요?

> 이 마지막 단계에는 더 이상 패턴은 중요하지 않다. … 패턴은 당신이 현실적인 것에 대해 수용적이 되는 것을 가르쳐줬다.

패턴 역시 도구요, 방편일 뿐입니다. 패턴은 현실적인 것에 대해 수용적이 되도록 가르친다는 말은 결국 우리가 궁극적으로 추구하는 것은 패턴이 아니라 현실이어야 한다는 이야기입니다. 물론 처음 단계에는 교육적인 목적에서, 어느 정도 패턴에 얽매여도 괜찮다고는 해도, 나중에 패턴을 잊고 패턴에서 자유로워지려면 처음부터 패턴에 대해 도구적 · 방편적인 인식을 가져야 합니다.

미국 캘리포니아 주립대학의 교수 베티 에드워즈(Betty Edwards)가 쓴 책 중에 『Drawing on the Right Side of the Brain』이라는 유명한 베스트셀러가 있습니다. 사람의 뇌와 그림 그리기의 관계에 대한 탁월한 책입니다. 에드워즈는 자신의 그리기 실력을 향상시키기 위해 우뇌를 적극적으로 사용하는 구체적인 방법들을 가르쳐줍니다. 그 중 대표적인 것 하나가 대상을 뒤집어 놓고 그리는 것입니다. 지금 실험해 보길 바랍니다. 1000원권 지폐를 바로 놓고 그걸 비슷하게 그려보고, 이번에는 그걸 위아래가 거꾸로 되게 놓고 따라 그려보십시오. 아마 무척 놀랐을 겁니다. "아니 내가 이렇게 그림을 잘 그리다니! 그것도 거꾸로!" 그렇습니다. 우리는 자신의 머리 속 패턴에 얽매여 세상을 제대로 보지 못 할 때가 많습니다. 실체가 코에 약간이라도 비슷하게 보이면 우리는 그것을 이미 우리 머리 속에 추상적으로 갖고 있던 기하학적 '코'의 패턴으로 대체해버리는 것입니다.

9. 디자인 패턴 공부

우선은 제 교육철학과 언어교습론, 그리고 공부론 이야기를 잠깐 하겠습니다.

기본적으로 교육은 교육자가 피교육자가에게 지식을 그대로 전달하는 행위가 아닙니다. 진정한 교육은 피교육자의 개인적 체험에 기반한 전폭적 동의에서 출발합니다. 저는 이를 동의에 의한 교육이라고 합니다.

제가 "주석문을 가능하면 쓰지 않는 것이 더 좋다"는 이야기를 했을 때 이 문장을 하나의 사실로 받아들이고 기억하면 당장 그 시점에는 학습이 일어나지 않는다고 봅니다. 대신 여러분이 차후에 여러 가지 경험을 하면서도 이 화두를 놓치지 않고 있다가 어느 순간엔가, "아! 그래, 주석문을 쓰지 않는 게 좋겠구나!"하고 자각하는 순간, 바로 그 시점에 학습과 교육이 이뤄지는 것입니다. 이는 기본적으로 삐아제와 비갓스키(Lev Vygotsky)의 구성주의를 따르는 것이죠. 지식이란 외부에서 입력받는 것이 아니고, 그것에 대한 모델을 학습자 스스로가 내부에서 구축할 때 획득할 수 있는 것이라는 사상이죠.

권법에서 주먹에 대해 달통한 도사가 '권을 내지르는 법'에 대한 규칙들을 정리해서 애제자의 머리 속에 아무리 쑤셔 넣는 데 성공한들 그 제자가 도사만큼 주먹이 나갈리는 만무합니다. 권을 내지르는 법을 유추해 내기까지 그 스승이 겪은 과정을 제자는 완전히 쏙 빼먹고 있기 때문입니다. 소위 '몸'이 만들어지지 않은 것이지요. 제자는 마당 쓸기부터, 물 긷기 등의 수련 과정을 겪어야만 하고 스승이 정리한 그 일련의 규칙에 손뼉을 치고 춤을 추며 기쁨의 동의를 할 수 있을 정도로 수련 과정이 축적된 이후에야 비로소 진정한 '가르침'이 이뤄지는 것이며, 청출어람의 가능성도 생각해 볼 수 있는 것입니다.

이런 동의라는 것은 학습자 자신만의 컨텍스트와 문제의식을 바탕으로 한 것입니다. 우리는 많은 경우, 어떤 지식과 동시에 그 지식의 필요성까지도 지식화해서 외부에서 주입을 받습니다. 하지만 진정 체화된 지식을 위해서는 스스로가 이미 문제의식을 갖고 있어야 합니다.

패턴도 마찬가지인데, 대부분 그 패턴의 필요성을 체감하지 못한 채 그냥 도식적 구조를 외우기에만 주력하는 사람이 많습니다만, 사실 그렇게 되면 어떤 경우에 이 패턴이 필요하고 어떤 경우에는 사용하면 안 되는지(GoF는 패턴을 정말 안다는 것은 그 패턴을 쓰면 안 될 때를 아는 것이라 했습니다) 등을 알기 힘듭니다. 설령 책에 나온 가이드를 암기했더라도요. 자신의 삶 속에서 문제의식이 구체적으로 실제 경험으로 형성되지 않았기 때문입니다.

GoF 중 한 명인 랄프 존슨(Ralph Johnson)은 다음과 같이 말합니다.

> 우리[GoF]는 책에서, 정말 그 패턴들이 필요하다는 것을 알만큼 충분한 경험을 갖기 전에는 그것을 [시스템 속에] 집어넣는 것을 피해야 한다고 말할 만큼 대담하진 못했다. 하지만 우리 모두는 그 사실을 믿었다. 패턴은 프로그램의 초기 버전이 아니고 프로그램 생애의 훨씬 나중에 가서야 비로소 등장해야 한다고 나는 늘 생각해 왔다.

결국은 어떤 패턴의 필요성을 자신의 경험 속에서 절감하지 못한다면 그 패턴을 제대로 아는 것이 아니라고 말할 수 있을 겁니다. 따라서 패턴 하나를 공부할 때는 가능한 한 실제 예를 많이 접해야 합니다. 그리고 패턴을 적용하지 않은 경우에서 그 필요를 느끼고 설명할 수 있게끔 다양한 코드를 접해야 합니다.

10. 소프트웨어 개발에 푹 빠지기

패턴 중에 보면 서로 비슷비슷한 것이 상당히 많습니다. 그 구조로는 완전히 동일한 것도 있죠. 초보자를 괴롭히는 것 중 하나입니다. 이것은 외국어를 공부할 때 문법 중심적인 학습을 하면서 부딪치는 문제와 비슷합니다. '주어+동사+목적어'라는 구조로는 동일한 두 개의 문장, 즉 'I love you'와 'I hate you'가 구조적으로는 동일할지라도 의미론적으로는 완전히 반대가 될 수 있는 겁니다. 패턴을 공부할 때는 그 구조보다 의미와 의도를 우선해야 하며, 이는 다양한 실례를 케이스 바이 케이스로 접하면서 추론화 및 자신만의 모델화라는 작업을 통해 하는 것이 최선입니다. 스스로 문법을 발견하고 체득하는 것이라고 할까요.

DP는 사전과 같습니다. 이 책은 순서대로 소설 읽듯이 읽어나가라고 집필된 것이 아니고, 일종의 패턴 레퍼런스로 쓰인 것입니다. 역시 GoF의 한 명인 존 블리스사이즈(John Vlissides)는 다음과 같이 말합니다.

> 여기서 내가 강조하고 싶은 것은 디자인패턴, 즉 GoF 책을 어떻게 읽느냐는 것이다. 많은 사람들은 그 내용을 온전히 이해하기 위해서는 순서대로 읽어야 한다고 느낀다. 하지만 GoF는 소설이 아니라 레퍼런스 북이다. 독일어를 배우기 위해 독영 사전의 처음부터 끝까지 하나하나 읽으려고 하는 경우를 생각해 보라. 그렇게는 결코 배울 수 없을 것이다! 독일어를 마스터하기 위해서는 독일어 문화에 자기 자신을 푹 담궈야(immerse) 한다. 독일어로 살아야 하는 것이다. 디자인패턴도 똑같다. 그걸 마스터하기 이전에 소프트웨어 개발에 자신을 푹 담궈야 한다. 패턴으로 살아야 하는 것이다. 만약 꼭 그래야 한다면 소설 읽듯이 디자인패턴 책을 읽어라. 하지만 거의 아무도 그 방식으로 유창해지지 못한다. 소프트웨어 개발 프로젝트의 열기 속에서 패턴이 동작하게 하라. 실제 디자인 문제를 직면했을 때 그 패턴들의 통찰을 이용하라. 이것이 GoF 패턴들을 자신의 것으로 만드는 가장 효율적인 방법이다.

어떤 지식을 체화하기 위해선 그 지식으로 살아야 한다는 말을 확인할 수 있습니다. 영어를 배우려면 영어로 살고, DP를 배우려면 DP로 살라는 단순하면서도 아주 강력한 말입니다.

어떤 특정 문장 구조를 학습하는 데 최선은 그 문장 구조를 이용한 실제 문장을 나에게 의미 있는 실제 컨텍스트 속에서 많이 접하고 스스로 나름의 모델을 구축하여 교과서의 법칙에 '기쁨에 찬 동의'를 하는 것입니다.

주변에서 특정 패턴이 구현된 코드를 구하기가 힘들다면 이 패턴을 자신이 만지고 있는 코드에 적용해 보려고 노력해 볼 수 있습니다. 이렇게 해보고 저렇게도 해보고, 그러다가 오히려 복잡도만 증가하면 "아 이 경우에는 이 패턴을 쓰면 안 되겠구나"하는 걸 학습할 수도 있죠. GoF는 패턴을 배울 때는 한결

같이 "이 패턴이 적합한 상황과 동시에 이 패턴이 악용, 오용될 수 있는 상황"을 함께 공부하라고 합니다.

이런 식의 '사례 중심'의 공부를 위해서는 스터디 그룹을 조직하는 것이 좋습니다. 혼자 공부를 하건, 그룹으로 하건 커리프스키(Joshua Kerievsky)의 「A Learning Guide To Design Patterns」 http://www.industriallogic.com/papers/learning.html를 참고하세요. 그리고 스터디 그룹을 효과적으로 꾸려나가는 데는 스터디 그룹의 패턴 언어를 서술한 「Knowledge Hydrant」 http://www.industriallogic.com/papers/khdraft.pdf를 참고하면 많은 도움이 될 겁니다. 이 문서는 뭐든지 간에 그룹 스터디를 한다면 적용할 수 있습니다.

LG2DP(「A Learning Guide To Design Patterns」) 뒷부분에 보면 DP를 공부하는 순서와 각 패턴에서 던질만한 질문이 같이 정리되어 있습니다. DP는 순차적으로 공부해야만 하는 것은 아닙니다. 효과적인 공부의 순서가 있습니다. sorry라는 단어를 모르면서 remorseful이라는 단어를 공부하는 학생을 연상해 보세요. 외국어 공부에서는 자기 몸에 가까운 쉬운 단어부터 공략하는 것이 필수적입니다. 이런 걸 근접 학습(Proximal Learning)이라고도 하죠. 등급별 어휘 목록 같은 게 있으면 좋죠. LG2DP에서 제안하는 순서가 그런 것 중 하나입니다.

랄프 존슨은 이런 순서의 중요성에 관해 다음과 같은 말을 했습니다.

> … 하지만 나는 언제나 싱글톤 패턴을 가르치기 전에 콤포짓, 스트래터지, 템플릿 메쏘드, 팩토리 메쏘드 패턴을 가르친다. 이것이 훨씬 더 일반적인 것들이며, 대다수의 사람들은 아마도 이것들 중 마지막 두 가지를 이미 사용하고 있을 것이다.

11. 마이크로패턴

그런데 사실 GoF의 DP에 나온 패턴들보다 더 핵심적인 어휘군이 있습니다. 마이크로패턴이라고도 불리는 것들입니다. DP에도 조금 언급되어 있긴 합니다. 이런 마이크로패턴은 우리가 알게 모르게 매일 사용하는 것들이고 그 활용도가 아주 높습니다. 실제로 써보면 알겠지만, DP의 패턴 하나 쓰는 일이 그리 흔한 게 아닙니다. 마이크로패턴은 켄트 벡의 『Smalltalk Best Practice Patterns』에 잘 나와 있습니다. 영어로 치자면 관사나 조동사 같은 것들입니다.

그리고 이런 마이크로패턴과 함께 리팩토링을 공부하는 게 좋습니다. 리팩토링은 패턴의 필요를 느끼게 해줍니다. 제가 리팩토링 공부에서도 언급했지만 OAOO를 지키면서 리팩토링을 하다보면 자연스레 디자인패턴에 도달하는 경우가 많습니다. 이 때는 지나친 엔지니어링(Overengineering)이 발생하지 않고, 오로지 필요한 만큼만 생깁니다. 이에 관해서는 커리프스키의 「Stop Over-Engineering!」(Software Development Magazine, Apr 2002, http://www.sdmagazine.com/documents/s=7032/sdm0204b/0204b.htm의 일독을 권합니다. 리팩토링이 디자인패턴을 어떻게 생성해낼 수 있는지 보여주고 있습니다.

1980년대 이후 최근 알렉산더의 향방도 이런 쪽으로 가고 있습니다. 그는 자신이 발표한 기존 패턴들이 너무 크다고 생각해 그런 패턴들을 구성하고, 자동으로 만들어 내며, 또 관통하는 더 작은 원칙들을 발견하는 데 노력해 오고 있습니다. 코플리엔(James Coplien)은 컴퓨터계가 알렉산더의 최근 발전을 쫓아가지 못한다고 늘 이야기합니다.

제대로 된 패턴 구현을 위한 다양한 접근 시도하기 우리의 지식이라는 것은 한 가지 표현양상(representation) 으로만 이뤄져 있지 않습니다. 사과라는 대상을 음식으로도, 그림의 대상으로도 이해할 수 있어야 합니다. 실제 패턴이 적용된 '다양한 경우'를 접하도록 하라는 것이 이런 겁니다. 동일 대상에 대한 다양한 접근을 시도하라는 것이죠. 자바로 구현된 코드도 보고, C++로 된 것도 보고, 스몰토크로 된 것도 봐야 합니다. 설령 '오로지 자바족'이라고 할지라도요(전 이런 사람들을 자바리안(Javarian)이라고 부릅니다. 자바와 바바리안(barbarian)을 합성해서 만든 조어지요). 이런 '오로지 하나만 공부하는 것'의 병폐에 대해서는 존 블리스사이즈가 쓴 「Diversify」 http://www.research.ibm.com/people/v/vlis/pubs/gurus-99.pdf 라는 글을 읽어보세요. 이렇게 다양화를 해야 비로소 자바로도 '상황에 맞는' 제대로 된 패턴을 구현할 수 있습니다. 패턴은 그 구현(implementation)보다 의도(intent)가 더 중요하다는 사실을 꼭 잊지 말고, 설명을 위한 방편으로 채용된 한 가지 도식에 자신의 사고를 구속하는 우를 범하지 않기를 빕니다.

이런 맥락에서 저는 DP를 공부할 때 GoF와 동시에 『Design Patterns Smalltalk Companion』을 필수적으로 읽기를 권합니다. 두 권은 말하자면 양날개입니다. 하나는 정적언어로 구현되었고(간간이 스몰토크 구현이 있긴 합니다), 다른 하나는 동적언어로 구현되어 있습니다. 언어와 패턴의 고리를 느슨하게 하고, 패턴을 여러 관점에서 신선하게 볼 수 있는 계기가 될 것입니다. 또, 한 쪽을 보고 이해가 잘 되지 않을 때 다른 쪽을 보면 쉽게 이해됩니다. 서로 상보적인 것이죠.

패턴도 결국 '문제해결'을 위한 한 가지 방편에 지나지 않습니다. 주변에서 "이 경우에는 무조건 이 패턴을 써야 합니다"하고 생떼를 쓰는 사람을 보면 씁쓸한 기분을 감출 수 없습니다.

책 중에 중요한 서적을 몇 권 소개하겠습니다.

- 『Design Patterns Explained』(Shalloway, Trott): 최근 DP 입문서로 급부상하고 있는 명저
- 『Design Patterns Java Workbook』(Steven John Metsker): DPE 다음으로 볼만한 책으로 쏟아져 나오는 시중의 조악한 자바 패턴 책들과는 엄연히 다르다. 워크북 형식을 채용해서 연습문제를 풀고 뒷부분의 답과 대조해 볼 수 있는 등 독학자가 공부하기에 좋다.
- 『Refactoring』(Martin Fowler): DP 공부 이전에 봐서 문제의식 형성하기(망치를 들면 모든 것이 못으로 보이는 오류 탈피)
- 『Design Patterns』: 바이블.
- 『Design Patterns Smalltalk Companion』: GoF가 오른쪽 날개라면 DPSC는 왼쪽 날개
- 『Pattern Hatching』(John Vlissides): DP 심화학습. 얇지만 밀도 높은 책.
- 『Smalltalk Best Practice Patterns』(Kent Beck): 마이크로 패턴. 개발자의 탈무드. 감동의 연속.
- 『Pattern Languages of Program Design』 1,2,3,4: 패턴 컨퍼런스 논문 모음집으로 대부분은 인터넷에서 구할 수 있음
- 『Pattern-Oriented Software Architecture』 1,2: 아키텍처 패턴 모음. 2권은 네트워크 애플리케이션의 아키텍처 모음.
- 『Concurrent Programming in Java』(Doug Lea): 컨커런트 프로그래밍에 대한 최고의 서적.
- 『Patterns of Software』(Richard Gabriel): 패턴에 관한 중요한 에세이 모음.
- Analysis Patterns』(Martin Fowler): 비즈니스 분석 패턴 목록. 비즈니스 애플리케이션 개발자에게 많은 도움이 됨.
- 『A Timeless Way of Building』(Christopher Alexander): 프로그래머들이 가장 많이 본 건축 서적. 패턴의 철학적・이론적 배경. '구약'('신약'은 올해 안에 출간 예정인 동저자의 『The Nature of Order』).
- 『A Pattern Language』(Christopher Alexander): 알렉산더의 건축 패턴 모음집.
- 『Problem Frames』(Michael Jackson): DP의 해결(solution) 지향식의 문제점과 극복 방안으로서의 문제 지향식 방법. 마이클 잭슨은 요구 사항 분석 쪽에서 동명의 가수만큼이나 유명.

DP를 처음 공부한다면, DPE와 DPJW를 RF와 함께 보면서 앞서의 두 권을 RF적으로 독해해 나가기를 권합니다(하버드 대학의 뚜웨이밍 교수는 요즘 칸트를 유교적으로 독해하고 있다고 합니다. 하나의 책을 다른 각도에서 독해하는 것, 여기서 배우는 것이 많습니다). 이게 된 후에는 GoF와 DPSC를 함께 볼 수 있습니다. 양자는 상호 보완적인 면이 강합니다. 이쯤 되어 SBPP를 보면 상당히 충격을 받을 수 있습니다. 스스로가 생각하기에 코딩 경험이 많다면 다른 DP 책 이전에 SBPP를 먼저 봐도 좋습니다.

이 정도의 책을 봤다면 POSA와 PLOPD 등에서 자신이 관심이 가는 패턴들을 찾아 읽는 것이 좋습니다. 그리고 동시에 알렉산더의 원저들을 꼭 읽기를 권합니다. 가브리엘의 책이 알렉산더의 사상 이해에 많은 도움이 될 것입니다.

패턴 공부를 해나가면서 남을 가르치는 것이 공부에 많은 도움이 됩니다(사실 자바 패턴 책 중에 어떤 것은 "내가 패턴을 처음 공부하면서 같이 쓴 것이다"라고 저자가 고백한 것도 있습니다). 보이스카웃에서는 보통 다음 과정을 통해 뭔가를 '학습'하게 한다고 합니다. 처음은 어떻게 하는지를 보여주고, 다음은 스스로 그것을 해보게 하고, 다음으로 그걸 남에게 가르치게 합니다. 이 때 중요한 것은 상대가 이해하지 못 하면 그 이유를 자기 자신에게서 찾는 것이 나에게 더 이득이 된다는 것입니다. "내가 설명을 잘못 했군"하고 생각하는 것이죠. 그러면 다음번에는 설명을 좀더 잘 할 수 있게 되고, 동시에 자기의 이해도 더욱 명료해지게 됩니다. 저는 'OOP 개념을 한 시간 만에 가르치기'나 '특정 언어 문법을 한 시간 만에 가르치기' 등을 하나의 도전으로 생각하고 즐깁니다. 가르치면서 동시에 배운다는 것은 정말 놀라운 경험입니다.

익스트림 프로그래밍 학습에서의 문제 앞의 경우와 비슷합니다. 익스트림 프로그래밍을 공부하는 사람들은 실제로 행해보지 않고 책만 들여다보면 안 됩니다. 그렇다고 책이 중요하지 않다는 말은 아닙니다. 하지만 자전거 타기는 자기 몸으로 직접 굴려봐야 합니다.

게다가 켄트 벡 스스로가 『XP Explained』를 만약 다시 쓴다면 뜯어고치고 싶은 부분이 상당히 된다고 말하는 것을 봐도 알 수 있듯이 초기 XP 이후 바뀌고 보완된 점이 상당수 있습니다. 따라서 책만으로 XP를 공부하기는 힘듭니다. 지금은 책 속의 XP가 사람들의 머리 속 XP에 한참 뒤쳐져 있습니다.

어찌 되었든 XP에는 무술이나 춤, 혹은 악기 연주 등과 유사한 면이 많습니다. 따라서 글을 보고 그것을 익히기는 쉽지 않습니다. 그나마 메일링 리스트 같은 '대화'를 보면 훨씬 더 많은 것을 얻을 수 있기는 하지만, 태권도 정권 찌르기를 말로 설명하는 것이 불가능에 가깝듯이 XP를 언어를 통해 익히기는 정말 어렵습니다. 우리의 언어는 너무도 성글은 미디어입니다(XP는 매 초, 매 순간 벌어지는 '일상적' 장면의 연속이 매우 중요합니다).

13. 익스트림 프로그래밍 공부

XP를 이해하려면 다음 기본 자료에 대한 이해가 우선해야 합니다(본지 2001년 12월호 「허실문답 XP 강화」 참조).

- 『XP Explained』(Kent Beck): XP 선언서
- 『XP Installed』(Ron Jeffries et al): C3 프로젝트에 적용한 예, 얻은 교훈 등
- 『Planning XP』(Kent Beck, Martin Fowler): 계획 부분 설명(관리자, 코치용)
- 『Refactoring』(Martin Fowler): 리팩토링에 대한 최고의 책
- 『XP Applied』: 유즈넷과 메일링 리스트의 논의 등 최근 자료를 반영
- 『XP Explored』: 가장 쉽고 구체적인 XP 안내서

이 중에서 XPI나 XPX를 먼저 권합니다. XPE는 좀 추상적인 서술이 많아서 봐도 느낌이 별로 없을 수 있습니다(2001년 본지 11월호에 제가 쓴 리뷰 참고). 여유가 되면 다음 자료를 더 참고합니다.

- 『The Timeless Way of Building』: 패턴 운동을 일으킨 알렉산더의 저작. 현장 고객(On-site Customer), 점진 성장(Piecemeal Growth), 소통(Communication) 등의 아이디어가 여기에서 왔음.
- 『XP in Practice』(Robert C. Martin 외): 두세 사람이 짧은 기간 동안 간단한 프로젝트를 XP로 진행한 것을 기록. 자바 사용(중요한 문헌은 아님).
- 『XP Examined』: XP 컨퍼런스에 발표된 논문 모음
- 『Peopleware』(Tom DeMarco): 개발에 있어 인간 중심적 시각의 고전
- 『Adaptive Software Development』(Jim Highsmith): 복잡계 이론을 개발에 적용. 졸트상 수상.
- 『Surviving Object-Oriented Projects』(Alistair Cockburn): 얇고 포괄적인 OO 프로젝트 가이드라인
- 『Software Project Survival Guide』(Steve McConnell): 조금 더 전통적인 SE 시각.
- 『The Psychology of Computer Programming』(Gerald M. Weinberg): 프로그래밍에 심리학을 적용한 고전. 코드 공유와 짝 프로그래밍에 필수인 비자아적 프로그래밍(Egoless Programming)이 여기서 나왔다.
- 『Agile Software Development』(Alistair Cockburn): 전반적 애자일 방법론에 대한 개론서
- 『Software Craftsmanship』(Pete McBreen): 장인으로서의 새로운 프로그래머 상
- 『Agile Software Development with SCRUM』(Schwaber Ken): 최근 확장성(Scalability)을 위해 XP+SCRUM의 시도가 애자일 쪽의 큰 화두임.
- 『A Practical Guide to eXtreme Programming』(David Astels 외): 저자들이 직접 참여한 프로젝트를 따라가 보면서 배움. 자바로 구현. XPP보다는 스케일이 큼.
- 『Agile Modeling』(Scott Ambler): 애자일 쪽에서 모델링이 무시되는 느낌이 있을 수 있는데, 그 쪽으로 깊게 천착한 사람이 앰블러임.
- 『Agile Software Development Ecosystems』(Jim Highsmith): 각각의 애자일 방법론에 대한 소개와 동시에 각 방법론의 대표자들 인터뷰가 백미.

- 『Test Driven Development』(Kent Beck): 곧(아마 올해 내에) 출간 예정인 최초의 TDD 서적. TDD를 모르면 XP도 모르는 것(TDD를 실제 적용하려면 적어도 반년 정도는 계속 훈련해야 함)
- IEEE Software/Computer, CACM, Software Development Magazine 등에 실린 기사
- 『XP Conference, XP Universe 등의 논문들(특히 최근 것들)
- 유즈넷, 메일링 리스트, 오리지널 위키 http://c2.com의 논의들

특히 유즈넷, 메일링 리스트, 오리지널 위키는 늘 가까이 하고 있어야 합니다. 이 세 곳을 살필 때, 특히 다음 사람들의 글은 꼭 읽어보고 항상 레이더를 열어두면 좋습니다(이 외에도 개발 경력 10년, 20년이 넘는 짱짱한 사람이 많으므로 눈 여겨 관찰하세요. 모든 글을 읽는 것은 무리이므로 그들의 대화를 일차적으로 읽어야 합니다).

켄트 벡, 론 제프리즈(Ron Jeffries), 워드 커닝엄(Ward Cunningham), 앨리스테어 코번(Alistair Cockburn), 마틴 파울러, 로버트 마틴 혹은 엉클 밥(Robert C. Martin aka Uncle Bob), 마이클 페더즈(Michael Feathers), 켄 아우어(Ken Auer), 윌리엄 웨이크(William Wake), 로이 밀러(Roy Miller), 데이브 토마스(Dave Thomas), 앤디 헌트(Andy Hunt), 랄프 존슨, 스카트 앰블러(Scott Ambler), 짐 하이스미스(Jim Highsmith), 조슈아 커리프스키(Joshua Kerievsky), 로렌트 보사빗(Laurent Bossavit), 존 브루어(John Brewer) 등

이런 자료들 외에, 기회가 된다면 주변에서 XP를 직접 사용하는 곳을 방문해서 하루만 같이 생활해 보기를 권합니다. 반년 공부를 앞당길 수 있습니다. 제가 공부할 때는 주변에 XP를 아는 사람이 없어서 혼자 공부했는데, 그것에 비해 XP를 직접 사용하는 상황에 처한 사람은 (제가 억울하리만큼) 더 적은 노력으로 몇 배 이상 빨리 몸에 익히는 것 같았습니다.

이게 힘들면 같이 공부하는 방법이 있습니다(앞서 언급된 스터디 그룹에 관한 패턴 참고). 이 때 같이 책을 공부하거나 하는 것은 시간 낭비가 많습니다. 차라리 공부는 미리 다 해오고 만나서 토론을 하거나 아니면 직접 실험을 해보는 것이 훨씬 좋습니다. 두 사람 당 한 대의 컴퓨터와 커대란 화이트보드를 옆에 두고 말이죠. 제 경우 스터디 팀과 함께 저녁 시간마다 가상 XP 프로젝트를 많이 진행했고, 짤막짤막하게 프로그래밍 세션도 많이 가졌습니다. 나중에 회사에서 직접 XP를 사용할 때 많은 도움이 되었습니다.

14. Refactor Me

제가 하고 싶은 말은 더 있지만 이 글은 일단 여기서 끝이 납니다. 서두에서 말씀드렸듯이 애초 쓰려던 '일반론'은 생략하고, 대신 독자들의 몫으로 남겨두려 합니다. 이 글 자체가 여러분의 리팩토링 수련의 연장(延長)이 되었으면 하는 바램입니다. 이 글과 함께 실생활에서 직접 실험을 해보면서 - 이 때 욕심 부리지 않고 한 가지씩 지긋이 해보는 느긋함과 음미의 정신이 필요할지도 모르겠습니다 - 자신의 경험을 축적해 나가고, 동시에 이 글을 적절히 리팩토링해서 자신만의 패턴을 차근히 만들어 나가길 바랍니다. 그렇습니다. 리팩토링은 대상에 대한 이해가 깊고 경험이 많을수록 더 잘할 수 있습니다.

이 글에 소개된 제 공부론은 어찌 보면 상당히 진부해 보이기도 할 것입니다. 하지만 저는 이런 상식적이고 일상적이며 심지어는 소소해 보이는 것들에서 많은 감동을 받아왔습니다. 이 글도 사실 제 감동의 개인사입니다. 저는 "만약 오늘 어떤 것에라도 감동한 것이 없었다면, 오늘은 뭔가 잘못 산 것이다"라는 신조를 갖고 있습니다. 그것이 컴퓨터이건 대화이건 상관없이 말이죠. 저는 날마다 감동하며 살려고 노력합니다. 그러나 이 감동에 뭔가 꼭 대단한 것이 필요한 것은 아닙니다. 저는 오히려 감동 받기 위해 스스로 대단해져야 할 필요를 느끼기도 합니다. '감동'이라는 것은 주어지는 것이 아닙니다. 나와 타자가 공조하여 만드는 대화입니다.

감동해야 체득할 수 있다고 생각합니다. 그리고 이 감동은 개인적 삶 속에서 자기가, 자신의 몸으로, 직접 얻는 것입니다. 工夫 열심히 합시다.