**소프트웨어 개발자 이야기**

http://blog.naver.com/knix008



**Volume I**

2018.09.02

**권수호**

**(**[suho.kwon@gmail.com)](mailto:suho.kwon@gmail.com))

**Copyright@2018, 2019**

본 문서의 내용을 인용하거나 상업적인 목적으로 사용하기 위해서는 저자와의 협의가 있어야 합니다. 개인적으로 프린트해서 읽는 것은 자유입니다. 가치 있는 글을 쓰도록 많은 사람들께서 도움을 주신다면 고맙게 생각하겠습니다.

목차

Part I. People 12

[ 소프트웨어 아키텍트(Software Architect) ] 13

[ 신입사원 지원자들에게 ] 15

[ 지식의 발전 과정에 대한 단상 ] 17

[ 초보 팀장의 팀원 관리 방법 ] 19

[ 소프트웨어 개발자의 삶이란? ] 20

[ 잘 안되는 회사는 어떤 회사일까? ] 24

[ 개발자가 듣고 싶지않은 말들 ] 24

[ 소프트웨어 개발의 창의성에 대해 ] 27

[ 업무에 관한한 거짓말은 하지 말라. ] 29

[ 리더의 조건 ] 31

[ 잔업에 대한 생각 ] 34

[ 오늘은 몇시간 일했나요? ] 36

[ 상사의 꾸지람에 대처하는 방법 ] 37

[ 그래서, 도대체 대책이 뭐야? ] 38

[ 행복이란? ] 39

[ 개발자의 마인드(Mind)? ] 41

[ 성공의 기준은? ] 42

[ 좋은게 좋은거다? ] 44

[ 해가 될 것인가 바람이 될 것인가? ] 45

[ 개발인가? 관리인가? ] 47

[ 갑자기 “집에 갔다”는 말을 듣는다는 것 ] 48

[ 새로운 조직을 맡게 된다면 ] 49

[ 잘듣는 것의 중요함 ] 50

[ 결정(Decision)해야하는 순간 ] 52

[ 개발자에게 해서는 안되는 말 ] 53

[ 가르침이란 ? ] 56

[ OSD: Ownership, Speed, Detail ] 57

[ 누가 고양이 목에 방울을 달 것인가? ] 59

[ 아무도 답을 주지는 않는다. ] 60

[ 기본으로 돌아가자. ] 61

[ 휴가에 대해서 ] 63

[ 삶의 여백(Slack) ] 64

[ 교육의 목적 ] 72

[ 지식의 실체 ] 73

[ 팀워크 ] 74

[ 매크로 관리 vs. 마이크로 관리 ] 76

[ 프로젝트의 실패란? ] 77

[ 라이언 일병 구하기 ] 77

[ 조직개편 ] 78

[ 인연 쌓기 ] 79

[ 협상의 시간 ] 80

[ 폭언에 맞서는 방법 ] 82

[ 일을 적게 하는 것이 성공하는 비결 ] 85

[ 소프트웨어 개발자의 일상 ] 87

[ 프로근성이란? ] 89

[ 피플웨어(Peopleware) ] 90

[ 공부를 하자. ] 91

[ 즐기는 자(논어의 한 구절)? ] 92

[ 고과 면담 ] 94

[ 이공계를 기피하는 현상? ] 95

[ 개발자를 흥이나게 만드는 것 ] 97

[ 장인정신 ] 98

[ 듣고 말하기 vs. 말하고 듣기 ] 99

[ 존중 받으려면 먼저 존중해야 한다. ] 101

[ 삶의 질(Quality) ] 102

[ 소프트웨어 개발자의 최종 목표 ] 103

[ 존중이란? ] 105

[ 무엇이 행복을 만드나? ] 107

[ 관리자의 생산성은 어디에? ] 108

[ 슈퍼맨(Superman)? ] 109

[ 대화가 필요해 ] 111

[ 비젼(Vision) ] 112

[ 개과천선 ] 113

[ 퇴직 사유서에 적힌 내용에 대한 단상 ] 114

[ 잘되는 회사의 특징 ] 116

[ 야근에 대한 단상 ] 129

[ 소프트웨어 공학에 대한 교육이 필요하다. ] 130

[ 신입사원이 배워야 할 것들 ] 132

[ 요원009에 대한 단상 ] 133

[ 프로 정신? ] 134

[ 상사 까기(?) ] 136

[ 소프트웨어 개발에서의 정치(?) ] 137

[ 이론과 실무 ] 139

[ 소프트웨어 개발자의 면접 ] 140

[ 주말 특근에 대해서 ] 142

[ 소프트웨어 장인 정신 ] 143

[ 사소한 일에 목숨 걸지말라!!! ] 145

[ 관리자의 역할 ] 146

[ 소프트웨어 개발자 문화 ] 148

[ 전문가와 비전문가의 차이 ] 150

[ 이직하는 사람들의 공통점 ] 151

[ 변명 혹은 진실? ] 152

[ 팀워크 ] 154

[ 문제를 대하는 태도 ] 155

[ 문제가 있는 과제를 떠맡다. ] 157

[ 슬랙(Slack) : 삶의 멋과 여유 ] 158

[ 소프트웨어 개발자의 목표 ] 160

[ 숟가락만 들고 있어서는 안된다. ] 162

[ 회사가 팔리다. ] 163

[ 통제 혹은 관리? ] 165

[ 프로(Professional) 프로그래머 ] 167

[ 존중, 팀의 시작 ] 169

[ 대화가 중요하다. ] 170

[ 지식 노동자? ] 173

[ 잘리다. ] 175

[ 바쁘면 진다. ] 176

[ 직업 안정성(Job Security) ] 178

[ 학교와 회사의 차이 ] 180

[ 명령과 선택 사이 ] 181

[ 어려운 시간이 오면 진실이 드러나게 된다. ] 183

[ 믿을 것은 자신밖에 없다. ] 184

[ 좋은 소프트웨어 개발자가 되고 싶다면 ] 186

[ 회사에서의 교육 ] 187

[ 처음 누구를 만나느냐가 중요하다. ] 189

[ 무엇이 되고 싶은가? 무엇을 하고 싶은가? ] 191

[ 하고 싶은 일을 하자. ] 192

[ 견제와 균형 ] 194

[ 책임과 권한 ] 196

[ 새로운 출발 ] 197

[ 소프트웨어 개발과 인간성 ] 199

[ 열심히 일할수록 망한다? ] 201

[ 책임이란? ] 203

[ 더 이상 물러설 곳은 없다 ] 205

[ 권한의 위임(Delegation) ] 207

Part II. Process 210

[ 문서 쓰기 ] 211

[ 회의 및 보고서를 없애자 ] 213

[ 과제 관리 ] 214

[ 누구를 위해 종은 울리나? ] 216

[ 급한 것과 중요한 것? ] 218

[ 낙관과 비관 사이 ] 221

[ 벽을 넘어서 ] 222

[ 품질과 기능 그리고 자원의 문제 ] 222

[ 코드리뷰는 어떻게 하지? ] 224

[ 지속적인 통합(CI)이란? ] 228

[ 글쓰기 연습이 필요하다. ] 230

[ 애자일(Agile)하게 일하기 ] 232

[ 소프트웨어 프로젝트 성공의 핵심은? ] 234

[ 천망회회 소이불실( 天網恢恢 疎而不失 ) ] 238

[ 시간과의 싸움에서 이기는 방법 ] 240

[ 조금씩 일하기 ] 242

[ Good to Great ] 243

[ 문제에 대한 인식과 공유 ] 244

[ 다 타버리다(Burnout) ] 247

[ 일을 적게해야 프로젝트가 성공한다. ] 249

[ 외주 과제의 관리 ] 249

[ 소프트웨어 플랫폼은 왜 만드는가? ] 250

[ 프로젝트가 실패하는 이유? ] 252

[ 규칙과 규율(Rule & Discipline) ] 253

[ 과제의 일정 지키기 ] 256

[ 무엇이 문제인지 아는 것이 문제풀이의 시작 ] 257

[ 소프트웨어 개발은 사람에 대한 이해다. ] 259

[ 생태계의 승자가 되는 길 ] 261

[ 린(Lean) 소프트웨어 개발 ] 263

[ 객체지향 프로그래밍에 대해서 ] 265

[ 과제를 시작할 때 가장 먼저 해야 할 일 ] 266

[ 소프트웨어 생산성에 대해서 ] 267

[ 우리는 시스템으로 일한다? ] 269

[ 과제 관리 시스템의 사용 ] 272

[ 테스트 자동화 시스템을 구축하자 ] 274

[ 요구사항과 과제 일정에 대해서 ] 276

[ TDD(Test-Driven Development)에 대해-01 ] 278

[ 개발자 테스트에 대한 단상 ] 280

[ 지금이 가장 적절한 순간이다 ] 282

[ 개발자 테스트(지난 날의 회고) ] 283

[ 지속적인 통합과 배포 ] 286

[ 시간의 법칙 ] 287

[ 한 번에 한 가지만 하기 ] 289

[ 해야 할 일과 하지 않아야 할 일 ] 291

[ 빠른 피드백(Feedback) ] 292

[ 테스트 자동화 ] 293

[ 유지보수(Maintenance)에 대해서 ] 294

[ 버그와 생산성에 대해서 ] 296

[ TDD(Test-Driven Development)에 대해-02 ] 297

[ 개발 생산성에 대해서 ] 300

[ 중요한 일과 급한 일 ] 307

[ 성공 스토리 ] 308

[ 코드 리뷰에 대한 오해 ] 310

[ 개발 일정에 대해서 ] 311

[ 빈익빈 부익부 ] 313

[ 동작하는 소프트웨어를 제대로 만들기 ] 314

[ “Man-Month” 미신? ] 316

[ 긍정적 일정계획과 비관적 추정 ] 317

[ 축구와 소프트웨어 개발이 닮은 점? ] 319

[ 실패하는 프로젝트의 시작 ] 320

[ 지속적인 개선에 대해서 ] 322

[ 계획과 관리 ] 323

[ 생존전략 : 절망을 받아들이는 방법 ] 324

[ 시작이 반이다. ] 326

[ MDD(Meeting Driven Development) ] 327

[ 과제 일정 준수는 가능한 일인가? ] 328

[ 외주 + 관리 ] 330

[ 창의력과 규율의 사이에서 ] 331

[ 좌로 이동(Left Shift) 전략 ] 332

[ 코드의 공유 ] 334

[ 일정 압박을 통해서 얻는 것은 없다. ] 335

[ 린(Lean)개발 방식 ] 337

[ 개선, 어디서부터 시작할 것인가? ] 338

[ 정도에 지름길은 없다 ] 340

[ 소프트웨어 개발 전략 ] 342

[ 프로세스 ] 344

[ 쉽게 가는 지름길은 없다. ] 345

[ 스크럼(SCRUM)을 시작하다. ] 347

[ 시간은 언제나 부족하다. ] 349

[ 측정해야 개선할 수 있다? ] 350

[ 밀어넣기와 빼오기(Push vs. Pull) ] 352

[ 초기 품질이 전체를 좌우한다. ] 355

[ 기능위주 개발전략 ] 357

[ 하고 있는 일 줄이기 ] 359

Part III. Product 361

[ 글로벌(Global) 소프트웨어? ] 362

[ 운영체제는 누가 움직이지? ] 364

[ 운영체제의 역할 ] 365

[ 프로그래머 십계명 ] 366

[ Model-View-ViewModel을 배워볼까? ] 372

[ 역시 살길은 서비스 ] 373

[ 학생들이 알아야 할 열가지 ] 376

[ 네트워크의 본심 ] 384

[ 간결함의 힘 ] 385

[ 어떻게 짜야 잘 짰다는 소리를 들을까? ] 388

[ 테스트 주도 개발(TDD) ] 390

[ 나는 프로그래머다. ] 392

[ Linux 커널을 아는가? ] 396

[ 소프트웨어 플랫폼이 뭐길래? ] 397

[ 나는 객체지향으로 일하는가? ] 419

[ 코딩의 법칙 ] 420

[ 소프트웨어 설계는 얼마나 해야할까? ] 424

[ 객체지향으로 가다. ] 426

[ 모델(Model) 추가에 따른 분기문의 문제점 ] 428

[ 버그에 대한 대책 ] 429

[ 글쓰기에서 도식화로 ] 431

[ 달을 가리키는데, 손끝을 보지말라!!! ] 432

[ 코딩 전략 ] 434

[ 싱글턴(Singleton)에 대한 생각 ] 436

[ 코드에서 발생하는 역참조 ] 438

[ 테스트 케이스에 대한 생각 ] 439

[ 성능(Performance) vs. 구조(Structure) ] 440

[ 절차지향과 객체지향 ] 443

[ 객체지향에 대한 단상 ] 444

[ 객체(Object)란? ] 445

[ 소프트웨어 설계에 대한 단상 ] 446

[ 계층 구조에 대해서 ] 448

[ 전략적 코딩 vs. 전투적 코딩 ] 450

[ 버그(Bug)를 바라보는 자세 ] 452

[ Good Code vs. Bad Code ] 454

[ 창조 vs. 모방 ] 455

[ 모델링(Modeling)이란? ] 456

[ 소프트웨어를 만든다는 것 ] 458

[ 객체지향으로 생각하기 ] 459

[ 사용자를 위해서 ] 460

[ 나쁜 코드? 아니면, 내가 짠 코드? ] 461

[ 좋은 구조 vs. 나쁜 구조 ] 463

[ 단순하게 살자 ] 465

[ 소프트웨어 설계의 원리 ] 466

[ 소프트웨어 개발=코딩+테스트+그리고,문서 ] 469

[ 낡은 코드의 관리를 어떻게 할 것인가? ] 471

[ SOLID에 대해서 ] 473

[ 좋은 이름 짓기 ] 475

[ 주석(Comment)에 대한 오해 ] 476

[ 함수의 길이에 대한 단상 ] 478

[ 함수의 인자수는 몇개가 좋을까? ] 480

[ 객체지향 이야기 ] 481

[ 코딩의 길을 묻다 ] 484

[ 코딩의 기본 : 분리와 응집 ] 485

[ 좋은 코드와 그렇지 않은 코드 ] 486

[ 플랫폼 개발이 실패하는 이유 ] 490

[ 단위 테스트에 대해서 ] 494

[ 이중화 vs 원가절감 ] 500

[ 디자인 패턴(Design Pattern)에 대해서 ] 502

[ 복잡한 문제 풀기 ] 503

[ 소프트웨어 설계 원칙 ] 504

[ 소프트웨어 품질에 성능만 있는 것이 아니다. ] 513

[ 이식성(Portability)이 좋은 코드 ] 514

[ 나만의 도구 상자 ] 515

[ 세계최고? 세계 최초? ] 517

[ 프로그래밍 언어의 선택 ] 518

[ 코드 개선에 대해서 ] 519

[ 고품질의 소프트웨어 만들기 ] 522

[ 프로그래밍 배우기 ] 523

[ 간결함(Simplicity)에 대한 단상 ] 525

[ 유지보수에 대한 이야기 ] 526

[ 버그가 많이 발생했다. ] 528

[ 코드에 옷을 입히다. ] 529

[ 오래되고 익숙한 툴 vs. 새롭고 낯선 툴 ] 531

[ 프로세스 vs. 코딩 ] 532

[ 내가 만들지 않은 코드가 최고(?) ] 534

[ 분석툴을 믿어야 한다. ] 536

[ 코드가 말한다(?) ] 537

[ 프로그래밍에서의 낭비 ] 539

[ 버그를 줄이는 방법 ] 539

[ 코드의 낭비 줄이기 ] 542

[ 테스트를 바라보는 관점 ] 544

[ TDD(Test Driven Development)에 대해-03 ] 545

[ 결함은 개인의 문제? ] 547

[ "간소함"에 대해서 ] 549

[ 코딩에 대한 단상 ] 550

[ 테크닉 vs. 원리 ] 552

[ 생각을 정리한다는 것 ] 554

[ 소프트웨어 설계의 기본이란? ] 556

[ 의존성이란? ] 558

[ 하드웨어에 투자하라. ] 560

[ 자동화가 주는 이점 ] 562

[ 아름다운 코드(Beautiful Code) ] 564

[ 버그와 생산성 ] 566

[ 환경과 코드의 분리 ] 568

**머리말**

소프트웨어 개발자로 일한지 20년이 넘었다. 많은 경험을 했다고는 할 수 없겠지만, 그래도 나름 다양한 경험을 했다는 것은 말할 수 있을 것 같다. 20년 동안 다양한 분야의 회사를 다녔으며, 실무 개발자에서 기획, 개발 실무 책임자, 중간 리더, CTO등의 역할도 했었다. 그런 모든 것이 경험이라고 한다면 적어도 같은 길을 걸을지도 모르는 사람들에게 조그마한 도움이 될 만한 것을 남기고 싶은 것은 욕심이 아닐 것이다. 때로는 성공했던 것들도 있고, 때로는 심하게 좌절한 것들도 있었다. 그런 경험들이 존재했기에 이렇게 글로 남겨서 누군가에게 도움이 되었으면 하는 바램도 생기게 되었을 것이다.

이 글은 누군가의 경험이지만 그 누군가에게만 한정된 경험은 아니다. 현재를 살아가는 소프트웨어 개발자들이 실무에서 마주할 수도 있는 다양한 경험에 대한 것이다. 그리고, 성공보다는 실패한 경험에 더 가깝다. 왜 실패를 했으며 어떻게 하면 실패로부터 조금이라도 멀리갈 수 있는지를 알려주는 이야기일 뿐이다. 개인의 이야기지만 개인이 아닌 사람들에게도 유용할 수 있는 것들이며, 또한 개인적인 경험으로 한정해도 충분히 의미를 찾을 수 있을 것이다. 유사한 배경을 가지고 같은 분야에서 일하는 사람들은 비슷한 고민을 하고 있을 가능성이 높다. 따라서, 나 자신의 고민을 이야기한다고 해서 그들의 고민과 크게 다르지는 않을 것이고 본다.

사람들의 경험이 제대로 전달되지 못하고 시행착오를 지속하는 이유는 실패를 솔직하게 이야기하지 않기 때문이다. 이 책에서 이야기하는 대부분의 경험은 실패에 대한 것이며, 다른 사람은 성공하기를 기원하는 마음으로 블로그 글들을 정리했다. 블로그의 특성상 반복적인 내용이 있을수도 있지만, 여러 번 이야기하는 이유 또한 명확하다. 중요하지 않은 것들은 반복할 이유도 없기 때문이다. 큰 일은 작고 잘 정의된 작은 일의 반복으로 이루어진다. 코드 또한 작고 간결한 것이 이상적이다. 작은 이야기들이지만 결코 의미까지는 작다고 생각하지 않기를 바라며, 이 책의 모든 독자들이 자신들의 실패를 부끄럽지 않게 이야기할 수 있도록 기원해 본다. 마지막으로 책을 쓴 사람이 개발자 출신이라고 해서 반드시 개발자만 이 책을 읽을 필요는 없다. 누군가의 실패는 또 다른 누군가에게는 소중한 간접 경험이 될 수 있다는 것만으로도 충분히 훌륭한 이야기될 수 있을 것이다.

**감사말**

블로그 글의 편집과 좋은 의견을 주신 김기호님에게 감사드립니다. 이 책을 대전 현충원에 계신 아버지와 자주 뵙지 못하는 어머니, 두 아들 준엽 준우, 아내 만정에게 드립니다.

# Part I. People

# [ 소프트웨어 아키텍트(Software Architect) ]

회사 마다 사정은 다르겠지만 항상 높으신 분들은 소프트웨어의 중요성을 말하시면서, 소프트웨어가 안보여서 잘 모르기는 하지만, 마치 공기와 같이 중요하다라는 표현을 가끔 사용한다. 글쎄? 안 보인다는 표현은 좀 그렇다. 사실은 잘 모른다는 표현이 더 맞을 것이다. 예전에 인도를 여행하면서 여행지를 읽을 때가 기억난다. 어떤 곳을 가든지 그 곳에 대한 정보를 알지 못 하고선 그냥 지나쳐 버리기 쉬운 것들이 많기 때문이다. 그래선지 항상 어떤 곳을 가기 전에 혹은 가는 동안에 세세한 정보를 모두다 챙기는 버릇이 생겼다. 그렇게 사전에 좀 공부하고 가면 생각보다 재미있는 여행이 되기 때문이다. 소프트웨어도 마찬가지가 아닐까? 모르고 안 보인다고 이야기할 것이 아니라, 좀 익히고 다시 보는 센스 정도는 있어야 한다. “세상은 아는 만큼 보이기 때문이다.”

소프트웨어 아키텍트라는 직업은 존재하지 않는다. 그렇다고, 그런 사람이 회사에 없다고는 하지 못한다. 즉, 어떤 사람이 "당신네 회사에는 소프트웨어 아키텍트정도 되는 사람이 몇 명이나 있어요?"라고 묻는다면, 대충 1명(좀 작은 회사에서는)은 있다고 이야기한다. 물론, 그 사람이 구체적으로 누구 인지도 머릿속에 떠오를 것이다. 그럼, 그러한 사람이 있다는 것을 어떻게 마음속에 떠올리게 될까? 사실 그게 더 중요하다. 우리가 소프트웨어를 개발하는 동안에 결국은 누군가의 도움을 받고, "아~, 역시 그 사람 능력 있어."라고 생각할 때, 그 사람이 바로 소프트웨어 아키텍트로 마음속으로 생각된다는 것이다.

물론, 모든 사람이 그렇다는 것은 아니다. 프로그램 능력이 대단한 사람이라고 해서 반드시 소프트웨어 아키텍트가 되지는 못한다. 뭔가 빠진 듯이 보이는 그런 사람들은 아직 내공이 더 쌓여야 한다. 만약, 그런 사람이 안 보인다면 이제 회사를 나올 생각을 해야 한다. 아무것도 배울 수 없는 회사는 그냥 그런 회사다. 배고픔을 잠시 잊게 하고 가족들에게 최소한의 안도를 주는 회사인 것이다.

어쨌든, 요즘엔 소프트웨어 아키텍트라는 직함이 어렴풋이 여기 저기서 보이기 시작한다. 아키텍처가 좋니 나쁘니, 플랫폼(Platform)이나 프레임워크(Framework)을 만들어야 한다는 말을 듣는다. 하지만, 정말 그런 것들이 보이기 시작한다면, 마음속으로 그릴 수 있는 인재를 회사에서 키울 수 있도록 만들어 주어야 한다. 공부는 끝이 아니라 새로운 시작을 위한 밑거름인 것이다.

소프트웨어 아키텍트는 정말 그럴싸한 말이지 않은가? "소프트웨어를 설계하는 사람”. 설계를 잘하는 사람이 있다면, 구현을 전문으로 하는 사람도 있다는 말이 된다. 누군가의 우아한 그림을 열심히 묵묵히 구현하고 검사 받는 사람들도 있다. 하지만, 현실은 어떤가? 대부분의 PL(프로젝트 리더)이 불명확한 요구사항을 가지고, 지금까지 경험으로 알고 있는 지식으로 대략적인 밑그림을 그리고, 이를 될 때까지 구현하고 있지 않은가? 아마 대부분의 개발자들은 그런 PL밑에서 일하고 있을 것이다. 나도 물론 예외가 아니라고 본다. 좀 더 나은 점이 있다면, 일정 계획에 지나치게 매달리지 않는다는 것이 다를 뿐. 또 다른 차이점이라면 출퇴근 시간도 딱히 규정하고 있지 않다는 것도 포함될 수 있을지도 모르겠다.

시작은 항상 “대화의 부족”이다. 모든 사람 사이의 문제는 이해의 부족에서 생기며, 상대방을 나의 잣대로 재어보려고 하는데서 생긴다. 모르면 알 때까지 물어봐야 하고, 전문가의 의견은 자신의 밑에 있는 팀원이라고 무시해서는 안 된다(간혹, 옆에 있는 직원보다 밖에서 훈수 두는 사람의 말을 더 잘 믿는 경향이 있다.). 아직도 부족한 것은 대화의 방법이며, 목적을 이해시키는데 필요한 다양한 수단 들이다. 어쨌든 소프트웨어 아키텍트라는 사람도 별반 다르지 않다. 소프트웨어가 제대로 만들어지도록 만들고, 필요하면 여기 저기 손을 걷어붙이고 나서야 한다.

따라서, 역시 "소프트웨어는 사람이 만드는 것"이다. 주어진 형식과 틀을 가지고, 모든 것이 프로세스 내에서 결정된다는 생각은 버리는 것이 좋다(절대 필요 없다는 것이 아니다.). 프로세스는 최소한의 도구이지, 성공으로 가는 지름길이 될 수는 없다. "성공한 소프트웨어 프로젝트는 사람을 남긴다."라는 말을 여기서 하고 싶다. 팔고 사는 일이 아닌 사람의 노력의 결정판(결과물)이 소프트웨어이기 때문이다.

소프트웨어 업계에 종사하는 사람들의 최종 목표는 무엇일까? 계속 개발자로 남는 사람일까? (즉, 코딩을 정말 잘해서 코딩 세상에서 뼈를 파묻고자 하는 사람) 아니면, 여러 사람과 같이 프로젝트를 하면서 관리자가 되는 것일까? 엔지니어에게 2개의 커리어패스(career path)가 있다면, 아마도 앞에서 나온 두 가지의 예가 그에 해당할 것이다. 최근의 경향에서 읽히고 있는 사실은 서서히 두 가지를 나누어서 보는 시각이 생기고 있다는 점이다. 즉, 회사 차원에서 보면 2가지가 다 필요하다는 사실에 대해서 공감하고 있다.

소프트웨어 업계는 아마도 가장 빠르게 기술이 발전해 나가는 분야 중의 하나일 것이다. 조금만 딴 생각에 빠져 있으면 하루가 멀다하고 새로운 것들이 마구 쏟아져 나온다. 사실 소프트웨어 개발은 가장 개발자로 살아남기 힘든 분야일지도 모른다. 하지만, 자세히 들여다보면 반드시 그렇지만도 않다는 것을 알 수 있다. 즉, 과거에도 개발하는 방식에서는 별 다른 차이점이 없었고, 현재도 그렇고, 미래에도 그러리라는 것들이 이미 존재한다. 물론 사용자의 요구가 달라지고, 개발되는 제품이나 개발 언어, 툴 등은 바뀔지라도 결과적으로 프로그래밍이라는 것은 사람과의 대화라는 관점에서는 동일하다. 개발은 항상 여러 사람의 협업으로 이루어지며, 사람이 의사소통하는 방법이 바뀌지 않는한 “좋은 소프트웨어를 만드는 방법론”은 변화가 없다(예를 들어, Unix를 개발한 철학이 아직도 유용하다).

새로운 개발 방법론이 등장하고 있지만, 완전히 새로운 방법은 아니다. 애자일(Agile) 방법론도 사실상 주기를 단축하게 하고, 여러 번의 반복적인 폭포수(water-fall) 모델과 크게 다르지는 않다. (이것에 대해서는 반론이 많겠지만, 개인적으로 그렇게 생각한다. 아마도 나 자신의 내공이 부족 한데서 나오는 편견일지도 모르지만, 구현해야하는 개별 항목들에 대해서는 폭포수 모델을 사용하고 있다.) 어쨌든 오래동안 업계에서 일했던 사람들의 경험의 깊이에서 주어지는 무게를 충분히 과제에서 항상 느낀다.

소프트웨어 아키텍처를 만들어내는 사람을 아키텍트(Architect)라고 부른다. 매트릭스라는 영화를 보면 아키텍트의 역할이 잠시 나온다. 즉, 전체 시스템을 조화롭게 설계하는 임무를 수행하는 사람이라고 보면 된다. 여기서, “조화롭다”는 의미는 어느 하나의 요구사항에 치우치지 않고, “전체 시스템을 안정적으로 구축할 수 있는 능력”이 있다는 것을 의미한다. 이것은 사실상 아주 중요한 능력이다. 즉, 시스템이란 한 부분이 지나치게 강조되면, 다른 부분은 약점이 되는 경우가 많은데, 안정적인 시스템을 만들기 위해서는 이를 조화롭게 해결해나가야 하기 때문이다. 성능을 추구하면 모듈성이나 확장성이 나빠질 수 있고, 지나치게 보안을 강조하면 실질적으로 시스템의 반응이 늦어질 수도 있다. 따라서, 우리는 아키텍트를 훌륭한 오케스트라의 지휘자로 비유해 볼 수도 있을 것이다.

소프트웨어 개발자로서의 궁극적인 목표는 그러한 아키텍트가 아닐까? 많은 실전 경험과 풍부한 지식, 거기에 걸맞은 비지니스(Business) 역량 및 대화 기술(Communication Skill)을 가진 사람이 아키텍트다. 하지만, 이러한 사람을 단순한 개발자로만 생각하기에는 뭔가 부족하다. 아마도 엔지니어가 가지는 고리타분함과 끈기, 고집 혹은 아집을 가지고 있다는 편견이 작용하고 있는지도 모른다.

# [ 신입사원 지원자들에게 ]

어느덧 신입사원이었던 시절이 20년(2017년 현재)이 훨씬 지났다. 신입으로 회사를 들어온다는 것 자체는 흥미롭고 두려운 일이다. 면접을 보기 위해서 회사로 갈 때, 왠지 자신과는 차원이 다른 사람들이 모여서 있는 것처럼 생각되어 쉽게 말을 붙이기도 힘들다. 소프트웨어 개발인력을 뽑는 사람들의 앞에서 자신이 얼마나 실력을 갖추고 있는지, 30분도 안 되는 시간에 이야기하려면 얼마나 많은 준비시간이 필요할까? 30분도 많다. 아니 그보다 더 적은 시간이 주어질지도 모른다. 현실적으로 몇 명의 사람이 면접관으로 들어가서 하루에 30여 명이 넘는 인력들에 일일이 질문을 해서, 그 사람의 총체적인 역량을 파악하는 것은 거의 불가능한 일에 가깝다. 하지만, 이런 것에도 노우하우(know-how)가 숨겨져 있음을 아는가? 어떤 질문을 던지고 나서, 면접자가 대답하는 수준을 보면 벌써 그 면접자가 합격인지 불합격인지를 대충은 파악이 된다(물론, 면접관의 역량도 한 몫을 하겠지만). 예를 들어, 실무를 경험해 본 인력에게는 쉬운 프로그래밍은 물어보지 않고, 했던 project가 무엇이며 자신이 어떤 부분에 기여했는지를 구체적으로 묻는다. 보통 이런 경우 경험 있는 사람이 선호되는 것은 어쩔 수 없다.

대체로 학부를 졸업한 면접자들은 실무에 약하다. 특히 소프트웨어를 전공했다고 하는 친구들이 대부분 프로그래밍 역량이 부족한 것은 사실이다. 요즘은 학교를 어떻게 다니는지는 모르지만, 교양처럼 전공을 듣고 나온 사람들도 많다. 마지막으로 프로그래밍 해 본 지가 거의 2년이 지났다는 친구도 있다. 그에 덧붙여서 프로그래밍을 쉽게 배울 수 있다고 믿는 친구들도 있다. "전 프로그래밍은 언제든 따라잡을 수 있다고 생각했기에, 좀 더 폭넓은 지식을 갖추기 위해서 이런 저런 것들을 더 공부했습니다."라고 이야기한다. 하지만, 정말 중요한 것은 소프트웨어를 전공했다면 프로그래밍은 당연히 잘해야 한다. 문제를 보고 어떻게 해결할 수 있는지를 직접 고민하고 구현 해야지만 제대로 공부한 것이지, 단순히 내용을 읽어서 머릿속에 기억하고 있다고 해서 끝이 아니다. 책의 각 장(Chapter)마다 있는 연습문제는 그냥 거기 빈 페이지를 채우기 위해서 있는 것이 아니라는 말이다.

면접자들의 특징은 자신이 잘 모르는 분야에 대한 질문을 받으면, 자신이 아는 최대한 유식한 단어로 그것을 설명하려고 애쓴다. 하지만, 어리석게도 함정에 빠지는 것은 대부분 그런 경우다. 면접관을 하는 사람들은 최소한 전공에 대해서는 면접자보다 더 많이 아는 경우가 대부분이며, 최신 기술은 학생보다 기업에서 일하는 사람들이 더 잘 알고 있다. 따라서, 대답에서 실수하거나, 혹은 어려운 용어가 나올 경우 그것을 되묻는 것으로 면접자를 난처하게 할 수 있기 때문이다. 쉽고 간결한 답변이 자신의 지식 정도를 더 확고히 표현하는 가장 좋은 방법이다. 만약 책이나 TV와 같은 데서 모르는 것이 나온다면, 그것의 정의를 찾고 자신이 생각한 바를 간략하게 정리하는 습관을 만들 필요가 있다. 스스로 정의해 보지 않으면 개념적인 것은 이해하기 힘들다.

신입 사원이 되기 위해서는 패기만이 필요한 것은 아니다. 기초적인 지식 없이 패기와 열정만으로 모든 것을 표현하려는 태도는 어리석게도 소프트웨어 관련 분야에서 일하고자 하는 사람에게는 어울리지 않는다. 차라리 영업이나 인사(물론, 이곳에서도 전문적인 지식은 필요하다. 여기서는 그냥 예를 들어서 그렇다는 것이지, 수준의 높고 낮음을 말하려는 의도는 아니다)를 지원하기 바란다. 전공 면접에 들어온 사람은 철저히 전공에 대한 지식을 평가받는 자리에 선 것이다. 따라서, 열정과 패기로 전공을 공부하지 않았던 지난 과거를 보상받으려는 생각은 버려야 할 것이다.

이제 다시 질문을 던진다. “당신은 신입 사원이 되기에 충분한 실력이 있다고 보이나요?” 만약 그렇지 않다면, 실무를 배울 수 있는 아르바이트라도 직접 현장에서 해보기 바란다. 물론 “프로페셔널 프로그래머”로서 일 할 수 있는 기본을 닦을 수 있는 그런 곳이면 충분하다. 어학연수로 돈 낭비하기 보다, 그냥 무더운 여름날 허름한 사무실이라도 찾아가서 열심히 자신의 코딩 실력을 갈고 닦기를 빈다.

# [ 지식의 발전 과정에 대한 단상 ]

어떤 것을 익혀야 좋을까? 회사에서 리포트를 쓰라는 말을 종종 듣는다. 즉, 현재 어떤 기술들이 유망하고 회사는 어떻게 하면 될 것인가를 미리 정리해본다는 것이다. 하지만, 실제로 개발자가 아는 시점과 개발이 완료되어 시장에 나오는 시점에는 상당 시간의 차이가 있으며, 듣고 뭔가를 하려고 생각하면 이미 늦는 것이 일반적이다. 벌써 남들은 저만큼 가서 그걸 기반으로 새로운 것을 만들고 있음이 틀림없다. 이렇듯 새로운 기술은 계속 등장하고 있고, 개발자들은 어떤 기술에 동참해야 할지 난감한 경우가 많다. 과연 난 어떻게 해야지 먹고 사는 데 지장이 없을까?

먼저 개인적인 생각을 조금 정리해보자면, 개인이 갖추고 있는 지식이란 오랜 시간의 배움과 경험으로 축적이 되며, 그러한 축적이 되새김을 통해서 깨달음에 도달하게 되는 순간에 진정한 자신의 것이 된다는 것이다. 마치 고등학교 수학 시간에 본 가우스 함수와 같이 일정 시간 동안의 꾸준한 학습과 경험이 있어야 하고, 어떤 특정한 시점이 되어야 새로운 단계(수준)로의 도약이 가능하다.

신입사원들을 지켜볼 때 대체로 두려워하는 경향을 많이 볼 수 있다. 즉, 자신만이 뒤쳐지고 있다거나, 혹은 남들은 잘하는데 자신은 왜 그런가 하는 등등의 생각이다. 길게 보면 아직 시작도 하지 않은 시점에 벌써 미래를 걱정하는 것이나 다름없다. 이러한 생각을 지속해서 할 때는 남모를 좌절을 하기도 한다. 하나의 지식이 자기 것이 되기 위해서는 상당한 시간의 투자가 필요하며, 어느 순간 깨닫게 되는 자각이 필요하다. 마치, 무협지에서 무공의 어떤 구결을 부지불식 간에 깨닫게 되듯이, 공력만 높다고 모든 것이 해결되는 것은 아니다.

모든 분야가 다 힘들지만, 우선 임베디드 시스템 분야에서 일하는 개발자의 경우 대략 3년 이상의 경험이 요구된다. 3년 정도는 해봐야지 어느 정도 이해를 하고 있다고 생각되고, 5년 정도가 지나야지 혼자서도 무엇인가를 할 수 있는 충분한 능력이 있다고 생각된다. 작은 회사에 가면 솔직히 이러한 기회를 가질 수 있는 여건이 열악하다. 이런 저런 급박하게 돌아가는 업무 속에 자기 생각을 정리도 하기 전에 다른 일에 투입되는 것이 일상 다반사다. 더욱이 심지어 경험이 있는 기존 사원들이 자신이 알고 있는 것을 숨겨 놓고 가르쳐 주지도 않는 경우도 많다.

솔직히 이야기 하자면, 그런 사람 밑에서 절대 일하지 말라고 조언하고 싶을 정도다. 진정한 자신의 몸 값을 높이는 길은 자신이 했던 것을 최대한 자세히 잘 정리해서 다른 사람들에게 알려주는 것이다. 그래야지 진정한 발전이 이뤄지지 혼자서 잘했다고 꼭꼭 숨겨서는 아무로 알아봐 주지 않는다. 개발자로 성공하고 싶다면, 끈기를 가지고 좋은 선배를 만나서 3년간 고생하기 바란다. 좋은 선배는 구별하기 쉽다. 자신이 한 일에 대한 것을 잘 알려주는 사람이 좋은 선배다(최소한 마인드(Mind)만큼은 좋은 선배일 것이 확실하다.).

절대 좌절하지 말자. 하루 1시간씩 10년을 한 분야에 투자하면, 어느 순간 자신이 최고의 전문가가 되어 있음을 알 수 있을 것이다(물론, 이를 위해서는 체계적인 배움의 시간이 필요하지만). 따라서, 좌절의 문제가 아니라 끈기의 문제다. 머리가 나빠서라고 표현하는 것은 마치 고등학교 때 선생님을 찾아온 학부형이 "우리 아이는 머리는 좋은데, 공부를 안 해서 못한다"는 말과 다른 게 없다. 천재적인 사람은 다른 분야에 이미 많이 있고, 실제로 필요한 사람은 손발을 움직여 일 할 사람들이다. 똑똑해서 일하는 것이 아니라, 열심히 해서 일하는 것이다.

자신이 모자란 분야는 다른 사람에게 물어보자. 남들이 물어도 충분히 잘 설명해 주도록 하자. 오랜 시간 동안 다른 사람들이 만들어 놓은 기반에서 모든 개발자는 일하는 것이기에, 이젠 충분히 다른 사람들에게도 베풀 줄 알아야 한다. 무엇을 베풀어야 할지 모른다면, 최소한 자기가 한 일에 대한 것을 꼼꼼히 문서로 만드는 일부터 시작하기 바란다. 적어도 그 문서는 작성자 본인보다 오랫동안 살아남을 것이기에 분명하기에.

주저리주저리 이야기했지만, 이제 다시 원래의 문제로 돌아가 보면 "무엇을 배워야 하나?"를 고민해보자. 새로운 기술은 하루가 멀다하고 등장하는데, 그럼 무엇을 배우고 알아야 할까? 이것에 대한 해답은 생각보다 쉽다. 서점에 가서 가장 많이 팔리는 책을 찾는 것보다, 가장 많이 주제가 되는 것을 찾아서 배우는 것이 좋다. 하지만, 알아야 할 것은 이미 서점에 책이 나와 있다는 것은 다른 사람들은 벌써 저만큼 가 있다는 것이다. 하지만, 어떡하겠는가? 궁한 사람이 우물을 파야 하고, 원하는 사람이 공부를 해야 한다는 것을.

남들보다 먼저 무엇인가를 하기 위해서는 분야를 잘 찾아야 한다. 외국에서 히트 치는 상품이 국내에서 항상 대박을 치는 것은 아니지만, IT 분야만큼은 그런 성향이 강하다. 선진국에서 어떤 것이 주목을 받고 있는지 항상 잘 관찰하고, 이에 따라 자신의 역량을 개발하는 계획을 만드는 것도 하나의 해법이 될 수 있을 것이다. 아직은 우리나라가 IT의 변방이기 때문이다. 그리고, 학습의 항상 “흉내내기”부터 시작한다.

# [ 초보 팀장의 팀원 관리 방법 ]

처음 팀을 맡았을 때 가장 힘든 것은 어떤 일들일까? 사실 모든 일은 처음이 가장 힘들다. 시작이 반이라는 말이 있듯이, 시작을 잘하면 나머지 문제들은 저절로 잘 될 것이다. 그럼 “어떻게 시작할 것인가”가 가장 중요한 일임을 짐작할 수 있을 것이다. 모든 시작에는 “말”이 있다. 태초의 세상을 만드신 창조주도 말로 시작했듯이, 팀이나 과제의 시작에도 말이 제일 먼저 나온다. 팀을 새롭게 맡은 팀장은 구성원들에게 “어떻게 팀을 이끌 것인지” 팀의 운영 방침을 하게 된다. 그렇다면, 어떤 말을 하는게 좋을까?

* 1. 앞으로 해야할 과제를 설명하고 목표를 제시한다.
  2. 팀을 어떻게 운영할 것인지 설명한다.
  3. 일정 계획이 있다면 어떤 일정으로 어떤 일을 할 것인지 미리 숙지시킨다. 일정 계획이 없다면, 일정을 만드는 일을 시작하자고 이야기 하면된다.
  4. 팀장이 되기 전에 개인적으로 그동안 했던 일에 대해 설명을 해준다.
  5. 팀원들 각자에게 그동안 했던 일에 대해서 간략하게 자기소개 형식으로 이야기하게 한다. 반드시 받아적는 모습을 보이고, 나중에 한 명씩 따로 불러서 어떤 일을 하고 싶은지 등의 희망 사항을 들어보는 것이 좋다.
  6. 자리 배치에 대한 이야기를 한다. 친한 사람을 모아 두아두거나 섞어도 좋지만, 일을 효율적으로 할 수 있도록 관련된 사람을 함께 두는 것도 좋은 방법일 것이다. 자리 배치는 중요한 일이기에 반드시 당사자의 확인을 거치는 것이 좋다.
  7. 출퇴근에 대한 제약은 두지 않는 것이 좋지만, 지나치게 튀는 팀원이 있으면 어떤 식으로든 다른 사람들과 맞춰서 일하도록 하는 게 좋다. “누군 고생하고 누구는 논다는 인상”을 주는 것은 좋지 않다. 다 같이 일하고 다 같이 쉰다는 생각을 심어 주도록 한다.
  8. 만약, 외부 인력이 배치되어 잠깐 같이 일하게 된다면, 그 인력도 똑같이 원래 있던 팀원처럼 대하도록 한다.
  9. 일일보고/주간보고/월간보고 등의 일정에 대한 것을 정하도록 한다. 어떤 형식으로 어떻게 할지를 정한다. 가능하면 없애는 것이 좋지만, 정기적인 회의는 필요하다.
  10. 될 수 있으면 예외를 만들지 않도록 한다. 예외가 늘어나면 편견이 생기게 되며, 편견이 생기면 조직은 깨진다. 그렇다고 예외가 없는 조직도 없다. 따라서, 예외를 규칙으로 만들지 않는 것이 좋다.

이처럼 다양한 이야기를 그냥 아무런 준비도 없이 그냥 한다면 팀장으로서 제대로 준비가 되어 있지 않다는 인상을 줄 수 있다. 미리 반드시 준비해서 하나씩 이야기하는 것이 보기에도 좋을 것이다.

초보 팀장으로 처음 사람을 통솔하는 역할은 그렇게 쉽지 않다. 잘 준비한 모습을 보이도록 하자. 사람 사이에는 항상 대화라는 것이 필요하다. 대화가 없어지면 아무리 친한 가족사이라도 깨지기 마련이다. 대화는 일방적인 전달이 아닌 주고받는 일이다. 그냥 자기 할 말만 하고 그냥 끝내면, 그건 대화가 아니라 단방향의 명령이나 지시일 뿐이다. 팀이 일 할 때는 반드시 대화가 가장 중요한 성공 요소다. 소프트웨어를 개발하든 하드웨어를 개발하든 상관없이 대화가 없으면, 그 만큼 오버헤드(Overhead)가 커지고, 과제의 지연은 당연한 일이 될 것이다.

회사에서 가장 힘든 일도 대화에 대한 것이다. 누군가 기대하는 것만큼의 수준을 만족하지 못할 때, 혹은 어떤 일에 대한 정보가 부족한 상황에서 일하다 보면, 서로 간의 오해는 항상 발생하게 된다. 오해로 인해서 서로에게 상처를 주고 상처받기 마련이다. 하지만, 중요한 것은 자신을 먼저 돌아보는 태도다. 남이 이렇게 해서 난 이렇게 했다가 아니라, 원래 정해진 원칙이 이렇기에 이렇게 했다고 이야기해야 한다. 당장 자신의 잘못을 남에게 돌리기는 쉽지만, 먼 훗날 그런 회피와 나태가 언제가 자신에게 날카로운 비수로 복수하는 날이 올 것이므로.

그렇다고 무한정 모든 것을 들어주어야 하는가? 그렇지는 않다. 회사에는 기본 규정(Rule)과 프로세스라는 것이 있으며, 그것을 통해서 일이 진행되는 것이다. 그것을 무시하고 무작정 진행해서 좋은 결과가 나왔다고 해서 기뻐할 일은 못 된다. 규정과 프로세스란 결과의 재생산이라는 측면에서 중요하기 때문이다. 따라서, 그것을 무시한다면 좋은 결과의 재생산은 보장받지 못하고, 그때 그때의 상황과 개인의 역량에 의존하게 된다. 회사의 입장에서는 어떻게 될지 모르는 일보다 확신이 서는 일을 좋아하기에, 규정과 프로세스를 항상 지키도록 요구하게 되는 것이다.

팀은 규정 프로세스로 관리해야 한다. 회사만이 아니라 개인도 예측할 수 있는 상황을 좋아하며, 예측을 벗어난 결과에 대해서는 놀라게 된다. 물론, 긍정적인 “놀람”이라면 좋을지도 모르지만, 일이란 항상 예측 가능한 방향으로 흐르는 것이 더 중요하다. 갑작스러운 변화는 불안한 마음을 불러일으킬 수 있으며, 예상하지 못한 행동이 발생할 가능성을 더 높이기 때문이다. 방침을 정했다면, 나머지 문제는 그것을 얼마나 일관되게 지키는가에 달려있다.

# [ 소프트웨어 개발자의 삶이란? ]

소프트웨어 개발자는 항상 근거를 명확히 해야 한다. 사람은 어떤 체계에서 특정 위치를 점유하고 살아간다. 학교에서는 선배와 후배가 있고, 직장에서는 직급으로 나누어진 체계가 있다. 항상 인간이라면 이러한 여러 가지 관계와 체계에서 살아간다. 소프트웨어 개발자도 예외는 아니며 업무를 지시하는 사람이 있고, 그 업무를 수행하는 사람도 있다.

직급으로 위에 있는 사람은 아래 사람에게 흔히 지시를 내린다. "어떤 일을 어떻게 언제까지 해서 보고하라." 라고 이야기한다. 사실 이렇게 자세히 이야기해 주는 사람도 드물다. 대충 말로 던지고 가거나, 혹은 그냥 끄적인 노트를 찢어서 준다. 메일로 업무를 지시하는 경우라면, 업무의 근거라도 남길 수 있으니 그나마 다행이다. 어쨌든 모든 작성되는 문서는 합의를 하는 것이 좋다. 중요한 문서는 반드시 합의가 있어야 한다. 일을 진행하는 데 있어서 가장 훌륭한 의사소통 도구는 직접 만나서 이야기하는 것이지만, 그것을 보완하는 문서의 작성은 필요 악이다.

솔직히 소프트웨어 개발자의 삶은 그렇게 행복해 보이지 않는 경우가 많다. 소프트웨어 개발만 전문으로 하는 회사가 아니라면, 대체로 하드웨어 개발을 담당했던 사람들이 부서장을 하는 경우가 많다. 아직 우리나라의 소프트웨어 개발 역사는 길지 않지만, 하드웨어 제품의 개발은 상대적으로 더 오래동안 해 왔기 때문이다. 그리고, 판매되는 제품에 대한 가장 큰 이윤을 만들어내는 것도 아직은 하드웨어라고 믿는 사람들이 많기 때문이다.

하드웨어 개발을 하던 분들 밑에서 일하는 소프트웨어 개발자들은 항상 쫓기는 기분이 든다. 특히 제품의 개발 막바지에 다다를수록 소프트웨어 문제들이 터져 나오고, 제품이 시장에 출시가 되고 나서도 사용자의 AS요청에 대한 끝없는 싸움을 해야 한다. 따라서, 소프트웨어 개발자의 삶이란 끝없는 개발과 AS사이에서 지쳐가기 마련이다. 하드웨어는 팔리기 시작하면 고치기 어렵다. 리콜(Recall)이란 돈 많은 회사에서나 가능하지, 돈 없는 회사는 그냥 망하고 만다. "계획 -> 분석 -> 설계 -> 구현 -> 테스트 -> 구현 -> 테스트 -> 다른 계획하는 중에 이전 것 수정 -> 다른 것 구현하는 중 이전 것 테스트 -> ... -> 한 제품의 AS는 종료 하지만, 남은 제품들을 계속 수정 테스트"라는 끝없는 cycle속에서 사는 것이 회사에서 소프트웨어를 개발하는 우리의 숙명이다.

소프트웨어 개발자의 삶은 그래도 행복한 편이다. 어찌 보면 우리는 아주 쉬운 일을 어렵다고 하고 있는지도 모른다. 책상 앞에서 늘어만 가는 뱃살을 부여잡고, 컴퓨터에 연결된 USB 선풍기의 바람을 쐬면서 일하는 소위 “정신 노동자”다. 공장에서 8시간 동안 서서 열심히 부품을 조립하고, 저녁 먹고 일당을 더 벌어보겠다고 2시간에서 4시간 야근을 하는 노동자보다 힘들다고 말할 수 있을까? 물론, 배운 것이 있고 소위 말해서 높은 수준의 정신적인 일을 한다는 이유를 내세울 수는 있겠으나, 기본적으로 육체적으로 힘든 일을 하는 사람들보다 어렵게 일한다고는 말할 수 없다. "당신들은 육체적으로는 힘들지만, 정신적으로는 편하지 않나?"라고 말하는 사람들을 볼 때마다 "그럼 너도 한 번 해 봐!"라고 말하고 싶어진다. 육체적으로 힘들면 정신적인 여유조차 없다. 소프트웨어 개발자의 삶은 그나마 행복한 편임을 알아야 한다.

나이가 들어가면 점차 현실감이 떨어진다. 기존에 알고 있던 지식이 낡은 것들이 되어버리고, 기존에 하던 일하는 방식에 대해서 변화를 요구하는 것들이 늘어난다. 소프트웨어 개발자는 더욱 그러하다. 서점에서 가장 많은 신간이 나오는 기술도서 코너가 소프트웨어 관련 분야다(물론, 정확히 확인해본 것은 아니고 느낌이 그렇다는 것이다. 어학 관련 도서랑 만화책도 비슷할 것 같기도 하지만). 3개월이 멀다고 새롭게 발표되는 제품이랑 온갖 새로운 플랫폼들에 대한 지식을 다 익힌다는 것은 거의 불가능에 가깝다. 관리자 급에 올라간 사람은 대부분 최신 기술을 신문이나 전시회에서 뭐가 있다는 정도만 아는 것도 다행이라 생각해야 한다.

소프트웨어 개발자로 일 할 때는 Visual Studio나 Eclipse, 각종 compiler 및 shell script 등등을 자유자재로 다루고, C에서 C++, C#, Java등등의 언어들을 능숙히 다룰 때가 있었는데, 불과 1년도 안 돼서 기억이 가물가물해진다. 쓰는 개발 툴이라고는 고작 Microsoft워드나 파워 포인트, 혹은 엑셀 등의 오피스가 유일하다(이걸 개발 툴이라고 불러야 하나?). 어쨌든, 시간은 많은 부분에 대해 손을 놓아야 한다는 사실을 입증한다. "나이는 숫자일 뿐이다." 하지만, 그 숫자가 잊혀 질정도가 되어야 정말 삶이 바뀔 수 있다.

소프트웨어 개발자의 삶은 아래서부터 위로 올라갈수록 점차 개발과는 동떨어진 세계로 바뀐다. 대부분 개발자는 이렇게 살아간다. 하는 일은 조금씩 추상화되어 결국에는 뜬구름 잡는 수준의 이야기를 하고, 가끔 그런 것들을 아래 사람에게 시켜서 회식 자리의 한 귀퉁이에서 씹히는 직장 상사가 되어간다. 자신에게 깍듯한 후배 사원들에게 알지 못하는 벽을 느끼고, 대충 1차만 마치면 빨리 자리를 떠나주는 것이 남들에게 민폐를 끼치지 않는 지름길임을 알고 있다. 어려운 이야기를 쉽게 풀어 쓸려고 노력하고, 발표자료에서는 조사와 각종 문장부호는 사라져간다. 문단에서 필요한 핵심 용어 2~3개만 열심히 이야기하려고 노력하고, 어떻게 하면 비유적으로 설명할 수 있을지 고민하게 된다. 결국에 작성된 자료에는 남아 있는 단어가 몇 개 안되서, 주말에 나와서 몇 자 더 끄적여 본다.

소프트웨어 개발자의 궁극적인 삶의 목표는 “여유와 낭만”이다. 가끔 옆에서 일하는 사람들의 책상을 보라. 특히, 소프트웨어 개발일을 하고 있는 개발자들의 책상위에는 나름대로 어떤 테마가 보인다. 어떤 사람들은 에세이나 소설을, 어떤 사람은 애니메이션이나 게임에서 등장하는 캐릭터를, 영화 포스터로 도배된 칸막이를 볼 수 있을 것이다(이것 역시 소프트웨어 개발자만 그렇다고는 할 수 없겠지만). 자신이 아직 충분히 즐기지 못하고 있는 것들에 대한 동경(혹은, 바램)일지도 모른다. 간혹 그런 것들이 놓여 있는 책상을 못마땅하게 생각해서, "사장님이 방문하시니 책상 정리합시다."라고 하면서 치우라고 하는 사람들도 있지만, 그래도 “비가 오고나면 어느새 새로운 풀들이 자라듯” 하나씩 원래 있던 자리로 찾아간다.

창조적인 일을 하고 싶어 하지만 반복적이고 단순한 일을 할 때가 많다. 창조적인 생각은 여유에서 온다. 옛날에 철학이나 수학과 같은 분야에서 많은 업적을 남긴 사람 중에서 힘든 육체적인 노동을 하던 사람들이 있었던가? 아직 찾아보지는 못했지만 아마도 없는 것 같다. 즉, 어떤 일에 대한 창의적인 발상을 하기 위해서는 사람에게 여유가 필요하다. 특허를 쓰라고 이야기 하면서 언제까지 몇 건이라고 정하는 것이 무슨 의미가 있을까? 그때까지 만들어낼 수 있는 특허가 지금은 당장 못 만들어 낼 “특별한 이유”가 있을까? 구현해서 확인할 어떤 시간이 필요한 특허도 있겠지만, 시간이 흐르면 자연스럽게 생겨나는 특허가 있다면, 그냥 그때까지 기다리면 되는데 뭐 때문에 고민하겠는가? 소프트웨어 개발자에게 특허를 아무리 이야기해도 지금 당장 코앞에 놓인 버그와 개발일정이 더 중요하다. 마음의 여유라고는 눈곱만큼도 없이 것이 현실이다.

소프트웨어 개발자의 생산성은 "Dn x Ds x PCn x PCp(단, 여기서 n은 1보다는 항상 크며, 3이하이다. p는 시간에 따라 증가하는 특성이 있다.)"에 비례한다. 여기서 “Dn”은 사용하는 모니터의 개수, “Ds”는 그 크기(해상도)를 말하며, “PCn”은 사용하는 PC의 수를, “PCp”는 그 PC의 성능을 말한다. 소프트웨어 개발의 대부분은 “Copy와 Paste”이며, 여러 개의 창을 띄워놓고 동시에 작업한다. 하지만, 대부분 회사에서는 책상의 크기를 줄이기 위해서 모니터를 하나 이상 올려놓기 어렵고, 자리도 너무 좁아서 옆에 있는 사람의 물건들이 가끔은 내 자리로 넘어와 있다. 의자에 자세를 꼿꼿이 세우고 앞을 바라보면, 가끔 평소에 별로 이야기도 하지 않는 타부서 사람과 눈이 마주쳐서 어색한 눈인사를 주고받는다. 현실은 미국이나 독일의 소프트웨어 개발자와 절대 같은 환경을 요구할 수 없지만, 생산성은 그들 소위 선진국에서 일하는 개발자보다 더 많이 달성해야 한다(그것도 더 짧은 시간에). 피곤하지만 어쩌겠는가? 책은 될 수 있으면 관물함에 넣어서 가끔만 찾아보고, 낮은 성능의 PC지만 오래 걸리는 일을 시킨 후에 커피 한잔 할 수 있는 여유를 주는 것으로 만족하자.

소프트웨어 개발자의 삶에 대해서 약간 비관적인 견해를 이야기했다고 해서, 미래에도 그럴 것이라는 섣불리 예측하지는 말자. 다행히 요즘 학생들은 공학 쪽을 별로 좋아하지 않는다고 한다. 덕분에 우리 개발자들은 좀 더 오래 회사 생활을 연장할 기회를 가질지도 모른다. 점차 줄어드는 인력으로 몸값도 올라갈지 모르지만, 인도나 중국 덕분에 그러기는 힘들 것으로 보인다. 어쨌든, 그래도 누군가에게 일을 시킬 때 관리자는 필요하며, 그런 사람으로 남을 수 있기에 매월 집으로 가져가는 “가상 화폐(계좌에만 존재하는 돈)는 꾸준히 와이프의 통장에 꽂아줄 수 있을 것이다. 조금은 마음의 여유를 가지고 유머라는 것을 항상 마음 속에 간직한다면, 현실에서의 여유는 찾지 못하겠지만, 자신 만의 정신적인 여유는 누구라도 가질 수 있을 것이다.

# [ 잘 안되는 회사는 어떤 회사일까? ]

우리가 다니는 회사가 잘 돌아가는 회사인지, 아니면 하루하루 힘들게 돌아가는 회사인지 알 수 있는 방법은 없을까? 개인적인 의견이지만, 잘 안되는 회사는 아래와 같은 특징을 가진다고 보인다.

* 1. 컨설팅 업체나 진단팀이 들락거린다.
  2. 지시사항이 수도없이 메일로 날라온다.
  3. 보고서를 작성하는 양이 날이 갈수록 늘어난다.
  4. PL(Project Leader)이 자질구레한 일로 항상 바쁘다.
  5. 일일보고, 주간보고, 월간보고 등 회의가 늘어나고, 참석자도 많아지며 시간도 길어진다.
  6. 교육이 갑작스럽게 많아진다.
  7. 버그의 해결률(새로 발견된 버그/해결한 버그)이 시간이 흘러도 변함이 없다.
  8. 개발팀이 예전에 개발했던 과제로 인해서 신제품 개발과 A/S를 동시에 처리하고 있다.
  9. 각종 시스템이 갑작스럽게 구축되고, 일을 양으로 하려는 경향이 강해진다.
  10. 정작 그렇게 구축한 시스템은 사용하지 않고, 내부적으로 기존의 방법대로 일한다.

대략 이러한 것들이 “나쁜 냄새”가 조금 풍기기 시작하는 것들이다. 외적으로도 그렇지만, 자신이 느끼기에 "내가 왜 여기서 이러고 있을까?"하는 생각이 든다면, 이만 자리에서 일어나 새로운 공기를 마실 때가 된 것이다. 잠시 바깥으로 나가 머리를 식힐 필요가 있다.

# [ 개발자가 듣고 싶지않은 말들 ]

이 번에는 개발자에게 상처를 주는 말을 생각해 보자. 우리는 어떤 말을 들었을 때 가장 기분이 나쁘고 집에 가고 싶어질까? 톱10식으로 나열을 해도 되지만, 주관적인 이야기라 그냥 중요도가 아닌 생각나는 식으로 한 번 적어보도록 하겠다.

**01. 너 없어도 프로젝트에 아무런 지장 없어.**

이런 말을 듣는다면 이미 팀이랑은 인연이 없는 사람이다. 그냥 다른 일을 찾아보도록 하자. 정말 아무런 지장이 안 생기지는 않더라도, 이 정도 대화할 수 있는 상황이라면 당신이나 혹은 그 팀은 이미 존재 의미를 상실한 상태이다. 복수할 생각은 하지 말자. 어디를 가더라도 4.5촌 안에는 다 만나기(연결되어 있기) 마련이다.

**02. 이것밖에 못 해? 너 정말 실망이야.**

기대를 충족시키지 못했다는 말이니, 좀 더 열심히 하라는 뜻으로 들으면 된다. 물론, 이렇게 말하기 전까지 상황을 만든 것은 쌍방과실 일지도 모르지만, 제대로 보고를 못 한 잘못은 본인에게 있다. 열심히 하지 않았다면 그것도 본인이 책임을 지고 개선된 모습을 보여야 한다. 하지만, 제대로 일을 지시도 안 했으면서 나온 결과만 보고 이야기한다면, 팀장의 자질이 의심스러워지는 것도 사실이다.

**03. 이번 주말에 나와?**

주말에 나와서 일하고 싶은 사람은 회사를 통털어서 한 명도 없을 것이다. 있다면 아마도 가정에서 버림을 받은 사람이거나, 주말에 어색한 가족 모임이 있는 사람임이 틀림없다. 팀원들은 이 말을 들었을 때 갈등하게 된다. 주말에 나오라는 소리인가? 아니면 정말 나올지를 묻는 말일까? 일단은 이런 말을 들었을 때 가장 쉬운 답변은 "그 때까지 일 못 마치면 나와야죠." 혹은, "부장님은 나오세요?" 라고 되묻고 나서 나온다고 할 때, "그럼 주말에 뵙겠네요."라는 대답을 하는 것이다. 최소한 물귀신 작전이라도 펼쳐서 혼자만 나오지는 않게 만드는 것이다.

**04. 요즘 잔업시간이 많이 줄었네.**

잔업은 일이 남은 사람들이 하는 것이다. 만약, 일없이 남아서 있기를 원한다면 그건 회사를 이용한 자기 개발이거나, 혹은 러시아워(Rush Hour)를 피하려는 방법일 것이다. 잔업 하기 전에 자신의 업무량이 합당한지 부터 고민해 보라. 너무 없다면 부서장이 잘못한 것이고, 너무 많아도 부서장이 잘못한 것이다. 능력이 부족하다면 그것도 부서장이 팀원의 능력 이상의 일을 준 것이라고 치부하자(능력없는 팀원에게 역량개발을 할 시간도 주지 않았기에). 하지만, 회사에서 뭔가 발전을 이루어 보기를 원한다면, 주어진 일 이상으로 노력하기 바란다.

**05. 요즘 무슨 일 있어?**

개인의 사생활을 묻는 것은 좀 이상하니, 그냥 둘러서 해보는 소리다. 정말 무슨 일이 있기를 바라는 것은 아닐 것이니, 그냥 "아, 요즘 좀 힘드네요." 정도로 말을 시작하는 것도 좋을 것이다. 그리고 나서, 대꾸를 보고 어떻게 할 지 정하도록 하자. 사실 힘든 일은 일 자체가 아니라 사람 사이의 관계인 경우가 많다. 일은 시간과 노력을 들이면 해결되며, 실제로 우리가 하는 대부분의 일은 고도의 테크닉과 집중을 요구하는 일이 아닌 경우가 많다. 물론, 코딩은 집중력이 필요한 일이지만, 그렇다고 8시간 동안 내내 코딩만 하지는 않는다.

**06. 요즘 경기가 어려우니 경비를 줄이도록 합시다. 특히, 회식은...**

정말로 회사가 힘들다면 동참하는 것이 좋다. 그렇다고 하더라도 회식은 개인 돈을 쓰는 한이 있더라도 하는 것이 좋다. 물론 너무 과한 경우는 문제가 있지만, 적당한 수준의 방어전은 어쩔 수 없이 해야 한다. 제일 먼저 줄여야 할 경비는 뭘까? 아마도 잔업시간이 아닐까? 잔업 하기 보다는 주어진 근무 시간 내에 일을 마치고 집에 갈 생각부터 하는 것이 좋다. 업무에 대한 집중도를 높이고, 웬만해선 잔업 및 주말 특근을 하지 못하는 분위기 유도가 필요하다. 개발하는데 필요한 툴에 대한 투자를 줄이는 것은 빈곤의 악순환을 일으키는 주범이다. 툴은 시간을 줄여주는 도구인데, 그것을 줄이자는 말은 비싼 인건비를 더 쓰자는 말과 다를 것이 없다. 낭비라고 생각하지 않는 것들이 진정한 낭비가 되는 경우가 많다. 그런 것들을 줄이는 것이 진정한 경비절감의 길이다.

**07. 너는 이게 문제야.**

정말 그것만 문제일까? 사람은 완벽한 신이 아니다. 많은 문제를 가지고 살아가는 것이 오히려 인간답다. 코딩을 못 한다고 인간이 아닌 신의 존재가 되도록 강요를 받는다면, 그게 올바른 일일까? 오히려, "너는 이건 정말 잘해" 라고 이야기하는 것이 훨씬 효과적이다. 그리고나서, “근데 이번에 만든 이것은 이렇게 좀 고쳐봐” 정도가 좋을 것이다. 문제를 이야기하기 전에 잘한 부분을 먼저 찾아내는 것이 효과적인 대화 방법인 것이다. 같은 말이라도 전달하는 방법에 따라, 그것을 받아들이는 사람의 생각에 다른 영향을 주게된다.

**08. 여기 폰트가 잘 안 맞고, 조사가 어색합니다. 다시 수정해서 보고하세요.**

한 두 번 겪어본 일이 아닐 것이다. 매번 결재를 올릴 때나 보고서를 작성할 때마다 겪는 일이다. 당하는 사람 입장에서는 처음에는 좀 신경 쓰다가 나중에는 무감각해지는 말이다. 결국에는 “어차피 써봐야 다시 고쳐야 하니 빨리라도 쓰자”는 생각에 대충대충 적고 만다. 물론, 재작업으로 퇴근 시간이 늦어지고, 주말까지 나와서 글자 고치고 치장하는 일로 시간을 다 보낼 것이다. 차라리 팀에서 사용하는 템플릿을 배경이 간단하고, 폰트도 쓰는 것만 위주로 정해서 만들자. 그리고, 써달라고 하는 부분이 무엇인지를 명확히 해서 주고받는 게 훨씬 효과적일 것이다. 물론, 그것을 일일이 봐야 하는 사람은 고달프겠지만.

**09. 올해는 사업이 안 좋아서 보너스는 기대하지 않는 게...**

내년에도 좋아질 가능성은 반반이다. 올해 받지 않은 것을 내년에 두 배로 받는다고 누가 장담할 수 있겠는가? 집에 돌아가면 S전자는 얼마나 받는다고 말하는 와이프와 신문은 거들떠보고 싶지도 않아진다. 물론, 누구를 탓할 것은 아니지만 왠지 내 잘못인 것 같아 쪼그라든다. 삶이 쪼그라들고 생각이 쪼그라들고 내 능력이 작아 보인다. 어쩌겠는가? 삶이란 항상 자기보다 앞선 것은 쳐다보지만 뒤로는 절대 눈을 돌리지 않는다는 사실을.

**10. 다음 주부터는 권장 휴가 기간입니다. .... 근데, 월요일 중요한 회의가 있으니 참석 바랍니다. 혹은, 우리 회사는 자율 출근제를 시행하기로 했습니다. ... 하지만, 우리 팀은 8시까지 출근해서 10시까지 일하는 것으로 하겠습니다.**

어쩌라는 말인가? 휴가를 가란 말인가 가지 말란 말인가? 출근 시간을 알아서 하라는 말인가 하지 말란 말인가? 회사의 정책을 따르는 것이 아니라 팀의 정책을 따르는 것이 일단은 내년도 연봉에 유리하다. 하지만, 회사의 정책이 이런 식으로 무시된다면 더 큰 흐름에서 "한 마음 한 뜻"이라는 말은 그냥 구호에 그칠 뿐이다. 아예 그냥 "권장 휴무는 사용하지 말고, 출퇴근 시간은 정해져 있으니 맞춰서 나오세요."라고 말하라. 줬다가 뺏는 것이 더 나쁘다.

이상과 같이 개인적인 소감을 좀 적어보았다. 물론, 모든 것이 주관적인 것이므로 특정인을 대상으로 하는 것은 아님은 밝혀둔다. 이 글로 가슴에 찔림을 당하는 사람이 있다면, 찌른 손은 저자가 아님을 다시 한번 말하고 싶다. 개발자들은 그렇게 생각한다는 것이다. 물론 그 개발자에 저자도 당연히 포함한다.

# [ 소프트웨어 개발의 창의성에 대해 ]

흔히들 “소프트웨어 개발자는 창조적인 일을 하고 있다”는 선입관을 가지고 있다. 물론 소프트웨어 개발자는 창조적인 것을 만들어내고 있다. 하지만, 창조라는 것은 일부 몇 명의 사람들에 해당하고, 대부분의 개발자는 남이 만든 프로그램을 유지 보수하거나 재사용하는 일에 매달린다. 그렇다고 이런 일이 사소한 일이라는 것은 아니다. 다만, 우리가 정말 창조적인 일을 하는지 질문하고 싶은 것이다.

창조나 창작은 “여유”에서 나온다. 몸과 마음이 여유롭지 않은 사람에게서 창조적인 결과물이 나오는 것은, 버그 없는 완벽한 프로그램을 짜는 것만큼 어려운 일이다. 미안한 이야기지만 소프트웨어 개발자가 일하는 대부분 환경은 여유로움과는 동떨어져 있는 것이 현실이다. 사무실에 들어서면 낮은 높이의 가림막이 설치된 좁은 책상과, 옆에서 하는 작은 속삭임도 충분히 들릴만한 거리를 두고 동료들과 붙어서 일한다.

옆에는 테스트를 위한 수십 대의 “24시간 동작 중”이라고 표말이 붙어있는 기계들 옆에서, 더운 여름에도 에어컨이 제대로 나오지 않는 사무실에서 야근을 해야 한다. 일은 끝없이 늘어만 나고, 결코 줄어들 것 같지 않아 보이지 않는다. 여기저기 필요한 회의를 끌려다니다가 보면, 어느새 저녁을 먹고 다시 사무실로 올라가야 할 시간이 된다(물론, 모든 소프트웨어 개발 회사가 이런 환경인 것은 아니지만, 아마도 절반 정도는 되지 않을까?).

창조는 집중하지 못하는 공간에서 발생하는 인간 뇌 활동의 일정을 맞춘 결과물이 아니다. 창조는 인간의 뇌활동이 흐름(Flow)상태에서 집중할 때, 기존의 영역을 벗어나 새로운 영역과 결합되었을 때 발현되는 새로움이다. 집중되지도 않는 환경에서 집중되지도 않는 일을 하면서, 창의적인 결과물을 내놓으라고 하는 것은 무리한 일일뿐이다. 많은 수의 특허가 의무적으로 만들어지고 있는 상황에서는 좋은 특허가 나오지 않는 것도 같은 이유 때문이다.

“신입 사원들이 회사에서 처음 1년 동안 배우는 것이 무엇이냐에 따라, 향후 15년 동안의 개발자 인생이 결정된다”고 하면 억측일까? 한 가지 예를 들어보자. 부서에 들어온 "홍길동"이라는 사원의 선배가 제대로 된 문서도 없는 코드를 던져주고, 이를 분석한 후 VOC(Voice of Customer)를 처리하라고 이야기한다면, 그가 배울 수 있는 유일한 것은 선배들의 어깨 너머로 보는 것과, 잘 짰는지 그렇지 않은지도 모르는 동작하는 코드를 가지고 만지작거리는 것 밖에는 없다.

후배라고 해서 선배가 짠 코드가 잘됐는지 못됐는지 이야기할 수 없는 분위기에서는, 그냥 시키는 일만 그때 그때 조금씩 할 뿐이다. 문서를 쓰라고 하는 윗 사람의 지시는 있었지만, 자신의 선배가 그렇게 하고 있지 않다면 대충 필요한 부분만 조금씩 정리할 뿐이다. 부서의 업무 지침도 없는 경우에는 누군가의 행동을 따라하는 것만큼 확실한 방법을 찾지 못한다. 따라서, 객관화된 지식이 아니라 주관적인 생각에 의존해서 일하게 되는 것이다. 문서보다는 오히려 “손 가락에 의존하는 것”이다.

이런 상황이라면 1년 후의 "홍길동"의 삶은 갓 들어온 신입보다 뭐가 나아졌을까? 조금 더 많이 아는 코드, 조금 더 익숙한 개발환경, 내가 맡은 부분에 대한 기능 정도 밖에 남지 않을 것이다. 그 다음 신입사원이 와도 똑같은 시행착오를 반복할 것이며, 역시 1년 후의 모습은 아마도 크게 달리지지 않을 것이다. 뭔가 잘못되었다는 것이 아니라, 효율의 문제이며 성장의 문제이다. 효율이 없다는 것은 비용이 많이 든다는 것이고, 성장이 없다면 도태된다는 이야기다.

소프트웨어를 개발하는 방법은 아주 오랜 시간 천천히 바뀌고 있지만, 그 방법을 적용한 결과물의 형태는 아주 빨리 변한다. 그렇다면, 15년간 사용해야 할 부분을 익히는 게 먼저일까? 아니면, 그 결과물을 배우는 것이 먼저일까? 당연히 우리는 전자를 먼저 선택할 것이다. 하지만, 현실은 그렇게 쉽지 않다. 당장 다가오는 프로젝트 마감일을 맞추는 것이 더 급하고, 선배의 꾸지람을 피하는 것이 최선처럼 보인다. 정말 중요한 것이 뭔지를 안다면, 최소한의 노력은 어떻게 하면 좋은 소프트웨어를 만들 것인가를 고민하는 것이지, 그것을 모르고 구현에만 집중하는 것은 문제를 해결하는 근본적인 접근 방법이 아니다.

품질이 높은 소프트웨어를 만드는 방법은 이미 정해져 있고, 이를 통해서 만들어지는 결과물이 창조적이라 정리할 수 있다. 만드는 방법까지도 창조적으로 가져 간다면, 우리는 아주 값 비싼 대가를 언제고 치르게 될 것이다. 적어도 선배라면 그런 품질이 높은 소프트웨어를 만드는 방법을 신입들에게 가르쳐야 하며, 필요한 문서나 절차는 미리 정하고 일하는 것이 기본이다. 결국, 자신의 나태함에 대한 복수는 잦은 야근과 주말 근무를 낳을 것이며, 날아오는 VOC속에 함몰되어 어디서부터 잘못되었지만 고민만 하고 살 것이다.

# [ 업무에 관한한 거짓말은 하지 말라. ]

우리는 흔히 “선한 거짓말”을 한다. 선한 거짓말이란 상대방에게 불안이나 불편을 주지 않기 위해서 하는 것으로, 좋은 결과나 영향이 있을 것이라는 예상에서 하는 말이다. 물론, 때로는 나쁜 결과가 있을지라도 좋은 면을 알려주기 위해서도 한다. 하지만, 이것은 그 사람에 대한 충분한 애정을 가지고 있고, 감정에 대한 것에만 활용해야 한다.

우리는 개발 업무를 하는 데 있어서 “선한 거짓말”을 해야하는 상황을 종종 마주친다. 특히, 문제가 발생했을 때 “좋은 게 좋은 거”라는 생각으로 쉬쉬하며, 내부에서 적당히 처리하고 넘어가려고 한다. 실제로는 다른 이유로 과제가 지연되고 있지만, 마치 다른 사람의 잘못이나 상황의 논리를 통해서 과제의 일정을 연기시키려고 애쓴다. 과연 이러한 것들이 정말 선의의 거짓말일까? 의도는 개발에 부담을 덜고자 하는 좋은 의도지만, 사실은 다른 사람들에게 잘못된 정보를 주고 있는 것일 뿐이다.

과제를 진행하면서 우리가 하는 흔한 거짓말에는 무엇이 있을까? 대체로 과제의 시작과 함께 거짓말은 시작된다. 어쩌면 그보다 훨씬 전에 만들어지는 거짓말도 있을 것이다. 즉, 그 과제를 시작하기 위해서 필요한 거짓말도 있다. 그리고, 과제를 하는 도중에도 잘못된 정보를 주고 있으며, 과제가 힘든 상황에서도, 과제를 완료하는 시점에도 거짓말은 한다. 특별히 악의는 없을지라도 우리는 이미 거짓말에 익숙한지도 모른다.

**01. 과제 기획단계에서 하는 거짓말**

과제를 기획하는 단계에서는 예측에 대한 거짓말이 많다. 어쩔 수 없이 해야 하는 과제도 있고, 지난 과제에 대한 연장 선상에서 해야 하는 과제도 있다. 물론, 자신이 하고 싶어서 하는 과제도 있다. 예측이 정확한 근거를 가지지 않는다면 사실 전부 다 거짓이다. 예상되는 수익이나 판가, 재료비 등등, 이런 숫자들을 나열해서 최대한 자신이 하려는 과제가 의미가 있음을 증명해야 하는 것이다.

모든 것이 다 거짓말일 수는 없겠지만, 상당 부분은 근거가 없는 수치일 뿐이다. 특히, 요구사항에 대한 분석도 제대로 이루어지지 않은 상태에서 예상하는 비용은 다 거짓이다. 어떤 일을 하게 될지 모르는데 어떻게 인력 계획과 시간이 나올 수 있다는 말인가? 그냥 경험적으로 그럴 것이라고 거짓을 사실인 양 이야기하고 있을 뿐이다. 경험이 중요한 것은 사실이지만, 그렇다고 모든 과제를 해본 사람은 없다. 모든 소프트웨어 과제가 동일하지 않다면, 그 전까지의 경험은 참고할 만한 수준 정도에 불과하다.

**02. 과제 진행에서 하는 거짓말**

윗 사람은 이렇게 묻는다. "과제 잘 되어가고 있어?". 아랫 사람은 "네, 잘 되고 있습니다."라고 대답한다. 하지만, 정말 자신 있다면 과제를 하는 데 있어서 현재의 위험(Risk)이 무엇인지, 이슈(Issue)에는 어떤 것들이 있는지. 조직의 관리에는 어떤 부분이 어려운지 등등, 과제 전반에 대한 이야기를 시간이 있을 때 마다 해야 할 것이다. 좋은 게 좋은 것이라고 묻어두면 나중에 가서는 걷잡을 수 없는 상황이 온다.

직장 상사가 원하는 것은 과제에 대한 어려움이 없는지를 알고 싶어 하며, 그것을 해결해주는 것이 본인의 업이라고 생각한다. 그저 "네" 라고 대답에서 이러한 것들을 얻기는 어렵다. 물론, 맨 날 과제의 문제만 이야기하는 것도 쉬운 일은 아니다. 십중팔구 깨질 것이 분명하다. 상사는 자신이 하는 일이 아무 문제없이 잘 되기를 기대하기 때문이다. 자신은 능력이 있는 사람이기에 아랫 사람들도 잘 다루고 과제도 문제없이 잘 될 것이라고 스스로 믿기 때문이다. 믿음에 금가는 일은 누구도 하기 싫어할 것이다. 하지만, 조그마한 화를 크게 키우지 말고, 필요할 때 적극적으로 이야기해서 적절한 조언을 구하는 것이, 과제를 관리하는 입장에서 요구되는 최소한의 “양심”이다.

**03. 업무 보고에서 하는 거짓말**

업무보고는 작게는 자신의 팀, 좀 더 크게는 자신이 속한 그룹, 더 크게는 사업부, 혹은 회사 차원에서 진행된다. 거짓말의 강도는 점점 작게 시작해서 나중에는 크게 된다. 한 번 한 거짓말을 메우기 위해서 다시 부메랑이 되어 돌아오는 거대한 파도를 맞닥뜨릴 용기가 있는가? 그냥 나 죽었다 하고 욕만 들으면 된다고 생각한다면, 이미 당신은 좀비(Zombie)일뿐이다. 그리고, 아랫사람이 보기에 유능한 직장 상사가 아니라, 고정관념에 사로잡힌 관료일 따름이다.

누군가 자신을 비난할 때, 그 비난의 이유가 자신에게 있다면, 가장 쉬운 위기를 해결하는 방법은 그냥 인정하는 것이다. "잘 못 했습니다. 다음에는 이런 일이 없도록 하겠습니다."라고 솔직히 인정한다면 누구도 당신의 잘못에 대해서 더는 이야기를 하지 않을 것이다(물론 똑같은 일이 반복된다면 문제겠지만). 업무보고는 정확해야 한다. 의사결정을 하는 사람에게 정확한 정보를 전달하지 않는다면 잘못된 결정을 할 확률이 높다. 잘못된 결정은 회사 전체 혹은 과제 전체에 대해서 악영향을 주게 될 것이 분명하기 때문이다.

**04. 과제 완료 시의 거짓말**

과제가 생각보다 잘 완료되거나, 혹은 조금 지연은 있었지만 어쨌든 별문제없이 끝났다면, 완료된 과제에 대해서 어떤 식으로든 기록을 남긴다. 잘했던 부분에 대해서는 경험을 공유하는 차원에서 진행될 것이며, 잘 못 한 부분에 대해서는 개선해야 할 점이 어떤 것들인지를 알기 위해서라도 하게 된다. 하지만, 대부분 이때쯤 되면 과제 진행 시의 어려움이나 해결했어야 할 문제점 등은 밝히기를 꺼린다. 끄집어 내봤자 좋을 것이 없기 때문이다. 하지만, 정작 가장 중요한 것은 최적 사례(Best Practice) 혹은 최악 사례(Worst Practice)의 경험을 솔직하게 이야기할 수 있어야 한다.

모든 문제를 과제의 종료와 같이 덮어버린다면, 다음 번의 과제에서 그런 문제는 똑같이 재발할 것이다. 물론, 끄집어 내면 다음번 과제를 하는데 어려움을 겪을 수도 있을 것이다. 모든 비난의 화살이 내게 올 수도 있다. 하지만, “말에는 힘이 있다”. 진정한 마음을 담아서 이야기하는데 듣지 않을 사람은 없다. 드라마에서 가짜 연기를 통해서도 눈물을 짓는 것이 일반 사람인데, 현실에서 진실을 이야기하는 사람을 이해하지 못한다는 것이 말이 되겠는가? 어려움이 있다면 맞서야지 도망간다고 해결되는 것은 아니다.

앞에서 몇 가지의 흔한 소프트웨어 개발에서의 거짓말들을 이야기했다. 하지만, 아마 이 글을 읽는 분들은 자신만의 거짓말을 더 가지고 있을 것이다. 그러한 모든 거짓말을 다 나열하기에는 쓸데없이 긴 이야기가 되기에, 자신만의 것은 자신만의 비밀로 묻어두고 이제는 진실을 서서히 이야기하기 시작해야 할 것이다. 결국, 일이란 거짓없는 신뢰가 구축되어야 성공할 수 있기 때문이다.

# [ 리더의 조건 ]

대략 35세가 되면 프로그래머로서 어떤 일을 주도적으로 추진해 나갈 수 있는 나이가 된다. 몇 명의 후배들과 같이 일하게 되며, 어떤 경우에는 수십 명의 사람들과 함께 일해야 한다. 능력이 좋다면 리더(Leader)가 되어 과제를 추진할 수도 있을 것이다. 프로그래머로서 리더의 임무를 수행해야 하는 때가 왔다는 것이다. 솔직히 이런 부분이 두려운 사람들도 있을 것이다. 주로 혼자서 하는 일을 오랫동안 했던 사람일수록 대체로 이러한 경향이 많다. 또한, 과거에 경험을 축적하지 않은 사람들도 같은 상황에 직면하게 될 것이다. 하지만, 사실 후자의 경우에는 윗 사람이 적절히 리더의 역할에서 빼주기도 하겠지만, 어쩔 수 없이 리더가 되어야 하는 경우도 비일비재하다.

그럼 이러한 상황에서 프로그래머로서 리더 역할에 필요한 것들은 무엇일까? 사실 “이런 것들을 미리 알고 있었다면”하는 생각에 과거를 회상해 보기도 하겠지만, 시간이 흘러야지만 알 수 있는 것들도 있다.

**01. 자신감**

지나친 낙관은 프로젝트의 마무리 단계에서 비극적인 결과를 초래하기 쉽다. 잘 될 것이라는 막연한 태도는 위험(Risk)을 간과하는 실수로 작용할 가능성이 높다. 지나친 자신감은 지나친 비관과 다르지 않다. 하지만, 적절한 자신감은 후배들에게 프로젝트를 해도 된다는 안도감을 줄 수 있다. 자신감을 내 보이기 전에 극복해야 할 Risk에는 어떤 것들이 있는지 반드시 꼼꼼히 따져보기 바란다. 자신감은 구체적인 사실에 근거할 수 있어야 효과가 크다. 아무것도 없는 상황에서 가지는 자신감은 자만 밖에 되지 못한다.

**02. 팀워크**

팀워크는 아무리 강조해도 지나치지 않다. 한 프로젝트의 리더라면 어떻게 팀원을 서로 융합하게 만들 것인지 늘 고민해야 한다. 팀에는 다양한 사람들이 있다. 일을 잘하는 사람에서 못하는 사람, 성실한 사람도 있고, 게으른 사람도 있다. 누군가는 눈치를 보면서 일하기도 하고, 어떤 사람은 자신이 해야할 일을 능동적으로 찾아서 해결하기도 한다. 이러한 사람들이 뭉쳐서 팀워크가 이루어지는 것이지, 누군가가 잘못을 따지다 보면 남는 것은 일정 지연과 만족할 수 없는 결과물 밖에 없다. 그렇다면, 어떻게 팀워크를 만들 수 있을까? 한 가지 방법으로는 편견을 없애고 사람들을 바라보는 것이다. 어쩌면 가장 어려운 부분일지도 모르겠지만, 발언할 수 있는 기회를 동등하게 주는 것도 편견을 없애는 한가지 방법이다. 편견이 있다는 것을 상대방이 느끼는 순간, 대화의 단절에서 시작해 팀워크는 깨지기 될 것이다.

**03. 비전**

소프트웨어 개발자는 자신이 성장하는 일을 하고 싶어 한다(적어도 성장하고 있다는 것을 느끼고 싶어한다). 1년 정도는 괜찮다고 하더라도, 2년 3년이 지나가도 성장하고 있지 않다고 느끼면, 회사에 출근하는 것은 고통의 연속이 될 것이 분명하기 때문이다. 아무런 감흥도 없이 일상의 업무를 끝없이 처리해 나가기는 쉽지 않다. 눈 앞에 걸어 다니는 부서장이 자신의 15년 후의 모습이라고 생각하면, 귀감이 되지 못하는 선배도 문제가 있다. 결국, 자신의 비전을 가지지 못하면 남에게도 앞 날을 내다볼 수 없도록 만드는 것이다. 가장 먼저 자신의 비전을 만들고, 다시 팀의 비전과 융화시켜 나가야 한다. 스스로 흥이 나야지 즐겁게 일을 할 수 있다. 스스로 납득할 수 있어야 일하는 것도 편해진다.

**04. 사고**

리더로서 폭넓은 사고가 필요하다. 짧은 시각에서 잠깐의 성취를 맛보다가 시간이 지나서 무너지는 사례는 흔히 볼 수 있다. 전투보다는 전쟁에서 승리해야 하는 것이 경쟁이다. 사소한 것들에 일희일비하는 것은 결국 조직의 성과를 낮추는 것밖에 될 수 없다. 자신이 하는 일에 대한 비판을 들었을 때 가장 먼저 나타내는 것이 방어기재다. 즉, 자신이 해왔던 일에 대해서 누군가가 비판하게 되면, 자신의 잘못을 잊고 방어하기에 만 급급하게 된다. 심하면 남이 새로운 것을 제시할 때도 절대 받아들이지 않게 된다. 모든 일은 비판받을 수 있으며 더 나은 미래를 위해서는 그것이 정상이다. 비판이 없다면 발전도 없다.

**05. 평가**

정확한 평가는 현재에 대한 반성과 미래에 대한 해법을 제공한다. 능력에 맞는 평가가 주어지지 않고, 친근감이나 진급 여부에 따른 평가가 주어질 때, 사람들은 쉽게 낙담하게 되고 자신이 가진 능력을 보여주지 않게 된다. 내가 가지고 있는 것을 마음껏 발휘하면 좋은 평가를 받을 수 있을 것이라는 기대는 누구나 갖는 당연한 보상심리다. 그것이 만족되지 않는다면 누가 솔선수범해서 일하게 되겠는가? 어쩌면 당연하다고 생각할지도 모르지만, 본인의 경우를 되새겨보라. 과연 그렇게 하였는지, 혹은 그렇게 평가를 받았는지. “나는 나중에 그런 리더가 되지 말아야지”하고 다짐했지만, 어느새 자신도 모르게 그렇게 하는 모습을 발견한다면, 학습된 무의식적인 반응에 길들여졌기 때문이다.

이상에서 몇 가지 생각나는 것들을 적어보았다. 이외에도 리더라면 자신만이 가지고 있는 특별한 것들이 있을 것이다. 최소한 아래 사람을 마음으로 안아주면 마음으로 대답해 줄 것이다. 인정해 주면 그 이상의 성과를 내게 되며, 칭찬하면 자신을 더 높은 곳으로 안내해 줄 것이다. 더 높은 곳이 어디인지는 모르지만, 마음속에 그리는 바를 반드시 이룰 수 있을 것이다. 적어도 “사람은 남는 장사”를 하기 때문이다.

# [ 잔업에 대한 생각 ]

소프트웨어 개발자는 항상 주어진 일정과 경쟁한다. 위로 부터 지속적인 일정 단축을 요구받고, 사용자에게서 끝없는 VOC(Voice of Customer) 미구현과 버그에 대한 불평을 듣는다. 한정된 사람에, 한정된 예산에, 한정된 일정을 가지고 싸움을 치르다 보면, “내가 지금 뭐 하고 있나?”라는 자조적인 물음을 자신에게 던지게 된다. “내가 뭐가 문제가 있어서 이러고 있지? 다른 곳에 가면 더 좋은 환경에 연봉도 더 많이 받을 수 있는데, 내가 뭐가 못나서 이렇게 고생하고 있을까?” 모든 것이 문제처럼 보인다면, 이미 과제는 여러 명의 뱃사공으로 인해 이런 저런 산들을 다 가본 상황일 것이다.

관리자가 가장 쉽게 선택할 수 있는 해결 방법은 “일하는 시간을 늘리는 것”이다. 즉, 오늘은 10시간을, 내일은 12시간을, 다음 주는 하루에 14시간을 일하면서 문제가 다 없어질 때까지 "월화수목금금금"으로 일할 것을 요구하는 것이다. 왜냐고? 항상 관리자는 관리되는 사람들에 대해서 배고픔을 느끼기 때문이다. 자신의 경험상 하위에 있는 개발자보다 더 많은 정보를 가지고 움직이기 때문에 이런 현상이 발생한다. 더 많이 알고 있기에 더 많이 요구하게 되고, 더 모르기 때문에 왜 일해야 하는지 의문을 가지는 것이다.

정작 관리자는 과제의 중요한 실제 내용에 대해서는 잘 모르고, 표면상으로 드러나는 많은 수치에 만 민감하게 반응한다. 마치 그것이 과제에 대한 모든 것을 설명할 수 있는 듯 생각하는 것 같다. 또한, 관리자는 문제가 많은 과제 일수록 더 많은 숙제를 주는 경향이 있으며, 잘 돌아가고 있다고 보고된 과제에 대해서는 별 다른 숙제를 주지 않고 축하하는 분위기를 만든다.

역으로 생각해보면, 잘 안되는 과제가 숙제까지 늘어나면 제대로 일에 집중할 수 있을까? 잘되고 있는 과제는 정말 잘되고 있는 것일까? 혹시, 나중에 모든 문제를 한번에 터뜨리려고 하는 것은 아닐까? 적어도 과제 관리자라면 잘 안되는 과제에 대해서는 어떤 도움을 줄 것인지 생각해야 하고, 잘되고 있는 과제는 정말 그런지 정밀하게 살피는 것이 기본이다.

대부분 야근을 상습적으로 하는 조직에서는 오전 시간을 쉽게 낭비하는 경향이 있다. 저녁에 늦게까지 회사에서 일하면, 집에서 제대로 된 휴식을 할 수 없다. 당연히 다음 날 출근은 늦어지게 되고, 흐리멍텅한 정신을 바로 세우기 위해서 커피도 몇 잔 마셔야 한다. 따라서, 오전 시간의 업무 집중도는 휴식 부족에서 오는 피로감으로 낮을 수밖에 없다. 정작 가장 정신적으로 활발해야 할 시간이 오전이지만, 제대로 일에 집중하기 어려운 시간이 되고 만다.

금요일까지 일을 마무리하고 주말에 쉬려는 사람들은 최선을 다해서 자신의 시간을 효율적으로 운영하려고 한다. 토요일에 나와야 한다는 의무감에 사로잡히면 금요일 해야 할 일은 토요일로 미뤄지는 경향이 있다. 왜 그럴까? 어차피 회사에 나와야 하고, 토요일 출근하면 특근 수당도 받을 수 있기 때문이다. 물론, 모든 개발자가 그런 것은 아니다. 가족과 함께하는 시간이 소중하지 않다고 생각하는 사람도 없다. 하지만, 그렇다고해서 회사에서 해야할 일을 남겨두고 주말에 집에서 편하게 쉴 수 없는 것도 현실이다. 토요일 출근은 의무가 아닌 “강요되지 않는 선택”일뿐이다.

물론, 똑똑한 관리자는 사람들에게 잔업에 대한 이야기나 주말 출근에 대한 이야기를 직접 하지 않는다. 하지만, 은연 중에 살짝 압박을 가하는 것은 마찬가지다. 어떤 과제 관리자도 자신의 과제가 지연되서 끝나기를 원하지는 않기 때문이다. 이런 일이 자주 반복되면 나중에는 윗 분들이 하라고 한다는 핑계를 댈 것이고, 과제의 버그 개수를 가지고 이야기할 수도 있을 것이다.

정작 중요한 사람은 못 고치고, 문제만 고치는 우를 범하게 될 가능성이 높다. 소프트웨어 개발은 기계의 문제가 아니라 사람의 문제다. 사람이 자신의 정신력을 발휘해서 결과물을 만들어 내기에, 사람에 대한 이해가 가장 큰 부분이다. 사람이 지치고 피곤함을 느끼면 코드의 품질도 나빠지게 되고, 결과적으로 회사에 대한 신뢰도 낮아지게 된다.

과제를 관리하는 사람은 개발자들에 대해서 최대한의 결과가 나올 수 있는 환경을 조성해야 한다. 개발자의 시간을 헛되이 낭비하는 일에 몰두해서, 각종 수치를 이용해 집요하게 괴롭히는 것은 과제의 성공에 별다른 영향을 주지 못한다. 상황만 악화시킬 뿐이다. 필요없는 회의와 문서들에 대한 짜증은 평소에는 그럭저럭 넘어 가더라도, 일이 바쁠 때 날아오는 요청은 사람을 당황하게 만든다. 의도하지 않게 엉뚱한 방향으로 내용이 흘러 가도록 할 수도 있다.

문제는 개인의 집중력 있는 과제에 대한 몰입이지, “몇 시간을 일했으며(얼마나 오래동안 일했으며), 몇 개의 남아있는지”가 아니다. 그 몇 개의 버그를 고치다 보면, 결국 사람은 망가지고 지고한 정신 세계에 또 다른 버그를 심어놓는 우를 범하게 된다. 정신에 심어진 버그는 시한 폭탄으로 훗날 당신의 장래를 어둡게 하는 가장 치명적인 오류(Defect)가 될 수도 있다. 잔업이 많아지면, 쌓인 피로를 회복하는 시간도 더 길어진다. 사람의 정신 능력은 휴식없이 무제한으로 뽑아 쓸 수 있는 것이 아니다. “적당한 휴식”(?)을 즐길 수 있는 사람이 더 좋은 성과를 내는 것은 논리적로도 당연한 결과다.

# [ 오늘은 몇시간 일했나요? ]

자주 있는 일이지만, 관리자들은 일하는 사람들이 몇 시간 회사에 있었는가를 가지고 개인의 성과를 판단하는 경향이 있다. 글쎄, 오늘 10시간 동안 있으면서 이런 저런 일들로 소일을 한 사람과 최대한 집중해서 반나절 동안 일한 사람 중 누가 더 잘했는지 판단할 수 있을까? 솔직히 이것은 사람의 역량을 판단하기 어렵다는 문제를, 가장 쉽게 해결하기 위한 한 가지 편법일 뿐이다. 하지만, 정확하지도 못하고 모든 사람이 동의하지도 않는 방법이다.

오히려, 몇 건의 리뷰 요청(Review Request)을 처리했으며, 몇 라인의 코드가 정확히 테스트되었는지 보는 것이 더 효과적이다. 개발자의 생산성을 작성한 코드의 길이로 측정하는 것도 효과적이지 않다. 작성한 코드가 얼마나 테스트되었는지 보는 것이 더 많은 도움을 줄 것이다. 예를 들어, 100라인의 검증된 코드와 1,000라인의 검증되지 않은 코드가 있다면, 무엇을 선택할까? 아마도 확실히 검증된 100라인의 코드를 선택하지 않을까? (물론, 그렇다고 생산성이 검증된 코드와 비례한다고 생각하지는 않지만. 기술적인 난이도와 사용자에게 주는 영향까지 고려하면 그나마 조금 더 정확할 수 있을지도 모르지만.)

사람은 근본적으로 창의적인 동물이다. 주어진 일을 창의적으로 해결하고, 자신에게 주어진 모든 자원을 효율적으로 사용하려는 경향을 가진다. 사람의 창의성을 단순 무식한 시간과 만들어낸 코드의 라인 수를 가지고 판단하는 것 자체가 말도 안 되는 소리다. 대부분의 엔지니어는 순수하며, 정말 게으름을 피우지 않으며, 주어진 일은 주어진 시간에 마치기 위해서, 최선을 다하는 능동적인 사람들이다. 야근하고 잔업을 해서 목표량을 맞추는 것은 손에 잡히는 물건을 만드는 것에 한정되지(직접 육체적인 노동을 동반하기에), 고도의 정신 활동을 통해서 만들어 내야 하는 소프트웨어와 아무런 관련이 없다.

급하게 해야할 일이라면 단기적인 추가근무가 가능 하지만, 지속적으로 근무 시간을 늘리는 것은 낭비만 키우게 된다. 한정된 집중력이 전체 늘어난 근무 시간에 고르게 펼쳐져, 낮은 효율을 낼 것이 분명하기 때문이다. 또한, 급조된 것은 나중에 다시 정상적으로 수정하지 않으면, 오류로 남을 가능성도 높다. 하지만, 이미 동작하는 코드를 고치지 않는 것이 개발자의 일반적인 태도이기 때문에, 개선되지 않은체 지속적으로 과제에 재사용될 것이 분명하다. 몇시간을 일하는 것이 중요한 것이 아니라, 그 시간 동안 무엇을 했는가를 보는 것이 중요하다. 일을 했다고 한다면, “완료 조건(Definition of Done)”에 대한 것을 반드시 지켰는지 확인하는 것이 낭비를 막는 중요한 도구가 될 것이다.

# [ 상사의 꾸지람에 대처하는 방법 ]

회사는 위, 아래, 동료로 구성된 조직이다. 군대와 다른 점이 있다면, 시간이 흐른다고 일이 저절로 해결되지 않으며, 탈영한(출근하지 않는)다고 잡으러 오지도 않는다. 다만, 아쉬운 것이 있다면 처자식을 먹여 살릴 걱정은 반드시 해야 한다는 점이다. 아침부터 상사로 부터 꾸지람을 듣는 것은 별로 유쾌한 경험이 아니다. 특히, 어느 정도 한 분야의 전문가로서 일하는 사람이, 그 분야에 대해서 잘 모르는 직장 상사로 부터 듣는 자존심을 긁는 말은, 순간적으로 사표를 내고 싶은 충동을 일으키기에 충분하다. 이럴 때는 어떻게 대처하는 것이 좋을까?

1. **그냥 한 귀로 듣고 한 귀로 흘린다.**
2. **나 죽었다고 그냥 눈 깔고 아래만 바라다본다.**
3. **"예" 하면서 중간중간 말을 듣는 것처럼 고개를 끄덕인다.**
4. **자신의 솔직한 의견을 말한다.**
5. **“조치를 취하겠다”, “바로잡겠다”, “시정하겠다”, 혹은 “더 자세히 알아보겠다”는 말로 일단은 그 자리를 모면한다.**

이외에도 다양한 의견이 있겠지만, 이 글을 읽는 사람이라면 아마도 “왠지 모를 억울함” 같은 감정도 느낄 수 있다. 자기 딴에는 열심히 한다고 했는데, 그것을 몰라주는 상사에 대한 원망같은 것이다. 그렇지 않다면, 그런 상사가 없어서 행복한 사람들도 있을지 모른다. 상사라고 해봐야 특별히 다른 것도 아니다. 자신도 똑 같은 실수를 했을 것이고, 그것으로 꾸지람을 받은 경험도 있을 것이다. 다른 점이라고 한다면, 꾸지람을 주는 사람과 듣는 사람의 위치가 바뀌었을 뿐이며, 다른 곳에서는 당하는 입장에 설 수도 있다.

어차피 모든 일은 사람이 한다. 아무리 공군이 폭격을 하고, 해군이 상륙작전을 위해서 육지에 다가 가더라도, 결국에는 땅 위에 병사가 올라가야 점령할 수 있다. 마찬가지로 개발환경이나 모든 것이 거의 자동화가 되어 있다고 하더라도, 결국에는 그것을 운영하는 사람이나 사용하는 사람에게 모든 것이 달려있는 것이다. 누가 누구를 나무란다고 해서 좋아지는 것은 없으며, 그렇다고 남의 탓 만 하는 것도 올바른 선택은 아니다. 중요한 것은 다음에는 어떻게 할 것인가를 정하는 것이며, 더 나은 결과를 위해서 함께 노력해야 한다는 점이다.

전문가로 대접받고 있다면 적어도 4번을 선택해야 할 것이다. 솔직함으로 비록 욕을 먹을지언정 상사가 상황에 대한 파악은 정확히 하고 있어야 하기 때문이다. 그냥 모르는 척하고 넘어가기에는 상사들이 너무 불쌍하지 않은가? 직장 상사도 정확한 상황을 알고 싶어하며, 일이 잘 풀리지 않으면 걱정으로 심하게 스트레스를 받고 있을 것이다. 따라서, 그들에게 조금이라도 도움을 주기 위해서는 솔직히 그냥 있는 그대로 이야기할 필요가 있다. 투명성을 확보하는 것은 상대에 대한 신뢰를 회복하는 시작이며, 현실을 직시하고 정확한 대응법을 만들기 위해서 반드시 필요하기 때문이다.

# [ 그래서, 도대체 대책이 뭐야? ]

일일 보고나 주간보고를 할 때면 이런 이야기를 듣는 경우가 종종 있을 것이다. "그래서, 대책이 뭐야?", "도대체 어떻게 하겠다는 거야?" 등등, 사실 이런 말을 들었을 때, 우리는 너무나 뻔한 대답밖에 할 수 없다. "더 열심히 하겠습니다."라고, 하지만, 사실 이것은 대책이라기보다는 기다려 달라는 말 밖에 되지 않는다. 기다리지 못하겠다면 바꿀 수밖에 없다. 바꾼다는 말은 프로젝트를 취소(Drop)하던가, 아니면 과제 리더(PL: Project Leader)를 새로운 사람으로 앉히는 것, 그리고, 새로운 개발팀을 투입하는 수밖에 없다.

하지만, 새로운 사람이 들어온다고 바뀔까? 물론, 바뀔 수도 있을 것이다. 새로운 사람이 새로운 방법으로 효과적으로 일한다면, 그렇게 될 수도 있을 것이다. 하지만, 대부분의 경우 이미 가라앉기 시작한 배에 탑승할 선장이나 선원들, 혹은 새로운 손님은 없을 것이다. 가라앉지는 않았을 지라도 자신의 손을 더럽히고 싶어 하는 사람은 없다. 단지 마지못해서 시키는 일을 할 사람만 찾을 수 있을 뿐이다. 이런 식으로 사람이 투입되어 좋은 결과를 만들기는 더 어려울 것이다.

소프트웨어 개발자로 일하기가 어려운 것은 끝없는 문제 속에서 해결책을 항상 만들어 나가야 하기 때문이다. 개발자가 바뀌고, 요구가 바뀌고, 고객도 바뀌는 상황에서 끝없이 지쳐가는 것이 현실이다. 가슴 속에 담은 이야기를 꺼내기에는 듣는 귀가 얇아서 그냥 덮어두고 넘어가는 경우도 많고, 그런 말을 한다고 지금 하는 일이 바뀔 리도 만무하기 때문이다. 오늘 한 일을 내일의 숙제로 남기지 않기 위해서 그저 최선을 다할 뿐이다. 희망이라는 것은 이런 상황에서 생기지 않으며, 꿈이라는 것도 잊히기 마련이다. 이미 소프트웨어 개발은 4D(Difficult, Dirty, Dangerous, and Dreamless)직종으로 변했고, 그 늪에서 허우적거리고 있을 뿐이다.

그래도 "대책이 뭐야?"라고 묻는다면, 조용히 지금 적고 있는 글을 멈추고, 밖으로 나가 높은 늦가을 하늘의 해지는 모습이나 볼 것이다. 왜냐고? 해답이 없는 질문이기 때문이다. 문제를 제시한 사람이 문제라면, 문제를 풀 수 있는 사람은 이미 질문을 받는 사람이 아니다. 급한 일정을 만들고, 가능성이 없는 기능을 만들라고 한 사람들이 문제인데, 문제를 의뢰받은 사람이 해결 방법을 찾을 가능성은 없기 때문이다. 그래도 찾으라고 한다면, 그때는 당연히 요구할 것들을 제시할 것이다. 과제의 수를 줄이고, 기능을 줄이고, 야근 시간을 없애고, 더 많은 일할 공간을 확보하고, 프로세스를 지키면서 개발할 수 있도록 해 줄 것을 요구할 것이다. 대책은 처음부터 원래 그 자리에 있어야 할 것들을 하는 것이 최선이다.

# [ 행복이란? ]

소프트웨어 개발과 행복이라는 것이 무슨 관련이 있을까? 혹시, 이런 질문을 자신에게 했을지는 모르겠지만, "지금 난 행복한가?"라는 물음에 어떤 답을 떠올리는가? 행복이란 무엇이며, 그 행복을 위해서 지금 하고 있는 일이 도움이 될까? 사실 우리가 하는 모든 일은 궁극적으로 행복 하고자 하는 욕구 때문이다. 가족의 행복, 자기 일에 대한 만족, 부서원에 대한 배려와 근무 만족 등등이다. 어쨌든 그런 것들을 위해서 일한다. (물론 반드시 그렇지는 않겠지만, 그리고, 우리의 뇌리에 남은 교육의 효과인지도 모르겠지만.)

주말 근무는 무의식적인 나태를 낳는다. 금요일에 끝날 일을 토요일로 미루게 되고, 주중에 해야 할 숙제를 주말을 이용해서 하게 된다. 그럴 때면, 아이들과 와이프는 나를 제외한 다른 경험을 가지게 된다. 점차 경험을 달리하면서 발생하는 괴리감도 생기고, 집에 가면 그에 따른 소외감도 가지게 된다. 주중에 하는 야근은 아침, 점심, 저녁을 전부 회사에서 먹도록 만든다.

아내가 해주는 식사를 애들과 같이 오손도손 모여 앉아서 먹어본 지도 오래 되어갈 것이다. 사실 식사 시간에 얼굴을 마주보는게 전부인 생활에서 그 마저도 하지 못하게 되는 것이다. 시간은 흘러가면 잡을 수 없고, 애들은 지나간 시간에 대한 원망을 다 커서 하게 된다. 과연 내가 가는 길은 행복을 위한 것인가? 아니면 나만의 만족이나, 혹은 가정을 지킨다는 핑계일까?

월요일 아침부터 몰아세우는 상사의 지시를 받는 것은 즐거운 아침을 재앙으로 만든다. 주말까지 주어진 숙제에 대한 해결책을 찾지 못하고, 어두운 얼굴로 집으로 돌아가게 만든다. “일등이 되고난 다음에 이야히 하자”라는 말을 듣지만, 일등할 가능성이 전혀 없다는 느낌이 드는 이유는 뭘까? 세계에서 일등을 하고난 다음에 해야 할 일은 “일등을 유지하는 일”이다.

쫓아갈 때의 여유는 없고, 쫓기는 자의 긴장만 남게 된다. 당연히 더 열심히 일해야 한다는 의무감은 키우지만, 왜 그렇게 일해야 하는지에 대한 대답은 들을 수 없다. 휴가나 여유를 달라는 말은 그저 그냥 해보는 소리가 아니라 반드시 필요해서 하는 말이지만, 누군가의 귀에는 “놀고 싶다는 말”로만 들리는 것처럼 생각된다.

일등은 행복에서 오며 행복은 만족에서 온다. 만족이란 당연히 현재가지고 있는 것을 충분히 즐길 때 오는 것이다. 즐거움을 가지고 일하는 것은 의무감과 책임감에서 하는 일과는 다르다. 스포츠에서 이야기하듯, “열심히 하는 사람은 즐기면서 하는 사람을 절대 이길 수 없다.” 일등은 즐기면서 해야 찾아오는 결과이기에, 강요받는 방식으로 일한다면 영원히 도달할 수 없는 목표가 될 뿐이다.

즐겁게 일하기 위해서는 일하는 방법을 선택할 수 있어야 하며, 일해야 할 순간을 스스로 결정할 수 있어야 한다. 남에게 강요당해서 마지못해하는 것으로는 결코 만족에 도달할 수 없으며, 행복과는 멀어질 수밖에 없다. 일등이 모든 문제를 해결해 줄 수 있을 것 같이 보일지는 몰라도, 일등이 된다고 해서 만족한다는 것은 아니다. 부족한 것을 채울 수 있어야 일등이 될 수 있는 것이지, 부족한채로 일등이 되는 것은 아니기 때문이다.

끝없은 개발과 야근, 주말 특근의 연속이 행복을 과연 줄 수 있을까? 아마 절대 그렇지 못할 것이다. 소프트웨어 개발자가 강요와 압박에 의해서 결정된 일정을 맞힐 수 있는 확률은 극히 낮다. 만약 일정을 맞춘다면, 그건 일시적인 성공일 뿐이며, 연속적인 성공이 보장되는 것도 아니다.

목표에 도달한 이후에 오는 회의감으로, 지쳐있는 자신의 모습을 되돌아본다면, 목표한 일정 달성이 절대 성공일 수 없다. 그런 일에 대한 결과는 즉시 나타나지 않을지 모르지만, 결국 회사에 대한 좌절을 겪게 만들 것이고, 열정과 도전은 자신이 가질 수 없는 사치일 뿐이라는 것만 확인하게 될 것이다. 행복을 느끼지 않는 일을 하는 사람이 가질 수 있는 것이라고는 작은 보너스와 잠시 동안의 성공에 대한 도취감 밖에 없다.

실망하고 좌절하게 되며, 결국에는 무슨 이야기를 듣더라고 아무 느낌이 없게 된다. 야단을 자주 맞는 아이는 야단에 대해서 익숙해지게 되며, 나중에는 무감각하게 되어 더 어긋나기 마련이다. 한번 어긋난 아이를 가정으로 돌아오게 만드는 것은, 아마도 어긋나기 전보다 몇 배는 어려운 일이 될 것이다. 비난이나 야단으로 해결할 수 있는 것은 불만을 쏟아내는 사람의 감정 풀이 밖에 없다.

듣는 사람은 귀를 막게 만들고, 주변에 있는 사람들은 자신의 차례가 오기만을 기다릴 뿐이다. 그리고, 절대 문제를 만들거나 새로운 것을 시도하는 어리석은 짓은 하지 않을 것이다. 실패하면 비판받는 입장이 될 뿐이며, 성공해도 해야할 일만 커지기 때문이다. 더 많이 일한다고 더 많은 연봉을 주는 것도 아니며, 더 열심히 한다고 항상 성공만 할 수 있는 것도 아니기 때문이다. 이미 경험적으로 그런 것을 깨닫게 된다면, 일에서 찾아내는 행복이란 존재할 수 없다.

문제를 해결하기 위해서는 스스로의 가치를 확신시켜주고 믿음으로 지켜보도록 해야 한다. 문제가 있다고 몰아세우는 것 만이 해결책은 아니다. 진정한 해결책은 직접 그 문제를 해결해야 할 사람들에 대한 믿음이며, 그 믿음을 적극적으로 보여주는 행위 자체다. 혹시 부족한 것은 없는지 먼저 살펴야 하고, 걸림돌이 될 수 있는 것을 찾아서 사전에 해결하는 것이다. 그저 문제에 대해서 누구의 책임인지만 따지고, 비난만 하는 것으로는 아무 문제도 해결되지 않는다.

행복을 원한다면 주변에 있는 사람을 행복하게 만들어야 한다. 그 사람들이 믿고 의지할 수 있는 사람이 된다면 행복해지지 않을 이유가 없는 것이다. 자신이 좋은 리더라고 생각한다면, 주변에서 일하는 사람들이 행복한지 보기 바란다. 만약, 자신의 주변 사람들의 행복에 관심이 없다면, 절대 리더의 역할은 맡지 말기를 빈다. 불행은 혼자만의 것이지 남에게 줄 것은 절대 못되기 때문이다.

# [ 개발자의 마인드(Mind)? ]

누군가로부터 마인드가 부족하다는 말을 듣는다면 어떤 생각이 들까? 소프트웨어 개발자로서 일하면서 이런 말을 듣는다면 그렇게 기분 좋은 생각은 절대 안 들 것이다. 도대체 개발자의 마인드가 뭘까? 연구소에서 일하는 마인드와 사업부에서 일하는 개발자의 마인드가 뭐가 차이가 날까? 누가 옳고 그름의 문제인가 아니면 어느 것이 어느 것보다 더 좋다는 말인가? 소프트웨어 개발자로서 가져야 할 마인드는 누가 옳고 그름의 문제도 아니고, 어느 것이 어느 것보다 더 좋다는 것도 아니다. 다만 희망과 두려움의 차이가 있을 뿐이다.

소프트웨어 개발자의 기본 마인드는 “자신이 개발한 코드에 대한 확신”이다. 확신 없이 코딩하고, 다른 사람의 코드와 통합한 후, 테스트에 넘기고 집에 간다면, 어떤 일이 발생할까? 말할 필요도 없이 버그의 원인을 찾기 위해서 다른 사람의 손을 더럽히게 될 것이다. 덩달아서 자신의 손도 절대 편하지는 않게 될 것이다. 코드에 대한 확신을 위해서는 테스트가 반드시 선행되어야 한다. 어떤 테스트를 해야하는 지는 알지 못하겠지만, 적어도 자신이 개발한 기능에 대해서는 확신이 들 만큼의 테스트를 반드시 해야한다. 한 가지 추가적으로 덧붙인다면, 다른 사람의 코드에도 변경의 영향을 주지 않기 위해서 확인하는 일이 필요할지도 모른다.

야전(Field)과 학교의 차이는 뭘까? 야전에서 상품화에 매달려서 문제 해결에 몰두하면서 밤을 새우는 사람은 마인드가 제대로 되었고, 학교에서 공부하면서 더 나은 것이 무엇인지 찾아가는 사람은 잘못된 마인드를 가지고 있는 걸까? 야근을 많이 하면 제대로 일하는 것이고, 야근이 없이도 정해진 일정에 맞게 개발하고 있는 사람들은 마인드가 없는걸까? 정해진 프로세스를 지키라고 이야기 하면서, 일정을 단축해서 발생하는 프로세스 산출물의 낮은 품질에 대해서는 말하지 않는 마인드는 도데체 뭘까? 마찬가지로 이것도 누군가가 더 잘하고, 누군가가 더 못하는 문제가 아닌 서로 다름에서 발생하는 차이일 뿐이다.

학교에서는 분석을 통해서 현재를 설명하고, 미래에 사용될 수 있는 이론을 만든다. 야전은 현재 가지고 있는 것을 최대한 활용해서 경쟁력 있는 문제 해결책을 제시하는 일을 할 뿐이다. 따라서, 둘 다 문제를 해결하기 위해서는 서로를 필요로 할 수밖에 없다. 이론의 증명을 위해서는 사례를 찾아내서 증명해야 하며, 현실 문제에 대한 해결책을 찾기 위해서는 기존의 해결 방법을 정리한 이론을 적용할 수밖에 없다. 누가 누구보다 못한 것이 아니라, 서로 상호 협력하는 관계가 될 수밖에 없는 것이다.

학교에서 배운 지식이 무의미하다고 생각하는 사람은 없지만, 학교에서 배운 지식을 그대로 따라하는 사람도 없다. 문제는 학교 때 배운 지식을 무시하는 경우가 있다는 점이다. 이유는 항상 이론과 현실은 “다르다”고 가정하지만, 자신이 풀어야 할 문제가 다를 뿐 풀어나가는 방법 자체가 다른 것은 아니다. 논리적인 해결의 과정없이는 문제를 제대로 풀었다는 것을 보장할 수 없다. 단순히 결과가 잘 나온다고 해서 논리적인 과정(혹은, 정해진 절차)을 거치지 않는다면, 그 결과는 일시적인 현상이 되거나 재현할 방법을 찾지 못하게 된다. 학교에서 배운 지식이 필요한 이유는 해결 방법을 논리적인 과정을 통해서 증명하고, 동일한 수준의 결과물을 반복 재생산해야 하기 때문이다.

마인드의 차이가 있다고 말 할 때는 “품질을 확보하기 위해 어떤 노력을 하는지”를 봐야한다. 단순히 회사에 몇 시간을 남아 있느냐를 가지고 이야기 해서는 안된다. 같은 8시간을 일 하더라도 소프트웨어 개발자의 개인별 성과는 10배 이상 차이가 날 수 있다. 또한, 품질을 확보하기 위해서는 "코드의 가독성을 어떻게 확보하는지", "역할과 책임의 분리는 어떻게 하고 있는지", "코드 리뷰를 통해서 동료 개발자와 경험과 지식을 나누는지", "지속적인 통합을 통해서 항상 배포할 수 있는 코드 상태를 유지하는지", "변경에 대응하기 위한 구조를 설계 및 구현하고 있는지", "자동화된 테스트를 통해서 지속적으로 고품질을 유지하고 있는지” 등을 확인해야 한다. 마인드의 차이는 바로 이런 활동들을 개발자들이 제대로 하고 있는지를 보면 쉽게 찾아낼 수 있다.

# [ 성공의 기준은? ]

소프트웨어 개발자로서 성공했다고 말할 수 있는 기준은 뭘까? 이런 질문을 받으면 난감해하는 경우가 많을 것이다. 사실 나이가 40을 넘게 되면 코딩에서는 밀려나고, 관리자도 아니고 개발자도 아닌 사람이 되어간다. 결국에는 자신이 이끄는 개발팀에서 하는 기술적인 것들은 하나도 모르고, 숫자만 달달 외우고 살아가게 된다. 이런 삶이 과연 소프트웨어 개발자로서 성공을 의미하는 것일까? 그날 그날 버그가 몇 개고 무슨 문제가 있는지를 아는 것 만으로 만족해야 할까?

어디서 날아오는지도 모르는 문서를 작성하고, 줄지도 늘지도 않는 조직 문제 해결하고, 차기 제품에 대한 진행 상황을 파악하면서, 각종 회의에 참석해서 이런 저런 이슈를 보고하고, 현재 개발 중인 소프트웨어의 문제점이 얼마나 오픈(Open)되고 클로즈(Close)되는지를 보고하다 보면, 결국 코드는 한 줄도 보지 하루를 마감하게 된다. 그렇다고, 코드를 작성하자니 다른 개발자의 눈치를 보는 것도 쉽지 않다. 코드 리뷰라도 늦어지는 날이면, 어김없이 불평 가득한 독촉 메일이 받은 편지함에 쌓인다.

소프트웨어 개발자의 궁극적인 역할 모델은 아키텍트(Architect)가 되는 것이라고들 한다. 하지만, 모든 회사에 있는 40대가 넘는 사람들이 전부 소프트웨어 아키텍트는 아니다. 그 나이가 되어서도 여전히 개발하시는 분들도 있으며, 어떤 경우에는 전체 개발하는 코드의 1/3 이상을 맡고 있는 경우도 있다. 물론, 그렇다고1/3이상을 혼자서 코딩하는 것은 아니지만, 자신이 책임지고 있는 부분이 시스템의 핵심일 것이다. 중요한 일은 경험이 많은 사람에게 할당되며, 대부분의 경험은 코드에 녹아있을 가능성이 높기 때문이다. 여러 사람이 작성하더라도 근간이 되는 부분은 항상 오랜 경험자의 몫으로 남는다.

소프트웨어 아키텍트는 수요가 많은 분야는 아니다. 오히려, 그런 수요를 모르는 사람들이 더 많다. 있어야 할 필요성이 크지 않은 분야도 있고(소프트웨어 비중이 너무 작거나, 거의 변화가 없는 부분의 영역), 비지니스 도메인(Business Domain)의 이해를 더 많이 요구하는 분야도 있다. 아키텍트라면 자신이 풀려고 하는 문제가 존재하는 도메인을 이해 해야하고, 문제 도메인에 적용할 수 있는 기술도 잘 이해하고 있어야 하지만, 기술보다는 관리에 치중하는 경우도 흔히 볼 수 있다. 또한, 특정 제품을 개발해서 퇴직할 나이가 될 때까지 한 가지 제품에 몰입하는 경우도 있다. 하나의 제품 라인업에 한 명이나 두 명 정도의 아키텍트가 필요하지만, 아키텍트에 개발보다 관리를 요구하는 것이 일반적이다.

소프트웨어 아키텍트로서 일 할 수 있는 역할도 있고 갖추어야 할 지식도 있는 사람들은 특정한 도메인에 만 의존하지 않는다. 이런 저런 도메인을 옳겨 가면서 일하는 경우도 많다. 도메인은 달라지지만, 좋은 소프트웨어를 만드는 방법은 달라지지 않기 때문인지도 모른다. 달라지는 것은 풀어야 할 문제이지, 새로운 소프트웨어 개발 방법론을 만들어내는 것은 아니기 때문이다. 도메인에서 요구하는 성능과 기능이 달라질 뿐, 그것을 위해서 새로운 도구로 설계를 해야하는 것도 아니다. 따라서, 문제에 의존하기 보다는 문제를 풀어나가는 방법을 아는 것이 더 중요한 지식이라고 볼 수도 있다. 소트트웨어 아키텍트라면 적어도 어떻게 문제를 풀것인가를 명확히 제시할 수 있어야 할 것이다.

소프트웨어 개발자는 돈을 많이 버는 직업이라고 할 수 있을까? 물론, 돈을 많이 벌었다는 소프트웨어 개발자의 이야기는 종종 들을 수 있다. 하지만, 대부분의 개발자는 큰 돈보다는 적은 돈이라도 꾸준히 받을 수 있는 안정적인 직장을 다니길 원할 것이다. 잦은 야근과 주말 특근을 한다고 해서 큰 돈을 가족에게 가져다주는 것도 아니다. 오히려 가족과 함께 할 수 있는 시간의 기회 비용이 더 커보이는 경우도 있다. 물론, 이마저도 하지 않으려고 한다면 좋은 회사는 다니기가 힘들지도 모른다. 그나마 야근비라도 탈 수 있는 회사는 생각보다 큰 회사일 가능성이 높다. 대부분의 회사는 정당한 “초과 작업 수당”이라는 개념이 없으며, 단순히 교통비 정도만 지급할 뿐이다. 만약 일당의 1.5배를 초과 근무 수당으로 줘야 한다면, 대부분의 회사는 망한다는 이야기를 할 것이다. 따라서, 초과 근무를 하더라도 명목은 항상 교통비일 뿐이다.

소프트웨어 개발자는 상대적으로는 높은 연봉을 받지만 그렇다고 하는 일의 난이도에 비례한다고 볼 수는 없다. 정신적인 노동을 하는 것이 육체적인 노동보다 편하기는 하지만, 그렇다고 쉬운 일은 결코 아니다. 순수한 창작물을 항상 만드는 것은 아니지만, 언어의 조합으로 특정 요구를 만족하게 하는 결과물을 만들어야 하기 때문에 일종의 창작이라고 보는 견해도 있다. 고객 만족이 언제나 최고의 우선순위를 가지기 때문에, "고객이 만족할 때까지"라는 표어를 걸어두고 일하게 되지만, 그렇다고 개발자의 편의가 전혀 들어가지 않은 코드도 없다. 적당한 시간이 되면 출시는 반드시 해야하기 때문에, 문제를 발견하더라도 잠시 묻어두고 지나칠 수도 있다. 상대적으로 높은 연봉은 받지만 하는 일에 비해서 많다고 볼 수 없는 이유는, 창작과 육체 노동의 경계가 분명하지 않은 상태에서 책임감을 가지고 많은 결정을 내려야 하기 때문이다.

소프트웨어 개발자로서의 특별한 성공은 사실 없다고 봐야 한다. 다만, 자신에 대한 만족감이 그 자리를 대신 채워줄 수 있을 것이다. 시장에 출시된 자신이 만든 제품을 보는 것은 대단히 즐거운 경험을 제공한다. 오랜 시간의 노고에 대한 상으로 사용자들의 긍정적인 평가가 주어지면, 그 즐거움은 다른 무엇으로도 대체할 수 없다. 시장에서 일등 제품이라도 된다면 남몰래 즐거운 미소를 지으며 술자리에서 자랑스럽게 이야기할지도 모른다. 비록 자신의 이름이 적힌 제품은 아니더라도 함께 했던 사람들에 대한 기억은 어디를 가더라도 항상 즐거운 기억으로 남을 것이다. 그 기쁨에 다시 한번 젖기 위해서라면 오늘의 힘든 야근과 주말 특근도 즐거운 마음으로 선택할 수 있을 것이다. 따라서, 성공이란 누군가가 정해주는 것이 아니라, 자신이 스스로 만들어가는 것임은 절대 잊지 말아야 할 것이다.

# [ 좋은게 좋은거다? ]

모두 다 행복할 수 있는 사회? 정말 꿈같은 생각이다. 물론, 꿈 마저도 없으면 무슨 재미로 사냐는 이야기를 할 수 있지만, 라디오에서 나오는 캠페인성 광고를 듣는 입장에서 과연 그게 가능할까? 라는 의구심이 먼저 든다. 또 다른 광고로 "무슨 일을 하려고 고민이세요. 소설가 XXX는 몇살에 무엇을 썼습니다. 화가 XXX는 몇 살에 어쩌구..." 이런 말을 들었을 때, “그럼 그런 사람들이 소프트웨어 개발자로 일한 경우는 있었나? 그리고, 그 사람들이 한국 사람이었나?” 하는 자괴감과 같은 질문을 혼자서 운전 중에 하게 된다.

모두 다 행복할 수 있다면, 그건 이미 이 세상 사람들이 아닐 경우일 것이다. 냉정하게 말해서 일등보다 꼴찌가 축하받는 사회는 없다. 적어도 꼴찌를 한 이유가 그럴듯하지 않다면, 다만 그렇게 되라고 꿈꾸고 있을 뿐이다.

흔히들 "좋은 게 좋은 거다" 라고 하면서 넘어가는 것들이 많다. 심도있는 감사에서 나온 결과를 가지고 옥신각신하면서, 높으신 분에게 보고할 때는 나쁘게 보이는 내용들은 순화되고 없어진다. 결정 권한을 가진 사람에게 보고 될 자료는 정말 이렇게 쓰면 안 된다. 정확한 데이터는 뭉개지고, “대략 이런 문제들이 있는데, 현재는 이렇게 개선했고, 나중에는 어떻게 될 것이다.”라는 말로 그냥 얼버무리고 넘어간다.

결국에는 가장 고위급에 보고되는 문서에 실제 문제는 은근슬쩍 없어지고, 아무도 책임질 필요가 없는 것들로 만 가득차게 된다. 무엇을 듣고 어떻게 판단하라고 이렇게 하는지는 잘 모르겠지만, 결국 "좋은 게 좋은 거다."라는 말로 결론 짓게 된다. 물론, 그렇다고, 이 글을 쓰고 있는 사람이 특별히 도덕적으로 탁월한 사람은 아니다. 단지, 적어도 양심을 가지고 있는 엔지니어라면 그래서는 안된다는 말을 하는 것이다.

"좋지 않은 것은 절대 좋은 것이 아니다." 좋아지려면 어떻게 할 것인가를 이야기해야 한다. 좋지 않다는 이야기만 해서는 결론이 없다. 문제점을 이야기하기는 쉽지만, 해결책을 제시하기는 쉽지 않다. 문제점을 찾는 것은 몇 번의 질문으로 가능하지만, 문제를 해결하는 것은 개인이나 조직 수준에서 개선이 필요하다. 간단한 개선이 있을 수도 있고, 아주 오랜 시간의 “정착” 기간이 필요한 문제도 있을 수 있다.

무엇이든 "예전에 선배가 이렇게 했으니까"라는 것보다 "예전에 선배가 왜 이렇게 했을까?"라는 물음에서 접근하는 것이 원인과 결과를 명확히 볼 기회를 제공한다. 그냥 던져주는 사냥감을 받아서 먹기보다는 어떻게 사냥했는지를 아는 게 더 중요한 것이다. 우린 이런 말을 많이 들어왔지만 정작 제대로 하지는 못했다. 남들과의 관계와 이목 때문에 "좋은 게 좋은 거다."라는 식으로 넘어가는 문제만 없다면, 진정한 의미의 변화를 볼 수 있지 않을까? “좋은 것은 좋은 것이지만, 좋지 않은 것은 좋지 않을 뿐이다.”

# [ 해가 될 것인가 바람이 될 것인가? ]

과제를 하다 보면 많은 사람을 만나게 된다. 오랫동안 같이 일해왔던 사람도 있을 것이고, 처음 만나는 사람도 있을 것이다. 각각의 개성이 다르고 일하는 방식도 다르다. 사실 다르다는 것을 알지만 인정하기는 힘든 것도 사실이다. 개인별로 다른 성향을 가지고 있기에 그런 것들을 반영해서 사람을 대해야 하겠지만, 마음이 가는 사람이 있고 그렇지 못한 사람이 있다. 이들을 다 이끌고 가야 하는 것이 바로 부서의 장을 맡은 사람의 역할인 것이다.

과제의 실무적인 책임을 맡은 사람은 그 책임에 맞는 행동이 요구된다. 전체 과제에 대해서 잘 이해하고 있어야 하며, 과제의 방향이 잘못되거나 일정이 어긋날 경우, 조치를 취하는 일도 해야 한다. 각각의 팀원에게 적합한 책임과 역할을 할당해야 하고, 그것이 잘 지켜질 때는 칭찬을, 그렇지 못할 때는 충고와 질책도 주어야 한다. 책임감 없이 그저 시키는 대로만 일하는 사람은 과제의 책임자로서 부적절하다. 하지만, 어쩔 수 없이 책임을 지게 된다면, 그러한 부적절함도 어쩔 수 없이 극복해야 만한다. 지위가 역할을 만들듯, 역할에 걸맞은 사람도 만들어 주기 때문이다.

“길 가던 사람의 옷을 누가 먼저 벗길까?”를 가지고, 해와 바람이 내기를 하는 이야기는 들어봤을 것이다. 바람은 아무리 강해도 옷깃을 더 세게 부여잡게 만들지만, 해는 따스한 햇볕으로 길 가던 사람의 옷을 벗기게 된다. 팀을 책임진 관리자도 마찬가지가 아닐까? 팀원들은 항상 자신이 익숙한 범위에서 멈춰서려는 습성이 있다. 모든 사람이 아마도 자신의 틀을 깨고 싶어 하지는 않을 것이다. 익숙함에 젖어있는 사람들을 안정 상태에서 벗어나게 만드는 것은 결코 쉬운 일이 아니다. 깨뜨리려는 노력에 비례해서 반발도 더 심해질 것이다. 결국 그러한 노력이 일시적으로는 성공하는 듯이 보일지라도, 조금만 시간이 지나면 곧 예전의 그 자리로 용수철처럼 돌아가게 된다.

익숙함이란 시간에 비례하는 말로 그만큼 오래동안 유지해 왔다는 의미다. 따라서, 쉽게 다른 것으로 대체할 수 있는 것이라면, 절대 익숙한 것이 아닐 것이다. 이래도 좋고 저래도 좋은 것은 자신이 관심이 없거나, 잘 모르는 부분에만 해당한다. 정말 바꾸기 힘든 것은 바로 이처럼 습관처럼 아무렇지도 않게 사용하는 잘못된 관행들이다(여기서는 바꿔야 할 좋지 않은 습관이라는 의미로 사용했으나, 만약 좋은 관행이라고 생각한다면 변경할 필요는 없다). 익숙하다는 것이 편리할 수는 있지만, 그렇다고 옳바른 것이라는 보장할 수 없다.

결국, 문제는 스스로가 그것을 풀어헤치고 나오게 만드는 방법이다. 외부에서 줄 수 있는 것은 스스로 변화를 할 수 있도록 충분한 햇살(환경)을 주는 것이며, 질문을 던지고 인대를 가지고 기다리는 것밖에 없다. 성급하게 제촉하면 마찬가지로 반발심만 키우게 되며, 그렇다고 그냥 내버려두면 관행은 변하지 않는다. 따라서, 스스로 벗어나야 하겠다는 생각이 들도록 서서히 환경을 변화시켜 나가야 할 것이다. 문제를 주고 그 문제를 스스로 해결해서 더 나은 곳으로 가기 위해 노력하도록 만드는 것이다.

더 나은 것이 무엇인지 보여주고, 그것이 왜 중요한지도 알려주어야 한다. 물론, 문제를 직접 풀라는 것은 아니다. 팀원과 같이 문제를 해결하려는 태도가 시작이며, 정확한 정보 및 현 상황에 대한 공통된 인식을 만들어 나가야 할 것이다. 같은 문제를 같이 고민해야 해결책이 나온다. 변화를 현실화시키기 위해서는 행동도 필요하다. 행동은 스스로 목마름을 경험해야 실행으로 옮겨진다. 따라서, 적절한 불편을 경험시켜줄 필요도 있는 것이다. 미리 나서서 해결해 준다면 빠르고 남에게 아쉬운 부탁을 하지 않아서 좋지만, 결국 변하는 것은 대상이 아니라 나 자신이 될 수밖에 없다. 차라리 그냥 따스한 햇살을 비추며 충분히 기다려주는 것이 팀원을 움직여 땀 흘려 노력하게 만들어 줄 것이다.

# [ 개발인가? 관리인가? ]

나이가 들면 실무 개발에서 멀어지고, 관리 업무에 치중하는 것이 대한민국 소프트웨어 개발자의 현실이다. 소프트웨어 개발을 잘하건 못하건 상관없이 나이가 들면 관리 업무가 늘어나게 된다. 개발에 집중하더라도 관리 업무는 늘게 되고, 결국 개발에서 멀어지는 것은 어쩔 수 없는 현실이다. 개발할 것인가 아니면, 관리를 할 것인가의 결론은 이미 개발이 아닌 관리라는 것이 현재의 답이다. 하지만, 이게 옳은 답일까?

어떤 사람은 이런 말을 한다. “개발 만하던 사람이 처음 관리를 맡았을 때, 역할에 따르는 일을 제대로 감당하지못하고 힘들어하게 된다. 결국에는 자신을 자책하고 일에 집중하지 못하게 되며 직장을 떠나게 되는 경우를 종종 봤다”는 것이다. 하지만, 이것은 생각하기에 따라 다른 이야기다. 그 사람은 “관리만 하는 개발자”에게 관리 역량을 기대하였고, “개발도하는 관리자”는 개발 역량을 우선시했다는 것이 옳다. 누구나 관리가 좋은 것은 아니기 때문이다.

소프트웨어 개발의 관리업무는 소프트웨어 개발에 한정되어야 한다. 하지만, 현실은 온갖 회의와 어디에 사용되는지도 모르는 자료만 양산하게 되고, 많은 보고서를 쓴다고 허비하는 시간이 점차 늘어나게 된다. 결국에는 자신의 팀에서 개발하는 것이 무엇인지를 수치로만 보게 되며, 실제 코드에서 점점 멀어지게 된다. 장인을 원하는 회사에서 시각 장애인(결코, 장애인을 폄하할 의도는 없다. 여기서는 코드를 보아도 이해하지 못한다는 의미로 사용했다.)이 되어 버린 자신의 존재를 느낀다면, 이미 원래의 자리로 돌아가기에는 늦어다. 그냥 관성으로 달릴 수밖에 없다.

이미 이때쯤 되면 개발자로서의 삶은 포기하고, 회사에서 오래 남을 수 있는 방법만 생각하게 된다. 왜냐하면, 이제는 자신의 기술을 돋보이게 만들 수 없고, 자신의 관리 능력은 조직 속에 있을 때 만 보이기 때문이다. 다른 직장으로 옮기는 것은 지금의 고민보다 더 큰 스트레스로 작용하고, 나이에 맞는 자리를 찾기도 어렵다. 나이든 개발자 출신의 관리자에게는 선택이 아닌 생존의 문제가 되어 버린다. 하지만, 정말 행복한가를 묻는다면 어떤 대답이 나올 수 있을까? 자신이 지금 하고 있는 일이 행복한가? 자신이 최우선의 가치를 두고 있는 것들에 부합하는가? 아마도 자신있는 대답은 들을 수 없을 것이다.

하지만, 그런 것들을 따져야 한다. 그렇지 않다면, 그 자리에서 내려와야 할 것이다. 자신과는 어울리지 않는 자리에 엉덩이를 붙이고 앉아 있다고 해결되지 않는다. 시간을 더 가지고 그 자리에 다시 오를지 생각해 보라. 정말 아니라고 생각하게 된다면, 누군가의 말처럼 더 좋은 기회를 달라고 요구하는 편이 좋다. 행복하지 않은 일은 행복을 갉아먹는 벌레처럼 점차 자신의 기본을 병들게 할 것이다. 정신이 약해지고 집중도도 낮아질 것이며, 체력은 당연히 뒷받침보다는 부담으로 작용하게 될 것이다. 우리의 일이 행복이 아닌 생존 수단이 될 때, 일을 하는 사람은 즐거운 마음을 버리고 꿈을 더 이상 찾지 않게 된다.

# [ 갑자기 “집에 갔다”는 말을 듣는다는 것 ]

회사는 유기적인 동물과 비슷하다. 연일 사람이 중요하다는 이야기를 하지만, 어느 날 갑자기 보이던 사람이 보이지 않게 된다. 들려오는 소리는 실적 부진으로 퇴사를 했다는 말 뿐. 내게도 언제 찾아올지 모르는 별로 달갑지 않은 소식이다. 한 동안 정신이 멍한 상태로 일도 안되고, 그냥 그날 그날 있는 일을 처리할 뿐 다른 대책은 없다.

"오랫동안 있을 것처럼 일하고, 언제 떠날지 모른다는 태도로 준비하라"는 말이 있다. 내게도 찾아올지 모르는 불청객은 무시한다고 되는 것이 아니다. “든 사람은 모르지만, 난 사람의 자리는 보이는 것”이 인지상정이듯, 나름의 대비가 항상 필요하다.

어떤 것이 필요할까? 솔직히 과제를 진행하는 동안 팀원이 교체되는 경우도 빈번하다. 이런 경우 사람에 의존적인 조직을 만들면 안 된다는 것은 자명하다. 그 사람이 없으면 일이 안 된다는 것은 조직을 잘못 관리해 왔다는 말 밖에 안되며 변명의 여지가 없다. 따라서, 일이 시스템에 의존적인 방향으로 변경되어야 한다.

일하는 체계 및 방법 등이 시스템화 되어서 자동으로 움직여 나가고, 인적인 변화에도 충격이 약하도록 구축되어 있어야 한다. 후배 사원에 대해서는 항상 적절한 교육이 주어져야 하며, 팀원들 간에 고른 실력을 갖추도록 노력해야 한다. 나 자신이 될지도 모르지만 누군가를 위해서도 이런 것들은 항상 필요하다.

시스템에 의존적이라는 말은 무엇일까? 즉, 역사(History)가 시스템에서 관리가 된다는 것이다. 문서든 소스 코드건 모든 것이 시스템에서 관리가 되고, 언제나 영속적으로 존재할 수 있도록 만들어 주어야 한다. 다양한 도구(Tool)에 대한 사용도 공용화 표준화 해야하며, 테스트도 항상 자동화될 수 있는 방향으로 움직여 나가야 한다.

사람이 바뀌더라도 품질은 항상 일정 수준으로 맞출수 있는 것이 이상적으로 관리되고 있는 상황이다. 인간에 대한 고려가 가장 중요하지만, 사람이 바뀌는 상황도 항상 대비해야 한다. 특히, 작은 조직에서 특정 몇 몇에 의해서 성과가 좌우되는 조직은 아주 위험하다. 꾸준한 보상도 중요하지만, 그런 보상으로 만 사람이 관리되는 것도 아니기 때문이다.

소프트웨어 개발은 사람에 대한 관리가 대부분이다. 사람이 가장 중요하며, 좋은 사람은 그렇지 않은 사람보다 극적인 생산성 차이를 보일 수 있다. 하지만, 수치로 만 사람을 보지 말고, 그 사람이 있어서 다른 사람에게 긍정적인 영향을 주고 있는지를 봐야한다. 진정한 관리란 긍정의 영향력을 타인에게 전달해, 남을 통해서 성과를 거두는 것이다. 자신이 잘난 것이 아니라 주변 사람들의 노력으로 빛나 보이는 것뿐이다.

# [ 새로운 조직을 맡게 된다면 ]

익숙한 일을 하다가 갑작스러운 조직변경으로 새로운 일을 맡는 것은 생각보다 많은 스트레스를 준다. 자신이 하던 일이 완전히 바뀌면 누구나 당황하는 것은 마찬가지다. 더군나 중요한 일이 자신에게 느닷없이 주어진다면 어찌해야 할 바를 모르고, 이런 저런 생각으로 잠을 못 이루기도 한다. 어떻게 하는 것이 최선일까? 아니면, 어떻게 자신을 바꾸어야 새로운 조직에서 살아남을 수 있을까?

첫 번째 해야 할 일은 당연히 해당 업무의 파악이다. 새로운 조직의 업무가 무엇이고, 어느 범위에서 일을 해야 하는지 정해야 한다. 조직의 업무는 주로 내부에서 해야 할 일과 외부적으로 전달해야 하는 일, 그리고, 마지막으로 외부에서 주어지는 일이 있다. 외부적으로 들어오는 일은 언제 들어올지 알지 못하는 것과 정기적으로 해야 할 일이 있으므로, 정기적으로 해야 할 일은 잘 파악해야 한다. 또한, 부서원 각각이 무슨 일을 하고 있으며, 역량 수준은 어떻게 되는지도 알아야 한다. 업무를 파악하기 위해서는 일일이 사람을 만나서 면담하는 것이 제일 빠를 것이다. 같은 질문을 여러 사람에게 물어보는 것도 좋은 방법일 것이다.

외부에 전달해야 하는 일은 내부에서 진행한 업무의 결과다. 이때는 명백히 일의 범위를 정해서 해야 하고, 책임 소재를 확실히 하기 위해서 업무 분담을 어떻게 하는지 정해야 한다. 결과물에 대한 일차적인 검증은 당연히 내부에서 책임진다. 내부에서 진행되는 일은 그냥 두어도 진행될 것이다. 하지만, 내부적으로만 알려진 일과 외부에 드러난 일은 관리를 달리해야 한다. 어느 곳을 가더라도 모든 것을 다 공개(Open)하는 조직은 드물다. 자신이 설 때까지 내부에서 진행하는 업무는 그대로 진행하도록 하고, 외부적으로 드러난 일은 차질이 없다는 것을 보장해야 한다. 이 때 내부적으로 진행되는 일에는 가능한 최소의 인력을 배치하는 것이 좋을 것이다.

두 번째로 해야 할 일은 새로운 일을 만드는 것이 아니라, 필요 없는 낭비를 줄이는 것이다. 필요 없는 보고와 회의는 가능한 최소화시키고 실행에 집중한다. 만약, 새로운 것을 반드시 해야 한다면, 기존의 것 중에 중요도가 낮거나 무의미한 일을 나중으로 미루거나 없애야 한다. 같은 인력으로 추가적인 새로운 것을 한다는 것은 일종의 업무의 부하만 높일 뿐이다. 효율적으로 가볍게 움직일 수 있는 조직을 만들어야 외부적인 변화에 빠르게 대처할 수 있다. 일에는 시기가 있듯이, 조직은 변화를 빠르게 대처하는 방법을 찾아야 한다.

세 번째로 해야 할 일은 팀원들의 업무 혼선을 방지하는 것이다. 관리의 원칙을 세우고, 이를 규칙(혹은, 문화라고 표현하는 것이 적당할 듯)로 정립하는 것이다. 여기서 가장 중요한 것은 불필요한 중복이 없어야 하고, 개별적으로 독립을 유지할 수 있어야 한다는 점이다. 물론 예외를 두지 않는 것은 어렵다. 하지만, 예외를 만들지 않도록 노력하는 것이 중요하다. 업무 처리에 대한 “규율”이 명확한 조직이라면, 업무에 대한 책임과 보상도 명확히 주어지게 된다. 책임 만 줄 것이 아니라, 권한도 이양하는 것이 필요하다. 즉, 업무를 수행할 때 필요한 결정을 현업을 담당자에게 이양하는 것이 좋다. 모든 일을 세밀하게 관리하려고 하면, 정작 중요한 것은 잃어버릴 가능성이 높다. 관리자도 사람이기에 놓치는 부분이 생길 수밖에 없다.

아직도 현업에서는 하드웨어를 담당하던 사람들이 소프트웨어 개발 관리 업무를 맡는 경우가 많다. 이 때는, 위에서 말한 것들을 진행하기가 쉽지 않고, 개발자가 말하는 것이 진실인지 거짓인지도 구분하기 어렵다. 이 때는 명확한 수치와 결과물의 품질을 가지고 확인해야 한다. 과제의 마일스톤(Milestone)을 정하고, 각 마일스톤 별로 나와야 할 산출물에 대한 것들을 명확히 해야 한다. 마일스톤은 주요 기능의 구현 및 검증이 완료되는 시점에 데모(Demo)를 통해서 사용자 혹은 이해 관계자가 과제의 상태를 눈으로 확인할 수 있도록 정해야 한다. 사용자나 이해 관계자는 자신들이 요구하고 있는 것이 어떻게 구현되고 있는지 보기를 원하며, 개발자는 이를 만족시켜야 할 의무가 있기 때문이다.

# [ 잘듣는 것의 중요함 ]

회의를 하면 몇 가지 부류의 사람이 있다. 높은 사람일수록 말을 많이 하고, 자신의 의견을 관철해서 결론을 빨리 내려는 경향이 강하며, 성격이 급한 경우가 대부분이다. 하지만, 그런 사람이 카리스마 있고 능력 있는 사람으로 받아들여지는 것도 현실이다. 미리 스스로 결론을 내리고 들어와서 사람들의 의견을 듣는척 하면서 자신이 생각하고 있던 것을 관철시키고, 사람들에게 강제적인 다짐을 받아낸다. 아무런 의견을 내지 않고 그냥 앉아서 남들이 하는 말에도 관심을 보이지 않는 사람도 있고, 누구에게 말하는지도 모를 허공으로 던지는 말을 하고는 그냥 무관심한 사람도 있다.

중요한 것은 잘 듣는 것이다. “다른 사람의 이야기를 자의로 해석하지 않고, 있는 그대로 듣는 것”이다. 해석하지 않는다는 것은 주관을 배제한다는 것이다. 이건 정말 쉽지 않다. 특히, 자신이 관련된 일에 대해서 논의하는 회의는 객관적인 관점을 유지하는 것은 무척 어렵다. 하지만, 중요한 것은 잘 듣는 것이고, 그에 따라 물러나거나 더 강하게 나아가야 할 수도 있다. 자신만이 아닌 회의에 참석하는 모든 사람의 이익이 되는 결론을 얻으려고 노력해야 한다. 자신의 일이 덜어진다고 해서 좋은 결과를 낳는 것은 결코 아니다. 모두가 이익이 되어야 자신도 발전할 수 있다. 굽힐 줄 알아야지만 존중도 받을 수 있는 것이다.

보고에서는 결론을 먼저 이야기해야 한다. 그리고, 결론은 누가 들어도 쉽게 이해할 수 있어야 한다. 회의에는 특정 분야의 전문가만 모이는 경우도 있지만 항상 그런 것은 아니다. 그렇다고 회의에 참석하는 모든 사람들을 이해시킬 수도 없다. 따라서, 그들 모두를 이해시키는 것은 어려울지 몰라도, 회의에서 가장 높은 의사결정 권한을 가진 사람에게 정확히 의사가 전달될 수 있을 정도로 쉬워야 한다. 사실 아는 것과 설명하는 것은 전혀 다른 것을 요구한다. 아는 것은 아직 표현되지 않았고 정리되지 않았다는 것을 의미하지만, 설명하는 것은 모두가 공유하는 지식이 된다.

자신의 의견을 말할 때 가장 쉬운 방법은 그림으로 보여주는 것이다. 그림은 복잡해서는 안 되고, 결론을 정확히 이야기할 정도의 수준에서 정리되어야 한다. 그림으로 사고하는 사람은 자신이 주장하는 바를 정확히 남에게 전달할 가능성이 높다. 말하는 사람의 의도를 글로 설명하기 보다, 시각화하는 방법을 동원하는 것이 주의를 집중하게 만들 수 있기 때문이다. 말과 그림이 합쳐서 주제를 설명하는 방법이 의사 전달의 강력한 도구이기 때문이다.

관리자로서 혹은 개발자, 하위 직급자로서 자신의 지위에 상관없이 항상 잘 들어야 한다. 사람은 자신의 이야기를 들어주는 사람과 대화하기를 원한다. 자신과 비슷한 사람을 친근하게 받아들이며, 자신에게 관심을 주는 사람을 향해서 눈을 마주한다. 결론을 내기보다는 자신이 하고 싶은 이야기를 다 말하고 싶어한다. 그냥 결론을 전달할 요량이면, 회의를 하지 말고 메일을 쓰는 편이 낫다. 회의는 대화를 통해서 결론에 도달하는 과정을 중시하지만, 명령은 지시의 형태로 한 방향으로 흐르기만 하기 때문이다. 그 편이 상대방의 시간 낭비를 줄이는 길이다.

나무의 높은 곳에 올라갈수록 가지는 가늘어지고, 의지할 것들은 점점 줄어든다. 하지만, 더 많은 것들을 볼 수 있어서 어디로 가야할지 판단할 수 있는 시야는 늘어난다. 더 많이 듣고, 더 자세히 보는 것이 그 나뭇가지까지 오른 사람의 역할이다. 자신의 말만 늘어놓는다면 누구도 말하려고 하지 않을 것이다. 말해봐야 해결해야 할 숙제만 늘어날 것이고, 약점만 드러내는 질문을 받을 것이 분명하기 때문이다. “사람이 입은 하나지만 귀가 둘인 이유는, 말은 줄이고 여러 사람의 의견을 들으라는 뜻”이다. 따라서, 가능한 많은 사람의 의견을 들어야 한다. 비록, 결정은 자신의 몫이지만, 실행은 실무자들의 행위로 나타나야 하기 때문이다.

# [ 결정(Decision)해야하는 순간 ]

어떤 일이든 결정해야 만 더 나아갈 수 있는 힘든 순간이 있다. 과제를 진행하면서 이런 순간이 적어도 한 번은 오기 마련이다. 대부분의 결정은 객관적인 사실에 기초해서 해야겠지만, 어떤 결정들은 근거가 없이도 해야 한다. 특히, 긴급한 일일수록 그럴 가능성은 크다. 어떤 경우에는 작은 결정이 나중에 다가올 어려움의 원인이 되기도 하며, 잘못된 결정이 엉뚱하지만 좋은 결과를 낳기도 한다.

사람은 하루에도 수많은 결정의 상황에 놓인다. “점심으로 무엇을 먹을까? 어떤 음료수를 마시지? 오늘은 어떤 길로 퇴근할까?” 등등 대부분은 순간적으로 판단을 해야 할 것들이 많다. 그에 대한 결과는 맛없는 점심이 될 수도 있고, 달고 맛없는 음료나 생각보다 일찍 집에 도착할 수도 있을 것이다. 하지만, 다음에도 똑같은 상황에 놓이게 된다면 어떻게 판단할까? 아마도 맛없는 식당에는 절대 가지 않을 것이고, 다른 음료를 선택하거나, 집에 가는 최단 경로를 선택할 것이다. 하지만, 식당의 요리사가 마침 바뀐다거나, 원하는 음료를 발견하지 못한 경우, 혹은 도로에서 사고가 나서 지체될 경우도 있다. 즉, 우리의 순간적인 판단의 근거는 과거의 경험에서 나오게 되며, 좋은 결과를 있었을 경우 다시 선택할 가능성 또한 더 높아진다.

물론, 중요한 결정은 앞에서 이야기한 습관같은 결정과는 다르다. 습관은 반복과 경험을 통해서 강화되지만, 중요한 결정은 반복적이지 않을수도 있으며, 경험해 보지 못했을 수도 있다. 하지만, 적어도 자신이 경험하지 못했다고 해서, 남들도 경험하지 못했을 것이라고는 추측할 수 없다. 거의 대부분의 상황은 이미 남들도 동일하게 겪었을 가능성이 크다. 따라서, 중요한 결정 역시 과거의 경험을 재활용해서 근거를 만들어낼 가능성도 높은 것이다. 찾지 못했다고 존재하지 않는다고 생각할 수는 없다. 그리고, 만약 자신이 처한 상황이 다른 사람이 경험한 상황과 완전히 다르다고 생각한다면, 다른 분야의 지식과 경험을 동원해서 해결할 가능성도 열려있다. (이런 것을 우리는 “창의적인 해결”이라고 부른다.)

소프트웨어 개발은 어떤가? 사실 소프트웨어 개발은 100년이 안 된 새로운 분야다. 건축이 인류의 역사에서 가장 오래된 분야라고 한다면, 소프트웨어 개발은 이제 막 태어난 갓난 아기와 같은 상태일 것이다. 하지만, 그 얼마 안 되는 기간 동안 과거의 어떤 분야에서 발생한 변화보다도 더 많은 변화를 만들어 나가고 있다. 심지어 우리 생활의 대부분을 지배하는 많은 것들의 탄생도 소프트웨어의 발전과 발을 맞추고 있다. 급속한 발전 속에서 남은 것은 수 많은 시도와 실패의 경험들이며, 실패에서 발전한 성공의 경험은 이제 어느 정도 경험을 넘어 이론의 토대를 갖추어 가고 있다.

과거의 경험에서 현재의 결정에 대한 정당한 근거를 찾아내야 한다면, 코딩에서 만나게 되는 다양한 문제의 해결책으로는 무엇이 있을까? 사실, 코딩 문제의 해결 방안은 아주 거창한 이론으로 집대성된 것이 아닌, 사소하지만 반드시 지켜야 할 기본적인 것들이 대부분이다. 즉, "남을 위하는 것이, 결국 나를 위한 것이다"라는 기본적인 진리에서 찾을 수 있다. 조금 더 남을 위해서 신경을 쓴다면, 소프트웨어의 발전을 위한 작은 벽돌을 한 장을 쌓는 길이 되기 때문이다. 자신의 개발 경험에 대한 전수, 실패와 성공에 대한 사례 분석 이야기들, 코딩 한 줄의 위대함, 사소한 결함에 대한 방지 팁(Tip) 등, 우리는 이미 많은 부분에서 타인의 조건없는 도움을 얻어 왔다.

다른 사람이 해야 할 순간적인 결정에 조금이라도 도움이 된다면, 자신의 경험(성공이든 실패든 상관없이)은 공유되어야 한다. 공유되는 가치는 전체 효율의 향상이며, 다른 사람의 시간과 돈의 낭비를 줄이는 것이다. 검색은 그 도움이 어디에 있는지 찾아주는 도움의 손길이며, 해당 사이트(Site)에 대한 연결(Link)은 따스한 마음 그 자체다. 이제 다시 중요한 결정의 순간이 온다면, 적어도 소프트웨어 개발 실패의 전철은 밟지 않을 것이 분명하다. (물론, 그 도움을 간절히 여기고 행동으로 옮겨야 하겠지만)

# [ 개발자에게 해서는 안되는 말 ]

소프트웨어 개발의 복잡도는 일하는 사람 수의 팩토리얼(Factorial)에 비례해서 증가한다(물론, 개인적인 의견이지만). 과제를 수행하는 사람의 수가 늘어날수록 개발해야 할 소프트웨어의 분량(라인 수)도 당연히 클 것이고, 그것을 만들기 위해서 필요한 서로 간의 대화 경로도 많아지게 된다. 과제 초기에 생각했던 것보다 이러한 오버헤드는 클 것임이 분명하다. 하지만, 우리는 개발을 시작할 때 이런 부분들에 대한 것은 전혀 고려하지 않고, 그냥 특정 모듈 단위의 개발 일정 만 산정해서 과제 기간을 추정(Estimation)하게 된다.

개발자의 수가 늘어나고 중간 관리자가 생기고, 그리고, 최상부의 관리자가 전체적인 소프트웨어 개발의 진척상황을 보게 된다. 하위 모듈(서브시스템)의 조직 단위에서 모인 보고 자료가 위로 전달되는 경로는 길어지게 되고, 결과적으로 여러 사람이 모여서 일하는 것이 점차 오버헤드를 증가시키는 방향으로 진행된다. 대체로 가장 잘 관리할 수 있는 사람의 수는 여러 사례를 통해서 “7+-2”사이에서 결정되기에, 그 이상의 사람의 필요한 일은 조직을 나누어 관리할 수밖에 없다 따라서, 사람이 많다고 해서 생산성도 높아질 것이라는 예상은 쉽게 빗나가게 된다.

다양한 사람이 모이다 보면 서로 간의 의사소통에서도 문제점을 노출하게 된다. 가장 깔끔한 대화 수단은 문서지만, 문서를 만들고 다시 검토하는 과정도 의사소통이 필요하다. 이런 일이 자주 반복되면, 문서 만드는 시간이 실제로 코딩하는 시간보다 더 길어질 가능성도 있으며, 동작하는 코드가 만들어지는 속도가 늦어지면 불안감도 커지게 된다. 불안감은 성급한 코딩으로 내몰게 되며, 점차 문서의 중요성 보다, 동작하는 코드를 만드는데 집중하게 된다.

물론, 코드도 의사소통의 가장 중요한 수단 중에 하나다. 하지만, 코드 이전에 필요한 것은 개발자들이 나누는 대화가 문서나 코드보다 선행되어야 한다. 대화가 중요한 이유는 얼굴을 맞대고 나누는 말은 기록을 넘어서 “분위기와 상황에 대한 더 많은 정보를 전달”할 수 있기 때문이다. 따라서, 대화를 나누는 행위 자체가 중요하며, 해서는 안되는 말과 해도 되는 말을 구분할 수 있어야 한다. 여기서는 몇 가지 해서는 안되는 말에 대해서 알아보도록 하겠다.

**01. 개발자의 능력을 의심하는 말**

: 사실 이것은 어떤 욕보다도 가장 기분 나쁘게 들릴 수 있다. 즉, 사람의 신뢰에 대한 문제가 되기 때문이다. 객관적인 잣대를 가지고 이야기를 시작하더라도 능력을 의심해서는 결코 안 된다. 오히려, 능력 측정을 잘 못 해서 일을 잘못 할당한 관리자의 잘못이 더 크다과 봐야한다. 개인의 능력이 부족하다고 판단되면, 대안을 만들어주는 것이 관리자의 역할인 것이다.

**02. 근무시간에 대한 말**

: "몇 시까지 일해, 주말에 나와서 일해"와 같이 “일의 내용이 없는 일의 시간”만 있는 말은 해서는 안 된다. 차라리, "어떤 일에 대한 결과를 언제까지 보고 싶다"라거나, 혹은 "될 수 있으면 주말 전까지는 끝내고, 주말에 푹 쉬어" 라고 말하는 것이 도움이 된다. 그렇게 하면, 사람들을 주말에 쉬기 위해서 주중에 더 열심히 일하게 될 것이다. 일을 남겨 두고 집에서 편하게 두 발 뻣고 잠자는 개발자는 없다.

**03. 버그의 수로 그 사람의 역량을 평가하는 말**

: 버그가 많고 적음으로 해당 부분에 대한 개발자를 나무라는 것은 옳지 않다. 대체로 일을 많이 하고, 잘하는 개발자일수록 양산하는 버그도 늘어나는 경향이 있다. 많이 코딩하면 당연히 버그도 많이 나온다. 물론, 개발자가 버그를 많이 만드는 것이 전적으로 옳다는 것은 아니다. 반복적인 실수를 하지 않는 것이 중요하다. 그렇다면, 차라리 버그의 수를 통계로 만들기 보다는, 해당 개발자가 작성한 코드를 보고 문제점을 지적하는 게 훨씬 효과적일 것이다. 다른 방법으로는 단위 테스트나 통합 테스트를 할 수 있는 “시간과 기준”을 만들 필요가 있다. 다그친다고 해서 일이 저절로 되는 것은 아니다.

**04. 일정을 줄이라는 말**

: 일정을 정하는데 가장 중요한 고려 요소는 “만족해야 할 기능의 수, 현재 개발자의 역량, 과제에 할당된 자금 수준, 그리고, 만족해야 하는 품질 수준”이다. 따라서, 이러한 것들에 대한 고려 없이 단순히 일정을 줄이라고 하면, 줄일 수 있는 것은 기능과 품질 밖에 없다. 결국은 안 팔리는 물건을 100% 일정 준수로 만들 뿐이다. 미안한 이야기지만, 이것은 회사의 의도가 결코 아닐 것이다. 일정 준수가 중요한 이유는 관리자의 업적이외에는 없다. 개발자는 단축된 일정을 만족시키기 위해서 더 많은 버퍼를 이야기할 것이고, 더 많은 과정의 생략만 낳을 뿐이다. 당연히 완료된 이후에 발생하는 비용은 누구의 책임도 물을 수 없다.

**05. 시키는 일만 하라는 말**

: 시키는 일 만하면 정말 아무 문제가 없나? 문제는 시키는 일이 전부가 아니라는 점이다. 여러 사람이 시킬수도 있고, 해야할 이외에 추가적인 일이 주어질 수도 있다. 시키는 일만 하는 사람이라면 그냥 단순한 기계처럼 일 할 뿐이다. 능동적인 생각이 없이 그저 주어진 명령을 “누가 했는지에 따른 우선순위로 스케줄링해서 처리할 뿐이다. 과제에서 우선순위가 높은 일을 하는 것이 아니라, 명령하는 사람이 좋아하는 일을 먼저 처리하게 된다. 그렇게 한 일에 대한 책임도 자신의 의지와는 상관없으므로 명령을 내린 사람의 져야한다. 회의는 점차 생명력을 잃게 되고 명령과 복종 만이 있을 뿐이다.

**06. “왜 말이 없어?” 라고 묻는 말**

: 말(대답)이 없어지기 전에 그 사람에게 어떻게 했는지 부터 살펴보기 바란다. 말이 없다는 것은 서로 간에 소통이 아닌 일방적인 지시나 책임 추궁 만 있었다는 의미다. 실수로 잘못 대답 했다가는 자신의 책임이 되기에 묵비권을 행사하고 있을 뿐이다. 소통하려거든 최소한 질문하는 사람 자체가 태도를 바꾸거나, 그게 힘들다면 그냥 보고서 만 받아서 읽어보는 편이 낫다. 좋은 질문은 좋은 대답을 낳는다. 그렇지 않은 질문은 침묵 만이 있을 뿐이다.

이런저런 이야기를 쓰다 보니, 다양한 생각이 드는 것이 사실이다. 중요한 것은 "내가 받기를 원하지 않는 것은, 남에게도 하지 말라는 것이다." 존중해야 존중받을 수 있으며, 같이 해야 고민을 덜 수 있다. 혼자 하는 일방적인 말은 소통이 아닌 연설이며, 연설의 효과는 짧을수록 집중해서 듣게 된다. 길어지는 연설은 졸림 만 낳을 뿐, 한쪽 귀에서 들렸던 말들은 그 옆에 있는 귀에서는 슬그머니 기어나가 버린다. 관리자가 되려거든 해야 할 말과 하지 말아야 할 말을 가려서 해야 한다. 남을 돕는 것이 결국 자신을 돕는 길임을 잊지 말아야 할 것이다. 개발자는 도움을 원하는 존재이지, 질책과 추궁으로는 움직이지 못한다.

# [ 가르침이란 ? ]

어떤 문제가 생겼을 때 혹은 하는 일이 지연되고 있을 때 가장 힘든 것은 사람들의 시선을 피하는 일이다. 관심의 집중을 피하는 것이 핵심이다. 시간을 벌어야 하는 상황에서 오히려 그 문제로 인해서 다른 사람의 집중적인 관심을 받게 되고, 그로 인해서 해야 할 일을 하지 못하는 상황이 된다. 지나친 관심은 다양한 보고서를 낳게되고, 과제에 집중하는 시간보다 숙제(?)하는 시간이 늘어나게 된다.

참으로 아이러니한 상황은 안되는 조직일수록 온갖 문서와 회의, 그리고 갑자기 늘어난 교육 및 간담회 등등으로 인해, 개인들은 더 지치고 힘들어하게 된다는 사실이다. 새로운 윗사람이 오면 그 사람이 원하는 것이 무엇인지를 알아야 하고, 그것에 맞춰 온갖 보고자료들을 만들어 질문에 대답해야 한다. 오히려 실행에 집중하라고 요구하는 구호가 무색할 지경이 된다.

진정한 가르침이란 문제의 핵심을 파악하려고 다른 사람을 힘들게 하는 것이 아니라, 그들이 최대한 자신의 역량을 충분히 발휘할 수 있도록 조용히 지켜봐 주는 것이다. 간혹, 어려운 상황이 닥쳤을 때, 도움의 손길을 원하는 바로 그 순간에 적절한 조언을 해주는 것이 최선이다. 이런 저런 문제가 벌어진 상황에서 거기에 다시 기름을 부어서 불난 집에 선풍기 틀고 있는 일을 해서는 안 된다. 경험이 있다면, 제발 그 경험을 “기다림의 미학”으로 묶어 놓기를 바랄 뿐이다.

물론, 위와 같은 생각은 전적으로 아직 조직의 최상부에 올라가지 못한 사람의 단순한 생각일 뿐이다. 높은 곳에서 바라보는 것은 어쩌면 전혀 다른 그림일 수도 있다. 마치, 지리 교과서에 나오는 사진 같은 지형은 평생 걸어만 다녀서는 못 보듯이, 더 높은 곳에 올라야 만 알 수 있을 것이다. 하지만, 높은 곳에 오르더라도, 그곳에 오르기 위해서 흘렸을 수 없는 땀방울이 왜 필요 했는지는 잊지 말기를 바랄 뿐이다.

“진정한 가르침은 기다리는 것”이다. 대답을 강요하는 것이 아니라, 대답을 함께 찾아가는 것이다. 비록 만족하지 못하는 답을 찾게되더라도 다음을 기약할 수 있도록 사람을 지키는 것이다.

# [ OSD: Ownership, Speed, Detail ]

OSD는 책임감, 속도, 세밀함이다. 물론, 임베디드 시스템을 개발하고 있는 사람이라면, OSD란 말을 들었을 때, “On Screen Display(화면을 계층화시켜서 영상과 정보를 별도의 겹쳐진 화면에 쓰는 기술)을 먼저 생각할 것이다. 이 세 가지를 해야 제대로 일 하는 것이라고 한다. 하지만, 정말 일하는데 이 세 가지 만 만족 되면 아무런 문제가 없을까? 사실 이것들에 대해서 왜 중요한지 근본적인 이유부터 알아야 할 것이다.

직장에서 느끼는 책임감은 어디서 올까? 그것을 윗사람이나 아랫사람 혹은 동료에 대한 신뢰에서 나온다. 신뢰가 없는 상태에서 책임을 묻는다는 것은 그냥 강요에 불과할 뿐이다. 즉, 맡겨진 일을 처리하라는 압박 밖에는 될 수 없다. 신뢰하는 사람에게는 믿음을 가지고 일을 주게 되며, 믿음을 가진 사람을 실망시키지 않기 위해서 최선을 다하려는 마음은 자발적으로 생길 수밖에 없다.

신뢰를 쌓는데는 오랜 시간이 걸리지만 한 순간에 무너질 수도 있다. 신뢰는 그 사람으로 부터 기대하는 행동이 있고, 그것이 올바른 형태로 나올 것이라는 믿음을 가질 수 있게 만든다. 설령, 그것이 잘못되더라도 의도적으로 그렇게 하지 않았으며, 불가항력으로 그렇게 했다고 믿게 만드는 마음이다. 쉽게 신뢰를 망치는 방법들은 많다. 그 중에서도 가장 쉬운 것은 상대방에 대한 존중없이 내뱉는 비꼬는 투의 말이다. 그 한마디면, 그 동안 공들였던 모든 것을 아주 빨리 없앨 수 있다.

속도는 중요하다. 사실 시간을 못 맞추고 출시되는 제품들은 시장의 평가를 받기도 전에 사라지기 마련이다. 남이 먼저 한 것을 따라 해 봐야 고객들은 이미 그 제품이 "One of Them"이라고 생각할 것이다. 그렇다면, 속도를 맞추기 위해서는 어떻게 해야 할까? 이미 답은 있다. 만드는 시간을 줄이는 수밖에 없다. 기능을 줄이고, 테스트해야 할 항목을 줄이고, 재활용을 하는 수 밖에는 달리 방법이 없다. 기능이 늘어나면 구현 시간도 오래 걸리고, 테스트해야 할 항목도 당연히 늘어나게 되며, 거기서 발생하는 문제점을 해결한다고 다시 한참을 고생해야 한다. 당연히 그렇게 만들어진 제품은 시장에서 요구하는 시점에 전달되지 못한다.

새로 개발하는 항목이 늘어날수록 재활용은 줄어들기 마련이고, 새로운 것은 새로운 문제점을 만들어낼 뿐이다. 없어도 되는 것은 과감히 없애자. 시스템의 기능 중에서 사용자가 사용하는 기능은 20% 밖에 안 되고, 나머지 80% 기능은 있는 줄도 모르는 경우가 많다. 물론, 어떤 사용자는 기존에 가진 기능이 없어졌다고 불평할지도 모른다. 하지만, 그때는 제품의 원래 목적이 더 좋아졌다는 것을 증명해서 고객을 설득할 수도 있다. 사용자들이 잘 사용하지도 않는 기능에 엄청난 공을 드리고도, 사용하기 편리하지 않은 기능들은 많이 있을 수 있다.

개발자의 눈이 아닌 사용자의 눈을 가지기는 쉽지 않지만, 일단은 내가 해보고 불편한 것들은 다 없애는 것이 최선이다. 한 번의 마우스 버튼이라도 줄이고, 입력되는 한 글자라도 줄일 수 있다면, 고객의 만족감은 줄인 만큼 더 좋아질 것이다. 기능을 편의성으로 도저히 다 덮을 수 없다면, 없애는 기능을 대체할 수 있는 더 좋은 기능을 하나 쯤 추가하는 것도 한 가지 방법이다. 하지만, 마냥 늘어만나는 기능 들을 그냥 가지고 움직이기에는 우리가 해야 할 일은 너무 많고, 시장은 기다려 주지 않을 것이다.

세심함은 모든 일에서 장인이 되기 위해 가져야 할 기초 능력이다. 세심한 고려가 없다면 명품은 있을 수 없다. 하지만, 우리가 만드는 것이 명품일까? 수십 수백 명이 붙어서 만드는 제품을 장인의 손길을 느낄 수 있게 하는 것이 가능한 일일까? 여기서 중요한 것은 세심함의 수준이다. 흔히 놓칠 수 있는 것들에 대한 고려이고, 그런 부분들은 대체로 기본에서 출발한다. 기본기가 없는 사람에게 세심함을 요구하는 것은 사실 웃기는 일일뿐이다. 알지도 못 하는 일을 아주 상세히 설명하라고 요구하는 사람이 있다면, 그런 명령을 준 사람에게 이렇게 이야기하라. "직접 하세요."

세심함의 수준은 점점 높아지는 것이고, 일정 수준을 넘어서게 되면 그 분야에 대한 장인이 되는 것이다. 하지만, 모든 것에 대한 장인이란 있을 수 없다. 복잡도가 높은 일을 하는 경우에는 특정 분야 밖에는 알 수 없는 것도 사실이다. 오히려 자신이 맡은 부분에 대한 세세함을 갖추고, 다른 사람이 맡은 부분은 권한을 맡기는 것이 더 좋다. 모든 것을 다 잘하는 슈퍼맨을 일개 직장인에게 요구한다면, 다른 사람들로부터 받는 기대를 만족시키지 못하는 상황이 온다. 좌절은 극히 개인적인 것이지만, 좌절로 인한 의욕 상실은 조직의 문제가 될 것이다. 세세함을 기대하는 것도 수준에 맞게 하기 바란다. 누구나 역할이 있듯이 그 역할의 수준에서 보는 관점과 세세함은 다르기 마련이다. 모든 것을 기억하라고 요구한다면, 수첩이나 자료만 찾게 만들 뿐이다.

OSD가 “On Screen Display”건, “Ownership, Speed, Detail”건 상관없이 중요한 것은 사람의 문제다. 사람에 대한 신뢰를 쌓는 것에 인색해서는 안 된다. 속도를 만족하게 하기 위해서는 해야 할 일과 하지 말아야 할 일을 명확히 하는 것이 중요하다. 또한, 세밀함은 그 사람이 맡은 역할 수준을 고려해서 기대해야 한다. “코끼리도 춤추게 하는 방법이 칭찬”이라고 말하지만, 자기와 같이 일하는 동료나 선후배에게 그런 칭찬에 인색한 것은 아닌지 생각해 보기 바란다. 자신이 원하는 것이 있다면, 먼저 주는 것이 소위 말하는 “권한을 가진 사람”이 해야 할 일은 것이다. 신뢰를 원한다면 먼저 신뢰하는 행동을 보여야 하며, 속도를 원한다면 가볍게 움직일 수 있는 조직 부터 만들어야 한다. 세심함은 말로 떠들것이 아니라, 무엇이 정말 중요한지 부터 파악해야 할 것이다.

# [ 누가 고양이 목에 방울을 달 것인가? ]

문제는 누가 힘든 일을 실제로 할 것인가를 정하는 것이다. 많은 사람이 이렇게 해야 한다 저렇게 해야 한다고 이야기할 수 있을지는 몰라도, 그 일을 하는 사람의 입장이 되어 직접 일을 하지 않는다면 아무 소용이 없다. 정말 중요한 것은 실행하는 것이다. 잘 되는 일에는 모든 사람이 숟가락을 놓지만, 잘되지 않는 일에는 아마도 나서지 않는다. 그냥 뒷짐 지고 이래라 저래라 할 뿐이다. 하지만, 정말로 필요한 것은 그런 것들이 아닌, 실제 업무를 하는 사람들이 최선을 다해서 일할 수 있도록 “환경을 만들고 배려하는 것”이다.

최선을 다해서 일하게 하는 것은 생각보다 쉬운 일이다. 즉, 그 사람들이 현재 집중하고 있는 일에 최고의 집중력을 가질 수 있도록 환경을 만들어주는 것이다. "왜 안되는지 보고해!"라고 말 하는 순간, 그 사람들에게는 일 보다는 보고가 우선이라는 것을 심어주게 된다. 지상 최고의 과제는 현재 개발 중인 것이 아니라 보고서가 되어 버린다. 중간에 끼어드는 일도 많다. 고위층의 누군가 지시한 일은 현재 지연되고 있는 과제의 개발과는 무관한 최고 우선순위를 지닌다. 하지만, 그 일은 잠시 지나고 나면 잊히지만, 지연된 과제는 그 일이 끝나고 나도 계속 남는다. 결국, 과제 일정 준수는 지키지 못할 공허한 약속이 되어 버린다.

사람이 갖추고 있는 능력을 최고로 발휘하게 만드는 것은 그 사람의 노력에 대한 인정이다. 간담회라는 자리를 통해서 의견을 듣는 것은 말 그대로 “간과 쓸개를 빼놓고 하는 회의” 이상은 되지 못한다. 해야 할 말 만 오가고, 누군가의 하고 싶은 말 만하고 만다. 솔직한 심정으로 자신의 마음 속에 있는 이야기를 하는 상황은 극히 제한적인 경우다. 그런 것들도 솔직하게 이야기할 수 있는 분위기를 만드는 것은 오랜 노력이 필요하다. 그런 노력 없이 그냥 하라고 해서는 아무것도 나오는 것이 없다. 차라리 그런 자리를 만들지 않는 것이 오히려 도움을 주는 일이 될 것이다.

말에는 힘이 있다. 그 말로 인해서 사람의 감정을 상하게도 하고, 또한 그 말로 인해서 어떤 어려움도 극복할 수 있는 용기를 주기도 한다. 한 마디의 말이 그 어떤 가치보다도 소중하다는 것을 알고, 오늘도 열심히 묵묵히 자리를 지키며 야근하고 있는 팀원들에게 힘을 줄 수 있도록 해야야 할 것이다. 누군가 고양이 목에 방울을 달아야 한다면, 차라리 고양이가 없는 조직이 더 나을지도 모른다. 눈치만보고 일하고 시키는 일만 할 것 같으면, 눈치를 볼 필요가 없이 일하는 조직이 더 효율적일 것이다. 고양이는 위협으로 쥐들을 몰아갈 수는 있지만, 쥐들은 최악의 경우 집을 옮길 수도 있다. 그리고, 더 중요한 것은 개발자들은 쥐가 아니라는 점이다.

# [ 아무도 답을 주지는 않는다. ]

일하다 보면 정작 중요한 결정이 필요할 때 아무런 답을 주지 않는 경우를 종종 본다. 또한, 중간에 끼어서 이러지도 저러지도 못하는 자신을 발견하기도 한다. 도대체 어떻게 하라는 것인가? 회사를 따를 것인가? 아니면 바로 위에 있는 부서장의 명령을 따를 것인가? 힘든 문제가 아닐 수 없다.

책임자의 역할이란 결정이 필요한 순간 그 결정을 합리적으로 내리고, 그 결정에 따라오는 결과를 받아들이는 것이다. 잘못은 자신의 것으로 하고, 상은 아래 사람에게 돌리는 것이 그 자리에 선 사람이 해야 할 일이다. 일에 집중할 수 있는 환경과 사람들의 자발적인 열성을 끌어내야 하고, 곳곳에 있을지도 모를 비효율을 없애 나가는 것이다.

그럼, 정말 이런 일들을 할 수 있을까? 안타깝지만 이상적인 것들은 쉽게 되지 않는 것들이다. 부서원들에게 야근을 강요해야 하고, 주말에 특근하는 것을 당연하게 받아들일 수 있도록 “도전적인 목표”를 세워야 한다. 윗 사람이 원하는 것이 무엇인지 항상 관찰하고, 그 코드에 맞춰야 행동해야 한다. 자신의 의견은 되도록 이야기하지 않고, 그냥 시키는 일 만 열심히 완성해 나가야 한다. 중요한 것은 내 생각이 아니라, 명령을 내리시는 분의 의도인 것이다.

물론, 이렇게 일하면 더 이상 희망이나 비전 같은 것은 찾지 않게된다. 사실 자신이 왜 일을 해야 하는지도 잊고 말 것이다. 후배들이 이렇게 생활하고 있는 선배를 보는 것도 힘들어 할 것이다. 그들에게도 희망이나 미래에 대한 비전이 없어 보일 것이기 때문이다. 결정을 내리는 것은 관리자의 몫이며, 책임도 당연히 관리자의 몫이다. 결정에 필요한 정보는 실무자들이 제공하는 것이지만, 그들에게는 정확한 정보를 전달할 의무만 있을 뿐이지, 결정은 오로지 관리자가 해야할 일이다. 따라서, 답을 주지 않는 관리자는 자신의 역할과 책임을 다하지 않고 권리만 주장하는 것이다. 그런 관리자를 만나는 것은 스트레스를 넘어선 고통일 뿐이다.

사실, 이런 이야기를 여기에 적는다고 달라질 것도 없다. 창조적인 생각은 여유에서 오고, 오늘 이렇게 적는 글은 그 생각의 여유를 표현할 수 있는 한 방법일 뿐이다. 비록 아무것도 달라질 것이 없더라도, 자신의 근본을 잊어서는 안 되기 때문이다. 어떤 상황에 처하더라도 자신의 역할과 책임을 잊어선 안된다. 프로페셔널로 일 한다면 적어도 자신이 책임져야 할 부분에 대한 것은 당연하게 받아들여야 한다. 판단의 근거를 명확하게 제시할 수 있어야 하며, 비록 욕은 먹을지라도 거짓을 참이라고 받아들여서는 안된다. 자신의 맡은 역할은 확실히 해 주어야만 회사에서 받는 연봉이 부끄러온 돈이 되지 않을 것이다.

# [ 기본으로 돌아가자. ]

예전에 대학원을 다니던 시절을 되돌아보면, 존경하던 교수님은 항상 고급 개념을 이야기하고 있을 때 질문을 던지시곤 했다. 고급 개념을 이해하는데 필요한 바탕 지식에 대한 것을 물어보는 것이다. 예를 들어, 파일 시스템을 열심히 설명하고 있으면 파일이 뭐냐고 물어본다. 사실 이런 질문을 처음 받으면 황당해하기 마련이다. 첫 번째는 그것을 왜 물어보실까?” 하는 의문이, 그리고 나서는 정말 그 질문에 대해서 답을 할 수 없다는 모래성 같은 “지식의 허망함”이 머리를 흔들고 지나간다. 정말 중요한 것은 “기본을 잘하는 것”이다.

누군가 이야기했다. "평생을 살기 위한 모든 것은 유치원에서 다 배운다." 우리가 정말 잘 해야 할 것들은, 무엇을 처음 시작할 때 배웠던 기초적인 것들을 지속해서 갈고 닦아서 무의식 중에도 행하는 것이다. 소프트웨어 개발도 마찬가지다. 초급 때 경험 많은 프로그래머에게서 배운 것들과 학교에서 교과서를 통해서 배웠던 기본기를 꾸준히 자신의 것으로 만들려고 노력하는 자세다. 누구나 처음 부터 잘 할 수는 없지만, 누군가는 반드시 자신보다 잘하기 마련이다. 그런 큰 차이도 기본이라는 사소한 작은 차이에서 시작된다.

소프트웨어 개발자로 일하면서 기본으로 해야 할 것들은 무엇일까? 사실 아주 평범하지만 제대로 지키지 못하는 것들일 뿐이다. 아래에 몇 가지를 나열해 보겠다.

**01. 항상 검사할 것**

; 함수의 호출 후 돌려받는 값은 반드시 확인해야 한다. 모든 함수는 될 수 있으면 복귀값을 전달하는 것이 좋다. 그리고, 그 함수가 호출될 때 넘겨주는 값들도 올바른 값인지 확인해야 한다.

**02. 이해하기 쉽도록 작게 만들 것**

; 짧은 프로그램을 짜는 것이 아니라, 이해하기 쉬운 프로그램을 작게 짜는 것이다. 복잡하다는 생각이 들면, 그것을 디버깅 할 생각을 하지 말고 더 작게 나누도록 해야 한다. 작은 것은 디버깅 하기도 쉽다는 것이 진리다. 읽기 쉽게 짜야지 이해하기도 쉽다.

**03. 데이터 정의와 관리는 한 곳에서 할 것**

; 데이터의 전달이 많아지고, 다루는 곳과 분리된 곳에서 정의되면 코드의 이해를 어렵게 한다. 따라서, 될 수 있으면 자료구조는 그 자료구조가 있어야 하는 곳에서 정의되는 것이 구조적 문제를 푸는 시작이다. 그리고, 자신이 어떤 데이터를 관리하는지 남은 알 필요가 없는 것이 좋다.

**04. 일관되게 만들 것**

; 변수, 클래스, 함수, 인터페이스, 파일 이름, 디렉터리 등등 모든 이름 붙일만한 것들은 일관된 규칙을 가지고 정하고, 그 규칙을 다른 개발자들도 같이 이해하고 따르게 해야 한다. 될 수 있으면 이름은 이해하기 쉬워야 하고, 이름으로도 그것이 왜 필요하고 무엇을 하는지 이해할 수 있게 해야 한다.

**05. 테스트할 것**

; 작은 단위로 만들어진 것은 테스트하기도 편하다. 작게 코딩하고 많이 테스트할수록 당연히 버그수는 줄어들기 마련이다. 자신이 만든 코드에 자신감을 가지는 가장 쉬운(효과적인) 방법은 결국 테스트 프로그램을 작성하는 것밖에 없다.

**06. 남들과 공유할 것**

; 공유되지 못하는 정보는 가치가 없다. 정보는 공유가 되어야 가치를 가지게 되며, 가치가 있어야 살아남게 된다. 작성되는 모든 코드는 공유되어야 하며, 어떤 사람이라도 개선할 수 있는 곳에 공개되어야 한다. 더 높은 수준으로 가기 위해서는 나보다 뛰어난 사람들이 세상에 많다는 것을 인정해야 한다.

**07. 최소한의 정보만 공개할 것**

; 너무 많은 정보의 공개는 혼동을 일으키는 주범이다. 될 수 있으면 적은 정보만 공개해야 나중에 변경하기도 쉽다. 결과적으로 알아야 할 것들이 많아지면 실수할 확률도 높아진다. 변경이 잦은 부분에 대해서는 인터페이스와 구현을 분리하도록 해야 할 것이다. “인터페이스는 계약”이며 구현은 계약을 이행하는 방법을 자유롭게 선택할 수 있도록 만든다.

**08. 동적인 상황은 최소화할 것**

; 동적인 부분이 완전히 제거되면, 버그가 생길 위험성은 현저히 낮아진다. 쓰레드(Thread)의 생성, 메모리의 할당, 불특정 상황에서 발생하는 이벤트(Event)등 사실 이런 것을 완전히 제거하기는 어렵다. 하지만, 될 수 있으면 그런 것들이 적어지는 방향으로 코딩할 필요는 있다. 목적은 “예측 가능하며 안정적”으로 원하는 성능을 만족하는 시스템을 만드는 것이다.

**09. 많이 사용되게 하고, 적게 사용하게 만들 것**

; 자신이 만든 코드를 최대한 많이 사용되게 만드는 것이 중복을 줄여준다. 다른 사람이 만든 코드를 최소한으로 사용하는 것이 의존성을 줄이는 방법이다. “코드의 중복과 의존성”은 시스템을 변경하기 어려운 구조로 만드는 주범이며 버그의 온상이다.

**10. 성능보다 구조를 먼저 생각할 것**

; 성능을 좋게 만들려고 짠 코드는 의존성이 높은 코드를 생성해 낸다. 결국, 무언가를 추가하려고 할 때, 그것을 만든 사람이 들인 것보다 많은 노력(Effort)을 소비하게 만든다. 구조화된 코드는 성능에 대한 오버헤드가 유지 보수를 위한 비용보다 항상 적을 것이다. 또한, 구조화된 코드가 최적화도 더 쉬운 것이 일반적인 진리다.

교과서는 여러분이 시험을 치는데 필요한 책장 위의 장식물이 아니다. 장식이 필요하다면 이사할 때 짐만 되는 책은 별로 도움이 안 될 것이다. 이론과 실제가 다르다고 이야기하는 사람들은 이론을 모르거나, 잘 못 이해하고 있을 가능성이 높다. 오히려 지극히 실무를 잘 알고 분석해서 만들어 놓은 것이 이론이며, 모든 경우의 해법은 아닐지라도 기본적으로 지켜야 할 규칙은 될 수 있을 것이다.

응용은 기본기가 바탕이 되어야 제대로 발휘될 수 있다. 기본을 무시한 응용은 역사를 다시 거슬러 인류가 그동안 만들어 놓은 지식 체계를 부정하는 것 밖에는 될 수 없다. 물론, 모든 이론이라고 다 옳은 것은 아니다. 하지만, 적어도 짧지 않은 소프트웨어 개발의 역사에서 밝혀진 사실은 충분히 인정할 수 있어야 한다. 소프트웨어의 개발이 가장 높은 수준의 인간 정신 활동의 산물이라고 생각하는 사람이라면, 최소한 자신이 하는 것에 대한 명확한 근거는 탄탄하게 가져가야 할 것이다. 기본은 어떤 문제를 풀더라도 변하지 않는다.

# [ 휴가에 대해서 ]

휴식은 일하기 위해서 필요한 부분이다. 사람과 기계가 다른 점이 바로 휴식을 해야 한다는 점이다. 창의적인 생각은 휴식에서 나온다. 여유없이 지속해서 일에 만 신경을 쓴다면, 제대로 된 해결책을 찾는 것보다는 빠른 해결책 만 찾게 된다. 피로가 누적된 결과는 현재로서는 눈에 잘 보이지 않을지 모르지만, 시간이 지나면 서서히 자신의 목을 스스로 조이게 될 것이다.

사람들은 미래를 말한다. "언제쯤 되면 좀 쉽게 일할 수 있을 것이다. 혹은, 언제가 되면 사업이 제대로 될 것이다. 프로젝트는 이때가 되어야 결과가 보일 것이다." 등등 여러 가지 미래에 대해 낙관적인 이야기를 한다. 하지만, 미래는 항상 예상하지 못한 일이 생길 것이고, 그 일 때문에 또 다른 어려움을 겪을 것이다. 어차피, 지금이나 미래는 별 다른 차이가 없다. 일이 잘 되면 잘 될수록 해야 할 일은 더 많아진다. 잘 안되면 잘 안 될 수록 또한 더 잘하려고 노력해야 한다. 따라서, 장래를 밝게 생각하는 것은 좋지만, 미래를 위해서 현재를 희생하도록 강요하는 것은 옳지 못한 결정이다.

오히려 현재를 살아가는데 필요한 따스한 말 한마디와 태도에서 동료와 선후배에 대해서 “존중(Respect)”을 느끼게 하는 것이 옳지 않을까? 미래는 그 때 사람들에게 맡겨두고, 지금 필요한 “감사”를 보내는 것이 더 현명하다. 누군가 휴가를 간다면 웃는 얼굴로 잘 다녀오라는 말 한마디, 진심 어린 걱정을 나누는 태도가 지금 우리에겐 더 필요한 부분이다.

휴가는 삶의 여백을 만든다. 잠시 쉬어 가는 공백은 바쁜 회사 일에서 자신만의 시간을 찾아내는 적극적인 자기 표현이다. 스스로 만들고 스스로 즐길 수 있다면, 그것보다 더 가치있는 일이 있을까? 여백은 새로운 마음 가짐을 만들고, 창조적인 생각을 돕기 위해서도 필요하다. 빈 곳이 없으면 제대로 그림을 맞추기도 어렵듯이, 쉬지 않는 사람은 창의적인 방법을 고민할 시간도 없다. 바쁘다는 이유로 미루기만 한다면, 정말 필요한 일을 할 수 있는 시간은 주어지지 않는다. 공백은 거기 있어야만 다른 것들을 더 돋아나게 보이게 만들기 위해서 필요한 시간인 것이다.

# [ 삶의 여백(Slack) ]

소프트웨어 개발자로 일하는 것을 3D업종에 비교하는 경우가 종종 있다. “Dirty, Dangerous, Difficult“에 추가해서 “Dreamless”를 포함한 4D업종이라고도 한다. 꿈이 없어지는 직업, 어찌 보면 가장 사람의 이동이 많을 수밖에 없다. 꿈이 없는 직업을 오래 동안 지속하는 것은 자신에 대한 욕구를 희생해야 하기 때문이다. 사람은 “생물학적 욕구, 안전의 욕구, 사회적 소속감, 자기 만족이나 존중감, 자아 실현”을 원한다. 희망없는 삶이란 생물학적 욕구나 안전만을 보장할 수 있을 뿐, 성숙한 인간으로 발전하기 위해서 필요한 “동기”를 불러 일으키지 못한다. 그렇다면, 왜 소프트웨어 개발이 꿈이 없어지는 직업이라고 불리는 불명예 달게 되었을까?

최적화하는 것은 어찌보면 지극히 당연한 일일지도 모른다. 물리적인 시간의 최적화, 비용의 최적화, 적은 비용으로 효과를 극대화시키는 일 등등 모든 경쟁에서의 승리는 얼마나 최적화할 수 있는가에 달린 듯이 보인다. 하지만, 정말 그럴까? 소비자의 지갑을 여는 데 있어서 그것이 최선일까? 소비자의 관점에서 생각해 보면 물건이 어떻게 만들어 지는지는 상관하지 않는다. 물건의 가격은 구매에 상당히 중요한 고려 요소이기는 하지만, 비용을 지급한 제품이 제대로 동작하지 않을 때 느끼는 실망감은, 두 번 다시 그 제품을 선택하지 않게 만든다. 조금 비싸더라도 원하는 기능을 제대로 하는 제품을 사는 것이, 장기적으로 봤을 때는 더 큰 만족감을 소비자에게 줄 수 있다.

물론, 보이지 않는 부속품을 만드는 사람들에게 이러한 것이 무슨 소용인가하고 이야기할 수도 있지만, 소비자는 언제나 최고의 만족을 주는 제품을 선택할 것이라는 점은 부정하지 못할 것이다. 따라서, 우리는 사용자의 만족을 위해서 일 해야지, 단순히 효율을 위해서 일해서는 안 된다는 것을 경험적으로 알게 된다. 여기서 경험이라고 이야기한 것은 실패한 자가 배울 수 있는 최고의 가치이다. 경험하지 못하면 절대 알 수 없는 것이 실패이기 때문이다. 하지만, 대부분의 경우 우린 이런 쓰라린 실패의 경험을 다른 방법으로 해결하려고 한다. 즉, 판매의 실패를 가격이 비싸다는 이유로, 혹은 이런 저런 기능이 없어서 안된다는 식으로 만회하려 한다. 이런 실패의 경험은 결국 “더 많은 제품을 더 싸게, 더 빨리 만들어 달라는 것”으로 귀결되고, 점점 더 특징 없는 많은 제품들로 라인업이 채워지게 만든다.

제품이 많아진다는 것은 유지 보수를 위한 비용을 증가시켜, 새로운 제품 개발을 더디게 만든다. 소프트웨어 개발 비용은 인건비가 대부분이며, 개발 시간과 비례해서 증가하는 경향을 보인다. 따라서, 고비용 저부가가치 생산이 이어지도록 만들 뿐이다. 당연히 인당 매출은 낮아질 수밖에 없고, 만들면 만들수록 비용만 증가하는 악순환을 만들 뿐이다. 이런 상황을 극복하기 위해서는 최적화에 대한 새로운 개념을 만들어야 한다. 비용만 줄이는(사람만 줄이고 제품의 수는 늘리는) 최적화 보다는 “최소의 비용으로 최대의 고객 만족”을 주는 “진정한 최적화”를 고민해야 한다.

중요한 것은 제품을 사용하는 고객의 "만족감"이다. 만족감은 어떤 기능에 대한 것, 안정성, 성능, 보안, 사용성, 호환성 등의 다양한 형태로 주어진다. 핵심이 되는 만족감은 그 중에서도 제품의 본래 목적이 무엇인지에 달려있다. 제품을 사려는 사람은 모든 기능을 다 보고 사지 않는다. 특정한 기능이 제대로 동작하는지, 그것을 어떻게 사용할 수 있는지 만 확인한다. 제품의 매뉴얼을 다 읽어보고 사용하는 사람은 없기에, 사용하기 쉽게 설계가 되어야 한다. 핵심이 되는 기능은 반드시 지원되고, 필요 없는 기능은 제거되어야 할 것이다. 끝없는 기능 추가는 제품의 완성도를 낮추고 복잡도만 높일 뿐이다. 비용이 가장 크게 증가하는 이유는 왜 있는지도 모르는 사용되지 않는 기능 때문이다.

삶의 여백은 "만족감"을 이루는 중요한 바탕이다. 여유가 없는 삶은 만족을 위해서 일하는 것이 아니라, 효율의 극대화 만을 위해서 일하는 것이다. 최적화는 효율의 극대화를 이룰 수 있을지는 몰라도 만족의 극대화와는 관계가 없다. 마치, 후회가 없으면 행복할 것이라고 믿는 것과 같은 생각이다. 스스로 마음의 여유가 없다면 시장의 변화에 대해 적응을 할 틈을 찾을 수 없으며, 남의 기쁨에 대해서도 소홀해질 수밖에 없다. 나의 삶에 틈이 없는데 어떻게 다른 사람(사용자, 고객)을 배려하려는 마음이 생길 수 있을까? 제품이 안 팔리는 이유를 물어본다면, 그 이유는 제품을 만드는 우리 자신의 삶에 여유가 없기 때문이다. 기쁜 마음으로 만들지 못한 음식이 어떻게 맛있을 수 있겠는가?

바쁜 우리에게 필요한 것은 조금 천천히 일하는 것이다. 천천히 정확히 할수록 더 빨리 갈 수 있다. 이것은 진실이다. 우린 어릴 때 부터 이런 말을 자주 듣지만, 지켜보는 사람들은 항상 빨리라는 말에 익숙해져 있다. 특히, 복잡함을 다루어야 하는 소프트웨어 개발자에게는 천천히 정확하게 하는 것이 필수지만, 너무나 자주 간과되고 있는 부분이기도 하다. 복잡한 문제를 해결하는 방법은 다양하게 존재하지만, 그 방법들 모두가 좋은 해결책은 아닐 수 있다.

사람은 단순한 일은 빨리할 수 있다. 기계적인 조립을 요구하는 것은 오랜 시간의 경험과 교육이 필요하지는 않다. 물론, 빠르기를 따져서 순위를 결정한다면 최적화 된 행동이 요구될 수도 있겠지만, 복잡함을 해결하는 것은 단순히 빠르게 처리하는 것으로는 “제대로 해결”하지 못하는 경우가 많다. 빠르게 처리하는 것이 반드시 최적화된 것이라고는 볼 수 없는 경우가 많으며, 빠름에 대한 큰 대가를 지불해야 하는 경우도 종종 있다.

컴파일러를 이용한 최적화에는 두 가지를 한꺼번에 해결할 수는 없다. 각각 시간(Time, 속도) 및 공간(Space, 크기)의 최적화이며, 한 번에 두 가지를 다 만족시키지 못한다. 여기에 한 가지 더 추가하자면, 복잡함을 다루는 기술을 얻지 않는다면, 유지보수에 대한 심각한 문제를 안고 출시해야 한다 점이다. 현실적으로 우리가 다루는 문제 만이 아니고, 그것을 해결하기 위한 코드 자체도 복잡해져 간다는 의미다. 복잡한 코드를 만들게 되면 추가적인 유지보수 비용도 점차 커지게 된다는 점이다. 따라서, 최적화된 코드를 만들기 위해서는 컴파일러의 최적화 옵션과 더불어 유지보수에 필요한 비용도 계산에 포함해야 할 것이다.

하지만, 실제 개발에서는 여유라는 것을 거의 찾아보기 어렵다. 한 가지 일이 끝나면 그다음 일이 이미 기다리고 있거나, 벌써 시작되었을 것이 분명하다. 슬랙(Slack)은 그럼에도 불구하고 여분의 자원과 시간을 만드는 것이다. 여유가 없으면 개선할 수 있는 시간도 없으며, 새로운 것을 배워서 적용할 수 있는 가능성도 없어지게 된다. 개발하는 중에 문서를 작성할 시간도 없으며, 교육을 위한 시간도 가질 수 없을 것이다. 개발하는 과정에서 나오는 문서는 생명력이 있지만, 개발이 다 끝난 후에 나오는 문서는 이미 죽어서 더는 사용되지 않을 가능성이 높다. 따라서, 이런 것들을 하기 위해서도 슬랙(여유)은 항상 필요하다.

여백이 없는 코드는 보기가 어렵다. 여백이 너무 많은 코드도 보기가 어렵기는 마찬가지다. 적절한 여백을 가지고 작성된 코드는 그 자체로 중요한 힘을 가지고 있으며, 유지 보수를 하는 사람들에게 큰 도움을 준다. 읽기 편해지고 이해하기 쉬워지기 때문이다. 삶의 여백도 마찬가지다. 지나치게 부족한 여백은 시간의 흐름에 따라 점점 더 줄어들게 된다. 바쁜 삶을 이어가다 보면, 정작 중요한 것들은 잊어버리고 무엇을 하는지도 모르고 살아가게 된다. 적절한 삶의 여백은 우리가 소프트웨어 개발자로서 살아갈 수 있는 힘을 찾도록 도와주며, 개선을 위한 새로운 지식을 익힐 수 있는 기회를 제공하기도 한다.

현실은 더 많은 일을 한꺼번에 하기를 요구하며, 바쁘지만 더 좋은 수준의 결과물을 만들도록 다그친다. 하지만, 복잡함을 다루어야 하는 우수한 소프트웨어 개발자도 사람이다. 그 사람들은 100m 경기를 하기 위해서 회사를 오는 것이 아니라, 마라톤을 뛰기 위해서 오늘도 한걸음 한걸음 조금씩 앞으로 나아간다. 동시에 여러가지 일을 하면 각각의 일에 대해서는 품질이 낮아질 수밖에 없으며, 컨텍스트 스위칭 오버헤드가 전혀 없는 것은 아니다. 새로운 것을 하기 위해서는 준비하는 시간이 필요하며, 지난 실행해서 남겨두었던 결과를 다시 정리할 시간도 필요하다.

자신의 능력 이상으로 주어진 일에 대해서는 아무런 책임감을 느낄 필요가 없다. 그것은 일을 맡긴 사람들의 잘못이며, 결코 일을 실제로 하는 사람들의 잘못은 아니다. 언제 끝날지도 모르는 경주를 매번 단거리 육상 선수처럼 뛸 수는 없다. 그렇게 뛰다가는 선수가 사라지거나, 경기가 끝나버릴 수도 있다. 책임은 자신이 정한 방식으로 집중해서 일할 때 생기는 것이지, 누군가가 지시한 방법으로 누군가를 대신해서 일하는 것으로는 가질 수 없다. 갑자기 주어지고, 아무렇게나 치고 들어오는 모든 일을 우선순위를 높여서 처리하는 것은 불가능하다. 그런 것이 가능하기 위해서는 멀티 코어를 활용하는 수밖에 없지만, 그것도 실제로는 효율저하가 생길 수밖에 없다. 더군다나, 사람은 절대 멀티 태스킹(Multi-Tasking)에 익숙해질 수 없다는 점도 알아야 할 것이다.

중간 관리자에게 가장 필요한 것은 "기다림"이다. 참고 기다리는 것이 어렵다는 것은 누구나 다 아는 사실이지 만, 위로부터의 압박과 아래로 부터의 고민 속에서 균형을 유지하는 것은 상당한 인내를 요구한다. 하지만, 기다려야 한다. 계획이 없는 기다림이 아닌, 계획적인 기다림을 해야 한다. 가고자 하는 길에 대한 큰 그림을 그리고, 차근차근 하나씩 움직여 나가야 한다. 물론, 혼자 만의 생각이 아닌 모두가 이해하고 동의할 수 있는 그림이 필요하다. 가능한 작은 변경을 자주 가져가며, 변경 후에는 변경 전과 무엇이 달라졌는지를 항상 파악해야 한다. 큰 변화는 상당한 시간의 검증을 요구하기에 기다림에 지친 사람들에게는 혼돈 만 줄 것이다. 결과적으로 즉시 파악할 수 있는 수준의 변경을 자주 가져가야 하며, 그것으로 신뢰를 쌓아갈 수 있는 길 만들어야 할 것이다.

아이들은 항상 노는 것을 가장 좋아한다. 자신이 재미있어 하는 것을 할 때는 시간이 없다고 이야기 하면서 좀 더 할 수 없는지 부모들에게 물어본다. 아마도 이것은 모든 사람에게 공통적일 것이다. 즉, 자신이 관심 있고 흥미를 느끼는 일에 대해서는 어떤 식으로든 더 많은 시간을 보내고자 노력하는 것이다. 본능에 가까운 이러한 성향은 일에 대해서도 마찬가지다. 자신이 하고 싶은 일과 하기 싫은 일을 구분하고, 때로는 하기 싫어하는 일도 어느 정도 해야 만 자신이 하고싶은 일을 할 기회를 얻을 수 있다는 것을 알게 해 준다. 아이들과 다르게 성인은 “인내”를 조금 더 많이 할 수 있다는 차이가 있을 뿐이다.

하지만, 하고싶어서 하는 일에 대해서도 어떤 상황이 닥치게 되면 스트레스라는 것을 알게 되고, 점점 더 멀어지고자 할 때도 종종 있다. 즉, 자기 일에 대해서 가지는 흥미를 외적인 요인에 의해서 강요받을 때 생기는 현상이다. "하던 일도 멍석 깔면 안 한다."라는 말이 있듯이, 누군가의 강요나 일정에 대한 압박, 혹은 지나친 간섭으로 몰입(Flow)이 어려워진다. 필요한 것은 스스로 결정할 수 있는 권한과 간섭받지 않고 책임질 수 있는 자유다. 물론, 이런 것들이 회사에서 가능하냐는 질문이 있을 수 있다. 물론, 가능하다. 종료 조건에 대한 명확한 정의와 일이 가진 의미를 설득할 수 있다면, 하는 방법이나 일정에 대해서는 개발자가 스스로 충분히 조절할 수 있다.

관리자나 개발자를 만나서 이야기하다 보면, 가장 많이 듣는 말이 "시간이 없다."라는 것이다. 정말 시간이 없는지 다시 물어봤을 때 나오는 대답은, "마음의 여유"가 없다는 말이다. 개발자건 관리자건 상관없이 다들 자신의 일에 대해서 진지하게 생각하고, 잠시도 떠나 있을 수 없다는 것이 바로 그런 대답이 나오는 이유다. 물리적인 시간의 여유도 없지만 정신적인 여유 마져도 부족하다는 뜻이다.

"시간"은 물리적인 것이다. 특정 시간에 어떤 일을 하게 되면, 그 시간에 다른 일은 할 수 없게 된다. 어떤 사람은 과제를 기획할 때, MM(Man Month: 월단위 인력 투입량)를 따져서, 0.5, 0.7등등의 MM을 일에 할당한다. 하지만, 사실 어떤 사람의 일부만 과제에 투입되는 것은 불가능하다. 차라리 그냥 1MM을 적어 넣는 것이 현실적이다. 0.1이라고 적어도 그 사람이 해야 할 일은 결코 작은 부분이 아닐 것이다. 오히려 그 일을 더 잘하게 하기 위해서는 온전한 "1MM"을 지켜주는 것이 관리자의 의무일 것이다. 소수점 이하로 나오는 MM은 상상속에서나 가능하다.

그렇다면, 왜 그렇게 하지 못할까? 이유는 일 할 수 있는 사람이 부족하다는 것이다. 즉, 믿고 맡길 수 있을 만한 사람이 없다는 뜻이다. 실력이 부족한 사람도 있고, 실력은 좋으나 혼자서 만 일하는 사람이 있고, 혹은 누군가에게 지속해서 지도를 받아야 일할 수 있는 사람도 있다. 어떤 식으로든 해당 분야를 처음 하게 되면, 도메인 지식(Domain Knowledge)를 갖추는데 필요한 시간이 있으며, 그것을 빠르게 따라잡기 위해서는 그 만큼 더 많은 노력이 필요하다. 하지만, 과제 계획은 개발자의 실력에 상관없이 1MM은 한 사람이 해야 할 일일 뿐이다. 사람을 배치하고 나면 정해진 시간내에 일이 완료되기를 기대하게 된다. 이것은 근거없는 믿음이며, 과도한 낙관일 뿐이다.

따라서, 중요한 것은 "1MM"이 어느 정도의 역량이 필요한 일인지 파악하는 것이다. 그리고, 소수점 이하로는 나누지 않아야 한다. 사람을 쪼개는 것이 불가능 하듯, 일도 당연히 쪼개서 맡길 수 없다. 어떤 일은 평범한 수준의 개발자가 1MM이 필요하지만, 능력있는 개발자라면 소수점 이하라도 가능할 것처럼 보일 수도 있다. 하지만, 실제로 그 일에 사람을 투입하고, 투입된 사람이 다른 일까지 동시에 진행하고 있는 상황이라면, 정해진 날짜보다 계획은 항상 지연될 것이다. 예를 들어, "0.3, 0.2, 0.5MM"으로 세 개의 일을 동시에 하라고 하면, 세 개의 일을 순차적으로 각각 진행할 때 보다 더 많은 시간이 걸릴 것이다. 결과물의 수준도 기대한 것에 미치지 못할 가능성이 크다.

학습과 가르침, 숙고의 시간이라는 여백을 만들어주지 않는다면, 제대로된 인재를 키워내기 힘들다. 일정의 압박만 가지고선 과제를 제대로 관리할 수 없으며, 개발자 개인의 역량을 알지 못하는 상황에서는 의미없는 MM만 따져서도 안된다. 질책과 다그침으로 지금 당장의 성과는 만들어 낼 수 있을지 몰라도, 지속적으로 유지하고 관리할 수 있는 역량있는 사람으로 성장시키지는 못한다. 오히려 인재를 키워나갈 수 있는 기초를 없앨 뿐이다. 일은 쪼개서 관리할 수 있지만, 사람은 온전히 하나로 관리해야 한다. 그 사람의 일을 관리하고자 한다면, 먼저 성숙한 인간대 인간으로 “존중”해 주어야 할 것이다. 성장을 위해서는 가르침도 필요하지만, 스스로 이해하고 깨달을 수 있는 시간적인 “여유”도 중요하다.

출근부터 정해진 시간에 떠밀리고 졸음에 겨워 도착하는 회사에서, 사람들은 삶의 여유를 찾을 수 있을까? 삶에 대한 여유로운 마음이 생기는 시간은 “조용히 시작할 수 있는 아침”과 “가족들과 대화할 수 있는 저녁 식사”에 있다. 이 두 가지만 제대로 된다면, 아마도 지금보다는 더 창조적인 것들로 하루 하루가 채워질 수 있을 것이다. 하지만, 우리의 삶이 유한하다는 것과 회사는 언제나 풍족한 자원을 보장해 주지 않는다는 점에서 우리의 기대는 항상 낮아질 수밖에 없다. 그것이 현실이며 우리가 생활하고 느끼는 한정된 공간이다.

회사는 왜 항상 부족한 상황을 만들까? 사실 모든 것이 풍족한 회사란 없다. 한정된 자원과 시간, 정해진 납품 일자에 맞춰서 일 할 수밖에 없다. 자본주의 자체가 부족한 것을 기반으로 더 많은 성과를 이루어 내기를 기대하기 때문이다. 또, 그렇게 이루어 낸 결과를 찬양하는 분위기도 한 몫하기도 한다. 회사에서 발간하거나 자체 방송의 뉴스에 나오는 것들만 봐도, "그럼에도 불구하고"라는 단어를 쓰면서 힘들었던 과거를 이겨낸 사례를 자랑스럽게 이야기한다. 모든 것이 다 갖추어진 상태는 당연히 성공할 수밖에 없는 결과를 가져온다고 가정하고 있는 것이다. 하지만, 정말 그럴까? 우리가 이겨낸 것이 불필요한 불편이라도 그렇게 이야기할 수 있을까? 야근을 밥 먹듯이 해서 돌아온 결과가 과연 그렇게 좋은 결과일까?

아마도 정당한 근무 수당을 지급해야 한다면, 그런 이야기는 하지 않을지도 모른다. 많은 시간을 노력해서 이루어진 결과라는 말을 뒤집어 보면, 아직 우리가 하는 일이 부가가치가 높지 않다는 뜻이기도 하다. 비용은 많이들고 이익이 적다면, 우리가 하고 있는 일에 대해서 다시 한번 생각해 봐야 할 때가 왔다는 신호다. 불합리함을 극복하기 위해서 낭비되는 시간이 많다면, 오히려 낭비를 줄이는 것이 더 효과적인 일 처리 방법일지도 모른다. 그냥 무작정 열심히 해서 이루어질 성과라면, 이미 다른 사람들도 그렇게 했을 가능성이 더 높다. 따라서, 정말 가치가 높은 일이라면 노력의 질도 달라져야 할 것이다. 단순히 열심히 하는 것으로는 도달 가능한 수준이 아니어야 한다.

불합리한 불편에는 어떤 것들이 있을까? 실행에 집중하라는 이야기는 다들 들어봤을 것이다. 개발에서 말하는 실행이란 개발 자체에 시간을 더 많이 쓰라는 이야기다. 따라서, 개발 자체의 활동이 아닌 것을 없애라는 말과 같다. 개발 외적인 것들에는 어떤 것들이 있을까? 아마도 대부분 개발자들 머리 속에는 불편한 진실이 한가지 떠오를 것이다. 즉, "도움이 안 되는 회의" 및 "쓸데없는 숙제", “불편을 초래하는 각종 회사의 시스템들”이다. 실제로 주간회의가 가장 걸림돌이 된다는 사람들도 있다. 주간회의의 대부분은 정보 공유라는 명목으로 모두 참석하기를 요구한다. 쓸데없는 숙제의 대부분은 개발을 모르는 관리자에 의해서 주어지고, 불편을 초래하는 시스템은 개발과 관련 없는 사람들이 관리 목적으로 만 사용한다.

가장 힘든 부분은 소위 "쓸데없는 말 한마디"이다. 개발자들은 기볹거으로 자신이 해야 할 일을 잘 알고 있다. 일정이 부족하다고 생각하면 자연스럽게 야근이나 주말 특근을 자발적으로 한다. 하지만, 관리자가 "몇 시까지 출근해서 몇 시까지 일해"라고 말하거나, "이번 주말을 모두 나와서 일해"라고 한다면, 그 순간은 넘어갈지 몰라도 개발자는 조금씩 저항하려는 마음을 쌓아두게 된다. 신뢰는 서로에 대한 믿음을 바탕으로 한다. 출퇴근 시간 조차도 서로 신뢰하지 못하는 관리자라면, 이미 서로에 대한 믿음은 극히 약할 것이다. 사실, 우리나라의 개발자 대부분은 책임감이 강하며, 개인보다 회사나 조직을 생각하는 마음이 강한 편이다. 물론, 모든 사람이 그렇지는 않겠지만, 대다수의 개발자는 자신이 맡은 일에 대해서 자신과 동일시하는 경향이 있다.

삶의 여유를 가지기 위해서는 어떤 것들이 필요할까? 아마도 대부분 개발자들은 제대로 퇴근 만할 수 있어도 대단히 만족한 삶이라고 생각할 것이다. 그렇다면 현실에서 찾을 수 있는 해결책은 정시 퇴근을 할 수 있는 방법을 찾아야 할 것이다. 낮에 일하고 밤에는 자신의 동굴로 들어가서 안전한 곳에서 쉬는 것은 인간이 태초부터 이어온 생활 방식이다. 이것을 어기는 것은 인간의 가장 오래된 전통을 무시하는 일이며, 쉬지 않고 일 한다고 해서 생산적인 것도 아니다. 피곤하면 실수하게 되고, 실수하면 재작업을 할 수밖에 없다. 출퇴근을 정시에 하기 위해서는 실무에서 낭비되는 시간을 줄이고, 반복적인 작업은 자동화를 통해서 해결해야 하며, 재작업을 줄이는 예방책을 만드는 길 밖에 없다.

학교 다닐 때를 생각해 보자. 가장 공부에 집중이 잘 될 때가 언제인가? 아마도 정신적인 긴장이 풀린 아침 시간이나 운동을 하고 약간 신체적인 긴장이 완화된 때가 아닐까? 사람은 휴식이라고 생각되는 것을 취하고 난 이후에 제대로 일에 집중할 수 있다. 휴식이 없다면 항상 긴장 상태가 되고, 긴장 상태의 지속은 결국 신체적 정신적인 건강을 망치고 만다. 오래 뛰어야 할 경주에서 빨리 낙오자가 되고 만다. 우리가 하는 일은 단거리 경주가 아니다. 천천히 자신의 컨디션에 맞춰서 준비하고, 페이스를 조절해서 결승점까지 뛰어가야 한다. 그리고, 결승점에 도달했다고 해서 끝나는 경주도 아니다.

관리자는 항상 긴장할 것을 요구한다. 지속해서 근무 강도를 높이는 것이 아니라, 처음부터 목표를 정한 근무 강도를 지키기를 요구한다. 만약 3~6개월짜리의 과제라면 가능할지도 모르지만, 1년에서 2년이 걸리는 과제를 그런 식으로 하는 것은 상상하기 싫다. 과제 중간에 사람들이 뛰쳐나가는 것도 충분히 이해할 만한 행동이다. 긴장이 오랜 시간 지속되면 사람 사이의 관계도 나빠지고, 의견충돌도 자주 발생하게 된다. 팀워크를 첫 번째로 해야 할 팀이 제대로 된 협력은 커녕, 서로 반목과 보이지 않는 벽을 쌓아가는 상황이 된다. 오히려 관리자라면 항상 긴장할 것을 요구할 것이 아니라, 이미 쌓인 긴장을 푸는데 더 많은 노력을 해야 한다.

삶의 여유는 창조적인 활동의 원천이기에 반드시 관리자가 지켜주어야 할 부분이다. 자신의 시간을 가지지 못하면 실력을 향상시키는 것도 어렵다. 새로운 것들에 대해서 관심이 멀어지기에, 최신 기술은 어느덧 남의 이야기가 되고 자신은 뒤처져 간다는 느낌만 받을 뿐이다. 인위적으로 팀당 몇 건의 세미나를 하라고 지시하기 보다는 자발적으로 공부하는 분위기가 되도록 만들어 주어야 한다. 개발자는 자신이 개발하는 과제뿐만이 아니라 자기 개발에도 높은 관심을 가지고 있기 때문이다. 오래 동안 관리자로 남은 사람들이 개발자와 멀어지는 것은, 개발의 현장에서 벗어나 개발자에게 더 이상 긍정적인 입력(Input)을 주지 못하기 때문이다. 자신도 그런 여유를 가진 적이 없었기에 남에게 가르쳐줄 것도 없기 때문이다.

삶의 여유는 작은 말 한마디에서 시작하고 세심한 돌보기에서 나온다. 자신이 관리하는 사람들은 보호를 받는 존재가 아니라, 오히려 그들을 통해서 나를 되돌아봐야 한다. 그들이 부족한 부분이 무엇인지를 고민해서 미리 준비해야 하고, 꾸준한 성과를 낼 수 있도록 개발자 각자가 갖추어야 할 부분을 이야기해 주어야 한다. 새로운 것들을 접하기 위해서 교육을 보내는 것도 중요하다. 가족보다 더 오랜 시간을 같이 해야하는 사람들이기에, 그들의 성장이 나의 발전이라는 것을 항상 마음에 두어야 한다. 잘한 것에 대한 칭찬도 해야 하며, 잘하지 못한 부분에 대한 따끔한 말 한마디도 “진심”으로 해야 한다. 사람의 마음을 얻는데는 시간과 노력이 필요하지만, 잃는 것은 순식간이다. 인생에 모든 중요한 것들은 이렇게 얻기는 어렵지만 잃는 것은 쉬운 것들임을 항상 기억해야 할 것이다.

# [ 교육의 목적 ]

간혹 회사 일이 잘 안풀릴 때 개발자들을 대상으로 설문 조사를 하면, "개발자의 전문성을 키우기 위해서 교육이 필요하다”고 결론이 나는 경우가 종종 있다. 문제는 이러한 교육이 강압적으로 시행되어 정말 필요한 것과는 전혀 상관없는 방향으로 소프트웨어 개발자에게 돌아온다는 것이다. 그럼 필요한 교육은 무엇이고, 필요하지 않은 교육은 무엇일까? 구분은 간단하다. 개발의 원칙(Principle)을 세우기 위한 교육인지, 아니면 단순한 기능 교육인지로 판단할 수 있다. 원칙을 세우기 위한 교육은 언제나 도움이 되지만, 단순한 기술이나 기능을 익히기 위한 교육은 지금 당장의 문제를 해결해 줄지 몰라도, 장기적인 관점에서는 크게 도움이 안된다.

원칙을 세우기 위한 교육은 다음과 같은 것을 포함하고 있어야 한다. 첫 째로, 일을 하는 기준을 명확히 제시할 수 있어야 한다. 왜 일을 해야 하는 지를 알지 못한다면, 일해도 성과를 평가하는 기준이 없다. 따라서, 일을 잘했는지 못했는지를조차 모르게 된다. 둘 째로, 일하는 방법을 구체적으로 전달해야 한다. 그냥 막연히 구호만 외치거나 혹은 남들이 해봤다고 하는 이야기만 전달해서는 안 된다. 업무를 수행하는데 구체성을 가지고 전달되어야만 실제 업무에서 적용해 볼 수 있다. 세 번째는 일의 성과를 측정할 수 있어야 한다. 일을 수행하는 기준과 일을 하는 방법을 배웠다면, 그것으로 인해서 달라지는 결과를 측정할 수 있어야 한다. 잘했는지 못했는지를 알기 위해서는 기준을 정하고, 일을 구체적으로 수행하고, 측정을 통해서 개선할 수 있어야 한다.

그렇다면 소프트웨어 개발자로서 필요한 교육에는 어떤 것들이 있을까? 모든 교육이 그렇듯 반드시 지켜져야 할 원칙을 합의해서 공유하는 것이다. 즉, 코딩 룰, 체크인(Check-in) 룰, 코드 리뷰 방법, 문서화, 단위 테스트, 통합 테스트, 자동 빌드, 정적분석 결과 개선 방안 등등 다양한 것들이 있다. 이런 것들의 특징은 업무의 가장 기본이라는 것이다. 또한, 이와 더불어 구체적인 가이드(Guide)를 제공주면 더 좋을 것이다. 일관성만 가지고 있다면, 어떤 가이드 라인이라도 상관없다. 양이 중요한 것이 아니고 준수 여부가 중요하며, 이를 통해서 모든 구성원이 일관성을 가지게 될 때 “공유하는 문화"라는 것이 생겨나기 때문이다.

측정이 필요하다면 오픈소스나 상업용으로 이용 가능한 다양한 툴들을 찾아보기 바란다. "Code Sonar"나 “Coverity”에서 제공하는 도구들도 코드의 품질에 대한 객관적인 측정방법으로 사용할 수 있다. 또한, 자동으로 테스트되는 코드의 범위를 척정하는 것도 개발 중에 중요한 품질 평가 기준이 될 수 있다. 처음에는 측정되는 수치가 나쁠 수 있기에 주의해서 사용해야 한다. 개발자들은 자신의 코드 품질이 나쁘다는 말을 가장 싫어한다. 힘들게 일하는데 비난을 듣는 것과 같다고 생각할 것이 분명하다. 따라서, 반드시 “개선의 도구”로 사용해야만 하며, 절대 “평가의 도구”로 활용해선 안될 것이다. 같은 도구라고 하더라도 사용하는 방법에 따라 사람에게 상처를 줄 수도 있기 때문이다.

중요한 것은 "원칙을 가지고 일하라"는 것이다. 원칙이 없는 것보다는 기준을 가지고 일을 했을 때, 성과를 눈으로 확인하고자 하는 사람들에게 명확한 근거를 제시할 수 있다. 소프트웨어 개발자가 일하는 환경은 언제나 쉽지 않은 환경이고, 보여줄 수 있는 것들을 많이 가질수록 성과에 목말라 하는 이해 관계자들에게 정확한 과제의 현실을 파악할 수 있도록 해 주기 때문이다. 진정한 교육이라면 10년 20년을 일 할 수 있는 도구를 제공하는 것이며, 제대로 일하는 것을 모르는 개발자들은 지속적인 실수를 할 수밖에 없다. 무엇이 옳고 그름의 문제가 아니라 미래를 어떻게 변화시킬 수 있는지에 관심을 갖도록 만들어 주어야 교육의 진정한 목적을 달성할 수 있는 것이다. 개발자는 스스로 발전하기를 원하지 정체되고 싶어하지 않는다.

# [ 지식의 실체 ]

프로그래밍을 배울 때 가장 빨리 배우는 방법은 무엇일까? 책을 열심히 보는 것도 한 가지 방법이지만, 직접 키보드를 두드려 예제를 실행시켜보는 것이 가장 효과적이다. 왜 그럴까? 이론과 실제는 다르기 때문이다. 하지만, 여기서 이야기 하고싶은 것은 교과서가 맞지 않는다는 뜻이 아니라, 교과서로 배운 것도 반드시 직접 손으로 만들어 봐야지 더 정확하게 이해할 수 있다는 것이다.

회사에서 흔히 능력 있다고 여겨지는 인력들은 자신의 코드에 대한 분석 결과를 잘 받아들이려고 하지 않는다. 문제의 원인이 무엇인지 이야기를 하면, 교과서에 나오는 이야기만 한다고 핀잔을 준다. 심한 경우에는 “직접 해봤나요?”라고 하면서 반감을 드러낸다. 그러면서 하는 이야기는 실제로는 자신의 방빕이 더 좋고 전혀 문제가 없다는 듯이 말 한다. 어찌보면 몸에 이미 익숙해진 옷을 벗어보라는 요구에 창피함을 느끼는지도 모르겠지만, 절대 자신이 작성한 코드에 대해서는 남들의 의견을 받아들이지 않으려는 태도를 보인다. 자신의 코드가 자신의 것이 아니라는 것을 모르고, 마치 회사에서 작성한 것이 자신의 것인 것처럼 소유권을 주장하는 경우도 있다. 자신과 코드를 동일시하는 것이다.

중요한 것은 자신이 코딩했다고 하더라도 자신의 코드가 아니라는 것을 인식해야 한다는 점이다. 물론, 자신의 코드에 대한 애착을 버리라고 하는 말은 아니다. 자신이 만든 코드 이기에 좀 더 가치 있게 만들 수 있도록 고민하라는 말이다. 남이 절대 고쳐서는 안 되는 절대 불변한 것이 아닌, 많은 사람과 소통하고 개선해 나가는 코드를 만들어 나가라는 것이다. 예를 들어, 오픈 소스 공동체에서는 자신들이 만든 코드를 공개하고, 서로의 의견을 반영해 더 많은 사람들에게 더 많은 혜택을 주려고 노력하고 있다. 비판이 아닌 개선을 추구하고 있는 것이다. 더 많은 사람이 검토할 수록 더 많은 개선이 가능하다. 지식이란 혼자서 옳다고 우기는 것이 아니라, 비판을 통해서 성숙해진다는 것이다.

지식이란 머리 속에만 있는 것이 아니라 실체화가 되어야 하고, 또한 사람들과 공유를 통해서 더 발전해 나가야 한다. 실체화된 것을 경험하지 못하면 공허한 지식일 뿐이다. 또한, 실체화되었다고 모두 옳은 것도 아니다. 따라서, 이론과 실체는 함께 병행해야 서로가 서로를 지탱해주는 구조를 만들 수 있다. 또한, 지식은 자신의 것이라는 생각을 버리지 못하면 결국 그로 인한 피해가 자신만이 아니라 모두의 것이 된다. 공유하지 않는다면 누군가의 소유만 주장하게 될 것이고, 그것을 흔쾌히 인정해 줄 동료나 선후배는 찾을 수 없게 된다. 사람은 누구나 인정받고 싶어하지만 상대를 인정해 주지 않는다면 자신도 인정받지 못한다. 다른 사람의 기여를 인정하지 않는다면 자신의 기여도 인정받지 못하는 것이다.

# [ 팀워크 ]

과제를 성공으로 이끄는 일 중에서 가장 시간이 오래 걸리면서도 가장 쉽게 실패할 수 있는 것이 팀워크이다. "팀으로 일하는 것"은 정말 쉽지 않다. 큰 노력에도 불구하고 아무런 성과가 없을 수도 있고, 적은 노력에도 불구하고 탁월한 성과를 보이기도 한다. 팀은 직급으로 일하지 않는다. 알게 모르게 내부적으로 서로가 동의한 조직 체계를 가지고 있으며, 기술적인 역량이나 혹은 리더쉽(Leadership) 등 보이지 않는 것들이 토대를 만든다. 팀워크가 깨진 팀은 서로 간의 신뢰가 없으며, 소통없이 그냥 직급에 따른 업무의 일방적인 전달만이 있을 뿐이다.

자신의 팀에 새로운 사람을 뽑야야 할 때 주로 어떤 요소를 보겠는가? 가장 기본적인 것은 그 사람의 업무 역량을 보는 것이다. 하지만, 업무 역량은 그 사람의 개인에 속하는 부분이고, 업무에 필요한 충분한 지식만 보는 것이다. 기본적인 역량이 안된 사람을 뽑는 것은 그 팀이 새로운 인력에 대해서 충분히 시간을 가지고 가르칠 여유가 있을 때 가능하다. 대부분의 신입 사원이 아마도 이런 부류에 속하지 않을까? 하지만, 이런 경우라도 기본적인 자세(Attitude)를 보고 뽑는 것이 일반적이다. 특히, 임원들의 경우에는 이미 실무에서 멀어졌고, 실제로도 실무를 모르기에 사람의 자세가 가장 중요한 기본 역량이라고 볼 것이다.

과제 관리자는 다양한 사람들이 한 목소리를 내도록 조율하는 지휘자와 같다. 자신의 몸짓에 아무런 목소리가 나오지 않는다면(혹은, 그냥 "네, 알겠습니다." 나 자신의 의견이 없이 "그렇게 하겠습니다."와 같은 단편적인 대답이 나온다면), 이미 팀 내부에 불협화음이 나오고 있는 뜻이다. 지휘자의 말을 듣지 않는 합창단이 어떻게 노래를 제대로 부를 수 있겠는가? 그럼 처음부터 모든 사람이 그런 태도로 과제 관리자와 이야기했을까? 아마도 그렇지는 않을 것이다. 처음에는 이런 저런 자신들의 의견을 솔직하게 이야기했을 것이다. 하지만, 한 번 두 번 쌓이는 실망감이 어느새 팀을 지배하게 되고, 자신의 목소리를 내는 것이 의미가 없다고 판단하게 되는 순간, 그냥 물러나 타인의 입장에서 시키는 대로 만할 뿐이다. 물론 시키지 않는 일이나, 자신과 상관없는 일은 절대 하지 않을 것이다. 시키는 일도 최대한 늦게 마치거나 열정없이 적당히 하고 말 것이다. 그럴수록 관리자는 더 많은 손짓 발짓을 할 것이고, 빈익빈 부익부와 같이 안되는 집안은 안 되는 일 만 가득 차 버릴 것이다.

중요한 것은 존중(Respect)이다. 팀원에 대한 존중이 없다면 신뢰는 먼 나라의 이야기가 되어 버린다. 자신과 같이 일하는 사람들을 소중히 생각하지 않는 사람이 어떻게 그 사람들의 믿음을 끌어낼 수 있겠는가? 개인기가 탁월한 선수 한 명이 아무리 열심히 해도, 축구는 11명이 뛰어야 골을 만들어낼 수 있다. 팀워크는 팀원 한 사람 한 사람을 다 소중하게 여기고 바라보는 것이다. "Respect"은 “Re(again: 다시)”+”spect(look: 본다)"라는 의미이고, "존중(尊重)"은 "높이어 귀하게 본다"라는 의미이다. 즉, 한 번이라도 더 봐주고 높여서 대하는데, 어느 누가 자신의 능력을 100% 발휘하지 않겠는가? 결국 팀원이 아니라 팀원을 관리하는 사람의 태도가 성과 차이를 만들어 내는 것이다.

하위 관리자를 쓰려고 한다면 그 사람에 대한 아래 사람들의 태도를 보기를 바란다. 자신에게 좋은 말만 하는 하위 관리자가 예쁘게 보일지는 몰라도, 당신의 팀은 내부에서 곪아가고 있을지도 모른다. 아래 사람의 목소리에 귀 기울이지 않는다면 잘못된 정보로 오판할 가능성이 높다. 일은 그 일을 직접 수행하는 사람이 가장 잘 안다. 그런 사람들의 의견을 무시한다면 어떻게 제대로 된 판단 근거를 찾아낼 수 있겠는가? 자신이 현재 정말 팀을 잘 관리하고 있다고 생각할수록, 더 열심히 팀의 목소리에 귀를 기울여야 한다. 혹시 자신만의 세상을 만들고 환상 속에서 살고 있다면, 깨지는 순간은 갑작스럽게 다가올 것이다. 과제를 잘 한 것은 팀원들의 몫이 되고, 잘못은 관리자의 것으로 받아들여야 한다. 결국, 결과물은 팀원이 만들지만 그런 팀원들이 일을 잘 할 수 있도록 환경을 만들어 주는 것은 관리자의 몫이다. 따라서, 잘못된 결과가 나왔다면 관리자가 일을 제대로 할 수 있는 환경을 만들지 못했다는 뜻이 되기 때문이다.

아쉽게도 자신의 잘못을 팀원에게 돌리는 관리자들도 있다. 신뢰가 없는 팀을 만든 탓을 팀원들에게만 하고 있는 것이다. 자신은 잘 하고 있는데 팀원들이 따르지 않는다고 이야기하기도 한다. 특정한 일이 잘 안되면 그 일을 담당하고 있는 팀원의 이름을 거론하거나, 다른 팀에게 이유를 돌리기도 한다. 아무리 좋은 팀이라고 하더라도 이런 일을 겪고나면 팀워크는 사라질 수밖에 없다. 비록 자신의 힘으로는 어쩔 수 없다 하더라도 책임은 항상 관리자의 몫이어야 한다. 또한, 불만 가득한 팀원의 목소리를 들어주는 일도 관리자 만이 할 수 있는 일이다. 결론을 이야기해서 결정만 하는 자리가 아니라, 더 많은 목소리를 듣어주는 것이 관리자의 진정한 일이다.

# [ 매크로 관리 vs. 마이크로 관리 ]

관리자는 크게 두 가지 부류가 있다고 하겠다. 매크로(Macro)와 마이크로(Micro) 관리자가 바로 그런 사람들이다. 구분의 기준은 사람을 어떤 관점으로 볼 것 인가로 나누어 진다. 자율성을 보장하고 결과를 중시해서 보는 사람과 세밀하게 하나 하나를 짚어가며 일일이 지시하는 사람이 있다. 둘 다 장단점이 있으며 어느 것이 더 좋고 나쁘냐는 중요한 것이 아니다. 각각은 필요에 의해서 생겨났으며, 개인의 오랜 경험에서 형성된 습관일 뿐이다. 또한, 자신만이 가지고 있는 관리에 대한 “노하우”처럼 생각될 수도 있다.

하지만, 반대로 관점을 바꿔서 생각해보자. 관리를 당하는 사람의 입장에서는 지나치게 적은 관심도 힘들지만 너무 세세한 관심도 부담스럽기는 마찬가지다. 필요한 만큼의 관심을 두기를 바랄 것이다. 하지만, "필요한 만큼의 관심"이란 정말 힘든 말이 아닐까? 그럼 어느 정도의 관심이 개발자 및 관리자의 입장에서 용인될 수 있는 수준일까? 답은 단순하다. 소통이 원활하게 잘 된다고 느끼는 수준에서 멈춰야 한다. 모든 관심의 핵심은 상대를 존중한다는 생각이 바탕이 되어야 하며, 진정어린 신뢰를 가지고 대해야 잔소리가 아니는 충고로 받아들일 수 있다.

대화는 한 쪽에서 다른 한 쪽으로 일방적으로 전달되는 연설이 아니다. 하지만, 대부분 관리자는 연설에 익숙하고, 남들도 자기처럼 생각한다는 착각에 빠지기 쉽다. 말이 너무 없으면 상대방이 자신의 요구를 잘 이해하고 정확히 수행할 것이라는 “근거 없는 믿음”을 갖게 된다. 너무 많은 지시 사항은 중요한 일의 우선순위를 흐트려, 무슨 일을 먼저 해야 할지 모르게 만든다. 이런 식의 대화라면 직접 일을 해야 하는 사람을 "맨붕"에 빠뜨리게 된다. 말을 많이 하면 일관성이 약해지게 되며, 잘못된 말을 할 가능성도 높아지게 된다. 이미 일어난 사실을 이야기 할 때도 과장이나 지나친 저평가가 있을 수 있다. 따라서, 대화를 하기 위해서는 자신의 입은 닫고 상대의 말을 더 많이 듣는 것이 “관리자의 대화 방법”이 되는 것이다.

매크로든 마이크로든 상관없이 중요한 것은 개발자를 믿고 신뢰하는 것이다. 세밀하게 업무를 파악하는 것도 중요하지만, 그걸 위해서 추가적인 숙제를 시켜서는 안 된다. 또한, 대략적인 것만 기억하고 정확히 그 업무를 파악하지 못하면, 결국 서로 간에 대화를 단절시키는 벽이 점점 더 높아지게 된다. 따라서, 꾸준한 대화를 통해서 무엇이 실무 개발자에게 필요하고, 최대한 성과를 끌어내기 위해서는 어떤 환경을 만들어 줄지를 묻는다면, 오해 없는 신뢰 관계를 만들기가 한결 수월할 것이다. 보이지 않는 압박이나 분위기를 만들어 상대를 궁지를 모는 것은 옳지 못하다.

관리자든 실무 개발자든 목표는 "일의 완료"에 있기 때문에, 서로 믿지 못하면 아군을 만들 수 있는 사람도 적군으로 돌려버리게 될 것이다. 자신이 매크로 관리자라고 생각한다면, 세부적으로 팀원들이 무슨 일을 하는지 파악하려고 노력해야 할 것이며, 마이크로 관리자라면 말을 줄이고 더 많이 들으려고 해야 한다. 숲과 나무를 동시에 파악할 수 있어야 거대한 산을 제대로 이해할 수 있다. 숲만 바라보면 나무들을 잊게되고, 나무만 바라보면 어디로 가야 할지 알 수 없게 된다. 일은 실무자들이 하지만 결과는 관리자의 책임이다. 제대로 알고 책임지는 사람이 되어야 하며, 세세한 것만 따지다 실패해서도 안될 것이다.

# [ 프로젝트의 실패란? ]

프로젝트의 실패는 단 한 문장으로 정의할 수 있다. "사람을 잃는 것"이다. 팀원이 회사를 떠나는 것이 가장 큰 실패다. 일정이 부러지고 온갖 욕들이 난무해도 사람이 그대로 남는다면, 경험은 언제든 프로젝트를 성공으로 이끌 수 있다는 것을 의미한다. 하지만, 같이 동고동락했던 팀원이 가버리면, 다음이라는 말이 무색하게 처음부터 다시 시작해야 한다. 물론, 실패와 성공은 언제나 동전의 양면이지만, 될 수 있으면 확률이 높은 쪽으로 선택해야 하지 않을까? 그 확률의 첫 번째 고려 요소가 바로 사람이다.

# [ 라이언 일병 구하기 ]

예전에 본 "라이언 일병 구하기"라는 영화에서 여러 명의 목숨을 걸고 한 사람을 살리는 것이 과연 옳은 일인가를 고민해 보자. 모든 사람의 목숨이 똑같이 소중하다는 논리로는 도저히 설명되지 않는다. 사실 이 영화에서 중요한 것은 명분을 찾는 것이다. 소중한 것들을 잃어야 하는 상황에서 그 소중한 것들보다 더 소중한 것을 지키고 있다는 “신념”을 보여주기 위한 것이다. 그것이 가족일 수도 있고, 어린아이의 목숨, 혹은 동료들일 수도 있다.

우리는 과제를 하기 위해서 어떤 것들을 희생하고 있을까? 회사에 다니면서 회의적인 마음이 들지 않은 사람은 없을 것이다. 매일 반복되는 업무에, 주말에도 출근해서 이런저런 잔소리를 들어가면서 일 한다면, 오히려 그런 마음이 들지 않는 것이 이상한 일이다. 하지만 또한 자신만을 위한 변명도 한다. 언젠가 나의 희생이 가족과 동료들에게 도움이 될 것이라는 자기 위안이다. 하지만, 정말 가치 있는 희생일까? 가치가 있다고 느낄 만큼 그런 것들이 희생되는 게 당연한 일일까?

가치를 어디에 두는가는 개인의 선택이지만, 그러한 선택으로 몰아넣도록 강요를 당하고 있다면 무언가 잘못된 것이다. 희생에 대한 대가를 회사에서 보상해 줄 수 있다면 충분히 동의할 수도 있겠지만, 이도 저도 없이 눈치만 봐야 하는 상황이라면 그 가치는 이미 자신과는 상관없는 일이 된다. 내가 다니고 있는 회사가 그만한 가치를 인정하고, 또한 존중하는 분위기라면 당신은 좋은 회사에 다니고 있는 것이다. 만약 그렇지 않고 강요된 희생만을 늘어놓는 관리자가 있다면, 그를 바꾸던가 아니면 그와 상관이 없는 일을 해야 할 때가 된 것이다.

무엇을 희생해서 무엇을 얻어야 한다는 것은 항상 있는 삶의 과정이다. 한 가지를 선택하면 그 만큼의 대가를 지불해야 한다. 하지만, 이 때도 반드시 희생시키지 않아야 할 “신념”은 가져야 한다. “올바름에 대한 기준과 가장 소중한 사람들과 함께 할 수 있는 시간”만큼은 절대 교환할 수 없는 가치가 있다고 생각해야 한다. 어떤 희생을 하더라도 그 두가지만큼은 반드시 지켜 나가야 한다. 그것을 위해서라면 어떤 희생도 의미를 상실하게 된다. 라이언 일병을 구하기 위해서 나섰던 사람들도 자신의 희생을 통해서 이루고 싶어하는 것이 있었으며, 그것만큼은 반드시 지켜내기 위해서 최선을 다했다. 신념은 고통스러운 경험을 동반할지도 모르지만, 그것이 없다면 인생 자체가 의미가 없는 것이 되기 때문이다.

# [ 조직개편 ]

회사는 쉴 새 없이 바뀐다. 바뀌는 이유는 다양하겠지만, 일단 시장이 바뀌기 때문이다. 새로운 관리자가 오면 가장 먼저 하는 일은 자신이 기존에 해오던 방식의 일 처리를 요구하는 것이다. 즉, 이미 자신이 익숙한 것에 대해서 다른 사람들을 적응시키려는 의도다. 모든 사람이 원해서 바뀌기보다는 윗사람이 원해서 바뀌는 경우가 대부분이다. 따라서, 변화는 언제나 위로부터 오는 것이고, 변화된 환경에서 살아가기 위해서는 그것에 적응하는 수밖에 없다.

어떤 상사가 좋은 상사일까? 사실 이것에 대한 정확한 답은 없다. 아래 사람을 위해주는 사람도 있고, 자신의 고집으로 일을 추진하는 사람도 있다. 너무 성격이 좋은 상사는 제대로 결과물을 만들어내는데 서툰 사람도 있고, 자신감으로 똘똘 뭉친 사람은 괴롭더라도 좋은 결과를 만들어내기도 하기 때문이다. 어쩌면, 이런 것에 대한 해답은 그 조직에 몸담은 사람으로서는 절대 판단할 수 없는 일인지도 모른다. 중요한 것은 그런 환경에서도 생존해야 한다는 사실이다.

조직개편은 한 해를 마무리하려는 시점에서 이루어진다. 한 해 동안 이런 저런 일들이 있고, 잘 했든 못했든 간에 뭔가를 했기에 생기는 일이다. 잘 했다면 좀 더 많은 권한이 주어질 것이고, 그렇지 못했다면 다른 일을 찾아서 떠나게 되는 것이 당연하다. 이런 저런 말들이 많이 오가기도 하지만, 정작 중요한 것은 그런 것들에 대해서 아무런 결정도 권한도 없다는 점이다. 따라서, 결과로 주어지는 것은 받아들일 수밖에 없는 일이고, 좀 더 좋은 결과를 내기 위해서는 미리 했어야 했을 일들을 다 했기를 기대할 뿐이다.

사람이 바뀐다고 갑작스럽게 모든 것이 변경되는 것은 아니다. 생각보다 공동체가 이루어온 문화는 잘 사라지지 않는다. 모두가 한꺼번에 바뀌는 것이 불가능하기에 한 사람이 바뀐다고 해서 기존에 있던 사람들이 한 번에 변화를 경험하는 것은 결코 아니다. 하지만, 서서히 가랑비에 속 옷이 젖듯이 스며드는 것이 무섭다. 한번 이야기를 듣고, 다시 조금 시간이 지나서 지속해서 이야기를 들으면, 결국 우리는 내가 바뀌었다는 것을 인식하지 못한 채 변화되어 간다. 마치 당연히 있어야 했던 것들이 있었을 뿐이라는 느낌을 받을지도 모른다.

조직의 개편은 많은 변화를 주려고 하는 시도이지만, 조직의 문화를 쉽게 변화시키지는 못한다. 조직의 문화는 자신이 결정권을 가지고 있지 않은 사람들이 만들어가는 것이고, 이런 문화를 바꾸기에는 인위적인 조작은 가장 마지막에 취할 수 있는 방법일 뿐이다. 중요한 것은 각자의 책임의 범위에서 이루어지는 개선 활동이다. 조금씩이라도 기존의 것을 개선해나가고 있다면, 이미 개편은 이루어지고 있다고 봐야할 것이다. 조직의 개편은 그 과정의 마지막에 이루어지는 것이 최선이고, 그렇지 않다면 변화된 조직에서의 자신의 위치를 잡는데 또 다른 노력이 낭비될 것이다.

# [ 인연 쌓기 ]

흔히들 아는 사람을 통해서 세 번만 건너뛰면 우리나라의 모든 사람과 연결이 가능하다고 한다. 예를 들어, 한 사람이 아는 사람이 천 명이라고 할 때, “1000x1000x1000”정도면 왠만한 사람은 거의 다 알 수 있다. 말 그대로 인적 네트워크가 얼마나 협소한가를 알 수 있는 단적인 예이다. 미국 영화에서도 "케빈 베이컨"이라는 사람이 출연한 영화를 통해서 할리우드의 모든 배우가 연결되어 있다는 말도 있다. 사람의 인연이 이토록 가깝고도 질긴 이유에는 우리의 기억력을 돕는 여러 가지 도구들이 나오기 때문이다.

한 동안 "I Love School"이라는 프로그램이 유행하던 시절이 있었다. 인터넷 사이트에서 자신이 졸업한 초/중/고/대학교들을 입력하고, 각자가 다녔던 해를 추가해 주면 친구들을 찾아주는 유명한 곳이었다. 물론, 그곳이 좀 더 발전했더라면 지금도 남아 있겠지만, 이미 사라져 지금은 찾아볼 수 없다. 이제는 그곳에서 서비스되던 것들이 핸드폰의 "Band"라는 곳에서 찾아볼 수 있는데, 나이가 들면 자신의 기억과 경험이 소중해지고 그런 것들을 통해서 지속해서 끈을 놓치지 않으려고 하는 것이 인지상정이다. 사람은 관계 속에서 삶의 의미를 찾으려는 성향이 있다.

그럼 이런 말을 주저리 주저리 하는 이유는 뭘까? 우리가 일하고 있는 직장 내에서의 인연도 그에 못지않게 중요하기 때문이다. 초/중/고/대학교에서 만들어진 인연은 아무런 목적이 없이 친하다는 이유로 모일 수 있지만, 직장에서의 인연은 아무래도 인성뿐만이 아니라 실력도 어느 정도 고려가 된다. 물론 편하게 만나는 데는 그런 것이 중요하지 않을지는 모르겠으나, 직장을 이직하거나 혹은 어떤 일을 위탁할 곳을 찾는다든가 할 때는 예전에 알던 그 사람을 기억에 떠올린다. 어떤 기준을 가지고 정말 그 사람을 만나는 것이 적절한지를 따져보게 된다. 설령 능력은 조금 모자라더라도 일 처리에 빈틈이 없다면 좋겠지만, 실력이 좋을지라도 인성이 좋지 않던가 책임감이 부족하다고 생각될 때는 절대 만남의 대상에 넣지 않을 것이다.

따라서, 지금 같이 일하고 있는 동료나 선/후배와의 관계가 중요하다는 것을 알아야 한다. 작은 관계라도 우리가 일하는 분야는 그리 넓지 않다. 몇 사람을 거치면 그 사람에 대한 과거의 행적은 쉽게 알 수 있는 상황이다. 사람을 대하는 태도에 대해서 특히 조심해야 한다. 존중하지 않는 사람은 존중받지 못할 것이 분명하기 때문이다. 인연을 쌓는 것은 오랜 시간이 걸리는 것이지만, 한 순간에 모든 것을 잃을 수 있다. 사람은 자신이 받은 것보다는 준 것을 더 많이 기억하는 경향이 있고, 받은 것은 쉽게 잊어버리고 만다. 큰 것은 대수롭지 않게 여길지라도, 작은 정이 쌓여서 만들어진 것은 소중하게 여긴다. 작은 인연들이라도 존중하는 마음으로 소중하게 지속시켜 나간다면 미미한 자신의 능력을 보완해 줄 수 있는 든든한 산이 되어줄 것이다.

# [ 협상의 시간 ]

소프트웨어 개발은 정치판과 같아지고 있는 경향이 있다. 일단 위험 신호가 들어오면 필요 없는 것들을 버릴 준비를 해야 한다. 과제에 관심을 가진 사람들은 어떤 기능을 가진 제품이 언제 나오는지에 대해 관심이 있지, 왜 그 때 나오지 못하는지는 관심을 가지지 않는다. 그리고, 솔직히 어떤 기능이 어떻게 사용되기에 중요한지, 그렇지 않은지도 잘 모르는 경우가 많다. 따라서, 과제 일정의 지연이 확연히 보이는 시점에서는, 해야 할 일을 줄이는 데 집중해야 한다. 정치판이 되는 이유는 과제를 자신의 힘을 표현하는 잣대로 삼으려는 사람들의 논리 때문이며, 그 속에서 희생되는 것은 개발자의 아까운 인생뿐이다.

기능은 과제가 만족시켜야 하는 기본 조건이다. 성능은 그 기능에 얼마나 만족하는지를 보여주는 척도다. 기능을 없애고 만족해야 할 성능 목표를 조정하지 않는다면 과제는 절대 정해진 일정을 만족시키지 못하는 경우가 많다. 따라서, 이런 상황에서는 개발자들이 협상으로 나가야 하는 시간이 온다. 협상에서의 가장 중요한 것은 상호 간에 만족할 수 있는 합의점에 도달하는 것이다. 따라서, 이 때는 누구의 잘못을 탓하기에는 이미 늦어버린 시점이기에 어떤 것이 가장 우선순위가 높은지 정렬을 해야 한다. 우선순위가 낮은 기능들은 언제인지는 정확히 알지 못하지만 다음으로 구현이 미뤄지게 된다.

우선순위가 높은 기능들은 어떤 것들이 있으며 누가 정하는 것일까? 사실 이것은 누군가가 정한다기보다는 만들려는 제품이 무엇이냐는 기본적인 질문에서 시작해야 한다. 어떤 점이 좋아서 우리의 제품이 경쟁력을 가질 수 있고, 사용자들이 그 기능에 만족할 수 있는가를 보아야 한다. 개발자는 구현하기 쉬운 것들을 위주로 이야기할 것이고, 상품기획이나 마케팅은 시장에서 많이 팔리거나 팔릴 가능성이 있는 제품들의 스펙(Spec.)을 가지고 이야기할 것이다. 따라서, 쉽게 결론에 도달할 수 있는 정렬된 우선순위 목록은 만들기 어려울 것이다.

누군가의 강한 주장과 토의, 혹은 강압적인 분위기로 기능의 우선순위 결정들은 이루어진다. 이런 과정에서 소프트웨어 개발자가 할 수 있는 것은 뭘까? 단순히 각 기능에 대해서 얼마나 걸릴 것이라는 예상 답안을 제출하는 것이 전부일까? 아니면, 솔직히 현재 과제의 인력으로는 그런 것들을 구현하는데 필요한 절대적인 시간과 인력이 부족함을 감정에 호소하는 것이 좋을까? 이미 늦었다는 것을 알고 있는 상황에서는 가능하면 시간을 확보하는 것이 최선이다. 물론, 그렇다고 그 시간동안 모든 기능을 구현하는 것은 힘들다. 늦어버린 과제는 더 지연되지 않는다고 보장하지 못한다. 실제로 늦어진 과제는 더 늦어질 가능성이 높으며, 이미 늦었다는 신호가 켜진 과제는 일정을 따라잡는 것이 거의 불가능에 가깝다.

어쨌든 일정에 대한 질문은 언제든 날아온다. 누군가를 통해서 끝없이 과제 책임자에게 전달되고, 과제 책임자는 언제든지 그것을 대답할 수 있도록 머리 속에 암기를 하고 다닐 수준이 되어야 한다. 소프트웨어 개발자들이 동의하지도 않고, 그렇게 될 것도 같지 않은 일정을 제시하는 것은 개발 내부의 반발심만 키우게 된다. 과제는 결국 더 지연될 수밖에 없다. 오히려 질문을 받으면, 즉시 대답하는 것을 회피하고 내부에서 정밀한 일정을 산정할 수 있도록 모두의 의견을 모아야 한다. 대체로 개발자들이 제시한 일정이 맞는 것이 사실이기 때문이다. 거짓말로 순간의 위기를 모면하는 실무 개발자는 극히 드물다. 하지만, 반복된 질문은 개발자의 아까운 시간만 낭비시킬 뿐이다. 그리고, 결코 최적의 일정을 만들더라도 지켜질 가능성은 희박하다. 최적은 거의 제대로 지켜질 가능성이 없다는 말과 동의어이다.

협상의 시간이 길어지면 모든 이해 관계자가 다 피곤해진다. 나중에 가서는 결국 나오지 않는 제품을 포기하거나, 혹은 아예 일정을 확실히 늦춰서 잡아버린다. 왜냐하면, 그와 같은 협상의 시간을 다시 가진다는 것이 서로에게 괴로운 경험이기 때문이다. 후자일 경우에는 과제 개발이 완료되어도 좋은 소리를 듣기는 어렵다. 이미 망가질 대로 다 망가진 계획은 누구에게도 즐거움을 주지 못할 것이기에 때문이다. 하지만, 그렇다고 하더라도 경험은 남는다. 실패했을 때는 그 실패의 원인을 반복해서는 안되고, 성공했을 때는 "Best Practice"라는 주제로 "얼마나 열심히 일했고, 어떻게 해서 성공시켰는지" 이야기할 것이다.

절대 잊어선 안 되는 것은 과제가 협상이 되도록 만들지 말아야 한다는 것이다. 이건 개발만의 일이 아니라 과제의 성공을 바라는 모든 사람이 함께 만들어 가야 할 공동책임이라는 것을 알아야 한다. 상품기획, 마케팅, 개발 책임자 등 모든 이해 관계자는 과제의 결과에 대해서 동일한 책임을 가진다. 과제가 일정 계획을 벗어난 것은 개발자에게만 책임이 있는 것이 아니다. 필요 이상의 기능을 제시한 사람, 말도 안되는 일정을 강요한 사람, 개발은 모르지만 자신의 숟가락은 올려두고 싶어하는 모든 사람이 함께 책임져야 한다. 물론, 이런 경우는 드물겠지만, 적어도 과제의 실패는 한 부분의 관리 실패에서 원인을 찾아서는 안된다. 혼자하는 과제는 없으며 과제의 실패를 “시스템의 실패”로 받아들여야 개선이 가능하기 때문이다.

# [ 폭언에 맞서는 방법 ]

회사 생활에서 이직의 첫 번째 주요 이유는 사람 사이의 관계에서 발생한다. 스트레스가 극도로 높아진 상황에서는 제대로 된 이성이라는 것이 존재하기 어렵우며, 모든 것이 짜증을 유발하는 원인처럼 보인다. 자신의 이직 사유를 살펴보면 잘 알겠지만, 이직 전의 몇 개월은 우울증에 가까울 만큼 그런 생각들로 잠을 잘 이루지 못하는 것을 경험했을 것이다. 그 중에서도 상사로 부터 듣는 폭언은 의욕감퇴 및 행동 주체로서의 판단능력에 대한 의심과 함께 자율성이라는 인간의 가장 높은 가치까지도 상실하게 만든다.

큰 회사는 나름 이런 상황을 순화하려는 노력을 보이려고 애쓴다. 아침에 나오는 방송은 연일 회사에서 잘한 일과 좋은 성과가 있었던 일을 알려주고, 간혹 특정 문제가 심각해지려고 할 때는 그 문제가 발생하지 않도록 지속해서 예방 교육을 실시한다. 하지만, 아침에 잠시 나오는 방송과 점심시간에 지나가듯 듣는 방송은 오히려 현실을 외면하는 듯한 인상을 주기 쉽다. 아무리 그런 교육을 한다고 해도 하루 아침에 사람이 바뀌지는 않는다. 방송은 현실을 개선하는데 크게 영향을 주지 못하지만, 위 사람이 하는 말은 직접적인 삶의 질에 심각한 영향을 줄 수 있다는 것이다.

폭언하게 되면 말하는 사람은 일시적인 욕구불만이 해결될지는 모르겠지만, 대부분의 듣는 사람은 내면으로 더 깊숙히 들어가게 된다. 웬만한 문제가 있다고 하더라도 폭언하는 사람 앞에서는 꺼내고 싶어하지도 않으며, 자신이 잘 못 이야기한 것으로 인해서 타인에게 질책이 갈지도 모르기에 알고 있는 문제도 그냥 덮어둔다. 절대 곪아 터지기 전에는 문제가 있다는 것을 이야기하지 않는다. 하지만, 그 때가 되면 이미 문제는 걷잡을 수 없을 만큼 커져 있는 경우가 많고, 그로 인해서 해결 방안은 더디게 진행될 것이다. 또한, 지속적인 질책을 듣게 되면 자신이 무엇을 잘하는지도 잊어버리게 되고, 새로운 생각을 한다는 것이 사치스럽다는 느낌까지 들게 된다. 결과적으로 우리 모두의 미래를 갉아먹고 만다.

그렇다면 이런 폭언을 들었을 때 우리는 어떻게 해야 할까? 여러가지 태도가 있겠지만, 가장 좋은 것은 그냥 잊어버리고 자기 일에 집중하는 것이다. ("개가 짖는다고, 같이 장단 맞출 필요는 없다".) 무시하는 것이 정신 건강에 이롭다. 깊이 생각하고 반성하는 것도 어느 정도 한계가 있다. 계속 반성하고 잘못했다고 이야기할 필요는 없다. 사람은 완전한 존재가 아니기에 무언가 실수를 하기 마련이고, 그런 실수의 경험에서도 배우는 것이 사실이기 때문이다. 실수가 용납되지 않는 회사는 거짓이 난무하거나 부풀려진 실적만이 있는 곳일 가능성이 높다. 애초에 그런 분위기를 만든 자체가 오히려 더 문제가 있는 것이다.

중요한 것은 '폭언을 일삼는 사람은 나보다 오래 동안 회사에 다닐 가능성이 크지 않은 사람"이기 때문이다. 그 사람이 가고 새로운 사람이 온다고 해서 크게 달라질 것은 없겠지만, 그래도 우리는 남아서 회사와 “또 하나의 가족”을 지켜야 한다. 남은 사람들은 회사를 잘 키워나가기 위한 준비는 미리 해야 한다. 지금 해야 할 일을 주변의 환경 때문에 못한다는 생각은 버리는 것이 좋다. 아무리 윗사람이 “무슨 새끼"처럼 보일지라도 그 사람에게도 배울 것은 충분히 있다. 적어도 그렇게 행동해서는 안된다는 것 정도는 배울 수 있을 것이기 때문이다. 그리고, 자기 일에 대한 완벽을 추구하는 태도는 언제나 환영받는 것이기에, 한 걸음 한 걸음 자신의 능력을 키우는 일은 절대 게을리 해서는 안 된다.

물론, 모든 것을 무시하고 가기에는 힘든 부분이 많을 것이다. 무시하기 어렵다고 부딪히게 되면 더 많은 것을 경험할 것이 분명하다. 자신의 능력으로 모든 것을 극복하기 힘든 순간이 찾아올 수도 있다. 그 때는 이렇게 생각하도록 하자. "잠시 바람 위에 일순의 휴식이 오면, 또 다른 여인이 나를 나으리라(칼릴 지브란의 "예언자"에서)”. “당신도 나도 언젠가 이별이 있음을 알고 있듯이, 오늘은 여기까지만 하자”. 중요한 것은 누가 더 오래 살아남는 것이지, 누가 더 크게 소리치는 가에 달려있는 것이 아니다. 사람은 잊히지만 자신이 한 일은 오래 동안 기억되기 마련이다. 기억되지도 못할 따위의 사람을 위해서 자신의 소중한 다시오지 못할 시간을 낭비하지는 말자.

자주 직장 내의 폭언이라는 이야기가 신문이나 방송에서 자주 등장하고 있다. 예전에는 직장에서 일할 수 있다는 것 자체 만으로도 만족하면서, 이 정도는 아무것도 아닌 것처럼 넘어갈 수 있는 것들이 이제는 그렇지 못한 세상이 되었다. 세상이 바뀌고 사람이 바뀌는 것을 알지 못하는 오래된 경험의 소유자들이 새로운 문화에 적응하지 못하는 현상처럼 보인다. 예전의 폭언(혹은 폭행까지도 동반한)은 극복해야 할 대상이지만, 현재는 사라져야할 나쁜 관습일 뿐이다. 하지만, 나쁜 전통(?)은 좀처럼 잘 사라지지 않는 경우가 많으며, 아직도 상습적인 폭언을 일삼는 사람들을 만나기도 한다. 그들이 그렇게 행동하는 이유는 개인적인 성향도 있겠지만, 자신의 잘못된 경험에서 나왔을 수도 있다. 잘못 배운 사람은 자신의 경험을 당연한 것으로 받아들이기 때문이다.

폭언 근절에 대한 메시지가 사내 아침 방송과 게시판에 자주 등장하다보니, 그것을 일삼던 사람들의 태도도 달라지고 있다. 직접적인 방법을 사용하던 이전과는 달리, 간접적으로 분위기를 얼음장처럼 차갑게 가져간다. 자신에게 하는 이야기처럼 남에게 욕설을 퍼붓기도 하고, 감탄사의 일부인것처럼 공개적으로 욕설을 큰 목소리를 낸다. 하지만, 이것 역시도 폭언이라는 것을 그 사람들은 아는지 모르는지 솔직히 유치하게 생각되는 것이 사실이다. 관행처럼 반복되는 감정적인 말들은 조직을 와해시키는 폭언과 다르지 않다는 것을 알아야 한다. 신뢰가 바탕이 되지 못한 조직은 오래가지 못한다. 폭언만 일삼는다면 당연히 신뢰하는 관계는 만들어지지 못한다. 의심과 눈치만 살피는 조직이라면 망하는 것도 시간 문제일 뿐이다.

지속해서 폭언을 듣게 되면 사람은 자신의 행동을 고치기 보다는 내부로 숨게되는 경향을 보인다. 즉, 행동은 소극적으로 바뀌고 문제가 있어도 이야기하기를 꺼려한다. 자신의 일이 아니라고 판단할 때는 절대 나서려고 하지 않게되며, 문제를 내부에서 해결하고 외부로는 공개가 되지 않도록 만든다. 자신이 관리하는 팀 내부에서 외부로 나가는 모든 대화 및 결정에 대해서는 자신의 통제하에 두기를 원하게 되며, 팀 간의 협업은 말 그대로 공식적인 회의에 의해서 정해진 것만 받아들이려고 한다. 팀과 팀의 경계는 더욱 높고 단단해지며, 이러한 부분들이 하나씩 쌓이게 되면 결국 협업은 말로만 떠들어대는 공허한 이야기가 된다.

폭언에 응답하는 방법은 좀 더 계획적이고 공식적이어야 한다. 폭언을 하는 상사는 적어도 나보다는 더 위 사람들과 가까울 것이 확실하기에 우리는 계획적이고 체계적으로 움직여야 한다. 사례를 기억으로 남기기 보다는 정확히게 기술해서 문서로 정리해야 한다. 또한, 회사에서 그런 사례를 조사하려는 노력이 보이면 근거로 제시해야 한다. 사실 무기명이라는 것은 별로 중요한 것이 아니다. 힘의 핵심에 가까이 있는 사람이 나보다는 더 많은 정보를 얻는 것은 당연하다. 무기명이라고 하더라도 밝혀질 가능성이 높다는 것은 반드시 명심해야 할 것이다. 하지만, 그렇다고 회사가 무턱대로 책임을 전가하지는 못한다. 근거를 가지고 이야기할 때는 더 큰 힘을 가지기 마련이다. 항상 기억해야 할 것은 “기억보다 기록에 의존”해야 한다는 것이다.

따라서, 무기명이라고 이야기할 때는 내부고발에 대한 위험 부담도 감수할 만큼의 각오가 있어야 한다. 어쨌든 중요한 것은 근거 자료다. 혼자 고민하는 것 보다는 인사부서와 같이 이야기를 해야 한다. 인사 담당자의 말은 믿지 않아야 한다. 그 사람도 시키는데로 할 뿐 특별한 힘을 가지고 있는 것은 아니다. 중요한 것은 몇 번이고 이것을 제기해야지 개선이 된다는 점이다. 못난 사람들은 여러 번 이야기를 들어야 자신의 행동을 고친다. (그렇다고 다 고쳐지는 것은 아니지만.) 당분간의 불함리한 대우는 감수해야 할지라도 오래동안 그런 시간을 가지는 것은 달갑지 않을 것이다. 만약 개선될 희망이 보이지 않는다면 회사를 옮기는 것도 진지하게 고민해야 할 것이다. 물론, 그 만큼 역량이 있어야 하겠지만, 그렇다고 자신의 미래를 희생할 만큼 가치가 있는 회사인지 고민해야 봐야 할 것이다.

혼자서 고민한다고 해결되는 것은 없다. 우울증을 짊어지고 살거나, 혹은 그것을 똑 같이 배워서 다른 사람에게 스트레스를 해소하거나, 정말 못견디겠다면 능력있을 때 회사를 그만두고 다른 직장을 알아보는 것이 최선일 것이다. 하지만, 조금이라도 남겨진 사람들을 생각한다면, 솔직하게 모든 정보를 공개해야 한다. 자신이 받은 모멸감과 자존심의 상처, 그리고 비인격적인 대우를 기록하고 알려야 한다. 구체적인 상황을 가지고 누구도 반박하지 못하게 논리적으로 이야기해야 한다. 회사를 떠나는 것이 "남은 사람에 대한 미안함"이 되지 않도록, 짊어져야 할 짐은 자신이 떠 안아야 한다. 비록 이번에는 고쳐지지 않더라도 조금이라도 아쉬움이 있다면 개선에 대한 시발점이 되도록 만들어야 할 것이다.

# [ 일을 적게 하는 것이 성공하는 비결 ]

관리자는 모든 일을 자세하게 알고 있어야 한다고 생각하는 사람들이 많다. 사실 그렇게 알고자 노력한다면 자신은 편해질지도 모르지만, 그렇게 하기 위해서는 많은 사람의 희생이 따른다는 것을 반드시 알아야 한다. 관리자는 일을 적게 할수록 과제가 더 잘 돌아가는 것이 현실이다. 특히 많은 사람이 모여서 일하는 곳에서는 관리자는 일하는 사람들을 위한 도구로 활용될 수 있어야 한다. 즉, 일을 잘 할 수 있는 환경을 만들어주는 데 집중해야 하지 방해가 되어서는 안될 것이다.

우리가 하는 보고의 형태를 보자. 상급자가 아래에 있는 관리자에게 업무의 진행 상황을 물어볼 때 주로 던지는 질문에는 어떤 것들이 있을까? 상세한 문제에 대한 질문을 던진다면, 이것은 잘못된 질문일 가능성이 높다. 사실 그런 질문에 대답하기 위해서는 과제에 직접 투입되어 해당 업무를 수행하거나, 혹은 그 업무를 수행하는 개발자를 붙잡고 상당 시간의 이해를 높이는 대화를 했을 경우에 가능하다. 따라서, 그 질문에 대한 대답을 얻기 위해서는 개발자들이 "소를 키우는 일"보다는 다른 일에 시간을 보낼 수밖에 없다. 보고서를 작성하고, 리뷰 회의를 하고, 다시 그것을 정리한 자료를 이쁜 그림을 붙여서 발표하게 될 것이다. 이런 저런 것들에 들어가는 오버헤드는 결국 개발자의 시간을 낭비하는 것으로 채워질 수밖에 없다.

그렇게 많이 안다고 해서 갑자기 버그의 수가 극적으로 줄어들까? 아니다. 일하면 할수록 버그의 수가 늘어나는 것이 정답이다. 일을 잘하든 못하든 코딩의 양이 늘어나게 되면 당연히 많은 버그를 발견하게 될 가능성이 높다. (물론, 코딩을 정말 잘하는 사람은 자신이 코딩한 부분에 대한 것은 놓치지 않고 테스트하겠지만) 어떤 관리자는 "버그의 수가 많은 개발자에게는 패널티(Penalty)를 주겠다."고 이야기하기도 한다. 참으로 어리석은 생각이라는 것을 스스로는 알고 하는 소리일까? 그런 관리자는 소프트웨어 개발의 속성을 모르고 하는 말임에 틀림없다. 가능한 테스트 케이스를 늘리라고 이야기하는 관리자도 있다. 이것 마찬가지로 어리석은 말이다. 왜냐하면 테스트 케이스를 늘리는 것은 쉽지만, 그렇다고 걸러지는 버그가 많아진다고는 이야기할 수 없기 때문이다. 따라서, 담당자는 무조건 테스트 케이스를 늘리기만 급급할 것이다.

테스트 케이스가 많다는 것이 테스트를 더 많이 한다고 생각할 수도 있겠지만, 테스트가 검증하는 코드의 범위를 알지 못하는 상태에서 늘어난 테스트 케이스는 시간 낭비만 가져올 뿐이다. 테스트를 수행하기 위한 사람과 시간도 함께 늘어난다는 것은 모를 것이다. 그리고, 늘어난 테스트 케이스가 항상 같은 코드만 실행하고 있다면 결과는 달라지지 않는다. 예를 들어, 받아들일 수 있는 입력이 1에서 10까지라고 할 때, “0, 1, 9, 10”은 의미가 있을지 몰라도, “4, 5”는 실행 결로가 달라지지 않을 가능성이 높기 때문이다. 그렇게 테스트 케이스를 늘려서 테스트 시간과 노력을 추가적으로 들이기보다는, 오히려 개발자의 테스트를 어떻게 하면 자동화를 시킬 수 있는지를 고민하는 것이 더 효과적일 것이다.

자동화되지 않은 테스트는 일정의 낭비와 과제에 대한 불안감을 키워서 결과적으로 보고서 작성과 누구의 잘못인지도 모를 결과 만을 만든다. 시간을 어떻게 효율적으로 사용할지에 대한 개념도 없는 사람의 무책임한 말 한 마디가 결국 과제의 일정 지연에 대해서 결정적으로 영향을 주었다는 것은 누구도 이야기하지 못할 것이다. 하지만, 실제 개발 업무를 수행하는 사람은 다 알 것이다. "제발 쓸데없는 보고서 작성과 시간을 비능률적으로 사용하는 회의를 안 했으면 합니다."라고 이런 저런 간담회에서 건의사항이 나올 것도 뻔하다. 자동화되지 않은 테스트는 결국 투입되는 사람과 시간을 늘리기만 할 뿐 과제의 시간을 줄이지는 못한다. 늘어난 테스트 케이스가 해줄 수 있는 것이 더 많은 버그를 발견할 수 기회라고 생각할지도 모르지만, 그런 일은 일어나지 않는다. 오히려 더 많은 버그 관련 미팅만 늘어날 것이다.

관리자가 해야 할 진정한 일은 개발자가 자신의 능력을 충분히 발휘해서 업무에 몰입할 수 있는 환경을 만들어 주는 것이다. 그 외에 쓸데없는 것들은 다 버려야 한다. 모든 것을 알려고 해서도 안 되며, 차라리 그 분야에 전문가인 사람을 키우는 데 시간을 보내야 한다. 권한을 주고 책임을 이양하라는 것이다. 물론, 돌아오는 비판은 관리자의 몫이며 주어지는 상은 모두의 것이 되게 하는 것도 관리자의 일이다. 따라서, 많은 일을 하려고 나서지 말고, 자신이 관리하는 사람들이 일하지 못하게 만드는 비능률적인 부분을 제거하는데 힘을 기울여야 한다. 일일보고, 주간보고, TF(Task Force), 기타 업무 회의 등등에 대한 자그마한 절약부터 시작해서, 업무 공간에 대한 안락함을 제공하는 등의 일이 관리자의 업무가 되는 것이다. 모든 것을 해결하기는 어렵겠지만, 최소한 개발자가 어떤 어려움을 가졌는지 진지하게 듣는 것이 핵심이다.

# [ 소프트웨어 개발자의 일상 ]

인생에 목표가 있다면 아마도 제대로 자신이 원하는 무언가를 해보는 것일 것이다. 하지만, 대부분의 사람들은 제대로 일을 할 준비가 되었다고 느끼는 순간 멈추고 만다. 혹은 이미 또 다른 욕구가 생겨버려서 목표가 변경되기도 한다. 뭔가 꾸준함을 가지고 할 수 있는 것을 찾는 것에 우리는 너무 소홀한지도 모른다. 만약 이런 생각이 든다면 잠시 하던 일을 멈추고 정말 자신이 원하는 것이 무엇인지 다시 한번 고민해 볼 필요가 있을 것이다.

소프트웨어 개발자의 일상은 어떤가? 아침에 사람으로 가득 찬 지하철과 버스를 타고, 허겁지겁 회사에 도착해서 상사의 눈치를 보고 자신의 자리에 앉는다. 컴퓨터를 켜고 자신에게 온 메일을 가장 먼저 체크한다. 혹시 어젯밤에 놓치고 간 것으로 인해서 불평하는 사람이 있는지도 확인하는 시간이 될 수 있다. 일단 자신에게 요청된 것을 급하게라도 마무리하고 나면, 약간의 안도감으로 "전자신문"들을 훑어본다. 혹시 새로운 기술이나 제품이 경쟁사에서 나오지 않았나를 확인하고, 상식 수준이라도 올리려고 여기저기 기웃거린다. 그러다 보면 느끼는 것이 남들은 벌써 저기까지 갔구나 하는 부러움과 함께 자신의 신세를 조금은 후회할지도 모른다.

오전의 미팅에서 어제 나온 버그들에 대한 성토회를 하고, 맥 빠진 기분으로 커피를 한잔으로 위안을 삼을지도 모른다. 길어지는 회의에서 정작 자신이 하는 말은 몇 마디가 안 됨을 알지만, 부쩍 늘어난 눈치로 상사의 찌푸린 얼굴만 쳐다보다가 "예"하고 물러나고 만다. "아니오"라는 대답을 하기 위해서는 많은 준비를 해야 하고, 자신은 아직 준비가 덜 되었다는 것만 느끼게 될 것이기 때문이다. 가장 쉬운 대답을 선택했지만, 왠지 그렇게만 대답하기에는 부족하다고 느낄지도 모른다. 그렇다고 딱히 대책이 있는 것도 아니기에 일단은 그렇게라도 회의를 마무리하는 것이 최선이라고 생각할 것이다.

컴퓨터 화면에는 어제 코딩하다 멈춘 부분을 다시 찾아내고 기억을 복구해서 다시 떠올린다. 일단은 머릿속으로라도 버그가 없다고 생각되면 코드를 컴파일하고, 별다른 문제가 없다면 체크인(Check-in)전에 코드 리뷰 요청을 보낸다. 운 좋으면 빨리 리뷰를 받아서 체크인 하게 되고, 운이 없으면 일일이 사람들을 찾아가서 설명해야 할지도 모른다. 어쨌든, 체크인 된 코드가 제대로 다시 컴파일 되는지도 확인해야 하는 것도 당연히 해야할 일이다. 하지만, 제대로 컴파일 안되는 경우도 다반사다. 남들도 이미 새로운 코드를 체크인 했을 가능성이 크기 때문이다.

단위 테스트와 같은 것도 이미 물 건너간 이야기다. TDD(Test-Driven Development)라는 것을 남들이 한다고 하는데, 자신이 속한 부서는 할 기미조차도 안 보이고 아무도 관심이 없다. 위에서 내려오는 테스트 커버리지(Test Coverage) 목표는, 수치상으로 함수나 문장(Statement) 수준에서 만족하도록 조작할 수도 있다는 것도 이미 충분히 잘 알고 있다. 그리고, 그것이 아무 의미가 없는 시간 낭비라는 말도 여러 동료에게서 들은 상태이다. 따라서, 일단 자신이 작성한 코드가 들어간 바이너리 이미지(Binary Image)를 수동으로 테스트해서 제대로 동작하는지만 간략하게 확인한다.

점심시간 이후는 정말 힘든 시간이다. 일에 대한 집중력이 떨어지고, 보고 있는 책도 눈에 들어오지 않는다. 위에서 떨어지는 갑작스러운 요청에 이런저런 “PPT” 자료에 그림을 만들다 보면 서너 시간은 휙 하고 지나가고 만다. 최근에는 코딩보다 "자주 평가"라는 말로 개발자가 사용자 시나리오를 확인하는 일이 늘어나고, 그 결과는 일정하지 못해서 문제가 줄어드는 것처럼 보이지도 않는다. 걱정이라면 주어진 과제 일정을 맞추지 못하는 것이 아니라, 얼마나 지연을 안 시킬 수 있을까를 고민하는 것이다.

시도 때도 없이 걸려오는 테스터의 전화를 받고 내용을 확인하다 보면, 제대로 책상에 앉아서 코드의 아름다움을 논하는 것은 "Google" 같은 회사에서나 가능할 것이라는 상상만 한다. 봐야 할 책은 책상의 한 귀퉁이에 쌓이지만, 정작 제대로 다 읽은 책은 한 권도 없다. 아내가 "또 책사면 죽을 줄 알아!"라고 한 소리를 떠올리지만, 그렇다고 새로운 지식에 대한 목마름이 가시지 않는다. 저녁 시간이 되어야 일 할 수 있는 분위기가 된다. 그렇게 전화통을 붙들고 떠들어대던 고참들도 그 시간쯤 되면 자리를 비우거나 자신 만의 일에 몰두한다. 조용한 분위기가 20분도 이어지지 않고, 서로의 얼굴을 마주하고 볼 수 있을 정도의 파티션 높이지만 조금이나마 평온이 찾아온다. 몰입은 안 되지만 몰두하려고 노력하는 것도 하루 일과 중 이 시간이 유일하다.

물론, 이미 야근할 것이라는 것을 결재로 올린 후라 이제는 정해진 시간까지는 일해야 한다. 얼마되지 않는 교통비지만, 그래도 한 달에 몇 십만 원을 집에 가져다줄 수 있다는 사실이 고마울 뿐이다. 와이프도 은근히 그걸 바라는 눈치 같기도 하다. 그나마 회식과 같은 일이 없을 때는 지친 눈으로 밤 11시까지는 집에 들어갈 수 있다. 애들은 다 이미 잠든 상태고, 와이프도 애들 곁에서 선잠에서 깬듯한 모습으로 "왔어?"하고는 다시 이불 속을 파고든다. 자고 있는 모습이 아름답다는 생각이 드는 것은 혼자만의 생각이 아닐 것이다.

하루를 마감하기 위해서 다시 집에 있는 노트북를 켜고, 이렇게라도 생활의 일부를 한 자 한 자 적어 놓는다. 소프트웨어 개발자로 애들이 아빠와 같은 삶을 살기를 원하지 않는다고는 하지만, 그래도 아들이 아빠 정도 만이라도 하기를 바라는 것도 사실일 것이다. 우리같은 소시민의 삶이 진정한 행복이라는 것도 알기 때문이다. 자그마한 행복을 위해서라면 큰 희생이 필요하다는 것을 이미 몸으로 깨우쳤기 때문이다. 늦은 밤 조금이라도 여유가 주어진다면, 잠든 아이들과 와이프의 평온한 얼굴을 보는 것이 하루에 누릴 수 있는 유일한 행복이다. 하루가 마무리된 곳에서 또 다른 하루를 다시 시작하지 않기 위해서, 조금이라도 새로운 것을 시도할 수 있는 유일한 “자신 만의 시간”이 비로소 찾아오는 것이다.

# [ 프로근성이란? ]

흔히들 프로의식, 혹은 프로 근성이라는 말을 쓴다. 그리고, 자신이 프로페셔널이라고 이야기한다. 그렇다면, 무엇이 프로와 아마추어를 나누는 특징이 될 수 있을까? "The Clean Code"라는 책에 있는 말로 프로는 자기가 한 일에 대한 책임(Responsibility)을 진다는 것으로 압축할 수 있다. 소프트웨어 개발자로서 우리는 고객을 만족시켜야 할 책임이 있고, 자신이 한 잘못에 대해서도 수정할 책임을 지고 있는 것이다.

소프트웨어 개발자는 일에 파묻혀 지내는 시간이 많다. 그러다 보면 도메인(Domain)에 대한 이해는 높아지지만, 새로운 기술에 대한 흐름은 놓치게 된다. 점점 더 그러한 시간이 오래 될수록 흘러간 기술에 대한 미련 속에서 살게되고 새로 들오는 개발자와는 멀어지게 된다. 사실 몰라서 못 쓰는 기술이 대부분일지도 모른다. 하지만, 프로 개발자라면 자신의 역량개발에 꾸준한 노력을 투자해야 한다는 것도 항상 염두에 두어야 한다.

소프트웨어 개발자는 자신이 한 일에 대해서도 끊임없이 개선해야 한다. 문제가 있는 코드를 발견했을 때 바로 수정할 수 있는 용기도 필요하다. "잘 동작하는 코드를 수정하지 말라"라는 말은 실제로는 틀린 말이다. 잘 동작하는 코드가 변경할 필요가 없다면 상관없겠지만, 변경해야 하는 상황이 왔는데도 변경하지 않는 것은 문제가 될 수 있다. 그리고, “세상에 변경되지 않는 것은 없다”는 것은 변하지 않는 진리다.

코드를 변경하지 않겠다는 것은 변경에 대한 대비를 자신이 안 했다는 것을 이야기할 뿐이다. 수정된 코드가 제대로 동작하는지 확인할 수 있는 개발자 수준의 테스트 방법을 가지지 못했다는 것을 스스로 인정하는 것이다. 변화에 대한 융통성을 가지고, 자신이 했던 일을 언제든 검증할 수 있도록 만드는 것이 프로 개발자와 아마추어 개발자의 차이다. 진정한 프로라고 한다면 아마추어처럼 이것은 되고 저것은 안된다고 할 것이 아니라, 오히려 그런 제약을 없애는데 노력을 더 투자해야 할 것이다.

프로 개발자는 세세함을 놓치지 않고 끊임없는 자신의 역량 향상에 노력한다. 자그마한 문제라도 더 좋은 코드를 짜려는 의지가 담길 때 수준이 달라졌음을 스스로 느낄 수 있을 것이다. 실력이 늘었다는 말은 특정한 새로운 기술을 배웠다는 것이 아니라, 일상에서 그 기술을 의식하지 않고도 충분히 활용할 수 있다는 말이다. 즉, 하루 하루의 일상에서 자그마한 부분에 대한 꾸준한 개선이 질적인 차이를 만들어야 프로라고 불릴 수 있다. 깔끔하고 좋은 구조의 코드를 만드는 것은 일상의 코딩 습관에 의해서 이루어지는 것이며, 그렇게 모여서 쌓인 결과물이 프로와 아마추어를 구분하는 “질적인 차이”를 보여주게 될 것이다.

# [ 피플웨어(Peopleware) ]

모든 차이를 만들어내는 것은 사람이다. 사람을 어떻게 움직이느냐에 따라 결과가 달라진다. 조직장으로서의 일은 그런 결과물을 만들어가는 “사람을 관리하는 것”이고, 제대로 된 결과물을 만들 수 있도록 “좋은 환경을 제공하는 것”이 임무다. 성과가 나는 조직은 실행에 집중하는 사람이 많은 조직이며, 그렇지 못한 조직은 끝없는 계획과 회의, 이런 저런 변경 등으로 생산성을 높이지 못하는 것들에 끌려 다닌다. 사람이 결과물을 만들며, 사람의 태도가 결과물의 품질을 수준을 결정하는 것이 소프트웨어 개발이다.

사람이 중요하다는 것은 아무리 강조해도 지나치지 않다. 소프트웨어를 개발하는 일은 노동 집약적인 산업이며, 고등 교육을 받은 사람들이 대량으로 투입되는 일이다. 하지만, 더 많은 경험과 더 높은 직급이 항상 더 우수한 소프트웨어를 개발할 수 있는 필수조건은 아니다. 현재 소프트웨어 개발을 총괄하는 임원들 대부분 소프트웨어 개발과 관련된 학과를 졸업하지 않았고, 하드웨어 개발에서 성과를 보여 개발팀을 리딩하고 있는 사람들이 많다. 모든 산업 분야는 아니라더라도 적어도 제조업에서 제품에 관련된 소프트웨어를 개발할 경우, 하드웨어를 전공한 사람이 소프트웨어를 전공한 사람보다 대체로 더 높은 직급으로 올라간다.

그 분들이 아는 지식은 최신의 지식이 아니며, 이런 저런 회의와 미팅에서 알게 된 단편적인 지식을 자신이 다시 가공한 것을 바탕으로 한다. 따라서, 이 분들은 소프트웨어 개발에 대해서는 거의 아는 게 없다고 봐도 과언이 아니다. (물론, 그렇지 않은 경우도 많다. 한참 벤처 붐이 일어난 1990년대 말과 2000년대 초반을 거치면서 소프트웨어 개발자가 임원으로 올라가는 경우가 늘어나기도 했다.) 아직은 비주류를 이루고 있지만, 이제는 점점 더 소프트웨어를 이해하고 고객에 대해 솔루션(하드웨어 및 소프트웨어를 함께)을 제공하는 일이 중요하다는 사실을 알기 시작했다. 그리고, 몇 몇 사람들은 소프트웨어가 가지는 근본적인 “복잡함의 관리”를 이해하고 싶어한다.

외국에서 나온 좋은 책들도 거의 실시간으로 번역되어 한글판으로 나오는 세상을 살고 있지만, 그렇다고 제대로 된 번역을 찾아보기도 힘든 것이 사실이다. 우리가 하는 일은 이론과 실무를 다 요구하는 일이다. 즉, 어떤 단어의 뜻을 안다고 되는 일이 아니고, 쓰임새를 정확히 이해해야만 할 수 있는 일이 된 것이다. 섣부른 지식은 실행의 오류를 낳고, 그 결과로 많은 사람이 고생하는 시대로 이미 들어섰다. "머리가 나쁘면 손발이 고생한다"는 말은 개인에게만 해당하는 농담이 아니며, 리더가 어떤 관점에서 일을 보느냐에 따라 "부서원들이 실행에 얼마나 집중할 수 있는지"가 결정된다. 리더의 잘못된 사고가 모든 부서원의 쓸모없는 고생을 만들수도 있다.

소프트웨어의 “Soft(변경이 쉬운?)”가 잘못된 의미로 해석되듯, “Ware(물건)”으로 개발자를 대하는 태도는 결코 좋은 코드(품질이 높은 코드)를 만들어낼 수 없다. 물건은 아무나 만들 수 있고 다른 동일한 물건으로 대체될 수 있지만, 좋은 개발자를 찾기는 쉽지 않다. 개발에서 좋은 코드를 만나는 즐거움은 자주 있는 경험이 아니며, 자신이 만든 코드가 그렇게 되기를 바라는 것도 사실상 어렵다. 좋은 코드를 만들기 위해서는 좋은 사람을 뽑고 좋은 환경을 제공해 주어야 하지만, 무엇이 좋은지도 모르는 사람들이 지배하는 공간에서는 “개발자 문화”가 성숙할 수 없다. 품질은 강요로 달성 가능한 목표가 될 수 없으며, 신뢰와 협력으로 이루어진다. 품질을 원한다면 품질 마인드(Mind)가 성숙할 수 있는 환경을 만들어 주어야 하지만, 아직도 비판과 희생만 찾는 태도를 가진 환경에서는 “품질이 높은 코드”란 결코 만날 수 없는 이상적일 뿐이다.

# [ 공부를 하자. ]

개발자로 일하면서 서서히 나이가 들어가면 배움에 대해서는 게을러지는 게 사실이다. 새로운 기술이 무엇인 있는지는 가끔 들여다보는 "전자신문"에서 보게 되는 일이 많고, 최신기술 트랜드에 대한 감은 점점 떨어져 간다. 우리가 일하는 분야는 전 세계가 가장 많은 관심을 두고 있는 분야고, 새로운 기술이나 개념들은 하루가 멀다하고 끝없이 나온다. 어떻게 보면 참 어려운 직업 중에 하나임에 틀림없다. 하지만, 그렇다고 지금 와서 다른 분야의 일을 찾기도 어렵다. 이미 전문가라는 타이틀을 달고 있는 사람들이 그 분야라고 없지는 않기 때문이다.

공부해야 한다는 사실은 항상 느끼지만 책을 사서는 끝까지 다 보기가 어렵다. 자신이 부족함을 아는 것이 시작이라고 하지만, 그 부족함이 너무나 많을 경우에는 그저 한숨만 나올 뿐이다. 그렇다고, 시간이 여유로운 것도 아니다. 맡겨진 일을 하다 보면 밤 늦게까지 일하기가 일쑤고, 피곤함에 지친 눈으로 책장을 넘기는 것은 이기지 못하는 졸음과 맞서 싸워야 하기 때문이다. 그렇다면 이런 것들을 이겨내면서 그래도 봐야 하는 것이 있다면 어떤 게 있을까? 그리고, 그것이 정말 우리의 인생에 중요한 것일까?

소프트웨어 개발자에 한정해서 이야기한다면, 권하고 싶은 것은 항상 “기본기”에 있다. 기본은 역시 "좋은 코드를 만드는 방법"에 기초한다. 소프트웨어 공학도 공부하기에 좋은 분야 중 하나다. 품질이 높은 소프트웨어를 만드는 틀은 바뀌지 않지만, 틀에 들어가는 내용이 바뀌기 때문이다. 틀이란 만든는 방법을 말하며, 내용은 해결해야 할 문제점이다. 따라서, 나중에라도 지속해서 사용될 수 있는 지식은 틀(Frame)이지, 틀로 찍어내는 빵은 언제고 바뀔 것이기 때문이다. 물론, 그렇다고 틀이 항상 동일한 것은 아니다. 틀도 시대의 흐름에 따라 바뀐다. 그래서 기본을 더 튼튼히 하라는 것이다. 기본은 모든 것의 근본에 해당한다. 틀을 만들기 위한 기본도 마찬가지다.

직급이 높아진다고 해도 항상 코딩을 가깝게 두어야 한다. 시간이 주어진다면 언제든 코딩할 수 있는 능력은 지니고 있어야 한다. 우리가 하는 일은 항상 실행되는 결과물을 전달하는 것이기 때문이다. 실행할 수 없는 결과물은 누구나 만들어내지만, 실행되는 결과물은 우리 소프트웨어 개발자만이 만든다. 아무리 기술이 발전해도 실행되는 결과물을 만들기 위한 중간 과정에서 사람의 역할은 바뀌지 않는다. 효과적인 해결책을 논리로 채워 넣고, 결과물을 만들기 위한 틀 속에 담아서 나오는 최종 산출물은 우리가 개발자이기 때문에 가능하다.

기본기를 익히는 일은 지속해서 추구되어야 할 가치다. 뛰기 위해서는 먼저 걸어야 하고, 걷기 위해서는 기는 연습부터 해야 한다. 기어다니기 위해서는 당연히 뒤집기부터 해야 한다는 사실을 우리의 이미 아이들로부터 충분히 배웠다. 우리의 몸도 어릴 때부터 이런 사실을 충분히 알고 있을 것이다. 학교 다닐 때 본 교과서에 나오는 이야기가 기본기라면, 기본을 무시한 응용이 가능할까? 하지만, 소위 말하는 "교과서적인 이야기만 한다."라는 태도를 보이는 사람들은 자신의 경험이 교과서보다 우선한다고 믿는 것 같다. 무엇이 좋은 코드를 만드는 핵심인지 알지 못하면서 빨리 뛰기만 바래는 것과 같다. 이것은 마치 교과서도 제대로 완독하지 못한 상태에서 수능 점수가 잘 나오기를 기대하는 것과 같다. 세상이 불편한 것은 기본적인 것들이 지켜지 않고, 기본을 지키는 행동이 쉽게 무시되기 때문이다. 기본기는 어떤 일을 하더라도 필요한 가장 중요한 것이라는 사실을 결코 잊지 않아야 할 것이다.

# [ 즐기는 자(논어의 한 구절)? ]

"알기만 하는 자는 좋아하는 자만 못 하고, 좋아하기만 하는 자는 즐기는 자만 못하다(知之者는 不如好之者요, 好之者는 不如樂之者니라)"라는 말이 논어에 나온다. 스포츠 중계를 보면 이런 말로 선수들이 경기를 즐겨야 한다고 이야기 하지만, 정말 그렇게 즐기기만 해서 경기에 이길 수 있을까? 아니, 사실 즐길만한 경기가 있기는 할까? 자신의 경기를 즐기는 사람은 그것을 위해서 자신의 모든 것을 희생하며 준비한 사람이다. 그것을 통해서 자신의 생의 의미를 발견할 수 있으며, 기쁨을 얻을 수 있는 사람일 것이다. 즐기기 위해서는 아는 단계와 선호하는 단계를 넘어서 의미를 찾을 수 있는 단계로까지 가야만 한다.

일을 즐기기는 정말 어렵다. 스트레스 자체가 흥미를 유발하도록 하기 위해서는 정말 특이한 성격을 가져야 한다. 스트레스는 스트레스일 뿐이고, 그것을 푸는 방법이 어떤 것인가만 다르다. 차라리 스트레스가 없는 직장을 만드는 게 더 빠르고 효과적이지 않을까? 하지만, 현실적으로 그건 불가능하다. 따라서, 적당한 스트레스 상황은 항상 있는 것이고, 한도를 넘지 않는 정도에서 타협하고 풀어나가야 한다. 여기서 "한도를 넘지 않는 정도의 스트레스"가 중요하다. 긴장감을 주는 정도의 스트레스라고 한다면 그것을 이길 수 있는 방법도 찾을 수 있지만, 그것을 넘어서는 스트레스는 사람의 정신을 피폐하게 만들며, 회피를 통해서 자신을 보호할 수 있는 방법만 찾도록 만든다.

적절한 긴장은 항상 필요하지만 과도한 긴장은 사람을 방어적으로 만든다. 과제 일정의 막바지에 도달할 수록 점점 더 많은 압박이 내외부로부터 주어진다. 일정을 어길 시에는 책임을 묻겠다는 말과 함께 좋은 고과를 받는 것도 물 건너갈 것이라고 이야기한다. 하지만, 정말 그럴까? 사실 일정을 정확히 준수하는 프로젝트는 그렇게 많지 않다. 그리고, 일정을 제대로 맞춘다고해서 좋은 고과가 주어지는 것도 아니다. 그런 말들을 하는 것 자체가 무의미 하며, 사람들의 일할 의욕을 저해할 뿐이다. 위협을 한다고 좋은 제품을 빨리 만들 수 있다면, 이미 회사는 비난과 폭언이 난무하는 곳으로 바뀌었을 것이다. 그리고, 모든 사람이 고과를 중요하게 생각하는 것도 아니다. 연봉에 반영되기는 하겠지만, 돈이 전부라고 생각하는 개발자는 거의 없다. 먹고 살 정도만 되면 자신의 역할이나 책임, 하고자 하는 것들을 더 중요하게 생각한다.

일정이 중요한 것은 임원들이 매년 새로운 계약을 해야하기 때문이다. 정확히 말하면, 과제에 대한 지표를 통해서 재계약 여부를 결정하겠다는 경영진의 압력 때문일 것이다. 우리에게는 그런 사람들의 재계약 여부가 중요한 것이 아니라, 과제의 성공(상업적인 성공)으로 인한 자부심과 금전적인 보상 및 승진이다. 따라서, 과제의 내용이 중요한 것이지 일정만 지키는 것은 의미없는 일이다. 일정을 맞춘다고 품질이 나쁜 제품이 시장에서 "Sold Out"될 만큼 팔리지는 않는다. 일정을 맞추지 못해서 상품이 실패하는 것이 아니라, 고객의 요구를 만족시킬 수 있는 기능과 품질이 제공되지 못해서 팔리지 않는 것이다. 빨리 만든다고 모든 제품이 성공한다면 왜 미리 시작할 생각은 하지 않았는지 스스로에게 질문해야 할 것이다.

즐기려면 오히려 과제의 스트레스를 이겨내고 자신이 완성한 제품으로 눈길을 돌리는 편이 좋다. 자신감으로 만든 제품이 시장에서 얼마나 팔리는지 매일 매일 알아보고 싶을 만큼, 관심이 가는 제품을 만드는 것이 즐겁지 않을 리가 없다. 쪼아대는 일정 때문에 간과한 부분이 계속 마음에 걸린다면, 자기의 지갑을 열어서라도 사고 싶은 물건일 리가 없다. 즐거움이란 열정이 만들어낸 결과물에 대한 자부심으로 표현된다. 그 표현할 방법을 찾지 못 하는 일이란 결국 구속일 수밖에 없고, 논어에 나오는 "즐기는 자"가 아닌 목에 줄을 걸고 끌려간 물가의 당나귀 밖에 될 수 없다. 즐기는 자는 스스로 결정한 일하는 방법과 시기를 조절할 수 사람이며, 그것이 불가능 하다면 목마르지 않아도 억지로 물을 마실 수밖에 없는 사람이 된다.

어떤 사람들을 데리고 일하고 싶은가? 마지못해 시키는 대로 하는 척하는 사람들인가? 아니면, 자신이 하는 일에 자존심을 걸고 열정을 다하는 사람들인가? 당연히 당신이 원하는 사람들은 후자일 것이다. 하지만, 당신의 관리 방식이 전자만 만들어 낸다면, 당신이 잘못된 것일까 아니면, 당신이 데리고 일하는 사람들이 잘못된 것일까? 민주주의에 익숙한 당신은 쉽게 알 수 있을 것이다. 선수가 문제라면 선수를 바꾸지만, 팀이 문제가 생기면 감독을 바꾸는 것이 당연하다. 즐기는 자가 사무실을 비우고 나면, 남는 것은 당신이 만든 세상을 즐기는 당신과 당나귀만 있을 것이다.

# [ 고과 면담 ]

대부분의 회사는 한 해를 마무리하기 전에 구성원에 대해서 올해 일했던 것에 대한 평가룰 진행한다. 평가가 공정해야 한다고 이야기 하지만, 반드시 일한 결과만이 고려되는 것은 아니다. 고과 면담은 반드시 해야 한다고 말 하지만, 대부분의 고과 면담은 대화는 없고 일방적인 통보가 있을 뿐이다. 시스템적으로 할 일들을 입력하고, 그에 따른 평가와 면담을 통해서 산출물에 대한 객관적인 평가를 진행해야 하지만, 시스템과 사람이 따로 노는 것이 사실이다. 기계가 사람을 평가하는 것이 아니라 사람이 사람을 평가하기에 주관이라는 것을 빼놓고는 이해가 안 된다. 객관성을 가장한 주관적인 평가가 이루어지는 지는 것이 현실이다.

실제로 본인도 개발자로 일하면서 제대로 고과 면담을 한 경험이 거의 없다. 지난 세월 동안 회사에서 일하면서 경험했던 고과 면담은 대체로 일방적으로 주어지는 등급이었고, 그 등급에 맞춰서 연봉을 받을 뿐이었다. 몇 번이고 이의를 제기했지만, 사실상 이미 평가가 끝난 고과를 바꾸기는 쉽지 않다. 그리고, 과제의 결과에 대한 것보다는 관리자의 선호도에 따라 주어진 결과만을 받아들 뿐이었다. 평가가 그냥 거기서 끝나면 상관없겠지만, 내년에 받아야 하는 연봉까지도 결정된다면 이건 중요한 문제다. 이렇게 한번 두번 물러서면 결국 자신의 노력에 대한 공정한 평가를 받을 수 없다는 경험만 가지게 되고, 회사에 대한 믿음은 서서히 약해지게 된다.

면담은 면담하겠다고 이야기를 하고 시작해야 하고, 그 해에 한 일의 결과물을 가지고 평가를 해야 한다. 다면적인 평가를 하는 것은 그 사람의 잠재능력 및 회사에 대한 긍정적인 영향도에 관련된 것이며, 이 또한 주변과의 협업이나 새로운 문화에 대한 적응도 등을 알 수 있는 부분이다. 고과를 공정하게 하기 위해서는 중간과정도 필요하다. 주기적인 면담과 코칭이 있어야 그나마 사람에 대해 정확한 평가를 할 수 있다. 주기적인 피드백(Feedback)이 없이 나중에 시간이 급해서 하는 면담은 마치 일정에 쫓겨서 대충 마무리하는 과제와 크게 다르지 않다. 평가의 오류는 여기서도 항상 발생할 수밖에 없다.

자신이 받는 올해의 성적표를 가볍게 생각해서는 안 된다. 또한, 평가자는 객관적으로 자신의 선호도(편향)와 분리해서 평가를 해야 한다. 법 앞에 모두가 동등하다는 말이 진실로 받아들여지는 세상이 아닌 이유는 남들과 다른 부당한 대우를 받았다는 느낌이 원인이다. 따라서, 작은 부분이라도 부당하다고 생각되는 부분들은 지워 나가야 한다. 공정하기 위해서는 관심을 가지고 지켜봐야 하며, 잘못된 지시로 인한 것이 아닌지도 따져봐야 한다. 평가자의 책임인 부분들에 대해서 피평가자에게 미루는 것은 올바른 행동이 아니다. 편향된 평가는 지금 당장은 위기를 모면할지는 몰라도 장기적으로 함께 일해야 하는 사람들의 믿음이 사라지게 만든다.

관리자는 팀원의 잘못에 대해서는 자신이 책임을 져야 하고, 팀의 성과에 대해서는 팀원들과 공정하게 나누어야 한다. 상호 신뢰라는 바윗돌도 작은 자갈들이 사이사이를 채워주고 있어야 흔들림이 없다. 흔들림 없는 조직은 큰 일만이 아니라 작은 일도 공정하게 처리할 수 있는 기준과 실행이 필요하다. 부당하게 대우를 받았다는 느낌은 타인과의 비교에서 생기는 상대적인 박탈감이며, 자신이 수긍할 수 없는 답을 듣는다면 당연히 결과를 받아들이지 못한다. 기준이 먼저 제시되고 실행은 항상 기준을 충족시키도록 해야한다. 적어도 자신의 성적표가 원하는 것이 아닐지라도 다음 고과에서는 무엇을 해야할지 알 수 있도록 긍정적인 피드백을 줄 수 있어야 할 것이다.

# [ 이공계를 기피하는 현상? ]

요즘 대학을 진학하고자 하는 학생들은 이공계를 꺼린다고 한다. 똑똑한 아이들은 전부 의사나 변호사 등의 "사"자가 붙은 직업을 가지기를 선호한다고 한다. 누군가의 잘못이라고 일방적으로 이야기하기에는 조금 이상하다. 사실 대부분의 대학 진학을 앞둔 부모들의 생각이 자신의 경험에 기초한다고 할 때, "돈"을 많이 벌어서 편하게 애들이 살기를 원할 것이다. 힘들게 새벽부터 지하철이나 회사 버스를 타고 출근해서 늦게 퇴근해서 돌아오는 자녀를 보고 싶지는 않을 것이다.

오래 전의 기억을 떠올리면, 누군가 "너는 커서 무엇이 될래?"라는 질문에 "과학자, 대통령, 군인" 등등을 이야기했다. 왜 그런 사람이 되고 싶은가를 물었을 때 멋져 보인다고 했던 것 같다. TV에서도 애니메이션에서 주도적인 역할을 하는 사람들은 과학자이거나 한 나라의 지도자, 혹은 사선에 뛰어든 군인이었다. 하지만, 나이가 들어서 보게 되는 드라마에서는 대부분 의사, 변호사, 디자이너, 젊은 회사의 CEO나 실장 등이 주연들의 직업으로 나온다. 그런 사람들이 세상을 움직이는 힘으로 암묵적으로 받아들이고 있다는 것을 반영하고 있다.

이공계를 나온 개발자들은 가끔 9시 뉴스에 나와서 "이번에는 뭔가를 개발했어요."하고 고리타분한 이야기만 몇 마디 인터뷰하는 것으로 끝난다. 이미 우리는 무의식적으로 이공계를 나오는 것은 흥미있는 일이 아니라고 생각하고 있는 것이다. 얼마나 열심히 야근하고 주말도 잊은채 근무해서 무엇을 개발해 봐야 돈은 결코 따라오지 않는다. 심지어 동종 업계로 이직하는 것도 어렵게 만든 법도 있다. 적어도 2년간은 자신이 경력과 무관한 일을 해야지 고소를 당하지 않는다. 컴퓨터 프로그래머들은 남들과 다른 이상한 사람들처럼 취급받으며, 소통할줄도 모르고 옷도 멋지게 입지 않는다. 두꺼운 안경과 어리벙한 얼굴을 하는 사람들이 주로 맡는 역할로 취급당한다.

이런 상황에서 이공계를 가야 한다고 아이들에게 이야기할 부모들이 있을까? 이공계를 가서 어릴 적 자신의 꿈을 현실로 만들어보고 싶은 아이들이 있을까? 고생하고 힘든 것만을 보게 되고 들리는 이야기도 암울하기만 하다면, 누구도 하기 싫은 직업이 될 것이다. 우스개소리로 소프트웨어 개발자의 삶은 4D(Difficult, Dirty, Dangerous, Dreamless, 이 중에서 Dreamless가 가장 강하게 연상된다.)에 속한다고 이야기하는 것이 당연하지 않을까? 이미 꿈보다는 현실의 삶이 중요한 의사결정 요소라면, 실질적인 삶의 질에 대한 개선이 시급하다. 삶의 질이 반드시 돈이나 명예와 같은 것에 달려있지는 않다고 이야기 하지만, 생존 자체가 문제인 경우에는 반드시 해결되어야 한다. 한 회사에 다니다가 다른 동종의 회사로 가지 못하도록 만드는 법 자체가 생존의 문제가 아니라면, 우리가 가지고 있는 기술은 쉽게 바꿀 수 있는 아무것도 아닌 것이 되고만다.

이미 기술자에 대한 경쟁력이 중국이나 인도보다 못하다는 말을 듣는 상황이지만, 그들보다 더 열심히 일하는 것이 대한민국의 개발자다. 우리는 개인이 일하는 것이 아니라 조직적으로 일한다. 개인은 선진국이나 인구가 많은 나라의 개발자보다 못할지는 모르지만, 조직적으로 일하고 밤 늦게까지 수고를 마다하지 않는 것은 한국 개발자뿐이다. 주말이고 공휴일이고를 가리지 않고 언제나 해야할 일이 있고 호출을 받으면 나와서 일하는 것도 한국 개발자만의 특징이다. 이렇게 열심히 일하고 만들어가는 사람들에게 뭔가 꿈을 가질 수 있는 것을 주는 것이 필요하지 않을까? 적어도 그 개발자의 자녀들이 아빠나 엄마를 자랑스럽게 생각할 수 있는 분위기를 만들어야 하지 않을까?

이런 상황이 된 이면에는 국회의원이나 정부, 기업의 높으신 분들만 책임이 있다고 말해서는 안된다. 책임은 그들에게 있는 것이 아니라 개발자 스스로에게 있다. 자신의 품위를 유지하지 못한 책임을 남에게 묻는 것은 옳지 않다. 존경받는 소프트웨어 개발자가 외국인만 거론하는 것도 마찬가지다. 우리 중 누군가는 스티브 잡스가 될 수도 있지만, 누구도 그와 같은 사람이 되려고 노력하지 않는 것도 사실이다. “아니오”라고 말할 수 있는 분위기를 만들지 못하는 책임은 외부에 있지만, “아니오”라고 이야기하는 것은 개인의 선택이다. 지켜야 할 소중한 가치가 있다면 희생할 각오 정도는 당연히 되어 있어야 한다. 그렇지 않다면 남이 주는 밥그릇만 지키는 애완견(?)이 될 뿐이다. 물론, 그 밥그릇도 시간이 지나면 다른 사람에게 돌아갈 것이 분명하다.

# [ 개발자를 흥이나게 만드는 것 ]

개발자가 일을 잘 하고 열심히 하게 만드는 가장 핵심적인 것은 무엇일까? 아마도 대부분의 높으신 분들은 “일에 대한 보상(Reward)”이라고 생각할 것이다. 사실 보상은 너무 먼 이야기고, 개발자의 하루 하루의 생활에 영향을 주는 것은 다른 곳에 있다. 보상이 주어지는 것은 주로 일이 완료되었을 때와 새로운 고용계약을 할 경우로 한정되어 있기 때문에 일상 생활에서 효과가 지속될 것이라고는 생각할 수 없다. 순간적인 높은 보상은 그것을 받을 때 잠시 동안 효과가 유지될 뿐이지만, 개발자의 일상은 일년 365일 반복적이기 때문이다. 보상의 강도가 높아도 1년간 효과가 지속되지는 않는다.

가장 효과적인 방법은 "앞으로 나아간다는 느낌(생각)"이다. 개발자라면 다 알겠지만, 자신이 성장이 멈추고 있다고 느낄 때가 가장 초조하다. 남들은 벌써 저 만큼 가 있는데 자신의 능력은 예나 지금이나 변함이 없다고 느낀다면, 새로운 일을 찾아볼 수밖에 없다. 하루 하루가 같은 일의 반복이라면 매너리즘에 쉽게 빠져들게 되며, 더이상 새로운 발전을 시도할 일도 없게 된다. 그냥 그저 그런 개발자로 남게 되더라도 지금의 자리만 지킬 수 있다면 새로운 시도를 할 이유도 없어진다. 결국 능력있거나 도전적인 개발자는 다른 기회를 찾아서 떠나게 되며, 남은 개발자를 가장한 사람들은 자리 싸움으로 서로의 눈치만 보게 된다.

그렇다면, 개발자가 자신의 능력을 충분히 발휘해서 열심히 일 할 수 있게 하는 것은 어쩌면 가장 쉬운 일 일지도 모른다. 예를 들어, 지금 하는 일이 어떤 의미를 가지고며 왜 중요한지, 개인의 커리어(Career)에 어떤 도움이 되는지를 지속해서 알려주는 것이다. 또한 일을 통해서 개인이 성장하도록 항상 새로운 것에 대해서 열린 자세로 대화(의견을 비판하지 않고, 공평한 시간을 배분해서 대화)하는 것이다. 지금까지와는 다른 방법론을 개발 과제에 동원해 보기도 하고, 비창의적이고 생산적이지 않은 것들을 자동화를 시켜서 개선하는 것도 한 가지 방법이 될 수 있다. 기존에 미비했던 점들을 인프라(Infra)나 프로세스 개선으로 보완하고, 개발자가 자신의 핵심 업무에 좀 더 몰입할 수 있도록 만들어 주는 것이다.

이공대를 졸업한 사람이 단순하다고 이야기하는 것은, 그 사람들이 열정을 가지고 한 가지 일에 매달려 장시간의 집중을 할 수 있는 역량이 있다는 뜻이다. 그 역량을 집중할 수 있는 환경을 잘 만들어 준다면 흥이 나서 일하는 것은 당연하지 않을까? 매번 "간담회"나 "회식"을 가장한 술판을 벌인다고 그런 것들이 될리가 없다. 사람은 지속해서 자신의 능력을 키워 나가고 싶어하고, 이를 잘할 수 있도록 환경과 지원을 아낌없이 해주는 것이 관리자의 역할이다. 얼마간의 물질적인 수단으로 “열정” 끌어내려고 한다면, 지속하지도 않고 일관성도 없는 잠시 동안의 즐거움만 추구하게 만들 것이다. 만족의 강도는 점점 높은 것을 원할 것이고, 결국에는 새로운 만족을 찾기 위해서 회사를 떠날 수밖에 없을 것이다. 성장은 개발자 만이 아니라 모든 사람이 원하는 삶의 목적인 것이다.

# [ 장인정신 ]

장인을 만드는 것은 "하루 하루 자신이 맡은 일을 충실히 하는 자세"에서 나온다고들 한다. "충실히 한다"의 뜻은 자신이 만든 것들이 본래의 쓰임새로 잘 사용되고, 남들이 그것을 통해서 도움(기쁨)을 얻을 수 있다는 것이다. TV사극을 보면 장인들은 남루한 옷을 입고 보잘것없는 삶을 유지하지만, 남아 있는 역사의 유산들은 고스란히 그들의 손을 통해서 만들어진 것들이다. 당시의 시대 상황에서는 몸을 써 만드는 것들이 사소하거 하찮게 여겨졌을지 모르지만, 그러한 “사소함에 깃든 세세함”이 오늘 날에도 그 가치를 인정받을 정도로 오랜 시간 동안 변함이 없다.

하지만 소프트웨어 개발자의 일상은 장인정신을 만들어 가는 것이 거의 불가능하다. 일단 동작하는 것이 만들어지고나면, 곧장 다른 해야 할 일들이 이미 쌓여있다. 그것들을 빨리 처리하지 않으면 무능하다고 생각할지도 모른다. 이런 현상이 생기는 이유는 일이 완료되었다는 기준이 명확하지 않으며 사람들마다 다르다는 점이다. 하나의 구현이 완료되기 위해서는 "정리(혹은 Refactoring이라고 하는 과정)"가 필요하다. 얼마나 정리할 것인가가 문제이기는 하지만, 어쨌든 만족할 만큼의 정리를 해야 한다. 또한, 당연히 해야 할 일에서 테스트가 빠져선 안 된다. 즉, 개발한 코드와 그것을 테스트할 수 있는 코드가 다 갖춰져야 완료되었다고 봐야 한다. 덧붙여 얼마나 코드가 테스트되었는지를 확인하기 위해서 테스트 커버리지(Test Coverage)를 측정할 수 있는 툴도 돌려봐야 한다.

"캠프장에 들어갈 때 보다 더 깨끗하게 만들고 나와야 한다"는 “보이스카웃 원칙”이라는 것이 있다. 리팩토링 하는 과정에서 이런 원칙들을 지키려고 한다면, 지속적인 개선이 무엇인지 잘 알 수 있다. 한 번에 모든 것을 만족시킬수는 없다. 말 그대로 "하루 하루 자신이 하는 일에 대해서 충실"하게 할 때 개선 활동은 습관이 될 것이다. 빨리하고자 하면 오히려 더 늦어진다. 게으름을 피우지 않는 느림은 미리 계획된 꾸준함이다. 그 꾸준함이 사소함을 사소함에 그치지 않게 만들어 준다. 인내를 가지고 조금씩 변화를 가져가는 것이 성공의 지름길이다. 모든 것을 한 번에 다 만족시키는 것은 세상에 없기에 천천히 가야한다.

과제를 하게 될 때는 지나치게 튀는 팀원을 주의해서 본다. 일을 빨리하는 게 능사가 아니라, 다른 사람과의 협업이 과제 성공의 중요한 요인이기 때문이다. 대체로 일을 잘하는 개발자와 잘 못 하는 개발자의 차이가 열 배 정도 날 수 있다고 하는데(-5와 +5사이의 차이), 그 차이 중의 일부를 다른 사람과 공유한다면 오히려 전체적인 상향 평준화를 이룰 수 있게 된다. 잘하는 사람이 항상 많은 일을 맡지 않아도 되기에, 성과를 많이 내는 사람은 더 많은 여유를 가질 수 있다. 일이 잘 분산되어 과제를 성공적으로 마칠 확률도 높아지게 된다. 이런 조직을 이끄는 사람은 자신의 팀원들이 고르게 발전해 나간다는 사실을 자랑스럽게 생각할 것이다. 모두가 장인이 되어야 할 필요는 없겠지만, 장인이 되려고 노력하는 사람들이 있는 회사가 다닐 맛이 날 것이다.

소프트웨어는 하나를 잘 만들면 무한 복제가 가능하다. 과거의 장인들은 하나를 만들어 소수의 사람들을 만족시켜 주었지만, 소프트웨어를 만드는 장인은 하나의 제품으로 무한한 사람을 만족시켜 줄 수 있다. 장인은 자신이 만드는 제품에 대해서 세세한 부분도 놓치지 않으며, 완벽에 가깝게 구현해 낼 수 있는 사람이다. 품질이 높은 소프트웨어가 시장에서 성공하지 못할 이유는 없다. 무한정 시간이 주어지지는 않을지도 모르지만, 그렇다고 아무렇게나 만들어서도 안된다. 적어도 자신이 만든 코드라면 작은 부분도 놓치지 않고 꼼꼼하게 살펴야 한다. 품질을 높일 수 있는 방법이라고 생각되는 모든 방법은 적용해보고 타당성을 검토해야 할 것이다. 장인은 하루 아침에 만들어지지 않는다. 완벽에 가깝게 도달하려는 노력을 꾸준히 한다면, 어느 순간 타인의 눈에 아름다운 모습으로 남게 될 것이다.

# [ 듣고 말하기 vs. 말하고 듣기 ]

제대로 들어야 정확하게 대답할 수 있다. 또한, 제대로 물어야 정확한 답을 들을 수 있다. 회사에서 "소통"이 중요한 화두로 이야기되는 것은, 역설적으로 이야기하면 제대로 된 소통이 안되고 있다는 신호다. 어떤 회사가 표어로 내건 것이 있다면, 그것이 그 회사에서 가장 잘 안 되고 있다는 의미다. "소통"은 벽을 허물어가는 것인데, 제대로 묻지도 않고 제대로 대답하지 않는다면, 상호 불신의 벽은 넘어서기 힘들다. 말하는 사람은 듣는 사람을 생각하지 못하고, 듣는 사람은 말하는 사람의 말에 귀를 기울이지 않는다. 믿음이 부족한 상황에서는 어떤 대화도 제대로 듣고 들리지 않는다.

서로 간에 신뢰가 부족한 상태에서 나누는 대화는 호통을 치거나 뭔가 껀수를 찾으려는 태도에서 나온다. "털어서 먼지 안 나오는 사람 없다"는 속담처럼, 깨려고 작정하고 물어보는 질문들을 일일이 대답하는 것은 거의 불가능에 가깝다. 완벽하게 대답한다면 또 다른 말도 안 되는 이유로 깨려고 덤벼들 것이다. 소프트웨어 개발 책임자에게 사용하는 칩(Chip)의 상세 스펙을 암기에 가깝게 요구하는 것이 무슨 의미가 있을까? CPU 클럭(Clock) 주파수, L1/L2 캐쉬의 크기, 명령과 데이터 캐쉬의 크기, 사용하는 램(RAM)의 종류 및 크기와 동작 주파수, 주변기기의 종류 등등 그런 것들을 다 암기하는 것이 소프트웨어 과제를 성공시키는 것과 무슨 관련이 있을까? 그런 것들을 다 안다고 해서 개발을 더 잘할 수 있다면 모르겠지만, 그런 정보들은 실무 개발자가 가져야 할 상식 수준의 지식이고, 과제를 맡은 책임자에게는 물어볼 필요도 없는 질문이다.

하지만, 이런 질문들을 던지는 이유는 뭘까? 사실 그런 질문은 본인이 소프트웨어 개발에 대해서 모른다는 것을 증명할 뿐이다. 잘 모르니 자신이 예전에 했던 일에서 생각나는데로 물어볼 수밖에 없다. 소프트웨어 개발에 관련된 질문을 할 수 없기에, 대답하기 곤란한 것들을 찾아서 질문을 만들고 있을 뿐이다. 과제의 일정을 정확히 암기하는 것도 아무런 의미가 없다. 단계별 정확한 일정을 전부 암기하고 있는 것이 과제에 무슨 도움이 될까? 차라리 그런 게 궁금하면 그냥 문서를 보거나 시스템에 접속해서 결과물의 진행 현황을 보는게 옳지 않을까? 그런 정보들을 자주 요구하면 개발자는 그런 내용만 프린트해서 수첩에 넣고 다니거나, 혹은 회의에 참석할 때 노트북을 가지고 가서 이야기할 수밖에 없다. 우리의 양 어깨 위에 있는 두뇌는 그런 것들을 넣어두기에는 너무 아깝다. 좀 더 생산적이고 창조적인 것 만으로도 충분히 번잡하다.

상대가 하는 이야기를 들어보려고도 하지 않고 그저 자신이 생각하는 것만 이야기한다면, 듣는 사람들은 점점 대화에서 초점을 잃고 다른 생각을 하게 된다. 대답도 건성으로 할 뿐이고, 이미 결정된 듯한 분위기에서 의견을 내는 것을 의미 없는 일로 받아들이게 된다. 회의에서 순서를 정해서 이야기하라고 할 때, 이미 서로 간의 불통은 탁자 밑에서 또아리를 틀고 누군가의 발을 물려는 독사처럼 눈만 두리번거리고 있을 것이다. 제대로 된 의견을 듣고 싶다면 자신의 말을 줄여야 한다. 남들이 말하도록 만들려면 잘 듣는 것이 먼저다. "시키는 대로만 하라니까!!!"라고 이야기한다면, 반대로 "시키지 않은 일은 절대 하지 마!!!"라고 들린다. 당신이 이야기하는 것은 건성으로 하고, 당신이 이야기하지 않은 것은 절대 하지 않게 될 뿐이다. 당연히 업무의 성과는 낮아질 수밖에 없다.

다른 사람의 의견을 알고 싶다면 당신부터 바꿔야 한다. 솔직한 이야기를 듣고 싶다면, 당신 스스로가 남들에게 솔직해야 한다. 타인을 존중하지 않으면 솔직한 이야기는 들을 수 없다. 제대로 된 질문을 하기 위해서는 스스로에게 그 질문을 먼저 해야 하고, 그것이 정말 의미가 있는지 스스로 먼저 대답해 봐야 한다. 의미 없이 던져진 말 한 마디가 여러 사람에게 피해를 주게 된다면 오히려 하지 않는 것이 좋다. 다른 사람의 행동을 변화시키고 싶다면 신뢰부터 만들어야 할 것이다. 호통과 강압적으로 주어지는 업무지시를 통해서 자발적인 행동을 기대하는 것은 무리다. 스스로 하겠다는 의지가 없다면 “좋은 회사(Good Company)”정도는 그럭저럭 될 수 있을지 몰라도, 결코 “위대한 회사(Great Company)는 되지 못할 것이다. 구성원이 스스로 만들어가는 규율이 없는 회사는 콩나물시루 같은 곳에서 주어진 양분만 먹고 자라는 기형적인 식물 밖에는 만들지 못한다.

# [ 존중 받으려면 먼저 존중해야 한다. ]

흔히 남들에 대한 험한 이야기를 하면서 자신은 전혀 잘못이 없는 듯이 말한다. 하지만, 원인이 없는 결과가 있을까? 자신이 가지고 있다고 믿는 권한은 직급으로 나오는 것이 아니고, 사람들의 신뢰에서 나온다는 사실을 알아야 한다. 무턱대고 지시만 해서는 명령받고 있는 사람의 열정을 절대 끌어내지는 못하며, 그렇게 만든 결과물에 대해서는 책임 의식도 가지지 않는다. 자신이 옳다고 믿는다고 해서 모든 사람이 그것을 옳다고 생각하지 않는다는 것을 알아야 한다. 팀이 동의한 문화를 만들 생각을 해야지 강요로 주어진 것들은 팀원의 행동에 아무런 변화를 주지 못한다.

사람들을 잔업시간으로 평가하거나 개발에서 발생할 수 있는 오류(Defect)의 수로 성과를 논하려고 한다면, 일하지 않아도 좋다는 것을 간접적으로 평가받는 사람에게 말하고 있는 것이다. 항상 잔업을 해야 한다면 시간에 대한 효율을 포기한다는 것이며 그 시간을 쓸모없이 채우려는 생각만을 강조하는 것이다. 시간에 대한 효율을 극대화할 이유가 없는 것이다. 주말에도 나와서 일해야 한다면 주말 전에 일을 끝내는 것은 의미가 없다. 결국 비용은 늘어나지만 효율은 오히려 더 떨어지게 된다. 오류의 수를 그 사람이 가진 자질과 연관시킨다면 일하지 않는 편이 오히려 더 좋은 평가를 받을 수 있다는 논리가 된다. 실수를 할 수 있다는 것이 부정된다면 시도 자체를 할 이유가 없다. 코드를 작성하지 않으면 오류는 발생하지 않는다.

소프트웨어 개발은 시스템을 구성하는 다양한 요소들의 역할과 책임으로 이루어진다. 특정 부분에서의 오류가 다른 부분에서 나타나는 것이 흔한 상황이며, 그런 오류들은 좀 더 시스템을 잘 이해하는 사람들이 맡아서 처리하는 경우가 많다. 이럴 때 그 오류를 일으킨 원인보다는 그 오류를 제거한 사람이 원인을 만든 사람처럼 생각되도록 시스템에서 관리하는 것은 바람직하지 않다. 오히려 그런 오류를 누가 얼마나 찾아냈는지를 평가하는 것이 옳다. 하지만 이것도 소프트웨어 개발자를 평가하는 잣대가 아니라 오히려 테스터 들을 평가하는 방법이다. 소프트웨어 개발자를 평가하고자 한다면 개발과정에서 얼마나 회사에서 정한 규칙(규정)을 잘 준수하는지 보는 것이 낫다.

그렇다고 모든 것을 프로세스(Process)에 담아서 강제로 규정하는 것은 솔직히 올바른 방법은 아니다. 산출물의 질보다는 양을 검사하게 될 가능성이 높기 때문이다. 또한 산출물의 질은 객관적인 평가가 어렵기 때문이기도 하다. 따라서, 이럴 경우에는 정말 필요한 결과물에 대해서만 객관적인 평가를 해야 한다. 소스 코드의 품질에 대한 평가가 중요한 핵심이다. 복잡도, 중복 코드, 역 호출 관계, 코딩 규칙(Coding Rule) 준수, 테스트의 코드 검증 범위(Code Coverage) 등이 그 방법으로 활용될 수 있다. 개발자들이 자신의 코드를 객관적으로 평가받을 수 있는 자리를 만들고, 결과를 적절하게 피드백(Feedback)해 준다면 자신의 역량이 커간다는 느낌을 얻을 수 있을 것이다. 물론 이러한 것이 고과로 이어진다면 그로 인한 스트레스도 많을 것이다. 하지만, 얼마 동안의 시간이 흐르고 회사 내부에서 이런 문화가 정착된다면, 말 그대로 코딩의 질은 확실히 그전보다 달라질 것이다. 그리고, 고과는 다른 방법을 통해서 진행하게 될 것이다.

# [ 삶의 질(Quality) ]

소프트웨어 개발자의 일이 4D 업이라고 불리기 시작한 것은 이미 오래된 이야기다. 3D에 더불어서 장시간의 근무에서 오는 희망이나 꿈이 없어지는 “Dreamless”가 더해진 것이다. 이런 현상은 점점 더 심해질 것으로 예측되며, 획기적인 개선이 없이는 당분간은 지속될 것으로 보인다. 왜 이런 현상이 발생할까? 이유는 의외로 간단하다. 소프트웨어 개발에 대한 몰이해와 투자의 부족에 기인한 인력 미확보 때문이다. 돈으로 모든 것을 해결할 수 있다는 생각은 금물이다. 어쨌든, 우리는 더 높은 가치에 초점을 두어야 한다.

소프트웨어 개발을 전문적으로 하는 회사의 경우에는 관리자 계층에 이미 소프트웨어를 전공한 사람들이 있을 것이다. 하지만, 제조와 관련된 부분에서는 아직 소프트웨어 전문 개발자로 일하다가 회사의 관리 계층까지 올라간 사람은 별로 많지 않다. 소프트웨어 개발의 특성을 이해하지 못한다는 것은 어떻게 하면 효율적으로 소프트웨어를 개발할 수 있는가에 대한 자신감이 없이, 기존에 하드웨어 개발에서 쌓은 경험으로 무작정 야근을 시키는 것이다. 야근은 그렇게 효과적인 방법이 되지 못하며, 개발 생산성을 높이기보다는 생산성은 그대로 두고 비용만 늘릴 뿐이다.

투자의 부족에 따른 적정 인력의 미확보는 어디서나 있는 일이다. 항상 제대로 쓸만한 사람이 없다는 이야기를 꾸준히 듣고 있다. 사실 소프트웨어 개발자로 일하는 대부분의 사람들이 상당히 지적인 수준을 가지고 있다고 보았을 때, 의사나 변호사 등등의 소위 전문가라고 불리는 사람들보다 훨씬 낮은 연봉으로 일하고 있다. IT부분에 대한 지원이 줄어들면 당연히 사람들은 지원하지 않게 되는 것이 인지상정이지 않은가? 고등학교 친구들 간의 모임에 나가서 누가 계산하는지 보기 바란다. 아마 대부분의 경우 소프트웨어 개발자로 일하는 사람은 드물 것이다. 대단한 우대를 해달라는 것이 아니라, 이 분야에서 일했을 때 인간다운 삶을 누릴 수 있다는 이야기를 듣고 싶을 뿐이다.

삶의 질은 다른데 있는 것이 아니라 항상 개인의 선택에 있다. 누군가 시간이 없어서 못 한다고 이야기할 때 정말 제대로 하고 있는지를 보기 바란다. 제대로 한다는 것은 일의 마무리에 달려있다. 어떤 식으로든 자신이 한 일에 대해서 올바르다고 검증할 수 있다면, 후반으로 갈 수록(혹은 다른 과제를 하더라도) 점점 더 개발속도는 증가하게 되어있다. 초반에 해야 할 일을 하지 않고 넘어간다면, 그 결과는 과제의 마무리 단계에서 나타나게 된다. 삶의 질을 결정하는 것은 시작을 어떻게 하고 어떻게 효과적으로 일을 수행할 것인가를 정하는가에 달려있다. 외부의 요인들이 호의적이지 않더라도 지금 반드시 해야 할 일을 다음으로 미뤄선 안 된다. 다음은 또 다른 일이 얼마든지 기다리고 있는 것이 현실이기 때문이다.

개발자 스스로 자신의 삶의 질을 개선하려고 노력하지 않는다면 아무도 관심을 두지 않는다. 스스로 자신의 삶이 형편없다고 생각한다면, 항상 “Yes”만 할 것이 아니라 “No”를 할 수 있는 용기를 가져야 한다. 물론 그것으로 인한 피해는 상당할 것이다. 하지만, 모두가 “No”를 할 수 있다면 이야기는 달라진다. 한 사람이 “No”한다면 그 사람만의 문제가 되지만, 모두가 “No”를 외칠 수 있다면, 정말 안되는 것이라고 인식하게 될 것이다. 무리한 일정과 너무 많은 기능, 역량이 부족한 개발자들을 통해서 얻을 수 있는 것은 “열등한 제품”밖에 없다. 모든 것이 갖추어진 상태에서 훌륭한 제품을 만들수 있다는 이야기가 아니다. 거부하지 않고 모든 것을 받아들이는 것으로는 현재의 상황을 전혀 개선할 수 없다는 것이다. 어렵더라도 누군가가 아닌 “자신”부터 나서지 않는다면 아무도 꿈이 없는 삶을 개선하려는 노력을 하지 않을 것이다.

# [ 소프트웨어 개발자의 최종 목표 ]

소프트웨어 개발자의 인생 최종 종착역은 "프라이드 치킨집"이라는 말이 있었다. 한 집 걸러 치킨 집이 많은 요즘 치킨집 사장을 하기도 그렇게 쉽지 않다. 그렇다면, 소프트웨어 개발자로 퇴직한 후의 삶으로 어떤 것이 좋을까? 아마도 많은 소프트웨어 개발자들이 솔직히 자신이 퇴직한 이후에 어떤 일을 하게 될지 모르고 있으며 계획이나 준비도 없다. 이 글을 쓰는 나 자신이 소프트웨어 개발에 벌써 20년에 가까운 시간을 보내고 있으면서도, 정말 필요한 미래에 대한 설계에는 자신감이 없다. 삶의 나머지 하반부를 써 나가야 하는 시점이 다가오는데 이렇다 할 생각이 없다면 이제는 정말 진지하게 고민해야 할 때라고 생각한다.

글쎄 잘은 모르겠지만 아직도 개발이 좋은 점은 많다. 일단 익숙하다는 것이고, 지겹지 않게 지속해서 뭔가 자극이 있다는 사실이다. 소프트웨어 기술은 나날이 발전하고 있어, 뭔가 새로운 것들이 하루가 멀다하고 나온다. 새로운 것들을 하나씩 익히는 것도 거의 불가능할 정도로 많은 것들이 혼재 되어있는 듯 보인다. 익숙한 것들은 익숙하게 되기까지의 노력과 과정이 필요하다. 학습이나 실습, 실무 등을 거치면서 만들어진 것들은 절대 한 번에 없어지지 않는다. 오히려 그 익숙함이 지나치게 될 때 사람들은 정체라는 덫에 갇히게 되며, 너무나 두꺼운 옷을 입어 제대로 운신하지 못한체 자신이 걸터앉은 자리가 영원할 것으로 생각하는지도 모른다. 이미 우물 안의 개구리가 되었다는 것이다.

뭔가 새로운 것에 대해 최고의 전문가가 되기 위해서는 "1만 시간"이 필요하다고들 이야기한다. 하루에 3시간이면 대략 10년이라는 시간이 필요하다. 하지만 새로운 것에 대한 전문가가 되기 전에 우리는 소프트웨어 개발업을 떠나야 할지도 모른다. 우리의 모든 경험과 실무에서 겪은 문제들을 덮어 두고, 누군가 운 나쁜 사람이 그것을 발견하기를 기대하며 새로운 삶으로 가게 된다. 조금만 여유를 가진다면 각자가 쌓은 경험을 후배들을 위해서 조금쯤은 두고 갈 수도 있지 않을까? 새로운 것에 대한 시간을 3시간을 주고, 나머지 하루 일과 중 5시간을 후배들을 위한 경험 쌓기에 활용할 수 있다면 어떨까? 책을 쓰거나 발표자료를 만들거나, 혹은 블로그을 통해서 자신이 이미 해결했거나 좋았던 경험들을 공유한다면 좀 더 세상에 도움이 되지 않을까?

사실 생각은 많지만 이렇게 행동하기는 쉽지 않다. 경력이 쌓일수록 맡은 일이 늘어나는 게 당연하고, 그러다가 보면 정작 자신의 삶에 중요한 일들은 뒤로 밀리고, 회사나 가족들을 위한 시간을 할애하는 데 모든 노력을 들인다. 나를 위한 시간은 나조차도 만들지 못한다. 일단 일에 대해서 어느 정도 책임을 벗어나야 한다. 자신이 하던 일들을 하나씩 후배들에게 위임해야 하고, 자신이 결정해야 할 것들만 몇 가지 남겨둔다. 일을 적게 해야 다른 일을 할 수 있기 때문이다. 그렇다고 모든 일에서 손을 떼고 있을 필요는 없다. 경험이 필요한 중요한 의사결정 사항들은 반드시 해야 한다. 기타 부차적인 일들은 다른 사람이 책임감을 느끼고 일 할 수 있도록 환경을 만들어 주어야 할 것이다.

결국 소프트웨어 개발자의 최종 목표는 "그동안 인내했던 것들을 찾아가는 여행"이 될 것이다. 비단 개발자만이 그런 것은 아니지만, 후배들을 위한 아낌없는 경험과 기술의 전수를 통해서 또 다른 목표를 발견할 수 있을 것이다. 어쩌면 컨설팅이라는 것을 통해서 그동안 알아왔던 지식을 정리할 기회도 있을 것이다. 흔히 컨설팅이 무용하다는 말도 많이 하지만 이것도 잘만 활용한다면 실제로 일해야 하는 사람들에게는 큰 도움이 된다. 자신들이 하지 못한 말을 그들을 통해서 경영진이나 의사결정권을 가진 사람들에게 할 수도 있기 때문이다. 물론 그렇게 되기 위해서는 제대로 된 컨설팅을 받는 것부터 시작해야 한다. 대부분의 경우 비판받을 것을 두려워한 사람들은 자신의 문제를 공개적으로 이야기하지 않는다. 가리기에 바빠서 뭔가 구체인 것을 내놓기보다는 얼버무리기에 급급하다. 어떤 새로운 시작이든 현재를 정확히 파악하는 것으로 시작해야만 무엇이 문제인지 정확히 발견할 수 있다.

무엇을 인내하면서 지금까지 걸어왔는지를 고민하는 사람은 아마도 이미 내려놓을 것들을 아는 사람들일 것이다. 모든 것을 다 자신이 하기에는 이미 너무 많은 업무가 책상 귀퉁이 여기저기에 쌓여있고, 이제는 더이상 개발자라고 아무도 생각하지 않는 사람이 되어 있을지도 모른다. 아주 쉬운 코딩 밖에는 생각나지 않고, 최신 개발 툴들에 대한 사용법도 모르는 것이 당연시될 정도일 것이다. 대부분의 관리자와 마찬가지로 진급을 통해서 얻는 것은 기술적인 깊이가 아닌 사람들 사이의 관계를 매끄럽게 유지하는 방법만이 있을 뿐이다. 안타까운 일이지만 좋은 관리자가 되는 것이 최소한 회사에 다닐 수 있는 유일한 길이라는 것을 깨달았는지도 모른다. 하지만, 이제는 좋은 관리자라는 것도 내려놓아야 한다. 정말 하고 싶은 일을 하기 위해서는 그런 것들로부터 멀리 서서 객관적인 시각으로 바라보아야 한다.

아직 명확히 정해진 것은 아니지만 10년은 더 다닐 수 있는 시점이니, 준비할 수 있는 시간은 충분히 있을지도 모른다. (직장에 다니는 시간이 30년이라고 가정할 때) 무언가 제대로 하려고 하자면 전문가가 되어야 하고, 최고의 장인이 되고자 한다면 전문가의 수준을 넘어선 "사상가"쯤 되어야 하지 않을까? 목표가 무엇이든 항상 준비하는 삶이 최선이다. "소프트웨어 개발자의 최종적인 목표" 또한 다르지 않다. 지금 주어진 시간이 없다고 자신에게 투자할 시간을 마련하지 않는다면, 10년이든 20년이 흐른 시점이든 상관없이 결국 그런 희망들은 이루어지지 않을 것이다. 잠 못 이룬 밤이든 일순의 휴식이든 항상 미래에 대한 준비는 우선순위가 최상이다. 세상에서 중요한 일을 하는 것처럼 자신의 소중한 미래도 미리 만들어 가기를 빌어본다.

# [ 존중이란? ]

회사에서 오래 살아남은 임원은 "독한" 사람들이라고 한다. 독한 사람들이 득세하는 회사는 조직에 속한 구서원들이 서글프다. 우울한 이야기지만 실제로 이런 사람들이 관리자로 있으면 일단은 수익이 나는 구조를 만들 가능성이 높다. 하지만 그 수익은 크지 않을 것이라는 사실을 잊어선 안 된다. 존중하지 않는 회사에서 창의적으로 일하고 탁월한 성과를 내는 조직문화가 만들어질 수 없기 때문이다. 탁월한 성과는 시켜야지 일하는 회사가 아니라 명확한 원칙을 가지고 일하는 문화를 가진 회사에서 가능한 일이다. 그리고, 그런 “독한” 사람들이 떠나고 나면 그동안 쌓여있던 불함리 함으로 인해서 어려운 순간을 겪어야 할지도 모른다.

남을 존중하지 않는 사람이 존중받을 수 있을까? 상식적으로 그런 일은 발생하지 않는다. 어릴 때 배운 "나에게 하고 싶지 않은 것은 남에게도 하지 말라"라는 말이 있다. 바꿔 말하면 존중받고자 한다면 먼저 존중해야 한다는 의미다. 물론 존중한다는 것은 항상 남을 “섬김”을 의미하지는 않는다. 잘못된 길로 갈 경우에는 냉정한 비판도 필요하다. 하지만 비판의 내면은 그 사람이 더 좋은 것을 하게 하는 것에 있는 것이지, 단순히 그 사람의 잘못을 탓하는 것이 아님을 이미 상대방도 충분히 알고 있어야 한다. 남을 제대로 섬기는 것이 어려운 이유는 자신이 남에게 대접받기 어려운 것과 같다.

회사에 다니는 동안 제일 싫었던 경험에는 어떤 것들이 있을까? 아마도 동료나 상사로부터 듣는 "비꼼"에 가까운 말일 것이다. "네가 하는 일이 다 그렇지", "그런 식으로 하면 짤라 버린다." 등등 이런 말들은 마음을 해친다. 신체적으로 건강할지 몰라도 이미 마음은 그 회사를 떠나있게 된다. 일이 잘 될 리가 없다. 그런 말들을 아무런 스스럼없이 회의 상에서 마치 자신의 신세 한탄처럼 하는 사람들도 있다. 다른 사람을 직접 비판하면 인사부서에서 뭐라고 하니 슬그머니 돌려서 이야기한다. 이건 더 나쁘다. 왜나하면 자신에게는 아무 해(책임)를 입히지 않으면서 남에게 상처를 주기 때문이다.

존중의 시작은 그 사람이 하는 일에 대한 이해로부터 시작된다. 자기가 모른다고 남들도 모르는 것은 아니다. 결과만을 두고 판단하는 것이 아니라 과정도 세밀하게 살펴야 한다. 남을 이해하는 것이 되지 않으, 존중이라는 단어는 회사에 쓸데없이 붙여놓은 슬로건 밖에 되지 않는다. 슬로건은 자신에게 하는 말이 아니라 남에게 "내가 이렇게 할 꺼야"라는 의미다. 하지만 정작 그 회사에 다니는 누구도 그 슬로건을 마음으로 받아들이지 않는다면 그것을 만들 필요가 있을까? "소통"하라고 하지만 소통하지 않는다. 회의라고 하지만 회의가 아니다. 따라서 정말 존중받기를 원한다면 자신과 같이 일하고 있는 한 사람 한 사람을 차분히 지켜보는 것부터 배워야 할 것이다.

# [ 무엇이 행복을 만드나? ]

“자신이 일하는 곳이 천국이라면?”, 그게 바로 완벽한 행복이 아닐까? 사실 대부분 회사원이 그렇듯이 집에서 보내는 시간보다 회사에서 보내는 시간이 더 길다. 일종의 "또 하나의 가족"이라고도 볼 수 있다. 젊은 시절의 대부분을 보내게 될 직장에서 불행한 시간으로 보내고 있다는 것은 슬픈 일이다. 직장을 다니는 것이 즐거움이 아니라면 단순히 집에 있는 가족들을 위한 생계수단 밖에는 되지 못한다. 가족과 함께하는 행복을 희생하면서 회사에 다니고 있다는 많은 사람은 가족을 위한 시간이 소중하다는 것을 모르지는 않지만 실제로 가족과는 동떨어져 있는 것도 사실이다.

천국과 같은 직장으로 인터넷에서 사진들을 찾아볼 수 있는 Google이라는 회사. 개발자라면 그런 곳에서 일하고 싶어 하는 것이 당연하다. 능력이 없어서 들어가지 못하지 좀 더 좋은 환경을 제공받을 수 있는 회사가 있다면 현재의 직장을 바꾸는 것은 어려운 일이 아니다. 회사의 가장 큰 손실은 함께하고 있는 인력이 회사를 떠나가는 것이다. 비용적인 측면도 그렇고, 향후 성장을 위해서도 사람들이 직장을 떠나는 것은 회사로서는 달가운 일이 아니다. 그런데도 많은 회사는 사람을 잡아두지 못한다. 이직의 가장 큰 이유는 회사 내의 시스템도 아니고 급여도 아니다. 급여는 다른 회사를 찾아야 하는 스트레스를 보상하지 못하며, 불편한 시스템도 익숙해지면 적응하는 것이 사람이다. 사람과 사람 사이의 관계, 혹은 기대와 현실의 괴리에서 이직의 원인을 찾아야 한다.

사람과 사람 사이의 관계 대부분은 직장 상사와 동료들과의 관계설정 실패에 기인한다. 자신이 잘못한 것도 있지만, 일단 개인의 입장에서는 남들이 잘못하고 있다고 판단하는 경우가 대부분이다. 중요한 것은 역시 사람들 사이에서 인정을 받지 못하고 있다는 느낌이 가장 클 것이다. 지속적인 질책을 당하고 있는 것도 이유가 될 수 있다. 질책은 사람을 위축되게 만든다. 아무리 잘하던 사람도 지속해서 질책을 받게 되면 자신감을 잃게 된다. 무슨 말을 해야 할 경우에도 입을 닫게 되고 말 그대로 복지부동한 형태로 바뀐다. 책임을 가능하면 지려고 하지 않게 되고, 결국에는 자신에게 피해가 돌아올 것 같은 일은 절대 하지 않게 된다. 창조적인 생각이 그런 상황에서 나온다는 것은 불가능하다. 변화는 내부에서 시작되는 것도 있지만 바깥의 환경에 맞춰서 행동하도록 교육받은 사람은 내부적인 변화를 표현하지 않는다.

행복은 개인적인 기준이지만 회사생활에서 희생해야 할 부분에 대한 정당한 보상을 요구한다. 물질적 혹은 금전적인 보상도 중요하지만 가장 중요한 것은 마음에 대한 보상이다. 질책을 통해서 건져 올릴 수 있는 것도 있지만, 칭찬이나 더 좋은 대안을 제시해서 개인의 창의적인 생각을 끌어낼 수 있도록 하는 것이 옳다. 우리가 동료나 직장 선후배로서의 관계를 유지하는 것은 강제에 의한 것이 아니라 서로 간의 존중에서 나오는 행동이다. 일방적인 지시만 있다면 기계와 다를 것이 없다. 시켜야지 일하는 것보다는 시키지 않아도 깔끔하게 일을 정리하고 더 나은 무언가를 고민하게 만들 수 있다면, 그것이 회사에 다니는 것을 행복이라는 것으로 받아들이지 않을 이유가 없다. 보상은 결국 그런 마음에 기쁨을 주는 것이며, 기쁜 마음으로 회사 일을 할 수 있다면 즐겁지 않을 이유도 없는 것이다.

# [ 관리자의 생산성은 어디에? ]

소프트웨어 개발자에 대한 생산성 이야기는 많이 들어보았을 것이다. 하지만, 정작 소프트웨어 개발자를 관리하는 업무를 수행하는 사람들에 대한 생산성은 어떻게 따지는지 들어본 적이 있나? 관리자의 생산성이 얼마나 높은가를 알기 위해서는 그 관리자의 관리 대상이 되는 소프트웨어 개발자의 생산성을 보면 된다. 높은 자리에 있는 관리자는 경험이라는 것을 믿고 1:1 관리를 하거나, 혹은 해당 분야의 대리인를 내세워서 하부 조직의 관리를 하게 된다. 어쨌든 이미 본인은 개발이라는 업무에서는 동떨어질 수밖에 없기 때문이다.

그런 관리자가 유일하게 즐거워하는 것은 일정데로 일이 잘 진행된다는 보고를 받을 때다. 하지만, 솔직히 즐거운 마음은 숨겨두고 겉으로는 그럴수록 더 잘하라는 채찍을 가한다. 말 그대로 "달리는 말에 더 채찍을 가한다"라는 표현이 잘 어울릴 것이다. 하지만, 정작 채찍을 당하는 것은 "말"이 아니라 "사람"이다. 그것도 대부분은 4년제 대학을 나왔고, 박사나 석사 학위를 가졌을지도 모르고, 외국에서 소위 말하는 아이비 리그(IVY League)대학을 나온 인력도 있다는 사실은 모른다. 소프트웨어 개발자는 채찍을 가한다고 일하는 사람들이 아님을 모르고 있는 것이다. 안타까운 일이지만 그런 것을 좋다고 생각하고 배우겠다고 나서는 하부 조직 관리자도 있다. 회식 자리면 어김없이 그 높으신 분의 옆자리를 차지하고 한 가지라도 놓칠세라 두 귀를 곧게 세우고 듣는다. 차리라 그냥 "난 당신의 영원한 종입니다."그렇게 이야기하는 게 더 좋지 않을까?

물론 모든 중간 관리자가 그렇다는 것은 아니다. 이런 사람도 있고 저런 사람도 있다. 하지만, 그런 중간 관리자가 소프트웨어 개발 실무자와 가장 가깝고, 또한 고위층에 정확한 정보를 줄 수 있는 위치에 있다는 것이 문제다. 팀 내의 분위기 및 과제에 대한 정확한 상황을 위로 전달할 의무와 고위층의 생각을 아래로 잘 정리해서 알려주어야 할 역할을 하기 때문이다. 험악한 고위층을 만나면 대부분은 아마도 욕설과 비슷한 숙제를 매일 매일 실무 개발자들을 동원해서 시킬 것이고, 결과적으로는 과제는 점점 하지 않아도 되는 일로 인해서 더 지연될 것이 뻔하다. 실무 개발자의 의견을 전달한다고 고위층과의 회의 시간에 건의 사항이라도 이야기하면 핀잔을 먹기 바쁠 것이고, 직접 그런 모습을 동료나 하위에 있는 개발자가 본다면 본인의 자존심에도 금이 갈 것이다. 하지만, 누군가 고양이 목에 방울을 달지 않고서는 더 이상의 진전은 없다.

어쨌든 관리자의 생산성은 자신을 제외한 자신의 관리를 받는 모든 사람의 생산성의 합이 되어야 한다. 하지만 정작 자신의 생산성을 윗사람의 평가로만 기대한다면, 자신이 관리하는 사람들이 만드는 생산성은 자신의 것이 아니게 되고, 책임을 져야하는 상황이 오면 남의 탓으로 돌리게 될 것이다. 이런 상태에서는 자신과 팀이 분리될 수밖에 없으며, 실무 개발자들도 관계가 없다는 것으로 받아들인다. 예민한 사람이 아니라도 그런 분위기는 쉽게 눈치챈다. 같은 배를 탔다고 이야기만 할 뿐 자신은 이미 구명정에 한 발을 디디고 선 것이다. 침몰해가는 배를 멀리서 바라보며 선원들에게 계속 남아서 지키라고 말하는 선장과 뭐가 다른가? 말로는 필요한 모든 조치를 다 해주겠다고 하면서, 정작 필요한 시점에는 "능력이 부족하면 시키는 대로 해!"라는 공포 분위기만 조성한다. 심지어 자신이 한 이야기도 제대로 기억하지 못해서 원인과 결과를 거꾸로 생각하는 사람도 있다.

못난 관리자의 생산성은 과제의 성공이나 생산성과는 아무 관계가 없다. 자신의 생산성은 윗사람과의 의사소통 및 평가에만 매달려 있을 뿐이다. 자신이 관리하는 사람들의 생산성을 높이는 것에는 관련이 전혀 없는 것이다. 테스트 케이스를 전부 일일이 개발자가 다 테스트하라고 이야기하는 멍청이는 아마도 소프트웨어 개발을 한 경험이 전혀 없을 것이 분명하다. 수시로 나오는 해야 할 일들을 우선순위도 없이 마구잡이로 던지는 관리자는 자신도 자신이 하는 일에 대해 아무런 확신이 없다는 것을 아랫 사람들에게 과감하게 보여주고 있다는 것을 알까 모를까? 자신에게 달콤한 말을 하는 사람들만 옆에 두기를 원한다면 그렇게 하라. 하지만 조만간 모든 것이 잘 되고 있다는 소리는 들리지만, 아무것도 제대로 끝나는 것이 없다는 것을 경험하게 될 것이다. 인사나 감사에 걸리지 않을 정도로만 아랫 사람을 괴롭히는 상사가 얼마나 유치한지는 일일이 말로 표현할 필요도 없다. 자신은 머리 좋다고 생각할지 모르지만 아쉽게도 그 정도의 잔꾀로 정말 좋은 사람들을 놓치고 있다는 것은 절대 알 수 없을 것이다.

# [ 슈퍼맨(Superman)? ]

영화 속에 등장하는 슈퍼맨은 사회의 악을 뿌리 뽑고 세상에서 어둠을 몰아내는 구세주로 나온다. 영화라는 세계관을 놓고 보면 그런 존재가 우리 곁에서 지켜주고 있으니 안심하고 살아갈 수 있다는 희망을 주기에 충분하다. 하지만, 우리는 현실을 살아가는 소프트웨어 개발자고, 그런 슈퍼맨이 우리 주변에 있어야 할 필요가 있을지는 더 두고 봐야 한다. 소프트웨어 개발에서 소위 말하는 “S(Super)”급 인력이라고 회사의 평가를 받는 인력(혹은, 회사 이름을 건 상을 받는 사람들)에는 크게 두 가지 부류가 있다. 한 부류는 정말 회사의 간판 스타라고 할 정도의 역할을 수행하는 사람이 있고, 또 한 부류는 회사의 성장보다는 자신의 직업 안정성(Job Security)를 더 염려하는 사람이 있다. 후자의 부류에 속하는 인력들은 정말 슈퍼급인지 충분한 검토를 해야 할 사람들이다.

필요한 역량을 잘 갖추고 있는 사람을 자신의 과제에 끌어들이는 것은 기쁜 일이다. 일단 그 사람이라면 뭔가는 해 주겠지 하는 안도감이 생길 것이다. 슈퍼맨과 같이 모든 분야의 지식을 통달하고 있으면서 코딩 생산성이나 기타 등등에서 남들보다 탁월한 성과를 이루어낸다. "지나침은 모자람만 못하다"는 말이 있듯이, 이 역시 그렇게 달가운 현상은 아니다. 나머지 팀원들의 자신감이 떨어지고, 슈퍼맨에게 바라는 것이 많아진다. 슈퍼맨도 사람일 뿐이다. 지칠 때도 있고 하기 싫은 일도 있기 때문이다. 이런 인력들도 업무 부하가 많이 걸리게 되면 결국 “번 아웃(Burnout) 상태로 변하기 쉽다. 혹은 자신의 역량을 과신해서 점차 팀워크를 해치는 상황이 오기도 한다. 협력이 필요한 어려운 일을 자신의 독단으로 처리하게 되고, 그에 대한 비판을 무시하게 되어 협업의 방해 요소로 작용하게 되는 것이다. 만약 정말 슈퍼급의 인력이라는 것을 스스로 인식한다면 팀에 녹아드는 사람이 되어야지 자신만을 위한 팀으로 변경해서는 안 된다.

회사의 이름을 걸어놓은 상을 받는 사람들이 정말 그 상을 받을 만큼 가치 있는 일을 한 것일까? 만약 독단적으로 행동하고, 주위 팀에 대해서는 배타적인 태도를 가지고 있다면 어떻게 할 것인가? 혹은 자신이 관리하는 과제나 사람들에게 절대적인 영향력을 행사해서, 모든 것을 혼자서 결정하고 비판을 들을 가능성을 사전에 막아 소통을 하지 않는다면 어떨까? 물론 그런 인력들이 상대적으로 단기적인 성과를 낼 가능성이 높을 수도 있다. 하지만, 장기적인 회사 차원의 성과를 고려했을 때 그런 슈퍼급이라고 “자신만이 생각”하는 인력은 회사의 독소가 된다. 회사의 성장을 저해하는 주범이 된다는 것이다. 남들의 비판이나 객관적인 평가 없이 자신의 능력이 탁월하다고 믿는 근거는 도대체 무엇일까? 몇 가지 제품을 꾸준히 상품화해 왔다는 자신감이 믿음의 근거일까? 오히려 그렇게 만든 제품들이 더 효율적으로 만들 가능성을 가로막고 있는 것은 아닐까? 좋은 교육을 받는다고 나아지는 것도 아니다. 오히려 그런 인력들은 교육과 인증도 자신을 보호하기 위한 한가지 수단일 뿐이다.

대학 졸업장을 가진다고 모든 사람의 능력이 같은 것은 아니다. 졸업장은 그냥 기초적인 소양을 갖추고 있다는 것을 보장할 뿐이지, 그것을 바탕으로 무언가를 쌓는 것은 자신의 또 다른 노력에 기반을 둔다. 겨우 그렇게 받은 졸업장을 가지고 자신이 마치 무엇인가를 이룬 사람처럼 군다면, 참으로 어리석은 사람이라고 밖에 볼 수 없다. 세상이 넓다고 밖에서만 평가를 받을 생각을 하기보다 자신의 바로 옆에서 같이 일하는 사람들의 평가에 귀를 기울여야 한다. 정말 슈퍼급이 되고자 한다면 기본적인 자세부터 다시 만들어야 한다. 소프트웨어 개발에서 필요한 기본은 “투명성과 비판에 대한 두려움이 없는 나눔”이다. 내가 한 일이 내가 가진 능력 일부를 보인 것이지 나의 인간성을 평가하는 것은 아니다. 또한, 회사에서 내가 만든 모든 것은 회사의 소유이지 개인의 소유가 아니다. 누군가의 도움으로 개선이 될 수 있다면 그것을 마다할 이유는 전혀 없는 것이다. 슈퍼급이 되기 위해서는 능력만이 전부가 아니며, 사람을 대하는 기본 자세인 “존중(Respect)”이 필요한 것이다.

# [ 대화가 필요해 ]

소프트웨어 프로젝트의 실패 요인 중에서 대표적인 것으로 "대화(Communication)의 부족"이 꼽힌다. 여기서 말하는 대화는 외부 고객과의 대화와 더불어 내부 고객과의 대화가 부족하다는 것으로, 요구 사항을 잘 못 파악해 정말 원하는 것이 무엇인지 알지 못한 상태에서 개발이 진행됨을 말하는 것이다. "정말 원하는 것"이 항상 문제다. 일단 외부 고객을 과제의 개발에 끌어들이는 것은 대단히 어려운 일이다. 내부 고객은 그나마 끌어들이기는 쉬우나, 과제 지연의 문제를 일으키기 쉬운 존재들이기도 하다. 과제에 이런 저런 의견을 내고, 우선순위를 마구잡이로 수정할 수 있기에 조심해야 할 사람들이다.

소프트웨어 과제 때문이 아니라 개발의 전체 과정에서도 대화는 중요한 구실을 한다. 대화가 힘들 경우에는 이메일이나, 혹은 문서, 컨퍼런스 콜(Conference Call)등으로 간접적으로 할 수 있지만, 이런 식의 대화는 업무를 진행하기 위한 대화이다. 이보다는 근본적으로 좀 더 깊이 들어가 개발팀 내부의 대화가 협업에는 가장 중요한 요소다. 일상적으로 마주하는 사람들 간의 원활한 의사소통은 공통 언어의 사용을 거부감 없이 받아들일 수 있으며, 과제의 성공이 공동의 노력이라는 것을 확산시키는 중요한 도구가 될 수 있다. 수평적인 팀원들 간의 대화도 중요하지만, 상하 관계를 구성하는 관리자와 관리의 대상이 되는 개발자 간의 대화는 팀이 성과를 내는데 중요한 부분이다.

사실 중요한 것들은 너무나 당연해서 잘 인식되지 못하는 것이 일반적이다. 하지만, 잘 보이지 않는다고해서 없는 것은 아니다. 다만 우리는 너무 당연해서 별다른 생각 없이 뱉어내는 말들이, 받아들이는 입장에서는 불편할 수도 있다. "대화가 필요하다"는 건의 사항을 받았을 때 가장 쉽게 취하는 방법은 "소통의 장"을 만든다는 핑계로 소위 말하는 "간담회(간과 쓸개를 다 꺼내 놓고 이야기하는 미팅?)"라는 것을 급히 만든다. 하지만, 정말 그런 모임에서 제대로 이야기할 수 있을까? 많은 사람이 모인 공개된 자리에서 주고받는 이야기는 형식적일 수밖에 없다. 그리고, 그런 초대를 받아간 자리에서 먹는 음식이 제대로 소화는 될까? 직급이 아무리 높아도 자신보다 더 높은 분과의 식사나 공개적인 대화는 항상 긴장을 유발한다. 그런 식의 공식적인 대화는 즐거운 마음으로 편하게 이야기할 수 있는 자리와는 관계가 멀다.

따라서 그런 분위기 속의 대화보다는 일상의 대화에 더 집중해야 한다. 거의 일상적으로 행해지는 회의나 미팅, 점심 식사 때의 흔한 이야기들이 편하게 오갈 수 있는 분위기가 중요하다. "대화가 필요하다"는 건의사항이 나온다는 이야기는 "제발 쓸데없는 일 좀 시키지 마세요."의 다른 표현이라는 사실을 아는지 모르는지. "교육이 부족하다"는 이야기는 "쉴 수 있도록 조금만 여유를 주세요."라는 뜻이지, 현재도 진행하고 있는 과제 때문에 제대로 못 가는 사외 교육을 보내 달라는 이야기가 아니다. 과제의 지연에 대해서는 몇 단계의 "왜(Why)"라는 질문으로 이런저런 이유를 만들어서라도 보고하게 하면서도, 왜 그런 부분들은 이해하지 못할까? 아마도 개구리 올챙이 시절을 모른다는 흔한 이야기 일지도 모른다. 하지만, 중요한 것은 언제나 사람의 마음을 움직여야 원하는 것을 얻을 수 있다는 점이다. 대화의 시작은 대화 대상의 마음을 이해하는 것부터 시작해야 한다. 설익은 말부터 꺼내려고 하지 말아야 한다.

# [ 비젼(Vision) ]

사람을 움직이는 힘은 미래에 대한 상상을 현실에 옮기려고 얼마나 활동할 수 있는지에 달려있다. 우리는 그것을 비젼(Vision)이라고 부른다. 비전이 없다고 느끼는 것은 현재 상황을 극복하고, 더 나아질 것이라는 희망이 없다는 것을 의미한다. 따라서, 비전이 있는 회사가 되기 위해서는 그 속에서 생활하는 개개인에 대한 발전 가능성의 열린 태도가 필요하며, 일을 통한 성장이 어떤 것인지 알 수 있도록 해줘야 한다. 일상의 업무를 진행하면서 새로운 방법과 체계적인 교육을 받지 않는다면 더 이상 일속에서 배울 수 있는 것은 없으며, 공정하지 못하고 독단적인 판단에 의존하는 부서장을 미래의 자기 모습 속에서 발견하고 싶어하지는 않을 것이다. 이런 생각이 든다면 떠나야 할 순간이 왔다는 신호다.

소프트웨어 개발자의 비젼은 무엇일까? 다양한 사람들이 다양한 비젼을 가지고 있겠지만, 중요한 것은 "나날이 발전하고 있다는 느낌"이다. 자신이 하는 일에 대한 자신감과 개선되고 있다는 생각, 그리고, 새로운 것을 배워서 미래에 사용할 수 있을 것이라는 상상(믿음)이다. 새로운 도구나 방법론을 익히고 새로운 기술을 배우는 것이 그래서 중요하다. 자기 일에 함몰되어 남들이 어떻게 개발하는지를 모르고 있는 것은, "우물 안의 개구리" 같은 생각이라는 것은 다들 잘 알 것이다. 우물이 마를 시기가 왔을 때, 정작 개구리는 우물을 벗어나지 못하게 된다. 더 큰 생각을 해야 할 시점이 왔을 때, 자신의 좁은 경험으로 대처할 수 있는 것이 한정될 수밖에 없다. 소프트웨어 개발자에게 필요한 것은 끝없이 탐구를 할 수 있는 환경을 만들어 주는 것이다. 그리고, 그러한 환경은 강요로 만들어지지 않는다.

"멍석을 깔아주면 놀지 않는다"라는 속담처럼 갑작스럽게 교육을 늘린다고 기뻐하지는 않는다. 오히려 하루에 2시간 정도는 오로지 자신만을 위한 일에 사용할 수 있도록 해주는 것이 더 쉽다. 혹은, 일주일에 하루는 자신이 하고 싶은 일을 하도록 만들어주는 것도 좋다. 물론 이렇게 했을 때 관리자들은 생산성이 떨어질 것으로 생각하겠지만, 소프트웨어를 만드는 것은 공장에서 물건을 조립하는 것과는 차원이 다른 일이다. 지적인 능력을 최대한 끌어올려서 써야 할 노동자들에게 육체적인 노동을 강요하는 것은 어리석은 행동이다. 구글과 같은 회사가 만든 제품 중에 이렇게 창조적으로 개인의 시간을 통해서 개발된 것들을 우리가 많이 사용하고 있다는 사실은 이미 잘 알려져 있다. 문제는 관리자들이 일정 달성에 대한 두려움으로 일을 가능한 빨리 끝내려고 한다는 점이다. 성급하게 일정을 정하고 급하게 일하다 보면 놓치는 것들이 반드시 있게 된다. 사용자들의 요구를 맞추지 못하거나 많은 버그로 인해서 출시 후 재작업으로 인한 신규 제품의 개발 지연이다.

개발자들이 비젼을 만들거나 회사가 만든 비젼을 공유하기 위해서는 그것이 자신에게도 도움이 된다고 확신할 수 있어야 한다. 자신이 단순히 회사에서 돈을 받아가는 존재가 아니라 성장하고 있다는 느낌을 얻을 수 있을 때, 자신과 회사의 비젼이 일치한다고 느낄 수 있을 것이다. 명령이나 호통으로 만들 수 있는 공유된 비젼은 없으며, 강제로 만들라고 해서 만들어지는 것도 아니다. 개발자들은 자신이 회사와 함께 성장하고 있으며, 자신이 하고 있는 일이 회사에서 중요한 일이라고 확인하고 싶어한다. 교육이 필요한 것은 그렇게 자발적으로 새로운 것을 원하는 욕구를 만족시켜주는 것이다. 모두에게 똑 같은 지식을 주입식으로 전달하는 것은 비젼이나 희망을 만들지 못한다. 더군다나 현실과 동떨어진 이상을 가르치는 것도 의미 없다. 그런 상태라면 정말 교육을 받아야 할 사람들은 개발자들이 아니라, 오히려 제대로 소프트웨어 개발자들을 관리하지 못하는 높으신 분들일 가능성이 높다. 소프트웨어는 마음이 급한 관리자가 만들기에는 너무나 복잡한 것이기 때문이다.

# [ 개과천선 ]

"개과천선"이라는 드라마가 있었다. 냉정하고 부와 명예를 위해서 살던 유명 법인 소속의 유능한 변호사가, 교통사고로 인한 기억 상실로 과거의 자신을 되돌아보고 새로운 삶을 살아간다는 내용이다. 소프트웨어 개발자들은 어떤 삶을 살아갈까? 하나의 과제를 열심히 달려서 완료하고 나면 무엇이 잘못되었는지 사후 검토(Postmortem)라는 과정을 거치게 된다. 하지만, 마치 기억 상실증이라도 가진 듯 다음 과제에서도 똑같은 실수(의도된 강제와 일정 지키기에 급급한 코드 마무리)를 반복한다. "사후 검토"에서 배운 것들은 새로운 과제와는 전혀 상관없는 것으로 생각해 버린다. 배워야 할 것은 예측의 정확도를 높이기 위해서 측정한 것들과 잦은 반복적인 실수를 예방할 대책임에도 불구하고 완료된 과제를 거들떠보기도 싫어한다(물론 측정한 것들이 없고 실수 예방을 위한 대책을 시행할 시간도 없이 새로운 과제를 시작하겠지만).

사실 "개과천선"해야 할 사람들은 실무 개발자들이 아니다. 그들은 어떤 일이 주어지든 그 상황에서 최고의 선택을 할 것이라는 신뢰를 충분히 할 수 있는 사람들이다. 오히려, 변화되어야 할 사람들은 그들에게 “명령과 고과”를 동시에 전달할 수 있는 힘을 가진 사람들이다. 배운 것이 없다면 모르겠지만 그 분들도 현업에서의 경험이 최소한 20년은 넘는 사람들일 것이다. (물론 소프트웨어 개발을 그렇게 한 사람들은 아닐 가능성이 높지만) 어쨌든 과제를 많든 적든 열 건 이상은 해봤을 것이 분명하다. 하지만, 그 분들에게 물어보면 고생한 추억에 대한 열정은 술자리의 안주거리로 자주 이야기 하지만, 그 후에 그런 경험을 가지고 변화를 시도했던 경험은 이야기하지 않을 것이 분명하다. 그 분들에게는 바쁜 것이 유일한 행복이었을 것이 분명하다.

바쁘다고 일을 잘할까? 단순히 바쁘다고 생각하고 일하는 것뿐이다. 그것을 통해서 더 좋은 품질을 가지는 제품은 만들지 못한다. 사람은 기계적으로 반복적인 상황에서 지속해서 최대로 빨리 움직이게 프로그래밍 되지 않았다. 특히 육체적이건 정신적이건 그런 상황에 오래동안 노출되면, 피로감으로 인해서 다른 부분에 대해 생각은 할 수 없는 상태가 된다. 예전에 누군가 "너희들은 육체로만 일하니 정신적으로 피곤하지 않아서 좋겠다."라고 하는 소리를 들었는데, 사실 육체가 느끼는 피곤함은 극복하는 방법이 다를 뿐, 다른 일에 신경 쓰지 못하는 것은 마찬가지다. 오히려 육체적으로 피곤한 사람들은 정신적으로도 극도의 피곤함을 느낄 가능성이 높다. 결국 중요한 부분을 놓치고 지나갈 확률이 높아지게 되는 것이다.

피로함을 극복할 수 있는 유일한 길은 휴식 밖에 없다. 휴식할 수 있는 시간을 주지 않고 "세계의 1등이 되고 난 다음에 쉬어!!!"라고 이야기한다면, 아마도 우리는 영원히 쉬지 않고 일해야 할 것이다. 왜냐하면 "세계 1등"이라는 목표 다음에는 "세계 1등 유지"라는 목표가 올 것이 분명하기에 때문이다. 그리고, 그런 일은 끊임없이 반복되는 요구의 일부일 뿐이라는 것이다. 세상에 존재하는 모든 회사는 세계 1등을 향해서 달려간다. 최소한 국내 1등은 하고자 노력할 것이다. 그런 상황이 오면 또 다른 목표를 세워야 하고, 다시 달려가기를 멈추지 않도록 “채찍과 당근”으로 유혹할 것이 분명하다. 개발자들이 탈진(Burnout)하게 되면 새로운 시도나 생각은 멈추게 되며, 조직이 커지면 움직임도 둔화되는 것이 일반적이다. 그런 모든 것을 극복하기 위해서는 우리 자신의 삶에서 진정한 혼자만의 “휴식”을 취할 수 있는 시간(방법)이 간절히 필요하다.

# [ 퇴직 사유서에 적힌 내용에 대한 단상 ]

예전에 다니던 회사를 그만둘 때를 되돌아 보면 아쉬운 점이 많이 남는다. 그중에서 가장 아쉬운 것은 "짤 릴 수도 있는 상황에서 발휘할 수 있는 과감함"이다. 이미 회사를 더 다니고 싶다는 생각을 접었을 때 과감히 불의와 맞설 수 있는 용기가 없었다는 것이다. "좋은 게 좋다"는 식으로 그냥 조용히 정리하고 나오는 것이 남아 있는 친구들에 대한 배려라고 생각했었는데, 지금에 와서 다시 생각해 보면 그것은 남은 사람들에 대한 유기에 가까운 무관심이었던 것으로 생각된다. 남은 사람들을 위해서는 개선된 상황에서 일할 수 있도록 해주는 과감한 행동이 필요했었던 것이다. 조용히 물러나기보다는 큰 소리로 떳떳하게 “옳지 못하다는 것”을 인식시켜 주었어야 했었다.

퇴직원에 퇴직 사유를 적는 공란에 윗사람이 이런 글을 적어 놓았다. "팀 관리 능력 미흡". 팀 관리 능력이 미흡하다는 이유를 퇴직 사유에 적어두는 센스를 가진 관리자가, 어떻게 자신이 관리하는 팀 원들(특히 대부분은 능력도 좋았던)이 그토록 많이 퇴사하도록 만들게 되었을까? 웃기는 소리다. 자신의 얼굴에 침 뱉기 식으로 쓴 글이라는 것을 조금만 생각해보면 쉽게 알 수 있는 것을 왜 그토록 모르는 것일까? 마지막으로 던진 질문은 "미안하지 않냐?"라는 말이었다. 그 질문도 역시 자신에게 던져야 할 질문이다. 이렇게 자신이 관리하는 인력이 퇴직하는 데 대해서 "미안하지 않냐?"라고 자신에게 먼저 물었어야 한다. 사실 그때 머릿속에 든 생각은 "그래, 미안하다. 당신보다는 함께 일했던 사람들에게 미안하다. 당신 같은 사람으로부터 보호하지 못한, 자신이 부끄럽다"라는 것이었다.

사실 앞뒤가 없는 상황에서는 각오할 것이 없다. 이미 잃을 것은 다 잃었고 얻은 것은 다 얻지 못했다. 따라서 더는 잃을 것이 없는 상황이라면 상대가 누구든 간에 상관없이 맞받아칠 수 있다. 더군다나 업계에서 자주 마주 설 일이 없는 사람이라면 한 방 먹여줄 필요도 있다. 직접 그렇게 하기 힘들다면 고위층이나 인사 부서에 대해서 시정을 요청할 수도 있다. 한 번에 안되면 나가는 마지막 날까지 지속해서 요청을 보낼 수도 있다. 가장 좋지 못한 것은 순둥이처럼 가만히 있는 것이다. 참고 인내하는 것은 더 좋은 미래를 열지 못한다. 원인이 되는 당사자에게도 좋지 못하다. 왜냐하면 그 사람은 자신이 옳다고 믿고 또 다른 희생양을 만들어 나갈 것이기 때문이다. 회사는 그곳에서 일하는 모두를 위한 곳이지 몇몇 사람들을 위한 모임이 아니다. 퇴직을 각오한다면 비난받아 마땅한 상대를 어떻게 요리할 것인지를 주도면밀하게 준비를 해야 한다.

모든 준비는 회사의 업무 규정에 따라서 해야 한다. 회사에서 임직원들에게 하지 말라고 하는 것들을 얼마나 위반했는지에 대한 사례를 꾸준히 구체적으로 모아서 정리해 두어야 한다. 가령 수요일은 일찍 퇴근해서 집에서 가족과 함께하라는 것을 회사에서 규정하고 있다면, 그것을 위반하고 야근을 강제로 한 경우(물론 공식적으로 그렇게 명령하지는 않겠지만, 야근 기록을 가지고 비난한 경우를 찾아서) 등을 기록하는 것이다. 심한 모욕감을 주는 이야기를 들었을 때도 구체적으로 기록해야 할 것이다(언제 어디서 누구와 함께 있을 때 어떤 말을 했는지). 그리고, 각종 이메일이나 문서 등도 자신에게 유리한 것은 챙겨두길 바란다. 이 모든 것을 한 번에 필요한 부서에 던져주고 유유히 걸어 나오면 된다. 복수심 같은 것은 잊어라. 그건 복수가 아니라 남아있는 동료들에 대한 최소한의 예의다. 그들이 조금이라도 더 좋은 환경에서 일할 수 있도록 만들어주는 인간으로서 가져야 할 최소한의 양심적인 행동인 것이다. 뒤돌아볼 필요는 없다. "더 나은 것은 항상 미래에 있기에 우리의 출발도 그것을 위함이다."

# [ 잘되는 회사의 특징 ]

문제는 쉽게 보이지만 해결방법을 찾는 것은 어려운 경우가 많다. 하지만 문제가 문제로 인식되지 않거나 은연중에 익숙한 상황에서는 문제를 찾아내는 것 조차도 어렵다. 익숙함이란 편안함과 더불어 이유를 묻지 않을 수 있다는 것이다. 잘 되는 회사들의 특징들을 이해할수 있다면, “왜” 잘 안되는 회사가 있는지도 쉽게 이해할 수 있을 것이다. 여기서는 그런 특징들을 나열해서 우리가 어떤 방향으로 가야하는지 잠시 알아보고자 한다.

**01. 결과보다는 과정을 중시한다.**

**02. 규율을 규율로 보지 않고 당연한 것으로 받아들인다.**

**03. 항상 학습(Study)하는 태도를 가진다.**

**04. 자체적인 토론을 통해서 해결책을 마련한다.**

**05. 대화를 중시한다.**

**06. 같이 밥을 먹는다.**

**07. 문서 작성에 능하다.**

**08. 결과물에 대한 검토를 지속해서 한다.**

**09. 누구의 잘못인가를 탓하기보다는 왜 잘못되었는지 이유를 먼저 찾는다.**

**10. 실행에 집중한다.**

물론 여기에 적은 10가지는 개인적인 생각을 정리한 것뿐이다. 번호는 임의로 붙인 것이기에 우선순위와는 관련이 없다. 중요한 것은 이런 것들이 어떤 외적인 압력에 의해서 실행되는 것이 아니라, 내부의 욕구가 분출되는 과정에서 생긴다는 것이다. 팀원이나 회사의 구성원들이 자체적으로 만들어가고 지속해서 개선하는 문화로 정착된다는 점이다. 불합리한 점들은 점차 정리되어 가고, 타당한 근거와 이유가 바탕이된 조직으로 변화되어 간다. 수직적인 조직 구조는 유지하지만 작은 조직의 내부는 수평적인 관계가 유지될 수 있다. 더 효율적인 조직을 만들어 노는 사람을 줄이는 것이 목표가 아니라, 스스로 해야 할 일을 찾아서 집중하는 형태로 바뀔 것이다.

**결과보다는 과정을 중시한다.**

대부분의 소프트웨어 개발 문제는 과정을 무시하는 데서 발생한다. 결과만을 빨리 보고 싶어 하는 성급함으로 일을 그르치게 되는 것이다. "우물에서 숭늉 찾는다"는 옛말이 있듯이, 소프트웨어 개발에서도 정해진 것들을 해야만 좋은 결과물을 얻을 수 있다. 여기서 말하는 정해진 것들이란 일명 프로세스라고도 이야기하고 있지만, 그런 정형화된 프로세스를 나름의 잣대를 통해서 적절히 운영해야 좋은 결과를 가져올 수 있다. 과정을 잘 지키면 결과는 그냥 따라오는 것이기 때문이다. 중요한 점은 하나의 결과를 만들기 위해서는 중간에 무수한 결과들(혹은, 결과물들)을 검증하는데도 신경 써야 한다는 것이다. 마지막 단계에서의 검증은 돌이킬 수 없는 지연을 낳을 확률이 높기 때문이다.

과정을 중시하는 것은 해야 할 일을 제 때 해야 하는 것과 관련이 있다. 제대로 된 요구사항을 파악하려는 노력을 해야 하고, 그것을 통해서 문제에 대한 해결책을 만드는 설계를 진행해야 하며, 구현을 통해서 실행할 수 있는 결과물을 만들어야 한다. 각각의 과정들 사이에는 반드시 검증이나 리뷰 형태로 공유 및 테스트를 실행해야 한다. 구현하는 과정에서도 점증적으로 조금씩 실행되는 코드를 만들어나가고, 증가된 부분에 대한 자동화 테스트를 구축하도록 노력해야 한다. 통합과 테스트는 반복적인 과정으로 빠르게 처리될 수 있도록 인프라 및 팀원들의 활동이 필요하게 되며, 항상 문제에 대해서는 투명하게 관리되어 즉각적인 수정이 이루어질 수 있도록 만들어야 한다. 고객의 피드백이 있다면 좋겠지만 그런 것을 기대할 수 있는 상황이 아니라면, 최소한 팀 내부에서라도 자체 검토를 해서 더 좋은 결과물을 만들도록 해야 할 것이다.

실행되는 코드에 집중하고 이를 작은 단위로 꾸준히 검증하는 것은 특히 중요하다. 이런 활동을 통해서 코드에 숨겨진 버그들을 60%이상 제거할 수 있기 때문이다. 각각의 리뷰와 검사 활동을 통해서 더 많은 버그를 사전에 제거할 수도 있다. 물론 이것은 생각보다 시간을 많이 소비해서 조급한 과제 리더는 항상 걱정이 앞서게 되는 것도 사실이다. 이때 필요한 것은 사전에 제거되는 버그 수를 지속해서 관리하는 것이다. 이를 통해서 나중에 나올 수 있는 버그를 얼마나 줄였는지 위안으로 삼을 수 있으며, 시스템 테스트 단계에서 발생하는 디버깅 활동의 시간을 줄일 수 있다는 점을 눈으로 확인할 수 있기 때문이다. 개발은 "기다림의 미학(인내)"이 필요하다. 팀원들을 다그쳐서 몰아대는 것으로는 제대로 된 해결책을 찾지 못한다. 성급함은 땜빵을 낳고, 땜빵은 더 많은 문제를 만들어 낼 것이다. 풍선의 한쪽을 누르면 다른 쪽이 부풀어 오르듯이, 지금 단축한 일정들은 나중에 그것보다 더 큰 일정 지연의 원인이 될 것이다. 너무 눌러버리면 풍선은 터진다는 점도 잊어선 안될 것이다.

결과도 물론 중요하다. 하지만, 결과물로 만들어진 제품의 품질이 더 중요하다. 대충 결과물을 만드는 조직과 품질이 우수한 훌륭한 결과물을 만드는 조직은 차원이 다르다. 그 다른 차원의 차이을 메워 주는 것이 바로 과정이며, 과정을 얼마나 충실히 소화해 낼 수 있는가는 팀의 역량에 의존한다. 대충 만드는 조직은 결과물의 품질이 들쑥날쑥할 것이 분명하다. 그것이 성능이든 변경 용이성이든 간에 상관없이 항상 최적화라는 원죄를 지니고 다닐 것이다. 하지만, 훌륭한 조직이 만든 제품의 품질은 일정한 수준의 성능을 지속해서 유지하면서 다른 품질 속성(Attribute)도 무시하지 않을 것이다. 또한, 그런 과정에 대한 준수가 품질에 미치는 영향을 잘 알기에 스스로가 몸으로 체험한 기억을 다음 과제에서도 활용할 가능성이 높다. 소프트웨어의 개발문화는 이렇게 과제에서 과제로 이어지면서 자연스럽게 모든 팀원이 동화되어가는 과정에서 만들어진다. 특정 부분만 강조한다고 좋은 개발문화가 만들어지지는 않는다.

**규율을 규율로 보지 않고 당연한 것으로 받아들인다.**

규율(Discipline)은 조직에서 생활하고 있는 사람이라면 기본적으로 가져야만 한다. 하지만, 여기서 말하는 규율은 회사에서 이야기하는 그것과는 다르다. 즉, "어떤 일을 하기 위해서 효율적으로 정해진 방침" 정도로 생각해 볼 수 있다. 대체로 이런 것들은 다양한 시스템으로 구현되는 경우가 많지만 사실 시스템보다는 사람이 더 중요하다. 실제로 해당 시스템을 이용하는 사람이 올바른 정신(Mind)를 가져야지 제대로 시스템이 운영될 수 있기 때문이다. 규율의 영어 정의는 아래와 같다. (아래 내용은 Dictionary.com에서 가져온 것이다.)

1. training to act in accordance with rules; drill: military discipline.
2. activity, exercise, or a regimen that develops or improves a skill; training: A daily stint at the typewriter is excellent discipline for a writer.
3. punishment inflicted by way of correction and training.
4. the rigor or training effect of experience, adversity, etc.:
5. the harsh discipline of poverty.
6. behavior in accord with rules of conduct; behavior and order maintained by training and control:
7. good discipline in an army.
8. a set or system of rules and regulations.
9. Ecclesiastical. the system of government regulating the practice of a church as distinguished from its doctrine.

규율이란 어떤 Rule에 따르는 훈련, 실력이나 개발을 하기 위한 요법, 훈육, 행동 방침에 따르는 행위, 규칙이나 규정을 정의한 시스템 등을 이야기한다. 이런 것들이 몸에 익숙하게 받아들여져 있는 상태가 규율이 규율처럼 보이지 않는 상태일 것이며, 비능률적이거나 비논리적인 것들이 없는 상태일 것이다. 그냥 하라는 대로 하는 것이 아니라, 왜 그렇게 해야 하는지를 이미 잘 알고 있다는 것이다. 또한 이런 것들이 체계화되어 시스템적으로 관리가 되고 있으며, 오히려 규율을 준수하지 않는 것이 이상한 행동으로 보이게 되는 상태이다.

소프트웨어 개발에서 필요한 규율이란 이미 많은 부분에서 이야기되어왔다. 즉, 품질 좋은 소프트웨어를 만들기 위해서 해야 할 일들을 다양한 프로세스의 단계별로 정의하고 있으며, 이를 자신의 상황에 맞게 최적화시켜서 사용하는 것이다. 이때 중요한 것은 최적화를 편의를 위한 것으로 자의적으로 해석하는 것이 아니라, 결과물의 품질을 높이는 데 목표를 두고 있다는 사실이다. 따라서 모든 활동의 목적인 결국 최종 결과물의 품질로 귀결된다. 해당 활동이 품질에 영향을 준다면 어떤 방법이든 사용해도 무방하다. (단, 지나친 것은 없느니만 못하다.) 그리고, 사용하고자 하는 도구나 방법론이 한 가지로 모든 것을 해결할 수 있다고 이야기한다면 주의해야 한다. 품질을 높이는 일에는 지름길이 없기 때문이다.

**항상 학습하는 태도를 가진다.**

“세상은 생각보다 넓고 아는 것만큼 보인다”는 말이 있다. 사실 우리가 생활하는데 필요한 지식 대부분은 유치원 때 배운다고 하지만, 그것은 말 그대로 기본적인 지식일 뿐이다. 그 때 배울 수 있는 것은 반드시 지켜야 할 것들이다. 사실 어떤 일을 하는데 필요한 지식도 가장 기본적인 원칙들에 가깝다. 기본을 아는 것이 성장에 필요한 토대를 튼튼히 하기 때문이다. 소프트웨어 분야는 항상 새로운 것들로 넘쳐난다. 너무나도 배울 것들이 많아지고 있는 것이 현실이다. 조금만 게을리해도 남들과 차이가 나는 듯이 보이기도 한다. 하지만, 조금만 고민해 보면 정말 그런 모든 것들이 필요할까? 물론 특정 분야의 일을 하기 위해서는 도메인(Domain) 지식이 필요하다. 그런 도메인 관련 지식은 직접 그 일을 해야 쌓이는 것이 가능하다. 단순히 책을 읽거나 수업을 듣는다고헤서 해결되지 않는다. 그리고, 항상 변화가 빈번하기도 하다. 그런 도메인에 특수한 지식을 제거하고 나면, 사실 소프트웨어 개발은 기초적인 개발 방법론에 대한 프레임만 남는다. 이것들이 바로 기본기를 이루는 토대가 되며, 특정 분야의 솔루션(Solution)을 개발하는 것은 그런 틀(Frame)에 살을 붙이는 과정일 뿐이다.

중요한 것은 소프트웨어 개발의 근간을 이루는 기본기에 충실해야 한다는 점이다. 그리고, 그런 기본에 관련된 지식들도 광범위하게 있으며, 이미 많은 부분은 정형적인 틀을 갖추고 있다. 시행착오를 통해서 어떻게 소프트웨어를 만들어야 고품질을 달성할 수 있는지를 이미 많은 사람이 정리해 놓았다. 소프트웨어 개발자에게 필요한 것은 그렇게 정리된 것들을 자신의 것으로 만드는 것이다. 경험이 없는 사람은 “어떻게(How-To)”에 집중한다. 경험이 많은 사람들은 “왜(Why)에 더 많이 집중한다. 공통점은 자신이 무엇을 모르는지를 알고 그 부분에 대해서 꾸준히 보완하고 있다는 것이다. 소프트웨어 개발자는 같은 개발자로부터 배운다. 옛날 도제제도와 같이 장인(대체로 개발 경력이 오래된 개발자)에게서 직접적인 전수를 받게 된다. 하지만, 아쉽게도 제대로 된 장인을 만나기는 하늘의 별 따기와 같다는 점과 만났다고 하더라도 배울 시간이 없을 수도 있다는 것이다. 왜냐하면, 그런 마스터(Master)급의 장인들은 대부분 코딩에서 손을 놓고, 이런 저런 회의에 참석한다고 바쁜 시간을 보내고 있을 것이기 때문이다.

마지막으로 남은 방법은 남들이 좋다고 이야기하는 책들에 대한 탐독과 실무에 적용을 작게나마 직접 손으로 해보는 것이다. 전체 팀에 적용하기 힘들다면(권한도 없겠지만), 자신이 하는 일만이라도 적용해 보자. 아마도 남들보다 조금이라도 일찍 일을 마칠 수 있다는 것에 한 표를 던질 수 있을 것이다. 남들이 보기에는 맨날 노는 듯이 보일지 모르지만, 어쨌든 주어진 일에 대한 결과는 항상 깔끔하게 나올 것이다. 해야 할 일을 적시에 하는 것들이 남들이 좋다고 이야기하는 책들에 다 나와 있으며, 그렇게 자기 일을 하는 사람이라면 점점 더 생산성이 좋아질 것이 분명하기 때문이다. 예를 들어, 단위 테스트나 방어적인 코딩, 낮은 결합도와 높은 응집도에 맞춘 모듈화, SOLID원칙 등등을 자신의 코딩에 적용하면, 당연히 변화에 대한 대응이 빨라질 것이다. 이와 같은 것들은 누적 효과가 있어서 6개월에서 1년 정도의 시간이 지나면 다른 사람들과의 차이가 눈에 분명하게 보일 것이다. 가장 주의해야 할 것은 논리적이지 못한 윗사람의 일방적인 지시다. 왜 그렇게 하는지도 모른 채 끌려가고 있는 신입사원들은 결코 좋은 경험을 쌓지 못한다. 회사 생활의 첫 시작이 중요한 것도 바로 이것 때문이다. 좋은 윗사람을 만나면 인생이 바뀔 수 있다. 반대로 그렇지 못한 윗사람을 만나게 된다면 스스로 다른 길을 찾아야 할 지도 모른다.

그렇다고 윗사람만 탓해선 안 된다. 어쨌든 자신의 인생이다. 자신의 소프트웨어 개발자 인생을 남에게 의탁하고 싶다면, 나중에 신규 팀원들이 생겼을 때 아랫사람에게 똑 같이 하고 있을 것이다. 왜냐하면, 윗사람에게 잘못 배운 것 외에는 아는 것이 없기 때문이다. 아마 이 때쯤 되면 도리어 "Why"라고 물어오는 아랫사람에게 화를 내게 될지도 모른다. “그냥 시키는데로 해”라는 것이 유일한 대답이 될 수도 있을 것이다. 자신이 잘 모르는 것, 경험해 보지 못한 부분들에 대해서 자신감이 없기 때문이다. "화"는 스스로가 이해가 되지 않는 불쾌감을 표현할 때 나온다. 자신이 모르고 있다면 솔직히 인정하고 같이 해결책을 찾는 것이 올바른 태도다. 단순히 무조건 찍어누르기만 한다면, 나중에는 개선에 대한 아무런 능동적인 대답을 받지 못할 것이다. 모른다는 것을 인정하는 것이 어렵다는 사실은 이미 다 알고 있을 것이다. 하지만, "그럼에도 불구하고" 모른다고 인정하는 것이 진정한 용기이며 개선을 위한 시작이 될 수 있다. 누구나 처음에는 기어 다니는 연습부터 시작하고 결국에는 뛰어다닐 수 있게 되기 때문이다.

**자체적인 토론을 통해서 해결책을 마련한다.**

대체로 회사의 분위기가 힘들어질수록 "소통"해야 한다고들 이야기한다. 하지만, 진정한 소통은 "계급장"을 떼고 할 때나 가능하다. 대부분의 경우 자신의 지위와 입장을 견지하면서 남의 의견보다는 자신의 주장에만 목소리를 높인다. 다른 사람의 의견에서는 소위 말하는 "꼬투리"를 잡으려고 애쓰고, 이미 결정된 후에도 지속해서 "뒷다리"를 잡는다. 자신이 유능하다는 것을 남들로부터 인정받고 싶기 때문이다. 하지만, 정말 유능한 사람은 남들의 의견을 충분히 잘 듣고, 자신의 의견을 객관적으로 전달하는 사람이 아닐까? 자신이 처한 조직의 상황이나 지위와는 무관하게(물론 완전히 무관할 수는 없겠지만) 회사를 위해서 이야기해야 할 것이다. 그리고, 이미 결정된 것에 대해서는 일단 죽이 되든 밥이 되든 한번 해보는 것이다. 최적의 결정이 아니더라도 의사 결정을 위한 충분한 이야기가 오가면 적절한 결정이라고 볼 수도 있는 것이다.

이런 경우가 있다. 지위를 남용한 경우로 일단 회의가 시작되면 자신의 의견을 말한다. 그리고, 다른 직위가 낮은 사람이 이야기하면 자신을 설득시키라고 요구한다. 만약 자신을 설득할 수 있는 논리가 부족하면, 자신이 결정한 사항에 대해서 의문을 가지지 말고 무조건 받아들이라고 한다. 이건 "소통"이 아니라 그냥 "말 잔치"일뿐이다. 그런 말 잔치의 상 위에 마련된 음식은 온통 다른 사람들이 소화하기 힘든 것들뿐이다. 아마도 이런 회의를 하고 나면 회의록에는 "모두 다 동의했음"과 같이 적혀질 것이다. 그리고, 그 다음 날 부터는 "왜 시키는 데로 안 하냐"고 비난만 할 것이다. 오히려 이런 회의라면 하지 않느니만 못하다. 한 가지 더 웃기는 것은 이런 회의를 업무 시간에 하지 않고, 저녁 식사 후 7시 정도에 한다는 것이다. 왜냐하면, 자신이 맡은 과제의 팀원들이 조금이라도 더 일을 많이 하도록 만들기 위해서다. 하지만, 팀원들은 그렇게 생각하는 윗사람들보다 더 똑똑하며, 그런 자그마한 것들을 이용하는 사람들에 대해서 실망하기 마련이다.

자체적인 토론이란 말 할 수 있는 자유를 의미하며, 자유로운 생각을 공유할 기회를 제공하는 것이다. 이런 분위기가 회의에서 나오는 조직이라면 아마도 일상적인 대화나 일하는 분위기 자체는 이미 달라져 있을 것이다. 자기 생각을 자유롭게 표현할 수 있는 기본 환경이 이미 갖춰진 조직일 것이다. 따라서 자체적인 토론의 결과를 통해서 주요한 문제에 관해서 의견을 모으고, 그렇게 만들어진 결정들은 자연스럽게 모든 팀원에게 "책임"으로 지워질 것이다. 물론 이때의 책임은 "기꺼이 받아들일 수 있는" 것들이며, 그 책임을 완수하기 위해서 소프트웨어 개발자들은 말하지 않더라도 야근을 밥 먹듯이 할 것이다. 적어도 제대로 된 관리자라면 먼저 아랫사람의 이야기에 대해서 경청(귀 기울여 듣기)부터 해야 한다. "자신을 이해시키라는 말"은 최악이다. 자신은 능력이 안되니 자기를 이해시키라고 한다면, 경험(일한 기간)이 부족하다는 것만으로 상대방의 생각을 깔아뭉개는 것밖에 되지 않을 것이다.

**대화를 중시한다.**

대부분의 경우 팀장이나 중간급 관리자들은 팀원들과 대화를 충분히 하고 있다고 이야기한다. 자신들은 이미 그런 부분에서는 아무런 문제가 없다는 듯이 행동한다. 하지만, 정말 중요한 대화는 하지 않고 대부분은 듣는 것 없는 전달만 하고 있을 뿐이다. 특히 무언가를 결정하려고 할 때는 일방적인 의견을 내놓는 일이 흔하다. 대화는 남의 말도 잘 들어야 하는 것이 기본이다. 대화에서는 자신의 의견을 말하는 것이 중요하듯 남의 의견도 잘 들어야 한다. 목표는 한가지지만 달성할 수 있는 방법은 다양할 수 있기 때문이다. 문제를 다양한 시각에서 봐야지만 전체적인 본질을 파악할 수 있지만, 자신의 시선으로만 대화를 이끌어가면 진정한 해결방법을 찾을 수 없다.

“소통”해야 한다는 현수막을 만들어서 출퇴근 시에 구성원이 보라고 걸어놓은 회사는, 반대로 이야기하면 제대로 된 대화를 전혀 하고 있지 않다는 역설적인 표현이다. 그런 것들을 이미 잘 하는 회사라면 당연히 그런 표어는 없을 것이다. 하지만, 표어를 걸어놓으면 오히려 더 반감이 가는 것이 사람의 마음이다. 그런 상투적인 표현보다는 오히려 더 투명한 관리를 할 방법을 찾는 것이 옳다고 생각한다. 대화는 억지로 하라고 해서 하는 것이 아니다. 대화의 시작은 관심이며, 서로 상대에 관해서 관심이 있을 때 소통이 시작된다. 대부분의 경우 윗사람이 아랫사람에게 관심을 가져야 좋은 시작을 할 수 있지, 아랫사람이 윗사람에게 먼저 다가가기는 쉽지 않다.

연봉이나 고과와 같은 것을 이야기하기 위한 대화는 좋은 시작을 하기에 부적절하다. 그런 대화는 이미 결과가 나온 상태에서 하게 되는 것이 대부분이며, 대화 상대를 설득하려는 의도를 가지기 때문이다. 그렇다고 그런 면담을 하지 말라는 것은 아니다. 정확히 이야기하면 고과나 연봉에 관련된 면담은 반드시 해야 한다. 그리고, 그 면담에서는 객관적인 자료를 가지고 임해야 한다. 민감한 대화일수록 충분히 객관적인 자료를 가지고 상대를 이해시켜야 하기 때문이다. 단순히 자신의 개인적인 취향으로 상대의 연봉이나 고과를 마음대로 결정해서는 안 된다. 중요한 잣대가 되는 것이 바로 연초나 연말에 만드는 개인별 성과지표(KPI: Key Performance Index)일 것이다. 물론 이것도 중간에 변경이 될 수 있지만, 얼마나 자신이 계획한 일을 잘 했는지를 객관적으로 증명할 수 있는 근거를 제공해 줄 수 있어야 한다. 이것이 잘 안될 경우에 사람들은 자신이 하는 일에 대한 의미를 잃게 되며 윗사람이나 회사를 신뢰하지 않게 된다. (그렇다고 KPI가 고과를 위한 정확한 근거가 된다는 뜻은 결코 아니다. 적어도 객관적인 자료를 통해서 이야기 해야 한다는 점을 표현하기 위한 예로 썼을 뿐이다.)

작은 대화가 중요하다. 작은 대화란 주로 식사 시간이나 이른 아침의 짧은 미팅, 혹은 지나가면서 던지는 순간의 인사말 정도가 좋다. 작지만 중요한 대화는 팀원 한 사람 한 사람에 대한 지극히 개인적인 것들로 채워지는 것이 좋을 것이다. 물론 그렇다고 해서 완전히 사적인 것은 대화의 주제가 되어서는 안 된다. 예를 들어 팀원의 집에 있는 문제(누군가 아프다던가 애들에 대한 것 등등)들에 대해서 기억하고, 어떻게 되고 있는지를 묻는다든가 하는 것은 좋다고 생각한다. 만약 와이프나 애들이 아파서 병원에 갔다고 한다면, 경과가 어떻게 되었는지 물어보는 것도 좋은 시도가 될 것이다. 팀원들은 자신이 개인적으로도 윗사람에게 중요한 존재라는 인식을 하게 되며, 이는 자연스럽게 친밀한 상태로 이끌어 갈 수 있도록 만들어 주기 때문이다. 그리고, 생활에서의 작은 발견들도 나눌 수 있으면 좋다. 그것을 통해서 점차 서로 동화되어간다는 느낌을 갖게 되기 때문이다. 중요한 것은 팀원과 점차 서로를 닮아가면서 더 높은 목적을 추구할 수 있는 상태에 도달하는 것이다.

**같이 밥을 먹는다.**

사실 팀으로 매일 매일 같이할 수 있는 시간은 밥을 먹는 시간밖에 없다. 특히 점심을 먹는 시간은 하루 동안에 할 수 있는 아주 중요한 최고의 단체 행동이다. 저녁을 같이할 수 있는 사람들도 있긴 하지만, 대부분은 각자의 일정에 따라 다 모이기가 힘들다. 대체로 한 개의 팀은 5에서 9명으로 이루어지며, 이 정도는 충분히 수용할 수 있는 자리가 식당에 마련되어 있다. 같이 밥을 먹는다는 것은 팀이 단합됨을 보여주는 아주 작은 예이지만, 꾸준히 같이 밥을 먹다 보면 이런 저런 다양한 일상적인 이야기들이 나오게 되고 서로의 생각을 같이 공유하게 된다. 밥을 먹는다는 행위가 중요한 것이 아니라 “같이 한다”는 것이 중요하다.

어떤 경우에도 자주 보는 사람과는 친해지는 게 당연하다. 물론 같이한다는 사실 때문에 힘들어서 사직서를 쓰는 사람도 있지만(대부분 회사에서의 문제는 사람과 사람 사이의 문제다.), 많은 시간을 같이하다 보면 상대를 이해할 수 있는 시간도 늘어나게 된다. 이런 저런 일로 부딪히는 상황이 늘어나더라도 밥 먹는 자리에서는 그런 이야기는 하지 않는다. 왜냐하면, 적어도 같은 팀으로 다른 사람들과 함께 하기 때문이다. 그리고, 부딪히는 일도 자주 하면 서로 간의 견해 차이를 잘 이해할 수 있게 된다. 오히려 팀 회의 시간에만 모여서 이야기하고 각자 따로 점심을 먹는다면 상대에 대해서 좋은 인상을 받지 못한다. 사람의 가장 기본적인 욕구를 같이 충족하는 사람들에게서는 팀워크라는 것이 자연스럽게 생겨나는 것이다.

팀에 적응하지 못하는 사람을 찾을 때 유심히 관찰할 부분은 함께 식사하는 사람이 누구인지를 보는 것도 한 가지 방법이다. 특히 문제가 있는 팀원의 경우에는 특정 부류의 사람들과만 이야기하고 식사한다. 혹은 밖에 나가서도 항상 특정한 사람들과 어울린다. 팀원 중에 누군가 그런 사람이 있다면 그 사람이 팀에 적응할 수 있도록 도와줄 대상으로 봐야 할 것이다. 회사에 좋은 식당을 갖추고 있지만 밖에서 식사하는 경우가 많은 사람의 경우에는 회사에 대한 신뢰가 없어진 사람일 가능성이 높다. 그런 사람은 다른 회사로 이직을 준비하거나, 마지못해 회사를 그냥 대충대충 다니고 있는 사람일 수도 있다. 밥 먹는 시간조차도 회사에 있고 싶어하지 않기 때문이다. 그리고, 회사에서 제공되는 것들이 자신에게는 더이상 필요하지 않기 때문이다.

과제를 제대로 완수하기 위해서는 이런저런 사람들의 노력이 다 합해져야 한다. 사람의 능력은 100배도 차이가 날 수 있으며, 항상 능력이 탁월한 팀원만을 아래에 둘 수 있는 것도 아니다. 따라서, 개개인의 능력에 적합하게 일을 주어야 하고 그에 맞는 합당한 보상도 해야 한다. 그들 모두는 자신의 개인적인 역량에 따른 차별화된 보상을 받을 수 있지만, 팀으로 일할 때의 역할에 충실하도록 만들어야 한다. 따라서, 팀워크란 이런 차이에 대한 인정과 그러한 차이를 합해서 목표를 이루도록 하는 조직으로 만들어주는 것이다. 팀워크의 바탕은 자주 같이함을 통해서 작지만 소중한 시간을 공유하는 것이며, 같이 밥을 먹는 것은 작지만 꾸준한 공동 활동을 늘려주는 습관이 될 것이다.

**문서작성에 능하다.**

회사에서 빠르게 진급하는데 필요한 능력 가운데 하나가 문서 작성이다. 물론 문서만 작성을 잘한다고 진급이 되는 것은 아니다. 하지만, 문서를 작성한다는 것은 논리적인 생각과 읽는 사람을 위한 전달 능력이 잘 갖추어져 있다는 것으로 상당한 고급 역량을 의미하는 것이다. 글을 잘 쓰는 사람은 생각보다 드물다. (특히 공학을 전공한 학생들의 경우에는) 대부분의 경우 간략하게 몇 줄 정도는 작성할 수 있을지 모르지만, 장문의 글을 쓰라고 하면 더는 쓸 것이 없다는 말만 한다. 정말 그럴까? 사실 대부분의 글쓰기는 일이 완료된 이후에 시작되기 때문에 쓸 것이 결론밖에 없는 것이다. 글쓰기는 일을 진행하는 동안에 해야 한다. 내용은 "왜(Why)"라는 질문에 대한 대답의 형식을 취해야 하며, 결정에 대한 근거를 남기는 것이 글쓰기다.

"왜"라는 질문은 선택에 대한 이유(근거)이다. 소프트웨어를 만드는 과정에서는 항상 선택이라는 문제가 생기며, 적절한 선택은 근거가 확실하게 있어야 한다. 설계도 마찬가지다. 어떤 설계를 했을 때 그 근거가 무엇인지를 명확히 남겨야 한다. 근거는 실험적이거나 경험적으로 알고 있는 것들이 될 수 있으며, 실험적인 경우에는 측정 방법과 결과가 명확해야 한다. 경험적인 경우에는 그 선택을 하게 된 동기를 적어야 하며, 다른 선택에는 어떤 것들이 있을 수 있는지 문서로 남겨야 한다. 물론 잘못된 선택도 문서로 남기는 것이 좋다. 다른 사람들도 똑같은 실수를 할 가능성이 있기에 잘하지 못한 선택에 대한 결과도 전달되어야 한다. 중요한 것은 문서가 사람보다 오래 남으며, 지속해서 관리되는 문서들은 전부 의미가 있는 것들이라는 점이다. 관리되지 않는 문서(업데이트가 안되는 문서)는 작성할 필요도 없다.

대부분의 경우 설계서를 작성하라는 요청을 받으면 몇 개의 그림만을 달랑 남겨둔다. 그리고, 그 그림조차도 너무 상세하거나 혹은 아무런 내용을 전달해주지 못할 정도로 대충 그린 것들이 많다. 이것은 문서라기보다는 그냥 메모 정도일 뿐이다. 그것도 개인에게만 도움이 되는 수준일 뿐이다. 그런 것들은 관리되지 않으며 시간이 조금 흐르면 아무도 기억하지 않는다. 누군가의 요청으로 성급하게 만들었기 때문에 내용도 부족할 것이 분명하다. 문서는 그렇게 작성해서는 안 된다. 누군가의 요청에 대한 응답이기에 명확해야 하며, 문서 자체의 형식이나 내용에서도 일관성이 있어야 한다. 코딩과 마찬가지로 문서를 쓰는 사람은 일관된 표현을 사용해야 하며, 사용하는 용어에 대해서도 항상 고민해야 한다. 문서는 큰 그림을 보여주고 세부사항들을 하나씩 나열하는 형식으로 전체적인 구조를 이룬다. 잘 정리된 문서는 전체 문서구조 보기 메뉴와 같은 것을 열었을 때 대체로 잘 정돈된 형태로 항목들을 균형있게 나열해서 보여준다.

문서를 처음 작성하는 사람들을 특정 양식(Form)이 없는지 찾아본다. 하지만, 중요한 것은 양식이 아니라 전달하고자 하는 내용이다. 또한, 내용을 채우는 것은 일이 완료된 후에는 불가능한 경우가 많다. 새로운 일이 이미 할당되었거나 이미 지난 일에 대해서 추가로 할 일들이 있기 때문이다. 따라서, 문서는 일하는 과정에서 만들어야 한다. 개발 단계에 맞게 만들어져야 하며, 항상 이전 단계의 문서들도 현재 상황에 맞게 변경 및 유지가 되어야 한다. 그래야만 문서는 생명력을 가지게 되며(여러 사람이 문서로부터 도움을 받을 수 있기에), 향후 자신이 했던 일에 대한 기억으로 자리 잡을 것이다. 문서를 쓰는 것을 낭비라고 본다면 개인의 기억에 의존한 팀이 된다. 기억에 의존한 팀의 생산성은 영속성(Sustainability)이 없으며, 특정 몇몇 사람에 의존하는 형태가 될 것이다. "사람이 중요하다"는 것은 맞지만 "특정 몇몇의 사람에만 의존"하는 것은 좋지 않다. 모든 일은 공개되어야 하며 언제라도 누군가에 의해서 재생산될 수 있을 때 의미가 커진다.

**결과물에 대해서 검토를 지속해서 한다.**

소프트웨어 개발은 시작부터 끝까지 이어지는 검토의 연속이다. 이유는 단순하다. 사람이 만든 것 중에서 가장 고도의 정신력이 필요한 분야이기 때문이다. 언제든 더 좋은 해결책이 있을 가능성이 있고, 지속해서 개선되어야 하기 때문이다. 검토는 더 좋은 것을 만들기 위해서 여러 사람의 힘을 모으는 활동이다. 서로 다른 경험과 지식을 가지고 있기에 검토는 그런 것들을 나눌수 있는 학습의 또 다른 형태라고도 볼 수 있다. 검토는 결과물에 대한 피드백을 주게 되며, 즉각적인 반응을 끌어낼 수 있다. 잘못된 것을 즉시 고치는 것은 나중에 그로 인해서 발생할 수 있는 비용이나 일정 지연을 최소화시키는데 필수적인 사전 조치다. 검토하지 않는 조직은 상대적으로 경직되어 있을 가능성이 높다. 경직된 조직은 더 좋은 해결책이 있더라도 누군가의 힘에 의한 논리로 무시당하기 일쑤다. 결국 과제에 결정적인 순간이 올 때 그런 모든 것들이 합쳐져 아무것도 개선할 수 없는 상태가 오게 되며, 누군가의 책임을 묻는 것은 이미 늦어버렸을 것이다.

검토를 해야 할 것들은 어떤 것들이 있을까? 첫 번째는 해결해야 할 문제의 정의 자체다. 어떤 문제를 해결해야 할지를 정하고 그것이 올바른지 확인해야 한다. 시스템과 사용자 간의 협업을 정의하고, 어떤 구성요소들이 필요한지를 대략 기술한 것도 검토 대상이다. 앞에서 만든 정의된 문제를 해결하기 위해서 어떤 구성요소들이 필요할지를 정하고, 그것이 적절하게 정의되었는지 확인하는 과정이다. 구성요소들에 대한 정의 이후에는 각각의 구성 요소에 필요한 더 작은 단위의 내부 구성요소들을 정의해야 하고, 어떻게 협업해서 각각의 구성요소에 주어진 책임을 완수할 수 있는지 상세하게 정의한다. 당연히 이런 것들이 제대로 책임을 완수할 수 있는지와 관계가 올바르게 설정되었는지도 확인(검토)해야 할 것이다. 코딩은 규정(Rule)에 기초해서 제대로 했는지, 로직(Logic)은 제대로 동작하는지, 테스트는 범위는 충분한지, 적절한 인터페이스를 가지고 있으며 너무 많은 역할(Role)을 가지고 있는 모듈들은 없는지 등등도 검토되어야 한다. 버그가 발생했다면 버그의 원인을 분석하고, 그것들을 분류해서 나중에 동일한 버그의 유형이 발생했을 때를 위한 대비도 해야 한다. 과제에서 산출된 모든 유의미한 결과물에 대해서 적절한 관리조치가 이루어져야만 해야 할 일을 제대로 하게 되는 것이다.

위와 같은 조치들은 즉각적으로 효과를 볼 수 있는 것들도 있지만, 과제의 후반부나 차기 과제를 진행하는 데 도움이 되는 것들도 있다. 과제의 연속선 상에서 생각해 볼 때 무언가 부족했다는 판단이 든다면, 그런 것들은 이미 이전 단계에서 해야 했던 일이다. 늦기는 했지만 지금이라도 조금씩 개선하지 않으면, 다음번 과제를 할 때 그와 비슷한 양상을 겪게 되고, 결국에는 누적 효과로 인해서 과제는 더 비용이 들어가거나 지연될 것이 분명하다. 지속해서 개선해야 할 것은 과제 자체가 아니라 오히려 과제를 진행하는 사람들의 태도나 정신 상태(Mindset)일지도 모른다. 왜냐하면, 항상 기본에 대해서는 잘 안다고 생각하면서도 실제로는 기본을 충실히 하지 않는 소프트웨어 개발자들이 많기 때문이다. 기본은 단순함에서 나온다. 단순하게 해야 할 일을 정하고 그것을 꾸준히 실행하는 것이다. 복잡한 것을 단순화시켜서 표현하고, 단순화된 표현들을 각각의 추상화 수준에서 동일하게 유지하는 것이다. 복잡한 코드를 멀리하고 단순하고 한 가지 일만 수행하는 코드들의 협업을 통해서 문제를 해결하는 방식이다.

결과물은 문서에만 한정하지 않는다는 것을 이미 앞에서 이야기했다. 결과물에 대한 평가없이 다음으로 넘어가면 반드시 문제가 발생한다. 그리고, 그 문제는 이전 단계에서 해결할 때와는 비교할 수 없을 만큼 복잡도(Complexity)가 증가해 있을 것이다. 결국 그렇게 더 어렵게 된 문제를 해결하기 위해서 더 큰 노력과 시간이 투자되어야 하고, 변경하기도 쉽지 않아 대량의 버그를 유발할 가능성도 커지게 된다. 문제를 발생시킨 시점과 가까울수록 문제를 수정하기도 쉽다. 따라서, 반드시 모든 결과물에 대한 검토 활동을 수행해야만 비용이 적게 든다는 것을 알아야 한다. 시간이 없다고 간과하고 넘어간 것들은 가까운 미래에 형태를 달리해서 발견될 것이다. 시간이 흐를수록 과제는 더 민감한 상태로 전개될 것이고, 이전 단계에서의 조그만 문제도 그때가 되면 수면 가까이에 도달한 거품처럼 부풀어 올라 있을 것이다. 터지는 것은 시간 문제일 뿐 감추려는 노력과는 무관할 것이다.

**누구의 잘못인가를 탓하기 보다는 왜 잘못되었는지 이유를 찾는다.**

문제의 발생에는 당연히 원인이 있기 마련이다. 하지만, 일단 문제가 발생한 상황에서는 대체로 누가 잘못했는지를 찾게 되는데, 사실 모든 원인은 잘못된 과정에서 나온다. 특히 소프트웨어 개발에서는 앞 과정에서의 잘못된 일이 나중에 영향을 주는 경우가 많다. 그리고, 근본 원인을 찾아 들어가면, 대체로 과제에 대한 일정 부족, 자원 부족, 지나치게 많은 기능, 역량 부족 등인 경우가 많다. 이런 것들을 제대로 제시하지 못하면서 지나치게 낙관적으로 밀어붙이는 태도는 결국 과정의 생략을 낳게 되고 과제의 실패로 귀결된다. 따라서, 자신이 권한이 있는 사람이라면, 결코 해서는 안 되는 것이 있다. 터무니없는 무리한 것들을 “강요”하는 것이다. 어차피 실패할 것이라면 차라리 처음부터 하지 않는 것이 올바른 선택일 것이다. 가능하면 그런 과제를 안하는 것이 실패를 조금이라도 피하는 기술이다.

어쨌든 일단 문제가 발생했다면 문제의 주변 상황만을 해결하려고 해서는 안 된다. 반드시 근본적인 원인을 찾아서 해결해야 한다. 하지만, 이것도 사실 어렵다. 소프트웨어 개발에서는 일정이 흐를수록 문제를 발견해도 그 근본 원인을 고치기가 힘들어지기 때문이다. 소프트웨어는 시간이 흐를수록 굳어가는 콘크리트와 같다. 초기에는 쉽게 수정을 할 수 있을 정도로 말랑말랑한 유체에 가깝지만, 점차 딱딱해져서 나중에는 고치기 위해서 전체를 허물고 다시 지을 수밖에 없게 된다. 소프트웨어는 점차 수정에 들어가는 비용이 시간이 흐를수록 증가한다는 것이다. 흔히 소프트웨어가 "소프트(Soft)"하다는 잘못된 생각으로 인해서 언제든 수정이 가능할 것으로 보는 사람들이 많지만, 이것은 문제의 근본 원인을 고치는 것이 아니라 단편적으로 주어진 상황만 모면하는(완화하는) 것일 뿐이다. 소프트웨어의 바탕을 이루는 구조적인 결함은 이런 식으로 고쳐지지 않으며 최적화는 튼튼한 구조에서만 이루어지는 성과다. 이른 최적화를 추종하면 빨리 경화(Hard)된 소프트웨어만 얻을 수 있을 뿐이다.

누군가를 탓하기는 쉽다. 그리고, 그 모든 잘못을 그 사람에게 씌우고 나면 뭔가 자신은 아무런 잘못을 하지 않는 것처럼도 생각 될수도 있을 것이다. 하지만, 멀리 보면 어쨌든 누군가는 해결해야 할 문제이고 그것을 처음 시작하도록 지시한 책임은 다른 사람에게 따로 있다. 최소한 임원에 해당하는 사람들이라면 마치 자신이 제대로 했는데 상대방이 따르지 않았다고 이야기할 것이 아니라, 상대방이 따를 수 있는 상황도 함께 같이 만들어 주어야 한다. 이미 자기 마음대로 결정한 것들을 일방적으로 통보하고, 회식자리를 만들어서 어쩔 수 없이 책임감을 느끼도록 유도하는 것은 올바른 방법이 아니다. 그리고 나서 나중에 문제가 발생하면 모든 책임을 상대방에게 묻는것은 솔직히 너무 유치하다. 유치하다 못해 헛웃음이 나올 지경이다.

소프트웨어 개발 과제에서 발생하는 대부분 문제는 적절한 시기에 적절한 활동(Activity)이 되지 않아서 발생한다. 따라서 문제의 원인은 과거로 거슬러 올라가 간과했던(생략했던) 것들을 챙겨 보아야 한다. 다음 번에 해야 할 과제에 대해서도 그 원인의 추적은 반드시 필요하다. 실패했다면 과제의 실패 원인이 무엇인지 분석해봐야 하고, 버그가 많이 발생했다면 다음 번 과제에서는 어떻게 하면 버그를 줄일 수 있는지 고민해야 할 것이다. 성능이 원하는 만큼 나지 않는다면 소프트웨어 구조를 다시 검토해야 하고, 일정의 지연이 발생했다면 일정 추정을 제대로 했는지도 반성해야 한다. 어쨌든 중요한 것은 지금이 아니라 항상 미래다. 물론 지금 제대로 하지 않는데 미래에 제대로 된 결과를 만들어 낼 수는 없을 것이다. 지금 이 순간 해야 할 일을 다시 검토하고 즉시 적용하는 것이 가장 좋은 선택이 될 것이다. 후회는 나중에 하더라도 행동은 지금해야 한다.

**실행에 집중한다.**

많은 회사가 실행에 집중한다고 이야기한다. 하지만 대부분 회사는 행동보다 보고에 집착한다. 무슨 일이 생기면 TF를 만들고, 그 일이 해결된 이후에도 계속 지속한다. 한번 만들어진 회의는 웬만해선 잘 안 없어지는 것이 현실이다. 매주 이러한 것들로 인해서 해야 할 숙제는 늘어나고, 결국에는 못다 한 숙제를 하기 위해서 오랫동안 회사에 남아 있게 된다. 휴가는 남들의 이야기가 되고, 숙제를 안 만들기 위해서 잘못한 것들은 보고도 하지 않는다. 팀 내부에서 비밀 장부를 관리하게 되고, 절대 외부로는 유출되지 않도록 입단속을 시킨다. 도대체 왜 이렇게 사는 걸까?

실행에 집중한다는 말은 더 좋은 품질의 소프트웨어를 적절한 시기에(고객들이 원하는 시기에) 주어진 비용으로(남아도는 경우는 거의 없는) 만드는 데 모든 노력을 다한다는 것이다. 따라서, 필요한 회의는 앞에서 이야기한 것들을 위한 것이어야 한다. 그렇지 않은 회의는(각종 보고 및 누가 무엇을 했는지, 버그가 몇 개이고, 옆집에는 누가 사는데 얼마나 많은 돈을 버는지 등등) 의미가 없다. 툭하면 만들어지는 TF는 반드시 일정을 정해야 하고, 3개월 이상이 걸리면 차라리 팀을 만들 것을 권한다. 30%의 노력은 무엇을 하고, 60%는 다시 다른 일을 하고, 나머지 10%는 관리하라고 이야기하면, 3가지 일을 아무것도 제대로 하지 못한다. 그냥 "지금은 이 일에 집중하세요."라고 지시하는 것이 옳다. 나머지 일들은 급하면 다른 사람에게 시켜야 한다. 멀티태스킹은 기계가 할 일이지, 사람이 할 일은 아니다. 특히 평범한 대부분 사람에게는 어울리지 않는다.

갑작스럽게 모든 결함을 다 조사해서 누가 몇 개의 결함을 왜 만들었는지 원인을 파악하라는 지시를 하면, 진행하고 있는 과제는 하지 말라는 것으로 받아들일 수밖에 없다. 가뜩이나 한참 SQA(소프트웨어 품질보증: Software Quality Assurance)를 진행하고 있는 상황에서 던져진 숙제는 가뜩이나 힘든 상황에 어려움만 가중한다. 몇만 개나 되는 테스트를 개발자들이 배포하기 전에 소위 말하는 "자주 평가"를 위해서 실행하라고 한다면, SQA는 필요 없는 조직이 된다. 이미 개발자들이 다 테스트하는 상황인데 이중으로 일할 필요가 있을까? "거부(Reject)"되는 개선 사항들이 늘어나고 있다는 것은 분명 좋은 징조는 아니지만, 그렇다고 모든 테스트를 다 실행하고 SQA에 전달할 수는 없다. 그리고, "회기 테스트(Regression Test)"의 개념도 없고 그런 준비가 하나도 안 된 상황에서는 소프트웨어 구조의 복잡함으로 인해서 발생하는 "부작용(Side Effect)"은 어느 정도 감수할 수밖에 없다. 자주 배포를 하라고 이야기하는 것은 배포가 완전히 자동화되어 있을 때나 가능하다. 그렇지 않을 때는 배포를 준비하는 것도 오버헤드가 된다.

실행에 집중하기 위해서는 그것을 뒷받침할 수 있는 문화와 그것을 실행할 수 있는 인프라가 있어야 한다. 단순히 구호만 외치고 그렇게 안 된다고 호통을 칠 일이 아니다. 호통치면 숙제가 생기고, 그것에 대한 자료를 만들어야 하고, 대책 회의도 해야 하고, 회의 끝에 나온 "실행 목록(Action Item)"에 실무 담당자를 지정해야 하고, 그 실무 담당자들은 다시 과제에 30% 숙제에 60% 잡무에 10%의 시간을 들여야 한다. 그리고, 매주 반복되는 2시간짜리 회의에 들어가서 5분간 발표하고, 나머지 긴 시간은 빠져나오지 못하고 “명상(?)”으로 보내야 한다. 실행에 집중하는 것은 단순하다. 그냥 그들이 원하는 것을 들어주기만 하면 된다. 소프트웨어 개발자들은 많은 것을 원하지 않는다. 자신이 제대로 하고 있다는 느낌이 들기 위해서 스스로 길을 찾아갈 것이다. 당신은 옆에서 지친 그들의 어깨를 살짝 어루만져 주기만 하면 된다.

# [ 야근에 대한 단상 ]

소프트웨어 업계의 만연한 부조리 중의 하나가 "야근"이다. 그와 더불어 "주말 특근"은 이미 어려운 상황에 있는 개발 과제를 정상화할 수 있는 유일한 대안으로 생각한다. 어떤 과제 책임자는 과제의 시작과 더불어서 야근을 밥 먹듯이 하라고 압박한다. "토요일 나오지 않을 사람들은 미리 보고하세요"라는 말로 교묘하게 사람을 지치게 만든다. 과제의 일정을 기획할 때도 마찬가지로 “MS-Project”에서 주말을 전부 일하는 것을 가정해서 수립한다. 휴가는 안 가는 게 당연하고 과제를 마친 후에(혹은, 세계 1등이 된 후에) 가라고 한다. 소프트웨어 개발 과제의 특성상 과제를 마치고 나면 시장에서 올라오는 문제를 해결하느라 사실 휴가 갈 시간은 없다. 그리고, 세계 1등이 된 이후에는 1등을 유지하라는 “지상 최고의 명령”으로 보상으로 주어지는 특별 휴가도 갈 시간이 없을 것이다. 경영자들은 어떻게 하면 효율적으로 더 많이 일을 시킬 것인가를 생각하지, 어떻게 하면 더 많은 여유를 개발자들에게 줄 것인가는 고민하지 않는다.

사실 야근과 주말 특근은 비용을 동반하고 있다는 것을 잘 모르는 것 같다. 혹은 그 비용이 작다고 생각하는 경우가 많다. 하지만, 사람이 로봇이 아닌 이상 야근하면 그 다음 날 오전은 집중하지 못하고 낭비된다. 주말 특근을 수시로 하게 되면 금요일까지 마칠 수 있는 일도 토요일로 넘어가는 경우가 많고 월요일은 휴일처럼 보낸다. 그런 일이 일상화되면 사람들은 일정에 대해서 무감각해지며 항상 시간이 부족하다고 생각하게 된다. 연쇄반응으로 생기는 결과들은 소프트웨어 개발 과제의 프로세스에 대한 특별한 주의를 상기시키지 못하게 되며, 단계별 결과물의 수준도 떨어지게 된다. 결국 모든 간과된 비용은 테스트 단계로 집중되며, 테스트는 언제 끝날지도 모르는 반복된 작업이 되고 만다. 작은 비용들이 모이면 소프트웨어 개발로 인해서 얻어야 할 이익을 삭감시키는 요인이 되며, 이렇게 개발된 제품들이 늘어날수록 야근이나 주말 특근으로 생기는 비용보다 더 큰 비용이 유지보수에서 만들어지게 된다. 회사의 경쟁력은 일을 더 열심히 할수록 더 낮아지는 아이러니한 상황이 되는 것이다.

해야 할 일은 야근과 주말 특근을 사람이 할 것이 아니라, 다른 것으로 대체하는 것이다. 사람이 없더라도 일을 진행할 방법을 생각하고 이를 실무에 적용하는 것이다. 대표적인 업무는 "단순 반복되는 작업"이 첫 번째다. 이 중에서도 소프트웨어 개발자들이 가장 익숙하지 않은 것이 자신의 코드를 직접 테스트하는 것이며 자동화시키는 일이다. "단위 테스트(Unit Test)"라는 말은 들어봤지만 그것을 제대로 하는 개발자는 드물다. 그리고, 체계적으로 자신이 한 일을 리뷰(Review)하는 것을 마치 여러 사람 앞에서 "자아비판"쯤으로 생각하는 개발자도 많다. 또한, 소위 말하는 직급이 높은 개발자들은 자신의 경력을 화려하게 포장해서 남들에게서 받는 스트레스를 없애려고도 한다. 이런 것들이 문화라는 이름으로 이미 익숙한 상황이라면, 그 문화를 변화시키려는 행동을 가장 먼저 해야 한다. 자동화를 시키기 위해서는 비용이 든다. 하지만 그 비용이라는 것은 언제나 나중에 발생할 일에 대한 안전망을 구축하는 비용으로, 결국에는 다시 얻게 될 이익의 일부일 뿐이다. 과제를 한 번만 진행하고 그만둘 것이 아니라면 얻게 되는 이득은 복리 이자처럼 지속해서 불어나게 된다. 중요한 것은 미리 이러한 것들을 잘 구축하면 야근과 주말 특근에 대해서 민감하게 대해야 할 필요 자체도 없어진다는 것이다.

야근과 주말 특근은 될 수 있으면 피해야 한다. 만약 필요하다면 한정적으로 시행해야 한다. 그런 것들이 필요한 시점은 배포 시간이 다가왔을 때 1달에서 3달 정도다(실제로는 최대 2주를 넘기지 않는 것이 좋지만). 그리고, 급히 진행해야 할 과제의 경우에도 6개월은 절대 넘지 말아야 한다. 야근이나 주말 특근으로 인해서 피해를 보는 것은 소프트웨어 개발자들만이 아니다. 그들의 가족들도 아빠와 엄마가 없는 저녁 시간과 주말을 이겨내야 하기 때문이다. 그들에게 좀 더 여유를 주는 방법을 생각해서 업무에 적용해야지만 개인의 만족도를 높일 수 있다. 회사를 구성하는 구성원들이 행복해야 “좋은 회사”에서 오랫동안 지속적인 높은 성장을 보여주는 "위대한 회사"로 갈 수 있다. 단순히 좋은 회사에 머무르기 위해서도 개인의 만족감은 중요하다. 반복적인 지루한 일은 사람이 잘 할 수 있는 부분이 아니며, 인건비를 들이기보다는 장비를 구매하는 것이 훨씬 싸다. 시스템을 구매해서 구축하는 비용은 연간 몇 억 이하가 되겠지만, 그것을 사람의 손으로 하기 위해서는 몇 십 억이라는 비용이 나가는, 항상 오류 가능성이 있는 일이 된다. 밤늦은 시간에 퇴근하는 직원들에게서 뿌듯한 자부심을 느끼기보다는, 일찍 퇴근해서 잘 쉬고 밝은 얼굴로 아침에 출근하는 직원들이 더 아름답게 느껴지길 기대해본다.

# [ 소프트웨어 공학에 대한 교육이 필요하다. ]

소프트웨어 개발자들이 4년제 대학을 나오더라도 코딩에 익숙하지 않은 것이 현실이다. 면접을 해보면 열에 다섯 명 정도는 코딩을 제대로 하지 못한다. 물론 대학마다 차이는 있겠지만, 소프트웨어 개발자로 입사를 하고자 하는 사람들이 정작 코딩을 못 한다면, 일반 시중에 있는 컴퓨터 학원 수업을 이수한 사람들과 무엇이 다르겠는가? (물론 그런 학원을 나와서 회사에서 중요한 핵심 인재로 일하고 있는 사람들도 많이 있다. 여기서는 전혀 그런 학원 출신들을 비하할 생각은 없다.) 제대로 된 교육을 돈 내고 배웠다면 최소한 자신이 전공하는 분야에서 실무적인 역량도 필요하다. 코딩은 기초가 되는 것으로 어떤 회사를 가더라도 소프트웨어 개발자로서는 당연히 해야 하는 일이다.

요즘은 주로 Java나 C#과 같은 객체지향 언어를 많이 활용하는 방법으로 교과 과정을 진행하지만, 여전히 C/C++은 나름의 영역을 가지고 꾸준한 인기를 얻고 있다. 웹에서 사용하는 다양한 스크립트 기반의 언어도 있지만, 임베디드나 유닉스/윈도우와 같은 시스템을 개발할 때는 아직도 오래된 언어를 사용하는 것이 일반적이며, 개발자의 몸값도 쉽게 떨어지지 않는다. 일반적으로 운영체제와 같은 곳에 포함되는 소프트웨어를 “시스템 소프트웨어”라고 부르며, 하드웨어 지식과 운영체제 내부에 대한 동작 메커니즘을 이해하는 것을 포함한다. 이런 인력들은 찾기도 힘들지만 찾았다고 해서 쉽게 데려올 수 있는 인력들도 아니다. 학생이라면 한가지 언어에 치중하기보다는 다양한 언어를 접해보는 것이 좋은 이유도 여기에 있다. 어떤 회사에서 어떤 일을 할지 모르는 상황이라면 한 분야만 열심히 하는 것으로는 부족할 것이다. 물론 다양한 언어 중에서 특정 언어에 대해서는 다른 개발자보다 경쟁력을 가지고 있는 것도 좋다. 어쨌든 한가지 언어에 대해서 완전히 파악하고 있다면 다른 언어를 배우기도 그렇게 어렵지는 않을 것이다. 특히 각 언어가 가지는 특징과 개념을 이해하는 것이 중요하다.

코딩도 중요하지만 코딩을 넘어서서 일상화된 소프트웨어 개발을 지속해서 하기 위해서는 소프트웨어 공학을 공부하는 것이 좋다. 소프트웨어 공학은 이미 실무에서 일반화되어 적용되고 있는 많은 부분을 이론적으로 완성해서 개발에 대한 이해를 넓혀줄 수 있기 때문이다. 흔히 잘못 실행되고 있는 실무에서 발생하는 오해들을 바로잡아줄 수 있는 유일한 대안이 될 수 있을 것이다. 특히 경력이 10년이 넘는 소프트웨어 개발 전문 인력들은 반드시 배워야 한다. 자신들의 지난 경험들을 체계적으로 정리해 볼 수 있기 때문이다. 기존에 잘못 이해하고 있는 부분에 대한 것을 개선해서 품질이 높은 소프트웨어를 개발할 수 있는 토대를 새롭게 연마할 수 있다. 대부분 10년 이상의 경력을 가진 개발자들이 빠지기 쉬운 함정은 자신의 경험이 전부라는 태도다. 윗사람에게서 잘 못 배운 것들을 아래 사람에게 반복해서 전달하고, 그로 인해 지속적으로 낮은 품질의 소프트웨어를 만들어내는 불편한 순환고리가 만들어지게 된다.

소프트웨어 공학을 단순히 교과서적인 지식 정도로 생각하면 곤란하다. 실무에 적용할 수 없는 낡은 지식 정도라고 보아서는 안 된다. 이미 소프트웨어 개발에서 발생하는 많은 문제점은 30년 전에도 동일하게 발생하던 것들이었으며, 그 때나 지금이나 소프트웨어 개발의 대부분은 사람이 담당한다는 것은 변하지 않았다. 생산성 차원에서도 혁신적으로 개선되지는 않았으며, 앞으로도 큰 개선이 이루어지기는 어려워 보인다. 따라서 선배들의 지혜를 자신의 것으로 만드는 데 시간을 아껴선 안 된다. 교과서를 익혀야 응용해서 문제를 풀 수 있고 걸음마를 연습해야 일어나서 뛸 수 있다. 이런 이야기를 경력있는 개발자들과 하게 되면, 너무 교과서적인 이야기를 한다고 비판한다. 하지만 "정말 안다는 것은 그것을 행동으로 옮길 때"라는 것은 이미 충분히 알려진 사실이다. 머리 속으로 안다고 생각할지라도 실제에 적용하는 것은 다른 차원의 이야기다. 제대로 해보지 않은 것을 가지고 마치 자신과는 관련이 없다고 이야기하는 것은 옳지 않다. 기초는 아무리 연습해도 부족한 것이 사실이며, 그런 기초 없이는 더 높은 탑을 쌓아 올리지도 못한다.

# [ 신입사원이 배워야 할 것들 ]

새로운 해가 시작되면 새로운 사람들을 만난다. 항상 기대는 높지만 새로 만나는 사람들은 그 기대를 만족시키기 어렵다. 그리고, 그것을 항상 그래왔던 것으로 생각된다. 왜냐하면 "개구리 올챙이 시절 모른다"는 말처럼 우리 자신이 겪었던 어려움이 지금 다시 돌이켜보면 너무나 쉬운 것처럼 보이기 때문이다. 그럼 신입사원들도 같은 시행착오(Trial and Error) 과정을 반드시 거쳐야 할까? 만약 그렇게 내버려 둔다면 발전이라는 말은 일단 머릿속에서 지워야 한다. 그리고, 그들에게 기대하는 수준도 낮춰야 할 것이다. 조직의 역량은 쌓여가는 것이다. 처음 시작을 어떻게 하는가에 따라 인생이 바뀔 수도 있다. 그런 시작에 이미 충분한 실패로 증명된 사실들을 빠르게 익힌다면 쌓여가는 부분들이 늘어나게 될 것이고, 그것으로 인해서 점차 조직은 더 좋은 것들을 경험할 기회가 많아지게 될 것이다.

소프트웨어 개발 분야에 입문한 신입사원들이 배워야 할 것들은 너무 많다. 한꺼번에 모든 것을 다 배우기는 불가능하기에 시간을 두고 하나씩 알아 나가야 한다. 여기서 중요한 것은 처음 시작할 때는 기초를 튼튼히 하는 것이 필요하다는 점이다. 대부분 문제는 기본을 제대로 하지 않는 것에서 생기기 때문이다. 그리고, 대부분의 해결책도 기본에서 찾을 수 있기 때문이기도 하다. 사실 고차원적인 문제는 그렇게 많지 않다. 소프트웨어를 만들 때 알 수 있듯이 특정한 복잡한 알고리즘이 사용되는 것은 전체 코드의 10% 미만이며, 나머지는 대부분이 I/O에 관련된 단순하고 기본적인 것들이다. 문제를 해결하는 방법은 반드시 몸으로 배워야 한다. 그냥 단순히 책을 읽어서 아는 것은 크게 도움이 되지 않는다. 일단은 자신이 일하는 곳에서 사용하는 개발 인프라나 도구들에 대해서 잘 사용할 수 있어야 한다. 그리고, 최소한 자신이 자신 있게 사용하는 개발 언어를 한가지 정도는 어떤 환경에서도 개발에 활용할 수 있어야 한다. Visual Studio, Eclipse, CodeBlock등의 툴에서 C/C++을 사용해서 동일한 일을 할 수 있는 소프트웨어를 개발하는 것도 한가지 예가 될 수 있다. 중요한 점은 직접 손으로 코딩하고 실행을 시켜야만 배울 수 있다는 것이다.

현재 임베디드 시스템의 소프트웨어를 만드는 일에 종사하는 많은 실무 인력들이 가장 못 하는 것이 "단위 테스트(Unit Test)"다. 코딩은 그럴듯하게 하지만 자신이 만든 코드를 제대로 검증하지는 못한다. 테스트는 전문 테스터가 하는 것으로 생각하지 말고, 최소한 자신이 하루 하루 만드는 코드만이라도 제대로 단위 테스트를 할 수 있어야 한다. 테스트하기 위해서는 다양한 테스트 관련 프레임워크들이 존재하기에 이를 활용한 테스트 케이스 및 스크립트를 꾸준히 작성하는 습관을 들여야 한다. 이것을 적어도 1년에서 동안 꾸준히 한다면, 어떤 개발 업무를 맡게 되더라도 자신 있게 일을 할 수 있는 기초를 갖추게 될 것이다. 그러한 자신감을 바탕으로 다른 사람의 일에도 영향을 주게 되며, 팀 내에서도 변화의 중심에 있을 수 있다. (물론 반대 여론도 많을 것이다. 하지만 변화는 그런 반대를 무릅쓰고라도 옳다고 믿는 것을 하는 것이다.) 제대로 된 검증을 할 수 있다는 한 가지만으로도 개인의 개발 생산성은 큰 폭으로 높아지게 되며, 좀 더 여유 있는 삶을 살아갈 수 있는 밑천을 마련될 것이기 때문이다. 여유는 많은 것을 줄 수 있다. 팀 전체가 더 높은 수준으로 나아가기 위해서는 반드시 이러한 “여유”를 함께 만들어야 한다.

신입사원들을 그냥 시키는 대로 하면 된다는 생각을 가질 수 있다. 하지만, 제대로 시킨다는 것은 어떻게 알 수 있을까? 그것을 알기 위해서도 단순히 명령을 수행하는 것만으로는 충분하지 않다. 신입이 경력보다 더 좋은 점은 고정관념(선입견) 없이 일할 수 있다는 점과 실수를 해도 이해해주는 분위기가 만들어질 수 있다는 것이다. (그렇다고, 온통 실수만 해서도 안 되지만) 모든 것을 다 경험할 수 없기에 항상 책을 가까이 두고 읽어야 하며, 그런 책들은 이미 선배들이 책장의 한 귀퉁이에 있을 것이다. 예를 들어, "1만 시간의 법칙"이라는 것을 들어본 적이 있을 것이다. 특정 분야의 전문가가 되기 위해서는 1만 시간이라는 노력이 필요하다고 이야기한다. 신입이 해야 할 일은 "1만 시간" 동안 자신의 목표에 대해서 계획을 세워서 진행해야만 지금과 달라질 수 있다. 그 시간 동안 무엇을 할 것인가는 개인에 따라 다르기에 특정한 것으로 지정할 수 없다. 하지만 그 긴 시간 동안 꾸준함으로 무장한 노력은 몇 년이 흐른 시점에는 아주 큰 차이로 다가올 것이다. 누가 무엇이 되어 어떻게 만나게 될지는 그것을 결정하는 바로 지금 결정된다.

# [ 요원009에 대한 단상 ]

세상 어느 곳에나 "요원 009"는 존재한다. 그들의 삶의 목표는 다른 사람과 마찬가지로 행복이지만, 그들 자신의 존재 의미를 남에게서(특히, 아랫사람에게서) 찾으려는 경향이 강하다. 그들 삶의 대부분은 조직 생활에 대한 완벽한 적응에서 나온다고 믿고 있지만, 조직의 소통 장애에 대한 대부분 이유가 되는 것도 사실이다. "요원 009"는 자신이 그런 줄도 모르고 대단히 유능한 사람이라고 믿는다. 자신만이 조직을 걱정한다고 믿고 있지만, 사실 조직은 그 사람을 더 걱정한다. 영향력을 발휘할 곳을 찾지 못한 그들 나름의 열정이 회사 내의 갈등에 한몫을 할 것이기 때문이다. 그들은 투쟁이라고 생각하는 분위기를 남들은 협업이라고 부를지도 모른다.

어디를 가더라도 그런 사람은 존재한다. 그리고, 그들과 갈등을 겪고 있는 사람도 있고, 그들을 무시하는 사람도 있으며, 그들과 동화되기를 원하는 사람도 있다. 은연중에 그들에게서 배우는 사람들도 있고, 배운 것을 금방 실천하는 능력을 보유한 제2의 "요원 009"도 있다. 문제는 그런 사람들이 많아지는 분위기다. 그런 분위기에서는 소위 자기 목소리를 가진 사람들은 살아가기 힘들다. 대부분 그런 목소리를 가진 사람들은 개인의 능력이 탁월한 경우가 많으며, 그들의 비운 자리를 채우기 위해서는 여러 명의 손도 모자란 경우가 많다. 그들은 나름 "요원 009"와 대립각을 세우며 최대의 인내를 발휘했다고 생각하지만, 사실 아쉬운 부분도 많다. 조금만 더 유연한 태도를 보였다면 어땠을까? 대부분의 "요원 009"는 자신이 경험도 있고 실력도 있다고 믿는다. 물론 그런 경우도 정말 있다. 하지만, 희망 사항과 진짜 전문가는 다르다. 전문가는 그 자체로 사람들의 존경심을 끌어내지만 "요원 009"는 자신의 능력을 인정받지 못해서 안달한다.

"요원 009"는 높은 자리로 올라갈수록 더 "요원 009"다움을 발휘한다. 자신이 모든 분위기를 주도해야 하며, 자신의 결정이 절대적인 기준이라고 생각한다. 반감을 품는 다른 요원들은 끝까지 자신과 같은 사람으로 만들려고 노력한다. 다름을 인정하는 것이 아니라 동질성을 강요한다. 다르다는 것은 나쁘다는 것과 동일하게 여기고, 그런 다른 요원들을 자신의 밑에 두는 것을 치욕으로 생각한다. 하지만, 그런 사람들이 떠나고 나면 그동안 잘못되었던 일들은 떠나간 사람의 잘못으로 떠넘긴다. 떠나게 된 이유는 자신과 전혀 상관없는 것이고, 떠난 놈은 맡은 바 소임을 하지 못해서 도망간 것으로 치부한다. 이유야 어떻든 자신은 한 치의 어긋남이 없다는 것이다. 작은 것들을 강조하면서(Detail)도 큰 그림은 보지 못한다. 너무나 “디테일”을 좋아해서 자그마한 것들을 기억하기를 항상 주위 사람들에게 강조하지만, 정작 자신은 그 사람들의 디테일한 성향은 파악하지 못한다. 자신이 이야기하는 모든 것들은 반드시 지켜져야 할 규칙이지만, 자신이 듣는 이야기들은 "아, 됐고."로 끊어버린다.

그들은 사회의 강자가 되려고 한다. 그리고, 그들은 사회의 강자가 될 수 있는 충분한 처세술도 발휘한다. 하지만, 그들이 사회의 강자가 되면 그들이 지배하는 사회는 불행하다. 다양함이 갈 곳을 잃어버리게 되고 자율은 점차 사라진다. 창의적인 생각들은 다르다는 이유로 무시되며, 개선이란 먼 곳에 있는 이야기가 되고, 당장 하루하루 급한 불만 끄다가 지내게 된다. 미래 먹거리는 당장 굶고 있는 상태에서는 "Out of 안중"이다. 그들이 필요한 사람들은 자신의 말을 잘 듣는 사람들이다. 명령한 것을 아랫사람들을 이용해서 잘 처리하는 수완을 가지고 있지만, 그렇다고 아랫사람에게서 존경을 결코 받지 못하는 사람들이다. 그들은 끼리끼리 모이며 결국 또 하나의 복제된 "요원 009"가 된다. 자신은 느끼지 못하지만 남들은 그것을 다 잘 알고 있다. 마치 직급이 권력인 양 자신의 명령을 따르지 않는 사람들에게 일방적인 강요를 하고 있다는 것을 자신만이 모른다. 자, 이제 생각해 보자. 이런 사람들이 점점 많아지는 조직 혹은 사회. 당신은 그곳에 잘 적응하고 있는가? 그렇다면, 당신도 미래의 "요원 009" 예비 후보이다.

# [ 프로 정신? ]

프로 정신이란 뭘까? 간혹 이런 요청들 받는다. "프로 정신으로 일해!". 나름대로 프로 정신을 정리해 보면 이런 것이 될 것이다. 즉, "자신의 일이라는 생각(Ownership)을 가지고 지시받은 업무를 수행한다." 혹은, "문제를 해결하기 위해서 자신이 가진 모든 역량을 발휘해서 고객의 만족 수준을 넘어서는 감동을 만들어낸다." 대략 이렇게 정의할 수 있겠다. 그렇다면 “자신의 일” 과 “감동”이라는 두 가지 단어를 다시 정의해야 한다. “자신의 일”은 말 그대로 자신의 것으로 받아들이는 자세를 이야기하며, “감동”은 예상한 것 이상의 효과를 보았을 때 나온다. 따라서, 프로 정신이란 “자신의 일처럼 생각하는 것과 기대치 이상을 이루려는 마음”이라고 볼 수 있다. 하지만, 우리의 현실이 그렇게 마음먹은 데로 다 되는 세상이라면 좋겠지만 그렇지 못하다. 여기서 "왜"란 질문이 필요하다.

“자신의 일처럼 생각하는 마음”을 가지라고 이야기하면서도 반드시 결정은 다른 사람이 아닌 자신이 해야 한다는 사람들이 있다. 자신은 이미 결정 권한을 이양했다고 하지만, 매번 반복적으로 보고와 지시를 늘어놓는다. “세세함(Detail)”을 너무 강조해서 별 쓸데없는 질문들을 늘어놓기도 하고, 그런 질문들에 적절한 대답을 찾지 못했을 경우 능력을 의심하는 말들을 쏟아 놓는다. 이것은 “권한 이양”이 아닌 “책임 회피”에 가깝다. 자신은 해줄 것을 다 해 주었으니, 이젠 “당신이 책임져” 하고선 일일이 간섭하는 형태다. 결국 일이 잘못되었을 때는 모든 책임은 실무자에게 떠 넘겨진다. 물론 일이 제대로 된다면 그렇게 결정한 자신의 공을 자화자찬할 것이다. “자신의 일처럼 생각하는 마음” 은 "믿음"에 근거한다. 상대가 자신에게 권한을 이양했다는 것은 그 만큼 믿음을 준 것이며, 그런 상황에서는 세세함에 대해서도 믿고 맡겨야 한다. “세세함”을 강조하는 관리자들은 믿지 못하고 의심하는 마음을 품게되며, 지속적인 보고는 그런 의심에 확신만 더 할 뿐이다.

“만족을 넘어선 감동”은 기대 이상의 것을 보았을 때 가능하다. 이런 점에서 대부분 과제는 기대 값이라도 제대로 하자는 것으로 자신을 틀에 묶는 것이 일반적이다. 이유는 단순하다. 위험(Risk)을 감내할 정도로 자신감이 없기 때문이다. 가능하면 책임을 면할 정도의 일만 하고 싶기 때문이다. 실패를 용인하지 않는 분위기와도 밀접한 관계가 있다. 하지만, 모든 과제가 성공하는 것은 불가능하다. 그리고, 실패한 과제도 새로운 실패 방법을 발견했다는 의미 정도는 있다. 실패를 용납하지 않는다는 것의 단적인 예가 "일정 단축"이다. “도전적으로 일정을 단축”하는 과제의 대부분은 도발적으로 사용자를 무시하거나 위험를 제대로 감내하지 않았기에 얻는 것도 없는 과제다. 그렇지 않고서는 제대로 개발할 수 있는 과제가 아닐 것이다. 일정을 단축하기 위해서는 기능을 줄이거나 낮은 품질로 제품을 만들 수밖에 없다. 그리고, 그런 제품은 만족과는 거리를 둘 수밖에 없을 것이다.

개발에서 발생하는 문제는 사실 심각한 것이 아니다. 왜냐하면, 내부에서 감출 수 있는 방법들이 있기 때문이다. 비용이 많이 들고 심각한 문제는 항상 배포가 이루어진 후에 발생한다. 제품이 실제로 사용될 때 발생하는 문제점들은 기존 제품에 대한 디버깅 만을 요구하는 것이 아니라, 신규 제품의 개발도 늦추게 만든다. 개발을 담당한 인력이 발생하는 문제도 함께 해결해야 하기 때문이다. 인력적으로만 따져도 개발 비용보다 두 배 이상이 들어간다. 그런 문제가 발생하면 회사의 신뢰도 급격하게 하락한다. 이런 일이 여러 번 반복되면 해당 제품을 만들어내는 회사는 믿음을 가질 만한 품질을 제공하지 못한다는 평판을 듣게 될 것이다. 프로정신으로 개발한다는 것은 결국 스스로 고객의 장기적인 만족 상태를 유지할 방안을 찾는 것이며, 그런 방안에 제대로 힘을 실어주는 일을 경영자 수준에서 해야만 가능하다. 일방적인 강요된 책임(권한이 아닌) 이양이나 제대로 평가되지 않은 위협 요소를 무시한 개발은 절대 프로정신이 아니다. 오히려 그런 것들로 인해 시장에서 실패한 제품을 만들게 되며, 그 책임은 개인 차원이 아닌 회사에서 치러야 할 빚(Debt)이 될 것이다.

# [ 상사 까기(?) ]

직장을 다니는 사람들은 모임이 깊어가면 당연히 나오는 이야기가 직장 상사에 대한 것이다. 그리고, 주로 그런 자리의 안주는 직장 상사가 잘 못 한 점(물론 지극히 개인적인 관점에서)이 된다. 안주는 질겨야 씹는 맛이 있는지 아주 잘근잘근 씹어 댄다. 사람은 누구나 결점이 있고 사소한 실수를 하는 존재이기에 안줏거리가 떨어질 날은 없으리라. 그리고, 자신도 시간이 되면 술판의 안주가 되어 이런 저런 자리에 이야기의 중심 인물로 등장할 날이 머지않을 것이다. 하지만, 이런 자리가 되어야만 이야기가 제대로 나오는 것은 무슨 이유일까? 꼭 술이 들어가야지 친구가 되나? 혹은 술의 힘을 빌려서 그런 이야기를 해야지 속이 후련할까? 잘 못 한 당사자는 없는데, 그렇게 이야기해봐야 아무도 기억하지 않는다. 아니 기억에 남게 되면 더 힘든 일이 생길지도 모른다. 그냥 술술 넘어가도록 순리대로 따르는 게 최선일 것이다.

다면평가라는 것이 있다. 한 사람을 여러 사람이 다양한 각도로(상위 하위에서) 평가해서 그 사람의 총체적인 역량을 확인하는 것이다. 물론 이렇게 확인된 결과를 가지고 임원을 자를 수는 없다. 임원은 "임시직원"이라 더 높은 곳에서 관리된다. 하지만, 수석(부장) 정도 수준은 어떻게 변경할 수 있는 근거로 작용할 수 있다. 당연하지만 그런 사람들도 임원이 알아서 챙겨준다면 어쩔 도리가 없다. 그래도 관리 기록에는 남게 되기에 나중에 진급심사에서 동점자가 있으면 좀 더 불리하게 작용할 가능성은 있다. 직장 상사의 괴롭힘은 물리적인이지 않기에 신체의 상처로 나타나는 경우는 드물다. 하지만, 정신적인 스트레스의 주요 원인이 될 수 있으며, 만성 위염과 탈모로 인한 원형 탈모등으로 나타날 수는 있을 것이다. 업무에 대한 집중도가 저하되고, 만사가 귀찮아지며, 회사에 출근하는 것이 참가하기 싫은 행사처럼 생각된다. 혼자서 밥 먹는 횟수가 늘어나게 되며, 결국에는 밥을 회사에서 먹지 않게 된다. 이미 회사와 자신과는 아무 상관이 없다고 생각하게 되는 것이다. 혼밥이 일상이 되면 자신이 고립된고 있다는 느낌에서 벗어나지 못한다.

상사와의 대화에서 비꼬는 태도는 좋지 않다. 상사도 아래 직원들에게 비꼬는 투로 이야기하는 것은 감정을 상하게 하는 것 이상으로 가장 해서는 안 되는 행동이다. 오히려 나무랄 일이 있으면 호통을 쳐야지 돌려서 비꼬아 이야기하는 것은 "비겁한 행동"이다. 상사가 비겁하게 보일 때 아래에서 일하는 사람들은 희망이 사라진다. 자신의 10년 후 혹은 20년의 모습이 앞에서 저렇게 돌아다니고 있는데 회사에 오래 다닐 마음이 생길까? 상사를 까는 것은 그래도 회사에 다닐 마음이 있을 때나 가능하다. 이미 마음이 떠난 상태에서는 그런 것도 귀찮아지고 그냥 빨리 다른 곳으로 가고 싶어한다. 상사와의 대화라는 자리는 일종의 "묵언수행"에 해당한다. 지겨운 이야기를 지속해서 견디는 "인내"를 발휘해야 하며, 절대 마음에 담아두어선 안 되는 금지된 단어들을 귀속 밖으로 밀어내는 "자신과의 싸움"이다. 그런 말들에 속아서 세뇌되면 결국 자신도 10년 후에 혹은 20년 후에 같은 이야기를 하고 있을 것이고, 술자리의 안주가 되어 있을 것이기 때문이다. 아무리 방수가 잘 되는 옷이라고 해도 이어지는 비에는 속옷에 습기가 차기 마련이다. 절대 젖지 않을 수 없다.

상사를 까려면 좀 더 근거를 가지고 까라. 누구에게 들었던 이야기보다는 직접 경험한 이야기로, 그리고 듣고 있는 상대에게 도움이 될 만한 정보를 주라. 힘들게 돈 들여서 술까지 마시고 할 일 없이 근거 없는 말들을 하기에는 너무 아깝지 않은가? 같이 일하는 동료는 회사에서 떠나기 전까지는 같이 있을 사람들이며, 같은 배를 탄 사람에 비유하면 결국 같은 운명을 가야 할 사람들이다. 비열한 상사 밑에 비열한 사람들이 많듯이 좋은 상사 곁에는 좋은 사람들이 모인다. 좋은 사람들과 같이할 준비가 되었다면 그들과의 모임을 소중하게 생각해야 한다. "언제 어디서 무엇이 되어 다시 만날 수 있는" 사람들이기에 즐거운 자리를 만드는 게 좋다. 가끔은 까야 할 상사도 있고 가끔은 그리운 사람도 있을 것이다. 그리운 사람에게는 연락을, 까야 할 사람은 안주로 버무려 기분 좋은 밤으로 기억되어야 할 것이다.

# [ 소프트웨어 개발에서의 정치(?) ]

흔히 일을 열심히 하고 성과도 좋지만 제대로 평가를 받지 못하는 사람들이 많다. 그런데 주위를 둘러보면 일은 그렇게 열심히 하지 않으면서도 항상 성과를 높이 인정받는 사람들도 있다(물론 개인적인 생각이지만). 뭔가 차이를 인식할 수 있는 능력이 부족하거나 자신이 한 일을 과도하게 좋다고 보는 태도일 수도 있다. 자아도취에 빠진 나머지 다른 사람의 생각은 고려하지 않고, 오로지 자신이 최선과 최적을 다 만족하게 했다고 보는 것은 옳지 않다. 항상 그렇듯 문제의 해결방법은 다양하게 있으며, 최적이 이론상에서만 가능한 경우도 많기 때문이다. 최선이라고 주장하는 것도 반드시 측정 가능한 수치로 나오지는 않는다. 항상 더 좋은 방법은 시간과 노력을 더 들이면 만들어질 수 있는 것이 개발이기 때문이다. 문제는 표현이다. 자신을 제외한 다른 사람들이 내가 한 일에 대한 객관적인 평가를 어떻게 하는가를 이해하는 것이다.

소프트웨어 개발에서는 만들어진 결과물을 어떻게 적용할 것인가를 고민해야 한다. 물론 이미 과제를 시작하기 전에 고려된 것들도 있고, Vision Statement라든가, 목적 등도 다 나열했을 것이다. 하지만, 정작 중요한 것은 만들어진 결과물이 어떤 도움을 사람들에게 줄 수 있는가를 봐야 한다. 비용은 얼마나 줄였고 이익은 얼마나 거둘 수 있는지, 기존의 제품과는 어떤 차별화를 가져갈 수 있는지 등도 고려될 수 있는 부분이다. 물론 이런 수치들이 만들어진 소프트웨어 자체를 대변하는 것은 아닐지 모르지만, 다른 사람들에게는 돈으로 환산될 수 있는 것들로 보인다. 다시 말하면 과제의 결과물들을 꾸미는 것도 할 줄 알아야 한다는 것이다. 외부에서 보는 사람 입장에서는 포장이 중요할 수밖에 없다. 높은 관리자의 위치에 오른 사람들이 보는 것이 "Font"의 크기나 “화려한 그림”에 한정되어 있다고 생각할 때, 결과물을 적절한 표현으로 남들에게 쉽게 전달할 수 있다면, 글씨 모양이나 맞춤법, 화려한 그림 등 그 어떤 것이라도 동원할 필요가 있다. 핵심은 내용이지만 내용을 표현하는 것도 중요하다는 뜻이다.

정치는 팀 내 혹은 팀과 팀 사이, 그리고, 팀과 그 상위의 조직 간에 발생할 수 있다. 관리자 입장에서는 자신의 위치에 위협을 주거나 귀찮은 일을 만들 가능성이 있는 팀과는 충돌이 발생할 가능성이 높고, 도움을 받을 수 있는 조직에게는 유연한 태도를 보일 수 있다. 정치가 없을 수는 없다. 왜냐하면 “일과 성과”라는 것은 좋은 먹이감이며, 많이 차지할수록 먹이사슬의 상위에 위치할 가능성이 높기 때문이다. 주의할 것은 먹이를 너무 많이 차지하려고 하면, 다른 조직의 반감이 높아지게 된다는 점과 상부의 먹이사슬에 위치한 “맹수”에게 통째로 잡아먹힐 가능성이 있다는 점이다. 어쨌든 우리는 이런 역학관계를 무시하고 과제를 할 수는 없다. 고객에 대한 관계(내부 혹은 외부), 과제 관련자들, 경영진 등등 우리와 관련을 맺고 있는 사람들이 없다면 과제의 존재가치도 없기 때문이다. 그런 상황에서 과제를 훌륭하게 수행하기 위해서는 모두의 도움을 끌어내는 방법 또한 과제의 생존을 위해서 반드시 알아야 한다. 가장 중요한 것은 주기적인 대화이다. 대화의 형태가 보고나 검토이건 상관없이 지속해서 이해 관계에 입력을 주는 활동으로 생각해야 할 것이다.

역학관계에 묶인 사람들은 “책임과 역할”이 존재한다. 그리고, 대부분 사람은 자신의 책임과 역할을 다 하려고 노력한다. (물론 아닌 사람들도 있겠지만) 결과물이 자신의 책임과 역할에 도움이 된다면 기꺼이 도움을 주는 것이 사람이다. (물론, 그들 상호 간에도 상충하는 것들이 존재할 수 있다.) 최소한 과제가 실패하기를 원하는 사람은 과제의 이해관계자(Stakeholder)에 속하지 않을 것이라는 가정과 그런 사람들은 쉽게 파악이 가능한 경우가 많다는 점에서 중요하지 않은 사람들이다. 하지만, 과제에 관심과 기대를 걸고 있는 사람들은 다르다. 그들은 과제를 통해서 자신이 얻고자 하는 것을 찾기를 원한다. 따라서 그런 사람들의 기대가 무엇인지를 파악하고, 그것을 지속해서 만족하게 해주는 것이 소프트웨어 개발에서의 정치다. 정치를 잘 못 해서 지속해서 의심을 사게 되면, 나온 결과물에 대해서도 신뢰가 떨어진다. 싫은 사람은 어떻게 하더라도 싫은 게 인지상정이기 때문이다. 하지만, 자주 만나서 이야기를 하다 보면 만족에 대한 위험를 줄여갈 수 있으며, 더 크게는 소프트웨어 개발과제에 대한 현 상황을 모두가 정확하게 파악할 수 있도록 도와줄 수도 있다. 결국 이런 모든 활동은 상대의 도움이 절실할수록 더 필요하다. 적어도 같은 회사에서 밥벌이를 하고 있는 사람들이라면 밥상을 걷어차버리는 일은 될 수 있으면 하고 싶어하지 않을 것이다. (물론 밥상 차리는 사람은 바뀔 수도 있겠지만)

# [ 이론과 실무 ]

개발 실무자들은 흔히들 개발 프로세스와 같은 것을 학교 때 해야 할 숙제 정도로 생각한다. 숙제한 후 담임 선생님께 가서 “다 했어요”라고 이야기하고, 채점이 끝나면 결과를 받아들고는 운동장에 나가서 논다. 나중에 집에 들어가서 엄마에게 채점표를 보여주고, 그 때서야 틀린 문제를 다시 푼다. 이미 결과는 나와있는 상태에서 야단만 맞고 공부하기 싫은 아이가 되어간다. 물론 좀 극단적일 수 있는 비유를 했지만, 사실 소프트웨어 개발자들은 개발 프로세스를 거추장스럽게 생각한다. 쓸데없는 숙제는 많이 해야 하지만 정작 개발할 시간을 잡아먹어서 개발을 지연시킨다고 본다. 바쁜 와중에 이런 저런 해야 할 일을 떠안기는 사람들이 좋아 보이지 않는 것은 당연하다. 하지만, 지난 50년이 넘는 시간 동안 연구된 품질이 우수한 소프트웨어를 개발하는 방법에 대해서는 공부하지도 않았으며 알지도 못한다. 특히, PL(Project Leader)이나 그룹장, 팀장들은 대부분 나이를 40을 넘겼지만, 체계적인 소프트웨어 개발 방법에 대한 교육을 받지 못하고 실무에서 쌓은 경험으로 아직도 "곰탕"처럼 우려먹고 있는 것도 사실이다.

어쩌면 바뀌어야 할 것은 개발자들의 기본 자세인지도 모른다. 새로운 직급으로 올라갈수록(혹은 새로운 일을 하게 될 때) 공부하지 않는다는 것이다. 어릴 때를 기억하면 뭔가 새로운 것을 보게 될 때 우린 곧잘 예전에는 어떻게 해왔을까를 고민하고 그것들을 나름대로 공부한다. 하지만, 경험과 직급이 올라갈수록 새로운 것을 받아들이는 속도는 늦어지며, “왜”라는 질문도 “그러려니”하는 추측으로 바뀐다. 새로운 직급에서 필요한 것은 이미 익숙하게 아는 코딩이 아니며, 좀 더 체계적으로 소프트웨어를 개발할 방안을 찾는 것들이어야 한다. 코드의 품질을 높이기 위한 다양한 방법론과 도구들은 이미 세상에 흘러넘치고 있지만, 정작 과제 관리자들은 그런 것들이 있는지 조차도 모르거나, 알고는 있지만 "시간이 없다"라는 핑계로 사용하지 않는다. 어떤 사람은 "그건 그냥 학교에서 공부하라고 있는 것이지 실무에서는 전혀 도움이 안 돼!"라고 실무 개발자들을 가르치기도 한다. 자신만 모르거나 이해를 못하는 것은 그나마 낫지만, 이제는 남들까지도 자신과 같이 만들려고 애쓴다는 것이다. 하지만, 날기 위해서는 빨리 달려야 하고, 빨리 달리기 위해서는 튼튼한 다리와 뛰는 방법을 알아야 한다. 또한 튼튼한 다리는 꾸준한 걷기와 나름의 기초 체력 훈련을 해야만 하는 것이 순리다. 기초없는 응용은 추락만 낳을 뿐이다.

그렇다고 모든 이론이 다 필요한 것은 아니며, 모든 이론이 자신이 개발하고 있는 과제에 적합한 것도 아니다. 하지만, 다양한 방법론들의 세부적인 지침들은 공통된 흐름을 가지고 있음을 발견할 수 있을 것이다. 즉, "요구사항을 잘 알아야 한다(혹은, Domain지식에 정통해야 한다.)", "조금씩 증가시키면서 개발해야 하고, 개발된 것들은 어떤 형식으로든 개발자의 테스트를 거쳐야 한다", "테스트는 자동화할 수 있는 것들은 모두 자동화하는 것이 좋다."등과 같은 것들을 자신들의 방법론에 끼워놓고 있다. 따라서, 아주 기본적인 것들은 공통된 지식으로 충분히 이해하고 있어야 한다. 어떤 방법론을 사용하더라도 기본기는 변하지 않을 가능성이 높기 때문이다. 그 기본기에 해당하는 것은 이미 실무적인 검증을 나름대로 거친 상태이며, 이제는 이론단계로까지 도달한 상태다. 따라서, "교과서적"이라는 비난 대신에 이미 기본으로 정착된 것들은 자신의 과제에 도입해서 적용해야만 할 것이다. 그렇지 않고 단순히 비판만 하고 있다면 결국 지난 50년 동안의 위대한 업적을 무시하고, 자신이 창조한 "무대포" 방법을 사용해서 소프트웨어 개발을 하고 있다는 말 밖에는 되지 않는다. “교과서도 제대로 알지 못하는 사람이 수능에서 좋은 점수를 받을 가능성은 없다.”

이론은 실무의 토대를 쌓고, 또한, 실무는 이론을 검증하는 역할을 한다. 즉, 서로 피드백과 기본 틀(Base Framework)을 제공하기에 둘 다 따로 놀지 않는다. 이론에 대한 피드백을 제공한 실무에서 이론을 배척할 이유는 없다. 한가지 마지막 남은 이유는 "시간이 없다"라는 것으로, 이것은 정치적인 영역에 넣어 두어야 한다. 제시된 과제 일정을 맞추기 위해서는 협상과 예측(Estimation)을 적절히 구사해야 하며, 불가능한 일정에는 “기능의 삭제나 낮은 품질”로 답해야 할 것이다. 불가능한 자원이라면 일정이라도 마음 놓고 무제한으로 늘릴 수 있도록 만들어야 한다. 물론 이런 것들은 실패할 가능성이 높다. 누구도 그런 과제를 하라고 내버려 두지는 않는다. 하지만, 협상의 폐로는 사용할 수 있을 것이다. 결국 실패의 책임은 개발자가 떠안는 현실에서는 최대한 “BJR(배째라)정신” 같은 무대포도 필요하다. 하지만, 자신이 하는 실무에서는 최대한 논리적으로 일하려고 애써야 하며, 이미 밝혀진 사실들에 기반을 두려고 노력해야 한다. 남들이 무식하다고 자신도 무식해야 할 필요는 없다. 생각 없는 PL은 과제를 망치지만, 생각없는 경영진은 회사를 망친다. 물론 작은 실패의 반복은 개발자도 망치게 된다. 스스로 최선을 다하고자 하면 이제는 남들이 지난 시간동안 어떤 것들을 이룩했는지 충분히 배울 때가 된 것이다.

# [ 소프트웨어 개발자의 면접 ]

소프트웨어 개발자를 뽑는 면접에 참여해 본 적이 있는 사람은 잘 알겠지만 우리가 사람을 뽑는데 쓰는 시간은 너무 짧다. 대략 30분 정도 내외가 되는데, 이렇게 부족한 시간에 정말 좋은 사람을 찾을 수 있을까? 개인의 역량은 차이가 심하다고들 한다. 대략 10배쯤 차이가 난다고 보았을 때, 그런 차이를 파악하기 위한 시간으로 30분이 주어진다면, 우리가 뽑을 수 있는 최고의 인력은 30분을 어떻게 견딜 수 있는가에 달려있다. 물론 질문의 수준도 중요하다. 대체로 면접 담당자는 자신이 잘 알고 있는 부분에 관해서 물어본다. 하지만, 질문하는 사람들도 모든 것은 알지 못하며, 구체적인 부분은 더 모른다. 예전의 경험에 비춰본다면 면접관의 수준이 질문의 수준이다. 낮은 수준의 질문을 던지는 사람이 있다면 아마도 그 자신도 낮은 수준일 가능성이 높다. 그렇다고 무조건 높은 수준의 질문을 던지는 사람이 있다면, “개구리가 올챙이 시절을 모르는 것” 밖에 되지 않는다.

낮은 질문의 대표적인 사례가 특정 언어에서 지원하는 키워드를 묻는 것이다. 물론 그 정도는 해당 언어를 일상의 개발에서 사용한다면 잘 알 수 있겠지만, 그것으로 그 사람의 전체 역량이 표현되는 것은 아니다. 사실 질문을 던지기 전에 우리가 뽑으려는 사람이 어떤 사람인지를 먼저 아는 것이 중요하다. 우리가 뽑는 인력의 역량이 특정 키워드만 잘 아는 사람의 수준이면 그렇게 뽑아도 무방하다. 하지만, 신입 사원과 경력 사원을 뽑는 것이 다르듯 사람을 쓰려고 하는 분야에 맞게 알맞은 질문을 던져야 한다. 아키텍트를 뽑는다면 최소한 설계가 무엇이며 어떻게 만들어 지는지를 질문해야 할 것이다. 곁들여서 소프트웨어 공학에 대한 질문 및 사람과의 관계 및 관리적인 부분에 대해서도 적절한 질문을 준비해야 할 것이다. 이런 사람들에게 특정 언어의 문법을 아는지에 대한 것을 질문한다면 별로 대답할 수 있는 것이 없을 수도 있다. 만들고자 하는 시스템의 내부 구성요소를 어떻게 정의할 수 있는지 묻는 것은 좋을 것이다. 하지만, 마찬가지로 면접 위원도 시간이 부족하니 급하게 생각나는 질문보다 미리 뽑을 사람에게 적합한 질문을 준비해서 가져 가도록 해야 할 것이다.

소프트웨어 분야에서 일하기 위해서는 기본적으로 코딩은 할 줄 알아야 한다. 하지만, 학교를 이제 막 졸업한 경우에는 코딩보다는 영어에 더 신경을 쓰는 것이 대한민국의 현실이다. 안타깝지만 대학을 갓 졸업한 인력 중에서 뽑아서 바로 현업에 투입할 수 있는 인력은 적다. 그리고, 업무에 투입하더라도 제한된 지식을 가지고 있는 경우가 많아서 새로 공부를 시켜야 하는 경우가 대부분이다. 물론 제대로 좋은 인력을 뽑더라도 가지고 있는 역량을 제대로 활용할 수 있는 업무를 시키지 못할 가능성도 있다. 회사는 바쁜 업무(지금 당장 비어있는)에 적합하지도 않는 사람을 투입하는 경우가 많으며, 그렇게 배치된 인력은 이전에 자신이 알던 지식을 제대로 사용할 가능성은 없다. 새로운 분야에서 새로운 업무를 처음부터 다시 배워야 한다면 언제쯤 제대로 된 사람을 쓸 수 있을까? 그나마 나이라도 어린 경우에는 발전 가능성을 보고 쓸 수도 있지만, 경력 사원을 그렇게 배치하면 결국 도태될 새로운 인력을 쓰는 것과 무엇이 다를까? 기본적으로 사람을 뽑는 것은 회사의 미래를 위해서 가장 중요한 일이다. 따라서, 30분의 시간이 중요한 것이 아니라 현재 우리가 부족한(미래를 위해서 필요한) 인력이 어떤 사람인지를 먼저 정의해야 한다. 그냥 무턱대고 뽑아서 아무렇게나 배치한다면 심각한 낭비만 부를 뿐이다.

면접에서 물어보는 질문은 편협하지 않아야 하지만 충분히 깊이도 있어야 한다. 너무 깊은 전문지식을 묻는다면 대답하기 어렵기에 면접자의 수준에 맞는 다양한 질문을 준비해야 한다. 다른 방법은 면접자에게 자신의 수준을 미리 적어서 가져오도록 하는 방법도 있다. 예를 들어, 레벨(Level)을 0~10까지로 정의한 다음 면접자에게 자신의 수준이 어디쯤 해당하는지 표시하고, 해당 수준에 필요한 전문지식으로 무엇이 있는지도 요약해서 쓰도록 요구한다. 그렇게 만들어진 것들을 읽어서 해당 분야에 전문가 수준이라고 생각할 수 있는 사람을 면접위원으로 선정한 후, 각각의 면접 위원들이 차례로 면접자가 대기하고 있는 방으로 들어가서 면접을 진행하도록 한다. 면접자에게는 한 번에 한 명에게만 집중할 수 있어서 좋고, 면접 위원들에게는 다른 사람의 편견이 개입할 여지를 줄여서 좋다. 면접자는 스스로는 충분한 면접 시간을 가질 수 있기에 편하게 이야기할 수 있는 분위기가 만들어질 수 있을 것이다. 물론 이런 식으로 한다면 생각보다 많은 시간을 투자해야 한다. 하지만, 나중에 1년이 될지 10년이 될지도 모를 시간 동안 잘못된 사람을 뽑아서 사용하는데 필요한 비용보다 더 적은 비용이 들어갈 것은 확실하다.

# [ 주말 특근에 대해서 ]

15년 전(2000 초반)에는 토요일 4시간 근무하는 것이 일상적인 일이었다. 그때는 토요일 나오는 게 당연하고, 또한 토요일 오후에는 여유를 가지고 개인적인 일들을 볼 수 있었다. 요즘은 소프트웨어 개발자들의 업무가 폭주하고 있는 상황이라, 토요일이 쉬는 날임에도 불구하고 회사에 출근하는 사람들이 늘었을지도 모른다. 주 5일 근무하는 것에 대해서 좋다 나쁘다고 판단하는 것은 그 사람의 입장에 따라 다르겠지만, 적어도 주말에 나와서 일하는 것이 평일보다 생산성이 높은 것은 아니라는 점을 짚고 넘어가고 싶다. 예전보다 업무의 생산성이 낮아지게 만드는 요인 중에 하나라는 것이다. (2018년 6월 현재는 주 52시간 일을 못하게 하지만, 작은 회사의 경우에는 그런 것들을 지키기 어려운게 사실이다.)

토요일 일 하러 가는 사람들의 특징은 금요일까지 일이 완료가 안 된다는 것이다. 새로운 일을 토요일에 시작하지는 않는다. 그렇다고 금요일까지 끝낼 수 있는 일을 억지로 토요일까지 할까? 사실 꼭 그렇지는 않겠지만 습관적으로 일이 늘어지게 된다. 토요일 출근해야 한다면 금요일 늦게까지 노력해서 일을 끝낼 필요가 없어진다. 그리고, 토요일 반드시 출근해야 하기에 뒤로 미루는 것이 당연하다. 따라서, 주말 특근은 특근이 아닌 연장근무일 뿐이다. 토요일 주말 특근은 비용이 많이 든다. 일반적인 대기업의 경우 평일 야근은 2시간 혹은 4시간 단위로 1만 5천 원에서 3만 원 사이의 수당(교통비라는 이름으로 불리지만)을 준다. 그리고, 12시 이후에는 택시비 정도는 실비로 지원한다. 하지만, 토요일은 다르다. 토요일 근무를 하게 되면 사람들은 자신이 받는 수당을 생각해서 특별한 일이 아니라면 8시간을 기어이 채우고야 만다. 회사는 1.5배의 일당을 지급해야 한다. 이런 추가비용이 1년간 지속하고 많은 인력이 주말 특근하게 된다면, 회사의 수익도 그만큼 낮아질 수밖에 없다. 실제로 어떤 개발자의 경우에는 한달에 100만원 이상의 야근과 주말 특근 수당을 챙겨가는 경우도 많다.

소프트웨어 개발 생산성을 이야기할 때 코드 라인의 수를 보는 경우가 간혹 있다. 물론 절대적인 측도는 되지 않지만, 가장 쉽게 알 수 있는(측정할 수 있는) 수치이기에 그것을 따진다. 하지만, 생산성의 반대인 비용을 이야기할 때는 주말 특근이나 주중 야근에 대한 것은 따로 따지지 않는다. 직접적으로 돈이 관련되어 있지만 야근을 누가 더 많이 했는지만 본다. 심지어 그것을 인사 고과에 반영하는 사람들도 있다. 야근 많이 하는 조직이 일을 잘하는 조직인지, 아니면 문제를 많이 일으켜서 그렇게 야근을 많이 해야 하는지는 구분하지 못한다. 이런 현상이 발생하는 근본적인 이유는 뭘까? 원인을 찾기 위해서 관리자는 소프트웨어 개발의 결과물들을 꼼꼼히 챙겨야 한다. 그리고, 각종 도구에 의해서 객관적으로 분석된 결과물에 대해서도 신뢰해야 한다. 가장 단순한 호출 그래프(Call Graph)라든가, 혹은 복잡도나 중복된 코드 정도는 항상 추이를 지켜보아야 한다. 약간 더 세부적인 사항으로 들어간다면, 그렇게 분석된 결과들과 소프트웨어 개발에서 발생한 버그와의 상관관계도 파악하는 것이 좋다.

주말 특근이 만병통치약은 아니다. 주말은 "또 하나의 가족(회사에서 같이 일하는 팀원들과 더 오랜 시간을 보내기에 오히려 진짜 가족처럼 보이기에)"과 보내는 것이 중요하다. 집에 일이 생기면 사람들은 업무에 집중하지 못하게 된다. 집에서 쉬지 못하면 회사는 쉬는 공간이 아니다. 집에 있는 사람들의 희생을 강요하면 회사는 윤리가 상실된 조직이 된다. 더 넓은 의미에서는 사회가 제대로 발전하지 못한다. 왜냐하면, 휴식없이 일하면 감당할 수 없는 수준으로 스트레스가 올라가 결국에는 개인의 삶이 파괴될 것이기 때문이다. 쉬는 것이 더 열심히 일할 수 있는 바탕이라는 것은 누가 가르쳐 주지 않아도 이미 인간 본성에 내재하여 있다. 하루 24시간의 1/3 정도를 휴식으로 보내고 있지 않은가? 때로는 휴식없이 일하며 몇 개월을 지낼 수도 있지만, 1년이나 그 이상의 기간 동안 최고의 성과를 낼 수 있는 조직이나 사람은 흔하지 않다. 그런 사람이라면 벌써 저 멀리 더 높은 수준의 일을 하고 있을 것이다. 우리 같은 지극히 평범한 개발자는 주말이 최고의 "자아 발견"의 시간이자 일에 대한 열정을 유지해주는 소중한 시간이다. 열심히 한다고 해서 항상 좋은 결과를 만들지는 못한다. 때로는 “멍 때림”을 통해서 자신만의 휴식을 즐길 수 있는 여유가 필요하다.

# [ 소프트웨어 장인 정신 ]

"소프트웨어 개발은 장인정신(Craftsmanship)을 기초로 해야 한다."라고 하는 것은 소프트웨어 개발이 기존에 명품을 만들던 장인들이 하는 활동과 유사하다는 것을 의미한다. 대량 생산의 시대에서는 분업화가 일반화되었지만, 소프트웨어 개발자는 분업보다는 제품 개발의 모든 과정(Process)에 참여하며, 사용자에게 최고의 품질을 제공하기 위해서 노력한다. 이는 예전의 장인들이 특정 사용자를 위해서 최고의 제품을 만드는 과정과 유사하지만, 다만 소프트웨어가 가지는 무한 복제 가능성으로 인해서 사용자가 대폭 확대되었다는 점이 다르다. 하지만, 만드는 방법에서는 여전히 모든 과정을 다 장인이 진행하게 되며 결과물에 대한 평가가 장인의 평판으로 이어진다는 점에서 누가 만들었는지도 알지 못하는 분업화와는 다르다.

코딩을 보도록 하자. 더 좋은 코드를 만드는 데는 끊임없는 노력이 필요하다. 권장하는 코딩에 대한 것들은 이미 많은 이야기를 들었을 것이고 생각한다. 하지만, 아직도 대부분 시장에서 팔리는 제품에 들어간 코드들은 그런 권장 사항을 제대로 지키지 못하는 것들이 많다. 왜 그럴까? 이는 장인을 만들어내는 시스템에 문제가 있다는 뜻이다. 장인 정신으로 무장한 개발자라면 아마도 다르게 코딩을 했겠지만, 제대로 된 교육을 받지 못하고 성장한 개발자들이 자신들이 오래 전에 배운 기억으로 지금도 코딩 한다. 문제가 생기는 것은 알지만 그렇다고 개선하려고도 하지 않는다. 장인을 만들기 위한 “도제 시스템”을 통한다면 이런 것들이 불가능하다. 배우는 사람은 장인의 옆에서 멘토링을 통해 성장하게 되며, 지속적인 검토(Review)와 피드백을 받기 때문이다. 잘못된 것은 수정하고 더 좋은 코드를 만든다는 것이 어떤 것인지를 제대로 배우고 익히며 몸으로 체험하기 때문이다. 당연히 만들어진 코드는 읽기 쉽고 변경과 확장 및 테스트도 쉽게 가능할 것이다.

현실적으로는 이런 성장 시스템을 갖추기가 어렵다고들 한다. 가장 어려운 부분은 이미 고참의 지위를 차지하고 있는 소프트웨어 개발자들이다. 그들은 변화를 수용하기가 가장 어렵다. 이런 사람들은 다양한 개발툴 중에서 하나를 변경하는 것도 힘들어한다. 제품을 개발 중이라면 새로운 개발툴에 대한 적응에 시간이 걸린다는 이야기를 한다. 제품을 개발하기 전이라면 새로운 툴이 더 좋은 생산성을 보장하지 않는다는 것을 밝히려고 애쓴다. 물론 도구로 모든 문제를 해결할 수는 없다. 하지만, 제대로 된 통합개발 환경을 사용하는 것은 작게라도 생산성을 높이기도 하며, 팀 내 의사소통에 대한 통합도 작게나마 이룰 수 있다. 소프트웨어 개발자들은 코딩을 가르치는 것을 꺼린다. 그리고, 자신의 코딩에 대해서 비판을 듣는 것을 달가워하지 않는다. 이런 분위기라면 아래 사람의 코드에 대해서 자신의 방식으로 바꾸라는 지시나 명령만 있을 뿐, 코드를 보고 좀 더 개선할 부분이 없는지를 함께 토론하지 못한다. 알고리즘이나 고차원적인 설계의 중요성은 늘어놓지만 그것을 실제 구현하는 것을 얕잡아보는 경우도 있다. 정말 좋은 제품을 만들고자 한다면 좋을 설계만큼이나 좋은 코딩도 중요하다는 것을 받아들여야 한다.

유지보수에 관련된 대부분 문제는 설계의 잘못일 수도 있다. 하지만, 설계를 실현하는 구현이 제대로 되지 못하면 유지보수 비용이 대부분 이익을 삭감하는 주요인이 될 것이다. 설계의 오류를 고치는 것도 좋은 비용 절감 방법이다. 그리고, 구현을 제대로 해서 유지보수를 하기 쉽게 만드는 것도 마찬가지로 중요하다. 소프트웨어 개발과제는 한 과정(Process)만을 끌어올린다고 모든 것들이 한꺼번에 수준이 높아지는 것은 결코 아니다. 따라서, 모든 과정을 제대로 수행해야 제대로 된 제품을 만들 수 있다. 분야별 전문가도 있겠지만 전체적인 전문가가 필요하며 이런 사람들이 과제를 이끌어나가는 핵심 역할을 해야 한다. 이런 저런 정치놀음과 큰 목소리 내기에 바쁜(일명 "Big Mouth")들은 과제나 조직에 전혀 도움이 되지 않는다. 정치적인 행동도 필요한 경우가 있지만 최소한 자신이 관리하는 팀에 대해서는 자제를 해야 한다. 달콤한 목소리를 가지고 유혹하는 것도 과제에는 도움이 되지 못한다. 오히려 기본으로 돌아가 "사람이 가장 중요한 자원"이라는 것을 아는 것이 중요하다. 낮은 목소리를 내더라도 그들을 관찰하고 그들이 원하는 것들을 적극 도와주는 것이 결과적으로 성공하는 길이다. 결국 장인정신이란 사람을 통해서 전파되며 사람이 최고의 가치를 가지는 존재임을 인식하는 것이다.

# [ 사소한 일에 목숨 걸지말라!!! ]

왜인지는 모르지만 경험이 많다고 스스로 생각하는 관리자들이 흔히 사소한 것 때문에 사람들의 신뢰를 쉽게 잃어버린다. 별로 대수롭지도 않은 일에 대해서 지나치게 고집을 부려 지금까지 열심히 일하고 있는 사람의 의지를 상실하게 만드는 경우가 종종 있다. 예를 들어, 다른 회사에 회의 때문에 출장을 갔다가 오는 길에 먹은 저녁 식사비에 대한 결재를 해주지 않는다던가, 징검다리 연휴에 고향을 가기 위해 낸 휴가를 반려하는 것 등등 별로 중요하지 않은 것들에 목숨을 거는 듯한 태도를 보인다. 하루쯤 일하지 않아도 회사는 무너지지 않는다. 아니, 특정한 사람이 빠진다고 회사가 문 닫을 위험한 상황에 있다면, 그 때까지 아무런 노력도 하지 않은 관리자라고 불리는 사람들의 책임이 더 큰 것이다.

사소한 것들을 신경 쓴다고 정작 중요한 것은 놓치고 있지 않은가? 사람의 마음을 잃기는 쉬워도 얻기는 힘들다는 것을 많은 경험으로 이미 알고 있지 않는가? 윗사람의 귀에 달콤한 목소리를 들려준다고 바쁜 사람들은 아랫사람의 목소리에는 아무런 관심을 가지지 않는다. 하지만, 결국 자신을 빛내도록 도와주는 사람은 윗사람이 아니라 아랫사람들인 것이다. 그들이 성과를 내야 자신이 빛날 수 있다는 것을 결코 이해하지 못한다. 아무래도 남들의 감정에 대해서 파악하는 데는 도무지 능력이 없는 듯이 보이다가도 윗사람이 시키는 일의 핵심은 잘 잡아서 그럴듯하게 자료를 만들어 보고한다. 하지만, 자신이 정말 아는 것은 포장하는 기술일 뿐, 내용의 핵심에 대해서는 전문가의 질문에 대답이 쉽게 막혀버리고 만다. 치장(Decoration)도 기술이기는 하지만 기본이 없는 치장은 싸구려 액세서리일 뿐이다. 지나치게 신경쓰다 보면 정작 제대로 입어야 하는 옷을 엉망으로 만들 뿐이다.

사소한 일은 그냥 사소한 일일뿐이다. 그런 것들에 신경 쓰지 않도록 부하직원들의 어려움을 돌보는 것이 오히려 더 중요하지 않을까? 그런 사소한 것들로 사람들 사이의 관계가 소원해지면 큰일을 그르치는 결과를 낳고 만다. 몇 만 원도 안 되는 경비에 대한 결재를 몇 번씩이나 거부하고, 내용에는 신경쓰지 않고 폰트 크기만 보며, 그럴듯한 발표를 마치 전문가의 의견인 양 믿는 사람들은 그들이 듣고 싶은 말만 해주는 사람들로 주위를 둘러싼다. 정말 중요한 사람들은 그 벽 너머로 사라져 목소리도 들리지 않을 것이다. 그 벽에 막혀있는 당신도 그들을 보지 못하는 것은 마찬가지다. 서로 상대방의 존재감을 느끼지 못하는데 누가 책임지고 일 하겠는가? 당신은 왜 자신이 하는 이야기를 사람들이 이해를 못 하는지 모르겠다고 생각할지도 모르겠지만, 당신이 하는 이야기에 대한 비판은 용납하지 않는다. 그 사람들은 당신보다 머리가 나쁜 사람들이 아니다. 그 사람들은 당신에게 달콤한 말은 하지 못할지라도 당신이 들어야 할 말은 하려고 노력하는 사람들이다. 그 사람들도 생각하는 주체로 자신의 의견도 가지고 있으며, 의견을 무시하는 쪽은 오히려 당신이라는 생각은 왜 하지 못 하는가?

"아, 그건 됐고."라는 말로 허리를 자르고 들어오는 대화는 무의미하다. 들으려고 하지 않는 사람에게 해줄 말은 없다. 자신은 다른 사람이 말하는 중간에 치고 들어오면서도 자신의 말은 절대 끊어져선 안 된다고 생각하는 태도는 도데체 뭔가? 들을 준비가 되지 않은 막힌 귀에는 말 대신에 귀이개가 더 필요하다. 모든 것을 자신이 알아야 한다는 “똑똑한 자의 무지함"에서 깨어나야 한다. 모든 것을 알아도 판단하지 못하는 사람이 그런 사소한 것들을 일일이 머리 속에 넣고 하나씩 다 판단해 준다면, 밑에 그룹장이나 팀장, 팀원들이 필요한 이유는 뭔가? 그냥 시키는 대로 하는 기계의 부속으로밖에 생각하지 않는 것이다. 인간은 자율적인 존재이며 자신에게 권한이 주어질 때 책임감을 가지고 더 열심히 일한다. 기계 부속인 사람들에게 생각을 강요하지 말라. 당연히 일의 결과에 대한 책임도 묻지 말라. 당신이 결정하고 시키는 대로만 하기를 원했다면 책임도 당신이 져야 한다. 그들이 움직이지 않는다고 불평하지도 말라. 기계는 스스로 움직이는 법이 없다. 그것도 역시 당신이 움직여 주어야 한다. "무엇을 상상하더라도 항상 그 이하를 볼 마음의 준비를 하기를 빈다." 모두가 사소함에 목숨을 건 댓가이기 때문이다.

# [ 관리자의 역할 ]

모든 것을 관리하는 것은 아무것도 관리하지 않는 것과 같다. 관리의 핵심은 모든 것에 대한 책임과 권한을 사람들에게 나누어주고, 그것이 잘 동작하도록 환경을 제공하는 것이다. 대부분 관리자들은 이런 의견에 동의하지 않을 것이다. 그들은 자신이 권력(권한)이 있다고 믿고 있으며, 그 권력을 실현하는 활동이 관리라고 본다. 기본적으로 자신이 더 많이 알고 있다고 생각하고, 자신의 판단에 개발자들이 따르기를 원한다. 물론 좋은 판단은 필요하다. 하지만, 정확한 판단을 위해서는 개발자들의 의견에 귀를 기울여야 한다. 더 중요한 것은 관리자는 실무를 모두 챙길 수는 없다는 점이다. 대부분의 시간을 회의와 미팅으로 보내는 사람이 어떻게 실무 개발자들이 코딩하는 내용까지도 알 수 있겠는가? 실무에서 일 년만 떨어져 지내도 최신기술을 따라잡는 것은 어렵다. 일하는 방법은 경험적으로 더 잘 알 수도 있지만, 문제는 그 일하는 방법도 제대로 배우지도 못하고 있다는 것이 현실이다. 따라서, 자신이 더 잘 안다고 주장할 수 있는 근거는 희박하다. 이런 상태라면 실무 담당자들을 그냥 두고 지켜보는 것이 현명할지도 모른다. 그리고, 관리자가 정말 필요한 순간은 개발자들이 도움을 요청하는 바로 그 때다.

개발자들이 도움을 원하는 순간은 결정이 필요한 때와 자신들이 곤란한 상황에 처해 있을 때다. 이럴 때는 적극적으로 개입해야 한다. 왜냐하면, 그들 수준에서는 결정 권한이 없거나 곤란한 상황을 해결할 재정적인 지원이나 시간을 만들 수 없기 때문이다. 따라서, 그런 경우가 생긴다면 과제가 원활하게 진행되는 방법을 찾아내는 것이 관리자가 해야 할 일이다. 모든 것을 감독하려고 한다면 모든 것을 놓치게 될 것이다. 중요한 몇 가지라도 확실히 챙긴다면 과제 진행은 빨라질 것이다. 방관도 안 되며 사소한 간섭도 안 된다. 반드시 챙겨야 할 것들은 이미 잘 알려진 것들이다. 과제의 일정, 필요한 인력, 배포 계획, 테스트 계획, 리스크(Risk) 리스트, 코딩 룰의 준수 여부, 테스트되는 코드의 범위(Coverage) 등등이다. 관리자의 고유한 역할로 가장 중요한 것은 우산이 되는 것이다. 위로부터 쏟아지는 빗방울들에 직접 개발자들이 노출되지 않도록 해야 한다. 물론 위와 아래를 엄격하게 분리하라는 이야기는 아니다. 어느 조직을 가더라도 전달자의 역할과 중재자의 역할은 필수다. 하지만, 팀은 공동 운명체라는 점은 분명하며, 개발자의 잘못도 역시 자신의 잘못이라는 것을 인식해야 한다. 개발자가 잘한 일에 대해서는 반드시 보상하는 것도 잊어서는 안될 것이다.

관리란 쉽고도 어려운 일이다. 누군가의 책임을 묻기 전에 자신이 책임을 다했는지를 먼저 뒤돌아봐야 한다. 책임은 "존중"에서 생기는 것이기에 충분히 상대를 존중했는지도 고려해야 한다. 소프트웨어 개발자가 손발이 되어 일해야지 과제가 완료될 수 있다. 그리고, 그들은 손발로만 존재하는 것은 아니다. 머리를 가지고 어떻게 손과 발이 되어 일할지도 생각하는 사람들이 소프트웨어 개발자다. 시키는 대로 움직이기를 바란다면 모든 것을 관리자가 해야 한다. 기술적이건 비기술적이건 생각은 관리자만 해야 할 것이다. 그리고, 명령의 결과에 대한 책임도 당연히 개발자에게는 없다. 그들에게 머리가 있다는 것을 잊은 관리자가 책임을 져야 한다. 소프트웨어 개발은 코딩보다는 생각하는 시간이 훨씬 길다. 하루에 열 라인을 짠다고 할 때, 손으로 입력하는 시간은 채 5분도 안 될 것이다. 하지만, 그 열 라인의 코드가 나오기 위해서 나머지 근무시간 동안 무수한 생각을 하게 되고, 최적이라고 생각하는 코드를 마지막 5분 만에 입력한다. 소프트웨어 개발자들은 충분히 지적이며 존중받기를 원하는 존재들이다. 이것을 절대 잊어서는 안 된다. 하지만, 현실에는 간혹 그런 것들을 잊은 관리자들이 존재하며, 마치 자기가 없으면 일이 안 될 것처럼 생각하는 오류를 범한다. 당신이 없더라도 조직은 돌아간다. 그렇지 않았다면 수많은 기업이 희망퇴직을 받았던 것은 어떻게 설명할 것인가?

관리자의 주된 업무는 개발자들이 개발에 전념할 수 있는 환경을 만들어주는 것이다. 그리고, 필요하다면 지난 과거의 경험을 가지고 코칭해 주는 것이다. 그리고, 자신의 경험이 정말 남들에게 들려줄 만큼 타당한지는 개인적으로 검증해야 한다. 세상에는 다양한 검증되지 않은 방법들이 있으며 그중에 한 자리를 차지할 필요는 없다. 검증하기는 쉽다. 이미 많은 사람이 그런 것들에 대해서 검증을 시도했다는 것을 인터넷만 뒤져도 쉽게 찾을 수 있다. 많은 코칭 방법과 과제관리 방법, 소프트웨어 엔지니어링 분야도 있다. 이런 것들이 교과서적이고 현실에 맞지 않는다고 이야기한다면, 지난 70년간 소프트웨어 분야에서 쌓아 올린 업적들을 무시하고 자신만의 검증되지 않은 방법을 따로 만드는 것이다. 관리자가 "내가 있기에 이 조직이 있는 거야."라고 이야기한다면, 듣는 사람들은 "당신만 없다면 정말 좋은 조직일 텐데"라고 생각한다. 관리자는 자신의 역할이 공기와 물처럼 인식되어야 하지 태풍처럼 몰아쳐서는 안 된다. 다 날려버리고 난 후에 남는 개발자들은 말 그대로 엉덩이가 무거워서 떠나는 것을 겁내는 사람들뿐이다. 그들은 일에 대한 자신감이 충만한 사람들이 아니며 생산성도 높지 않다. (물론, 인내를 가지고 충직하게 묵묵히 자기 일을 하는 개발자를 싸잡아서 이야기하는 것은 결코 아니다.) 그런 사람들만 남은 회사를 원한다면 오늘의 기상예보를 "집중 강우와 돌풍"이라고 얼굴에 써놓기 바란다. 최소한 남들이 당신의 기분이라도 쉽게 알 수 있도록.

# [ 소프트웨어 개발자 문화 ]

소프트웨어 개발은 사람에 대한 이야기다. 사람만이 실제 실행 가능한 프로그램을 만들 수 있으며, 개인적인(혹은 팀의) 차이가 뚜렷이 나타나는 것이 소프트웨어 개발의 특성이다. 소프트웨어 개발에 능숙한 사람과 그렇지 못한 사람과의 생산성의 차이가 대략 10배가 난다고들 한다. 그렇다고 10명이 해야 할 일을 잘 하는 1명의 개발자가 할 수 있다는 이야기는 아니다. 못하는 사람으로 발생하는 수정비용이 5배나 더 많이 든다면, 잘하는 사람 한 명이 5배 정도의 비용을 줄 일 수 있으며 10배의 차이는 당연하게 받아들여야 할 것이다. 그렇다고10배나 더 많은 임금을 주는 것도 아니기에 개발자의 역량은 소프트웨어 개발에서 핵심 자산이라고 할 수 있다. 그렇다면 개발자의 역량에 가장 크게 영향을 주는 것은 무엇일까? 그걸 알아야만 좋은 팀을 유지할 수 있는 근간이 만들어질 수 있기 때문이다.

사람이 모여서 일하는 조직에는 나름의 방식이 있다. 업무적인 관계 이외에도 다양한 것들이 있으며, 대화나 혹은 같이 일하는 동안에 발생하는 상호 협력이 일정한 형태로 발생함을 알 수 있다. 즉, 공유되는 "문화"가 있다는 것이다. 팀의 역량에 대해서 말하고자 할 때는 개인의 역량은 일단 접어둔다. 그리고, 소프트웨어 개발은 개인의 역량보다는 조직의 역량에 달려있는 경우가 많다. 팀의 역량은 "문화"로부터 나며 그들이 하고자 하는 일에 얼마나 자신을 "투자"할 수 있는가에 달려있다. 이것은 "명령"으로 되는 것이 아니며 상호 존중이 되어야 나온다. 아이들을 키울 때 무조건적인 명령으로는 아이를 책상 앞에 잡아둘 수 있지만, 책상 앞에서 무엇을 하는가는 아이의 의지에서 나온다. 소프트웨어 개발자와 같은 성인을 아이에 비유하는 것은 어울리지 않을지도 모르지만, 인간의 본성으로 본다면 "강제"로 지시받기 보다는 자기 스스로가 정한 바를 이루려고 할 때 자신의 모든 역량을 발휘할 가능성이 높다.

책임감을 느끼게 만들기 위해서는 스스로가 결정할 수 있어야 한다. 명령받은 것을 이행하는 것은 의지가 없어도 가능할지 모르지만 결과물의 수준 차이는 크다. 그냥 돌아가는 기능과 "뛰어난 코드로 만들어진" 수준 높은 기능은 그 차이가 분명하다. 군더더기가 없을뿐 아니라 이해하기도 쉽고 코드를 읽는 사람에게 명확한 의도를 전달할 수 있기 때문이다. 의도가 불명확한 코드는 반드시 문제가 발생한다. 그리고, 그 문제가 만들어진 시점과 고쳐지는 시점의 차이는 클 것이다. 이해하기 힘든 코드는 문제를 숨기게 되며, 발견했들 때는 이미 다른 코드와 연동해서 복잡한 상황을 만들었을 가능성이 높기 때문이다. 심지어 원래 개발했던 사람이 더 이상 그 자리에 없을 수도 있다. 유지보수하는 사람도 그 의도의 모호함으로 인해서 코드를 보는 것이 어려울 것이다. 책임감은 의도를 명확히 자신의 것으로 표현한다. 스스로가 결정한 부분에 대해서 목적을 모르고 코딩하지는 않았을 것이다. 또한 자신이 그 코드에 책임감을 느끼듯이, 나중에 누가 보더라도 읽기 쉬운 코드로 만들기 위해서 노력할 것이다. 자신의 코드를 자신의 평판과 관련짓기 때문에 지속해서 코드를 개선해 나갈 것이다. 지속적인 개선이 바로 작성자의 기본 기인 것이다.

소프트웨어 개발자 문화는 최소한 팀 수준에서는 존재한다. 그 문화가 잘못된 경우는 있지만 없는 경우는 없다. "신입 사원은 코드를 수정하지 말라"고 하는 문화도 있다. 최소한 6개월 동안은 분석만 하도록 할 수도 있다. 하지만, 가장 큰 학습은 실제로 코딩해 보는 것이다. 코드는 눈으로 익히는 것이 아니라 손으로 익힌다. 손으로 동작하는 코드를 만들고 그것을 지속해서 개선해 나가는 과정을 반복해야지만 배우게 된다. 누군가 그것이 잘못되었는지 잘 했는지도 알려주어야 하고, 개선할 수 있는 부분에 대해서 피드백을 주는 것이 필요하다. 소프트웨어 개발자들이 상호 협력과 경쟁을 통해서 발전하는 문화를 만들어야 이런 부분들이 자연스럽게 이루어질 수 있다. 일방이 지시하고 다른 일방이 그것을 따르는 것이 아니라 서로 도움을 주고받는 것이 소프트웨어 개발자 문화다. 자신의 코드를 존중하듯이 남의 의견도 소중하게 들어야 한다. 더 좋은 방법이 있다면 서로 지식을 나누는 것이 필수다. "나만 가지기를 원하는 코드는 아무도 다시 사용하고 싶지 않은 코드"일뿐이다. 남들로부터 배우기를 중단하는 순간 이미 당신은 소프트웨어 개발자가 아닌 주변인에 불과함을 깨달아야 한다. 그런 주변인은 이래라 저래라 할 권리가 없다. 그냥 주어지는 대로 받아먹고 말하는데로 따르기만 하면 된다.

# [ 전문가와 비전문가의 차이 ]

소프트웨어 개발에서는 비전문가들이 넘쳐난다. 지금까지의 경험으로 비추어 봤을 때 능력 있는 전문가를 찾는 것은 정말 힘든 일이기 때문이다. 그런 사람들은 "인력시장(?)"에서 찾는 것은 어려우며, 대체로 "인맥"이나 "평판"으로 아는 사람을 통해서 찾을 수 있기 때문이다. 그리고, 그런 사람들을 데려오는 것은 금전적인 혜택이나, "자아실현"과 같은 어려운 주제를 설득력 있게 보여주어야 한다. 그런 사람들 대부분은 이미 다니고 있는 회사에서 충분히 혜택을 받고 있을 것이며, 자신의 분야에서 두각을 나타내 이직하더라도 쉽게 자리를 찾을 수 있다. 그렇다고 전문가를 구한다는 것이 완전히 불가능한 것은 아니다. 하지만, 전문가를 알아보는 "안목"이 있어야 한다. 그 "눈"을 갖추는 것이 좋은 인력을 뽑는 핵심이다. 그러기 위해서는 최소한 전문가와 비전문가를 구별하는 기준 정도는 당연히 있어야 하지 않을까?

전문가는 자신이 믿는 것을 이야기하지만 비전문가는 자신이 아는 것을 이야기한다. 전문가는 이미 아는 것의 수준을 벗어나 자신이 몸으로 직접 체험한 것에 믿음을 가지게 되며, 단순히 지적인 현학적 이야기보다는 경험담을 위주로 사람들을 자신의 믿음에 동참시킨다. 전문가는 어려운 개념을 쉬운 비유를 들어서 설명하지만, 비전문가는 어려운 이야기를 자신도 모르는 단어들을 나열해서 증명하려고 한다. 전문가일수록 어려운 개념들을 쉽게 설명한다. 비전문가는 자신이 이해하지 못한다는 것을 가리기 위해서 모호한 어려운 단어들을 나열한다. 전문가는 해야 할 일들을 이야기하지만 비전문가는 따라야 할 규칙들을 나열한다. 전문가는 적절한 시기에 적절한 처방을 내리지만 비전문가는 자신이 해야 할 일의 범위를 정하기에 정신없다. 따라서, 시기나 장소에 상관없이 나열할 수 있는 모든 좋은 것들을 동원한다.

전문가는 듣는 데 능숙하지만 비전문가는 말하는 데 능숙하다. 전문가는 다양한 사람들의 의견을 듣고 그것을 조화롭게 해결하려고 노력하지만, 비전문가는 자신이 아는 것이 전부라고 믿는다. 미숙한 자존심은 다른 사람들의 헛된 희망을 심어주거나 상처만 줄 뿐이다. 전문가는 자신이 전문가인지도 잘 모른다. 그저 끝없이 개선되고 있다는 느낌으로 자신을 평가할 뿐이다. 비전문가는 남들로부터 받는 인정이 중요하며, 특히 자신의 윗사람이 자신을 어떻게 평가하고 있느냐를 중요하게 생각한다. 전문가는 그런 면에서 동료들의 평가를 더 중요시한다. 자신과 같이 일하는 사람들이 더 중요하지 보고는 나중에 적절한 시기에 해도 된다고 생각한다. 전문가는 결과뿐만이 아니라 과정도 중요하게 생각한다. 비전문가는 결과를 내는 데 급급해서 과정을 제대로 지키려고 하지 않는다. 전문가는 전체 시스템을 파악하지만 비전문가는 시스템의 특정 부분에 대한 것만 강조할 뿐이다.

가장 큰 전문가와 비전문가의 차이점은 사용하는 단어의 깊이에서 두드러진다. 비전문가는 어려운 단어들을 입에 붙이고 살지만, 전문가는 상대의 눈높이에서 이해할 수 있는 단어를 골라서 사용한다. 즉, 어떤 사람과 대화를 하더라도 이해시킬 수 있는 충분한 용어사전을 가지고 있으며, 사용하는 용어들의 의미를 정확히 파악하고 있다는 점이다. 따라서, 깊이 있는 전문가를 뽑으려고 한다면 사용하는 단어를 위주로 파고들어 가면 알 수 있다. 물론, 이것도 한 가지 방법일 뿐이다. 자신이 전문가라고 생각된다면 자신이 가지고 있는 방식으로 사람을 뽑아보라. 그리고, 그렇게 뽑힌 사람이 나중에 어떤 결과물을 만드는지 지켜보면 알 수 있을 것이다. 안목이 있는 사람이라면 정말 제대로 된 사람을 뽑게 되겠지만, 그렇지 못하다면 자신의 주위에 있는 사람들에게 고운 눈길은 주지 못할 것이다. 세상에는 자신보다 더 똑똑한 전문가들이 많으며, 그들 모두는 하루 하루 더 나아가는 삶이 얼마나 중요한지를 즐기면서 살고 있을 것이다. 만약 이미 비전문가를 뽑았다고 생각한다면, 전문가로 성장할 수 있도록 도움을 주는 것을 아껴서는 안될 것이다. 누구나 처음부터 전문가로 태어나는 것은 아니기 때문이다.

# [ 이직하는 사람들의 공통점 ]

회사는 사람을 잃는 것을 가장 두려워해야 한다. 새로운 사람을 고용해서 안정적으로 역량을 발휘하는데까지 걸리는 시간은 생각보다 길기 때문이다. 하지만, 무슨 일인지 회사는 사람을 바꾸기 쉬운 존재로 인식한다. 사람을 일종의 무생물과 같은 자원(Resouce)으로 취급하는 "인적자원"이라는 말은 오해일 뿐이다. 사람은 "자원"이 아닌 "가치"다. 소프트웨어는 사람이 만들며 개발을 흔히들 부가가치를 더하는(만드는) 활동이라고 말하기 때문이다. 즉, 사람을 바꾸면 그 만큼의 가치가 줄어들고, 원래의 가치를 회복하기 위해서는 시간이 걸린다는 점이다. 사람은 쉽게 바꿀 수 있는 존재가 아니며, 그 사람이 가지는 역학관계와 그가 창출한 부가가치는 떠난 후에도 여전히 남는다. 하지만, 그 가치는 시간이 지나면 없어지는 것이며, 심할 때는 다른 가치들을 감가상각시키는 요인으로도 작용할 수 있다. 가치는 보존하려는 노력 없이는 유지되지 않는다.

이직하는 사람들의 이야기를 들어보면, 사람과 사람 사이에서 발생하는 마찰 때문이라고들 한다. 더 정확히 이야기하면 윗사람 혹은 같은 동료 간에 생기는 문제로 인해서 회사를 옮긴다. 그리고, 좀 더 자세히 들여다보면 "미래에 대한 희망"이 없다는 점을 지적한다. 즉, 비전(Vision)이 없는 회사에는 남고 싶어하지 않는다. 비전이 없는 직장 상사의 태도도 한몫을 한다. 마치 언제든 바꿀 수 있는 부품처럼만 여겨진다면 그렇게 할 수 있다. 그리고, 실제로도 언제든 교체할 수 있다고 이야기하는 직장 상사들도 많다. 부품처럼 여겨지는 존재는 주인의식이 없다. 부품처럼 바꿔치기 되는 존재는 책임감도 없다. 부품은 그냥 필요 때문에 사용되고 닳아 없어지면 교체될 뿐이기에 의지를 갖추고 일 할 이유가 없다. 수명이 다하는 전에 주어진 일만 하면 된다. 판단능력이나 의견을 제시할 필요도 없다. 더 나은 미래라는 것이 없기 때문이다. 미래가 없고 비전이 없는 상황에서는 더 나아지려는 개선 활동 자체가 의미를 상실한다.

연봉 계약을 할 때면 "왜 나의 가치가 이것밖에 안 되지?"라는 자신 스스로에게 질문을 던진다. 자신의 능력을 충분히 발휘했다고 생각하지만, 막상 평가를 받아보면 뭔가 불만족한 부분이 보인다. 누군가는 A+를 받고 누군가는 D-(혹은 “NI: Need Improvement”)를 받는다. 하지만, 그것이 정당하다는 근거는 항상 부족하다고 생각한다. 그 부족한 부분에 대한 불만이 쌓이면 이미 일은 "대충 대충"이라는 단어로 표현된다. 열정을 가지고 일 할 이유가 없는 것이다. 따라서, 평가는 정당한 근거를 가지고 해야 한다. 하지만, 평가는 또한 지극히 정치적이다. 가장 중요하게 보는 것은 평가하는 사람이 중요하게 생각하는 것이 무엇인가에 달려있으며, 평가자의 주관에 의해서 지배되는 것이 사실인 것이다. 평가자와 얼마나 자주 이야기하는지, 그 사람의 생각에 얼마나 동기화를 맞추는지가 핵심이다. 회사의 정책도 평가자의 의도와 상반된다면 직접적인 평가자를 따라야 한다. 이런 분위기라면 회사가 거창하게 떠들어대는 어떤 구호도 무의미하다. "법보다 주먹이 먼저"라는 이야기는 회사생활에서도 적용되는 법칙이다. 누군가 그런 분위기를 만들고 있다면 회사(조직, 팀)는 비전을 잃어버린다. 비전이 아무리 훌륭해도 그것을 실현하려는 사람이 없고, 개선보다 우선하는 개인이 있기 때문이다.

회사를 그만둔 첫날 느끼는 것은 "고요함"이다. 밤잠을 설치게 했던 수많은 고민이 일시에 사라진다. 돌아앉으면 세상이 달라 보인다. 그렇게 괴롭기만 하던 회사 생활이 자신과는 아무런 상관이 없다고 느낀다. 그리고, 그 고요함이 주는 안도감에 맘껏 몸을 맡긴다. 새로운 에너지가 쏟는 것 같기도 하고, 뭔가 지금까지 하지 않았던 일들을 하나씩 기억해 낸다. 그리고, 새롭게 생긴 의욕을 그런 일들 속에 하나씩 담아 본다. 이직은 직장인이 가지는 가장 큰 스트레스이기도 하지만, 이직이 결정된 순간의 즐거움은 회사에서 맛보는 그 어떤 즐거움보다도 크다. 자신이 원해서 일어나는 변화는 고민의 해결이 아니더라도 즐거운 것은 사실이다. "인사는 회사의 권한이다"라고 이야기하는 상사에게 "이직은 개인의 권한입니다."라고 대답할 수 있는 것이 회사다. 제대로 관리를 못 해서 혹은 일이 서툴러서 내보낸다는 상사의 이직동의 서류는 말 그대로 자산의 무능함을 스스로가 인정한다는 말일 뿐이다. 왜? 아랫사람을 결정권이 있는 사람으로 보지 않고 부속품쯤으로 사용했다는 증거이기 때문이다. 권한은 주지 않으면서 책임만 전가하는 그런 상사였다는 것에 서명하는 것이기 때문이다. 정말 역량이 부족했다면 그 자리에 올려놓은 권한이 있는 사람 자체가 문제가 있다는 뜻이기도 하기 때문이다.

# [ 변명 혹은 진실? ]

질책은 사람을 주눅들게 한다. 그런 분위기에 익숙한 사람들은 조그마한 지적에도 변명하거나 버럭 화를 내곤 한다. 뭔가 쌓여온 감정이 제대로 풀리지 않았다는 뜻이다. 이런 상황은 "과제 리뷰" 회의와 같은 곳에서 자주 볼 수 있다. 주로 질책성 발언을 많이 해야만 똑똑하다는듯이 공격적인 질문들을 마구잡이로 날리고, 한쪽에서는 그것을 위한 변명(혹은, 해명)과 같은 말들을 늘어놓는다. 요점은 과제를 잘 하자는 것이지 누군가를 깎아내림으로 해서 자신을 돋보이고자 하는 것이 아니다. 그런데도 이런 일은 반복적으로 일어난다. 그리고, 마치 누군가의 잘못을 더 많이 지적할수록 능력 있는 사람으로 받아들이는 알 수 없는 분위기도 일조하고 있다. 변명할 수 없는 것은 늦어진 과제 일정이나 현재 가지고 있는 문제점들이다. 그런 것들을 해결하기 위해서 모인 자리인데 그것을 나무라는 것에만 목소리를 높여서는 지연되고 문제가 많은 과제에 아무런 도움이 되지 못한다.

진실은 무엇일까? 과제에 문제가 생기면 가려져 있던 진실들은 서서히 수면위로 드러나게 된다. 화려한 PPT와 언변으로 감추었던 것들이 하나씩 둘씩 물 위로 떠 오르고, 지금까지 아군이라고 생각했던 사람들도 적으로 돌변한다. "신뢰"가 무너지기 때문이다. 믿고 맡겼던 사람에게서 받은 충격은 머리속에만 머물지 않고 온갖 나쁜 표현들이 입에서 쏟아져나온다. 지금까지 함께 쌓아 올렸던 모든 것이 한순간에 무너지게 되는 것이다. 진실은 마주하기 싫어서 덮어놓더라도 언젠가는 드러나기 마련이다. 하지만, 너무 이른 시간에 밝혀지는 진실도 문제가 있다. 왜냐하면, 그것으로 인해 신뢰를 쌓을 시간을 가지기도 전에 기회 자체를 잃어버릴 것이기 때문이다. 시간이 지나면 이른 시간에 알려진 진실들은 그래도 가치가 있는 것으로 판명된다. 그것을 대비할 시간을 충분히 가질 수 있기 때문이다. 너무 늦은 시간이란 제품의 배포(Release)가 얼마 남지 않은 때다. 그런 시간이라면 "진실"을 직시하는데 많은 고통이 따르는 것이 어쩌면 당연한 일일지도 모른다.

하지만, 변명은 해야 한다. 왜 그렇게 되었는지는 모두가 알고 있어야 한다. 그리고, 당신만이 그것을 완전히 책임질 필요도 없다. 과제는 혼자서 하는 것이 아니기 때문이다. 과제에 관련된 모든 사람이 책임이 있는 것이다. 개발자들만이 아니라 과제 기획을 제대로 하지 못한 상품기획, 중간 결과물에 대해 검증을 하지 않은 테스터들, 과정 준수를 보고하지 않은 프로세스 담당자들, 과제의 진척상황을 제대로 파악하지 못한 권한있는 관리자들, 자세히 검토하지도 않고 무작정 일정을 주장한 사람들은 모두 다 함께 책임을 져야 한다. 그들 모두는 과제를 진행하는데 아무런 책임이 없다고 생각한다면 그건 이미 실패할 과제를 개발팀에 주었다는 말밖에 되지 않는다. 그들이 과제를 실행해도 좋다고 판단한 근거와 믿음에 대한 책임도 그들이 가지고 있다는 뜻이다. 진실로 어려운 상황에서 외면하는 그들의 얼굴을 보는 것은 어쩌면 안타까운 생각이 먼저들 수 있다. 왜? 그들의 무지함이 그들의 경직된 표정으로 대변되기 때문이다. 그들은 자신이 하는 일이 무엇인지를 제대로 이해하고 있지 못하기 때문이다.

물론 변명만 해서는 안 된다. 이제는 현실로 들어와서 새로운 추정과 계획을 수립해야 한다. 경직된 표정을 짓던 사람들에게서 도움을 얻어야 하며, 실무 개발자들이 과제에 더 집중해서 일할 수 있는 환경을 만들어야 한다. 이 순간이 정말 중요하다. 실무에서는 문제가 많은 과제일수록 과제에 집중하지 못하는 현상이 발생하기 때문이다. 문제가 많을수록 보고서를 더 많이 써야 하고, 각종 비판적인 회의에 더 많이 참석해야 한다. 실무 개발자들도 그런 회의에 호출을 받게 되며 각종 숙제가 한꺼번에 날아든다. 평소에는 과제에 관심을 기울이지 않던 사람들이 날마다 이상한 포맷으로 보고서를 만들어서 채워 넣어주기를 요청하고, 그로 인해서 개발자들은 이런 저런 일로 뛰어다니다 하루를 다 보낸다. 결국 정말 일할 수 있는 시간은 저녁식사후 더 이상의 호출이 없는 시간이나 토요일 모임으로 임원들이 출근하지 않는 때뿐이다. 하루는 쉽게 가고 일주일도 쉽게 가지만 문제는 지속해서 줄어들지 않고 조금씩 더 늘어나게 된다. 해결책은 과제에 집중할 수 없게 만드는 것들을 줄이고 문제를 푸는 시간을 더 늘리는 것이다. 과제와 생존을 같이할 사람들에게 적극적으로 도움을 요청해야만 이런 일을 조금이라도 줄일 수 있을 것이다. 문제가 있다는 보고서는 아무리 많이 써도 문제는 결코 해결되지 않는다.

# [ 팀워크 ]

소프트웨어 개발은 팀워크가 기본이다. 한 사람이 개발하는 소프트웨어는 내부적인 사용을 목적으로 할 경우가 있을지 모르지만, 대부분 소프트웨어는 팀으로 개발을 하게 된다. 따라서, 팀이 가장 잘 운영될 방법을 찾는 것이 생산성에 가장 큰 영향을 준다. 팀장이 해야 할 일의 기본은 팀이 잘 일할 수 있도록 만들어주는 것이다. 팀장은 팀을 대변하는 사람이지 팀 자체는 아니며, 팀장의 의견은 독단이 아닌 팀원들과의 협의를 통해서 만들어져야 한다. 팀장은 팀원들이 가장 잘 일할 수 있도록 도움을 주는 존재이지 팀원들 위에 군림하는 자리가 아니다. 직급이 올라가면 높아지는 것은 책임이지 권력이 아닌 것이다. 자신의 판단이 팀원 전체를 잘못된 곳으로 이끌 수도 있으며, 권력이란 팀원들이 쥐어준 것이지 자신이 잘나서 얻은 것이 아니기 때문이다.

말을 물가로 이끌어 갈 수는 있지만 강제로 물을 먹일 수는 없다. 하지만, 우리 관리자란 사람들은 항상 목마르지 않은 말에게 억지로 물을 먹이려고 시도한다. 그들이 갈증을 내는 것은 “제대로 일하는 방법”이다. 그들은 무리한 계획들이 왜 아무렇지도 않게 자신들에게 주어지는지 정말 궁금해한다. 무리한 계획에 대한 동의가 없었음에도 불구하고, 그것을 맞추지 못했을 때는 책임지라는 말도 곧잘 듣는다. 왜 그런 책임이 그들에게 있는지 아무런 설명도 없다. 버그가 많이 발생하는 이유를 밝히고, 그 버그를 발생시킨 책임까지 묻겠다고 한다. 버그가 발생하는 원인과 결과가 뒤바뀐 것이다. 버그를 만들고 싶어서 만드는 개발자가 있을까? 아마도 그런 개발자는 없을 것이다. 최소한 자신이 만드는 제품에 대해서는 항상 최선을 다하려고 하는 것이 개발자의 기본자세다.

버그는 과거의 잘못에 대한 복수다. 즉, 제 때 했어야 할 일을 하지 않아서 발생하는 현재의 결과다. 결과만을 본다면 누군가의 실수라고도 볼 수 있겠지만, 원인을 찾는다면 그렇게밖에 할 수 없었던 과거의 결정을 보아야 한다. 라인업(Lineup) 구색을 갖춘다고 늘려놓은 모델들, 시장의 요구라고 가져온 수많은 기능들, 턱없이 모자란(잘려나간) 개발 일정, 개발자의 부족으로 인한 늘어난 외주인력, 제대로 코딩에 대한 교육을 받지 못한 중견 개발자들, 계속되는 야근과 주말 특근으로 인한 누적된 피로, 숙제와 회의로 인한 실제 업무 시간의 단축, 여기 저기서 걸려오는 전화나 큰 목소리로 떠드는 사무실에서의 낮은 집중력, 매일 화가 난 것처럼 말하는 관리자들 등등, 이런 모든 것이 버그의 원인이 아니라면 다른 원인이 따로 있을까? 책임감의 부족이 원인이라고 이야기하는 사람이 있다면, 한번이라도 제대로 된 결정할 수 있는 권한을 주었던 적이 있었는지를 먼저 물어보라.

사람은 말이 아니다. 말보다 훨씬 지적인 존재다. 그리고, 그들은 협력이라는 것을 역사적으로 오랜 시간 동안 이미 해왔다. 그들은 전문화된 영역을 가지고 있으며, 분업을 통해서 서로 자신이 잘 할 수 있는 부분에 대해 최고의 결과물을 만들려고 노력한다. 그들은 자신의 가치를 자신이 만든 결과물로 인정받기를 원하지, 아무런 근거 없는 느낌이나 편향으로 판정받기를 원하지 않는다. 팀워크란 결국 권한이 주어진 존재들이 가치가 높은 결과물을 만들어내려고 노력을 집중할 때 자연스럽게 생긴다. 단순히 명령받은 것과 똑같은 수준의 결과물을 만들어 낼 수 있다면, 그건 사람이 아니라 기계일 뿐이다. 사람은 개인의 역량에 엄연한 차이가 존재하며 잘하는 사람도 있고 못하는 사람도 있다. 그리고, 그들을 조화롭게 이끌어 더 높은 수준으로 만들어 놓는 것이 팀장의 역할이기도 하다. 그들이 자신보다 더 못한 존재라서 이끄는 것이 아니라, 먼 길을 같이 가야 할 동료이기 때문에 함께 할 수 있는 것이다.

# [ 문제를 대하는 태도 ]

소프트웨어 개발자들은 끝없는 문제의 인식(발견)과 개선이 주 업무다. 문제를 발견하는 것은 개발자만이 하는 일은 아니지만, 가장 먼저 문제를 마주치는 것은 소프트웨어 개발자다. 개발자는 문제들이 어쩔 수 없이 발생하는 일련의 과정이라고 보지만, 관리자들은 그런 "어쩔 수 없는" 문제들에 대해서 민감하게 반응한다. 근본적으로 문제를 대하는 태도가 다른 것이다. 문제의 “근본 원인”을 찾으라고 이야기하는 관리자의 대부분은 실제로 소프트웨어 개발에서 발생하는 문제를 직접 마주 선 적이 없었을 것이다. 왜냐하면, 근본 원인은 개발자들이 대부분 만들지 않고 외적인 부분에서 영향을 많이 받기 때문이다. 그렇다고 전혀 개발자의 원인 제공이 없었다는 것은 아니다. 이야기의 요지는 "근본 원인"이라는 것이 전적으로 개발의 탓만은 아니라는 점이다.

근본 원인으로는 많은 것이 있겠지만, 주로 "일정 단축 요구"이나 비이성적으로 최적화된 일정 계획에 기인한다. 그리고, 두 번째는 "적절한 기술 역량의 미확보"가 될 수 있으며, 세 번째는 "제대로 된 기능에 대한 정의"가 없다는 것이다. 물론, 이 세 가지 중에서 우선순위를 두기는 어렵지만, 과제 일정에 대한 것은 상당 부분 기술적인 것보다는 "정치적인 결정"이 많이 차지한다. 아마추어 과제 리더가 프로 관리자를 잘 못 만나면, 그저 "Yes"라는 대답을 하게 되고 상당 부분의 책임을 혼자 지게 되는 "실수"를 범하게 된다. 따라서, 문제 과제가 될 확률이 높은 과제는 제대로 일정 계획을 수립하는 절차 없이 일방적으로 전달된 일정계획이다. 설사 절차가 있다고 하더라도 이의를 제기하거나 문제점을 이야기했을 때 제대로 의견이 반영될 기회도 없었을 것이다. 주로, 회사의 최고 책임자로부터 강압적(비공식적)으로 하달되며, 그것에 대해서 이의를 제기하는 것은 절대 용납하지 않는 분위기일 것일 것이다.

개발자가 문제를 대하는 태도는 “어떤 식으로든 해결하면 된다는 것”이다. 하지만, "어떤 식"으로 해결된 문제는 조금만 시간이 지나도 다시 "원래의 문제"로 돌아오게 된다. 즉, 반복된 문제를 겪게 되는 것이다. 제대로 문제를 해결하지 않으면 그 문제는 해결된 것이 아니라 잠깐 덮어둔 것일 뿐이다. 덮어둔 문제는 언제든 재발할 가능성이 대단히 높다. "Side Effect"이라고 불러도 좋은 문제가 발생하면 QA에서는 왜 자꾸 비슷한 문제들이 발생하는지를 분석하겠지만, 원인은 결국 "제대로 된 해결방식의 부재"라고 밖에 말할 수 없다. 이를 분석하라고 지시한 관리자는 부수적으로 발생하는 문제에 대해서 "책임"을 묻겠다고 하겠지만, 앞에서 이미 이야기했듯이 이것은 "개발의 문제"가 아닌 "관리의 문제"라는 점이다. 무언가를 해결할 수 있는 적절한 방법과 시간을 주지 않은 상태에서 무조건 빨리 수정하라고 하면, 누가 문제의 "제대로 된 해결책"을 찾겠는가? 숙제를 빨리하라고 다그치면 아이들은 대충해 버리고 만다. 채점하면 당연히 틀리는 문제들이 많이 나올 것이다. 대신에 "문제를 해결하는 방법"을 시간이 아무리 걸리더라도 찾으라고 한다면 이야기는 달라진다. 문제를 분석하는 방법부터 배울 것이다.

관점의 전환은 "문제를 빨리 해결하는 것"이 아니라, "문제를 제대로 해결하는 것"이다. 그리고, "문제의 발생을 사전에 막는 조치"가 필요한 것이지, "문제가 발생했을 때 해결하는 것"은 올바른 선택이 아니다. 따라서, 두 가지를 제대로 해야 한다. "문제를 사전에 거를 방법"을 적용할 "여유"와 "문제가 발생했을 때 제대로 된 해결책"을 만들어나갈 "원칙"을 수립하는 것이다. 그것에 따라 일을 하는 개발자는 책임을 물을 수 없다. 그것에 따라 일하지 않는 개발자는 개선(일종의 경고)이 필요하다. 하지만, 실무에서는 항상 "빠른 조치"를 요구한다. 즉시 바꾸라고만 이야기하지 세부적인 것들은 알아서 하라고 한다. 알아서 하면 늦다고 호통치고 자신이 시키는 대로 하지 않는다고 나무란다. 관리자는 입으로 일을 하지 몸으로 실천하지는 않는다는 것을 제대로 된 "실제 사례"로 보여준다. 개발자들은 "실천하지 못하는 이론(말)"에 대해서는 관심이 없다. 개발자들은 안 그래도 이미 피곤하고, 그런 질책과 같은 문제 해결에 대한 요구를 받아들일 만큼 대단한 인격을 소유한 사람도 드물기 때문이다.

# [ 문제가 있는 과제를 떠맡다. ]

과제의 담당자로 일하게 되면, 하고 싶은 과제만 하는 것은 아니다. 그리고, 잘 진행되고 있는 과제를 맡을 확률은 더 낮다. 잘 진행되고 있는 과제는 담당자를 바꾸는 일이 좀처럼 없기 때문이다. 잘 진행되지 않는 과제들은 담당자가 여러 번 바뀔 수 있고, 과제의 중요한 순간(일정)을 이미 지났을 수도 있다. 그리고, 이전에 해야 했을 것들을 생략하는 바람에 생긴 문제들로 인해서 이미 일정대로 과제를 마칠 확률이 "0(Zero)"에 가깝다. 그런 과제는 될 수 있으면 맡아서는 안 되지만, 어쩔 수 없이 등 떠밀려서 맡게 되면 어떻게 해야 할까? 자신이 책임을 질 수 없는 상황에서 책임을 지라는 “부당한 명령”을 받는다면 어떻게 해결할 수 있을까? 실무에서는 이런 문제들은 흔히 발생한다. 이전에 과제를 담당하는 사람이 퇴사하거나, 혹은 그 사람의 능력을 의심한 관리자에 의해서 과제 담당자가 교체되어 나갈 때, 어려움에 처한 과제는 누군가에게 떨어지기 마련이다.

이런 과제를 맡게 되면 첫 번째로 할 일은 "과제가 문제가 있음을 공식적으로 선언하는 것이다." 즉, 지금부터 진행하는 일에 대해서는 이미 벌어진 상황을 정리하기 위한 것이지, 과제를 원래 계획대로 복구하기 위함이 아니라는 것을 모든 이해 관계자에게 알리는 것이다. 이미 과제는 손 쓸 수 없는 상황까지 온 것이고, 당신 혼자서 어떻게 한다고 목표대로 완료될 수 있을 가능성은 거의 없다. 당신이 책임감 없이 발뺌한다고 손가락질할 사람은 없다. 왜냐하면, 당신만큼 그 과제를 해결할 역량이 있는 사람이 있었다면, 이미 그 사람에게 과제가 넘어갔어야 옳다. 따라서, 자신감을 가지고 이야기해도 좋다. 이것은 "발뺌"이 아니라, 과제의 새로운 시작을 알려주는 것이다. 그리고, 과제를 완료하기 위한 조건들을 찾아야 한다. 특히, 일정에 대해서는 새로운 추정이 필요하다. 이미 지나버린 일정들을 검토해서 원래 계획된 일정들을 조정해야 한다. 새로운 인력의 투입이 필요하다고 이야기하는 것보다는 "기능의 축소"를 먼저 주장해야 한다. 새로운 인력은 일정을 줄이는 데 별 효과가 없을 것기 때문이다. 오히려 기능들에 대한 "마일스톤"을 그려서 언제 어떤 기능들이 완성될 수 있는지 확실히 해야 한다. 그리고나서, 이 모든 것들은 권한이 있는 높은 사람의 “동의"를 받을 수 있어야 한다. 그렇지 않다면 과제를 안하는 것이 옳다.

관리자들은 자신이 책임질 일을 하지 않으려고 하는 성향이 많다. 그들은 자신이 책임을 져야 하는 일에 대해서는 항상 "서명(결재)"을 늦춘다. 이런 저런 것들을 항상 따져서 빠져나갈 만한 것들은 미리 만들어 둔다. 하지만 그들이 관리자라 불리는 이유는 “책임과 권한”을 같이 가지기 때문이지 그것의 분리에 있는 것은 결코 아니다. 따라서, 반드시 "결재"를 통해서 개발 일정에 대한 수정을 이루어야 한다. 물론, 그 수정에 대한 사용자 및 QA, 기타 과제 관련자들이 동의도 해야 할 것이다. 어쨌든 여기서도 중요한 것은 "공통된 인식"이라는 점이다. 누구 하나 일방적인 생각을 강요하는 것은 늦어진 과제를 더 늦게 만들 뿐이다. 이미 늦어진 일정을 맞추라고 지속해서 "고통스러운 명령"을 준다고해서 과제에 도움이 되는 것은 아니다. 그런 말로 망가진 과제가 원상태로 복구될 수 있었다면 이미 그렇게 되었을 것이다. 궁극적으로 소프트웨어 개발은 안정적으로 동작하는 품질이 높은 소프트웨어를 사용자에게 제공하는 것이지, “폭언과 협박”으로 개발자들을 괴롭힌다고 결과물이 나오는 것은 결코 아니다. "문제가 있는 과제"는 모두가 "문제를 제대로 인식"해야 변경할 수 있다. 그렇지 못하다면, 계속 실패를 거듭할 뿐이며, 새로운 사람이 과제의 담당자로 온다고 해서 결과가 바뀔 수 있는 것은 아니다.

보고와 동의가 이루어졌다면 이제는 실제 업무를 해야 할 시간을 확보하는 것이다. 실제 업무는 완료 기준으로 "기능의 구현"이 아니라, "기능의 테스트" 여부를 따져야 한다. 80% 구현이 목표가 아니라 100% 구현과 테스트가 목표다. 즉, 구현되지 않았다는 것은 "테스트되지 않았다는 말과 동일시해야 한다." 그렇지 않으면 과제의 진행 상황을 오판할 가능성이 있다. 대부분 개발자는 자신의 구현 일정에 테스트 일정은 반영하지 않는다. 테스트는 구현과 분리된 활동으로 생각하기에 그렇다. 하지만, 대부분의 과제 기간을 차지하는 것은 "테스트와 디버깅"이다. 테스트되지 않은 코드는 완료된 것이 아니며 아직 구현 중이라는 뜻이다. 따라서, 일정을 재수립하는데 반드시 테스트 기간을 추가해 주어야 한다. QA에서 진행하는 테스트보다 개발자가 자신의 코드를 검증하는 시간을 확보할 수 있어야 한다. 이렇게 하지 않는다면 오류가 발생한 시점과 오류가 검출된 시점의 차이로 인해 QA에서 걸리는 문제들을 해결한다고 시간을 다 보낼 것이다. 늦어진 일에 대해서는 "돌다리도 두드려보고 건넌다"는 자세가 필요하다. 하나씩 검증해나가는 것이 결국 더 빨리 가는 지름길이 된다. 빨리 가고자 한다면 가는 중에 만나는 많은 장애물로 인해서 다시 돌아와야 하는 길만 길어질 뿐이다.

# [ 슬랙(Slack) : 삶의 멋과 여유 ]

사무실에서 일하는 사람들을 자세히 관찰하면 항상 바쁜 사람도 있고 항상 여유로운 사람도 있다. 물론, 그들이 일하는 수준의 차이는 있겠지만 동등한 역량을 가지고 있다고 볼 때, 누가 더 많은 일을 처리할 수 있을까? 아마도 상식적으로 항상 여유로운 사람이라고 할 수 있을 것이다. 바쁜 사람은 일이 점차 쌓여가지만, 여유로운 사람은 일이 들어왔을 때 즉각적인 반응을 할 수 있기 때문이다. 물론, 이것은 가정이 현실과 동떨어진 지나친 면이 있지만 우리의 현실을 보자는 것이다. 현실에서의 소프트웨어 개발자들은 항상 바쁘다. 새로운 과제를 개발하는 일, 기존의 과제에 대한 유지보수 업무, 각종 윗사람이 건네준 숙제, 회의자료 작성 및 회의 참석, 각종 TF, 일주일이 어떻게 가는지도 모르지만, 각종 주간보고들을 준비하는 활동들 필요하다(아마도 더 많은 일이 있겠지만). 아무리 야근을 해도 일을 다 하기도 벅찬 것이 현실이다. 물론 그렇다고 남보다 일을 못 하는 것도 아닐 것이다. 하지만, 항상 윗사람은 일을100% 이상을 할당하고도 모자란다고 생각한다.

이렇게 바쁜 상황이라면 조금이라도 추가되는 일이 생기면 모든 일이 지연된다. 하지만, 추가적인 일은 언제든 발생할 수 있으며, 지금 하는 일에도 영향을 주게 된다. 실시간 시스템에서의 CPU 사용률이 대략 69% 이상이 되면 주기적으로 실행되는 작업의 실시간성이 깨지게 된다고들 이야기한다. 즉, 작업을 마쳐야 할 시간인 데드라인(Deadline)을 넘어서게 되고, 작업의 중요도에 따라 시스템에 치명적인 영향을 주거나 작업의 결과의 유효성이 떨어지게 된다. 우리가 하는 일들은 주기성을 가진다. 특정 시점에 특정 작업이 주기적으로 진행된다. "코딩->테스트->통합->테스트" 등의 활동이 동시 다발적으로 일어난다. 주어지는 작업이 작다면 데드라인 주기가 더 짧아질 것이고, 주어지는 작업이 덩어리가 크다면 더 늦은 주기를 가진다. 하지만, 현실에서는 대체로 작은 작업이건 큰 작업이건 가리지 않고 무작위로 들어온다. 그리고, 이런 작업도 마치 실시간 태스크의 우선순위와 비슷한 면이 있다. 만약 우리 자신의 가동률(CPU의 사용률)을 69% 이상으로 항상 유지된다면 더는 다른 추가적인 작업이 끼어들 여유가 없어지게 된다. 최소한 주기적인 작업이 데드라인을 넘어서 지연을 발생시키지 않기 위해서는 그렇다는 것이다.

이런 것들은 수학적으로 증명된 것이다. 하지만, 소위 관리자라는 사람들은 자신이 관리하는 개발자의 역량을 100% 다 끌어내려고 한다. 더 많은 일을 시키고 일정도 지키도록 요구한다. 물론 개발자들이 만든 결과물의 품질도 높기를 원한다. 상식적으로 안 되는 것들을 밥 먹듯이 하는 야근과 주말 특근으로 해결하라고 지시한다. 하지만, 이런 것들이 지속될수록 쌓이고 지연되는 일들이 늘어날 것이다. 작업의 데드라인은 지키는 것이 무의미해질 것이고, 빨리 처리하다가는 품질도 놓치게 된다. 작업의 처리량도 결국에는 줄어들 것이다. 이미 쌓여있는 것들을 해결하기에도 손이 부족할 것이다. 자신의 컴퓨터가 100% CPU를 사용하게 되면 어떻게 되는가? 마우스를 조금만 움직이는 것도 힘들 것이다. 입출력(I/O)의 대표적인 HDD에 대한 작업이 늘어나면, 거의 기어가는 수준의 컴퓨터를 볼 것이다. 이 때 우리는 어떻게 반응하는가? 아마도 값이 비싸지 않다면 그냥 집어던질지도 모른다. 그게 아니라면 더 많은 돈을 주고 더 좋은 성능의 컴퓨터를 구매할 것이다. 하지만, 시간이 흐르면 이것마저도 잘 안 된다. 항상 새로운 컴퓨터를 사야 한다. 만약 한 번에 한 가지 응용프로그램만 동작시킨다면 어떨까? 대부분의 경우 낮은 사양의 컴퓨터라도 무난하게 사용할 수 있을 것이다(화려한 그래픽의 게임은 예외). 물론, 이것이 우리가 처한 현실에 대한 좋은 비유가 아닐 수도 있다. 하지만, 여기서 중요한 것은 우리 삶에 여분의 시간이 꼭 필요하다는 것이다.

더 일을 즐겁게 빨리 처리하기 위해서는 일을 더 작게 나누어야 하고, 소프트웨어 개발자에게 지나치게 많은 일을 시켜서는 안 된다. 소프트웨어 개발자는 자신이 맡은 일을 잘게 나누어 주기적으로 반복해야 한다. "짧은 코딩->짧은 테스트"의 루프를 지속해서 실행해야 한다. 작은 일은 빨리 처리되고 많은 일은 늦게 처리된다. 그리고, 문제는 많은 일을 한 번에 할 경우에 더 많이 발생한다. 함수의 길이도 짧아야 보기도 쉽고 고치기도 쉽다. 함수가 하는 일이 한 가지라면, 그 함수를 고치는 이유도 한 가지여야 한다. 짧은 반복으로 개발하게 된다면 이런 것들이 가능하다. 따라서, 코딩에서 테스트까지 걸리는 시간을 줄이는 것이 핵심이다. 좋은 코드란 결국 잘 읽히는 코드이며 쉽게 수정할 수 있는 코드다. 삶의 여유는 이런 것들 속에 녹아있다. 생각있는 관리자라면 아랫사람을 항상 100%를 활용하려고 노력하지 말아야 한다. 그들에게 여유를 주어야 그들의 삶의 질이 개선되고 일은 더 빨리 처리될 것이다. 여백이 없는 삶은 너무나 각박하다. 개인적인 발전도 없다. 여백은 그림의 바탕을 채우지만, 그런 여백이 없이는 제대로 그림을 감상할 수도 없다. 항상 바쁜 사람들은 일을 쌓아놓기만 할 뿐 처리하지는 못한다. 누구를 위한 일인지도 모른체 그저 조금이라도 일을 줄이기만 원하게 될 것이다.

# [ 소프트웨어 개발자의 목표 ]

회사 일을 하면서 목표를 가져본 적이 있는가? 개인적으로 승진이나 보너스를 목표로 일한다는 것은 일의 선후 관계를 잘 못 이해한 것이다. 승진이나 보너스는 일의 목적이 아니라 일의 결과로 따라붙는 일종의 액세서리기 때문이다. 목표은 그런 부차적인 것들이 아니다. 정말 자신이 하고 싶은 것들이 되어야 한다. 소프트웨어 개발자의 목표는 당연히 "멋진 코드"를 만들어 남들을 감동시키는 것이다. 남들이란 내부의 동료 개발자를 비롯한 외부의 사용자도 포함한다. 물론 높으신 관리자분들도 있을 것이다. 하지만, 그들을 만족시키는 것은 일단 나중의 일이다. 중요한 것은 자기 자신이 먼저 만족해야 한다. 그렇다면, 지금까지 개발한 소프트웨어에 대해서 스스로 만족하는가? 자신이 만든 코드에 대해서 충분히 자신감을 가질 수 있는가? 사실 이런 부분들에 대해서는 언제나 의문을 가지는 것이 사실이다. “내가 만든 코드가 얼마나 안정적으로 동작할까?”는 테스터로부터 검증받기 전에는 전혀 알지 못한다는 것이다. 간단한 몇 가지는 테스트했을지도 모르지만, 그렇다고 안정적일 것이라는 확신은 가지지 못할 것이다.

코드를 잘 짠다는 것은 어떤 의미일까? 아마도 코드를 짜는데 필요한 언어적인 문법을 잘 이해하고 활용할 수 있다는 뜻일 것이다. 코드가 문법적으로 완전하다는 것(오류가 없다는 것)을 확인하는 일은 전적으로 컴파일러에 의존한다. 문법에 어긋난다면 컴파일되지 않을 것이기 때문이다. 따라서, 문법적인 것을 잘 이해하고 활용하는 것은 프로그래머가 가진 가장 기초적인 역량일뿐이다. 컴파일러가 알려주는 경고(Warning) 메시지를 이해하고 이를 해결하는 것이 그 다음 단계다. 이 정도는 언어의 문법 정도는 쉽게 이해하는 사람이라면 어렵지 않게 모든 경고 메시지를 지울 수 있을 것이다. 가장 안 좋은 것은 컴파일러의 경고 메시지를 아예 무시하는 것이다. 경고 메시지가 버그가 아니라고 가볍게 생각하고 컴파일러의 경고 메시지 옵션을 꺼버린다. 컴파일러의 경고는 실제로는 "경고"가 아니다. 경고는 발생할 가능성이 있는 버그를 사람에게 미리 알려주기 위한 것이다. 공장같은 곳에서 빨간 램프가 들어오면 사람들은 지시를 따르지만, 소프트웨어 개발자는 너무나 쉽게 경고를 무시하고 그냥 지나쳐 버린다. 특별히 소프트웨어 개발자가 타고난 능력이 있는 것이 아니라면 근거없는 자만심으로 밖에 설명할 방법이 없다.

컴파일되고 컴파일러의 경고 메시지가 없다는 것을 확인했다면 이제는 진짜 문제를 찾아야 한다. 사실 문제를 찾는 방식은 인위적으로 문제를 일으켜서 찾는 것이 가장 빠르다. 즉, 입력을 주고 출력을 확인하는 방법이 시작이라고 볼 수 있다. 입력을 제대로 처리하지 못한다면 문제가 될 것이기 때문이다. 특히 단위 테스트를 하는 것은 정말 중요하다. 프로그램의 실행할 수 있는 가장 작은 단위를 미리 검증한다는 것은 나중에 있을 통합을 위한 개발자의 기본적인 의무다. 검증되지 않은 코드는 통합을 지연시킬 것이 분명하기 때문이다. 코드는 될 수 있으면 자주 통합하는 것이 좋다. 자주 통합되면 변경된 부분으로 발생하는 통합 문제를 조기에 발견하고 해결할 수 있기 때문이다. 시간이 지나면 통합에 따른 비용도 커진다. 통합 후에도 이전 단계에서 개발된 단위 테스트를 실행해 볼 수 있으며, 점진적인 통합에 필요한 자동화된 테스트들을 만들었다면 실행해 볼 수 있다. 따라서, 이런 것들에 대한 자동화를 위해서 들어가는 "공수(Effort)"는 과제에서 필요한 활동으로 명시적인 시간이 주어져야 한다. 왜냐하면, 자동화는 더 높은 품질을 가지는 코드를 제공하기 위해서 필수적인 활동이며, 결과적으로 사람의 손을 덜어줄 수 있기 때문이다. 또한 생산적인 활동에 개발자들이 집중할 수 있도록 도와주기도 하기 때문이다. 이는 일정을 늘리는 것이 아니라, 일정 속에 포함되어 일정 단축에 기여하는 “가치를 가지는 활동”이라는 점을 반드시 명심해야 한다.

개발자라면 자신의 코드가 테스트 가능하다는 것에 대해서 자부심을 가져도 좋다. 왜냐하면 다른 개발자들이 테스트를 실행함으로써 작성된 코드를 좀 더 명확히 알 수 있도록 만들었기 때문이다. 하지만, 한가지 부족한 것이 있다면 코드 자체의 품질이다. 남은 문제는 이제 가독성이다. 즉, 작성된 코드들이 읽기가 편한지 스스로 질문해야 한다. 자기가 짠 코드는 당연히 이해하기 쉬울 것이다. 그렇다면 다른 사람이 짠 코드를 보면 어떨까? 서로 자신이 작성한 코드를 남들에게 보여주고, 무슨 일을 하는지 코드 자체만으로도 충분히 설명할 수 있어야 한다. 만약 이해를 위해서 많은 시간이 필요하다면 코드에는 개선할 부분이 많다는 것으로 생각해야 할 것이다. 누군가 자신이 짠 코드에 대해서 질문해 온다면, 그것도 역시 코드를 개선해야 할 부분이 있다는 뜻으로 해석해야 한다. 주석을 늘리는 방법은 좋은 해결책이 아니다. 코드의 주석은 오류가 많은 부분이며, 컴파일러가 검증할 수 있는 부분도 아니고, 정적 분석이나 기타 툴(Tool)로도 검증할 수 없다. 순전히 사람이 주석을 항상 관리해 주어야 하므로 실제 코드와 일치될 가능성도 시간이 지나면 낮아진다. 주석은 절차가 아닌 이유를 설명해야 하는 것이지만, 흔히 코드의 실행 절차나 중복된 설명만 넣어두기에 코드가 달라지면 아무런 의미가 없다. 마치 소설에 등장하는 인물들이 역할에 걸맞은 이름이 있듯이 코드에 등장하는 모든 것들도 "좋은 이름"을 가져야 한다. 역할을 유추할 수 있는 이름이라면 이름 자체만으로도 충분히 코드를 읽는 사람에게 의미를 전달할 수 있기 때문이다.

# [ 숟가락만 들고 있어서는 안된다. ]

어느 회사를 가더라도 밥을 먹고자 하는 사람은 많지만, 정작 밥을 지으려고 하는 사람은 적다. 개발에 해바라기처럼 바라보는 사람들은 많지만, 정작 개발에 도움을 주는 일을 하는 사람들은 드물다. 관리하고 싶어 하지만 관리받는 사람들을 위한다고 보이지는 않으며, 개인적인 욕심이나 자신이 해야 할 일을 개발로 떠 넘기는 경우도 흔히 볼 수 있다. 밥을 짓는 사람은 한 명인데도, 쌀은 얼마나 넣어야 하는지, 물은 어느 정도 높이로 해야 하는지, 불은 어떻게 조절해야 하는지 등등 다양한 주문을 여러 사람이 훈수한다. 과제를 하면서 과제에 실질적인 도움을 주기보다 이런저런 이유로 흔들어대는 것이다. 하지만, 아무도 과제가 실패했을 경우 결과에 대해서는 책임을 지지 않는다. 결국 모든 책임은 과제 개발자들의 몫이 되어버리고 만다. 개발자들은 요구된 것을 개발하는 데 전력을 다 한다. 하지만, 요구된 것들이 자꾸만 달라진다면 그것을 만족하게 하는 것은 점점 더 어려워진다. 결과는 뻔하다. 과제의 지연밖에는 더 나올 것이 없다.

자신이 정의하지 못하는 것을 하라고 시킨다면 당연히 결과물에 대한 책임은 개발자들의 것이 아니다. 정하지 못하도록 한 다양한 근거를 제공한 사람들의 몫일 뿐이다. 물론 개발자는 우수한 품질의 소프트웨어를 만들어낼 책임을 진다. 하지만, 그것을 만들기 위해서 다양한 부문에서 개발에 도움을 주어야 한다. 무엇을 만들지를 정해 주어야 하며, 어떻게 평가할 것인가를 같이 검토해야 한다. 무엇을 어떤 일정으로 개발할지도 협의를 해야 하며, 어떤 것들이 과제에서 우선순위가 높은 것들인지도 함께 확인해야 한다. 이것을 모두 다 개발이 알아서 하라고 한다면, 나중에 바꿀 경우에 비용이 추가로 발생하거나 일정이 지연되리라는 것은 마음속에 항상 가지고 있어야 할 것이다. 하지만, 현장에서는 너무나 쉽게 변경을 개발로 전달한다. 개발 기간에 대한 것은 말할 필요도 없다. 누구의 머리에서 나온지는 모르지만, 일정은 언제나 미리 정해져서 개발로 통보된다. 개발자가 의견을 주더라도 “왜 안돼?”라는 반복된 질문만 받을 뿐이다. 개발에서는 전혀 동의한 것들이 아닌 기능들도 반드시 언제까지 개발되어야 한다고 이야기한다. 물론 그렇게 정해진 일정을 변경하는 것 자체도 불가능한 경우가 많다. 만약 개발이 이를 받아들이지 않을 때는 경영진까지 동원에서 개발에 압력을 가하는 경우도 허다하다. 무슨 기능이 언제까지 개발되지 않으면 “매출이 부러진다”는 말도 자주 듣는 표현이다.

정말 원하는 것을 얻기를 바란다면 개발의 과정에 참여해야 한다. 소프트웨어 개발과정에서 코딩하고 설계를 하라는 말이 아니다. 정확하게 개발할 것이 무엇인지를 정의하고, 개발해야 할 기능들에 대한 우선순위를 정해서, 개발이 제대로 된 방향으로 가고 있는지를 같이 보라는 말이다. 개발자와 꾸준히 대화해야 하고, 고객의 의견을 정확히 전달하는 것도 필요하다. 개발자들은 신이 아니다. 그들은 많은 불확실성에도 불구하고 소프트웨어 개발을 진행하고 있다. 그들도 모든 것을 알고 난 후에 결정하는 것이 결코 아니다. 과제를 해나가면서 이해를 높여나가고, 결과적으로 과제의 완료에 가까워질수록 더 많은 지식을 가지게 된다. 과제에 참여하는 것은 과제가 더 잘되도록 정확한 입력을 주기 위한 것이며, 과제가 제대로 되는지를 같이 확인하는 것이다. 필요없는 부분이 구현되고 있다면 당연히 잘못된 것이다. 모든 있으면 좋을 기능들을 조합해서 요구한다면, 아무것도 구현하지 못하거나 최소한만 구현할 것이다. 결과적으로 이런 식으로 구현된 기능들은 사용자를 만족시킬 수도 없을 것이다. 개발된 기능들이 원하는 것이 맞는지도 확인해야 하며, 그것이 정확히 고객이 원하는 것인지 지속해서 확인받아야 한다. 이런 과정을 통해서 얻은 정보는 다시 개발에 피드백(Feedback)되어, 과제의 추정에 대한 정확도를 점차 높여 나가야 할 것이다.

지속적인 피드백이 중요하다면 기능별로 구현되어야 한다. 계층적이고 모듈위주의 개발이 수평적인 개발이라고 한다면, 기능적인 개발은 수직적인 개발이라고 볼 수 있다. 사용자들이 사용할 수 있는 기능들을 빨리 보여주기 위해서는 수직적인 개발이 반드시 필요하다. 수직적인 개발에서 중요한 것은 시스템의 핵심이 되는 근간에 대한 개발이 선행되어야 하고, 나머지 곁가지 기능들은 나중에 우선순위에 따라 개발되어야 한다. 따라서, 개발을 돕고자 한다면 정말 중요한 기능이 무엇인지를 식별하고, 그렇지 않은 기능들은 가능한 추가하지 않아야 할 것이다. 코드가 커질수록 위험도가 높아지기에 불필요한 기능은 가능한 개발을 하지 않는 것이 가장 좋다. 보고서는 될 수 있으면 줄이는 것이 좋다. 필요한 회의만 해야 한다. 일일 보고, 주간 보고, 월간 보고, 분기 보고 등등 각종 보고서 및 숙제를 한다고 사용되는 시간은 생각보다 크다. 심지어 진행 중인 과제를 다 마치지도 못했지만 새로운 일을 기획한다고 개발자들을 동원하는 경우도 있다. 이런 것들을 원하는 사람의 입장에서는 필요하기에 시키겠지만, 과제의 지연에는 아무런 책임감도 느끼지 않을 것이다. 개발자들의 시간을 존중하지 않으면서 어떻게 자신이 원하는 것을 받아내려고 하는 것일까? 결국 소프트웨어 과제는 제대로 동작하는 결과물을 만들어야 완료된다. 문서나 보고서, 회의를 아무리 많이 한다고 동작하는 소프트웨어가 만들어지는 것은 아니다. 손을 벌리기에 앞서 도움을 주는 손길을 건네는 것이 항상 먼저라는 사실을 잊어서는 안될 것이다.

# [ 회사가 팔리다. ]

매출과 이익이 늘어나지 않는 회사는 성장하지 못하고 제자리 걸음만 하는 것이다. 매출과 이익은 물건이 팔려야 늘어난다. 그리고, 그에 따라 사람의 충원과 자금의 흐름이 원활해질 수 있다. 매출만 늘린다고 해서 이익이 같이 늘어나지는 않는다. 하지만, 성장의 측면에서는 어쨌든 매출이 늘어나야만 규모의 경제를 이룰 수 있다. 제품이 시장에서 성공적이라면 매출과 이익이 같이 늘어나겠지만, 싼 물건으로 매출만 늘리려고 한다면 어느 순간 한계에 도달할 수밖에 없다. 따라서, 회사의 꾸준한 성장을 위해서는 시장을 만족시킬 수 있는 제품을 지속해서 공급해야 한다. 하지만, 단기간의 임기 보장만 받고 일하는 임원들의 생각은 장기적인 시장에서의 성공을 바라보지 못한다. "임원"은 “임시 직원”일 수밖에 없다는 생각에 단기적인 성과에만 집중하는 것이 그들에게는 현실인 것이다.

플랫폼(Platform)과 같은 전략이 성공하기 위해서는 단기적인 성과보다는 장기적인 성장에 초점을 두어야 한다. 따라서, 새로운 플랫폼을 개발하는 것은 회사 차원에서의 지원을 해야 하는 일이 될 수밖에 없다. 그렇지 않다면 플랫폼은 갈 방향을 쉽게 잃고 만다. 임시 직원들은 이렇게 이야기한다. "플랫폼을 증명하지 못하면 안 쓰겠어." 무엇을 증명하라는 것인지는 잘 모르지만, 개인적인 생각에는 "비록 어렵겠지만 이번에는 반드시 플랫폼을 사용해보도록 하자."라고 말하는 것이 옳다고 생각한다. 이것이 권한과 책임을 가진 임원들이 가져야 할 열린 태도가 아닐까? 하지만, 임시직인 분들은 플랫폼보다는 당장 사용할 수 있는 것들로 제품을 빨리 만들어서 시장에 내놓기를 원한다. 자신의 임기 동안에 발생하는 매출이 항상 문제가 되기 때문이다. 그들에게는 장기적인 비전(Vision)은 남의 일이다. 그들의 눈앞에는 당장 연말에 있을 재계약만이 중요할 뿐이다. 시월 말까지 의미 있는 무언가를 만들어내지 못하면 내년에도 자신의 자리가 있을지 장담하지 못하기 때문이다.

대부분 소프트웨어를 개발하는 회사는 소위 말하는 "기존 코드(Legacy)"를 가지고 있다. 그리고, 그런 코드가 시장에서 팔리는 제품에 들어가 있으며 스파게티처럼 얽힌 상태인 경우가 많다. 단위 테스트(Unit Test)나 회귀 테스트(Regression Test)는 자동으로 절대 하지 못하고, 아무리 자동화시키려고 노력해도 통합된 코드를 외부에서 입력을 주고 출력을 확인하는 형태의 시스템 테스트 정도만 자동화하는 수준이다. 새로운 기능을 추가하면 다양한 버그들이 새롭게 검출되고, 조그만 수정에도 금방 취약점을 노출하고 만다. 컴파일러의 경고도 제대로 처리하지 않은 경우도 많고, 정적 분석 툴과 같은 것을 사용해서 버그 유발 가능성이 있는 코드들을 찾아내면 상당히 많은 사소한 오류(Defect)들이 있음을 볼 수 있을 것이다. 생각보다 우리가 사고 있는 제품에는 많은 버그가 내장되어 있을 가능성이 높다. 대기업이 좀 더 잘하는 것은 운용 가능한 자원들이 많다는 점이다. 즉, 그러한 자원들을 활용해서 빨리 버그를 수정하고, 자동화 테스트를 조금씩 늘릴 수 있으며, 시장에서 보고되는 문제점들을 상품개발 조직과 분리해서 처리하는 조직을 새로 만들 수 있다는 점이다.

하지만, 대기업도 자원의 효율을 따지면 그렇게 높은 성과를 거두고 있는 것이 아닐수도 있다. 조직이 제대로 된 코드를 가지고 있는 경우가 드물기 때문이다. 오히려 몇 명 안 되는 개발자들이 제대로 된 코드를 만들어 지속해서 제품화를 시키고 있는 중소기업보다 효율이 낮을수도 있다는 뜻이다. 인당 매출을 비교해 보면 상대적으로 대기업이 효율적이지 못할 수도 있기 때문이다. 제대로 된 코드는 테스트와 디버그 시에 필요한 노력(Effort)이 상대적으로 크게 적어야 한다. 즉, 코드를 짤 때 테스트를 지속해서 자동화시켜 왔다면, 버그당 수정 비용이 극히 적을 것이 분명하기 때문이다. 한 번에다 개발된 모듈들을 통합해서 테스트하는 경우라면 테스트 자동화는 생각할 수 없으며, 발생한 버그를 찾기 위해서는 지속해서 범위를 줄여가며 사람이 일일이 분석해야 할 것이기 때문이다. 좋은 툴(Tool)을 사용할 수 있는 것도 대기업에 유리한 점이지만, 좋은 툴이 우수한 소프트웨어를 보장하는 것은 결코 아니다. 단지 버그 수정에 들어가는 노력을 조금 줄여줄 수 있을 뿐, 버그가 없는 코드를 만들어내는 것은 아니기 때문이다.

회사는 결국 누군가의 실패로 인해서 다른 회사에 팔리게 된다. 관리를 맡은 "임시직원"들의 실패가 원인일 수도 있고, 중간 관리를 맡은 팀장들의 소프트웨어 개발에 대한 무지도 한 몫 했을 수도 있으며, 시장을 잘 못 분석하고 팔리지 못하는 제품을 터무니없는 일정으로 만들어 달라고 주문한 상품기획의 문제일 수도 있다. 마케팅도 자신의 실적에만 매달린 영업으로 개발에 부담을 가중시켰을 수도 있다. 개발자들이 검증되지 않은 코드를 통합해서 버그를 찾기 어렵게 만든 것도 신제품의 출시를 늦춘 결과로 이어졌을 수도 있다. 큰 실패는 작은 실패의 반복에서 발생하며, 동일한 실패를 2번 이상 하는 것은 실력이라고 박에 볼 수 없다. 실력이 없는 회사는 선택받지 못하는 것이 시장이다. 지갑 속의 내 돈을 주고 물건을 산다면 당연히 더 안정적이고 원하는 기능을 만족하게 하는 제품을 살 수밖에 없는 것이 소비자의 심리다. 회사는 그런 고객의 요구를 만족시키는 제품을 만들지 못했기 때문에 실패한 것이다. 누군가의 책임을 묻기에 앞서 회사 내의 모든 사람은 각자의 역할에서 최선을 다했는지를 먼저 물어야 할 것이다. 최선이란 "열심히 하는 것"이 아니라 "바른길로 가는 것"임을 장기적인 성장을 바라보는 관점에서 찾아야 한다. 단지 늦게까지 야근한다고 해서 모든 것이 잘 될 것이라는 막연한 해결책으로는 "바른길"을 결코 걸어가지 못한다.

# [ 통제 혹은 관리? ]

관리자들도 처음부터 관리자였던 것은 아니다. 그들도 신입사원. 대리, 과장, 부장 등을 거쳐서 그 자리에까지 오른 사람들이다. 그런 자리에 오르기까지 수많은 것을 경험했고, 다양한 과제도 해봤을 것이 분명하다. 이미 실무 개발자들보다 높은 곳에서 거시적인 정보들을 가지고 과제들을 하나씩 보고 있을 것이 분명하다. 과제의 돌아가는 자세한 상황은 잘 모르지만 정기적으로 보고는 받고 있을 것이다. 그들은 많은 회의에 참석하며 다양한 과제들로부터 많은 문제점을 보고 받는 것도 사실이다. 자신이 한 말을 모두 다 기억하는 것도 힘들 정도로 이런 저런 자리에서 많은 의견을 주고받을 것이다. 경험이 많고 지식이 많다고 스스로 생각하는 관리자들은 모든 것을 혼자서 독단적으로 처리하려고 할 때 문제를 일으킨다. 소위 말하는 "마이크로 관리(Micro Management)"를 지속해서 하기에 스트레스만 늘리게 된다. 모든 결정 권한을 자신만이 가지고 있다고 보기에 남에게 권한을 이양하는 것 자체를 탐탁치않게 생각하기 때문이다.

권한이 많을수록 좋다고 생각한다면 남에게 권한을 절대 넘겨주지 않는다. 모든 것을 자신이 알아야만 다른 사람이 올린 결재에 도장을 찍는다. 아무리 사소한 것이라고 해도 반드시 그런 것들을 챙겨야 직성이 풀린다. 사람에 대한 신뢰를 명령을 얼마나 충실하게 수행하는가에 따라 나누고, 이런 저런 안되는 이유나 어려움에 대한 설명은 듣지 않으려고 한다. 관리자가 경험이 많은 것은 사실일수도 있지만, 그렇다고 모든 분야의 전문가인 것은 결코 아니다. 세상의 지식은 세분화되어 있고 갈수록 전문화되고 있는 상황에서, 모든 것을 독단적으로 결정한다는 것이 얼마나 위험한 일인지를 깨닫지 못한다. 하드웨어와 관련된 일을 하는 사람이 소프트웨어에 대한 결정을 내린다면 그것이 과연 올바를까? 아마도 결정의 결과로 오는 모든 것들은 자신의 명령을 제대로 수행하지 않은 사람의 책임으로 돌려질 것이다. 명령은 하지만 스스로 책임은 안 지는 것이 그런 사람들의 공통된 특징이다. 모든 권한은 다 가지면서도 권한에 따르는 책임은 다른 사람이 지도록 만드는 교묘한 화술을 과거의 경험으로부터 충분히 익혔기 때문이다. 물론 이런 관리자들만 있는 것은 아니지만, 그런 관리자를 만나는 것이 드물지 않다는 점은 명확하다.

관리자는 자신이 관리하는 조직을 성장시킬 책임을 진 사람이다. 성장한다는 말은 일의 성과가 나야 한다는 것이다. 성공적인 일을 더 많이 할수록 조직은 확장될 것이고, 그렇게 함으로써 새로운 인력에 자리가 생기게 된다. 결국 조직이 성장한다는 것은 기존의 인력이 승진한다는 말이며, 새로운 인력으로 충원이 꾸준히 들어온다는 뜻이다. 하지만, 단순히 일을 많이 하기에 필요한 인력을 충원하는 것으로는 "성장"이라는 말을 쓰기는 어렵다. 성장이란 결국 구성원 개개인의 역량이 커가는 것에 더 잘 어울린다. 하지만, 이런 것들을 쉽게 용납하는 조직이 있을까? 예를 들어, 한창 과제의 막바지라 바쁜 와중에 일주일간 사외교육을 간다고 한다면 용납할 수 있을까? 혹은 팀 내 세미나를 그런 와중에 한다고 하면 쉽게 받아들이는 팀장이 있을까? 사실 이런 도전은 항상 있는 일이고 용납하지 않는다면 달라지는 것은 없다. 하지만, 따지고 든다면 과제의 계획을 세울 때 구성원의 교육 등에 대한 필요한 시간을 확보하지 못한 관리자의 책임이 더 크다. 하지만, 그도 해결할 방법이 딱히 없는 것도 사실이다.

문제는 이런 관리자들은 일을 너무 열심히 한다는 점이다. 마치 자기가 없으면 회사가 돌아가지 않을 것처럼 생각한다. 남들에게 권한을 주면 왠지 믿지 못하는 결과가 나올 것이라는 선입견에 매달리기도 한다. “사소한 관리”가 좋은 관리라고 배웠을 수도 있다. 아마 관리자 자신도 원래 그렇지는 않았겠지만 지금의 자신과 비슷한 사람 밑에서 일을 배웠을지도 모른다. 개인의 성향이라고 볼 수도 있지만 직장 내에서의 관리 스타일은 윗사람에게서 물려받는 경우가 많다. 하루 하루 마주치는 자리에서 알게 모르게 조금씩 젖어 들어갔었을 것이고, 이제는 자신과 구별되지 않을 정도로 동일시되었을 것이다. 이런 관리자들은 일을 많이 할수록 일이 더디게 만든다. 왜냐하면 의사 결정 권한이 한 사람에게 집중되어 "병목(Bottleneck)"현상이 발생하는 지점이 되기 때문이다. 사람들은 점차 그 관리자에게 보고하기 위해서 "대기열(Queue)"에서 기다리는 시간이 늘어날 것이고, 결정되지 않은 것들은 절대 실행되지도 않을 것이다. 모든 결정을 그 사람에게 물어보고 할 것이고, 스스로는 책임을 질 일을 만들지 않을 것이 분명하다. 능동적이고 주체적인 생각은 버리게 될 것이며, 뒤돌아서면 책임감도 안 가질 것이다. 그냥 의무적으로 해야 할 일을 그때 그때 욕먹지 않을 정도로만 하면 된다고 생각할지도 모른다.

통제하고 싶다면 다른 것들을 하라. 개발자들이 일하는 데 방해가 되는 애로사항들을 통제하라. 그것들을 해결하는데 모든 시간을 쏟아붓고 개발자들이 말하는 것을 귀담아 들어야 한다. 그들은 목적을 가지고 의식적인 행동을 하는 사람들이기에 당신의 경험만 가지고 판단하려고 해서는 안 된다. 이미 당신이 한 경험의 유통기간은 지났다. 더구나 체계화되지 않은 경험은 약보다는 독에 가깝다. 자신의 말만 잘 듣는 사람들만 옆에 두고 싶다면, 그냥 강아지나 많이 사서 데리고 다녀라. 살랑살랑 꼬리를 흔들며 주는 먹이만 잘 받아먹고, 끈으로 묶어둬서 어디 다른 곳으로 가지도 못하는 애완견이 당신에게는 더 어울릴 것이다. 그런 사람이 되고 싶다면 말릴 생각은 없다. 통제와 관리는 다른 것이다. 통제는 행동의 범위를 설정하고 넘어갔을 경우 벌을 준다는 것이며, 관리는 목적을 가지고 방향성을 제시한다는 것이다. 통제를 관리라고 생각하는 관리자들은 가축의 우리를 지키는 농장 주인과 같다. 언제고 농장에서 늙고 병든 것들이 있다면 솎아낼 것이 분명하다. 고분고분 잘 따르는 사람들도 한순간에 내쳐짐을 당하는 것이 우연은 아닐 것이다. 당신은 농장 주인이 되고 싶은가, 아니면 당신들의 "동료"들과 함께 고난의 바다를 건너는 "선장"이고 싶은가? 언제든 선택은 당신에게 있다. 하지만, 주어진 시간은 그리 길지 않을 것이다.

# [ 프로(Professional) 프로그래머 ]

프로가 뭘까? 진정한 프로의식이란 뭘까? 혹시 우린 아마추어적인 생각에 물들어 사는 것은 아닐까? 아마추어란 목표보다는 상황에 좌지우지되고, 앞으로 나가기를 주저하며 하지 못하는 이유만 찾는다. 상황은 항상 어떤 것도 쉽도록 내버려 두지 않는다. 도달해야 할 목표는 항상 높이 잡혀있고, 이런저런 이유로 우린 앞으로 나아가기보다는 기존에 해왔던 익숙한 것들에 기대고 만다. 프로란 "그럼에도 불구하고"를 몸으로 보여주는 사람들이다. 그럼에도 불구하고 해야 할 일을 제 때에 하는 사람들이다. 그들은 "제 때"에 대한 감각이 있으며, 그 순간을 놓치면 돌이킬 수 없는 결과가 발생한다는 것을 잘알고 있다. 그리고, 그 한순간을 위해서 모든 것을 인내하며 내부 역량을 키우는 사람들이다. 모두가 가려진 현실에서 "진정으로 원하는 것"을 때를 놓치지 않고 찾아낸다. 그래서, 프로들은 아마추어가 보기에는 엉뚱하리만치 힘든 시간 속을 거친다고 보이지만, 그들 나름의 목표는 뚜렷하다. "절대 필요한 것은 반드시 제 때 하고야 만다."

그렇다고 이 글을 쓰는 사람이 "그런 프로"라고 생각하지는 않기를 바란다. 저자 또한 그런 프로가 되기를 꿈꾸지만 아직 "그 때"를 찾아내는 "감각"과 "절대 필요한 것을 하고야 마는 인내"를 가지지 못했다. 결국, 진실은 "단순함"에 있다. "단순히 자신이 해야 할 일을 해야 할 때에 하는 것", 그것이 정도(正道)다. 힘들지는 모르지만 바른길을 가는 것이 중요하다. 급박한 일정, 모자란 인력, 어려운 기술, 이런 저런 휘둘림, 그런 것들이 없으면 오히려 너무 쉬운 일일 것이다. 정도는 "그럼에도 불구하고 해야 할 일"을 절대 잊지 않는 것이다. 노력의 결과는 반드시 빛을 볼 것이다. 만약 누군가 "어리석다"라고 비난하거나 지름길처럼 보이는 유혹을 제시하더라도 손을 흙탕물에 넣는 것을 마다하지 않아야 한다. 프로는 진흙창 속에서도 가치를 찾아내고, 목표를 이룰 수 있는 역량이 있는 사람들이다. 흰 와이셔츠를 입고 멋진 넥타이를 맨다고 프로가 되는 것은 아니다. 노련한 관리자는 어려움에 부딪쳤을 때 언제나 프로라고 불리는 사람들을 부르게 된다. 그들은 어떻게 하면 그런 상황들을 파악하고 순리에 맞게 일을 풀어가는지를 아는 사람들이다.

회사에는 많은 아마추어가 있다. 그들은 나름의 처세술을 익힌 사람들이라 섣불리 바뀌지 않는다. 그들은 나름의 방법론을 가지고 있지만, 문제를 해결하기보다는 회피하는 해결방법을 먼저 찾는다. 개선보다는 잊음을 선택하지만 같은 문제를 반복해서 경험한다. 하지만, 안타깝게도 자신의 힘으로 그런 인생의 고단한 반복 속에서 절대 헤어나오지 못한다. 그들은 문제를 찾아내는 데는 익숙하지만, 문제를 해결하는 제대로된 방법은 모른다. 그들은 시간이 없음을 탓하지만, 시간이 주어지면 엉뚱한 일을 한다. 그들은 사람을 많이 원하지만 제대로 된 사람을 키우지는 못한다. 그들은 많은 자원을 주면 일을 벌일 생각을 하지, 기존에 했던 일을 정리할 생각은 하지 않는다. 그들이 아마추어인 이유는 "그럼에도 불구하고 개선하자"는 것이 아니라, "그럼에도 불구하고 그냥 하자"는 것이다. 그들이 정말 원하는 것이 무엇인지 하루 하루의 일상 속에서 찾지 못하고, 그 하루만 생각하고 사는 사람들이다. 그들의 생각속에는 자신만이 있으며, 그 외의 사람들은 그들을 위해 존재하는 사람들로 생각한다. 사람이 정말 중요한 핵심이라는 생각은 꿈에도 하지 않는다.

잘못되었다고 인식했을 때 자신의 잘못을 인정하는 것이 프로다. 프로는 자신을 끊임없이 개선하려고 노력하는 사람들이기에 잘못된 점을 발견하면 "기본"으로 돌아가려고 한다. 프로와 아마추어의 가장 큰 차이는 자신의 잘못을 인식하고 개선하려는 노력을 하는가에 달려있다. 기본이란 주춧돌과도 같아서 잘못된 기반 위에는 웅장한 건물을 지을 수는 없다. 사상누각처럼 지어진 건물들은 조금만 흔들려도 쉽게 무너지기 마련이다. 기본이란 용어 하나 하나의 정의에서 시작하며, 그런 용어들이 모여서 문장을 이루고 단락이 되어 결국 전체 글이 완성된다. 프로들은 자신이 사용하는 용어 하나 하나에도 세심한 주의를 기울이며, 정확한 지식을 듣는 사람의 귀에 맞게 전달하는 능력을 갖춘 사람들이다. 너무 어려운 부분들은 비유를 들어 이해하기 쉽게 만들지만, 그렇다고 놓치는 부분이 있는 것은 결코 아니다. 상황에 맞는 적절한 선택과 그 선택으로 인해서 주어지는 결과를 받아들인다. 그리고, 다음의 선택에서는 이전의 선택으로 알게된 부분들을 고려해서 더 나은 선택을 유도한다. 아마추어들은 자신의 선택이 최고라도 내세우지만 프로들은 자신의 선택이 최선이었다고 이야기한다. 그들은 자신의 한계를 인식하고 있으며, 그 한계를 극복하기 위해서 하루 하루 가설과 실험을 꾸준히 반복한다.

# [ 존중, 팀의 시작 ]

소프트웨어 개발자에게 최대한의 성과를 끌어내기 위해서는 그들 스스로가 문제를 해결할 수 있는 능력이 있다는 것을 인정해야 한다. 그들의 역량이 부족하다면 역량을 키울 수 있는 능력이 있다고는 것을 인정해야 한다. 역량을 키우는 것조차도 의심스럽다면 아마도 과제를 그들에게 잘 못 준 것일지도 모른다. 어쨌든 자신의 팀원들에 대한 믿음이 시작이다. 팀워크는 믿음으로 다져야 오래간다. 하지만, 이런 팀워크를 망치는 것도 순간이다. 팀으로 녹아들지 못하고 관리와 명령만이 자신의 업무라고 생각한다면, 팀을 감시하는 역할이지 절대 팀원이 될 수 없다. 팀은 스스로 결정하고 책임도 함께 나누는 것이지, 누군가의 일방적인 지시와 명령으로 유지되지 않는다. 만약 그런 상황에서 일한다면 그냥 울타리 속의 가축이라고 인정해야 할지도 모른다.

믿음은 상대에 대한 존중에서 시작된다. 같이 일하는 사람들을 존중하지 않는다면 도대체 누구를 존중할 수 있겠는가? 인정받고 있다는 것을 안다면 사람들은 최선을 다하려고 노력한다. 비록 그 결과물이 그 사람의 역량 수준에서 최선일지는 모르지만, 그래도 자신을 더 높은 곳으로 올려 놓으려는 끝없는 개선을 할 것이다. 하지만 실무에서는 그렇지 못한 경우를 많이 본다. 팀이라고 이야기는 하지만 일관된 무시와 질책만이 난무한다. 슬픈 현실이라고 자책하면서도 자신의 "동굴"로 숨어버리고 움츠러든다. 자신이 가진 능력의 최소한만을 보여주고 될 수 있으면 문제를 일으키고 싶지 않아 한다. 문제가 없으면 개선도 없다. 문제가 없는 조직이 좋은 조직일까? 그렇지 않다. 문제의 수준이 차이가 나겠지만 그것을 인식하고 개선하려고 노력하는 것이 일반적인 태도다. 문제가 없이 보이는데도 뒤처진다고 느낀다면 이미 많은 곳에서 타성에 젖어서 행동하는 사람들을 쉽게 발견할 수 있을 것이다.

존중받지 못하는 사람들은 다른 사람을 존중하지도 못한다. 특히 자신이 존중받지 못한다고 생각되는 관리자를 만난다면, 그런 사람의 명령에는 별다른 의미를 담아두지 않는다. 그냥 듣고 최소한의 요구만 만족시키게 행동할 것이다. 그런 관리자 밑에서 일하는 사람들의 실력도 점차 줄어들 것은 분명하다. 일에 대한 열정이 없는 사람에게 일의 깊이를 느끼는 말을 들어본 적이 있는가? 그냥 시키니까 하는 것일 뿐이다. 책임이 있는 자리에 올라도 타성에 젖기는 마찬가지다. 하루 하루 문제점들을 나열할 뿐 근본적인 해결책은 찾으려고 노력하지 않는다. 근본적인 해결책은 어디에 있는지 알고 있을지도 모르겠지만 그것을 찾으려는 노력 자체를 하지 않게 되는 것이다. 왜? 잘해도 그만 못해도 그만이기 때문이다. 하지만, 어떤 사람은 그런 윗사람을 배우려고 드는 경우도 있다. 소위 말하는 "존경"이라는 단어를 쓰면서 다른 사람들 앞에서도 거리낌 없이 이야기한다. 미안하지만 만약 당신이 그런 사람이라면, 당신은 존경할 대상을 잘 못 찾은 것이다. 당신이 그 사람으로부터 배운 것들이 조직의 힘을 얼마나 약화시켰는지 모르겠지만, 시간이 흐르면 결과는 나타나기 마련이다. 그 결과는 일을 아무리 많이 하고 열심히 해도 계속 지연되는 경험하는 것이다.

책임감이 없는 조직은 일정에 대해서 아무런 제약을 받지 않는다. 그들이 만족시켜야 할 대상이 그들을 존중하지 않는데, 기대를 만족시키고 싶은 필요를 느낄까? 아마도 당장 눈 앞에 펼쳐지는 상황만 모면하려고 할 것이다. 당신이 모든 세부적인 것을 다 알고 시간에 맞춰서 적절한 명령을 수시로 내리고 확인하지 않는다면, 과제는 당신의 손에서 벗어날 것이 분명하다. 당신이 아무리 자세한 계획을 세우더라도 과제에서는 언제나 예상하지 못한 상황이 발생한다. 모든 것을 다 상세하게 관리하기를 원한다면, 그 틀 안에 사람은 넣지 말아라. 사람은 관리하는 대상이 아니라 양성의 대상이어야 한다. 양성하는 것이 관리라고 생각하기도 하겠지만, 양성은 개선해야 할 사항에 대한 피드백이다. 그 사람이 성장하는 데 도움을 주는 것이지 지시만 내린다고 되는 것이 아니다. 그 사람의 역량이 향상되는 것을 지켜보는 것이지 현재의 모자람을 비판하는 것이 아니다. 팀은 과제를 해결하면서 성장해야 한다. 동일한 실패를 반복하면서도 아무것도 배우지 못한다면, 안타깝지만 당신은 자신의 책임을 다하지 못하고 있는 것이다. 팀원들에게 성장할 기회를 제공하지 못하고 있으며, 그들의 생각에 제약을 가하고, 그들이 더 뛰어난 사람이 될 가능성을 당신이라는 “울타리" 속에 가두고 있기 때문이다. 이제는 "사람"을 사람답게 바라봐야 할 때다. 적어도 당신 또한 존중받는 존재가 되기 위해서는 남들도 존중할 수 있어야 한다.

# [ 대화가 중요하다. ]

소프트웨어 개발은 대화로 시작해서 대화로 끝난다. 무엇을 개발할지를 정하는 대화에서, 어떻게 설계할 것인지를 정하는 대화, 정해진 내용을 개발에 제대로 전달하는 대화가 필요하다. 개발된 내용이 제대로 되었는지를 확인하는 것도 무엇을 만들지를 정하는 것에서 출발한다. 그리고, 이렇게 만들어진 제품을 사용자가 직접 사용하면서 제대로 개발되지 못한 부분에 대한 피드백을 대화로 주고받는다. 개발하는 동안에는 개발자 간에, 혹은 개발팀 간에 대화가 유지가 되어야 하며, 이런 대화는 특정 시점을 두고 하는 것보다는 자주 활발하게 하는 것이 중요하다. 관리자는 개발자에게 무엇을 해야 할지를 알려주는 대화를 하겠다고 생각하지만, 사실은 개발자가 자신이 할 일을 정한다. 따라서, 관리자는 개발자가 무엇을 개발할지를 대화를 통해서 알 필요가 있는 것이지, 일방적인 지시만 내리는 사람이 아니다. 사실 무엇을 만들지를 제대로 다 정의하는 것이 가장 힘든 부분일 것이다. 고객도 만들어진 제품을 보기 전에는 자신이 무엇을 원하는지 모르기 때문이다.

시장에 나온 제품들을 보도록 하자. 생각보다 비슷한 기능을 하는 많은 종류의 제품들이 고객의 선택을 기다리고 있다는 것을 알 수 있을 것이다. 제품 자체가 고객의 “요구(Needs)”를 완벽하게 구현하지 못하기에 선택을 기다리는 것이다. 모든 제품이 똑같은 요구를 만족하게 하는 것이 아니라는 점에서 고객의 요구를 파악하는 수준도 다르다. 따라서, 필요한 것은 역시 고객과의 대화다. 하지만, 우리는 이 부분을 제대로 하는 경우를 극히 드물게 보며, 직접적으로 고객과 접촉할 수 있는 기회도 적기에 다른 대안을 찾는다. 상품기획이 아마도 고객의 요구를 가장 잘 파악하는 조직일 것이다. 따라서, 최소한 소프트웨어 개발자라면 항상 상품기획이 무엇을 원하는지를 아는 것이 중요하다. 물론 상품기획도 고객의 모든 요구룰 다 아는 것은 아니지만, 그래도 개발자보다는 나은 수준일 것이 분명하다. 상품기획 또한 개발에 중요한 요구를 전달하는 데 있어서 모든 것을 요구해서는 안 된다. “가치”가 있는 요구를 우선순위로 전달하는 것을 잊어서는 안 된다. 될 수 있으면 정확한 요구의 이해를 위해서 직접 만나서 이야기하는 것이 가장 좋다. 기록을 위해서 문서로 남기는 것도 중요한 활동이다. 우선순위가 없는 요구는 언제 만족되어야 하는지 알 수 없기에 당분간 구현할 필요도 없다.

팀 간의 대화가 중요한 이유는 서로 가지고 있는 기술이 공통점을 가지거나, 문제 해결에 있어서 유사한 경우가 많기 때문이다. 팀 간의 대화가 단절되는 이유는 경쟁이 심할 경우다. 서로 동일한 제품을 대상으로 개발을 진행할 경우에는 대화의 단절이 발생한다. 하는 일이 다른 팀 간의 경쟁은 오해에서 발생하거나 잘못된 역할 인식에 기인할 수 있다. 테스트팀과 개발팀의 관계에서 흔히 보이는 이러한 태도는 협업으로 인해 더 좋은 제품을 개발한다는 목표에 어긋나는 행동이다. 일방적으로 지적하고 일방적으로 방어를 하는 행동을 보이는 이유는 서로 간에 장벽이 있다는 것을 의미한다. 진정한 협업은 역할을 충실히 하는 것뿐이다. 그리고, 공개적으로 모든 일을 처리해야 역할을 제대로 하는지 관련된 모두가 알 수 있다. 서로 간에 “짜고 치는 고스톱”쯤으로 일을 처리한다면, 결국 그 제품을 사는 소비자만 피해를 볼 뿐이다. 물론 그렇게 피해를 본 소비자가 우리 제품을 다시 선택할 가능성은 없다. 이렇게 돌고 도는 피해는 자신의 주머니 속에 있는 지갑에 돈이 들어오는 것을 방해할 뿐, 제품의 품질에는 아무런 도움이 되지 않는다. 팀이 경쟁하는 상황은 자원의 낭비라고 보는 경우가 있지만 그렇지 않다. 우리는 불확실성을 다루는 일을 하기에 어떤 것이 더 좋을지 판단이 서지 않는 경우가 많다. 그런 경우를 위한 위험(Risk)관리 차원으로도 볼 수 있다. 결국 서로에게 이익이 되는 행동이라는 것을 이해해야 한다.

관리자와 개발팀의 대화는 아무래도 가장 중요하리라 생각한다. 일반적으로 관리자는 개발팀 내부에서 시장의 요구사항을 대변한다고 생각한다. 더 많은 정보를 가지고 있다고 생각되며, 무리한 일정이라는 것을 알면서도 개발에 일정을 지킬 것을 요구한다. 시장의 요구사항 자체 보다도 그들의 요구가 개발에서는 더 높은 우선순위를 차지하는 것은 어쩔 수 없다. 그들이 “고과”라는 관리 도구를 들이대면 거부할 수 없는 힘을 느끼기 때문이다. 결국 “Yes”라고 말할 수밖에 없다. “No”라고 이야기한다면 다음 해의 연봉 계약이 어떻게 될지 알 수 없는 상황이 될 것이다. 하지만, 우리에게는 “No”라고 말할 수 있는 용기가 필요하다. “그럼에도 불구하고” “No”라고 이야기해야만 대화가 성립하기 때문이다. 일방적인 지시는 그냥 듣기만 할 뿐이다. “No”라고 이야기하는 순간 “왜”라는 질문을 받을 것이고, 이야기를 풀어나가기 위한 과정을 시작할 수 있다. 최소한 관리자들에게 과제를 진행하기 위해서 무엇이 정말 어려운 점인지 인식시킬 수 있는 중요한 기회이기 때문이다. 관리자는 기술적으로 뛰어난 사람들이 아니다. 그들이 알고 싶어하는 것은 숫자에 관련된 것이며, 그것은 기술적인 스펙과는 거리가 먼 “돈”에 관련된 것이거나 “날짜”에 대한 것이다. 그것을 모를 때 그들은 불안해하고, “더 빨리”, “더 많이”, “더 싸게”를 요구한다. 그들이 아는 언어로 그들이 원하는 것을 가르쳐주지 않는다면 대화는 성립하지 않을 것이다.

우리는 대화의 대상에 대해서 공부를 해야 한다. 그들이 무엇을 알기를 원하는지 파악해야 대화를 제대로 할 수 있다. 개발자와 대화를 하기 위해서는 개발자의 언어를, 관리자를 위해서는 관리자의 언어를, 고객에게는 고객이 이해하는 언어를 사용해야 한다. 기술적인 것들이 필요한 경우라면 다양한 전문 용어로 이야기해도 좋지만, 그들이 원하는 진정한 것을 표현할 수 있는 그들의 언어로 대화하는 연습을 해야 한다. 대부분의 오해는 대화하지 않아서 쌓이는 것이며, 제대로 된 언어를 사용하지 않아서 발생한다. 만약에 자신이 관리자라고 할 경우에는 좀 더 신중하게 생각해야 한다. 말 한마디의 해석이 중요하기 때문이다. 관리자의 한마디는 전체 팀을 최대의 역량을 발휘할 수 있게도 만들지만, 수동적으로 지시만 이행하는 조직으로도 만들 수 있기 때문이다. 현실에서는 좋은 관리자를 만난다는 것이 흔한 일이 아니다. 높은 곳에 올라서기 위해서 그들이 포기한 가치 중의 하나가 “존중(Respect)”과 “신뢰(Trust)”인 경우가 많기 때문이다. 존중하지 않아야 말을 잘 듣고 신뢰하지 않아야 실패하지 않는다고 믿기 때문이다. 그들에게는 “책임(Ownership)”이라는 단어가 최고로 여겨지는 것은 그만큼 제대로 책임을 지지 않는다는 역설적인 사실과도 연관이 깊다. 실패한 과제에 대한 책임은 해당 과제를 실행한 조직의 것이지 자신만의 것이 아니다. 물론 성공한 과제의 경우에는 “책임감” 있게 일했다고 생각할 것이다. 하지만, 존중과 신뢰가 없는 조직은 최고의 성과를 내는 조직과는 비교할 수준이 못 된다. 결국, 시간이 지나면 자신의 말에 대한 책임을 자신이 져야 한다는 사실을 깨닫게 될 것이다.

# [ 지식 노동자? ]

소프트웨어 개발자는 일명 “지식 노동자”라고 불린다. 이 말은 일반적인 노동자와 달리 봐달라는 표현처럼 들린다. 하지만 노동의 본질이라는 측면에서 본다면 일을 해야지만 먹을 것을 얻는다는 점에서는 달라질 것이 없다. 다만 차이가 나는 부분은 양이 아닌 질이 중요하다는 점일 뿐이다. 하지만, 그 부분도 조금만 생각해보면 그렇게 큰 차이는 아니다. 누구도 형편없는 품질의 제품을 만들고 싶어하지 않기 때문이다. 따라서, 소프트웨어 개발만 특별히 지식 노동자라고 부를 이유는 없다고 생각한다. 우리는 특권을 가진 사람들이 아니며, 우리의 일도 다른 일과 마찬가지로 사람들을 좀 더 편하게 살 수 있도록 만드는 것일 뿐이다. 남들보다 조금 더 지식이 집약된 분야의 일을 한다고 지식 노동자라고 불리고 싶다면 그래도 상관없다. 하지만, 정말 우리가 하는 일이 정말 지식의 축적과 활용을 제대로 하고 있는지는 스스로에게 물어보아야 할 것이다. 만약 배우는 것 없이 대충 알고 있는 지식을 반복해서 사용하고 있다면 늘어나는 것은 요령일 뿐 제대로 된 지식과는 거리가 멀기 때문이다.

우리는 지식을 어떻게 축적하고 있을까? 일하면서 배우는 지식 대부분은 개인이 가져가고 있다. 그것을 문서로 만드는 것은 최소한의 미래에 대한 대비이며, 대부분의 지식은 대화로 전달되고 있다. 그리고, 작성한 코드 대부분도 컴퓨터와의 대화를 위한 것이 아니라, 코드를 읽는 다른 사람과의 대화를 목적으로 하는 고수준 언어(High Level Language)로 작성되고 있다. 문서를 만드는 것을 싫어하는 개발자들이 많기에 코드를 대화의 핵심 매개체로 주로 사용한다. 하지만, 그런 코드를 이용한 대화는 사실 더 다양한 지식을 요구한다. 대화하기 위해서 남들이 이해하기 쉬운 코드를 만드는 것은 말처럼 “쉬운”일이 아니기 때문이다. “가독성(Readability)”은 문제의 핵심에 대한 이해를 충분히 하고 쉬운 표현으로 바꿀 수 있는 능력을 요구하기 때문이다. 그리고, 이런 일을 정말 잘하는 사람을 우리는 전문가라고 부른다.

그렇다면 과연 우리는 전문가라고 자신있게 이야기할 수 있는가? 아마도 대부분의 소프트웨어 개발자들은 자신이 전문가로 인식되는데 부담을 가질 것이다. 왜냐하면 자신이 부족하다는 것을 이미 잘 알고 있기 때문이다. 그리고, 더 나은 것을 찾는 데 있어서 현실적으로 노력 할만한 "여유”가 부족하기 때문이다. 우리가 하는 일의 대부분은 여유가 거의 없는 일이다. 매일 같이 반복되는 야근과 주말 특근 시간에 해결할 일이 산적해 있기 때문이다. 조금이라도 여유를 찾으려면 위로부터 끊임없이 숙제가 떨어지거나, 기존에 만든 제품에 대한 하자(유지)보수로 인해서 정신없다. 새로운 제품을 개발하면서도 이미 판매된 제품의 유지보수와 기능 개선도 함께 병행해야 하기 때문이다. 우리는 지식 노동자라고 하면서도 제대로 된 지식을 축적할 만한 여유는 애시당초 없는 사람들이다. 비관적인 이야기라고 비판할지도 모르지만 현실에서 우리가 부대끼는 문제들은 바로 이런 것들이다.

지식 노동이라고 하면 기존의 지식을 이용하는 것만이 아니라 새로운 지식을 창출하는 부분이 더 중요하다. 지식은 세대를 통해서 전달되며 차곡차곡 기초부터 쌓여서 계층화를 이뤄간다. 가장 아래부분이 가장 단단해야 함은 말할 나위가 없다. 그리고, 발전이라는 것은 그러한 바탕 위에 새로운 무언가를 더 쌓는 것이다. 그 쌓이는 무엇인가가 또 다른 지식의 토대를 이룰 것임은 말할 필요조차 없다. 우리는 그런 일을 하는 사람들이어야 한다. 하루 하루의 삶 속에서 매몰되어 간다면 결국 남는 것은 정리되지 못한 경험밖에 없다. 정제되지 못한 경험들을 나열한다고 남에게 도움이 되는 것은 결코 아니다. 무엇인가를 더 보태는 삶이 되어야지 기존의 가치를 연장하려는 노력으로는 사람이 지식 노동이라고 불릴 수는 없다. 자신이 일하는 도메인에 대해서 많이 아는 것도 중요하지만, 지식을 구체화하고 현실에 가치를 더 할 수 있는 일반화된 기술도 필요하다. 따라서, 우리는 추상적인 “그림”을 현실에 고정할 “틀”도 함께 생각해야 한다. 도메인이라는 모호한 그림을 실체화시켜 줄 수 있는 구현 기술도 중요하다는 것이다. 알고리즘만 만든다고 모든 것이 해결되는 것은 아니며, C/C++ 혹은 Java, C# 등의 언어로 실행시킬 수 있도록 구현하는 것도 중요하다. 지식이 머리속에만 존재한다면 누군가에게 도움이 되는 진정한 가치를 만들지 못하기 때문이다.

지식은 쉽게 사라지지 않는다. 그것을 만든 사람보다도 더 오래 살아남을 수도 있다. 지식은 꾸준한 토대를 닦는 일에서 시작하지만, 어느 순간 비약적인 도약(Quantum Jump)이 있을 수 있다. 따라서, 지식 산업에 속한 일에 종사하는 우리는 기존의 가치와 새로운 방법론을 배우는데 게을리해서는 안 된다. 지속성을 가지고 이런 지식의 토대를 구축하지 않는다면 도약은 있을 수 없다. 대부분의 중간 관리자는 도약을 포기하고 기존의 가치만 최우선으로 생각한다. 하지만, 그것이 정말 가치 있는 지식이었나를 고민해 봐야 할 것이다. 대부분의 경우 객관적인 근거 없는 자신감이나 경험이라고 말할지도 모른다. 하지만 경험이 모든 것을 해결해 주지 않는다. 결국, 경험은 개인적인 부분에 한정된 것이며 객관적인 평가를 받은 것이 아니기 때문이다. 오히려 그런 경험을 중시한다면 흘러간 과거에서 나왔던 것들을 보는 편이 빠를 것이다. 30년 혹은 그 이전에 발생했던 문제들이 아직도 소프트웨어를 개발하는 현장에서 똑같이 존재하고 있는 것이 사실이다. 믿을 수 없이 단축된 일정과 아무런 근거 없는 요구사항들, 제대로 관리되지 않는 프로젝트들이 현장에서는 넘쳐난다. 개발자를 부속품쯤으로 여기는 높으신 관리자들의 성향도 사라지지 않았으며, 지식 노동자라고 불러주면서도 지식과 관련없이 시키는데로만 일하라고 듣기도 한다. 이런 식으로 일한다면 우리는 단순히 우리가 아는 한정된 지식에 값을 매겨서 팔고 있는 육체적으로 장시간 일하고 있는 노동자에 불과할 뿐이다.

# [ 잘리다. ]

회사에서 잘리면 어떤 기분일까? 아마 아무런 생각이 안 들고 멍한 기분일 것이다. 내일 아침에 눈을 떠서 갈 곳이 없는 남자는 집에서 어떤 존재가 될지 상상이 가지 않는다. 아마 누구나 자신이 잘릴 것이라는 막연한 가능성은 알고 있겠지만 그것을 부정하면서 살고 있을 것이다. 그리고, 그 순간이 왔을 경우에는 그냥 아무런 말없이 조용히 사라질 것이다. 아침에 일찍 갈 일이 없고, 막히는 도로에서 시간을 소모할 필요도 없고, 명령받는 사람도 명령하는 사람도 없으니 내릴 명령도 따라야 할 명령도 없을 것이다. 그렇게 사람들의 기억속에서도 사라져갈 것이다. 그리고, 그런 자리에 오른 사람이라면 그 정도의 각오는 항상 해야 할 것이다. 언제 어떻게 될지 알 수 있는 사람은 없기 때문이다. 하지만, 적어도 같이 술 한잔 나눌 수 있는 사람은 필요하지 않을까? 가끔 찾아와서 옛날 이야기를 나눌수 있을 정도의 친구는 있어야 하지 않을까? 누군가 자신이 떠난 자리를 허전하게 생각할 사람은 있어야 하지 않을까? 사람들을 올바른게 대했다면 그 정도는 기본이겠지만, 자신의 행동에 따르는 결과를 예측하지 않았다면 결국 혼자서 산을 오르는 것만이 유일한 위안이 될 것이다.

존경받는 사람이 되기를 원하는가? 그렇다면 존중하는 방법부터 알아야 한다. 남을 존중하지 않는데 그들로부터 진정한 존경을 끌어내지는 못한다. 실력이 좋다고 믿는 사람들은 다른 사람의 생각은 무시하는 경향이 많다. 하지만, 세상은 자신이 아는 지식의 테두리에 갖혀 있지 않다. 항상 누군가는 보다 더 잘하는 사람이 있기 마련이며, 그런 사람들의 좋은 가르침은 언제나 "진실"처럼 다가온다. 그런 것들을 무시하고 살아온 삶이 오히려 창피하게 보이기도 한다. 물론 나름대로 철학을 가지고 사람들을 대했을 것이다. 하지만, 그 철학이 올바른 것이었는지는 안방의 침대에서 한참을 고민해야 할 것이다. 왜? 적어도 자신이 걸어온 인생이 혹시라도 잘못된 길은 아닌지 확인해야 하기 때문이다. 그리고, 제대로 살았다면 자신이 항상 옆에 두던 인물들뿐 아니라 자그마한 도움이라도 주었던 사람들이 종종 연락해 올 것이다. 조금쯤 시간이 흐르고 마음이 차분히 가라앉으면 송별회로 불러 낼지도 모른다. 송별회는 말 그대로 그들과의 관계를 정리하는 마지막 시간이다. 물론 참여한 사람들의 생각은 제각각일 것이다. 표정 속에서 사람들의 축하하는 분위기라도 읽게 된다면 어떤 생각이 들까? 그저 사실을 있는 그대로 받아들일 수밖에 없을 것이다. 존경받지 못할 행동의 대가일 뿐이기 때문이다.

허탈할 것이다. 잘리고 집으로 돌아가는 길은 왠지 길어 보일 것이다. 아니면, 아무런 생각이 없이 그냥 운전대만 잡고 있을 것이다. 틀어놓은 노래도 들리지 않고 신호등은 본능적으로 따르고 있을 것이다. 이젠 뭘 하지라는 생각은 아직 들지 않을 것이다. 잘렸다는 사실을 아직 받아들이고 싶지 않고, 내일도 그냥 일상의 일부처럼 같은 생활을 할 것으로 생각할 것이다. 하지만, 내일은 달라진다. 회사의 내일에는 당신의 내일이 없다. 당신의 내일을 만들기 위해서 했던 것들에 대한 잘 잘못은 사람들에게 기억될 것이고, 이젠 그것들을 지우기 위해서 남은 사람들은 노력할 것이다. 당신의 지시사항들은 떠난 사람의 책임이라 묻히고 말 것이다. 당신이 했던 잘한 일은 이어받는 사람들의 몫이 되고, 당신이 잘못한 일들에 대해서는 온전한 비판을 당신의 뒤에서 들어야 할 것이다. 누군가 소곤거리는 소리에도 지나치게 반응할 수 있겠지만, 지금의 당신은 그런 소리에 무감할 것이다. 현실감이 떨어져 있기 때문이다. 지금까지 당신이 열심히 추구했던 것들은 무엇인가? 지금의 당신에게 중요한 것들인가? 아마도 아닐 것이다. 그렇다면 당신은 왜 그 자리에 있을 때 그렇게 행동했는가? 아마도 그 때는 지금을 상상 속에서만 그렸을 것이다. 자, 이제는 현실이다. 당신의 다음 선택은 무엇인가?

어쩌면 우리들의 삶이 이처럼 언젠가는 성장을 위해서 잘려나가는 나뭇가지일지도 모른다. 회사는 튼튼한 줄기를 언제나 찾아내려고 할 것이고, 잘못된 가지라고 판단되면 정원사의 가지치기 대상이 것이다. 하지만, 그 잘려나가는 순간까지는 계속해서 성장하려고 애써야 하고 열매를 맺으려고 노력해야 한다. 성장하지 않는 가지는 더 빨리 잘리고 말기 때문이다. 열매를 주지 못하는 나무는 더 생각할 필요도 없다. 삶은 반복적인 것이고 언제가 똑같은 상황에 부닥치게 될 것이고, 최소한 존경받는 삶을 살았다고 자부심을 가지고 싶어 할 것이다. 누군가의 기억에서 지워지지 않는 존재가 되어, 그 "위대한 생각"이 새로운 세대를 통해서 다시 잎사귀를 키우고 꽃을 피워 열매를 맺게 하고 싶을 것이다. 비록 우리는 사라지겠지만 또 다른 우리를 키워낼 수 있는 토대를 만들어 주고 싶은 것이다. 당신은 그런 사람이었나? 아마 존경받지 못한다고 생각했다면, 당신이라는 가지는 이미 말라서 비틀어진 상태일 것이다. 당신에게서 좋은 것들은 더 이상 나오지 않았을 것이다. 최고가 되라고 외치는 공허한 말뿐인 당신의 그림자는 새로운 잎사귀들이 자라기에는 힘든 그늘에 불과하다. 잘리고 나서 다음 날 아침에 일어나 반드시 지난날을 되새겨 보기 바란다. 당신이 정말 그들에게 필요한 존재였던가? 아마도 뼈저린 후회를 한다면 새로운 인생의 문을 두드릴 자격을 얻을 수 있을 것이다.

# [ 바쁘면 진다. ]

관리자들은 소프트웨어 개발자들이 항상 120%로 일하기를 바란다. 하루의 일과가 8시간이라면 야근으로 10시간에서 12시간 동안 책상 앞에 앉아있는다. 정시에 퇴근이라도 하려고 하면 "벌써 가?"라는 소리와 함께 암묵적인 압박을 준다. 하지만, 우리가 하루에 일하는 시간이 정말 8시간일까? 혹은 야근을 포함해서 10~12시간 동안 집중해서 일할까? 그렇지는 않다고. 대부분 정말 집중해서 일한다고 생각되는 시간은 하루에 4시간이 안 될 것이다. 중간에 끼어드는 회의와 소란한 사무실, 이런 저런 사람들과의 대화 등으로 상당 부분의 시간은 활용되지 못한다. 그리고, 전날 늦게까지 일한 후 다음날 오전에는 제대로 집중력을 발휘하지 못하는 것도 이유일 수 있다. 제대로 일할 수 있는 시간이 저녁으로 옮겨가는 이유는 그제야 급한 것들이 정리되고 정말 자신이 해야 할 일에 집중할 수 있기 때문이다. 야행성으로 바뀌는 사람들의 공통점은 자신의 시간을 충분히 확보하고 싶은 열정이 있다는 것이다.

관리자들이 소프트웨어 개발자를 아무리 다그치더라도 그들은 변하지 않는다. 그리고, 그렇게 해서도 안 된다. "말을 물가에 끌고 갈 수는 있지만 목마르지 않은 말에게 억지로 물을 먹일 수는 없다". 아무리 책상 앞에 오래 앉아있게 하더라도 생각은 이미 집에 가 있다. 차라리 그들에게 "의무"라는 족쇄를 풀어주고, 스스로 능동적으로 일하는 팀 문화를 만들어야 자신들의 능력을 발휘하기 시작할 것이다. 할 일을 스스로 결정하고 몰입하는 환경을 만들어야 주어야 한다는 말이다. 아이들을 가르칠 때 억지로 공부한 아이와 스스로 찾아서 공부한 아이의 성취도가 확연히 다르다는 사실을 잘 알 것이다. 하지만, 소프트웨어 개발자들은 그런 아이들보다 적어도 더 많은 교육과 고정된 관념을 가진 사람들이다. 그들에게 억지로 물을 먹이려고 한다면 관리자 스스로가 물에 빠져 허우적거리는 모습을 볼 것이다. 당신은 그들을 능동적으로 움직일 수 있는 어떠한 도구도 가지지 못했기 때문이다. 잘해야 돈으로 조금 보상하는 수준으로는 머리굵고 직업병처럼 배 나온 사람들을 따르게 할 수 없다. 눈앞에서 잠시 머리를 숙일지는 몰라도 당신이 가고 나면 곧바로 잊어버릴 것이다.

바빠선 지게 되어 있다. 수학 문제를 풀 때를 생각해보자. 서둘러서 계산한 결과가 정확한 적이 얼마나 되던가? 천천히 했으면 100점을 받았을 것을 서둘러서 제대로 문제를 보지 않아 틀리는 경우를 자주 경험했을 것이다. 하물며 수학처럼 문제별로 나누어진 것들도 그런데 복잡한 관계를 형성하고 있는 소프트웨어는 어떻겠는가? 결과적으로 바쁘면 품질을 확보하기는 더 어려워진다. 그렇다고 개발자를 놀게 만들라는 말은 아니다. 바쁘면 안 바쁘게 만들 수 있는 방법을 찾으라는 말이다. 항상 시간이 없다고 이야기하는 사람들을 보면 몇 년이 지나도 그 자리에서 머무는 것을 본다. 그들은 언제나 바쁘다는 이야기만 할 뿐 조금도 더 나아갈 생각은 하지 않는다. 바쁘다는 것이 핑계는 아니지만, 그것이 모든 변화 거부에 대한 변명이 되는 것도 아니다. 노는 사람들이 부럽다면 그들이 왜 놀면서 일해도 결과가 좋은지를 연구해야 하지 않을까? 물론, 일의 내용이 다를 수도 있다. 하지만, 일에 대한 접근 방법이 다르다면 반드시 더 좋은 방법이 있다는 것을 깨달을 것이다. 하루 하루의 생활에서 자그마한 개선이 1년만 가도 확실히 그 차이를 볼 수 있을 것이다. 신입 사원이 들어와서 지금의 모습과 1년 후의 자신의 모습을 볼 때, 흔히들 다른 부서의 동기들과 알게 모르게 비교를 하곤 한다. 그들이 자신은 뒤처져가는데 다른 부서의 신입사원들은 일을 받아서 척척 해내고 있는 모습을 발견한다면 어떤 마음이 들까? 물론 개인의 능력의 차는 인정해야 하지만, 적어도 선배라면 바쁘다는 핑계로 후배를 가르치는 일을 게을리해선 안된다. 시간이 없다고 미래까지 포기해서는 안되는 것이다.

시간은 똑같이 주어진다. 누구나 24시간을 살아가고, 대부분 사람은 100살이 되기 전에 흙으로 돌아간다. 항상 바빠다면 개선은 기대하기 어렵다. 누군가 악순환의 고리를 끊어주기를 바라지만, 그것은 쉽지 않은 일이다. 왜냐하면 해야할 일을 가장 잘 아는 사람은 바로 당신이기 때문이다. 담당자를 대신해서 자신의 일을 가장 잘 할수 있는 사람은 없다. 바쁘다면 일은 항상 지체될 것이 분명하고, 지체된 일은 뒤로 갈수록 더 많이 지체되게 된다. 1분이 10분이 되고, 다시 1시간, 1일, 1주일, 한 달, 석 달, 육 개월, 일 년 등등 얼마나 늘어날지는 알 수 없다. 당신의 과제가 성공하기 위해서는 일을 줄여야 한다. 그리고, 한 템포 쉬어가면서 일하지 않는다면 계속 지체만 될 것이다. 개선 또한 불가능하다. 개선은 여유가 있어야 할 수 있는 선택이기에 그렇다. 누군가 바쁜 와중에 찾아와서 "이것이 잘못되었으니 고치세요"라고 한다면, 당신의 첫 반응은 아마도 "웃기고 있네"라는 생각이 머리속을 스쳐 지날 것이다. 하지만, 정작 웃고 있는 사람은 당신이 아니라, 지연된 제품 개발을 어디선가 지켜보고 있는 다른 경쟁사의 소프트웨어 개발자들이다. 경쟁은 바빠야 이길 수 있는 것이 아니라 한가해야 이길 수 있다. 더 높은 가치를 만들 수 있는 "삶의 여유"를 찾아야 한다. 그것이 "혁신적인 생각의 기초"가 되기 때문이다. 바쁘게만 일한다고 가치있는 일을 하고 있는 것은 결코 아니다.

# [ 직업 안정성(Job Security) ]

생계를 위해서 직장을 다녀야 한다면 살아남는 것이 제일 중요한 일이다. 하지만, 직장에서의 생존이 중요해지기 시작하면 이미 반쯤은 생명이 죽어버린 개발자와 같다. "생존 욕구"는 가장 낮은 단계의 욕구로 아직도 그 수준에서 머물고 있다면, "자아실현(Self-Realization)"은 너무나 먼 이야기라 거의 관심도 없을 것이다. 하지만, 우리 대부분 직장인이 처한 현실은 어쨌든 생존이 최고 우선이다. 내가 살아남아야 남도 눈에 보인다. 당장 먹을 것이 없는데 우아한 옷이 무슨 상관이겠는가? 그래도 지켜야 하는 직장인의 도리는 있다. 소프트웨어 개발자라면 적어도 "윤리 의식"과 스스로에 대한 "정당함"은 항상 갖추고 있어야 한다. 자신이 개발하던 과제가 다른 과제로 인해서 중요한 변화의 순간에 있다면 어떤 결정을 내릴 것인가? 이것은 기존의 환경에 적응해서 사는 특정 개발자들에게 생존문제와 즉각적으로 연계되어 있기 때문에 더욱 민감한 일이 될 수 있다.

새로운 플랫폼을 도입하거나 새로운 프로세스를 도입하려고 할 때 가장 반대가 심한 쪽은 실무 개발자들이다. 그들의 저항감은 자신들이 지금까지 쌓아 놓았던 안전한 토대를 허물고 새로운 것으로 대체하자는 말로 들린다. 그리고, 그들은 최대한 자신들의 역량을 발휘해서 어떤 식으로든 기존의 자신이 만든 가치를 지켜내려고 할 것이다. 이것은 높은 직위로 갈수록 더 심해지게 되며 내부적인 경쟁을 유도하게 된다. 이런 상황이라면 새로운 것을 도입하려는 팀과 옛 것을 지키려는 팀들 간에는 협업할 수 없는 상태가 된다. 둘은 서로를 비판할 기회가 주어진다면 모든 능력을 남김없이 활용할 수 있는 준비를 할 것이다. 어떤 식으로든 자신들의 밥그릇을 지켜야 하기 때문이다. 이런 상황은 기존의 제품에 대한 더 이상의 유지보수나 개선 노력이 어려운 경우나, 새로운 제품에 기존의 것을 적용하기 어려울 때 발생한다. 기존 제품의 구조적인 문제점들이 나열되고, 개발팀의 힘으로는 역부족인 경우 새로운 개발팀을 구성해서 개선과제를 준다. 기존 개발팀은 상품화 과제를 계속 진행하게 되지만, 새롭게 구성된 개발팀은 기존 개발팀의 결과물을 바탕으로 개선을 추진하게 된다.

관리자의 입장에서는 둘 간의 경쟁이 항상 나쁜 것은 아니다. 그리고, 회사 차원에서도 더 좋은 제품을 만들려는 노력은 좋은 결과로 나올 수도 있다. 하지만, 문제는 한정된 자원으로 두 가지 일을 다 해야 한다는 점이다. 그리고, 시장은 항상 새로운 기능 추가를 요구하고 플랫폼이나 프로세스도 그것에 맞춰서 변화해 나가야 한다는 것이다. 플랫폼 개발팀이 어느 정도 기존 제품의 기능을 따라갔다고 생각할 때, 기존 개발팀에서 새로운 기능이 추가될 경우 "기능상의 미비"라는 약점을 가질 수밖에 없다. 의지가 있는 책임있는 관리자라면 새로운 플랫폼을 제품에 도입하는 데 힘을 실어줄 수 있을 것이다. 그리고, 그 핵심은 역시 협력의 형태로 주어질 것이다. 협력이란 서로 일방의 희생일 요구하는 것이 아니라, 상호 이익이 생길 때 효과를 발휘할 수 있다. "변하지 않는 진실은 모든 사물은 항상 변한다는 것"이다. 소프트웨어도 지속적인 유지보수를 거치면서, 더이상 고치는 것이 오히려 새로 개발하는 것보다 못하다고 판단될 때는 새롭게 만들어야 한다. 단순히 자신의 "Job Security"를 위해서 반대를 외친다면, 안타깝지만 이제는 그 자리에서 떠날 때가 되었다는 말이다. 떠나야 할 사람은 변화에 적응하지 못하는 당신일 가능성이 높다. 물론, 사내의 정치 싸움에서 승리할 수도 있겠지만, 변화를 거부하는 회사는 결국 망가질 것이다. 시장은 당신의 생존보다 더 좋은 제품을 선택할 것이 분명하기 때문이다.

코딩을 자신만이 알아보게 하는 사람들도 있다. 구조가 간단하다고 이야기하면서 제대로 된 문서는 하나도 없는 상태에서 가독성이 떨어지는 코드를 내민다. 그것으로 당신의 생존 욕구는 채워줄지 몰라도 다른 사람의 갈증은 해결하지 못한다. 당신은 꾸준히 자신의 힘을 유지할수 있을지는 몰라도 당신 이외의 사람들은 그런 당신으로부터 떠나갈 것이다. 떠받들어줄 사람이 없는 왕이 되고 싶다면 그냥 그 자리에서 왕관을 쓰고 있어도 상관없다. 아무도 그런 것에 관심을 두지 않을 것이다. 남의 개선으로부터 배우는 것도 생명력을 유지하는 한가지 방법이다. 그렇다고 하더라도 자신이 만들었다는 거짓말은 하지 말도록 하자. 상식없는 억지도 쓰지 말아야 한다. 그건 오히려 당신의 무지함만 보여줄 뿐이다. 필요하다면 도움을 요청해야 한다. 모른다면 물어야 하고 제대로 된 대답을 들었다면 고마워할 줄도 알아야 한다. 그것은 당신의 무지를 일깨워준 것에 대한 최소한의 해야할 도리다. 솔직함은 언제든 사람의 손을 빌릴 수 있는 훌륭한 도구이기에 자신의 잘못을 인정하고 도움을 요청하라. 생존이 필요하면 생존의 뒷받침이 될 수 있는 안도감이 어디서 오는지를 정확히 알아야 한다. 근본적인 안도감은 변화로부터 자신을 보호해 줄 수 있는 "유연함(Flexibility)"에 있지, 변화를 거부하는 "경직됨(Rigidity)"이 아니다. 예전으로 돌아가 암흑의 동굴 속에서 안전함을 느낀다면, 두려움에 잠 못 드는 긴 밤만이 당신을 기다릴 것이다.

# [ 학교와 회사의 차이 ]

학교에서는 개인이 경쟁하고 회사에서는 조직이 경쟁한다. 하지만, 엄밀하게 말하면 회사도 개인의 경쟁이다. 평가는 개인별로 다르게 받기 때문이다. 팀이 성과가 없다면 그 팀에 속한 모든 팀원이 나쁜 평가를 받아야 하지만 그렇지는 않다. 누군가는 A를 받고 또 다른 누군가는 D를 받는다. 회사의 평가체계 자체가 개인별로 주어지는 방식이기 때문이다. 다만 성과를 보상할 경우에는 팀 단위가 된다. 그리고, 한 가지를 더 보탠다면 평가가 객관적인 자료에 기반을 두지 않는 경우도 더 많다는 것이다. 평가를 위해서는 피평가자들에 대해서 가장 가까운 곳에서 일한 사람이 평가자 역할을 맡는 것이 타당하지만, 대부분의 경우 그보다 더 위에 있는 사람이 평가하게 되며 고과권을 가진다는 것은 나름의 큰 힘을 피평가자들에게 발휘할 수 있는 위치로 올려놓는다. 평가의 기준을 이야기하지만 공정하게 하는 것은 거의 불가능에 가깝다. 그냥 느낌에 의지한 평가가 대부분이라는 것이다. 근거자료는 부족하고 고과 면담도 이뤄지지 않는 경우도 흔하다. 평가 자체가 공정하기 어렵다는 것이다.

고과 면담할 시간이 없어서 올해의 성과와 개인적인 것들을 이메일로 보내라는 관리자도 있었다. 그들이 정말 시간이 없었다면 시스템에 옮겨 적을 시간도 없어야 한다. 쉽게 그냥 "당신들과는 할 이야기가 없다"라고 말하는 것이 오히려 솔직한 태도다. 자신이 의지하는 직원들과 가장 중요한 대화를 하지 않는 관리자는 이미 자신의 역할을 포기한 것과 같다. 한 사람 한 사람을 다 기억할 수는 없지만 그래도 조금 더 알아보기 위해서라도 개별 면담을 해야 하지 않을까? 학교는 그나마 정확한 결과를 보고 개인의 성적을 정한다. 학생들은 중간고사, 기말고사, 주기적으로 있는 작은 평가들을 받고, 자신이 제출한 과제의 평가도 더해서 최종 결과를 받아든다. 하지만, 회사는 그런 것이 없다. 시간에 맞춰서 일하지만 회사가 정한 시간이 아닌 고객이 정한 시간에 맞춘다. 따라서, 결과물이 나오는 시간과 평가받는 시간이 일치하지 않는 경우도 흔하다. 과제는 진행하지만 과제의 결과물이 한 번에 끝나는 경우는 드물다. 지속해서 계속 유지보수와 추가적인 기능들을 보완해야 하기 때문이다. 학생들의 과제는 해당 과목이 이수되면 버려질 수도 있지만, 회사가 만든 제품은 시장에서 얼마나 사용될지 예상하기 어렵다.

학교는 비용과 일정의 제약이 덜한 대신 인력의 지속성이 부족하다. 항상 새로운 일을 찾아야 하지만, 그것을 지속해서 발전시켜 나갈 수 있는 꾸준히 투입되는 인력들이 없다. 회사는 사업의 영속성 및 효율성을 추구하기에 한 분야에서 오랫동안 일하는 사람들이 많다. 한 가지 과제를 신입사원 때부터 맡아서 정년이 될 때까지도 30년 동안 하는 경우도 있다. 학교는 다양한 과목에서 우수한 성적을 받아야 장학금을 주지만, 회사는 한 가지 일에 특화될수록 몸값이 높아질 수 있다. 특히 남들이 힘들어하는 분야를 할수록 더 높은 몸값을 받을 수 있다. 학교는 남들이 안가는 학과의 경쟁률이 낮지만, 회사는 전체 시스템을 구축할 필요가 있기에 부문에서 사람을 필요로 한다. 물론 그렇다고 이것이 항상 진실은 아니다. 학교에서 만든 성적이 허위가 아니면 회사에서도 개인의 기본기는 어디로 가지 않는다. 하지만, 경쟁의 형태가 바뀌어 선택과 집중이 더 중요해진다는 것이다. 이론적으로 옳은 것이 항상 최고가 되는 것은 아니다. 회사는 주어진 예산을 활용해서 시장에 팔 수 있는 물건을 만들어야 하지만, 흔히 예산을 벗어나는 일이 발생한다, 학교는 정해진 학점 내에서 과목을 더 듣는다고 등록금을 추가로 달라고는 하지는 않지만, 수준을 만족시키지 못하면 추가 등록을 통해서 해결할 수 있다.

학교에서 익힌 지식이 회사에서 완전히 무용지물은 아니다. 하지만, 회사에서 익힌 것들이 이론적으로 항상 최적은 아니지만 더 오래 살아남는 경우도 있다. 특히 그것이 검증되었다고 여겨진다면 이론을 무시하기도 한다. 중요한 것은 돈을 많이 버는 것이지 학문적으로 올바른 방법을 선택하는 것은 아니라고 보기 때문이다. 이론적으로 정리는 안 되었지만 현상을 재현할 방법이 있다면 사업화 대상이 될 수 있다. 이론은 나중에 더 많은 관찰을 통해서 정리될 순간이 있을 것이기 때문이다. 남들보다 먼저 시장에 출시하는 것이 이론보다 중요하기 때문이다. 이런 상황에서는 개발 방법론이 어떤 것이 좋고 나쁘냐를 따지는 것은 의미 없는 일이다. 어떤 방법론이라도 제품을 빨리 만들 수만 있다면 아무 상관이 없다. 시간을 놓치면 시장은 다시 열리지 않을 수도 있기 때문이다. 이런 이유로 인해서 회사들은 많은 실패를 반복적으로 경험하는 것도 사실이다. 근거 없는 확신으로 진행한 과제가 결국에는 발목을 잡고, 더는 나아가지 못하게 만들기도하기 때문이다. 회사는 실패를 경험하지 않기 위해서 학교에 도움을 구하고, 학교는 도움을 줌으로써 과제에 대한 비용을 가져간다. 서로 보완적이라고 믿지만 학교에서 만든 결과물이 회사에 바로 적용되지 않는 경우도 흔히 볼 수 있다.

# [ 명령과 선택 사이 ]

경험이 많을수록 그 경험에 의지하게 된다. 하지만, 모든 상황을 경험한 사람은 세상에 없으며, 유사한 상황이라고 하더라도 소프트웨어 개발은 항상 새로운 요소가 많다. 따라서, 경험에 근거한 명령고 항상 옳은 판단이 되는 것은 아니다. 경험이 많다면 경험 속에 녹아있는 “겸손함”도 함께 배워야 한다. 겸손하지 않고 상대를 존중하지도 않는 명령은 하지 않는 것이 좋다. 하지만, 우리는 곧 잘 대화가 되지 않는 상황에 처하면 그냥 무조건 하라는 이야기한다. 선택은 개인의 몫이다. 명령받은 것을 무조건 따른다면 이미 선택권은 없다. 그렇다고 명령의 형태로 주어지는 요청들을 거부할 권한을 가진 사람은 거의 없다. 왜냐하면, 선택 후에 돌아오게 되는 모든 결과가 충분히 예상되기 때문이다. 명령권을 가진 자들은 고과 권한도 함께 가지고 있기 때문이다. 그들의 손아귀에서 빠져나갈 방법은 별로 없다.

과제는 명령을 실행하는 것이 아니다. 과제는 팀원들의 선택으로 진행된다. 그 선택을 스스로가 하느냐 그렇지 않은가에 따라 결과물의 품질이 달라질 수 있다. 누가 시켜서 억지로 하는 과제라면 최소한의 노력을 들여서 명령을 한 사람을 만족시키려고 할 것이다. 물론 만족할 가능성은 그렇게 크지 않다. 만족이란 누가 시켜서 일하는 사람으로부터 끌어낼 수 있는 결과가 아니기 때문이다. 선택권이 주어진 팀은 자신들의 능력 범위에서 가장 높은 만족도를 제공하려고 노력할 것이다. 물론 그 이상을 끌어낼 가능성도 있다. 비록 만족시키는 데 실패하더라도 더 높은 역량을 간직한 팀이 될 가능성이 높다. 자기 일을 스스로 결정할 수 있는 선택이라는 것을 해본 경험이 있기 때문이다. 선택 전에 필요한 것들을 충분히 고민했을 것이고, 결과에 따른 책임감도 이미 느꼈기 때문이다. 결국 선택이라는 것을 할 수 있어야 책임감도 느낄 수 있게 된다. 명령만 받는 사람은 선택권이 없는 사람들이며 명령에 따른 결과를 책임질 이유도 없다.

관리자는 명령을 선택이라고 믿는 경향이 강하다. 그리고, 자신 스스로도 그렇게 믿도록 만든다. 내려진 명령을 자신이 한 판단이 아니라 팀원들이 결정한 것이라는 굳게 믿어버린다. 그 후에는 책임이 오로지 팀과 그 팀원들의 몫이라고 여긴다. 하지만, 그런 생각은 자유일지도 모르지만 결과는 분명하게 받아들여야 한다. 명령을 수행한 책임은 담당자에게 있는 것이 아니라, 명령을 내린 사람에게 있다는 것이다. 그리고, 동의를 가장하더라도 명령을 내린 당사자는 권한을 가진 사람일 뿐이다. 권한이 없는 사람은 책임질 일도 없다. 책임이 모호한 과제는 당연히 실패할 것이다. 고지식한 관리자는 책임은 강요하지만 권한은 제대로 주지 않는다. 세심한 관리와 지속적인 괴롭힘을 구분하지 못하는 것도 마찬가지다. 자신은 모든 과제를 다 잘 알아야 한다는 강박관념에 사로잡혀 있다는 것을 모른다. 모든 과제를 다 잘 안다면 이미 관리자의 역할이 아닌 팀원이 되어있을 것이다. 팀원과 같이 행동하지 않으면서도 모든 결정을 혼자서 해버린다면, 과제의 리더는 필요없는 자리일 뿐이다. 중간 관리자의 역할은 상위 관리자가 정확히 판단할 수 있는 근거와 진행 현황을 보고하고, 팀 내부에서 맞닥뜨릴 수 있는 문제에 대해서 순간 순간의 판단을 내리는 것이다. 하지만, 권한이 주어지지 않고 책임만을 문책 당한다면, 어떤 판단도 스스로 할 수 없는 존재가 된다. 조직을 무용지물로 만드는 가장 쉬운 방법은 선택하지 못하도록 명령만 하는 것이다.

팀과 팀원들은 자신의 판단으로 선택할 수 있어야 한다. 그리고, 그런 결정이 왜 이루어졌는지 상위 관리자에게 알려주어야 한다. 물론 상위 관리자의 역할은 과제의 모든 세세한 결정을 자신의 업무라고 여겨서는 안 되며, 큰 방향을 가진 결정들에 대한 논의와 논리를 근거로 한 판단을 내려야 한다. 세세한 판단은 큰 방향에 대한 지침에 기반을 둔 팀원들이 스스로 내리는 것이다. 그 방향이 잘못되었을 때의 책임은 당연히 관리자의 몫이 되며, 과제의 진행에 걸림돌을 제거하는 것도 그가 해야 할 일이다. 팀이 내리는 모든 결정에 대한 근거가 확실하다면 그것에 대한 잘못을 팀에 물어선 안 된다. 하지만, 팀이 제대로 방향을 찾아가지 못할 경우에는 관리자의 적극적인 개입도 필요하다. 하지만, 이 때도 명령이 아닌 선택권을 팀에 주어야 한다. 명령은 결국 가장 마지막 수단이 되어야지, 가장 손쉬운 선택이 되어서는 안 된다. 누구나 쉽게 명령은 내릴 수 있지만, 누구나 명령의 책임을 가지려고는 하지 않는다. 선택권이 주어지지 않은 팀은 주도적인 판단력을 잃어버리게 되며, 다른 누구의 만족보다 관리자만을 위해서 일하게 될 것이다. 정말 당신만 만족하면 다른 모든 사람도 만족할 수 있을까? 그렇지 않다고 생각한다면 결정은 팀에 맡기는 것은 올바른 선택이다.

# [ 어려운 시간이 오면 진실이 드러나게 된다. ]

최적과 최선은 무엇이 다를까? 최적은 모든 가능성의 조합에서 가장 좋은 것을 찾아낸다는 뜻이다. 최선은 모든 가능한 입력의 조합이 불가능하거나 그렇게 하기 위한 비용이 많이 드는 경우, 특정 범위 내에서 가장 좋은 것을 선택하는 것이다. 우리가 찾아내는 대부분의 해결책은 최선의 선택인 경우가 많다. 이유는 단순하다. 최적을 만들어낼 만큼의 시간과 비용이 없다는 것이다. 그렇다고해서 우리가 선택한 모든 해가 전부 최선인 것은 아니다. 최선이 되기 위해서는 일반화하기에 충분한 실험이 필요하지만, 대부분은 몇 번의 실험으로 최적이라고 오해하고 만다. 도대체 얼마나 많은 실험을 해야지 우리는 최선의 선택을 할 수 있을 만큼의 충분한 근거를 확보할 수 있을까? 만약 실험 자체가 필요없는 최선의 방법이 이미 있다면 어떨까? 이미 많은 사람이 같은 실수를 반복해서 충분한 근거를 확보하고 있다면, 우리는 그것을 그냥 사용해도 무방하지 않을까? 하지만, 사람들은 자신이 직접 해보지 않은 것은 쉽게 믿지 않는다. 그것이 비록 잘못된 것이라고 할지라도 아집과 신념을 구분하지 못한다.

어려운 시간이 오면 진정한 친구와 적을 구분할 수 있다고 한다. 누구나 자신의 입장에서는 최고의 선택을 하였다고 하지만, 잘못된 것을 선택하면 잘못된 결과만을 낳을 뿐이다. 책임을 맡는 자리에 있다면 자신의 선택이 어떤 결과를 불러올지 정도는 생각할 수 있어야 한다. 소프트웨어 개발의 속성을 모르는 높으신 임원들은 자신의 무지함을 집착이나 고집으로 극복하려고 애쓰겠지만 잘못된 판단은 잘못된 판단일 뿐이다. 높은 자리는 아집이나 선입견보다 논리적인 근거를 바탕으로 유지되어야 한다. 비이성적인 판단으로 순간의 위기를 모면하고 흐름에 올라탈 수 있을지는 몰라도, 그 흐름의 끝에 있을지도 모르는 낭떨어지는 대비가 되지 않는다. 무조건 열심히 한다고 다 되는 것이 아님을 이미 경험하지 않았던가? 회사가 망하는 것은 열심히 하지 않아서가 아니라 제대로 하지 않아서다. 제대로 한다는 것이 무엇인지 몰라서 우왕좌왕하다가 기회를 놓치고 말았기 때문이다. 장기적인 성과를 보지 못하고 단기적인 성과에 집착해서 망가진다. 무엇이 올바른 결정이었는지는 시간이 알려준다. 어려운 시간을 겪으면서도 버텨낸 회사들은 최적은 아니지만 최선의 선택을 하기 위해서 필요한 변화를 기꺼이 받아들였기 때문이다.

변화하지 않는다면 살아남지 못한다. 이건 자연의 섭리와도 같다. 오래동안 변화하지 않고 존재할 수 있는 기업은 없다. 중소기업은 특화된 기술로 장인 수준에 올라서 살아남을 수 있을지는 몰라도 큰 집단을 이룬 기업에서는 그런 정신이 점점 드물어진다. 조직이 커지면 커진만큼 움직임에 대한 민첩성은 떨어지게 되고 특정 기술에 대한 의존도는 낮아진다. 최근에는 많이 생산하는 것보다 정확히 생산하는 것이 더 옳다는 생각에 민첩도를 강화하기도 하지만, 아직도 변화에 순응하는 태도는 느리게만 보인다. 왜 그럴까? 밑바탕에는 장인정신을 인정하지 않는 태도와 전문가의 의견보다는 자신의 손맛을 우선시하는 상명하복의 문화가 있기 때문이다. 회사에는 많은 전문가가 있다. 하지만, 그 전문가들의 결과물에 대한 판단은 이른바 그 분야에서 오래동안 일한 경험이 있다는 비전문가의 "감"에 의존한다. 그리고, 판단을 좌지우지하는 사람들은 대부분 고위층에 속한다. 그들이 정말 전문 디자이너보다 "감각"이 있을까? 소프트웨어를 개발하는 방법론의 전문가는 있지만 그들이 제시하는 프로세스는 언제나 실무 개발에서 무시되기 일쑤다. 웹 프로그래밍 배운 사람을 임베디드 소프트웨어 개발에 배치하고, 코덱(Codec)을 전공한 사람에게 UI를 만들도록 하는 것이 정말 타당한 일일까?

어려운 시간이 오면 이런 모든 것들이 눈에 드러날 것이다. 매출이 나고 이익이 날 때는 당신들의 세상을 만난 것처럼 축하하고 승진할지는 몰라도, 힘든 시간이 오면 그런 사람들은 아무런 도움이 못 될 것이다. 그들은 기존의 가치만을 맹신해서 변화를 거부한 사람들이기에 새로운 술잔을 채워줄 그릇이 되지 못하기 때문이다. 그들의 잔은 이미 가득 차 있고, 다른 것을 받아들이기 위해서는 비워야 한다는 사실을 모른다. 가득 찬 술이 아까워서 비우지 못하고 있는 것이다. 그런 사람들에게 새로움이란 그냥 또 다른 같은 한 잔의 술일 뿐이다. 가치가 같다고만 여기는 것이다. 하지만, 분명히 다르다. 그것은 여태껏 당신들이 찾으려고 노력한 “그것”이기 때문이다. 그리고, 그것을 알기 위해서는 아깝다고 생각하는 것들을 버려야 한다. 버리기가 싫다면 그냥 그것만을 부여잡고 살아야 할지도 모른다. 그럼 더더욱 뒤처질 것이고 결국 더 이상의 발전은 힘들어질 것이다. 장인정신이라도 있다면 살아남을지도 모르겠지만, 자신의 기술을 갈고 닦지 못한 당신들이 장인이 될 가능성은 전혀 없다. 노력하지 않고는 아무것도 얻을 게 없고, 비우지 않고는 다시 채울 수 없다는 간단한 진리를 잊고 산지 너무 오래되었기 때문이다.

# [ 믿을 것은 자신밖에 없다. ]

소프트웨어 개발자로 회사생활을 하다 보면 제대로 일이 진행되지 않거나, 진행하던 일이 갑자기 바뀌는 경우가 흔하다. 그리고, 때로 그런 일이 생기면 일할 마음을 잃고 방황하기도 한다. 아무 생각 없이 인터넷을 돌아다닌다든가, 의자에는 앉아있지만 생각은 이미 컴퓨터 앞을 떠나 다른 곳에 가 있다. 주어지는 업무는 잡다한 것으로 채워지고, 별로 영양가도 없어 보이는 일을 윗사람은 계속 시킨다. 윗사람도 무엇을 해야 하는지 모르기 때문이다. 그도 항상 더 위에 있는 누군가의 지시를 받아서 일하기만 해왔지, 자신이 해야 할 일을 스스로 결정한 적이 없기 때문이다. 일을 스스로 결정하고 언제 어떻게 무엇을 할지를 정해본 적이 없는 사람들의 공통점은 주어지는 일이 없을 때 아무것도 못 한다는 점이다. 갑작스러운 이런 상황들을 극복하기 위해서는 홀로 지내는 법을 알아야 한다. 믿을 수 있는 사람은 어쨌든 자신밖에 없기 때문이다.

회사에서 시간이 생기면 주로 어떻게 보내는가? 설마 예전에 만든 제품의 코드를 개선한다고 손대고 있지는 않을 것이다. 대부분 동작하는 코드를 시간이 난다고 고치는 경우는 거의 없다. 그렇다면 주로 허드레 잡일을 하고 있거나 못다 읽은 각종 서적을 뒤적이고 있을 것이다. 이때는 인내의 시간이 필요하다. 아무도 무엇인가를 요구하지 않기에 기존에 주어진 일을 하던 방식으로는 견디기 힘든 시간일 것이다. 가장 먼저 할 일은 계획을 세우는 일이다. 단기, 중기, 장기의 계획이 필요하다. 단기는 6개월이나 1년, 중기는 2년 내지 3년, 장기는 5년 이상이면 충분하다. 그 각각의 시기에 맞게 자신의 모습을 상상해 보고, 상상한 모습을 위해서 필요한 것들을 나열해 보자. 생각보다 해야 할 것들 많을 수도 있고 그렇지 않을 수도 있다. 너무 많다면 중간 목표를 다시 세우고, 세워진 목표들을 조금 더 단기에서 중기로, 중기에서 장기로 옮기도록 한다. 자신의 미래의 모습이 없다면 지금부터 만들어도 상관없다. 무엇이 되고 싶은가를 정하는 자체가 중요하다. 그렇게 무엇인가 되고 싶다는 "의지"가 만들어지게 되기 때문이다.

아무것도 하지 않는 것보다 무엇인가를 하는 것이 좋겠지만 마음을 추스르는 데는 한계가 있다. 일단은 재미있는 것을 해야 한다. 어떤 일을 본격적으로 하기 전에 전문가가 되기 위해서 대략 “10,000시간”이 필요하다고들 한다. 재미있는 일이 자신의 경력에 도움이 된다면 더 좋겠지만, 하는 일과는 직접적인 관련이 없을수도 있다. 하지만, 그래도 상관없다. "마음의 안정"을 얻기 위한 도구로 활용할 수 있기 때문이다. 즐거운 일은 상상하는 것만으로도 얼굴 표정을 바꾼다. 찌푸린 인상에 미소가 피어오르고 사람들과의 대화에도 따스함을 전해준다. 소프트웨어 개발자의 궁극적인 목표가 "도메인에 대한 지식을 익혀서 그 분야의 최고 소프트웨어 전문가가 되는 것"이라면, 현재 하고 있는 일이 무엇인지 깊이있게 봐야 한다. 단순히 자기에게 주어진 부분만 보아서는 안 된다. 차분히 정리가 된다면 다시 기본부터 시작하는 것이 좋다. 특히, 해당 분야에서 시용되는 "용어"들은 정말 잘 익혀두어야 한다. 전문 용어의 사용은 당신이 전문가임을 나타내는 것과 동시에 정확한 의사전달의 기초가 되기 때문이다. 정말 전문가가 되고 난 후에는 그런 용어들도 다른 더 쉬운 말들로 대체될 수 있을 것이다.

시간은 누구에게나 똑 같이 주어진다. 누군가는 많은 것을 이루고 누군가는 남이 이룬 것만을 바라보게 된다. 차이는 단순하다. 무언가를 하고자 하는 "의지"가 없기 때문이다. "권한에 따른 책임"이 주어지는 일을 해보지 않아서 주어지는 일만 열심히 받아먹었기 때문이다. 이제는 그것을 바꾸어야 하지 않을까? 당신에게 무엇을 주어야 할 지도 모르는 사람들 속에서, 유일하게 믿을 것은 자신밖에 없다는 생각이 들지 않는가? 누구도 대신 당신의 삶을 살아주지는 않는다. 1년 후, 5년 후, 10년 후의 당신의 모습을 결정하는 것은 현재의 당신밖에 없다. 가능성을 현실로 바꾸는 노력을 하지 않는다면 그저 회사 한 구석에 처박혀 상상만으로 살아가는 당신의 모습을 보게 될 것이다. 걸어 다니는 당신의 미래가 저만큼 이미 다가오고 있을지도 모른다. 그것이 "부장"이건 "수석"이건 간에 당신은 자신의 미래를 그들의 무개념함에서 찾을 수 있을 것이다. 그 때 똑같이 아무 생각 없이 사람들에게 잡일을 시키면서 시간을 보내게 할 것인가? 아니면 그들의 "인생 선배"로서 정말 중요한 것들을 같이 이룰 수 있는, "권한과 책임"을 간직한 사람들로 "자율적인" 소프트웨어 개발자가 되도록 만들고 싶은가? 그것 또한 오늘 하게되는 당신의 선택에 달려있을 것이다.

# [ 좋은 소프트웨어 개발자가 되고 싶다면 ]

아마도 소프트웨어 개발자를 하고 싶은 사람들은 "좋은 소프트웨어 개발자"는 어떻게 되는지 궁금해할 것이다. "좋은 소프트웨어 개발자"라는 말이 멋지게 들리지는 않지만, 어쨌든 기왕에 이 길로 들어선 사람들이라면 경쟁에서 이기고 싶어할 것이다. 하지만, 소프트웨어 개발은 경쟁보다는 협력을 요구하는 일이다. 개인 간의 경쟁은 의미가 없다. 같은 회사의 경계 내에 있는 팀 간의 경쟁은 어느 정도 용납은 될 수 있지만, 그렇다고 자원의 중복 투자가 발생하는 것은 경계의 대상이다. 고객과의 경쟁은 있을 수 없는 일이며, 고객의 만족이 회사가 돈을 버는 유일한 수단이다. 따라서, 고객과의 협력도 중요한 경쟁력의 원천이다. 경쟁보다는 성공을 목표로 삶는 것이 좋다. 누가 누군가를 이기고 올라서야 하는 것이 아니라, 함께 성공을 위해서 노력하는 것이 핵심이다. 협력하는 사람들의 성공에 대한 기준은 단 하나다. 둘 다 만족할 수 있는 수준에서의 가치를 만들어 내는 것이다. 경쟁에서는 둘 간의 순위를 정하고 그에 따르는 가치를 나누어 가져가지만, 협력은 승자독식이나 패자의 독박 같은 것은 없다.

좋은 소프트웨어 개발자가 되기 위해서는 먼저 사람의 가치를 알아야 한다. “사람을 존중하는 것”이 첫걸음이다. 기술을 이해하는 것도 중요하지만 기술이전에 사람이 있다는 것을 잊어선 안 된다. 좋은 개발자나 좋은 관리자는 어떻게 사람을 다루는가에 의존적이다. 모든 일에는 항상 사람이 가장 중요한 성공 요소다. 물론 기술도 중요하다. 쌓아놓은 기술이 없는 사람 밑에서는 좋은 기술자들이 배출되기 힘들다. 따라서, 좋은 개발자가 되기 위해서는 성숙할 수 있는 토대를 제공해 줄 수 있는 사람이 되어야 한다. 도메인을 떠나서 가장 중요한 기술적인 토대는 "좋은 소프트웨어"를 개발할 수 있는 몸에 배인 습관을 만드는 것이다. 또한, 좋은 소프트웨어를 만들기 위해서는 대화하는 방법도 익혀야 한다. 자신만 아는 단어들이 아닌 만들어진 제품을 사용하게 될 사람들이나 같이 개발에 참여한 개발자들이 공유하는 언어로 설명할 수 있어야 한다. 당장의 편의보다는 고생스럽지만 기본적인 것들에 대한 정확한 정의도 알고 있어야 할 것이다. 기본기는 아무리 갈고 닦아도 항상 부족함을 느끼게 되며, 누구보다 더 많이 연습해야 실전에서 한 번에 끝낼 수 있기 때문이다.

세상의 가치있는 일들은 화려한 것들이 아니다. 진정한 가치를 전달하기 위해서는 많은 사소한 것들이 필요하며, 사소하지만 중요한 것들이 만족되어야 비로소 빛을 보게 된다. 백조가 물 위를 우아하게 떠다닐 수 있는 것도 물속에서 쉼 없이 휘젓는 갈퀴가 있는 것처럼, 중요한 것을 이루기 위해서는 바탕이 되는 사소하지만 중요한 것들을 꾸준히 연습할 필요가 있다. 장인들의 손길이 묻어난 작품은 세세한 부분들이 전체와 통일성을 유지하기에 높은 가치를 만들어 낸다. 단순한 선 하나도 수 많은 연습을 통해서 만들어지기 때문이다. 단순한 코드 한 줄에도 그런 장인 정신이 녹아있을 수 있다. 세상의 모든 프로그램은 언어에서 제공하는 단어들의 조합에서 그치는 것이 결코 아니다. 단어와 단어들 사이의 연관 관계와 그것이 거기에 있도록 한 상황(Context), 그리고, 전체적인 시스템의 조화를 통해서 올바른 자리에 찾는 것이다. 따라서, 코딩은 단순히 알고리즘을 컴퓨터가 이해하는 언어로 옮겨놓는 것이 아니라, 알고리즘에 생명을 불어넣어 전체 시스템의 가치를 높이는 활동이어야 한다. 코딩은 검증을 포함한 시스템을 조금씩 더 높은 수준으로 이끌어 주는 중요한 일인 것이다.

좋은 소프트웨어 개발자가 되고 싶다면 먼저 자신을 행복하게 해줄 수 있는 것을 직접 손으로 만들어 보아야 한다. 일단 자신이 만족해야 남도 만족시킬 수 있기 때문이다. 최종 목표는 자신이 만족하는 것에서 끝나지 않고 남도 만족하는 코드를 만드는 것이다. 혹시 "내가 만족하는데 무슨 상관이야"라고 이야기할 수도 있을 것이다. 하지만, 당신은 결코 혼자서 일하는 것이 아니다. 만족시켜야 하는 사람들도 반드시 제품의 실제 사용자만 있는 것도 아니다. 내부 고객들도 당신의 코드를 기다리고 있으며, 당신 또한 그들의 작품을 기다리는 경우도 있다. 소프트웨어는 당신만의 작품이 아닌 모두의 결과물이기 때문이다. 남을 존중하지 않는 사람은 절대 좋은 코드를 만들 수 없다. 그냥 혼자서 자신만의 만족을 위한 코드를 짜고 있을 뿐이다. 돈을 버는 것이 인생의 주요 목표이라면 아마도 그것은 가장 낮은 수준의 목표일지도 모른다. 당신이 생계를 위한 코딩을 한다면, 당신의 기초적인 "생존권"이 보장되지 못한 상황일 것이다. 조금 더 높은 수준을 목적으로 코딩하고 있다면, 일단은 충분한 생존권은 보장받은 상태일 것이다. 다음의 목표는 당신을 남에게 알리고자 할 것이고, 최종적으로 당신 스스로가 진리를 찾아나서는 탐구를 즐기게 될 것이다. 그리고, 남을 만족시키는 것이 결과적으로 스스로를 만족시키는 시작이라는 것을 알게 될 것이다.

# [ 회사에서의 교육 ]

팀은 항상 팀원들의 능력이 향상될 수 있는 환경을 만들어야 한다. 팀원들 하나 하나가 역량이 향상되어야 팀의 수준도 올라간다. 소프트웨어도 마찬가지다. 가장 약한 고리가 전체 시스템의 품질 수준이 되기 때문에, 팀원 전체가 골고루 수준이 높지 않으면 팀이 만드는 제품 또한 수준이 낮을 수밖에 없다. 팀원들이 세미나를 하겠다고 이야기한다면, 제대로 된 팀장이라면 그런 분위기를 자신 스스로 미리 만들지 못했음을 탓해야 한다. 하지만, 어떤 팀장의 경우는 세미나를 일로 보지 않고 바쁘지 않아서 할 수 있는 소일꺼리 정도로 여기도 한다. 세미나나 교육은 일이 없어서 하는 것이 아니라, 일을 더 잘 하기 위해서 하는 것이다. 이런 상황이 지속된다면 회사의 게시판에는 교육을 보내달라고 요구하는 목소리가 커진다. 그리고, 교육에 참석했을 때 회사 업무를 보지 못하도록 외부교육으로 잡아달라고 할 것이다. 팀원들은 실제로 교육을 받고 싶어하지만, 잠시 회사를 떠나 삶의 여유를 찾고 싶은 것이다. 교육의 내용도 중요하지만 바쁜 회사 생활에서 자신을 되돌아볼 수 있는 시간도 필요하다.

부족한 것을 채우려고 하는 것은 인간의 기본적인 욕구다. 돈이 부족하면 돈을 더 벌기를 원하고, 지식이 부족하면 공부할 수 있는 시간을 요구한다. 시간이 부족하다고 느낄 때는 어떨까? 시간은 부족하지만 절대 더 만들 수는 없다. 하지만, 여기도 속임수는 있다. 시간이 부족하다고 생각하는 대부분 이유는 구현보다는 테스트에 있다는 점이다. 따라서, 테스트할 시간을 줄이는 방법을 제대로 찾는다면 좀 더 여유를 가지고 일할 수 있다. 물론 이것은 이론적인 이야기가 아니다. 학교에서는 가르치지 않지만 실무에서는 반드시 배워야 하는 것은 코드를 테스트할 수 있도록 만드는 방법이다. 회사에서의 교육은 주로 실무에서 바로 적용할 수 있는 테크닉에 집중한다. 물론 이론적인 교육도 중요하다고 이야기는 하지만, 실제 업무와 연관을 가지지 않으면 교육의 효과는 오래가지 못하기 때문이다. 중요한 것은 개발자의 부족한 시간을 벌수있는 방법이 급하게 필요하다는 점이다.

코드는 짧아야 한다. 이것도 교육이 필요한 부분이다. 대부분 개발자는 코드를 길게 짜는 것에 익숙하지만, 짧게는 만들지는 못하는 이상한 버릇이 있다. 그리고, 길게 짠 코드를 수정하라고 하면 대체로 긍정적인 반응보다는 부정적인 답변들이 먼저 날라온다. 학교에서 배우는 알고리즘들은 대부분 길이에 상관없는 예제로 나온다. 하지만, 실무 코드는 짧고 간단해야 하며, 한 가지 일에만 집중해야 한다. 그리고, 대부분 코드는 복잡한 알고리즘과는 상관이 없어야 한다. 그런 복잡한 알고리즘들은 잘 정리되어 소프트웨어의 한 귀퉁이에 포장해서 보관해야지, 코드의 여기 저기에 퍼져있으면 수정하기 어려운 난해한 문제로 돌변한다. 교육이 필요하다면 좋은 코드를 만드는 방법을 듣는 것이 현실적이다. 그것이 가장 효과적으로 개발자의 고민을 해결해 주는 근본적인 치유책이 될 수 있기 때문이다. 우리가 힘들어하고 배움에 목말라하는 것은 결국 우리의 인생이 낭비된다는 느낌이 들기 때문이다. 만약 더 생산적이고 더 재미있는 일을 한다면 과연 교육이라는 것이 필요할까? 아마 자신이 필요한 것들은 이미 배웠을 것이다.

개발자 교육은 당장에 써먹을 수 있는 것들이어야 한다. 그리고, 다루어야 할 것은 개발자들이 시간을 벌기 위한 도구들이다. 도구에는 직접적인 생산성에 도움이 것들도 있겠지만, 도구에 숨어 있는 의미를 찾는 것도 중요하다. 가장 큰 효과를 볼 수 있는 방법은 직접적인 실습을 동반한 코드의 개선에 대한 것이다. 단시간에 깊은 도메인 지식을 가르치는 것은 불가능하다. 가능하다면 오히려 그것이 이상할 뿐이다. 팀 리더를 위한 교육도 해야 한다. 팀 리더들은 팀을 관리하는 방법을 익히지 못한 상태에서 실무 개발자들과 충돌하는 경우가 잦기 때문이다. 현장에서 마주치는 많은 불합리한 것들을 개선해야 할 의무가 그들에게 있으며, 일선에서 개발자들을 이끄는 사람도 바로 그들이기 때문이다. 소프트웨어를 개발하는 프로세스 교육은 어떤 문서를 만들어야 하고, 어떻게 써야 하는지, 그리고, 어떤 조건이어야 단계의 통과 조건을 만족시킬 수 있는지"에 그쳐서는 안된다. 개발자들이 어떻게 코드의 품질을 개선할 수 있으며, 자신의 시간을 효율적으로 사용할 수 있는지 방법을 알려줄 수 있어야 한다. 교육을 받는 사람이 소비자라고 한다면, 소비자가 가질 수 있는 가치를 극대화하는 것이 교육의 목표이다. 따라서, 개발자들이 원하는 것을 제대로 전달할 수 있어야 목표를 달성할 수 있다.

# [ 처음 누구를 만나느냐가 중요하다. ]

모든 관계는 만남에서 출발한다. 사회에 첫발을 내딛는 순간 누구를 만나느냐가 남은 개발자 인생을 결정할 수도 있다. 이유는 단순하다. 만나는 사람들부터 배우게 되는 것들이 앞으로 회사 생활을 어떻게 할 것인가를 결정지을 수 있기 때문이다. 처음 회사를 다니게 되면 새로운 일에 대해서 아무것도 모르는 상태이며, 일을 가르쳐주는 사람과 같이 보내는 시간이 가족과 함께하는 시간보다 길어지게 된다. 누군가와 이렇게 오래동안 같이 해본 경험은 아마도 없었을 것이다. 회사는 "또 하나의 가족" 개념을 가질 수 있는 곳이기에, 어떤 사람들을 만나고 어떤 사람들 속에서 지내느냐는 마치 완전히 다른 문화에 적응하는 것만큼이나 어려운 일이다. 사용하는 단어는 같을 수도 있지만, 그것이 의미하는 뜻은 달라질 수 있으며, 한 번의 실수는 용납되지만 반복된 실수는 실력으로 치부될 수도 있다. 신용을 얻기는 어렵지만 잃어버리는 것은 순간의 말 실수가 될 수도 있다. 하지만, 우리는 누구를 만나는 것이 두려움이라고 보기보다 흥미롭게다고 처음에는 생각한다. 아직은 "문화적 충격(Culture Shock)"의 첫 단계를 거치고 있기 때문이다.

직장 생활에서 처음 만나는 선배는 절대적인 존재다. 그로부터 모든 것을 배우기 때문이다. 물론 자신의 전공도 일을 위한 중요한 밑거름이 될 수 있겠지만, 협업이라는 것을 배워야만 하고 학교에서는 해보지 않은 일들을 몸으로 익혀야 한다. 이미 내부적으로 정해진 규칙들도 따라야 하기에 익숙함보다는 몸에 맞지 않은 옷을 입은 것처럼 어색하게만 보일 것이다. 좋은 선배는 그런 헝클어진 옷깃을 매만져서 한 사람의 독립된 존재로 이끌어주는 사람이다. 하지만, 선배는 새로운 후배를 만날 준비를 제대로 했을까? 선배는 우리가 선택해서 만날 수 있는 사람이 아니며, 선배 역시 우리를 선택해서 만나는 것도 아니다. 그도 준비되지 않은 순간에 우리를 만나고 있을지도 모른다. 한 가지 다른 점이 있다면 좀 더 먼저 팀에 적응했다는 점이다. 그도 실수했을 것이고, 그도 누군가의 밑에서 배웠을 것이다. 제대로 배웠다면 제대로 가르치겠지만, 제대로 배울 수 있는 사람을 만나지 못했다면 제대로 전해줄 것도 없을 것이다. 기술은 세대를 거치면서 새로운 것이 오래된 것을 대체하며 발전한다. 하지만, 문화적인 부분은 팀에 고유한 것으로 팀이 존재하고 그것을 영위하는 사람들이 있다면 쉽게 변화하지 않는다.

좋은 팀에 들어가는 것은 당신의 선택이 아니지만, 좋은 팀을 만드는 것은 당신의 책임이다. 좋은 팀이란 "개인의 역량을 충분히 발휘할 수 있도록 환경을 제공하는 팀"이다. 개인의 역량을 충분히 발휘하도록 도와주는 팀이란 팀원들이 스스로가 목표를 정하고 실행하는 조직이다. 목표를 주기 보다는 목표를 설정하는 데 익숙하며, 필요 없는 낭비는 철저히 제거해 가는 팀이다. 그들에게 있어서 최고의 가치는 성과를 꾸준히 내는 것이지 단기간의 성과로 평가받고 싶어하지는 않는다. 짧은 시간에 좋은 성과를 내기 위해서는 무리가 따른다는 것을 이미 충분히 경험으로 알고 있기 때문이다. 장기적으로 유지될 수 있는 토대를 쌓는 데 집중하며, 기본이 팀 구성원들의 역량 향상에 있음을 말을 안 해도 분위기로 보여줄 것이다. 지금 그런 팀에 속해서 일하고 있다면 별다른 어려움을 못 느낄 것이다. 마치 물이 흐르듯 모든 것이 자연스럽기 때문이다. "낭비"란 자연스럽지 못한 것이 있을 때 생기는 것이기에, 제거되었다면 불편함을 못 느끼는 것이 당연하다. 좋은 팀이 스스로 생명력을 얻는 것은 부단히 낭비를 줄이려고 노력했기 때문이다.

비효율을 의식적으로 찾아내지 않아도 막힘이 보이는 곳이 근원이다. 그런 것들을 줄이려고 노력하지 않는다면 전체적인 시스템이 비효율을 경험하게 된다. 막힘은 곳곳에서 생길 것이고 어디가 해결해야 할 곳인지도 분명하지 않게 된다. "부익부 빈익빈"이란 이런 상황을 두고 이야기하는지도 모른다. 안되는 조직은 어떤 일을 해도 안 된다고 느낄 것이다. 하지만, 중요한 것은 문제의 근원이 "비대한 프로세스"나 "잘못된 의사결정"에 상관없이 조직의 실무를 담당하는 사람의 책임은 아니다. 해결하지 못한 문제들이 지속해서 쌓여서 개인으로까지 이어졌기 때문이다. 성급한 결정을 내린 사람들, 급하다고 줄여버린 일정들, 생략된 리뷰들, 줄어든 단계를 강요한 사람들의 책임이다. 줄인다고 줄어드는 일은 이미 가치가 없는 것들이다. 가치가 있는 일들은 기본부터 다르다. 기본을 가꾸는 사람들이 모인 조직이라면, 책임을 강요하지 않더라도 스스로가 인정하게 된다. 신중한 결정을 내릴 것이고 불가능한 일정에 대해서 주의를 환기시킬 것이다. 당신이 어떤 조직에 속하든 당신의 책임은 아니지만, 최소한 당신이 속한 조직이 오늘보다는 내일을 위해서 살도록 만드는 것은 이제부터 당신의 책임이다.

# [ 무엇이 되고 싶은가? 무엇을 하고 싶은가? ]

목표가 있는 사람들은 행동에서 차이가 난다. 그들의 일과는 일정한 계획으로 움직이고 있으며, 일주일이건 한달이건 상관없이 꾸준하게 일한다. 절대 튀는 일이 없이 항상 일정한 습관을 보인다. 출근하는 시간이나 퇴근하는 시간도 일정하다. 오랫동안 일하는 것이 아니라 지속해서 리듬을 유지하면서 일한다. 그리고, 그런 사람들이 성과를 낸다. 학교에서 시험을 치는 것처럼 단시간에 밤을 새워서 공부하는 것은 회사와 같은 곳에서 하는 일에는 적합하지 않다. 대부분 과제들은 개발자 개인의 꾸준함을 요구하는 일이 많다. 따라서, 사람을 뽑을 때면 행동이나 성격 유형을 잘 파악하고 과제에 투입해야 한다. 과제는 협업의 산물이기에 누군가의 튀는 행동이 팀워크를 해칠 수 있다. 능력이 출중한 사람이지만 잘 어울리지 못한다면, 그 사람에게 적합한 일을 따로 찾아주는 것도 한가지 방법이다. 무조건 팀에 융화되어 개성을 상실하라는 것이 아니라, 개성을 인정받더라도 다른 사람과의 협업이 우선이라는 말이다.

소프트웨어 개발자로 10년을 일했다고 하자. 어느 날 뒤돌아본 자신의 흘러간 인생에 충분히 만족하는가? 만족할 수 있는 삶을 살았다면, 그 삶의 바탕에는 작지만 꾸준한 개선을 이루면서 살았다는 뜻일 것이다. 한 번에 모든 것을 성취할 수 없다는 것을 알기에 하루 하루의 삶이 작은 개선의 연속이었을 것이다. 무엇을 하고 싶냐를 물었을 때 자신 있게 대답할 수 있는 사람이 얼마나 될까? 결국은 무엇이 되어, 하고 싶은 무엇을 하는 것이 목표가 아닐까? 그리고, 지금 하는 일이 그것을 위한 것인지 아닌지를 고민하는 것이 당연한 것이 아닐까? 그냥 막연히 주어지는 일을 한다고 그 목표에 조금씩 다가가는 것은 아니다. 자신이 하고 싶은 일을 찾아갈 수 있는 사람이 되어야 한다. 스스로 찾지 못하고 남이 주는 일만 하다가는 무엇이 될지도 남이 결정하게 된다. 무엇을 할지를 알지 못한다면 무엇을 해도 상관없는 사람이 되고만다. 그렇게 하루 하루를 보내다 보면 남는 것은 아무것도 없다. 특별할 것 없는 하루 하루를 살고 있을 것이고, 특별한 하루를 기대하지도 않게 된다. 그냥 무사한 하루가 되기를 바랄 뿐이다. 모든 것이 무사히 잘 되고 있다면 만족하는가? 혹시 무사하다는 말속에 변화하지 못한다는 함정이 숨겨졌다는 것을 깨닫지 못하고 있는 것은 아닐까?

스스로 질문하고 스스로 답해야 한다. 자신이 무엇을 해야 할지를 판단하는 것은 남이 아닌 자기 자신이어야 한다. 그렇게 되기 위해서는 자신만의 원칙을 만들어야 한다. 원칙이 없는 판단은 상황의 논리만을 따라가게 되며, 잘못된 결정을 내렸을 때는 남만 탓하게 된다. 다른 사람들은 당신의 잘못을 뒤집어쓰기 위해서 있는 것이 아니다. 그들도 그들 나름의 최선을 다하고자 노력하는 사람들이다. 당신 스스로가 질문에 대해 대답을 하지 못하는데, 그들에게 뭔가를 지시하거나 그들로부터 동의를 끌어내는 것이 가능할까? 설득은 자신부터 해야 한다. 그렇지 못하다면 자신감이 없는 결정과 어정쩡한 태도밖에는 남지 않는다. 모든 데이터가 훌륭하다고 해도 결정은 사람이 내린다. 그 결정을 내리는 사람들은 가장 중요한 순간이면 "감"을 가지고 판단한다. "감"이란 어떤 것을 해야 할 동기가 충분한지를 느끼는 개인적인 감정이다. "그럼에도 불구하고" 해야 할 일을 해야 한다는 느낌이다. 실패할 줄 알고 시작하는 일은 없다. 실패하는 일은 시작부터 실패하는 것이다. 그리고, 자신의 질문에 대한 답을 구하지 못했기에 잘못된 시작을 하는 것이다. 성공에 대한 "감"을 키우지 못해서 후회하게 되는 것이다.

성공에 대한 정의는 다를 수는 있다. 실패해도 성공하는 방법은 있다. 즉, 의도된 실패는 성공의 기반이 되는 것이다. 그럼 왜 실패할 줄 알면서 과제를 시작하는 것일까? 그것은 아무도 그것을 해본적이 없기 때문에 하는 것이다. 누구도 경험하지 못했기에 하고 싶어지는 것이다. 그리고, 자신이 그 누군가가 되고 싶어서 하는 것이다. 실패에서 해답을 찾고, 다시 더 좋은 무엇이 되고 싶어지기 때문에 어려운 싸움을 시작하는 것이다. 주변의 도움이 단절되고 가용할 자원이 마른 상황에서도 그것이 "올바른 길"이라고 믿기 때문에 같이 떠날 수 있는 것이다. 비록 성공의 정의가 다르기는 하겠지만 누군가는 구정물에 손과 발을 담그고 정리를 해줘야 한다. 더러움이 옷에 묻을지는 몰라도 마음에는 묻지 않는다. 성공적인 실패를 했다면, 그 경험은 누구라도 뺏어가지 못한다. 그것은 당신이 찾아낸 소중한 보석이며 힘겨운 시간을 함께 공유한 사람들의 몫이다. 스스로 질문하고 대답하면서 찾아낸 "정도"이며, 지름길은 아니지만 답에 이르게 해주는 반드시 가야 할 길이다. 후회는 실패 속에 물들어 있을 때 생기는 사치다. 스스로 무엇이 되고 싶고 어떻게 그것이 될 수 있는지를 답할 수만 있다면, 이미 마음속에서는 다음을 준비하고 있을 것이다. 다음 번에는 "더 멋진 실패"를 경험하고 싶어질 것이다. 그리고, 그 실패는 다른 사람들에게는 성공이었다고 기억될 것이다.

# [ 하고 싶은 일을 하자. ]

사람들은 이렇게 말한다. "세상에서 하고 싶은 일만 하고 사는 사람은 없어." 하지만, 하고 싶지 않은 일만 하고 산다면, 자신을 위해서 사는 것이 아니라 가족이나 다른 책임감을 주는 사람들을 위해서 사는 것일 뿐이다. 정작 자신이 하고 싶은 일을 하려는 이유는 그런 것들이 아닐 수 있다. 만약 회사에 다니면서 자신이 흥미 있는 일을 찾고 그것을 하겠다고 지원한다면 어떨까? 대부분의 경우 관리자는 수동적인 사람보다 능동적인 사람을 좋아한다. 물론, 그 사람이 하겠다는 일이 타당하기만 하다면 지적능력이 있는 사람이라면 자원하기 전에 그것이 무슨 일인지는 잘 파악하고 있을 것이다. 그리고, 나름의 분석도 충분히 했을 것이다. 완전히 새로운 생각을 가져야 하는 과제라면 설득하기 어려울 수도 있겠지만, 그렇다고 한들 그것이 무슨 상관인가? 도움을 받을 수 없다면 혼자서라도 할 수 있다. 마음을 먹었다면 혼자서 남들이 도와주지 않더라도 할 수 있는 방법을 찾을 것이다. 그 방법을 찾아가는 과정 또한 중요하다. 우리는 길을 갈 때 길만 보고 가지는 않는다. 기억하기 위해서 주변도 보게 되며, 무엇이 어디에 있는지도 기억하게 된다. 다시 그 길을 가야 할 경우를 대비해서 더 효과적인 방법을 스스로 계산하는 것이다.

물론 하고 싶은 일만 하고는 살 수 없다. 소위 말하는 팀이 있고, 팀에서 해야 할 일이 있기 때문이다. 회사에서 시키는 일도 있고, 주기적으로 보고해야 할 일도 있다. 이런 것들은 회사 생활을 하는 모든 사람이 공통으로 가지고 있는 일들이며, 하고 싶어서 하는 일이 아닌 그냥 하는 일들이다. 하고 싶은 일이란 자신의 경력으로 만들 수 있는 일이다. 즉, 과제화시켜서 할 수 있는 일이다. 개인적으로 혼자 공부를 하는 것도 좋겠지만, 더 좋은 것은 과제 자체가 자신이 하고 싶은 일이라면 최적이다. 하고 싶은 일을 한다면 자신이 생각할 수 있는 최선이라는 것을 발휘할 것이다. 모든 것이 그것에 맞춰지고 다른 중요하지 않은 일들은 그 외의 기타 등등으로 나누어질 것이다. 성과는 그런 상태에서 나온다. 하고 싶지 않은 일을 하게 될 때는 최소한의 요구만 만족시키려고 한다. 요구를 만족시켰다고 생각되면 더 이상 추가적인 일을 하지 않는다. 시간이 부족하기도 하지만, 그 일로 인해서 자신이 발전한다는 생각을 하지 않기 때문이다. 그냥 그 상태로 만족할 뿐이다. 이런 상태가 지속되면 개인은 물론 조직도 문제가 생긴다. 기술적인 해결책은 점차 복잡해지고, 이를 해결하기 위해서는 지속적인 "기술적인 빚(Technical Debt)"을 해결해야 할 필요가 있기 때문이다. 단순히 기본적인 요구만 만족시켜서는 안 된다는 것이다.

하고 싶은 일이 있다면 좋은 현상이다. 하고 싶은 일이 없는 것과는 차원이 다르다. 하고 싶은 일이 있는 사람은 세부적인 방법을 스스로 찾는다. 일이 주어지는 것이 아니라 일을 만들어가기 때문이다. 일을 만드는 데 필요한 것들을 찾고, 그것에 필요한 일정계획을 수립하고, 중요한 일을 먼저 하기를 원할 것이다. 이것은 자연스러운 과정이다. 지금까지 그렇게 하지 않았다면, 이제는 새로운 출발을 해야 할 순간이다. 아무도 당신의 인생을 책임져주지 않는다. 새로운 일을 찾을 경우(이직하게 되는 경우)에도 마찬가지다. 스스로 자신이 자격 있음을 증명하지 못한다면 누구도 당신을 고용하지 않는다. 경력에 도움이 되지 않는다고 모든 일을 하지 말라는 것이 아니다. 만약 하기 싫은 일을 해야 한다면, 비록 그것이 자신에게 주는 가치가 적다고 하더라도 열심히 하라는 말이다. 그 속에서도 자신이 하고 싶은 것을 찾을 수 있다면 다행이지만, 그렇지 않다고 하더라도 인생을 낭비하는 것은 옳지 않다. 상세한 것을 자세히 알기 전까지는 배우는 것을 멈추지 않아야 한다. 가장 기본적인 것에 대해서 자신감이 생기지 않는다면, 더 아래로 내려가서 찾아야 할 것이다. 단지 피상적인 겉핥기만 할 것이라면 하지 않느니만 못하다. 아까운 인생을 그런 곳에 낭비할 필요는 없다.

세상은 생각보다 더 넓다. 일을 찾는 사람은 세상을 보지 않고 자신만을 본다. 자신이 할 수 있다고 믿는 일만 보기 때문이다. 하지만, 자신이 할 수 있다고 믿는 것들도 처음에는 해보지 않은 일이다. 그런 세상은 극히 좁은 곳일 뿐이다. 물론 그것이 당신이 하고 싶은 일을 하려는 조건이 될 경우에는 한정된 범위에서만 찾아야 하겠지만, 거의 찾기가 어려울 수도 있다. 하지만, 기본을 잘 갖추고 있는 사람이라면 일이 바뀌더라도 새로운 경험에 대해서 유연하게 적응할 수 있을 것이다. 적응하지 못하면 도태되기 마련이다. 적응하기 위해서는 "생존"부터 시작해야 한다. 그 위에 "남들의 인정"이나 "자아실현" 등이 가능하다. 생존하기 위해서는 "기본기"가 필수다. 살아가는 가장 "원시적인(Primitive)" 형태의 노력이 있어야 한다. 따라서, 자신이 원하는 일을 하고 싶다면 기본기부터 길러야 한다. 축구는 "기술, 스피드, 체력"이 기본기이고, 소프트웨어 개발은 "코딩, 테스트, 원리"가 핵심이다. 그것도 익히지 않고서 뭔가를 찾고자 한다면, 자신이 좋아하는 일을 찾기도 전에 "부족함"만을 깨닫게 될 것이다. 그리고, 부족함은 또한 채우기 위해서 있는 것이다. 새로운 것을 하기 위해서는 "비워야" 한다는 것도 알 것이다. 하지만, 비운다고 통째로 잔을 던져버리지는 않는다. 그 잔이 바로 기본기다. 무언가를 채우고 형태를 이루기 위해서 기본적으로 갖추어야 할 것들이 있다. 힘들지만 잔을 만들고 채우는 것은 자신의 재미를 찾아가는 즐거운 여정이 될 수 있을 것이다.

# [ 견제와 균형 ]

권력은 분리되어야 한다. 이것은 정치나 법에만 한정된 것이 아니다. 일을 제대로 하기 위해서는 일을 기획하는 사람과 일을 진행하는 사람, 일이 제대로 되었는지를 확인하는 사람이 나누어져 있어야 한다. 작은 회사라면 이것이 중복될 수도 있지만, 어느 정도 규모를 갖추게 되면 각각의 역할 분화가 일어나게 된다. 원리는 견제를 통한 힘의 균형을 이루는 것이다. 모든 권력이 한 사람에게만 집중된다면 문제가 있는 일이라도 이슈(Issue)를 제기하지 못하게 되며, 결국은 그 피해가 전체 조직에 영향을 주게 된다. 하지만, 사람이란 모든 권력을 독점하고 싶어하는 것도 인지상정이다. 독점함으로써 좀 더 빠른 의사결정을 내릴 수 있다는 장점이 있지만, 잘못된 판단을 내렸을 경우 모두에게 심각한 피해를 낳기도 한다. 견제가 없는 상황에서는 어떤 조직이 다른 조직보다 우선하게 되며, 전체 조직은 그 조직의 하부 역할만 수행하게 된다. 각자의 업무를 하는 것이 아니라, 모든 업무는 하부조직이 도맡아서 하게 된다. 우선하는 조직은 보고 받기를 원하고 명령을 전달하는 임무만 수행할 것이다.

회사의 임원들은 자신의 조직이 커지기를 원하지 줄어드는 것을 보고 싶어하지 않는다. 더 큰 조직을 맡게 되면 그만큼 더 많은 책임과 권한을 가지게 되는 것이 일반적이다. 즉, 조직의 크기와 권력이 비례한다고 볼 수 있다. 하지만, 문제는 큰 조직은 의사결정 과정이 쉽지 않다는 점이다. 조직이 커지면 전문화하는 경향이 있지만, 조직 내부의 전문화가 진행될수록 벽은 더 높아진다. 점차 조직 내부의 원활한 의사소통과 인력의 이동이 어려워진다. 전문화가 나쁜 것이 아니라 그것으로 인한 의사소통의 단절이 문제가 되는 것이다. 대부분의 좋은 소프트웨어를 만드는 회사들은 작은 조직이 성과가 높다는 사실을 알고 있다. 처음에는 높은 품질과 고객이 원하는 기능을 가진 소프트웨어를 빨리 시장에 출시하지만, 회사가 커갈수록 속도가 점점 느려진다. 전문화된 조직 내부의 의사소통의 통로가 길어지고, 결과물의 이동이 여러 단계를 거치기에 중요한 정보가 전달과정에서 소실되는 경우도 많다. 이렇다 보니 더 많은 정보를 더 정확하게 전달할 필요가 생기게되고, 보완하는 방법으로 대화보다는 문서에 의존하게 된다. 하지만, 문서는 모든 정보를 전달하기에는 부족한 도구이며, 작성된 내용의 의미도 전달하는 과정에서 다르게 해석될 가능성이 있다. 조직이 커져서 힘을 모을 수는 있지만, 비효율이 파고들 가능성도 함께 커지는 것이다.

부서간에 서로 견제한다고 비효율적이라고 할 수는 없다. 견제란 일을 바라보는 관점이 다르다는 것을 의미한다. 보는 관점에 따라 해석도 달라질 수 있다. 물론 보는 관점이 다르다고 해서 본질이 달라지는 것은 아니다. 다양한 시각을 통해서 일을 제대로 하려는 것일 뿐이다. 한 덩어리로 된 조직보다는 의사결정 과정이 더 길어질 수도 있다. 충돌이 생기면 중재할 수 있는 최후의 방법은 모든 하부 조직의 명령 계통이 만나는 정점에서만 가능하다. 거기까지 올라가는 도중에 생기는 비효율은 조직 간의 의견교환 과정이라고 보는 것이 타당하다. 물론 이때도 권력의 최상위에는 한 사람만이 있다. 하지만, 기획과 개발, 검증이 서로 다른 목소리를 낼 수는 있다. 어떤 조직이 다른 조직보다 위에 서는 것은 아니며, 일방적인 지시를 내리고 듣는 것도 아니기 때문이다. 기획과 개발이 같이 있다면 개발할 수 있는 것만 기획하거나 기획이 개발을 다그칠수도 있다. 검증과 개발이 같이 있다면 품질확보에 대한 노력을 기울이지 않을 위험이 있다. 기획과 검증이 같이 있다면 계획은 잘 세울 수 있겠지만 개발은 힘들어 할 것이다.

"견제와 균형"이 이루려는 목표는 같다. "최고의 품질을 가지는 제품을 값싸게 빨리 제공해서 고객을 만족시키는 것이다." 이를 위한 협업은 가능하지만 야합해선 안 된다. 그리고, 서로의 권력(힘, 결정권)을 탐해서 시간 낭비라는 비효율을 발생시켜서도 안 된다. 목표를 위해서 필요한 조치를 하는 것은 옳지만, 단순히 자신의 권력을 나타낼 상대를 찾아서는 안 된다. 좁은 시각으로 "작은 권력"이라도 표현하고 싶다면, 당신을 바라보는 사람들의 눈보다 등을 바라보게 될 것이다. 이미 돌아선 사람들을 다시 돌아오도록 만들기는 어렵다. 돌아서기 전에 제대로 대화하는 방법을 배워야 한다. 자신의 역할이 무엇인지를 명확히 해야 하고, 때로는 역할을 넘어서는 일이라도 필요하다면 해야 한다. 일에는 누구의 역할도 아닌 경계를 연결하는 부분이 존재하며, 그런 것들도 누군가의 손으로 채워야 하기 때문이다. 그렇게 조직과 조직 사이에 존재하는 자그마한 부분들이 탄탄한 조직을 만드는 접착제와 같은 구실을 하기 때문이다. 자신의 역할만 했다고 강 건너 불구경만 한다면, 불난 곳에서 나와야 할 것들이 당신의 손에 들어오지 않을 것이다. 필요하다면 불을 끄려고 함께 노력하거나 최소한 도움을 줄 방법을 찾는 것이 옳다. 그렇다고 해서 자신의 고유한 역할을 게을리하거나 적당한 타협으로 가서는 안 된다. 지켜야 할 부분은 지키는 것이 자신의 분수를 아는 행동이다.

# [ 책임과 권한 ]

책임이 있는 곳에 권한이 있어야 한다. 권한을 주어야 책임도 따라온다. 그리고, 다른 사람의 인생을 좌우할 수 있는 자리에 있는 사람들의 책임과 권한은 비례한다. 권한이 주어지는 만큼 책임도 커진다는 뜻이다. 관리자의 자리에 오르기위해 필요한 것은 무엇일까? 일을 잘한다고 그 자리에 올라갈 수 있을까? 누군가 이런 말을 했다. "진급은 자신이 거기에 적합한 능력을 갖추고 있다는 것을 인정한 증거이지만, 그 능력이 지속될 것이라는 확신은 아니다." 우리는 자리에 맞게 자신도 변화시켜야 생존할 수 있다. 게임의 룰은 언제나 살아남을 자를 고르게 되고, 자리에 맞는 역량을 갖추지 못한 사람들은 새로운 자리를 찾아서 떠나게 된다. 물론 살아남은 자도 떠난 자도 "완전히 산 것"은 아니다. 다음의 선택이 남아 있으며 그 선택은 끝없이 이어질 것이다. 많은 사람을 관리하는 자리에 있는 사람은 자신이 하는 말이나 작은 표정 하나에도 책임을 져야 한다. 그만큼 더 조심해야 한다는 뜻이다. 별 생각 없이 던지는 말 한마디를 누군가는 평생을 지니고 살아갈 신조로 여길지도 모르며, 혹은 절대 치유되지 않을 상처가 될 수도 있기 때문이다.

다른 사람의 인생을 결정하는 것이 불가능하다고 생각할 수도 있겠지만 조직에서는 가능한 일이다. 만나는 상사나 직장 선배들을 통해서 요청받는 일이 무엇인가에 따라, 당신의 남은 인생을 그 일을 하면서 살게 될지도 모르기 때문이다. 좋은 사람을 만나서 좋은 경험을 쌓는다면, 당신은 훌륭한 인생의 멘토를 얻는 게 될 것이다. 하지만, 그런 직장 선배를 만나는 것이 노력한다고 되는 것은 결코 아니다. 인생은 그렇다. 내가 원해서 선택하지 않은 것들이 그냥 주어지는 것이다. 주어진 상황에서 최선을 다하는 것이 우리가 할 수 있는 유일한 선택이고, 그것에서 의미를 찾아야 조직에서 살아남게 된다. 팀이 빛나는 일을 한다면 그 성과는 같이 나눌 것이다. 팀이 잘못된 길로 간다면 그 책임은 관리자의 몫이 된다. 누군가의 인생은 그런 사람들이 충분히 결정할 수 있다. 그 만큼 권한을 가진 사람들은 책임감을 느껴야 한다는 의미다. 사실 절대로 남의 인생을 의도적으로 조작할 수 있는 사람은 없다. 다만 그렇다고 믿고 있을 뿐이다. 관리자란 그렇게 믿고 책임감 있는 결정을 내릴 자리에 있는 사람이다. 자신의 인생만이 걸린 일이 아니기 때문이다. 스스로의 결정에 대해서 책임감을 가지고 행동한다면, 최소한 어떤 상황에서도 믿을 수 있는 사람들을 곁에 두는 행운을 얻게 될 것이다.

권한만을 가지고 다른 사람들의 생각을 억누르려고 한다면, 책임감은 압력에 비례해서 줄어들 것이다. 책임만 강요한다면 아무것도 책임지지 않는 사람들만 곁에 남을 것이다. 권한과 책임은 같이 가는 것이지 각자 홀로 가지 않는다. 하지만, 현실에서 만나는 많은 관리자는 둘 간의 비례관계를 종종 잊어버린다. 자신의 보잘것없는 직책이 대단한 힘이 있는 듯이 말하지만, 회사에서의 관계를 벗어나면 아무것도 남는 것은 없다. 돌아서면 남남이 되고마는 것이다. 간혹 밥 한번 먹자는 연락이 올 지는 몰라도 딱히 만나서 할 말도 없을 것이다. 회사에서 만나는 사람들이 정말 "또 하나의 가족"처럼 느껴진다면, 그 사람들에게 무엇을 해주는 것이 좋을까? 아마도 자신이 알게 된 것 중에 가장 가치가 높은 것을 주려고 해야 하지 않을까? 가족과는 아무렇지 않게 그런 것들을 나눌 수 있기 때문이다. 진정으로 누군가의 인생에 영향을 주고 싶다면 소중한 가치를 "함께 해야" 한다. 함께 한다는 것은 그것을 이루기 위해서 다른 가치를 희생할 수 있다는 말이다. 자신은 모든 것을 다 가지고 아무것도 주지 않으려고 하는데, 남이 무엇인가를 내놓기만을 기대한다면 기만일 뿐이다. 같이 한다는 것은 권한뿐 아니라 책임도 같이 나눈다는 의미다. 팀의 리더가 되고 싶다면 먼저 자신의 권한부터 내려놓아야 한다. 리더는 팀을 위해서는 기꺼이 "종(Servant)"이 되어야 할 것이다. 중하게 떠 받들어 준다면 그들도 최선의 노력을 다 할 것이기 때문이다.

조직을 떠나도 관계는 이어진다. 직장 상사는 인생의 선배가 되고 함께 일한 팀원들은 아끼는 후배가 된다. 그들은 같은 장소에서 같이 시간을 보냈다고 그렇게 되는 것이 아니다. "인연"이란 단순히 같이한 시간만을 이야기하지 않기 때문이다. 강한 영향은 한 마디의 말속에도 담길 수 있으며 작은 손짓 하나로도 표현될 수 있다. "배려"하는 마음에서 나온 진심 어린 것들이 그 바탕이 된다. 관계를 만들고 그것을 확장하면서 더 깊어지는 것이다. 자신만을 위해서 산다면 자신도 바꾸지 못한다. 남을 위한다면 결국 자신이 바뀌는 것을 보게 될 것이다. 그리고, 다른 사람들도 서서히 변화를 경험하게 될 것이다. 팀원들에게 권한을 주라. 그럼 책임감을 느끼고 일할 것이다. 비록 실패를 경험하더라도 자신의 모든 것을 끌어낸 결과이기에 받아들이는 태도가 다를 것이다. 그리고, 또 다른 성장을 위한 고통의 순간이었음을 시간이 흐르면 알게 될 것이다. 관리자로서 가장 자랑스러운 것은 팀원들의 성장하는 모습을 보는 것이다. "가족"보다 더 오래 같이하는 "또 다른 가족"이기에 그 기쁨을 함께할 수 있다. 책임감을 말로만 주었다면, 이제는 실질적인 권한도 함께 찾아주도록 할 때인 것이다.

# [ 새로운 출발 ]

우린 어떤 일을 마치거나 일하던 자리를 떠나게 되면 새로운 출발을 한다고 말한다. 새로운 출발은 사실 새롭지 않은 출발이 될 가능성이 높다. 이유는 완전히 새로운 일을 하는 경우는 거의 없기 때문이다. 물론 은퇴하게 된다면 이야기는 달라진다. 은퇴는 정말 새로운 일을 찾아야 할 때다. 중간에 직장을 옮기는 것은 더 좋은 대우나 마음으로 하고 싶은 일을 찾게 되는 경우다. 하지만, 어쩔 수 없이 다니고 싶어하던 직장을 떠나기도 한다. 자신이 원하는 것은 아니지만 더 그곳에 있으면 앞이 보지 않는 상황을 만날 것이기 에상되기 때문이다. 비전(Vision)이란 개인만이 가지는 것이 아니라 회사도 가져야 한다. 회사가 비전을 제시 못 한다면 개인의 비전도 없어진다. 비전이 없는 사람들 속에 있다는 것은 과거와 현재, 미래에도 변화가 없다는 것이다. 그리고, 그런 곳이라면 자신의 미래가 자신의 앞에 서성거리는 것을 눈치챌 것이다. 그런 순간이 온다면 더 이상 변화를 늦추어서는 안 된다. 스스로가 변해야 한다. 누군가의 미래가 암울하게 보이지 않게 스스로 모범이 되어야 한다. 비단 겉모습만이 아니라 내부적으로도 충분한 역량을 갖추어야 할 것이다.

자신의 미래를 다른 사람의 바람직하지 않은 모습에서 찾는 것은 고통스러운 일이다. 지금 보고 있는 선임, 책임, 부장, 상무, 전무 등등 그런 모든 사람이 자신의 미래가 될 수도 있다. 그들은 내 눈앞에서 걸어 다니고 있으며, 만나서 밥 먹는 것도 불편한 사이일 뿐이다. 하지만, 그들이 곧 자신의 미래가 된다는 생각은 해보지 않았을 것이다. 그 조직에 있게 된다면 언젠가 자신도 그들의 일부가 될 것이다. 만약 바람직한 모습을 그들이 보인다면, 그들을 배우는 것은 즐거운 일이다. 그들이 관심이 있는 것을 하기를 원하고, 그들의 요구를 적극적으로 경청할 것이다. 그들이 나의 가장 최우선 고객이 되기 때문이다. 그럼 무엇으로 당신의 미래를 만들어가길 원하는가? 기술자라면 전공 분야에 대해서 최고가 되는 것, 대화를 주도하지만 경청하는 태도, 일을 지시하지만 일방적이지 않은 소통방법 등일 것이다. 여기에 자신이 원하는 모습을 몇 가지 더 더한다고 크게 바뀌는 것은 없을 것이다. 당신이 받고 싶을 것을 먼저 주지 않는다면 누구도 당신의 가치를 발견하지 못한다는 것을 이미 현실에서 충분히 경험했을 것이다. 시간에서 배울 수 있는 것은 인내만이 아니다. 비록 작은 것이라고 해도 오래동안 노출되면 자신도 모르는 사이에 변하게 된다. 그 변화가 좋은 것이든 나쁜 것이든 상관없이 우리는 서로 영향을 주고받고 있는 것이다.

당신의 모습을 보자. 하루 하루의 일상에서 당신은 다른 사람에게 어떤 모습으로 보일까? 호통만 치는 모습? 아니면 화를 내면서 신경질적으로 회의를 하는 모습? 주위에는 마음에 들지 않는 사람들로만 둘러싸여 있다는 생각? 마음에 들었다가 몇 번의 실수를 인해서 싫어하게 된 사람들? 그런 모든 것들은 당신의 마음이 그렇게 결정한 것이다. 그들이 스스로 그렇게 된 것이 아니다. 모든 것이 당신에게 보고해야만 집행된다면, 당신 스스로가 가장 큰 “병목지점(Bottleneck)”이 될 것이다. 믿지 않는 마음이, 존중하지 않는 말투가, 남보다 자신이 더 생각이 깊다는 우월감이 당신을 망치는 것이다. 당신은 자신을 대단하다고 생각할지도 모른다. 하지만, 그건 당신이 그 자리에 있을 수 있는 진정한 이유를 모르기 때문에 잘못된 생각일 뿐이다. 자기 일을 묵묵히 수행하며 스스로 실력을 키워가고 있는 사람들이 당신이 그 자리에 있도록 만든 소중한 존재들이다. 당신은 그들의 위에 있는 것이 아니라 그들과 함께 있는 것이다. 그들이 당신의 가장 듬직한 조력자가 되어야지, 당신이 그들을 관리한다고 생각한다면 오산이다. 그들은 당신이 관리하지 않아도 존재할 수 있는 충분한 자격이 있다는 것을 “입사”를 통해서 이미 증명했다. 당신이 있어야 그들이 존재한다고 생각한다면, 언젠가 그 믿음이 깨어지는 날 모든 관계도 함께 사라질 것이다. 계약만으로 관계를 유지한다면, 계약의 유효기간이 다하면 의무도 사라지기 때문이다. 당신이 진정으로 그들 중의 한 사람이 되기를 원한다면, 그들의 눈으로 바라보아야 한다. 당신의 눈만이 항상 진리를 찾는 절대 기준은 아니기 때문이다.

인정할 부분은 인정해야 한다. 나이가 들어서 새로운 것을 배우기도 힘들고, 변화를 주도하지는 못해도 따라잡기도 버거워한다. 과거의 경험만을 강조한다면 실무자들을 이해시키기도 어려울 것이다. 실무자가 하는 이야기를 믿어줘야 한다. 비록 그들이 거짓을 말한다고 해도 일단은 믿어야 한다. 어쨌든 가장 실무에 가까운 곳에서 일하는 사람들은 그들이다. 항상 새로운 것들은 그들의 몫이며, 한 발자국 멀리서 그들을 바라본다고 달라지는 것은 없다. 다만 그들에게 좀 더 힘을 실어주고 꾸준한 관심을 나타내는 것만으로도 충분하다. 그들이 있기에 당신이 있다. 그들과 함께하는 새로운 출발이라면, 당연히 당신은 그들을 위한 최고의 조력자가 되어야 한다. 단순히 "도와줄 것이 있으면 이야기하라"라고 말하지 말고, 담담하게 지켜봐 주는 것이 최선일지도 모른다. 그들은 많은 것을 바라지 않을 것이다. 당신 자신은 또 다른 준비를 해야 한다. 한걸음 앞에서 그들이 제대로 길을 가고 있는지 꾸준히 지켜봐야 하기 때문이다. 쉬운 것은 없다. 한번 발을 담그면 계속 가야 한다. 갈 길이 멀다면 준비도 필요하다. 그리고, 가는 도중에 새롭게 알게 되는 것들도 많을 것이다. 그 길의 종착역에 무엇이 있을지는 모르지만, 길을 가는 사람은 모든 경우를 대비해야 한다. 그 모든 대비가 부질없을지라도 그것이라도 하지 않는다면 불안감은 당신의 마음으로부터 시작될 것이다. 항상 앞을 보고 달려가지만 남겨진 것들도 무엇인지 돌아보아야 한다. 정말 그 길을 떳떳하게 잘 걸어왔는지 스스로에게 질문해야 할 것이다.

# [ 소프트웨어 개발과 인간성 ]

사람이 소프트웨어를 만든다. 아직은 그렇다. 기계가 소프트웨어를 만드는 날이 언젠가 올지는 모르지만, 아직은 사람이 소프트웨어를 만든다. 이유는 단순하다. 사람이 원하는 것은 사람만이 이해하기 때문이다. 기계는 사람이 원하는 대로 만든 코드를 실행할 뿐이다. 결국, 가장 중요한 것은 "사람"이다. 이 단순한 진리를 너무 쉽게 잊는 것도 사람이다. 개발의 현장에서는 도무지 사람이 중요하다는 것을 생각하지 않는 경우가 많다. 관리자는 개발자를 쉽게 바꿀 수 있는 부품처럼 다루기도 하고, 하루에 12시간 이상을, 일주일에 100시간을 일하게 만드는 경우도 있다. 당연히 토요일과 일요일은 없다. 이런 식으로 3년 동안 일하는 경우를 본 적도 있다. 물론, 이렇게 일해서는 제대로 일을 하지 못한다. 미안한 말이지만 일하는 시간이 늘어날수록 더 많은 버그와 더 많은 일정 지연을 보게 될 것이다. 그리고, 그로 인한 시장에서의 기회 상실로 이익은 줄어들고 비용은 늘어나게 될 것이다. 결국 이미 늦어버린 때가 되어서야 버그가 언제 발생할지도 모를 제품을 출시할 뿐이다. 다시 한번 이야기하지만 이유는 단순하다. 개발자도 사람이라는 것을 이해하지 못했기 때문이다. 그들이 사람으로 받아들여지지 않기에 기계적으로 시키는 일만 할 뿐이다. 누구로 대체되더라도 충분히 할 수 있을 정도의 일만 할 것이다.

인간성을 아는 것은 인간이 가져야 할 기본적인 존엄(가치)을 이해하는 것으로 시작할 수 있다. 인간은 기계가 아니며 주어진 불가능한 일정을 맞추지도 못한다. 불가능한 일정은 언제나 "시장의 논리"라는 이유로 타당성을 부여받지만, 그것을 만족시키지 못하는 개발팀에게는 항상 "비난"만 주어진다. 시장의 논리가 정말 타당할까? 무조건 기능이 많은 제품을 빨리 싸게 만드는 것이 최선일까? 그게 가능한 일이라면 남들도 이미 다 했을 것이다. 그것이 어렵기 때문에 도전할 만한 가치가 있다고 이야기한다면, 오히려 가능한 일정을 만들 수 있도록 미리 계획되었어야 한다. 이런 것들이 로드맵(Roadmap)으로 주어지지 않고, 그때 그때 주어진다면 어떤 개발조직도 일정을 만족시킬 수 없다. 로드맵은 그냥 주어지지 않는다. 누군가는 시장의 미래를 예측해서 필요한 자원을 미리 확보해야 한다. 하지만, 대부분 로드맵과 자원계획은 따로 논다. 이상적인 계획만을 이야기하지 현실에 대한 타당한 검토는 생략되어 있다. 목표는 제시하지만 방법은 나오지 않는다. 개발이란 생각을 현실화시키는 활동이다. 불가능한 이상을 구현하는 것이 아니다. 일정을 강요한다고 결과물이 만들어지는 것도 아니다. 오히려 앞당겨진 일정은 불완전한 제품을 만들도록 개발자에게 압박을 가하고, 그런 제품은 시장의 선택에서 멀어질 것이다. 물론 일정믈 만족하면 회사에서 포상은 받을 수 있을 것이다. 하지만, 회사에서 주는 포상이 고객을 만족시켜주는 것은 결코 아니다.

개발자는 "신뢰"받기를 원한다. 또한 신뢰받는 사람은 신뢰를 표시하는 사람을 만족시키기를 원한다. 따라서, 그들은 자신들에 대한 신뢰의 보답으로 최선을 다할 것이다. 당연히 신뢰를 지탱해 주는 것은 "존중"이다. 하지만, 일반적으로 개발자들은 존중받지 못한다. 관리자는 "고과권"을 가지고 개발자들을 위협하는 주요 도구로 사용한다. 고과는 연봉과 보너스의 기준 및 진급에 영향을 준다. 명확한 기준을 제시하지 못하는 고과는 관리자의 마음대로 개발자를 평가할 수 있도록 만든다. 실제로도 그렇다. 고과는 관리자들이 원하는 데로 주는 것이 일반적이다. 물론, 성과를 무시하지는 않겠지만, 그렇다고 성과만을 가지고도 평가하지는 않는다. 그래서, "정치"가 필요하다. 하지만, 회사는 일하는 곳이지 "정치"를 하는 곳이 아니다. 정치꾼들이 많아지면 결국 회사의 경쟁력은 사라진다. 연못이나 늪이 사라지면 그곳에 살던 동식물도 운명을 같이한다. 자신이 살고 있는 생태계가 없어진다면, 자신 또한 살아남지 못하는 것이다. 아무리 신뢰를 받는다고 해도 “비전이 없는 회사”에 남아있고 싶어하지는 않을 것이다. 하지만, "비전"을 함께 만들 수 있는 회사라면 이야기는 다르다. 지금 당장은 힘들더라도 운명을 같이하고 싶어할 것이다. 조직력이 지탱되지 않는다면 아무리 함께 일하더라도 다른 생각을 가지게 된다. 생각은 공유되지 않을 것이고 문제는 쉽게 숨겨질 수 있다. 신뢰를 만드는 것은 존중이며, 서로의 생각을 공유하는 것이 소통의 기본이다. 조직력은 그런 신뢰를 바탕으로 한 소통으로 만들어진다.

소프트웨어를 만드는 사람은 사람 사이의 기본적인 것들을 중요하게 생각한다. 그런 것들이 이루어지지 않는다면 더 이상 “위대한 꿈”을 꾸지 않게된다. 그리고, 무리하지 않는 수준에서 적당히만 일한다. 자기계발과 같은 것들도 남의 일일뿐이다. 기회가 주어지지도 않겠지만, 기회가 있더라도 할 이유를 찾지 못한다. 그냥 시키는 일만 하면 되는데 무리해서 문제가 생길지도 모를 일을 할 필요를 느끼지 않는다. 하지만, 기본적인 것들은 간과되기도 쉽다. 기본적인 것이기에 당연하다고 생각하고 무시하는지도 모른다. 정말 중요한 것은 이런 사소한 것들이다. 사람과 사람 사이의 간격이 늘어나는 것은 언제나 사소한 오해나 의도적인 작은 왜곡에서 발생한다. 나쁜 관리자들은 자신들의 의도에 맞게 남들을 교묘하게 이용하고 있지만, 스스로는 절대 그렇게 생각하지 않는다. 왜냐하면, 자신에게 도움이 되는 일이기에 남들에게도 도움이 될 것이라고 단언한다. 남들은 자신을 위해서만 존재하고, 그들의 가치는 자신을 통해서만 실현될 수 있다고 믿는다. 안타깝지만 그런 사람들은 무엇이 정말 중요한지를 깨닫지 못한다. 자신이 보고 싶은 것만 보고, 자신이 듣고 싶은 것만 듣을 뿐이다. 모든 것을 자신의 가치 기준으로 판단할 뿐인 것이다. 애시당초 다른 시각이 있을 수도 있다는 것 자체를 인정하지 않는다. 관리자는 개발자가 자신과 다른 시각을 가질 수 있고, 표현하는 방법도 다르다는 것을 충분히 이해해야 한다. 우리는 조직 일부로만 존재하는 것이 아니라 조직과 함께 존재하기에 가치를 지니는 것이다. 명령만 수행하는 존재가 아니라 조직의 가치를 능동적으로 만들어가기 때문이다.

# [ 열심히 일할수록 망한다? ]

충분히 오해를 살 수 있는 이야기를 하는지도 모르겠지만, "바쁘게 일할수록 회사는 망한다." 이 말은 바쁘다는 것을 핑계로, 혹은 정말 바쁘게 일해서는 “제대로 일할지 못한다”는 역설적인 의미다. 아이들이 숙제할 때 빨리하기를 원하는 부모는 없다. 빨리 보다는 정확하게 하기를 원한다. 하지만, 회사에서는 “정확히” 보다는 “빨리”를 더 원한다. 혹은, “정확히 빨리하기”를 원한다. 모든 결정을 빨리 내려야 하고, 그 결정과 더불어 즉시 결과물을 보기를 원한다. 하지만, “빨리 일하면 정확히 하기 힘들다.” 정확히 한다는 말은 작은 변화에 대해서 일일이 검증을 한다는 것을 의미하며, 이는 빨리 전체적인 결과물을 보기를 원하는 사람들에게는 더디게만 보일 것이다. 물론 실제로는 그렇지 않다. “정확하게 일하는 것”은 결과적으로 더 빨리 결과물을 만들어내며, 품질도 우수한 결과를 보인다는 것이다. 다만 효과가 과제의 후반으로 갈수록 더 많이 나타난다는 것과 과제의 초반과 중반까지는 개발자들이 제대로 일하고 있는 것처럼 보이지 않을지도 모른다는 점이다. 당연히 개발자는 제대로 일하고 있음이 분명하다. "진척도"보다는 "품질"에 집중하고 있기에 그렇게 보일수도 있다. 만약 관리자가 조금만 소프트웨어 개발의 본질에 대해서 이해하고 있다면, "일을 왜 더 빨리 하지 않냐?"라는 멍청한 질문은 하지 않을 것이다. 대신에 "일이 얼마나 검증되었는지"에 대한 질문을 할 것이다.

소프트웨어를 만들어본 관리자와 그렇지 않은 관리자가 있고, 만들어본 관리자도 잘 만들어본 관리자와 그렇지 못한 관리자가 있다. 첫 번째 기준은 소프트웨어 개발을 모른다는 것(해본적이 없다는 것)이고, 두 번째는 품질을 확보하는 방법을 모른다는 것이다. 소프트웨어 개발을 해보지 않은 사람들이 가지는 선입견은 소프트웨어가 "Soft"하다는 생각이다. 품질을 이해하지 못하는 사람들은 단계별로 소프트웨어를 만드는 것이 좋다고 생각한다는 것이다. 둘 다 잘못된 판단이다. 소프트웨어는 "Soft"하지 않다. 소프트웨어는 변경에 "시간"이라는 비싼 비용이 든다. 어떤 경우에는 완전히 새롭게 만드는 것보다 더 비용이 들어갈 수도 있다. 소프트웨어를 정해진 단계를 완전히 준수해서 개발하는 것은 변경이 반영될 가능성이 작다. 그리고, 피드백(Feedback)을 받는 루프(Loop)가 길어져 문제를 발견하는 시점 자체가 늦어진다. 당연히 일정이 생각보다 길어지는 일이 자주 발생하게 된다. 물론 이런 절차적인 개발과정이 필요한 부분도 있다. 하지만, 모든 소프트웨어를 그런 식으로 개발하지는 않는다. 적어도 관리자라면 자신이 개발하는 제품의 특성을 이해하고, 그것에 맞는 "개발방법"을 생각할 수는 있어야 한다. 그것 자체가 힘들다면 노력을 대신할 수 있는 사람을 찾거나, 직접 개발 담당자들에게 권한을 넘겨주어야 한다. 더 잘 할 수 있는 전문가들이 있는데도 불구하고 굳이 자신이 직접 할 필요가 없다는 말이다. 더 중요한 일이 많지 않다면 직접 공부해서 할 수도 있겠지만, 개발 실무에서 그런 관리자를 찾기는 정말 어렵다.

열심히 일하는 것은 나쁜 것이 아니다. 다만, 열심히 제대로 하려고 노력해야 하지, 열심히 바쁘게 일하기만 해서는 안 된다. 바쁘면 창조적인 생각을 할 틈이 없어지며, 혁신이나 개선이라는 좋은 말들은 뒷전으로 밀린다. 바쁘다는 것이 그런 것들의 좋은 핑게꺼리가 되어버리기 때문이다. 하지만, 회사는 항상 바쁘지만 창조적으로 일해야하고, 개선해야 한다고 이야기할 것이다. 그것은 위선이며 거짓말일 뿐이다. 그렇게 일하면 점점 더 바쁘게 되고, 결국은 일도 제대로 못 하고 개선도 제대로 못하게 된다. 둘 다 못할 바에는 하나를 포기하는 것이 더 효과적일 것이다. 당연히 개선이나 창조적인 것들은 직접적으로 돈(수익)과 연결시키기 어렵다는 이유로 개발에서 무시될 것이다. 실무 개발자에게 두 개의 과제를 동시에 “50:50”의 비율로 하라고 한다면, 제대로 과제가 진행될 수 있을까? 아마도 개발자가 일 사이를 "Context Switch"한다고 보내는 시간을 생각하면, 절대 두 개의 과제에 “50:50”의 비율이 나오지 않는다. 대략 “40:10:40”의 비율로 일 사이를 왔다 갔다하는 비용이 삽입될 것이다. 점점 더 많은 일을 동시에 하게 되면, 실제 업무에 활용되는 시간은 더 줄어들 된다. 열심히 하지만 일에 대한 효율은 악화하고, 결국에는 제대로 되는 일이 하나도 없게 된다. 제대로 하지도 못하면서 열심히 하는 태도가 문제의 근원은 아니다. 그렇게 하게 만든 사람들 자체가 “근본 원이(Root Cause)"을 제공한 것이다. 자신들이 그렇게 만들었지만 매번 결과가 왜 제 때 높은 품질로 나오지 않느냐고 다그치기만 한다. 일이 늦어지고 품질이 낮아지도록 만든 한 사람들이 적반하장격으로 무엇이 문제인지를 묻는 것이다.

열심히 일하지 않아도 성공할 수 있다면 가장 좋은 해결책이 될 것이다. 더 정확히 이야기한다면, 열심히 하는 “목표”를 바꾸어야 한다는 것이다. 열심히 하지 않는 태도는 어쨌든 보기 좋은 것은 아니다. 실제로도 관리자에게 좋은 인상을 주지 못한다. "열심히"의 정의는 제 때에 제대로 일하는 것이며, 결과물의 품질을 최고로 확보하는 것이다. 품질이 제대로 나오는 단계를 지나야 구체적인 성과를 낼 수 있다. 구체적인 성과가 있어야 관리자들의 지원을 얻을 수 있으며, 그래야지만 과제에 힘을 받을 수 있게 되는 것이다. 따라서, 구체적인 성과를 일찍 보여줄 방법을 찾아야 한다. 놀라고 하는 것이 아니라 “가시적으로 중요한 성과들을 지속해서 보여줄 방법”을 찾으라는 것이다. 그리고, 다른 사람들이 뭐라고 하든 상관하지 않을 탁월한 성과가 무엇인지를 명확히 제공할 수 있어야 한다. 그렇지 않다면 아무리 좋은 성과도 내도 관리자들은 무시할 것이다. 다른 팀들은 아예 자신들과 다르다는 이유만으로 상대도 해주지 않을 것이다. 우리가 열심히 일하지 않아서 망한다고 생각하는 관리자들은 세상의 곳곳에 널려있다. 그들은 일이 잘 안될수록 더 열심히 일하기만을 요구하고, 더 정밀한 프로세스를 도입할 것을 원한다. 그런 것들로 세상을 자신의 손바닥 안에 두기를 원하지만 불가능한 일이다. 프로세스가 정밀하고 딱딱히 굳어질수록 그것을 수행하는 실무 개발자들은 무시할 것이기 때문이다. 무시하는 게 불가능하다면 최소한만 할 것이고 양으로 승부할 것이다. 문서의 내용보다는 페이지 수가 많을수록 더 일을 잘한다고 생각한다면 그렇게 해도 상관없다. 하지만, 문서가 아무리 많더라도 제대로 동작하는 결과물을 만드는 것은 결국 개발자 손끝에서 나온다. 개선과 혁신의 출발점도 항상 그들의 여유로운 마음에서 시작한다.

# [ 책임이란? ]

책임이란 자신이 하겠다고 약속한 것을 제대로 이행하는 것을 의미한다. 책임은 누가 주는 것이 아니라 원해서 맡는 것이다. 따라서, 일방적으로 전달한 명령에 대해서는 책임이 없다. 어쩔 수 없이 맡을 수밖에 없는 분위기에서 이야기한 약속도 구속력이 없기는 마찬가지다. 강요에 의한 것은 약속이 아니라 협박일 뿐이기 때문이다. 그런 것들을 "동의"라고 부르는 관리자들은 흔하며, 자신이 듣고 싶은 것만 듣고 원하는 것만 말하는데 익숙한 사람들이다. 책임이란 반드시 상호 간의 "존중"을 바탕으로 해야 한다. 스스로 일의 범위를 정할 수 있어야 하고(물론, 회사의 비즈니스 목표에 부합하는), 그에 따르는 지원을 충분히 받을 수 있어야 한다. 대체로 일방적으로 전달되는 명령은 범위와 일정만을 이야기하고 지원해주는 것들은 별도로 없다. 따라서, 책임질 수 없는 것들을 던지는 것밖에 되지 않는다. 소프트웨어 개발자들은 이런 환경에서 일하는데 이미 충분히 익숙하다. 그리고, 그런 것들이 어떤 결과를 만들어 내는지도 경험으로 잘 알고 있을 것이다. 일의 범위, 시간, 비용, 품질이라는 것 중에서 시간과 비용, 품질이 변경되지 않는 한, 바꿀 수 있는 것은 단 한 가지 밖에 없다. "일의 범위"가 그 대상이 되는 것은 당연하다. 하지만, 이것도 사실 변경이 여의치 않다. 모든 기능이 없이는 제대로 된 제품이 아니라고 수없이 이미 이야기를 들었을 것이 분명하기 때문이다. 그리고, 만약 제품이 팔리지 않을 경우를 대비해서라도 상품기획이나 마케팅은 일정을 맞추지 못한 기능에 대해서 자신들의 책임을 개발로 전가할 것이 분명하다.

책임은 약속을 기반으로 한다. 약속은 일종의 계약으로 제대로 실행되고 있는지는 계속해서 확인해야 한다. 만약 당신이 집을 지어달라고 건축업자와 계약을 했다고 가정한다면, 계약서에 도장만 찍고 실제로 집이 어떻게 만들어지고 있는지를 한 번도 확인하지 않는다면 어떻게 될까? 아마도 나중에 그 집에 이삿짐을 가지고 들어간다면 크게 실망할지도 모른다. 당신이 계약 당시에 보았던 그럴듯한 그림속의 집이 아닐 수도 있고, 이런 저런 하자로 인해서 제대로 집에서 살지 못할 수도 있다. 따라서, 약속이란 그것을 이행할 사람과 그 결과물을 인수할 사람 사이에 꾸준한 협의가 필요하다는 것을 알 수 있다. 만약 서로 위협이 될 수 있는 부분들만 강조한다면, 상대방의 일방적인 희생을 강요하는 계약이 될 것이다. 서로가 "Win-Win"할 수 있는 협력이라는 형태의 약속(계약)이 된다면, 분명 양측은 서로의 책임을 다하려고 노력할 것이다. 그것이 양측에 다 이익이 되기 때문이다. 그렇다면 양측에 모두 이익이 되는 이런 계약이 가능할까? 물론 회사 대 회사의 관계에서는 힘들 수도 있다. 하지만, 개인 대 개인의 관계에서는 어떨까? 예를 들어, 관리자와 실무 개발자 사이에 서로 이익이 되는 약속이나 계약을 맺는 것은 어떨까? 관리자는 개발자에게 과제에 정의된 목표를 다하라고 요구할 수 있다. 개발자는 관리자에게 약속을 이해하기 위한 지원을 요청할 수 있다. 서로 이익이 될 수있는 유일한 방법은 목표한 과제를 완료하는 것이다. 어느 일방만이 요구를 하는 것이 아니라, 서로가 목표로 한 일에 대해서 약속을 이행하는 것이다. 약속 자체가 성립하지 않는다면 서로에게 책임감 있게 일하기는 힘들 것이다.

현재의 가치는 미래를 비춰볼 수 있는 거울로 삼아야 한다. 그렇다고 현재의 가치에만 집중하면 망한다. 현재의 가치는 미래로 가는 길에서 잠시 쉬어갈 수 있는 휴게소와 같은 것이다. 잠깐 머무를 생각을 해야지, 그것에만 모든 것을 쏟아붓는다면 더 이상의 큰 가치를 만들어내지 못한다. 그렇다고 미래의 가치에만 집중해도 망한다. 당장 먹고 살 꺼리가 없기 때문이다. 쉬지 않고 목표만 향해서 걸어 간다면, 결국은 지쳐서 중도에 포기할 수밖에 없다. 따라서, 현재의 제품 개발과 미래의 플랫폼 개발은 병행되어야만 한다. 대부분 중간 관리자는 플랫폼이 가지는 미래의 가치를 인정하지 않는 경향이 있거나, 상품화를 담당하고 있는 자기 일이 아니라고 보는 경우가 많다. 미래에 자신의 자리가 없을지도 모른다는 강박관념과 두려움이 플랫폼의 가치를 현재에 고정시키려고 하기 때문이다. 어쨌든 살아는 남아야 미래를 볼 수 있다고 믿는다. 결과적으로 가치의 중요성이 편향될 경우에는 실무 개발자의 "탈진(Burnout)"만 낳게 된다. 다 타버리고 재만 남은 개발자들은 인간 허수아비처럼 살아갈 뿐이다. 서로를 위한 계약이란 애시당초 존재하지 않는 상태가 되어버리고, 일방적인 명령이 난무하지만 제대로 이행되지도 못한다. 관리자는 실무 개발자의 "우산"이나 "바람막이"가 되어야지, 몰아치는 비와 바람이 되어서는 안 된다. 가장 흔한 잘못은 “같은 팀이 아니라고 느끼도록 만드는 것”이다. 마치 자신은 아무런 잘못이 없고 목표를 이루지 못한 모든 책임이 개발자에게만 있는 듯이 질문을 더지는 것이다. 잘못된 목표를 제시한 것 자체가 개발자들에게 면책권 준 것이라는 사실을 잊은 것이다. 미래의 가치를 지나치게 빨리 보고자 한다면 그만큼 충분한 자원을 미리 확보했어야 한다. 그런 성의도 없으면서 최소의 노력으로 최대의 효율을 거두려고 것은 탐욕이거나 사기일 뿐이다. 열매를 따기 위해서는 나무가 잘 자라는 환경부터 만들어야 할 것이다.

책임이 동반되지 않는 일은 최소한의 목표로 일하게 만든다. 사람은 자신의 선택권이 없는 일에는 집중하지 않는다. 협박이나 다그치는 분위기에서 억지로 떠맡은 일이 제대로 되기는 어렵다. 물론 책임감을 그런 일에도 느끼고 최선을 다한다면 상관없을 수도 있다. 실제로 많은 중간 관리자들이 그렇게 일하기도 한다. 하지만, 그것만이 최고의 선택은 아니다. 일을 많이 하면 제대로 되는 일이 없듯이, 우린 "선택과 집중"이라는 좋은 말을 이미 알고 있다. 일이 잘 안되면 여러가지 일을 할 것이 아니라 한 가지라도 제대로 해야 한다. 메뉴가 많은 식당이 큰 성공을 하지 못하듯이, 많은 제품을 만들어야 한다면 제대로 만드는 제품은 몇 가지가 안된다. 때로는 그것마저도 운에 맡길 수밖에 없을 것이다. 회사는 많은 제품을 만들어서 제품군을 다양하게 갖추려고 노력하겠지만, 그로 인해서 얻는 이익은 위험의 분산에 정도에 따라 줄어들게 된다. 그렇게라도 위험을 줄여야 할 필요도 있겠지만, 다양한 제품보다는 현재의 가용 자원에 한정해서 일해야 한다. 만약 일정을 절반으로 줄이고자 한다면, 과제의 초반부터 인력을 4배 이상 투입해야 할지도 모른다. (물론, 그렇다고 일정을 맞출 수 있다고 장담하지 못하지만) 잡화점 식으로 제품을 만들 것이 아니라면, 우리는 반드시 해야할 일을 줄이는 방법을 찾아야 한다. 일을 많이 한다고해서 항상 최고의 효율을 내는 것은 아니다. 축구 선수는 90분간 전력을 다해서 뛰지 않으며, 뛰어야 할 때를 위해서 체력을 비축한다. 마라톤 선수도 초반부터 전력으로 뛰다가는 전체 코스를 완주하지 못한다. 자신의 리듬을 알기에 조절할수도 있는 것이다. 승부를 내야할 때를 기다리며 한 순간에 능력의 최대치 끌어내는 것이다. 자신을 잘알고 자신이 원하는 바를 위해서 전력을 기울이기 때문에 가능하다. 그런 사람이라면 이미 책임이라는 단어 자체도 잊고 일과 자신을 동일시하고 있을 것이 분명하다.

# [ 더 이상 물러설 곳은 없다 ]

어디가 마지막일까? 한번 물러나기 시작하면 끝이 보이지 않는 양보를 계속해야 한다. 이런 저런 이유로 하지 못한다는 말을 들어야 했다면, 이제는 다른 방법을 생각해야 할 때다. 처음부터 새로 모든 것을 검토해야 할지도 모른다. 일을 진행하는 사람들이 여유가 없어서 중요한 것들을 하지 못하는 상황이라면, 과제 진행 자체에 근본적으로 문제가 있다는 뜻이기 때문이다. 실패란 사소한 것일 수도 있지만, 실무를 진행하는 사람들에게는 심각한 고통이 따른다. 개발자라면 자신의 능력이 부족하다는 것을 깨닫는 중요한 계기가 될 수도 있겠지만, 어쩔 수 없는 동의로 인한 좌절은 자신감에 심각한 손상을 주게된다. 개발자는 자신감이 없으면 어떤 일도 제대로 하지 못한다. 어떤 일도 적절한 추정을 하지 못하며, 단정적인 단어들은 자신의 사전에서 빠져버리게 된다. "네, 알겠습니다."라는 대답은 "이제 행동으로 옮길 준비가 되었다."가 아니라, "이제 그만 하세요."라는 의미로 해석해야 할 것이다.

과제의 책임자들은 자신에게 물어야 한다. "정말 할 수 있을까?" 물론 무턱대고 시작한 일이 좋은 결과를 내기도 한다. 그렇지만 대부분의 과제는 상식과 비상식의 조합으로 일정 계획이 수립되고, 절대 변경할 수 없는 "게임의 규칙"인 것처럼 받아들인다. 상식적인 것들은 일반적으로 일을 진행하는데 필요한 검증된 활동들이며, 비상식적인 것들은 경험 없이 추정된 수치들이다. 보고 받는 입장에서는 수치에만 치중하지 상식적인 부분들은 간단히 넘어간다. 하지만, 과제의 일정에 대한 예측이나 자원에 대한 배분 등은 말 그대로 수치일 뿐이다. 물론 계획이 없는 것보다는 계획이 있는 것이 더 좋다고 하지만, 계획을 변경되지 않는 "절대적인 것"으로 받아들인다면, 장기판 위의 모든 말에게는 부담으로 작용한다. 서로 경쟁하는 상태를 만들고 협력보다는 대립하는 관계가 된다. 서로를 믿지 못하는 상황이라면 어떻게 일을 진행할 수 있을까? 자그마한 요청 하나에도 오랜 시간 회의가 필요하고, 누구도 추가적인 일을 떠맡으려고 하지 않을 것이다. 누군가의 책임을 물어야 하는 상항이 오게 된다면, 누구도 책임지지 않기 위해서 노력할 것이다. 그리고, 그 모든 판단은 윗사람의 말 한마디로 결정된다. 모든 결정해야 할 사안들이 그 사람에게 향하게 된다. 책임지지 않기 위해서는 결정해서도 안 되기 때문이다.

중요한 일은 곧 잘 생략된다. 중요한 일이 중요하다고 생각되지 않는 것처럼 쉽게 지나가고, 누구의 기억에도 남아 있지 않게 된다. 이미 돌이킬 수 없는 순간이 되어서야 진작했어야 할 중요한 일이 빠졌다는 것을 알게하게 된다. 이 때는 과제를 더 할 것인지 말 것인지를 결정해야 하는 순간이며, 대부분의 경우는 그냥 문제가 많아도 계속하는 방향으로 결정된다. 늦은 결정이 옳지 않다는 것이 아니라, 이미 결정해야 할 마지막 순간을 놓친 후에 결정한다는 점이다. 중요한 일들은 반드시 결정해야 할 때가 있는 일들이다. 그렇지 않다면 당연히 중요한 일이 아닐 것이다. 때를 넘기면 결정이 어떤 식으로 나더라도 결과에 영향을 주지 못한다. 최후의 결정해야 할 순간을 찾는 것은 일종의 "감"에 의존할 수밖에 없다. 과제의 비상식적인 부분들은 "감"에 의존해야 할지도 모르지만, 상식적인 부분들은 이미 많이 사람들이 받아들이고 있는 것들이다. "감"은 그냥 키워지는 것이 아니라 논리적인 절차와 과정을 통해 생성되어야 한다. 일에 대한 피드백을 즉각적으로 받을 수 있다면, 어떤 일이 얼마나 걸리고 현재 어떤 수준에 도달해 있는지 대략적으로라도 파악할 수 있다. 일은 후반으로 갈수록 점점 더 상식적인 예측이 가능한 수준으로 바뀔 것이다. 구체적인 것들이 조금씩 자신의 모습을 찾아가기 때문이다. 비록 문제가 많은 과제라고 하더라도 대략적인 추이는 충분히 알 수 있다.

더는 물러설 곳이 없는 상황이란 결정을 더 이상 미룰 수 없는 때다. 하지만, 그렇다고 하더라도 구체적인 정보가 주어지는 그 순간 까지는 결정할 수 없다. 따라서, 이런 상황을 만나기 전에 구체적인 정보를 만들어야 한다. 코드의 상태를 모른다면 어떤 일을 더 해야 할지를 알 수 없다. 막연히 테스트의 결과를 기다리기만 한다면 결정은 이미 물 건너간 것이다. 테스트 시점은 과제의 후반부에 속하며, 그 시점에서 변경은 치명적인 결과를 낳기 때문이다. 시간은 누구에나 동일하게 주어지며 무한하지 않다. 과제는 "일정 준수는 생명이며 품질을 자존심이다."라는 말을 듣는다면, 자존심은 버리고 생명을 아껴야 한다고 생각할 것이다. 하지만, 오히려 개발자라면 마지막 순간에도 품질만큼은 절대 양보해서는 안 된다. 테스터들만 품질을 지킨다고 생각한다면, 당신은 이미 생명도 자존심도 포기한 개발자다. 그들은 당신과 같은 팀이 아니며 그들의 할 일을 할 뿐이다. 더 많은 버그를 찾아내고, 더 높은 수준의 테스트를 생각하는 것이 그들의 일이다. 물러서지 않아야 할 사람은 개발자뿐이다. 그들은 자신들의 노력과 힘을 합쳐서 만든 제품에 대해서 공정한 평가를 받길 원하기 때문이다. 결정이 늦어지면 당신의 퇴근 시간도 함께 늦어질 것이고, 당신을 집에서 기다리는 “또 하나의 가족”이라고 불리는 사람들도 늦게까지 잠들지 못할 것이다.

# [ 권한의 위임(Delegation) ]

코딩을 오랫동안 해온 사람이라면, “위임(Delegation)”이라는 단어를 들어본 경험이 있을 것이다. 위임은 해야 할 일을 다른 곳에 위탁한다는 의미로 자신이 해야 할 주요 업무(역할)만을 수행하고 나머지 부가적인 업무는 다른 부분에 “책임과 권한”을 주고 처리하도록 요구한다는 뜻이다. 만약 위임받는 쪽이 제대로 일을 하지 못하거나 원하는 결과가 나오지 않는다면, 위임은 좋은 선택이 못되며 위임한 일까지도 맡아서 처리해야 할 것이다. 하지만, 이렇게 모든 것을 혼자서 처리하는 것은 일을 복잡하게 만들고 처리율(Throughput)도 낮아지게 된다. 시스템은 구성요소들이 서로 유기적으로 결합해서 움직여야 제대로 된 역할을 수행할 수 있다. 복잡함을 기본 성질로 가진 이러한 시스템들은 모든 것을 한 부분에서 처리하기보다 기능들이 역할에 따라 나뉘어져 책임과 권한을 가지고 자율적으로 수행하도록 구성하는 것이 일반적이다. 즉, 복잡함을 해결하는 근본 원리는 "분할정복(Divide and Conquer)"이라는 것이다. 시스템에 대한 최적의 해를 찾는 것이 전체 최적화라면 좋지만, 비용과 시간이라는 물리적인 한계 때문에 시스템 전체 최적화보다는 최선의 해를 선택할 가능성을 열어두고 있다. 이 때도 진화적인 방법을 통해서 지속적인 최적화로 나아가는 방법을 선택할 수 있다. 권한의 위임은 선택을 역할과 책임을 나누어가진 각 부분이 스스로 결정할 수 있도록 만들어 준다.

위임을 받게 되는 쪽은 실제 업무를 수행하는 부분들이다. 현장에서 판단을 가장 잘 할 수 있는 충분한 정보를 가진 곳도 바로 그곳이다. 정보의 전달 과정에서는 반드시 정보의 손실(가공)이 발생하고, 경로가 길어지면 근거는 희박해질 수밖에 없다. 실무에서는 문제라고 이야기하는 부분들이 상부로 전달될수록 점점 더 많이 가공되어, 최종 전달을 받는 입장에서는 무슨 문제가 있는지조차 모호하게 되기도 한다. 이는 전달 과정에서 필연적으로 발생하는 관점의 차이와 권한의 역학 관계에 의해서 발생하는 것으로, 위임의 경로를 필요 이상으로 가져가선 안 된다는 의미다. 모든 권한을 다 가진 사람은 자신이 바쁘게 일하는 것이 다른 사람을 열심히 일하게 하는 것이라는 생각을 하고 있을지 모르지만, 실제로는 그 사람 덕분에 다른 사람들은 놀게 된다. 즉, 모든 결정권을 그 사람이 가지고 있는 상황에서는 다른 사람들은 그 사람이 판단하기만 기다릴 뿐이다. 결정권이 없는 사람들은 아무런 책임감을 느끼지 않기에, 실무에서 발생하는 문제에 대해서도 해결책을 스스로 찾으려고 하지 않는다. 자신이 조금이라도 편하게 일할 수 있기를 바랄 뿐이며, 만약 문제가 있어서 보고할 일이 생기면 나중으로 미루거나 내부에서 무마하려고 할 것이다. 권한을 가지고 있는 사람에게 보고하는 것으로 문제를 키우고 싶지 않기 때문이다. 위임하지 않고 모든 일을 한 사람의 판단에 맡기게 되면, 잘못된 정보를 바탕으로 하는 판단은 전체 조직을 위태로운 상황에 처하게 만들지도 모른다. 모든 것을 경험하고 역량도 뛰어난 관리자라면 그런 일도 쉽게 처리할 수 있을지 모르지만 현실에서 그런 사람을 찾기는 어렵다.

위임은 사람의 역량을 키워주며 일을 주도적으로 처리할 수 있도록 만들어준다. 믿고 맡기면 당연히 일을 주도적으로 할 수밖에 없다. 누구도 대신해서 결정해 주는 사람이 없기 때문이다. 자신이 결정을 내릴 수 있는 상황이 되면, 그전까지 일하던 방식으로는 도달할 수 없는 새로운 기준이 만들어진다. 즉, 자신이 만족하지 못하는 수준으로 일할 수 없게된다. 자신의 이름을 걸고 하는 일이라 절대로 적당히 할 수 없다. 책을 써본 경험이 있는 사람은 자신의 글에 대해서 책임을 느끼게 된다. 거짓을 이야기해서는 안 되며 자신감 없는 논리도 있을 수 없다. 더 많이 알려고 노력하고 결과물에 대해서 부족함을 느끼면 개선하게 된다. 개선은 억지로 밀어붙인다고 되는 것이 아니다. 개선해야 하는 대상들이 개선의 주체가 될 수 있도록 만들어 주어야 한다. 각종 TF에 참여하는 사람들은 왜 그 TF에 참석하는지도 모르는 경우가 많다. 단순히 TF에 참여하는 이유가 직급이 될 수도 있고 마땅히 해야할 일이 없어서 일 수도 있다. 그런 사람들이 참여한 TF의 목표는 욕먹지 않을 수준에서 개선을 찾아내는 것이다. 따라서, 대단한 것을 바라지도 않을 것이다. 하지만, 문제를 스스로 인식하고 개선하여야겠다고 마음먹은 사람들이라면 그런 식으로 대충하는 것을 용납하지 않을 것이다. 위임은 책임만을 묻는 것이 아니라 권한도 함께 하기에 이런 것들이 가능하다. 현실의 관리자들은 책임은 이양하면서도 권한은 잘 주지 않는다. 이유는 자신들의 정치적인 영향력(힘)를 절대 놓고 싶은 생각이 없기 때문이다. 위임이라는 말은 사용하지만 위임에 필요한 방법은 잊어버리고 있는 것이다. 자신들이 하고 싶은 이야기를 하고 듣고 싶은 이야기만 들어서는 제대로 된 조직을 만들지 못한다는 것을 그 자리에 가서도 모른다. 소통이란 자신이 듣고 싶은 이야기를 해주는 사람들과만 한다. 위임은 서로가 서로에 대한 믿음과 대화로 이뤄갈 수 있다. 위임은 책임과 권한이 함께 움직이는 일이라는 사실을 알아야 한다.

일을 잘하기 위해서는 작은 일을 한 번에 하나씩만 처리하는 것이 좋다. 자신의 "작업 대기열"이 넘치도록 지켜보고 있으면 안 된다. 이를 위해서는 자신이 하는 일에 대한 관리를 스스로가 할 수 있도록 만들어야 한다. 관리자는 전체 팀 차원의 작업 대기열(Queue)를 유지할 책임이 있으며, 작업 큐에 존재하는 각각의 작업에 대해서 팀원들이 능동적으로 처리할 수 있도록 “권한을 위임”해야 한다. 만약 작업 큐에 해야할 일이 지속해서 쌓인다면(넘친다면), 큐에 존재하는 작업의 양(큐의 길이)을 조절해 주어야 한다. 즉, 들어오는 작업의 양을 조절하는 조치가 필요하다. 각 개인도 마찬가지로 자기 일이 과중하다고 생각되면, 지속해서 큐의 입력을 제한해야 한다. 이렇게 하는 이유는 작업이 지나치게 오래동안 큐에서 지연되지 않게 만들고, 중요한 작업에 집중할 수 있도록 하기 위함이다. 작업을 큐에 쌓는 사람은 큐의 제한된 크기로 인해서 중요하고 급한 것을 선별해서 넣을 것이다. 작업하는 실무자의 입장에서는 한 번에 한가지의 작업을 하기에, 언제나 현재의 작업이 완료되면 다음 작업을 큐에서 가져와서 처리하게 된다. 비는 공간이 생길 때는 작업을 큐에 넣기 위해서 기다리는 사람들이 협의해서 어떤 작업을 다음에 할 것인지 결정하게 된다. 작업의 선별은 요청자들이 하지만 작업의 처리는 위임받은 실무자들이 자신의 작업 처리 "속도"에 맞게 한다. 이렇게 하면 사람들이 게을러진다고 이야기할지도 모르지만, 팀으로 일하게 되면 동료로부터 피드백을 받기에 그럴 수가 없다. 성과는 팀으로 평가를 받기에 스스로 반성하고 더 노력하게 될 것이다. 외부로부터의 비난보다는 내부에서의 피드백이 개인을 변화시키는 데는 더 효과적인 방법이다. 아무리 좋은 말이라도 관계가 소원한 사이에 듣는 말은 그냥 한 귀로 듣고 한 귀로 흘러갈 뿐이다. 개선이나 변화는 능동적으로 권한과 책임을 가지고 일하는 개인이 있어야만 가능하다.

# Part II. Process

# [ 문서 쓰기 ]

문서를 쓰는 것은 몹시 어렵다. 정확히 말하자면, 좋은 문서를 작성하는 것은 몹시 어렵다. 하지만, 생각해보자. 정말 문서가 쓰기 어려워서 안 썼는지? 아니면, 그냥 쓸 시간이 없다고 미루어서 안 쓰게 되었는지. 결론은 아마도 대부분 후자일 것이다. 글쓰기를 프로그램의 코드를 짜기보다 어렵다고 생각하는 사람들인 프로그래머들은 대부분 아마도 글쓰기를 싫어하는 부류에 속할 것이다. 신문이나 기타잡지에 글을 쓰는 사람들은 대부분이 글로서 돈을 버는 사람이니 잘 써야 한다. 하지만, 우리 같은 부류의 소프트웨어 엔지니어들이 정말 글을 잘 써야지만 생존할 수 있을까?

소프트웨어 엔지니어도 글을 잘 써야 한다. 좀 더 적극적으로 이야기 하자면, 명확히(혹은 정확하게) 자신의 의사를 전달하거나 이유를 설명할 때는 반드시 글을 잘 써야 한다. 글을 잘 쓴다는 것이 소설을 잘 써야 한다는 것이 아니라, 정확히 간결하게 자신의 생각을 잘 전달할 수 있어야 한다는 것이다. 우리가 쓰는 대부분 글(기술 문서)이 이러한 범주에 속한다. 잘못된 글은 잘못된 생각을 낳고, 잘못된 생각은 누군가의 피해를 발생시켜, 결국에는 그 위험이 자신에게까지 영향을 주기 때문이다. 따라서, 우리는 우리가 쓰는 글에 대해서 글을 읽는 상대방이 잘 이해할 수 있도록 만들어야 할 의무가 있다(코딩도 크게 다르지 않다).

글은 많은 사람이 생각을 공유하는 매개체이다. 만약 글이 없다면 어떻게 일을 진행하겠는가? 구두로? 가능한 일이다. 예전에 인도의 브라만은 자신들의 기도문을 구전으로 대대로 전달했다고 한다. 몇 천 년을 이어온 전통이, 사는 지역은 다르지만 기도문은 동일하게 암송하도록 만들었다고 한다. 몰래 남들이 엿듣는 것을 방지하기 위해서, 어두운 곳에 아들과 아버지가 들어가서 지속해서 암송을 시켰다고 한다. 하지만, 우리가 하는 일은 소프트웨어를 개발하는 일이다. 고객이 있고 프로젝트에 관련된 많은 사람이 있다. 그들 모두를 어두운 곳에 모아서 암송을 시킬 수는 없는 일이다.

따라서, 우리는 문서라는 인류의 가장 오랜 의사전달 수단을 이용해서(물론, 여기에는 이견이 있을 수 있지만), 하고자 하는 것과 방법 등을 공유해야 한다. 문서는 지식과 경험을 공유하기 위해서 작성한다. 나와 남의 서로 다른 지식과 경험을 공유하는 것이다. 공유하지 못하는 것은 도태된다. 누군가가 만들어두었지만, 시간이 흘러 다시는 관리가 되지 않을 때, 그것을 버리게 되는 것이 일반적이다. 공유되지 않는 경험은 고립을 낳는다. 고립된 생각으로 모든 일을 처리하고자 한다면, 주변의 반발이 생기고 결국에는 주변과 조화를 이루지 못해 독단하게 된다. 프로젝트를 혼자 하던 시대는 지나갔다. 회사에서 대부분 프로젝트는 팀으로 진행되며, 프로젝트의 중요한 결정도 팀 회의로 이루어지는 것이 일반적이다(물론, 반드시 그런 것은 아니지만). 따라서, 공유는 항상 올바른 결정의 토대가 되며, 문서는 그 기본이라고 생각할 수 있다.

그렇다면, 어떤 문서가 의미가 있을까? 어떤 문서가 소프트웨어 개발자가 작성해야 하는 문서일까? 소프트웨어 개발자가 써야 하는 문서는 써야 할 말을 간결하고 이해하기 쉽게 쓴 문서다. 흔히 회사에서 어떤 회의자료를 만들라고 하면 당연히 PPT(Power Point 문서)를 만들고 본다. 하지만, 여기서 중요한 점은 PPT가 문서의 역할을 하기는 하지만 부족하다는 점이다. PPT는 발표자가 발표하는 자리에서 사용하는 자료이다. 발표자가 없는 자리에서 읽기는 쉽지 않다. 잘 정리된 PPT가 많은 정보를 전달할 수는 있지만, 세세한 상황까지는 전달하기 어렵다. 따라서, 적어도 소프트웨어 개발자가 작성하는 문서는 PPT보다는 일반적인 문서의 형식을 갖추는 것이 중요하다.

형식을 갖춘 문서란 무엇인가? 여기서 형식이란 그 문서의 역할과 관련이 있다. 대략적으로 이야기하자면 소프트웨어 개발자로서 요구사항을 기술한 문서, 설계문서 정도는 잘 작성해야 한다. 요구사항은 개발하고 있는 소프트웨어가 어떤 요구를 만족시켜야 하는지를 기술하는 것이다. 이를 통해서 어떤 것을 구현할 것인가를 명확히 하는 데 도움이 된다면 어떤 것이라도 상관이 없다. 설계문서는 앞에서 요구된 기능을 어떻게 구현할 것인가를 표현하는 것이다. “어떤 요구사항은 어떻게 구현된다.”와 같이 명확히 주어진 요구사항을 반영했다고 판단될 근거가 있다면 형식은 상관이 없다. 폼(Form)을 주면 채워서 주겠다는 사람들이 있다. 폼은 형식을 통일하는 것이지 내용을 통일하는 것은 아니다. 살아있는 내용이 아니면 아무리 좋은 폼을 갖춘다고 하더라도, 문서 자체에 없던 의미가 생겨나는 것은 아니다.

문서를 쓰는 것은 습관이다. 습관적으로 문서를 작성하는 능력을 키우지 않고서는 좋은 문서를 쓸 수 없다. 일상에서 이러한 일을 할 기회는 아주 많다. 특히, 요즘 같은 블로그가 일반화된 시대에서는 더욱 그렇다. 하루 하루 일기를 아이들이 써나가는 모습에서, 그렇게 글쓰기를 익히는 것이 가장 효과적이라고 이미 증명된 듯하다(적어도 부모의 입장에서는). 소프트웨어 개발자도 마찬가지다. 오늘 읽었던 책이나 혹은 오늘 개발했던 것을 하루에 한 페이지라도 정리하고 넘어가는 것이 가장 좋다. 나중에 밀린 문서를 한 번에 쓰고자 하면 절대 좋은 문서가 나오지 못하며, 문서의 생명력도 오래가지 못한다. 즉, 실수했던 것도 문서로 남기는 것이 좋으며, 판단의 근거를 알려주는 수고도 아껴서는 안 된다. 실수로부터 배우는 것이 소프트웨어 개발이다. 남들의 아주 많은 실수에 대한 보상으로 우리는 지금과 같은 세상을 살기에 더욱 그렇다. 따라서, 만약 자신이 신출내기 개발자라면 오늘 하루도 그냥 잠들지 말고 블로그에 오늘 했던 일이나 배웠던 것이라도 한 자 적도록 하자. 문서를 쓰는 것이 편안하게 자신의 내면과 솔직한 대화하는 즐거운 시간이 되도록 만들자.

여러 가지 두서없는 이야기를 했다. 하지만, 중요한 것은 글쓰기는 공유를 위해서 필요한 활동이며, 공유되는 지식이야말로 오랜 시간 생명력을 가지고 발전해 나갈 것이다. 소프트웨어 개발자는 사실 언어를 여러 가지 배우고, 그 언어로 해야 할 일을 해석해서 프로그래밍이란 도구로 글을 적어나가는 아주 고된 일을 하는 사람이다. 스스로 신세를 타락시켜서 꿈이 없는 세상 속을 헤매지 말고, 프로그래밍 언어로 끝없는 창조적인 작품들을 열심히 만들어 나가야 한다. “문서는 그 문서를 작성한 사람보다 항상 더 오래 살아남는 법”이다. 퇴직한 후에도 코드와 문서는 그 회사에 고스란히 남는다.

# [ 회의 및 보고서를 없애자 ]

회사에서 평균 하루에 얼마나, 혹은 일주일에 얼마나 많은 시간을 회의시간으로 사용하는가? 회사에서 평균적으로 쓰는 보고서의 양은 또 어떤가? 우리는 회사에서 수많은 회의를 하고 산다. 무슨 이유인지는 모르지만, 한 사람 혹은 몇몇 사람이 회의시간의 대부분 시간을 사용하고, 자신이 이야기하는 시간은 2시간 회의에서 5분 남짓하다. 하지만, 그래도 한 번의 회의라면 그냥 참고 들을 만 하지만, 많게는 온종일 여러 개의 회의에 참석해야 하는 경우도 비일비재하다. 보고서는 또 어떤가? 일간보고, 주간보고, 월간보고, 분기보고, 반기 보고, 년간보고 등등 자신이 조금이라도 관련된 모든 프로젝트에 대해서는 항상 따라오는 일이다. 직급이 올라갈수록 이런 회의 및 회의자료 준비, 각종 보고서로 정작 중요한 자신의 팀에 대한 일은 놓치기 일쑤다. 마치 일이 바빠서 가족의 생일이나 중요한 행사에 참석하지 못하는 것처럼.

“회의는 없는 것이 좋고, 보고서는 안 쓰는 것이 좋다”. 이 말은 진리라고 생각한다. 사람이 하루에 평균적으로 자신이 지금 진행하고 있는 일에 집중할 수 있는 시간은 몇 시간이 안 된다(오버타임을 제외하고 4시간이나 될까?). 사실 아주 비효율적으로 회사생활을 하는 것이 현실이다. 대부분이 관심을 두지 않거나, 혹은 몇몇 사람의 욕심으로 하는 회의나 보고서 쓰기는 안 하는 것이 최선이다. 회의할 것을 한번 안 하게 되면 그 만큼의 자료작성 노력이나 회의로 낭비되는 시간을 줄이게 되고, 이런 시간을 일에 좀 더 투자할 수 있게 된다.

보고서를 쓰는 노력 일부라도 차라리 자신이 하는 프로젝트를 문서화시키는데 투자하는 것이 옳다. 문서라면 쓰기 싫어하는 사람들도 어떻게든 보고서는 열심히 작성한다. PPT를 주어진 템플릿(Template)을 이용해서 글자 크기와 종류를 정해서 작성하고 그림을 그려서 넣는 것은 못 잡아도 꽤 오랜 시간이 필요하다. 어떤 경우에는 한 달 내내 10여 페이지의 보고서를 작성한다고 몇 번의 회의와 리뷰로 시간을 보내기도 한다. 하지만, 장담하건대 PPT를 통해서 지식을 축적하는 것은 거의 불가능하다. 오히려 Word나 HWP 문서로 만드는 것이 훨씬 관리되고 오래 남는다. 시간이 가면 PPT에 작성했던 내용은 무슨 내용인지도 가물가물해진다.

그러면, 어떻게 이런 “필요악”이라고 생각하는 회의와 보고서를 어떻게 없앨 수 있을 것인가? 사실 찾아보면 아주 많다. 그리고, 그 중 일부는 이미 사용하고 있다. 간단히 예를 들면, 첫 번째로 Email을 사용하는 방법이다. 관련된 사람들에게 의견을 묻는 Email을 주고, 업무를 지시하면 된다. Email이 보기 힘들다고 생각한다면, 이슈(Issue)관리 시스템과 같은 것을 통해서 업무를 지정하고 중요한 결정을 남겨둘 수 있다. 필요하다면 실제 업무를 담당하고 있는 말단 사원이라도 직접 의견을 제시하고, 이에 대한 문서화된 기록(History)을 남겨둘 수 있다. 보고서는 어떻게 할 것인가? 대부분의 능력이 된다고 생각되는 회사에서는 과제를 관리하는 시스템을 다 갖추고 있다. 과제를 관리한다는 말은 그 시스템을 통해서 과제의 진척 정도(상황)이라든지, 혹은 산출물 등을 다 확인할 수 있다. 보고서를 받지 않고 그냥 그 시스템에 접속해서 보면 현재 프로젝트들이 어떻게 돌아가고 있는지 다 파악할 수 있다는 것이다.

물론, 그러한 시스템에 거짓 정보를 넣는다면 결국 거짓을 보고 받게 된다는 점이 문제가 되지만 이는 쉽게 해결할 수 있다. 즉, 과제의 결과물에 대한 검토를 절차에 따라 하게 되면, 어떤 프로젝트의 누가 어떤 거짓 정보를 입력했는지 정도는 즉시 알 수 있다. 대부분 관리자는 나이가 많다. 부장급만 되어도 마흔을 넘겨서 신입과 20년에서 30년 차이를 보일 수 있다. 한 세대가 다른 것이다. 배운 것이 다르고, 생각하고 느끼는 것이 다르다. 하지만, 변하지 않는 것은 시스템으로 일하면 훨씬 더 능률적으로 일할 수 있다는 점있으며, 남겨진 기록은 거짓말을 하지 못한다는 것이다.

결론적으로 이야기하면, 회의와 보고서가 없는 회사가 궁극적으로 성장할 수 있다고 본다. 우리가 일하는 회사는 이합과 집산이 가족과는 다르다. 언제고 사람이 들고 날 수 있는 것이 회사다. 특정 누군가가 지배적이고, 그 사람이 없으면 회사가 제대로 굴러가지 못하게 된다면, 결국 그 사람 만을 위한 회사이지 함께 일하는 임직원 모두를 위한 회사는 아니다. 시스템적으로 일하는 회사는 일에 더 많은 중요성을 두지, 기타 자질구레한 업무들에 중요성을 강조하지 않는다(이것은 프로세스가 없는 회사라는 뜻이 아니다). 누가 야근을 하든 안 하든 프로젝트의 결과는 정해진 시점에 나오기 마련이고, 모든 프로젝트 진행에 관련된 경험은 시스템에 쌓이지 사람에게 쌓이지 않는다(물론, 사람도 경험은 쌓아간다). 새로 어떤 사람이 들어오든 상관없이, 시스템에 등록된 자료만 가지고도 충분히 빨리 적응할 것이다.

# [ 과제 관리 ]

우리는 회사에서 많은 프로젝트를 수행한다. 신입은 남들이 이미 구성한 내용의 프로젝트 항목에서 자신이 맡을 업무를 할당받는 것으로 시작해서, 조금 더 경력이 쌓이면 몇 명을 자신의 관리영역에 두고 독자적인 업무를 수행하기도 한다. 팀장이 되면 프로젝트를 기획하고, 팀원과 같이 결과물을 얻기 위한 활동을 하게 된다. 하지만, 프로젝트의 리더(PL)가 되는 것이 이처럼 경력을 쌓는다고만 해서 가능할까? 이 물음에 대해서는 여러 가지 측면에서 고려할 것이 있지만, 일단 이렇게 답할 수 있을 것이다. "제대로 된 PL밑에서 제대로 된 프로젝트 수행 방법을 경험하고, 몇 번의 프로젝트를 진행해본 경력이 있다면 가능하다."라고 하겠다. 중요한 점은 "제대로"라는 말이다.

일상에서 우리가 만나는 PL들은 “제대로 된 경험”을 쌓지 못한 경우가 많다. 예를 들어, 전혀 다른 분야에서 프로젝트를 수행했던 사람이 PL로 와서 이렇게 저렇게 하라고 지시하는 경우가 허다하다(하드웨어만 만들어 본 사람이 소프트웨어 과제의 리더 역할을 하는 것처럼). 실제로 완전히 일자무식인 사람이 와서 자신이 마음에 들 때까지 무한정 “삽질 신공”으로 일에 대한 능률과 집중도를 최하로 만드는 경우가 비일비재하다. 그렇다면, 이런 상황을 마주했 때 스스로는 어떤 생각을 가지게 될까? 가장 취하기 쉬운 것은 "Let it be"정신이다. 즉, "냅둬". 자발적으로는 아무것도 하지 않고, “시키는 일만 하자는 주의”다. 어떤 경우에는 PL과 회의라는 회의는 다 참석해서 싸우는 경우도 있다. 이때는 네가 죽나 내가 죽나 해보자는 자신의 능력을 과신하는 경우로, 반드시 패배할 가능성이 높다. 왜냐면, 고위층에서 PL을 시킬 때는 나름대로 이유가 있기 때문이다. 그럼, 어떤 태도를 보이는 것이 가장 효과적일까? 그것은 개인의 성격에 따라서 다르겠지만, 일단은 “시키는 일에 대해서 명확하고 논리적으로도 멋지게 해내는 것”이 필요하다. 그리고나서, 개선점을 이야기하는 것이 향후 인간관계 형성에 있어서 피곤함을 덜어줄 것이다. 중요한 것은 회사의 일을(프로젝트를) 완수하는 것이지, 그 프로젝트에 관련된 사람들이 다 말아먹고 망하는 것이 아니기 때문이다. 회사가 돈을 벌어야 개인의 지갑도 두둑해진다.

프로젝트 관리라는 것이 그럼 무작정 경험 만을 바탕으로 할까? 그렇지는 않다. 경험과 지식이 있어야 한다. 지식만 있으면 머리로만 일하려고 해서 생기는 이상과 현실의 괴리에서 좌절하게 될 것이고, 경험만 들이밀면 나날이 더 똑똑한 부하직원들에게서 경멸의 눈초리를 받을 것이다. 경험이 있다면 이론을, 이론이 있다면 겸허하게 유경험자를 인정하는 태도를 가져야 한다. 물론 그렇다고 일방적으로 팀원들에게 휘둘리는 것은 옳지 않다. 이때 사용할 수 있는 가장 강력한 무기가 회사의 "프로세스"라는 것이다. 원칙으로 밀고 나가는 것이다. 간혹 타협은 있을 수 있으나 굴복은 허락되지 않는다. 2보 전진을 위한 1보 후퇴는 있을 수 있지만, 앞으로 갈 생각도 하지 않는 후퇴는 없다. 어쨌든 이 시대의 많은 팀장은 변화의 물결에 그냥 빠져 죽지도 않고 이리저리 부유하면서 살고 있다. 하려면 제대로 배우고 관리해야 한다. 경험이 술자리에서 좌석 순을 정하는 것이 아니라면, 최소한 제대로 배워서 그것을 자신의 경험으로 되새김질할 수 있는 능력은 있어야 한다. 열심히 일하는 부하직원과 동료에게 부담스러운 존재는 되지 말아야 하기 때문이다.

# [ 누구를 위해 종은 울리나? ]

일하다 보면 사람마다 보는 관점이 다양하다는 것을 느낄 때가 많다. 대부분의 경우 이러한 관점의 차이는 자신의 경험 및 지식의 정도, 미래에 대한 나름의 예측에서 나온다고 본다. 생각이 다르면 말을 달리하게 되고, 말이 달라지면 결국 행동도 달라지기 마련이다. 다른 일도 마찬가지지만 소프트웨어를 개발하는 팀에서 일한다면, 이러한 다르다는 것이 과제를 지연시키는 다양한 형태로 표출되는 경우가 많다.

과제의 지연은 사람을 궁핍하게 만들고 변명하게 만드는 아주 고약한 놈인데, 이를 그냥 보고만 있기에는 과제 책임을 맡은 사람으로서는 속이 탈 수밖에 없다. 팀을 관리하기 위해서는 먼저 팀에서 지켜야 하는 규정을 미리 정하는 것이 좋다. 먼저 사용하는 모든 문서의 양식을 같게 가져가는 것은 아주 중요다. 파워포인트의 템플릿을 만들고 모든 사람이 앞으로 사용하는 발표자료는 그것을 사용해서 작성하도록 한다. 워드 문서의 경우에도 특정 포맷을 만들어서, 특별한 경우가 아니면 항상 그 포맷으로 문서를 작성하도록 만드는 것이다.

소스 코드를 관리하는 규칙을 만들고, 만약 새로 작성된 코드가 있을 시에는 누군가의 리뷰가 있기 전에는 절대 메인 라인(Main Line)의 소스 트리(Source Tree)에는 넣지 못하게 막는다. 새로운 팀원이 왔을 때는 기존에 하던 코딩 룰(Coding Rule)을 빨리 따라서 할 수 있도록 코딩 가이드(Coding Guide)를 작성해서 주도록 하고, 각종 일을 하는데 필요한 메뉴얼과 소스 코드의 위치 등을 미리 정해 두도록 한다. 모듈 간의 의존성을 낮추려는 방안을 고안하고, 보안에 관련된 코딩을 쉽게 할 대체 방안도 미리 제공해 줄 수 있도록 만들어 두는 것도 좋다. 하지만, 이 모든 것에 앞서 결국 팀은 유기적으로 창조적인 일을 하고 싶어하는 사람으로 이루어져 있다는 것을 잊어서는 안 될 것이다. 또한, 규칙은 복잡할수록 안 따르기 마련이고, 너무 간단한 규칙은 무의미하다고 생각하기 마련이므로, 적절한 수준에서 트레이스 오프(Trade-Off) 할 필요가 있다.

직급이 높은 사람은 창조적인 일을 “규정”대로 하기를 원한다. 물론 소프트웨어 개발만 창조적인 일이라는 것은 아니지만, 대부분의 사람들은 “소프트웨어 개발은 창조적인 정신활동의 결과물”이라고 보는 것이 일반적이다. 하지만, 한정된 자원과 기간을 정해놓고, 원하는 결과물을 만드는 것은 쉬운 일이 아니다. 대부분의 경우 우리가 만들고자 하는 것에 대해 정확한 정보가 없이 계획을 수립하는 것이 다반사이므로, 프로젝트의 지연은 늘 있는 일상과도 같은 것이라고 쉽게 체념하는 것이, 많은 PL(Project Leader)들이 겪는 공통적인 문제다.

항상 그렇지만 프로젝트는 어떤 절차를 준수해서 결과물을 중간중간 만들기를 원한다. 개발에 익숙하지 않은 사람들에게는 교육을 제공해야 하고, 각종 세미나도 해야 한다. 주간회의, 월간회의, 분기회의, 과제심의, 과제완료, 과제 중간점검회의 등등 아주 많은 회의는 "회의감"을 들게 하기에 아주 많다. 모든 프로세스가 귀찮아 질 때쯤이면 무의식적으로 결과물의 품질을 포기하는 경향이 많은데, 이것은 결국 뒤에 오는 사람이나 다른 프로젝트를 수행해야 하는 사람들에게는 전혀 도움이 안 되는 일이다.

"품질은 자존심이고, 납기는 생명이다." 따라서 나는 자존심을 버리겠다는 뜻밖에 안 된다. 하지만, 진정 우리가 프로페셔널(Professional)이라고 한다면, 오히려 개인의 자존심을 더 지키는 것이 옳지 않을까? 물론, 그것 때문에 납기 문제를 일으켜서 기회를 놓칠지도 모르지만. 어쨌든 중요한 것은 조금이라도 과제를 성공시킬 확률을 높이기 위해서는 회사에서 정한 절차를 따르는 것이 효과적이다. "악법도 법이다"라고 외치며 독약을 단번에 마신 분보다는 소프트웨어 개발자가 생명을 걸 일은 드물기 때문이다.

QA(Quality Assurance)는 모든 개발자의 적처럼 보인다. 상품화하다 보면 매번 극복해야 할 대상 1위에 속하는 사람들이 그 분들이다. 마치 권력의 핵심처럼 도무지 아무리 공략해도 열리지 않는 성문처럼 보인다. 하지만, 태도를 바꿔놓고 보면 그 사람들은 우리가 만든 제품의 가장 마지막 단계에서 품질을 유지할 수 있도록 도와주는 고마운 분들이다. 물론 일하는 방식이나 절차가 썩 마음에 들지는 않지만, 나름대로 자신의 기준을 가지고 일하고 있는 것은 확실하다. 우리가 놓칠 수 있는 많은 문제를 걸러낸다고 힘겹게 일하고 있는 것은 사실이다.

QA가 개발자로 자신의 위치를 바꿔보면, 개발자들은 상품화 일정에 허덕이며 보고된 많은 문제를 해결한다고 고생하는 사람들이라는 것을 금방 알 수 있을 것이다. 개발자들은 자존심과 생명을 지키기 위해서 열심히 일한다(대체로 고등교육을 다 마친 사람은 자신이 해야 할 일이 있으면 최선을 다하는 경향이 있다는 가정). 그리고, 그런 개발자들이 더 잘 일할 수 있도록 해주는 것이 무엇보다 필요하다고 생각하게 될 것이다. "역지사지"라는 말이 이보다 더 잘 어울리는 경우가 있을까? 우리는 싸움을 하려고 회사에 오는 것이 아니다. 우리는 우리가 사랑하는 가족들을 위해서 회사에 오는 것이며, 내가 개발한 제품이 시장에서 고객에게 인정받아 많이 팔리는 것을 즐겁게 생각하기에 기꺼히 출근하는 것이다.

수치가 모든 것을 말해주지는 않지만 적어도 기억할만한 것은 제공한다. 어떤 과제를 수행했을 때 그 과제 속에서 발생했던 오류(Defect)들의 모든 내용은 기억할 수 없지만, 최소한 몇 개의 오류가 발생했는지는 오랫동안 기억에 남는다. 뭔가 일을 했을 때 가장 좋은 표현 방법은 그것을 수치로 표현하려고 노력하는 것이다(“$”이나 “\”으로 나오는 값을 원하겠지만). 견해의 차이를 극복하는데도 숫자적인 표현을 하는 것이 좋다. 잘못된 수치는 잘못된 결과를 도출하므로 조심해야 하지만, 가능한 모든 결과를 수치로 남기고 이를 논리적으로 풀어서 근거를 제시한다면 서로 간의 불신과 오해는 상당 부분 줄어들 것이기 때문이다. 과학기술에 관련된 논문에서 실험의 결과를 그래프로 그리고, 그렇게 된 이유를 이론으로 정립하듯이(혹은 그 반대로 이론을 정립하고, 그것을 실험해서 결과를 수치화시키기도 하지만), 개발자가 하는 일은 분석되어야 하며, 논리적으로 기록되어야 한다. 일의 결과가 비록 조금 나쁘더라도 멋지게 일을 정리하고자 한다면 반드시 지켜야 할 활동이다.

다양한 차이를 어떻게 잘 녹여서 하나의 과제를 수행할 것인가를 기술적으로 처리하는 것은 어렵다. 모든 사람이 인사관리의 달인이 될 수는 없기에, 이러한 것들이 보이지 않는 과제 지연의 요소로 작용한다. 위험(Risk)으로 등록해서 관리하든, 힘으로 의견을 무시하고 지나가든, 어쨌든 사람이 하는 일에는 항상 차이가 있기 마련이고, 이러한 차이를 최소화하려고 노력하는 모든 활동은 결과물에 주요한 영향을 미친다. 결국 모든 일은 사람이 하는 것이고 도구는 도움을 줄 뿐이다. 사람과 사람 간의 일은 대화를 통해서 해결해야 한다고 하지만, 그것이 힘들 때는 시스템의 도움을 얻기도 해야 하고, 지켜야 할 규정에 의존하기도 해야 한다. 스스로 완벽한 존재가 아닌 사람이기에 견해의 차이는 인정할 수밖에 없지만, 좀 더 남을 배려하는 태도로 긍정적인 생각을 가지고 결과를 보도록 하자. 남을 이해하는 길이 자신이 하고자 하는 일의 지름길임을 알고, 열어놓는(Open Minde) 것이 결국에는 더 많은 것이 들어올 수 있는 길을 닦는 것임을 기억하자.

# [ 급한 것과 중요한 것? ]

급한 것과 중요한 것? 컴퓨터 프로그래밍은 다양한 지식을 요구한다. 컴퓨터 관련 학과를 졸업한 사람도 산업의 현장에 배치를 받으면 가장 먼저 해야 할 일은, 자신이 일하는 분야의 도메인 지식(Domain Knowledge)를 익히는 것이다. 즉, 전공 지식이라고 불리기도 하는 이러한 것들은 컴퓨터 프로그래밍을 하려는 일의 목적에 활용되기 위해서는 필요하다.

예를 들어, 4년제 학부 과정을 졸업한 학생이 전자 회사에 배치를 받았다고 하면, 업무에 필요한 지식에는 어떤 것들이 있을까? 기본적으로 응용 프로그램이 동작하는 환경을 잘 알아야 하며, 사용하는 라이브러리와 시스템 콜(Call), 운영체제의 핵심인 커널의 구조, 그리고, 제어하려는 하드웨어 등등 많은 분야의 지식을 한꺼번에 익히라는 요구를 받을 수 있다. 하지만, 사실 한 번에 이러한 지식을 전체적으로 습득하기란 쉽지 않다. 또한, 자신이 맡은 분야의 문제를 해결하기 위해서 그 문제 도메인 자체에 대한 지식도 필요하다. 즉, 누군가 도와주기 전에는 일을 제대로 시작도 하지 못할 가능성이 크다.

도메인에 관련된 지식은 해당 분야에서 일하지 않고서는 제대로 알 방법이 거의 없다. 즉, 해당 분야의 최신 기술이나 실제 적용 시 발생하는 문제점들을 상품화를 통해서 경험하지 않고서는 알 방법이 없다. 물론, 잡지나 웹 사이트와 같은 곳을 뒤져서 나오는 지식으로 알 수 있다는 사람들도 있지만, 해당 분야에서 알 수 있는 지식은 시행착오를 거쳐서 정제된 지식이다. 따라서, 경험해 보지 않는다면 결코 알 수 없는 지식인 것이다.

또한, 이와 같은 지식은 특정 도메인에서만 습득될 수도 있기에, 다른 도메인에 적용하는 것도 쉽지 않다. 이런 분야의 지식은 개발의 시작 시에는 미미한 정도 밖에 갖출 수 없지만, 개발이 완료되는 시점에는 가장 높은 수준에 도달하게 된다. 소프트웨어 개발자는 하드웨어나 특정 도메인 전문가의 도움을 받아서 실생활에서의 문제를 해결해 나가기 때문에, 이론적인 수준에서는 부족할지 모르지만 그것을 실무에서 해결하는 능력은 최고 수준에 도달할 수 있게 되는 것이다.

소프트웨어 개발자는 다양한 일을 하기를 원하는 경우가 많다. 특히, 한 제품을 오랫동안 하게 되면 나태해지기도 한다. 몇 번의 상품화를 거치게 되면 반복적인 일에 흥미를 잃어버리게 되고, 점차 외부의 자극에 대해서도 무감각하게 변하게 된다. 이런 일이 오랫동안 지속하면 "내가 도대체 여기서 뭘 하고 있는 거지?"라는 의문을 가지게 된다. 물론 이것은 소프트웨어 개발자만이 가지는 것은 아니라고 본다.

대체로 경험상 이러한 물음을 스스로 던지게 되는 시점은 3년에서 5년 사이다. 그 의문이 조금 사그라들면 다시 한 10년 정도는 견딜 수 있는 나름의 자기 방어적인 이유를 몇 가지 가지게 되고, 조금 더 지나면 거의 자신의 꿈에 대해서는 잊고 살아가게 된다. 가장 힘든 순간은, 아마도 의문과 자기만의 이유, 꿈 등을 거의 생각하지 않고 살아가다가, 문득 자신이 살아온 길에 대해서 뒤돌아볼 때, 아무것도 자랑할 만한 것이 없다고 느낄 때가 아닐까?

급한 일은 급하게 처리하고, 중요한 일은 중요하게 처리해야 한다. 이 말은 급한 일을 다루는 것과 중요한 일을 다루는 것을 나누어서 해야 한다는 것이다. 인력의 여유가 있는 조직이라면 급한 일을 하는 사람과 중요한 일을 하는 사람을 구분할 수 있을 것이다. 여유가 없는 조직에서는 최소한 급한 일을 처리하는 코드와 중요한 일을 처리하는 코드가 나누어져 있어야 한다. 다음에 일정한 시간 간격을 두고 지속해서 통합(Merge)을 해주는 것도 잊어서는 안된다. 결국, 중요한 일을 해결하는 것이 급한 일을 해결한다는 것임을 잊어서는 안 될 것이다.

사실 말은 쉽게 할 수 있지만, 이 글을 쓰는 사람조차도 실천하기는 힘들다. 매일 매일 발생하는 버그를 해결하기도 바쁜데, 어떻게 구조의 개선이나 성능의 향상을 동시에 할 수 있겠는가? 중요한 일을 잘하기 위해서는 절차와 방법이 필요하다. 절차는 일을 처리하는 순서를 말하며, 방법은 처리할 때 어떤 식으로 할 것인가를 정한 것이다. 이런 것들은 일을 시작하기 전에 미리 정해 두어야 한다. 가장 쉬운 예로 소프트웨어를 개발할 때 코딩은 어떤 식으로 하자고 정하는 코딩 표준(Coding Standard) 혹은 코딩 규칙(Coding Rule)이 있다.

오류(Defect)를 발견했을 때도 테스터와 개발자가 어떤 식으로 처리한다는 규칙도 있다. 새로운 요구사항이 접수가 되었을 때, 그것을 어떻게 반영할지를 정해두는 것도 좋은 예이다. 새로운 코드를 작성했을 때, 그것을 관리되는 코드의 메인 라인(Main Line)에 반영하기 위해서 어떤 식으로 리뷰(Review)를 할지도 정할 수 있다. 사실 이러한 것들을 아주 일반적인 것으로 아마도 소프트웨어를 개발하는 사람이라고 한다면 “유치원에서 배웠다”는 듯이 이야기를 많이 한다. 하지만, 현실에서는 지키기 어렵다는 이유로 실무에서 간과되는 것이다. 마치 유치원 때 배웠던 건널목을 건너는 방법일 잊어버리고, 빨간불이라도 차가 오지 않으면 무단횡단을 하는 것과 같다. 현실과 타협하게 되면 절차와 방법은 의미를 잃어버리게 되며, 지키지 않는 규칙은 규칙이 없는 것 보다 못한 결과를 만들게 된다(불신과 비협조가 나타나게 되므로).

툴(tool)의 도움을 받고 싶다면 현재하고 있는 업무나 결과물을 잘 분석해서 쓰고자 하는 툴에 맞게 변경하는 것이 좋다. 사람들은 흔히 자신이 익숙하게 쓰던 툴을 새로운 것으로 바꾸는 것을 싫어한다. 신기술을 개발하는 사람도 예외는 아닌데, 이것은 몸에 잘 맞는 오래된 옷을 버리고 새로운 옷을 입었을 때의 어색함과 같다. 처음 산 신발이 발에 잘 안 맞아서 여기 저기 발에 물집이 잡히지만, 조금만 지나면 적당히 늘어나고 딱딱한 가죽이 부드러워지듯 새로운 툴도 적응하는 데는 반드시 시간이 필요하다. 일하는 방식을 바꿔야 할 필요도 있고, 작성하고 있는 소스코드의 배치도 변경해야 할 때도 있다. 중요한 것은 새로운 툴을 도입했다면 그 툴에 대해서 충실하게 사용할 수 있도록 자신을 바꾸는 것이다. 물론, 현실에서는 옷을 바꾸는 것이 좋겠지만, 도구는 그 도구를 사용했을 때 얻어낼 수 있는 효과를 최대로 끌어내도록 나를 바꾸는 것이 더 현명하다.

끝없이 불평하기보다는 새로운 툴을 익히는 즐거움을 오히려 맞보기를 권한다. “ClearCase”를 처음 쓸 때 느꼈던 것이 “Perforce”를 쓸 때라고 달라지지는 않는다. 신기한 기능과 그 툴 만이 갖추고 있는 장점들을 맛보려고 노력하는 편이 오히려 즐겁다. 그렇지 않으면 인프라(Infra)를 담당하는 사람들과 빈번하게 부딪히는 것을 감수하고 살아가는 수밖에는 별다른 방법이 없다.

소프트웨어도 사람이 만드는 것이기에, 사람에게 가장 중요한 동기와 비전, 노력에 대한 인정과 존중하는 마음을 서로에게 가져야 한다. 예를 들어, 새로운 신입사원이 들어왔다고 가정하면, 첫 모습은 정말 부족해 보이는 것이 많을 것이다. 이것도 모르고 저것도 모르고, 항상 허둥지둥하고 어떻게 일을 처리할지 몰라서 시간만 낭비하는 것처럼 보일 것이다. 이런 신입사원들이 필요한 것은, 어찌 보면 선배들에게는 아주 사소한 것이지만, 잠시 시간을 내어 어떻게 하는지 직접 보여주고, 필요한 문서를 찾아서 갖다주고 설명을 곁들여 주는 것이다. 이번 과제를 통해서 배울 수 있는 것들에는 어떤 것들이 있으며, 몇 년 후에는 어떤 사람이 될 수 있다는 것을 보여주는 것, 작은 것이지만 열심히 해서 숙제를 검사 맞는 학생처럼 조마조마한 순간에 따스한 커피 한잔과 말 한마디가 그 사람의 인생을 좌우할 수 있다. 야단치는 것은 잠깐 효과가 있지만, 칭찬하는 것은 오랫동안 기억에 남는다. 툴이 아무리 좋아도 직접 사람과 사람 사이의 대화보다 좋은 것은 아직 발명되지 못했다고 생각하기 때문이다.

특정 도메인에 치우치지 않더라도 모든 소프트웨어 개발에는 공통적인 것들이 있다. 예들 들어, 개발절차나 방법론과 같은 큰 영역에서부터, C언어를 사용할 때는 포인터(Pointer) 연산에 주의해야 한다는 것과 같은 세밀한 것까지 다양한 것들이 있다. 결국 소프트웨어 개발자는 이러한 공통적인 것들이 자신의 자산으로 남아있게 된다. 다양한 분야를 옮겨 다니면서 여러 가지 일을 하게 될 때, 해당 분야의 도메인 지식은 항상 바뀌게 되고, 결국, 남는 것은 “문제를 해결하는 방법과 절차”다.

물론, 한 분야에서 오랫동안 일한다고 해서 이러한 것이 없다는 것은 아니다. 오랫동안 동일한 일을 하게 되면 결국에는 추상적인 것들이 머릿속에 남게 되고, 사실 이러한 것이 추상화되고 정제되면 방법과 절차라는 것으로 남게 된다. 중요한 것은 그 틀에 익숙해져서 묻혀 살기 보다는 그 틀을 나름대로 바꿀 수 있는 능력까지도 갖추라는 것이다. 어떤 툴을 사용하든 목적을 이룰 수 있다면 상관없다. 소프트웨어 개발자로 열심히 일하고 있는 동료들에게 조금이라도 도움이 될 수 있다면 불합리하다고 생각되는 것들은 바꿀 필요있다. 뭔가 새로운 것을 해보려고 노력하는 부하 사원이 있다면, 그 결과에 대한 두려움을 안겨주지 말고 어떻게 그것을 잘할 수 있는지 방법을 같이 찾아보기 바란다. 급한 일은 급하게 처리해야 하지만, 결과적으로 남는 것은 그 일을 한 경험과 일에서 발견한 나름의 “성공하는 방식”이 될 것이기 때문이다.

# [ 낙관과 비관 사이 ]

어떤 프로젝트를 하게 될 때, 우리는 계획을 낙관적으로 세워야 할까? 아니면 비관적으로 보는 게 옳을까? 계획은 비관으로 세우고, 일은 낙관적으로 해야 목표를 이룰 수 있지 않을까? 모든 프로젝트 관리자들은 나름의 방법이 있겠지만, 어쨌든 목표를 이루기 위해서 어떤 식으로 든 추정(Estimation)할 수밖에 없다. 여기서 모든 고민의 문이 활짝 열린다.

프로젝트에서 추정은 일정, 비용, 필요한 자원(사람을 포함해서)등을 얼마나 사용할 것인지를 정하는 활동이다. 관리자들은 최적의 일정을 요구하며, 그것을 “동의”라는 이름으로 강제화 시키기를 원한다. 과제 담당자는 최선을 다해서 가장 늦춘 일정을 만들기를 원하며, 가능한 많은 버퍼(Buffer)를 추정에 반영하려고 한다.

따라서, 관리자는 낙관하는 경향이 있고, 과제 담당자는 비관적으로 계획을 수립할 수밖에 없다. 하지만, 그 누구도 과제가 “정말 언제 끝날지”는 아무로 모른다는 점이다. 즉, 아무리 완벽한 추정을 하더라도 바뀌는 게 일반적인 현상이다. 지나치게 낙관할 필요도, 지나치게 비관할 필요도 없는 것이다. 그냥 낙관과 비관 사이에 얼마나 과제가 끝날 수 있을지 확률로 대답하는 것으로도 충분한 일이다. 실제로도 예상은 항상 4배에서 0.25배까지 변할 수 있다는 것이 이론적인 결론이기도 하다.

# [ 벽을 넘어서 ]

하는 일이 그런지 맨날 남의 부서에서 하는 것들을 가져와서 이리저리 만지고 다듬어 나름 더 좋은 구조를 만들어가고 있다. 하지만, 어떻게 보면 아주 성가시게 사람을 귀찮게 하기도 하고, 그 사람들이 열과 성을 다해서 만든 작품을 비평하기도 해야 한다. 한 두 번은 괜찮더라도 이러한 것들이 반복적이다 보면, 그 사람들과 함께 일하기가 불편해지는 것은 당연지사다. 누가 자신이 만든 것을 가지고 이래라 저래라하는 것은 솔직히 모든 사람이 싫어한다.

하지만 어쩌겠는가? 회사의 목표는 “더 좋은 것은 내일에 있다”는 가정 하에 일을 추진해야 있으며, 뭔가를 끊임없이 개선해야 하는 것도 어쩔 수 없는 일이다. 소프트웨어 개발은, 지금 없다면 만들고, 부족하다면 채워주고, 남으면 걷어내는 주는 것이, 마치 나무를 키우는 조경사처럼 꾸준히 관리해 주지 않으면 안 되기 일이기 때문이다.

보통 품질보증(Quality Assurance)나 소프트웨어 공학(Software Engineering)관련 분야에서 일하는 사람들이 이런 일을 하지만, 개발자로 일하는 경우에도 이런 활동을 하는 사람이 필요하다. 하지만, 매일 같이 개발자를 만나서 이런 이야기를 하기란 쉬운 일이 아니다. 앞에서는 받아들이는 듯 미소를 짓다가, 돌아서서 험담하는 것은 어쩔 수 없다. 뒷통수가 가려운 말도 흘려들으면 그만이다. 그렇다고 하더라도 사람들이 가지는 불만은 “긍정적인 피드백”이 될 수도 있다.

절대 잊지는 말자. 우리는 어떤 형태로든 동업자이며, 회사의 사업이 번창해야 자신의 주머니를 가득 채울 수 있다는 사실을. 깨진 그릇으로는 물을 담을 수 없듯이, 깨진 조직(벽을 쌓고 일하는 조직)은 제대로 된 제품을 만들지 못한다. 벽을 허무는 제일 좋은 방법은 “소통”이며 최악의 상태는 단절(Silo)이다. 어렵게 일하지 말고 쉽게 일하자. “어려운 것은 내가 해야 할 과제지 내가 만나는 사람들은 절대 아니다.”

# [ 품질과 기능 그리고 자원의 문제 ]

윗 사람들은 항상 빨리하라고 이야기한다. 빨리하고, 품질도 완벽해야 하고, 시장에서 원하는 모든 기능이 다 들어가야 한다고 말한다. 하지만, 그런 일이 현실에서 가능할까? 빨리 만들면 실수할 가능성도 크다. 품질이 완벽해지려면 테스트하는 시간도 길어진다. 시장에서 원하는 모든 기능을 다 구현하기 위해서는 사람도 더 많이 필요하며, 더 오랜 시간 개발해야 한다. 결론적으로 불가능하다는 뜻이다. 어떤 한 부분을 강조하면 다른 부분이 줄어들거나 더 커지게 된다. 마치 물 풍선의 한 쪽을 움켜쥐면, 다른 쪽이 커지는 것과 같은 원리다(물론, 반비례 관계도 있지만).

어리석은 사람들이라는 생각을 하면서도, 한편으로는 “그들도 분명히 나이가 어렸을 때는 개발자로 일했을텐데 왜 그렇게 말하지?”라는 의문이 든다. 자원이 없다면(사람이 없거나, 시간이 없다면) 당연히 기능은 축소되어야 하고, 혹은 기능을 다 구현하기 위해서는 품질은 당분간 잊어버리는 것이 정신 건강에 도움을 준다. 즉, 품질과 기능은 서로 반비례 관계이고, 또한 품질이나 기능은 자원에 비례하기 때문이다. 품질이 좋으면서도 기능이 많은 코드는 시간이 오래 걸리고, 사람이 많이 투입되어야 가능하다. 하나를 고정해놓는 상태에서 다른 것을 늘리기 위해서는 근본적인 개발 방법의 개선이 없이는 불가능하다. 하지만, 대부분 개발자는 자신의 개발 방법을 쉽게 바꾸지 못한다. 따라서, 결국 일정은 지연되고, 원하는 품질의 확보는 요원한 일이 되고마는 것이다.

어째서 이런 일이 자주 발생하는 것일까? 이것은 사실 누구의 잘못도 아니다. 각자 자신이 놓인 상황에서 최선으로 일한다고 가정하기 때문이다. 옳고 그름의 문제가 아닌, 적절한 수준에서 타협 해야하는 문제이기 때문이다. 뭔가를 포기하지 못하는 우둔함을 가지고 일하게 되면, 그 일을 직접 담당하고 있는 사람들의 마음을 잃어버리게 된다. 차츰 이러한 것이 쌓이게 되면 더는 상대방에 대한 기대를 접게 되고, 결국에는 시키는 일만 하는 개발자와 보고자 하는 것만 보는 관리자가 되고 만다. 누구의 잘못도 아닌 일로 인해서 이렇게까지 하면서 회사에 다녀야 할까?

아마 능력 있는 사람이라면 회사를 조만간 옮길 확률이 높고, 참을성 많은 사람이라면 한 귀로 듣고 한 귀로 흘려보내는 비능률적인 일이 발생하게 될 것이다. 기본적으로 우리는 같은 이해를 바탕으로 항상 역지사지를 해야 한다. 관리자는 일정을 단축함으로써 얻는 시장선점의 비용이 기능의 축소나 품질의 만족보다 큰가를 따져야 할 것이며, 개발자는 관리자가 원하는 일정을 맞추기 위해서 온 힘을 다해야 한다. 서로 신뢰할 수 있는 분위기는 상대방에 대한 만족 정도에서 나오기 때문이다.

능력 없는 관리자는 일정만 살피고, 불평 많은 개발자는 이런저런 부족한 것들만 이야기한다. 이런 사람들이 많은 회사라면, 아마 조만간 감사를 받거나, 혹은 성장이 없는 정체에 빠질 가능성이 농후하다. 선행 개발이라는 것을 말로만 하고, 사실상 상품화에 집착하는 회사는 결국 미래의 먹거리를 당겨와서 오늘의 배를 채우고 있을 뿐이다. 그나마도 다 없어지고 나면, 더 이상의 발전은 불가능하다. 미래는 현재가 투사되어 나타나는 것인데, 제대로 된 현재를 가지고 일하지 못하는 상황에서 미래는 보나마나 암울할 것임이 분명하다.

문제를 발견하기는 어렵지 않다. 오히려 문제는 이미 다 알고 있는 경우가 많다. 다만 그것을 어떻게 고칠 것인가를 모르는 것이 더 큰 문제일 뿐이다. 물건은 만들어서 팔아야 돈이 되고, 사람도 그 돈으로 더 고용해야 필요한 자원(?)을 확보할 수 있다. 확보한 자원은 다시 개발에 투입되어 더 좋은 품질의 제품을 만들 수 있기에, 돌고 도는 “빈익빈 부익부” 문제가 되고 만다. 마치 인생의 윤회 속에 고뇌를 가진 인간처럼, 끝없어 보이는 업무 속에서 해결책을 찾지 못하고 있는 것처럼 보인다.

그럼 어떻게 해야 이러한 문제를 해결하는 시작을 할 수 있을까? 단적인 방법은 기존에 개발한 제품을 다시 리팩토링 하거나, 최소의 변경으로 품질을 개선해서 후속 제품을 만들어내는 것이다. 즉, 최소의 비용으로 최대의 효과를 얻을 수 있는 일들을 찾아야 한다. 이 때는 재활용이라는 그럴싸한 말 밖에는 다른 용어가 없다. UI만 변경하는 것도 한 가지 방법일 것이다. 사용자가 정말 사용하기 좋은 UI를 제공하는 것만으로도 큰 기능의 추가 없이 일단의 만족감은 줄 수 있다. 호환성을 개선하는 것도 좋다. 기존의 제품이 몇몇 제품과만 호환된다면, 차기 버전에서는 더 많은 호환성을 가지면 되는 것이다.

핵심은 개발에 “여유” 찾는 것이다. 과제 수를 줄이고, 기능을 줄이고, 코드의 변경을 줄이는 것이다. 여유가 없이는 개선도 불가능하다. 꾸준히 개선하지 않으면 “열화”되며, 관리해야하는 복잡함도 악화되는 것이 일반적인 소프트웨어가 가지는 속성이다. 과거의 비용만 줄이기 위해서 하는 “버그 수정”으로는 이런 소프트웨어가 가지는 속성에 대한 해결책이 아니다. 과거의 유산만 쥐고 살려고 애쓰다보면 새로운 것은 찾지 못하게 되며, 시장 경쟁력은 당연히 뒤쳐질 것이다. 코드를 개선할 수 있는 여유를 찾지 못하고 계속 앞만보고 달린다면, 코드 속에 쌓여가는 쓰레기를 치우는 것도 큰 부담으로 밖에 생각되지 않을 것이다.

# [ 코드리뷰는 어떻게 하지? ]

일반적으로 코드리뷰가 가장 많은 버그를 잡을 수 있다고 이야기한다. 하지만, 그만큼 사람들의 시간을 잡아먹는 활동도 드물다. 물론, 나중에 발생할 버그를 고치는데 시간을 들이는 것보다 나을 수는 있지만, 그래도 부담스러운 것은 사실이다. 그렇다면, 사람들이 부담스럽다고 생각하는 것에는 무엇이 있는지 먼저 알아보도록 하자.

**01. 사람들이 시간을 내서 모여야 한다.**

**02. 내(혹은 다른 사람이)가 작성한 코드를 리뷰한다.**

**03. 무엇을 리뷰해야하고 어떻게 리뷰 해야할지도 모른다.**

개발자의 시간을 뺏는 것은 아주 큰 손실이다. 특히 몸 값이 비쌀수록 시간의 낭비는 받아들이기 어렵다. 더구나 만약 프로젝트가 QA(Quality Assurance: 품질 확인)과정를 겪고 있다면, 수많은 버그 속에서 하루하루 생활하기도 힘들어 죽을 지경일 것이다. 그런 상황에서 남의 코드까지 보면서 자신의 시간을 낭비하기(?)는 정말 싫을 것이다. 먼저, 자기 발등에 떨어진 불부터 꺼야만 여유를 가지고 남의 코드를 볼 수 있다. 또한, “모여야 한다는 것”도 부담스럽다.

특정 시간에 할 말도 없는 회의에 모여 앉아서, 남들이 하는 이야기만 들으며 졸리는 시간을 눈뜨고 지키고 있는 것 자체가 힘들다. 물론, 코드리뷰를 할 때 항상 모여서 이야기를 해야하는 것은 아니다. 온라인 코드리뷰가 일상화된 팀은 하루의 일부 시간을 코드리뷰를 위해서 할당하고, 그 시간만큼은 개인의 시간을 철저하게 보장하기도 한다. 그리고, 서버를 통해서 코드에 대한 수정 사항을 기록으로 남기기에, 사람들이 모여 앉아서 서로 코드에 대한 의견을 주고받는 시간이 흔하지 않을 수 있다.

자신이 작성한 코드를 리뷰를 받는 것은 학교에서 숙제를 제대로 했는지 확인당하는 느낌일 수도 있다. 즉, 자신이 만든 결과물에 대한 장단점을 이야기를 해야 한다는 점에서 부담스러울 수밖에 없다. 부담되는 일은 피하는 것이 인지상정이다. 따라서, 좀처럼 자신이 만든 코드를 리뷰 받기를 싫어한다. 특히 선임 개발자나 혹은 개발자로서의 어느 정도 경력을 쌓은 사람들은 다른 사람이 자신의 코드에 대해서 뭔가 이상이 있다고 이야기하는 것을 듣기 싫어한다. 자기보다 능력이 못하다고 생각하는 상대에게서 듣는 충고를 달가워할 사람은 없을 것이다.

코드리뷰에서 모든 코드를 세세한 사항까지 전부 리뷰하는 것은 너무나 많은 시간이 걸린다. 그렇다고 그냥 넘기기에는 꺼림직한 면이 있고, 많은 사람이 모인 자리에서 시시콜콜한 모든 문제를 끄집어내는 것도 남들에게 피해를 주는 것 같을 것이다. 무엇을 어느 정도 리뷰하는 것이 좋을지 몰라서 그냥 그때그때 생각나는데로 이야기하고, 어떤 부분은 그냥 넘어가고 어떤 부분은 너무 자세히 리뷰하는 것도 부담스럽다. 듣는 사람이 기분이 안 좋을지도 모른다는 생각에 일단은 간단한 오류는 그냥 넘어가 보지만, 그런 것들이 너무 자주 보이다 보니, 솔직히 코드를 리뷰해도 별로 좋아질 것 같이 안보이기도 한다.

그렇다면, 이런 것들을 극복할 수 있는 방법은 무엇일까? 아주 간단한 “사실에 근거한 코드리뷰”를 하는 것이다. 즉, 코드를 보라는 것이지 그 코드를 작성한 사람의 됨됨이를 보라는 것이 아니다. 또한, 투자하는 시간을 줄이기 위해서 자주 짧게 코드리뷰를 할 필요가 있다. 모든 사람이 모일 필요없이 해당 코드에 관련된 사람들만 모이는 것이 효과적이며, 코드 자체가 이해하기 어렵다고 생각된다면, 이해하기 쉬운 코드를 먼저 만들라고 유도하는 것이다. 사실에 근거한 리뷰를 위해서는 “체크 리스트”를 만들 필요가 있다. 즉, 객관적인 리뷰 기준을 미리 정의하고 그것을 위주로 코드리뷰를 실시하는 것이다. 체크 결과는 개인에 대한 비판이 아니라, 기준을 가진 잣대를 적용하는 것이다. 따라서, 받아들이는 사람의 입장에서도 기분 나쁘게 생각할 필요가 없다.

코드리뷰를 실시하는 시점도 중요하다. 주로 저장소에 커밋(Commit)하기 전이나, 혹은 커밋 이후 통합이 완료된 코드를 통해서도 할 수 있다. 통합이 되지 않은 코드의 리뷰는 나중에 통합 이후에 변경될 가능성이 있다는 점에서 효과적이지 않을 수 있다. 커밋 이후에 하는 코드리뷰는 통합은 문제가 없었지만, 리뷰 이후에 코드의 수중이 있을 수 있다는 점에서 불완전한 리뷰가 될 가능성이 있다. 따라서, 한번의 코드리뷰도 리뷰로는 리뷰 이후의 변경에 대해서 대응하지 못한다. 코드리뷰 이후에 변경된 코드에 대해서도 당연히 리뷰하는 것이 정답이며, 될 수 있으면 통합 전에 리뷰를 완료하는 것이 좋다.

개발자들은 코드리뷰 과정이 빠르게 진행되는 개발을 지연시킬 수 있다는 점을 근거로, 통합 전 코드리뷰를 거부할 가능성도 있다. 이럴 때는 적절한 합의를 통해서 통합 후에 변경된 코드에 대해서 코드리뷰를 실시하는 것이 더 효과적일 것이다. 개발자의 흐름을 방행하고 인위적인 방해처럼 느끼게 만든다면, 오히려 형식적인 코드리뷰로 흘러갈 가능성이 높기 때문이다. 따라서, 최대한 개발자의 의견을 반영한 코드리뷰를 실행하는 것이 개발자의 동참을 유도해 줄 것이다. 하지만, 그렇다고 아무렇게나 코드리뷰를 하도록 내버려두라는 말은 아니다. 제대로 코드리뷰를 실행하고 있는지 꾸준이 관찰(Monitoring)해야 할 것이다.

이러한 것을 하기 위해서는 자동화된 인프라(Infra)를 구축하는 것이 좋다. 사람들이 쉽게 사용할 수 있는 시스템을 구축하고, 그곳에 리뷰 요청과 리뷰 결과 및 History등을 남겨야 한다. “ReviewBoard”, “Geritt”, “Cruicible” 혹은 “Code Collaborator”와 같은 툴을 이용해서 쉽게 구축할 수 있다. 리뷰의 내용은 모든 사람이 공유할 수 있도록 해야한다. 비슷한 문제나 동일한 상황을 만났을 때 어떻게 대처할 수 있는지 알 수 있기 때문이다. 아무리 작은 차이라도 쌓이면 큰 결과를 만든다. 아무리 작은 코드라도 쌓이면 시스템을 망가뜨릴 수 있다. 따라서, 사소한 변경도 모두 리뷰의 대상이다. 그리고, 중요한 코드라고 생각되는 부분은 온라인 보다는 오프라인 형태의 코드리뷰가 더 효과적으로 문제를 찾는데 도움이 될 것이다.

리뷰를 요청하는 대상은 관련된 일을 진행하는 사람으로 처음에는 한정하는 것이 좋을 것이다. 그래야 관련된 변경에 대해서 즉시 피드백(Feedback)을 정확하게 얻을 수 있기 때문이다. 하지만, 반드시 그래야 할 필요는 없다. 전혀 관련된 부분이 아니라고 하더라도 때로는 도움을 줄 수 있는 경우도 있기 때문이다. 예를 들어, 코드의 가독성이 부족한 경우를 알기 위해서는, 오히려 코드에 익숙하지 않은 사람이 도움이 되기도 한다. 너무 잘 알고 있는 사람은 자신의 코드에 어떤 약점이 숨어 있는지 정확히 보지 못하는 경우가 종종 있기 때문이다. 그리고, 어차피 시스템은 모든 코드가 함께 동작하기에 모든 개발자가 다 이해해야 할 부분은 어느 정도는 항상 있기 마련이다.

리뷰 요청은 반드시 두 사람 이상이 완료해야지 코드리뷰를 완료하는 것이 좋다. 특정 한 사람만이 리뷰하는 것보다는 여러 사람이 같이 리뷰하는 것이 효과적이기 때문이다. 완료를 해야 하는 사람의 수는 프로젝트의 성격에 따라 달르겠지만, 2~5명 정도가 적당하다고 본다. 즉, 너무 많으면 시간이 오래 걸려 개발진척이 더딜 것이고, 너무 적으면 소수의 의견에 의해서 영향을 받기 때문이다. 모두가 다 리뷰를 하지만, 그 중에서 반드시 리뷰 요청을 확인해서 처리해야 하는 사람은 한정하는 것이 좋다.

또한, 리뷰하는 사람들과 리뷰를 요청하는 사람을 항상 고정적으로 주고받는 식으로 만들기보다는 순환적인 관계를 유지하는 것이 좋다. 예를 들어, A가 B에 리뷰를 요청했다면, B는 C에서 요청을, C는 D에게, 다시 D는 A에게 리뷰를 요청하는 것이 좋다. 직급의 상하 관계에 상관없이 만들고, 관련된 지식을 순환적인 고리를 이용해서 전파할 수 있기 때문이다. 직급의 상하 관계가 리뷰에 영향을 줄 수 있으며, 고정된 관계는 항상 내부에서만 지식이 전파되도록 만든다. 리뷰에 참가하는 모든 팀원이 지식을 나누고 확장시킬 수 있도록 해 주어야 한다.

리뷰의 대상은 모든 코드라고 했다. 리뷰에서 첫 번째로 볼 것은 가독성(Readability)이다. 즉, 코드가 자연스럽게 읽히는지 파악하는 것이다. 잘 읽힌다면 문제가 있더라도 쉽게 고칠 가능성이 높은 코드이고, 잘 읽히지 않는 코드라면 그 반대일 것이 분명하기 때문이다. 코딩 스타일과 같은 것은 요즘 좋은 편집기들이 많이 있기에, 그냥 자동으로 맞춤 기능을 이용하면 충분히 스타일을 전체적으로 적용할 수 있다. 다음으로 중요한 것은 로직(Logic)상의 문제를 잡는 것이 바람직할 것이다. 오류 처리나 복구와 같은 부분도 리뷰의 대상이 된다. 단위 테스트(Unit Test)도 하고 있다면, 커버리지(Coverage)와 결과를 확인하는 것도 좋다.

시스템의 가장 약한 고리가 전체 시스템의 품질 척도가 된다는 사실은 이미 다 알려진 사실이다. 조금이라도 약한 고리가 있는지 확인하고, 이를 고치는 것은 시스템을 개발하는 모든 개발자의 몫이며, 관련된 몇몇 사람만이 담당할 일은 아니다. 코드리뷰의 목적은 단순히 버그를 찾아내는데 그치지 않고, 더 좋은 코드를 만드는 기술과 경험을 나누는 활동이다. 따라서, 코드리뷰는 개발 시간의 낭비가 아니며, 오히려 문제를 조기에 발견해서 해결할 수 있는 방법인 것이다. 대충하려고 한다면 오히려 낭비로 이어질 가능성이 높기에, 해야 한다면 반드시 정확한 방법과 절차를 정하고 꾸준히 해야 한다. 자신이 개발한 코드라고 자신만의 것이 아니며, 남의 코드에 대한 의견을 주는 것은 효과적인 피드백으로 상대방에게 도움을 주는 활동이라는 점을 알아야 할 것이다.

# [ 지속적인 통합(CI)이란? ]

한국 사람은 조직력이 강하다는 말을 많이 들었을 것이다. 축구 경기를 보면 해설자들은 한국 축구의 강점으로 조직력과 체력을 항상 이야기한다. 과거 히딩크 감독이 월드컵 대표팀의 감독을 맡았을 때, 기자들의 한국 축구의 가장 약한 점이 무엇인지에 대해서 질문을 받았을 때 "체력"이라고 말했다고 한다. 체력을 다져야 기술이 나오고, 제대로 된 축구를 할 수 있다는 점을 강조한 것이다. 그전까지 한국 축구 경기를 하면 체력과 근성이라고 생각했던 기자들은 다소 의외의 답변에 조금 당황하기는 했지만, 어쨌든 월드컵 4강이라는 기록을 남긴 것은 사실이다. 그 후 부터 한국 축구는 강한 체력을 바탕으로 한 압박 플레이에 익숙해졌고, 아직도 그런 경향을 가지고 있다.

이제 한국 축구의 강점인 남은 한 가지, “조직력”을 보도록 하자. 정말 조직력이 강할까? 조직력이란 오랜 훈련의 결과로 나온다. 하지만, 프로선수로 전 세계에서 뛰다가 갑자기 국가대표로 소집 받아서 돌아온 선수들이 이런 조직력을 제대로 만들어 낼 수 있을까? 가끔 보이는 경기력에 대한 실망의 원인은 바로 조직력이 제대로 갖춰지지 않기 때문에 생기는 현상이라고 본다. 축구는 1명이 뛰는 경기가 아니라, 11명이 골고루 제대로 뛰어야지 결과를 만들어내는 경기이기 때문이다. 또한, 벤치를 지키는 대체 선수의 여유가 없이는 결코 시즌을 잘 보내기 힘든 것도 마찬가지다(물론, 리그 경기에 한정된 이야기라고 볼 수 있지만, 단기간에 많은 경기를 소화해야 하는 토너먼트 같은 게임 방식도 고른 실력을 갖춘 대체 선수를 얼마나 확보하는가에 따라 다른 결과가 날 수 있다).

소프트웨어 개발도 마찬가지다. 한 사람의 개발자가 아무리 뛰어나다고 해도 혼자서 모든 것을 다 해결할 수는 없다. 대부분 프로젝트는 팀이 아니고선 끝내기 어려운 것들이 많다. 혼자서 학교에서 내주는 숙제를 하는 것과는 차원이 다른 노력이 필요하다. 이처럼 팀으로 일을 하게 될 때는 반드시 점진적인 통합(CI: Continuous Integration)이 필수다. 모든 팀원이 개발한 코드를 조금씩 지속적으로 통합해서 테스트를 자동으로 실행하는 것이 안 된다면, 결과적으로 주어진 예산과 일정 내에서 과제를 끝내는 것은 우연에 가까운 일이 된다. 하지만, 현실의 개발자는 생각보다 이런 것들이 하찮은 일이라고 생각하는 경향이 있다. 자신의 코드가 완성되어야지만 통합하겠다는 생각이 강하다. 부족한 부분이 있다면 통합을 미루고 일주일이고, 이주일이고 시간을 보내다가 완성된 시점에 전체 코드와 통합하면 된다고 생각한다.

물론, 이렇게 개발한 코드가 잘 테스트가 되어있고, 거의 인터페이스의 변경도 없으며, 다른 코드와 의존성 관계도 없다면, 크게 문제될 것이 없을 수도 있다. 하지만, 대부분 코드는 다른 코드에 대해서 의존성이 높고, 개발 중인 코드라 인터페이스의 변경도 아무런 통보 없이 이루어 진다. 단위 테스트라도 잘되어 있으면 좋겠지만, 모듈화가 안 된 코드를 단위 테스트한다는 것은 사실상 거의 불가능하다. 현실의 개발자는 자주 이야기하고, 밖에 나가서 술이나 한잔 하거나 친하게 사적인 활동을 하면 팀워크가 잘 된다고 생각할지도 모른다. 하지만, 정작 자신이 하는 일에 대해서 팀원들에게 별로 도움을 주지 못하면서도, 일 외적인 부분에서 만 팀워크를 챙기는 모순이 생긴다.

지속적인 통합은 팀워크를 기르기 위한 첫 걸음이다. 꾸준히 팀원들이 개발한 코드를 통합하고, 테스트하고, 결과를 각자에게 피드백을 주어서 개선하게 하는 것이야 말로, 가장 기본적인 개발 활동에 필요한 팀워크인 것이다. 상대방에 대한 배려로 항상 빌드(Build)가 가능한 코드를 유지해주고, 남들이 체크인(Check-in)한 코드와 충돌(Conflict, Collision)를 일으키지 않도록 자신의 코드를 수정하며, 필요하면 즉각적인 테스트를 실행해서 상대방에서 자신이 작성한 코드에는 오류가 없으니 믿고 사용해도 좋다고 보장하는 일련의 활동이다.

한꺼번에 많은 코드를 커밋(Commit)해서 전체 빌드 시스템에 커다란 오류를 유발하고, 그로 인한 버그로 시스템 전체 코드의 버그를 뒤지는 어려운 일을 발생시키게 된다면, 이것은 개인의 문제를 넘어서 팀원 전체의 시간을 생산적이지 못한 일에 소모하도록 만든다. 결코, 유쾌한 마음으로 일찍 퇴근하는 것과는 상당히 먼 이야기가 되어버린다. 한번의 실패는 크게 문제가 되지 않지만, 반복적인 실패가 발생하고 있다면, 시스템적으로 해결해야 한다는 뜻이다. 통합은 개별 활동의 응집을 통해서 더 큰 문제를 해결하는 방법을 찾아나가는 행동이다. 따라서, 통합에 지속적으로 문제가 따른다면, 전체 시스템을 개발하는 활동도 느려질 수밖에 없다. 이런 문제가 자주 발생한다면, 통합을 더 자주 빨리 할 수 있는 체계적인 방법을 고안해야만 한다.

소프트웨어 개발에서 필요한 체력은 “전문가적인 지식”에서 나온다. 즉, 얼마나 관련 분야에 대한 지식 수준을 갖추고 있는가에 의존한다. 혼자서 배울 수 있는 부분은 한정된 시간에서 한정된 분야에 치우칠 수밖에 없다. 시스템은 다양한 지식을 가진 사람들이 모여서 만든 것이기에 전체적인 시스템을 다 이해하기 위해서는 남의 도움이 절실하다. 이런 상황에서 위와 같은 팀워크가 잘 갖춰진 팀이라면, 당연히 코드리뷰와 같은 활동이 활성화되어 있을 것이며, 누가 실수했든 상관없이 쉽게 이전의 상태로 돌아가서 수정할 수 있을 것이다. 이런 시간이 모여야 축적된 시스템에 대한 지식이 쌓일 것이고, 그 과정을 통해서 소프트웨어 개발자로서의 체력은 저절로 좋아질 것이다. 작은 반복은 그 일을 하는데 필요한 노력의 절대치를 줄여줄 것이며, 의식하지 않더라도 자동적으로 필요한 일을 할 수 있도록 만들어 준다.

# [ 글쓰기 연습이 필요하다. ]

애자일(Agile) 개발 방법론에 대한 많은 책이 시중에 나와 있다. 그 중 몇 권을 사서 읽어보면 대체로 전부 외국에서 개발했던 경험을 토대로 이야기를 전개하고 있다. 한국적인 사고의 토대에서 애자일 방법을 써서 개발한 성공담(“Best Practice”라고 해야 할까?)은 찾아보기가 힘든 것이 사실이다. 왜 그럴까? 개인적인 추측이지만, 우리나라의 개발자는 글쓰기 역량이 부족하다는 점을 이야기하고 싶다. 자신이 쌓은 경험은 머리 속에만 남겨두고, 글로 정리하지 않는 것이 일반적인 개발자들의 글쓰기에 대한 태도다. 물론, 모든 개발자가 그런 것은 아니지만, 대체로 개발자들이 글쓰기를 싫어하는 것은 사실이다.

고객과의 대화에서 알게 된 사용자 시나리오를 통해서 설계하고, 이를 코드로 수만 라인을 짜는 사람도 정작 블로그에는 몇 개의 사진만 달랑 있고, FaceBook이나 Twitter등에는 140자도 안 되는 글만 하루에 2~3개씩 올리는 게 전부다. 수만 라인의 코드에 대해서는 정확히 이해하면서 어떻게 1페이지 분량의 문서를 쓰는데는 약할까? 근본적인 문제는 아마도 글에 대한 지나친 두려움이 아닐까? 예전에 이런 이야기를 했던 적이 있다. "대학에서 국문학 시간에 교수님이 질문을 던졌다. 글의 3요소가 뭡니까?” 나를 지목하고 던진 질문에 아무 생각 없이 “글을 쓰기 위한 시간과 필기구, 그리고 쓸 수 있는 노트 정도?” 그 대답에 교수님은 잠시 생각을 가다듬고 나서, “흠 그것도 맞는 대답이군."

사실 글을 쓰는 것을 아주 쉽게 시작해야 한다. 처음에는 블로그를 통해서 글을 쓰는 연습을 하는 것도 좋은 방법이다. 자신의 일기를 적는 차원에서 시작한 글은 어떤 주제를 찾아 나가게 된다. 주제를 찾는다는 것은 그것에 대해서 생각을 정리할 수 있는 지식과 경험, 그리고 여유가 있다는 말이 되며, 또한 정리된 생각은 더 높은 수준에서의 응용이 가능하다는 것이다. 수준 높은 글이 아니더라도 일단 글은 써야 한다. 채팅하듯이 자신에게 이야기하는 글도 좋고, 특정 전문적인 주제를 가지고 자신이 아는 한도 내에서 적는 것도 좋다. 해야 할 것은 글을 쓰는 것이지, 반드시 좋은 글을 적어야 하는 것은 아니다.

좋은 글은 많이 써 본 사람이 쓴다. 소설가는 생업이 글쓰기 이기에, 많이 읽고, 쓰고, 수정한다. 물론, 창조적인 부분은 보충하기 어려운 것이므로 개인의 성향에 따른 문제이지, 우리가 쓰는 글에 창조적인 것까지 처음 부터 기대할 필요는 없다. 대부분의 개발자가 쓰는 글은 기술에 대한 것이며, 창의적인 부분보다는 사실을 전달할 목적으로 작성된다. 따라서, 창의성은 나중을 위해서 잠시 뒤로 미뤄두어도 상관없다. 그렇다고 좋은 글을 쓰지 말라는 것도 아니다. 좋은 글이란 무엇인가 전달할 내용을 명확하게 이해하기 쉽도록 쓴 것일 뿐이다. 우리는 소설속의 인물을 새롭게 창조해야 할 필요도 없으며, 있는 그대로의 사실을 자신의 해석을 통해 글을 읽는 사람에게 지식으로 전달하는 일일뿐이다.

그럼, 소프트웨어 개발자로서 어떤 글을 쓰는 것이 좋을까? 소프트웨어 개발자가 써야 하는 글은 크게 두 가지로 볼 수 있다. 하나는 문서로 전달되어야 하는 것이며, 다른 하나는 실행되는 코드에 넣어야 할 것으로 나누어진다. 문서로 전달되어야 할 부분은 기본적인 설계와 중요한 부분에 대한 자세한 설명(혹은, 결정한 이유) 정도가 될 것이다. 코드에 넣어야 할 것은 주석이라고 생각하기 쉬운데, 그것보다는 코드 자체다. 즉, 코드를 읽는 사람이 쉽게 이해할 수 있는(개발자의 의도를 파악할 수 있는) 코드를 만들라는 것이다. 쉽게 읽힌다는 것은 다양한 관점이 있을 수 있지만, 먼저 아래와 같이 몇 가지로 구분해 볼 수 있다.

**01. 추상화 수준에서 같음.**

**02. 사용하는 이름에 대한 적절함.**

**03. 추적할 필요가 없는 명백함.**

“추상화 수준에서 같음”은 계층구조에서 동일 수준의 설명을 제공해 준다는 뜻이다. 즉, 하나의 계층에서 어떤 부분은 추상적인데, 어떤 부분에 대해서는 지나치게 세밀한 경향이 있다면, 코드는 이해하기가 어렵다. “사용하는 이름에 대한 적절함”이란 이름을 읽는 즉시 하는 역할을 알 수 있게 만드는 것이다. 대부분의 상숫값은 전부 의미가 희미하다. 될 수 있으면 이름을 가지는 상수를 사용하는 것이 적절하다. “추적할 필요가 없는 명백함”도 코드를 이해하는 데 많은 도움을 준다. 즉, 내부의 상세한 구현에 대해서 알 필요없이 이해가 되는 코드를 만든다면, 문제는 더 줄어들 것이다. 하위의 계층에 있는 소스코드를 볼 필요가 없을 정도로 짜진 코드라면, 그냥 읽는 것만으로 그 자리에서 이해가 될 것이다.

코멘트가 있는 코드를 만들기는 쉽지만, 코멘트가 없으면서도 이해되는 코드를 만드는 것은 어렵다. 글쓰기는 이때도 마찬가지로 필요하다. 등장 인물에 적절한 이름을 붙여주고 그 역할에 어울리는 별명을 붙인다면, 전체를 읽어보지 않더라도 대략적인 내용은 파악할 수 있기 때문이다. 즉, 등장 인물의 이름이나 하는 일, 관계를 통해서 앞으로 어떻게 사건이 진행될 것인지 어렴풋이 생각해 볼 수 있는 것이다. 물론, 너무 뻔한 이야기는 인기를 끌지 못한다. 하지만, 소프트웨어 개발은 너무나 뻔한 이야기가 통하는 세상이다. 뻔하지 않은 놀라운 진실은 소프트웨어 개발에 어울리지 않는다. 오히려 놀아운 이야기를 내포하고 있다면, 그것으로 인해서 버그나 성능 부족을 낳을 가능성이 더 높기 때문이다.

창작의 산물인 글과 소프트웨어 개발자의 코딩 결과물인 소스 코드는 많은 유사성이 있으며, 결과적으로 그것을 보는 사람에게 감동(혹은, 이해)을 선사해 줄 수 있도록 만들어야 한다. 목적은 차이가 있지만, 글쓰기라는 점에서는 다르지 않다. 글을 잘 쓰는 사람은 생각을 잘 정리할 수 있으며, 문제를 해석하는 것도 결과적으로 생각을 정리하고 절차적으로 단계를 구성하는 것이라고 볼 수 있다. 등장 인물을 정의하고, 그들간의 관계를 설정하고, 보여주어야 할 것만 갈등(관계)로 드러나게 만든다면, 소설이 아니라더라도 충분히 흥미롭게 읽을 수 있는 코드를 만들 수 있을것이다. 소프트웨어 개발자라면 자신이 만든 결과물에서 이런 것들을 쉽게 찾을 수 있게 만드는 것이 진정한 장인이 되는 길이 아닐까?

# [ 애자일(Agile)하게 일하기 ]

요즘(아니, 요즘이 아니라 훨씬 오래전부터) 애자일이라는 것이 주목받고 있다. 애자일은 민첩하게 문제에 대처한다는 의미로 사용되는 용어다. 즉, “민첩하게 고객의 요구사항에 대응하고”, “그때 그때 주어지는 문제를 풀어나가는 방안”을 이야기한다. 애자일이 그럼 가장 최근에 나온 소프트웨어 개발방법일까? 꼭 그렇지만은 않다. 한국에서 소프트웨어 개발자로 살아가는 사람들은 이미 많은 부분이 애자일이라는 말로 표현될 수 있다는 것에 공감할 것이다.

한국 사회에서는 개발자건 비 개발자건 상관없이 사회적인 관계를 중요시하고, 그것을 해치는 것을 아주 두렵게 생각한다. 즉, “인간 관계를 공식적인 계약보다 우선”하는 것이 사실이다. 물론, 그렇다고 계약이 중요하지 않은 것은 아니지만, 개인의 능력보다는 “협력을 권장”하고 있다. “변화를 거부하기 보다는 어떻게 해서라도 변화를 받아들이려는 노력을 하는 것”이 개발자의 기본 행동이며, “동작하는 프로그램을 만들기 위해서라면 문서는 나중에 작성하는 것을 선호한다.” 해석의 차이는 있을 수 있지만, 이러한 것들이 애자일을 이루는 핵심적인 “선언”이라는 점은 충분히 알 수 있을 것이다.

하지만, 몇몇 개발자는 아직도 자신이 빠진 늪에서 헤어나오지 못하고, 발생한 문제를 잡는다고 항상 분주한 것이 사실이다. 문제의 원인을 파악하는 것은 나중에 할 일이고, 일단은 불을 끄는 것이 급하기 때문이다. 고객이건 내부의 직장 상사건 상관없이 원인을 보기보다는 결과가 나타난 숫자에 관심을 가지는 것이 더 일반적이기 때문이다. 그렇다면, 우리가 행동하는 방식은 애자일과 크게 다르지 않다고 이야기 하면서도, 왜 애자일에서 필요한 조건들은 만족시키지 못하고 있는 것일까? 혹시 우리가 너무 쉽게 애자일을 가져다 자신이 하는 일에 편의를 만족시키기 위해서 사용하고 있는 것은 아닐까?

그렇다면, 애자일 하기 위한 조건은 무엇일까? 생각보다 애자일을 시행하기 위해서는 여러 가지 전제 조건들을 만족시켜야 한다. 먼저 우리의 의식이 충분히 성숙했다는 것을 파악하고 나서 해도 늦지 않다. 애자일하게 일하는 사람들의 의식 수준이 낮다면, 그것이 의미하는 바를 제대로 하고 있지 못할 가능성이 높기 때문이다. 그렇다면, 이러한 의식(혹은, 생각)에는 어떤 것들이 있을까? 가장 중요한 것은 문제를 문제로 받아들이고, 더 나은 것을 추구하는 "합리적인 생각"이다. 합리적이지 않다는 것은 뭔가 누군가의 비 합리적인 생각이 강요하고 있는 상황일 것이며, 잘못된 강요로 인해서 애자일의 근본을 잘못 해석하고 있을지도 모르기 떄문이다.

예를 들어, 해결해야 할 문제나 요구들은 항상 우선순위를 가진다. 그 우선순위를 정하는 사람은 누구인가? 개발자? 혹은 관리자? 해답은 바로 고객이다. 우리가 만든 소프트웨어를 사용할 고객이 중요하다고 보는 것이 가장 높은 우선순위를 가진다. 따라서, 우리의 생각도 개발자에서 고객으로, 관리자에서 고객에 집중해야 한다. 하지만, 고객보다는 팀장이 우선이고, 팀장 보다는 단축 일정을 요구하는 경영층이 우선이다. 그들의 입맛에 좋다고 생각되는 기능이 최고 우선순위를 가지게 된다. 따라서, 개발자는 고객보다는 항상 내부의 힘에 의해서 개발 요구를 받아들일 수밖에 없는 것이다.

고객의 요구사항은 어떻게 주어지는가? 마케팅이나 상품기획 부서가 있다면 일단은 그들의 의견을 들어보는 것이 좋을 것이다. UI 디자인을 전담하는 부서가 있다면, 그 부서에서 생각하는 사용자에게 친근한 접근법도 익힐 필요가 있다. 개발자가 중요해서, 혹은 관리자가 더 좋다고 판단해서 정하는 우선순위는 말 그대로 시장을 반영하지 않은 “개발 내부의 우선 순위”일뿐이다. 하지만, 우리의 제품은 결국 고객의 지갑에서 돈이 나올 때 그 가치를 인정받게 된다. 합리적인 생각의 출발점은 항상 고객의 관점이어야 한다는 것이다. UI도 그렇고, 주요 기능 및 성능 요구사항도 항상 그렇게 결정되어야 한다. 개발하는 과정에 “고객의 참여”는 당연히 필요하다. 요구가 없는 개발은 마치 모래성을 열심히 만들어 놓고, 언제 있을지도 모를 파도가 오지 않기를 바라고 있는 것이나 다를 게 없다.

하지만, 고객을 모르고 개발해야 하는 경우도 있는 것이 사실이며, 심지어 개발하려는 소프트웨어의 기능도 정의하지 못하고 개발을 시작할 때도 있다. 무엇을 개발할지 모르기에 큰 그림을 그려서 일하는 것이 힘들고, 결국은 소프트웨어 개발자가 막장으로 끌려들어가 강요되는 야근과 주말 특근을 밥 먹듯이 해야 한다. 우리 인생의 대부분은 그런 곳에서 디버깅을 하면서 “인생의 가장 아름다운 시간”인 듯 착각하면서 지내고 만다(특히, 나이든 개발자는 "우리도 그런 때가 있었지."라는 말로 과거를 회상한다.). 하지만, 정말 중요한 것은 "적을 알고, 나를 알면 100전 100승"이라는 오래된 이야기 속에 있음을 깨닫아야 할 것이다. 오래된 이야기지만 그만큼 정확한 이야기도 드물다.

합리적인 생각은 개발의 기본 태도를 바꾼다. 먼저, “개발주기를 빨리 가져가는 것”이 피드백을 빨리 받을 수 있기 때문에 중요하다. 개발주기를 빨리 가져간다는 것은 무엇일까? "코딩->Unit Test->Code Review->Commit->Build->정적분석->각종 분석 Report 생성->통합 Build->통합 Test->배포"까지의 과정을 순식간에 진행할 수 있어야 한다. 당연히 자동화된 인프라(Infra)를 사용해서 실시간으로 개발자가 자신의 코드에 대해서 피드백을 얻을 수 있어야 한다. 빠르게 피드백을 얻지 못하면 늦어진 시간만큼 버그가 쌓일 가능성이 높다. 따라서, 잦은 피드백은 오히려 개발자의 부담이 아니라, 코드에 대한 방어적인 차원의 대응책인 것이다.

그렇다면 인프라 구성의 기본 원칙은 무엇인가? 여기서 필요한 개발자의 의식 전환은 “열린 생각(Open Mind)“이다. 즉, 자신의 문제를 감추려고 하지 않고, 문제를 있는 그대로 받아들여서 합리적인 해결책을 찾는 것이다. 대부분의 소프트웨어 개발자는 자신이 만든 코드에 문제가 있다는 것을 의식적으로 "회피"한다. 회피가 조금 심한 표현이라면 “방어”라는 말로 바꾸어도 좋다. 자신에게 있을지도 모르는 비난에 대해서 핑계를 찾게 되고, 심하면 아예 코드를 저장소(Repository)에 등록하지 않고 자신만이 보는 파일 서버의 디렉토리에 넣어둔다. 혹시, 남에게 전달해야 하는 결과물이 있으면, 바이너리(Binary) 파일이나 라이브러리(Library)로 만들어서 주는 경우도 있다. 하지만, 남들이 보는 것을 꺼리는 코드는 절대 좋은 코드일 수 없으며, 자신만이 아는 지식은 전파되지 않는다.

열린 생각의 진정한 시작은 “상대방에 대한 배려”에서 생겨난다. 누가 보더라도 명확하다면 그 코드는 비슷한 문제를 해결하고자 하는 사람들에게 가을날의 산들바람과 같을 것이다. 머리를 맑게 하고 더 멋진 것을 향해서 생각을 펼칠 수 있게 해 주기 때문이다. 애자일도 크게 다르지 않다. 애자일로 일하는 것은 합리적인 생각에서 나온 절차나 방법과 항상 남을 위한 열린 생각을 가지고 소프트웨어를 개발하는 것일 뿐이다. 따라서, 애자일 이건 아니건 상관없이 항상 소프트웨어 개발자는 합리적인 방법으로 일하고, 남을 배려하는 마음이 곧 자신의 버그를 해결하는 지름길임을 기억해야 하는 것이다. 협력을 중요시한다면, 더 많은 대화를 나누어야 할 것이며, 대화의 한 방법인 코드와 문서도 없어서는 안된다는 것을 알게 될 것이다. 동작하는 코드가 우선이라면, 문제가 없는 코드를 만드는 것도 개발자의 책임이 된다. 고객을 배려하기 위해서는 반드시 열린 마음으로 자신을 내보이지 않을 이유가 없는 것이다.

# [ 소프트웨어 프로젝트 성공의 핵심은? ]

프로젝트의 성공과 실패를 좌우하는 핵심은 어떤 무엇일까? 모든 프로젝트는 시간, 사람, 기능, 품질, 돈 등이 한정되어 있다. 이런 것들을 활용해서 고객을 만족시키는 것이 성공의 잣대로 활용된다. 제품의 성공과 프로젝트의 성공은 다르다. 즉, 프로젝트의 성공은 앞에서 나열한 것들을 다 맞추면서 고객이 만족하는 것을 말하고, 제품의 성공은 시장에서 많이 팔려서 이익을 많이 남기는 것을 말한다. 따라서, 프로젝트의 성공이 반드시 제품의 성공으로 이어지는 것은 아니기 때문에, 일단 우리가 여기서 다룰 문제는 프로젝트(과제)의 성공으로 한정하도록 하겠다.

프로젝트를 진행하면서 가장 먼저 하는 일은 “일정을 산정하는 것”이다. 물론, 모든 정보가 부족한 상황에서 일정을 산정해야 한다. 기본적으로 주어진 것은 어떤 제품이라는 모호한 설명과 이를 구현할 현재 인력 구성으로 프로젝트의 기간을 산정(추정: Estimate)한다. 하지만, 사실 이것은 움직이는 물체를(얼마의 속도인지는 프로젝트의 특성마다 다르다.) 내가 가진 도구를 이용해서 맞추는 것과 같다.

경험이 많으면 좀 더 확률적으로 높은 적중률을 보일 수 있지만, 기본적으로 실패할 가능성이 항상 존재한다. 따라서, 오차를 먼저 줄여야 프로젝트의 성공을 보장할 가능성을 높일 수 있다. 당연한 방법이지만 일정 산정을 최대한 뒤로 미루는 것이 효과적이다. 즉, 고객(혹은 해야 할 일을 알려주는 부서)과 협의를 통해서 해야 할 일을 구체적으로 정의하는 활동을 한 후에 일정을 산정해야 한다. 그렇지 않다면, 그냥 말 그대로 허공에 외치는 "공약(빈 약속?)"밖에는 될 수 없을 것이다.

사람이 소프트웨어 프로젝트의 핵심이다. 즉, 소프트웨어라는 것은 "고도의 지적인 능력을 집중력 있게 발휘해야 하는 정신 활동의 산출물"이기 때문이다. 단순 반복적인 작업은 기계로 해결할 수 있지만, 사람의 정신 활동은 기계를 이용해서 하기에는 현재의 기술 수준으로는 거의 불가능하다. 따라서, 사람은 프로젝트 성공에서 핵심 요소이며, 사람에 대한 적절한 관리가 없다면 프로젝트 자체가 무의미하다.

사람에 대한 관리는 "언제 출근해서 언제 퇴근하는가에 대한 것"이라고 생각하는 사람들이 의외로 많다. 고도의 지적인 활동을 하는 사람에게 단순히 출 퇴근 만을 가지고 관리를 한다는 것이 과연 얼마나 타당할까? 무엇을 하고 있는지 보는 것이 아니라, 얼마나 자리에 오래 동안 앉아 있는지를 본다면, 차라리 사람 대신에 다른 것을 의자에 앉히는 것이 비용을 줄여줄 것이다. 의자가 일하는 것이 아니라, 사람이 일한다는 사실을 종종 잊는 사람들은 항상 자신의 눈 앞에 몇 명의 사람이 있는지만 중요하게 생각하는 경향이 있다.

꾸중을 많이 듣는 아이들은 사회활동에서 주눅이 든다. 다른 애들과 함께 노는 곳에서도 잘 놀지 못하고 혼자서 외톨이가 될 수 된다. 약이 되는 호통은 앞으로 나아갈 방향을 제시하는 것이지, 사람의 존재가치를 부정하는 것이 아니다. 자존심에 상처를 주면 순간적으로는 나아질지는 모르지만, 조금만 지나면 그냥 한 귀로 듣고 한 귀로 흘린다. 따라서, 개발자는 존중과 기다림의 대상이어야 한다. 정보를 서로 공유해야 하며, 함께 가야 할 동반자로 믿어야 한다. (물론 그래도 안되는 사람이 있다면 그것은 그 사람의 문제이며, 팀의 문제가 될 수 있으니 냉정한 판단을 해야 할 것이다.)

기다리지 못하면 급한 마음에 다그치게 된다. 존중하지 못하면 신뢰할 수 없는 분위기를 만들 뿐이다. 신뢰받지 못한다고 생각한다면, 신뢰를 주지 않는 것이 인간이다. 따라서, 믿음을 주지 못하는 관계는 지속될 가능성이 없는 것이다. 과제는 지속성을 가져야 하지만, 사람이 지속적인 관계를 맺지 못하게 되면, 과제 존립 자체가 위협받을 수밖에 없다. 함께하기 위해서는 명확한 “비전(Vision)”을 제시할 수 있어야 하며, 나아가는데 걸림돌이 되는 것을 제거할 수 있는 권한도 가져야 할 것이다.

모든 상품기획 담당자는 대부분 비슷한 성향을 가진다. 즉, 경쟁사보다 나은 제품을 개발하기를 요구한다. 하지만, 사실 그렇게 해서 팔 수 있는 대수가 몇 대인지는 말하지 못한다. 왜냐면, 기본적으로 미래는 아무도 예측할 수 없기 때문이다. 그리고, 한 가지 덧붙이자면, 미래에 나올 제품을 그 시점에서 가져야 할 경쟁력이라고 믿는 기능에 대해서 요구한다. 즉, 그 때쯤 되면, 이런 기능들이 일반화될 것이고, 따라서 개발자는 이런 기능들을 제공해야 한다고 이야기한다. 하지만, 한가지 잊은 것이 있다. 그 때 팔려는 제품을 지금부터 만들어야 하고, 지금부터 만들기 위해서는 현재 나온 기술이 뒷받침 되어야지 가능하다는 것이다.

미래에 나올 기술을 기준으로 개발하는 것은 현실성이 없다. 따라서, 우리가 이용할 수 있는 기술은 현재 안정된 기술뿐이며, 여기서 필요한 것은 트레이드 오프(Trade-off)와 정말 고객이 원하는 기능이 무엇인가를 찾는 것이다. 그렇게 찾은 것들을 현재의 기술을 가지고 해결할 수 있는가 없는가를 같이 고민하는 것이다. 시장의 모든 제품이 가지는 장점들을 만족시키기 위한 개발은 그 모든 제품을 만드는 노력보다 더 큰 비용이 들어가는 일이라는 것을 알아야 한다.

품질은 어떤 변명으로도 회피할 수 없다. 품질은 반드시 최고 수준을 만족할 수 있도록 최대한의 노력을 해야 한다. 앞에서 이야기한 기능은 몇 가지 제한되더라도, 제공되는 기능에 대해서는 완벽한 수준의 품질로 만들어야 하는 것이 목표다. 버그를 완벽히 제거하는 것이 최고 품질이라고 할 수 있으나, 사실 이것은 소프트웨어에 대해서는 불가능한 목표다. 신이 아닌 다음에야 사람이 만든 것이 완벽할 수는 없기 때문이다. 그리고, 개발 도중에 완벽하게 모든 버그를 제거하는 것도 불가능하기 때문이다.

"테스트의 목적은 버그를 찾아내기 위해서 소프트웨어를 실행하는 것"이지 소프트웨어에 버그가 없다는 것을 증명하는 아니다. 아무리 많은 테스트를 한다고 하더라도, 결국 버그는 항상 발생할 수밖에 없다. 정해진 일정과 비용을 가지고 한정된 테스트만 가능한 것이 현실이기 때문이다. 따라서, 품질은 정해진 스펙(Spec.)을 만족하게 하는 수준에서 결정이 되며, 그러한 스펙(Spec.)을 지속해서 높여나가는 것이 실용적이다.

과제 비용은 과제의 일정과 민감하게 연결되어 있다. 대부분은 소프트웨어 프로젝트는 사람의 인건비가 비용의 상당 부분을 차지한다. “더 적은 사람으로 더 많은 기능을 더 짧은 시간에” 만들어서 고객의 만족을 이끌어내면, 단축한 만큼의 비용이 줄어들기 때문이다. 프로젝트의 일정이 지연되면, 추가적인 사람에 대한 “MM(Man Month)”가 투입되고, 결국에는 비용 상승을 유발해서 그만큼의 이익이 줄어들게 만든다.

하지만, 줄일 수 있는 일정은 사실상 예측하기가 어렵고, 그 기간에 만들어질 기능에 대해서도 고객이 만족하기 어려울 수 있다. 따라서, 과제의 비용은 “일정/개발자 수/기능 수” 등에 복합적으로 연결되어 있기에, 그것들과 함께 관리되어야 한다. 단순히 몇 명의 개발자를 추가하면 과제가 일찍 끝나는 것이 결코 아니다. 여기서 중요한 것은 비용을 관리해서 그 과제의 위험도를 지속해서 파악하는 것이 중요하다. 앞에서 이야기한 품질 역시 일정이나 개발자 수, 개발해야 하는 기능에 의존성을 가진다.

일정 단축을 위해서 뭔가를 희생해야 한다면, 포기할 수 있는 것은 당연히 “기능의 수”가 될 가능성이 높다. 대부분 일정이 중요한 과제는 기능의 수를 줄이는 방법으로 대응해야 한다. 사람을 추가적으로 투입하는 것은 과제 초반에만 의미를 가진다. 테스트를 늘리는 것은 과제 후반에 더 많은 버그를 제거해야 할 가능성이 높다.

개발자의 수를 줄이는 것은 당연히 해결책이 아닐 것이다. 따라서, 일정 단축을 위해서는 중요도가 낮은 기능을 포기하는 것이 가장 효과적이다. 특히, 품질 수준을 유지하기 위해서도 기능의 수를 낮추는 것이 좋다. 많은 기능은 많은 시스템을 구성하는 요소간의 상호 작용을 의미하며, 결국 시스템을 더 복잡하게 만들어 버그 발생 가능성을 높이기 때문이다.

여러 가지로 소프트웨어 프로젝트의 성공에 지대한 영향을 주는 요소들을 살펴보았지만, 가장 중요한 것은 역시 사람이다. 과제의 리더라면 소프트웨어 개발자를 잘 관리할 수 있는 알찬 계획과 방법을 알아야 하며, 핵심은 "정보의 공유와 기다림의 미학"이다. 급한 마음으로 사람들을 재촉한다면, 진정한 의미있는 변화는 만들지 못한다. “말을 물가로 끌고는 갈 수 있지만, 목마르지 않은 말에게 물을 먹이지는 못하기 때문이다.”(만약, 이것이 가능하다면, 목마르지 않은 말에게 물을 먹이는 "물고문"밖에 달리 표현할 말은 없다.) 정보는 일을 하는데 필요한 “과제의 투명성”을 확보하는 것이며, 기다림은 “제대로 일하는 것을 보장하는 방법”이다. 따라서, 관리자에게 주어진 권한은 이런 “과제 개발의 투명성과 제대로 개발자들이 일할 수 있는 환경”을 만들어주는데 집중해야 한다.

# [ 천망회회 소이불실( 天網恢恢 疎而不失 ) ]

천망회회 소이불실(天網恢恢 疎而不失)이라는 말은 "하늘의 그물은 촘촘하지는 않지만, 거르지 못하는 것이 없다."라는 말이다. 즉, 세상의 악은 하늘의 그물이 아무리 넓어도 걸러준다는 뜻이다. 소프트웨어 개발도 딱 이와 같으면 어떨까? 사실 이 말을 처음 본 것은 “황성”이 그린 무협 만화다. 이 말이 마음에 와닿는 이유는 적절한 방법으로 창조적인 생각은 흘려보내지만, 적절히 오류를 걸러내는 방법이 없겠느냐는 생각이 들었기 때문이다. 버그를 차단하는 적절한 방법론을 가지고 있다면, 우리가 하는 소프트웨어 개발이 어렵지만은 않을 것이기 때문이다.

버그가 발견되는 곳에서 더 많은 버그를 찾아낼 가능성이 높다. 즉, 발견된 버그 이상으로 더 많은 버그가 있을 가능성이 있다는 말이다. 버그가 많이 발생하는 코드의 특징은 뭘까? 단연 복잡함(Complexity)에 기인한다고 볼 수 있다. “사람의 이해를 방해하는 요소”들이 복합적으로 버그를 발생시키는 요인이 된다는 것이다. 사람의 이해를 방해하는 가장 큰 원인은 뭘까? 첫 번째 이유는 “너무 길어서 이해하기가 어렵다”는 것이다. 코드의 길이가 길어지면, 사람은 집중력이 떨어진다. 한 번에 코드를 보지 못하고 상하로 이동하면서 코드를 보는 것은 쉬운 일이 아니다. 대부분의 사람은 천재가 아니기에 많은 코드를 한 번에 기억하는 것은 어려운 일임이 분명하다.

두 번째 이유는 “여러 번 읽어야지 이해할 수 있다”는 점이다. 한 번에 코드를 보고 이해를 할 수 있다면 아주 잘 작성된 코드임이 분명하다. 잘 작성된 코드를 만나는 것은 실무 현장에서 축복과 다름없다. 대부분 코드는 최소한 세 번 이상은 봐야지 이해가 된다. 자주 봐야지 이해할 수 있는 코드는 잘못 사용할 가능성이 높으며, 수정하기도 쉽지 않다. 당연히 더 많은 집중력이 필요하며, 작은 실수를 유발할 가능성도 높다. 따라서, 누군가 코드를 이해하지 못하는 사람이 고치게 되면, 버그로 이어지게 될 것이다. 이런 코드들의 특징은 대부분 이름을 재대로 사용하지 않았거나, 혹은 다른 코드에 지나치게 의존하는 경향이 있다. 연결 고리가 많을수록 기억해야 할 것들이 늘어나기 때문이다.

세 번째 이유는 “중복”이다. 중복된 코드는 여러 곳에서 나타나며, 완전히 글자 하나까지 복사된 코드에서 논리(Logic)가 복사된 코드까지 다양한 형태가 있다. 완전히 복사된 코드는 그나마 변경하기 쉽지만, 논리적인 복사는 생각보다 처리가 힘들다. 중복된 코드가 발생하는 원인은 논리적인 유사성으로 쉽게 문제를 해결하려는 개발자의 편의 때문이다. 즉, 비슷하게 보이기 때문에 복사해서 갖다붙인 것이다. 문제는 그것으로 완전히 문제를 해결한 것이 아니라는 점이다. 나중에 다시 복제된 한 곳을 수정하게 되면, 나머지 복제된 모든 코드를 찾아서 똑 같이 수정해 주어야 하기 때문이다. 당연히 모든 복제된 코드를 머리 속에 유지하기는 불가능하다.

네번째 이유는 “추상화의 오류”다. 같은 수준에서 다루어져야 할 것이 아닌, 구체적인 것과 추상적인 것이 섞여서 나타날 때 갑작스러운 논리의 비약이 발생하고 혼돈이 생기는 것이다. 같은 수준의 추상화를 가진다는 말은 같은 수준의 단어를 사용한다는 것이다. 예를 들어, “데이터를 복사”하라는 것과 “데이터의 특정 비트를 변경해서 다시 기록하라”는 것은 추상화 수준이 적절하지 않다. 오히려, “데이터의 비트를 조작”하라와 “데이터를 기록”하라와 같이 구분해서 사용하는 것이 추상화 수준에는 더 어울린다. 추상화 수준의 오류는 코드를 읽는 사람이 개별적인 상세한 것을 기억하기 어렵다는 것을 의미한다. 따라서, 추상화 수준을 섞어서 만들게 되면, 기억할 수 있는 범위도 줄어들 수밖에 없다.

따라서, 이런 것들을 잡아줄 수 있는 그물을 만들어 놓는다면, 우리는 개발에 있어서 가장 큰 조력자를 옆에 두는 것과 같은 효과를 누릴 수 있게 된다. 먼저 너무 긴 코드와 복잡한 코드는 상호 비례관계에 있어서 찾기가 쉽다. 복잡한 코드는 길고, 긴 코드는 복잡한 경우가 많다. 그렇다고, 항상 짧은 코드를 유지할 수 있도록 노력하는 것이 프로그램의 고급 테크닉을 요구하는 것도 아니다. 짧은 코드란 논리적으로 잘 나누어진 코드를 말하는 것이지, 긴 코드를 무조건 짧게 짜라는 것이 아니다. 본질적인 복잡함을 없애는 것은 큰 문제를 작은 문제로 나누는 것이다. 즉, “나누어 정복하기(Divide & Conquer)"라는 전략이 핵심이다.

이해하기 쉬운 코드란 읽기 쉬운 코드다. 읽기가 어렵다면, 일단 그 코드는 오류를 발생시킬 가능성이 높다. 이것은 도구(Tool)로서 걸러지는 부분이 아니라, 상대방의 코드를 읽고 이해하기 쉬운지 어려운지를 사람이 판단해야 하는 부분이다. 따라서, 비용이 많이 든다고 볼 수 있다. 코드 리뷰는 상대방의 코드가 읽기 쉬운지 어려운지를 판단할 수 있는 가장 좋은 방법이다. 논리적인 오류를 찾는 것은 코드에 대한 이해가 선행되어야 하며, 읽기가 쉽다면 오류를 찾는 것도 쉬워질 것이다.

논리적인 중복이 아닌 물리적인 중복은 좋은 툴들이 많으므로, 이를 이용하면 쉽게 찾아질 것이다. 매번 빌드(Build)에서 분석되는 내용을 기초로 피드백을 주고, 수치로 관리하다 보면, 코드의 오류가 줄어드는 것을 확인할 수 있을 것이다. 논리적인 중복은 찾기가 어려운데, 이는 비슷한 일이 각기 다른 상황에서 처리하기 때문이다. 이는 팀워크로 일하고, 상대방의 코드에 대한 이해가 높아질 때만 제거가 가능하다. 따라서, 적극적인 협업 이외에는 달리 자동화할 수 있는 부분이 크지 않다. 하지만, 이것도 특정 부분에서의 반복이 있을 수 있으니, 자동화된 도구에서 만들어내는 결과를 근거로 먼저 찾아보는 것이 한 가지 해결 방법일 것이다.

추상화의 오류는 해당 코드가 하는 일을 분석해야 알 수 있다. 서로 깊이가 다른 부분이 있는지를 코드를 읽어가면서 파악해야 한다. 즉, 계층적으로 같은 위치에 포함될 필요가 없는 부분이 있는지를 찾아보는 것이 한 가지 방법이다. 프로그램의 호출 그래프(Call Graph)를 그려주는 도구와 같은 것을 이용해서 어떤 계층 들이 있는지를 확인하고, 동일 계층에서 현저하게 추상화가 다른 부분을 찾아 나가는 것이다. 또한, 같은 블록(Block)에 놓여있는 코드들이 동일한 추상화를 가지는지도 검토해야 한다. 같은 블록을 확장해서 함수가 될 수도 있으며, 하나의 모듈이 추상화 단위가 될 수도 있다. 추상화란 구체적인 구현을 모르더라도 어떤 역할을 하는지 파악할 수 있는 능력을 의미하기에, 동일한 수준의 추상화란 결국 “동일한 수준의 일 처리”를 찾아내는 것일 뿐이다.

이 정도만 해도 사실 버그를 유발할 가능성이 높은 부분들에 대한 "하늘의 그물"은 어느 정도 마련한 것이다. 물론 그것으로 충분하다는 것은 아니지만, 생각보다 많은 오류를 찾아서 없애고, 앞으로 발생할 오류의 가능성도 상당 부분 낮출 수 있을 것이다. 어쩌면 만족할 수 있는 완벽한 품질은 소프트웨어에서 불가능한 목표일지도 모르겠지만, 그렇기에 또한 우리가 하는 일이 다른 부분보다 더 많은 관심을 받고 있는지도 모른다. 따라서, 다소 부담스러운 관심일 지라도 무관심보다는 행복하다고 믿도록 해야 할 것이다. 아쉬운 것이 있다면, 더 좋은 “그물”을 지속적으로 만들고 관리해야 한다는 것이다. 한번 잘했다고 일이 끝나는 것은 아니다. 완벽한 그물이 없다면, 항상 개선할 수 있는 부분이 있다는 말과 같다. 따라서, 지속적으로 더 좋은 그물을 만들기 위해서 필요한 노력을 아끼지 않아야 할 것이다.

# [ 시간과의 싸움에서 이기는 방법 ]

프로젝트에서 주어지는 것은 “사람, 돈(예산), 시간(일정), 구현해야 할 기능”으로 분류할 수 있다. 여기에 추가해야할 것이 있다면, 그것들을 가지고 최고의 품질을 구현해야 한다는 사실이다. 사람은 어떤 프로젝트를 수행해도 항상 부족하다. 그리고, 제대로 된 사람을 넉넉하게 구하고 나서 프로젝트를 시작하는 경우는 정말 드물다. 돈은 조직의 생존에 관련된 문제이므로, 절대 넉넉하게 주어지지 않는다. 개발을 위해서 구입해야하는 장비도 충분히 사지 못하는 경우도 있다. 시간은 개발자가 정하는 것이 아니라 항상 누군가의 요구라는 이유로 터무니없이 짧게 주어지는 것이 일반적이다. 품질은 언제나 “완벽”함을 추구해야 하고, 100% 기능을 구현하는 것이 당연한 목표가 된다.

그렇다면 프로젝트의 시간은 무엇일까? 사실 시간의 많고 적음은 사람, 돈, 품질, 기능과 연결되어 있다. 어떤 한 가지를 높이면 다른 것들이 영향을 받게 된다. 즉, “Y= F(X)”라고 했을 때, 프로젝트의 시작 때 “Y”값 고정된다, 따라서, “F(X)”를 결정하는 것은 “X”의 인자로 사용할 수 있는 “X1, X2, X3... Xn”를 어떻게 조절할 것인가에 의존적이다. 그리고, 어떤 인자는 “+”로, 다른 인자는 “-“값으로 정해야만 정해진 시간에 과제를 마칠 수 있을 것이다. 무조건 “+”만 된다면, 일정이 길어지는 것을 당연하게 생각해야 할 것이다.

기본적으로 시간은 돈이다. 따라서, 돈이 많으면 시간은 줄일 수 있다. 사람이 많아도 시간을 줄일 수 있지만, 투입하는 시점이 문제가 되곤한다. 너무 늦게 사람이 투입되면 시간은 더 지연되고 만다. 너무 많은 사람이 투입되도 지연이 발생한다. 사람은 통신 그래프 상의 노드이며, 투입되는 사람에 비례해서 통신 해야하는 채널 수가 기하급수적으로 증가하기 때문이다. 기능이 많아도 시간은 길어진다. 그리고, 길어진 시간만큼 인건비도 늘어나게 된다. 당연히 또 돈이 늘어나게 되는 결과로 이어진다. 품질을 높이기 위해서는 사람이 많아야하고, 돈이 더 들어야하고, 기능은 줄여야 한다. 시간이 더 많이 주어지면 품질도 높아지는 경향이 있다. 어쨌든 품질도 돈이다. 따라서, 시간은 확실히 돈이다.

시간의 싸움에서 이기는 길은 오로지 기능의 축소 밖에 남지 않는다. 정해진 시간에 모든 기능을 다 만든다는 것은 몹시 어려운 일이다. 100개의 기능을 구현하는데 1년이 걸리는 일을 6개월 만에 끝내라고 한다면, 품질이 낮은 소프트웨어 밖에는 볼 수 없을 것이다. 차라리 50개의 기능이 안정적으로 동작하는 품질이 높은 소프트웨어를 기대하는 편이 현명하다. 다음 6개월 동안 나머지 기능 50개를 추가하는 것이 현실적인 계획이 될 것이다. 따라서, 시간의 싸움은 결국 어떤 기능을 소프트웨어가 제공할 것인가로 귀결되며, 가능한 제품의 핵심 경쟁력이 될 수 있는 기능을 우선순위를 높여서 빨리 개발해야 한다. 나머지 우선순위가 낮은 기능들은 나중에 “돈(시간, 여유)”이 생기면 구현해도 된다. 지금 당장 중요한 것이 시간이라면, 나머지는 “돈”으로 시간을 사야한다.

따라서, 시간과의 싸움에서 이기고 싶다면, 우리가 진정으로 만족시켜야 할 사용자를 지켜봐야 한다. 사용자들이 “정말 원하는 기능”을 제시간에 공급하는 게 가장 현명한 선택이 될 것이기 때문이다. 구현된 기능의 20%만 80%의 사용자가 사용한다면, 우리가 구현하는 80%의 기능은 거의 사용되지 않는다고 봐야한다. 밤새워 사용되지도 않는 기능을 구현한다고 복잡한 코드를 만들기 보다, 사용하는 기능에 집중해서 “소중한 시간(돈)을 투자”하는 것이 훨씬 경제적이다. 하지만, 어떤 기능이 정말 사용되고 어떤 기능이 사용되지 않을지를 구분하는 것도 쉬운 일이 아니다. 더군다나, 한번 구현된 기능은 코드에서 제거하는 것도 힘들다. 결론은 우리는 시간과의 싸움에서 패배할 가능성이 더 높다는 것이다. 그게 아니라면, 우리가 싸워야 할 대상을 잘못 선택하고 있는지도 모른다.

# [ 조금씩 일하기 ]

대부분의 소프트웨어 개발자는 자신이 작성한 코드가 많다는 것에 자부심을 가진다. “많은 코드를 작성하는 사람이 일을 잘한다”는 기준은 만족시킬 수 있겠지만, 과연 그렇게 일하는 사람이 효과적으로 일하고 있는 것일까? 얼마 전에 전체 코드를 담당하고 있는 사람 별로 몇 %씩 맡고 있는지(코딩 했는지) 조사한 적이 있었다. 15명의 개발인력이 각각 얼마나 많은 코드를 담당하고 있는지 확인 했었다. 모듈을 기준으로 담당자를 확인한 결과(체크인 기록도 같이), 과제에 심각한 문제가 있다는 것을 쉽게 발견할 수 있었다.

한 사람이 30% 정도의 소스코드를 관리하고 있다는 말을 듣는 순간, 그 사람이 코딩을 잘했는지 그렇지 않은지를 판단하기에 앞서 팀에 문제가 있다는 생각부터 들었다. 왜? 한 사람에 의해서 좌우되는 코드의 양이 너무 컸기 때문이다. 그렇다면 다른 사람들은 놀고 있다는 뜻인가? 아마도 그건 아닐 것이다. 뭔가 하고 있을 가능성이 크다. 서브 모듈이나 모듈의 일부 함수 혹은 작은 부분이라도 코딩을 하고 있을 것이다. 하지만, 그것을 감안하더라도 누군가에게 “힘”이 집중되고 있는 것은 옳지 않다. 그만큼 내부에서 자신의 힘을 이용하고 있을 가능성이 있기 때문이다.

우리에게 정말 필요한 것은 슈퍼(Super)급 개발자가 아니라, 고른 실력을 갖춘 팀원들이다. 물론, 중소기업처럼 적은 수의 개발자가 많은 일을 하는 경우는 예외지만(이것도 나중에 문제가 될 수 있다), 대부분의 과제는 정의된 모듈을 각각의 개발 인력이 고르게 나누어 개발하는 것이 일반적이기 때문이다. 따라서, 한 사람의 슈퍼 개발자 보다는 실력이 고른 개발자들이 더 많이 필요하게 된다. “좋은 팀”이라고 한다면, 팀원들의 실력을 상향 평준화하는데 많은 시간을 투자하며, 꾸준히 역햘 개발을 위해서 노력하기 때문이다.

작은 단위로 코딩하는 것은 작은 단위로 테스트 및 통합, 빌드(Build) 등등이 꾸준히 일어나고 있다는 것을 가정한다. 만약, 그렇게 하고 있지 않다면, 테스트 및 통합과 빌드 등에 있어서 항상 어려움이 많은 과제를 수행하고 있을 것이다. 한 번에 모든 것을 다 하고 나서 통합하겠다고 마음먹는다면, 다른 사람이 하는 일을 무시하는 마음이 자리잡고 있을지도 모른다. 꾸준히 조금씩 변경을 가해서, 자신이 한 일과 다른 사람이 한 일을 동기화(Synchronization)시키는 것이 과제의 진행에서는 핵심이다. 작은 과제를 진행할 때도 이처럼 반복적으로 일이 진행되어야 한다. 모든 어려운(복잡한) 일은 한 번에 해결되는 것은 없다고 봐도 과언이 아니다.

그렇다면, 작은 단위의 코딩은 어떻게 하는 것일까? 프로그램에서 가장 작은 단위로 일을 쪼개면 어디까지 내려갈까? 변수의 추가나 이름의 변경이 최소 단위일까? 아니면 주석의 수정 정도도 들어갈까? 물론, 이런 것들도 작은 단위가 될 수 있을 것이다. 하지만, 실용적인 측면에서 본다면, 프로그램에서 작은 단위의 변경은 대체로 하나의 함수(객체 지향이라면 클래스)에 관련된 변경일 것이다. 새로운 함수의 추가나 새로운 기능을 위한 수정 등이 작은 변화의 시작이라고 볼 수 있다. 따라서, 이러한 변경을 했다면, 즉시 테스트와 주석 등이 실행 및 갱신되어야 하고, 또한 버전 관리 시스템에 체크인되어, 다른 코드와 아무런 충돌을 발생시키지 않는다는 것을 보장해야 한다.

함수의 변경은 반드시 테스트되어야 한다. 테스트를 어떤 식으로 할 것인가가 항상 문제지만, 개발자의 손을 떠나기 전에는 어쨌든 개발자가 직접 변경되거나 추가된 코드에 대해서 테스트를 한다는 것이 원칙이다. 될 수 있으면 자동화되어야 하겠지만, 자동화가 불가능한 상황도 있을 수 있다. 하지만, 테스트를 자동화를 하지 않고 그냥 넘어 간다면, 결국 테스트에 들어가는 비용이 늘어나게 된다. 사실 자동화는 대부분의 경우 사람을 투입하는 것보다 훨씬 비용이 싸다. 새로운 서버를 구매하는 것이 한 사람을 고용하는 것보다 훨씬 싸다는 것은 누구나 다 알 것이다. 물론, 개발자가 자동화된 테스트를 위해서 작성 및 관리해야 해야 할 코드는 늘어나지만, 반복적인 테스트로 인해서 비용을 충분히 낮출 수 있다(재활용이 늘어나기에).

작은 단위로 일하는 것에 싫증이 난 개발자도 있을 것이다. 남들보다 코딩을 훨씬 잘하는 사람도 있고, 어떤 경우에는 일반적인 개발자보다 5에서 10배에 달라는 차이를 보이는 경우도 있을 것이다. 하지만, 중요한 것은 혼자서 일하는 것이 아니라는 점이다. 공동 개발에서는 아무리 작은 변경이라도 치명적인 오류를 발생시킬 가능성이 있으며, 항상 남과 동기화를 염두에 두고 진행해야 한다. 작게 조금씩 일하면 문제는 항상 마지막 작은 증분(Increment)에 존재한다(물론, 어려운 버그는 찾기 힘들겠지만). 따라서, 버그를 찾아내는 것도 쉽다. 한 번에 많은 코드를 통합하면 많은 버그가 발생하지만, 한번에 하나의 함수만 통합하면 버그는 그 함수에 있을 가능성이 높다. 따라서, 작게 일하는 것은 멍청한 것이 아니라, 경제적인 코딩 방법인 것이 분명하다.

# [ Good to Great ]

최근에 본 책의 제목이 "Good to Great"이다. "좋은 것에서 위대한 것으로"라는 뜻으로, 어떤 기업이 위대한 기업이 될 수 있는가를 분석해 놓았다. 책의 앞 부분만 읽어도 대략 무슨 이야기를 하는지 충분히 이해할 수 있는 내용이라, 마치 기업의 보고서를 써놓은 듯한 인상이 들기도 한다. 흔히 조직이 나아갈 방향에 대해서 이야기할 때 이 책을 많이 인용된다. 결국 "사람의 문제"라는 단순한 진리를 많은 사람이 공을 들여서 써 놓았다(개인적인 의견일 뿐이다).

성과가 있는 기업이 아니라, 성과가 탁월한 기업을 대상으로 한 분석이라, 일단은 익숙하지 않은 회사들과 익숙하지 않은 이름들이 나오는 것도 사실이지만, 중요한 것은 "규율"을 잘 지키고 “지속해서 추구하는 사람"이 탁월한 성과를 만들어 낸다는 것이다. 일반화에서 나오는 오류는 따지지 말고, 이것을 한번 소프트웨어 개발에도 적용하면 어떨까? 소프트웨어 개발에서 "규율"과 이것을 잘 지키는 팀원들을 가지고 일 한다면 과연 탁월한 성과가 나올까?

소프트웨어 개발 성숙도를 따질 때, 흔히들 CMMI(Capability Maturity Model Integrated)를 많이 보는데, 여기에서 중심은 사람에 대한 관리적인 측면들이다. 개발자들이 개발 프로세스를 잘 이해하고, 이를 성실하게 지속해서 지키고 있는지를 본다. 공통된 점은 조직을 구성하는 사람에 대한 이야기라는 점이다. 소프트웨어 개발자가 어떤 수준이며, 어떤 형태(방식)로 일하고 있는지를 객관적인 기준으로 측정하고 있다. 따라서, 어떤 목표를 추구하고 있는 조직은 구성원들에 대한 "규율"과 조직의 목표를 추구하려는 "지속적인 의지"의 수준을 평가한다고도 볼 수 있다.

큰 회사 규모에서의 이러한 노력도 중요하지만, 우리가 당면한 문제는 소규모 조직에서의 적절한 “규율과 지속성”을 어떻게 사람들에게서 끌어내는가이다. 과제 관리자는 적절한 사람을 적절하게 뽑을 수 있어야 하고, 또한 과제에 맞는 규율을 설정하고, 이를 지속적으로 관리해 나가야 한다. 경험적인 측면이 전혀 무시될 수 없기에, 이미 남들이 만들어 놓은 방법론을 적절하게 변형해서 프로젝트에 최적인 해결책을 찾아내는 것도 중요하다.

하지만, 개발 방법론은 이론이기에 무조건적인 맹신보다 무공비급의 "오의"처럼 핵심을 이해한 후에 적절하게 조절해서 운용하는 것이 가장 좋다. 물론, 이것은 모든 과제 관리자들의 문제이고, 나만의 해결책이 모든 사람의 플랫폼처럼 이용될 수는 없을 것이다. "어느 날 갑자기 잘됐다고 자랑하기보다는 어떻게 해보니 이렇게 되더라. 그래서 난 이렇게 했다" 라고 이야기하는 것이 훨씬 일을 분석적으로 하는 첫 시도가 아닐까 생각하게 된다. 좋은 회사에서 위대한 회사로 가는 길이 쉽지 않듯이, 좋은 리더에서 위대한 리더로 가는 길도 결코 쉽지는 않다. 하지만, “규율”을 정하고, 그것을 꾸준히 지켜나가는 “지속성”은 결과를 만드는데 있어서 항상 중요한 역할을 해왔다.

# [ 문제에 대한 인식과 공유 ]

소프트웨어 개발이 어느 정도 진행되면 개발에서 발생했던 문제점들은 점차 줄어들게 되고, 특정 시점이 되면 배포할 수 있는 수준이 될 수 있을 거라는 막연한 기대를 하게 된다. 하지만, 이러한 기대에 대해서 찬물을 끼얹는 일이 있으니, 그것이 바로 이중장부다. 즉, 품질을 담당하는 부서와 개발을 담당하는 부서 간에 적절한 타협이 이루어져, 발생한 문제를 고위층에 보고하지 않게 되는 경우다.

이런 현상이 발생하는 이유는 소위 높으신 분들의 성과가 지표로 관리되고, 개발 일정이 강요에 의해서 주어져, 일정지연이 절대 용납하지 않는 상황에서 발생한다. 하지만, 개발되는 모든 소프트웨어는 버그가 있는 것이 사실이며, 또한 일정이란 구현해야 할 기능의 적고 많음에 따라 지연이 될 수 있다는 사실을 높으신 분들은 이해하지 못한다.

높으신 분들은 자신들의 개발자에게 완벽한 제품을 만들기를 항상 주문한다. 물론, 주어진 일정은 여러 번 회의를 하고 나면, “상상 그 이상으로” 짧아져서 개발자에개 통보되는 것이 일반적이다. 당연히 주어진 일정에 무리해서 맞추기 위해서는 지속적인 야근과 주말 특근이 필요하게 된다. 무리한 일정은 사람을 극단으로 몰아가게 되며, 올바른 구현보다는 손쉬운 구현으로 몰아가게 된다.

버그의 수는 당연히 처음 개발할 때와 비교해서 전혀 줄어들지 않을 것이고, 코드에 대한 관리적인 부담은 가중된다. 결국, 해당 과제를 완수하고 나면 그것을 정리하는데 더 큰 비용이 들어가게 된다는 것을 발견하지만, 개선에 대한 노력에는 전혀 손을 담그지 못한다. 그 때 가서 해결하면 된다는 막연한 기대를 하지만, 그 때가 되어도 절대 제대로 된 개선을 하지 못한다.

덮어놓은 문제가 발생하지 않기를 바랄 뿐, 언제 발생한다고 해서 새로운 것은 아니다. 이미 알고 있는 문제들이 시간이 지나면 하나씩 둘씩 불거져 나올 것은 뻔하다. 품질 부서를 불러대고, 하드웨어, 소트트웨어 개발자를 몰아세우지만, 결국 해결 방법이 없다는 것을 발견하고는 “또 하나의 가족”을 회사에 만들기만 원하게 된다. 정시 퇴근이란 말은 잊혀진 단어가 될 것이다.

이런 식의 과정을 몇 번 반복하고 나면 상태가 조금은 나아질까? 글쎄, 아마도 “관리하는” 소프트웨어의 크기나 기능에 큰 변화가 없다면 모를까, 이런 식의 관리 방법은 사람만 피곤하고 지치게 만들 뿐, 해결책이 아니라 상황을 악화시킨다. 해결책은 이미 개발자가 잘 알고 있다. 충분한 시간을 투여해서 소프트웨어의 구조를 개선하는 것이다.

소프트웨어 개발자들이 능동적으로 어떤 기능을 언제 적용할 것인가를 로드맵(Roadmap)으로 정할 수 있어야 한다. 무턱대고 최고 최초의 기능을 개발 스펙으로 일정까지 못막아 들이미는 상품기획이나 마케팅의 의견을 수용 하다가는, 일정지연 및 품질이 깨어지는 것은 어쩔 수 없는 결과가 될 뿐이다. 물론, 상품기획이나 마케팅이 잘못한다고 이야기하는 것이 아니다. 그들도 열심히 나름 노력하고 있다. 문제는 항상 “실적”에 매달리는 사람들이 저지르는 “눈 먼 실수” 때문이다. 장기적인 경쟁력 보다는 단기적인 성과에 치우치기 때문이다(당연히 그들도 말하지 못하는 고충이 있을 것이다).

따라서, 항상 로드맵을 가지고 움직일 것을 건의하고 싶다. 즉, 언제 어떤 기능이 추가되고, 해당 기능이 들어간 제품을 언제 배포(Release)하는지 개발자와 협의하라는 것이다. 로드맵은 직접적으로 보여줄 수 있는 것들을 위주로 정리해서 만들도록 해야한다. 내부적인 기능 검증이 필요한 항목들을 시각적으로 표현할 수 있어야 한다. 즉, 과제의 투명성을 확보하기 위해서는 실제로 해당 기능이 동작한다는 것을 보여주어야 한다. 단순히 “구현률 98%”로는 부족하다.

기능 추가에 대해서는 반드시 개발자의 의견을 물어서 일정계획을 세워야 한다. 하드웨어 변경이 있을 때에는 무리한 기능 추가보다는 기존 기능 검증에 주력해야 한다. 만약, 하드웨어 변경이 없는 파생 모델의 경우라면, 기존 하드웨어의 성능 한계 내에서 가능한 새로운 기능을 추가하도록 일정 계획을 세우도록 해야 한다. 그리고, 될 수 있으면 최소의 기능만을 추가해야 할 것이다. 한꺼번에 많은 변화를 주게되면, 품질과 일정이라는 두 마리 토끼를 함께 잡지 못하게 된다.

개발하는 중간에 들어오는 추가 요구사항에 대해서는 철저히 일정에 반영하도록 해야 한다. 일정이 도저히 나오지 않는다면, 새로운 일정에서 언제 해당 기능이 구현될지를 협의하도록 한다. 그리고, 추가된 기능과 일정에 대해서는 모든 이해 관계자가 함께 알고 있어야 한다. 단순히 누군가만 아는 것은 기능의 누락이나 일정 지연의 중요한 원인이 될 수 있기 때문이다. 이 역시도 과제를 투명하게 만들기 위한 조치일 뿐이다.

문제에 대한 인식이 되었다면, 개발자 및 품질부서가 숨겨야 하는 이중장부는 없애야 한다. 문제가 있다는 것을 인정하고, 이를 시인한 후 개선하는 것이 개발자의 일이다. 숨겨가면서 일 하는 것은 “양심있는” 개발자가 할 일이 아니다. 물론, 이렇게 이야기한다면 “정치를 못한다”나 “현실을 무시하는 소리다”라는 비난을 받을수도 있다. 하지만, “정치나 현실 인식”이란 회사가 제대로 운영될 때나 가능한 소리다. 망할지도 모르는 회사에서 양심마져도 없다면, 자신의 인생이 너무 아깝지 않을까?

욕을 먹더라도 해야 할 일은 해야 한다. 숨겨진 버그를 드러내고, 잘못된 수치를 수정해야 한다. 현실을 정확히 인식해야 올바른 계획을 세울 수 있다. 이상적인 일정 만 생각하는 높으신 분들이 왜 그렇게 빈번한 일정 지연이 발생하는지 알아야 하기 때문이다. 문제가 있다는 것을 인식하지 못한 상태에서 발생하는 문제에 대해서는 누구도 책임지지 못한다. “책임”은 높은 분들이 가장 많이 사용하는 단어지만, 스스로는 지키고 싶어하지 않는 단어이기도 하다. 이제는 몰라서 못했다는 말을 듣지 않기 위해서라도 높으신 분들이 정확히 과제의 상태를 알 수 있도록 투명성을 확보해야 할 것이다.

# [ 다 타버리다(Burnout) ]

"번아웃(Burnout)"이라는 단어는 극도로 피곤을 느끼거나, 혹은 연료를 다 소모한 경우를 말한다. 소프트웨어 개발자로서 일하다 보면, 일상화된 야근과 주말 특근으로 인해서 피로감과 건강 이상을 호소하는 사람들을 만나게 된다. 주로 육체적인 피로감과 함께 일에 대한 책임감으로 스트레스를 많이 받는 상황에서 번 아웃 된다. 한 가지를 더 추가한다면, 직장 상사의 괴롭힘도 무기력감과 책임 회피 혹은 부정으로 이어져 번아웃을 부추기는 경향이 있다.

“침대가 가구의 역할 만 하는 것이 아니듯, 소프트웨어 개발자도 코딩 만하지는 않는다”. 바쁘게 일한 다음에는 마음의 여유가 필요하며, 육체적인 휴식의 시간도 가져야 한다. 가정도 돌봐야 하고 미래를 위한 자기 개발에 대한 투자도 해야 한다. 번아웃된 사람들의 공통된 이야기는 "미래가 없다"라는 말이다. 미래를 생각할 정도의 여유도 없이 달려가고 있다는 말이며, 이런 식으로 일 한다고 해서 성과가 날 것 같지도 않다는 뜻이다. 이게 정상적인 상황이 아님은 굳이 이야기하지 않더라도 알 수 있을 것이다.

그럼 어떻게 번아웃과 같은 상황을 방지할 수 있을까? 사실 모든 과제의 PL(Project Leader)은 개발인력의 기술력과 육체적, 정신적인 건강을 꾸준히 유지하는 것이 과제에 얼마나 중요한지 충분히 알고 있다. 물론, 알고 있다고해서 모든 문제가 해결되는 것은 아니지만, 될 수 있으면 그런 상황을 제공하지 않으려는 노력은 할 수 있다. 아래에 나열한 것들은 그런 노력에는 어떤 것들이 있는지 개인적인 생각을 정리한 것이다.

**01. 해야 할 일의 범위를 명확히 정하고, 개발자에게 마감 시간을 정하도록 맡겨라.**

; 대부분의 실무 개발자는 자기 일을 잘 파악하고 있으며, 얼마나 시간이 걸릴지도 대략적으로 예측할 수 있다. 일의 우선순위 및 상관관계를 파악하고 있으며, 테스트도 얼마나 걸릴지 정도는 추측할 수 있다. 일정을 가장 잘 알고 있는 사람은 개발의 실무를 맡고 있는 사람이며, 타인이 아무리 일정 단축을 이야기하더라도 크게 바뀌지는 않는다. 따라서, 그냥 믿고 맡기는 것이 최선이다.

**02. 일을 가능한 작게 쪼개서 한 달 동안 어떻게 일할 것인지 정하도록 만들어라.**

; 일에 대해서 명확한 목표의식을 심어주기 위해서 상세한 세부 활동을 정의하고, 이를 확인할 방법을 고안해야 한다. 그 일이 완료되었음을 확인하기 위한 절차도 좋다. 일은 구체적이어야 하며, 작게 나누어져 있어야 한다. 적어도 다음 한 달 정도의 일은 충분히 작게 나누어져야 한다. 하루나 이틀 단위는 좋지만, 한 주나 한 달을 넘어서는 일은 더 작게 쪼갤 필요가 있다.

**03. 만들어진 일정 계획을 모든 사람이 볼 수 있도록 하고, 담당자가 일정을 업데이트 하라.**

; 자기 일에 대한 진척도를 스스로 파악하고 관리할 수 있도록 만들어라. 자신의 의지로 일을 진행하고 있다는 느낌을 주는 것이 중요하다. 다른 사람이 자신의 미래를 결정하는 것이 아니라 스스로 자신의 미래를 개척하고 있다고 생각하게 만들어야 한다. 방법과 일정을 자신이 정할 수 있다면, 일의 목표만 설득하면 된다.

**04. 요구사항의 변경이나 추가가 발생하면, 반드시 우선순위와 일정을 조정하라.**

; 추가적인 요구사항이나 변경이 발생하면, 기존의 우선순위와 일정에 영향을 준다. 따라서, 반드시 재조정하는 과정이 필요하다. 이런 활동이 부족하면 개발자들은 수동적으로만 움직이게 된다. 특히, 요구사항의 변화가 일정에 영향을 준다는 사실을 모든 이해 관계자들이 알고 있어야 한다. 관련된 몇 사람 만 아는 변경으로인한 과제 지연은 이해 관계자의 신뢰에 손상을 주기 때문이다.

**05. 주어진 일에 대해서 완료한 경우에는 반드시 보상하라.**

; 사람은 칭찬받기를 원하고, 자신이 일한 부분에 대해서 보상 받기를 원한다. 칭찬은 “고래 만 춤추게 하는 것이 아니다”. 보상은 휴가가 될 수도 있고, 공개적인 자리에서의 박수, 혹은 상금 등등 그때 그때 상황에 따라 변할 수 있다. 큰 보상을 주는 것은 어렵지만, 작은 보상은 언제나 가능하다. 특히, 공개적인 자리에서 “인정”하는 것은 중요한 보상 방법이다. 물론, 그렇다고 개발자를 초등생처럼 대하라는 것은 아니다. “인정할 만한 일”을 한 사람에게 충분히 보상을 하는 것이다.

사람은 자신이 “할 일”과 “일해야 할 때”, “일하는 방법”을 결정할 수 있다면, 자발적으로 나서게 된다. 번아웃이 발생하는 이유는 육체적으로도 지쳤지만, 정신적인 손상도 심하다는 것을 의미한다. 충분한 휴식이 없다면 원래의 “의지”를 회복하기 힘들다. 스스로 능동적인 존재로 인식하지 못하고 시키는 일만 하는 존재가 된다면, 같은 일을 하더라도 스트레스의 강도는 더 높을 것이다.

소프트웨어 개발자는 자신에게 책임과 권한이 주어졌다고 느껴야지 동기를 얻는다. 정해진 일정을 맞추고, 정해진 기능을 구현하고, 정해진 일만 하는 것은 소프트웨어 개발자의 역량을 과소평가하는 것이다. 고도의 집중력을 요구하는 복잡한 문제를 해결하는 존재로 다시 되돌려 놓아야 할 것이다.

# [ 일을 적게해야 프로젝트가 성공한다. ]

물론, 모든 프로젝트가 그렇다는 것은 아니다. 하지만, 될 수 있으면 적게 일 할수록 성공할 확률은 높아진다. 특히, 소프트웨어를 개발하는 프로젝트라면, 주어진 요구사항을 만족시키기 위해서 최소한의 일만 할 방법을 찾는 게 중요하다. 소프트웨어의 복잡성은 코드의 길이와 조건문의 수에 비례하며, 복잡한 코드일수록 버그가 발생할 확률은 더 높아진다. 따라서, 가능한 코드를 적게 작성하는 것이 버그의 발생 가능성도 줄여준다.

가능한 일을 적게 하는 방법으로는 어떤 것이 있을까? 첫 번째는 이미 검증된 소프트웨어를 이용하는 것이 새로 개발하는 것보다 좋다. 즉, 검증의 노력과 코딩의 노력 줄이고, 기존 코드를 재활용해서 높은 ROI(Return of Investment)를 얻을 수 있다. 소프트웨어 개발이라는 것이 하루 이틀 동안 해온 일이 아니고, 이미 수십 년을 해왔기 때문에 기본적으로 풀어야 할 문제들에 대한 해법들은 많이 나와 있다. 정말로 자신이 풀어야 할 문제에 핵심 역량에 집중하고, 그 외의 부분들을 이미 나와있는 해법을 이용하면 된다.

두 번째 방법으로는 테스트의 도움을 받는 것이다. 물론, 테스터가 개발자당 한 명 정도로 1:1의 관계를 이루고 있는 회사도 있겠지만, 대부분 회사는 그렇지 못하다. 따라서, 여기서 받을 수 있는 테스터의 도움을 자동화된 툴로 대체해서 이용하는 방법이다. 가능한 한 적게 일 하면서도 버그를 줄 일 수 있는 유일한 개발 방법은, 개발자가 자신의 코드를 직접 자동화된 테스트 프로그램으로 검사하는 것이다. 물론, 처음에는 과제 진행이 더디게 보일 수도 있다. 하지만, 과제 완료가 가까워질수록 늘어나는 "월화수목금금금"은 충분히 막아줄 것이다.

보통의 개발자는 어떻게 하면 코드를 “빨리 많이 작성할 것인가”를 고민하지만, 똑똑한 개발자는 어떻게 하면 “작성해야 할 코드를 줄이고, 테스트를 더 많이 할 것인가”를 고민한다. 작성해야 할 코드만 작성하고, 불필요한 중복은 제거할 것이다. 물론, 항상 자동화된 테스트는 누가 이야기하지 않더라도 당연히 해야 할 일로 받아들인다. 오늘 100라인의 검증된 코드를 작성할 수 있다면, 단순히 계산해도 한 달이면 2,000라인의 코드가 검증된 상태로 남을 것이다. 일 년이면 적어도 20,000라인 정도는 충분히 자동화된 검증을 통과하는 코드를 만들 수 있을 것이다.

# [ 외주 과제의 관리 ]

회사가 자원(Resource)이 많아서 외주(Out-Sourcing)를 전혀 사용하지 않으면 좋겠지만, 가용할 수 있는 자원은 언제나 부족하다. 필요한 자원이 없으면 일단은 외부의 도움을 받아 문제를 처리해야 한다. 이 때 필요한 것이 외주 과제에 대한 관리 방법이다. 외주도 개발의 한 형태다. 따라서, 어떻게 관리하느냐에 따라 만들어지는 제품의 품질도 달라질 수 있다. 방치된 과제는 결과물의 수준이 낮은 것 처럼, 제대로 관리되지 않는 외주 과제도 형편없는 결과물로 그 대가를 치르는 경우는 흔히 볼 수 있다.

외주 관리의 수준은 외주를 진행하는 업체의 자원이나 노하우(Know-How), 내부적으로 가지고 있는 프로세스에 의존적 이겠지만, 과제를 진행하는 입장에서 아래와 같은 부분들을 기본적으로 잘 챙겨야 할 것이다.

1. **개발 항목**
2. **개발 항목에 대한 세부 설명 및 제약 사항**
3. **평가 시기 및 방법, 결과**
4. **산출물 내역 및 검토 결과, 변경 요청 내역 및 조치 결과**
5. **상세 일정 계획(일/주/월/분기/반기 단위 계획)**
6. **일정 업데이트(1~4주 단위 업데이트)**
7. **의사 결정 및 회의 기록**
8. **일일/주간/월간 보고**
9. **문제점 및 해결 방안, 상태**
10. **중간 데모(Demo) 계획(일정 및 데모 내용)**

물론, 여기서 나열한 열 가지 이외에 다양한 것들이 있을 수 있다. 중요한 점은 외주 개발에 대한 관리를 지속해서 해야 한다는 점이다. 즉, 적어도 자신의 과제를 하듯이 외주 과제를 관리해야 만 품질 높은 산출물을 얻을 수 있다는 것이다. 앞에서 나열한 것들도 실제로 자신이 과제를 한다고 가정했을 때 필요하다고 생각되는 것들을 보여준 것이다.

“품질은 과정에서 나오는 것”이지 결과만 보는 것은 아니다. 어떤 제품도 높은 품질을 달성하기 위한 중간 과정 없이는, 고품질을 만족시키지 못한다. “아는 만큼 보이고, 보이는 만큼 개선할 수 있으며, 개선된 만큼 좋은 품질”은 나오기 마련이다. 외주 과제를 제대로 진행하기 위해서는 최소한 그 일 만 전담해서 관리할 수 있는 충분한 역량을 가진 인력을 배정해야 할 것이다.

# [ 소프트웨어 플랫폼은 왜 만드는가? ]

기존에 이미 잘 동작하는 소프트웨어가 있다면, 도데체 왜 소프트웨어 플랫폼을 만드는 걸까? 얼마나 시간이 걸릴지도 모르는 일을 굳이 시작할 필요가 있을까? 사실 소프트웨어는 상품화를 전전하다 보면 금방 누더기가 된다. 빨리 새로운 기능을 추가해야 하고, 다양한 제품 설정을 위해서 분기되는 코드도 점차 늘어나게 된다. 다양한 컴파일 옵션이 추가되고, 각각의 옵션에 따라 코드 내부의 분기도 증가하는 추세를 보인다.

결국, 기존 코드를 유지보수하는 것이 새로 개발하는 것 보다 못한 순간이 오게 된다. 하지만, 이 때는 플랫폼을 새롭게 시작하기에는 이미 늦다. 미리 그런 상황이 오기 전에 준비하고 갈아탈 채비를 했어야 한다. 하지만, 기존 소프트웨어 개발자들은 새로운 플랫폼을 만드는 것을 거부하고, 이미 상품화에 적용한 코드 만 옹호하려고 할 것이다. 특히, 일정이 압박이 심할수록 변경을 최소화하려는 경향이 강해질 것이다. 따라서, 플랫폼을 새롭게 만들기 보다, 현재 가지고 있는 코드를 플랫폼이라고 부르기 시작하게 된다.

문제는 그런 식으로 플랫폼이라고 이야기한다면, 모든 개발된 모든 소프트웨어를 플랫폼이라고 이야기하더라도 “아니요”라고 할 수 있는 기준이 없다. 예를 들어, 저가/중가/고가 제품에 대해서 각각의 플랫폼이 존재하게 될 것이다. 추가적으로 하드웨어가 조금 바뀌더라도 용도에 맞춰서 다시 새로운 플랫폼을 정의하게 될 것이다. 즉, 제품이 곧 플랫폼이라는 말이 되고 만다. 따라서, 이런 상황이라면 상품 개발과 플랫폼 개발은 서로 나누어져 개발이 진행되어야 한다. 물론, 그렇게 한다고 해서 신규 플랫폼에 대한 기존 개발자의 반감은 쉽게 사라지지 않을 것이다.

플랫폼을 만드는 이유는 신규 개발해야 할 소프트웨어의 비중을 낮추고, 기존에 이미 검증된 소프트웨어를 재활용해서, 빨리 시장에 출시할 제품을 만들자는 것이다. 따라서, 적어도 플랫폼이라고 한다면, 빠른 기능의 추가가 가능해야 하며, 새로운 환경에 쉽게 적용될 수 있는 구조를 내포하고 있어야 한다. 코드 자체가 모듈화 계층화를 철저히 지켜야 하며, 변경은 가능한 국지적으로만 발생해야 한다. 당연히 변경의 영향은 다른 모듈에 영향을 주지 않아야 한다. 처음 부터 이런 플랫폼을 만들기는 어렵기 때문에, 기존의 소프트웨어가 이미 있는 경우에는 구조 개선에 초점을 맞춰서 플랫폼을 구성하게 된다.

기존의 소프트웨어 구조를 개선해서 플랫폼을 만들어도 기존 개발자들의 반발은 어쩔 수 없이 겪어야 한다. 특히, 개선이 완료되고 적용할 시점이 다가오면 점점 더 거세게 반발할 것이다. 이미 상품화를 진행하고 있는 제품에 들어간 기능이 지원되지 않는다는 불평, 개선된 부분이 별거 아니라는 말, 자신들의 의견이 반영되지 않아서 자신들이 적용할 이유가 없다는 생각 등등 다양한 불만들이 나올 것이다. 하지만, 정말 문제가 있는 것은 그런 이유를 핑게로 개발된 플랫폼의 적용을 연기하는 관리자들의 결정이다.

세상에 완벽한 소프트웨어가 없듯이 플랫폼도 점점 더 개선되고 발전해 나갈 뿐이다. 모든 기능이 지원되고, 모든 내외부 요구사항을 다 반영한 후에 플랫폼을 적용하겠다는 말은, 그냥 “사용하고 싶지 않다”는 말일 뿐이다. 따라서, 기존 개발자라는 산을 넘지 못하면 소프트웨어 플랫폼을 만드는 것은 불가능하다. 만들었다고 하더라도 상품화에 적용하는 것은 또 다른 차원의 일이 되고 만다. 플랫폼에 대한 의지를 가지고 기존 개발자들의 반발을 해소할 수 있는 방법을 찾지 못한다면, 오히려 개발에 큰 낭비만 초래하고 말 것이다. 의지가 없다면 플랫폼을 개발하는 사람들도 힘을 얻지 못한다.

실무에 능하다고 모든 작성하는 코드의 품질이 우수한 것은 아니다. 실무에 능할수록 도메인 지식에는 강하지만 프로그래밍 이론에는 약하다. 실제로 전문 개발자라고 불리는 대부분이 소프트웨어 공학을 공부한 경험이 없는 게 사실이다. 그리고, 소프트웨어 공학 지식이라는 것이 별거 아니라고 생각하기도 한다. 도메인에 익숙한 사람일수록 자신의 코드에 문제가 있다는 것을 받아들이지 않는 경향이 있으며, 동작하는 코드는 절대 고치려고 생각하지 않는다. 물론, 문제가 발견되면 고치기야 하겠지만, 그것 역시 최소로 고치거나 가장 쉽게 고치려는 경향을 보인다. 당연히 구조적이거나 모듈화된 코드를 만들지는 않는다(모듈이라고 말은 하지만, 모듈이 가져야하는 특징을 찾아보기 힘들다).

왜라는 이유보다는 “해보니 잘 되더라”, 혹은 “이미 해봤다”는 논리가 가장 강력한 방패다. 물론, 나중에 안되면 더 큰 비용이 든다는 점과 이미 해본 것보다 더 좋은 것이 있다는 것은 모르고 하는 소리일 뿐이다. 플랫폼 개발과 상품화 개발이 서로 다른 것이 아니며, 결국 협동해야 만 더 좋은 결과물을 만들어 낼 수 있다. 플랫폼을 만드는 궁긍적인 이유는 결국 “돈을 빨리 벌자”일뿐이다. 효율적으로 더 적은 인력을 투입하고 더 빨리 제품을 만들어, 남들보다 더 많이 팔자는 단순한 이유다. 회사가 돈을 더 많이 벌 수 있다면, 구성원도 당연히 더 많은 돈을 벌 수 있다. 거부감이 크다는 이유 만으로 좋은 기회를 놓친다면, 결국 그 피해도 구성원의 몫으로 고스란히 돌아갈 수밖에 없는 것이다.

# [ 프로젝트가 실패하는 이유? ]

소프트웨어 개발 프로젝트는 왜 실패하는 걸까? 여기서 말하는 실패란 주어진 돈(예산)과 일정, 구현된 기능을 따졌을 때 적어도 하나가 만족되지 않았다는 것을 말한다. 성공과 실패를 이런 기준으로 나눈다면 아마도 이 글을 읽는 대부분의 사람들은 자신이 참여했던 프로젝트가 실패였다는 것을 깨닫게 될 것이다. 물론, 성공한 사람들도 많겠지만, 개인적인 생각으로는 10%도 안되는 과제만 성공했다고 이야기할 수 있을 것이다. 그렇다면, 과제를 실패하게 만드는 요인은 뭘까? 아래에 그 몇 가지를 적어 보았다.

**첫 번째 이유는 말도 안 되는 일정이다.**

**두 번째 이유는 부족한 사람이다.**

**세 번째 이유는 요구사항의 끝없는 변경이다.**

**네 번째 이유는 그나마도 있던 경비의 축소다.**

우리 같은 소프트웨어 개발자는 위에서 나열한 것들과의 싸움에 익숙해져야 한다. 아니, 정확히 말하면 패배에 익숙해져야 한다. 물론, 항상 패배하는 것보다는 한 두 번 정도는 승리해도 좋다. 하지만, 항상 경계해야 할 것은 패배가 만성화 되는 것이다. 아무런 느낌도 없이 패배를 받아들인다면, 결코 성공적인 프로젝트를 할 수 있는 기회는 오지 않는다. 최소한 성공은 못 하더라도 근처까지는 가야 하지 않을까? 실패에서 배운 사람들은 발전하지만, 실패를 그냥 잊는 사람들은 발전하지 못한다. 실패한 원인을 찾고 그것을 개선하려고 새로운 시도를 해야지만 변화가 생긴다.

말도 안 되는 일정을 이야기할 때는 지난 경험을 통해서 얻은 각종 통계적인 것들을 제시해야 하고, 부족한 사람에 대해서는 부족분만큼 업무에 반영하지 않아야 한다. 끝없는 요구사항의 변경에 대해서는 이해관계자와 지속적인 논의가 필요하며, 과제의 실제 진행 상황을 잦은 데모를 통해서 알려야 한다. 경비가 부족하다면 해결할 다른 대안을 찾는 것이 과제 책임자가 맡은 해야할 일이다. 이러한 모든 것들이 실패로 끝나더라도, 적어도 경험이라는 것은 남기는 장사가 될 것이다.

일정 계획은 변경되기 위해서 존재하며, 사람은 언제나 넉넉하게 주어지지 않는다. 요구사항을 다 구현한다고 제품이 팔리는 것도 아니며, 돈이 부족한 것은 어디를 가나 마찬가지다. 과제 실패가 제품의 실패를 말하는 것도 아니기에, 부족한 상황에서도 반드시 얻어야 할 것들이 있다. 첫 번째는 실무 개발자의 신뢰를 얻어야 한다. 두 번째는 실패한 이유를 정확히 파악하고 다음 번 도전에서는 성공할 수 있는 방법을 찾아야 한다. 요구사항의 변경은 어쩔 수 없지만, 추가되는 요구사항은 관리해야 한다. 부족한 예산은 별다른 도리가 없지만, 나머지 것들이라도 일단 챙겨야 할 것이다. 한 번은 실패할 수 있지만 반복되는 실패는 실력이라고 밖에 보지 않는다.

# [ 규칙과 규율(Rule & Discipline) ]

"Good To Great"이라는 책에서 평범한 회사에서 위대한 회사로 성장한 조직은 “규칙과 규율”이 서 있었다는 것을 가장 중요한 성공 요인으로 꼽고 있다. 소프트웨어는 혼자서 만드는 것보다 여러 사람으로 이루어진 팀이 개발한다. 따라서, 조직이 관련된 소프트웨어 개발에서도 이런 규칙과 규율이 그저 그런 수준의 프로그램에서 품질이 우수한 프로그램을 만드는데 핵심이 될 수 있다.

규칙(혹은, 규정)은 간단해야 한다. 즉, 여러 가지 사례(Case)에 잘 부합하는 간단한 규칙을 만드는 것은 대단히 어려우며, 반드시 지켜야 할 것들만 명시적으로 정하는 것이 좋다. 복잡하고 긴 규칙은 기억하기도 힘들고 관리가 되지도 않는다. 구약성서(종교에 대해서는 별로 아는 것이 없지만)에 나오는 "십계명"도 딱 열 가지만 지키라고 말하고 있다. (사람이 한 번에 기억할 수 있는 개수는 “7+-2[다섯에서 아홉개 사이]”라고 한다. 따라서, 열 가지는 사실 조금 많은 편이다.)

문제는 규칙을 어떤 식으로 적용할 것인가를 정하는 것이다. 규칙을 적용하는 것을 규율로 만들어야 한다. 규칙이 방법이라면 그것을 꾸준히 적용하는 것이 규율이다. 따라서, 정한 규칙을 규율로 만들기 위해서는 팀원의 동의를 얻어야 한다. 모든 팀원의 동의가 없다고 하더라도, 최소한 설득하는 과정은 있어야 한다. 공유되지 않는 규율은 아무도 따르려고 하지않기 때문이다. 규칙만 있고 규율이 없다면, 그것 역시 무의미하다. 따르지 않는 규칙은 만드는 것도 낭비다.

코딩의 일차적인 검증은 테스트를 통해서 이루어지지만, 통합 과정에서 필요한 것은 코드 리뷰(Code Review)다. 하지만, 코드 리뷰를 한다고 이야기하는 사람들 조차도 코드 리뷰에서 무엇을 중점적으로 확인할 것인가는 잘 모른다. 무엇을 봐야 할지를 정하지 않았다면, 지금 부터라도 만들기 바란다. 그리고, 처음 한다면 내용보다는 외적인 표현을 먼저 보는 것이 가장 쉬울 것이다. 사실 내용은 충분한 이해를 필요로 하며, 시간적인 비용이 지불되어야 하기 때문이다. 따라서, 관련된 사람들만 제대로 볼 수 있다.

**01. 코딩 룰을 준수하는가?**

- 띄어쓰기(Indentation), 공백(Blank), "{}"의 시작과 끝, 특정 코딩 스타일의 사용 여부, 함수, 클래스, 변수, 상수 등의 이름을 주는 방법, 주석 양식 등을 잘 지키고 있는지 본다. 눈으로 확인하기도 쉽지만, 특정 도구를 사용할 수도 있다.

**02. 코드가 읽기 쉬운가?**

- 코드는 컴퓨터 만을 위한 것이 아니다. 사람이 읽을 수 없는 코드는 유지보수성과 멀어지게 되고, 결국 시간이 흐르면 새로 작성하는 비용이 고치는 비용보다 싸지게 된다. 가독성은 다른 모든 품질보다 우선적으로 생갹해야 하며, 작성자가 아니라 타인의 눈을 통해서 확인되어야 한다.

**03. 코드의 추상화 수준이 일정한가?**

- 같은 수준의 추상화 수준을 지키고 있는지 본다. 즉, 어떤 부분은 지나치게 자세하고, 어떤 부분은 특정 함수를 그냥 호출하는 정도만 있다면, 추상화 수준이 서로 다른 것들이 섞여 있는 뜻이다. 지나치게 자세한 부분은 다른 함수로 분리하고, 너무 추상적인 부분은 그냥 코드를 직접 가져와도 될 것이다.

**04. 잘 정의된 한 가지 일을 제대로 하는가?**

- 코드는 맡은 일을 제대로 해야지 의미가 있다. 코드를 읽었을 때, 여러 가지 일을 하는 부분이 하나의 논리적인 구분 단위(디렉토리, 파일, 함수, 코드 블록 등등)에 섞이지 않도록 한다. 한 번에 하나의 일 만 집중적으로 하는 것들을 만들어서, 그것을 조합하는 방식으로 처리하는 것이 문제를 해결하는 기본 전략(Divide and Conquer)이다.

**05. 작성된 코드에 대해서 테스트는 있는가?**

- 가장 힘든 것이 테스트다. 특히, 개발자가 가장 어려워하는 부분이 단위 테스트와 통합 테스트다. 가장하기 어려운 부분이지만, 이것을 넘어서야 만 실질적인 소프트웨어 품질을 확보할 수 있고, 가족들과 함께 저녁을 같이 먹을 수 있다. 부품 단위의 품질이 확보되어야, 모듈, 서브 시스템, 시스템 등의 품질을 확보할 수 있는 기초가 만들어지기 때문이다.

일단은 위와 같은 다섯 가지 방법에 집중해서 코드를 리뷰하는 것이 좋을 것이다. 나머지 더 필요한 부분이 있다면, 조금씩 추가해 나가면서 일하면 된다. 코드 리뷰를 진행하면 도메인 영역에 중요한 점검할 항목들을 추가적으로 발굴할 수 있을 것이며, 잦은 문제가 발생하는 부분에 대한 새로운 규칙을 만들 필요도 생기게 될 것이다. 따라서, 일단은 간단한 규칙을 정하고, 그것 만은 반드시 지키는 것을 규율로 가져가야 할 것이다.

모든 것을 사람에게 맡겨둔다는 것은 객관적인 공통된 이해를 끌어내기가 어려운 경우도 있을 것이다. 그리고, 관리자들은 모든 것이 값으로 표현되기를 원한다. 왜냐하면, 자신보다 더 높은 관리자에게 "측정할 수 없다면, 개선할 수 없다."라는 말로 설득해야 하기 때문이다. 수치적인 품질을 나타내려면 자동화된 도구를 사용하는 수밖에 없다. 즉, 도구를 사용하는 것이 객관적인 수치를 확보해주기 때문이다. 객관성이 결여된 수치는 투명성의 확보나 실무자들의 행동 변화를 일으키지 못한다.

문제를 짊어지고 가는 사람에게는 이러한 도구가 성가신 존재다. 자신이 개발한 부분에 대해서 비판받기를 원하지 않기에 분석된 결과를 받아들이지 못하는 경우도 간혹 있다. 하지만, 다른 과제의 측정된 수치를 가지고 비교를 한다면(아마도 더 품질이 높은 소프트웨어를 고르거나, 혹은 인터넷에서 적용한 예를 찾아서 이야기하는 것이 좋을 것이다.), 결과를 수긍할 수도 있을 것이다. 때로는 자신의 과제는 다른 과제와 다르다는 점을 강변하기도 한다. 물론, 다른 과제와 도메인이 다르기는 하겠지만, “좋은 코드”가 가져야 할 특징까도 다른 것은 아니다. "개선은 현실을 인정하는 것부터 시작해야 한다."

규칙을 정하고 개발자들이 따르게 만들어야 한다. 또한, 객관적인 수치를 사용해서 시각적으로 표현할 수 있어야 한다. 그런 모든 활동이 바로 소프트웨어 개발팀에게는 규정과 규율을 가지고 일하는 마인드를 심는 것이다. 결국, 규칙과 규율을 적용한 성과는 코드의 품질로 나타나며, 개발자에게 더 많은 여유를 줄 것이기 때문이다. (물론, 일이 줄어들지는 알 수 없다.) 소프트웨어 개발팀을 처음 맡은 사람은 팀 내에서 불합리한 것을 없애는 것도 중요하지만, 합리적인 새로운 규칙과 규율을 정하는 것도 게을리해선 안된다. 목표는 항상 효과적이고, 발 빠른 대응을 할 수 있는, 품질이 우수한 소프트웨어를 만드는 것이다.

# [ 과제의 일정 지키기 ]

아마도 개발자라면 가장 힘들어하는 것이 과제의 일정을 지키는 일이다. 왜 이렇게 과제의 일정을 지키는 것이 힘들까? 사실 과제의 일정을 지키는 것은 거의 기적이라고 해야 옳을 것이다. 왜냐하면, 미래를 예측하는 일이기 때문이다. 미래를 예측하는 데 있어서 정확도를 높일 수 있는 유일한 방법은 과제를 해 본 경험이다. 즉, 유사한 경험을 통해서 이미 익숙해진 상황에서 만 가능하다. 하지만, 현실은 개발자들에게 항상 새로운 것을 요구하고, 그것에 대한 모든 위험을 파악하지 못한 상태에서 일을 시작하기 때문이다.

사실 무슨 일을 해야 할지 제대로 파악하지 못한 상태에서 시작하는 일이 많다. 상품 기획에서 주어지는 라인업(Line-Up) 계획은 미리 논의하게 되고, 내년에 해야 할 일들을 예측하면서 약속을 해야 하기 때문이다. 이렇게 이야기하는 약속이 과연 정확한 예측이 될 수 있을까? 주어진 요구 사항에 대해서 최대한 자세히 분석하고, 이를 바탕으로 WBS(Work Breakdown Structure)를 만들어서 “MS-Project”와 같은 툴로 일의 양에 대해 산출을 하고, 사용할 수 있는 자원(인력이나, 돈, 기타 등등 사용할 수 있는 모든 주어진 환경)을 배정해서 계획을 세운 상태에서나 가능하다.

한꺼번에 많은 과제가 몰리는 상황에서 이러한 것들을 다 하기는 사실 거의 불가능하다. 따라서, 유일한 대안은, 될 수 있으면 아무것도 바꾸지 않으려고 개발자는 방어적인 자세로 일정계획을 추정하게 된다. 가능한 많은 일정 버퍼를 두려고 일정을 늘리기도 하고, 충원 계획은 고려하지 않은체 최소의 기능만 구현할 수 있는 계획을 만든다. 사실 이것 마저도 지켜지지 않을 가능성이 있다. 새로 만들어야 하는 제품만 있는 것이 아니라, 이미 출시한 제품에 대한 유지보수도 추가해야 하기 때문이다. 따라서, 새로운 제품에 투입할 수 있는 인력은 전체 인력의 절반도 안될 가능성이 있다.

방어적인 자세로 만들어진 제품은 절대 경쟁력을 가지지 못한다. 싸다는 점을 제외하면 시장을 이끌 수 있는 제품은 만들지 못하게 된다. “왜 세계 1위를 못하는가?”를 묻는다면, 우리에게는 세계 1위의 제품보다는 과제 일정을 맞추는 것이 더 중요하기 때문이다(관리자들이 그것을 원하기에). 싸고 좋게 만들어서 많이 팔면 되지 않느냐고 묻겠지만, 이미 싸게 만드는 것에 특화된 경쟁 상대들이 있음을 잊어선 안 된다. (그들은 이미 좋게도 만들고 있다.) 우린 싼 제품으로 그들을 넘어설 수 없다. “차별화된 고부가 제품을 적절한 시간에” 만들어내야 만 그나마도 힘들어진 생명줄을 조금이라도 연장시킬 수 있을 뿐이다.

고품질의 제품을 만들기 위해서는 더 적게 일해야 한다. 일을 많이 할수록(기능이 많아질수록) 문제가 많은 제품이 될 가능성이 높다. 모든 것이 다 되는 제품은 하나의 기능이라도 망가지면 아무것도 동작하지 않는 시스템이 될 것이다. 따라서, “선택과 집중”을 해야 한다. 잘 사용되지 않는 기능들은 과감히 삭제하고, 사용 편의성에 집중하는 것이 좋다. 복잡한 기능은 없애거나 자동화를 해야 할 부분이다. 사용자들이 놀랍다고 생각할 수 있는 기능 몇 가지를 넣어서 파는 것이 오히려 제품의 경쟁력을 갖추는데 핵심이 될 것이다. 물론, 그 핵심도 쉽게 사용할 수 있어야 할 것이다. 일을 줄이기 위해서는 필요없는 일을 안하는 방법이 최선이다. “필요”의 기준은 당연히 “시장의 가치”로 따져야지, 힘있는 누군가의 선택으로 확정해선 안될 것이다.

우린 세계 최고의 제품을 만들지 못하는 것에 익숙하다. 세계 최고가 되기를 원한다면, 과제의 일정이나 따지는 일을 당분간 하지 않아야 한다. 급하다면, 기존의 제품으로 잠시 시간을 벌어야 한다. 사용자 편의성을 개선하고 기능을 축소하는 대신 원가를 절감한 모델을 선보이는 것이다. 대신에 어떤 기능들이 정말 필요하고, 어떤 기능들을 제외할지, 그리고, 그것을 사용하는 사람들이 느끼게 될 만족을 상상하도록 해서 차기 제품을 기획해야 한다. 마지막으로 그런 제품을 만들어가는 개발자들의 즐거운 표정도 포함할 수 있어야 할 것이다.

# [ 무엇이 문제인지 아는 것이 문제풀이의 시작 ]

소프트웨어를 개발하는 사람들은 항상 개발하는 시간보다 디버깅하는 시간이 더 많다고 투덜댄다. 사실 대부분 시간을 버그를 찾고, 그것을 해결한다고 보내고 있다. 따라서, 개발의 생산성은 당연히 버그를 얼마나 더 빨리 풀 것인가에 집중하기 마련이다. 하지만, 빨리 푼다고 문제의 원인이 없어지는 것은 아니다. 그럼 어떻게 버그를 찾아서 해결하는 것이 효과적일까?

버그를 푸는 원칙은 간단하다. 버그를 만들어내는 부분을 지역화(Localize)시키는 것이다. 버그가 일어난 원인이 다른 곳과 관련성을 맺지 않게 하는 것이 핵심이다. 따라서, 버그 줄이는 조치는 이미 코딩을 완료시키는 부분에서 있는 것이 아니라, 코딩이 시작되기 전부터 시작된다고 생각해야 한다. 즉, 어떤 전략을 가지고 코딩을 할 것인가가 버그를 찾아내는 시간을 줄이는 중요한 부분이라는 점이다.

버그의 유형으로 버그가 어디쯤 있을 것인가를 판단하기는 쉽지 않다. 왜냐하면, 소프트웨어 버그는 시스템의 다양한 부분과 연결되어 일어나기 때문이다. 어려운 유형의 버그는 시스템이 상당 시간 동작한 후에 간혹 발생하는 것들로, 이것을 해결하기 위해서는 제품에 대한 경험과 지식을 함께 요구하기 때문이다. 간혹 발생한다는 말은 대부분의 상황에서는 유효한 값이 되지만, 특정 상황에서는 문제가 된다는 뜻이다. 따라서, 상황을 재현하기도 어렵고, 버그가 발생했을 때와 파악되는 순간이 달라질 수도 있다.

쓰레드와 관련된 대표적인 버그는 스택 오버플로우(Stack Overflow)를 들 수 있다. 즉, 쓰레드를 위해서 주어진 스택의 크기를 넘어서는 접근이 발생하는 경우로, 다른 쓰레드가 사용하는 메모리 영역을 파고들어, 컨텍스트 스위칭(Context Switching)이 발생할 경우 엉뚱한 주소에 있는 값으로 제어를 옮기도록 만든다. 이때 발생할 수 있는 오류는 “세그먼테이션 폴트(Segmentation Fault)”나 “정의되지 않은 명령(Undefined Instruction)”등이 있을 수 있다. 이 때는 스택을 기존의 크기의 두 배 정로 조정하는 것으로 해결할 수 있다. 증상은 주로 불규칙한 위치에서 시스템이 죽는(System Crash) 현상으로 나온다.

정적분석 툴을 이용한(Coverity Integrity Check와 같은 툴) 자원 누수(Resource Leak)나 잠재적인 오류를 찾아내는 것도 실제로 수정해야 할 버그의 수를 줄이는데 효과적인 방법이다. 물론, 이러한 툴은 비용이 만만찮다는 점과 문제가 발생했을 때 해결 유형을 일괄적으로 모든 개발자들에게 전파해야 한다는 점에서 어려움이 있다. 중요한 소프트웨어라면 충분한 ROI(Return of Investment: 투자효율)를 얻을 수 있을 것이다. 하지만, 정적 분석 툴이 모든 문제의 해결책은 되지 못하며, 실행 시에 발생할 수 있는 다양한 오류는 사람의 손을 거쳐야지 만 해결할 수 있다.

컴파일러에서 발생시키는 오류나 경고(Warning)은 반드시 제거되어야 한다. 최소한 기계적으로 알려주는 이러한 부분에 대해서는 기본적으로 문제가 생기지 않아야 다음 단계로 넘어갈 수 있다. 다른 사람에게서 전달받은 코드를 컴파일 했을 때, 수많은 경고 메지시(Warning)가 나온다면 코드 품질에 대해서 의구심을 가지게 되는 것이 일반적이다. 실제로 컴파일러의 경고 메시지를 무시하는 경우도 많으며, 심지어 경고 메시지를 끄고 컴파일 하는 경우도 종종 있다. 컴파일러의 경고 수준은 항상 최상으로 하는 것이 올바른 방법이며, 모든 경고 메시지는 전부 삭제할 수 있도록 코드를 수정해야 한다.

버그를 국지화(Localize)하기 위해서 가장 쉬운 방법은 함수(혹은, Member Function or Method)의 입력과 출력값을 조사하는 것이다. 함수에 대한 호출 후, 복귀(return)값을 확인하는 것과 함수로 전달되는 파라미터 값이 올바른 범위를 가지는지 검사하는 것이다. 검사는 디버깅 시에 만 필요할 수도 있고, 실행 시에도 필요할 수 있다. 시스템 호출(System Call)도 실패할 수 있기에, 어떤 경우라도 항상 실패를 대비하는 것을 습관처럼 가져가야 한다. 오류는 주로 변수의 유효 범위의 경계 부근(Boundary Condition)에서 많이 발생한다. 따라서, 극한의 값을 가질 수도 있다는 것에 대한 가정은 항상 해야 한다.

메모리 누수(Leak)과 관련된 문제는 시스템 자원의 할당(Allocation)과 해제(Release)의 불일치에서 발생한다. (또한, 암묵적인 컴파일러의 개입에 의해서도 발생할 수 있다.) 즉, 할당과 해제를 하는 시점이 달라서, 메모리를 해제하지 않아서 발생한다. 주로 72시간 내의 동작에서 발생할 수 있는 버그로, 주기적으로 시스템의 자원 할당량을 조사해 보면 어디쯤인지 판단할 수 있다. 툴을 이용해서 원인을 찾는 것이 가장 빠른 방법이지만, 때로 툴에 의해서 찾지 못하는 경우도 빈번하다. 예를 들어, 특정 “3rd party”로 부터 전달받은 라이브러리를 잘못 사용해서 생기는 자원 낭비는 상당히 까다로운 문제다. 이때는 “3rd party” 라이브러리 자체를 같이 모니터링 해야 하는 부담이 생긴다.

메모리의 할당과 해제의 위치를 일치시켜주는 것이 효과적인 메모리 누수를 줄이는 방법이다. 가능한 메모리 할당과 해제는 같은 곳에서 발생하도록 만들어야 한다. 같은 함수, 같은 파일, 같은 클래스, 같은 디렉토리, 같은 서브시스템등과 같이 동일한 수준에서 할당과 해제를 할 수 있도록 만들어주면 관리하기가 편하다. 물론, 반드시 한 쌍이 존재할 수도 있지만, 시스템이 종료될 때까지 사용될 것이라고 가정하면, 꼭 그렇지 않을 수도 있다. 이 때는 한정된 메모리를 풀(Pool)과 같이 만들어서 동적인 상황을 제어할 수 있는 방법이 있다.

버그는 소프트웨어 개발자의 부주의를 파고들며, 그런 것은 항상 프로그램 곳곳에 널려있다. 작은 기초적인 방법과 원칙들을 잘 지켜나가면, 이러한 시간의 낭비를 미연에 방지할 수 있다. 삶을 좀 더 편하게 만들어 주기 위해서는, 항상 과제를 시작할 때 이러한 원칙들을 만들어 모든 개발자가 공유할 수 있도록 하는 것이 중요하다. 사소한 것들이 위협으로 다가오지 않도록 만들기 위해서는 습관적인 행동에 대한 기준과 꾸준한 반복이 필요하다. 좋은 습관은 문제의 원인을 근본적으로 차단하는 반복된 행동으로 정착시킬 수 있다.

# [ 소프트웨어 개발은 사람에 대한 이해다. ]

흔히들 소프트웨어 개발 관리라고 하면, 코드의 품질(버그의 개수나 빈도, 가독성 등 눈으로 확인할 수 있는 것들)과 같은 외적인 것을 관리하려는 경향이 있다. 하지만, 사실 가장 중요한 것은 코드를 만들어내는 사람이 핵심이다. “어떤 사람을 쓸 것이며 쓰고 있는 사람을 어떻게 관리할 것인가”가 최종 결과물에 큰 영향을 주게 된다.

관리자는 경험적으로 과제의 진척도를 남은 “업무의 양이나 버그의 수”로 파악하려는 경향이 강하다. 따라서, 이런 외적인 수치들에 대해서 항상 꾸준한 보고를 원한다. 하지만, 남은 업무의 양은 사실 정확히 예측하기가 난해한 부분이며, 수정해야 하는 버그의 수는 소프트웨어 개발자가 보는 관점에서 얼마든지 달라질 수 있다. 따라서, 오히려 관리적인 측면에서 과제의 완성도를 보고 싶다면, 다른 수치를 기억하는 것이 바람직하다. 예를 들어, 현재 코드가 가지고 있는 복잡도가 얼마나 되는지, 중복된 코드는 몇 라인이나 되는지, 테스트 커버리지가 얼마나 되는지 등등 의미도 있고 보기에도 좋은 수치들은 얼마든지 있다.

그럼, 그런 숫자들이 왜 소프트웨어 개발자라의 관리와 관련된 것일까? 사실 이런 숫자들은 이미 검증된 것이라고 봐도 무방하다. 복잡도가 높다는 말은 코드 수정이 어렵다는 말이며, 버그 발생 가능성이 크다는 의미다. 중복된 코드는 버그이 온상이 될 수 있으며, 테스트되지 않은 코드는 언제든 버그를 유발할 수 있다는 것이다. 사실 이런 부분들이 다 지켜진다고 해서 버그가 없다고 이야기할 수는 없지만, 적어도 소프트웨어 개발자의 기본기에 대해서 관심을 가질 만한 근거는 된다. 즉, 코드 자체가 가지는 좋고 나쁨을 직접적으로 측정할 수는 없지만, 간접적으로라도 품질을 알 수 있는 지표가 될 수 있다는 점이다.

기본을 지키도록 만드는 것이 관리라고 한다면, 소프트웨어 개발자들이 제대로 일하고 있는지 확인할 수 있는 방법을 찾아야 만할 것이다. 따라서, 앞에서 본 수치들은 그것을 알 수 있는 한 가지 방법이라고 생각해 볼 수 있다. 복잡하지 않고 이해하기 쉬운 코드의 구현, 중복된 코드를 제거하는 리팩토링, 자동화된 테스트 스크립트를 만드는 것 등이 실질적인 소프트웨어 개발자의 기본기이기 때문이다. 관리란 결국 해야할 일을 적시에 할 수 있도록 만들어주는 것이며, 기본이 없다면 그것을 갖출 수 있도록 도와주는 일도 포함해야 한다. 일정이나 버그의 수만 가지고는 이런 기본 지키기를 제대로 하고 있는지 제한적으로만 확인할 수 있을 뿐이다.

규칙이 있는 것은 없는 것보다 좋다. 잘못된 규칙이라도 일단 만들어두면 그 자체가 힘이된다. 그리고, 잘못된 결과를 얻으면 무엇을 고처야 할지도 알게된다. 개선이란 피드백을 통해서 지속적으로 긍정적인 변화를 추구하는 활동이며, 정해진 규칙이 없다면 어디서 부터 시작해야 할지도 알 수 없게된다. 과제들을 수행하는 과정에서 찾아내는 오류는 개선을 위한 피드백이 될 수 있으며, 점차 도메인에 특화된 규칙으로 수렴하게 될 것이다. 지표는 개선을 위한 방향임과 동시에 목표를 설정하는데도 도움을 줄 수 있다. 무엇을 잘했는지 알 수 없다면 얼마나 더 해야 할지도 알지 못하게 된다. 사람은 목표를 정할 수 있어야 행동으로 옮긴다.

소프트웨어는 사람이 만드는 결과물이며, 결과물의 품질을 높이기 위해서 필요한 것은 사람이 가지는 “행동의 방향성을 찾는 것”이다. 코드의 품질은 그 일을 담당하고 있는 사람의 품질에 대한 인식에 따라 차이가 나며, 객관적인 기준이 없이는 개선하기도 어렵다. “측정할 수 있어야 개선할 수 있다는 말”은 무엇을 개선할 것인지 목표를 정해야 한다는 의미다. 따라서, 소프트웨어 개발 관리란 품질 목표를 설정하고 사람들을 목표한 방향으로 움직이도록 만드는 일이 되는 것이다. 사람의 태도가 변하지 않으면 코드의 품질도 달라지지 않는다. 코드의 품질은 개발자의 마음에서 나오며, 손 끝에서 완성되기 때문이다.

# [ 생태계의 승자가 되는 길 ]

제품 개발에서는 플랫폼이라는 말을 많이 사용한다. 다양한 문맥에서 사용되기에 의미의 혼동이 있을 수도 있겠지만, 어쨌든 플랫폼이라고 이야기하는 순간 재활용률이 높은 소프트웨어를 만들어야 한다는 것은 이미 잘 알려진 사실이다. 하지만, 이러한 재활용률은 사람들에게 흔히 잘못된 환상을 심어주는 경향이 있다. 즉, 재활용하면 더 빠르게 제품을 만들 수 있다고 생각하게 만든다. 하지만, 플랫폼을 사용한다고 해서 크게 달라지는 것이 없을수도 있다.

재활용이 더 빨리 제품을 만들어내기 위해서는 플랫폼이라고 부를 수 있는(정의할 수 있는) 기준에 달려있다. 만약 기존에 개발한 소프트웨어를 편의상 플랫폼이라고 부른다면, 코드 품질 수준에 따라 개발 기간은 달라질 것이다. 품질이 높지 않은 코드를 재활용하기 위해서는 전체 범위에 걸쳐 코드 수정이 발생할 수도 있다. 당연히 버그 발생도 많아질 것이고, 안정화시키기 위한 시간도 길어질 수밖에 없다. 새로운 기능을 추가하는 것도 마찬가지다. 따라서, 플랫폼의 기준을 정의하지 않는다면, 재활용을 통한 개발 시간 단축이란 꿈같은 일일 수밖에 없다.

제품의 개발 일정 단축은 구현해야 할 기능의 수와 이용할 수 있는 자원에 의존적이다. 이런 저런 기능을 다 포함하면서 개발 기간도 단축하라고 한다면, 사람을 대폭 늘리 수밖에 없다. 하지만, 여기에는 한 가지 문제가 있다. 사람이 많고 과제의 크기가 커지면 커질수록 실패할 가능성도 비례해서 커진다는 점이다. 기능이 늘어나면 복잡한 일이 되며, 사람이 많아지면 의사 결정과 전달이 힘들어지기 때문이다. 따라서, 잦은 동기화와 설계에 대한 시간을 충분히 가져야 하지만, 그런 일은 좀처럼 생기지 않는다. 동기화는 점점 더 늦어지게 되고, 설계없이 코딩하는 일이 빈번해진다.

모든 소프트웨어 개발과제는 항상 새로운 것이다. 과거에 사용되었던 코드를 이용한다고 해서 같은 소프트웨어가 되는 것은 결코 아니다. 새로 만드는 것이 기존과 동일한 소프트웨어를 만드는 것이라면, 사람을 투입해서 개발할 이유도 없다. 다르기 때문에 새로 개발하는 것이다. 따라서, 개발 기간을 무작정 줄이라고 하는 것은 새로운 가치를 추가하지 말라는 말과 같다고 생각해야 한다. 그런 제품은 개발해봐야 당연히 팔리지 않을 것이다. 물론, 그렇다고 재활용할 것이 전혀 없다는 것은 아니다. 재활용을 통해서 얻을 수 있는 것은 일부 검증된 코드일 뿐, 변경의 영향이 제한적으로만 시스템에 영향을 준다고 보장할 수 있는 방법은 없다.

새로운 가치는 지상에 없던 것을 만들는 것이 아니라, 기존에 이미 존재하던 것을 개선하고 엮어서, 더욱 편리하고 쉽게 원하는 것을 해주는 것이다. 플랫폼이라는 것은 어떤 서비스의 토대가 되는 것을 말하며, 새로운 서비스를 쉽고 빠르게 만들어낼 수 있다면 플랫폼을 사용하지 않을 이유가 없다. 문제는 플랫폼의 사용자가 누구이며, 그들이 무엇을 원하는가를 파악하는 것이다. 원하는 것이 무엇인지 모르는 상태로 플랫폼을 구축하는 것은, 앞으로 사용되지 않을 서비스를 공들여 제공하는 것과 마찬가지다. 플랫폼은 사용되어야 가치가 있으며, 사용되기 위해서는 사용자의 요구를 반드시 만족시킬 수 있어야 한다.

플랫폼이 잘되기 위해서는 다양한 사용자의 요구를 받아들여야 하지만, 너무나 다양한 요구 사항들을 모두 다 받아들일 수는 없다. 냉장고를 만드는데, 다리미나 전자레인지 등의 요구 사항까지 넣어서 만들 수는 없기 때문이다. 적절한 타겟(Target)을 선택하고 빠르게 피드백을 받아서 지속적으로 플랫폼을 개선해 나가는 것이 중요하다. 모든 소프트웨어는 버그를 항상 내포하고 있다. 버그라는 만성 질환은 언젠가 크게 문제가 될 수 있으므로, 꾸준히 개선하는 노력이 필요하다. 새로운 기능을 추가하면서 빠르게 사용자의 요구를 흡수하고 대응해 주는 것, 이것이 바로 플랫폼이 살아남기 위한 핵심이다. 빠르게 요구에 대응해 나간다고 해서 무조건 빨리 고치라는 것이 아니다. 빠르게 움직일 수 있도록 변화에 유연한 구조를 가지라는 말이다.

새로운 변경은 새로운 테스트를 요구한다. 기존에 했던 통과했던 테스트도 반복적으로 모든 변경에 대해서 실행해야 한다. 관리자는 새로 추가된 기능만 검사하기를 원할지 모르지만, 새로운 변경은 기존의 코드와 어떤 식으로든 관계를 맺고 있는 것이 사실이다. 따라서, 아무리 조그마한 변경이라도 전체 테스트 케이스에 대한 실행은 어쩔 수 없이 해야 한다. 개발이 진행될수록 테스트해야 할 아이템은 늘어나게 되며, 테스트로 부터 받는 피드백 간격은 더 길어지게 된다. 따라서, 플랫폼을 갖추기 위해서는 테스트 자동화는 반드시 필요하다. 플랫폼 코드의 대부분이 자동화된 검증을 가지지 않는다면, 플랫폼을 적용한 과제도 테스트 과정이 길어질 수밖에 없다.

과제 초반에는 어떻게 만들 것인지를 고민하지만, 과제 후반은 어떻게 검증할 것인가에 집중하게 된다. 플랫폼 과제는 추가적인 기능의 구현은 줄일 수 있지만, 기존 코드의 변경은 어쩔 수 없다. 대부분의 코드를 재활용할 수 있지만, 그렇다고 변경이 없는 것은 아니다. 새로운 환경, 새로운 기능, 새로운 인터페이스가 구현되어야 하며, 이들 각각은 다시 새로운 테스트를 만든다. 기존의 플랫폼이 검증되었다고 해서 재활용되는 모든 코드를 검증하지 않을수도 없다. 플랫폼 재활용의 대상은 플랫폼에 들어가는 코드만이 대상이 아니며, 관련된 모든 테스트와 개발환경, 하드웨어 및 소프트웨어 툴들도 재활용 해야한다. 특히, 자동화된 테스트는 플랫폼을 이용한 개발 속도 향상에 중요한 영향을 주는 부분이다. 먼 길을 가기 위해서는 필요한 도구들도 착실히 준비해 나가야 할 것이다.

# [ 린(Lean) 소프트웨어 개발 ]

“린(Lean)”이란 단어의 뜻은 "기대다, 의지하다, 야윈" 등이다. 소프트웨어 개발에서 “린”이라는 단어를 사용하는 경우는 사용자의 기본적인 요구사항을 반영한 베타(Beta) 버전을 만들어 배포하고, 사용자 피드백을 통해 지속적으로 개선해 나가는 방법을 말한다. 모든 파악된 요구사항을 한번에 반영한 소프트웨어를 배포하는 것이 아니라, 우선순위화 된 핵심 요구사항만 빠르게 구현해서 반복적으로 여러 번 배포하는 것이다. 각각의 배포는 사용자의 피드백을 빨리 얻을 수 있다는 점에서 꾸준한 개선을 추구한다고 볼 수도 있다.

여기서 중요한 점은 "사용자에게 진정한 도움(혹은 가치)"을 빠르게 제공해 주겠다는 개념이다. 모든 소프트웨어의 진정한 가치는 사용자에게 얼마나 도움을 주는가에 달려있으며, 사용자의 요구에 얼마나 가깝게 만들 수 있는가에 따라 가치가 정해진다. 많은 소프트웨어 프로젝트가 진행 중이지만, 대부분의 경우 사용자의 요구사항을 정확히 파악하는 것은 사실상 불가능하다. 프로젝트의 시작 시점에 완전한 요구사항을 받는 경우는 거의 없다. 프로젝트를 진행하는 전체 기간으로 봤을 때, 사용자 요구사항의 분석에는 거의 시간을 들이지 않거나, 혹은 그 부분에 들어간 노력에 대해서는 지나치게 가볍게 생각하는 것이 일반적이다.

소프트웨어 공학은 사용자의 요구사항을 고치는 것이 향후 구현된 소프트웨어를 고치는 것에 비해서 비용적으로 200배 이상의 절감효과가 있다고 이야기 하지만, 정작 요구사항을 파악하는 일에 투자하는 시간은 인색한 것이 사실이다. 그렇다면 왜 이런 일이 발생하는 것일까? 실제 소프트웨어 개발 과제를 진행하는 초기에는 불확실성이 너무도 많다. 과제 일정은 과제를 시작하기도 훨씬 전에 정해져 버린다. 제품에 탑재되는 소프트웨어를 만드는 회사의 경우, 다음 해에 해야 할 일을 미리 그 전 해에 정하고, 대략적인 일정을 상품기획과 같이 논의하게 된다. 이 때 운이 좋으면 그나마도 일정에 여유를 둘 수 있지만, 대부분의 경우 동의한 일정은 이러저런 보고를 거치고 나면 반 토막이 나버리고 만다.

반 토막난 덕분에 요구사항을 충분히 고려할 시간은 주어지지 않는다. 개발 기간의 대부분은 구현과 테스트 차지하며, 요구사항을 제대로 파악할 여유가 없었기에 설계도 부실할 것이 당연하다. 설계 없는 코딩은 잦은 재작업을 낳지만, 일단은 구현율을 높이는 것이 지상 최고의 목표가 되고 만다. 테스테에 전달해야 할 코드를 만들기 위해서는 코드에 문제가 있더라도 크게 잘못된 선택이라고 생각하지는 않는다. 그리고, 문제가 생기면 그 때 가서 고치면 된다고 생각 한다. 이것이 우리가 소프트웨어 개발자로 일하는 분야의 현실이다.

사용자의 진정한 요구(Needs)를 고려하지 않고 시작된 과제는 철저한 분석이 있을 수 없다. 무엇을 만들지 모르는 상황에서 설계와 구현을 시작해야 한다. 전달받은 요구사항이 있다고 하더라도, 그 요구사항이 얼마나 타당한지에 대한 검토도 부족하다. 예를 들어, RAID(Redundant Array of Inexpensive Disks)라는 것을 구현해야 한다면, RAID를 구현하는데 필요한 기술의 타당성 조사와 사용자들이 RAID를 어떻게 사용할지 거의 고려하지 않는다. 결국, 구현된 내용을 가지고 테스트를 시작하면, 다양한 문제로 인해 반복적인 수정과 테스트의 과정을 거치게 된다. 따라서, 최종 결과물은 개발자의 선택일 뿐이지 사용자를 위한 것이 되지 못한다.

“이런 것을 방지해 줄 수 있는 방법은 없을까?” 이런 질문을 하는 것은 당연하다. 필요한 것은 사용자들의 빠른 피드백을 통한 반복 학습 밖에 없다. 시장은 빠르게 변해가고 사용자의 나날이 높아지고 있는 상황에서 일 년이라는 개발기간은 너무 긴지도 모른다. 오늘 생각한 것들이 내일의 사용자 요구를 만족할 것이라고는 가정할 수 없는 것이다. 따라서, 지속해서 시장(혹은, 작은 실험을 통해서)에서 검증받지 못한 제품은 빠르게 도태되어 갈 수밖에 없고, 새로운 요구사항의 반영도 느려질 수밖에 없다. 따라서, 많은 요구사항을 한꺼번에 제품에 적용하기 보다는 조금씩 자주 소프트웨어를 배포하는 방법을 사용해야 할 것이다. 물론, 이렇게 개발하지 못하는 제품도 있다.

예를 들어, 표준 스펙을 만족시켜야 하고 한 번 배포하고 나면 고치지 못하는 경우도 있다. 하지만, 이 때도 스펙의 강제 규정은 만족시키고 옵션은 뒤로 미루는 방법을 취하거나, 고객의 최종 제품에 포함되기 전에 공동으로 정한 마일스톤마다 기능의 추가와 검증하는 시간을 가질 수도 있다. 물론, 이렇게 되기 위해서는 고객과의 “신뢰”부터 먼저 쌓아야 만할 것이다. 고객에게 가치를 줄 수 있어야 제품으로서 인정받을 수 있으며, 반복을 통해서 완성도를 높여간다면 고객의 신뢰도 함께 높아질 것이다. 한 번에 모든 기능을 제공해서 만족시키기 위해서는, 고객에게 믿음을 줄 수 있는 기회도 적거니와 그 기회를 살리지 못하면 다음 기회를 얻기도 어려워진다.

만들어진 제품은 반드시 사용자에게 도움을 주어야만 한다. 아무 도움이 되지 못하는 제품에 지갑을 열 사용자는 없다. 물론, 시장을 독점하고 있는 경우에는 선택의 여지가 없겠지만, 우리가 만드는 제품은 6개월이면 거의 비슷한 제품이 나오는 게 일반적이다(아니, 그보다 더 짧을 수도 있다). 빠르게 움직이기 위해서는 몸이 가벼워야 하며, 한 번에 모든 기능을 제공하는 선택은 발 걸음만 더디게 할 뿐이다. 사용자는 무한정 기다려주는 사람들이 아니다. 자신의 요구가 만족되는 제품이 있다면 얼마든지 쉽게 지갑을 열 것이다. “린”은 무엇을 만들어야 하는지 빨리 사용자에게 직접 묻는 방법이며, 사용자의 답을 통해 빠르게 개선해 나가는 개발 방법이다. 따라서, 몸이 가벼울 때 사용자의 의견을 듣는 것이 재작업해야 할 비용도 크게 줄여줄 것이다.

# [ 객체지향 프로그래밍에 대해서 ]

절차지향 중심의 프로그래밍을 시작한 사람은 객체지향 프로그래밍에 쉽게 적응하지 못한다. 객체지향 개념을 익히기 전에 이미 알고 있던 절차지향 언어의 문법만 바꾸어서 사용하기 때문이다. 실제로 객체지향 언어를 사용하더라도 여전히 절차지향 개념으로 무장한 사람들이 우리 주변에는 많다. 심지어 객체지향 언어부터 배운 사람도 절차지향적으로 코딩하는 경우도 흔하다. 그 만큼 객체지향과 절차지향이 다른 개념을 가지고 있다는 뜻이며, 절차지향이 사람이 일반적으로 편의를 위해서 사용하는 “나누어진 절차”를 쉽게 표현하기 때문이기도 하다.

객체지향 개념을 이해하지 못한 사람들이 개발과제의 리더 역할을 하는 것도 사실이다. 개념을 익히는 데 실패한 사람은 도구가 가지고 있는 외적인 표현만 이용할 뿐이지, 그 내면에 숨겨진 원리는 이해하지 못한다. 결국 새로운 언어를 사용하더라도 기존의 언어를 사용하는 것과 크게 달라지는 것이 없다고 생각하게 된다. 나아가 그런 사고가 다른 사람의 생각에까지 영향을 주게 되고, 결국 기존의 절차지향적인 방법에서 한 걸음도 나아가지 못한다. 언어가 달라지면 생각도 달라져야 하며, 제대로 언어를 이해하고 사용하기 위해서는 단어만 바꾸는 것이 아니라 표현법도 달라진다는 것을 이해해야 한다.

기본적으로 객체지향은 일을 하는 주체가 누군인가에 중점을 둔다. 절차지향은 일의 주체가 마치 하나인 것처럼 모든 일을 처리하지만, 객체지향에서는 각각의 나누어진 주체가 자신의 역할에 맞는 일을 협력을 통해서 해결해 나간다. 따라서, 객체지향에서는 해야 할 일에 대한 분석을 통해서 일의 주체와 역할을 찾아내는 것이 핵심이 되며, 절차지향에서는 얼마나 효과적으로 일을 하는가가 중요한 관심사가 된다. 객체지향에서는 다양한 주체들이 존재하고, 그러한 주체들이 해야 할 일의 범위가 명확하게 된다.

절차지향에서는 하나의 행동 주체가 있게 되므로 역할의 범위를 나누는 것은 의미가 없다. 나누더라도 역할을 가지는 주체를 두기보다는 일의 종류와 순서를 묶어서 관리하는 것으로 역할의 범위를 정한다. 객체지향에서는 주체가 가진 역할과 책임에 따라 다른 주체들과 관계를 맺을 수 있으며, 주체들 간에 협력을 위해서 주고받는 메시지가 주요한 내용이 된다. 절차지향은 흐름과 그 흐름에 담긴 내용(데이터)에 집중하지만, 객체지향은 주체 역할을 하는 객체들을 생성해서 그들 간의 계약을 유지하고 관계를 정립해 간다.

여기서 중요한 점은 어떤 언어가 더 좋다 나쁘다는 이야기가 아니다. 적절한 상황에 적절한 언어를 사용하면 되는 것이다. C언어건 C++, Java, C#이건 상관없이 사용해야 할 때라고 판단이 되면, 필요한 언어를 사용해서 구현하면 된다. 하지만, 어떤 언어가 가지고 있는 개념이 무엇인지, 왜 그런 개념이 필요한지 정도는 이해할 필요가 있다. 단순히 언어의 문법을 이용해서 코딩 한다고 C가 C++이 되지는 못한다. 아는 것과 가르치는 것이 다르듯, 어떤 언어의 문법을 안다고 해서 구현을 잘 했다고도 할 수는 없다.

사람의 언어도 문화적인 배경 없이 탄생하지 못하듯, 컴퓨터 프로그래밍의 세상에서도 저변에 깔린 개념과 이유를 제대로 이해하지 못한다면, 영원히 그 언어를 모르는 것과 마찬가지일 것이다. 특정 언어를 사용하는 목적은 그 언어를 통해서 문제를 바라보는 관점이 바뀐다는 것을 의미하며, 제대로 코딩하기 위해서는 그 언어가 가지는 문제 해결 방법을 사용해야 한다.

# [ 과제를 시작할 때 가장 먼저 해야 할 일 ]

소프트웨어 개발자는 항상 새로운 일을 하는 것을 즐긴다. 두려움이 없는 것은 아니지만 새로운 도메인을 자신의 경력에 추가하는 것을 마다하지 않는다. 그리고, 시행착오는 있겠지만 근본적으로 품질이 좋은 소프트웨어를 개발하는 방법에는 큰 차이가 없다. 차이가 나는 것은 도메인 지식에 대한 이해 정도일 뿐이다. 그럼, 이렇게 새로운 일을 시작할 때 소프트웨어 개발자로서 가장 먼저 해야 할 일은 뭘까? 한 가지 답은 "개발 체계를 갖추는 것"이다. 개발 체계에도 다양한 것이 있겠지만, 그 중에서도 팀 내의 모든 개발자가 반드시 지켜야 할 것들을 만드는 일이 가장 중요하다.

팀 내의 모든 개발자가 따라야 하는 것에는 어떤 것들이 있을까? 생각해보면 여러 가지가 있겠지만, 새로운 팀으로 새로운 프로젝트를 시작하게 된다면 가장 먼저 공통으로 사용할 문서의 템플릿(Template)을 만드는 것이다. 팀원들에게는 같은 문서 템플릿을 사용하게 됨으로써 팀의 고유한 정체성(Identity)을 확인할 수 있는 계기가 될 것이다. 또한, 누가 문서를 보더라도 어느 팀에서 작성된 문서인지 단번에 알아보게 될 것이다. 소스 코드를 작성할 때 필요한 템플릿을 정의하는 것도 잊지 말아야 한다. 코드의 템플릿에는 지켜야 할 코딩 룰과 같은 것을 미리 예제로 적어두는 것도 좋다. 목적은 누가 보더라도 마치 한 사람이 작성한 것처럼 모든 문서나 코드들이 보여야 한다. 즉, 모든 팀원들이 “일관성”을 지키고 있다는 뜻이 되기 떄문이다.

규율(규칙)에 기초한 팀은 팀원들이 항상 새로운 것과 마주쳤을 때 어떻게 대응할 것인가에 대한 일치된 의견을 내놓을 수 있다. 난관에 부닥친 팀 일수록 그것을 헤쳐나가는 데는 팀워크가 반드시 필요하며, 그런 팀워크를 유지할 수 있는 것들은 과제의 처음부터 동의한 원칙을 기초로 하는 것이 좋다. 예를 들어, 출퇴근 시간, 의견을 제시하는 방법, 회의 시간과 일정, 주간보고 방법, 문제점 등록과 관리 방법 등등 반드시 필요한 것들에 대해서 하나씩 만들어갈 수 있다. 하지만, 모든 것을 원칙으로 묶으려고 하지는 말아야 한다. 꼭 필요한 몇 가지만 정해서 그것만큼은 반드시 지키도록 하는 것이 좋다. 너무 많아서 지키지 못하는 것보다 적은 수라도 일관되게 꾸준히 지켜가는 것이 더 좋은 팀으로 이끌어 줄 것이다.

# [ 소프트웨어 생산성에 대해서 ]

언젠가 아시는 분이 "소프트웨어 개발자의 생산성을 높이려면 무엇이 가장 필요한가?"를 물었었다. 대답은 "개발자들에게 30"급 대형 모니터를 사주세요."였다. 사실 소프트웨어 개발자의 대부분 업무는 남(혹은, 자신)이 작성한 코드를 보고 수정하는 일이다. 아마도 90% 정도는 이미 작성된 코드를 읽는 것이고, 10% 정도가 실제로 새로운 코드를 작성하는 일일 것이다. 따라서, 얼마나 많은 수의 윈도우 창을 한 번에 볼 수 있는가에 따라 개발 속도가 영향을 받게 된다. 한 번의 클릭이라도 줄이고, 한 번의 목 움직임이라도 줄일 수 있다면, 전체적으로 개발의 효율은 크게 향상될 수 있다. 적어도 생각이 끊어질 일은 없을 것이기 때문이다.

컴파일이 오래 걸리면 도중에 밖에 잠시 나와서 차를 마시며 이야기를 하는 것도 일종의 낭만이 될 수도 있기는 하겠지만, 어쩔 수 없이 그렇게 밖에 할 수 없는 상황이라면 이야기가 달라진다. 제대로 된 개발 서버를 구축하지 못하고 있다는 것이 사람들에게 짜증을 유발하기 때문이다. 정말 필요한 것에는 코딩을 매우 잘하는 비싼 엔지니어도 있겠지만, 그런 사람은 구하기가 아주 힘들다. 금방 찾을 수도 없고 설령 찾았다고 해서 그런 사람이 쉽게 다른 회사로 움직여 줄지도 확실하지 않다. 차라리 그냥 좋은 개발 서버를 여러 대 사주는 것이 가장 확실한 생산성 향상을 위한 첫걸음이 될 수 있을 것이다. 사람은 성장시켜야하지만 기계는 들어오는 즉시 재대로 된 성능이 발휘할 수 있기 때문이다. 개발에 필요한 도구들도 마찬가지다. 아무리 비싼 도구라고 할지라도 사람보다 비싼 것은 없다.

하지만, 현실은 생각보다 어렵다. 대부분의 경우 새로운 장비를 구매하는 것은 여러 가지 절차를 따라야 하고, 예산이 미리 잡혀 있지 않을 때에는 여러 번의 보고와 함께 재수 없으면 욕도 먹어야 한다. 웃기는 일이지만 일을 잘 하려고 노력하는데도 욕을 먹는 상황이 자주 발생한다. 미리 생각하지 않아서 예산을 잡아두지 못했다면, 더 큰 낭패를 경험할 수도 있다. 하지만, 쓸데없이 장비를 사서 그냥 놀리는 개발자가 있는지 주위를 돌아본다면, 아마도 백 명 중에 한 두 명은 그럴지도 모른다. 하지만, 나머지 98명은 그렇지 않을 것이다. 한 두 명의 사람을 찾기 위해서 나머지 98명에게 불편을 준다는 것이 과연 효과적일까? 아마도 절대 그렇지 않을 것이다. 장비 가격보다 훨씬 유지비가 비싼 사람들을 그런식으로 괴롭혀봐야 높은 생산성은 멀어질 뿐이다.

대부분 관리자는 외적인 결과물에 대해서만 보고 싶어한다. 버그가 몇 개 발생했고, 몇 개의 버그가 해결되었으며, 몇 개가 장기로 남아있는지 등을 알고 싶어한다. 하지만, 정작 중요한 것은 개발자들이 최선을 다해서 일할 수 있는 환경을 만들어 주는데는 상대적으로 신경을 덜 쓴다. 중요한 과제라는 명분으로 몇 명 들어가지도 못하는 방에 많은 사람들을 몰아넣고, 장비에서 발생하는 열기에 여름을 찜통으로 보내도록 만든다. 이른바 VIP(Value Innovation Center)라고 이름은 붙여 놓았지만, 사람의 가치는 크게 개의치 않는 태도를 보인다. 물론, 그렇다고 개발자들이 개인 룸이 필요하다고 요구하는 것도 아니다. 하지만, 소프트웨어가 고도의 정신 활동이 필요한 일이라면, 집중할 수 있는 환경 정도는 만들어 줄 수 있어야 하지 않을까? 툭하면 이런 저런 사람들이 와서 “순시”를 한다면, 그것 자체도 오버헤드일 뿐이다.

최고의 성과는 최고의 사람이 만드는 것이 아니라 최고의 환경에서 나온다. 물론, 이것은 개인적인 생각이지만, 적어도 성과를 만들기 위해서는 일에 집중할 수 있는 환경 정도는 만들어 주어야 할 것이다. 전화 벨이 수시로 울리고 이런 저런 사람들의 이야기가 방안을 울리도록 들리는 환경이라면, 제대로 일 할 수 있는 시간은 “밤의 정적”과 함께 찾아올 것이다. 한 마디로 칸막이도 없고 여유도 없는 공간에서 고도의 정신 활동이 필요한 사람들이 최고의 성과를 내기를 바라는 것은 웃기는 말일 뿐이다. 게다가 툭하면 보고하러 내려오라는 전화가 온다면, 왜 그런 곳에 사람을 넣어두고 일을 시키는지 궁금해지기 시작할 것이다. 소프트웨어 개발자는 닭장 속에서 알을 전문적으로 낳는 양계 닭이 아닌 사람이다. 사람은 사람답게 대접받아야 좋은 성과를 낸다.

언제인지도 모를 날을 기약하면서 끝없이 일하는 것은 불가능하다. 길어야 일 년 정도? 아마 그 이상 좋지 않은 환경에 방치되면, 더 이상의 과제에 대한 흥미, 회사에 대한 충성심, 사람에 대한 신뢰 같은 것을 기대해서는 안 된다. 관리자는 자신이 관리하는 사람들에게 일 할 수 있는 최적의 환경을 제공하기 위해서 최선을 다해야 하고, 그것이 어렵다면 상황에 대해서 공감할 수 있도록 대화해야 한다. 그냥 일방적으로 주어지는 명령은 개발자의 머릿속에 항상 "Google 같은 회사에서 일해보고 싶다"라는 생각만 떠올릴 뿐이다. 물론, 우리들 모두가 “Google”처럼 일 할 수는 없다. 하지만, 구글보다 더 상호 신뢰하는 사람들로 팀을 만들 수는 있다. 문제는 신뢰하기 위해서 신뢰할 수 있는 환경부터 먼저 만들어 주어야 한다는 점이다.

# [ 우리는 시스템으로 일한다? ]

시스템을 우리말로 풀이하면 "체계"정도로 해석된다. 운영 시스템 혹은 운영 체계라고 표현되는 "Operating System"은 컴퓨터를 운영하는 일을 도맡아서 대신해 주는 소프트웨어다. 응용 프로그램 측에서 보면 이러한 운영 체계는, 내가 해야 할 일을 대신 처리해 주어서 편의성을 제공해 준다고 볼 수 있다. 즉, 도움을 얻어서 효과적으로 일을 수행할 수 있게 해주는 것이 시스템이다. 세세한 부분까지는 모르더라도 최소한 그러한 서비스를 어떻게 사용할 수 있는지 정도만 알아도 일을 하는 데는 아무런 지장을 받지 않는다.

우리가 일하는 환경은 어떤가? 우린 정말 시스템으로 일하고 있을까? 예를 들어, 어떤 회사에 과제를 관리하는 시스템을 도입했다고 하자. 그 시스템에서 할 수 있는 것은 과제의 시작에서 끝까지 모든 것을 기록할 수 있는 소프트웨어가 설치되어 있을 것이다. 과제를 진행하는 도중에 마주치는 다양한 위험(Risk)과 문제점(Issue), 그리고 버그나 해야 할 일 등 다양한 것들이 입력 된다. 필요하다면 일일보고나 주간보고도 입력해야 할 수도 있다. 그럼 왜 이런 것들이 필요할까? 사실 이것은 일의 기록를 남기고 개발자 간에 경험을 공유하기 위해서 존재한다고 봐야 한다. 기록이란 일의 책임을 명확히 해서 누구 언제 무슨 일을 했으며, 어떤 문제로 고생 했는지등의 다양한 것들을 함께 나누고 고민하기 위해서 필요하다.

하지만, 현실에서는 그런 시스템을 도입은 했지만 정작 제대로 사용하지 못하고 기존의 틀에서 벗어나지 못한 경우를 많이 볼 수 있다. "Dash Board"에서 확인할 수 있는 것들을 일일이 작성된 문서와 함께 구두로 보고를 해야 하는 일이 반복된다. 이미 시스템에 입력으로 들어가 있지만, 제대로 된 데이터가 있는지 검증할 수도 없으며, 그런 시스템을 높으신 분들은 사용하지 못하기 때문이다. 입력하는 사람도 자신의 업무 성과에 대한 평가자료로 활용될 수 있기에, 선택적으로 입력하고 싶은 것만 시스템에 넣어둔다. 이런 시스템이라면 형식적으로 운영될 것이 분명하다. 오히려 중복된 일이 되고 만다. 심지어 버그 같은 민감한 정보는 외부에서 보지 못하도록 시스템에 등록된 것과 내부에서 관리하는 것이 다를 수도 있다.

현실이 이렇다면 우리는 정말 시스템으로 일하고 있는 것일까? 아니면, 시스템과 우리는 각자 다른 길로 가고 있는 것은 아닐까? 도움을 받는 것이 아니라, 길들기를 바라는 사람들에게 위안을 주기 위해서 그렇게 일하는 척하고 있는 것은 아닐까? 그렇다고 한다면, 시스템을 버리고 처음부터 "왜?"라는 질문을 다시 모든 이해관계자가 고민하도록 해야 한다. 우리는 시스템의 노예로 살 수 없으며, 누군가에게 보여주기 위해서만 일하는 것이 아니기 때문이다. 시스템은 사람을 돕기 위해서 존재하는 것이며, 가장 큰 도움은 일을 늘리는 것이 아니라 일을 줄여주는 것이다. 개발자를 믿지 못하게 만드는 시스템이라면 차라리 없는 편이 오히려 더 도움이 될 것이다.

관리자들은 소프트웨어는 눈에 보이는 것이 없어서 어렵다고들 이야기한다. 하지만, 눈에 소프트웨어가 개발되는 과정을 보여주어도 보지못하기는 마찬가지다. 마치 가리키는 달은 보지 못하고 손가락 끝만 바라볼 뿐이다. 왜 이런 일이 생길까? 윗 사람이 소프트웨어 개발을 몰라서 그렇다고 이야기하면 조금이라도 위안이 얻을 수 있을까? 하지만 정작 우리의 표현력이 문제일지도 모른다. 원하는 것을 쉽게 이해시킬 수 있는 능력이 부족한 것일 수도 있다. 관리자들이 보지 못한다고 그것으로 끝나는 것이 아니다. 어떤 식으로라도 그들의 이해를 얻지 못한다면, 더 힘들어 질 수밖에 없다.

소프트웨어의 품질 지표를 보여줄 수 있는 다양한 도구(Tool)는 이미 있다. 복잡도, 코드의 중복도, 역참조와 같은 계층구조 위반, 코딩 룰 위반, 단위 테스트의 코드 커버리지(Coverage) 등등 지표로 제시할 수 있는 것들은 이미 충분히 많다. 하지만, 중요한 것은 그런 것들이 무엇을 의미하는지 정확히 설명하고, 그러한 것들이 왜 중요한지 이해시키기가 힘들다는 점이다. "그래서, 어떻게 하라고?"라는 질문이 윗사람에게서 나오는 순간, 머리 속이 하얗게 변하면서 보고하는 사람과 보고를 듣는 사람 사이에 “넘을 수 없는 보이지 않는 벽”이 있다는 것을 자각하게 된다. 서로 대화는 나누고 있을지는 몰라도, 서로 생각하는 것이 다를 뿐이다. 위 사람이 원하는 것은 단지 한 가지뿐이다. “그래서, 언제 끝난다는 말이지?”. 물론, 대답은 “아무로 모른다”는 것이다.

사실 위 사람이 원하는 것은 만들고 있는 제품의 현재 진행 상황이 정확히 어떻게 되며, 얼마나 시간이 지나면 사용할 수 있을지를 알고 싶은 것이다. 따라서, 각종 수치로 화려하게 장식된 발표 자료를 보는 것은, 듣는 사람의 입장에서는 자신과 관계없는 다른 세상의 이야기를 졸음을 참고 들어야 하는 고통일 수도 있다. 물론 처음의 몇 번이나 과제를 마무리할 때 작성된 코드의 평가척도 정도로 잠시 볼 수는 있겠으나, 좋은 말도 한 두 번 들으면 질린다. 정말 반드시 해야 할 것은 구현된 기능들을 직접 보여주는 것이다. 즉, 현재 구현된 기능들은 어느 정도 수준이며, 아직 구현이 안된 것에는 어떤 기능들이 있는지 눈으로 확인시켜주는 것이다. 그리고, 과제를 더 잘하기 위해서는 어떤 도움이 필요한지 알려주는 것이다.

내부 이해 관계자들은 과제가 잘 되기를 바라는 사람들이며, 우리가 만족을 시켜주어야 할 또 하나의 고객이다. 따라서, 그 사람들이 정말 원하는 것을 파악하는 것은 과제를 성공적으로 완수하는데 반드시 필요하다. 개발할 때 도구로 사용하는 시스템은 개발자를 위해서 만들어진 것이지, 그것을 가지고는 모든 이해 관계자들을 즐겁게 해줄 수는 없다. 과제의 진척 정도를 경험이 있는 사람들이 봤을 때는 이해할 수 있겠지만, 한 번도 소프트웨어를 개발한 적이 없는 사람에게는 뜬 구름잡는 소리일 뿐이다. 현실에서 소프트웨어를 제대로 경험한 위 사람을 만나는 것은 쉬운 일이 아니며(특히, 하드웨어와 소프트웨어를 같이 개발해는 회사라면), 그런 사람들을 바꾸거나 무시할 수도 없다. 따라서, 최대한 그분들이 알고 싶은 것을 잘 파악해서, 그것에 맞게 내용을 요약해서 보여주어야 한다. 예를 들어, 기술적인 빚을 시간으로 변환해서 보여줄 필요도 있다. 그것을 통해서 과제 완료 시점에 대한 추정을 할 수 있다. 특히 대부분의 위 사람들은 돈과 시간에 민감하게 반응하기 때문이다.

개발 시스템은 문화에서 비롯된다. “개발자 문화”가 정착되고 그것을 좀 더 효과적으로 만들기 위해서 다양한 시스템이 존재하게 된다. 체계적으로 일하는 절차도 그것이 효과적으로 품질이 좋은 제품을 만드는 데 필요하기 때문에 생겨난 것이다. 이미 개발되어 많이 사용되고 있는 것들이 있다면(심지어 그것을 전문적으로 파는 회사가 있다는 말은), 그것이 개발자에게 필요하고 사용했을 때 효과를 볼 수 있었다는 의미다. 따라서, 이미 어느 정도 개발자들 사이에 공감대가 있다는 뜻이 된다.

하지만, 새로운 개발 시스템을 도입하는 것은 쉬운 일이 아니다. 즉, 지금까지 익숙했던 것들에 대한 변화를 요구하기 때문이다. 자신의 발에 익숙해진 신발을 버리고 새로운 신발을 신으라고 하면, 처음에는 발 뒤축이 까지기도 하고 걷는 데 불편함을 겪기도 한다. 하지만, 이내 조금만 시간이 흘러가면 이전에 신던 신발처럼 익숙해지는 것도 사실이다. 잠깐의 어색함을 벗어나면 더 효과적으로(혹은, 효율적으로) 일을 할 수 있게 된다. 물론, 억지로 발보다 작은 신발을 신는 것은 답이 아닐 것이다. 그 때는 오히려 신발을 바꿔야 할 필요도 있을 것이다. 다른 회사에 어울린다고 해서 자신의 회사에도 유용하리라고 생각하는 것은 잘못된 결과를 낳을 수 있다.

소프트웨어 개발에서는 제대로 된 툴을 선택하는 것이 개발의 효율에 극적인 영향을 줄 수도 있다. 물론, “은빛 탄환은 없다(No Silver Bullet)”에서 그런 툴이 존재하지 않는다고 이야기했지만, 실제로는 어느 정도 효과가 있는 경우도 있다. 특히, 제대로 된 도구를 사용하지 못하던 조직이라면 도입으로 인한 효과는 더 커질 수 있다. 예를 들어, 버전 관리 시스템을 사용하지 않던 조직이라면, 도구의 도입으로 얻는 효과가 기존에 이미 사용하고 있던 조직보다 상대적으로 더 클 것이다. 버그를 관리하는 방법이 문서를 이메일이나 파일 서버에 저장하는 방식으로 유지했던 조직은, 시스템으로 버그를 관리하기 시작하면 업무에서 발생하는 큰 오버헤드를 제거할 수 있다.

컴퓨터가 잘 하는 것은 단순 반복이며 사람이 잘 하는 일은 복잡한 문제를 푸는 것이다. 따라서, 모든 단순 반복적으로 해야 할 일은 자동화하는 것이 좋다. 그게 안 되면 최소한 입력 회수라도 가능한 줄이려고 노력해야 한다. 이러한 자그마한 효율 향상이 꾸준히 모이고 쌓이게 되면, 큰 혜택으로 돌아올 것은 분명하다. 반복은 사람을 지치게 만들며 실수를 유도한다. 따라서, 자동화를 통해서 수작업을 대체할 수 있다면, 실수도 줄어들고 속도도 빨라질 수밖에 없다. 물론, 사람이 반드시 해야하거나 자동화하기 위해서 비용이 너무 큰 경우도 있다. 이 때는 가능한 비용을 줄이되 모든 것을 자동화하려는 노력은 포기해야 한다.

사실 개발에 사용하고 있는 시스템이 무엇인가가 중요한 것이 아니라, 개발자들이 시스템을 바라보는 태도가 더 중요하다. 도구는 일을 효율적으로 하는데 도움을 줄 수 있지만, 그것도 사람이 도두를 적극적으로 활용할 때 가능한 일이다. 따라서, 시스템을 도입하되 그것을 사용하는 사람을 가장 먼저 고려해야 한다. 예를 들어, 편리하지 않은 UI(User Interface)나 복잡한 사용법은 시스템 사용을 어렵게 만든다. 또한, 실제 일하는 방식과 시스템에서 구성한 작업 흐름이 달라도 문제가 될 수 있다. 시스템으로 일하기 위해서는 사용하는 시스템뿐만이 아니라, 사용자의 자세도 중요하다. 자신이 지금 하고 있는 업무 처리 방법만 고수한다면, 어떤 시스템을 도입하더라도 성과가 크게 달라질 것은 없다.

변화를 받아들이는 열린 문화를 가지고 있는 조직이라면, 새로운 시스템을 도입하는데도 큰 어려움을 겪지 않을 것이다. 대부분 크고 관료화되어 있으며 팀 간 벽이 높은 경우에는 새로운 변화를 잘 받아드리려고 하지 않는다. 지금하고 있는 방법이 조금 불편하더라도 바꾸기 보다는 유지하기를 바란다. 이것은 어쩔 수 없는 사람의 본성이다. 이미 익숙하져 버린 것을 버릴 이유는 없다고 보기 때문이다. 그리고, 그것이 불편하더라도 자신과 관련된 것이 아니라면, 개선할 이유도 없다고 생각한다. 유연하고 적극적인 생각을 가진 조직이라면 전체 최적화에 관심이 많으며, 비록 자신이 직접적으로 도움을 받지 않더라도 필요한 일은 추진한다. 바로 그것이 조직 “문화의 힘”이며, 큰 성과를 내는 조직들이 가지고 있는 기본기다. 탄탄한 기본기는 조금 천천히 갈지도 모르지만 꾸준히 갈 수 있고 잘못가지 않을 길을 선택하도록 도와주기 때문이다.

# [ 과제 관리 시스템의 사용 ]

관리자는 과제 관리시스템을 원하지만 정작 사용할 줄 모르는 경우가 많다. 하드웨어 전공자가 소프트웨어와 하드웨어가 연계된 과제를 맡게 되면, 자신이 잘 모르는 소트트웨어 관련 부분의 일은 담당자에게 거의 전권을 맡긴다. 제대로 된 소프트웨어 개발 담당자를 가지고 있다면 모르겠지만, 그렇지 못할 경우에는 맡겨진 권한이 얼마나 큰지도 모르고 친분 관계가 있는 특정인을 파트(Part)의 리더로 선택하는 경우가 있다. 즉, 그 사람의 개발 경력을 가지고 그 사람이 잘 할 수 있을 것이라는 막연한 기대를 하게 된다.

하지만, 개발 경험이 많다고해서 역량있는 개발자가 되는 것은 아니다. 좋은 소프트웨어를 만들기 위해서는 “좋은 경험”이 필요하지만, 그런 경험을 가진 개발자는 정말 찾기 어렵다. 따라서, 차선이라고 한다면 무엇이 올바른 소프트웨어 개발 방법인지 이해라도 하고 있는 사람을 찾아야 할 것이다. 따라서, 문제는 제대로 된 소프트웨어 개발자가 어떤 사람인지 알아야 한다. 가장 쉬운 방법은 소프트웨어를 개발하는데 필요한 툴을 무엇을 사용하고 있으며, 왜 사용하는지를 질문하는 것이다. 아마도 제대로 된 개발자라면 통합이나 협업에 많은 신경을 쓸 것이고, 과제의 진척 정도를 객관적인 시각에서 관리자가 바라볼 수 있도록 제공해 줄 것이다.

다양한 도구와 단어로 관리자의 눈을 흐리게 하는 “현란한” 개발자도 있겠지만, 요점은 과제의 문제점(Issue)과 위험(Risk) 등의 정보를 객관적으로 어떻게 보여줄지 아는 사람을 골라야 한다. 모든 과제는 나름의 문제와 위험 요소를 가지고 있으며, 현재 만들고 있는 소스 코드도 있을 것이다. 따라서, 이러한 것들을 분석한 결과를 시각적으로 충분히 제시할 수 있는 역량이 있다면 일단은 합격할 가능성이 높다.

여기에 한 가지 더 추가하고 싶은 것은 뽑으려는 사람이 관리하고 있는 팀 내부 "소통"이 어떤가를 보는 것이다. 제아무리 시스템이 잘 되어 있고, 각종 분석 수치가 좋다고 하더라도, 그것들은 실체의 그림자와 같은 수치일 뿐이다. 정확한 과제의 진행 상황이나 팀의 발전 가능성을 알기 위해서는 팀 내부에서 제대로 된 협업을 하고 있는지 확인해야 한다. 팀의 내부뿐만이 아니라 외부와도 적극적으로 의견들 주고받고 있다면, 좋은 역량을 가지고 있을 가능성이 높다. 자신이 가진 권한을 지나치게 확대 해석하거나 독단적인 판단이 많다면 팀을 망칠 가능성이 높다. 팀은 자체적으로도 탄탄해야 하지만, 외부와 교류가 없는 팀은 말 그대로 우물 안 개구리가 될 뿐이다.

과제 관리 시스템을 제대로 사용하지 못하는 과제 관리자가 되고 싶지는 않을 것이다. 자신은 사용하지도 않으면서 개발자들에게 사용하라고 말하는 것 자체도 어색하고 부끄러운 일이다. 자신이 직접 하나하나 메뉴를 눌러보고 어떤 것들이 있는지 알고 있어야 한다. 그리고, 지속해서 자신이 관리하는 개발자들에 대해서도 과제 관리 시스템을 사용하도록 권해야 한다. 두 번 반복적으로 일을 하지 않기 위해서는 보고 방법도 일원화하는 것이 좋다. 이 때도 시스템을 적극 활용하는 것이 좋을 것이다. 주간보고, 일일보고, 월간보고 등을 시스템에서 입력하고, 그것으로 과제의 진척도와 문제점을 파악하고 관리하는 것이 구두 보고나 전체 회의로 대부분의 업무 시간을 쓰는 것보다 훨씬 효과적일 것이다.

잘 안되는 회사의 여러가지 특징 중의 하나가 "지나치게 많은 회의"와 “보고서 작성”이다. 잘 안될수록 "실행에 집중"하는 것이 아니라, “실행 계획과 보고에 집중”한다. 일은 실행이 중요하지 계획과 보고만 한다고 진행되는 것이 아니다. 실제적으로 일 할 수 있는 시간을 가능한 많이 확보해야 한다. 물론, 계획이 중요하지 않다는 것은 아니다. 하지만, 계획했으면 실행할 수 있는 시간도 필요하다. 문서도 중요하지만 불필요한 숙제로 얼룩진 보고서를 작성하는 것이 정말 의미가 있을까? 교육도 중요하지만 실무와는 동떨어진 정신 교육이나 성과만 강조하는 연설 같은 교육이 정말 모든 구성원이 다 들어야 할 필요가 있을까?

하지만, 여전히 많은 임원들은 과제 관리 시스템을 바라보지 않고 보고서만 보고 산다. 날아오는 이메일만 보는데도 하루 시간의 대부분을 소모하고 있다. 각종 대면 보고와 형식적인 회의만 하다가 정말 중요한 것들은 놓치고 있는지도 모른다. “소를 키워야 할 사람”들은 보고서를 쓴다고 시간을 보내고 있고, PC의 하드 디스크는 PPT와 DOC만 잔뜩 채워간다. 이게 정말 다 필요한 일일까? 물론, 문서는 중요하다. 기록이 있어야 근거가 생긴다. 하지만, 모든 근거가 정당한 것도 아니다. 정말 필요한 것은 말 그대로 실행에 집중하는 것이며, 보고서 보다는 실체에 가까운 것을 바라볼 수 있는 안목이 있어야 한다. 이미 좋은 시스템을 갖추고 있다면 그것을 이용하지 않을 이유는 없다. 대화가 필요한 것은 시스템으로도 해결하지 못하는 문제를 직접 만나서 해결하기 위한 것이다.

# [ 테스트 자동화 시스템을 구축하자 ]

하루에 몇 번 정도 남들이 작성한 코드와 자신의 코드를 통합해서 컴파일하고 있을까? 아마도 대부분은 한 번 정도, 그리고 본인의 코드에 대한 컴파일은 수시로 하고 있을 것이다. 여러 명이 협업할 때 가장 힘든 과정 중 하나가 다른 사람의 코드와 통합하는 일이다. 자신이 코딩할 동안 남도 같이 코딩하고, 각각의 코드에서 발생하는 미세한 차이로 인해서 컴파일이 안되는 경우는 흔히 발생한다. 빌드가 되었다고 해도 기존에 잘 동작하던 기능들이 안 되는 경우가 빈번하게 발생한다. 도데체 무엇이 문제일까? 좀 더 신속하고 정확하게 알 방법은 없을까? 예전에 되던 것이 이번에는 안 된다면 어떻게 그것을 빨리 찾아내서 고칠 수 있을까? 사실 이런 것들이 사소하다고 생각할지도 모르지만, 협업이 가지는 가장 풀기 어려운 문제다.

소프트웨어 개발자가 야근하는 이유의 절반 정도는 자신에게 있다고 해도 과언이 아니다. 이유는 다음과 같다. "소프트웨어 개발은 코딩 만이 전부가 아니다. 구현은 개발자 수준의 테스트도 포함한다." 개발자가 코딩을 완료했다고 해서 일이 끝나는 것이 아니며, 자신의 코드를 스스로 검증하지 못한다면 야근은 예약된 것이나 마찬가지다. 코드가 증가를 하게 되면 코드 간의 상호작용이 늘어나게 되며, 테스트를 해야 할 조합(Combination)도 급속도로 늘어날 것이기 때문이다. 따라서, 해야 할 일도 코드의 양에 비례해서 지수적으로 증가하게 되는 것이다. 개발의 후반으로 갈 수록 야근과 주말 근무가 많아지는 것도, 결국은 개발자 수준에서의 테스트가 누락되어 나타나는 현상일 뿐이다.

야근하지 않기 위해서는 코딩보다는 테스트에 더 많이 집중해야 한다. 테스터가 해야 할 부분들을 얼마나 줄여줄 수 있는가(과제의 후반부에 발생할 버그의 수를 얼마나 줄일 수 있는가?)를 고민해야 한다. 이전에 개발한 제품에 기능을 추가하고, 사용자 인터페이스를 조금 바꾼 것밖에 없어 보이는데도 과제의 지연은 발생할 수 있다. 즉, 지난번에는 테스트되지 않은 코드 부분이 이번 변경으로 인해서 새롭게 테스트될 것이고, 그로 인해서 버그가 나올 수 있는 환경이 충족되는 것이다. 또한, 테스트 자동화되지 않았을 경우 이전에 했던 테스트 케이스까지 포함해서 추가된 기능의 테스트 케이스도 같이 검증해야 하기 때문이다.

예전에 했던 테스트 기간이 한 달이었다면, 이번에 하는 테스트 기간은 새로운 기능까지 포함되었기에 한 달 이상이 걸릴 것은 자명하다. 테스터를 늘린다면 지원하는 제품의 수와 신규 개발되는 제품까지 포함해서 점점 더 많은 사람이 필요하게 된다. 물론, 이미 검증된 부분이 있다는 가정하에 줄어들 것이라고 예상하지만 실제는 다르다. 소프트웨어는 완전히 분리된 모듈로 존재할 수 없으며, 한 모듈의 변경이 있을 경우 전체 모듈이 어떤 영향을 받을지 확신하지 못한다. 따라서, 변경은 전체 소프트웨어에 대한 테스트를 반복적으로 거칠 수밖에 없도록 만든다.

결국 개발자의 빌드와 테스트는 같이 진행되어야 하며, 이런 것들이 자동화되어 개발자의 코드가 갱신될 때마다 수행될 수 있어야 한다. 그래야만 과제 후반부에 발생할 수 있는 버그의 수를 최대한 줄여줄 수 있을 것이다. 물론, 과제 초기에는 구현과 테스트 자동화의 병행 때문에 다소간 일정이 지연되는 것처럼 보일 수 있다. 하지만, 결국에는 테스트 단계의 시간을 줄여서 전체 과제의 일정을 따라잡을 수 있게 된다. 또한, 이렇게 자동화된 테스트는 다음 과제를 수행할 때도 도움을 줄 수 있다. 차기 과제가 지금 개발하고 있는 코드에 기반을 둔다면, 새로운 기능에 대한 테스트 케이스의 추가와 자동화만 필요하다. 따라서, 재활용하는 부분이 커질수록 테스트도 함께 재활용될 가능성이 높으며, 과제의 일정에도 지속해서 긍정적인 영향을 주게 된다. 따라서, 미래를 위해서 지금 시간을 투자하는 것은 전혀 아까운 노력이 아니며, 충분한 보상이 개발자 자신에게 돌아온다는 것을 알아야 한다.

코딩은 테스트와 분리된 과정이 아니며, 문제가 발생하면 즉시 검증할 수 있는 도구가 필요하다. 따라서, 코딩과 테스트를 얼마나 가깝게 배치할 수 있는가가 전체 과제 개발 일정에 중요한 영향을 준다. 문제를 쌓아두면 복잡한 문제가 되지만, 즉시 해결하면 간단한 방법으로도 해결 가능하다. 코딩과 테스트를 결합하지 않으면 자동화된 테스트를 구축하기 어렵다. 코드는 점점 복잡하게 변해가는 특성이 있으며, 복잡한 코드를 분리해서 테스트하는 것은 결코 쉬운 일이 아니다. 하루에 100라인의 코드를 자동으로 테스트할 수 있다면, 한 달이면 거의 3,000라인을 자동으로 검증할 수 있다. 6개월이면 18,000 라인이 자동으로 검증될 수 있으며, 대부분의 경우 한 과제에서 개인 개발자가 담당하는 코드는 20,000라인이 되지 않는다. 자동 검증이 되는 20,000라인의 코드를 가지는 것이 그렇지 않은 코드를 가지는 것보다 당연히 유리할 수밖에 없다. 조금 느린듯이 보이지만 결국 더 빨리 가는 길을 선택하게 되는 것이다.

# [ 요구사항과 과제 일정에 대해서 ]

과제를 수행하는 일정이나, 자원 등에 가장 영향을 주는 것이 요구사항(Requirement)이다. 구현을 원하는 요구사항이 무엇인지에 따라 기간이 얼마나 걸리고, 어떤 인력을 얼마나 투입해야 하는지 등등이 모두 정해지게 된다. 하지만, 요구사항에 대한 파악 없이 일정부터 잡고 보는 게 현실이다. 잘 아는 분야의 기술이고 여러 번 구현했다면 큰 문제가 되지 않겠지만, 신규 기술이 들어가는 제품에 대해서 로드맵(Roadmap)을 정하고 일정부터 뽑아 달라고 하는 요구가 끊임없이 개발로 넘어온다. 필요한 인력에 대한 수요를 조사해서 언제쯤 보강될 것이라는 계획도 함께 제출해 보지만, 제 때에 원하는 사람을 뽑는 것은 거의 불가능에 가깝다. 물론, 큰 회사라면 상관없겠지만(사실 큰 회사도 좋은 인력을 뽑는 것은 쉽지 않다.), 중소 기업이라면 인력 부족이라는 어쩔 수 없는 상황과 항상 싸워가면서 경영진의 일정 단축 요구에 응해야 한다.

대략 일 년의 기간이 필요한 개발이라고 하면, 바로 위에 있는 관리자는 2개월을 단축하라고 요구한다. 상품기획은 거기서 다시 2개월 단추을 요구하고, 과제를 최종 결정하는 사장님까지 가면 6개월 정도의 단축된 일정이 개발에 통보된다. (말 그대로 통보다.) 기간이 줄어들고 개발 역량은 그대로라면 어쩔 수 없이 해야 할 일을 줄일 수밖에 없다. 하지만, 이것도 쉽지 않다. 한 번 요구된 기능이나 성능은 추가되거나 변경되는 경우는 많지만 없어지는 경우는 드물기 때문이다. 과제 책임자는 어쩔 수 없이 팀원들을 설득해야 하고, 설득되었는지도 모를 팀원들은 어쨌든 과제를 기한에 맞추어서 끝내려고 야간과 주말 출근도 마다하지 않는다. 시간은 점차 흘러가고 어느 순간 개발 생산성이 높아지지 않는 상황이 오면(오히려 낮아지는 순간이 오면), 일정 지연은 개발자의 책임으로 돌아온다.

일정 지연의 발생은 다양한 형태로 팀을 궁지로 몰아넣는다. 가장 먼저 오는 파도는 바로 윗 사람이 원인이 뭐냐고 묻는 것에서 시작한다. 사실 원인은 이미 다 아는 것이 아닐까? 자신이 제시한 일정이 현실과는 동떨어진 것이라는 것은 자명하기 때문이다. 개발팀과 유지보수팀이 분리되지 않은 상황에서는 더 극적인(?) 일이 많다. 갑작스럽게 고객으로부터 클레임(Claim)이나 문제가 생겼다는 것을 전달받고, 사태가 심각할 경우에는 다양한 사람들이 개발팀에 계속 압박을 넣는다. 현재 진행 중인 과제는 인터럽트(Interrupt)를 받고, 급한 불을 끄는데 인력을 투입할 수밖에 없다. 과제는 당연히 추가적인 지연이 발생하지만, 그렇다고 일정이 변경되는 것은 결코 아니다.

사실 이쯤 되면 과제를 담당하는 책임자는 정신적으로 스트레스를 심하게 받는 상태가 될 수밖에 없다. 운 좋게 문제들이 쉽게 풀리기를 기대하는 것은 말 그대로 우연이다. 우연에 기대서 모든 것을 정할 수는 없다. 과제의 후반부가 되면, 이미 개발 기간의 상당 부분을 과제에 관련없는 일로 채웠다는 것을 알게된다. 출시 일정은 하루 하루 다가오고, 정신적인 압박은 심하면 심했지 절대 줄어들지 않는다. 이런 상황을 완화시킬 수 있는 유일한 길은 추가적인 인력을 투입하는 것이라고 생각할지도 모르지만, 추가 인력은 추가적인 지연만 만들 뿐이다. 테스트를 더 많이 하면 더 빨리 개발할 수 있다는 생각도 어리석은 판단이다. 테스트는 버그를 발견하기 위해서 프로그램을 실행하지만, 코드에 내제한 버그를 제거하지는 못한다. 더 많은 테스트 케이스를 만드는 것도 무리다. 과제 후반의 코드는 추가되는 테스트로 검증할 수 있는 범위가 크게 증가하지 않기 때문이다.

더는 일정을 맞추는 것이 힘들다고 판단되면 가장 먼저 무엇을 해야 할까? 일을 하지 않는 방법이 최선이다. 즉, 요구된 기능과 성능을 포기하는 것이 선택이 아닌 필수가 되어버린다. 어떤 기능이 더 우선되는 기능인지, 그리고 어떤 비기능적인 것이 더 중요한 것인지를 판단해야 한다. 하지만, 우선 순위로 정리된 요구사항의 리스트를 만드는 것도 쉬운 일이 아니다. 다양한 사람들이 자신이 원하는 기능이 제일 우선순위가 높다고 주장할 것이고, 그런 것들을 모아서 정리하고 다시 이야기하는 의미없는 회의의 연속이 될 것이다. 결론은 잘 나지 않고 문제를 잠시 미루는 일 만하게 된다. 점점 더 다가오는 출시 날은 편하게 잠드는 것도 쉽게 허락하지 않는다. 인생이 힘들고 고달프게 보이게 만드는 것들로 가득 차 보이는 것도 아마 그 때쯤일 것이다. 하지만, 결국 몇 가지는 포기할 수밖에 없고, 차기 변경 점에 적용할 것을 동의하게 되는 것이 일의 수순이다. 안되는 것을 억지로 우긴다고 되는 것이 아니기 때문이다. 그리고, 그 약속도 제대로 지켜질 수 있을지는 아무도 확신하지 못한다.

어쨌든 요구사항은 일부가 삭제되거나 차기 버전으로 미루어지고, 결국 제품은 일정 지연을 줄여서 출시된다. 프로젝트의 측면에서는 실패지만 제품은 지연되어서라도 나오게 된다. 사실 이런 제품이 시장에서 성공하는 것은 운일 뿐이다. 경쟁사는 이미 저 앞에 있는 것처럼 보이고 가격까지 저렴하면 판매 실적은 해가 바뀐다고 해서 나아지지는 않는다. 그럼 왜 이런 일이 발생했는지에 대한 반성을 연 말쯤에는 하게 될 것이다. 그때 나오는 다양한 이야기 중에 "과도한 일정 단축", "개발 역량 부족", "인력 부족(일종의 역량)", "지나치게 많은 요구사항" 혹은 "지나치게 높은 성능"과 같은 것들을 모두가 다 모인 자리에서 반성문에 넣어서 보고서를 만들 것이다.

그렇다고 다음 번 과제를 진행하는데 이런 것들을 반영할 수 있을까? 그렇지는 않을 것이다. 과제에 투입되는 인력은 여전히 부족할 것이고, 실적에 시달리는 윗 사람의 일시적인 정신적 만족감을 위해서 “도전적인” 일정 계획을 보고할 것이다. 상품기획도 고객과의 관계를 생각해서 경쟁력 있는 로드맵을 과감하게 잡을 것이 분명하다. 물론 그것에 대해서 고객은 이미 속은 경험이 있기에, 예전보다 일정 준수에 대한 믿음은 많이 약해졌을 것이 분명하다.

여기서 한가지 의문이 드는 질문은 “과제 후반부에서 요구사항을 줄이는 것이 과연 효과가 있을까?”이다. 사실 과제 후반부에 요구사항을 줄이는 것은 개발 내부에서 반발할 수 있다. 이미 구현된 기능(테스트는 끝나지 않았지만)을 삭제하는 것도 만만한 일이 아니기 때문이다. 구현된 기능을 제거하는 것도 일이다. 코드를 제거하는데도 시간이 필요하다는 것이다. 그냥 줄이라고 해서 다음 날 바로 줄어드는 것이 아니기 때문이다. 어떤 경우에는 제거하는 것보다 차라리 끝까지 계속 구현하는 것이 더 빠를 수도 있다. 이미 구현된 코드는 다른 코드와 의존관계를 만들며, 코드의 변경이 전체 코드에 영향을 주기 때문이다.

코드 간의 상호 작용으로 인해서, 제거할 부분에 대해서 미리 대비하지 않은 경우에는 일을 더 꼬이게 만들 수 있다. 모듈화와 계층적인 구조가 필요한 이유가 이 때문이다. 모듈화 되지 않은 코드는 구현도 제거도 쉽지 않다. 계층화가 되지 않은 코드는 상호 참조와 생략된 계층으로 인해서 발생하는 의존성을 가진다. 따라서, 일정 부족으로 인해서 급하게 구현된 코드는 사용하지 않는 기능이라고 하더라도 그냥 코드에 포함할 수밖에 없다. 제거 자체가 불가능하기 때문이다. 따라서, 구현해야 할 요구사항에 대한 관리는 과제의 초기부터 제대로 해야 한다. 그렇지 않으면 과제의 후반부에 기능을 줄인다고 해서 일이 대폭 줄어들지는 않을 것이기 때문이다.

요구사항의 우선순위는 지속해서 검토해야 한다. 이해관계자 간에 협의가 되지 않고 일방적으로 주어지는 요구사항은 과제의 집중도를 약하게 만들기 때문이다. 변경되지 않는 것은 관리되지 않거나 관리할 필요가 없는 것들이다. 관리란 변경이 발생하는 것이 무엇인지, 어떻게 변경할 것인지를 체계적으로 정의하는 것이다. 우선순위로 주어진 요구사항은 일의 범위와 일정에 큰 영향을 주고, 또한 그 우선순위는 고정된 것이 아니라 시장의 상황에 맞게 끝없이 변한다. (혹은 누군가의 취향에 따라서도 변할 수 있지만) 변화에 대한 관리를 어떻게 하느냐에 따라 과제의 성공 여부가 결정난다고 해도 과언이 아닐 것이다.

# [ TDD(Test-Driven Development)에 대해-01 ]

TDD로 과제를 수행하겠다는 과제 리더(Project Leader:PL)의 말을 들었을 때, 가장 먼저 머리 속에 떠오른 생각은 "저 사람이 제대로 TDD를 이해하고 있을까?"라는 의문이었다. TDD는 개발 방법론에 대해 완전히 새로운 것을 받아들일 준비가 되어 있어야 가능하다. 개발자만 그것을 안다고 되는 것이 아니라, 관리자도 함께 해야만 제대로 할 수 있기 때문이다. 테스트를 먼저 개발하는 습관은 소프트웨어 개발자에게 익숙하지 않다. 대부분 개발자는 요구사항을 분석한 후, 자기 생각을 스케치하고, 코딩한 후에 테스트를 진행한다. 이와 같은 일반적인 과정에 익숙한 사람에게 TDD를 하라고 한다면, 어디서부터 시작해야 할지 막막해지는 게 사실이다.

시작은 단순한 것부터 해야 한다. 예를 들어, 해야 할 일부터 먼저 정리하자. 어떤 일을 해야 하는지를 보고 리스트(List)로 정리하도록 한다. 리스트에 우선순위도 함께 고려되면 좋을 것이다. 그리고나서, 가장 우선순위가 높은 것을 어떻게 구현할지 먼저 고민하도록 한다. 고민은 머리로만 하지 말고, 간단히 그림을 그려서 정하도록 한다. 테스트 환경을 구축하고 나서, 가장 구현 우선 순위가 높은 함수에 대한 스펙을 만든다. 스펙은 함수의 이름과 가져야 할 파라미터와 복귀 값을 정하는 것으로 충분하다. 이제는 테스트를 만들어 스펙을 충족하는지 여부를 검증하는 것이다. 당연히 아직 구현된 코드가 없기 때문에 테스트는 실패할 것이다. 이제는 테스트를 성공시키기 위해서 함수를 구현해야 할 것이다.

TDD는 이런 식으로 테스트를 먼저 만들고, 테스트를 성공시킬 만큼의 코드를 반복적으로 구현해 나가는 활동이다. 구현 과정에서 발생하는 코드의 중복이나 낭비는 리팩토링을 거치면서 제거하게 된다는 것도 빼놓을 수 없다. 즉, “테스트->코딩->리팩토링” 과정을 빠르고 짧게 반복적으로 실행하는 것이다. 이를 통해서 얻는 장점은 버그가 발생하는 순간에 수정할 수 있는 기회를 얻는다는 점이며, 테스트를 고려한 개발로 인해서 소프트웨어 구조 개선과 자동화된 테스트를 확보하게 된다는 것이다.

빨리 자신의 코드에 대한 피드백(Feedback)을 받을 수 있다는 점이TDD의 핵심 가치다. 대부분의 소프트웨어 개발은 피드백 사이클이 느리다는 단점이 있으며, 코딩과 테스트 실행 사이의 인터벌(Interval) 간격이 커질 수록 버그가 파고들 공간이 늘어난다. TDD에서 만들어진 호출은 테스트 스크립트(Test Script)가 되고, 만들어진 함수는 테스트된다. 또한, 테스트 자체가 함수에 대한 사용 메뉴얼이 될 수도 있으며, 실용적인 예를 제공해 줄 수도 있다. 그리고, 아직까지 프로그램의 실행만큼 가장 확실한 검증 방법은 존재하지 않는다.

물론, 테스트 스크립트를 만들고 관리하는 일이 쉬운 일은 아니다. 하지만, 나중에 코드의 수정으로 문제가 생겼을 경우, 어떤 테스트가 실패하는지 재빨리 확인할 수 있으므로, 결국에는 일을 덜어주는 효과를 가져다준다. 이때 좀 더 높은 효율을 가지려면 좋은 툴을 익히는 것도 중요하다. 조그마한 변경이라도 반드시 테스트를 거쳐야 한다. 변경은 아무리 작더라도 전체 시스템에 영향을 주기 때문이다. 개발자가 많은 테스트를 가능한 "자동으로" 할 수 있도록 만드는 것은 중요한 일이다. 대부분의 소프트웨어 개발자는 코딩에 10%, 디버깅에 80%의 노력이 투자하기 때문이다. (나머지는 10%는 문서나 회의로 남겨두더라도) 따라서, 디버깅 시간을 줄일 수 있다면 더 생산적인 활동에 집중할 가능성도 커지게 된다.

다시 처음으로 돌아와서, 이런 생각들을 TDD를 경험하지 못한 과제 PL이 제대로 이해할 수 있을까? 실제로 소프트웨어 공학에서 이야기하는 단위 테스트를 많이 하는 것을 TDD로 오해하는 사람도 있다. 혹은, “Agile”을 외치는 사람들도 제대로 “XP(Extreme Programming)”의 실천 방법이 왜 필요한지조차 이해하지 못하는 경우도 있다. 테스트 케이스의 수에는 민감하게 반응하지만, 테스트 커버리지가 무엇인지는 궁금해하지 않는다. 어떤 개발 방법론을 말로 떠들기 보다 실제로 그것이 가지는 가치를 이해하는 것부터 시작해야 한다. 마치 C언어처럼 C++언어를 사용하지 않기 위해서 객체지향 개념을 이해하려고 노력하듯, 개발 방법론도 그것이 가지는 핵심 가치와 그것을 하기 위한 실천 방법들을 알아야 한다. 단순히 비슷하게 한다고 해서 되는 것이 아니라, 초보자처럼 정확하고 천천히 반복적으로 실천해야 한다. 습관이란 반복을 통해서 생성되며 의식적인 노력없이는 기존의 태도를 바꾸지 못한다.

# [ 개발자 테스트에 대한 단상 ]

소프트웨어 개발에서 줄 일 수 있는 유일한 일정은 테스트 밖에 없다. 테스트가 줄어들지 않으면 개발일정의 단축은 불가능하다. 하지만, 테스트는 기본적으로 테스터가 하는 일이라고 생각하는 순간 개발일정 단축은 개발자의 책임 범위를 벗어나는 일이 된다. 수동적인 입장에서 문제가 없기를 비는 것은 아무런 대책이 되지 못한다. 오히려 개발자가 버그를 줄이는 적극적인 방법을 익히는 것이 현실적으로 더 올바른 선택이라고 볼 수 있다. 문제는 버그를 개발자들이 줄일 수 있는 방법을 찾는 것이며, 테스트는 개발자들이 발견하지 못한 문제를 최종적으로 테스터들이 점검하는 의미에서 실행되어야 한다.

문제는 개발 기간 속에 담겨있는 개발 활동 간의 관계 파악에서 출발해야 한다. 즉, 어떤 필요한 활동에 대해서 시간이 없다고 그냥 넘어가 버리면, 그것이 점점 수면 위로 올라올 경우 걷잡을 수 없는 거품이 되어버려 전체 과제 기간을 집어 삼켜버리기 때문이다. 문제가 커지게 되면 그 원인을 찾는다고 시간을 보내게 되며, 대책을 세운다고 또 다른 시간을 낭비하게 된다. 중요한 것은 문제를 제거하는 것이지만, 정작 다른 사람의 손에서 이러쿵 저러쿵 논의되는 상황이 되어버린다. 따라서, 개발 기간의 초반에 행해지는 활동에 대해서 좀 더 파고들지 않으면 안 된다.

소프트웨어 개발자라면 항상 자신이 짠 코드에 대해서 완벽하다고 자부해야 하겠지만, 사실 왜 완벽한지를 검증하려고는 하지 않는다. 막연히 자신이 짠 코드라서 잘짰다고 생각하는 것은 어딘가 좀 근거가 부족해 보인다. 차라리 코드가 얼마나 잘 테스트되었는지 객관적으로 증명할 방법을 가지고 이야기하는 것이 옳을 것이다. 코드의 테스트 커버리지(Test Coverage)를 측정하는 것도 한 가지 방법이다. 도메인에 필요한 코딩 룰을 검사하고, 정적분석을 통해서 잠재적인 오류가 없다는 것을 보여주며, 코드 리뷰를 통해서 다른 개발자의 동의를 얻는 것도 객관적인 증명 방법으로 활용할 수 있다.

하지만, 실무를 담당하는 개발자들은 종종 마치 당연하기라도 한듯이 위에서 말한 것들은 그냥 시간이 없다는 말로 그냥 넘어가 버린다. 버그가 발생하지 않기만을 빌지만 그런 일은 결코 없다는 것도 잘 알려진 사실이다. 오히려 테스트 초기에 대량으로 발생하는 버그는 어쩔 수 없는 일이라고 생각하기도 한다. 과제 후반으로 갈수록 시간은 더 부족해지고, 버그를 아무리 수정해도 새로 발견된 버그로 인해서 줄어들지 않는 일이 반복된다. 심리적으로 방어적이고 위축되는 것은 어쩌면 당연하다. 시간이 부족해지면 근본적인 원인을 제거하는 것보다 현상을 완화시키려고 노력하게 만든다. 해야할 일을 제 때 하지 않으면 다시 할 수 있는 순간은 결코 찾아오지 않는다.

개발자의 테스트는 단위 테스트와 통합 테스트로 나누어 진다. 단위 테스트는 함수와 클래스를 대상으로 하며, 구현 활동의 일부로 취급되어야 한다. 통합 테스트는 하나 이상의 모듈을 대상으로 호출 시나리오를 기반으로 실행되며, 단위 테스트의 확장으로 가능하다. 물론, 개발자의 테스트도 일이다. 따라서, 당연히 비용이 들어간다. 대략적으로 테스트 없이 코드를 작성하는 것보다 최대 150%정도 추가적인 노력을 투자해야 할 수도 있다. 하지만, 이런 노력이 검증된 코드의 범위를 늘릴 수 있다면, 테스터에게 전달되는 코드에서 발생하는 버그 수는 극적으로 낮추는 효과를 볼 수 있을 것이다. 잠재적으로 버그가 될 수 있는 부분들을 최대한 많이 걸러낼 수 있다면, 개발자가 생각하지 못한 부분에 대해서만 테스터들이 버그를 발견할 가능성이 높기 때문이다.

제대로 일하기 위해서 가장 처음에 할 일은 자신이 작성한 코드에 대한 테스트를 만드는 것이다. 이것이 익숙해 지면 테스트를 미리 만들고 코딩하는 것으로 가면 된다. 중요한 것은 언제든 실행할 수 있는 테스트를 가지고 있다는 점이며, 변경에 대한 안전망(Safety Net)이 있다는 것이다. 이미 버그를 가장 많이 사전에 제거할 수 있는 방법으로 단위 테스트와 통합 테스트 등이 효과적이라는 것은 잘 알려진 사실이다. 부족한 것이 있다면 그런 경험을 가진 개발자들이 많지 않다는 점과 효과를 제대로 이해하는 과제 책임자들이 없다는 것이다. 일정이 부족하다는 생각은 개발자를 검증되지 않은 코드를 만들도록 부추기며, 관리자들이 “테스트와 디버깅의 반복”을 최대한 많이 가지도록 이끌어 간다. 하지만, 버그를 만들고 그것을 고치는 것보다, 버그를 사전에 제거하고 남은 버그만 고치는 것이 부족한 일정을 조금이라도 더 효율적으로 이용할 수 있는 방법이다.

오늘부터라도 자신이 만드는 코드에 대해서 가장 작은 실행단위로 테스트를 하는 방법을 만들어야 한다. 모르겠다면 남은 어떻게 했는지 배울 필요도 있다. 시중에는 이미 이런 것과 관련된 많은 책과 도구들이 나와 있으며, 게으르지만 않다면 충분히 익힐 시간은 언제든 만들 수 있다. 대부분 나쁜 습관은 몰라서 고치지 못하는 것이 아니라, 알면서도 바꾸지 못하기 때문에 이어진다. 작은 변화를 통해서 조금씩이라도 고쳐나갈 수 있다면, 결과는 크게 바뀔 수 있는 것이다. 개발자의 테스트는 습관이며, 좋은 습관을 가진 개발자가 발전하고 있다는 느낌을 가질 수 있다. 스스로 유능한 개발자라고 믿고 싶다면 객관적으로 자신의 역량을 검증할 수 있는 방법을 사용해야 하며, 막연한 느낌이 아닌 구체적인 근거를 제시할 수 있어야 할 것이다.

# [ 지금이 가장 적절한 순간이다 ]

문제를 발견한 순간이 수정할 수 있는 가장 좋은 시간이다. 시간이 흐르면 했던 일도 까먹게 되고, 결국 나중에 고생한 후에 문제의 원인을 알게 된다. 따라서, 뭔가 문제가 있다고 생각되면 즉시 바꿀 생각을 해야 한다. 소프트웨어를 초기부터 개발하는 경우에는 처음부터 실행할 수 있게 하는 것이 중요하다. 소프트웨어의 최종 목적은 실행되는 것이기에, 그것에 한 발짝 더 다가가는 것은 의미 있는 일이다. 모든 소프트웨어는 실행되기 전에는 어떤 오류가 있을지 알 수 없다. (물론, Code Review를 통해서도 중요한 버그들을 걸러내기야 하겠지만, 어쨌든 실행되어야 한다.) 마찬가지로 테스트의 정의도 버그를 확인하기 위해서 프로그램을 실행하는 것일 뿐이다.

“DRY(Do not Repeat Yourself)”라는 것은 많이 들어봤을 것이다. "반복하지 말라"라는 말이다. 하지만, 상업적으로 검증이 완료되어 팔리고 있는 많은 소프트웨어가 정말 이런 원칙에 따라서 만들어졌을까? 실제로는 그렇지 않다는 것이 사실이다. 소프트웨어 전문 개발 회사들은 이런 문제를 이미 오래 전에 인식했으며 다양한 방법으로 해결하는 툴도 제공하고 있다. 주로 정적인 분석(Static Analysis)을 통해서 코드의 중복이 발생한 부분을 찾아내는 방법을 사용한다. 그런데도 이런 문제가 지속해서 발생하고 있는 이유는 프로그래머의 개발 편의에서 그냥 그렇게 했다는 것이 맞을 것이다. 중복된 코드를 발견하면 그것이 어떤 이유에서건 수정해야 할 부분으로 인식할 필요가 있지만, 다양한 핑게로 나중으로 미루는 것이 일반적 태도다.

문제를 초기에 진압하는 비용은 적지만, 문제가 있다는 것을 알고 나중에 해결해야지 하고 생각하는 순간 수정을 위한 비용은 예측하기 곤란한 상태가 된다. 건물의 기둥이 제대로 받쳐주지 못하는데 이런 저런 것들을 그것에 의지해서 덧붙여 나간다면 무너지는 것도 시간문제일 뿐이다. 따라서, 지금 알고 있는 문제는 지금 해결하는 것이 최선이다. 나중에 알게 될 문제는 그 때 해결하면 된다. (미리 안다면 더 좋겠지만) 문제를 쌓아두면 복잡한 문제가 되지만, 해결하고 넘어가면 단순한 문제로 남게된다. 코드는 시간이 흐르면 복잡하게 변하는 것이 일반적인 진화 과정이다. 따라서, 단순한 문제도 시간이 지나면 복잡한 문제로 변하고 만다.

문제는 적게 만드는 방법은 많은 사람이 이야기해 왔듯이 불필요한 부분에 대한 정리다. 불필요한 인터페이스를 없애고, 불필요한 의존성을 제거하고, 불필요한 코드와 주석 등을 없애는 것이다. 변수의 이름을 정리하고, 함수의 이름도 정리하고, 클래스 및 파일들의 이름도 정리해야 한다. 디렉터리도 코딩에서는 중요한 정보를 줄 수 있으므로 정리해야 할 대상이다. 오히려 모든 산출물이 정리의 대상이라고 해야할 것이다. 미뤄두었던 숙제를 한 번에 하는 것은 많은 시간과 추가적인 오류를 만들듯이, 이런 활동들은 개발 기간 동안 지속해서 이루어져야 한다. 벼락치기는 뇌를 번갯불에 태워버리고 순식간에 사라질 운명적인 것들에 적합하다. 코드는 개발자 자신보다 더 오래 살아 남을 가능성이 높기에, 항상 세심한 관리를 해 주어야 한다. 지금을 놓치면 나중이라는 시갼은 결코 다시 오지 않는다.

# [ 개발자 테스트(지난 날의 회고) ]

지난 시간동안 임베디드 시스템(Embedded System)을 만드는 책임자로 일해왔던 경험을 잠시 이야기하려 한다. 개발하려고 했던 시스템은 인터넷과 특정 사용자 어플리케이션 등을 이용해서 시스템에 접근할 수 있었으며, 시스템에서 전달되는 정보는 주기적인 스트림(Stream)의 형태로 클라이언트에 전달되는 것이었다. 문제는 이미 있는 시스템을 신규 플랫폼으로 완전히 교체하는 것이었으며, 기존 코드(Legacy Code)에 대규모의 변경을 가해야 하는 상황이었다. 물론, 변경에 대해 대비를 할 수 있는 단위 테스트와 같은 것은 전혀 없었다.

먼저 내게 주어진 인력의 구성을 보도록 하자. 해당 도메인을 잘 모르는 신입 사원 3명과 과제가 시작되고 3개월이 지난 시점에 투입된 고참 책임 1명과 다른 분야에서 일하다가 온 3년 차 책임 1명이 전부였다. 내가 만들어야 하는 플랫폼은 대략 40만 라인의 코드를 가지고 있었으며, 우린 전부 해당 분야에 대한 전문 지식은 거의 없었다. 사실 이런 상황에서 과제를 진행하는 것은 위험 부담이 너무나 크다는 것을 이미 잘 알고 있었다. 하지만, 어쩔 수 없이 진행해야 할 상황이었고, 관련 부서에서는 협력할 수 있는 선임급의 도메인 경험이 있는 연구원들을 배치해 주겠다는 약속을 받은 상태였다. 하지만, 그런 일을 일어나지 않았다. 코드를 제외한 관련 부서의 지원은 앞에서 나열한 몇 명의 인력을 제외하고는 거의 없었다.

가장 먼저 닥친 문제는 기존의 코드를 받는 일이었다. 여러 번의 요청에 마지못해 전달해준 기존의 코드는 컴파일도 제대로 되지 못했고, 이유를 확인한 결과 몇 개의 모듈이 빠진 상태였다. 겨우 컴파일이 되었을 때 컴파일러의 경고 메시지가 수백 개(대략 220여개)가 화면을 가득 채웠으며, 이런 이야기를 해당 과제의 PL에게 물었을 때 돌아온 대답은 "컴파일 옵션에서 경고 메시지 출력을 빼세요."라는 말이었다. 잠시동안 내 귀를 의심했지만, 잠시 후 “그럼 그렇지.”라는 허탈감이 들었다. 어쨌든 이것도 신입 개발자를 교육하는 좋은 기회라고 생각했으며, 모든 컴파일러 경고 메시지를 없애는 작업이 시작되었다. 코드를 개선하기 전에 일단 기계적으로 자동 검증되는 문제점을 고치는 것 부터 시작하는 것이 좋다고 생각했기 때문이다.

목표는 일단 하위 계층에 속한 모듈들을 정리해서 깔끔하게 만드는 것이었다. 왜냐하면, 기존의 코드는 상위, 하위 계층이라는 개념이 없이 만들어진 스파게티 코드의 전형으로 보였기 때문이다. 문서화도 신경을 써야 했다. 최대한 기본 틀에 대해서만큼은 제대로 만들어 보자는 것이 지금의 인력으로 할 수 있는 일의 전부라고 생각했었다. 제대로 하기 위해서는 설계 및 제공해야 할 API(Application Programming Interface)에 대한 문서가 필요했으며, 플랫폼 과제는 내부에서 사용자를 확보해야 하기에 문서화는 더욱 중요했다. 따라서, 기술 문서를 작성하는 전담 부서에 도움을 요청해서 지원받았다. 사실 기술 문서를 작성하는 부서와 좋은 관계를 유지하고 있어서 충분히 가능한 일이었다.

부서원들은 일하면서 일을 배워야 했다. 좋은 코드가 무엇인가를 알아야 했기에 "Clean Code", "Code Craft", "Code Complete"와 같은 책으로 세미나도 시켜야 했고, 기초부터 알기 위해서 C++을 다시 공부기도 했다. 코드 리뷰를 도입하기 위해서 "Review Board"를 설치하고, 코드의 품질 수준을 알기 위해서 "Jenkins"라는 빌드 서버에 "Simian", "CppCheck"와 같은 툴들을 도입했으며, 정적 테스트 분석을 위해서 "Coverity의 CIC, CSA" 등도 사용했다. 특히 정적 분석의 경우에는 오류(False)의 유형을 분석해서, 해당 유형 별로 대처 방안을 작성한 문서도 작성했었다.

매일 변경되는 코드에 대해서 단위 테스트를 하기는 어렵다고 생각했지만, 어쨌든 그것을 위한 초석을 다진다는 의미에서 "Eclipse"와 같은 툴에 "GDB Remote Debugging"과 “Subversion”에 대한 체크 인, 체크 아웃, “ReviewBoard”연동, “CppUnit” 플러그 설정도 완료했다. 이것으로 대략적인 개발 환경에 대한 구축은 완료되었지만, 변경된 코드에 대한 테스트 환경을 어떻게 만들 것인가는 중요한 숙제로 남아 있었다. 수시로 코드는 변경되지만 적절한 테스트가 없다면 작업 자체가 무의미가 될 수도 있기 때문이었다.

일(Daily) 단위의 변동에 대한 검증을 위해서는 선택한 방법은 시스템 테스트를 실시하는 것이었다. 적어도 변경에 의해서 과거에 동작하던 기능에 문제가 생겨서는 안되기 때문이었다. 가장 쉽게 자동화할 방법을 찾기위한 고민 끝에 웹(Web)을 통한 자동 테스트를 할 수 있는 "Selenium"을 선택했다. 다행히 우리가 개발하는 제품은 웹 사용자 인터페이스가 있었으며, 그것을 통해서 제품의 기능 대부분을 제어할 수 있었기 때문이다. 자동화 테스트를 위해서는 스크립트들도 개발자가 직접 만들어야 했었다. 한 가지 문제점은 “Selenium”은 “ActiveX”와 관련된 부분과 제대로 동작하지 않는다는 것이었지만 테스트가 불가능할 정도의 큰 제약 사항은 아니었다.

회사 차원에서 세부적인 프로세스와 도구 및 툴에 대한 지원이 없었기에, 앞에서 나열한 도구의 대부분은 팀 차원에서 내부의 서버에 오픈 소스를 이용해서 구축했다. 기존의 경험이 없던 인력들이 모여서 그런지, 생각보다 내부적인 반발은 없었다. 시간이 흐를수록 개발자들은 빠르게 환경에 적응할 수 있었으며, 오히려 신입 사원으로 들어온 개발자의 경우에는 다른 팀에서 일하는 동료들보다 더 많은 것을 단기간에 습득할 수 있었다. C언어도 제대로 사용하지 못하던 개발자들이 6개월이 지난 시점에는 거의 선임급 사원이 C언어를 구사하는 수준으로 올라선 것처럼 보였다. 특히 코드 리뷰를 통해서 서로 학습할 수 있는 분위기를 만든 것이 이런 실력 향상의 주요한 요인처럼 보였다.

하지만, 여전히 문제는 끊임없이 나왔으며, 특정 문제에 대해서는 지속적으로 검출되는 것을 막아내지 못했다. 그런 문제에 대한 해결책으로 마지막에 했던 것은 해당 부분의 코드를 완전히 새로 작성하는 것이었다. 그것이 완성되기 전에는 지속적으로 늘어나는 버그에 대해서 위로부터 다양한 질책과 비난을 감내해야 했었지만, 그 와중에 완성한 코드는 생각보다 안정적이었다. 지금 생각해 보더라도 그 때 내린 결정이 제대로 된 것이었다는 확신할 수 있을 정도다. 기존의 코드는 문제가 발생할 때 땜질 식으로 표면적인 증상을 완화하는데 집중했지만, 우리가 내린 결정은 근본적을 문제점을 해결하는 방안이었기 때문이다. 때로는 가까운 길로 가기 보다 먼 길을 돌아가는 것이 더 좋은 결과를 만들어 낼수도 있다는 결론을 얻었다.

아직 다 하지 못하고 남은 일은 단위 테스트를 도입하는 것이었으며, "Fitness"가 그 부분을 채워줄 수 있다고 생각했다. 하지만, 정작 중요한 것은 결국 테스트를 먼저 만들어야 한다는 사실을 연구원 개개인이 깨닫는 것이었다. 수정된 내용이 매일 매일 검증받지 못하면, 다음에 진행하는 일도 의미가 없음을 알아야 했다. 하루의 시작과 함께 수정된 내용은 그 날이 지나기 전에 반드시 테스트가 되어야 하며, 테스트 결과가 개발자에게 피드백으로 빠르게 주어지도록 만드는 것이 핵심이었다. 즉, 문제를 발견했을 때 즉시 수정해야 개선이 이루어질 수 있다는 것이다. "개인과 조직 차원에서 코드에 대한 청결함을 유지하는 것"과 "변경에 대한 빠른 검증 및 피드백” 이 과제를 성공적으로 완료할 수 있도록 만드는 효과적인 방법이라는 것을 깨달았다.

과제는 기존의 여러 제품에 대해서 동시에 적용할 수 있는 수준의 결과물을 얻었으며, 데모도 여섯 개의 모델에 대해서 4종류의 서로 다른 CPU상에서 실시했다. 하지만, 과제는 정해진 일정, 인력, 비용으로 끝났지만, 과제를 적용한 제품은 끝내 만들지 못했다. 결과물의 품질 수준은 QA부서를 통해서 공식적인 검증을 거쳤지만, 과제 결과물을 제품에 적용하는 것은 기술적인 것이 아니라, “권한을 가진 사람의 결단과 의지”와 밀접한 관련이 있다는 것을 다시 확인할 수 있었다. 비록 제품화에는 실패했지만 과제에 참여했던 모든 개발자는 "우리는 팀(Team)이다." 라는 생각만큼은 공유하고 있었다고 본다. 적어도 이것 하나는 언제나 가슴 속에 담고 남은 회사 생활의 여정을 이어갈 것이다.

# [ 지속적인 통합과 배포 ]

소프트웨어 개발의 시작에 필요한 것은 사람만이 아니다. 적절한 수준의 품질을 가지는 소프트웨어를 첫 날부터 만들어 낼 수 있는 준비를 해야 하는 것이 정답이다. 즉, 첫 날부터 이미 컴파일되는 코드를 저장소에 넣어 두어야 하며, 자동 빌드와 테스트, 인수 테스트, 배포 등이 될 수 있는 환경을 구축해야 한다. 성공으로 향하는 모든 것은 이 때부터 시작된다.

시작이라고 하기에 거창할지도 모르지만, 우리는 대체로 시작을 관대하게 보는 경향이 있다. 즉, 첫 날 안된 것을 둘째 날이나 혹은 그 다음으로 미루다 보면 그 달의 마지막 날이 될 수도 있다. 하지만, 이렇게 미뤄둔다고 숙제가 없어지는 것은 아니다. 자신이 맡은 과제의 미뤄둔 숙제가 언제고 반드시 일정지연이라는 보답을 하게 될 때가 있을 것이다. CI(Continuous Integration)와 CD(Continuous Delivery)가 중요한 이유는 언제든 배포할 준비가 된 상태로 과제의 코드 베이스(Code Base)를 유지한다는 점이다. 누군가 와서 데모를 부탁할 때, 데모 날짜를 정하고 그 날에 맞춰서 모든 것을 준비하는 것은 실패할 가능성이 높다. 일종의 “데모 증후군”처럼 자신의 자리에서는 잘 되지만, 하필 그 날이 되면 제대로 동작하지 않는 경우를 종종 볼 수 있다. 이것은 개발 환경과 실행 환경이 일치하지 않을 때 자주 겪는 문제다.

소프트웨어가 가지는 궁극의 목적은 “고객의 사용 목적에 맞는 동작하는 소프트웨어를 제공하는 것”이다. 동작한다는 말은 실행할 수 있어야 한다는 이야기며, 가능하면 버그가 없어야 한다는 뜻이다. (완벽하게 버그가 없으면 좋겠지만 불가능한 일이다.) 그러한 소프트웨어를 제공하기 위해서 한 번의 클릭(Click)으로 자동 완성할 수 있다면 얼마나 좋을까? 사실 다수의 개발자가 참여하는 과제에서 가장 많은 시간을 소비하는 일은 전체 코드를 통합해서 제대로 동작하는지 검증하는 일이다. 이런 활동이 매일 매일 자동으로 되고 있다면, 사람의 공수(Effort)를 줄여 더 완벽한 소프트웨어를 만드는 일에 투입할 수 있게 된다.

하지만, 경험한 회사들 대부분은 그런 것을 별로 중요하게 생각하는 분위기가 아니었다. (물론, 지금 이 글을 읽고 있는 대부분 사람들이 동의하지 않을지도 모르지만) 특히 테스트 환경이 복잡하고 힘든 곳은 더욱 심하다. 예전에 과제 책임자를 만나서 이런 저런 이야기를 나누던 중, "얼마나 자주 코드 저장소에 체크인 하는가”를 질문한 적이 있었다. 그 때 “대략적으로 자신이 담당한 코딩 부분을 완료한 후에 정상적으로 동작한다는 것을 간단히 확인하고 체크인 한다”라는 답을 들었다. 즉, 적어도 하루 단위의 통합이 아니라 일주일에서 이 주일, 혹은 그 이상의 기간 동안 전체적인 코드의 통합은 이루어지지 않고 있다는 말이었다. (심지어 버전관리 시스템이 없는 경우도 보았다. 버전관리 시스템이 있다고 하더라도 마치 파일 서버처럼 한 사람이 모든 코드를 통합해서 관리하는 경우도 보았다.)

통합을 짧게 자주 하지 못하면 통합은 어려운 일이 된다. 빌드는 지속적으로 깨진 상태가 되고, 그것을 수정하기 위해서 코드도 수시로 변경될 것이다. 당연히 자동화된 테스트는 불가능할 것이며, 검증된 코드와 그렇지 않은 코드도 구분하기 힘들 수 있다. 사람들은 점차 자신의 코드에 집중하기 보다, 다른 사람의 코드와 통합하는데 더 많은 시간을 투입하게 된다. 이런 일이 반복적으로 발생한다면, 개발 시간의 대부분은 통합과 빌드에 투자한다고 봐도 과언이 아닐 것이다. 어려운 일은 자주 작게 하는 것이 해결방법이지만, 그런 습관이 없는 개발자들은 생각보다 훨씬 많다. 그런 사람이 경험을 갖춘 개발 리더가 되면 문제는 더 심각해진다. 오히려 좋은 습관이 나쁜 습관에 밀려나는 현상이 생기기 때문이다.

자동으로 빌드를 하지 못하고 자동으로 테스트가 되지 않는 코드가 있다면, 우린 시작부터 잘못된 길로 들어섰다는 느낌이 가져야 한다. 나중에 찾아올 대가에 대한 대비를 미리 갖추어야 한다. 가장 중요한 것은 어쨌든 테스트다. 소프트웨어 개발자는 테스트로부터 자유롭지 못하며, 자신이 만든 코드에 대해서 제대로 동작한다는 것을 보장할 의무가 있다. 테스트는 후반부의 작업이 아니라, 구현(코딩)의 일부로 받아들여야 한다. 이것이 힘들다고 이야기한다면, 소프트웨어 개발보다는 다른 분야의 일자리를 빨리 알아보는 것이 좋을 것이다. 개발 첫 날 부터 빌드와 테스트를 할 수 있다면, 마지막 날까지 그것이 유지될 가능성이 높다. “깨진 빌드와 테스트”를 “깨진 창”처럼 방치하지만 않는다면, 누구도 자신으로 인해서 상태가 악화되는 것을 보고 싶은 소프트웨어 개발자는 없을 것이다.

# [ 시간의 법칙 ]

누구에게나 주어지는 시간의 길이는 동일하다. 차이가 있다면 주어진 시간 동안 무엇을 했는가에 따라 미래가 달라질 수 있다는 점이다. 과제를 하는 것도 마찬가지다. 누구에게나 동일한 시간이 주어지는 것은 아니지만, 그래도 어느 정도의 시간은 항상 주어지는 게 일반적이다. 그 시간 동안 무엇을 했는가에 따라 미래의 다른 과제에 영향을 주기 시작한다. 특히, 시장 문제에 대한 대응은 언제나 개발자의 마음을 무겁게 하는 주제일 것이다. 시장에서 문제가 발생하면 투입되어야 할 비용이 크다. 개발 초기에 문제를 바로잡았을 경우에 비해서 수십 배가 되기도 한다. 또한, 그런 문제가 발생하면 신규 개발 과제에 투입되는 노력은 줄어들게 되며, 지연될 가능성도 높아지게 된다.

시장에서 발생하는 문제는 시장에 소프트웨어가 배포되는 순간부터 발생하기 시작한다. 모든 문제의 원인이 그렇듯 개발자의 부주의나 이해 부족에서 발생하는 경우가 많다. 고객이 무엇을 원하는지 모르고 개발할 때 생기는 문제점은 고객의 불편일 수도 있고 성능 부족일 수도 있다. 하지만 정말 문제가 되는 것은 개발하는 중간에 그런 것들을 일일이 전부 다 고려할 수 있는 여건이 안된다는 점이다. 그렇다면 어떻게 해야 이런 것들을 미리 해결할 수 있을까? 방법은 한 가지뿐이다. 고객의 입장에서 제품을 사용해보는 것이 유일한 해결책이다. (물론 설령 그렇다고 하더라도 모든 고객의 요구사항을 알 수 있는 것은 아니지만)

고객의 입장에서 제품을 사용해 보는 것이란 무엇일까? 다른 말로 표현하면 고객이 원하는 품질을 만족하는 시킬 수 있는가를 확인하는 것이다. 기능적이건 비기능적이건 고객은 자신이 원하는 바를 모호하게 가지고 있는 경우가 많다. 그런 것들을 일일이 다 알 수 없기에 미리 제품 조금 만들어 고객(사용자)의 의견을 들어서 조금씩 고쳐나가게 된다. 따라서, 제품을 만드는 중간 단계 마다 체크 포인트(Check Point)를 넣어서, 특정 기능을 보여주고 그것에 대한 피드백을 개발자들이 직접 들어야 한다. 너무 많이 개발이 진척된 상황에서는 수정에 부담이 크기에, 가장 비용이 적은 개발 초기 단계부터 이런 활동을 시작해야 한다.

문서의 항목을 수정하는 데는 5분의 노력이면 충분하지만, 잘못된 구현을 고치기 위해서는 구현, 디버깅, 단위 테스트, 통합 테스트, 인수 테스트, 배포 등, 많은 사람의 노력이 비용으로 들어가야 한다. 따라서, 원하는 스펙이 확정되기 전에 개발에 들어가는 것은 결코 옳바른 일이 아니다. 하지만, 현실은 스펙보다 말이 우선하고, 판단은 분석보다는 경험에 의지하는 바가 크다. 어쨌든 개발의 시작을 구현(코딩)의 시작이라고 보는 관점은 아직도 많은 부분에서 우월하다. 아직도 많은 개발자들이 경험이 이론에 우선한다고 생각하기 때문이다. 물론, 경험과 이론은 각각 따로 존재하는 것이 아니라 상호 보완적으로 생가되어야 할 부분이다.

조금만 여유를 가지고 바라보도록 하자. 스펙이 없는 상황에서 구현이 진행되기 위해서는 기존의 제품이 기반한 공용화된 부분에 한정해야 한다. 즉, 이미 사용할 것이 확정된 것들은 구현을 미리 시작해도 상관없다. 공용화 기반 위에 사용자가 원하는 기능을 구현해 나갈 것이다. 시간을 줄이기 위해서 미리 예상되는 기능을 구현하는 것이 아니라, 기능들을 구현하기 위해서 필요한 기반을 조성하는 것일 뿐이다. 기능은 언제든 변경이 있을 수 있으며, 사용자가 원하지 않는 기능은 구현할 필요도 없다. 또한, 잘못 이해하고 있는 기능도 사용자의 피드백이 필요한 부분이다. 구현했다고 다 사용되는 것이 아니라, 사용자가 원해야 필요한 기능이 되는 것이다.

시간은 준비하는 사람에 대해서 보답을 확실하게 하는 편이며, 준비하지 않고 행하는 사람에게는 잔인하게 복수한다. “시작이 반”이라는 말은 제대로 된 준비를 한 사람에게만 해당하는 것이고, 시작과 함께 고생문을 열지 않도록 하기 위해서는 제대로 된 시작을 해야한다. 초기에 투자하는 비용이 크다고 과제가 지연되거나 초과 비용이 지출되는 것은 아니다. 핵심은 어디에 중점을 두고 투자할 것인가를 정하는 것이다. 예를 들어, 꾸준한 사용자의 피드백을 얻을 수 있도록 과제를 준비한다면, 과제의 후반에 집중되는 재작업의 대부분을 피할 수 있을 것이다. 또한, 과제를 진행하는 동안 품질 수준을 높이기 위해서 필요한 조치들에 대한 투자도 결코 오버헤드로 생각해서는 안된다. 오히러 발생 가능성이 있는 오류를 사전에 제거해서 복잡한 상황이 발생하지 않도록 만드는 안전망이 되어줄 것이다. 변덕이 심하다고 고객만 탓할 것이 아니라, 그런 고객을 미리 만족시킬 수 있는 방법을 찾는 것이 오히려 더 효과적인 개발이 될 것이다.

# [ 한 번에 한 가지만 하기 ]

멀티 태스킹(Multi-tasking)이란 말을 들어본 적이 있을 것이다. 동시에 여러가지 일을 수행한다는 의미로 생각하겠지만, 소프트웨어를 개발하는 사람들은 하나의 CPU 코어(Core)는 한 번에 한 가지 일만 처리할 수 있다는 사실을 잘 알고 있다. 사람의 눈으로 보기에 여러가지 일이 동시에 처리되고 있는 것처럼 보이지만, 실제로는 시간을 적절히 나누어서 여러가지 일이 마치 동시에 처리되는 것처럼 보여줄 뿐이다. 인간의 뇌도 그와 비슷하다고 생각할 수 있다. 특히 몰입을 요구하는 소프트웨어 개발 업무는 한 번에 여러가지 일을 처리하는 것이 어울리지 않는다. 시간을 분할해서 여러가지 일을 동시에 처리하는 것처럼 보여줄 수는 있을지 몰라도, 각각의 일이 처리되는 품질은 한 가지 일만 전담할 때보다 낮아지게 된다. 컴퓨터도 처리해야 할 작업이 늘어나면 개별 사용자에게 낮은 품질의 서비스만 보장뿐이다.

한 번에 한 가지만 처리하는 함수, 한 번에 하나의 역할만 하는 클래스, 한 번에 하나의 일만 하는 함수들이 묶인 파일, 그런 파일들이 모여서 시스템에 한 가지 역할에만 충실한 서브 시스템 등등, 우리는 시스템을 이렇게 만드는 데 익숙해져야 한다. 여러가지 일을 처리하기 위해서는 여러가지 상황을 고려해야 하고, 그렇게 고려할 것이 늘어나다 보면 코드도 자연스럽게 길어지게 된다. 길어진 코드는 테스트하기 어렵고, 읽고 이해하는 것도 만만찮은 일이다. 물론 처음부터 그렇게 길게 코드를 만들지는 않았겠지만, 조금씩 기능을 추가하다 보면 어느덧 처음 생각했던 간결함은 사라져 버리고, 복잡한 코드가 자리잡게 되는 것이다. 복잡함은 다양한 생각이 한 곳에 있을 때 나타나는 현상이며, 누락된 완결함(Integrity)이 인지할 수 있는 현상으로 나타나는 것이 버그다.

한 번에 한 가지만 잘하기 위해서는 짧은 코드를 만들어야 한다. 코드를 축약해서 짧게 만드는 것이 아니라, 이해할 수 있는 코드를 짧게 짜야 한다는 의미다. 컴파일러만 이해가 가능한 코드는 자신만의 세상을 살아가는 사람들에게 어울리는 것이고, 대부분의 우리가 만드는 코드는 끊임없이 변경을 위해서 읽혀져야 하는 코드이기 때문이다. 짧은 코드는 여러가지 일을 한 번에 하지 않는다. 최소의 일만 열심히 한다. 다른 곳에 한눈을 팔 필요도 줄어든다. 작은 부분에 대한 몰입은 짧은 시간이 걸리지만, 큰 부분에 대한 몰입은 제대로 집중하는 것만으로도 일과의 대부분을 소모해야 한다. 지속적인 인터럽트(메일, 전화, 옆 동료) 상황에서는 집중하는 것이 쉽지 않다.

우리의 사무실은 몰입과는 거리가 멀다는 것을 잘 알고 있을 것이다. 선진국의 개발자들이 자신만의 작은 방을 가지고 일할 때, 우리는 그 와 비슷한 크기의 방안에 5명에서 10명까지도 들어가서 일한다. 누구 한 사람만 전화 통화를 해도, 그 주변의 사람들은 전부 다 그 사람이 최근에 어떻게 지내는지는 금방 알게 된다. 따라서, 짧게 몰입과 산만을 반복하는 환경에서 일하기 위해서도 반드시 짧은 코드를 짜는 데 익숙해져야 한다. 일을 나누어서(Break-down) 적절한 크기(25라인이나 그 보다 조금 더 긴, 혹은 한 화면을 벗어나지 않을 크기의 길이로)로 나누어서 코딩한다면, 최소한 그 부분에 대해서만큼은 사소한 실수를 크게 줄일 수 있을 것이다. 물리적인 크기가 생각의 범위를 줄이는 것이 아니라, 생각이 물리적인 크기에 의존해서 계층적으로 표현될 때 추상화되 더 쉽다. 사람의 사고란 추상화를 바탕으로 하기에, 구체적인 내용을 숨기고 일의 내용에 집중할 수 있게되는 것이다.

작은 코드는 테스트를 돕는다. 코드 내에서 외부의 자원이 필요하다면, 그 자원은 내부에서 생성하기보다는 넘겨주는 방식이 좋다. 내부에서 생성한 자원은 관리하기가 어렵지만, 외부에서 전달되는 자원은 해당 부분의 코드를 테스트할 때 인위적으로 만들어 줄 수 있기 때문이다. 필요하다면 그런 자원들에 대한 변경도 테스트의 한 부분으로 활용할 수 있다. 작은 코드는 테스트해야 할 경로도 짧기에 여러 개의 분기 문을 가진 코드보다 적은 테스트 케이스(Test Case)로 검증할 수 있다. 작은 부분들로 나누어진 코드는 재활용을 위해서도 편하다. 인쇄를 할 때 활자를 조합해서 인쇄 틀에 넣듯이, 작은 원소들로 이루어진 것들은 다른 부분에 다시 사용될 가능성이 높다. 작은 것을 잘 만들고 관리할 수 있다면, 그것을 모아서 큰 것을 만들지 못할 일은 없다. 오히려 큰 것을 한 번에 만들려고 노력을 낭비하기 보다는 작게 나누어진 일을 조합해서 문제를 해결하는 태도가 컴퓨터 분야의 역사에도 어울리는 방법이다. 유닉스(Unix)가 그 사실을 실제로 증명하고 있기 때문이다.

# [ 해야 할 일과 하지 않아야 할 일 ]

자신이 하고 싶은 일을 잘 알고 있는 사람이 얼마나 될까? 일하고 있지만 자신이 정말 원해서 하는 일은 많지 않으리라 생각한다. 원하는 일을 다 하고 살 수 있다면, 아마도 그 사람은 조직 체계의 정점에 도달한 상태인지도 모른다. (물론, 그렇다고 그렇게 마음대로 할 수 있는 사람은 거의 없을 것이다.) 대부분은 그저 시키는 일을 할 뿐이고, 간혹 자신이 먼저 손들어서 지원하기도 한다. 그런 상황이라고 해도 역시 자신이 원하는 일을 찾기는 쉽지 않으며, 대부분의 사람은 정확히 자신이 원하는 것을 말하는 것도 익숙하지 않다.

윗사람들은 인사는 회사의 권한이라고 이야기하면서, 본인의 의사와는 전혀 상관없는 일에 사람을 투입하곤 한다. 하지만, 한가지 잊고 있는 것이 있다면, "인사는 회사의 권한이지만, 이직은 개인의 권한"이라는 점이다. 하지만, 이직은 선택할 수 있는 마지막 수단이 되어야 하지, 항상 꺼내놓고 사용할 수 있는 패는 아니다. 그리고, 이직이 좋은 패가 되기 위해서는 일단 개인의 능력이 중요하다. 개인적인 능력이 없는 사람은 사용할 패도 없어진다. 따라서, 일단은 자신의 능력을 키우는 데 시간과 노력을 들여야 한다. 그리고, 대체로 회사의 일은 개인의 역량보다는 팀이나 조직의 역량을 우선하는 것이 일반적이다.

마음대로 정한 배치를 통해서 나오는 개인의 역량은 과거에 무엇을 했는가가 중요한 것이 아니라, 이제부터 무엇을 할 것인가가 핵심이 된다. 변화는 항상 있지만 자신이 하는 일이 바뀔 때마다 사람은 가장 많은 부담과 스트레스를 받는다. 특히, 나이가 어느 정도 되는 경력 개발자는 새로운 일에 대한 두려움이 있기 마련이다. 이런 일을 자주 겪은 사람은 익숙할지도 모르지만, 한 분야에서만 오랫동안 일해온 사람에게는 마치 회사를 나가라는 소리와 같이 들리기 마련이다. 준비는 언제고 있을 변화를 대비하는 것이지 이미 지나간 일에 대한 것이 아니기에, 이것도 회사 생활을 하는 개인이 항상 준비해야 할 필수 사항이다.

최선을 다해서 하겠다는 말을 들었을 때 가장 먼저 생각나는 단어는 "아마추어"다. 최선은 초보가 하는 행동이고, 새로운 일을 하는 사람이라면 즉흥적으로 대답하지는 않을 것이다. 윗사람에게 "잘 할 수 있겠어?"라는 질문을 듣는다면, "일단 생각해 보겠습니다."라고 다음으로 대답을 미루는 것이 좋다. 나중에 다시 그런 질문을 받는다면, 그때 가서 충분한 생각을 한 후에 대답해도 된다. 이 때의 대답은 자신의 솔직한 의견이 될 것이다. "생각해 봤지만 어려울 것 같습니다."라는 것도 좋은 대답이 될 수 있다. (물론, 그렇다고 바뀌지는 않겠지만) 윗사람은 “확답을 다는 것”으로 자신의 결정을 아래 사람의 책임으로 묶으려고 하는 경향이 있다. 나중에 올 결과에 대한 책임도 결국은 아랫사람의 몫이 될 것이 분명하다.

"해야 할 일은 항상 변화에 대한 준비이고, 하지 않아야 할 일은 준비없는 대응이다." 즉, 어떤 일을 새롭게 시작하기 위해서 평소부터 착실한 준비가 필요하다. 자신이 하던 일만 잘하면 되는 것은 실무자(전문가: Specialist)를 요구하는 시절의 이야기고, 점차 나이가 들어가면 이제는 자신이 모르던 일도 할 수 있도록 대비(Generalist)해야 한다. 책임을 문책 당하는 일에 익숙해 지지 않으려면, 자신의 의사를 명확히 윗사람에게 전달해야 한다. 불가능한 일을 하라는 "인사적인 명령"을 받았을 때는 서슴없이 이직도 고려해 볼 수 있어야 한다. 세상에는 아직도 할 일이 많다고 생각하는 사람이 되도록 꾸준히 준비해야 할 것이다.

# [ 빠른 피드백(Feedback) ]

최근의 개발은 예전에 익숙하게 사용하던 방법론들의 개선을 반영하는 것이 당연하다. 하지만, 실제로 개발팀에서 적용하고 있는 방법론은 아직도 과거를 버리지 못하고 있는 것이 사실이다. 역설적으로 혁신과 창의성을 발휘해야 할 개발팀이 개선된 방법론을 적용하는데 있어서 가장 큰 걸림돌이다. 무엇을 개선하기 위해서는 그것을 직접 몸으로 실행해야 하는 조직이 가장 먼저 바뀌어야 하지만, 과거의 경험에 의지해서 새로운 것을 조직적으로 거부하는 것이 개발 조직이 가지고 있는 변화에 대한 일반적인 태도다.

예전에는 어떤 개발 과제가 결정되면 일정 계획을 작성하고, 마일스톤에 맞게 나와야 할 산출물의 형태를 열거해서 관리하면 전체 과제 진행에 큰 문제가 없었다. 하지만, 최근의 개발은 지속적인 피드백을 중요한 입력으로 다룬다. 즉, 입력이 어떻게 변경되는가에 따라 출력을 실시간으로 생성해 낼 수 있는지가 중요하다. 시장은 지속적으로 변화를 요구하며, 사용자는 구현된 것을 보지 않고서는 자신이 가지고 있는 요구사항을 알지 못한다. 개발이 완료되는 시점에 피드백을 얻을 수 있다면, 이미 그것을 만족시킬 수 없다는 것은 자명하다. 따라서, 지속적인 피드백이 없다면 목표를 달성할 가능성도 희박해지는 것이다.

이와 같은 시장이나 사용자의 요구사항 변화에 맞게 지속해서 산출물을 제시할 수 있어야 하는 것이 최근의 개발 경향이다. 변경에 대해서 언제나 준비가 되어 있어야 하며, 변경이 미치는 영향에 대한 것을 지속해서 확인할 수 있어야 한다. 멈춰진 표적을 맞히는 것이 아니라 시간에 따라 움직이는 목표를 얼마나 잘 따라가는지가 중요하다. 변경에 대한 대비는 다양한 방법으로 갖출 수 있다. 예를 들어, MVC(Model View Control)와 같이 계층화되고 역할이 나누어진 구성을 갖추거나, 각종 패턴 등을 활용한 변경 점에 대한 관리가 있을 수도 있다. 가장 어려운 부분은 그러한 변화가 만들어내는 결과물에 대한 검증이다.

처음 개발 시에 들어간 비용이 100이라고 할 때, 코딩은 대략 50%를 넘지 않을 것이다. 실제로 디버깅과 테스트에 들어가는 비용이 50% 이상일 것이다. 하지만, 기존의 코드를 기반으로 한 2차 개발은 어떻게 될까? 문제는 여기서 시작된다. 새로운 제품을 개발하는 비용은 기존의 비용에 +알파(Alpha)가 될 것이 분명하다. 추가적으로 코딩되는 라인 수는 적지만 코드 간에 발생하는 상호작용은 늘어서 복잡도가 높아진다. 또한, 그러한 상호작용으로 발생하는 버그의 수도 비례해서 늘어날 것이고, 버그를 찾아내는 테스트 시간도 점차 길어질 것이다. 소프트웨어 개발에서 발생하는 비용의 대부분은 인건비이며, 사람을 늘리지 않는다면 개발 시간도 줄일 수 없다. 하지만, 사람 수가 늘어난다고 해서 생산성이 향상되는 것은 아니다. 사람의 능력에는 편차라는 것이 존재하며, 늘어난 사람들의 의사소통을 위한 비용도 함께 증가하기 때문이다.

결과적으로 우리는 끊임없는 작은 피드백이 핵심이 되는 개발 사이클을 유지하여야 한다. 작은 증분(Increments)이 연속적으로 기존의 코드에 추가되고, 이를 검증할 방법을 자동화시켜야 재작업 비용을 절감할 수 있다. 사람을 늘리기는 힘들기에 자동화할 수 있는 것들을 최대한 늘려가야 하고, 작은 단위의 검증을 지속해서 확대해야 한다. 작은 것은 검증하기도 쉽고, 자동화하기도 편하다. 작은 변화는 피드백을 받아야 할 분량도 작다. 만약, 수정해야 한다면 작게 수정하기가 훨씬 쉽다. 작은 체크인이 코드 리뷰나 테스트 케이스를 작성하는 데도 유리하며, 지속해서 빌드(Build)될 때 컴파일 오류나 테스트 오류를 찾기도 편하다. 따라서, 작게 유지하는데 필요한 노력의 비용은 나중에 모아서 한번에 크게 내야 할 재작업 비용보다 훨씬 저렴할 것이다.

# [ 테스트 자동화 ]

테스트는 테스터의 몫이라고 이야기한다면 이미 과제는 잘못되고 있을 가능성이 높다. 테스트는 개발자의 몫이 되어야 한다. 점차 테스터가 했던 일의 일부를 "Left-Shift(좌로 이동?)"라고 일컫는 구현 단계에서 개발자들이 해야 하는 업무로 바꾸고 있다. 물론 이렇게 이야기하면 개발자가 해야 할 일이 늘었다고 생각할 수도 있겠으나, 사실 개발자의 업무는 늘어난 것이 아니다. 오히려 줄어들게 만들 기회를 얻었다는 것이 더 정확하다. 개발자가 담당하는 테스트는 자신이 구현한 코드가 정확히 동작한다는 것을 보장하기 위한 것이며, 테스터가 담당하는 테스트는 기능이 목적에 맞게 구현되었다는 것을 확인하는 것이다.

테스트는 모든 변경에 대해서 동일하게 실행되어야 한다. 기존에 만들어둔 자동화된 테스트가 있다면, 그것도 반드시 새로 만들어지 테스트와 함께 실행되어야 한다. 소프트웨어는 고도로 복잡한 두뇌 활동의 결과물이므로, 변경의 영향이 어떻게 나타날지 예상하기 어렵기 때문이다. 철저한 테스트 외에 달리 해결할 방법이 없다. 이러한 테스트를 수동으로 해야 한다면 무한정 많은 인적 자원을 투입해야 하기에, 테스트의 자동화는 반드시 이루어져야 할 개발 활동이 되는 것이다. 개발자가 자신이 만든 코드의 테스트를 자동화시킬 수 있는 방법을 모른다면, 그것으로 인해서 발생하는 일정 지연의 책임은 오로지 개발자의 몫이 되고만다.

테스트 자동화에서 가장 쉬운 부분은 아마도 외부 시스템과의 상호 연동(Interaction)일 것이다. 정해진 절차와 방법으로 외부의 시스템과 서로 통신한다면, 일차적으로 정해진 프로토콜을 이용해서 순차적으로 테스트를 실행하면 될 것이다. 이것은 분리된 테스트 스크립를 실행할 수 있는 코드의 개발을 의미한다. 즉, 외부 시스템과 주고받는 것들을 순서별로 정리해서 적절히 입력으로 전달해 주면서 그 반응을 살펴보고 참인지 거짓인지 구분하면 된다. 물론, 제품의 품질을 보는 것은 상당히 코딩하기 어렵다. 예를 들어, 주어지는 입력에 대해서 표시되는 사진 이미지의 품질을 알기 위해서는 측정 방법을 정의하고 비교할 수 로직을 구현해야 한다. 따라서, 이런 부분은 결국 사람의 개입이 필요할 것이다. 하지만, 출력되는 값이 참 혹은 거짓을 구분할 수 있는 것이라면 크게 어렵지 않게 구현할 수 있을 것이다.

개발자 테스트의 핵심은 역시 "단위 테스트(Unit Test)"에 있다. 하나의 함수를 최소 실행 단위라고 했을 때 함수마다 독립된 테스트가 가능해야 한다. 따라서, 함수를 분리해서 독립적으로 테스트하기 위해서는 각각의 함수를 별도의 실행 프로그램으로 만들 수 있어야 하며, 또한 각각의 분리된 실행 가능한 함수들은 모아서 한 번에 자동으로 실행할 수 있어야 한다. 개발자가 테스트할 때 막히는 부분이 바로 이곳이다. 이것을 넘어서야만 앞으로 더 나아갈 수 있다. 이를 위해서는 다시 원초적인 문제로 돌아와서 자신의 코드를 봐야 한다. 어떤 코드가 테스트하기 쉬운 코드인지를 이해하고 그것에 맞게 자신의 코드를 만들어야 하기 때문이다.

라인 수가 적은 코드는 테스트하기 쉽다. 함수 인자가 적은 코드는 테스트 하기쉽다. 내부에서 한 가지 일만 수행하는 코드는 테스트하기 쉽다. 함수가 다른 코드에 의존적이지 않을수록 테스트하기 쉽다. 따라서, 결론은 깔끔한 코드("Clean Code")를 짜야 한다는 것이다. 자동화의 첫 걸음은 테스트하기 쉬운 코드를 만들어내는 것부터 시작해야 하고, 테스트를 실행 가능한 코드로 만들도록 해야 한다. 어느 부분이 테스트가 안 되는지 파악하기 위해서 범위 검증("Code Coverage") 도구를 사용해야 하며, 독립적이고 쉬운 테스트를 만들기 위해서 테스트 프레임워크를 자신이 사용하는 프로그래밍 언어별로 익혀두는 것이 효과적이다. IDE(Integrated Development Environment)로 되어있는 테스트 환경보다는 스크립트(Script) 언어로 실행될 수 있도록 만들어 두는 것이, 나중에 코드의 통합과 빌드를 위해서 유리하다.

# [ 유지보수(Maintenance)에 대해서 ]

소프트웨어 개발자에서 필요한 스펙(Spec.) 혹은 요구사항(Requirement)들을 정리하면, 기능적인 것과 기능 외적인 것으로 나누어지는 것은 이미 잘 알고 있을 것이다. 이 중에서도 특히 소프트웨어의 구조에 영향을 주는 부분은 기능 외적으로 있어야 할 것들이다. 왜냐하면, 기능을 이루는 요소는 기능 외적인 요소에 의존해서 구현되기 때문이다. 기능 외적인 구현에 영향을 주는 요소를 비기능 요구사항이라고 하며, 개발자들이 가장 우선시하는 것은 성능과 자원의 사용량이다. 비기능 요구사항에는 여러가지 있지만, 특정 요소만 강조하게 되면 잘못된 코딩으로 이어지는 경우가 종종 있다.

성능을 지나치게 의식하며 코드를 개발자의 편의로 짜는 경향이 강해진다. 중요한 것은 구조를 먼저 만들고 최적화를 통해서 성능을 개선하는 방향으로 가야 하지만, 최적화를 이유로 구조를 무시하는 경향이 생긴다. 성능은 기능 외적인 요구사항 일부이지 전부가 아니다. 사용 편의성, 변경 용이성, 보안성, 유지보수성 등등 다양한 것들이 기능 외적인 부분에 속한다. 이 중에서도 특히 개발자에게 중요한 것은 유지 보수성을 항상 챙기는 것이다. 유지 보수성은 새로운 기능의 추가가 얼마나 쉬운가와 문제가 발생했을 때 얼마나 쉽게 대처할 수 있는가를 보는 것이다.

유지보수는 전체 소프트웨어 생명주기에서 가장 비용이 많이 드는 것으로 알려져 있다. 대략 60% 이상의 비용이 유지보수와 관련된 활동에 투입된다고 한다. 처음 개발 시에 40%의 비용이 든다면, 이 비용은 거의 고정적으로 들어가는 비용일 것이다. 문제는 유지보수가 힘든 코드를 만들면, 얼마나 큰 비용이 추가적으로 발생할지 아무도 알 수 없다는 점이다. 비록 신제품의 개발에 비용 부분이 많이 치우쳐 보이게 되지만, 내부적으로 보면 개발자의 일부는 유지보수와 끝없이 싸우고 있는 게 현실이다. 개발에 투자되는 비용은 표면적으로 과제를 시작할 때 추정을 하지만, 개발이 완료된 후에 추가적으로 들어가는 비용에 대해서는 아무도 모른다.

과제의 시작과 함께 생각할 것은 유지보수에 대한 것을 어떻게 할 것인가를 구현에 반영하는 일이다. 유지보수란 변경을 발생시키는 일이기에 당연히 변경에 대한 대비가 주제가 될 것이다. 변경에 대한 대비의 핵심은 회귀 테스트(Regression Test)를 얼마나 효율적으로 수행할 수 있는가로 귀결된다. 코딩에서는 변경 가능성이 있는 부분에 대해서 구체적인 것에 의존적이지 않은 코드를 만드는 것과 추상화를 위한 인터페이스 정의하는 것이다. 추상화는 복잡성을 효과적으로 다루는 기술이며, 변경이 제공되는 인터페이스 너머에 존재하도록 만들어 영향도를 최소화시킨다.

소프트웨어 개발자는 해야 할 일이 많은 사람이다. 자신이 만든 결과물에 대해서 언제 끝날지도 모를 때까지 지속해서 유지보수를 해야 한다. 심지어 다니던 회사를 다른 곳으로 옮겨도 연락이 오기도 한다. 어떤 경우에는 지구 반대편에 있는 사람이 이메일로 연락해 오는 경우도 있으며, 이미 몇 년 전에 했던 코드에 대해서 다시 꺼내서 보는 경우도 있다. 따라서, "깨끗한 코드(Clean Code)"를 만들고, 변경을 검증할수 있는 “동작하는 테스트”를 만들어야 한다. 그것도 부족하다면 주석이나 설계문서, 프로그래머 가이드, 설치 문서 등등 많은 것들로 보완해야 한다. 자신이 과거에 했던 일을 끊어낼 수 있어야 새로운 일에 몰두할 수 있게 된다.

# [ 버그와 생산성에 대해서 ]

개발의 후반부는 대부분 테스트와 디버깅(Debugging)이다. 버그가 많이 발견될수록 재작업 시간이 더 길어지는 것은 당연하다. 대부분의 소프트웨어 개발이 시간적인 여유가 없어지고, 일정 지연을 발생시키는 이유는 버그의 발견이 과제의 후반에 집중되어 있기 때문이다. 통합이 늦어지면 버그도 늦게 발견된다. 통합하기 전에는 모듈 단위로 자신의 코드를 검증할 수 있지만, 통합된 이후에는 전체적인 관점에서 테스트를 진행할 수 있게된다. 성능 문제로 과제 일정이 늘어지거나 상품기획이나 마케팅, 혹은 더 높은 임원으로부터 질책을 받기 시작하는 것도, 늦어진 통합과 끝없이 발견되는 버그에 원인이 있다.

원인을 알면 그것을 고치는 방법도 알게 된다. 첫 번째로 해야 할 일은 과제의 시작과 함께 항상 통합되고 실행 가능한 소프트웨어 버전을 만드는 일이다. 두 번째는 버그의 발견을 과제 시작과 동시에 함께 하는 것이다. 즉, 테스트 일부를 "좌이동(Left-Shift)"시켜 개발에서 직접 수행하는 것이다. 개발에서 수행되어야 할 테스트는 인수 테스트를 제외한 부분이며, 가능한 자동화를 해야 한다. 자동화를 할 수 없는 부분까지 개발에서 담당할 수는 없다. 테스터를 옆에 두고 일하면 되지 않느냐라고 이야기 하지만, 이것은 개발자의 집중력을 분산시킬 우려가 있기에 권장하고 싶지는 않다. 오히려 개발자가 자동화된 테스트를 만들 시간을 주고, 테스터는 구현된 기능을 위주로 테스트를 자동화할 수 있는 방법을 찾는 것이 좋다.

테스터가 할 일을 개발자가 한다고 이상하게 생각하는 사람도 있을지 모르겠지만, 사실 코드 수준에서 자동화된 테스트는 개발자가 하는 것이 옳다. 만약 그것도 여의치 않다면, 테스터를 옆에 두고 자동화 테스트 스크립트를 만들라고 하는 것이 더 옳을 것이다. 어쨌든 기본적으로는 코드를 세밀하게 테스트하는 것은 개발자의 몫이다. 개발자 업무가 늘었다고 불평한다면, 개발 후반부의 스트레스, 온갖 욕과 촉박한 일정으로 인해서 좌절하는 상황을 미리 제거할 수 있다는 사실을 알려주어야 한다. 외적으로는 과제의 이해관계자가 구현 진척도가 늦다고 이야기할 수 있는데, 이 부분도 테스트 단계에서 예측 가능성을 높이는 활동이라는 점을 주지시켜야 한다. (하지만, 그렇다고 쉽지는 않다. 설득하지 못하면 하라는 대로 하는 척이라도 해야 한다.)

개발 생산성을 유지하기 위해서는 하드웨어 개발에서 사용하는 방법을 차용한다. 즉, 부품을 구성하는 단위를 테스트하고, 부품을 테스트하고, 부품으로 구성된 모듈을 테스트하고, 모듈로 구성된 시스템을 테스트한다. 버그를 조금이라도 줄이기 위해서는 단계별로 테스트를 철저히 하는 수밖에 없다. 비록 그런 것으로 인해서 전체 일정에서 구현이 지연되더라도 결국에는 제품의 일정은 맞출 수 있을 가능성이 높아지기 때문이다. 최소한 무리하게 당겨진 일정에서 얼마나 과제 지연이 될 것인가에 대한 논리적인 지표라도 제공해 줄 수 있다. 아무런 근거도 없는 예측 보다는 테스트가 된 부분이 얼마고, 그렇지 못한 부분에 대해서는 어떻게 테스트할 것이며, 테스트 한 결과가 어떻게 나오는지 보여주는 것이 대략적인 예측을 가능하게 만들어 줄 것이다.

테스트가 개발에서 자동화되지 못할 때는 무조건 테스터가 열심히 자주 테스트할 수밖에 없다. 사실 이것은 사람에 의존적인 일이라 얼마나 걸릴지 예측한다는 것 자체가 아이러니한 일이다. 좋은 테스터는 버그를 잘 찾아내 주고, 그렇지 못한 테스터는 어디에 문제가 있을 것인가를 예측하지도 못한다. 오늘은 이곳을 중점적으로 하고, 내일은 다시 다른 부분을 중점적으로 하는 말 그대로 막무가내 주먹구구식으로 일할 것이다. 이렇게 해서는 언제 어떤 오류가 발생할지를 예상하기란 불가능하다. 테스트 케이스를 늘리면 되지 않겠냐고 생각하는 사람들도 있는데, 시스템은 다루어야 할 파라미터가 무한 조합에 가깝다. 따라서, 테스트 케이스를 늘리는데도 한계가 있고, 그것을 테스트하는 일정과 비용도 만만한 일이 결코 아니다. 제대로 테스트 케이스를 늘리는 것도 일이지만, 그런 테스트 케이스를 테스트하는 것도 비용만 추가할 뿐이다.

과제 후반부에 집중된 테스트로는 버그를 사전에 차단하지 못한다. 문제를 만들고 문제를 다시 찾아서 해결하는 것은 낭비일 뿐이다. 그리고, 그 낭비는 사전에 차단하는 비용보다 훨씬 더 비싸다. 문제는 발생하는 순간 없애야 하며, 코딩이 문제를 만든다면 당연히 테스트도 코딩의 과정으로 생각해야 한다. 가능한 빨리 통합할 수 있는 방법으로 코드를 개발해야 하며, 가능한 빨리 테스트할 수 있도록 만들어야 한다. 사람이 테스트하는 것은 낭비이며, 개발자는 자신이 작성한 코드에 대해서 가능한 많은 자동화된 테스트를 실행할 수 있어야 한다. 버그를 사전에 차단하는 비용은 일의 일부로 생각해야 하며, 비록 과제 진척이 느려지는 것처럼 보일지라도 그것이 더 빨리 가는 지혜다.

# [ TDD(Test-Driven Development)에 대해-02 ]

테스트 주도 개발(TDD: Test Driven Development)는 이미 충분히 성숙한 환경을 가지고 있지만, Java 등과 같은 고수준의 객체지향 언어를 이용한 개발에 치중하고 있는 것이 사실이다. C나 C++등을 개발 언어를 사용하는 임베디드 시스템을 만드는 개발자들은 아직 그런 수준에서 이야기를 공부하고 있는 수준이며, 개발에 적용하는 것을 계속 미루고 있다. 이미 개발된 제품의 코드를 다시 개발하는 것은 어찌 보면 투자에 비해서 얻는 것이 적을 수 있다는 생각과 잘 돌아가는 코드를 괜히 건드려서 문제를 일으키고 싶은 생각이 없기 때문이다.

사실 어찌 보면 당연한 일이다. 자신이 개발한 코드도 아닌데, 그것을 애써 고쳐서 문제를 만들고 싶지도 않고, 지금도 바빠서 야근에 주말 특근까지 하는 현실에서 왜 긁어 부스럼을 만들겠는가? 하지만, 잠시만 시간을 내서 문제를 들여다보면, TDD가 아니더라도 뭔가 새로운 개발 방법이 필요하다. C언어로 개발하던 시스템을 C++ 언어로 바꾼다고 해서 코드가 객체 지향적으로 바뀌지는 않는다. 객체 지향적인 사고방식이 모자란 상태로 C++언어의 특징 만을 이용했을 뿐이므로, 코드에 대한 재활용이나 정보은닉(Information Hiding), 이해의 분리(Separation of Concern), 디자인 패턴(Design Pattern) 등등에 대한 것과 리팩토링(Refactoring)도 부족할 것이다.

TDD로 가기 위해서는 개발자의 역량 향상이 필수다. 지금까지 그런 부분들보다는 단순히 UML을 이용해서 디자인 문서를 쓰는 일과 회사에서 만든 프로세스를 따라가는 것과 같은 외적인 보여주기에 치중해 왔다. 이런 것들에 치중한 이유는 아무래도 지표(Key Performance Index)에 민감하게 반응하는 성과관리 때문으로 보인다. 프로세스나 감사(Audit)를 담당하는 조직은 하는 일이 그런 지표를 관리해서 회사의 프로세스가 향상되고 있다는 것을 보여주어야 하고, 이것을 KPI의 잣대로 들이대는 동안 책임을 맡은 관리자들은 실무 개발자에게 공포에 가까울 정도로 압력을 가하기 때문이다. 따라서, 개발자는 최소한의 방어로 이런 지표들이 잘 나오도록 야합을 하거나, 혹은 그럴듯하게 만드는 최소한의 노력을 해왔다.

TDD는 실무 소프트웨어 개발자의 수준이 높아야 가능하다. 최소한 10명의 팀원이 있다고 가정할 때, 2~3명 정도는 TDD에 대해서 잘 알고 있어야 하며, 단위 테스트(Unit Test)를 코딩에서 활용해 본 경험과 리팩토링, 객체지향 분석 및 설계를 해본 경험이 충분히 있는 인력이어야 한다. 그 나머지 개발인력들은 그런 사람들을 리더로 소규모 조직으로 나누어서 각각의 모듈을 개발하는 데 투입되어야 한다. 하지만, 현실은 어떤가? 개발자가 가장 약한 부분이 자신이 만든 코드에 대한 테스트를 하지 못한다는 점이다. 즉, 테스트를 위주(“Test-Driven”)로 개발해야 하지만, 테스트를 어떻게 해야 하는지 모른다는 점이다. 또한, 객체지향 언어를 사용하지만 디자인 패턴과 같은 중복성을 줄이는 기법에 대해서 무지하며, 개발환경에 대해 개선을 할 줄 모르고 예전에 사용하던 것에만 사로잡혀 있다.

TDD를 하기 위해서 개발자가 가장 먼저 해야 할 일은 코딩(Coding)을 제대로 익히는 일이다. 아이러니하게도 개발자가 가장 잘 하는 것으로 생각하는 코딩이 모든 문제의 핵심에 있다는 것이다. 즉, 테스트를 쉽게 할 수 있도록 코딩 해야 한다. 테스트를 쉽게 할 수 있는 코딩은 한 마디로 표현하면 테스트할 부분이 작아야 한다는 말과 같다. 코딩 라인 수가 늘어날수록 테스트해야 할 부분도 늘어나게 된다.

물론 전체적으로 동일한 일을 하는 함수의 경우 절대적인 크기는 거의 비슷할 것으로 생각한다(코딩 스킬에 따라 차이가 날 수도 있지만, 어느 정도 비슷한 라인 수를 가질 것이라는 가정). 하지만 좀 더 작은 함수로 여러 개를 만들어서 일을 처리한다면, 잘게 나누어진 함수들을 테스트하는 것은 큰 함수 하나를 테스트하는 것보다 쉬운 일이다. 따라서, 작은 단일한 일을 수행하는 함수들을 많이 만드는 것이 좋다. 그렇다고 무조건 작은 크기의 함수를 만드는 것이 아니라, 함수 내부의 코드가 동일한 수준에서 추상화 수준을 만족해야 한다.

함수의 파라미터 수는 당연히 없는 게 가장 좋다. 있어야 한다면 최소한으로 해야 한다. 함수가 어떤 값을 리턴(Return)해주는 것이 그 함수가 올바로 수행되었는지를 확인할 수 있는 방법을 제공한다. 함수 내부에서 새롭게 생성되는 상태나 함수가 의존적인 부분은 외부에서 주어지는 것이 좋다. 함수 자체를 테스트하는 것과 함수가 의존하는 부분을 분리해 낼 수 있다면, 함수 자체의 독립적인 테스트가 가능하기 때문이다. (이런 것을 Dependency Injection이라고 표현한다.). 될 수 있으면 하나의 함수는 시스템의 다른 부분에 의존하는 것이 적을수록 좋다. 의존하는 것이 늘어날수록 독립적으로 테스트하기 위해서, 선행 조건을 테스트 코드로 인위적으로 만들어야 하는 부분이 늘어날 것이기 때문이다.

함수의 독립성을 높이기 위해서는 모듈화를 높일 수밖에 없다. 따라서, 모듈화의 근본 개념인 의존성 줄이기(Decoupling)는 반드시 지켜야 할 원칙이다. 중복된 코드가 많아질수록 오류가 있을 확률이 높다는 것은 다 아는 사실이다. 이런 부분들에 대해서 리팩토링을 지속해서 해주어야 테스트가 쉽다. 비슷한 일을 수행하는 부분들은 묶어두어야 한다. 또한, 사용하는 데이터가 동일하다면, 그런 것들을 묶어서 하나의 모듈로 만들어 주어야 한다. 사용되는 데이터와 그것을 이용하는 함수는 가까울수록 좋다. (객체지향 언어라면, 클래스로 묶어서 정보를 제한적으로 사용할 수 있도록 해야 한다.)

기존의 코드가 이미 있다면, 부분적으로 자주 사용되는 것들에 대해서 의존성을 분리해 나가는 것이 좋다. 전체를 한 번에 수정하기에는 쉽지 않은 일이기 때문에, 조금씩 계획을 세우고 지속적으로 추진해야 한다. 시간이란 한정된 자원이기에 무계획으로 개선 활동을 할 수는 없다. 따라서, 가장 효과적인 방법을 찾는 것이 바로 호출이 많이 되는 부분을 개선하는 작업이다. 개선 전과 개선 후 시스템의 안정성을 확인할 수 있는 것도 역시 단위 테스트를 만드는 수밖에 없다. 미리 분리할 고리를 만들었다면, 그 고리에 대한 개선 전략을 세워야 한다. 호출에 사용되는 파라미터들이 어떤 것이 있고, 어떤 값들이 각각의 파라미터에 들어가야 하는지 확인해야 한다. 또한, 함수의 호출결과 어떤 값들이 다시 복귀값으로 나오는지도 확인해야 한다.

기존코드의 내부는 어떤 식으로 바뀌어도 외부와의 인터페이스만 정확히 동일하다면 상관없다. 때로 상위 계층의 개발자가 하위 계층의 개발자에게 기존 코드와 구조를 동일하게 해달라는 요청을 하기도 하는데, 이것은 아무런 근거가 없는 말이다. 서로 계약관계를 확실히 이행만 한다면 인터페이스를 동일하게 가져가면 내부구조는 어떤 식으로 변경되더라도 문제가 없어야 한다. 계약은 이미 인터페이스의 스펙(Spec.)에 다 명시된 사항이며, 내부구조에 대한 계약은 한 적도 없고 할 이유도 없는 것이다. 내부 구조를 외부에 동의를 받아서 구현한다면, 외부로부터 내부 구조가 보호받지 못한다는 말 밖에 되지 않기 때문이다. (어리석고 무지한 사람은 모든 것을 다 잘 알아야 한다고 생각하겠지만, 그런 수준의 코드는 이미 시간이 만료되었다.)

# [ 개발 생산성에 대해서 ]

소프트웨어 개발을 시작할 때 대부분 관리자는 개발자들이 매일 열심히 늦게까지 일하기를 기대한다. 그러한 기대는 팀워크가 어떤가에 좌우되며, 급조된 팀에서는 기대하기가 쉽지 않다. 개발자 개인들은 동기가 없으면 그런 식으로 일하는 것이 사실상 불가능하다. 또한, 출근해서 퇴근까지의 시간이 길다고 해서 생산성이 높다고도 볼 수 없다. 잘 알려진 사실이지만, 개발자 간의 생산성은 개인의 역량에 따라 200% 이상의 차이를 보일 수 있기 때문이다.

어떤 개발자의 경우 8시쯤에 회사에 도착해서 화장실이나 밥 먹으러 식당에 가는 시간을 제외하고 거의 자신의 책상에 앉아서 코딩하는 사람이 있었다. 보통의 경우 초중급 개발자 3명의 몫을 했으며, 시스템의 통합과 타겟(Target)에 올리는 작업, 중요한 시스템의 구조를 정의하고 구현하는 일을 했다. 또한, 주변 동료들의 역량을 향상하기 위해서 필요한 세미나를 진행했으며, 코드 리뷰 및 테스트에도 남다른 감각을 지니고 있는 개발자였다. 퇴근 시간은 항상 6시 정도였고 주말에는 근무하지 않았다. 하지만, 대부분의 경우 자신이 해야 할 일에 대한 것은 일정을 준수하고 있었고, 결과물의 품질도 우수한 편이었다.

위 같은 경우 그 개발자에게 잔업 및 주말 특근을 하라는 말이 의미가 있을까? 자신이 해야 할 일을 일정대로 처리하고 있다면, 사실 그렇게 자신의 시간을 회사를 위해서 더 사용하라고 하는 것은 어찌 보면 낭비에 가깝다. 보통의 경우 밤에 일해야만 한다는 것을 알게 되면 낮에는 집중하지 않게 된다. 주말이나 휴일에 나와서 일해야 한다면, 금요일까지 할 수 있는 일도 주말이나 휴일로 늘어지게 된다. 밤에 집중이 잘 된다고 하는 사람들의 낮 시간을 관찰하면, 대부분 비효율적인 회의와 이런 저런 쓸데없는 일들로 시간이 낭비되고 있는 것을 볼 수 있다. 환경적인 요인도 있고 사람에 의한 요인도 있지만, 어쨌든 집중해서 일하도록 분위기를 제공하지 못한 것은 관리자와 회사의 책임이다. 그런 책임을 개발자에게 전가하는 것은 옳지 않다. 오히려, 실행에 집중할 수 있는 시간을 늘려주는 것이 현명한 선택이다.

늦은 밤까지 일한 사람의 아침 시간은 거의 낭비된다. 제대로 업무에 집중할 수 없을 만큼 피곤함이 지속된다. 커피와 담배 등등의 도움을 받으려고 하지만, 오히려 그런 것은 일시적인 집중에만 도움이 된다. 벼락공부의 효과는 오래가지 않는다는 것은 모두가 아는 사실이다. 하지만, 우리 관리자들은 벼락공부를 일 년 내내 하라고 이야기하고 있다. 과제의 시작부터 끝날 때까지 지속해서 100m 달리기와 같이 최고 속도를 유지하라고 한다. 마라톤에서 요구하는 것과 단거리 경기가 요구하는 것이 다름은 잘 알면서도, 자신의 불안감이 해소되도록 강요에 가까운 요구를 하게 된다.

소프트웨어 개발의 생산성은 어떻게 표현할 수 있을까? 하루에 코딩하는 라인(Line)의 수를 생산성으로 보는 것은 이미 지나간 이야기다. 개발의 난이도에 따라 라인 수는 얼마든지 조작할 수 있기 때문이다. 사용자 인터페이스(UI)와 같은 코드는 많은 라인 수를 차지하지만, 비즈니스(Business)의 핵심 로직이 아니며, 또한 초당 수행되는 횟수도 적다. 디바이스 드라이버(Device Driver)를 개발한다면, 실제 라인 수는 비교적 짧지만 I/O 요청에 따라 수행 횟수는 오히려 더 높을 수 있다. 소프트웨어 생산성은 오히려 하나의 소프트웨어를 개발하는 것으로 측정하기보다는 지속적인 개발에서 얼마나 일정이 효과적으로 단축되고 있는지를 보는 것이 옳다고 생각한다.

처음 개발할 때는 1년이 걸렸다면, 동종의 제품을 추가적인 기능(Feature)를 몇 가지 넣어서 개발할 때 얼마나 개발시간이 단축되는지를 측정해서, 이를 처음에 개발할 때 걸린 시간으로 나누는 것이다. 물론, 투입된 사람도 고려되어야 한다. 투입된 인력에 따라 생산성은 많은 차이를 보일 수 있기 때문이다. 중요한 것은 한 번의 개발로 생산성을 측정하는 것은 의미가 없다는 말이다. 초기 개발에서 만들어진 소스 코드가 제대로 되어있고, 회귀 테스트(Regression Test)를 위한 방안들이 잘 구현되어 있다면, 사실 개발시간은 획기적일 정도로 줄어야 맞다. 기본적으로 SQA(Software Quality Assurance)나 PQA(Product Quality Assurance)와 같은 곳에서 소모되는 시간은 어쩔 수 없다고 하더라도, 기능 구현에 드는 시간과 버그 수는 줄어들어야 한다.

만약 지속해서 일정이 지연되고, 기능 추가에 따른 버그의 수에 변함이 없다면, 이미 잘못된 개발방법을 사용하고 있다고 봐야 한다. 즉, 제대로 된 테스트를 갖추고 있지 못하기에 변경에 대한 검증을 개발에서 수행하지 못한 것이다. 개발은 자신이 한 일에 대한 기본적인 검증을 반드시 해야 한다. 가장 좋은 방법은 역시 테스트 자동화를 이용하는 것이다. 하드웨어와 관련된 동작 검증에 어려움이 있다고 호소한다면, 하드웨어를 시뮬레이션(Simulation)할 수 있는 도구를 만들어야 한다. 예외 상황을 어떻게 검증하냐고 이야기한다면, 그 예외를 일으키는 방법을 소프트웨어적으로 고안해야 한다. 즉, 하드웨어 인터페이스 부분을 갈아탈 수 있는 구조로 만들어 인위적인 테스트 환경을 구성할 수 있어야 한다.

이런 모든 활동이 초기에는 일정을 잡아먹는 부담으로 작용하는 것이 사실이다. 누군가 내가 해야 할 일을 해주기를 바라고, 또한 그런 도구가 구축되면 편할 것이라는 생각은 한다. 하지만, 정작 그런 것을 만들자고 하면 시간이 없다는 말로 주저하는 것도 사실이다. 중요한 것은 초기 일정을 잡아먹더라도 개발의 후반으로 가서는 오히려 일정을 단축하게 해준다는 점이다. 또한, 이렇게 만들어진 도구나 테스트 스크립트(Test Script)도 소스 코드의 재활용과 같이 다시 다른 과제에서도 재활용될 것이기에 그 효과는 지속해서 유지가 될 것이다. 생산성은 소스 코드의 품질과 그것을 증명할 수 있는 도구를 얼마나 잘 활용하는가에 달려있다. 단지 개발자의 시간 활용에만 의존적인 것이 아니다.

야근과 주말 특근이 개발 생산성에 얼마나 도움이 될까? 모든 관리자가 원하는 인재상은 항상 언제 어디서나 찾았을 때 자리를 지키고 있는 개발자다. 자신이 일하는 모든 시간에 질문을 했을 때 재빨리 대답하는 사람이 일을 잘한다고 생각한다. 관리자는 일어서서 주변을 살펴보았들 때 모든 사람이 다 자리에 앉아 있기를 대부분 바란다. 왜 그렇게 많은 사람이 앉아있는 모습을 좋아할까? 사실 이것은 자신의 불안감을 해소할 방법을 찾지 못한 아마추어적인 생각을 하는 관리자들의 자연스러운 강박관념이다.

왜 아마추어적이라고 이야기하는가? 사람들이 앉아서 일하고 있다는 사실 자체 만으로 앞으로 나간다고 생각하기 때문이다. 자리에 있지 않은 사람은 일하지 않는 사람이고, 주말에 나오지 않는 사람은 노는 사람이라는 아주 간단한 이분법적인 생각 때문이다. 매주 주간회의를 통해서 일의 진척도를 파악하는 것도 마찬가지다. 주간회의 보다는 일의 진척도를 파악하기 위해서 과제 관리 시스템을 보는 편이 옳지 않을까? 대쉬보드(Dash Board)를 만들어놓고, 거의 사용하지도 않으면서 지속해서 회의를 소집하는 것은 뭔가? 시스템에 입력하라고 하면서, 프린트로 준비된 자료를 읽는다면, 도대체 왜 과제 관리 시스템을 만들라고 했는지에 대한 이유를 찾을 수 없다.

프로라면 적어도 회사에서 사용하는 각종 시스템들에 대해서는 잘 이해하고 있어야 한다. 시스템에서 파악이 안 되는 것은 이메일을 사용해서 물어보고, 그것도 안 되면 직접 담당자를 찾아가서 물어보면 되지 않을까? 왜 굳이 모든 개발자를 다 모아놓고, 두 시간이나 되는 시간을 같이 낭비하라고 강요하는가? 그 시간에 좀 더 개발하는 편이 훨씬 더 효율적이지 않을까? 야근 시간을 2시간 늘리기보다, 야근하지 않도록 일에 집중할 수 있는 환경과 시스템을 구축하는 것이 인건비를 훨씬 더 잘 사용하는 방법이라고 생각해본 적이 없는 것이다. 새로운 사람을 뽑자는 말에는 동의하면서, 개발하는 데 필요한 장비를 사는 것에는 인색한 사람도 마찬가지로 아마추어적인 관리자다. 인건비보다는 장비 구매가 훨씬 싸기 때문이다. 30인치 모니터가 100만 원이라고 한다면, 그걸 사지 않아서 낭비되는 개별 개발자의 시간은 한 달에 100만 원 이상이 될 수도 있다. 소프트웨어 개발의 대부분은 남의 코드를 읽는 것이기 때문이다.

작지만 개발자 편의를 위한 활동들이 지속하여야 하는 것도 그와 같은 생산성의 향상에 필요한 부분이기 때문이다. 집중하는 데 필요하다고 자기 자리는 다 비운 채, 작은 방에 많은 수의 개발자를 넣는 것과 같은 것을 무슨 특별한 배려라고 생각하는 관리자도 있다. 하지만, 조금만 이야기가 들려도 그 방의 모든 사람이 다 알게 되는 답답하고 좁은 곳에서 일하는 것은 인내심을 가지고 집중하도록 훈련하는 것 이외에는 별다른 도움이 되지 않는다.

야근과 주말 특근보다는 자동화 테스트를 만들어 밤이나 주말에 돌리는 편이 훨씬 더 효과적이지 않을까? 테스트를 많이 하라고 해서 모든 개발자에게 테스터 들이 해야 할 일을 할당하는 아둔한 머리가지고는 제대로 된 생산성을 운운해선 안 될 것이다. 모르면 그냥 아는 사람을 잘 찾아서 맡기기라도 하는 편이 개발자들에게 도움이 될 것이다. 무지함은 자신도 힘들겠지만 그런 관리자 밑에서 일하는 사람들의 희망도 좌절시킨다는 것을 알아야 할 것이다. 개발자가 해야할 테스트는 테스터가 해야할 테스트와 엄연히 다르다. 그리고, 그런 개발자들이 해야할 테스트를 하기 위해서는 테스트 자체를 만드는 것을 일로 보아야만(취급해야만) 가능하다.

소프트웨어 개발은 고도의 정신 집중을 요구하는 분야다. 보통의 경우 정신을 집중하는데 필요한 시간이 대략 15분 정도라고 한다. 즉, 50분 동안 교육하는 경우 초반 15분은 집중하는 데 사용된다고 본다. 일반적인 사람의 경우 20분 이상 집중 상태를 유지하는 것이 어렵다고 한다. 따라서, 15분의 집중을 위한 시간과 20분의 집중을 포함하면 대략 35분이 걸린다. 다시 15분간의 혼돈된 시간을 보내고 나면, 1시간의 교육이 끝나게 된다. 이것을 유추해 볼 때, 정작 우리가 제대로 두뇌를 이용하는 시간은 1시간 중에서(10분의 휴식 시간을 포함해서), 20분 정도(1/3) 밖에 되지 않는다.

하루에 8시간을 근무할 경우, 집중할 수 있는 시간은 2시간에서 3시간 사이다. 나머지 시간은 어떤 식으로든 낭비가 발생하고 있다. 그렇다고 야근을 시킨다면 다음 날 아침의 시간에 대한 낭비가 발생하게 된다. 왜냐하면, 사람은 어떤 식으로든 부족한 부분을 메우기 위해서 무의식 중에 방어적으로 변하기 때문이다. 집중에 필요한 15분을 확보한다고 해서 그 후에 발생하는 20분의 완전한 몰입이 보장된다고도 할 수 없다. 우리가 일하는 사무실 공간은 항상 몰입에 방해가 되는 것으로 가득하기 때문이다. 따라서, 우리가 실제 하루에 일할 수 있는 시간은 그리 많지 않다.

대략 1년에 한 명의 소프트웨어 개발자가 20,000라인 정도의 코드를 짤 수 있다고 한다. (이정도만 한 사람이 작성할 수 있다면 하지 못할 일이 거의 없다고 봐야한다. 대부분은 그것보다 훨씬 못할수도 있다.) 물론 여기에는 테스트와 디버깅 시간도 포함된다. 우리가 일하는 날짜가 1년에 대략 250일(계산의 편의를 위해서 그냥 250일로 하자. 300일은 사실 토, 일요일만 제외해도 모자란다.)로 했을 때, 한 사람이 1일당 80라인 정도를 작성한다고 생각할 수 있다. 일단은 좀 많이 잡아서 하루에 100라인이라고 하자. 실제로는 아마 대부분 개발자가 하루에 100라인을 일 년 내내 코딩하는 것은 어려울 것이다. 대체로 100라인보다는 20라인 정도가 될 것이다. 왜냐면 기존에 이미 개발된 코드를 활용하고, 그것에 몇 가지 기능을 덧붙이는 코딩을 하기 때문이다. 평균적으로 20라인 정도를 작성한다고 해도 1년에 250일을 일 한다면 5,000라인에 가깝다.

물론 개인적인 생각이 들어간 부분이 있기는 하지만, 대부분 개발자는 1년에 5,000라인을 개발하는 것도 쉬운 일이 아닐 것이다. (실제로 자신이 작성한 코드의 크기를 보기 바란다.). 더군다나 크기가 큰 소프트웨어를 개발하는 것은 복잡도와 의존성이 상대적으로 크기에, 조금만 코드에 수정을 가하더라도 오류를 일으킬 가능성이 높다. 따라서, 더 높은 복잡도와 의존성은 더 높은 집중력이 요구된다고 할 수 있다. 하지만, 오늘 20라인의 코드에 집중해서 노력한다면 어떨까? 혹은 20라인의 코드에 대해서 테스트를 위한 코드 20라인을 더한다면 어떨까? 하루 하루 코딩한 결과를 그 날이 지나기 전에 완벽하게 확인할 수 있다면, 적어도 테스트가 된 20라인의 코드는 매일 생산할 수 있게 된다.

이러한 노력이 쌓인다면, 적어도 집중에 필요한 15분과 집중력을 발휘하기 힘든 20분의 시간이 헛되이 다음날로 전달되지는 않을 것이다. 학교 다닐 때 공부는 머리로 하는 게 아니고 체력으로 한다는 말이 있지만, 말 그대로 이것은 뒤를 돌아보지 않는 종착점이 있을 때 가능한 이야기다. 한 번의 고비를 넘기면 새로운 세상이 보이는 게 아니라, 다음에 더 높은 산이 있는 것이 소프트웨어 개발에서는 일반적이다. 따라서, 15분이라는 시간은 미래를 위한 투자로 남겨두어야 한다. 그 시간만큼은 온전히 자신이 많든 코드에 대해서 완전히 테스트하는데 투자해야 한다. 이런 식으로 꾸준히 15분의 시간을 투자하게 된다면, 생산성은 작은 노력의 총합보다 더 커지게 될 것이 분명하다.

결국, 소프트웨어 개발자의 생산성은 어떤 코딩을 하느냐에 좌우된다고 볼 수 있다. 파일 I/O로 따지자면, 소프트웨어 개발자가 하는 일은 대부분 읽기(Read)에 속하지, 쓰기(Write)는 자주 있는 것이 아니다. 이미 쌓여있는 유산을 바탕으로 새로운 것을 추가하는 일이다. 쌓아 올리기 위해서는 어떤 부분을 어떻게 맞추어야 할 것인가를 고민해야 하고, 무너질 가능성이 있는 부분은 새로운 수정을 해야 한다. 수정이나 추가는 이미 있는 것들을 잘 이해해야 가능한 일이기에, 코딩 결과물이 얼마나 이해하기 쉽냐에 좌우된다고 볼 수 있다.

기본적으로 사용하는 언어에 대한 기초적인 코딩 경험을 가지고 있는 사람이 보았을 때, “쉽게 이해가 가는 코드가 좋은 코드”이다. 한참을 생각하고 이런 저런 값을 넣어서 가상으로 시뮬레이션(Simulation) 해야지만 이해가 가는 코드라면, 고치기도 어렵고 새로운 기능을 추가하기도 쉽지 않을 것이다. 당연히 오랜 시간을 일해도 제대로 된 코드가 나오기가 힘들다. 따라서, 이런 것을 보았을 때 초급 개발자가 익혀야 할 기술은 읽기 좋은 코드를 만드는 방법일 것이다. 읽기 좋은 코드를 만드는 것은 좋은 이름을 선택하는 것에서 시작하며, 추상적인 것에서 구체적인 것으로 어떻게 전개하는가에 달려있다. 즉, 논리적인 사고를 점차 추상적인 부분에서 시작해서 구체적인 부분으로 세밀화 해 나가는 것이다.

좋은 코드를 뒷받침하기 위해서는 올바른 테스트해야 한다. “소프트웨어 아키텍처는 시스템을 구성하는 요소와 그들 간의 관계를 설정하는 활동”이라고들 정의하는데, 구성 요소들은 대부분 기능적인 부분에서 유추가 될 수 있고 관계는 비기능적인 부분에서 나온다. 특히 비기능적인 부분에 대해서 "성능"만을 강조하면, 좋은 구조를 생각하기 보다는 효율만을 지나치게 강조하는 측면이 있어서 좋은 코드로 잘 연결되지 않는다. 변경 용이성, 이식성, 확장성 등등을 다 고려해야지만 좋은 구조가 나올 수 있다. 품질 적인 측면에서는 테스트 용이성("Testability")도 충분히 고려해야 한다. 개인적으로 소프트웨어 개발 생산성에 가장 큰 영향을 미치는 부분이 이러한 테스트에 관련된 것으로 보고 있는데, 이유는 테스트를 통합이 이루어진 후에 하는 경우 얼마나 많은 버그가 시스템 내에 존재할지 예상할 수 없다는 점과 어떤 부분에서 버그를 유발하는지 파악하기 힘들기 때문이다.

어쨌든 가장 중요한 것은 소프트웨어 개발자의 코딩 역량이다. 사실 실무를 잘한다고 생각하는 대부분 사람이 코딩을 자기 나름의 기준으로 하는 것 때문에 곤란한 경우에 처하는 경우가 많다. 많은 코드를 만들어 내고 시스템 대부분의 중요한 구조를 코딩한 개발자도 자신이 만든 늪에서 허덕이며 야근을 밥 먹듯이 한다. 이것은 제대로 된 코딩보다는 일단 동작하도록 만드는 코딩에 익숙하다는 말 밖에 되지 않는다. 이때는 오히려 코딩에서 잠시 손을 놓고 자신이 만든 코드의 기본적인 부분들이 잘못된 것이 아닌지 고민해야 한다. 앞에서 이야기한 읽기 쉽고 이해하기 쉬운 코딩에서 멀어지지 않았는지 심각하게 생각해 봐야 할 때다. 자신이 만든 코드는 자신의 것이 아니며, 과제에 이해 관계를 가진 모든 사람의 것이라는 것을 잊어선 안 될 것이다.

소프트웨어 개발자의 생산성을 이야기할 때 얼마나 오랫 동안 일하는가와 얼마나 많은 코드를 작성하는가로 따지는 것은 웬만한 초보 관리자들은 다 한다. 하지만, 이렇게 만든 생산성 지표가 과연 의미가 있을까? 실제로 과제를 수행하는 전체 시간으로 따져봤을 때, 한 과제만 실행하는 것과 여러 과제를 다 총괄해서 보면 이야기는 달라진다. 대부분의 근시안적인 관리는 각각의 과제의 성공만을 따지지, 전체적인 회사의 영속성과 관련된 전체 과제의 생산성 측면은 보지 않는다. 문제는 여기에 있다.

과제의 비용은 (개발비용 + 개발비에 속하지 않지만 개발에 필요한 비용 + 유지보수 비용)으로 볼 수 있다. 이 중에서 직접적인 개발 비용은 항상 과제를 시작하는 시점에서 논의 대상이 되지만, 개발이 진행 중인 상황에서 투자되는 비용에 대해서는 별도로 논의한다. 예를 들어, 야근이나 주말 특근 등에 사용하는 비용은 과제 시작 전에 논의되지 않는다. 하지만, 이 부분에 대해서 인당 1개월당 50~60만 원이 추가로 들어간다면, 10명이 하는 과제일 경우 1달에 600만 원 이상의 개발비가 추가된다. 또한, 여러 명이 할 경우 의사소통에 필요한 비용도 만만찮게 차지하게 되며, 개발자가 하루 동안 실제 과제에 투자하는 시간은 줄어들고, 다른 일에 뺏았기는 시간은 늘어나게 된다.

과제를 하는 중에 이전 과제에서 발생한 문제를 해결하라는 압력이 들어오면, 그것도 과제에 대한 비용을 추가시키는 요소다. 과제의 일정에 영향을 주기에 초기에 생각했던 과제 비용은 더 늘어나게 된다. 따라서, 이런 부분들이 지속해서 추가되면 과제에 대한 비용은 초기에 세웠던 계획과는 전혀 상관없이 무조건 과제를 완료해야만 한다는 압박감만 키우게 된다. 개선에 투자된 비용이 없기에 차기 과제를 하게 될 때 발생하는 비용도 상승하기 마련이다. 하나를 고치면 수많은 버그 속에서 과제 일정 지연은 당연한 것으로 받아들이게 되며 개발자의 사기도 떨어지게 된다. 이런 현상이 반복되면 과제의 일정을 맞추라는 사내 방송의 "구호"성 외침은 드라마 속의 비현실이 되고 만다.

노련한 관리자라면, 과제를 시작할 때 과제 계획에 비용만을 나열하지는 않을 것이다. 개발 방법론과 그것을 적용하기 위한 세부적인 것들을 미리 준비하고, 팀원들에게 훈련시키게 될 것이다. 새로운 개발 방법론을 매번 과제에 다르게 적용하는 것이 아니라, 점진적으로 개선되는 모습을 보이기 위해서 팀원들과 논의하게 될 것이다. 자신이 모르는 것을 모른다고 인정하기는 쉽지 않지만, 팀원이 잘 아는 것까지 자신이 나서서 이래라 저래라 하지는 않을 것이다. (혹은, 부지불식간에 녹여 들도록 만들 것이다.) 유지보수의 중요함을 잘 않다면 그것을 위한 방법을 생각하고 과제 시작부터 팀원들과 실전에서 하나씩 점검해 볼 것이다. 누구나 경험이 완전한 사람은 없다. 하지만, 경험하지 못했다고 부정하는 것은 옳지 않다. 부족하다면 다른 곳에서 찾아서라도 적용해 보려고 노력할 것이다.

오래된 일은 아니지만, 예전에 모 학자가 "권력 이동"이라는 책을 쓴 것을 기억한다. 산업의 발전에 따라 기존에 힘(Power)을 가지고 있던 조직에서, 새로운 형태의 조직으로 역학 관계의 변화가 일어난다는 내용이었던 것으로 알고 있다. 논리적인 비약일지는 모르지만, 소프트웨어 개발에 있어서도 초기의 가내 수공업적으로 만들던 것에서 벗어나 전문화되고 많은 양의 코드를 만드는 것으로의 진화가 있었다. 처음 만들 때는 간단한 목적으로 만들어진 프로그램들이 점차 합쳐지고 변경되면서 스스로가 마치 생명력을 가진 것처럼 새롭게 태어나고 있다.

초급 소프트웨어 개발자일 때는 시키는 일만 열심히 하면 된다. (물론, “열심히”에 대한 것의 정의는 다르겠지만) 중급 정도 되면, 3~6명 정도의 개발자와 같이 일하게 되고, 이때는 자신이 맡은 모듈 수준에서의 결정 권한이 주어진다. 이후 좀 더 고급으로 올라서면 관리적인 업무를 일부 수행하게 되고 시스템 수준에서의 결정을 하게 된다. 이 때는 자신과 같이 일하는 모듈 단위의 3~6명 정도의 동료들도 같이 일한다. 문제는 이때 발생한다. 초급 수준과 중급에서 익혔던 소프트웨어 개발에 대한 지식이 많은 양의 프로그램을 작성하는데 필요한 지식과는 다르다는 점이다. 자신이 하던 방식을 고수하게 될 때 이미 그 역할을 맡은 모듈의 리더나 항상 새로운 지식으로 무장된 초급 개발자들가 의견 충돌이 일어나게 된다.

이때는 오히려 자신이 기존에 하던 일들을 하나씩 다른 사람에게 이관해야 할 시기다. 물론, 그 전에 그 일을 맡게 될 사람의 역량도 충분히 검토해 보아야 하고, 역량을 개발할 기회도 충분히 주어야 한다. 모든 것을 자신이 다 하려고 하면, 중간관리자들의 역할을 뺏는 상황이 되고, 사실상 계층구조와 같은 조직보다는 일인이 독재하는 체계로 가고 만다. 물론 능력이 출중한 사람은 그렇게 할 수 있을지도 모르지만, 과제가 1년에 끝날 일이 아니라면 중간 관리자의 역량을 키우는 것이 좋다. 모든 세세한 부분을 다 파악하고자 한다면, 정작 자신이 현재의 자리에서 해야 할 일은 하지 못 하게 된다. 관리자가 되고 싶지 않다면 차라리 개발자로 남는 것이 좋다. 물론, 그냥 개발자로 남아선 안 되고 핵심 개발자가 되어야 할 것이다.

그럼 과연 핵심 개발자가 되는 것이 쉬운 일일까? 아마도 소프트웨어 개발에 발을 들여놓은 사람들은 그것이 힘들다는 것을 잘 알 것이다. 즉, 소프트웨어 개발에 대한 모든 부분을 잘 이해해야 하고, 자신이 속한 조직의 개발 문화를 만들어가는 역할을 해야 하기 때문에 일반적인 개발자와는 다르다. 지식산업의 가장 상층부는 지식이 돈이 되게 하는 사람들로 채워지고, 그 돈 되는 방법을 제대로 만들 수 있는 방법을 아는 사람들이 핵심 개발자다. 따라서, 지금까지 자신이 하는 일만을 신경 쓰던 사람들도 핵심 개발자가 되기 위해서는 돈을 만드는 전 과정을 이해해야 하고, 상충한 의견을 시스템에 잘 반영할 수 있도록 조율하고 만들어 나가야 한다. 이 또한 역할의 변경에 따른 기존 책임의 분산을 통해서 자신의 책임을 더 추상적인 일에 투입해야 한다는 의미가 된다.

# [ 중요한 일과 급한 일 ]

소프트웨어 개발자가 시간이 없다고 이야기할 때, 항상 문제는 중요한 일과 급한 일을 구분하지 못한다는 것에 원인이 있다. 물론 모든 원인의 제공자는 개발자라기 보다는 명령을 내리는 사람에게 있지만, 중요한 일은 과제가 성공적으로 수행되기 위해서 반드시 실행되어야 하는 것들이다. 급한 일은 대부분 비동기적으로 하달되는 명령이나 난데없이 끼어드는 일로, 과제의 성공과는 거리가 먼 경우가 많다. 급한 일은 지금 바로 해결해야 할 일이며, 그렇지 않을 경우 심각한 위험에 노출될 가능성이 있는 일이다. 그럼 앞에서 이야기한 중요한 일과 급한일 두 가지 일이 함께 주어졌을 때 어떤 일을 먼저 할 것인가?

법보다 주먹이 가깝다는 말이 있다. 사실 중요한 일을 먼저 하라고 이야기는 하지만, 주먹이 날아오는 상황에서는 급한 일을 먼저 하기 마련이다. 급한 일이 발생하는 빈도가 낮다면 크게 과제에 대해서 영향을 주지 않겠지만, 점점 늘어나는 것이 일반적이고 줄어들지는 않는다. 따라서, 이런 상황이라면 과제에 피해가 가는 것은 불을 보듯 명확하다. 또 다른 야근의 불씨를 남기는 것이다. 상품기획이나 마케팅, 소비자 만족(CS: Customer Satisfaction), 기획, 기타 협력 관계에 있는 부서 및 감사실, 상부 관리자, 주간보고, 월간보고, 과제 점검회의, 기획회의, 특허, 각종 T/F등등 지속해서 급한 일을 만들어 내는 곳은 많다.

개발자가 이런 모든 것들을 대응하기는 쉽지 않다. 과제의 모든 부분을 상세히 알고 있는 관리자라면, 그런 요청들에 대해서 혼자서 대응하는 것도 가능하리라. 즉, 코딩도 하고, 문서도 작성하고, 기획도 하고 등등 거의 슈퍼맨적인 역량이 필요할 것이다. 그렇지 않다면 어쩔 수 없이 자신의 팀원에게 업무가 내려가고, 그것을 확인하기 위한 시간도 필요할 것이다. 이런 일이 많으면 많을수록 개발자는 정작 중요한 일을 하는 시간이 줄어들게 된다. 과제에 집중하는 시간도 줄어들고 일정도 지연된다. 그리고, 한참 후에 이런 말을 들을 것이다. "왜 과제가 지연되었어?" 몰라서 묻는지 아니면 다 잘 알지만 그래도 열심히 하라고 이야기하려는 것인지는 모르겠지만, 짜증이 날 수밖에 없다.

가장 이상적인 해결 방법은 급한 일을 줄이는 데 있다. 급한 일도 과제의 일정에 반영되어야 하고(최소한 일정 계획을 수립할 때, 이를 대응하기 위한 일정도 같이 포함해야 한다.), 이런 일들을 개발부서까지 전달하는 것도 자제되어야 한다. 혹은, 그것만을 전담해서 해결할 사람들을 따로 조직으로 만드는 것도 좋다. 급한 일은 급하게 처리해야 하고, 중요한 일을 중요한 일로 처리되어야 하기에 동일한 사람이 해선 안 된다. 분리되지 않는다면 결국 급한 일과 중요한 일은 구분이 되지 않고, 모두 다 우선순위가 높은 일이 되어버린다. 과제가 되어버리고 마는 것이다. 누구도 이런 상황을 달가워할 사람은 없다. 일해도 욕을 먹을 수밖에 없는 구조를 만들면, 누가 일하고 싶어하겠는가? 그냥 급한 일을 요청하는 부서에서 일하고 싶어 할 것이다.

# [ 성공 스토리 ]

소프트웨어를 만들기 위해서는 분석이라는 과정이 반드시 선행되어야 한다. 물론 무엇을 만들 것인가를 알지 못하는 상황에서는 이런 일도 필요 없을 것이다. 만들고자 하는 시스템이 무엇인지 정해진다면, 그것을 만들기 위한 선행 과정으로 분석을 진행해야 한다. 분석(Analysis)은 해야 할 일과 그것에 필요한 자원, 그리고 그것을 하는데 필요한 시간을 정하는 데 반드시 필요하다. 하지만 대부분 과제는 사실 분석이라는 과정 없이 "언제"라는 것만 주어지는 것이 현실이다. 어떤 시스템을 어떻게 만들지를 모르는 상황에서 "날짜"를 못 박아서 소프트웨어 개발자에게는 악몽과 같은 시련을 예약하게 된다.

흔히들 "성공 스토리"에 대해서 이야기하면, "얼마나 많은 시간을 소프트웨어 개발자들이 일했는지"를 말한다. 누군가 얼마만큼의 희생을 통해서 제품이 만들어졌다고 이야기하는 것은 너무나 위험한 발상이다. 사실 그렇게 일하지 않는 개발자를 찾기가 어려운가? 아마도 대부분 개발자는 자기 삶을 희생하면서 일하고 있을 것이다. 주중에 있는 일상적인 야근시간과 주말의 특근은 말 그대로 개인의 희생에 바탕을 둔다. 누군들 자신이 만든 제품이 시장에서 실패하리라는 것을 알고 만들겠는가? 자신의 피와 땀이 묻은 제품이 시장에서 잘 팔리기를 간절히 기원하면서 하루 하루 “자신의 삶과 가족과 함께하는 시간”을 희생하면서 제품에 투입하고 있을 것이다.

문제는 이러한 생각이 너무 일반적으로 퍼져있다는 것이다. 모든 개발자가 그렇게 일하는 것이 마치 인지상정인 것처럼 인식되는 분위기다. 회사는 항상 부족하게 과제를 운영하는 것이 올바른 생각이라고 보이도록 만든다. 과제를 운영하는 부족한 시간은 무엇으로 메울 것인지는 당연히 개발자의 몫이 되고 만다. 부족한 것들은 나중에 시장에서 판매될 제품에도 영향을 주게 되며, “2%의 부족”이 제품의 완성도와 직결된다. 더 심각한 것은 소프트웨어의 완성도는 "기술적인 빚(Technical Debt)"으로 남아서, 후속 과제의 개발 생산성에 심각한 위협으로 작용한다는 사실이다. 조금씩 쌓여가는 카드 빚과 같이 나중에 돌려막기도 되지 않을 정도의 심각한 상황이 오게 될지도 모른다.

이런 식이라면 차라리 일을 적게 하는 게 더 도움이 된다. 일을 많이 할수록 빚은 더 쌓여가게 되고, 나중에는 사람도 잃어버리게 될 것이기 때문이다. 시스템을 만드는 회사들의 가장 큰 자산은 제품이 아니라 그 제품을 만들어갈 수 있는 역량을 가진 개발자들이다. 개발자를 잃는다는 것은 회사의 미래가 없다는 말이다. 잘 생각해 보기 바란다. 하나 둘씩 핵심 개발자들이 떠나가는 회사에서 일한다면, 왜 그들이 자신이 다니던 회사를 등지고 나가는지 깨달아야 한다. 그들이 나가는 이유는 자신의 희생에 대한 정당한 평가와 그들 삶에서 빼앗긴 부분들(희생한 부분들)에 대한 나름의 보상이 없다는 것이다. 추가로 이야기한다면, 불합리한 것들이 합리적인 것으로 탈바꿈하여 되돌아오는 것에 대한 반응과 언제 개선될지도 모르는 암울한 미래에 대한 기대감 상실이다.

성공 스토리는 그 후에 시작되는 과제를 위한 실패 사례가 될 것인지, 아니면 정말 좋은 성공 스토리가 될 것인가를 고민해야 한다. 성공하는 데 필요한 것이 단순히 열심히 하는 것에 그치지 않고 분석적으로 무엇이 잘못되었고, 무엇이 개선되어야 할 부분이며, 무엇이 핵심 성공 인자 인지를 알아야 한다. 단순히 열심히 해서 그렇게 되었다고 이야기하는 팀장이 있다면, 이미 그 팀장은 말 그대로 출 퇴근 시스템 이외에는 아무 역할도 못 한다는 역설적인 의미가 된다. 좀 더 구체적으로 무엇이 정말 부서원들에게 힘이 되어 주었는지를 그 팀장 아래의 사람들은 잘 알고 있을 것이다. 무관심이 관심보다 더 나은 경우도 있으니 그것도 참고하기 바란다. 성공 스토리 못지않게 실패한 스토리도 나눌 수 있는 공간이 필요하다. 우리는 왜 실패했으며 실패할 수밖에 없었는지 정확히 깨닳아야 한다. 성공은 수 많은 실패의 결과물이 모여서 만들어지기 때문이다.

# [ 코드 리뷰에 대한 오해 ]

어린 아이의 숙제를 같이 봐주는 부모라면 아마도 리뷰가 얼마나 중요한지 기본적으로 잘 알고 있으리라 생각한다. 그렇다고 직장에서 동료가 코딩한 것을 리뷰하는 것을 부모와 같은 마음으로 하라는 이야기는 절대 아니다. 중요한 것은 리뷰를 통해서 같은 생각을 공유하는 것이다. 소프트웨어 개발에서 가장 중요한 요소는 고객 및 팀 내의 의사소통과 팀 외부와의 협업이기에, 코드 리뷰는 일종의 팀 내부의 아이디어 공유를 위해서는 필수다.

코드 리뷰에 대한 거부감은 소프트웨어 개발자라면 한 번쯤은 다 경험했을 거라고 생각한다. 자신이 만든 코드를 남에게 검사 받는다는 것이 그렇게 유쾌한 경험은 아니다. 그렇지만 이러한 거부감을 극복해야만 더 높은 수준의 개발자로 성장하는 것이 가능하다는 것도 이미 잘 알고 있을 것이다. 자신이 만든 코드는 자신의 것이 아니라 모두의 것이기에, 품질이 높은 코드를 짜는 것은 모든 소프트웨어 개발자 개인의 의무이기에 코드 리뷰는 당연한 것으로 받아들여야 한다. 하지만 그럼에도 불구하고 현실적으로 다양한 어려움이 있는 것도 사실이다.

코드 리뷰를 하기 어렵다고 느끼는 가장 큰 걸림돌은 "시간"에 대한 것이다. 당장 출시 해야하는 상황에서 코드 리뷰를 할 시간이 없다고 이야기한다는 점이다. 하지만 리뷰도 안 된 코드를 체크인(Check-in)해서 디버깅에 필요한 시간이 더 길어진다면 어떻게 할 것인가? 이미 여러 리포트로 나와 있지만, 코드 리뷰를 통해서 발생할 수 있는 버그의 25% 정도를 제거할 수 있다고 한다. (비공식적인 코드 리뷰의 경우) 25%의 버그를 디버깅하는 것보다 작성된 코드를 미리 검토하는 것이 부족한 시간을 아끼는 길이 아닐까? 따라서, 시간이 없어서 코드 리뷰를 할 수 없다는 것은 더 이상 좋은 변명거리가 될 수 없다.

그런데도 코드 리뷰에 대한 반감을 느끼는 것은 아마도 경험의 미숙에서 오는 자기방어 본능일 것이다. 더 많은 경험과 학습이 있었다면 당연하다고 받아들였을 것을, 자기만의 세상을 구축하고 남들이 이미 만들어 놓은 다양한 품질을 높이는 방법을 배척한 결과라고 밖에는 볼 수 없다. 실무 현장에서 이런 이야기를 많이 듣는 것도 사실 사람이 잘못되어서가 아니다. 경험적으로 어떻게 해야 할지 모르고 새로운 것을 적용하는 방법을 제대로 교육받지 못해서다. 따라서 어떤 일을 제대로 하기 위해서는 적절한 방법을 충실하게 익히는 것부터 시작해야 한다. 가장 중요한 것은 코드 리뷰에서 무엇을 볼 것인가를 정하는 것이다. 코드 리뷰에서 무엇을 확인할 것인가를 관련된 모든 사람들이 공통으로 이해하는 것부터 시작해야 한다.

먼저 할 일은 자신들의 코딩에 대한 스타일을 정하는 것이다. 개발자들 간에 서로 동의한 코딩 규칙(Coding Rule)을 만들어 나가는 것부터 시작해야 한다. 가독성(Readability)을 높이고 코딩의 오류를 줄일 수 있는 기본적인 지침을 만드는 것이다. 만약 이를 위반한 코드를 리뷰에서 발견한다면 당연히 수정 대상이 되어야 한다. 코딩 규칙을 만들 때 한가지 주의할 점은 너무 많은 규정은 만들지 않아야 한다는 점이다. 개발자가 기억할 수 있는 중요한 규칙은 7+-2(5~9)개 수준이어야 한다. 그 외의 규칙은 자동으로 점검해주는 도구를 통해서 코딩 시에 바로 확인할 수 있도록 해주는 것이 좋다. 물론 나중에 자동 검증 툴에서 당연히 검증할 수도 있어야 한다.

코드 리뷰는 누군가를 비판하는 것이 아니라, 더 좋은 코드를 만드는 방법을 공유하는 것이다. 코딩 스타일은 코드의 외적인 형태를 정의하는 것이고, 코딩 룰은 오류 가능성을 사전에 배제하는 하가지 방법이다. 코드 리뷰에서 확인하는 것은 사전에 협의된 객관적인 기준을 점검하는 자리이며, 잘못을 따지는 것이 아니라 다른 방법을 제안하는 것일 뿐이다. 코드는 혼자서 개발하는 것이 아닌 팀원 전체의 노력의 합이다. 코드에서 발생하는 문제는 모두의 것이기에 코드 리뷰는 개인의 성향을 대상으로 하는 것이 아니다. 모두의 것은 누구의 것도 아니라는 책임감 없는 말도 아니다. 모두의 것이기에 더 잘 가꾸어야 할 필요가 있는 것이다. 코드 리뷰를 통해서 지식의 공유가 일어나게 되며, 누가 무슨 일을 어떻게 하고 있는지 더 정확하게 이해할 수 있는 것이다. 코드 리뷰는 코드를 통한 대화를 나누는 것일 뿐이다.

# [ 개발 일정에 대해서 ]

소프트웨어 개발의 성공 여부를 판단하는데 필요한 기준은 정해진 자원과 일정 및 예산으로 사용자가 만족하는 소프트웨어를 만들어내는 것이다. 잘 알겠지만 이것을 지켜서 개발되는 소프트웨어는 별로 없다. 항상 개발자가 할 수 있는 것보다 더 많은 것을 기대하고, 더 적은 비용을 들이기를 원하기 때문이다. 결국 무리한 일정과 부족한 인력을 가지고 시작된 프로젝트들은 사용자가 만족할 만한 수준의 제품보다는, 회사의 프로세스에서 용납하는 최저 수준의 품질로 출시될 뿐이다.

개발 일정을 관리하는 것은 점쟁이 수준의 능력을 요구한다. 높으신 분들의 질문은 항상 언제 제품이 나올 것인가에 집중된다. 과정은 중요하지 않다. 그 분들의 당면한 제1의 목표는 매출과 순익이다. 따라서 언제 제품이 나와야 얼마를 벌 수 있는가를 상품기획이나 마케팅 등에서 정보를 얻어오고, 중요한 회의에서 매번 동일한 질문을 다른 표현을 빌려 개발팀에 물어본다. 개발 일정은 이미 예상 답안이 아니라 반드시 지켜야 할 개발자의 신뢰 차원으로 발전하게 되고, 그것을 맞추지 못하면 내년의 연봉과 고과는 이미 물 건너간 것이다. 따라서, 개발 일정을 정하고 관리하는 것은 항상 불협화음이 많은 부분이 될 수밖에 없다.

개발 일정은 예상일 뿐이다. 따라서 예상을 맞추지 못한다고 벌을 주겠다고 하면, 도전적인 일정은 절대 나올 수 없다. 누가 자신의 연봉과 고과를 두고 도전적인 일정을 잡겠는가? 그리고, 그 일정을 지연시켰을 때 들어야 할 온갖 비난과 질책, 이어지는 수많은 보고서와 지연 대책 회의를 좋아하겠는가? 만약 그런 회사에서 근무하고 있다면 절대 일정에 있어서 도전적으로 말하지 않아야 하며, 가능한 긴 일정을 잡을 수 있는 온갖 방법을 다 동원하는 게 좋을 것이다. 특히 다음 해에 대한 계획을 세울 때는 반드시 충분한 일정을 잡을 수 있도록 지금부터라도 완벽한 준비를 하는 것이 오히려 지금 진행 중인 과제를 하는 것보다 더 중요한 일이 될 것이다.

사실 제품 개발 계획을 만드는 시기는 다른 제품을 이미 개발하는 중간에 이뤄지는 경우가 많다. 따라서 계획을 세울 때는 그 계획에 대한 타당성 여부를 검토할 시간이 별로 없다. 지금 하는 제품의 개발이 지연되면 계획도 함께 지연되어야 하지만, 절대 그런 일은 일어나지 않는다. 즉, 제품의 계획과 현재 개발 중인 제품의 계획은 별개라고 생각하는 사람들이 많다. 하지만, 두 제품의 개발자는 같은 사람이며, 코드의 기반도 같은 것을 사용한다. 어쩌면 현재 개발 중인 제품의 업그레이드 버전일 수도 있다. 지금 개발 중인 제품에 투입할 자원도 없는데 내년에 만들 제품의 계획을 세우는 것을 별개의 일로 생각한다면, 그런 명령을 내린 사람들의 뇌도 완벽히 분리된 상태로 나누어져 있을 것이 분명하다.

도전은 실패를 용납할 수 있는 용기가 있어야 실행 가능하다. 실패하지 않도록 계획을 준비하는 것도 중요하지만, 실패할 가능성이 높다는 사실도 항상 인지해야 한다. 실패의 책임을 개발자에게 묻는 것은 어리석은 일이며, 오히려 그런 계획을 수립하도록 압력을 행사한 사람들이 가져가야 한다. 개발자의 연봉과 고과는 과제 실패의 책임이 아닌 과제에 대한 열정의 지표여야 한다. 발생시킨 버그의 개수나 밀도(Density)를 가지고 이야기하기 보다는, 제거한 버그의 개수를 가지고 말하는 것이 옳지 않을까? (물론 그렇다고 인위적으로 버그를 만들어 넣을 개발자는 없을 것이라고 가정 해야겠지만) 관리자는 자신의 경험이 더 많다고 자신하지만 그런 오래된 경험이 새로운 제품이나 새로운 분야에 동일하게 적용될 리는 없다. 자신의 능력이 안 된다고 생각되면 실무자에게 물어보라. 개발자는 이미 자신의 문제에 대한 대부분의 해결책을 알고 있을 것이 분명하다.

# [ 빈익빈 부익부 ]

대부분의 상품화 개발팀이 겪고 있는 문제는 코드의 복잡함에 대한 이해 및 변경이 어렵다는 것에 기인한다. 그리고, 상품화 개발의 특성상 절박한 일정에 따른 결함의 감내 및 구현 품질의 일관성 부족에서 나온다. 일정이 급하기에 구현하는 데 급급하게 되고, 품질이 부족한 코드도 쉽게 저장소에 넣는 것을 허락하게 된다. 급한 불을 끄는 것이 우선이고 나중에 올 문제는 그때 풀면 된다는 논리다. 하지만, 결국 모든 예상은 깨지고 나중에 풀어야 할 문제는 제대로 풀리지 않고 또 다른 기술적인 부채(Technical Debt)를 쌓는 일이 될 것이 분명하다.

가난한 사람은 지속해서 더 가난한 상태가 되고, 여유가 있는 사람은 계속 더 풍요로운 삶을 영위하게 된다. 이런 “빈익빈 부익부” 현상이 무언인가 잘못되었다고 보는 것이 타당할까? 일단 경제학적인 것은 제외하고, 개발에서 생기는 지속적인 야근이나 주말 특근을 가져오는 일을 방지하는 방법을 이야기해 보도록 하자. 소프트웨어 개발에서 흔히 이야기하는 문제의 발견이 이르(Early)면 이를수록 나중에 치러야 하는 비용이 크게 줄어든다는 것은 모두가 아는 사실이다. 이 사실을 기반으로 했을 때 우리가 가장 집중적(모든 정신적인 에너지를 다 소모해서)으로 일해야 할 시점은 바로 개발 계획을 만들 때가 된다.

계획의 수립과 요구사항에 대한 검토 및 설계가 중요하다는 말은 모든 소프트웨어 개발자들이 공통으로 알고 있는 사항이다. 하지만, 대부분의 경우 이런 것들을 제대로 수행해서 제품을 개발하는 것은 거의 불가능하다고 여긴다. 크게 두 가지 이유가 있는데, 첫 번째 이유는 앞에서도 이야기했지만 개발 일정의 부족에 기인한다. 제대로 정의하지도 못한 채로 개발에 들어가기 때문에 중간에 생기는 판단해야 할 결정들은 개발자가 임의로 하게 된다. 그리고, 그 뒷감당도 역시 개발자가 하게 되고, 중간에 들어오는 변경요구에 대해서도 실랑이를 벌이는 일이 일반적인 개발과정이 되고 만다. 안타까운 현실이지만 우리의 관리자들은 소프트웨어 개발에 대한 지식이 짧은 상태에서 임원이 된다. 개발을 많이 해본 사람도 코딩에는 자신이 있을지 몰라도 소프트웨어 품질에 대한 개선을 어떻게 해야 할지 모르는 경우가 많다. 실제로 대부분의 제조업에서 일하는 임원들은 소프트웨어를 개발한 적도 없는 사람이 더 많은 것이 현실이다.

이런 시간이 반복되면, 인력은 많이 들어가지만 제대로 된 산출물을 일정에 맞춰서 개발하는 것이 어렵다는 것을 알게 된다. 지속적으로 개발 비용(일정)은 증가하지만, 제때에 나오지 못하는 제품은 시장에서 큰 돈을 벌기 어렵게 된다. 결국 이익은 줄어들고 줄어든 이익으로 뽑을 사람도 줄어든다. 줄어든 사람으로 예전에 개발한 제품에 대한 지원과 신규 제품을 만드는 것을 진행하게 되고, 결과적으로 일정은 더 늘어나게 된다. 누구 하나 잘못한 것은 없어 보이는데도 점차 회사는 수익 구조가 나빠지게 된다. 이를 극복하는 것이 대부분 회사가 처한 가장 중요한 문제이다. 어떻게 이런 상황을 반전시킬 수 있을까? 사실 대단한 방법이 있는 것은 아니다. 이미 다 알고 있는 방법이 답이다. 즉, 개선하기를 원한다면 개선할 수 있는 사람을 확보하고, 그들에게 개선할 권한과 시간을 주는 것이다. 지금 현재의 급한 일에만 매달리게 되면 개선할 수 있는 기회는 절대 오지 않는다. 지금 하는 일만 열심히 하면 그 일을 더 잘하는 경쟁자가 결국 승리할 뿐이다.

효율을 극단적으로 높이면 한 사람이 세 사람 몫의 일을 하기를 원할 것이다. 하지만, 그 한 사람이 진정으로 세 사람의 일을 하게 되려면, 그 사람이 만든 결과물을 세 사람이 사용하도록 만들어야 하지, 세 사람이 해야 할 일을 한 사람이 모두 다 짊어지게 해선 안 된다. 소프트웨어 개발이라고 한다면, 프레임워크나 플랫폼과 같은 것을 만들어서 소프트웨어의 재사용률을 높이는 것이 한 가지 방법이다. 분리된 팀에서 그런 활동을 해서 상품화 조직으로 결과물을 재사용할 수 있도록 넘겨주는 것이, 효율보다는 생산성의 극대화를 이루는 길이다. 코드의 재활용은 아주 오래된 이야기지만 제대로 받아들여지고 있지 않은 것이 현실이다. 그런 현실의 최대 걸림돌은 "동작하는 코드를 왜 바꿔야지?"라는 질문이다. 오래동안 누적된 빚을 갚을 용기(혹은, 의지)가 없는 것이다 이미 쌓인 빚이 많다면 아무리 열심히 일해도 빚이 쌓이는 구조 밖에는 만들지 못한다.

# [ 동작하는 소프트웨어를 제대로 만들기 ]

소프트웨어 개발자로 회사 생활을 할 때 가장 큰 스트레스로 작용하는 것은 역시 일정(Schedule) 관리이다. 관리자는 절대 넉넉한 일정을 주지 않기에 개발 PL(Project Leader)은 확신이 없는 일정에 의도적으로 몇 개월을 더해서 과제 계획을 제시한다. 하지만 이렇게 제시된 일정은 몇 번의 회의와 그 속에서 여러 번에 걸친 질책을 듣고 나면 반 정도가 줄어서 돌아오곤 한다. 그렇게 느려지는 이유가 무엇인지 꼬치꼬치 캐묻고 자신이 최적의 일정이라고 생각하는 날짜를 못박기 일쑤다. 상품기획과는 항상 제품의 라인업(Lineup) 회의에서 결론이 나지 않는 평행선을 달리게 되고, 결국에 가서는 최종 결정권자가 알아서 그냥 정해 버린다. 이런 상황에서 합리적인 과제 일정을 추정하는 것은 의미가 없다. 여기에 더해서 과제 일정 지연을 "고과"와 연계하겠다고 이야기한다면 거의 멘붕된 상태로 자신의 책상 앞에 돌아오게 된다. 자신이 담당하고 있는 개발자들에게는 미안한 마음으로 결정 사항을 공유하게 되고, 피곤한 눈길을 애써 외면할 수밖에 없다.

회사를 그만두게 되었을 때 들었던 가장 웃기는 이야기는 다음과 같다. "너, 나한테 미안한 것 없어?", 상무라는 사람이 한 말이다. 그 말은 들었을 때 가장 먼저 머리 속에 들었던 생각은 함께 고생하던 소프트웨어 개발자들에게 좀 더 좋은 환경에서 일하게 해 주지 못한 것이 떠올랐다. 하지만, 단언컨대 그런 질문을 던진 사람에게는 하나도 미안한 생각이 들지 않았다. 왜냐하면, 문제의 원인이 누구에게 있는가를 그 사람은 알지 못하기 때문이다. 소프트웨어 개발이라는 것을 제대로 한 번도 해보지 못하고 제품 개발 팀장을 맡은 사람이 이해할 수 있는 수준을 벗어났기 때문이다. 최소한 회사에서 상업적인 소프트웨어 개발을 15년 이상(그리고, 컴퓨터 공학 4년제 대학과 석사나 박사 학위가 있는) 해본 사람의 말에 귀를 기울일 수 있는 준비가 되지 못한 사람에게는 어떤 이야기도 통하지 않는다. 오히려 미안해야 할 사람은 그런 질문을 던진 사람이지 책임만 있고 권한은 없는 사람이 사과할 필요는 없는 것이다.

"동작하는 소프트웨어"는 사람과 시간을 들이면 만들 수 있다. 하지만 문제는 항상 그 다음이다. 소프트웨어는 건축과 달라서, 하나를 만들든 두 개, 혹은 수십 만 개를 만들어도 동일한 소프트웨어는 하나만 만든다. 그리고, 그것을 유지보수하고, 로드맵 상에 있는 제품들에 다시 적용하게 된다. (물론 다른 제품은 다른 기능이 추가될 것이고 기존에 출시된 제품도 끝없이 유지보수 해야 하겠지만) 그냥 동작하기만 하는 소프트웨어는 학교에서 숙제로 할 수준이지 상업적으로 적용하기 위해서는 "변경 용이성"과 "테스트 용이성", "확장성", "이식성" 등 다양한 부분을 고려해서 만들어야 한다. 이것은 "동작하는 소프트웨어를 제대로 만들기"위해서 필요한 것이다. 이를 위해서는 “제대로 만들 시간”을 주어야 한다는 사실을 높으신 분들은 너무나 자주 놓친다. 예를 들어, 100MM(Man-Month: 한 사람이 한 달동안 투입)가 들어가야 할 과제가 있다고 하면 열 사람이 십개월 동안 일해야 한다. 그렇다고해서 같은 일을 이십 명이 그 절반의 기간에 끝내지는 못한다. 그런 식의 계산이라면 사람의 수를 늘리면 하루도 걸리지 않고 끝낼 수 있다는 말이 되기 때문이다.

사실 백 명이 일 개월에 끝내 수 있는 과제라면 아마도 이미 다 만들어 놓은 것을 거의 고치지 않고 아주 조금(조금이라도 고치면 치명적일 수도 있지만)만 변경하는 정도의 수준일 것이다. 즉, 미리 어떻게 만드는지와 그 해결책, 해당 소프트웨어 대한 적절한 교육을 받은 개발자들이 다 갖추어진 상황일 것이다. 하지만, 현실은 절대 그렇지 않다. 열 명이 십 개월에 개발할 내용을 백 명이 한 달에 개발하게 되면, 제대로 된 개발 프로세스나 사후관리 등에 대한 것은 전혀 고려되지 않는다. 말 그대로 “그냥 동작하는 소프트웨어”를 만들어 사용자에게 배포하는 것일 뿐이다. 누구도 원하지 않는 소프트웨어가 만들어질 것이 분명하다. 소프트웨어는 누군가의 말(명령)로 자동으로 코드가 만들어지는 것이 아니다. 누군가가 일정을 당기라고 요구하면, 구현해야 할 중요한 기능의 일부는 제거해야 한다. 제품으로서의 가치를 손상하더라도 일정을 맞출 수 있는 해법은 그것밖에 없다. 그리고, 몇 개의 버그가(혹은, 몇 백 개의 버그, 몇 천 개의 버그 등등) 숨어서 고객에게 제품과 함께 갈 것이다. 일정 단축을 지시하는 것은 소프트웨어 개발 품질을 낮추는 가장 큰 요인이라는 것을 그 분들도 이제는 충분히 알아야 한다.

# [ “Man-Month” 미신? ]

소프트웨어 개발자라면 한 번쯤은 다 들어봤을 "Mythical Man-Month"라는 책을 보면, 소프트웨어를 만드는 관리자로서 눈여겨 볼 부분이 많다. 특히 아무리 시간이 흘러도 변화되지 않는 업계의 관행은 지금도 많은 개발자의 삶을 파탄으로 이끌고 있다는 생각이 든다. 회사는 "최고의 자산은 사람"이라고 이야기하지만, 언제든 대체가 가능한 대표적인 자원(?)으로 사람을 생각하고 있는 것도 사실이다. 하지만, 대체 가능한 자원이라고 생각하면서도 대체가 가능한 시점을 기다리느라 바로 잘라 버리지는 않는다. 충분히 바꿀 수 있는 시점이 되었다고 믿는 순간까지 기다렸다고 나가라고 하는 것이 오히려 회사로서는 유리하기 때문이다.

실패하는 과제들을 보면서도 새로운 방법을 생각하지 못하고, 혹은 새로운 방법을 적용할 시간조차도 주지 않으면서 달라지기를 원한다. 정작 달라져야 할 사람은 소위 말하는 “입으로 관리하고 관련된 분야의 지식도 전혀 없는 지시만 일삼는 사람”들이다. 탁상 머리 행정은 공무원만의 전유물이 아니다. 토요일 날 특근을 하면 돈을 올려주겠다고 이야기하면, 어쩔 수 없이 나와야 한다는 강박관념에 금요일까지 끝낼 수 있는 일도 토요일로 미루게 된다. 어차피 특근해야 하는데 빨리 일을 끝낼 필요가 전혀 없기 때문이다. 회사로서는 토요일 특근비를 제공하고 점심도 멱여 주겠지만, 결과적으로 생산성은 나아지지 않을 것이다. 오히려, 토요일을 철저하게 쉬게 만들어야 그 주에 끝내야 할 일들은 마무리가 된다. 회사에서 대한 충성심이나 선의에 기대기 보다 사람은 생각보다 영리하다는 것을 알아야 한다. 그리고, 돈은 적절한 보상이 아니라는 것도 알아야 할 것이다.

팀을 구성할 때 능력 있는 사람들로 개발자들로 모으기를 원하면서도, 그들 개개인에 대한 비전이나 경력(Career)에 대한 것은 신경 쓰지 않는다면, 그 팀이 오랫동안 유지될 수 있을까? 자신만의 생각을 강요하고 다른 의견일 경우에는 자신을 설득하라고 요구한다면, 누가 제대로 된 대화를 할 수 있을까? 소프트웨어 개발자들은(모든 개발자가 마찬가지겠지만) 자신이 하는 일 속에서 만족감과 발전하고 있다는 느낌을 중요하게 생각한다. 지속되는 단순 업무와 의사 결정 과정에서의 제외는 과제에 대한 집중도를 낮추게 된다. 그냥 시키는 대로 하겠다는 생각을 하는 순간, 과제는 이미 자신의 것이 아닌 남의 것이 되고마는 것이다. 남의 과제에 잠시 발을 두고 있을 뿐이기에 자신의 책임도 없으며 개선할 필요도 느끼지 못한다. 과제를 하는 순간만 모면하면 되기 때문이다. 이런 식으로 팀원들이 과제에 참여하게 된다면 당연히 성공적으로 과제가 마무리되기를 기대할 수는 없다.

관리자는 자신의 불안을 아랫사람의 야근과 특근으로 메우려고 한다. 자신의 무능함을 말도 안 되는 꾸짖음으로 표현하고, 자신의 비겁함을 아랫사람의 책임으로 돌린다. 윗사람에 대해서는 귀를 열고 펜으로 적어 놓지만, 아랫사람의 이야기에 대해서는 열어보지 않은 메일처럼 얼마나 쌓여가는지도 모르고 있을 것이다. 잘못은 그런 그들을 그 자리에 오르게 만든 사람들이지, 그 사람의 관리를 받는 사람들이 아니다. 왜 매출이 오르지 않고 지속해서 일정이 지연되는지, 그리고, 고객들을 왜 우리가 만든 물건을 사지 않는지는 그런 사람들이 고민해야 한다. 이유는 하나다. 고객이 원하지 않는 제품을 만들어내기에 팔리지 않는 것이다. 일정 준수보다는 기능이 제대로 사용자가 원하고 편하게 이용할 수 있는지가 더 중요하다. 100%일정을 준수했다는 거짓말보다 정말 필요한 제품을 만든데 집중해야 한다. 결국 소프트웨어 개발자가 해결해야 할 것은 사용자의 문제(혹은 불편함)이지, 높으신 관리자의 권력이나 지위 향상이 아니다. 정말 회사를 위한다면 고객이 원하는 제품을 만드는 데 모든 노력을 집중해야지, 자신의 무지를 해결하기 위해서 자질구레한 일을 시키는 관리자가 되어서는 안된다.

# [ 긍정적 일정계획과 비관적 추정 ]

소프트웨어 개발자는 일정 추정에 대해서 도전 정신을 가져야 한다고 경영진이나 고위층에게서 지속해서 요구한다. 그리고, 그 결과는 과제의 지연이나 제대로 된 기능이 없는 저질의 제품으로 이어지고, 책임은 고스란히 개발자의 몫으로 남겨진다. 결국 일해도 욕먹고 일하지 않으면 의지가 없다고 욕먹는다. 어쩔 수 없이 욕먹을 인생을 지속하게 된다. 그러다가 성공하는 과제라도 있으면 다행이지만, 그렇지 못할 경우에는 회사에서 패배자 취급을 당하기도 해야 한다. 좋은 고과는 물 건너가고 그나마도 운이 좋다면 그럭저럭 회사 생활을 영위하는 별 볼일 없는 사람이 되고 만다.

우리가 하는 과제 대부분은 계획을 먼저 세운다. 과제의 내용을 제대로 파악할 시간이 없는 상태에서 계획부터 만들고 본다. 과제에 대한 분석 및 위험(Risk)에 대한 대응책을 마련한 게 없는 상태에서 과제의 주어진 목표에 대한 일정을 통보받는다. 개발자가 제대로 된 계획(일정)을 수립하기 전에 남들이 다 정해서 로드맵이라는 이름으로 알려준다. 반대 의견을 제시하고 필요하지 않은 기능이나 성능과 같은 것을 제외하려고 하면, 상품 기획이나 기타 부서에서 그렇게 만든 제품은 판매할 수 없다고 완강하게 반대한다. 이런 대립 상황이 평행상태로 지속되면 결국에 피해를 보는 것은 개발자들이다. 높으신 분들은 항상 긍정적인 낙관에 기대어 미래를 생각하기 때문이다. 비관적인 추정을 내놓아서 “근거 없는 밝은 미래”를 깨는 것은 절대 용납되지 않는다.

높으신 분들에게 유일하게 중요한 것은 제품 개발이 완료되는 날짜다. 제품의 품질이나 어떻게 만드는지, 코드의 재활용을 위한 구조적인 아름다움이나 테스트의 자동화 등은 조금도 가치가 없다. 임시 직원인 "임원"들은 매년 계약을 갱신해야 하고, 2년짜리 과제나 장기적인 개선에 대한 것은 그들의 관심이 아니다. 자신이 가고 난 다음에 좋아질 것으로 생각되는 일에는 조금도 자원을 투입하고 싶어하지 않는다. (할 이유도 없다고 보는 것이 더 정확하다.) 회사의 장래나 그 속에서 일하고 있는 개발자들의 미래는 이미 자신들의 관심 밖에 있다. 도전적인 과제를 하는 것도 자신의 KPI(Key Performance Index)에 위험이 있으니, 될 수 있으면 자신이 책임지지 않는 범위에서 진행을 허용한다. 그런 과제들은 자신의 MBO(Management by Objectives)에 속하지 않도록 만든다. 결국에는 안정적으로 도달할 수 있는 것들에 대해서만 책임을 지려고 하고, 조그마한 단기적인 성공에는 상을 주지만, 장기적인 개선 과제는 상품화와 관련이 없을 때는 외면하고 만다.

도전적인 일정을 만드는 이유는 과제의 성공 가능성을 높이기 위한 “일정 버퍼(Buffer)”를 생각하기 때문이다. 10개월이 걸린다고 추정된 과제가 있다고 할 때, 8개월 만에 끝내라고 요구하고 나머지 두 달은 그래도 끝내지 못할 경우를 대비한다. 물론 이렇게 정해진 일정으로 인해서 발생하는 더 많은 빚은 생각할 여력이 없다. 일단은 매출 목표를 맞추어야 하고, 다른 부서의 임원과의 경쟁에서 이겨야 하기 때문이다. 회사의 이익이나 개발자의 경력 개발등은 그들의 관심에서 벗어날 수밖에 없는 것이다. 최대한 빨리 만들어서 시장에서 조금이라도 팔려야 다음번 고과에서 좋은 성적을 얻을 수 있고, 그래야만 계약 기간이 연장될 수 있기 때문이다. 장기적인 비전 같은 고상한 사람들의 몫이라고 던져버린지 오래다. 그에 맞춰서 개발자들은 항상 바쁘고 급한 코딩에 집착할 수밖에 없는 것이다. 그런 집착이 오래가면 굳어진 사고로 이어지게 되며, 가르치지 않아도 세대를 거치면서 이어지는 “불량 문화”가 자리잡게 된다.

이런 것들을 극복할 방안은 최소로 일하고 자주 보여주는 것이다. 조금씩 일할 수 있는 목표를 여러 개의 마일스톤(Milestone)으로 정해서, 각각의 마일스톤마다 구현된 것을 지속해서 이해 관계자들에게 알려 지나친 낙관을 하지 못하도록 만들어야 한다. 나중에 과제 완료 시점에 한꺼번에 문제점들이 터져 나오지 않도록 조금씩 조금씩 과제의 진행 과정에서 보여주는 것이다. 또한 마일스톤 상에서는 반드시 우선순위가 가장 높은 항목들이 먼저 개발되도록 만들어야 한다. 그래야만 적어도 팔리는 제품을 언제든 만들어 낼 가능성이 조금이라도 커지기 때문이다. 항목의 우선순위에 대한 것도 항상 이해 관계자들과 같은 생각을 가질 수 있도록 공유해야 한다. 관리되지 않는 다는 것은 변화가 없다는 말의 동의어일 뿐이며, 관리란 지속적인 업데이트가 발생하기 때문에 필요하다. 결론적으로 이야기하면 일정 계획은 될 수 있으면 비관적으로 잡아야 하며, 그것에 도달하기 위해서는 긍정적으로 변화를 받아들이도록 임해야 할 필요가 있다.

# [ 축구와 소프트웨어 개발이 닮은 점? ]

축구를 잘하기 위한 3대 요소라고 할 수 있는 것은 뭐가 있을까? 개인적인 생각이지만 체력, 기술, 스피드가 그것이라고 본다. 축구 선수의 체력은 90분간 뛸 수 있을 정도로 강인해야 하고, 기술은 어릴 때부터 확실하게 잘 배워두어야 할 개인기 부분이며, 스피드는 모든 운동의 기본으로 남보다 한발 앞서 움직이거나 슈팅 타임을 한 박자 빠르게 가져가야 하기 때문이다. 물론 여기서 한 가지 더 추가할 부분이 있다면 당연히 팀 워크가 될 것이다. 축구는 혼자 하는 것이 아니고, 여러 명이 한 가지의 목표를 달성하기 위한 스포츠이기 때문이다. 하지만 팀워크는 어쨌든 개인적인 것이 아니라서 일단은 앞의 세 가지만을 가지고 개인의 입장에서 반드시 갖추어야 할 것들로 한정하겠다(팀워크 또한 개인의 자세에 달려있는 부분이지만).

소프트웨어의 개발자는 오랫동안 일해야 한다. 생각의 연속선 상에서 지속적인 집중력을 요구하는 일이 많기에 기본적으로 체력이 뒷받침되어야 한다. 많이 먹는다고 일을 잘하는 것은 아니지만, 그만큼 열량을 소모하는 사람은 활력이 있는 사람일 것이다. 어쨌든 우리 주변에는 천재적인 사람은 몇몇이 되지 않기에, 일에 집중(몰입)을 오랫동안 할 수 있는 사람이 적합한 사람일 것이다. 기술은 당연히 문제를 해결할 수 있는 개인의 역량이다. 무슨 도구를 사용하건 상관없이 사람이 해야 할 부분은 반드시 있으며, 도구가 모든 문제를 해결해 주지 않기에 항상 개인기를 다듬고 충분한 역량을 확보하고 있어야 한다. 스피드는 한 가지 일을 하더라도 정확하게 빨리하는 사람을 좋아하게 만든다. 다양한 부분에서 스프드를 요구하기에 경험적인 측면과 신기술에 대한 이해 및 적용도 가능한 열린 마음으로 받아들일 수 있어야 할 것이다.

조금 억지 같기는 하지만 어쨌든 팀으로 하는 경기는 대부분 단체로 개발하는 소프트웨어와 무관하지는 않다. 같은 팀 내에서도 개인의 역량은 큰 차이를 보인다고 하고, 팀 간에는 두 자리수 이상의 차이를 보일 수 있다고 한다. 팀과 팀의 역량 차이는 당연히 팀워크라고 볼 수 있으며, 팀 내의 차이는 개인기라고 볼 수 있다. 개인기가 우수한 인력도 필요하지만, 경기에서 이기기 위해서는(프로젝트를 성공적으로 마무리하기 위해서는) 팀워크라는 개인과 개인을 묶어주는 연결 고리가 필요하다. 소프트웨어가 다양한 모듈(컴포넌트)로 개발되고 이를 묶어주는 인터페이스가 있듯이, 팀워크가 필요한 부분은 모듈과 모듈 사이의 공간일 것이다. 소프트웨어는 각각의 구성 요소들이 밀접하게 결합(Coupling)되는 것은 좋지 못하다고 이야기하기에, 분리를 위해서 잘 정리된 인터페이스를 만들어나가는 “밀접하게 협력하는 팀워크”는 반드시 필요하다. 말할 필요도 없이 팀워크는 함께 한다는 의식이 모두와 공유될 때 가능하다.

관리자(코치)는 개인의 기량을 키우고 어떤 일도 함께 이겨나가는 능력(체력)을 길러주는 일을 해야 한다. 빠른 일 처리를 위해서는 정확하게 구현되었음을 확인할 방안이 필요하며(시간을 줄이는 것은 테스트에서 발생하는 버그의 수를 미리 제거할 수 있을 만큼 줄여 놓아야 한다.), 팀워크가 생길 수 있도록(일하기 편한 환경이 되도록) 이끌어야 한다. 당연한 말들이지만 대부분 관리자는 자신의 역할이 힘을 의미한다고 생각한다. 하지만 실제로는 관리자라는 역할은 팀 내에서는 낮은 곳에 위치해야 하고, 팀 외부에는 단합된 힘의 대표자로 보이도록 해야 한다. 모든 책임은 관리자의 몫이어야 하고 모든 영광은 팀원들이 가져가야 한다. 관리자는 팀원들이 커가는 모습에서 만족을 얻어야지 일의 결과를 자신만의 것으로 만들어 승진을 목표로 하는 것은 어리석은 행동이다. 회사에서 발생하는 이직 사유의 대부분은 관리자를 제대로 만나지 못한 것이라는 사실을 잊어서는 안 된다. 사람을 잃는 것이 가장 큰 손실임을 기억해야 하고, 구성원 개개인을 존중할 수 있는 분위기를 만드는 것이 관리자의 해야할 일이다.

# [ 실패하는 프로젝트의 시작 ]

실패한 프로젝트의 정의는 뭘까? 대부분 일정, 자원, 품질에서 만족하지 않는 것을 말한다. 미리 정의된 것들이 어긋나서 프로젝트의 이해 관계자들이 받아들일 수 없는 결과를 만들어내는 것이다. 자세히 들여다보면 사실 실패하는 프로젝트는 대부분 실패할 것이라는 것을 예측한 초기 예상에서 달라지지 않는다. 실패할 상황을 만들어놓고 시작하기에 실패할 수밖에 없다. 소프트웨어 개발에 무지한 높으신 분들은 일정을 반으로 줄이라고 요구하고, 필요한 인력은 대부분은 이미 다른 과제에 투입된 상황이라 50%만 배정해 주면서도 품질은 세계에서 최고 수준을 요구한다. 당연히 소프트웨어 개발자도 그런 우수한 품질을 가진 제품을 만들기를 원한다. 하지만 기대가 높으면 그것이 안 되었을 때 받아야 할 비난의 수위도 높다. 이런 과제는 아무도 맡기 싫어하는 것이 당연할 것이다.

예전에 근무한 회사에서 다른 사람이 관리하던 과제를 맡은 적이 있다. 물론 그 과제를 맡고 싶어서 자원한 것은 아니었고(이미 악명이 높은 과제였기에), 갑작스러운 통보와 바로 다음 날로 이어진 조직개편으로 어쩔 수 없는 상황에서 맡게 되었다(아마 맡지 않겠다고 우겼으면, "인사는 회사의 권한이다."라는 말을 들었을 것이 분명하다. 하지만, "퇴사는 개인의 권한이다."라는 말도 있다.). 처음 한 일은 그 과제의 내용과 일정에 대한 파악이었다. 이미 외주로 진행하는 부분도 있었고 과제의 절반 정도 기간이 지난 시점이었다. 하지만 다른 팀에서 하는 과제의 영향으로 이미 끝난 과제에 인력의 절반 이상을 투입하고 있었으며, 기존의 플랫폼도 신규 기능을 추가하면서 코드를 통합하는 과정 중이라, 원래 계획했던 과제는 제대로 시작도 못 한 상황이었다. 대략 10~20% 정도의 진척도로 진행되고는 있었다고 보이지만, 남은 기간으로 봤을 때 과제의 실패가 유력해 보였다.

그런 상황을 처음 보고했을 때 들었던 말은 "일정의 변경은 없다."는 것이었다. 그리고, "도전 정신이 부족하다"는 말도 나왔다. 그 정도에서 그치면 좋았겠지만 그 보고 이후에는 지속해서 이런 저런 이유로 공격적인 비난을 쏟아내고, 과제에 크게 관련이 없는 숙제도 갑작스럽게 큰 폭으로 증가했다. 물론 숙제를 혼자 하면 좋겠지만 처음 맡은 일이라 팀원들에게 할당할 수밖에 없는 숙제도 나날이 늘어나는 상황이 되었다. 일정 계획은 즉시 대답할 수 있도록 달달 외우라고 맨날 들볶았지만, 이미 그런 의미 없는 암기된 일정 계획으로는 아무것도 보장할 수 없는 상황이었다. 구현되지 않은 부분들이 전체 기능의 20%가 넘는 상황이 되었으며, 테스트 자체가 무의미할 수밖에 없었다. 당연히 테스트 팀에서는 미구현된 부분을 집중적으로 공격했고, 그 결과를 "전과"로 의기양양하게 윗선에 보고서를 올렸다. 빈곤의 악순환과 같은 고리를 모조리 받아들이고 인내를 가지고 버텨야 하지만 그럴 수 있는 상황도 못되었다.

팀원들은 장시간의 근무로 인해서 신경과민 상태에서 힘들어 했으며, 팀장은 일정 변경은 안된다는 말만 되풀이할 뿐이었으며, 새로운 중간 관리자는 사태를 파악하고 일정계획을 다시 수립하기 위해서 매일 반복적으로 실무 개발 담당자들을 불러데기 바빴다. 나아지는 것은 없었다. 일정은 앞으로 최소한 3개월 이상 지연이 될 것을 개발자들이 예상했지만, 높으신 분들은 앞당기라는 말만 되뇌었으며, 결국 계획은 이전과 동일하다는 결정만을 들을 수 있을 뿐이었다. 차라리 새로운 일정 계획을 수립하는 일을 하지 않았으면 모르겠지만, 그런 일 때문에 빼앗긴 일 할 시간은 과제를 더 지연시킬 뿐이었다. 과제를 많이 했다는 경험이 있는 사람들이라고 생각했는데, 관리자들은 말 그대로 관리만 했거나 소프트웨어 개발 경험이 거의 없다고 생각되는 일들만 지속해서 지시했다. 실패로 가는 과제에 실패할 이유를 하나 더 더할 뿐, 결코 개발자를 도울 생각은 하지 않는 사람처럼 보였다. 엎친 데 덮친 격으로 과제 책임자는 동일한 보고를 여러 번 반복해야 했고(주간보고만 3번 이상), 그런 반복된 보고는 아무런 감흥도 줄 수 없었다.

실패하는 과제의 원인은 다양하게 있지만, 대부분 잘못 수립된 일정 계획과 너무 많은 구현 해야할 기능, 적절하지 못한 인력 등으로 인해서 발생한다. 특히 일정 계획은 높으신 분들의 주된 관심사이며 유일한 자신들의 존재 목적이다. 매출이 발생하지 않으면 내년 계약이 불확실한 사람들은 최대한 자신의 목숨줄을 위해서 다른 사람들을 괴롭힐 수밖에 없다. 그리고, 불안한 마음은 계속 일정 계획을 더 줄이도록 요구하게 만든다. 과제는 시작도 하기전에 장미빛 미래를 그리고 있을지도 모르지만, 실무 담당자들은 원하는 시나리오대로 과제가 끝나지 않을 것이라는 것을 다들 예상하고 있을 것이다. 실패하는 과제는 이미 실패라는 예상을 뒤집지 못한다. 성공할 가능성이 50%를 넘으면 그나마 해볼 마음이라도 생기겠지만, 이미 실패할 수밖에 없는 길로 걸어들어가고 있는 것이다. 실패는 부끄럽지만 그것으로부터 배우지 않으면 반복된 실패를 할 수밖에 없다.

# [ 지속적인 개선에 대해서 ]

소프트웨어 개발은 현실에 기반을 두고 꿈을 향한 다리를 만들어가는 일이다. 다 만들어진 다리도 지속해서 보완해 주지 않으면 모든 건축물이 그렇듯 무너지고 만다. 미국의 샌프란시스코에 있는 금문교(Golden Gate Bridge)라는 다리는 그 자체로 사람들을 끊임없이 끌어당기는 매력이 있다. 그 다리를 만들기 위해서 많은 사람이 희생됐고, 현재는 그 다리를 유지하기 위해서 또 다른 큰 노력을 기울이고 있다. 따라서 모든 사라져가는 것들을 현실에 잡아두기 위해서는 끝없는 개선이 해답이다. 다른 한 가지 방법은 대체할 수 있는 무언가를 다시 만들어가는 것이 있다. 어쨌든 중요한 것은 두 가지 일이 병행되어야 경험이 세대를 두고 이어져 갈 것이라는 점은 확실하다.

한창 바쁜 상품화 팀에 이런 저런 개선을 하라는 압박이 들어오는 경우가 많다. 제3자의 눈으로 봤을 때는 불합리하고 이상한 점이 많이 보이기 때문에 그런 것이다. 하지만 그 속에서 일하는 사람들은 나름의 최선을 다해서 그 일을 하고 있는 것이다. 개선은 여유가 있을 때나 가능하다. 딜레마는 여기서 생겨난다. 계속 지금의 상황을 유지하는 것은 무너지는 시간을 늦추는 것밖에 되지 못하지만, 그렇다고 대규모 개선을 하기에는 여유가 없다. 이것을 어떻게 해결해야 하는 가에 대한 해답을 찾는 것은 모든 과제를 담당한 책임자가 느끼는 고민일 것이다. 한 가지 확실한 대답은 상품화에 대부분의 노력을 쏟고 있는 조직에서는 그런 것이 절대 불가능하다는 점이다. 따라서, 우리는 "여분"의 무엇이 필요하다. 상품화에 사용된 코드를 개선해서 더 좋은 구조와 확장성 및 이식성, 변경 용이성 등등을 가진 소프트웨어를 만들어 갈 새로운 힘이 필요하다.

새로운 팀을 만드는 것은 상품화를 담당하는 팀의 핵심적인 인력을 필요로 한다. 물론 이들 핵심 인력 들은 새로운 것을 하기를 좋아하고, 자신의 단점이 무엇이며 지금까지 개발한 소프트웨어가 어떤 문제가 있는지를 명확히 이해하는 인력들이어야 한다. 그들 한두 명을 뺀다고해서 과제가 무너지는 것은 아니다. 이미 그들이 개발한 코드가 있고, 그것을 이용해서 기존 제품에 기능을 추가하는 것이 상품화의 역할일 것이기 때문이다. 그렇게 빠져나온 인력과 소프트웨어 설계에 정통한 인력이 붙고 실제로 코딩을 담당할 추가 인력 들을 뽑아서 넣는다면 충분하다. 상품화를 담당하는 팀에 인력 1~2명이 더 있다고해서 과제 개발이 특별히 빨라지는 것은 아니기에 이것은 충분히 가능하다. 물론 “그렇게 하면 차기 제품은 어떻게 개발하나요?”라는 질문은 하겠지만, 그것은 이미 답이 정해져 있다. 개선된 소프트웨어로 플랫폼을 만들어서 대체할 것이기 때문이다.

기존 소프트웨어의 지속적인 개선의 한계는 이미 여러 가지 책 속에서 나와 있다. 아무리 지속적인 개선을 많이 하더라도, 결국에는 제품에 내재한 버그의 수가 한계 수준 이하로는 떨어지지 않을 것이다. 더 이상의 새로운 기능을 추가하기 힘든 상황이 올 것이기 때문이다. 소프트웨어가 “소프트(Soft)”하다는 것은 지속적인 개선(변경)을 할 수 있기 때문이지만, 그런 개선의 속도가 점차 느려져서 나중에는 도저히 개선할 수 없는 “하드(Hard)”한 상황이 오기 때문이다. 지금의 과제에서 미래를 위한 여분(여유)를 찾지 않는다면 결국에는 자신이 가지고 있는 직책과 과제가 함께 물속으로 무너져 내리는 모습을 보게 될 것이다. 우리의 꿈이 있는 저쪽 피안의 세계에 도달할 수 있는 다리는 "Golden Gate"가 아닌 "Hell Gate"가 열리게 될 것이다. 여유는 누군가가 만들어주는 것이 아니라 스스로 찾아서 만드는 것이다. 모두에게 주어지는 시간은 한정적이며 동일하다. 누가 얼마나 더 잘 활용하는가에 따라 결과가 달라질 뿐이다.

# [ 계획과 관리 ]

모든 일에 선행하는 것은 계획을 짜는 일이다. 계획은 앞으로 어떻게 할 것인가에 대한 추정으로 어떤 목적을 가지고 얼마만큼의 비용과 인력을 사용할 것인가를 예측한다. 개인적으로 여행을 가려고 해도 계획을 세우고, 언제 어떤 일을 할 것인가에 대한 대략적인 안(Plan)을 수립하게 된다. 소프트웨어 개발의 경우에는 "과제 계획서"를 만드는 것으로 그것을 대체하게 되지만, 사실 계획은 계획일 뿐이다. 어떤 일이 과제 중에 발생할지 완벽하게 알 수 있는 방법은 없다. 따라서, 우리는 "관리"라는 것이 필요하다. 꾸준히 현실의 상황을 반영해서 계획을 수정하는 것이 관리이기 때문이다. 하지만 대부분의 경우 계획만 있고 그 계획에 대해 관리는 제대로 하지 않는다. 정말 문제가 되는 것은 계획도 부족하지만, 그 부족한 계획을 지속해서 관리하지도 않고 아무도 관심을 두지 않는다는 점이다. 만들어진 과제 계획서는 한번만 쓰이고 버려지는 것이 일반적이다.

소프트웨어 개발에서는 과제의 건강한 상태를 유지하기 위해서 항상 기준에 따라 관리해 나가야 한다. 초반에 예측하지 못했던 일이 비일비재하게 발생하는 상황에서는 이런 관리가 더 필요하게 된다. 초반의 부족한 계획을 좀 더 현실에 맞추고 발생할지도 모르는 위협(Risk)들에 대해서도 지속적인 대응방안을 마련해 나가야 한다. 따라서 이런 모든 과정이 없이는 "건강한" 과제를 유지하는 것은 거의 불가능하며 현재 과제가 "건강하게" 진행 중인지에 대한 평가도 할 수 없다. 기준은 초반의 계획에 대한 달성 정도로 관리 목표를 세울 수 있지만, 고민 없이(물론 개별 과제는 많은 고민을 하고 계획되었겠지만, 사실 이런 부분은 항상 과제의 진행 과정에서 변화되기 마련이다.) 만들어진 계획을 지속해서 유지하는 것은 별로 의미가 없다. 첫 과제에 대한 현황 평가와 동시에 다음 마일스톤에 대한 계획은 수정을 동반할 수밖에 없다.

지속해서 관리한다는 말은 지속적으로 현실의 상황을 반영하기 위해서 변경된다는 뜻이다. 하다못해 핸드폰도 지속적인 펌웨어 업데이트가 있는데, 자신의 고과에 관련된 것에 대해서도 충분히 관리해야 하지 않을까? 요구사항도 바뀌며 현실에서 동원할 수 있는 자원도 변경된다. 이렇게 변경된 계획에 대해서는 과제의 이해관계자들과 같이 나눌 수 있어야 한다. 하지만 현실은 과제 일정 변경과 같은 말을 꺼내기도 힘들게 만든다. 만약에라도 일정 지연이 예상된다는 것과 같은 보고는 윗사람의 질책을 각오해야 하는 대표적이 경우다. 정당한 이유라도 만들지 못하면 책임은 오로지 과제의 담당자들이 져야한다. 정당한 이유가 없다면 정당하지 않은 이유라도 정당하게 만들어야 할 필요까지 느끼게 만든다. 실제로도 정확한 일정을 과제의 초반에 만들 가능성은 거의 없으며, 최소 25%에서 최대 400%까지 다양하게 과제 일정 지연(혹은 단축)은 발생할 수 있다. 운이 좋다면 빨리 끝낼 수 있을지도 모르지만 그런 경우는 거의 없다고 봐야할 것이다. 대부분은 지나치게 짧은 일정과 구현해야 할 많은 기능들을 요청받기 때문이다. 변경되지 않는 계획은 계획이 아닌 부담일 뿐이다.

위와 같은 문제가 인식되는 순간은 대부분의 경우 시스템 테스트에 들어가는 때다. 왜냐하면, 그 때까지는 내부에서 그냥 침묵을 지킬 수 있는 상황이지만, 다른 팀(테스트 팀)이 관련되는 순간 모든 것이 백일하에 밝혀지기 때문이다. 이 상황에서는 윗사람에게서 듣는 이야기가 "도대체, 과제를 어떻게 관리해서 이 지경이냐?"는 말일 것이다. 하지만 사실 그 상황이 오기 전에 이야기할 수 있는 분위기를 만들었는지에 대해서 “양심의 가책”을 받아야 할 사람은 개발자가 아닌 과제를 그렇게 추진하게 만든 사람들이다. 개발자들은 대부분의 경우(물론 예외도 있지만) 자신이 하는 일에 대해서는 솔직하게 이야기하는 경향이 있다. 거짓말을 하지 않는 것이 그들의 기본 덕목이기 때문이다. 초기 계획을 부실하게 세우도록 시간을 주지 않았던 점, 과제를 턱없이 부족한 일정으로 하라고 지시한 점, 개발자들의 의견을 묵살(무시)한 점, 제대로 소프트웨어 개발을 알지도 못하면서 마치 다 아는 듯이 이런 저런 지시로 과제 진행을 방해한 점, 팀워크를 해치려고 했던 이런 저런 활동들, 많은 보고를 하게 만들어 실무 관리자들이 자기 일에 몰입하지 못하게 만든 점 등등 다양한 잘못을 했다는 것을 윗사람들은 꼭 알아야 한다. 반성은 잘못된 결정을 한 사람들이 해야지, 그것을 따르고 행동으로 옮긴 사람들이 해야하는 일이 아니다.

# [ 생존전략 : 절망을 받아들이는 방법 ]

소프트웨어 프로젝트를 제대로 완료하는 데 필요한 전략을 "생존전략"이라고 부르는 책(“Software Project Survival Guide”, 스티브 맥코넬 지음)이 있다. 내용은 그냥 접어두고 책의 제목만 보자면 이미 소프트웨어를 개발하는 것 자체가 생존과 관련된 일이 되어버렸다는 것을 깨닫게 된다. 실패한 과제의 담당자는 회사에서 쫓겨나 가족의 생계를 걱정해야 할 신세가 되어야 하는 것일까? 아니면 소프트웨어 과제의 성공이 우리 생명을 걸고 해야 할 정도로 위험이 많은 일인가? 만약 위험 부담이 그렇게 큰일이라면 생명 수당도 우리의 월급에 포함되어야 할 것이다. 가족의 미래를 위한 보상도 커야 하고 과제 이외의 일에서는 참여하지 않아도 되는 예외를 인정해 주어야 한다. 왜냐하면 우리는 생존보다 더 절실한 문제는 세상에 존재하지 않기 때문이다.

생존을 위해서는 반드시 역할과 그에 따르는 책임이 명확해야 한다. 자신의 생존이 남의 손에 달려있다면 당연히 위협에 대해서 아무런 조치를 취하지 않을 것이 분명하기 때문이다. 따라서 과제를 진행하는데 필요한 결정 권한이 과제 실무 담당자에게 확고하게 주어져야 하며, 그에 따른 결과도 책임지도록 만들어야 한다. 사실 이 부분이 가장 힘든 것이기도 하다. 왜냐하면, 대부분 경영진은 권한을 주는 데는 인색하지만 책임을 묻는 데는 집요하기 때문이다. 이런 저런 결정에 대해서 개입하고 자신이 마음대로 결정을 내리지만, 그에 따른 책임은 다른 사람에게 미루는 것이 그들의 속성이다. 어찌 보면 회사 내에서 정치적인 생존을 위한 자신만의 "Know-How"일지도 모르지만, 당하는 입장에서는 "책임 회피"로 밖에는 보이지 않는다. 그리고, 해당 분야의 전문가가 아닌 이상 모든 것을 구체적으로 알기는 불가능하다. 실무자에게 최대한 결정권을 이양하는 것이 그분들에게도 분명 도움이 될 것이다. 양심적인 개발자는 자신이 결정한 부분에 대해서는 책임을 지는 것이 도리라고 생각하기 때문이다. 그리고 대부분의 개발자는 양심을 져버리지 않기 때문이다. (어떤 조직이라도 예외는 항상 있다.)

생존하기 위해서는 꼼꼼한 관리가 필요하다. 자신의 생명을 담보로 하는데도 그렇게 하지 않을 사람은 없을 것이다. 소프트웨어 개발에서의 “꼼꼼한 관리”란 모든 개발 활동에 대한 "가시성"을 확보하기 위한 노력이다. 과제의 목적이 무엇이며, 언제까지 어떤 일을 해야 하고, 어떤 것들을 관리해야 할 것이며, 어떻게 관리할 것인가를 정하는 것이다. 관리의 대상은 다양하지만, 주로 과제를 하는데 필요한 활동들에 대한 것만을 고려해야 한다. 회사의 정형화된 프로세스가 과제의 특성마다 변경될 필요가 있는 이유도 마찬가지다. 바쁘다고 대충하고 넘어가선 안 되는 일들은 반드시 사전에 정의되어 있어야 하고, 과제의 이해관계자들이 함께 점검을 해야 한다. 이것을 통해서 과제가 어떻게 진행되고 있는가를 정기적으로 관련자들이 다 같이 공유하고 고민해 볼 수 있기 때문이다. 변경은 과제의 현 상황에 맞게 이루어져야 한다. 과제의 변경 사항들도 갑자기 하늘에서 뚝 떨어지는 최고 우선순위의 일이 아니라 합의된 일정에 대한 영향을 고려해서 언제 반영될지를 결정해야 할 것이다.

반드시 해야 할 중요한 일은 항상 먼저 해야 한다. 뒤로 갈수록 시간이 부족한 것이 소프트웨어 개발과제의 특성이다. 나중에 하면 될 거라고 생각한 모든 일은 나중에 절대 하지 못하게 되거나, 과제의 심각한 지연을 유발하게 된다. 그렇다면 반드시 해야 하는 일이라는 것은 누가 정하는가? 판단 기준은 당연히 "고객의 입장"이다. 윗사람의 생각은 "고객의 입장"중에서 극히 일부만을 반영할 뿐 실질적인 고객은 아니다. 모든 입장을 다 고려하면 과제는 제대로 진행될 수 없다. "고객의 입장"에서 상충하는 것들은 트레이드 오프(Trade-Off)가 필요하며, "고객의 입장"에서 반드시 필요한 기능을 먼저 구현하는 것이 생존 가능성을 조금이라도 높여줄 것이다. 최소한 그것만이라도 배포할 준비를 완료할 수 있다면 고객은 기다려줄 가능성이 높다. 나중에 추가적으로 우선 순위가 높은 기능들을 꾸준히 배포해 줄 수 있다면, 적어도 고객의 신뢰를 잃지는 않을 것이다. (이때도 마찬가지로 "Simple is The Best"라는 것이 진리다.)

# [ 시작이 반이다. ]

우리가 잘 아는 말 중에 "시작이 반이다"라는 속담이 있다. 소프트웨어 개발도 예외가 아니다. 잘 준비된 시작을 하게 되면 과제는 성공할 가능성이 커진다. 물론 좋은 시작이 전부는 아니기에 나머지 반을 찾는 활동을 개발 기간 동안 꾸준히 해야 한다. 불행하게도 우리는 그동안 제대로 된 시작을 맛보지 못한 것이 사실이다. 그리고, 앞으로도 제대로 된 시작을 할 기회가 자주 있지 못하리라는 것도 잘 알고 있을 것이다. 과제의 일정은 정확한 추정으로 받아들일만 한 것이 아니라, 정치적인 상황이나 시장 상황이라는 말이 덧붙어서 항상 "과감한 도전 정신"을 발휘하라는 요구와 함께 무리하게 주어지기 때문이다.

이런 상황이라면 과제에서 반드시 해야 할 일도 "과감하게 도전적으로" 잘려나가게 된다. 제대로 된 코딩을 하지도 않게 되며, 대충 요구되는 기능이 구현되었다면 테스트도 적당히 하고 넘어가게 된다. 당연히 문서 같은 귀찮은 것들은 생략하는 것이 “도전적”으로 맞다. 설계하는 것도 사치다. 설계에 대서 리뷰를 하는 것은 거의 불가능한 일이며, 전체적인 과제의 개발 규정이나 문화 같은 것은 "생존 문제"를 해결하기 전에는 절대 이야기하지 않을 것이다. 일일 빌드나 자동화된 테스트 따위(혹은 단위 테스트)는 한방에 모아서 나중에 하는 전략으로 바뀐다. 일단 코딩하고 나중에 다 만들어진 후에 통합하고, 그 때 가서 테스트하면 될 일을 미리 걱정할 이유가 없다. 당연히 사용자에게(고객에게) 중간에 보여줄 결과물은 “PPT”로 정리한 과제 진행 현황과 대략적으로 기술한 WBS(Work Breakdown Structure)의 "몇 % 완료"정도가 전부일 것이다. 실제로 동작하는 것은 아무것도 없지만 과제 진척률은 계속 높아질 것이다. 하지만, 절대로 95% 이상의 완료율은 90% 다음에 한참이 지나도록 도달하지 못할 것이다.

이런 현상이 발생하는 이유는 해야 할 일을 뒤로 미루고 하지 말아야 할 일을 했기 때문이다. 즉, 일종의 "기술적인 빚(Technical Debt)"이 누적되어, 나중으로 갈수록 더 어려운 일들이 산더미처럼 밀려들 것이다. 예를 들어, 코드가 단위 테스트가 되지 않은 상태에서 다른 코드와 통합(Merge)이 일어나면, 코드의 복잡도는 지수적으로(Exponential) 증가한다. 당연히 그렇게 복잡한 코드를 디버깅 하는 시간은 더 오래 걸릴 수밖에 없고, 가독성을 생각하지 않고 작성한 코드는 읽는 사람이 이해하는데도 더 많은 시간을 잡아먹을 것이다. 제대로 설계되지 않은 코드는 성능 문제로 인한 구조 변경을 방해할 것이며, 테스트하는 중간에 최적화 작업도 병행하게 될 것이다. 당연히 버그는 줄어들지 않고 지속적으로 늘어갈 것이고, 성능 개선은 10%에 도달하기도 힘든 상황이 될 것이다. 빚이란 갚아나가지 않는 한 지속해서 누적되며, 나중에는 아무리 많은 인력을 추가한다고 해도 결코 높은 수준의 품질을 기대할 수 없다. 이쯤 되면 소위 임원이라는 사람이 와서 한 마디 할 때가 된다. "왜 우리가 만드는 소프트웨어는 문제가 많죠?" 대답은 아주 간단하다. "당신이 그렇게 만들라고 했기 때문입니다."

그 때서야 자신의 책임을 회피하려는 임원은 반드시 "시작이 반이다."라는 말을 기억해야 한다. 소프트웨어 개발은 모르더라도 제대로 된 준비를 하고 있는지는 자신이 직접 챙겼어야 했다. 결국 임원은 자신의 밑에 있는 사람들이 성과를 제대로 만들어야 회사에서 살아남을 수 있다. 준비할 시간도 제대로 챙겨주지 못하는 무능함과 책임 전가형 임원은 자신의 위치에서 해야 할 일을 제대로 하지 못한 것이다. 이것이 과제의 실패로 연결되고 비용(Cost)이 이윤(Profit)을 넘어서 적자 상태인 회사를 만들게 된다. 한 사람의 관리자를 잘못된 자리에 앉혀서 개발팀의 생산성을 극적으로 떨어뜨리게 되면 관리자 자신만 망가지는 것이 아니다. 회사에 다니는 다양한 사람들이 회복할 수 없는 상처를 받을 수 있다. 적어도 과제를 실패로 몰아가는 결정이 어떤 것인지 정도는 알아야 할 것이다. 과제의 시작에 필요한 것은 "과감한 도전 정신"이 아니라, 철저히 계산된 계획이며 남들이 이미 성공한 경험으로 토대를 만들어 놓은 공학적인 방법임을 개인 공부를 해서라도 알아야 한다.

# [ MDD(Meeting Driven Development) ]

보통의 경우라면 “MDD”를 “모델 주도 개발(Model Driven Development)”이라고 했을지도 모르겠지만, 과제가 수렁 속에 빠져들어 힘든 상황이 되면 다른 뜻의 말이 된다. 불필요하다고 생각되는 회의가 과제의 심각한 방해 요소가 되기 때문이다. 대부분의 경우 과제에 관심을 극도로 많이 가진 임원이 과제의 현황을 보고하라는 요청으로 시작된다. 하지만, 이렇게 시작된 정기적인 미팅은 지속해서 증가하고, 일일 보고의 형태로 아침 저녁 시간을 이용해서 집요한 질문 공세로 이어지게 된다. 만약 하나라도 놓치게 되면 거친 비난(비판)과 숙제를 동반한 폭풍이 몰아치게 된다.

미팅으로 과제가 완성될 것 같으면 회의만 하는 것도 좋은 방법이다. 하지만 우리가 만들고 있는 것은 고도의 집중력과 세밀한 계획을 통한 지속적인 정신노동을 요구한다. 숙제를 시키면 숙제를 하는 사람이 있을 것이고, 십중팔구 실무 담당자가 그 숙제를 도맡게 될 것이다. 개발 과제의 지연 이유는 개발자가 개발에 집중하지 못하는 것인데, 이런 식으로 지속적인 제동(Interrupt)이 걸리는 상황 속에서는 지연이 늘어나면 늘어났지 줄어들 가능성은 없다. 높으신 분들을 회의를 통해서 자신의 무지함과 불안감을 달랠 수 있겠지만, 개발자는 자신이 만든 결과물에서 자존심을 찾아야 한다. 과제관리 시스템을 구축하고 있는 경우에도 임원들은 직접 불러서 과제의 현황을 확인하기를 원하며, 자신이 만든 검증되지 않은 방법론을 통해서 과제를 진행하려고 애쓴다.

제발 개발은 개발자에게 맡겨주기 바란다. 그렇다고 모르면 가만 있으라는 이야기가 아니다. 개발자가 일을 잘 할 수 있는 최적의 환경과 여분의 시간을 만들도록 하는 것이 높으신 분들이 해야 할 일이다. 많은 숙제를 하라고 시키는 것은 숙제가 과제보다 더 중요하다는 의미 밖에 되지 않는다. 미팅이 늘어날수록 생산성은 떨어진다. 단순히 미팅 시간만이 개발 시간에서 빠지는 것이 아니라, 그 미팅을 준비하는 시간과 미팅의 결과로 인한 과제에 대한 변경 발생이 지속해서 과제가 산으로 가도록 만든다. 과제 담당자의 권한은 점점 약해지고 모든 것은 미팅의 주재자가 알아서 결정할 것이 분명하다. 높으신 분들이 생각한 것이 그대로 구현되는 세상에서 살고 싶은 것은 개발자도 마찬가지다. 하지만 누군가는 흙탕물에 손을 담그고 지져분해질 것을 각오하고 치열하게 살아가야 한다. 말씀으로 모든 것을 만들었던 것은 태초의 하느님 밖에 없다는 사실을 높으신 분들도 깨닳아야 할 것이다.

“MDD”라는 말이 이렇듯 우스개 소리처럼 들리는 순간 이미 고상한 놀이의 시간은 다 지나간 것이다. 이제부터는 적극적으로 중요하지 않은 일들은 다 잘라야 한다. (하지만, 사실 잘라내기도 쉽지 않다. 이미 구현한 코드에서 우선순위가 낮다고 생각되는 코드를 제거하는 것보다 새로 짜는 것이 나은 경우도 있다.) 중요한 일은 먼저 해야 한다. 중요한 일은 반드시 해야 할 일이고, 반드시 해야 할 일은 과제의 시작에서 할 일들이다. 그리고 반드시 해야 할 일들은 과제 기간동안 지속해서 해야 할 일이다. 과제에서 반드시 필요한 일들은 한 번에 완료가 되는 것들이 아니다. 지속적인 변경과 수정이 발생하는 일이며 적절한 통제 속에 이뤄져야 하는 것들이다. 제대로 하지 못한 일들은 결국 제대로 할 시간을 과제의 후반으로 갈수록 더 많이 요구하게 되고, 결국 예상했던 일정은 공허한 약속으로 변하게 된다. 말은 그럴듯하게 할 수 있을지도 모르지만 모든 가치있는 일을 일루는 것은 누군가의 진정한 노력이다.

# [ 과제 일정 준수는 가능한 일인가? ]

소프트웨어 개발 과제를 담당한 경험이 있는 사람들은 한결같이 이야기하는 것이 있다. 과제 일정에 대한 예측이 정말 어렵다는 것이다. 큰 회사에서 일하는 사람들은 한 해를 마무리하기도 전에(이미 시작되어 진행 중인 과제를 완료하기도 전에) 다음 해에 해야 할 과제를 기획하게 된다. 지금 하고 있는 과제와 큰 차이가 없을지는 몰라도 새로운 기능과 새로운 하드웨어, 새로운 UI등을 적용해서 새로운 제품을 만들어내야 한다. 이런 경우 대략 기존의 경험을 이용해서 얼마나 걸리게 될지를 예상하게 되며, 그 예상기간에 2개월 정도의 버퍼링(Buffering) 기간을 설정해서 과제 계획서를 작성하고 보고하게 된다. 그렇게 보고된 계획은 당연히 한 번에 통과되지 않고 수차례에 걸쳐서 상위 관리자의 리뷰 후에 단축된 기간(일명 “도전적인 일정”)으로 다시 돌아온다.

"아니요"라는 대답은 하기 힘들다. "아니오"라고 대답하는 순간 "무능력자"라는 의미의 "도전 정신이 없다"는 이야기를 상사로부터 들어야 하고, 매출이 떨어지는 게 개발에서 제대로 일정을 맞추지 못하기 때문이라는 소리를 상품기획이나 마케팅 부서로부터 들어야 한다. 하지만, 동의하지 않은 일정에 대해서 책임은 언제든 항상 개발자의 몫이 된다. 그렇게 해선 안 된다는 것을 알고는 있지만, 일종의 분위기를 이용해서 개발자에게 책임을 지우게 되는 것이다. 위에 있는 상사는 아무런 잘못이 없다는 듯, "당신들이 동의해서 그렇게 정한 것 아니냐"고 책임을 회피한다. 이런 상황까지 오면 두 번 다시는 그런 과제를 책임지는 역할은 하고 싶지가 않고, 도전적인 정신은 거짓으로 주어진 책임감일 뿐이라는 것을 알게 된다.

소프트웨어 과제에서 일정을 준수해서 완성된 과제들은 대부분 과제 일정에 여유가 많았거나, 작은 크기의 과제일 확률이 높다. 단순 기능을 구현하는 제품을 만드는 것은 이미 만들어 본 경험이 있을 때 대체로 정확히 일정을 맞출 확률이 높다. 하지만, 규모가 큰 과제의 경우에는 10명 이상의 인력이 투입되고, 10만 라인 이상의 크기로(혹은 100만에서 200만 라인 이상) 커질 수 있으며, 이럴 경우에는 과제 일정을 맞출 확률이 25%도 되지 못한다. 거기에 개발자가 동의하지 못한 "도전적인 일정"을 가지고 이야기를 한다면, 아마도 일정 준수 확률은 10% 미만이 될 것이다. 한 가지 더 보태서 관리자가 쓸데없는 일로 숙제를 지속해서 안겨준다면, 다시 일정준수 확률은 5%가 체 안 될 것이다. 점점 실제 과제에서 벗어난 일을 하는 경우가 많아질수록 과제는 지연될 확률이 높아진다. 하지만, 과제 지연이 예상된다고 보고하는 순간부터 이상하리만치 많은 숙제를 내는 것이 소프트웨어 개발 경험이 없는 관리자들의 특징이다. 아마도 이것은 잘못된 학습(오랜 악습)의 결과일지도 모른다.

과제 일정의 준수는 사실상 만족시키기 어려운 목표다. 줄어든 과제 일정은 줄어든 기능과 성능만을 확인할 수 있다. 따라서 현실과 타협을 시도할 수밖에 없는 과제가 되고, 그 상황에서도 사용자가 만족할 만하고 유지보수가 가능한 코드를 만들어나가는 것이 소프트웨어 개발자가 할 일이다. 과제 일정이 줄어들었다고 해서 필요한 조치들을 생략해서는 절대 안 된다. 결국 그렇게 쌓인 빚들이 언젠가는 재앙으로 돌아올 것이기 때문이다. 줄어든 일정을 가지고 할 수 있는 일의 범위를 한정시키고, 이것을 과제 관련된 사람들이 모두 다 공유할 수 있도록 해야 한다. 추가된 기능을 구현할 시간이 없다면 기존 기능이라도 개선하는 방향으로 가야 한다. 기존 기능을 개선하지 못한다면 UI라도 변경하는 것이 보기는 좋을 것이다. 사람을 더 추가한다고해서 지연된 과제 일정을 당길 수는 없다. 그리고, 생략된 과정에 대한 변명은 수준 낮은 과제의 결과물로 해결되지 않는다. 결국 모든 책임은 개발자의 몫이 되고, 누구도 그런 결과물에 대해서 자신의 책임이라고 인정하지 않을 것이기 때문이다. "아니오"라고 이야기할 수 없다면, "네, 하지만 이런 것들은 안 돼요."라고 이야기하는 것이 최선일 것이다.

# [ 외주 + 관리 ]

소프트웨어 업게에서 바쁜 회사는 외주를 많이 활용한다. 인력적으로 부족한 부분을 메우기 위해서 외주업체를 고용하지만, 바쁘다는 핑계로 그냥 일을 맡기기만 하고 제대로 된 관리를 하지 않는 것이 문제다. 관리하고 있다고 말하는 것들도 대부분은 일정 및 결과물을 챙기는 것으로 한정되어 있어서, 과제를 마칠 시점이 되면 서로 간에 바라는 바가 달라서 여러 가지 문제를 일으킨다. 그리고, 외주업무 자체가 돈이 오가는 거래라 오래된 외주 거래처와는 비리가 발생하기도 한다. 소프트웨어 외주의 더 심각한 문제는 나중에 생기는 유지보수에 대한 비용산정이 제대로 되기 어렵다는 점이다. 소프트웨어 개발은 전체 생명주기(Life Cycle)에서 유지보수가 비용의 40~70%를 차지할 수 있기 때문이다. 유지보수를 위해서는 다시 이전에 외주를 진행했던 회사와 계약을 하던가 유지보수를 포함해야 하지만, 결국 외주 회사도 그러한 부분에 대한 인력지원이 여유 있는 부분은 아니다.

외주를 주기 위해서는 관리를 어떻게 해야 하는지 알아야 한다. 외주를 활용할 때는 핵심이 되는 부분은 외주하지 않는 것이 당연하지만 때때로 이런 부분이 무시된다. 외주 하더라도 기존 소프트웨어 개발자가 과제의 크기에 따라 몇 명은 100% 할당이 되어야 한다. 하지만, 대부분 관리자는 이런 부분을 몰라서 그냥 한 명이 여러가지 외주 과제를 진행하면 되는 것으로 생각한다. 외주 관리쯤은 아무것도 아닌 일로 취급해 버리는 것이다. 소프트웨어 개발 외주는 일반적인 과제 개발과 다르지 않다. 개발해야 할 내용을 사전에 잘 파악해야 하고(정의해야 하고), 세부적인 설계에 대한 평가와 지속적인 코드에 대한 검토 및 관리가 수반되어야 한다. 또한 적절한 테스트 케이스를 외주 회사와 같이 발굴해야 하고, 코드의 품질에 대한 부분도 세밀하게 관리해야 한다. 각종 툴 및 개발환경과 자동화에 대한 부분도 개발 초기부터 신경 써야 한다. 이런 일들을 제대로 하기 위해서는 소위 말해서 전문가 수준의 관리가 이루어져야 하며, 단순히 소일거리 정도로 생각해서는 제대로 된 결과물을 만들어내지 못할 것이다. 외부 결과에 대해서는 담당자만이 아니라 관련된 모두가 책임을 함께 가져야 한다.

내부에서 개발하는 과제보다 더 많은 관리가 필요한 것이 외주개발 과제다. 가능한 최고의 인력이 담당해야 제대로 된 성과가 나올 수 있다. 외주로 개발된 과제의 경우 지속적인 유지보수가 안되는 경우가 많은데, 이는 업무를 담당했던 인력이 외주를 제대로 진행하지 않아서 발생하는 문제다. 제대로 했다고 하더라도 담당자가 없거나 다른 업무로 바쁘더라도 유지보수는 어렵게 된다. 그렇게 제대로 관리하지 못하게된 원인을 제공한 사람도 같이 책임을 져야 한다. 100% 자신의 역량을 투입해야 함에도 불구하고 이런 저런 일을 같이 시켜놓고는 왜 잘 못 관리했는지 책임을 묻는다면, 제대로 책임질 수 있는 사람이 몇 명이나 있을까? 사실 외주관리를 제대로 된 업무로 생각하는 사람도 없는 것이 현실이며, 유지보수의 시절이 찾아오게 되면 결국 과거의 진행 과정을 따지게 된다. 만약 위에서 말한 소프트웨어 개발과제의 생애주기(Life Cycle)에서 중요한 핵심 업무를 제대로 진행했다면 어떨까? 내부에서도 하기 어려운 업무를 외주에서 다 하기는 정말 힘들지만, 그래도 그렇게 관리했다면 최소한 없는 것보다는 나을 것이다.

외주의 다음에 오는 "관리"라는 단어를 잊지 말아야 한다. 관리는 지속적인 업데이트를 하고 있다는 말이며, 관리되고 있다는 것은 누군가의 노력이 꾸준히 투자되고 있다는 뜻이다. 관리되는 상황은 품질의 상한과 하한값이 있다는 이야기이며(품질의 본질이 측정하기 어려울 때는 외적으로 드러나는 수치들이 이용될 수 있다.), 이들 값을 정해진 범위 내에서 지속적으로 유지하려는 노력이 필요하다. 따라서, 외주는 외부에서 개발할 뿐이지 내부에서 개발하는 것과 크게 다르지 않으며, 고객의 관점을 지속해서 외주 업체에 제시해야 한다. 추가적으로 단순히 고객의 입장 만을 생각하는 것이 아니라 향후 유지보수를 위해서 필요한 조치들도 취해야 한다. 예를 들어, 자동화된 테스트와 구조에 대한 설계 및 검토 활동, 코드에 대한 검토는 필수다. 이런 활동들이 없다면 결국 외주 결과물은 쓰레기가 되어가고, 앞으로 더 큰 비용을 들이더라도 개선되지 않는 결과를 낳게 될 것이다. 모든 소프트웨어가 지속적으로 진화(Evolution)를 거친다고 가정할 때, 진화가 더딘 소프트웨어는 결국 경쟁에서 도태된다. 외주 비용을 들인다고 해서 모든 것이 해결되는 것이 아님을 잊지 말아야 할 것이다.

# [ 창의력과 규율의 사이에서 ]

소프트웨어 개발자들은 자신들이 창의적인 일을 하고 있다고 믿는 성향이 있다. (혹은 창의적인 사람이라고 생각한다.) 하지만, 그들이 하는 과제를 관리하는 사람들은 그들을 정해진 틀에서 일률적으로 통제하려고 한다. 창의적인 인재상을 이야기하면서도 정해진 틀을 벗어나지 못하게 하는 것이 회사의 정책이다. 방향을 제시하지만 그 방향으로 가는 방법들은 창의적으로 찾기를 원한다. 소프트웨어 개발 과정도 마찬가지다. 창의적으로 문제의 해결 법을 찾아가지만 정해진 프로세스를 따라야 한다. 프로세스가 오버헤드가 된다고 생각하고 소프트웨어 개발자의 창의성을 떨어뜨린다고 보는 것은, 문제에 대한 해결책을 체계적으로 찾아가는 것을 거부하는 태도일 뿐이다. 이것은 틀(Template)과 내부를 채우는 내용(Contents)을 동일하게 보는 태도 때문이다.

틀은 필요하다. 틀은 문제를 더 쉽고 빠르게 접근하는 방법을 제공하기 때문이다. 틀이 없다면 시행착오를 해야 하고, 시행착오의 결과로 다시 새로운 틀이 만들어질 것이기 때문이다. 하지만, 이미 수없이 시행착오를 했다면 새로운 틀을 찾기 위해서 다시 시행착오를 할 필요는 없지 않을까? 소프트웨어 업계는 이미 오래동안 그런 경험을 축적해왔으며, 이제는 어느 정도 문제를 해결하는 방법(틀)을 갖추고 있다. 따라서, 이렇게 만들어진 방법을 거부할 이유는 없으며, 그런 방법을 어떻게 문제 해결에 적용할 수 있을 것인가를 고민하는 편이 옳을 것이다. 이런 틀이 소프트웨어 개발 프로세스라는 말로 표현된다면, 그런 프로세스의 산출물들은 창의적인 것들이 될 것이다. 산출물을 만들어내는 방법을 나열한 것이 틀이라고 한다면, 그것을 적용해서 만들어내는 과정에서는 창의적인 생각이 녹아들어 갈 것이다. 틀 자체는 모양을 유지할 뿐 내용을 충실하게 만드는 것은 개발자의 몫이다.

코딩도 마찬가지다. 코드를 만들어내는 과정에서 필요한 창의성은 요구사항을 설계로 변화시키는 과정과 설계를 실제 코드로 만들어내는 과정, 만들어진 코드가 제대로 동작하는지를 확인하는 활동들이다. 하지만, 그런 변환과 검증의 과정에서 필요한 것은 어떤 것이 더 바람직한 구조인지, 읽기 편한 코드인지, 만들어진 코드의 품질이 어느 수준을 만족하게 하는지 등에 대한 것은 정해진 틀에서 판단해야 할 부분이다. 이미 잘 알겠지만 소프트웨어의 품질은 직접적인 측정이 어렵기에 부가적인 결과물들을 이용해서 분석하게 된다. 예를 들어, 중복된 코드의 분량이 얼마나 되는지, 코드가 정해진 코딩 룰을 따르는지, 계층화 규칙을 위반한 호출은 얼마나 되는지, 자동화된 테스트를 통해서 검증되는 코드가 전체 코드에서 차지하는 분량은 얼마나 되는지, 코드의 복잡도(Complexity)가 지나치게 높은 부분은 얼마나 되는지 등을 통해서 파악하게 된다.

결국 창의력과 규율은 같은 곳에 머문다. 명작이라고 불리는 그림들을 보면 틀은 그림을 가리지 않는다. 틀에 들어간 그림은 작가의 창의력의 산물이며, 틀은 단순히 그림이라는 현실의 구현체를 벽에 고정하는 역할만 할 뿐이다. 소프트웨어 개발자가 가져야 할 역량은 창의력 만이 아니라, 정해진 규율을 현실의 업무에 적용하는 능력도 있어야 한다. 창의력이 돋보이려면 고정되어야 할 틀이 필요하며, 그 틀을 과제에 적당한 것으로 선택하면 된다. “RUP”, “Waterfall”, “Spiral” “Agile”등 어떤 틀이라도 상관없다. 자신이 선택한 틀이 자신이 해결하려는 문제에 최적이면 된다. 창의력과 규율은 서로를 돋보이기 위해서 필요한 것이지 반대되는 개념이 아니며, 지향성 없는 창의력은 기괴한 결과 만을 낳을 뿐이다. 지나친 규율 또한 창의력을 가리는 덮개가 될 뿐이며, 그런 것을 통해서는 뱔현되는 위대함 또한 없다. 장인의 손길은 항상 세심한 것에 머물지만, 세심한 것들은 항상 그것을 돋보이는 틀 위에 있기 때문이다.

# [ 좌로 이동(Left Shift) 전략 ]

소프트웨어 개발에서 "좌로 이동(Left Shift)"라는 말은 개발 프로세스가 왼쪽에서 오른쪽으로 흘러갈 때, 오른쪽에서 해야 할 일을 왼쪽에서 더 빨리 시작한다는 의미로 사용한다. 대표적인 경우가 테스트 단계에서 해야 할 일의 일부를 개발 단계에서 미리 실행하는 경우다. 하지만, 단어가 의미하는 바는 조금 더 심오한 뜻이 있다. 무조건 테스터들이 하는 시스템 테스트 수준의 일을 개발에서 실행하는 것이 아니다. 개발에서의 테스트는 당연히 단위 테스트, 통합 테스트 등이 있다. 이것을 개발에서 제대로 해야 한다는 뜻을 포함한다. 일반적으로 이와 같은 것을 잘 못 이해해서 무조건 개발이 테스터가 하는 테스트를 많이 하면 좋다고 생각하는 관리자들이 있다. 테스터의 테스트 케이스를 전부 가져와 일명 “자가 테스트”라는 말로 개발자가 실행하고 결과물을 넘겨주어야 한다고 주장한다. 물론 배포 전에 간단한 스모크 테스트(Smoke Test)정도는 개발자가 실행해도 좋지만, 그렇다고 모든 테스트 케이스를 개발자가 실행해야 한다는 논리는 아니다.

소프트웨어 개발자의 의무는 자신이 개발한 코드(남이 개발한 코드를 제외하고)에 문제가 없다는 것을 입증하는 것이다. 즉, 자신이 만든 증가분(Increments)에 대해서 주어진 목적을 완벽하게 수행하는지 실제 코드를 실행해서 증명해야 한다. 가장 좋은 검증 방법은 역시 실행뿐이기 때문이다. 그리고, 이런 증명은 코드를 통합해 나가는 과정에서도 필요하게 되며, 이를 위해서는 반드시 테스트 케이스를 실행할 방법을 미리 코딩해 두어야 한다. 따라서, 개발은 자신의 코드와 자신의 코드를 실행해서 테스트할 수 있는 코드까지도 만들어야 하며, 이를 이용해서 문제가 없다고 판단될 때(코드 리뷰도 포함해서) 저장소에 체크인(Check-in)을 할 수 있는 것이다. 이는 실제로 시간이 많이 소모되는 과정이며, 과제 리더나 관리자들이 느끼는 불안감이 가장 큰 부분이기도 하다. 구현해야 할 기능들은 많은데 쓸데없는 데 시간을 보내고 있다고 느끼기 때문이다. 그리고, 그렇게 해서 얻는 장점이 무엇인지도 이해하지 못하기 때문이다.

테스트된 코드와 그렇지 못한 코드의 품질은 차원이 다르다. (물론, 단순히 코드의 품질을 테스트만 따지는 것은 아니지만) 테스트를 언제든 자동으로 실행할 수 있다는 것과 모든 테스트를 개발자가 일일이 손으로 해야 하는 상황을 비교하면 결론은 자명하다. 그리고, 앞으로 있을 제품 개발에 재활용될 수 있는, 테스트되고 언제든 다시 테스트할 수 있는 코드를 가지고 있다는 것은 말 그대로 변화에 대한 안전망이 갖춰진 상태라는 것이다. 비용적인 측면에서도 테스트된 코드는 일정 계획을 예측 가능한 상태로 만들 수 있다. 일반적으로 테스트되지 않은 코드는 언제 버그가 발생할지 알지 못하기에 과제 완료 시점을 추정하기도 어렵우며, 얼마나 더 테스트해야 할지도 알지 못한다. 말 그대로 그냥 할 수 있는 만큼 테스트해서 문제가 더 없다면 출시한다는 말 밖에 되지 않는다. 이렇게 검증된 소프트웨어는 당연히 시장에서 좋은 평가를 받지 못한다. 왜냐하면, 필드(Field)는 개발자의 책상과는 다르기 때문이며, 어떤 환경에서 어떤 사람이 제품을 사용할지 아무도 예상하지 못하기 때문이다.

"좌로 이동(Left Shift)"은 개발자가 자신의 코드를 테스트하겠다는 의지를 갖추고 꾸준히 개선하는 시간이 확보되어야 가능한 일이다. 사실 대부분의 높으신 관리자는 하드웨어 관련 일만 해서 어떻게 소프트웨어를 테스트하는지도 모르는 경우가 많다. 그들이 지시하는 말을 그대로 하게 되면 개발만 더 지연될 뿐이다. 그런 지시대로 했는데도 제대로 일이 되지 않는다는 것을 알게 될 때 그들의 말은 바뀐다. "왜 시키는 대로 안 해?"라는 것이다. 시키는 대로만 했다면 결과가 좋을 것이라고 당연히 예측할 수 있겠지만, 그런 결과는 사람의 손으로 만들어지기 어렵다. 그들이 시키는 일은 최대한 많이, 최대한 열심히, 최대한 빨리 하는 것이기 때문이다. 그렇게 일할 수 있는 사람은 없다. 사람은 창조적인 일에 잘 적응하지 단순 반복적인 일에는 실수를 만들 가능성이 높다. 제대로 된 관리자라면 과제 기간에 덧붙여 개발의 각 단계에 필요한 것들이 무엇인지를 파악하고, 그것을 적극적으로 관찰하는 활동과 개발자들이 그 활동들을 할 수 있도록 환경을 만들어 줄 것이다. 목소리가 크다고(화만 낸다고) 좋은 관리자는 아니다. 작은 목소리라도 정확하게 이해될 수 있다면 충분히 힘을 가질 수 있다. 잘 모르겠다면 공부라도 하는 것이 좋다. 이미 당신이 좋아하는 "시장(Field)"에는 수많은 예제들이 널려있기 때문이다. 빨리 가고자 하면 목표는 더 멀어질 뿐이다.

# [ 코드의 공유 ]

낮은 수준의 개발자는 자신의 코드를 숨긴다. 조금 더 높은 수준의 개발자는 자신의 코드를 자랑하지만, 최고 수준의 개발자는 타인에게 도움이 되는 코드를 작성한다. 최소한 팀이나 회사 내에서는 그렇다는 뜻이다. 인터넷이나 블로그 등을 통해서 활동하는 많은 개발자도 있다. 그들은 자신이 겪은 문제를 해결하고 남에게도 알려준다. 왜? 이유는 단순하다. 그것이 먼저 가본 사람이 해야 할 의무라고 생각하기 때문이다. 그리고, 그것을 통해서 자신이 더 발전할 수 있다고 믿기 때문이다. 모든 문제를 혼자서 해결하는 것은 불가능하다. 그리고, 대부분 프로그램은 혼자서 개발하지 않는다. 많은 사람이 이미 이루어낸 것들을 기반으로 새로운 것을 조금 더할 뿐이다.

지식은 공유되어야 한다. 지식의 구현인 코드도 공유되어야 한다. 공유된다는 것은 남의 의견을 듣는다는 것이며, 타인의 의견을 받아들여 개선할 수 있다는 의미다. 회사 내에는 크게 두 가지 부류가 있다. 자신이 만든 코드를 보여주지 않으려는 사람과 적극적으로 남들의 의견을 청취해서 개선하려는 사람으로 나눌 수 있다. 전자의 경우에는 그럴듯하게 보이는 코드와 자신감이 뭉쳐져서 남들의 의견은 별로 도움이 안 된다고 믿는 사람들이다. 그들은 자신의 코드가 회사의 자산이라는 점을 머리 속으로만 이해하지 가슴으로는 느끼지 못한다. 또한 일에서는 자신과 코드를 동일시해서 코드에 문제점이 있다는 이야기를 받아들이지 못한다. 당연히 발전도 없다. 문제를 자신만이 해결할 수 있다고 믿고 있으며, 남들이 자신의 코드를 건드리는 것을 극히 싫어한다. 언듯 보면 뛰어난 능력을 갖추고 있으며, 이런 저런 숙제들도 곧잘 해온다. 문제는 그런 사람들로 구성된 조직의 효율성은 낮으며, 끝없는 문제로 유지보수의 어려움을 겪고 있을 가능성이 높다.

생산성이 높은 사람은 공유에 대한 마인드(mind)도 높다. 남들이 지적하는 문제에 대해서 솔직하게 인정하고, 개선점을 찾아서 더 좋은 방법으로 구현한다. 따라서, 같은 문제를 반복해서 일으킬 가능성이 낮으며, 다양한 지식에 대한 탐구의 끈을 놓지 않는다. 다양성에 대해서 저항하기 보다는 쉽게 받아들이며 끝없이 개선하려는 의지를 보인다. 이런 부류들은 소위 말하는 "장인"으로서 성장할 가능성이 높다. 그들은 자신이 만든 결과물로 인해서 혜택 받을 사람들을 상상하며 즐거워할 것이다. 이런 식으로 자신의 일에서 즐거움을 찾아나갈 것이다. 팀에서는 약간 소외되어 있을지도 모르고 인정받지 못할지도 모르지만, 회사의 입장에서는 반드시 필요한 인력들이다. 소프트웨어는 결국 "대화와 소통"이 최대의 자산이다. 대화와 소통 없이는 제대로 된 개발이 이루어지지 않는다. 고객과의 대화, 팀 내의 대화, 팀 간의 대화, 팀원과 부서장과의 대화 등을 포함해서 혼자서 하는 코딩에서도 대화는 끼어든다. 혼자 개발하더라도 코드를 작성한 자아와 그것을 읽고 있는 자아는 다를 수 있다는 뜻이다. 공개되고 공유된 코드는 항상 사람들과 대화를 유지하는 적극적인 방법인 것이다.

작은 차이가 모이면 크다. 소프트웨어 개발자들은 그런 작은 차이들을 쌓아나가는데 익숙해야 한다. 작은 수정과 작은 테스트가 모여서 100만, 1000만 라인의 코드가 나온다. "Big Bang"은 신이 우주를 창조하는 과정에서만 나오지 소프트웨어 개발에서는 절대 나와선 안되는 방법이다. 작은 변화들을 끝없이 더 좋은 방향으로 추구하는 태도가 소프트웨어 개발자의 기본기다. 더 좋은 것이 있을지 모른다는 막연한 기대를 그냥 접어두지 않고 남들의 의견을 적극적으로 들어보는 것은, 마치 집단이 가지는 지식을 활용해서 더 좋은 것을 추구하고 발전시켜나가는 것과 같다. 이는 이미 "오픈소스(Open Source)"라는 형태로 우리 소프트웨어 개발자들의 생활에 중요한 일부분이 되었으며, 반대로 가는 것은 흐름을 놓치고 스스로 홀로 고립되는 길일뿐이다. 자아와 자신의 코드와 구분하지 못하는 어떤 친구가 내게 물었다. "진급했는데 이제 어떻게 하죠?" 짧은 답으로 마무리했다. "너가 했던 것을 남들에게 줘. 그리고, 그들이 잘 할 수 있도록 도와줘. " 그것만이 그 자신과 다른 사람들을 위해서 더 좋은 미래를 만들어 갈 수 있는 방법이라고 생각했기 때문이다.

# [ 일정 압박을 통해서 얻는 것은 없다. ]

소프트웨어 개발의 지상 최고의 목표는 "고객이 원하는 소프트웨어를 정해진 일정과 비용으로 출시"하는 데 있다. 그리고, 그러한 목표는 항상 일정과 비용, 품질이라는 함께 다 이룰 수 없는 딜레마 속에서 해답을 찾으려고 시도하도록 만든다. 일정을 줄이라는 압박은 고객이 원하는 기능을 제외하거나 낮은 품질이라도 제품을 출시하라는 말과 동일하다. 관리자들은 일정을 줄이면 다른 것들이 나빠진다는 사실을 도외시한다. 왜냐하면, 그들의 재계약은 제품의 출시 시기에 있지 그 제품이 제대로 기능하고 품질이 우수한지에 달려있지 않기 때문이다. 높으신 관리자들은 자신의 목숨줄이 왔다 갔다 하는 판에 이런 “고상한 것”들을 따질 여유가 없다. 하지만, 그렇게 출시된 제품은 시장의 냉정한 평가 속에서 살아남기 어려울 것이다. 그리고, 연장된 재계약을 통해서 이를 보완하려고 하겠지만 이미 때는 늦게 된다. 시장은 그 제품에 대해서 알 만큼 알았고, 보완하려고 하는 노력(비용)은 단축한 개발 기간에 벌어들인 수익보다 크게 될 것이다. 소프트웨어는 생각보다 결코 "소프트(Soft)" 하지 않기 때문이다.

소프트웨어를 변경하기가 쉽다고 생각하는 사람들은 소프트웨어 개발자들의 노력에 감사해야 한다. 100층짜리 건물을 지으면서 1층을 완전히 새로 짓는 선택은 하지 않지만, 소프트웨어 게빌에 대해서는 이런 결정을 요구한다. 하지만, 나머지 99개 층의 하중을 떠받들 설계를 다시 하기는 쉽지 않다. 건물은 각 층이 위층과 아래층으로 나누어지고, 아래로 갈수록 더 튼튼하게 만든다. 하지만 소프트웨어는 지하에서 시작해서 100층까지가 유기적으로 연결되어 한 층의 변경이 다른 99개 층에 영향을 줄 수 있다. (물론, 그렇게 코딩하는 것은 권장 사항은 절대 아니다.) 소프트웨어를 변경하기 위해서는 단지 5%의 코딩이 필요하지만, 나머지 95%의 생각들이 있어야 가능하다. 오랜 경험과 분석, 그리고 변화에 대한 적절한 대응조치가 필요하다는 것이다. 하지만, 대부분 개발자는 전체 시스템을 이해하지 못한다. 잘 해야 한 두 명 정도의 개발자만이 전체 시스템을 이해할 뿐이다. 이런 상황에서 모든 것을 통제하에 두기는 지극히 어렵다. 일은 분업화되고 세부적인 것은 각 개발자의 몫이 된다. 리뷰를 통해서 어느 정도 공유는 가능하지만 직접적인 코딩을 통해서 구현하고 있는 개발자와 이해의 수준이 다를 것이다.

일정과 품질, 기능을 다 잡으라고 이야기하는 관리자가 있다. 하지만, 그게 가능한 일이라면 그 사람이 직접 개발을 해야 할 것이다. 왜? 불가능하기 때문이다. 그런 불가능한 것을 지시하는 관리자가 있다면, 그 사람은 소프트웨어 개발을 직접 해본 경험이 없거나 작은 과제만 해본 사람일 것이다. 작은 과제는 대부분 5명 미만의 인력으로 완수되는 짧은 기간동안 완료할 숭 있는 기능도 몇 가지 안되는 과제다. 이때는 개개인의 역량의 차이가 확연하게 드러난다. 과제의 크기가 커질수록 각종 오버헤드는 더 늘어난다. 이런 과제는 사람을 더 많이 투입한다고 기간이 줄어들지 않는다는 이야기다. 서투른 관리자는 일이 잘 안되면 더 많은 사람을 투입하라고 이야기한다. 하지만, 더 많이 투입될수록 더 늦어질 것이다. 당연한 이야기지만 소프트웨어 개발은 해당 도메인(Domain) 지식을 익혀가는 과정이다. 새롭게 투입된 인력들도 그 과정을 반복해야 한다. 그 반복 때문에 이미 투입된 개발자들에게는 도움이 되지 못한다. 가능하다면 과제 초반부터 같이 투입되었어야 한다. 사람을 달라는 요청에 대해서 전혀 도움이 안되는 신입사원이나 외주인력을 아무렇게나 배치하는 관리자가 있다면, 그 어리석음에 대해서 아무런 대책이 없는 사람이다. 그 사람도 그런 경험을 자신이 했으리라 생각하지만, "개구리 올챙이 시절 모른다"라는 속담과 같이 똑같은 실수를 반복하고 있는 것이다.

일정의 압박은 결과적으로 서툰(품질이 열악한) 소프트웨어를 만들어낼 뿐이다. 그리고, 그로 인해서 발생하는 각종 비용에 대해서는 아무도 책임지려고 하지 않을 것이다. 그런 과제를 맡은 PL(Project Leader)나 PM(Product Manager)들은 슬픈 존재들이다. 왜냐하면, 결국 책임은 관리자가 지지 않고 나중에 발생하는 비용까지도 그들이 도맡아야 하기 때문이다. 그들의 평판도 나빠질 것이다. "No"라고 이야기할 수 있는 중간관리들을 찾기는 어렵지만, 그런 사람이 있다는 것은 과제에 대해서 건전한 생각을 하는 집단이 있다는 뜻이다. 그런 회사라면 오히려 희망이 있다. 적어도 그들만큼은 제대로 일하고 있기 때문이다. "Yes"라고 이야기한다고 해서 과제가 제대로 돌아가리라는 것을 보장하지는 못한다. 앞에서 감미롭게 이야기하는 목표 완수에 대한 "Yes"는 결과적으로 제대로 된 소프트웨어를 만드는 데 아무런 도움이 못 된다. 각종 문제점과 위험요소(Risk)들을 이야기하는 사람들의 말에 귀를 기울여야 한다. "도전적인 목표"를 가지는 것은 좋지만 계획도 도전적이어선 안된다. 계획은 말 그대로 지킬 수 있는 가능한 목표이어야 한다. 주말까지 모든 시간을 야근해야지 이룰 수 있는 목표라면 조금이라도 문제가 생기더라도 저절로 계획은 차질을 빚게 될 것이 뻔하다. 사람은 기계가 아니다. 이제는 기계처럼 다룰 사람을 찾으려 해서는 안 된다. 사람은 존중받기를 원하고 그래야만 자신의 과제에 "몰입"될 것이기 때문이다. 계획은 충분히 비관적으로 만들고 문제에는 긍정적으로 접근하는 자세가 필요한 것이다.

# [ 린(Lean)개발 방식 ]

이미 린(Lean)이라는 단어는 많은 곳에서 사용되고 있다. 린(Lean)은 생산 방식의 기본은 자동차 산업에서 왔지만 현재는 다양한 분야에서 사용 중이며, 이제는 소프트웨어 산업에서도 스타트 업(Startup)에서는 일반화된 일이기도 하다. 린은 물건을 만드는 주체가 물건의 품질을 책임진다는 것이 핵심이다. 즉, 누구의 지시로 단순하고 반복적인 작업을 하는 것이 생산을 담당하는 노동자의 역할이 아니라, 스스로 능동적으로 문제점에 대한 지속적인 개선을 이루려고 노력하는 것이다.

여기서 노동자를 소프트웨어 개발자로 바꾸면 결국 사람에 대한 이해가 모든 문제의 해결점으로 귀결된다. "사람을 남기는 장사"를 하라고 이야기하는 것을 TV 드라마에서 본 적이 있을 것이다. 소프트웨어 개발도 결국 개발자를 얼마나 존중하고 그들에게 책임과 권한을 주는가에 달려있다. 소프트웨어 개발자의 기본 소양은 자신이 만든 코드에 대한 문제를 지속해서 개선해 나가면서 실패에서 배우는 과정의 반복이라는 것이다.

소프트웨어 개발에서 가장 흔한 실패는 아마도 자신이 만든 코드의 컴파일 실패일 것이다. 또한, 컴파일할 때 발생하는 "경고(Warning)"도 실패의 일종이다. 단위 테스트의 실패를 통해서도 배울 것이다. 실패가 발생했다면 즉시 문제를 해결하는 것이 중요하다. 실패를 덮어씌운다고 성공으로 변하지는 않는다. 그러한 사소한 실패도 시간이 흐르면 다시 원상태로 회복할 수 있는 순간을 놓치게 된다. 소프트웨어는 변경이 쉽다고 생각되어 왔지만, 변경을 대비를 하지 않은 소프트웨어에 대해서는 가혹하리만큼 비용을 요구하기 때문이다. 그 비용은 일정의 지연도 동반하며, 사용자의 불만과 회사의 낮은 평판도 포함해야 한다.

완벽한 제품을 만들기는 어렵지만, 완벽한 제품을 만들기 위한 노력은 지속해서 이뤄져야 한다. 실패를 통해서 배우지 않는다면 실패는 그냥 실패로 남는다. 배운다는 것은 경험의 축적을 의미하며, 그렇게 배운 값 비싼 경험들은 공유되고 재생산될 수 있어야 한다. 팀의 다른 사람들도 같은 실패를 경험할 가능성이 높기 때문이다. 린 개발방식은 빠른 실패를 권장하고 있으며, 이를 극복할 수 있도록 방법을 만들어 적용하기를 요구한다. 테스트를 주도적으로 개발에 적용하고, 조금씩 증분을 만들어 지속해서 통합하는 것이 실패를 통해서 배우는 방법이다.

# [ 개선, 어디서부터 시작할 것인가? ]

개선해야겠다는 생각은 소프트웨어를 개발해 본 경험이 있는 사람들은 항상 느끼는 점이다. 하지만, 어디서부터 시작할 것인가를 정하는 것은 개발자들의 많은 반발을 가져올 수 있다. 당연한 이야기지만 노력은 적게 들이고 효과는 크게 볼 수 있는 방법을 찾으려는 것이 일반적이다. 대체로 이런 생각을 가장 먼저 구현하는 방법을 툴(Tool)에서 찾는다. 하지만, 툴에서 나오는 결과를 믿지 않는다면 어떻게 할 것인가? 예를 들어, 버전 관리 시스템이 필요 없다고 이야기는 개발자들이 있다면? 정적 분석 결과를 믿지 못하겠다는 반응을 보인다면, 심지어 컴파일러가 만들어주는 경고 메시지도 무시하는 개발 책임자가 있다면 어떻게 할 것인가? 따라서, 문제는 어떤 툴을 쓰고 안 쓰고의 문제가 아니라, 툴을 운영하는 사람과 툴에 입력을 넣는 사람들의 자세(Attitude)다.

소프트웨어 과제에서 가장 중요한 성공요소는 "사람"이라고 이야기한다. 그러한 사람의 역량을 높이고 전체적인 향상을 가져오기 위해서는 어쨌든 시간이 필요하다. 하지만, 그런 시간을 불필요하다고 보는 경향이 많으며, 개선 활동을 위한 시간은 만들어주지 않은 체 항상 과제를 일찍 마치라고만 요구한다. 1년에 할 일을 6개월 만에 끝내는 것은 가능하다. 하지만, 품질이나 기능, 사용자 만족도, 유지보수와 같은 곳에 들어가는 비용은 전혀 고려하지 않는다. 1년에 할 일을 6개월 만에 마치면, 뭔가 필요한 비용을 지불하지 않았다는(절감했다는) 뜻이다. 하지만, 그것이 품질 비용이고 유지보수를 위해서 미리 지불해야 할 비용이었다면 이야기는 달라진다. 이런 비용들은 과제의 후반이나 배포 이후에 발생하며, 때로는 원금은 물론이고 불어나는 이자로 인해서 이자까지도 톡톡하게 치러야 할 가능성이 높다. 가장 쉬운 툴 하나도 제대로 도입하지 못하는 이유도 이러한 여유가 없기 때문이다. 물론 충분한 시간을 주기는 어렵다고 하더라도 일을 제대로 할 수 있는 시간은 확보해야 한다. 시간이 부족하고 사람이 부족하다고 이야기하는 목소리를 단지 개발자가 불평으로만 생각한다면, 개선에 대한 요구를 꺼내서는 안될 것이다. 여유없는 개선은 불가능하며 투자없이는 효과도 없는 것이다.

개선의 시작은 작은 부분이어야 한다. 그 작은 부분은 개발자 개인의 코드에서 찾아야 할 것이다. 작은 코드의 개선이라도 그것이 쌓이면 효과는 기하급수적으로 증가할 것이다. 예를 들어, 모듈 간의 독립성을 증가시키도록 개선된 코드들은 내부의 복잡도와 외부의 복잡도 모두를 감소시켜줄 것이다. 모듈이 해야 할 일만 하게 되며 변경이 필요한 일도 모듈에 할당된 역할(Role)로 한정시킬 수 있기 때문이다. 즉, 잘 정의된 명확한 역할과 책임(R&R)이 개선의 근간이 될 수 있다. 읽기가 쉬운 코드는 사람의 이해를 돕게 될 것이며, 남들이 보기에도 편하니 유지보수나 버그를 수정하는 노력도 줄어들 것이다. 이런 작은 개선들은 바쁘다고 절대 놓쳐서는 안되는 것들이다. 완벽한 문서를 쓰고 좋은 상용툴을 동원해서 작업하는 것도 좋지만, 실질적인 개선은 소프트웨어 개발자의 하루 하루의 일상에서 시작되어야 한다. 그런 작은 개선들은 하루의 일과로 따지면 극히 일부분에 지나지 않지만, 10만 시간의 법칙처럼 오랜 시간을 거치게 되면 큰 차이를 만든다. 작은 차이를 꾸준히 실천하는 것이 소프트웨어 개발자의 기본자세다.

그렇다고 작은 영웅이 되라는 이야기는 아니다. 자기가 하는 일에 조금 더 전문성을 가미하자는 말일 뿐이다. 일상적인 구현이 아닌 좀 더 생각하는 구현을 하고, 간단히 약자로 처리할 수 있는 변수 이름이나 상숫값들에 대해서 조금만 더 의미를 가지는 이름을 쓰자는 것이다. 긴 함수를 만드는 것보다 짧고 인자 수가 적은 함수를 만들고, 남들이 좋다고 이야기하는 코딩방법들을 꾸준히 익혀서 실무에 적용해 보는 것이다. 글을 쓰기 위해서는 많은 책을 읽고 경험을 쌓아야 한다고 이야기한다. 하지만, 코딩하기 위해서는 왜 그런 것들을 하지 않는것일까? 남들이 어떻게 코딩했는지를 배우고, 이를 실제로 자신의 코드에 적용하는 과정을 통해서 우리는 더 나은 개발자가 될 수 있다. 잘 만들어진 소설이 상업적인 인기를 끌듯이, 잘 만들어진 코드는 같은 팀에서 일하는 구성원들의 주의를 끌어당긴다. 서로 어떻게 만드는 것이 좋은 것인지를 경험을 공유하면서 익힌다면, 좀 더 빨리 목표에 근접할 수도 있을 것이다. 대부분의 상업적인 소프트웨어 개발은 이미 개인적인 일이 아니다. 남들의 경험을 통해서도 우린 더 많이 배울 수 있다. 영웅이 될 필요는 없지만 끝없는 개선이라는 목표를 향해서 한발 한발 나아가는 동료들은 누군가의 도움이 즉시 필요할지도 모른다. 그 날을 위한 시작이 오늘이라면 내년에는 적어도 저녁이나 주말에 집에서 기다리는 가족에게 좋은 아빠 혹은 엄마가 될 수 있을지도 모른다.

# [ 정도에 지름길은 없다 ]

뭔가를 쉽게 해결하려고 하면 무리를 하게 된다. 지름길을 가기 위해서는 지름길에 숨겨진 위험을 파악해야 하지만, 미래를 예측하는 것만큼이나 어렵다. 소프트웨어 개발도 마찬가지다. 생략된 절차들은 걸러주어야 할 부분을 놓치게 되고 코드의 안전망은 깨지게 된다. 결과적으로 지름길이라고 선택한 것들에서 얻을 수 있는 이익보다 그것을 선택함으로써 발생하는 위험들을 고스란히 비용으로 지급하게 되며, 만들어진 소프트웨어의 변경에 경직성으로 인해 수정 비용은 기하급수적으로 증가하게 된다. 따라서, 개발에서 무엇을 해야 하고 하지 말아야 할지를 정하는 기준이 필요하다. 이때 기준으로 사용할 수 있는 것은 개발의 편의성이 아니라 고객에게 전달되는 가치다. 즉, 고객에게 어떤 가치를 제공할지를 가장 우선적으로 생각해야 한다. 결국 고객이 우리가 만든 소프트웨어를 살 것이기 때문이다.

고객의 가치를 높이기 위해서는 고객이 무엇을 원하는지를 알아야 한다. 당연히 요구사항은 필수다. 무엇을 만들어야 할지를 모르는 상태에서 일정과 필요한 사람의 수를 말하는 것은 의미 없다. 하지만, 요구사항은 간과되기 쉽고 귀찮기도 하다. 자신이 무엇을 원하는지도 고객 스스로가 모를 때도 있다. 하지만, 고객을 이해해야 하고 고객이 우리가 만든 소프트웨어를 통해서 해결하고자 하는(얻고자 하는 이익) 것이 무엇인지를 정확히 알아야 한다. 힘든 과정이지만 고객을 충분히 조사해야 한다. 성능을 중요시할 수도 있고 안정성이 중요할 수도 있다. 여기서 필요한 것은 고객의 언어를 잘 이해하는 것이다. 그들이 말하는 언어에 대한 이해가 선행된다면 고객의 요구 중 대부분은 미리 파악할 수 있을 것이다. 고객이 내부에 있을 수도 있다. 즉, 최종적인 제품의 사용자가 외부에 있을 수 있지만, 내부에서 사용자들의 요구를 들어서 소프트웨어 개발팀에 전달하는 역할을 하는 사람들도 있다. 그들이 이야기하는 것이 무엇인지 정확히 파악하기 위해서는 자주 만나야 하며, 개발 도중에도 자주 이야기해야 한다. 그들의 이야기를 들어야 한다고 이야기하는 것이지, 그들이 원하는 것은 무엇이든 하라는 뜻은 아니다.

설계는 고객의 가치를 어떻게 실현할지를 정하는 활동이다. 설계가 100% 완전할 수 없다. 실제 구현에서는 불가능한 것들도 있을 수 있기 때문이다. 고객의 요구사항도 구현된 것을 보기 전에는 단정할 수 없다. 고객은 구현된 것을 기준으로 새로운 요구를 발견하기도 하기 때문이다. 따라서, 요구사항도 100% 완전하게 만들고 시작할 수 없는 항목이다. 설계의 주요 테마는 시스템을 구성할 요소들을 파악하고, 각각의 요소들이 가져야 할 역할과 책임 범위를 확인하는 과정이다. 여기서 개발자의 경험이 중요한 역할을 하며, 경험이 쌓일수록 더 좋은 설계가 나올 가능성이 높다. 하지만, 설계를 잘 해야 한다는 당위성만 있지 어떻게 해야 할지 모르는 경우도 많다. UML과 같은 툴을 사용해도 좋지만, 툴이 모든 것을 해결해 주지는 않는다. 결국, 툴로 만드는 내용(Contents)은 사람이 채워 넣어야 한다. 내용을 만들 수 있는 것은 기존의 경험이며, 이를 어떻게 모든 사람에게 동일한 언어로 표현할 수 있는가가 중요하다. 가능한 한 자세하게 해야 한다고 이야기하는 것도 무리다. 왜냐하면, 구현에서 변경이 있을 가능성이 높기 때문이다. 따라서, 설계는 두 부분으로 나누어서 진행해야 한다. 첫 번째는 대략적인 큰 그림을 그리는 단계이고, 두 번째는 작은 마일스톤(Milestone)을 위한 상세한 그림을 그리는 것이다.

구현계획은 우선 순위화된 기능들이 높은 순서대로 어떻게 만들어질 것인가를 보여준다. 여러 번의 반복적인 과정이 올 수 있지만, 요즘은 각각의 반복마다 작은 프로젝트를 실행한다고 가정하고, 요구사항, 설계, 구현, 테스트를 완료해야 한다는 것이다. 각각의 반복은 완전한 기능 구현이 목표는 아니지만, 우선 순위화된 기능들 일부(해당 반복에서 구현해야 할 기능)에 대해서는 완전히 구현해야 한다. 완전한 구현은 테스트 통과를 의미하며, 코딩만 다 했다고 끝나는 것이 아니다. 따라서, 이렇게 구현된 기능을 직접 눈으로 확인할 수 있는 수준까지 가야 한다. 배포할 수 있는 수준에 도달해야 한다는 뜻이다. 대부분의 경우 개발자들은 구현(코딩)만 되면 기능이 완료되었다고 생각하고 다음 기능의 구현으로 넘어간다. 하지만, 구현은 테스트를 포함한 활동이며, 구현된 것들에서 발생하는 오류(컴파일 경고 메시지, 복잡한 코드, 함수나 변수의 이름들, 중복된 코드들, 중복된 테스트 코드들 등)를 제거해야 한다. 일치하지 않는 상세 설계의 내용도 수정되어야 하며, 구현에서 파악된 요구사항들에 대해 업데이트도 해야 한다. 이런 것들은 생략할 수 없는 활동이다. 오류는 발생한 시점에서 고치기가 가장 쉽다. 시간이 지날수록 고치는 비용은 기하급수적으로 증가할 것이기 때문이다.

개발 내부의 테스트와 배포, 설치는 자동화할 수 있는 부분이 많다. 그리고, 고객이 인수하기 위해서 테스트를 하는 과정에 대해서도 자동화를 할 수 있는 부분들이 많다. 매번 수동으로 이런 모든 일을 한다면 상당히 번거로울 뿐만 아니라 실수도 끼어들게 된다. "머피의 법칙"은 사람이 어떻게 할 수 없겠지만, "데모의 법칙"은 충분히 해결할 수 있다. 대부분 환경의 차이에서 발생하기 때문에 고객의 환경을 충분히 고려한 배포 전략이 필요하다. 고객의 입장에서 사용했을 때 어려운 부분이 없는지, 고객이 정말 원했던 기능인지 확인해야 한다. 고객은 자신이 원하는 기능을 항상 일찍 받아보기를 원한다. 따라서, 일찍 중요한 기능들을 먼저 구현해서 배포(데모)할 수 있도록 해야 한다. 그런 기능들이 잘못되었다고 피드백을 주면, 다음 배포 버전에는 반드시 반영할 수 있도록 해야 할 것이다. 고객은 자신이 가치를 두고 있는 것을 받기를 원하지 개발자가 의미를 두고 있는 부분을 받아들이기를 원하지는 않는다. 기술적인 난이도나 아이디어의 신선함보다는 고객의 가치를 최대화할 수 있는 방법을 선택해야 할 것이다.

유지보수는 가장 어려운 개발활동 중의 하나다. 유지보수를 개발활동에 포함시키는 것은 그것으로 인해서 실제로 개발자의 시간에게 영향을 주기 때문이다. 신규 개발과 유지보수가 분리되지 않고 동시에 진행되기 때문이다. 물론, 전담 인력을 배치하고 유지보수를 처리하는 경우도 있지만, 이런 여유가 없이 개발과 유지보수를 하나의 팀에서 대부분 처리한다. 또한, 유지보수를 제품이 배포된 이후에 시작되는 활동에 한정하지 않을 수도 있다. 사실상의 유지보수는 이미 과제의 시작에서부터 고려 대상이다. 즉, 유지보수룰 위한 적극적인 활동은 과제의 시작부터 필요하다. 설계 및 구현 등에도 유지보수를 위한 방안이 필요하며, 테스트, 배포, 설치 등도 전부 포함한다. 유지보수룰 쉽게 하기 위한 설계와 구현, 테스트와 배포 등이 고려되지 않는다면, 그것에 들어가는 비용은 새로운 과제를 하는 것보다 더 많이 들어갈지도 모른다. 소프트웨어는 닳아서 없어지는 것이 아니라 지속해서 개선되는 특성이 있으며, 꾸준히 진화하지 않으면 시장에서 사라지게 된다. 하드웨어는 교체의 대상이지만 소프트웨어는 닳아 없어지지 않기에 교체되지 않고 개선될 뿐이다. 개선하기 쉬운 구조로 되어있지 않은 소프트웨어는 결국 시간이 흘러가면 도태될 수밖에 없다. 경쟁력이 상실된다는 의미다. 시장은 언제나 제품들로 흘러넘치고 고객은 기다려주지 않을 것이기 때문이다.

# [ 소프트웨어 개발 전략 ]

이미 많은 사람이 이야기했지만, 제품을 기능별로 나누어서 여러 번 배포(Release)하는 전략이 소프트웨어 개발에서는 필수라고 생각한다. 이것이 중요한 이유는 개발자에게 최소한의 안전 장치를 두도록 할 수 있도록 하기 때문이다. 즉, 모든 기능을 한 번에 배포하는 것은 사실상 실패할 가능성이 높다. 그렇게 개발되는 소프트웨어들은 과정에서 완성도를 높이기보다 과제의 후반에 집중된 테스트와 디버깅을 하게된다. 버그의 발생은 이미 복잡하게 된 코드의 수정을 동반하며, 또 다른 버그의 원인이 되기도 쉽다. 기능의 통합도 늦어지기에 충돌로 인한 문제의 해결에 필요한 비용도 더 많아지게 된다. 따라서, 중간 과정의 기능별 완성도는 전체 기능이 구현될 때까지는 높아지지 않는다. 따라서, 테스트에 들어갈 수 있는 상황도 대부분의 기능이 구현된 시점 이후가 된다. 물론 모든 프로젝트가 이런 것들 때문에 실패하는 것은 아니다. 잘 정의된 요구사항(변경이 없는)이 있거나, 이미 개발 경험이 풍부하다면(참조할 소스코드가 있는) 중간에 결과물을 내놓지 않아도 성공할 가능성이 높을 수 있다. 하지만, 그렇다고 하더라도 변경이 전혀 없는 소프트웨어 개발과제는 없으며, 이미 경직된 코드 베이스(Code Base)를 가지고 있을 경우에는 변경의 영향도를 제한할 수 있는 방법도 없다.

성공 가능성이 조금이라도 높은 방법으로 과제를 수행해야 한다. 과제 중간에 반복을 통해서 구현된 기능에 대한 완성도를 높이고, 지속해서 그런 기능들을 추가시켜 나간다면 과제는 당연히 성공할 가능성이 높다. 그리고, 과제의 중간에 일정 단축이 필요한 경우에도 언제든 배포할 수준의 품질을 만족하는 코드를 생성할 준비가 되어있을 것이다. 즉, 일정이 촉박하다면 그 일정 내에서만 성실하게 구현된 기능들을 배포할 수도 있다. 시간이 허락한다면 다음 버전에서 필요한 추가적인 기능들을 다시 배포할수도 있을 것이다. 모든 기능을 동시에 개발하게 되면, 기능 간의 완성도가 미흡해서 상시적으로 전체 소프트웨어를 배포하는 것이 불가능하다. 완성도가 낮은 기능은 배포 전에 많은 버그를 양산할 것이고, 아무리 타협을 하더라도 도저히 배포할 수준이 안되는 제품을 시장에 내놓는다는 것은 자살행위와 같을 뿐이다. 따라서, 성실하게 한 가지 기능이라도 완성도를 꾸준히 높여가는 것이 목표가 되어야 한다. 기존에는 모든 기능이 구현된 소프트웨어를 "알파, 배타, RC1...N"과 같이 안정성을 높이는 시간을 가졌다면, 점진적이고 반복적이며 진화적인 개발 방법은 각각의 마일스톤마다 기능이 충분히 동작하는 안정된 코드를 배포하는 것이다. 따라서, 안정된 기능들은 지속해서 안정 상태에 놓이게 되며, 자동화된 테스트가 있다면 새로 개발되는 기능들에 대한 추가적인 검증 노력만 필요할 것이다.

물론 이론적으로는 그렇다는 것이다. 이를 구현하기 위해서는 기존에 하던 코딩 및 테스트에 대한 생각이 변해야 하며 다양한 지식을 요구하게 된다. 해보지 않은 것을 새롭게 도전하는 것은 대단한 용기가 필요한 일이다. 마치 용수철이 회복하려는 힘에서 벗어나기 위해서는 상당히 큰 힘이 필요한 것과 같다. 마치 지구가 잡아당기는 인력을 탈출하기 위해서 로켓들이 대부분의 자기 무게를 연료를 위해서 사용하듯이 새로운 것을 받아들이는 데는 큰 추진력을 요구한다. 이것을 개인 차원에서 진행하기에는 상당한 어려움이 따르며, 반드시 지원하는 조력자를 잘 구해야 한다. 모든 일이 그렇듯 전폭적인 지원이 없이는 성공하기 어려운 일이다. TF(Task Force)와 같은 형식으로 시작하는 조직은 각 팀에서 일부의 인력을 차출 받는다. 하지만, 경험상 그렇게 차출된 인력들이 팀에서 우수한 인력일 가능성은 낮다. 자신의 앞가림을 하기도 바쁜 팀장들이 우수한 팀원을 TF에 내놓을리 없다. 따라서, 그런 일을 하기 위한 첫 단추는 도메인(Domain)에 대한 지식을 충분히 가진 인력들로 이루어진 전문가들을 어떻게 확보할 수 있는가에 성공과 실패가 달려있다. 굳이 TDD나 애자일 방법론에 대한 이야기에 한정된 것이 아니다. 모든 중요한 일의 성공요소에는 "우수한 인력"이라는 것이 기본 가정이기 때문이다. 하지만, 대부분 팀에서 차출한 인력들은 회사에 입사한 지 얼마 안되는 신입사원이거나 업무에서 중요하지 않은 허드렛일을 하는 사람들일 것이다. 그들이 나쁘다는 것이 아니라 적합하지 않을수도 있다는 말이다.

과제를 성공시키기 위해서는 모든 기능에 우선순위를 정하는 것이 좋다. 우선순위가 없는 기능은 필요없는 기능이거나 제대로 모르는 기능일 가능성이 높다. 우선순위는 개발자들이 홀로 정하는 것이 아니라 해당 기능을 요청한 사람과 함께 정해야 한다. 너무 많은 기능을 한 번의 반복 주기에서 구현하는 것은 실용적이지 않다. 따라서, 얼마나 많은 기능을 구현해야 할지를 정하는 것이 추정(Estimation)이다. 예측은 틀릴 수 있지만 하지 않는 것보다는 좋다. 왜냐하면, 그러한 예상없이 시작한 것들에 대해서는 마감시한이 있을 수 없기 때문이다. 목표는 세우고 실행과의 차이를 평가해서 다음 목표를 수립할 때 반영하도록 해야 한다. 사람들은 목표가 타당하기를 요구한다. 개발자는 조금이라도 더 적게 일하는 것이 목표일지도 모른다. 고객은 조금이라도 더 많이 일을 시키는 것이 목표일 수도 있다. 그런 목표를 서로 이해하고 조정하는 절차가 반드시 필요하다. 안정적인 기능을 가지고 이야기한다면 고객은 양보할 여지를 남겨두지만, 완성도가 떨어지는 기능을 가지고 이야기한다면 고객의 신용을 잃는다. 어떤 식으로든 안정하게 동작하는 소프트웨어를 제공하는 것이 소프트웨어 개발자가 해야 할 일이다. 그리고, 그런 계획을 꾸준히 실천하며 고객에게 보여줄 수 있다면, 결코 고객의 신용을 잃지는 않을 것이다. 고객은 오히려 다음 번에 전달받을 소프트웨어에 대한 기대를 높일지도 모른다. 물론, 그렇다고 하더라도 항상 기대 이상을 보여주지는 못할 것이다. 하지만, 적어도 보여줄 것은 충분히 가지고 있을 것이다.

# [ 프로세스 ]

옷을 사러 가면 흔히 디자인이 멋진 옷은 사이즈가 맞지 않는 경우가 많다. 그렇다고 몸에 맞는 옷을 고르자니 마음에 들지 않는다. 이리저리 돌아다녀 보지만 딱히 살만한 옷을 고른다는 것이 힘들 뿐이다. 그래서, 대충 적당하게 입을 만한 것으로 고르고 만다. 하지만, 막상 집에 와서 다시 입어보면 장롱 속에 처박아둔 옷이랑 별반 다르지 않다고 느껴지기도 한다. 우리가 따르는 소위 소프트웨어 개발프로세스도 그런 것이 아닐까? 실제로 일하는 절차(Process)를 정리하고 싶기는 하지만, 정리하고 나면 왠지 어색하고 절차대로 하기 싫어지는 것이 인지상정이다. 그렇다고 소프트웨어 개발프로세스가 없으면 왠지 뭔가 부족한 것 같아 다시 여기 저기를 손보게 된다. 조금 시간이 지나면 소프트웨어를 더 잘 만들고 싶은 생각에 프로세스를 세부적으로 자세하게 만들게 되고, 더 많은 문서의 템플릿(Temlate)들을 정의하게 된다. 하지만, 과연 그렇게 해서 만들어진 소프트웨어들이 정말 제대로 잘 동작할까? 물론 동작은 할 것이다. 왜냐하면, 어쨌든 소프트웨어 과제의 최종 종착지는 "실행되는 코드"이기 때문이다.

옷이 몸에 안 맞는다고 몸을 바꿀 수는 없는 노릇이고, 몸에 맞추려고 옷을 찢을 수도 없는 게 현실이다. 소프트웨어 개발에서 필요한 적당한 크기의 기성복은 애시당초 없다. 제단사가 와서 정확히 자신의 몸의 수치를 재고 가서 그것에 맞게 만든 옷이 아니라면, 무엇인가는 잘못되었을 것이 분명하다. 옷을 하나 사는데도 이렇게 힘든데 하물며 사람들이 여럿이 모여서 만들어가는 과정을 정의하는 것이 쉽지 않은 일임은 충분히 짐작할 것이다. 우리는 만드는 제품의 규모, 기술적인 난이도, 사람의 수, 일정 등을 고려해서 소프트웨어 개발프로세스를 정의해야 한다. 즉, 모든 과제는 과제 나름의 필요한 절차를 가질 수 있다는 것이다. 이런 것들을 종합해서 볼 때 문제는 절차가 아닌 내용이라는 뜻이 된다. 과제의 결과물인 내용에 대해 정확한 평가를 하지 않고서는 단계가 끝났다고 이야기하지 못한다는 것이다. 물론 소프트웨어를 개발하는 방법론은 다양하게 있을 수 있다. 그러한 다양한 방법론들이 이야기하는 것도 마찬가지로 과제에 대한 "재단(Tailoring"이 필요하다는 것이다. 따라서, 최고의 프로세스는 개발팀이 스스로 찾아낸 “최상의 품질을 높이기 위한 구체적인 방법과 절차"가 되어야만 할 것이다.

소프트웨어 개발 프로세스는 문서를 많이 작성해야 한다는 선입견을 품은 개발자들이 많다. 문서는 동작하는 소프트웨어를 만들어주는 밑그림이 되어야지, 동작하는 소프트웨어에 제약을 가하는 것이 되어서는 안 된다. 대부분 문서는 제한을 이야기하고 한정적인 범위에서의 변경을 말한다. 하지만, 실제 동작하는 소프트웨어는 언제나 그보다 더 많은 것을 말하는 것이 사실이다. "백문이 불여일견"이라는 말처럼 아무리 잘 문서로 만들더라도 "동작하는 소프트웨어를 직접 보는 것"보다는 못하다. 따라서, 소프트웨어 개발프로세스는 동작하는 품질이 높은 소프트웨어를 개발하는 데 더 많은 초점을 맞춰야 한다. 그럼 어떻게 그렇게 할 수 있을까? 가장 쉬운 방법은 다양한 도구를 활용해서 객관적인 잣대로 측정하는 것이다. 측정된 결과가 단계에서 요구하는 수준을 만족하는지 확인하는 것이다. 버그의 수, 복잡도, 정적분석에서의 결함 수, 함수들의 라인 수, 테스트가 커버하는 코드의 비율, 테스트 케이스 통과 여부, 리뷰된 코드의 양 등등이 통과 지표로 삼을 수 있는 근거들일 것이다. 이런 것들은 문서가 말하는 것이 아니라 일하는 과정에 저절로 녹아 들어가 있는 것들이다. 각종 지표는 자동으로 추출이 되어야 하고, 이것을 만족시켜야만 각 단계를 넘어설 수 있는 것이다.

그렇다고 문서가 전혀 도움이 되지 않는 것은 아니다. 문서가 도움이 되는 경우는 시스템을 설명하는 큰 그림과 결정의 근거를 보여줄 때다. 그리고, 시스템이 만족시켜야 할 설계 목표를 제시하고 이를 사람들과 공유할 때이다. 모든 사소한 결정의 바탕이 되는 근거는 이유와 함께 정확히 기술되어야 한다. 이런 문서들이 필요 없다고 한다면 매번 새로운 결정을 새로운 바탕 위에서 하게 되고, 전체적인 시스템의 통일성(Integrity)은 쉽게 깨지게 될 것이기 때문이다. 결정에 대한 의도와 이유를 명확히 하는 것은 실수를 반복하지 않겠다는 의지를 표현하는 것이다. 비록 처음에는 실패하더라도 반복적인 개발을 통해서 반드시 성공시키겠다는 의미다. 가설은 잘못될 수 있지만 지속해서 세우고 검증하는 일은 반드시 해야 할 일이다. 따라서, 이런 이유라면 충분한 문서화의 동기가 될 수 있다. 소프트웨어 개발프로세스는 이처럼 최소한의 필요한 문서와 동작하는 품질이 우수한 소프트웨어를 만드는 방법들로 채워져야 한다. 그렇지 않다면 또 하나의 거추장스러운 옷이 되어 장롱 속 어딘가에서 잠들어 있을 것이 분명하다. 곰팡내를 맡을 때쯤이면 새로운 옷으로 바뀔 것이다.

# [ 쉽게 가는 지름길은 없다. ]

어떤 과제든 쉽게 해결 나는 것은 없다. 그런 과제라면 이미 주어졌을 리가 없다. 해야할 가치가 있는 과제라면 반드시 고비는 찾아올 것이다. 과제의 성공은 그런 고비를 어떻게 극복하는가에 달려있다. "쉬운 것은 없다". 준비가 소홀한 과제는 실패할 가능성이 높다. 운이 좋아서 성공할 수 있을지는 모르지만, 확인되지 않은 운에 모든 과제를 매번 기댈 수만은 없을 것이다. 운이 다하면 본래의 실력이 나오고야 만다. 소프트웨어 개발의 본질은 어떤 사람을 쓰는가에 달려있다고 하지만, 실제로는 “어떻게 사람을 쓰느냐”에 달려있다. 팀에는 실력이 있는 사람도 있고 그렇지 못한 사람도 있기 마련이다. 최고의 실력을 갖춘 사람만이 과제에 투입되는 것은 아니다. 기술의 난이도가 높을수록 요구되는 역량도 높겠지만, 대부분의 경우 과제의 완료에 가까워지면 역량도 올라가기 마련이다. 물론, 그 과정에서 사람을 키우기 위해서 어떤 노력을 기울였냐에 따라 다른 결과가 나올 것이다.

사람을 키우는 일이 있고 사람을 망치는 일이 있다. 사람을 키우는 일은 적절한 난이도의 문제를 그 사람에게 주고, 꾸준히 옆에서 관심을 두고 피드백을 주는 것이다. 사람을 망치는 일은 일을 주더라도 믿음을 가지지 않는 것이다. 누군가 나를 믿어주고 있을 때 가장 힘이 났던 경험이 있었다면 무슨 말인지 충분히 이해할 수 있을 것이다. 스스로 커가는 사람의 뒤에는 보이지 않는 누군가의 꾸준한 관심이 있었다는 것을 간과해선 결코 안 된다. 하지만, 결과만을 가지고 이야기한다면 우리가 하는 일의 의미는 결국 한가지로 귀결될 수밖에 없다. "성과"가 있는 곳에 "보상"이 따른다는 것이다. 실패한 것도 성과가 되어선 안 된다는 뜻인가? 지금까지 살아오면서 한 번이라도 실패하지 않은 사람이 있는가? 우리는 실패 속에서도 배운다는 것을 너무나 쉽게 잊어버리고 산다. 실패는 묻어버리고 성공만을 축하한다. 실패도 성공만큼 중요하다고 공식적인 자리에선 이야기 하지만, 실패를 용납하지 않는 것도 우리 자신의 일부분이다. 이런 것들이 바뀌지 않는다면 일을 통해서 성장하는 사람을 보지 못하게 될 것이다.

어떤 식으로 실패할 것인가? 가설을 세우고 그것을 검증하기 위해서 빨리 실패를 하는 것이 좋다. 적어도 나중에 실패하면 회복할 수 없는 상처를 입지만, 재빨리 실패하면 성공하는 방법을 배울 수 있다. 누군가 천 번째 성공을 위해서 999번의 실패에서 배웠다는 말을 했었다. 그리고, 그 마지막 성공으로 999번의 실패에 대한 보상을 충분히 받을 수 있었을 것이다. 빠른 실패는 성공에 그만큼 다가갈 수 있지만, 실패하지 않으려고 완벽하기만을 요구한다면 잘못된 선택을 강요하는 것과 같다. 실패를 용납하지 않는 분위기에서는 "왜"라는 물음을 던지지 않는다. "왜"라고 묻는 순간 실패에 대한 불안감이 크게 다가오기 때문이다. 불안감을 느끼면서도 계속 갈 수밖에 없는 것이 더 큰 실패를 만들게 된다. 아기가 걸음마를 배울 때 작은 실패를 수없이 경험해야 제대로 일어설 수 있다는 것을 이미 알고 있다. 하지만, 현실의 문제에 부딪히면 그런 것들은 머리속에서 사라지고 만다. 결국 작은 불안감이 증폭되어 현실화될 때 "쉽지 않은 일"이 되어버리고 만다. 적극적으로 불안을 정복하는 작은 실패를 경험하지 않는다면, 큰 실패를 당연한 것으로 받아들여야 할 것이다.

미래에 대한 불확실성과 복잡함으로 인해서 우리는 "계획"이라는 것을 만든다. 하지만, 일이 진행되면 세웠던 계획은 여지없이 무너지고, 여기 저기서 미처 생각하지 못했던 다양한 문제들이 발생한다. 원래의 계획에 대한 변경을 이야기하는 것은 불문율로 제외된다. 계획했던 것들을 다 끝마치지 못하거나 더 지연시키는 것만 남는다. "쉽게 가려고 한다"면 “어려운 길”을 선택할 것이다. 계획은 반드시 수정되어야 한다. 미래는 고정된 것이 아니라 변화하기 때문이다. 좋은 의도든 나쁜 의도든 상관없이 결국 모든 것은 바뀌기 마련이며, 우리가 할 수 있는 유일한 것은 그런 변화를 받아들이는 태도를 갖추는 것뿐이다. 변화는 단지 지금까지 경험했던 수많은 실패 중의 하나이며, 곧 다시 성공을 위한 새로운 가설과 검증이 뒤따를 것이다. 쉽게 빨리 실패하는 방법을 찾아야 한다. 비록 그것이 말처럼 쉽지는 않을지라도 회복할 수 없는 큰 실패보다 빨리 작은 실패를 경험하는 편이 성공을 위해서는 큰 보탬이 될 것이다.

# [ 스크럼(SCRUM)을 시작하다. ]

소프트웨어를 만들어서 밥을 먹고 사는 사람이라면 애자일(Agile) 방법론에 대해서는 한 번쯤 들어봤을 것이다. 하지만, 아마도 제대로 경험한 적은 없을 것이다. 아이러니하지만 아는 것과 실제로 실행하는 것은 다르다. 물론 아는 것이 있어야 제대로 실천할 수 있다는 것도 맞는 말이다. 스크럼과 XP(Extreme Programming)을 조합해서 일해봤다면, 아마도 신생 프로젝트를 처음부터 수행한 경험이 있는 사람들일 것이다. 그리고, 대부분은 SI(System Integration)분야에서 일한 사람들일 것이다. Java나 C++같은 객체지향 언어를 사용했을 것으로 생각되고, 단위 테스트 프레임워크는 사용하는 언어에서 주로 사용되는 것을 썼을 것이다. 그런 툴(Tool)을 적용하는 데 있어서 문제는 없었을까? 물론 툴은 단지 도구일 뿐이다. 애자일 방법론이 근본적으로 바꾸려는 것은 소프트웨어 개발자의 생각이다. 애자일 방법론은 지금까지의 방법으로는 새롭고 변화가 심한 복잡한 문제를 제 때 해결하지 못한다는 반성에서 나온 것이다.

애자일 방법론을 적용할 때 소프트웨어 개발자를 가장 힘들게 하는 부분은 어디일까? 문서화는 일단 제외하자. 문서는 필요한 것만 작성한다고 보기 때문이다. 다양한 역할들이 존재하지만 이것도 제외하자. 왜냐하면, 우린 이미 그런 역할을 하는 사람이 있기 때문이다. 역량이나 역할의 이해가 부족하기는 하지만, 전혀 그런 사람들이 없는 것은 아니기 때문이다. 프로세스는 애자일 방법론을 따르기에 백로그(Backlog)을 어떻게 만드는가는 중요할 것이다. 하지만, 이것도 몇 번 하면 일정한 예측이 가능한 범주 내로 들어올 것이다. 스프린트(Sprint) 동안 할 일이 많고 적음은 다음 스프린트에서 고려가 되기에 이것도 그렇게 어렵지는 않을 것이다. 이런 저런 것들을 다 제외하고 나면 역시 남는 것은 소프트웨어 품질에 대한 확인이다. 매번 스프린트마다 품질을 확보하는 일은 쉽지 않다. 그리고, 실제적으로도 가장 큰 노력이 들어가는 부분이다.

소프트웨어 개발자의 생각을 바꿀 수 없다면, 혹은 그들의 협조를 얻을 수 없다면 애자일 방법론을 적용하는 것은 불가능하다. 일하는 주체의 생각이 변화해야만 방법론은 힘을 발휘할 수 있다. 권한을 가진 사람의 지원도 필수다. 그들이 과제를 전폭적으로 지지하지 않는다면 무슨 일이 가능하겠는가? 고객도 항상 가까이에 있어야 한다. 그 사람들이 원하는 우선순위가 높은 가치를 빨리 제공할 수 있도록 개발팀을 도와야 한다. 테스터들은 팀의 내부에서 코드를 수시로 검증할 수 있어야 한다. 개발자들은 과제의 투명성을 높이기 위해서 스스로 얼마나 일을 진행했는가를 검증받을 수 있어야 한다. 테스트 코드를 만들고, 자신이 만든 테스트 케이스가 얼마나 코드를 확인(실행)할 수 있는지 수치로 보여주어야 한다. 상품화에 집중하는 조직을 만들고 외부와 대화 채널을 단일화하는 것도 중요하다. 변경이 지속해서 반영될 수 있는 소프트웨어 구조도 설계 및 구현되어야 한다. 따라서, 이런 모든 것들을 하기 위해서는 지금까지와는 다른 개발자의 태도에 근본적인 변화가 필요할 수밖에 없다.

모든 것을 한꺼번에 하는 것은 망하는 지름길이다. 가장 기본적인 것부터 하나씩 차근차근 접근해야 한다. 첫 발은 먼저 지속적인 빌드가 가능하도록 만드는 것이지만, 사실 이것은 빌드할 수 있는 코드 자체가 있어야 진행할 수 있는 일이다. 좀 더 기본적인 것으로 파고들면 소프트웨어 개발자 스스로가 만드는 단위 테스트 및 통합 테스트가 대상이 될 것이다. 이것들은 남이 대신해주는 것이 아니라 소프트웨어 개발자 스스로가 해야 하는 일이기 때문이다. 기초적인 테스트 역량을 확보하지 못하면 매번 스프린트마다 안정적인 품질을 확보할 수 없다. “애자일은 안정적이고 동작하는, 우선 순위화된 고객의 요구사항을 만족시키는 소프트웨어를 가능한 한 빨리 제공하는 방법”이다. 따라서, 안정적인 소프트웨어는 반드시 먼저 확보해야만 모든 것이 가능하다. 개발자라면 현재의 개발 프로세스에서 어디가 가장 큰 병목 구간인지 이미 잘 알고 있을 것이다. 개발자는 막판에 밀어닥치는 버그로 인해서 끝없는 디버깅과 테스트를 반복하는 것을 가장 두려워한다. 미루어진 모든 것을 한꺼번에 마지막 단계에 하기 때문에 망하는 것이다.

속도(Velocity)를 유지하기 위해서는 중간 품질 수준을 높여야 한다. 일을 했다고 말할 수 있는 근거를 확실히 정하고, 그것을 만족시켰을 때만 100%라고 해야 할 것이다. 일이 완료되었다는 의미는 테스트되었다는 것을 포함한다. 상세 설계와 구현, 테스트가 되지 않았다면 일은 끝난 것이 아니라 미뤄진 것일 뿐이다. 그런 것들을 포함해서 몇 %의 구현이라고 말하는 것은 의미가 없다. 0아니면 100%만이 있을 뿐이다. 속도는 달성한 목표에 대한 수치이며, 100%완료되지 않은 일은 포함될 수 없다. 중간 품질을 유지하기 위한 방법은 테스트 밖에 없으며, 지속적인 속도를 유지하기 위해서는 자동화는 필수다. 그렇지 않다면 일을 했다고 하더라도 또 다른 변화로 인해서 처음부터 다시 해야할 뿐이다. 반복되고 단순한 일들을 자동화시키지 않는다면, 사람의 노력이 추가로 들어갈 수밖에 없으며 과제 후반으로 갈수록 개발 속도가 점점 늦어지게 된다. 스프린트의 속도를 일정하게 유지하기 위해서는 일의 완료조건을 엄격히 정해야 하며, 점차 느려지고 있다면 원인을 밝히고 추가적인 조치를 취해야만 한다. 사람을 위한 일이 사람이 아닌 일로 바뀌는 원인을 찾을 수 없다면 어떤 방법론을 쓰더라도 결국 원래의 자리로 되돌아올 뿐이다.

# [ 시간은 언제나 부족하다. ]

소프트웨어 개발자들이 가장 많이 하는 변명이 "시간이 없다"는 말이다. 그들에게는 실제로도 거의 시간이 없다. 시간이 없는 근본적인 이유는 다른 곳에 있을지도 모르지만, 개발자에게만 한정시켜서 본다면 결국 일하는 방법에 문제가 있음을 발견하게 된다. 예를 들어, C++로 코딩은 하지만 C와 유사한 형태의 코딩에 겉옷만 그럴듯하게 입힌 경우도 있으며, C의 표준함수는 사용하면서도 STL은 믿을 수 없어서 자신들이 직접 구현했다는 엉뚱한 대답을 듣곤한다. 코딩 규약을 이야기하는데 코드의 가독성보다는 성능이 중요하다는 이야기를 하고, 복잡한 코드를 지적하면 이미 검증되고 상품화되어 안정적이라고만 이야기한다. 이런 모든 것에 대한 변명으로 "시간이 없다"는 말이 왠지 어울리지 않아 보이지 않는가? 이것은 시간이 없어서 그런 것이 아니라 무지에서 비롯된 것이며, 자신이 아는 지식의 테두리가 좁아서 그런 것이다. 심지어 신입사원도 다 아는 것들을 웃기지도 않을 논리를 가져다 붙이는 소위 말하는 "소프트웨어 아키텍트"라고 불리는(?) 사람도 있다. 그들은 스스로 자신이 그만한 직책에 맞는 사람인지 심각하게 고민해야 할 것이다.

“언제나 시간이 부족하다는 것”은 맞는 이야기다. 경쟁은 시간을 기다려주지 않기 때문이다. 시장의 선두업체가 아니라면 누군가는 이미 앞서 나가고 있을 것이 확실하다. 시간이 부족하면 포기할 수 있는 것은 무엇일까? 기능만이 유일하게 포기할 수 있는 것에 속한다. 기능이 적다고 물건이 팔리지 않는다는 불평을 마케팅이나 상품기회에서 듣게 된다면 팔릴 수 있는 기능을 다시 정의해야 할 것이다. 사용자의 80%에 해당하는 대부분이 20%의 기능만을 사용한다는 통계적인 이론을 다시 거론할 필요도 없을 것이다. 기능은 줄이더라도 품질은 절대 양보해서는 안 된다. 내적인 품질과 외적인 품질 둘 다 중요한 제품의 구성요소다. 외적인 품질은 사용자들의 경험에 관련된 것으로 당연히 좋아야 한다. 소프트웨어에서 말하는 내적인 품질은 소스코드 자체를 말하는 것으로, 이것은 사업이 지속해서 성장할 수 있는가를 판가름할 수 있는 바탕이 된다. 즉, 소프트웨어의 내부적인 품질이 좋지 못하면 시간은 항상 나의 것이 아니라 남의 것이 될 수밖에 없다. 기다려주지 않는 사용자는 이미 다른 제품을 사서 쓸 것이기 때문이다. 내부적인 품질의 단적인 가장 쉬운 척도는 구현된 함수들의 길이를 보는 것이다. 길이가 긴 함수가 코드에 많다면 내부적인 품질은 좋지 않을 것이 분명하다.

품질은 지금 당장에 만족해야 하는 것도 있고 시간이 흘러도 만족하여야 할 것이 있다. 지금 당장 만족해야 할 것은 버그를 제거하는 것이다. 하지만, 시간이 흘러가면 이런 것들은 근본 원인을 제거하는 것이 아니라 코드를 망치는 길로 들어서도록 만든다. 지금 당장은 어떻게든 비난은 피해갈 수 있겠지만, 결국에는 문제를 더 복잡하고 풀기 어렵게 만드는 것이 "빨리 고쳐라”는 말이다. 서둘러 고치면 잠시 문제가 보이지 않는 것처럼 생각되지만, 조금만 변경해도 다시 같은 문제를 발견할 수 있을 것이다. 품질은 당장 만족되어야 할지라도 대충 만족되어서는 안 된다. 시간을 절약하고 싶다면 문제를 해결하는 근본적인 접근 방법에 있어서 변화를 주어야 한다. 문제를 발견했을 때 즉시 해결하는 것이 가장 좋다. 하지만, 실제로는 대부분 문제가 있음에도 문제를 발견하지 못하고 넘어가는 경우가 많다. 제대로 문제를 발견하는 노력을 기울이지 않았기 때문이다. 자신이 작성한 코드에 대한 충분한 검증 없이 다른 사람의 코드와 통합시키기 때문이다. 이런 식으로 개발한다면 단순한 몇몇 기본적인 문제만 제거되었을 가능성이 크다. 이렇게 대략적으로 테스트된 코드들이 합쳐져서 복잡한 구조를 만들어가면, 아무리 많은 입력의 조합으로 테스트하더라도 빈 곳은 언제나 있기 마련이다. 물론, 모든 입력을 조합해서 테스트하는 것이 불가능하다는 것을 잘 알지만, 스스로 자신감을 채워줄 수 있을 정도의 테스트는 반드시 할 수 있어야 한다.

품질을 양보하게 되면 회사는 무너진다. 일정을 양보해도 회사는 어려워진다. "품질은 자존심이고 일정은 생명이다."라는 말로 자존심을 잃더라도 생명을 지키는 방법을 선택한다면 회사의 미래는 없다. 따라서, 일정 안에 끝내야 하는 기능의 선택에 집중해야 한다. 그 기능에 대해서는 어떤 희생을 치르더라도 반드시 높은 품질을 확보해야 한다. 품질 확보의 기본은 실행할 수 있는 코드의 가장 작은 단위가 될 것이다. 기초적인 품질은 시스템을 구성하는 단위 부품을 어떻게 만드느냐에 달려있으며, 그것들을 조합한 전체적인 최적화를 어떻게 가져가는가에 의존적이다. 통합이라는 이름으로 단위 부품을 조합하는 과정이 최적화의 요건이다. 부품의 성능만 최적으로 맞춘다면 국지적인 최적화에 그치게 되며, 통합된 형태로 최적화를 보아야 한다. 단계별로 최적화를 이루어야 하며, 하나씩 분리해서 통합되어야 한다. 통합 자체는 자주 이루어져야 하지만 통합을 위한 준비와 통합 후에 따르는 문제는 작고 반복적으로 해결할 수 있어야 한다. 문제가 끼어드는 틈은 논리적인 비약이 생기는 순간이다. 갑작스러운 변화는 이해의 범주를 넘어서게 되며, 이 순간이 버그가 침투하기 좋은 때다. 이것을 얼마나 줄여줄 수 있는가가 시스템의 안정화를 위해서는 가장 중요하다. 코딩에서 논리적인 비약이 없어야 하며 테스트도 단계별로 진행되어야 한다. 그것만이 유일한 품질에 대한 해답이다. 품질을 지킨 회사는 결국 살아남을 것이다. 물론, 품질만으로는 살아남지 않는다. 생존의 조건 중 가장 기본이 품질이라는 뜻이다. 그것조차도 없는 회사는 경쟁의 다음 라운드에서 잊히게 될 것이다.

# [ 측정해야 개선할 수 있다? ]

개선하기 위한 선행조건은 측정해야 한다는 것이다. 쉽게 이야기하면 측정할 수 있는 것들만 개선할 수 있다는 말과 같다. 측정할 수 없다면 무엇이 얼마나 개선되었는지 확인할 수 없다. 하지만, 실제로도 그럴까? 오히려 측정하기 어려운 것들이 정말 중요한 개선점이 되는 경우가 종종 있다. 간단한 예로 "개발자의 역량"과 같은 것을 들 수 있다. "개발자의 역량을 향상할 수 있는 방안"을 만들라는 지시를 받았다고 하자. 먼저 개발자의 현재 수준을 알아야 하고, 부족한 부분이 무엇인지를 파악해야 한다. 현재 수준이라는 것은 어떤 기준으로 만들어질 수 있을까? 전문 영역에 속하는 것들은 다양한 분야가 있고 각 개발자가 서로 다른 전문 분야에서 일하고 있다면, 이들의 역량을 측정하는 객관적인 기준을 만들기는 어렵다. 따라서, 객관적인 잣대로 개발자의 역량을 측정하는 것은 사실상 불가능하며, 가장 가까운 곳에서 관찰할 수 있는 위치에 있는 사람이 전체적인 역량 중에서 일부분에 한정해 이야기할 수 있을 뿐이다. 객관성을 지니기 위해서 부문에 상관없이 공통적인 문항들을 가지고 평가한다면, 전문 영역을 제외한 평가이기 때문에 개발자들이 쉽게 결과를 받아들이지도 않을 것이다. 이런 관점에서 본다면, 측정할 수 있어야 개선할 수 있는 것이 아니라 측정되는 것만 개선할 뿐이다.

측정할 수 없는 것을 측정하라고 요청을 받았다면 어떻게 해야 할까? 한 가지 방법은 측정할 수 없는 항목들이 투영되어 나타나는 결과물에 대한 측정으로 대체하는 것이다. 즉, 직접적인 측정은 객관적인 결과를 만들기 어렵기에, 측정 항목이 영향을 주고 있다고 생각되는 다른 측정할 수 있는 항목들을 찾는 것이다. 예를 들어, 개발자의 역량을 직접 측정하기는 어렵지만, 그들이 만들어내는 코드의 품질은 측정할 수 있다. 이 때도 역시 한가지 주의할 점이 있다. 결과물과 결과물을 만들어내는 사람을 동일시해서는 안 된다는 것이다. 품질이 나쁜 코드가 발생한 원인은 복합적인 이유 때문이기에, 단순히 개인의 문제라고 한정해서 보기는 어렵다. 만약 개발자의 업무가 과중해서 제대로 집중할 수 없는 상황에서 급하게 코딩을 했다면, 결과물도 당연히 그러한 상황을 반영할 것이기 때문이다. 이것은 사람의 문제가 아니라 사람을 둘러싸고 있는 환경과 상황이 문제도 포함한다는 것이다. 따라서, 개선의 목표는 개인이 결과물을 만드는 환경과 방법에 대한 것이지 사람의 잘못을 따지기 위한 일이 되어서는 안된다. 만약 프로젝트를 수행하는 팀의 역량을 측정하고자 한다면 어떻게 해야할까? 이 때는 프로젝트를 실행하고 있는 사람들의 역량을 측정해서는 안되며, 프로젝트가 제대로 진행되고 있다는 것을 나타내주는 지표를 측정해야 한다.

성공하는 프로젝트를 만들고자 한다면 당연히 성공하는 과제들의 패턴을 봐야 한다. 어떻게 했기에 다른 사람들은 실패하지만 해당 과제는 성공하는지 분석해야 한다. 과제를 성공으로 이끄는 요소에는 여러가지가 있겠지만, 가장 먼저 생각할 수 있는 것은 "소수의 정예 인력"이 첫 번째가 될 것이다. 하지만, 이것은 사람에 대한 이야기이며, 그렇지 않은 상황에서는 적용하기 어렵다. 이것이 의미하는 것은 사람에 대한 투자를 지속해서 해야 한다는 말과 우수한 인력을 확보하기 위해서 노력을 멈춰선 안 된다는 뜻으로 해석하면 충분하다. 두 번째 요소는 과제의 초반에 필요한 노력을 충분히 하고 있는지를 보는 것이다. 이것은 몇 가지 체크리스트를 활용해 문답형식을 정리할 수 있을 것이다. 굳이 점수로 환산하고자 한다면 가능은 하겠지만 절대적인 지표가 되는 것은 아니다. 잘 한다, 조금하고 있다, 안 한다 정도로만 구분해도 충분할 것이다. 하지만, 이런 형식의 조사는 주관식일 뿐이며, 답변하는 사람의 판단과 양심에 기대는 방법이다. 관리자에게 투명한 결과를 보여주고 싶다면, 결국은 최종 결과물인 만들어진 소프트웨어를 가지고 이야기해야 한다. 객관성을 확보하기 위해서는 구현해야 할 기능에 대한 난이도(필요한 노력의 크기)나 시간으로 표현된 값을 사용해야 하고, 발견된 버그의 수와 해결된 버그의 수를 비교한 간단한 그래프, 실제로 기능을 구현하기 위해서 투입된 인력의 수(혹은, 수로 표현된 노력-시간이건 날짜건 상관없다.) 등을 사용할 수 있다. 각 반복을 마칠 때 이런 값들은 과제가 어떻게 진행되고 있는지에 대한 "측정된 상태”로 인정받을 수 있을 것이다. 물론 이것도 완전히 객관적인 자료는 아니지만, 점차 정확도를 높일 가능성은 충분히 있다.

모든 보고된 수치가 다 좋아 보인다고 해서, 실제로 그 소프트웨어의 품질이 우수하다고 말할 수 있을까? 아마 이것은 해당 소프트웨어를 개발하고 있는 사람들이 줄 수 있는 답이 아닐 것이다. 앞에서 이미 말했듯이 보여주는 것만 보아서는 본질을 알 수 없다. 그림자만 보고 실제로 어떤 형상인지 정확히 알 수 없듯이, 각종 숫자를 표현된 결과를 본다고 해서 내용까지도 다 알 수 있는 것은 아니다. 한가지 위안이 될 만한 것은 그런 숫자라도 좋은 소프트웨어라면 일단은 기본적인 품질은 갖추었다고 볼 수 있다. 최소한 그런 것들이라도 만족시켜야 과제를 성공으로 이끌 수 있는 것이다. 하지만, 눈에 보이지 않는 것들도 절대 잊어선 안 된다. 과제를 실행하는 실무 개발자들의 자세와 그들이 모여서 일하는 팀의 분위기, 고객과의 관계 및 이해 당사자들의 지원과 협조, 테스트의 중요성에 대한 자각과 실무 적용, 문제를 인식하고 처리하는 방법과 절차, 공통으로 따르는 규정에 대한 이해와 관심 등등, 다양한 것들이 복합적으로 만족되어야 훌륭한 소프트웨어를 만들어낼 수 있다. 한마디로 말하면 "품질이 우수한 소프트웨어를 만들기 위한 문화"를 서로 공유하고 있는 것이다. 문화 수준을 측정하지는 못한다. 하지만, 그러한 문화가 없다면 우리 스스로 정체성이 없다는 증거이며, 역량을 결집할 구심점을 잃었다는 뜻이다. 측정하지 못하는 것들은 측정할 수 있는 것들의 이산적인 분포 사이의 틈을 채워주며, 간접적으로 표현되는 지표들 속에 묻어 나온다. 측정하지는 못하지만 반드시 만들어 나가야 할 중효한 것들이 있다는 사실은 변하지 않는다.

# [ 밀어넣기와 빼오기(Push vs. Pull) ]

일이 생각한 대로 잘되지 않을 때 사람들을 "Push한다”는 표현을 써서 “압박을 준다”라는 의미로 사용한다. 하지만, 일을 시키는 것이 자체가 나쁜 것은 아니지만, 일을 직접 하는 당사자로서는 인간의 본성과는 어긋난다는 것을 알 것이다. 즉, "시키는 일보다는 자신이 하고 싶은 일을 선택할 수 있을 때 더 높은 성과가 난다." 문제는 여기에만 있는 것은 아니다. 시키는 형태의 일이 일상화가 되면, 일은 하는 사람은 점점 더 많은 일을 한꺼번에 해야 하는 상황에 도달하게 된다. 자신의 책상 위에 쌓인 이런 저런 일들 속에서 옮겨다니게 되고, 이로 인해 업무 공백과 낭비가 발생하게 된다. 맡겨지는 일들은 다른 일의 완료를 기다리지 않는다. 일의 우선순위는 작업을 처리하는 사람이 동의한 것이 아니라, "Push"하는 사람의 직급이 무엇이냐에 따라 결정된다. 일의 중요도가 아닌 사람의 중요도가 일을 우선순위를 결정하는 것이다. 물론 고객도 그런 사람 중의 하나지만, 고객에게 전달되는 가치(Value)를 가지고 일의 우선순위를 결정하지는 않는다.

이런 방식은 소프트웨어 개발자를 "수동적으로 주어진 일만 수행하는 부품”으로 대하는 태도다. "Pull" 방식은 소프트웨어 개발자가 능동적으로 일을 가져와서 할 수 있도록 만들어준다. 그리고, 그 일을 하는 동안에는 다른 일들은 "대기" 상태로 남게 되며, 대기 상태의 일들도 우선순위에 따라 처리순서가 달라질 수 있다. 먼저 주어진 일이라고 해서 반드시 먼저 처리된다고는 볼 수 없다. 항상 현재 처리중인 일을 완료한 후에 대기하고 있는 일에서 제일 우선순위가 높은 일을 가져온다(Pull). 따라서, 개발자의 입장에서는 명확한 일 처리의 순서를 알 수 있으며, 과도한 부하가 걸리고 있다는 것을 즉각적으로 알 수 있다. 처리되지 못하고 남아있는 일들의 대기열이 줄어들지 않거나 더 길어진다면 부하가 높다는 의미다.

예전에 지정 좌석이 없는 극장에서 영화를 본 경험이 있는 사람이라면, 흥행 중인 영화를 보기 위해서는 서서 보거나 좌석이 아닌 복도에 앉아야 했었다. 하지만, 지정 좌석이 생긴 이후에는 좌석이 없는 경우에는 더 이상 관객을 받지 않고, 관객들이 영화를 편하게 즐길 수 있는 상황이 되었다. 하지만, 개발 현장에서는 아직도 이런 것들이 통하지 않고 있다. 무리하게 많은 관객을 극장으로 밀어(Push)넣고, 제대로 영화에 집중하지 못하는 상황을 지속해서 만들고 있는 것이다. 관객의 입장에서는 들인 비용만큼 제대로 서비스를 받지 못해서 불만족스러운 상황이 지속이 될 것이다. 물론 극장 주인은 돈을 많이 벌 수 있다고 생각할지도 모른다. 하지만, 극장 간의 경쟁이 없다면 그렇게라도 관객들이 영화를 보러오겠지만, 경쟁이 치열한 상황이라면 아마도 그런 극장에 관객들이 즐거운 마음으로 가지는 않을 것이다.

관객을 일이라고 보고 극장주가 개발자라고 본다면, 지나치게 많은 일에 치중하는 것은 "품질"을 떨어뜨릴 수밖에 없다. 품질을 만족시키면서 일 처리 속도도 늘리고 싶다면 더 많은 투자밖에 답이 없다. 하지만, 현실은 투자는 하지 않으면서 효율만을 높이려고 노력할 뿐이다. 일은 포화상태가 되고 품질이 낮은 제품밖에 생산하지 못한다. 결국 그런 모든 노력이 이익(가치)으로 연결되지 않는 비율이 커지게 되고, 회사는 "이윤 추구"라는 목표에서 멀어지게 된다. 비용은 지속해서 늘어나지만, "Push" 되는 일도 덩달아 늘어나게 되는 현상이 발생하게 될 것이다. 이쯤 되면 이익을 늘리는 방법은 더 많은 일을 해야 한다는 강박관념에서 벗어나지 못하게 되고, 악순환은 회사의 매출이 지속해서 떨어져 이익이 거의 남지 않는 상태에 도달해도 변하지 않을 것이다.

이런 상황을 해결하는 유일한 방법은 해야 할 일 자체를 줄이는 것이다. 그리고, 일을 제대로 하기 위해서 더 많은 프로세스와 더 정교한 방법론을 도입하는 것이 아니라, 개발자가 동의하는 수준에서 실행에 집중하는 것이다. 소통은 당연히 벌어진 틈을 메워주는 중요한 도구다. 좋은 제품을 만든다는 것은 실제 요구사항에 가깝게 일을 하는 것을 말하며, 요구사항을 주는 사람들과 자주 만나서 이야기하는 것이 중요하다. 문서를 많이 작성한다고 해결되는 것은 아니다. 문서를 쓰는 것도 중요하지만, 문서보다 일단은 대화를 시도하는 것이 중요하다. 일마다 중요도를 반영한 우선순위를 가진 "Ticket"을 발행하고, 이를 개발자들이 "Pull" 해서 처리하는 방법을 생각해야 한다. 지나치게 많은 "Ticket"들이 돌아다니지 못하도록 통제해서 개발자가 충분히 소화할 수 있는 만큼의 일을 맡기는 것이다. 누군가 와서 아무 때나 발행된 "Ticket"을 개발자에게 처리하라고 지시한다면, 과거에 일하던 방식(Push)과 전혀 다를 것이 없다.

도구는 단순할수록 더 잘 사용할 수 있다. 다양한 기능이 있는 도구는 더 쉽게 고장나거나, 다양한 기능을 제대로 다 사용하지 못하는 경우가 많다. 복잡한 것은 도움이 안된다. 가능한 간소한 절차들을 모든 개발자와 이해 관계자들이 일관되게 사용하는 것이 중요하다. 마치 규칙이 많으면 아무도 규칙에 신경을 쓰지 않는 것과 같다. 간단해서 모두가 쉽게 지킬 수 있는 규칙을 만든다면 오히려 그것만이라도 반드시 지키려고 할 것이다. 복잡한 프로세스나 문서보다는 간단하지만, 반드시 지켜야 할 규정을 만들어서 사용해야 한다. 일 자체를 줄이는 것은 많은 사람이 반대할 가능성이 높다. 가장 반대가 심한 사람들은 아마도 일의 결과물을 팔아야 할 사람들일 것이다. 팔리는 제품이나 숫자와 관련된 통계치가 반영된 정보는 주지 않지만, 줄어든 일로 인해서 잃어버릴 시장만은 반드시 언급할 것이다. 하지만, 그런 시장은 지금도 없고 미래에도 없을 가능성이 높다. 그렇게 제대로 예측된 시장이 있다면 지금과 같은 어려운 상황이 오기 전에 미리 대응했을 것이기 때문이다.

일을 하라고 시키는 것 보다 스스로 하는 방법을 구현해야 한다. 일의 흐름이 자체적으로 조절될 수 있도록, 지나치게 많은 일이 몰리지 않게 만들어야 최대의 성과가 난다. "효율"보다는 "품질"을 먼저 생각해야 한다. "효율"이 우선하는 경우는 많은 제품을 싸게 빨리 만드는 경우밖에 없다. 지식에 관련된 일을 하는 사람들에게 빨리 만들라고 한다면, 대충 만들라는 말과 다르지 않다. 품질을 양보한 효율이란 가치 없는 일에 "삽질"하는 것밖에 되지 않는다. "빨리"라고 이야기하는 회사는 "제대로"라는 말을 빠뜨리지 않는다. 그 이유는 단순한 "책임회피"에 지나지 않다. 자신들도 그렇게 일해본 경험이 없으며, 제대로 일하는 방법도 모르는 사람들이 하는 이야기일 뿐이다. 현실적으로 이야기하려면 "기능이 조금 빠져도 되니 중요한 기능을 제대로 만들라"고 해야 한다. 제대로 기능을 만들기 위해서는 막무가내로 던지는 숙제들을 거부할 수도 있어야 한다. 시도 때도 없이 벌어지는 회의와 각종 TF들도 그만해야 한다. 숙제는 숙제로 이어지고, 회의는 또 다른 숙제를 부른다. 그리고, TF는 끝없이 이어져 상설 팀이 되고 만다.

일은 자신의 의지로 결정할 수 있어야 한다. 그리고, 과도한 일은 개인과 조직을 위해서도 좋은 현상이 아니다. 일은 밀어 넣는다고 다 처리되는 것이 아니며, 그렇게 밀어 들어간 일들은 결과적으로 좋은 결과를 가져오지 못한다. 일은 그 일을 해야하는 사람들이 원하는 데로 당겨올 수 있어야 하고, 그렇게 하는 것이 일을 높은 품질수준으로 처리하는 방법이다. 급하게 실행된 일의 결과는 고치는 비용이 더 많이 든다. 그리고, 그런 상황이 반복된다면 고치는 사람이나 그렇게 만든 사람이나 둘 다 힘들기는 마찬가지다. 관리자들은 "멀터태스킹(Multi-Tasking)"에 능해야 한다고 이야기하지만, 근본적으로 한 번에 한 가지씩 하는 것이 더 빠르다.

컴퓨터의 스케줄링 알고리즘은 멀티태스킹을 위해서 잦은 "환경 전환(Context Switching)"이 필요하다. 일이 점점 더 늘어갈수록 일간의 동기화 비용도 커지게 되며, 중간에 다른 일을 처리하게 되면 이전에 하던 일을 다시 시작하기 위한 비용도 많이 들어가게 된다. 결국 그런 비용들이 실제로 CPU를 사용하는 작업보다 더 많아지게 되면, 사용자는 그냥 멍하니 화면만 지켜보고 가만히 있어야 한다. 사람과 컴퓨터가 다른 점이 있다면, 컴퓨터는 하드웨어 성능을 높이기 위해서 더 좋은 기계로 쉽게 교체를 해도 되지만, 사람은 대체할 수 없는 소중한 존재라는 것이다. 그 자리에 있는 사람만이 최적의 인력이라는 사실을 믿어야 한다. 적어도 그 사람에게 일을 시키고 있다면 믿어야 한다. 그렇지 않다면 그 사람에게 일을 시키는 자체가 오류일 뿐이다.

# [ 초기 품질이 전체를 좌우한다. ]

초기에 품질을 확보하는 방향이 전체 과제를 성공으로 이끌 확률을 높인다. 초기 품질이란 개발의 시작과 동시에 품질을 확보할 수 있는 다양한 활동을 시작해야 한다는 의미이다. 실패하는 대부분 과제는 "일정에 대한 지나친 두려움"으로 과도하게 빠른 구현을 하기 때문에 발생한다. 개발자들이 흔히 말하는 “구현완료”에는 "검증(Test)"이 빠져있으며, 테스트는 구현이 완료된 시점에 가능한 활동이라는 잘못된 생각을 가지고 있다. 전체 시스템을 한 번에 완성하려고 해서는 안 된다. 구현과정은 “수직(Vertical)”으로 나뉜 기능들을 단계적으로 완성해 나가야 한다. 이 때는 각 기능의 구현이 완료되는 시점에 해당 기능의 테스트도 완료되어야 한다. 기능이 구현되었다고 이야기하는 것은 해당 기능이 검증되었다는 것을 의미하며, 사용자에게 최소한 그 기능이라도 쓸 수 있는 상태가 되었다는 것을 알려줄 수 있다. 따라서, "Big Bang"방식의 한번에 전체를 통합하는 일은 발생하지 않는다. 물론 기능구현의 과정에서 잦은 통합과 테스트가 이루어져야 하고, 그런 활동을 개발의 초기부터 완료까지 꾸준히 해야한다. 품질 확보는 테스터들만이 관련된 일이 아니며, 개발자가 주도적으로 자신이 개발한 코드의 품질에 확신을 얻을 수 있어야 한다.

과제의 시작에서 대충이란 없다. 필요한 만큼을 반드시 한다는 생각을 가져야 한다. 완벽이라는 말도 없다. 완벽한 것은 이미 다 만들어진 제품도 불가능한 일이다. 필요한 일이란 "과제의 비전", "주요한 마일스톤(Milestone)", "위험(Risk)과 관리방법", "지원(Support)", "사람(개발자, 관리자)" 등을 정하는 것이다. 설계는 중요한 활동이지만 완전한 설계를 만들수는 없다. 구현 내용이 전혀 바뀌지 않는다면 모르겠지만, 그렇지 않을 경우 설계에도 영향을 줄 수 있다. 요구사항을 잘 기술해야 한다고들 생각하겠지만, 모든 요구사항을 과제 초반에 확정 짓기도 사실상 불가능하다. 물론 요구사항을 먼저 결정하고 진행하는 과제들도 있다. 특히 방산과 관련된 업체에서는 그런 일들이 흔히 일어난다. 변경이 용납되지 않는 과제들은 변경을 허용하는 것을 막기 위해서도 요구사항들을 명확하고 구체적으로 적어야 한다. 하지만, 대부분의 소프트웨어 과제들은 지속적인 변경이 발생하는 것이 일반적이며, 심지어 만들고 있던 코드들을 다 버리고 완전히 새로 만들어야 하는 경우도 있다. 대충하지 않으려면 필요한 것들은 제대로 해야 한다. 이를 위해서는 구현의 첫날부터 실행되는 코드와 그것을 테스트하기 위한 코드가 둘 다 존재해야 한다. 사실상 과제의 대부분 활동은 "코드를 작성하는 일"이 아니라 "작성된 코드를 검증"하는 일이기 때문에, 테스트가 과제 일정에 가장 큰 영향을 준다. 따라서, 과제의 시작부터 코딩을 테스트와 분리된 활동으로 봐서는 안 된다.

과제의 후반은 대부분이 "테스트와 디버깅"으로 이루어진다. 하지만, 사실 과제를 이렇게 전반부와 후반부로 나누는 것도 오래된 "폭포수(Waterfall)" 모델에서 가정하고 있을 뿐이다. 좀 더 실용적인 과제 실행 방법은 지속해서 기능을 추가하면서 진화하는 방향으로 가는 것이다. 진화란 기능의 추가만을 이야기하는 것은 아니다. 기능이 추가될수록 기존의 코드가 엉망이 되어간다면, 기능을 더 추가하기 전에 기존의 코드를 정리하는 시간이 먼저와야 한다. 이렇게 대규모로 정리하는 시간이 필요하다는 의미는 이미 망가진 코드를 가지고 작업하고 있다는 뜻이다. 이런 순간이 오기 전에 과제의 시작부터 코드를 꾸준히 개선하는 활동을 해 왔어야 한다. 방법은 이미 많이 나와 있다. 어떤 방법을 쓰더라도 중요한 것은 코드가 좀 더 청결해졌고 이해하기 쉽고 간단해 보인다는 느낌이 드는 것이다. 코드나 설계는 "간단"해야 한다. 따라서, 과제의 초기에는 “코드를 어떻게 항상 청결한 상태로 관리할 것인가”를 팀원들이 함께 공유해야 한다. 필요하다면 교육을 먼저 해야 할지도 모른다. 교육 시간을 과제 기간에 포함하지 않는다면 이런 것들을 전부 과제 지연으로 보이게 만드는 활동이라고 생각될 가능성이 있다. 개인적인 교육도 물론 포함하는 것이 좋다. 팀이 과제를 더 잘하는 데 필요한 일은 과제와 관련되지 않은 활동이 아니라 과제 자체이기 때문이다.

시스템 전체의 품질은 시스템을 구성하는 각 부분 중에서 가장 낮은 수준에 맞춰진다. 따라서, 그런 부분들을 찾아내서 없애는 것이 중요하다. 구현하는 사람에 따라 품질은 차이를 보일 수 있지만, 전체적으로 봤을 때 품질을 만족하게 하는 것은 개인이 아닌 팀이 해야 할 일이다. 시스템의 특정 부분에 문제가 있다고 하더라도 팀 전체가 나서서 해결해야 할 일이 된다. 누군가를 비난하는 것보다는 먼저 나서서 해결책을 찾는 것이다. 초기 품질을 위한 준비보다 구현으로 빨리 넘어간다는 것은, 구현된 만큼 검증되지 않은 부분이 늘어난다는 의미다. 따라서, 구현을 빨리 많이 할수록 문제는 나중에 더 많이 발생할 것이다. 구현 활동 내에 검증 업무가 녹아 들어갈 수 있다면, 개발자가 말하는 "완료(Done)"라는 단어도 의미가 달라진다. 개발 초기부터 이런 "완료"라는 말의 정의를 팀의 모든 개발자가 이해할 수 있다면, 조금 늦게 가더라도 시스템에서 검증된 부분은 점차 늘어나게 된다. 과제 마감날이 다가오더라도 일정 수준의 품질을 "확신"할 수 있는 것도 이런 검증된 코드가 늘어나기 때문이다. 최소한 기존의 개발방법에서 발생했던 문제점 개수보다 반 이상 줄어든 볼 수 있을 것이다. 버그 하나를 수정하는데 들어가는 "시간"을 고려할 때 이는 결코 작은 비용이 아니다. 시장에서 기회 상실로 인해서 발생하는 이익의 감소와 인건비 등의 낭비를 생각하면 "일정보다는 제공된 기능의 품질"을 우선해야 한다. 하지만, "일정을 맞추기 위해서는 품질을 먼저 확보해야 한다”는 간단한 진리는 생각보다 사람들을 이해시키기 어렵다. “성급한” 관리자들은 머리로는 이해해도 몸으로는 전혀 실천하지 못하기 때문이다. 때로는 머리로 이해하는 것 자체도 불가능한 경우가 있다.

# [ 기능위주 개발전략 ]

기존의 개발방법에서는 모든 것을 결정한 후에 코딩으로 들어가는 것이 정석이었다. 이를 위해서 각종 문서의 작성과 리뷰(Review)등이 중요시되었으며, 코딩 시에도 개발자들은 자신들이 맡아서 해야 할 부분(모듈)에 대해서 작업을 분담하는 방식이었다. 이런 형태의 개발에서는 통합이 완료된 후에 기능에 대한 테스트를 할 수 있다. 통합이 자주 일어난다고 하더라도 전체 기능에 대한 테스트는 뒤로 미뤄질 수밖에 없다. 개별기능에 대한 테스트도 물론 개발에서 진행하기는 하겠지만, 개별 기능을 테스트하기 위해서는 하부에 필요한 모듈들이 구성되어야 한다. 계층화된 설계를 한 후 일을 블록별로 개발자들이 나누어 가지기에, 한가지 기능을 검증하는 것은 관련된 모든 모듈을 담당하고 있는 개발자들의 동의가 있어야만 가능하다. 즉, 더 많은 의사소통이 필요한 구조를 가지게 된다. 소통해야 할 필요가 늘어나면 오류가 끼어들 가능성이 커진다. 소통이란 사람과 사람사의 대화만 포함하는 것이 아니라, 개발자들이 담당하고 있는 모듈간의 호출 관계도 포함하기 때문이다. 따라서, 더 효과적인 개발 방법은 한 가지 기능을 전체 개발자가 함께 개발하는 것이다. 한 가지 기능의 개발이 완료되면, 다음 기능의 개발로 이동하게 된다. 이 때는 의사소통의 목표가 한 가지 기능에 집중되기에, 과제에 투입된 전체 인력의 대화 집중력을 높이는 데도 도움이 될 수 있으며, 전체 블록을 완성하기 전에 테스트를 진행할 수 있다.

이처럼 기능별로 구현하는 것은 언 듯 보기에는 설계가 완료되지 않은 상태에서 구현을 먼저 한다고 생각할지도 모르지만, 한 가지 기능을 구현하기 위한 충분한 설계는 먼저 선행되어야 한다. 기능별 구현에서 중복개발의 위험이 있다고 볼 수도 있겠지만, 툴을 이용한 검증과 시스템 전반에 대한 공통된 이해를 기반으로 하기 때문에 중복의 위험은 줄어든다. 또한, 완전히 밑그림(아키텍처)이 없는 개발이 아니라 대략적인 밑그림을 구성한 후에 기능별 세부 설계에 들어간다. 대부분 개발자도 기본적인 설계 활동이 필요하다는 것은 이미 충분히 이해하고 있다. 기능위주 구현에서 중요한 것은 구현된 기능의 완성도가 높아야 한다는 것이다. 즉, 기능 자체가 충분히 검증되어야 한다. 그렇지 않다면 결국 검증을 다시 뒤로 미루는 일이 발생하고, 과제의 막판에 가서 대부분의 시간을 테스트와 디버깅에 소모하게 된다. 이는 기존의 방법과 달라지는 것이 아무것도 없다는 뜻이다. 따라서, 기능을 구현하는데 필요한 테스트 방법을 충분히 익혀야 한다. 단위 테스트, 통합테스트, 시스템 테스트에 해당하는 각각의 방법을 고민해야 한다. 단위 테스트 및 통합테스트에 필요한 도구를 구해야 하고, 시스템 테스트를 위한 자동화 방안을 기본 설계부터 반영할 수 있어야 한다.

기능위주 개발전략은 중간에 데모를 자주 해야 한다. 얼마나 개발이 진척되었는지를 완료된 기능 위주로 "실행"해서 보여주는 것이 과제의 진행 상황을 가장 정확하게 전달하는 방법이기 때문이다. "몇 %가 구현되었다"라고 말하는 것은 의미 없는 일이다. 소프트웨어는 90%가 개발되었다고 하더라도, 나머지 10%를 채우기 위해서 다시 이전에 투입한 90%의 시간이 추가로 들어갈 수 있기 때문이다. 기능별로 개발하고 완성도를 높인다면 최악의 경우라도 개발 완료된 기능을 배포할 수 있다. 만약 구현되는 기능들이 우선순위가 높은 것들이라면, 우선순위가 낮은 기능들이 빠진다고 해서 협상이 불가능한 상황은 아닐 것이다. 추가로 우선순위가 낮은 기능들을 배포를 하는 일정을 보여줄 수 있다면, 일단은 완성된 기능들을 위주로 배포하는 것도 가능할 것이다. 그리고, 실제로 사용되지 않을 기능들을 구현하는 공수(Efforts)를 줄일 기회도 된다. 기능별로 개발자원을 집중하는 것은 전체 시스템을 이해할수 있도록 팀원들을 유도하며, 지식의 전수와 역량을 향상하는데도 좋은 도구로 사용할 수 있다. 대부분 개발자들은 몇몇 모듈에 오래동안 집중해서 전체 시스템이 어떻게 동작하는지 제대로 이해하지 못 하는 경우가 종종 있다. 그리고, 그런 상황이 지속하면 일에 대한 집중도가 낮아지고, 다른 일을 하고 싶어서 떠나가기도 한다. 기능위주 개발에서는 각각의 기능을 모든 개발자들이 이해할 수 있도록 만들어 주기에 특정 개발자가 과제에서 빠진다고 해서 치명적인 일정지연이 발생하지 않는다.

기능 위주의 개발 전락을 "깊이 우선 개발"이라고 한다면, 기존의 개발은 "폭 우선 개발"이라고 할 수 있다. 기능위주 개발을 할 경우에도 시스템의 근간이 되는 부분은 핵심이기에 반드시 먼저 개발되어야 한다. 전체 시스템을 구성하는 틀(Frame)이 되기에, 이를 제대로 구현하지 않는다면 다른 부분들이 제대로 구현될 수 없기 때문이다. 이 때 핵심은 주로 "입력을 받아서 출력을 생성하는 과정"의 주요 제어와 데이터 통로(Path)를 구성하는 부분과 시스템의 상태 변경을 담당하는 부분들이다. 이러한 핵심적인 부분은 전체 시스템의 주요 품질속성을 결정하는 하기에 프로토타입과 같은 것을 만들어서 미리 검증(Feasibility Study)을 하는 것도 좋은 방법이다. 또한, 모든 것을 내부에서 다 개발하려고 하는 것은 좋은 전략이 아니다. 활용할 수 있는 것들을 구할 수 있다면 최대한 코딩을 절약해야 한다. 코딩을 많이 할수록 테스트해야 할 부분도 늘어나고 버그가 발생할 가능성도 커진다. 이미 검증된 것이 있다면 그것을 먼저 활용해야 한다. 만약, 만족할 만한 기능이 제공되지 않거나 불필요한 기능들이 많다면, 나중에 대체하는 것도 신중히 생각해 볼 수 있다. 기능을 구현할 때는 그 기능에 대해서 잘 알고 있는 도메인 전문가가 필요하다. 도움을 받을 수 없는 상황이라면 해당 기능의 구현을 뒤로 미뤄두어야 한다. 먼저 구현할 수준이 될 정도로 이해가 되는 기능들을 위주로 개발을 진행하면 된다. 나중에 좀 더 많은 도메인 지식을 얻게 되었을 때 그 기능을 구현하도록 해야 할 것이다. 전체 시스템의 구조에서 제어의 흐름을 관리하는 부분은 도메인 지식과 무관할 수 있으니, 그 기능을 뼈대로 초기에 개발을 진행할 수도 있을 것이다. 예를 들어, 실행 인프라 구조와 같은 것들은 미리 구현할 수도 있다. 하지만, 이것도 하나의 기능이 완료될 수 있을 정도만 구현해야 할 것이다.

# [ 하고 있는 일 줄이기 ]

한 번에 한 가지 일만 하는 사람은 능력이 떨어지는 사람이 아니라 집중력이 좋은 사람이다. 이는 보는 관점에 따른 차이며, 여러 가지 일을 동시에 하는 사람보다 더 빨리 일을 처리한다는 사실을 알아야 한다. 반응시간이 빠른 것을 원한다면 일을 작게 나누는 것이 좋다. 마치 여러 가지 일을 동시에 하는 것처럼 보이려고 한다면, 작은 일을 빨리 처리하고 다음 작은 일로 넘어가면 된다. 하지만, 이 때 발생하는 처리해야 할 일의 차이로 인한 "컨텍스트(Context) 전환 비용”을 줄이는 것이 핵심이다. 컨텍스트 전환 비용을 줄이려고 한다면, 자신이 잘하는 분야의 일들을 맡으면 되지만, 이럴 경우 경험이 고착되어 성장의 깊이만 따지고 폭은 보지 못하게 된다. 일은 작게 나눌수록 더 빨리 결과를 확인할 수 있다. 모호하게 일을 주면 받는 사람의 입장에서는 다시 일을 정의하기 위해서 시간을 보내야 한다. 일을 작게 나누는 것을 같이하고, 이렇게 나누어진 각각을 담당자가 선택해서 가져가도록 하는 것이 좋다. 이는 자신의 일 처리 속도에 맞게 리듬을 유지할 수 있기에 결과적으로 더 빠르고 정확한 결과를 가져올 것이다.

일을 줄이는 것은 일에 대한 완료 기준을 높이는 것과 같이 병행해야 한다. 즉, 일을 작게 만들었다면 각각의 일에 대한 완성도를 높여야 한다는 의미다. 완성도를 높이는 것은 검증을 동반하는 것으로 검증(테스트) 자동화 및 오류를 걸러줄 수 있는 필터(Filter)를 만들어야 한다. 자동화는 테스트 스크립트를 만드는 것으로 가능하며, 필터는 코드를 리뷰하고 정적분석(문법 오류, 문맥 오류, 복잡도 제거, 중복코드 제거 등), 코딩 규칙 등을 동원해서 해결할 수 있다. 따라서, 이런 것들이 일상화되어 작은 단위의 일들에 대한 검증이 지속적으로 이루어진다면, 해야 할 일들이 더 빨리 처리되는 것을 경험하게 될 것이다. 버그를 만들어내는 곳은 개발 현장이며 바로 그곳에서 버그를 제거하는 활동이 가장 효과적이다. 만드는 곳과 검증하는 곳이 분리되면 될수록 버그를 수정하는 비용이 늘어나게 된다는 것은 상식적으로 이해가 가능하다. 일의 크기를 줄이고 한 번에 처리하는 일을 최소화시켜주면 개발자는 완성도를 높이는데 집중할 수 있게 된다. 따라서, 시간이 흘러갈수록 완성도 높은 일들이 쌓이게 되며, 그런 일들 간에 생기는 영향을 고려한 빠른 통합도 자주 할 수 있게 된다. 통합을 자주 하게 되면 당연히 코드 간의 영향도를 미리 검증할 수 있는 환경을 만들 수 있기에, 나중에 한꺼번에 통합하는 것보다 더 작은 단위에서 검증할 수 있다. 완성도가 높은 코드를 통합 비용이 더 쌀 것이라는 점은 충분히 짐작할 수 있다.

일을 줄이기 위해서는 쌓아두는 공간을 줄여야 한다. 보통의 경우 개발팀은 자신이 해야 할 일을 쌓아두고 순차적으로 빼내는 방식을 사용하지만, 이럴 경우에는 일의 우선순위를 고려하기가 쉽지 않다. 대부분의 경우 이것도 하고 저것도 해달라는 말만 들을 뿐이다. 그렇게 할 수 없다고 이야기한다면, 더 높은 직급의 사람의 "지시사항"이라는 명목으로 일이 떨어질 것이다. 흔한 일로 과제를 진행하다가도 높으신 분이 와서 하라는 일은 절대 거절할 수 없는 명령으로 받아들여진다. 과제가 깨지더라도 그걸 해야 하는 상황이 생긴다. 물론 과제의 지연이나 연기는 절대 없다. 나중에 과제가 지연되었다는 보고를 한다면, 당연히 자신이 지시했던 일은 생각하지 못하고 지연된 과제만 탓하게 될 것이 분명하다. 결과에는 원인이 있다는 생각은 절대 못 하고, 과정은 무시한체 결과만 문제 삼을 것이 분명하다. 버그도 마찬가지다 과정을 제대로 거치지 못한 결과물에 버그가 없기를 바라는 것은 최정상급의 개발자도 할 수 없는 일이다. 모든 개발 툴에 디버깅 기능이 있듯이, 최고의 개발자가 사용하는 개발 툴에도 디버거는 있다. 그들 또한 실수를 하지만 회복하는 시간이 빠를 뿐이다. 방법은 단순하다. 코딩과 테스트를 동시에 하는 방법을 사용한다는 점이다. 기술을 알고 있는 것과 구현하는 것은 다르다. 기술은 공부를 통해서 알고 구현을 통해서 탄탄해지지만, 구현역량은 실수를 통해서 길러지기 때문이다. 실수를 한다고 실패하는 것은 아니다. 실수를 통해서 해서는 안되는 것을 하나 더 익혔다는 것이 중요하다.

어떤 식으로든 지금 진행 중인 일을 줄여주어야 한다. 많은 회의(주간, 월간, 품질 등등의 회의)와 다양한 TF, 갑자기 떨어지는 명령, 부서장의 호출 등, 회사에서는 개발자의 집중을 방해하는 것들이 정말 많다. 일을 줄이기도 해야 하지만 일을 집중해서 할 수 있도록 만들어 주어야 한다. 소를 키우는 사람이 있어야 식탁이 풍요로울 수 있다. 밥을 짓는 사람은 하나인데 밥을 먹으려고 숟가락을 들고 이래라 저래라 하는 뒤짐지고 선 사람들이 많으면 일다운 일이 줄어든다. 즉, 일의 속성이 다르다는 것이다. 개발 활동에는 직접 구현하는 일 이외에도 부가적으로 해야 할 일들이 많다. 요구사항을 정리하거나 일정계획을 세우는 일 등, 과제 개발에 관련된 활동과 그 외에 행정적인(사무적인) 활동 및 위사람의 명령 이행 등의 부가적인 일들이 발생한다. 이 둘 간의 비율이 8:2정도라면 어느 정도 과제에 집중할 수 있는 상황이 되지만, 7:3이나 6:4정도의 비율이라면 과제 계획을 세울 때 거의 30%에서 40%의 기간을 더 추가해서 넣어야 한다. 그것을 일정 버퍼라고 생각해서는 안된다. 그것은 버퍼가 아니라 엄연히 회사에서 일정을 잡아먹는 오버헤드다. 하지만, 높으신 분들은 그런 것을 인정하려고 하지 않는다. 자신들이 필요한 시간은 남에게서 뺏았어 오지만, 정작 타인에게 필요한 시간은 절대 용납하지 않는다. 높은 분은 아무 때나 사람을 불러서 회의할 수 있지만, 실무자들은 자신의 시간을 쪼개서 대응해야 한다. 부족한 시간은 야근이나 주말 특근을 끌어들이는 수밖에 없는 것이다. 일을 적게 하기 위해서는 필요한 일만 해야 하고 개발에 관련된 시간의 비율을 최대한 확보해 주어야 한다. 주 52시간이 되어도 해야할 일만 할 수 있도록 최대한 하고 있는 필요없는 일을 줄여 주어야 한다.

# Part III. Product

# [ 글로벌(Global) 소프트웨어? ]

소프트웨어를 만든다는 것은 참으로 어려운 일이다. 물론, 어떤 일을 하더라도 어렵지 않은 것은 없겠지만, 눈으로 확인하고자 하는 사람들에게 소프트웨어는 보여주기가 어렵다는 측면에서 힘든 분야 중에 하나라고 할 수 있다. 개발자들은 제품의 개발 일정에 맞춰서 일을 해야 하지만, 항상 개발은 늦어지고 버그(Bug)는 늘어난다. 이런 저런 회의로 시간을 다 빼앗기고 있는 상황에서 다시 문서로 만드느라 제대로 정리되는 것도 없다. 나중에 결과물로 만든 소프트웨어가 시장에 나가게 될 때는 더 많은 일이 생긴다.

테스트 룸(Room)에서 몇 명이 하던 테스트를, 시장에서는 다양한 사람들이 정말 이해되지 않는 방식으로 자신 만의 테스트 결과를 인터넷에 올린다. 팀장은 그런 글만 모아서 회의시간에 개발자를 깨기 바쁘다. 야근은 밥 먹듯이 해야 하고, 야근 끝에 좀 쉬려면 다시 다른 신규 프로젝트를 시작해야 한다. 좀 더 심하면 AS와 새로운 프로젝트를 동시에 하게 되기도 한다. 제품이 늘어날수록 AS해야 할 것도 많아지고, 점점 더 많은 개발자를 고용해야만 제대로 시간을 맞춰서 고객에게 대응할 수 있다. 하지만, 단언 하건데 이런 것은 기대하지 말라. 소프트웨어 개발에만 한정된 것은 아니지만, 될 수 있으면 새로운 충원 인력은 기대하지 않는 것이 좋다. 사람은 시간과 돈에 따라 고용될 수 있고, 또는 눈앞에서 갑자기 사라질 수 있다. 항상 프로젝트 계획서에는 "충원"이라는 이름의 사람이 몇 명씩 등장하지만, 그 사람은 어디서 무엇을 하는 사람인지 과제가 끝날 때까지 보지 못하는 경우가 대부분이다.

소프트웨어를 만드는 것이 어렵지만 어려운 것을 안 할 수는 없다. 개발을 해야지 회사도 나도 이익을 같이 공유할 수 있다(비율은 틀리겠지만). 남들이 다 하는 것을 한다면 결과적으로 경쟁력이 없으며, 언젠가 더 싸고 좋은 제품이 나오면 고객은 쉽게 마음을 바뀌게 된다. 어렵다고 안 하는 것이 아니라, 어려우니까 쉽게 할 수 있는 방법은 없을까? 회사에 처음 입사를 하면, 대부분의 경우 선배가 몇 개의 읽어야 할 책과 지금까지 개발된 소스코드를 준다. 물론, 소스 코드가 안 주어지는 경우도 많고, 그냥 "이거 해라"라고 말만 하기도 한다. 대부분의 경우 개발할 내용 만을 말로 전달한다.

좀 더 개선된 방법으로 Email로 날아 오기도 한다. 하지만, 그 일을 처음부터 잘 할 수 있다고 기대하지는 말자. 공부한다고 몇 개월은 흘러야 할 것이다. 그리고, 이제 제대로 코딩하려고 하면 이런저런 자질구레한 일들을 단지 신입이라는 이유만으로 해야 한다. "과연 내가 무엇을 하고 있나?"하고 진지하게 고민할 때가 되면, 짧지 않은1년이란 시간은 의미 없이 흘러가 버리고 말 것이다. 그럼 누가 도대체 왜 이렇게 만든 것일까? 사실 가장 잘못한 사람은 본인 스스로라고 생각하면 맞을 것이다. 혹은 과거의 본인이거나 미래의 본인이 될 수도 있다. 과거의 본인은 본인과 같이 개발자로 일하는 선배, PL, 팀장, 혹은 사장이 되어 있을 것이고, 미래의 본인은 그런 사람들 밑에서 본 것 없이 자란 또 다른 선배가 될 “나” 자신인 것이다. 지금 상황은 이미 10년이나 그 이전의 시간에 심어진 씨앗들이 자란 결과이며, 선배의 잘못된 관행을 후배가 반복해서 따라하고 있을 뿐이다. 그럼 그냥 이대로 두는 것이 좋을까? 당연히 그렇지 않다.

잘못 자란 나무가지는 잘라내고, 새로 씨앗을 뿌리거나 묘목을 심어야 한다. 첫 번째 해야 할 일은 바로 프로젝트를 시작할 때 “문서 포맷”을 만드는 일이다. 작성할 문서의 종류를 정하고, 형식을 정하고, 코딩 방식을 정하고, 모든 사람이 쉽게 접근해서 수시로 변경할 수 있는 곳에 저장해 두어야 한다. 형식은 귀찮은 것이지만, 일관된 모습으로 정리해주는 기초적인 방법이다. 두 번째로 해야 할 것은 “공유”다. 문서의 공유와 소스코드의 공유, 비전의 공유, 회사의 목표 및 성과에 대한 공유 등이다. 공유는 불법적인 것을 빼면 다 좋은 방향으로 가기 마련이다.

의사소통도 공유의 한 방법이다. PL과 임원, 사장, 팀원이 동일한 비전을 공유하고, 그것에 매진할 때 프로젝트가 제대로 돌아갈 것이다. 세 번째는 “팀워크”다. 축구를 하더라도 혼자서 열심히 뛰어서는 골을 넣을 수 없다. 현대 산업화된 사회에서는 혼자서 할 수 있는 일은 거의 없다. 누군가의 도움을 빌리거나 에코 시스템(Eco-system)에 속하지 못하면, 거의 자급자족처럼 살 수밖에 없다. 소프트웨어 개발의 어쩌면 가장 중요한 요소가 팀워크일 수도 있다. 한 사람이 너무 튈 경우에는 전체 팀워크를 해칠 가능성이 높다. PL의 권한이 너무 비대 해서 이런 저런 일에 다 나선다면, 나중에 팀원들은 입을 닫고 수동적으로만 움직일 것이다. 생각을 배제한 행동만을 강요한다면, 사람이 아닌 도구로 보는것밖에 되지 않는다.

소프트웨어는 아직도 사람이 만드는 것이며, 사람의 손으로 이루어진 것 중에서 가장 복잡한 체계를 구축해 나가고 있다. 전 세계 수억에서 수십억의 사람들이 소프트웨어의 도움을 매일 매일 받으면서 살고 있고, 또한 다양한 곳에서 다양한 형태로 개발되고 있다. 사람이 집단으로 모인 곳에서는 특정 집단에 맞는 문화가 생긴다. 문화는 공유하는 생활 양식이라고 배웠는데, 소프트웨어를 만드는 사람들이 모인 곳에서는 그런 문화가 필연적으로 생길 수밖에 없다. 그러면, 어떤 문화가 개발에 도움이 되는 문화일까? 그 문화를 만들어가는 주체는 누구이며, 또한 누가 그런 문화를 만들도록 해 주어야 할까? 답은 뻔하다. 팀에 구성된 사람들 모두가 그런 문화를 만드는 데 일조를 해야 한다.

PL은 개발자가 개발에만 집중에서 일할 수 있는 최적의 환경을 만들어 주는 데 노력해야 하며, 팀의 구성원들은 자신의 업무에 집중해서 “품질”이 우수한 문서와 코드를 생성해 내야 한다. 고객의 의견을 잘 들어서 SRS(Software Requirements Specification)를 만들고, 이를 바탕으로 제대로 설계하고, 구현 및 테스트를 정확히 해내는 문화를 만들어야 한다. 누가 언제 무엇을 어떻게 왜 고쳤는지를 명확히 하고, 개인이 맡은 모듈에 대해서 문서화 및 자동화된 테스트를 붙여주며, 지속적으로 기능을 업그레이드시켜 만들어 가지 않는다면, “제대로 된” 소프트웨어 개발 문화는 만들어내지 못할 것이다. 문화라고 있는 것이 야근(주말 특근)이나 술집에서 원샷을 강요하는 것만이 있는 회사라면, 이제 다른 생각을 해야 할 때다. 회사를 옮기는 것은 가장 마지막의 선택이다. 자신이 바뀌지 않는다면 어떤 변화더 일어나지 않는다.

정리하자면, 결국 소프트웨어 개발은 사람이 하며, 사람은 개발에 적합한 문화를 가진 조직에서 최대의 성과를 발휘할 수 있다. 중요한 것은 그런 문화를 개발에 관련된 사람들이 스스로가 만들어가지 않으면 아무도 나서서 해주지 않는다는 것이다. 급한 일만 하다 보면 급하게 회사의 수익이 기울어가고, 중요한 일을 앞서서 하면 급한 일은 줄어들기 마련이다. 중요하지만 급하지 않은 일은 시간이 지나면 급하고 중요한 일이 되기 때문이다. 멀리 갈 사람은 신발 끈부터 제대로 매듯이, 하나씩 차분히 준비해 나간다면 소프트웨어 개발자가 나의 자녀에게도 매력적인 직업이 될 수 있는 날이 반드시 올 것이다.

# [ 운영체제는 누가 움직이지? ]

사람들은 간혹 운영체제(OS: Operating System) 자체가 특별한 형태로 스스로 동작하도록 만들어진 응용프로그램 같을 것이라는 생각을 한다. 이렇게 이해하는 것도 어찌 보면 당연하다. 즉, 자신이 만든 프로그램에서 OS에서 제공하는 함수들을 호출하는 과정이 마치 누군가에게 일을 대신시키는 것과 같이 보이기 때문이다. 엄밀히 말한다면 그냥 함수의 호출이고, 호출의 주체는 응용 프로그램이며, 호출을 수행하는 주체도 사실은 응용 프로그램의 연장이라고 보는 것이 옳다.

응용 프로그램에서 호출된 OS의 API(Application Programming Interface)는 수행되는 환경만 OS속에 있을 뿐이지, 항상 호출한 응용 프로그램이 가진 컨텍스트(Context)에서 동작한다. 즉, 해당 API를 호출한 응용프로그램이 있다는 뜻이다. 따라서, 누가 호출했는지를 알고, 누구에게 다시 호출이 끝나서 돌아갈지도 알 수 있다는 뜻이다. 물론, 호출이 마무리되는 순간에 다른 응용프로그램이 끼어들 수도 있지만, 어쨌든 중요한 점은 누가 호출했는지 명확하다는 점이다. 여기가 바로 OS의 진입점(Entry Point) 중의 하나(시스템 호출)이다.

OS로의 진입점은 앞에서 이야기한 OS가 제공하는 API(시스템 호출) 이외에 다른 한 가지가 더 있다. 즉, 전체 시스템에 마치 심장과 같이 끝없이 피를 공급해주는 역할을 하는 요소다. 그것이 바로 인터럽트(Interrupt)이다. 인터럽트는 CPU에서 하던 일을 잠시 멈추고 급하게 처리하고자 하는 이벤트(Event)다. 이러한 이벤트 중에서도 가장 중요한 요소가 바로 주기적으로 일정한 간격으로 발생하는 시간에 관련된 인터럽트다. 일정한 주기로 10ms나 100ms등을 설정(1ms도 가능하다)해서 끝없이 인터럽트를 알려주면, 그것을 처리하기 위해서 OS가 동작하게 된다. 따라서, OS로의 또 다른 진입점이 만들어질 수 있으므로, 이것을 두 번째의 진입점이라고 생각할 수 있다. 하지만, 이 두 번째 진입점의 경우에는 앞에서 설명한 OS의 API와는 다른 성격을 가지고 있다. 즉, 누가 호출했는지 주체가 모호하다는 점이다. 물론, 인터럽트가 발생한 곳은 알 수 있지만, 인터럽트가 발생한 시점에 실행되고 있던 응용프로그램과는 전혀 무관하다. 따라서, OS는 인터럽트를 즉각 처리할 수 있는 응용프로그램(프로세스: Process)을 알 수 없기에, 인터럽트를 기다리고 있을지도 모르는 응용프로그램의 상태를 변화시켜 처리 기회를 제공해줄 뿐이다.

정리하자면, OS는 응용프로그램의 호출과 인터럽트에 의해서만 동작한다. OS는 특정한 프로세스와 같은 스케줄링의 한 대상이 되는 것이 아니라, 호출이나 인터럽트의 연장선에서 이를 처리하기 위한 함수와 같은 역할만 수행할 뿐이다. 따라서, 좀 더 극단적으로 이야기하자면, 잘 정의된 라이브러리(Library)나 이벤트 처리자(Handler) 정도의 역할을 수행한다고 볼 수도 있을 것이다.

# [ 운영체제의 역할 ]

이제부터는 운영체제(Operating System)에 대한 이야기를 하나씩 해보록 하자. 최근에 iPhone OS라든가 안드로이드(Android) OS등 다양한 스마트 폰(Smartphone)용 운영체제가 나왔다고들 말한다. 하지만, 엄밀하게 말하면 이것들이 운영체제를 의미하는지는 다시 한번 생각해 봐야 한다. 전통적으로 운영체제라면 다음과 같은 것들을 다룬다.

1) **스케줄링(CPU Scheduling)**: CPU를 어떤 작업이 어떤 순서대로 사용할지 정하는 것.

2) **파일 시스템(File System)**: 저장장치에 파일을 읽고 쓰는 것.

3) **메모리 관리(Memory Management**): 메모리를 할당 해제하는 것

4) **네트워크(Network)**: 시스템의 내부 혹은 외부와 통신하는 것.

이렇게 크게 4가지의 영역으로 나누면, 사실상 GUI(Graphic User Interface)와 같은 그래픽적인 요소들은 OS의 큰 기능이 아니다. 물론, 사람들이 많이 관심을 가지지만 접근하기 어려운 디바이스 드라이버(Device Driver)같은 것도 포함되지 않냐고들 하는데, 이는 대부분 파일 시스템의 일부로 이미 포함되어 있다. 네트워크는 TCP/IP와 같은 인터넷을 사용하는 프로토콜 만을 포함하는 것이 아니라 IPC(Inter-Process Communication)도 포함한다. 즉, 서로 정보를 주고받는다는 것을 의미한다.

iPhone OS의 경우에 핵심(Kernel)이 되는 부분에 Mach 커널(Kernel)과 BSD를 혼합해서 사용한다고 알려져 있다. Android OS의 경우에는 Linux가 그 역할을 대신 수행하고 있다. 따라서, 이러한 커널 만을 한정해서 운영체제로 본다면, 당연히 iPhone OS와 Android OS라는 말은 옳은 표현이 아니다. 오히려 iPhone 플랫폼(Platform)이니 Android 플랫폼이라는 단어가 더 어울린다. 즉, 앞에서 이야기한 4가지의 핵심적인 기능은 내부의 커널(Kernel)이라고 불리는 곳에서 다 처리가 되기 때문이다. 물론, iPhone이나 Android가 통신이나 메모리 할당 등과 같은 목적으로 서비스 계층을 정의하고 있지만, 그렇다고 스케줄링이나 파일 시스템을 커널이 아닌 수준에서 정의하고 있는 것은 아니다.

하지만, 커널만 있다고 시스템을 구성하는 것이 끝일까? 그렇지 않다. 우리는 응용프로그램을 특정 운영체제에서 동작시키기 위해서 커널 이외에 많은 부분이 관련된다는 것을 이미 알고 있다. 예를 들어, 명령어를 입력받아서 실행을 대행해 주는 쉡(Shell)이나 공용으로 사용되는 라이브러리(Library)들이 그러한 것들이다. 따라서, 이러한 모든 부분을 다 합쳐서 이야기하는 경우가 운영체제의 일반적인 정의가 될 수도 있다. iPhone이나 Android의 경우에는 더 많은 것들이 추가되는데, 즉 프로그램을 더 쉽게 개발하고, 더 쉽게 사용될 수 있는 부분들이 녹아들어 있다. 개발자는 자신이 하고자 하는 일에 집중하고, 기타 잡다한 일들을 대신 처리하는 부분들이 기본적으로 미리 제공되고 있다.

정리하면, 일반적으로 신문이나 잡지에서 일상적으로 사용하는 용어에 대한 정확한 이해가 필요하며, 왜 그렇게 표현하는지 알아야 한다. 피상적으로 들어서 그냥 그렇거니 하는 것 보다 좀 더 자세한 용어에 대한 정의를 알고나면, 사물에 대한 더 깊은 통찰이 가능해진다. 세상은 "아는 만큼 보인다." 뒤집어 말하면 "모르면 속기 마련이다”라는 뜻이 된다.

# [ 프로그래머 십계명 ]

프로그래머로 살아가면서 10가지 지켜야 할 계명이 있다면 무엇일까? 그냥 우연찮은 기회에 성서에 나오는 십계명과 유사하게 만들면 어떨까 하는 생각으로 아래와 같이 간략하게 정리해 보았다.

**01. 방법론을 섬기지 말라.**

**02. 남이 만든 코드에 전적으로 의지하지 말라.**

**03. 특정 방법론을 따른다고 말하지 말라.**

**04. 빨간 날은 반드시 쉬라.**

**05. 소스 관리 시스템을 공경하라.**

**06. 모듈의 상하 관계를 구분하라.**

**07. 관련 없는 모듈과 통하지 말라.**

**08. “Copy&Paste”를 남발하지 말라.**

**09. 나의 버그를 남의 것이라고 말하지 말라.**

**10. 내 이웃의 개발자를 탐하지 말라.**

좀 뻔한 이야기들이지만 이번에는 이것들을 잠시 이야기해보도록 하겠다. 각각의 항목에 대해서 그냥 편하게 풀어나가 보겠지만, 말 솜씨가 부족해서 어려운 부분이 있다면 저자가 아직 실력이 부족한 탓이라고 생각하기 바란다.

**01. 방법론을 섬기지 말라.**

흔히들 "우리는 어떤 방법론으로 개발하고 있어"라는 이야기들을 많이 한다. 하지만, 정작 중요한 것은 그러한 방법론이 아니라, 체질화된 습관이라고 하는 것이 더 옳을 것이다. 예나 지금이나 프로그램을 개발하는 것은 거의 비슷하며, 어떻게 하면 좀 더 가시적으로 남들에게 소프트웨어 프로그래머가 일하고 있다는 것을 보여줄 것이냐만 바뀌었다고 생각한다. 물론, 이렇게 이야기하면 반론을 제기할 분들이 많겠지만, 근본적인 프로그램을 개발하는 방법은 초창기와 그렇게 다르지 않다.

빨리 만들어서 빨리 보여주고, 빨리 고쳐서 언제까지 할 수 있을지 보여주자는 것 등등의 방법론을 따르고 있다고 이야기하기 전에, 정말 그러한 방법론이 우리 현실과 맞는지 확인해 봐야 할 것이다. 만약 잘 안 맞는 옷을 억지로 입었다고 판단된다면, 그러한 방법론보다는 현장에서 적절히 할 수 있는 조치를 취하는 것이 옳다. 방법론은 참고할 만한 것이지 모든 문제의 정답은 아니기 때문이다. 또한, 그러한 방법론을 쓰고 있다고 이야기들을 하지만, 정말 그런가? 혹은, PL만 그렇다고 믿고 있는 것은 아닌지 질문을 자기 자신에게 질문을 던져보기 바란다. 섬기지 말고 복종하도록 만들어야 하고, 습관처럼 그냥 일상적으로 사용하는 것이어야 한다.

**02. 남이 만든 코드에 전적으로 의지하지 말라.**

남이 만들어 놓은 코드들을 가져다가 사용하는 경우가 아마도 10에 8~9정도는 될 것이다. 자신이 만든 제품에서 남이 만든 코드를 대부분 90% 이상은 사용하고 있을 것이다. 물론, 완전히 처음부터 만들어 나가는 과제도 있지만, 그렇지 못한 과제가 대부분이다. 프로그래머의 대부분 일은 어쩌면 이렇게 복제(copy)하는 일일지도 모른다. 하지만, 남이 만든 코드이기 때문에 전적으로 옳다고 생각하지는 말자. 반드시 호출(call)의 결과는 확인되어야 하며, 문제가 발생할 가능성도 얼마든지 있다.

“가장 좋은 코드은 자신이 코딩하지 않는 것”이라는 우스갯소리가 있다. 이는 이미 만들어진 잘 동작하는 코드를 새로 개발할 필요는 없다는 것이지, 그 코드가 전혀 문제가 없다는 말은 아니다. 따라서, 항상 습관적으로 어떤 함수를 호출한 후에는 반드시 호출의 결과를 확인해야 한다. OS에서도 문제가 발생할 수 있으며, 3rd Party에서 제공한 코덱과 같은 곳에서도 언제든 에러가 생길 수 있다. 항상 방어적으로 코딩해야 그만큼 오류도 적게 만든다. 공격적으로 코딩 속도를 가져가기도 해야 하겠지만, 문제는 그냥 넘어가는 곳에 항상 숨어 있다. 모든 코드를 한 줄 한 줄 다 보는 것은 어렵겠지만, 최소한 하나의 함수 단위로 확인하는 것은 충분히 가능하다.

**03. 특정 방법론을 따른다고 말하지 말라.**

사실 특정 방법론을 따른다고 이야기를 하는 사람들이 프로젝트를 망칠 가능성이 더 높을 수 있다(물론 반드시 그런 것은 아니지만). 하지만, 그렇게 말하는 것이 더 똑똑해 보일 수는 있을 것이다. 일을 잘 할 것처럼 보이는 사람들은 이론에 밝은 사람들이다. 자신의 의견을 논리적으로 잘 표현하고, 멋진 방법론을 가져다가 장점 만을 취해서 개발하고 있다고 이야기할 수도 있기 때문이다. 하지만, 말과 실제는 다르다. “벼는 익을수록 고개를 숙인다”는 말이 있듯이, 개발에 정말 이골이 난 사람은 방법론에 의지하지 않는다. 오히려 프로젝트의 정확한 “맥”을 짚어서 어디서 문제가 발생할 것 같고, 어떻게 하면 그것을 해결할 수 있다는 것까지도 이미 알고 있다.

풍부한 실전 경험만큼 좋은 공부는 없다. 물론 이론적인 토대가 없이 이루어지는 경험은 지속적인 성공을 보장해 주지 못한다. 하지만, 무서운 것은 이론도 잘 알면서 실전 경험도 갖춘 사람들이다. 특정 방법론을 따른다고 말하기보다 "내가 생각하기에 이번에는 이런 식으로 해보면 어떨까?"라고 이야기 하면서, 묵묵히 자기 일을 해나가는 사람들이 여러분의 곁에 두는 것이 좋을 것이다.

**04. 빨간 날은 반드시 놀아라.**

회사는 이익을 극대화하기 위해서 모인 조직이라고 말을 한다. 회사의 이익이 개인의 이익으로 돌아오기 위해서는 최소 6개월에서 1년이 걸린다. 즉, 보너스나 연봉 계약을 해야지 개인의 이익이 된다. 빨리 이익을 조금이라도 나누어 가지고 싶다면 빨간 날도 나와서 일하라. 물론 프로젝트를 지원하는 부서의 경비가 얼마나 될지는 몰라도, 그런 사람들에게 휴일근무 수당으로 다만 얼마라도 줄 것이다. 하지만, 그렇게 일하면 과연 능률(혹은 생산성)이 오를까? 보통의 직장인이라고 한다면 주말에 나와야 할 경우, 금요일까지 해야 할 일을 연장하는 경우가 많다. 토요일 어차피 나와야 한다면 금요일까지 일을 끝낼 필요가 없다는 것이다. 또한, 주말에 나올 경우 회사는 개인에게 수당을 줘야하며, 각종 부대시설과 식사도 준비해야 한다.

집으로 가져가는 돈도 돈이지만, 회사에서 낭비되는 돈도 늘어나게 된다. 능률이 단순한 산술식으로 성과를 일한 시간으로 나눈다고 본다면, 일한 시간이 두배로 늘어나면서 성과가 보통 때의 두 배로 늘어나지 않으면, 생산성 향상은 이루기 어렵다. 실제로는 일정의 일부 단축만 있을 뿐이다(물론, 그 일정 단축이란 나중에 다른 비용의 발생으로 이어질 수 있다). 그러한 일정 단축에 소비된 비용이 있다는 것을 반드시 기억해야 한다. 그리고, 실제로 놀지 않으면 같은 시간을 일하더라도 집중력은 떨어지게 된다. 야근한 다음 날의 오전 시간은 고스란히 낭비된다. 따라서, 과감히 빨간 날은 놀아라. 놀지 않으면 일하는 시간만 길어질 뿐, 결코 생산성이 높아지지는 않는다.

**05. 소스 관리 시스템을 공경하라.**

소프트웨어를 개발하는 회사라면 적어도 하나 이상의 소스 관리 시스템을 사용한다. 소스 관리 시스템에서 유지하는 것은 소스 코드의 모든 변경 사항과 변경 사유, 변경한 시간과 담당자에 대한 log이다. 이렇게 좋은 기능을 두고 어떤 사람들은 소스 코드 자체에 자신 만의 comment를 남겨둔다. 혹은, 데드 코드(Dead code)를 그냥 유지해서 코드의 가독성(Readability)이나 수정을 방해한다. 소스 관리 시스템이 없는 회사도 존재한다. 자신이 만든 코드를 남들이 보지 못하는 곳에 두고, 프로젝트를 진행하는 팀원들에게는 라이브러리로 바이너리(Binary) 파일 만을 제공하기도 한다. 자신이 가진 노우하우(Knowhow)를 마치 대단한 것처럼 생각해 남들과 공유를 하지 않지만, 남들이 잘 만든 코드는 항상 자신의 것으로 만들려고 노력한다. 이는 작게 보면 고립된 프로그래머로서 끝나는 것이고, 크게 보면 회사가 영속하지 못하는 이유가 된다.

단언하건데 발전하고자 한다면 공개하라. 남들의 비판(혹은 조언)이 두렵다면 회사를 그만두고 나가는 편이 낫다. 같이 일하는 팀원들을 믿지 못하고 두려워한다면, 제대로 프로젝트가 될 리가 없다. 항상 먼저 솔선수범해서 모든 코드를 공유해야 한다. 소스 코드 관리 시스템이 없는 세계를 선호하는 사람이라면, 반드시 혼자서 개발하기 바란다. 남들까지 고생시킬 필요는 없기 때문이다.

**06. 모듈의 상하 관계를 구분하라.**

사람들은 그림을 그리기를 좋아한다. 그림은 실제 세상을 반영하는 그림도 있을 것이고, 상상의 어떤 것일 수도 있다. 하지만, 어쨌든 사람은 그림으로 자신의 내면을 형상화시킬 수 있는 능력이 있다는 것이다. 실제 동작하는 프로그램을 만들려고 할 때, 이러한 추상화 과정이 필요하다. 어떤 일을 처리하기 위해서 유사한 기능들은 하나로 모아서 모듈(module)로 정의하고, 어떤 것들은 그러한 모듈에서 제공되는 기능을 이용해서 다시 만들어지기도 한다. 어떤 모듈들은 특정 모듈들로 향하는 호출이 발생하기도 하고, 호출되는 모듈들은 또 다른 모듈들을 호출하기도 한다. 이렇게 호출하는 모듈들을 상위에, 호출되는 모듈들을 하위로 구분하게 되면, 생각 자체가 하위로 갈수록 구체적으로 변하게 되며 상위로 갈수록 추상화하는 경향이 생긴다.

상하 관계가 명확하지 않게 되면, 호출들이 순환(cycle)을 이루는 구조가 생기게 되며, 순환에 관련된 어느 한 모듈을 수정하면, 그와 연결된 다른 부분에 대한 수정이 발생할 가능성도 높아지게 된다. 일반적으로 이러한 상하 관계를 특징지을 때 “계층(Layer)이라고 한다. 이렇게 계층을 만들어서 구조화시킨 것을 계층화된 아키텍처(Layered Architecture)라고 부른다. 이런 모델을 볼 때는 단순히 모듈을 쌓아 나간다고 생각하지 말고, 생각의 추상화를 높여나간다고 보기 바란다. 일반적으로 추상화가 높은 수준(Level)에서 하는 활동이 지적으로 높은 수준이 필요하다고 보는 경향이 있는데 반드시 그렇지는 않다. 시스템은 손발이 없이 머리로만 만들 수는 없기 때문이다. 가장 구체적인 명령어에서 부터 잘 구현되어야 탄탄한 구조를 가지기 때문이다. 마치, 높은 건물을 짓기 위해서 지하에 수없이 많은 철근 빔(Beam)을 꽂아 넣는 것과 같기 때문이다.

**07. 관련 없는 모듈과 통하지 말라.**

관련이 없다는 말은 특정 모듈에서 제공하는 기능을 이용하기 위해서 억지로 끼워 맞추지 말라는 뜻이다. 만약, 그렇게 접근을 해야 일이 생긴다면, 차라리 해당 모듈에서 그 기능을 분리하는 것이 옳다. 아니면, 추상화 수준을 한 단계 더 끌어 올려서 상위에서 해당 모듈에 접근하도록 만드는 방법도 있다. 어쨌든 중요한 것은 모듈을 정할 때 R&R(Role and Responsibility)를 명확히 하라는 것이다. 모듈의 응집성(Cohesion: 관련 있는 것들로만 묶이도록 만든 정도)을 높이기 위해서, 해당 모듈의 역할과 책임 범위를 명확히 하라는 것이다. 너무 많은 기능을 수행하는 모듈이라면, 여러 개의 모듈로 쪼개서 단순한 정해진 역할을 하는 모듈 여러 개로 나누는 것이 좋다. 만약 공통된 기능을 여러 모듈에서 똑같이 수행한다면, 그런 기능들을 모아서 새로운 모듈로 만드는 것이 옳다.

잘 정의된 모듈은 재사용성이 높다는 것은 어쩌면 당연한 말일 것이다. 정의된 인터페이스만 사용되도록 만들어 졌으며, 다른 모듈 간의 의존성도 최소화시켜서 제작되었다면, 이용하는 측면에서 고려할 부분이 적기 때문이다. 흔히 성능상 어쩔 수 없이 이렇게 만들었다는 것으로 보통 자신들이 개발한 코드에 대해서 변명을 하는 경우가 많다. 성능이 아니면 실제 상품화 때문에 이렇게 될 수밖에 없었다고도 이야기한다. 하지만, 중요한 것은 항상 기본에 있으며, 기본을 튼튼히 하는 활동을 게을리했다는 것은 앞으로 늘어날 비용에 대해서 책임을 줄이는 것이 아님을 알아두기 바란다.

**08. “Copy&Paste”를 남발하지 말라.**

코드를 작성하다 보면 흔히들 역할이 비슷한 코드를 “copy&paste”한다. 저자의 경우에도 이런 경험이 있다. 즉, 비슷 비슷한데 조금씩 달라지는 부분 때문에 이런 일들이 많이 발생한다. 하지만, 분명히 말하건대 이렇게 작성된 코드는 항상 세심한 주의를 필요한다. 즉, “copy”하고, “past”e하고 수정하는 동안, 잘못 수정된 부분이 존재할 가능성이 크기 때문이다. 이미 알겠지만 이럴 때의 가장 쉬운 대처 방법은 비슷한 부분을 모아서 함수화 시키고, 달라지는 부분을 파라미터(Parameter)로 전달하는 것이다. 물론, 이것도 그렇게 쉽지 않은 경우가 있다. 이럴 때는 그 부분에 대해서 더 작은 단위로 쪼개서 공통된 부분을 찾아내는 것도 가능할 것이다.

중요한 것은 “copy&paste”해서 사용하는 코드들을 잘 관리하라는 것이다. 오류는 코딩의 무의식적인 부주의에서 발생하므로, 이런 부분이 생길 때는 더욱 주의를 기울이는 것도 버그를 줄이는 한 가지 방법이 될 수 있다. 인터넷에서 가져온 코드를 “copy&paste”할 때는 가져온 코드의 라이센스(License)에 주의해야 한다. 오픈소스(Open Source)라고 해서 라이센스가 없다고 생각해서는 안된다. 남이 만든 코드는 남의 지적인 산물이다. 그렇다고 모든 코드를 내가 만들어야 한다는 것은 아니지만, 최소한 남의 코드에 대한 존경심(혹은, 존중하는 마음)은 발휘해야 할 것이다. 그것이 상업적인 보답이든, 아니면 그냥 단순한 문구의 삽입(Insert)이든 간에 라이센스에 대해서 항상 기억하기 바란다.

**09. 나의 버그를 남의 것이라고 말하지 말라.**

프로젝트는 다양한 사람이 다양하게 나뉘어져 진행되는 경우가 많다. 구조화가 잘못되어 남이 한 부분과 내가 한 부분이 서로 얽혀 있을 수도 있고, 혹은 잘 나누어져 있다고 하더라도 한 부분에서 문제가 안 되는 것이 다른 부분에서는 문제가 발생할 수 있다. 버그는 의도와는 상관없이 일어나는 것이므로, 자신이 만든 코드든 남이 만든 코드든 시스템의 버그라는 측면에서 비난할 문제가 아니라 해결할 꺼리라는 것이다. 더 중요한 것은 그것이 제품 출시 전에는 반드시 수정되어야 할 사항이라는 것이다.

버그는 남의 것이라고 우겨서 처리될 일이 아니다. 프로젝트는 팀워크가 생명이며, 잘못을 남을 탓하는 분위기로 흘러가서는 일정지연이나 실패를 경험할 수밖에 없다. 오히려 남의 버그라도 자신이 해결책을 알고 있다고 하면 직접 수정해도 무방하다. 어차피 수정에 대한 것은 전부 소스 관리 시스템에 누가 했는지까지 로그로 남게 되며, 되돌리기도 어렵지 않기 때문이다. 남을 탓하기보다, 혹시 자기 자신의 실수는 없었는지 부터 먼저 확인해 보기 바란다. 코드는 내 것 남의 것이라는 구분이 있는 것이 아니라, 과제에 관련된 모두의 것이기 때문이다.

**10. 내 이웃의 개발자를 탐하지 말라.**

사실 이 부분은 조금 원조 십계명을 이용한 것이다. 하지만, 이런 상황은 실제로 종종 발생한다. 예를 들어, 어떤 개발자를 잠시 이웃한 팀의 협조를 얻어서 부족한 부분에 대한 코딩을 맡겼는데, 상당히 실력이 좋은 개발자인 경우 개발 과제가 끝난 후에도 같이 일하고 싶어서 높은 분들께 요청하는 경우가 있다. 하지만, 한 가지 명심할 것은 그러한 개발자를 왜 자신의 개발팀에서는 키워내지 못했는지를 먼저 고민해 보는 것이 좋다. 상품화를 담당한 팀은 항상 바빠서 거의 교육에 갈 기회가 적다. 교육을 가더라도 사내 교육일 경우에는 교육장에서 전화를 붙들고 살아야 하고, 교육 후에는 다시 자신의 자리에 돌아가서 밀린 업무를 처리해야 한다. 참 피곤한 교육이 되고마는 것이다. 이럴 바에는 다음부터 교육을 안 가고 만다고 생각하는 경우도 있다.

사람은 무한정 결과를 만들어내는 존재가 아니다. 뭔가 입력(Input)을 얻어야 출력(Output)을 생산해 낸다. 새로 배우는 것이 없는데 새로운 것을 만들라고 한다면, 그게 가능이나 한 일이 겠는가? 기술은 6개월이 멀다하고 새로운 것이 나온다. 아니 3개월도 길다. 새로운 제품은 수도 없이 지속해서 만들어져 나온다. 시장을 이끄는(Leading) 존재가 아니라면, 그러한 제품의 대응 제품만 만들다가 좋은 세월 다 보낸다. 내 이웃에 있는 멋진 개발자를 넘겨다보기보다는 자신 밑에 있는 개발자를 키워서 좋은 인재로 만들수 있도록 해야한다.

이상으로 10가지의 덕목(?)과도 같은 이야기를 풀어 보았다. 물론 이러한 것들에 대해서 동의하지 못하는 부분도 많을 것이다. 하지만, 최소한 한 번쯤 그런 생각을 가지고 자신을 뒤돌아보는 것도 필요하다고 생각한다.

# [ Model-View-ViewModel을 배워볼까? ]

최근에서야 C#이라는 언어를 알게 되었다. 물론, 개인적으로 새로운 것을 공부한다는 차원에서 이것 저것을 뒤져보다가 재미있을 것 같아서 한번 책을 구입해서 읽었다. WPF(Windows Presentation Foundation)도 C#으로 뭔가를 하기 위해서는 한 번쯤 봐야겠다는 생각에서 코딩과 함께 실제로 구현해보기도 했다. 아주 오래전에 MFC나 Win32로 윈도우즈 프로그램을 짜던 시절에 잠시 본 이후에는 거의 손을 놓고 있었는데, 이렇게까지 발전하고 있다는 사실이 새삼 대단하게 느껴진다.

먼저, WinForm과 같은 것은 잘 모르니 제외하고, WPF를 이용해서 UI를 구현할 수 있다는 것이 참으로 대단하게 느껴졌다. 디자인(Design)과 코드의 분리 하나만 보더라도 개발자의 근심을 덜어준 것에 대해서 정말 감사하게 생각한다. 물론, Microsoft사에서 만든 것이지만, 어쩌면 이러한 것들이 진보라고 불리기에 조금도 부족함이 없다고 생각한다. 예전에 잠시 임베디드 시스템(Embedded System)에서 소프트웨어를 개발할 당시 스크립트(Script)를 이용해서 프로그램의 프로토타입(Prototype)을 자동으로 생성하면 어떨까 하는 생각도 했었다. 복잡하지 않은(?) 임베디드 시스템에서는 기기(Device)는 유한상태 기계(Finite State Machine)로 표현될 수 있고, 각각의 상태를 전이(Transition)하는데 필요한 이벤트와 상태전이시에 해야 할 동작(action)을 스크립트를 이용해서 구성할 수 있다고 봤었다.

그렇게 생각하면, 마치 (S, S', e, a)와 같이 Event e를 받으면, S라는 상태에서 S’라는 상태로 전이하고 Action a를 해주면 된다와 같은 간단한 표현식을 얻을 수 있게 되는데, 이와 같은 스크립트로 모든 기기의 상태 전이를 다 기술하기만 한다면, 실제로 스크립트를 분석(Parsing)해서 원하는 응용 프로그램에 대한 기본 골격(Framework)을 만들 수도 있을 것으로 생각했었다. 물론, 실제로 구현까지는 가지 않았지만, 어쨌든 개발자들이 실수할 수 있는 상당 부분을 자동으로 처리할 수 있을 것이라고 믿었다. 만약, 이미 존재하는 프레임워크를 이용할 수 있다면, 개발에 드는 시간과 비용을 상당 부분 줄일 수 있을 거라는 가정했었다.

개발에서 중요한 것은 정작 구현하고자 하는 내용(contents)이 주가 되어야 하는데 그렇지 못한 경우가 흔하다. 예를 들어, 빈번한 UI 변경에 대한 요구를 수없이 고객에게서 들어야 하고, 그럴 때마다 코딩 및 디버깅 노력도 상당히 필요한 것이 사실이다. XML과 같은 언어로 UI 구현을 대체하고, 거기에 한 수 더 떠 그래픽 도구를 이용한 시각 편집 기능을 더하니, UI 변경에 대한 요구는 디자인 문제로 변화되어 버린다.

결론적으로 이야기하면, 바로 WPF와 같은 것을 잘 이용하기 위해서는 상당한 정도의 지식이 사실 필요하다. 기존의 코딩에서 사용하던 모델에 대한 것과 MVVM과 같은 새로운 모델, XML 스크립트에 대한 기초적인 이해, 관리되는 코드(Managed code)와 네이티브(Native code)간의 협력(Interaction)에 대한 방안, 기존 코드의 이전(Migration), 디자인 패턴(Design pattern)등등 상당한 지식을 요구한다. 어쨌든, 나름 처음 시작하는 개발자라면 새로운 것을 해볼 것을 권하고 싶다. 오랜 시간의 노력이 쌓여서 만들어진 기술들을 잘 익힌다면, 다음 세대로의 좋은 통로 역할은 할 수 있을 거라고 본다. 결국, 오늘도 미래의 과거일 뿐이며, 우리가 한 일을 기반으로 “더 좋은 것은 항상 미래에 구현될 것”이기 때문이다.

# [ 역시 살길은 서비스 ]

경제학에서 이야기하는 서비스 산업은 실질적인 성장을 추구하는 것이 아니라고 한다. 잘은 모르지만 생산의 결여로 인해 일자리를 창출하는 능력이 모자란다는 표현인 것 같다. 일자리를 창출하지 못하니, 그 만큼 실질적으로 사람들이 돈을 벌 기회는 적어지고, 소비만 하는 사회로 변화되어 간다는 것 같다(하지만, 소비할 수 있는 돈은 어디서 나올까? 아마도 나라는 지속적으로 빚이 늘어날 것으로 보이지만).

하지만, 예외적으로 소프트웨어의 경우는 “좋은 품질의 서비스를 구축하는 것”이 핵심이라고 볼 수 있다. 소프트웨어의 존재 이유는 사람을 편하게 하기 위한 것이므로. 좋은 품질의 서비스를 구축하기 위해서 많은 부분에서 노력해야 한다. 많은 부분이라고 하지만, 구체적으로 어떤 부분들을 이야기하는 것일까? 크게 3가지가 요즘에는 이슈가 되고 있는데, 각각은 클라이언트 입장에서 사용자 인터페이스(User Interface), 서버(Server), 인프라(Infra)로 나누어 볼 수 있겠다. 클라이언트는 가격이나 성능 면에서 가장 열악한 환경에서 운영된다. 서버는 거의 최상 수준에서, 그리고, 인프라는 그 중간 단계 수준에서 사용된다고 볼 수 있다. 하지만, 좋은 품질의 서비스를 제공하기 위해서는 이 세 가지 구성요소가 적절히 구성되어야 하며, 어느 하나라도 부족하면 변덕이 심한 사용자들은 쉽게 다른 곳으로 가버리고 만다.

소프트웨어적으로 가장 중요한 부분은 역시 개발자 측면을 고려할 때 서버에 들어가는 많은 기술이다. 각종 오류 상황에서도 적절한 동작을 할 수 있는 기술과 수많은 공격에도 꿋꿋하게 운영될 수 있는 보안기술, 사용자의 수많은 데이터를 저장하고 이를 검색해서 원하는 결과를 도출할 수 있는 DB(DataBase) 등등 많은 것들이 필요하다. 클라이언트는 다양한 플랫폼 상에서 동작할 수 있어야 한다. iOS나 Android도 될 수 있으며, 웹(Web)을 기반한 애플릿(Applet)에서도 가능해야 한다. 인프라의 경우는 사용자별 차이를 두는 서비스 정책과 사용자의 위치에 무관하게 안정된 서비스를 제공해 줄 수 있는 기술이 필요하다. (물론, 훨씬 더 많은 기술이 필요하겠지만) 소프트웨어는 서비스의 질(Quality)를 높이기 위해서 기계에만 전적으로 의존할 수 없으며, 반드시 인간의 지적인 사고 과정(사람에 대한 이해와 시스템의 설계 등)이 있어야 한다는 점에서 일자리는 끝없이 늘어날 수밖에 없다.

인간의 모든 요구가 만족되지 않는다면, 사람에게 더 좋은 편의를 제공해주기 위해서 다양한 서비스가 탄생할 것이다. 잠시 화제를 약간 바꿔서, 애플에서 공개(2011년 6월 7일)한 iCloud를 보도록 하자. 왜 이것을 보냐고 묻는다면 Cloud를 이용해서 사용자가 직접 경험해 볼 수 있는 것에 대해서 애플에서 내놓은 첫 번째 대답이기 때문이다. 애플에서 생각하는 서비스는 바로 컨텐츠(Contents)에 대한 무한 재활용(중복의 제거와 복제 개념을 함께 가지는)이라는 관점에서 상당히 효과적인 Cloud 서비스의 예이다. 이와 비슷하게 삼성전자에서도 오피스(Office) 개념을 추가한 Cloud 서비스를 발표할 것이라는 내용이 있기도 하다. 즉, 이제는 구체적인 서비스를 어떻게 효과적으로 제공할 것인가로 사용자들에게 접근하고 있다.

단순히 단말기는 파는 일이 아니라, 서비스를 팔아서 이익을 창출하는 비지니스 모델들이 하나씩 등장하고 있다. 기기의 가격은 사실상 레드오션(Red Ocean)이 되면 무의미해진다. 즉, 가격의 하락이 아주 극심한 것이 사실이다. 저가 시장에서 Nokia가 중국에서 화웨이나 ZTE등에 밀려나듯이 가격은 더는 경쟁력이 될 수 없다(2017년 중국 시장에서는 삼성도 밀리고 있는 것이 사실). 단말기는 사실 거의 공짜로 주어도 되는 시대는 이미 오래전에 왔다. 몇 십만원짜리 단말기를 1원 가격에 제공하고, 대신에 Infra나 서비스를 제공해서 얻는 돈으로 단말기 업체와 서비스 제공자(Service Provider)들이 먹고 산 지는 오래된 이야기다. 앱(App)을 파는 시기도 이제는 지나가고 있다. 공짜로 제공되는 앱들이 광고를 등에 지고 등장하기 때문이다. 이제는 진정 어떻게 먹고 살지를 고민해야 할 때가 된 것이다.

다시 본래 하던 이야기로 돌아와서, 소프트웨어 개발자로서 그럼 어떻게 향후 먹고 살 것인가를 고민할 차례다. 다른 말로 표현하면, “소프트웨어 개발자로서 어떻게 하면 향후 지속해서 생산성 높은 일을 할 수 있을 것인가”를 이야기해야 한다. 결론은 역시 우리는 서버 기술을 개발해야 한다는 것이다. 서버에 서비스를 제공할 수 있는 소프트웨어를 개발하고, 이를 유지 보수 및 안정적으로 동작할 수 있는 기술을 지속해서 개발 해야지만 앞으로도 비지니스를 영위하는데 기반을 갖출 수 있다는 것이다. 특히 대규모의 서비스를 제공할 수 있는 솔루션(Solution)을 찾지 못한다면, 그러한 서비스를 제공하는 업체들에 지속해서 사업의 기반을 의지할 수밖에 없을 것이다.

하지만, Cloud를 제공할 정도의 기반이 있는 업체는 정말 큰 기업이다. 대규모의 서버를 구축해야 하고, 그러한 서버들을 유지 보수하는데 막대한 자금이 든다. 그러한 자금은 초기 투자는 오로지 기업 내부에서 해결해야 할 것이고, 나중에 사용자에게서 받아내는 수밖에 없다. 사용자는 변덕이 심하기 때문에 자주 옮겨 다닐 것이고, 지속적인 자금을 공급받기 위해서는 서비스의 종류뿐만이 아니라 안정적인 품질도 중요해질 수밖에 없다. 인력도 마찬가지다. 빨리 새로운 서비스를 제공해 주기 위해서는 다양한 새로운 기술을 가진 사람들이 필요하며, 지속해서 투입될 수밖에 없을 것이다. 한 가지 다행스러운 점은 고객은 한번 서비스에 익숙해지면 잘 옮기지 않는다는 점이며, 어느 정도의 만족감이 들 때는 다른 서비스로 눈을 돌리지 않는다는 점이다. 선점한 상황에서는 그러한 효과는 상당 시간 지속될 수 있을 것이다.

이제 정말 대규모의 네트워크가 연결된 세상이 서서히 열리고 있다. 기술의 발전은 인간의 편의를 가지고 왔지만, 특정 기술에 대한 종속도 만만찮게 강해지고 있다. 종속될 것인가 종속시킬 것인가를 이야기하기 보다는 사람이 중심이 된 더 좋은 서비스를 제공하고자 노력하는 것이 우리 소프트웨어 개발자의 몫이다. 눈앞에 보이는 화려한 것 보다는 사람의 깊숙이 숨겨진 욕구를 만족시켜주는 다양한 품질 좋은 서비스를 만들어야 하는 것이, 우리 소프트웨어 개발자의 진정한 해야 할 일인 것이다.

# [ 학생들이 알아야 할 열가지 ]

이번에는 Software를 개발하는 분야에서 일하고 싶은 학생들이 알아야 할 열 가지를 이야기해 보겠다. 물론, 여기서 이야기하는 내용은 전적으로 본인의 생각과 경험에서 나온 것일 뿐이다. 하지만, 경험이 체계화되면 지식이 되듯 그냥 흘려서 듣기보다는 한 번쯤 고민해 볼 가치는 있다고 생각한다.

**00. 코딩**

소프트웨어 개발과 관련해서 일하고자 하는 학생들은 기본적으로 코딩을 할 수 있어야 한다. 어떤 언어(Language)를 사용하는가는 시대의 흐름과 개인적인 성향에 따라 차이가 있겠지만, 최소한 자신이 목적한 바를 실제 프로그램으로 구현할 수 있는 능력은 있어야 한다. 중요한 것은 우리가 일하는 분야에서는 코딩이 가장 기본적인 스킬(Skill)이라는 것이다. 간혹 코딩은 언제든 배울 수 있다고 생각해서 다른 곳에 신경을 더 많이 썼다고 이야기하는 학생들을 만나는데, 이건 정말 잘못된 생각이다. 코딩이 바탕이 되지 않는다면 배운 지식은 무의미하기 때문이다. 마치 실제가 없는 지식은 모래성같이 파도가 밀려오면 그냥 허물어져 버리기 때문이다. 어떤 언어를 잘 써야 할 것인가를 두고 질문을 하는 사람들이 있다. 실제로는 한가지 언어에 능통한 사람은 다른 언어도 빨리 배운다. 따라서, 어떤 언어라도 정말 잘 사용할 수 있는 수준까지 도달했다면 이미 기본적인 코딩 능력은 있다고 볼 수 있다.

하지만, 잘 짠 코드와 그렇지 않은 코드가 있듯이 코딩은 쉽게 배울 수 있는 것이 아니다. 어쩌면 가장 오랫동안 배워야 할 것인지도 모른다. 잘 짠 것과 그렇지 못한 것을 나누는 기준은 생각보다 간단하다. 즉, "남이 보기에 쉽게 짠 코드"가 잘 짠 것이다. 남이 짠 코드를 읽었을 때의 거부감이 적을수록 그 코드는 잘 짰다는 봐야 할 것이다. 인터넷 상에서는 무수히 많은 남이 짜놓은 코드를 볼 수 있을 것이다. 자기가 보기에 쉽다고 생각되는 코드들의 공통점을 찾고, 열심히 따라서 연습하는 것 이외에는 사실상 코딩 능력을 키울 방법은 없다. 직접 해보지 않고서는 알 수 없는 것이 이론과 실제의 차이다(그렇다고, 이론이 중요하지 않다는 것은 절대 아니다.).

**01. 읽기**

쓰기 위해서는 읽어야 한다. “입력(Input)이 없이는 출력(Output)이 없다”는 것은 진리다. 물론 쓰레기 입력은 쓰레기 출력을 만들 확률이 높다(간혹 쓰레기 값이 입력으로 주어지더라도 정말 멋진 출력을 만들어내는 경우도 있지만). 읽기도 그냥 읽는 것이 아니라 바르게 읽는 것이 중요하다. 또한, 좋은 것을 읽는 것이 무엇보다 중요하다. 바르게 읽는다는 것은 무슨 뜻인가? 바르다는 것은 글을 읽을 때 정확히 저자가 의도하는 이야기를 잘 이해하는 것을 말한다. 정작 중요한 내용은 다 놓치고, 중요하지 않은 내용만 머리 속에 담아두는 경우가 많은데 이는 시간 낭비일 뿐이다. 읽기를 하되 내용을 잘 이해하는 것이 중요하다.

실제로 어떤 책의 글귀를 잘 못 이해하게 되면 큰 낭패를 보는 경우가 종종 있다. 이는 번역의 오류이든, 혹은 읽는 사람의 지식 수준이 낮아서 그렇든 다양한 해석을 불러올 수 있는 글들은 널려있다. 정말 하고자 하는 이야기는 사실 한번 읽어서는 잘 이해가 되지 않는 경우도 많다. 지적 수준이 높아질수록 기본적인 것들에 대한 개념이 중요함을 많이 깨닫는데, 이럴 때 일수록 느끼는 것이 "그때는 왜 그렇게 못했지"라는 후회다. 이런 후회를 조금이라도 줄이고자 한다면 내용을 충분히 이해할 수 있는 만큼 읽어보는 것이다. 좋은 것을 읽는 것이 중요하다고 했는데, 그럼 어떤 것이 좋은 것일까? 시중에는 다양한 책들이 있지만 검증을 통해서 반드시 읽어야 할 도서 정보는 이미 다 정리되어 있다. 특정 분야의 책에서 “고전이라고 불리는 책”들은 어느 분야이든 있기 마련이다.

최신 기술을 다루는 책들도 많지만, 고전이라고 여겨지는 그 분야의 대표적인 책들은 절대 사서 봐야 할 것들이다. 될 수 있으면 원서로 볼 것을 권장한다. 혹자는 원서는 보는데 시간이 너무 많이 걸려서 보기 힘들다는 사람들도 있지만, 문제는 우리가 다루는 분야가 소프트웨어라는 것이다. 사실상의 모든 용어가 영어다. 번역서로 보면 어색한 부분이 많이 있을 수밖에 없는 분야다. 다른 사람들과 이야기를 나눌 때 번역된 용어를 사용하면 오해가 생길 수 있다는 것이다. 영어가 안되는 사람은 최소한 전문용어는 반드시 영어로 기억하기 바란다. 훈민정음이 만들어질 때는 컴퓨터가 없었으니, 세종대왕님께서도 그 정도는 충분히 이해해 주실 것이다.

**02. 쓰기**

쓰는 것은 가장 기본적인 의사소통 수단이다. 물론 코딩도 의사소통 수단이기는 하다. 예전에 이런 이야기를 들은 적이 있다. "문서는 없지만 모두 코드에 녹아 있으니 열심히 공부하세요". 솔직히 처음 이 말을 처음 들었을 때, 머리 통을 후려 갈겨주고 싶었지만 참았었다. 코드에 있는 것은 분석과 이해를 위해서 필연적으로 상당한 시간이 필요하다. 처음 프로젝트에 투입된 사람이 가장 힘들어하는 부분이 바로 읽어야 할 문서는 없고, 소스 코드만 잔뜩 있는 상황이다. 아마도 그런 경험이 없는 개발자는 거의 없을 것이다.

쓰기도 많이 써야지 잘 쓴다. 모든 것이 그렇듯 연습을 많이 한 사람이 당연히 잘하기 마련이다. 문제는 쓰기를 두려워하는 태도에 달린 것이지, 시간이 부족한 것이 아니라는 점이다. 개발자가 항상 핑계로 이야기하는 것이 바로 글 쓸 시간이 없다는 말이다. 가슴에 손을 얹고 한번 진지하게 고민해보라. 정말 시간이 부족해서 못 쓴 것인지 아니면 글쓰기가 두려워서 못 쓴 것인지. 개발자는 글쓰기를 두려워해서는 안 된다. 글은 매일 써야 하는 일종의 일기와 같은 것이다. 물론 기술에 대한 글은 좀 더 체계적으로 익혀야 하겠지만, 그런 것을 다 제외하고 일단은 많이 쓰기부터 해야 한다. 그래야지만 두려움이 사라지게 되는 것이다.

대부분 개발자는 과제에 대한 설계서와 같은 것을 프로젝트를 마무리할 때 쓰려는 성향이 있다. 이것은 이미 잘 알겠지만 일의 순서가 바뀐 것이다. 설계서는 일을 시작하기 전에 작성하는 것이고, 프로젝트의 마무리에서는 그렇게 정리된 문서들을 추려서 보고서를 만드는 것이다. 설계서(결정의 내용과 이유를 정리한 문서)는 변경될 수 있다. 그리고, 일을 해 나가면서 생겼던 문제점이나 해결 방안들은 지속해서 문서화되어야 한다. 따라서, 개발자가 쓰는 글은 항상 일과 함께 진행되어야 한다. 지속해서 수정해야 하며 여러 사람과 공유되어야 한다. 오랜 시간 동안 인간이 인간다울 수 있었던 것은 말이 아닌 문서로서 만들어진 것들이 있었기 때문이다. 물론, 당사자의 말보다 더 값진 것은 없을지라도, 문서는 그 문서를 만든 사람보다 항상 더 오래 살아남는다. 시간과 공간을 달리하더라도 대화를 나눌 수 있는 유일한 방법이 바로 문서다.

**03. 발표**

자신이 한 일을 돋보이게 하는 것은 발표를 통해서 이루어 진다. 공식적으로 어떤 일의 마일스톤(Milestone)이 되는 것도 이러한 발표를 통해서 이루어 진다. 발표는 일종의 자기 PR이다. 동일한 일을 했더라도 누가 더 발표를 잘하는가에 따라 일을 잘했는지 못 했는지가 판가름 나기도 한다. 설령 누군가가 더 일을 잘한다고 개발자들 사이에 인식이 되어 있더라도, 높으신 분들이 보기에는 발표를 효과적으로 멋지게 하는 사람에게 후한 점수를 주기 마련이다.

발표는 대상을 잘 파악하는 데서부터 시작한다. 대상이라 함은 참석하는 사람 중에 가장 높은 직위를 가진 사람이 누군가에게 영향을 많이 받는다. 그 사람이 이해하는 수준으로 발표를 해야 하는 것이다. 발표 스킬(Presentation Skill)이라고 해서 다양한 방법으로 어떻게 발표하는지를 가르칠 정도이니, 발표가 얼마나 중요한지는 새롭게 강조를 안 해도 충분히 잘 이해할 것이다. 여기서 중요한 점은 발표하는 사람들과 발표를 듣는 사람이 경쟁 관계가 아니라 협동 관계라는 사실을 파악하는 것이다. 관심이 있어서 발표에 참석하고, 무언가 얻어갈 것이나 동의를 구할 것이 있어서 발표한다는 점이다. 이러한 협동 관계에서는 일방적인 전달은 쓸데없이 힘만 소비하는 효과적인 방법이 아니다. 차라리 편안한 자세로 이야기하는 것이 가장 좋다. 하지만, 실제로 이러한 발표 자리를 즐길 수 있는 사람은 그렇게 많지 않으며 항상 긴장하기 마련이다.

“자신감”이 자신이 가진 가장 큰 무기라면, 남의 공격에 견딜 수 있는 “솔직함”은 가장 큰 방어 수단이라고 할 수 있다. 어떻게 남을 설득할 것인지를 생각하지 말고, 어떻게 남의 이야기를 더 잘 들을 것인가에 신경써야 한다. 발표 시에는 많은 말들이 오갈 수 있다. 어떤 사람은 공격적이기도 하고, 어떤 사람은 나를 대변해서 요약해 주기도 한다. 중요한 것은 그들 모두가 함께 한다는 생각이다. 소중한 자신의 시간을 내서 나의 말을 듣기 위해서 참석했다는 것이다. 따라서, 최소한 발표를 위해서는 자신이 발표하는 내용에 대해서만큼은 충분히 숙지하고 있어야 한다.

모르는 용어를 주저리주저리 이야기한다고 해서 좋은 발표는 아니다. 전문적인 용어가 난무한다고 해서 유식해 보이지도 않으며, 제스처를 멋지게 취한다고 누가 촬영해서 동영상을 유투브같은데 올려놓지도 않는다(물론, 아주 근사한 자리에서 멋지게 해야 할 필요가 있는 발표는 제외하고). 학술 발표에서 제품 설명회와 같은 발표를 해서도 안되며, 제품 설명회 자리에서 이론적인 공식들을 토론해서도 안 될 것이다. 듣는 사람들이 어떤 것을 알기를 원하는지를 제목으로 정했으면 그것에 충실한 발표를 하기 바란다(간혹, 제목과는 전혀 상관없는 발표를 보게 되는 경우도 많다). 자료가 발표를 하는 것이 아니라, 발표자가 한다는 사실을 잊지 말아야 할 것이다.

**04. 대화**

소프트웨어는 혼자서 개발하는 것이 아니다. 항상 끊임없이 누군가와 대화를 하고, 이를 통해서 원하는 목적을 달성하게 만드는 것이 개발자의 할 일이다. 누군가 개발에서 가장 중요한 것이 무엇이냐고 묻는다면, "대화(Communication)"라고 이야기할 것이다. 그 만큼 프로젝트를 성공으로 이끄는 가장 큰 부분이 대화이다. 고객과의 대화, 팀장과 팀원의 대화, 마케팅과 개발자 간의 대화, 사장과 직원의 대화 등등 대화는 개발의 전 과정에서 일어나는 일상적인 이야기다.

속담에 "말 한마디로 천 냥 빚을 갚는다"고 한다. 실제로 가장 힘들었을 때 솔직한 말 한마디는 대부분 갈등을 해소할 수 있는 지름길이다. 대화에도 역시 기술이 필요하다. 다양한 기술들이 있어서 여기서 전부 말하기는 힘들지만, 단순히 대화의 기술에서 그친다면 말을 잘하는 사람이 되지만, 대화에 진정성이 들어가면 그 사람은 존경의 대상이나 혹은 친구가 될 수 있다. 혼자서 골방에 틀어박혀 열심히 프로그램을 짠다고 해서 개발이 끝나는 것이 아니다. 특히, PL(Project Leader)이 되고자 한다면, 다양한 사람들과 대화할 필요가 있다. 그렇다고 아주 대단한 웅변가가 되라는 말은 아니다. 어떤 목적만을 가지고 이야기하기 보다는 지나가는 말 한마디라도 그 사람에게 항상 “관심”이 있다는 표시를 하는 것 만으로도 충분하다. 그러한 관심만큼 대화의 상대방에게 만족을 주는 것은 없다.

**05. 도구**

사람은 도구를 사용할 수 있는 동물이다(물론, 동물 중에도 도구를 사용하기도 하지만). 소프트웨어 개발에도 다양한 도구가 사용된다. 자신이 사용하는 도구를 얼마나 잘 사용하느냐에 따라 생산성은 극적으로 달라질 수 있다(그렇다고, 한 가지 도구가 생산상을 극적으로 향상시켜주는 것은 없다고 이미 많은 사람들이 이야기했다. 관련해서 “No Silver Bullet” 찾아보면 된다.).

소프트웨어 개발에서 필요한 도구는 편집기(Editor), 통합 개발 환경(IDE: Eclipse나 Visual Studio와 같은), Source code navigator, SVN(Subversion), Trac(Issue관리), Mantis(Issue관리) 등등 많은 것들이 있다. 이러한 다양한 도구를 메뉴얼을 보지 않고도 세세한 기능을 잘 활용할 수 있다면, 그 자체로도 훌륭한 경쟁력이 될 수 있다(단축기를 사용하는 것도 포함해서). 설치 및 운영까지도 할 수 있다면 더 좋겠지만, 그렇게 하지 못할 정도로 비싼 도구들도 많기에 일단은 잘 사용하는 데 초점을 맞추는 것이 좋다.

잘 쓸려면 어떻게 해야 할까? 쉽게 보면 일단 자기가 하는 모든 숙제나 프로젝트에 적용해보는 것이다. 새로운 것에 대한 거부감은 누구나 가지고 있다. 마치 새 신발을 샀을 때 생기는 발의 굳은살처럼 처음에는 사용하는 것이 낯설기만 할 것이다. 하지만, 이것도 적응하기 마련이다. 만약, 자신의 목적에 정확히 들어맞는 도구가 없다면 어떻게 할 것인가? 장기적인 관점에서 볼 때는 새롭게 만드는 것도 한 가지 방법일 것이다. 단기적으로는 일단 여러 가지 관련된 이미 개발된 도구들을 가지고 조합해서 원하는 결과를 만들어 내는 것이 좋을 것이다. 사람은 도구를 써서 손 발이 해야 할 일을 최소화하고, 자신이 목적한 것에 더 집중할 수 있다. 따라서, 과감히 다양한 것들을 배우도록 노력해야 한다. 최소한 자신 있게 사용할 수 있는 도구들을 옆에 항상 두고 익숙하게 사용할 수 있어야 진정한 개발자라고 할만하다.

간혹, 자신이 사용하던 도구만 고집하는 개발자들을 만날 수 있는데, 심지어 새로운 도구에 대해서 거부감을 가지고 적극적으로 반대를 하는 경우도 있다. 마치, 어떤 제품의 약점만 들추고, 정말 중요한 것은 간단히 무시하기도 한다. 참된 개발자면 자신이 사용하는 연장 보다, 자신이 해야할 일이 무엇인지에 대 집중할 것이다. 도구는 그 일을 하는데 필요한 만큼 잘 활용할 수 있으면 된다. 그것이 반드시 100점은 아닐지라도 충분히 사용할 만한 가치가 있다면, 자신의 연장 통에 넣어두지 않을 이유는 없는 것이다. “실력없는 사람이 연장만 탓할 뿐”이다.

**06. 그림**

여기서 말하는 그림이란 소프트웨어의 구조를 보여주는 모델(Model)을 그리는 것을 말한다. 무턱대고 아무런 그림을 그릴 수 있는 것은 아니다. 그림을 그리기 위해서는 머리 속에 있는 생각을 끊임없이 현실 세상에 투영할 수 있도록 사고를 해야 한다. 교과서에서 보는 내용을 계통도를 그려보는 연습도 좋다. 전체적으로 사물을 이해하는 능력은 언제나 중요하다. 전체적인 이해가 없이 좋은 시스템을 만드는 것은 어려우며, 항상 개발자로서 뭔가 부족하다고 깨닫는 이유도 이러한 전체적인 시각의 부재에서 비롯된다.

그림을 그리기 위해서는 추상적인 것을 형상화시키는 능력을 갖춰야 한다. 어떤 사물이나 개념을 형상화시켜서 모델을 만들고, 그 모델의 장단점을 파악하는 것은 구현(Implementation)을 위해서 반드시 필요한 과정이다. 모델링(Modeling)을 하지 않고 구현부터 시작하는 경우에는, 나중에 지급해야(수정해야) 할 비용만 커지기 때문이다. 소프트웨어를 건물 짓기에 많이 비교하는데, 건물과 마찬가지로 한번 지어지고 나면 고치는 데는 상당한 비용이 든다는 점도 기억해야 할 것이다.

모델링에 필요한 툴들은 무엇이 있을까? 아무래도 가장 많이 사용하는 UML(Unified Modeling Language)을 사용하는 것이 일반적일 것이다. 하지만, 표현방법에 너무 집착하지는 말자. 표현방법에 치중하다 보면 정작 중요한 내용은 간과하기 쉽기 때문이다. 그림은 분석을 통해서 만들어진다. 기존에 혹은 새로 주어진 문제에 대한 분석 없이는 더 좋은 것을 만들어낼 수 없다. 프로젝트는 항상 기존에 주어진 문제에 대한 답을 찾거나, 누군가의 요구를 만족시켜야 하기에, 처음에 그리는 그림은 전체적인 숲을 대상으로 하는 것이 좋다. 지속적인 개선을 통해서 점차 나무의 형태를 갖춰가고, 결국에는 줄기와 나뭇잎 하나 하나까지도 세밀하게 신경써서 그리도록 하는 것이 설계다. 어느 정도의 시간이 흘러서 숲과 나무가 갖춰지면, 이제는 그것을 실제로 가꾸는 것이 남은 할 일이다.

자신이 하는 일을 추상화시켜서 개념적으로 표현하는 것은 어느 분야에서나 필요하며 개발자가 지녀야 할 기본 자질이다. 추상적인 개념을 정의하지 못하면 일을 나누는 것이 불가능하며, 각각의 구성 요소에 대한 역할과 책임이 불명확하게 된다. 따라서, 중복과 조화롭지 못한 연결이 늘어나게 되며, 결국 시스템은 시간이 지날수록 복잡하게 변해 간다. 추상화란 사물의 특징만을 잡아 역할과 책임을 나누는 기본적인 스킬일 뿐이다. 추상화된 역할과 책임을 모델이라는 도구를 이용해서 그림으로 표현하는 것은, 관련된 사람들이 구현해야 할 시스템에 대한 “동일한 이해”를 가지도록 돕는다.

**07. 실무**

기업이 사람을 채용할 때 가장 중요하게 보는 것이 바로 “어떤 일을 해 봤는가?”이다. 그 사람의 능력을 짧은 인터뷰 시간에 판단하기는 힘들다. 하지만, 그 사람이 무슨 일을 해봤으며, 그것을 통해서 어떤 일을 할 수 있을 것인가를 판단하는 것은 그렇게 어렵지 않다. 전공 분야에 대한 용어의 이해를 몇 단계 질문하다 보면, 대략 그사람의 어떤 능력이 좋은가 판단할 수 있다(물론 면접관의 능력에 따라 차이는 나겠지만). 따라서, 가능하면 실무를 많이 해볼 것을 권장한다.

어떤 회사에 소속되어 일하든, 아니면 경진대회 같은 것에 참여하든 자신의 실무 능력을 키워나가야 한다. 주의할 점은 팀으로 일할 때 자신이 맡은 분야는 "팀원들이 일을 잘할 수 있는 분위기를 만들었어요."와 같이 대답해서는 안 된다는 것이다. 차라리 "제가 맡은 분야는 이러 이러한 분야이며, 전 어떤 문제를 어떤 식으로 해결했어요.”라고 말하는 것이 좋다. 신입사원을 채용하려는 회사는 PL급을 뽑지를 않는다. 실제로 일할 사람을 뽑는다. 손을 더럽힐 사람이 필요한 것이지, 우아한 백조의 걸음걸이가 필요한 것이다. PL급은 실무와 이론을 겸비한 사람을 뽑기에 주로 경력을 가진 사람들이 채용대상이다.

삼성전자와 같은 회사에서 시행하고 있는 소프트웨어 멤버쉽(Software Membership)같은 것도 좋은 기회다. 그런 곳에 가기 위해서는 남들이 다하는 것보다는 자신만의 독창성을 가지는 작품을 준비하는 것이 중요하다. 단순히 마이크로 마우스(Micro Mouse)와 같이 남들이 많이 하는 작품을 만드는 것은 면접관들에게 그렇게 좋은 인상을 주지는 못한다.

면접은 가능성만 보는 것이 아니라 그 사람의 능력도 본다. 문제를 해결하는 능력과 자신의 주장을 논리적으로 전달하는 사람을 주의 깊게 본다. 정확히 듣고 의도를 파악해서 논리적으로 대답하는 것이 중요하다. 공대에 있으면서 어학연수로 외국에 한동안 나가서 말 그대로 어학만 배우고 오는 것은 어찌 보면 낭비다. 공대생에게 필요한 것은 실무능력이며, 어학은 전공영어에 대한 이해 정도면 충분하다. 기술자끼리의 대화에 화려한 수식어구를 사용하지는 않으며, 이미 전공에서 배운 용어들이 난무하기 때문이다. 정확히 용어를 이해하고 이를 실무에 적용한 경험이 있다면 어렵지 않게 외국 기술자들과 대화를 나눌 수 있다. 실무를 절대 가볍게 생각하지 말고, 진지하게 사소한 것일지라도 나무가지부터 차근차근 보기 바란다.

**08. 이론**

대체로 실무를 잘하는 사람일수록 이론에 약하다. 실무를 많이 해서 프로그래밍 숙제가 주어지면 잘 만들어서 가져오지만, 나중에 구현한 내용에 대해서 이론적으로 설명해 보라고 하면 대체로 조용한 사람들이 그런 부류다. 주로 이런 사람들은 3~5년이 고비다. 직장에 들어오면 실무를 잘하기에 여러 가지 실무적인 일을 남들보다 더 많이 맡는 경향이 있다. 그리고, 계속 열심히 자신에게 주어진 일을 해 나가고, 진급도 대체로 빠른 편이다. 능력을 인정받기에 다른 사람들의 평가도 후하다. 하지만, 3~5년이 지나면 자신이 하는 일에 대해서 회의감이 든다. 다양한 일을 하지만 대체로 반복적인 업무를 하게 되며, 차츰 다른 사람과의 차이도 줄어드는 것을 느끼게 된다. 결국은 자신이 부족한 것이 무엇인지를 깨닫게 되고, 다시 학교로 돌아가게 되는 것이다.

기왕에 학교에 다닌다면 이론적인 깊이를 더하도록 노력하자. 학교 공부랑은 별로 상관없이 자신이 하고자 하는 것만 하면 된다는 생각은 잘못된 것이다. 그렇게 생각하면 학교보다는 학원에 가는 것이 더 좋다. 배우는 것도 실무 위주의 프로젝트이니 금방 따라가는 것처럼 보일 것이다. 하지만, 수영에서 기초를 잘 배운 사람이 먼 거리를 더 빨리 갈 수 있듯이, 학교에서 기본을 잘 다진 사람이 오랫동안 업계에서 살아남을 수 있다. 극과 극을 달리는 학교 성적표(교양 과목과 전공 과목에서 보이는 현격한 차이)는 좋지 않다. 항상 꾸준한 사람이 어디서든 빛을 보기 마련이다.

이론은 남들이 이미 실제로 경험한 것들을 잘 정리해 놓은 시험공부를 하려는 노트와도 같다. 물론 스스로가 이론을 정립할 수준이 된다면 더할 나위가 없겠지만, 최소한 남이 정리해 놓은 것이라도 잘 알아야 하지 않을까? 교양 과목을 열심히 듣고 학점을 잘 받는 것도 중요하지만, 정말 중요한 전공은 다 놓치고 지나간다면 의미 없는 학교 생활이 될 뿐이다. 언제든 만회할 수 있을 것처럼 보이던 것들이 어느 날 갑자기 절대 따라잡지 못할 벽과 같이 느껴지는 것은 과거의 나태함에 대한 복수이다. 비록 지금 당장은 모든 것을 다 이해하지는 못할지라도, 노력만큼은 언제든 보답을 받기 마련이다.

**09. 열정**

예전에 누군가 이런 이야기를 했다. A부터 Z까지 철자마다 1에서 26까지의 숫자를 주고, 합쳐서 100점이 되는 단어는 "ATTITUDE"라고. 즉, "마음 가짐"만 잘되어 있으면 안 될 것이 없다는 것이다. 열정을 가지고 하는 사람에게는 아무도 이길 수 없다. 즐긴다는 것도 그러한 열정이 바탕이 되지 않는다면 불가능한 일이다. 가능성이란 단어도 열정이 없는 사람에게는 자신의 사전에서 찾을 수 없는 단어이며, 발전하기 위해서는 그만큼의 노력이 필요하다. 열정을 가지고 하지 않고 단순히 주어지는 것만을 받아들이다 보면, 인생은 기다림의 연속일 뿐이다. 찾아 나서지 않는다면 발견할 수 없는 것이 당연하지 않을까?

공대생이라면 집요하리만큼 파고드는 자세가 필요하다. 어떤 주제가 주어졌을 때, 쉽게 만족하기보다는 항상 이유를 찾는 것이 기본이다. 열정만을 가져서도 안 되겠지만 열정이 없어서도 안 된다. 몸은 움직이는데 마음이 가지 않는다면 그만큼 힘든 일이 기다릴 것은 자명하다. 강사로 교단에 서 있는 사람들의 눈에는 수업을 경청하는 사람과 그렇지 않은 사람을 한 눈에 알아볼 수 있다. 관심이 없는 수업에 참여한 학생들을 억지로 끌고 다니면서 물을 먹이고 싶어 할 양치기는 없다. 그냥 목말라서 죽더라도 목마름의 근거를 알아야 하지 않겠는가?

오래전에 한 교수님이 계셨다. 그 교수님은 수업의 첫 시간에 들어와서 이런 말을 했다. "여러분 무엇을 알고 싶어 하나요?" 거기 모인 학생들은 난감해하면서 아무런 대답도 못 했다. 교수님은 그냥 "그럼, 이만"하고선 수업을 끝내버렸다. 다음 시간도 마찬가지로 시작했고, 학생들도 마찬가지로 아무런 대답을 하지 않았다. 당연히 교수님도 수업을 거기서 마치고 그냥 나가버렸다. 결국, “무엇을 알고 싶어 하는지도 모르는 사람에게는 아무것도 가르칠 것이 없다”는 뜻이다. 내가 무엇에 갈증을 느끼고 무엇을 하고 싶어 하는지도 모른 체 학교에 다니고 있는 것은 아닌지 자신에게 질문해보라. 그리고, 이 길이 아니라고 생각될 때는 다시 왔던 길을 돌아가기를 권한다. 그 상태로 더 많이 가면 돌아가는 길만 길어질 뿐이다.

앞에서 나열한 여러 가지 이야기는 나에 대한 반성일 수도 있을 것이다. 여기서 나열한 순서는 특별한 의미를 가지지 않는다. 그냥 머리 속에서 맴돌던 것들을 순서대로 꺼냈을 뿐이다. 인간은 자신이 못했던 것을 남들에게 강요하는 나쁜 습성이 있는지는 모르지만, 어쨌든 학교 다닐 때는 잘 모르고 그냥 지나쳤던 것들을 한 번쯤은 정리해 두어야겠다는 생각으로 쓴 것이니, 너무 부담을 가지고 읽지 않았으면 한다. 이런 모든 것 들을 다 제대로 했다면 아마도 이 글을 쓰는 본인도 지금쯤은 다른 사람이 되어 있을지도 모를 일이다. 결국, 중요한 것은 어린 시절에 듣던 나이 드신 분들의 충고가 아닐런지. 같은 소프트웨어 분야의 개발자로서 선배의 말을 잘 들으면 “자다가도 떡이 생길지 모르니” 한번쯤은 고민해 보기 바란다.

# [ 네트워크의 본심 ]

인간은 이성적인 동물이다. 자신이 누군가를 밝히고 일할 때는 명확히 옳다는 신념을 가지고 행동하기를 원하지만, 익명으로 행해지는 행동에 대해서는 책임지길 원하지 않는다. 네트워크는 익명으로 이루어진 사회이며(혹은, 실명을 가장한 익명), 책임도 자신이 아닌 다른 누군가가 대신 지길 원한다. 물론, 이것은 인간의 양면성을 이야기하는 것이지, 인간이 본질적으로 옳다 그르다는 문제를 이야기하는 것이 아님은 알 것이다.

네트워크를 이루고 사는 방식도 진화하고 있다. 처음에는 텍스트(Text)로 된 신경을 써서 읽어야지 아는 수준에서, 점차 한눈에 들어오는 그림으로, 그리고, 이제는 나 대신에 네트워크에서 자동으로 내가 알아야 할 것들을 알려주는 수준에 도달했다. 하지만, 네트워크가 알려주는 정보가 모두 진실일까? 진실이기보다는 내가 알고 싶어 하는 것(내가 보고 싶어 하는 것)을 골라서 주고 있는 것일지도 모른다. 혹은, 누군가 자신에게 불리한 정보를 이미 걸러서 알려주는 것일 수도 있다.

네트워크는 이미 어떤 사람이 어떤 페이지들을 주로 오픈(Open)했고, 무슨 물건을 샀는지, 어떤 글을 썼는지와 같은 정보를 이용해서, 그 사람을 판단하고 관련된 정보를 그 사람에게 전달해 주고 있다. 친한 사람들의 근황은 잘 알려주면서 그렇지 못한 사람이 어떻게 살고 있는지는 전혀 관심을 기울이지 않도록 만들어 준다. 인간은 네트워크를 통해서 관계를 확대하는 것이 아니라, 오히려 관계가 점차 줄어들고 있는지도 모른다.

자신이 알고자 하는 것만 아는 세상, 자신이 듣고자 하는 것만 듣는 세상. 소통이 막힌 세상이 네트워크의 본심이 아닐까? 하지만 어쨌든 오늘도 인터넷에 들어와서 검색하는 많은 사람은 자기 생각과 경험을 공유하고 있으니, 또한 네트워크의 양면성은 사람의 본질을 닮아갈 것은 분명하다. 네트워크 자체가 사람이 살아가는 세상이며, 그 세상 역시 실제 세상과 다르지 않게 변해갈 것이다. 결국 사람의 생각이 옮겨가는 모든 곳은 사람의 사회를 본 딴 형태로 발전해 나가기 마련이다. 소외된 사람도 네트워크의 일부일 뿐이다.

# [ 간결함의 힘 ]

간결함은 힘을 가진다. 간결하다는 것은 어떤 것을 보았을 때 그 구조나 내용, 혹은 의미하는 바를 금방 알 수 있다는 것과 동일하다고 생각한다. 간결하게 작성된 문서를 보았을 때 사람들은 그것이 의미하는 바를 즉각적으로 이해할 수 있고, 저자가 원하는 것이 무엇인지를 쉽게 잡아낼 수 있다. 필요한 내용만 나열된 것은 사람들의 기억을 돕고, 절대적인 진리라도 되듯이 머리 속에서 언제든 쉽게 끄집어 내올 수 있다. 어지럽게 나열된 것들을 특정한 표나 틀로 만들어서 정리하면, 그것이 설령 잘못된 것일지도 모르지만, 일단은 믿음으로 지켜보는 것이 일반적이다.

간결함은 완전함을 의미한다. 간결하다고 해서 모든 것이 다 빠져야 한다는 것은 절대 아니다. “그 자체로서 스스로를 설명하기에 충분하다”는 것을 의미한다. 가끔 발표회에서 지나치게 간결함에 너무 치우쳐 정작 전달한 내용은 빼먹고 엉뚱한 것들만 몇 자 나열한 자료를 보는 경우가 있다. 발표자는 자신의 발표 실력이 높음을 표현하기 위해서 암묵적으로 스티브 잡스의 스타일을 본받아 보려고 했지만, 그런 자료를 보았을 때 그냥 밖에 나가서 쉬고 싶어진다. 따라 하기 위해서는 최소한 그보다 10배는 더 많이 연습해서 더 정리를 해야 한다. 쉽게 따라 할 수 있는 것은 가치가 오래가지 않는 것들이기 때문이다. 그리고, 발표 스타일은 쉽게 따라하기도 힘들다. 나름의 카리스마와 “축약”의 묘미를 그들은 이미 잘 알고 있기 때문이다.

소프트웨어도 당연히 간결해야 한다. 간결하게 작성된 소프트웨어에 필요한 기능이 없다는 것은 아니다. 여기서 말하는 간결함은 “사용자의 기능/비기능 요구사항을 만족하면서 자신을 설명하기에 충분하다는 것”을 의미한다. 복잡한 알고리즘을 구현하는 것은 어려운 일이며, 그것을 이해하기는 더 어려운 일이다. 이러한 일을 즐기면서 하는 사람들이 있다면, 그것은 자신의 지적 허영을 충족할 뿐이지 결코 남을 위한 것은 아니다. 소프트웨어 개발자라면 항상 남을 위해서 한 번 더 생각해야 한다. 자신이 구현한 기능/비기능에 즐거워하는 사용자들, 자신의 코드를 보고 "우와~"하는 감탄사를 말하는 다른 개발자들을 상상해 보라. 간결함은 그런 모두를 위한 것이어야 한다.

소프트웨어를 작성할 때는 어떤 간결함을 갖는 것이 좋을까? 모르긴 해도 다음과 같은 것들은 필요하다고 보인다(물론, 여기서 제시하는 것들에 대해서는 충분히 이견이 있을 수 있다).

* 01. 모든 이름을 붙여야 할 것들은 정확히 자신을 잘 설명할 수 있어야 한다. 파일, 변수, 함수, 클래스(Class), 구조체(Structure), 포인터 등등 모든 프로그램의 구성요소(Entity)들에 이름을 적절히 주어야 한다.
* 02. 구체적인 것과 추상적인 것은 나누어서 계층으로 분리되어야 한다. 구체적인 것과 추상적인 것이 섞여 있으면, 코드를 읽는 사람의 입장에서는 이해하기 어렵다(자신이 무엇을 보고 있는지 알기 힘들다).
* 03. 지나치게 축약된 짧은 코드보다 풀어서 쓴 긴 코드가 더 좋다. 짧게 축약해서 만들면 코딩을 잘하는 것 같지만, 그것을 보고 이해해야 할 사람의 입장에서는 죽을 맛이다. 그냥 쉽게 쭉 나열식으로 써도 상관없다. 코드가 스스로 잘 설명될 수 있도록 해야한다.
* 04. 긴 코드 보다 짧게 나누어진 여러 개로 만드는 것이 좋다. 짧은 코드는 하나의 개념에 한정시켜서 작성되므로 쉽게 이해가 가지만, 길게 작성된 코드는 한 번에 다 이해하기 어렵다.
* 05. 한 번에 한 가지 일만 하도록 한다. 여러 가지 일을 한 번에 처리하는 것은 어렵다. 따라서, 될 수 있으면 한 번에 한 가지만 처리하는 코드를 작성하고, 이러한 코드들을 합쳐서 큰 일을 처리하는 코드를 만드는 것이 효과적이다. 문제가 생겨도 쉽게 고칠 수 있으며, 수정되었을 때 나타나는 파급효과도 작다. 이해하는 것도 당연히 더 쉽다.
* 06. 반드시 결과는 확인한다. 보통의 경우 항상 사람들은 긍정적인 부분만을 보는 경우가 많다. 하지만, 부정적으로 "그렇게 안 되었을 경우에는 어떻게 되지?"를 고민해야 한다. 대부분 오류는 이런 곳에서 발생한다. 함수의 호출 결과나 예상한 입력이 아닌 경우도 확인해야 한다.
* 07. 지나치게 많은 출력을 보여주기보다는 꼭 필요한 출력만을 보여주고, 나머지는 정보는 로그(Log)로 남긴다. 필요하다면 남겨진 로그를 분석하는 프로그램을 만들 수 있도록, 출력 형식도 고려하도록 한다. 가능한 정보는 “텍스트”로 만드는 것이 좋다. 대부분의 툴들은 “텍스트”를 잘 다루기 때문이다.
* 08. 프로그램과 프로그램 간의 연결 및 모듈 간의 연결, 함수 간의 연결, 쓰레드(Thread)간의 공유 변수 등등은 최소화할 수록 좋다. 전달되는 인수도 최소화할 수록 좋다. 전달하는 내용이 줄어들면, 그걸로 인해서 발생하는 오류도 당연히 줄어든다. 필요하다면 한번에 묶어서 전달하는 것도 가능하다.
* 09. 프로그램은 짧게 작성하는 것(최적화하는 것)에 힘들이지 말고, 깔끔하게 작성하는 데 노력하자. 다소 좀 용량이 커지거나 성능이 조금 낮더라도 충분히 감내할 수 있는 하드웨어는 있기에, 추가로 유지보수에 드는 노력을 줄이는 것이 효과적이다. 물론, 최적화 노력이 의미가 없다는 것이 아니라, 최적화 전에 구조화 부터 하라는 의미다.
* 작게 반복해서 해결하려고 노력해야 한다. 한번에 많이 하려고 하면 실패할 가능성이 높다. 반복할 때는 기능 구현에 초점을 맞춰야 하며, 모듈별 구현은 통합이나 테스트 문제를 나중으로 미루기 때문이다. 문제를 빨리 찾고자 한다면 구현된 기능별로 테스트를 반드시 실행해야 한다.

이외에도 몇 가지 더 있을 수 있겠지만 일단 생각나는 것만 보면 대략 이 정도다. 핵심을 보면 간결하게 자신을 스스로 잘 설명해 주는 코드가 가지는 가치는, 다른 어떤 것보다 우선적으로 달성해야 할 소프트웨어 개발자의 목표이다. 어떻게 보면 쉽게 달성될 수 있을 것처럼 보이지만, 생각없이 코딩하다보면 정말 달성하기 힘든 일이 될 수도 있다.

간결함은 노력의 결과로 만들어지는 습관이며, 하루 이틀 신경쓴다고 달성할 수 있는 목표가 아니다. 따라서, 의식적인 노력을 반복적으로 할 때만 자신의 코드 속에 녹아들게 만들 수 있을 것이다. 처음에는 의식적으로 반복해야 하지만, 일상이 되면 습관처럼 자연스러운 행동으로 나올 것이다.

# [ 어떻게 짜야 잘 짰다는 소리를 들을까? ]

소프트웨어 개발자는 매일 프로그래밍을 하면서 산다고 해도 지나친 말이 아니다. 어떨 때는 걸어가면서 머리 속에서 프로그램을 짜기도 하고, 버스나 지하철 속에서 졸면서 생각하기도 한다. 물론, 전부 다 그렇다는 것은 아니다. 무엇인가에 집중할 때 이런 일이 종종 생긴다. 어떤 아이디어가 생각나면 그 자리에서 잠시 메모를 하고, 사무실이나 컴퓨터를 사용할 수 있게 되면 코딩부터 하게 된다. 프로그래머만이 이런 것은 아니며, 일에 집중하는 모습을 보이는 사람들은 흔히 이런 경험을 했을 것이다.

생각나는 대로 프로그래밍을 하는 습관은 정작 중요한 점을 놓치기 쉽다. 글을 쓰는 사람은 구조를 가지고 쓴다는 말을 흔히 한다. 소설의 예를 들면 이해가 빠를 것이다. 소설가의 경우 글로 쓸 소재를 발견하면, 교과서에서 이미 보았듯이 “구성”을 먼저 생각한다(물론, 소설가의 머리속을 들어가본 적은 없지만). 머리 속에서 생각을 하든, 아니면 칠판이나 노트에 쓰든, 뭔가 정리를 한번 한 후에 쓰기 시작한다. 긴장감이 있어야 할 부분은 어디며, 독자에게 카타르시스를 줄 부분은 어딘지를 잘 생각해서 글을 쓰는 것이다.

프로그램을 만드는 일도 기본적으로 글을 쓰는 일과 같다고 생각한다. 개발자들은 흔히 어떤 프로그래밍 언어(Language)를 사용한다고 이야기한다. C언어, C++, C#, Fortran, Pascal, Basic 등등 다양한 많은 프로그래밍을 위한 언어들은 과거에서 지금까지 꾸준히 개발됐다. 이러한 언어들의 특징은 사람의 생각을 기계가 이해하기 알맞은 구조로 변환해 준다. 즉, 사람은 자신이 생각하는 주제로 글을 쓰고, 그 글을 읽어서 컴퓨터가 처리하는 것이다. 하지만, 프로그래밍의 경우는 글을 읽히는 대상이 컴퓨터만 해당하는 것이 아니라, 같이 일하고 있는 동료 개발자나 혹은 미래의 불특정 사람이 될 수 있다는 점이다. 즉, 사람을 대상으로도 코드는 작성된다는 것이다.

어떤 글은 감동을 주지만, 어떤 글을 그야말로 읽는 것 자체가 고역이다. 그럼 어떤 이유에서 그런 감동을 주는지 보도록 하자. 사람들이 감동하는 부분은 그 글에서 작가의 생각을 충분히 읽어내고 공감하는 데서 온다. 즉, 작가의 의도를 이해하고, 자기 생각과 경험에 비춰서 어떤 감정의 변화를 유발한다고 볼 수 있다. 좋은 코드는 그 자체로 감동이 된다. 개발자에게 감동을 주는 부분은 코드가 작성자의 의도를 명확하게 이해하기 쉽도록 구성되어 있으며, 설계와 일맥상통할 때라고 볼 수 있다.

대부분의 경우 처음 어떤 코드를 넘겨 받았을 때, 우리는 코드를 설계한 문서를 요구한다. 그리고, 대부분의 경우 그런 문서는 거의 없거나, 실제의 코드와 다르게 되어 있는 경우가 많다. 여기서 감동은 이미 쑥 들어가 버리고 만다. 그럼 이제는 문서에서 받은 느낌을 바꾸기 위해서 코드를 보기 시작할 것이다. 하지만, 아무리 읽어도 이해가 안 가는 부분들이 많이 있고, 여기저기에 데드 코드(Dead Code)들이 주석으로 막혀있다. 주석 위에는 이름의 이니셜(Initial)과 날짜만 적혀있을 뿐, 왜 그렇게 했는지는 찾지 못한다. 작성한 사람을 찾기도 힘들지만, 찾았다고 해서 물어봐도 명확한 답을 얻기 힘들 것이다.

감동을 주기 위해서는 코딩에 있어서 “달인”이 되어야 한다. "생활의 달인"이라는 TV 프로그램을 보면 전부 몸으로 달인이 된 경우만 나온다. 물론, 그런 사람들이 하는 일이 중요하지 않다는 것이 아니다. 여기서 말하고 싶은 것은 프로그래밍을 한다면 적어도 그런 달인의 경지를 목표로 끊임없이 노력해야 한다는 것이다. TV속에 나오는 달인들도 몇 년을 동안 반복해서 몸으로 익혀서 그러한 경지에 도달했다. 3갑자의 내공(대략 60년 x 3 = 180년? 일명 노화순청, 환골탈태의 경지)이 쌓일 정도는 아니지만, 최소한 3년 이상은 그런 과정을 겪으면서 노력했다는 것이다.

프로그래밍을 한다면 적어도 그 정도는 되어야 할 것이다. 자신의 코딩 스타일을 노력을 통해서 바꾸지 않는다면, 정말 코딩을 잘한다는 사람들의 반열에는 오르지 못할 것이다. 오프 소스 과제의 리더급 정도 되는 사람들이나, 소스 코드 트리(Tree)를 관리하는 인물들은 이미 그런 반열에 이미 올라섰다고 봐도 된다. 그들이 작성한 코드는 혼자만의 노력으로 이루어진 것이 아니라, 수많은 사람들의 리뷰(Review)와 참여(Contribution)를 통해서 단련된 것이다.

이제 프로그램을 짜는 일은 단지 그냥 생각을 그대로 옮기는 일이 아니라는 것을 충분히 이해했을 것이다. 프로그램을 짠다는 것은 되새김질한 생각을 구조화시켜서 프로그래밍 언어를 통해서 표현하는 것이다. 프로그램의 구조는 추상화(Abstraction)의 결정체로 이루어지며, 우리는 대부분 추상화시키는 과정을 머릿속에서 진행한다. 아주 세밀한 부분은 나중으로 미루고 관점을 높여서 본다. 먼저 산을 보고, 나무를 보고, 나뭇잎을 보고, 나뭇잎에 붙은 곤충들을 들여다본다. 그리고 때로는 곤충을 관찰하다가 그것이 붙어 있는 나무를 보고, 숲을 보고, 다시 산을 보기도 한다. 이렇게 가장 큰 그림에서 가장 작은 구체적인 그림까지 왔다 갔다 하면서 생각을 추상화하게 된다.

이를 가장 잘 표현할 수 있는 것이 계층화(Layering)라는 개념이며, 상위 계층으로로 갈수록 추상화 수준은 높아지며, 하위 계층으로 갈 수록 세세한 것들이 구체화되어간다. “Bottom-up”이건 “Top-down”이건 상관없이 이렇게 왔다 갔다 하면서 구현할 단위로 세분화하게 되는 것이다. 생각은 추상화를 통해서 정말 자신이 하려고 하는 일에 집중할 수 있다. 또한, 구체화하면 할수록 구현에 집중할 수 있다. 우리는 추상화와 구체화라는 양 극단의 속에서 왔다 갔다 하면서 자유로이 이동할 수 있는 정도의 수준에 올라야, 제대로 구조화된 프로그램을 짤 수 있게 될 것이다.

여기까지 왔다면 이미 어느 정도 구현해야 할 항목이 무엇이며, 전체 시스템이 대략 어떻게 이루어져 있는지 파악할 수 있을 것이다. 실제로 구현 단계에 들어가면 이제는 또 다른 노력이 요구된다. 개별 구조들은 다 정리를 했는데, 이것을 디렉토리(폴더)별로 어떻게 배치할 것인가를 정하는 문제와 다른 사람들과 어떻게 협업을 해야 할 지를 정해야 한다. 이때는 다양한 툴과 팁(tip)들이 적용될 수 있을 것이며, 또한 개발 방법론도 다양하게 적용할 수 있을 것이다.

소프트웨어 공학을 들먹이지 않더라도 이미 개발자는 어느 정도 몸으로 것들이 있을 것이라고 본다. 중요한 것은 머릿속에 그려진 그림을 옮겨 놓기 위해서 물리적으로 잘 분리를 시키는 것이다. 개념적으로만 분리하면 된다고 생각하는 사람들이 있는데, 이것은 곧 허물어질 수 있는 여지가 많다. 새로운 프로그래머가 들어와서 제대로 교육받지 못하면, 코드는 순식간에 고치기 여려운 구조로 바뀔 수 있다. 따라서, 개발되어야 할 모듈을 생각했던 구조와 동일한 물리적인 구성을 가지도록 정하는 것이다. 물론. 이것에 대해서는 다양한 의견이 있을 수 있지만, 최소한 구조 설계 설계서에 나왔던 이름을 이용해서 디렉토리를 만들어서 관리하는 습관을 들이는 것이 좋다. 최소한 문서와 코드를 일치시켜 찾고 싶은 것이 어디에 있는지는 알 수 있기 때문이다.

장인의 냄새를 자신의 코드에서 맡고자 한다면 멈추지 않는 노력이 필요하며, 이는 체계적으로 배워야 조금이라도 빨리 도달할 수 있는 목표다. 시중에는 이미 많은 좋은 책이 있지만, 여기서 몇 가지 이야기한다면, “Code Craft”, “Clean Code”, “Code Complete” 등이 도움이 될 수 있을 것이다. 프로그래밍은 빨리 많이 짜는 것이 능사가 아니며, 항상 유지보수와 관리가 필요한 손을 많이 타는 애완동물이다. 조금만 애정 주기를 게을리하면, 곧바로 달려와서 내 엉덩이를 물어 뜯을지도 모른다. 하지만, 애정을 들인 만큼 충분한 댓가를 주는 것이 또한 코딩이다.

# [ 테스트 주도 개발(TDD) ]

TDD(Test Driven Development)라는 방법을 최근에 알게 되었다. 몇 권의 책을 사서 읽어보고 사용할 수 있는 방법이 무엇인지 찾아본 후에 내린 결론은, 개발자의 코딩 문제점을 해결하기 위한 근본적인 대책이라는 결론을 내렸다. 왜 그러한 결론이 나온 것인지 보도록 하겠다.

**01. 전체 개발과정의 대부분은 테스트와 디버깅(Debugging)이다.**

어떤 프로그램을 만든다는 것은 주어진 요구사항을 구현하는 일이다. 주어진 요구사항이 잘 구현되었는지를 확인하는 방법은 테스트밖에 없으며, 테스트하는 과정에서 발생한 모든 버그는 고쳐야 한다. 물론, 잠재적인 버그는 모든 프로그램에 다 있지만, 적어도 발견된 버그에 대해서는 개발자의 입장에서 용서가 안 되기 때문이다. 전체 개발 과정에 들어가는 공수를 100이라고 했을 때, 테스트와 디버깅이 대략 70을 차지한다(유지보수를 추가한다면, 당연히 유지보수가 70%를 차지 하지만). 즉, 실제로 코딩하는데 들어가는 노력은 30정도 빡에 안될 것이다. 그렇다면, 우리는 70%이라는 공수가 들어가는 것에 대해서 어떤 대책을 세우지 않으면 안 된다. TDD는 바로 이것에 대한 좋은 해답을 제시할 수 있다고 본다.

**2. 테스트를 먼저 작성하는 것은 당연한 활동이다.**

우린 어떤 모듈을 개발하면서 그 모듈이 제대로 동작한다는 것을 보여주기 위해서 반드시 뭔가를 작성하는 버릇이 있다. 개발자라면 최소한 자신이 개발한 모듈의 동작을 검증하기 위한 프로그램을 반드시 짠다(적어도 테스트를 한번은 돌려보기 위해서 실행해 본다). 여기서, 검증하는 프로그램과 검증의 대상이 되는 프로그램이 나누어지는데, 검증하는 프로그램을 “xUnit”과 같은 “테스트 프레임워크”를 활용하는 것이 “테스트 자동화”를 위해서는 필요하다. 모든 테스트는 소스코드 변경 시 반드시 반복해서 실행되어야 하기 때문이며, 또한 그것을 편리하게 해 줄 수 있는 것이 테스트 프레임워크이기 때문이다.

**3. 리팩토링(Refactoring)을 하기 위해서는 먼저 제대로 동작하는 코드를 얻어야 한다.**

모든 테스트는 일단 동작하는 프로그램을 대상으로 할 수밖에 없다(제대로 동작하는 것은 테스로 확인하는 것이지만). 동작하지 않는 코드를 가지고 테스트를 할 수는 없다. 동작시키기 위해서 가장 간단한 방법은 “동작하는 테스트 프로그램”을 먼저 만드는 것이다. 그 테스트 프로그램이 테스트 대상이 되는 기능을 수행하지 못한다면, 제대로 구현하지 못했다는 뜻이다. TDD에서는 테스트를 먼저 기술하고, 그 테스트를 만족시키는 만큼만 코딩할 것을 요구한다. 따라서, 테스트를 통과했다면, 적어도 테스트하고자 하는 기능은 구현되었다고 봐야한다.

검증을 어떻게 하는지 먼저 계획을 수립하고, 구현을 행동에 옮기는 것이 프로그램뿐만이 아니라 다른 분야에서도 일반적이다. 어떤 것을 구현하는데 필요한 것은 시간과 돈, 사람들이며, 충분히 주어지면 개발하지 못할 것은 거의 없다. 제시하는 모든 스펙(Spec.)은 검증의 대상이며, 그것을 만족시키는 것이 개발의 목표다. 따라서, 검증 기준을 제시하고 구현하는 것은 전혀 새로운 일이 아니라는 것이다.

리팩토링은 테스트할 수 있을 때 가능하다. 외부에 보이는 행동을 변화시키지 않고 내부구조를 변경해서, 더 좋은 구조를 가지도록 만드는 것이기 때문이다. 변경된 코드가 외부에 같은 동작을 한다는 것을 보장하지 않으면, 리팩토링 할 이유가 없는 것이다. 엄밀한 의미로 리팩토링이라기 보다 그냥 구조 변경이며, 변경 이전과 이후에 같은 동작을 보장하는 것이 테스트이기 때문이다. 리팩토링 전에 테스트가 존재하지 않으면, 리팩토링 후에 같은 동작을 한다고 보장할 수 있는 방법이 없다.

몇 가지 간단한 것들을 적어보았지만, 개발자가 이러한 TDD를 제대로 하기 위해서는 근본적인 생각(Mind, 혹은 자세)의 변경이 없으면 불가능하다. 지금의 개발 형태로는 주어진 시간과 인력, 예산으로는 적기(Time-to-Market)에 시장의 변화에 제대로 대응하기 어려우며, 오후 5시에 정시 퇴근하는 개발자도 찾기도 힘들 것이다. 주말에 집에서 제대로 휴식하기 위해서는 최대한 컴퓨터들이 자동으로 해주는 무엇인가를 필요로 하며, TDD는 개발 방법에 대한 개발자의 “사소한” 습관이 되어야 할 것이다.

# [ 나는 프로그래머다. ]

우리는 매일 같이 프로그램을 짜고, 테스트하고 버그를 잡고 있다. 이러한 일을 거의 특별한 일이 없으면 매일 같이 반복되고 있으며, 가끔은 멍하니 있을 때 좋은 아이디어가 떠오르기도 한다. 우리가 프로그래머라고 이야기할 수 있는 것은 이러한 것이 우리의 “업”이기 때문일 것이다. 그렇다면, 우리는 “진정한 프로그래머일까?”. 만약, 아래와 같이 일하고 있다면 “프로그래머”라고 이야기할 수 있을 것이다(물론, 이것도 전적으로 개인적인 생각일 뿐이다.).

**01. 오늘 작성한 코드에 컴파일러가 던지는 경고 메시지(warning message)는 없는가?**

우리가 작성하는 모든 코드에 대해서 절대 컴파일러가 만들어내는 경고 메시지는 발생하지 않아야 한다. 최소한 그 정도는 다른 사람에 대한 예의로 처리하는 기본자세가 필요하다. 기계적으로 점검할 수 있는 단순한 것들을 사람이 수정하지 못한다면 말이 되겠는가? 단순히 발생하는 경고가 많을 것이므로 어렵지 않게 고칠 수 있을 것이다. 만약 경고가 나오는 코드를 다른 사람들이 본다면 어떨까? 과거에 다른 사람의 코드에서 컴파일러 경고 메시지가 나오는 것을 보았을 때의 느낌이, 자신의 코드에서도 남들이 똑같은 느낄 수 있다는 것은 결코 유쾌한 경험이 아닐 것이다.

**02. 오늘 작성한 코드를 테스트할 수 있는 코드를 짰나?**

굳이 TDD(Test Driven Development) 라는 말을 사용하지 않더라도, 모든 작성된 코드는 테스트될 수 있어야 한다. 가장 쉬운 방법은 테스트 코드를 먼저 작성하고, 원래 짜고자 한 함수를 만들어 나가는 것이다. 나중에 검증 시간을 단축하기 위해서라도 코드를 작성하는 바로 그 순간에 가장 필요한 노력이다. 테스트를 만드는 노력을 게을리한다면, 언제고 과거에 대한 복수가 가장 힘든 시기에 찾아올 것이다. 추가적으로 테스트 코드들을 한 번에 전부 컴파일해서, 한 번에 전부 실행해 볼 수 있다면 좋을 것이다. 결국, 단순 반복은 사람이 절대 할 일이 아니라는 것이며, 프로그래머는 무의미한 반복에서 벗어날 수 있어야 창조적인 일에 힘을 기울일 수 있다. 자신이 하는 일에 빠져서 무의식적으로 반복하고 있다고 느끼는 순간, 더는 창조적으로 일할 수 없다는 것을 알게 될 것이다.

**03. 오늘 작성한 코드가 저장소에 들어가서 제대로 전체 코드와 컴파일 되는가?**

오늘 작성한 코드는 반드시 저장소에 들어가서 다른 사람이 작성한 코드와 통합되어야 한다. 비록 실행에 오류가 발생할 가능성은 있지만, 그래도 반드시 컴파일은 될 수 있어야 한다. 저장소에 가끔 자신의 코드를 넣어주는 센스를 가지고 있고, 될 수 있으면 충분히 점검 후에 넣어 주겠다는 선택이 최고라고 믿는 사람이 있다면, 통합(Merge)은 힘들고 버그를 많이 유발하는 과정이 될 것이다. 생각보다 통합은 어려운 일이며, 통합 후의 테스트는 그동안 다른 사람들이 했던 일에 대한 것도 모두 점검해야 하므로 많은 시간이 걸릴 것이다. 가장 좋은 것은 조금 동작하는 코드를 만들어서 조금 통합하고, 통합된 부분을 조금 테스트하는 것이다. 조금씩만 전진하라. 한꺼번에 많이 가려고 욕심을 부리는 순간 해야할 일만 늘어날 것이다. 힘든 일은 나누어서 조금씩 하는 것이 최선이다.

**04. 오늘 작성한 코드를 가져와서 미리 작성된 테스트를 자동으로 실행할 수 있나?**

이미 설명했듯이 자신이 작성한 코드는 테스트될 수 있어야 한다. 통합된 코드를 가져와서 다른 사람이 작성한 테스트 및 자신이 작성한 테스트들을 모두 자동으로 실행해 보기를 권장한다(최소한 자신이 만든 테스트라도). 가능하면 테스트 전용 서버를 두고, 거기서 자동화된 테스트 들이 동작하는 것을 확인해야 한다. 오늘 한 작업이 마무리되고, 내일 아침에 출근했을 때 어제 작성한 코드에 어떤 문제들이 발생했는가를 통보받을 수 있다면, 조금 더 퇴근 시간은 빨라질 것이다. 그렇지 않다면, 결국 테스트의 늪에서 허우적거리는 자신을 발견할 것이며, 한 번 더 좌절할 수밖에 없을 것이다. 나날이 작아져만 가는 자신을 보는 것은 힘든 일이기에 최대한 자신감을 가질 방법들을 찾아보기 바란다. 가장 확실한 것은 테스트를 실행해서 검증하는 것밖에 없다.

**05. 오늘 작성한 코드는 다른 사람이 리뷰하도록 요청했나?**

코드 리뷰는 변경된 코드가 적을수록 빨리할 수 있으며, 리뷰하는 사람도 부담이 적다. 리뷰를 특정한 날을 정하고 한다면, 보는 사람이나 리뷰를 요청하는 사람이나 둘 다 부담이 될 수밖에 없다. 특히, 리뷰가 끝나고 나서 리뷰 보고서(리뷰 결과서)를 쓰는 것도 일로 생각될 것이다(물론, 리뷰 결과가 시스템에 등록되는 코드에 코멘트라면 조금 편하겠지만). 시스템을 만들 때 핵심이 된다고 생각되는 부분은 전체 팀원이 다 모여서 리뷰할 필요가 있지만, 지역적인 변경이 다른 곳에 영향을 거의 미치지 않는다면, 가능하면 빨리 리뷰하는 것이 좋다. 리뷰하는 분량은 훨씬 작아질 것이며, 리뷰에 대한 집중력도 높아질 것이다. 코드 리뷰가 시간 낭비라는 생각은 코드 리뷰가 버그를 가장 많이 줄이는 활동이라는 점을 간과한 것이다. 지식의 전파와 시스템에 대한 공통된 인식을 높인다는 점에서도 코드 리뷰는 절대 시간 낭비가 아니다. 오히려 코드 리뷰를 제대로 하지 못한 경험을 지나치게 일반화시킨 것일 뿐이다.

**06. 오늘 요청된 리뷰는 다 마쳤는가?**

남이 요청한 리뷰는 완료해 주어야 한다. 그렇게 하지 않으면, 그 사람은 자신이 작성한 코드를 저장소에 집어넣지 못하므로(규칙으로 정한 경우), 상쾌한 마음으로 집에 가기가 꺼려질 것이기 때문이다. 리뷰의 정책은 “저장소에 넣기 전에 반드시 해야 한다(Pre-commit Review)”와 “저장소에 일단 먼저 넣고 나서 해야 한다(Post-commit Review)”로 나눌 수 있는데, 저장소를 좀 더 깨끗하게 유지하기를 원한다면 반드시 리뷰가 끝난 후에 넣는 것이 좋을 것이다(개발이 바쁜 경우에는 후자를 선택하는 경우도 있다). 다른 사람이 요청한 리뷰를 보지도 않고, 그냥 허가한다면 결국 그 결과는 팀의 생산성에 영향을 주게된다. 제대로 리뷰되지 않은 코드는 문제를 안고 다음으로 진행하는 것과 같기 때문에, 장기적으로 과제 개발 속도를 더디게 만든다(일종의 기술적인 부채(Technical Debt)를 남기기 떄문이다). 오늘 요청된 리뷰는 시간이 걸리더라도 반드시 완료하도록 하자. 요청한 사람의 지위가 아니라, 코드의 질이 중요하다는 사실도 잊지 말자. 일이란 직책으로 하는 것이 아니라 결과로 말하는 것이다.

**07. 오늘 작성한 코드들이 팀에서 정한 코딩 규칙을 잘 지키고 있나?**

“코딩을 어떻게 할 것인가를 정하는 것이 별로 도움이 안된다”고 말하는 사람은 없을 것이다. 대부분 사람은 실수나 다른 방법을 통해서 자신의 스타일을 만들어 간다. 코딩도 마찬가지로 실패했던 경험이 많을수록 더 좋아하지는 경향이 있다. 하지만, 남들이 이미 반복한 실수를 다시 할 필요가 있을까? 실수를 방지하는 방법은 근본적으로 실수할 수 없게 하는 것이다. 하지만, 사람이란 언제나 실수를 할 수 있기 때문에 어쩔 수 없는 부분이기도 하다. 다만, 실수를 가능한 줄여나가는 노력은 항상 필요하다. 더욱이 더 중요한 것은 팀에서 작성한 모든 코드가 마치 한 사람이 짠 듯이 보일 때, 객관적으로 품질도 좋다는 것이 일반적이다. 왜냐고 묻는다며, 팀 워크가 그만큼 잘 이루어지고 있으며, 모든 프로그래머가 잘짜여진 규칙에 의해서 협업을 하기 때문이라고 생각하기 때문이다. 안정적으로 정착된 코딩 규칙을 가지고 있는 팀은 서로에 대한 신뢰가 바탕을 이루기 때문이다.

**08. 오늘 작성한 코드를 남이 읽기 쉽게 짰나?**

가장 좋은 코드는 어떤 코드일까? 여러 책을 찾아봐도 답은 거의 동일하다. 즉, "남이 잘 읽을 수 있는 코드"를 가장 좋은 코드라고 말한다. 읽기 쉽다는 것은 어떤 의미일까? 코드를 위에서 아래로 읽어나갈 때 크게 집중하지 가지지 않더라도 쉽게 읽히는 코드다. 대부분 프로그래머는 이런 경험이 있을 것이다. 하지만, 남이 짠 코드만큼 읽기 힘든 것은 없다. 때로는 이러한 코드를 만나면 새로 짜고 싶다는 충동을 느낄 것이며, 만약 실제로 자신이 짠 코드로 원래의 코드를 대체 하고나면 언젠가 또 다른 사람이 그 부분을 다시 새로 짜고 있을 것이다. 한 번 할 때 제대로 하자. 읽기 힘든 책을 주고 읽어보라고 하는 것은 남을 괴롭히기 위한 것이지, 도와주려는 것이 아님을 명심하자. 그럼 읽기 쉬운 코드는 뭘까? 간단히 이야기하면 간단하게(Simple) 짠 코드이다. 복잡한 연산식 대신에 간단한 수식으로 풀어놓는 것, 적절한 변수의 이름과 함수의 이름을 주는 것, 하나의 클래스는 하나의 역할만 수행하는 것, 모듈이나 함수 간에 너무나 친한 관계(Coupling)를 만들지 않는 것 등등 다양한 것들이 있다. 조금만 신경 쓰면 남들에게 도움을 많이 줄 수 있고, 결국에는 자신도 다른 사람들로부터 도움을 얻을 수 있는 코드를 만들 수 있을 것이다.

**09. 오늘 작성한 코드에 선언한 변수들은 초기화가 되었는가?**

지역 변수들에 대한 초기화는 중요하다. 생각보다 많은 아주 간단한 문제가 똑똑한 프로그래머를 좌절시키는 사례가 있다. 변수는 항상 적절한 값으로 초기화하고 사용하는 것을 명심하자. 사용하지 않는 변수를 제거하는 것도 중요하다. 쓰지도 않는 변수를 남겨놓을 필요는 없다. 그런 변수들은 오히려 사람들에게 오해를 불러일으킬 확률이 높다. 사용되지 않는 함수의 인자도 제거해야 한다. 함수가 넘겨받는 인자들에 대한 값의 유효성도 반드시 점검해야 한다. 만약 잘못된 값을 받는다면, 내가 작성한 코드가 잘못된 값까지 변경해서 처리할 필요는 없다. 잘못된 요청에는 잘못되었다고 답변해주는 것도 규칙이다. 또한, 호출 후에 그 결과로 넘어오는 결과값도 반드시 검사해야 한다. 모든 것이 잘 될 것이라고 근거 없이 예측하는 것은 찾아내기 힘든 버그를 낳을 가능성이 있다. 안전 위주로 방어적으로 프로그램하는 습관이 없다면, 잘못된 가정으로 동작하는 프로그램을 만들게 될 것이다.

**10. 오늘 작성한 코드에 전역변수를 남발하지는 않았는가?**

시스템의 부수적인(Side-Effect) 오류가 발생하는 원천은 전역으로 사용하는 자료구조다. 누가 언제 어떻게 고쳤는지 알 수 없기에, 그리고 내가 수정한 것이 다른 사람이 작성한 부분에 어떻게 영향을 줄 수 있는지 모르기에 절대 금기 해야 할 필요가 있다(물론, 완전히 제거하는 것은 어렵지만). 모든 사람이 다 쓰고, 또 그것이 편하다고 이야기한다면 다시 한번 생각해 보기 바란다. 남이 어떻게 고칠지 미리 완벽하게 예상할 수 있지 않다면, 사용하지 않는 것이 자신의 코드라도 최소한 방어하는 방법이 될 것이다. 그래도, 전역적으로 사용할 자료구조가 필요하다면, 절대 그 자료구조에 대한 직접적인 접근은 막는 것이 좋다. 차라리 적절한 인터페이스(여기서는 함수나 매크로 정의 정도)를 사용해서 그러한 자료구조에 접근하는 톨게이트를 만들어 두는 것이, 미래를 위한 보험을 드는 것임을 잊지 말자. “누구나 접근할 수 있다면, 아무도 책임지지 않는다”와 같은 말이다.

이외에도 여러 가지 질문이 있지만, 기본적으로 위와 같은 10가지만 제대로 지킨다면, 개인적인 생각으로는 아주 훌륭한 프로그래머가 될 수 있다고 본다. 물론, 이 글을 쓰는 본인도 모든 것을 다 잘 지키고 있다고 할 수는 없다. 더군다나 직급이 올라가면서 회의, 결재, 보고서만 작성하고 있고, 개발도구는 오피스 제품만 사용하고 있는 나로서는 일정 관리만 보게 되는 관리자다. 이 핑계 저 핑계를 대고는 있지만, 결국 어쩔 수 없이 팀원들이 언제까지 무엇을 할지만 보고 있을 뿐이다. 최소한 이러한 관리라도 잘하기 위해서, 팀이 가야 할 방향을 제시하고, 어떤 것이 좋은 것인가를 팀원들에게 알리고, 새로운 지식을 주워 담는 일은 게을리할 수는 없다. 프로그램을 짤 때 각각이 해야 할 역할(Role)과 책임(Responsibility)을 명확히 한다면, 그것이 사람이든 기계든 상관없이, 우리가 하는 일에서 발생하는 대부분의 복잡도는 사전에 제거할 수 있을 것이다.

# [ Linux 커널을 아는가? ]

요즘 인터넷 서점을 몇 군데 돌아다보면 상당히 여러 종류의 Linux 커널에 대한 책들이 나오고 있음을 알 수 있다. 흔히들 사람들이 Linux에 대해서 일했냐고 물으면, 대부분은 디바이스 드라이버(Device Driver)나 응용 프로그램관련, 혹은 서버의 설정에 대해 일을 했다고만 이야기 한다. 하지만 조금 더 어려운 Linux 커널에 대한 질문을 던지면 대부분이 대답하지 못한다. 사실 Linux에 대한 일을 몇 년씩 해오면서도 대부분의 사람은 Linux 커널에 대해서 무지한 것이 사실이다.

물론 예전에도 pSOS+나 VxWorks, 혹은 Nucleus나 ThreadX와 같은 RTOS(Real-Time Operating System)를 사용할 때도, 대부분의 사람은 내부 구조에 대해서는 모르고 그냥 사용했던 것이 사실이다. uC/OS-II와 같은 RTOS의 경우에는 너무나 간단하고, 소스 코드 및 그것을 설명하는 책도 있어서 어느 정도 내부구조에 대한 공부가 가능하지만, 그 외의 상용 RTOS의 경우에는 소스 코드에 접근할 수 없었기 때문이기도 하다(Nucleus나 ThreadX, FreeRTOS는 코드를 다운로드 받거나 구해서 볼 수 있다).

하지만 이제는 상황이 달라졌다. Linux 커널은 모두가 다 볼 수 있고, 모두가 실력이 된다면 수정해서 재배포도 가능하다. 사실, Linux와 같은 커널 코드를 분석하기는 쉽지 않은 작업이다. 운영체제에 대한 기본 개념에서 부터 시작해, HW에 대한 이해, 각종 프로그래밍 테크닉등 알아야 할 것들이 너무나 많다. 그리고, 제대로 된 분석서도 부족한 것이 사실이다(물론 제대로 된 분석서는 있다. 알고 싶은 특정 부분에 대한 설명이 빈약한 경우도 있지만). 그런데도 Linux 커널에 대한 이해는 꼭 필요하다.

응용 프로그램을 만들건, 디바이스 드라이버를 만들건 상관없이, Linux 커널과 연관성을 가지고 있기 때문이다. 커널을 잘 이해하면 더 좋은 성능을 낼 수 있는 프로그램을 작성할 수 있기 때문이다. 간단한 예를 들어보면, 폴링(Polling)과 인터럽트(interrupt) 방식의 차이를 이해하지 못해서 생기는 CPU 시간의 낭비나, 지나치게 많은 쓰레드(thread)를 생성해서 컨텍스트 스위칭(Context switching) 오버헤드로 인한 성능의 저하 문제 등, 생각보다 현실적으로 Linux 커널에 대한 지식이 필요한 부분이 꽤 많다.

그럼 왜 사람들은 Linux 커널에 대한 이해를 힘들어 할까? 앞에서도 잠시 이야기했지만, 정확히 이해하기 위해서는 많은 분야의 지식이 필요하기 때문이다. 현실적으로 단기간에 이러한 모든 지식을 습득하는 것이 힘들고, 장기간에 걸쳐서 경험의 축적이 생기면서 이해가 가능한 부분이 있기 때문이다. 학습이 “Y=F(X)”로 주어지는 함수라면, “X”가 투입되는 노력이라고 하고, “Y”가 그 학습의 결과라고 할 때, “X”가 상당 부분 투입되더라도 “Y”가 바뀌고 있다는 확신이 없을 수도 있다. 마치, “가우스 함수(Gaussian Function)의 최대 정수 값을 구하는 것”과 같은 학습 상황에서는 변화를 확인하기 전에 투입되는 꾸준한 노력은 마치 보상이 없는 듯이 생각된다. 하지만, 어느 순간 자신이 지금까지 알던 단편적으로 지식들이 한꺼번에 조합되면서 새로운 시각이 열리게 되는 것을 경험하게 될 것이다.

# [ 소프트웨어 플랫폼이 뭐길래? ]

소프트웨어를 개발하는 회사들은 조금 돈을 벌기 시작하면 플랫폼(Platform)이 있어야 한다는 이야기를 한다. 왜 그럴까? 초기에는 사실 제품을 만든다고 플랫폼이란 것을 생각하지 않고 마구 만들기 시작한다. 구조(아키텍처)라는 것이 거의 없다고 봐도 무방하다. 그냥 통으로 하나의 프로그램이다. 물론, 구조를 표현하라고 하면 그럴듯한 그림들을 보여준다. 하지만, 실제 코드를 들여다보게 된다면, 아마 그게 그려진 그림처럼 보이지는 않을 것이다. 이렇게 말하는 이유는 물리적인 것만을 보고 하는 이야기가 아니라, 논리적인 연결 관계를 가지고 말하는 것이다.

이렇게 만들어진 소프트웨어는 시장의 냉정한 평가에서 탈락할 수도 있고 성공적일 수도 있다. 하지만, 시간이 지나면 시장은 변하고 사용자의 요구사항은 끝없는 기능 추가로 이어진다. 한 두 번 이런 일을 하다가 보면 버그는 지속적으로 늘어나고, A/S 문제로 신규 개발보다 기존 출시제품의 보완에만 매달리게 된다. 결국 경쟁사가 추격해 오게 되고, 한발 앞서 갔지만 시장을 만든 후 고스란히 다른 경쟁자에게 내주고 만다. 소프트웨어 플랫폼은 어떤 소프트웨어를 개발하더라도 필요한 것이다. 아래에서는 소프트웨어 플랫폼이 가지는 특징을 몇 가지 나열해 보도록 하겠다.

**01. 소프트웨어 플랫폼은 변경에 대한 통제가 가능하다.**

먼저 소프트웨어가 계층구조(Layered Architecture)로 되어있다고 가정한다면, 변경에는 크게 세 가지가 있을 수 있다. 첫 번째는 하위로부터의 변경이다. 즉, 하드웨어의 변경이 발생할 경우에 대한 대응이 필요하다. 두 번째는 사용자의 요구사항(즉, 외적 행위의 변경 혹은 추가) 변경이다. 이것은 주로 상위로부터 하위로 전달되는 변경이다. UI(User Interface: 사용자 인터페이스)의 변경도 있을 수 있고, 기능의 추가도 있을 수 있다. 기능의 추가는 아래로 파급력을 가질 수 있으면, 다양한 구성 요소의 상호관계를 변경할 수도 있다. 세 번째 변경은 내부구조의 변경이다. 이것은 개발 내부에서 발생하는 것으로, 주로 성능개선 및 유지보수와 관련된 변경이다. 따라서, 소프트웨어 플랫폼은 이러한 변경 요소들에 대해서 통제할 수 있어야 한다.

**02. 소프트웨어 플랫폼은 조직의 구성을 반영한다. 혹은, 소프트웨어 플래폼이 조직의 구성을 바꾼다.**

둘 중 하나가 이루어지면 소프트웨어 플랫폼을 만들 수 있다고 본다. 즉, 조직의 구성에 따른 구조가 소프트웨어 플랫폼을 설계할 때 반영되거나, 혹은 이상적으로 설계된 소프트웨어 플랫폼을 만들도록 조직을 구성하게 된다. 어쨌든, 소프트웨어 플랫폼도 사람이 만드는 것이기에, 인간의 정신활동 산물이며 동시에 인간의 조직화를 보여주는 도식이기도 하다. 최상부 계층에서 작업하는 인력들은 고객(내부 혹은 외부)과의 접촉이 많을 것이고, 아래 계층으로 점차 사람과의 대화는 줄어든다.

**03. 소프트웨어 플랫폼은 만들 당시에 쓰이는 하드웨어 보다는, 만든 후에 사용될 하드웨어의 성능에 맞춰서는 안 된다.**

이 말은 최소한 이전에 만들었던 하드웨어를 무시해서는 안 된다는 것이다. 흔히, 소프트웨어 플랫폼은 구조상 기존의 제품에 비해서 상대적으로 오버헤드를 가질 수밖에 없다는 말을 하고 있지만, 정작 그것을 가지고 사용해야 할 사람들은 예전보다 성능이 안 좋아졌다고 불평할 것이기 때문이다. 물론, 초기 플랫폼의 정착이 어려운 것은 사실이지만, 그렇다고 플랫폼 사용자들의 불평을 무시하는 것도 좋은 해결책은 아니다.

이럴 경우에는 “Basic/Standard/Pro” 버전과 같은 것을 사용해서, 기능과 성능의 차이점을 잘 이해시켜야 할 것이다. 결국은 새로운 하드웨어에서 사용될 것이지만, 설득해야 할 사람들은 외부에 있는 고객만이 아니라, 내부에 있는 개발자들도 포함되기 때문이다. 외부 고객은 만들어진 제품에 어떤 소프트웨어 플랫폼이 들어있는지 관심을 가지지 않는다.

소프트웨어 플랫폼은 다양하게 분류할 수 있겠지만, 대략 두 가지 정도로 크게 나눌 수 있을 것이다. 주로 상품화를 담당하는 조직의 관점에서 보면, 개발된 전체 소프트웨어를 플랫폼이라고 생각하는 경향이 강하며, 이때 자신들이 만든 모든 소프트웨어를 합쳐서 플랫폼이라고 불리기를 원한다. 주로, 크기가 작고(100만 라인 이하?) 제한적인 목적(혹은, 특정한 목적으로 만)으로 사용될 소프트웨어를 만드는 경우가 여기에 속한다. 펌웨어(Firmware) 같은 것이 이 같은 부류에 적합하다.

응용 프로그램에 서비스를 제공한다는 생각으로 만드는 플랫폼은 응용 프로그램과 어떻게 플랫폼 코드를 분리할 것인가를 고민해서 개발된다. 응용 프로그램에서 공통으로 사용되는 기능들을 플랫폼에 포함되기를 원하며, 응용 프로그램의 확장성으로 인해 범용적으로 사용되는 소프트웨어를 만들 경우가 여기에 속한다. 물론, 코드의 양은 대체로 많이 증가하는 것이 일반적이다. 다양한 응용 프로그램에 제공할 서비스를 만족시키기 위해서, 특정 응용 프로그램에서 사용되지 않는 코드도 포함할 수 있다.

소프트웨어 플랫폼을 만드는 방식은 기존에 이미 존재하는 코드를 개선하는 방법과 완전히 새로운 코드를 작성하는 경우로 나누어 볼 수 있다. 물론, 그 중간에 기존의 코드를 이용하고, 미구현된 부분에 대해서 새롭게 구현하는 방법도 있을 것이다. 어쨌든 완전히 새로 짜는 경우는 드물다. 10에 9은 이전에 이미 존재하던 코드를 이용한 재구성에 가까울 것이다. 이때 우리는 생각보다 많은 도구를 이전의 개발에서 얻을 수 있다.

가장 중요한 것은 만들어진 플랫폼이 잘 구현되고 있는지 확인하는 방법이다. 이것은 과제의 모호성을 없애고, 효율적으로 구현을 진행하기 위해서 반드시 필요하다. 다 짜고 나서 검증하겠다는 생각은 버려야 할 것이다. 작게 바꾸고 나서 해볼 수 있는 테스트는 다 해야 한다. 물론, 그렇다고 모든 테스트를 모든 변경에 대해서 다 하라는 것은 아니다. 적어도 자동화시킬 수 있는 부분에 대한 테스트는 매일 밤 실행하는 것이 좋을 것이다.

조금 고치고 테스트를 많이 수행하기 위해서는 자동화된 테스트가 필수다. 그렇지 않다면, 소프트웨어 플랫폼은 영원히 해결하지 못할 숙제가 될 가능성이 높다. 대부분의 실패도 여기서 발생한다. TDD(Test Driven Development)와 같은 개발 방법도 미구현된 부분에 대해서 활용해 볼 수 있을 것이다. 하지만, 모듈 간 의존성(Coupling)이 높을 경우에는 이것도 쉽지 않다. 이때는 차라리, 모듈 간의 역할 정의에 기초해서 코드의 구조를 크게 변경하는 일을 먼저 해야 한다. 이 때도 역시 테스트는 필요하다.

가장 쉬운 것은 사용자 입력에 기반을 둔 UI를 위주로 한 기능 검증 방법이다. 즉, 사용자 UI 시나리오를 만들고, 이를 자동화시켜서 조금 고친 코드를 자동으로 테스트하는 것이 효과적일 것이다. 하지만, 실제로 사용자 UI와 함께 동작하는 경우는 수동적인 테스트를 할 수밖에 없는 경우도 있다. 웹 응용 프로그램을 위한 플랫폼이라면 자동화 도구를 구할 수 있는 방법들이 있지만, 일반적인 PC용 응용 프로그램은 사용자 UI를 자동으로 동작시키는 것이 쉬운 일이 아니다.

물론, 모든 코드를 테스트하는 것은 어렵고, 심지어 자동화된 테스트를 제대로 하지 못한 경우도 있다. 어쨌든, 효과적인 도구를 찾아서 이러한 부분에 대한 테스트 전략을 수립하지 않는다면, 동작하는 코드를 고치는 것은 결코 쉬운 일이 아니다. 동작하는 코드는 동작하기 전보다 훨씬 변화에 대한 거부감을 키울 수 있다. 대부분의 개발자는 절대 문제를 만드는 것을 좋아하지 않기 때문이다. 즉, 기술적인 것이 문제가 아니라 사람이 문제가 되는 경우가 플랫폼 개발에서는 종종 있는 일이다.

소프트웨어 플랫폼은 역할과 책임이 얼마나 잘 드러내는가를 묻는다. 플랫폼을 구성하는 요소들은 고유한 자신 만의 역할을 가져야 하기 때문이다. 물론, 역할은 책임을 동반하고, 이러한 책임은 계약(Contract)을 통해서 다른 모듈이나 구성 요소에 알려지게 된다. 역할은 상위에서 하위로 갈수록 구체적으로 변하게 되며, 필요없는 확장성은 배제되어야 할 것이다. 변경될 수 있는 부분(물론, 언젠가는 모두 변경이 되겠지만)과 변경되면 안 되는 부분으로 나누어지고, 변경될 수 있는 부분에 대한 확장 방법이 만들어지게 될 것이다.

디자인 패턴(Design Pattern)과 같은 것을 배워두는 것도 플랫폼을 만드는 데 도움이 될 것이다. 하지만, 모든 결정은 가장 기본적인 원칙에 충실하면 된다. 필요없는 확장성이나 기능으로 인해 코드를 복잡하게 만들고 있는지 확인해야 할 것이다. 소프트웨어를 만드는 기본적인 원칙은 의존성의 제거(Decoupling)와 응집성(Cohesion)을 높이는 것이다. 따라서, 관련이 없는 코드가 묶여져 있는지 확인하는 것과 중복을 제거하는 것이 기본적으로 지켜야 할 규칙이다.

나무와 땅을 연결해 주는 뿌리는 한 가지에서 나오지만, 뿌리가 깊게 내리기 위해서는 자신만의 역할을 충실하게 해주는 것들로 가지런하게 뻗어 나가는 것을 발견할 수 있을 것이다. 같은 방법으로 응용 프로그램의 입장에서 봤을 때, 코드가 마치 나무의 뿌리와 같이 고르게 뻣어나가고 있다면, 일단은 성공적인 코드라고 생각해도 무리는 없다. 뒤집에서 나무의 가지가 하늘을 향해서 뻣어나가는 것도 튼튼한 코드 기반을 갖추고 있다고 생각해도 된다.

소프트웨어 플랫폼은 성능도 충분히 만족할 수 있도록 돕는다. 전체적인 성능을 개선하는 작업은 작은 부분에서 큰 부분까지 끝없는 반복적인 노력을 동반한다. 잘못된 구조를 고치는 것은 큰 줄기를 새로 만드는 것으로 전체적인 변경이 크게 발생할 가능성이 높다. 작은 가지를 잘라내서 뻗어 나갈 방향을 잘 정해주는 정원사와 같이, 필요 없는 코드는 제거하는 것도 좋다. 나중에 사용될 것이라고 생각한 코드는 절대 사용될 일이 없을지도 모른다. 따라서, 만약 나중에 요구된다면 구현도 나중에 하는 것이 좋을 것이다(물론, 변경에 대한 확장은 항상 마음 속에 담아 두어야겠지만).

흔히 소프트웨어 플랫폼을 만들면 오버헤드가 커져서 결국 소프트웨어를 사용하지 못한다고 생각하는 경우가 많다. 하지만, 구조를 개선하고, 필요 없는 코드를 제거하다 보면, 결국은 더 효율적인 시스템을 구축하기 마련이다. 개발에서 생각하지 못하거나 시간이 없어서 해결하지 못한 것들은 하나씩 해결하면, 결국에는 잡을 수 있는 것이 성능적인 부분이다. 시스템의 가장 약한 고리는 전체 시스템의 가장 약한 부분이 되기에, 그런 부분들을 골라서 고칠 수 있다면 전체적인 시스템은 안정적으로 변화되기 마련이다.

어떻게 그러한 것들을 찾을 것인가를 검토한다면, 다음과 같은 부분을 보기 바란다. 첫 째는 중복되는 코드가 없는지를 보라. 그것을 제거하는 것이 특별히 중요하지 않을 것 같지만, 약한 고리 중의 하나를 발견할 수 있을 것이다. 둘 째는 복잡한 코드가 없는지를 보라. 복잡한 코드는 이해도 어렵고 고치기도 어렵다. 당연히 오류가 발생할 가능성이 높고, 성능도 제대로 발휘되지 않을 가능성이 높다. 세 번째로는 지나치게 긴 코드를 보라. 길게 짰다는 말은 이해하지 못했다는 말과 같고, 여러 가지 일을 한꺼번에 하고 있다는 것이다. 따라서, 이런 부분을 집중적으로 관리하다 보면, 저절로 성능은 개선될 것이다. 각종 다양한 프로그래밍 팁(Tip)이 있지만, 그런 것들은 경험에서 얻기 바란다. 팁도다는 항상 원리가 우선이며, 고치다 보면 스스로 알게 될 것이다.

소프트웨어 플랫폼을 어떻게 만들 수 있을까? 크게 보면 예전에 있던 코드를 재사용해서 만드는 것과 완전히 새롭게 만들어가는 두 가지 방법이 있을 것이다. 자신이 속한 조직에 얼마나 많은 자원(resource)이 있는지와 소프트웨어 플랫폼을 만들었을 때, 얼마나 효과가 있을지를 잘 생각해서 결정해야 할 문제다. 만약, 거의 변화되는 부분이 없고, 시간이 흘러도 추가되는 기능이 없다면, 소프트웨어 플랫폼을 새롭게 만드는 것은 무의미하다. 괜한 돈만 잡아먹을 뿐이다. 하지만, 매번 사용할 때마다 변경이 자주 일어나고, 기능 추가나 수정이 빈번한 경우에는 플랫폼을 생각해 보는 것이 좋을 것이다. 물론, 아주 적은 분량의 코드를 가지는 시스템을 플랫폼으로 만드는 것도 그렇게 큰 이득은 없다. 이럴 경우에는 적절히 라이브러리(Library)화 수준에서 변경에 필요한 부분들만 조금 정리하는 수준에서 만족하는 것이 좋을 것이다.

기존에 이미 사용하는 코드가 있는 상황에서의 플랫폼을 만드는 것은 쉽지 않은 노력이 든다. 왜냐하면, 기존 코드를 이해하는 것과 함께, 조금씩 수정을 가하면서 원하는 구조가 나올 때까지 반복적으로 테스트를 해야 하기 때문이다. 한꺼번에 잘 모르는 코드를 많이 고치면 당연히 버그를 수정한다고 대부분의 시간을 보낸다. 이 때는 한 번에 하나만 고친다는 것을 원칙으로 가져가도록 한다. 즉, 작게 고치고 자주 테스트하는 방법이다.

고쳤을 경우 어떻게 테스트할 것인가를 고민해야 하는데, 이때는 리팩토링 개념으로 접근해서 TDD와 같은 단위 테스트 방법을 생각해야 할 것이다. 물론 테스트는 자동화되어 있어야 한다. 자동화되지 않은 테스트를 하고, 조금씩 고치는 것은 인간이 잘할 수 있는 부분을 과감히 포기하고, 기계가 잘하는 일을 사람이 선택하는 것이기 때문이다. 또한, 이때는 정적분석 툴과 같은 것도 큰 도움이 될 수 있다. 단계는 다음과 같이 진행된다.

Step 1. 역할과 책임의 확인

각 모듈(혹은, 모듈이라고 생각하는 것들)을 논리적으로 생각했을 때 연관성이 있는 부분들로 크게 구분한다. 구분된 것들에게 이름을 부여하고, 각각을 적절한 디렉토리로 묶어 준다. 물론, 컴파일러 스크립트(compile script)를 수정해서 한 번에 빌드(build)를 할 수 있도록 한다. 이때 나오는 오류는 주로 컴파일러에서 발생하기에 수정도 쉽게 할 수 있다.

Step 2. 역할과 책임의 강제

정적분석 툴과 같은 것을 가지고, 상호 참조나 계층간의 역참조(Reverse Reference)가 발생하는지 확인한다. 그런 툴이 없을 때는 생각보다 오래 걸리는 작업이므로 사람의 공수가 더 들어갈 수 있다. 상호 참조나 역참조가 발생한다면, 이를 해결하기 위한 리팩토링을 실시한다. 이름의 변경에서부터 추상화, 인터페이스(Interface) 등 다양한 방법을 통해서 참조를 적절한 위치로 다시 옮겨주도록 한다. 필요하다면, 헤더 파일과 같이 타입을 선언하는 부분을 분리해서, 세밀한 정의로 바꾸는 것도 필요하다. 이 부분은 생각보다 오래 걸리는 일로, 경우에 따라 전체적인 코드의 수정이 발생할 수 있다. 이때 나오는 오류는 주로 사용자 시나리오상의 오류이므로, 사용자 시나리오 기반의 테스트 수준에서 해결하도록 한다. 자동화된 통합 테스트 스크립트(Integration Test Script)가 있다면 많은 도움을 받을 수 있을 것이다.

Step 3. 내부 구성의 변경

위의 과정에서 대체로 상하좌우로 대략적인 모듈화는 완료가 되었다. 이제는 이렇게 나누어진 내부 구조를 다듬는 단계이다. 코드의 복잡도(Complexity)를 낮추고, 중복된 코드를 제거하도록 한다. 이 단계에서 필요한 것은 단위 테스트를 실행하는 것이다. 즉, 각각의 내부 구성 모듈에 대해서 제대로 구조가 변경되었는지 확인하는 절차다. 이 단계에서 발생하는 오류는 주로, 구현 오류이므로 단위 테스트 프레임워크와 같은 툴을 사용해서, 먼저 테스트를 고려해서 코드를 수정하는 활동을 진행하는 것이 좋을 것이다.

Step 4. 통신의 표준화

이제는 각각의 모듈, 계층 간에 필요한 인터페이스를 정하는 일이다. 위에서 만들어진 코드들은 아직 잘 정의된 인터페이스와는 좀 거리가 있기에, 이를 문서화시키기 전에 적절하게 정리할 필요가 있다. 내부 구조를 접근하는 외부 호출을 차단하고, 정의된 인터페이스 만을 사용할 수 있도록 접근 제약을 설정한다. 여기 진행되면 이제 대부분 과정은 마친 것이다. 여기서 만약 부족하다고 판단되면, Step 1로 돌아가서 Step 4까지 반복적으로 수행하면 된다.

위와 같은 플랫폼을 개발하는 활동 과정에서 가장 필요한 것은 개발 인프라(Infra)다. 코드 리뷰, 빌드, 테스트 등등 자동화할 수 있는 것은 모두 다 자동화를 하는 것이 좋다. 개발 주기(테스트 간격)를 빨리 가져가면, 조금이라도 더 좋은 품질의 소프트웨어를 만들 가능성이 커지기 때문이다. 대략 1일 단위 정도의 주기를 가져가는 것이 이상적이다. 개발자 개인은 그것보다 더 빠른 테스트 주기를 가지는 것이 필요하지만, 전체적인 팀의 균형을 위해서는 1일 단위(퇴근 시간을 생각해서)로 모든 활동이 완료될 수 있도록 유지해야 한다. 즉, 그 날 변경이나 추가된 코드는 퇴근 전에 통합해서 밤에 자동으로 테스트되도록 해야한다.

기존에 아무런 레거시 코드(Legacy code)가 없는 경우에는, 앞에서 본 Step 1에서 Step 4의 과정은 나중으로 미루어 두어도 된다(완전히 필요가 없다는 것이 아니라, 개발된 소프트웨어 플랫폼을 지속적해서 개선하는 데 필요하다). 이런 상황에서 소프트웨어 플랫폼을 개발하기 위해서는 아래와 같은 전략을 취하도록 하자.

**Step 1. 요구사항들을 수집한다.**

수집된 요구사항을 어떻게 테스트할 것인가에 대한 자료를 수집한다. 즉, 테스트 케이스를 먼저 설계하도록 한다. 이 때는 모듈 단위의 단위 테스트를 위한 테스트 케이스라기보다, 사용자 시나리오 기반의 테스트 케이스가 될 것이다. 수집된 요구사항 별로 대분류, 중분류, 소분류, 미세 분류와 같이 좀 더 작게 구분할 수 있는 만큼 최대한 구분한다. 물론, 테스트 케이스가 많으면 많을수록 좋지만, 실제로 발생하지 않는 케이스는 솎아내야 한다. 이를 위해서 실험 계획법(DOE: Design of Experiment)과 같은 것을 사용해도 좋을 것이다.

**Step 2. 앞에서의 나눈 분류의 기준으로 Architecture를 설계한다.**

필요한 것은 수집된 요구사항들 간의 연관 관계를 세밀히 분석하고, 공통으로 사용할 수 있는 것들이 없는지 분석한다. 공통적으로 나오는 부분에 대한 것들을 대분류, 중분류, 소분류와 같은 식으로 전개하면서 큰 그림을 먼저 그린다. 최상위에는 사용자를 기준으로 한 대분류가 도출되고, 최하위에는 구현되는 시스템이 동작하는 하드웨어나 운영체제, 혹은 프레임워크가 나올 것이다. 이들 간의 관계를 화살표로 엮도록 한다. 이때, 계층 간의 구분을 위해서 각각의 계층에서 하는 역할을 기준으로 응용 프로그램, 서비스, 미들웨어(Middleware), 시스템 소프트웨어등으로 구분해서 작성하면, 좀 더 명확한 계층별 책임을 명시할 수 있을 것이다.

**Step 3. 앞에서 나온 결과물을 바탕으로 이제는 구현해야 할 것과 다른 곳에서 얻을 수 있는 것들에는 무엇이 있는지 조사하도록 한다.**

될 수 있으면 이용할 수 있는 코드는 그대로 재사용하는 것이 좋다. 예를 들어, DB(DataBase)와 같은 것을 새롭게 구현할 필요는 없으며, 메시지 전송 구현을 위한 DDS(Distributed Data Service)나 JMS(Java Messaging Service)와 같은 것은 그대로 사용해도 무방하다. 물론, 이러한 것들도 래퍼(Wrapper)와 같은 것을 정의해서, 다른 동일한 역할을 하는 모듈로 변경할 수 있는 여지는 남겨두도록 한다. 이 단계까지 했으면 어떤 것들을 구현해야 할 지가 정의된다.

**Step 4. 각각의 모듈에 대한 요구사항을 정리한다**.

앞에서 분류한 요구사항을 기준으로 사용자 시나리오의 바탕에서 필요한 구성 요소의 인터페이스를 정하는 게 목적이다. 처음에는 일단 입력과 출력만 맞추도록 한다. 팀은 이것을 기준으로 구현을 시작할 수 있다. 이 부분에서 여러 번의 논의가 필요하며, 될 수 있으면 데이터를 주고 일을 시키기는 방식보다는, 데이터의 흐름을 제한하고 일을 요청하는 방향으로 하는 것이 좋다. 어쨌든 데이터의 무리한 이동보다는 데이터를 가지고 있는 모듈에 요청하는 취하는 방식이 좋을 것이다. 그래야만 데이터가 사용되는 곳에 모이게 된다.

**Step 5. 구현한다.**

이 때 필요한 것은 구현 오류를 최대한 빨리 찾아줄 수 있는 적절한 테스트 케이스와 이를 이용해서 빠르게 테스트할 수 있는 방법이다. 따라서, TDD(Test Driven Development)와 같은 방법론이 필요하다. 자신이 만든 모듈에 대해서 테스트 범위(Test Coverage)가 100%가 될 수 있도록 테스트 케이스를 보완하면서, 정해진 인터페이스 규약을 만족하도록 코딩한다. 코드는 일 단위로 빌드(Daily Build)가 될 수 있어야 하며, 만들어진 모든 단위 테스트는 자동으로 실행해서 다음 날 결과를 개발자에게 통지할 수 있어야 한다.

계층별로 필요한 테스트 스크립트는 계층별로 존재하는 테스트 케이스를 응용해서 통합 테스트를 실행할 수도 있을 것이다. 어쨌든 중요한 것은 될 수 있으면 모든 테스트를 자동으로 할 수 있도록 만들고, 사소한 변화도 다 검증할 수 있도록 만드는 것이 핵심이다. 물론, 이런 것이 힘들다면, 가능한 중요하다고 생각되는 모듈에 한정해서 실시하도록 한다.

**Step 6. 지속적해서 개선한다.**

Step 1에서 Step 5까지를 지속해서 반복한다. Step 1에서 Step 5까지 모든 요구사항에 대해서 다 하기는 어렵다. 왜냐하면, 요구사항이 개발 초기에 전부 정해져서 시작되는 과제는 국방 쪽 과제나 대규모 SI(System Integration)업 정도 밖에 없을 것이다. 따라서, 이럴 경우에는 아는 요구사항을 최대한 우선 순위별로 차근차근 Step 1에서 Step 5까지를 반복하는 것이다. 이렇게 해서 만들어진 결과물은 다시 유지보수를 지속해서 해야 한다. 유지 보수 단계는 앞에서 이미 레거시 코드 를 가지고 진행했던 과정과 동일하다. 차이라고 한다면, 이제는 좀 더 품질이 좋은 코드를 가지고 반복한다는 것뿐이다.

이번에는 소프트웨어 플랫폼을 만드는 실제적인 과정에 대해서 조금 더 깊숙히 알아봤다. 가능하다면 최대한 검증을 빨리 하는 것이 제일 좋다. 조금이라도 변경이 있으면 즉시 검증하라. "난 이 모듈을 다 만들고 검증하겠어"라고 생각하는 순간, 내가 만든 버그가 어떤 영향을 줄지 예측하지 못하게 된다. 오늘 버그가 내일 어떻게 바뀔지 누가 장담하겠는가? 모든 소프트웨어는 항상 베타 버전(Beta Version)이라는 말이 있다. 베타 버전에서 출시 후보(Release Candidate)가 되기 까지는 수없이 많은 노력이 필요하며, 이러한 노력을 줄기이 위해서 조금이라도 초기에 더 많이 테스트한다면, 뒤로 갈수록 퇴근 시간이 늦어지는 것도 미리 방지할 수 있을 것이다(물론 그렇다고 절대 야근이 없어지지는 않겠지만).

플랫폼을 구축하기 위해서는 어떤 것들이 필요할까? 기본적으로 이것을 가능하게 하기 위해서는 “여분의 자원”이 필요하다. 즉, “당장 급한 것을 하는 대신에 중요한 것을 할 자원”이 있어야 한다. 이러한 것이 없는 상황에서는 불가능한 것이 플랫폼을 구축하는 일이다. 최소의 인력으로 최대의 효과를 거두고 싶어 하는 것이 모든 기업의 추구하는 바이지만, 소프트웨어 개발에 있어서 품질은 시간 및 사용 가능한 자원에 비례하고, 기능이 많아질수록 버그는 늘어난다(이것을 철의 삼각형[Iron Triangle]이라고 부른다). 기능을 줄인다면 그나마 품질은 좋아질 것이다. 하지만, 대부분의 경우에는 이것도 하고 저것도 가능해야 한다는 것이 주어진 지상 최고의 목표인 경우가 많다.

이와 같은 상황에서 소프트웨어 Plattorm 개발팀이 취할 수 있는 최고의 선택은 기능에 대해서 마일스톤(Milestone)을 제시하는 수밖에 없다. 즉, “언제까지는 이러한 기능들이 가능하고, 다음 버전에서는 어떤 기능들이 추가될 것이며, 그 다음 버전은...” 이런 식으로 과제를 잡을 수밖에 없다. 그나마도 용납이 되지 않는다면, 그냥 죽어라고 일하다 장렬하게 전사할 뿐이다. 성능도 안되고 품질도 안되는 플랫폼은 불신만 낳게 되며, 만들어도 쓰지 않는 플랫폼은 그 동안의 노력을 의미 없게 만들 뿐이다.

플랫폼을 만들기 위해서는 상품화 개발팀의 적극적인 도움이 가장 절실하다. 즉, 플랫폼에 투입되는 여분의 인력이 상품화 개발팀에서 왔으면 좋겠지만, 그렇지 않고 새롭게 구성된 팀일 경우에는 어쩌면 이 부분이 가장 중요할 것이다. 즉, 플랫폼이 적용될 제품의 특성을 알지 못한다면, 플랫폼을 만들어도 쓰지 않을 가능성이 높다. 제품의 기능에 대한 명세(Spec.)를 이해해야 하며, 만들어진 플랫폼이 만족해야 할 품질 특성(Quality Attribute)도 알아야 하며, 이러한 것이 반영된 소프트웨어 아키텍처도 설계해야 한다.

만약, 기존의 코드가 이미 있다면 해당 코드에 대한 이해도 절실히 필요하다. 대부분 소프트웨어 개발자라면 한번은 겪어 봤겠지만, 주석이 잘못되었거나 아예 없는 남의 코드를 이해하는 것은 정말 어려운 일이다. 주석이 없이 잘 짜인 코드를 만나는 것은 모래 밭에서 바늘 찾는 것과 같이 드문 일이라는 것은 익히 잘 알고 있을 것이다. 이런 상황에서 이해를 도와주는 문서라도 있다면 그나마 많은 도움이 되겠지만, 사실상 앞에 것들이 제대로 안되어 있는데 문서라고 있겠는가? 전달받은 “코드에 녹아있는 생각”을 정리하라고 맡기는 것은, 무더운 여름날을 더 짜증스럽게 만드는 일이다. 가장 큰 도움은 상품화 개발팀과 함께 일할 수 있는 “책임 있는 사람의 결정”이다.

플랫폼을 설계하는 일은 무엇일까? 소프트웨어의 설계는 특정 시스템의 구성 요소가 가지는 “역할과 책임의 범위를 정하는 것”이다. 또한, 그렇게 “정의된 요소들 간의 관계를 설정하는 것”이다. 최대한 간략한 구조를 추구해서 필요없는 것은 만들지 않고, 요구사항을 만족할 수 있는 다른 대안은 없는지를 검토해서, 적절한 해결책을 찾아 나가는 과정이다. 이때 요구되는 것이 바로 “한 가지 일만 하라고 말하는 응집성(Cohesion)”과 “최소의 의존성(de-coupling)을 가지라는 것”이다.

의존성을 개선하는 방법은 호출의 수를 줄이는 것이 가장 기본이며, 내부의 데이터 구조에 대해서 신경 쓰지 않고도 일을 처리할 수 있도록 만드는 것이다. 즉, 제어(Control) 및 데이터의 의존을 최소화하는 일이다. 이를 적극적으로 하다 보면 결국에는 전체 시스템에서 발생하는 버그의 수를 최소화할 수 있으며, 나중에 특정 부분을 수정하더라도 부수적인 오류(Side-Effect Bug)가 발생하지 않도록 만들 수 있을 것이다.

설계에서 모든 것을 검증하기는 어려우며, 개발자라면 나름대로 자신이 코딩한 부분에 대해서는 반드시 검증을 제대로 할 방법을 갖추어야 한다. 아무리 완벽한 설계라고 하더라도 구현에서 잘못을 저지를 경우에는 설계 자체를 의심스럽게 만든다. 따라서, 설계를 제대로 했다고 가정한다면, 그것을 구현한 내용도 검증되어야 한다. 설계는 당연히 주어진 요구사항을 만족시키는 것이며, 요구사항은 기능과 비기능 부분이 만족되었을 때 검증되었다고 이야기한다. 물론, 설계에서 바로 코드로 전환되기는 어렵기에, 설계 리뷰를 통해서 구현 전에 미리 머리속으로라도 검증하는 방법을 취하기도 한다. 혹은, 간략한 프로토타입을 통해서 설계가 올바른 것인지를 확인하는 것이 필요할 때도 있다. 예광탄(Tracer Bullets) 개발과 같은 것도 한 가지 방법이라고 볼 수 있다.

"소프트웨어의 구조는 조직을 반영한다."라는 말이 있다. 아주 당연한 말이지만, 사람이 하는 일이라 당연히 사람 사이의 관계를 반영하는 것이 바로 플랫폼의 구조이다. 흔히들 구조를 소프트웨어 아키텍처로 이야기하는데, 어쨌든 중요한 것은 그것을 만들어가는 사람을 무시하면 어떤 것도 실패할 수밖에 없다는 사실이다.

플랫폼을 만드는 데 있어서 가장 힘든 점은 개발하는 소프트웨어 자체에 있는 것이 아니라, 이것을 사용하게 될 “이해 관계자”들의 생각이라는 점이 아이러니하다. 실제로 소프트웨어 플랫폼을 도입하고자 했을 때 가장 많은 반발을 보이는 사람들이, 바로 그것을 가져다가 상용화에 적용해야 할 사람들이다. 자신들이 직접 짜지 않은 코드에 대해서 신뢰가 가지 않는다고 다양한 핑계를 가지고 덤벼든다.

그렇다면, 플랫폼을 가장 쉽게 정착시키는 방법은 무엇일까? 크게 두 가지의 방법이 있을 수 있다. 각각은 아래로 부터의 필요에 의한 플랫폼의 개발과 위로 부터의 강압에 의한 플랫폼 개발이다. 당연히 첫 번째 것이 플랫폼을 정착시키는 데는 가장 효과적이지만, 불행하게도 현실에서는 두 번째 경우를 맞닥뜨리는 경우가 많다. 아래로 부터의 필요에 의한 개발은 개발자의 동참을 쉽게 끌어내며, 윗 사람의 최소 지원만으로도 쉽게 갈 수 있는 길이 열린다. 그렇다고, 윗 사람에 대한 설득과 개발에 전혀 문제가 없다는 것은 아니다. 하지만, 개발자들 사이에서 맞닥뜨리는 불편한 관계는 거의 없기에 적용에 큰 어려움은 없을 것이다.

위로 부터의 강압에 의한 플랫폼 개발과 적용은 심각한 실무자의 거부에 봉착하는 사례가 많다. 기존의 개발자들은 이미 자신이 개발한 코드가 상품화를 통해서 동작하는데, 왜 굳이 새로운 플랫폼을 적용해야 하는지 이해하지 못하며, 플랫폼을 개발하는 중간에 개선이라는 이름으로 자신들의 기존 코드가 가진 문제점들을 다 까발릴 위험이 있기 때문이다. 즉, 자신들이 열심히 돈을 벌어다 주었는데, 어디서 나타났는지도 모르는 놈들이 와서 온갖 기존 코드에 대한 나쁜 점들을 들쑤셔 자신들의 코드가 더 좋다고 이야기한다면, 기분 좋아할 사람은 아무도 없을 것이기 때문이다.

한 가지 확실한 것은 회사의 과제는 개인의 기분으로 하는 것이 아니라, 효과적인 분석과 시스템적(Systematic)으로 한다는 것이다. 기존의 어떤 것이 좋지 않다는 것을 알게 된다면, 적극적으로 변화에 동참해서 좀 더 자신을 발전시키는 것이 오히려 더 나은 미래를 만들 수 있지 않을까? 현재 발생하는 문제점을 해결할 방안이 남의 손에 있다면, 오히려 그런 사람들에게 적극적으로 손을 내미는 것이 현명한 선택일 것이다. 물론, 그렇다고 거부감이 완전히 사라지는 것은 아니다. 그렇다고 감정만으로 자신과 다르다는 것만 강조하는 것도 올바른 태도는 아닌 것이다.

소프트웨어 플랫폼은 이미 동작하는 코드를 무시할 수 없다. 완전히 새로 개발하기에는 자원과 시간이 항상 충분하지 않기 때문이다. 주어진 목표를 이루기 위해서 이미 동작하는 코드가 있다면 그것을 잘 활용할 수 있으면 된다. 기존의 코드를 분석해서 어떤 부분에 약점을 가지고 있는지를 알게 되면, 그것에 대한 처방을 내리면 된다. 기존의 코드가 하나도 없다면 이제부터는 새로 설계하고, 각각의 모듈에 대한 역할과 책임을 잘게 나누는 과정의 연속이 될 것이다. 기존의 코드가 구조화가 미진하다면 서로 간의 의존성을 약하게 만들어줘야 하며, 역할이 중복되거나 섞여 들어가 있다면 분리해 주면 될 것이다. 이러한 과정을 반복하다 보면, 대략적으로 어떤 구조를 가지는 것이 바람직한 지를 파악할 수 있을 것이며, 일차적인 목표는 그렇게 만들어진 바람직한 결과물의 형태를 정의하고 만들어가면 될 것이다.

나아가 더 좋은 플랫폼을 만들기 위해서는, 약간 근본적인 개념에 대해서 하나씩 정의를 해 나가야 한다. 성능도 기존보다 최적화할 수 있는 방안을 찾아야 하며, 계층구조 및 복잡도와 중복 코드도 잘 관리해야 할 것이다. 이런 과정의 연속을 통해서 더 좋은 코드가 만들어질 것이며, 테스트를 할 방안도 생겨날 것이다. 소프트웨어는 사람이 만들며, 사람사이의 관계가 원만할수록 더 품질이 좋은 플랫폼을 만들 가능성이 높다. 조직의 성숙도는 결국 사람 사이의 관계가 얼마나 잘 어우러지고 있는가를 표현하는 한가지 방법이며, 그 결과물이 소프트웨어 플랫폼으로 만들어질 것이다.

"세세함 속에 장인의 손길이 머문다"라는 말이 있다. 집을 짓는 장인은 집만 보는 것이 아니라, 햇살이 어디서 드는지, 바람은 어디로 불어와서 어디로 나가는지, 문고리는 어떻게 생겨야 하는지, 가구들은 어디에 배치가 되어야 하는지 등등 많은 생각을 담을 수 있는 설계를 만들어 낸다. 우리가 만드는 모든 것들은 이러한 장인의 손길이 빚어내야지만, 그 자체로의 완벽함을 갖출 수 있게 된다. 코딩에서 필요한 “장인의 손길”은 무엇일까? 같이 한 번 생각해 보도록 하자.

**01. 코딩은 단순히 주어진 목적만을 수행하기 위해서 구현되는 것이 아니다.**

코딩은 주어진 목적(해결해야 할 요구사항)을 수행하기도 해야 하지만 유지보수가 더 중요하다. 코딩을 하다 보면 항상 빠지는 함정은 “성능 만족”이라는 것이며, 성능을 위해서는 모든 (바람직 한) 것을 희생할 수 있다는 태도다. 소프트웨어 아키텍처는 기능 요구사항과 비기능 요구사항을 반영하는 소프트웨어 구조를 정의하며, 이때 필요한 비기능 요구사항의 한 가지가 성능일 뿐이다. 그 외에도 비기능 요구사항에는 유지보수, 보안, 이식성, 변경 가능성 등등 많은 것들이 해당 도메인(Domain)에 맞게 적절히 다루어져야 한다.

대부분의 소프트웨어는 성능을 극한으로 요구하지 않는다. 높은 성능을 요구하는 소프트웨어는 극한의 환경(메모리가 적거나, 사용하는 CPU의 성능이 낮은)에서 동작하지만, 우리가 만드는 대부분의 소프트웨어는 그렇지 않은 경우가 더 많다. 또한, 높은 성능을 요구하는 소프트웨어는 코딩 양이 적어지는 경향이 있지만, 일상에서 사용하는 소프트웨어는 코딩 양이 기하급수적으로 증가하는 경향이 있다(예를 들어, Linux가 대표적).

복잡도가 높은 소프트웨어는 코드 변경과 기능 추가, 개발자의 무의식적인 습관에서 탄생하지만, “그것을 어떻게 유지보수할 것인가?”는 사업의 영속성에 심각한 영향을 줄 수 있다. 복잡한 코드는 버그를 유발할 가능성이 높으며, 기능을 추가하거나 변경하기도 어려울 가능성이 높다. 따라서, 빠르게 변화하는 시장 환경에서 점점 더 살아남기 힘들게 된다. 같은 요구사항을 구현하더라도 코드의 수준(코드 자체의 품질) 차이가 나는 이유도 바로 이것 때문이다.

**02. 코딩은 단순하게 해야 한다.**

"완전하다는 것은 더 추가할 것이 없다는 것이 아니라, 더 빼야 할 것이 없다는 것이다."라는 말이 있다. 사용할지도 모르는 기능을 구현한 코드는 제거되어야 한다. 정확히 이야기하면 아예 만들려고 생각하지도 말아야 한다. 사실 코드에 한번 추가되면 빼는 것이 추가하는 것 보다 더 어렵기 때문이다. 사용할지도 모르는 모든 코드는 그냥 제거하는 것이 가장 좋다. 간혹, 확장을 위해서 필요하다는 말을 하는 사람들이 있지만, 확장은 확장할 때 고려하면 된다.

물론, 이때 고려할 대상에서 계층화(Layering)나 모듈화는 포함하지 않아야 한다. 계층화나 모듈화는 구현의 의존성을 낮추고 응집성을 높여, 유지보수성을 높이는 기술이다. 확장 만을 위해서 필요한 것이 아니라, 소프트웨어의 기본적인 구현 원리라는 것이다.

코딩을 단순하게 하라는 것은 쉽게 생각하면 한 번에 하나만 하라는 말과 동일하다. 대부분의 코딩 문제는 한 번에 여러 가지를 해결하려는 욕심에서 발생한다. 이런 욕심들이 모이게 되면 당연히 하나의 함수, 클래스 등, 프로그램을 구성하는 모든 것이 복잡하게 된다. 단순한 코딩 방법은 디버깅도 단순하게 만들며, 코드의 재활용률을 높이는 기본기가 될 수 있다. 잘 나누어진 임무를 수행하는 모듈들은 그 자체로 완벽하며, 다른 문제의 해결에도 재활용할 수 있는 기반이 된다.

예를 들어, C언어의 기본 라이브러리의 경우 다양한 함수로 구성되어 있지만, 하나의 함수는 한 가지 일만 수행하도록 만들어져 있다. 물론, 때로는 부가적인 효과(Side-Effect)을 가지는 함수들이 있지만, 이러한 함수들은 항상 "주의(Notice)"가 따르는 것을 찾아볼 수 있을 것이다. 즉, 한 번에 하나의 일만 하는 함수, 클래스 등을 잘 정의해서 활용해 나가면, 궁극적으로 개발자가 해야 할 일이 극적으로 줄어들 것은 자명하다.

**03. 내가 만든 코드는 나의 것이 아니다.**

소설을 쓰는 작가들은 자신이 집필한 책으로 펴낼 때, 흔히 "자식이 품을 떠난다."라고 이야기한다. “품에서 떠난 자식”은 세상의 온갖 풍파를 맞으며, 그 자체로 생명력을 지니고 살아가게 되며, 작가와 동일시되지는 않고 비판의 중심에 서게된다. 물론, 가끔씩 책과 저자를 동일시하는 경향도 있지만, “표현의 자유”를 완벽하게 지킨다면 엄격히 구분해서 비판하는 것이 필요하다. 코딩도 비슷한 관점에서 생각한다면, 만들어진 코드와 코드를 작성한 사람을 동일시할 필요는 없을 것이다.

오늘 코딩한 것을 커밋(Commit)하기 전에 될 수 있으면 많은 사람이 볼 수 있도록 코드 리뷰를 요청해야 한다. 그래야만, 전체 시스템을 구현을 위해서 살아남을 가치가 있는 코드가 되기 때문이다. 코드는 아무리 작은 변화라도 전체 시스템에 영향을 주기에, 반드시 작성된 모든 코드는 리뷰 절차를 거쳐야 한다. 물론, 현실적인 시간 부족을 이유로 일부 코드에 대해서는 리뷰를 생략할 수도 있지만, 시스템의 중요한 기능에 관련된 코드는 전부 리뷰하는 것이 원칙이다.

시스템의 전체 품질은 가장 낮은 부분을 따르게 된다. 사용자는 이러한 허점을 과장해서 표현할 수도 있지만, 어쨌든 우리가 만든 시스템이라는 사실은 바뀌지 않는다. 많은 사람이 코드를 리뷰하면 다양한 시각에서 개선점을 찾아낼 수 있으며, 경험이 많고 적음이나 직급의 상하와 관계없이 새롭게 배울 수 있는 것들을 발견하게 된다. 코드 리뷰는 경험과 지식을 공유할 수 있는 중요한 활동이며, “버그를 가장 많이 찾아내는 검증된 방법”이기도 하다. 따라서, 시스템의 전체 품질을 높일 수 있다면, 코드 리뷰를 거부할 정당한 이유가 없다는 것이다.

중요한 것은 항상 이렇게 하는 것이 옳다는 믿음이다. 자신이 만들었다고 자신만의 것으로 생각한다면, 시스템의 오류도 자기의 몫으로 받아들일 준비를 해야 한다. 이런 상태라면 함께 일하고 있는 팀은 존재하지 않는다. 서로 자신의 것을 감추고 남의 것을 얻으려 한다면, 원활하게 돌아가는 팀을 기대해선 안 될 것이다. 단지 팀이라는 이름하에 모여있는 개인들 만이 존재할 뿐이다.

**04. 코드는 반드시 테스트되어야 한다.**

테스트될 수 없는 코드가 있다면 사용되지 않는 코드다(사실 전체 코드의 많은 부분이 테스트되지 않을 수 있다. 하지만, 함수 단위로는 테스트로 실행될 수 있어야 한다). 실행되지 않는 코드가 있다면 지우는 것이 당연하다. 물론, 예외 상황이나 아주 드물게 실행되는 코드도 있다.

그런 코드라고 하더라도 단위 테스트를 사용해서 테스트할 수 없는 것은 아니다. 하드웨어를 직접 접근하는 코드라고 하더라도 더미(Dummy)나 모의 객체(Mock Object)를 이용한 테스트가 가능하다. 특정 하드웨어 주소에 접근해야 한다면, 특정 변수에 대한 접근으로 대체할 수도 있다. 어쨌든 코드는 노력하기에 따라 테스트할 수 있다는 것이다.

현실은 조금 더 암울한데, 대부분 회사에서(아마도 우리나라 회사들의 대부분이 그렇다고 생각되는데)는 테스트를 제대로 못 하는 것이 사실이다. 테스트를 단순하게 생각해서는 안 된다. 계획적으로 체계적으로 해야 한다. 계획은 누가 어떤 부분을, 언제, 어떻게 테스트할 것인지 정하는 것이다. 계획을 과제 초기부터 세우지 않으면, 제대로 된 테스트를 하지 못할 가능성이 높다. 과제 계획에서 테스트 계획을 세우는 과정이 있지만, 대부분은 일정에 대한 것을 이야기할 뿐, 단위 테스트나 통합 테스트와 같은 개발자가 해야하는 테스트는 생략하는 것이 일반적이다.

TDD(Test-Driven Development)와 같은 것을 사용하는 기업이 있다면, 거의 테스트 범위(Test Coverage)가 적어도 80% 이상은 될 것이다. 물론, 이러한 테스트가 거의 자동으로 이루어지기에 소프트웨어 개발자들은 대부분 시간을 개발에 집중하고 있을 것이 자명하다. TDD가 아니더라도 테스트할 수 있는 방법을 찾고, 이를 위한 소프트웨어 아키텍처를 만드는 것은 중요하다. 테스트 주기가 빨라지고 범위가 넓어질수록 시스템은 더 안정되고, 더 빨리 개발될 것이기 때문이다.

개발자가 하는 테스트는 개발자의 입장에서 하는 테스트이기에, 사용자가 가진 문제는 찾지 못할 가능성이 높다. 사용자 관점의 테스트는 개발자의 테스트와 다르며, 당연히 개발에서 분리된 조직이 실행하는 것이 일반적이다. 사용자 관점의 테스트가 분리된 조직에서 실행되지 않으며, 객관적인 테스트 결과를 얻기 어려우며, 개발자의 입장을 대변할 가능성이 높다.

**05. 코드는 역참조를 포함해서는 안 된다.**

이 부분에 대해서는 다양한 의견이 있다. 개발을 진행하게 되면, 편의상 여기 저기에서 정의된 함수와 변수, 헤더 파일 등을 자신의 코드에서 끌어당겨 사용하게 된다. 개발의 편의나 성능적인 문제를 해결하기 위해서 주로 이러한 현상이 발생한다. 사실 개발하는 도중에는 역참조가 발생하는지 제대로 알지 못하는 경우도 있다. 코드를 주기적으로 분석하지 않는다면, 자신이 만든 코드가 어떤 호출 구조를 가지고 있는지 알지 못한다. 문제는 분석의 결과를 나중에 알게되었을 때, 고칠 수 있는 시간이 없다는 것이다. 대부분 결과를 아는 시점이 과제에서 가장 바쁠 때이기 떄문이다.

성능상의 문제가 될 수 있는 부분은 함수가 여러 계층을 거쳐야 하는 경우다. 하위 계층에 정의된 함수를 불러오면 금방 처리할 수 있지만, 역할과 책임으로 나뉘어진 계층과 모듈을 통과해야 하기 때문이다. 계층화된 소프트웨어 구조에서는 대략 3 ~ 5 개 정도의 계층 간의 호출이 하나의 일을 처리하는 데 필요하다면, 역참조가 발생하는 코드의 경우에는 직접적인 호출로 처리될 것이다. 하지만, 3 ~ 5개 정도의 호출이 큰 비용이 될 수 있을까?

사실 이러한 호출의 결과로 발생하는 함수 호출의 오버헤드는 크지 않다. CPU는 함수 호출을 위해서 많은 최적화를 이미 했으며, 컴파일러 수준에서도 함수 호출을 위한 다양한 최적화 방안을 적용하고 있다. 소프트웨어 개발자만 함수 호출을 고민한 것이 아니라, CPU 개발자나 컴파일러 제작자도 구조화된 코드를 지원하기 위한 방법을 오래동안 고민해 왔기 때문이다.

오히려 역참조(Reverse Reference)를 가지는 코드는 성능에서 얻는 이익보다는 유지보수에서 스파게티처럼 꼬이게 될 가능성이 높기에, 지속적인 코드의 추가와 변경 비용이 발생할 것이다. 역참조를 찾아내는 것을 사람이 하기보다는 정적 분석 툴(Static Analysis Tool)을 이용하는 것이 바람직하다. 그리고, 찾아낸 결함을 제거하는 것은 리팩토링(Refactoring) 테크닉을 참고하기 바란다.

많이 살펴보지는 못했지만, 몇 가지 플랫폼을 만들 때 주의해야 할 것들을 나열해 보았다. 코딩은 고도의 정신적인 활동에서 나오는 결과물이기에 “장인의 손길”처럼 세세한 부분을 놓치지 않아야 한다. 중요한 것은 끊임없는 노력에 의한 습관화이며, 자신 만의 철학을 담아내는 그릇을 빚는 일이다.

사업이 어느 정도 수준에 올라 수익이 나기 시작하면, 비용을 줄이고 더 많은 제품을 만들어 사업을 확장하려고 하는 순간이 온다. 이때 가장 큰 문제가 되는 것이 이미 팔려나간 제품에 대한 A/S와 새로운 제품의 기능 추가 비용이 생각보다 크다는 사실이다. 급하게 개발된 소프트웨어는 유지보수가 어려우며, 새로운 기능을 추가하는 것도 개발자에게 많은 시간을 요구한다. 개발자는 "월화수목금금금"을 이어가게 되고, “또 하나의 가족”을 회사에도 만들 것이다.

이런 상황은 다음과 같은 것을 고려하지 못한 데서 발생하는 문제다. 즉, 소프트웨어 요구사항을 기능적인 요구사항과 비기능적인 요구사항으로 나눌 때, 지나치게 비기능적인 요구사항인 "성능"에 치중한 결과다. 비기능적 요구사항에는 "성능"만이 있는 것이 아니라, 확장성, 유지 보수성, 변경 용이성, 보안성, 이식성 등 다양한 것들이 존재한다. 소프트웨어 아키텍처에서 주로 구조적인 부분에 영향을 주는 것이 “비기능 요구사항”들이며, 적절한 대비책을 사전에 갖추지 못하면 개발 속도를 늦추는 결과를 가져온다.

그렇다면, 이 문제를 어떻게 해결할 수 있을까? 일단 가장 먼저 생각해 볼 것은 이식성에 대한 부분이다. 즉, 새로운 하드웨어나 새로운 운영체제, 새로운 개발환경(개발도구나 구매하거나 도입한 라이브러리, 패키지 소프트웨어 등)에서 발생할 수 있는 지나친 의존성 문제는 없는지 확인해야 한다. 만약, 지나치게 특정한 부분에 의존성을 보인다면, 해당하는 부분들에 대해서는 인터페이스를 정의해서 추상화(Abstraction) 할 필요가 있다. 인터페이스를 상속받아서 의존적인 부분이 구현된다면, 변경의 영향도는 인터페이스를 기준으로 나누어지게 된다. 따라서, 변경하는 시간도 상대적으로 짧아질 것이다.

두 번째로 생각해 볼 수 있는 것은 제품 모델(Model)에 대한 분기이다. 즉, 기능의 변경으로 인해서 모델의 변경이 있을 수 있으며, 이러한 부분에 대한 대응이 없다면 점차 코드는 “if()”나 “switch()”와 같은 것들을 통해서 분기하는 일이 빈번하게 될 것이다. 만약, 모델분기를 위해서 이러한 것들을 사용하고 있다면, 한곳에 모아서 없앨 방법을 찾아야 한다(물론, 가장 빠르게 분기할 수 있는 방법이지만, 중복된 “if(), switch()”코드가 전체 코드로 퍼지게 된다). 컴파일러 옵션을 통해서 모델을 분기하는 것도 마찬가지다. 자신의 코드에서 “#ifdef”와 같은 조건부 컴파일이 여기 저기 보인다면, 그 부분도 제거할 수 있는 방법을 찾아야 한다. 제품 모델에 대해서 별도의 처리를 필요로 하는 부분을 인터페이스로 추상화하고, 이를 상속받아 해당 부분을 구현하거나 공백(Blank)으로 선언하면 간단하게 처리할 수 있을 것이다.

컴파일 옵션으로 분기를 해야하는 경우라면, 차라리 파일로 분리해서 제품별로 컴파일 스크립트를 분리하는 것이 좋다. “#ifdef”와 같은 것으로 코드를 지져분하게 만들기 보다, 깔끔하게 디렉토리와 파일로 분리시키는 것이 코드의 유지 보수성을 높여준다. 코드를 작게 만들어 변화의 범위를 한정시킬 수 있게 되는 것이다.

MVC(Model View Control)라는 소프트웨어 구조를 생각해 보자. 이것은 사용자의 인터페이스를 담당하는 계층(View)과 실제적인 데이터의 처리를 담당하는 계층(Model), 그리고, 그 사이에서 중재 역할을 하는 계층(Control)으로 이루어진다. 이 구조는 담당하는 역할과 책임을 명확히 해서 변경의 범위를 제한하는 것이다. 특히, 윈도우 프로그래밍에서 사용자 인터페이스를 로직과 섞어서 UI의 변경이 로직의 변경을 동반하게 될 때 효과적인 해결책을 사용할 수 있다. 웹과 같은 프로그래밍을 할 때도 동일한 “역할과 책임” 구분 방법을 적용할 수 있다.

역할을 사람을 기준으로 하나씩 계층화를 시켜나가면, 프로그램을 짜는 개발자의 생각도 추상적인 것에서 구체적인 것으로 자연스럽게 흘러가게 된다. 이런 것을 방해하는 것은 사람의 잠재적인 의식이며, 흔히 잠들기 전에 끝없이 꼬리를 무는 생각으로 인해서 발생하는 불면증과 비슷하다. 계획에 없던 코드를 만들게 되고, 그 코드들이 서로 엮이게 되면서 풀어야 할 문제를 더 복잡하게 만든다. 따라서, 역할과 책임을 명확히 하고, 서로 분리시켜서 구현하는 것이 복잡한 문제를 다루는 기술인 것이다.

따라서, 각각의 계층에 대한 역할과 책임을 먼저 제대로 정의해야 한다. 계층화는 앞에서 보았던 비기능적인 요구사항을 반영하기 위한 기본 구조이며, 상하로 이웃한 계층 간의 통신 만을 허용해야 한다. 그렇게 함으로써 유지보수 문제는 줄어들 수 있을 것이다(완전히 제거하지는 못함). 지금 만들고 있는 소프트웨어가 있다면, 분석 툴을 이용해서 호출 그래프(Call Graph)를 그려보기 바란다. 혹시, 계층을 무시한 호출이나 참고들이 없는지 점검해 보라. 위반한 개수가 늘어날수록 버그도 늘어나는 경향이 있음을 확인할 수 있을 것이다.

모듈의 테스트는 개발자의 당연한 의무이지만, 대부분 개발자가 제일 하기 힘들어하거나 못하는 분야 중에 하나이다. 전체 프로그램을 통합한 시스템(System) 테스트는 개발자의 몫이 아니라, 테스트 분야를 전공하는 사람들이 해야하는 일이다. 개발자가 해야하는 모듈 테스트가 왜 힘든가를 보면, 일단 개발자는 자신이 어떤 것을 다 만들기 전에는 테스트하고 싶지 않다는 잠재적인 생각이다. 즉, 제대로 동작하지도 않는데 테스트를 해봐야 뻔히 실패할 것이 눈에 보이기 때문이다. 하지만, 이제는 생각을 바꾸어야 할 때다. 일하는 순서상 당연히 실패할 수밖에 없으며, 이를 성공시키는 것이 개발자의 할 일이기 때문이다. 이를 위한 방법론으로는 TDD(Test Driven Development)가 있으니, 자신이 하는 프로젝트에 한 번 적용해 보기 바란다.

TDD를 적용하면 기존의 개발 방법보다 최대 150%의 노력이 더 들지만, 버그의 수는 최대 97%가량 줄어든다는 통계도 있다(정규적인 코드리뷰, 간략한 코드 리뷰, 단위 테스트, 시스템 테스트 등의 다양한 방법을 조합해서). 하지만, TDD를 적용하기에는 생각보다 힘든 경우도 많다. 특히, 다른 모듈과의 상호 동작이 많은 경우에는 상당히 까다로운 일이 될 수 있다. 또한, 이미 작성된 레거시(Legacy) 코드(일반적으로 자동화된 테스트가 없는 코드를 말함)에도 이러한 일을 해야 하는지 의문이 들 것이다. 이 때는 이미 잘 동작하는 코드는 일단 범위에서 제외하는 것이 나름 좋은 해결책이 될 수 있다. 새로운 코드를 작성하거, 기능 변경 및 버그를 수정할 때 테스트도 함께 작성할 수 있도록 하면 된다.

개발 주기를 빨리 가져가는 것은 아주 중요한 일이다. 개발의 주기는 "설계->구현->테스트->디버깅"의 반복이라고 볼 수 있다. 반복을 3개월 단위로 3번을 가져갈 수도 있고, 한 달을 하나의 반복으로, 혹은 극단적으로 줄여서 하루를 반복 단위로 가져갈 수도 있다. 가능한 빨리 자주 테스트를 하는 것이 효과적인 개발 방법이라는 것은 이미 많은 책이나 자료를 통해서 알려져 있는 사실이다. 이때 중요한 것은 테스트에 대한 대비이다. 즉, 다른 부분은 전부 짧게 가져갈 수 있지만, 수정된 코드나 추가된 코드가 제대로 동작하는 것을 어떻게 보장할 것인가가 중요하다.

기능한 자동화된 테스트 스크립트(Test Script)를 이용해서 자주 테스트를 실행해 보는 것이 좋다. 테스트를 실행한 한 후에는 반드시 어떤 코드들이 테스트되었는지 커버리지(Coverage)를 확인해야 하고, 만약 새롭게 작성된 코드가 테스트되지 않는다면, 테스트 케이스를 수정해서 커버리지를 늘려야 할 것이다. 테스트 실행을 위해서 시스템을 구축하는 일도 중요하다. 개발 초기에 이런 비용을 투자하지 않으면, 개발하는 동안 낭비가 발생할 가능성이 높다. 자동화를 위한 노력도 “개발 활동”이라고 생각해야 할 것이다. 즉, 당연히 사람을 배치하고 시간과 노력을 투자해야 한다는 것이다.

요즘은 오픈 소스로 이런 것들을 구축하는데 필요한 대부분의 프로그램을 인터넷에서 내려받아 설치할 수 있다. 단위 테스트 프레임워크와 통합, 자동화된 빌드, 테스트, 테스트를 위한 가상 환경, 배포까지도 전부 자동화가 가능하다. 이러한 시스템을 도입하는데 가장 큰 걸림돌은 경영층이 아니라 개발자 자신이라는 점도 잊지 않아야 할 것이다. 실제로 경영층이나 고위층은 뭐가 필요한지 모르는 경우가 많다. 그들이 개발자로 일하던 시대에는 그런 것들을 보지 못했을 가능성이 높으며, 직접적인 돈에 관련된 것이 아니라고 생각할 때는 일로 보지 않는 경향이 있기 때문이다. 물론, 그렇다고 그 사람들이 무식하거나 개발을 잘 모른다는 것은 아니다. 다만, 보는 관점이 다르다는 뜻일 뿐이다.

소프트웨어 플랫폼은 만드는 것만 중요한 것이 아니라, 유지보수와 테스트성(Testability) 등의 다양한 비기능적인 요구사항을 만족하게 해야 한다. 개발에 진정한 영향을 주는 것은 “만들고 난 다음에 어떻게 할 것인가?”이며, 소프트웨어 생애주기(Life Cycle)동안 투입되는 대부분의 노력은 “유지보수” 비용이 얼마인가에 크게 의존한다. 따라서, 이러한 것들을 만족하게 하지 못하면 플랫폼으로서 제한적인 역할만 할 수 있을 뿐이며, 개발자들은 또 하나의 가족 틈에서 자신의 진정한 가족 사랑이 무엇인지를 확인하게 될 것이다(?).

흔히들 플랫폼을 만들면 다양한 목적으로 재사용이 가능할 것으로 생각하는 사람들이 있다. 특히, 하드웨어를 전공한 사람들의 경우에는 소프트웨어에 대한 막연한 동경인지는 모르겠지만(아니면 잘못된 이해인지는 모르지만), 소프트웨어 변경 비용이 거의 들지 않는다고 생각한다. 하지만, 하드웨어를 변경하는 것이 소프트웨어를 변경하는 비용보다 오히려 싼 경우도 있다.

예를 들어, 수 백만 라인에 달하는 소프트웨어를 새로운 장치(Device)에 넣는다는 것은, 수 백만 줄을 작성한 코드의 검증과 하드웨어에 의존적인 부분에 대한 모든 수정이 가해져야 한다는 점을 간과한 것이다. 또한, 특정 목적으로 개발된 플랫폼을 다른 목적으로 사용하기 위해서는, 대부분의 모듈에 대한 재사용을 포기해야 할 수도 있다는 점을 이해하지 못한다. 다시 말하면, 플랫폼이 모든 문제의 해결해 주는 “은 빛 탄환(Silver Bullet)”은 아니라는 것이다.

플랫폼은 일정 수준의 범용성을 가지고 개발되지만, 관련된 모든 제품에 다 적용할 수 있는 것은 아니다. 예를 들어, 비용이 중요한 제품(Low-end)과 하이엔드(High-end) 사용을 위한 제품에 동일한 플랫폼을 적용할 수는 없는 것이다. 제품 생산 가격을 줄이기 위해서는 사용하는 CPU, 메모리, 외부 저장장치등 다양한 부분을 고려해서 플랫폼을 구성해야 한다. 고 사양 제품은 그런 부분을 고려하는 것보다 일관성과 재사용에 초점을 맞추는 것이 일반적이다. 물론, 중급 제품(Mid-end)제품의 경우에는 고성능 제품과 호환할 수 있는 모듈들이 더 많을 것이다.

전혀 목적이 다른 곳에 사용하고자 한다면, 재사용되는 범위는 상위에서 하위로 구성된 계층구조에서 하위에 있는 몇 몇 일부 모듈만 해당할 것이다. 예를 들어, 운영체제의 경우 특정 목적으로 개발된 것도 있지만, 일반적인 목적인 Linux나 Windows, MacOS는 특정 응용 프로그램 만을 위해서 설계된 것이 아니다. 즉, 이것은 재사용은 높이지만, 성능에서는 전용 운영체제 보다 못하다는 것을 전제로 해서 만들어진 것이다.

디바이스 드라이버(Device Driver)와 같은 것도 운영체제가 달라지면 만드는 방법도 달라진다. Linux에서 사용하는 디바이스 드라이버를 Windows와 같은 곳에서 사용되는 NDIS나 WDF 혹은 WDM에 사용할 수는 없다(물론, 디바이스 드라이버 소프트웨어가 하드웨어에 직접 접근하는 순서나 사용하는 시나리오 정도는 재활용이 가능하지만). 운영체제마다 디바이스 드라이버에 대한 인터페이스 및 메모리 할당, 인터럽트와 I/O처리를 위한 부분이 다르기 때문이다.

어쨌든 운영체제 및 일부 시스템 라이브러리 정도는 사용하는 목적이 달라지더라도 범용성을 가지기에 재사용이 가능하지만, 비지니스 로직(Business Logic)을 처리하는 계층은 완전히 새롭게 구현해야 한다. 비지니스 로직은 기능에 해당하는 부분이며, 그것이 달라지면 당연히 변경해야 할 부분이 되는 것이다. 동일한 기능을 가진다면 당연히 재사용될 수 있을 것이지만, 재사용되는 상황에 따라 구현이 달라질 가능성은 여전히 남아 있다.

제품군이 달라도 일정 부분에 대한 소프트웨어는 재사용이 가능하다. 예를 들어, 전화기(Phone)용 소프트웨어 플랫폼을 TV에도 사용하는 것이 가능할까? 충분히 가능한 이야기다. 즉, 전화 호출(Call)에 필요한 소프트웨어 모듈의 스택(Stack)을 제거하고, 전화기에 치중된 사용자 인터페이스를 변경하고, 멀티미디어 디코딩(Multimedia Decoding) 기능을 강화한다면 충분히 가능하다(실제로 안드로이드를 이용해서 전화기가 아닌 TV나 네비게이션 장비를 만들 수 있다).

물론, 일부 응용 프로그램들은 변경해야 할 수도 있다. 예를 들어, 웹 브라우저(Web browser)는 넓은 화면에 맞게 UI를 변경해야 할 것이고, TV를 항상 보고자 하는 사람들을 위해서 PIP(Picture in Picture)기능을 멀티 태스킹(Multi-Tasking)으로 동작시켜야 할 것이다. 또한, 채널을 돌아다니기(Channel Navigation)나 채널 검색을 위해서 필요한 기능도 있을 것이다. 따라서, 단순히 플랫폼을 새로운 제품에 적용하는 재사용만이 문제가 아니라, 다양한 변경이 전반적으로 발생할 수 있다는 것을 간과해선 안된다.

실제로 소프트웨어는 “Soft”하지 않다는 말을 많이 한다. 처음 10만 라인 정도는 쉽게 변경할 수 있다고 하더라도(물론, 이것도 힘들긴 하지만), 100만 라인이나 그 이상이 되는 코드를 변경하는 것은 완전히 새로운 문제가 되어버리고 만다. 새로 처음부터 개발하는 것보다 더 오랜 시간이 걸릴 가능성도 있다. 따라서, 소프트웨어는 만들어지고 나면 점차 경화(Hard)되어 간다고 보는 관점이 옳은 것처럼 보인다. 비용이 많이 들어가는 변경은 오히려 하드웨어를 변경하는 것보다 더 경제적으로 악영향을 줄 가능성이 있기 때문이다.

따라서, 대부분 개발자는 될 수 있으면 동작하는 코드를 변경하지 않으려는 경향이 강하며, 이것도 주어진 일정과 비용 한도 내에서 수행되기를 원한다. 특히, 잘못 구현된 소프트웨어 구조를 변경하는 작업은, 일정 계획을 엉망으로 만들고, 비용을 기하급수로 증가시켜, 결과물이 사용자를 만족시키지 못하게 만든다. 소프트웨어는 아무런 대가 없이 쉽게 고쳐질 것이라는 근거없는 기대를 절대 해서는 안된다. 물론, 개발 초기부터 변경을 위해서 구조화되고 확장성 있는 코딩 방법을 동원하는 것이 필요하지만, 이것도 사람이 하는 일이라 항상 실수는 있기 마련이다.

소프트웨어 플랫폼을 만드는 일은 정치적인 일이다. 기술적인 노력은 당연히 필요한 것이며, 정작 만들고 나서 사용할 것인가가 중요한 과제다. 여기서 만드는 방법을 논하기보다는 어떻게 적용할 것인가를 알아보도록 하자. 사용하는 대상이 누군가를 아는 것이 가장 먼저 해야 할 일이다.

소프트웨어 구조는 조직의 구조를 따른다. 윗 사람의 권위가 높은 조직은 윗 사람의 의지가 반영된 소프트웨어 구조가 나오고, 구성원이 동의한 구조보다는 급조된 구조가 될 가능성이 높다. 그리고, 경직된 조직일수록 프로그램도 구조화되지 않을 가능성이 높다. 슈퍼급(?)(여기서는 다른 개발자 보다 권한이 큰 개발자를 일컫는다.)의 개발자가 있는 조직은 슈퍼급의 문제로 야근의 빈도도 높아질 것이다. 코드 리뷰는 동등한 입장에서라기보다는 일방적인 입장에서 진행될 것이며, 커밋(commit)은 혼자서 파일 서버처럼 저장소(repository)를 관리할 것이다. 부서 간에는 대화가 없어지고, 코드를 보여주는 것도 결재를 받고 해야 할 수도 있다.

결국, 플랫폼을 개발하는 것은 기술적인 문제가 아니라 사람의 문제다. 소프트웨어가 가지고 있는 근본적인 복잡도를 다루는 사람이 핵심이다. “사람을 다루는 것”이 플랫폼을 만들 책임을 가진 사람이 해야 할 일이다. 기술적인 것은 크게 장애가 되지 못한다. 시간을 두고 생각하면 풀어내지 못하는 문제는 없다(물론, 그 시간이 항상 부족한 것도 사실이지만). 하지만, 완성도가 있는 것을 만들어도 결국에는 사장될 수 있는 것이 경쟁이며, 경쟁은 기술적인 우위보다 시장의 검증을 우선한다. 즉, 무너진 구조를 가진 코드라도 시장에서의 검증을 토대로 더 오래 살아남을 수 있다는 것이다. 따라서, 플랫폼이 살아남기 위해서는 시장에서 검증된 “기존 코드”보다 “확신”을 주어야 한다.

그렇다면, 어떻게 해야지 이것을 극복할 수 있을까? 한 가지 방법은 플랫폼을 가져다 사용할 사람들을 고객이라는 관점에서 바라보는 것이다. 지금 만들고 있는 플랫폼의 이해 당사자(Stakeholder)를 제대로 파악해야 한다. 그 사람들의 적극적인 참여를 개발하는 과정에서 끌어낼 수 있다면, 플랫폼은 자연스럽게 성공할 것이다. “Give and Take”라는 말이 있듯이, 먼저 주고 나중에 받는 것이 고객을 대하는 기본이다.

그들에게 제공해 줄 수 있는 것이 무엇이 있는지 보고, 그것을 먼저 제시해서 그들이 도움을 받았다는 것을 은연 중에 알게 만드는 것이다. 그것이 나중에 플랫폼을 현업에 적용할 때 양보를 이끌어내는 지름길이다. 또한, 실제 사용하는 고객의 요구사항을 제대로 알고 있는 사람들도 그들이며, 도메인(Domain)에 대해서도 최고의 전문성을 가지고 있다. 그들은 이미 오랫동안 해당 분야의 소프트웨어 개발을 담당했기 때문이다.

# [ 나는 객체지향으로 일하는가? ]

“우리는 지금 C++로 작업을 하고 있습니다.”라는 말을 듣는다. C++이라는 언어는 객체지향 개념을 구체화시키기 위해서 사용하는 프로그래밍 언어다. 정확히 말하면 객체지향이 먼저고, C++은 구현을 위한 툴(도구)로 사용하는 것이다. 그렇다면, 객체지향이란 뭘까? 많은 고민을 했겠지만, 결국에는 그것도 경제성의 산물일 뿐이다. 즉, 어떻게 문제를 다루어야지 유지보수가 쉽고, 재활용할 수 있는 부분을 늘릴 수 있을까를 고민한 결과라는 것이다.

객체지향을 사용해서 들어가는 공수(투입되는 사람의 수와 시간)를 줄일 수 있고, 개발 기간을 단축해서 인건비를 줄일 수 있다는 가정하기 때문이다. 따라서, 유지보수와 재활용이라는 두 가지 이유 때문에 우리가 사용하는 것이 객체지향 언어인 C++다(물론, 객체지향 언어에는 그 이상의 장점이 있지만).

유지보수와 재활용성을 높이려면 어떤 것이 필요할까요? 가장 쉬운 것은 역할과 책임(Role & Responsibility)을 명확하게 정의하는 것이다. 모든 일은 정의에서 시작된다. 소프트웨어 개발은 요구사항을 정의하고, 소프트웨어의 구조를 정의하는 것으로부터 시작한다. 시스템이 할 일을 정의하고, 시스템을 구성하는 요소를 정의하고, 각 요소에 필요한 하위 구성 요소를 정의하는 것이 설계다.

또한, 정의한다는 것은 “역할과 책임의 범위를 설정”하는 것과 같다. “System->Subsystem->Package->Component/Module->File->Class->Function->Variable”등등, 각각은 자신의 역할만 충실히 수행하도록 설계되고 고안되어야 한다. 그 이상도 그 이하도 아닌 딱 자기가 맡은 임무만 충실히 수행하고, 그 역할의 범위 내에서 책임을 지면 된다 것이다. 만약, 해결해야 할 문제가 자신이 책임져야 할 범위를 벗어난다면, 다른 객체에게 요청(Request)해서 처리하면 된다.

객체지향은 이를 위해서 각각의 객체가 해야 하는 일과 책임 범위를 명확히 하라고 이야기한다. 그리고, 객체가 해야 할 일은 자신의 책임 범위에 두고자 한다. 즉, 다른 객체는 알 필요(간섭할 필요)가 없다는 말이다. 알 필요가 없다는 것은 객체가 가지는 내부 정보를 알려주어서도 안 되고, 알려고 해서도 안 된다는 뜻이다. 객체가 가지는 정보가 외부에 많이 공개될수록, 다른 객체의 간섭을 받을 가능성이 높아지며, 독립적인 역할을 수행하지 못하게 될 가능성이 커지게 된다.

큰 문제의 풀이는 작은 문제들의 풀이의 합으로 구성되고, 각각의 작은 문제들은 서로 독립성이 최대로 보장되어야 한다. 이때 문제는 하나의 작은 문제 풀이는 객체로 표현될 수 있으며, 서로 독립적으로 자신의 역할을 충실히 수행해서 해결책을 만들어 나가게 된다. 이를 하나씩 합치면 결과적으로 시스템이 해결해야 하는 큰 문제를 해결해 나갈 수 있는 것이다.

먄약, 이렇게 코딩하고 있지 않다면, 우리는 C++을 사용하고 있을지는 몰라도 객체지향적으로 코딩하고 있지는 않다는 말이 된다. 사실, 대부분의 개발자가 이런 문제에서 자유롭지 않을지도 모른다. 언어는 잘 사용하고 있을지 몰라도, 근본이 되는 개념에 대해서는 제대로 익숙해지지 않은 상태일 것이다. 코딩이란 언어의 문법만이 중요한 것이 아니라, “언어가 가지는 철학과 가치”를 잘 이해할 때 제대로된 코드를 만들 수 있는 것이다.

# [ 코딩의 법칙 ]

사람들은 어떤 게임이든 상관없이 이기려고 플레이 한다. 점심 시간에 음료수나 커피 내기로 가위 바위 보로 하게 될 때도, 어떤 것을 내면 이길 수 있을지 마음 속으로 순식간에 계산한다. 작은 승부는 작은 고민으로, 큰 승부라면 당연히 몇 날 몇 일을 고민할 것이다. 고민의 결과는 승리할 가능성이 조금이라도 높은 것을 “선택”하는 것으로 끝나며, 선택 결과와 상관없이 승패가 결정되고 만다. 물론, 이런 승부는 각각의 선택이 동일한 확률이라는 가정하에 진행되기 때문에, 결과가 그렇게 나온다고 볼 수도 있다. 하지만, 우리가 일상에서 하는 내리는 결정은 어떨까? 과연 동일한 확률이라고 가정하고 선택하는 것일까?

소프트웨어 개발자들은 매일 코딩을 한다. 하지만, 승부의 세계처럼 냉정하게 조금이라도 이길 수 있는 확률이 높은 방향으로 코딩하는 것은 아니다. 즉, 버그와 싸움에서 조금이라도 승리할 수 있도록 코딩하고 있지 않다는 것이다. 사실 버그가 줄어드는 방향의 코딩은 이미 많은 책에서 다루고 있다는 것은 개발자라면 이미 알고 있을 것이다(물론, 학교에서는 잘가르치지 않지만). 하지만, 정작 자신의 코딩에 있어서만큼은 그런 코딩 방법을 적용하지 않는 것도 현실이다.

개인 개발자들이 하는 코딩의 대부분은 게임에서 이길 가능성이 높지 않다. 좋은 코딩 습관을 만들어야 하는 것이 중요하지만, 온갖 그렇게 하지 않을 핑계를 찾고 있는 것도 개발 실무자들이 가지는 아이러니다. 물론, 이유는 주로 외부에서 오지만, 행동의 결정을 내리는 것은 어쨌든 개발자 본인이라는 점을 잊어서는 안된다. 코딩에서 이길 확률이 높은 방법은 “버그 발생 가능성을 사전에 차단하는 것”이며, “변화를 수용하고 읽기 쉬운 코드를 만드는 것”이다. 여기서는 이러한 코딩에 있어서 버그를 상대로 승리할 수 있는 구체적인 코딩 방법에 대해서 몇 가지 논의해 보도록 하자.

**01. 큰 문제는 작은 문제로 나누어라.**

한 번에 모든 문제를 해결하려고 달려들다가는 제대로 문제의 해답을 구하기도 전에 지쳐버릴 수밖에 없을 것이다. 물론, 우리가 구하는 모든 해법이 최적이라고는 할 수 없다. 하지만, 큰 문제는 작은 문제로 나눌 수 있는 경우가 많고, 작은 문제는 더 작은 문제로 나눌 수 있다. 따라서, 문제의 해답을 구하는 것은 큰 문제를 얼마나 작은 문제로 나눌 수 있는가에 달려있다고 볼 수 있을 것이다.

예를 들어, 도시 전체의 차량 통행량을 알고 싶어한다고 가정하자. 가장 작은 단위로 골목길에서 부터 시작해서, 일반 도로, 간선도로로 이어지면서 통행량을 측정해 나갈 것이다. 전체 도시의 차량 통행량을 한번에 알기는 어렵지만, 작은 단위에서 측정한 결과값을 계층적으로 차곡차곡 쌓아가면서 최종 결과값을 만들 수 있을 것이다.

나누는 것은 문제를 작게 만들어서 지역적인 해답을 찾을 가능성을 높이는 방법이다. 물론, 개별적인 지역적인 문제의 해답이 정확할수록, 모아진 결과값도 정확할 가능성이 높다. 가장 구체적인 해답에서 상위의 더 높은 수준으로 측정한 값들을 전파시키다 보면, 전체 수준에서도 문제의 해답에 가까운 결과를 얻을 수 있을 것이다. 물론, 이때 얻어진 최정 결과값이 진짜 해답이라고 단정할 수는 없지만, 전체를 한번에 놓고 해결하려고 내놓은 답보다 정답에 가까울 확률은 높을 것이다.

**02. 중복은 모든 버그의 서식처다.**

코드의 중복은 항상 발생하지만 충분히 제거될 수 있다. 다만, 게으른 개발자는 그것을 쉽게 용납하며, 가까운 미래에 혹독한 대가를 치르게 될 수 있다는 것이다. 코드의 중복이 나쁜 것은 그것을 고치는 사람이 완벽하지 않기 때문이다. 모든 코드를 다 기억하고 있는 사람은 없기에, 하나의 코드 변경을 모든 중복된 코드 부분에 전부다 적용하는 것은 결코 쉬운 일이 아니다.

“Copy-Paste”가 프로그래머의 가장 많이 사용하는 도구라는 것은 익히 알려진 사실이다. 심하게 이야기하면, 복제해서 사용하는 코드가 개발된 코드의 90%이상이 될 가능성도 있다. 책에서 배끼기도 하고 인터넷에서 가져오기도 한다. 오로지 자기가 가진 지식만으로 모든 문제를 해결할 수는 없기에, 남들이 이미 작성한 코드를 사용하는 것은 훌륭한 방법이 될 수 있다. 물론, 코드의 라이센스 문제없고, 사용하려는 이유와 목적이 정확히 같다고 한다면 그대로 가져와서 사용해도 무방할 수 있다.

하지만, 자신의 코드에 동일하거나 비슷한 코드들이 여러 곳에 복제되어 존재하는 것은 문제가 될 수 있다. 동일한 이유에서 발생하는 변경이 복제된 모든 코드를 함께 바꿔주어야 하기 때문이다. 원래 저작자가 아닌 경우에는 자신의 변경이 전체 시스템에 어떤 영향을 미치게 될지도 모른다. 따라서, 유지 보수를 하기 위해서는 복제된 모든 코드를 일일이 추적해서 관리해 주어야 한다. 물론, 이런 것들은 대부분 문서가 없으며, 누군가의 머리 속에만 존재할 가능성이 높다. 따라서, 하나의 변경은 항상 다수의 버그를 유발하는 원인으로 작용하게 될 가능성이 높아지게 된다.

**03. 복잡한 것은 간단하게 만들어라.**

코드의 “IF, Switch” 등으로 묶인 부분이 코딩을 해나가는 동안 점점 더 커진다면, 아마도 하나의 함수가 1,000줄(물론, 300줄이라고 해서 길지 않은 것은 아니지만)에 육박하는 코드를 만들 확률이 높다. 그렇게 긴 함수를 봤을 때 처음으로 드는 생각이 무엇인지 자신에게 솔직히 물어보라. 아마도, "제기랄 어떤 놈이 이렇게 코드를 짰어?" 라고 욕을 하고 있을지도 모른다. 마찬가지로 똑 같은 말을 자신의 코드도 들을 수 있을 것이다. 자신이 코드를 그렇게 짜고서도 욕먹지 않을 만큼 뻔뻔스럽지 않다면, 될 수 있으면 짧고 간단한 코드를 만들어야 한다.

"완벽함이란 더 추가할 것이 없다는 것이 아니라, 더 뺄 것이 없는 상태를 말한다."라고 누군가 말했다. 이 말은 소프트웨어 개발에도 유효하다. 하지만, 이렇게만 말하면 “짧게 코딩할 수 있는 모든 비법을 배워야 하나?” 라는 의문을 가질 수 있을 것이다. 무조건 코드를 짧게 짜는 팁(Tip)을 많이 안다고 해서 간단한 코드를 만들 수 있는 것은 아니다. 간단한 코드란 남이 보았을 때 쉽게 개발자의 의도를 파악할 수 있는 것을 말한다.

복잡함은 버그를 유발할 가능성이 높을 뿐만 아니라 제거하기도 어렵다. 밤 늦게 야근해서 조금이라도 가정에 도움이 되는 사람이 되고 싶다면(야근비라도 받고 싶다면), 열심히 복잡하게 코드를 만들어도 좋겠지만, 조금이라도 다른 사람에게 도움을 주고 싶다면 간단한 코드를 만들어야 한다. 의도가 명확한 코드는 나를 위한 것이 아니라, 타인을 위한 이타적인 행동의 결과물이다. 야근으로 벌 수 있는 돈은 얼마되지 않지만, 이타적인 행동의 결과는 반드시 배로 보상받기 마련이다.

**04. 역할과 책임(Role & Responsibility)을 지키자.**

프로그램을 단위가 작은 모듈이나 혹은 클래스로 나뉘었다고 가정하자. 이렇게 정해진 모듈이나 클래스가 정해진 역할 이상을 하고 있는지 확인해야 한다. 예를 들어, DB(DataBase)에 대한 관리를 하는 클래스를 만든다고 하자. 즉, “DB Manager”와 같은 클래스가 될 것이다. 하지만, 프로그램을 개발하다 보면 역할과 책임은 점점 모호해지고, DB를 관리하는 클래스는 소켓(Socket)을 열어 통신 기능을 추가적으로 구현하고 있을지도 모른다. 어느 순간 데이터를 받아서 저장하고, 검색하고, 읽고 쓰는 일도 같이하고 있을 것이다.

우리가 하는 일은 제대로 된 하나의 역할만 충실히 하기에도 너무나도 복잡하다. 여러가지 일을 한꺼번에 하게되는 이유는 생각의 연결고리를 끊이없이 이어기가 때문이다. 미리 정의한 역할과 책임의 구분없이 코드를 만들기 때문이다. 따라서, 생각의 고리를 끊기 위해서는 자신이 코딩하고 있는 부분이 어떤 역할과 책임을 가지는지 명확히 해야한다. 만약, 작성해야 하는 코드가 그 범위를 벗어난다면, 다른 모듈로 책임을 이관해야 한다. 해당 역할이 정해진 모듈이 없다면 새로운 모듈을 설계에 추가하면 된다. 설계라고 고치지 못할 이유는 없는 것이다.

역할과 책임을 잘 정의하는 것은 사람에게도 중요하며, 사람이 만들어내는 창조적인 결과물인 코드에도 당연히 중요하다. 너무 많은 역할을 하나의 클래스나 모듈에 넣는 것은, 결국 실수할 가능성만 높이게 된다. "이 기능이 내가 만들고 있는 소프트웨어의 모듈이나 클래스의 고유한 역할일까?", 이 단순한 질문만으로도 많은 문제를 해결할 수 있을 구조를 만들 수 있을 것이다. 소프트웨어 설계는 역할과 책임의 할당이며, 그것을 구체화시키는 것이 코딩인 것이다.

**05. 모든 것을 다 하는 것은 아무것도 안 하는 것과 같다.**

오래된 문제로 “전역 자료구조(Global Data Structure)”에 대한 비판이 있다. 전역 자료구조는 참기 어려운 유혹이다. 하나의 자료구조를 정의해서, 요구하는 모든 모듈이나 클래스에서 가져다 사용할 수 있으면, 성능이 높고 효율적인 프로그램을 작성할 수 있을 것이라고 생각할 수 있다. 물론, 우리는 관습에 가까울 정도로 이러한 것을 사용하지 말도록 교육을 받아왔다. 어떤 코딩에 관련된 책을 찾아보더라도 전역 자료구조는 될 수 있으면 사용하지 말라고 되어있다.

모든 문제를 해결하는 “은빛 탄환(Silver Bullet)“은 없다는 것을 소프트웨어 공학에 관련된 여러 가지 책에서 이미 충분히 읽었을 것이다. 하지만, 자기 자신에게 스스로 질문해 보라. 정말 그런 코드가 하나도 없을까? 아마도 제대로 교육받은 개발자들이라고 하더라도, 전역 변수 하나 정도 사용해도 별 문제없을 것이라고 확신하고 있을 것이다. 물론, 그렇게 만들어진 전역 변수는 절대 하나로만 끝나지 않는다. 마치 “깨진 창문 효과”처럼, 걷잡을 수 없이 전역 변수의 수는 급격하게 늘어날 것이다.

모든 것을 다 하는 것보다는 한가지라도 제대로 하려고 노력하는 것이 좋다. 하나의 함수나 메쏘드가 여러가지 일을 처리하기 위해서는, 입력의 값과 종류가 늘어나야 한다. 각각의 입력에 대해서 분기해야 하는 회수도 늘어나게 된다. 결과적으로 코드는 길어지게 되고, 기능의 추가나 변경이 발생할 경우 빈번하게 수정하는 코드가 될 것이다. 한 가지만 충실하게 수행하는 코드는 입력의 값과 종류가 단순할 것이며, 변경의 이유도 한 가지로 한정될 것이다. 당연히 잦은 변경은 발생하지 않을 것이며, 변경의 범위 자체도 줄어들게 된다. 코드가 변경이 잦지 않다는 말은 그 만큼 안정적이라는 의미다. 따라서, 버그가 발생할 가능성도 줄어드는 것이다.

모든 것을 다 하는 코드를 없애는 것은 그 만큼의 대가를 충분히 얻을 수 있다. 지금 당장은 크게 도움이 안된다고 생각할지도 모르지만, 가까운 미래나 과제 완료 이후에 발생할 수 있는 추가 및 변경에 대해서, 비용 절감(시간 단축)이란 보상이 돌아올 것이다.

**06. 주려고 해야 받을 수 있다.**

닫힌 마음을 가진 개발자는 열린 생각을 하는 사람들보다 두려움이 많다. 두려운 생각은 기회가 와도 기회인지 모르고 그냥 흘려보낸다. 누군가 자신이 한 일의 진정한 가치를 알고자 한다면, 사람들에게 두려움 없이 자신 있게 보여줄 수 있어야 한다. 보여주지 못할 정도의 코드라면 고쳐라. 고치지도 못할 코드라면 버려라. 버리고 새로 시작하지 못할 것은 없다.

물론 상품화를 하는 코드에 대해서 이렇게 말한다면 미친놈 취급을 받을 것이 자명하다. 하지만, 스스로 부끄러운 코딩은 하지 말아야 한다. 어떤 이유에서건 결국에는 자신의 동료와 선후배에게 시간이나 금전적인 손실을 주게 될 것이기 때문이다. 한 번에 모든 것을 고치기는 힘들 것이다. 따라서, 최소한 어떤 일정으로 어떻게 고쳐나갈 것인가를 명확히 제시하자. 남에게 자신의 코드의 취약점과 좋은 점등을 잘 설명하고 의견에 경청하다 보면, 결국은 자신이 받는 것이 더 많다는 것을 알 수 있을 것이다.

한 명보다는 열 명이, 열 명보다는 백 명의 사람이 보는 것이 버그가 발생할 확률을 줄이는 것은 당연하다. 세상은 인재로 넘쳐나고 자신은 그중에 제일 둔재라고 생각하면, 두려움보다는 배울 것들로 넘쳐난다는 것을 발견할 수 있을 것이다.

# [ 소프트웨어 설계는 얼마나 해야할까? ]

흔히들 소프트웨어를 개발할 때, 설계를 먼저하고 구현은 나중에 해야 한다고 말 한다. 하지만, 실제로 이러한 것이 가능할까? 대부분 코드는 사실 완전한 설계에서 나오지 않는다. 디자인 리뷰(Design Review)에 듣는 많은 이야기 중의 하나가 구체적인 인터페이스 스펙(Spec.)을 보여달라는 요청이다. 하지만, 그러한 구체적인 인터페이스 스펙을 모두 정의할 정도면, 이미 구현의 대부분은 마무리되어 테스트를 기다리는 입장에서나 가능한 소리다. 그럼, 도데체 우리는 얼마만큼의 노력을 소프트웨어 설계에 투자해야 하는 것일까? 아래는 몇 가지 경우에 대해서 정리해 본 것이다.

**01. 기존에 이미 동작하는 코드가 있는 상황에서 리팩토링 하는 과제의 경우**

기존의 동작하는 코드를 개선하는 과제의 경우에는 설계가 과연 필요할까? 당연히 필요하다. 즉, 어떤 청사진으로 과제를 진행할 것인가에 대한 원칙 정도는 마련되어 있어야 하며, 개선점이 되는 부분에 대해서 대략적인 클래스 다이어그램(Class Diagram)이나 모듈 구조 정도는 작성하는 것이 방향을 잡는 데 도움이 된다. 즉, 개선 방향을 설정하고 그쪽으로 서서히 진행되어 가기에, 어디까지 개선되었는지 진척도를 파악하기도 쉽기 때문이다.

하지만, 완전히 새로운 설계를 적용하기 힘든 점이 많을 수 있다. 즉, 새로운 구조를 정의하고 기존의 코드를 수정하다 보면, 얽혀있는 코드를 풀어내는데 상당한 비용과 시간을 들여야 하기 때문이다. 이때는 중요한 것부터 하나씩 차근차근 설계하면서 코드도 동시에 수정을 병행하는 것이 좋다. 설계와 코딩은 서로 피드백을 주고받는 관계이기 때문이다.

**02. 기존의 이미 동작하는 코드에서 새로운 기능을 추가하는 과제의 경우**

기존의 코드까지도 다 설계서로 만들 필요가 있을까? 아마 필요없을 것이다. 아니면, 최소한 이해하는 수준에서의 대략적인 그림만 나오면 충분할 것이다. 하지만, 만약 기존 코드에 대한 설계서가 없다면 어떻게 할 것인가? 이 때는 설계서를 만들어야 한다. 기존의 코드를 이해하기 위해서도 코드에서 설계서를 뽑아내는 것이 필요하다. 사실 대부분의 기존 코드는 설계서가 없을 가능성이 더 크다. 하지만, “다른 사람도 안했으니 나도 안해도 된다”는 생각은 옳지 않다.

세세한 부분은 이미 다른 사람이 구현한 것이며 문서도 그 사람이 담당하는 것이 옳다. 과제의 초보자가 이해의 목적으로 코드를 분석하는 것은 좋지만, 발등에 떨어진 불을 끄고 있는 동안에는 기존 담당자가 설계서를 만드는 것이 좋다. 주로 새로운 기능의 추가는 즉각적인 반응이 필요한 시급한 일정의 과제가 될 가능성이 높기 때문이다. 시간적인 여유가 조금 있다면, 기존 설계와 변경된 설계를 함께 검토해 볼 필요도 있을 것이다. 무엇이 어떻게 변경되었는지 모든 사람이 공유할 필요가 있을지도 모른다. 그리고, 여러 사람이 볼 수록 더 좋은 구조가 나올 가능성도 크다.

**03. 기존의 코드가 없는 완전히 새롭게 작성해야 하는 과제의 경우**

완전히 새롭게 만들어야 하는 과제의 경우는 비교적 자유도가 높다. 따라서, 시간이 허락하는 한도 내에서 가능한 자세한 설계를 진행하는 것이 좋다. 하지만, 너무 많은 시간을 낭비하지 말자. 시간은 한정된 것이며 자원은 생각보다 비싸다. 설계를 검토하는 것도 중요하지만, 결국엔 코딩을 해야 한다. 소프트웨어의 특성상 구현은 반드시 따라오는 결과이기 때문이다. 한번에 완벽한 결과물을 만들려고 노력하지 말고, 반복적으로 개선하는 방향을 선택하는 것이 좋다는 뜻이다.

대략적인 “중요한 구현부”에 대한 설계가 완성되면, 그것을 가지고 구현에 들어가도 된다. 중요 구현부란 높은 성능을 요구하거나 자주 사용되는 부분 등, 전체 시스템의 구현에 핵심이 되는 부분을 말한다(주로 비기능적으로 만족시켜야 할 요구사항을 구현하는 부분). 즉, 이 부분에 대해서는 반드시 설계 및 검토 과정이 필요하다. 그 외의 중요도가 낮은 부분들에 대해서는 코딩을 진행하면서 설계해도 크게 지장은 없을 것이다.

참고로, 애자일(Agile) 방법론에서는 설계에 관련된 활동에 큰 비중을 두는 것 같지는 않지만, 그래도 설계 및 검토에 관련된 부분은 존재한다. “지나친 문서화 보다 동작하는 코드를 선호”할 뿐이지, 설계를 하지 않아야 한다고 이야기하는 것은 아니다. 소프트웨어 개발자라면 최소한 설계를 어떻게 하는지(반드시 UML을 사용해서 할 필요는 없다. 과제 관련자들이 동의[서로 같게 해석하는]한 형태의 이해하기 쉬운 것이라면 어떤 것이라도 상관없다.)는 한 번쯤 고민해야 하며, 설계는 코딩을 위한 준비이지, 코딩과 나누어진 활동이 아님을 기억해야 한다.

# [ 객체지향으로 가다. ]

새로운 신입사원들은 객체지향 언어에 익숙하다. Java나 C++, C#등의 언어를 통해서 학교 숙제를 열심히 한 사람은, 어느정도 객체지향이 무엇인지 어렴풋하게 나마라도 개념을 가지고 있을 것이다. 물론, 그렇다고 신입사원이 객체지향적으로 코드를 작성한다고는 확신하지 못한다. 경력 사원들은 객체지향 언어에 대해서 문법 정도는 알고 있을지도 모른다. 경험해 보지 않으면 문법 자체도 모를 가능성이 높다. 대부분의 임베디드 시스템 개발자들은 C++같은 객체지향 언어를 업무에서 사용해 본 경험 조차도 없을 것이다.

조금만 더 자세히 들여다보면, 신입으로 들어온 사원들은 객체지향에 대해서 그렇게 심각한 고민 없이 주어진 숙제를 하기에 바빴을 것이다. 경력 사원과 마찬가지로 역시 문법에만 익숙한 경우가 많다. 혹은, 문법 자체도 크게 잘알지 못하고, 사용하던 것만 매번 반복했을 수도 있다. 정작 객체지향적으로 코딩한다는 것이 무엇인지 정확히 이해하지 못하고 있다는 것이다. 문제는 여기서 발생한다. 객체지향을 가장한 객체지향적이지 않은 코드들이 실제로 많다는 것이다.

객체지향 문법으로 객체지향적이지 않게 작성된 코드는, 과거의 남겨진 유물에 색을 덧칠하거나 살을 덧붙인 것일 뿐이다. 즉, 과거에 부닥쳤던 문제들을 다시 똑같이 반복할 뿐이다. 클래스는 구조체를 대신해서 썼을 뿐이며, 메쏘드는 함수의 대체 수단 정도로 밖에 활용되지 않을 것이다. 클래스의 정의를 물어본다면 객체를 만들어 내는 틀(Template)이며, 객체는 클래스의 메모리 인스턴스(Instance)라는 순환적인 정의 밖에 하지 못한다. 그렇다면 도데체 “객체지향적”이란 말은 무엇이며, 어떻게 해야 객체지향으로 갈 수 있는 것일까? 아래에서는 객체지향에 대한 몇 가지 생각을 정리해 보았다.

**01. "특정 객체가 하는 일을 다른 객체가 모르게 하라."**

**02. "특정 객체는 정해진 자신의 역할만 수행하라."**

쉽게 말해서 역할을 명확히 정의하고, 객체 간에 필요한 통신은 정해진 “계약(Contract)”을 이용하라는 말이다. 역할을 명확히 하라는 것은 자신이 해야 할 일의 범위를 정하고, 그 외의 일은 다른 객체에 요청(Request)하라는 것이다. 이때, Class의 단위를 지나치게 크게 가져가면, 마치 전역 변수와 같은 Class가 만들어질 것이고, 너무 작게 가져가면, 관리해야 할 Class가 지나치게 많아질 것이다. 따라서, 역할에 대한 적절한 범위를 설계에서 결정하여야 한다는 말이다. 일반적으로 Class간의 의존성을 줄이는 것을 Decoupling이라고 한다(굳이 Class간이라고 한정할 필요는 없다). 또한, 해야 할 일만을 하라는 이야기를 Cohesion이라고 표현한다. 즉, 의존성을 줄이고 응집성을 높이는 일련의 활동을 Class와 같은 개념을 활용해서 적절히 사용할 수 있다면, 거의 객체지향이라는 개념에 도달했다고 볼 수 있다.

이제 남은 객체지향의 원리는 상속이다. 상속은 사실 객체지향 개념에서 재활용을 줄이는 중요한 요소로 사용되지만, 또한 상속하므로 해서 코드 개선 때문에 변경되는 부분이 늘어나는 양면의 칼이 될 수 있다. 즉, 코드의 재사용성을 높이지만, 한편으로는 버그를 늘릴 가능성도 상존한다. 부모에서 수정된 내용이 상속을 통해서 자식에게 흘러갈 가능성도 있고, 부모의 의도를 잘못 해석한 자식을 만들 가능성도 존재한다. 하지만, 상속이 나쁜 것이 아니라 올바른 상속을 사용하자는 것이지 쓰지 말라는 것은 아니다.

객체지향은 기존의 프로그래밍에 대한 틀을 바꾸었으며, 사람들의 생각에 대한 변경을 요구하고 있다. 객체지향으로 코딩하든 하지 않든 객체 지향적인 개념을 이해하고, 이를 자신의 코딩에 도입하는 것은 유익한 일이다. 복잡한 문제를 작은 문제로 나누고, 이를 다시 적절한 추상화를 통해서 해야 할 역할을 나누어, 잘 조직화한 코딩으로 해결책을 찾아 나가는 것은 어쩌면 좋은 코딩의 가장 기본적인 자세이다. 어떤 프로그래밍 언어를 사용한다고 해서 더 좋아진다는 보장은 없다. 프로는 도구에 의존하지 않고 언제나 고객의 만족을 극대화시켜주기 때문에 존재 가치가 있는 것이다. 프로는 어떤 도구나 언어를 사용한다고 해서 필요한 사람은 아니기 때문이다.

# [ 모델(Model) 추가에 따른 분기문의 문제점 ]

제품을 만들다 보면 후속 제품에서 모델을 추가하면서 들어가는 신기능에 대한 부분을 따로 관리해야 하는 오버헤드가 발생한다. 이때 가장 쉬운 해결방법은 "#if xxxx defined"와 같은 방법을 통해서 조건부 컴파일 옵션을 사용하거나, 혹은 실행문에서 "if xxxx else yyy"나 "switch()"문을 사용하는 것이다. 하지만, 문제는 이런 식으로 계속 대응하다 보면, 원래 코드는 금방 지저분하게 변해버리고, 이해하기 어려운 코드를 변질된다. 결국, 모든 버그의 온상이 되는 것도 시간일 뿐이다. 그렇다면, 이러한 모델의 추가에 따른 분기를 어떤 식으로 처리하는 것이 가장 효과적일까? 해답은 단순히 컴파일 옵션이나 "if", "switch"문 등을 사용하지 않는 것이다.

C언어는 상속이라는 개념이 없다. 따라서, 당연히 상속할 수 있는 방법도 없다. 따라서, 상속을 통해서 다형성을 제공해 주기 위해서는, 주로 함수의 포인터를 상용해서 위에서 제기한 문제를 해결한다. 즉, 해당 모델에 고유한 기능을 함수로 만들어서, 상위 모듈에서 함수의 포인터를 이용해서 사용할 수 있게 만들어주는 것이다. C++과 같은 객체지향 언어에서는 인터페이스를 사용하면 간단히 해결된다. 즉, 초기에 지원해야 하는 모델들을 클래스를 이용해서 생성하고, 새로운 기능에 대한 인터페이스를 상속받아 필요할 때 구현하면 된다. 나중에 모델에 따른 기능을 분기하기 위해서, 상위 모듈은 미리 정의된 인터페이스를 통해서 접근하게 되고, 모델에 해당하는 클래스에서 제공하는 멤버 함수를 호출하게 될 것이다.

따라서, 소프트웨어의 구조를 설계하는 입장에서는 향후 기능의 확장이 어디서 발생할 것인가를 파악하고, 관련된 인터페이스를 명확히 하는 것이 중요하다. 사전에 대응할 수 없다면, 위와 같은 상황을 만날 경우 리팩토링(Refactoring)을 어떻게 할 것 인가를 항상 머리 속에 담아 두어야 한다. 리팩토링은 중복을 제거하고 더 읽기쉽고 이해하기 쉬운 코드를 만드는 것을 목표로 하기에, 기능 확장으로 인한 코드의 중복이나 가독성이 나빠지는 상황에 대응하기 위해서 사용할 수 있는 방법이다. 따라서, 리팩토링이 필요한 시점은 컴파일 옵션의 지나친 사용으로 인한 코드의 가독성 문제가 생기는 시점이나, 비슷한 내용의 “if()”나 “switch()”문이 코드의 여기 저기에 등장하고 있다는 것을 깨닫는 때가 될 것이다. 리팩토링을 통해서 역할이 명확히 분리된다면, 누가 어떤 처리를 언제 할 것인가를 정의할 수 있게 될 것이다.

# [ 버그에 대한 대책 ]

소프트웨어 개발자는 버그의 개수가 늘어날 때 불안감을 느낀다. 버그라는 말의 유래는 이미 많이 알려졌듯이, 실제로 컴퓨터를 망가트린 원인이 벌레(Bug)였다는 것이다. 하지만, 컴퓨터의 오동작을 일으키는 버그가 소프트웨어 개발자의 정신 건강을 해치는 버그로까지 진화하고 있는 상황에서, 늘어나는 버그는 결코 유쾌한 상황이 아니다. 버그를 “통제하고 관리하고 분석”해서 더 좋은 소프트웨어를 만들어내는 것이 개발자가 해야하는 일이다.

버그 발생의 근본적인 이유는 무엇일까? 코드에서 발생하는 버그는 주로 너무 많은 것을 하려는 경향에서 생기는 “생각의 복잡함”과 관련이 있다. 즉, 버그는 작성된 소스 코드의 길이에 비례하는 경향이 있으며, 수행하는 기능이 많아지거나, 의존하는 것들이 늘어나게 될 때 발생한다(비례 보다는 좀 더 높은 빈도로 발생하지만).

짧은 코드일수록 버그를 만들 가능성이 당연히 작을 수밖에 없다. 그렇다고, 모든 짧은 코드가 다 좋은 코드는 아니다. “이해가 쉬운 짧은 코드를 많이 짜는 것이 좋다”는 것이다. 가장 좋은 코드는 "내가 작성하지 않은 코드"라는 말이 있듯이, 새로 작성하는 코드의 분량이 적고 이해하기 쉽다면, 버그가 발생할 확률도 낮아질 것이다. 따라서, “생각을 단순화시키는 짧고 이해하기 쉬운 코드”가 핵심인 것이다.

버그를 발생시키는 다른 주요한 요인에는 “지나친 믿음”도 있다. 즉, 호출하는 입장에서 호출하는 함수가 모두 잘 동작할 것이라는 “지나친 기대”로 오류 처리에 대해서 미진한 부분이 있으며, 호출 받는 입장에서는 호출하는 측이 항상 올바른 값으로 호출해 줄 것이라는 믿음에 있다. 사실 이런 것은 아주 기본적인 확인을 요구하는 것으로 쉽게 고칠 수 있는 부분이다. 항상 오류에 대비해서 정상적인 흐름이 아니라, 비정상적인 흐름을 가정하는 것이 도움이 된다.

이를 위해서는 작성하는 코드에 대한 테스트를 충분히 하는 것이 필요하며, 오류 상황에 대한 처리가 제대로 되는지를 비정상적인 상황에서도 테스트 해야한다. 물론, 이렇게 이야기하면 너무 당연한 이야기 같지만, 실제로 어려운 것은 테스트는 “코너 케이스(Corner Case)”와 같은 예외적인 상황을 만드는 것이다. 즉, 거의 일어날 것 같지 않은 경우에 대비해서 테스트를 작성하는 것이다. 비용이 많이 발생할 수도 있기에, 모든 예외적인 상황에 대한 테스트는 하지 못하지만, 필수적으로 발생할 가능성이 있는 것들은 반드시 확인해야 한다.

코드의 복잡함을 줄이는 것은 뜻 밖에 쉬운 해결책이 있다. 즉, “한 번에 반드시 하나만 하는 원칙”을 세우는 것이다. 두 가지의 일을 하나의 함수나 클래스, 패키지, 모듈, 컴포넌트 등에서 처리하지 않는 것이다. 물론, 여기서 말하는 한 번에 한 가지 하라는 원칙은 추상화 수준에서의 책임과 역할을 한 가지로 정하라는 이야기일 뿐이다. 즉, 계층적으로 나누어진 각각의 추상화 수준에서 역할(책임은 당연히 따라오기에)을 한 가지만 가지라는 말이다.

이런 이야기가 어렵다면, 그냥 “한 번에 한 가지 일 만 처리하는 짧은 함수”를 꾸준히 만들면 된다. 클래스의 경우는 자신이 가진 메소드에 필요한 속성만 가지게 만드는 것이다. 클래스는 속성과 그 속성으로 할 수 있는 일이 역할을 정의하기 때문이다. 이것을 연습하다 보면 저절로 다른 부분들을 해결되는 것을 경험할 수 있을 것이다(모듈화나 책임과 역할의 명확화, 요청을 통한 처리, 코드의 중복성 제거, 이해하기 쉬운 코드 등등의 문제).

일반적으로 중복된 코드 문제는 자주 발생한다. 코드에서 발견할 수 있는 중복은 도구로 쉽게 찾을 수 있는 “동일 코드의 중복”과 도구로는 쉽게 찾지 못하는 “로직(Logic)의 중복”으로 나누어 볼 수 있다. 중복된 코드는 지속해서 코드를 변경하는 과정에서 흔히 발생하며, 개발자 간의 의사소통이 원할하지 못할 경우에도 발생할 수 있다. 개발자들은 문제를 해결하는 가장 쉬운 길을 선택하는 경향이 강하며, 편집기(Editor)의 “Copy-Paste” 기능을 절대 신봉하기 때문이다.

코드의 중복을 눈으로 찾는 것은 어렵기에 도구를 활용해야 한다. 이때 도구가 표현하는 값의 의미를 알아야 한다. 즉, 얼마나 코드가 동일해야 복제된 코드로 볼 것인가를 결정해야 한다. 대부분의 경우 여섯 라인 이상의 코드가 같다면, 복제된 코드로 보는 것이 일반적이다. 하지만, 처음부터 이런 기준으로 복제된 코드를 찾는다면, 개발자들의 반발이 클 가능성이 있다. 따라서, 처음에는 조금 완화된 기준으로 적용하고, 서서히 기준을 높여가는 것이 좋을 것이다. 하지만, 그렇다고 너무 높은 라인수를 기준으로 적용하는 것도 좋지는 않으니, 과제를 직접 분석해서 얼마가 적당한지 개발자 들과 합의하면 될 것이다.

전체 코드에서 "if(){}else{}", "switch()"로 유사한 분기가 많은 경우도 주의해서 봐야한다. 이런 분기가 많다는 이유는 계층화해야 할 부분이 누락되었다는 뜻이 된다. 추가적으로 전체 소프트웨어의 아키텍처 분석을 통해 유사한 기능을 수행하는 모듈이 중복해서 들어있는지 확인하는 방법도 있다. 보통의 경우 이런 확인 작업은 코드 통합이 발생하는 경우에 자동으로 실행되도록 만든다. 따라서, 빌드 서버에서 전 처리나 후 처리 정도에 해당하는 자동화 스크립트를 통해서 할 수 있다. 결과는 당연히 관련된 사람들에게 자동으로 통보되도록 만들어야 할 것이다.

로직의 중복은 찾기가 쉽지 않다. 로직의 중복은 거의 유사한 일을 하는 코드가 분산되어 있기 떄문에, 일일이 사람이 찾아내지 않으면 걸러낼 수 없다. 즉, 도구는 문자열에 대한 비교만 할 뿐, 코드의 토큰(Token)이 바뀌면 중복을 찾아내지 못하게 된다. 이런 중복은 사람들이 일을 나누어서 하다 보면 흔히 발생한다. 예를 들어, 유틸리티(Utility)성으로 만드는 함수나 클래스 등이 여러 모듈에 중복해서 존재할 수 있다. 이것을 해결하는 방법은 아키텍처에서 추가적인 모듈을 정의해서 관련 함수나 클래스를 모아 공용화 하는 방법이 있으며, 지속적인 “코드 리뷰”를 활성화 해서 “구현 지식을 공유”하려는 노력을 해야한다.

# [ 글쓰기에서 도식화로 ]

글은 사람의 이해를 돕는 데 필요하다. 특히, 쓰는 사람과 읽는 사람이 거리나 시간상으로 차이가 있을 때 사용할 수 있는 유용한 소통 도구다. 하지만, 글로 적혀 있는 것을 읽고 이해하기는 쉽지 않으며 기억하기도 어렵다. 가장 좋은 방법은 역시 도식화(혹은, 시각화)하는 것이다. 어떤 것이든 도식화해서 그림으로 만들려는 노력이, 보는 사람의 이해를 돕고 기억도 오래 가도록 만들어 줄 것이다.

도식하는 그림을 구성하는 요소와 정의된 요소들간의 관계를 표현한다. 각각의 요소는 내부 상태를 가질 수 있으며, 요소들간에 대화를 위한 창구도 가질 수 있다. 대화는 동등한 관계에서 이루어지 지며, 어떤 요소는 다른 요소를 포함하거나 지정할 수 있다. 또한, 특정 요소는 자신을 확장해서 다른 요소를 만든데 사용되기도 한다. 일을 처리하기 위해서는 시스템을 구성하는 요소들의 협업이 필요하며, 각각의 요소는 주어진 역할과 그에 따르는 명확한 책임 범위를 가진다. 따라서, 자신의 역할 범위를 넘어서는 일은 다른 요소에게 의뢰하는 것으로 만족한다.

사실, 위에서 이야기한 것들을 UML(Unified Modeling Language)이 이미 잘 하고 있다. 하지만, 여기서 중요한 것은 UML을 사용해서 도식화하는 것이 아니라, 자신이 해결하고자 하는 문제를 도식화(모델링)해야 한다는 점이다. UML은 도구일 뿐 목적이 아니기에, 상호 간에 동의한 방법을 통해서 도식화할 수 있으면 충분하다. 도식화한다는 것은 문제의 상세한 부분을 겆어내고 풀어야 할 문제에 집중할 수 있도록 만들어 준다. 따라서, 모델의 역할은 문제를 설명하고 해결 방안을 찾아나가기 위한 방법을 제공하는 것이라고 볼 수 있다. 물론, 최종 모델은 문제의 해결책이 될 수 있겠지만, 중간 단계에서 만들어지는 많은 그림들은 생각을 효과적으로 공유할 수 있도록 만들어 준다.

그림으로 표현하는 것은 상호 관계를 이해하는 데 큰 도움이 된다. 글로는 표현하기 어렵거나 이해하기 어려울 때 그림을 그릴 필요가 있는 것이다. 즉, 글로는 부족한 내용을 그림으로 채워줄 수 있도록 해야 한다. 대부분 문서는 그림이 있으면 글이 없거나, 어려운 글이 있으면 그림이 없는 경우가 종종 있다. 잘 쓴 글이라면 남들이 쉽게 읽을 수 있고, 주장하려는 바를 명확히 전달할 수 있어야 한다(물론, 기술에 한정해서). 글과 그림은 서로 부족한 부분을 보완해주고 균형을 이루어 주는 역할을 한다. 어느 하나에 집중하는 것 보다 둘 다 동시에 적절하게 사용하는 것이 효과적인 의사소통 방법인 것이다.

# [ 달을 가리키는데, 손끝을 보지말라!!! ]

흔히들 "뒷다리를 잡는다."라는 말로, 자신이 하고자 하는 것이 옳은데, 남들이 자꾸만 이런저런 이유를 대면서 방해를 한다고 이야기한다. 이런 경우는 일을 하다 보면 흔히 발생하는 것으로, 특히 현재 담당자가 있는 일을 개선하려고 할 때 부서 간의 의견충돌에서 많이 생긴다. 즉, 이미 열심히 일하고 있는데, 잘 모르는 사람이 와서 이래라 저래라 하면서 개선을 요구하니, 바빠서 죽을 지경인데 질책을 당하는 느낌이 들기 때문이다. 아무리 좋은 말도 상황에 따라 반가운 소식이 될 수도 있고, 혹은 그 반대인 부담이 될 수 있다.

그렇다고, 항상 분위기 잘 파악해서 이리저리 회피하라는 것은 아니다. 그렇게 일하면 자기 일에 대한 자신감은 없어지고, 그냥 남의 눈에 보기 좋은 결과만 만드는데 집착하게 된다. 중요한 것은 "달을 가리키는데, 달은 안 보고 손끝만 보는 사람"은 되지 말라는 것이다. 하려는 일의 최종 목표가 옳다고 판단되면 그것에 동참할 생각을 해야지, 이런저런 이유를 들어서 다른 사람이 하는 일을 방해해서는 안 된다. 그리고, 더 중요한 것은 자신이 하는 일에 대해서는 자신이 최고의 전문가일 수는 있지만, 그 외의 다른 일에 대해서도 전문가인 척을 해선 곤란하다.

소프트웨어 개발에 있어서의 최종적인 목표는 무엇인가? 당연히 이것에 대한 답은 "품질 좋은 코드로 사용자의 요구사항을 만족하게 하는 동작하는 소프트웨어"다. 사용자의 만족은 측정하기 어려우니, 일단 품질 좋은 코드 만이라도 먼저 보도록 하자. 품질 좋은 코드를 만들어내기 위해서, 일반적인 공산품의 생산의 과정을 보는 것이 도움이 될 것이다. 물건을 만들 때는 부속품부터 하나씩 규격을 만족하는지 본다. 부속이 규격을 만족한다면 하나씩 조립을 하고, 조립된 모듈 단위로 다시 규격을 만족하는지 본다. 이렇게 반복적으로 더 큰 단위의 결합이 발생하고, 더 큰 단위의 검사가 시행된 후에 최종적인 제품이 나오게 된다. 최종적인 제품은 다시 사용자가 돈을 주고 살 만한지 다시 한번 검증받게 되며, 얼마나 오래 사용할 수 있는지도 검사된다.

여기서 중요한 것은 만드는 과정에서 실행되는 품질 측정이다. 소프트웨어도 마찬가지다. 만들어 가면서 끊임없이 제대로 규격을 만족하는지 검사해야 한다. 과정 중간 중간에 필요한 테스트를 해야 하고, 또한 점진적인 통합에서도 검사는 수행된다. 최종적인 결과물에 대해서 통합 시험과 사용자의 인수 여부를 결정한 수락(Acceptance) 검사도 하게된다. 따라서, 소프트웨어 개발에 있어서 최종 목표를 만족시키기 위해서는 각 단계별로 있는 테스트를 어떻게 하는가에 따라 결과물의 품질이 유지된다고 볼 수 있다. 물론, 대량 생산을 목표로 하는 일반적인 제품과 양산 과정이 없는 소프트웨어 제품은 다르다고 이야기할 수 있을지 모르지만, 그렇다고 과정이 없는 제품 개발이 가능할리도 없다.

품질을 확보했다고 이야기할 때, 소프트웨어에는 무엇이 기준이 될 수 있을까? 흔히 "측정할 수 없다면, 개선할 수 없다"라는 말을 많이 들었을 것이다. 그렇다면, 무엇을 측정하고 무엇을 개선할 것인가? 측정이라는 말에는 기본적으로 "수치적인 표현” 이라는 것이 녹아 들어가 있는 개념이다. 즉, 수치로 나타낼 수 있는 표현을 사용해야지만, 사람들의 믿음이 생긴다는 것이다. 믿음은 자신감이 바탕이 되며, 이러한 자신감은 소프트웨어의 품질에 대한 시각적인 지표를 이용해서 만들어져야 한다. 따라서, 우리는 개발하는 동안 파악할 수 있는 모든 것 들을 지표화시켜서, 해당 지표를 개선하는 방향으로 일을 해야 한다는 결론에 도달하게 된다. 지표가 좋다면 품질을 유지하고 있는 것이며, 그렇지 않다면 무엇을 개선해야 할지 결정하면 된다.

소프트웨어개발에서 지표화시킬 수 있는 것들에는 무엇이 있을까? 가장 쉬운 것은 자동화 도구로 만들어 낼 수 있는 것들이다. 다음과 같은 것들이 있으니 참고로 사용해도 된다. 물론, 이것은 개인적인 생각으로 만든 것이며, 과제에 적합한 것들으로 변경해도 무방하다.

**01. 중복코드가 얼마나 있는가?**

**02. 코딩 룰을 따르지 않는 코드가 얼마나 있는가?**

**03. 컴파일 시 발생하는 경고 메시지는 얼마나 있는가?**

**04. 상호 참조가 발생하는 경우는 얼마나 있는가?**

**05. 코드의 복잡도가 얼마나 되는가?**

**06. 길이가 긴 함수나 클래스의 비율은 얼마나 되는가?**

**07. 얼마 정도의 코드를 테스트가 커버(Cover) 하는가?**

**08. CPU 점유율을 얼마나 되는가?**

**09. 시스템의 반응 속도는 얼마나 되는가?**

**10. 필요한 메모리는 얼마나 되는가?**

물론, 이것이 전부는 아니다. 개인적인 생각으로 소프트웨어의 내부 및 외부 품질이라는 것을 측정하는데 필요한 몇 가지 만 나열한 것이다. 일부는 소스코드 자체의 품질이며, 어떤 것들은 전체적인 성능과 관련된 것들이다. 어쨌든 자신이 만들고자 하는 소프트웨어가 품질이 좋다는 것을 나타낼 수 있는 간접적인 지표로 삼을 만한 것들이다. 코드의 품질을 직접적으로 표현하는 것이 어렵기 때문에, 간접적으로 표현할 수 있는 여러가지 지표를 통해서 원래의 품질을 예상하는 것이다.

이런 것들을 이야기할 때 한가지 조심할 것은 버그의 개수가 테스트에서 많이 발견되었다고 나쁜 품질의 소프트웨어를 만들었다고 이야기하는 경우다. 사실 모든 소프트웨어는 버그를 가지고 있으며, 단지 내재하고 있는 버그가 표출되지 않은 경우에 대해서 “제한된 검증”이 완료된 것일 뿐이다. 테스트가 고도화될 수록 찾아내는 버그는 늘어날 것이고, 결과적으로 그러한 버그가 수정되기에 최종 사용자가 발견하는 버그는 줄어들기 때문이다. 하지만, 중복되는 버그는 주의해서 봐야 한다. 그러한 버그는 구조적인 문제점을 내포한 경우가 많고, 급하게 “땜빵”해서 고쳐진 경우가 많을 것이기 때문이다.

소프트웨어는 단순하지 않다. 가장 단순한 소프트웨어라고 할지라도 사람이 만든 것이고, 사람의 사고를 따라잡는 기계는 아직 나오지 않았다(물론, 특정 분야에서는 그런 기계가 있을 수 있지만). 단순하지 않은 문제는 나누어서 해결하는 것이 좋으며, 각각의 나누어진 부분들에 대한 검증은 전체 해결책을 찾는데 대한 훌륭한 지표가 된다는 점을 잊어서는 안될 것이다. 단순하지 않기 때문에 그 만큼 더 많은 검증 노력이 필요한 것이다.

# [ 코딩 전략 ]

지금까지 여러 가지의 코딩에 대해 이야기를 했지만, 아직 해야 할 말은 많은 것 같다. 학교에서 하는 코딩과 실무에서 접하는 코딩 간에는 많은 차이가 있기 때문이다. 실무에서의 코딩은 정상적인 상황보다는 이상이 발생했을 때 어떻게 대처해야 하는지를 더 중요하게 여긴다. 이상 동작을 하는 상황에서도 시스템을 항상 안정적으로 유지해야 하기 때문이다. 여기서는 몇 가지 팁(Tip)만을 이야기하겠지만, 기본 가정은 시스템은 안정적이지 않을 수 있다는 것을 항상 염두에 두기 바란다.

**01. 모든 함수는 실패할 수 있다**.

함수는 실패할 수 있다는 가정에서 출발해야 한다. 시스템 호출(System Call)로 제공하는 함수도 실패를 한다. 예를 들어, 용량이 모자란 저장 장치에 대한 쓰기(Write) 요청은 실패할 확률이 높다. 물론, 실패하지 않고 대기(Wait)를 통해서 해결할 수 있겠지만, 실패가 발생할 수도 있다는 점은 항상 대비하고 있어야 한다. 실패가 발생했을 때 가장 빠른 오류 회복은 실패를 보고하는 것이다. 즉, 오류 번호를 이용해서 호출자에게 실패의 원인을 알려주는 것이다. 실패가 발생했을 때 호출자에게 실패한 사실을 알려주지 않을 수도 있다. 실패가 시스템에 아주 위험한 상황이 아니고, 쉽게 오류를 회복할 수 있다면 사용자가 모르게 시스템에서 오류를 처리하고, 원래의 상태로 복귀하는 것도 괜찮은 방법이다. 중요한 것은 실패할 수 있다는 가정을 항상 머릿 속에 두어야 한다는 것이다.

**02. 모든 호출자는 잘못된 값으로 호출을 시도할 수 있다.**

프로그램을 디버깅하는 상황에서 잘못된 값을 가지고 호출하는 호출자를 보는 경우가 많다. 이때는 잘못된 값이라는 것을 호출자에 알려주어야 한다. 이를 위해서 넘겨받은 파라미터 리스트(Parameter List)의 값을 일일이 검사하는 코드가 들어가야 한다. 물론, 이때 "#if XXXX "와 같은 컴파일 옵션 처리로 디버그 모드에서 만 검사하는 방법을 사용하기도 한다. 즉, 최종 배포되는 바이너리 이미지에는 포함하지 않도록 만든다. 이미 검증이 완료된 코드에 대해서 속도를 향상시키고자 하는 의도도 있다. 하지만, 만약 호출자가 의도하지 않은 호출을 할 가능성이 조금이라도 있다면, 당연히 올바른 값으로 호출하는지 반드시 확인하는 절차를 코드로 만들어야 할 것이다.

**03. 모든 상숫값은 의미를 명확히 밝혀야 한다.**

코딩은 알게 모르게(?) 맺게 되는 계약(Contract)이다. 따라서, 이런 계약은 반드시 명확해야만 서로 오해할 여지가 없다. 코드에서 흔히 보는 일반적인 상숫값들은 사람이 과도하게 생각하도록 만드는 주범이다. 다른 곳에서 어떤 값으로 선언되었지는 의미를 모르기에 코드를 보는 사람에게 혼돈을 준다. 따라서, 될 수 있다면 모든 상숫값은 반드시 의미를 가지는 이름(Name)으로 대체 해야 한다. 아무리 사소한 것이라도, "0, 1, -1" 보다는 “False, True, Error”와 같은 값으로 하는 것이 프로그래머의 의도를 명확히 하기 때문이다. 계약은 명확할수록 위반하지 않을 가능성도 높아진다.

**04. 모든 코드는 테스트되어야 한다.**

개발자가 사실상 가장 어렵다고 생각하는 것이 바로 테스트다. 어떻게 해야 할지도 모르고, 알고 있다고 해도 주어진 시간 내에 하기도 어렵다. 테스트를 가장 쉽게 하는 방법은 개발(코딩 과정)에 테스트를 포함하는 것이다. 개발과 테스트가 다르다고 생각하는 순간 테스트는 물 건너가고 반복된 “테스트-디버깅” 과정만 남게 된다. 디버깅을 줄이는 것이 테스트다. 테스트는 그 자체로 디버깅에서 해야 할 일을 줄여준다.

개발자가 하는 테스트는 “단위 테스트” 및 “통합 테스트”다. 이 둘은 개발에서 완료해 주어야만 “반복된 테스트-디버깅” 시간이 줄어든다. 모든 코드를 전부 실행하는 테스트를 하는 것은 어려운 일이다. 하지만, 100% 완벽하기 보다는 70%라도 테스트된 부분을 가져가는 것이 좋다. 오늘은 70%이지만, 다음에는 그보다 나은 테스트 범위를 가져갈 수 있을 것이다. 구체적인 방법을 알고 싶다면, TDD(Test Driven Development)나 단위 테스트 관련 프레임워크를 찾아보기 바란다.

**05. 모든 오류는 처리될 수 있어야 한다.**

시스템은 안정적으로 동작해야 하며, 사용자는 어떤 오류에 대해서도 안정적으로 동작하는 시스템을 구매하기를 원한다. 오류로 부터 회복이 중요한 이유는 사용자의 개입이 최소화 된다는 것이다. 심각한 오류로 인한 복구는 리부팅(Rebooting, Reset) 밖에 없는 경우가 있다. 이때도, 어떤 이유로 리부팅 하는가를 로그(Log)로 남겨서 나중에라도 볼 수 있어야 한다.

간단한 오류의 회복은 해당 동작이 시작되기 전으로의 상태 복구가 있을 수 있다. 따라서, 시스템에 로그를 남기는 것도 몇 단계로 나누어서 처리하는 것이 좋다. 사용자에게 간단한 정보 만을 전달하는 경우도 있고, 디버깅을 목적으로 하는 경우도 있다. 따라서, 최소한 4단계 이상의 로그는 남길 수 있어야 하며, 이 중에서 디버깅 목적의 로그는 출시에서 제외하는 것이 좋다. 잣칫 잘못하면 로그를 통해서 시스템을 공격할 가능성도 있기 때문이다.

이상과 같이 몇 가지를 살펴보았지만, 코딩은 그리 쉬운 것이 아니다. 특히, 상업적인 성공을 중시하는 소프트웨어인 경우에는 사용자들의 입력이 어떻게 될 것인지, 혹은 시스템이 설치되어 운영되는 환경에 대한 고려가 필요하다. 죽지 않는 시스템을 만드는 것이 소프트웨어 개발자의 최종 목적이어야 하며, 죽더라도 “우아하게 죽는” 시스템을 만들도록 노력해야 한다. 아무런 말도 없이 그냥 죽는 시스템은 마치 연인이 떠나갈 때 이유 없이 "헤어지자"라는 말을 남기는 것과 다르지 않다.

# [ 싱글턴(Singleton)에 대한 생각 ]

싱글턴은 개발자들이 흔히 코딩에서 사용하는 디자인 패턴(Design Pattern) 중에 하나다. 싱글턴을 이용하면 시스템에서 정해진 수만큼의 개체(여기서는 Object보다는 Entity라는 개념이 더 어울린다고 본다)를 생성해서 전역적으로 사용할 수 있다. 예를 들어, 시스템에 단지 하나의 개체만 존재해야 하고, 다른 부분에서 이 개체를 사용해서 일을 처리해야 할 경우, 개체를 싱글턴으로 생성하게 된다. 즉, 생성된 개체 이외에는 동일한 것이 없다는 가정이다.

물론, 딱 하나만 존재하는 개체에 대해서만 싱글턴을 사용하는 것은 아니다. 여러 개의 개체가 있더라도 시스템 전체에 정해진 수만큼만 있어야 한다면, 싱글턴을 이용해서 한정된 개체들을 관리할 수 있다. 이때 각각의 개체에 대한 직접적인 생성은 할 수 없으며, 처음 생성될 경우에 그 수도 정해지게 된다. 또한, 생성된 개체들을 얻을 수 있는 방법도 제공된다. 생성된 개체를 사용하기 위해서는 반드시 제공되는 방법을 통해서만 써야 할 것이다.

하지만, 부주의하게 싱글턴을 사용하게 되면 나빠지는 부분이 있다. 사람들이 이처럼 유용한 싱글턴을 너무 많이 사용하는데서 나타나는 문제점이다. 즉, 마치 전역(Global)으로 사용할 수 있도록 만든 변수처럼 프로그램의 여러 부분에서 싱글턴을 사용하는 경우다. 전역변수나 함수를 가지고 있다는 것은 사용하기 편하다는 장점이 있지만, 공통 자원을 사용하는 코드들이 결합력이 커지는 효과가 생긴다. 즉, 코드 간에 의존성이 강해진다. 여러 곳에서 효과적으로 호출(사용) 할 수 있어서 편하기는 하지만, 결과적으로는 프로그램의 전체적인 의존성을 높이게 되어, 한 곳에서 발생한 변경이 다른 부분에 영향을 줄 가능성이 높아진다. 따라서, 이런 식으로 구성된 프로그램은 디버깅과 기능 추가 및 변경 등에 있어서 제한적일 수밖에 없다.

한가지 더 우려가 되는 부분은 여러 관련이 없는 자료구조와 함수를 싱글턴으로 묶어서, 마치 유틸리티(Utility)성 함수들의 모음으로 만드는 경우다. 일종의 전역 도움 함수(Global Helper Function)처럼 코드의 어떤 곳에서도 가져다 사용할 수 있도록 만든다. 싱글턴이 유용한 것은 개체를 생성하는 방법을 제한하고, 사용하는 방법도 일정하게 제약할 수 있기 때문이지만, 이런 식으로 글로벌 자료구조를 늘려가는 것은 올바른 선택이 아니다. 정말 그런 함수들이나 자료구조가 필요하다면, 함수와 자료구조를 관련된 부분들로 나누어 새로운 클래스를 만들어서 정의하고, 각각이 있어야 할 위치를 전체 아키텍처에서 찾아주는 일을 먼저 해야한다. 쉽게 코딩하면 쉽게 망할 수도 있다는 점을 알아야 할 것이다.

결론적으로 말하자면, 전역 변수나 전역 함수처럼 쓰기 위해서 만드는 싱글턴 보다는 정말 시스템에 제한된 수의 개체들을 관리하기 위해서 싱글턴을 사용해야 한다. 싱글턴은 리소스나 전역 자료구조에 대한 생성 및 접근 제어를 통해서 “일관된 관리”를 목적으로 사용되어야 한다. 단지, 싱글턴을 사용했기 때문에 “안정된 코드”라거나, 혹은 “디자인 패턴에 의한, 디자인 패턴을 위한, 디자인 패턴의” 코드를 만들어서는 안된다. 쓰기 쉽다고 오용하게 된다면, 그 만큼 치러야 하는 댓가도 커진다. 디자인 패턴은 문제의 유형과 풀이 방법의 일반화된 특성을 나열한 것일 뿐, 모든 문제의 해결책은 아니다. 소프트웨어의 가장 기본 원리인 “약한 결합(Decoupling)”과 “높은 응집성(High Cohesion)”이 디자인 패턴을 적용하기 이전에 만족되어야 한다.

# [ 코드에서 발생하는 역참조 ]

소프트웨어 개발에서 가장 중요한 부분은 요구사항 분석과 설계 과정이라고 한다. 물론, 이 말에 대해서는 100% 공감한다. 제대로 분석도 하지 않고, 어떻게 구현할지도 정하지 않은 상태에서 코딩한다는 것은 말이 안 되기 때문이다. 무엇을 해야 할지도 모르는데 구현할 수 있겠는가? 하지만, 여기서 한 가지 간과하는 것은 구현(Implementation)의 중요성에 대한 강조가 없다는 것이다. 소프트웨어 개발자는 어쨌든 앞에서 진행한 요구사항 분석과 설계를 구현해야지 결과를 얻을 수 있다. 구현은 앞으로 코드를 유지보수 해야 할 사람들에게 요구사항 분석이나 설계보다 더 현실적인 문제이기 때문이다.

구현의 정석을 여기서 전부다 논의하는 것은 사실 불가능하다. 아마도 책으로 한 권을 써도 다 하지 못할 것이다. 그리고, 대부분 그런 것을 다룬 책에서는 다음과 같은 말을 쉽게 찾아볼 수 있다. 즉, “좋은 코드란 어떤 것인가?”와 같은 질문에 대해서, "남이 쉽게 읽을 수 있는 코드"라고 대답한다. 당연히 남이 쉽게 읽고 수정할 수 있는 코드에는 몇 가지 전제 조건이 있다. 그 중에서 가장 중요한 것은 "참조"의 문제다(물론, 변수의 이름이나, 함수의 길이, 의미를 가지는 이름의 사용, 주석 등등 다양한 것들이 있지만).

참조란 한 모듈이나 클래스에서 다른 모듈이나 클래스에 속한 값, 변수, 함수, 메소드 등과 같은 것을 이용하는 데서 발생한다. 참조가 역방향을 취한다는 말은 “A”와 “B”가 서로 참조하고 있을 때, 혹은 “A->B->C->A”와 같은 순환 참조가 발생하는 경우다. 즉, 상위 하위로 구분한 계층구조에서 하위의 구성요소가 상위에 정의된 코드를 접근한다는 뜻이다. 그리고, 그러한 접근의 구조가 체인(Chain)과 같은 형태가 되어, 서로 영향을 주고받는 관계를 이루게 된다.

이와 같은 문제가 생기는 원인은 다양하겠지만, 원인은 "손쉬운 구현의 유혹"이다. 즉, 쉽게 지금의 문제를 개선(혹은, 새로운 기능을 추가)하려고, 다른 부분에서 해결해야 할 일을 직접 자기가 처리하려고 할 때 발생한다. 혹은, 이미 구현된 코드를 쉽게 끌어당겨서 사용하려고 할 때도 포함할 수 있다. 이를 해결하기 위해서는 정확히 자신이 담당한 일(정의된 역할) 만 처리하고, "일을 직접 하지 말고, 요청하라"처럼 자신에게 해당하지 않는 일은 처리를 담당하는 부분에 위임(delegation) 해야 한다. 사용하는 자료구조와 함수의 위치 잘못된 경우에는 새로운 계층이나 모듈(혹은, 클래스)로 만들어서 분리해서 사용해야 한다.

물론 모든 상호 참조가 역참조이지는 않다. 하지만, 서로 간에 참조가 많아지면, 둘이 아닌 하나의 덩어리로 취급하는 것이 일반적이다. 즉, 나누어 구현되어 있지만, 실제로는 하나의 역할을 하는 모듈일 가능성이 높은 것이다. 물론, 그렇게 구현하면 크기가 큰 모듈 들이 늘어나게 되며, 최악의 경우에는 마치 한 덩어리처럼 동작하게 된다. 따라서, 이런 경우에는 모듈이라는 개념이 사라지고, 그냥 덩치 큰 소프트웨어만 존재할 뿐이다. 당연히 이러한 모듈은 유지보수가 힘들고, 기능 추가도 쉽지 않을 것이 자명하다.

문제는 항상 작게 나누어서 해결하는 방법으로 접근해야 한다. 물론, 전체적인 최적화의 관점에서도 생각해야 하는 것도 맞는 말이다. 전체적인 최적화의 관점에서 문제를 파악하고, 이를 다시 나누어서 각 부분에서 해야 할 일을 정의해 나간다면, 성능과 분리를 함께 만족할 수 있는 구조를 만들어 낼 수 있다. “최적화된 코드는 구조화가 불가능 하지만, 구조화된 코드는 최적화가 가능하다”는 것이 일반적으로 알려진 사실이다. 성능은 주로 대량의 데이터 이동이나 계산에서 발생하는 것이며, 이를 하나로 만들어서 전체로 부터 고립시켜 해결한다. 여러 개의 요소가 결합하여 “주요 경로(Critical Path)”를 생성한다면, 그 부분만 함께 묶어서 최적화하는 것이다. 그렇다고, 최적화를 위해서 역참조가 용납되는 것은 아니다. 이 때도 다양한 방법을 이용해서 역참조는 제거하는 것이 최적화를 위한 준비 작업이다. 나중에 있을 더 나은 개선을 위해서도 꼭 필요하기 때문이다.

역참조 문제는 사람의 사고와 관련성이 있다. 즉, 설계 없는 구현과 지속적인 기능 개선을 통해서 점차 코드가 엉망이 되어가기 때문이다. 역참조가 많은 코드는 한 부분의 변경이 자신에게 만 영향을 주는 것이 아니라, 순환 구조를 만드는 모든 부분에 영향을 준다. 순환 참조는 문제를 찾기 어렵게 만들고, 전체 코드는 서서히 망가져(썩어: Rotten) 갈 것이다. 어느 시점이 되면 더는 기능 추가나 개선을 할 수 없는 지경에 이르게 되기 때문에, 이 때까지 문제를 남겨두는 것은 언제 터질지 모르는 “폭탄 돌리기”와 같다. 설령 자신의 차례가 돌아오지 않을지도 모르겠지만, 옆에 있는 다른 동료가 피해를 볼 수 있다는 것을 알아야 할 것이다.

# [ 테스트 케이스에 대한 생각 ]

테스트 케이스(Test Case:TC)는 어떤 기능이 제대로 동작하는지, 혹은 제대로 동작하지 않는 상황을 검증하는데 필요한 예(Example)이다. 이러한 테스트 케이스를 늘리면 과연 사용자가 발견하는 버그의 수가 줄어들 수 있을까? 그리고, 테스트 케이스를 무한정 늘리는 것이 가능할까? 또한, 그렇게 만들어진 예를 전부 다 사람이 수행하는 것이 올바른 일일까? 이것에 대한 답은 의외로 간단하다. 사람이 하는 일은 아무리 완벽하게 하더라도 빈틈이 있을 수 있으며, 단순히 수를 늘린다고 해서 모든 버그를 다 찾을 수 있는 것도 아니다. 더군다나 사람이 직접 수동으로 테스트를 해야 한다면, 거의 테스트만으로 개발 일정의 대부분을 다 써야 할지도 모른다.

그렇다면 해결 방법은 없을까? 가장 먼저 생각해야 할 방법은 테스트를 자동화하는 것이다. “자동화를 어떻게 할 수 있을까?”에 대한 해답은 소프트웨어 구현마다 다르다. 따라서, 자신이 처한 상황에서 최대한 자동화할 수 있는 방법과 도구를 찾아야 한다. 여기서 중요한 것은 사용자 시나리오와 그것을 기록하고, 되풀이해서 실행할 수 있는가를 검토하는 것이다. 자동화를 구축할 때도 될 수 있으면 테스트를 완전히 수행하는데 필요한 시간이 짧아야 한다. 며칠 동안 실행해도 결과가 나오지 않는다면, 이미 개발은 저 멀리 가고 있을 것이기 때문이다.

테스트 케이스를 작성하는 것도 요령이 있다. 즉, 절대 나올 수 없는 조합을 찾아서 먼저 없애는 것과 하나의 테스트 케이스를 통해서 여러 부분을 검증할 수 있는 것을 찾는 것이다. 테스트를 잘 하는 사람은 버그 유발 가능성이 높은 부분에 대한 정확한 이해를 하고 있다. 만들려는 제품에 대한 정확한 이해를 가지고 있다면, 이런 부분에 대한 버그 찾기도 쉬울 것이다. 테스트 케이스의 수자만 늘리는 사람은 아무리 많은 수의 TC를 가지고 있어도 한정된 범위의 버그 만 찾을 뿐이다.

어쨌든, 테스트 케이스는 개발을 위해서 필요한 것이고, 개발에서는 테스트를 자동화시킬 수 있는 방법을 만들어내는 것이 중요하다. 테스트라고 해서 모든 것을 품질 관련 부서의 업무로 남긴다면, 개발이 정말 해야 할 일을 하지 않는 것이다. 자신이 만든 버그를 다른 사람의 손에 맡겨둔다는 것은 말이 되지 않는다. 따라서, 개발은 최대한 자신이 개발한 내용에 대해서 확신을 가질 수 있도록 테스트를 실행해야 하고, 검증된 코드를 품질부서로 넘겨 자신이 찾지 못한 버그를 찾을 수 있도록 해 주어야 한다. 품질 부서는 개발에 도움을 주는 부서이지, 시시콜콜한 문제를 가지고 “딴지”를 거는 부서가 아니다. 그들의 버그를 찾는 노력이 고수준이 될수록, 개발도 수준 높은 일을 할 수 있다는 것을 알아야 한다.

# [ 성능(Performance) vs. 구조(Structure) ]

우리는 어떤 일을 할 때 항상 성능적인 요구사항을 먼저 고려하게 된다. 측정해야 할 부분도 주로 수치로 나오는 데이터가 많은 부분에 치중하게 된다. 기존에 작성된 코드를 변경하려고 할 때, 가장 먼저 저항에 부닥치는 부분이 바로 성능에 관련된 것들이며, 이미 검증이 되었다고 하는 자기 방어적인 논리로 변경을 거부하는 것이 개발자들의 기본 태도다. 그렇다면, 검증된 코드는 변경해서는 안 되는가? 혹은, 성능만 나오면 모든 문제가 사라지는가? 성능은 시스템이 가져야 할 비기능 요구사항 중 하나일 뿐이며, 쉽게 순위가 바뀌는 항목 중에 하나일 뿐이다.

소프트웨어 문제의 대부분은 “유지보수”와 “확장성, 변경 용이성”에 있다. 유지보수를 위해서는 잘 정리된 코드가 아무래도 더 효율적일 것이고, 확장성이나 변경 용이성을 위해서는 모듈화가 잘 된 코드가 당연히 더 좋을 것이다. 하지만, 막상 이렇게 이야기할 때는 동의 하다가도, 개선된 코드를 사용하는 것에 대해서는 거부감을 가지는 것이 또한 개발자다. 왜? 자신이 만든 코드가 아니라는 점이 가장 큰 걸림돌로 작용하기 때문이다. 즉, 자신이 코딩하지 않은 코드는 사용하지 않겠다는 말을 둘러서 하는 것일 뿐이다. 자신이 작성한 코드만 믿는다면, 사실 모든 코드를 자신이 개발해야 옳을 것이다. 하지만, 소프트웨어 개발에서 그런 일이 발생하는 경우는 거의 없다.

자신이 만들지 않고도 사용하고 있는 코드가 얼마나 많은지 먼저 보기 바란다. 사실 소프트웨어 개발이란 90%에 가까운 코드는 이미 다른 사람이 작성한 것을 가져오고, 10% 정도의 코드만 자신이 작성해도 대단히 성공적이다. 예를 들어, 리눅스 운영체제는 자신이 만들지 않았지만 잘 사용하면서, 옆에 있는 팀의 다른 사람이 개선한(혹은, 작성한) 코드를 한사코 사용하지 않겠다는 것이 논리적으로 타당한가?

운영체제라서 그렇게 사용해도 된다고 할 지도 모르겠지만, 그렇다면 SQLite3와 같은 DB(데이터 베이스)는 왜 사용할까? 그것도 자신이 개발한 것이 아니라면, 스스로 DB를 만들어서 써야 할 것이다. 이런 식으로 하나씩 따져 묻는다면, 결국 자신이 만든 코드는 전체 제품의 코드에서 정말 하찮은 부분 밖에 되지 못한다는 사실을 깨닫게 될 것이다. 안타까운 것은 그런 사실을 고민조차 해본적이 없다는 것이다. 이런 저런 것들을 이유로 말 할지라도, 결국 이유는 “당신이 만든 코드를 사용하기 싫다”라는 것밖에 없다.

사실 회사에서 일이 힘들어서 그만둔다는 사람은 별로 없다. 대부분의 경우 “사람이 싫어서 그만둔다”. 이렇게 꽉막힌 사람들과 일하는 것은 정말 힘들다. 일보다 사람이 문제가 되는 경우다. 물론, 그렇다고 그런 사람들이 특정 회사에만 있는 것도 아니다. 어디를 가도 그런 사람들을 한 두명은 만날 수 있다. 회사를 옮겨도 꼭 누군가는 그런 역할을 하고 있다. 그리고, 성능적인 문제보다 구조적인 문제를 이야기할 때면, 소위 말하는 “전문가”라는 타이틀을 쥐고 거부의사를 강하게 드러내는 것도 이런 사람들이다.

기존(Legacy) 코드를 잘 활용하는 전략은 중요하다. 즉, 자신이 코딩하지 않고 이미 많이 검증된 코드를 이용하는 것은 효율적인 개발방법이다. MFC로 코딩하면서 모든 UI코드를 손수 만드는 것은 효율적이지 않다. 오히려, “잘 만들어진 것”을 구입해서 이용하거나, WPF등을 사용하는 것이 효과적인 개발방법이다. DB나 파일시스템(File System)을 새로 만드는 것은 자신이 지금 하고 있는 과제보다 더 큰 일 일 수 있다. 개발 기간도 오래 걸리고 검증도 오래 걸리기 때문이다. 또한, 새로 만든다고 해서 기존 코드보다 월등히 좋다고 보장하는 것도 아니다.

문제는 기존의 코드가 “잘 만들어진 것”이어야 한다는 점이다. 그리고, 많은 검증을 거쳤으며 변경할 필요도 거의 없어야 한다는 것이다. 그렇지 않다면, 기존의 코드가 가진 문제점이 새로 작성되는 코드와 함께 문제 만 더 키울 뿐이다. 기존의 코드가 “가독성이 있고, 역할과 책임의 분리가 명확하며, 의존성이 작다면” 충분히 좋다고 생각할 수 있을 것이다. 하지만, 사용하기 복잡하고, 조그마한 코드 수정에도 쉽게 버그가 발생한다면, 이는 “재사용 전에 구조적인 개선” 부터 해야 할 것이다.

구조를 이야기하면 이론에만 밝은 사람으로 오해하는 경우가 종종 있다. 양산 경험 없이 이론 만 밝다고 비난받는다. 하지만, 그 이론이 무엇인지는 제대로 알아보려고도 하지 않는다. 대부분의 소프트웨어에서 발생하는 문제들은 아주 기초적인 구조화 잘못임을 받아들이지 못하는 것이다. 예를 들어, 자신의 소프트웨어가 가지고 있는 구조를 계층적인 아키텍처라고 이야기하는 과제는 계층화가 잘못된 경우가 많다. 적절한 추상화 및 모듈화를 고려하지 않고 개발된 경우가 많으며, 책임과 역할의 구분이 모호해서 하나의 모듈이 다양한 역할을 하는 경우도 있다.

새로운 기능을 추가할 경우, 수정해야 할 부분이 전체 모듈로 퍼져나가는 경우도 있으며, 모듈 내부를 수정했는데도, 엉뚱한 다른 관련없는 모듈에서 문제가 발생하기도 한다. 객체 지향 언어를 사용하면서도 정작 객체가 지녀야 할 특성은 모르며, 상속을 해야할 경우와 하지 말아야 할 경우를 구분하지 못한다. 단지, 객체지향 언어를 사용하기 때문에, 객체지향적으로 구현했다고 자랑스럽게 이야기한다. 무엇을 객체로 만들지, 어떤 정보를 외부에 보여주어야 할지, 인터페이스를 이용한 관심의 분리(Seperation of Concern)은 무엇인지는 관심도 두지 않는다. 단지 빠르고 편하게 구현하는데 집중할 뿐이다.

그렇다면, 정말 이론이 성능과는 아무런 관련이 없는 것일까? 교과서도 모르는 상태에서 수능 시험을 잘 칠 수 있다고 정말 믿는 것일까? 학원만 다녀서 학교는 필요없다고 이야기하는 것일까? 아무도 그렇게 말하지는 않을 것이다. 이미 만들어봤고 경험했기 때문에 “교과서에 적혀 있는 이론들이 제대로 동작하지 않는다”고 믿는다는 뜻이다. 하지만, 정말 그럴까? 아마도 그렇게 이야기하는 대부분의 사람은 제대로 교과서데로 일해 본 경험이 없을 것이다. 누군가의 어깨 너머로 슬쩍 본 지식이나, 서점에서 산 “기초 문법”정도 만 보고 실무에 뛰어 들었을 것이다. 이것은 몰라서 못하는 것이 아니라 자만심이 눈을 가리고 있는 것이다.

비기능 요구사항인 성능이 구조를 정의하는데 핵심적인 역할을 하는 것은 맞는 말이다. 하지만, 그렇게 만들어진 구조가 소프트웨어 개발의 핵심 원리를 위반할 경우에는 유지보수와 확장성, 변경 용이성과 같은 또 다른 비기능적인 요구사항은 만족시키지 못한다. 적절한 트레이오프가 있어야 함은 맞지만, 특정 요구사항으로 인해서 다른 요구사항이 무시되는 것은 올바른 선택이 아님을 명심해야 한다. “최적화된 코드는 구조화하기 어렵지만, 구조화된 코드는 최적화하기 쉽다”. 즉, 이미 구조적으로 의존성이나 역할 관계의 정의가 끝났기에, 어디가 정말 성능을 높여야 할 부분인지 쉽게 알 수 있다. 또한, 최적화를 위한 코드 수정에도 더 강건할 수밖에 없기 때문이다.

# [ 절차지향과 객체지향 ]

최근에 학교를 졸업한 신입 개발자를 제외하면, 아직 객체지향 언어에 대해서 익숙하지 못한 개발자가 많다. 특히, 임베디드 시스템 만드는 개발자는 C언어나 어셈블리(Assembly) 언어에 더 익숙하다. 하지만, 점차 객체지향 언어를 사용하는 개발이 늘어나고 있는 분위기고, 이에 발 맞춰서 경험 있는 임베디드 시스템 개발자들도 점차 C++언어의 사용을 늘려가고 있다. 하지만, 안타깝게도 실제로는 C++을 사용하는 C언어 프로그램을 개발하는 것이 현실이다.

절차지향적인 생각을 객체지향적인 언어로 표현한다고 객체지향 프로그램이 되는 것은 아니다. 당연히 객체지향 언어를 사용한다고 해서 자동으로 유지보수성이 좋아지고 재사용성이 높아지는 것도 아니다. 그렇다고, 이런 이야기를 한다고 해서 개발자의 자질이 부족하다거나 책임을 묻는 것은 아니다. 단지, 제대로 된 객체지향 프로그램을 만들고자 한다면, 무엇을 어떻게 해야하는지 같이 고민해 보고 싶을 뿐이다.

절차지향적인 언어로 코딩할 때는 주로 동사에 치중하는 경향이 있다. 즉, “누가 무엇을 어떻게 한다."에서 "어떻게 한다”에 중심을 둔다. 예를 들어, "네트워크를 이용해서 동영상을 전달"이라는 요구사항을 듣는다면, 해야 하는 일의 절차를 동사로 식별한다. "동영상을 캡처(capture) 한다. 인코딩(Encoding) 한다. 버퍼에 저장한다. 버퍼에서 데이터를 가져온다. 데이터를 네트워크로 전달한다." 같이 해야 할 일과 순서에 집중한다.

하지만, 객체지향적인 언어로 코딩 한다면 주로 일을 하는 주체(혹은 객체)에 치중해서 명사를 보게 된다. 네트워크를 담당하는 주체, 동영상을 담당하는 주체, 버퍼를 관리하는 주체, 인코딩을 담당하는 주체 등에 주요 관심을 둔다. 그리고, 식별된 주체에 역할을 정의한 후, 책임을 할당하게 된다. 주어진 요구사항을 구현하기 위해서는 이러한 객체들 간에 관계를 설정하고, 각각의 객체들은 다른 객체에 자신이 하고자 하는 일을 요청(Request)하는 방식으로 대화를 나눈다.

예를 들어, 동영상을 담당하는 객체는 영상 캡처를 하고, 인코딩을 담당하는 객체에 인코딩을 요청하게 된다. 인코딩 된 영상 데이터의 저장은 버퍼링을 담당하는 객체에 요청을 보내서 처리하고, 네트워크를 담당하는 객체는 버퍼링 된 데이터를 네트워크를 통해서 전달한다. 즉, 각자가 담당한 역할을 충실히 수행하는 것으로 요구사항을 구현하게 된다. 다시 말하면 객체들의 협업으로 구현된다.

객체지향적으로 프로그램을 개발하기 위해서 항상 객체지향 언어만 사용해야 하는 것은 아니다. C언어를 사용하더라도 객체지향이 가지는 장점을 활용할 수 있다. 문제는 무엇이 객체지향이고, 무엇이 그렇지 않은 것인지 개념적으로 확실히 이해해야 한다는 것이다. 개념은 모른체 도구만 열심히 사용한다면, “닭 잡는데 소 잡는 칼을 쓰는 것”처럼 보일 것이다. 더 큰 문제는 소를 잡아야 하는데, 닭을 잡을 칼을 들고 덤비는 것이다.

객체지향이 절차지향보다 좋다 나쁘다를 말하는 것은 의미없다. 무엇이 더 좋다고 판단하기 전에, 적어도 해당 언어를 사용하기 위해서 필요한 개념부터 이해해야 한다. 사고가 바뀌지 않으면 어떤 언어를 사용해도 절차지향 코드가 될 수밖에 없다. 문법 만 그럴듯하게 사용한다고 해서 객체지향 프로그래밍이 되는 것은 아니다. 내재된 개념을 받아들이지 않고 단지 코딩 팁(Tip) 만 찾게 된다면, “돼지 목에 진주 목걸이”라는 말이 낯설게만 들리지는 않을 것이다.

# [ 객체지향에 대한 단상 ]

객체지향 프로그래밍을 하기 위해서는 객체지향적인 분석이 선행되어야 한다. 분석 과정은 어떤 객체가 필요한지 확인하는 절차이며, 이를 이용해서 객체 간의 관계를 정의하는 활동을 진행하게 된다. 객체 간의 관계가 정해지면 일련의 시나리오가 만들어지게 되며, 시나리오를 구체화하는 것이 구현이다. 하지만, 실무 개발자들은 이런 과정을 일일이 거치면서 일하지 않는다. 경험이 있는 개발자는 자신이 이미 알고 있는 사실을 이용해서 대략적인 객체만 식별한 후 구현에 들어간다. 경험이 없는 개발자는 경험있는 개발자를 따라한다.

C언어를 사용하는 개발자들이 C++을 만났을 때 느끼는 가장 큰 어려움은, 개념에 대한 이해를 바탕으로 객체를 분석하지 못한다는 점이다. 즉, 언어가 가진 문법적인 특징은 알고 있지만, 언어를 필요로 하게 만든 이유를 분석에 활용하지 못한다. 기존에 구조체(Structure)로 정의된 자료구조와 그것을 이용하는 함수를 묶어서 클래스로 정의하는 것이 전부다. 이것은 그냥 C++로 만들어진 C코드일 뿐이지, C++언어가 주는 사상을 녹여서 나오는 결과물이 아니다. 따라서, 구현하려는 시스템의 설계에 객체지향 개념이 반영된 것이라고 볼 수 없다.

문법적인 팁(Tip)을 열심히 갈고 닦는다고 해서, 그 분야의 최고 전문가는 될 수 없다. 이것은 마치 강가에서 혼자 수영을 배운 사람이 장거리를 갈 수 없는 것과 같은 이치다. 수영장에서 제대로 된 코치의 가르침이 없다면, 물가에서 노는 정도 밖에 할 수 없다. 모든 배움에는 왕도가 없다. 기본 개념부터 차근차근 습득하지 않고 바로 코딩에 적용하는 것은, 나중에 발생할 유지 보수 비용 만 증가시킬 뿐이다. 제대로 배우지 않았기에 단기적인 목표만 만족시킬 뿐, 장기적인 이익에는 기여하지 못하게 되는 것이다.

객체지향을 적용하기 위해서 필요한 것은 “개념에 대한 이해와 좋은 가르침”이다. 모르면 배우는 것이라도 잘 해야 한다. 배움은 열린 마음(Open Mind)에서 나오며, 기존의 사고만 고집한다면 새로운 것이 들어갈 공간이 없다. 좋은 가르침은 이론에만 한정된 것이 아니라, 남들은 정말 객체지향으로 어떻게 코딩하고 있는지 직접 보는 것이다. 단지 “내가 했기 때문에 좋은 코드”가 아니라, 남들이 보기에도 좋은 코드를 만들기 위해서는, 다른 사람의 시각으로 검토된 피드백이 필요하다. 따라서, 자신의 코드를 여러 사람이 볼 수 있는 공간에 제공하는 것도 좋은 가르침을 얻는 길이다. 내가 하지 못하는 일은 남에게 부탁(요청)해서 해결하는 것이, 동등한 객체들이 조직을 구성하는 방식이기 때문이다.

# [ 객체(Object)란? ]

객체지향을 이야기한다면, 당연히 객체부터 먼저 정의를 할 수 있어야 한다. 하지만, 안타깝게도 대부분 이런 질문에 대해서, "객체는 클래스의 메모리 내 인스턴스(Instance)입니다”와 같은 다소 교과서적인 답을 한다. 당연히 다음 질문으로 "그럼 클래스는 뭔가요?"를 던지게 된다. 그리고, 마찬가지로 대답은 “객체를 찍어내는 템플릿입니다.”가 될 것이다. 객체와 클래스가 이렇게 순환 관계처럼 정의를 가지게 되면, 둘 다 제대로 정의된 것이 아니다. 객체에 대해서는 여러 가지 정의가 있지만, 나 만의 정의를 한다면 아래와 같다.

**"객체는 시스템에서 역할과 책임을 부여받은 최소 단위이며, 주어진 역할을 수행하기 위해 필요한 내부 상태를 가지는 속성(Attribute)과 상태의 변경을 위한 행위(Behavior)로 구성된다."**

뭔가 아직 조금 교과서에서 나오는 것과 크게 다르게 보이지는 않지만, 일단 "역할과 책임"이 있다는 것과, "내부적인 상태"를 가질 수 있다는 점, 그리고 이를 이용해서 “해야 할 일"이 있다는 점은 명확하다. 즉, 한 가지 역할 만을 수행하고, 책임 범위가 명확히 정해져 있으며, 더 이상 분해할 수 없는 최소 단위가 되는 것이 “객체”라는 것이다.

역할과 책임을 분해하는 이유는 복잡한 문제를 해결하는 일반적인 방식이기 때문이다. 한 번에 큰 문제를 해결하는 것은 어렵다. 문제 해결에 영향을 주는 여러가지 요소가 있을 수 있으며, 또한 요구사항 자체도 복잡하고 구체적이지 않기 때문이다. 따라서, 이러한 복잡하고 한 번에 해결하기 어려운 문제는 분리해서 작은 문제로 만들어야 한다. 분리된 작은 문제 각각에 대한 해답을 구하고, 분리된 문제들의 해답을 모으는 반복 과정을 통해서 전체 문제에 대한 해답을 구하게 된다. 이런 방법을 “분할 정복(Divide & Conquer)”이라고 부르며, 복잡한 문제를 다루는데 사용한다. 따라서, 객체도 따지고 보면 분할 정복된 하나의 해답을 전해주는 메모리 내 동작하는 코드의 일부일 뿐이다.

클래스는 객체를 추상화시키는 개념이다. 즉, 객체들이 가지는 공통된 특징을 끄집어내 이름을 붙여준 것이 클래스다. 이름을 준다는 것은 “추상화”시켜 표현할 수 있다는 뜻이다. 각각의 객체들이 가지는 구체적인 속성 값이 중요한 것이 아니라, 그것을 표현할 수 있는 추상화된 표현이 필요할 때 클래스를 사용하게 된다. 따라서, 객체가 없이는 클래스도 없는 것이며, 클래스가 없으면 객체의 공통된 속성과 행위를 알 수 없게 된다. 우리는 클래스를 통해서 객체를 사용하게 되지만, 구체적인 행위는 각각의 객체가 만족시켜야 할 “계약”으로 실현된다.

# [ 소프트웨어 설계에 대한 단상 ]

“소프트웨어 설계란 무엇일까?” 여러 번 같은 질문을 던지고 대답하려고 노력해 보지만, 대답은 항상 역할과 책임(Role & Responsibility)에 대한 생각에 머문다. 흔히, 제대로 설계해야 문제점을 줄 일 수 있다고들 이야기 하지만(물론, 실제로도 그렇게 생각하지만), 정작 개발 일을 하다보면 설계 보다는 구현에만 집중하는 것이 현실이다. 설계는 될 수 있으면 간단히 하고 넘어가고, 대부분의 시간을 구현에 투자하는 것이 현명하다고 믿는다.

물론, 설계가 부족할 수밖에 없는 이유도 있다. 이미 작성된 재활용할 코드가 있거나, 요구사항을 제대로 정의하지 않은 상태에서 과제를 시작하기 경우도 있기 때문이다. 코드를 재활용하게 되면, 기존에 작성된 설계서를 찾게 된다. 만약, 작성되지 않았다면 원래 없었다는 말로 넘어가고, 요구사항이 없으면 무엇을 개발해야 하는지 알 수 없다. 따라서, 설계 문서를 작성할 근거도 없다. 하지만, 항상 이런 것은 아니다. 프로세스가 요구하는 최소한의 산출물은 등록해야 다음 단계로 넘어갈 수 있기 때문이다. 그리고, 프로세스는 내용보다 있는지 없는지만 확인하는 경우가 많다. 아무도 내용까지 차분하게 읽어보지는 않는다.

지속적으로 변경되는 구현을 설계와 일치시키는 일은 쉽지 않다. 대부분 개발자는 설계를 하고 나면, 설계된 내용을 잊어버린다. 구현하는 동안 문서를 새로 업데이트 하는 일도 거의 없다. 간혹 설계한 문서를 구현과 일치시키고자 지속해서 변경하는 개발자도 있지만, 대부분의 설계 문서는 구현과 일치하지 않는 것이 일반적이다. 일정에 쫓기고, 버그가 늘어나는 상황에서, 문서까지 챙길 수 있는 사람은 거의 없기 때문이다. 과제가 완료된 후, 혹은 새로운 과제를 시작하기 전에 설계 문서를 고치면 된다고 생각할지 모르겠지만, 그 때가 되면 설계에 관련된 결정이나 근거는 이미 기억에서 사라지고 없다.

따라서, 업데이트 된 문서는 결과만 위주로 정리되며, 그나마도 시간이 있어야 가능하다. 과제 완료가 새로운 과제의 시작을 알려줄 뿐, 팀으로 옮겨서 일할 수도 있다. 끝나버린 과제에 대해서 미련을 가지고 정리하는 일은 남아 있는 사람들의 몫이며, 다른 팀으로 가버린 사람들의 생각까지는 문서에 담지 못한다. 당연히 불완전한 문서가 될 수밖에 없다.

설계 문서의 수준이 낮다고 해서 설계를 등한시하는 것은 옳지 않다. 밑그림이 없는 상황에서 구현에 들어가면, 재작업으로 인해 더 큰 비용이 들어가게 되기 때문이다. 잘못된 구조의 코드를 고치는 것은, 마치 건물을 지을 때 저층을 뜯고내고 새로 만드는 것과 같다. 사실 소프트웨어 구현에서는 이런 일이 자주 발생할 수 있으며, 일정 준수 측면에서도 재작업이 가장 큰 악영향을 준다. 따라서, 재작업을 가능한 줄이는 것이 비용을 줄이고 일정을 맞추는 핵심 활동이 되는 것이다.

건축이라면 위 층을 남겨두고 아래 층을 완전히 뜯어내는 일은 없을 것이다. 재건축이라면 몰라도 대부분은 항상 아래를 유지하면서 조금씩 고치는 것이 일반적이다. 소프트웨어 아키텍처를 건축에 비유하기도 하지만, 실제로는 이렇게 큰 차이를 있다. 소프트웨어가 “소프트”하지 않게되면 비용은 큰 폭으로 증가한다. 즉, 기존의 토대를 남겨두고 기능 추가나 변경을 하지 못하는 상황이 되고마는 것이다. 설계는 소프트웨어가 “하드”하지 않도록 만든데도 중요한 역할을 한다. 잘 정의된 변경점을 정의하는 것 만으로도 왠만한 변경은 충분히 감내할 수 있기 때문이다.

설계는 “구현할 수 있는 충분한 뼈대”를 제공하는 수준에서 만족하여야 한다. 완벽한 설계를 한 후에 구현에 들어가는 것은 모든 요구사항이 과제의 시작과 동시에 완벽히 분석되고, 더 이상 변경이 없어야 만 가능한 일이다. 또한, 소프트웨어를 받쳐주는 하드웨어의 변경 및 새로운 추가 기능도 없어야 한다. 설계 만으로는 구현 후 어떤 성능을 보여줄지 정확히 알 수 없다. 따라서, 구현 성능이 만족되지 않는다면, 설계 자체도 변경할 수밖에 없는 상황이 온다. 물론 꼭 성능만 그렇다는 것은 아니다. 다른 비기능적인 요구사항들도 기능적인 요구사항과 마찬가지로 언제든 변경될 가능성이 있다.

설계는 움직이는 표적을 맞추기 위해서 총을 겨누는 일이지, 표적 자체를 맞추기 위해서 날아가는 총알은 아니다(변경이 불가능한 것이 아니기에). 뼈대를 제공하는 수준에서 만들어진 설계는, 구현을 진행하면서 이런저런 것들로 변경이 발생할 수 있다. 따라서, 적절한 설계 수준은 구현해야 할 요소들을 식별하고, 이들 간의 관계를 정의할 수 있어야 하며, 어떤 식으로 중요 기능이 수행될 것인가를 나타내는 정도일 것이다. 이렇게 만들어진 설계는 각각의 구현해야 할 요소들에 일정과 개발자를 할당하기 위한 근거가 될 수 있다. 소프트웨어 개발은 시간과의 싸움이며 변화에 대한 응대(Response)다. 설계는 변화의 방향성을 제시할 수 있으며, 움직이는 목표을 향해서 최대한 가깝게 다가 가도록 도와줄 것이다.

# [ 계층 구조에 대해서 ]

모든 소프트웨어 개발자들은 한 번쯤 계층화(Layering)라는 이야기를 들어봤을 것이다. 그리고, 컴퓨터 관련 전공을 했다면 OSI 7 계층(Layer)이라는 것도 대부분 알고 있을 것이다. 이처럼 계층화된 구조는 흔히 복잡한 문제를 해결하기 위해서 사용하는 방법일 뿐이다. 소프트웨어가 복잡하다고 믿는다면 계층구조를 적용해서 구현하는 것은 지극히 자연스러운 일이다.

하지만, 실제 구현에서는 개발자들이 자신이 구현한 코드를 계층구조라고 이야기 하지만, 그렇지 못한 경우를 흔히 발견할 수 있다. 즉, 계층구조라고 이야기할 때는 구조 자체가 만족시켜야 할 조건이 있으며, 그것을 만족시키지 못하면 계층구조라고 부를 수 없다. 머리 속으로는 계층구조를 정의하고 있을지 몰라도, 그것이 실제 코드에 반영되지 못한다면 계층구조로 구현했다는 말을 해서는 안된다. 그렇다면, 도데체 왜 개발자들은 계층구조와 동떨어진 구현을 하고 있는 것일까?

계층화는 상위 계층에서 하위 계층으로 추상적인 것이 구체적인 것으로 변화된다는 의미다. 또한, 인접한 상위 계층과 하위 계층 사이에서만 호출과 복귀가 일어나야 한다. 즉, 인접한 하위 계층을 건너뛰고 그 아래의 계층에 대한 직접적인 접근을 해선 안된다. 예외적으로, 상하로 길게 위치한 모듈이 있을 수 있으며, 그럴 경우에도 화살표를 통해서 한 쪽 방향으로만 호출이 가능하다. 따라서, 계층구조를 만족시키기 위해서는 하위 계층의 상위 계층에서 대한 호출이나 참조가 없어야 하며, 계층을 건너뛴 호출이나 참조도 발생하지 않아야 한다.

계층화(Layering)는 생각의 폭에 대한 것이다. 즉, 사람의 생각을 계층화시켜서 구체적인 것 보다 전체를 쉽게 이해하기 위한 것이다. 즉, 하나의 계층은 그 계층 수준에서의 추상화를 표현하고 있어야 한다는 뜻이다. 예를 들어, 파일에 데이터를 쓰기 하는 과정을 보도록 하자. 최상위 수준에서는 “데이터 쓰기”라는 함수가 주어진다면, 그 다음 수준에서는 데이터를 쓰기 위한 “버퍼를 할당”한 후, 쓰려고 하는 “데이터를 복제”할 것이다. 버퍼가 준비되지 않았다면, 버퍼 할당을 위해서 대기할 수도 있다. 버퍼 복제가 완료되면 출력을 위한 디바이스 드라이버에 버퍼의 주소를 넘겨주고, 출력을 요청하게 될 것이다. 즉, 각각의 계층을 내려가면서 요청은 점점 더 구체적인 형태로 변화된다.

소프트웨어 개발에서는 이와 비슷한 방법으로 해결해야 할 문제의 추상적인 해결책을 최상위에 두고, 이하의 계층들에서 자신의 수준에 맞는 추상화를 통해 문제를 풀어나간다. 이런 식으로 최하위까지 내려가게 되면, 실질적인 하나 하나의 “구체적인” 해결책이 나오게 되는 것이다. 물론, 이것은 구조적인 프로그램이나 객체 지향적인 프로그램도 크게 다르지 않다. 즉, 문제는 잘게 부수는 과정(Decompose, Divide)을 통해서 조각을 만들고, 각 조각의 부분 해를 구한 다음, 다시 부분 해를 합쳐서 전체적인 해결방안(Solution, Conquer)을 만들어 나가기 때문이다. 이때 중요한 것은 자신이 속한 계층에서 필요한 추상화 수준이 무엇인지를 항상 기억해야 한다. 즉, 같은 계층에서는 같은 수준의 추상화가 이루어져야 한다는 점이다.

계층화를 방해하는 요소로는 계층을 건너뛰는, 혹은 하위계층에서 상위계층으로의 호출 및 참조 가 있다. 이런 것이 발생한다면, 순환적인 의존 관계를 상위계층과 하위계층이 만들고 있다는 의미가 된다. 따라서, 각각의 계층에서 발생하는 변경이 다른 계층까지 영향을 주는 일이 발생한다. 상위계층과 하위계층을 구분하는 간단한 방법은 호출이나 참조의 수를 계산해 보는 것이다. 즉, 참조나 호출이 더 많은 쪽이 일반적으로 상위계층일 가능성이 높다. 따라서, 하위계층에서 상위계층으로 향하는 호출이나 참조는 잘못된 오류로 볼 수 있다. 이와 같은 경우를 역참조(Reverse Reference)라고 부른다.

역참조와 같은 문제를 해결하는 방법은 의외로 쉽다. 참조가 발생한다는 말은 곧 해당하는 부분이 자신에게 필요하다는 말이 된다. 따라서, 참조되고 있는 부분을 끄집어내서 원래 필요로 하는 곳으로 옮겨주면 된다. 혹은, 반대로 참조를 요청하는 코드를 참조하고 있는 코드 속으로 옮겨주는 것이다. 옮겨주는 것이 어렵다고 생각한다면, 공통으로 참조하는 부분을 분리된 새로운 모듈로 만들어주면 된다. 문제는 이때 옮겨야 하는 자료구조나 변수들이 있는 경우인데, 이 때문에 함수의 인터페이스에 파라미터의 추가가 발생할 가능성은 있다.

근본적으로 계층화 위반을 발생시키지 않기 위해서는 인터페이스(Interface)를 위주로 하는 코딩 전략이 필요하다. 될 수 있으면 모듈의 내부 정보에 의존하지 않아야 하며, 제공된 인터페이스만 사용해서 코딩해야 한다. 필요한 인터페이스가 있다면, 그것을 제공해야 하는 모듈이 누구인지 를 확인해야 한다. 직접 그 일을 처리하기 보다는 필요한 자료구조를 정의한 모듈에 “요청”하는 방식으로 코딩해야 한다. 즉, 다른 곳에서 제공하는 인터페이스를 이용해서 코딩해야지, 그곳에 있는 자료구조나 객체의 내부 상태에 의존해서 코딩하면 안 된다. 또한, 자신이 만든 모듈이 한 가지 일(역할)에 만 집중하고 있는지도 항상 주의깊게 관찰해야 한다.

# [ 전략적 코딩 vs. 전투적 코딩 ]

소프트웨어 개발자는 꾸준히 코딩해야 먹고 사는 직업이다. 비록 관리자 업무가 많은 사람 일지라도 자신의 코딩 역량은 꾸준히 유지하는 것이 좋다. 새로운 언어가 나오거나 새로운 방법론이 나오면, 그것을 통해서 얻을 수 있는 개념은 반드시 확인하는 습관을 지녀야 한다. 낡은 사고나 방법 혹은 툴을 가지고 10년이고, 20년이고 지속해서 일할 수 있는 직장은 흔하지 않다(물론, 그렇게 먹고 사는 사람들도 있겠지만). 낡은 것들을 자신의 핵심 역량이라고 믿는 사람이라면, 혹시 밑에 있는 부하직원들의 발전을 가로막고 있지나 않은지 스스로 확인해 보기 바란다.

소프트웨어의 문제는 항상 코딩에 있다. 구조가 나쁜 이유를 설계에서 찾지만, 구현 자체에 설계를 제대로 반영하지 못해서 발생하는 문제가 대부분이다. 특히, 형식적인 설계를 하는 회사라면 대부분의 문제는 코딩으로 연결되는 고리가 없기 때문이다. 전략적인 코딩은 어떻게 코딩할 것인가를 미리 정하고 구현을 시작하는 것이다. 즉, 코딩이 지향해야 하는 명확한 목표(가치)를 가지고 있다는 것이다. 전투적인 코딩은 지향할 목표 없이 개발자의 역량에 모든 결정을 무조건 맡기는 것이다. 어떤 선택이 더 나은 결과를 가져오는지는 시간이 흘러가면 차츰 알게 된다.

우리는 흔히 전투적인 코딩에 익숙하다. 이유를 대부분 “시간과 관행”이라는 두 개의 축으로 나누어 볼 수 있다. 시간이 없어서 기존 코드를 그냥 사용했거나, 혹은 너무 많은 문제를 처리하다 보니 결과적으로 그렇게 되는 경우다. “윗사람이 그렇게 해왔으니 나도 그렇게 한다”는 아무 이유 없는 관습화된 것들이 전투적인 코딩으로 개발자를 내몬다. 코딩을 많이 하면 잘하는 것처럼 보이는 이유도 마찬가지다. 계획되지 않은 코드를 아무리 많이 만든다고 해서 좋은 코드가 나오는 것은 아니지만, 일단은 그렇게라도 만들지 않으면 불편한 마음을 다스릴 수 없기 때문이다.

설계서를 만드는 이유는 대화(communication)를 목적으로 한다. 하지만, 만들어진 설계서를 보는 경우는 신입사원 때 잠시뿐일 것이다. 그렇다고, 그렇게 만들어진 설계서가 실제 코드를 이해하는데 도움이 되지도 않는 것도 사실이다. 설계와 코드의 불일치 문제를 해결하기 위해서, 항상 개발이 완료된 시점에 문서를 새로 고쳐야지 하는 생각만 하게된다. 하지만, 회사는 사람을 절대 놀리는 법이 없다. 이미 작성된 설계서가 있다면, 수정하는 것은 일이 없는 사람들이 한가할 때 하는 여가 정도로 밖에 생각하지 않는다. 또한, 개념없는 관리자는 어떤 일이라도 시켜서 개발자들을 바쁘게 만들어야지 인정받는다고 믿는다. 따라서, 하나의 과제가 마무리가 되면 다른 과제를 시작하거나, 시장에서 나오는 문제를 해결한다고 정신없는 시간을 보낼 것이 분명하다. 나중이라는 시간은 절대 오지 않을 것이다.

그렇다면, 이 글에서 말하는 전략적인 코딩은 무엇일까? 사실 이 질문에 대한 대답은 아주 단순한 곳에서 시작한다. 비록 잘 쓴 설계서는 아니더라도 최소한 만들어 놓은 그림(Diagram) 정도는 있을 것이다. 이것을 기반으로 디렉토리(Directory) 구조를 만드는 것이 가장 먼저 해야 할 일이다. 그림에 등장하는 사각형 박스를 하나의 디렉토리로 만들고, 내부에 정의된 작은 블록(Block)들을 하위 디렉토리로 만들어 나가는 것이다. 이때, 디렉토리의 이름은 반드시 설계서에 나오는 이름을 붙이는 것이 중요하다.

좋은 이름을 붙이려면 어떤 디렉토리에 무슨 파일이 있는지 예측가능 해야 한다. 디렉토리에 포함된 파일들이 무슨 역할을 하는지 내부를 뜯어보지 않아도 파악할 수 있어야 한다. 파일의 이름도 마찬가지다. 파일을 열어보지 않더라도 무슨 일을 하는지 이름으로 알 수 있어야 한다. 같은 방법으로 함수, 클래스, 변수 등등 모든 소프트웨어의 작은 구성 요소들은 “좋은 이름”을 가져야 한다. 설계에 등장하는 이름과 코드를 구성하는 요소들의 이름이 같아야 설계서가 의미를 가진다. 설계한 이름과 실제 사용되는 이름이 달라지면 상상력만 키울 뿐 정말 필요한 정보는 전달하지 못하게 된다.

다음 단계는 모든 디렉토리를 각각 따로 분리해서 빌드(Build)를 할 수 있도록 만들어야 한다. 마치, 각각의 디렉토리별로 하나의 라이브러리(Library)를 만든다고 보면 될 것이다. 이렇게 만들어진 라이브러리들은 최종 바이너리 이미지(Binary Image)를 만들 때 사용될 것이다. 전체 빌드는 최상위 디렉토리에서 커맨드 라인(Command Line) 명령어로 가능하도록 만들어 두어야 한다. 나중에 빌드 서버(Build Server)나 테스트 서버(Test Server)를 이용해서 지속적인 통합 (CI: Continuous Integration)을 위해서 사용될 것이다.

이제는 사용할 소스 코드와 헤더 파일 등의 표준 파일 포맷(Format)을 만들어야 한다. 회사에서 사용하는 “Copyright” 문구와 각종 코딩 규칙(Coding Rule)들이 간략하게 들어간 정도면 충분하다. 코딩은 여러 사람이 하더라도 마치 한 사람이 작성한 것처럼 만들어야 한다. 코딩 룰을 정의하되 너무 많은 규칙은 만들지 말아야 한다. 한 장 정도에 요약해서 정리하는 것이 좋다. 누가 보더라도 15분 이상 읽는 데 시간이 걸린다면, 지키지 않을 가능성이 크기 때문이다. 예를 들어, 아래와 같은 몇 가지 정도면 충분할 것이다.

**01. 모든 함수는 Return값을 가지도록 만들자.**

**02. 모든 함수의 Return값은 반드시 확인하도록 한다.**

**03. 모든 함수는 Parameter값의 유효함을 확인하도록 한다.**

**04. 상수로 정의된 모든 값은 이름을 붙이는 방식으로 한다.**

**05. 같은 자료구조는 할당과 해제가 같은 수준에서 되도록 만든다.**

**06. 메모리의 할당은 초기에만 하고, 해제는 마지막에만 하도록 한다.**

**07. 모든 함수의 길이는 짧게 만들고, 길어지면 다른 함수로 분리한다.**

**08. 함수 내의 변수는 지나치게 큰 자료구조를 사용하지 않는다.**

**09. 전역 함수와 자료구조는 만들지 않는다.**

**10. 보여줄 정보만 보여준다.**

이상과 같은 몇 가지를 코딩 룰에 덧붙여서 적어놓던가, 혹은 프린트를 해서라도 책상에 붙여놓고 일하면 최소한 코딩이 난잡해지는 것은 어느 정도 막을 수 있을 것이다. 이런 것들이 사소하다고 생각할지도 모르지만, 사소한 것을 지키는 것이 코딩의 기본이다. 과제 후반에 몰리게 되면 기초는 무시되고, 급한 수정을 남발하는 경우가 많기 때문이다. 무한 반복되는 디버깅과 테스트를 줄이기 위해서는 작지만 중요한 것들을 습관처럼 지켜야 한다.

코딩도 전략을 가지고 해야지 깔끔하게 된다. 설계 따로, 코딩 따로 로 간다면, 문서는 낭비를 넘어 치워야 할 쓰레기가 되고, 코드는 수정해야 할 "기술적인 빚(Technical Debt)"만 쌓이게 된다. 빚은 쌓아두기 보다는 보이는데로 해결해야 해야 한다. “빚도 자산이다”라는 말도 있지만, 적어도 코딩에서 발생하는 빚은 낭비일 뿐이다. 빚은 시간이 흐르면 이자가 불어나게 되며, 원금을 갚기는커녕 개인의 정신적인 파산과 더불어 회사의 경제적인 파산을 몰고 올 수도 있다. 따라서, 쌓여가는 빚을 어떻게 갚아야 하는지 고민만 하기 보다, 차근차근 일수 통장에 도장 찍듯이 줄여 나가도록 해야 한다.

# [ 버그(Bug)를 바라보는 자세 ]

버그가 늘어가는 것을 기분 좋게 바라보고 있을 개발자는 아무도 없을 것이다. 마음은 다급해지고, 여러 곳에서 버그에 대한 대처방안을 세우라는 독촉을 받게 되며, 다가오는 배포 일정에 대해서 심리적인 압박감을 심하게 느낄 것이다. 스트레스성 위염과 불면, 그리고, 그 불면으로 인한 피로감에 다크써클을 눈 주변에 끼고 살게 된다. 우리에게는 지금 당장 버그를 해결할 방안이 없는데도 여기 저기서 아우성 들이다. 그저 버그가 조금이라도 덜 생기길 빌 수밖에 없다. "오늘도 무사히 아무런 버그가 없기를"이라는 표어 밑에서 기도를 드리고 있는 자신의 모습을 애처롭게 상상하게 된다.

그럼, 늘어가는 버그를 대처하고, 다가오는 일정을 어떻게 맞출 것인가? 사실 이미 버그가 많이 있다는 말은 개발이 상당히 진행된 상태이며, 그 동안 제대로 된 테스트를 실행해 보지 않았다는 뜻이다. 통합이 늦어졌거나, 단위 테스트가 부족했고, 추가적인 요구사항에 대한 구현이 미비 하다는 말이다. 구현이 다 되었다고 해더라도 개발자가 자신의 코드를 제대로 돌보지 못했다는 신호다. 미리 방지했어야 만 하는 오류지만, 지금의 상황에서 어쨌든 최선을 다해서 대처해야 만한다. 이런 상황을 해결하기 위한 방법으로는 여러가지가 있을 수 있겠지만, 여기서는 아래의 몇 가지만 보도록 하겠다.

첫 번째 방법은 미구현된 요구사항을 다음번 배포로 미루는 것이다. 지금까지 완성한 기능을 가지고 테스트와 협의를 하는 것이다. 완성되지 않았고 성능 및 안정성이 불안한 기능을 넣어서 테스트를 정해진 기간 내에 통과하기를 기대하는 것은, 누워서 감 떨어지기를 기다리는 것과 같을 수 있다. 즉, 운에 기대는 것일 뿐이다. 언제 떨어질지 모르는 감을 먹으려고 입 만 벌릴 수밖에 없다. 오히려 기능들을 과감히 삭제하는 것이 제품을 훨씬 빨리 안정화시키는 지름길이다. 물론, 이것도 불가능한 경우가 있다. 예를 들어, 구조화되지 않은 상태의 코드는 완료되지 않은 기능을 제거하는 것도 어렵다. 차라리 버전 관리 시스템에서 추가 기능구현 이전으로 돌아가는 것이 더 빠를 수도 있다. (개인적으로 이것이 최선이라고 생각하지만, 언제나 선택은 독자의 몫이다.)

두 번째 방법은 일정을 지연시키는 것이다. 하지만, 일정 지연은 그동안 들였던 모든 노력에 대한 반응으로 “욕” 밖에는 못들을 것이다. 이로 인해 발생하는 금전적인 손실은 그대로 소프트웨어 개발팀의 책임으로 돌아올 것이다. 하지만, 사실 이게 제품에 대한 시장의 실망을 사전에 방지할 수 있는 유일한 방법이기도 하다. 완성되지 않은 기능을 내놓는다거나, 혹은 그럴듯하게 포장해서 판매하는 것은 결국 제품의 신뢰도만 손상시키기 때문이다. 일정 지연이 필요하다는 것을 받아들이도록 만들기 위해서는 상당한 시간의 손실이 필요할지도 모른다. 하지만, 중요한 것은 “그래서 언제냐?”에 대한 답이다. 높으신 분들은 “그 날”을 원한다.

세 번째 방법은 극단적인 경우로, 해당 제품에 대한 개발을 멈추고 더 안정적인 방법을 찾도록 다른 제품을 만드는 것이다. 예를 들어, 하드웨어 사양을 극단으로 끌어내야지만 겨우 원하는 기능이 나올까 말까 하는 상황에서는, 노력을 아무리 해도 성공할 가능성이 극히 낮다. 차라리 하드웨어 사양을 높여서, 원활하게 동작할 수 있을 정도의 성능을 추가로 확보하게 만드는 것이 더 빠를 수 있다. 하지만, 이것은 어찌 보면 변명이라고 생각될 수도 있다. 면밀한 검토 없이 일을 진행하다가, 잘 안되니 바꾼다는 인상을 심어주기 때문이다. 그리고, 그렇게 바꾼다고 해서 일이 줄어들지 않을 수도 있다. 새로운 하드웨어를 찾지 못할수도 있고, 새로 개발해야 할 경우에는 안정성도 확보하지 못하기 때문이다. 당연히 시간과 비용도 추가적으로 대폭 들어가야 할 것이다.

어떤 식으로든 버그를 바라보는 관점은 긍정적이어야 한다. 될 수 있으면 해당 버그에 대해서 분석적으로 접근해서, 무엇이 문제인지 근본적인 원인을 찾으려고 노력해야 한다. 심지어, 큰 구조 변경을 해야 할 경우가 생길지도 모른다. (그런 경우가 온다면, 오랫 동안 각종 보고로 시달릴 가능성이 높다. 왜냐하면, 버그에 대한 대처 없이 긴급하게 코드의 구조를 바꾸어야 하므로, 그 기간 동 버그는 쌓여만 갈 것이기 때문이다.) 버그는 발생한 이후가 문제가 아니라, 발생하기 이전이 더 큰 문제를 내보하고 있다. 발생하면 어쩔 수 없이 해결해야 하지만, 발생하기 이전에 이미 버그를 예상할 수밖에 없는 개발을 하고있기 때문이다. 따라서, 문제가 조금 줄어들었다고 생각한다면, 사전에 제거할 수 있는 방안을 마련하는 것이 반복적인 실패를 경험하지 않도록 해 줄 것이다.

# [ Good Code vs. Bad Code ]

코딩을 배우기 시작할 때는 제대로 배우는 것이 중요하다. 사실 우리나라의 현실에서 코딩을 제대로 배우기는 쉽지 않은 일이다. 대기업에 들어가서 그때까지 학교에서 배운 것들이 실무에는 별로 크게 상관이 없다는 것을 알게 되고, 새로운 습관을 만들어 가는데는 많은 시간이 걸린다. 이런 시간을 기다려줄 수 있는 회사는 그렇게 많지 않다. 시중에서 찾을 수 있는 책들은 특정 언어에 대한 사용 방법이나 코딩 기법은 가르쳐 주지만, “좋은 코드”에 대해서는 별로 이야기 해 주지 않는다. 물론, 외국 책으로 공부하면 된다고 이야기할 지 모르지만, 그나마 영어를 잘하지 못하는 공대생인 경우에는 진도는 안나가고 잠 만 오는 일일뿐이다.

“좋은 코드”와 “그렇지 못한 코드”(“Bad Code”라고는 하지 않겠다. 어쨌든, 우리가 만들어가는 코드이기에)의 가장 단순한 구분 기준은, 다른 개발자가 읽어보고 어떤 이야기를 하는지 들어보는 것이다. 좋은 이야기가 나온다면 좋은 코드일 것이고, 그렇지 않다면 당연히 개선해야 할 부분이 있다는 것이다. 좀 더 객관적인 기준이 필요하다면, 여러 가지 툴을 사용해서 분석해 볼 수도 있을 것이다. 여기서 사용될 수 있는 방법은 코드의 중복이 얼마나 있는지, 코드에서 분기가 얼마나 많은지를 확인하는 복잡도, 파일이나 디렉터리 간의 참조의 정도와 구조상의 역참조를 보는 방법 등 다양한 것이 있다. 이와 같은 방법을 사용하는 이유는 코드에 대한 객관적인 잣대를 찾기 어렵기 때문에, 코드가 보이는 다양한 특성을 종합해서 코딩 품질을 유추해야 하기 때문이다.

하지만, 이렇게 분석된 결과를 받아들이지 못한다면 아무런 의미가 없다. 정작 당사자는 아무런 문제를 못 느끼고, 기존의 방법데로 불편 없이 개발하고 있을 것이다. 문제는 이러한 코드를 다른 사람이 유지 보수를 하거나 기능을 확장해야 할 경우에 발생한다. 중복이 많은 코드는 같은 변경에 대해서 여러 곳을 수정해야 할 것이고, 분기가 많은 코드는 분기를 전부다 이해한 후에 수정해야 할 것이다. 역참조가 있으면 변경의 영향이 다른 파일이나 디렉토리에 들어있는 코드에도 영향을 주기 때문이다. 따라서, 당사자는 느끼지 않을지도 모르지만, 유지보수의 불편함은 이미 코드 내부에 깊숙히 숨겨지 있을 가능성이 높다.

유지 보수는 다른 사람을 위한 활동이다. 나 자신을 위해서만 일하고, 거기서 멈춘다면 그냥 그저 그런 개발자로 남는다. 개발의 전문가가 되기 위해서는 유지보수 등 기능적 요구 사항 이외의 부분에도 집중해야 하며, 소프트퉤어의 구조(Architecture)는 이러한 비기능적인 부분에서 나온다는 것도 이해해야 할 것이다. 코딩 활동은 기능적인 것을 구현하는 것에서 끝나지 않고, 비기능적인 것들을 효과적으로 다른 사람들에게 전달하는 것에 중요한 가치로 둔다. 따라서, 다른 사람이 바로 자신이 될 수 있다는 것을 잘 이해하고, 언제든 누가 그 일을 맡더라도 주어진 시간 내에 개발을 완료할 방법을 만드는데 게으르지 않을 것이다.

어떤 것이든 개선을 하기 위해선 시간과 노력이 필요하다. 하지만, 대부분 관리자는 개선과 관련된 활동을 나누어진 “일”으로 보지 않으며, 개발하면서 함께 진행하기를 원한다. 하지만, 현실적으로 이렇게 일을 하는 것은 어렵다. 왜냐하면, 급한 일을 먼저 하게 되지 중요한 일을 먼저 하지는 않는다는 사실이다. 눈 앞에 쌓인 버그를 해결하기를 원하지 코드가 아름답기를 바라는 관리자는 어디에서도 찾아보기 힘들다. 특히, 소프트웨어 개발보다는 하드웨어 개발에 더 중요성을 두는 조직에서는 소프트웨어 개선은 언제나 나중에 시간 있을 때나 해야 할 일이다. 그리고, “결코 나중이란 시간은 절대 오지 않는다”는 것도 잘 알고 있을 것이다.

소프트웨어 플랫폼을 적용하는데 가장 강력한 반대자는 같은 개발 일을 하는 소프트웨어 개발자라는 사실은 잘 알려지지 않았다. 이유는 기존의 틀을 허물어야만 새로운 것이 들어갈 틈이 생기기 때문이다. 기존 개발자들은 새롭게 개발된 플랫폼의 다양한 문제점을 이야기하면서 적용을 가로막을 것이다. 하지만, 좋은 코드의 가치를 객관적으로 설명하고, 그것에 대한 충분한 공통된 이해를 가져가려는 노력은 항상 해야 한다. 문제의 근본 이유를 보여주는 다양한 지표를 조사하고, 이런 것들이 왜 문제가 될 수 있는지를 보여주어야 한다. 소프트웨어 개발은 복잡함을 다루는 분야이며, 어떻게 다루는가에 따라 좋은 코드와 그렇지 못한 코드가 나온다. 마찬가지로, “복잡함을 다루는 태도”에서 “제대로 된 개발자”와 “그렇지 못한 개발자”로 커나가게 되는 것이다.

중요한 것은 지금 이 순간에도 그런 차이를 보일 수 있는 코드를 짜는 사람들이 있다는 것이고, 그 격차는 시간이 갈수록 더욱 벌어지게 될 것이다. 마침내는 도저히 건널 수 없는 강으로 변해, 모든 것을 처음부터 새로 시작하는 편이 더 빠르다고 생각될 날이 올 것이다. 개선에 필요한 비용을 지금부터라도 고려해서 남을 위한 코딩에 조금씩 신경 쓰는 것이, 미래를 보장하는 오늘의 숙제일 것이다.

# [ 창조 vs. 모방 ]

우리는 항상 창조적인 생각(Creativity)을 하라고 요구받는다. 하지만, 창조적인 생각은 쉽지 않은 일이고, 모방으로 가는 것이 더 빨리 남을 따라잡는 길 처럼 보인다. 창조가 무조건 좋고 반드시 해야 할 것이라면, 모방이라는 말은 왜 생겼을까? 학교를 가면 가장 먼저 어떤 것을 하는지는 잘 알 것이다. 우리는 조상들이 이미 이루어 놓은 것들을 가능한 빨리 배우도록 강요받는다. 따라서, 창의적인 생각은 일단 기존의 것을 모방하는 것으로 부터 시작한다고 볼 수 있다.

모방은 우리가 익숙하게 즐겨하는 행동이며, 동일하다는 것은 “안정감”이 있다는 것으로 받아들여지는 것이다. 창조는 그런 모방의 바탕에서 새로운 것을 추가하는 것이지, 아무것도 없는 것에서 새롭게 만들어내는 것이 아니다. 모방을 통해서 우리는 학습이라는 것을 하게 되고, 점차 학습의 수준이 높아져 감에 따라 자신만의 고유함을 기존의 것에 더하게 되는 것이다. 따라서, 누군가를 통해서 배운 것을 그대로 한번 해 보는 것은 학습의 수준을 높이는 과정에서 자연스러운 일이라고 할 수 있다.

하지만, 여기서 주의할 것은 학습은 학습에서 그쳐야 한다는 것이다. 누군가의 것을 자신의 독창성을 이해시키는데 사용하는 어리석은 짓은 하지 않아야 한다. 빨리 배우기 위한 목적은 자신의 생각을 표현하기 위한 효과적인 도구를 만드는 것이지, 그 도구 자체가 “창조”라는 목적을 대체할 수 있는 것은 아니기 때문이다. 따라서, 지금 하고 있는 일에 창의적인 부분은 없고 모방뿐이라면, 그것에 만족하고 현재에 안주하지 않기를 바란다. 모방은 시작일뿐이지 우리가 가야할 목표는 언제나 창조라는 점을 기억해야 할 것이다. 도구룰 익혔으면 올바른 일에 써야하는 것이다.

# [ 모델링(Modeling)이란? ]

모델링(Modeling)이란 해결하고자 하는 문제에 대한 특징을 모아서 추상화(Abstraction)하는 것이다. 모델링이 중요한 이유는 문제의 세부 사항을 생략하고 핵심에 더욱 집중할 수 있기 때문이다. 이미 이러한 노력은 다양한 분야에서 진행되어 왔으며, 소프트웨어에서는 문제를 해석하고 논의하는 과정에 필요한 도구로 사용되고 있다. 따라서, 모델링은 문제 해결의 모델을 정의하고 그것을 가지고 정말 문제가 해결될 수 있는지를 검토하는 활동이다.

모델링은 소통하는 방법이다. 같은 뷰(View: 시점, 시각)를 가지고 문제를 바라 봐야지 해결 방법을 더 효율적으로 찾을 수 있다. 문제를 서로 다른 견해로 해석한다면, 문제를 정의하는 데만도 오랜 시간이 걸릴 수 있다. 따라서, 모델링은 문제를 정의하고 더 나은 해결책을 찾는 시발점이 될 수 있다. 흔히 소프트웨어 개발에 대해서 잘 모르는 사람들은 개발자들이 어떻게 일하는지 눈에 보이지 않아서 상태를 파악하기 어렵다고들 이야기한다. 하지만, 이미 다양한 시각화 도구가 있는 소프트웨어 개발이야 말로 비용을 적게 들이고도 문제의 해결할 방법을 찾아내는데 최적화되어 있는지도 모른다.

모든 것을 한 장의 그림에 세세하게 표현하는 것은 어렵다. 복잡한 설계도를 보는 것 자체가 사실 처음 보는 사람에게는 짜증을 유발할 뿐이다. 가끔 설계 문서를 보다 보면 이런 지나치게 상세한 문서를 보게 되는 경우가 있는데, 이것은 문서를 읽는 사람의 이해보다 개발자 편의로 작성한 것일 뿐이다. 사실 개발자 편의라는 것도 옆에 있는 다른 개발자를 위한 것도 아니다. 자기 일에 대한 만족감을 표현 했을지는 모르지만, 그런 문서는 쓰레기일 뿐이다. 읽는 사람을 전혀 고려하지 않고 작성된 코드와 마찬가지로 이해하기 위해서는 오랜 “시간”이라는 비용이 추가되기 때문이다.

문서를 작성할 때도 필요한 것이 추상화다. 한 단계 더 크게 보는 것이 전체적인 맥락을 잘 파악하고 구조를 이해하는데 도움을 줄 수 있다. 큰 그림을 보고 전체적인 맥락을 파악한 후에 각각의 세부적인 그림을 보는 것이 훨씬 더 이해하기 쉽다. 모델링을 하기 위해서는 추상화 수준을 정해야 한다. 비약이 심한 추상화는 오히려 맥락을 이해하는 것을 방해하며, 너무 세세한 수준의 추상화는 전체 숲이 보이지 않도록 만든다. 따라서, 추상화되는 정보는 계층적인 표현을 사용할 필요가 있으며, 각각의 인접한 계층간 추상화 수준 차이는 갑작스러운 비약이 없어야 한다.

시스템을 전체적으로 큰 구성 요소로 나누고, 해당 구성 요소들을 다시 세부적으로 나누어서 각각을 하나씩 뜯어 나가면서 살펴보는 것이 일반적인 사람의 사고 방식이다. 계층적으로 설명할 때도 하나의 계층에 너무나 많은 것을 포함하려고 시도해서도 안 된다. 대략 “7+-2” 정도의 개수를 포함하면 충분하다. 예를 들어, 하나의 계층에는 5개에서 9개의 하위 개념의 추상화를 포함할 수 있으며, 각각의 나누어진 추상화 요소에 다시 5개에서 9개까지의 세분화된 하위 구성 요소들이 포함될 수 있다. 이런 식으로 나누어 나간다면, 보는 사람의 입장에서도 과부하 없이 전체 시스템을 차근차근 이해할 수 있을 것이다.

자신이 만든 모델이 문제 해결에 도움을 주지 못하고 오히려 이해를 방해한다면, 혹시 자신의 생각이 지나치게 세부적이거나 복잡하지 않은지 먼저 질문해야 한다. 질문에 대한 답이 긍정적이라면, 이제는 사람들과 충분히 공유할 때가 된 것이다. 모델링은 사람이 사물이나 현상을 파악하는 효과적인 도구이며, 실체를 직접 다루지 않고 정신 활동만으로 문제를 해결할 수 있도록 돕는다. 모델은 단순하지만 실체를 이미 반영하고 있으며, 세부적인 것을 생략함으로써 문제의 핵심에 더 가깝게 다가갈 수 있도록 만들어 준다. 따라서, 모델링은 인간이 할 수 있는 대표적인 정신 활동의 산물이며, 핵심은 항상 해결하고자 하는 문제에 초점을 둔 추상화에 있다.

# [ 소프트웨어를 만든다는 것 ]

모든 개발이 그렇듯이 소프트웨어도 개발하기 전에 먼저 생각을 정리하는 단계를 거쳐야 한다. 코딩 먼저 전략을 취하는 경우는 학교에서 숙제하거나 아주 잘 조직화된 팀이 아닌 경우에는 성공하기 힘들다. 물론, 학교 숙제를 하거나 조직화된 팀이라도 당연히 무엇을 어떻게 구현할지 정도는 생각하고 시작한다. 소프트웨어 구조를 설계라는 것을 “아키텍처”를 만든다고 한다. 그렇다면 무엇이 소프트웨어의 구조를 이루고 있으며, 어떻게 만드는 것이 효과적일까? 이번에는 소프트웨어 개발의 근간이 되는 구조에 대해서 잠시 알아보도록 하겠다.

소프트웨어 구조의 뼈대(Frame)를 이루는 것은 모듈과 모듈들에서 제공되는 함수다. 건축과 비교해서 이런 것들이 기둥을 이룬다고 하면, 기둥들 사이의 벽과 내장되는 인테리어들이 각각 모듈의 책임과 역할의 범위(scope) 및 내부 변수들의 선언이라고 할 수 있다. 따라서, 어떻게 이런 것들이 배치되는가에 따라 건물의 외형은 물론이고, 내부적인 안락함이나 내구성도 결정될 수 있을 것이다. 물론, 안락함을 느끼는 것은 제품의 사용자이며, 외형을 파악하고 변경에 대한 내구성을 알 수 있는 사람은 소프트웨어 개발자가 될 것이다.

소프트웨어를 만드는 것을 건축에 비유하는 것은 건축이 가장 오래된 학문 중의 하나이기 때문이다. 수학이 존재하기 전에도 먹고 마시고 자야 할 공간은 있어야 했기 때문이다. 새로운 학문의 체계를 세우기 위해서 가장 오래된 것에서 차용할 수 있는 것들을 가져다 쓰는 것은 자연스러운 일이다. 하지만, 당연한 말이지만 소프트웨어 개발은 엄연히 건축과 다르다. 소프트웨어의 “Soft”라는 단어 때문에 많은 개발자들이 피해를 보고 있다. 건물을 만들 때는 요구하지 않는 변경들이 소트웨어에서는 너무나 자주 쉽게 받아들여지고 있기 때문이다. 물론, 소프트웨어는 변경이 가능하다. 하지만, 그 대가를 제대로 지불하려고 하지 않기 때문에 문제가 심각하게 되는 것이다.

건물 내의 방들의 배치나 인테리어를 이동하는 것은 어렵지 않다. 하지만, 건물의 뼈대를 변경하는 것은 재건축을 불러올 만큼 크고 힘든 일이 될 수 있다. 소프트웨어 아키텍처란 건물 내의 방들의 위치에도 신경 쓰지만, 사실은 건물의 뼈대를 만들어가는 것이다. 잘못된 구조는 건물의 내구성에 큰 영향을 주게 되며, 조그마한 주변 환경의 변화에도 쉽게 무너질 수 있다. 붕괴는 되지 않더라도 그것에 맞게 다시 구조를 만드는 것은 비용과 시간이란 측면에서 큰 낭비를 초래할 수 있음을 명심해야 한다.

건물을 설계하는데도 몇 가지 원칙이 있다. 이러한 원칙은 이미 오랜 시간을 통해서 굳어진 것으로, 대체로 해결하려는 문제에 대한 전형적인 해결책(이미 문제에 적용해서 효과가 입증된)으로 주어진다. 소프트웨어 개발에도 이런 것들이 있으며, 아키텍처 패턴이나 "디자인 패턴(Design Pattern)"을 예로 들 수 있다. 또한, 건물을 짓는 사람들 간에 효과적인 의사소통을 위해서 도면이 필요하듯, 소프트웨어 개발에도 도면과 같은 역할을 하는 "UML(Unified Modeling Language)"과 같은 것이 존재한다. 또한, 결과물(Deliverables)을 검증받기 위해서 다양한 평가를 하듯이, 코드 리뷰나 정적 분석, 단위 테스트, 통합 테스트, 시스템 테스트, 인수 테스트 등과 같은 과정을 거치게 된다. 특히 중요한 과정은 실제로 건물을 만드는 사람들이 자신이 맡은 부분에 하자가 발생하지 않도록 확인하는 것이다. 최소 구성단위(Unit)에서 발생하는 오류가 없다는 확인은 그것을 만드는 사람이 직접 해야 한다는 말이다.

우리가 오늘 한 코딩이 "개집"을 만드는 것인지, 아니면 100층 이상 되는 랜드마크(Landmark)를 만드는 것인지 고민해서 일해야 한다. 집에서 키우는 개를 위해서는 미리 계획된 생각없이 그냥 만들어도 어느 정도 결과물이 나오겠지만, 수많은 사람이 생활할 수 있는 고층 건물을 그런 식으로 만드는 것은 상상할 수 없을 것이다. 사람들의 생명을 담보로 위험을 감수하지 않으려면, 후세에 남겨줄 수 있는 기념비적인 건축물이 되어야 하지 않을까? 소프트웨어를 만드는 것은 작은 결정들의 연속으로 이루어 진다. 작은 결정들은 큰 결정을 거스르지 않아야 하며, 전체 골격을 유지할 수 있는 형태로 나아가야 한다. 최고의 결정을 내리기 위해서는 최선을 다한 “생각”이 필요하며, 그런 생각의 정리 없이 구현에 들어가는 것은 위험 속으로 스스로를 밀어넣는 행동일 뿐이다.

# [ 객체지향으로 생각하기 ]

객체(Object)라는 말의 뜻은 무엇일까? 우리 말로는 사전에 "의사나 행위가 미치는 대상" 혹은 "문장 내에서 동사의 행위가 미치는 대상"이라고 정의되어 있다. 객체(Object)의 영어 어원은 "Ob + Ject"로 어떤 것에 대해서(Against, in the way) 던지다(Ject)라는 말을 합쳐놓은 것이다. 즉, 어떤 것을 던질 수 있는 대상이라는 말이다. 특정 대상은 가질 수 있는 특성(Attribute)가 있고, 던져진 것을 받을 수 있는 방법(Method)을 가진다. 즉, 특성과 그것을 대상으로 다루는 방법을 제공하기에 우리는 대상에게 일의 처리를 맡길 수 있게 되는(던지는) 것이다.

여기서 중요한 것은 "대상에게 일을 맡긴다"는 관점이다. 그 대상이 어떻게 일 할지는 알 수 없지만, 내가 원하는 일을 할 것이라는 것은 확신할 수 있기에 일을 줄 수 있다. 일을 맡긴 후에 우리가 기대할 수 있는 것은, 대상이 일을 처리하고 원하는 결과를 만들어서 돌려줄 것이라는 확신이다. 따라서, 대상이 하는 구체적인 일의 처리과정은 몰라도 되며, 우리의 관심사는 대상으로 부터 분리가 되어(Separation of Concern), 하고자 하는 일에만 집중하게 된다. 이렇게 대상들 사이의 관계를 설정하고, 요청을 주고받는 과정에서 하고자 하는 일이 처리되는 것이 객체 지향적으로 시스템을 만들어나가는 과정이 된다.

당연한 말이지만 개념이 부족하면 제대로 사용하는 것도 힘들다. 대부분의 초급용 책에서는 객체라는 것이 무엇인지 정확히 정의하지 않고 그냥 넘어간다. 문법을 가르치는 책은 어떻게 코딩해야 한다는 점만 강조한다. 실제로 회사의 신입 개발자들에게 클래스(Class)나 객체에 대한 정의를 물어보면 대부분 제대로 된 대답을 하지 못하는 경우가 많다. 코딩의 테크닉만 강조하다 보니, 제대로 된 이해보다는 코드를 만드는 일에만 익숙해져 버린다. 설계도 마찬가지다. 설계 과정을 생략한 상태에서 코딩에 들어가 구조적인 문제로 많은 사람을 고생하게 만든다. 객체 지향으로 코딩하기 위해서는 개념부터 이해해야 하고, 그 개념을 도구로 구현하기 위해서 언어를 사용해야 한다.

물론, 객체지향이 모든 문제를 해결해 주는 것은 아니다. C++이 아닌 C와 같은 절차 지향 언어를 사용해도 객체 지향 개념을 반영할 수 있다면, 충분히 좋은 코드를 만들수 있다. 핵심은 제대로 된 개념을 가지고 코딩하자는 것이다. 세상에는 다양한 언어들이 있지만, 그 언어들이 표방하는 개념은 대부분 큰 차이가 나지 않는다. 좋은 코드가 가져야 할 특성은 언제나 동일하기 때문이다. 따라서, 어떤 언어의 문법에 의존하기 보다는 언어를 활용해서 개념을 구현하는데 집중해야 한다. 기본기란 개념을 익숙하게 만들어야 형성될 수 있으며, 개인 개발자의 충분한 경쟁력이 될 수 있다. 무엇을 배웠다고 해서 그것을 올바로 사용할 수 있다는 것은 아니다. 언어를 제대로 사용하기 위해서는 내포된 생각과 그것을 적용한 결과물이 있어야만 가능하다.

# [ 사용자를 위해서 ]

소프트웨어 개발자로 일하다 보면 고객의 입장을 잊어버리기 쉽다. 특히 사용자 인터페이스(UI)와 같은 것을 만들 때 생기는 선택은 어쩔 수 없이 개발 편의로 가는 경우가 많다. 하지만, 정말 이것이 사용자를 위한 것일까? 사용자는 사용하기 쉽고 단순한 것을 선호한다. 이런 저런 설정을 많이 할수록 사용하기 복잡해지며 쉽게 짜증 낸다.

소프트웨어 개발자가 아닌 일반 사용자의 입장에서 자신이 구매한 제품을 생각해 보면, "왜 이렇게 복잡하게 만들었지?", 혹은 "도대체 어떻게 쓰라는 거야?" 등등 많은 불평을 한 적이 있을 것이다. 자신이 개발하는 제품에 대해서는 못했던 것들을, 남들이 개발한 제품에 대해서는 사용자의 입장에서 불평을 쏟아내고 있는 것이다. 그렇다면 왜 이런 현상이 발생할까? 사실 우리는 우리가 만드는 제품에 대해서 사용자의 입장에서 사용하는 것이 불가능하기 때문이다.

사용자가 정말 원하는 것은 무엇일까? 소프트웨어 개발자라면 적어도 사용자 요구사항을 다룬 유명한 카툰(Cartoon)은 한 번 정도 본적이 있을 것이다. 사실은 사용자 자체도 모르는 것을 개발해야 할 때도 있다. 하지만, 원칙은 오히려 간단하다. 최대한 "간단(Simple)"하게 만들라는 것이다. 사용하기 복잡한 기능이 있다고 하더라도 될 수 있으면 사용자의 직접적인 개입 없이 이용할 수 있도록 만들라는 것이다. 이를 위해서는 체계적인 사용자 시나리오의 개발이 무엇보다 필요하다. 예를 들어, UI를 개발 편의로 만들기보다 UI 전문가의 의견을 적극적으로 반영할 수도 있다. 복잡한 효과를 주기 위해서는 많은 분량의 코드가 필요하지만, 사용자들은 오히려 잘 정돈되고 쓰기 쉬운 것을 원할 수도 있다. 이렇게 간략화 시키다 보면 실제로 구현할 코드의 양도 줄어들게 될 것이다.

하나의 시스템을 만드는 데 필요한 코드를 100이라고 했을 때, 대략적으로 사용자 UI를 구현하기 위해서 필요한 코드가 50%를 넘어가는 경우가 많다. 즉, 50% 이상의 코드를 사용자가 제품을 쓰기 쉽게 만드는 데 집중해야 한다는 말이다. 또한, UI는 쉽게 변경이 될 수 있어야 한다. 사용자가 쓰기 불편한 기능의 개선을 요구하는 경우, 재빨리 그것을 반영할 수 있는 구조를 가져야 한다. 당연히 개발자는 이런 것들을 반영해서 변경 용이성도 비기능적 요구사항에 반영해야 한다.

일반적으로 시스템의 20%의 기능만이 사용되며, 80% 가까운 기능은 사용되지 않는다고들 한다. 따라서, 꼭 필요한 기능만 구조화시켜서 보여주는 것이 효과적이며, 코딩 양도 줄이는 한 가지 방법이 될 수 있다. 사실 필요 없는 기능은 코딩을 안 하는 것이 최선이다. 나중에 사용될 가능성이 있는 기능도 코딩할 필요가 없다. 미래는 예측하기 어려우며 미리 코딩한다고 해서 제대로 사용될 가능성이 있는 것도 아니기 때문이다.

# [ 나쁜 코드? 아니면, 내가 짠 코드? ]

코딩이란 인간 정신 활동의 가장 위대한 업적 중에 하나라고들 이야기한다. 왜냐하면, 절대적으로 인간만이 가진 고도의 정신 활용을 요구하기 때문이다. 그렇다면, “Bad Code”라는 것이 나오는 것은 잘못된 정신 활용이란 말인가? 먼저 “Bad Code”의 정의부터 보도록 하자. "Bad Code란 유지보수나 기능추가가 어려운 코드"를 말한다. 즉, 변경하기 힘들고 이해하기도 어려운 코드를 의미한다. 따라서, “Bad Code”란 현재는 잘 동작하고 있는 코드라고 하더라도, 언제든 쉽게 부서질 수 있으며, 조그마한 변화에도 쉽게 적응하지 못하는 코드라고 볼 수 있다.

소프트웨어 개발에 들어가는 비용의 대부분은 인건비가 차지하고 있다. 즉, 사람이 관련된 비용이란 뜻이다. 또한, 인건비는 일정에 비례해서 증가한다. 개발 시간이 지연되면 사람에 들어가는 비용이 증가하고, 시장의 기회를 잃어버리는 손실과 고객에 대한 약속을 어기는 대가를 지블해야 한다. 이런 모든 비용의 이유가 바로 "Bad Code"에서 출발한다고 보면 쉽게 이해될 수 있다. 개발자는 자신의 코드와 다른 사람이 개발한 코드를 이해해야만 새로운 기능을 구현하거나 버그를 수정할 수 있으며, 코드가 한 줄이라도 작성되어 있다면 새로운 코드를 추가하는 과정은 유지보수와 같다고 생각할 수 있기 때문이다.

인간은 기본적으로 남이 이미 해놓은 것들을 이용해서 새로운 것을 개발하려는 효율을 따지는 동물이다. 즉, 이미 해 놓은 것을 버리고 완전히 새로운 것을 개발하는 것은 자주 있는 일이 아니다. 일단 하나의 제품을 만들면 다음 제품은 당연히 이전에 개발된 것을 이용해서 몇 가지 기능을 추가하거나 개선하고 안정성을 보완해서 만들게 된다. 따라서, 처음 만들 때 잘못 만들게 되면, 치러야 하는 비용은 기존의 것을 개선하는 것과 새로운 기능을 구현하는 것을 포함하게 된다. 간혹 이런 비용들이 완전히 새로 만드는 것보다 더 커지는 상황이 되기도 한다. 예를 들어, 시장 실패 비용이 제품을 만들어 얻을 수 있는 이익보다 커져서, 제품 자체를 완전히 폐기 처분해야 할 수도 있다.

유지보수를 위한 코딩을 처음 제품을 처음 만들 때부터 확실히 하는 것이 절대적으로 중요하다. 어떤 제품을 만들고 나서 후속 제품을 적기에 출시하지 못해서 망하는 많은 소프트웨어 회사들은 아마도 처음 출시에서 제공된 실패 원인을 극복하지 못해서 나온 결과일 것이다. (혹은, 처음 출시를 성공적으로 했다고 하더라도 두 번째 출시에 너무 많은 욕심을 반영한 결과일 수도 있다.) 개발이 20%라면 유지보수와 개선이 60% 이상인 소프트웨어 개발에서, 20%에 모든 가용자원을 쏟아붓고 나면 80%는 이미 놓치고 시작하는 것과 같다. 유지보수의 핵심은 똑같은 품질의 제품을 시기를 달리해서 다른 사람이 생산할 수 있는가를 보는 것이다. 만약 그렇지 못한 조직과 프로세스(Process)를 가지고 있다면, 누가 언제 제품을 만드는가에 따라 제품의 품질을 달라질 수밖에 없다.

작은 시작은 “Bad Code”였지만 그로 인해서 회사의 흥망성쇠가 달라진다는 것이다. 어쩌면 작게 막을 수 있던 균열이 시간이 지나 댐을 무너뜨리는 것과 같은 이치일 것이다. 될 수 있으면 코드의 품질을 높이는 방법에 지속해서 투자해야 하며, 이를 개발에 적극적으로 적용하는 것이 나중에 있을 물난리를 사전에 방지해 줄 수 있을 것이다. 사물은 시간이 지나면 불확실성이 높아진다는 "엔트로피" 법칙이 있듯이, 관리되지 않는 코드도 시간이 지나면 엉망으로 변하게 된다. 하지만, 이를 미리 방지할 수 있는 나름의 방법을 만든 것은 어려운 일이 아니다. 방법을 못찾아서 하지 못하는 것이 아니라, 방법은 알지만 적용하지 못하는 상황이 오히려 더 위험하다.

아쉬운 것은 소프트웨어 개발자의 무지 속에서 나날이 유지보수 비용이 늘어난다는 점이다. 자신의 코드에 대해서 책임감을 갖는 것은 좋지만, 자만심은 버려야 한다. 열린 마인드(Open Mind)를 가진 사람일 수록 사물을 바라보는 다양한 관점이 있다는 점을 받아들이지만, 닫힌 마음(Fixed Mind)을 가지고 있는 사람은 아집을 고집처럼 믿고 산다. 더 나쁜 것은 그런 사람들의 직책이 높다면 문제가 심각해질 수도 있다는 점이다. 개인을 떠나 조직 자체에 문제를 일으키기 때문이다. 이것을 해결하는 방법은 객관화된 방법을 통해서 현실을 직시하도록 보여주는 것이지만, 사실 이것 자체도 먹혀들지 않는 경우가 많다. 따라서, 새로운 시각을 가질 수 있도록 새로운 자리로 배치하는 것도 한가지 좋은 방법이 될 수 있다. 자신의 자리에서는 보지 못하던 것을 보여주기 위해서는 관점을 달리 할 수밖에 없는 입장에 서도록 만드는 것이다.

# [ 좋은 구조 vs. 나쁜 구조 ]

소프트웨어 아키텍처는 문제를 해결하기 위한 구성요소를 정의하고 관계를 설정하는 것이 핵심이다. 문제를 해결하는 방법은 여러 가지가 있을 수 있으며 구체적인 결정을 실체화시키는 것이 아키텍처를 정의하는 과정이다. 어떤 공간이 주어졌다고 했을 때, 그 공간을 어떻게 활용할 것인가를 다룬다고 생각해 볼 수도 있다. 방과 창을 어떻게 배치하고, 화장실은 어디에 둘 것이며, 모두 다 모여서 활동할 수 있는 공간을 어디에 배정할 것인가를 정하는 것이다. 이때 배치된 공간 속에서 살아갈 사용자가 느끼는 만족감이 해당 소프트웨어 아키텍처의 좋고 나쁨을 판단할 수 있는 근거가 될 수 있을 것이다.

자신이 사는 집을 생각해보자. 기둥이나 가구 배치를 잘못하면, 옮기는 것은 사람의 힘으로는 도저히 엄두도 못 낼 정도로 어려운 일이 될 것이다. 잘못 설치한 LAN선 포트를 사용하기 위해서는 어떤 방의 문을 항상 열어두어야 할 수도 있으며, 창문이나 방문의 위치도 공간을 활용하는데 제약사항으로 작용할 수 있다. 방도 몇 개가 필요하며 어디에 배치해야 하는지 사전에 결정해야 한다. 배치된 방에는 누가 무슨 목적으로 거주할 것인지 정해야 하지만, 한 번 정하고 나면 각각의 방에 어떤 것들을 두어야 할지는 대체로 명확하다. 즉, 정해진 사람이 살아갈 공간에 그 사람이 필요한 물건들을 두면 된다.

방을 어떻게 꾸밀 것인가는 그 방에서 살아야 할 사람의 취향에 따라 달라질 수 있다. 아기가 있다면 요람을 가져다 둘 것이고, 벽지는 아마도 밝은색을 주로 사용할 것이다. 할머니나 할아버지께서 계신다면 손 때가 묻은 가구들이 놓이게 될 것이다. 작은 방에는 작은 TV가, 넓은 거실에는 55"이상이 되는 TV가 들어가고, 누워서 볼 수 있는 소파 등도 벽에 밀착해서 배치될 것이다. 가전 제품이나 가구, 소품 등의 배치는 그 사용 목적과 배치된 장소에 따라 결정될 것이다. 여기서 중요한 점은 목적과 어울리지 않는 배치 결정은 거주하는 사람에게 불편을 줄 수 있으며, 제대로 사용되지 않을 가능성이 높다는 점이다. 또한, 나중에 배치를 새로 해야 할 경우, 크기와 무게에 따라 움직이는데 필요한 시간이나 노력의 크기가 달라질 수 있다는 것이다.

좋은 구조는 주어진 공간에서 살아가는 사람에게 편안함을 안겨줄 것이고, 그렇지 못한 구조는 불편함과 함께 항상 뭔가에 쫓기는 느낌을 줄 것이다. 뭔가를 변경하려고 할 때, 편안함을 느낀다면 당신이 만든 소프트웨어는 좋은 구조를 가진 것이다. 만약, 조금이라도 변경하려는 노력에 대해서 심한 거부감을 가진다면, 오로지 당신만이 그 집의 주인일 뿐 그 집에 사는 다른 사람들에 대한 배려는 전혀 없다는 뜻이다. (혹은, 당신도 이미 타인이 된 존재일지도 모르지만). 소프트웨어 개발자도 넓은 의미에서 보면 이해관계자임과 동시에 핵심 사용자라고 볼 수 있다. 물론, 구현된 기능만 사용하는 것이 아니라, 그 기능의 구현까지도 담당하고 있다는 점에서 차이는 있지만, 편안함과 안락함을 느껴야 한다는 것은 변하지 않는 사실이다.

스스로 살 집을 고르거나 설계할 때는 이렇게 많은 고민을 하면서도, 자신이 프로의식을 가지고 하는 일에 대해서는 아무 생각 없이 하는 경우가 의외로 많다. 정말 중요하다면 자신이 평생 살 집처럼 만들어야 하지 않을까? 정말 중요하다면, 자신의 후손들까지도 편안하게 거처할 곳이 되도록, 더 많이 생각해서 구조적인 결정을 해야하지 않을까? 사용할 수 있는 모든 방법을 동원해서 가장 적합한 구조를 찾으려고 노력해야하는 것이 당연하지 않을까? 물론 잠시 살다가 떠나갈 집이라면 이런 생각을 가질 필요가 없을 것이다. 가족이라고 해봐야 혼자 밖에 없다면 불편함을 감내하는 것도 큰 어려움은 아닐 것이다. 자신과 자신의 가족같은 동료들을 생각한다면, 설계에 투자되는 노력은 결코 큰 것이 아니다. 오히려 잘못된 선택으로 발생하는 재작업으로 인한 비용이 모두의 고통이 되지 않도록 해야 할 것이다.

건물을 짓는 것이 소프트웨어를 만드는 것과 유사한 점은 많지만, 사실 다른 일이라는 것은 명확하다. 건물은 100층을 지어놓고 하부를 지탱하는 구조를 변경할 수 없지만, 소프트웨어는 100만 라인을 만들어 놓고도 구조가 나쁘니 변경하자는 이야기를 너무나 쉽게 한다. 100층짜리 건물에서 1층을 다시 만드는 것이 얼마나 힘든지 모르고 하는 소리라 흘려들을 수도 있겠지만, 소프트웨어는 항상 변화에 재빨리 대응해야 하는 것도 사실이다. 하지만, 소프트웨어는 생각보다 "Hard"하다. 우리가 할 수 있는 최선의 선택은 변화(Variability)에 대한 것을 예상하고 구조를 설계해야 한다는 것이다. 완벽하게 미래를 예상하는 것은 불가능 하지만, 일부 예상되는 변화는 사전에 고려할 수 있다. 또한, 변화가 발생한 상황에서도 다양한 방법을 통해 추가적인 변경을 한정된 범위에서 조절하도록 만들 수도 있다. 변경으로 인해서 발생하는 재작업의 폭이 넓어지는 경향을 가진다면, 변화를 위해서 필요한 구조를 미리 정의하는 일부터 해야 할 것이다.

# [ 단순하게 살자 ]

"Simple is the best"라는 말은 모두 다 알고 있을 것이다. 하지만, 정작 우리가 하는 일에 대해서는 자신이 그렇게 하고 있는지를 생각하지 않는 것도 사실이다. 지속해서 뭔가 새로운 것을 추가하기에 바쁜 우리의 삶이 반영된 것은 아닐까? 이미 가지고 있는 것을 어떻게 하면 더 잘 사용할 수 있을까를 고민하기 보다는, 부족함을 느끼며 항상 새로운 기능을 추가하는데 모든 노력을 기울인다. 사용자의 입장에서 과연 그런 것들이 다 필요할까? 제대로 사용할 수도 없는 기능들로 인해서 개발자의 소중한 노력을 혹시 낭비하고 있는 것은 아닐까?

"단순함은 간결함에서 나온다." 이 말은 꼭 해야 할 일만 해야 한다”는 뜻이다. 코딩할 때 짧은 코드를 짜기 위해서 각종 프로그래밍 팁(Tip)을 익히라는 의미가 아니며, 꼭 필요한 것 만 넣고 필요없는 것들은 제거하라는 뜻이다. 코딩 활동의 결과물인 실행 가능한 프로그램은 사용자 고객을 대상으로 하지만, 그 결과물에 포함된 소스 코드는 다른 개발자를 위한 것이다. 따라서, 소스 코드의 간결함은 다른 개발자가 코드를 읽는 데 도움을 주는 것은 포함하되, 그렇지 않은 것은 전부 제거하라는 뜻이 된다.

코드에 있어서 필요 없는 것들은 무엇이 있을까? 아마도 코드를 읽는 입장에서 가장 어려운 것은 매직 넘버(Magic Number)로 표현되는 상수 값일 것이다. 어떤 의미를 가지는지 알기 위해서는 작성자에게 직접 물어보거나 코드의 다양한 부분을 이해해야 한다. 주석으로 처리된 코드도 문제가 있다. 과감하게 지우기에는 뭔가 아쉬운 마음이 들고, 그렇다고 남겨 놓는다고 해서 사용되지도 않는다. 코드를 읽는 사람의 입장에서는 “왜 주석으로 처리했는지”에 대해서 항상 의문을 가질 수밖에 없다. 사실 이런 것들을 그냥 삭제해도 크게 문제가 없다. 이미 버전 관리 시스템과 같은 것을 사용하고 있다면, 모든 코드의 변경 기록은 항상 남아있기 때문이다.

"누가 언제 고쳤다"라는 주석이 코드를 읽는 사람에게 어떤 정보를 줄 수 있을까? 이런 주석들은 제대로 코드 관리 체계를 사용하지 않아서 발생한다. 즉, 이전에 파일 서버를 이용해서 코드의 복사본을 관리하던 습관이 그래도 남은 것이다. 사실 이런 것들은 코드를 읽는 사람에게 아무런 정보도 주지 못한다. 물론, 잘 모르겠을 때는 누구에게 물어본다는 정보 정도는 줄 수 있지만, 그 사람이 회사에 남아 있지 않을 가능성도 있다. 따라서, 정말 필요한 정보는 누가 언제 고쳤는지가 아니라, 수정한 이유(Why)를 적어놓는 것이다. 주석의 목적은 코드를 설명하는 것이 아니라 이유를 밝히는 것이다. 코드는 자주 변경될 수 있지만 이유는 지속적으로 유지되는 경우가 많기 때문이다. 그리고, 유지 보수의 효율 측면을 봤을 때도 주석은 이유를 적는 것이 올바른 선택이다.

함수 파라미터의 이름이 “x1, x2, y1, y2” 등과 같이 표현된 것보다 “name, age, weight, height”로 역할을 알려주는 것이 정보를 전달하는데 유리하다. 함수, 상수, 변수, 파라미터, 파일, 디렉토리 등도 좋은 정보를 제공하기 위해서 역할을 알 수 있는 이름이 필요하다. 미래의 누군가와 지금의 작성자가 서로 대화하기 위해서 암호 같은 코드를 이용한다면, 제대로 된 의사소통이 가능하지 않을 것이다. 코드는 사람과 사람 사이의 대화이며, 대화를 나눌 때는 명확한 의미를 전달하는 것이 핵심이다.

코딩할 때 가장 먼저 선배로부터 배우는 것이 "전역 변수를 사용하지 말라."라는 말이다. 하지만, 전역 변수를 사용하지 않으려고 하다 보면, 코딩 하기 어려운 경우가 생기고 더 빨라 보이는 지름길로 가려고 그냥 사용해 버린다. 쉽게 해결하려는 유혹이 누군가의 코칭을 제대로 동작하지 못하게 만드는 순간이다. 전역 변수가 필요한 이유에 대해서 좀 더 세밀하게 관찰하면, 모듈이나 계층화의 오류와 관련이 있다. 정보는 그것을 필요로 하는 곳에 둔다는 것이 원칙이다. 만약, 필요한 정보가 다른 모듈이나 계층에 있다면, 정보를 접근하기 위한 인터페이스를 제공해 주어야 한다. 그렇게 해야만 전체적인 전역 변수에 대한 사용을 통제할 수 있기 때문이다.

이런 저런 이야기를 하다 보니 단순하게 살기란 결코 쉬운 일이 아닌 것 같기도 하다. 하지만, 위와 같은 것들이 처음에는 복잡하게 느껴지겠지만, 지속적인 반복을 통해서 오히려 이전보다 더 "간결한 코드"가 나오도록 유도해 줄 것이다. 잘 정리되고 보기도 편한 코드는 그 자체가 힘을 가지고 있으며, 다른 개발자의 삶을 풍요롭게 만들어 줄 것이다. 코딩에 있어서는 자신도 타인에 포함해야 한다. 시간이 흐르면 코드는 기억에서 서서히 사라지게 되며, 자신이 작성했던 코드라고 하더라도 이해하기 위해서 시간이 필요하게 된다. 따라서, 남을 돕는 것이 결국 자신을 위한 길이 되는 것이다. 간결함이란 정보를 명확히 전달하는 것을 말하며, 불필요하고 구체적인 정보를 걸러 필요한 것만 보이도록 만드는 것이다.

# [ 소프트웨어 설계의 원리 ]

소프트웨어를 만들고자 할 때 생각을 먼저 하라는 말을 들은 적이 있을 것이다. 이런 생각은 자신 만이 하는 것이 아니라 여러 사람이 공유해야 하이기에, 우리는 다양한 툴을 사용해서 생각을 정리한다. 예를 들어, UML(Unified Modeling Language)를 이용해서 다양한 소프트웨어를 구성하는 클래스들의 관계를 만들고, 실행 순서 및 상태 변경, 배치 등의 그림을 그린다. 하지만, 정작 중요한 것은 그림을 그리는 방법이 아니라 내용이다. 내용을 정확히 전달할 수 있다면 형식은 도구일 뿐이다.

소프트웨어 구조를 설명하는 자료들은 대체로 네모난 박스(Box)들이 쌓인 것을 보여준다. 즉, 이런 그림들은 계층화(Layering)라는 관점에서 만들고자 하는 것을 대략적으로 크게 보여주기 위한 것이다. 내부의 관계를 조금 더 들여다보면, 상위에 위치한 박스에 들어갈 코드는 하위의 박스에서 제공되는 서비스만을 사용해야 한다는 제약 사항이 있음을 알 수 있다. 또한, 하위에 있는 박스에 들어갈 코드는 상위 박스들에 인터페이스를 제공해야 할 의무가 있으며, 상위에 위치한 코드를 호출하는 일이 있어서는 안 된다는 것이다. 따라서, 이런 제약 조건을 알지 못하고 그려진 그림은 제대로 된 소프트웨어 구조를 보여준다고 할 수 없다.

하나의 계층을 뜯어보면 상호 관련이 많은 부분이 있을 것이다. 즉, 두 개 이상의 모듈들이 서로에 대한 참조를 많이 하는 경우를 간혹 발견하게 되는데, 이 때는 맡은 역할이 잘 못 나누어진 경우라고 볼 수 있다. 모듈 A가 해야 할 일을 모듈B가 하고 있고, 모듈B가 해야 할 일을 모듈A가 하는 경우라고 볼 수 있다. 따라서, 이때는 자신이 해야하는 역할을 명확히 하고 다른 모듈이 해야할 일을 해당 모듈로 일괄적으로 옮기는 것이 바람직하다. 따라서, 프로그램의 설계는 이처럼 자신이 맡은 역할을 하나씩 세분화하면서 분리해주는 과정의 반복이 된다.

설계에서 중요한 부분은 어떤 부분에 변경이 있을 것인가를 가려내는 것이다. 시간이 흐르면 기능의 추가나 변경이 있을 것이고, 이것을 대비해서 미리 그런 부분들을 정의하는 것이 좋은 설계이다. 그렇다고, 너무 많은 가정을 하는 것은 옳지 못하지만, 변화를 감지하고 미리 그에 대한 방안을 수립할 필요는 있다. 변화는 시간이 흘러 변경이 발생하는 변화도 있지만, 동시에 같이 사용되지는 못하지만 한 번에 하나는 반드시 실행될 수 있는 것들도 있다. 또한, 여러 가지가 동시에 사용되는 변화도 있을 수 있다. 예를 들어, A라는 클라이언트(Client)는 시리얼(Serial)을 통한 연결을 지원하고, B라는 클라이언트는 이더넷(Ethernet)을 통한 네트워크를 사용할 수도 있기 때문이다. 따라서, 데이터의 전송이라는 부분에서 두 가지를 동시에 지원해야 하는 경우도 있다. 물론, 클라이언트 측에서는 어떤 통로를 통해서 지원받고 있는지 모르는 것이 좋다.

변경에 대한 대비는 다양한 방법이 있지만, 변경이 국지적으로 발생해야만 한다는 원칙은 있다. 즉, 변화가 전체 시스템에 주는 영향을 최소화해야 한다는 것이다. 국지적인 변화는 프로그램을 동작하게 만드는 각각의 구성요소들이 서로 독립성이 높을 때 가능한 일이다. 즉, 서로 간에 참조를 많이 하고 있다면 한 부분의 변화는 당연히 다른 부분에 영향을 줄 가능성이 커지기 때문이다. 또한, 특정 상황에 적합하게 성능을 최적화한 상태라고 한다면, 그러한 변화를 만들기 위해서는 상당한 시간을 소모한다. 따라서, 최적화와 일반화 사이에서 적절한 타협점을 찾는 것도 중요한 부분이다.

원칙은 "먼저 제대로 된 구조를 정의하고, 정의된 모듈에서 최적화한다."는 것이다. 즉, 좋은 구조를 먼저 만들고 나중에 최적화를 해야 한다는 것이다. 최적화를 위주로 코딩된 것을 고치기는 어렵지만, 좋은 구조를 가진 코드는 변화(최적화)에 유연하게 대응할 수 있기 때문이다. 소프트웨어의 요구사항은 지속해서 변경될 것이고, 이에 맞춰서 소프트웨어의 구조도 끊임없이 개선될 것이다. 소프트웨어는 개선의 주기가 짧아야지만 경쟁력을 가질 수 있다. 변화에 대응하는 시간이 길어지면 길어질수록 투자되는 노력의 낭비가 커지게 되며, 이는 곧 회사의 비용 부담이 커지게 된다는 의미가 된다.

소프트웨어는 고도의 지적인 활동으로 만들어가는 결과물이다. 아무리 자동화가 되어간다고 하지만 창조적인 부분에 대한 것은 여전히 사람이 관여한다. 또한, 지금까지 만들어진 어떤 것들 보다도 더 많이 사람들 간의 정신적인 협업이 강조되는 일이기도 하다. 사람과 사람 사이의 가장 중요한 상호 작용은 소통(Communication)이며, 이것을 어떻게 할 것인가를 정하는 것이 소프트웨어 구조를 설계하는 핵심이다. 즉, 조직의 구조가 소프트웨어의 구조를 정의하는 경우가 종종 있기 때문이다. 물론, 그것이 바람직하다고 보지는 않지만, 대화의 채널과 구조가 소프트웨어의 구조에 반영되는 것은 어쩔 수 없다.

소프트웨어 구조를 개선하는 데 어려움을 겪고 있다면, 조직의 구조를 변경해 보기 바란다. 이것은 역발상에 가까운 방법으로 소프트웨어 구조를 설계하고 그에 맞는 조직을 구성해서 일을 시작하는 것이 아니라, 반대로 먼저 기능적으로 구분된 조직을 만들어서 소프트웨어를 공동으로 개발하도록 요청하는 것이다. 즉, 조직의 구성에 맞게 새로운 형태의 소프트웨어를 만들어가는 협업 방식을 스스로 만들어낼 것이 분명하기 때문이다. 사람이 만들어가는 것이 소프트웨어 이기에 조직구조의 변경은 소프트웨어 구조의 설계에도 반드시 영향을 주게 된다. 물론, 대략적으로 바람직한 큰 그림을 가지고 조직을 변경할 필요는 있다. 그렇지 않다면, 어떤 그림으로 구조가 만들어질지 예상할 수 없기 때문이다.

구조의 핵심이 되는 것은 무엇일까? 주로 핵심적인 구조에 대한 것은 검증을 요구한다. 검증은 “요구되는 기능”과 “비기능적인 부분”을 다 포함하며, 핵심 기능과 비기능이 만족되어야 다른 기능이나 비기능적인 것들이 의미를 가지게 되는 부분이다. 따라서, 개발의 초기에는 이 부분에 대한 것을 집중적으로 논의해야 한다. 논의에서 중요한 것은 어떻게 만들 것인가가 아니라, “어떻게 검증할 것인가”에 중점을 두어야 한다. 만드는 것은 시간이 흘러가면 해결할 수 있지만, 만들어진 결과물을 측정하는 것은 시간이 주어진다고 해결될 문제가 아니다. 따라서, 이런 부분에 대한 고려를 설계에 미리 충분히 반영하는 것이다.

예를 들어, 어떤 소프트웨어가 외부에서 주어지는 이벤트(Event)에 반응하는 시스템을 만든다고 할 때, 주어진 입력에 맞게 움직이는 유한 상태 기계(Finite State Machine)을 만드는 것과 유사한 방식의 설계가 나올 수 있을 것이다. 사용자 테스트를 자동화하기 위해서는 외부 이벤트를 소켓(Socket)과 같은 것을 통해서 전달하는 방식을 사용할 수도 있을 것이다. 즉, 사용자 입력을 소켓을 통해서 외부에서 자동으로 넣어주는 방식으로 구현된 시스템이 동작하는지를 시나리오 기반에서 확인할 수도 있다. 검증하는 방법을 고려한 설계는 그렇지 못한 설계에 비해서 개발 시간을 단축하는데 필요한 도구를 만들어 줄 수 있다.

협업하는 방식은 다양한 개발 인프라(Infra)를 사용하는 것을 고려해야 한다. 즉, 서로 간에 시간과 공간의 차이를 두고 일하는 경우에는 협업을 위한 인프라를 구축해서, 만들어진 결과물에 대한 주기적인 빌드(Build), 분석(Analysis), 검사(Test), 배포(Release)등과 같은 것들이 적절히 피드백(Feedback)을 빨리 줄 수 있도록 해야 한다. 피드백이 빠르면 빠를수록 문제의 영향도는 줄어든다.(영향도가 줄어든다는 말은 비용이 절감된다는 의미로 해석할 수 있다.) 사용자 피드백도 빠르면 빠를수록 좋다. 빨리 문제를 찾아서 개선할 가능성이 높아지면, 다른 사용자에 대해서도 같은 효과를 발휘할 수 있기 때문이다.

이처럼 소프트웨어 설계에는 사람의 역량이 만들고자 하는 시스템의 성능을 결정짓는 중요한 요인이 된다. 따라서, 시스템을 만들고자 할 때 적절한 역량을 가지고 있는 사람을 잘 찾는 것은 대단히 중요한 일이다. 만약 그 역량을 가지고 있는 사람을 찾기가 힘들다면, 최소한 일정한 품질 수준을 꾸준히 재현할 수 있는 시스템을 구축해야 할 필요도 있다. 사람이 잘 하는 부분이 창조적인 부분이라면, 그 외의 부분은 자동화를 통해서 개발자가 역량을 키워나갈 수 있는 부가적인 도구를 사용할 수 있도록 해주어야 한다. 반복적이고 단순한 업무는 자동화를 통해서 해결하고 부가가치가 높고 창의적인 활동에 사람의 노력을 집중할 수 있는 환경을 만들어야 할 것이다.

# [ 소프트웨어 개발=코딩+테스트+그리고,문서 ]

소프트웨어 개발 활동은 코딩과 테스트를 동반하며, 내용 정리를 위해서 문서화를 포함한다. 문서화를 제외하더라도 반드시 코딩과 테스트는 개발자 수준에서 해야 할 일에 포함되어야 한다. (문서화도 중요하지만, 일단 구현(Implementation) 측면을 좀 더 강조하기 위해서 이렇게 썼을 뿐이다.) 따라서, 개발자는 자신의 코드에 문제가 없다는 것을 증명하는 것도 반드시 해야 할 일이다. 코딩과 테스트를 분리된 활동으로 생각하면 코딩과 테스트가 일어나는 시점의 분리가 일어나게 되며, 분리된 시간의 간격만큼 버그가 발생할 가능성도 높아진다.

대부분의 개발자는 자신이 구현해야 할 코드의 분량만 일로 생각하는 경우가 많다. 이럴 경우에는 테스트해서 제대로 동작한다는 것을 보장하는 시간이 포함되지 않기에, 자신이 만든 일정 계획을 달성하지 못할 가능성이 커진다. 물론 완전한 테스트는 나중에 전문 테스터가 하겠지만, 일단은 자신이 구현한 작은 단위에서의 테스트라도 통과하는 것이 중요하다. 그래야만 남들에게 제대로 일했다는 것을 알려줄 수 있는 객관적인 검증 방법을 얻을 수 있는 것이다. 100% 구현했다고 해서 100% 테스트된 코드가 되는 것은 아니다. 즉, 100% 구현했다고 해서 그 일이 종료된 것은 아니기 때문이다.

전체 개발 기간의 대부분은 구현보다 테스트 및 디버깅이 차지한다는 것은 이미 잘 알려진 사실이다. 유지보수도 포함한다면 대부분의 소프트웨어 라이프 사이클(Life Cycle)은 유지보수에 투자된다. (거의 67%이상) 따라서, 테스트 및 디버깅 시간을 어떻게 줄일 수 있느냐에 초점을 맞춰야만 생산성이 높은 개발자가 될 수 있다. 생산성이란 특정 시간에 코딩 몇 줄을 하느냐가 아니라, 얼마나 정확히 하느냐로 측정되어야 한다. 예를 들어, 100라인의 코드가 있다고 할 때, 초당 100번 수행되는 코드와 1,000,000번 수행되는 코드는 생산성 측면에서 단순히 라인 수만으로 비교하기 어렵다. 그렇다고 어느 코드가 더 중요하고 덜 중요한지 따질 수 있는 문제도 아니다. 버그는 어떤 코드에서도 발생할 수 있으며, 영향도는 사용자의 입장에서 따져야 하기 때문이다.

중요한 것은 검증된 코드가 얼마나 되며, 검증하기 위해서 필요한 노력이 얼마나 되는지 아는 것이다. 단순히 코딩만 완료된 코드는 테스트 비용과 버그 수정 비용이 투자되어야 하며, 그 비용이 얼마나 될지는 예상하기 어렵다. 또한, 검증하기 위해서 투자된 노력을 줄이지 못하면 테스트와 디버깅 간격은 더 길어질 수밖에 없다. 이처럼 생산성을 따지기 위해서는 “일의 완료 조건(Definition of Done)을 명확히 해야한다. 그렇지 않다면 일을 완료된 것이 아니며, 몇 %가 완료되었다는 말도 의미를 상실한다. 오히려 얼마나 많은 코드가 검증되었으며, 그것을 검증하는 시간이 얼마나 걸린다는 것을 알 수 있다면, 나머지 검증되지 않은 코드에서 발생하는 문제점을 파악하는데 걸리는 시간을 단순 계산으로 예측할 수 있게 된다. 이렇게 예측된 값도 물론 100% 확실한 것은 아니지만, 근거를 가지는 예측이 되는 것이다.

자신이 개발한 코드는 자신이 가장 잘 알고 있다. 그리고, 자신이 그것을 실행하는 방법도 아마 가장 잘 알 것이다. 누군가 내가 개발한 코드를 사용하고자 할 때, 내가 남긴 문서를 보는걸 선호할까? 아마 아닐 것이다. 차라리 내가 만든 코드를 컴파일해서 제대로 동작하는지를 보고 싶어 할 것이다. 즉, 모든 코드는 실행될 때 의미를 가진다는 것이다. 컴파일이 안되고 제대로 실행되지도 않는 코드를 넘겨준다면 받는 사람의 입장에서는 황당한 경험이 될 수 있다. 따라서, 최소한 자신이 만든 코드를 실행해서 간단히 테스트하는 방법을 만들어 주는 것이 좋다.

그렇다고 문서가 중요하지 않다는 것은 아니다. 문서가 담는 내용은 코드로 표현할 수 없는 고민과 결정의 과정이며, 추가적인 시행착오를 방지하기 위해서 필요한 것들이다. 문서를 나중에 작성하면 “결정에 중요한 영향을 준 것”을 포함하지 못하는 경우가 많으며, 코드를 작성하는 과정에서 고려해야 할 요소들이 빠지기도 한다. 따라서, 문서는 코딩과 테스트를 포함한 개발 활동의 일부처럼 생각되어야 한다. 물론 100% 완벽한 코드가 없듯이 문서도 100% 완결된 형태가 되지는 않는다. 즉, 꾸준히 문서를 관리할 수 있어야 한다는 의미다. 문서에 포함되지 않는 세부적인 결정은 코드를 통해서 알면된다. 하지만, 주석이나 코드로 인해하지 못하는 부분에 대해서는 문서를 통해서 설명할 수 있어야 할 것이다.

여러 사람이 작업할 경우에도 마찬가지다. 자신이 맡은 부분의 모듈을 독립적으로 테스트할 수 있는 방안을 미리 만들면, 그것을 이용해서 나중에 버그가 있는지를 검증하는 것도 한결 쉬워질 것이다. 물론, 그렇게 완전히 독립적으로 만들지 못하는 경우도 있을 것이다. 그 때도 테스트를 도와주는 툴들을 이용해서 점검할 수 있는 방법은 찾을 수 있다. 모듈의 테스트에 대해서는 이미 많은 사람이 고민했으며, 환경과 도구들도 충분하다. 사용하지 않아서 그렇지 방법은 얼마든지 찾을 수 있다. 중요한 것은 테스트도 개발자 업무의 일부라는 것을 인식해야 한다는 점이다.

흔히들 테스트는 전문 테스터의 업무고 나중에 해야 할 일로 생각하지만, 그것도 엄연히 개발자의 업무 중 일부이다. 테스터가 하지 못하는 것을 개발자는 할 수 있다. (그렇다고, 테스터의 업무가 낮은 수준이라는 말은 아니다. 전문 테스터는 개발자를 하다가 업을 바꾼 경우도 많다.) 자신이 만든 코드를 남에게 의지하는 것은 숙제를 대신해 달라고 맡기는 것과 같다. 숙제를 검사하는 것은 테스터가 할 일이지만, 숙제 자체는 담당자가 완료해야 한다. 자신이 만든 것에 대해서 자부심을 가지고 싶다면, 반드시 이런 업무를 일정에 반영하고 일의 마무리도 확실히 지어야 할 것이다.

# [ 낡은 코드의 관리를 어떻게 할 것인가? ]

우리가 만나는 대부분 코드는 이미 남이 작성해 놓은 코드다. 모든 것을 새로 작성해서 개발하기에는 주어진 시간이 부족하고, 그럴만한 실력도 없는 것이 사실이다. 어쨌든 우리가 이용할 수 있는 가장 좋은 코드는 "내가 작성하지 않은 코드"다. 즉, 남이 이미 작성해 놓은 것을 충분히 잘 이용하는 것이 효율적인 개발에서는 중요하다. 물론 이것이 항상 최선의 길은 아니며, 내가 원하는 속도를 내기 위해서는 직접 코딩을 해야 할 경우도 있다. 하지만, 될 수 있으면 남이 이미 개발해 놓은 것을 이용하는 것이 종종 더 좋을 결과로 이어지는 경우가 많다.

어쨌든 우리는 낡은 코드와 씨름을 해야 하는 것이 일상이다. 작은 수정이 일으킬지도 모르는 일에 대해서도 걱정을 해야 하며, 변경 결과를 확인하는 것도 즉각적으로 할 수 없어서 테스트 팀에 코드가 넘어갈 때까지 기다려야 하는 것도 현실이다. "동작하는 코드를 수정하지 말라"는 속담이 있다는 것은 이미 잘 알 것이다. 문제는 그 동작하는 코드를 수정해야 할 필요가 있을 때 발생한다. 소프트웨어 개발자는 기능을 추가하거나 버그를 수정할 경우 생기는 예상하지 못하는 부작용(Side-Effect)에 대해서 어떻게 품질을 보장할 것인가를 다루어야만 한다.

이용할 수 있는 가장 좋은 방법은 낡은 코드에 의존적이지 않은 새로운 코드를 짜서 기능을 서서히 대체해 나가는 방법이다. 하지만, 이것은 기존 코드가 남아 있는 상황에서 사용하기 위해서는 많은 제약이 따른다. 또한, 코드의 중복도 어쩔 수 없이 따라오는 문제이기 때문에 그렇게 선호되는 방법은 아니다. 그리고, 시간도 생각보다 오래 걸릴 가능성이 높다. 따라서, 우리는 남이 이미 작성한 코드를 어쩔 수 없이 수정하거나 확장할 수밖에 없다. 새로운 기능을 구현하기 위해서는 기존 코드에서 분기하는 것이 좋지만, 이미 작성된 코드를 무시하는 것은 또 다른 위험 부담이 될 수밖에 없다.

시스템의 조그마한 변화는 이전에 동작하던 기능들이 변화가 없음을 반드시 보장해 주어야 한다. 따라서, 우리가 선택할 수 있는 변화는 코드가 관리되는(Managed) 상태로 항상 만들어 두어야 한다는 것이 된다. 이것은 기존 코드가 차지하고 있는 부분에 대한 입력과 출력이 동일하게 잘 유지가 되고 있음을 보장해야 한다는 것이다. 즉, 기존의 낡은 코드에 대한 입출력 기록을 유지하고, 변경된 코드에서도 제대로 나오는지 확인할 방안이 필요하다. 이를 확인하기 위한 테스트 주기도 당연히 짧아야 한다. 변경과 테스트의 간격은 버그의 유입 가능성에 영향을 주기 때문이다. 테스트 대상은 모든 변경을 포함해야 한다. 따라서, 이런 테스트를 가능하게 만들기 위해서는 개인 개발자가 테스트 케이스를 직접 작성하고 실행할 수 있어야 한다.

변경된 부분에 대한 점검은 하루의 일과 중에 수시로 해야할 필요가 있다. 변화의 부분을 최소화하고 빨리 테스트하는 것도 필요하다. 너무 많은 변화가 동시다발적으로 이루어지면, 변화는 관리할 수 없는 수준까지 도달하게 되며, 이런 상황이 반복되면 과제를 언제까지 완료할 수 있다는 예상을 아무도 할 수 없게 된다. 지속적으로 버그가 발생하는 상황에서는 닫혀지는 버그 수와 새로 열리는 버그의 수가 평행선을 그리기 때문이다. 즉, 버그 발생 추세가 누그러지지 않기 때문에 일정 그래프의 버그 개수가 0이 되는 시점이 그려지지 않게 된다.

낡은 코드를 “레거시 코드(Legacy Code)”라고 부르는 이유는 테스트가 없어서 변경의 결과를 예상할 수 없기 때문이다. 기능추가는 기존의 코드를 감싸서 해결할 수 있지만, 기존 코드의 변경은 어떤 결과를 만들어 낼지 알 수 없다. 최선의 방법은 기존의 코드를 가능하면 건드리지 않고 변경할 수 있는 부분을 분리 구현하는 방법이다. 이 때도 잊어서는 안되는 것은 테스트를 추가할 수 있도록 구현해야 한다는 점이다. 추가적인 레거시 코드를 만들고 싶지 않다면, 조금이라도 테스트할 수 있도록 만들어야 한다. 이것이 비록 지금 당장은 어렵더라도 조금이라도 미래에 갚아야 할 빚(Debt)을 줄이는 대가가 큰 투자다. 미래의 또 다른 내가 경험하고 싶지 않은 버그라면 지금 줄여주는 것이 최선의 선택인 것이다.

# [ SOLID에 대해서 ]

복잡한 것을 쉽게 풀어 쓸 수 있는 사람은 정말 대단한 능력을 갖춘 것 같다. 다 아는 사실이지만 “아는 것”과 “가르치는 것”은 다르다. 정말 잘 알고 있어야 하며, 직접 경험해봐야 쉽게 가르칠 수 있다. 코딩 경험이 많은 사람은 자신의 경험을 바탕으로 “좋은 코드”를 만드는 방법을 이미 여러 책에서 많이 읽었을 것이다. 아래는 그 방법의 하나인 "SOLID" 원칙(Principle)이다. 수식으로 증명된 것은 아니지만, 대가들의 경험에서 나온 말이니 한 번쯤 생각해 보고 지나가는 것이 좋을 것이다.

**1. Single Responsibility Principle(단일 책임 원칙)**

; "한 번에 한 가지 일만 하라"는 말이다. 당연한 이야기겠지만, 작은 프로그램 일수록 버그를 발생시킬 확률은 그 만큼 더 적다. 한가지 일 만 하는 함수나 클래스 일수록 버그가 있을 가능성도 당연히 낮아진다. 여러 가지 일을 처리하기 위해서는 전달해야 할 정보도 늘어나게 되며, 내부의 로직(Logic)도 복잡해지게 된다. 따라서, 작게 단일 역할만 충실히 수행하는 단위로 코드를 만드는 것은 전체적인 복잡도를 낮추고 버그가 스며들 기회를 줄여줄 수 있다. 될 수 있으면 코딩을 적게 하는 것이 중요하고, 한 번에 하나의 일만 처리하게 하는 것이 버그에 효과적으로 대응하는 방법인 것이다. 추가적으로 한 가지 일만 하기에 변경의 이유도 그 하나의 일에 관련된 것으로 줄어든다. 당연히 다른 코드에 주는 영향도 줄어들게 될 것이다. 한 번에 한 가지만 하는 것들을 많이 만들어 둘수록, 그런 것들에 대한 재활용 가능성도 당연히 높아지게 된다.

**2. Open-Closed Principle(개방 폐쇄 원칙)**

; "클래스는 확장을 위해서 개방적이고, 수정에 대해서는 폐쇄적이어야 한다"는 말이다. 이것은 자신이 하는 일을 남들이 모르게 하라는 의미로 해석할 수 있다. 남을 클라이언트라고 보면 클라이언트는 서버가 어떻게 요청을 처리하는지를 몰라야 한다. 그래야만 서버의 변경이 클라이언트에 영향을 주지 않게 되기 때문이다. 물론 서버는 클라이언트와 계약(Contract)된 일을 정확히 처리하여야 할 의무가 있다. 하지만, 그 계약만 충실히 지킨다면 내부에서는 어떤 변경이 있어도 상관없다는 뜻이 된다. 확장은 내부의 일이 되지만 변경의 영향이 외부에는 영향을 주지 않는 것이다. 이것을 조금더 확장하면 모듈 내부의 변경이 외부에 영향을 주어서는 안되며, 모듈은 자신의 인터페이스를 통해서 제공되는 기능은 충실히 수행해야 한다는 의미가 된다. 따라서, 모듈은 외부에 영향을 주지 않고도 내부구조의 변경을 쉽게 할 수 있어야 한다.

**3. Liskov Substitution Principle(리스코프 대체 원칙)**

; "상위 클래스는 하위(상속받는) 클래스로 대체될 수 있어야 한다."는 말이다. 이것은 상속 계층에서 최상위의 클래스를 서버로 보고, 클라이언트들이 접근하게 될 때, 서버 클래스에서 제공하는 서비스들은 모두 다 상속받는 클래스에서 제대로 제공해야 한다는 것을 의미한다. 즉, 제대로 된 상속이 사용된다는 것을 보증하는 것으로 상속에 대한 정당한 이유가 있는가를 확인해야 한다는 의미다. 사실 상속은 여러 가지 의미에서 다소 위험이 있는 것이 사실이다. 즉, 상속의 계층이 길어질게 되는 경우, 상위 클래스에서의 변경이 하위의 상속받는 모든 클래스에 영향을 주기 때문이다. 세밀하게 검토된 상속이 아닌 경우에는 안 하는 편이 나을지도 모른다. 상속을 대신해서 인터페이스를 사용하는 경우가 대표적이다. 대체란 정해진 계약을 수행할 수 있는 것이라면 어떤 것이라도 상관없다는 뜻이며, 인터페이스만 동일하다면 그것을 사용하는 측에서는 계약이 충실히 실행될 것이라고 가정한다.

**4. Interface Segregation Principle(인터페이스 분리 원칙)**

; "모든 일을 다 처리하는 인터페이스를 만들기 보다는, 한 가지 일만 잘 처리하는 것을 만들라."는 말이다. 복합적인 일을 처리하는 인터페이스를 구현하기보다, 한 번에 한 가지 계약(일)만 제대로 수행하는 것을 만드는 것이 좋다는 뜻이다. 사람은 여러가지 일을 한 번에 처리하는데 익숙하지 않으며, 생각도 병렬(Parallel)로 할 수 없다. 코딩에 실수가 자주 발생하는 이유는 이처럼 익숙하지 않은 방법으로 여러가지 일을 한 번에 처리하고자 하는 욕심 때문이다. 코드가 스파게티처럼 엉키는 이유도 마찬가지다. 따라서, 외부에 제공할 인터페이스 각각을 하나의 목적을 구현하도록 세분화를 시켜야 한다. 세분화된 인터페이스는 한 가지 일에 집중할 수 있기에 당연히 버그가 발생할 가능성도 낮으며 전달되는 정보의 양도 적어진다. 내부 구조가 단순화될 것이기에 복잡도도 낮아질 것이 분명하다.

**5. Dependency Inversion Principle(의존성 역전 원칙)**

; "코드는 구현이 아니라 추상적인 것에 의존해야 한다."는 말이다. 남이 어떤 일을 하는지는 사실 그 일을 직접 처리하지 않으면 알 길이 없다. 즉, 우리는 어떤 사람에게 일을 처리해 달라고 요청할 뿐이지, 구체적으로 어떻게 해서 처리하라고 까지는 요구하지 않는다. 마찬가지로 상세 구현에 의존적인 코드를 만든다면, 자그마한 변경에도 문제를 만들어 낼 것이기 때문이다. 즉, 추상적인 것은 구체적인 것에 의존하지 않아야 하며, 오히려 구체적인 것이 추상적인 것에 의존하도록 만들라는 의미다. (보통의 경우 추상적인 것이 구체적인 것에 의존적이지만, 이 경우에는 구체적인 것이 추상적인 것에 의존하고 있어서 "의존성이 바뀌었다"는 말로 표현되었다.) 상위 계층이 의존할 수 있는 것은 하위 계층에서 제공하는 인터페이스에 한정되며, 하위 계층은 상위 계층이 사용하는 인터페이스를 제대로 구현할 의무를 가진다. 따라서, 상위 계층이나 하위 계층 모두 추상적으로 제공되는 인터페이스에 의존하는 관계가 되며, 내부의 구체적인 것에 의존하지 않게 된다.

SOLID원칙은 주로 객체지향 프로그래밍에서 사용하는 가장 대표적인 원칙이지만, 객체지향 프로그래밍에서만 사용할 필요는 없다. 절차지향 언어를 사용하더라도 위에서 이야기하는 원칙에 맞도록 코딩한다면, 변화에 유연한 코드를 만들 수 있다. 따라서, 특정 언어에 의존해서 생각의 범위를 축소시키기 보다, 언어가 가진 개념을 활용할 수 있는 코드를 구현하는 것이 더 중요하다. 이런 기본적인 원칙들을 꾸준히 지켜나가다 보면 개인의 역량도 향상될 것은 분명하다. 부가적인 결과물로 “품질이 우수한 코드”도 함께 따라올 것이다.

# [ 좋은 이름 짓기 ]

소설 속에 등장하는 인물들의 이름을 눈여겨본 적이 있는가? 대체로 이름이 이상한 사람은 웃기는 캐릭터나 악당으로 나온다. 시중을 들거나 험한 일을 하는 사람들의 이름은 심하게 이야기하면 촌스런운 이름을 가진 경우가 많았다. TV 드라마 속의 사투리를 쓰는 사람들도 맡은 배역이 십중팔구 집에서 일하는 가정부거나 낮은 계층의 사람들인 경우가 많았다. 요즘은 그런 것들이 많이 완화되었지만, 이름이나 들리는 말의 어감으로 역할을 해석하는 것은 어쩔 수 없다. 우스꽝스러운 이름이나 사투리는 사람이 가진 지식이나 사회적 지휘와 상관없이 우리에게 편향을 만들어내고 있는 것이다. 하지만, 이를 잘 활용하면 길게 설명하지 않고도 충분한 정보를 효과적으로 전달할 수도 있다.

좋은 이름은 효과적으로 정보를 전달한다. 적절한 이름을 가지고 있어야만 그 존재를 알고 사용할 수 있게되기 때문이다. 인터넷 주소를 지정할 때 사용되는 IP 주소는 기억하기 어렵지만, DNS를 이용한 글자로 된 주소는 기억하기 쉽다. 전화번호를 기억하기 보아는 주소록에서 이름으로 전화를 거는 것이 편하다. 계좌번호를 일일이 기억하기 어려워서 자주 사용하는 계좌를 이름으로 등록하는 경우도 마찬가지다. 좋은 이름이 되기 위해서는 그 이름으로 대표하는 역할을 연상할 수 있어야 한다. 그리고, 실제로도 그 이름이 가진 역할 이상으로 해서도 안된다. 추상화란 이름을 사용해서 사물이나 현상에 역할을 부여하고 이해하는 것이다. 그리고, 우리는 이미 이런 것들을 일상에서 너무나 쉽게 사용하고 있다.

코드에 등장하는 여러 구성요소들에 대한 이름을 어떻게 지을 것인지 선배들로 부터 들었던 규칙들이 있을 것이다. 코딩 룰과 같은 것으로 정해진 규칙이 있을수도 있고, 혹은 어깨 너머로 이야기 들었던 경우도 있을 것이다. 하지만, 대체로 이름 자체를 만드는 규칙이지 이름이 전달하는 정보에 대한 것은 아닐 것이다. 중요한 것은 그 이름이 충분히 역할을 표현할 수 있는 능력이 있어야 하며, 그 이상도 그 이하도 아닌 딱 그 정도의 일만 처리하도록 내부의 코드를 구현하는 것이다. 대부분 이런 일은 코드를 구현하기 전에 미리 계획되어야 한다. 그리고, 코딩 과정에서 발생하는 다양한 생각의 추상화 수준 차이는 새로운 내부의 이름을 만들어내서 처리하도록 해야 한다. 이름을 짓고 그 이름이 하는 역할의 경계를 명확하게 코드로 표현하는 것은, 마치 계약을 만들고 충실히 그 계약을 수행하는 것과 같다고 볼 수 있다. 어긋난 계약은 신뢰를 낮추게 되며, 잘못된 행동을 유발할 수 있다.

좋은 이름은 의도를 가감없이 전달할 수 있어야 한다. 오해를 불러일으킬 수 있는 모호한 이름이나 여러가지 의미를 가진 이름은 사용하면 안된다. 코드를 읽는 사람의 입장에서 혼동하지 않고 명확히 역할을 파악할 수 있는 이름이 좋은 이름이다. 이름으로 그것이 하는 역할을 유추할 수 있다면 충분히 좋은 이름이라고 할 수 있으며, 구현하는 입장에서는 이름이 가지는 역할 이상의 내용은 내용에 포함하지 않아야 한다. 이름이 가지는 역할은 반드시 하나로 구체적으로 정해질 수 있어야 하며, 구현 내용에는 이름으로 이해했던 역할을 충실히 구현해 주어야 한다. 만약, 코드를 읽었을 때 이름에서 알 수 없는 부분이 있었다면, 이는 코드를 수정해야 한다는 의미가 된다. 좋은 이름이 가져야 할 규칙도 중요하다. 어떤 이름을 짓거나 읽었을 때, 일관된 인상을 코드를 읽는 사람에게 주어야 하기 때문이다. 나중에 새로운 이름을 추가할 때도 같은 방식으로 결정된 이름을 사용해야 한다.

코딩이란 결국 프로그래머의 의도를 언어를 통해서 독자에게 전달하는 것이다. 독자는 사용자가 아니며, 내부의 개발자나 공개된 코드를 시간과 장소에 제약받지 않고 읽고 변경할 사람들이다. 그들에게 명확한 의도를 전달하는 것은 경제적으로도 중요한 행위이며, 코드의 생명력을 유지할수 있는 확실한 방법이다. 좋은 이름을 만들고 관리하는 것 자체는 큰 노력이 필요한 것이 아니지만, 그것을 읽고 이해해야 불특정 다수의 노력을 절약하기 위해서는 반드시 필요한 일이다. 내가 아닌 타인의 시선으로 코드가 무엇을 하는지 설명할 수 있다면, 그것 자체가 자신에게도 도움이 되는 일이다. 미래의 독자에 빠질 수 없는 사람이 바로 코드 작성자 자신이기 때문이다. 살이 있는 코드는 끊이없이 수정되고 발전해가기 마련이다. 읽기 어렵고 이해하기도 어렵다면 당연히 변경하기도 힘들다. 그런 코드는 타인뿐만 아니라 자신에게도 결코 유쾌한 경험이 아닐 것이다.

# [ 주석(Comment)에 대한 오해 ]

코딩을 하다 보면 코드 이외에 가장 자주 보는 것은 아마도 주석(Comment)일 것이다. 자동으로 만들어진 것도 있지만, 코드를 작성한 사람이 자기 생각이나 의무감에서 만들어둔 주석들을 보게 된다. 하지만, 주석 중에서 쓸모 있었던 것들이 생각해보면, 그렇게 도움이 된 경우는 많이 없었을 것이다. 역시 주석보다는 코드를 이해하려고 노력할 것이다. 그리고, 당연히 그것이 정답이다. 주석으로 생각을 표현하기 보다는 코드 자체가 이해되는 것이 먼저이기 때문이다.

의미 없는 주석들의 대부분은 개발자의 편의 때문에 만들어진다. 물론, 요즘 같은 자동화가 잘 되어 있는 편집기를 사용하거나, IDE(Integrated Development Environment: 통합 개발 환경)를 가지고 작업하는 경우에는 개발자가 만들지 않더라도 많은 주석이 들어가지만, 의미 없는 대부분 주석은 여전히 개발자의 몫이다. 문제는 이런 주석들에 대한 관리가 부실하다는 점이다. 누가 남기는지를 알고 싶지 않더라도, "2013.12.26 xxx"라는 것들이 흔히 보인다. 누가 이렇게 했으니 그 사람의 책임이라는 뜻인가? 아니면, 누군가 그렇게 만들어 두었으니 고치지 말라는 의미인가? 무의미한 주석은 없는 것보다 못하다.

의미가 있는 주석이 되기 위해서는 관리가 되어야 한다. 누가 남겼는가는 이미 알 수 있는 방법이 충분히 있다. 예를 들어, 웬만한 소스 코드 관리(버전 관리) 도구들은 전부 체크인(Check-In) ID를 가지고 라인별로 누가 만들었는지 전부 구분한다. 코드를 가지고 개발에 적용하고 있는 모든 사람이 관리의 대상이며, 필요 없는 주석을 제거하는 것도 개발자들의 몫이다. 코드는 항상 깔끔하게 유지가 되어야 하는 것은 잘 알면서도, 코드에 들어가 있는 주석은 관리하지 않는다. 잘못된 주석은 오해를 낳고 오해는 버그를 만든다는 사실을 알아야 한다. 이렇게 만들어진 버그는 원인을 찾는 것도 어렵다. 당연히 될 것이라고 믿었던 것에 배반당할 수 있기 때문이다.

“Doxygen”과 같은 형태의 주석은 의미가 있는 것일까? 사실 API(Application Programming Interface)를 자동으로 문서화시키고자 하는 노력은 많이 해봤을 것이다. 항상 최신의 API 문서를 제공할 목적으로 “Doxygen”과 같은 것을 이용해 본 경험도 있을 것이다. 가장 힘들었던 점은 무엇인가? 코드의 수정이 “Doxygen” 주석의 수정으로 이어지지 않는 경우에 문제가 발생한다는 점이다. 즉, 코드와 “Doxygen” 주석을 함께 유지하는 것은 쉬운 일이 아니다. 따라서, 만약 “Doxygen”을 활용할 생각을 하고 있다면, 반드시 코드와 일치시키기를 권장한다. 또한, 내부적으로 사용되는 것들에 대한 주석보다는 외부로 보여주어(Export)야 할 것들만을 챙기기 바란다. 될 수 있으면 그런 것들로 한정해서 사용한다면 관리 노력이 절감될 것이다.

주석에 눈길이 잘 가지 않고 코드에 더 많이 주의를 기울인다는 사실에서, “주석이 없어도 이해가 되는 코드”를 짜야 한다는 것이 정답이다. 코드 자체가 이해가 안 되는 상황에서 무조건 주석이 없는 코드가 좋다고 이야기하는 것은 옳지 않다. 코드의 품질은 읽는 것에서 출발해서 실행으로 끝난다. 따라서, 잘 읽히는 코드가 좋은 코드이지 주석이 많은 코드가 좋은 코드는 아니다. 주석이 필요한 이유는 말 그대로 코드로서는 도저히 표현할 수 없는 "이유(Why)"를 설명하기 위한 것이다. 좋은 코드는 주석의 도움이 필요한 것이 아니라, 그 자체만으로도 충분히 읽는 사람에게 의도를 전달할 수 있어야 한다.

# [ 함수의 길이에 대한 단상 ]

보통 하나의 함수(혹은 클래스의 메쏘드(method))가 차지하는 길이(명령 라인의 수, 하나의 라인에는 하나의 명령문이 있다고 가정)가 얼마가 되면 적당할까? 이런 고민은 프로그램을 짜는 사람이면 한 번쯤은 해봤을 것이다. 해답은 이미 나와 있다. 즉, "짧으면 짧을수록 좋다"이다. 하지만, 짧다는 것의 정의가 어느 정도인지 다시 의문이 든다. 사실 이것은 상대적인 개념이다. 절대적인 수치를 제시하기는 어렵고, 대략 “한 번에 하나의 일을 처리하는 것”이 적절한 크기라고 보면 된다. 논리적으로 이렇게 구분한다면 지키기 어려운 규칙이 되기에, 물리적인 한계를 설정하는 것이 편하다.

하나의 일을 처리한다는 것의 전제는 추상화 수준에 달려있다. 예를 들어 "네트워크를 통해서 데이터를 보낸다"는 수준과, ”연결된 소켓에 버퍼 포인터가 가리키는 바이트(Byte)를 쓴다”는 수준의 추상화는 완전히 다르다. 즉, 동일한 추상화 수준에서 한 가지 일을 처리하는 함수를 만드는 것이 핵심이다. 그리고, 그 함수는 당연히 짧아야 한다. 함수 내부에 존재하는 모든 명령문은 동일한 추상화 수준이 되어야 하고, 그렇지 못할 경우에는 한 수준 더 낮은 함수로 분리 구현해서 호출을 통해 해결해야 한다.

이런 식으로 동일한 수준에서의 한 가지 일만 수행하는 작은 함수들을 엮어 나가게 될 때, 어떤 사람은 함수 호출 오버헤드가 크다고 이야기할지도 모른다. 그 만큼의 오버헤드가 시스템에 미치는 영향이 과연 생각하는 것처럼 그렇게 클까? 그보다 더 높은 가치가 있는데, 그런 것에 얽매이는 것이 올바른 것일까? 첫 번째 질문에 대한 대답은 “영향은 미미하다는 것”이다. 두 번째 질문에 대한 대답은 “좋은 구조와 쉬운 이해가 항상 모든 것에 앞서야 한다”는 점이다. 사실 함수 호출은 복귀 주소의 저장과 레지스터 수준의 인자 전달, 명령과 데이터 캐쉬의 변경이 있을 수 있다.

오히려 긴 함수를 만들게 되면 사용하는 전달받아야 할 정보가 늘어나게 되며, 처리에 따른 로직이 복잡해질 가능성이 크다. 또한, 인자 수가 많아지면 기억해야 할 정보가 늘어나 함수의 이해를 방해하고 테스트도 어렵게 만든다. 단위 테스트를 할 때 필요한 입력의 범위와 가지수가 늘어나기 때문이다. 될 수 있으면 인자의 개수를 최소로 유지하는 것이 좋다. 전달받는 정보가 없다면 가장 좋겠지만, 4개 이하로 유지할 수 있다면 충분하다. 컴파일러나 CPU는 함수를 프로그래머가 기본적으로 사용할 것이라고 믿기에, 이미 함수 호출에 대한 많은 최적화 방법을 구현하고 있으며 일반적으로 생각하는 것보다 큰 오버헤드는 발생하지 않는다.

함수의 길이를 고려할 때 개념적인 추상화 수준이 같은가를 기준으로 해야한다. 전달받는 인자의 수도 최소화시켜야 하고, 내부에 “if, for, while, switch” 등의 코드 블록들도 적으면 적을수록 좋다. 중첩된 것들은 똑똑한 사람들의 전유물이지, 우리 같은 평범한 개발자를 위한다면 될 수 있으면 없어야 한다. 차라리 그 부분을 분리해서 함수로 만드는 편이 옳다. 작은 함수는 테스트하기도 편하고 이해하기도 적당한 수준이 될 것이다. 따라서, 당연히 오류가 줄어들 것은 분명하다. 지나치게 똑똑한 사람이 자신만의 독특한 코드를 짜는 것을 용납해서는 안 되는 이유가 여기에 있다. 버그를 찾기위해서 코드를 읽는 시간이 길어지면 길어질수록, 개발자의 생산성도 비례해서 함께 낮아질 것이기 때문이다.

일 년에 대략 20,000라인(일반적으로 그렇다고 이야기한다. 하지만, 실제로 이 정도는 많이 작성하는 편이다. 물론 사용하는 언어에 따라 작성하는 코드의 길이도 다르다.)의 코드를 1명이 개발한다고 가정할 때, 우리는 하루에 대략 100라인 정도(200일 정도 일 년에 일한다고 가정할 때)를 코딩할 것이다. 하루에 10라인으로 이루어진 10개의 제대로 된 함수를 만들고, 이를 테스트할 수 있는 스크립트를 만들어 오류가 없다는 것을 확인할 수 있다면, 우리가 할 일은 훨씬 줄어들 것이다. 10라인으로 된 함수는 다른 사람이 디버깅 한다고 해도 쉽게 버그를 찾아낼 수 있을 것이다. 100라인으로 된 함수를 만들기 위해서는 필요한 인수도 늘어나고, 테스트를 만들기도 어려워진다. 또한, 그것을 한 번에 읽어서 이해하고 디버깅 하기 위한 노력도 배가 될 것이다.

생산성의 비밀은 사실 이렇게 사소한 것들에 담겨 있다. 남들보다 높은 수준의 개발 생산성을 달성하는 것은, 결코 힘들고 어려운 일에 달린 것이 아니라 자그마한 지속적인 노력에 좌우된다. 무지가 용서되는 것은 남들에게 피해가 가지 않을 때다. 소프트웨어 개발에서 발생하는 무지는 자신만이 아닌 남들에게도 큰 피해를 주게 된다. 잘 이해가 안 되는 알고리즘이나 아키텍처의 수정보다는 현실적으로 눈 앞에 놓인 함수의 크기를 줄여나가는 것이 오히려 큰 도움이 될 것이다. 간단한 방법이지만 효과가 확실하기에 충분히 시도해 볼 가치가 있다. 반대하는 사람들은 대부분은 성능 이야기를 하지만, 오히려 작은 함수로 구현된 코드에서 어디가 병목구간이 되는지 확인하는 것이 최적화에도 도움이 된다. 짧고 작게 나누어지 함수들은 확인된 병목구간에 한정해서 다시 합쳐질 수도 있다.

# [ 함수의 인자수는 몇개가 좋을까? ]

프로그램을 실행하기 위해서는 최소한 하나의 함수는 필요하다. 하지만, 하나의 함수가 가질 수 있는 인자 수는 정하지 않고 개발자의 편의로 만드는 경우가 많다. 특히, C나 C++ 등의 언어로 코딩을 하게 될 때는 프로그래머가 임의로 인자 수를 늘리는 경향이 있다. 하지만, 이것도 원칙이 필요한 일이다. 인자 수가 늘어나게 되면 함수를 사용하는 다른 사람들은 각각의 인자가 어떤 역할을 하는지 알아야 한다. 따라서, 많으면 많을수록 복잡한 프로그램이 될 확률이 높아지기 때문이다. 사람의 뇌는 단기 저장소에 기억할 수 있는 용량에 한계가 있으며, 가능한 적은 것을 보관할수록 효과적으로 이해를 도울 수 있다.

일반적으로는 대략 4개 정도까지만 인자를 가져가라고 이야기한다. 하지만, 인자가 없는 함수가 가장 이해하기 편하고 사용하기도 쉽다. 따라서, 결론은 인자가 없는 함수를 만들라는 것이다. 인자가 없다면 사용하는 측에서는 함수가 어떤 역할을 하는지만을 알면 된다. 잘 만들어진 함수라면 함수의 이름 자체가 자신의 역할을 대변해 줄 것이다. 즉, 함수 자체도 한 가지 역할에 충실히 만들어져야 좋은 이름을 붙일 수 있다. 이름이 머릿속에 잘 떠오르지 않는다면, 2가지 이상의 역할을 하나의 함수가 수행하고 있을 가능성이 높다. 이때는 각각을 나누어서 더 작은 이름을 붙일 수 있는 함수로 만들면 된다.

함수의 복귀값은 반드시 있어야 한다. 복귀값은 수행된 결과를 확인해서 함수가 제대로 동작했는지 알 수 있는 유일한 길이다. (물론, 인자에 그러한 역할을 주는 경우도 있지만, 이는 함수의 이해를 더 어렵게 만들 가능성이 있다. 함수는 입력에 대해서는 어떤 변경도 하지 않는 편이 좋다.) 함수의 복귀값은 반드시 검사해야 한다. 평소에 거의 에러(Error)를 발생시키지 않는 함수라고 할지라도 반드시 복귀값을 검사해야 한다. 흔히 초보자들이 실수하는 부분이 함수의 호출 후에 복구값을 확인하지 않고 막연히 성공했을 거라고 추측하는 것이다. 따라서, 모든 함수의 복귀값은 반드시 확인해야 한다.

함수의 복귀값은 단위 테스트에도 중요한 역할을 하므로 만들어두는 것이 좋다. 즉, 단위 테스트의 "단위"가 함수 단위이기 때문에 함수의 호출 후 복귀값이 어떻게 되는가를 가지고 “확인(Assert)를 하기 때문이다. 각각의 함수가 전역변수를 공유한다면 문제는 복잡해진다. 단위 테스트는 매번 전역변수에 대해서 가정을 해야하기 때문이다. 파일 범위 내에서만 전역변수로 사용하면 되지 않겠느냐는 생각도 하지 않아야 한다. 공유되는 자원이 최소화될 수 있도록 만들어야 각각의 테스트 케이스들도 독립성이 높아질 것이고, 특정 상황(Context)에 의존적인 코드가 줄어들 것이기 때문이다. 병렬로 처리하거나 멀티쓰레드를 위해서도 전역변수는 없애는 것이 좋다.

함수의 인자를 없애기 위해서는 작은 함수를 만드는 것이 필수적이다. 최소한의 인자를 사용하기 위해서는 인자로 넘겨주는 값을 캡슐(Capsule)화시키는 것도 한 가지 방법이다. 즉, 캡슐화된 인수를 전달받아서, 그 내부의 값을 알아내는 방법까지 존재한다면 그것을 이용하는 것이 최선이다. 모든 캡슐화의 근본 이유는 변화에 대한 영향을 최소화시켜주기 때문이다. 내부 자료구조에 의존적인 코딩은 저장된 상태 및 자료구조 자체의 변경으로 인해서 버그가 발생할 가능성이 있지만, 인터페이스(Interface)에 의존적인 코드는 그런 오류를 사전에 제거할 수 있다. 캡슐화는 함께 사용되어야 할 자료를 묶어줌과 동시에 함께 사용해야 할 인터페이스도 제공해 준다.

# [ 객체지향 이야기 ]

사람과 사람 사이의 관계가 있듯이 객체들 간의 관계가 코드의 구조에 대한 큰 영향을 주게 된다. 객체들이 서로 아무런 관계가 없다면 서로 다른 시스템에 속한 것이며, 우리가 개발해야 할 시스템의 범주를 넘어선다고 볼 수 있다. 사람과 사람 사이의 관계는 가까울수록 좋다고 생각하고, 서로 간에 많이 알수록 친근하다고 여긴다. 하지만, 코드로 구현되는 객체들 간의 관계는 서로 사적인 관계를 줄일수록 시스템은 변화(변경)에 대해 더 효과적으로 대응할 수 있다. 객체지향 기술을 이용한 시스템 개발은 이런 객체들의 모임이 느슨하게 결합하도록 만들어 주는 것이 핵심이다.

하나의 객체가 다른 객체와 관련을 맺는 형태는 "IS-A", "HAS-A", "USE-A"가 있다. "IS-A"관계는 상속(Inheritance)을 의미하며, "A라는 객체는 B라는 객체이다."라는 것을 통해서 같은 종류들을 묶어서 추상화(Abstraction)시켜주는 역할을 한다. "HAS-A"관계는 내포 관계를 의미하며, "A라는 객체는 내부에 B라는 객체를 가지고 있다."와 같은 의미이다. 따라서, "HAS-A"관계는 사물을 인식단위를 확장해준다고 볼 수 있다. 부품이나 군집의 형태를 가진 객체를 표현하는 방법을 제공하는 것이다. "USE-A"라는 관계는 "A라는 객체는 B라는 객체를 사용한다"라는 의미로, A가 자신의 책임(Responsibility)을 다하기 위해서는 B에 의존한다는 의미다. 하나의 객체가 해야 할 일을 남에게 미루는 것이 아니라, 자기가 해야 할 고유한 일은 자신이 처리하고, 그 외의 좀 더 세밀하거나 관련이 없는 일은 다른 객체에 위임(Delegation)해서 처리하겠다는 뜻이다.

객체 간의 관계를 정한다고 실행방법이 정해지는 것은 아니다. 즉, 시스템을 구성할 때 어떻게 프로세스(Process)와 쓰레드(Thread)를 만들지는 객체 관계로 만들어지는 것은 아니다. 프로세스와 쓰레드는 실제 시스템에서 스케줄링 단위(실행 순서를 정하는 단위)를 구성하는 것으로, 프로그램의 내부에서 객체 간의 관계를 이용한 실행을 담당하고 있다. 실제 시스템은 객체를 실행단위로 보는 것이 아니라, 프로세스나 쓰레드 단위로 시스템의 자원을 사용할 수 있도록 허가한다. 따라서, 우리가 구축하려는 시스템은 직접적인 관계를 맺는 것들과 시스템의 자원을 활용해서 관계를 맺는 것들로 구성된다. 이때 처리해야 할 일의 단위를 어떻게 적절하게 실행 단위로 나누고(Partition), 나누어진 각각의 실행 단위를 어떤 방법으로 연결해 줄 것인가가 구현의 중요한 이슈가 된다. 너무 잘게 나누면 많은 수의 프로세스와 쓰레드가 만들어질 것이고, 너무 크게 나누면 동시에 실행할 수 있는 부분들이 순차적으로 처리되어 제대로 된 성능을 발휘하지 못할 수도 있다.

추상화라는 것을 통해서 시스템은 각종 변경에 대해 대처한다. 추가적인 기능에 대한 대응을 프로세스나 쓰레드의 내부에서 객체 관계를 이용해 처리할 수도 있고, 더 높은 성능이나 효율을 만족시키기 위해서 확장성 있는 설계를 제공할 수도 있다. 시스템을 구성하는 원리는 역할이 잘 정의된 작은 요소(Entity)들을 묶어서 더 높은 수준의 추상화를 제공하는 것이다. 확장성 있는 설계는 특정 환경에 의존하지 않는 것을 말하며, 외부의 환경에서 분리될 수 있는 추상화된 인터페이스나 인스턴스 생성이 가능한 구조를 가지고 있다. 객체는 시스템을 구성하고 메모리 상에 공간을 차지할 수 있는 최소 단위이며, 이를 조합해서 전체 시스템에서 제공해야 할 기능을 안정적으로 구성할 수 있다.

객체(Object)를 “지향”해서 설계한다는 말을 들어본 적이 있을 것이다. 하지만, 대부분의 개발자는 객체를 "지양"해서 코딩하고 있는 것도 사실이다. 객체지향 설계는 객체에 대한 개념적인 이해가 기반이 되어야 할 수 있다. C 코딩을 주로 한 사람들에게서 보이는 프로그래밍 경향은 모든 문제를 어떤 절차로 풀어갈 것인가에 집중하는 태도다. 이런 상태에서는 정의된 절차에 나오는 행위들을 기준으로 프로그램을 블록으로 나누어 코딩하게 된다. 객체지향 설계는 사용자의 요구사항을 구현하는 “실행 주체”에 집중하는 경향을 보인다. 사용자의 요구사항을 기술하는 문장을 분석해서 필요한 객체(주체)를 뽑아내고, 이를 기반으로 객체들 간의 역할과 관계를 정의한다.

객체는 나름의 목적으로 특성과 행위를 가진다. 행위는 메쏘드(Method)의 형태로 다른 객체로부터 요청(Request, Message)이나 통신(Communication)을 담당하고, 특성은 객체의 상태를 유지하기 위해서 사용된다. “Set()/Get()”등으로 상태를 얻거나 설정하기 위해서 사용하는 메쏘드도 있으나, 직접 상태를 설정하거나 얻는 것은 객체의 독립성을 떨어뜨릴 수 있기에 제한하는 것이 좋다. 객체는 고유한 목적으로 생성되기에 내부 상태에 대한 것을 외부로 드러내게 되면, 그만큼 객체로 존재해야 할 이유가 사라진다. 객체는 독립적으로 있을 때 요청을 처리하는 주체가 되어야 하며, 처리하는 방식에 대해서는 외부로 알려주지 않아야 한다.

객체는 단 하나의 목적을 위해서 만들어져야 다루기가 쉽고 이름을 붙이기도 쉽다. 여러 가지 목적을 수행하는 것처럼 보인다면, 더 쪼개서 하나의 목적만 수행하는 객체들로 나누어야 한다. 목적 자체가 너무 추상적이라면, 하위에는 좀 더 구체적인 목적을 수행하는 객체들의 모임이 존재할 것이다. 어쨌든 추상적인 수준에서는 반드시 하나의 역할만 수행해야 한다. (이를 SRP[Single Reponsibility Principle]라고 한다.) 역할이 하나라는 것은 책임도 하나만 지고 있다는 말이 되며, 그 이상의 일을 수행하면 객체 내부가 복잡해지게 된다. 객체지향은 복잡한 문제를 잘게 쪼개진 단일 책임만을 가지는 여러 객체들의 협업으로 해결해 나가는 사고 방식이다.

여러 개의 객체가 하나의 객체에서 파생된다는 의미는 각각의 객체들을 만들기 위한 추상적인 객체를 정의하고, 파생되는 객체는 정의된 추상적인 객체를 기준으로 구체적으로 만든다는 의미다. 따라서, 추상적인 객체는 공통된 행위와 특성을 가지는 클래스로 만들 수 있다. 마찬가지로 상속도 추상화 수준을 낮추어 구체화시키는 것으로 볼 수 있다. 추상화된 클래스를 사용하는 프로그램이 의존성이 약하다는 것은 구체적인 구현에서 멀리 떨어져 있기 때문이다. 구체적일수록 변경에 대한 두려움이 많게 되며, 추상적 일수록 구현은 자신 이외의 부분에 대한 의존성을 낮출 수 있다. 전체 시스템에서 이러한 추상적인 부분들은 변경점과 같은 역할을 할 수 있으며, 이런 점들이 많아지게 되면 전체 시스템의 유연성은 향상될 것이다. 본질적으로 코딩은 미래에 있을 변경에 대한 유연함을 가져가야 한다.

"디자인 패턴(Design Pattern)"은 추상화를 통해서 코드에서 발생하는 "중복"을 제거하고, 기능을 효과적으로 추가할 수 있는 방법을 제시한다. “중복"은 코드에서 자주 발견되지만, 제거가 제대로 되지 않는다. 중복에는 코드의 중복도 있지만, 논리(Logic)의 중복도 있다. 같은 코드가 여러군데 있는 것은 변경이 발생했을 때 각각의 중복된 코드 모두를 검토해야 할 가능성이 있기에, 오류가 발생할 가능성이 높다. 중복도 앞에서 이야기한 한가지 역할에 충실한 객체를 만들다 보면 자연적으로 없어지는 부분이다. 객체지향에서는 역할이 중복된 객체를 만들 이유가 없으며, 역할의 중복이 발생했다면 뭔가 잘못된 설계를 했다는 의미가 된다. 물론 논리적인 중복이 발생할 경우에는 기계적으로 판단하기는 어려우며, 사람이 직접 코드를 이해해서 수정해 주어야 한다.

하나의 객체를 놓고 본다면, 객체 자체는 하나의 일을 처리하는 완전한 시스템과 같아야 한다. 수행하는 일의 단위가 작을수록 잘 정의된 구체적인 일을 하게 될 가능성이 높고, 시스템 내에 동일한 임무를 수행하는 객체가 있을 필요가 없게 되어 재활용할 가능성도 커진다. 객체지향의 중요한 특성 중 하나인 재활용률을 높이는 것도 이런 식으로 만든 잘 나누어진 작은 객체들이 있을 때 가능하다. 코드를 통째로 재활용하는 것은 어려우며, 사용되지 않고 낭비되는 부분이 늘어날 것이기 때문이다. 전체 시스템을 한 번에 다 보여주는 클래스 다이어그램은 이해하기 어렵고 고치기도 쉽지 않다. 큰 그림을 보여주고 다시 세부적인 그림으로 나누어서 부분으로 보여주는 것이 이해하기 쉽다. 사람의 사고는 복잡한 것을 다룰 때 큰 그림을 그리고나서 작은 그림들 하나 하나를 이해하는 경향이 있기 때문이다.

정리하면, 객체지향의 핵심은 “복잡성을 다루는 추상화의 원리”를 이해하는 것이 가장 중요하다. 추상화된 인터페이스에 의존하는 구현은 변경에 대해서 유연함을 확보할 수 있게 되며, 단일 책임을 지는 객체를 구현해서 구체화 및 재활용성도 높일 수 있다. 변경에 대해서는 열려(Open) 있으면서, 내부적인 수정이 외부에는 노출되지 않도록 닫혀(Close) 있기 때문이다. (OCP: Open Close Principle) 객체지향 언어의 문법보다는 개념을 익히는데 노력해야 하며, 문법만을 익히고 코딩하게 되면 이런 부분들을 놓치게 된다. 결과적으로는 절차지향 코딩과 유사한 객체지향 코드가 만들어질 뿐이다. 코딩은 문법이 아닌 개념이며, 언어를 지배하는 것은 어떤 문법을 잘 사용하는 것이 아니라, 그 언어가 구현한 개념을 어떻게 활용하는지에 달려있다.

# [ 코딩의 길을 묻다 ]

점심시간을 통해 중급 정도의 개발자와 이야기를 나누어 보았다. 주제는 "소프트웨어를 어떻게 만드는 것이 좋은가?"에 대한 것이었는데, 이런 저런 말을 하다 보니 결론은 관리자가 어떻게 해야 제대로 된 코드를 만들 수 있는가로 귀결되었다. 항상 바쁘다고 이야기하는 조직과 별로 바쁜 것이 없어 보이는 조직 중에서 어느 쪽이 생산성이 더 높은가를 비교하게 될 때, 무엇을 기준으로 하는 것이 좋을까? 간단한 방법은 동일한 코드를 가지고 일했을 때, 얼마나 많은 버그가 발생하는지를 보는 것이다. (절대적인 기준은 아니며, 편의로 선택한 방법이다.)

생산성이 더 높은 조직이 제대로 된 코드를 만들 확률이 상대적으로 더 높다. 생산성은 주어진 시간에 더 많은 일을 하거나(더 많은 가치를 만들어 내거나), 더 적은 버그를 만드는 것이라고 볼 수 있기 때문이다(버그는 비용을 발생시키기 때문). 따라서, 코딩 방식은 결국 생산성 문제와 직결되며, 하루에 몇 라인을 코딩 하느냐가 중요한 것이 아니라, 얼마나 정확하게 코딩 했느냐가 생산성에 영향을 주게 된다. 이를 통해서 향후에 추가적으로 들어가야 할 유지보수 비용을 얼마나 절감할 수 있느냐도 추정해 볼 수 있을 것이다. 제대로 된 코딩은 장기적으로 비용을 줄이는 효과를 가져올 것이라고 가정했기 때문이다.

제대로 된 "코딩의 길"은 결국 생산적인 프로그래머가 되어야 한다는 개발자 자신의 의지에 관련된 것이다. 따라서, 관리자는 제대로 된 개발자를 만들기 위해서 개발자의 의지를 끌어내 주는 것이 그 역할이다. 제대로 된 개발자의 의지(자세)는 결과적으로 좋은 코드 품질을 보장하게 될 것이고, 버그를 줄여 개발 기간(재작업 시간)도 단축시켜 줄 것이다. 당연히 버그가 적은 제품은 시장에서도 평가가 좋을 것이다. 선순환의 고리가 여기서 만들어지게 된다. 관리자는 선순환의 가장 앞선 출발점에 서 있으며, 개발자와 직접 마주하는 영향력이 높은 지위에 있는 사람이다. 그 사람의 기술적 역량과 판단이 제대로 서 있어야 제대로 된 출발을 할 수 있는 것이다.

개발자가 제대로 코딩을 하기 위해서 가장 먼저 해야 할 일은 모든 것이 투명한 분위기를 만드는 것이다. 솔직해야지 문제의 원인을 제대로 찾을 수 있다. 숨기는 것이 많을수록 폐쇄적인 분위기가 조성될 것이고, 그런 분위기라면 아무리 좋은 도구와 방법론을 사용한다고 해도 영리한 개발자는 빠져나갈 구멍을 찾게 된다. 코드는 공개가 되어야 하고, 그것을 통해서 모든 사람이 코드를 동등한 입장에서 평가할 수 있어야 한다. 자신이 부족한 부분은 남에게서 얻어올 수 있지만, 부족한 부분을 감춘다면 남들도 도와줄 방법을 찾을 수 없다. 투명성은 각종 툴에서 측정되는 수치를 믿는 것도 한 가지 방법이다. 예를 들어, 복잡도나 역참조, 중복된 코드를 찾는 툴들이 제시하는 값을 사용해도 될 것이다.

코딩은 결국 사람의 자세에서 결정된다. 자신이 열린 마음을 가지고 있다면, 타인의 의견이나 비판에 대해서도 열린 마음으로 들을 수 있다. 닫힌 마음을 가진 사람들은 다른 사람과 비교를 통해서 자신이 우수함을 증명하려고 하겠지만, 그런 것으로는 조직의 목표를 지속적으로 달성하지 못한다. 자신이 떳떳하다면 숨길 것이 없다. 자신이 정말 능력있고 자질이 우수한 개발자라면 남에게 숨겨야 할 코드는 없을 것이다. 관리자는 개발자들이 열린 마음으로 코드에 대한 토론을 할 수 있는 분위기를 만들어 주어야 하며, 개발자들은 자신의 직위가 아닌 동등한 입장에서 코드에 대한 대화를 나눌 수 있는 자세를 가져야 한다. 코드는 사람이 만들며 우수한 사람은 배우기 좋은 환경에서 빠르게 성장할 수 있다.

# [ 코딩의 기본 : 분리와 응집 ]

소프트웨어 개발의 핵심을 단 두 단어로 표현하자면, 분리(Decoupling)과 응집(Cohesion)이라고 할 수 있다. 이 단어들에 들어있는 의미를 잘 이해하고 못 하고의 차이는 앞으로 올 모든 사태의 근본 원인이 된다. 어떤 일을 할 때 “다섯 번의 왜(Why”)를 질문하고 대답하게 된다면, 그 속에 들어있는 진짜 원인을 찾을 수 있다고들 한다. 소프트웨어 개발 결과물에 대한 것도 마찬가지라고 볼 수 있다. 사람에 대한 교육이나 여유, 마음의 자세 등도 결과물의 품질에 영향을 줄 수 있지만, 그것을 제외한 코딩의 결과물에 대해서만 한정해서 이야기한다면 그렇다는 것이다.

코딩은 결국 “관계를 분리시키는 방식으로 개별 단위의 응집성을 높이면 된다”는 결론을 얻게 된다. 하지만, 사실 이런 결과물을 만들어내기 위해서는 많은 시행착오를 해야하고, 어느 정도 수준에 이르러서야 비로소 깨닫게 된다. 시행착오를 줄이려는 방법으로는 일단 일을 최대한 잘게 쪼개는 방법을 배워야 한다. 더는 쪼개기 힘든 수준(대략적으로 말하자면, 한 가지 일을 수행하는 함수 수준)까지 나누어서 코딩한 후, 더 쪼갤 수 있는데 합쳐져 있는 것들은 없는지 검사해야 한다. 대체로 블록으로 처리된(C나 C++에서는 "{}"들) 것들이 여러 개가 겹쳐서 있다면, 더 나눌 수 있다는 것을 의미하며, 조건문이나 반복문 속에 들어있는 지나치게 복잡한 코드도 분리의 대상이 될 수 있다.

일을 잘게 쪼개는 방법은 다양한 측면에서 세밀하게 시행되어야 한다. 객체지향 프로그래밍에서는 시스템을 구성하는 요소들이 객체로 이루어져 있다는 것을 의미하며, 객체 간의 통신으로 일을 처리한다. 따라서, 객체를 가장 작은 단위로 볼 수 있으며, 객체를 구성하는 상태와 메쏘드(Method)는 객체의 내부에서 더 쪼갤 필요가 없는지 세밀하게 검토되어야 한다. 객체들로 나눌 때는 일의 주체가 누가 되는지 확인해야 하며, 그 일이 영향을 미치는 대상이 누구인가도 봐야 한다. 즉, 주체와 목표 대상이 객체로 될 가능성이 높기 때문이다. 어떤 영향을 주는지는 객체들 간의 통신(메시지)이 담당하게 되며, 전달되는 내용이 메쏘드의 인자가 된다.

객체 간에 겹치는 역할이 존재하게 된다면, 중복될 가능성이 높기에 제거해야 한다(혹은, 객체를 더 잘게 나누어서 중복된 역할을 분리해야 한다.). 여러 가지 역할을 한 객체에 넣어주면 하나의 객체가 책임져야 할 일이 많아지게 되므로, 반드시 관련된 것들만 모아서 작은 객체로 만들어야 응집력이 높아진다. 이렇게 작고 단일한 역할을 수행하는 객체들을 하나씩 정의하며, 주어진 사용자 시나리오를 만족시켜주는 것이, 객체지향으로 시스템을 만들어가는 과정이다. 이때 한 가지 더 고려해야 할 것이 있다면, 변경을 위한 고리를 만들어주는 것과 비슷한 것들을 한 번에 묶어 재사용을 높이는 여러가지 기법들이다.

미리 미래에 사용될 것을 구현할 필요는 없다. 시스템을 확장성이 있게 만드는 것은 변화에 대한 예측이지, 예측된 변화를 구현하는 것이 아니다. 물론 어떤 변화를 미리 대응할 것인가를 정하기는 쉽지 않다. 대부분의 해당 분야 전문가라면 이미 어느 정도의 기술 흐름에 대한 생각은 가지고 있는 경우가 많으므로, 그 정도 수준에서 변화에 대한 대비만으로도 충분할 것이다. 그리고, 이미 분리와 응집을 생각하면서 설계와 구현을 진행하고 있다면, 예측하지 못하는 변경에 대해서도 큰 두려움을 가질 필요는 없다. 코딩의 기본기를 탄탄히 하는 것이 결국 시작의 작은 차이를 낳게되며, 결과물이 성숙해 갈수록 차이에 대한 보상도 함께 커질 것이다.

# [ 좋은 코드와 그렇지 않은 코드 ]

소프트웨어 개발자라면 아마도 맡은 일 때문에 보게 된 코드가 도무지 이해가 되지 않는 경우를 한 번쯤은 경험해 봤을 것이다. 실제로 어떤 버그가 있으니 고치라거나 분석해서 보고해 달라는 요청을 받게 되었을 때, 코드는 거의 이해하기 불가한 수준으로 작성되어 있어서 “맨붕” 상태가 되는 경우는 흔히 있는 일이다. 자신이 작성한 코드도 시간이 흐르면 기억이 가물가물한데, 갑작스럽게 받아든 남이 작성한 코드에 대해서 심한 불평을 늘어놓게 되는 것은 어쩔 수 없는 일이다. 하지만, 만약 이런 코드를 자신이 작성했다면 어떨까? 솔직히 자신은 절대 그렇지 않다고 떳떳할 수 있을까?

사람은 자신에게는 한없이 관대하지만 남에게는 엄격한 것이 인지상정이다. 자신이 만드는 코드는 이해하기 쉽다고 보면서 남들이 만든 코드는 눈에 잘 들어오지 않는다. 하지만 이런 것들이 지속된다면 뭔가 문제가 있는 것이다. 누구의 문제로 보이는가? 코드를 읽는 사람의 문제로 본다면 아마도 자신의 능력을 과신하거나 자신만의 새로운 세상을 만들어가는 사람일 것이다. 따라서 코드를 만드는 사람의 잘못으로 보는 것이 오히려 해결 방법을 찾을 가능성이 높다. 사용하는 언어에 대해서 충분한(?) 이해 능력이 있는 사람이 보았을 때, 쉽게 이해가 되는 코드를 만드는 것이 소프트웨어 개발자가 해야하는 일이기 때문이다. 여기서 말하는 충분한 능력이란 적어도 코딩 경험이 있는 사람을 말하며, 이해하기 위해서 관련 도메인 지식도 갖추고 있다고 가정한 것이다.

프로그램을 만드는 스타일(Style)은 코딩하는 사람의 수만큼이나 다양할 수 있지만, 읽는 사람은 그런 스타일보다 명확하게 코드를 이해를 할 수 있도록 해주길 바란다. 회사에서 만드는 대부분의 소프트웨어는 혼자서 만들지 않는다. 혼자서 만들지 않기에 더 어렵다. 소프트웨어 개발은 정신적인 활동이 대부분이며, 이렇게 만들어진 산출물을 남과 공유해야 하기 때문이다. 말이나 문서로 생각을 정확히 전달하는 것이 어렵다는 것은 이미 다 잘 알 것이다. 따라서 프로젝트 원활하게 수행하기 위해서 소프트웨어를 개발하는 사람들이 사용하는 “공통된 언어”는 중요한 의사소통 수단이다. 문서화를 시키면 충분할 것으로 생각하지만 실제로 그렇지 않다. 문서는 현실(코드)을 반영하는데 시간적인 차이(Gap)가 있다. 즉, 코드가 변경되고 나서 문서에 반영되기 까지는 시간 차이가 있을 수밖에 없다. 가장 좋은 소프트웨어 개발자 간의 의사소통 수단은 "이해하기 쉬운 동작하는 코드"다.

코드를 이해하기 쉽도록 만들기 위해서는 크게 두 가지를 고려해야 한다. "이름 짓기(Naming)"에 대해서 신경써야 하며, "코드의 길이”를 짭게 만들어야 한다. 이런 부분들이 잘 지켜진 코드를 현장에서 직접 코딩하는 사람들이 실제 자신의 코드에 반영해 주어야 한다. 혼자만의 코드를 짜지 않는다면 타인의 검토(Review)를 받아야 좋은 이름과 간결한 코드가 나올 수 있다. 검토 과정을 통해서 이해하기 어려운 부분에 대한 수정 요청을 받을 것이기 때문이다. 따라서 코드 리뷰의 첫 번째 목표는 당연히 "가독성"이 되어야 한다. 가독성이 높은 코드는 앞으로 그 코드에 대해서 디버깅과 유지보수를 해야 할 사람들에게는 축복처럼 받아들여질 것이다. 개발의 초기부터 모든 코드에 대해서 일관되게 가독성을 유지하려는 노력을 기울인다면 시간이 흘러갈수록 그 효과는 크지게 될 것이다.

남이 개발한 코드를 보는 것은 솔직히 끔찍한 경험이다. 특히, 제대로 된 도메인(Domain) 경험이 없는 상태로 초기에 받아보게 되는 코드는 삶의 기쁨을 잊어버리기에 충분할 만큼 싫은 일이다. 코드가 어디서부터 시작하는지를 찾는 것부터 시작해서, 호출되는 함수를 따라가면서 각종 자료구조와 상수 값들을 일일이 확인하는 일은 상당한 시간의 정신 집중을 요구한다. 힘든 것은 그렇게 오랫동안 집중할 수 있는 시간도 없거니와, 왜 그렇게 작성되었는지 이유를 알려줄 수 있는 사람도 없다는 점이다. 이해는 잠시 제처두더라도 누가 왜 그렇게 작성했는지 알 수 없는 코드를 읽는 것 자체가 부담스러울 수밖에 없는 것이다.

끔찍한 경험에서 벗어나는 길은 다른 일로 자원을 하거나 회사를 그만두는 것뿐이다. 좀 더 적극적으로 “험하게 작성된 코드”를 이해하기 위해선 상당한 인내력이 필요하다. 예전의 경험으로 봤을 때, 그렇게 작성된 코드 대부분은 문서화도 되어있지 않을 가능성이 높다. 코드 내의 주석(Comment)도 기대할 수 없고, 주석이 있다고 하더라도 도움이 안 되는 "누가 몇 월 몇 일에 작성 했음" 정도가 남아 있을 것이다. 더 안 좋은 것은 그런 코드 내에 주석으로 처리된 코드들도 듬성듬성 들어있거나, "#if 0 ... #elseif ... #endif"와 같이 사용되지 않는 코드들도 많이 남아 있는 경우다. 코드의 길이는 심지어 하나의 함수가1,000라인을 넘는 경우도 있고, 중첩된 "if()"문들과 "switch()", “for()”, “while()”등으로 인해 코드가 어디서 논리적으로 끊어지는지 알기도 힘들 수 있다.

그래도 우리가 그런 코드를 읽을 수 있는 수준이기에 "암호문"과는 다르게 보일지 모르겠지만, 사실 경우에 따라 암호문을 해독하는 것이 더 간단할 수도 있다. 왜냐하면, 최소한 암호는 어떤 규칙 하에 만들기 때문이다. 위와 같이 작성된 코드는 사실 "무규칙"이나 다름없다. 규칙이라고 한다면 “자신이 만들고 싶은데로 코드를 만드는 것” 이외에는 없다. 왜 그렇게 작성했는지 알려달라는 질문에 돌아오는 답은 "코드에 녹아 있습니다."라는 말뿐이다. 당장 달려가서 뒤통수라도 한 대 때리고 싶다는 느낌을 받는 것은 나만의 감정이 아닐 것이다. 마주치는 코드 중에 이렇게 코딩된 것들이 놀라울 정도로 많다는 현실이 개탄스러울 뿐이다.

언젠가 어떤 교수님이 우리나라 IT분야의 소프트웨어 개발자들의 역량이 외국과 비교하면 떨어지지 않는다고 이야기하는 것을 들었다. 물론 개개인이 충분한 능력을 갖추고 있다는 것에는 동의한다. 엉망으로 작성된 코드를 보고 꾸준히 디버깅과 기능 추가를 해서 팔 수 있는 물건으로 만들고 있다는 사실 때문이다. 인내력도 대단해서 그렇게 작성된 코드라도 야근과 주말 특근을 이용해서 꾸준히 개선해 나가고 있기 때문이다. 하지만, 만약 학교에서 알고리즘이나 이론만을 가르치지 않고, 조금 더 나아가 소프트웨어 공학, 리팩토링, 디자인 패턴, 아키텍처, 객체지향 분석 설계, 객체지향 패러다임에 대한 개념, 테스트 주도 개발, 애자일, 함수형 언어 등등을 학생들이 실제 코드와 같이 익힐 수 있다면, 10년 후에 유지보수를 하는 개발자들은 정말 좋은 환경에서 개발에 몰두할 수 있을 것이다.

사실 야근이나 주말 특근의 기본적인 원인은 "조급한 마음"과 "말도 안 되는 일정"에 기인한다. 조급함은 코딩을 빨리 끝내라고 소프트웨어 개발자를 몰아대고, 당겨진 일정은 기능 구현만 완료되면 바로 다른 기능을 구현하는 일로 사람들을 투입하게 만든다. 이런 상황에서 제대로 된 “장인정신”으로 코드를 지속해서 개선하는 것은 불가능하다. 좋은 문서를 기대하는 것은 "우물에서 숭늉을 찾는 것"과 다르지 않다. 소프트웨어 개발은 “빨리빨리”가 아니라, 원칙과 그 원칙을 끝없이 반복해서 실행하고 개선하는 과정이다. 무원칙이 원칙이라고 한다면 우리가 만드는 소프트웨어 제품의 품질과 유지보수는 포기해야 한다. 지속적인 요구사항 변경과 기능추가, 끝없는 성능 향상에 대한 요청이 들어오는 상황에서 제대로 만들어지지 않은 제품은 생명력이 길지 않다. 다시금 비용을 들여 대규모의 수정을 해 나가지 못한다면, 결국 회사는 문을 닫고 부동산 업이나 해야 할 것이다.

# [ 플랫폼 개발이 실패하는 이유 ]

플랫폼은 소프트웨어 재사용을 위한 개발 결과물이다. 하지만, 실제로 개발되는 플랫폼은 제대로 역할을 하지 못하거나, 상품화와 격리된 상황에서 제품에 적용되지 못하고 사라지고마는 경우가 많다. 안타까운 일이지만 플랫폼을 개발해야 할 이유를 모르거나, 자신의 명리만을 찾는 사람이 플랫폼의 주도권을 가지면, 플랫폼의 성공은 이미 거의 물 건너 갔다고 봐야 할 것이다. 그런 사람들은 항상 저울 위에 물건을 올려놓고 지금 당장 어떤 것이 더 값이 나가는지 재어보기만 할 뿐이다. 현재의 가치만을 보고 미래에 어떤 가치가 생겨날지는 관심이 없다. 자신의 계약 연장에만 매달리지 회사의 미래는 안중에 없기 때문이다.

플랫폼을 제대로 개발하기 위해서는 개발하는 방법을 알아야 하고, 개발되는 플랫폼이 적용될 제품에 대해서도 알고 있어야 한다. 그 만큼의 역량이 있는 사람을 찾기는 쉽지 않기에 반드시 현재 상품화 개발팀에서 역량 있는 인력들을 플랫폼 개발에 차출할 수 있어야 한다. 하지만, 아주 당연하게도 이런 일은 "의지"가 없는 관리자에게서는 절대 일어나지 않는다. 그냥 개발자의 머리수만 헤아려 몇 명이면 되냐는 멍청한 질문만 한다. 잘 알려진 이야기지만 한 사람의 개발 역량은 대략 10배가 차이가 날 수도 있다. 실제로 예전의 경험을 보면 플랫폼이 중요하다고 이야기들은 하지만, 대부분 각 팀에서 차출되는 인력들은 이미 그 팀에서 퇴출 대상으로 낙인 찍혔거나, 신입사원으로 배정받은 인력들이 대부분이었다. 직접적으로 플랫폼 개발에 투입할 인력을 고를 수 있으면 좋지만 절대 그런 일은 용납되지 않는다. 사람들은 자신이 지금 하고 있는 일이 가장 중요하며, 불확실한 미래에 투자하는 것을 반기지 않는다.

의지가 없는 관리자를 만나서 만들 수 있는 플랫폼은 없다. 결국 만들어봐야 그냥 프로토타입 수준의 제품에 적용해보고, 이런저런 핑계로 시간을 끌 것이 분명하다. 가장 많은 핑계는 "라인업 모델이 많아서, 지금 당장 적용하기 어려우니 이 모델 저 모델에 시험삼아 한 번 적용해봐."라는 말이다. 이미 상품화되어 나가고 있는 모델에 적용하는 것은 어렵지 않겠지만, 그런 모델을 운용 중에 변경(Running Change)하는 것이 얼마나 위험한지는 보지 않아도 뻔하다. 그냥 시간 끌기에 불과할 뿐이다. 이런 일은 해서는 안 되는 일이다. 아무런 위험도 없이 새로운 플랫폼을 도입하는 것이 가능할까? 위험이라고는 조금도 감내하려고 하지 않는 관리자는 이미 리더가 아니다. 그는 그냥 자신의 생존지침(Survival Guide)대로 아래 사람들을 도구로 사용하는 사람일 뿐이다.

플랫폼의 실질적인 적은 기존 상품 개발팀의 개발자들이다. 그들의 논리는 자신이 개발하지 않은 코드는 믿지 못하겠다는 것이다. 자신이 만들어 놓은 "좋은 구조(정말 좋은 구조라면 공개적으로 토론해보고 남들의 말에도 귀를 기울여야 하지 않을까?)"를 반드시 플랫폼에 포함해야 한다고 이야기할 것이다. 하지만, 아이러니 하게도 그렇게 만들어 놓은 구조 때문에 새롭게 플랫폼을 만들고 있다는 것은 왜 모르는 것일까? 상품 개발팀에서 핵심인 인력이 반드시 플랫폼 개발팀과 함께 일해야만, 어떤 것이 좋은 구조인지 이해하게 되고 실질적인 개선이 이루어질 수 있다. 상품 개발팀의 협조없이 개발된 플랫폼은 나중에 그것을 사용하게끔 설득하는 것은 거의 불가능에 가까운 일이다. 누군가의 의지도 없다면 이미 플랫폼은 시작도 하기전에 실패할 과제가 될 뿐이다.

완벽한 소프트웨어가 있다고 믿고 있는 사람이 요즘도 있을까? 아마 없을 것이다. 마찬가지로 완벽한 플랫폼도 존재하지 않는다. 하지만 제품에 적용하려는 단계에 도달하면 항상 사람들은 플랫폼의 완벽한 검증이 필요하며, 한 번에 모든 제품 라인업(Lineup)에 적용할 수 있게 되어 있는지를 묻는다. 어리석은 질문일 뿐이다. 완벽한 소프트웨어를 기대한다면 처음 개발된 플랫폼에서 찾을 것이 아니라, 이전에 개발된 제품에서 찾는 편이 빠를 것이다. 이미 자신들도 그렇게 하지 못하고 있어서 플랫폼을 개발하자는 요구가 나온 것이다. 하나의 제품에 새로 개발된 플랫폼을 적용하기 시작하는 것은 "작은 첫걸음"일뿐이다. 그리고, 그것을 확장해 나가는 것이 "대단한 도약"이라는 것을 시간이 흐르면 자연스럽게 알게 될 것이다.

과거에 출시한 모든 제품을 하나의 플랫폼으로 다 갈아타기 위해서는, 갈아탈 제품들에 대한 신규 플랫폼 적용을 테스트를 해야 한다. 한 두 제품이라면 모르겠지만 수십 종에 이르는 제품에 다 적용해서 검증한다는 것은 차라리 플랫폼을 사용하지 말자고 이야기하는 것과 같다. 신규 플랫폼은 신규 제품에 가장 먼저 적용하는 것이 옳다. 또한 모든 소프트웨어가 완벽하지 않듯이, 신규로 만든 플랫폼이라고 완벽할 수는 없다. 중요한 것은 상품화 적용을 통해서 품질을 지속해서 높이고, 내부구조를 다듬어야 한다는 것이다. 플랫폼도 소프트웨어이기에 사용자의 빠른 피드백이 가장 확실한 검증 방법이 된다는 것이다.

소프트웨어 플랫폼을 만드는데 필요한 요구사항은 뭘까? 사실 일반적인 제품에 대한 요구사항과 소프트웨어 플랫폼에 대한 요구사항은 조금 다르다. 기본적인 기능이야 제품이든 제품에 적용할 토대가 되는 소프트웨어 플랫폼이든 그렇게 차이는 나지 않을 것이다. 하지만, 소프트웨어 플랫폼은 회사 내부의 생산성을 향상해야 한다는 요구를 받아들여서 만들어지기에, 요구사항을 전달하는 사람이 상품 개발 조직에서 오는 경우가 많다. 그 외에 높으신 분들의 생각들이 더해져 기본적인 "비기능적 요구사항"을 구성하게 된다. 문제는 요구사항에 대한 이해를 잘못해서, 소프트웨어 플랫폼의 아키텍처까지도 상품개발팀이나 이해 관계자의 마음대로 정하려고 하는 지나친 간섭에 있다. 응용 프로그램 개발자가 소프트웨어 플랫폼의 내부구조를 이렇게 저렇게 만들어 달라고 이야기하는 것이 플랫폼 개발에 과연 무슨 도움을 줄 수 있을까? 내부구조를 모르고도 충분히 구현할 수 있고, 응용 프로그램의 목적을 수행할 수 있어야 하는 것이 더 옳바른 것이 아닐까?

사실 소프트웨어 플랫폼을 만들고자 하는 가장 기본적인 욕구는 기존에 존재하는 소프트웨어의 의존성이 높고 복잡한 코드를 개선해서 확장과 유지보수성을 높이고자 하는 것이다. 즉, 구조적인 개선을 통한 비용의 개선을 하고자 하는데, 기존 개발자가 나서서 구조에 대한 변경을 막고 나선다면 소프트웨어 플랫폼을 만드는 게 무슨 의미가 있겠는가? 오히려 기존 개발자가 요구해야 할 부분은 제공되는 인터페이스에 대한 일관성과 그것을 잘 정의하기 위한 협업이 아닐까? 소프트웨어 모듈간의 대화는 인터페이스라는 계약(Contract)으로 만들어지는 것이지, 내부의 모든 것을 이해해야 만들 수 있는 것이 아니다. 계약을 정하고 그것을 성실히 수행한다면, 서로 불만을 가질 이유가 없는 것이다. 예를 들어, C언어로 코딩할 때 소프트웨어 개발자가 표준 C 라이브러리의 내부를 모두 다 이해하거나, 운영체제의 커널 내부구조를 자신이 개발하려는 응용프로그램에 적합하도록 변경을 요구하는 경우가 있을까? 아마도 간단히 자신이 사용하는 인터페이스가 무엇이고, 어떤 효과를 거둘 수 있느냐 정도만 파악할 것이다. 다른 요구사항이 발견되거나 변경이 발생할 경우 전혀 분리되지 않은 상태로는 전체 시스템이 영향을 받을 수밖에 없는 것이다. 실수는 변경의 영향도가 커질수록 더 많이 발생할 수밖에 없다.

이런 식의 개입이 늘어나게 되면 소프트웨어 플랫폼의 실패는 과제를 시작하기 전에 이미 정해진 경로였음을 개발의 후반이 되면 깨닫게 될 것이다. 최고 경영자만 좋아하는 소프트웨어 플랫폼은 실제 제품 적용으로 이어지지 않으면 아무런 가치가 없다. 오히려 그것을 개발하기 위해서 투자된 비용만 낭비된 것이다. 진정한 소프트웨어 플랫폼은 플랫폼 개발자와 상품화 개발자 사이의 틈(Gap)을 다 없애고 나서야 가능하다. 그 전에는 자신의 성과를 가져가기에 바쁘고, 개선이라는 것은 말 그대로 미래에 있을 어떤 좋은 것이라는 일종의 상상에 지나지 않는다. 말로만 하는 개선에 우린 이미 아주 익숙하지 않은가? 그 많은 보고서와 회의를 통해서 나오는 모든 개선은 이미 업무에 반영되었거나, 혹은 반영하는 척 수치로 보고만 했을 뿐이라는 것을 다 알지 않는가? 이젠 실제로 뭔가를 바꿔야만 한다. 그렇게 하지 않는다면, 우리를 보고 미래를 상상해야 할 사람들에게서 희망이라는 것을 빼앗아가는 일 밖에 하고 있지 않은 것이다.

일에는 항상 우선순위라는 것이 따른다. 대부분의 경우 가장 우선순위가 높은 일은 회사의 최고 경영자가 내리는 명령(지시사항)일 것이다. 그리고, 자신이 속한 사업부의 사업부장, 연구개발 센터의 최고 임원, 팀장, 그리고, 마지막으로 자신의 직접적인 상사가 내리는 일이 그 우선순위로 여겨진다. 여기에는 고객이 원하는 일에 대한 우선순위는 없거나, 가장 낮은 수준일 것이 분명하다. 당면한 문제를 해결하고 직접적인 고과권자의 업무 할당을 무시할 수 있는 용기가 없기 때문이다. 솔직히 조직 구조상 멀리 있는 사람이 내리는 명령보다 바로 옆에 있는 고과권자가 내리는 명령이 우선순위가 높다. 회사의 정책보다는 팀장의 주먹이 더 가깝기 때문이다.

과제에 이상 유무를 판단할 수 있는 사람은 그 과제를 직접 진행하는 사람일 것이다. 나머지 사람들은 과제에서 한 발을 빼고, 마치 자신은 아무런 관련이 없는 듯이 비판만 한다. 비판까지는 좋지만 때로 숙제도 낸다. 무엇인가 잘못되고 있다는 보고를 하면 관련된 모든 사람을 갑작스럽게 호출해서 이런저런 명령을 내린 후, 즉시 결과를 가져오기를 기다린다. 하지만, 그런 결과가 바로 나올 수 있는 일이면 이미 정리되어 있거나 시스템으로 이미 관리되고 있을 것이 분명하다. 안타깝게도 그런 것을 기대할 정도의 수준이 못 되는 명령권자는 자신이 무슨 명령을 했고, 그 결과로 업무에 대한 지장이 얼마나 생길지를 예상하지 못한다. 자신이 중요하게 생각하는 것이 최고의 우선 순위로 받아들여지기를 기대할 뿐이다. 하지만, 누군가는 과제를 계속 진행하고 있어야 일이 마무리된다. 숙제만 하고 있으면 정작 중요한 일은 뒤로 미뤄지는 것이 당연하다.

숙제를 열심히 한다고 과제를 성공시키는 것은 아니다. 숙제를 안 한다고 해서 과제가 실패하는 것도 아니다. "실행에 집중하라"라는 말이 있듯이, 우리에겐 "소를 키울 사람"도 필요하다. 숙제는 아무런 우선순위가 없이 급하게 처리되고, 그런 바쁜 일들이 지속하면 숙제의 내용도 점차 나빠지게 된다. 결국 그로 인해서 더 많은 숙제를 해야 하는 빈곤의 악순환과 같은 상황이 매일 매일 반복적으로 일어난다. 사실 숙제를 안 한다고 해서 생명의 지장이 생기거나, 그로 인해서 물리적으로 상처를 받는 것은 아니다. 하지만, 그런 상황이 반복적으로 일어나면 아무리 뛰어난 인력이라고 해도 점차 자신이 뭐 하고 있는지도 모르는 바보가 되어간다는 것을 느끼게 된다.

플랫폼에 중요한 기능(Feature)을 넣는 작업은 도메인 지식이 풍부한 사람을 요구한다. 즉, 해당 도메인의 비즈니스 로직(Business Logic)을 잘 아는 사람이 필요하다. 하지만, 그런 사람을 플랫폼 개발팀이 요청하면, 곧장 듣는 말이 해당 인력의 업무중 50%를 지원해 주라는 답을 듣게된다. 그 사람이 가진 역량의 50%를 이렇게 나누어서 공유(Sharing)하는 것이 가능한 일인가? 그렇게 여러 가지 일을 동시에 처리하라고 이야기하면, 당사자의 입장에서는 두 가지 일이 다 중요하지 않은 일이 되고 만다. 어떤 일에도 집중해서 자신의 역량을 충분히 발휘할 수 있다고 생각하지 않게 된다. 또한, 그 사람이 속한 부서에서의 일이 우선이지 절대 지원해 주어야 할 일에 대해서 높은 우선순위를 할당하지 않는다. 따라서 그렇게 만든 “가상의 조직”으로 플랫폼을 만들라고 하는 것 자체가 웃긴 일이 되어버리고 만다. 항상 사람이 모자란다고만 이야기하지 근본 이유를 밝히고 해결하고자 하는 노력은 우선순위가 높아진 일(?) 때문에 더 멀리 밀려 나고만다.

플랫폼이 실패하는 이유는 다양하지만 이것도 결국은 사람의 문제다. 플랫폼은 기술이 아니라 사람이 만들며, 플랫폼이 해결해야 할 문제도 사람과 조직의 문제다. 못만드는 것이 아니라 안만드는 것뿐이다. 서로 이해 관계가 다르고, 주장하는 바가 다르고, 사람이 다르며, 목표도 다르다. 한 방향으로 가도 제대로 못하는 경우가 많은데, 이런 식으로 나누어진 방향으로 제각각 걸어가는 상황이라면 실패는 당연하게 받아들여야 한다. 우선순위가 높은 일은 자신의 지적인 호기심을 채우거나 남을 괴롭히는 습관적인 숙제가 아니다. 가치를 만들고 그 가치를 공유할 수 있는 시스템을 정의하지 못한다면, 플랫폼은 언제나 실패한다. 실패가 받아들여질 수 없는 상황이라면 플랫폼이라고 불리는 여러가지 것들을 제 각각 또 만들 것이다. 자신이 하는 일이 플랫폼이라고 우기는 일도 있을 것이고, 그것만이 해결책이라고도 이야기할 것이다. 맞는 말이다. 적어도 플랫폼이 어떠해야 한다는 정의가 없다면 모든 만들어진 제품이 다 플랫폼이다. 실패할 것이라면 만들지 않는 것이 최선의 선택이 될 수밖에 없는 것이다.

# [ 단위 테스트에 대해서 ]

소프트웨어 개발자라고 한다면, 코딩에서 가장 어려운 부분이 단위 테스트(Unit Test)라는 것을 잘 알고 있을 것이다. 가장 친근해야 하는 실무 개발자들이 대부분 단위 테스트에 익숙하지 않은 것이 사실이다. 문제는 단위 테스트라는 것이 체계적으로 배우기가 어려우며, 코딩과 같이 진행되어야 제대로 할 수 있다는 점이다. 단위 테스트는 이론적인 부분보다는 응용(실용)에 가까운 부분이며, 학교에서 논하는 과학(Science)이라기보다는 공학(Engineering)의 실무 적용(실천 방법)에 가깝다. 하지만 생산성의 가장 큰 부분은 개발의 후반부에 집중되는 테스트 시간을 어떻게 줄이고, 디버깅 된 코드에 대해서 회기 테스트(Regression Test)를 얼마나 효율적으로 실행할 수 있는가에 달려 있다. 이를 위해서는 자동화된 단위 테스트는 필수 과정으로 받아들여져야 하지만 그렇지 못한 것이 현실이다.

단위 테스트는 호출(Call)로 실행할 수 있는 가장 작은 단위에서 시작한다. 따라서 함수(혹은 메쏘드, 멤버 함수)가 테스트의 단위가 된다. 시스템 통합 테스트는 시스템을 설계하는 단계에서 반영되는 것이 좋다. 이때의 테스트는 시스템 수준에서 검증할 수 있는 유용한 구조나 방법을 미리 정의하고 적용하는 것을 말한다. 하지만, 단위 테스트는 설계된 것을 실제로 구현할 때, 어떤 방식으로 해야지 쉬운지를 알고 코딩을 해야 한다. 단위 테스트의 대상이 되는 함수를 어떻게 만들어야 쉽게 테스트를 할 수 있는지를 아는 것이 중요하다. 함수는 특정한 한 가지 일을 수행하기 위해서 만들어지는 것으로, 인수(Argument)를 호출에 대한 입력으로 전달해서 그 결과를 받아본다. 따라서 함수가 입력 값에 대해서 제대로 된 결과를 돌려주는지 외부에서 단순한 비교를 통해서 검증하게 된다. 함수의 내부가 정확히 동작하는지를 확인하기 위해서는 별도의 방법을 동원해서 처리할 수도 있다.

함수의 복귀값이 없으면 테스트하려는 함수가 제대로 동작하는지를 외부에서 알기가 어렵다. 따라서, 모든 함수는 복귀값을 가지는 것이 좋다. 함수가 파라미터를 여러 가지를 가지면, 테스트해야 할 경우의 수가 많아진다. 테스트할 경우의 수가 적을수록 테스트가 쉽기에, 함수는 될 수 있으면 적은 수의 파라미터를 가지도록 정의하는 것이 좋다. 물론 하나도 없다면 테스트 케이스도 한 가지 밖에 없기에 제일 좋다. 물론 이때도 함수의 내부가 외부에 의존적인 경우에는 선행 조건을 여러 번 설정해야 하기 때문에 테스트 케이스는 여러 개가 될 수도 있다. 함수가 해야 할 일도 파라미터의 수가 적을수록 작아지는 경향이 있기에, 파라미터의 수를 줄이면 함수의 크기도 작아지는 경향이 있다. 테스트할 범위가 작아지기에 테스트 케이스가 확인하는 코드의 범위(Test Coverage)에도 좋은 영향을 준다. 하지만, 한 가지 단점은 길이가 짧은 함수들이 많아져서 호출 경로(Call Path)가 길어질 수 있다는 것이다. 사실 이것은 별로 큰 문제가 되지 않는다. C/C++에서 인라인(inline) 함수와 같은 것을 활용하면, 컴파일러에서 충분히 최적화를 잘 지원해 줄 것이기 때문이다. 그리고, 짧은 함수는 대부분 이해하기 쉽기에 변경과 유지 보수 측면에서도 긍정적인 영향을 준다.

대부분의 경우 C/C++에서는 50라인보다 작은 함수를 만들기를 권장하고 있지만, 논리적으로 한 가지 일만 하는 함수를 만들면 된다. (될 수 있으면 한 가지 일만 하는 짧은 함수를 만들어야 할 것이다) TDD(Test Driven Development: 테스트 주도 개발)가 아니더라도 하나의 함수를 짧게 만들고, 반드시 자동으로 테스트할 수 있는 테스트 스크립트(Test Script)를 만들어야 한다. 최소한 각각의 함수에 대해서 한 개 이상의 테스트 스크립트가 존재해야 한다. 테스트를 잘 했는지 못했는지는 테스트 범위(Test Coverage)의 측정을 통해서 알 수 있다. 함수, 분기, 명령문 단위로 측정해서 제대로 실행되지 않은 부분이 있다면, 테스트 케이스를 늘려서 보완하면 된다. 100% 모든 경로를 다 실행하는 테스트 케이스를 찾아내기는 어려우니, 테스트 케이스를 보완해서 여러 번 실행해야 한다. 간단한 테스트에 대한 이론적인 기반은 소프트웨어 개발자도 알고 있는 것이 이 때는 도움이 될 수 있다. 자신이 만든 코드가 제대로 동작하는지 확인하는 것은 소프트웨어 개발자의 기본적인 의무이며, 반드시 일로서 받아들여야만 지속적인 실천이 가능하다.

코드가 짧을수록 테스트하기 쉽다는 것은 어쩌면 당연한 이야기로 들릴 수 있다. 하지만, 사실 이렇게 당연한 이야기를 하는 것이 실무에서는 잘 받아들여지지 않는다. 그렇게 하도록 훈련을 받은 기억도 없으며, 선배들과 비슷한 방법으로 오늘도 신참 개발자들은 길고 이해하기 힘든 함수들을 만들어내고 있다. 결과적으로 가독성도 떨어지고, 시간이 조금 흐른 뒤에는 고치기도 어려운 상태가 된다. 테스트 케이스 조차도 없는 이른바 "레거시(Legacy Code)"가 되어버리고 만다. 우리는 배운데로 코딩한다. 배우지 못한 것은 따라하면서 코딩하게 되고, 잘못된 코딩이이라고 느끼지 않고 받아들이게 되는 것이다. 이미 만들어진 코드는 절대 고쳐서는 안되는 불문률이 되어버리고, 그것을 고칠 수 있는 사람도 작성자 이외에는 아무도 없다. 결국 남겨진 유산은 버려지지 못하고 지속적으로 현재와 미래에 영향을 주게 된다.

실무에서 새롭게 코딩하는 부분이 10% 정도라면, 그 외의 활동은 코드를 읽고 복사해서 붙여넣는 과정의 반복이다. 그렇게 복사된 코드를 반복적으로 디버깅하는 것이 일과의 대부분이다. 사실 코드를 만들고, 그 코드를 테스트하는 코드를 만드는 것은 어려운 일이 아니다. 나중에 디버깅 해야하는 시간을 생각한다면, 아주 작은 시간이 소모될 뿐이다. 예를 들어, 하루에 100라인의 코드를 만든다면, 그것을 테스트하기 위해서 200라인 정도의 코드를 새로 작성한다고 해서 특별히 오랜 시간이 소모되는 것은 아니다. 검증된 100라인의 코드가 검증되지 않은 200라인의 코드보다 디버깅 시간을 단축시켜줄 수 있기 때문이다. 그리고, 그렇게 만들어진 자동화된 테스트 케이스는 필요에 따라 언제든 다시 실행할 수 있다.

짧은 코드를 완성한 후에는 테스트 코드를 만들어 반드시 제대로 동작하는지 확인하는 것이 필요하다. 짧은 코드는 컴파일 시간도 짧고 실행 시간도 짧아야 한다. 실행 시간이 길다면 테스트를 이용해서 코드 동작에 대한 피드백을 받는 시간이 길어진다. 피드백을 받는 시간이 길어지면 개발자들이 제대로 활용하지 않는 시간이 길어지게 되며, 점차 테스트 코드의 실행이 느려질 가능성도 있다. 예를 들어, 하나의 테스트 케이스를 실행하는데 10초가 걸린다면, 10개만 해도 100초가 걸리게 된다. 이런 식으로는 만들어진 전체 테스트 케이스를 자주 실행해서 확인하기 어려워질 것이다. 특히, 특정 조건이나 시간이 되어야 실행되는 함수의 경우에는 인위적인 조작을 통해서 조건을 빨리 만족시켜줄 필요가 있을 것이다. 이를 위해서 필요한 조치도 테스트 코드의 일부로 작성되어야 할 것이다.

하나의 함수가 여러 함수나 모듈에 의존하게 되면, 테스트 케이스를 만드는 것을 어렵게 한다. 해당 함수가 여러 모듈이나 함수를 호출하는 경우로 모든 의존적인 부분을 대체할 더미(Dummy: 가짜 함수)들을 만들어 주어야 하기 때문이다. 따라서 애초부터 소프트웨어를 설계할 때 이런 의존성이 최소화가 될 수 있게 하는 것이 필요하다. 만약 의존성이 특정 모듈의 내부 상태에 기반을 둔다면, 테스트는 더 어려워질 것이다. 이때는 의존성을 줄이기도 해야 하지만 의존성이 내부적인 상태에 기반하지 않도록 만들어야 한다. 즉, 단위 테스트를 쉽게 하기 위해서는 좋은 소프트웨어가 가져야 할 의존성 분리(Decoupling)를 반드시 지키는 설계와 구현이 필요하다는 것을 알 수 있다. 또한 하나의 함수 내부에서 생성되는 중요 구성요소(Entity: 변수나 자료구조, 메모리 공간 등)들을 가질 경우 함수의 내부에서 의존성을 가지는 것이 되기에, 될 수 있으면 함수의 외부에서 제공하는 형태로 변경하는 것이 좋다.

위와 같이 외부에서 의존성을 함수로 주입하게 되면, 의존성 주입(Dependency Injection)이라는 용어를 사용한다. 즉, 함수가 내부에서 새로운 타입의 자료구조를 생성하고 그것에 의존적인 일을 수행하도록 하지 않고, 생성할 자료구조를 외부에서 만들어 인수로 함수로 넘기는 방식이다. 이런 방식을 사용하게 되면 만들어진 자료구조가 정확히 사용되고 있는지를 외부에서 테스트 스크립트가 관찰할 수 있으며, 테스트하려는 함수 자체의 구현도 짧아지게 되는 효과가 있다. 또한 만들어진 자료구조를 생성하고 초기화하는 등의 작업도 외부에서 진행할 수 있으며, 특정 상황을 재현하기도 쉬워진다. 될 수 있으면 함수 내에서 새로운 타입의 자료구조를 생성하지 말고, 외부에서 전달하는 방식으로 코딩하는 것이 좋다. 이 같은 경우에는 함수 자체의 호출의 결과를 확인(Assert)하는 것과 생성된 자료구조의 값을 확인하는 것, 실행되는 순서(혹은, 시나리오)의 정합성을 판단할 수 있는 방법을 쉽게 테스트 스크립트로 구현할 수 있다.

하드웨어 추상화 계층(HAL: Hardware Abstraction Layer)은 하드웨어와 잘 정의된 인터페이스를 담당한다. Linux나 Windows와 같은 운영체제는 이미 잘 정의된 하드웨어와의 인터페이스를 응용프로그램에 제공하고 있다. 따라서, 이런 경우에는 단순히 운영체제 수준에서 제공되는 인터페이스를 랩퍼(Wrapper)를 통해서 감싸는 것으로 하드웨어 와의 통신에 관련된 모듈 들을 테스트할 수 있는 테스트 코드를 구축할 수 있다. 실시간 운영체제(RTOS: Real-Time OS)나 혹은 운영체제가 없는 마이크로 콘트롤러(Micro-Controller)를 가정한다면, HAL을 정의해 두는 것이 필요하다. 즉, 하드웨어와 제어하는 소프트웨어를 분리할 수 있는 계층을 미리 마련해 두는 것이다. HAL은 단위 테스트를 위한 것과 실제 하드웨어와의 통신을 담당하는 두 가지를 다 갖춰야 할 것이다. 조금은 부담스러울 수도 있으나, 하드웨어 없이 소프트웨어의 검증을 할 수 있다는 점에서 충분한 도움을 얻을 수 있을 것이다.

HAL 계층의 인터페이스는 “Open(), Close(), Read(), Write(), IOControl()”등 Unix나 Linux에서 일반적으로 간단히 가져올 수 있는 것들이다. 좀 더 낮은 수준의 HAL을 구현하고자 한다면, 직접적으로 하드웨어 주소를 접근하는 부분에 대해서 인터페이스를 정의해야 하지만, 일반적으로 각각의 하드웨어는 제어하는 방법이나 순서가 다른 것이 일반적이다. 이러한 인터페이스들은 하드웨어 자체의 구조를 모르고, 단순히 응용 프로그램에서 필요한 일을 할 수 있도록 도와주는 역할만 수행한다. 따라서, HAL 계층을 인위적으로 만들어 하드웨어를 접근하는 함수를 테스트하기 위해서는 하드웨어 입출력에 대한 값을 수집해야 한다. 혹은 제대로 된 호출의 결과를 정의해 줄 수 있어야 한다. 이렇게 만들어진 HAL은 나중에 응용프로그램을 위해서 만족시켜야 할 스펙과 같은 기능도 수행하게 되어, 새로운 하드웨어로 포팅할 때에도 유용하게 사용될 수 있을 것이다. “IOControl()”의 경우는 하드웨어에 대한 특별한 명령(Command)처리를 담당하기 때문에 일반화시키기 위해서는 좀 더 노력을 들여야 할 수도 있다.

하드웨어에 대한 랩퍼(Wrapper)를 만들어서 사용하게 되면 평소에는 잘 발생하지 않는 오류 상황을 인위적으로 만들어 낼 수 있다. 이런 것을 오류 삽입(Fault Injection)이라고 하는데, 하드웨어도 완벽하지 않기에 생각하지 못한 오류들을 발생시킬 가능성이 있으며, 실제로도 여러 오류를 보고하기 위해서 하드웨어에 상태(Status)값을 가지는 레지스터가 존재하는 경우도 있다. 또한, 특정 시간이 되어야 발생할 수 있는 시나리오를 단위 테스트할 때도 활용할 수 있는데, 인위적으로 시간을 조작해서 테스트하려고 하는 함수를 실제 시간에 관계없이 호출할 수도 있다. 이런 부분들은 단위 테스트 실행 시간을 줄여주게 되어 효과적으로 빨리 원하는 테스트를 실행할 수 있도록 만들어 준다. 단위 테스트 자체는 개발자가 자신의 개발 환경에서 빠르게 실행해서 피드백(Feedback)을 얻을 수 있어야 제대로 활용될 수 있다.

사용자 입력을 단위 테스트에 전달해야 할 상황이 있다면(즉, 사용자 인터페이스를 담당하는 함수를 단위 테스트한다면), 이벤트(Event)를 전달하는 메커니즘을 활용할 수도 있다. 물론, 이때는 거의 시스템 통합 테스트(System Integration Test)에 가깝기에 단위 테스트라고 부르기는 어려울지 모른다. 특히, 이런 경우에는 재현(Replay)에 대한 방안을 찾아야 하기에, 스크립트와 같은 형태의 사용자 입력을 지원하는 방법을 사용해야 할 것이다. (예를 들어, “Selenium”의 경우 웹 기반의 사용자 인터페이스를 테스트하는데 효과적으로 사용할 수 있는 도구다.) 필요하다면 기존의 사용자 입력 방법은 그대로 두고, 새로운 사용자 입력 방법을 사용할 수 있는 에이전트(Agent)를 테스트하려는 소프트웨어에 추가할 수도 있다. 물론, 이런 부분들이 오버헤드처럼 보일지도 모르지만, 테스트를 빨리 할 수 있다는 점에서 충분히 고려해 볼만한 일이다.

단위 테스트는 각각의 테스트가 완벽히 분리되어 실행되어야 한다. 따라서, 테스트를 실행하는 데 필요한 상황(전제 조건)까지 만들어야 하는 것들이 있을 수 있다. 예를 들어, 데이터의 전송을 하기 위해서는 테스트를 실행하기 전에 미리 데이터 전송이 가능한 상황을 만들어야 한다는 점이다. 사실 이런 상황을 만들기 위한 과정이 쉽지 않을 수도 있다. 실제 테스트 대상이 되는 함수보다 테스트 코드가 더 커질 수도 있다. 이것을 쓸데없는 일이라고 생각하기 시작하면 단위 테스트를 왜 해야 하는가에 회의감이 들 수도 있다. 이런 현상이 발생하는 이유는 함수의 설계 자체가 단위 테스트를 하기 어렵기 때문이다. 미리 단위 테스트할 것이라는 것을 알고 만든 코드와 나중에 단위 테스트가 추가되는 코드는 구조가 다르다. 즉, 단위 테스트를 하기 위해서는 설계 자체부터 충분히 고려해야 한다는 뜻이다.

단위 테스트를 만들어가는 과정을 개발자들이 지루하게 느끼는 것은, 단위 테스트로 인해서 비효율적으로 일한다는 생각(필요없는 일을 하고 있다는 느낌)을 가지게 되기 때문이다. 과제 관리자가 남에게 보여주기 위해서 단위 테스트를 하게 된다면 이런 생각은 더 커질 것이다. 그리고, 단순히 테스트 커버리지를 측정해서 좋은 결과를 얻기 위해서 단위 테스트를 사용하게 되면, 개발자들은 다양한 인위적인 방법으로 테스트 커버리지를 높이기 위한 방법을 찾아서 사용하게 될 것이다. 이런 상황은 누구도 원하는 것이 아니다. 개발자에게 도움을 준다는 인식이 없는 상태에서는 의미있는 단위 테스트를 할 수 없다. 개발자들은 자신의 실력이 향상되고 있으며, 앞으로 나아가고 있다는 느낌을 중요하게 생각한다. 따라서, 단위 테스트를 목표를 달성하기 위한 수단으로 활용하는 것이 아니라, 개발자의 동기를 자극할 수 있도록 만들 필요가 있다. 단순히 테스트 커버리지를 높이기 보다는 그것을 통해서 무엇을 배울 수 있는지를 이야기하는 것이 좋을 것이다.

실제로 테스트 커버리지를 높이기 위해서 단위 테스트를 인위적으로 조작하는 경우도 있다. 회사의 개발 프로세스에서 특정 테스트 커러비지 수치를 일종의 단계 종료 조건으로 설정한 경우, 그것을 만족하게 만들기 위해서 편법으로 코드에 조건부 테스트를 걸어서 10% 미만의 코드에 대해서 테스트를 막아놓는 경우도 있었다. 이런 태도는 비록 과제를 완료시키기 위한 필수조건을 만족하게 하겠다는 생각에서 나온 것이겠지만 “개발자의 양심”상 올바른 행동이 결코 아니다. 차라리 회사에서 제시하는 평가 기준을 모든 과제에 동일하게 적용하는 것이 옳지 못하다는 것을 이야기하고, 현재 시점에서의 테스트 커버리지가 어떻게 되는지 정확히 보고하는 것이 올바른 선택일 것이다. 눈가림은 과제 책임자 자신에게는 잠시동안 이로울지 몰라도 함께 일하는 팀원들에게는 잘못된 가르침을 줄 가능성이 높다. 투명하지 못한 과제 관리는 이해 관계자 모두에게 잘못된 신호를 줄 수 있으며, 그로인해서 발생하는 모든 책임은 관리자만의 몫이 아닌 팀원 전체의 것이 된다.

단위 테스트가 좀 더 효과적으로 적용되기 위해서는 테스트 케이스를 작성하는 것을 어떻게 하면 쉽게 할 수 있는지 고민해야 한다. 좋은 도구를 사용한다면 작업 일부를 자동화시킬 수도 있을 것이다. 또한, 테스트 케이스를 작성하는 방법을 각각의 경우에 따라 어떻게 작성해야 하는지 정리해서 팀원들과 함꼐 공유하는 것도 좋다. 팀원 개개인의 노력을 공유해서 좀 더 쉽고 새로운 방법을 찾아내면서, 설계도 개선할 방안들을 만들어 나가야 할 것이다. 리팩토링이나 디자인 패턴도 같이 익혀나가는 것이 중요하다. 실제로 TDD와 같은 방법을 사용하기 위해서는 기본적으로 그런 기술들을 익히고 있어야 대화 및 의도를 명확하게 설계와 코드에서 드러낼 수 있기 때문이다. 적어도 단위 테스트라는 것이 자신들의 역량 개발에 많은 도움이 된다는 것을 개발자 스스로 체험하는 것이 가장 중요하다.

단위 테스트를 실행하는 것은 빨라야 하고 즉각적인 결과를 얻을 수 있어야 한다. 또한, 소프트웨어 개발자가 쉽게 접근할 수 있는 방법이 필요하다. 이것들이 잘 이뤄지지 않을 경우 개발자는 자신이 하는 일에 대해서 지연요소로 단위 테스트를 꼽게 되는 일이 생길 수 있다. 구현해야 할 것들을 많은데, 그것들 하나 하나를 전부 테스트하는 것은 자신이 구현할 내용에 더해 대략 150%의 일이 더 추가될 수도 있기 때문이다. 이런 현상은 구현 일정에 쫓기게 되면 흔히 생기며, 개발자는 코딩만 하고 끝내는 시점까지를 구현으로 보는 경우가 많기 때문이다. 일반적으로 심한 스트레스 상황에서는 가능한 빨리 일을 처리하고 싶어하는 것이 사람의 심리다. 그런 상황에서 단위 테스트는 크게 중요하게 생각되지 않을 가능성이 높다.

TDD(Test Driven Development)를 적용하는 것은 상당한 수준의 코딩 능력을 요구한다. TDD의 단계를 보면, 실패하는 테스트를 먼저 만들고 그것이 성공하도록 구현(Implement)한다. 또한, 새로 구현된 코드는 기존의 코드와 합쳐져 리팩토링도 해야 한다. 리팩토링 시에 기본이 되는 지식은 디자인 패턴과 같은 중복과 반복적인 수정을 줄이는(코드의 중복과 비효율을 없애고 변화에 대응할 수 있도록) 각종 기술(Technique)을 익혀야 한다. 당연히 테스트를 쉽게 하는 소프트웨어 구조도 생각해야 할 것이고, 요구사항에 대한 우선순위 및 각각의 요구 사항에 대한 사용 시나리오도 만들어 나가야 한다. 이런 과정들을 잘 이해하고 있다면 TDD로 접근하는 것이 빠르겠지만, 우리나라의 대부분 개발자들은(특히 임베디드 시스템 개발자) 이런 것에 익숙하지 않다.

문제는 과제의 시작 전에 적용할 방법론에 대해서 충분한 이해 없이, 결과를 위주로 한 잣대를 과제에 들이미는 것이다. 당연히 소프트웨어 개발자들은 자신이 지금까지 해왔던 것들을 바꾸라는 요구를 쉽게 받아들일 가능성은 없다. 그냥 또 다른 회사의 지침(정책)이 과제에 떨어져서 자신들을 괴롭히고 있다는 생각만 하게 될 것이다. "제발 개발에 집중할 수 있는 시간을 주세요."라는 아우성이 여기서도 나올 수 있다. 새로운 방법론을 적용하는 것은 그만큼의 교육이 필요하다는 것과 그것을 작게라도 업무에 적용해서 경험하는 과정이 필요하다. 하지만 대부분의 소프트웨어 개발인력은 한 과제가 완료되면 간략한 사후 검토(Post-mortem) 회의를 거친 후 즉각 다른 과제로 투입된다. 실무를 통해서 익힐 수 있는 것은 실패의 반복일 뿐이고, 그렇게 얻은 경험으로 다시 다른 과제의 실패에 뛰어들게 되는 것이다. 변화는 반복적으로 과제를 하는 것 만으로는 이루어지지 않는다.

따라서, TDD와 같은 개념을 바로 과제에 적용하기 전에 테스트 자동화라는 관점을 먼저 소프트웨어 개발자들이 알 수 있도록 해야 한다. 그런 관점에서 어떻게 하면 테스트가 쉬운 코드를 만들 수 있는지에 대한 학습이 선행되어야 할 것이다. 테스트 자동화는 다양한 수준에서 진행될 수 있지만, 가장 효과가 큰 것은 아무래도 테스트가 가능한 함수들과 그것을 테스트하기 위한 테스트 코드들이 합쳐져, 빌드 서버(Build Server) 및 테스트 서버(Test Server)에서 자동으로 수행될 수 있도록 만드는 것이다. 이를 위해 테스트 프레임워크나 빌드 서버에 설치되는 각종 자동 실행 도구들이 이미 오픈 소스로 제공되는 것들이 있으며 구축은 단순 시간의 문제다. 과제 관리자라면 소프트웨어 개발자에게 코딩의 완료를 묻지 말고 해당 기능(혹은 함수)에 대한 테스트 코드가 있는지와 그 테스트 코드를 수행했을 때 얼마나 많은 코드가 검증될 수 있는지 직접 확인해 보아야 할 것이다. 한번에 모든 것을 이루기는 어렵지만 그렇다고 손을 놓고 있다면 시간이 흘러도 달라질 것은 아무것도 없다.

# [ 이중화 vs 원가절감 ]

지금은 대부분의 임베디드 소프트웨어는 이중화로 구성하는 것이 일반적이다. 소프트웨어의 이중화란 필요한 펌웨어(Firmware)를 두 개를 가지고 있는 것을 말한다. (물론 다른 이중화도 있다.) 저장장치가 문제가 생길 경우를 대비해서 회복하는 방법으로 이중화를 선택한 것이다. 펌웨어 업데이트도 당연히 두 개의 이중화된 펌웨어 각각에 대해서 다 해주어야 한다. 하지만, 문제는 이중화를 하기 위해선 필요한 저장공간의 비용 문제가 따른다는 점이다. 예를 들어, Linux와 같은 것을 펌웨어의 운영체제로 사용하기 위해서는 “루트 파일 시스템(Root File System)” 공간 등이 차지하는 비중이 크기에 기본적으로 큰 메모리 공간이 필요하다. 따라서, 이중화된 공간을 사용하기 위해서는 비용도 그만큼 증가할 수밖에 없는 것이다. (경우에 따라서는 시스템 복구 목적과 시스템 사용 목적 각각에 대해서 다르게 구성할 수도 있다.)

예전에 다니던 회사에서 비용 문제로 이중화를 하지 않은 경우를 보았다. (사실은 이중화란 개념이 없었던 것으로 보인다.) 이럴 경우 새로운 기능의 펌웨어 업그레이드나 기존 펌웨어의 오동작이 발생할 경우 복구하는 비용이 많이 들어가게 되고, 사용자는 그 시간 동안 사용할 수 없는 제품에 대해서 좋은 경험을 가질 수가 없었다. 이러한 비용에는 단순히 소프트웨어 A/S만 해당하는 것이 아니라, A/S를 담당한 직원이 직접 고객을 찾아가서 해당 물건을 수거할 수밖에 없었고, 수거하는 절차 및 방법도 상당히 까다로웠다. 단순히 비용이 많이 든다고 이중화를 지원하지 않는다는 것은 어떻게 보면 사용자를 전혀 고려하지 않은 일방적인 회사 입장의 결정으로 보인다. 사용자는 자신의 실수 혹은 시스템에 이상이 있더라도 항상 제품이 잘 동작할 것이라는 막연한 기대를 가지고 있기에, 이런 상황이 빈번하다면 제품에 대한 신뢰에 심각한 문제가 있다고 보는 것이 당연한 태도다.

기본적으로 사용자가 원하는 것을 만족시켜줄 의무는 제품을 만드는 사람들에게는 있다. 돈을 주고 사간 제품에 문제가 있으면 고객은 즉각 반응하게 되고, 그런 것들이 한 번 두 번 쌓이게 되면 결국은 시장의 신뢰를 잃고 마는 것이다. 비용을 조금 줄이다가 시장을 잃어버릴 수 있다는 생각은 왜 하지 못하는 것일까? 그것은 아마도 자신이 만든 제품에 대한 자신감이 없다는 것이 이유일 것이다. 자신이 고생해서 만든 제품에 대한 자존심이 있다면, 최고의 품질과 고객 만족을 위해서 최선을 다할 것이다. 그냥 마지못해서 만들어내는 제품에는 장인 정신이 없기에, 대충 만들어서 QA에서 버그가 없다고 판단되면 시장에 내다팔기에 급급할 것이다. 최고의 엔지니어가 되려면 품질 기준을 최고로 가져가야 하지만, 단순히 일정을 최고로 짧게 만들어야 하는 제품에서는 그런 것들을 찾을 볼 수 없다.

"좋은 제품을 빠르고 싸게 만들어야 한다."고 회사는 강조한다. 좋은 제품은 빨리 만들기도 어렵고 싸게 만들기도 어렵다. 따라서, 좋고 빠르게 싸게 만들려면 몇 가지는 포기해야 한다. 포기되는 것이 고객의 만족감이라고 한다면 싸기만 한 제품은 절대 팔리지 않을 것이다. 그런 시장은 이미 지나간 흐름일 뿐이다. 좋은 제품을 제값을 받고 팔아야 한다. 빨리 만들기 위해서는 만드는 과정에서 어떻게 품질을 잡을지를 고민해야 하고, 그 고민의 결과를 적용할 수 있도록 내부적으로 관리해야 할 방법도 있어야 한다. 그냥 단순히 목표만 이야기하고 그것을 마치 어겨서는 안 되는 절대 진리인 듯이 지시하는 것은 하늘의 별을 따라고 하는 것과 다를 게 없다. 내부의 문화는 "말씀"으로 한 번에 이뤄지는 것이 절대 아니기 때문이다. 힘을 가진 "말씀"을 지속시키기 위해서는 조직이 필요하고, 문화로 받아들이기 위한 절대적인 시간이 있어야 하기 때문이다.

# [ 디자인 패턴(Design Pattern)에 대해서 ]

소프트웨어로 같은 문제를 해결하는 방법은 다양하게 있을 수 있다. 이미 많은 문제를 소프트웨어 개발자들은 풀었으며, 그러한 경험의 축적은 오랜 시간 동안(대략 60년?) 이루어져 왔다. 하지만, 새롭게 개발에 투입되는 인력들을 사실 그런 사실을 잘 모르고 일을 시작하게 되며, 지식도 부족한 것이 사실이다. (어쩌면 모든 분야의 신입이 겪는 문제점라고 생각할 수 있지만) 디자인 패턴은 이렇게 오랫동안 다양한 과제에서 반복적으로 발생한 중복과 의존성이라는 문제를 해결하는 정제된 방법이다. 사실 디자인 패턴을 제대로 이해하고 사용하기 위해서는 나름의 경험과 공부가 필요하며, 단순히 책으로 보고 배울 수 있는 수준의 것은 아니다. 각각의 문제를 해결해 본 경험이 있는 경우에는 왜 그렇게 해야하는지 이해할 수 있지만, 그렇지 않은 사람들은 문제의 본질보다는 겉모습에 현혹 당하기 쉽다.

디자인 패턴을 이해하는 데 필요한 지식을 보면, 객체지향 사고와 프로그래밍 언어, UML(Unified Modeling Language) 표기법 정도는 알아야 한다. 물론 아주 자세한 지식수준보다는 실제 사용한 경험이 중요하다. 따라서 기존 C 프로그래머가 디자인 패턴을 익히기는 쉽지 않다. 효과적으로 디자인 패턴을 이해하기 위해서는 이미 경험한 문제 상황을 잘 파악해야 하기에, 코딩 경험이 최소 3년 이상은 있어야 할 것이다. 문제를 실무에서 경험해보지 않으면 무엇이 정말 문제인지 알 수 없기 때문이다. 객체지향 관점에서 시스템을 분석할 수 있는 능력도 있어야 하고, 객체 지향 코딩의 핵심 원리인 “SOLID원칙”과 코드의 의존성 분리와 응집성이 왜 중요한지도 알아야 한다.

디자인 패턴을 익힌 후 가장 먼저 할 수 있는 일은 아마도 자신이 이미 작업하고 있는 코드를 개선하는 활동일 것이다. 이것은 또 다른 차원의 지식을 요구한다. "외적인 행동(Behaviour)는 변경 없이 내부적인 구조를 변경하는 것"이기 때문이다. 이를 위해서는 다시 "리팩토링(Refactoring)"과 같은 지식이 요구된다. 또한 리팩토링의 선행 지식으로는 "단위 테스트(Unit Test)"기법을 익숙하게 적용할 수 있는 능력이 있어야 한다. 따라서 "객체지향 프로그래밍 🡪 UML 🡪 디자인 패턴 🡪 단위 테스트 🡪 리팩토링”으로 연결되는 학습의 고리가 있어야 최종적으로 실무에 사용할 수 있는 수준이 되었다고 볼 수 있다. 만약 새로운 코드를 개발하는 시점이라고 한다면 “테스트 주도 개발(Test Driven Development: TDD)”도 추가적으로 필요할 것이다. 물론 C언어 개발자라고 해서 디자인 패턴을 사용하지 못한다는 것은 아니다. 개념적으로 동일한 것을 적용할 수는 있지만, 객체지향 언어가 기본적으로 제공하는 많은 것들을 활용하지 못하기에 구현 부담이 커질 수는 있다. 어쨌든 새로운 것을 알고 이를 현실 세계에 제대로 적용하기 위해서는 다양한 부분에 대한 이해가 선행되어야 한다는 점은 사실이다.

과제 담당자는 팀원들에게 단순히 "Design Pattern을 적용해서 코딩해"라고 이야기하는 것은 어디서부터 시작해야 할지에 대한 아무런 해결책을 주지 못한다. 팀원들의 역량이 그 수준에 이르기 위해서는 "학습"이라는 시간이 필요하며 그 뒤를 이은 "경험"이라는 것이 있어야 한다. 급한 불을 끄기 위해서 막무가내로 지시를 하면 아무도 그것을 따르지 않는다. 단순히 "따르는 척" 할 뿐이다. 이런 것들이 오랫동안 누적되면 소프트웨어 개발자들은 윗사람에 대해 기대를 접게되며 숙제만 늘었다는 불평을 할 것이다. 진정한 개선은 소프트웨어 개발자들이 자신의 의지로 지속적인 개선을 시도하는 "개발자 문화"에 있다. 누군가의 지시로 시작될지도 모르지만 내부적인 필요에 의한 "자발적인 학습"과 이를 통한 실무 개발에서 발생하는 문제 해결을 터득해 나가는 것도 한 가지 방법이다. 디자인 패턴을 배웠다고해서 모든 문제가 공짜로 해결되지도 않으며, 디자인 패턴을 적용했다고해서 모든 문제가 해결된 것도 아니다. 상황을 이해하고 적절한 도구를 사용하는 것일 뿐 무엇을 적용했기에 문제가 없다는 것은 책임있는 개발자가 취할 태도가 아니다.

# [ 복잡한 문제 풀기 ]

소프트웨어 개발은 복잡성(Complexity)을 어떻게 다룰 것인가에 따라 성공과 실패가 주어진다. 여기서 말하는 성공은 당연히 주어진 예산과 자원, 일정하에서 사용자의 요구사항(풀어야 할 문제)을 제대로 해결하는 것을 말한다. 성공한 제품보다는 성공적인 과제 수행의 측면에서 판단한 것이다. 실패한 과제라도(취소된 것이 아니라면) 시장에서 성공할 가능성이 있다. (물론 성공적인 과제보다는 확률이 낮겠지만) 소프트웨어의 복잡성은 모듈화의 실패와 모듈 내부의 복잡성에 기인하는 부분이 크며, 모듈간의 유동적이고 불안정한 인터페이스로 인해서 발생한다. 하드웨어의 경우에는 고정된 인터페이스를 이용해서 각각의 부품이 연결되지만, 소프트웨어는 수시로 변경될 가능성이 있는 연결 구조로 만들어지기 때문이다.

소프트웨어 개발자가 달성해야 할 목표는 내재된 복잡한 문제를 어떻게 풀 것인가를 고민하는 것이다. 생각보다 문제는 단순한 데 있다. 소프트웨어 개발자의 코딩 대부분은 고난도의 알고리즘을 개발하는 것이 아니라, 적절한 난이도의 많은 코드를 개발하는 것이다. 난이도를 적절하게 만들기 위해서는 큰 문제를 조각으로 분리하고, 분리된 것들 간에 최소의 인터페이스만 정의하는 것이다. 이 때 지나치게 많은 인터페이스는 문제를 복잡하게 만드는 주요 요인이므로 반드시 절제하는 것이 좋다. 하나의 서브시스템은 “7+-2”라는 법칙에 근거해서 지속해서 쪼개 나가면 된다. 어떤 모듈의 내부 구성은 다시 “7+-2”개로 구분되며, 다시 그렇게 나누어진 것들을 지속해서 “7+-2”개의 범위에서 잘게 쪼개면 된다. 최종적으로 남는 "함수나 메쏘드"수준에서는 하나의 개념을 구현해야 하며, 내부에는 최소의 변수를 유지하도록 해야 한다. 인자(argument)의 수는 될 수 있으면 없는 것이 좋지만 최대 4개 수준까지 만들어 주는 것이다. 이런 방법을 한 마디로 표현하면 “분리 정복(Divide & Conquer)”이라고 할 수 있다.

물론 이런 것이 쉽게 이루어지는 것은 아니다. 하지만, 목표는 복잡한 문제를 서로 관련성이 최소화되는 방향(Orthogonal)으로 지속해서 쪼개가는 것이다. 쪼개진 문제들을 푸는 데는 어려운 코딩이 필요한 것이 아니다. 단순한 문제를 단순히 풀어나가는 코딩이 된다는 것이다. 전체적인 것으로 봤을 때 이런 구조는 마치 나무의 뿌리에서 시작해서 땅속을 잔뿌리를 길게 뻗어 나가는 것처럼 보일 것이다. 그 역도 존재한다. 더 뻗어 나갈 필요가 없는 순간은 비즈니스 로직과 하위의 유틸리티(Utility)성의 계층이 만나는 접점이며, 이 때는 다시 하위 계층에 대한 의존성이 극대화가 되도록 하는 것이 필요하다. 하위 계층에서 제공하는 인터페이스들을 상위 계층이 가능한 많이 사용하도록 만드는 것이다. 즉 상위 모듈의 하위 모듈에 대한 의존성이 높아지도록 만든는 것이다. 이것은 선형적인 호출에 의한 의존성으로 문제의 복잡성을 높이지는 않는다.

복잡한 문제는 모든 것을 한 번에 해결하려고 접근 해서는 안 된다. 전체적으로 한 방에 해결할 수 있는 방법은 존재하지 않는다. 그리고, 사람의 두뇌는 다양한 문제를 한 번에 풀 수 있을 정도로 특화되어 있지 않다. 따라서, 문제에 대한 분석 및 해결 방안은 어떻게 문제를 나눌 것인가에 의존적이게 되며, 문제를 나누는 방법은 서로 관련이 적은(없는) 부분들을 분리하는 것이다. 예를 들어, 데이터를 입력하는 것과 데이터를 출력하는 것은 나누어질 수 있다. 또한, 그 사이에 있는 데이터를 조작하는 것도 독립적으로 존재할 수 있는 부분이다. (물론, 데이터 자체의 양식에 대해서는 관련을 맺게 되겠지만, 최소한의 의존성을 유지하는 방향이 좋다.) 한 번에 하나의 작은 문제를 다룰 때 가장 효과적으로 해법을 제시할 수 있으며, 그것만이 소프트웨어가 가지는 내재된 복잡성에 대한 빈번한 실패에서 오는 위기를 극복하는 좋은 대응법임을 알게 될 것이다.

# [ 소프트웨어 설계 원칙 ]

소프트웨어를 만들어 가는 과정에서 가장 중요한 부분은 제대로 된 사용자들의 요구사항을 수집하는 것이다. 이렇게 만들어진 요구사항 중에서 시스템의 기능적인 부분으로 구현해야 할 구성요소를 찾고, 비기능적인 요구를 만족하는 구성요소들 간의 관계를 만들어 가는 것이 소프트웨어를 설계하는 활동이다. 하지만 대부분의 경우 요구사항을 전달받는 과정은 아주 단순하며, 비기능적인 요구사항은 대부분 성능에 대한 것에 한정된다. 따라서 이러한 것들을 기반으로 해서 만들어지는 시스템은 대부분 확장성이나 유지 보수성, 변경 용이성 등의 다른 비기능적인 요구들을 만족하지 못하게 되며, 기능적인 요구사항들도 개발 중에 흔한 변경으로 인해서 더 복잡한 시스템을 만들어내는 결과를 가져온다.

문제는 여기서 끝이 아니라는 점이다. 소프트웨어는 하드웨어와는 달라서 지속적인 변경에 대해 대비를 해야 하는데, 위와 같이 개발된 시스템을 가지고 변경을 하게 되는 순간 무엇인가 문제가 있다는 것을 깨닫게 된다. (물론 이미 자신이 만든 코드에 대한 자신감이 없을 수도 있다.). 하지만, 대부분의 경우 일명 "땜빵" 코드를 짜게 되고 설계는 점점 오리무중으로 빠지게 된다. 아예 설계가 없거나 설계서와 구현이 너무 큰 차이를 보이게 될 것이다. 물론 이것은 "혼자만 겪는 문제"는 결코 아니다. 대부분의 IT 업종에서 발생하는 일이고 쉽게 개선하는 것도 어려운 일이다. 하지만, 이런 상황을 개선할 수 있는 몇 가지 원칙들이 있다. 최소한 이것이라도 지키면서 코딩을 하면 나중에 어떻게든 조금이라도 편하게 일할 수 있다. (하지만 이것도 결국은 "땜빵"에 불가할 수 있으니, 처음부터 제대로 만드는 것이 더 중요하다.) 사실 이러한 원칙들은 이미 오래 전에 경험과 통계로 검증된 것들이라 별로 새로운 것들은 아니다.

첫 번째 원칙은 "코딩을 쉽게 하라"이다. 복잡한 구조를 만드는 것이 아니라 작고 이해하기 쉬운 구조를 만들어 나가야 한다. 짧은 코드를 짜야 하지만 막무가내로 짧게 짜라는 것이 아니라 이해하기 쉬운 짧은 코드를 짜라는 말이다. 한 가지 일만 해야 하는 코드가 짧게 다듬어져 있고 이해하기도 좋다면, 유지보수를 하는 입장에서도 쉽게 고칠 수 있는 코드가 될 것이다. 하나의 코드 조각은 항상 하나의 일만 해야 한다. 하나의 함수(혹은 메쏘드)는 반드시 하나의 일만 해야 한다. (추상적인 의미에서 한 가지 역할을 하는 것이다.). 구체적인 수준으로 나아가기 위해서는 추상화 수준을 한 단계 낮춰야 하고, 그렇게 낮아진 수준에서도 무조건 한 가지 일을 하도록 만들어야 한다. 짧은 코드는 많은 변수를 원하지 않는다. 최소의 파라미터 개수를 유지해야 한다. (아예 없으면 더좋다.). 왜냐하면 그렇게 만든 짧은 함수에 대해서 테스트하는 것이 쉽기 때문이다. 단순히 몇 개의 인수 값만으로 함수를 호출해서 결과 값을 비교하면 될 것이기 때문이다. 변수가 많다는 말은 여러 가지 복합적인 일을 수행한다는 뜻이기에, 최적인 상황은 아무런 변수 없이 한 가지 일만 수행하는 함수를 만드는 것이다. 함수 내부에서 상태의 변경이 없다면 버그가 발생할 가능성도 줄어들게 된다.

두 번째 원칙은 "변경이 일어나는 부분에 대해서는 구체적인 것보다는 추상적인 구조를 만들라."이다. 하지만 변경이 어디서 발생할 것인가가 모호한 경우가 있다. 처음 개발하는 제품이라면 그런 변경에 대한 부분을 찾기는 쉽지 않다. 편의상 "if() else"나 "switch() case" 등이 자주 사용되는 경우에는 그런 변경이 나타날 가능성이 높다. 흔히 칩셋(Chipset)이나 자신의 설정에 대해서 이런저런 방법으로 제어를 분기하는 경우가 많은데, 이 때 주로 변경이 일어날 수 있는 조건들이 구체화된다. 어쨌든 이런 부분들을 만나면 일단은 뭔가 랩퍼(Wrapper)와 같은 것이 필요하다는 것을 생각해 주기 바란다. 변경에 대한 대비란 변경에 의존적이지 않은 코드를 만들라는 것이다. 변경은 어차피 있을 것이라는 가정하에 그것에 의존적인 코드를 짜는 것은 나중에 코드를 유지보수하는데 어려움이 따르기 때문이다. 물론 이런 변경이 다양한 형태를 가질 수 있고 구현하는 언어에 따라 대처하는 방법이 달라질 수도 있다. 하지만 중요한 것은 그것을 처리하는 나름의 지식(Knowhow)을 함께 알아야 한다. 설계하는 과정에서는 변경에 대비하고 있어야 하며, 구현하는 과정에서는 모든 변경을 다 만들어줄 필요는 없다. 변경이 발생할 때 그 부분을 변경에 유연한 구조로 만들어주기만 하면 된다.

물론 위와 같은 것들을 하게 될 때 성능이 떨어진다는 이유로 거부를 하는 경우를 흔히 볼 수 있을 것이다. 트레이드오프(Trade-off)가 없다면 말이 안 되기 때문이다. 좋은 구조를 만드는 것은 다양한 시스템의 요구사항을 만족하기 위한 것이기에, 이 때도 역시 사용자들의 요구사항과 비즈니스적인 목표를 보고 결정해야 한다. 하지만 무조건 성능을 위주로 한 코딩의 대가는 상상 이상이 될 것이기에 신중한 결정이 필요하다. 초기부터 성능을 높이는 코딩은 지역적인 최적화만을 가능하게 만든다. 예를 들어, 85%에서 90%로 가는 것은 가능할 것이다. 하지만 전체 시스템적인 측면에서의 최적화는 구조도 함께 고려해야 한다. 현재를 100%로 잡을 경우 200%나 300%로 가기 위해서는 구조가 반영되어야 하기 때문이다. 따라서 설계는 시스템의 다양한 측면을 고려해야 하며, 설계를 유연하게 유지하기 위해서는 쉽게 코드를 변경할 수 있는 다양한 방법들이 코딩에서 반드시 반영되어야 한다.

모든 문제를 한 번에 해결하는 것은 천재라면 가능할지도 모른다. 하지만 우리는 대부분 천재가 아니고, 한 번에 해결하는 것(Silver Bullet)이 불가능하다는 것을 이미 잘 아는 사람들이다. 따라서 한 번에 모든 것을 해결한다는 툴(Tool)도 거짓이라는 것을 잘 알고 있으며, 한 번만 설계하는 것도 좋지 못하다는 것을 잘 알 것이다. 따라서, 소프트웨어 설계는 반복의 과정이라는 것이 옳다. 반복적인 설계를 한다는 말은 설계가 살아있는 결과물을 만들어내고 지속해서 수정된다는 것을 의미한다. 관리란 바로 이렇게 끝없이 개선해 나가는 전 과정을 의미하는 말이다.

모든 관리되는 것들은 항상 나름의 영향력을 가진다. 설계는 구현에 대한 청사진을 제시하고, 다시 구현에서 발견된 설계상의 문제점은 설계를 더욱 완벽하게 만들어 가게 된다. 하지만 대부분의 실수는 여기서 발생한다. 한번 만들어진 설계에 대해서 발견된 오류를 변경하지 않고 그대로 둠으로써, 이후에 많은 설계상의 오류들이 관리되지 않은 채로 코드에 남게 된다. 코드는 점점 설계와 무관해지고 설계지침이 없는 코드는 점점 복잡하게 변화되어 나갈 것이다. 이런 설계는 관리 시스템의 저장 공간만 차지하고, 간혹 새로 들어온 사람들이 한 번쯤 훑어보는 문서가 되어간다. 관리에서 요청하는 프로세스의 필수 산출물로만 활용될 뿐 실용적인 측면에서는 아무런 가치를 지지니 못하게 된다.

좋은 설계를 하기 위해서는 설계에 대한 검토와 개선을 꾸준히 해야 한다. 어쩌면 이것은 당연한 말일지도 모르지만 제대로 지켜지기는 힘들다. 왜냐하면, 대부분의 소프트웨어 개발자는 문서를 쓰는 것에 익숙하지 않고 좋은 구조의 문서를 만드는 것은 더 못하기 때문이다. 하지만 좋은 문서를 만들기는 의외로 쉽다. 소설이나 시를 쓰듯이 비유나 은유와 같은 것을 필요로 하지 않기 때문이다. 기술 문서는 정확한 내용을 기술하는 것이 전부다. 더불어 설계에 필요한 것은 그러한 기술적인 결정들에 대한 이유를 적어놓는 것이 핵심이다. 어떤 결정을 두 번 반복할 이유가 없도록 만들기 위해서 반드시 왜 그러한 설계를 했는지 사유를 정확히 기술해 주어야 한다. 세밀하게 모든 것을 다 적는다고 생각해서도 안 된다. 세부적인 것은 언제든 변경될 가능성이 있기 때문이다. 따라서 적정 수준의 결정된 것들과 그것들에 대한 이유를 나열하는 것으로 충분하다.

설계는 큰 문제를 분석하면서 반복적으로 조금씩 구체화된다. 시스템을 입력/처리/출력이라는 관점에서 나누고, 각각에 대해서 필요한 구성요소들을 정의한다. 그렇게 정의된 구성요소를 다시 입력/처리/출력이라는 관점에서 모듈화시키고, 각각의 모듈들을 분리해서 개별적인 역할로 나눈다. 나누어진 모듈의 구성요소들을 계속 세밀하게 연결하고, 역할과 책임을 주는 것을 반복하는 것이 설계이다. 결국 세밀하게 나누는 절차가 개발자의 코딩 역량 수준에 적합한 추상화 수준을 만족하면 바로 코딩에 들어가면 된다. 정도는 다르겠지만 어쨌든 코딩할 수 있는 충분한 근거를 제공할 수 있는 수준에서 설계의 구체화는 마무리되고 실제 구현을 시작하게 된다. 구현은 설계를 검증하는 활동이며 잘못된 설계는 반드시 수정되어 항상 최신 구현 내용과 일치할 수 있도록 유지되어야 한다. 따라서 구현은 설계에 대한 검증과 피드백을 주는 활동이라고 볼 수도 있다.

소프트웨어에 대한 재사용은 항상 중요한 이야기 소재가 된다. 소프트웨어를 재사용하기 위해서는 어떤 것들이 필요할까? 하지만 그 전에 어떻게 재사용을 할지부터 먼저 정해야 한다. 정책적으로 하이엔드(High End) 제품과 로우엔드(Low End)제품은 공용화 하기 어려운 경우가 많다. 미들엔드(Middle End) 제품의 경우에는 하이엔드와 같이 공용으로 활용할 수 있는 부분이 많지만, 제품군이 달라질 경우에는 재사용의 범위도 줄어들 수밖에 없다. 플랫폼 수준에서의 재사용과 모듈 수준에서의 재사용은 다른 문제다. 플랫폼 수준에서의 재사용은 주로 동일한 제품군 내에서 의미를 가지며, 모듈 수준에서의 재사용은 제품군 수준을 벗어나 동일한 기능 수준의 재사용으로 볼 수 있다. 둘 중에 어떤 것이 더 좋다고는 볼 수 없지만, 이미 만들어진 소프트웨어를 가지고 재활용을 할 수 있다는 점은 충분히 매력적임은 틀림없다.

어떻게 만들어야 재사용하기 쉬울까? 개발자에게는 이것이 진정한 질문이 될 것이다. 재사용을 원하는 소프트웨어는 플랫폼이건 모듈이건 간에 “좋은 인터페이스”를 가지는 것이 기본이다. 외부에서 사용할 수 있는 인터페이스를 제공하고 충분한 사용 예와 문서 형태의 사용 방법을 알려줄 수 있어야 한다. 통합을 위해서는 기존의 소프트웨어에 대한 변경이 있을 수 있는데, 이 때 기존 소프트웨어의 변경보다는 재사용되는 부분에 의존적인 것들을 분리해서 관리할 필요가 있다. 일종의 랩퍼(Wrapper) 형태로 정의해두고 재사용 부분에 대한 통합을 가져가는 것이 좋다. 의존성의 분리(De-Coupling)는 여기서도 중요한 의미를 가지며 될 수 있으면 의존적인 부분들을 하나의 모듈 형태로 묶어두고 접근할 수 있는 통일된 방법을 인터페이스로 제공하는 것이다. 예를 들어, HAL(Hardware Abstraction Layer)를 통해서 일관된 하드웨어 접근을 가능하게 만드는 것이다.

항상 소프트웨어를 설계할 때는 재사용할 부분이 무엇인지를 확인해야 한다. 또한 그렇게 재사용되는 부분을 통합하기 위한 방법을 고려해야 하며, 의존성을 관리하기 위해서 코드를 인터페이스를 활용해 분리할 수 있는 방안을 마련해 두어야 한다. 이런 것들을 확장하면 자신이 만든 플랫폼이나 모듈들도 쉽게 재사용될 수 있다는 확신이 생길 것이다. 동일 제품군이나 혹은 다른 제품군에서도 재사용이 가능한 것들을 만들어 낼 수 있게 된다. 이때 자신이 만든 것들을 재사용하는 사람의 입장을 고려해서 다양한 사용 예와 일관된 인터페이스를 제공해 줄 수 있도록 해야 한다. 인터페이스가 일관되지 않으면 기본 가정을 유지하기 어려워지기 때문이다. 따라서 기본적으로 하나의 모듈을 개발한다면 인터페이스 부분에 대해서 더 세밀한 관리를 해야하며, 테스트할 수 있는 방안도 함께 제공될 수 있어야 한다. 이런 기반이 미리 갖추어져 있지 않다면 코드의 재사용은 말로만 가능할 뿐 실용적인 가치는 없는 일이 된다.

위와 같은 것들을 다 제공하면서 코딩한다는 것은 사실 이상적인 경우에 가깝다. 모든 모듈에 대한 테스트 케이스와 예제, 문서 등을 제공하는 것은 사실상 빠듯한 개발 일정에서는 불가능에 가까운 일이다. 따라서 이것은 회사 차원에서 재사용 전략을 어떻게 가져갈 것인가를 만들어서 과제에 기본적으로 적용할 수 있어야만 가능하다. 상품화에 정신없는 개발자에게 이런 것들을 시키려면 일정 계획에 이미 반영이 되어 있어야 한다. 단순히 "왜 내가 말한 데로 안 해!!!"라고 이야기하는 것은 자신이 할 일은 제대로 안 하면서 남에게 강요만 하는 꼴이 된다. 일을 시키는 사람은 투입될 노력이 얼마나 되는지 예상하고 시켜야 한다. 말로는 동작하는 소프트웨어를 만들 수 없다. 소프트웨어는 구현되고 실제로 제대로 동작해야지 의미가 있다. 재사용을 위한 전략을 세웠다면 그것을 실무에서 어떻게 적용할 것이고, 그에 따른 소프트웨어 구조 및 테스트 효율화 방안, 신규 개발과제에 적용하기 위한 모듈을 등록하고 유지할 수 있는 방안 등도 함께 있어야 한다.

소프트웨어 개발에서는 항상 과제 초반의 활동에 대해서 많은 이야기를 한다. 초반에 발생하는 작은 오류가 후반으로 갈수록 200배 이상의 비용을 발생시킬 수 있다는 것을 보여주면서 시작을 중시하는 태도를 보이고 있다. 설계 활동도 과제의 시작을 어떻게 하는가에 큰 영향을 받게 되며, 잘못된 요구사항 분석은 설계 수정에도 비용을 유발할 수 있다. 하지만 현실적으로 설계를 위한 준비 시간은 거의 없다. 새로운 과제를 할 경우에는 기존에 했던 과제에 대한 평가자료(추정의 근거가 될만한 것들)들이 없기에 설계에 얼마나 많은 시간을 할애할지 정확히 추측하기도 어렵다. 기존의 제품이 있는 경우에는 완전히 새로 설계하기도 쉽지 않은 것이 사실이다. 무리하게 설정된 과제 기간에서 살아남기 위해서는 어떻게든 변경을 최소화해야 하고 추가되는 기능도 줄여야 하기 때문이다. 초기 설계의 실패나 미비는 최종 결과물의 오류를 낳게 되지만, 제대로 설계할 시간이 주어지지 않는 것이 현실이다.

기존의 코드를 이용하는 것은 언제나 부담스러운 일이다. 특히, 기존 코드의 구조를 이해하지 못하는 인력으로 과제를 추진하는 것은 거의 자살에 가깝다. 기존에 존재하는 코드의 모든 것을 블랙박스(Black Box)화 시키는 것은 거의 절망에 가깝다. 왜냐하면 새로운 기능을 위해서 기존 코드를 변경해야 하기 때문이다. 그렇다면 이런 문제를 해결하는 방법은 무엇일까? 일단은 기존 코드를 최대한 건들지 않고 그냥 두는 것이다. "작동하는 코드는 변경하지 말라"는 법칙처럼 그냥 두는 것이다. 어떤 것을 추가해야 할지를 먼저 검토하고, 만약 기존 코드를 변경해야 할 경우라면 차라리 기존 코드를 복사한 후 그것을 수정하는 방법이 옳을 것이다. 무리하게 변경을 시도하기 보다는 일단 신규 코드를 기존 코드와 완전히 분리해서 동일한 역할을 하는 함수로 만든다. 그리고나서 컴파일 옵션을 통해서 신규 코드가 실행 이미지에 추가되도록 만든다. 기존 테스트나 신규 테스트는 기존 결과에 대한 변경이 없다는 것을 보장하는 수단으로 활용할 수 있을 것이다. 마지막으로 신규로 작성된 부분에 대해서는 자동화된 단위 테스트도 만들어 두어야 한다.

예를 들어, A라는 함수가 기존의 코드라고 하면, A 함수 상위에 랩퍼를 만들어서 신규로 작성된 B도 호출될 수 있도록 만들 수 있다. 이해하지 못하는 코드를 억지로 끼워 맞추는 방법 대신 추가적인 계층을 구성해서 넣는 것이 더 효과적이다. 어쨌든 우리가 원하는 것은 기존 코드의 변경을 최소화하는 것이기 때문이다. 설계 문서에는 추가되는 코드에 대해 어떻게 구성되어 있고, 다른 모듈과 어떤 관계가 있는지 설명해 줄 수 있어야 한다. UI와 같은 부분을 수정하는 것은 생각보다 쉬울 수 있다. 새로운 컨트롤(Control)을 정의하고 그것에 맞는 이벤트 핸들러(Event Handler)를 정의하는 프레임워크(Framework)가 미리 정의된 환경일 가능성이 높기 때문이다. 디자인 패턴을 활용할 수 있는 부분들도 많을 것이다. 예를 들어, 파싸드(Facade), 어댑터(Adapter), 브릿지(Bridge) 등의 패턴들은 전체적으로 클라이언트 코드와 서버 코드를 분리해서 변경할 수 있는 구조를 제공해주기 때문이다. C언어로 개발하더라도 잘 정의된 인터페이스를 활용해서 “관심의 분리(Separation of Concern)"라는 개념을 구현할 수 있다.

대부분 기존에 만들어진 코드는 고치기가 어렵다는 것이 정설이다. 어떻게 만들었든 간에 초반에는 지켜지던 규칙들이 시간이 흐를수록 점차 희미해져 가고, 결국에는 완전히 엉망이 된 코드가 되어간다. 엉망이 되기 전에 무언가 대책을 세우지 않으면 지속적인 유지관리 비용이 신규개발 비용보다 커지게 되는 상황이 발생할 것이다. 이런 것을 방지할 수 있는 단 한 가지 방법은 지속해서 모듈 사이의 간격을 늘리는 것이다. 모듈 간의 의존성을 지속해서 줄이고, 필요하다면 새로운 계층을 도입하고, 인터페이스를 기준으로 코딩하는 방법을 사용하는 것이다. 이런 방법을 통해 점차 독립적으로 재사용될 수 있는 구조로 모듈을 구성해나가는 것이다. 물론 성능을 문제로 그렇게 하면 안된다는 개발자들이 있을 것이다. 하지만 시간이 흐를수록 하드웨어의 성능은 개선될 것이 분명하고, 남은 것은 그런 좋은 성능에서도 제대로 관리되지 않는 소프트웨어가 있을 뿐이다. 약간의 성능을 낮추더라도 크게 문제가 되지 않는다면, 오히려 완벽한 분리를 지향해서 코드를 만드는 것이 옳다. 정말 성능이 문제가 될 상황이라면 차라리 소프트웨어를 전면적으로 재개발하는 것이 올바른 판단이다. 기존의 복잡한 구조를 유지하면서 달성할 수 있는 최적화된 성능은 그리 높지 않기 때문이다.

소프트웨어로 해결하고자 하는 문제들(혹은 개선하고자 하는 것들)은 복잡한 경우가 많다. 물론 간단한 문제도 있겠지만 기본적으로 여러 명이 과제를 해야 할 정도의 소프트웨어는 복잡함을 체계적으로 다루는 것이 핵심이다. 사회구조를 보면 다양한 계층이 있다. (물론 현대는 수평적인 관계라고 말하고 있지만, 누구도 그렇다고 생각하지는 않을 것이다.) 다양한 직업이 있으며 그것에 해당하는 역할도 세분되어가고 있다. 소프트웨어도 그렇게 고도화되고 세부적인 역할과 기능들이 정의되어감에 따라, 점차 더 복잡성이 높아져 가고 있는 것이 추세다. 이런 복잡성을 다루는 탁월한 기법은 "계층화된 구조(Layered Architecture)"로 설명하는 것이다. 계층화된 구조는 상세한 세부구조를 알지 못하더라도 전체적으로 파악할 힘을 주게 되며, 시스템을 구성 단위들로 나눌 수 있도록 만든다. 또한 각각의 계층이 해야할 역할과 책임 범위도 명확해진다.

계층(Layering)화는 사람이 가장 잘 할 수 있는 것 중에 하나다. 이미 우리는 계층화에 익숙하게 적응하고 있기 때문이다. 계층화는 지식의 축적 및 재사용에 대해서도 중요한 부분을 차지하고 있다. 계층화를 통해서 추상화(Abstraction) 수준을 높일 수 있으며, 특정 하위 계층에 대한 의존성을 분리해서 다른 것으로 대체할 수 있는 인터페이스를 정의할 수 있다. 네트워크로 구축되어 그래프(Graph)처럼 이런 저런 구성요소들(Components)과 연결된 경우를 상상해 보자. 이런 경우에는 잘 정의된 인터페이스를 정의하기가 어려우며, 다른 구성요소들과 규칙 없는 연결 관계는 변경의 영향을 다른 부분으로 파급시킬 위험에 처하게 할 것이다. 따라서 설계는 마치 "피라미드"와 같은 형식으로 전체적인 균형을 이루고 있는 모습으로 생각해 볼 수 있다. 추상화를 하기 위해서는 "하향식(Top-Down)" 접근이 큰 그림을 만들기에는 더 좋다. 최상위에서 정의된 추상화된 개념을 하위 계층에서 반복적으로 구체화시켜나가는 것이다. “상향식(Bottom-Up)”방식에서는 하위 구성요소를 모아서 더 큰 개념으로 모으고, 다시 모아진 개념들을 추상화시킬 계층을 정의하는 방식으로 최상위 개념까지 도달하게 된다. 실제 설계에서는 상향식 및 하향식을 섞어서 여러 번 검토해야 하겠지만, 초반에는 하향식으로 쪼개기를 반복하는 것이 편리하다.

그림은 특정 도구를 이용해서 그려야만 하는 것이 아니다. 도구는 생각을 도울 수 있는 것이라면 어떤 것이라도 상관없다. UML(Unified Modeling Language)을 사용하는 목적이 도구를 100% 활용하는 것이 아니라 사용자에게 유용한 실제 소프트웨어를 제공하는 것이며, 개발자와 관련된 이해관계자들이 서로 의견을 나누기 위한 것이다. 설계는 비교적 단순화된 형태로 시스템에 필요한 구성요소 간의 관계에 집중하고, 부수적인 것들은 생략해도 일을 진행하는 데는 큰 무리가 없다. 언제 설계를 끝낼 것인가를 결정하는 것은 시스템에 필요한 주요 구성요소들이 대부분 판별되고, 주요한 인터페이스들이 사용자가 제시한 주요 시나리오를 만족시키는 수준이 되는 순간이다. 그 때가 되면 코딩으로 전환해도 된다. 어차피 몇 번의 반복(Iternation)을 진행할 것이기에 그 때 점점 더 구체적인 것들이 만들어질 것이다. 완전한 설계란 존재하지 않으며(마치 결과물로 만들어진 소프트웨어가 완벽하지 않듯이) 지속해서 개선해야 할 것들의 일부분이다. 설계의 방향은 이미 이야기했듯이, 추상적인 부분에서 구체적인 부분으로, 외부에서 내부로 진행하는 것이 옳다. 외부 인터페이스를 정의하는 구성요소(Component)를 일단 정의하고, 그러한 구성요소들을 다시 더 세분화(분할)시키는 방향으로 시스템의 주요 부분들이 구성되도록 반복해서 만든는 것이다.

설계의 대부분 활동은 역할 및 책임을 구분하는 것이며, 어울리는 이름을 각각의 구성요소에 할당해 주는 것이다. 역할은 시스템의 설계가 진행되어감에 따라 더 구체적으로 주어질 것이고, 그 역할에 맞는 이름도 점차 더 길어지게 될 가능성이 높다. 정확한 역할을 서술하기 위해서는 지나치게 짧은 이름은 어울리지 않으며, 이름만들기(Naming) 규칙을 미래 정해두고 사용하는 것이 효과적이다. 같은 수준의 추상화를 반영할 수 있도록 같은 계층에 속한 구성요소들의 이름은 매니저(Manager), 핸들러(Handler)와 같은 것을 함께 사용하는 것도 좋다. 이름만 가지고도 어떤 수준의 추상화를 반영하는지 알 수 있도록 만들 수 있기 때문이다. 또한 크게 구분된 역할도 이름에 반영하는 것이 좋다. 예를 들어, 서로 다른 서브 시스템에 속하는 두개의 다른 모듈은 이름에 해당 서브 시스템의 약자(혹은 전체 이름)가 포함를 포함할 수도 있다. 이름 짓기가 중요한 이유는 전체 시스템의 구성요소들은 고유한 역할과 책임 범위를 가지고 있기 때문이다. 마치 소설책에서 등장인물이 각각 다르고 그들 간의 역할 및 협업으로 이야기를 이끌어가듯이, 시스템의 설계도 구성요소들을 구분해서 역할을 할당하고 그들 간의 협력으로 문제를 해결해 나가기 때문이다.

소프트웨어 설계를 실패하는 경우는 초기부터 너무 상세한 것(Detail)에 집착하는 것이다. 코딩을 잘 못 하는 첫걸음이 최적화에 초기부터 집중하는 것처럼, 설계도 숲을 보지 못하고 나무에 집착하는 것은 좋은 결과를 내기 어렵다. 문서가 잘 정돈(Structured)되었다는 것을 알기 쉬운 방법은 전체 구조를 정리한 목차를 보면 알 수 있듯이, 마찬가지로 좋은 소프트웨어 구조도 어떻게 구성되어 있는지를 한 눈에 쉽게 알아볼 수 있으면 된다. 다른 점이 있다면 문서는 그 자체로 의미를 가지지만, 소프트웨어 구조는 최종적으로 구현이 되어 동작해야 한다는 점이다. 어쨌든 둘 다 언어를 이용해서 구현(혹은 작성)된다는 점은 동일하며, 인간의 지적인 능력을 최대한 활용해야 한다는 사실도 마찬가지다.

소프트웨어 설계는 주로 하향식으로 진행된다. 추상적인 사고에서 출발해서 구체적인 구현으로 나아간다. 이런 과정 중에 입력이 되는 데이터가 있으며, 이를 가공하기 위한 절차(혹은 주체) 및 그 절차에 필요한 구성 요소들이 다시 각각의 추상화 수준에서 추가된다. 하향식의 첫 번째 과정이 완료되면 대략적인 구조가 완성된다. 이 때부터는 구현이 시작된다고 볼 수 있으며, 구현은 상향식으로 진행하는 것이 단위 테스트를 위해서 필요한 노력을 절감해 줄 것이다. 대략적인 시스템의 구성 요소들이 확인된 상황에서 구성요소들간의 관계를 만족하게 해줄 인터페이스(Interface)구현을 하게 되며, 이러한 인터페이스가 만족되는지 확인하기 위한 테스트 케이스의 설계와 테스트를 통과하기 위한 실제 구현이 추가될 것이다. 상향식 방식의 코딩은 상위 구성 요소들을 테스트하기 위해서 필요한 하위 구성 요소들의 인터페이스를 이전 단계에서 구현할 수 있기에 통합 테스트에 유리하다. 물론 그렇다고해서 반드시 상향식으로 구현할 필요는 없으며, 처한 상황이나 현실에 따라 취할 수 있는 방법은 달라질 수 있다.

한 번에 모든 설계를 마치기는 사실상 어렵다. 구현에서 발생하는 피드백을 통해서 잘못된 설계를 수정할 수 있으며, 지속적으로 진화하는(Evolutionary) 것이 설계의 특성이기 때문이다. 사전에 모든 정보가 알려지지 않은 상황에서는 특히 구현 피드백을 얻는 것이 훌륭한 설계를 위해서는 반드시 필요하다. 여러 번의 반복을 통해서 과제를 진행하는 경우, 각 반복의 마지막에 데모와 같은 것을 진행해서 과제와 관련된 사람들에게 꾸준히 좋은 인상을 줄 필요도 있다. 사실 소프트웨어가 보이지 않는 무언가를 만든다고 오해하는 사람들이 많은데, 이런 데모를 통해서 무엇이 되고 무엇이 아직 안 되었는지를 확실히 알려주어야 한다. 설계서도 마찬가지로 얼마나 일이 진척되는지에 대한 한 가지 지표로 제시될 수 있다. 구현되지 않은 부분이 어딘지를 정확히 알려주고 얼마나 검토가 완료되었는지 공감할 수 있다면, 과제의 진척 상황에 대한 가시적인 자료로 사용할 수 있다. 어쨌든 개발자는 뭔가를 지속해서 보여주어야 할 의무가 있다. 가서적으로 보여주는 활동을 통해서 점차 현실의 문제에 다가가는 것처럼 보이기 때문이다.

소프트웨어를 만드는 사람들은 남의 코드를 완전히 이해하기를 원하는 경우가 많다. 하지만 실제로 필요한 것은 내부적인 구현이 어떻게 되어있는가를 아는 것보다, 어떻게 블랙박스(Black Box) 형태로 사용할 것인가이다. 그것이 만족되지 않기에 내부 구조를 파고 들어가게 되는 것이다. 만약 외부로 보이는 행위(Behavior)를 잘 정의된 인터페이스에 대해서 정확히 알 수 있다면 내부적인 구현까지 보고싶지는 않을 것이다. 따라서 될 수 있으면 인터페이스들은 그 역할을 명확히 설명할 수 있어야 하며, 해당 인터페이스를 제공하는 구성 요소가 속한 부분에 어울리는 것이어야 한다. 다시 말하면, 인터페이스의 이름에서 예상할 수 있는 결과를 도출할 수 있어야 할 것이다. 사람은 상상하기를 좋아하지만 자신이 잘 모르는 부분까지 상상해서 구현하는 것은 바람직하지 않다. 설계는 이런 부분들을 반영해서 적절한 제한을 가할 수 있어야 하며, 구현에 너무 많은 자유도를 주는 것은 버그를 유발할 가능성을 높이게 된다.

# [ 소프트웨어 품질에 성능만 있는 것이 아니다. ]

흔히 소프트웨어 품질이 높다고 말할 때 대부분의 경우 "성능"이 좋다는 말로 생각하는 경우가 많다. 하지만, 소프트웨어의 품질에는 성능만 있는 것이 아니다. 오히려 성능이 높은 소프트웨어를 처음부터 짜려고 노력할 때 다른 측면에서 필요한 "품질"이 낮아지는 경향이 있다. 한정된 자원과 시간에 많은 기능 및 비기능 요구사항을 만드는 소프트웨어는 특정 부분만을 강조할 때 나머지는 “정신을 놓게 된다는 것”을 반드시 명심해야 한다. 기본적으로 좋은 구조를 정의하고 이를 바탕으로 전체적인 성능 최적화를 시도하는 것이 부분적인 성능 최적화를 처음부터 시도하는 것보다 더 좋은 결과를 가져올 확률이 높다. “구조적인 코드는 최적화가 가능하지만, 최적화된(?) 코드는 구조화가 불가능”하기 때문이다.

실제로 잘못된 구조로 인해서 발생하는 성능 저하문제는 다양하게 나타나지만, 대표적으로 주어진 일을 처리할 때 결과가 나오는 시간의 산포가 정규분포상에서 넓은 종형태로 나오는 경우다. 다른 말로 표현하면 평균은 비슷하지만, 값들이 넓은 범위에 걸쳐 있어서 흩어지는 것이다. 이런 경우에는 일정 수준의 성능을 지속해서 보여주지 못하고 들쑥날쑥한 결과를 보여주게 되어, 사용자 입장에서 기대하는 것과는 차이가 나게 된다. 좋은 구조로 되어 있다는 것은 “시스템이 항상 일정 수준 이상의 성능을 꾸준히 보여줄 수 있다”는 말로 표현할 수 있다. 중요한 것은 좋은 구조가 먼저 있어야 하며, 그 구조는 구현하고자 하는 소프트웨어가 동작하는 환경에서 가장 일반적으로 제공되는 것들과 잘 융합해야 한다는 점이다.

예를 들어, 이미 쓰레드(Thread)와 같은 것을 지원하는 Linux에서 자신이 만든 태스크(Task)를 실행하기 위해서 또 다른 응용 프로그램 수준의 스케줄러(Scheduler)를 구현하는 것은 효과적이지 않을 수 있다. (이렇게 이야기하는 것은 효과적인 경우도 있기 때문이다. 가령 단순히 몇 개의 함수 호출만을 짧은 시간에 호출할 목적이라면, 자신이 만든 스케줄러가 모든 함수 호출에 대해서 쓰레드를 선언하는 것보다는 나을 수 있기 때문이다.) 만약 스케줄러가 필요한 상황이라면, 쓰레드 단위로 동시 실행이 필요한 부분을 나누고, 쓰레드 간의 통신방법에 대한 일관된 정의를 사용하는 편이 좋을 것이다. 좋은 구조의 다른 예로는 기능의 추가나 변경이 쉬운 구조를 적용하는 것이 있을 수 있다. 새로운 기능을 추가하기 힘들다고 생각되면, 다른 구조를 전면적으로 도입할 것을 검토할 시기가 왔다는 뜻으로도 해석할 수 있다. 물론 이것은 상품 개발팀이 판단하기에는 어려운 일일 수도 있다. 왜냐하면 그들이 당면한 최고의 목표는 제품을 빨리 양산하는 것이기 때문이다. 하지만, 이런 상황이라고 하더라도 결국 좋은 구조를 포기한 대가는 늘어나는 버그로 인한 일정 지연이 될 가능성이 높다.

좋은 구조를 만드는 조직은 우수한 인력(?)으로 구성되어야 한다. 제품 경험도 필요하며 다양한 코딩에 대한 이론도 필요하다. 사용하는 언어에 대한 상세한 특징을 아는 것도 중요하지만, 주로 요구되는 능력은 "인터페이스(Interface)"를 정의할 수 있어야 한다는 것이다. 좋은 이름을 작명하는 방법과 문서화하는 역량등 자기 일에 대해서 완벽함을 추구해야 한다. 자신이 한 코딩이 자신만을 위한 것이 아니며, 다른 사람들을 위해서 필요한 것들도 세심한 배려가 필요하다. 테스트의 중요성을 잘 알고 있어야 하며, 어떻게 자신의 코드를 테스트하는지도 경험적으로 알아야 한다. 소프트웨어의 품질은 성능만이 아니라 결국은 유지보수와 관련된 활동을 얼마나 쉽게 할 수 있도록 만들어 주는가에 달려있다. 구조적인 강건함은 추가되는 기능이 얼마나 국지적인 변경을 발생시키는가에 달려있다. 그리고, 그렇게 추가된 기능을 테스트할 수 있어야 하며, 그 외의 기능들에 대해서도 함께 테스트가 충분히 이루어져야 한다. 따라서, 소프트웨어 품질의 대부분은 어떤 사람을 어떻게 쓸 것인가가 가장 중요한 결정 요소가 된다.

# [ 이식성(Portability)이 좋은 코드 ]

이식성(Portability)은 특정 환경에서 동작하는 소프트웨어를 다른 환경으로 변경해도 동작할 수 있는 능력을 말한다. 다른 운영 환경에서 동작하는 것과 특정 하드웨어에서 동작하던 소프트웨어를 다른 하드웨어로 옮겨도 쉽게 동작시킬 수 있는 것을 포함한다. 보통 이식성이 높다는 말은 소프트웨어가 계층적(Layered)으로 잘 구성되어 있다는 뜻이다. 즉, 계층화를 시켜서 특정 환경에 의존적인 부분을 분리하고 있는 것이다. 이러한 계층화의 핵심은 어떻게 각 계층의 인터페이스를 정의할 것인가이다. 하지만 의존적인 부분이 분리될 수 있는 인터페이스를 정의하는 것은 결코 쉬운 일이 아니다. 환경이 바뀌면 그 환경에서 제공하는 기본적인 인터페이스도 바뀌게 되며 심지어 필요한 인터페이스가 없을 수도 있기 때문이다.

만약 이식성이 높은 코드를 만들고자 한다면 기본적으로 포팅(Porting) 계층을 정의하는 것이 좋다. 즉, 환경에 의존적인 부분들을 다루는 특정 계층을 미리 정의하는 것이다. 그리고, 그 계층보다 상위의 코드들은 전부 포팅 계층이 제공하는 인터페이스만 사용해서 구현해야 한다. 계층을 건너뛰고 직접 접근하려고 하는 것들은 전부 막아야 한다. 만약 하위 계층에서 제공하는 인터페이스가 불충분할 경우에는 새로운 인터페이스를 추가해서 구현하거나, 그것을 사용하는 부분을 다른 제공되는 인터페이스를 사용하도록 유도해야 한다. 포팅 계층의 인터페이스가 모든 것을 다 제공하기 위해서 지나치게 늘어날 가능성이 있는데 이를 제한할 필요도 있다. 즉, 새로운 인터페이스를 추가하기 위해서는 동의할 경우에만 허가하는 것이다. 지속해서 인터페이스를 늘리려는 요구가 있을 때는, 반드시 기존 인터페이스와의 호환성을 검토한 후에 추가하도록 한다.

이식성이 좋은 소프트웨어는 다른 환경으로 전환할 경우 부분적인 수정만을 동반해야 한다. 즉, 국지적인(Local) 변경(구현)에 한정되어야 한다. 이런 저런 부분들을 다 수정한다면 어디를 어떻게 고쳤는지를 알기 어렵다. 그리고, 앞으로 새로운 환경으로 전환할 경우에도 혼선이 빚어질 가능성이 높다. 사용하던 하드웨어의 변경이 있다고 해서 제품의 기능이 갑작스럽게 변경되지는 않는다. 몇 가지 추가적인 기능이 있을 수 있고, 성능 등의 개선이 있을 가능성은 있지만, 주요 사용 시나리오는 변경이 되지 않는 경우가 많다. 따라서, 이런 비즈니스 로직을 실제 환경에 적용하는 것은 큰 문제가 되지 않아야 한다. 실제로도 하드웨어 변경으로 인해서 신규 하드웨어에 소프트웨어를 포팅하는 데 일주일이 채 안 걸릴 수도 있다. 이식성이 높은 코드를 짜는 것은 향후 발생할지도 모르는 환경적인 변화에 대한 대응책이기에 가능한 변경 부위를 줄이려는 노력을 해야 하며, 결국 “구체적인 부분에 의존적인 코드”를 짜는 것은 이런 일을 더디게 할 뿐이다.

소프트웨어는 완전히 새롭게 개발하는 경우는 드물다. 더군다나 상업적으로 이미 작성된 코드의 경우에는 더욱 그러하다. 따라서, 기능 추가에 대한 것은 기능의 분기에 민감한 부분을 찾아서 주요 코드와 분리를 시켜야 하고, 환경에 관련된 부분들도 마찬가지로 분리해서 다루어야 한다. 이런 과정을 지속해서 하면 결국 도달하는 것은 잘 정리된 모듈화된 코드가 될 것이다. 이미 개발된 코드들은 이런 식으로 차츰 개선해 나가는 것이 필요하며, 완전히 새로 개발하는 코드의 경우에는 인터페이스와 구현을 나누어서 정의해야 앞으로 있을 변경에 대한 대비책이 될 수 있다. 물론 이렇게 조금씩 정의되는 모듈들에 대해서 단위 테스트를 함께 구현해 나갈 수 있으면 더 좋다. 왜냐하면 꾸준한 개선을 해도 항상 일정 수준의 품질을 유지해 줄 수 있는 방법이 자동화된 테스트이기 때문이다. 프로그래밍이란 결국 작은 나무들로 이루어진 숲을 함께 바라보는 것이다.

# [ 나만의 도구 상자 ]

인간이 도구를 사용하는 것은 불편함에 대한 욕구를 만족하게 하기 위한 것이다. 물론 인간만이 도구를 사용한다는 것은 아니다. 하지만, 인간이 더 많은 욕구를 가지고 있기에 더 다양한 도구를 사용하는 것은 확실하다. 소프트웨어 개발에 있어서도 도구를 사용하는 것은 그렇지 않은 경우보다 훨씬 생산성이 높다는 것은 이미 잘 알고 있을 것이다. 도구는 이미 소프트웨어 개발자의 일상에 자리 잡고 있으며, 자신이 잘 사용하는 도구들을 바꾸려고 하지 않는다는 점에서 그 중요성을 충분히 짐작할 수 있을 것이다. 하지만, 정작 새로운 도구가 나오고 더 효율적이라고 하더라도 소프트웨어 개발자는 자신이 사용하던 도구를 바꾸려고 하지 않는다. 이는 도구를 불편함을 해결해줄 수 있는 유용한 것에서 더 나아가 자신만의 방식(Style)을 만들어버렸기 때문이다.

하지만, 이런 식으로 굳어진 방식은 문제를 일으킬 수 있다. 왜냐하면, 더 좋은 도구가 나왔을 때 과거의 경험으로 인해서 그것을 부정하는 경향이 생긴다는 것이다. 예를 들어, 편집기로 “VI”를 사용했던 사람이(물론, VI가 나쁘다는 뜻은 아니다.) 이클립스(Eclipse)를 가지고 코딩하는 것을 싫어해서 통합개발 환경을 멀리하는 경우가 있다. 이것을 가지고 강제적으로 모든 개발자에게 적용하는 것은 힘든 일이며, 익숙하지 않은 도구에 대해서 반감을 품고 반항하는 경우도 많이 볼 수 있다. 하지만, 앞에서 이야기했듯이 도구는 도구일 뿐이다. 자신이 사용하는 도구에 자신의 인격이 담기는 것은 아니다. 그리고, 같은 팀에 속한 사람들이 한 가지 도구를 같이 사용하는 것은(생산성이 좋은 도구를) 시너지 효과를 볼 수 있는 부분이 많다. 도구를 사용해서 문제를 해결하는 역량을 서로 협력을 통해서 키울 수 있기 때문이다.

자신이 익숙한 도구들에 대한 리스트를 만들어보면 어떨까? “Visual Studio”와 같은 도구는 Windows 프로그램을 개발할 경우에는 거의 필수와 같이 사용한다. Java를 코딩하는 사람들은 Eclipse를 많이 쓸 것이다. Linux에서 C++로 코딩할 경우에도 Eclipse를 사용할 수도 있고, 다른 툴들을 이용해서 통합된 개발 환경에서 컴파일 및 실행과 디버깅을 시각적으로 하게 할 수 있다. 그리고, 원격 디버깅(Remote Debugging)과 같은 것을 할 경우에도 위에서 제시한 툴들을 이용할 수 있다. (원격 디버깅은 타겟(Target)에 디버깅 대리자(Debugging Agent)를 심어서, 원격에서 프로그램을 실행하면서 디버깅 할 수 있는 것을 의미한다.) 이런 툴들의 List를 하나씩 나열하고 새롭게 개선된 부분이나 추가해야 할 것들을 적다보면, 상당히 많은 것을 자신이 사용한다는 것을 깨닳을 수 있을 것이다. 도구를 잘 사용한다는 것은 자신에게 맞는 적절한 도구를 이용하는 것도 있지만, 남들이 잘 활용하는 도구를 자신도 알아둘 필요가 있다는 것을 알게 될 것이다. 무조건 이것이 아니면 안 된다는 것보다 어떤 도구를 어디에 적절하게 사용해야 하는지를 아는 것이 좋다.

IDE(Integrated Development Environment)을 구축하고, 필요한 것들을 한꺼번에 얻을 수 있다면 상당히 효율적으로 일하는 것임은 틀림없다. 하지만 개발 인프라에 대한 자동화를 생각한다면, 여기서 한 걸음 더 나아가야 한다. 즉, 명령어 라인(Command Line)상에서도 입력을 통해서 컴파일이 가능하고, 각종 만들어진 테스트들을 실행할 수 있도록 만들어주어야 한다. 이것을 하는 이유는 나중에 빌드와 테스트, 배포 등이 자동으로 이루어지기 위한 것이다. 단순 반복적인 작업은 될 수 있으면 자동화를 하는 것이 좋다. 특히, 테스트 스크립트를 실행하는 것은 반드시 자동화를 이룰 수 있어야 한다. 테스트에 대한 자동화에서 얻을 수 있는 가장 큰 부분은 비싼 인건비를 절약한다는 점과 언제 어떠한 변경에 대해서도 검증을(대략적이라도) 할 수 있다는 점이다. 이 부분은 상당히 크다. 하지만, 대부분의 코드는 초기에 이런 것들을 계획하지 않고 시작되기에 중간에 이런 일을 하는 것은 대단히 어렵다. 테스트에 대해서 아무런 대책이 없었기에 근본적인 코드의 변화를 가져올 수 있는 일은 될 수 있으면 안하는 것이 인지상정이기 때문이다. 아무리 좋은 툴도 이것은 해결하지 못하며 다만 조금 도와줄 수 있을 뿐이다.

# [ 세계최고? 세계 최초? ]

예전에 다닌 모회사의 모임원은 이런 말을 좋아했다. "세계 최고입니다.", "세계 최초입니다." 하지만, 그것과 소비자들이 자신의 지갑을 열어서 사는 제품은 다르다. 예를 들어, 카메라의 센서가 탁월하게 좋다고 하더라도, 그 센서가 열이 많이 발생한다면 소비자들이 돈을 내고 살까? 세상에서 가장 높은 해상도를 구현하더라도 제대로 보기 위해서 많은 것들을 설치해야 한다면 그게 무슨 가치가 있을까? 하지만, 모회사의 모임원은 그런 말을 주주 사용했었다. 자신이 마치 모든 것을 다 안다는 듯이 여기저기 떠들고 다니지만, 정말 사용자들이 원하는 것은 모르고 있다. 아마도 기술지상주의를 최고의 목표로 삼고 있는 대부분 개발자도 마찬가지일 것으로 생각한다. 세상에서 최고 좋기 때문에 팔리는 것이 아니라 사용자가 원해서 팔리는 것이다.

먼저 사용자들이 좋아하는 제품은 무조건 이뻐야 한다. 왜냐하면, 사용자는 자신이 남들보다 감각에서 앞선다는 것을 체험하고 싶어 한다. 두 번째는 사용자들이 제품을 사용하기 쉬워야 한다. 이건 사실 아주 어려운 일이다. 제품에 대한 통찰력이 필요하며 불필요한 것들에 대한 과감한 삭제와 숨김이 있어야 하지만, 대부분의 경우는 많은 것을 보여주고 싶어 하기에 실패가 잦다. 여기서 중요한 것은 사용자의 언어로 표현해 주어야 한다는 점이다. 사용자는 개발자만큼 전문가가 아니다. 그리고, 누가 보더라도 매뉴얼을 읽지 않더라도 대부분의 핵심 기능은 사용할 수 있어야 한다. 세 번째는 사용자들은 빠른 반응을 원한다. 따라서, 모든 것이 완료된 후에 뭔가 반응을 보이기보다는 준비되는 데로 반응할 수 있는 것이 좋다. 그리고, 중간에 취소가 있기에 그런 부분에 대한 대응도 일관되게 만들어져야 한다.

세계에서 최고라는 제품들은 많이 있고, 항상 더 좋은 것들은 꾸준히 나온다. 그런 기록은 언제든 깨질 수 있다. 세계 최초의 기능은 사용되지 않을 가능성이 높다. 왜냐하면, 그런 기능들이 꾸준히 등장하고 있지만, 주목을 받는 것은 극히 일부에 지나지 않기 때문이다. 오히려, 기능은 단순화되어야 하고 반드시 있어야 할 기능만이 필요하지, 이런저런 "최고/최초"의 기능들은 제품의 경쟁력에 제한적인 영향만 줄 뿐이다. "최고/최초"라고 외치는 제품들 대부분은 그것을 사용하기 위해서는 복잡한 절차가 있거나 제한된 상황에서만 가능한 경우가 많다. 우리가 만드는 제품은 세상에서 가장 사용자가 필요한 제품이어야지, 기술력을 뽐내거나 개발자의 편의대로 만들어져서는 안 된다. 그리고, 그런 기능이나 성능에 들이는 노력 일부라도 편의성이 높은 제품에 투자한다면, 오히려 최고의 제품은 되지 못하더라도 사용자가 꾸준히 찾는 제품은 충분히 될 수 있을 것이다.

조직에서 최고를 지향하는 것은 나쁘지 않다. 하지만, 그런 것에 모든 것을 투자하는 조직은 나머지를 희생하고 있는 경우가 많다. 제품이 종합예술이라는 것을 안다고 하지만, 대부분의 사람들은 자신이 관심 있는 것에만 집중하고, 나머지 자신이 잘 모르는 부분에 대해서는 그 업적을 편향된 시각으로 바라본다. 높으신 분들은 하드웨어를 만지작거리던 버릇을 버리지 못하고, 소프트웨어도 마치 하드웨어처럼 만들어지기를 원한다. 하지만, 이것은 결국 자신의 발목에 쇠사슬을 채우고 자신의 무지함을 남들에게 광고하는 것밖에 되지 않는다. 소프트웨어는 하드웨어처럼 만들어지지 않으며 만들려고 해서도 안 된다. 소프트웨어는 기술만 있는 것이 아니라, 예술적인 특성(창의적인 성격)도 함께 가지고 있다. 소프트웨어는 눈에 보일 듯하면서도 잡히지 않으며, 손으로 만지작거리기에는 너무 정교해서 쉽게 고장 날 수도 있다. 잘 모른다면 그냥 전문가에게 맡기는 편이 그나마 최선의 방법이다.

# [ 프로그래밍 언어의 선택 ]

요즘은 Android와 iOS와 같은 모바일 프로그래밍이 강세를 떨치고 있다. 돈을 버는 방법으로서도 그렇고, 경험 및 자신의 포트폴리오를 만드는 데도 적합하다고 생각된다. 하지만, 한가지 잊어선 안되는 것이 있다. 그런 모든 시스템도 역시 Assembly와 C/C++언어가 기반이 되고 있다는 사실이다. Java나 Python등이 필요 없다는 이야기가 아니다. 아직은 C/C++과 Assembly 언어들이 많은 시스템에서 사용되고 있다는 말이다. 하나의 프로그래밍 언어가 지배적인 시점은 지났다고 하지만 지나친 치우침은 옳지 못하다. 특히 학생인 경우에는 다양한 언어를 경험하는 것이 앞으로의 진로에도 많은 도움이 되리라고 생각한다. 다양한 프로그래밍 언어를 익히는 것이 어려울수도 있겠지만, 어려울수록 더 많이 자주 해야 한다는 것이 진실이다.

특정 응용 프로그램을 개발하기 위해서는 주변 상황을 정확히 파악하고 있어야 한다. 만들고자 하는 제품에 최적의 언어를 선택하는 것은 중요하다. 또한 만들고자 하는 제품에 탑재될 OS도 선택 가능한 범위가 얼마나 되는지 알아야 한다. 예전의 경우를 돌이켜보면(Linux가 일반화되지 않은 시점) 상품화를 담당하는 사람들은 Linux를 사용하는 것을 배척했었다. 하지만, 지금은 모바일 폰에서 기지국 장비, 캠코더, 디지털 TV, 감시 카메라나 영상 저장장치 등 다양한 부문에서 Linux를 사용하고 있다. 예전에는 Linux에 대해서 안정성이 부족하다거나 익숙하지 않아서(기존에 개발했던 소프트웨어가 이미 있는 상태였기에) RTOS(Real-Time Operatign System: 실시간 운영체제, VxWorks, pSOS+, Nucleus, uC/OS-II등)를 사용하는 것을 선호했다. 메모리 보호 개념은 거의 없고 개발자가 직접 해야 할 부분이 많기는 하지만, 그래도 그걸 편하다고 생각했었다. 주로 사용하는 언어는 당연히 C/C++(사실 C++도 거의 사용안했지만)였다.

중요한 것은 이미 제공하고 있는 것들은 다시 개발할 필요는 없다는 점이다. 하지만 "내가 코딩하지 않은 프로그램이 가장 좋은 프로그램이다"라는 원칙을 지키지 않고, 그냥 다시 개발하는 경우는 흔하다. 회사 내부에서 자신이 개발하지 않았기에 믿을 수 없다는 의심도 팽배했다. (심지어 표준 템플릿 라이브러리를 사용하지 않는 것이 코딩 표준이라고 우기는 사람도 있었다. 이미 표준 템플릿 라이브러리에서 제공하고 있는 기능도 다시 코딩해서 사용했는데, 이유는 다양하지만 결국 성능이 모자라다는 말이 핵심이었다.) 어떻게 보면 이해가 되는 부분도 있지만, 남이 발명한 것을 다시 발명하고 테스트할 필요는 없다는 것이 일반적인 원칙이다. Linux의 커널 스케줄러가 담당하는 쓰레드 스케줄링을 사용하지 않고, 자신이 만든 응용프로그램 수준의 스케줄러를 선호하는 경우도 있었다. 나중에 실제로 두 가지를 놓고 비교했을 때 성능의 개선에 별로 도움이 되지 않았으며, 새로 구현된 스케줄러의 경우 들쑥 날쑥한 성능을 보이는 것(Jitter)으로 확인되었다. 활용할 수 있는 것들은 최대화하고 추가적인 코딩은 최소화하는 것이 시스템을 빨리 안정화시키는 핵심이다. 만약 필요한 것이 더 있다면 문제가 되는 부분만 대체하는 것이 좋은 선택이라는 것이다. 언어적인 도움을 이용해서 최적화하는 것도 마찬가지로 “좋은 구조”라는 바탕이 필요하다. 사실 최적화는 끝없는 노력의 과정으로 시간과 인력을 어떻게 활용하는가에 달린 문제일 뿐이다. 최적화된 코드는 구조화하기 어렵지만, 구조화된 코드는 최적화도 더 쉽다.

프로그래밍 언어를 선택하는 것은 응용에 가장 최적인 것으로 해야 한다. "최적"이라는 말은 사용자 경험에 의존적인 것으로, 개발자가 임의로 판단해서는 안 된다. PHP가 Web에 많이 사용된다고 해서 최적의 사용자 경험을 제공하는 것은 아니다. 빠른 반응을 요구하는 사용자 응용 프로그램의 경우에는 C/C++로 구현한 CGI(Common Gateway Interface)를 사용할 수도 있다. 차별화된 제품을 개발하기 위해서는 대부분의 사람이 선택하지 않는 길로도 갈 수 있다. 이때 중요한 점은 확장성과 변경 용이성, 사용성, 사용자 반응성 등에 대한 트레이드오프(Trade-Off)가 어떻게 되는지를 정확히 이해해야 한다. 최적화가 어렵다면 적절한 해법을 간단히 구현하는 것도 한 가지 방법이다. 실무에서 부딪히는 문제는 다양한 해법이 존재할 수 있다. 프로그래밍 언어가 훌륭한 해결책을 제시할 수도 있고 간단한 설계가 그 해법일 수 있다. 한가지 문제를 다양한 시각으로 볼 수 있을 때만 문제에 대한 해결책을 제대로 내놓을 수 있는 것이다. 무슨 문제인지도 모르는 상황에서 너무 많은 결정을 급하게 내리는 잘못을 범하지 않아야 한다. 돌이킬 수 없는 결정은 과제를 돌이킬 수 없는 상황으로 몰고갈 가능성이 크기 때문이다.

# [ 코드 개선에 대해서 ]

소프트웨어 개발자로서 일한다면 "기존의 코드(Legacy Code)"에 대한 불만과 그것을 개선하는 것이 어렵다는 절망감을 항상 느낄 것이다. 그리고, 대부분의 경우 관련된 문서가 없다거나 코드 내에 쓸모없는 주석만 가득하다는 사실도 부담으로 받아들일 것이다. 놀라운 점은 그런 불평을 말하는 사람의 코드 조차도 나중에 다른 사람이 보고 똑같은 말을 되풀이한다는 것이다. "도대체 어떤 놈이 짠 거야?"라고 이야기를 하다가도 돌아서선 부지불식간에 익힌 "잘못된" 지식으로 같은 일을 반복하고 있는 것이다. 이런 문제들을 개선하겠다는 생각이 있다면 조금씩 나아가는 방법을 선택해야 한다. 한 번에 완전히 바꾸겠다고 덤벼들면 크게 잃게 될 가능성이 높기 때문이다.

"보이스카웃(Boy Scout)"원칙이란 것이 있다. 무슨 뜻이냐면 캠핑장을 자신들이 도착했을 때보다 조금 더 깨끗하게 정리하고 떠난다는 것이다. 그렇게 조금씩 개선해야 "Legacy Code"를 변화시킬 수 있다. 이때 필요한 방법은 역시 테스트다. 즉, 개선하기 전과 개선 후에 동일한 동작을 보장해야 한다. 그러기 위해서는 변경 전의 동작을 알아야 하며 변경 후에도 그것이 보장되는지 검증해야 한다. 사람이 손으로 일일이 그런 일을 한다는 것은 불가능하며, 새로 코드를 작성하는 편이 더 옳을지도 모른다. 따라서, 기존의 안정된 코드를 반드시 사용하겠다는 결정이 있다면, 제일 먼저 해야 할 일은 외부적(External)으로 보이는 동작들에 대한 검증 방법을 만들어야 한다. 사실 이 부분이 쉽지 않다. 외부적으로 보이는 행동은 동일한 입력에 대해서 동일한 결과를 출력해야 한다는 사실에 기반하며, 각각의 모듈로 구분되지 않은 코드는 전체를 한번에 테스트할 수밖에 없기 때문이다.

잘 아는 사실이겠지만 소위 말하는 "레거시 코드(Legacy Code)"들의 특징은 복잡하다는 것이다. 그리고, 그런 복잡함과 더불어 이해하기 힘든 많은 "매직 넘버(Magic Number)"들이 존재하며, 당연하겠지만 모듈의 개념도 없이 마구잡이로 짜인 코드들이 많다. 이런 것들을 하나씩 검증하는 일은 결코 쉬운 일이 아니다. 어쩌면 정말 완전히 새로 짜는편이 더 빠른 방법일 수 있다. 하지만, 그렇다고 완전히 방법이 없는 것은 아니다. 가장 쉬운 방법은 "Legacy Code" 전체를 하나로 그냥 묶어서 일단은 테스트 케이스를 만드는 것이다. 즉, “Legacy Code”가 외부에 제공하는 인터페이스를 이용해서 상위 계층(물론, 상위 계층이라는 개념이 불분명한 경우도 있지만)에서 입력의 종류와 출력의 종류를 다 찾아낸다. 그리고나서, “Legacy Code”의 계층을 분석해서 계층 간의 역 호출을 일일이 제거한다. 이 때는 해당 호출이 필요한 부분들에 대해서 함수를 옮긴다거나(Move Function), 끄집어내서(Extract) 원래 임무를 수행해야하는 모듈이나 파일로 분리한다. 이 때도 한 번에 많은 부분을 수정하기 보다는 하나씩 진행해야 한다. 앞에서 얻은 입출력 값을 이용해서 테스트 스크립트를 만들고 지속해서 동일함을 검증해야 한다.

필요하다면 "Legacy Code" 덩어리가 의존하는(호출하는) 모듈들에 대해서 “Dummy”들을 만들어 줄 수도 있어야 한다. 즉, 상하에서 "Legacy Code"를 고립시키는 전략이다. (최악의 경우에는 전체 코드가 한 덩어리일 가능성도 있다.) 의존하는 모듈들에 대해서는 “Mock”과 같은 단위 테스트를 위한 도구를 사용할 수도 있을 것이다. 너무 많은 “Dummy”가 필요하다면, 한 번에 하나씩만 추가해서 진행하도록 한다. 즉, 한 번에 하나의 호출 경로(Call Path)만 처리하는 것이다. 이런 식으로 조금씩 불려 나가다 보면 결국 "Legacy Code" 덩어리는 완전히 원래의 코드에서 분리되어 고립될 것이다. 이제는 "Legacy Code" 자체를 공략할 차례다. 공략하는 방법은 가장 호출이 빈번하게 일어나는 코드부터 시작하는 것이 좋다. 왜냐하면, 그만큼 많은 테스트 케이스를 확보해서 나중에 활용할 가능성이 높기 때문이다. 빈번한 호출이 일어나지 않는 모듈들은 분리되기도 쉬울 가능성이 높기 때문이다. 어쨌든 중요한 부분을 먼저 쳐나가는 전략이 필요하다고 본다. 이 때도 역시 모듈 간의 상하 관계를 정의하기 위해서 분석 툴을 이용해서 호출이 어떤 식으로 발생하는지를 눈여겨봐야 한다.

핵심은 의존성(Coupling)을 지속해서 줄여나가는 것이다. 그리고, 그렇게 줄어드는 의존성에 추가로 응집성(Cohesion)을 곁들여야 한다. 함수의 길이 및 파라미터의 개수를 줄여야 하며, 함수가 지나치게 많은 일을 하는 것을 막아야 한다. 이렇게 만들어진 함수들을 다시 역할별로 분류하고 새로운 이름을 붙여주어야 하며, 분리된 각각의 함수에 필요한 테스트 케이스도 추가해야 한다. 한꺼번에 너무 많은 일을 하기보다는 한 번에 하나씩 꾸준히 진행하는 것이 좋다. 함수가 점차 개선되면 그것을 사용하는 함수들도 다시 정리되어 갈 것이다. 이 때는 더 큰 범위의 테스트 케이스가 필요하게 되며 하위에서 상위로의 개선이 발생할 것이다. 상위에서 하위로의 개선은 호출되는 함수들에 대한 “Dummy”들을 만들어 넣거나, 앞에서 이야기한 방법으로 입력과 출력값을 조사해서 그것으로 대체하는 방법을 사용할 수도 있다. 다만 호출되는 함수의 수가 갑작스럽게 증가할 경우에는 하나의 함수를 개선하는데 추가적인 노력이 일반적인 함수보다 많이 들 것이다. 어쨌든 이것도 지나쳐야 할 관문으로 생각하고 원했던 출력이 올바르게 나오는지를 계속 테스트를 늘려가면서 실행해야 한다.

소프트웨어는 지속해서 유지보수를 필요로 하는 까다로운 놈이다. 잠시 게을리하는 순간 보란 듯이 소프트웨어 개발자를 욕 먹인다. 또한 잘 만들어진 소프트웨어 일수록 더 많은 유지보수가 필요하다. 왜냐하면, 그런 소프트웨어 일수록 사용자(고객)의 요구가 더 많이 들어올 가능성이 높기 때문이다. 잘 못 만들어진 소프트웨어는 시장에서 도태되고 유지보수도 할 필요가 없을 것이다. 어쨌든 훌륭한 개발자는 그런 가능성에 대해서 열린 태도를 항상 가져야 한다. 누군가가 원해서가 아니라 자신의 만족을 위해서도 필요하다. 그렇다고 무조건 고치라는 것은 아니다. 변화에도 전략이 필요하다. 혼자 만든 것이 아니라 상업적으로 이익을 나누어야 할 소프트웨어는 시장이라는 높은 벽을 무시할 수 없다. 시장은 기다려주지도 않거니와 비싼 제품에 대해서는 기대하는 바도 크다. 그런 것들을 만족하게 하지 않을 때는 바로 찾아보기 어려워지는 것도 우리가 하는 일의 숙명인지도 모른다.

# [ 고품질의 소프트웨어 만들기 ]

품질을 강조하는 것은 어느 회사나 마찬가지다. 하지만, 강조하는 말로 끝나는 경우와 강조하는 시스템을 구축하는 경우, 그리고, 강조하는 문화를 만드는 경우는 각각 성과를 달리한다. 결론적으로 이야기하자면 가장 쉬운 것은 말이며, 그 다음으로는 돈을 들여서 구축하는 시스템이고, 가장 효과적인 것은 품질에 대한 내부 문화를 만드는 것이다. 말은 누구나 할 수 있지만 투자에는 인색하며, 그렇게 투자를 하더라도 제대로 된 일상에 대한 적용이 없다면 변화는 일어나지 않는다. 이것은 우리가 개인이 아닌 조직으로 일하기 때문이다. 한 개인이 바뀌는 것은 어렵지 않으나, 개인을 넘어서 팀, 혹은 전체 조직으로 퍼져나가는 것은 오랜 시간과 비용이 들 수 있다. 대부분 개발자와 이야기하면 자신이 다니는 회사의 문제들을 잘 알고 있으면서도 "이래서 안돼. 저래서 안돼"하고는 그냥 말의 수준에서 머문다.

변화하지 않는다면 더 이상의 개선은 없다. 그렇다고 변화만을 고집하면, 맨날 변경만 하다가 시간을 다 보낸다. 체계적인 변화를 해야 한다는 것이다. 시작은 말로 하는 것이 좋다. 주로 높은 직급이나 혹은 권한이 있는 사람이 먼저 말을 꺼내는 것이 좋다. 진짜 문제가 무엇인지를 이야기하는 것으로부터 시작해야 한다. 가장 힘든 것은 솔직하게 이야기하는 것이다. 문제가 무엇인지를 감추어서 잠깐의 시간을 벌었다고 생각할지는 모르겠지만, 결국 그 피해는 온전히 모두가 다 감내해야 한다. 굳이 자신이 아니더라도 회사 내의 다른 누군가는 그러한 비용을 다 갚아 나가고 있을 것이 분명하다. 이기적인 생각으로 위기를 모면하고 위로 올라가면 될 것으로 생각한다면 오산이다. 왜냐하면, 결국 그렇게 저질러 놓은 것들은 은행 복리 이자보다 더 많은 부채로 나중에 다가올 것이기 때문이다. 잘 나갈 때는 무시할 수 있을 것처럼 보이겠지만, 힘든 시기에는 그 빚으로 인해서 원금도 다 깎아 먹을 것이기 때문이다.

말을 이어서 시스템의 구축은 객관성과 효율성을 높이는 방법이다. 사람의 기억은 오래갈 수 있을지 몰라도 왜곡이라는 것이 끼어들 수 있다. 그리고, 복잡한 모든 것을 기억하는 능력보다는 추상화시켜 요점을 집어내는 데 적합하도록 사람은 학습되어 있다. 중요한 것은 사람에 대한 비난을 모면할 수 있다는 점에서도 시스템의 구축은 반드시 되어야 한다. 복잡도(Complexity)나 중복코드(Duplicate Code), 잘못된 호출 관계(Reverse Call, Skip Call) 등은 쉽게 도구를 활용할 수 있는 분야다. 지속적인 통합(Continuous Integration)이나 자동화된 테스트 등도 충분히 가능하다. 정적분석이나 단순한 컴파일을 가지고도 많은 버그를 개선할 수 있다. 기계적으로 확인되는 버그들은 모두 고치는 것이 당연한 일이며 기본이다. 나머지 어려운 잡기 힘든 버그에 사람의 소중한 시간을 써야 한다. 가장 중요한 자원은 사람이며 그들의 노동 생산성을 극대화하는 것은 단순한 버그 찾기보다는 어려운 논리적인 문제를 풀어나가는 것이어야 한다. 도구는 결과를 객관화를 시켜준다는 측면에서 개인의 편향을 제거해 논란을 불러 일으킬 가능성을 줄여준다.

시스템의 구축은 개발자들이 귀찮아하는 것 중에 하나다. 개발자는 일을 체계적으로 하는 것을 좋아한다고 이야기는 하지만, 그렇게 할 시간이 없다는 말과 함께(실제로도 힘들지만) 시스템을 무시하는 경향을 가지고 있다. 시스템적으로 일하는 것이 왠지 자신이 가진 역량을 제대로 발휘하는 데 도움이 안되며, 자신이 만든 코드에 문제가 있다고 이야기하면 자존심이 상한다고 느낀다. 정작 중요한 것은 자신이 만드는 소프트웨어가 남들에게 도움을 주기 위해서라는 목적은 어디에 갔는지 찾지도 못하면서, 전혀 도움이 되지 않는 코딩 방법을 고수하는 경우도 종종 있다. 회사를 오래 다니고 싶다고 그렇게 해선 안 된다. 이미 이야기했듯이 오래 다닐 수 있는 회사는 그런 코딩이 걸림돌이 될 것이기 때문이다. 그런 회사에 다닌다면 이미 회사의 비전(Vision)이 사라지고 중노동만이 남은 곳이다. 당신이 떠나기 싫어도 회사는 (망해서) 당신을 떠나게 만들어 줄 것이 분명하다.

고품질의 소프트웨어를 개발하는 마지막 단계는 소프트웨어 개발자 개인의 문화이며, 그러한 품질을 중요하게 생각하는 문화가 형성되도록 하는 것이 관리자의 역할이다. 더 좋은 코드는 나만을 위한 것이 아니며 제품의 사용자부터 제품에 관련된 모든 사람을 행복하게 만드는 것이기 때문이다. 문화는 선택해야 할 경우 공유되는 가치가 높은 것이 우선되도록 만드는 힘이 있다. 고유한 문화를 가지고 있다는 말은 다른 조직과 달르다는 것을 인식하게 만들며, 품질이 우선하는 문화는 작은 결정들의 연속 과정인 코딩에서도 차별화된 경쟁력을 만들 수 있을 것이다. 작은 선택이 쌓이면 큰 결과를 만들게 되며, 잘못된 작은 선택들은 큰 실패로 끝날 수밖에 없다. 물론 당장은 작은 성공을 맞볼 수도 있을 것이다. 하지만, “행복한 시간”은 그리 오래가지 않을 것이고, 남은 시간동안 크고 작은 버그로 인해서 많은 비용을 지불해야 할 것이다. 가장 큰 비용는 더 좋은 것을 찾지못하고 사소한 것들에 개발자 자신의 인생을 낭비하는 것이다.

# [ 프로그래밍 배우기 ]

소프트웨어 개발자들은 어떻게 코딩하는 방법을 배울까? 아마도 대부분의 경우 학교의 수업 시간에 지정된 도서나 강의, 숙제 등을 통해서 배울 것이다. 그것만으로 만족하지 못하면 서점의 여기 저기를 둘러볼 것이고, 인터넷에 나온 서적에 대한 리뷰도 보고, 다른 친구들은 어떤 책으로 배우는지도 확인할 것이다. 물론 고전(Classics)은 어느 분야건 상관없이 항상 있다. 하지만, 고전 책으로는 배울 수 있는 것은 기본적인 것들이라고 생각하고, 그냥 세상에 나온 책 중에서 베스트 셀러 코너에 있는 것을 사는 게 일반적이다. 일반화된 지식이 의미 없다는 것이 아니라 지향하는 바가 다른 것을 엉뚱한 곳에 사용하는 것이 문제다.

대부분의 프로그래밍 언어 책들은 문법에 대한 상세한 설명과 다양한 사용 예를 가지고 있지만, 프로그래밍의 원리나 좋은 코드를 만드는 방법에 대해서는 점처럼 자주 다루지 않는다. 이런 책들은 대부분 초급 프로그래머에게는 적당한 내용이지만 중급이나 고급 프로그래머가 보기에는 부족하다. 초급 프로그래머로서 자신이 활용할 수 있는 코딩 기법들은 대부분 초급 프로그래머가 보는 책에 나오는 것들을 베껴서 사용하거나 인터넷에 누군가가 작성한 간단한 샘플 예제에서 가져온 것들이다. 조금 더 나은 편이라고 한다면 오픈소스로 공동체가 공유하는 코드를 이용하기도 한다. 상업적인 코딩에 초급 수준의 책에서 나오는 코딩 방법을 사용한다면 당연히 결과는 좋지 않을 것이다. 대부분의 예제는 1,000라인이 안되지만 제품에 들어가는 코드는 수십만에서 수백만 라인에 이르기도 한다. 학교 숙제를 하던 수준의 코딩으로는 남을 위해서 돈을 받으며 개발하는데 한계가 있을 수밖에 없다.

우린 좋은 소프트웨어를 만드는 방법을 반드시 찾아야 한다. 아니, 이미 많은 사람이 그런 방법에 대해서 찾았다고 생각하고 있고, 그런 것들을 잘 정리된 책이나 인터넷 사이트에 이미 만들어 두었다. 다만 그런 책을 살 수 있는 돈은 있지만, 읽어서 얻은 지식을 자신의 실무에 적용할 시간이 없다. 정확히 말한다면 실무에 적용하기 위한 시간을 그냥 내버려 두는 회사는 거의 없다. 회사는 항상 빠른 결과물을 요구하고, 그것도 터무니없는 일정으로 만들라고 지시한다. 빨리 가기 위한 체계적인 방법을 거부하고 걷는 연습도 없이 뛰라고 한다. 수영을 강이나 바다에서 혼자서 배운 사람들과 수영장에서 코치를 통해서 배운 사람 중 누가 더 빨리 멀리 갈 수 있는지는 모든 사람이 다 아는 사실이다. 하지만, 이상하게도 소프트웨어 개발에서는 그런 것들은 아무 의미를 지니지 못한다. 누구도 처음부터 코딩을 잘하는 사람으로 태어나지 않는다는 사실을 왜 받아들이지 못하는지 정말 이해하기 힘들다. 코딩도 이론과 경험이 필요하다는 사실을 잊어서는 안될 것이다.

우리는 코딩에 대한 기초를 배울 때 반드시 좋은 소프트웨어가 가져야 할 특성(Attribute)들에 대해 확고하게 이해하고 있어야 한다. 그렇지 않다면 "소프트웨어의 위기(Software Crisis)"라는 말을 또다른 10년이 지나도 다시 들어야 할지도 모른다. 모든 시작은 잘못된 첫 단추에 있을 가능성이 높다. 현실의 장벽도 이미 동작하고 있는 "낡은 코드(Legacy Code)"에 대한 개선이나 기능추가다. 이미 누군가 잘못 코딩한 것을 버리지 못하고 지속해서 기능을 추가하기 바쁘다. 이미 누더기가 된 코드를 걸레로 만들고, 나중에는 너무 더러워서 씻지도 못하는 상태가 될 때까지 반복적으로 사용하게 될 것이다. 이 때쯤이면 유지보수 비용이 다시 개발하는 비용보다 많이 들지만 남아 있는 인력 중에는 완전히 새로 만들만한 역량 있는 사람들은 없을 것이다. 이미 힘든 경험을 한 상태로 "Burnout"되어 회사를 떠났을 가능성이 높다.

프로그래밍을 공부할 때도 다른 과목과 마찬가지로 원리에 대한 이해가 선행해야 한다. 빨리 만든다고 좋은 것이 아니며, 어떻게 만들어야 하는지 정확히 이해해야 한다. 누군가를 가르칠 때도 원리와 그것을 구현하기 위한 “좋은 방법”을 함께 이야기해야 한다. 초급 개발자들이 보는 책들이라고 해서 이런 것들을 이야기하지 않을 이유는 없다. 좋은 책은 오랫동안 널리 읽히는 책이다. 고전이라고 알려진 책들은 반드시 읽히는 이유가 있으며, 왜 보아야 하는지도 이유가 명확하다. 먼저 회사에 들어왔다고 선배가 되는 것이 아니라, 무엇이 올바른 방법인지 먼저 알아서 나중에 들어올 사람들이 실수를 저지르지 않도록 해야 제대로 대접받을 수 있을 것이다. 코딩의 핵심원리에 대해서 이야기했다면 예제들은 반드시 그것을 지키고 있다는 것을 보여주어야 한다. 개발자들은 활용할 수 있는 코드를 알고 싶어하며, 자신이 본 것을 따라하는 것으로 배움을 시작하기 때문이다.

# [ 간결함(Simplicity)에 대한 단상 ]

소프트웨어가 실패하는 주요 원인 중 하나가 “복잡하다는 것”이다. 소프트웨어는 필연적으로 복잡할 수밖에 없다. 소프트웨어를 개발할 때 코딩을 배우는 과정을 언어(Language)를 배운다고 한다. 언어는 문화를 바탕으로 하며 표현할 수 있는 한계를 가진다. 그리고, 같은 언어라고 할지라도 표현하는 방법에 따라 모호함을 가질 수도 있다. 또한 정리되지 않은 생각으로 책을 써가면 제대로 된 책을 만들 수 없듯이, 정리되지 않은 설계를 구현으로 가져가면 문제가 생기기 마련이다. 복잡함은 다양한 해결방안이 가능한 경우에 생길 수 있으며, 소프트웨어 개발도 마찬가지로 문제에 대한 다양한 해결책을 언어로 표현하는 것이다. 따라서, 좋은 소프트웨어를 개발하기 위해서는 단순히 문제를 푸는 것 이상으로 해야 할 부분이 존재하며, 이를 어떻게 문제의 풀이 과정에 적용해 나가는지에 따라 수준이 다른 개발자가 된다.

사물에 존재하는 복잡함은 단지 표면적으로 보이는 복잡함이며, "심오한/깊은 간결함(Deep Simplicity)"으로부터 나온다고 한다. 철학적으로 보일런지는 모르지만, 요점은 "복잡한 문제를 해결하는 데 있어서 복잡한 해결책을 만들 필요는 없다는 것"이다. 쉽게 구현하는 방법이야말로 궁극적으로 소프트웨어 개발자가 추구해야 할 목표라는 뜻이다. 코드가 길어질수록 기능이 많아질수록 점점 더 많은 버그가 발생할 확률은 높아진다. 그리고, 생각하는 단위가 커질수록 버그가 끼어들(스며들) 가능성도 커진다. 따라서, 집중해야 할 부분은 작고, 한 가지 일만 하는 코드를 만들어야 한다는 점이다. 한 가지 일을 하는 코드가 짧아질 가능성이 높은 코드다. 여러 가지 일을 복합적으로 처리하는 코드는 필연적으로 길어지게 되며, 다른 부분의 코드와 연결되는 부분도 늘어나게 된다. 또한 다양한 파라미터(Parameter)들로 인해서 전달하거나 전달받는 정보의 종류(기억해야 할 정보)도 늘어나게 된다. 상호 작용이 커질수록 코드는 변경에 유연하지 못하게 되며, 결국 복잡함을 복잡함으로 해결하려는 노력으로 받아들여질 수밖에 없다.

"완벽하다는 것은 더할 것이 없다는 것이 아니라 더 뺄 것이 없다는 것이다."라는 말을 들어봤을 것이다. 완벽하기 위해서는 필요없는 군더더기들이 없다는 뜻이지, 모든 것을 다 할 수 있다는 것이 결코 아니다. 제품을 고를 때도 모든 것들 다할 수 있는 제품보다는 한 가지 일만 충실히 하는 제품이 고장 날 확률이 낮다. 따라서, 항상 늘어나는 기능들에 대해서는 주의해야 한다. 정말 필요한 기능인지 따져야 하며, 그 기능으로 인해서 변경되어야 할 부분들에 대해서는 명확한 정의가 필요하다. 하지만, 필요 없는 기능이라고 단정할 수 있는 근거는 무엇일까? 그리고, 그 기능을 제외했을 경우에 누가 가장 큰 영향을 받는가는 반드시 파악되어야 한다. 필요없는 기능은 고객에게 가치를 전달하지 못하는 것들이다. 고객의 가치를 알지 못한다고 이야기할지도 모르지만, 대체로 기대하는 가치는 기대하는 가격과 비슷한 의미를 가진다. 회사에게 수익이 된다는 말은 고객이 많이 사용한다는 의미로 생각해야 할 것이다. 직접적인 고객을 찾기 힘들다면 수익이 큰 기능순으로 구현하면 된다. 어쨌든 소프트웨어 개발자의 의무는 고객의 요구(Needs)를 만족하게 하는 품질 좋은 소프트웨어를 만드는 것이기 때문이다.

될 수 있으면 작게 만들고 작게 고치고 작게 증가시켜서 많은 테스트를 해야 한다. 변경은 테스트를 동반하며, 작게 만들면 생각의 범위가 줄어들어 집중하게 되고, 작게 고치면 실수를 줄일 수 있으며, 작게 증가시키고 많은 테스트를 하면 버그의 범위도 한정되어 빨리 수정할 수 있기 때문이다. 테스트는 될 수 있으면 많이 해야 한다. 많이 하기 위해서 자동화는 필수이며, 통합과 동시에 모든 테스트가 자동으로 실행될 수 있어야 한다. 이렇게 만들어진 코드에서도 간결함은 문제 해결책의 "깊은 곳"에 있지만, 그 간결함을 바탕으로 만들어진 "피상적인 복잡함"은 소프트웨어 개발자의 관리 한계를 넘어서지 않을 것이다. 필요한 기능만을 가지고 언제든 쉽게 고쳐질 수 있는 이상적인 코드를 가지기는 쉽지 않기에, 끝없는 개선을 해야 한다. “장인의 손길은 세세함에 머문다"는 말이 있듯이, 간결하지만 세세함(Detail)을 다루는 기술을 반복적으로 끊임없이 익혀야 한다.

# [ 유지보수에 대한 이야기 ]

소프트웨어 개발은 실행 가능한 소프트웨어를 사용자에게 배포(Release)하는 것으로 끝나지 않는다. 소프트웨어 개발은 배포 후에도 지속적인 개선이 필요한 일이며, 주로 새로운 기능의 추가나 기존 기능에서 발생하는 오류를 수정하는데 많은 시간을 보낸다. 대략 배포 후에 들어가는 비용이 전체 소프트웨어의 생명주기에 필요한 비용의 40%에서 80%까지 들어갈 수 있으며, 이를 지속해서 하기 위해선 새로운 제품의 개발조직과는 별개의 유지보수 조직을 운영할 필요도 있다. 왜냐하면, 여러 과제를 동시에 진행하는 조직은 모든 과제를 제대로 하지 못할 가능성이 높기 때문이다. 충분한 인력을 갖추지 못한 현실 세계에서는 신제품 개발과 유지보수를 병행하는 조직들이 많으며, 유지보수로 인해서 새로운 제품을 제 때에 만들어내지 못하고 점점 시장에서 뒤쳐지는 현상이 발생한다.

별도의 유지보수 조직을 가지고 있더라도 대부분의 경우 유지보수 업무를 시시하게 생각한다. 능력이 다른 사람보다 못하다고 판단되는 사람들에게 유지보수 업무가 가는 경우가 많다. 하지만, 원래의 코드를 분석해서 문제를 해결하는 일은 오히려 새로운 개발을 하는 일보다 더 어려운 일이며, 소프트웨어 개발자라면 다들 잘 알고 있을 것이다. 가장 힘든 일은 "남의 코드를 보고 이해하는 일"이다. 남의 코드를 보는 것을 싫어하며 그냥 새로 짜는 것을 오히려 즐긴다. 이유는 간단하다. "남이 짠 코드가 이해하기 힘들다."는 것이다. 하지만, 그렇다고 자신이 짠 코드를 다른 사람도 이해하기가 쉬울까? 모두 다 남 탓을 한다면 지금 이해하기 어려운 코드는 도대체 누가 만든 것일까? “누워서 침 뱉기” 밖에 되지 않는다. 원인이 되는 것을 계속 찾아가다 보면 도달하는 결론은 "열심히는 했지만 잘하지는 못했다"이다. 원인을 개발 팀 외부에서 찾는다면 “일정의 압박”이 가장 큰 부분일 것이다. "급할수록 돌아가야 한다."나 "Go Slow to Go Fast."와 같은 속담에서 알 수 있듯이, 정작 중요한 일에는 노력(Effort)을 들이지 않고 문제가 발생했을 때 고치느라 정신없이 시간을 다 보내고 있는 것이다.

문제가 발생하는 곳은 언제나 과제의 시작 시점과 가깝다. 유지보수는 과제의 제일 마지막 단계에 속한다. 따라서, 시작에서 잘못된 것들을 마지막 단계에서 수정하느라 고생한다는 말이다. 문제의 발생과 문제의 해결이라는 간격을 좁히면 좁힐수록 빨리 수정할 수 있다. 따라서, 문제가 발생한 순간에 고치는 게 원칙이다. 문제를 확인하였지만 고치지 않고 넘어가는 경우가 있을까? 아마도 제대로 된 개발자라면 그렇지 않을 것이다. 하지만, 대부분 그게 문제가 되는지 몰라서(혹은, 왜 문제가 일어나는지 이유를 알지 못해서) 내버려 두고 넘어간다. 문제가 된다는 것을 확인하는 방법이 없다는 것이다. 이를 해결하기 위해서는 개발에서 반드시 자신의 코드를 검증할 수 있는 체계를 갖추어야 한다. 컴파일, 통합 빌드, 정적분석, 코딩 규칙 검사, 코드리뷰, 단위테스트 등이 그냥 있는 게 아니라, 바로 이런 문제를 발견하려는 방법들이다. 문제의 해결을 다음 단계로 지연시키는 것이 아니라, 문제의 발견을 문제가 만들어지는 시점(코딩하는 시점)에서 찾아내 그것을 해결하도록 만드는 데 도움을 주기 위함이다. 따라서, 이런 활동들이 무의미하고 시간이 많이 드는 일이라고 간과하고 넘어간다면, 과제의 후반이 어떻게 진행될지는 "안봐도 비디오"일 수밖에 없다.

유지보수도 마찬가지다. 해야할 일을 줄이기 위해서는 과제 초반부터 많은 것들이 고려되어야 하고 지속해서 검증되어야 한다. 유지보수 활동이 제대로 된 부가가치를 높이는 일이어야 우수한 개발자들이 투입될 수 있다. 단순히 남이 만든 문제만 찾아서 해결하는 끝없는 야근은 아무도 책임지고 싶은 일이 아니다. 코딩의 수준을 높이지 않으면 유지보수의 수준이 낮아진다. 코드의 품질을 높이지 않으면 유지보수 담당자들의 삶의 질이 낮아진다. 코드가 제대로 검증되지 않으면, 회사의 실적도 예상할 수 없는 수준이 될 것이다. 아무 상관이 없는 것처럼 보일지라도 하나 하나의 과정들이 충족되어야만 전체적인 개선이 일어날 수 있다. 소프트웨어의 개발과정은 이런 필요한 일을 충족시키는 활동이며, 높은 수준의 품질을 가지는 제품을 만드는 것이 궁극적인 목표다. 버그투성이의 검증도 안 된 제품을 만드는 것은 소프트웨어 개발자의 자긍심을 낮추는 행위이며, 이를 조장하는 관리자가 있다면 개인의 욕구만 만족시키고 조직의 전체적인 만족도를 낮추고 있다는 것을 알아야 한다. 먼 길을 준비도 없이 무작정 떠난다면, 가는 도중에 다시 원점으로 돌아와야만 할 순간이 반드시 올 것이다. 유지보수는 힘들게 돌아온 바로 그 원점에서부터 다시 시작하게 될 것이다.

# [ 버그가 많이 발생했다. ]

모든 소프트웨어에는 아직 찾지 못한 버그(Bug)가 있다. 다만 그것이 발생하는 확률이 다를 뿐이다. 소프트웨어 개발에서 가장 어려운 부분은 그런 버그들을 찾아서 고치는 일이다. 버그를 줄이는 방법이나 활동들은 다양하게 있지만, 문제는 대부분의 경우 그런 방법들을 너무 늦게 사용하기 시작한다는 점이다. 버그가 발생한 시점에서 멀어질수록 버그를 찾는데 들어가는 시간과 비용은 커진다. 따라서, 그것을 수정하는 비용도 당연히 커진다. 비용은 곧 시간이다. 과제의 지연은 버그의 발생빈도가 높아갈수록 당연한 것으로 받아들여야 한다. 그럼 왜 버그를 발견하고 수정하는 활동들이 뒤로 밀려나야만 하는 걸까?? 이유는 단순하다. 아직도 우리가 소프트웨어 개발의 역사가 가지는 오랜 경험을 무시하고 있기 때문이다. 소프트웨어 공학적으로 프로세스를 강화한다고 생산성이 높은 조직이 되는 것은 결코 아니다. 공학적인 틀(Frame)의 프로세스들은 말 그대로 "운용의 묘"를 살려 틀을 바탕으로 창의적인 결과물 만들어내는 과정을 가려서는 안되기 때문이다.

요구사항, 분석, 설계 등의 과정에서 발생하는 문제들의 수정비용은 상대적으로 적다. 그리고, 그런 곳에서 발생하는 오류에 대한 것들은 즉시 찾아내기도 쉽다. 따라서, 과제 관련자들이 모여서 리뷰를 하는 것이 가장 확실하게 오류를 빨리 제거하는 방법이다. 코딩은 조금 다른 접근 방법을 취한다. 소프트웨어 개발자들이 작성하는 코드는 실행시켜보는 것이 버그를 발견하는 가장 확실한 방법이다. 그리고, 실행 전에 해야 할 것들도 있다. 코드 리뷰, 정적분석(Static Analysis), 단위 테스트(Unit Test), 통합 테스트(Integration Test) 등이 있으며, 이를 CI(Continuous Integration) 서버와 연동해서 자동화시켜야 한다. 이 부분에서의 핵심은 프로젝트를 잘게 나누어 단기 과제화시키는 것이다. 즉, 완전히 완료된 소프트웨어를 한 번에 배포(Release)할 생각은 접어야 한다는 말이다. 중간 결과물에 대한 품질을 최대한 끌어올려 거의 배포할 수 있는 수준에 도달하는 것을 목표로 잡아야 한다. 물론 모든 개발해야 할 기능들을 그 수준까지 높이라는 말은 아니다. 작게 나누어진 과제에서 구현할 기능들만큼은 최소한 높은 품질 수준을 달성하도록 노력해야 한다는 뜻이다.

단기 과제로 잘게 나눌 때 핵심이 되는 기능은 가능한 먼저 개발되어야 한다. 핵심이 되는 기능은 폭을 우선하는 것이 아니라 깊이를 우선으로 구현되어야 한다. 하나의 기능을 구현하는 것은 입력에서 가공, 출력까지 수직적으로 구현되어야만 한다. 그렇다고 해서 내부의 모든 구성 요소들이 한꺼번에 구현되는 것이 아니라 해당 기능에 필요한 만큼씩만 구현되어야 한다. 딱 그 정도만 구현하면 충분하다. 따라서, 첫 번째 배포 버전은 우리가 만들고자 하는 소프트웨어의 핵심 기능만 제공하고, 그것만을 중점적으로 테스트되어야 한다. 나머지 잘게 나누어진 단기 과제들은 우선순위별로 정렬된 기능들을 높은 우선순위에서 낮은 우선순위로 차례로 구성하면 된다. 구현되는 기능의 수는 개발자의 수와 능력에 비례해서 잡으면 될 것이다. 물론 각각의 단기과제들의 끝에서 반드시 다음 단계를 어떻게 수행할지에 대한 계획도 있어야 하고, 완료된 기능들은 배포수준에 도달해 있어야 한다. 배포수준에 도달했다고 생각이 되고 의미 있는 기능이 구현되었다고 생각되면, 과제와 관련된 사람들을 모아서 과제의 진척도를 보여주기 위한 중간 데모를 하는 것도 좋다. 사람들의 의구심에 가득 찬 눈초리를 순화시킬 수 있는 좋은 기회로 이용할 수 있다. 중요한 것은 구체적인 성과를 지속해서 보여주고 그런 성과들이 명확히 드러나야 한다는 점이다. 적어도 개발자라면 거짓말을 해서는 안된다. 보여주는 것이 가장 확실한 방법이다.

정리하면 과제의 후반에 테스트를 집중하는 것은 위험(Risk)부담이 큰일이다. 위험 부담을 낮추고 복잡한 문제를 한번에 하나씩 해결해 나가는 것이 해답이다. 대부분 과제가 초반에는 잘 돌아가는 듯이 보이다가도 뒤로 갈수록 욕먹고 잔업과 특근으로 점철되는 것은 문제가 발생한 시점에서 문제를 제대로 해결하지 않기 때문이다. 발견된 문제는 반드시 가장 빨리 고쳐져야 한다. 그리고, 그 발견되는 시점이 발생한 시점에서 가능한 한 가까워야 한다. 새로운 기능도 문제를 해결하기 전에는 더 추가되어서는 안 된다. 새로 추가된 기능으로 문제를 찾는 것은 점점 더 어려워질 것이기 때문이다. 초보 관리자들이 실수하는 이유는 성급함에 있다. 급하게 만들어진 코드들은 버그들이 숨어있을 가능성이 높고, 그렇게 만들어진 코드들이 통합되면 버그들은 방어막을 가지게 된다. 버그는 자신이 발견되지 않도록 가능한 위장하게 되며, 그런 위장막을 벗겨서 버그의 원인을 찾는 것은 여러 사람을 다치게 만들 수 있다. 과제의 관리자나 권한이 있는 임원이라면 발생한 버그 자체나 개수를 보지 말고 근본적인 원인이 무엇인지를 사람에게서 찾으려고 노력해야 한다. 버그는 저절로 만들어지는 것이 아니라 사람이 만들어 내기 때문이다.

# [ 코드에 옷을 입히다. ]

코딩 스타일을 정하는 것은 코드에 일종의 옷을 입히는 것 같다. 코드가 하는 일을 정확히 파악하는 것은 대단히 중요한 일이며, 이를 꾸미는 것도 의미가 없는 일이 아니라 반드시 개발자라면 잘 익혀두어야 하는 부분이다. 대부분의 코드 개발은 기능이 동작하는지만 확인하는 수준에서 마무리가 된다. 이렇게 개발된 코드는 더 이상의 개선은 없고 반복적인 디버깅만 남아있을 뿐이다. 디버그 과정의 대부분은 남이 작성한 코드와 자신이 작성한 코드를 읽는 시간이다. 즉, 코드를 이해하는데 대부분의 시간을 사용하고, 이해가 된 부분에 대해서 버그를 찾아서 수정하는 것은 짧은 시간 동안만 이루어진다. 따라서, 우리가 해야 할 일은 코드가 쉽게 이해되도록 만드는 것이며, 조금이라도 문제를 줄이고 빨리 찾아서 해결하는데 걸리는 시간을 단축시키는 일이다.

코드에 옷을 입히는 일은 코딩을 시작하기 전에 프로젝트에 관련된 사람들이 코딩 스타일을 정하는 것으로부터 시작된다. “{}"를 어느 위치에서 시작할 것인가를 정하고, "Tab"은 공백문자로 몇 자로 할지, "함수"의 이름은 어떤 식으로 주고, "디렉터리" 이름을 어떻게 정할지, 각각의 파일들은 어떤 형식과 이름으로 만들지, "Preprocessing"과 관련된 "Directive"들은 어떻게 처리할지, "Enum"이나 "Macro"들은 어떤 방식으로 사용할지, 함수의 "인자"에 대한 기본 타입은 무엇이며 몇 개를 한계로 할지, 함수의 “Return”값은 어떻게 하고, 함수들이 넘겨받는 인자들의 "유효 범위"를 전수 검사를 할 것인지와 그렇게 검사할 때 사용하는 "Macro"들은 어떻게 할지, "Debug”을 위한 “Log”문은 어떻게 만들며, "Integer" 값에 대한 타입 및 기타 자료구조에 대한 타입들을 사용할 때는 이식성을 위해서 "type.h"와 같은 것을 만들지 말지 등등을 정하게 된다. 이렇게 정해진 규칙들은 과제에 참여하는 인력들과 논의를 통하게 되며, 한번 정해진 룰에 따라 모든 코드를 작성하게 된다. 만들어진 전체 코드는 누가 보더라도 마치 한 사람이 코딩한 것처럼 받아들여져야 하는 것이 핵심이다.

사람들이 모두다 똑같이 옷을 입었다는 것을 상상하면 매우 단조롭고 개성도 없는 것처럼 보인다고 생각할지 모르겠다. 하지만, 적어도 코딩에서는 이렇게 하는 것이 필요하다. 즉, 논리적인 비약과 생략 없이 일관된 스타일로 코딩을 해야 한다. 누가 보더라도 쉽게 이해가 가능한 코드는 다양한 스타일이 혼재된 것이 아니라 한 가지 스타일로 이루어진 것이다. 이럴 필요까지 없다고 생각하고 자신만의 코딩 스타일을 팀원들과 다르게 사용한다면, 이것은 자신만을 위한 코딩이지 팀을 위한 코딩은 아니다. 누가 보더라도 명확히 이해가 가는 코드만이 “좋은 코드(Good Code)”다. 그렇지 않은 코드는 개선되어야 한다. 그리고, 코딩 룰에 대한 것을 아예 프로그램을 개발하는 도구에 설정해 둘 수도 있다. 툴을 사용할 경우에는 코딩을 마쳤을 때, 정의된 규칙을 이용해서 자동으로 모든 코드에 대한 것을 일괄적으로 바꿔줄 수 있을 것이다. 코딩 스타일을 체크하는 것도 코드 리뷰에서 필요하다. 정적 분석도 코딩 스타일을 지키는지 확인해야 한다. 여러 단계로 자동이나 수동으로 진행하는 이유는 실수를 사전에 방지하기 위함이다. 사람은 언제나 실수를 하기 마련이고, 단순한 실수를 찾는 것은 기계가 가장 잘 하는 분야이기 때문이다. 사람은 좀 더 생산적인 일에 집중할 수 있게 된다. 수정이 귀찮은 일이라고 한다면 개선이란 먼 이야기다. 나날이 조금씩이라도 더 좋은 코드를 만들어가는 습관을 지녀야 할 것이다.

규칙은 지켜야 의미를 가진다. 지키지도 못할 약속을 하는 것은 서로 간에 신뢰만 손상할 뿐이다. 규칙을 만들때는 절대 지켜야 할 것만 나열하는 것이 좋을 것이다. 너무 많은 규칙은 지키지 못할 가능성이 높으며, 규칙을 만든 의도를 희석할 수도 있다. 너무 많은 옷을 입으면 활동하기 불편하기에 적절한 수의 규칙을 만들어야 할 것이다. 대체로 10개를 넘지 않는 범위에서 함께 모여서 만들면 좋을 것이다. 다른 필요하다고 생각되는 규칙들은 도구를 이용해서 해결하는 것을 생각해 볼 수도 있다. 도구로 검사하기 어렵거나 기존에 이미 문제를 일으킬 가능성이 높은 코드를 제거하는 것은 사람이 해야할 일이다. 코드 리뷰에서 검사 항목으로 정하고 꾸준히 사람이 보는 수밖에 없다. 코드에 적절한 옷을 입히고 잘 이해되도록 돕는 것은 생산성이 낮은 일이 아니라는 점은 분명하게 인식해야 한다. 코드는 만드는 시간보다 읽는 시간이 훨씬 더 긴 활동이라는 것을 반드시 기억해야 할 것이다.

# [ 오래되고 익숙한 툴 vs. 새롭고 낯선 툴 ]

소프트웨어 개발자들 자신들이 최신 기술 속에서 산다고 생각하지만, 최신 툴을 사용하는 것에는 거부감을 가진다. 이미 손에 익숙한 툴을 버리는 것을 마치 오래된 애인과 헤어지는 것처럼 생활의 일부를 잃어버린다고 느낀다. 하지만, 단언컨데 애인을 바꾸는 것보다는 더 짧은 시간이 걸릴 것이다. 세상에는 수많은 툴들이 있으며 모든 툴을 다 알 필요는 없지만, 항상 생산성이 높고 꾸준히 지원을 받을 수 있는(꾸준히 업데이트가 일어나며 커뮤니티가 구축된) 툴들은 충분히 손에 익혀놓는 것이 좋다. 예를 들어, “Source Insight”나 “UltraEditor”와 같은 상용 툴들을 익숙하게 사용하는 사람에게 “Eclipse”와 같은 통합개발 환경을 사용하라고 하면, "그렇게 하려면 불편해요.", "너무 무거워서 일하기 힘들어요."라는 대답이다. 하지만, 그런 대답의 이면에는 "그냥 지금 있는 툴이 편하니 방해하지 마세요"가 있다.

툴은 시간이 지나면서 꾸준히 발전하고 있다. 상용으로 파는 도구뿐만이 아니라 오픈소스로 개발되는 툴들도 꾸준히 개선되고 있다. 예전에 잘못된 선입관을 가졌다면, 지금은 이미 개선되었을 가능성이 높다. 무겁고 잘 동작하지 않는 툴들이 지금까지 살아남아 있을 가능성은 없기 때문이다. 그리고, 충분한 커뮤니티를 가지고 있는 툴이라면, 최소한 지원에 관련된 문제도 이미 해결되었을 가능성이 높다. 즉, 툴로서 생존환경을 이미 구축한 상황이라는 말이다. 따라서, 우리는 그런 툴들을 거부할 정당한 이유(?)를 찾기가 점점 어려워지고 있다. 오히려 그런 툴들을 적극적으로 활용할 수 있는 방안을 찾는 것이 더 좋지 않을까? "잘 하지 못하면 더 많이 자주 하라"라는 말이 있다. 시간이 부족하다고 이야기하지만, 모든 툴들은 20%의 기능을80%라는 사용자가 쓰고 있을 것이다. 따라서, 사용하기 어렵다는 말도 별다른 설득력이 없다. 대부분 자주 사용하는 기능은 짧은 시간 내에 배울 수 있다. 그렇지 않다면 툴이 사용자의 불편을 감수하고 살아남기는 어렵기 때문이다.

기존의 기능들 이외에 추가적인 이익을 주는 부분이 없다고도 핑계를 댈 수 있다. 하지만, 분명한 것은 이것도 잘못된 것이라는 점이다. 예를 들어, 에디터 기능만을 사용하는 툴과 통합 개발환경은 그 차이가 어마어마 하다. 물론, 전문적인 에디터는 아니지만 대부분의 전문 에디터들이 제공하는 기능들은 다 제공되는 통합 개발 툴은 있다. 에디터의 전문가가 아니라면 사용하지 않는 기능까지도 전부 통합 개발 환경이 제공해야 한다고 이야기한다면, 그런 전문가의 수준까지 도달한 소프트웨어 개발자라면 오히려 사용하는 툴에 대한 제한은 하지 않을 것이다. 따라서, 이것도 논리적인 이유가 되지 못한다. 최신의 개발 툴들은 통합 환경과 커맨드 라인을 둘 다 제공하는 방법들을 가지고 있다. 소프트웨어 개발자들은 통합 환경에서 개발하고, 거기서 생성되는 결과물들은 커맨드 라인으로 빌드(Build)할 수 있다는 말이다. 둘 다를 제공할 이유는 CI(Continuous Integration)와 같은 환경에서 사용할 수 있어야 하기 때문이다. 그리고, 자동화 툴들을 이미 고려해서 통합 개발 툴들은 만들어져 있다. 이미 툴의 사용자들이 많이 요구를 했을 것이기 때문이다.

손에 익숙한 낡은 "손도끼"를 사용하는 것이 옳을 것인가? 아니면, 손에 다소 익숙하지는 않지만 생산성이 높은 "기계 톱"이 더 나을까? 둘 간의 비교 평가는 당신의 손에 맡겨지겠지만, 장기전인 관점에서의 생산성에 대해서 충분한 고려를 해야 한다. 단기적으로는 지금 사용하는 툴들이 익숙해서 꾸준한 생산성을 발휘할 수 있겠지만, 더 나은 생산성 향상까지는 도달하지 않는다. 왜냐하면, 이미 당신은 최고 수준의 생산성을 만들고 있을 것이기 때문이다. 하지만, 더 나은 생산성을 제공할 수 있는 툴은 처음에는 사용하기 서툴지라도 점차 더 높은 생산성을 이룰 수 있도록 만들어주며, 생산성 향상도 꾸준히 높여나갈 수 있다. 시간이 부족하다고 개선을 미루는 순간 더 많은 시간 부족에 시달리게 되는 것이 복잡한 일의 전개 과정이다. 시간은 예전에도 부족했고, 지금도 부족하며, 나중에도 만회할 수 있는 자원이 아니다. 다만 동일하게 주어지는 시간을 누가 더 효율적으로 사용할 수 있느냐가 차이를 만들 뿐이다. 따라서, 지금이 아니면 그것을 익힐 수 있는 시간은 없다. 낡고 손에 익숙한 도구를 버릴 이유도 없지만, 그렇다고 쉽고 빠르게 사용할 수 있는 도구를 거부할 이유도 없는 것이다. 도구 상자에 쌓여가는 것들이 늘어날수록 다양한 방법으로 문제를 해결할 가능성도 커지게 될 것이다.

# [ 프로세스 vs. 코딩 ]

소프트웨어 개발자는 개발 프로세스(Process)를 싫어한다. 좀 더 정확히 이야기하면 소프트웨어 개발자는 개발 프로세스에서 이야기하는 각종 문서를 쓰기를 싫어한다. 하지만, 제대로 개발이 되지 않을 때 생기는 회사의 대응은 "소프트웨어 개발 프로세스 강화"다. 그리고, 소프트웨어 엔지니어링 조직을 만들어 개발 방법론(프로세스)의 보완에 나선다. 결국, 문제가 빈번한 게 발생하는 과제들이 있으면 개발 프로세스는 나날이 복잡해지고 강화되어 간다. 그리고, 그런 프로세스의 주요 입력은 항상 검증되지 못하거나 내용이 부실한 결과물들로 채워진다. 누구도 읽지 않는 문서들만 시스템에 등록되어 남는다. 그리고, 그 문서들의 내용은 이미 현실의 코드와는 상당 기간 "동기화"되지 않아서 소프트웨어의 전체적인 구조나 특성, 기능등을 파악하는데 전혀 도움이 되지 못한다. 이런 일의 지속적인 반복으로 인해 실제 업무보다는 행정적인(사무적인) 업무들에 소비되는 시간은 점점 늘어나고, 제대로 관리되지 못하는 과제의 전형들은 다시 반복적으로 발생한다. 누구도 잘못 하지 않는 것처럼 보이지만 모두가 잘못하고 있는 상태가 되어버린다.

문제는 프로세스가 소프트웨어 개발자들에게 또 다른 일이 된다는 것이다. 비대해진 프로세스는 그 자체가 일이다. 그것을 관리하는 것부터, 그것을 지켜지는지 확인하는 것과 등록한 결과물들을 생성하고 지속해서 관리(업데이트)하는 일까지 모든 것이 일이 된다. 전담해서 해야 할 사람들이 따라붙고 프로세스는 더욱 정교하게 만들어져 간다. 마치 아무도 쉽게 빠져나가지 못하는 철조망을 만들고 싶다는 듯이 일사불란하게 진행된다. 그렇다면 그렇게 들어가는 개발의 오버헤드(Overhead)는 누가 제거해 줄 것인가? 불행히도 아무도 제거해 주지 못한다. 왜냐하면, 프로세스에 지속적인 영양분을 공급하는 것은 개발자들이기 때문이다. 그것을 공급해 주기 위해서는 프로세스를 위한 여분의 시간이 필요하지만, 실제로 그런 업무들을 명확히 파악해서 얼마나 많은 시간이 필요한지 과제의 일정에 반영하지 못한다. 따라서, 프로세스에 필요한 활동들은 제대로 뿌리를 내리지 못하게 되며, 보여주기 위한 형식상의 결과물만 그럭저럭 과제관리 시스템에 채워 넣기 바쁘다. 하지만, 누구도 보지 않는 그런 결과물들을 시스템에 남겨놓는 것이 무슨 의미가 있을까? 관리자들은 자신의 책상 위에 놓인 프린트된 보고서를 읽을 시간도 없다. 그들도 "바.쁘.다!!!"

“코딩은 생각을 구현하는 활동”이다. 따라서, 생각이 먼저다. 생각한 후에 구현해야 한다. 어떻게 구현할지를 먼저 머리속이나 노트에 표현해야 하고, 이를 실제 동작하는 코드로 만드는 과정이 구현이다. 코딩은 테스트도 포함한다. 코딩만 하고 테스트를 하지 않는 것은 자신이 해야 할 일의 일부만 하는 것이다. 프로세스는 생각을 정리하고, 규정에 맞게 구현되었으며, 제대로 테스트가 되었는지를 확인하는 것이다. 프로세스는 그림의 틀이지 그림 자체가 아니다. 따라서, 코딩 자체에 대해서 특정 형식에 맞게 제한을 가할 수는 있지만, 코드 자체의 내용까지는 관여해서는 안 된다. 틀이 그림을 가리는 경우는 없기 때문이다. 프로세스는 코드라는 그림을 과제의 목표에 고정하는 틀의 역할로 충분하다. 너무 많은 제약을 가하면 그 자체를 거부하는 현상이 발생한다. 하지만, 물이 지나가는 길을 유도하듯이 도랑을 잘 파놓는다면, 물은 그 흐름을 거스르지 않고 흘러갈 것이다. "천망회회 소이불실"이라는 말이 있다. 즉, "하늘의 그물을 넓고 성기지만, 거르지 못하는 것이 없다"라고 했다. 프로세스는 이처럼 구현에 충분한 자유도를 주지만, 문제점은 반드시 찾아서 거를 수 있는 수준이어야 한다. 따라서, 프로세스를 운영하거나 정립하는 사람은 항상 주의해야 한다. 자신이 "힘이 있는 존재"가 아닌 "도움을 주는 존재"로 인식될 수 있도록 "낮은 곳으로 임해야 한다."

절차상으로 보았을 때 “상세 설계와 코딩, 작성한 코드에 대한 단위 테스트는 함께 묶여 있다”고 봐야 한다. 즉, 버그를 발생한 시점과 찾는 시점, 고치는 시점이 함께 이뤄져야 개발을 빨리 진행할 수 있다. 설계는 지속해서 변경이 일어나는 과정이라고 봐야 할 것이다. 물론 개발 초기에 큰 그림("High Level Design") 수준으로 미리 정해져야 한다. 대략적인 시스템의 구조를 정의하는 것은 코드의 근간(구조)을 결정하는 중요한 일이다. 상세 설계는 구현 바로 전에 발생하는 것이며, 코딩하는 동안 상세 설계는 변경될 수 있다. 코딩의 결과물인 코드와 테스트 스크립트들은 분석을 통해, 기계적으로 찾아낼 수 있는 버그들을 걸러낸다. 그리고, 지속적인 통합의 절차를 거치면서 모듈 간의 불일치 및 통합 테스트가 진행된다. 실제 타겟(Target)에 적용해서 테스트되어야 할 일부도 자동화를 할 수 있으며, 반드시 사람의 개입이 필요한 부분만 매뉴얼(Manual) 테스트로 맡겨진다. 이런 모든 과정에서의 산출물들은 코드의 품질을 높이는 것이지, 코딩하기 어렵게 만드는 것들이 아니다. 코드는 나날이 품질과 기능이 좋아져야 제대로 성장하는 것이다. 그리고, 그런 것들을 높이는 방법은 개발자만이 할 수 있는 것이 아니고 프로세스와 같이 체계적으로 움직일 때 실수가 줄어든다. 좋은 도구와 절차를 적절히 이용할 수 있어야만 제대로 할 수 있을 것이다.

# [ 내가 만들지 않은 코드가 최고(?) ]

소프트웨어 개발자들은 남들이 만든 코드를 믿지 않는 경향이 있다. 일례로 예전에 "우리 과제에서는 STL(Standard Template Library)를 사용하지 않습니다."라고 이야기하는 개발자를 본 적이 있다. 그 개발자는 "STL이 무겁고 느리다."고 자신들이 그것을 사용하지 않는 것을 코딩 룰이라고 했다. 사실 이런 말을 들으면 그 개발자와는 더 이상의 대화를 나누기가 힘들어진다. 왜냐하면, 자신이 만든 코드의 신뢰도를 "STL"보다도 더 높이 평가하는 이유를 알 수 없고, 기본적인 문제에 대한 접근하는 사고방식의 차이가 너무 크기 때문이다. 그냥 "아~ 그렇습니까." 정도로 대화를 마무리한다. 최소한 "STL"을 만드는 사람들은 아마도 지구상에 가장 똑똑한 개발자에 속할 것이며, 이미 많은 경우를 대비해서 안정성을 최대한 고려해서 만들었을 것이다. 수많은 사람이 "STL"을 사용했기에 이미 검증도 되었다고 볼 수 있으며 최신기술도 아니다. 물론 항상 STL을 사용하는 것이 좋은 선택이라고 주장하는 것은 결코 아니다. 가능한 남들이 이미 만들어 놓은 것을 재활용하자는 것이다.

가능한 코딩을 적게 하는 것이 버그를 발생시킬 가능성도 낮다. 이 말은 이미 검증된 것을 구할 수 있다면, 그것을 다시 만들 필요는 없다는 뜻이다. 의미 없는 개발은 버그를 낳고, 그 버그로 인해서 과제 완료일은 더 빨리 다가오는 것처럼 느껴질 것이다. 그 때까지도 수정하지 못한 문제나 이상하게 오동작하는 것들은 다음 개발까지 묻어두거나 "재현 불가"처리가 된다. 하지만, 그렇게 땅속에 깊게 묻어둔다고해서 터지지 않을 것이라고는 생각하지 말아야 한다. 특정 조건만 맞으면 언제든 폭발할 가능성은 항상 존재한다. 코딩을 적게 하라고 해서 코드를 짧게 만들라는 것은 아니다. 어떤 기능을 구현하기 위해서 가능한 재활용할 수 있는 것들을 찾아서 사용하라는 말이다. 짧고 복잡한 코드를 짜는 것은 퍼즐 경진대회의 입상을 위해서는 좋을지 몰라도, 팀으로 일하는 회사의 과제에서는 사용하지 않는 것이 좋다. 물론 성능을 극단적으로 끌어올려야 하는 코딩도 있다. 하지만, 그것은 알고리즘의 문제이지 그것을 구현하는 코드의 가독성(Readability)을 낮게 만들라는 의미는 결코 아니다. 어떤 상황이라도 가독성은 코딩에 있어서 반드시 구현되어야 할 속성이다.

소프트웨어 개발을 혼자서 하는 경우를 제외하면 모든 과제는 어떤 식으로든 남의 코드를 가져다 쓴다. 그 코드를 믿지 못하겠다는 것은 같이 일하는 팀원의 역량을 믿지 못하겠다는 말과 같다. 그리고, 자신의 코드도 남들이 믿지 못할 가능성이 충분히 있다고 봐야 한다. 이를 어떻게 극복할 것인가? 첫 번째는 남들과 같이 코드를 공유해야 한다. 공유란 그냥 복사해서 사용하라는 이야기가 아니다. 팀원들이 만든 결과물에 대해서 리뷰를 같이하라는 것이다. 두 번째는 적어도 자신의 코드에는 문제가 없음을 스스로 테스트로 보증하라는 것이다. 즉, 테스트 스크립트를 활용해서 자신이 만든 코드를 사용하는 예제도 만들고, 그와 동시에 테스트를 실행해서 유효한 입력에 대해서 제대로 동작하고 있다는 것을 객관적으로 증명하라는 이야기다. 세 번째는 서로 대화를 자주 하라는 것이다. 믿음은 잦은 대화를 통해서 생겨난다. 대화의 일부는 인터페이스가 될 수도 있고, 잦은 통합(Integration)이 될 수도 있다. 자주 통합하면 그만큼 코드 간에 발생하는 대화(호출)의 오류를 줄일 수 있다. “고립된 개발”은 자신의 코드를 소중하게 유지할 수 있을지는 몰라도, 전체 최적화된 시스템은 "편협한 생각"을 통해서 만들어지지 않는다.

시작은 자신의 코드가 완벽하지 않다는 것에 대한 인식이다. 자신의 코드가 완벽하지 않기에 항상 더 개선할 부분이 있다고 생각해야 한다. 그리고, 남들이 잘 만들었다고 생각하는 코드를 익혀야 한다. 보는 안목을 길러야 자신을 반성할 수 있다. 어떻게 코딩하는 것이 좋은 것인가를 스스로가 인지하지 못한다면 편협한 시각만 키울 뿐이다. 이런 의미에서 C++에서 사용하는 "STL"은 일종의 표준 C라이브러리와 같다고 보는 것이 옳다. 좋은 것은 불러서 사용하면 그만이다. 그렇다고 모든 소위 말하는 프레임워크(Framework)들을 과제에서 반드시 사용해야 한다고 말하는 것은 아니다. 문제를 가장 잘 해결할 방법을 찾으라는 것이지, 반드시 그런 것들을 써야 한다는 이야기는 아니다. 기술적으로 우수하다고 해서 자신이 풀어야 할 문제의 가장 좋은 해답은 아니다. 기술 만능주의에 빠져서는 실제로 해결할 문제를 제대로 풀지는 못한다. 예전에 사용했다고 해서 지금 풀어야 할 문제에 대해서도 해결책이 될 수 있는 것은 아니다. 기술은 항상 발전하며 “더 좋은 것은 언제나 미래에 있다”.

# [ 분석툴을 믿어야 한다. ]

소프트웨어 개발은 상당히 많은 자동화를 이룬 것이 사실이다. 물론, 코딩의 완전한 자동화는 불가능하다는 것은 이미 알려진 사실이다. 설계와 코딩은 자동화 툴로 어떻게 할 수 없는 부분이다. 하지만, 분석에 관련해서는 많은 부분이 자동화되어 있다. 문법적인 것은 컴파일러가 충분히 커버할 수 있다. 경고(Warning) 수준을 최대로 해서 컴파일하는 것이 기본이다. 그리고, 각종 툴들을 이용해서 분석된 결과를 통한 문제점(Defect)들도 버그 제거의 좋은 시작점이다. 주로 정적 분석 툴을 이용해서 CI(Continuous Integration) 서버에서 통합과 동시에 분석이 일어나게 되며, 기계적으로 분석된 결과는 사람의 검토 활동을 통해서 해결되어야 한다. 즉, 기계적으로 분석된 것을 가장 먼저 처리해 주고, 인간의 고차원적인 두뇌 활동으로 해결해야 하는 버그에 집중하는 것이 좋다. 인건비가 주된 비용인 소프트웨어 개발에서는 사람은 더 높은 가치를 만드는 활동에 전념하는 것이 좋기 때문이다. 단순한 실수 때문에 유발될 수 있는 버그들은 도구를 활용해서 처리하는 것이 비용이 싸다.

소프트웨어 개발자들은 툴이 자신의 코드에서 버그를 찾아내 수정하라고 요청하면 크게 두 가지 반응을 보인다. 첫 번째는 고맙다는 말과 함께 수정하는 방법을 물어온다. 두 번째는 자신이 만든 코드에서 툴이 분석한 오류(Defect)는 툴의 오류이며 절대 수정할 수 없다고 거부한다. 첫 번째 반응은 개선의 시작에 대해서 좋은 신호를 준다고 볼 수 있다. 툴의 유지보수와 검증을 담당한 엔지니어들은 이것에 대해서는 항상 도와주려는 마음을 가지고 있으며, 최대한 쉽게 고칠 방안들을 만들어서 알려줄 수도 있다. 두 번째 반응에 대해서는 난감해한다. 고치도록 만들고 싶지만 거부감이 심한 사람에게는 어떤 좋은 말도 지금 상황에서는 들리지 않는다. 아마 그렇게 말하는 그들도 자신들이 조금만 마음의 여유가 있었다면 그렇게 이야기하지 않았을 것이다. 자신들의 문제를 차근차근 검토해서 실제로 무엇이 문제인지 확인하고 수정하는 방법을 물어왔을 것이다. 하지만, 배포가 얼마 남지 않은 상황이라면 문제가 있다는 보고를 공개적으로 받는 것을 쉽게 받아들이기 힘들어 할 것이다.

툴은 개발의 효율 향상을 가져올 수 있다. 이미 수많은 툴들이 개발되어 있으며, 상용 혹은 오픈소스 툴들이 시장에는 많이 나와 있다. 툴이 모든 것을 해결해 주지는 않지만, 그렇다고 전혀 쓸모없는 것은 아니다. 따라서, 필요하다면 적극적으로 활용하는 것이 옳다. 정적 분석툴에서 찾아주는 것들은 사람이 간과할 수 있는 쉬운 오류들이며, 개발자 본인이 놓칠 수 있는 실수를 사전에 점검할 수 있는 기회를 준다. 예를 들어, 안전한 코딩(Secure Coding) 오류에 대한 것들을 대부분 개발자가 놓치고 있는 부분이며, 자신의 코드에 얼마나 복사된(혹은, 중복된) 코드들이 있는지도 파악하지 못한다. 이러한 코드들은 위험도가 높은 것들이지만 쉽게 고칠 수 있는 부분이다. 코드 리뷰를 착실하게 해도 이런 부분들을 존재할 수 있다. 특히, 자원 유출(Resource Leak)과 같은 오류들은 사람이 찾기 어려운 부분이지만 툴은 쉽게 찾아주는 경향이 있다. 자원 유출 문제는C언어를 사용할 때 메모리 할당과 해제의 불일치로 인해서 흔히 발생할 수 있으며, C++에서는 생성자의 잘못된 정의로 발생할 수도 있다. 이런 부분들이 있다면 해결책과 함께 가이드(Guide)를 필요로 하게 되며, 그런 가이드들은 개발자들 간에 공유되어야 할 것이다.

자신의 코드를 객관적으로 들여다보는 것은 어렵다. 마치 장기나 바둑을 두는 사람이 자신의 수에 대해서 훈수를 두는 사람이 더 잘 볼 수 있듯이, 객관화된 시각으로 코드를 감시할 수 있는 장치(방안)가 필요하다. 개발자들은 자신의 코드에 개선이 필요하다는 신호를 절대 간과해서는 안 된다. 비록 자신의 코딩 방법이 맞다고 주장하더라도 기계적으로 검사된 결과가 자신의 논리에 앞설 수도 있다는 점을 이해해야 할 것이다. 모든 확인된 오류는 반드시 제거된(변경된) 후에 코드를 배포하는 것이 옳다. 자신의 코드에 대한 객관성을 유지하는 것은 어렵다. 따라서, 도구에서 검토된 의견을 받아들이는데 가능한 선입견이 없도록 만드는 것이 가장 좋다. "False Positive"로 처리하기 전에 정말 그것이 오류가 아닌지를 명확히 해야 한다. 자신의 의견만을 주장할 것이 아니라 남들의 의견도 반영해서 "잘못된 검출"이라고 이야기해야 한다. 그렇지 않다면 공허한 말싸움만 있을 뿐이다. 말로는 아무것도 해결되지 않는다. 그리고, 기계적인 방법으로 검출된 오류는 개인의 역량과는 무관할 수 있다. 따라서, 개인의 역량이 정말 필요한 것에 집중하기 위해서도 그런 오류들은 다 수정하는 것이 옳다. 비록 그것으로 인해 과제 기간이 늘어날 가능성이 있다고 하더라도, 발견된 오류는 될 수 있으면 빨리 수정하는 것이 앞으로 있을 개선을 위해서도 좋은 토대가 될 것이다.

# [ 코드가 말한다(?) ]

아무리 훌륭하고 세밀한 프로세스(Process)를 가지고 있다고 하더라도, 그것이 품질 수준이 높은 동작하는 소프트웨어를 만드는 것을 보장하는 것은 아니다. 물론 그런 프로세스를 가지지 않은 것보다 품질수준이 높은 소프트웨어를 만들 가능성은 있겠지만, 그렇다고100% 확신하지는 못한다. 결국 동작하는 소프트웨어를 사용자에게 전달해 그들이 요구하는 바를 충실히 제공하는 것이 개발의 목표다. 결과적으로 프로세스는 그러한 소프트웨어를 만드는 절차 및 규정을 이야기할 뿐, 직접 만드는 사람의 역량까지는 정의하지 못한다. 좋은 사람을 구하는 것이 좋은 프로세스보다 나은 결과를 만든다는 것은 자명하다. 그들이 결국 생각을 실체화하는 역할을 담당할 것이며, 설계와 구현은 자동화된 프로세스가 아닌 사람의 노력이 필요하기 때문이다. 그리고, 사람 간의 역량 차이는 엄연히 존재한다는 것이 현실이기도 하기 때문이다.

코드는 말을 해야 한다. 그렇다면 무슨 말을 할 것인가? 코드는 사람이 이해하기 쉬운 언어로 사람과 대화할 수 있어야 한다. 코드는 여러 사람이 공유하는 것으로 제대로 개발자의 의도를 전달하지 않는 코드는 이해하는데 많은 시간을 보내야 한다. 코드는 쉽게 수정할 수 있는 구조를 가져야 하며, 수정한 의도도 제대로 표현될 수 있어야 한다. 코드는 간결해야 하지만 요구를 수용할 수 있을 만큼 충분히 유연하기도 해야 한다. 그렇기 때문에 좋은 코드를 만드는 것은 어렵다. 어려운 일이기에 꾸준히 개선하는 방법을 선택하는 것이 핵심이다. 한 번 만에 좋은 코드를 만드는 것은 한 번 만에 걸작을 만드는 것과 같다. 이런 것은 소위 말하는 천재들이 할 일이고 대부분 코드는 평범한 개발자들이 만든다. 평범한 개발자가 이해하기 쉬운 코드를 만든다면 어느정도 기본 소양을 갖춘 개발자라면 누구라도 기능추가와 개선이 쉬운 코드일 것이다. 복잡하게 만드는 것은 누구라도 할 수 있지만, 간결하고 이해하기 쉬운 코드를 만드는 것은 정말 어려운 일이다. 그리고, 충분히 도전할 만한 가치가 있는 일이다.

코드가 말을 해야 하지만 코드만 가지고 이야기해선 안 된다. 흔히들 개발자들이 가장 쓰기 싫어하는 문서도 필요하다. 문서의 역할은 목적을 설명해야 한다. 즉, 구현에 대한 방향성을 제시하기 위한 충분한 근거를 제공할 수 있어야 하며, 시스템을 한눈에 파악할 수 있는 수준이어야 한다. 좀 더 깊이 들어간다면 시스템을 구성하는 요소들에는 어떤 것들이 있으며, 그런 요소들이 어떤 관계를 맺고 있는지도 표현할 수 있어야 한다. 그리고, 그런 요소 간의 역할을 명확히 설명할 수 있어야 하며, 그렇게 역할을 나눈 이유도 충분히 표현되어 있어야 한다. 즉, 문서는 코드를 읽기전에 큰 그림을 그릴 수 있을 정도의 정보는 제공해 줄 수 있어야 한다. 문서가 좋은 소프트웨어를 만드는 보증 수표는 아니지만, 구현의 일관성을 제공하는 토대는 될 수 있다. 시스템을 구현할 때 필요한 기본 방향에 대한 좋은 길잡이 역할을 할 수 있을 것이다. 문서화에 신경 쓴다는 것은 관리하고 있다는 말이며, 관리란 역시 꾸준한 개선이 이루어진다는 이야기다. 꾸준한 개선(업데이트)은 코드만이 아니라 문서도 대상에 포함되어야 할 것이다.

코드는 테스트되어야 한다. 테스트된다는 것은 실행할 수 있다는 말이다. 소프트웨어 개발은 시작 첫날부터 실행 가능한 코드를 만들어야 한다. 그리고, 개발이 진행되는 동안 항상 실행 가능한 상태를 유지해야 한다. 며칠이 지난 후에 실행 가능하다고 이야기하는 것은 그동안에 생기는 문제에 대해서는 해결하지 않았다는 것과 같다. 문제는 발생한 시점에 해결하는 것이 가장 비용이 적게 든다. 따라서, 모든 문제는 발생 즉시 해결하는 것이 원칙이다. 따라서, 컴파일의 실패, 코드 통합(Integration)의 실패, 테스트의 실패, 자동화된 배포의 실패도 문제이며, 작은 개선을 꾸준히 하지 않고 크게 나중에 한 번만 하려고 하기 때문에 반복적으로 발생한다. 문제가 발생하면 그것을 먼저 해결해야 한다. 두고 볼 문제는 없다. 발생한 시점에 문제를 해결하지 않으면, 그것으로 인한 추가적인 문제 발생의 원인을 제공해 주는 것밖에 되지 않는다. 코드를 테스트하는 것은 문제를 찾는 시작이며, 작고 빠르게 실행해야 문제도 즉시 확인할 수 있다. 테스트는 문제가 없다는 것을 증명하는 것이 아니라, 문제가 있는지를 확인하기 위해서 프로그램을 실행하는 것이기 때문이다. "일신우일신"이란 말은 소프트웨어 개발자에게도 진리다.

# [ 프로그래밍에서의 낭비 ]

모든 낭비는 줄이는 것이 좋다고 교육을 받아왔다. 이것이 사실이라면 소프트웨어 개발에서 발생하는 낭비를 줄이는 것도 분명 옳은 일일 것이다. 코딩에서 발생하는 낭비는 추가적인 디버깅(Debugging)과 유지보수를 동반하는 활동이다. 가장 많이 발생하는 낭비는 필요없는 코드라고 볼 수 있다. 코드를 적게 짤 수록 디버깅 해야 할 범위도 줄어들고, 유지보수를 하기도 편해진다. 필요 이상으로 구현한 코드는 결과적으로 낭비를 불러오는 활동이다. 그것이 개발에만 국한된 것이라면 비용은 많지 않을지도 모르지만, 품질보증, 문서, 사용자 지원 등의 다양한 부분에서 필요없는 코드들은 추가적인 비용을 요구한다.

코드 한줄 잘 못 짠다고 크게 문제될 것은 없다고 여기는 것은 너무나 안이한 생각이다. 프로그램을 정말 잘 짜야 하는 이유는 그것이 곧 우리의 수입을 줄어들게 만들고 회사를 또 “하나의 가족”으로 생각하도록 강요하는 원흉이기 때문이다. 버그는 비용이며 집에 있는 가족들과 함께 할 시간을 줄어들게 만들기 때문이다.

대부분 코드는 작성되는 게 아니라 복제된다. 아마도 이것은 코딩하는 대부분 사람의 활동이 남들의 코드를 복제하는 과정이라고 단정할 수 있기 때문이다. 책을 보고 비슷한 문제를 찾는 것, 인터넷에서 이미 개발된 알고리즘의 구현을 찾아서 살짝 고쳐 붙이는 것, 다른 코드의 부분에 있는 동일한 부분을 "Ctrl-C, Ctrl-V"하는 것 등등 우리는 생각보다 많은 복제를 하고 있다. 물론, 모든 코드를 그렇게 짜지는 않지만, 복제된 코드가 여기저기 널려있다는 것을 확인한다면 어떤 반응을 보일까?

대부분 개발자는 그런 것을 문제로 인식하지 않는다. 작은 코드의 반복적인 복제 사용이 오히려 속도를 개선한다고 보는 경우도 있다. 하지만, 버그의 시작은 방심의 결과다. 복제된 코드들은 한 부분의 수정이 다른 부분의 수정으로 이어지지 않는다는 점에서 잠시만 잊어버리기만 해도 버그를 유발할 가능성이 높다. 기능 추가나 변경에 대해서 쉽게 깨질 수 있는 취약한 부분이 되기 때문이다.

# [ 버그를 줄이는 방법 ]

소프트웨어 속에 있는 모든 버그를 잡는 것은 불가능하다. 다양한 방법을 조합해서 사용할 수 있다면 좀 더 효율이 높은 버그 필터를 갖출 수 있다. 버그의 속성은 문제가 발생한 순간에 가장 명확하게 보이다가, 점차 시간이 흐를수록 찾는 노력과 수정하는 데 필요한 시간이 길어진다는 점이다. 따라서, 마지막 테스트 단계에 발생할 수 있는 버그를 조금이라도 줄이는 방법들을 순차적으로 적용할 수 있다면, 전체 개발에 필요한 노력과 비용을 조금이라도 줄일 수 있게 된다.

가장 먼저 사용할 수 있는 방법은 컴파일러다. 컴파일러는 문법적인 오류뿐만이 아니라 코드에 있을 수 있는 허점을 노리는 버그에 대한 "경고"를 해준다, 가능한 높은 수준의 "경고"를 해주도록 컴파일러를 설정할 필요가 있다. 이것만으로도 상당히 찾기 어려울 수 있는 버그들 일부가 정리될 수 있을 것이다. 모든 컴파일러의 경고는 그냥 넘어가서는 안된다. 비록 그것이 잘못된 보고라고 할지라도 충분히 검토할 가치가 있다. 그리고, 될 수 있으면 그런 경고를 인위적으로 못하도록 막는것 보다 제거하는데 집중해야 할 것이다.

컴파일러는 개발자를 위해서 필요한 도구다. 도구는 얼마나 잘 활용할 수 있는가에 따라 생산성이 달라질 수 있다. 그리고, 컴파일러가 해줄 수 있는 수준의 것들에 사람의 머리를 쓰는 것은 낭비다. 따라서, 이런 부분들은 기계적으로 수정되어야 한다. 컴파일러가 해줄 수 있는 일을 마쳤다면, 이제는 테스트 도구룰 사용할 차례다. 개발하는 언어에 따라, 그리고 사용하는 개발환경에 따라 달라질 수 있지만, 개발자가 익숙하게 사용할 수 있는 테스트 프레임워크(Framework)를 사용하는 것이다.

코드의 가장 작은 단위들은 독립적으로 검증될 수 있어야 한다. C언어에서는 코드의 가장 작은 구성요소(실행단위)를 함수(Function)로 볼 수 있고, C++와 같은 객체지향 언어는 클래스(Class) 수준이 될 수 있을 것이다. 따라서, 이런 수준에서 사용할 수 있는 단위 테스트 도구를 익혀야 한다. 이 과정을 마치고 나면 단위 수준에서 발생할 수 있는 버그들은 대체로 다 잡힐 것이다. 물론 단위 테스트는 단순한 도구 수준 이상의 의미를 가진다. 단위 테스트를 할 수 있을 정도의 코드라면 이미 구조화되어 있을 가능성이 높기 때문이다.

이제는 모듈과 모듈 간의 연결을 검사할 차례다. 버그가 발생하는 주요 원인은 잘못된 함수의 사용과 함수 사용에 따른 결과의 미확인이 많은 부분을 차지를 하고 있다. 따라서, 이를 위해서는 함수의 인수를 최소화시킬 필요가 있고, 함수의 복귀 값과 호출시 넘어오는 파라미터 값을 명시적으로 검사해야 한다. 잘못된 값을 가진 인수를 받았을 때는 함수가 오류를 돌려줄 수 있어야 한다. 기본적으로 제공되는 라이브러리 함수들에 대해서도 호출 후의 결과값은 반드시 확인해야 한다. 여기에 추가해서 안정성을 위주로 하는 코딩 전략을 배워야 한다. 예를 들어, 문자열과 관련된 함수들과 버퍼(Buffer) 오버런(Overrun)을 발생시킬 수 있는 함수들은 주의해서 사용하고, 권장하는 대체할 수 있는 함수들을 사용해야 할 것이다. 이 과정을 거치고 나면 모듈과 모듈 사이의 연결 정도 수준에서는 특별한 문제가 없음을 확인할 것이다.

다음 단계는 하위 시스템 수준(Subsystem)에서의 통합 절차가 필요하다. 이때는 통합 테스트 단계를 수행하기 위해서 편리한 툴(Tool)을 찾아야 한다. 입력 값들의 집합과 상위 인터페이스로 사용할 함수들, 특정 하드웨어나 환경이 없을 때 대체할 방법도 필요하다. 이 단계에서 필요한 것은 하위 시스템의 모듈 간에 동작하는 호출들을 추적할 필요가 있다. 호출 그래프(Call Graph)를 그려서 설계 단계에서의 흐름과 같은지도 확인해야 한다. 아직 완전한 시나리오를 적용하기는 어렵지만 각 함수의 호출이 원하는 순서를 따라가는지 확인할 수 있다. 하위 시스템 수준에서의 설계 명세와 비교해서 잘못된 점이 있는지 확인하고, 상세 설계의 변경이 필요한 경우에는 설계를 수정하는 것도 가능하다.

하위 시스템을 구성하는 모듈들의 통합은 이미 과제의 초기부터 지속해서 이루어져 왔을 것이다. 즉, 모든 코드는 초기부터 지속해서 통합되는 절차를 진행해야 한다. 이때 필요한 것이 기능별 구현이다. 기능적으로 구현하기 위해서는 계층구조에서 수직적으로 필요한 것들이 다 있어야 한다. 수직적인 기능위주의 구현 전략은 모듈 방식의 구현과는 다르다. 해당 기능에 필요한 부분만 모듈에 구현하고 검증을 반복하는 형태로 개발하는 것이다. 따라서, 기능이 요구하는 것 이상의 구현을 하지 않는 것이다.

몇몇 기능의 완료가 된 상황이라면, 이제는 구현된 기능이 자동으로 테스트되는 방법을 찾아야 한다. 안정적인 기능을 지속해서 확대하는 것이 핵심전략이다. 제대로 작동하는 기능도 새로운 코드가 추가되면 제대로 동작하지 않을 가능성이 높다. 따라서, 안정된 기능들이 꾸준히 그 상태로 남기 위해서는 모든 코드의 증분(Increments)에 대해서 이미 구현된 기능들에 대한 검증이 자동화되어야 한다. 기능들이 점차 추가되면서 시스템은 모습을 갖춰가기 시작한다.

시스템은 오류가 발생할 수 있다는 것이 기본 가정이다. 잘 발생할 가능성이 없는 것들도 검증이 되어야 한다. 갑작스럽게 전원이 나가거나, 파일 시스템이 가득 차서 더는 파일을 생성할 수 없는 경우도 있다. 데이터베이스 서버와의 통신이 두절될 수도 있고, 하드웨어의 고장이 발생할 가능성도 있다. 물론, 이런 모든 것을 다 가정하는 것은 불가능하지만, 자신이 만든 시스템이 사용자에게 치명적인 해(손실)를 입힐 수 있는 부분들에 대해서는 반드시 검증되어야 한다. 인위적으로 이런 오류들을 발생시키기가 쉽지 않을 수 있기에 시뮬레이터와 같은 것들이 필요할 수도 있다. 미리 개발의 초기에 이런 부분들을 신경 썼다면 도구를 가지고 있겠지만, 없다면 만드는 방법을 생각해서 설계에도 반영해야 한다. 즉, 테스트가 쉬운 소프트웨어 구조를 미리 정의할 수 있어야 한다.

위의 모든 과정에서 항상 코딩 룰(Rule)은 지켜져야 하고, 모든 작성된 코드들은 사람의 눈으로 검토되어야 한다. 조금 비싼 도구를 사용해서 "정적분석(Static Analysis)"을 할 수도 있다. 정적 분석에서 발견하는 오류(Defect)들도 반드시 수정되어야 한다. 흔히 개발자들은 자신의 코드에서 발생한 정적 분석의 결과를 쉽게 받아들이지 않는다. 하지만, 단언컨대 정적 분석기는 기계적으로 버그를 찾아주기에 사람보다 더 효율적이다. 따라서, 나온 결과를 의심하기보다는 수정하는 것이 옳다.

코딩의 전 과정에서 발생하는 복제된 코드에 대한 제거와 의미 있는 이름을 함수 및 변수, 상수, 파일, 디렉터리 등에 사용하는 것도 잊어서는 안 된다. 매크로에 대한 사용은 될 수 있으면 줄여야 하고, "printf()"와 같은 것을 사용해서 디버그 정보를 출력하는 것도 줄여야 한다. 읽기 쉬운 코드를 만드는 것도 버그를 줄이는 효과가 있다. 코드는 작성한 사람보다 더 오래동안 회사에서 생명력을 이어갈 수 있기 때문이다.

이런 모든 과정이 쉽지는 않겠지만 매일 매일의 활동 속에 전부 녹아들어야 한다. 그날 새롭게 작성된 코드에 대해서는 퇴근 전에 반드시 검증하는 것이 효과적인 버그에 대한 대처 방법이기 때문이다. 버그의 검출에서 제거까지 걸리는 시간이 팀의 역량을 나타내는 가장 쉬운 지표로도 사용될 수 있을 것이다. 높은 수준의 역량을 가진 조직은 오류를 검출하는 방법과 시기를 스스로 조절할 수 있는 충분한 힘을 가지고 있기 때문이다.

# [ 코드의 낭비 줄이기 ]

불필요한 코드는 항상 주의를 필요로 한다. 예를 들어, 주석으로 처리된 코드들을 보자. 코드를 이해하는데 전혀 도움을 주지 않으면서도 우리를 당황하게 만드는 주범이다. 이런 코드들을 당연히 지워도 된다. 왜냐하면, 우리는 코드 관리 시스템이라는 유용한 도구를 가지고 있기 때문에 언제든 원상태로 복구할 수 있다. 반복된 코드들도 문제다. 당연히 실행 파일에서 크기를 차지하게 되고, 수정해야 할 코드를 여러 곳으로 분산시키며, 결과적으로 실수할 수 있는 확률을 높인다. 하지만, 개발자들은 이런 것들을 너무나 가볍게 생각한다. "지나칠 수 없는 가벼움"이지만 그냥 무시한다. 당연히 한번 제대로 동작한다고 생각되는 코드는 다시 거들떠도 보지 않는다. 다른 기능을 구현하는데 바로 달려드는 것이 일반적이다. 물론 그렇게 개발된 코드가 문제가 없는 경우도 있겠지만, 제대로 검토되고 손보지 않으면 전체 코드 베이스가 쉽게 복잡해지는 것은 시간문제일 뿐이다. 당연히 시간이 지나면 간단하던 것들은 몇 층의 두께로 쌓인 양파껍질 같은 문제가 되어간다.

지나치게 많은 일을 하는 코드는 제대로 하는 일이 없는 코드다. 가능하다면 변수, 함수, 파일, 디렉터리 단위로 한 문장(접속사를 제외하고)으로 말할 수 있는 단위의 일을 처리해야 한다. 두 가지 이상의 일을 하고 있다고 생각되면 나누어야 한다. 필요 없는 주석도 실행되는 코드에는 속하지 않지만 문제가 될 수 있다. 잘못된 주석은 잘못된 지식을 전달할 수 있기에 코드에 들어가지 않아야 한다. 코드에는 "누가 언제 만들었다"와 같은 주석이 필요 없다. 코드 관리시스템은 체크인(Check-in)한 사람을 일일이 기록하는 수고를 마다하지 않기 때문이다. 코드를 체크인하게 될 때는 반드시 로그(Log)를 남겨야 한다. 그렇게 만들어지는 로그들은 특정한 형식을 가져서 오류의 수정인지, 아니면 추가적인 기능인지, 개선 활동인지 등을 알려주어야 한다. 나중에 그런 로그들을 모아서 과제에 대한 지표로 사용할 수도 있다. 물론 이러한 지표들은 과제 개발자들을 비난하기 위한 것이 아니라, 과제를 개선하기 위한 용도로 사용되는 자료들이다. 지표를 평가의 목적으로 사용하는 사람들은 결국 거짓 정보만을 받게 된다. 개발자들은 지표를 인위적으로 조작할 정도로 충분히 영리하기 때문이다. 개선의 목적으로 사용한다면 더 품질이 좋고 일정이 단축된 결과물로 자신의 이력서를 채우게 될 것이다. 무엇을 원하는가는 사물을 보는 시각에 달려있다.

시스템에는 변경되지 않는 부분과 변경이 허용되는 부분이 있다. 이를 잘 식별하는 것이 유연성이 높은 소프트웨어 구조를 설계하는데 바탕이 된다. 하드웨어는 지속적인 변경이 일어나는 부분이다. UI도 마찬가지로 바뀔 가능성이 높다. 표준적인 것들은 바뀔 가능성이 높지 않지만, 만약 표준이 여러 가지가 있다면 해당 표준을 사용하기 위한 상위 수준에서의 대책이 필요하다. 스케줄링 방식도 변경될 수 있고, 파일의 포맷도 변경될 수 있다. 따라서, 이런 것들은 변경에 유연한 방식으로 만들어져야 한다. 각 언어의 특징에 따라 이런 유연성을 확보할 방법들이 있기에 이를 활용하면 된다. C언어의 경우에는 "struct"나 "function pointer"를 활용할 수 있고, C++이나 Java의 경우에는 다형성(Polymorphism)을 이용할 수 있을 것이다. 식별된 중요한 변경요인은 반드시 시스템 설계에 반영해야 한다. 물론 이런 식으로 코딩하는 것이 최적이 아닐 것이라는 의심이 들 수도 있다. 하지만, 구조의 최적화가 오히려 더 좋은 성능을 가져다준다. 급한 최적화는 국지적인 성능만 높일 뿐, 시스템의 전체 최적화에는 역효과를 가져올 수 있다. 변경에 대한 대응은 일종의 리스크(Risk)를 관리하는 것으로 생각할 수 있다. 리스크는 아직 발생하지 않은 것이지만 언제든 문제가 생길 수 있는 부분으로 소프트웨어 개발에서는 항상 관리되어야 할 영역이다.

코드는 잘 정리가 되어야 한다. 일단 길다고 생각되는 것은 전부 의심해 봐야 한다. 너무 많은 일을 하지 않는지, 너무 복잡한 것은 아닌지 주의깊게 살펴보아야 한다. 복잡한 것은 추상화시켜서 해결한다. 간단히 복잡한 부분을 뽑아내서 다른 디렉터리나 함수, 변수 등으로 변경하면 된다. 다른 모듈이나 파일, 함수에 연결이 많은 코드는 제대로 역할을 나누지 않아서 발생하는 현상이다. 따라서, 역할을 나누고 해당 부분을 원래 있어야 할 부분으로 옮겨놓아야 한다. 직접 일하려고 하지 말고 시켜야 한다. 물론 일을 한 쪽(혹은, 디렉터리, 함수)의 책임이 따르지만, 직접 그 일을 처리하게 되면 복잡성이 높아지고 의존적인 코드가 되어버린다. 사실 이런 부분은 경험에 의존적인 부분이지만 시작은 단순한 것이 좋다. 단순히 긴 코드를 대상으로 짧은 조각의 코드들로 나누는 것부터 시작할 수 있다. 함수의 길이는 대략 100~200라인 정도가 좋다고 이야기들 하지만, 복잡하다고 생각되는 부분이나 논리적으로 연결이 약한 부분들을 분리해서 나누어 짧은 코드로 만드는 것이 좋다. 잘 정리된 코드를 받는 사람의 입장에서 생각해보면, 그들도 누가 그런 코드를 짰는지 궁금해할 것이다. 그 사람이 누구인지 알게 된다면 존경의 눈빛으로 보게 될 것이다. 그리고, 자신의 코드에도 그런 것들을 도입하려고 노력할 것이다. 가장 좋은 가르침은 직접 보여주는 것이다. 말은 눈앞에서 대화할 때 유효하고, 글은 시간이 지나면 해석이 달라질 수 있지만, 동작하는 코드는 언제든 실행시켜 결과를 확인해 볼 수 있다. 잘 정리된 코드를 만드는 것이 기능을 새로 추가하는 것보다 더 중요하다는 것이다.

# [ 테스트를 바라보는 관점 ]

하드웨어를 만들 경우를 생각해보자. 우리는 사용하는 칩(Chip) 하나 하나의 기능과 성능을 확인한 후 사용한다. 그리고, 칩들을 연결한 상태에서도 정확히 인터페이스를 맞추고 있는지를 확인하는 간단한 프로그램을 작성한다. 그렇게 연결된 칩들이 제 기능을 한다는 것이 보장될 경우 더 높은 상위 수준의 코드를 작성한다. 핵심은 하드웨어 들이 제대로 동작하는지 조금씩 테스트해 본다는 사실이다. 그럼 소프트웨어는 어떻게 만들고 있을까? 일단은 필요한 것들을 대략적으로 머리속에 구성한다. 그렇게 구성된 요소 중의 제일 하위에 있는 요소를 끄집어내서 알고리즘을 만들고 구현할 것이다. 인터페이스에 대한 설계는 아직 완성되지 않았지만 대략 필요하다고 생각되는 것들로 일단은 코드를 만들고 볼 것이다. 테스트하는 코드를 잠시 구현한 후, 호출해서 원하는 결과가 나오는지도 확인한다. 이제는 만들어진 인터페이스를 이용하는 더 높은 수준의 코드를 작성하고, 다시 잠시 테스트를 만든 후 동작하는지를 확인한다.

예상하지 못한 문제는 언제나 가장 마지막에 온다. 앞에서 든 소프트웨어 개발 방법에는 전혀 문제가 없어 보이지만, 부품들이 하나씩 조립되면서 더 복잡한 시스템으로 진화를 거듭하면서 문제를 점점 더 끌어들이고 있는 것이 사실이다. 즉, 테스트를 작성하는 것이 항상 구현보다 나중에 온다는 사실이다. 구현을 지속하면 하나의 함수가 점점 더 길어질 것이고, 중간에 다른 모듈의 통합에 있어서 필요한 요소들이 추가될 것이다. 파라미터의 수는 늘어나고 값의 범위도 점점 다양해진다. 이쯤 되면 자신의 코드에서 어떤 부분이 테스트가 되고 어떤 부분이 테스트가 되지 않는가는 뒷전으로 밀리게 된다. 기능을 구현하는데 모든 시간이 다 들어가고, 테스트는 점점 더 뒤로 미뤄질 것이다. 아마도 앞에서 작성한 테스트들도 이미 변경된 코드에는 적합하지 않을지도 모른다. 하나의 테스트 코드도 점점 더 복잡해지고, 결과적으로는 코딩의 스피드를 따라가지 못하게 된다. 검증받지 못한 코드들이 늘어남에 따라 시스템은 어떤 문제가 있을지를 알 수 없는 상태가 되고, 코드의 복잡함은 더 많이 지수적으로 증가할 것이다. 이제는 거의 테스트는 포기하고 기능 구현에만 매달리게 될 것이다. 버그는 나중에 테스터들이 알아서 찾아줄 것이라는 믿음만 굳혀가게 된다.

이런 식의 소프트웨어 개발이라면 일정, 비용, 품질을 제대로 다 만족시키기 어려워진다. 하지만, 현실의 개발자들은 대부분 이런 식으로 일한다. 테스트를 만드는 것이 별로 의미 없는 일이라고 생각하는 관리자들이 제대로 테스트 스크립트를 만들 시간적인 여유를 주지 않기 때문이며, 개발자 자신도 테스트를 어떻게 해야 할지 모르기 때문이다. 하지만, 세상에는 여러 유용한 도구들이 많이 있으며, 그런 도구들을 배척할 타당한 이유도 없다. “xUnit”이라고 불리는 것들을 사용하거나 그와 유사한 개념의 도구를 직접 만들어도 좋다. 어떤 코드가 테스트되는지를 알고 싶다면 “GCOV”나 “LCOV”를 사용할 수도 있다. 그리고, 메모리나 성능에 관련된 것을 이용하고자 한다면 “Valgrind”라는 것도 있다. 게다가 테스트를 위주로 개발하는 방법론인 TDD(Test-Driven Development)도 있다. 그렇다면 개발자들은 왜 그런 것에 전혀 관심을 가지지 않는 것일까? 이유는 단순하다. “모르기 때문이다.” 그런 것들이 왜 도움이 되는지 이해하지 못하기 때문이며, 그런 것들을 알려주고 함께 할 사람들이 부족하기 때문이다. 하지만, 이미 소프트웨어의 복잡도는 사람의 인지능력 수준을 넘어서고 있고 지속해서 빠르게 증가하고 있다. 이제는 더 이상 미룰 수 있는 여유가 없다. 시간이 부족하다면 도구를 믿어야 한다. 극적인 생산성의 향상이 있기를 처음부터 바랜다면 힘들 것이다. 하지만, 소프트웨어 개발자의 마음을 움직이고 싶다면, 그런 것들을 적극적으로 활용해서 품질을 끌어올리는 일이 왜 중요한지 인식시켜야만 한다.

개발자가 테스트하는 것을 고운 시선으로 바라보는 팀장은 별로 없다. 아무리 많은 테스트를 하더라도 걸러지지 않는 것들이 존재하기 마련이다. 팀장들이 개발하던 시기에는 그냥 시간을 무작정 투자하면 결과가 나오는 시대였기에 그들의 이해를 바래는 것은 의미 없다. 그들이 개발하던 시기의 소프트웨어들은 길어야 수만 라인이었지만, 지금은 수백만, 수천만 라인의 코드를 하나의 제품에서 다루어야 한다. 개발자의 규모도 커졌다. 따라서, 즉각적인 코딩에 대한 피드백을 얻을 수 없다면, 수 많은 실행 경로를 가지는 코드의 품질 수준을 끌어올리는 것은 불가능하다. 퇴근을 일찍 해서 집에 가서 사랑하는 애들과 아내, 남편과 행복한 시간을 보내고 싶다면, 이제는 일하는 방식을 바꾸어야 하는 것이다. 툴이 모든 것을 해결해 주지는 않지만, 최소한 “소프트웨어 개발의 안전망” 역할은 충분히 해줄 수 있다. 변경이 매일같이 발생하는 코드를 항상 안정된 상태로 만들기 위해서는 반드시 짧은 주기로 테스트를 실행해서 피드백을 얻어야 한다. 따가운 시선이 있다고 해서 자기 일을 미룰 수 없다는 것을 누구나 알 것이다. 그리고, 결과적으로 그런 안전망을 가진 소프트웨어 개발자와 그렇지 못한 개발자의 생산성은 시간이 흐를수록 점점 더 차이가 벌어질 수밖에 없다. “좋은 코드”를 만들고 싶다면 테스트하는 방법부터 배워야 한다. 그것이 학교에서는 잘 가르치지 않는 중요한 부분이지만, 회사에서는 꼭 필요한 부분이다.

# [ TDD(Test Driven Development)에 대해-03 ]

소프트웨어 개발자라면 TDD라는 말은 한 번쯤은들어봤을 것이다. 이런 저런 책을 뒤지기도 해보고, 혹은 남들이 써놓은 글을 읽어도 봤을 것이다. 하지만 도무지 이런 것을 실제로 접해볼 기회는 좀처럼 없었을 것이다. TDD를 했을 때 우리가 얻을 수 있는 이점이 무엇인지도 사실 알지 못하는 것이 사실이다. TDD가 줄 수 있는 가장 큰 혜택이라면, 아무래도 변화에 대한 두려움을 없앨 수 있다는 점이다. 소프트웨어 개발자의 하루 하루는 꾸준한 변경이 일어나는 일로 채워진다. 코딩을 조금이라도 한다면 당연히 전체 시스템에 대한 변경이 있는 것이다. 아무리 사소하고 부분적으로 변경했다고 하더라도 시스템 전체에 영향을 준다는 말이다. 그리고, 그런 변화 속에서 안정성을 찾아가는 방법을 만들어내는 것이 우리 소프트웨어 개발자의 몫이다. 마치 대단한 일처럼 들리지만, 불확실한 삶 속에서 보험을 들듯이 변경에 대해서 일종의 선 투자로 위험(Risk)을 줄이려는 것과 마찬가지일 뿐이다.

TDD를 시작하고 싶다면 최소 두 사람이 함께 하는 것이 좋다. 그리고, 그것을 지원하는 부서장이 있으면 더 좋겠지만, 부서장들에게 TDD를 이야기해봐야 아는 사람이 거의 없을 것이다. 그들에게 단위 테스트라는 것을 이야기해도 알지 못하는 경우가 많은데 TDD까지는 바라지 말아야 한다. 어쨌든 시도는 해봐야 하고 파일럿(Pilot)으로 삼을 만한 것을 정하고 작게 시작하는 것이 좋다. 그리고, 그것을 위한 도구에는 어떤 것들이 있는지도 조사해야 한다. C언어를 위해선 “Unity”나 “CppTest”, “GoogleTest”와 같은 것을 사용해도 좋을 것이다. Java나 C++, C#은 여러가지 다양한 도구를 가지고 있으니 알아서 찾아보기 바란다. 두 명 이상이 서로의 의견을 주고받으면서 테스트를 만드는 패턴을 공부하면서 시도한다면, 서로의 피드백이 모두의 역량 향상에 많은 도움이 될 것이다. 일단은 너무 큰 수준에서 한꺼번에 많은 일을 하는 것은 지양하자. 작은 수준에서 시작하는 것은 기존의 코드에 대한 유지보수를 위한 단위 테스트를 만들기가 어렵기 때문이기도 하다. 기존 코드들은 이미 상당히 복잡한 상태일 것이 분명하고, 의존성으로 인해서 단위 테스트 프로그램을 만들기에 쉽지 않을 것이다. 따라서, 새로 작성되는 코드를 대상으로 목표(Target)를 잡는 것이 합리적이다.

일정 지연의 대부분은 테스트가 미흡하다는 것과 관련이 있다. 학교에서는 가르치고나서 숙제의 결과만을 가져오기를 원하지, 코드의 품질이 좋은지 나쁜지는 따지지 않는다. 즉, 그 코드는 한 번만 사용되고 대부분 버려진다. 하지만, 회사에서 만드는 코드들은 한 번만 사용되고 버려지지 않는 것들이 대부분이다. 따라서, 결과뿐만이 아니라 코드의 내부적인 품질도 중요하다. 코딩의 가장 기본적인 요소는 “의존성의 분리(Decoupling)”와 “높은 응집성(High Cohesion)”이다. 따라서, 이렇게 만들어진 코드는 내부적으로도 품질이 우수하며, 외부적으로 보더라도 사용자에게 안정된 품질을 제공한다. 중요한 점은 TDD가 “약한 의존성”을 가지며, “응집성이 높은” 코드를 만든데 도움을 준다는 점이다. 높은 의존성을 가지는 코드는 테스트가 어렵기 때문에, 미리 테스트를 정의하고 테스트를 만족시킬 수 있도록 코딩한다면 의존성이 약한 코드가 만들어지는 것이다. 응집성이 약한 코드는 여러가지 일을 하기에 테스트의 통과 조건을 명확하게 만들어 해야 하는 일의 범위를 줄여줄 수 있다. TDD를 진행하는 과정에서 인터페이스는 정제될 것이며, 높은 자동화된 테스트 범위를 가지는 코드가 만들어져 버그도 줄어들 것이 분명하다.

유지보수에 있어서 들어가는 비용을 대부분의 개발자가 간과하는 경향이 있다. 즉, 유지보수 활동이 개발과 함께 시작한다고 생각하지 않는 것에서 문제는 출발한다. 이런 이유로 인해서 유지보수를 생각하지 않는 설계와 부족한 테스트 자동화로 이어지게 되고, 새로운 기능의 추가 및 버그의 수정 등에 들어가는 테스트 비용이 점점 커지게 된다. 또한, 유지보수를 별개의 조직이 할 경우에는 작성된 코드의 가독성에 심각한 영향을 주지만, 개발자들이 코딩할 때는 그런 것들에 제대로 신경을 쓰지 않는다. 따라서, 유지보수를 맡은 사람은 남이 작성한 읽기 힘든 코드를 이해해서 수정해야 하고, 자동화된 테스트가 없는 상황에서 수동으로 시스템에 버그가 없는지를 확인해야 한다. 고친 부분만 테스트하고 출시하면 된다고 생각하면 마음은 편하겠지만, 시스템은 아무리 작은 변화라 해도 항상 전체가 영향을 받는 경우가 많다. 특히 복잡하게 뒤엉킨 코드의 경우에는 변경의 영향도를 평가하는 자체가 불가능에 가깝다. 따라서, 모든 기능을 다시 테스트할 수밖에 없는 것이 숙명이다. 변경의 영향도를 파악해서 해당 부분만 테스트해도 충분하다고 믿는다면, 자신의 시스템의 안정성에 대해 지나친 상상을 하고 있는지도 모른다. 따라서, TDD로 만든 자동화된 테스트들은 시스템의 변경에 대한 보증서를 개발자들이 가지는 것과 같다.

# [ 결함은 개인의 문제? ]

예전에 이런 상사가 있었다. “지금까지 발생한 모든 결함을 조사해서 개인별로 결함 수가 어떻게 나타나는지 보고하고, 그 내용을 보고 결함을 많이 발생시킨 개발자에게 일종의 평가 불이익을 주겠다”고 메일을 보내왔다. 그렇다면, 결함을 발생시키는 것과 결함을 해결하는 사람을 어떻게 구분할 수 있을까? 그리고, 그 결함을 만든 요인이 담당자의 잘못일까 아니면, 결함의 결과만을 처리할 뿐 근본적인 원인은 다른 곳에 있는 경우에는 누구에게 책임을 물어야 할까? 이런 저런 고민 끝에 말도 안된다는 답장을 보냈고, 결과적으로 해당 상사의 꾸지람을 아침부터 1시간가량 들어야 했다. 소프트웨어 개발자라면 모두다 이해하겠지만 코딩을 많이 할수록 결함은 늘어나기 마련이다. 만약 결함의 수를 가지고 개발자를 평가한다면 과연 누가 코딩을 할까? 결함의 원인이 잘못된 소프트웨어의 구조로 인한 것이고, 그것을 제대로 확인할 수 없는 상황이라면 누구의 잘못을 탓해야 할까? 따라서, 탓해야 할 부분은 그런 것들이 용납되도록 내버려 둔 프로세스와 단계 진행에 대한 승인을 한 사람들의 몫이라는 것이다.

결함의 수를 가지고 개인의 생산성을 논하는 것은 웃기는 일이다. 물론, 결함이 없는 코드를 만들려고 노력은 해야 하지만, 한편으로는 그런 코드를 짤 수 있는 시간적인 여유를 주지 않는 것도 사실이다. 왜 그렇게 일을 하지 않느냐고 역설적으로 물어보는 상사들은, 자신이 과제 협의 시에 어떻게 과제 일정을 정했는지를 잊고 있다. "당겨"라는 한마디로 결함의 수가 늘어가리라는 것을 몰랐다는 말인가? 풍선효과와 같이 한쪽이 눌러지면 다른 한쪽이 부풀기 마련이다. 일정을 당기면 품질이 나쁘고 제대로 구현되지 않은 코드들이 만들어진다는 것을 배우지 못한 관리자가 분명하다. 생산성은 결함없이 동작하는 기능들을 가진 소프트웨어를 만들어내는 것이지, 누가 몇 개의 버그를 만들었냐는 지협적인 이야기가 되어서는 안 된다. 팀이 만들어낸 결함이 많다면, 팀이 과제를 진행하는 동안에 어떤 상황에 부딪쳤는가를 생각해 보아야 한다. 그들은 무리한 일정과 제대로 된 도구의 혜택이나 교육도 없이 무작정 시간에 쫓겨가며 야근과 특근을 했을 것이 분명하다. 그런 개발 현장이라면 품질은 그냥 깊숙히 처박아두고 다른 이야기를 해야 할 것이다.

결함은 개인의 문제가 아니기에 프로세스나 관리방법의 문제가 된다는 것은 이미 이해했을 것이다. 따라서, 우리는 프로세스의 개선과 관리방법이 개선되어야 한다는 것을 충분히 알게 되었다. 프로세스의 개선은 품질을 위주로 한 테스트와의 통합이 답이다. 이를 위한 방법으로는 "지속적인 통합(Continuous Integration)"과 "자동화된 테스트(Automatic Test)"가 과제 기간동안 지속해서 실행되어야 한다. 관리방법의 개선은 관리자의 인식 변화도 필요하다. 그들이 알고 싶어하는 것이 과제가 언제 끝나냐가 되어서는 안 된다. 물론, 과제 일정도 중요하다. 하지만, 더 중요한 것은 “언제 사용자가 필요한 기능이 완성되는가”이다. 둘 다 같은 이야기처럼 보일수도 있으나, 받아들이는 입장에서는 큰 차이가 있다. 전자는 과제를 전체적으로 하나로 보는 것이라면 후자의 입장에서는 과제가 지속해서 개선되고 진화되는 것으로 생각하기 때문이다.

(여기서 부터) 진화하는 과제라면 단번에 모든 것을 이루려고 하지 않는다. 지속해서 어떤 기능들이 필요한지를 고려하고 그것을 위한 노력을 들일 것이 분명하다. 한 번에 모든 것을 이루려는 생각은 그 과제 이후의 과정과 중간 과정은 중요하지 않다. 결과만을 중요시해서 내적 가치는 잊어버릴 것이 분명하다. 결국, 나오는 것은 자신의 임기 동안에만 적당히 유지할 수 있는 정도의 품질일 것이다. 물론, 자신이 회사를 그만둔 후란 생각하지 않는 것이 당연하겠지만.

관리 방법의 결함이나 프로세스의 결함은 그럼 어떻게 평가할 수 있을까? 프로세스의 결함은 병목(bottleneck) 현상이 생기는 부분이 어딘지를 알면 수정할 수 있다. 프로세스는 눈이 굴러가면서 뭉치듯이 몇 번의 반복을 통해서 향상된 기능과 품질을 제공해주는 구조가 대부분이다. 관리의 결함은 그럼 무엇일까? 가장 쉽게 알 방법은, 개발자들이 실제 일을 하는 것보다 다른 일을 하는 시간이 일과 중에 얼마나 되는가를 보는 것이다. 만약, 일과 중(8시간이라고 가정할 때), 4시간 미만으로 실제 일을 하고 있다면 잘못된 관리방법으로 지탱되고 있다고 봐야 할 것이다(즉, 50% 미만의 가동률을 보인다는 것이기에). 만약 하루에 10시간을 일하는데, 8시간 이상을 실제 업무에 쏟고 있다고 해도 문제는 있는 것이다. 즉, 이때는 개발자들이 너무나 많은 일을 하고 있다는 증거로, 일을 줄여야 한다는 뜻이다. 어쨌든 중요한 것은 관리 방법과 프로세스의 개선은 하지 않으면서, 모든 버그의 원인을 개발자의 몫으로 돌리려는 태도가 문제가 있다는 이야기다. 과제에는 개발자만이 참가하는 것이 아니다. 다양한 팀들이 상호작용을 하면 그들 모두가 과제의 성과라는 "열매"를 공유하기 때문이다. 성과는 탐이 나지만 직접 발을 담그고는 싶지 않다면 팀이라고 할 수 없다. 결함은 그들 모두의 노력이 결합하여야 줄어들게 되며, 결국 그들 모두의 책임으로 여겨져야 한다.

# [ "간소함"에 대해서 ]

"Simple is the Best"라는 말은 많이 들어봤을 것이다. 하지만, 정작 그 말은 들었지만, 그것을 정확히 이해하고 실무에 적용하는 것은 그렇게 "간단"하지 않다. "누구나 알아볼 수 있도록 코딩하라"고 이야기 하지만, 얼마나 더 쉽게 해야 "가독성"을 만족할 수 있는지에 대한 감도 오지 않는다. 말은 쉽지만 그렇게 하기는 정말 어렵다. 대부분 어렵다는 것은 해본 경험이 없다는 것과 일치한다. 즉, 이미 그 상황을 겪어봤다면 특별히 어려운 일은 없다. “어려울수록 자주 많이 해보는 것”이 해법이다. 마찬가지로 코드를 읽기 쉽게 만들고자 한다면, 그렇게 만들도록 더 많이 자주 노력하는 방법밖에 없다. 무작정 해보는 것이 아니라 체계적으로 계획을 세우고 해야 한다. 자신의 경험이 아무리 많다고 해도 남들이 이미 겪은 일을 다시 반복할 필요는 없기 때문에, 서점에 나와 있는 관련된 책을 기초로 "자주 많이" 해보는 것이 최선일 것이다.

실무에서는 같이 일하는 동료가 큰 도움이 될 수 있다. "코드 리뷰"를 할 경우 가독성을 위주로 작성된 모든 코드를 검토하는 것도 한가지 방법일 것이다. 좋은 코드를 짜는 방법을 규율로 정하고 이를 따르는 것도 좋다. 규율을 정하는 과정에서 만나는 사람들 간의 많은 대화가 더 좋은 방법을 만들어낼 것이기 때문이다. 이 때는 한가지 주의해야 할 것이 있다. 누군가의 일방적인 생각이 동의를 얻지 못하고 규율로 정해지는 것을 경계해야 한다. 간혹 팀에서 가장 직위가 높은 사람이 마음대로 규정을 정해서 전체 팀을 통제하려고 들기 때문이다. 팀에서 정하는 모든 규칙은 팀원들의 동의가 필수이며 일방적으로 강요한다고 지켜지는 것이 아니기 때문이다. 또한, 잘못된 습관은 고치기 어렵기에 더욱 주의해야 한다. 좋다고 이야기하는 것들을 배워서 자신의 팀에 맞게 적용하는 것은 가능하다. 이 때도 너무 많은 규정을 만들어서 오히려 역효과가 나는 것을 방지해야 한다. 너무 많은 규칙은 지키기 어렵지만 금방 잊혀버리고 말 것이다.

"간소함은 더할 것이 없는 게 아니라 뺄 것이 없는 것이다"라는 말에서 알 수 있듯이, 미래를 예측해서 미리 구현하거나 필요할 것처럼 생각되는 기능을 구현하는 것은 옳지 않다. 가능한 구현할 내용을 최소화하고 명확히 정해지지 않은 것들은 구현하지 않는 것이 좋다. 어차피 구현하더라도 전혀 사용되지 않는다면 구현할 이유 자체도 없다. 그런 것들은 그냥 시스템이 더 복잡해지게만 만들 뿐이다. 그렇다면 어떤 것들이 군더더기냐를 가려내는 시각이 필요하다. 가장 쉬운 방법은 코드에서 전혀 도움이 되지 않는 부분들을 삭제하는 것이다. 필요없는 주석과 주석화되 사용되지 않는 코드들이 대상이다. 길어진 함수는 짧은 함수들로 대체할 수 있다. 중복된 코드도 제거 대상이다. 파라미터의 수가 많다고 생각되면 해당 함수가 나누어질 수 없는지 다시 한번 생각해야 한다. 간소함은 약자를 사용하는 것이 아니기에 이해되지 않는 약자들은 좀 더 이해하기 쉽게 풀어서 써준다. 모든 상수값에는 이름을 주도록 한다. 이름이 의미하는 것이 한 번에 쉽게 이해될 수 있도록 만드는 것이 중요하다. 조건부 컴파일을 위한 매크로들도 정리해야 한다. 하나 둘씩 필요없는 코드들이 사라지고 나면 가치가 없다고 생각되는 기능들을 모아서 협상한 후 제거한다. 가능한 한 작은 프로그램을 짜야 하며 의도가 명확하게 드러나도록 코딩해야 한다.

"정리 정돈"은 모든 일의 기본이다. 작업 중에는 난잡하게 어질러 놓을 수 있지만, 마치고 나면 도구와 재료들은 원래의 위치를 찾아야 하고, 바닥에 떨어진 것들은 쓸어서 버려야 한다. 코드도 마찬가지다. 개발 중에 난잡하게 만들어놓은 것들은 동작이 검증됨과 동시에 "정리 정돈"되어야 한다. 기능이 동작하는 것만 확인하고 다음 기능의 구현으로 넘어간다면 청소할 것들이 나날이 쌓이게 된다. 나중에는 어떤 것을 정리해야 할지도 모르는 상황이 올 것이다. 그 때쯤 되면 더는 과제를 지연시킬 이유도 찾지 못할 것이다. "간소함"은 깨끗하게 정리된 상황에서 빛을 발하게 되는 것이지, 이미 엉망이 된 상황에서는 정리부터 시작해야 한다. 대부분의 이른바 "Legacy Code"들은 이런 상황일 것이다. 그리고, "도대체 누가 이렇게 개떡같이 짰어?"라고 동료들에게 이야기할 것이다. 만약 자신이 짠 코드가 남들의 손에 들어가서 똑같은 소리를 듣는다면 어떤 기분이 들까? 정말 정당한 이유로 만든 코드라고 아무리 이야기해봐야 변명으로 밖에 들리지 않을 것이다. 어떤 이유를 가져와 설명하려고 해도 읽기 불편한 코드는 "읽기 쉽지 않은" 코드일 뿐이다. 정리되지 않은 코드는 개발자의 게으름만 보여줄 뿐, 핑계를 댄다고 달라지는 것은 없다. 변명하지 말고 먼저 행동하는 것이 유일한 해결 방법이다.

# [ 코딩에 대한 단상 ]

코딩은 프로그래머의 생각을 표현하는 한 가지 방법이다. 코드 이상의 확실한 전달 방법은 없다. 동작하는 소프트웨어를 만들어 보여주는 것이 가장 확실하다. 하지만, 문제도 역시 코드에 있다. 코드가 동작은 하지만 이해하기 어려운 부분들이 많다는 점이다. 동작하는 코드만을 믿어야 하지만, 코드의 내용이 알아보기 힘들 정도로 되어 있다면 당연히 문서를 먼저 보려고 할 것이다. 코드를 반영하는 것이 문서이기 때문이다. 여기도 문제는 있다. 코드는 실행할 수 있지만, 문서는 실행할 수 없다는 점이다. 따라서, 문서가 제대로 되어있는지를 검증할 방법이 없다. 문서는 코드를 완전히 반영하지 않을 수도 있으며, 문서의 내용이 잘못된 경우도 있다. 그럼, 어떻게 해야 할까? 가장 쉬운 방법은 코드를 다듬는 것이다. 물론 알아보기 쉬운 코드로 만드는 것이 핵심이다. 그리고, 추가로 해야 할 일이 있다면, 코드를 설명하는 문서를 너무 세세하게 적기보다는 의도와 결정의 이유를 표현하고, 대략적인 프로그램의 구조를 설명하는 것이다.

문서와 코드를 일치시키는 것은 항상 많은 노력을 요구한다. 자동화하기 위해서 "Doxygen"과 같은 도구를 써보지만, 실제로 코드와 문서를 완전히 일치시키는 일은 생각보다 많은 시간이 필요하다. 지속적인 수정이 있는 코드와 “Doxygen”에서 사용하는 주석을 같이 수정하는 것은 항상 귀찮은 일이다. 따라서, 강조하고 싶은 것은 코드 자체의 가독성을 높이는 방법이다. 사실 코드의 가독성은 많은 문제의 근본 원인을 제거할 수 있는 좋은 방법이다. 같이 일하는 사람들이 버그를 해결하는 시간을 오래 걸리도록 만들기도 하고, 새로운 기능을 추가할 경우에도 많은 버그를 발생시키는 원인이 될 수도 있다. 유지보수로 가면 말할 필요도 없다. 자신이 만들지 않은 코드를 개선하는 사람들에게는 "가독성"은 반드시 지켜야 할 생명과도 같다. 하지만, 실무 개발자들은 너무나 “쉽게(?)” 코드를 짠다. 그들이 생각이 없어서 그렇게 짜는 것은 아니며, 오히려 생각을 너무 많이 하기 때문이다. 예를 들어, 하나의 함수를 간단하게 짜고 나서 나중에 다시 기능을 조금 변경하는 경우가 발생할 경우, 새로운 함수를 만들고 리팩토링하는 대신, 이미 있는 함수를 재활용하려고 할 것이다. 조건문이 추가되고 함수로 전달되는 파라미터의 수가 늘어난다. 그에 따라 필요한 코드들이 함께 늘어나게 되며, 점차 그 함수는 여러가지 일을 한번에 처리하는 "다재 다능한 함수"가 될 것이다. 이것이 바로 함정이다.

많은 일을 하기보다는 한 가지 일에 충실한 코드를 만들어야 한다. 복잡할수록(길어질수록) 문제가 일어날 가능성은 높아진다. 간단하게 구현된 물건들은 잔 고장이 적다. 코드도 마찬가지다 단순하게 구현할수록 버그가 발생할 확률이 낮아진다. 단순하게 구현한다는 말은 무조건 짧게 구현한다는 뜻이 아니다. "단순함"이란 모든 복잡해 보이는 사물의 표면에 숨겨진 깊은 내부에 반드시 갖추어야 필수적인 항목이다. 소프트웨어와 같이 복잡성을 다루어야 하는 분야에서는 반드시 익혀야 하는 기본기다. "단순한 구현"이란 짧은 구현이 아니라 간략한 구현이라는 뜻이다. 짧은 구현을 한다고 이런 저런 말도 안 되는 문법을 사용하는 것보다, 남들이 보기에 명료하게 의도를 이해할 수 있도록 구현되어야 한다는 것이다. 그리고, 꼭 필요한 만큼만 구현되어야 한다는 점이다. 미래를 예측한 구현보다는 필요할 때 변화를 가져올 수 있는 것을 말한다. 이런 것들을 익히기 위해서는 인터페이스(Interface)를 어떻게 간단히 만들 수 있는가를 고민해야 한다. 인터페이스가 단순할수록 그 너머에 있는 구현(Implementation)은 관심(Concern)밖이 될 수 있기 때문이다. 이 말은 인터페이스가 충분히 원하는 것을 이룰 수 있도록 만들어주기에, 그 너머에 무엇이 있어도 상관없다는 뜻이다. 인터페이스를 기준으로 서로의 약속이 충실히 이행된다면, 변화에 대해서도 "유연한(Flexible)" 코드 구조를 만들어낼 수 있다.

코드를 단순화시킬 목적으로 "테스트주도 개발(TDD: Test Driven Development)"을 도입하는 것도 좋은 선택이다. 이 방법의 핵심은 "필요한 만큼만 코딩한다."와 "테스트할 수 있는 코드를 만든다."이다. 테스트를 통과한 코드의 "외부에 보이는 행위(Behavior)는 변경하지 않으며서 내부적인 구조를 개선한다(Refactoring)."라고 요약할 수 있다. 모든 코드의 개선 바탕에는 테스트가 깔려있다는 것을 알고 있을 것이다. 테스트가 있지 않으면 구조 개선의 결과를 확인할 수 없기 때문이다. TDD에서도 역시 이 부분이 중요한 활동이다. 개발자가 자신의 코드를 즉시 테스트할 수 없다면 개선을 이룰 방법이 없다. 결과에 대한 피드백(Feedback)이 나오는 시기가 늦어질수록 코드는 점점 더 소프트(Soft)해 지지 않고 하드(Hard)해 질 것이기 때문이다. 어쩌면 코드 변경 자체가 불가능하다고 생각될 만큼의 큰 비용이 들 수도 있다. 만들어내는 가치보다 더 큰 비용을 요구한다면 누가 개선하려고 하겠는가? 결국 "변경은 비용을 동반하는 가치 창출의 방법"이다. 하지만, 한번 잘 못 만들어진 코드는 변경해도 창출할 수 있는 가치는 제한된다. 왜냐하면, 이미 만들 수 있는 가치보다는 더 큰 비용만 청구할 것이기 때문이다. 제한된 가치라도 만들어내면 다행이지만 그렇지 못한 경우도 많다. 이런 모든 것들의 기본이 코드를 만드는 개발자의 손끝에서 결정된다고 본다면, 우리의 하루 하루의 생활이 그리 의미 없는 일은 아니다. 우리의 사소한 손가락 움직임이 세상을 바꿀 수도 있으며, 미래를 아직 열린 문으로 남아있도록 만들수도 있기 때문이다.

# [ 테크닉 vs. 원리 ]

소프트웨어 개발자들은 자신이 가진 코딩 기술(Skill)을 뽐낼 수 있는 경험을 중요하게 생각한다. 일종의 테크닉이라고도 볼 수 있는 “좋은 코드”를 만드는 일반화되지 않은 자신만의 기술이라고 믿는다. 이론을 다루는 사람들은 원리가 중요하다고 생각한다. 원리를 알면 문제를 쉽게 푸는 방법을 알 수 있다고 이야기한다. 사실 둘 다 중요하다. 원리만 가지고 좋은 코드를 짤 수는 없고, 원리를 구현하는 좋은 코딩 방법이 동반되어야만 멋진 소프트웨어를 만들 수 있다. 학교 숙제를 잘 한다고 해서 회사에서 코딩을 잘하는 것은 아니지만, 학교 숙제라도 잘해야 기본적인 코딩이라도 할 수 있다. 원리란 문제 해결의 바탕이 되는 것이고, 이를 현실에 표현하는 것이 코딩이다. 따라서 어느 한 쪽만 더 중요하다고 볼 수는 없다. 코딩은 현실의 문제를 푸는 방법이며, 컴퓨터상에서의 실제로 동작하는 것을 전제로 한다. 사용하는 언어 및 컴파일러, 각종 라이브러리 및 프레임워크, 운영체제 등 다양한 부분이 상호작용해서 실행 가능한 프로그램이 만들어진다. 원리는 이들이 상호작용하는 이론적인 배경이 되며, 그런 것들을 이해하지 못하면 최적화된 시스템을 만들지 못한다.

코딩을 잘한다는 말을 듣는 친구들은 이론이나 영어, 기타 학과 수업을 게을리하는 경우가 많다. 모든 대단히 유명한 소프트웨어 개발자가 좋은 대학을 나온 것은 아니지만, 더 성장하기 위해서는 코딩만 익힌다고 최고가 되는 것은 아니다. 이론은 성장의 바탕이 되며, 영어는 세상을 살아가는 기초가 된다. 기타 학과 수업들은 남들이 무엇을 했는지 이해할 수 있는 도구가 될 수 있기에, 최고는 아니지만 최선은 다해야 한다. 모든 것을 다 잘하기는 어렵다. 하지만, 어떤 것들이 있는지 정도는 최소한 알아야 하지 않을까? 나중에 필요할 경우 그것이 무엇인가를 찾을 수있을 정도는 되어야 한다. 코딩만이 천직이라고 생각한다면 그렇게 살아도 무방하다. 마흔이 되고 오십이 되어도 코딩을 할 수 있는 회사를 찾으면 될 것이다. 하지만, 현실은 어떨까? 아마도 그 때쯤 되면 코딩보다는 남들을 가르치거나 리더로서의 역할을 해나가고 있을 것이다. 뭔가 부족한 리더보다는 능력있고 비전을 제시할 수도 있어야 할 것이다. 비전이란 나름의 판단 기준으로 만든 "통찰"이라고 볼 수 있다. 앞으로 어떻게 될 것이고, 그것을 위해서 현재 무엇을 할지를 제시할 수 있는 사람이 되어야 할 것이다. 비전을 제시하지 못하는 과제는 하고 싶어하지 않을 것이며, 그런 과제를 이끌고 있는 리더는 당연히 인기도 없을 것이다.

이론만을 강조한다면 세상은 고고한 학처럼 사는 사람들로 붐빌 것이다. 하지만, 우리 주변에는 허드레 잡일이라고 생각되는 것들도 열심히 하는 사람들이 있다. 물론 그들에게는 그냥 주어진 일일뿐이다. 세상은 이론만으로는 돌아가지 않는다. 이론을 만들기 위해서 훨씬 오래전부터 이미 현실이라는 것이 존재했으며, 이론을 검증 위해서도 실행은 필요하다. 하지만, 몸을 움직이고 손으로 하는 일에 대해서 우리는 낮은 수준이라고 이야기하는 것을 종종 볼 때가 있다. 그런 생각은 옳지 않다. 누군가는 변화를 위해서 굳은 일을 하는 것일 뿐이다. 우리 주변에는 이런 저런 이론을 떠들거나 퍼 나르는 사람들이 많다. 그런 사람들이 늘어날수록 변화는 오히려 더디다. 누군가는 실제 행동으로 옮겨야지만 변화가 생긴다. 아무리 많은 생각도 그것이 현실에 발판을 두지 않는다면 공상에 불과할 뿐이다. 물론 그런 비난을 듣는다면 무지하다고 할지도 모른다. 하지만, 아는 것이 많기보다는 작은 것이라도 실천하는 삶이 더 위대하다. 그 자그마한 실천으로 인해서 오늘의 세계가 내일의 세상으로 바뀌고 있기 때문이다. 누군가는 오늘도 자신의 손으로 직접 그런 것들을 실천하고 있기에 사회가 유지되고 발전하는 것이다.

자그마한 변화라도 꾸준하게 이루어진다면 결국에는 큰 변화가 발생한다. 큰 변화를 만드는 것은 사소한 작은 것들이지만, 그것이 방향성을 가지고 지속해서 생긴다면 결과는 무시하지 못할 수준이 될 것이다. 어쩌면 새로운 이론을 만들어야 하는 거대한 압력으로 작용할 수도 있을 것이다. 한 번에 거대한 변화가 일어나기보다는 작지만 꾸준하게 진행되어야 한다. 이런 것들은 개인의 하루 하루의 삶에 관련된 것이며, 말로만 떠드는 일이 아니라 자신의 손을 직접 건네는 일이다. 비난부터 하는 일이 아니라 도움을 주는 것으로부터 시작된다. 누군가의 실수를 나무라기는 쉽지만, 잘못을 감싸고 돕는 것은 어려운 일다. 남을 돕는 것이 자기 일이 아니라고 생각한다면, 비난할 것은 남이 아니라 자기 자신일 뿐이다. 어쨌든 당신이 속한 조직의 시스템이 실수가 발생하는 것을 사전에 막지 못했기 때문이다. 당신이 미리 고치지 못한 시스템의 오류를 그 사람으로 인해서 찾았냈기 때문이다. 따라서, 버그를 수정하는 것은 실수를 유발한 사람과 당신의 몫이다.

작은 변화를 일으키기에 필요한 것은 당신의 손과 머리다. 둘 다 제대로 갖추어지지 않은 상태에서 하는 일들은 당연히 안정적이지 않다. 어느 한 쪽만을 가치 있게 본다면, 실체와 본질을 분리하려는 생각일 뿐이다. 둘은 항상 함께해야 제대로 된 결과물을 만들 수 있다. 원리를 벗어난 테크닉은 시행착오를 낳으며, 실용성없는 이론은 공염불에 불과하다. 이론과 그것을 가능하게 만드는 테크닉은 서로를 떠 받치고 있는 두 개의 축이며, 어느 하나만 신경쓴다면 나머지 하나는 쥐죽은듯 조용하게 지낼 것이다. 테크닉만 강조하는 개발자는 스스로 파놓은 함정에 빠져서 허우적거리며 시간을 보내게 되고, 이론만 강조하면 실제 동작하는 결과물은 제대로 만들어내지 못하게 된다. 서로가 서로에게 도움을 주고받을 수 있는 관계를 만들어야, 시스템의 개선과 안정을 가져오게 만들 수 있을 것이다.

# [ 생각을 정리한다는 것 ]

사실 대단한 것은 아니지만 일을 잘하는 사람들의 특징을 보면 정리를 잘한다. 청소를 잘하는 사람도 대체로 일을 잘하는 사람들이다. 항상 자신의 자리를 떠날 때는 정리정돈하는 습관을 지니고 있다. 컴퓨터에 들어있는 데이터들도 날짜나 검색을 효과적으로 할 수 있는 나름의 구조화 방법을 통해 각종 도구를 이용해서 정리해 둔다. 생각이 끝없이 이어지는 경우에는 그것을 잘 정돈된 형태로 만들어두는 마인드맵(Mindmap)을 사용하고, 일정을 정리하는 용도로는 스케줄러 프로그램을 주로 이용한다. 회의 시간은 언제나 5분 전에 먼저 도착하는 것도 다 이유가 있다. 그들은 항상 무리하지 않고 언제나 여유 있는 일정을 만든다. 그것이 과제이건 일상의 일이건 상관없이 항상 여유를 행동에 반영한다. 여유를 가져다주는 것은 주어진 일을 하는 것이 아니라, 자신만의 방식으로 일을 정리하기 때문이다. 물론, 항상 여유있는 일정으로 일을 할 수 있는 상황이 아닐수도 있지만, 그렇다고 정리 없이 무작정 다음으로 넘어가지는 않는다.

생각은 정리되지 않으면 흩어지고 만다. 아무리 좋은 생각이라고 그것이 구체적으로 표현되지 않으면 잊혀지고 만다. 대단한 것들은 아니지만 블로그에 적어본다거나 항상 가지고 다니는 다이어리에라도 끄적여 놓으면, 나중에 그와 관련된 것들이 꼬리를 물고 생각이 지속해서 이어질 수 있다. 그리고, 그런 생각들이 관련을 맺어가면서 좀 더 명확한 이미지로 머리속에 각인된다. 형태가 없이 정리된 것들은 단순한 잡생각일 뿐이다. 마찬가지로 일정을 정리하는 것도 점점 더 복잡하고 많은 인간관계를 맺어가기 위해서는 필수적인 요소다. 이런 것들을 강박관념에 가깝게 하는 것은 보기 좋지 않지만, 그냥 필요로 하는 정도라면 크게 어려운 일이 아니다. 생각은 한번 떠올랐다가는 금방 사라진다. 좋은 생각은 아무 때나 나오는 경우가 많기에, 언제라도 기록할 수 있는 준비가 되어 있어야 한다. 기록이 어렵다면 녹음이라고 하는 것이 좋다. 사라지는 생각들을 붙잡아 두기 위한 어떤 방법도 다 사용할 수 있다. 문제는 기록하고 저장하는 매체가 아니라 습관이다. 습관적으로 이런 일들을 잘하는 사람들은 남들로부터 신뢰를 얻고, 결국 자신이 이야기한 것에 대해서는 반드시 끝낼 것이다. 정리되지 않는 삶은 바쁘기만 할 뿐 지향하는 목표가 뚜렷하지 않다. 무엇에 더 선택과 집중을 할지를 모르기 때문이다.

어떤 책의 한 구절을 읽었는데 제대로 이해하지 못하다가, 시간이 한참 흐른 후에 자신도 모르게 그 뜻을 문득 깨닫게 되는 경우가 있다. 이것은 기억이라는 것이 단순히 시간에 관련된 것이 아니며, 전체적으로 한꺼번에 동작한다는 사실을 보여주는 증거라고 생각된다. 공부를 잘 하는 방법을 알면 오랜 시간을 수고하지 않아도 우수한 성적을 거둘 수 있다고 이야기한다. 하지만, 이것은 사실 몇몇 사람에게만 해당할 뿐이다. 대부분 사람은 몇 배로 열심히 해야 겨우 따라갈 수 있는 성적을 만들 수 있다. 생각을 정리하는 것은 이런 것에도 도움을 줄 수 있다. 열심히 머리 속에 기억하기를 반복하는 것이 아니라, 일부를 정리된 기록으로 남겨두고 요약하는 것이다. 코딩도 마찬가지다. 난잡한 코드를 만드는 것도 정리되지 않은 생각을 코드에 반영하기 때문이다. 만약 떠오르는 생각을 정리해서 깔끔하게 만든 후에 코딩 한다면 충분히 좋은 코드가 나올 수 있다. 하지만, 대부분은 한번 읽은 책은 던져두고 보지 않듯이, 한번 만든 코드에 대해서는 정리하지 않는 버릇이 있다. 공부를 잘하거나 회사에서 우수한 사원이라는 소리를 듣고 싶다면, 자신만의 정리 방법이 필요하다. 어떤 누구라도 정리되지 않는 노트를 빌려보고 싶어하지 않으며, 이해되지 않는 소스코드 때문에 불편을 겪고 싶지도 않기 때문이다.

생각을 정리할 때 가장 도움이 되는 것은 형태(Shape)를 갖추는 것이다. 형태란 모습과 상대적인 위치가 결정된다는 것이다. 모습은 그것이 나타내는 의미를 보여주고, 위치는 그것이 어떤 문맥(Context)에서 사용되는지 알려준다. 따라서, 이 두 가지가 결합하여야 생각이 명확해질 수 있다. 코드는 어떻게 정리할 수 있을까? 코드는 역할(Role)을 단위로 군집을 이룬다. 모듈은 그러한 역할을 모아서 한 가지 개념으로 만드는 단위다. 모듈도 그것이 사용되는 문맥을 가질 수 있으며 배치되는 위치가 중요하다. 잘못 배치된 모듈은 연결을 어색하게 만들고 생각을 끊어지게 한다. 이런 느낌이 든다면 일단 코딩을 멈추고 구성 요소간의 역할 관계를 정리할 생각을 해야 한다. 아키텍처라는 거창한 말을 사용하지 않더라도 부자연스러운 것들을 줄이려고 노력해야 한다. 한 번 두 번 이런 것들을 그냥 넘어가면, 하나씩 쌓이는 쓰레기로 뒤덮인 코드를 보게 될 것이다. 조금만 시간이 지나면 악취를 풍기게 되고, 결국에는 치우기도 부담스러운 수준으로 발전하게 된다. 쓰레기와 코드가 다른 점은 코드는 그냥 버릴 수 없다는 점이다. 안타깝지만 코드는 악취가 나더라도 계속 사용해야 한다. 그에 따라 악취는 점차 더 심해져만 갈 것이다. "정리정돈"은 일상의 습관인 것처럼, 자신의 힘으로 매일 하지 않으면 안 된다. 지겹지만 그렇다고 내일로 미루어서는 결코 안 되는 일이다. 생각이 정리되지 않은 상태에서 만들어진 코드들은 매일 정리정돈하는 것 만이 유일한 해결 방법이다.

# [ 소프트웨어 설계의 기본이란? ]

소프트웨어를 설계하는 일은 성능만 최고로 발휘하는 것을 목표로 하지 않는다. 성능도 가져야 할 목표 중 한 가지일 뿐이며, 그것으로 인해 모든 다른 목표들을 무시해도 된다는 것은 아니다. 설계에서는 흔히 “품질속성(Quality Attribute)”이라고 불리는 다양한 요소들이 성능과 대등한 수준에서 다루어져야 한다. 여러가지가 있겠지만 일반적으로 소프트웨어를 만들 때 중요한 품질속성은 사용자가 소프트웨어를 어떻게 보는가에 대한 것이다. 이 때도 물론 성능과 같은 것이 중요한 요소가 될 수는 있다. 즉, 사용자가 느끼는 외적으로 표현되는 품질이기 때문이다. 하지만, 내적인 품질속성으로 중요하게 다루어야 할 기본적인 것들은 “가독성(Readability)”과 “테스트 용이성(Testability)”이다. 내부적인 품질은 소프트웨어를 지속해서 개선하는 데 도움이 되는 것들이 주를 이룬다. 한 번만 사용하고 버려질 것이라면 이런 것들을 신경 쓰지 않아도 된다. 학교에서 숙제로 만드는 프로그램들이 아마도 이런 부류에 속할지도 모른다. 데모용으로 만드는 것들도 마찬가지다. 하지만, 정말 그렇게 한 번만 쓰고 버려지게 될까? 습관이란 무서운 것이다. 한번 잘 못 하기 시작하면 다시 되돌리기가 쉽지 않다. 모든 소프트웨어는 제대로 만드는 것이 정답이다. 제대로 만든다는 것은 생각을 먼저하고 행동으로 옮긴다는 말이다. 생각에 필요한 것은 결정의 근거를 명확히 하는 것이다. 논리적인 해답을 찾아 나가는 연습은 그것이 학교 숙제이건 작은 데모를 위한 것이건 상관없이 항상 중요하다. 습관이 작은 행동의 무의식적인 반복 연습으로 만들어지기 때문이다.

좋은 설계는 변화에 유연해야 한다. 개발자라면 소프트웨어는 언제든 변경될 가능성이 있다는 것을 잘 알 것이다. 따라서, 소프트웨어를 정말로 "소프트(Soft)"하게 만드는 데 익숙해져야 한다. 소프트웨어라는 말을 쓰면서도 실제로는 “소프트”하지 않은 설계들이 많다. 변경은 가능하지만 많은 고통을 감내해야 하는 경우도 있다. 소프트웨어가 변경이 쉽다고 비용이 들지 않는 것은 결코 아니다. 때로는 변경 비용이 벌어들이는 수익보다 훨씬 클 수도 있다. 미리 변경을 대비해서 설계된 소프트웨어는 당연히 비용도 크지 않다. 사실 소프트웨어를 변경하는 것은 정말 어려운 일 중 하나다. 조그만 변경도 시스템의 전체에 영향을 줄 수 있기 때문이다. 변경의 영향도를 측정하고 변경의 영향을 받는 부분만 테스트하면 된다고 생각했다면 잘못된 생각이다. 아무리 작은 변화라고 하더라도 시스템 전체에 영향을 줄 수 있다. 따라서, 변경되기 전의 시스템의 모든 동작은 변경 후에도 동일하게 유지되어야 한다. 이를 위해서는 자동화된 테스트가 반드시 있어야 할 것이다. 만약 자동화된 테스트가 없다면, 변경은 시스템의 품질을 떨어뜨리는 주요 원인으로 작용하게 될 것이다. 변경 후에도 전체 시스템이 동일한 동작을 한다고 보장할 방법이 없거나, 실수가 잦고 비용이 많이드는 테스트를 수행해야 하기 때문이다. 소프트웨어 개발자라면 이런 것들을 이미 많이 경험했을 것이다. 이유는 다양하지만 결국 한 가지로 요약할 수 있다. 우리는 아직도 제대로 소프트웨어를 만드는 방법을 알지 못한다는 것이다.

버그를 제거하기 위해서는 코드를 구성하는 최소 단위가 실행되어야 한다. 소프트웨어가 아무리 크고 복잡하더라도, 최소 구성 단위는 함수나 모듈들이다. 그런 단위들 각각이 테스트를 위해서 빌드되고 실행될 수 있도록 만들고, 그것들을 다시 조합해서 더 큰 테스트 단위로 만들어야 한다. 이를 지속해서 더 큰 단위로 만들어가면 결국 만들고자 하는 소프트웨어 전체가 테스트 단위가 될 것이다. 단위 테스트란 코드를 구성하는 최소의 실행 단위에서 테스트를 실행하는 것이며, 그 각각이 제대로 수행되는가를 검증하는 활동이다. 소프트웨어의 설계란 전체 시스템에 대한 전략을 짜는 것과 각각의 단위를 위한 전술적인 것으로 나눌 수 있다. 전체 전략을 짜는 것은 대부분 구현의 초기에 필요한 활동이며, 각각의 전략적인 결정에 대한 세부 사항을 전술로 처리하게 된다. 전쟁에서는 전략이 중요하다는 것을 알 것이다. 하지만, 전술은 대부분 개성에 맡긴다. 하지만, 자신의 스타일이 최고라고 자만하지는 않도록 해야 한다. 언제나 더 좋은 것은 항상 있기 마련이다. 그리고, 상황이라는 것이 새로운 변수로 작용할 수 있다. 전술은 코딩 역량에 의존해서 개인간에 많은 차이를 보이지만, 전체적인 성능보다는 국지적인 성능 개선에 집중하는 경향이 있다. 물론, 이것도 무시할 수는 없지만 전체적인 성능개선을 위해서는 소프트웨어의 아키텍처(구조)를 만드는데 집중하는 것이 좋다. 직접적인 코딩에 영향을 주는 것은 코딩 역량이며, 이를 위해서는 테스트하기 쉬운 함수를 만드는 것이 무엇보다 중요하다. 대부분 개발자들은 아직도 제대로 테스트하는 방법을 모르고 있으며, 단위 테스트보다 전체 코드가 통합된 상테인 시스템 수준의 테스트에 치중할 뿐이다.

소프트웨어 설계의 기본이란 단일한 임무를 수행하는 함수들의 모음인 모듈을 제대로 만드는 것이다. 모듈의 외부와 내부에 대한 연결을 정의하고, 데이터의 흐름을 통해서 어떤 가공 과정이 필요한지를 나열하게 된다. 그 가공 과정에 필요한 절차를 정하고, 정해진 절차를 위해서 필요한 하부 구성요소들을 다시 정의한다. 하부 구성 요소들 간의 제어 및 데이터 흐름도 연결된 형태로 표시하고, 데이터를 위한 자료구조를 만들게 된다. 의존 관계를 표현하고, 중복된 역할을 하는 부분이 없는지 검사하게 되며, 공통된 부분을 발견하면 묶을 수 있는지를 검토한다. 너무 큰 덩어리가 있다면 더 잘게 쪼갠다. 대체로 큰 것보다는 작은 것이 오류를 발생시킬 가능성이 낮고 테스트하기도 쉽다. 또한 작은 것들이 재활용하기도 편하다. 테스트를 쉽게 하기 위해서는 코드의 내부에서 발생하는 의존성을 최소화시키고, 외부에서 의존성을 주입(Dependency Injection)할 수 있도록 만들어야 한다. 외부에서 주입된 의존성은 테스트를 위해서도 사용될 수 있으며, 테스트 대상이 되는 코드의 의존성 자체를 분리하는 방법으로도 사용할 수 있다. 그렇지 않다면 의존하고 있는 코드도 테스트 일부로 동작해야 하기에 검증 단위가 커지게 된다. 검증 단위가 커진다는 말은 테스트해야 할 경우의 수가 많아진다는 뜻이며, 테스트되지 않는 코드가 늘어날 가능성도 커진다는 의미다. 따라서, 이런 부분들을 고려했을 때, "일상적인 설계 개선”도 지속하는 것이 좋다. 좋은 설계란 큰 조각들과 작은 조각들이 어떤 크기에서 보더라도 고른 특성(크기, 추상화, 역할 등등)을 가지도록 만드는 것이다.

# [ 의존성이란? ]

소프트웨어가 가지는 다양한 의존성(Dependency)은 개발 속도를 줄이는 마찰력으로 작용한다. 의존성은 정보의 공개를 얼마나 해야하는가에 영향을 준다. 더 많은 상대방에 대한 정보가 필요하다면 더 높은 의존성을 가지게 된다. 물론 이러한 정보에 데이터만 들어가는 것은 아니다. 시간적인 의존성 및 제어에 대한 의존성 등도 포함된다. 의존성이란 것을 함께 묶어서 생각해야 한다는 의미로 "커플링(Coupling)"이라고 부르기도 한다. 하지만, 의존성이 완전히 커플링과 같은 것은 아니며, 변경에 대한 영향을 받는가를 기준으로 의존성과 커플링을 나누기도 한다. 의존성은 하나의 모듈(여기에서 모듈은 독립적인 일을 수행하는 단위라고 볼 수 있다.)이 다른 모듈을 어떻게 이용하는지를 보는 것이고, 커플링은 의존성 중에서도 영향이 미치는 범위를 보는 것이다. 한 모듈이 다른 모듈을 사용할 때 당연히 모듈 간의 의존성은 존재한다. 하지만, 커플링이 되지 않도록 인터페이스(Interface)정의하고 사용할 수는 있다. 의존성이 지나치게 크게되면 모듈 간에는 커플링이 존재하게 되며, 하나의 모듈에 대한 변경이 해당 모듈을 사용하는 연결된 모든 모듈에 영향을 주게 된다. 이 때는 변경에 대해서 모듈들이 서로 자유롭지 못하게 되어 버그를 발생시킬 가능성이 커진다. 의존성은 뜻하지 않은 오류를 발생시키며 찾기 어려운 버그를 만들어 낸다. 따라서, 일의 경계에서 발생하는 버그를 줄이는 것이 일정지연을 줄여줄 수 있다.

의존이란 일의 책임이 누구에게 있는가를 보는 것이다. 하나의 모듈은 하나의 역할을 가지는 것이 옳다. 만약 여러 가지 임무를 수행할 필요가 있다면 분리된 모듈로 나누어야 한다. 물론 더 상위의 모듈은 분리된 모듈들을 이용해서 더 높은 추상화 수준에서 한 가지 책임과 역할을 질 것이다. 어쨌든 모듈을 나누는 것은 "역할과 책임(Role & Responsibility)"이라는 것을 명확히 해나가는 과정이라고 볼 수 있다. 모호한 책임을 지고 구현할 수는 없다. 그리고, 모든 역할을 한 모듈로 모으는 것도 어리석은 생각이다. 모듈 이름이 "Common"이나 "Base", "Help"와 같은 것들은 역할이 모호할 가능성이 높다. 그런 모듈들은 하는 역할을 분리한 것이 아니라, 작다고 생각되는 역할들을 한 곳에 모아서 관리하겠다는 뜻으로 해석할 수 있다. 이 때 발생할 수 있는 문제는 해당 모듈에 의존하는 모듈들이 많아져 전체 시스템의 유연성에 병목지점이 된다는 것이다. 구분되지 않은 역할을 하는 모듈은 당연히 복잡도가 높을 것이고, 모듈 내부에 있는 대부분의 함수가 길어질 것이다. 긴 함수들은 내부에 여러 가지 역할이 혼재한다는 의미이기 때문이다. 그런 함수들이 만들어진다는 것은 전달되는 파라미터(Parameter)들의 수도 여러 개가 되고, 각각의 파라미터들이 가지는 값의 범위와 형태도도 다양하다는 것이다. 즉, 역할의 분화가 되지 않아서 다양한 조합의 충분한 입력을 얻어야 하기 때문이다.

호출의 의존성은 이미 잘 알 것이다. 즉, 함수의 호출값과 복귀값에 대한 의존성이다. 호출하는 함수가 호출되는 함수를 자세히 알지 못해도 사용할 수 있는 것이 좋다. 파라미터의 수는 줄이는 것이 좋고, 그것이 가지는 값도 한정된 범위가 좋다. 그리고, 부작용(Side-Effect)이 없는 함수를 만들어서 제공해야 한다. 복귀값도 호출하는 함수의 제어를 변경시켜서는 안 된다. 많은 데이터를 파라미터로 넘겨서도 안 된다. 필요한 정보만 서로 주고받아야 하고, 자신이 해야 할 일 이상을 하는 함수를 만들지 않아야 한다. 함수 이름에 동사 하나만 맵핑(Mapping)이 되면 적당하다. 물론 그 동사에 대한 목적어도 하나이어야 한다. 함수의 호출 순서가 중요하다면, 그 함수를 사용하는 측면에서 익혀야 할 정보가 더 많다는 의미다. 특정 시간에 호출해야 한다는 것은 함수가 시간에 민감한 결과를 돌려준다는 것으로도 해석할 수 있다. 이것도 인터페이스에만 한정된 의존성은 아니다. 시간적인 의존성까지 가진다는 말이되기 때문이다. 내부 데이터 구조에 민감한 함수를 사용해야 한다면, 함수가 제공하는 모든 자료구조를 이해해야 한다. 이런 모든 것을 다 만족시키는 것이 어렵다면, 최대한 만족시킬 수 있는 방법으로 코딩해야 한다. 그래야만 최소한의 유연성을 확보할 수 있기 때문이다. 사람은 전체를 한 번에 보는 "추상화" 능력을 갖추고 있지만, 세밀한 부분을 다 기억하는 것은 힘들어 한다. 그리고, 설령 기억했다고 해서 언제까지나 기억이 유지되는 것도 아니다. 따라서, 적절한 분할(Divide)이 추상화를 위해서는 필수적이다.

자신이 전문가다운 개발자라고 생각한다면, 정말 그런지 자신의 코드를 보길 바란다. 함수들만 의존적이지는 않다는 것을 발견할 것이다. 함수나 자료구조를 정의한 헤더 파일도 연장선에서 봐야 한다. 의존적인 자료구조를 많이 사용할수록 점점 더 해당 인터페이스를 사용하기가 쉽지 않다는 것을 알 수 있을 것이다. 어떤 개발자는 코딩하고 나서 남들이 손도 못 대게 하는 경우도 있다. 그것은 "경험"이 많은 것이 아니라 "아집"일뿐이다. 변경하지 못하게 만들어 다른 사람들을 배제하려는 목적으로 개발한 코드다. "직업의 안정성(Job Security)" 측면에서 그렇게 만들었다고 대답한다면 잘못된 생각이다. 그것으로 인해서 자신의 직업이 오히려 더 위태롭게 될 것이기 때문이다. 이유는 단순하다. 아무도 당신이 코딩을 잘한다고 생각하지 않을 것이고, 뒤에서는 문제가 많은 코드를 양산한다고 비난할 것이다. 대놓고 그렇게 하지 않는 이유는 당신의 직급이 조금 더 높아서이겠지만, 그렇다고 당신을 롤 모델(Role Model)로 삼고 싶어하지는 않을 것이다. 당신은 기술적으로는 남보다 더 많이 알지도 모르지만, 그 기술이라는 한 꺼풀이 벗겨지고 나면 벌거벗은 모습 속에서 당신이 그렇게 감추고자 했던 추한 모습들이 여과없이 보여 질 것이다. 그것을 애써 가리려고 노력할수록 당신은 더 수렁에 빠지게 되며, 당신의 직급이 더 높을수록 당신만 그곳에 갇혀 지내지는 않을 것이다. 고통을 함께하는 동료들이 있다고 즐거워할 것인가? 그렇게 만든 모든 책임은 당신에게 있고, 그것을 개선하지 못하게 만드는 것도 당신일 뿐이다. 당신은 비난받아 마땅한 존재가 되는 것이다.

# [ 하드웨어에 투자하라. ]

사람은 뽑으면서도 사용하는 장비에는 인색하게 투자하는 회사들이 있다. 어리석은 생각이다. 소프트웨어 개발자의 생산성을 높이려면 버려지는 짜투리 시간을 줄여야 한다. 따라서, 단 한 번의 마우스 클릭(Click)이라도 줄여주는 게 오히려 더 낫다. 이유는 단순하다. 소프트웨어 개발에서는 어떤 식으로든 빨리 "피드백(FeedBack)"을 받는 게 최선이기 때문이다. 컴파일을 시켜놓고 40분을 기다려야 한다면, 대부분 개발자는 잠시 자리를 떠나게 된다. 하지만, 컴파일에 5분이 걸리지 않는다면, 책상 앞에서 잠시 다른 일을 하면서 결과가 어떻게 나오는지를 기다릴 것이다. 코딩의 대부분 활동은 남이 만든 코드를 읽는 과정이라는 것은 이미 잘 알고 있을 것이다. 그리고, 남이 만든 코드를 직접 손으로 옮겨적는 경우(Copy & Paste)도 많다. 어쨌든 이 때도 큰 모니터를 사용한다면 훨씬 더 코드들이 눈에 잘 들어온다. 어떤 시스템을 사용할 때 한 번이라도 클릭을 줄일 수 있다면, 더 빨리 시스템을 사용할 수 있다. 이런 모든 것들이 직접적으로 개발자의 생산성에 영향을 준다. 큰 모니터를 사는 것은 사람을 한 명 더 뽑는 것보다 훨씬 적은 비용이 들지만 실제 효과는 크다. 하지만, 회사는 하드웨어 구입 예산은 줄이면서도 비싸고 좋은 사람은 열심히 찾는다. 정말 좋은 사람을 찾기는 쉽지 않지만, 좋은 장비를 사는 것은 상대적으로 너무나 쉬운 일이다. 어디에 투자하는 것이 더 효과적인지는 뻔한 이야기다. 사람을 제대로 활용하기 싶다면 좋은 하드웨어도 충분히 갖출 수 있도록 해야한다.

개발 현장에는 사무용 PC 한 대와 개발용 PC 한 대는 기본적으로 주어져야 한다. 사무용은 성능이 조금 낮아도 되지만 개발용은 가장 최신 사양을 사는 것이 좋다. 너무 비싼 것보다는 SSD나 CPU, RAM 등의 가용자원이 많은 것이 좋다. 대략 200만 원 정도면 충분히 좋은 사양을 구할 수 있을 것이다. 개발자는 개발용 PC에서 더 오랜 시간을 보내기 때문에 빠른 기계일수록 좋다. 때로는 개발팀에서 개발용 서버를 유지하는 경우도 있는데, 이 때는 멀티 코어를 가지고 있는 기계를 사서 컴파일 시간을 최대한 단축시켜야 한다. 개발자들도 컴파일러가 멀티 코어를 충분히 활용할 수 있는 옵션을 선택하도록 한다. Eclipse같은 개발환경은 약간 무겁기에 개발서버에 Quad-Core CPU를 사용할 경우 3명이 한 대 정도 활용할 수 있으면 좋을 것이다. “Xmanager”나 “Exceed”, “NXClient”등을 활용해서 Linux용 개발 서버의 데스크톱 환경을 Windows로 끌어와서 사용할 수도 있지만, Network 환경에서는 속도가 느려질 수 있으므로 가능한 개발자들이 자유롭게 사용할 수 있는 개발용 PC를 할당하는 것이 좋을 것이다. 사용하는 모니터의 해상도는 특히 중요하다. 될 수 있으면 30" 모니터가 좋겠지만 최소한 27" 이상은 되어야 코드를 보기 편하다. 해상도는 1920x1080은 기본이고, 그것보다 더 높은 해상도를 지원할 수 있는 모니터가 좋다. 모니터가 크면 코드를 보기 위한 눈의 이동이 최소화되며, 한 번에 보는 양이 많아져 이리저리 마우스를 이동하면서 볼 필요가 없다. 한가지 문제점은 모니터의 해상도가 높아지면 함수와 같은 단위 코드의 길이가 늘어난다는 점이다. 개발자가 긴 코드를 만드는 것은 결코 좋은 습관이 아니다.

예전에 이런 경우를 본 적이 있다. 개발 장비를 사기 위해서 예산을 미리 정하고, 할당된 예산이 없거나 부족할 때는 각종 절차에 따라 구입 승인을 위한 보고서를 작성했었다. 하지만, 사람의 일이란 항상 예상을 벗어나기 마련이고, 개발 장비에 대한 추가 확보가 어려운 상황에서 개발자는 늘어났었다. 개발 장비를 사기 위해서 올린 요청 때문에 관리자에게 보고해야 했었고, 그로 인해 여러 번 반복적으로 다시 보고해야 하는 상황이 발생했다. 결국 거의 몇 달이 지난 후에 개발 장비를 구할 수 있었고, 개발은 이미 절반이나 시간이 흘러간 상황이었다. 물론 미리 예측을 잘하지 못한 잘못은 있지만, 그렇다고 개발 장비를 구매하는데 필요한 절차와 보고 등으로 낭비한 시간도 적지 않았다는 점은 문제라고 볼 수 있다. 보고에서 몇 번 퇴짜를 받고 나면, "내가 왜 이러고 있지?"하는 회의감까지 들 때가 있다. 나중에는 아예 보고하는 것 자체가 귀찮아지기까지 한다. 아래 사람들에게는 그냥 지금 있는 장비를 최대한 효율적으로 사용하라는 이야기 하지만, 결국은 필요한 것을 제 때에 사지 못하는 것은 같다. 장비를 사는 것은 낭비가 아니라 투자이며, 적절하게 투자하지 못하면 생산성도 떨어지기 마련이다. 새로운 PC를 구매해야 하는 상황은 대략 3년의 주기로 발생하고, 늦어지면 늦어질수록 개발 효율은 떨어진다. 하지만, 회사는 한 번 산 장비는 잘 바꿔주지 않는다. 그냥 그대로 사용할 수밖에 없다. 개발자의 생산성은 생각의 속도를 따라갈 수 있는 기계에 의존한다. 타자를 예로 들면 처음에는 타이핑을 배우기 위해서 자판의 위치를 보면서 코드를 입력하지만, 조금 지나면 자판의 위치를 모르고도 자연스럽게 손가락이 반응한다. 마치 코드가 머리속에 떠오르면 자동으로 손가락이 움직이는 것처럼 보일 것이다. 물론 생각을 그대로 코드로 옮기는 것은 위험하지만 어쨌든 개발자들은 그렇게 일한다. 생각을 따라잡을 수 있을 정도의 스피드를 가지고 있는 장비를 산다면, 즉각적인 피드백을 얻는 것은 사람의 노력 여하에 달려있게 된다.

사람에 대한 투자는 사람을 뽑는 것으로 끝이 아니다. 팀에 맡긴다고 해서 사람이 성장하는 것도 아니다. 사실 팀에 맡기는 것은 팀장이 어떤 사람이냐에 따라 성장이 달라질 수 있고, 또한 팀의 분위기에 크게 영향을 받는다. 대부분 팀은 사람의 성장에는 크게 투자를 안 하는 편이다. 주로 일에서 배우도록 만든다. 하지만, 제대로 일하는 방법을 배우는 것은 결코 쉬운 일이 아니다. 대부분의 개발자는 남을 가르치는 것에 익숙하지 않고 개인주의 성향도 강하기 때문이다. 제대로 배운 개발자 밑에 있는 것은 행복한 일이지만, 그런 인생의 멘토를 얻는다는 것은 회사 생활에서 드문 경우다. 따라서, 이런 것들은 전체 조직 차원에서 나서야 할 부분이다. 업무에서 벗어나 어떻게 일하는 것이 제대로 일하는 것인지 배워야며, 개선을 실현할 수 있도록 체계적으로 지원해 주어야 한다. 실무 관리자들은 대부분 바쁘다는 말로 교육을 중요하게 생각하지 않는다. 하지만, 개발자 교육은 미래를 만들기 위해서 반드시 필요하다. 바쁘다고 미루는 만큼 개발 현장에서의 변화도 느려지게 된다. 대부분의 소프트웨어 개발자들은 생각보다 훨씬 더 경험이 없다. 경험이 있는 사람들도 제대로 된 경험인지 모르는 경우가 많으며, 심지어 자신이 코딩을 대단히 잘 하는 것처럼 생각하는 경우도 있다. 경험의 많고 적음을 떠나서 올바른 개발 경험을 가지는 것은 또 다른 세상이다. 15년을 경험해도 1년 동안 경험한 것보다 못할 수도 있다. 자신이 만드는 시스템에 대해서 도메인 지식은 풍부하게 있을지도 모르지만, 그것을 세상에 구현하는 일은 다른 차원의 일이라는 것을 깨닳아야 한다. 학교나 회사에서 몇 년을 배워도 깨닫지 못하는 것은 경험이 부족해서가 아니라 제대로 배우지 못했기 떄문이다. 따라서, 사람에 대한 교육과 투자는 지속적인 개선을 위해서 선택이 아닌 필수라는 사실을 알아야 할 것이다.

# [ 자동화가 주는 이점 ]

테스트 자동화는 소프트웨어 개발에서 가장 비용 대 효과가 큰 부분이다. 비용이 적게 들면서도 효과는 지속해서 유지되는 부분이다. 물론, 테스트 자체도 리팩토링(Refactoring)이 필요하다는 점은 인정하지만, 테스트가 없는 프로그램을 개발하는 것은 시각장애인이 아무 도구도 없이 길을 걷는 것과 같다. 어떤 시점에 어떤 문제를 만날지를 전혀 예상하지 못한다. 앞으로 나아갈수록 점점 더 큰 비용이 들어갈 것이 분명하다. 따라서, 테스트 자동화는 반드시 소프트웨어 개발에서 해결해야 할 가장 큰 부분이다. 빌드(Build)의 자동화는 자주 통합할 가능성을 준다. IDE를 통한 사용자의 "Click"보다는 도구을 이용해서 특정한 시간에 자동으로 만들어질 수 있도록 해야 한다. 배포 및 회기테스트(Regression Test)도 역시 자동화가 있어야 하는 중요한 부분이다. 자동화는 이처럼 사람이 했을 때 실수가 자주 발생할 수 있는 부분을 대체하는 수단이다. "어렵고 실수가 많을수록 자주 하라!"는 것 말처럼 자주 실행해야 할 부분들은 자동화 해야한다. 물론, 자동화를 위한 비용이 적은 것은 아니다. 특히 개발자가 만드는 자동화된 테스트는 그 비용이 개발 속도를 늦추는 것처럼 보일 수도 있다. 하지만, 한 걸음 더 나아가기 위해서 기초를 탄탄하게 다진다고 생각하는 것이 옳다. 그런 것들을 무시하고 개발된 사례에서 알 수 있듯이, 결국에는 더 큰 비용(개발 지연)이 들어간다는 것을 경험으로 증명할 수 있을 것이다.

자동화를 할 수 있는 대상은 이미 알듯이 “자주 해야 하지만 자주 못 하는 것들”이며 사람의 손을 많이 타서 실패할 가능성이 높은 항목들이다. 통합(Merge)과 같은 것은 사람의 손이 필요하며 자동화도 어렵다. 물론 통합과 관련된 "Three-Way Merge"방법이 있긴 하지만, 충돌(Conflict)이 발생하면 사람이 해결할 수밖에 없다. 통합이 늦어질수록 비용은 커지게 되며, 늦추면 늦출수록 충돌 문제도 심각해진다. 따라서, 개발 첫 날부터 시작해야 하는 것이 통합과 컴파일을 할 수 있는 코드를 만드는 일이다. 물론 실행도 당연히 되어야 한다. 실행된다는 의미는 테스트도 되어야 한다는 것을 포함한다. 자동화는 이처럼 과제의 시작과 동시에 진행하는 것이 가장 비용이 싸다. 과제의 중간에 테스트와 같은 것을 자동화하겠다면 지불해야 하는 비용도 커진다. 과제의 중간에 해야 하는 자동화도 물론 있다. 만약 사용자 시나리오와 같이 완전한 형태의 데모를 목적으로 한다면, 이 때는 필요한 것들이 개발 완료된 이후에 할 수 있다. 이런 것들도 실제로는 시나리오 단위의 테스트 자동화가 개발과 동시에 이루어지는 것이 원칙이다. 원칙이란 지키라고 있는 것이지 지키지 않아도 된다는 생각한다면 아무것도 변하는 것은 없다. 그런 생각으로는 자동화를 해도 얻는 이익이 제한될 것이다. 따라서, 자동화 목표를 정확히 모든 개발자가 다 이해하고 이를 지속해서 추진해야 효과가 클 것이다.

자동화의 가장 큰 이점은 "자신감"을 통한 "신뢰"의 회복이다. 개발 현장에서는 자신이 개발한 코드도 믿지 못하는 상황이 흔하게 발생한다. 자신이 개발한 코드에 문제가 없다고 생각하는 개발자는 드물다. 물론 그런 개발자가 있다고 하더라도 100% 장담은 하지 못한다. 100% 장담은 아니라도 90%라도 장담을 할 수 있다면 어떨까? 이런 경우는 측정할 수 있어야 한다는 것을 가정하고 있다. 측정을 통해서 먼저 얼마나 많은 코드가 테스트되고 있는지를 확인해야 한다. 이것도 충분히 자동화를 할 수 있는 부분이다. 단위 테스트를 실행해서 코드의 "문장(Statement), 분기(Branch), 함수 호출(Function Call)"수준에서 "범위(Coverage)"를 측정하는 것을 사용하면 된다. 물론100% 테스트 범위를 가진다고해서 버그가 전혀 없다고는 장담하지 못한다. 왜냐하면, 다양한 상황(Context)이라는 것이 존재하기 때문이다. 하지만, 이를 통해서 얻을 수 있는 것은 상당 부분의 코드가 제대로 동작하며, 적어도 주어진 입력에 대해서는 버그가 없다는 것을 알 수 있다는 것이다. 이것만을 확인할 수 있다고 하더라도 상당수의 개발자는 자신감을 충분히 회복할 수 있다. 그리고, 더 중요한 것은 과제에 대해서 투명성을 제공할 수 있기에 추정(Estimation)이 좋아진다는 점이다. 즉, "Done Done"이라는 "일이 제대로 완료가 되어야 정말로 된 것이다"라는 것을 결정할 수 있다는 점이다. 종료된 일은 테스트가 완료된 일을 말한다. 또한 종료되었다는 말은 해당 기능을 배포할 준비가 완료되었다는 의미다. 따라서, 정말 일을 다 했다는 의미("Done Done")에 대해 선언을 할 수 있게 된다. 개발자가 이야기하는 ”90% 완료”를 가지고는 아무도 얼마나 더 많은 일을 해야할지 전혀 예상하지 못한다. 하지만, 테스트까지도 되고 통합되어 배포할 준비가 완료된 일에 대해서는 충분한 확신을 줄 수 있다.

자동화는 원할 경우 그 일을 다시 할 수 있다는 말이다. 누군가 불시에 결과물을 원하더라도 즉시 만들어낼 수 있는 수준이 되어야 한다. 이를 어렵다고 이야기한다면, 아직도 실수를 많이 할 수 있는 환경에서 일하고 있다는 뜻이다. 그리고, 배포를 위해서 필요한 모든 절차를 제대로 수행할 수 없다는 뜻이기도 하다. 사람마다 달라지고 환경마다 차이가 발생한다면 이를 지속해서 자동화할 필요가 있다. 자동화의 이점은 과제의 후반으로 갈수록 빛을 발한다. 물론 이런 것들이 필요치 않다고 이야기하는 경우도 많이 본다. 예를 들어, 사람이 빌드를 하고 이를 배포 담당자에게 바이너리(Binary)로 인계하는 경우도 종종 볼 수 있다. 하지만, 조금 생각만 생각해도 이런 과정에서 발생하는 다양한 문제(사람의 실수를 포함해서)를 다시 경험하고 싶어하지는 않을 것이다. 잘못된 소프트웨어를 제공해서 문제가 발생해 재배포를 했던 경험들은 다 있을 것이다. 그럴 때마다 상대가 느끼는 감정은 "불신"으로 이어지게 된다. 자동화된 배포는 이런 불신을 해소하는 방법으로도 중요하다. 과제의 후반부는 잦은 배포와 테스트를 위한 수정, 디버깅으로 이어지는 불편한 밤들의 연속이다. 이때는 아무리 어려움을 호소해도 도와줄 방법은 제한적이다. 결국은 스스로가 판 무덤에 자기 스스로 파묻힐 뿐이다. 자신감은 점차 사라지고 과제를 어떤 식으로라도 마무리짓는 데 치중하게 된다. 설정된 완료 시간이 다가올수록 공포감은 극대화되고, 내외부의 압력도 덩달아서 높아만 갈 것이다. 제대로 된 코드를 만든다는 것이 이 상황이면 거의 불가능하다는 것은 안 봐도 뻔하다. 시간이 부족하다는 말도 거짓이 아님을 잘 안다. 하지만, 처음부터 제대로 된 개발을 하지 않아서 생기는 이런 문제들은 이미 제어할 수 있는 수준을 넘어섰다. 모든 약이 그렇듯 처방이 유효한 시간은 미리 정해져 있다. 지금 하지 못하면 나중에는 더 하기 힘들어진다.

# [ 아름다운 코드(Beautiful Code) ]

코딩이란 무엇일까? 단순히 컴퓨터가 실행할 수 있는 명령어들의 나열이라고만 생각하기에는 너무나 많은 오해가 생긴다. 가장 큰 오해는 "어떤 식으로 만들어도 실행만 되면 상관없다는 것"이다. 하지만, 실행이 되고 성능을 만족한다면 아무런 문제가 없을까? 실행이 되는 프로그램을 짜는 것은 어렵지 않은 일이다. 요구되는 성능을 만족시키는 코드를 짜는 것은 조금 더 어려운 일이다. 물론 턱없이 높은 성능을 만족시킬 수는 없다. 논리적으로 구현 가능한 경우에 한정해서 구현할 수 있을 것이다. 성능은 소프트웨어의 품질에 관련된 것으로 구조적인 측면도 생각해서 만들어야 하기 때문이다. 여기에서 간과하고 있는 것은 "성능"이 다른 모든 소프트웨어가 갖추어야 할 품질들을 무시할 수 있을 만큼 중요한 것은 아니라는 점이다. 사용자는 같은 값이라면 성능이 더 좋은 소프트웨어를 사려고 할 것이다. 하지만, 외적으로 드러나는 품질은 내적으로 만족시켜야 할 품질의 외적인 표현일 뿐이다. 내부적으로 만족시키지 못하는 품질을 가지고, 외적인 품질을 만족시키는 것은 어럽다는 뜻이다. 기본적으로 내적인 품질을 만족시켜야 외적인 품질도 만족시킬 가능성이 높은 것이다. 따라서, 성능과 같은 외적인 품질에 너무 많은 중심를 두기보다는, 먼저 내적인 품질을 높이는데 주력해야 한다. 사용자가 가지는 소프트웨어에 대한 가치도 한 번의 만족보다 지속적인 만족을 요구하기 때문이다.

내적인 품질은 주로 "코드의 이해 및 변경"과 관련된 항목들이다. "읽기 쉽고 변경하기 쉬운가?", "변경했을 때 다른 코드에 주는 영향이 최소화되어 있는가?", "테스트하기 쉬운가?" 등등 주로 사람(소프트웨어 개발자)의 이해를 바탕으로 한다. 즉, 회사 내부에서 소프트웨어를 개발하고 있는 사람들에게 중요한 것들을 이야기한다. "아름다운 코드"란 "눈이 즐겁고 이해가 쉬운 코드"다. 개발자라면 다들 남이 작성한 코드를 본 경험이 있을 것이다. 그리고, "도대체 왜 이렇게 짠 거야?"라면서 누군지도 모를 다른 개발자들을 욕한 경험은 있을 것이다. 그리고, 그 대상이 자신이었던 적도 있을 것이다. 이런 일이 생기는 이유는 "코딩은 개인적인 일이다."라는 잘못된 생각에서 출발하기 때문이다. 소프트웨어 개발자들이 "개인적인 사람"들이라는 선입견으로 보지 않아야 한다. 소프트웨어 개발자들은 "사회적으로 충분히 성숙한 사람"들이어야 한다. 대화에 익숙하고 팀원들 간의 상호작용에 적극적인 사람들이어야 한다. 그런 사람들만이 "이타적인" 코드를 짜는데 필요한 기본을 갖출 수 있다. 타인을 생각하지 않는 코딩은 결국 자신만을 위한 코딩이고, 그렇게 만들어진 코드들은 지속적으로 내부 사용자들을 만족시키기 어렵다. 다른 사람의 코드를 보았을 때 "아름답다"라고 생각한 적이 있었던가? 대부분의 경우는 아마도 "아니오"라는 대답일 것이다. 우린 남의 코드를 읽는 데도 익숙하지 않지만, 쉽게 읽히는 코드를 짜는 데도 여전히 익숙하지 않기 때문이다. 이것이 문제의 출발점이다. 코딩은 “아름다움을 추구하는 끝없이 익혀야 할 기술"이기 때문이다.

추상화 수준이 높은 일을 하는 사람들은 "손과 발이 움직이는 일"에 대해서 수준이 낮은 일로 보는 경향이 있다. 멋진 다이어그램을 그린다고 멋진 코드가 나오는 것은 아니다. 좋은 알고리즘이라고 해서 반드시 항상 좋은 해결책이 되는 것은 아니다. 어떤 경우에는 최적보다는 차선책이 더 효과적인 해결책이 될 수도 있다. 따라서, 추상적인 일은 실무와 연결되어 있어야 가치를 가진다. 아키텍트라고 해서 실무 개발자들이 만드는 코드에서 동떨어져 생각해서는 안 된다. 그들이 만든 설계도가 어떻게 사람이 살기 좋은 안락한 집으로 바뀌는지 직접 경험해야 하기 때문이다. 경험이 많은 아키텍트일수록 자신의 경험이 잘 녹아든 구현하기 쉬운 구조를 설계할 것이다. 기존의 아키텍처 패턴에서 배운 것을 상황에 맞게 적용해야 한다는 사실도 잘 알고 있을 것이다. 경험이나 사례에 기반을 두지 않은 설계는 100% 완전한 설계로 생각하는 것은 위험하다. 오히려, 경험이 축적될 때까지 개선을 허용하는 구조를 가져야 한다. 따라서, "아름다운 구조"란 변화를 가져올 수 있는(변경하기 쉬운) 내적인 대비가 충분한 구조를 말한다. 한번 결정된 구조를 변경하지 못하도록 속박하는 것은 올바른 판단이 아니다. 구조란 상황(요구사항)이 바뀌면 변경을 수용할 수 있는 충분한 "공간"이 있어야 한다. 변경 비용이 전체 시스템에 막대한 영향을 주는 경우도 발생할 수 있기 때문이다. 이런 관점에서 볼 때 건물을 짓는 것과 소프트웨어를 만드는 일은 서로 다르다는 것을 알 수 있다. 소프트웨어는 구조적인 변경을 비교적 쉽게 할 수 있는 반면, 건물은 절대 고쳐서는 안되는(수정하지 못하는) 부분이 존재하기 때문이다. 물론 변경의 결과까지 충분히 고려한 것은 결코 아니다.

코드를 만드는 일은 가볍게 생각할 것이 아니다. 만약 학교에서 코딩을 가르친다면 가장 먼저 해야 할 일은 좋은 코드를 구별하는 능력을 키우는 것이어야 한다. 많은 코드를 읽어야 할 것이고, 프로그램의 구조에 대한 생각을 키울 수 있어야 한다. "구조적인 프로그래밍"의 기초가 다져진 후에 추상적인 것들이나 새로운 개념을 익혀야 할 것이다. Java나 C++과 같은 객체지향 언어를 아는 것도 중요하지만, 전체적으로 어떤 구조를 만들어야 좋은지부터 이해하는 것이 선행되어야 한다. 단순한 기술이라고 아무렇지도 않게 이야기하는 "코딩"은 절대 쉬운 일이 아니라는 점을 알아야 한다. 숙제를 검사할 때는 결과값이 중요하지만, 그것이 어떻게 만들었는지도 함께 보여주는 그림을 그려야 한다. 사회성을 키우기 위해서는 팀워크로 일할 수 있는 과제를 주어야 할 것이며, 특정인의 능력에 따라 팀의 성과가 달라지는 것을 보고싶지 않다면 각자의 역할별로 구현해야 할 부분을 할당해야 한다. 가능하면 이런 경험들이 쉽게 잊히지 않도록 각종 측정 도구를 동원해서 과제가 어떻게 진행되는지도 기록해야 한다. 목표는 "고객이 원하는 품질을 충분히 만족시키면서 개발자들이 원하는 아름다운 코드를 만드는 것"이다. 따라서, "코딩 이론"과 "실무"가 함께 병행되어야만 양측의 만족이 극대화될 수 있다. 학교 학점에만 치중한다면 회사에서 발생하는 반복적인 문제의 고리를 끊어낼 수 없다. 언제나 "시작이 반"이다. "아름다운 코드"는 못만들더라도, "아름다운 코드를 작성하려는 노력"은 지금 바로 시작해야 할 것이다.

# [ 버그와 생산성 ]

버그는 생산성에 있어서 가장 치명적인 영향을 미친다. 개발 대부분 시간은 사실상 디버깅과 테스트를 하는 시간이라는 것을 소프트웨어 개발자라면 다 알 것이다. 그리고, 대체로 과제를 맡은 PL(Project Leader)은 디버깅을 위한 시간을 과제 후반에 길게 잡아두길 원한다. 하다못해 최소한의 일정 버퍼로 1달에서 3달까지의 기간을 계획에 추가적으로 두는 일반적이다. 물론 이렇게 과제 담당자가 하는 이야기를 관리자들은 이미 잘 알고 있고, 일정계획에 버퍼를 넣었을 것이라고 추측을 한다. 과제를 협의하는 시간이 오면 그런 추측은 "일정 단축 요구"라는 것으로 나타나게 된다. 이유는 과제의 PL과 마찬가지다. 이렇게 일정을 단축하도록 요청을 해야 원래의 일정을 만족시킬 수 있을 것이라는 생각을 하기 때문이다. 하지만, 이렇게 정한 일정은 위로 올라갈수록 더 단축되기 마련이다. 비슷한 생각들을 가지고 일정은 계속 줄어들기만 한다. 나중에 개발팀이 받아드는 일정은 버퍼는 전혀 없고, 기능구현도 하기 벅찰 만큼의 시간이 된다. 개발은 말로 하는 것이 아니다. "말씀"으로 무엇인가를 만들 수 있는 능력은 "신"만이 가능하다. 불확실한 추정과 명령으로 만들어진 일정은 절대 만족시킬 수 없다. 오히려, 그것을 맞추려고 노력할 때 과제가 망가질 가능성이 더 커진다. 당겨진 일정은 "과정의 생략"을 낳기 때문이다. 그 중에서도 가장 큰 생략은 "이른 테스트"와 "구조화의 노력"이며, 그것들로 인해서 발생하는 비용은 과제의 시간이 흘러갈수록 더 커지게 된다. 비용은 절대 선형적(Linear)으로 증가하지 않는다. 비용은 사람이 들어간 만큼, 단계가 지나가는 만큼 지수적(Exponential)으로 증가할 것이다. 지금 당장은 어떻게 해결할 수 있을지라도 시간이 조금씩 흘러가면 어떻게 해볼 수 있는 수준을 벗어나게 된다.

문제가 되는 것은 버그를 미리 제거하는 활동을 생산성에 나쁜 것으로 보는 태도다. 버그를 많이 발생시키는 개발자가 일을 못 한다고 인식한다는 점이다. 버그는 어쩔 수 없이 발생할 수밖에 없지만, 버그로 만드는 과정을 지켜봐야 한다. 사전에 버그를 줄이기 위한 활동에 대해서 지원하지 않으면서 버그를 만들었다고 비난하는 것은 옳지 않다. 버그만 줄일 수 있다면 생산적인 일을 할 수 있는 시간(여유)을 벌수 된다. 즉, 과제 후반에 집중된 테스트 시간을 줄이는 것이 유일한 일정 단축 방법이다. 코딩하는 중간에 코드을 확인하는 과정을 넣어서, 개발자가 제대로 했는지를 검사할 수 있어야 한다. 간단한 문제들이 쌓이게 되면 복잡한 문제로 변질되기에, 작은 코드의 변화를 즉시 확인하는 것이 가장 효과적인 버그를 줄이는 방법이다. 테스트 때문에 구현이 늦어진다는 것은 말이 안 되는 이야기다. 테스트는 구현시간을 줄이는 효과를 가진다. 코드의 변화가 작은 상황에서 테스트하는 것은 즉각적인 버그를 찾는 데 도움이 되기에, 나중에 버그를 찾는데 필요한 비용을 줄여준다. 따라서, 테스트는 더 자주 빨리 하는 것이 결과적으로 생산성이 더 높은 활동이다. "잘 안되고 하기 힘든 것은 더 자주하라"는 말이 있듯이, 테스트를 잘하지 못한다면 더 자주 테스트할 수 있는 방법을 찾아야 한다. 결론은 역시 테스트 자동화가 해답이다. 자동화되지 않은 테스트는 비용(시간)이 많이 들지만, 자동화된 테스트는 디버그 활동보다 비용이 적게 든다. 따라서, 할 수 있는 한 최대한 테스트를 자동화하는 것이 생산성 향상을 위해서는 필수적인 일이다. 생산성은 더 높은 완성도를 가지는 코드를 얼마나 빨리 만드는가에 대한 것이기에 자동화된 테스트가 생산성을 높일 수 있는 유일한 방법이 되는 것이다.

유일한 생산성 향상 방법을 구현하는 데는 비용이 든다. 하지만, 사람들은 그 비용이 시간과 개발자의 역량이라는 점을 모른다. 그것들을 보장하지 않으면서 해답을 찾으려고 하는 것이 풀어야 할 문제다. 코딩하는 것은 일하고 있는 것으로 보지만, 코드를 검증하고 있는 것은 일하고 있지 않다고 본다. 그리고, 심할 때는 테스트와 코딩을 분리해서 생각한다. 설계 없이 코딩하는 것이 문제라는 정도는 이제는 이해하는 수준에 도달했다고 보지만, 테스트를 개발자가 자신의 코드에 대해서 직접 해야 한다는 것은 아직도 동의를 얻지 못한 부분이다. 개발자들은 코드를 만드는 것에는 익숙하지만(어쩌면 익숙하다는 표현이 어울리지 않을지도 모르지만), 테스트하는 방법은 잘 배우지 않는다(모른다). 이런 현상은 코딩을 못 해서가 아니라 코딩을 많이 해서 발생한다. 변화란 아무리 작아도 시스템에 영향을 준다는 사실을 명심해야 한다. 따라서, 아무리 작은 코드의 변화라도 원칙적으로는 테스트를 거쳐야 한다. 변경하기 전에 코드의 행위를 알 수 있다면, 변경한 후에 어떤 변화가 있었는지 알 수 있다. 코드의 행위를 알 수 있는 방법을 테스트라고 한다면, 먼저 테스트를 만들어 변경 후에 어떻게 동작하는지 확인할 수 있다. 즉, 코드를 만들기 전에 그 코드를 위한 테스트를 만들라는 뜻이된다. 테스트를 통과할 수 있는 작은 변화된 코드를 만드는 것이다. 이를 반복해나가면 테스트와 코드를 둘 다 완성하는 것이 최종적인 "구현의 완료"가 된다. 구현은 결국 검증되어야 의미를 가지기 때문이다. 그렇다면 테스트를 위한 비용(혹은, 시간의 가치를)은 어떻게 확보할 것인가? 현실의 문제는 항상 더 많은 일을 해야 한다는 강박관념에서 나온다. "많은 일"을 하지 않으면 목표에 도달하지 못한다는 생각이 속도를 높이라고 계속 요구한다. "동일한 시간에 더 많이 생산하는 것이 비용을 줄인다"는 오해가 지식노동에 영향을 주고 있는 것이다. 지식노동의 결과물에 대한 생산성을 이야기해야 한다면, 만들어진 결과물의 "품질"을 중심을 두어야 한다. 지식노동자들이 만드는 것은 오직 하나이며, 결과물이 무한 복제되는 특성을 가지기 때문이다. 우리가 하는 일은 많은 결과물을 빨리 만들어내는 것이 아니라, 단지 "하나의 품질이 높은 결과물"을 만들어야 하기 때문이다.

지식노동의 결과물은 복제의 비용이 상대적으로 극히 적다. 생산성은 만들어낸 결과물의 숫자가 아니라, 그 결과물이 주는 가치이다. 개수가 중요한 요소가 아니라 만들어진 결과물의 내외적인 품질이 생산성의 척도가 되어야 한다. 버그는 품질에 영향을 주는 것으로 생산성을 낮추는 요소이기에, 이를 사전에 제거하는 길이 생산성을 극대화하는 방법이다. 생산성은 버그의 개수에 반비례한다고 생각할 수도 있으나, 버그는 발견되는 시점에 따라 해결하기 위한 비용이 달라진다. 가장 큰 비용이 발생하는 부분은 시장에서 사용자에 의해 버그를 발견하는 것이다. 구현 버그만을 대상으로 한다면 개발자가 발견하는 버그의 수는 많을지라도 비용은 오히려 더 적게 들어갈 수 있다. 따라서, 최대한 코딩이 일어나는 시점에 버그를 찾는 방법을 마련해야 한다. 더 많은 버그를 개발자들이 찾는 것이 최고의 선택이 될 것이다. 임베디드 시스템에서는 하드웨어가 없으면 테스트도 할 수 없다고 생각할지 모르지만, 하드웨어에 의존하는 코드를 분리할 수 있다면 테스트 자동화도 충분히 가능하다. 사용자 인터페이스(User Interface: UI)를 테스트하기 어렵다고 할지도 모르지만, 대부분의 경우는 비즈니스 로직(Business Logic)과 그래픽을 처리하는 부분(윈도우의 컨트롤등)이 분리되지 못해서 발생하는 문제일 뿐이다. 설계와 구현에서 충분히 테스트를 고려해서 만들어진 코드라면 불가능한 일이 아니다. 기존의 코드 때문에 테스트할 수 없다고 생각한다면, 기존의 코드는 따로 떼놓고 생각하도록 해야한다. 최소한 새로 작성하는 코드라도 테스트를 할 방법을 만들어야 할 것이다. 분리가 힘든 기존의 코드라면 그냥 묶어서 전체를 테스트하는 것도 고려할 수 있다. 문제는 테스트하지 않는 것이지 테스트를 못 하는 것이 아니다. 할 수 있는 한 최대한 테스트를 자동화하는 것이 조금이라도 생산성을 더 높이는 방법이다.

# [ 환경과 코드의 분리 ]

코드는 그것을 동작시키는 환경에서 분리되는 것이 유리하다. 임베디드 제품에 들어가야 코드를 만들 때도 환경과 분리해서 구현하는 것이 좋다. 중요한 것은 그런 코드를 만드는 것은 이식성을 높일 수 있기에, 시뮬레이터의 구현을 통해서 제품에 설치하지 않은 상태에서도 테스트를 실행할 수 있다는 점이다. 이를 위해서 들인 비용은 나중에 충분히 다시 회수할 가능성이 높다. UI(User Interface)의 경우에는 동작하는 환경과 밀접한 관련성을 가지는 경우가 대부분이다. 즉, Windows UI와 Mac OS에서 사용하는 UI에 대한 구현은 달라질 수 있다. 따라서, 구현 시에도 가능성을 열어놓기 위해서 UI와 비즈니스 로직(Business Logic)을 분리해야 한다. OS와의 의존성이 있는 경우도 이를 분리하는 방법을 사용할 수 있으나, 표준 입출력 관련 라이브러리를 사용하는것으로 충분할 수도 있다. 대부분의 운영체제는 나름의 호환성을 유지하기 위해서 자주 사용되는 것들은 거의 동일하게 구현하고 있기 때문이다. 코드를 환경에서 분리하면 최적의 성능을 발휘하는 것은 어려울지도 모르지만, 새로운 환경에 재빨리 적용할 수 있는 제품을 만들수는 있다. 그리고, 하드웨어 비용은 점차 더 저렴해지고 동작 속도는 빨라지고 있기에, 성능 최적화를 위해서 하나의 하드웨어(운영체제)에서만 동작하는 코드를 만들 이유도 점차 줄어들고 있다.

분리가 어려울 때도 최소한 사용되는 자료의 컨텐츠(Contents)라도 일관되게 다양한 시스템에서 사용할 수 있도록 만드는 것이 좋다. 요즘은 서비스를 제공하는 서버는 동일하게 두고, 이에 접근하는 클라이언트를 다양한 플랫폼으로 이식하는 방법을 취하고 있다. 이는 프로토콜과 컨텐츠를 통한 일종의 통합이라고 볼 수 있으며, 클라이언트 프로그램의 기능을 작게 유지해서 새로운 서비스 추가에 효과적으로 대응할 수 있도록 만든다. 이를 좀 더 확대해서 해석해 본다면, "플랫폼을 이용한 생태계(Ecosystem)의 활성화"에서 “컨텐츠를 통한 호환성 확보"로 시장이 이동하고 있다고 생각할 수도 있다. 어쨌든 서비스를 만드는 측면에서 본다면 다양한 플랫폼에 대한 지원은 번거로운 일일 수밖에 없기 때문이다. 따라서, 점차 어떻게 클라이언트 장비를 만들 것인가에 대한 주도권 싸움에서 "어떤 서비스를 제공할 수 있도록 서버를 구축할 것인가?"로 옮겨가고 있다. 분리되는 형태는 제어(Control)의 분리도 있을 수 있고, 제어를 타고 다니며 처리되는 컨텐츠의 형태도 있을 수 있다. 따라서, 이런 것들을 고려한 코딩이 필요하다. 우리가 만드는 소프트웨어는 언제 어떻게 사용될 것인가를 알 수 없는 상황이 올 수도 있기에, 확장된 환경도 충분히 고려해서 설계되어야 할 것이다.

테스트하는 경우라면 테스트 스크립트를 작성할 때 테스트 입력과 테스트 스크립트를 분리할 수 있다. 테스트 스크립트 내에 테스트 데이터를 넣게 되면, 테스트를 변경할 경우에 스크립트도 변경되어야 한다. 테스트 데이터도 다양한 종류가 있을 수 있기에 입력을 읽어서 변화시켜주는 부분도 필요하다. 변화된 중간 결과를 저장한다면 다른 테스트를 위해서 재활용할 수도 있다. 테스트도 소프트웨어의 구조를 설계할 때 중요한 입력으로 사용된다. 즉, 테스트를 쉽게 하기 위한 구조를 미리 정의해서 이를 코드에 반영하는 것이다. 사실 테스트를 쉽게 하기 위한 구조를 반영하지 않은 코드들이 더 많다. 이런 코드를 가지고 테스트를 하는 것은 새로 코딩을 하는 것만큼이나 어렵다. 테스트를 쉽게 할 수있는 구조는 내부적으로 만들어지는(생성하는) 물리적인 요소를 줄이고, 외부에서 제공해주는 방식으로 정의하는 것이다. 코드 자체는 의존성이 적을수록 좋다. 따라서, 외부 환경과 독립적으로 실행되는 것이 가장 좋다. 물론 이런 것들이 불가능한 경우도 있다. 이 때는 환경을 분리하기 위한 계층구조(Layering)를 정의해서 그것에 의존하도록 코딩하는 것이다. UI의 경우에는 주로 이벤트(Event)를 이용해서 메시지를 전달하기에 외부에서 메시지를 생성해서 비동기 이벤트를 사용할 수 있도록 소프트웨어 구조를 만든 것도 좋은 방법이 될 수 있다. 환경과 구현을 분리하는 일은 잘 정의된 인터페이스를 이용할 수 있게 하면 그만큼 더 쉬워진다.

하드웨어에 대한 의존성을 가지고 있는 코드를 짠다면, 주로 하드웨어의 레지스터(Register)에 접근하는 부분이 가장 높은 의존도를 가지게 될 것이다. 만약 하드웨어의 레지스터를 인위적으로 위치를 변경해 줄 수 있도록 특정 주소로 고정(Fix)하지 않는다면, 이것도 분리해서 구현 가능하다. 따라서, 의존성은 가능한 모든 코드에서 줄이는 것은 충분히 가능한 일이다. 개별 모듈의 측면에서 보면 다른 모듈 역시 실행 환경이라고 생각할 수 있다. 따라서, 모듈 간의 의존성도 줄이는 것이 중요한 목표가 될 수 있다. 모듈 간의 의존성은 모듈의 내부 구현으로부터의 독립을 의미하며, 주어진 인터페이스만 사용해서 구현해야 한다는 것이다. 될 수 있으면 모듈의 내부 구현이 어떻게 되어있는지 모르고도 잘 사용할 수 있는 구조를 만들어야 한다. 코딩에서 가장 잘 안 되는 부분이 사실 이런 부분이다. 대부분의 경우 개발자는 해당 자료구조를 직접 사용해야 더 높은 성능이 난다고 믿기 때문이다. 하지만, 성능도 시스템을 설계할 때 고려해야 할 한 가지 요소일 뿐이다. 다른 중요한 품질 특성을 보여주는 요소들도 많다. 성능만이 유일한 선택 조건이 아니라는 뜻이다. 확장성, 변경성, 유지 보수성, 사용성, 보안, 가독성 등등 다양한 것들이 시스템을 구현하는 사람이 고려해야 할 부분이다. 이들 요소들 간의 트레이오프(Trade-Off)를 고려 햐지 않고 무조건 한 가지 요소만 위주로 구현한다면, 그로 인한 다른 요소들에 들어가는 추가 비용은 감수해야 할 부분이다. 얼마나 큰 비용이 들어갈지는 지금 당장 예상이 안 될 수도 있지만, 적지 않은 비용이 소프트웨어를 배포한 이후에 발생할 수도 있다는 사실은 잊지 않아야 한다.

**맺음말**

지금까지 많은 이야기를 적었지만 삶의 일부분에 대한 지극히 개인적인 관점에서 본 경험들일 뿐이다. 아마도 더 크고 위대한 진실은 아직도 저 멀리서 우리가 찾아주기만을 간절히 기원하고 있을지도 모른다. 무엇이 옳은지 무엇이 그른지를 명확하게 구분할 수 있다면 실수란 존재하지 않을 것이다. 그렇기에 사람은 항상 실수를 한다. 한가지 아쉬운 것은 반복된 실수를 너무나도 당연하게 받아들인다는 점이다. 당연하게 받아들이는 실수는 단지 운이 안좋았거나 개인의 탓으로 밖에 돌리지 못한다. 실수가 개인만의 문제라면 더 뛰어난 사람으로 바꾸면 되겠지만, 모든 실수는 그것을 방지하지 못한 시스템의 오류일 뿐이다. 그리고, 그러한 실수를 최대한 줄이기 위해서 시스템을 개선하는 것만이 반복적인 실수에 대한 유일한 해결책이다.

실패하지 않기 위해서 성공하는 방법들은 이미 충분히 들었을 것이다. 하지만, 그렇다고 모두 성공하지는 못한다. 오히려 왜 실패했는지를 정확히 깨닫게 된다면 조금이라도 성공에 더 가깝게 다가갈 수 있을 것이다. 성공은 너무나 뻔한 것들을 제대로 하는 것이지만, 실패는 구체적인 방법을 알려주기 때문이다. 문제가 있다고 이야기하기는 쉽지만 고치는 것이 어려운 것도 같은 이유 때문이다. 누구나 좋은 말은 할 수 있지만 실제로 손을 내밀어 도와주지는 않는다. 성공하는 방법은 작은 실패를 빨리 경험해서 가능한 경우의 수를 줄이는 것이다. 실패하지 않는다고 성공한다는 보장은 없지만, 그렇다고 실패없는 성공이 있는 것도 아니다. 문제는 실패에 대한 두려움이다. 우리들의 마음이 두려움으로 가득차 실패에 대해서 적극적으로 회피하려는 본성에 있다.

우리가 지금 가고 있는 길을 남들도 이미 지나갔을 갔을 것이다. 세상에 새로운 것은 없으며 길을 지나가는 사람만이 새로울 뿐이다. 지나치는 풍경은 변함이 없지만 그것을 보는 눈은 각각 다르다. 무엇을 보는가는 오로지 개인의 노력에 의존하며, 즐거운 길이 될지, 아니면 고단한 길이 될지는 개인의 결정에 따른다. 하지만, 그래도 위안이 되는 것이 있다면 다른 사람의 경험을 엿들을 기회가 있다는 것이다. 세상은 외롭게 혼자 살아가는 곳이 아니며, 비슷한 문제를 고민하는 사람들도 의외로 많다. 적어도 무엇이 잘못되었는지 이야기할 수 있는 공간은 있으며, 좋은 대답을 들을지는 알 수 없지만 관심을 가져주는 사람들은 충분히 찾을 수 있다. 두려움으로 홀론 떨고 있다면, 이제는 그런 것들을 하나씩 수면위로 끌어올려서 이야기해야 한다. 더 늦어진다면 또 다른 누군가가 같은 실수를 반복하고 있을지도 모르기 때문이다. 이제 그 첫번째 이야기를 이렇게 마친다.

**- 감사합니다. -**