내 코드가
그렇게
이상한가요?

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

YOI CODE / WARUI CODE DE MANABU SEKKEI NYUMON -
HOSHU SHIYASUI SEICHO SHITSUZUKERU CODE NO KAKIKATA
by Daiya Senba

Copyright ⓒ 2022 Daiya Senba
All rights reserved.
Original Japanese edition published by Gijutsu-Hyoron Co., Ltd., Tokyo
This Korean language edition published by arrangement with Gijutsu-Hyoron Co., Ltd., Tokyo
in care of Tuttle-Mori Agency, Inc., Tokyo, through Botong Agency, Seoul.

이 책의 한국어판 저작권은 Botong Agency를 통한 저작권자와의 독점 계약으로 인사이트가 소유합니다.
저작권법에 의하여 한국 내에서 보호를 받는 저작물이므로 무단전재와 무단복제를 금합니다.

내 코드가 그렇게 이상한가요?
좋은 코드/나쁜 코드로 배우는 설계 입문

전자책 1쇄 발행 2023년 6월 26일 **지은이** 센바 다이야 **옮긴이** 윤인성 **펴낸이** 한기성 **펴낸곳** (주)도서출판인사이트 **편집** 나수지
등록번호 제2002-000049호 **등록일자** 2002년 2월 19일 **주소** 서울특별시 마포구 연남로5길 19-5 **전화** 02-322-5143 **팩스** 02-3143-5579 **블로그** https://blog.insightbook.co.kr **이메일** insight@insightbook.co.kr **ISBN** 978-89-6626-413-1

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

프로그래밍 인사이트

내 코드가 그렇게 이상한가요?

좋은 코드/나쁜 코드로 배우는 설계 입문

센바 다이야 지음 | 윤인성 옮김

인사이트

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

차례

들어가며 xiv
감사의 글 xviii

1장 잘못된 구조의 문제 깨닫기 1

1.1 의미를 알 수 없는 이름 2
1.2 이해하기 어렵게 만드는 조건 분기 중첩 4
1.3 수많은 악마를 만들어 내는 데이터 클래스 6
1.3.1 사양을 변경할 때 송곳니를 드러내는 악마 7
1.3.2 코드 중복 9
1.3.3 수정 누락 9
1.3.4 가독성 저하 9
1.3.5 초기화되지 않은 상태(쓰레기 객체) 10
1.3.6 잘못된 값 할당 10
1.4 악마 퇴치의 기본 11

2장 설계 첫걸음 13

2.1 의도를 분명히 전달할 수 있는 이름 설계하기 14
2.2 목적별로 변수를 따로 만들어 사용하기 15
2.3 단순 나열이 아니라, 의미 있는 것을 모아 메서드로 만들기 16
2.4 관련된 데이터와 로직을 클래스로 모으기 18

3장 클래스 설계: 모든 것과 연결되는 설계 기반 21

3.1 클래스 단위로 잘 동작하도록 설계하기 23
3.1.1 클래스의 구성 요소 23
3.1.2 모든 클래스가 갖추어야 하는 자기 방어 임무 25

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

3.2 성숙한 클래스로 성장시키는 설계 기법 **26**
3.2.1 생성자로 확실하게 정상적인 값 설정하기 26
3.2.2 계산 로직도 데이터를 가진 쪽에 구현하기 28
3.2.3 불변 변수로 만들어서 예상하지 못한 동작 막기 29
3.2.4 변경하고 싶다면 새로운 인스턴스 만들기 30
3.2.5 메서드 매개변수와 지역 변수도 불변으로 만들기 30
3.2.6 엉뚱한 값을 전달하지 않도록 하기 32
3.2.7 의미 없는 메서드 추가하지 않기 33

3.3 악마 퇴치 효과 검토하기 **34**

3.4 프로그램 구조의 문제 해결에 도움을 주는 디자인 패턴 **37**
3.4.1 완전 생성자 37
3.4.2 값 객체 38

4장 불변 활용하기: 안정적으로 동작하게 만들기 41

4.1 재할당 **42**
4.1.1 불변 변수로 만들어서 재할당 막기 43
4.1.2 매개변수도 불변으로 만들기 44

4.2 가변으로 인해 발생하는 의도하지 않은 영향 **44**
4.2.1 사례 1: 가변 인스턴스 재사용하기 45
4.2.2 사례 2: 함수로 가변 인스턴스 조작하기 47
4.2.3 부수 효과의 단점 49
4.2.4 함수의 영향 범위 한정하기 50
4.2.5 불변으로 만들어서 예기치 못한 동작 막기 50

4.3 불변과 가변은 어떻게 다루어야 할까 **54**
4.3.1 기본적으로 불변으로 54
4.3.2 가변으로 설계해야 하는 경우 54
4.3.3 상태를 변경하는 메서드 설계하기 55
4.3.4 코드 외부와 데이터 교환은 국소화하기 57

5장 응집도: 흩어져 있는 것들 59

5.1 static 메서드 오용 **60**
5.1.1 static 메서드는 인스턴스 변수를 사용할 수 없음 61
5.1.2 인스턴스 변수를 사용하는 구조로 변경하기 61
5.1.3 인스턴스 메서드인 척하는 static 메서드 주의하기 62
5.1.4 왜 static 메서드를 사용할까? 63
5.1.5 어떠한 상황에서 static 메서드를 사용해야 좋을까? 63

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

5.2 초기화 로직 분산 63
5.2.1 private 생성자 + 팩토리 메서드를 사용해 목적에 따라 초기화하기 66
5.2.2 생성 로직이 너무 많아지면 팩토리 클래스를 고려해 보자 67

5.3 범용 처리 클래스(Common/Util) 68
5.3.1 너무 많은 로직이 한 클래스에 모이는 문제 69
5.3.2 객체 지향 설계의 기본으로 돌아가기 70
5.3.3 횡단 관심사 70

5.4 결과를 리턴하는 데 매개변수 사용하지 않기 71

5.5 매개변수가 너무 많은 경우 75
5.5.1 기본 자료형에 대한 집착 76
5.5.2 의미 있는 단위는 모두 클래스로 만들기 79

5.6 메서드 체인 81
5.6.1 묻지 말고 명령하기 82

6장 조건 분기: 미궁처럼 복잡한 분기 처리를 무너뜨리는 방법 85

6.1 조건 분기가 중첩되어 낮아지는 가독성 86
6.1.1 조기 리턴으로 중첩 제거하기 88
6.1.2 가독성을 낮추는 else 구문도 조기 리턴으로 해결하기 91

6.2 switch 조건문 중복 93
6.2.1 switch 조건문을 사용해서 코드 작성하기 94
6.2.2 같은 형태의 switch 조건문이 여러 개 사용되기 시작 95
6.2.3 요구 사항 변경 시 수정 누락(case 구문 추가 누락) 96
6.2.4 폭발적으로 늘어나는 switch 조건문 중복 98
6.2.5 조건 분기 모으기 99
6.2.6 인터페이스로 switch 조건문 중복 해소하기 101
6.2.7 인터페이스를 switch 조건문 중복에 응용하기(전략 패턴) 105

6.3 조건 분기 중복과 중첩 118
6.3.1 정책 패턴으로 조건 집약하기 119

6.4 자료형 확인에 조건 분기 사용하지 않기 124

6.5 인터페이스 사용 능력이 중급으로 올라가는 첫걸음 127

6.6 플래그 매개변수 128
6.6.1 메서드 분리하기 129
6.6.2 전환은 전략 패턴으로 구현하기 130

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

7장 컬렉션: 중첩을 제거하는 구조화 테크닉 133

7.1 이미 존재하는 기능을 다시 구현하지 말기 134

7.2 반복 처리 내부의 조건 분기 중첩 135
7.2.1 조기 컨티뉴로 조건 분기 중첩 제거하기 136
7.2.2 조기 브레이크로 중첩 제거하기 137

7.3 응집도가 낮은 컬렉션 처리 139
7.3.1 컬렉션 처리를 캡슐화하기 141
7.3.2 외부로 전달할 때 컬렉션의 변경 막기 144

8장 강한 결합: 복잡하게 얽혀서 풀 수 없는 구조 147

8.1 결합도와 책무 149
8.1.1 다양한 버그 153
8.1.2 로직의 위치에 일관성이 없음 154
8.1.3 단일 책임 원칙 154
8.1.4 단일 책임 원칙 위반으로 발생하는 악마 155
8.1.5 책임이 하나가 되게 클래스 설계하기 155
8.1.6 DRY 원칙의 잘못된 적용 157

8.2 다양한 강한 결합 사례와 대처 방법 160
8.2.1 상속과 관련된 강한 결합 160
8.2.2 인스턴스 변수별로 클래스 분할이 가능한 로직 169
8.2.3 특별한 이유 없이 public 사용하지 않기 172
8.2.4 private 메서드가 너무 많다는 것은 책임이 너무 많다는 것 177
8.2.5 높은 응집도를 오해해서 생기는 강한 결합 178
8.2.6 스마트 UI 181
8.2.7 거대 데이터 클래스 181
8.2.8 트랜잭션 스크립트 패턴 182
8.2.9 갓 클래스 183
8.2.10 강한 결합 클래스 대처 방법 184

9장 설계의 건전성을 해치는 여러 악마 185

9.1 데드 코드 186

9.2 YAGNI 원칙 187

9.3 매직 넘버 188

9.4 문자열 자료형에 대한 집착 191

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

9.5 전역 변수 191
9.5.1 영향 범위가 최소화되도록 설계하기 192

9.6 null 문제 193
9.6.1 null을 리턴/전달하지 말기 196
9.6.2 null 안전 197

9.7 예외를 catch하고서 무시하는 코드 198
9.7.1 원인 분석을 어렵게 만듦 198
9.7.2 문제가 발생했다면 소리치기 199

9.8 설계 질서를 파괴하는 메타 프로그래밍 200
9.8.1 리플렉션으로 인한 클래스 구조와 값 변경 문제 200
9.8.2 자료형의 장점을 살리지 못하는 하드 코딩 202
9.8.3 단점을 이해하고 용도를 한정해서 사용하기 204

9.9 기술 중심 패키징 204

9.10 샘플 코드 복사해서 붙여넣기 207

9.11 은 탄환 207

10장 **이름 설계: 구조를 파악할 수 있는 이름** 209

10.1 악마를 불러들이는 이름 211
10.1.1 관심사 분리 212
10.1.2 관심사에 맞는 이름 붙이기 213
10.1.3 포괄적이고 의미가 불분명한 이름 215

10.2 이름 설계하기 - 목적 중심 이름 설계 216
10.2.1 최대한 구체적이고, 의미 범위가 좁고, 특화된 이름 선택하기 217
10.2.2 존재가 아니라 목적을 기반으로 하는 이름 생각하기 218
10.2.3 어떤 비즈니스 목적이 있는지 분석하기 219
10.2.4 소리 내어 이야기해 보기 219
10.2.5 이용 약관 읽어 보기 221
10.2.6 다른 이름으로 대체할 수 없는지 검토하기 222
10.2.7 결합이 느슨하고 응집도가 높은 구조인지 검토하기 223

10.3 이름 설계 시 주의 사항 224
10.3.1 이름에 관심 갖기 224
10.3.2 사양 변경 시 '의미 범위 변경' 경계하기 224
10.3.3 대화에는 등장하지만 코드에 등장하지 않는 이름 주의하기 224
10.3.4 수식어를 붙여서 구별해야 하는 경우는 클래스로 만들어 보기 225

10.4 의미를 알 수 없는 이름 230
10.4.1 기술 중심 명명 231
10.4.2 로직 구조를 나타내는 이름 232
10.4.3 놀람 최소화 원칙 233

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

10.5 구조에 악영향을 미치는 이름 235
10.5.1 데이터 클래스처럼 보이는 이름 235
10.5.2 클래스를 거대하게 만드는 이름 236
10.5.3 상황에 따라 의미가 달라질 수 있는 이름 241
10.5.4 일련번호 명명 243

10.6 이름을 봤을 때, 위치가 부자연스러운 클래스 244
10.6.1 '동사 + 목적어' 형태의 메서드 이름 주의하기 244
10.6.2 가능하면 메서드의 이름은 동사 하나로 구성되게 하기 246
10.6.3 부적절한 위치에 있는 boolean 메서드 247

10.7 이름 축약 249
10.7.1 의도를 알 수 없는 축약 249
10.7.2 기본적으로 이름은 축약하지 말기 250
10.7.3 이름을 축약할 수 있는 경우 250

11장 주석: 유지 보수와 변경의 정확성을 높이는 주석 작성 방법 251

11.1 내용이 낡은 주석 252
11.1.1 주석은 실제 코드가 아님을 이해하기 253
11.1.2 로직의 동작을 설명하는 주석은 낡기 쉽다 254

11.2 주석 때문에 이름을 대충 짓는 예 254

11.3 의도와 사양 변경 시 주의 사항을 읽는 이에게 전달하기 256

11.4 주석 규칙 정리 257

11.5 문서 주석 257

12장 메서드(함수): 좋은 클래스에는 좋은 메서드가 있다 261

12.1 반드시 현재 클래스의 인스턴스 변수 사용하기 262

12.2 불변을 활용해서 예상할 수 있는 메서드 만들기 263

12.3 묻지 말고 명령하라 263

12.4 커맨드/쿼리 분리 265

12.5 매개변수 267
12.5.1 불변 매개변수로 만들기 267
12.5.2 플래그 매개변수 사용하지 않기 267
12.5.3 null 전달하지 않기 267
12.5.4 출력 매개변수 사용하지 않기 268
12.5.5 매개변수는 최대한 적게 사용하기 268

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

12.6 리턴 값 268
12.6.1 '자료형'을 사용해서 리턴 값의 의도 나타내기 268
12.6.2 null 리턴하지 않기 270
12.6.3 오류는 리턴 값으로 리턴하지 말고 예외 발생시키기 270

13장 모델링: 클래스 설계의 토대 273

13.1 악마를 불러들이기 쉬운 User 클래스 274

13.2 모델링으로 접근해야 하는 구조 277
13.2.1 시스템이란? 277
13.2.2 시스템 구조와 모델링 278
13.2.3 소프트웨어 설계와 모델링 279

13.3 안 좋은 모델의 문제점과 해결 방법 281
13.3.1 User와 시스템의 관계 282
13.3.2 가상 세계를 표현하는 정보 시스템 282
13.3.3 목적별로 모델링하기 283
13.3.4 모델은 대상이 아니라 목적 달성의 수단 285
13.3.5 단일 책임이란 단일 목적 286
13.3.6 모델을 다시 확인하는 방법 286
13.3.7 모델과 구현은 반드시 서로 피드백하기 287

13.4 기능성을 좌우하는 모델링 289
13.4.1 숨어 있는 목적 파악하기 290
13.4.2 기능성을 혁신하는 '깊은 모델' 291

14장 리팩터링: 기존의 코드를 성장시키는 기술 295

14.1 리팩터링의 흐름 296
14.1.1 중첩을 제거하여 보기 좋게 만들기 297
14.1.2 의미 단위로 로직 정리하기 298
14.1.3 조건을 읽기 쉽게 하기 299
14.1.4 무턱대고 작성한 로직을 목적을 나타내는 메서드로 바꾸기 300

14.2 단위 테스트로 리팩터링 중 실수 방지하기 301
14.2.1 코드 과제 정리하기 302
14.2.2 테스트 코드를 사용한 리팩터링 흐름 303

14.3 불확실한 사양을 이해하기 위한 분석 방법 312
14.3.1 사양 분석 방법 1: 문서화 테스트 312
14.3.2 사양 분석 방법 2: 스크래치 리팩터링 314

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

14.4 IDE의 리팩터링 기능 315
14.4.1 리네임(이름 변경) 315
14.4.2 메서드 추출 317

14.5 리팩터링 시 주의 사항 318
14.5.1 기능 추가와 리팩터링 동시에 하지 않기 318
14.5.2 작은 단계로 실시하기 319
14.5.3 불필요한 사양은 제거 고려하기 319

15장 설계의 의의와 설계를 대하는 방법 323

15.1 이 책은 어떤 설계를 주제로 집필한 것인가? 324

15.2 설계하지 않으면 개발 생산성이 저하된다 325
15.2.1 요인 1: 버그가 발생하기 쉬운 구조 326
15.2.2 요인 2: 가독성이 낮은 구조 326
15.2.3 나무꾼의 딜레마 326
15.2.4 열심히 일했지만 생산성이 나쁨 327
15.2.5 국가 규모의 경제 손실 327

15.3 소프트웨어와 엔지니어의 성장 가능성 328
15.3.1 엔지니어에게 자산이란 무엇인가? 329
15.3.2 레거시 코드는 발전을 막음 329
15.3.3 레거시 코드는 고품질 설계 경험을 막음 329
15.3.4 레거시 코드는 시간을 낭비하게 만듦 330

15.4 문제 해결하기 330
15.4.1 문제를 인식하지 못하면 설계에 대한 생각 자체가 떠오르지 않음 330
15.4.2 인지하기 쉬운 문제와 인지하기 어려운 문제가 있음 330
15.4.3 이상적인 형태를 알아야 문제를 인식할 수 있음 332
15.4.4 변경 용이성을 비교할 수 없는 딜레마 333

15.5 코드의 좋고 나쁨을 판단하는 지표 334
15.5.1 실행되는 코드의 줄 수 334
15.5.2 순환 복잡도 336
15.5.3 응집도 336
15.5.4 결합도 337
15.5.5 청크 337

15.6 코드 분석을 지원하는 다양한 도구 339
15.6.1 Code Climate Quality 339
15.6.2 Understand 339
15.6.3 Visual Studio 340

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

15.7 설계 대상과 비용 대비 효과 341
15.7.1 파레토의 법칙(80:20의 법칙) 342
15.7.2 코어 도메인: 서비스의 중심 영역 342
15.7.3 중점 설계 대상 선정에는 비즈니스 지식이 필요함 343
15.8 시간을 다스리는 능력자 되기 344

16장 설계를 방해하는 개발 프로세스와의 싸움 345

16.1 커뮤니케이션 346
16.1.1 커뮤니케이션이 부족하면 설계 품질에 문제가 발생 346
16.1.2 콘웨이 법칙 347
16.1.3 심리적 안정성 348
16.2 설계 348
16.2.1 '빨리 끝내고 싶다'는 심리가 품질 저하의 함정 348
16.2.2 나쁜 코드를 작성하는 것이 좋은 코드를 작성하는 것보다 오래 걸린다 349
16.2.3 클래스 설계와 구현 피드백 사이클 돌리기 350
16.2.4 한 번에 완벽하게 설계하려고 들지 말고, 사이클을 돌리며 완성하기 350
16.2.5 '성능이 떨어질 수 있으니 클래스를 작게 나누지 말자'는 맞는 말일까? 351
16.2.6 설계 규칙을 다수결로 결정하면 코드 품질은 떨어진다 351
16.2.7 설계 규칙을 정할 때 중요한 점 352
16.3 구현 353
16.3.1 깨진 유리창 이론과 보이스카우트 규칙 353
16.3.2 기존의 코드를 믿지 말고, 냉정하게 파악하기 354
16.3.3 코딩 규칙 사용하기 356
16.3.4 명명 규칙 357
16.4 리뷰 358
16.4.1 코드 리뷰 구조화하기 358
16.4.2 코드를 설계 시점에 리뷰하기 359
16.4.3 존중과 예의 359
16.4.4 정기적으로 개선 작업 진행하기 361
16.5 팀의 설계 능력 높이기 362
16.5.1 영향력을 갖는 규모까지 동료 모으기 362
16.5.2 천리길도 한 걸음부터 364
16.5.3 백문이 불여일견 364
16.5.4 팔로우업 스터디 진행하기 364
16.5.5 스터디 그룹에서 발생할 수 있는 문제 해결 노하우 365
16.5.6 리더와 매니저에게 설계의 중요성과 비용 대비 효과 설명하기 366
16.5.7 설계 책임자 세우기 367

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

17장 설계 기술을 계속해서 공부하려면 369

17.1 추천 도서 370
17.1.1《현장에서 유용한 시스템 설계 원칙(現場で役立つシステム設計の原則)》 370
17.1.2《읽기 좋은 코드가 좋은 코드다》 371
17.1.3《리팩터링 2판》 371
17.1.4《클린 코드》 372
17.1.5《레거시 코드 활용 전략》 372
17.1.6《레거시 소프트웨어 리엔지니어링(Re-Engineering Legacy Software)》 372
17.1.7《레거시 코드를 넘어서(Beyond Legacy Code)》 373
17.1.8《엔지니어링 조직론으로의 초대(エンジニアリング組織論への招待)》 373
17.1.9《프로그래밍의 원칙(プリンシプル オブ プログラミング)》 374
17.1.10《클린 아키텍처》 374
17.1.11《도메인 주도 설계》 374
17.1.12《설계를 통한 보안(Secure by Design)》 375
17.1.13《도메인 주도 설계 철저 입문》 376
17.1.14《도메인 주도 설계 모델링/구현 가이드(ドメイン駆動設計 モデリング/実装ガイド)》 377
17.1.15《도메인 주도 설계 샘플 코드와 FAQ(ドメイン駆動設計 サンプルコード＆FAQ)》 377
17.1.16《테스트 주도 개발》 377

17.2 설계 스킬을 높이는 학습 방법 379
17.2.1 학습을 위한 지침 379
17.2.2 악마의 구조를 파악하는 연습 380
17.2.3 리팩터링으로 설계 기술력 높이기 381
17.2.4 동작하는 코드를 작성했다면, 다시 설계하고 커밋하기 383
17.2.5 설계 기술서를 읽으며 더 높은 목표 찾기 384

옮긴이의 글 385
참고 문헌 386
찾아보기 390

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

들어가며

소프트웨어를 개발하면서 다음과 같은 경험을 해 본 적 있지 않나요?

- 어떤 곳의 코드를 변경하니 다른 곳에서 버그가 발생했다.
- 코드를 변경했을 때 영향을 끼치는 부분이 어디인지 여기저기 찾아다녔다.
- 코드를 읽고 이해하는 데만 하루가 걸렸다.
- 쉽게 생각했던 사양 변경과 버그 수정에 며칠을 소비했다.

이럴 때 괴로워하면서도, '원인이 무엇인지 모르겠다'고 생각한 적이 있지 않나요? 소스 코드를 작성할 때 무언가 문제가 있는 것 같다고 느끼긴 했지만, '어떻게 개선해야 할지 모르겠다'고 생각하진 않았나요?

일단 원인을 알 수 없는 이유는 무엇일까요? 이는 '이상적인 구조'를 모르기 때문입니다. 소프트웨어와 관련된 비유는 아니지만, 정사각형은 '네 변의 길이가 모두 같고, 내각이 모두 직각인 도형'이라는 것을 알고 있을 것입니다. 이 정사각형의 정의를 알고 있기 때문에, 변의 길이가 일부 다르거나 내각의 일부가 직각이 아닌 도형을 보았을 때 '이것은 정사각형이 아니다'라고 인식할 수 있습니다. 소프트웨어 설계에서도 마찬가지입니다. '이상적인 구조'를 알면 좋지 않은 구조를 명확하게 인지할 수 있습니다.

서양에서는 악마의 정체를 알면 지배하고 복종시킬 수 있다고 믿었습니다. 흑사병 같은 전염병이 창궐했을 때, 사람들은 '악마' 때문이라며 공포에 떨었으나 병원균을 발견하자, 전염병에 대처하는 방법이 획기적으로 발전했습니다. 악마의 실체(즉 정체)를 인식할 수만 있어도, 제대로 대처할 수 있게 됩니다.

이 책에서는 개발 능력을 떨어뜨리고 소프트웨어의 성장을 방해하는 설계/구현상의 문제를 '악마'에 비유합니다. 악마의 정체를 알면 대처할 수 있는 것처럼, 설계/구현상의 문제도 정체를 파악할 수 있으면 올바르게 대처할 수 있습니다.

이 책은 소프트웨어 개발에 숨어 있는 악마를 인식하고 퇴치할 수 있게 도와주는 설계 테크닉을 다룹니다.

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

개발 효율을 떨어뜨리고 악마를 불러들이는 나쁜 구조에는 어떤 것이 있는지 하나하나 확인해 보고, 원인과 해결 방법을 설명합니다. 이를 통해 여러분은 '악마의 정체를 꿰뚫어보는 눈'과 '악마를 퇴치하는 무기'를 갖추게 될 것입니다.

이 책을 활용해서 악마를 퇴치하여 빠르게 성장하는 소프트웨어를 만들어 봅시다.

이 책의 대상 독자

이 책은 객체 지향 프로그래밍 언어를 사용하는 소프트웨어 개발자를 대상으로 쓰였습니다.

그중에서 (1) 객체 지향 프로그래밍 언어에 대한 기초 지식은 있지만, 설계를 잘 모르거나 자신감이 없는 사람, (2) 지금부터 설계를 제대로 배우고자 하는 사람을 대상으로 합니다.

객체 지향 설계를 다루는 이유

저는 애플리케이션 아키텍트로서, 시스템 설계와 관련된 일을 하고 있습니다. 예를 들어 유지 보수가 어려워진 시스템을 재설계하거나, 새로운 시스템을 만들 때 확장성을 높이기 위한 설계를 하고 있습니다.

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

이 책에는 샘플 코드로 나쁜 코드가 아주 많이 등장합니다. 이는 모두 실제로 본 적 있는 코드를 이해하기 쉽게 사례를 조금 바꾸어 재구성한 것입니다.

저는 이러한 나쁜 코드를 객체 지향 설계로 해결하고 개선하는 일을 합니다. 따라서 이 책은 저의 실무 노하우를 담고 있습니다.

설계는 문제를 효율적으로 해결하는 구조를 만드는 것입니다. 객체 지향에는 복잡한 로직을 구분하고 정리하여 질서 정연한 구조로 개선하는 다양한 설계 기법이 있습니다. 이 책에서는 나쁜 코드를 좋은 코드로 바꾸는 실천적인 객체 지향 설계 기법을 설명합니다.

이 책에서 사용하는 프로그래밍 언어

이 책의 샘플 코드는 일부를 제외하고 모두 자바로 작성되어 있습니다.

자바를 사용하는 이유는 사용자가 많은 프로그래밍 언어이기 때문입니다. 또한 자바로 된 설계 관련 자료가 굉장히 많으므로, 이 책을 끝낸 이후에도 관련 내용을 공부하기 좋습니다.

저는 C#, C++, 루비, 자바스크립트를 포함한 여러 객체 지향 프로그래밍 언어를 사용해 보았습니다. 그래서 이 책은 자바뿐만 아니라 모든 객체 지향 언어에서 활용할 수 있는 설계 방법을 중심으로 집필했습니다. 자바에서만 활용할 수 있는 기능이나 프레임워크 등을 다루진 않았습니다. 따라서 객체 지향 프로그래밍 언어를 사용하는 개발자라면, 다른 프로그래밍 언어로 대체해서 생각하는 데 문제없을 것입니다.

또한 저는 실무적으로 웹 애플리케이션, 윈도우 애플리케이션, 임베디드 소프트웨어 등 다양한 분야에 개발 경력이 있으며, 개인적으로 게임 개발 경험도 있습니다. 이러한 경험을 기반으로, 이 책은 웹 애플리케이션뿐만 아니라, 다양한 소프트웨어 개발에 폭넓게 활용할 수 있는 설계 기법을 중심으로 집필했습니다.

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

이 책의 구성

장	내용	단계
1. 잘못된 구조의 문제 깨닫기	잘못된 구조의 폐해를 통해 설계의 중요성을 다룹니다.	입문
2. 설계 첫걸음	잘못된 구조를 개선하는 간단한 예를 통해서, 설계가 무엇을 의미하는지 다룹니다.	입문
3. 클래스 설계: 모든 것과 연결되는 설계 기반	이 책 전체의 기반이 되는 클래스와 객체 지향 설계의 기초를 다룹니다.	실전
4. 불변 활용하기: 안정적으로 동작하게 만들기	예측할 수 있는 코드를 작성하는 데 필요한 불변성을 다룹니다.	실전
5. 응집도: 흩어져 있는 것들	코드가 여러 곳에 분산되는 문제를 해결하는 방법을 다룹니다.	실전
6. 조건 분기: 미궁처럼 복잡한 분기 처리를 무너뜨리는 방법	복잡한 조건 분기를 정리하고 구조화하는 방법을 다룹니다.	실전
7. 컬렉션: 중첩을 제거하는 구조화 테크닉	복잡한 리스트 처리를 정리하고 구조화하는 방법을 다룹니다.	실전
8. 강한 결합: 복잡하게 얽혀서 풀 수 없는 구조	여러 역할의 코드가 결합되어 유지 보수하기 어려워진 클래스를 분할하는 방법을 다룹니다.	실전
9. 설계의 건전성을 해치는 여러 악마	앞에서 소개하지 못한 나쁜 코드와 그 대처 방안을 다룹니다.	실전
10. 이름 설계: 구조를 파악할 수 있는 이름	이름과 구조의 밀접한 관계를 살펴보고, 구조를 개선하는 데 도움이 되는 이름 설계를 다룹니다.	실전
11. 주석: 유지 보수와 변경의 정확성을 높이는 주석 작성 방법	읽는 사람을 혼란스럽게 만드는 나쁜 주석과 읽는 사람에게 도움을 주는 좋은 주석을 다룹니다.	실전
12. 메서드(함수): 좋은 클래스에는 좋은 메서드가 있다	메서드 설계 방법을 집중적으로 다룹니다.	실전
13. 모델링: 클래스 설계의 토대	클래스의 구분과 구조의 기반이 되는 모델링에 대해 다룹니다.	심화
14. 리팩터링: 기존의 코드를 성장시키는 기술	이미 구현된 나쁜 코드를 좋은 코드로 개선하는 리팩터링 방법에 대해 다룹니다.	심화
15. 설계의 의의와 설계를 대하는 방법	이 책의 설계 의의인 '성장성'을 중심으로 설계를 다시 한 번 생각해 봅니다.	심화
16. 설계를 방해하는 개발 프로세스와의 싸움	코드 품질을 악화시키는 개발 프로세스 관련 문제들을 살펴봅니다.	심화
17. 설계 기술을 계속해서 공부하려면	이 책을 읽은 후 보면 좋을 참고 도서를 소개하고, 학습법을 다룹니다.	심화

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

감사의 글

이 책은 많은 분의 도움으로 완성되었습니다. 도움 주신 분들을 소개하겠습니다.

일단 이 책을 검토해 주신 마스다 토오루(増田亨) 님, 카토 준이치(加藤潤一) 님, 타니모토 신(谷本心) 님, 오카무라 켄(岡村謙) 님에게 감사하다는 말을 전합니다. 훌륭한 엔지니어 분들의 리뷰 덕분에, 이 책의 품질과 가치가 크게 향상되었습니다.

또한 지금까지 저와 함께 일한 모든 동료 분께 감사드립니다. 동료들과 함께 일하면서 얻은 수많은 경험이 이 책을 만드는 바탕이 되었습니다.

설계 커뮤니티에 있는 모든 분들에게도 감사의 말을 전합니다. 커뮤니티를 통해 얻은 지식 덕분에 이 책이 풍성해질 수 있었습니다. 또한 여러분과의 소통이 지금까지 제가 설계를 열심히 할 수 있던 원동력이 되었습니다.

부모님께도 감사의 말을 전합니다. 어렸을 때 비싼 컴퓨터를 사 주신 부모님 덕분에 제가 프로그래머의 길을 걸을 수 있었고, 지금까지 계속할 수 있었습니다.

책 집필을 제안해 주시고, 편집에 참여해 주신 기술평론사의 노다 다이키(野田大貴) 님께도 감사의 말을 전합니다. 첫 집필이라 모르는 부분이 많았는데, 친절하게 도와주셨습니다.

게임 제작 도구 RPG Maker(RPG 쯔꾸르) 관계자 분들과 이 도구를 활용해서 게임을 만드는 모든 분께도 감사의 말을 전합니다. 제가 RPG Maker로 만든 설계에 관한 풍자 동영상을 노다 다이키 님께서 보고 연락을 주셔서 이 책을 집필할 수 있었습니다. 많은 게임 제작자 분들의 아이디어에 자극을 받아 동영상을 만들 수 있었습니다.

그리고 지금까지 함께 가정을 꾸려 온 아내와 아이에게도 감사의 말을 전합니다. 가족의 지원이 있었기에 1년 9개월에 걸친 긴 집필 기간을 버틸 수 있었습니다.

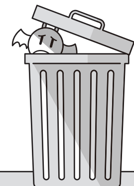
그 밖에도 여러 관계자 분께 감사와 경의를 표합니다. 모두 감사합니다.

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

1장

잘못된 구조의 문제 깨닫기

1.1 의미를 알 수 없는 이름
1.2 이해하기 어렵게 만드는 조건 분기 중첩
1.3 수많은 악마를 만들어 내는 데이터 클래스
1.4 악마 퇴치의 기본



미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

좋은 구조[1]로 개선하기 위해서는 일단 나쁜 구조의 폐해를 인지해야 합니다. 그런 다음 폐해를 개선할 수 있는 좋은 구조를 배우면, 나쁜 구조와 좋은 구조의 차이를 파악하여 설계를 개선할 수 있습니다.

과거에 참여한 개발 프로젝트에 여러 문제가 생긴 적이 있습니다. 어떻게 해도 버그[2]가 계속 발생했으며, 코드 품질이 배포할 수 있는 수준에 이르지 못했습니다. 야근이 일상이 되었고, 하루하루 일상이 피폐해져 갔습니다.

처음에는 문제들의 원인조차 알지 못했습니다. 그런데 다양한 기술서를 접한 뒤, 코드를 이해하기 쉽게 만들고, 버그 발생 가능성을 줄이는 좋은 설계가 있음을 알게 되었습니다. 그 덕분에 문제의 원인이 각종 나쁜 구조임을 깨달았습니다.

설계의 중요성을 깨닫기 위해서는 설계를 소홀히 했을 때 어떠한 폐해가 발생하는지 알아야 합니다. 여기서 폐해란 다음과 같은 것을 말합니다.

- 코드를 읽고 이해하는 데 시간이 오래 걸림.
- 버그가 계속해서 발생함.
- 나쁜 구조로 인해서 더 나쁜 구조가 만들어짐.

이 장에서는 이처럼 나쁜 구조로 인해 발생할 수 있는 폐해를 여러 예제를 들어 간단하게 살펴보겠습니다.

1.1 의미를 알 수 없는 이름

좋지 않은 이름이 일으키는 악영향을 소개하겠습니다.

코드 1.1을 보면 이 로직이 무엇을 의미하는지 알 수 있나요?

1 프로그램 구조는 클래스와 메서드 등 여러 가지로 세세하게 나눌 수 있습니다. 이 책에서는 특별한 언급이 없는 이상, '구조'는 프로그램 구조 전반을 의미합니다.
2 버그는 '시스템 명세를 만족하지 못하는 프로그램의 결함'을 의미합니다.

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

코드 1.1 기술 중심 명명

```
class MemoryStateManager {
  void changeIntValue01(int changeValue) {
    intValue01 -= changeValue;
    if (intValue01 < 0) {
      intValue01 = 0;
      updateState02Flag();
    }
  }
  ...
}
```

아마 전혀 알 수 없을 것입니다.

그래도 조금 살펴보면 자료형 이름을 나타내는 `Int`, 메모리 제어를 나타내는 `Memory`와 `Flag` 등 프로그래밍이나 컴퓨터 용어를 기반으로 이름 붙였음을 알 수 있습니다. 이처럼 기술을 기반으로 이름 붙이는 것을 **기술 중심 명명**이라 부릅니다(10.4.1절 참고).[3]

이어서 다음 예를 살펴봅시다.

코드 1.2 일련번호 명명

```
class Class001 {
  void method001();
  void method002();
  void method003();
  ...
}
```

코드 1.2처럼 클래스와 메서드에 번호를 붙여서 이름 짓는 것을 **일련번호 명명**이라고 합니다(10.5.4절 참고).

이와 같이 기술을 기반으로 이름을 짓거나, 일련번호를 매겨 이름을 지으면 코드에서 어떠한 의도도 읽어 낼 수 없습니다.

이렇게 이름 지은 코드는 이해하기 어렵습니다. 읽고 이해하는 데 시간이 오래 걸립니다. 게다가 충분히 이해하지 못한 상태로 코드를 변경하면 버그가 발생합니다.

3 이 책에서 설명을 간단하게 하고자, 몇 군데에서 `Manager`라는 이름을 사용했습니다. 10.5.2절에서 언급하겠지만, `Manager`라는 이름도 문제를 많이 발생시키는 좋지 않은 이름입니다.

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

이러한 위험을 줄이고자 스프레드시트 등을 사용해 일람표를 만들기도 합니다. 각 클래스와 메서드의 역할과 기능을 설명하는 문서입니다. 하지만 이런 문서는 바쁜 업무로 인해 유지 보수가 거의 이루어지지 않습니다. 코드는 계속 해서 변경되는데 문서 유지 보수가 따라가지 못하면, 문서는 거짓말을 시작합니다. 결국 정확하지 않은 문서는 오히려 후임 담당자가 코드를 읽고 이해하기 어렵게 하며, 그 결과 버그가 더 쉽게 발생하게 됩니다.

그리고 의도를 제대로 이해하지 못하고 코드를 변경하면, 로직이 필요 이상으로 복잡해질 수도 있습니다. 따라서 의도와 목적을 드러내는 이름을 사용하는 것이 좋습니다. 이렇게만 해도 구조가 간단하고 명확해집니다(10장 참고).

1.2 이해하기 어렵게 만드는 조건 분기 중첩

조건 분기는 조건에 따라 처리 방식을 다르게 하는 데 사용되는 프로그래밍 언어의 기본 제어 구조입니다. 그런데 조건 분기를 어설프게 사용하면, 악마가 되어 개발자를 괴롭힙니다.

코드 1.3은 RPG(롤플레잉 게임)에서 마법 발동의 조건을 구현한 예입니다.

코드 1.3 여러 번 중첩된 로직

```
// 살아 있는지 판정
if (0 < member.hitPoint) {
  // 움직일 수 있는지 판정
  if (member.canAct()) {
    // 매직포인트에 여유가 있는지 판정
    if (magic.costMagicPoint <= member.magicPoint) {
      member.consumeMagicPoint(magic.costMagicPoint);
      member.chant(magic);
    }
  }
}
```

RPG에서 어떤 멤버에게 마법을 쓰라고 지시한다고 해서 그 멤버가 마법을 무조건 발동하지는 못합니다. 자신의 순서가 돌아오기 전에 적의 공격을 받아서

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

전투 불능 상태가 될 수도 있으며, 수면 또는 마비 등의 마법에 걸려 움직이지 못할 수도 있기 때문입니다. 따라서 앞의 코드처럼 '마법을 발동할 수 있는 상태인지' 여러 번 판정해야 합니다.

이 코드는 if 조건문 내부에 if 조건문, 그리고 그 안에 또 if 조건문이 있는 형태입니다. 이러한 상태를 if 조건문이 **중첩**되어 있다고 합니다.

중첩이 많을수록 코드의 가독성이 나빠집니다. 어디서부터 어디까지가 if 조건문 처리 블록[4]인지 확인하기 힘들기 때문입니다. 만약 코드 1.4처럼 중첩되어 있다면, 코드를 읽고 이해하기 정말 힘들 것입니다.

코드 1.4 거대한 중첩

```
if (조건) {
  //
  // 수십~수백 줄의 코드
  //
  if (조건) {
    //
    // 수십~수백 줄의 코드
    //
    if (조건) {
      //
      // 수십~수백 줄의 코드
      //
      if (조건) {
        //
        // 수십~수백 줄의 코드
        //
      }
    }
    //
    // 수십~수백 줄의 코드
    //
  }
  //
  // 수십~수백 줄의 코드
  //
}
```

4 {}(중괄호)로 이루어진 처리 범위

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

코드를 보고 무슨 이런 코드가 다 있나 싶기도 하지만, 이런 코드는 실제로 존재합니다.

코드를 이렇게 작성하면 조건이 복잡해질수록 코드를 읽고 이해하기 힘듭니다. 이해가 힘들면 디버깅과 기능 변경에 더 오랜 시간이 걸립니다. 게다가 분기 로직을 정확하게 이해하지 못하고 기능을 변경하면, 버그가 발생할 수도 있습니다(자세한 내용은 6.1절에서 설명합니다).

1.3 수많은 악마를 만들어 내는 데이터 클래스

데이터 클래스는 설계가 제대로 이루어지지 않은 소프트웨어에서 빈번하게 등장하는 클래스 구조입니다. 데이터 클래스는 단순한 구조이지만, 수많은 악마를 만들어 낼 수 있습니다.

금액을 다루는 서비스를 예로 들어 데이터 클래스의 어떤 점이 나쁜지 살펴봅시다.

업무 계약을 다루는 서비스에서 계약 금액을 처리하는 요구 사항을 클래스로 구현해야 한다고 합시다. 아무 생각 없이 구현하면, 코드 1.5와 같은 클래스 구조가 만들어집니다.

코드 1.5 데이터밖에 없는 클래스 구조

```
// 계약 금액
public class ContractAmount {
  public int amountIncludingTax;  // 세금 포함 금액
  public BigDecimal salesTaxRate; // 소비세율
}
```

세금이 포함된 금액과 소비세율을 public 인스턴스 변수로 갖고 있으므로, 클래스 밖에서도 데이터를 자유롭게 변경할 수 있는 구조입니다. 이처럼 데이터를 갖고 있기만 하는 클래스를 **데이터 클래스**라고 부릅니다.

그런데 데이터 클래스에는 데이터뿐만 아니라, 세금이 포함된 금액을 계산하는 로직도 필요한데, 이러한 계산 로직을 데이터 클래스가 아닌 다른 클래스

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

에 구현하는 일이 벌어지곤 합니다. 설계를 따로 고려하지 않아 생기는 일입니다. 예를 들어 코드 1.6처럼 다른 클래스에 계산 로직이 구현된 경우를 본 적 있지 않나요?

코드 1.6 ContractManager에 작성된 금액 계산 로직

```
// 계약을 관리하는 클래스
public class ContractManager {
  public ContractAmount contractAmount;

  // 세금 포함 금액 계산
  public int calculateAmountIncludingTax(int amountExcludingTax,
                                         BigDecimal salesTaxRate) {
    BigDecimal multiplier = salesTaxRate.add(new BigDecimal("1.0"));
    BigDecimal amountIncludingTax =
        multiplier.multiply(new BigDecimal(amountExcludingTax));
    return amountIncludingTax.intValue();
  }

  // 계약 체결
  public void conclude() {
    // 생략
    int amountIncludingTax =
        calculateAmountIncludingTax(amountExcludingTax, salesTaxRate);
    contractAmount = new ContractAmount();
    contractAmount.amountIncludingTax = amountIncludingTax;
    contractAmount.salesTaxRate = salesTaxRate;
    // 생략
  }
}
```

작은 규모의 애플리케이션이라면 이러한 구조가 특별히 문제되지 않습니다. 하지만 애플리케이션의 규모가 커진다면, 수많은 악마를 불러들입니다.

어떤 악마가 나타나는지 차근차근 살펴봅시다.

1.3.1 사양을 변경할 때 송곳니를 드러내는 악마

업무 계약 서비스에서 소비세와 관련된 사양이 변경되었다고 합시다. 구현 담당자는 소비세율과 관련된 로직을 변경했습니다.

그런데 며칠이 지나 '소비세율이 변경되지 않았다'라는 장애 보고가 올라왔습니다. 원인을 조사해 보니, 다른 곳에도 세금 포함 금액을 계산하는 로직이

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

있었던 것입니다. 마찬가지로 구현 담당자는 이곳의 로직도 수정했습니다.

그런데 얼마 지나지 않아서, 또다시 '소비세율이 변경되지 않았다'라는 장애 보고가 올라왔습니다. 조사해 보니 또 다른 곳에도 세금 포함 금액을 계산하는 로직이 있었습니다.

'이런 부분이 더 있는 게 아닐까?'라고 생각한 담당자는 소비세와 관련된 부분을 소스 코드 전체에서 찾기 시작했습니다. 그리고 놀랍게도 세금 포함 금액을 계산하는 로직이 수십 곳에 있음을 확인했습니다.[5]

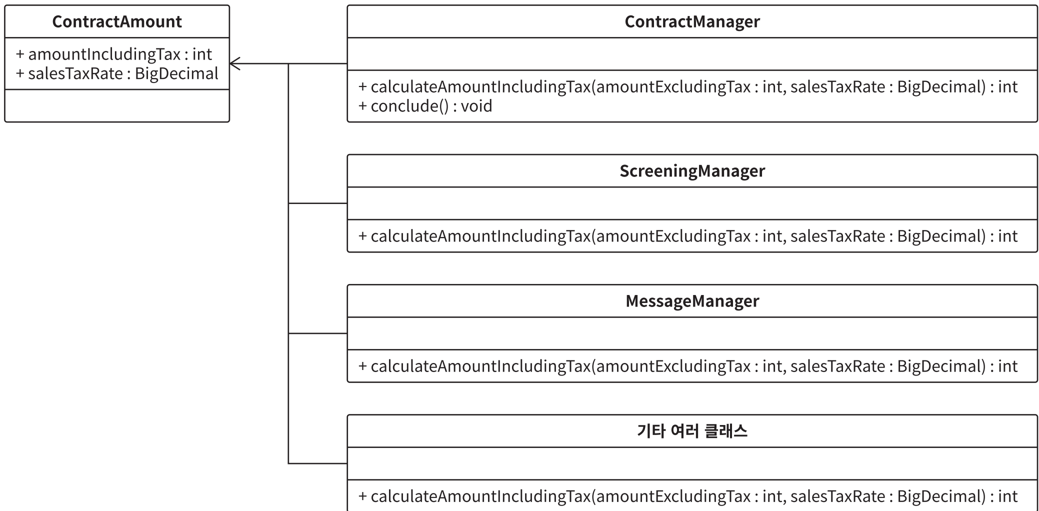


그림 1.1 데이터 클래스로 인해 발생하는 중복

왜 이런 일이 생긴 것일까요? 일단 세금 포함 금액은 여러 상황에서 필요하므로 여러 곳에 구현되기 쉽습니다.

계산 로직을 어느 한곳에 만들어 두면, 사람들이 모두 그것만 사용하고 따로 구현하지는 않겠지 하고 생각할 수도 있습니다. 하지만 설계에 관심이 없다면, 필요한 로직이 이미 구현되어 있다는 사실을 모르고 따로 구현해 버릴 수도 있습니다.

5 실제로 제가 경험했던 코드입니다. 이때 저를 포함한 담당자 전체가 소비세 관련 로직을 확인하고 변경하는 데 굉장히 많은 시간과 노력을 들였습니다.

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

이런 상황은 데이터를 담고 있는 클래스와 데이터를 사용하는 계산 로직이 멀리 떨어져 있을 때 자주 일어납니다. 둘이 떨어져 있으므로, 계산 로직의 존재 자체를 몰라 계속 구현하는 것입니다.

이처럼 데이터와 로직 등이 분산되어 있는 것을 **응집도**가 낮은 구조라고 합니다. 그럼 응집도가 낮아 생길 수 있는 여러 가지 문제를 살펴봅시다.

1.3.2 코드 중복

관련된 코드가 서로 멀리 떨어져 있으면, 관련된 것끼리 묶어서 파악하기 힘듭니다.

업무 계약 서비스 예시에서 살펴본 것처럼, 이미 기능이 구현되어 있는데도 동료 개발자들이 '이 기능이 아직 구현되어 있지 않구나?'라고 오해하고, 같은 로직을 여러 곳에 구현할 수도 있습니다. 의도하지 않게 **코드 중복**이 발생하는 것입니다.

1.3.3 수정 누락

코드 중복이 많으면, 사양이 변경될 때 중복된 코드를 모두 고쳐야 합니다. 하지만 이 과정에서 일부 코드를 놓칠 수 있으며, 결국 버그를 낳습니다.

1.3.4 가독성 저하

가독성이란 코드의 의도나 처리 흐름을 얼마나 빠르고 정확하게 읽고 이해할 수 있는지를 나타내는 지표입니다. 코드가 분산되어 있으면, 중복된 코드를 포함해서 관련된 정보를 다 찾는 것만으로도 시간이 오래 걸립니다. 따라서 가독성이 떨어지는 것입니다.

작은 애플리케이션이라면 문제없을 수도 있지만, 수만 또는 수십만 줄의 소스 코드에 기능이 분산되어 있다면 시간뿐만 아니라 체력과 정신력도 쓸데없이 소모될 것입니다.

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/

1.3.5 초기화되지 않은 상태(쓰레기 객체)

코드 1.7의 코드가 어떻게 실행될지 예측해 봅시다.

코드 1.7 쓰레기 객체

```
ContractAmount amount = new ContractAmount();
System.out.println(amount.salesTaxRate.toString());
```

코드를 실행하면 `NullPointerException`이 발생합니다. 소비세율 `salesTaxRate`는 `BigDecimal`로 정의되어 있으므로, 따로 초기화하지 않으면 null이 들어갑니다. `ContractAmount`가 추가로 초기화해야 하는 클래스라는 것을 모르면, 버그가 발생하기 쉬운 불완전한 클래스입니다.

이처럼 '초기화하지 않으면 쓸모 없는 클래스' 또는 '초기화하지 않은 상태가 발생할 수 있는 클래스'를 안티 패턴[6] 쓰레기 객체라고 부릅니다.

1.3.6 잘못된 값 할당

값이 잘못되었다는 것은 요구 사항에 맞지 않음을 의미합니다. 예를 들어 다음과 같은 상태입니다.

- 주문 건수가 음수가 나오는 경우
- 게임에서 히트포인트(HP) 값이 최댓값을 넘는 경우

현재 데이터 클래스는 코드 1.8처럼 소비세율을 음수로 대입해도 값이 들어갑니다. 따라서 잘못된 값이 쉽게 들어갈 수 있는 구조입니다.

코드 1.8 잘못된 값이 들어간 경우

```
ContractAmount amount = new ContractAmount();
amount.salesTaxRate = new BigDecimal("-0.1");
```

잘못된 값이 들어가지 않게, 데이터 클래스를 사용하는 쪽의 로직을 살짝 변경해서 유효성을 검사하게 만들 수 있습니다. 하지만 사용하는 곳마다 검사 로직

6 (옮긴이) 안티 패턴이란 '하지 않는 것이 좋은 패턴'을 의미합니다.

미리보기용 PDF 입니다. 구매: https://ebook.insightbook.co.kr/