

객체 지향 리엔지니어링 패턴

세르게이 디마이어

스테판 뒤카스

오스카 니어스트라스

이승범 옮김

2024-11-09 버전

객체 지향 리엔지니어링 패턴 추천사

리팩터링하는 “방법”은 이미 여러 문헌에서 잘 다루고 있다. 하지만 “언제”와 “왜”는 경험을 통해서만 배울 수 있다. 이 책은 시스템 재설계를 언제 시작해야 하는지, 언제 중단해야 하는지, 그리고 리팩터링을 통해 어떤 효과를 기대할 수 있는지를 배우는 데 도움이 될 것이다.

— 켄트 벡, *Three Rivers Institute* 디렉터

이 책은 실용적이고 실무적인 리엔지니어링 지식과 전문성을 이해하기 쉽고 사용하기 쉬운 형태로 제시하고 있다. 따라서 이 책의 패턴은 리엔지니어링(reengineering)을 사용하는 데 관심이 있는 모든 사람이 자신의 작업에 적용하는 데 도움이 된다. 이 책이 진작에 내 서재에 있었더라면 바램이 있다.

— 프랭크 부시만, 지멘스 시니어 수석 엔지니어

이 책은 제목이 이야기하는 것 그 이상이다. 효과적인 리엔지니어링이란 다른 사람의 코드를 의도적이고 효율적으로 읽어 예측 가능한 변화를 만들어내는 것이다. 저자가 숙련된 리엔지니어링 행동의 패턴으로 강조하는 것과 동일한 프로세스는 읽기 쉽고 유지 관리가 가능한 소프트웨어 시스템을 만드는 데 필요한 기술로 쉽게 적용할 수 있다.

— 아델 골드버그, *Neometron, Inc.*

만약 어떤 사람이 내 사무실로 많은 문서와 CD 두 장이 들어 있는 커다란 상자(회사에서 리엔지니어링하고자 하는 소프트웨어의 설치 디스크와 자료)를 가져온다면, 이 책의 저자를 제 곁에 두게 되어 기쁠 것 같다. 그렇지 않다면 저자들의 책을 갖는 것이 차선책이다. 이 책에는 거창한 내용도, 과대 광고도, 쉽다는 약속도 없습니다. 그저 프로젝트를 해결하는 데 유용한 지침이 담긴 현실적이고 읽기 쉬우며 매우 유용한 책입니다. 그 고객이 사무실에 도착하기 전에 이 책을 구입하여 살펴보자! 여러분과 여러분의 회사가 많은 고민을 덜 수 있을 것 입다.

— 린다 라이징, 인디펜던트 컨설턴트

This book is available as a free download from <http://scg.unibe.ch/oorp/>.

Copyright © 2003 by Elsevier Science (USA).

Copyright © 2008, 2009 by Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz.

이 책의 콘텐츠는 크리에이티브 커먼즈 저작자표시-동일조건변경허락 3.0 Unported 라이선스에 따라 보호된다. 이용자는 다음 권리를 갖는다.

공유 — 복제, 배포, 및 공중송신으로 이용이 가능하다.

리믹스 — 변형하여 이용이 가능하다.

다음과 같은 조건을 따라야 한다

저작자표시. 적절한 출처 와, 해당 라이센스 링크를 표시하고, 변경이 있는 경우 공지 해야 한다.

(합리적인 방식으로 이렇게 하면 되지만, 이용 허락권자가 귀하에게 권리를 부여한다거나 귀하의 사용을 허가한다는 내용을 나타내서는 안 된다.)

동일조건변경허락. 이 저작물을 리믹스, 변형하거나 2차적 저작물을 작성하고 그 결과물을 공유할 경우에는 원 저작물과 동일한 조건의 라이선스를 적용하여야 한다.

- 재사용하거나 배포하려면 다른 사람에게 이 저작물의 라이선스 조건을 명확히 알려야 한다. 이를 위한 가장 좋은 방법은 이 웹 페이지로 연결되는 링크를 제공하는 것이다. <https://creativecommons.org/licenses/by-sa/3.0/deed.ko>
- 저작권 소유자의 허가를 받으면 위의 조건 중 어느 것이든 면제될 수 있다.
- 이 라이선스의 어떤 내용도 저작자의 저작인격권을 손상하거나 제한하지 않는다.



공정 거래 및 기타 권리의 위의 내용에 영향을 받지 않는다. 다음은 사람이 읽을 수 있는 법률 규정 요약본이다(전체 라이선스).

<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

Published by Square Bracket Associates, Switzerland. SquareBracketAssociates.org

ISBN 978-3-9523341-2-6

First Open-Source Edition, June, 2008. Revised, September, 2009.

엔, 사라, 그리고 닐스에게.
플로렌스, 쿠엔틴, 티보, 그리고 카미유에게.
안젤라 그리고 프라다에게.

서문

마틴 파울리 서문

오랫동안 소프트웨어 개발 프로세스에 관한 대부분의 책에서 아무 것도 없는 소프트웨어 개발을 위해 편집기 화면의 빈 시트에서 시작할 때 무엇을 해야 하는지에 대해 이야기하는 것은 매우 의아했다. 그 상황이 사람들이 코드를 작성하는 가장 일반적인 경우가 아니기 때문이다. 대부분의 사람들은 자신이 만든 것일지도 모른다. 기존 코드 베이스를 변경해야 한다. 이상적으로는 이 코드 베이스가 잘 설계되고 여러 사항을 잘 고려한 것이겠지만, 우리 모두는 얼마나 이상적인 세계가 어려운지 알고 있다.

따라서 이 책은 불완전하지만 가치 있는 코드 베이스로 무엇을 할 수 있는지에 대한 관점에서 쓰여졌기 때문에 의미가 있다. 또한 학계의 관점과 산업계의 관점이 효과적으로 혼합되어 있다는 점도 마음에 든다. 저는 초창기 베른의 짤 쌀한 초겨울에 파무스 그룹을 방문한 적이 있다. 현장과 연구실을 오가며 실제 프로젝트에 아이디어를 시도하고 연구실로 돌아와 반성하는 방식이 마음에 들었다.

이 책에는 그 경험이 고스란히 담겨 있다. 이 책은 어려운 코드 기반을 다루기 위한 빌딩 블록을 제공하고 리팩터링과 같은 기술을 언제 사용해야 하는지 설명한다. 리엔지니어링(reengineering)이 여전히 혼란 일인 상황에서 이런 종류의 책이 너무 적다는 것은 안타까운 사실이다. 하지만 적어도 이런 맥락의 책이 많지 않은 상황에서 이 책이 얼마나 좋은 책인지 보여주는 예라고 생각하니 다행이다.

마틴 파울리, *Thought Works, Inc.*

랄프 존슨 서문

좋은 패턴의 징후 중 하나는 그를 읽은 전문가가 '당연히 누구나 알고 있겠지'라고 말하지만 초보자는 '흥미롭긴 한데 잘 될까'라고 말할 가능성이 높다는 것이다. 패턴은 따라하기 쉬워야 하지만 가장 가치 있는 패턴은 뻔하지 않아야 한다. 전문가들은 경험을 통해 패턴이 효과가 있다는 것을 알게 되었지만 초보자는 패턴을 사용하기 전까지는 패턴을 믿고 자신의 경험을 발전시켜야 한다.

지난 몇 년 동안 나는 이 책의 패턴을 다양한 사람들에게 알려주고 토론할 기회를 가졌다. 나의 패턴 토론 그룹에는 수십 년의 컨설팅 경력을 가진 회원들이 몇 명 있는데, 그들은 이 패턴을 사용한 이야기를 금세 그룹에 들려주었다. 젊은 멤버들은 패턴의 가치를 확신하고 있었기 때문에 그 이야기를 좋아했다.

나는 소프트웨어 엔지니어링 수업에서 학생들에게 리엔지니어링에 관한 섹션의 일부로 패턴 중 일부를 읽게 했다. 패턴에 흥미를 보인 학생은 한 명도 없었지만 수업은 순조롭게 진행되었다. 학생들은 패턴을 평가할 경험이 없었기 때문이다. 그런데 한 학생이 여름방학을 마치고 저에게 돌아와서 이 강의의 모든 내용 중에서 리버스 엔지니어링에 관한 패턴이 가장 유용했다고 말했다. 처음에는 패턴이 믿을 만하다 생각했고, 나중에는 그 내용을 믿을 수 있었다.

소프트웨어 리엔지니어링에 대한 경험이 많다면 이 책에서 많은 것을 배우지는 못할 것이다. 하지만 함께 일하는 사람들에게 책을 선물하고 싶고, 그들과 대화할 때 이 책의 어휘를 사용하고 싶을 것이기 때문에 어쨌든 읽어봐야 한다. 리엔지니어링을 처음 접한다면 이 책을 읽고 패턴을 익힌 다음 시도해 보라. 가치 있는 많은 것을 배울 수 있을 것이다. 패턴은 실용적이고 실용적인 지식은 경험해야 완전히 이해할 수 있으므로 시도하기 전에 패턴을 완전히 이해하리라고 기대하지 마라. 그럼에도 불구하고 이 책은 큰 도움이 될 것이다. 적용할 것이 있으면 훨씬 쉽게 배울 수 있으며, 이 책은 그를 위한 신뢰할 수 있는 가이드를 제공한다.

랄프 존슨, 일리노이 대학 어바나 샘페인 캠퍼스

차 례

서문	vii
서문	xv
I 개요	1
1 리엔지니어링 패턴	3
1.1 왜 리엔지니어링인가?	3
1.2 리엔지니어링 수명 주기	9
1.3 리엔지니어링 패턴	14
1.4 리엔지니어링 패턴의 형식	16
1.5 리엔지니어링 패턴 지도	16
II 리버스 엔지니어링	21
2 방향 설정	23
2.1 맥심에 동의하기	27
2.2 내비게이터 지정하기	29
2.3 원탁 회의에 말하기	31

2.4	가장 가치 있는 것 먼저 하기	33
2.5	증상이 아닌 문제 수정하기	37
2.6	고장이 나지 않으면 수정하지 않기	39
2.7	간단하게 유지 하기	41
3	첫 번째 접근	43
3.1	유지보수자와 담소나누기	49
3.2	한 시간 안에 모든 코드 읽기	59
3.3	문서 스키밍하기	67
3.4	데모 중 인터뷰하기	75
3.5	모의 설치 수행하기	85
4	초기 이해	91
4.1	퍼시스턴트 데이터 분석하기	97
4.2	디자인 추측하기	109
4.3	예외적인 엔티티 연구하기	119
5	디테일한 모델 캡처	129
5.1	코드에 대한 질문을 연결하기	135
5.2	이해하기 위해 리팩터링하기	141
5.3	단계 별 실행해 보기	147
5.4	컨트랙트 찾기	151
5.5	과거로부터 배우기	157
III	리엔지니어링	163
6	테스트라는 생명보험	165
6.1	진화 활성화를 위한 테스트 작성하기	169

6.2	기본 테스트를 증가시키기	175
6.3	테스트 프레임워크 사용하기	179
6.4	구현이 아닌 인터페이스 테스트하기	187
6.5	비즈니스 규칙을 테스트로 기록하기	193
6.6	이해하기 위해 테스트 작성하기	197
7	마이그레이션 전략	201
7.1	사용자 참여시키기	205
7.2	자신감 구축하기	209
7.3	시스템 점진적 마이그레이션하기	213
7.4	목표 솔루션 프로토타입하기[n]	217
7.5	실행 버전 항상 보유하기	221
7.6	변경할 때마다 회귀 테스트하기	223
7.7	뉴 타운으로 가는 브리지 만들기	225
7.8	올바른 인터페이스 제시하기	229
7.9	퍼블릭 인터페이스와 게시된 인터페이스 구분하기	233
7.10	폐기된 인터페이스 지원 중단하기	237
7.11	친숙도 보존하기	241
7.12	최적화하기 전에 프로파일러 사용하기	243
8	중복 코드 감지	245
8.1	기계적으로 코드 비교하기	249
8.2	도트 플롯으로 코드 시각화하기	255
9	책임 재배포	263
9.1	동작을 데이터 가까이 이동하기	269
9.2	탐색 코드 제거하기	279

9.3 신 클래스 분할하기	289
10 다형성 적용한 조건문 변환	297
10.1 자신 타입 체크 변환하기	301
10.2 클라이언트 타입 검사 변환하기	311
10.3 상태 추출하기	321
10.4 전략 추출하기	325
10.5 널 객체 도입하기	329
10.6 조건문을 등록으로 변환하기	333
IV 부록	343
A 섬네일 패턴	345
A.1 테스팅 패턴	346
A.1.1 지속적인 문제 재테스트하기	346
A.1.2 퍼지 기능 테스트하기	346
A.1.3 오래된 버그 테스트하기	347
A.2 리펙터링	347
A.2.1 필드 캡슐화하기	347
A.2.2 메서드 추출하기	347
A.2.3 메서드 이동하기	348
A.2.4 속성 이름 바꾸기	348
A.2.5 메소드 이름 바꾸기	348
A.2.6 조건문을 다형성으로 치환하기	349
A.3 디자인 패턴	349
A.3.1 추상 팩토리	349
A.3.2 어댑터	349

A.3.3	파사드	350
A.3.4	팩토리 메서드	350
A.3.5	경량	350
A.3.6	널 객체	350
A.3.7	수량	351
A.3.8	싱글턴	351
A.3.9	상태	351
A.3.10	상태 패턴	352
A.3.11	전략	352
A.3.12	템플리트 메서드	352
A.3.13	비지터	353

서문

동화

옛날 옛적에 고객이 원하는 것이 무엇인지 정확히 알고 있는 착한 소프트웨어 엔지니어가 있었다. 착한 소프트웨어 엔지니어는 현재 그리고 이 후 수십 년 동안 고객의 모든 문제를 해결할 수 있는 완벽한 시스템을 설계하기 위해 열심히 노력했다. 완벽한 시스템이 설계, 구현되고 마침내 배포되었을 때 고객은 정말 매우 만족했다. 시스템 유지 관리자는 완벽한 시스템을 계속 가동하기 위해 할 일이 거의 없었고, 고객과 유지 관리자는 그 후에도 행복하게 살았다.

현실은 왜 이 동화와 같지 않을까?

좋은 소프트웨어 엔지니어가 없기 때문일까? 사용자가 자신이 원하는 것이 무엇인지 잘 모르기 때문일까? 아니면 완벽한 시스템이 존재하지 않기 때문일까?

이 모든 항목은 어느 정도 진실이 포함되어 있을 수 있지만, 진짜 이유는 아마도 몇 년 전 매니 리먼과 레스 벨레디가 발견한 소프트웨어 진화의 기본 법칙과 더 관련이 있을 것이다. 이 법칙 중 가장 눈에 띄는 두 가지는 다음과 같다[LB85].

- 지속적 변화의 법칙(*The Law of Continuing Change*) — 실제 환경에서 사용되는 프로그램은 반드시 변화하거나 그 환경에서 점차적으로 유용성이

떨어진다.

- 복잡성 증가의 법칙(*The Law of Increasing Complexity*) — 프로그램이 진화함에 따라 더 복잡해지고, 구조를 보존하고 단순화하기 위해 더 많은 리소스가 필요하다.

다시 말해, 모든 요구 사항을 파악하고 완벽한 시스템을 구축할 수 있다고 생각한다면 이는 스스로에게 거짓말을 하는 것이다. 우리가 할 수 있는 최선은 새로운 작업 요청받을 때까지 충분히 오래 살아남을 수 있는 유용한 시스템을 구축하는 것이다.

책의 내용

이 책은 소프트웨어 엔지니어링의 가장 흥미롭고 도전적인 측면은 새로운 소프트웨어 시스템을 구축하는 것이 아니라 기존 시스템을 재탄생시키는 것일 수 있다는 깨달음의 결과이다.

1996년 11월부터 1999년 12월까지 저자들은 FAMOOS (ESPRIT Project 21975—객체지향 소프트웨어 진화를 위한 프레임워크 기반 접근법(*Framework-based Approach for Mastering Object-Oriented Software Evolution*))라는 유럽 산업 연구 프로젝트에 참여했다. 파트너로는 노키아(핀란드), 다임러 벤츠(독일), 세마 그룹(스페인), 포승스젠티룸 인포매틱 카를스루에(독일), 베른 대학교(스위스)가 참여했다. 노키아와 다임러 벤츠는 모두 객체 지향 기술의 얼리 어답터였으며, 이 전략을 통해 상당한 이점을 얻을 수 있을 것으로 기대했다. 그러나 이제 그들은 변화하는 요구사항에 적응하기 매우 어려운 매우 크고 가치 있는 객체 지향 소프트웨어 시스템을 보유하고 있었기 때문에 기존 시스템의 전형적인 문제를 많이 경험하고 있었다. FAMOOS 프로젝트의 목표는 이러한 객체 지향 레거시 시스템을 계속 유용하게 사용하고 향후 요구 사항의 변화에 더 잘 적응할 수 있도록 재탄생시키는 도구와 기술을 개발하는 것이었다.

프로젝트 초기에는 이러한 대규모 객체 지향 애플리케이션을 프레임워크, 다시 말해 다양한 프로그래밍 기법을 사용하여 쉽게 재구성할 수 있는 일반적인 여러 애플리케이션으로 전환하는 것이 목표였다. 그러나 이것이 말처럼 쉽지 않다는 것을 금방 알게 되었다. 기본 아이디어는 좋았지만 레거시 시스템에서

어떤 부분을 변환해야 하는지, 정확히 어떻게 변환해야 하는지 결정하는 것은 쉽지 않았다. 사실 레거시 시스템에 무엇이 문제인지 파악하는 것은 말할 것도 없고, 애초에 레거시 시스템을 이해하는 것만으로도 사소한 문제가 아니다.

저자들은 이 프로젝트를 통해 많은 것을 배웠다. 대부분의 경우 레거시 코드가 전혀 나쁘지 않다는 것을 알게 되었다. 레거시 코드에 문제가 있었던 유일한 이유는 원래 시스템이 설계되고 배포된 이후 요구사항이 변경되었기 때문이었다. 변화하는 요구사항에 따라 여러 번 조정된 시스템은 원래의 아키텍처와 설계를 인식할 수 없는 디자인 드리프트(*design drift*)를 겪게 된다. 이로 인해 리먼과 벨라디의 소프트웨어 진화 법칙에서 예측한 대로 추가적인 조정이 거의 불가능하게 되는 상황이 발생한다.

그러나 가장 놀라운 사실은 우리가 살펴본 사례 연구 각각이 언번들링(unbundling)¹, 요구 사항 확장, 새로운 환경으로의 포팅 등 매우 다른 이유로 리엔지니어링(reengineering)이 필요했지만, 이러한 시스템의 실제 기술적 문제는 이상할 정도로 유사하다는 사실이었다.

운이 좋아 문서가 있다고 해도 이를 신뢰할 수 없기 때문에 모든 리엔지니어링(reengineering) 활동은 리버스 엔지니어링(reverse engineering)으로 시작해야 한다는 사실을 발견했다. 기본적으로 소스 코드를 분석하고, 시스템을 실행하고, 사용자와 개발자를 인터뷰하여 레거시 시스템의 모델을 구축할 수 있다. 그런 다음 더 발전하기 위한 장애물이 무엇인지 파악하고 이를 해결해야 한다. 이것이 바로 레거시 시스템을 현재 알고 있는 모든 새로운 요구 사항을 미리 알 수 있었다면 구축했을 시스템으로 전환하는 리엔지니어링(reengineering)의 본질이다. 하지만 모든 것을 다시 구축할 여유가 없으므로 비용을 절감하고 가장 중요한 부분만 리엔지니어링해야 한다.

indFAMOOS 프로젝트 이후, 저희는 다른 많은 리엔지니어링 프로젝트에 참여했고, FAMOOS의 결과를 더욱 검증하고 개선할 수 있었다.

이 책에서 우리는 객체 지향 시스템을 리엔지니어링해야 하는 다른 사람들에게 도움이 되기를 바라며 우리가 배운 것을 정리했다. 모든 문제에 대한 해답을 제공하지는 못했지만, 먼 길을 가는 데 도움이 될 여러 간단한 기술을 확인했다.

¹ 그룹으로 되어 있던 제품 및 서비스 패키지를 분리하는 프로세스 -옮긴이

패턴의 중요성

패턴은 반복되는 모티프, 즉 반복해서 발생하는 이벤트 또는 구조를 말한다. 디자인 패턴(*Design patterns*)은 반복되는 디자인 문제에 대한 일반적인 해결책이다[GHJV95]. 이러한 디자인 문제는 완전히 똑같지는 않지만 매우 유사하기 때문에 그 해결책은 소프트웨어가 아니라 베스트 프랙티스를 전달하기 위한 문서화된 것(*documents that communicate best practice*)이다.

패턴은 최근 몇 년 동안 다양한 종류의 문제를 해결하는 베스트 프랙티스를 문서화하는 데 사용할 수 있는 형식으로 등장했다. 많은 종류의 문제와 해결책을 패턴으로 표현할 수 있지만, 가장 단순한 종류의 문제에 적용하는 것은 적절하지 않다. 문서화 형식으로서의 패턴은 고려 중인 문제가 여러 가지 상충하는 주요한 요구사항(*forces*)을 다루고 설명하는 해결책이 여러 가지 트레이드오프(*tradeoffs*)를 고려할 때 가장 유용하고 흥미롭다. 예를 들어, 잘 알려진 많은 디자인 패턴은 디자인 복잡성(design complexity)이 증가하는 대신 런타임 유연성(run-time flexibility)이 좋아진다.

이 책에서는 레거시 시스템을 리버스 엔지니어링(reverse engineering)하고 리엔지니어링(reengineering)하기 위한 패턴 카탈로그를 제공한다. 이러한 패턴 중 어떤 것도 맹목적으로 적용해서는 안 된다. 각 패턴은 몇 가지 주요한 요구사항(*forces*)을 해결하고 몇 가지 트레이드오프(*tradeoffs*)를 감수한다. 패턴을 성공적으로 적용하려면 이러한 트레이드오프를 이해하는 것이 필수적이다. 결과적으로 패턴 양식은 리엔지니어링 프로젝트 과정에서 확인한 모범 사례를 문서화하는 가장 자연스러운 방법인 것 같다.

패턴 언어(*pattern language*)는 일련의 복잡한 문제를 해결하기 위해 조합하여 사용할 수 있는 관련 패턴의 집합이다. 우리는 패턴의 클러스터가 서로 잘 조합되어 작동하는 것처럼 보였기 때문에 작은 패턴 언어으로서 클러스터를 하나의 장으로 제시하도록 이 책을 구성했다.

우리는 이러한 클러스터가 어떤 의미에서든 “완전”하다고 생각하지 않으며, 리엔지니어링의 모든 측면을 포괄하는 패턴을 가지고 있는 것처럼 가장하지도 않는다. 이 책이 객체 지향 리엔지니어링을 위한 체계적인 방법을 제시한다고 주장하지도 않는다. 다만 흥미로운 시너지를 발휘하는 여러 모범 사례를 접하고 확인했다고 주장할 뿐이다. 패턴 클러스터 내에는 강력한 시너지가 있을 뿐만 아니라, 패턴 클러스터는 중요한 방식으로 상호 연관되어 있다. 따라서

각 장에는 패턴이 “언어”로서 어떻게 기능할 수 있는지 제안하는 패턴 맵이 포함되어 있을 뿐만 아니라 각 패턴이 동일한 클러스터 또는 다른 클러스터에서 다른 패턴과 어떻게 결합되거나 구성될 수 있는지도 나열하고 설명한다.

대상 독자

이 책은 주로 객체 지향 시스템을 리엔지니어링해야 하는 실무자를 대상으로 한다. 극단적인 관점을 취한다면 모든 소프트웨어 프로젝트가 리엔지니어링 프로젝트라고 할 수 있으므로 이 책의 범위는 상당히 넓다.

이 책에 나오는 대부분의 패턴은 객체 지향 소프트웨어 개발 경험이 조금이라도 있는 사람이라면 누구나 쉽게 읽을 수 있을 것이다. 이 책은 세부 사항을 문서화하는 것을 목표로 삼았다.

감사 인사

이 패턴을 발견할 수 있는 배경을 제공한 노키아, 다임러 벤츠, 포승스센트룸 인포메틱 카를스루에, 세마 그룹 와 같은 FAMOOS 프로젝트 파트너들에게 무엇보다 먼저 감사를 전한다. 프로젝트를 시작하는 동안 Juha (Julho) Tuominen, Roland Trauter, Eduardo Casais, Theo Dirk Meijler와 같은 사람들이 중요한 역할을 했다. 특히 이 책의 프로토타입 공동 저자인 다음과 같은 분들께 감사를 전한다. Holger Bär, Markus Bauer, Oliver Ciupke, Michele Lanza, Radu Marinescu, Robb Nebbe, Michael Przybilski, Tamar Richner, Matthias Rieger, Claudio Riva, Anne-Marie Sassen, Benedikt Schulz, Patrick Steyaert, Sander Tichelaar 및 Joachim Weisbrod이다.

우리는 ESPRIT 프로젝트 21975(FAMOOS)에 대한 유럽 연합의 재정 지원과 NFS-2000-46947.96 및 BBW-96.0015 프로젝트에 대한 스위스 정부의 재정 지원에 감사드린다. 앤트워프 대학교는 “객체 지향 리엔지니어링”이라는 제목의 보조금으로 재정 지원을 제공했고, 플랑드르 과학 연구 기금은 “소프트웨어 진화의 기초”라는 연구 네트워크를 통해 후원했다.

이 책의 일부 자료는 1998년과 1999년 겨울 학기에 베른 대학교에서 열린

대학원 과정 “객체 지향 소프트웨어 리엔지니어링”과 OOPSLA에서 열린 여러 튜토리얼에서 발표되었다. 이 강좌와 튜토리얼의 참가자들이 보내준 피드백과 의견에 감사드린다. 또한 여러 패턴 워크숍에 참여하여 이 책의 많은 패턴에 대해 귀중한 피드백을 주신 베른 대학교의 소프트웨어 컴포지션 그룹 회원들에게도 감사를 전한다. Michele Lanza, Pietro Malorgio, Robbe Nebbe, Tamar Richner, Matthias Rieger, Sander Tichelaar이 그 회원들이다.

이 책에 소개된 패턴 중 일부는 다른 곳에 소개된 바 있다. indEuroPLoP의 쉘퍼드(shepherd)인 Kent Beck(1998), Kyle Brown(1999), Neil Harrison(2000), Mary Lynn Manns(2000), Don Roberts(1998), Charles Weir(1998)와 이러한 패턴을 논의했던 저자 워크숍(Writer's workshop)의 모든 참가자들에게 감사를 전하고 싶다. 패턴의 형태와 힘에 대해 도움을 주신 Jens Coldewey에게 특별히 감사드린다.

이 책의 여러 챕터에 대해 워크샵을 진행한 Ralph Johnson의 소프트웨어 아키텍처 그룹 멤버와 친구들에게도 감사를 전한다. John Brant, Brian Foote, Alejandra Garrido, Peter Hatch, Ralph Johnson, Brian Marick, Andrew Rosenfeld, Weerasak Witthawaskul 그리고 Joe Yoder이다. 방대한 크기의 워크샵 녹음을 MP3 형식으로 다운로드하고 재생하면서 우리 모두는 정말 “벽 위의 파리”²가 되어 자신을 바라보는 기분이었다.

저희의 프로젝트를 혼신적으로 도와준 Morgan Kaufmann의 편집자 Tim Cox와 그의 조수 Stacie Pierce에게 감사를 전한다. 또한 처음에 Tim과 연락을 취하게 해준 DPunkt Verlag의 Christa Preisendanz에게도 감사드린다. 특히 두 차례에 걸친 철저한 검토에 감사드리며, 이 책의 최종 초안이 일부 검토자가 기대했던 최종 결과물과는 전혀 달라서 아쉬울 뿐입니다! 행간을 읽어주시고 많은 패턴을 설명해 주신 리뷰어 여러분께 감사드린다. Kyle Brown, Thierry Cattel, Oliver Ciupke, Koen De Hondt, Jim Coplien, Gert Florijn, Neil Harrison, Mary Lynn Manns, Alan O'Callaghan, Don Roberts 그리고 Benedikt Schulz가 그들이다.

²제 3자의 입장에서 담백하게 내용을 담는 다큐멘터리 제작 스타일 - 옮긴이

제 I 편

개요

제 1 장

리엔지니어링 패턴

1.1 왜 리엔지니어링인가?

유산(레거시, legacy)이란 물려받은 것 중에 가치가 있는 것이다. 마찬가지로, 레거시 소프트웨어는 여러분이 물려받은 가치 있는 소프트웨어이다. 상속받았다는 것은 다소 구식이라는 것을 의미할 수 있다. 오래된 프로그래밍 언어나 구식 개발 방법을 사용하여 개발되었을 수 있다. 개발자가 여러 번 바뀌었고 많은 수정과 변형의 흔적이 남아 있을 가능성이 높다.

어쩌면 레거시 소프트웨어가 그렇게 오래되지 않았을 수도 있다. 빠른 개발 도구와 인력의 빠른 이직으로 인해 소프트웨어 시스템은 상상 이상으로 빠르게 레거시가 될 수 있다. 하지만 소프트웨어의 가치가 높다는 사실은 이를 그냥 버릴 수 없다는 것을 의미한다.

레거시 소프트웨어는 비즈니스에 매우 중요하며, 이것이 바로 모든 문제의 근원이다. 비즈니스에서 성공하려면 변화하는 비즈니스 환경에 적응할 수 있도록 끊임없이 준비해야 한다. 따라서 비즈니스를 운영하는 데 사용하는 소프트웨어도 적응력이 있어야 한다. 다행히도 많은 소프트웨어는 업그레이드할 수 있고, 더 이상 쓸모가 없어지면 간단히 폐기하고 교체할 수도 있다. 하지만 레거시 시스템은 많은 비용이 들지 않는 한 교체하거나 업그레이드할 수 없다. 리엔지니어링의 목표는 레거시 시스템의 복잡성을 충분히 줄여 적절한 비용으

로 계속 사용하고 적응할 수 있도록 하는 것이다.

소프트웨어 시스템을 리엔지니어링해야 하는 구체적인 이유는 매우 다양할 수 있다. 예를 들어 다음과 같다.

- 개별 파트를 더 쉽게 개별적으로 판매하거나 다른 방식으로 결합할 수 있도록 모놀리식 시스템(monolithic system)을 언번들링(unbundle)¹ 하려고 할 수 있다.
- 성능(performance)을 개선하고 싶을 수 있다. (경험상 올바른 순서는 “먼저 동작하게 하고, 제대로 동작하게 하고, 빠르게 동작하게 하는 것(first do it, then do it right, then do it fast)”이므로 성능에 대해 생각하기 전에 코드를 정리하는 리엔지니어링을 하는 것이 좋다.)
- 시스템을 새 플랫폼으로 포팅(*port the system to a new platform*)하고 싶을 수도 있다. 그 전에 플랫폼에 종속된 코드를 명확하게 분리하기 위해 아키텍처를 변경해야 할 수도 있다.
- 새롭게 구현하는 첫 번째 단계로 디자인 추출(*extract the design*)을 수행할 수 있다.
- 유지보수 비용을 절감하기 위한 단계로 새로운 표준이나 라이브러리와 같은 신기술을 활용(*exploit new technology*)하고 싶을 수도 있다.
- 시스템에 대한 지식을 문서화하여 유지보수를 더 쉽게 함으로써 인적 종속성을 줄이려고(*reduce human dependencies*) 할 수도 있다.

시스템을 리엔지니어링하는 데에는 여러 가지 이유가 있을 수 있다. 앞으로 살펴보겠지만 레거시 소프트웨어의 실제 기술적 문제는 매우 유사한 경우가 많다. 이러한 사실 덕분에 리엔지니어링 작업의 일부분이라도 매우 일반적인 기술을 사용할 수 있다.

리엔지니어링 필요성 인식

레거시 문제가 있는지 어떻게 알 수 있을까?

¹ 그룹으로 되어 있던 제품 및 서비스 패키지를 분리하는 프로세스 -옮긴이

“고장 나지 않았으면 고치지 말자(If it ain’t broke, don’t fix it.)”는 것이 일반적인 통념이다. 이러한 태도는 종종 중요한 기능을 수행하고 있고 잘 작동하는 것처럼 보이는 소프트웨어에 손을 대지 않으려는 핑계로 받아들여진다. 이 접근 방식의 문제점은 무언가가 “고장”날 수 있는 방법이 다양하다는 점을 인식하지 못한다는 것이다. 기능적인 관점은, 무언가는 설계된 바대로 기능을 더 이상 수행하지 못하는 경우에만 고장난 것이다. 그러나 유지 관리 관점에서 보면 소프트웨어는 더 이상 유지 관리할 수 없는 경우 고장난 것이다.

그렇다면 소프트웨어가 곧 고장날 것이라는 것을 어떻게 알 수 있을까? 다행히도 문제가 발생하고 있다는 것을 알려주는 경고 신호가 많이 있다. 아래 나열된 증상은 일반적으로 단독으로 발생하는 것이 아니라 한 번에 여러 개씩 나타난다.

문서가 없거나 더 이상 사용되지 않음(Obsolete or no documentation).

사용되지 않는 문서는 많은 변화를 겪은 레거시 시스템의 분명한 신호이다. 문서가 없다는 것은 원래 개발자가 프로젝트를 떠나자마자 문제가 발생할 수 있다는 경고 신호이다.

테스트 누락(Missing tests). 최신 문서보다 더 중요한 것은 모든 시스템 구성 요소에 대한 철저한 단위 테스트와 모든 중요한 사용 사례 및 시나리오를 포괄하는 시스템 테스트의 존재 여부이다. 이러한 테스트가 없다는 것은 시스템이 높은 위험이나 비용 없이 발전할 수 없다는 신호이다.

기존 개발자 또는 사용자의 이동(Original developers or users have left).

테스트 커버리지가 좋은 깨끗하고 잘 문서화된 시스템이 아니라면, 그보다 훨씬 덜 깨끗하고 문서화가 잘 안 된 시스템으로 급속히 악화될 것이다.

사라진 시스템의 내부 지식(Inside knowledge about system has disappeared).

이것은 나쁜 신호이다. 문서가 기존 코드 베이스와 동기화되지 않는다. 아무도 실제로 어떻게 작동하는지 모른다.

제한적 전체 시스템 이해(Limited understanding of the entire system).

아무도 상세 내용을 이해하지 못할 뿐만 아니라 전체 시스템에 대해 제대로 파악하고 있는 사람도 거의 없다.

상용화 시간이 오래 걸림(Too long to turn things over to production).

프로세스의 어딘가에 문제가 있다. 변경 사항을 승인하는 데 너무 오래 걸리는 것일 수 있다. 자동 회귀 테스트가 누락되었을 수도 있다. 또는 변경 사항을 배포하기가 어려울 수도 있다. 이러한 어려움을 이해하고 대처하지 않으면 상황은 더욱 악화될 것이다.

간단한 변경에 너무 많은 시간이 소요(Too much time to make simple changes).

이것은 리먼과 벨레이디의 복잡성 증가의 법칙(Law of Increasing Complexity) 이 시작되었다는 분명한 신호이다. 이제 시스템이 너무 복잡해져서 간단한 변경조차 구현하기 어렵다는 뜻이다. 시스템을 간단하게 변경하는 데 너무 오랜 시간이 걸리면 복잡한 변경을 하는 것은 당연히 불가능할 것이다. 간단한 변경이 완료되기를 기다리는 백로그가 있으면 어려운 문제를 해결할 수 없다.

줄어들지 않는 버그(Need for constant bug fixes).

버그는 절대 사라지지 않는 것 같다. 버그를 수정할 때마다 그 옆에 새로운 버그가 나타난다. 이는 애플리케이션의 일부가 너무 복잡해져서 더 이상 작은 변경의 영향을 정확하게 평가할 수 없다는 것을 의미한다. 또한 애플리케이션의 아키텍처가 더 이상 요구 사항과 일치하지 않으므로 작은 변경도 예기치 않은 결과를 초래할 수 있다.

유지 관리 종속성(Maintenance Dependencies).

한 곳에서 버그를 수정하면 다른 곳에서 또 다른 버그가 다른 곳에 나타난다. 이는 종종 시스템의 논리적으로 분리된 구성 요소가 더 이상 독립적이지 않을 정도로 아키텍처가 악화되었다는 신호이다.

오래 걸리는 빌드 시간(Big build times).

재컴파일 시간이 길면 변경 작업이 느려진다. 빌드 시간이 길다는 것은 시스템 구성이 너무 복잡해서 컴파일러 툴이 효율적으로 작동하지 않는다는 의미일 수도 있다.

어려운 제품 분리(Difficulties separating products).

제품에 대한 고객이 많고 각 고객에 맞게 릴리스를 조정하는 데 어려움이 있다면 아키텍처가 더 이상 작업에 적합하지 않은 것이다.

중복된 코드(Duplicated code)

중복 코드는 시스템이 발전함에 따라 자연스럽게 발생하며, 거의 동일한 코드를 구현하거나 소프트웨어 시스템의 여

리 버전을 병합하는 지름길로 사용된다. 공통 부분을 적절한 추상화로 리팩토링하여 중복 코드를 제거하지 않으면 동일한 코드를 여러 곳에서 수정해야 하므로 유지 관리가 점점 어려워 질 수 있다.

코드 스멜(Code Smells). 중복된 코드는 “나쁜 냄새”가 나므로 변경해야 하는 코드의 예이다. 긴 메서드, 큰 클래스, 긴 매개변수 목록, 스위치 문, 데이터 클래스 등은 켄트 벤과 다른 사람들이 문서화한 몇 가지 예이다 [FBB⁺99]. 코드 스멜은 종종 시스템이 재설계되지 않은 채 반복적으로 확장되고 조정되었다는 신호이다.

객체의 특별한 점

이 책에서 설명하는 많은 기술이 모든 소프트웨어 시스템에 적용될 수 있지만, 저자들은 객체 지향 레거시 시스템에 초점을 맞추기로 했다. 이러한 선택에는 여러 가지 이유가 있지만, 무엇보다도 많은 객체 지향 기술 얼리 어답터들이 객체로 전환함으로써 기대했던 이점을 실현하기가 매우 어려웠다는 사실을 발견하고 있는 중요한 시점이라고 생각했기 때문이다.

이제 Java에도 레거시 시스템이 상당히 많이 존재한다. 소프트웨어가 레거시 시스템으로 변하는 것은 시간(*age*)가 아니라 재설계되지 않고 개발 및 적용되어 온 정도(*rate*)이다.

이러한 경험에서 도출할 수 있는 잘못된 결론은 “객체가 좋지 않아 다른 것이 필요하다”는 것이다. 이미 우리는 패턴, 컴포넌트, UML, XMI 등 많은 새로운 트렌드를 향해 달려가는 것을 목격하고 있다. 이러한 발전 중 모두 좋은 것인지만 어떤 의미에서는 모두 요점을 놓치고 있다.

이 책에서 얻을 수 있는 결론 중 하나는 객체는 꽤 좋지만 “객체를 잘 관리해야 한다”는 것이다. 이 점을 이해하려면 유연성(flexibility), 유지보수성(maintainability), 재사용성(reuse)이 그토록 좋은 객체 지향 시스템에서 왜 레거시 문제가 발생하는지 생각해 보자.

우선, 기존의 객체 지향 코드 베이스로 작업해 본 사람이라면 누구나 눈치챘을 것이다: 객체를 찾기가 어렵다라는 부분이다. 실제로 객체 지향 애플리케이션의 아키텍처는 일반적으로 숨겨져 있다. 눈에 보이는 것은 여러 개의 클래스와 상속 계층 구조뿐이다. 하지만 런타임에 어떤 객체가 존재하고 어떻게

협업하여 원하는 동작을 제공하는지는 알 수 없다. 객체 지향 시스템을 이해하는 것은 리버스 엔지니어링의 과정이며, 이 책에서 설명하는 기법은 이 문제를 해결하는 데 도움이 된다. 또한 코드를 리엔지니어링하면 아키텍처를 이해하기 쉬운 더 투명한 시스템에 도달할 수 있다.

둘째, 기존 객체 지향 애플리케이션을 확장해 본 사람이라면 누구나 깨달았을 것이다: 재사용은 공짜로 얻을 수 없다. 실제로 코드를 재사용할 수 있도록 설계하는 데 상당한 노력을 기울이지 않는 한 코드를 재사용하는 것은 매우 어렵다. 또한 재사용에 대한 투자는 올바른 조직 인프라를 구축하기 위한 경영진의 의지가 필수적이며, 명확하고 측정 가능한 목표를 염두에 두고 수행해야 한다 [GR95].

우리는 아직 재사용을 적절히 고려하는 방식으로 객체 지향 소프트웨어 프로젝트를 관리하는 데 능숙하지 않다. 일반적으로 재사용은 너무 늦게 이루어 진다. 우리는 객체 지향 모델링 기법을 사용하여 매우 풍부하고 복잡한 객체 모델을 개발하고, 소프트웨어를 구현할 때 무언가를 재사용할 수 있기를 바란다. 그러나 그때가 되면 엄청난 노력을 기울이지 않는 한 이러한 풍부한 모델이 어떤 종류의 표준 컴포넌트 라이브러리에 매핑될 가능성은 거의 없다. 우리가 소개하는 몇 가지 리엔지니어링 기법은 사후에 이러한 구성 요소를 발견하는 방법을 다룬다.

그러나 핵심 인사이트는 객체의 “올바른(right)” 설계와 구성은 초기 요구 사항만으로 알 수 있거나 알 수 있는 것이 아니라, 이러한 요구 사항이 어떻게 진화하는지를 이해한 결과라는 것이다. 세상이 끊임없이 변화하고 있다는 사실을 단순히 문제로만 볼 것이 아니라 해결의 열쇠로 삼아야 한다.

어떤 성공적인 소프트웨어 시스템도 레거시 시스템의 증상을 겪게 된다. 성공적인 객체 지향 레거시 시스템은 아키텍처와 설계가 더 이상 변화하는 요구 사항에 대응하지 못하는 성공적인 객체 지향 시스템일 뿐이다. 지속적인 리엔지니어링 문화는 유연하고 유지 관리가 가능한 객체 지향 시스템을 구축하기 위한 전제 조건이다.

1.2 리엔지니어링 수명 주기

리엔지니어링과 리버스 엔지니어링은 종종 같은 맥락에서 언급되며, 때때로 용어가 혼동되기도 하므로 그 의미를 명확히 하는 것이 좋다. 치코프스키와 크로스는 이 두 용어를 다음과 같이 정의한다.

“리버스 엔지니어링은 대상 시스템을 분석하여 시스템의 구성 요소와 상호 관계를 파악하고 시스템을 다른 형태 또는 더 높은 수준의 추상화로 표현하는 프로세스이다.”라고 정의한다.

즉, 리버스 엔지니어링은 본질적으로 시스템과 시스템이 어떻게 작동하는지 파악(*understand*)하는 것과 관련이 있다.

“리엔지니어링은 ... 새로운 형태로 재구성하고 그 결과로 후속 구현하기 위해 대상 시스템을 분석하고 대상 시스템을 변경(*alteration of a subject system*)하는 것을 말한다.”

반면에 리엔지니어링은 일반적으로 실제 또는 인지된 문제를 해결하기 위해, 더 구체적으로는 추가 개발 및 확장을 준비하기 위해 시스템을 재구조화(*restructuring*)하는 것과 관련이 있다.

“리버스 엔지니어링(reverse engineering)”이라는 용어의 도입은 분명히 “포워드 엔지니어링(forward engineering)”을 정의하기 위한 것이므로 다음과 같이 정의할 수 있다.

“포워드 엔지니어링(Forward Engineering)은 높은 수준의 추상화와 논리적, 구현 독립적인 설계에서 시스템의 물리적 구현으로 이동하는 전통적인 프로세스입니다.”

물론 이러한 포워드 엔지니어링 프로세스가 정확히 어떻게 작동할 수 있는지 또는 작동해야 하는지는 큰 논란의 여지가 있지만, 대부분의 사람들은 이 프로세스가 반복적이라는 사실을 인정하며, 소프트웨어 개발의 소위 나선형 모델(*spiral model*)[Boe88]에 부합한다는 점을 인정한다. 이 모델에서는 요구사항을 수집하고, 위험을 평가하고, 새 버전을 엔지니어링하고, 결과를 평가하는 과정을 반복하여 소프트웨어 시스템의 연속적인 버전을 개발한다. 이 일반적인

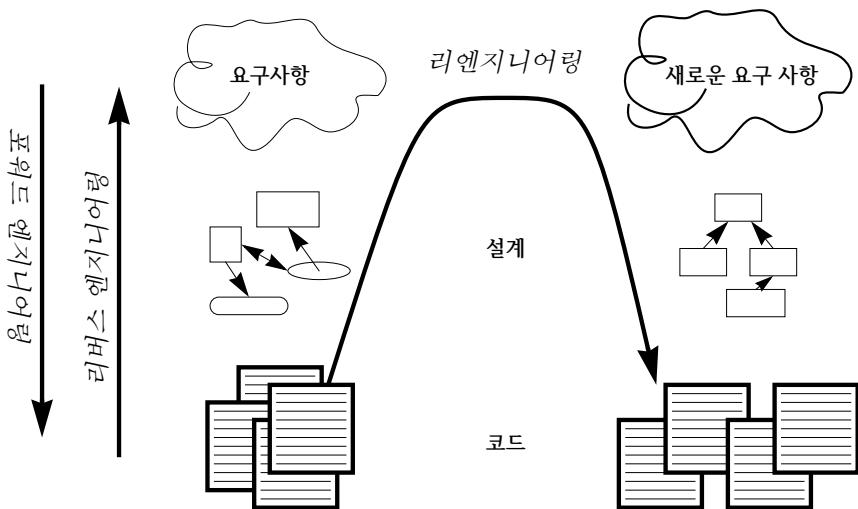


그림 1.1: 포워드 엔지니어링, 리버스 엔지니어링 그리고 리엔지니어링

프레임워크는 실제로 사용되는 다양한 종류의 보다 구체적인 프로세스 모델을 수용할 수 있다.

포워드 엔지니어링이 요구 사항과 모델에 대한 높은 수준의 관점에서 구체적인 구현을 향해 나아가는 것이라면, 리버스 엔지니어링은 구체적인 구현에서 보다 추상적인 모델로 거꾸로 나아가는 것이고, 리엔지니어링은 구체적인 구현을 다른 구체적인 구현으로 변환하는 것이다.

그림 1.1 은 이 개념을 설명한다. 포워드 엔지니어링(*forward engineering*)은 높은 수준의 추상적인 모델과 아티팩트에서 구체적인 모델과 산출물을 이동하는 과정으로 이해할 수 있다. 리버스 엔지니어링(*reverse engineering*)은 코드에서 상위 수준의 모델과 산출물을 재구성한다. 리엔지니어링(*reengineering*)은 하나의 낮은 수준의 표현을 다른 표현으로 변환하는 과정이며, 그 과정에서 상위 수준의 산출물을 다시 생성한다.

여기서 주목해야 할 핵심 사항은 리엔지니어링은 단순히 소스 코드를 변환하는 것이 아니라 시스템을 모든 수준에서 변환하는 것이라는 점이다. 그렇기 때문에 리버스 엔지니어링과 리엔지니어링을 같은 맥락에서 이야기하는 것이 합리적이다. 일반적인 레거시 시스템에서는 소스 코드뿐만 아니라 모든 문

서와 사양이 동기화되지 않은 것을 발견할 수 있다. 따라서 리버스 엔지니어링은 이해하지 못하는 것을 변형할 수 없기 때문에 리엔지니어링의 전제 조건(*prerequisite*)이다.

리버스 엔지니어링

무언가가 실제로 어떻게 작동하는지 이해하려고 할 때마다 리버스 엔지니어링(reverse engineering)을 수행한다. 일반적으로 소프트웨어를 수정, 확장 또는 교체하려는 경우에만 리버스 엔지니어링이 필요하다. 소프트웨어를 리버스 엔지니어링해야 하는 이유는 때때로 사용 방법을 이해하기 위해서일 수도 있다. 이는 리엔지니어링이 필요하다는 신호일 수도 있다. 따라서 리버스 엔지니어링 작업은 일반적으로 리엔지니어링에 대비해 소프트웨어를 재문서화(*redocumenting*)하고 잠재적인 문제를 식별하는 데 중점을 둔다.

리버스 엔지니어링을 하는 동안 다양한 정보 소스를 활용할 수 있다. 예를 들자면 다음과 같다.

- 기존 문서 읽기
- 소스 코드 읽기
- 소프트웨어 실행해 보기
- 사용자 및 개발자 인터뷰하기
- 테스트 케이스 코딩 및 테스트 해보기
- 트레이스 생성 및 분석하기
- 다양한 도구로 소스 코드 및 트레이스의 하이 레벨 뷰(*high-level views*) 생성하기
- 버전 히스토리 분석하기

이러한 활동을 수행하면서 점진적으로 개선된 소프트웨어 모델을 구축하고, 다양한 질문과 답변을 찾고, 기술 문서를 정리하게 된다. 또한 수정해야 할 문제에 집중하게 될 것이다.

리엔지니어링

시스템을 리엔지니어링하는 이유는 다양할 수 있지만, 실제 기술적인 문제는 일반적으로 매우 유사하다. 일반적으로 거시적인 아키텍처 문제와 세분화된 설계 문제가 혼합되어 있다. 대략적으로 문제를 분류해 보면 다음과 같다.

- 문서 부족(*Insufficient documentation*): 문서가 존재하지 않거나 현실과 불일치한다.
- 부적절한 레이어링(*Improper layering*): 누락되거나 부적절한 레이어링은 이식성(portability)과 적응성(adaptability)을 방해한다.
- 모듈화 부족(*Lack of modularity*): 모듈 간의 높은 결합도(coupling)은 진화(evolution)를 방해한다.
- 코드 중복(*Duplicated code*): “복사, 붙여넣기 및 편집(copy, paste and edit)”은 빠르고 쉽지만 유지보수의 악몽(maintenance nightmares)으로 이어진다.
- 기능 중복(*Duplicated functionality*): 유사한 기능이 여러 팀에 의해 재구현되어 코드 비대화(code bloat)로 이어진다.

객체 지향 소프트웨어에서 발생하는 세분화된 가장 일반적인 문제는 다음과 같다.

- 상속의 오용(*Misuse of inheritance*): for composition, code reuse rather than polymorphism 컴포지션(composition)의 경우 다형성(polymorphism)이 아닌 코드 재사용(code reuse)
- 상속 누락(*Missing inheritance*): 중복 코드(duplicated code)와 case 문에서 동작 선택하기
- 잘못 위치한 오퍼레이션(*Misplaced operations*): 잘못된 응집력(cohesion) — 클래스 내부가 아닌 외부의 오퍼레이션의 위치
- 캡슐화 위반(*Violation of encapsulation*): 명시적 형 변환, C++ “friends”
- 클래스 남용(*Class abuse*): 낮은 응집력(cohesion) — 클래스를 네임스페이스처럼 사용

마지막으로, 변경하거나 교체하려는 시스템의 모든 부분에 대한 테스트 케이스를 철저하게 개발하여 리엔지니어링 활동을 위한 코드 기반을 준비한다.

리엔지니어링에는 마찬가지로 여러 가지 상호 관련된 활동이 따른다. 물론 가장 중요한 것 중 하나는 시스템의 어떤 부분을 수리하고 어떤 부분을 교체해야 하는지 평가하는 것이다.

실제로 수행되는 코드 변환은 여러 가지 범주로 나뉜다. 치코프스키와 크로스에 따르면 다음과 같다.

“리스터럭처링(*Restructuring*)는 시스템의 외부 동작을 유지하면서 동일한 상대적 추상화 수준에서 한 표현 형식에서 다른 표현 형식으로 변환하는 것이다.”

리스터럭처링은 일반적으로 소스 코드 변환(예: 비정형 “스파게티” 코드에서 구조화된 코드 또는 “고투리스(goto-less)” 코드로 자동 변환)을 의미하지만 디자인 수준에서의 변환도 포함할 수 있다.

리팩토링(*Refactoring*)은 객체 지향 맥락 내에서 리스터럭처링하는 것이다. 마틴 파울러는 이렇게 정의한다.

“리팩토링(*Refactoring*)은 코드의 외부 동작을 변경하지 않으면서 내부 구조를 개선하는 방식으로 소프트웨어 시스템을 변경하는 프로세스이다.”

— 마틴 파울러, [FBB⁺99]

소프트웨어 “리엔지니어링(reengineering)”과 소프트웨어 “유지보수(maintenance)”의 차이를 구분하기 어려울 수 있다. IEEE는 소프트웨어 유지보수를 정의하기 위해 여러 가지 시도를 해왔으며, 여기에는 다음도 포함된다.

“결함을 수정하거나 성능 또는 기타 속성을 개선하거나 변경된 환경에 맞게 제품을 조정하기 위해 납품 후 소프트웨어 제품을 수정하는 것”.

대부분의 사람들은 ‘유지보수’는 일상적인 작업인 반면 ‘리엔지니어링’은

그림 1에서 볼 수 있듯이 시스템을 재구성하는 과감하고 대대적인 노력이라고 생각할 것이다.

그러나 리엔지니어링은 단지 삶의 한 방식이라고 주장하는 사람들도 있다. 조금 개발하고, 조금 리엔지니어링하고, 조금 더 개발하는 식으로 [Bec00]. 실제로 건강하고 유지 관리 가능한 소프트웨어 시스템을 확보하기 위해서는 지속적인 리엔지니어링 문화가 필요하다는 개념을 뒷받침하는 좋은 증거가 있다.

그러나 지속적인 리엔지니어링(continuous reengineering)은 아직 일반적인 실천법이 아니기 때문에 이 책에서는 주요 리엔지니어링 노력의 맥락에서 패턴을 제시한다. 그럼에도 불구하고 독자들은 여기서 소개하는 대부분의 기법이 소규모 반복(iteration)에서 리엔지니어링에도 적용될 수 있다는 점을 염두에 두어야 한다.

1.3 리엔지니어링 패턴

글의 형식으로서의 패턴은 건축가 크리스토퍼 알렉산더가 1977년 그의 획기적인 저서인 *패턴 랭귀지(A Pattern Language)*에서 소개했다. 이 책에서 알렉산더와 그의 동료들은 방에서 건물과 도시에 이르기까지 다양한 종류의 물리적 구조물을 설계하는 체계적인 방법을 제시했다. 각 문제는 여러 가지 주요한 요구사항(force)을 해결하는 일반적인 해결책인 반복되는 패턴(patter)의 형태를 제시했지만, 특정 상황에 따라 각 문제에 고유한 방식으로 적용되어야 한다. 각 패턴에서 제시된 실제 해법은 그다지 흥미롭지 않았고, 오히려 패턴이 전달하는 주요한 요구 사항(force)과 트레이드오프(tradeoff)에 대한 논의가 포함되어 있다.

패턴은 디자인 문제에 대한 반복적인 해결책을 문서화하는 방법으로 소프트웨어 커뮤니티에서 처음 채택되었다. 알렉산더의 패턴과 마찬가지로, 각 디자인 패턴에는 해결해야 할 여러 가지 주요한 요구사항과 패턴을 적용할 때 고려해야 할 여러 가지 트레이드오프가 포함되었다. 패턴은 전문가들이 사용하는 실제 기법뿐만 아니라 그 뒤에 숨은 동기와 근거를 전달하는 간결한 방법인 것으로 밝혀졌다. 이후 패턴은 디자인 이외의 소프트웨어 개발의 여러 측면, 특히 소프트웨어를 설계하고 개발하는 프로세스(process)에 적용되었다.

리엔지니어링 프로세스는 다른 프로세스와 마찬가지로 많은 표준 기법이

등장했으며, 각 기법은 다양한 주요한 요구사항을 해결하고 많은 트레이드오프를 포함할 수 있다. 모범 사례를 전달하는 방법으로서의 패턴은 이러한 기법을 제시하고 논의하는 데 특히 적합하다.

리엔지니어링 패턴(*reengineering patterns*)은 레거시 소프트웨어 수정에 관한 지식을 체계화하고 기록하여 문제를 진단하고 시스템의 추가 개발을 방해할 수 있는 약점을 파악하는 데 도움이 되며, 새로운 요구 사항에 더 적합한 솔루션을 찾는 데도 도움이 된다. 리엔지니어링 패턴은 모든 리엔지니어링 작업에서 참조할 수 있는 안정적인 전문 지식 단위로, 완전한 방법론을 제안하지 않고 프로세스를 설명하며 특정 도구를 “판매”하려는 것보다는 적절한 도구를 제안한다.

리버스 엔지니어링 및 리엔지니어링 패턴의 대부분은 소프트웨어 설계와 관련이 있다는 점에서 디자인 패턴과 표면적으로 유사하다. 그러나 디자인 패턴은 디자인 문제에 대한 특정 해결책을 선택하는 것과 관련이 있는 반면, 리엔지니어링 패턴은 기존 디자인을 발견(*discovering an existing design*)하고, 어떤 문제(*problem*)가 있는지 파악하고, 이러한 문제를 해결(*repairing*)하는 것과 관련이 있다는 점에서 중요한 차이가 있다. 결과적으로 리엔지니어링 패턴은 순전히 주어진 디자인 구조보다는 발견과 변형의 프로세스(*process of discovery and transformation*)와 더 관련이 있다. 이러한 이유로 이 책에 나오는 대부분의 패턴의 이름은 어탭터 [p. 349] 또는 파사드 [p. 350] 와 같이 구조 지향적이기보다는 항상 실행 버전 보유하기 [p. 221] 와 같이 프로세스 지향적이다.

디자인 패턴이 반복되는 디자인(*design*) 문제에 대한 해결책을 제시하는 반면, 리엔지니어링 패턴은 반복되는 리엔지니어링(*reengineering*) 문제에 대한 해결책을 제시한다. 리엔지니어링 패턴으로 생성되는 산출물이 반드시 디자인 일 필요는 없다. 리팩터링된 코드처럼 구체적일 수도 있고, 리버스 엔지니어링 패턴의 경우 시스템 작동 방식에 대한 인사이트처럼 추상적일 수도 있다.

좋은 리엔지니어링 패턴의 특징은 (a) 기존 시스템 상태와 비교하여 목표 산출물의 장점, 비용 및 결과를 명확하게 드러내는 것이지, 그 결과가 얼마나 우아한지가 아닌 것이고, (b) 리엔지니어링 프로세스(*process*)에 대한 설명, 즉 시스템의 한 상태에서 다른 상태로 이동하는 방법을 명확하게 설명하는 것이다.

리엔지니어링 패턴은 코드 리팩터링 그 이상을 포함한다. 리엔지니어링 패턴은 증상을 감지하는 것으로 시작하여 새로운 솔루션에 도달하기 위한 코드

리팩터링으로 끝나는 프로세스를 설명할 수 있다. 리팩터링은 이 프로세스의 마지막 단계에 불과하며, 새로운 솔루션을 구현하기 위해 코드를 자동 또는 반자동으로 수정하는 기술적 문제를 해결한다. 리엔지니어링 패턴에는 리팩터링의 일부가 아닌 다른 요소도 포함되는데, 리엔지니어가 직면하고 있는 제약 조건을 고려하여 증상의 맥락을 강조하고 리팩터링된 솔루션이 가져올 수 있는 변경의 영향에 대한 논의가 여기에 포함된다.

1.4 리엔지니어링 패턴의 형식

(TODO: PNG 파일 처리 필요 혹은 현상태 유지)

이 책에서 사용하는 형식(form)을 설명하는 간단한 패턴의 예가 그림 1.2에 나와 있다. 패턴마다 다른 종류의 문제를 다루기 때문에 실제 사용되는 형식은 패턴마다 약간 다를 수 있지만 일반적으로 같은 종류의 큰제목(heading)을 볼 수 있다.

패턴의 이름(name)은 잘 선택하면 패턴을 기억하기 쉽고 동료들과 토론하기 쉽게 한다. (“이해를 위한 리팩터링 을 하지 않으면 여기서 무슨 일이 일어나고 있는지 알 수 없을 것 같다.”) 의도(intent)는 패턴의 본질을 매우 간결하게 전달하고 현재 상황에 적용 가능한지 여부를 알려주어야 한다.

많은 리엔지니어링 패턴은 코드 변환과 관련이 있으며, 이 경우 다이어그램을 사용하여 어떤 종류의 변환이 일어나는지 설명할 수 있다. 일반적으로 이러한 패턴에는 해결해야 할 문제를 감지하는 단계와 변환 전후의 상황을 보여주는 코드 조각이 추가로 포함된다.

1.5 리엔지니어링 패턴 지도

이 책의 패턴은 앞서 제시한 리엔지니어링 라이프사이클(reengineering lifecycle)에 따라 정리되어 있다. 그림 3에서 이 책의 각 챕터가 라이프사이클에 따라 패턴의 클러스터로 표현된 것을 볼 수 있다. 이 다이어그램은 패턴이 순서대로 적용될 수 있음을 시사한다. 하지만 실제로는 리버스 엔지니어링과 리엔지니어링 작업을 반복할 가능성이 더 높다. 이 다이어그램은 ‘워터폴(Waterfall)’라

If It Ain't Broke, Don't Fix It	<i>The name is usually an action phrase.</i>
<i>Intent:</i> Save your reengineering effort for the parts of the system that will make a difference.	<i>The intent should capture the essence of the pattern</i>
Problem Which parts of a legacy system should you reengineer? <i>This problem is difficult because:</i>	<i>The problem is phrased as a simple question. Sometimes the context is explicitly described.</i>
<ul style="list-style-type: none"> Legacy software systems can be large and complex. Rewriting everything is expensive and risky. <i>Yet, solving this problem is feasible because:</i> <ul style="list-style-type: none"> Reengineering is always driven by some concrete goals. 	<i>Next we discuss the forces! They tell us why the problem is difficult and interesting. We also pinpoint the key to solving the problem.</i>
Solution Only fix the parts that are “broken” — that can no longer be adapted to planned changes.	<i>The solution sometimes includes a recipe of steps to apply the pattern.</i>
Tradeoffs <i>Pros</i> You don't waste your time fixing things that are not only your critical path.	<i>Each pattern entails some positive and negative tradeoffs.</i>
<i>Cons</i> Delaying repairs that do not seem critical may cost you more in the long run.	
Difficulties It can be hard to determine what is “broken”.	<i>There may follow a realistic example of applying the pattern.</i>
Rationale There may well be parts of the legacy system that are ugly, but work well and do not pose any significant maintenance effort. If these components can be isolated and wrapped, it may never be necessary to replace them.	<i>We explain why the solution makes sense.</i>
Known Uses Alan M. Davis discusses this in his book, <i>201 Principles of Software Development</i> .	<i>We list some well documented instances of the pattern.</i>
Related Patterns Be sure to Fix Problems, Not Symptoms.	<i>Related patterns may suggest alternative actions. Other patterns may suggest logical followup action.</i>
What Next Consider starting with the Most Valuable First.	

그림 1.2: 일반적인 리엔지니어링 패턴의 표면

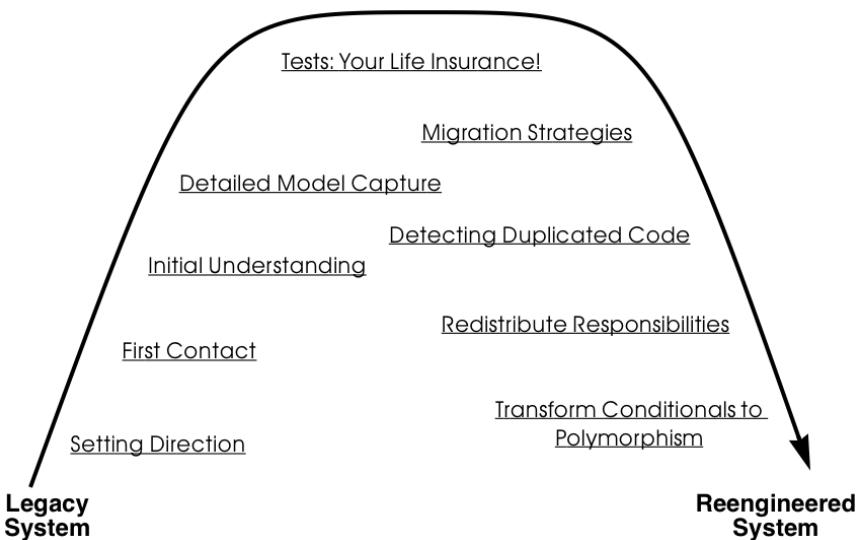


그림 1.3: 리엔지니어링 패턴의 클러스터 지도

이프사이클이 단순한 것과 같은 의미에서 단순하다. 비록 순차적으로 수행되지 않고 반복적으로 수행된다는 것을 알고 있더라도 다양한 소프트웨어 엔지니어링 활동과 그 관계를 추적하는 데 유용한 방법이 될 수 있다.

각 패턴 클러스터(cluster of patterns)는 공통의 문제 집합을 해결하기 위해 결합될 수 있는 관련 패턴의 집합인 간단한 “패턴 언어(pattern language)” —로 표시된다. 따라서 각 장은 일반적으로 해당 장의 패턴에 대한 개요와 맵으로 시작하여 패턴이 어떻게 연관될 수 있는지 제안한다.

방향 설정에는 리엔지니어링 노력의 초점을 어디에 맞출지 결정하고 진행 상황을 파악하는 데 도움이 되는 몇 가지 패턴이 포함되어 있다. 첫 번째 접근은 레거시 시스템을 처음 접할 때 유용할 수 있는 패턴 집합으로 구성되어 있다. 초기 이해는 주로 클래스 다이어그램의 형태로 레거시 시스템의 첫 번째 간단한 모델을 개발하는 데 도움이 된다. 디테일한 모델 캡처는 시스템의 특정 구성 요소에 대한 보다 상세한 모델을 개발하는 데 도움이 된다.

테스트라는 생명보험은 레거시 시스템을 이해하는 데 도움이 될 뿐만 아니라 리엔지니어링 작업에 대비하기 위해 테스트를 사용하는 데 중점을 둔다.

マイグレーション 전략은 리엔지니어링하는 동안 시스템을 계속 실행하고 새 시스템이 사용자에게 받아들여질 가능성을 높이는 데 도움이 된다. 중복 코드 감지는 다른 버전의 소프트웨어에서 코드를 복사하여 붙여넣거나 병합했을 수 있는 위치를 식별하는 데 도움이 될 수 있다. 책임 재배포는 너무 많은 책임을 가진 클래스를 발견하고 리엔지니어링하는데 도움이 된다. 다양성 적용한 조건문 변환은 객체 지향 디자인이 시간이 지남에 따라 손상된 경우 책임을 재분배하는데 도움이 된다.

제 II 편

리버스 엔지니어링

제 2 장

방향 설정

리엔지니어링 프로젝트를 시작하면 경영진, 사용자, 자신의 팀 등 다양한 측면에서 영향을 받는다. 기술적으로 가장 흥미로운 부분이나 가장 쉽게 고칠 수 있을 것 같은 부분에 집중하고 싶은 유혹에 빠지기 쉽다. 하지만 최선의 전략은 무엇일까? 리엔지니어링 작업의 방향을 어떻게 설정하고, 일단 시작한 후에는 어떻게 방향을 유지할까?

포스: 주요한 요구사항

- 일반적인 리엔지니어링 프로젝트에는 서로 다른 방향을 가지는 많은 이해관계가 얹혀 있다. 기술적, 인체공학적, 경제적, 정치적 고려 사항으로 인해 여러분과 여러분의 팀은 집중력을 확립하고 유지하기가 어려울 것이다.
- 리엔지니어링 프로젝트의 커뮤니케이션은 기존 프로젝트의 개발팀의 유무에 따라 복잡해질 수 있다.
- 레거시 시스템은 시스템의 미래에 최선이 아닐 수 있는 특정 아키텍처 (architecture)로 사용자에게 강요할 수 있다.
- 레거시 소프트웨어에서 많은 문제를 발견할 수 있으며 우선순위를 설정하기가 어려울 것이다.

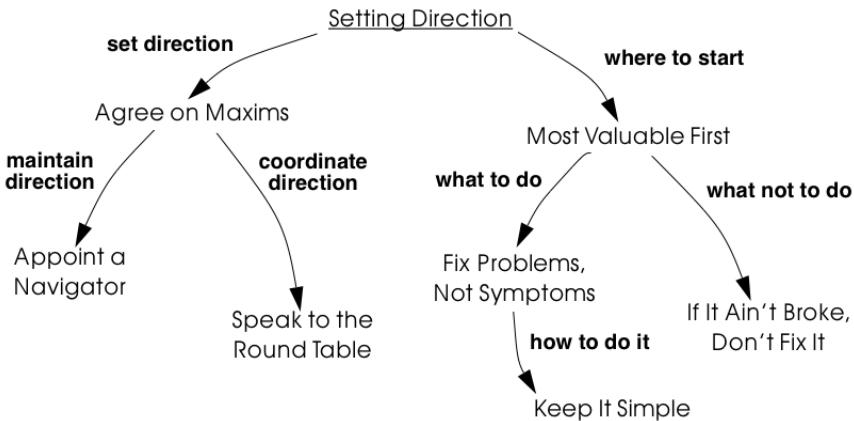


그림 2.1: 리엔지니어링 프로젝트에서 방향을 설정하고 유지하기 위한 원칙과 지침

- 프로젝트에 가장 적합한 것이 아니라 가장 관심 있는 기술적 문제에 집중하는 유혹에 빠지기 쉽다.
- 레거시 시스템의 문제가 있는 구성 요소를 래핑(wrapping)할지, 리팩터링(refactoring)할지, 다시 작성(rewrite)할지 결정하기 어려울 수 있다. 이러한 각 옵션은 서로 다른 위험을 해결하며 필요한 노력, 결과를 평가할 수 있는 속도, 향후 수용할 수 있는 변경 사항의 종류에 따라 다른 결과를 가져온다.
- 시스템을 재설계할 때 가능한 모든 상황에 대처하기 위해 새 솔루션을 과도하게 설계(over-engineer)하고 싶은 유혹을 받을 수 있다.

Overview

방향 설정은 모든 개발 프로젝트에 적용할 수 있지만 리엔지니어링 작업과 특별한 관련성이 있는 패턴의 클러스터이다. 따라서 이를 설명하기 위해 문제(Problem), 해결책(Solution) 및 토론(Discussion)를 가지는 간결한 패턴 형식(streamlined pattern format)을 선택했습니다.

리엔지니어링 팀 내에서 무엇이 문제이고 어떻게 달성할 수 있는지에 대

한 공통된 이해를 확립하기 위해 맥심에 동의하기 를 작성해야 한다. 아키텍처 비전을 유지하기 위해 내비게이터 지정하기 를 수행해야 한다. 모두가 라운드 테이블에 말하기 를 통해 프로젝트 상태에 대한 팀 인식을 유지해야 한다.

올바른 문제와 중요한 결정에 집중할 수 있도록 가장 가치 있는 것 먼저를 사용하는 것이 현명하다. 이렇게 하면 사용자 참여시키기 및 자신감 만들기에 도움이 된다는 점에 참고하자. 래핑, 리팩터링 또는 재작성 여부를 결정하려면 중상이 아닌 문제 수정 를 사용해야 한다. 변경을 위한 변경은 생산적이지 않으므로 고장이 나지 않으면 수정하지 않기 를 사용하자. 새 시스템을 매우 유연하고 범용적으로 만들고 싶은 유혹이 있을 수 있지만, 거의 항상 단순함 유지하기 을 사용하는 것이 더 좋다.

2.1 맥심에 동의하기

문제 팀에서 공통의 목적 의식(purpose)을 확립하려면 어떻게 해야 하는가?

해결 프로젝트의 핵심 우선순위를 정하고 팀이 궤도에 오르는 데 도움이 될 지침 원칙을 파악하자.

토론 모든 리엔지니어링 프로젝트는 상충되는 수많은 이해관계에 대처해야 한다. 경영진은 제품의 경쟁력을 개선하고 유지보수 비용을 줄임으로써 레거시를 보호하고자 한다. 사용자는 기존 업무 패턴을 방해하지 않으면서 향상된 기능을 원한다. 개발자와 유지 관리자는 업무가 더 단순해지기를 원한다. 팀원들은 새로운 시스템의 모습에 대해 각자의 아이디어를 가지고 있을 수 있다.

우리의 비즈니스 모델(business model)은 무엇인가? 또는 누가 무엇을 책임지는가?와 같은 근본적인 질문에 대한 명확한 이해가 없다면 이해관계가 상충되어 팀이 분열되고 목표를 달성하지 못할 위험이 있다. 맥심(Maxim)은 여러 방향으로 끌려가는 프로젝트를 이끄는 데 도움이 되는 행동 규칙이다. 골드버그와 루빈은 [GR95]에서 “모두가 테스트와 디버깅에 책임이 있다”, “처음부터 제대로 할 수는 없다” 등 수많은 맥심의 예를 제시한다.

이 장의 모든 패턴은 팀을 안내하고 궤도에 맞추기 위한 것이므로 패턴이라기보다는 맥심(maxim)¹으로 읽을 수 있다. 예를 들어 가장 가치 있는 것을 먼저 와 같은 맥심은 팀이 기술적으로는 흥미롭지만 레거시 시스템을 보호하거나 가치를 더하지 않는 한계적인 측면에 리엔지니어링 노력을 낭비하지 않도록 하기 위한 것이다. 맥심에 동의하기는 그 자체가 격언이므로 팀이 언제 방향타가 없는지 감지하는 데 도움이 될 수 있다.

기억해야 할 핵심 사항은 맥심의 수명이 제한적일 수 있다는 점이다. 채택된 맥심의 유효성을 주기적으로 재평가하는 것이 중요하다. 잘못된 맥심에 동의하거나 올바른 맥심에 동의하지만 시기가 잘못되면 프로젝트가 완전히 궤도에서 벗어날 수 있다.

¹행동에 본보기가 될 내용을 담고 있는 짧은 격언-옮긴이

2.2 내비게이터 지정하기

문제 복잡한 프로젝트 진행 중에 아키텍처 비전을 어떻게 유지하는가?

해결 아키텍처 비전이 유지될 수 있도록 내비게이터 역할을 담당할 사람을 지정하자.

토론 모든 시스템의 아키텍처는 시간이 지남에 따라 새로운 요구 사항과 관련성이 떨어지면서 성능이 저하되는 경향이 있다. 리엔지니어링 프로젝트의 과정은 레거시 시스템을 몇 년 더 지속하고 발전시킬 수 있는 새로운 아키텍처 비전을 개발하는 것이다. 내비게이터가 없으면 기존 시스템의 설계와 아키텍처가 새 시스템에 스며들어 그대로 이어지게 된다.

새로운 아키텍처가 해결해야 할 가장 중요한 문제가 무엇인지 결정하고 리엔지니어링 프로젝트 초기에 이러한 측면을 테스트할 수 있도록 가장 가치 있는 것을 먼저 를 해결해야 한다.

건전한 아키텍처는 증상이 아닌 문제 수정에 도움이 된다.

앨런 오캘러핸은 내비게이터를 “불꽃의 수호자(Keeper of the Flame)”라고 부르기도 한다 [ODF99].

2.3 원탁 회의에 말하기

문제 팀 소통의 동기화를 어떻게 유지하는가?

해결 간단하고 정기적인 원탁 회의를 개최하자.

토론 레거시 시스템에 대한 지식과 이해는 항상 분산되어 있으며 대개 숨겨져 있다. 리엔지니어링 팀도 고고학을 수행하고 있다. 레거시 시스템에서 추출된 정보는 이를 활용하기 위해 반드시 공유해야 하는 귀중한 자산이다.

누구도 회의(meeting)를 할 시간이 없지만, 회의가 없으면 커뮤니케이션은 임시방편적이고 무작위로 이루어진다. 정기적이고 집중적인 원탁 회의는 팀원들이 현재 상황에 대한 정보를 공유할 수 있도록 하는 목표를 달성할 수 있다. 원탁 회의는 짧게 진행하되 모든 사람이 참여해야 한다. 간단한 접근 방식은 모든 사람이 지난 회의 이후 무엇을 했는지, 무엇을 배웠는지, 또는 어떤 문제가 발생했는지, 다음 회의까지 무엇을 할 계획인지 말하도록 하는 것이다.

원탁 회의는 적어도 일주일에 한 번, 많게는 매일 개최하는 것이 좋다.

회의록은 진행 상황을 기록하는 데 중요하지만 회의록을 작성하는 것은 귀찮은 일이 될 수 있다. 회의록을 간단하게 작성하려면 특정 기한(deadline)까지 수행해야 할 결정사항(decisions) 및 조치사항(actions)만 기록하자.

벡과 파울러는 원탁 회의를 짧게 진행하는 방법으로 “스탠드업 미팅(Stand Up Meetings)”(의자 없는 회의)을 추천한다 [BF01].

2.4 가장 가치 있는 것 먼저 하기

문제 어떤 문제에 먼저 집중해야 할까?

해결 고객에게 가장 가치 있는 측면부터 작업을 시작하자.

토론 레거시 시스템에는 수많은 문제가 있을 수 있으며, 그 중에는 중요한 문제도 있고 고객의 비즈니스에 전혀 중요하지 않은 문제도 있을 수 있다. 가장 중요한 부분에 먼저 집중하면 문제가 되는 올바른 문제를 파악할 가능성이 높아지며, 마이그레이션할 아키텍처 또는 새 시스템에 어떤 종류의 유연성을 구축할지 등 가장 중요한 결정을 프로젝트 초기에 테스트할 수 있다.

고객에게 가치 있는 시스템 부분에 먼저 집중함으로써 여러분과 팀원, 고객이 프로젝트에 대해 최대한 협신할 수 있게 한다. 또한 리엔지니어링 노력이 가치 있고 필요하다는 것을 입증하는 긍정적인 결과를 초기에 얻을 가능성이 높아진다.

그럼에도 불구하고 이 패턴을 적용하는 데는 여러 가지 어려움이 있다.

고객은 누구인가?

- 모든 레거시 시스템에는 많은 이해관계자가 있지만 그 중 단 하나만 고른다면 고객이다. 누가 결정권을 가져야 하는지 명확하게 이해한 경우에만 우선순위를 설정할 수 있다.

무엇이 가치 있는 일인지 어떻게 알 수 있을까?

- 고객에게 가장 가치 있는 측면이 무엇인지 정확히 평가하는 것은 어려울 수 있다. 한 회사에서 아키텍처를 바꾸고 싶어서 시스템을 모듈화할 수 있는지 평가해 달라고 요청한 적이 있다. 하지만 오랜 논의 끝에 실제로는 비즈니스 규칙을 더 명확하게 표현할 수 있는 시스템이 필요하다는 것, 즉 한 명의 프로그래머만 이해하는 위험을 줄이기 위해 신입 프로그래머라도 더 쉽게 이해할 수 있는 시스템을 원한다는 것이 밝혀졌다.
- 고객의 비즈니스 모델을 이해하려고 노력하라. 이를 통해 시스템의 다양한 측면의 가치를 평가하는 방법을 알 수 있다. 비즈니스 모델과 직접 관련이 없는 모든 것은 순전히 기술적인 측면의 문제일 가능성이 높다.

- 고객이 얻고자 하는 측정 가능한 목표가 무엇인지 파악하라. 예를 들어 응답 시간 개선, 새로운 기능의 출시 시간 단축, 개별 고객의 요구에 더 쉽게 맞춤화 등 시스템의 일부 측면 또는 시스템의 진화에 대한 외형적 표현이어야 한다.
- 주요 목표가 주로 기존 자산을 보호하는 것인지, 아니면 새로운 기능이나 기능의 측면에서 가치를 추가하는 것인지 이해하려고 노력하라.
- 변경 로그를 살펴보고 시스템에서 시간상으로 가장 많은 변경 활동이 있었던 위치를 파악하라. 가장 가치 있는 산출물(artifact)는 대개 가장 많은 변경 요청을 받은 산출물이다(과거로부터 학습 [p. 157] 참조).
- 고객이 우선순위를 설정할 의향이 없거나 설정할 수 없는 경우에는 계획 세우기 게임 (Planning Game)을 수행하자[BF01]. 모든 이해관계자로부터 요구사항을 수집하고 식별 가능한 각 작업에 필요한 노력에 대한 대략적인 추정치를 만든다. 초기 첫 번째 마일스톤에 대한 초기 노력 예산이 주어지면 고객에게 예산에 맞는 작업을 선택하도록 요청한다. 각 반복 (iteration)마다 이 플래닝 게임 적용을 실천한다.
- 인식 변화(changing perceptions)에 주의하라. 처음에는 고객이 문제 자체보다는 레거시 시스템에서 나타나는 특정 증상에 주목할 수 있다(증상이 아닌 문제 수정 [p. 37] 참조).

기대치가 너무 높아질 위험이 있지 않는가?

- 좋은 초기 결과를 제공하지 못하면 많은 것을 배울 수 있지만 신뢰도를 잃을 위험이 있다. 따라서 고객에게 가치를 제공할 뿐만 아니라 성공 가능성이 높은 초기 작업을 신중하게 선택하는 것이 중요하다. 따라서 초기 작업의 노력을 추정할 때 세심한 주의를 기울이자.
- 성공의 열쇠는 작고 빈번한 반복(iteration)을 계획하는 것이다. 고객이 파악한 초기 작업이 너무 커서 단기간(예: 2주)에 초기 결과를 보여줄 수 없다면 더 짧은 반복으로 해결할 수 있는 작은 하위 작업으로 세분화하자. 첫 단계에 성공하면 기대치가 높아지겠지만, 단계가 작아도 나쁘지 않다.

가장 가치 있는 부분이 쥐의 둥지(rat's nest)라면 어떨까?

- 안타깝게도 레거시 시스템을 리엔지니어링하는 것은 정상적이고 주기적인 혁신 프로세스가 아니라 절박한 상황에서 이루어지는 경우가 많다. 시스템에서 가장 중요한 부분이 가장 복잡하고 난공불락이며 수정과 더 벼깅이 어려운 부분일 수도 있다.
- 높은 변경률(high changes rates)은 소프트웨어 결함이 많다는 신호일 수도 있다. 소프트웨어 결함의 80%는 일반적으로 코드의 5%에서 발생하므로 “최악을 먼저 고친다”는 전략은 시스템에서 가장 심각한 문제의 원인을 제거함으로써 큰 성과를 거둘 수 있다 [Dav95]. 그럼에도 불구하고 상당한 위험이 따른다.
 - 조기에 긍정적인 결과를 입증하기 어려울 수 있다.
 - 정보가 거의 없는 상태에서 시스템의 가장 복잡한 부분을 다룰 수 있다.
 - 실패할 가능성이 높다.
- 문제가 있는 구성 요소를 래핑할지, 리팩터링할지 또는 다시 작성할지 여부를 증상이 아닌 문제 수정 를 확인하여 결정한다.

시스템에서 작업할 가장 중요한 부분이 무엇인지 결정한 후에는 리엔지니어링 노력에 사용자 참여시키기 [p. 205]를 수행하여 신뢰 구축하기 [p. 209]을 수행해야 한다. 시스템 점진적 마이그레이션하기 [p. 213]을 사용하면 사용자가 리엔지니어링된 시스템을 사용하면서 지속적인 피드백을 제공할 수 있다.

2.5 증상이 아닌 문제 수정하기

문제 보고된 모든 문제를 어떻게 해결할 수 있을까?

해결 이해관계자의 요청이 아닌 문제의 원인을 해결하자.

토론 이것은 매우 일반적인 원칙이지만 리엔지니어링과 특히 관련이 있다. 이해관계자 (stakeholder)마다 시스템에 대한 관점이 다르고 시스템의 일부만 볼 수도 있다. 그들이 고치기를 원하는 문제는 시스템의 더 깊은 문제를 드러내는 것일 수도 있다. 예를 들어 특정 사용자 작업에 대한 즉각적인 피드백을 받지 못하는 것은 데이터 흐름 아키텍처의 결과일 수 있다. 해결 방법을 구현하면 문제가 악화되어 더 많은 해결 방법이 필요할 수 있다. 이것이 진짜 문제라면 적절한 아키텍처로 마이그레이션해야 한다.

리엔지니어링 작업 중 흔히 발생하는 어려움은 레거시 컴포넌트를 래핑할지, 리팩터링할지, 다시 작성할지 결정하는 것이다. 가장 가치 있는 것 먼저 하기는 시스템의 문제에 어떤 우선순위를 부여할지 결정하는 데 도움이 되며, 어떤 문제가 중요한 경로에 있는지를 알려준다. 증상이 아닌 문제 수정하기는 문제의 현상이 아닌 문제의 원인에 집중하도록 한다. 예를 들자면 다음과 같은 것들이 있다.

- 레거시 컴포넌트의 코드가 기본적으로 안정적이고 문제가 주로 클라이언트 변경에서 발생하는 경우, 코드가 아무리 엉망이라도 구현이 아니라 레거시 컴포넌트에 대한 인터페이스에 문제가 있을 가능성이 높다. 이러한 경우 올바른 인터페이스 제공하기 [p. 229]를 적용하여 인터페이스만 수정하는 것을 고려해야 한다.
- 레거시 컴포넌트에 결함이 거의 없지만 시스템 변경에 큰 병목 현상이 발생하는 경우 향후 변경의 영향을 제한하기 위해 리팩터링해야 할 수도 있다. 보다 깔끔한 디자인으로 마이그레이션하기 위해 신 클래스 분할하기 [p. 289]를 적용하는 것을 고려할 수 있다.
- 레거시 컴포넌트에 많은 결함이 있는 경우, 레거시 데이터를 새 구현으로 마이그레이션하기 위한 전략으로 새로운 마을로 다리 만들기 [p. 225]을 적용하는 것을 고려하자.

이 패턴은 고장이 나지 않으면 수정하지 않기 과 충돌하는 것처럼 보일 수 있지만 실제로는 그렇지 않다. 실제로 '깨지지 않은' 것은 문제의 원인이 될 수 없다. 예를 들어 래핑은 해결 방법처럼 보일 수 있지만 실제 문제가 레거시 컴포넌트에 대한 인터페이스에만 있는 경우 올바른 해결책이 될 수 있다.

2.6 고장이 나지 않으면 수정하지 않기

문제 레거시 시스템에서 어떤 부분을 리엔지니어링하고 어떤 부분을 그대로 둬야 할까?

해결 “고장난” 부분, 즉 계획된 변경에 더 이상 적용할 수 없는 부분만 수정하자.

토론 변화를 위한 변화가 반드시 좋은 것만은 아니다. 레거시 시스템에는 보기 흉할 수 있지만 잘 작동하고 유지 관리에 큰 부담이 되지 않는 부분이 있을 수 있다. 이러한 구성 요소를 분리하고 포장할 수 있다면 교체할 필요가 없을 수도 있다.

무언가를 “수정”할 때마다 시스템의 다른 무언가가 손상될 위험도 있다. 또한 사소한 문제에 귀중한 시간과 노력을 낭비할 위험도 있다.

리엔지니어링 프로젝트에서 “고장난” 부분은 레거시를 위험에 빠뜨리는 부분이다.

- 새로운 요구 사항을 충족하기 위해 자주 조정해야 하지만 높은 복잡성과 디자인 드리프트(design drift)²로 인해 수정하기 어려운 컴포넌트(component).
- 중요하지만 전통적으로 많은 결함을 포함하는 컴포넌트.

안정적이고 레거시 시스템의 미래를 위협하지 않는 소프트웨어 산출물은 코드가 어떤 상태에 있든 “고장난” 것이 아니므로 다시 엔지니어링할 필요가 없다.

²설계와 구현의 차이-옮긴이

2.7 간단하게 유지 하기

문제 새 시스템에 얼마나 많은 유연성을 구축해야 할까?

해결 더 일반적이고 복잡한 솔루션보다 적절하고 간단한 솔루션이 더 낫다.

토론 이것은 리엔지니어링에 특별한 의미를 갖는 또 다른 일반적인 원칙이다. 우리는 실제로 얼마나 많은 일반성과 유연성이 필요한지 추측하는 데 서툴다. 많은 소프트웨어 시스템은 생각할 수 있는 모든 기능이 추가되면서 비대해진다.

유연성(flexibility)은 양날의 검입니다. 리엔지니어링의 중요한 목표는 미래의 변화를 수용하는 것이다. 하지만 지나친 유연성은 새로운 시스템을 너무 복잡하게 만들어 오히려 미래의 변화를 방해할 수 있다.

어떤 사람들은 “재사용을 위한 계획(plan for reuse)”이 필요하므로 누군가에게 유용할 수 있는 모든 소프트웨어 개체가 가능한 한 많은 노브와 버튼으로 가능한 가장 일반적인 방식으로 프로그래밍되도록 추가적인 노력을 기울여야 한다고 주장한다. 하지만 누가 어떤 용도로 어떤 것을 사용할지 예측하는 것은 거의 불가능하기 때문에 이런 방식은 거의 효과가 없다. 최종 사용자 소프트웨어도 마찬가지이다.

“일할 수 있는 가장 간단한 것을 하라”는 익스트림 프로그래밍의 맥심이다. [Bec00]의 맥심으로, 모든 리엔지니어링 노력에 적용된다. 이 전략은 사용자가 평가하고 대응할 수 있는 간단한 변경 사항을 신속하게 도입하도록 장려하기 때문에 사용자 참여 시키기 [p. 205] 및 신뢰 구축 하기 [p. 209] 을 강화한다.

복잡한 작업을 수행하면 (실제로 필요한 것이 무엇인지) 잘못 추측할 수 있고 수정하기가 더 어려워진다. 일을 단순하게 유지하면 더 빨리 끝내고, 더 빨리 피드백을 받고, 더 쉽게 오류를 복구할 수 있다. 그런 다음 다음 단계로 넘어갈 수 있다.

제 3 장

첫 번째 접근

당신은 의사의 일상 업무를 지원하는 *proDoc*이라는 소프트웨어 시스템을 개발하는 팀의 일원이다. 주요 기능 요구 사항은 (i) 환자 파일의 유지 관리와 (ii) 환자 및 건강 보험에서 지불해야 할 금액을 추적하는 것이다. 스위스의 의료법은 매우 복잡하고 정기적으로 변경되기 때문에 걱정할 경쟁자가 거의 없다. 그럼에도 불구하고 최근 한 신생 스타트업 기업이 *XDoctor*라는 경쟁 제품으로 상당한 시장 점유율을 확보했다. *XDoctor*의 판매 특징은 플랫폼 독립성과 인터넷과의 통합이다. 이 시스템은 내장된 이메일 클라이언트와 웹 브라우저를 제공한다. *XDoctor*는 또한 건강 보험과의 거래 처리를 위해 인터넷을 활용한다.

당신의 회사는 시장에서의 입지를 확보하기 위해 *XDoctor*를 구매했으며 이제 거래에서 가능한 한 많은 금액을 회수하려고 한다. 특히 *XDoctor*에서 인터넷 기능을 분리하여 *proDoc*에 재사용하고자 한다. 두 제품을 하나로 통합하는 방법에 대한 첫 번째 평가와 계획을 수립해 달라는 요청을 받았다. 처음에는 경쟁 제품의 기술적 세부 사항에 대해 알려진 것이 거의 없다. 원래 네 명으로 구성된 개발팀에서 한 명만 회사에 합류했다. 그의 이름은 데이브이고 많은 종이(문서?)와 두 장의 CD가 들어 있는 커다란 상자를 사무실로 가져왔다. 첫 번째는 Windows, MacOS 및 Linux용 인스톨러가 포함된 *XDoctor* 설치 디스크이다. 다른 하나는 약 500,000줄의 Java 코드와 10,000줄의 C 코드가 들어 있다. 책상 위에 놓인 이 상자를 절망적으로 바라보면서 “도대체 어디서부터 시작해야 할까”라는 생각이 들 것이다.

포스: 주요한 요구사항

리엔지니어링 프로젝트가 얼마나 자주 시작되는지는 놀랍다. 두 회사가 합병한 후뿐만 아니라 나중에 파산한 회사에서 코드 라이브러리를 얻거나 전체 유지 보수 팀이 매우 가치 있지만 이해할 수 없는 코드 조각을 남기고 프로젝트를 종료한 프로젝트도 접했다. 물론 당연한 질문은 “어디서부터 시작해야 할까”이다. 이 질문은 리엔지니어링 프로젝트에서 답해야 할 중요한 질문 중 하나이기 때문에 한 장 전체를 할애하여 이 질문에 답하도록 한다.

이 클러스터의 모든 패턴은 리엔지니어링 프로젝트의 초기 단계에 적용될 수 있다. 완전히 새로운 시스템을 마주하고 있으며, 며칠 내에 이 시스템으로 무언가를 할 수 있는지 판단하고 진행 방법을 제시해야 한다. 그러나 이러한 초기 평가는 결정의 장기적인 영향을 고려하면서 정확한 결과를 신속하게 도출해야 하기 때문에 어려운 작업이다. 신속하고 정확한 결과와 장기적인 효과 사이의 내재적 충돌을 해결하기 위해 이 클러스터의 패턴은 다음과 같은 포스를 해결해야 한다.

- 레거시 시스템은 크고 복잡하다. 레거시 시스템을 다룰 때 규모(scale)는 항상 문제가 된다.¹ 그러나 리엔지니어링 팀 하나가 할 수 있는 작업에는 한계가 있으며, 레거시 시스템이 너무 크거나 복잡하면 한 번에 작업을 완료할 수 없다. 따라서 시스템을 관리 가능한 부분으로 나누고, 여기서 관리 가능한 부분은 하나의 리엔지니어링 팀으로 처리할 수 있는 부분이다.

한 팀이 관리할 수 있는 범위는 리엔지니어링 프로젝트의 목표, 기존 시스템의 상태, 팀의 경험과 기술, 조직의 문화에 따라 달라진다. 우리 팀은 35명으로 구성되었으며 50만 100만 줄의 코드를 처리할 수 있었다. 그러나 이러한 수치는 여러분이 직면한 리엔지니어링 프로젝트에 맞게 조정되어야 한다. 경험상 한 팀이 처음부터 작성할 수 있는 만큼의 코드를 리엔지니어링할 수 있다고 가정하자. 팀이 실제로 얼마나 리엔지니어링했는지 로그를 기록하여 리엔지니어링 프로젝트 기간 동안 예상치를 개선하자.

코드를 분할해야 하는 경우 현재 시스템 구조와 유지 관리 팀의 조직을 최대한 가깝게 유지하자. 시스템 구조를 잘 이해하고 나면 프로젝트 목표에 더 적합한 대안을 고려하자.

¹FAMOOS 프로젝트에서 우리는 50만 줄의 C++에서 250만 줄의 Ada에 이르는 시스템과 마주했다.

- 시간은 부족하다. 프로젝트 초기에 시간을 낭비하면 나중에 심각한 결과를 초래할 수 있다. 특히 리버스 엔지니어링 중에는 불확실성을 느끼면 문제의 근본을 해결하는 대신 당분간 바쁘게 지낼 수 있는 활동을 시작하고 싶은 유혹에 빠지기 쉽다. 결론적으로 시간을 가장 소중한 자원으로 생각하라. 따라서 시간이 많이 걸리는 모든 활동은 나중으로 미루고 프로젝트의 첫 날에는 프로젝트 목표의 실현 가능성은 평가하는 데 사용하자. 이 클러스터의 모든 패턴은 프로젝트의 기회와 위험을 빠르게 파악하기 위한 것이므로 프로젝트의 전반적인 방향을 설정하는 데 도움이 된다.
- 첫인상은 위험하다. 불완전한 지식을 바탕으로 중요한 결정을 내린다는 것은 잘못된 결정을 내릴 가능성이 있다는 것을 의미한다. 시스템을 처음 접할 때 이러한 위험을 피할 수 있는 방법은 없지만 항상 출처를 다시 확인 (*always double-check your sources*)하면 그 영향을 최소화할 수 있다.
- 사람들은 서로 다른 의제를 가지고 있다. 일반적으로 리엔지니어링할 시스템에 대해 많은 경험을 가진 여러 사람이 한 그룹에 합류하게 된다. 원래 개발 팀의 구성원이 여전히 남아 있을 수도 있고, 리엔지니어링 팀에 오랫동안 시스템을 유지 관리해 온 사람이 포함될 수도 있다. 적어도 리엔지니어링 프로젝트를 요청할 만큼 이 시스템을 충분히 신뢰하는 최종 사용자와 관리자가 있을 것이다. 여러분은 리엔지니어링 기술과 전문 지식으로 팀을 보완해야 하므로 누구를 상대하고 있는지 파악해야 한다.

일반적으로 새로운 동료는 세 가지 범주에 속합니다. 첫 번째 범주는 리엔지니어링이 필요하다고 믿고 당신이 리엔지니어링을 할 수 있게 그리고 그것을 도와줄 수 있다고 믿어붙이는 충실히(*faithful*) 사람들이다. 두 번째 범주는 회의적(*skeptical*)인 사람들로, 자신의 일자리를 지키고 싶거나 프로젝트 전체를 처음부터 다시 시작해야 한다고 생각하기 때문에 이 모든 리엔지니어링 작업이 시간 낭비라고 생각하는 사람들이이다. 세 번째 범주는 리엔지니어링이 성과를 거둘 수 있을지에 대한 확고한 의견이 없기 때문에 일단 지켜보자는 펜스 시터(*fence sitter*)의 범주이다. 결론적으로 프로젝트를 성공으로 이끌기 위해서는 신봉자들을 계속 설득하고, 펜스 시터들의 신뢰를 얻으며, 회의론자들을 경계해야 한다.

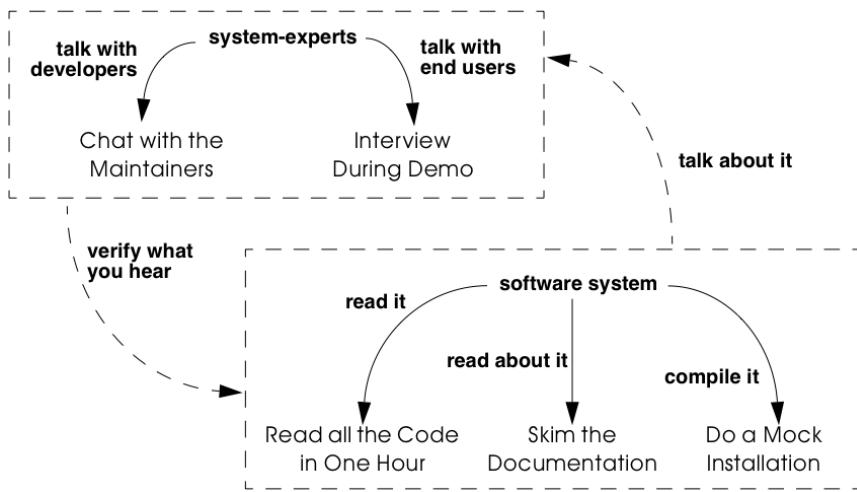


그림 3.1: 시스템에 첫 번째 접근을 진행하는 동안 프로젝트의 구현가능성(feasibility)을 평가한다.

Overview

시스템과 처음 접촉할 때 시간 낭비가 가장 큰 리스크이므로 이러한 패턴은 1주일 정도의 짧은 기간 동안 적용해야 한다. 이번 주가 지나면 주요 이슈를 파악하고 그 지식을 바탕으로 추가 활동을 계획하거나 — 필요한 경우 — 프로젝트를 취소해야 한다.

유지보수자와 담소나누기 및 데모 중 인터뷰하기 패턴은 관련된 사람들과 친해지는 데 도움이 될 것이다. 경험상 4일 동안 정보를 수집하고 마지막 날을 사용하여 이 모든 정보를 첫 번째 프로젝트 계획으로 정리하자. 패턴을 적용하는 데 엄격한 순서는 없지만 책에서 제시하는 순서가 일종의 일반적인 순서이다. 그럼에도 불구하고 재차 확인해야 할 필요성 때문에 이러한 패턴의 조각을 조합하는 경우가 종종 있다. 예를 들어 유지보수자(maintainer)와의 두 번째 미팅에서는 보통 데모 중 인터뷰하기로 시작하지만 한 시간 안에 모든 코드 읽기와 문서 스키밍하기에서 배운 내용에 대해 질문을 하기도 한다. 또한 인터뷰가 끝나면 소스 코드와 문서를 빠르게 확인하여 말한 내용을 확인한다.

특정 상황에서는 리소스 부족으로 인해 일부 패턴이 적용되지 않는 경우가

있다. 예를 들어, 모든 유지보수자가 퇴사한 경우 유지보수자와 담소나누기 를 사용할 수 없다. 또한 특정 시스템에는 외부 사용자 인터페이스가 없기 때문에 최종 사용자와 데모 중 인터뷰하기 를 시도하는 것이 무의미할 수도 있다. 이러한 패턴 중 일부는 프로젝트 목표와 무관할 수 있으므로 반드시 문제가 되는 것은 아니다. 하지만 리소스의 부재는 프로젝트에 추가적인 리스크을 초래하므로 첫 번째 프로젝트 계획에 이를 기록해야 한다.

다음 단계

필요한 정보를 확보했다면 이제 첫 번째 프로젝트 계획을 작성할 차례이다. 이러한 계획은 프로젝트를 시작할 때 일반적으로 사용하는 계획과 매우 유사하므로 회사에서 사용하는 표준 문서 템플릿을 사용해야 한다. 필요한 경우 최소한 다음 항목이 포함되도록 규칙을 수정하자.

- **프로젝트 범위.** 프로젝트의 배경, 목표, 목표 달성을 여부를 확인하는 데 사용할 기준을 포함하여 프로젝트에 대한(반 페이지의) 간단한 설명을 준비 한다. 계획의 이 부분을 작성하려면 사용자 참여시키기 [p. 205] 및 맥심에 동의하기 [p. 27] 를 사용한다.
- **기회.** 프로젝트 목표를 달성하는 데 기여할 것으로 예상되는 요소를 파악 하자. 숙련된 유저보수자와 파워 유저의 가용성(availability), 소스 코드 의 가독성(readability) 또는 최신 문서의 존재 여부 등 첫 번째 접촉에서 발견한 항목을 나열하자.
- **리스크.** 프로젝트 진행 중에 문제를 일으킬 수 있는 요소를 고려하자. 코드 라이브러리가 누락되었거나 테스트 슈트(test suite)가 없는 등 발견하지 못했거나 품질이 떨어지는 항목을 나열하자. 가능하면 각 리스크에 대한 발생가능성(likelihood)(가능성 없음, 가능성 보통, 가능성 높음)과 영향 도(impact)(높음, 보통, 낮음)에 대한 평가를 포함하자. 특히 중요 리스크, 즉 발생가능성이 보통/높고 영향이 보통/높은 리스크 또는 가능성이 높고 영향이 낮은 위험에 대해 특별한 주의를 기울여야 합니다.
- **진행/진행 불가 결정.** 어느 시점에서 프로젝트를 계속 진행할지 아니면 취소할지 결정해야 한다. 위의 기회와 리스크를 사용하여 그 결정에 대해 토론하자.

- 활동. (“진행” 결정의 경우) 프로젝트 목표를 어떻게 달성할 것인지 설명하면서 향후 기간에 대한 조망(fish-eye view)를 준비하자. 조망도에서는 단기 활동은 상당히 상세하게 설명하고, 이후 활동은 대략적인 개요로 충분하다. 대부분의 경우 단기 활동은 초기 이해에 설명된 패턴과 일치할 것이다. 이후 활동은 이후 챕터를 확인하자.

활동 목록은 기회를 활용하고 (중요한) 리스크 줄여야 한다. 예를 들어, 최신 문서가 있다는 것을 기회로, 테스트 슈트가 없다는 것을 중대한 리스크로 나열한 경우, 문서를 기반으로 테스트 스위트를 구축하는 활동을 계획해야 한다.

3.1 유지보수자와 담소나누기

의도 시스템 유지보수 담당자와의 토론을 통해 프로젝트의 역사적, 정치적 맥락에 대해 배워보자.

문제

리엔지니어링하려는 레거시 시스템의 역사적, 정치적 맥락을 제대로 파악하면 어떻게 해야 하는가?

이 문제가 어려운 이유는 다음과 같다.

- 문서가 있는 경우 일반적으로 솔루션에 대한 결정은 기록하지만 솔루션에 영향을 준 요소에 대해서는 기록하지 않는다. 따라서 시스템의 역사에서 중요한 사건(즉, 역사적 맥락)은 거의 문서화되지 않는다.
- 시스템은 가치가 있지만(그렇지 않다면 시스템을 리엔지니어링할 필요가 없다) 경영진은 통제력을 상실했다(그렇지 않다면 시스템을 리엔지니어링할 필요가 없을 것이다). 소프트웨어 시스템과 관련된 사람 중 적어도 일부가 엉망이므로 레거시 시스템의 정치적 맥락은 본질적으로 문제가 있다.
- 시스템과 관련된 사람들이 사용자를 오도할 수 있다. 특히 시스템에서 문제가 있는 부분을 담당하거나 자신의 일자리를 보호하려는 경우 고의적으로 사용자를 속이는 경우가 있다. 대부분의 경우, 특히 수석 개발자가 다른 프로젝트에 참여하고 주니어 직원만 남아 시스템 유지보수를 담당할 때 무지에서 비롯된 오해를 불러일으킬 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 유지보수 팀(*maintenance team*)과 논의할 수 있다. 그들은 원래 시스템 컨테스트에 대해 모든 것을 알지 못할 수도 있지만 시스템이 현재 상태가 된 경위에 대해서는 상당 부분 알고 있을 가능성이 높다.

해결

시스템 유자보수자와 논의하자. 레거시 시스템에 밀접하게 관여해 온 기술 담당자들은 시스템의 역사와 그 역사에 영향을 미친 사람 관련 문제를 잘 알고 있다.

오해의 소지가 있는 정보를 피하려면 유자보수자를 “전우(brothers in arms)”로 대하자. 그들이 하는 일에 대해 설명하는 데 시간을 내준다면 그들의 일을 더 쉽게 해줄 수 있는 것(더 많은 보람, 더 많은 인정, — 그들을 설득할 가능성이 가장 높은 것)을 제공하는 일종의 협상을 시도해 보자. 이렇게 하면 리엔지니어링 프로젝트의 후반 단계에서 필요한 존경심을 얻을 수 있다는 추가적인 이점도 있다.

힌트

다음은 유자보수자와 논의할 때 도움이 될 수 있는 몇 가지 질문이다. 이러한 질문은 (공식 회의록이나 안건이 없는) 비공식 회의 중에 하는 것이 가장 좋지만, 회의가 끝난 후 주요 결론, 가정 및 우려 사항을 기록하기 위해 메모할 준비를 해야 한다.

- 지난 한 달 동안 가장 쉽게 고칠 수 있었던 버그는 무엇이었나? 그리고 가장 어려웠던 버그는 무엇이었나? 각각을 고치는 데 얼마나 걸렸나? 특정 버그를 고치는 것이 왜 그렇게 쉬웠거나 어려웠나?

이러한 질문은 유지보수 작업에 관심이 있다는 것을 보여주기 때문에 좋은 출발점이 된다. 또한 이러한 질문에 답함으로써 유지보수자는 자신의 뛰어난 능력을 보여줄 수 있는 기회를 얻게 되어 업무에 대한 부담감을 덜 수 있다. 마지막으로, 답변은 나중에 더 높은 수준의 토론에서 사용할 수 있는 유지보수 문제에 대한 몇 가지 구체적인 예를 제공한다.

- 유지보수 팀은 버그 리포트 (*bug report*) 및 기능 요청을 어떻게 수집하는가? 어떤 요청을 먼저 처리할지는 누가 결정하는가? 버그 리포팅이나 기능 요청을 관리자에게 배정하는 것은 누가 결정하는가? 이러한 이벤트는 어떤 종류의 데이터베이스에 기록되는가? 버전 또는 구성 관리 시스템 (*configuration management system*)²이 마련되어 있는가?

²소프트웨어의 변경사항을 체계적으로 추적하고 통제하는 시스템으로 형상 관리 시스템이라고

이러한 질문은 유지보수 프로세스의 조직과 유지보수 팀의 내부 작업 습관을 이해하는 데 도움이 된다. 정치적 맥락에 관한 한 팀 내 관계(작업 할당)와 최종 사용자와의 관계(버그 리포트 수집)를 평가하는 데 도움이 된다.

- 오랫 동안 개발/유지보수 팀의 일원이었던 사람은 누구였는가? 프로젝트에 어떻게 합류/탈퇴했나? 이것이 시스템의 릴리스 기록에 어떤 영향을 미쳤나?

이러한 질문은 레거시 시스템의 역사를 직접적으로 다루는 질문이다. 사람들은 일반적으로 이전 동료에 대해 잘 기억하기 때문에 사람에 대해 물어보는 것이 좋다. 나중에 그들이 어떻게 프로젝트에 합류하거나 떠났는지 물어보면 정치적 맥락도 파악할 수 있다.

- 코드는 얼마나 좋은가? 문서는 얼마나 신뢰할 수 있는가?

이 질문은 특히 유지보수 팀 스스로가 시스템 상태를 얼마나 잘 평가할 수 있는지 확인하는 것과 관련이 있다. 물론 나중에 그들의 주장을 직접 확인해야 한다(한 시간 안에 모든 코드 읽기 및 문서 스키밍하기 참조).

- 이 리엔지니어링 프로젝트를 시작한 이유는 무엇인가? 이 프로젝트에서 무엇을 기대하는가? 결과를 통해 무엇을 얻을 수 있는가?

리엔지니어링 프로젝트를 통해 유지보수자가 무엇을 얻을 수 있는지 묻는 것은 이후 단계에서 염두에 두어야 할 사항이므로 매우 중요하다. 경영진이 프로젝트에서 기대하는 것과 유지보수자가 기대하는 것의 (때로는 미묘한) 차이점에 귀를 기울이자. 차이점을 파악하면 정치적 맥락을 파악하는 데 도움이 된다.

트레이드오프

장점

- 정보를 효과적으로 얻는다. 소프트웨어 시스템의 수명 주기에서 중요한 이벤트는 대부분 구두로 전달된다. 유지보수자와 토론하는 것이 이 풍부한 정보 소스를 활용하는 가장 효과적인 방법이다.

- 동료와 친해지자. 관리자와 토론하면 동료에 대해 처음으로 평가할 수 있는 기회를 갖게 됩니다. 따라서 리엔지니어링 프로젝트의 후반 단계에서 필요한 신뢰를 얻을 수 있을 것입니다.

단점

- 사례 증거만 제공된다. 여러분이 얻는 정보는 기껏해야 사례(anecdotal) 일 뿐이다. 인간의 뇌는 어떤 사실을 기억하는지에 대해 선택적일 수밖에 없으므로 관리자의 기억이 불충분할 수 있다. 더 나쁜 것은 유지보수자가 시스템의 원래 개발자가 아닌 경우가 많기 때문에 정보가 처음부터 불완전할 수 있다는 것이다. 따라서 다른 방법으로 얻은 정보를 보완해야 한다 (예: 문서 스키밍하기, 데모 중 인터뷰하기, 한 시간 안에 모든 코드 읽기 및 모의 설치 하기 참조).

어려움

- 사람들은 자신의 일자리를 보호한다. 일부 유지보수자는 일자리를 잃을까 두려워서 필요한 정보를 제공하지 않을 수도 있다. 리엔지니어링 프로젝트가 그들의 업무를 더 쉽고, 더 보람 있고, 더 가치 있게 만들기 위해 존재한다는 것을 설득하는 것은 여러분의 몫이다. 따라서 유지보수자에게 리엔지니어링 프로젝트에서 무엇을 기대하는지 직접 물어봐야 한다.
- 팀이 불안정할 수 있다. 소프트웨어 유지보수는 일반적으로 2급 업무로 간주되어 주니어 프로그래머에게 맡기는 경우가 많으며 유지보수 팀이 자주 바뀌는 경우가 많다. 이러한 상황에서는 유자보수자가 소프트웨어 시스템의 역사적 진화에 대해 말할 수는 없지만, 그 정치적 맥락에 대해서는 많은 것을 알 수 있다. 실제로 팀의 불안정성은 프로젝트의 위험을 증가시키고 획득한 정보의 신뢰성을 떨어뜨리기 때문에 이러한 불안정성을 인지해야 한다. 따라서 수년 동안 개발/유지보수 팀의 일원이었던 사람이 누구인지 물어봐야 한다.

예시

XDoctor를 인수하는 동안 회사는 원래 개발 팀을 설득하여 두 소프트웨어 시스템을 하나로 통합하려고 노력해 왔다. 안타깝게도 팀원 중 한 명인 데이브만 남기로 동의하고 나머지 세 명은 다른 회사로 떠났다. 두 제품을 통합하는 방법에 대한 계획을 개발하는 것이 당신의 임무이므로 데이브를 점심 식사에 초대하여 시스템에 대한 비공식적인 대화를 나눈다.

이 대화를 통해 많은 것을 배우게 된다. 좋은 소식은 데이브가 건강 보험과의 거래를 처리하는 인터넷 통신 프로토콜을 구현하는 일을 담당했다는 것이다. 이 기능은 제품에 부족한 핵심 기능 중 하나였기 때문에 팀에 이러한 경험을 추가하게 되어 당신은 기쁠 것이다. 더 좋은 소식은 데이브의 전 동료가 객체 지향 기술에 대한 경험이 많았기 때문에 합리적인 설계와 가독성 높은 소스 코드를 기대할 수 있다는 것이다. 마지막으로 버그 리포트가 거의 제출되지 않았고 대부분의 버그가 빠르게 처리되었다는 이야기도 들었다. 마찬가지로 보류 중인 제품 개선 사항 목록이 존재하며 그 수가 상당히 적다. 따라서 고객들이 제품에 상당히 만족하고 있으며 이 프로젝트가 전략적으로 중요할 것이라고 결론을 내린다.

좋지 않은 소식은 데이브가 주로 동료들에게 무시당하고 나머지 시스템의 설계 활동에서 제외된 하드코어 C 프로그래머라는 것이다. 프로젝트에 남게 된 동기에 대해 물었을 때 그는 원래 인터넷 기술을 실험하는 데 관심이 있어서 합류했지만, 지금까지 해온 저수준 프로토콜 작업이 지루해져서 더 흥미로운 일을 하고 싶다고 말한다. 물론 ‘더 흥미로운 일’이 무슨 뜻인지 물어보니 그는 객체로 프로그래밍하고 싶다고 대답한다.

토론이 끝난 후 데이브가 작성한 코드의 품질을 평가하기 위해 소스 코드를 확인하기로 마음먹었다. 또한 보류 중인 버그 목록과 개선 요청을 살펴보고 병합해야 할 두 제품의 기능을 비교하고자 한다. 마지막으로 교육 부서에 연락하여 객체 지향 프로그래밍에 대한 강좌가 있는지 알아보는 것도 새 팀원에게 동기를 부여할 수 있는 방법이 될 수 있으므로 고려한다.

근거

“우리 업무의 주요 문제는 본질적으로 기술적인 것이 아니라 사회학적 문제이다.”

— 톰 디마르코, [DL99]

소프트웨어 프로젝트와 관련된 사회학적 문제가 기술적인 문제보다 훨씬 더 중요하다는 전제를 받아들인다면, 리엔지니어링 프로젝트 참여자들은 최소한 연구 대상 시스템의 정치적 맥락을 알아야 합니다.

“시스템을 설계하는 조직은 해당 조직의 커뮤니케이션 구조를 복사한 디자인을 만드는 제약을 가진다.”

— Melvin Conway, [Con68]

콘웨이의 법칙 (*Conway's law*)은 종종 다음과 같이 의역됩니다: “컴파일러를 개발하는 그룹이 4개라면, 4패스 컴파일러(4-pass compiler)³를 얻을 수 있다”

개발팀이 어떤 방식으로 구성되었는지 아는 것이 중요한 이유 중 하나는 이 구조가 소스 코드의 구조를 어느 정도 반영할 가능성이 높기 때문이다.

두 번째 이유는 리엔지니어링 프로젝트 계획을 수립하기 전에 팀원의 역량과 리버스 엔지니어링할 소프트웨어 시스템의 특성을 알아야 하기 때문이다. 유지보수자와의 토론은 이러한 지식을 얻는 방법 중 하나이며, “시간은 부족하다(*time is scarce*)”는 원칙을 고려할 때 매우 효율적인 방법이다.

“유지 보수 관련 사실 #1. 60년대 후반과 70년대 내내 생산 시스템 지원 및 유지보수는 분명히 2급 업무(second-class work)로 취급되었다.”

· 유지보수 관련 사실 #2. 1998년에도 생산 시스템 지원 및 유지보수는 계속해서 2급 업무로 취급되었다.”

— 롬 톰센, [Tho98]

유지보수자와 이야기를 나누다 보면 소프트웨어 유지 관리가 2급 업무로 간주되는 경우가 많다는 사실을 알게 된다. 만약 여러분이 대화하는 유지보수

³4번의 분리된 단계를 거쳐야만 결과가 나오는 컴파일러-옮긴이

팀이 그런 경우라면 논의가 심각하게 방해받을 수 있다. 유지보수 팀이 자주 바뀌었기 때문일 수도 있고, 이 경우 유지보수자가 그간의 발전 과정을 잘 모르기 때문일 수도 있다. 또는 논의하는 사람들이 자신의 업무에 대해 매우 보호적인 태도를 취하기 때문에 여러분이 알아야 할 내용을 알려주지 않을 수도 있다.

알려진 용도

리엔지니어링 프로젝트를 진행하면서 유지보수 팀과의 회의를 하는 중에 프로젝트의 킥오프(kick-off)하는 습관을 만들었다. 돌이켜보면 이러한 회의가 나머지 프로젝트에 필요한 신뢰를 구축하는 데 얼마나 중요한지 알 수 있었다. 유자보수자는 자신의 직업에 대한 자부심이 강하고 비판에 매우 민감하다는 사실을 뼈저리게 깨달았다. 따라서 킥오프 미팅은 “유자보수자 중심(maintainer oriented)”으로, 즉 유자보수자가 무엇을 잘하고 무엇을 더 잘하고 싶은지 보여줄 수 있도록 해야 한다는 점을 강조합니다. 초보인 내가 이 명청한 관리자들에게 제대로 된 일을 하는 방법을 가르쳐주겠다는 태도로 참여하면 거의 틀림없이 재앙을 초래할 것입니다.

“RT-100 —은 1980년대 후반 타사 소프트웨어 공급업체가 개발하여 1990년에 노텔(Nortel)에 인수되었다. 이후 3년 동안 노텔은 이 시스템을 개선하고 유지보수한 후 다른 공급업체에 아웃소싱하여 체계적으로 재작성했다. 이 노력은 실패로 돌아갔고 시스템은 1994년 중반에 다시 노텔로 돌아왔다. 이 무렵에는 원래 설계 팀이 해체되어 흩어져 있었고 제품의 6개 고객 조직은 상당히 불만이 많았다. RT-100은 노텔의 애틀랜타 테크놀로지 파크 연구소에 배정되었다. 그곳에는 ACD 소프트웨어에 대한 경험이 있는 직원이 없었고, 다른 프로젝트의 취소로 인해 직원들의 사기가 상당히 저하되어 있었다.”

— 스펜서 루가버, 짐 화이트, [RW98]

위의 인용문은 한 리엔지니어링 프로젝트의 사례를 설명하는 논문에서 인용한 것으로, 리엔지니어링 프로젝트가 시작될 때의 전형적인 절박함을 잘 묘사하고 있다. 하지만 — 논문 자체에 설명되어 있듯이 — 역사적, 정치적 맥락에 대한 이러한 초기 평가 덕분에 프로젝트가 성공할 수 있었던 것은 어떤 요소가

이해관계자들을 만족시키고 결과적으로 새로운 리엔지니어링 팀에게 동기를 부여할 수 있는지 잘 알고 있었기 때문이다.

DESEL 프로젝트 (시스템 진화의 용이성을 위한 설계)의 사례 연구 중 하나에서 스테芬 쿡은 도메인의 어떤 측면이 변경될 가능성이 있고 어떤 측면이 안정적으로 유지될 가능성이 있는지 가장 잘 알고 있는 유지보수자와 대화하는 것이 중요하다고 말한다 [CHR01]. 따라서 유자보수자는 시스템이 어떻게 구축될 수 있었는지에 대한 지식, 즉 거의 문서화되지 않은 지식을 가지고 있다. 그러나 이 논의 과정에서 “진화를 위한 설계(design for evolution)”라는 사고방식을 강조하여 유자보수자가 자신이 해결해 온 최신의 여러 문제에서 벗어나도록 해야 한다.

관련 패턴

소프트웨어 개발 팀의 조직 방식을 명시적으로 다루는 몇 가지 패턴 언어가 있다 [Cop95] [Har96] [Tay00] [BDS⁺00]. 앞으로의 엔지니어링 상황을 위한 것이지만, 유지보수자와 논의할 때 상황을 더 빨리 평가하는 데 도움이 될 수 있으므로 이를 숙지하는 것이 좋다.

다음 단계

토론하는 동안 선불리 결론을 내리지 않도록 주의해야 한다. 따라서 토론을 통해 알게 된 내용은 다른 출처를 통해 확인해야 한다. 일반적으로 이러한 출처는 시스템을 사용하는 사람들(데모 중 인터뷰하기), 문서(문서 스키밍하기), 시스템 자체(예: 한 시간 안에 모든 코드 읽기) 등이다. 모의 설치 수행하기).

이 검증을 통해 프로젝트를 완전히 취소할 가능성을 포함하여 레거시 시스템 문제를 해결하기 위한 초기 계획을 세울 수 있는 확실한 근거를 마련할 수 있다. 유자보수자와의 논의는 다양한 방식으로 이 계획에 영향을 미친다. 우선, 유지보수 팀의 협력 의지를 파악할 수 있어 작업 계획에 상당한 영향을 미친다. 둘째, 시스템을 가치 있게 만드는 부분과 대부분의 유지보수 문제를 일으킨 이벤트를 포함하여 시스템의 역사를 알고 있다. 당신 회사의 계획은 가치 있는 부분을 되살리고 이러한 유지보수 문제를 해결하는 것을 목표로 할 것이다.셋째, 유지보수 팀이 다른 이해관계자들과 어떻게 소통하는지에 대해 알게 되며,

이는 계획이 받아들여지는 데 중요하다.

3.2 한 시간 안에 모든 코드 읽기

의도 간단하지만 집중적인 코드 검토를 통해 소프트웨어 시스템의 상태를 평가한다.

문제

소스 코드의 품질에 대한 첫인상을 어떻게 얻을 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 소스 코드의 품질은 시스템의 개발 및 유지보수에 참여한 사람에 따라 상당히 달라질 수 있다.
- 시스템 규모가 커서 정확한 평가를 위해 검사해야 할 데이터가 너무 많다.
- 소프트웨어 시스템에 익숙하지 않아서 관련성이 있는 것을 필터링하는 방법을 모른다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같습니다.

- 여러분이 구현에 사용 중인 언어에 대한 합리적인 전문성(expertise)이 있으므로 프로그래밍 관용구(programming idioms) 및 코드 스멜(code smells)을 인식할 수 있다.
- 여러분의 리엔지니어링 프로젝트에 명확한 목표가 있으므로 그 목표를 달성하는데 필요한 코드 품질을 평가할 수 있습니다.

해결

소스 코드를 읽을 수 있는 적당히 짧은 시간(예: 약 1시간)의 학습 시간을 스스로에게 부여하자. 방해받지 않도록 하고(전화기를 뽑고 이메일 연결을 끊고), 코드와 최대한 접촉할 수 있도록 메모를 아껴서 하자.

이 읽기 세션이 끝나면 다음과 같은 내용을 포함한 간단한 보고서를 작성하자.

- 리엔지니어링이 실현 가능한지 여부와 그 이유(혹은 안 되는 이유)에 대한 일반적인 평가 항목
- 중요해 보이는 엔티티(예: 클래스, 패키지, …)
- 파악된 의심스러운 코딩 스타일(예: “코드 스멜” [FBB⁺99]);
- 추가 조사가 필요한 부분(예: 테스트).

이 보고서는 짧게 작성하고 소스 코드에 언급된 대로 엔티티의 이름을 언급 한다.

힌트

“시간은 부족하다(time is scarce)”는 원칙에 따라 약간의 준비가 필요하다. 체크리스트를 작성하면 읽기 세션에서 집중하는 데 도움이 될 수 있다. 이러한 체크리스트는 다양한 출처에서 가져와 작성할 수 있다.

- 개발 팀에서 품질 보증의 일환으로 코드 리뷰 (*code review*)를 사용했을 수 있다. 사용했다면 리뷰에 사용된 체크리스트를 포함해야 한다. 그렇지 않다면 해당 코드의 종류를 검토하는 데 사용되는 일반적인 체크리스트를 사용해 보자.
- 일부 개발팀에서는 코딩 스타일(*coding style*)을 적용하기도 하는데, 적용 했다면 이를 알아두는 것이 좋다. 특히 명명 규칙(*naming convention*)은 코드를 빠르게 스캔하는 데 매우 중요하다.
- 프로그래머는 코딩 이디엄 (*coding idiom*)(예: C++ [Cop92] [Mey98] [Mey96]; Smalltalk [Bec97])를 사용하여 일반적인 언어 구성을 인식할 수 있다.
- 답변을 듣고 싶은 질문(*question*)이 있을 수 있다.

다음은 추가 검토를 위한 좋은 시작점을 제공하기 때문에 체크리스트에 추가할 수 있는 몇 가지 항목이다.

- 기능 테스트(*functional test*) 및 단위 테스트 (*unit test*)는 소프트웨어 시스템의 기능에 대한 중요한 정보를 전달한다. 이러한 테스트는 시스템이

예상대로 작동하는지 확인하는 데 도움이 되며, 이는 리엔지니어링 중에 매우 중요하다(테스트라는 생명 보험 참조).

- 추상 클래스 및 메서드는 설계 의도를 드러낸다.
- 계층 구조에서 상위 클래스는 종종 도메인 추상화를 정의하고, 하위 클래스는 이에 대한 변형을 도입하도록 돋는다.
- 싱글톤 [p. 351] 패턴의 발생은 시스템의 전체 실행에 대해 일정한 정보를 나타낼 수 있다.
- 놀랍게도 대형 구조체는 종종 중요한 기능 덩어리를 지정한다.
- 주석(*comment*)은 특정 코드의 설계 의도에 대해 많은 것을 알려주지만, 종종 오해의 소지가 있을 수 있다.

트레이드오프

장점

- 효율적으로 시작하자. 짧은 시간 안에 코드를 읽는 것은 초보자로서 매우 효율적이다. 실제로 시간을 제한하면서도 모든 코드를 살펴보도록 강요하면 두뇌와 코딩 전문 지식을 주로 사용하여 중요해 보이는 부분을 걸려 낼 수 있다.
- 성실하게 판단하자. 코드를 직접 읽으면 소프트웨어 시스템에 대한 편견 없는 시각을 갖게 되고, 세부 사항에 대한 감각과 직면하고 있는 문제의 종류를 엿볼 수 있다. 소스 코드는 시스템의 기능을 설명하기 때문에 그 이상도 이하도 아닌 유일한 정확한 정보의 원천이다.
- 개발자 어휘를 익히자. 소프트웨어 시스템 내부에서 사용되는 어휘를 습득하는 것은 시스템을 이해하고 다른 개발자와 소통하는 데 필수적이다. 이 패턴은 이러한 어휘를 습득하는 데 도움이 된다.

단점

- 낮은 수준의 추상화만 획득가능하다. 이 패턴을 통해 솔루션 도메인에 대한 인사이트를 어느 정도 얻을 수 있지만, 이것이 문제 도메인 개념에

어떻게 매핑되는지에 대해서는 거의 알 수 없다. 따라서 더 추상적인 다른 표현으로 얻은 정보를 보완해야 한다(예: 문서 스키밍하기 및 데모 중 인터뷰하기).

어려움

- 큰 규모를 다루기 어렵다. 모두를 읽는 방법은 크기가 큰 코드를 다루기 어렵게 하며, 경험상 시간당 10,000줄의 코드가 적당하다. 크고 복잡한 코드를 마주할 때는 짧은 시간(2시간 이내)에 집중적으로 읽는 것이 가장 효과적이므로 더 많은 시간을 할애하여 코드를 읽으려고 하지 않는다. 대신 소스 코드를 분할할 명확한 기준이 있다면 일련의 세션을 나눠서 진행하자. 그렇지 않다면 모든 코드를 살펴보고 다른 부분보다 더 중요해 보이는 부분을 표시한 다음 유지보수자와 담소나누기에 따라 다른 세션에서 읽으면 된다.

그러나 “시간은 부족하다(Time is Scarce)”는 원칙을 고려할 때 간결하게 작성해야 한다. 따라서 크고 복잡한 코드를 다룰 때는 세부 사항에 너무 신경 쓰지 말고 리엔지니어링 적합성에 대한 초기 평가라는 코드 읽기의 목표를 상기하자.

- 주석이 오해를 불러일으킬 수 있다. 코드의 주석에 주의를 기울이자. 코드 멘트는 소프트웨어의 기능을 이해하는 데 도움이 될 수 있다. 하지만 다른 종류의 문서와 마찬가지로 주석도 오래되었거나 더 이상 사용되지 않거나 단순히 잘못된 것일 수 있다. 따라서 주석을 찾을 때는 체크리스트에 해당 주석이 도움이 되는지, 오래되었는지 여부를 표시하자.

예시

데이브(원래 개발팀에서 유일하게 남은 사람으로 저수준 C 코드를 담당했던 사람)와의 토론에서, 여러분은 그들의 시스템이 주로 Java로 작성되었고 일부 저수준 부분은 C로, 데이터베이스 쿼리는 SQL로 작성되었다는 것을 기억할 것이다. 이 모든 언어에 대한 경험이 있으므로 코드를 읽을 수 있다.

먼저 체크리스트를 작성하고 일반적인 항목(코딩 스타일, 테스트, 추상 클래스 및 메서드, 계층 구조에서 상위 클래스, ...) 외에 해결하고 싶은 몇 가지 질문

에 관한 몇 가지 항목을 추가한다. 그 중 하나는 “C 코드의 가독성(readability of the C-code)”인데, 새 팀원인 데이브의 코딩 스타일을 확인하고 싶기 때문이다. 두 번째는 “데이터베이스 스키마의 품질(quality of the database schema)”인데, 조만간 두 시스템의 데이터를 통합해야 한다는 것을 알고 있기 때문이다. 세 번째는 “통화 처리(handling of currencies)”이다. 스위스가 유로 지역에 가입하고 6개월 이내에 모든 금융 데이터를 이 새로운 통화로 변환해야 하기 때문이다.

C 코드를 읽다 보면 이 부분이 상당히 난해하다는 것을 알 수 있다(수수께끼 같은 암어로 된 짧은 식별자, 긴 다중 출구 루프, …). 그럼에도 불구하고 인터넷 프로토콜을 처리하는 모듈에는 단위 테스트가 있으므로 시스템에 통합할 가능성에 대해 더 확신을 가질 수 있다.

Java 코드는 한 시간에 50.000줄의 코드를 읽을 수 없다는 규모의 문제가 있다. 따라서 무작위로 몇 개의 파일을 선택하면 대부분의 클래스 이름에 UI 또는 DB라는 두 글자 접두사가 붙는 것을 발견할 수 있다. 2 계층 아키텍처(2-tiered architecture)(데이터베이스 계층과 사용자 인터페이스 계층)를 나타내는 명명 규칙을 의심하고 이를 자세히 조사하기 위해 메모를 작성한다. 또한 다양한 클래스 및 속성 이름이 의료 도메인에 의미 있는 것으로 인식한다(예: 이름, 주소, 건강 보험, … 속성을 가진 클래스 DBPatient). 심지어 DBCurrency 클래스를 인식하므로 개발자가 필요한 예방 조치를 취했기 때문에 유로로 전환해도 큰 문제가 발생하지 않을 것이라고 가정한다. 대부분의 클래스와 메서드에는 Javadoc 규칙을 따르는 주석이 있으므로 적어도 일부 문서는 최신 상태일 것으로 예상한다. 마지막으로, 화면에 표시되는 다양한 문자열을 포함하는 대형 싱글톤 객체를 식별하여 시스템을 지역화할 수 있을 것이라는 결론을 내린다.

이 모든 것이 다소 희망적으로 보이지만 실망스러운 부분도 많이 있다. 가장 비관적인 이유는 매개변수 목록이 많고 조건문이 복잡한 긴 메서드가 많다는 점이다. 이들 중 상당수는 UI 로직(버튼과 메뉴 항목의 활성화/비활성화)과 비즈니스 로직(데이터베이스 레코드 업데이트)이 혼합되어 있는 것 같다. 한 가지 (가격 계산)가 특히 복잡해 보이므로 이를 더 조사하기 위해 메모를 작성한다.

데이터베이스와 관련해서는 의료 서비스 도메인의 맥락에서 의미 있는 다양한 테이블 이름과 열 이름을 다시 인식한다. 언뜻 보기에는 스키마가 정규화된 것처럼 보이므로 여기에서도 리버스 엔지니어링이 흑허거 있을 것으로 보인다. 이 데이터베이스는 또한 일부 저장 프로시저를 사용하므로 추가 조사가 필요

하다.

읽기 세션이 끝나면 다음 노트에 결론을 요약한다.

- 인터넷 프로토콜 통합이 가능함: 단위 테스트와 담당 프로그래머가 있다.
- 명명 규칙에 따라 2계층 아키텍처가 의심됨: 비즈니스 로직은 어떤가? — UI와 혼합되어 있는가? (추가 확인 필요!)
- 의미 있는 식별자가 포함되어 있는 읽을 수 있는 코드: 리버스 엔지니어링의 효과가 기대된다.
- 통화 개체 존재: 유로 변환이 가능해 보인다(추가 조사!).
- 사용된 자바독(Javadoc) 규칙 사용: 문서 확인이 필요하다.
- 가격 계산이 복잡해 보임: 그 이유는 무엇인가?
- 데이터베이스 스키마가 유용해 보임: 저장 절차는 추가 조사가 필요합니다.

근거

코드 리뷰(coder review)는 동료가 작성한 프로그램에서 문제를 찾는 데 매우 효과적인 수단으로 널리 인정받고 있다 [GG93] [Gla97]. 이러한 검토를 비용 효율적으로 수행하기 위해서는 두 가지 중요한 전제 조건이 충족되어야 합니다: (a) 검토자가 관련 질문에 집중할 수 있도록 체크리스트(checklist)를 준비해야 하고, (b) 리뷰어는 장시간(최대 2시간) 집중할 수 없으므로 리뷰 세션은 짧게 유지해야 한다.

나는 속독 강좌를 들었고 '전쟁과 평화'를 20분 만에 읽었다. 러시아에 관한 책이다.

— 우디 앤런

기존의 코드 리뷰와 소프트웨어 시스템을 처음 접할 때 수행하는 코드 리뷰에는 중요한 차이점이 있다. 전자는 일반적으로 오류를 감지하기 위한 것이고, 후자는 첫인상을 얻기 위한 것이다. 이 차이는 세부 사항에 덜 신경을 써야 하므로 더 많은 코드를 읽을 수 있다는 것을 의미한다. 코드 리뷰에 대한 일반적인 가이드라인에서는 시간당 약 150개의 문을 검토할 수 있다고 명시하고 있다

[BP94]. 그러나 첫 번째 접촉에서는 이러한 상세한 분석이 필요하지 않으므로 검토할 코드의 양을 늘릴 수 있다. 심각한 경험적 조사를 수행하지는 않았지만 경험상 시간당 10,000줄의 코드가 적당해 보인다.

알려진 용도

이 패턴은 켄트 베이 제안한 것으로, 그는 기존 시스템에서 컨설턴트 업무를 시작할 때 항상 적용하는 기법 중 하나라고 말했다. 롭슨은 코드 읽기를 “시스템에 대한 지식을 얻는 가장 조잡한 방법(crudest method)”으로 보고하며, 기존 프로그램을 이해하는 데 가장 일반적으로 사용되는 방법이라고 인정한다.^{91a} 일부 사례 연구 보고서에서도 소스 코드 읽기가 리엔지니어링 프로젝트를 시작하는 방법 중 하나라고 언급하고 있다 [BH95] [JC00].

이 패턴을 작성하는 동안 우리 팀원 중 한 명이 이 패턴을 적용하여 리팩토링 브라우저 (Refactoring Browser)를 리버스 엔지니어링했다. [RB97]. 그 사람은 Smalltalk에 익숙하지 않았지만 클래스 인터페이스를 검사하는 것만으로도 시스템 구조에 대한 느낌을 얻을 수 있었다. 또한 특수 계층 구조 브라우저는 주요 클래스 중 일부를 식별하는 데 도움이 되었고 주석은 코드의 어떤 부분을 수행해야 하는지에 대한 유용한 힌트를 제공했다. 패턴을 적용하는 데는 1시간이 조금 넘게 걸렸는데, 비교적 작은 시스템과 Smalltalk에 익숙하지 않아서 진행 속도가 느려 보였기 때문에 이 정도면 충분해 보였다.

이 패턴의 특히 흥미로운 사례는 FAMOOS 프로젝트가 끝날 무렵에 발생했다. 일주일 동안 이질적인 리버스 엔지니어들로 구성된 팀이 일종의 리버스 엔지니어링 경연 대회에 참가하기 위해 현장 방문을 떠났다. 4일 동안 주어진 리버스 엔지니어링 도구를 사용하여 특정 C++ 시스템에 대해 최대한 많은 것을 배우는 것이 과제였다. 그런 다음 5일째에는 원래 개발자에게 결과를 보고하여 검증을 받는 것이었다. 팀원 중 한 명이 과제를 너무 일찍 끝내고 한 시간 안에 모든 코드 읽기에 도전하는 기회를 가졌다. 이 한 사람은 모든 토론에 참여할 수 있고 개발자의 의견 중 일부를 설명할 수 있을 정도로 시스템에 대해 훨씬 더 잘 파악하고 있었습니다.

다음 단계

한 시간 안에 모든 코드 읽기 를 완료한 후에는 리엔지니어링에 적합한지 평가하기 위해 모의 설치 수행하기 을 수행해야 한다. 또한 문서 스키밍하기 를 수행하여 결과를 보완하고, 시스템을 일관성 있게 파악할 수 있는 기회를 극대화하기 위해 데모 중 인터뷰하기 를 수행하면 된다. 실제로 리엔지니어링 프로젝트를 어떻게 진행할지 결정하기 전에 유지보수자와 담소나누기 를 한 번 더 수행하는 것이 좋다.

시스템과의 첫 접촉이 끝나면 프로젝트를 어떻게 진행할지(또는 취소할지) 결정해야 한다. 코드를 읽으면 다양한 방식으로 이 결정에 영향을 미친다. 우선, 코드의 품질(예: 코딩 이디엄 및 의심스러운 코딩 스타일의 존재 여부)을 평가하여 프로젝트 리엔지니어링의 타당성을 평가했다. 둘째, 추가 탐색을 위한 좋은 출발점이 될 수 있는 몇 가지 중요한 엔티티를 식별했다.

중요한 엔티티(예: 클래스, 패키지, 한 시간 안에 모든 코드 읽기 의 결과인 클래스, 패키지, ...) 목록을 사용하여 퍼시스턴트 데이터 분석하기 [p. 97] 및 예외적인 엔티티 연구하기 [p. 119] 를 시작할 수 있다. 이렇게 하면 소스 코드, 특히 문제 도메인을 나타내는 방식에 대해 구체적으로 이해할 수 있다.

3.3 문서 스키밍하기

의도 제한된 시간 내에 문서 (*documentation*)를 읽고 관련성을 평가한다.

문제

문서에서 도움이 될 수 있는 부분을 어떻게 식별할 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 문서가 있는 경우 개발팀이나 최종 사용자를 위한 것이므로 일반적으로 리엔지니어링 목적과 즉각적인 관련이 없다. 더 좋지 않은 것은 일반적으로 현재 상황과 관련하여 최신 정보가 아니므로 오해의 소지가 있는 정보가 포함되어 있을 수 있다는 것이다.
- 리엔지니어링 프로젝트가 어떻게 진행될지 아직 알 수 없으므로 문서의 어느 부분이 관련성이 있는지 알 수 없다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 어떤 형태의 문서(*documentation*)를 사용할 수 있으므로 적어도 시스템과 관련된 사람을 돋기 위한 설명이 있다.
- 여러분의 리엔지니어링 프로젝트에는 명확한 목표(*clear goal*)가 있으므로 문서에서 가치 있는 부분과 쓸모없는 부분을 선택할 수 있다.

해결

리엔지니어링 프로젝트에 흥미로워 보이는 시스템 측면을 요약한 목록을 준비하자. 그런 다음, 이 목록을 문서와 대조하면서 문서가 얼마나 최신 상태인지 대략적으로 평가하자. 마지막으로 다음과 같이 간단한 보고서로 결과를 요약하자.

- 시스템 문서가 유용한지 여부와 그 이유 (혹은 그렇지 않은 이유)에 대한 일반적인 평가를 적는다.

- 문서에서 유용하다고 생각되는 부분의 목록과 그 이유(예: 요구 사항 사양, 원하는 기능, 중요한 제약 조건, 설계 다이어그램, 사용자 및 운영자 설명서)를 작성한다.
- 각 부분에 대해 설명이 얼마나 최신 상태인지에 대한 인상을 작성한다.

힌트

리엔지니어링 프로젝트의 목표와 사용 중인 문서의 종류에 따라 주요 관심사에 맞게 읽기 프로세스를 조정할 수 있다. 예를 들어 원래 시스템 요구 사항에 대한 인사이트를 얻고 싶다면 시스템 사양을 살펴보고, 실제로 구현된 기능에 대한 지식은 최종 사용자 설명서나 튜토리얼 노트에서 수집해야 한다. 여유가 있다면 설계 문서(예: 클래스 다이어그램, 데이터베이스 스키마, …)를 이해하려고 너무 많은 시간을 소비하지 말고, 리엔지니어링의 후반 단계에서 큰 도움이 될 수 있으므로 이러한 문서의 존재 여부와 신뢰성을 기록해 두자.

실제 시스템과 관련하여 문서가 오래되지는 않았는지 확인하자. 항상 시스템 납품 날짜와 버전 날짜를 비교하고 신뢰할 수 있는 것으로 의심되는 부분을 기록해 두자.

시간이 제한되어 있다는 사실 때문에 가장 유용한 정보를 추출할 수 있는 방법을 생각해야 한다. 다음은 주의해야 할 사항에 대한 몇 가지 힌트이다.

- 목차(*table of contents*)는 구조와 제시된 정보에 대한 간략한 개요를 제공한다.
- 버전 정보 및 날짜(*Version numbers and dates*)는 문서의 해당 부분이 얼마나 최신 상태인지 알려준다.
- 그림(*figure*)은 정보를 전달하기 위한 좋은 수단이다. 그림 목록이 있는 경우 문서의 특정 부분에 대한 빠른 액세스 경로를 제공할 수 있다.
- 화면 덤프(*screen-dump*), 샘플 출력물(*print-out*), 샘플 리포트(*report*), 명령어 설명(*command description*)은 시스템에서 제공하는 기능에 대해 많 은 것을 알려준다.
- 정식 사양(*formal specifications*)(예: 상태 차트(*state-chart*))이 있는 경우 일반적으로 중요한 기능에 해당한다.

- 색인(index)가 있는 경우에는 작성자가 중요하다고 생각하는 용어가 포함되어 있습니다.

트레이드오프

장점

- 높은 추상화 수준을 제공한다. 문서는 사람이 읽어야 하므로 일정 수준의 추상화가 필요하다. 이 추상화 수준이 리엔지니어링 프로젝트에 충분히 높지 않을 수도 있지만 적어도 몇 가지 디코딩 단계를 건너뛸 수 있다.
- 관련 부분에 집중한 흥미로워 보이는 부분의 목록을 미리 준비하면 읽기 세션이 목표 지향적이 되어 가치 있는 것을 찾을 확률이 높아진다. 또한 설명이 얼마나 최신인지 빠르게 평가할 수 있어 관련 없는 부분에 시간을 낭비하지 않아도 된다.

단점

- 중요한 사실 누락한다. 개요 모드에서 빠르게 읽으면 문서에 기록된 중요한 사실을 놓칠 수 있다. 하지만 찾고자 하는 내용을 목록으로 작성하면 이러한 효과를 어느 정도 줄일 수 있다.
- 관련 없는 정보만 찾을 수 있다. 문서의 어느 부분도 리엔지니어링 프로젝트와 관련이 없는 것처럼 보일 가능성이 적다. 그런 상황에서도 이제 문서에 대해 걱정하지 않아도 되니 읽는데 들인 시간은 가치가 있다.

어려움

- 다른 청중을 대상으로 한다. 문서는 제작 비용이 많이 들기 때문에 최종 사용자(예: 사용자 매뉴얼) 또는 개발팀(예: 디자인)을 위해 작성된다. 또한 문서를 유지 관리하는 데도 많은 비용이 들기 때문에 시스템의 안정적인 부분만 문서화한다. 따라서 찾은 정보는 직접적인 관련이 없을 수 있으므로 신중한 해석이 필요하다.
- 문서에 불일치가 있다. 문서는 실제 상황과 관련하여 거의 항상 최신 정보가 아니다. 리엔지니어링 프로젝트의 초기 단계에서는 이러한 불일치를

인식할 수 있는 지식이 부족하기 때문에 이는 매우 위험하다. 따라서 문서에만 의존하여 중요한 결정을 내리지 말고, 먼저 다른 방법으로 결과를 확인하기 바란다, 한 시간 안에 모든 코드 읽기 및 데모 중 인터뷰하기).

예시

데이브와의 비공식적인 대화와 코드 읽기 세션이 끝나면 시스템의 흥미로운 측면이 무엇인지 대략적으로 파악할 수 있다. 관련 정보가 포함되어 있는지 확인하기 위해 문서를 훑어보기로 결정한다.

읽고 싶은 측면의 목록을 작성하여 스스로 준비한다. 디자인 다이어그램, 클래스 인터페이스 설명(자바독?), 데이터베이스 스키마와 같은 당연한 항목 외에도 유로(사용자 설명서에 유로 변환에 대한 내용이 있는가?), 인터넷 프로토콜 사양 등이 목록에 포함되어 있다.

다음으로 데이브에게 가서 소프트웨어 시스템과 관련된 모든 문서를 요청한다. 데이브는 작은 미소를 지으며 여러분을 바라본다. “정말 다 읽어보실 건 아니죠?” “꼭 그렇지는 않아요.” “하지만 적어도 이걸로 뭔가 할 수 있는지는 알고 싶어요.”라고 말한다. 데이브가 아까 준 상자를 살펴보고 종이로 가득 찬 폴더 3개 — 설계 문서 —와 소책자 1개 — 사용 설명서를 건네준다.

사용 설명서부터 시작해서 — 빙고: 색인에서 유로에 대한 항목을 발견한다. 해당 페이지를 넘기면 유로가 실제로 약 5페이지로 구성된 하나의 챕터라는 것을 알 수 있으므로 해당 페이지 번호를 표시하여 추가 학습을 진행한다. 다음으로 목차를 훑어보면 ‘프랑스어/독일어로 전환하기’라는 제목이 있다. 이 페이지를 읽으면 소프트웨어 현지화가 문서화된 기능이라는 것을 알 수 있다. 현지화는 체크리스트에 없었지만 여전히 중요하므로 기꺼이 이에 대한 메모를 추가한다. 이 모든 것이 상당히 유망해 보이므로 사용자 설명서의 릴리스 날짜를 확인하면 꽤 최근의 것임을 알 수 있다. 정말 좋은 시작이다!

디자인 문서의 첫 번째 폴더(‘클래스라는 제목의 폴더)를 열면 예상했던 것과 거의 비슷하게 Javadoc에서 생성된 클래스 인터페이스의 인쇄물을 찾을 수 있다. 종이로 읽는 것은 그다지 흥미롭지 않지만 어쨌든 페이지를 계속 훑어보게 된다. 첫인상은 각 클래스와 메서드와 함께 제공되는 실제 설명이 상당히 얇다는 것이다. 무작위로 세 페이지를 더 자세히 살펴보면 이러한 인상은 더욱 확고해진다. 다음으로 인터넷 프로토콜을 구현하는 C 코드와 인터페이스하는

클래스에 대한 설명을 찾아보면 빈 설명도 발견할 수 있다. 문서의 릴리스 날짜를 리트머스 테스트한 결과 이 문서가 상당히 오래되었다는 것을 알았으므로 온라인 문서를 확인하기 위해 메모를 작성한다.

두 번째 폴더에는 데이터베이스 스키마에 대한 생성된 설명이 들어 있는데, 각 테이블에 대해 각 열의 목적이 무엇인지 설명한다. 자바독 클래스 인터페이스 설명과 마찬가지로 문서 자체는 내용이 적어도 데이터베이스의 각 레코드가 무엇을 나타내야 하는지를 찾을 수 있는 방법이 있다. 여기에서도 문서 릴리스 날짜가 확인하니 동일한 문서의 온라인 버전을 확인하는 것이 필요하다는 것도 알 수 있다.

언뜻 보기에도 세 번째 폴더에는 프로젝트와 막연하게 관련만 있어 보이는 다양한 잡다한 문서 사본이 들어 있는 것처럼 보인다. 첫 번째 문서는 의약품 가격 목록이고, 다음 열 개는 의료법에서 발췌한 것이다. 하지만 계속해서 페이지를 넘기다 보면 건강보험과 통신하는 데 사용되는 인터넷 프로토콜을 설명하는 것으로 보이는 유한 상태 다이어그램(finite state diagram)을 발견하게 된다. 분명히 이 문서는 기술 사양의 일부 페이지를 복사한 것이지만 안타깝게도 원본에 대한 참조는 포함되어 있지 않다. 이 문서의 릴리스 날짜도 누락되어 있어 이 사양이 오래된 것인지 확인할 수 있는 방법이 없다.

다음 보고서를 작성하고 읽기 세션을 마무리한다.

- 사용자 설명서가 명확하고 최신 상태임: 기능에 대한 블랙박스 타입의 설명을 위한 좋은 참고처이다.
- 유로 화폐 관련 정보가 제공됨(513-518쪽): 현지화도 제공된다(723-725쪽).
- 클래스 인터페이스 설명이 생성되어 있음: 정보가 적지만 온라인에서 확인 가능하다.
- 데이터베이스 스키마에 대한 문서가 생성됨: 간략하지만 온라인에서 확인 가능하다.
- 인터넷 프로토콜에 대한 유한 상태 머신: 상태 의심스럽다. 테이브에게에게 확인하도록 한다.
- 기타 문서(가격표, 지침 전단지, ...)가 들어 있는 폴더 1개

근거

“소프트웨어 개발 조직에서 전체 소프트웨어 개발 노력의 20~30%를 문서화에 사용하는 것은 드문 일이 아니다.”

— 로저 프레스맨, [Pre94]

소스코드와 달리 문서는 소프트웨어 시스템을 인간에게 적합한 추상화 수준에서 설명하기 위한 것이다. 따라서 문서에는 분명히 정보 “덩어리(nugget)”가 포함될 것이며, 문제는 관련 정보를 찾는 방법이다. 거의 모든 리엔지니어링 프로젝트에 존재하는 두 가지 일반적인 상황 때문에 관련 정보를 찾는 것이 어렵다.

“모든 사례 연구에서 문서가 존재하지 않거나 불만족스럽거나 일관성이 없다는 문제에 직면한다.”

— ESEC/FSE 1997 Workshop on Object-Oriented Re-engineering, [DG97]

우선, 문서가 실제 상황과 일치하지 않을 가능성이 높다. indFAMOOS 프로젝트에서 조사한 5개의 사례 연구에서 모든 관리자가 불만을 제기한 유일한 문제는 ‘불충분한 문서화’였다. 그럼에도 불구하고 오래된 정보라도 적어도 과거에 시스템이 어떻게 작동해야 했는지 알려주기 때문에 유용할 수 있다. 오늘 날 어떻게 사용되는지 유추할 수 있는 좋은 출발점이다.

“이러한 시스템에 대해 존재하는 문서는 일반적으로 전체 아키텍처가 아닌 개별적인 부분에 대해 설명한다. 게다가 문서는 시스템 전체와 여러 미디어에 흩어져 있는 경우가 많다.”

캐니 윙, et al., [WTMS95]

둘째, 문서는 일반적으로 포워드 엔지니어링 컨텍스트에서 생성되므로 리엔지니어링 목적이 아니다. 예를 들어, 생성된 설계 문서(예: 데이터베이스 스키마, 자바독)는 일반적으로 상당히 최신이지만 너무 세분화되어 있어 리엔지니어링 프로젝트의 초기 단계에서는 유용하지 않다. 사용 설명서는 소프트웨어 시스템에 대한 블랙박스 타입의 설명이므로 그 안에 무엇이 들어 있는지에 대한 청사진 역할을 할 수 없다. 여기에서도 설명서를 실제로 관심 있는 부분을 유추할 수 있는 좋은 출발점으로 삼아야 한다.

알려진 용도

피엘드슈타트와 햄렌의 연구에 따르면 '유지보수 프로그래머는 기능을 개선할 때 문서를 공부하는 시간보다 원래 프로그램을 공부하는 시간이 약 3.5배 더 길지만, 개선 기능을 구현하는 데 소요되는 시간은 그만큼 더 길다'고 합니다. [Cor89], [FH79]. 이 등식은 문서 연구의 상대적 중요성에 대한 설명으로 받아들일 수 있다.

"사례 연구는 CTAS의 기존 설계 전반과 특히 CM을 이해하려는 노력에서 시작되었다. — CTAS에 대한 문서에는 동기 부여 및 아키텍처 개요, 소프트웨어 구조, 사용자 설명서, 기본 알고리즘에 대한 연구 논문이 포함되어 있다. 그러나 시스템이 무엇을 계산하는지 또는 환경에 대해 어떤 가정을 하는지에 대해 개괄적으로 설명하는 문서는 없는 것으로 보인다. 또한 CTAS 구성 요소 간의 관계(통신 방식, 제공하는 서비스 등)를 설명하는 설계 문서도 없다. 우리는 코드에서 이러한 정보를 유추할 수밖에 없었는데, 이는 많은 상업 개발 노력에 공통적으로 나타나는 문제이다."

— 다니엘 잭슨, 존 채핀, [JC00]

위의 인용문은 문서를 공부할 필요가 있다는 것을 잘 요약하고 있지만, 문서가 알아야 할 모든 것을 여러분에게 알려주지는 않는다. 그들이 언급한 사례 연구는 항공 교통 관제 시스템(CTAS)에 관한 것으로, 약 80개의 KLOC C++ 코드로 구성된 핵심 구성 요소 *CommunicationsManager(CM)*를 리버스 엔지니어링한 것이다.

다음 일화는 문서가 어떻게 오해를 불러일으킬 수 있는지를 보여준다. ind-FAMOOS 사례 연구 중 하나에서 약 12개의 하위 시스템을 연결하는 분산 시스템을 약 100개의 하위 시스템을 연결하도록 확장할 수 있는지 평가해 달라는 요청을 받았다. 이 평가에서 우리는 모든 TCP/IP 연결을 유지 관리하는 클래스를 연구했는데, 이 클래스에서는 모든 열린 연결이 일종의 루프 테이블에서 어떻게 유지되는지 설명하는 코멘트가 있었다. 코드에서 루프 테이블을 찾았지만, 그 작동 방식에 대한 설명을 테이블을 조작하는 연산에 매핑할 수는 없었다. 반나절 동안 고민하다가 포기하고 관리자에게 물어보기로 했다. 그의 대답은 "아, 하지만 이 클래스 주석은 더 이상 사용되지 않아요. 이제야 말씀하시니 그

클래스를 다시 디자인할 때 삭제했어야 했는데요."라는 답변이었다.

다음 단계

특정 결과를 확인하기 위해 한 시간 안에 모든 코드 읽기 바로 뒤에 문서 스 키밍하기 을 추가할 수 있습니다. 또한 유지보수자와 대화나누기 및 테모 중 인터뷰하기 를 통해 특정 의혹을 확인하는 것도 좋다.

시스템과의 첫 접촉이 끝나면 프로젝트를 어떻게 진행할지 (또는 취소할지) 결정해야 한다. 관련 문서를 발견하면 적어도 이 정보를 재현할 필요는 없다는 것을 알 수 있다. 더 좋은 점은 관련성이 있지만 부정확해 보이는 문서의 경우 추가 탐색을 위한 좋은 출발점이 있다는 것이다(예: 퍼시스턴트 데이터 분석하기 [p. 97] 및 디자인에 대해 추측하기 [p. 109]).

3.4 데모 중 인터뷰하기

의도 데모를 보고 데모를 제공하는 사람과 인터뷰하여 소프트웨어 시스템의 기능에 대한 첫인상을 구한다.

문제

소프트웨어 시스템의 일반적인 사용 시나리오와 주요 기능에 대한 아이디어를 어떻게 얻을 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 일반적인 사용 시나리오는 사용자 유형에 따라 매우 다양하다.
- 사용자에게 물어보면 무엇이 잘못되었는지 불평하는 경향이 있는 반면, 리버스 엔지니어링의 경우 주로 무엇이 가치 있는지에 관심이 있다.
- 시스템 규모가 커서 정확한 평가를 위해 검사해야 할 데이터가 너무 많다.
- 소프트웨어 시스템에 익숙하지 않아서 관련성이 있는 것을 필터링하는 방법을 모른다.

하지만 다음과 같은 이유로 이 문제를 해결할 수 있다.

- 작동 중인 시스템과 소프트웨어 시스템 사용 방법을 보여줄 수 있는 사용자가 있다는 점을 활용할 수 있다.

해결

데모를 보고 시연하는 사람을 인터뷰하여 작동 중인 시스템을 관찰한다. 인터뷰 부분은 적어도 데모만큼이나 유익하다는 점에 유의하자.

이 데모가 끝나면 같은 정도의 시간을 할애하여 발견한 내용을 포함한 보고서를 작성하자.

- 몇 가지 일반적인 사용 시나리오.
- 시스템에서 제공하는 주요 기능 및 그 기능에 대한 평가.
- 시스템 컴포넌트와 컴포넌트의 책임.

- 시스템 사용과 관련된 숨겨진 이야기를 보여주는 기이한 사례.

힌트

데모를 제공하는 사용자는 이 패턴의 결과에 결정적인 역할을 하므로 사람을 선택할 때 주의하자. 따라서 데모를 제공하는 사람을 여러 번 바꿔서 데모를 여러 번 수행하자. 이렇게 하면 사람들이 중요하다고 생각하는 요소의 차이를 확인할 수 있고 소프트웨어 시스템의 가치에 대한 다양한 의견을 들을 수 있다. 열렬한 지지자나 열렬한 반대자를 항상 경계하자. 그들이 확실히 관련 정보를 제공하겠지만, 편견을 피하기 위해 보완적인 의견을 찾기 위해 추가 시간을 할애해야 한다.

다음은 어떤 사람을 찾아야 하는지, 어떤 종류의 정보를 기대할 수 있는지, 어떤 종류의 질문을 해야 하는지에 관한 몇 가지 힌트이다. 물론 어떤 사람과 이야기해야 하는지는 리엔지니어링 프로젝트의 목표와 프로젝트와 관련된 조직의 종류에 따라 크게 달라지므로 이 목록은 시작점으로만 제공된다.

- **최종 사용자(end-user)**는 시스템이 외부에서 어떻게 보이는지 알려주고 일상적인 업무 상황에 기반한 몇 가지 세부적인 사용 시나리오를 설명해야 한다. 소프트웨어 시스템이 도입되기 전의 업무 관행에 대해 질문하여 비즈니스 프로세스 내에서 소프트웨어 시스템의 범위를 평가한다.
- **관리자(manager)**는 시스템이 나머지 비즈니스 도메인 내에서 어떻게 적합한지 알려주어야 한다. 시스템과 관련된 비즈니스 프로세스에 대해 질문하여 리엔지니어링 프로젝트와 관련된 무언의 동기가 있는지 확인한다. 리엔지니어링은 그 자체가 목표가 아니라 다른 목표를 달성하기 위한 수단일 뿐이므로 이는 매우 중요하다.
- **영업 부서(sales department)**의 담당자는 귀사의 소프트웨어 시스템을 경쟁 시스템과 비교해야 한다. 사용자가 가장 많이 요청하는 기능의 데모를 요청하고(이것이 반드시 가장 높이 평가하는 것과 같을 필요는 없다!) 과거에 어떻게 발전해 왔으며 앞으로 어떻게 발전할 수 있는지 물어보자. 이 기회를 통해 존재하는 다양한 유형의 최종 사용자와 소프트웨어 시스템의 발전 방향에 대한 인사이트를 얻을 수 있다.
- **헬프 테스크(help desk)**의 담당자가 대부분의 문제를 일으키는 기능을 시

연해야 한다. 데모의 이 부분에서 실제 비즈니스 관행과 소프트웨어 시스템에서 모델링한 방식이 일치하지 않을 수 있으므로 사용자에게 어떻게 설명하는지 물어보자. 소프트웨어 시스템과 관련된 숨겨진 이야기에 대한 느낌을 얻기 위해 기이한 사례를 공개하도록 유도하자.

- 시스템 관리자(*system administrator*)는 소프트웨어 시스템의 이면에서 일어나는 모든 일(예: 시작 및 종료, 백업 절차, 데이터 보관, ·)을 보여주어야 한다. 시스템의 신뢰성을 평가하기 위해 과거의 끔찍한 사례를 요청하자.
- 유지보수자/개발자(*maintainer/developer*)가 하위 시스템 중 일부를 보여줄 수 있다. 이 하위 시스템이 다른 하위 시스템과 어떻게 통신하는지, 왜(그리고 누가!) 그렇게 설계했는지 물어보자. 이 기회를 통해 시스템의 아키텍처와 설계에 영향을 미친 장단점에 대한 인사이트를 얻을 수 있다.

변형

자신에게 데모하기. 인터뷰 중 데모하기의 축소된 변형은 리버스 엔지니어가 시행착오 과정을 통해 시스템을 직접 시연하는 것이다. 이러한 데모는 분명히 데모를 강화하는 그룹 역학(group dynamics)이 부족하지만 다른 한편으로는 설계자/유지보수자와의 토론을 위한 준비 기법으로 사용될 수 있다.

트레이드오프

장점

- 중요한 기능에 중점을 둔다. 데모를 제공한다는 사실은 인터뷰 대상자에게 가치 있는 기능을 시연하도록 부드럽지만 강하게 요구할 것이다. 리버스 엔지니어의 주된 관심사는 당연히 그 기능이다.
- 정성적 데이터를 많이 제공한다. 인터뷰를 진행하면 일반적으로 다른 방법으로는 추출하기 어려운 풍부한 관련 정보를 얻을 수 있다.
- 신뢰도를 높인다. 인터뷰를 수행하면 인터뷰 대상자에게 해당 시스템에 대한 자신의 의견에 진정으로 관심이 있다는 것을 보여줄 수 있다. 따라서

인터뷰는 리엔지니어링 프로젝트의 결과에 대한 최종 사용자의 신뢰를 높일 수 있는 특별한 기회를 제공한다.

단점

- 일부 사례 증거만 제공한다. 얻는 정보는 기껏해야 유지보수자와 담소나 누기 와 마찬가지로 일부 사례 정보에 불과하다. 인터뷰 대상자는 잊어버렸거나 흥미롭지 않다고 판단하여 중요한 사실을 생략할 가능성이 거의 높다. 이러한 효과는 시연을 통해 어느 정도 상쇄할 수 있지만, 다른 방법으로 얻은 정보를 보완할 준비를 하자(예: 문서 스키밍하기 참조), 한 시간 안에 모든 코드 읽기 및 모의 설치 수행하기 참조).
- 시간이 부족할 수 있다. 최소 한 명 이상이 데모를 수행할 수 있어야 한다. 이는 간단한 요구 사항처럼 보이지만 실제로는 달성하기 어려울 수 있다. 일부 시스템(예: 임베디드 시스템)은 사람이 직접 시연할 수 없으며, “시간은 부족하다(time is scarce)”는 원칙에 따라 시스템을 시연할 수 있는 사람과 약속을 잡는 데 시간이 너무 오래 걸리는 경우도 있다.

어려움

- 인터뷰 경험이 필요하다. 질문의 표현 방식은 인터뷰 결과에 상당한 영향을 미친다. 안타깝게도 모든 리버스 엔지니어가 좋은 인터뷰를 진행하는데 필요한 기술을 갖추고 있는 것은 아니다. 경험이 없는 경우 데모의 흐름에 따라 적절한 질문을 유도하자.
- 인터뷰 대상자 선정이 어려울 수 있다. 열렬한 지지자나 열렬한 반대자를 인터뷰하는 것은 피해야 한다. 안타깝게도 리엔지니어링 프로젝트 초기에는 인터뷰 대상자를 잘 선정할 수 있는 지식이 부족하다. 따라서 다른 사람의 의견에 의존하여 선정하되, 인터뷰 대상자의 열정(또는 열정 부족)에 따라 결과를 조정할 준비를 하자.
- 실시간 소프트웨어를 다루어야 할 수 있다. 특정 종류의 시스템(특히 실시간 시스템)의 경우 소프트웨어 시스템을 조작하면서 질문에 답할 수 없는 경우가 있다. 이러한 상황에서는 데모를 보면서 질문을 적어두고 나중에 실제 인터뷰를 진행하자.

예시

이제 소스 코드와 문서를 확인했으니 *XDoctor* 시스템을 리엔지니어링하는 것이 가능하다는 것을 거의 확신할 수 있을 것이다. 하지만 사용자가 시스템에서 어떤 점을 높이 평가하는지 잘 모르기 때문에 정확히 무엇을 리버스 엔지니어링해야 하는지 여전히 의문이 든다. 영업 부서를 통해 현재 사용자 중 한 명과 연락을 취하고 다음 날 약속을 잡는다. 또한 인터넷 프로토콜의 상태(문서에서 찾은 상태 차트 사양 포함)와 나머지 시스템과의 적합성이 걱정되어 데이브에게 가서 인터넷 프로토콜의 데모를 보여줄 수 있는지 물어본다.

데이브는 기꺼이 자신의 작업을 보여주며 즉시 키보드로 타이핑을 시작한다. “보세요, 이제 서버를 시작했어요.” 그가 화면에 나타난 작은 콘솔 창을 가리키며 말한다. “잠깐만요. 무슨 명령을 입력했죠?”라고 대답한다. “LSVR; 알 다시피, Launch Server의 약자죠”. 조금 놀라서 당신은 데이브에게 이 서버를 시작하고 종료하는 방법을 설명하는 매뉴얼이 있는지 물어본다. 데이브는 그런 것은 없지만 전체 시스템을 시작하는 배치 파일에서 유추하는 것은 꽤 쉽다고 설명한다. 심지어 LSVR과 관련된 몇 가지 명령줄 옵션이 있으며, 이 옵션들은 모두 READ.ME 파일과 -h(elp) 옵션을 통해 문서화되어 있다고 알려준다. 다음으로 데이브는 테스트 프로그램을 시작하고(예, LSVRTST를 통해 호출됨) 콘솔 창에서 서버가 실제로 트래픽을 수신하는 동안 테스트 프로그램이 송수신된 모든 메시지의 긴 로그를 뺏어내는 것을 볼 수 있다. 물론 테스트가 성공했는지 어떻게 알 수 있는지 물어보지만, 실망스럽게도 그는 로그를 수동으로 검사하면 된다고 말한다. 주제를 바꿔서 이 하위 시스템이 실제로 클라이언트 시스템에서 실행되고 있다고 생각하기 때문에 이 하위 시스템을 서버라고 부르는 이유를 물어보기로 한다. 이 질문은 열띤 토론을 불러일으키고 결국 그림 6에 표시된 것과 같은 아키텍처 다이어그램으로 이어져 원격 서버(건강 보험에서 관리하고 수락), 로컬 서버(LSVR의 L은 아마도 “실행(Launch)”이 아니라 “로컬(Local)”을 의미할 것이다) 및 일부 로컬 클라이언트를 보여준다. 이 설명을 통해 전체 시스템이 어떻게 작동하는지 어느 정도 이해할 수 있다. 기본 개념은 LAN 네트워크를 통해 로컬 서버에 연결된 여러 책상에 여러 대의 클라이언트 컴퓨터가 있다는 것이다. 로컬 서버는 데이터베이스와 건강 보험에 대한 인터넷 연결을 유지 관리한다. 작은 종이에 그려진 다이어그램을 들고 데이브에게 이 인터넷 프로토콜이 어디에서 시작되었는지 물어본다. 이 질문은 이 프로토콜이 독일에서 설계되었고(따라서 국가 차트와 함께 문서화되어 있는 이유) 현재

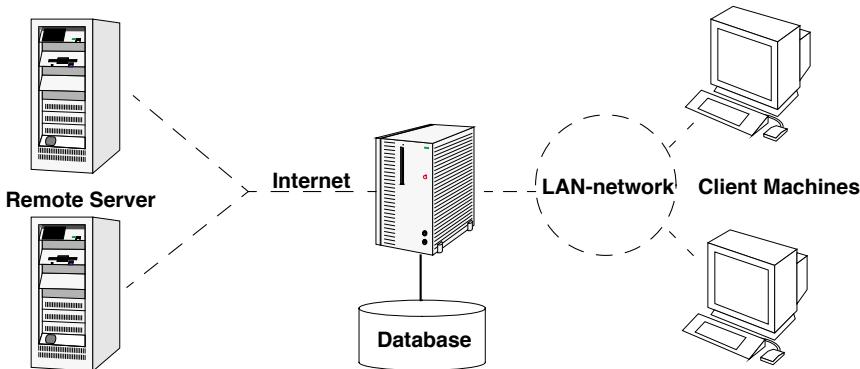


그림 3.2: 유지보수자와의 토론에서 유추한 아키텍처 다이어그램

전국 의료 보험 회사에서 채택하고 있다는 긴 이야기의 시작이 되었다.

다음 날, 양복을 입고 의사인 메리 요한센과의 미팅을 하기 위해 차를 타고 출발한다. 자신을 소개하는 동안 그녀는 별로 반갑지 않다는 인상을 받는다. 방문 이유를 설명하고 대화하는 동안 의사가 회사가 *XDoctor* 소프트웨어를 인수하는 것에 대해 상당히 걱정하고 있다는 것을 알게 된다. 데모와 인터뷰의 주된 목적은 귀사가 현재 사용자에게 가장 잘 서비스를 제공할 수 있는 방법을 알아보는 것이며, 지원을 중단할 의도는 없다는 것을 그녀에게 확신시키기 위해 최선을 다한다. 안심한 그녀는 실제 데모를 시작했다. 당연히 가장 마음에 드는 기능은 “서류 작업을 할 비서를 줄일 수 있다는 점”을 의미하는 건강보험 자동 거래 처리 기능이다. 하지만 요한센 박사는 내장 이메일, 스프레드시트로 내보내기(“이 파일을 회계사에게 이메일로 보내기만 하면 된다”), 여러 통화로 결제(“유로로 처리하면 정말 좋다”) 등 미처 몰랐던 다른 기능도 보여줬다. 데모를 진행하는 동안 그녀는 처음에는 시스템이 약간 불안정했고(베타 테스터 역할을 한 것으로 보인다), 몇 가지 이상한 실수(환자 목록이 성이 아닌 이름으로 정렬됨)가 있었지만 전반적으로 시스템에 매우 만족하고 있다고 말한다.

사무실로 돌아와서 로컬 서버를 테스트하기 위한 명령 순서와 자동 거래 처리 및 여러 통화로 결제하기 위한 사용 시나리오가 포함된 간단한 보고서를 작성한다. 보고서에는 아키텍처 다이어그램(그림 6)과 다음과 같은 관찰 사항도 포함된다.

- 인터넷 프로토콜 테스트는 수동으로 수행하여 회귀 테스트(regression test)에 대해 조사한다.
- 인터넷 프로토콜 사양은 독일 의료 보험 컨소시엄에서 제공받았다.
- 환자 목록 정렬해보고, 성 대신 이름 기준으로 정렬되는 것을 알았다.

근거

“인터뷰 대상자의 답변에 유연하게 대응할 수 있는 능력은 인터뷰가 널리 사용되는 이유 중 하나이다.”

— 사이몬 베넷, *et al.*, [BMF99]

“인터뷰는 인터뷰 수행자가 상황에 맞게 인터뷰를 조정할 수 있기 때문에 아직 무엇을 찾고 있는지 모르는 탐색적 연구에 적합하다”

— 제이콥 넬슨, [Nie99]

소프트웨어 시스템을 사용하는 사람들을 인터뷰하는 것은 중요한 기능과 일반적인 사용 시나리오를 파악하는 데 필수적이다. 하지만 리엔지니어링 초기 단계에서는 무엇을 물어봐야 할지 모르기 때문에 미리 정의된 질문을 하는 것은 효과적이지 않다. 단순히 사람들이 시스템에 대해 어떤 점을 좋아하는지 물어보면 모호하거나 의미 없는 답변이 나올 수 있다. 게다가 사용자들은 레거시 시스템에 대해 불평하는 경향이 있기 때문에 매우 부정적인 답변을 얻을 위험이 있다.

“분석의 진정한 도전은 전문가가 다른 사람, 즉 분석가에게 개념을 전달해야 할 때 시작된다. 개념은 매우 풍부하고 광범위한 경우가 많기 때문에 일반적으로 전문가가 자신의 이해 전체를 하나의 총체적인 표현으로 적절하게 전달하는 것은 불가능하다.”

— 아델 골드버그, 캐니 루빈, [GR95]

포워드 엔지니어링 상황과 비교할 때 리버스 엔지니어는 작동 중인 소프트웨어 시스템이 있고 이를 활용할 수 있다는 한 가지 큰 장점이 있다. 이러한 상황에서는 데모를 요청하여 사용자에게 주도권을 넘기는 것이 안전하다. 우선, 데모를 통해 사용자는 자신의 말로 이야기를 전달할 수 있지만 데모는 일종의

유형적 구조를 부과하기 때문에 이해할 수 있다. 둘째, 사용자는 작동하는 시스템에서 시작해야 하기 때문에 무엇이 작동하는지 보다 궁정적인 태도로 설명할 수 있다. 마지막으로, 데모를 진행하는 동안 면접관은 정확한 질문을 많이 하고 정확한 답변을 얻을 수 있으므로 시스템 사용법에 대한 전문 지식을 파악할 수 있다.

알려진 용도

이 패턴의 주요 아이디어인 사용자가 시스템을 사용하는 동안 시스템을 설명하게 하는 것은 일반적으로 사용자 인터페이스를 평가하는 데 사용된다. “소리 내어 생각하기(Thinking aloud)는 가장 가치 있는 사용성 엔지니어링(usability engineering) 방법일 수 있다. 기본적으로 소리 내어 생각하기 테스트는 피험자가 지속적으로 큰 소리로 생각하면서 시스템을 사용하도록 하는 것이다.” [Nie99] 요구사항 도출(requirements elicitation)을 위한 신속한 프로토타입 제작에도 동일한 아이디어가 종종 적용된다 [Som96].

FAMOOS 프로젝트 초기의 한 일화에서는 이 패턴의 변형인 자신에게 직접 시연하기를 적용하여 소프트웨어 시스템이 작동하는 것을 보고 무지한 질문이 어떻게 유지보수 팀 내에서 잡자고 있는 전문성을 축발할 수 있는지를 보여준다. 사례 연구 중 하나인 데이터베이스 계층, 도메인 객체 계층, 사용자 인터페이스 계층으로 구성된 3계층 시스템(3-tiered system)의 전형적인 예로 ‘비즈니스 객체를 가져와야 한다’는 요청을 받았다. 두 명의 개인에게 이 작업을 맡겼는데, 한 사람은 소스 코드 브라우저와 CASE 도구를 사용하여 해당 비즈니스 객체를 나타내는 클래스 다이어그램을 추출했다. 다른 한 명은 로컬 PC에 시스템을 설치하고 약 한 시간 동안 사용자 인터페이스를 가지고 놀면서(즉, 시스템을 직접 시연하면서) 자신이 관찰한 몇 가지 이상한 점에 대한 10가지 질문 목록을 작성했다. 그 후, 시스템의 수석 분석가-설계자와 시스템을 리버스 엔지니어링 하려고 시도한 두 사람과 함께 회의가 진행되었다. 분석가-설계자는 클래스 다이어그램을 보고 이것이 실제로 비즈니스 객체라는 것을 확인했지만, 뭔가 빠진 것이 있는지, 설계의 근거가 무엇인지에 대해서는 말해주지 않았다. 우리가 10 가지 질문을 던지고 나서야 그는 설계 과정에서 직면한 문제에 대해 매우 열정적이고 매우 상세한 설명을 시작했고, 심지어 이야기 도중 클래스 다이어그램을 가리키기도 했다! 분석가 겸 디자이너의 이야기를 듣고 나서 소스코드에서 클

래스 다이어그램을 추출한 사람의 첫 반응은 '이런, 소스코드에서 그런 걸 읽은 적이 없다'는 것이었다.

관련 패턴

최종 사용자와 상호 작용하는 방법에 관한 많은 좋은 조언이 “고객 상호 작용 패턴(Customer Interaction Patterns)” [Ris00]에 구체화되어 있다. 이 패턴의 주요 메시지는 “판매가 아닌 관계(It's a Relationship, Not a Sale)”라는 것으로, 최종 사용자와의 접점에서 신뢰 관계를 구축하는 것을 목표로 해야 한다는 점을 강조한다.

다음 단계

최적의 결과를 얻으려면 여러 종류의 사람들과 데모 중 인터뷰하기를 여러 번 시도해야 한다. 취향에 따라 이러한 시도를 한 시간 안에 모든 코드 읽기 및 문서 스키밍하기의 전, 후 또는 혼합하여 수행할 수 있다. 그 후에는 유지보수자와 담소나누기를 사용하여 일부 결과를 확인한다.

시스템과의 첫 접촉이 끝나면 프로젝트를 어떻게 진행할지 (또는 취소할지) 결정해야 한다. 데모를 보면 사람들이 시스템을 어떻게 사용하는지, 어떤 기능을 높이 평가하는지에 대한 느낌을 알 수 있다. 따라서 소프트웨어 시스템의 중요한 부분과 리버스 엔지니어링이 필요한 부분을 알 수 있다. 사용 시나리오는 디자인에 대해 추측 [p. 109] 및 비즈니스 규칙을 테스트로 기록 [p. 193]와 같은 패턴의 입력으로도 사용할 수 있다.

3.5 모의 설치 수행하기

의도 시스템을 설치하고 코드를 다시 컴파일하여 필요한 산출물을 사용할 수 있는지 확인한다.

문제

시스템을 (재)빌드할 수 있다는 것을 어떻게 확신할 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 여러분에게는 시스템이 새롭기 때문에 시스템을 빌드하는 데 필요한 파일을 모른다.
- 시스템이 라이브러리, 프레임워크, 패치에 의존할 수 있으며 올바른 버전을 사용할 수 있는지 확실하지 않다.
- 시스템이 크고 복잡하며 시스템이 실행되는 정확한 구성이 불분명하다.
- 유지보수자가 이러한 질문에 답하거나 설명서에서 답을 찾을 수 있지만 이 답변이 완전한지 확인해야 한다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 소스 코드(*source code*)와 필요한 빌드 도구(예: 메이크파일, 컴파일러, 링커)에 접근할 수 있다.
- 실행 중인 시스템과 유사한 환경(예: 설치 CD 및 올바른 운영 체제가 설치된 컴퓨터)에서 시스템을 재설치(*re-install*)할 수 있다.
- 시스템에 일종의 자체 테스트(*self test*)가 포함되어 있을 수 있다. (테스트라는 생명 보험 참조)를 사용하여 빌드 또는 설치가 성공했는지 확인할 수 있다.

솔루션

제한된 시간(최대 하루) 동안 깨끗한 환경에서 시스템을 설치 및 빌드해 보자. 시스템에 자체 테스트가 포함되어 있으면 실행한다.

힌트

설치 및 빌드 프로세스를 완전히 이해하는 것이 아니라 다시 수행할 수 있는지 확인하는 것이 핵심이다.

빌드 및 설치 프로세스 중에 발생하는 모든 작은 오류와 해결 방법을 기록하면 시스템 구성과 라이브러리, 프레임워크 및 패치에 대한 종속성에 대해 알 수 있으므로 이를 기록하자. 예를 들어 특정 위치에서 시스템을 컴파일할 수 없거나 특정 컴퓨터에서만 액세스할 수 있는 오래된 레거시 라이브러리가 필요하거나 특정 라이브러리 패치가 필요하다는 사실을 알게 될 수 있다.

결국 시스템을 완전히 빌드하거나 설치하는 데 성공하지 못했을 수도 있다. 이는 리엔지니어링 프로젝트에 큰 영향을 미칠 가능성이 높은 위험에 해당하므로 계속 진행하기 전에 빌드 및 설치 절차를 연구하고 필요한 경우 이를 조정할 계획을 세워야 한다.

이 빌드 및 설치 실험이 끝나면 다음 내용이 포함된 보고서를 준비하자.

- 사용된 라이브러리, 프레임워크 및 패치의 버전(*version number*)
- 인프라(데이터베이스, 네트워크 툴킷, 포트, ...) 간의 종속성(*dependency*).
- 발생한 문제(*problem*)와 해결 방법.
- 개선(*improvement*)에 대한 제안 사항.
- (불완전한 설치 또는 빌드의 경우) 솔루션 및 해결 방법의 가능성을 포함한 상황에 대한 평가(*assessment*).

트레이드오프

장점

- 필수 전제 조건. 시스템을 (재)구축하거나 (재)설치할 수 있는 능력은 리엔지니어링 프로젝트에 필수적이므로 이 문제를 초기에 평가해야 한다. 구축 또는 설치가 어렵거나 불가능한 것으로 판명되면 필요한 수정 조치를 계획하자.

- 정확성 요구. 빌드 및 설치 프로세스를 복제하면 필요한 구성 요소에 대해 정확하게 파악해야 한다. 특히 마이그레이션 프로젝트의 경우 모든 구성 요소를 대상 플랫폼에서도 사용할 수 있어야 하므로 이 정보는 매우 중요하다.
- 신뢰성을 높이기. 빌드 또는 설치 후에는 어려운 단계를 직접 경험하게 된다. 개선에 대한 구체적인 제안을 쉽게 할 수 있어야 유지 관리 팀에 대한 신뢰도가 높아질 것이다.

단점

- 지루한 활동. 특히 대부분의 문제가 지금은 관심이 없는 사소한 세부 사항에 달려 있기 때문에 시스템 설치 실패의 원인을 추적하는 동안 매우 비생산적으로 느껴질 것이다. 모의 설치 수행하기에 할애하는 시간을 제한하면 이 효과를 어느 정도 줄일 수 있지만, 그러면 시스템 구축이나 설치에 성공하지 못했기 때문에 더욱 비생산적으로 느껴질 것이다.
- 불확실성. 이 패턴은 정밀성을 요구하지만 일부 구성 요소를 재설계한 후 실제로 시스템 구축에 성공할 수 있다는 보장은 없다. 특히 신뢰할 수 있는 자체 테스트가 누락되면 빌드 또는 설치가 완료되었는지 확인할 수 없다.

어려움

- 고민하지 않고 계속 시도하기. 복잡한 시스템을 구축하거나 설치할 때 외부 요인(누락된 구성 요소, 불명확한 설치 스크립트)으로 인해 쉽게 실패할 수 있다. “다음번에는 되겠지”라는 생각에 이런 성가신 문제를 계속 수정하고 싶은 유혹에 빠지기 쉽다. 이러한 세부 사항에 얹매이기보다는 시스템을 구축하는 것이 아니라 구축 프로세스에 대한 통찰력을 얻는다는 주요 목표를 놓치지 않는 것이 중요하다. 따라서 시간을 제한하고 발생하는 문제를 문서화하는 데 집중하여 나중에 문제를 해결할 수 있도록 해야 한다.

예시

일부 최종 사용자와 테모 중 인터뷰를 수행했으며, 그 결과 리엔지니어링 프로젝트에서 유지해야 할 중요한 기능에 대한 감을 잡았다. 하지만 프로젝트를 수락하기 전에 시스템을 변경할 수 있는지 여부를 확인해야 한다. 따라서 시스템을 새로 빌드할 수 있는지 확인하기 위해 간단한 실험을 해보기로 결정한다.

데이브가 사무실에 두고 간 상자에서 모든 소스 코드가 들어 있는 두 번째 CD를 가져온다. 디렉토리를 살펴보다가 최상위 메이크파일 하나를 발견하고 시도해 보기로 결정한다. 모든 파일을 시스템의 Linux 파티션에 복사하고 프롬프트에 `make all` 명령을 입력한다. 잠시 동안 모든 것이 순조롭게 진행되며 시스템은 수많은 자바 컴파일 성공을 보고한다. 하지만 안타깝게도 몇 분 후 `java.sql` 라이브러리가 누락되어 `make`가 실패한다. JDK1.1이 설치되어 있다는 사실을 깨닫게 되지만, 문서에서 JDK1.3이 설치되어야 한다고 언급한 것을 기억한다. 마지막에 전체 디렉토리 구조를 휴지통에 버리고 JDK1.1을 제거한 다음 JDK1.3을 다운로드하여 설치한 다음(다운로드하는 데 시간이 오래 걸리므로 커피 한 잔을 가져와야 한다) 다시 시작한다. 이번에는 C 코드 컴파일이 시작될 때까지 메이크가 순조롭게 진행된다. 라이브러리 파일이 누락되어 첫 번째 컴파일이 즉시 실패하고 C 파일을 열어 이 실패의 원인이 정확히 무엇인지 확인합니다. `assert.h`는 시스템에서 사용할 수 있는 표준 라이브러리므로 검색 경로에 문제가 있는 것이 분명하다. 그때는 거의 점심 시간이었고 오늘 이 빌드 실험을 끝내기로 계획했기 때문에 전체 C 컴파일은 나중에 하기로 결정했다. 어쨌든 데이브가 팀에 같이 있고, 그가 이 C 코드를 작성했으니 컴파일하는 방법을 보여줄 수 있을 것이다.

점심 식사 후에는 작성한 코드가 정상인지 확인하고 싶을 것이다. "void main()"를 grep하면 `XDoctor.java` 파일에 메인 항목이 포함되어 있음을 알 수 있으므로 `java XDoctor`를 입력하여 시스템을 실행한다. 실제로 테모에서 보셨던 시작 화면이 나타나고 “시스템이 데이터베이스에 연결 중입니다”라는 작은 상태창이 나타난다. 그 직후 “예상치 못한 일이 발생했습니다” 메시지와 함께 시스템이 실패하고 데이터베이스 누락이 원인인 것으로 의심된다. 이 문제를 나중에 조사하기로 결정하고 설치 절차를 확인하기로 한다.

시스템을 설치할 수 있는지 확인하기 위해 설치 CD를 Macintosh의 CD 드라이브에 넣는다. 자동으로 일반적인 설치 창이 나타나고 설치 프로세스를

원활하게 진행한다. 설치 프로세스가 완료되면 설치 프로그램에서 시스템을 시작하기 전에 컴퓨터를 재부팅하라는 메시지가 표시된다. 어떤 시스템 확장 프로그램이 설치되었는지 확인하고 컴퓨터를 재부팅한 다음 바탕화면에 나타난 XDoctor 아이콘을 더블 클릭한다. 안타깝게도 라이센스 키를 입력하라는 창이 나타납니다. CD 상자를 살펴보니 라이센스 키는 별도의 편지로 받았어야 하는데 당연히 받지 못했다. “아쉽네, 라이선스 키가 제공되지 않았을 때 *proDoc*처럼 시스템의 데모 버전을 실행할 수 있으면 좋았을 텐데” 하는 생각이 든다. 좌절하여 포기하고 다음과 같은 보고서를 작성하기로 결정한다.

- JDK1.3으로 make가 작동하는 것으로 보임: 이 빌드가 완료되었는지 확인할 수 없다.
- C 컴파일 실패: 데이브에게 빌드 데모를 요청한다.
- 라이선스 더 자세히 조사: 시스템이 어떻게 보호되는가?
- 제안: 라이선스 키가 제공되지 않으면 데모 모드로 실행한다(참조: *proDoc*).
- 제안: 호출 시 사전 조건을 확인한다. *XDoctor.main()*; 새로 빌드한 후 “예 기치 않은 일이 발생했다”라는 메시지와 함께 시스템이 종료된다.

알려진 용도

FAMOOS 사례 연구 중 하나에서는 중앙 서버와 소켓을 통해 통신하는 분산 시스템을 간단한 명령어를 사용하여 리엔지니어링해야 했다. 첨부된 편지에 따르면 “필요한 모든 것이 들어 있는” tar 파일이 들어 있는 테이프를 받았다. 그러나 시스템을 재구축하고 다시 설치하는 것이 어려웠고, 설치 스크립트를 자세히 살펴보고 관리자에게 설명을 요청해야 했다. 결국 보안 및 연결 문제로 인해 중앙 서버와 통신할 수는 없었지만 시뮬레이션 모드에서 시스템을 테스트할 수 있었다. 실험이 완전히 성공하지는 못했지만 시스템 아키텍처에 대한 인사이트를 얻을 수 있었다. 특히 시뮬레이션 모드가 중앙 서버를 모방하는 방식과 이것이 소스 코드와 메이크파일에 인코딩되는 방식은 나머지 프로젝트 기간 동안 중요한 정보를 제공했다.

우리가 수행한 감사 프로젝트의 첫날이 끝날 무렵, 다음 날 아침 새로 설치 한 것을 보여 달라고 요청했다. 저희는 이를 데모 중 인터뷰하기를 위한 준비를

위한 순수한 요청이라고 생각했지만, 설치 도중 한 명의 관리자가 설치 CD를 준비하기 위해 밤을 새워야 했다는 사실을 알게 되었다. 그 후의 토론을 통해 우리는 이 시스템이 사용자 기반이 고정되어 있고 인터넷을 통해 매주 업데이트를 다운로드하도록 설계된 시스템이라는 사실을 알게 되었다. 이는 이전에 한 시간 안에 모든 코드 읽기 를 위해 노력하는 동안 관찰한 많은 특이한 점을 설명해 주었고, 남은 감사 프로젝트 동안 설계 문제를 드러내는 데 많은 도움이 되었다.

구성 관리 시스템(configuration management system)으로 작업할 때는 코드를 다시 컴파일하기 전에 먼저 코드를 깨끗한 구성으로 가져오는 것이 좋다. 예를 들어 Smalltalk 시스템의 경우, 시스템을 구성하는 Envy 구성 맵을 먼저 로드한 다음 코드를 깨끗한 이미지로 로드하는 것이 일반적인 조언 중 하나이다 [PK01].

다음 단계

결론을 보고하기 전에 유지보수자와 담소나누기 를 하는 것이 좋다. 유지보수자는 여러분의 조사 결과를 확인하고 오해를 풀 수 있을 것이다. 개선에 대한 구체적인 제안은 관리자와 논의하는 것이 가장 좋은데, 이는 여러분이 진정으로 그들을 돋고자 하는 의지가 있음을 설득할 수 있는 가장 좋은 방법이기 때문이다.

빌드 또는 설치가 완전히 실패하는 경우 데모 중 인터뷰하기 를 모의 설치 수행하기 와 결합하는 것이 좋다. 이 경우 유지보수자에게 빌드 또는 설치 프로세스를 시연해 달라고 요청하고 불분명한 단계에 대해 질문하자.

제 4 장

초기 이해

여러분의 회사는 의사들이 사용할 수 있는 *proDoc*이라는 의료 정보 시스템을 개발하여 제공하고 있다. 이제 회사는 다양한 의료 보험 회사와 거래를 수행하기 위해 인터넷 지원을 제공하는 경쟁 소프트웨어 *XDoctor* 제품을 인수했다. 두 제품을 하나의 시스템으로 통합해야 한다.

*XDoctor*를 처음 평가한 결과, 몇 가지 컴포넌트를 복구하여 통합해야 한다는 사실이 밝혀졌다. 물론 소프트웨어 컴포넌트를 성공적으로 복구하려면 내부 구조와 나머지 시스템과의 연결을 이해해야 한다. 예를 들어, 여러분의 회사는 고객에게 “단 1바이트의 데이터도 손실되지 않는다”고 약속했으므로 데이터베이스 내용을 복구하고 결과적으로 데이터베이스 구조와 상위 계층이 데이터베이스에 어떻게 의존하는지 이해해야 한다. 또한 귀사는 건강 보험 회사와의 거래 지원을 지속하고 확대하겠다고 약속했으므로 이러한 원격 서비스와 통신하는 데 사용되는 네트워크 통신 컴포넌트를 복구해야 한다.

포스: 주요한 요구사항

리엔지니어링 프로젝트에서 이와 유사한 상황이 자주 발생한다. 시스템 및 사용자와의 첫 번째 접근 [p. 43]을 통해 어떤 기능이 가치가 있고 왜 복구해야 하는지 명확하게 알 수 있다. 하지만 소프트웨어 시스템의 전반적인 설계에 대한 지식이 부족하기 때문에 이 기능을 레거시 시스템에서 제거할 수 있는지 여부와

이에 소요되는 비용을 예측할 수 없다. 이러한 초기 이해는 리엔지니어링 프로젝트의 성공을 위해 매우 중요하며, 이 장에서는 이를 얻는 방법을 설명한다.

첫 번째 접근의 패턴을 통해 소프트웨어 시스템에 대한 첫 번째 아이디어를 얻는 데 도움이 되었을 것이다. 이제 이러한 아이디어를 초기 이해로 구체화하고 추가 리버스 엔지니어링 활동을 지원하기 위해 이러한 이해를 문서화할 때이다. 리버스 엔지니어링의 이 단계에서 가장 중요한 우선 순위는 나머지 프로젝트의 안정적인 기반을 구축하는 것이므로 발견한 내용이 정확하고 적절하게 문서화되어 있는지 확인해야 한다.

발견한 내용을 올바르게 문서화하는 방법은 프로젝트의 범위와 팀의 규모에 따라 크게 달라진다. 10명 이상의 개발자가 참여하는 복잡한 리버스 엔지니어링 프로젝트에는 몇 가지 표준 문서 템플릿과 구성 관리 시스템이 필요하다. 반면에 3인 미만이 참여하는 평범한 프로젝트는 중앙 서버에서 공유되는 느슨한 구조의 파일로 충분히 관리할 수 있다. 하지만 모든 상황에 적용되는 몇 가지 내재적인 포스(force)이 있다.

- 데이터는 기만적이다. 기존 소프트웨어 시스템을 이해하려면 데이터를 수집하고 해석하여 일관된 시각으로 요약해야 한다. 일반적으로 데이터를 해석하는 방법에는 두 가지 이상의 방법이 있으며, 대안 중에서 선택할 때 항상 구체적인 증거가 뒷받침되지 않는 가정을 하게 된다. 따라서 정보의 출처를 다시 확인하여 탄탄한 토대 위에 이해를 구축해야 한다.
- 이해는 반복이 필요하다. 이해는 인간의 두뇌 내부에서 일어나므로 일종의 학습 과정에 해당하다. 리버스 엔지니어링 기술은 우리의 두뇌가 새로운 아이디어를 받아들이는 방식을 지원해야 하므로 매우 유연해야 하고 많은 반복과 역추적(backtracking)을 허용해야 한다. 따라서 학습 과정을 자극하기 위해 반복(iteration)과 피드백 루프(feedback loop)를 계획해야 한다.
- 지식을 공유해야 한다. 시스템을 이해했다면 이 지식을 동료들과 공유하는 것이 중요하다. 동료가 업무를 수행하는 데 도움이 될 뿐만 아니라 여러분의 이해도를 높일 수 있는 의견과 피드백도 얻을 수 있다. 따라서 벽에 지도를 붙이자. 발견한 내용을 잘 보이는 곳에 게시하고 피드백에 대한 명시적인 규정을 마련하자. 이를 수행하는 방법은 팀 조직과 업무 습관에 따라 달라진다. 일반적으로 팀 회의는 정보를 공유하는 좋은 방

법이지만(원탁 회의에 말하기 [p. 31] 참조), 커피 머신 근처의 벽에 큰 그림을 붙이는 것도 좋은 방법일 수 있다.

- 팀은 소통이 필요하다. 시스템에 대한 이해를 구축하고 문서화하는 것은 목표가 아니라 목표를 달성하기 위한 수단이다. 시스템을 이해하기 위한 진정한 목표는 프로젝트에 참여하는 다른 사람들과 효과적으로 소통하는 것이므로, 이해한 것을 문서화하는 방식은 그 목표를 뒷받침해야 한다. 예를 들어 동료가 엔터티 관계 다이어그램(Entity-Relationship diagram)을 읽을 줄만 안다면 UML 클래스 다이어그램을 그릴 필요가 없고, 최종 사용자가 그 범위를 이해할 수 없다면 사용 사례를 작성하는 것도 의미가 없다. 결국, 팀원이 사용하는 언어를 사용하자. 팀원들이 여러분이 문서화 한 내용을 읽고, 이해하고, 댓글을 달 수 있도록 여러분이 이해한 내용을 문서화할 언어를 선택하자.

개요

소프트웨어 시스템을 처음 이해할 때 가장 우려되는 것은 잘못된 정보이다. 따라서 이러한 패턴은 신뢰할 수 있는 유일한 정보 소스인 소스 코드에 주로 의존 한다.

원칙적으로 소스 코드를 연구하는 방법에는 하향식(top-down)과 상향식 (bottom-up)의 두 가지 접근 방식이 있다. 실제로 모든 리버스 엔지니어링 접근 방식은 이 두 가지를 조금씩 모두 사용해야 하지만, 그래도 구분할 가치가 있다. 하향식 접근 방식에서는 높은 수준의 표현에서 시작하여 소스 코드와 비교하여 검증한다(예: 디자인 추측하기의 설명 참조). 상향식 접근 방식에서는 소스 코드에서 시작하여 관련성이 있는 것을 필터링하고 관련 엔터티를 상위 수준 표현으로 캐스팅한다. 이 접근 방식은 퍼시스턴트 데이터 분석하기 및 예외적인 엔터티 연구하기에서 사용된다.

이러한 각 패턴을 적용하는 데 선호되는 순서는 없다. 먼저 퍼시스턴트 데이터 분석하기 을 수행한 다음, 디자인 추측하기 을 통해 결과 모델을 구체화하고 마지막으로 이 지식을 활용하여 예외적인 엔터티 연구하기 를 수행하는 것이 당연할 수 있다. 따라서 패턴은 이러한 순서로 표시된다. 그러나 시스템의 많은 부분이 데이터베이스와 관련이 없을 경우(일부 시스템에는 어떤 형태의 퍼시스턴트 데이터도 없다) 데이터베이스를 학습하지 않은 상태에서 디자인 추측하기

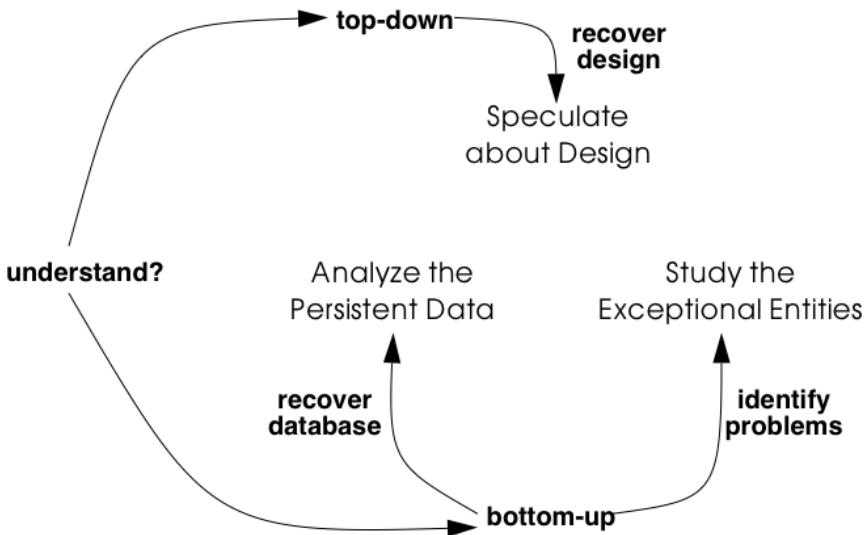


그림 4.1: 소프트웨어 시스템의 초기 이해를 구하고 이를 상위 수준의 표현으로 변환한다.

을 수행해야 한다. 그리고 디자인 추측하기 으로 시작하기 어려운 상황이라면 예외적인 엔티티 연구하기 가 초기 가설을 제시해 줄 것이다.

이러한 각 패턴에 할애해야 하는 시간은 리엔지니어링 프로젝트의 목표에 따라 크게 달라진다. 원칙적으로 이러한 패턴 중 어느 것도 오래 걸리지 않지만 각 패턴은 여러 번 적용해야 한다. 팀이 나머지 프로젝트를 진행할 만큼 충분히 이해했는지 여부는 패턴을 적용한 후에야 평가할 수 있기 때문에 얼마나 많은 주기가 필요할지 예측할 수 없다. 따라서 이러한 패턴은 사례별로 적용해야 한다.

다음 단계

여러분이 추가로 이해한 내용을 프로젝트 계획에 반영해야 한다. 예를 들어 퍼시스턴트 데이터 분석하기 및 디자인에 대한 추측 은 시스템의 일부를 문서화하며, 이 문서를 기회(Opportunity)로 사용해야 한다. 반면에 예외적인 엔티티 연구하기 를 사용하면 의심스러운 컴포넌트를 발견할 수 있으며, 이러한 컴포

넌트는 리스크(Risk)로 관리해야 한다.

여러분이 충분히 이해하여 충분한 기초를 마련했다면 나머지 프로젝트에 중요한 컴포넌트에 대한 세부 정보를 채워 넣어야 한다. 디테일 모델 캡처 [p. 129]에 설명된 활동이 이러한 세부 정보를 채우는 데 도움이 될 수 있다.

4.1 퍼시스턴트 데이터 분석하기

의도 데이터베이스 시스템 내부에 보관해야 할 정도로 중요한 개체에 대해 학습한다.

문제

Which object structures represent the valuable data ? 어떤 개체 구조가 중요한 데이터를 나타낼까요?

이 문제는 다음과 같은 이유로 어렵다.

- 귀중한 데이터는 외부 저장 장치(예: 파일 시스템, database)에 안전하게 보관해야 한다. 그러나 이러한 데이터 저장소는 종종 다락방(attic)¹ 역할을 하므로 거의 정리되지 않고 많은 정크를 포함할 수 있다.
- 메모리에 로드될 때 중요한 데이터는 복잡한 객체 구조로 표현된다. 안 타깝게도 외부 저장 장치에서 제공하는 데이터 구조와 주 메모리에 있는 객체 구조 사이에는 큰 차이가 있다. 예를 들어 상속 관계는 레거시 데이터베이스에서 명시적으로 제공되는 경우가 거의 없다.
- “값(Valueable)”은 상대적인 속성이다. 저장된 데이터의 상당 부분이 리 엔지니어링 프로젝트와 관련이 없을 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 소프트웨어 시스템은 데이터를 영구적으로 유지하기 위해 어떤 형태의 데이터베이스를 사용합한. 따라서 데이터베이스 내부의 데이터에 대한 정적 설명을 제공하는 일종의 데이터베이스 스키마가 존재한다.
- 데이터베이스에는 데이터베이스 내부의 실제 개체를 검사하는데 필요한 도구가 함께 제공되므로 레거시 데이터의 존재를 활용하여 결과를 미세 조정할 수 있다.
- 구현 언어의 데이터 구조를 데이터베이스 스키마에 매핑하는 데 어느 정도 전문 지식이 있으며, 데이터베이스 스키마에서 클래스 다이어그램을 재구성할 수 있을 정도이다.

¹ 서양에서는 오래된 물건을 다락방에 보관하곤 한다. -옮긴이

- 시스템의 기능과 프로젝트의 목표를 대략적으로 이해하고 있으므로(예: 첫 번째 접근을 통해 얻은 정보) 데이터베이스의 어느 부분이 프로젝트에 유용한지 평가할 수 있다.

해결

데이터베이스 스키마를 분석하여 어떤 구조가 가치 있는 데이터를 나타내는지 필터링한다. 나머지 팀원들을 위해 해당 지식을 문서화할 수 있도록 해당 엔티티를 나타내는 클래스 다이어그램을 도출한다.

단계

아래 단계에서는 시스템이 객체 지향 애플리케이션의 일반적인 경우인 관계형 데이터베이스(*relational database*)를 사용한다고 가정한다. 그러나 다른 종류의 데이터베이스 시스템을 사용하는 경우에도 이러한 단계 중 많은 부분이 여전히 적용될 수 있다. 단계 자체는 가이드라인일 뿐이므로 직관과 역추적을 충분히 활용하여 반복적으로 적용해야 한다.

준비. 관계형 데이터베이스 스키마에서 클래스 다이어그램을 도출하려면 먼저 테이블을 클래스로 표현하는 초기 모델을 준비한다. 소프트웨어 도구를 사용하여 이 작업을 수행할 수도 있지만 인덱스 카드 세트를 사용하는 것도 좋다.

1. 모든 테이블 이름을 열거하고 각 테이블에 대해 같은 이름의 클래스를 만든다.
2. 각 테이블에 대해 모든 열 이름을 수집하고 이를 해당 클래스에 속성으로 추가한다.
3. 각 테이블에 대해 후보 키(candidate key)를 결정한다. 그중 일부는 데이터베이스 스키마에서 직접 읽을 수 있지만 일반적으로 더 자세한 분석이 필요하다. 후보 키를 제안하는 경우가 많으므로 모든 (고유) 인덱스를 반드시 확인하자. 명명 규칙(ID 또는 #을 포함한 이름)도 후보 키를 나타낼 수 있다. 의심스러운 경우 데이터 샘플을 수집하여 후보 키가 데이터베이스 모집단 내에서 실제로 고유한지 확인한다.

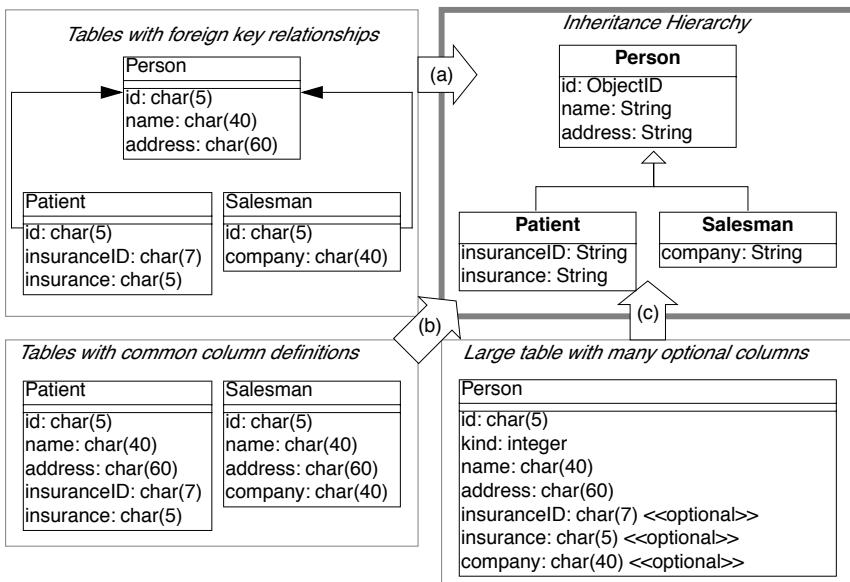


그림 4.2: 계열 관계형 테이블을 상속 계층 구조에 매핑하기. (a) 일대일, (b) 둘다, (c) 룰업

4. 테이블 간의 모든 외래 키(foreign key) 관계를 수집하고 해당 클래스 간의 연결을 생성한다. 외래 키 관계는 데이터베이스 스키마에 명시적으로 유지되지 않을 수 있으므로 열 유형과 명명 규칙(naming convention)을 통해 이를 유추해야 한다. 여기에는 동음이의어(homonym)(=열 이름과 유형은 동일하지만 의미는 다른)와 동의어(synonym)(=열 이름이나 유형은 다르지만 의미는 동일한)가 존재할 수 있으므로 신중한 분석이 필요하다. 이러한 어려움에 대처하려면 최소한 인덱스와 뷰 선언이 빈번한 탐색 경로를 가리키므로 이를 확인해야 한다. 가능하면 데이터베이스에 대해 실행되는 SQL 문의 조인 절(join clause)을 확인한다. 마지막으로, 데이터 샘플을 검사하여 특정 외래 키 관계를 확인(verify)하거나 반박(refute)한다.

상속 통합(Incorporate inheritance). 위의 단계를 마치면 관계형 데이터베이스에 저장되는 테이블을 나타내는 클래스 집합을 갖게 된다. 그러나 관계형 데이터베이스는 상속 관계를 나타낼 수 없기 때문에 외래 키에서 상속 관계를 유추해야

한다. (5-7단계에서 상속 관계의 세 가지 표현에 대한 용어는 [Fro94]에서 유래 했다.)

5. 일대일(*One to one*) (그림 4.2 (a)). 기본 키가 다른 테이블의 외래 키 역 할도 하는 테이블을 확인한다. 이러한 외래 키는 상속 관계를 나타낼 수 있기 때문이다. 이러한 테이블에 대해 실행되는 SELECT 문을 검사하여 일반적으로 이 외래 키에 대한 조인을 포함하는지 확인한다. 이 경우 테이블 이름과 해당 소스 코드를 분석하여 이 외래 키가 실제로 상속 관계를 나타내는지 확인한다. 그렇다면 외래 키에 해당하는 연결을 상속 관계로 변환한다.
6. 롤다운(*Rolled down*) (그림 4.2 (b)). 클래스 계층 구조가 여러 테이블에 분산되어 있고 각 테이블이 하나의 비추상 클래스를 나타내는 상황을 나타낼 수 있으므로 열 정의의 공통 집합이 있는 테이블을 확인한다. 중복된 열 정의의 각 클러스터에 대해 공통 수퍼클래스를 정의하고 해당 속성을 새 클래스 내부로 이동한다. 소스 코드에서 새로 생성된 클래스에 적용할 수 있는 이름을 확인한다.
7. 롤업(*Rolled up*) (그림 4.2 (c)). 열이 많고 선택적 속성이 많은 테이블은 전체 클래스 계층 구조가 단일 테이블에 표시되는 상황을 나타낼 수 있으므로 확인한다. 이러한 테이블을 발견한 경우 이 테이블에 대해 실행되는 모든 SELECT 문을 검토한다. 이러한 SELECT 문이 명시적으로 열의 하위 집합을 요청하는 경우 요청된 하위 집합에 따라 이 하나의 클래스를 여러 클래스로 나눌 수 있다. 이러한 클래스의 이름에 대해 열거형 유형 번호를 포함하는 'kind' 열과 같은 하위 유형 정보의 인코딩이 있는지 확인한다.

연관 통합(*Incorporate associations*). 데이터베이스에서 추출한 클래스 다이어 그램이 너무 작을 수 있다. 실제 상속 계층 구조의 클래스가 새로운 속성을 정의하지 않아 데이터베이스에서 누락되었을 수 있다. 또한 테이블 및 열 이름이 이상하게 들릴 수도 있다. 따라서 클래스 다이어그램을 소스 코드와 비교하여 확인하면 추가적인 인사이트를 얻을 수 있으므로(디자인 추측하기 참조), 이를 고려해 보자. 그런 다음 나머지 연관성을 구체화한다.

8. 연관 클래스(*association class*), 즉 두 개체가 연관되어 있다는 사실을 나

타내는 클래스를 결정한다. 가장 일반적인 예는 두 개의 외래 키로 구성된 후보 키를 가진 테이블로 표현되는 대다수 연결이다. 일반적으로 후보 키가 여러 외래 키의 연결인 모든 테이블은 연결 클래스의 잠재적 사례이다.

9. 상호 보완적인 연관성(complementary association)을 병합한다. 클래스 A가 클래스 B에 대한 외래 키 연관을 갖고 있고 클래스 B가 클래스 A에 대한 역외래 키를 갖는 경우가 있다. 이 경우 두 연관을 양방향으로 탐색 가능한 단일 연관으로 병합한다.
10. 외래 키 (foreign key) 대상을 해결한다. 데이터베이스에서 상속 계층이 롤업 또는 롤다운된 경우 테이블을 구성하는 클래스에서 테이블이 분해된 후 외래 키 대상이 모호해질 수 있다. 외래 키 대상이 계층 구조에서 너무 높거나 낮을 수 있으며, 이 경우 해당 연결에 참여하는 클래스가 너무 적거나 너무 많을 수 있다. 이러한 상황을 해결하려면 일반적으로 데이터 샘플과 SQL 문을 분석하여 실제로 어떤 클래스가 연결에 참여하는지 확인해야 한다.
11. 한정된 연관(qualified association), 즉 특정 조회 키(look-up key) 혹은 한정자(qualifier))를 제공하여 탐색할 수 있는 연관을 식별한다. 일반적인 예는 주문 번호가 한정자 역할을 하는 대다수 연결이다. 일반적으로 후보 키가 외래 키와 추가 열을 결합하는 모든 테이블은 잠재적인 한정자 연결이며, 추가 열은 한정자를 나타낸다.
12. 연관(association)에 대한 다중성(multiplicity)을 기록한다. 모든 연관은 외래 키 관계에서 파생되므로 모든 연결은 구조상 선택적 1 대 다 연관이다. 그러나 null이 아닌 선언, 인덱스 및 데이터 샘플을 검사하여 연관의 각 역할에 대한 최소 및 최대 다중성을 결정할 수 있는 경우가 많다.

검증. 데이터베이스 스키마만으로는 완전한 클래스 다이어그램을 도출하기에는 근거가 너무 약하다는 반복되는 지적에 유의하자. 다행히도 레거시 시스템에는 채워진 데이터베이스와 그 데이터베이스를 조작하는 프로그램이 있다. 따라서 데이터 샘플과 내장된 SQL 문을 사용하여 재구성된 클래스를 검증할 수 있다.

- 데이터 샘플. 데이터베이스 스키마는 기본 데이터베이스 시스템과 모델에서 허용하는 제약 조건만 지정한다. 그러나 문제 도메인에는 스키마에

표현되지 않은 다른 제약 조건이 포함될 수 있다. 데이터베이스에 저장된 실제 데이터의 샘플을 검사하여 다른 제약 조건을 유추할 수 있다.

- *SQL 문(SQL statements)*. 관계형 데이터베이스 스키마의 테이블은 외래 키를 통해 연결된다. 그러나 명시적인 외래 키가 없더라도 일부 테이블은 항상 함께 액세스되는 경우가 있다. 따라서 데이터베이스 엔진에 대해 실제로 어떤 쿼리가 실행되는지 확인하는 것이 좋다. 이를 수행하는 한 가지 방법은 프로그램에 포함된 모든 SQL 문을 추출하는 것이다. 또 다른 방법은 데이터베이스 시스템과 함께 제공되는 추적 기능을 통해 실행된 모든 쿼리를 분석하는 것이다.

오퍼레이션 통합(Incorporate operations). 데이터베이스에서 추출한 클래스 디어그램은 데이터 구조만 나타내며, 그 구조를 조작하는 데 사용된 오퍼레이션은 나타내지 않는다는 점을 분명히 알아야 한다. 따라서 결과 클래스 디어그램은 반드시 불완전할 수밖에 없다. 코드를 데이터베이스에서 추출한 모델과 비교하면(디자인 추측하기 및 계약 찾기 [p. 151] 참조) 추출된 클래스에 대한 연산을 통합할 수 있다.

트레이드오프

장점

- 팀 커뮤니케이션을 개선한다. 데이터베이스 스키마를 캡처하면 리엔지니어링 팀 내 및 프로젝트와 관련된 다른 개발자(특히 유지보수 팀)와의 커뮤니케이션이 개선된다. 또한 많은 개발 방법론에서 데이터베이스 설계의 중요성을 강조하기 때문에 프로젝트와 관련된 모든 사람은 아니더라도 많은 사람들이 데이터 스키마가 있다는 사실에 안심할 수 있다.
- 가치 있는 데이터에 집중하자. 데이터베이스는 백업과 보안을 위한 특별한 기능을 제공하므로 중요한 데이터를 저장하기에 이상적인 장소이다. 데이터베이스 스키마를 이해하면 중요한 데이터를 추출하여 향후 리엔지니어링 활동 중에 보존할 수 있다.

단점

- 범위가 제한된다. 데이터베이스는 오늘날 많은 소프트웨어 시스템에서 매우 중요하지만 전체 시스템에서 차지하는 비중은 극히 일부에 불과하다. 따라서 이 패턴에만 의존하여 시스템을 전체적으로 파악할 수는 없다.
- 정크 데이터가 포함되어 있다. 데이터베이스에는 가치 있는 데이터보다 훨씬 더 많은 데이터가 포함되며, 레거시 시스템이 얼마나 오래되었는지에 따라 아무도 제거하지 않아서 많은 정크 데이터가 저장되어 있을 수 있다. 따라서 복구한 데이터베이스 스키마를 리엔지니어링 프로젝트의 요구 사항과 일치시켜야 한다.
- 데이터베이스 전문 지식이 필요하다. 이 패턴을 사용하려면 기본 데이터베이스에 대한 많은 지식과 데이터베이스 스키마를 구현 언어에 매핑하는 구조가 필요하다. 따라서 이 패턴은 선택한 데이터베이스에서 구현 언어로의 매핑에 대한 전문 지식을 갖춘 사람이 적용하는 것이 바람직하다.
- 시스템의 동작 내용이 부족하다(*Lacks behavior*). 데이터베이스에서 추출한 클래스 다이어그램은 매우 데이터 지향적이며 동작이 거의 또는 전혀 포함되어 있지 않다. 진정한 객체 지향 클래스 다이어그램은 데이터와 동작을 모두 캡슐화해야 하므로 그런 의미에서 데이터베이스 스키마는 그림의 절반만 보여준다. 그러나 일단 데이터베이스 모델이 존재하면 나중에 누락된 동작을 추가할 수 있다.

어려움

- 오염된 데이터베이스 스키마. 데이터베이스 스키마 자체가 항상 중요한 객체에 대한 클래스 다이어그램을 재구성하는 데 가장 좋은 정보 소스는 아니다. 많은 프로젝트에서 데이터베이스 액세스를 최적화해야 하므로 데이터베이스 스키마를 깨끗하게 유지하는 것을 희생하는 경우가 많다. 또한 데이터베이스 스키마 자체는 시간이 지남에 따라 진화하기 때문에 서서히 성능이 저하된다. 따라서 데이터 샘플과 내장된 SQL 문을 분석하여 클래스 다이어그램을 개선하는 것이 매우 중요하다.

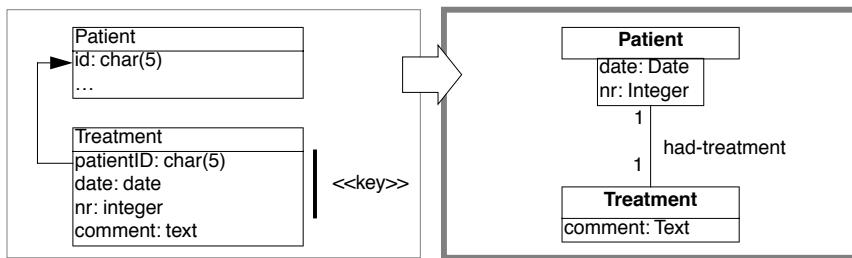


그림 4.3: 외래 키(patientID)와 두 개의 추가 열(date, nn)로 구성된 키를 통해 한정된 연관(qualified association)을 식별한다.

예시

XDoctor를 인수하면서 여러분의 회사는 기존 고객층을 계속 지원하기로 약속했다. 특히 고객에게 단 1바이트의 데이터도 손실되지 않을 것이라고 보장했는데, 이제 상사가 데이터베이스 구조를 복구해 달라고 요청한다. 자신의 제품 사용 경험을 통해 의사들이 환자 파일에 많은 관심을 갖고 있으며 그러한 정보를 잃어버리는 것은 용납할 수 없다는 것을 알고 있다. 따라서 데이터베이스 내부에 환자 파일이 저장되는 방식을 분석하는 것부터 시작하기로 결정한다.

먼저 모든 테이블 이름을 검색하여 **Patient**라는 이름의 테이블을 찾으려고 하지만 안타깝게도 찾을 수 없다. 그러나 **Person**이라는 테이블에서 거의 일치하는 항목이 있는데, `insuranceID`와 같은 열 이름을 보면 적어도 일부 환자 정보가 저장되어 있음을 알 수 있다. 그럼에도 불구하고 많은 열 이름이 선택 사항이므로 환자 정보가 다른 종류의 정보와 혼합된 롤업된 표현이 의심된다. 따라서 소스 코드를 확인하고 **Person** 테이블을 쿼리하는 모든 내장된 SQL 문을 찾는다(예: `grep "SELECT * Person"`). 실제로 이러한 쿼리가 사용되는 두 개의 클래스, 즉 **Patient**와 **Salesman**이 있으며 각 클래스에서 쿼리된 열의 하위 집합에서 그림 4.2에 표시된 상속 계층을 유추할 수 있다.

이제 **Patient**를 복구했으므로 환자가 받은 치료를 저장하는 테이블을 찾기 시작한다. 실제로 **Person** 테이블에 대한 외래 키가 있는 **Treatment** 테이블이 있다. 그러나 **Person**을 **Patient** 및 **Salesman** 클래스로 분해했으므로 외래 키의 대상을 확인해야 한다. `patientID`를 통해 **Person** 및 **Treatment** 테이블을 조인하면(`SELECT DISTINCT name, kind FROM Person, Treatment WHERE`

Person.id = Treatment.patientID) 선택한 모든 사람이 실제로 Patient에 해당하는 종류를 가지고 있음을 확인할 수 있다. 따라서 외래 키의 대상을 Treatment에서 Patient로 설정한다(그림 4.2 의 왼쪽 참조). 다음으로 Treatment에 정의된 인덱스를 확인하고 patientID - date - nr 열에 고유 인덱스가 있음을 확인하여 이 열이 후보 키 역할을 한다는 결론을 내린다. lctTreatment의 후보 키는 두 개의 추가 열과 결합된 외래 키로 구성되어 있으므로 한정된 연관(qualified association)이 의심된다. 이 가정을 확인하기 위해 데이터 샘플 (SELECT name, date, nr FROM Person, Treatment WHERE Person.id = Treatment.patientID ORDER BY name, date, nr)을 분석하여 날짜와 번호가 특정 환자의 치료를 고유하게 식별하는지 확인한다. 결과적으로 외래 키를 각 역할에 대해 다중성 1(multiplicity of one)을 가지는 정규화된 연관 had-treatment로 변환한다.

근거

객체 모델은 데이터 구조를 간결하게 설명하고 구조적 제약사항을 포착하기 때문에 데이터베이스 애플리케이션에 중요하다.

— 마이클 블라하 외 [BLM98]

잘 정의된 중앙 데이터베이스 스키마를 갖는 것은 영구 데이터를 다루는 대규모 소프트웨어 프로젝트에서 흔히 볼 수 있는 관행이다. 특정 데이터 구조에 액세스하는 방법에 대한 공통 규칙을 지정할 뿐만 아니라 팀원 간에 작업을 분담하는 데도 큰 도움이 된다. 따라서 다른 리버스 엔지니어링 활동을 진행하기 전에 데이터베이스의 정확한 모델을 추출하는 것이 좋다.

데이터베이스 모델을 추출하는 것은 본질적으로 상향식(bottom-up) 접근 방식이다. 데이터베이스 스키마에 포함된 대략적인 정보에서 시작하여 만족스러운 클래스 다이어그램이 나올 때까지 다듬는 것이다. 데이터베이스 스키마는 이미 더 자세한 표현에서 추상화된 것이기 때문에 이러한 상황에서는 상향식 접근 방식이 매우 효과적이다.

모든 데이터는 해당 클래스 내에서 숨겨야 한다.

— 아서 리엘, 휴리스틱 2.1 [Rie96]

정보 숨김은 중요한 설계 원칙이며, 대부분의 저자는 클래스의 경우 모든 데이터가 클래스 내에서 캡슐화되고 해당 클래스에 정의된 연산을 통해서만 액세스해야 한다는 데 동의한다. 안타깝게도 데이터베이스에서 추출한 클래스 다이어그램은 데이터베이스의 특성 때문에 모든 데이터를 노출하게 된다. 따라서 이 클래스 다이어그램은 데이터베이스에 대한 잘 설계된 인터페이스를 향한 첫 단계일 뿐이다.

알려진 용도

데이터베이스 시스템의 리버스 엔지니어링 및 리엔지니어링은 잘 탐구된 연구 분야이다 [Arn92] [MJS⁺00]. 여러 실험에 따르면 잘못 설계된 데이터베이스 시스템에서도 데이터베이스 구조를 복구하는 것이 가능하다는 사실이 밝혀졌다. 예를 들어, 주요 RDBMS 공급업체의 데이터 사전과 기계 부품에 대한 데이터를 저장하는 운영 데이터베이스의 리버스 엔지니어링에 관한 실험에 관한 보고서 [PB94]를 참조하자. [HEH⁺96]는 프로토타입 데이터베이스 리버스 엔지니어링 툴킷과 이 툴킷이 적용된 5가지 산업 사례에 대해 설명한다. 데이터베이스 리버스 엔지니어링의 예측 불가능한 특성을 설명하기 위해 [JSZ97]에서는 관계형 데이터베이스 스키마에서 클래스 다이어그램을 추출하는 도구의 핵심으로 퍼지 추론 엔진을 사용하는 것에 대해 설명한다.

다음 단계

퍼시스턴트 데이터 분석하기 은 소프트웨어 시스템의 영구 데이터에 대한 클래스 다이어그램을 생성한다. 이러한 클래스 다이어그램은 매우 대략적이며 주로 데이터의 구조에 관한 것이지 데이터의 동작에 관한 것이 아니다. 그러나 디자인 추측하기 및 계약 찾기 [p. 151]를 적용하여 더 구체화할 수 있는 이상적인 초기 가설로 사용될 수 있다.

다른 데이터베이스로 마이그레이션해야 하는 경우, 데이터베이스 모델에 대한 이해를 테스트라는 생명 보험 [p. 165]에 설명된 대로 테스트 스위트에 적용(cast)해야 합니다.

관계형 데이터베이스에 객체 지향 데이터 구조를 매핑하는 다양한 방법을 설명하는 패턴, 관용구 및 패턴 언어가 존재한다는 점에 유의하자 [BW96].

[KC98b]. 데이터베이스 스키마를 리버스 엔지니어링할 때 이를 참조하면 도움이 될 수 있다.

4.2 디자인 추측하기

의도 소스코드에 대한 디자인의 가설을 확인하여 소스코드에 대한 디자인을 점진적으로 구체화(refine)한다.

문제

소스 코드에서 디자인 개념이 표현되는 방식을 어떻게 복구할 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 많은 디자인 개념이 있으며 사용되는 프로그래밍 언어에서 이를 표현하는 방법은 무수히 많다.
- 소스 코드의 대부분은 디자인과 관련이 없으며 오히려 다음과 같은 문제와 관련이 있다. 구현 문제(글루 코드, 사용자 인터페이스 제어, 데이터베이스 연결 등)와 관련이 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 시스템 기능에 대한 대략적인 이해를 통해 예를 들어 를 통해 얻은 문서 스키밍하기 [p. 67] 및 데모 중 인터뷰하기 [p. 75]를 통해 얻을 수 있음), 따라서 초기 어떤 디자인 문제를 해결해야 하는지 알 수 있다.
- 개발 전문 지식(development expertise)이 있으므로 해당 문제를 어떻게 디자인할지 상상할 수 있다. 문제를 어떻게 설계할지 상상할 수 있다.
- 소스 코드의 주요 구조에 대해 어느 정도 익숙하다(예를 들어 한 시간 안에 모든 코드 읽기 [p. 59]로 얻은 정보가 있다). 그러므로 길을 찾을 수 있다.

해결

개발 전문 지식을 활용하여 디자인을 나타내는 가상의 클래스 다이어그램을 만들어보자. 클래스 다이어그램의 이름이 소스 코드에 있는지 확인하고 그에 따라 모델을 조정하여 해당 모델을 구체화하자. 클래스 다이어그램이 안정화될 때까지 이 과정을 반복하자.

단계

1. 시스템에 대한 이해를 바탕으로 소스 코드에서 무엇을 기대할 수 있는지에 대한 초기 가설 역할을 하는 클래스 다이어그램을 작성하자. 클래스, 연산 및 속성의 이름은 자신의 경험과 잠재적인 명명 규칙을 바탕으로 추측하자(문서 스키밍하기 [p. 67] 참조).
2. 클래스 다이어그램에 있는 이름(즉, 클래스, 속성 및 연산 이름)을 열거하고 사용 가능한 도구를 사용하여 소스 코드에서 해당 이름을 찾아보자. 소스 코드 내의 이름이 항상 그 이름이 나타내는 개념과 일치하는 것은 아니므로 주의하자.² 이를 방지하기 위해 소스 코드에 나타날 가능성에 따라 이름의 순위를 매길 수 있다.
3. 소스 코드에 나타나는 이름을 추적하고(가설 확인) 소스 코드의 식별자와 일치하지 않는 이름을 추적하자(가설과 모순됨). 불일치는 시스템을 이해할 때 반드시 거쳐야 하는 학습 과정을 촉발하므로 궁정적이라는 점을 기억하자.
4. 불일치를 기반으로 클래스 다이어그램을 조정한다. 이러한 조정에는 다음이 포함될 수 있다.
 - (a) 이름 바꾸기(*renaming*): 소스 코드에서 선택한 이름이 가설과 일치하지 않는 것을 발견한 경우 사용한다.
 - (b) 리모델링(*remodelling*): 디자인 개념의 소스 코드 표현이 모델에 있는 것과 일치하지 않는다는 것을 알게 될 때 사용한다. 예를 들어 연산을 클래스로 변환하거나 속성을 연산으로 변환할 수 있다.
 - (c) 확장(*extending*): 클래스 다이어그램에 나타나지 않는 소스 코드의 중요한 요소를 발견한 경우 사용한다.
 - (d) 대안 찾기(*seeking alternatives*): 소스 코드에서 디자인 개념을 찾을 수 없는 경우 사용한다. 여기에는 불일치가 거의 없는 경우 동의어를 시도하는 것이 포함될 수 있지만 다음과 같은 경우도 수반될 수 있다. 불일치가 많을 때는 완전히 다른 클래스 다이어그램을 정의해야 할 수도 있다.
5. 만족스러운 클래스 다이어그램을 얻을 때까지 2 4 단계를 반복한다.

²리버스 엔지니어링 경험 중 하나에서 영어와 독일어가 혼용된 소스 코드를 마주한 적이 있었다. 예상할 수 있듯이 이것은 문제를 매우 복잡하게 만든다.

변형

비즈니스 객체에 대해 추측한다. 시스템 설계에서 중요한 부분은 문제 도메인 (problem domain)의 개념이 소스 코드에서 클래스로 표현되는 방식이다. 이 패턴의 변형을 사용하여 소위 “비즈니스 객체”를 추출할 수 있다.

초기 가설을 구축하는 한 가지 방법은 요구 사항의 명사 구문을 초기 클래스 이름으로, 동사 구문을 초기 메서드 이름으로 사용하는 것이다([WBWW90] [BS97] [Boo94]에서 클래스와 그 책임(responsability) 찾기에 대해 자세히 다루고 있다). 어떤 객체가 어떤 역할을 수행하는지 알아내는 데 도움이 될 수 있는 사용 시나리오를 통해 이 정보를 보강해야 할 수도 있다. 데모 중 인터뷰하기 [p. 75]를 통해서 이러한 시나리오를 구할 수 있을 것이다. (참조 시나리오 및 사용 사례는 [JCJO92] [SW98], 역할 모델링에 대해서는 [Ree96] [RG98]를 참조하자.)

패턴에 대해 추측한다. 패턴은 “주어진 맥락에서 공통적인 디자인 문제에 대한 반복적인 해결책”이다. 특정 패턴이 어디에 적용되었는지 알면 기본 시스템 설계에 대해 많은 것을 알 수 있다. 이 변형은 아키텍처 [BMR⁺96], 분석 [Fow97] 또는 디자인 패턴 [GHJV95]의 발생에 대한 가설을 검증한다.

아키텍처에 대해 추측한다. “소프트웨어 아키텍처는 소프트웨어 시스템의 하위 시스템과 컴포넌트 및 이들 간의 관계에 대한 설명이다.” [BMR⁺96](일명 컴포넌트와 커넥터(Components and Connectors) [SG96]). 소프트웨어 아키텍처는 일반적으로 시스템의 대략적인 수준 설계와 연관되어 있으므로 전체 구조를 이해하는 데 매우 중요합니다. 소프트웨어 아키텍처는 리버스 엔지니어링이 매우 어려운 영역인 여러 협력 프로세스가 있는 분산 시스템의 맥락에서 특히 관련이 있다.

이 변형은 어떤 컴포넌트와 커넥터가 존재하는지, 또는 분산 시스템의 맥락에서 어떤 프로세스가 존재하는지, 어떻게 시작되고 어떻게 종료되며 어떻게 상호 작용하는지에 대한 가설을 세우고 구체화한다. 아키텍처 패턴의 카탈로그는 [BMR⁺96]를, 잘 알려진 아키텍처 스타일 목록은 [SG96]를 참조하자. 동시 프로그래밍에 적용될 수 있는 몇 가지 일반적인 패턴과 이디엄은 [Lea96]를, 분산 시스템의 아키텍처 패턴은 [SSRB00]를 참조하자.

트레이드오프

장점

- 규모확장이 잘 지원된다. 소스 코드에서 무엇을 찾을 수 있을지 추측하는 것은 규모 확장에 유리한 기법이다. 이는 대규모 객체 지향 프로그램(100개 이상의 클래스)의 경우 상향식 접근 방식이 금방 비실용적이 되기 때문에 특히 중요하다.
- 투자는 성과를 돌려준다. 이 기술은 리소스와 도구 측면에서 상당히 저렴하며, 이해의 양을 고려할 때 확실히 비용이 낮다.

단점

- 전문 지식이 필요하다. 소스 코드에서 보이는 것을 인식하려면 이디엄, 패턴, 알고리즘, 기술에 대한 방대한 지식 레퍼토리가 필요하다. 따라서 이 패턴은 전문가가 적용하는 것이 바람직하다.
- 시간이 많이 소요된다. 이 기술은 리소스와 도구 측면에서 상당히 저렴하지만 만족스러운 표현을 도출하기까지 상당한 시간이 필요하다.

어려움

- 일관성 유지는 어렵다. 리버스 엔지니어링 프로젝트가 진행되고 소프트웨어 시스템에 대한 이해도가 높아지는 동안 클래스 다이어그램을 최신 상태로 유지할 계획을 세워야 한다. 그렇지 않으면 노력이 혼수고가 될 수 있다. 따라서 클래스 다이어그램이 소스 코드에 사용된 명명 규칙에 크게 의존하고 있는지, 클래스 다이어그램이 구성 관리 시스템의 제어를 받는지 확인하자.

예시

*XDoctor*를 인수하면서 여러분의 회사는 기존 고객층을 계속 지원하겠다고 약속했다. 그리고 스위스가 6개월 이내에 유로 지역에 가입할 예정이므로 마케팅 부서에서는 유로 전환이 제대로 지원되는지 확인하고자 한다. 1차 평가 결과

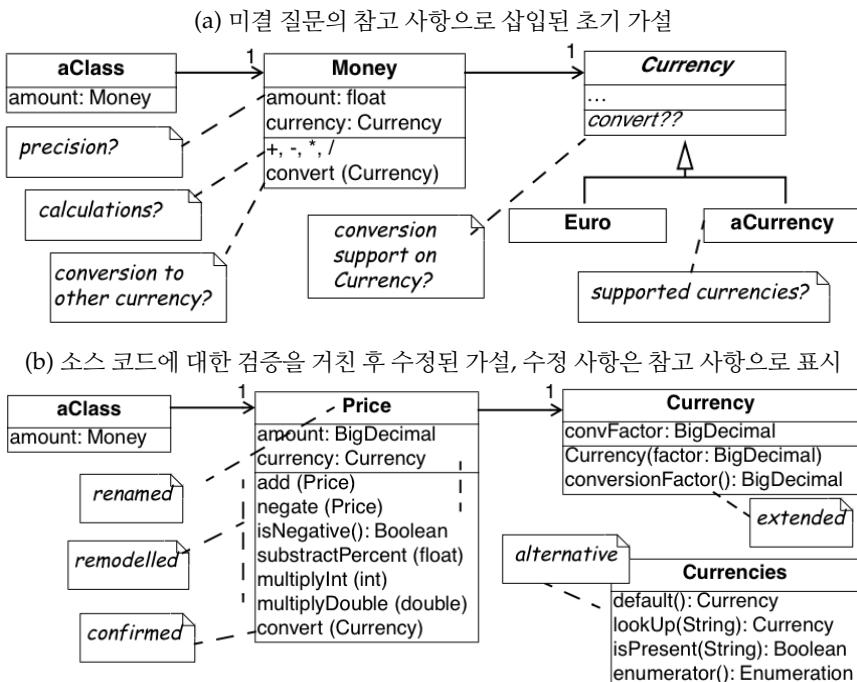


그림 4.4: 유로화 표시에 관한 가설 구체화하기. (a) 다양한 통화에 대한 서브클래스, (b) 통화에 대한 경량 패턴 접근 방식

유로화가 어느 정도 지원되는 것으로 확인되었다(즉, 사용자 설명서에 설명되어 있고 **Currency**라는 클래스가 존재한다). 이제 상사가 법적 의무를 충족할 수 있는지, 그렇지 않다면 소프트웨어를 조정하는 데 얼마나 걸리는지 조사해 달라고 요청한다.

이전 코드 검토에서 디자인이 상당히 훌륭하다는 것을 알았으므로 디자이너가 수량 [p. 351] 패턴을 일부 변형한 것을 적용했을 것으로 생각한다. 따라서 그림 4.4 (a)에 표시된 클래스 다이어그램의 형태로 초기 가설을 정의한다. 화폐의 양(부동 소수점 숫자)과 사용 중인 통화(**Currency** 클래스의 인스턴스)의 두 가지 속성을 가진 **Money** 클래스가 하나 있다. `lctMoney` 클래스에서 더하기, 빼기, 곱하기, …와 같은 표준 연산과 다른 통화로 변환하기 위한 연산 하나를 수행한다고 가정한다. **Currency**에는 지원되는 모든 통화에 대한 서브클래스와 한 통화에서 다른 통화로의 변환을 지원하는 연산이 있어야 한다. 물론 몇 가지

질문에 대한 답이 없는 경우도 있으므로 클래스 다이어그램에 메모해 두어야 한다.

1. Money의 금액에 대한 정밀도는 어떻게 되는가?
2. Money의 인스턴스에서 어떤 계산이 허용되는가?
3. Money의 인스턴스를 다른 통화로 어떻게 변환하는가?
4. 이 변환은 내부적으로 어떻게 이루어지는가? lctCurrency 클래스의 지원은 어떻게 이루어지는가?
5. 어떤 통화가 지원되나요?

이러한 질문에 답하기 위해 소스 코드를 통해 가설을 검증하고 그에 따라 클래스 다이어그램을 조정한다. 파일 이름을 훑어보면 **Currency** 클래스는 있지만 **Money**라는 클래스는 없고, 모든 소스 코드를 grep 검색하면 **Money** 클래스가 존재하지 않음을 확인할 수 있다. 어떤 패키지가 **Currency**를 임포트하는지 검색하면 소스 코드의 실제 이름이 **Price**라는 것을 금방 알 수 있고 그에 따라 **Money** 클래스의 이름을 변경할 수 있다.

Price 클래스 내부를 살펴보면 금액이 고정 소수점 숫자로 표시되어 있음을 알 수 있다. 아래와 같은 짧은 주석이 있다.

Michael (Oct 1999) -- Bug Report #324 -- Replaced
Float by BigDecimal due to rounding errors in the
floating point representation. Trimmed down the
permitted calculation operations as well.

Price 클래스의 인터페이스를 확인하면 계산 연산이 실제로 최소로 포함되어 있다는 것을 알 수 있다. 덧셈과 뺄셈(피연산자가 음수인 덧셈을 통해 뺄셈을 수행해야 함)과 백분율을 취하고 다른 숫자와 곱하기 위한 몇 가지 추가 연산만 있다. 그러나 가격 변환에 관한 가설을 확인하는 변환 연산도 볼 수 있다.

다음으로 **Currency**의 서브클래스를 찾아보지만 아무것도 찾을 수 없을 것 같다. 당황한 여러분은 다른 해결책을 생각해 보기 시작하고 잠시 후 경량 [p. 350] 패턴 적용의 가능성을 고려한다. 결국, 각 통화에 대해 별도의 서브 클래스를 갖는 것은 추가 동작이 포함되지 않기 때문에 약간의 오버헤드가 발생한다. 또한 경량 패턴 접근 방식을 사용하면 단일 유로 객체로 유로 통화의

모든 발생을 표현함으로써 많은 메모리를 절약할 수 있다. 이 대안을 검증하려면 **Currency**에 대한 생성자 메서드의 모든 발생을 찾아보면 grep **Currency**가 트릭을 사용하여, 실제로 허용되는 모든 통화가 포함된 글로벌 테이블을 캡슐화하는 클래스를 발견하게 된다. 초기화 메서드를 살펴보면 다음과 같은 사실을 알 수 있다. 실제 테이블에는 두 가지 통화에 대한 항목이 포함되어 있다. 유로와 벨기에 프랑이다.

마지막으로, 실제 변환을 좀 더 자세히 살펴보기 위해 **Price.convert** 연산과 **Currency** 클래스의 내용을 살펴보자. 약간의 검색을 통해 각 **Currency**에는 단일 변환 계수가 있다는 것을 알게 된다. 이렇게 하면 변환은 가능한 모든 통화 간에 두 가지 방식으로 작동해야 하지 않는가? 두 가지 방식으로 작동해야 하지 않을까? 하지만 **conversionFactor** 메서드의 모든 호출을 확인하면 변환이 기본 통화라는 개념을 중심으로 설계되었다는 것을 추론할 수 있다 (예: **Currencies.default()** 연산). 그리고 **conversionFactor**가 주어진 통화를 기본 통화로 변환한다. **Price.convert** 연산을 확인하면 실제로 기본 통화에 대한 테스트가 있으며 이 경우 변환이 단순 곱셈에 해당한다는 것을 알 수 있다. 다른 경우에는 기본 통화로의 중간 변환을 포함하는 2단계 계산을 통해 변환이 수행된다.

결과가 상당히 만족스러우면 클래스 다이어그램을 그림 10(b)에 표시된 것과 같이 조정한다. 이 모델에는 원래 가설에 대한 수정 사항을 애너테이션(annotation)으로 표시되어 있으므로 동료들이 추론 과정을 재구성할 수 있도록 원래 모델과 수정된 모델을 모두 구성 관리 시스템에 저장한다. 또한 결과를 요약한 다음 보고서를 제출한다.

유로로 변환. 유로 변환 기능을 사용할 수 있지만 추가 작업이 필요하다. 하나의 중앙 클래스(**Currencies**)는 하나의 기본 통화(**Currencies.default**)를 포함하여 지원되는 통화 목록을 유지 관리한다. 유로로 변환하려면 이 클래스의 초기화를 변경하여 기본값이 유로가 되도록 해야 한다. 데이터베이스에 저장된 모든 가격도 변환해야 하지만 이는 우리가 연구한 범위를 벗어난다.

후속 작업을 수행합니다.

- 구성 파일에서 기본 통화 및 변환 계수를 읽도록 **Currencies** 클래스의 초기화를 조정한다.
- **Prices**을 어떻게 변환해야 하는지 데이터베이스를 확인한다.

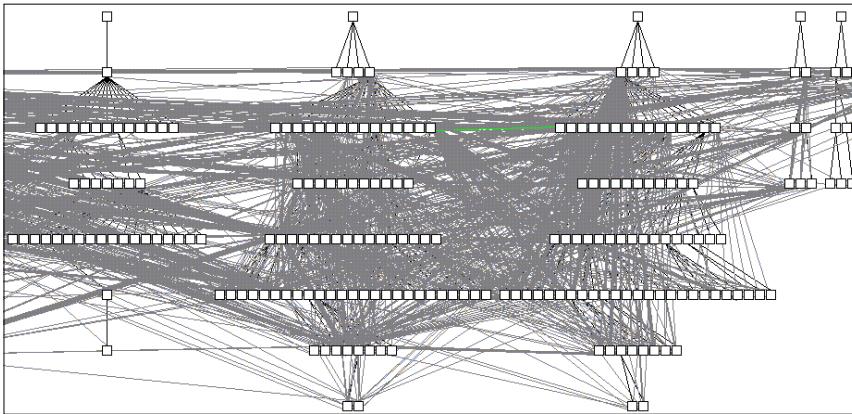


그림 4.5: 상향식 설계 추출 접근 방식으로 얻은 화이트 노이즈(white-noise). 이 그림은 중간 크기의 시스템에 대한 모든 메서드 호출과 속성의 접근으로 보강된 상속 계층 구조의 일부를 보여준다. 이 시각화는 CodeCrawler [DDL99] [Lan99]를 이용하였다.

근거

디자인 추출에 대한 순진한 접근 방식은 상향식으로, 먼저 소스 코드에서 완전한 클래스 다이어그램을 만든 다음 노이즈를 제거하여 압축하는 것이다. 불행히도, 상향식 접근 방식은 대규모 시스템에서는 작동하지 않는다. 왜냐하면 일반적으로 많은 화이트 노이즈가 발생하기 때문이다(예를 들어 중간 규모의 시스템에 대한 연관성을 가진 상속 계층을 보여주는 그림 4.5를 참조하자). 게다가 이러한 상향식 접근 방식은 중요한 개념 대신 관련 없는 노이즈에 집중하게 만들기 때문에 이해도를 크게 향상시키지 못한다.

“우리는 일을 바로잡기 전에 일을 옮바로 하지 못한다. *We get things wrong before we get things right.*”

— 앤리스터 콙번, [Coc93]

레거시 문제에 대한 진정한 이해를 얻으려면 학습 과정을 거쳐야 한다. 디자인 추축하기는 이러한 학습 과정을 자극하기 위한 것이므로 가설과 모순되는 증거는 가설을 확인하는 증거만큼이나 가치가 있다. 실제로 불일치는 대안적인

해결책을 고려하고 장단점을 평가하게 하며, 바로 그 순간 진정한 이해가 생겨난다.

알려진 용도

[MN97]에는 Microsoft의 소프트웨어 엔지니어가 이 패턴(논문에서는 “리플렉션 모델(Reflection Model)”이라고 함)을 적용하여 Microsoft Excel의 C 코드를 리버스 엔지니어링한 실험에 대한 보고서가 있다. 이 이야기의 좋은 점 중 하나는 그 소프트웨어 엔지니어가 시스템의 해당 부분을 처음 접한 사람이었고 동료들이 그에게 설명하는 데 많은 시간을 할애할 수 없었다는 것이다. 하지만 짧은 토론 끝에 그는 초기 가설을 세운 다음 소스 코드를 사용하여 점차적으로 이해를 구체화할 수 있었다. 이 논문에는 모델 지정, 모델에서 소스 코드로의 매핑, 모델에 대한 코드 확인에 도움이 되는 경량 도구에 대한 설명도 포함되어 있다.

논문 [Big89] [BMW93] [BMW94] 문서에서는 이 패턴을 성공적으로 사용한 몇 가지 사례를 보고한다(여기서는 이를 “개념 할당 문제(concept assignment problem)”라고 부른다). 특히, 저자들은 고급 탐색 기능, 프로그램 슬라이싱, 프롤로그 기반 쿼리 언어가 포함된 DESIRE라는 도구 프로토타입을 설명한다. 이 도구는 여러 회사의 많은 사람들이 최대 220 KLOC의 프로그램을 분석하는 데 사용되었다. 다른 잘 알려진 애플리케이션으로는 Rigi 그룹이 보고한 것으로, 이 그룹은 2백만 줄이 넘는 PL/AS 코드 [WTMS95]로 구성된 시스템에 이 패턴을 적용했다.

이러한 접근 방식은 소스 코드 [GW99] [WG98]의 정적 분석만을 기반으로 객체 지향 설계를 절차적 구현에 매핑하는 데 사용할 수 있음이 입증되었다. 그럼에도 불구하고 새로운 접근 방식은 더 풍부하고 다양한 정보 소스를 활용하려고 한다. 예를 들어 DALI는 메이크파일과 프로파일러의 정보도 분석한다 [BCK98] [KC98a] [KC99]. 반면에 가우디는 정적 호출 그래프와 런타임 추적 [RD99]를 혼합하여 가설을 검증한다.

다음 단계

이 패턴이 끝나면 디자인의 일부를 나타내는 클래스 다이어그램이 생긴다. 디자인 품질에 대한 인상을 얻기 위해 예외적인 엔티티 연구하기를 사용할 수 있다. 좀 더 정교한 모델이 필요하다면 상세 모델 캡처 [p. 129]의 패턴을 고려해 보자. 리버스 엔지니어링 작업이 마이그레이션 또는 리엔지니어링 프로젝트의 일부인 경우, 테스트라는 생명 보험 [p. 165]에 설명된 대로 디자인에 대한 이해를 수행하고 그것을 테스트 스위트에 투영해야 한다.

4.3 예외적인 엔티티 연구하기

의도 측정값을 수집하고 예외 값을 연구하여 잠재적인 설계 문제를 식별한다.

문제

대규모 소프트웨어 시스템에서 잠재적인 설계 문제를 신속하게 식별하려면 어떻게 해야 하는가?

이 문제는 다음과 같은 이유로 어렵다.

- 문제가 있는 설계와 좋은 설계를 구분하는 쉬운 방법은 없다. 설계의 품질을 평가하는 것은 설계가 해결하려는 문제의 관점에서 이루어져야 하므로 설계만으로는 결코 유추할 수 없다.
- 어떤 코드가 디자인 문제를 나타내는지 확인하려면 먼저 그 코드의 내부 구조를 풀어 해쳐야 한다. 문제가 있는 코드의 경우 이는 일반적으로 매우 어렵다.
- 시스템은 규모가 크기 때문에 모든 코드의 설계 품질을 자세히 평가하는 것은 불가능하다.

다음과 같은 이유로 이 문제를 해결할 수 있다.

- **메트릭 (metrics)** 도구를 사용할 수 있으므로 소스 코드의 엔티티에 대한 여러 측정값을 빠르게 수집할 수 있다.
- 시스템 기능에 대한 대략적인 이해(*rough understanding*)가 있으므로(예: 첫 번째 컨택 을 통해 획득) 시스템 컨텍스트에서 설계의 품질을 평가할 수 있다.
- 소스 코드에 필요한 살펴보기 도구(*tools to browse*)가 있으므로 특정 엔티티가 실제로 문제가 되는지 수동으로 확인할 수 있다.

솔루션

소프트웨어 시스템을 구성하는 구조적 엔티티(예: 상속 계층 구조, 패키지, 클래스 및 메서드)를 측정하고 수집한 정량적 데이터에서 예외를 찾아보자. 이러한

예외가 설계 문제를 나타내는지 수동으로 확인한다.

힌트

측정을 통해 소프트웨어 시스템에서 문제가 있는 설계를 식별하는 것은 데이터 수집과 해석 모두에 대한 전문 지식이 필요한 섬세한 작업이다. 다음은 원 수치 (raw number)를 최대한 활용하기 위해 고려할 수 있는 몇 가지 힌트이다.

- 어떤 도구를 사용할 것인가? 소스 코드 엔티티의 다양한 속성을 측정하는 도구는 상용 및 퍼블릭 도메인을 막론하고 많다. 그럼에도 불구하고 이러한 도구를 정기적으로 사용하는 개발팀은 거의 없기 때문에 이 패턴을 적용하기 전에 메트릭 도구를 찾아야 할 가능성이 높다.

원칙적으로 개발팀에서 사용하는 도구를 살펴보고 코드에 대한 데이터를 수집하는 데 사용할 수 있는지 확인하는 것부터 시작하자. 예를 들어, 린트(lint)와 같은 코드 검증 도구는 측정의 기초가 될 수 있다. 현재 사용 중인 개발 도구 중 데이터를 수집할 수 있는 도구가 없는 경우에만 메트릭 도구를 찾아보자. 이 경우, 설치와 학습에 귀중한 시간을 소비하고 싶지 않다면 단순성을 도구 채택의 주요 기준으로 삼아야 한다. 두 번째 도구 채택 기준은 메트릭 도구가 사용 중인 다른 개발 도구와 얼마나 쉽게 통합 되는지이다.

- 어떤 메트릭(metric)을 수집할 것인가? 일반적으로 복잡한 메트릭은 더 많은 계산을 수반하지만 더 나은 성과를 내는 경우는 드물기 때문에 간단한 메트릭을 고수하는 것이 좋다.

예를 들어, 큰 메서드를 식별하려면 모든 캐리지 리턴이나 새로운 줄을 세어 코드 줄을 세는 것으로 충분하다. 대부분의 다른 메서드 크기 메트릭은 어떤 형태의 구문 분석이 필요하며 이러한 노력은 일반적으로 이득을 얻을 만한 가치가 없다.

- 어떤 메트릭 변형을 사용할 것인가? 일반적으로 어떤 메트릭 변형을 선택하든 선택 사항을 명확하게 명시하고 일관되게 적용한다면 큰 차이가 없다. 여기에서도 특별한 이유가 없는 한 가장 간단한 변형을 선택하는 것이 좋다.

예를 들어 코드 줄을 세는 경우 주석 줄을 포함할지 제외할지 또는 소스

코드가 예쁜 인쇄를 통해 정규화된 후 줄을 세는지 여부를 결정해야 한다. 그러나 잠재적인 디자인 문제를 찾을 때 주석 줄을 제외하거나 소스 코드를 정규화하는 추가 작업을 수행하는 것은 일반적으로 효과가 없다.

- 어떤 임계값을 적용할 것인가? 신뢰성이 필요하므로 임계값을 적용하지 않는 것이 좋다.³ 우선, 임계값을 선택하는 것은 개발팀에서 적용한 코딩 표준에 따라 이루어져야 하며 이러한 표준에 반드시 액세스할 수 있는 것은 아니기 때문이다. 둘째, 임계값을 사용하면 시스템 내부에 얼마나 많은 정상 엔티티가 있는지 알 수 없으므로 이상 징후에 대한 관심이 왜곡될 수 있다.
- 결과를 어떻게 해석할 것인가? 이상 징후가 반드시 문제가 되는 것은 아니므로 측정 데이터를 해석할 때는 주의를 기울여야 한다. 엔티티가 실제로 문제가 있는지 여부를 평가하려면 동일한 엔티티에 대해 여러 측정값을 동시에 검사하는 것이 좋다. 예를 들어, 대규모 클래스에 대한 연구에만 국한하지 말고 클래스의 크기와 하위 클래스 수 및 상위 클래스 수를 결합하면 클래스 계층 구조에서 클래스가 어디에 위치하는지에 대해 알 수 있다.

그러나 서로 다른 측정값을 하나의 숫자로 결합하는 공식은 구성 요소에 대한 의미를 잃게 되므로 피해야 한다. 따라서 첫 번째 열에는 엔티티의 이름을 표시하고 나머지 열에는 다른 측정 데이터를 표시하는 표에 결과를 표시하는 것이 좋다. 이러한 테이블을 다양한 측정 열에 따라 정렬하면 예외적인 값을 식별하는 데 도움이 된다.

- 이상값을 빠르게 식별하는 방법은 무엇인가? 측정 데이터를 표로 표현하여 예외적인 값을 식별할 수는 있지만, 이러한 접근 방식은 지루하고 오류가 발생하기 쉽다. 대부분의 측정 도구에는 대량의 측정값을 스캔하는 데 도움이 되는 몇 가지 시각화 기능(히스토그램, 분산형 차트, ...)이 포함되어 있으며, 이는 일반적으로 잠재적인 디자인 문제에 빠르게 집중하는 데 더 좋은 방법이다.
- 이후에 코드를 살펴봐야 하는가? 측정값만으로는 엔티티가 정말 문제가 있는지 여부를 판단할 수 없으므로 항상 사람의 평가가 필요하다. 메트릭은 잠재적인 문제가 있는 엔티티를 빠르게 식별하는 데 큰 도움이 되지만

³ 대부분의 메트릭 도구에서는 임계값 간격을 지정한 다음 측정값이 그 간격에 해당하는 엔티티만 표시하여 특정 엔티티에 집중할 수 있다.

확인을 위해서는 코드 탐색이 필요하다. 대규모 엔티티는 일반적으로 매우 복잡하므로 해당 소스 코드를 이해하는 것이 어려울 수 있다는 점에 유의하자.

- 일반 엔티티는 어떤가? 숙련된 프로그래머는 중요한 기능을 잘 설계된 여러 컴포넌트에 분산하는 경향이 있다. 반대로 예외적인 엔티티는 정말 중요한 코드가 리팩터링되었기 때문에 관련이 없는 경우가 많다. 따라서 휴리스틱을 적용하는 것일 뿐이라는 점을 인지해야 한다. 단순히 중요하지 않다고 판단되어 디자인 문제를 나타내지 않는 코드를 연구하고 있을 수 있다.

트레이드오프

장점

- 규모확장이 잘된다. 메트릭은 대규모 시스템에 쉽게 적용할 수 있는데, 주로 메트릭 도구를 사용하면 전체 엔티티의 약 20%에 대해 추가 조사가 필요하기 때문이다. 여러 메트릭을 적절히 조합하면(가급적 시각화 형식을 사용하여) 시스템의 어느 부분이 잠재적인 설계 문제를 나타내는지 매우 빠르게 추론할 수 있다.
- 전체적인 관망이 매력적이다. 적절한 도구를 지원하면 메트릭 데이터의 시각적 표현을 생성하여 설계의 장점과 문제점에 대한 즉각적인 통찰력을 얻을 수 있다.

단점

- 결과가 부정확하다. 예외적인 측정값을 가진 일부 엔티티는 문제가 없는 것으로 판명될 수 있다. 메트릭은 휴리스틱일 뿐이며 잘못되었을 가능성이 높습니다. 또한, 메트릭은 솔루션이 리엔지니어링 목표에 기여하지 않기 때문에 해결할 가치가 없는 문제를 드러낼 수도 있다. 안타깝게도 이는 소스 코드를 분석한 후에야 알 수 있다.
- 우선순위 고려하기 어렵다. 잠재적인 문제를 식별하는 것은 쉽지만, 문제의 심각성을 평가하는 것은 정말 어려운 부분이다. 특히 리엔지니어링

프로젝트에서는 해결할 수 있는 시간보다 훨씬 더 많은 문제를 파악하게 된다. 목록의 우선순위를 정하려면 시스템과 리엔지니어링 프로젝트에 대한 충분한 이해가 필요하다.

어려움

- 데이터를 해석하기가 지루합니다. 코드의 품질을 측정하려면 여러 측정 값을 수집해야 한다. 특히 대규모 소프트웨어 시스템을 다룰 때는 이러한 다중 값 튜플을 해석하고 비교하는 작업이 상당히 지루하다. 따라서 여러 측정값을 동시에 분석할 수 있는 시각화를 사용하자.
- 전문 지식이 필요하다. 측정 데이터의 해석은 어렵고 많은 전문 지식이 필요하다. 다행히도 이러한 전문 지식의 일부는 설계 휴리스틱의 형태로 문서화되어 있으며([Rie96] [LK94] 참조), 나머지는 실무에서 습득할 수 있다.

예시

데이터베이스 분석과 *XDoctor*의 설계는 상당히 안심할 수 있었다. 몇 가지 개선 할 점이 있었지만 전반적인 품질은 상당히 좋았다. 하지만 이러한 느낌을 확인하고 싶어서 여러 품질 지표를 수집하고 시각화할 계획이다. (물론 시각화는 일반 스프레드시트로도 할 수 있지만, 이 경우에는 *CodeCrawler* 도구 [DDL99] [Lan99]를 사용하기로 했다.)

클래스 크기 개요(*Class Size Overview*). 우선 *XDoctor*를 구성하는 모든 클래스의 원시 물리적 크기에 대한 인상을 얻을 수 있다. 코드 줄 수(lines of code LOC)와 인스턴스 변수 수(number of instance variables, NIV)로 클래스 크기를 측정하고 체커보드 그래프(*checkers graph*)를 사용하여 크기의 상대적 비율을 표시한다. 이러한 그래프에서 모든 노드는 사각형으로 표시되며, 사각형의 크기는 한 크기(여기서는 LOC)에 비례하고 회색 값은 다른 크기(여기서는 NIV)에 비례한다.

그림 4.6은 *XDoctor*에 대한 체크보드 그래프를 보여준다. 그림에 따르면 몇 가지 주목할 만한 예외를 제외하고는 클래스 크기가 상당히 고르게 분포되어 있어 안심할 수 있다. 예를 들어, 1495개로 코드 줄 수가 가장 많은 B 클래스와

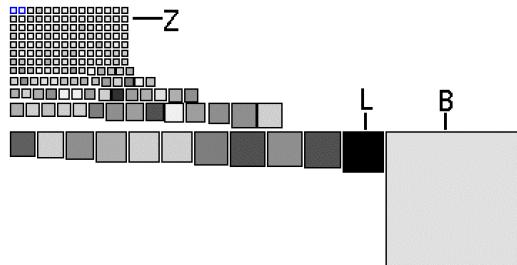


그림 4.6: 코드 줄을 노드 크기로 표현하는 것과 인스턴스 변수 수를 회색조 (gray value)로 표현하는 클래스 크기 개요를 파악할 수 있다.

인스턴스 변수가 가장 많고 코드 줄 수가 두 번째로 많은 L 클래스가 있다. Z 행의 클래스는 매우 작고 일부는 비어 있다는 점에서 예외적이다.

클래스 상속. 다음으로 상속 계층 구조의 다양한 하위 트리를 살펴봄으로써 상속이 사용되는 방식에 대한 느낌을 얻는다. 따라서 계층 구조 중첩 수준(hierarchy nesting level, HNL)과 하위 클래스 수(number of descendant classes, NDC)를 기준으로 클래스를 측정한다. 상속 트리 내 클래스의 크기를 평가하기 위해 크기 측정도 포함합니다. 따라서 메서드 수(number of methods, NOM), 인스턴스 변수 수(number of instance variables, NIV) 및 코드 줄 수(lines of code, LOC)도 수집한다. 상속 트리(*inheritance tree*)를 사용하여 다양한 하위 트리와 각 하위 트리 내의 클래스 크기 비율을 시각화한다. 이러한 트리의 모든 노드는 직사각형 모양이며 각 노드의 높이, 너비, 회색 값은 세 가지 측정값을 표시한다.

그림 4.7에서는 각 노드의 높이, 너비 및 회색 값은 NOM, NIV 및 LOC를 나타내는 *XDoctor*에 대한 상속 트리를 보여준다. 왼쪽에는 클래스의 크기가 매우 유사한 몇 가지 일반적인 상속 트리, 즉 작은 상속 트리가 있다. 예외적인 값 중 하나는 앞서 살펴본 것과 동일한 B이지만, 이제 70개의 메서드를 정의하는 대형 슈퍼클래스 A도 포함되어 있어 더욱 의심스럽다. 앞서 보셨던 L이 여기서는 단독 클래스로 나타납니다. K, F, G에 뿌리를 둔 계층 구조는 매우 흥미로워 보이는데, 4단계의 상속으로 깊게 들어가며 하나의 큰 루트 클래스와 많은 작은 하위 클래스를 가지고 있다. H와 I, 그리고 M과 N은 모두 큰 형제 클래스의 경우로, 이는 공통 슈퍼클래스로부터 상속되는 것이 너무 적다는 것을 의미할 수

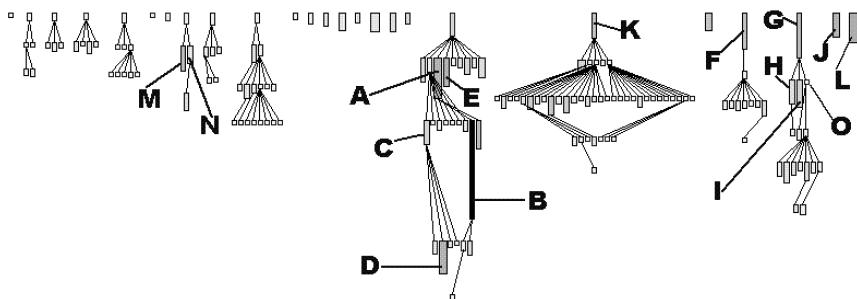


그림 4.7: 클래스 크기에 초점을 맞춘 상속 트리. 노드 너비는 인스턴스 변수의 수를, 노드 높이는 메서드의 수를, 회색 값은 코드 줄의 수를 나타낸다.

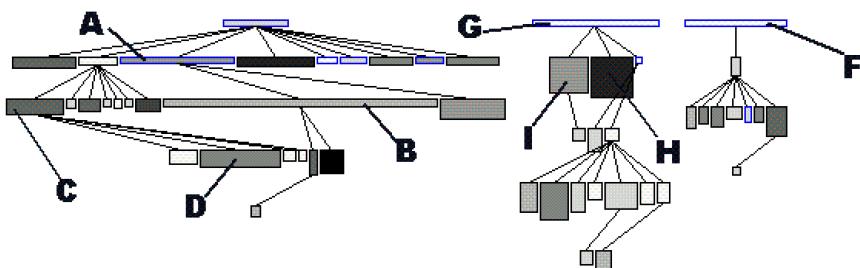


그림 4.8: 메서드 상속을 중심으로 한 상속 트리이다. 노드 너비는 추가된 메서드의 수를, 노드 높이는 재정의된 메서드의 수를, 회색조는 확장된 메서드의 수를 나타낸다.

있다. 그러나 이는 코드 탐색을 통해 확인해야 한다.

메서드 상속. 특정 상속 트리를 더 자세히 분석하려면 하위 클래스의 메서드가 상위 클래스의 메서드와 관계에 대해 조사한다. 따라서 각 클래스에 대해 슈퍼클래스에 정의된 메서드를 재정의하는 메서드의 수(number of methods overriding, NMO), 슈퍼클래스에 추가된 메서드의 수(number of methods added, NMA), 슈퍼클래스에 정의된 메서드를 확장하는 메서드의 수(number of methods extending, NME)를 보여주는 테이블을 생성한다. 여기에서도 상속 트리를 사용하여 측정값에서 예외적인 값을 식별한다.

그림 4.8 에서는 앞에서 식별한 A, G 및 F 서브트리가 표시되지만 이제 각

노드의 높이, 너비 및 회색 값은 NMO, NMA 및 NME를 나타낸다. 루트 클래스는 좁은 흰색 직사각형으로 표시되는데, 이는 루트 클래스는 재정의하거나 확장할 수 없기 때문에 정상적인 현상이다. 하위 클래스에 관해서는 두 가지 현상을 관찰할 수 있다. 한편으로 A의 서브클래스는 많이 추가하지만 재정의는 거의 하지 않는데, 이는 코드 재사용이 이 상속 트리의 주요 목적임을 시사한다. 반면에 F와 G의 서브클래스는 추가하는 메서드보다 재정의하는 메서드가 더 많은데, 이는 동작을 전문화하기 위한 후크 메서드와 상속 트리가 많음을 시사한다. 여기에서도 이러한 가정은 코드 탐색을 통해 확인해야 한다.

메서드 크기 개요(Method Size Overview). 메서드 본문에서 잠재적인 문제를 식별하는 방법의 한 예로 코드 줄 수(LOC)와 전송된 메시지 수(number of messages, MSG)의 비율을 들 수 있다. 대부분의 메서드 본문에서 이 두 측정값은 상관관계가 있지만, 상관관계가 없는 메서드는 일반적으로 특수 코드를 나타낸다.

이 상관 관계를 연구하기 위해 두 측정값을 나눌 수 있다.⁴ 그러나 그러면 구성 측정값에 대한 의미가 사라져 해석이 어려워진다. 따라서 각 메서드가 작은 사각형으로 표시되고 x, y 위치가 상관관계가 있는 측정값을 나타내는 상관관계 그래프(correlation graph)를 사용하여 관계를 시각화한다. 이러한 그래프에서 측정값이 상관관계가 있는 노드는 대각선을 중심으로 모여 있고, 예외는 대각선에서 벗어난 노드이다.

그림 4.9에서는 가로축(왼쪽에서 오른쪽)은 전송된 메시지 수를, 세로축(위에서 아래로)은 코드 줄 수를 나타내는 상관관계 그래프를 보여준다. 왼쪽 상단 모서리에서 대부분의 노드가 서로 겹쳐져 있는 큰 클러스터를 볼 수 있다. 이는 대부분의 메서드가 15줄 미만의 코드와 10개의 메시지를 전송한다는 것을 의미 하므로 안심할 수 있다. 예외는 그림의 가장자리에 나타난다. 예를 들어, 노드 A는 45줄의 코드에 99개의 메시지가 담긴 대규모 메서드이다. 노드 D(및 그 이웃 노드들)도 한 줄의 코드에 많은 메시지가 들어 있는 메서드이다. 코드 탐색을 통해 많은 메서드가 초기화 메서드라는 것을 알 수 있다. 대각선 반대편에는 16줄의 코드가 있지만 전송된 메시지가 없는 메서드를 나타내는 노드 B가 있다. 코드 탐색을 통해 메서드 본문 전체가 주석 처리된 경우임을 알 수 있다.

⁴메트릭 이론에서는 숫자의 임의 조작을 금지하고 있으므로 먼저 측정값의 규모 변경이 가능하지 확인해야 한다[FP96]. 그러나 둘 다 갯수를 세는 측정값이므로 비율 조정을 하는 나눗셈이 허용된다.

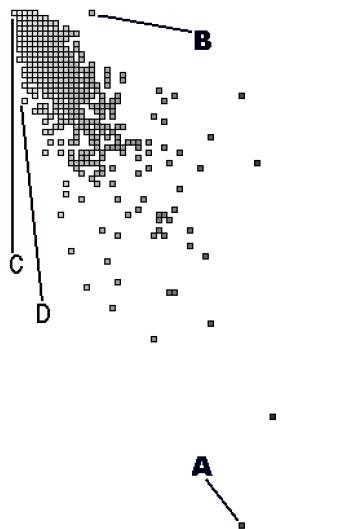


그림 4.9: 상관 그래프, x 위치는 전송된 메시지 수를 표시하고 y 위치는 코드 줄을 표시한다.

근거

측정할 수 없는 것은 제어할 수 없다.

— 톰 디마르코, [Mar82]

문헌의 여러 곳에서 소스 코드를 측정하는 것이 문제 식별에 도움이 된다고 언급되어 있다([LK94] [FP96] [MLM96] [Nes88] 참조). 이러한 실험에 사용되는 대부분의 측정 도구는 히스토그램과 키비아트 다이어그램(Kiviat diagrams)을 통해 정보를 시각화한다. 그러나 예외적인 개체를 식별할 때 임계값의 영향을 연구한 연구는 거의 없으며, 우리의 경험에 따르면 임계값은 그다지 중요하지 않다 [DD99].

안타깝게도 현재까지의 연구는 결과의 정확성에 대해 결론을 내리지 못하고 있다. 지금까지 얼마나 많은 문제가 발견되지 않았는지 계산하는 실험은 존재하지 않으며, 발견된 문제의 심각성을 평가하는 연구도 없다. 따라서 리버스 엔지니어링을 위한 지표의 신뢰성을 평가하는 것은 불가능하다.

알려진 용도

FAMOOS 프로젝트 중 한 사건은 단순한 접근법이 보다 전문적이고 복잡한 접근법을 얼마나 잘 수행할 수 있는지에 대한 사례 증거를 제공했다. 며칠 동안 한 사업부를 방문해 CodeCrawler 도구를 시연했던 적이 있다. 처음에는 개발자들이 “또 하나의 메트릭 도구(yet another metrics tool)”를 보게 될 것 같아서 상당히 회의적이었다. 첫 번째 놀라움은 첫날에 이미 결과를 보여줬을 때였다. 다른 도구는 특수한 C++ 기능과 매크로를 많이 사용하기 때문에 일반적으로 C++ 코드를 파싱하기까지 며칠의 설정 시간이 필요하다고 개발자들은 말했다. 게다가 두 번째 놀라웠던 점은 이러한 단순함 때문에 결과물의 품질이 떨어지지 않았다는 점이다. 프로그래머들은 우리가 발견한 대부분의 디자인 이상 징후를 확인하면서도 우리가 관찰한 몇 가지 사항에 흥미를 보였다. 이후 토론 과정에서 그들은 최소한 디자인 대안을 고려했다.

다음 단계

이 패턴을 적용하면 디자인 품질에 대한 전반적인 인상과 몇 가지 잠재적인 디자인 문제를 파악할 수 있다. 이 지식을 바탕으로 최소한 리엔지니어링 프로젝트의 목표가 여전히 달성 가능한지 다시 생각해 보아야 한다. 만약 목표가 달성 가능하다면, 예를 들어 책임 재배포 [p. 263] 및 조건을 다형성으로 변환 [p. 297] 의 패턴을 사용하여 이러한 디자인 문제 중 일부를 해결하고 싶을 것이다. 이러한 문제 중 일부를 해결하려면 해당 설계에 대한 자세한 이해가 필요할 수 있으며, 이는 상세 모델 캡춰 [p. 129] 의 패턴을 통해 얻을 수 있다.

제 5 장

디테일한 모델 캡처

첫 번째 컨택의 패턴은 소프트웨어 시스템에 익숙해지는 데 도움이 되었을 것 이고, 초기 이해의 패턴은 시스템에서 가장 중요한 엔터티를 이해하는 데 도움이 되었을 것이다. 이제 여러분의 최우선 과제는 리엔지니어링 작업에 중요한 시스템 부분의 세부 모델을 구축하는 것이다.

지금까지 적용한 패턴보다 훨씬 더 많은 기술 지식, 도구 사용, 노력 투자를 수반하는 디테일한 모델 캡처와 관련된 패턴이 대부분이다. 이는 당연한 일이다. 초기 이해를 쌓은 후에야 더 집중적인 노력 투자가 성과를 거둘 수 있는지 여부를 판단할 수 있기 때문이다.

포스: 주요한 요구사항

시스템에 대한 인상은 이미 가지고 있지만, 더 자세한 모델을 추출하기 어려울 수 있는 몇 가지 포스(force)가 작용하고 있다.

- 세부 사항이 중요하다. 브룩스 [Bro87]가 주장했듯이 소프트웨어 엔지니어링은 추상화 장벽이 내재되어 있기 때문에 다른 엔지니어링 분야와 다르다. 다른 공학 분야는 자연 법칙에 의존하여 관련 없는 세부 사항을 숨기지만 소프트웨어 공학은 덜 견고한 토대 위에 구축해야 한다. 따라서 세부 사항에 주의를 기울이는 것이 필수적이다. 문제는 모든 것을 조사할

수 없기 때문에 중요하지 않은 세부 사항을 어떻게 걸러낼 수 있느냐는 것이다.

- 디자인에는 암시적인 부분이 계속 존재한다. 코드를 읽다 보면 많은 디자인 결정이 분명해지지만 왜, 어떻게 이러한 결정이 내려졌는지는 명확하지 않을 것이다. 특히 어떤 디자인 결정은 쉽게 내릴 수 있었고 어떤 결정은 많은 고민을 불러일으켰는지 알기 어려울 것이다. 그럼에도 불구하고 리엔지니어링 프로젝트에서 이러한 지식이 중요한 이유는 같은 실수를 반복하지 않기 위해서이다. 따라서, 설계 근거를 발견했다면 이를 제대로 기록해 두어야 한다. 이렇게 하면 후임자가 처음부터 다시 시작해야 하는 번거로움 없이 발견한 내용을 기반으로 작업할 수 있다.
- 디자인은 진화한다. 반복 개발을 강조하는 객체 지향 개발 프로세스에서 변화는 성공적인 시스템의 필수 요소이다. 결과적으로 설계 문서는 실제 상황과 관련하여 항상 시대에 뒤떨어지게 된다. 그러나 이는 또한 변화 자체가 시스템 설계가 어떻게 그리고 왜 지금과 같이 발전했는지 이해하는 열쇠라는 것을 의미한다. 따라서 중요한 디자인 이슈가 소스 코드와 이 코드가 시간이 지남에 따라 변경된 방식에 반영될 것이라고 가정한다.
- 정적 구조(*static structure*) 대 동적 동작(*dynamic behavior*). 객체 지향 소스 코드는 어떤 클래스가 정의되어 있고 클래스 계층 구조에서 어떻게 배열되어 있는지를 알려준다. 런타임에 어떤 객체가 인스턴스화되고 시스템을 지원하기 위해 어떻게 협업하는지 파악하는 것은 훨씬 더 어렵다. 그러나 세분화된 수준에서는 특히 다형성의 사용으로 인해 후자가 전자보다 훨씬 더 관련성이 높다. 따라서 세부적인 디자인을 추출하려면 필연적으로 동적 동작을 연구해야 한다.

개요

디테일한 모델 캡처의 패턴은 코드에 숨겨진 디자인 산출물을 노출하는 데 도움이 되는 일련의 활동을 제안한다. 이러한 패턴 중 일부, 특히 코드에 대한 질문을 연결하기 은 가볍운 도구이지만 대부분은 상당한 노력이 필요하므로 적용하여 얻을 수 있을 것으로 기대하는 것이 무엇인지 신중하게 평가해야 한다.

그림 5.1은 패턴 간의 몇 가지 가능한 관계를 제안한다. 패턴 중 가장 기본적이고 적용하기 가장 쉬운 패턴은 코드에 대한 질문을 연결하기이다. 소스 코

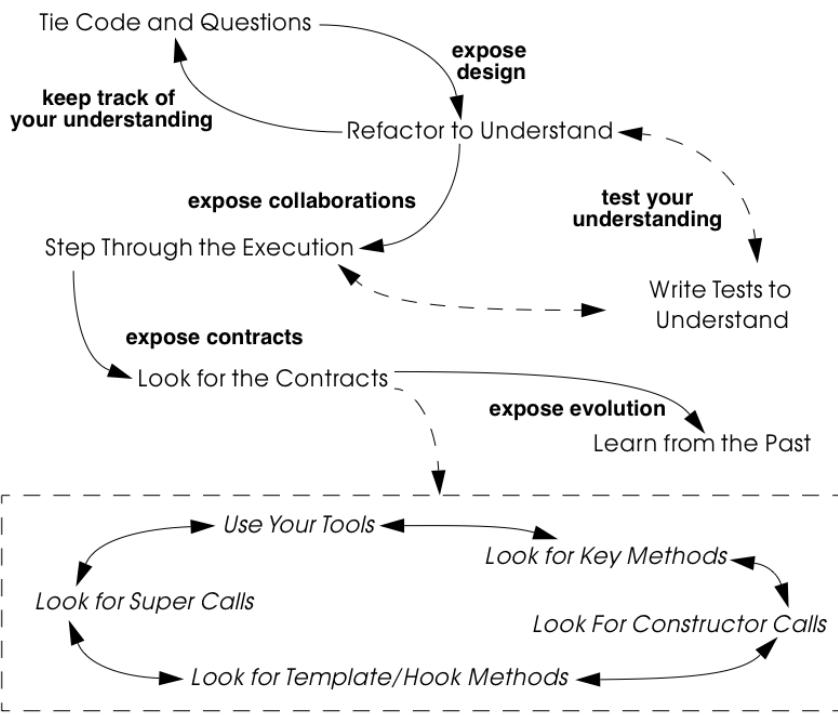


그림 5.1: 상세 모델 캡춰하기의 패턴은 소프트웨어 시스템의 설계를 노출하고 이해도를 추적하는 데 도움이 된다.

드를 살펴보면서 주석, 질문, 가설 및 수행 가능한 작업을 추적하여 소스 코드에 직접 주석이 적용되는 지점에 주석을 달아 놓자. 이 패턴은 이 클러스터의 다른 패턴과 잘 작동하며 리엔지니어링 프로젝트 전반에 걸쳐 생산적으로 적용될 수 있다.

이해하기 위해 리팩터링하기는 암호화된 코드의 설계를 노출하는 데 도움이 된다. 이 패턴의 의도는 코드 베이스 자체를 개선하는 것이 아니라 이해력을 향상시키는 데 있다는 점을 이해하는 것이 중요하다. 리팩터링의 결과를 유지하기로 결정할 수도 있지만, 이 시점에서 이것이 목표가 되어서는 안 된다. 리팩터링은 코드와 관련된 다양한 가설을 테스트하기 위한 실험으로 간주해야 한다.

소스 코드는 클래스 계층 구조에 대한 매우 정적인 보기만 제공하므로 단계별 실행해 보기 을 사용하여 어떤 객체가 인스턴스화되고 런타임에 어떻게 상호 작용하는지 알아보는 것이 유용하다.

시스템에서 클래스의 인터페이스를 추출하는 것은 매우 쉽지만, 이러한 인터페이스가 어떻게 사용될 수 있는지 또는 어떻게 사용해야 하는지에 대해 많은 것을 알려주지는 않는다. 실제로 필요한 것은 각 클래스에서 지원하는 컨트랙트 찾기 를 수행하는 것이다. 컨트랙트는 어떤 클라이언트-공급자 관계가 존재하는지, 클래스의 공용 인터페이스가 그 관계를 어떻게 지원하는지 알려준다. 관용적 코딩 관행과 디자인 패턴은 일반적으로 이러한 계약을 직접적인 방식으로 표현하므로 이를 인식할 수 있도록 스스로 훈련해야 한다.

마지막으로, 소스 코드에서 다양한 디자인 산출물을 추출할 수는 있지만 시스템이 어떻게 그런 식으로 진화했는지에 대한 통찰력을 반드시 얻을 수 있는 것은 아니다. 특히 특정 설계 결정이 정말 정당한 것인지, 아니면 자의적인 것인지 궁금할 수 있으며, 설계의 일부가 얼마나 안정적인지 궁금할 수 있다. 코드 베이스의 여러 버전을 비교하고 기능이 제거되거나 리팩터링된 부분을 집중적으로 살펴봄으로써 과거로부터 과거로부터 배우기 를 할 수 있다.

다음 단계

이제 시스템 일부의 세부 사항을 마스터했으니 테스트라는 생명 보험의 패턴을 적용하여 실제 리엔지니어링을 준비할 때이다. 특히 이해하기 위해 리팩터링하기 와 같은 패턴은 실험에 자신감을 줄 수 있으므로 이해하기 위해 테스트 작성

하기 [p. 197] 와 같은 패턴을 사용하는 것이 좋다. 또한 단계 별 실행해 보기, 컨트랙트 찾기, 과거로부터 배우기 같은 패턴은 어떤 구성 요소가 어떤 기능을 구현하는지 파악하는 데 도움이 된다. 이 지식은 구현이 아닌 인터페이스 테스트하기 [p. 187] 및 비즈니스 규칙을 테스트로 기록하기 [p. 193]에 사용해야 한다.

5.1 코드에 대한 질문을 연결하기

의도 리엔지니어링 활동과 관련된 질문과 답변을 소스 파일에 직접 저장하여 코드와 동기화한다.

문제

코드와 그에 대해 궁금한 점을 어떻게 이해하고 추적하고, 향후 코드가 발전하는 동안 이러한 코멘트를 코드와 동기화하고, 다른 팀원들과 공유하는가?

이 문제는 다음과 같은 이유로 어렵다.

- 분석 중인 시스템에 대해 알고 있는 것과 모르는 것을 작성하는 것은 지루하고 시간이 많이 걸린다.
- 시스템을 이해하는 것은 움직이는 과녁과 같아서 문서화를 최신 상태를 유지하기가 어렵습니다.
- 질문과 인사이트가 떠오르는 즉시 기록하지 않으면 이를 추적할 수 없다.
- 지식을 팀과 공유하여 그 가치를 극대화하고 싶다.
- 로그 파일, 게시판 또는 이메일 배포 목록에 질문과 답변을 기록하면 팀 내에서 지식을 전파하는 데 편리하고 팀이 이해한 내용을 편리하게 검색 할 수 있지만 코드 조각을 볼 때 어떤 질문과 답변이 해당 코드와 관련이 있는지 알기 어려울 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 코드에 애너테이션(annotation)을 달 수 있으므로 참조하는 코드 요소에 물리적으로 가깝게 이해한 내용을 기록할 수 있다.

솔루션

코드를 작업하는 동안 직면하고 있는 질문에 직접 즉시 애너테이션을 달아 보자.

원칙적으로 코드에 애너테이션을 다는 방법에는 두 가지가 있다.

- 주석 기반 에너테이션. 이 접근 방식은 프로그래밍 언어의 주석 규칙을 사용하므로 텍스트 중심 환경에 더 적합하다. 일반 주석과 에너테이션을 구분하기 위해 몇 가지 규칙이 필요하다.

```
/* #to: John #by: SD #on: 3/12/99 *****
   Screws up when we have nested I Fs. */
```

그런 다음 프로그램 환경의 일부인 기본 도구를 사용하여 주석을 검색하고 수정할 수 있다. 약간의 추가 노력으로 모든 주석 기반 에너테이션을 쿼리, 추출 및 교차 색인하는 도구를 쉽게 만들 수 있다.

- 메서드 기반 에너테이션. 이 접근 방식은 오늘날의 많은 프로그래밍 환경에서 제공하는 기능인 특정 메서드를 호출하는 메서드를 쿼리할 수 있는 가능성을 활용한다. 이 아이디어는 몇 개의 문자열을 인수로 받아들이고 메서드 본문이 비어 있는 전역 메서드를 선언하는 것입니다. 특정 코드에 주석을 달고 싶을 때마다 해당 메서드를 호출하여 주석을 매개변수로 전달하면 된다.

```
this.annotateCode("#to: John #by: SD #on: 3/12/99",
  "Screws up when we have nested I Fs.");
```

그런 다음 프로그래밍 환경의 쿼리 및 탐색 기능을 사용하여 이 특수 메서드가 호출되는 위치, 즉 주석이 발생하는 위치를 식별할 수 있다. 대부분의 프로그래밍 환경은 작은 스크립트를 통해 확장할 수 있으며, 이 경우 모든 주석에 대한 보고서를 생성하는 도구를 개발할 수 있다.

코드를 덜 변경할수록 오류가 발생할 가능성이 줄어든다는 점에 알아두자. 따라서 주석 기반 버전이 메서드 기반 버전보다 더 안전하다.

힌트

- 에너테이션은 참조하는 코드에 가능한 한 가깝게 기록하자.
- 에너테이션은 나중에 참조할 수 있도록 기록하려는 코드에 대한 질문, 가설, “할 일” 목록 또는 단순히 관찰 사항이 될 수 있다.
- 규칙을 사용해 에너테이션을 구별하자. 예를 들어 팀 맥락에서는 주석을 작성한 개발자의 이니셜과 주석을 입력한 날짜를 포함하자. 이렇게 하면 쉽게 쿼리할 수 있다.

- 회사 관행을 따르자. 댓글이 영어가 아닌 다른 언어로 작성된 경우 가능한 경우 계속 진행하자. 그러나 선택의 여지가 있다면 소스 코드가 작성된 언어(대부분의 경우 영어)와 다른 언어로 주석을 작성하지 않도록 하자. 그렇지 않으면 다른 컨텍스트를 만들어 독자가 그 사이에서 전환해야 한다.
- 질문 중 하나에 대한 답변을 발견하면 향후 독자를 위해 주석을 즉시 업데이트하거나 더 이상 관련이 없는 질문은 간단히 삭제하자.

트레이드오프

장점

- 자연적인 동기화. 코드와 주석을 물리적으로 가깝게 유지하면 코드와 주석이 동기화될 확률이 높아진다. 코드를 수정하는 동안 자연스럽게 에너테이션을 수정하거나 더 이상 사용되지 않는 경우 에너테이션을 제거할 수 있다.
- 팀 커뮤니케이션 개선. 코드에 대한 질문을 연결하기를 사용하면 팀원이 별도의 커뮤니케이션 채널(이메일, 게시판, ...)을 열지 않아도 된다. 팀원들은 어떻게든 자신이 작업하는 코드를 읽어야 하므로 코드를 커뮤니케이션 채널로 사용하여 다중화할 수 있다.
- 컨텍스트 설명 최소화. 코드에 주석을 달면 즉시 컨텍스트를 파악할 수 있다. 이렇게 하면 질문의 맥락을 설명할 필요성을 최소화하고 질문과 주석을 문서화하는 데 드는 노력을 줄일 수 있다.

단점

- 본질적으로 수동적. 입력한 질문이 반드시 누구에게 전달되는 것은 아니며, 전달되더라도 수신자가 제때 읽거나 답변할지 확신할 수 없다. 주석을 수집하고 적절한 사람에게 알리려면 추가 도구가 필요할 수도 있다.
- 프로세스 비호환성. 많은 회사가 계층적 보고 구조를 중심으로 조직되어 있다. 이러한 조직에서는 코드에 대한 질문을 연결하기가 정상적인 커뮤니케이션 채널을 우회하기 때문에 거부될 수 있다. 또한 일부 기업

관행에서는 프로그래머가 코드로 할 수 있는 작업에 강력한 제약을 가하기 때문에 이 패턴을 사용할 경우 잠재력이 제한될 수 있다. 예를 들어, 주석이 더 이상 사용되지 않을 때 제거할 수 없다면 너무 많은 노이즈가 발생하여 유용하지 않게 된다.

어려움

- 적합한 세분성 찾기. 모든 종류의 주석과 마찬가지로 적절한 양의 세부 정보를 소개하는 데 주의를 기울여야 한다. 간결하거나 난해한 주석은 금방 가치를 잃고, 장황한 주석은 독자의 주의를 코드 자체로부터 분산시킬 수 있다.
- 프로그래머가 주석을 작성하도록 동기 부여하기. 프로그래머는 일반적으로 주석이나 문서를 작성하는 것을 좋아하지 않는다. 동기를 부여하는 한 가지 방법은 코드 리뷰나 상태 회의 중에 주석을 사용하는 것이다. 이렇게 하면 주석이 즉각적인 이점을 제공한다.
- 답변의 품질. 다른 종류의 문서와 마찬가지로 잘못된 답변이 제공될 수 있다. 이러한 상황에 대처하는 한 가지 방법은 팀 내에서 주석을 정기적으로 검토하는 것이다.
- 에너테이션 제거하기. 어떤 경우에는 주석을 제거하고 싶을 수도 있다. 예를 들어, 고객에게 “깨끗한” 버전의 소스 코드를 제공해야 하거나 컴파일러가 빈 메서드 바디의 호출을 제거할 만큼 똑똑하지 않은 경우이다. 이 경우 에너테이션을 필터링할 수 있는 적절한 도구가 있는지 확인하자.

근거

이 패턴의 뿌리는 문학적 프로그래밍(*literate programming*)에 있다. [RS89][Knu92]. 문학적 프로그램은 프로그램 텍스트와 주석 사이의 일반적인 관계를 뒤집어 실행 코드가 문서 안에 포함되는 것이 아니라 그 반대가 된다. 문학적 프로그래밍은 코드와 문서를 물리적으로 가깝게 유지하는 데 중점을 둔다. 물리적으로 가까우면 코드와 문서를 동기화하는 데 드는 노력이 줄어든다.

알려진 용도

주석 기반 에너테이션. 다양한 프로그래밍 환경에서는 코드 내 에너테이션 관리를 암시적으로 지원한다. 예를 들어, Emacs에는 e-tag라는 도구가 내장되어 있어 [CRR96] 파일 집합의 상호 참조 데이터베이스를 쉽게 생성할 수 있다. 반면에 Eiffel 환경에서는 댓글(및 코드)에 다양한 수준의 가시성을 할당할 수 있다. 에너테이션에 비공개 범위를 지정하면 에너테이션을 쉽게 분리하면서도 외부에 표시되지 않도록 할 수 있다.

벨기에의 멀티미디어 분야 회사인 MediaGeniX는 체계적인 코드 태깅 메커니즘을 사용해 변경 사항에 대한 정보를 기록했다. 프로그래밍 환경은 코드가 변경될 때마다 코드 변경 동기(버그 수정, 변경 요청, 새 릴리스), 개발자 이름, 수정 시간을 설명하는 태그가 자동으로 주석이 달리는 방식으로 변경되었다. 마지막 태그만 코드에 보관되지만 구성 관리 시스템을 통해 이전 태그와 변경 사항을 검사할 수 있다. 이 태그에는 개발자가 원하는 내용을 작성할 수 있는 자유 필드도 포함되어 있어 질문과 답변에 자주 사용된다.

메서드 기반 에너테이션. Squeak 개발팀 [IKM⁺97]는 이 기술을 질문을 추적하기 위한 것이 아니라 오픈소스 개발 프로젝트에서 커뮤니케이션을 원활하게 하기 위한 수단으로 사용했다. 이 팀에서는 메서드 플래그를 호출하는 방식으로 코멘트를 도입했는데, 이는 Object 클래스에 정의되어 있다. 개발자는 flag: 메시지를 보낸 모든 발신자를 쿼리하여 주석을 찾을 수 있다. 또한 이 메서드는 심볼을 인자로 받도록 정의되어 있다. 이를 통해 예를 들어 #noteForJohn 기호로 플래그가 지정된 모든 에너테이션을 보다 구체적으로 검색할 수 있다.

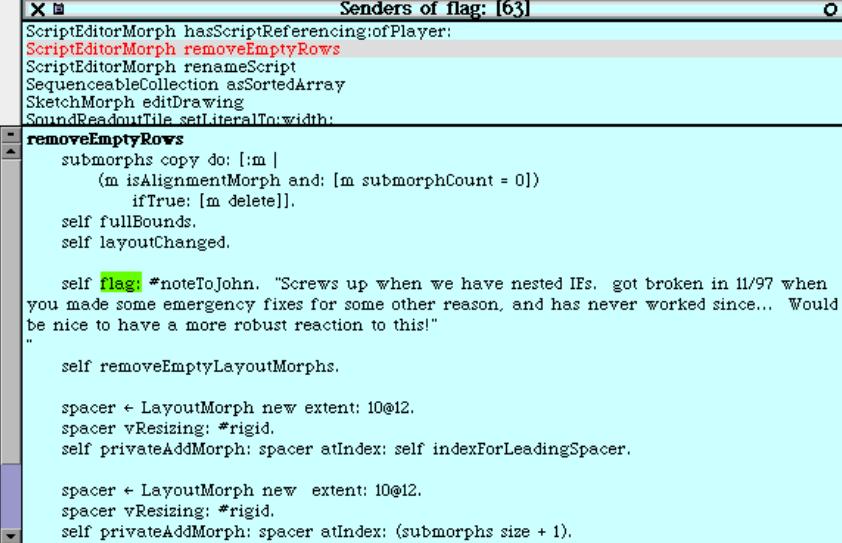
Object>>flag: aSymbol

"Send this message, with a relevant symbol as argument, to flag a message for subsequent retrieval. For example, you might put the following line in a number of messages:

```
self flag: #returnHereUrgently
```

Then, to retrieve all such messages, browse all senders of #returnHereUrgently."

그림 5.2는 Squeak2.7 환경에서 플래그: 메시지의 모든 발신자를 위쪽 창에 표시한다. 그러면 아래쪽 창에 flag: 메서드 호출이 포함된 removeEmptyRows 메서드의 코드가 강조 표시되어 표시된다. flag: 메시지는 인자 #noteToJohn과 함께 전송된다. 에너테이션의 실제 내용은 코멘트로 이어진다.



```

Senders of flag: [63]
ScriptEditorMorph hasScriptReferencing:ofPlayer:
ScriptEditorMorph removeEmptyRows
ScriptEditorMorph renameScript
SequenceableCollection asSortedArray
SketchMorph editDrawing
SoundReadoutTile setLiteralTo:width:
removeEmptyRows
  submorphs copy do: [:m |
    (m isAlignmentMorph and: [m submorphCount = 0])
      ifTrue: [m delete].
    self fullBounds.
    self layoutChanged.

  self flag: #noteToJohn. "Screws up when we have nested IFs. got broken in 11/97 when
  you made some emergency fixes for some other reason, and has never worked since... Would
  be nice to have a more robust reaction to this!"
  " self removeEmptyLayoutMorphs.

  spacer ← LayoutMorph new extent: 10@12.
  spacer vResizing: #rigid.
  self privateAddMorph: spacer atIndex: self indexForLeadingSpacer.

  spacer ← LayoutMorph new extent: 10@12.
  spacer vResizing: #rigid.
  self privateAddMorph: spacer atIndex: (submorphs size + 1).

```

그림 5.2: Squeak에서 메시지의 모든 발신자 찾기.

관련된 패턴

코드에 대한 질문을 연결하기는 이해하기 위해 리팩터링하기와 함께 잘 작동한다. 코드의 의문점은 종종 리팩터링을 통해 해결될 수 있다. 반대로 이해하기 위해 리팩터링하기를 사용하면 새로운 질문이 제기되고 에너테이션으로 입력할 수 있다.

5.2 이해하기 위해 리팩터링하기

의도 소프트웨어 시스템의 작동 방식에 대한 이해를 검증하고 반영하기 위해 소프트웨어 시스템의 일부를 반복적으로 리팩터링한다.

문제

암호화된 것 같은 코드를 어떻게 이해할 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 암호화된 것 같은 코드는 읽기 어렵기 때문에 이해하기 어렵다.
- 코드가 어떻게 작동하는지 어느 정도는 알 수 있지만 코드가 여러분의 아이디어를 반영하지 않아 확인하기가 어렵다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 코드 조각이 상대적으로 작고 경계가 명확하게 정의되어 있다.
- 개발 도구에서 빠르게 편집-컴파일 반복하기를 허용하므로 약간의 변경을 수행하고 소스 코드를 컴파일할 수 있는지 또는 테스트가 계속 실행되는지 확인할 수 있다.
- 소스 코드 엔티티 간의 종속성(예: 어떤 메서드가 주어진 작업을 호출하는지, 어떤 메서드가 주어진 속성에 액세스하는지 등)을 쿼리할 수 있는 소스 코드 브라우저가 있으므로 그 목적을 유추할 수 있다.

솔루션

반복적으로 코드의 이름을 바꾸고 리팩터링하여 의미 있는 이름을 도입하고 코드의 구조가 시스템이 실제로 수행하는 작업을 반영하는지 확인한다. 변경할 때마다 레그레션 테스트(regression tests)가 가능한 경우 이를 실행하고, 그렇지 않은 경우 자주 컴파일하여 변경 사항이 합당한지 확인한다. 코드를 리팩토링한 후 코드를 어떻게 처리할지 결정하자.

힌트

여기서 가장 중요한 목표는 코드를 개선하는 것이 아니라 시스템을 이해하는 것이다. 따라서 코드에 대한 변경 사항은 코드에 대한 이해를 테스트하기 위한 “실험(experiments)”으로 취급해야 한다. 따라서 시작하기 전에 코드의 복사본을 만들어야 한다. 코드를 리팩터링한 후에는 변경한 내용을 릴리스할 수도 있지만, 미리 결정하고 싶지는 않을 것이다. 리팩터링 실험을 통해 실제로 코드가 개선될 수도 있지만, 아직 코드를 이해하지 못하기 때문에 일을 엉망으로 만들 가능성도 똑같이 높다. 이 단계에서는 크게 중요하지 않다. 첫 경험을 하고 나면 리팩터링을 제대로 할 수 있는 더 나은 위치에 서게 될 것이다.

변경 사항이 아무것도 깨뜨리지 않았는지 확인할 수 있는 테스트가 없다면 리팩터링을 제대로 수행하기 어렵다. 적절한 테스트가 없다면 리팩터링 실험 결과를 보관하는 것을 심각하게 고려하지 않아야 한다. 그러나 이해하기 위해 테스트 작성하기를 이해하기 위해 리팩터링하기 와 함께 적용하는 것을 고려하자.

코드에서 설계 결정을 보다 명확하게 할 수 있는 리팩터링 작업을 선택해야 한다. 이 반복적인 재구조화 중에 적용되는 일반적인 리팩터링은 속성 이름 바꾸기 [p. 348], 메서드 이름 바꾸기 [p. 348] 및 메서드 추출하기 [p. 347]이다.

다음 가이드라인은 코드의 가독성을 개선하기 위해 이러한 리팩터링을 적용하는 위치와 방법을 찾는데 도움이 될 것이다. 이러한 가이드라인 중 다수는 Smalltalk 프로그래밍 [Bec97]에서 좋은 표준 관행으로 간주된다. 그러나 다른 프로그래밍 언어에도 동일하게 적용된다. 어떤 순서로든 적용될 수 있으며, 각 지침은 다른 지침을 이해하는 데 도움이 된다.

- 역할(role)을 전달하기 위해 속성(attribute)의 이름을 바꾼다. 암호화된 걸 같은 이름을 가진 속성에 집중하자. 속성의 역할을 파악하려면 모든 속성 액세스(접근자 메서드 호출 포함)를 살펴보자. 그런 다음 역할에 따라 속성(attribute)과 해당 접근자(accessor)의 이름을 바꾸고 모든 참조를 업데이트하고 시스템을 다시 컴파일하자.
- 의도를 전달하기 위해 메서드 이름 바꾸기. 의도를 드러내는 이름이 없는 메서드의 의도를 검색하려면 모든 호출과 속성 사용을 조사하고 메서드의 책임을 추론한다. 그런 다음 의도에 따라 메서드의 이름을 바꾸고 모든

호출을 업데이트한 후 시스템을 다시 컴파일한다.

- 목적(purpose)을 전달하기 위해 클래스 이름 바꾸기. 이름이 불분명한 클래스의 목적을 파악하려면 해당 클래스의 클라이언트를 조사하여 누가 해당 연산을 호출하는지 또는 누가 해당 클래스의 인스턴스를 생성하는지 조사하자. 그런 다음 목적에 따라 클래스 이름을 변경하고 모든 참조를 업데이트한 후 시스템을 다시 컴파일하자.
- 중복된 코드 제거. 중복된 코드를 발견하면 한 곳으로 리팩토링하자. 이렇게 하면 리팩토링 전에는 알아차리지 못했을 사소한 차이점을 식별하고 미묘한 디자인 문제를 드러낼 수 있다.
- 조건 분기를 메서드로 대체. 큰 가지가 있는 조건이 있는 경우 해당 가지를 새로운 (프라이빗) 메서드(private method)로 추출하자. 이러한 메서드의 이름을 지정하려면 조건을 충분히 이해하여 의도를 드러내는 이름을 선택할 수 있을 때까지 조건을 연구하자.
- 메서드 본문을 일관된 추상화 수준으로 리팩토링. 코드 블록을 구분하는 주석이 있는 긴 메서드 본문은 단일 메서드 본문의 모든 문이 동일한 수준의 추상화를 가져야 한다는 경험 법칙을 위반한다. 분리된 각 코드 블록에 새로운 (프라이빗) 메서드(private method)를 도입하여 이러한 코드를 리팩토링하고, 주석에 기록된 의도를 따라 메서드 이름을 지정하자.

트레이드오프

장점

- 디자인 노출. 리팩토링 프로세스를 통해 코드에 대한 이해도가 향상될 뿐만 아니라 코드 구조에 대한 이해도도 명확해진다. 이렇게 하면 코드에 대한 질문을 연결하기 또는 이해하기 위해 테스트 작성하기 [p. 197]를 통해 이해도를 더욱 쉽게 문서화할 수 있다.
- 증가하는 유효성 검사. 일반적으로 이해는 단일 계시의 일부로 발생하는 것이 아니라 이전 이해가 다음 반복의 기반이 되는 반복적인 과정의 결과로 발생한다. 이해를 위해 리팩토링하기는 작은 단계와 빈번한 검증(테스트를 실행하거나 자주 컴파일하여)을 강조하기 때문에 이러한 접근 방식을 권장한다.

단점

- 오류 발생 리스크. 코드를 적게 변경할수록 오류가 발생할 가능성이 줄어든다. 작은 리팩터링은 동작을 보존해야 하지만, 간단한 리팩터링이라도 코드가 깨지지 않는지 확인하는 것은 쉽지 않을 수 있다. 적절한 레그레션 테스트가 마련되어 있지 않은 경우 변경 사항을 도입하는 것이 위험하거나 필요한 테스트를 개발하는 데 비용이 많이 들 수 있다. 이러한 이유로 소프트웨어의 작업 복사본에서만 이해하기 위해 리팩터링하기를 시도하는 것이 중요하다.

어려움

- 도구 지원. 수동으로 코드를 리팩토링하는 것은 지루하고 위험할 수 있다 [FBB⁺99]. 리팩토링 브라우저 (*Refactoring Browser*)와 같은 다양한 도구가 있다. [RB97]와 같은 다양한 도구는 리팩터링 작업을 크게 간소화하며, 특히 메서드 추출하기와 같이 사소하지 않은 리팩터링을 적용하는 데 도움이 된다.
- 변경 수락. 다른 사람의 코드를 리팩토링하는 것은 자신의 코드를 리팩토링하는 것보다 훨씬 더 어려울 수 있다. 많은 회사에서 코드 소유권 문화가 강하기 때문에 다른 사람의 코드를 개선하는 것은 종종 모욕으로 간주된다. 이것이 바로 리팩터링된 버전을 반드시 다른 팀원에게 공개해서는 안 되는 이유 중 하나이다.
- 멈춤 시기. 문제를 발견했을 때 코드 변경을 중단하기 어려운 경우가 많다. 여기서 여러분의 일차적인 목표는 시스템을 이해하는 것임을 기억하자. 그 목표를 달성했다면 이제 멈춰야 할 때이다.

알려진 용도

돈 로버츠와 존 브랜트는 ESUG '97과 Smalltalk Solutions '97에서 리팩터링 브라우저를 시연하는 동안 이해하기 위해 리팩터링하기라는 용어를 만들었다. 그들은 코드의 이름을 바꾸고 리팩토링하여 알고리즘을 점차적으로 이해하는 과정을 보여주었다. 이후 이 패턴을 반복하는 동안 코드가 서서히 이해되기 시작했고 디자인이 점차 코드에 명시적으로 드러나기 시작했다.

저희는 이 패턴을 FAMOOS 사례 연구에 직접 적용했다. 우리는 약 3000 줄의 C++로 이루어진 하나의 메서드를 이해해야 했는데, 이 메서드는 중첩된 조건 명령을 가졌다. 먼저 리프 조건 분기를 메서드로 대체하면서 중첩 구조를 점차적으로 파악해 나갔다. 몇 번의 반복 끝에 이 메서드가 실제로 작은 명령어에 대한 완전한 파서를 구현하고 있다는 것을 발견했다.

해리 스니드는 모든 goto 문을 제거하여 대규모 Cobol 프로그램을 리팩터링한 여러 리엔지니어링 프로젝트에 대해 리포트를 만들었다. 그러나 나중에 개발자들이 그의 변경 사항을 거부했기 때문에 [Sne99]를 다시 goto문을 도입할 수밖에 없었다.

관련된 패턴

“가구 배치”(*Arranging the Furniture*) [Tay00]는 새 프로젝트에 새로 들어온 사람이 편안하게 시작할 수 있도록 도와주는 패턴이다. 이 패턴의 해결책은 “코드를 깔끔하게 정리하여 ‘적용’하도록 도입하는 사람이 유도해야 한다”이다.

다음 단계

이해하기 위해 리팩터링하기는 코드에 대한 질문을 연결하기와 함께 잘 작동한다. 리팩터링은 단순히 코드에 주석을 다는 것보다 구현하는 데 비용이 많이 들기 때문에 먼저 주석을 다는 작업을 한 다음 리팩터링하자. 또한 리팩터링할 때 이해할 수 있는 테스트 작성하기 [p. 197]를 고려하자. 테스트는 소프트웨어 산출물의 작동 방식에 대한 이해를 문서화하고 리팩터링은 디자인을 노출하는데 도움이 되므로 이 두 가지 활동은 서로를 강화한다. 또한 테스트는 리팩터링으로 인해 문제가 발생하지 않았는지 확인하는 데 도움이 된다.

이해를 위해 리팩터링하기를 완료했다면, 변경 사항을 어떻게 처리할지 결정해야 한다. 실험 코드를 폐기하는 경우 코드에 대한 질문 연결하기를 적용하여 습득한 지식으로 코드 베이스에 주석을 달아야 한다.

5.3 단계 별 실행해 보기

의도 시스템의 객체들이 어떻게 협업하는지 디버거에서 예제를 단계별로 실행해 보고 이해한다.

문제

런타임에 어떤 객체가 인스턴스화되고 어떻게 협업하는지 어떻게 알 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 소스 코드는 런타임에 인스턴스화된 객체와 객체 간 상호 작용 방식이 아닌 클래스 계층 구조를 노출한다.
- 공동 작업은 일반적으로 코드를 통해 분산되어 있다. 시스템에서 어떤 클래스와 메서드가 정의되어 있는지 쉽게 확인할 수 있지만, 소스 코드만으로는 어떤 이벤트 시퀀스로 인해 객체가 생성되거나 메서드가 호출되는지 알기 어려울 수 있다.
- 다형성(polymorphism)이 있는 경우 어떤 객체가 어떤 서비스 제공자의 클라이언트인지 구분하기가 특히 어려울 수 있다. 한 객체가 다른 객체가 제공하는 특정 인터페이스를 사용한다고 해서 전자가 실제로 후자의 클라이언트라는 의미는 아니다.
- 코드를 읽는다고 해서 어떤 구체적인 시나리오가 발생할 수 있는지 알 수 없다. 실제 실행 흐름은 모든 참여 객체의 내부 상태에 따라 달라지며 이는 소스 코드에서 직접 유추할 수 없다.
- 소스 코드는 어떤 객체가 수명이 긴지, 어떤 객체가 일시적인지(즉, 단일 메서드의 실행에 국한되는지) 알려주지 않는다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 몇 가지 일반적인 사용 시나리오를 알고 있다.
- 디버거 내에서 코드를 실행할 수 있다.
- 시스템의 일부에 주의를 집중하고 있다.

솔루션

각 시나리오를 실행하고 디버거를 사용하여 코드를 단계별로 살펴보자. 어떤 객체가 협업하고 어떻게 인스턴스화되는지 관찰하자. 그런 다음, 이러한 관찰을 일반화하고 나중에 참조할 수 있도록 코드에 대한 질문을 연결하기 및 비즈니스 규칙을 테스트로 기록하기 [p. 193]를 사용하여 알게된 지식을 기록해 두자.

힌트

실행 중인 시스템의 모든 문을 일일이 살펴보기에는 시간이 너무 많이 걸린다. 여기서는 이해하기 어려운 시스템의 특정 측면에 초점을 맞추고 있다고 가정 한다.

- 시스템이 관심 있는 코드에 들어갈 때 실행을 중단하도록 브레이크포인트(breakpoint)를 설정한다.
- 개체의 내부 상태(internal state)를 변경하여 대체 실행 경로가 어떻게 트리거되는지 확인한다.
- 현재 실행 스택에 있는 메서드 다시 시작(restart a method)을 사용하여 유사한 시나리오를 빠르게 확인할 수 있다.

트레이드오프

장점

- 현실적 뷰. 실행 중인 프로그램을 단계별로 살펴보면 시나리오가 어떻게 전개되는지 정확하게 파악할 수 있다. 또한 관련된 객체의 내부 상태를 검사하고, 새로운 객체가 어떻게 생성되는지 확인하고, 어떤 상황에서 어떤 객체가 협업하는지 관찰할 수 있다.
- 복잡성 다루기. 소규모로 소스 코드를 분석하여 객체 협업을 유추할 수 있다. 예를 들어 슬라이싱 도구는 소스 코드의 어떤 문이 주어진 변수의 영향을 받는지 알려줄 수 있다. 그러나 크고 복잡한 시스템의 경우 가능성과 상호 작용의 수가 너무 많다. 따라서 객체가 어떻게 협업하는지 알아내는 유일한 합리적인 방법은 실행 추적을 연구하는 것이다.

단점

- 시나리오 기반. 제한된 시나리오 집합으로 제한해야 하므로 관찰된 개체-협업은 반드시 불완전할 수밖에 없다. 물론 대표 시나리오를 선택하기 위해 최선을 다해야 한다. 안타깝게도 이러한 선택은 다시 원점으로 돌아 가게 된다. 왜냐하면 대표적인 시나리오 집합이 있는지 확인하는 유일한 방법은 가능한 모든 개체 협업을 포함하는지 확인하는 것이기 때문이다.
- 제한된 적용가능성(*applicability*). 시간이 중요한 역할을 하는 시스템의 경우 실행을 단계별로 진행하면 시스템의 동작을 비현실적으로 파악할 수 있다. 더 나쁜 것은 동시 또는 분산 시스템의 경우 동시 코드를 단계별로 살펴본다는 사실만으로도 시스템 실행 자체가 교란될 수 있다는 점 입이다. 따라서 양자 입자의 정확한 위치를 결정하면 해당 입자에 대한 다른 속성이 불확실해지는 하이젠베르크의 불확실성 실험(Heisenberg's uncertainty experiments)과 같은 효과를 얻을 수 있다.

어려움

- 도구 의존성. 단계별 실행해 보기 를 위한 좋은 디버거가 있어야 한다. 중단점을 동적으로 설정하고 제거할 수 있어야 할 뿐만 아니라 관련된 객체의 상태를 검사할 수 있는 수단도 제공해야 한다. 또한 대체 경로를 쉽게 확인할 수 있도록 디버거는 객체의 내부 상태를 변경하거나 현재 실행 스택에 있는 메서드를 다시 시작할 수도 있어야 한다.

다음 단계

단계별 실행해 보기 (아마도 데모 중 인터뷰하기 [p. 75]에서 유추 가능)를 수행하려면 구체적인 시나리오가 필요하다. 이러한 시나리오를 테스트 케이스로 인코딩하는 것을 고려해 보자. 그런 다음 이해하기 위해 테스트 작성하기 을 반복적으로 수행하면 협업 객체의 상태에 대한 통찰력을 구체적인 테스트로 공식화할 수 있으므로 이해하기 위해 테스트 작성하기 [p. 197] 을 반복적으로 수행할 수 있다.

단계 별 실행해 보기 을 사용할 때 협업하는 개체가 서로의 인터페이스를 사용하는 방식을 계속 주시하는 것이 좋다. 그 후에는 컨트랙트 찾기 에 습득한

지식을 활용할 수 있다.

5.4 컨트랙트 찾기

의도 클라이언트가 현재 사용하는 방식을 연구하여 클래스 인터페이스(class interface)의 적절한 사용법을 추론한다.

문제

클래스가 어떤 컨트랙트(contracts)를 지원하는지 어떻게 결정하는가? 즉, 클래스가 의도한 대로 작동하기 위해 클라이언트 클래스에서 무엇을 기대하는지 어떻게 알 수 있을까?

이 문제는 다음과 같은 이유로 어렵다.

- 클라이언트/공급자 관계와 계약은 코드에 암시적으로만 존재한다. 코드에서 인터페이스를 추출하기는 쉽지만 인터페이스를 올바르게 사용하는 방법을 반드시 알려주지는 않는다. 명시적으로 문서화되어 있지 않으면 (a) 메서드가 호출되어야 하는 적절한 순서, (b) 제공되어야 하는 유효한 매개변수, (c) 어떤 메서드가 어떤 클라이언트에서 호출되어야 하는지, (d) 서브클래스에 의해 재정의되어야 하는 메서드는 무엇인지 추측하기 어려울 수 있다.
- 입력 및 범위 지정 규칙은 종종 프로그래머가 공급자의 인터페이스를 손상시키도록 강요한다. 또한 캡슐화 구조(encapsulation)(예: 퍼블릭/프라이빗 선언)는 구현 문제에 대처하기 위해 자주 오용된다. 예를 들어 데이터베이스 및 사용자 인터페이스 툴킷에는 공용 접근자 메서드가 필요한 경우가 많다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 시스템 구조에 대해 잘 이해하고 있다.(예: 초기 이해를 통해 얻은 정보) 따라서 주요 클래스와 덜 중요한 클래스를 구분할 수 있다.
- 클래스가 클라이언트와 그 하위 클래스에서 제대로 사용되고 있음을 신뢰한다.

솔루션

클라이언트가 클래스 인터페이스를 사용하는 방식을 드러내는 일반적인 프로그래밍 관용구를 찾아보자. 컨트랙트, 즉 클래스가 클라이언트에게 기대하는 바를 명시적으로 선언하는 형태로 관찰한 내용을 일반화하자.

힌트

여기서 목표는 클래스에 대한 인터페이스가 다른 클라이언트에서 사용되는 방식을 노출하여 클래스가 어떻게 협업하는지 이해하는 것이다. 코드를 철저하게 분석하면 지칠 수 있으므로 코드의 모든 줄을 살펴보지 않고 컨트랙트를 노출할 수 있는 방법이 필요하다.

컨트랙트는 코드에 암시적으로만 존재하지만, 대부분의 경우 코드에는 다양한 클래스 간에 특정 관계가 존재한다는 힌트가 있을 것이다. 이러한 힌트는 사용 중인 프로그래밍 언어의 관용구, 개발팀에서 사용하는 규칙, 심지어 일반적인 디자인 패턴으로 나타날 수도 있다.

정확히 무엇을 찾아야 하는지는 상황에 따라 다르지만 일반적으로 유용한 몇 가지 예는 다음과 같다.

도구 사용. 클래스 간의 관계에 대한 개요를 파악하려면 사용 가능한 도구를 최대한 활용하자. 수작업으로 코드를 분석하여 클래스 간의 관계를 유추할 수도 있지만, 두 개 이상의 클래스에 적용하면 그 과정이 지루하다.

많은 조직에서 설계 추출 또는 왕복 엔지니어링 도구를 사용하여 시스템을 문서화한다. 너무 많은 시간을 투자하지 않고도 분석 중인 시스템의 초안 보기 를 쉽게 생성할 수 있다. 하지만 관련 없는 세부 정보가 포함된 “박스와 화살표 (boxes and arrows)” 다이어그램이 넘쳐날 수 있다는 점에 대비하자. 하지만 디자인 추출 도구를 사용하면 필터와 코드 해석 방법을 지정할 수 있으므로 매핑이 정의되면 여러 추출에 걸쳐 재사용할 수 있다.

디자인 개요는 계층 구조의 주요 클래스(즉, 다른 많은 클래스가 상속하는 추상 클래스), 부분-전체 관계 등을 식별하는 데 도움이 될 수 있다.

주요 메서드 찾기. 가장 중요한 메서드에 집중하자. 시스템에 대한 지식이 있으면 시그니처(signature)를 기반으로 주요 메서드를 알아볼 수 있다.

- 메서드 이름. 주요 메서드에는 의도가 드러나는 이름 [Bec97]가 있을 가능성이 높다.
- 매개 변수 타입. 시스템의 주요 클래스에 해당하는 유형을 가진 매개변수를 받는 메서드는 중요할 가능성이 높다.
- 반복 매개 변수 타입. 매개변수는 객체 간의 일시적인 연결을 나타낸다. 메서드 서명에서 동일한 매개변수 타입이 자주 반복되는 경우 중요한 연관성을 나타낼 가능성이 높다.

생성자 호출 찾기. 특정 클래스의 객체를 인스턴스화하는 방법과 시기를 이해하려면 생성자를 호출하는 다른 클래스에서 메서드를 찾아보자.

특히 생성자에 전달되는 매개변수와 매개변수가 공유되는지 여부에 주의를 기울이자. 이렇게 하면 어떤 인스턴스 변수가 생성된 객체의 일부인지, 어떤 변수가 공유 객체에 대한 참조에 불과한지를 파악하는 데 도움이 된다.

생성자 메서드를 호출하면 **부분-전체 관계**(part-whole relationship)가 드러날 수 있다. 클라이언트가 생성자 메서드의 결과를 속성에 저장하면 이 클라이언트는 아마도 전체로 사용될 것이다. 반면에 클라이언트가 생성자 메서드에 인자로 자신을 전달하면 클라이언트는 부분으로 작동할 가능성이 높다.

생성자 메서드를 호출하면 팩토리 메서드 [p. 350] 또는 심지어 추상 팩토리 [p. 349]가 노출될 수도 있다. 만약 그렇다면 연구 중인 클래스를 서브클래싱하여 시스템을 확장할 수 있다는 것을 알 수 있다.

템플릿/후크 메서드 찾기. 클래스를 전문화하는 방법을 이해하려면 서브클래스에 의해 재정의되는 (프로텍티드) 메서드를 찾고, 이를 호출하는 공용 메서드를 식별하자. 호출하는 공용 메서드는 거의 확실하게 템플릿 메서드 [p. 352]입니다. 클래스 계층구조를 확인하여 재정의된 메서드가 추상화(abstract)인지, 이 경우 서브클래스가 구현해야 하는지, 아니면 기본 구현이 제공되는지 확인한다. 후자의 경우 **후크 메서드(hook method)**이며, 서브클래스는 이를 재정의하거나 기본값에 만족할 수 있다.

각 템플릿 메서드에 대해 다른 혹 메서드를 나타낼 가능성이 있으므로 호출하는 다른 모든 메서드를 확인하자.

슈퍼 호출 찾기. 클래스가 서브클래스에 대해 어떤 가정을 하는지 이해하려면 슈퍼 호출을 찾아보자. 슈퍼 호출은 서브클래스가 상속된 메서드를 애드혹(ad-

hoc) 방식으로 확장하는 데 사용할 수 있다. 그러나 수퍼 호출은 재정의된 메서드가 수퍼 호출에 의해 명시적으로 호출되지 않는 한 특정 메서드가 서브클래스에 의해 재정의되어서는 안 된다는 사실을 표현하는 경우가 매우 많다.

이 관용구는 Java에서 여러 생성자를 정의하는 클래스에서 많이 사용된다. 예를 들어 `java.lang.Exception`의 모든 서브클래스는 기본 생성자와 `String` 인자를 받는 생성자를 모두 정의해야 한다. 이러한 생성자는 예외 서브클래스가 올바르게 초기화되도록 수퍼 생성자를 호출하는 것 외에는 특별한 작업을 수행하지 않아야 한다.

트레이드오프

장점

- 신뢰. 문서보다 소스 코드를 더 신뢰할 수 있다.

단점

- 남아 있는 나쁜 습관. 코드에 특정 관행이 나타난다고 해서 그것이 올바른 방법이라는 의미는 아니다. 클라이언트와 서브클래스가 준수하는 계약이 클래스가 실제로 지원하는 계약과 반드시 일치하는 것은 아니다.
- 노이즈. 소스 코드를 탐색하는 것은 채굴과 비슷하다. 가끔씩 보석을 발견할 수 있지만 먼저 많은 흙을 파헤쳐야 한다. 관용적 사용법에 주의를 집중하면 노이즈 요소를 상당 부분 줄일 수 있을 것이다.

알려진 용도

많은 연구자들이 클라이언트가 클래스 인터페이스를 어떻게 사용하는지 분석하는 방법을 연구했다. 예를 들어 브라운 [Bro96], 플로레인 [FMvW97], 우이츠 [Wuy98]는 모두 코드에서 디자인 패턴의 증상을 찾을 수 있다는 것을 보여주었다. 또한 재정의된 메서드 분석을 기반으로 후크 메서드를 반자동으로 탐지하는 기법에 대한 쉐워 [SRMK99]의 보고서도 있다. 후자의 기법은 클래스 계층 구조를 시각화하고 많은 메서드가 재정의되어 후크 메서드를 정의할 가능성이 높은

클래스를 강조하는 특별한 방식으로 인해 확장성이 매우 뛰어나다. 또한 스템
야트 외 저자들 [SLMD96]은 서브클래스가 슈퍼클래스에 어떻게 의존하는지를
포착하고(이러한 의존성을 재사용 계약(*reuse contracts*)이라고 명명했다) 나중
에 슈퍼클래스가 변경될 때 잠재적인 충돌을 감지하는 것이 가능하다는 것을
보여주었다.

다음 단계

식별한 컨트랙트의 유효성을 검사하는 한 가지 방법은 단계별 실행해 보기 을
사용하는 것이다. 단계 별 실행해 보기 은 사용하면 다양한 개체 간의 협업을
발견할 수 있다. 이 시점에서 해당 협업을 관리하는 컨트랙트 찾기 를 사용할 수
있다.

코드가 읽기 어려운 경우 이해하기 위해 리팩터링하기 를 먼저 수행한 후
컨트랙트 찾기 를 수행할 수 있다. 컨트랙트가 어떻게 현재 상태로 진화했는지
이해하려면 과거로부터 배우기 를 활용할 수 있다.

5.5 과거로부터 배우기

의도 시스템의 이전 버전을 비교하여 디자인에 대한 통찰력을 얻는다.

문제

시스템이 왜 그렇게 설계되었는지 어떻게 알 수 있을까? 시스템의 어떤 부분이 안정적이고 어떤 부분이 안정적이지 않은지 어떻게 알 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 개발 과정에서 얻은 교훈은 문서로 기록되는 경우가 거의 없다. 게다가 개발자의 디자인 결정에 대한 인식과 기억은 시간이 지남에 따라 왜곡되는 경향이 있다. 따라서 소스 코드에만 의존할 수밖에 없고 거기서부터 학습 과정을 재구성해야 한다.
- 시스템이 크고 여러 버전으로 출시되었기 때문에 분석할 소스 코드의 양이 많다. 텍스트 비교 도구(예: Unix diff)는 처리하는 크기에 맞게 확장되지 않는다.
- 두 후속 릴리스 사이의 변경 사항을 식별하는 도구가 있더라도 대부분의 변경 사항은 새로운 기능 추가와 관련이 있다. 학습 과정을 재구성하고 이것이 수업 디자인에 어떻게 통합되었는지에 대한 주된 관심은 예전 기능에 어떤 일이 일어났는지에 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 여러분은 시스템 구조에 대해 잘 이해하고 있다(예: 초기 이해를 통해 얻은 정보), 따라서 적절한 하위 시스템에 집중할 수 있다.
- 시스템의 후속 릴리즈에 액세스할 수 있으므로 버전별 소스 코드를 비교하여 변경 사항을 재구성할 수 있다.
- 개별 소스 코드 엔티티에 어떤 일이 발생했는지 조사할 수 있는 수단이 있다. 예를 들어, 소스 코드의 엔티티 크기를 정량화하고 이 수치를 비교의 기준으로 사용할 수 있는 메트릭 도구(metrics tools)를 사용할 수 있다. 또는 소스 코드 엔티티의 특정 변경 사항에 대한 정보를 제공할 수 있는 구성 관리 시스템을 사용할 수도 있다.

- 사용 중인 구현 언어에 대한 충분한 리팩터링에 대한 전문 지식이 있으므로 소스 코드에 대한 리팩터링의 영향을 통해 리팩터링을 인식할 수 있다. 또한 어떤 리팩터링이 적용되었는지 알고 나면 이 전문 지식을 바탕으로 기본 설계 근거를 추측할 수 있다.
- 특정 연산을 호출하는 메서드(다형성 연산의 경우에도)를 퀘리할 수 있는 소스 코드 브라우저가 있으므로 클래스 간의 종속성을 파악하고 리팩터링의 영향을 받는 방식을 조사할 수 있다.

솔루션

메트릭 (*metrics*) 또는 구성 관리 도구를 사용하여 기능이 제거된(*removed*) 엔티티를 찾으면 이러한 엔티티는 통합 설계의 신호이므로 이를 찾아보자. 또한 자주 변경되는 엔티티는 디자인의 불안정한 부분을 가리킬 수 있으므로 자주 변경되는 엔티티를 찾아보자.

힌트

여러분의 목표는 시스템이 어떻게 그리고 왜 현재 상태로 발전해 왔는지를 파악하는 것이다. 특히 시스템의 어느 부분이 크게 리팩터링되었는지, 어느 부분이 안정화되었는지, 어느 부분이 활동의 핫스팟인지 파악하고 싶을 것이다.

소프트웨어 시스템에서 크게 확장된 부분은 설계의 진화가 아니라 단순히 성장의 신호일 뿐이다. 반면에 소프트웨어가 제거된(*removed*) 부분은 시스템 설계가 변경되었다는 신호이다. 설계가 어떻게 변경되었는지 이해하면 설계의 안정성에 대한 통찰력을 얻을 수 있다.

불안정한 디자인. 시스템의 동일한 부분에서 반복적인 성장과 리팩터링이 감지되면 설계가 불안정하다는 신호일 수 있다. 습관적으로 발생하는 변경 및 확장을 더 잘 수용하기 위해 해당 부분을 재설계할 기회가 있음을 나타낼 수 있다.

성숙하고 안정적인 설계. 성숙한 하위 시스템은 어느 정도 성장과 리팩토링을 거친 후 안정기에 접어든다. 초기 버전의 서브시스템은 성장과 리팩터링을 거친 후 새로운 클래스와 서브클래스만 추가되는 시기를 거친다. 계층 구조가 안정화되면 계층 구조의 최상단에 있는 클래스는 적당한 성장만 보일 뿐 리팩터링은 거의 이루어지지 않는다.

트레이드오프

장점

- 중요한 디자인 아티팩트에 집중. 변경 사항은 디자인이 확장되거나 통합되는 부분을 가리키며, 이는 다시 기본 디자인 근거에 대한 통찰력을 제공하기 때문이다.
- 시스템에 대한 편견 없는 시각을 제공. 소프트웨어에서 기대할 수 있는 것에 대한 가정을 공식화할 필요가 없기 때문이다(디자인 추측하기 [p. 109]과 같은 하향식 기법과는 대조적임).

단점

- 상당한 경험 필요. 리버스 엔지니어가 리팩터링이 특정 구현 언어의 코딩 관용구와 어떻게 상호 작용하는지 잘 알고 있어야 한다는 의미에서 상당한 경험이 필요하다.
- 상당한 도구 지원이 필요. 특히 (a) 메트릭 도구 또는 구성 관리 시스템, (b) 다형성 메서드 호출을 역추적할 수 있는 코드 브라우저가 필요하다.

어려움

- 변경 사항이 많을 경우 부정확함. 동일한 코드에 너무 많은 변경 사항이 적용되면 변경 프로세스를 재구성하기가 어려워지기 때문이다.
- 이름 변경에 민감함. 이름을 통해 클래스와 메서드를 식별하는 경우¹ 그렇다. 그러면 이름 변경 작업이 제거 및 추가 작업으로 표시되어 데이터 해석이 더 어려워진다.

근거

많은 객체 지향 시스템은 반복적 개발과 점진적 개발의 조합을 통해 탄생했다 ([Boo94] [GR95] [JGJ97] [Ree96] 참조). 즉, 원래 개발팀은 문제 영역에 대한

¹ 일부 구성 관리 시스템은 이름 변경 작업을 추적하여 문제를 완화할 수 있다.

전문 지식이 부족하다는 것을 인식하고 각 학습 단계가 새로운 시스템 릴리스로 이어지는 학습 프로세스에 투자했다. 이러한 학습 과정을 재구성하는 것은 시스템 설계에 구현된 근거를 이해하는 데 도움이 되므로 가치가 있다.

학습 과정을 재구성하는 한 가지 방법은 초기 단계를 복구하는 것이다. 객체 지향 용어로 이러한 단계를 리팩터링이라고 하며, 따라서 이 패턴은 과거에 적용되었던 리팩터링을 복구하는 방법을 알려준다. 이 기술 자체는 소스 코드의 두 후속 릴리스를 비교하여 크기가 줄어든 엔티티를 식별하는데, 이는 다른 곳으로 이동한 기능의 일반적인 증상이기 때문이다.

알려진 용도

저희는 Smalltalk로 구현된 세 개의 중간 규모 시스템에서 실험을 진행했다. citeDeme00a에 보고된 바와 같이, 이 사례 연구는 몇 가지 간단한 휴리스틱을 통해 기능이 제거된 시스템 부분에 주의를 집중함으로써 리버스 엔지니어링 프로세스를 지원할 수 있음을 시사한다. 예를 들어 클래스가 분할된 위치나 메서드가 형제 클래스로 이동된 위치를 감지할 수 있다. 물론 이러한 리팩터링은 리팩터링의 의도를 추측하기 위해 더 자세히 조사해야 한다. 결코 쉬운 일은 아니지만 경험상 그만한 가치가 있는 것으로 입증되었다. 예를 들어 한 특별한 사례에서 메서드가 형제 클래스로 이동된 여러 클래스를 발견했다. 자세히 살펴보니 리엔지니어가 순환 종속성을 끊기 위해 이러한 메서드를 옮겼고 실제로는 레이어를 도입한 것이었다.

다른 연구자들도 리버스 엔지니어링 프로세스를 지원하기 위한 변경 사항을 검토한 결과를 보고했다. 예를 들어, 볼외 연구자는 코드 뷰에 코드 수명을 나타내는 색상으로 주석을 달았다 [BE96]. 반면에 자자예리는 시스템의 소프트웨어 릴리스 이력을 조사할 때 3차원 시각적 표현을 사용한다 [JGR99]. 또한 같은 사람들이 소프트웨어 모듈 간의 논리적 종속성을 감지하기 위해 어떤 변경 요청이 어떤 소프트웨어 모듈에 영향을 미치는지 조사했다 [GHJ98].

다음 단계

이제 디자인에서 안정적인 부분을 발견했으므로 이를 재사용하고 싶을 것이다. 이 경우 몇 가지 예방 조치를 취하자. 먼저 해당 부분의 인터페이스를 문서화하

고(컨트랙트 찾기 참조), 해당 테스트 케이스를 작성하세요(구현이 아닌 인터페이스 테스트하기 [p. 187] 참조).

반면에 설계의 불안정한 부분은 삭제해야 한다. 그럼에도 불구하고 불안정한 부분이 리엔지니어링 프로젝트에 중요해 보인다면 어떤 변경 요청이 불안정성을 유발했는지 찾아야 한다. 이 경우 유지보수자와 담소나누기 [p. 49] 또는 데모 중 인터뷰하기 [p. 75]에서 이 지식을 바탕으로 들어오는 변경 요청의 종류에 더 적합하도록 해당 부분을 어떻게 재구성할지 결정하자.

제 III 편

리엔지니어링

제 6 장

테스트라는 생명보험

여러분은 리엔지니어링 프로젝트의 시작 단계에 있다. 여러분은 소중한 레거시 시스템의 많은 부분에 대해 근본적인 수술을 수행해야 한다는 것을 알고 있다. 비즈니스가 의존하는 시스템을 변경할 때 발생하는 위험, 즉 기존에 작동하던 기능이 중단될 위험, 잘못된 작업에 너무 많은 노력을 기울일 위험, 필요한 새 기능을 시스템에 통합하지 못할 위험, 유지보수 비용이 증가할 위험을 어떻게 최소화할 수 있을지 고민하고 있을 것이다.

이 클러스터에 제시된 패턴은 리엔지니어링 변경으로 인한 위험을 줄이기 위해 리엔지니어링 컨텍스트에서 테스트를 사용하는 효과적인 방법을 제시 한다.

경고. 테스트는 이 장에서 몇 페이지에 걸쳐 자세히 다룰 수 없을 정도로 풍부하고 중요한 주제이다. 리엔지니어링 프로젝트와 특히 관련이 있는 몇 가지 중요한 테스트 패턴을 파악하고 몇 가지 주요 이슈를 간략하게만 설명할 것이다. 예를 들어, 바인더는 책 전체를 객체 지향 시스템 테스트에 할애하고 있다 [Bin99].

포스: 주요한 요구사항

이러한 패턴은 레거시 시스템의 진화에 대한 다양한 위험 요소와 관련된 공통적인 포스를 공유한다. 각 패턴은 노력과 리스크 사이의 일정한 균형을 달성하기

위해 이러한 주요 요구사항 중 일부를 해결한다.

리엔지니어링 포스

- 레거시 시스템에는 테스트 절차가 정의되어 있지 않은 경우가 많다.
- 새로운 버그가 발생하지 않고 시스템의 일부를 변경하는 것은 어려운 작업이다.

시스템 개발 포스

- 시스템의 모든 측면을 테스트할 수 있는 것은 아니다.
- 동시성(concurrency) 및 사용자 인터페이스와 같은 특정 측면은 테스트하기 어렵다.
- 시간에 쫓기면 테스트 작성은 항상 가장 먼저 제거되는 작업이다.
- 소수의 사람에게만 시스템에 대한 모든 지식이 집중되어 있으면 프로젝트의 미래에 큰 위험을 초래할 수 있다.

인적 측면 포스 (고객)

- 고객은 궁극적으로 테스트 비용이 아니라 시스템의 새로운 기능에 대한 비용을 지불한다.
- 불안정하거나 버그가 있는 시스템은 고객에게 허용되지 않는다.

인적 측면 포스 (개발자)

- 프로그래머는 좋은 코드를 작성하기 때문에 테스트가 필요하지 않다고 생각한다.
- 프로그래머는 한 달 후 프로젝트를 떠날 수도 있기 때문에 장기적인 목표에 대해 동기 부여받지 않는다.
- 프로그래머는 문제를 파악하는 데 낭비되는 시간을 줄일 수 있는 도구와 프로세스에 더 관심이 많다.

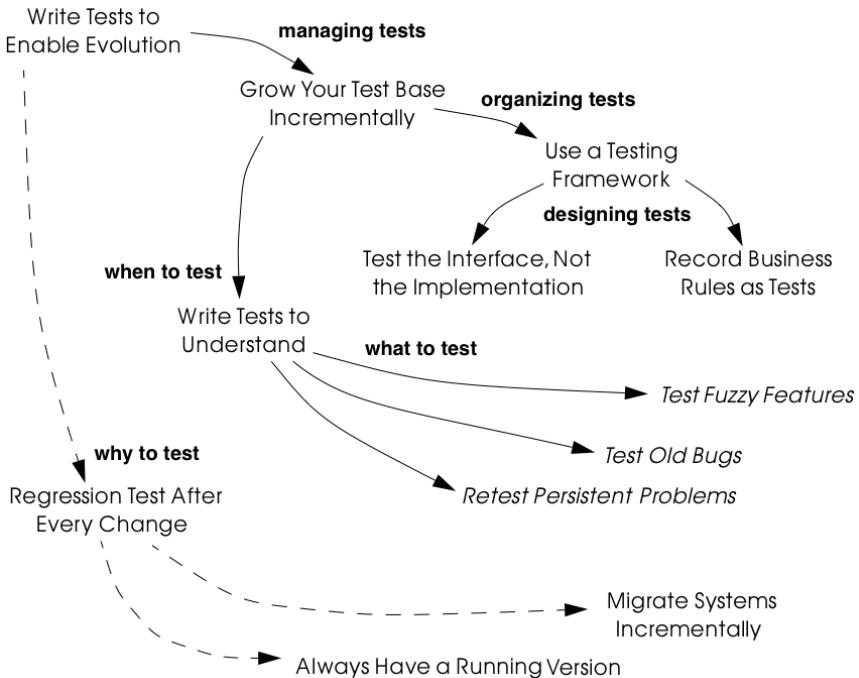


그림 6.1: 언제, 왜, 어떻게, 무엇을 테스트해야 하는가.

- 버그 수정은 재미가 없다.
- 테스트 작성은 고급 작업으로 간주되지 않는다.

개요

그림 6.1에 표시된 것처럼, 진화 활성화를 위한 테스트 작성하기는 이 클러스터의 시작점이다. 이 클러스터는 리엔지니어링 프로젝트에 체계적인 테스트가 중요한 이유와 어떤 종류의 테스트가 필요한지 설명한다. 이 클러스터는 필요에 따라 새로운 테스트를 도입하는 전략을 옹호하는 기본 테스트를 증가시키기 를 기반으로 한다.

테스트의 점진적 도입을 효과적으로 관리하려면 테스트 프레임워크 사용하기를 사용하여 테스트 제품군을 구조화하고 구성하는 것이 중요하다. 테스트

프레임워크는 특정 스타일의 테스트를 설계하는 데 도움이 된다. 특히, 블랙박스 테스트 전략을 사용하여 구성 요소의 구현이 아닌 인터페이스 테스트하기를 수행하면 시스템 변경 시 테스트가 더 유용해지는 경향이 있다. 또한 비즈니스 규칙을 테스트로 기록 를 사용하면 급격한 변화가 있는 경우에도 비즈니스 규칙을 명시적으로 표현하고 실행 중인 시스템과 지속적으로 동기화할 수 있는 효과적인 방법을 확보할 수 있다.

테스트는 여러 가지 이유로 여러 시점에 도입될 수 있다. 이해하기 위해 테스트 작성하기 은 변경 사항을 구현하기 위해 이해해야 하는 시스템 부분에 테스트 노력을 투자할 것을 권장한다. 보다 구체적으로는 퍼지 기능 테스트하기, 오래된 버그 테스트하기, 특히 지속적인 문제 재테스트를 사용하는 것이 좋다.

이 클러스터의 패턴은 리엔지니어링을 위해 마이그레이션 전략 [p. 201] 를 직접 지원한다: 변경할 때마다 회귀 테스트하기 [p. 223] 는 시스템을 점진적으로 변경할 때마다 모든 것이 계속 실행되는지 확인하여 신뢰를 구축하는 데 도움이 됩니다. 사실상 이 테스트는 항상 실행 버전 보유하기 [p. 221] 의 필수 전체 조건이며, 이를 통해 시스템 점진적 마이그레이션하기 [p. 213] 을 수행할 수 있다.

6.1 진화 활성화를 위한 테스트 작성하기

의도 체계적인 테스트 프로그램을 적용해서 레거시 코드에 대한 투자를 보호하자.

문제

리엔지니어링 프로젝트의 리스크, 특히 다음과 같은 리스크를 어떻게 최소화할 수 있는가?

- 레거시 시스템을 단순화하지 못함.
- 시스템에 더 많은 복잡성을 도입.
- 기존에 작동하던 기능이 중단.
- 잘못된 작업에 너무 많은 노력을 소비.
- 향후 변화를 수용하지 못함.

이 문제는 다음과 같은 이유로 어렵다.

- 시스템의 일부가 잘 이해되지 않거나 숨겨진 종속성이 있을 수 있으므로 변경의 영향을 항상 예측할 수 없다.
- 레거시 시스템을 변경하면 문서화되지 않은 측면이나 종속성으로 인해 시스템이 불안정해질 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 실행 중인 시스템이 있으므로 무엇이 작동하고 무엇이 작동하지 않는지 확인할 수 있다.
- 시스템의 어떤 부분이 안정적이고 어떤 부분이 변경될 수 있는지 알 수 있다.

해결

자동화되고 반복 가능하며 저장 가능한 테스트에 기반한 테스트 프로세스를 도입하자.

힌트

Well-designed tests exhibit the following properties:

- **자동화(Automation).** 테스트는 사람의 개입 없이 실행되어야 한다. 완전 자동화된 테스트만이 시스템을 변경할 때마다 시스템이 이전과 동일하게 작동하는지 효율적으로 확인할 수 있는 방법을 제공한다. 테스트를 실행하는 데 필요한 노력을 최소화하면 개발자는 테스트 사용을 주저하지 않게 된다.
- **지속성(Persistence).** 테스트는 자동화할 수 있도록 저장되어야 한다. 각 테스트는 테스트 데이터, 수행할 작업 및 예상 결과를 문서화한다. 예상한 결과를 얻으면 테스트가 성공하고, 그렇지 않으면 실패한다. 저장된 테스트는 시스템이 작동할 것으로 예상되는 방식을 문서화한다.
- **반복성(Repeatability).** 변경 사항이 구현된 후 테스트를 반복할 수 있으면 시스템에 대한 신뢰도가 높아진다. 새로운 기능이 추가될 때마다 기존 테스트 풀에 새로운 테스트를 추가하여 시스템에 대한 신뢰도를 높일 수 있다.
- **유닛 테스트(Unit testing).** 테스트는 개별 소프트웨어 컴포넌트에 연결하여 시스템의 어느 부분을 테스트하는지 명확하게 식별할 수 있도록 해야 한다 [Dav95].
- **독립성(Independence).** 각 테스트는 다른 테스트에 대한 종속성을 최소화해야 한다. 종속성이 높은 테스트는 일반적으로 하나의 테스트가 중단되면 다른 많은 테스트도 중단되는 눈사태 효과(avalanche effects)를 초래한다. 실패 횟수가 감지된 문제의 크기를 정량적으로 나타내는 것이 중요하다. 이렇게 하면 테스트에 대한 불신을 최소화할 수 있다. 프로그래머는 테스트를 믿어야 한다.

트레이드오프

장점

- 테스트는 시스템에 대한 신뢰도를 높이고 동작을 보존하는 방식으로 시스템의 기능, 디자인, 아키텍처까지 변경할 수 있는 능력을 향상시킨다.

- 테스트는 시스템의 산출물을 사용하는 방법을 문서화한다. 일반적인 문서와 달리 동작하는 테스트는 시스템에 대한 항상 최신 설명을 제공한다.
- 보안(security)과 안정성(stability)를 우려하는 고객에게 테스트에 대한 것을 판매의 포인트를 삼는 것은 일반적으로 문제가 되지 않는다. 시스템의 장기적인 수명의 보장에도 테스트는 좋은 이야기 거리가 될 수 있다.
- 테스트는 향후 시스템 진화를 가능하게 하는 데 필요한 환경을 제공한다.
- Smalltalk, Java, C++, 심지어 Perl과 같은 모든 주요 객체 지향 언어에 대한 간단한 단위 테스트 프레임워크가 존재한다.

단점

- 테스트는 무료가 아니다. 테스트를 작성하려면 리소스를 할당해야 한다.
- 테스트는 결함의 존재 여부만 입증할 수 있다. 레거시 시스템(또는 모든 시스템)의 모든 측면을 테스트하는 것은 불가능하다.
- 불충분한 테스트는 잘못된 확신을 줄 수 있다. 모든 테스트가 실행되었기 때문에 시스템이 잘 작동한다고 생각할 수 있지만 전혀 그렇지 않을 수도 있다.

어려움

- 수많은 테스트 접근 방식이 존재한다. 개발 프로세스에 맞는 간단한 접근 방식을 선택하자.
- 레거시 시스템은 규모가 크고 문서화되지 않은 경향이 있기 때문에 테스트하기가 어렵다. 때때로 시스템의 일부를 테스트하려면 크고 복잡한 설정 절차가 필요하며, 이는 부담스러워 보일 수 있다.
- 경영진이 테스트에 투자하는 것을 꺼릴 수 있다. 다음은 테스트에 찬성하는 몇 가지 근거이다.
 - 테스트는 시스템의 안전성을 개선하는 데 도움이 됩니다.
 - 테스트는 시스템 기능에 대한 확실한 신뢰를 보여준다.
 - 자동화된 테스트가 있으면 디버깅이 더 쉬워집니다.

- 테스트는 애플리케이션과 항상 동기화되는 간단한 문서이다.

- 개발자는 테스트 도입을 꺼릴 수 있다. 테스트가 현재의 개발 속도를 높일 뿐만 아니라 향후 유지 관리 작업의 속도를 높일 수 있다는 비즈니스 사례를 구축하자. 버그를 수정하는 데 하루를 보낸 후 변경 사항이 유효한지 확인하는 데 3일을 더 소비하는 개발자와 이야기를 나눈 적이 있다. 자동화된 테스트가 일상 업무에서 프로그램을 더 빨리 디버깅하는 데 도움이 될 수 있다는 것을 보여 주자 그는 마침내 확신을 갖게 되었다.
- 테스트는 개발자에게 지루할 수 있으므로 최소한 올바른 도구를 사용하자. 단위 테스트의 경우, SUnit과 그 다양한 변형은 간단하고 무료이며 Smalltalk, C++, Java 및 기타 언어 [BG98]에서 사용할 수 있다.

예시

다음 코드는 Java[BG98]에서 JUnit 을 사용하여 작성한 단위 테스트를 보여준다. 이 테스트는 Money 클래스에 정의된 add 연산이 예상대로 작동하는지, 즉 $12 \text{ CHF} + 14 \text{ CHF} = 26 \text{ CHF}$ 가 되는지 확인한다.

```
public class MoneyTest extends TestCase {
    public void testSimpleAdd() {
        Money m12CHF= new Money(12, "CHF");           // (1)
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26, "CHF");
        Money result= m12CHF.add(m14CHF);             // (2)
        assertEquals(result.currency(), expected.currency())
            && result.amount() == expected.amount();      // (3)
    }
}
```

이는 테스트가 가져야 하는 다음 속성을 충족한다.

- 이 테스트는 자동화되어 있다. 동작이 올바른 경우 부울 값 참을 반환하고 그렇지 않으면 거짓을 반환한다.
- 테스트가 저장된다. 테스트 클래스의 메서드이다. 따라서 다른 코드처럼 버전 관리가 가능하다.
- 테스크가 반복 가능하다. 초기화 부분(1)은 테스트를 실행하고 무한히 재 실행할 수 있는 컨텍스트를 생성한다.

- 다른 테스트와 독립적이다.

이러한 속성을 가진 테스트를 사용하면 장기적으로 테스트 스위트(test suite)를 구축하는 데 도움이 된다. 버그를 수정하거나 새로운 기능을 추가한 후 또는 이미 존재하는 시스템 측면을 테스트하기 위해 테스트를 작성할 때마다 시스템에 대한 재현 가능(reproducible) 및 검증 가능(verifiable) 정보를 테스트 스위트에 추가하는 것이다. 특히 시스템 리엔지니어링의 맥락에서 이러한 재현 가능하고 검증 가능한 정보는 변경 후 시스템의 측면이 손상되었는지 확인할 수 있어 중요하다.

근거

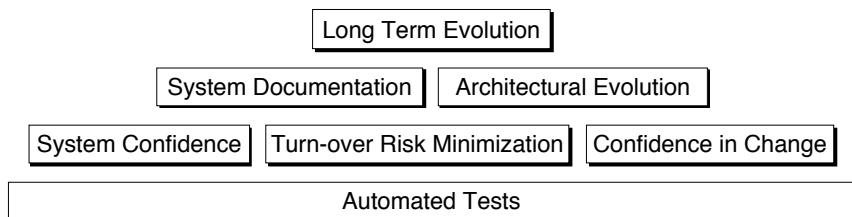


그림 6.2: 자동화된 테스트는 리엔지니어링의 기반(foundation)이다. 시스템에 대한 신뢰를 구축하고, 리스크를 줄이며, 시스템을 변경할 수 있는 능력에 대한 자신감을 높여준다.

테스트는 시스템의 일부가 검증 가능함(verifiable) 방식으로 작동하는 방식을 지정하고 언제든지 실행하여 시스템이 여전히 일관적인지 확인할 수 있기 때문에 시스템에 대한 신뢰도를 나타낸다.

“... 테스트는 단순히 프로그램에 결함이 있음을 드러낼 뿐 결함의 부재를 검증하는 데 사용할 수 없다. 프로그램이 올바르다는 신뢰도를 높일 수 있다.”

— 앤런 데이비스, 원칙 111 [Dav95]

체계적인 테스트는 익스트림 프로그래밍 (Extreme Programming)에서 크게 장려한다. [Bec00] 변화하는 요구사항에 맞춰 프로그램을 신속하게 조정하는

데 필요한 기본 기술 중 하나이다. 레거시 시스템을 변경하는 것은 위험한 작업이다. 변경 후에도 코드가 계속 작동할까? 예상치 못한 부작용이 얼마나 많이 나타날까? 자동화되고 반복 가능한 일련의 테스트가 있으면 이러한 위험을 줄이는 데 도움이 된다.

- 실행 가능한 테스트 세트는 시스템에 대한 신뢰도를 높여준다. (“이 코드가 정말 작동하는 게 확실한가요?” “네, 여기 이를 증명하는 테스트가 있어요.”)
- 실행 중인 테스트 세트는 시스템에 대한 재현 가능(*reproducible*)하고 검증 가능(*verifiable*)한 정보를 나타내며, 애플리케이션과 항상 동기화되어 있다. 이는 일반적으로 다음 날이면 이미 약간 오래된 문서가 되는 대부분의 일반적인 문서와 대조적이다.
- 테스트를 작성하면 개발 프로세스 초기에 버그를 발견할 수 있으므로 생산성이 향상된다.

관련 패턴

진화 활성화를 위한 테스트 작성하기는 항상 실행 버전 보유하기 [p. 221]의 전제 조건이다. 포괄적인 테스트 프로그램이 준비되어 있어야만 시스템 점진적 마이그레이션하기를 사용할 수 있다.

기본 테스트를 증가시키기 및 구현이 아닌 인터페이스 테스트하기는 시스템이 진화하는 동안 테스트 스위트를 점진적으로 구축하는 방법을 소개한다.

6.2 기본 테스트를 증가시키기

의도 특정 시점에 필요한 테스트만 점진적으로 도입하여 테스트의 비용(cost)과 이점(benefit)을 균형 있게 조정한다.

문제

언제 테스트를 도입해야 하는가? 언제 중단할 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 리엔지니어링 프로젝트에서는 테스트 작성에 너무 많은 시간을 할애할 여유가 없다.
- 레거시 시스템은 방대한 경향이 있으므로 모든 것을 테스트하는 것은 불가능하다.
- 레거시 시스템은 잘 문서화되어 있지 않고 잘 이해되지 않는 경향이 있다.
- 시스템을 원래 개발했던 개발자는 떠났을 수 있고 시스템 유지 보수자는 시스템의 내부 작동에 대해 제한된 지식만 가지고 있을 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 우리는 취약한 부분이나 변경하고 싶은 부분이 어디에 있는지 알고 있다.
- 우리는 프로그래머에게 테스트를 통해 이점을 얻을 수 있다고 설득할 수 있다.

해결

작업 중인 시스템의 일부에 대해 테스트를 점진적으로 도입하자.

힌트

- 우선순위를 신중하게 평가하고 처음에는 가장 중요한 구조 요소에 대해서만 테스트를 개발하자. 시스템을 리엔지니어링하면서 새로운 기능, 영향

을 받을 수 있는 레거시 부분 및 그 과정에서 발견한 버그에 대한 테스트를 도입하자.

- 이전 시스템의 스냅샷을 잘 보관해 두면 나중에 원래 시스템과 새 버전 모두에 대해 실행해야 하는 테스트를 도입할 수 있다.
- 비즈니스 가치에 집중하자. 시스템에서 가장 중요한 산출물이 있는 부분에 대한 테스트를 작성하기 시작하자. 비즈니스 규칙을 테스트로 기록하기를 시도해 보자.
- 버그 수정 또는 문제에 대해 관리하고 있는 경우 오래된 버그 테스트하기를 시작점으로 적용하자.
- 허용 가능한 문서와 시스템의 원 개발자가 있는 경우 퍼지 기능 테스트하기 [p. 346]를 적용하는 것이 좋다.
- 구현이 아닌 인터페이스 테스트하기 을 적용하고, 큰 추상화부터 테스트를 시작한 다음 시간이 허락한다면 테스트를 구체화하자. 예를 들어 파이프라인 아키텍처가 있는 경우 전체 파이프라인의 출력이 올바른 입력이 주어졌을 때 올바른지 확인하는 테스트를 작성하기 시작하자. 그런 다음 개별 파이프라인 컴포넌트에 대한 테스트를 작성하자.
- 향후 구현이 변경될 가능성이 있는 부분(하위 시스템, 클래스, 메서드)을 블랙박스 테스트하자.

트레이드오프

장점

- 필요한 테스트만 개발하여 시간을 절약할 수 있다.
- 프로젝트가 진행됨에 따라 가장 중요한 테스트의 기반을 구축할 수 있다.
- 진행하면서 신뢰도를 높일 수 있다.
- 향후 개발 및 유지 관리 활동을 간소화할 수 있다.

단점

- 테스트해야 할 중요한 측면을 잘못 추측할 수 있다.

- 테스트는 잘못된 확신을 줄 수 있다 — 테스트되지 않은 버그가 여전히 시스템에 숨어 있을 수 있다.

어려움

- 테스트에 적합한 컨텍스트를 설정하려면 상당한 시간과 노력이 필요할 수 있다.
- 테스트할 구성 요소의 경계를 파악하는 것도 어렵다. 테스트할 부분과 테스트의 세분화 수준을 결정하려면 시스템과 리엔지니어링 방식을 잘 이해해야 한다.

예시

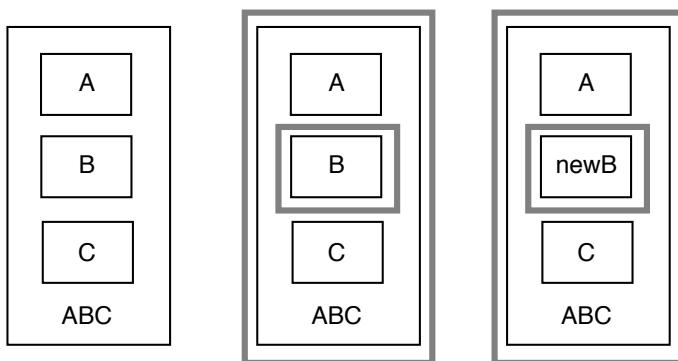


그림 6.3: 변경하려는 시스템 부분에 대한 테스트를 도입.

처음에는 변경하려는 하위 시스템 및 구성 요소에 대해서만 테스트를 도입 한다. 하위 시스템 ABC와 그 컴포넌트 B에 대한 몇 가지 테스트를 도입하고, 구현이 아닌 인터페이스 테스트하기 을 적용하여 B에 대한 테스트가 newB에 대해서도 통과해야 한다는 것을 확인한다.

컴포넌트 B에 대한 테스트만 도입하면 A 및 C와의 통합을 테스트하지 못하므로 중요한 모든 측면을 테스트하지 못할 수 있으므로 버그가 발견되고 수정됨에 따라 점진적으로 새로운 테스트를 추가하는 것이 중요하다.

근거

점진적 테스트 전략을 사용하면 모든 테스트가 완료되기 전에 리엔지니어링 작업을 시작할 수 있다. 현재 변경하려는 시스템 부분과 관련된 테스트에만 집중함으로써 테스트에 대한 최소한의 투자로 변경을 가능하게 하고, 테스트 기반을 확장하면서 팀이 자신감을 쌓을 수 있도록 도와준다.

관련 패턴

테스트 프레임워크 사용하기를 사용하여 테스트를 구조화 하자.

구현이 아닌 인터페이스 테스트하기는 임의의 세부 수준에서 테스트를 개발하기 위한 전략을 제공한다. 비즈니스 규칙을 테스트로 기록하기는 비즈니스 로직을 구현하는 구성 요소를 테스트하기 위한 또 다른 전략을 제공한다. 이해를 위해 테스트 작성하기는 시스템을 리버스 엔지니어링하는 동안 테스트 기반을 준비할 수 있도록 도와준다.

6.3 테스트 프레임워크 사용하기

의도 테스트를 쉽게 개발, 구성 및 실행할 수 있는 프레임워크를 제공하여 개발자가 회귀 테스트를 작성하고 사용하도록 장려한다.

문제

팀이 체계적인 테스트를 채택하도록 어떻게 장려하는가?

이 문제는 다음과 같은 이유로 어렵다.

- 테스트는 작성은 지루하다.
- 테스트를 수행하려면 상당한 양의 테스트 데이터를 사용하기 위해 가져와야 할 수 있다.
- 테스트 실패와 예기치 않은 오류를 구분하기 어려울 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 대부분의 테스트는 테스트 데이터를 생성하고, 몇 가지 작업을 수행하고, 결과가 예상과 일치하는지 확인하고, 테스트 데이터를 정리하는 등 동일한 기본 패턴을 따른다.
- 테스트를 실행하고 실패 및 오류를 보고하는 데 필요한 인프라가 아주 적다.

해결

개별 테스트 케이스에서 테스트 스위트를 구성할 수 있는 테스트 프레임워크를 사용한다.

단계

JUnit 및 SUnit과 같은 단위 테스트 프레임워크[BG98]와 같은 단위 테스트 프레임워크와 대부분의 프로그래밍 언어에 사용할 수 있는 다양한 상용 테스트

하네스 패키지가 있다. 사용 중인 프로그래밍 언어에 적합한 테스트 프레임워크를 사용할 수 없는 경우 다음 원칙에 따라 쉽게 직접 만들 수 있다.

- 사용자는 테스트 데이터를 설정하고, 실행하고, 결과에 대한 어설션(assertion)을 만드는 테스트 케이스를 제공해야 한다.
- 테스트 프레임워크는 어설션(assertion) 실패와 예기치 않은 오류를 구분 할 수 있는 테스트로 테스트 케이스를 래핑해야 한다.
- 테스트 프레임워크는 테스트가 성공할 경우 최소한의 피드백만 제공해야 한다.
 - 어설션(assertion) 실패는 어떤 테스트가 실패했는지 정확하게 표시 해야 한다.
 - 오류는 더 자세한 피드백(예: 전체 스택 추적)을 제공해야 한다.
- 프레임워크는 테스트를 테스트 스위트로 구성할 수 있어야 한다.

트레이드오프

장점

- 테스트 프레임워크는 테스트의 구성을 간소화하고 프로그래머가 테스트를 작성하고 사용하기 쉽게 한다.

단점

- 테스트에는 협력(commitment), 규율(discipline) 및 지원(support)이 필요하다. 팀원들에게 체계적인 테스트의 필요성과 이점을 설득하고 일상적인 프로세스에 테스트를 통합해야 한다. 이러한 규율을 지원하는 한 가지 방법은 팀에 한 명의 테스트를 위한 코치를 두는 것이다. 내비게이터 지정하기 [p. 29]를 수행할 때 이를 고려하자.

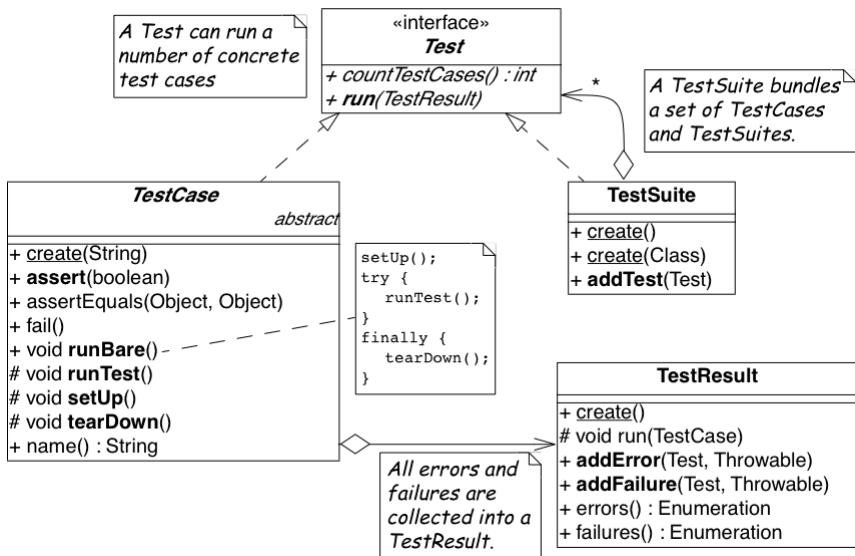


그림 6.4: JUnit은 위에서 설명한 최소 체계보다 훨씬 더 많은 유연성을 제공하는 널리 사용되는 Java용 테스트 프레임워크이다.

예시

JUnit은 위에서 설명한 기본 체계를 상당히 개선한 Java용 테스트 프레임워크로 널리 사용된다. 이 프레임워크는 사용자가 자신의 테스트를 TestCase의 서브클래스로 정의해야 한다는 것을 그림 6.4을 통해 알 수 있다. 사용자는 setUp(), runTest() 및 tearDown() 메서드를 제공해야 한다. Ictsetup() 및 tearDown()의 기본 구현은 비어 있으며, runTest()의 기본 구현은 생성자에 지정된 테스트 이름인 메서드를 찾아서 실행한다. 그런 다음 이러한 사용자 제공 후크 메서드는 runBare() 템플릿 메서드에 의해 호출된다.

JUnit은 추가적인 TestResult 클래스를 통해 실패 및 오류 보고를 관리한다. JUnit의 설계에서 실제로 테스트를 실행하고 오류 또는 실패를 기록하는 것은 TestResult의 인스턴스이다. 그림 6.5에서는 TestCase가 실행 메서드에서 TestResult의 인스턴스로 제어권을 전달하고, 이 인스턴스는 다시 TestCase의 runBare 템플릿 메서드를 호출하는 시나리오를 볼 수 있다.

TestCase additionally provides a set of different kinds of standard

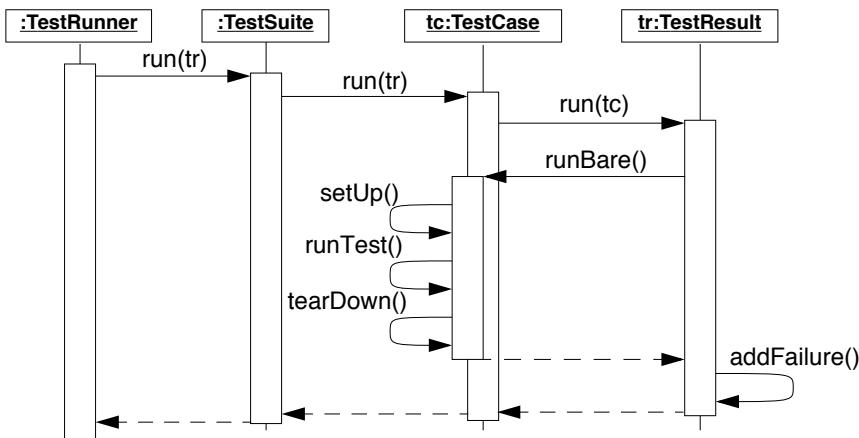


그림 6.5: JUnit에서 테스트는 실제로 `TestResult`의 인스턴스에 의해 실행되며, 이는 `TestCase`의 `runBare` 템플릿 메서드를 호출한다. 사용자는 `setUp()` 및 `tearDown()` 메서드와 `runTest()`에서 호출할 테스트 메서드만 제공하면 된다.

assertion methods, such as `assertEquals`, `assertFails`, and so on. Each of these methods throws an `AssertionFailedError`, which can be distinguished from any other kind of exception. `TestCase`는 `assertEquals`, `assertFails` 등과 같은 다양한 종류의 표준 어설션 메서드 집합을 추가로 제공한다. 이러한 각 메서드는 다른 종류의 예외와 구별할 수 있는 `AssertionFailedError`를 던집니다.

프레임워크를 사용하기 위해서는 일반적으로 테스트하려는 특정 클래스인 `Hashtable`에 대한 테스트 집합을 번들로 묶는 새 클래스(예: `TestHashtable`)를 정의한다. 테스트 클래스는 `junit.framework.TestCase`를 확장해야 한다.

```

import junit.framework.*;
import java.util.Hashtable;

public class TestHashtable extends TestCase {
  
```

테스트 클래스의 인스턴스 변수는 실제 테스트 데이터인 퍽스처(fixture)를 가진다.

```

private Hashtable boss;
private String joe = "Joe";
  
```

```
private String mary = "Mary";
private String dave = "Dave";
private String boris = "Boris";
```

테스트 케이스의 이름을 매개변수로 받는 생성자가 있어야 한다. 그 동작은 슈퍼클래스에 의해 정의된다.

```
public TestHashtable(String name) {
    super(name);
}
```

`setUp()` hook 메서드를 재정의하여 픽스처를 설정할 수 있다. 수행해야 할 정리 작업이 있는 경우 `tearDown()`도 재정의해야 한다. 기본 구현은 비어 있다.

```
protected void setUp() {
    boss = new Hashtable();
}
```

그런 다음 픽스처를 사용하는 테스트 케이스를 얼마든지 정의할 수 있다. 각 테스트 케이스는 독립적이며, 픽스처의 새로운 복사본을 갖게 된다. (원칙적으로 전체 인터페이스를 실행할 뿐만 아니라 테스트 데이터는 일반적인 경우와 경계 사례를 모두 포함하도록 테스트를 설계해야 한다. 여기에 표시된 샘플 테스트는 완전하지 않다.)

각 테스트 케이스는 ‘test’ 문자로 시작해야 한다.

```
public void testEmpty() {
    assert(boss.isEmpty());
    assertEquals(boss.size(), 0);
    assert(!boss.contains(joe));
    assert(!boss.containsKey(joe));
}

public void testBasics() {
    boss.put(joe, mary);
    boss.put(mary, dave);
    boss.put(boris, dave);
    assert(!boss.isEmpty());
    assertEquals(boss.size(), 3);
    assert(boss.contains(mary));
    assert(!boss.contains(joe));
    assert(boss.containsKey(mary));
    assert(!boss.containsKey(dave));
    assertEquals(boss.get(joe), mary);
```

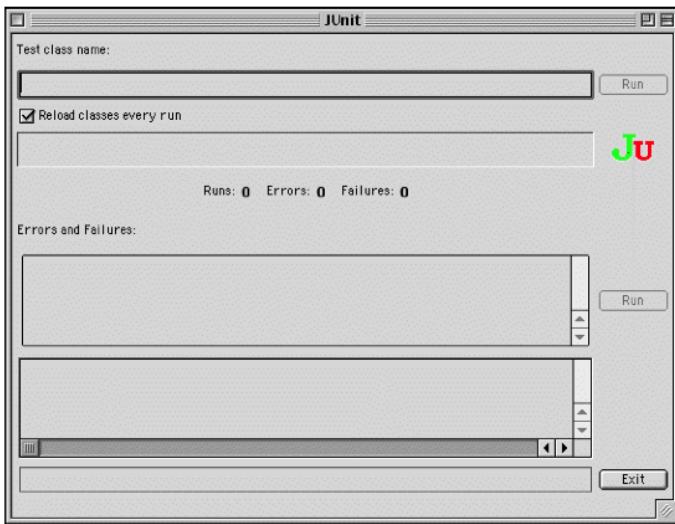


그림 6.6: An instance of `java.ui.TestRunner`. `java.ui.TestRunner`의 인스턴스.

```

        assertEquals(boss.get(mary), dave);
        assertEquals(boss.get(dave), null);
    }
}

```

이 클래스에 정의된 테스트 케이스에서 `junit.framework.TestSuite`의 인스턴스를 빌드하는 정적 메서드 `suite()`를 제공할 수 있다.

```

public static TestSuite suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new TestHashtable("testBasics"));
    suite.addTest(new TestHashtable("testEmpty"));
    return suite;
}
}

```

테스트 케이스 클래스는 그것이 의존하는 모든 클래스와 함께 컴파일되어야 한다.

테스트를 실행하려면 JUnit 프레임워크에서 제공하는 여러 *test runner* 클래스 중 하나(예: `junit.ui.TestRunner`)를 시작하면 된다(그림 6.6 참조).

이 특정 테스트 실행기는 사용자가 테스트 클래스의 이름을 입력할 것으로

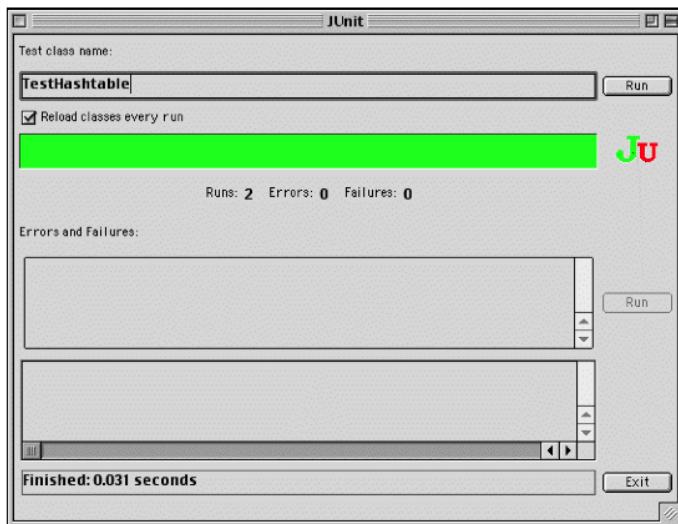


그림 6.7: 테스트 실행에 성공.

예상한다. 그런 다음 이 클래스에 정의된 테스트를 *run*할 수 있다. 테스트 러너는 스위트 메서드를 찾아서 이를 사용하여 **TestSuite**의 인스턴스를 빌드한다. 정적 **suite** 메서드를 제공하지 않으면 테스트 러너는 **test***라는 이름의 모든 메서드가 테스트 케이스라고 가정하여 자동으로 테스트 스위트를 빌드한다. 그런 다음 테스트 러너가 결과 테스트 스위트를 실행한다. 인터페이스는 얼마나 많은 테스트가 성공했는지 보고한다(그림 6.7 참조). 테스트 실행이 성공하면 녹색으로 표시된다. 개별 테스트가 실패하면 빨간색으로 표시되고 실패로 이어진 테스트 케이스에 대한 세부 정보가 제공된다.

근거

테스트 프레임워크를 사용하면 테스트를 더 쉽게 구성하고 실행할 수 있다.

테스트를 계층적으로 구성하면 작업 중인 시스템 부분과 관련된 테스트만 쉽게 실행할 수 있다.

알려진 용도

테스트 프레임워크는 Ada, ANT, C, C++, Delphi, .Net(의 모든 언어), Eiffel, Forte 4GL, GemStone/S, Jade, JUnit Java, JavaScript, k 언어(ksql, kbd에서), Objective C, Open Road(CA), Oracle, PalmUnit, Perl, PhpUnit, PowerBuilder, Python, Rebol, ‘Ruby, Smalltalk, Visual Object 및 UVisual Basic 등 수많은 언어를 지원하며, 그 종류는 무수히 다양하다.

켄트 벡과 에릭 감마는 JUnit [BG98]의 맥락에서 좋은 개요를 제공한다.

6.4 구현이 아닌 인터페이스 테스트하기

또 다른 이름: 블랙박스 테스트[Pre94]

의도 구현 세부 사항보다는 외부 동작에 초점을 맞춘 재사용 가능한 테스트를 구축하여 시스템 변경에도 살아남을 수 있다.

문제

소프트웨어 레거시를 보호할 뿐만 아니라 시스템이 변경되어도 계속 가치가 있는 테스트를 개발하려면 어떻게 해야 할까?

이 문제는 다음과 같은 이유로 어렵다.

- 레거시 시스템에는 시스템이 발전하더라도 계속 작동해야 하는 많은 기능이 있다.
- 시스템을 재설계하는 동안 테스트 작성에 너무 많은 시간을 할애할 여유가 없다.
- 시스템을 변경할 때 변경해야 하는 테스트를 개발하는 데 노력을 낭비하고 싶지 않다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 시스템 컴포넌트에 대한 인터페이스는 테스트해야 할 항목을 알려준다.
- 인터페이스는 구현보다 더 안정적인 경향이 있다.

해결

컴포넌트의 퍼블릭 인터페이스(public interface)를 실행하는 블랙박스 테스트를 개발하자.

힌트

- Be sure to exercise boundary values (*i.e.*, minimum and maximum values for method parameters). The most common errors occur here.
- 경계 값(즉, 메서드 매개변수의 최소값과 최대값)을 실행해야 합니다. 가장 일반적인 오류는 여기서 발생합니다.
- Use a top-down strategy to develop black-box tests if there are many fine-grained components that you do not initially have time to develop tests for.
- 처음에 테스트를 개발할 시간이 없는 세분화된 구성 요소가 많은 경우 하향식 전략을 사용하여 블랙박스 테스트를 개발합니다.
- Use a bottom-up strategy if you are replacing functionality in a very focussed part of the legacy system.
- 레거시 시스템의 매우 집중된 부분의 기능을 교체하는 경우 상향식 전략을 사용합니다.

트레이드오프

장점

- 퍼블릭 인터페이스를 실행하는 테스트는 구현이 변경될 경우 재사용할 수 있는 가능성이 높다.
- 블랙박스 테스트는 종종 동일한 인터페이스의 여러 구현을 실행하는 데 사용할 수 있다.
- 컴포넌트의 인터페이스를 기반으로 테스트를 개발하는 것이 비교적 쉽다.
- 외부 동작에 초점을 맞추면 시스템의 필수적 부분을 다루어야 하는 테스트의 범위가 상당히 줄어든다.

단점

- 블랙박스 테스트가 반드시 가능한 모든 프로그램의 경로를 다루지 않는다는 테스트가 모든 코드를 커버하는지 확인하려면 별도의 커버리지 도구를 사용해야 할 수도 있다.
- 컴포넌트에 대한 인터페이스가 변경되는 경우에도 테스트를 조정해야 합니다.

어려움

- 클래스가 블랙박스 테스트를 지원하는 데 적합한 인터페이스를 제공하지 않는 경우가 있다. 객체의 상태를 샘플링하기 위해 접근자를 추가하는 것이 간단한 해결책이 될 수 있지만 일반적으로 캡슐화가 약해지고 객체가 블랙박스의 역할을 하지 못하게 된다.

예시

진화 활성화를 위한 테스트 작성하기 예 제시된 테스트를 다시 살펴보자. 앞서 살펴본 코드는 **Money** 클래스에 정의된 추가 연산이 예상대로 작동하는지 확인하는 것이었다. 그러나 (3)줄의 어서트는 실제로 **Money** 클래스의 내부 구현에 따라 달라지는데, 이는 동등성(equality) 부분에 접근하여 동등성을 검사하기 때문이다.

```
public class MoneyTest extends TestCase {
    // ...
    public void testSimpleAdd() {
        Money m12CHF= new Money(12, "CHF");           // (1)
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26, "CHF");
        Money result= m12CHF.add(m14CHF);             // (2)
        assertEquals(result.currency(), expected.currency())
            && result.amount() == expected.amount();      // (3)
    }
}
```

그러나 **Money** 클래스가 **Object**에 정의된 기본 **equals** 연산을 재정의하는

경우(그렇게 하려면 `hashCode`도 재정의해야 함), 마지막 어서트 문은 단순화 될 수 있으며 내부 구현과 독립적이 될 것이다.

```
public class MoneyTest extends TestCase {
    // ...
    public void testSimpleAdd() {
        Money m12CHF= new Money(12, "CHF");           // (1)
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26, "CHF");
        Money result= m12CHF.add(m14CHF);             // (2)
        assertEquals(expected,result);                 // (3)
    }
}
```

근거

컴포넌트의 인터페이스는 다른 컴포넌트와의 협업의 직접적인 결과물이다. 따라서 블랙박스 테스트는 시스템에서 가장 중요한 상호작용을 테스트할 수 있는 좋은 기회이다.

인터페이스는 구현보다 더 안정적인 경향이 있으므로 블랙박스 테스트는 시스템의 주요 변경 사항에서도 살아남을 가능성이 높으며, 따라서 테스트 개발에 대한 투자를 보호할 수 있다.

알려진 용도

블랙박스 테스트는 표준 테스트 전략 [Som96]이다.

관련 패턴

비즈니스 규칙을 테스트로 기록하기는 비즈니스 규칙 실행에 초점을 맞추어 일반적인 테스트 개발과는 다른 전략을 채택한다. 테스트할 컴포넌트가 비즈니스 로직을 구현하는 컴포넌트인 경우 이 방법을 사용하면 좋다. 대부분의 다른 컴포넌트의 경우 구현이 아닌 인터페이스 테스트하기 이 더 적합할 수 있다.

복잡한 알고리즘을 구현하는 컴포넌트는 인터페이스 분석만으로는 알고리즘이 처리해야 하는 모든 경우를 파악하지 못할 수 있으므로 블랙박스 테스트에

적합하지 않을 수 있다. 화이트 박스 테스트 [Som96]는 알고리즘을 통해 가능한 모든 경로를 커버할 수 있도록 테스트 케이스를 생성하는 알고리즘 테스트의 또 다른 표준 기법이다.

6.5 비즈니스 규칙을 테스트로 기록하기

의도 비즈니스 규칙을 테스트로 명시적으로 인코딩하여 시스템이 구현하는 비즈니스 규칙과 동기화 상태를 유지한다.

문제

실제 비즈니스 규칙(*actual business rule*), 해당 비즈니스 규칙에 대한 문서(*documentation*) 및 시스템 구현(*implementation*)이 모두 변경되는 동안 어떻게 동기화 상태를 유지할 수 있을까?

이 문제는 다음과 같은 이유로 어렵다.

- 일반적인 문서는 금방 구식이 되어 시스템이 비즈니스 규칙에 대한 설명을 실제로 구현하는지 확인할 수 없다.
- 비즈니스 규칙은 코드에 암시되어 있는 경우가 많다. 어떤 소프트웨어가 특정 비즈니스 규칙의 계산을 담당하는지 명확하지 않을 수 있다.
- 개발자의 이직으로 인해 점점 더 많은 사람들이 시스템에 대해 점점 더 적게 알게 됨으로써 비즈니스에 높은 리스크를 초래할 수 있다.
- 대부분의 경우 특정 규칙을 아는 프로그래머나 사용자는 한 명뿐이며, 그 사람은 내일 퇴사할 수도 있다.
- 비즈니스 규칙은 새로운 법률의 도입과 같은 외부 요인으로 인해 변경될 수 있으므로 이를 명시적으로 표현하는 것이 중요하다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 대부분의 비즈니스 규칙은 일련의 표준 예제로 잘 표현되며, 각 예제에는 잘 정의된 특정 작업을 수행해야 하고 명확하고 관찰 가능한 결과를 가져온다.

해결

비즈니스 규칙을 테스트 케이스, 작업 및 결과에 대한 테스트로 기록하는 실행 가능한 테스트를 작성하자. 테스트 수행이 도중에 멈추면 동기화되지 않았다는

것을 알 수 있다.

힌트

- 개발자와 클라이언트는 테스트를 작성할 수 있다. 개발자는 특정 기능이나 코드와 관련된 테스트를 작성할 수 있다. 사용자는 여러 단위 테스트를 함께 묶는 사용 사례의 형태로 통합 테스트를 작성해야 할 수도 있다 [Dav95] [Bec00].
- 구현 전략이나 최적화 측면에는 관심이 없고 비즈니스 규칙에만 관심이 있다는 점에 유의하자.

트레이드오프

장점

- 규칙이 명시적이므로 사람의 기억에 의존하는 일이 줄어든다.
- 레거시 시스템을 리엔지니어링하기 전에 비즈니스 규칙을 기록해야 한다.
- 비즈니스 규칙을 테스트로 기록하면 진화할 수 있다. 새로운 기능을 추가 해야 할 때 회귀 테스트를 실행하여 기존 비즈니스 규칙이 여전히 올바르게 구현되었는지 확인할 수 있다. 반면에 비즈니스 규칙이 변경되면 해당 테스트를 업데이트하여 변경 사항을 반영할 수 있다.

단점

- 테스트는 구체적인 시나리오만 인코딩할 수 있으며 실제 비즈니스 규칙 자체의 로직은 인코딩할 수 없다.
- 매우 많은 비즈니스 로직을 처리해야 하는 경우 모든 경우를 테스트하는 것은 비현실적일 수 있다.

어려움

- 비즈니스 규칙을 기록한다고 해서 비즈니스 규칙을 추출하는 것은 아니다. 현재의 기술로는 코드에서 비즈니스 규칙을 추출하는 것은 꿈같은 일이다.
- 원래 개발자와 사용자가 모두 떠난 시스템에서는 비즈니스 규칙을 기록하는 것이 어려울 수 있다.

예시

이 예에서는 직원이 자녀를 위해 추가로 받는 금액을 계산한다. 규칙에 따르면 한 개인 또는 부부가 양육하는 모든 자녀에 대해 일정 금액을 받는다. 기본적으로 부모는 12세 미만의 모든 자녀에 대해 매월 150,000프랑을, 12세에서 18세 사이의 모든 자녀와 18세에서 25세 사이의 모든 자녀에 대해 자녀가 일을 하지 않고 교육 시스템에 남아 있는 한 180,000프랑을 받는다. 한 부모는 50% 이상 일하고 있는 경우 이 금액의 100% 전액을 받는다. 부부는 두 파트너의 합산된 근로 비율과 동일한 비율의 지원금을 받는다.

다음 Smalltalk 코드는 다양한 계산에 대한 예상 결과를 하드코딩하는 테스트를 보여준다. 결과를 인쇄하고 올바른지 손으로 확인할 필요 없이 자동으로 결과를 확인할 수 있으며 회귀 테스트의 역할을 한다. 둘째, 다양한 계산의 예상 결과를 문서화한다.

```
testMoneyGivenForKids
```

```
| singlePerson80occupationWithOneKidOf5
  couplePerson40occupationWithOneKidOf5
  couplePerson100occupationWith2KsidoF5
  couplePersonWithOneKidOf14 |
```

```
"cases are extracted from a database after the system has
performed the computation"
```

```
singlePerson80WithOneKidOf5 := extract....
couplePerson40occupationWithOneKidOf5 := extract....
couplePerson100occupationWithOneKidOf5 := extract....
couplePersonWithOneKidOf14 := extract....
"tests"
```

```
"We test that the right amount of money is computed correctly"
```

```
self assert: singlePerson80occupationWithOneKidOf5 moneyForKid =  
    150.  
self assert: couplePerson40occupationWithOneKidOf5 moneyForKid =  
    150*4.  
self assert: couplePerson100occupationWith2KidsOf5 moneyForKid =  
    150*2.  
self assert: couplePersonWithOneKidOf14 moneyForKid = 180.
```

근거

테스트는 시스템이 수행하는 작업을 문서화하는 좋은 방법이다. 비즈니스 규칙을 테스트로 문서화하면 비즈니스 규칙에 대한 설명이 구현과 동기화되도록 보장할 수 있다.

리엔지니어링 프로젝트의 시작은 시스템에 대한 지식을 명시적인 테스트로 문서화하는 프로세스를 설정하기에 좋은 시점이다.

관련 패턴

레거시 시스템을 리버스 엔지니어링하는 동안 이해하기 위해 테스트 작성하기를 수행할 수 있다. 이 과정에서 비즈니스 규칙을 테스트로 기록하기를 자연스럽게 수행할 수 있다. 이런 식으로 기본 테스트를 증가시키기 를 수행하면서 테스트 기반을 준비할 수 있다.

6.6 이해하기 위해 테스트 작성하기

의도 코드에 대한 이해를 실행 가능한 테스트의 형태로 기록하여 향후 변경을 위한 기반을 마련한다.

문제

테스트나 정확하고 정밀한 문서가 없는 레거시 시스템의 일부에 대한 이해를 어떻게 발전시킬 수 있을까?

이 문제는 다음과 같은 이유로 어렵다.

- 코드는 항상 이해하기 어렵다.
- 코드가 실제로 무엇을 하는지에 대한 가설을 세우고 검증하고 싶다.
- 시스템의 동작을 가능한 한 정확하게 지정하고 싶다.
- 이해한 내용을 기록하여 전달하고 싶지만 코드 변경을 시작하자마자 쓸 모 없어질 문서를 작성하는 데 시간을 낭비하고 싶지 않다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 코드 조각이 비교적 작고 경계가 명확하게 정의되어 있다.
- 테스트를 지정하고 유효성을 검사할 수 있다.

해결

가설과 결론을 실행 가능한 테스트로 인코딩하자.

트레이드오프

장점

- 테스트는 이해도를 검증하는 데 도움이 된다.

- 테스트는 시스템의 특정 측면에 대한 정확한 사양을 제공할 수 있다. 테스트는 모호할 수 없다.
- 테스트는 다양한 수준의 이해도를 얻기 위해 적용할 수 있다. 예를 들어 블랙박스 테스트는 역할과 협업에 대한 이해를 구체화하는데 도움이 되는 반면, 화이트박스 테스트는 복잡한 논리의 구현에 대한 이해를 높이는 데 도움이 될 수 있다.
- 개발하는 테스트는 향후 리엔지니어링 작업에 도움이 된다.
- 테스트를 사용하면 테스트 대상 개체의 생성 및 사용에 대해 정확하게 파악할 수 있다.

단점

- 테스트 작성은 시간이 많이 걸린다.

어려움

- 테스트할 개체가 특정 추상화를 나타내지 않는 경우 개체를 테스트할 수 있는 잘 정의된 컨텍스트를 얻는 것은 특히 어렵다. 이해하고자 하는 객체가 생성되는 위치를 찾는 것이 도움이 될 수 있다.
- 동시 시스템(concurrent system)은 테스트하기 어려운 것으로 알려져 있으므로 테스트에서 중요한 측면(예: 레이스 컨디션 처리)을 놓칠 수 있다.

근거

By writing automated tests, you exercise parts of the system you want to understand, while recording your understanding and setting the stage for future reengineering effort. 자동화된 테스트를 작성하면 이해하고자 하는 시스템 부분을 연습하면서 이해도를 기록하고 향후 리엔지니어링 작업을 위한 발판을 마련할 수 있다.

관련 패턴

테스트를 작성하기 전에 이해를 위해 리팩터링하기 [p. 141]를 수행할 수 있다.
테스트를 작성할 때 코드에 대한 질문 연결하기 [p. 135]를 수행해야 한다.

제 7 장

マイグ레이션 전략

리엔지니어링 프로젝트가 잘 진행되고 있다. 레거시 시스템을 잘 이해하고 있으며 진화 활성화를 위한 테스트 작성하기 [p. 169] 을 시작했다. 방향 설정 프로세스를 거쳤고, 가장 가치 있는 것 먼저 하기 [p. 33] 를 처리하기로 결정했다.

새 시스템이 사용자들에게 받아들여질지 어떻게 확신할 수 있을까? 이전 시스템을 사용하는 동안 새 시스템으로 어떻게 마이그레이션할 수 있을까? 새 시스템이 완성되기 전에 어떻게 테스트하고 평가할 수 있을까?

포스: 주요한 요구사항

- 빅뱅 마이그레이션(Big-bang migration)은 실패 리스크가 높다.
- 한 번에 너무 많은 변경 사항을 도입하면 사용자가 거리감을 느낄 수 있다.
- 지속적인 피드백은 어렵고 비용이 많이 들 수 있지만 궤도를 유지하는 데 도움이 된다.
- 사용자는 업무를 완수해야 하며 불완전한 솔루션때문에 인해 방해받고 싶지는 않다.
- 레거시 데이터는 시스템을 사용하는 동안 보존되어야 한다.

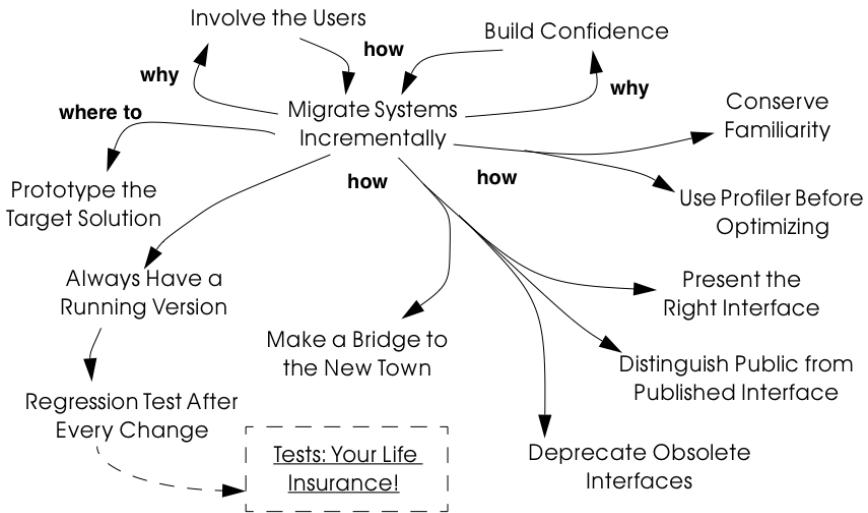


그림 7.1: 레거시 시스템을 마이그레이션하는 방법, 이유 및 대상

개요

레거시 시스템을 리엔지니어링한 다음 배포하는 것만으로는 충분하지 않다. 사실 이렇게 시도하면 (새로운 영역에서 대규모 폭포수 프로젝트가 종종 실패하는 것과 같은 이유로) 반드시 실패할 것이다. 새로운 솔루션을 점진적으로 도입하여 사용자들의 신뢰와 협조를 얻을 준비가 되어 있어야 하며, 기존 시스템에서 새 시스템으로 아직 배포 중인 동안 점진적이고 고통 없이 마이그레이션하는 전략을 채택해야 한다.

이 클러스터의 핵심 메시지는 시스템 점진적 마이그레이션하기이다. 그러나 이것은 말처럼 쉽지 않다. 그림 25에서 시스템 점진적 마이그레이션하기를 수행하려면 수많은 다른 패턴을 고려해야 한다는 것을 알 수 있다. 시스템 마이그레이션에 대한 방대한 문헌이 존재하기 때문에 이 주제를 자세히 다루지는 않겠다. 그러나 객체 지향 레거시 시스템을 리엔지니어링하는 데 가장 중요하다고 생각되는 패턴을 선택하고 주요 요점을 요약했다. 적절한 경우에 대해 독자는 추가 정보 출처를 찾을 수 있다.

이 클러스터의 중심 패턴은 시스템 점진적 마이그레이션 이지만, 핵심 동기는 사용자 참여시키기와 자신감 구축하기에서 제공된다. 이 처음 세 가지

패턴은 리스크를 최소화하고 성공 확률을 높이기 위한 기본 패턴이다.

- 사용자 참여시키기는 전체 리엔지니어링 프로세스에 사용자를 긴밀하게 참여시키고, 중간 결과물을 사용하게 하고, 강력한 지원을 제공함으로써 사용자가 새로운 시스템을 받아들일 가능성을 높인다. 단계별로 시스템 점진적 마이그레이션하기와 자신감 구축하기를 사용하면 더 쉽게 달성 할 수 있다.
- 자신감 구축하기는 사용자에게 가치 있는 결과를 정기적으로 제공함으로써 회의론과 의심을 극복하는 데 도움이 된다.
- 시스템 점진적 마이그레이션하기는 기존 시스템을 점진적이고 점진적으로 새 시스템으로 대체할 것을 권장한다. 그런 다음 진행하면서 새로운 결과를 통합할 수 있으므로 자신감 구축하기 및 사용자 참여시키기에 도움이 된다.

다음 사례도 준수하지 않으면 시스템 점진적 마이그레이션하기를 수행하기가 매우 어렵다.

- 목표 솔루션 프로토타입하기는 사용하여 새 아키텍처와 새로운 기술적 리스크를 테스트한다. 이미 실행 중인 시스템이 있기 때문에 프로토타입이 필요 없다고 생각하기 쉽지만, 이는 거의 항상 실수이다.
- 실행 버전 항상 보유하기는 변경 사항을 자주 통합하여 동기화 상태를 유지하는 데 도움이 된다.
- 변경할 때마다 회귀 테스트하기는 실행 중이던 모든 것이 계속 실행되도록 함으로써 실행 버전 항상 보유하기에 도움이 된다. 여기에는 진화 활성화를 위한 테스트 작성하기 [p. 169] 가 전제되어 있다.

상황에 따라 시스템 점진적 마이그레이션하기에 도움이 될 수 있는 다양한 방법이 있다.

- 뉴 타운으로 가는 브리지 만들기는 (데이터) “다리”의 은유를 도입하여 레거시 컴포넌트에서 대체 컴포넌트로 데이터를 점진적으로 마이그레이션하는 동시에 두 요소가 함께 실행될 수 있도록 한다. 모든 데이터가 전송되면 레거시 컴포넌트를 폐기할 수 있다.

- 올바른 인터페이스 제시하기는 이전 기능을 래핑하여 실제로 원하는 추상화를 내보내 목표 시스템을 점진적으로 개발하는데 도움이 된다.
- 퍼블릭 인터페이스와 게시된 인터페이스 구분하기는 리엔지니어링 팀 내에서 병렬 개발을 용이하게 하기 위해 안정적인 (퍼블릭) 인터페이스(public interface)와 불안정한 (게시된) 인터페이스(published interface)를 구분한다.
- 폐기된 인터페이스 지원 중단하기를 사용하면 클라이언트를 즉시 무효화하지 않고도 사용되지 않는 인터페이스를 정상적으로 폐기할 수 있다.

마지막으로, 다음 두 가지 방법은 급진적이지만 불필요한 변경을 피하는 데 도움이 될 수 있다.

- 친숙도 보존하기는 사용자가 어색함을 느낄 수 있는 급격한 인터페이스 변경을 도입하지 않도록 경고한다.
- 최적화하기 전에 프로파일러 사용하기 [p. 243]은 문제가 있음을 입증하고 문제의 원인을 정확히 파악하여 성능 문제를 고려하도록 한다.

7.1 사용자 참여시키기

고객과 관계맺기 (*Engage Customers*)로도 알려져 있다. [Cop95]

의도 모든 단계에서 사용자를 참여시켜 변경 사항의 수용을 극대화한다.

문제

사용자가 리엔지니어링된 시스템을 받아들일 것이라고 어떻게 확신할 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 이전 시스템이 작동한다. 투박하지만 사용자들은 작동 방식을 알고 있고 문제를 해결하는 방법을 알고 있다.
- 사람들은 자신의 삶을 더 편리하게 만들어 주지 않는 한 새로운 것을 배우는 것을 싫어한다.
- 시스템 개선에 필요한 사항에 대한 사용자의 인식은 시스템이 발전함에 따라 변화하는 경향이 있다.
- 사용자는 종이 위에만 있는 설계를 평가하는 것은 어려워 한다.
- 사용할 준비가 되지 않은 새로운 시스템을 사용자가 좋아하게 만드는 것은 어렵다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 사용자는 자신의 요구가 진지하게 해결되고 있다고 판단되면 새로운 솔루션을 시도할 것이다.
- 사용자는 유용한 정보를 제공하면 피드백을 줄 것이다.

해결

사용자를 새로운 개발에 직접 참여시키고 새로운 시스템을 사용할 수 있도록 긴밀하게 지원하자.

단계

사용자가 우선순위가 어디에 있는지 말할 수 있도록 하자. 가장 가치 있는 것 먼저 하기 [p. 33]로 시작하자. 우선순위를 정기적으로 전달할 수 있는 작은 단계로 나누면 자신감 구축하기 [p. 209]와 같이 사용할 수 있다.

사용자와 개발자 간의 접촉을 장려할 수 있는 환경을 조성하자. 물리적 위치가 중요하다.

정기적으로 중간 결과물을 전달하고 피드백을 받을 수 있는 간단한 절차를 마련하자. 초기 프로토타입은 특히 리스크가 큰 신기술이나 접근 방식을 평가하는 데 도움이 될 수 있다. 좋은 전략은 사용자가 새 시스템이 구축되는 대로 사용할 수 있도록 시스템 점진적 마이그레이션하기 [p. 213]를 수행하는 것이다. 사용자가 느끼는 친숙도를 떨어 놓리지 하기 위해 친숙도 보존하기 [p. 241]를 설정해야 한다.

트레이드오프

장점

- 요구사항이 지속적으로 검증되고 업데이트되므로 올바른 방향으로 나아갈 가능성이 높아진다.
- 사용자가 유용한 결과를 얻고 있고 지원을 받고 있다고 느끼면 유용한 피드백을 제공하기 위해 더 많은 노력을 기울일 것이다.
- 사용자가 프로젝트 전반에 걸쳐 참여하므로 프로젝트 후반에 특별한 교육 세션이 필요하지 않다.

단점

- 개발자는 사용자를 지원하는 것이 시스템 리엔지니어링 작업에 방해가 된다고 느낄 수 있다.
- 사용자 참여가 성공하면 기대치가 높아지고 팀에 추가적인 압박이 가해진다. 예를 들어, 유든은 프로토타입은 기대치를 지나치게 높일 수 있으며 아직 작동하지 않는 부분을 항상 명확히 해야 한다고 이야기 한다 [You97].

어려움

- 결과를 보여 줄 수 있기 전까지는 사용자를 참여를 시작하기 어려울 수 있다.
- 모든 사용자를 참여시킬 수는 없으며, 포함되지 않은 사용자는 소외감을 느낄 수 있다.

근거

실제 고객의 요구 사항을 다루려면 피드백 루프가 필요하다. 사용자를 참여시키고 지원함으로써 이러한 피드백 루프가 활성화 되도록 할 수 있다.

“제품 품질 유지(*Maintaining product quality*)’가 여기서 해결해야 할 문제 가 아니라는 점에 유의하자. 제품 품질은 고객 만족의 한 요소일 뿐이다.”라고 코플리언은 지적했다. [Cop95]

관련 패턴

이 클러스터의 거의 모든 패턴은 사용자 참여시키기 를 지원한다. 시스템 점 진적 마이그레이션 하기 를 통해 사용자가 리엔지니어링 중인 시스템을 함께 작업하도록 하여 자신감 구축하기 를 구현한다.

계획 게임 (*Planning Game*) [BF01]는 반복적으로 스토리를 파악하고, 비 용을 추산하고, 출시할 스토리를 제품에 적용함으로써 사용자 참여시키기 를 구현하는 효과적인 기법이다.

7.2 자신감 구축하기

의도 규칙적으로 성과를 보여줌으로써 전반적인 성공 가능성을 높인다.

문제

모든 종류의 소프트웨어 프로젝트에 대해 고객과 팀원이 종종 갖는 높은 수준의 회의감(skepticism)을 어떻게 극복할 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 요구 사항을 충족하고 제시간에 완료되며 예산 범위 내에서 진행되는 소프트웨어 프로젝트는 거의 없다. 대부분의 프로젝트에 수반되는 회의주의는 쉽게 패배주의(defeatism)로 이어질 수 있으며, 프로젝트는 자기 충족적 예언처럼 실패할 수 있다.
- 사용자가 진정으로 원하거나 필요로 하는 것을 거의 얻지 못한다.
- 레거시 시스템을 실제로 잘 유지할 수 있다고 사용자나 팀원들을 설득하기 어려울 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 모든 문제를 한 번에 해결할 필요는 없습니다.

해결

가능한 한 빨리 긍정적인 결과를 보여줌으로써 긍정적인 분위기를 조성하고, 정기적으로 계속 그렇게 하자.

단계

짧은 간격을 두고 새로운 결과를 전달하자. 각 단계에서 실질적인 가치를 보여줄 수 있는 최소한의 결과가 무엇인지 사용자와 함께 합의하자.

트레이드오프

장점

- 사용자와 개발자 모두 실제 진행 상황을 측정할 수 있다.
- 작은 단계의 비용을 더 쉽게 추정할 수 있다.

단점

- 사용자와 자주 동기화하는 데 시간이 걸린다.
- 사용자는 새 시스템을 이전 시스템과 함께 사용하는 데 필요한 추가 작업을 거부할 수 있다.
- 프로젝트 초기에 좋은 결과를 보여 주는 데 성공하면 기대치가 너무 높아 질 수 있다.

어려움

- 일부 요구 사항은 특히 시스템의 아키텍처 변경을 수반하는 경우 작은 단계로 나누기 어려울 수 있다.
- 리엔지니어링 팀은 가장 중요한 정보 소스 중 하나인 원래 시스템의 개발자를 소외시키지 않도록 주의해야 한다.
- 사용자를 설득하는 것만으로는 충분하지 않으며 경영진의 동의를 얻는데도 신경을 써야 한다. 작은 단계로 경영진을 설득하기는 어렵다. 정기적으로 대규모 데모를 계획하자.

근거

작은 단계를 밟아나가면 개별 단계가 실패할 리스크를 줄일 수 있다. 긍정적인 결과를 자주 얻으면 자신감을 키우는 데 도움이 된다. 같은 맥락에서 익스트림 프로그래밍 (Extreme Programming)은 소규모 릴리즈 (Small Releases)를 지지한다. [Bec00]. 부정적인 결과라도 진행 상황을 모니터링하고 상황을 더 잘 이해하는 데 도움이 되므로 자신감을 키우는 데 도움이 된다.

관련 패턴

목표 솔루션 프로토타입 만들기 와 뉴 타운으로 가는 브리지 만들기 를 사용하면 작은 단계로 결과를 쉽게 보여줄 수 있다.

사용자 참여시키기 를 사용하면 자신감 구축하기 를 더 쉽게 수행할 수 있다.

7.3 시스템 점진적 마이그레이션하기

치킨 리틀(Chicken Little)로도 알려져 있다. [BS95]

의도 기능을 빈번한 충분 개념으로 배포함으로써 빅뱅 리엔지니어링의 복잡성과 리스크를 피하자.

문제

새 시스템 배포는 언제 계획해야 하는가?

이 문제는 다음과 같은 이유로 어렵다.

- 프로젝트는 “빅뱅” 요구 사항 사양을 미리 작성하여 대규모로 계획하고 자금을 지원하는 경우가 많다.
- 실제 요구사항은 뒤늦게 명확해지는 경우가 많다. 특히 처음부터 완벽하게 작동하지 않는다면 사용자는 익숙한 것과 근본적으로 다른 새로운 시스템 사용을 거부할 것이다.
- 새 시스템을 배포하는 데 시간이 오래 걸릴수록 사용자 피드백을 받기 위해 더 오래 기다려야 한다.
- 불완전한 시스템을 배포할 수 없다. 사용자는 불완전한 솔루션에 시간을 낭비하고 싶어하지 않는다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 확장 및 수정할 수 있는 동작하는 시스템이 있다.

해결

가능한 한 빨리 레거시 시스템의 첫 번째 업데이트(update)를 배포하고 대상 시스템으로 점진적으로 마이그레이션하자.

단계

- 레거시 시스템을 여러 부분으로 분해한다.
- 한 번에 하나씩 처리할 부분을 선택한다.
- 해당 부분과 해당 부분에 의존하는 부분에 대한 테스트를 수행한다.
- 레거시 컴포넌트를 래핑, 리엔지니어링 또는 교체하기 위한 적절한 조치를 취한다.
- 업데이트된 컴포넌트를 배포하고 피드백을 받는다.
- 반복한다.

트레이드오프

장점

- 사용자 피드백을 조기에 받고 자신감 구축하기를 수행한다.
- 문제가 발생하면 즉시 알 수 있다.
- 사용자는 새 시스템이 구축되는 동안 학습한다.
- 시스템이 항상 배포된다.
- 시스템은 항상 테스트 중이므로 테스트를 건너뛸 수 없다.

단점

- 시스템을 변경하는 동안 시스템을 계속 실행하려면 더 많은 노력을 기울어야 한다.

어려움

- 새로운 아키텍처로 마이그레이션하는 것이 어려울 수 있다. 새 아키텍처를 적용하기 위해 목표 솔루션 프로토타입하기를 사용할 수 있다, 그리고 기본 구성 요소를 마이그레이션하는 동안 레거시 인터페이스를 숨기려면 올바른 인터페이스 제시하기를 이전 시스템에 적용할 수 있다.

- 실행 중인 시스템을 변경하는 것은 리스크가 크다. 반드시 변경할 때마다 회귀 테스트하기를 수행하자.

근거

실행 중인 시스템에서 최고의 사용자 피드백을 얻을 수 있다. 사용자는 매일 사용하는 시스템에서 피드백을 제공하는 더 많은 동기를 얻고 참여할 수 있다.

알려진 용도

Migrating Legacy Systems [BS95]에서는 이 패턴을 '치킨 리틀 (Chiceken Little)'이라는 이름으로 소개한다(점진적으로 마이그레이션한다는 것은 '치킨 리틀 단계를 밟는다'는 뜻이다). 이 책에서는 점진적 마이그레이션을 위한 전략과 기법에 대해 자세히 설명한다.

관련 패턴

가장 가치 있는 것 먼저 하기 [p. 33]를 적용하여 먼저 작업할 레거시 컴포넌트를 선택한다. 아키텍처 무결성을 유지하기 위해 내비게이터 지정하기 [p. 29]를 적용한다.

마이그레이션할 때 진화 활성화를 위한 테스트 작성하기 [p. 169], 기본 테스트를 증가시키기 [p. 175]를 실행하자. 레거시 컴포넌트를 리엔지니어링하거나 교체할 때 항상 테스트를 다시 작성할 필요가 없도록 구현이 아닌 인터페이스 테스트하기 [p. 187]을 사용하자. 변경할 때마다 회귀 테스트하기를 사용하면 실행 버전 항상 보유하기 [p. 221]을 사용할 수 있다.

리엔지니어링하거나 교체할 의도가 없는 레거시 컴포넌트에 대해서는 올바른 인터페이스 제시하기를 적용하는 것이 좋다.

교체하려는 레거시 컴포넌트에서 데이터를 마이그레이션해야 하는 경우 뉴타운으로 가는 브리지 만들기 [p. 225]을 적용할 수 있다.

7.4 목표 솔루션 프로토타입하기n

의도 프로토타입을 만들어 새 대상 솔루션으로 마이그레이션할 리스크를 평가한다.

문제

새 목표 시스템에 대한 아이디어가 효과가 있는지 어떻게 알 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 작동 중인 시스템을 급격하게 변경하는 것은 리스크가 크다.
- 설계 변경이 기존 기능에 어떤 영향을 미칠지 예측하기 어려울 수 있다.
- 작동하는 솔루션이 테스트되지 않은 솔루션보다 더 신뢰할 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 새로운 아이디어를 테스트하기 위해 레거시 시스템 전체를 리엔지니어링 할 필요는 없다.

해결

새로운 개념을 위해 개발한 프로토타입을 가지고 새롭고 창별하는 요구 사항에 대해 평가하자.

단계

- 리엔지니어링 프로젝트의 가장 큰 기술적 러시크를 파악하자. 일반적으로 다음과 같은 우려 사항이 있다.
 - 새로운 시스템 아키텍처의 선택
 - 레거시 데이터의 새 시스템으로의 마이그레이션
 - 새로운 기술 또는 플랫폼을 통한 적절한 성능 또는 성능 향상(예: 특정 트랜잭션 처리량을 달성할 수 있음을 입증)

- 기술 옵션의 실현 가능성을 순수하게 평가하기 위한 탐색용 프로토타입(exploratory prototype) 즉, 버려질 프로토타입(throwaway prototype)을 구현할지 아니면 최종적으로 새로운 목표 시스템으로 진화할 진화적 프로토타입(evolutionary prototype)을 구현할지 결정합니다.
 - 탐색용 프로토타입(exploratory prototype)은 매우 정확한 질문에 답할 수 있도록 설계되어야 한다. 이러한 질문은 새 플랫폼이 레거시 시스템에서 설정한 성능 제약을 충족할 수 있는지와 같은 순전히 기술적인 질문일 수도 있고, 사용자의 참여와 평가가 필요한 사용성 질문일 수도 있다. 탐색용 프로토타입은 (이 프로토타입이 제공하는 답변이 새 시스템에 영향을 미치기는 하지만) 다른 문제나 질문을 해결하기 위해 설계될 필요가 없으며 마이그레이션할 시스템의 일부가 아니다.
 - 반면에 진화형 프로토타입(evolutionary prototype)은 궁극적으로 레거시 구성 요소를 대체하기 위한 것이므로 목표 아키텍처를 반영해야 한다. 새로운 아키텍처는 레거시 서비스를 가장 적절하게 지원할 뿐만 아니라 레거시 솔루션의 유용성을 제한하는 장애물도 극복한다. 프로토타입은 이러한 리스크에 먼저 대응할 수 있도록 설계되어야 한다.

트레이드오프

장점

- 레거시 시스템의 모든 기능을 구현할 필요가 없으므로 프로토타입을 빠르게 구축할 수 있다.
- 프로토타입을 실행하기 위해 레거시 시스템의 일부를 해킹할 수 있다.
- 목표 시스템에 대한 아이디어가 타당하다면 빠르게 학습할 수 있다.

단점

- 사용자는 버려질 프로토타입을 평가하는 데 많은 시간을 할애할 동기가 높지 않을 수 있다.

- 버려질 프로토타입을 버리지 않고 계속 개발하고 싶은 유혹을 받을 수 있다.

어려움

- 결국 이미 실행 중인 시스템이 있기 때문에, 자신이나 고객에게 프로토타입의 필요성을 설득하기 어려울 수 있다.
- 진화적 프로토타입을 완성하는 데 너무 많은 시간이 걸릴 수 있다. 레거시 컴포넌트에 올바른 인터페이스 제시하기를 적용하여 프로토타입에 레거시 서비스를 위한 좋은 인터페이스를 제공하는 것을 고려하자.

근거

프로토타입은 특정 기술적 접근 방식이 올바른지 아닌지를 빠르게 알려줄 수 있다. 맨먼스 미신(*The Mythical Man-Month*)의 브룩스 [Bro75]는 처음부터 제대로 만들기는 어렵기 때문에 ‘버릴 것은 버려라’라고 조언한다. 러브는 여기서 한 걸음 더 나아가 객체 지향 시스템의 경우 “버릴 것을 각오하고 두 번 구현해야 한다(write two to throw away)”고 경고한다[Lov93]. 푸트와 요더는 무엇보다도 버릴 임시 코드(Throwaway Code)가 도메인 요구 사항을 명확히 하는 가장 좋은 방법이라고 주장하지만, 프로토타입이 “큰 진흙 뭉치(Big Ball of Mud)”로 발전할 리스크가 있다고 경고하기도 한다[FY00].

관련 패턴

뉴타운으로 가는 다리 만들기 을 적용하여 레거시 데이터를 진화하는 프로토타입으로 마이그레이션하는 것을 고려할 수 있다.

7.5 실행 버전 항상 보유하기

의도 주기적으로 시스템을 리빌드하여 변경 사항에 대한 신뢰도를 높인다.

문제

올바른 길을 가고 있다고 어떻게 고객을 확신시킬 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 개발 중인 소프트웨어 시스템을 데모하거나 사용자와 문제를 논의하기 어려울 수 있다. 시스템의 안정적으로 실행되는 버전이 없는 경우가 많기 때문이다.
- 여러 버전의 시스템에서 변경 사항을 통합하는 작업은 느리고 번거로울 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 시스템 통합을 위해 컴포넌트 “구현 완료”를 기다릴 필요가 없다.

해결

매일 새로운 변경 사항과 개발 사항을 통합하는 규칙을 확립하자.

단계

- 버전 관리 및 형상 관리(configuration management) 시스템을 마련하자.
- 작업 중인 부분에 대한 회귀 테스트가 마련되어 있는지 확인하자.
- 시스템 컴포넌트를 체크 아웃하고 다시 체크인하는 짧은 트랜잭션 규율을 마련하자. 변경 사항이 실행 중인 시스템에 통합될 수 있도록 가능한 한 짧게 반복(iteration)을 계획하자.

트레이드오프

장점

- 항상 데모할 수 있는 작동하는 버전이 있다.
- 회귀 테스트를 실행하기 위해 항상 작동하는 버전을 사용할 수 있다.
- 변경 사항을 빠르게 검증할 수 있어 자신감 구축하기에 도움이 된다.

단점

- 변경 사항을 지속적으로 통합해야 한다.

어려움

- 대규모 시스템의 경우 빌드 시간이 매우 길어질 수 있다. 빌드 시간을 단축하려면 먼저 시스템을 재설계해야 할 수 있다.
- 일부 종류의 대규모 수정 사항을 개별적으로 통합할 수 있는 의미 있는 업데이트로 나누기가 어려울 수 있다.

근거

많은 실무자들은 리스크가 크고 고통스러운 빅뱅 통합을 피하기 위한 방법으로 지속적 통합 프로세스를 지지한다 [Boo94].

관련 패턴

변경할 때마다 회귀 테스트하기는 통합 중에 결함이 발생할 리스크를 최소화 한다.

지속적 통합(Continuous Integration) [Boo94] [Bec00]는 실행 버전 항상 보유하기에 대한 입증된 방법이다.

7.6 변경할 때마다 회귀 테스트하기

의도 이전에도 효과가 있었던 것이 여전히 효과가 있는지 확인하여 신뢰를 구축한다.

문제

마지막으로 변경한 내용이 시스템을 손상시키지 않는다는 것을 어떻게 확신할 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 복잡한 시스템에서는 작은 변경이 예상치 못한 부작용을 초래할 수 있습니다. 무해해 보이는 변경이 즉시 발견되지 않고 무언가를 망가뜨릴 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 시스템이 어떻게 동작해야 하는지 표현하는 테스트 스위트(test suite)를 작성했다.

해결

안정적인 상태에 도달했다고 생각될 때마다 회귀 테스트 스위트를 실행하자.

트레이드오프

장점

- 실행 버전 항상 보유하기 가 더 쉽다.
- 진행하면서 자신감 구축하기 가 더 쉽다.

단점

- 테스트를 끊임없이 작성해야 한다.

어려움

- 레거시 시스템에 적절한 회귀 테스트가 정의되어 있지 않을 수 있다. 시스템이 진화하도록 하려면 기본 테스트를 증가시키기 [p. 175]를 사용해야 한다.
- 테스트는 결함이 있다는 것만 보여줄 수 있고 결함이 없다는 것은 보여주지 못한다. 망가져 있는 부분을 정확하게 테스트하지 못했을 수 있다.
- 테스트를 실행하는 데 시간이 많이 걸릴 수 있으므로 변경 사항으로 인해 영향을 받을 수 있다고 생각되는 테스트만 실행하는 것이 좋다. 변경 사항에 대한 “에드혹” 테스트를 피하기 위해 테스트를 분류하되, 모든 테스트를 적어도 하루에 한 번씩 실행하자.

근거

회귀 테스트는 이전에 실행된 테스트가 여전히 동작되고 있음을 알려준다. 발견한 결함과 새로운 기능에 대한 테스트를 지속적으로 구축하면 재사용 가능한 테스트 기반을 확보하게 되어 변경 사항이 건전하다는 확신을 갖게 되고 문제를 조기에 발견하는 데 도움이 된다.

데이비스는 “변경할 때마다 회귀 테스트하기”를 표준 소프트웨어 개발 실천법으로 [Dav95]를 권장한다.

관련 패턴

이미 진화 활성화를 위한 테스트 작성하기 [p. ??]를 시작했어야 한다.

익스트림 프로그래밍(Extreme Programming)의 일반적인 실천법은 새로운 기능을 구현하기 [JAH01] 전에 테스트를 작성하는 것이다. 리엔지니어링의 맥락에서는 변경을 하기 전에 실패하고 변경이 올바르게 구현되면 통과하는 테스트를 작성하는 것을 고려해야 한다. (안타깝게도 변경이 올바른 경우에만 성공하는 테스트를 설계하는 것은 일반적으로 불가능하다!).

회귀 테스트는 지속적인 문제 재테스트하기 [p. 346]에 도움이 된다.

7.7 뉴 타운으로 가는 브리지 만들기

뉴 타운으로 가는 다리(Bridge to the New Town) [Kel00], 데이터 유지 — 코드 전달(Keep the Data — Toss the Code) [BS95]로도 알려져 있다.

의도 브릿지를 사이에 두고 새 시스템을 병렬로 실행하여 레거시 시스템에서 데이터를 마이그레이션한다.

문제

두 시스템이 병렬로 실행되는 동안 레거시 시스템에서 대체 시스템으로 데이터를 점진적으로 마이그레이션하려면 어떻게 해야 하는가?

이 문제는 다음과 같은 이유로 어렵다.

- 레거시 시스템의 일부 컴포넌트는 수리가 불가능하여 교체해야 한다.
- 중요 컴포넌트의 갑작스러운 교체(Big-bang replacement)는 매우 리스크가 크다.
- 레거시 컴포넌트에 의해 조작되는 데이터는 마이그레이션하는 동안 사용 가능한 상태로 유지되어야 한다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 동작하는 레거시 시스템이 있다.

해결

새 컴포넌트가 레거시 시스템에서 데이터를 가져 올 준비가 되면 레거시 시스템에서 교체할 시스템으로 데이터를 점진적으로 전송하는 (데이터) 브리지를 만들자.

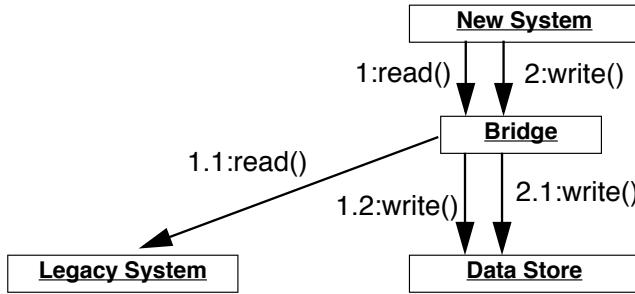


그림 7.2: 브리지(bridge)를 사용하면 데이터를 새 시스템으로 투명하게 전송할 수 있다.

단계

- 동일한 논리 데이터 엔티티를 처리하는 레거시 시스템과 교체 시스템의 컴포넌트를 구별한다.
- 데이터가 아직 마이그레이션되지 않은 경우 새 컴포넌트에서 레거시 데이터 소스로 읽기(read) 요청을 리디렉션하는 “데이터 브리지 (data bridge)”를 구현한다. 브리지는 필요한 모든 데이터 변환을 담당한다. 새 컴포넌트는 브리지를 인식해서는 안 된다.
- 레거시 컴포넌트를 조정하여 새 데이터가 최신 상태로 유지되도록 쓰기(write) 요청을 새 컴포넌트로 리디렉션한다.
- 모든 데이터를 옮기면 브리지와 레거시 컴포넌트를 제거한다.

트레이드오프

장점

- 모든 레거시 데이터를 마이그레이션하지 않고 새 시스템 사용을 시작할 수 있다.

단점

- 레거시 데이터와 새 데이터 사이에 간단한 매핑이 없는 경우 데이터 브리지를 올바르게 구현하기가 까다로울 수 있다.
- 일부 데이터가 옮겨지면 되돌리기가 어려울 수 있다.
- 데이터 브리지는 성능 오버헤드를 추가한다. 이는 허용될 수도 있고 허용되지 않을 수도 있다.

어려움

- “단계별 마이그레이션 방식(*Stepwise migration scheme*)은 대규모의 레이어드 비즈니스 시스템에서 매우 효과적인 것으로 입증되었다. 체크인/체크아웃 지속성이 있고 결합도가 높고 매우 촘촘하게 짜여진 오브젝트 망이 있는 CAD 애플리케이션에서는 일반적이지 않다.” [Kel00]

알려진 용도

브로디와 스톤브레이커는 레거시 시스템 마이그레이션(*Migrating Legacy Systems*)에서 데이터 브리지 및 게이트웨이 사용에 대해 훨씬 더 자세히 설명한다. [BS95].

켈러는 “뉴 타운으로 가는 브리지(The Bridge to the New Town)” [Kel00]에서 레거시 데이터 마이그레이션의 기술적 문제에 더 중점을 두고 있으며, 이 패턴이 성공적으로 적용된 수많은 사례를 지적한다.

이 패턴에는 레거시 시스템 전체를 교체할 것인지, 아니면 컴포넌트 하나만 교체할 것인지, 그리고 사용자가 두 시스템에 동시에 액세스할 수 있어야 하는지 여부에 따라 다양한 변형이 가능하다.

근거

이전 시스템과 새 시스템 간의 브리지를 사용하면 새 시스템이 완성되기 전에 사용자가 새 시스템의 기능을 사용할 수 있도록 할 수 있다. 브리지는 두 시스템을 서로 분리하여 레거시 시스템의 영향을 받지 않고 새로운 아키텍처 비전에

따라 새 시스템을 개발할 수 있도록 한다.

관련 패턴

브리지는 시스템 점진적 마이그레이션하기 를 지원하여 자신감 구축하기 를 지원한다.

7.8 올바른 인터페이스 제시하기

시맨틱 래퍼(Semantic Wrapper) [O'C00], 양탄자 밑으로 쓸어 넣기(Sweeping it Under the Rug) [FY00]로도 알려져 있다.

의도 기존 구현에 반영되어 있지 않더라도 올바른 추상화를 드러나도록 레거시 시스템을 래핑한다.

문제

マイグ레이션 프로세스 중에 새 목표 시스템이 레거시 서비스를 어떻게 액세스해야 하는가?

이 문제는 다음과 같은 이유로 어렵다.

- 목표 시스템 구현이 아직 완료되지 않았으므로 마이그레이션하는 동안 레거시 서비스에 의존해야 한다.
- 레거시 시스템이 목표 시스템에 필요한 인터페이스를 제공하지 않는다.
- 레거시 컴포넌트 측면에서 새 컴포넌트를 구현하면 목표 시스템이 레거시 아키텍처 및 디자인에 대해 편향이 발생한다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 레거시 서비스에 직접 액세스할 필요가 없다.

해결

새 시스템에 포함할 추상화를 찾아내고 기존 소프트웨어를 래핑하여 새 추상화를 에뮬레이트한다.

힌트

예를 들어 객체 지향 시스템 내에서 사용되는 절차적 그래픽 라이브러리를 생각해 보자. 이 라이브러리를 객체 지향 방식으로 다시 구현하려면 너무 많은

비용과 시간이 필요하다. 유ти리티 클래스(즉, 정적 메서드는 있지만 인스턴스가 없는 클래스)로 래핑하는 것이 더 쉽겠지만, 진정한 객체 지향 인터페이스를 제공하지만 기본 절차적 추상화를 사용하여 구현되는 약간 두꺼운 래퍼를 작성하는 것이 더 현명할 것이다. 이렇게 하면 새로운 시스템이 레거시 추상화에 의해 오염되지 않는다.

트레이드오프

장점

- 처음부터 적절한 추상화를 사용할 수 있다면 목표 시스템을 레거시 서비스에서 떼어내기가 더 쉬워진다.
- 레거시 설계가 새 대상에 악영향을 미칠 리스크를 줄일 수 있다.

단점

- 새 인터페이스가 안정적이지 않을 수 있으므로 개발자가 사용을 꺼릴 수 있다.

어려움

- 절차적 추상화를 단순히 유ти리티 클래스로 포장하고 싶은 유혹을 뿌리치기가 어려울 수 있다.

알려진 용도

앨런 오캘러한 [O'C00]은 대규모 비즈니스 크리티컬 레거시 시스템을 객체 지향 및 컴포넌트 기반 기술로 마이그레이션하는 ADAPTOR 패턴의 맥락에서 이 패턴을 “시맨틱 래퍼 (Semantic Wrapper)”로 간략하게 제시한다.

근거

올바른 인터페이스 제시하기는 레거시 설계의 관점에서 생각하지 않고 대체 접근 방식을 더 쉽게 고려할 수 있도록 해준다.

관련 패턴

올바른 인터페이스 제시하기는 둘 다 래퍼를 구현 기법으로 사용하기 때문에 표면적으로는 Adapter [p. 349] 와 비슷하다. 그러나 Adapter는 호환되지 않는 인터페이스를 클라이언트가 기대하는 다른 인터페이스에 맞게 조정한다. 반면 올바른 인터페이스 제시하기는 레거시 컴포넌트에 더 적합한 새 인터페이스를 소개한다.

반드시 폐기된 인터페이스 지원 중단하기를 사용하자.

올바른 인터페이스 제시하기로 구현된 새 인터페이스가 안정적이지 않은 경우 공개 인터페이스와 게시된 인터페이스 구분하기를 사용해야 한다.

7.9 퍼블릭 인터페이스와 게시된 인터페이스 구분하기

게시된 인터페이스 (Published Interface) [O'C00]로도 알려져 있다.

의도 불안정한 “게시된 인터페이스(*published interface*)”와 안정적인 “퍼블릭 인터페이스(*published interface*)”를 구분하여 병렬 개발을 용이하게 한다.

문제

새 인터페이스가 아직 개발 중인 동안 레거시 인터페이스에서 새 목표 인터페이스로 마이그레이션하려면 어떻게 해야 하는가?

이 문제는 다음과 같은 이유로 어렵다.

- 가능한 한 빨리 새 목표 시스템으로 마이그레이션을 활성화하려고 한다.
- 새 대상 컴포넌트의 인터페이스를 너무 일찍 고정하고 싶지 않다.
- 널리 사용되는 컴포넌트의 인터페이스를 변경하면 개발 속도가 느려진다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 제공하는 인터페이스의 상태를 제어할 수 있다.

해결

시스템의 나머지 부분에서 사용할 수 있는 컴포넌트의 퍼블릭 인터페이스와 하위 시스템 내에서 사용할 수 있지만 아직 프라임 타임에 사용할 준비가 되지 않은 컴포넌트의 불안정한 ‘게시된’ 인터페이스를 구분한다.

힌트

“게시된” 인터페이스는 어떤 프로그래밍 언어에서도 지원되지 않으므로 네이밍 규칙(naming convention)을 사용하거나 원하는 효과를 얻기 위해 다른 기

능을 악용해야 할 수 있다.

- Java에서 이러한 인터페이스를 **protected**로 선언하거나 패키지 범위를 지정하는 것을 고려하자(undeclared). 인터페이스가 안정화되면 **public**으로 다시 선언할 수 있다.
- C++에서 게시된 인터페이스가 **private** 또는 **protected**인 컴포넌트를 선언하고, 이를 사용할 수 있는 클라이언트를 **friends**로 선언하는 것을 고려하자. 인터페이스가 안정화되면 컴포넌트를 **public**으로 다시 선언하고 **friends**의 선언을 삭제한다.
- Smalltalk에서, 게시된 컴포넌트의 카테고리를 선언하는 것을 고려해 보세요. 또한, 안정된 메시지와 불안정한 메시지를 구분하기 위해 게시된 메시지 카테고리를 선언하는 것도 고려해 보자.
- 불안정한 컴포넌트나 인터페이스의 이름을 “게시된” 상태를 나타내도록 꾸미는 것을 고려하자. 컴포넌트가 퍼블릭이 되면 이름을 바꾸고 모든 클라이언트에 패치를 적용하거나 이전 이름으로 버전을 폐기하자(폐기된 인터페이스 지원 중단하기).

트레이드오프

장점

- 게시된 인터페이스의 클라이언트는 해당 인터페이스가 변경될 수 있음을 알고 있다.

단점

- 인터페이스를 “게시된” 것으로 식별하는 것은 순전히 규칙(convention)과 규율(disipline)의 문제이다.
- 인터페이스를 게시된 인터페이스에서 퍼블릭으로 변경하면 새 인터페이스로 업그레이드해야 하는 클라이언트에게 일정한 오버헤드가 발생한다.

어려움

- 클라이언트는 불안정한 게시된 인터페이스를 사용해야 하는지, 아니면 레거시 서비스를 계속 사용해야 하는지 선택의 기로에 놓일 수 있다.

알려진 용도

게시된 인터페이스는 ADAPTOR 패턴의 또 다른 패턴이다. [O'C00].

근거

Clients are in a better position to evaluate the risk of using a component if they know its interface is declared to be “published” but not yet public. 클라이언트는 컴포넌트의 인터페이스가 “게시된” 인터페이스로 선언되었지만 아직 퍼블릭 인터페이스가 되지 않은 경우 컴포넌트 사용의 리스크를 평가할 수 있는 더 나은 위치에 있다.

관련 패턴

레거시 컴포넌트에 올바른 인터페이스 제시하기를 적용하면 새 인터페이스가 안정적이지 않을 수 있으므로 공개 인터페이스와 게시된 인터페이스 구분하기를 해야 한다. 새 인터페이스가 안정화되거나 안정적인 대체 컴포넌트로 대체되면 인터페이스가 퍼블릭 인터페이스가 될 수 있다.

인터페이스를 퍼블릭으로 업그레이드하면 액세스 방식이 변경될 수 있다. 반드시 폐기된 인터페이스 지원 중단하기를 수행하자.

7.10 폐기된 인터페이스 지원 중단하기

Also Known As: Deprecation [SP98] 지원 중단(Deprecation) [SP98]로도 알려져 있다.

의도 사용되지 않는 인터페이스를 “지원 중단(deprecated)”으로 플래그 지정하여 클라이언트가 퍼블릭 인터페이스 변경에 대응할 시간을 준다.

문제

모든 클라이언트를 무효화하지 않고 인터페이스를 수정하려면 어떻게 해야 하는가?

이 문제는 다음과 같은 이유로 어렵다.

- 퍼블릭 인터페이스를 변경하면 많은 클라이언트의 지원이 중단될 수 있다.
- 폐기된 인터페이스(obsolete interface)를 그대로 두면 향후 유지 관리가 더 어려워진다.
- 모든 변경이 더 나은 것은 아니다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 이전 인터페이스와 새 인터페이스가 일정 기간 동안 공존할 수 있다.

해결

이전 인터페이스를 ‘지원 중단됨(deprecated)’으로 플래그를 지정하여 다음 릴리스에서 거의 확실하게 제거될 것임을 클라이언트에 알린다.

단계

- 퍼블릭 인터페이스를 변경해야 한다고 결정했지만 모든 클라이언트를 중단하고 싶지는 않다. 새 인터페이스를 구현하되 이전 인터페이스는 “지원

중단(deprecate)”하자. 지원 중단 메커니즘은 클라이언트에 인터페이스가 변경되었으며 대신 최신 인터페이스가 권장됨을 알려야 한다.

- 지원 중단된 인터페이스가 어느 정도까지 계속 사용될 수 있는지, 그리고 영구적으로 폐기할 수 있는지 평가하자. 향후 릴리스에서 제거를 고려하자.
- Java 는 언어 기능으로 지원 중단(deprecation)을 지원한다.
 - javadoc 문서에 `@deprecated` 태그를 추가하여 기능을 지원 중단 하자. 이 태그는 javadoc 문서 생성기에서 인식될 뿐만 아니라 지원 중단된 기능을 사용하는 코드를 `-deprecated` 옵션으로 컴파일할 경우 컴파일 타임 경고도 생성한다.
- 다른 접근 방식은 다음과 같다.
 - 지원 중단되는 인터페이스를 문서에서 사용자에게 알리기만 하면 된다.
 - 지원 중단된 인터페이스 또는 컴포넌트를 이동하거나 이름을 바꾼다. 클라이언트는 계속 사용할 수 있지만 지원 중단된 양식을 계속 사용하려면 조정하고 다시 컴파일해야 한다.
 - 지원 중단된 컴포넌트를 런타임 경고를 생성하거나 로그 파일에 경고를 출력하는 동등한 컴포넌트로 대체한다.
 - 다른 방법으로 컴파일 시간 또는 링크 시간 경고를 생성하도록 프로그래밍 환경 또는 지원 중단된 컴포넌트 자체를 구성하는 것을 고려하자.

트레이드오프

장점

- 클라이언트는 변경 사항에 즉시 적응할 필요가 없다.
- 변경할 수 있는 여유 시간이 있다.

단점

- 클라이언트는 지원 중단을 무시할 수 있습니다.

어려움

- 지원 중단된 컴포넌트의 모든 클라이언트를 추적하기 어려울 수 있다.
- 지원 중단된 컴포넌트를 실제로 폐기할 시기를 결정하기 어려울 수 있다.
- 인터페이스는 유지하되 의미를 변경하려면 새 컴포넌트를 도입하고 이전 컴포넌트는 지원 중단해야 할 수 있다. 특정 메서드가 이제 예외(exception)를 던지는 대신 기본값(default value)을 반환해야 하는 경우(또는 그 반대의 경우)가 이에 해당할 수 있다.

알려진 용도

퍼디타 스티븐스와 롭 폴리는 복잡한 시스템에서 진화하는 API를 관리하기 위한 일반적인 관행으로 지원 중단(Deprecation) 을 소개한다 [SP98].

근거

지원 중단(Deprecation)은 변경의 영향을 평가할 수 있는 시간적 여유를 제공한다.

7.11 친숙도 보존하기

의도 사용자를 소외시킬 수 있는 급진적인 변경은 피하기

문제

사용자가 익숙한 업무 수행 방식을 방해하지 않으면서 레거시 시스템을 대대적으로 개편하려면 어떻게 해야 할까?

이 문제는 다음과 같은 이유로 어렵다.

- 레거시 시스템에 상당한 변화가 필요하다.
- 사용자가 레거시 시스템에 만족하지 않지만 잘 이해하고 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 새 솔루션으로 점진적으로 마이그레이션할 수 있다.

해결

각 새 릴리스 사이에 상대적으로 적은 수의 변경 사항만 지속적으로 도입하자.

트레이드오프

장점

- 사용자는 릴리스 간에 작업 습관을 크게 바꿀 필요가 없다.

어려움

- 때로는 급진적인 변화가 필요하다. 익숙함을 유지하면서 명령줄 인터페이스에서 GUI로 마이그레이션하는 것은 어려울 수 있다.

근거

릴리스 간에 변경 사항이 너무 많으면 숨겨진 결함의 리스크가 커지고 사용자가 받아 드릴 가능성이 낮아진다.

리먼과 벨레디의 “‘친숙도 보존의 법칙 (Law of Conservation of Familiarity)’에 따르면 시스템의 릴리스 간 점진적 변화는 시간이 지나도 거의 일정하게 유지된다 [LB85]. 다른 작업을 수행하면 불필요한 리스크가 발생하기 때문에 이는 비교적 자연스러운 현상이다.

관련 패턴

친숙도 보존하기를 수행하려면 시스템 점진적 마이그레이션하기를 수행해야 한다. 어떤 변경이 허용되는지 파악해야 하기 위해서 사용자 참여시키기를 해야 한다. 변경의 잠재적 영향을 평가하기 위해 목표 솔루션 프로토타입 만들기를 해야 한다.

7.12 최적화하기 전에 프로파일러 사용하기

의도 병목 지점이 어디인지 확인하여 불필요한 “최적화(optimizations)”에 리엔지니어링 노력을 낭비하지 않는다.

문제

명백히 비효율적인 코드는 언제 다시 작성해야 할까?

이 문제는 다음과 같은 이유로 어렵다.

- 소프트웨어를 리엔지니어링할 때 레거시 코드에서 순진한 알고리즘을 많이 접할 수 있다.
- 어떤 것이 성능에 영향을 미칠지 예측하기 어려울 수 있으며, 순수한 가능성으로 많은 시간을 낭비할 수 있다.
- 최적화된 코드는 단순하고 순진한 코드보다 더 복잡한 경우가 많다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 성능 문제가 있을 수 있는 위치를 알려주는 도구가 있다.

해결

시스템의 “명백히 비효율적인” 부분을 최적화하고 싶을 때마다 먼저 프로파일러를 사용하여 실제로 병목 현상이 있는지 확인하자.

최적화가 효과를 가져올 것이라고 프로파일러로 확인하지 않는 한 어떤 것도 최적화하지 말자.

계속 진행하기로 결정했다면 성능 향상을 입증할 수 있는 벤치마크를 준비하자.

트레이드오프

장점

- 전체 성능에 영향을 주지 않는 최적화에 시간을 낭비하지 말자.

단점

- 너무 순진한 알고리즘이 시스템에서 더 오래 남아 있게 된다.

근거

약간의 코드를 최적화함으로써 얻을 수 있는 성능 향상은 프로그램이 일반적인 실행에서 해당 코드에 얼마나 많은 시간을 소비하는지에 따라 달라진다. 프로파일러를 통해서 그 시간이 얼마나 되는지 알수 있다.

“동작하게 하고, 제대로 동작하게 하고, 빠르게 동작하게 하기 (Do it, then do it right, then do it fast)”는 여러 출처에서 인용된 잘 알려진 격언이다. 이 격언의 기원은 컴퓨터 과학 분야가 아닌 다른 곳에서 유래했을 가능성이 높다. 그 근거는 성능 문제에 너무 일찍 몰두하면 시스템을 복잡하고 유지보수하기 어렵게 만들 리스크가 있다는 것이다. 그보다는 먼저 작동하는 해결책을 찾은 다음 이해가 되면 정리하는 것이 좋다. 마지막으로, 중요한 성능 병목 현상을 파악할 수 있다면 그때가 바로 차이를 가져올 수 있는 부분만 최적화할 수 있는 시기이다.

결론적으로, 성능에 심각한 영향을 미치지는 않지만 다른 변경을 더 쉽게 할 수 있다면 약간 복잡한 ‘최적화’ 코드를 더 간단한 ‘순진한’ 솔루션으로 대체하는 것도 좋은 생각일 수 있다.

데이비스의 “최적화하기 전에 프로파일러 사용하기”에 대한 논의도 참조하자. [Dav95].

관련 패턴

이해하기 위해 리팩터링하기를 수행하면 “제대로 하기(do it right)”를 위한 두 번째 단계가 시작된다.

제 8 장

중복 코드 감지

파울러와 벡은 소프트웨어 리팩터링의 필요성을 나타내는 상위 10가지 코드 스멜 중 첫 번째로 중복 코드(duplicated code)를 꼽았다 [FBB⁺99]. 그들이 설명 하길, 코드를 복제할 때마다 나중에 비용을 지불해야 할 무언가(거의 기성품에 가까운 소프트웨어)를 지금 받는다는 의미에서 대출을 받는 것과 같다고 한다. 대출을 받는 것이 잘못된 것은 아니지만, 중복을 제거하기 위해 코드를 리팩터링하는 시간을 들여 지금 소액을 갚거나 (복잡성 및 유지 관리 비용의 증가로) 나중에 많은 금액을 갚는 것 중 하나를 선택할 수 있다.

경험적 연구 데이터에 따르면 일반적으로 산업용 소프트웨어의 8%에서 12%는 중복된 코드로 구성되어 있다 [DRD99]. 이 수치는 많지 않아 보일 수 있지만 실제로는 매우 높은 중복률을 달성하기 어렵습니다. (중복률이 50%라도 되려면 얼마나 많은 노력이 필요할지 상상해 보자!) 따라서 15~20%의 중복률은 심각한 것으로 간주된다.

다음과 같은 이유로 중복 코드를 식별하는 것이 중요하다.

- 중복 코드는 구현된 모든 기능의 변형을 변경해야 하므로 변경 사항을 도입하는 데 방해가 된다. 일부 변형을 놓치기 쉬우므로 다른 곳에서 버그가 발생할 가능성이 높다.
- 중복 코드는 시스템의 로직을 식별 가능한 산출물(클래스, 메서드, 패키지)로 그룹화하는 대신 복제하고 분산시킨다. 따라서 시스템을 이해하고

변경하기가 더 어려워진다. 논리적 부분 간의 관계를 이해하는 대신 먼저 식별한 다음 그 관계를 이해해야 한다.

중복 코드는 다양한 이유로 발생한다.

- 프로그래머가 이전에 수행한 작업과 원격으로 유사한 기능을 구현할 때마다 기존 코드를 새 작업의 모델로 사용하는 것은 당연하다. 기존 프로시저를 재조합하는 문제라면 작업은 간단할 것이다. 그러나 동작이 더 복잡한 경우 가장 쉬운 방법은 이전 코드를 복사, 붙여넣기 및 수정하여 기능을 달성하는 것이다. 이전 코드와 새 코드가 모두 다른 애플리케이션에 속한 경우 피해는 최소화된다. 하지만 같은 시스템에 속해 있다면 중복 코드가 발생하게 된다.
- 때때로 다른 애플리케이션 또는 동일한 애플리케이션의 다른 버전 간에 코드를 복사, 붙여넣기 및 수정하는 경우가 있다. 여러 버전을 동시에 유지보수해야 하거나 서로 다른 애플리케이션 또는 버전을 병합해야 하는 경우 코드 중복 문제가 즉시 발생한다.

리엔지니어링 관점에서 보면 일반적으로 사람들은 시스템에 중복 문제가 있는지 여부를 알 수 있다. 첫째, 개발팀이나 관리자가 알려줄 것이다. 둘째, 일반적으로 프로젝트에서 중복이 발생했다는 몇 가지 분명한 징후가 있다. 예를 들어, 두 명의 개발자가 기존 코드를 복사하여 붙여넣지 않고는 8개월 이내에 4배만 줄의 코드를 개발할 수 없다. 시스템을 분석하는 동안 우연히 중복된 코드를 발견할 수도 있다. 그러나 시스템에 중복된 코드가 있다는 것을 아는 것과 정확히 어떤 부분이 중복되었는지 아는 것에는 큰 차이가 있다.

개요

중복 코드 감지는 두 가지 패턴으로 구성된다. 중복 코드를 감지하는 방법을 설명하는 기계적으로 코드 비교하기, 그리고 간단한 이차원적 시각화를 통해 중복 코드를 더 잘 이해할 수 있는 방법을 보여주는 도트 플롯으로 코드 시각화하기로 구성되어 있다.

시스템에서 중복을 감지하고 이해한 후에는 다양한 전략을 결정할 수 있다. 메시드 추출하기 [p. 347] 와 같은 다양한 리팩터링 패턴을 사용하면 중복을

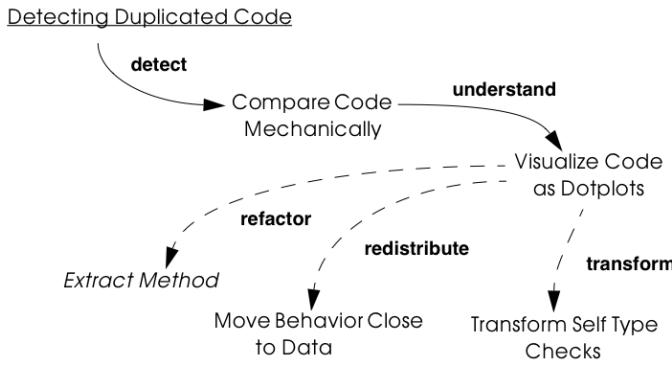


그림 8.1: 중복 코드 감지 를 지원하는 두 가지 패턴.

제거하는 데 도움이 될 수 있다. 중복은 책임이 잘못 배치되었다는 신호일 수 있으며, 이 경우 데이터 가까이 동작 이동하기 [p. 269]를 해야 할 수도 있다.

복잡한 조건문도 중복의 한 형태이며, 여러 클라이언트가 대상 클래스에 속해야 하는 동작을 중복해야 함을 나타낼 수 있다. 패턴 클러스터 다양성 적용된 조건문 변환 을 사용하면 이러한 문제를 해결하는 데 도움이 될 수 있다.

8.1 기계적으로 코드 비교하기

의도 모든 소스 코드 파일을 한 줄씩 비교하여 중복 코드 (*duplicated code*) 발견하기.

문제

애플리케이션 코드의 어느 부분이 중복되었는지 어떻게 발견하는가?

이 문제는 다음과 같은 이유로 어렵다.

- 코드가 중복되었다고 의심할 수 있지만 중복이 발생한 위치에 대한 증거가 없다. 예를 들어 두 명의 프로그래머가 일부 코드를 복제하지 않고는 1년 동안 4백만 줄의 코드를 개발할 수 없다는 것을 알고 있다.
- 코드를 검색하는 것은 중복을 발견하는 효과적인 방법이 아니며, 우연히 중복된 코드를 발견할 수 있을 뿐이다.
- 프로그래머는 코드를 복사하여 붙여넣었을 뿐만 아니라 변수를 수정하거나 프로그램의 모양을 변경했을 수도 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 대부분의 중복 코드는 기계적 절차를 통해 감지할 수 있다.

해결

소프트웨어 소스 코드의 각 줄을 다른 모든 코드 줄과 텍스트로 비교한다.

단계

- 주석, 탭, 공백을 제거하여 코드 줄을 정규화한다.
- 흥미롭지 않은 코드 요소가 포함된 줄을 제거한다(예: 그냥 `else` 또는 `}`).
- 각 줄을 다른 모든 줄과 비교한다. 해싱으로 검색 복잡성을 줄인다.
 - 전처리: 각 줄의 해시 값 계산한다.

- 실제 비교: 동일한 해시 벅킷의 모든 줄을 비교한다.

변형

이 방법은 변수 이름 변경으로 인해 중복된 코드의 일부 인스턴스를 식별하지 못할 수 있다. 모든 변수 식별자를 삭제하거나 공통 기호에 매핑하면 특정 식별자의 세부 사항을 추상화하면서 유사한 코드 패턴을 감지할 수 있다. 그러나 이 변형은 일부 코드의 시맨틱 처리(syntactic processing)를 필요로 한다.

트레이드오프

장점

- 이 접근 방식은 간단하며 적당한 리소스만 사용해도 좋은 결과를 제공한다.
- 전체 구문 분석기(parser)가 아니라 어휘 분석기(lexical analyzer)만 구축하면 된다는 점에서 언어에 거의 독립적이다. 그렇기 때문에 원하는 정교함의 수준에 따라 간단한 perl 스크립트로도 충분할 수 있다.
- 간단한 통계와 비율은 쉽게 계산할 수 있으며 리소스 할당이나 신규 인력 채용에 대한 논의에서 신뢰성을 높이거나 더 많은 힘을 얻는데 도움이 될 수 있다.

단점

- 복사 후 많이 편집된 코드는 중복된 코드로 식별하기 어려울 수 있다.
- 중복 코드가 많이 포함된 시스템은 많은 데이터를 생성하여 효과적으로 분석하기 어려울 수 있다

예시

코드 중복이 의심되는 C++로 작성된 시스템의 경우를 생각해 보자. 하지만 직접 코드를 작성하지 않았기 때문에 실제 중복이 어디에서 발생하는지 알 수 없

다. 중복 코드 조각이 어디에 있는지 어떻게 감지할 수 있을까? 가장 간단한 방법은 먼저 코드를 정규화하여 코드에서 모든 공백을 제거한 다음 각 코드 줄을 자체적으로 비교하는 작은 스크립트를 작성하는 것이다.

정규화하면 다음 코드가 변경된다.

```
...
// assign same fastid as container
fastid = NULL;
const char* fidptr = getFastid();
if(fidptr != NULL) {
    int l = strlen(fidptr);
    fastid = new char[l+1];
    char *tmp = (char*) fastid;
    for (int i = 0; i < l; i++)
        tmp[i] = fidptr[i];
    tmp[l] = '\0';
}
...
```

into

```
...
fastid=NULL;
const char* fidptr=getFastid();
if(fidptr!=NULL)
int l=strlen(fidptr);
fastid=new char[l+1];
char* tmp=(char*) fastid;
for(int i=0;i<l;i++)
tmp[i]=fidptr[i];
tmp[l]='\0';
...
```

그 후 코드를 한 줄씩 비교하면 중복된 줄의 순서를 알려주는 보고서가 생성된다.

```
Lines:fastid=NULL;;const char* fidptr = getFastid();;if(fidptr!=NULL);
int l = strlen(fidptr);;fastid = new char[l+1];;
Locations:
</typesystem/Parser.C>6178/6179/6180/6181/6182
</typesystem/Parser.C>6198/6199/6200/6201/6202
```

다음은 이 기능을 수행하는 perl 스크립트 샘플이다.

```
#!/usr/bin/env perl -w
# duplocForCPP.pl – detect duplicated lines of code (algorithm only)
# Synopsis: duplocForCPP.pl filename ...
# Takes code (or other) files and collects all line numbers of lines
# equal to each other within these files. The algorithm is linear (in
# space and time) to the number of lines in input.

# Output: Lists of numbers of equal lines.
# Author: Matthias Rieger

$equivalenceClassMinimalSize = 1;
$slidingWindowSize = 5;
$removeKeywords = 0;

@keywords = qw(if
    then
    else
    for
    {
    }
);

$keywordsRegExp = join '|', @keywords;

@unwantedLines = qw( else
    return
    return;
    return result;
}else{
#else
#endif
{
}
;
};

push @unwantedLines, @keywords;

@unwantedLines{@unwantedLines} = (1) x @unwantedLines;

$totalLines = 0;
$emptyLines = 0;
$codeLines = 0;
@currentLines = ();
@currentLineNos = ();
%eqLines = ();
```

```
$inComment = 0;  
  
$start = (times)[0];  
  
while (<>) {  
    chomp;  
    $totalLines++;  
  
    # remove comments of type /* */  
    my $codeOnly = " ";  
    while(($inComment && m|\/*|) || (!$inComment && m|\*/|)) {  
        unless($inComment) { $codeOnly .= $` }  
        $inComment = !$inComment;  
        $_ = $';  
    }  
    $codeOnly .= $_ unless $inComment;  
    $_ = $codeOnly;  
  
    s|//.*$||; # remove comments of type //  
    s/\s+/g; #remove white space  
    s/$keywordsRegExp//og if $removeKeywords; #remove keywords  
  
    # remove empty and unwanted lines  
    if((!$_ && $emptyLines++)  
       || (defined $unwantedLines{$_} && $codeLines++)) { next }  
  
    $codeLines++;  
    push @currentLines, $_;  
    push @currentLineNos, $.;  
    if($slidingWindowSize < @currentLines) {  
        shift @currentLines;  
        shift @currentLineNos;  
    }  
  
    # print STDERR "Line $totalLines >$_<\n";  
  
    my $lineToBeCompared = join ", ", @currentLines;  
    my $lineNumbersCompared = "<$ARGV>"; # append the name of the  
    file  
    $lineNumbersCompared .= join '/', @currentLineNos;  
    # print STDERR "$lineNumbersCompared\n";  
    if($bucketRef = $eqLines{$lineToBeCompared}) {  
        push @$bucketRef, $lineNumbersCompared;  
    } else {  
        $eqLines{$lineToBeCompared} = [ $lineNumbersCompared ];  
    }  
}
```

```

if(eof) { close ARGV } # Reset linenumbers-count for next file
}

$end = (times)[0];
$processingTime = $end - $start;

# print the equivalence classes

$numOfMarkedEquivClasses = 0;
$numOfMarkedElements = 0;
foreach $line (sort { length $a <= length $b } keys %eqLines) {
    if(scalar @{$eqLines{$line}} > $equivalenceClassMinimalSize) {
        $numOfMarkedEquivClasses++;
        $numOfMarkedElements += scalar @{$eqLines{$line}};
        print "Lines: $line\n";
        print "Locations: @{$eqLines{$line}}\n\n";
    }
}

print "\n\n\n";
print "Number of Lines processed: $totalLines\n";
print "Number of Empty Lines: $emptyLines\n";
print "Number of Code Lines: $codeLines\n";
print "Scanning time in seconds: $processingTime\n";
print "Lines per second: @{[$totalLines/$processingTime]}\n";
print "-----\n";
print "Total Number of equivalence classes: @{[scalar keys %eqLines]}\n";
print "Size of Sliding window: $slidingWindowSize\n";
print "Lower bound of equiv-class Size: $equivalenceClassMinimalSize\n";
print "Number of marked equivalence classes:
    $numOfMarkedEquivClasses\n";
print "Number of marked elements: $numOfMarkedElements\n";

```

알려진 용도

소프트웨어 리엔지니어링의 맥락에서, 이 패턴은 최대 100만 줄의 C++ 가 포함된 FAMOOS 사례 연구에서 중복 코드를 탐지하는 데 적용되었다. 또한 4백만 줄의 코드가 포함된 COBOL 시스템에서 중복 코드를 탐지하는 데도 적용되었다. DATRIX 는 대규모 통신 시스템의 여러 버전을 조사하여 총 8,900만 줄의 코드를 모두 [LPM⁺⁹⁷]로 분석했다.

8.2 도트 플롯으로 코드 시각화하기

의도 도트 플롯의 패턴을 연구하여 중복의 본질에 대한 통찰력을 얻는다.

문제

소프트웨어 시스템에서 코드 중복의 범위와 성격에 대한 통찰력을 어떻게 얻을 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 시스템에서 중복된 코드가 어디에 존재하는지 아는 것만으로는 그 성격을 이해하거나 이에 대해 무엇을 해야 하는지 파악하는 데 도움이 되지 않는다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 하나의 그림이 천 마디 말보다 가치가 있다.

해결

두 축이 각각의 소스 코드 파일(동일한 파일일 수 있음)을 나타내는 행렬로 시각화하고 점들은 소스 코드 줄이 중복되는 곳에 표시한다.

단계

두 개의 파일 A와 B를 분석하려는 경우 다음과 같다.

- 두 파일의 내용을 정규화하여 노이즈(공백 등)를 제거한다.
- 행렬의 각 축이 정규화된 파일의 요소(예: 코드 줄)를 나타내도록 한다.
- 두 요소 사이의 일치를 행렬에 점으로 표시한다.
- 획득한 그림을 해석한다. 대각선은 두 파일 간에 중복된 코드를 나타낸다.

단일 파일 내부의 중복을 분석하기 위해 해당 파일의 요소를 양쪽 축에 플롯 한다.

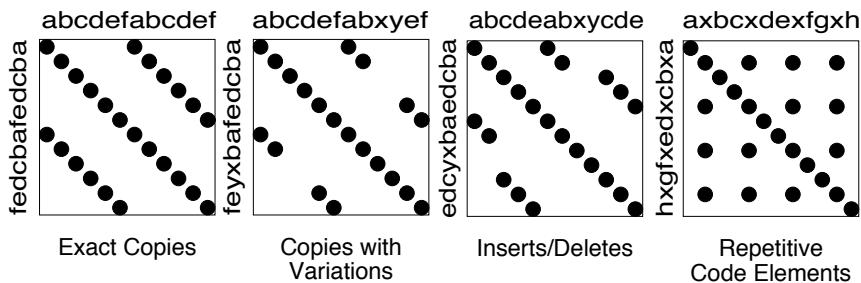


그림 8.2: 가능한 점의 시퀀스 및 관련 해석.

해석

얻어진 행렬의 해석은 그림 8.2에 설명되어 있다.

그림에 표시된 점으로 형성되는 몇 가지 흥미로운 구성은 다음과 같다.

- 정확한 복사본: 점의 대각선은 소스 코드의 복사된 시퀀스를 나타낸다.
- 변형이 있는 복사본: 시퀀스에 구멍이 있는 것은 복사된 시퀀스의 일부가 변경되었음을 나타낸다.
- 인서트/삭제: 오른쪽이나 왼쪽으로 이동된 부분이 있는 끊어진 시퀀스는 코드의 일부가 삽입되거나 삭제되었음을 나타낸다.
- 반복코드 요소: 직사각형 구성은 동일한 코드가 주기적으로 발생함을 나타낸다. 예를 들어 C 또는 C ++ 스위치 문의 개별 대소문자 끝의 쉼표나 #ifdef 일부 조건과 같은 반복되는 전처리기 명령이 있다.

트레이드오프

장점

- 이 접근 방식은 코드 정규화만 언어 구문에 의존하기 때문에 언어에 크게 구애받지 않는다.
- 이 접근 방식은 도트 플롯을 통해 코드의 특정 부분을 더 자세히 살펴볼 수 있기 때문에 알려지지 않은 대량의 코드를 리버스 엔지니어링할 때

효과적이다.

- 이 아이디어는 간단하지만 놀라울 정도로 잘 작동한다. 이 접근법의 간단한 버전은 유능한 프로그래머라면 적절한 도구를 사용하여 며칠 안에 구현할 수 있다. (실력 있는 학생 중 한 명은 이를 만에 엘파이로 작은 도트플롯 브라우저를 만들었다.)

단점

- 도트 플롯은 쌍으로만 비교를 표시한다. 전체 소프트웨어 시스템에서 중복된 요소의 모든 인스턴스를 식별하는 데 반드시 도움이 되는 것은 아니다. 이 접근 방식을 쉽게 확장하여 각 축에 여러 파일을 표시할 수 있지만 여전히 비교는 쌍 단위로만 이루어진다.

어려움

- 도트 플롯 시각화 도구의 나이브한 구현은 대규모 시스템의 확장이 쉽지 않을 수 있다. 대규모 데이터 집합에 맞게 접근 방식을 조정하고 최적화하면 접근 방식의 단순성이 저하될 수 있다.
- 데이터의 해석은 언뜻 보기보다 더 미묘할 수 있다. 실제로 여러 파일을 비교하는 동안 대각선은 실제보다 더 많은 중복을 나타내며, 이는 그림 8.3 및 그림 8.4에 표시된 것처럼 여러 파일에서 중복된 조각을 비교하고 있기 때문이다.
- 화면 크기는 시각화할 수 있는 정보의 양을 제한한다. 소위 “벽화” 시각화 (mural visualization) 접근 방식인 [JS96]로 일부 성공을 거두었다. 그러나 이러한 기법은 단순한 도트 플롯보다 구현하기가 훨씬 더 어렵고 추가적인 노력을 기울일 가치는 없다.

예시

그림 8.3에서는 중복이 제거되기 전과 후의 두 가지 버전의 소프트웨어에 대한 dotplot을 볼 수 있다. 첫 번째 버전은 왼쪽 위 사각형에 비교되어 있다. 대각선 아래 선은 모든 코드 줄이 그 자체와 비교되고 있음을 보여준다. 더 흥미로운

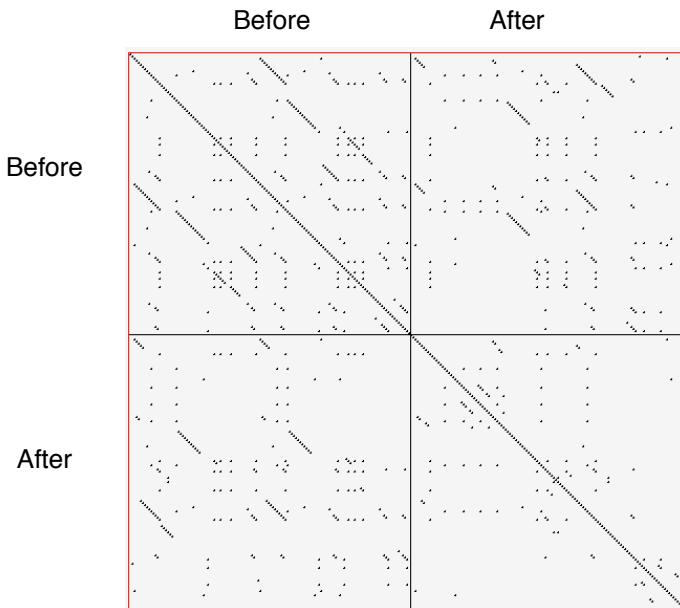


그림 8.3: 리팩토링 전후의 코드 중복.

점은 도트 플롯에 여러 개의 다른 대각선이 나타나는데, 이는 이 파일 내에서 코드가 중복되었다는 것을 의미한다. 동일한 파일의 두 번째 버전이 오른쪽 아래 사각형에서 자신과 비교되고 있다. 여기에서는 주 대각선을 제외하고는 크게 중복된 부분이 없는데, 이는 중복된 코드가 모두 성공적으로 리팩터링되었다는 사실을 반영한다.

왼쪽 아래 사각형과 오른쪽 위 사각형은 서로의 거울 이미지이다. 이전과 이후 파일이 어떻게 재구성되었는지 알려준다. 강한 대각선이 없으므로 상당한 재구성이 이루어졌음을 알 수 있다. 대각선 줄무늬는 이전 버전에서 어떤 부분이 살아남았고 새 버전에서 어디에 나타나는지 보여준다. 도트 플롯만 보면 코드의 약 절반이 살아남았고 나머지 절반의 코드가 크게 재작성되었음을 짐작할 수 있다.

도트 플롯은 여러 파일에서 중복을 감지하는 데도 유용하다. 그림 8.4 은 두 개의 Python 파일을 비교한 도트 플롯을 보여준다. A와 A를 비교하면 본질적으로 내부 중복이 없음을 알 수 있다. 파일의 하단에는 매트릭스 패턴으로

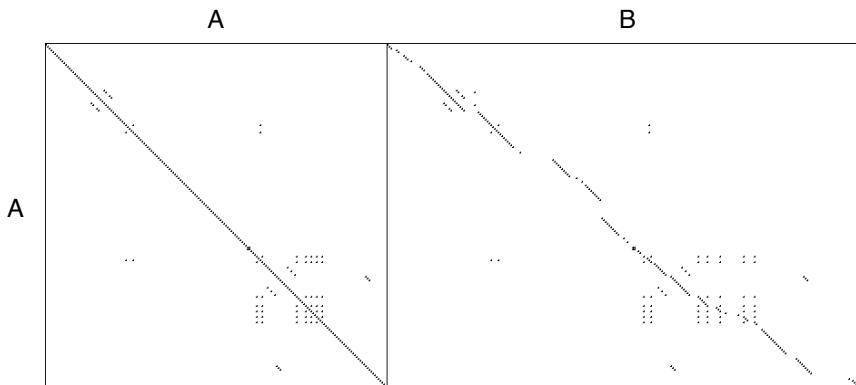


그림 8.4: 파일 A에 대해서 자신과 두 번째 파일 B를 비교하고 있다.

표시된 스위치 문이 있을 가능성이 높다.

하지만 파일 A와 파일 B를 비교하면 엄청난 양의 중복을 발견할 수 있다. 파일 B는 파일 A를 다양한 방식으로 확장한 복사본에 불과한 것처럼 보인다. 자세히 조사한 결과 사실로 밝혀졌다. 실제로 파일 A는 실수로 릴리스에 남겨진 파일 B의 이전 버전에 불과했다.

도트 플롯은 다른 문제를 감지하는 데도 유용할 수 있다. 그림 8.5는 개별 구성 코드를 호출하는 데 사용되는 유형 변수에 대한 스위치 문을 나타내는 네 개의 복제본을 표시한다. 중복된 코드는 아마도 다형성 적용한 조건문 변환 을 적용하여 제거할 수 있을 것이다.

알려진 용도

이 패턴은 생물학적 연구에 적용되어 DNA 서열 [PK82]을 검출하는 데 사용되었다. 도트플롯 도구 [Hel95]는 수동 페이지, 문학 텍스트 및 파일 시스템의 이름에서 유사성을 감지하는 데 사용되었다. FAMOOS 프로젝트에서 이 패턴은 소프트웨어 소스 코드의 중복을 식별하는 도구 [DRD99]인 Duploc 을 구축하는 데 적용되었다. Dup 도구 [Bak92]는 X-Window 시스템의 소스 코드를 조사하는 데 사용되었으며, 도트 플롯 매트릭스 그래픽 표현을 사용한다.

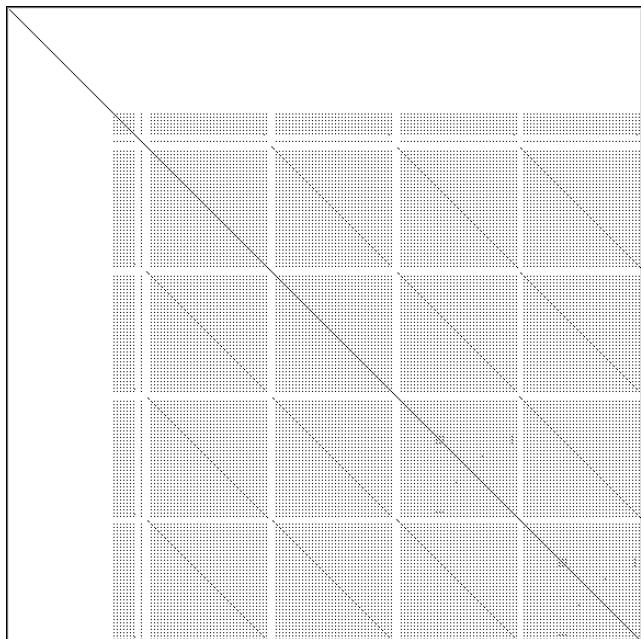


그림 8.5: 4개의 스위치 문으로 생성된 도트 플롯이다.

관련 패턴

중복 코드를 감지한 후에는 특히 메시드 추출하기 [p. 347] 와 같은 수많은 리팩터링 패턴이 적용될 수 있다.

클라이언트가 너무 많은 책임을 맡기 때문에 중복 코드가 발생하는 경우가 많다. 이 경우 데이터 가까이 동작 이동하기 [p. 269] 를 사용하면 중복을 제거할 수 있다.

도트 플롯은 또한 큰 조건부 구조를 감지하는 데 도움이 된다. 이러한 조건문을 제거하여 보다 유연한 설계를 달성하려면 다형성 적용된 조건문 변환 을 사용해야 한다.

제 9 장

책임 재배포

여러분은 대규모 공공 행정 기관의 모든 직원 기록을 관리하는 정보 시스템을 재설계하는 업무를 담당하고 있다. 최근의 정치적 격변으로 인해 민영화, 새로운 법률, 새로운 규제에 대응하기 위해 시스템에 많은 변화가 필요하다는 것을 알고 있지만 정확히 어떤 변화가 있을지는 모른다. 기존 시스템은 명목상으로는 오래된 절차지향 시스템을 객체 지향으로 재구현한 것으로 구성되어 있다. 이 코드에는 객체를 가장한 데이터 컨테이너(data container)와 개별 하위 시스템의 로직 대부분을 구현하는 거대한 프로시저 '신 클래스(God Class)' 등 많은 의사 객체가 포함되어 있다. TaxRevision2000이라는 한 클래스는 기본적으로 3000줄 길이의 케이스 문으로 구성된 단일 메서드를 가지고 있다.

시스템이 비교적 안정적인 한 이 설계는 특별한 문제를 일으키지 않았지만, 이제는 데이터 캡슐화(encapsulation)가 취약하여 비교적 작은 시스템 변경에도 수개월의 계획, 테스트 및 디버깅이 필요하다는 것을 알게 되었다. 보다 객체 지향적인 디자인으로 마이그레이션하면 시스템이 더 견고해지고 향후 요구 사항에 더 쉽게 적응할 수 있을 것이라고 확신한다. 하지만 어디에 문제가 있는지 어떻게 알 수 있을까? 어떤 책임을 재분배해야 할까? 어떤 데이터 컨테이너를 재설계해야 하고, 어떤 컨테이너를 래핑해야 하며, 어떤 컨테이너는 그대로 두는 것이 더 나을까?

포스

- 데이터 컨테이너(데이터에 대한 액세스만 제공하고 자체 동작은 없는 객체)는 여러 하위 시스템 간에 정보를 공유하는 간단하고 편리한 방법이다. 그중에서도 데이터 컨테이너는 데이터베이스 엔티티에 대한 액세스를 제공하는 가장 쉬운 방법이다.
- 그러나 데이터 컨테이너는 데이터 표현을 노출하므로 많은 애플리케이션 컴포넌트가 데이터 컨테이너에 의존하는 경우 변경하기 어렵다. 결과적으로 데이터 컨테이너가 확산되면 비즈니스 로직을 구현할 때 탐색 코드가 취약해진다.
- 늙은 개가 새로운 교육을 받아들이는 것은 어렵다. 많은 디자이너가 기능적 분해(functional decomposition)에 대한 교육을 받았기 때문에 객체 디자인을 할 때도 동일하게 작업한다.
- 하지만 기능적 분해는 모든 작업을 수행하는 큰 클래스와 그 주위에 무수히 많은 작은 공급자 클래스를 포함하는 신 클래스를 생성하는 경향이 있다. 신 클래스는 다른 메서드나 인스턴스 변수에 많은 영향을 미치기 때문에 확장, 수정 또는 서브클래싱하기가 어렵다.

개요

이 클래스터는 잘못 배치된 책임 문제를 다룬다. 두 가지 극단적인 사례는 미화된 데이터 구조에 불과하고 식별 가능한 책임이 거의 없는 클래스인 데이터컨테이너와 너무 많은 책임을 맡는 프로시저 괴물인 신 클래스이다.

데이터 컨테이너와 신 클래스가 시스템의 안정적인 부분에 묻혀서 변경되지 않는 경우와 같이 경계선에 있는 경우도 있지만, 일반적으로는 취약한 설계의 징후이다.

데이터 컨테이너는 디미터의 법칙 (Low of Demeter, LOD) [LHR88]¹의 위반으로 이어진다. 간단히 말해, 디미터의 법칙은 멀리 떨어져 있는 클래스 간의 결합도를 줄이기 위한 여러 가지 설계 지침을 제공한다. 디미터의 법칙은 객체

¹ 최소한의 지식 원칙(The Principle of Least Knowledge)을 강조하며 모듈은 자신이 조작하는 객체의 속사정을 몰라야 한다는 법칙-옮긴이

또는 클래스에 초점을 두느냐에 따라, 그리고 어떤 프로그래밍 언어를 사용하느냐에 따라 다양한 형태로 나타나지만, 기본적으로 메서드는 인스턴스 변수, 메서드 인자, 셀프(self), 슈퍼(super), 수신자 클래스에만 메시지를 보내야 한다는 것을 명시하고 있다.

디미터의 법칙 위반은 일반적으로 간접 클라이언트(indirect client)가 인스턴스 변수 또는 중간 공급자(intermediate provider)의 주변을 통해 간접 공급자(indirect provider)에 접근하는 탐색 코드(navigation code)의 형태를 취한다. 따라서 간접 클라이언트와 프로바이더는 불필요하게 결합도가 높아져 향후 개선 사항을 실현하기가 더 어려워진다(그림 9.2). 중간 공급자는 데이터 컨테이너의 형태를 취하거나 접근자 메서드를 제공하여 캡슐화되어 있는 데이터를 제공할 수 있다. 많은 데이터 컨테이너가 존재하는 디자인은 종종 간접 클라이언트가 간접 제공자에게 도달하기 위해 일련의 중간체를 탐색해야 하는 복잡한 탐색 코드로 인해 어려움을 겪을 수 있다.

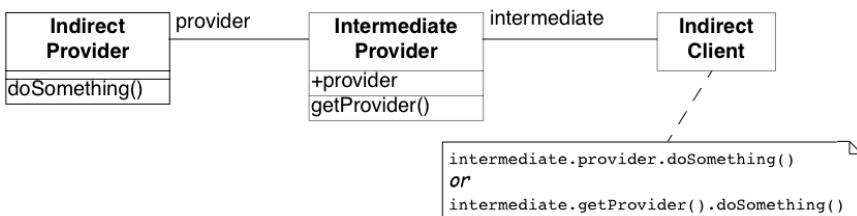


그림 9.1: 간접 클라이언트가 중간 공급자를 통해 간접 공급자로 이동하여 두 공급자를 불필요하게 결합함으로써 디미터의 법칙을 위반한다.

데이터 컨테이너는 책임이 너무 적은 반면, 신 클래스는 너무 많은 책임을 맡는다. 신 클래스는 수천 줄의 코드와 수백 개의 메서드 및 인스턴스 변수로 구성된 전체 하위 시스템을 구현하는 단일 클래스일 수 있다. 특히 악의적인 신 클래스는 정적 인스턴스 변수와 메서드로만 구성되며, 즉 모든 데이터와 동작이 클래스 범위를 가지며, 신 클래스는 인스턴스화되지 않는다. 이러한 신 클래스는 순전히 괴물과 같은 프로시저에 불과하며 이름만 객체 지향적이다.

때때로 유ти리티 클래스(utility class)로 알려진 일부 프로시저 클래스가 편리하기도 하다. 가장 잘 알려진 예는 수학 라이브러리에 대한 객체 지향 인터페이스나 알고리즘 모음이다. 그러나 실제 신 클래스는 라이브러리가 아니라 전체

애플리케이션 실행을 제어하는 완전한 애플리케이션 또는 하위 시스템이다.

신 클래스와 데이터 컨테이너는 종종 함께 존재하며, 신 클래스가 애플리케이션의 모든 제어를 맡고 다른 클래스는 영광스러운 데이터 구조로 취급한다. 신 클래스는 너무 많은 책임을 맡기 때문에 이해하고 유지하기가 어렵다. 인터페이스의 복잡성과 명확한 서브클래싱 계약의 부재로 인해 상속을 통한 신 클래스의 점진적 수정 및 확장은 거의 불가능에 가깝다.

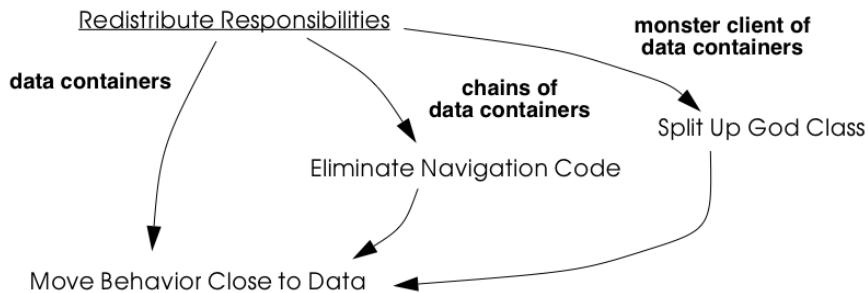


그림 9.2: 데이터 컨테이너는 잘못 배치된 책임의 가장 명확한 신호이다. 이 세 가지 패턴은 동작을 데이터에 가깝게 이동하여 책임을 재분배한다.

이 클러스터는 책임을 재분배하여 데이터 컨테이너와 신 클래스를 제거함으로써 캡슐화를 개선하는 여러 패턴을 제공한다.

- 동작을 데이터 가까이 이동하기 [p. 269]는 간접 클라이언트에 정의된 동작을 중간 데이터 컨테이너로 이동하여 보다 “객체처럼” 만든다. 이 패턴은 간접 클라이언트를 데이터 컨테이너의 콘텐츠에서 분리할 뿐만 아니라 일반적으로 데이터 컨테이너의 여러 클라이언트에서 발생하는 중복된 코드를 제거한다.
- 탐색 코드 제거하기 [p. 279]는 리엔지니어링 단계 측면에서 동작을 데이터 가까이 이동하기 과 기술적으로 매우 유사하지만 그 의도는 다소 다르다. 이 패턴은 탐색 코드를 제거하기 위해 데이터 컨테이너 체인 아래로 책임을 재분배하는 데 중점을 둔다.
- 신 클래스 분할하기 [p. 289]는 모든 데이터를 외부 데이터 컨테이너로 이동하고, 데이터 컨테이너를 객체로 승격하기 위해 동작을 데이터 가까이 이동하기를 적용하고, 마지막으로 남아 있는 파사드를 제거함으로써

프로시저 신 클래스를 더 단순하고 응집도 높은 여러 클래스로 리팩터링 한다.

9.1 동작을 데이터 가까이 이동하기

의도 간접 클라이언트의 동작(behavior)을 동작이 처리하는 데이터가 포함된 클래스로 이동하여 캡슐화를 강화한다.

문제

클래스를 단순한 데이터 컨테이너에서 실제 서비스 프로바이더로 전환하려면 어떻게 해야 할까?

이 문제는 다음과 같은 이유로 어렵다.

- 데이터 컨테이너는 실제 동작이 아닌 접근자 메서드(access method)나 퍼블릭 인스턴스 변수(public instance variable)만 제공하므로 클라이언트가 동작을 사용하는 대신 직접 동작을 정의해야 한다. 새 클라이언트는 일반적으로 이 동작을 다시 구현해야 한다.
- 데이터 컨테이너의 내부 표현이 변경되면 많은 클라이언트를 업데이트해야 한다.
- 데이터 컨테이너는 동작을 정의하지 않고 인터페이스가 주로 접근자 메서드로 구성되어 있기 때문에 다양성으로 사용할 수 없다. 따라서 클라이언트는 주어진 컨텍스트에서 어떤 동작이 필요한지 결정할 책임이 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 클라이언트가 데이터로 어떤 작업을 수행하는지 알고 있다.

해결

간접 클라이언트에 의해 정의된 동작을 해당 동작이 수행되는 데이터의 컨테이너로 이동한다.

탐지

다음 사항들을 찾는다.

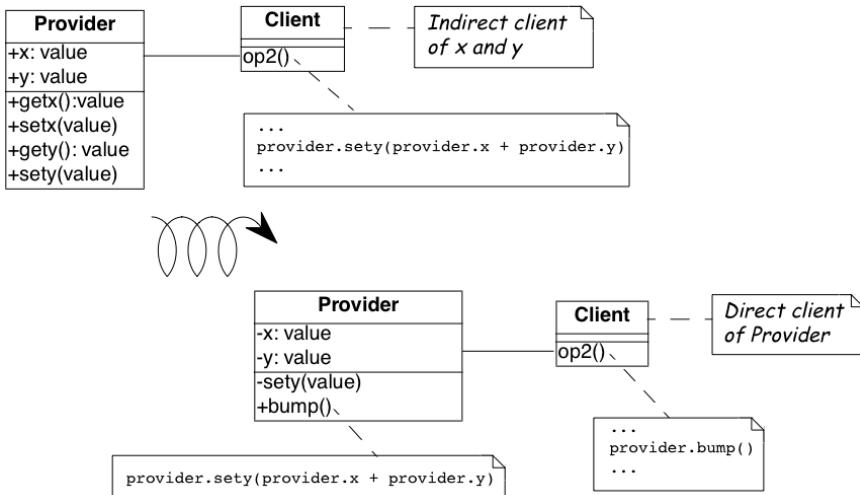


그림 9.3: 단순 데이터 컨테이너에 불과했던 클래스가 실제 서비스 프로바이더로 전환된다

- 데이터 컨테이너, 즉 대부분 퍼블릭 접근자 메서드와 몇 가지 동작 메서드를 정의하는 클래스(즉, 메서드 수가 속성 수보다 약 2배 더 많음).
- 별도의 공급자 클래스의 데이터를 조작하는 중복된 클라이언트 코드. 여러 클라이언트가 다른 동작을 구현하는 경우, 대신 클라이언트 타입 검사 변환하기 [p. 311]를 적용하는 것을 고려하자.
- 클라이언트 클래스에서 일련의 접근자 메서드를 호출하는 메서드(탐색 코드 제거하기 참조).

단계

동작을 데이터 가까이 이동하기는 Extract 메서드 추출하기 [p. 347] 및 메서드 이동하기 [p. 348]를 사용하여 리팩터링한다. 문제의 동작을 클라이언트 메서드에서 추출한 다음 프로바이더 클래스로 이동해야 하기 때문이다.

- 이동하려는 클라이언트 동작을 식별한다, 즉 전체 메서드 또는 프로바이더 데이터에 액세스하는 메서드의 일부이다.

- 데이터 컨테이너의 접근자 메서드 호출을 찾는다.
 - 동일한 프로바이더 데이터에 액세스하는 여러 클라이언트에서 중복된 코드를 찾는다.
2. 프로바이더 클래스에 해당 메서드가 없는 경우 프로바이더 클래스에 해당 메서드를 생성한다. 코드를 이동해도 이름 충돌이 발생하지 않는지 확인하자. 리팩터링 브라우저 [RB97]와 같은 도구는 이러한 단계를 자동화 한다.
- 추출된 기능이 인수가 있는 완전한 메서드인 경우 인수가 프로바이더 클래스의 속성과 충돌하지 않는지 확인한다. 충돌하는 경우 인수의 이름을 바꾼다.
 - 추출된 함수가 임시 변수를 사용하는 경우 로컬 변수가 대상 범위의 속성 또는 변수와 충돌하지 않는지 확인합니다. 충돌하는 경우 임시 변수의 이름을 바꾼다.
 - 추출된 함수가 클라이언트 클래스의 로컬 변수(속성, 임시 변수...)에 액세스하는지 확인하고, 그렇다면 이러한 클라이언트 변수를 나타내는 메서드에 인자를 추가한다.
3. 새 메서드에 의도가 드러나는 이름을 지정한다. 무엇보다도 의도를 드러내기 이름에는 메서드가 속한 클래스에 대한 참조가 포함되지 않으므로 메서드의 재사용성이 떨어진다. 예를 들어, Set 클래스에 `addToSet()` 메서드를 정의하는 대신 단순히 `add()`로 이름을 지정하는 것이 좋다. 마찬가지로, 메서드 이름에는 정렬된 랜덤 액세스 컬렉션을 암시하는 반면 `search()`라는 이름에는 그러한 의미가 없으므로 `binarySearch()` 메서드를 Array 클래스에 정의하는 것은 좋은 생각이 아니다.
4. 클라이언트에서 올바른 매개변수를 사용하여 새 프로바이더 메서드를 호출한다.
5. 클라이언트 코드를 정리한다. 이동된 기능이 클라이언트 클래스의 완전한 메서드인 경우.
- 이전 메서드를 호출하는 모든 메서드를 검사하고 이제 새 프로바이더 메서드를 대신 호출하는지 확인한다.

- 클라이언트에서 이전 메서드를 제거하거나 지원 중단한다. (폐기된 인터페이스 지원 중단하기 [p. 237]).

동일한 객체에 정의된 호출 메서드도 프로바이더로 이동해야 하는 경우가 있을 수 있다. 이러한 경우 메서드를 위한 단계를 반복한다.

6. 여러 클라이언트에 대해 반복한다. 이미 프로바이더로 전송된 코드를 이동할 필요가 없으므로 여러 클라이언트에서 중복된 코드는 2단계에서 제거된다. 프로바이더에 동일하지는 않지만 유사한 메소드가 많이 도입된 경우 중복된 조각을 보호된 헬퍼 메소드로 간주하는 것이 좋다.

트레이드오프

장점

- 데이터 컨테이너가 명확한 책임이 있는 서비스 프로바이더로 전환된다.
- 서비스 프로바이더는 다른 클라이언트에게 더 유용해진다.
- 클라이언트는 더 이상 프로바이더 동작을 구현할 책임이 없다.
- 클라이언트는 프로바이더의 내부 변경에 덜 민감하다.
- 시스템 내 코드 중복이 감소한다.

단점

- 이동된 동작이 클라이언트 데이터에도 액세스하는 경우 이러한 액세스를 매개변수로 전환하면 프로바이더의 인터페이스가 더 복잡해지고 프로바이더에서 클라이언트로의 명시적 종속성이 도입된다.

어려움

- 클라이언트 코드를 실제로 데이터 프로바이더로 옮겨야 하는지 여부가 명확하지 않을 수 있다. `IctStream` 또는 `Set`과 같은 일부 클래스는 실제로 데이터 프로바이더로 설계되었다. 다음과 같은 경우 코드를 프로바이더로 옮기는 것을 고려하자.

- 기능(functionality)은 프로바이더의 책임을 나타낸다. 예를 들어 클래스 Set은 합집합과 교집합과 같은 수학적 연산을 제공해야 한다. 반면에 일반 Set은 Employees 집합에 대한 연산을 담당해서는 안 된다.
 - 함수는 프로바이더의 속성에 액세스한다,
 - 기능이 여러 클라이언트에 의해 정의된다.
- 프로바이더가 실제로 데이터 컨테이너로 설계된 경우 데이터 프로바이더의 인스턴스를 래핑하고 관련 동작을 보유하는 새 프로바이더 클래스를 정의하는 것을 고려하자. 예를 들어, EmployeeSet은 Set 인스턴스를 래핑하고 더 적합한 인터페이스를 제공할 수 있다.

레거시 솔루션이 최종 솔루션인 경우

데이터 컨테이너는 기존 데이터베이스에 객체 인터페이스를 제공하기 위해 데이터베이스 스키마에서 자동으로 생성되었을 수 있다. 코드를 다시 생성해야 하는 경우 변경 사항을 쉽게 되므로 생성된 클래스를 수정하는 것은 거의 항상 좋지 않은 생각이다. 이 경우 래퍼 클래스를 구현하여 생성된 클래스와 연관되어야 하는 동작을 보유할 수 있다. 이러한 래퍼는 생성된 데이터 컨테이너를 실제 서비스 프로바이더로 변환하는 어댑터 [p. 349]로 작동한다.

라이브러리에 정의된 클래스에 중요한 기능이 누락되어 있는 경우가 있다. 예를 들어, String 클래스에 convertToCapitals 연산이 누락되어 있는 경우를 들 수 있다. 이러한 경우 일반적으로 라이브러리에 코드를 추가하는 것이 불가능하므로 클라이언트 클래스에서 정의해야 할 수 있다. 예를 들어 C++에서는 코드를 사용할 수 없을 때 재컴파일을 피하거나 클래스를 확장하는 유일한 방법 일 수 있다 [ABW98] (378페이지). Smalltalk에서는 라이브러리를 확장하거나 수정할 수 있지만, 추가 코드를 분리하여 향후 라이브러리 릴리스와 쉽게 병합하고 충돌을 빠르게 감지할 수 있도록 각별히 주의해야 한다.

비지터 [p. 353] 디자인 패턴의 의도는 다음과 같다: “객체 구조의 요소에 대해 수행할 연산을 요소 자체와는 별도로 클래스에 표시한다. 비지터를 사용하면 연산이 수행되는 요소의 클래스를 변경하지 않고도 새로운 연산을 정의할 수 있다.” [GHJV95]. 비지터 패턴은 클래스가 별도의 프로바이더 클래스의 데이터에 액세스하도록 하려는 몇 안 되는 경우 중 하나이다. 비지터를 사용하면

안정된 클래스 집합을 변경하지 않고도 새 연산을 동적으로 추가할 수 있다.

구성 클래스(*configuration class*)는 시스템의 구성을 나타내는 클래스이다(예: 전역 매개변수, 언어에 따른 표현, 적용 중인 정책). 예를 들어 그래픽 도구에서 상자의 기본 크기, 가장자리, 선의 너비 등을 이러한 클래스에 저장하고 필요할 때 다른 클래스가 이를 참조할 수 있다.

매핑 클래스(*mapping class*)는 객체와 객체의 사용자 인터페이스 또는 데이터베이스 표현 간의 매핑을 나타내는 데 사용되는 클래스이다. 예를 들어, 소프트웨어 지표 도구는 사용자가 계산할 지표를 선택할 수 있도록 위젯 목록에서 사용 가능한 지표를 그래픽으로 표현해야 한다. 이러한 경우 서로 다른 메트릭의 그래픽 표현은 내부 표현과 분명히 다를 것이다. 매핑 클래스는 연관성을 추적한다.

예시

고객들의 반복되는 불만 중 하나는 정보 시스템에서 생성된 보고서를 변경하는데 너무 많은 시간이 걸린다는 것이다. 유지 관리자와 이야기를 나누다 보면 보고서 생성이 매우 지루하다는 것을 알게 된다. “항상 똑같은 코드를 작성해야 해요.”라고 유지관리자인 크리스가 말한다. “데이터베이스에서 레코드를 가져와서 해당 필드를 인쇄한 다음 다음 레코드로 넘어가면 되요.”

데이터 컨테이너의 경우를 강력하게 의심하고 코드를 자세히 살펴보면 의심을 확인할 수 있다. 데이터베이스와 인터페이스하는 거의 모든 클래스에는 접근자 메서드만 포함되어 있으며, 보고서를 생성하는 프로그램에서는 이러한 접근자를 사용할 수밖에 없다. 한 가지 눈에 띠는 예는 Payroll 애플리케이션의 경우로, TelephoneGuide 애플리케이션과 공통점이 많아서 공통 기능을 Employee 클래스로 옮기기로 결정한 경우이다.

예전

그림 9.4에 표시된 것처럼 Payroll 및 TelephoneGuide 클래스는 모두 Employee 인스턴스를 데이터 컨테이너로 취급하여 레이블을 인쇄한다. 따라서 Payroll과 TelephoneGuide는 Employee 속성의 간접 클라이언트이며, Employee 클래스에서 제공해야 하는 인쇄 코드를 정의한다. 다음 코드는 Java에

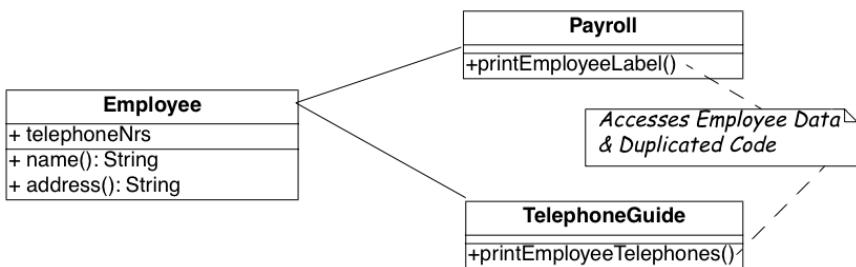


그림 9.4: Payroll 및 Telephone 클래스는 Employee 클래스의 내부 표현에 액세스하여 표현(representation)을 인쇄한다.

서 어떻게 표시되는지 보여준다.

```

public class Employee {
    public String[] telephoneNumbers = {};
    ...
    public String name() {
        return name;
    }
    public String address() {
        return address;
    }
}

public class Payroll {

    public static Employee currentEmployee;

    public static void printEmployeeLabel () {
        System.out.println(currentEmployee.name());
        System.out.println(currentEmployee.address());
        for (int i=0; i < currentEmployee.telephoneNumbers.length; i++) {
            System.out.print(currentEmployee.telephoneNumbers[i]);
            System.out.print(" ");
        }
        System.out.println("");
    }
}

public class TelephoneGuide {

    public static void printEmployeeTelephones (Employee emp) {
        System.out.println(emp.name());
        System.out.println(emp.address());
    }
}
  
```

```

for (int i=0; i < emp.telephoneNumbers.length - 1; i++) {
    System.out.print(emp.telephoneNumbers[i]);
    System.out.print(" -- ");
    System.out.print(emp.telephoneNumbers[
        emp.telephoneNumbers.length - 1]);
    System.out.println("");
}
...

```

두 인쇄 메서드는 본질적으로 동일한 기능을 구현하지만 약간의 차이점이 있다. 그 중에서도 `TelephoneGuide.printEmployeeTelephones`는 전화번호를 인쇄할 때 다른 구분 기호를 사용한다.

단계

사용할 구분 기호를 나타내는 특수 매개 변수를 정의하여 다른 구분 기호를 쉽게 처리할 수 있다. 따라서 `TelephoneGuide.printEmployeeTelephones`는 다음과 같이 재작성된다.

```

public static void printEmployeeTelephones
    (Employee emp, String separator) {
    ...
    for (int i=0; ...
        System.out.print(separator);}
    ...
}
...
```

다음으로, `printEmployeeTelephones` 메서드를 `TelephoneGuide`에서 `Employee`로 옮긴다. 따라서 코드를 복사하고 `emp` 매개 변수에 대한 모든 참조를 속성 및 메서드에 대한 직접 참조로 바꾼다. 또한 새 메서드에 의도를 드러내는 이름이 있는지 확인하여 메서드 이름에서 `Employee` 부분을 생략하여 `printLabel`이라는 메서드를 생성한다.

```

public class Employee {
    ...
    public void printLabel (String separator) {
        System.out.println(name);
        System.out.println(address);
        for (int i=0; i < telephoneNumbers.length - 1; i++) {
            System.out.print(telephoneNumbers[i]);
        }
    }
}
```

```

        System.out.print(separator);
    }
    System.out.print(telephoneNumbers[telephoneNumbers.length - 1]);
    System.out.println("");
}

```

그런 다음 `Payroll.printEmployeeLabel` 및 `TelephoneGuide.printEmployeeTelephones`의 메서드 본문을 `Employee.printLabel` 메서드의 간단한 호출로 대체한다.

```

public class Payroll {
    ...
    public static void printEmployeeLabel () {
        currentEmployee.printLabel(" ");
    ...
}

public class TelephoneGuide {
    ...
    public static void printEmployeeTelephones (Employee emp) {
        emp.printLabel(" -- ");
    ...
}

```

마지막으로 어떤 다른 메서드가 `name()`, `address()` 및 전화번호를 참조하는지 확인한다. 이러한 메서드가 존재하지 않는다면 해당 메서드와 속성을 `private`로 선언하는 것이 좋다.

이후

동작을 데이터 가까이 이동하기를 적용한 후 `Employee` 클래스는 이제 하나의 인수를 받아 다른 구분 기호를 나타내는 `printLabel` 메서드를 제공한다(그림 9.5 참조). 이제 클라이언트가 `Employee`의 내부 표현에 의존하지 않기 때문에 더 나은 상황이다. 또한 동작을 작동하는 데이터 근처로 이동시킴으로써 클래스는 구현하는 구조 대신 프로바이더가 제공하는 서비스에 중점을 둔 개념적 엔티티를 나타낸다.

근거

관련 데이터와 동작을 한 곳에 보관하자.

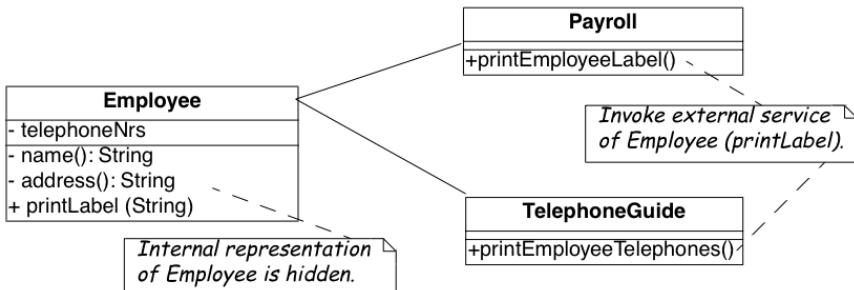


그림 9.5: Payroll 클래스는 Employee 클래스의 퍼블릭 인터페이스를 사용하여 Employee의 표현을 인쇄하며, 데이터 액세서는 비공개가 되었다.

— 아서 리엘, 휴리스틱 2.9 [Rie96]

데이터 컨테이너는 구조를 노출하고 클라이언트가 동작을 공유하기보다는 정의하도록 강요하기 때문에 진화를 저해한다. 서비스 프로바이더에게 데이터 컨테이너를 장려하면 클래스 간의 결합도를 줄이고 데이터와 동작의 응집도를 향상시킬 수 있다.

관련 패턴

필드 캡슐화하기 [p. 347] 는 설계 단계에서 메서드를 정의해야 하는 위치를 결정하는 데 도움이 되는 휴리스틱을 제공한다. 이 텍스트는 동작을 데이터 가까이 이동하기를 적용하기 위한 근거를 제공한다.

9.2 탐색 코드 제거하기

디미터의 법칙 [LHR88]으로도 알려져 있다

의도 연결된 클래스 체인으로 책임을 전가하여 변경의 영향을 줄인다.

문제

객체 그래프를 탐색하는 클래스로 인한 결합도를 줄이려면 어떻게 해야 할까?

이 문제는 다음과 같은 이유로 어렵다.

- 클래스의 인터페이스 변경은 직접 클라이언트뿐만 아니라 해당 클래스에 도달하기 위해 탐색하는 모든 간접 클라이언트에도 영향을 미친다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 탐색 코드는 일반적으로 잘못 배치된 책임과 캡슐화 위반의 신호이다.

해결

간접 클라이언트에 의해 정의된 동작을 해당 동작이 작동하는 데이터 컨테이너로 반복적으로 이동한다.

실제 리엔지니어링 단계는 기본적으로 동작을 데이터 가까이 이동하기의 단계와 동일하지만 문제의 양상이 다소 다르므로 다른 탐지 단계가 적용된다는 점에 유의하자.

탐지

간접프로파이더를 찾는다.

- 클래스가 변경될 때마다, 예를 들어 내부 표현이나 공동 작업자를 수정하여 변경할 때마다 직접 클래스뿐만 아니라 간접 클라이언트 클래스도 변경해야 한다.

- 퍼블릭 속성, 접근자 메서드 또는 클래스의 값 속성(value attribute)으로 반환되는 메서드가 많이 포함된 클래스를 찾는다.
- 대부분 데이터 클래스를 포함하는 큰 애플리케이션 계층 구조는 종종 간접 프로파이더 역할을 한다.

탐색 코드(navigation code)가 많이 포함된 간접 클라이언트를 찾는다. 탐색 코드는 두 가지 종류가 있다.

- 속성 액세스 시퀀스(sequence of attribute accesses), e.g.,**a.b.c.d** 여기서 **b**는 **a**의 속성, **c**는 **b**의 속성, **d**는 **c**의 속성이다. 이러한 시퀀스의 결과는 변수에 할당되거나 마지막 객체의 메서드가 호출될 수 있으며, e.g.,**a.b.c.d.op()**이 될 수 있다. 이러한 시퀀스 탐색은 모든 어트리뷰트가 보호되는 Smalltalk에서는 발생하지 않는다.
- 접근자 메서드 호출의 시퀀스(sequence of accessor method calls). Java와 C++에서 이러한 시퀀스는 **object.m1().m2().m3()**의 형태를 갖는다. 여기서 **object**는 객체를 반환하는 표현식, **m1**은 **object**의 메서드, **m2**는 **m1**의 호출로 반환된 객체의 메서드, **m3**의 호출로 반환된 객체의 메서드 등 등이 있다. Smalltalk에서 탐색 코드는 다음과 같은 형식의 리시버 **m1 m2 ... mn**을 갖는다. 동일한 탐색 코드 시퀀스는 동일하거나 다른 클라이언트에서 다른 메서드에서 반복된다.

탐색 코드는 간단한 패턴 매칭으로 감지할 수 있다. 그러나 실제로 결합된 클래스로 이어지는 메서드 호출 탐색 시퀀스를 탐지하려면 한 객체를 다른 객체로 변환하는 호출 시퀀스를 필터링해야 한다. 예를 들어, 다음 두 Java 표현식은 객체 변환을 다루기 때문에 문제가 되지 않는다.

```
leftSide().toString()
i.getValue().isShort()
```

이 경우를 처리하려면 다음과 같이 하면 된다.

- 두 개 이상의 호출을 찾기. 또는
- 원시 타입으로 변환하거나 원시 타입에서 변환하기 위한 표준 메서드 호출을 포함하여 알려진 객체 변환 호출을 고려 대상에서 제거하기.

추가 변수를 사용하면 탐색 코드를 위장할 수 있으므로 코드를 읽어야 하는 경우가 종종 있다. 예를 들어 다음 Java 코드에는 호출 체인이 포함되어 있지 않다.

```
Token token;
token = parseTree.token();
if (token.identifier() != null) {
    ...
}
```

그러나 호출 체인을 포함하는 다음 코드와 동일하다.

```
if (parseTree.token().identifier() != null) {
    ...
}
```

Smalltalk. *Smalltalk*에는 미리 정의된 제어 구조체가 없고 제어 구조체를 구현할 때도 메시지를 사용하기 때문에 *Smalltalk* 코드에서 단순히 호출 시퀀스를 검색하는 것은 많은 노이즈를 생성할 수 있다. 위장된 탐색 코드가 있는 위의 예제는 *Smalltalk*에서 다음과 같이 읽힐 것이다. (*isNil*과 *ifFalse:[...]* 메시지에 주목하자).

```
| token |
token := parseTree token.
token identifier isNil ifFalse:[...]
```

탐색 코드와 동등한 버전이 된다.

```
parseTree token identifier isNil ifFalse: [...]
```

다음 코드 세그먼트에는 일련의 호출이 포함되어 있지만 첫 번째는 부울 테스트를 다루고 두 번째는 변환(사실 변환 남용)을 다루기 때문에 문제가 되지 않는다.

```
(a isNode) & (a isAbstract) ifTrue: [...]
aCol asSet asSortedCollection asOrderedCollection
```

Java. *Java* 또는 *C++*의 경우, 기본 데이터 타입과 제어 구조는 객체를 사용하여 구현되지 않으므로 간단한 패턴 일치는 노이즈가 적다. 예를 들어, 다음과 같은 간단한 Unix 명령은 노이즈가 적다.

```
egrep '.*\(\).*\(\).*\(\).*\.\.java
egrep '.*\..*\..*\..*\.\.java'
```

클래스 간 탐색 코드 결합도의 예인 다음과 같은 코드 줄을 식별하고 위에서 언급한 변환을 필터링한다.

```
a.getAbstraction().getIdentifier().traverse(this)
a.abstraction.identifier.traverse(this)
```

보다 정교한 매칭 표현식을 사용하면 대괄호 또는 다른 조합의 괄호로 인해 발생하는 노이즈를 줄일 수 있다.

AST 매칭! 트리 매칭을 표현하는 방법이 있는 경우 탐색 코드를 감지할 수 있다. 예를 들어, 리팩토링 브라우저 (Refactoring Browser)와 함께 제공되는 다시 쓰기 규칙 편집기 (Rewrite Rule Editor)는 다음과 같다. [RBJ97]는 '@object 'mess1 'mess2 'mess3 패턴을 사용하여 탐색 코드를 감지할 수 있다. 결과 분석 범위를 좁히려면 도메인 객체에 속하는 메시지만 고려하고 라이브러리 객체의 메서드 선택자(예: `isNil`, `not`, `class`, ...)를 모두 제거해야 한다.

단계

탐색 코드를 제거하는 방법은 재귀적으로 동작을 데이터 가까이 이동하기 를 사용하는 것이다. 그림 9.6 은 이 변환을 보여준다.

1. 이동할 탐색 코드 구별하기(*Identify*)
2. 동작을 데이터 가까이 이동하기 를 사용하여 탐색의 한 수준을 제거하기 를 적용하기(*Apply*). (이 시점에서 회귀 테스트가 실행되어야 한다.)
3. 필요한 경우 반복(*Repeat*)하기

주의. 리팩터링 프로세스는 클라이언트에서 프로바이더로 코드를 푸시하는 것에 의존한다는 점에 유의하는 것이 중요하다. 이 예에서는 `Car`에서 `Engine`으로, `Engine`에서 `Carburetor`로 푸시한다. 일반적인 실수는 클라이언트 클래스 수준에서 프로바이더 속성 값의 속성에 액세스하는 접근자를 정의하여 탐색 코드를 제거하려고 하는 것이다(예: `Car` 클래스에 접근자 `getCarburetor`를 정의하는 것). 클래스 간의 결합도가 줄어드는 대신 퍼블릭 접근자의 수가 증가하고 시스템이 더 복잡해진다.

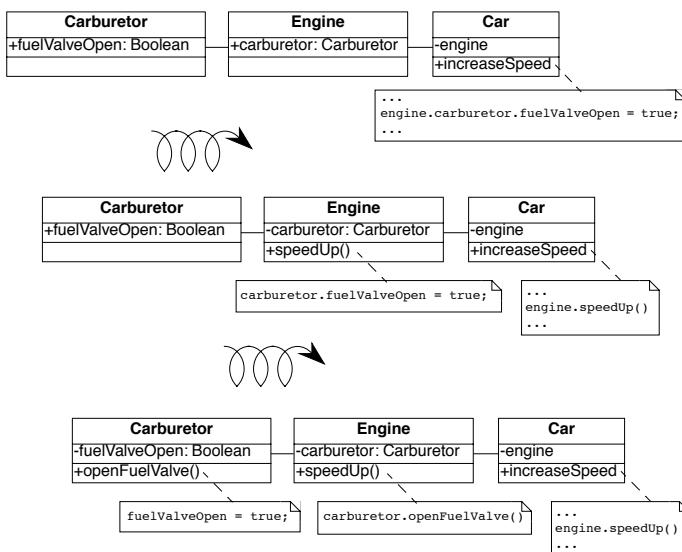


그림 9.6: 데이터 컨테이너 체인을 서비스 프로바이더로 변환하여 탐색 코드를 없애고 클래스 간 결합도를 줄일 수 있다.

트레이드오프

장점

- 클래스 간의 종속성 사슬이 제거되어 최하위 레벨의 클래스 변경이 더 적은 수의 클라이언트에 영향을 미친다.
- 시스템에 암시되어 있던 기능이 이제 새 클라이언트에서 이름을 지정하고 명시적으로 사용할 수 있다.

단점

- 탐색 코드 제거하기를 체계적으로 적용하면 인터페이스가 커질 수 있다. 특히 클래스가 컬렉션인 인스턴스 변수를 많이 정의하는 경우 탐색 코드 제거하기를 사용하면 기본 컬렉션을 보호하기 위해 많은 수의 추가 메서드를 정의해야 할 수 있다.

어려움

- 탐색 코드 제거하기를 언제 적용할지 결정하는 것은 어려울 수 있다. 단순히 클래스 공동 작업자에게 요청을 위임하는 메서드를 정의하는 것이 항상 해결책이 될 수는 없다. 내부 정보를 제공하면 클래스의 인터페이스가 축소될 수 있다. 예를 들어 클래스가 잘 정의된 몇 가지 동작을 구현하지만 다른 공동 작업자에게 파사드 [p. 350] 역할을 하는 경우, 클래스의 인터페이스를 줄이기 위해 공동 작업자에게 직접 액세스 권한을 부여하는 것이 더 간단할 수 있다.

레거시 솔루션이 최종 솔루션인 경우

객체가 그래픽으로 표시되거나 데이터베이스에 매핑되는 경우 탐색 코드가 최상의 솔루션일 수 있다. 이러한 경우 목표는 클래스 간의 구조적 관계를 실제로 노출하고 모방하는 것이다. 내비게이션 코드를 제거하는 것은 쓸데없는 일이 될 것이다.

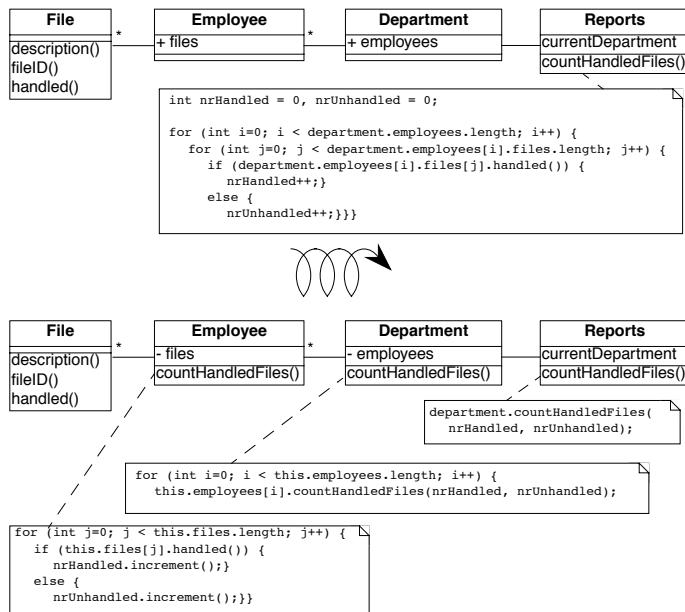


그림 9.7: Reports 클래스와 File 및 Employee 클래스 간의 불필요한 종속성을 제거하는 방법.

클라이언트가 간접 프로바이더와 대화해야 하는 경우도 있다. 직접 프로바이더가 특정 속성(OOID, 키...)이 주어진 특정 객체를 반환하는 객체 서버 역할을 하는 경우가 이에 해당한다. 이 경우 클라이언트는 클라이언트가 메시지를 보내는 객체(간접 프로바이더)를 반환하는 객체 서버(직접 프로바이더)를 호출한다.

예시

Employee, Payroll, TelephoneGuide 클래스를 수정한 후 전체 프로젝트를 다시 빌드하는 데 30분이 걸렸다는 것을 알 수 있다. 다음에 크리스(유지관리자 중 한 명)를 만나면 왜 이렇게 오래 걸렸냐고 물어보자. “아마 Employee 클래스를 변경하셨을 거에요.” 그는 “많은 클래스가 그 클래스에 의존하기 때문에 감히 그 클래스를 건드리지 못했어요.”라고 대답한다.

이 Employee 클래스를 더 자세히 살펴보고 불필요한 종속성을 많이 찾기로 결정한다. 예를 들어 (그림 9.7에 표시된 것처럼) 모든 employees이 처리하는 파일 수를 각 Department에 대해 계산하는 countHandledFiles 메서드 하나를 구현하는 Reports 클래스가 있다. 안타깝게도 Department와 File 사이에는 직접적인 관계가 없으므로 ReportHandledFiles는 모든 files을 열거하고 handled() 상태에 액세스하기 위해 부서의 employees를 탐색해야 한다.

아래의 Java 코드는 탐색 코드 제거하기를 적용하기 전과 후의 상황을 보여준다. 굵은 텍스트 요소는 적용 전후 상황의 문제와 해결 방법을 강조한다.

예전

```
public class Reports {
    ...
    public static void countHandledFiles(Department department) {
        int nrHandled = 0, nrUnhandled = 0;

        for (int i=0; i < department.employees.length; i++) {
            for (int j=0; j < department.employees[i].files.length; j++) {
                if (department.employees[i].files[j].handled()) {
                    nrHandled++;
                } else {
                    nrUnhandled++;
                }
            }
        }
    ...
}
```

countHandledFiles 메서드는 현재 부서의 employees과 각 files을 요청하여 처리된 파일 수를 계산한다. IctDepartment 및 Employee 클래스는 이러한 속성을 공개로 선언해야 한다. 이 구현에서는 두 가지 문제가 발생한다.

- Reports 클래스는 Department, Employee 및 File 간의 연결을 열거하는 방법을 알고 있어야 하며, 이 정보는 각 클래스의 퍼블릭 인터페이스에서 액세스할 수 있어야 한다. 이러한 퍼블릭 인터페이스 중 하나가 변경되면 이 변경 사항은 관련된 모든 클래스에 영향을 미친다.
- countHandledFiles 메서드는 employees 및 files 변수에 직접 액세스하여 구현된다. 이렇게 하면 Reports 클래스와 Department 및 Employee 클래스가 불필요하게 결합된다. IctDepartment 또는 Employee 클래스가 연결된 객체를 골드화하는 데 사용되는 데이터 구조를 변경하면 Re-

ports 클래스의 모든 메서드를 조정해야 한다.

단계

해결책은 중첩된 for 루프를 별도의 메서드로 추출하여 적절한 클래스로 옮기는 것이다. 이것은 실제로 두 단계의 과정이다.

먼저 Reports.countHandledFiles에서 외부 for 루프를 별도의 메서드로 추출하고(이름도 countHandledFiles로 지정), 이를 Department 클래스로 옮긴다.

```
public class Department {
    ...
    public void countHandledFiles
        (Counter nrHandled, Counter nrUnhandled) {
        for (int i=0; i < this.employees.length; i++) {
            for (int j=0; j < this.employees[i].files.length; j++) {
                if (this.employees[i].files[j].handled()) {
                    nrHandled.increment();
                } else {
                    nrUnhandled.increment();
                }
            }
        }
    }

    public class Reports {
        ...
        private static void countHandledFiles(Department department) {
            Counter nrHandled = new Counter (0), nrUnhandled = new Counter
            (0);
            department.countHandledFiles(nrHandled, nrUnhandled);
        }
    }
}
```

다음으로, Department.countHandledFiles(countHandledFiles라고도 함)에서 내부 for 루프를 추출하여 Employee 클래스로 이동한다.

```
public class Employee {
    ...
    public void countHandledFiles
        (Counter nrHandled, Counter nrUnhandled) {
        for (int j=0; j < this.files.length; j++) {
            if (this.files[j].handled()) {
                nrHandled.increment();
            } else {
                nrUnhandled.increment();
            }
        }
    }
}
```

```
public class Department {
    ...
    public void countHandledFiles
        (Counter nrHandled, Counter nrUnhandled) {
        for (int i=0; i < this.employees.length; i++) {
            this.employees[i].countHandledFiles(nrHandled, nrUnhandled);}
    ...
}
```

`employees` 및 `files` 변수에 대한 모든 직접 액세스를 제거하면 이러한 속성을 프라이빗(private)으로 선언할 수 있다.

예시

객체 “O”의 메서드 “M”은 다음 종류의 객체의 메서드만 호출해야 한다.

1. 자체
2. 매개변수
3. 이 메서드가 생성하는 모든 객체
4. 이 메서드의 직접 컴포너트 객체

— 디미터 법칙

디미터의 법칙[LHR88]을 위반하는 탐색 코드는 잘못된 동작의 잘 알려진 증상이다 [LK94] [Sha97] [Rie96]. 클래스 간에 불필요한 종속성이 발생하고 결과적으로 클래스의 표현을 변경하려면 모든 클라이언트를 조정해야 한다.

관련 패턴

탐색 코드 제거하기 와 기계적으로 코드 비교하기 [p. 249] 는 서로를 강화한다. 여러 클라이언트에 분산되어 있는 탐색 코드는 시스템 전체에 중복된 코드를 분산시킵니다. 기계적으로 코드 비교는 이 현상을 감지하는 데 도움이 된다. 탐색 코드 제거하기 는 중복된 코드를 한곳으로 모아 리팩터링하고 제거하기 쉽게 해준다.

9.3 신 클래스 분할하기

더 블롭 (The Blob) [BMMM98], 신 클래스 (God Class) [Rie96]로도 알려져 있다.

의도 책임이 너무 많은 클래스를 여러 개의 작고 응집도 높은 클래스로 분할 한다.

문제

너무 많은 책임을 맡은 클래스를 어떻게 유지 관리할 수 있을까?

이 문제는 다음과 같은 이유로 어렵다.

- 너무 많은 책임을 맡음으로써 신 클래스는 애플리케이션의 제어권을 독점하게 된다. 거의 모든 변경 사항이 이 클래스에 영향을 미치고 여러 책임에 영향을 미치기 때문에 애플리케이션의 발전이 어렵다.
- 신 클래스에 섞여 있는 여러 추상화를 이해하기 어렵다. 여러 추상화의 데이터 대부분은 서로 다른 위치에서 액세스한다.
- 시스템의 다른 기능이나 다른 객체에 영향을 주지 않으면서 기능을 변경할 위치를 파악하는 것은 어렵다. 또한 다른 객체의 변경은 신 클래스에 영향을 미쳐 시스템의 진화를 방해할 가능성이 높다.
- 신 클래스의 동작 일부를 블랙박스 방식으로 변경하는 것은 거의 불가능하다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 문제를 한 번에 해결할 필요는 없습니다.
- 시맨틱 래퍼 (Semantic Wrapper)를 사용하여 래핑하고 인터페이스를 표시할 수 있다.

해결

신 클래스의 책임을 협업하는 클래스 또는 신 클래스를 끌어낸 새 클래스에 점진적으로 재분배한다. 신 클래스에 파사드만 남으면 파사드를 제거하거나 지원 중단한다.

탐지

신 클래스는 다양한 방식으로 인식할 수 있다.

- 하나의 거대한 클래스는 다른 많은 클래스를 데이터 구조로 취급한다.
- “루트” 클래스 또는 다른 거대한 클래스는 “System”, “Subsystem”, “Manager”, “Driver” 또는 “Controller”와 같은 단어가 포함된 이름을 가진다.
- 시스템에 대한 변경이 항상 동일한 클래스에 대한 변경을 초래한다.
- 클래스에 대한 변경은 클래스의 어느 부분에 영향을 미치는지 식별할 수 없기 때문에 매우 어렵다.
- 클래스를 재사용하는 것은 너무 많은 디자인 문제를 다루기 때문에 거의 불가능하다.
- 해당 클래스가 시스템 또는 하위 시스템의 속성 및 메서드 대부분을 보유한 도메인 클래스이다. (일부 UI 프레임워크는 메서드가 많은 큰 클래스를 생성하고 일부 데이터베이스 인터페이스 클래스에는 많은 속성이 필요할 수 있으므로 이 임계값이 절대적인 것은 아니다.)
- 클래스에 분리된 인스턴스 변수에 대해 작동하는 관련 없는 메서드 집합이 있다. 클래스의 응집도가 일반적으로 낮다.
- 클래스가 작은 수정에도 컴파일 시간이 오래 걸린다.
- 클래스가 많은 책임을 맡고 있어 테스트하기 어렵다.
- 클래스가 많은 메모리를 사용한다.
- 사람들이 말하길 “이것이 시스템의 핵심이다”라고 말한다.
- 신 클래스의 책임을 물어보면 다양하고 길고 불명확한 대답을 듣는다.

- 신 클래스는 유지 관리자의 악몽이므로 어떤 클래스가 크고 유지 관리가 어려운지 물어자. 담당하고 싶지 않은 클래스가 무엇인지 물어보자. (변형: 사람들에게 작업하고 싶은 클래스를 선택하게 하자. 모두가 피하는 클래스는 신 클래스가 될 수 있다.)

단계

이 솔루션은 신 클래스로부터 점진적으로 멀어지는 동작에 의존한다. 이 과정에서 데이터 컨테이너는 신 클래스가 데이터에 대해 수행하던 기능을 획득함으로써 더욱 객체와 유사해진다. 일부 새로운 클래스도 신 클래스에서 추출된다.

다음 단계에서는 이 프로세스가 이상적으로 작동하는 방식을 설명한다. 그러나 신 클래스는 내부 구조가 매우 다양할 수 있으므로 변환 단계를 구현하는데 다른 기술을 사용할 수 있다. 또한 신 클래스는 한 번에 치료할 수 없으므로 안전한 진행 방법은 먼저 신 클래스를 가벼운 신 클래스로 변환한 다음 지인에게 동작을 위임하는 파사드 [p. 350]로 변환하는 것이다. 마지막으로 클라이언트는 리팩터링된 데이터 컨테이너와 다른 새 객체로 리디렉션되고 파사드는 제거될 수 있다. 이 과정은 그림 39에 설명되어 있다.

다음 단계는 반복적으로 적용된다. 변경할 때마다 회귀 테스트하기 [p. 223] 을 적용해야 한다.

1. 신 클래스의 인스턴스 변수의 응집도 있는 하위 집합을 식별하고 이를 외부 데이터 컨테이너로 변환한다. 신 클래스의 초기화 메서드가 새 데이터 컨테이너의 인스턴스를 참조하도록 변경한다.
2. 신 클래스가 데이터 컨테이너로 사용하는 모든 클래스(1단계에서 생성한 클래스 포함)를 식별하고 동작을 데이터 가까이 이동하기를 적용하여 데이터 컨테이너를 서비스 프로바이더로 승격한다. 신 클래스의 원래 메서드는 단순히 이동된 메서드에 동작을 위임한다.
3. 1단계와 2단계를 반복적으로 적용하면 신 클래스에는 큰 초기화 메서드가 있는 파사드 외에는 아무것도 남지 않는다. 초기화에 대한 책임을 별도의 클래스로 이동하여 순수한 파사드만 남도록 한다. 클라이언트를 이전 신 클래스가 이제 파사드인 객체로 반복적으로 리디렉션하고 파사드를

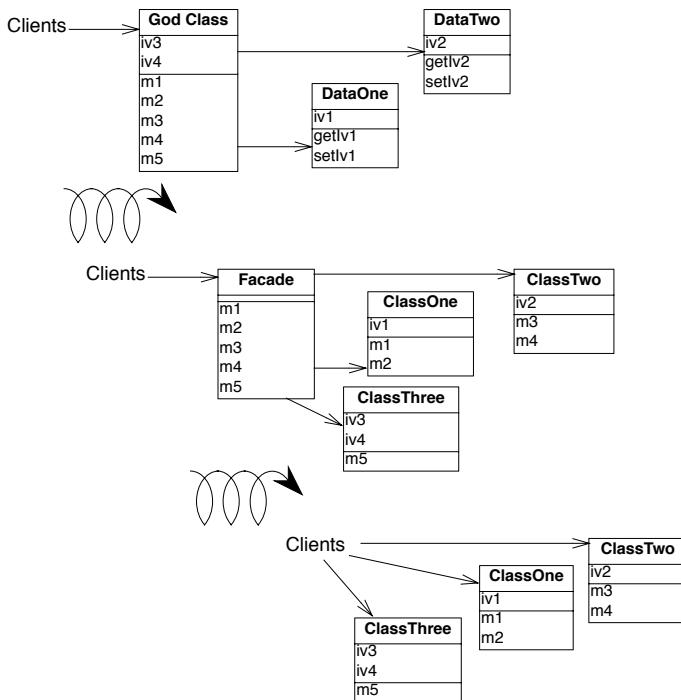


그림 9.8: 신 클래스는 두 단계로 리팩터링되는데, 먼저 데이터 컨테이너에 책임을 재분배하거나 새 클래스를 스포ن하여 파사드만 남을 때까지 리팩터링하고 두 번째 단계는 파사드를 제거한다.

지원 중단하거나(폐기된 인터페이스 지원 중단하기 [p. 237] 참조) 단순히 제거다.

트레이드오프

장점

- 애플리케이션 제어는 더 이상 단일 모놀리식 엔티티에 중앙 집중화되지 않고 각각 잘 정의된 책임 세트를 맡은 엔티티에 분산되어 있다. 디자인은 절차 지향 디자인에서 자율적으로 상호 작용하는 객체를 기반으로 하는 객체 지향 디자인으로 진화한다.
- 원래 신 클래스의 일부가 더 쉽게 이해되고 유지보수성이 향상되었다.
- 원래 신 클래스의 일부는 이슈가 적기 때문에 더 안정적이다.
- 시스템 종속성이 단순화되어 전체 컴파일 시간이 단축될 수 있다.

단점

- 신 클래스를 분할하는 것은 길고 느리고 지루한 과정이다.
- 유지 관리자는 더 이상 하나의 신 클래스로 이동하여 수정할 동작을 찾을 수 없다.
- 클래스 수가 늘어날 것이다.

어려움

- 신 클래스 메서드는 그 자체로 너무 많은 책임이 있는 큰 프로시저 추상화일 수 있다. 이러한 메서드는 인스턴스 변수와 메서드의 응집도 높은 집합을 클래스로 분리하기 전에 분해해야 할 수 있다.

레거시 솔루션이 최종 솔루션인 경우

무엇의 리스크가 더 높을까? 신 클래스를 신 클래스 분할하기로 만들것인가, 아니면 그대로 둘것인가? 진짜 신 클래스는 크고 다루기 힘든 짐승이다. 이를 더 강력한 추상화로 분할하면 상당한 비용이 발생할 수 있다.

핵심 문제는 신 클래스를 유지해야 하는지 여부이다. 신 클래스를 확장하거나 수정할 필요가 거의 없는 안정적인 레거시 코드로 구성되어 있다면 리팩터링하는 데 많은 노력을 투자할 필요가 없다.

반면에 불안정하고 변화하는 요구사항에 자주 적응해야 하는 것이 신 클래스의 여러 클라이언트라고 가정해 보자. 이 경우 클라이언트에게 깔끔한 인터페이스가 제공되지 않으므로 신 클래스로부터 보호되어야 한다. 대신 올바른 인터페이스 제시하기 [p. 229]를 적용하면 클라이언트와 신 클래스 사이에 깔끔한 객체 지향 추상화 계층을 도입할 수 있으며 클라이언트를 더 쉽게 진화시킬 수 있

예시

시스템에 신 클래스 혹은 신 개체를 만들지 말자.

— 아서 리엘, 휴리스틱 3.2 [Rie96]

신 클래스는 낮은 수준의 절차 지향 추상화만 달성하기 때문에 진화를 방해 하므로 변경 사항이 신 클래스, 데이터 컨테이너 및 클라이언트의 많은 부분에 영향을 미칠 수 있다. 신 클래스를 객체 지향 추상화로 분할하면 변경 사항이 더 지역화되는 경향이 있으므로 구현하기가 더 쉬워진다.

관련 패턴

푸트과 요더의 “큰 진흙 뭉치 (Big Ball of Mud)” [FY00]는 소프트웨어 개발에서 신 클래스가 자연적으로 발생한다는 점에 주목한다.

“사람들이 일하기 때문에 큰 진흙 뭉치가 만들어진다. 많은 영역에서 작동하는 것으로 입증된 유일한 방법이다. 실제로 더 높은 수준의 접근 방식이 아직 경쟁력이 있음을 입증하지 못한 분야에서 효과가 있다.

큰 진흙 뭉치를 비난하는 것은 우리의 목적이 아니다. 시스템이 진화하는 초기 단계에서는 캐주얼 아키텍처가 자연스러운 현상이다. 그러나 독자들은 우리가 더 나은 것을 열망할 수 있기를 희망한다는 점을 의심하지 않을 수 없을 것이다. 아키텍처의 불쾌감을

유발하는 포스(force)와 압력, 그리고 언제 어떻게 직면할 수 있는지를 인식함으로써 아키텍트들이 앞으로 수년간 지배적인 위치에 설 수 있는 진정으로 내구성 있는 인공물이 출현할 수 있는 발판을 마련하고자 한다. 핵심은 시스템과 프로그래머, 나아가 전체 조직이 시스템이 성장하고 성숙함에 따라 해당 도메인과 그 안에서 다가오는 아키텍처 기회에 대해 학습하도록 하는 것이다.”

— 푸트와 요더 [FY00]

올바른 인터페이스 제시하기 [p. 229]는 신 클래스 자체를 수정하거나 확장 할 필요가 거의 없을 때 적용해야 하는 경쟁 패턴이다.

제 10 장

다형성 적용한 조건문 변환

중복 코드, 데이터 컨테이너, 신 클래스에 이어 객체 지향 소프트웨어에서 가장 눈에 띄는 잘못된 책임의 징후 중 하나는 일부 인자의 타입을 테스트하는 거의 전적으로 Case 문으로 구성된 거대한 메서드가 생긴다는 것이다.

Case 문이 본질적으로 나쁜 것은 아니지만, 객체 지향 코드에서는 테스트를 수행하는 객체가 테스트 대상 객체에 더 잘 분산되어야 할 책임을 떠맡고 있다는 신호인 경우가 많다. 큰 조건문은 중복된 코드와 마찬가지로 시간이 지남에 따라 자연스럽게 발생한다. 소프트웨어가 새로운 사례를 처리하도록 조정됨에 따라 이러한 사례는 코드에서 조건문으로 나타난다. 이러한 큰 조건문의 문제는 장기적으로 코드를 훨씬 더 취약하게 만들 수 있다는 것이다.

포스

다음과 같은 주요한 요구 사항인 포스가 작용하고 있다.

- 시간이 지남에 따라 요구 사항이 변경되면 소프트웨어 시스템의 클래스는 새롭고 특수한 경우를 처리하도록 조정되어야 한다.
- 시스템에 새 클래스나 하위 클래스를 추가하면 네임스페이스가 복잡해 진다.
- 새로운 요구 사항을 처리하기 위해 작동 중인 소프트웨어를 조정하는 가

장 빠른 방법은 코드의 특정 지점에서 특수한 경우에 대한 조건부 테스트를 추가하는 것입니다.

- 시간이 지남에 따라 단순한 디자인은 특수한 경우에 대한 많은 조건문으로 이루어진 테스트로 인해 복잡해지는 경향이 있다.
- Case 문은 여러 클래스에 걸쳐 다양한 경우를 분산하는 대신 모든 변형을 한곳으로 그룹화한다. 그러나 Case 문이 두 곳 이상에 나타나면 유연성이 떨어지는 디자인으로 이어진다.
- 일부 프로그래밍 언어에서는 다형성보다 다양한 동작을 구현하는 데 Case 문이 더 일반적으로 많이 사용된다.

큰 조건문은 클라이언트가 구현하는 동작을 프로바이더 클래스로 옮겨야 한다는 신호인 경우가 많다. 일반적으로 새로운 메서드가 프로바이더 계층 구조에 도입되고 조건문의 개별 케이스가 각각 프로바이더 클래스 중 하나로 이동다.

증상은 쉽게 알아볼 수 있지만 기술적 세부 사항과 선호하는 솔루션은 상당히 다를 수 있다. 특히 프로바이더 계층 구조가 이미 존재하고 조건문이 프로바이더 인스턴스의 클래스를 명시적으로 확인하는 경우 리팩터링은 비교적 간단하다. 그러나 프로바이더 계층 구조가 존재하지 않는 경우가 많고 조건은 암시적으로 타입 정보만 모델링하는 어트리뷰트를 테스트한다. 또한 조건은 외부 클라이언트뿐만 아니라 프로바이더 계층 자체에서도 발생할 수 있다.

개요

다형성 적용 조건문 변환은 이러한 큰 조건문을 제거하기 위해 책임을 재분배하는 방법을 설명하는 패턴 언어로, 클래스 간의 결합도를 줄이고 향후 변경에 대비한 유연성을 향상시킨다.

이 패턴 언어는 다형성을 시뮬레이션하기 위해 조건문을 사용할 때 발생하는 가장 일반적인 문제를 해결하는 6가지 패턴으로 구성되어 있다. 자신 타입 검사 변환하기와 클라이언트 타입 검사 변환하기는 명시적 타입 검사를 수행할 때 발생하는 가장 일반적인 경우를 다룬다. 조건문을 등록으로 변환하기은 덜 자주 발생한다. 또한 상태 추출하기, 전략 추출하기 및 널 객체 도입하기도 포함되어 있다. 세 가지 기존 디자인 패턴(상태 [p. 351])을 복사하기 위해서가

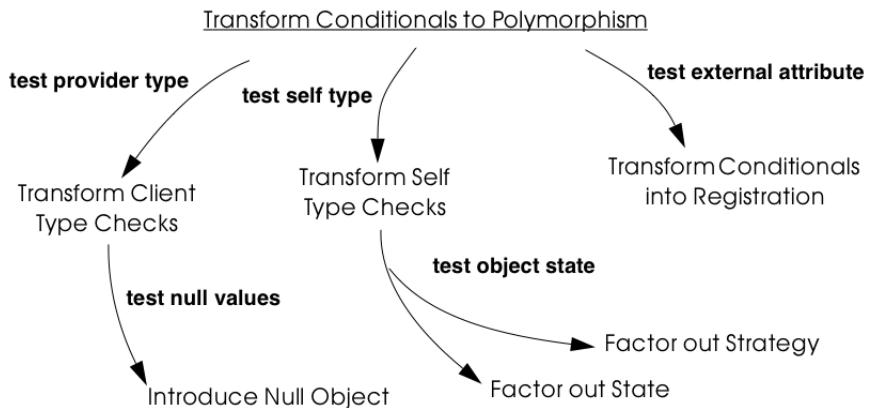


그림 10.1: 조건부 변환을 다형성으로 구성하는 패턴 간의 관계

아니다, 전략 [p. 352] 및 널 객체 [p. 350]를 복사하기 위한 것이 아니라 리엔지니어링 컨텍스트에서 이러한 디자인 패턴을 적용하여 타입 검사 조건문을 제거하는 방법을 보여주기 위한 것이다.

그림 10.1 summarizes the relationships and the differences between the patterns.

- 자신 타입 체크 변환하기는 각 타입 케이스에 대해 서브클래스 도입을 통해 프로바이더 클래스에서 타입 정보에 대한 조건부를 제거한다. 조건부 코드는 새 하위 클래스 중 하나의 인스턴스에 대한 단일 다형성 메서드 호출로 대체된다.
- 클라이언트 타입 검사 변환하기는 클라이언트 클래스의 타입 정보에 대한 조건부를 각 프로바이더 클래스에 새 메서드 도입으로 변환합니다. 조건문은 새 메서드에 대한 단일 다형성 호출로 대체된다.
- 상태 추출하기는 테스트 중인 타입 정보가 동적으로 변경될 수 있는 자신 타입 체크 변환하기의 특수한 경우를 처리한다. 변하는 상태를 모델링하기 위해 프로바이더 클래스에 상태 객체가 도입된다. 그리고, 조건은 새 상태(State) 객체의 메서드 호출로 대체된다.
- 전략 추출하기는 자신 타입 체크 변환하기의 또 다른 특수한 경우이다. 의 특수한 경우로, 다양한 프로바이더 사례를 처리하는 알고리즘이 새로

운 전략 객체 도입에 의해 추출된다. 상태 추출하기 와의 주요 차이점은 상태가 아닌 알고리즘이 동적으로 달라질 수 있다는 점이다.

- 널 객체 도입하기 는 클라이언트 탑 검사 변환하기 의 특수한 경우를 다룹니다. 이는 수행되는 테스트에서 프로바이더가 정의되어 있는지 여부를 확인한다. 조건은 적절한 기본 동작을 구현하는 널 객체를 도입하여 제거한다.
- 조건문을 등록으로 변환하기 은 처리할 객체의 일부 속성에 따라 조건문이 외부 도구를 시작하는 상황을 다룬다. 해결책은 도구가 플러그인으로 등록되는조회 서비스를 도입하는 것이다. 그런 다음 조건은 등록된 플러그인에 대한 간단한 조회로 대체된다. 그러면 도구 사용자를 변경하지 않고도 새 플러그인을 추가하거나 제거할 수 있으므로 솔루션은 완전히 동적이다.

10.1 자신 타입 체크 변환하기

의도 서브클래스를 구현한 후크 메서드 호출로 복잡한 조건문을 대체하여 클래스의 확장성을 개선한다.

문제

클래스는 객체의 현재 '타입'을 나타내는 일부 속성을 테스트하는 복잡한 조건문에 여러 가지 가능한 동작을 묶어두기 때문에 수정하거나 확장하기가 어렵다.

이 문제는 다음과 같은 이유로 어렵다.

- 개념적으로 간단한 확장이지만 조건부 코드에는 많은 변경이 필요하다.
- 조건부 코드가 포함된 메서드를 복제하고 수정하지 않으면 서브클래싱이 거의 불가능하다.
- 새 동작을 추가하면 항상 동일한 메서드 집합이 변경되고 항상 조건부 코드에 새 Case 문이 추가된다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 자체 타입 검사는 다형성을 시뮬레이션한다. 조건부 코드는 대신 어떤 서브클래스를 가져야 하는지 알려준다.

해결

복잡한 조건 분기가 있는 메서드를 식별한다. 각각의 경우 조건부 코드를 새로운 혹은 메서드 호출로 대체하자. 조건의 경우에 해당하는 서브클래스를 식별하거나 도입한다. 이러한 각 서브 클래스에서 원래 Case 문에서 해당 케이스에 해당하는 코드를 사용하여 후크 메서드를 구현한다.

탐지

대부분의 경우 코드를 작업하는 동안 타입 구분이 눈에 띠기 때문에 굳이 어디에서 검사가 이루어지는지 감지할 필요가 없다. 그러나 시스템의 알려지지 않은

부분에 유사한 관행이 있는지 신속하게 평가할 수 있는 간단한 기술을 갖는 것은 흥미로울 수 있다. 이는 시스템 상태를 평가하는 데 유용한 정보 소스가 될 수 있다.

- 타입 정보를 모델링하는 객체의 일부 불변 속성에 대한 복잡한 결정 구조를 가진 긴 메서드를 찾는다. 특히 생성자에서 설정되고 변경되지 않는 속성을 찾아보자.
- 유형 정보를 모델링하는 데 사용되는 속성은 일반적으로 일부 열거형 또는 일부 유한한 상수 값 집합에서 값을 가져온다. 이름이 일반적으로 클래스에 연결될 것으로 예상되는 엔티티 또는 개념을 나타내는 상수 정의 (예: RetiredEmployee 또는 PendingOrder)를 찾아보자. 조건문은 일반적으로 고정 속성의 값을 이러한 상수 값 중 하나와 비교하기만 한다.
- 특히 여러 메서드가 동일한 속성을 캐싱 클래스를 찾아보세요. 이는 속성이 타입을 시뮬레이션하는 데 사용되고 있다는 또 다른 일반적인 신호이다.
- Case 문이 포함된 메서드는 길이가 긴 경향이 있으므로 코드 줄별로 메서드를 정렬하거나 크기에 따라 클래스 및 메서드를 시각화하는 도구를 사용하는 것이 도움이 될 수 있다. 또는 조건문이 많은 클래스나 메서드를 검색할 수도 있다.
- 클래스 구현을 별도의 파일에 저장하는 것이 일반적인 C++ 또는 Java 와 같은 언어의 경우 조건 키워드(if, else, case 등)의 발생 빈도를 검색하고 세는 것이 간단하다. 예를 들어 UNIX 시스템에서는 다음과 같이 할 수 있다.

```
grep 'switch' `find . -name "*.cxx" -print`
```

이 명령은 디렉터리 트리에서 확장자가 .cxx인 파일 중 switch가 포함된 모든 파일을 열거한다. agrep와 같은 다른 텍스트 처리 도구는 더 세분화된 쿼리를 생성할 수 있는 가능성을 제공한다. indPerl과 같은 텍스트 처리 언어는 일부 종류의 쿼리, 특히 여러 줄에 걸쳐 있는 쿼리를 평가하는 데 더 적합할 수 있다.

- C/C++: 레거시 C 코드는 union 타입을 사용하여 클래스를 시뮬레이션할 수 있다. 일반적으로 union 타입은 실제 타입을 인코딩하는 하나의 데이

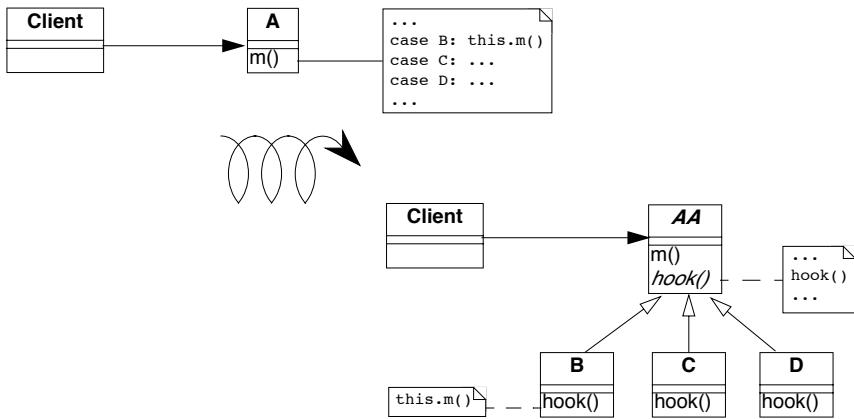


그림 10.2: 명시적 타입 검사를 자체 다형성 메서드 호출로 변환

터 멤버를 갖는다. 이러한 데이터 멤버를 전환하는 조건문을 찾아 union 을 캐스팅할 타입과 사용할 동작을 결정하자.

C++에서는 void 포인터로 선언된 데이터 멤버가 있는 클래스를 찾는 것이 매우 일반적이다. 다른 데이터 멤버의 값에 따라 이러한 포인터를 주어진 타입으로 형변환하는 조건문을 찾아보자. 타입 정보는 열거형 또는 (더 일반적으로) 상수 정수 값으로 인코딩할 수 있다.

- *Ada*: Ada 83은 다형성(또는 하위 프로그램 액세스 타입)을 지원하지 않았기 때문에, 다형성을 시뮬레이션하기 위해 구별된 레코드 타입(discretiated record type)을 사용하는 경우가 많다. 일반적으로 열거 타입은 여러 변형을 제공하며, 다형성으로의 변환은 Ada95에서는 간단하다.
- *Smalltalk*: Smalltalk는 타입을 조작하는 몇 가지 방법만 제공한다. 명시적 타입 검사를 나타내는 `isMemberOf`:와 `isKindOf`: 메서드의 응용 프로그램을 찾아보자. 타입 검사는 `self class = anotherClass`와 같은 테스트 또는 계층 구조 전체에서 `isSymbol`, `isString`, `isSequenceable`, `isInteger` 와 같은 메서드를 사용하여 속성 테스트를 통해 수행될 수도 있다.

단계

- 변환할 클래스와 그 클래스가 구현하는 다양한 개념적 클래스를 식별한다. 열거 타입이나 상수 집합이 이를 잘 문서화할 수 있다.
- 구현되는 각 동작에 대해 새 하위 클래스를 도입한다(그림 10.2 참조). 클라이언트가 원래 클래스가 아닌 새 서브클래스를 인스턴스화하도록 수정한다. 테스트를 실행한다.
- 조건문을 통해 다양한 동작을 구현하는 원래 클래스의 모든 메서드를 식별한다. 조건문이 다른 문으로 둘러싸여 있는 경우 별도의 보호된 후크 메서드로 이동시킨다. 각 조건문이 자체 메서드를 차지하면 테스트를 실행한다.
- 조건문의 Case 문을 해당 하위 클래스로 반복적으로 이동하여 주기적으로 테스트를 실행한다.
- 조건부 코드가 포함된 메서드는 이제 모두 비어 있어야 한다. 이를 추상 메서드로 대체하고 테스트를 실행하자.
- 또는 적절한 기본 동작이 있는 경우 새 계층 구조의 루트에서 이를 구현하자.
- 인스턴스화할 서브클래스를 결정하는 데 필요한 로직이 사소하지 않은 경우 이 로직을 새 계층 구조 루트의 팩토리 메서드로 캡슐화하는 것을 고려하자. 새 팩토리 메서드를 사용하도록 클라이언트를 업데이트하고 테스트를 실행한다.

트레이드오프

장점

- 이제 모든 동작을 포함하는 단일 클래스의 메서드 집합을 변경하지 않고도 새로운 동작을 충분 방식으로 추가할 수 있다. 이제 특정 동작을 다른 변형과 독립적으로 이해할 수 있다.
- 새 동작은 다른 동작과 독립적으로 데이터를 나타내므로 가능한 간섭을 최소화하고 분리된 동작의 이해도를 높일 수 있다.

- 이제 모든 동작이 공통 인터페이스를 공유하므로 가독성이 향상된다.

단점

- 이제 모든 동작이 여러 개의 관련 추상화로 분산되어 있으므로 동작에 대한 개요를 파악하기가 더 어려워질 수 있다. 그러나 개념이 서로 연관되어 있고 추상 클래스로 표현되는 인터페이스를 공유하므로 문제가 줄어든다.
- 클래스 수가 많으면 디자인이 더 복잡해지고 잠재적으로 이해하기 어려워진다. 원래 조건문이 단순하다면 이 변환을 수행할 가치가 없을 수도 있다.
- 명시적 타입 검사가 항상 문제가 되는 것은 아니며 때로는 용인할 수 있다. 새 클래스를 만들면 애플리케이션의 추상화 수가 증가하고 네임스페이스가 복잡해질 수 있다. 따라서 다음과 같은 경우에는 명시적 타입 검사를 새 클래스 생성의 대안으로 사용할 수 있다.
 - 메서드 선택이 고정되어 있고 향후에 진화하지 않을 집합이다. 그리고,
 - 몇 군데에서만 타입 검사가 수행된다.

어려움

- 필수 하위 클래스가 아직 존재하지 않기 때문에 여러 타입을 시뮬레이션 하는 데 조건문이 사용되는 경우를 구분하기 어려울 수 있다.
- 변환된 클래스의 인스턴스가 원래 생성되었다면 이제 다른 하위 클래스의 인스턴스를 생성해야 한다. 인스턴스화가 클라이언트 코드에서 발생한 경우 이제 해당 코드를 올바른 클래스를 인스턴스화하도록 조정해야 한다. 클라이언트에서 이러한 복잡성을 숨기려면 팩토리 객체 또는 메서드가 필요할 수 있다.
- 클라이언트의 소스 코드에 액세스할 수 없는 경우 생성자에 대한 호출을 변경할 수 없으므로 이 패턴을 적용하기 어렵거나 불가능할 수 있다.

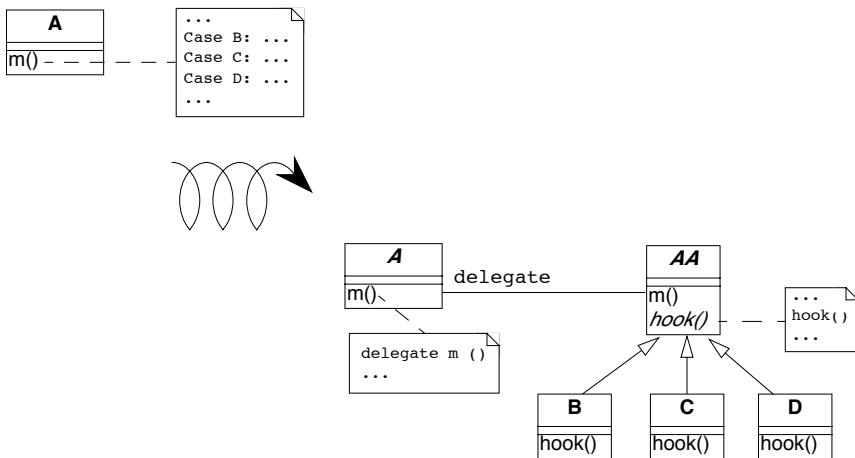


그림 10.3: 클래스를 하위 클래스화할 수 없는 경우 단순 텔리게이션과 자신 타입 체크 변환하기를 결합

- Case 문이 둘 이상의 속성을 테스트하는 경우 더 복잡한 계층 구조를 지원해야 할 수 있으며, 여러 상속이 필요할 수도 있다. 클래스를 각각 고유한 계층 구조를 가진 부분으로 분할하는 것을 고려해 보자.
- 원래 조건문을 포함하는 클래스를 서브클래스할 수 없는 경우, 자신 타입 체크 변환하기를 텔리게이션으로 구성할 수 있다. 원래 클래스의 상태와 동작의 일부를 메서드가 텔리게이션할 별도의 클래스로 이동하여 다른 계층 구조에서 다형성을 활용하는 것이 그림 10.3에서 설명하는 아이디어이다.

레거시 솔루션이 최종 솔루션인 경우

그럼에도 불구하고 명시적 타입 검사가 올바른 해결책이 될 수 있는 몇 가지 상황이 있다.

- 조건부 코드는 특수 도구에서 생성될 수 있다. 예를 들어 어휘 분석기와 구문 분석기는 우리가 피하고자 하는 종류의 조건문 코드를 포함하도록 자동으로 생성될 수 있다. 그러나 이러한 경우 생성된 클래스는 수동으로

확장하지 말고 수정된 사양에서 간단히 다시 생성해야 한다.

예시

우리는 메시지를 전송하여 대규모의 물리적 기계를 제어하는 복잡한 시스템을 작업했다. 이러한 메시지는 **Message** 클래스로 표현되며 다양한 타입이 있을 수 있다.

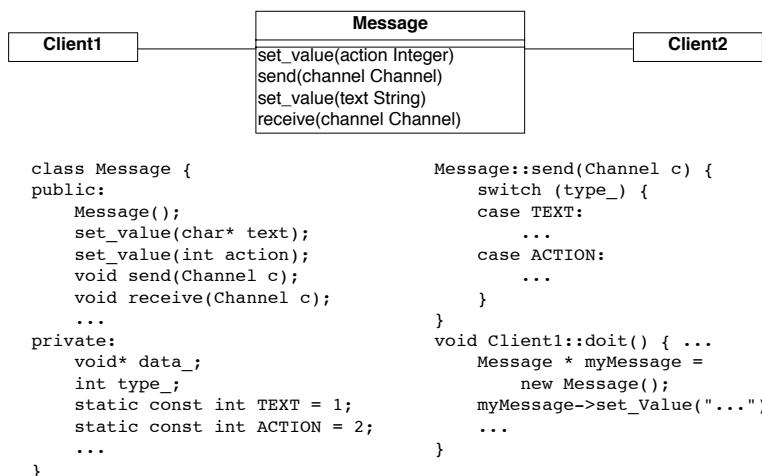


그림 10.4: 초기 디자인 및 소스 코드

이전

메시지 클래스는 코드와 그림에 표시된 것처럼 네트워크 연결을 통해 전송하기 위해 직렬화해야 하는 두 가지 종류의 메시지(TEXT 및 ACTION)를 래핑한다. 새로운 종류의 메시지(예: VOICE)를 보낼 수 있기를 원하지만 이를 위해서는 그림 10.4에 표시된 것처럼 **Message**의 여러 메서드를 변경해야 한다.

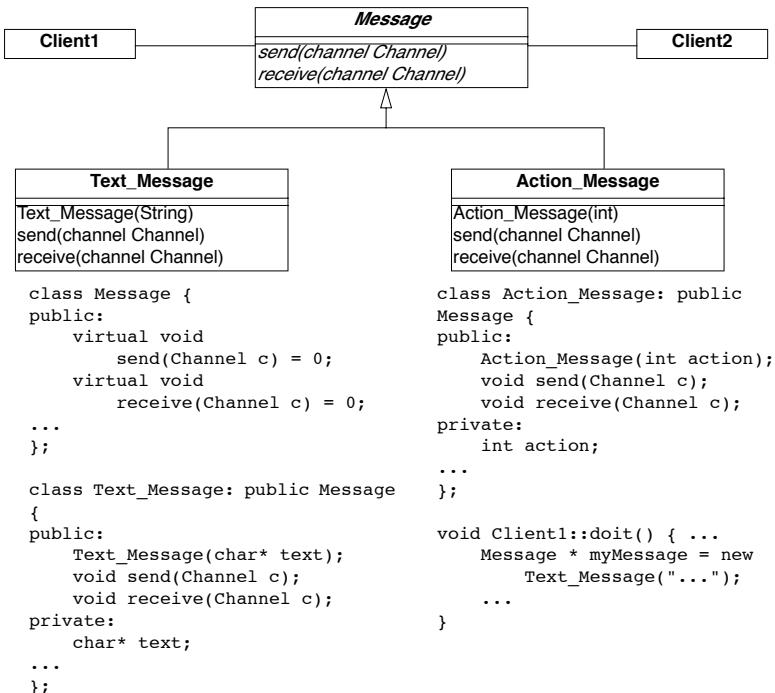


그림 10.5: 결과 계층 구조 및 소스 코드

이후

Message는 개념적으로 두 개의 다른 클래스인 Text_Message와 Action_Message를 구현하므로 그림 10.5에 표시된 것처럼 이들을 Message의 서브클래스로 도입된다. 새 클래스에 대한 생성자를 도입하고, 클라이언트가 Message가 아닌 Text_Message 및 Action_Message 인스턴스를 생성하도록 수정하고, `set_value()` 메서드를 제거한다. 이 시점에서 회귀 테스트가 실행되어야 한다.

이제 `type` 변수를 켜는 메서드를 찾는다. 각각의 경우, 스위치가 이미 전체 메서드를 차지하고 있지 않는 한, 전체 Case 문은 별도의 보호된 후크 메서드로 이동한다. `send()`의 경우 이미 그렇게 되어 있으므로 후크 메서드를 도입할 필요가 없다. 다시 말하지만, 모든 테스트는 여전히 실행되어야 한다.

이제 `Message`에서 그 서브클래스로 `Case` 문을 반복적으로 이동한다. `Message::send()`의 `TEXT` 케이스는 `Text_Message::send()`로 이동하고 `ACTION` 케이스는 `Action_Message::send()`로 이동한다. 이러한 케이스를 이동할 때마다 테스트는 계속 실행되어야 한다.

마지막으로, 원래의 `send()` 메서드는 이제 비어 있으므로 추상적으로 재선언할 수 있다(예: `virtual void send(Channel) = 0`). 다시 테스트가 실행되어야 한다.

근거

여러 데이터 타입으로 가장한 클래스는 디자인을 이해하고 확장하기 어렵게 만든다. 명시적 타입 검사를 사용하면 여러 가지 동작을 혼합하는 긴 메서드가 생성된다. 새로운 동작을 도입하려면 새로운 동작을 나타내는 새로운 클래스 하나를 지정하는 대신 모든 메서드를 변경해야 한다.

이러한 클래스를 여러 데이터 타입을 명시적으로 나타내는 계층 구조로 변환하면 단일 데이터 타입과 관련된 모든 코드를 한데 모아 응집도를 높이고, 일정량의 중복 코드(즉, 조건부 테스트)를 제거하며, 디자인을 더 투명하게 만들어 결과적으로 유지보수성을 높일 수 있다.

관련 패턴

자신 타입 체크 변환하기에서 변환할 조건은 클래스 자체의 속성으로 표현되는 타입 정보를 테스트한다.

조건문이 호스트 객체의 변경가능한(*mutable*) 상태를 테스트하는 경우, 대신 상태 추출하기 [p. 321] 또는 전략 추출하기 [p. 325]를 적용하는 것을 고려하자.

조건이 프로파이더 클래스 자체가 아닌 클라이언트에서 발생하는 경우 클라이언트 타입 검사 변환하기 [p. 311]를 적용하는 것을 고려하자.

조건문 코드가 세 번째 핸들러 객체 선택하기를 위해 두 번째 객체의 일부 타입 속성을 테스트하는 경우 대신 조건문을 등록으로 변환하기 [p. 333]을 적용하는 것을 고려하자.

10.2 클라이언트 타입 검사 변환하기

의도 프로바이더의 타입을 검사하는 조건문 코드를 새 프로바이더 메서드에 대한 다양성 호출로 변환하여 클라이언트/프로바이더 결합도를 줄인다.

문제

클라이언트가 명시적으로 프로바이더의 타입을 확인하고 프로바이더 코드를 작성할 책임이 있는 경우, 클라이언트와 서비스 프로바이더 간의 결합도를 어떻게 줄일 수 있는가?

이 문제는 다음과 같은 이유로 어렵다.

- 프로바이더 계층 구조에 새 하위 클래스를 추가하려면 특히 테스트가 많이 필요한 클라이언트 변경을 해야 한다.
- 클라이언트가 프로바이더의 책임이어야 하는 작업을 수행하기 때문에 클라이언트와 프로바이더는 결합도가 높은 경향이 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 조건문은 동작을 옮겨두어야 하는 클래스를 알려준다.

해결

프로바이더 계층 구조에 새 메서드를 도입한다. 클라이언트 조건의 해당 케이스를 해당 클래스로 이동하여 프로바이더 계층 구조의 각 하위 클래스에서 새 메서드를 구현한다. 클라이언트의 전체 조건문을 새 메서드에 대한 간단한 호출로 대체한다.

탐지

자신 타입 체크 변환하기에서 설명한 것과 기본적으로 동일한 기술을 적용하여 Case 문을 감지하되, 계층 구조를 구현하는 별도의 서비스 프로바이더의

타입을 테스트하는 조건을 찾는다. 또한 동일한 프로바이더 계층 구조의 다른 클라이언트에서 발생하는 Case 문도 찾아야 한다.

- *C++*: 레거시 C++ 코드는 런타임 타입 정보(RTTI)를 사용하지 않을 가능성이 높다. 대신, 타입 정보는 현재 클래스를 나타내는 열거형에서 값을 가져오는 데이터 멤버에 인코딩될 가능성이 높다. 이러한 데이터 멤버를 전환하는 클라이언트 코드를 찾아보자.
- *Ada*: 타입 테스트 감지는 두 가지 경우에 해당한다. 계층 구조가 단일 판별 레코드로 구현된 경우 판별자에 대한 Case 문장을 찾을 수 있다. 계층 구조가 태그가 지정된 타입으로 구현된 경우 타입에 대한 Case 문을 작성할 수 있으며(불연속적이지 않음), 대신 if-then-else 구조가 사용된다.
- *Smalltalk*: 자신 타입 체크 변환하기에서와 같이 `isMemberOf:` 및 `isKindOf:`의 적용과 `self class = anotherClass` 같은 테스트를 찾는다.
- *Java*: 특정 알려진 클래스에서 객체의 멤버십을 테스트하는 연산자 `instanceof`의 응용 프로그램을 찾아보자. Java의 클래스는 Smalltalk에서처럼 객체가 아니지만, 가상 머신에 로드되는 각 클래스는 `java.lang.Class`의 단일 인스턴스로 표현된다. 따라서 테스트를 수행하여 두 개의 객체 `x`와 `y`가 같은 클래스에 속하는지 확인할 수 있다.

```
x.getClass() == y.getClass()
```

또는 클래스 이름을 비교하여 클래스 멤버십을 테스트할 수도 있다.

```
x.getClass().getName().equals(y.getClass().getName())
```

단계

1. 명시적 타입 검사를 수행하는 클라이언트를 식별한다.
2. 조건문 코드에서 수행되는 작업을 나타내는 프로바이더 계층 구조의 루트에 빈 메서드를 새로 추가한다(그림 10.6 참조).
3. 조건문의 Case 문을 일부 프로바이더 클래스로 반복적으로 이동하여 해당 메서드 호출로 대체한다. 각 이동 후에는 회귀 테스트를 실행해야 한다.

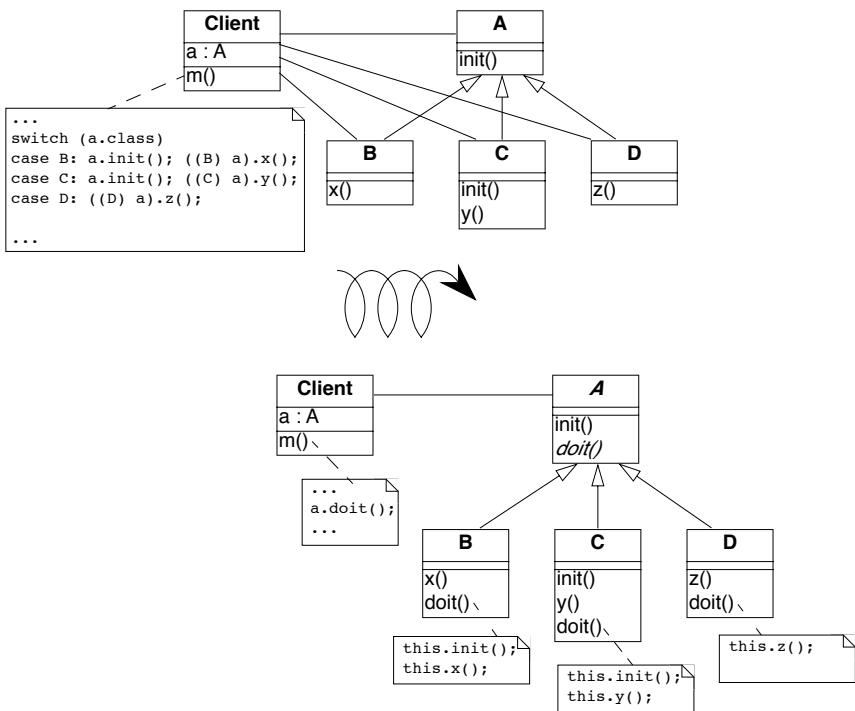


그림 10.6: 다형성 메서드 호출로 호출해야 하는 클라이언트의 메서드를 결정하는 데 사용되는 명시적 타입 검사 변환이다.

4. 모든 메서드가 이동되면 조건문의 각 Case 문이 새 메서드 호출로 구성되므로 전체 조건문을 새 메서드 호출 한 번으로 대체할 수 있다.
5. 프로파이더의 루트에서 메서드를 추상화하는 것을 고려하자. 또는 여기에 적절한 기본 동작을 구현하자.

고려할 다른 단계

- 여러 클라이언트가 정확히 동일한 테스트를 수행하고 동일한 동작을 수행하는 경우가 있을 수 있다. 이 경우 클라이언트 중 하나를 변환한 후 중복된 코드를 단일 메서드 호출로 대체할 수 있다. 클라이언트가 다른 테스트를 수행하거나 다른 조치를 취하는 경우 각 조건에 대해 패턴을 한

번씩 적용해야 한다.

- Case 문이 프로바이더 계층 구조의 모든 구체적인 클래스를 포함하지 않는 경우, 관련 클래스의 공통 수퍼클래스로 새로운 추상 클래스를 도입해야 할 수 있다. 그러면 새 메서드는 관련 하위 트리에만 도입된다. 또는 기존 상속 계층 구조에서 이러한 추상 클래스를 도입할 수 없는 경우 루트에서 비어 있는 기본 구현 또는 부적절한 클래스를 호출될 경우 예외를 발생시키는 구현을 사용하여 메서드를 구현하는 것을 고려하자.
- 조건문이 중첩된 경우 패턴을 재귀적으로 적용해야 할 수 있다.

트레이드오프

장점

- 프로바이더 계층은 다른 클라이언트도 사용할 수 있는 새로운 다형성 서비스를 제공한다.
- 이제 클라이언트의 코드가 더 잘 정리되어 더 이상 프로바이더가 책임져야 하는 문제를 처리할 필요가 없다.
- 단일 프로바이더의 동작에 관한 모든 코드가 이제 한 곳에 모인다.
- 프로바이더 계층 구조가 통일된 인터페이스를 제공하므로 클라이언트에 영향을 주지 않고 프로바이더를 수정할 수 있다.

단점

- 때로는 다양한 케이스를 처리하는 코드를 한 곳에서 보는 것이 편리할 때가 있다. 클라이언트 탑 검사 변환하기는 로직을 개별 프로바이더 클래스에 재배포하므로 전체적으로 살필 수 있는 기회는 손실된다.

어려움

- 일반적으로 프로바이더 클래스의 인스턴스는 이미 생성되어 있으므로 인스턴스 생성을 찾을 필요가 없지만 인터페이스를 리팩터링하면 프로바

이더 클래스의 모든 클라이언트에 영향을 미치므로 이러한 작업의 전체 결과를 고려하지 않고 수행해서는 안 된다.

레거시 솔루션이 최종 솔루션인 경우

그럼에도 불구하고 클라이언트 타입 검사는 프로바이더 인스턴스가 아직 존재하지 않거나 해당 클래스를 확장할 수 없는 경우 올바른 솔루션이 될 수 있다.

- 인스턴스화할 클래스를 알기 위해 타입 변수를 테스트해야 하는 추상 팩토리 [p. 349] 객체가 있을 수 있다. 예를 들어 팩토리는 텍스트 파일 표현에서 객체를 스트리밍하고 스트리밍된 객체가 어느 클래스에 속해야 하는지 알려주는 일부 변수를 테스트할 수 있다.
- 레거시 GUI 라이브러리와 같이 객체 지향이 아닌 라이브러리와 인터페이스하는 소프트웨어는 개발자가 디스패치를 수동으로 시뮬레이션해야 할 수 있다. 이러한 경우 프로시저 라이브러리에 대한 객체 지향 파사드를 개발하는 것이 비용 효율적인지 의문이다.
- 프로바이더 계층 구조가 고정된 경우(예: 소스 코드를 사용할 수 없기 때문에) 동작을 프로바이더 클래스로 전송할 수 없다. 이 경우 래퍼 클래스를 정의하여 프로바이더 클래스의 동작을 확장할 수 있지만 래퍼 정의의 복잡성이 추가되어 이점을 없을 수 있다.

예시

예전

다음 C++ 코드는 클라이언트가 수행할 작업을 결정하기 위해 명시적으로 전화의 검사 인스턴스를 타입으로 지정해야 하므로 잘못 배치된 책임을 보여준다. 굽게 표시된 코드는 이 접근 방식의 어려움을 강조한다.

```
class Telephone {
public:
    enum PhoneType {
        POTSPHONE, ISDNPHONE, OPERATORPHONE
    };
    Telephone() {}
```

```
PhoneType phoneType() { return myType; }

private:
    PhoneType myType;
protected:
    void setPhoneType(PhoneType newType) { myType = newType; }
};

class POTSPhone : public Telephone {

public:
    POTSPhone() { setPhoneType(POTSPHONE); }
    void tourneManivelle();
    void call();
};

...

class ISDNPhone: public Telephone {
public:
    ISDNPhone() { setPhoneType(ISDNPHONE);}
    void initializeLine();
    void connect();
};
...

class OperatorPhone: public Telephone {
public:
    OperatorPhone() { setPhoneType(OPERATORPHONE); }
    void operatorMode(bool onOffToggle);
    void call();
};

void initiateCalls(Telephone ** phoneArray, int numOfCalls) {
    for(int i = 0; i<numOfCalls ;i++ ) {
        Telephone * p = phoneArray[i];

        switch(p->phoneType()) {
            case Telephone::POTSPHONE: {
                POTSPhone *potsp = (POTSPhone *) p;
                potsp->tourneManivelle();
                potsp->call();
                break;
            }
            case Telephone::ISDNPHONE: {
                ISDNPhone *isdn = (ISDNPhone *) p;
                isdn->initializeLine();
            }
        }
    }
}
```

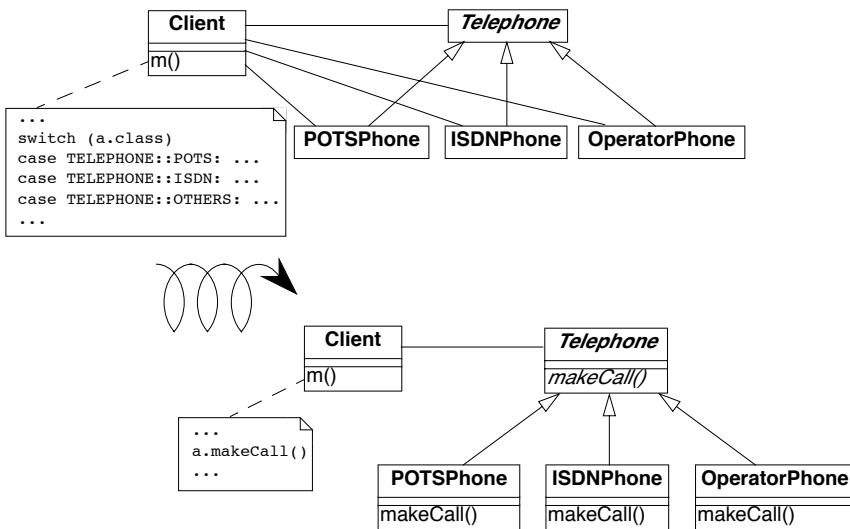


그림 10.7: 명시적 타입 검사를 다형성 메서드 호출로 변환한다.

```

isdn->connect();
break;
}
case Telephone::OPERATORPHONE: {
    OperatorPhone *opp = (OperatorPhone *) p;
    opp->operatorMode(true);
    opp->call();
    break;
}
default: cerr << "Unrecognized Phonetype" << endl;
};
}
}

```

이후

패턴을 적용한 후 클라이언트 코드는 다음과 같이 표시된다. (그림 10.7 도 참조하자.)

```

class Telephone {
public:

```

```
Telephone() {}  
virtual void makeCall() = 0;  
};  
  
Class POTSPhone : public Telephone {  
    void tourneManivelle();  
    void call();  
public:  
    POTSPhone() {}  
    void makeCall();  
};  
void POTSPhone::makeCall() {  
    this->tourneManivelle();  
    this->call();  
}  
  
class ISDNPhone: public Telephone {  
    void initializeLine();  
    void connect();  
  
public:  
    ISDNPhone() {}  
    void makeCall();  
};  
void ISDNPhone::makeCall() {  
    this->initializeLine();  
    this->connect();  
}  
  
class OperatorPhone: public Telephone {  
    void operatorMode(bool onOffToggle);  
    void call();  
public:  
    OperatorPhone() {}  
    void makeCall();  
};  
void OperatorPhone::makeCall() {  
    this->operatorMode(true);  
    this->call();  
}  
void initiateCalls(Telephone ** phoneArray, int numOfCalls) {  
    for(int i = 0; i<numOfCalls ;i++ ) {  
        phoneArray[i]->makeCall();  
    }  
}
```

근거

리엘은 “객체 탑입에 대한 명시적 Case 문을 이용한 분석은 일반적으로 예려 상황이다. 대부분의 경우 다형성을 사용해야 한다.”고 했다 [Rie96]. 실제로 클라이언트의 명시적 탑입 검사는 클라이언트와 프로바이더 간의 결합도를 높이기 때문에 책임이 잘못 배치되었다는 신호이다. 이러한 책임을 프로바이더에게 전가하면 다음과 같은 결과가 발생한다.

- 클라이언트는 모든 구체적인 하위 클래스 대신 프로바이더 계층의 루트만 명시적으로 알면 되므로 클라이언트와 프로바이더의 결합도가 더 약해진다.
- 프로바이더 계층 구조는 클라이언트 코드를 손상시킬 가능성을 줄이면서 더 우아하게 진화할 수 있다.
- 클라이언트 코드의 크기와 복잡성이 줄어든다. 클라이언트와 프로바이더 간의 협업이 더욱 추상화된다.
- 기존 디자인에 암시되어 있던 추상화(예: 조건문의 Case 문 동작)가 메서드로 명시되어 다른 클라이언트에서 사용할 수 있다.
- (동일한 조건문이 여러 번 발생하는 경우) 코드 중복이 줄어들 수 있다.

관련 패턴

클라이언트 탑입 검사 변환하기에서는 프로바이더 클래스의 탑입 정보에 따라 조건이 만들어진다. 널 객체 도입하기에서도 같은 상황이 발생하는데, 메서드를 호출하기 전에 널(Null) 값에 대한 조건문 테스트가 이루어진다. 이러한 관점에서 볼 때 널 객체 도입하기는 클라이언트 탑입 검사 변환하기의 특수화(specialization)라고 할 수 있다.

조건문을 등록으로 변환하기 은 클라이언트의 조건이 인자를 처리할 제3의 객체(일반적으로 외부 애플리케이션 또는 도구)를 선택하는 데 사용되는 특수한 경우를 처리한다.

이 리엔지니어링 패턴의 핵심 리팩터링은 조건문을 다형성으로 치환하기 [p. 349] 으로 [FBB⁺99]에 설명된 단계를 참조할 수 있다.

10.3 상태 추출하기

의도 개체의 상태에 대한 복잡한 조건문 코드를 상태 디자인 패턴을 적용하여 제거한다.

문제

현재 상태의 복잡한 평가에 따라 동작이 달라지는 클래스를 어떻게 더 확장 가능하게 만들 수 있을까?

이 문제는 다음과 같은 이유로 어렵다.

- 객체의 메서드에 여러 개의 복잡한 조건문이 분산되어 있다. 새로운 동작을 추가하면 이러한 조건문에 미묘한 방식으로 영향을 미칠 수 있다.
- 새로운 가능한 상태가 도입될 때마다 상태를 테스트하는 모든 메서드를 수정해야 한다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 객체의 인스턴스 변수는 일반적으로 각기 다른 추상 상태를 모델링하는데 사용되며, 각 상태에는 고유한 동작이 있다. 이러한 추상 상태를 식별 할 수 있다면 상태와 동작을 더 간단한 관련 클래스 집합으로 분해할 수 있다.

해결

즉, 상태 종속 동작(state-dependent behavior)을 별도의 객체로 캡슐화하고, 이러한 객체에 대한 호출을 델리게이션하고, 이러한 상태 객체의 올바른 인스턴스를 참조하여 객체의 상태를 일관되게 유지하는 상태 [p. 351] 패턴을 적용 한다. (그림 10.8 참조)

자신 타입 체크 변환하기에서와 같이 정량화된 상태를 테스트하는 복잡한 조건부 코드를 상태 클래스에 대한 델리게이션 호출로 변환한다. 상태 [p. 351]

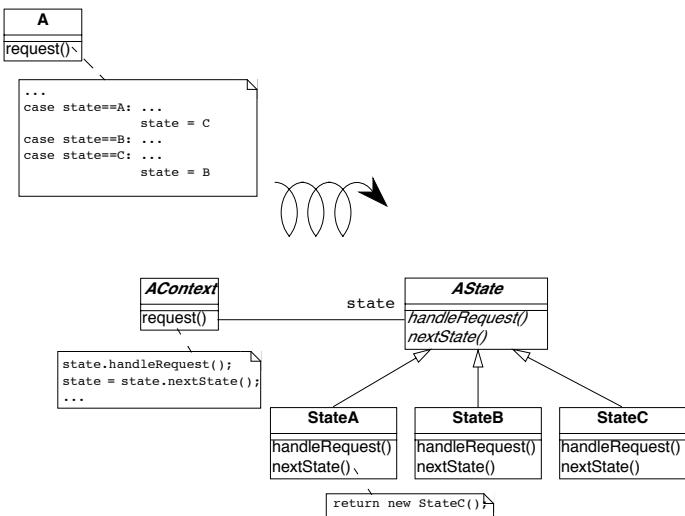


그림 10.8: 명시적 상태 조건문을 사용하여 상태 패턴을 흉내 낸 것에서 상태 패턴이 적용된 상황으로 변환한다.

패턴을 적용하여 각 조건 사례를 별도의 상태 객체에 델리게이션한다. 이 문제에 대한 자세한 설명과 토론은 상태 와 상태 패턴 [p. 352]를 읽어보기 바란다. [ABW98] [DA97]. 여기서는 패턴의 리엔지니어링 측면에만 초점을 맞춘다.

단계

1. 상태의 인터페이스와 상태의 개수를 식별한다.

운이 좋으면 각 조건이 같은 방식으로 상태 공간을 분할하고 상태 수는 각 조건의 Case 문의 수와 같을 것이다. 조건이 겹치는 경우에는 더 세밀한 분할이 필요하다.

상태의 인터페이스는 상태 정보에 액세스하고 업데이트하는 방법에 따라 달라지며, 후속 단계에서 개선해야 할 수도 있다.

2. 상태의 인터페이스를 나타내는 새 추상 클래스 **State**를 만든다.

3. 각 상태에 대해 **State**의 새 클래스 서브클래스를 만든다.

4. 각 상태 클래스에서 1단계에서 식별한 인터페이스의 메서드를 새 메서드에 조건의 해당 코드를 복사하여 정의한다. **Context**에서 인스턴스 변수의 상태가 **State** 클래스의 올바른 인스턴스를 참조하도록 변경하는 것을 잊지 말자. **State** 메서드는 항상 다음 상태 인스턴스를 참조하도록 **Context**를 변경해야 할 책임이 있다.
5. **Context** 클래스에 새 인스턴스 변수를 추가한다.
6. **State** 클래스에서 상태 전환을 호출하려면 **State**에서 **Context** 클래스로의 참조가 필요할 수 있다.
7. 기본 상태 클래스 인스턴스를 참조하도록 새로 생성된 인스턴스를 초기화한다.
8. 인스턴스 변수에 대한 호출을 멸리게이션하도록 테스트가 포함된 **Context** 클래스의 메서드를 변경한다.

단계 4는 리팩터링 브라우저의 메서드 추출하기 작업을 사용하여 수행할 수 있다. 각 단계가 끝난 후에도 회귀 테스트는 계속 실행되어야 한다는 점에 유의하자. 중요한 단계는 동작이 새로운 상태 객체에 멸리게이션되는 마지막 단계이다.

트레이드오프

장점

- **제한적 영향:** 원래 클래스의 퍼블릭 인터페이스는 변경할 필요가 없다. 상태 인스턴스는 원래 객체에서 멸리게이션을 통해 액세스되므로 클라이언트는 영향을 받지 않는다. 간단한 경우 이 패턴을 적용해도 클라이언트에 미치는 영향은 제한적이다.

단점

- 이 패턴을 체계적으로 적용하면 클래스 수가 폭발적으로 증가할 수 있다.
- 이 패턴은 다음과 같은 경우에 적용해서는 안 된다.

- 가능한 상태가 너무 많거나 상태 수가 고정되어 있지 않은 경우
- 코드에서 상태 전환이 언제 어떻게 발생하는지 파악하기 어려운 경우

레거시 솔루션이 최종 솔루션인 경우

이 패턴은 가볍게 적용해서는 안 된다.

- 상태가 명확하게 식별되고 변경되지 않을 것으로 알려진 경우 레거시 솔루션은 모든 상태 동작을 여러 하위 클래스에 분산하는 대신 기능별로 그룹화할 수 있는 이점이 있다.
- 파서와 같은 특정 도메인에서는 상태에 대한 조건부로 인코딩된 테이블 기반 동작이 잘 이해되며, 상태 객체를 제외하면 코드를 이해하기 어렵고 따라서 유지보수성이 떨어질 수 있다.

알려진 용도

*Design Patterns Smalltalk Companion [ABW98]*는 단계별 코드 변환에 대해 설명한다.

10.4 전략 추출하기

의도 전략 디자인 패턴을 적용하여 적합한 알고리즘을 선택하는 조건문 코드를 제거한다.

문제

어떤 변수 값의 테스트에 따라 동작이 달라지는 클래스를 어떻게 하면 더 확장 가능하게 만들 수 있을까?

이 문제는 다음과 같은 이유로 어렵다.

- 조건문 코드가 포함된 모든 메서드를 수정하지 않고는 새로운 기능을 추가할 수 없다.
- 조건문 코드는 적용할 알고리즘에 대해 유사한 결정을 내리는 여러 클래스에 분산되어 있을 수 있다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 대체 동작은 본질적으로 상호 교환이 가능한다.

해결

즉, 알고리즘 종속 동작을 다형성 인터페이스가 있는 별도의 객체로 캡슐화하고 이러한 객체에 대한 호출을 텔리게이션하는 전략 패턴을 적용한다(그림 10.9 참조).

단계

1. 전략 클래스의 인터페이스를 식별한다.
2. 전략의 인터페이스를 나타내는 새 추상 클래스 **Strategy**를 만든다.
3. 식별된 각 알고리즘에 대해 **Strategy**의 새 클래스 서브클래스를 만든다.

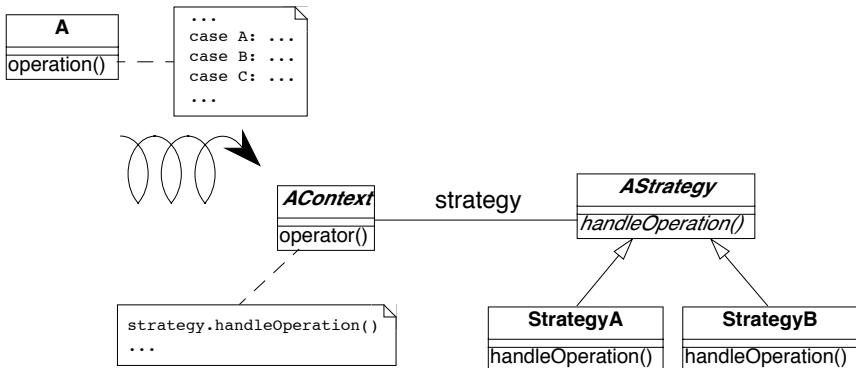


그림 10.9: 명시적 전략 조건문으로 전략 패턴을 흉내낸 것에서 전략 패턴이 적용된 상황으로 변환한다.

4. 테스트의 해당 코드를 메서드에 복사하여 각 전략 클래스에서 1단계에서 식별한 인터페이스의 메서드를 정의한다.
5. 현재 전략을 참조하기 위해 **Context** 클래스에 새 인스턴스 변수를 추가 한다.
6. **Context**가 유지 관리하는 정보에 대한 액세스를 제공하려면 **Strategy**에서 **Context** 클래스로의 참조가 필요할 수 있다(어려움 참조).
7. 기본 전략 인스턴스를 참조하도록 새로 생성된 인스턴스를 초기화한다.
8. 테스트를 제거하고 인스턴스 변수에 대한 호출을 델리게이션하여 테스트 가 포함된 **Context** 클래스의 메서드를 변경한다.

단계 4는 리팩터링 브라우저의 메서드 추출하기 작업을 사용하여 수행할 수 있다. 각 단계가 끝난 후에도 회귀 테스트는 계속 실행되어야 한다는 점에 유의하자. 중요한 단계는 동작이 새로운 **Strategy** 객체에 델리게이션되는 마지막 단계이다.

트레이드오프

장점

- 제한적 영향: 원래 클래스의 퍼블릭 인터페이스는 변경할 필요가 없다. **Strategy** 인스턴스는 원래 객체에서 딜리게이션을 통해 액세스되므로 클라이언트는 영향을 받지 않는다. 간단한 경우 이 패턴을 적용해도 클라이언트에 미치는 영향은 제한적이다. 그러나 이전에 구현된 모든 알고리즘이 이제 **Strategy** 클래스로 이동하기 때문에 **Context** 인터페이스가 축소된다. 따라서 이러한 메서드의 호출을 확인하고 케이스별로 결정해야 한다.
- 이 패턴을 적용하면 **Context**의 인터페이스 수정에 영향을 주지 않고 새 전략을 플러그할 수 있다. 새 전략을 추가해도 **Context** 클래스와 해당 클라이언트를 다시 컴파일할 필요가 없다.
- 이 패턴을 적용하면 **Context** 클래스와 **Strategy** 클래스의 인터페이스가 더 명확해진다.

단점

- 이 패턴을 체계적으로 적용하면 클래스가 폭발적으로 늘어날 수 있다. 20개의 서로 다른 알고리즘이 있는 경우 각각 하나의 메서드만 있는 20개의 새 클래스를 갖고 싶지 않을 수 있다.
- 객체 폭발(Object explosion). 전략은 애플리케이션의 인스턴스 수를 증가시킨다.

어려움

- **Context**와 **Strategy** 객체 간에 정보를 공유하는 방법에는 여러 가지가 있으며, 트레이드오프는 미묘할 수 있다. **Strategy** 메서드가 호출될 때 정보를 인자로 전달하거나, **Context** 객체 자체를 인자로 전달하거나, **Strategy** 객체가 해당 컨텍스트에 대한 참조를 보유할 수 있다. **Context**와 **Strategy** 사이의 관계가 매우 동적인 경우 이 정보를 메서드 인자로 전

달하는 것이 더 바람직할 수 있다. 이 문제에 대한 자세한 논의는 전략 [p. 352] 패턴에 대한 문헌에서 확인할 수 있다[GHJV95] [ABW98].

예시

Design Patterns Smalltalk Companion [ABW98]는 단계별 코드 변환에 대해 자세히 다룬다.

관련 패턴

전략 추출하기의 증상과 구조는 상태 추출하기와 비교해 볼 수 있다. 주요 차이점은 상태 추출하기는 객체의 가능한 상태가 다른 동작을 식별하는 반면, 전략 추출하기는 객체 상태와 무관한 상호 교환 가능한 알고리즘과 관련이 있다는 사실에 있다. 전략 추출하기를 사용하면 기존 전략 개체에 영향을 주지 않고 새 전략을 추가할 수 있다.

10.5 널 객체 도입하기

의도 널 객체 디자인 패턴을 적용하여 `null` 값을 테스트하는 조건문 코드를 제거한다.

문제

널(null) 값에 대한 반복 테스트가 있는 경우 클래스의 수정 및 확장을 쉽게 하려면 어떻게 해야 할까?

이 문제는 다음과 같은 이유로 어렵다.

- 클라이언트 메서드는 실제로 메서드를 호출하기 전에 항상 특정 값이 널(null)이 아닌지 테스트한다.
- 클라이언트 계층 구조에 새 하위 클래스를 추가하려면 일부 프로바이더 메서드를 호출하기 전에 널(null) 값을 테스트해야 한다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 클라이언트는 프로바이더가 널(null) 값을 나타내는지 알 필요가 없다.

해결

즉, 클라이언트 클래스가 널 테스트를 수행할 필요가 없도록 널 동작을 별도의 프로바이더 클래스로 캡슐화하여 널 객체 [p. 350] 패턴을 적용한다.

탐지

관용적인 널 테스트를 찾아보자.

널 테스트는 프로그래밍 언어와 테스트 대상 엔티티의 종류에 따라 다른 형태를 취할 수 있다. 예를 들어 Java에서 널 객체 참조의 값은 `null`인 반면, C++에서 널 객체 포인터의 값은 0이다.

단계

파울러는 필요한 리팩터링 단계에 대해 자세히 설명한다 [FBB⁺99].

1. 널 동작에 필요한 인터페이스를 식별한다. (일반적으로 널이 아닌 객체의 인터페이스와 동일하다.)
2. **RealObject** 클래스의 수퍼클래스로서 새 추상 수퍼클래스를 만든다.
3. 이름이 **No** 또는 **Null**로 시작하는 추상 슈퍼클래스의 새 하위 클래스를 만든다.
4. **Null Object** 클래스에 기본 메서드를 정의한다.
5. 검사한 인스턴스 변수 또는 구조체를 초기화하여 이제 **Null Object** 클래스의 인스턴스를 하나 이상 보유하도록 한다.
6. 클라이언트에서 조건부 테스트를 제거한다.

여전히 깔끔한 방식으로 `null` 값을 테스트하고 싶다면 파울러[FBB⁺99]가 설명한 대로 **RealObject** 및 **Null Object** 클래스에 `isNull`이라는 쿼리 메서드를 도입할 수 있다.

트레이드오프

장점

- 패턴을 적용한 후 클라이언트 코드가 훨씬 간단해졌다.
- 프로바이더의 인터페이스를 수정할 필요가 없으므로 패턴을 적용하기가 비교적 간단하다.

단점

- 프로바이더 계층 구조가 더 복잡해진다.

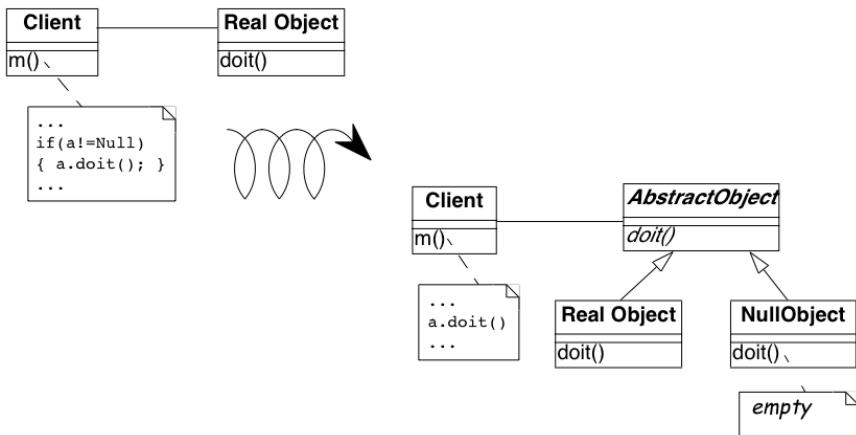


그림 10.10: 명시적인 널 값 테스트에 기반한 상황에서 널 객체가 도입된 상황으로 변환한다.

어려움

- 여러 클라이언트가 널(Null) 객체의 합리적인 기본 동작에 동의하지 않을 수 있다. 이 경우 여러 개의 널(Null) 객체 클래스를 정의해야 할 수 있다.

레거시 솔루션이 최종 솔루션인 경우

- 클라이언트가 공통 인터페이스에 동의하지 않는 경우.
- 변수를 직접 사용하는 코드가 거의 없거나 변수를 사용하는 코드가 한 곳에 잘 캡슐화되어 있는 경우.

예시

다음 Smalltalk 코드는 울프[Woo98]의 저작에서 가져온 것이다. 처음에 이 코드에는 다음과 같은 명시적 널(null) 테스트가 포함되어 있다.

VisualPart>>objectWantedControl

```

...
↑ctrl isNil
ifFalse:

```

```
[ctrl isControlWanted  
    ifTrue:[self]  
    ifFalse:[nil]]
```

이 코드는 다음과 같이 변환 가능하다.

VisualPart>>objectWantedControl

```
...  
↑ctrl isControlWanted  
    ifTrue:[self]  
    ifFalse:[nil]
```

Controller>>isControlWanted

```
    ↑self viewHasCursor
```

NoController>>isControlWanted

```
    ↑false
```

10.6 조건문을 등록으로 변환하기

의도 클라이언트의 조건문을 등록(*registration*) 메커니즘으로 대체하여 시스템의 모듈성을 개선한다.

문제

도구를 추가하거나 제거해도 클라이언트의 코드가 변경되지 않도록 서비스를 제공하는 도구(*tools*)와 클라이언트(*client*) 사이의 결합도를 줄이려면 어떻게 해야 할까?

이 문제는 다음과 같은 이유로 어렵다.

- 모든 종류의 도구를 한 곳에서 찾을 수 있으면 시스템을 쉽게 이해하고 새로운 도구를 추가하기 쉽다.
- 그러나 도구를 제거할 때마다 조건문에서 하나의 Case를 제거해야 하며, 그렇지 않으면 특정 부분(도구 클라이언트)에 제거된 도구의 존재가 여전히 반영되어 시스템이 취약해질 수 있다. 그러면 새 도구를 추가할 때마다 모든 도구 클라이언트에 새 조건문을 추가해야 한다.

그러나 이 문제를 해결할 수 있는 이유는 다음과 같다.

- 긴 조건문을 사용하면 사용되는 도구의 다양한 타입을 쉽게 식별할 수 있다.

해결

각 도구가 자체 등록을 담당하는 등록 메커니즘을 도입하고, 조건문을 수행하는 대신 등록 저장소를 쿼리하도록 도구 클라이언트를 변환한다.

단계

1. **플러그인 객체**(*plug-in objects*), 즉 도구를 등록하는 데 필요한 정보를 캡슐화하는 객체를 설명하는 클래스를 정의한다. 이 클래스의 내부 구조는 등록 목적에 따라 다르지만, 플러그인은 도구 관리자가 그것을 구분

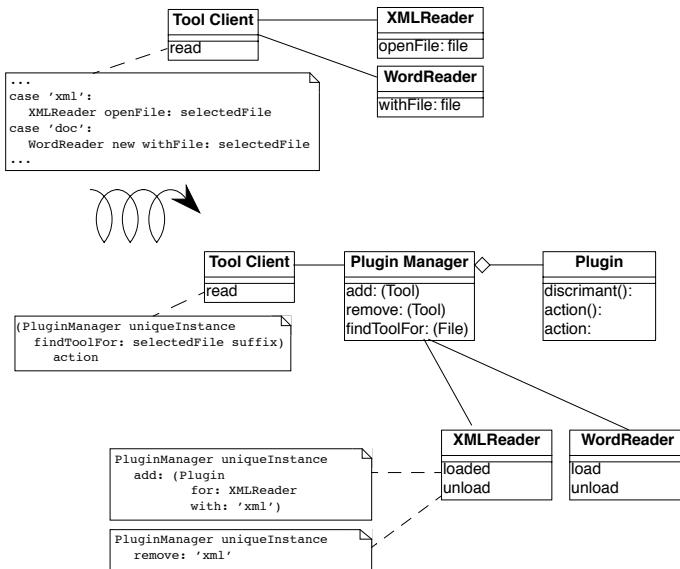


그림 10.11: 등록 메커니즘을 도입하여 도구 사용자의 조건문을 변환한다.

(*identify*)하고, 표현된 도구의 인스턴스를 생성(*create*)하고 메서드를 호출(*invoke*)할 수 있도록 필요한 정보를 제공해야 한다. 도구 메서드를 호출하려면 메서드 또는 블록 클로저나 내부 클래스와 같은 유사한 메커니즘이 플러그인 객체에 저장되어 있어야 한다.

2. 플러그인 객체를 관리하고 도구 클라이언트에서 도구의 존재 여부를 확인하기 위해 쿼리할 플러그인 관리자(*plug-in manager*)를 나타내는 클래스를 정의한다. 플러그인 관리자의 새 인스턴스가 생성될 경우 사용 가능한 도구를 나타내는 플러그인이 손실되지 않아야 하므로 이 클래스는 확실히 싱글턴(*singleton*)이 될 것이다.
3. 조건의 각 경우에 대해 주어진 도구와 연결된 플러그인 객체(*object*)를 정의한다. 이 플러그인 개체는 해당 도구가 로드될 때 자동으로 생성 및 등록되어야 하며, 해당 도구를 사용할 수 없게 되면 등록이 취소되어야 한다. 때로는 도구 클라이언트의 정보를 도구에 전달해야 할 때도 있다. 도구가 호출될 때 현재 도구 클라이언트를 인자로 전달할 수 있다.

4. 전체 조건식 표현식을 도구 관리자 개체에 대한 쿼리로 변환한다. 이 쿼리는 쿼리에 연결된 도구를 반환하고 이를 호출하여 원하는 기능에 액세스해야 한다.
5. 도구를 직접 활성화하는 도구 클라이언트 작업을 모두 제거한다. 이제 이 동작은 플러그인 관리자의 책임이다.

클라이언트 또는 플러그인 개체가 도구를 호출할 책임이 있을 수 있다. 플러그인 객체는 이미 도구를 나타내는 방법을 표현하는 책임을 가지고 있으므로 이 책임을 플러그인 객체에 맡기고 클라이언트는 도구 동작이 필요하다는 말만 하도록 하는 것이 좋다.

예시

Squeak [IKM⁺⁹⁷]에서 `FileList`는 스몰토크 코드, JPEG 이미지, MIDI 파일, HTML 등 다양한 종류의 파일을 로드할 수 있는 도구이다. 선택한 파일의 접미사에 따라 `FileList`는 사용자에게 다양한 작업을 제안한다. 이 예시에서는 파일 형식에 따라 다양한 파일을 로드하는 모습을 보여준다.

예전

`FileList` 구현은 파일의 접미사에 따라 가능한 다른 작업을 나타내는 다양한 메뉴 항목을 생성한다. 메뉴의 동적 부분은 파일 접미사를 인자로 취하고 메뉴 항목의 레이블과 `FileList` 객체에서 호출해야 하는 해당 메서드의 이름이 포함된 메뉴 항목을 반환하는 `menusForFileEnding:` 메서드에 정의되어 있다.

```
FileList>>menusForFileEnding: suffix
```

```
(suffix = 'jpg') ifTrue:
  [↑MenuItem label:'open image in a window'.
   selector: #openImageInWindow].
(suffix = 'morph') ifTrue:
  [↑MenuItem label: 'load as morph'.
   selector: #openMorphFromFile].
(suffix = 'mid') ifTrue:
  [↑MenuItem label: 'play midi file'.
   selector: #playMidiFile].
(suffix = 'st') ifTrue:
```

```
[↑MenuItem label: 'fileIn'.
  selector: #fileInSelection].
(suffix = 'swf') ifTrue:
  [↑MenuItem label: 'open as Flash'.
  selector: #openAsFlash].
(suffix = '3ds') ifTrue:
  [↑MenuItem label: 'Open 3DS file'.
  selector: #open3DSFile].
(suffix = 'wrl') ifTrue:
  [↑MenuItem label: 'open in Wonderland'.
  selector: #openVRMLFile].
(suffix = 'html') ifTrue:
  [↑MenuItem label: 'open in html browser'.
  selector: #openInBrowser].
(suffix = '*') ifTrue:
  [↑MenuItem label: 'generate HTML'.
  selector: #renderFile].
```

메뉴에 선택자가 연결된 메서드는 **FileList** 클래스에서 구현된다. 여기서는 두 가지 예제를 제공한다. 먼저 메서드는 필요한 도구를 사용할 수 있는지 확인하고, 사용할 수 없으면 경고음을 발생시키고, 그렇지 않으면 해당 도구가 생성된 다음 선택한 파일을 처리하는 데 사용된다.

```
FileList>>openInBrowser
Smalltalk at: #Scamper ifAbsent: [↑ self beep].
Scamper openOnUrl: (directory url , fileName encodeForHTTP)

FileList>>openVRMLFile
| scene |
Smalltalk at: #Wonderland ifAbsent: [↑ self beep].
scene := Wonderland new.
scene makeActorFromVRML: self fullName.
```

이후

해결책은 각 도구에 스스로 등록할 책임을 부여하고 **FileList**가 사용 가능한 도구의 레지스트리를 쿼리하여 호출할 수 있는 도구를 찾도록 하는 것이다.

단계 1. 해결책은 먼저 주어진 도구의 등록을 나타내는 **ToolPlugin** 클래스를 생성하는 것이다. 여기에 접미사 파일, 메뉴 레이블, 도구가 호출될 때 수행될 작업을 저장한다.

```
Object subclass: #ToolPlugin
instanceVariableNames: 'fileSuffix menuLabelName blockToOpen'
```

단계 2. 그런 다음 `PluginManager` 클래스가 정의된다. 등록된 도구를 보관하는 구조를 정의하고 등록된 도구를 추가, 제거, 찾는 동작을 정의한다.

```
Object subclass: #PluginManager
instanceVariableNames: 'plugins'
```

```
PluginManager>>initialize
plugins := OrderedCollection new.
```

```
PluginManager>>addPlugin : aPlugin
plugins add: aRegistree
```

```
PluginManager>>removePlugin: aBlock
```

```
(plugins select: aBlock) copy
do: [:each| plugins remove: each]
```

```
PluginManager>>findToolFor: aSuffix
"return a registree of a tool being able to treat file of format
aSuffix"
```

```
↑ plugins
detect: [:each| each suffix = aSuffix]
ifNone: [nil]
```

`findToolFor:` 메서드는 블록을 받아 이를 만족하는 플러그인 객체를 선택 할 수 있으며, 현재 주어진 파일 형식을 처리할 수 있는 모든 도구를 나타내는 플러그인 목록을 반환할 수 있다는 점에 유의하자.

단계 3. 그런 다음 도구가 메모리에 로드될 때 스스로 등록해야 한다. 여기에 서는 각 도구에 대해 플러그인 객체가 생성되었음을 보여주는 두 가지 등록을 제시한다. 도구는 파일 이름이나 디렉토리와 같은 `FileList` 객체의 일부 정보가 필요하므로 수행해야 하는 작업은 이를 호출하는 `FileList` 객체의 인스턴스를 매개변수로 사용한다(아래 코드에서 `[:fileList |...|]`).

Squeak에서 클래스가 클래스(정적) `initialize` 메서드를 지정하면 클래스가 메모리에 로드된 후 이 메서드가 호출된다. 그런 다음 `Scamper` 및 `Wonderland` 클래스의 클래스 메서드 `initialize`를 특수화하여 아래에 정의된 클래스 메서드

toolRegistration을 호출한다.

Scamper class>>toolRegistration

```
PluginManager uniqueInstance
    addPlugin:
        (ToolPlugin
            forFileSuffix: 'html'
            openingBlock:
                [:fileList |
                self openOnUrl:
                    (fileList directory url ,
                     fileList fileName encodeForHTTP)]
            menuLabelName: 'open in html browser')
```

Wonderland class>>toolRegistration

```
PluginManager uniqueInstance
    addPlugin:
        (ToolPlugin
            forFileSuffix: 'wrl'
            openingBlock:
                [:fileList |
                | scene |
                scene := self new.
                scene makeActorFromVRML: fileList fullName]
            menuLabelName: 'open in Wonderland')
```

Squeak에서는 클래스가 시스템에서 제거되면 removeFromSystem 메시지를 받는다. 그런 다음 모든 도구가 스스로 등록을 취소할 수 있도록 이 메서드를 모든 도구에 맞게 특수화한다.

Scamper class>>removeFromSystem

```
super removeFromSystem.
PluginManager uniqueInstance
    removePlugin: [:plugin| plugin forFileSuffix = 'html']
```

Wonderland class>>removeFromSystem

```
super removeFromSystem.
PluginManager uniqueInstance
    removePlugin: [:plugin| plugin forFileSuffix = 'wrl']
```

단계 4. 이제 **FileList** 객체는 선택한 파일의 접미사에 따라 올바른 플러그인

객체를 식별하기 위해 `ToolsManager`를 사용해야 한다. 그런 다음 주어진 접미사에 사용할 수 있는 도구가 있으면 도구와 연결된 작업 블록의 인자로 `FileList`를 전달하도록 지정하는 메뉴 항목을 생성한다. 도구가 없는 경우 아무 작업도 수행하지 않는 특수 메뉴가 생성된다.

`FileList>>itemsForFileEnding: suffix`

```
| plugin |
plugin := PluginManager uniqueInstance
           findToolFor: suffix ifAbsent: [nil].
↑ plugins isNil
ifFalse: [Menu label: (plugin menuLabelName)
           actionBlock: (plugin openingBlock)
           withParameter: self]
ifTrue: [ErrorMenu new
          label: 'no tool available for the suffix ', suffix]
```

트레이드오프

장점

- 조건문을 등록으로 변환하기 을 적용하면 동적이고 유연성 있는 시스템을 얻을 수 있다. 도구 클라이언트에 영향을 주지 않고 새로운 도구를 추가할 수 있다.
- 도구 클라이언트는 더 이상 특정 도구의 사용 가능 여부를 확인할 필요가 없다. 등록 메커니즘은 작업을 수행할 수 있도록 보장한다.
- 도구와 도구 클라이언트 간의 상호 작용 프로토콜이 이제 정규화되었다.

단점

- 도구 표현을 나타내는 객체(플러그인)와 등록된 도구를 관리하는 객체(플러그인 관리자)를 위한 두 개의 새 클래스를 정의해야 한다.

어려움

- 조건의 분기를 플러그인 개체로 변환하는 동안 플러그인 개체를 통해 도구와 관련된 작업을 정의해야 한다. 명확한 분리와 완전한 동적 등록을

보장하려면 이 작업은 도구 클라이언트가 아닌 도구에 정의해야 한다. 그러나 도구는 도구 클라이언트의 일부 정보를 필요로 할 수 있으므로 작업이 호출될 때 도구 클라이언트를 매개변수로 도구에 전달해야 한다. 이렇게 하면 도구와 도구 클라이언트 간의 프로토콜이 도구 클라이언트에서의 단일 호출에서 추가 매개변수가 있는 도구에 대한 메서드 호출로 변경된다. 이는 또한 경우에 따라 도구 클라이언트 클래스가 도구가 도구 클라이언트 권한 정보에 액세스할 수 있도록 새로운 공용 또는 친구 메서드를 정의해야 함을 의미한다.

- 각 조건부 분기가 하나의 도구에만 연결되는 경우 플러그인 개체는 하나만 필요하다. 그러나 동일한 도구를 여러 가지 방법으로 호출할 수 있는 경우에는 여러 개의 플러그인 객체를 만들어야 한다.

레거시 솔루션이 최종 솔루션인 경우

- 도구 클라이언트 클래스가 하나만 있고 모든 도구를 항상 사용할 수 있으며 런타임에 도구를 추가하거나 제거하지 않을 경우 조건문이 더 간단하다.

관련 패턴

조건문을 등록으로 변환하기 및 클라이언트 유형 검사 변환하기는 어떤 객체에 대해 어떤 메서드를 호출할지 결정하는 조건문 표현식을 제거한다. 두 패턴의 주요 차이점은 클라이언트 타입 검사 변환하기는 동작을 클라이언트에서 서비스 프로바이더로 이동한다는 점이다, 반면 조건문을 등록으로 변환하기는 외부 도구로 구현되어 이동이 불가능한 동작을 처리한다.

C++ 코드에서 스위치 문을 흉내낸 코드 찾는 스크립트

이 Perl 스크립트는 C++ 파일에서 메서드를 검색하여 Case 문을 대체하여 사용될 수 있는 if then else 문 즉, elseIf에서 X를 , //... 또는 캐리지 리턴을 포함한 일부 공백으로 대체할 수 있는 표현식과 일치하는 관련된 코드들이 있는지 나열한다.

```
#!/opt/local/bin/perl
$/ = ':-';
# new record delim.,
$elseifPattern = 'else[\s\n]*{?[\s\n]*if';
$linecount = 1;
while (<>) {
    s/(//.)/g; # remove C++ style comments
    $lc = (split /\n/) - 1; # count lines

    if($elseifPattern) {
        # count # of lines until first
        # occurrence of "else if"
        $temp = join("", $` , $&);
        $l = $linecount + split(/\n/, $temp) - 1;
        # count the occurrences of else-if pairs,
        # flag the positions for an eventual printout
        $swc = s/(else)([\s\n]*{?[\s\n]*if)
                  /$1\n      * HERE *$2/g;
        printf "\n%ss: Statement with
               %2d else-if's, first at: %d",
               $ARGV, $swc, $l;
    }
    $linecount += $lc;
    if(eof) {
        close ARGV;
        $linecount = 0;
        print "\n";
    }
}
```


제 IV 편

부록

부록 A

섬네일 패턴

리엔지니어링과 특별히 관련되지는 않지만 리엔지니어링 프로세스와 관련이 있는 패턴도 많이 있다. 이 장에서는 이 책의 어느 지점에서 구체적으로 언급된 패턴만 나열했다. 다음 세 가지 범주로 분류했다.

- 테스팅 패턴(*Testing patterns*). 이러한 패턴은 테스트 작업에 집중하는 데 도움이 된다. 물론 이 주제에 대한 방대한 문헌이 있지만, 주요 출처는 델라노와 라이징 [DR98]의 패턴 언어이다. 예를 들어 바인더는 책 한 권 전체를 이 주제에 할애하고 있다 [Bin99].
- 리팩터링 패턴(*Refactoring patterns*). 이 패턴은 리엔지니어링 프로젝트 중에 적용할 수 있거나 포워드 엔지니어링 프로젝트 중에도 적용할 수 있는 개별 리팩토링 단계에 중점을 둔다. 주요 출처는 파울러의 공저 [FBB⁺99]와 로버츠의 박사 학위 논문 [Rob99]이 있다.
- 디자인 패턴(*Design patterns*). 리엔지니어링 작업의 결과는 특정 디자인 패턴을 제자리에 배치하는 경우가 매우 많다. 여기서는 리엔지니어링 상황에서 가장 흔히 나타나는 디자인 패턴 몇 가지를 소개한다. 물론 주요 출처는 디자인 패턴 책 [GHJV95]이다.

A.1 테스팅 패턴

A.1.1 지속적인 문제 재테스트하기

문제: 구현 중인 기능에 관계없이 시스템에서 어떤 영역을 집중적으로 테스트해야 하는가?

솔루션: 현재 문제를 해결할 뿐만 아니라 후속 테스트에 사용할 수 있도록 지속적으로 발생하는 문제 영역과 테스트 케이스 목록을 작성하여 이를 검증하자. 새로운 기능이 추가되지 않더라도 이러한 영역을 철저하게 테스트하자. 릴리스가 출시되기 전에 마지막으로 한 번이라도 정기적으로 다시 테스트하자.

출처: Patterns for system testing [DR98].

참조처: 변경할 때마다 회귀 테스트하기 [p. 223].

A.1.2 폐지 기능 테스트하기

문제: 시스템에서 발생할 수 있는 문제 영역을 정확히 찾아내어 최소한의 시간으로 가장 많은 문제를 발견하려면 어떻게 해야 할까?

솔루션: 시스템에서 제공되는 문서를 학습하자. 모호하거나 제대로 정의되지 않은 영역을 찾아보자. 이러한 영역을 더 철저하게 다루는 테스트 계획을 작성하고 해당 영역에 집중적으로 테스트하자. 디자이너가 기능에 대해 모든 것을 말할 수 있다면 아마도 작동할 것이다. 테스트 중에 주의가 필요한 것은 그들이 이야기 해줄 수 없는 부분이다.

출처: Patterns for system testing [DR98].

참조처: 기본 테스트를 증가시키기 [p. 175].

A.1.3 오래된 버그 테스트하기

문제: 최소한의 시간에 가장 많은 문제를 발견할 수 있도록 시스템의 어떤 영역을 테스트 대상으로 삼아야 하는가?

솔루션: 이전 릴리스의 문제 보고서를 검토하여 테스트 사례를 선택하는 데 도움을 받자. 오래된 문제를 모두 테스트하는 것은 비효율적이므로 시스템의 마지막 유효성 검사 스텝샷 이후에 보고된 문제를 살펴보자. 문제 보고서를 분류하여 추가 테스트에 사용할 수 있는 추세가 확인되는지 확인하자.

출처: Patterns for system testing [DR98].

참조처: 기본 테스트를 증가시키기 [p. 175].

A.2 리팩터링

A.2.1 필드 캡슐화하기

다른 명칭: 추상 인스턴스 변수(Abstract Instance Variable) [Rob99].

의도 퍼블릭 필드가 있으면, 프라이빗으로 설정하고 접근자를 제공하자.

출처: Refactoring: Improving the Design of Existing Code [FBB⁺99].

참조처: 탐색 코드 제거하기 [p. 279].

A.2.2 메서드 추출하기

의도 함께 그룹화할 수 있는 코드 조각이 있다. 이 조각을 메서드의 목적을 설명하는 이름을 가지는 메서드로 추출하자.

출처: *Refactoring: Improving the Design of Existing Code* [FBB⁺99].

참조처: 이해하기 위해 리팩터링하기 [p. 141], 도트 플롯으로 코드 시각화하기 [p. 255], 데이터 가까이 동작 이동하기 [p. 269]

A.2.3 메서드 이동하기

의도 메서드가 정의된 클래스보다 다른 클래스의 더 많은 기능을 사용하거나 사용할 메서드가 있다. 가장 많이 사용하는 클래스에서 새 메서드를 만든다. 기존 메서드를 단순한 텔리게이션으로 바꾸거나 완전히 제거하자.

출처: *Refactoring: Improving the Design of Existing Code* [FBB⁺99].

참조처: 이해하기 위해 리팩터링하기 [p. 141], 동작을 데이터 가까이 이동하기 [p. 269]

A.2.4 속성 이름 바꾸기

의도 인스턴스 변수의 이름을 바꾸고 그에 대한 모든 참조를 업데이트한다.

출처: *Practical Analysis for Refactoring* [Rob99].

참조처: 이해하기 위해 리팩터링하기 [p. 141].

A.2.5 메소드 이름 바꾸기

의도 메소드의 이름이 그 목적을 드러내지 않는면, 메소드 이름을 변경하자.

출처: *Refactoring: Improving the Design of Existing Code* [FBB⁺99].

참조처: 이해하기 위해 리팩터링하기 [p. 141]

A.2.6 조건문을 다형성으로 치환하기

의도 개체의 타입에 따라 다른 동작을 선택하는 조건문이 있다. 조건의 각 레그를 서브클래스의 재정의 메서드로 옮긴다. 원본의 메서드는 추상화하자.

출처: *Refactoring: Improving the Design of Existing Code* [FBB⁺99].

참조처: 클라이언트 타입 검사 변환하기 [p. 311]

A.3 디자인 패턴

A.3.1 추상 팩토리

의도 구체적인 클래스를 지정하지 않고 관련 또는 종속 개체의 패밀리를 만들기 위한 인터페이스를 프로바이더가 제공한다.

출처: *Design Patterns* [GHJV95].

참조처: 컨트랙트 찾기 [p. 151], 클라이언트 타입 검사 변환하기 [p. 311].

A.3.2 어댑터

의도 클래스의 인터페이스를 클라이언트가 기대하는 다른 인터페이스로 변환한다. 어댑터를 사용하면 호환되지 않는 인터페이스 때문에 함께 작동하지 않던 클래스가 함께 작동할 수 있다.

출처: *Design Patterns* [GHJV95].

참조처: 올바른 인터페이스 제시하기 [p. 229], 동작을 데이터 가까이 이동하기 [p. 269].

A.3.3 파사드

의도 서브시스템의 인터페이스 집합에 통합 인터페이스를 프로바이더로 제공한다. 파사드는 하위 시스템을 더 쉽게 사용할 수 있는 상위 레벨 인터페이스를 정의한다.

출처: *Design Patterns* [GHJV95].

참조처: 탐색 코드 제거하기 [p. 279], 신 클래스 분할하기 [p. 289].

A.3.4 팩토리 메서드

의도 개체를 생성하기 위한 인터페이스를 정의하되, 인스턴스화할 클래스는 서브클래스가 결정하도록 한다. 팩토리 메서드는 클래스가 인스턴스화를 서브 클래스로 연기할 수 있도록 한다.

출처: *Design Patterns* [GHJV95].

참조처: 컨트랙트 찾기 [p. 151]

A.3.5 경량

의도 공유를 사용하여 많은 수의 세분화된 개체를 효율적으로 지원한다.

출처: *Design Patterns* [GHJV95].

참조처: 디자인 추측하기 [p. 109]

A.3.6 널 객체

의도 널 객체는 동일한 인터페이스를 공유하지만 아무 일도 하지 않는 다른 객체에 대한 대리자(surrogate)를 제공한다. 따라서 널 객체는 아무것도 하지

않는 방법에 대한 구현 결정을 캡슐화하고 해당 세부 사항을 공동 작업자에게 숨긴다.

출처: *Null Object* [Woo98].

참조처: 널 객체 도입하기 [p. 329].

A.3.7 수량

문제: 6피트 또는 \$5와 같은 값을 나타냅니다.

솔루션: 금액과 단위를 모두 포함하는 수량 타입을 사용한다. 달러와 같은 통화는 일종의 단위(unit)이다.

출처: *Analysis Patterns: Reusable Objects Models* [Fow97].

참조처: 퍼시스턴트 데이터 분석하기 [p. 97].

A.3.8 싱글턴

의도 클래스에 인스턴스가 하나만 있는지 확인하고 이에 대한 전역 액세스 지점을 제공한다.

출처: *Design Patterns* [GHJV95].

참조처: 한 시간 안에 모든 코드 읽기 [p. 59].

A.3.9 상태

의도 객체의 내부 상태가 변경될 때 객체의 동작을 변경하도록 허용한다. 객체가 클래스를 변경하는 것처럼 보인다.

출처: *Design Patterns* [GHJV95].

참조처: 상태 추출하기 [p. 321].

A.3.10 상태 패턴

의도 *State* 패턴 패턴 언어는 *State* 패턴을 구체화(refine and clarify)한다.

출처: *State Patterns* [DA97].

참조처: 상태 추출하기 [p. 321].

A.3.11 전략

의도 알고리즘 제품군을 정의하고, 각 알고리즘을 별도의 클래스로 캡슐화하며, 각 클래스를 동일한 인터페이스로 정의하여 상호 교환할 수 있도록 한다. 전략을 사용하면 알고리즘을 사용하는 클라이언트와 독립적으로 알고리즘을 변경할 수 있다.

출처: *Design Patterns* [GHJV95].

참조처: 전략 추출하기 [p. 325].

A.3.12 템플릿 메서드

의도 오퍼레이션에서 알고리즘의 골격을 정의하여 일부 단계를 하위 클래스로 연기한다. 템플릿 메서드를 사용하면 서브클래스가 알고리즘의 구조를 변경하지 않고 알고리즘의 특정 단계를 재정의할 수 있다.

출처: *Design Patterns* [GHJV95].

참조처: 계약 찾기 [p. 151].

A.3.13 비지터

의도 객체 구조의 요소에 대해 수행할 오퍼레이션을 나타낸다. 비지터를 사용하면 연산이 수행되는 요소의 클래스를 변경하지 않고도 새 오퍼레이션을 정의할 수 있다.

출처: *Design Patterns* [GHJV95].

참조처: 동작을 데이터 가까이 이동하기 [p. 269].

참고 문헌

- [ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.
- [Arn92] Robert S. Arnold. *Software Reengineering*. IEEE Computer Society Press, Los Alamitos CA, 1992.
- [Bak92] Brenda S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.
- [BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.
- [BDS⁺00] Mike Beedle, Martine Devos, Yonat Sharon, Ken Schwaber, and Jeff Sutherland. Scrum: A pattern language for hyper-productive software development. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, pages 637–652. Addison Wesley, 2000.
- [BE96] Timothy Ball and Stephen Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.
- [Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.
- [Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.
- [BF01] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addison Wesley, 2001.

- [BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.
- [BH95] Olin Bray and Michael M. Hess. Reengineering a configuration management system. *IEEE Software*, 12(1):55–63, January 1995.
- [Big89] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22:36–49, October 1989.
- [Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.
- [BLM98] M. Blaha, D. LaPlant, and E. Marvak. Requirements for repository software. In *Proceedings of WCRE '98*, pages 164–173. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.
- [BMF99] Simon Bennett, Steve McRobb, and Ray Farmer. *Object-Oriented System Analysis and Design using UML*. McGraw Hill, 1999.
- [BMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley Press, 1996.
- [BMW93] Ted J. Biggerstaff, Bharat G. Mittbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering (ICSE 1993)*. IEEE Computer, 1993.

- [BMW94] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, May 1994.
- [Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
- [Boo94] Grady Booch. *Object Oriented Analysis and Design with Applications*. The Benjamin Cummings Publishing Co. Inc., 2nd edition, 1994.
- [BP94] Jack Barnard and Art Price. Managing code inspection information. *IEEE Software*, 11(2):59–69, March 1994.
- [Bro75] Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley, Reading, Mass., 1975.
- [Bro87] Frederick P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, April 1987.
- [Bro96] Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Master’s thesis, North Carolina State University, 1996.
- [BS95] Michael L. Brodie and Michael Stonebraker. *Migrating Legacy Systems*. Morgan Kaufmann, 1995.
- [BS97] David Bellin and Susan Suchman Simone. *The CRC Card Book*. Addison Wesley, 1997.
- [BW96] Kyle Brown and Bruce G. Whitenack. Crossing chasms: A pattern language for object-rdbms integration. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 227–238. Addison Wesley, 1996.
- [CHR01] Stephen Cook, Rachel Harrison, and Brian Ritchie. Assessing the evolution of financial management information sys-

- tems. In *ECOOP 2001 Workshop Reader*, volume 2323 of *LNCS*. Springer-Verlag, 2001.
- [CI92] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. In Robert S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.
- [Coc93] Alistair Cockburn. The impact of object-orientation on application development. *IBM Systems Journal*, 32(3):420–444, March 1993.
- [Con68] Melvin E. Conway. How do committees invent? *Datamation*, 14(4):28–31, April 1968.
- [Cop92] James O. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison Wesley, 1992.
- [Cop95] James O. Coplien. A development process generative pattern language. In James O. Coplien and Douglas Schmidt, editors, *Pattern Languages of Program Design*, pages 183–237. Addison Wesley, 1995.
- [Cor89] Thomas A. Corbi. Program understanding: Challenge for the 1990’s. *IBM Systems Journal*, 28(2):294–306, 1989.
- [CRR96] Debra Cameron, Bill Rosenblatt, and Eric Raymond. *Learning GNU Emacs*. O’Reilly, 1996.
- [DA97] Paul Dyson and Bruse Anderson. State patterns. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*. Addison Wesley, 1997.
- [Dav95] Alan Mark Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.
- [DD99] Serge Demeyer and Stéphane Ducasse. Metrics, do they really help? In Jacques Malenfant, editor, *Proceedings of Lan-*

- guages et Modèles à Objets (LMO'99), pages 69–82. HERMES Science Publications, Paris, 1999.
- [DDL99] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings of 6th Working Conference on Reverse Engineering (WCRE '99)*. IEEE Computer Society, October 1999.
- [DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 166–178, New York NY, 2000. ACM Press. Also appeared in ACM SIGPLAN Notices 35 (10).
- [DG97] Serge Demeyer and Harald Gall, editors. *Proceedings of the ESEC/FSE Workshop on Object-Oriented Re-engineering*. TUV-1841-97-10. Technical University of Vienna — Information Systems Institute — Distributed Systems Group, September 1997.
- [DL99] Tom DeMarco and Timothy Lister. *Peopleware, Productive Projects and Teams*. Dorset House, 2nd edition, 1999.
- [DR98] David E. DeLano and Linda Rising. Patterns for system testing. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pages 503–527. Addison-Wesley, 1998.
- [DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance*

- (ICSM'99), pages 109–118. IEEE Computer Society, September 1999.
- [FBB⁺99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [FH79] R.K. Fjeldstad and W. T. Hamlen. Application program maintenance study: report to our respondents. In *Proceedings of GUIDE 48*. The Guide Corporation, 1979.
- [FMvW97] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 472–495, Jyvaskyla, Finland, June 1997. Springer-Verlag.
- [Fow97] Martin Fowler. *Analysis Patterns: Reusable Objects Models*. Addison Wesley, 1997.
- [FP96] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.
- [Fro94] Stuart Frost. Modelling for the rdbms legacy. *Object Magazine*, pages 43–51, September 1994.
- [FY00] Brian Foote and Joseph W. Yoder. Big ball of mud. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design*, volume 4, pages 654–692. Addison Wesley, 2000.
- [GG93] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison Wesley, 1993.
- [GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM*

- '98), pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.
- [Gla97] Robert L. Glass. *Building Quality Software*. Prentice-Hall, 1997.
- [GR95] Adele Goldberg and Kenneth S. Rubin. *Succeeding With Objects: Decision Frameworks for Project Management*. Addison Wesley, Reading, Mass., 1995.
- [GW99] Harald Gall and Johannes Weidl. Object-model driven abstraction-to-code mapping. In *Proceedings of the 2nd Workshop on Object-Oriented Reengineering (WOOR 1999)*. Technical University of Vienna — Technical Report TUV-1841-99-13, 1999.
- [Har96] Neil B. Harrison. Organizational patterns for teams. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 345–352. Addison Wesley, 1996.
- [HEH⁺96] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Database reverse engineering: From requirements to CARE tools. *Automated Software Engineering*, 3(1-2), June 1996.
- [Hel95] Jonathan I. Helfman. Dotplot patterns: a literal look at pattern languages. *TAPOS*, 2(1):31–41, 1995.
- [IKM⁺97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, November 1997.

- [JAH01] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison Wesley, 2001.
- [JC00] Daniel Jackson and John Chapin. Redesigning air traffic control: An exercise in software design. *IEEE Software*, 17(3):63–70, May 2000.
- [JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison Wesley/ACM Press, Reading, Mass., 1992.
- [JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse*. Addison Wesley/ACM Press, 1997.
- [JGR99] Mehdi Jazayeri, Harald Gall, and Claudio Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society Press, 1999.
- [JS96] Dean F. Jerding and John T. Stasko. The information mural: Increasing information bandwidth in visualizations. Technical Report GIT-GVU-96-25, Georgia Institute of Technology, October 1996.
- [JSZ97] Jens. H. Jahnke, Wilhelm. Schäfer, and Albert. Zündorf. Generic fuzzy reasoning nets as a basis of reverse engineering relational database applications. In *Proceedings of ESCC/FSE '97*, number 1301 in LNCS, pages 193–210, 1997. in-proceedings.
- [KC98a] Rick Kazman and S. Jeromy Carriere. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, B.C., 1998.

- [KC98b] Wolfgang Keller and Jens Coldewey. Accessing relational databases: A pattern language. In Robert Martin, Dirk Riehle, and Frank Bushmann, editors, *Pattern Languages of Program Design 3*, pages 313–343. Addison Wesley, 1998.
- [KC99] Rick Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, April 1999.
- [Kel00] Wolfgang Keller. The bridge to the new town — a legacy system migration pattern. In *Proceedings of EuroPLoP 2000*, 2000.
- [Knu92] Donald E. Knuth. *Literate Programming*. Stanford, California: Center for the Study of Language and Information, 1992.
- [Lan99] Michele Lanza. Combining Metrics and Graphs for Object Oriented Reverse Engineering. Diploma Thesis, University of Bern, October 1999.
- [LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.
- [Lea96] Doug Lea. *Concurrent Programming in Java, Design Principles and Patterns*. The Java Series. Addison Wesley, 1996.
- [LHR88] Karl J. Lieberherr, Ian M. Holland, and Arthur Riel. Object-oriented programming: An objective sense of style. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, volume 23, pages 323–334, November 1988.
- [LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.
- [Lov93] Tom Love. *Object Lessons — Lessons Learned in Object-Oriented Development Projects*. SIGS Books, New York, 1993.

- [LPM⁺97] Bruno Lagu , Daniel Proulx, Ettore M. Merlo, Jean Mayrand, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of ICSM (International Conference on Software Maintenance)*. IEEE, 1997.
- [Mar82] Tom De Marco. *Controlling Software Projects*. Yourdon Press, 1982.
- [Mey96] Scott Meyers. *More Effective C++*. Addison Wesley, 1996.
- [Mey98] Scott Meyers. *Effective C++*. Addison Wesley, second edition, 1998.
- [MJS⁺00] Hausi A. M ller, Jens H. Janhke, Dennis B. Smith, Margaret Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering 2000*. ACM Press, 2000.
- [MLM96] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance (ICSM)*, pages 244–253, 1996.
- [MN97] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *IEEE Computer*, 8:29–36, 1997.
- [Nes88] Paolo Nesi. Managing OO project better. *IEEE Software*, July 1988.
- [Nie99] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1999.
- [O'C00] Alan O'Callaghan. Patterns for architectural praxis. In *Proceedings of EuroPLoP 2000*, 2000.
- [ODF99] Alan O'Callaghan, Ping Dai, and Ray Farmer. Patterns for change — sample patterns from the adaptor pattern language. In *Proceedings of EuroPLoP 1999*, 1999.

- [PB94] William J. Premerlani and Michael R. Blaha. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5):42–49, May 1994.
- [PK82] J. Pustell and F. Kafatos. A high speed, high capacity homology matrix: Zooming through SV40 and polyoma. *Nucleic Acids Research*, 10(15):4765–4782, 1982.
- [PK01] Joseph Pelrine and Alan Knight. *Mastering ENVY/Developer*. Cambridge University Press, 2001.
- [Pre94] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1994.
- [RBCM91] D.J. Robson, K. H. Bennet, B. J. Cornelius, and M. Munro. Approaches to program comprehension. *Journal of Systems and Software*, 14:79–84, February 1991. Republished in [Arno92a].
- [RB97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.
- [RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 13–22, Los Alamitos CA, September 1999. IEEE Computer Society Press.
- [Ree96] Trygve Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.
- [RG98] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, pages 117–133, October 1998.
- [Rie96] Arthur Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

- [Ris00] Linda Rising. Customer interaction patterns. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, pages 585–609. Addison Wesley, 2000.
- [Rob99] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.
- [RS89] Trygve Reenskaug and Anna Lise Skaar. An environment for literate Smalltalk programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 337–346, October 1989.
- [RW98] Spencer Rugaber and Jim White. Restoring a legacy: Lessons learned. *IEEE Software*, 15(4):28–33, July 1998.
- [SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [Sha97] Alec Sharp. *Smalltalk by Example*. McGraw-Hill, 1997.
- [SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA '96 (International Conference on Object-Oriented Programming, Systems, Languages, and Applications)*, pages 268–285. ACM Press, 1996.
- [Sne99] Harry M. Sneed. Risks involved in reengineering projects. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE)*. IEEE, 1999.
- [Som96] Ian Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.
- [SP98] Perdita Stevens and Rob Pooley. System reengineering patterns. In *Proceedings of FSE-6*. ACM-SIGSOFT, 1998.

- [SRMK99] Reinhard Schauer, Sébastien Robitaille, Francois Martel, and Rudolf Keller. Hot-Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*. IEEE Computer Society Press, 1999.
- [SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2 — Networked and Concurrent Objects*. John Wiley and Sons, 2000.
- [SW98] Geri Schneider and Jason P. Winters. *Applying Use Cases*. Addison Wesley, 1998.
- [Tay00] Paul Taylor. Capable, productive, and satisfied: Some organizational patterns for protecting productive people. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design*, volume 4, pages 611–636. Addison Wesley, 2000.
- [Tho98] Rob Thomsett. The year 2000 bug: a forgotten lesson. *IEEE Software*, 15(4):91–93,95, July 1998.
- [WBW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.
- [WG98] Johannes Weidl and Harald Gall. Binding object models to source code: An approach to object-oriented rearchitecting. In *Proceedings of the 22nd Computer Software and Application Conference (COMPSAC 1998)*. IEEE Computer Society Press, 1998.
- [Woo98] Bobby Woolf. Null object. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, pages 5–18. Addison Wesley, 1998.

- [WTMS95] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, January 1995.
- [Wuy98] Roel Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.
- [You97] Edward Yourdon. *Death March*. Prentice-Hall, 1997.

찾아보기

- Ada, 44, 303, 312
Adapter (패턴), 231
ADAPTOR 패턴, 230, 235
AST 매칭, 282
Belady, Les, xv
C++, 44, 65, 145, 234, 254, 273, 302, 312, 315, 329, 340
COBOL, 254
Cobol, 145
CodeCrawler, 116, 128
DALI, 117
database, 97
DATRIX, 254
Deprecation, 237
DESEL 프로젝트, 56
Design Patterns, xviii
DESIRE, 117
dotplot, 257
Dup, 259
Duploc, 259
Eiffel, 139
Envy, 90
ESPRIT, xvi, xix
Extract 메서드 추출하기 (패턴), 270
FAMOOS, xvi, xix, 44, 65, 82, 89, 128, 145, 259
forces, pattern, forces 을 참고
hook 메서드, 183
IEEE, 13
Java, 7, 154, 234, 238, 274, 281, 286, 302, 312, 329
Javadoc, 63
JUnit, 172, 179, 181
Lehman, Manny, xv
MediaGeniX, 139
Migrating Legacy Systems, 215
pattern
language, xviii
Perl, 340
Python, 258
Rigi, 117
RT-100, 55
RTTI, 312
Smalltalk, 65, 90, 160, 195, 234, 273, 281, 303, 312, 331
SQL, 99, 101–103
Squeak, 139, 335, 337

- State (패턴), 352
- SUnit, 179
- tradeoffs, pattern, tradeoffs을 참고
- UML
 - 클래스 다이어그램, 98, 102, 118
- UNIX, 302
- Unix, 157
- XP, 익스트림 프로그래밍을 참고
- 가구 배치, 145
- 가장 가치 있는 것 먼저 (패턴), 25
- 가장 가치 있는 것 먼저 하기 (패턴), 33, 37, 201, 206, 215
- 가장 가치 있는 것을 먼저 (패턴), 27, 29
- 간단하게 유지 하기 (패턴), 41
- 게시된 인터페이스, 233, 235
- 경량 (패턴), 114, 350
- 계약 찾기 (패턴), 102, 106, 353
- 계획 게임, 207
- 계획 세우기 게임, 34
- 고객과 관계맺기, 205
- 고장이 나지 않으면 수정하지 않기 (패턴), 25, 38, 39
- 골드버그, 아델, 27, 81
- 공개 인터페이스와 게시된 인터페이스
 - 구분하기 (패턴), 231, 235
- 과거로부터 배우기 (패턴), 132, 133, 155, 157
- 과거로부터 학습 (패턴), 34
- 구현이 아닌 인터페이스 테스트하기 (패턴), 133, 161, 168, 174, 176–178, 187, 190, 215
- 기계적으로 코드 비교 (패턴), 288
- 기계적으로 코드 비교하기 (패턴), 246, 249, 288
- 기능 중복, 12
- 기본 테스트를 증가시키기 (패턴), 167, 174, 175, 196, 215, 224, 346, 347
- 나선 개발 수명주기, 9
- 내비게이터 지정하기 (패턴), 25, 29, 180, 215
- 널 객체 (패턴), 299, 300, 329, 350
- 널 객체 도입하기 (패턴), 298, 300, 319, 329, 351
- 넬슨, 제이콥, 81
- 노키아, xvi, xix
- 노텔(Nortel), 55
- 뉴 타운으로 가는 다리(Bridge to the New Town), 225
- 뉴 타운으로 가는 브리지 만들기 (패턴), 203, 211, 215, 225
- 뉴타운으로 가는 다리 만들기 (패턴), 219
- 다시 쓰기 규칙 편집기, 282
- 다임러 벤츠, xix
- 다형성 적용 조건문 변환 (패턴 클러스터), 298
- 다형성 적용된 조건문 변환 (패턴 클러스터), 247, 261
- 다형성 적용한 조건문 변환 (패턴 클러스터), 19, 259
- 단계 별 실행해 보기 (패턴), 132, 133, 147, 149, 155
- 단계별 실행해 보기 (패턴), 149, 155
- 단순함 유지하기 (패턴), 25
- 단위 테스트, 60
- 더 블롭, 289
- 데모 중 인터뷰 (패턴), 88

- 데모 중 인터뷰하기 (패턴), 46, 47, 52, 56, 62, 66, 70, 74, 75, 83, 89, 90, 109, 111, 149, 161
- 데이터 서비스, 앤런, 173, 224, 244
- 데이터 가까이 동작 이동하기 (패턴), 247, 261, 348
- 데이터 브리지, 226
- 데이터 유지 — 코드 전달(Keep the Data — Toss the Code), 225
- 데이터 컨테이너, 263, 265
- 데이터베이스
- 스키마, 97
- 델라노, 데이비드, 345
- 도트 플롯으로 코드 시각화하기 (패턴), 246, 255, 348
- 동작을 데이터 가까이 이동하기 (패턴), 266, 269, 270, 277–279, 282, 291, 348, 349, 353
- 동작하게 하고, 제대로 동작하게 하고, 빠르게 동작하게 하기, 244
- 디마르코, 톰, 54, 127
- 디미터 법칙, 288
- 디미터의 법칙, 264, 265, 279
- 디버거, 147
- 디자인 드리프트(design drift), xvii
- 디자인 추측하기 (패턴), 93, 94, 100, 102, 106, 109, 116, 159, 350
- 디자인 패턴, 345
- 디자인에 대한 추측 (패턴), 94
- 디자인에 대해 추측 (패턴), 83
- 디자인에 대해 추측하기 (패턴), 74
- 디테일 모델 캡처 (패턴 클러스터), 95
- 디테일한 모델 캡처 (패턴 클러스터), 18, 129, 130
- 라운드 테이블에 말하기 (패턴), 25
- 라이징, 린다, 345
- 러브, 톰, 219
- 레거시 소프트웨어, 3
- 레이어링
- 부적절, 12
- 로버츠, 도널드, 345
- 로버트, 돈, 144
- 루가버, 스펜서, 55
- 루빈, 케니, 27, 81
- 리먼, 매니, 242
- 리버스 엔지니어링
- 정의, 9
- 리버스 엔지니어링(reverse engineering), 11
- 리스터링처링
- 정의, 13
- 리엔지니어링
- 정의, 9
 - 지속적, 8
 - 지속적인, 14
- 리엔지니어링 패턴, 14, 15
- 형식, 16
- 리엘, 아서, 105, 277, 294
- 리팩터링
- 정의, 13
- 리팩토링 브라우저, 65, 144, 282
- マイ그레이션 전략 (패턴 클러스터), 18, 168
- 맥심, 27
- 맥심에 동의하기 (패턴), 25, 27, 47
- 메서드 이동하기 (패턴), 270, 348
- 메서드 이름 바꾸기 (패턴), 142
- 메서드 추출하기 (패턴), 142, 144, 323, 326, 347
- 메소드 이름 바꾸기 (패턴), 348
- 메시드 추출하기 (패턴), 246, 261
- 메트릭, 119, 158

- 모듈화
 부족, 12
- 모의 설치 수행하기 (패턴), 56, 66, 78, 85, 87, 90
- 모의 설치 하기 (패턴), 52
- 목표 솔루션 프로토타입 만들기 (패턴), 211, 242
- 목표 솔루션 프로토타입하기 (패턴), 203, 214
- 목표 솔루션 프로토타입하기n (패턴), 217
- 문서, 67
 부족, 12
 폐기, 5
- 문서 스키밍하기 (패턴), 46, 51, 52, 56, 62, 66, 67, 74, 78, 83, 109, 110
- 문학적 프로그래밍(literate programming), 138
- 바인더, 로버트, 345
- 방향 설정 (패턴 클러스터), 18, 24, 201
- 버그 리포트, 50
- 버려질 프로토타입, 218
- 버릴 임시 코드(Throwaway Code), 219
- 베넷, 사이몬, 81
- 벡, 켄트, 7, 31, 65, 245
- 벨레디, 레스, 242
- 벽화 시작화, 257
- 변경할 때마다 회귀 테스트하기 (패턴), 168, 203, 215, 222–224, 291, 346
- 복잡성 증가의 법칙(Law of Increasing Complexity), 6
- 복잡성 증가의 법칙(The Law of Increasing Complexity), xvi
- 빔, 배리, 9
- 브랜트, 존, 144
- 브로디, 마이클, 227
- 브룩스, 프레드릭, 129, 219
- 블라하, 마이클, 105
- 블랙박스 테스트, 176, 190
- 비즈니스 규칙을 테스트로 기록 (패턴), 83, 168
- 비즈니스 규칙을 테스트로 기록하기 (패턴), 133, 148, 176, 178, 190, 193, 196
- 비즈니스 모델, 27
- 비지터 (패턴), 273, 353
- 빈더, 로버트, 165
- 사용자 참여 시키기 (패턴), 41
- 사용자 참여시키기 (패턴), 25, 35, 47, 202, 203, 205, 207, 211, 242
- 상세 모델 캡춰 (패턴 클러스터), 118, 128
- 상세 모델 캡춰하기 (패턴 클러스터), 131
- 상속
 누락, 12
 오용, 12
- 상태 (패턴), 298, 299, 321, 322, 351
- 상태 추출하기 (패턴), 298–300, 309, 321, 328, 352
- 상태 패턴 (패턴 랭귀지), 352
- 상태 패턴 (패턴 언어), 322
- 새로운 마을로 다리 만들기 (패턴), 37
- 세마 그룹, xix
- 소규모 릴리즈, 210
- 소프트웨어 유지보수, 13
 정의, 13
- 소프트웨어 재사용, 8
- 속성 이름 바꾸기 (패턴), 142, 348
- 수량 (패턴), 113, 351
- 스니드, 해리, 145
- 스톤브레이커, 마이클, 227
- 스티븐스, 퍼디타, 239

- 시맨틱 래퍼, 230, 289
 시맨틱 래퍼(Semantic Wrapper), 229
 시스템 점진적 마이그레이션 (패턴), 202
 시스템 점진적 마이그레이션 하기 (패턴), 207
 시스템 점진적 마이그레이션하기 (패턴), 35, 168, 174, 202, 203, 206, 213, 228, 242
 신 클래스, 289
 신 클래스, 263, 265
 신 클래스 분할하기 (패턴), 37, 266, 289, 293, 350
 신뢰 구축 하기 (패턴), 41
 신뢰 구축하기 (패턴), 35
 실행 벼전 항상 보유하기 (패턴), 203, 215, 221–223
 싱글턴 (패턴), 351
 싱글톤 (패턴), 61
 아키텍처, 29, 33, 111
 아키텍처(architecture), 23
 알렉산더, 크리스토퍼, 14
 양탄자 밑으로 쓸어 넣기(Sweeping it Under the Rug), 229
 어댑터 (패턴), 273, 349
 어뎁터 (패턴), 15
 예외적인 엔티티 연구하기 (패턴), 66, 93, 94, 118, 119
 오래된 버그 테스트하기 (패턴), 168, 176, 346
 오케러한, 앤런, 230
 올바른 인터페이스 제공하기 (패턴), 37
 올바른 인터페이스 제시하기 (패턴), 204, 214, 215, 219, 229, 231, 235, 294, 295, 349
 외래 키, 101
 요더, 조셉, 219, 294
 울프, 바비, 331
 원탁 회의에 말하기 (패턴), 31, 93
 윙, 캐니, 72
 유든, 에드워드, 206
 유지보수, 50
 유지보수자와 담소나누기 (패턴), 46, 47, 49, 62, 66, 78, 83, 90, 161
 유지보수자와 대화나누기 (패턴), 74
 이해관계자, 33, 37
 이해를 위한 리팩터링 (패턴), 16
 이해를 위해 리팩터링하기 (패턴), 143, 145, 199
 이해를 위해 테스트 작성하기 (패턴), 178
 이해하기 위해 리팩터링하기 (패턴), 132, 140–142, 144, 145, 155, 244, 348
 이해하기 위해 테스트 작성하기 (패턴), 132, 142, 143, 149, 168, 196, 197
 이해할 수 있는 테스트 작성하기 (패턴), 145
 익스트림 프로그래밍, 41, 173, 210
 익스트림 프로그래밍(Extreme Programming), 224
 인터뷰 중 데모하기 (패턴), 77
 일할 수 있는 가장 간단한 것을 하라, 41
 자신 타입 검사 변환하기 (패턴), 298
 자신 타입 체크 변환하기 (패턴), 299, 301, 306, 309, 311, 312, 321
 자신감 구축하기 (패턴), 202, 203, 206, 207, 209, 211, 214, 222, 223, 228
 자신감 만들기 (패턴), 25
 잘못 위치한 오퍼레이션, 12
 잭슨, 다니엘, 73
 전략 (패턴), 299, 300, 325, 328, 352

- 전략 추출하기 (패턴), 298, 299, 309, 325, 328, 352
- 조건문을 다형성으로 치환하기 (패턴), 319, 348
- 조건문을 등록으로 변환하기 (패턴), 298, 300, 309, 319, 333, 339, 340
- 조건을 다형성으로 변환 (패턴 클러스터), 128
- 중복 코드, 249
- 중복 코드 감지 (패턴 클러스터), 19, 246, 247
- 중복된 코드, 6
- 증상이 아닌 문제 수정 (패턴), 25, 29, 34, 35
- 증상이 아닌 문제 수정하기 (패턴), 37
- 지속적 변화의 법칙(The Law of Continuing Change), xv
- 지속적 통합(Continuous Integration), 222
- 지속적인 문제 재테스트 (패턴), 168
- 지속적인 문제 재테스트하기 (패턴), 224, 346
- 지원 중단(Deprecation), 237, 239
- 진화 활성화를 위한 테스트 작성하기 (패턴), 167, 169, 174, 189, 201, 203, 215, 224
- 진화적 프로토타입, 218
- 채핀, 존, 73
- 책임 재배포 (패턴 클러스터), 19, 128
- 첫 번째 접근 (패턴 클러스터), 18, 46, 91, 92, 98
- 첫 번째 컨택 (패턴 클러스터), 119, 129
- 초기 이해 (패턴 클러스터), 18, 48, 94, 129, 151, 157
- 최적화하기 전에 프로파일러 사용하기 (패턴), 204, 243, 244
- 추상 인스턴스 변수(Abstract Instance Variable), 347
- 추상 팩토리 (패턴), 153, 315, 349
- 치코프스키, 엘리엇, 9, 13
- 치킨 리틀, 213, 215
- 친숙도 보존의 법칙, 242
- 친숙도 보존하기 (패턴), 204, 206, 241, 242
- 캡슐화
- 위반, 12, 279
- 캡슐화 위반, 캡슐화, 위반을 참고
- 컨트랙트 찾기 (패턴), 132, 133, 149, 151, 155, 161, 349, 350
- 켈러, 올프강, 227
- 코니그스버그, 앤런, 64
- 코드 리뷰, 60
- 코드 스멜, 7, 60
- 코드 중복, 12
- 코드에 대한 질문 연결하기 (패턴), 145, 199
- 코드에 대한 질문을 연결하기 (패턴), 130, 135, 137, 140, 143, 145, 148
- 코딩 이디엄, 60
- 코플리언, 제임스, 207
- 콕번, 앤리스터, 116
- 콘웨이, 멜빈, 54
- 콘웨이의 법칙, 54
- 쿡, 스테芬, 56
- 크로스, 제임스, 9, 13
- 큰 진흙 뭉치, 294
- 큰 진흙 뭉치(Big Ball of Mud), 219
- 클라이언트 유형 검사 변환하기 (패턴), 340
- 클라이언트 타입 검사 변환하기 (패턴), 270, 298–300, 309, 311, 314,

- 319, 340, 349
- 클래스 남용, 12
- 탐색 코드 제거하기 (패턴), 266, 270, 279, 284, 286, 288, 347, 350
- 탐색용 프로토타입, 218
- 테스트
 - 누락, 5
 - 테스트 주도 개발(Test-Driven development), 224
 - 테스트 프레임워크 사용하기 (패턴), 167, 178, 179
 - 테스트라는 생명 보험 (패턴 클러스터), 61, 85, 106, 118, 132
 - 테스트라는 생명보험 (패턴 클러스터), 18
 - 테스팅, 165
 - 템플리트 메서드 (패턴), 153, 352
 - 템플렛 메서드 (패턴), 352
 - 톰캣, 룹, 54
 - 파사드 (패턴), 15, 284, 291, 349, 350
 - 파울러, 마틴, 13, 31, 245, 330, 345
- 패턴
 - 언어, 18
 - 패턴(pattern)
 - 주요한 요구사항(forces), xviii
 - 트레이드오프(tradeoffs), xviii
 - 팩토리 메서드 (패턴), 153, 350
 - 퍼블릭 인터페이스와 계시된 인터페이스 구분하기 (패턴), 204, 233
 - 퍼시스턴트 데이터 분석하기 (패턴), 66, 74, 93, 94, 97, 106, 351
 - 퍼지 기능 테스트하기 (패턴), 168, 176, 346
 - 폐기된 인터페이스 지원 중단하기 (패턴), 204, 231, 234, 235, 237, 272, 293
 - 포승스젠틀룸 인포매틱 카를스루에, xix
- 포워드 엔지니어링
 - 정의, 9
- 푸트, 브라이언, 219, 294
- 풀리, 룹, 239
- 프레스맨, 로저, 72
- 프레임워크, xvi
- 필드 캡슐화하기 (패턴), 278, 347
- 한 시간 안에 모든 코드 읽기 (패턴), 46, 51, 52, 56, 59, 65, 66, 70, 74, 78, 83, 90, 109, 351
- 항상 실행 버전 보유하기 (패턴), 15, 168, 174
- 화이트, 짐, 55
- 화이트박스 테스트, 190
- 회의(meeting), 31
- 후크 메서드(hook method), 153

