ISTQB Certified Tester

Foundation Level Syllabus

Version 4.0

한글 Korean 1.0

(사) KSTQB

Korean Software Testing Qualifications Board

International Software Testing Qualifications Board

저작권 고지 Copyright Notice

목 차

Acknowledgement

이 ISTQB Certified Tester Foundation Level(CTFL) 실러버스는 ISTQB(International Software Testing Qualifications Board)의 전문가 그룹에 의해 제작되고, 2023년 4월 21일 ISTQB 총회(General Assembly)를 거쳐 공식적으로 출시된 문서입니다.

ISTQB Foundation 실러버스 워킹그룹과 Agile 실러버스 워킹그룹은 해당 실러버스 제작 및 리뷰에 참여한 모든 멤버국 회원들과 테스팅 전문가에게 감사의 말을 전합니다.

이전 각 버전의 제작에 참가했던 저자들과 이번 버전의 테크니컬 리뷰에 참가하고 의견을 내어준 Dr. Stuart Reid, Patricia McQuaid, Leanne Howard님과 각 국가회원국 멤버들에게도 감사를 전합니다.

이 실러버스의 한글화 작업을 위해 국내에서도 많은 분들께서 시간과 노력을 투자해 주셨습니다.

ISTQB CTFL 실러버스의 번역, 전문가 소견, 제안과 검토에 참가해 주신 최영재, 송홍진, 권혜영, 유희진님과, 교차 검토 및 기술적인 편집을 지원해 주신 (사)KSTQB의 권선이, 천호정, 이순미님께도 감사의 말을 전합니다.

이 문서를 활용하면서 관련해 의견이 있으신 분들은 언제든 info@kstqb.org로 내용을 보내주시기 바랍니다. 이 실러버스는 내/외부 검토의견이나 수정사항이 일정정도 수집되는 시점에 이를 반영해 업데이트 될 수 있습니다.

감사합니다.

The KSTQB Team

0. 서론 Introduction

0.1. 실러버스(교수요목)의 목적 Purpose of this Syllabus

이 실러버스는 ISTQB Foundation Level에서 국제 소프트웨어 테스팅 자격 제도의 기초 뼈대를 형성하며, ISTQB는 다음과 같은 목적으로 이 실러버스를 제공한다:

1. 각 국가 지역별 자격위원회가 현지 언어로 번역하고 교육기관을 승인하도록 지원.

   각 국 자격위원회는 실러버스를 자국 언어 필요성에 맞춰 조정하거나, 현지 출판을 위한 참고자료를 추가할 수 있음.

2. 인증기관이 이 실러버스의 학습목표(LO: Learning Objectives)를 따르는 각국의 현지 언어로 된 시험문제를 제작하도록 지원.
3. 교육기관이 교재 및 수업관련 자료를 만들고 적절한 교수방법을 정하도록 지원.
4. 자격증 취득 희망자가 교육 과정의 일환, 또는 독립적으로 자격시험을 준비하도록 지원.
5. 국제 소프트웨어 또는 시스템 엔지니어링 커뮤니티가 관련 직업의 발전이나 서적 및 기사의 기초로 사용할 수 있도록 지원.

0.2. 소프트웨어 테스팅에서의 CTFL 자격증

ISTQB CTFL(Certified Tester Foundation Level) 자격은 소프트웨어 테스팅에 관련된 모든 사람을 대상으로 한다. 테스터, 테스트 분석가, 테스트 엔지니어, 테스트 컨설턴트, 테스트 관리자, 사용자 인수 테스터 및 개발자가 여기 포함된다.

이 CTFL 자격은 또한 제품 소유자, 프로젝트 관리자, 품질 관리자, 소프트웨어 개발 관리자, 비즈니스 분석가, IT 디렉터나 경영 컨설턴트 등 소프트웨어 테스팅에 대한 기본적인 이해를 원하는 모든 사람에게 적합하다. CTFL 자격증 소지자는 더 상위 수준의 소프트웨어 테스팅 자격증 취득의 자격을 가진다(예: ISTQB Advanced Level, ISTQB Expert Level 외 도메인 특화 스페셜리스트 모듈 등).

0.3. 테스터를 위한 커리어 패스 Career Path for Testers

ISTQB® 체계는 경력의 모든 단계에 있는 테스팅 전문가들을 지원하며, 지식의 폭과 깊이를 모두 제공한다. ISTQB® 위원회 인증 자격증을 취득한 개인은 Core Advanced Level(TA: 테스트 분석가, TTA테크니컬 테스트 분석가, TM테스트 매니저)과 이후 Expert Level(TM 테스트 관리 또는 ITP테스트 프로세스 개선)에

관심이 있을 수 있다. 애자일 환경에서 테스팅 실무기술을 개발하고자 하는 사람은 Agile Technical Tester 또는 Agile Test Leadership at Scale 인증을 고려할 수 있다. Specialist stream은 특정 테스트 접근법 및 테스트 활동(예: 테스트 자동화, AI 테스팅, 모델 기반 테스팅, 모바일 앱 테스팅)을 갖춘 분야, 특정 테스트 영역(예: 성능 테스트, 사용성 테스트, 인수 테스트, 보안 테스트)과 관련된 분야, 또는 특정 산업 분야(예: 자동차 또는 게임)에 대한 테스팅 지식에 집중하는 분야 등에 대한 심층적인 학습과 연구를 제공한다. ISTQB의 인증 테스터 체계에 대한 최신 정보는 www.istqb.org에서 확인 가능하다.

0.4. 비즈니스 성과 Business Outcomes

ISTQB Foundation Level을 취득하는 사람에게 기대할 수 있는 14가지 비즈니스 성과는 다음과 같다.
ISTQB 파운데이션 레벨 인증 테스터는 다음을 수행할 수 있습니다:

FL-BO1	테스팅이 무엇이며, 왜 유익한지 이해할 수 있다.
FL-BO2	소프트웨어 테스트의 기본 개념을 이해할 수 있다.
FL-BO3	테스팅 상황에 따라 구현해야 할 테스트 접근 방식과 활동을 식별할 수 있다.
FL-BO4	문서화 품질을 평가하고 개선할 수 있다.
FL-BO5	테스팅의 효과와 효율성을 높일 수 있다.
FL-BO6	테스트 프로세스를 소프트웨어 개발 라이프사이클에 맞출 수 있다.
FL-BO7	테스트 관리 원칙을 이해할 수 있다.
FL-BO8	명확하고 이해하기 쉬운 결함 보고서를 작성 및 전달할 수 있다.
FL-BO9	테스팅과 관련된 우선순위와 노력에 영향을 미치는 요인을 이해할 수 있다.
FL-BO10	교차 기능 팀(cross-functional team)의 일원으로 일할 수 있다.
FL-BO11	테스트 자동화와 관련된 리스크와 이점을 파악할 수 있다.
FL-BO12	테스팅에 필요한 필수 기술을 파악할 수 있다.
FL-BO13	테스팅에 대한 리스크의 영향을 이해할 수 있다.
FL-BO14	테스트 진행 상황 및 품질에 대해 효과적으로 보고할 수 있다.

0.5. 시험용 학습목표 및 인지수준

학습목표는 비즈니스 성과(Business Outcome)를 지원하며, CTFL 시험문제를 제작하는 데 사용된다.

일반적으로 실러버스의 모든 내용은 소개 및 부록 부분을 제외하고는 모두 K1 수준에서 테스트 가능하

다. 즉, 응시자는 여섯 개의 챕터(chapters)에서 언급한 키워드 또는 컨셉(개념)을 인식, 기억 또는 상기해서 문제를 풀게 된다.

특정 학습 목표의 지식 수준은 각 챕터의 시작 부분에 표시되며 다음과 같이 분류된다:

- K1 : 기억(remember)
- K2 : 이해(understand)
- K3 : 적용(apply)

학습 목표의 더 자세한 내용과 예제는 부록 A(영문 실러버스 Appendix A 참조)에 나와 있다.
각 챕터의 제목 바로 아래에 키워드로 나열된 모든 용어의 정의를 기억해야 한다(K1).
이는 학습 목표에 명시적으로 언급되지 않은 경우에도 마찬가지다.

0.6. Foundation Level 시험

CTFL 시험은 이 실러버스를 기반으로 한다. 문제를 풀기 위해선 이 실러버스의 한 부분 이상을 학습자료로 사용해야 할 수 있다. 서론과 부록을 제외한 실러버스의 모든 부분에서 문제 출제가 가능하며, 표준(standards), 관련 서적 및 다른 ISTQB 실러버스 등이 참고문헌으로 포함된다(7장). 그러나 내용은 그런 표준, 서적이나 다른 ISTQB 실러버스로부터 발췌/요약되어 이 실러버스에 소개돼 있는 내용 이상으로는 시험에 출제되지 않는다.

0.7. 인증 Accreditation

ISTQB 국가/지역별 자격위원회(ISTQB Member Boards)는 이 실러버스를 따르는 교육기관을 인증(accredit)할 수 있다.

교육기관은 인증을 수행하는 국가 자격위원회/기관으로부터 인증에 대한 가이드라인을 얻어야 한다. 국가 자격위원회에 의해 인증된 과정은 이 실러버스에 부합하는 것으로 인정되며 과정의 일부로 ISTQB 시험을 진행할 수 있다.

0.8. 표준(Standards) 관련 Handling of Standards

Foundation 실러버스에 참조된 표준(예: IEEE 또는 ISO 표준)이 있으며, 이러한 참조/참고 문서는 품질 특성에 관한 ISO 25010 참조에서와 같이, 프레임워크를 제공하거나, 독자가 원하는 경우 추가 정보의 출처를 제공하기 위한 것이다. 표준 문서는 시험 대상이 아니다. 표준에 대한 자세한 내용은 7 장을 참고.

0.9. 최신정보 유지 Staying Current

소프트웨어 산업은 빠르게 변화한다. 이러한 변화에 대처하고 이해관계자들이 관련 최신 정보를 접할 수 있도록 ISTQB 실무 그룹은 www.istqb.org 웹사이트에 지원 문서 및 표준 변경 사항을 참조할 수 있는 링크를 생성했다. 이 정보는 Foundation 실러버스에 따라 문제로 출제되지는 않는다.

0.10. 상세 수준 Level of Detail

실러버스의 상세 수준은 국제적으로 일관된 교육과정과 시험 진행을 가능케 한다. 이 목표 달성을 위해 실러버스는 아래와 같이 구성된다:

- Foundation Level 의 의도를 설명하는 일반 교육목표
- 학생들이 기억해야 하는 용어 목록
- 인지학습 결과로써 달성해야 하는 각 지식 영역에 대한 학습목표
- 수용/인용된 문헌과 표준 등의 출처에 대한 참조를 포함하는 주요 개념의 설명

실러버스 내용은 소프트웨어 테스팅의 전체 지식 영역에 대한 설명이 아니며, Foundation Level의 교육과정에서 다루는 상세 수준을 반영한다. 이 실러버스는 테스트 개념과, 적용하는 SDLC와 무관하게 모든 소프트웨어 프로젝트에 적용할 수 있는 테스트 개념과 기술에 중점을 둔다.

0.11 이 실러버스의 구성 How this Syllabus is Organized

문제는 6개의 장(Chapter) 내용에서 출제된다. 각 장의 최상위 제목에서는 한 장 당 시간을 지정한다. 시간 정보는 장 레벨 이하에서는 제공되지 않는다. 인증 교육과정의 경우, 실러버스는 최소한 1,135분(18시간 55분)의 강의가 필요하며, 다음과 같이 6개 장에 분산되어 있다:

- 1 장 : 180 분 - 테스팅의 기초
  - 테스팅 관련 기본 원칙, 테스팅이 필요한 이유 및 테스트 목표를 학습한다.
  - 테스트 프로세스, 주요 테스트 활동 및 테스트웨어를 이해한다.
  - 테스팅에 필요한 필수기술을 이해한다.

- 2 장 : 130 분 - 소프트웨어 개발 수명주기와 테스팅
  - 다양한 개발 접근방식에 테스팅이 어떻게 통합되는지 학습한다.
  - 테스트 우선접근 방식과 DevOps의 개념을 배운다.
  - 다양한 테스트 레벨, 테스트 유형 및 유지관리 테스팅에 대해 학습한다.

- 3 장 : 80 분 - 정적 테스팅
  - 정적 테스팅의 기본 사항, 피드백과 리뷰 프로세스에 대해 학습한다.

- 4 장 : 390 분 - 테스트 분석 및 설계
  - 다양한 소프트웨어 작업 산출물에서 테스트 사례를 도출하기 위한 블랙박스, 화이트박스 및 경험 기반 테스트의 적용 방법을 배운다.
  - 협업 기반 테스트 접근법에 대해 학습한다.

- 5 장 : 335 분 - 테스트 활동 관리
  - 일반적으로 테스트를 계획하는 방법과 테스트 노력을 추정하는 방법을 배운다.
  - 리스크가 테스트 범위에 어떤 영향을 가져올 수 있는지 알아본다.
  - 테스트 활동을 모니터링하고 제어하는 방법을 배운다.
  - 구성 관리가 테스트를 지원하는 방법을 배운다.
  - 명확하고 이해하기 쉽게 결함을 보고하는 방법을 배운다.

- 6 장 : 20 분 – 테스트 지원 도구
  - 도구를 분류하고, 테스트 자동화의 단점과 장점을 이해한다.

제 1 장 테스팅의 기초(Fundamentals of Testing) | 180분

용어

커버리지(coverage), 디버깅(debugging), 결함(defect), 오류(error), 장애(failure), 품질(quality), 품질 보증(quality assurance), 근본 원인(root cause), 테스트 분석(test analysis), 테스트 베이시스(test basis), 테스트 케이스(test case), 테스트 완료(test completion), 테스트 컨디션(test condition), 테스트 제어(test control), 테스트 데이터(test data), 테스트 설계(test design), 테스트 실행(test execution), 테스트 구현(test implementation), 테스트 모니터링(test monitoring), 테스트 대상(test object), 테스트 목적(test objective), 테스트 계획(test planning), 테스트 절차(test procedure), 테스트 결과(test result), 테스팅(testing), 테스트웨어(testware), 밸리데이션(validation), 베리피케이션(verification)

1장 학습 목표

1.1 테스팅이란 무엇인가?

FL-1.1.1 (K1) 일반적인 테스트 목적을 식별할 수 있다

FL-1.1.2 (K2) 테스팅과 디버깅을 구별할 수 있다

1.2 테스팅이 왜 필요한가?

FL-1.2.1 (K2) 테스팅이 필요한 이유를 예를 들어 설명할 수 있다

FL-1.2.2 (K1) 테스팅과 품질 보증의 관계를 상기할 수 있다

FL-1.2.3 (K2) 근본 원인, 오류, 결함, 장애를 구별할 수 있다

1.3 테스팅 원리

FL-1.3.1 (K2) 테스팅의 7 가지 원리를 설명할 수 있다

1.4 테스트 활동, 테스트웨어, 테스트 역할

FL-1.4.1 (K2) 다양한 테스트 활동과 업무를 요약할 수 있다

FL-1.4.2 (K2) 정황이 테스트 프로세스에 미치는 영향을 설명할 수 있다

FL-1.4.3 (K2) 테스트 활동을 지원하는 테스트웨어를 구분할 수 있다

FL-1.4.4 (K2) 추적성을 유지하는 것의 가치를 설명할 수 있다

FL-1.4.5 (K2) 테스팅에 참여하는 다양한 역할을 비교할 수 있다

1.5 테스팅 필수 기술 및 모범 사례

FL-1.5.1 (K2) 테스팅에 필요한 보편적인 기술의 예를 제시할 수 있다

FL-1.5.2 (K1) 전체 팀 접근법의 장점을 상기할 수 있다

FL-1.5.3 (K2) 독립적 테스팅의 장단점을 구분할 수 있다

1.1. 테스팅이란 무엇인가?

소프트웨어 시스템은 우리 생활의 많은 부분과 밀접하게 연관되어 있다. 소프트웨어를 사용하면서 기대와는 다른 동작을 경험한 사람도 많을 것이다. 올바르게 동작하지 않는 소프트웨어는 금전적, 시간적, 비즈니스 평판 손실은 물론, 심하게는 부상이나 사망에 이르기까지 다양한 문제를 일으킬 수 있다. 소프트웨어 테스팅은 소프트웨어 품질을 평가하고, 소프트웨어 사용 시 나타나는 장애의 위험을 줄여줄 수 있다.

소프트웨어 테스팅은 결함을 식별하고 소프트웨어 산출물의 품질을 평가하는 일련의 활동이다. 테스트의 대상이 되는 이런 산출물을 테스트 대상(test object)이라고 한다. 흔히 테스팅이 단지 소프트웨어를 실행하고 결과를 확인하는 테스트 수행(test execution)에 국한된다고 오해하는 경우가 많다. 그러나 소프트웨어 테스팅에는 다른 활동이 포함되며, 이 활동은 소프트웨어 개발수명주기(SDLC)에 따라서 달라진다(2장 참조).

테스팅에 대한 또 다른 오해는 테스팅이 전적으로 테스트 대상의 베리피케이션(verification)에만 초점을 맞춘다는 것이다. 테스팅은 시스템이 주어진 요구사항을 충족하는지 확인하는 베리피케이션을 포함하지만, 시스템이 운영 환경에서 사용자 또는 기타 이해관계자가 필요한 바를 만족하는지를 확인하는 밸리데이션(validation)도 포함된다.

테스팅은 동적 또는 정적일 수 있다. 동적 테스팅은 소프트웨어를 실행하지만, 정적 테스팅은 그렇지 않다. 정적 테스팅은 리뷰(3장 참조)와 정적 분석을 포함한다. 동적 테스팅은 테스트 케이스를 도출하기 위해 다양한 테스트 기법과 테스트 접근법을 사용한다(4장 참조).

테스팅은 기술적인 활동에 국한되지 않으며, 적절한 계획/관리/추정/모니터링/제어도 필요하다(5장 참조).

테스터는 도구를 사용하지만(6장 참조), 테스팅은 주로 테스터가 전문 지식을 갖추고 분석 기술을 사용해 비판적 사고와 시스템적 사고를 적용하는 지적 활동이라는 점을 기억해야 한다(Myers 2011, Roman 2018).

ISO/IEC/IEEE 29119-1 표준은 소프트웨어 테스팅 개념에 대한 추가 정보를 제공한다.

1.1.1. 테스트 목적

일반적인 테스트 목적은 다음과 같다.

- 요구사항, 사용자 스토리, 설계, 소스 코드 등 작업 산출물 평가
- 장애 유발 및 결함 식별
- 테스트 대상에 필요한 커버리지 보장
- 소프트웨어 품질 부족으로 인한 리스크 수준 완화
- 정의된 요구사항의 충족 여부를 확인하는 베리피케이션
- 테스트 대상의 계약, 법률, 규제 요구사항 준수 여부를 확인하는 베리피케이션
- 이해관계자가 정보에 입각한 결정을 내리는데 필요한 정보 제공
- 테스트 대상의 품질에 대한 자신감 획득

- 테스트 대상의 완성 여부와 이해관계자의 기대 충족 여부를 확인하는 밸리데이션

테스팅의 목적은 정황에 따라 달라질 수 있다. 정황은 테스트 대상인 작업 산출물, 테스트 레벨, 리스크, 사용하는 소프트웨어 개발수명주기(SDLC), 그리고 기업 구조, 경쟁사 구도, 시장 출시 시기 등의 비즈니스 정황을 포함한다.

1.1.2. 테스팅과 디버깅

테스팅과 디버깅은 별개의 활동이다. 테스팅은 소프트웨어 결함으로 인한 장애를 유발하거나(동적 테스팅) 테스트 대상에 있는 결함을 직접 식별한다(정적 테스팅).

동적 테스팅(4장 참조)이 장애를 유발했을 때 디버깅은 장애(결함)의 원인을 찾고, 그 원인을 분석하고 제거한다. 이때 일반적인 디버깅 프로세스는 다음과 같다:

- 장애 재현
- 분석(근본 원인 식별)
- 원인 해결

이후 확인 테스팅(confirmation testing)으로 문제가 제대로 수정됐는지 확인한다. 확인 테스팅은 처음 테스트를 수행한 사람이 다시 수행하는 것이 바람직하다. 수정 사항이 테스트 대상의 다른 부분에 장애를 일으키지 않았는지 확인하기 위해 리그레션 테스팅(regression testing)을 추가로 수행할 수 있다(확인 테스팅과 리그레션 테스팅에 대한 자세한 내용은 2.2.3 참조).

정적 테스팅으로 결함을 식별한 경우 디버깅은 결함을 제거하는 데 중점을 둔다. 정적 테스팅에서는 장애를 유발하지 않고 직접 결함을 식별하기 때문에 장애를 재현하고 분석할 필요가 없다(3장 참조).

1.2. 테스팅이 왜 필요한가?

품질 제어 활동의 일환으로 테스팅은 정해진 범위, 시간, 품질, 예산 내에서 합의된 목표를 달성하는 데 도움을 준다. 성공에 기여하는 테스팅이 테스트팀의 활동으로 국한된 것은 아니다. 모든 이해관계자는 자신이 가진 테스트 기술을 사용해 프로젝트 성공에 기여할 수 있다. 컴포넌트, 시스템, 관련 문서를 대상으로 한 테스팅은 소프트웨어 결함을 식별하는 데 도움이 된다.

1.2.1. 성공을 위한 테스팅의 기여도

테스팅은 결함을 식별하는 비용 효율적인 방법이다. 식별한 결함은(테스팅 활동이 아닌 디버깅을 통해) 제거할 수 있기 때문에 테스팅은 테스트 대상의 품질 향상에 간접적으로 기여하게 된다.

테스팅은 소프트웨어 개발수명주기(SDLC)의 여러 단계에서 테스트 대상의 품질을 직접 평가하는 방법을

제공한다. 이런 평가결과는 대규모 프로젝트 관리 활동에서 릴리스 여부의 판단과 같은, 소프트웨어 개발수명주기(SDLC) 다음 단계로의 이동여부 결정에 기여하게 된다.

테스팅은 개발 프로젝트에서 사용자를 간접적으로 대변한다. 테스터는 개발수명주기 전반에 걸쳐 그들이 이해하고 있는 사용자 요구사항을 고려한다. 사용자 대표 집단이 개발 프로젝트에 참여하게 할 수도 있지만, 높은 비용과 적합한 사용자의 가용성 부족으로 쉽지 않은 경우가 많다.

계약 또는 법적 요구사항을 충족하거나 규제 표준 준수를 위해 테스팅이 필요할 수 있다.

1.2.2. 테스팅과 품질 보증(QA)

품질 보증(QA, Quality Assurance)과 테스팅이라는 용어를 혼용해서 사용하는 경우가 많지만, 품질 보증과 테스팅은 다르다. 테스팅은 품질 제어(QC, Quality Control) 활동에 속한다.

품질 제어는 적합한 수준의 품질 달성을 지원하는 활동에 초점을 맞춘 제품 중심의 교정 접근법이다. 테스팅은 품질 제어의 주요 활동이며, 정형 기법(모델 확인 및 정확성의 증명), 시뮬레이션, 프로토타이핑도 품질 제어에 속한다.

품질 보증은 프로세스의 구현과 개선에 초점을 맞춘 프로세스 중심의 예방 접근법이다. 좋은 프로세스를 올바르게 준수하면 좋은 제품이 만들어진다는 가정을 기반으로 한다. 품질 보증은 개발 및 테스팅 프로세스 모두에 적용되며, 프로젝트에 참여하는 모두의 책임이다.

테스트 결과는 품질 보증과 품질 제어 모두에 사용된다. 품질 제어는 결함을 수정하는 데 사용하며, 품질 보증은 개발 및 테스트 프로세스가 잘 동작하고 있는지 확인하기 위한 피드백으로 사용한다.

1.2.3. 오류, 결함, 장애, 근본 원인

사람은 오류(실수)를 저지르며 이에 따라 결함(결점, 버그)이 발생하고, 이는 결국 장애로 이어질 수 있다. 사람은 시간적인 압박, 작업 산출물의 복잡성, 프로세스, 인프라, 상호작용 등 다양한 이유로, 또 단순히 피로하거나 충분한 훈련이 부족해서 오류를 범할 수 있다.

결함은 요구사항 명세서나 테스트 스크립트와 같은 문서, 소스 코드, 빌드 파일과 같은 지원 산출물(supporting artifact)에서 나올 수 있다. 소프트웨어 개발수명주기(SDLC) 초기에 만든 산출물의 결함을 발견하지 못했을 때 이후 결함이 있는 산출물로 이어지는 경우가 많다. 소스 코드에 있는 결함이 실행되면 시스템이 수행해야 할 작업을 수행하지 못하거나, 수행하지 않아야 할 작업을 수행해 장애를 일으킬 수 있다. 실행될 경우 항상 장애를 일으키는 결함도 있지만, 특정 상황에서만 장애를 일으키거나 장애로 이어지지 않는 결함도 있다.

장애의 원인이 오류와 결함만 있는 것은 아니다. 방사선이나 전자기장으로 인한 펌웨어 결함 때문에 발생하는 장애처럼 환경 조건으로 발생하는 것도 있다.

근본 원인은 문제 발생의 근본적인 이유(예: 오류로 이어지는 상황)를 말한다. 근본 원인은 근본 원인 분

석(root cause analysis)을 통해 식별한다. 근본 원인 분석은 보통 장애가 발생하거나 결함을 식별한 경우 수행한다. 근본 원인을 처리(예를 들어 제거)하면 유사한 장애나 결함을 예방하거나 그 빈도를 줄일 수 있다.

1.3. 테스팅의 원리

지난 몇 년간 모든 테스팅에 적용할 수 있는 포괄적 지침을 제공하는 다양한 테스팅 원리가 제안돼 왔다. 이 실러버스는 그 중 일곱(7)가지를 소개하고 있다.

1. **테스팅은 결함의 존재를 밝히는 활동이지, 결함이 없음을 증명하지는 않는다.** 테스팅은 테스트 대상에 결함이 있음을 보여줄 수 있지만, 결함이 없다는 것을 증명할 수는 없다(Buxton 1970). 테스팅으로 테스트 대상에 결함이 잔존해 있을 확률을 줄일 수 있지만, 결함이 전혀 발견되지 않았다 하더라도 그 소프트웨어가 완벽하다는 뜻은 아니다.
2. **완벽한 테스팅은 불가능하다.** 모든 것을 테스팅한다는 것은 매우 간단한 소프트웨어를 제외하고는 불가능하다(Manna 1978). 따라서 완벽하게 테스트하려고 하기보다 테스트 기법(4 장 참조), 테스트 케이스 우선순위 지정(5.1.5 참조), 리스크 기반 테스팅(5.2 참조)을 사용해 테스트 노력을 집중해야 한다.
3. **조기 테스팅으로 시간과 비용을 절약할 수 있다.** 프로세스 초기에 발견해 제거한 결함은 연관된 다른 산출물의 후속 결함으로 이어지지 않는다. 소프트웨어 개발수명주기(SDLC) 후반에 발생하는 장애가 줄기 때문에 품질 비용이 절감된다(Boehm 1981). 결함을 조기에 식별하기 위해 정적 테스팅(3 장 참조)과 동적 테스팅(4 장 참조) 모두 최대한 이른 시점에 시작해야 한다.
4. **결함은 집중된다.** 보통 대부분의 결함은 소수의 시스템 컴포넌트에 집중되어 발생하는 경향을 보이며, 운영 장애의 대부분 역시 소수의 컴포넌트에서 발생한다(Enders 1975). 이런 현상은 파레토 원리(Pareto principle)의 예이다. 예상 결함 집중 영역과, 실제로 테스트나 운영 중 관측한 결함 집중 영역은 리스크 기반 테스팅의 주요 입력으로 사용된다(5.2 참조).
5. **테스트 효과는 줄어든다.** 만일 같은 테스트를 계속해서 반복하면, 결국 해당 테스트의 신규 결함 식별 효과는 점점 줄어들게 된다(Beizer 1990). 이런 현상을 극복하기 위해 기존 테스트와 테스트 데이터의 수정 및 새로운 테스트 작성이 필요할 수 있다. 그러나 자동 리그레션 테스팅처럼 같은 테스트를 반복하는 것이 유익한 결과로 이어지는 경우도 있다(2.2.3 참조).
6. **테스팅은 정황에 의존적이다.** 모든 상황에 적용할 수 있는 하나의 테스팅 접근법은 없다. 테스팅은 정황에 따라 다르게 진행한다(Kaner 2011).
7. **결함-부재는 궤변이다.** 소프트웨어 베리피케이션이 시스템의 성공을 보장할 것이라고 기대하는 것은 잘못된 생각이다. 정의한 모든 요구사항을 철저히 테스트하고, 발견한 모든 결함을 수정하더라도 사용자의 요구나 기대에 못 미치거나, 고객의 비즈니스 목표 달성에 도움이 되지 않고 경쟁 시스템에 비해 부족한 시스템이 만들어질 수도 있다. 따라서 베리피케이션과 함께 벨리데이션도 수행해야 한다(Boehm 1981).

1.4. 테스트 활동, 테스트웨어, 테스트 역할

테스팅은 정황에 의존적이지만, 상위 수준에서 봤을 때 만약 없다면 테스팅이 테스트 목적을 달성하기 어렵게 되는 보편적인 테스트 활동이 있다. 이런 테스트 활동이 테스트 프로세스(test process)를 구성하게 된다. 테스트 프로세스는 여러 요인을 기반으로 주어진 상황에 맞게 조정될 수 있다. 테스트 프로세스에 포함할 테스트 활동과 그러한 활동의 구현 방법과 수행 시기는 보통 해당하는 상황의 테스트 계획을 할 때 결정한다(5.1 참조).

이어지는 몇 개의 절은 테스트 활동 및 업무, 정황이 미치는 영향, 테스트웨어, 테스트 베이시스와 테스트웨어 간의 추적성, 테스팅 역할의 측면에서 일반적인 테스트 프로세스를 설명한다.

ISO/IEC/IEEE 29119-2 표준은 테스트 프로세스에 대한 추가 정보를 제공한다.

1.4.1. 테스트 활동과 업무

테스트 프로세스를 구성하는 주요 활동은 보통 다음과 같다. 이런 활동의 상당수는 순차적으로 수행되는 것처럼 보일 수 있으나, 실제로는 반복적 또는 병렬로 구현되는 경우가 많다. 테스팅 활동은 일반적으로 시스템과 프로젝트에 맞게 조정한다.

테스트 계획(Test planning)은 테스트 목적을 정의한 다음 전반적인 상황에 따른 제약 조건 내에서 목적을 가장 잘 달성할 수 있는 접근법을 선택하는 것이다. 테스트 계획은 5.1에서 자세히 설명하고 있다.

테스트 모니터링과 제어(Test monitoring and control). 테스트 모니터링은 지속적으로 모든 테스트 활동을 점검하고, 실제 진행 상황을 계획과 비교하는 활동이다. 테스트 제어는 테스트 목적을 달성하는 데 필요한 조치를 하는 활동이다. 테스트 모니터링과 제어는 5.3에서 상세히 다루고 있다.

테스트 분석(Test analysis)은 테스트 베이시스를 분석해 테스트 가능한 기능을 식별하고, 관련된 테스트 컨디션을 정의하고, 우선순위를 정하는 활동이 포함되며, 이와 관련된 리스크와 리스크 수준도 같이 고려된다(5.2 참조). 또한, 테스트 베이시스와 테스트 대상을 평가해 결함을 식별하고, 테스트 용이성을 평가한다. 테스트 분석을 지원하기 위해 테스트 기법을 사용하는 경우가 많다(4장 참조). 테스트 분석은 측정할 수 있는 커버리지 조건으로 "무엇을 테스트할 것인가?"라는 질문에 대한 답을 제공한다.

테스트 설계(Test design)는 테스트 컨디션을 테스트 케이스와 기타 테스트웨어(예: 테스트 차터)로 구체화하는 작업을 포함한다. 이 활동은 테스트 케이스 입력값을 구체화하는 데 도움이 되는 커버리지 항목의 식별을 포함하는 경우가 많다. 이 활동을 지원하기 위해 테스트 기법(4장 참조)을 활용할 수 있다. 테스트 설계는 테스트 데이터 요구사항 정의, 테스트 환경 설계, 기타 필요 인프라와 도구 식별도 포함한다. 테스트 설계는 "어떻게 테스트할 것인가?"라는 질문에 답을 제공한다.

테스트 구현(Test implementation)은 테스트 실행에 필요한 테스트웨어(예: 테스트 데이터)를 만들거나 획득하는 작업을 포함한다. 테스트 케이스는 테스트 절차(test procedure)로 묶을 수 있으며, 테스트 스위트(test suite)로 조합하는 경우도 많다. 수동 및 자동 테스트 스크립트를 만들게 된다. 효율적인 테스트

실행을 위해 우선순위를 반영한 테스트 실행 일정으로 테스트 절차를 정리한다(5.1.5 참조). 테스트 환경을 구축하고 올바르게 설정되었는지 확인한다.

테스트 실행(Test execution)은 테스트 실행 일정에 따라 테스트를 수행하는 것(테스트 런)을 포함한다. 테스트는 수동이나 자동으로 실행할 수 있다. 테스트 실행은 지속적 테스팅(continuous testing), 페어 테스팅 세션(pair testing sessions) 등 다양한 형태로 이루어질 수 있다. 실제 테스트 결과를 기대 결과와 비교한다. 테스트 결과는 기록된다. 이상 현상을 분석해 가능한 원인을 파악한다. 이런 분석을 통해 관찰한 장애를 기반으로 이상 현상을 보고할 수 있다(5.5 참조).

테스트 완료(Test completion) 활동은 일반적으로 프로젝트 마일스톤(예: 릴리스, 반복 주기 완료, 테스트 레벨 완료)에서 수행하며, 해결되지 않은 결함에 대해서는 변경 요청서 또는 제품 백로그 항목을 만든다. 향후 유용할 수 있는 테스트웨어를 식별해서 보관하거나 적절한 팀에 인계한다. 테스트 환경은 합의된 상태로 종료하게 된다. 테스트 활동을 분석해 향후 반복 주기, 릴리스, 프로젝트를 위한 교훈과 개선 사항을 파악한다(2.1.6 참조). 테스트 완료 보고서를 작성해 이해관계자에게 전달한다.

1.4.2. 정황에 따른 테스트 프로세스

테스팅은 단독으로 수행되지 않는다. 테스트 활동은 조직에서 수행하는 개발 프로세스의 필수적인 부분이다. 테스팅 비용은 이해관계자가 부담하게 되며, 테스팅의 최종 목표는 이해관계자의 비즈니스 목표 달성을 지원하는 것이다. 따라서 테스팅 수행 방식은 다음과 같은 여러 정황 요소에 따라 달라진다:

- 이해관계자(필요, 기대, 요구사항, 협력 의지 등)
- 팀원(기술, 지식, 경험 수준, 가용성, 훈련 필요성 등)
- 비즈니스 도메인(테스트 대상의 중요도, 식별된 리스크, 시장 요구사항, 구체적인 법적 규제 등)
- 기술적 요인(소프트웨어 유형, 제품 아키텍처, 사용된 기술 등)
- 프로젝트 제약 조건(범위, 시간, 예산, 자원 등)
- 조직적 요인(조직 구조, 기존 정책, 적용한 실천법 등)
- 소프트웨어 개발수명주기(SDLC)(공학적 실천법, 개발 방법론 등)
- 도구(가용성, 사용성, 규정 준수 등)

이런 요소는 테스트 전략, 적용된 테스트 기법, 테스트 자동화 수준, 필요 커버리지 수준, 테스트 문서 상세화 수준, 보고 등 많은 테스트 관련 문제에 영향을 미친다.

1.4.3. 테스트웨어

테스트웨어는 1.4.1에서 설명한 테스트 활동의 결과물로 만들어진다. 조직마다 테스트웨어를 생성/구체화/명명/구성/관리하는 방식에 상당한 차이가 있다. 적절한 형상관리(5.4 참조)는 작업 산출물의 일관성과 무결성을 보장한다. 다음은 작업 산출물의 일부를 나열한 목록이다:

- **테스트 계획 작업 산출물**은 다음을 포함한다: 테스트 계획, 테스트 일정, 리스크 관리 대장(risk register), 시작 및 완료 조건(5.1 참조). 리스크 관리 대장은 리스크 발생 가능성, 리스크 영향, 리스크 완화 정보가 들어있는 리스크 목록이다(5.2 참조). 테스트 일정, 리스크 관리 대장, 시작 및 완료 조건은 종종 테스트 계획서에 들어간다.
- **테스트 모니터링과 제어 작업 산출물**은 다음을 포함한다: 테스트 진행 상황 보고서(5.3.2 참조), 제어 지침 문서(5.3 참조), 리스크 정보(5.2 참조)
- **테스트 분석 작업 산출물**은 다음을 포함한다: (우선순위가 지정된) 테스트 컨디션(예: 인수 기준, 4.5.2 참조), (바로 수정하지 않았다면) 테스트 베이시스의 결함에 관한 결함 보고서
- **테스트 설계 작업 산출물**은 다음을 포함한다: (우선순위가 지정된) 테스트 케이스, 테스트 차터, 커버리지 항목, 테스트 데이터 요구사항, 테스트 환경 요구사항
- **테스트 구현 작업 산출물**은 다음을 포함한다: 테스트 절차, 자동 테스트 스크립트, 테스트 스위트, 테스트 데이터, 테스트 실행 일정, 테스트 환경 요소. 테스트 환경 요소의 예로는 스텁(stubs), 드라이버, 시뮬레이터, 서비스 가상화 등이 있다.
- **테스트 실행 작업 산출물**은 다음을 포함한다: 테스트 로그, 결함 보고서(5.5 참조)
- **테스트 완료 작업 산출물**은 다음을 포함한다: 테스트 완료 보고서(5.3.2 참조), 향후 프로젝트 또는 반복 주기 때 개선할 실천 항목, 문서로 기록한 교훈, 변경 요청서(예: 제품 백로그 항목)

1.4.4. 테스트 베이시스와 테스트웨어 간의 추적성

효과적인 테스트 모니터링과 제어를 구현하려면 테스트 프로세스 전반에 걸쳐 테스트 베이시스의 개별 요소, 개별 요소와 관련된 테스트웨어(예: 테스트 컨디션, 리스크, 테스트 케이스), 테스트 결과, 식별한 결함 간의 추적성을 구축하고 유지하는 것이 중요하다.

정확한 추적성은 커버리지 평가를 지원하며, 측정 가능한 커버리지 기준이 테스트 베이시스에 정의되어 있을 때 매우 유용하다. 커버리지 기준은 테스트 목적 달성 상태를 나타내는 활동을 촉진하는 핵심 성과 지표로 기능할 수 있다(1.1.1 참조). 예를 들어:

- 테스트 케이스에서 요구사항으로의 추적성을 통해 테스트 케이스가 요구사항을 커버하고 있는지 확인할 수 있다.
- 테스트 결과에서 리스크로의 추적성을 통해 테스트 대상의 잔존 리스크 수준을 평가할 수 있다.

좋은 추적성은 커버리지 평가 외에도 변경사항의 영향을 파악할 수 있게 하고, 테스트 감사를 용이하게 하며, IT 운영 및 관리(IT governance) 기준을 충족하는 데 도움이 된다. 좋은 추적성은 또한 테스트 진행 상황 및 완료 보고서에 테스트 베이시스 개별 요소의 상태를 명시하여 보고서를 더 쉽게 이해할 수 있게 한다. 이해관계자에게 테스팅의 기술적 측면을 이해하기 쉬운 방식으로 전달하는 데 도움이 되기도 한다. 추적성은 비즈니스 목표 대비 제품 품질, 프로세스 역량, 프로젝트 진행 상황 등을 평가할 수 있는 정보를 제공한다.

1.4.5. 테스팅에서의 역할

이 실러버스는 두 가지의 주요 테스팅 역할, 즉 테스트 관리 역할과 테스팅 역할을 다룬다. 이 두 역할에 할당되는 활동과 업무는 프로젝트 및 제품의 정황, 역할 담당자의 기술 수준, 조직 상황 등의 요소에 따라 달라진다.

테스트 관리 역할은 테스트 프로세스, 테스트팀 그리고 테스트 활동 리더십에 대한 전반적인 책임을 지는 것이다. 테스트 관리 역할의 주요 관심 영역은 테스트 계획, 테스트 모니터링과 제어, 테스트 완료 활동이다. 테스트 관리 역할의 수행 방식은 정황에 따라 달라진다. 예를 들어, 애자일 소프트웨어 개발에서 테스트 관리 업무의 일부를 애자일팀이 처리할 수 있다. 여러 팀 또는 조직 전체의 협업이 필요한 업무는 개발팀 외부의 테스트 관리자가 수행할 수도 있다.

테스팅 역할은 테스팅의 공학(기술)적인 측면에 대한 전반적인 책임을 진다. 테스팅 역할은 주로 테스트 분석, 테스트 설계, 테스트 구현, 테스트 실행 활동에 초점을 둔다.

시기에 따라 역할을 수행하는 사람이 달라질 수 있다. 예를 들어, 테스트 관리 역할은 팀 리더, 테스트 관리자, 개발 관리자 등이 수행할 수 있다. 또한, 한 사람이 테스팅과 테스트 관리 역할을 동시에 수행하는 경우도 있다.

1.5. 테스팅의 필수 기술 및 모범 사례

기술(skill)이란 어떤 일을 잘 해내는 능력으로, 그 사람이 가진 지식, 경험, 적성에서 비롯된다. 우수한 테스터는 업무를 잘 수행하기 위한 몇 가지 필수적인 기술을 갖추어야 한다. 우수한 테스터는 팀플레이, 즉 협업에 능한 사람이어야 하며, 다양한 수준의 독립성으로 테스팅을 수행할 수 있어야 한다.

1.5.1. 테스팅에 보편적으로 필요한 기술

보편적인 것들이지만, 다음 기술은 테스터에게 특히 요구되는 것들이다:

- 테스팅 지식(테스팅 효과를 높이기 위해, 예: 테스트 기법 활용)
- 철저함, 신중함, 호기심, 세부사항에 대한 주의력, 체계적인 접근(결함, 특히 찾기 어려운 결함 식별을 위해)
- 우수한 의사소통 기술, 경청하는 자세, 팀플레이(모든 이해관계자와 효과적으로 상호작용하고, 다른 사람에게 정보를 전달하고, 상대의 이해를 구하고, 결함을 보고하고, 논의하기 위해)
- 분석적 사고, 비판적 사고, 창의성(테스팅 효과를 높이기 위해)
- 기술 지식(테스팅 효과를 높이기 위해, 예: 적절한 테스트 도구 사용)
- 도메인 지식(최종 사용자/비즈니스 대표자를 이해하고 그들과의 소통을 위해)

테스터는 좋지 않은 소식을 전해야 하는 경우가 많다. 나쁜 소식일 경우 그것을 전하는 사람을 탓하는

것은 인간이 가지고 있는 기본 성향 중 하나이다. 따라서 테스터에게 의사소통 기술은 매우 중요하다. 테스트 결과의 전달을 제품이나 작성자에 대한 비판으로 오해할 소지가 있다. 확증 편향(confirmation bias)은 현재 가지고 있는 믿음과 맞지 않는 정보를 받아들이기 어렵게 만든다. 테스팅이 프로젝트 성공과 제품 품질에 상당히 기여함에도 불구하고, 테스팅을 파괴적인 활동으로 간주하는 사람도 있다. 이런 인식을 개선하려면 결함과 장애관련 정보를 건설적인 방법으로 전달해야 한다.

1.5.2. 전체 팀 접근법

테스터에게 중요한 기술 중 하나는 팀 환경에서 효과적으로 일하고, 팀 목표에 긍정적으로 기여하는 능력이다. 익스트림 프로그래밍(2.1 참조)에서 시작된 전체 팀 접근법(Whole Team Approach)은 이런 능력을 기반으로 한다.

전체 팀 접근법에서는 필요 지식과 기술을 갖춘 팀원이라면 누구나 모든 작업을 수행할 수 있고, 모든 팀원이 품질에 대해 책임진다. 같은 공간을 사용하는 것(co-location)은 의사소통과 상호작용을 용이하게 하므로 팀원들은 (물리적 또는 가상의) 작업 공간을 공유한다. 전체 팀 접근법은 팀의 활력을 높이고, 팀 내 의사소통과 협업을 강화하며, 프로젝트의 성공을 위해 팀이 가진 다양한 기술을 활용해 시너지를 창출한다.

테스터는 필요한 수준의 품질 달성을 위해 다른 팀원과 긴밀히 협력하게 된다. 여기에는 비즈니스 담당자와 협력해 적절한 인수 테스트를 작성하고, 개발자와 협력해 테스트 전략을 협의하고, 테스트 자동화 접근법을 결정하는 것도 포함된다. 따라서 테스터는 테스팅 지식을 다른 팀원과 공유하게 되며, 제품 개발에 영향을 주게 된다.

정황에 따라 전체 팀 접근법이 적절하지 않을 수도 있다. 예를 들어, 안전이 치명적인 경우에는 높은 수준의 테스트 독립성이 필요할 수 있다.

1.5.3. 테스팅의 독립성

일정 수준의 독립성은 저자와 테스터의 인지 편향(cognitive biases) 차이로 테스터가 결함을 더 효과적으로 식별할 수 있도록 한다(Salman 1995). 그러나 독립성이 친숙함을 대체하는 것은 아니다. 예를 들어, 개발자는 자신이 작성한 코드에서 많은 결함을 효율적으로 찾아낼 수 있다.

작업 산출물은 저자가 직접 테스트하거나(독립성 없음), 저자의 같은 팀 동료가 테스트하거나(일정 수준의 독립성), 저자의 팀 외부에 있지만 조직 내에 있는 테스터가 테스트하거나(높은 독립성), 조직 외부의 테스터가 테스트할 수 있다(매우 높은 독립성). 대부분의 프로젝트는 여러 수준의 독립성으로 테스팅을 수행하는 것이 가장 좋다(예: 개발자가 컴포넌트와 컴포넌트 통합 테스팅을 수행하고, 시스템과 시스템 통합 테스팅은 테스트팀이 수행하고, 비즈니스 담당자가 인수 테스팅을 수행).

독립적인 테스팅의 주요 이점은 배경, 기술적 관점, 편향이 다르기 때문에 독립적인 테스터가 개발자와 다른 유형의 장애와 결함을 식별할 가능성이 높다는 점이다. 또한, 독립적인 테스터는 시스템 명세를 작

성하고 구현하는 과정에서 이해관계자의 가정을 검증하고, 그것에 대한 이의를 제기하거나 반증할 수 있다.

하지만 단점도 있다. 독립적인 테스터는 개발팀과 차단되어 협업과 의사소통이 어려울 수 있으며, 개발팀과 적대적인 관계로 이어질 수 있다. 개발자가 품질에 대한 책임감을 잃을 수도 있다. 독립적인 테스터가 병목으로 인식되거나, 출시 지연의 원인으로 비난 받을 수도 있다.

제 2 장 소프트웨어 개발수명주기(SDLC)와 테스팅 | 130분

용어

인수 테스팅(acceptance testing), 블랙박스 테스팅(black-box testing), 컴포넌트 통합 테스팅(component integration testing), 컴포넌트 테스팅(component testing), 확인 테스팅(confirmation testing), 기능 테스팅(functional testing), 통합 테스팅(integration testing), 유지보수 테스팅(maintenance testing), 비기능 테스팅(non-functional testing), 리그레션 테스팅(regression testing), 시프트-레프트(shift-left), 시스템 통합 테스팅(system integration testing), 시스템 테스팅(system testing), 테스트 레벨(test level), 테스트 대상(test object), 테스트 유형(test type), 화이트 박스 테스팅(white-box testing)

2장 학습 목표

2.1 소프트웨어 개발수명주기(SDLC)에서의 테스팅

FL-2.1.1 (K2) 소프트웨어 개발수명주기(SDLC)가 또는 선택된 소프트웨어 개발수명주기가 테스팅에 미치는 영향을 설명할 수 있다

FL-2.1.2 (K1) 모든 소프트웨어 개발수명주기(SDLC)에 적용되는 좋은 테스팅 프랙티스를 상기할 수 있다

FL-2.1.3 (K1) 개발에서 테스트 우선 접근법의 예를 들 수 있다

FL-2.1.4 (K2) DevOps 가 테스팅에 미치는 영향을 요약할 수 있다

FL-2.1.5 (K2) 시프트-레프트 접근법을 설명할 수 있다

FL-2.1.6 (K2) 프로세스 개선을 위한 방법으로 회고의 사용을 설명할 수 있다

2.2 테스트 레벨과 테스트 유형

FL-2.2.1 (K2) 테스트 레벨을 구별할 수 있다

FL-2.2.2 (K2) 테스트 유형을 구별할 수 있다

FL-2.2.3 (K2) 확인 테스팅을 리그레션 테스팅과 구별할 수 있다

2.3 유지보수 테스팅

FL-2.3.1 (K2) 유지보수 테스팅과 유발요인을 요약할 수 있다

2.1. 소프트웨어 개발수명주기(SDLC)에서의 테스팅

소프트웨어 개발수명주기(SDLC) 모델은 상위 수준에서 소프트웨어 개발 프로세스를 추상화해서 표현한 것이다. 소프트웨어 개발수명주기(SDLC) 모델은 개발 프로세스의 여러 단계와 활동 유형이 논리적, 시간상으로 서로 어떻게 연관되는지 정의한다. 소프트웨어 개발수명주기(SDLC) 모델의 예로 순차적 개발 모델(예: 폭포수 모델, V-모델), 반복적 개발 모델(예: 나선형 모델, 프로토타이핑), 점진적 개발 모델(예: 통합 프로세스) 등이 있다.

소프트웨어 개발 프로세스 내 일부 활동을 구체적인 소프트웨어 개발 방법과 애자일 실천법으로 설명하기도 한다. 예를 들면, 인수 테스트 주도 개발(ATDD), 행위 주도 개발(BDD), 도메인 주도 설계(DDD), 익스트림 프로그래밍(XP), 기능 주도 개발(FDD), 칸반(Kanban), 린(lean) IT, 스크럼(Scrum), 테스트 주도 개발(TDD) 등이 여기에 해당한다.

2.1.1. 소프트웨어 개발수명주기(SDLC)가 테스팅에 미치는 영향

테스팅의 성공을 위해 소프트웨어 개발수명주기(SDLC)에 맞는 조정이 필요하다. 소프트웨어 개발수명주기(SDLC) 모델의 선택은 다음에 영향을 미친다:

- 테스트 활동 범위 및 시기(예: 테스트 레벨 및 테스트 유형)
- 테스트 문서 상세화 수준
- 테스트 기법 및 테스트 접근법 선택
- 테스트 자동화 범위
- 테스터의 역할과 책임

일반적으로 순차적 개발 모델에서 테스터는 초기 단계에서 요구사항 리뷰, 테스트 분석과 설계에 참여한다. 실행 가능한 코드는 보통 개발 후반에 생성되므로, 동적 테스팅은 소프트웨어 개발수명주기(SDLC) 초기에 수행하기 어려운 경우가 많다.

일부 반복적 점진적 개발 모델은 반복 주기마다 동작하는 프로토타입이나 제품 증분이 만들어진다고 가정한다. 이는 반복 주기마다 모든 테스트 레벨에서 정적 테스팅과 동적 테스팅을 수행할 수 있음을 의미한다. 증분을 자주 전달하려면 빠른 피드백과 광범위한 리그레션 테스팅이 필요하다.

애자일 소프트웨어 개발에서는 프로젝트의 어느 시점에도 변화가 생길 수 있다고 가정한다. 따라서 애자일 프로젝트는 리그레션 테스팅을 수월하게 하는 가벼운 작업 산출물과 테스트 자동화를 선호하게 된다. 또한, 수동 테스팅도 사전 테스트 분석과 설계가 필요하지 않은, 경험 기반 테스트 기법(4.4 참조)으로 진행하는 경향이 있다.

2.1.2. 소프트웨어 개발수명주기(SDLC)와 우수한 테스팅 프랙티스

선택한 소프트웨어 개발수명주기(SDLC) 모델에 상관없이, 다음과 같은 우수한 테스팅 프랙티스가 있다:

- 모든 소프트웨어 개발 활동에 상응하는 테스트 활동을 두어, 모든 개발 활동이 품질 제어의 대상이 되게 한다.
- 테스트 레벨(2.2.1 참조)마다 구체적이면서 독립적인 테스트 목적을 설정해, 중복은 피하고, 적절하면서 포괄적인 테스팅이 가능하게 한다.
- 특정 테스트 레벨을 위한 테스트 분석과 설계를 소프트웨어 개발수명주기(SDLC)의 상응하는 각 개발 단계에서 시작해, (테스팅 원리 중) 조기 테스팅(1.3 참조) 원칙을 준수할 수 있게 한다.
- 테스터가 문서 초안이 가용한 즉시 작업 산출물 리뷰에 참여하도록 해서, 시프트-레프트 전략(2.1.5 참조) 지원을 위한 조기 테스팅과 결함 발견이 가능하도록 한다.

2.1.3. 소프트웨어 개발 주도를 위한 테스팅

테스트 주도 개발, 인수 테스트 주도 개발, 행위 주도 개발은 서로 유사한 개발 접근법으로 개발 방향 결정을 위한 수단으로 테스트를 정의한다. 이런 접근법은 코드 작성 전에 테스트를 정의하므로, 조기 테스팅 원리(1.3 참조)를 구현하고 시프트-레프트 접근법(2.1.5 참조)을 따르게 한다. 반복적 개발 모델을 지원하며, 다음과 같은 특징을 가진다:

테스트 주도 개발(TDD):

- (광범위한 소프트웨어 설계 대신) 테스트 케이스를 통해 코딩 주도(Beck 2003)
- 테스트를 먼저 작성하고, 이를 충족하도록 코드를 작성한 다음, 테스트와 코드를 리팩토링(refactoring)

인수 테스트 주도 개발(ATDD) (4.5.3 참조)

- 시스템 설계 프로세스 중 인수 조건에서 테스트 도출(Gärtner 2011)
- 테스트는 해당 테스트를 만족해야 할 애플리케이션 영역을 개발하기 전에 작성

행위 주도 개발(BDD):

- 애플리케이션의 기대 동작을 이해관계자가 이해하기 쉽도록, 간단한 자연어로 작성해 테스트 케이스로 표현 - 일반적으로 Given/When/Then 형태를 사용(Chelimsky 2010)
- 이후 테스트 케이스는 자동으로 실행 가능한 테스트로 변환

위의 모든 접근법에서 테스트는 향후 적용/리팩토링 시 코드 품질 보장을 위해 자동화 테스트로 유지 가능하다.

2.1.4. 데브옵스(DevOps)와 테스팅

데브옵스는 개발(테스팅 포함)과 운영이 협력해 공통된 목표를 달성하도록 시너지 창출을 목표로 하는 조직 차원의 접근법이다. 데브옵스는 개발(테스팅 포함)과 운영이 가진 생각의 차이를 줄임과 동시에 각자 하는 일의 가치를 서로 동등하게 보도록 조직문화의 변화가 필요하다. 데브옵스는 팀의 자율성, 빠른 피드백, 통합 도구 체인, 지속적 통합(CI)과 지속적 배포(CD)와 같은 기술 실천법을 장려한다. 팀은 데브옵스 배포 파이프라인(pipeline)을 통해 높은 품질의 코드를 더 빠르게 빌드/테스트/릴리스할 수 있다 (Kim 2016).

테스팅 관점에서 데브옵스의 이점은 다음과 같다:

- 코드 품질, 그리고 변경사항이 기존 코드에 악영향을 미치는지 여부에 대한 빠른 피드백을 제공한다.
- 지속적 통합은 개발자가 컴포넌트 테스트 및 정적분석과 함께 높은 품질의 코드를 제출하도록 장려함으로써 시프트-레프트 테스팅 접근법(2.1.5 참조)을 장려한다.
- 안정적인 테스트 환경 구축을 촉진하는 지속적 통합(CI)/지속적 배포(CD)와 같은 자동화 프로세스를 장려한다.
- 비기능 품질 특성(예: 성능, 신뢰성)의 가시성을 높여준다.
- 배포 파이프라인을 통한 자동화로 반복적인 수동 테스팅의 필요성을 줄여준다.
- 자동 리그레션 테스트의 규모와 범위가 늘어나 리그레션 발생 리스크가 최소화된다.

데브옵스도 다음과 같은 리스크와 어려움을 가진다:

- 데브옵스 배포 파이프라인을 정의하고 설정해야 한다.
- 지속적 통합/지속적 배포 도구를 도입하고 유지보수해야 한다.
- 테스트 자동화를 위한 추가 자원이 필요하며, 그것을 설정 및 유지보수하기가 어려울 수 있다.

데브옵스는 높은 수준의 테스팅 자동화를 동반하지만, 수동 테스팅 또한 (특히 사용자 관점에서) 여전히 필요하다.

2.1.5. 시프트-레프트 접근법

조기 테스팅 원리(1.3 참조)는 테스트를 소프트웨어 개발수명주기(SDLC) 초기에 수행하도록 하는 접근법이기 때문에 시프트-레프트라고 지칭하기도 한다. 시프트-레프트는 테스트를 더 일찍 수행해야 한다는 것을 의미하지만(예: 코드가 구현되거나 컴포넌트가 통합될 때까지 기다리지 않고), 그렇다고 소프트웨어 개발수명주기(SDLC) 후반의 테스트를 무시해도 된다는 의미는 아니다.

테스팅에서 시프트-레프트를 달성하는 좋은 프랙티스가 있으며, 다음은 그중 일부를 나열한 것이다:

- 테스팅 관점에서 명세를 리뷰한다. 이런 명세 리뷰 활동을 통해 모호성, 불완전성, 불일치 등 잠재적인 결함을 발견하는 경우가 많다.

- 코드를 작성하기 전에 테스트 케이스를 작성하고, 코드 구현 중 코드를 테스트 하네스(test harness)에서 실행한다.
- 빠른 피드백을 제공하고, 코드 저장소에 소스 코드를 저장할 때 자동 컴포넌트 테스트를 함께 제출하도록 하는 지속적인 통합, 가능하다면 지속적인 배포까지 적용한다.
- 동적 테스팅 전 또는 자동화된 프로세스의 일부로 소스 코드의 정적분석을 완료한다.
- 가능한 한 컴포넌트 테스트에서부터 비기능 테스팅을 수행한다. 비기능 테스트는 완성 시스템과 실제 환경을 대변하는 테스트 환경이 가용한 소프트웨어 개발수명주기(SDLC) 후반에 수행하는 경향이 있으므로, 이는 일종의 시프트-레프트가 된다.

시프트-레프트 접근법은 프로세스 초기에 훈련, 공수, 비용이 추가로 들지만, 프로세스 후반의 공수와 비용의 절감을 기대할 수 있다.

시프트-레프트 접근법을 위해서는 이해관계자들이 개념을 이해하고 받아들이는 것이 중요하다.

2.1.6. 회고 및 프로세스 개선

회고("프로젝트 종료 후 회의" 또는 프로젝트 회고라고도 함)는 프로젝트나 반복 주기가 끝날 때, 릴리스 마일스톤에서, 또는 필요시 진행할 수 있다. 회고의 시기와 구성은 사용 중인 소프트웨어 개발수명주기(SDLC) 모델에 따라 달라진다. 이 회의에서 참가자(테스터 외에도 개발자/설계자/제품 소유자/비즈니스 분석가 등)는 다음에 대해 논의한다:

- 무엇이 성공적이었고, 유지해야 할 것은 무엇인가?
- 무엇이 부족했고, 개선할 수 있는 점은 무엇인가?
- 향후 개선 사항을 도입하고 성공 요소를 유지하려면 어떻게 해야 하는가?

결과는 기록해야 하며, 이를 테스트 완료 보고서(5.3.2 참조)에 포함하는 경우가 많다. 회고는 지속적인 개선을 성공적으로 구현하기 위해 반드시 필요하며, 권장된 모든 개선 사항에 대한 후속 조치가 이루어지는 것이 중요하다.

테스팅 관점에서 일반적인 이점은 다음과 같다:

- 테스트 효과성/효율성 향상(예: 프로세스 개선 권고 사항 구현을 통해)
- 테스트웨어 품질 향상(예: 테스트 프로세스를 함께 검토함으로써)
- 팀의 결속 및 학습 향상(예: 문제를 제기하고, 개선점을 제안할 기회를 제공함으로써)
- 테스트 베이시스 품질 개선(예: 요구사항의 범위나 품질에서 부족한 점을 발견하고, 수정함으로써)
- 개발과 테스팅 간의 협업 개선(예: 정기적으로 협업 과정을 검토하고 최적화함으로써)

2.2. 테스트 레벨과 테스트 유형

테스트 레벨은 함께 구성하고 관리하는 테스트 활동 집합이다. 각 테스트 레벨은 특정 개발 단계의 소프트웨어와 관련해 수행하는 테스트 프로세스의 인스턴스(instance)이다. 단계에 따라 소프트웨어는 개별 컴포넌트부터 완성된 시스템에 이를 수 있으며, 경우에 따라서는 시스템의 시스템일 수도 있다.

테스트 레벨은 소프트웨어 개발수명주기(SDLC) 내의 다른 활동과 연관성을 가진다. 순차적 소프트웨어 개발수명주기(SDLC) 모델은 한 레벨의 완료 조건이 다음 레벨의 시작 조건에 포함되도록 테스트 레벨을 정의하는 경우가 많다. 또 이것과는 다른 반복적 모델도 있다. 개발 활동이 여러 테스트 레벨에 걸쳐 진행되고, 시간이 지나면서 테스트 레벨이 서로 겹치기도 한다.

테스트 유형은 특정 품질 특성 관련 테스트 활동의 집합으로, 이런 테스트 활동은 대부분 모든 테스트 레벨에서 수행할 수 있다.

2.2.1. 테스트 레벨

이 실러버스는 다음 다섯 가지 테스트 레벨을 설명한다:

- **컴포넌트 테스팅**(단위 테스팅이라고도 함)은 컴포넌트를 개별적으로 테스트하는 데 중점을 둔다. 테스트 하네스 또는 단위 테스트 프레임워크와 같은 구체적인 지원 수단이 필요한 경우가 많다. 컴포넌트 테스팅은 일반적으로 개발자가 자신의 개발 환경에서 수행한다.
- **컴포넌트 통합 테스팅**(단위 통합 테스팅이라고도 함)은 컴포넌트 간의 인터페이스와 상호 작용을 테스트하는 데 중점을 둔다. 컴포넌트 통합 테스팅은 상향식, 하향식, 빅뱅 등 통합 전략 접근법에 따라 크게 달라진다.
- **시스템 테스팅**은 전체 시스템 또는 제품의 전반적인 동작과 기능에 중점을 두며, 엔드투엔드(end-to-end) 동작에 대한 기능 테스팅과 품질 특성에 대한 비기능 테스팅을 포함하는 경우가 많다. 실제 환경을 대변하는 테스트 환경에서 완성된 시스템으로 테스트하는 것을 선호하는 비기능 품질 특성도 있다(예: 사용성). 서브-시스템에 대한 시뮬레이션을 사용하기도 한다. 시스템 테스팅은 독립 테스트팀이 수행할 수 있으며, 시스템의 명세와 관련이 있다.
- **시스템 통합 테스팅**은 다른 시스템 또는 외부 서비스와 테스트 대상 시스템의 인터페이스를 테스트하는 데 중점을 둔다. 시스템 통합 테스팅은 가급적 운영 환경과 유사한 적절한 테스트 환경을 사용한다.
- **인수 테스팅**은 밸리데이션과 배포할 준비, 즉 시스템이 사용자의 비즈니스에 필요한 사항을 충족하는지를 확인하는 데 중점을 둔다. 인수 테스팅은 실제 사용자가 수행하는 것이 이상적이다. 인수 테스팅의 주요 유형으로 사용자 인수 테스팅(UAT), 운영 인수 테스팅, 계약 및 규제 인수 테스팅, 알파 테스팅, 베타 테스팅 등이 있다.

테스트 레벨은 테스트 활동의 중복을 피하기 위해 다음과 같은 다양한 속성을 고려해 구분한다:

- 테스트 대상
- 테스트 목적
- 테스트 베이시스
- 결함과 장애
- 접근법과 역할

2.2.2 테스트 유형

프로젝트에 다양하게 적용할 수 있는 여러 테스트 유형이 있다. 이 실러버스는 다음 네 가지 테스트 유형을 다루고 있다:

기능 테스팅은 컴포넌트 또는 시스템이 수행해야 하는 기능을 평가한다. 기능은 테스트 대상이 '무엇을' 해야 하는지를 의미한다. 기능 테스팅의 주요 목적은 기능 성숙도(완전성), 기능 정확성, 기능 타당성(적합성)을 확인하는 것이다.

비기능 테스팅은 컴포넌트 또는 시스템의 기능 특성 이외의 속성을 평가한다. 비기능 테스팅은 "시스템이 얼마나 잘 동작하는지" 테스트하는 것이다. 비기능 테스팅의 주요 목적은 비기능 소프트웨어 품질 특성을 확인하는 것이다. ISO/IEC 25010 표준은 비기능 소프트웨어 품질 특성을 다음과 같이 분류한다.

- 수행 효율성(Performance efficiency)
- 호환성(Compatibility)
- 유용성(Usability)
- 신뢰도(Reliability)
- 보안(Security)
- 유지 가능성(유지 관리성)(Maintainability)
- 이동성(Portability)

수명주기 초기에(예: 리뷰와 컴포넌트 테스팅 또는 시스템 테스팅 도중) 비기능 테스팅을 시작하는 것이 바람직할 때가 있다. 기능 테스트에서 비기능 테스트를 도출하는 경우도 많다. 이때 기능 테스트에서 기능이 수행되는 동안 비기능 제약 조건의 충족 여부를 확인하게 된다(예: 특정 시간 내에 기능이 수행되는지 확인하거나, 기능이 새로운 플랫폼으로 이식될 수 있는지 확인). 비기능 결함을 늦게 발견하면 프로젝트의 성공에 심각한 위협이 될 수 있다. 비기능 테스팅은 매우 특수한 테스트 환경이 필요한 경우도 있다. 예를 들어, 사용성 테스팅에는 사용성 랩(usability lab)이 필요하다.

블랙박스 테스팅(4.2 참조)은 명세를 기반으로 하며, 테스트 대상 외부에 있는 문서에서 테스트를 도출한다. 블랙박스 테스팅의 주요 목적은 명세와 비교해 시스템의 동작을 확인하는 것이다.

화이트박스 테스팅(4.3 참조)은 구조 기반이며, 시스템의 구현 또는 내부 구조(예: 코드, 아키텍처, 워크플

로우, 데이터 플로우)에서 테스트를 도출한다. 화이트박스 테스팅의 주요 목적은 테스트를 통해 내부 구조를 인수에 필요한 수준까지 충분히 커버하는 것이다.

위에서 언급한 네 가지 테스트 유형은 모든 테스트 레벨에 적용할 수 있지만, 레벨마다 관심의 대상은 다를 수 있다. 언급한 모든 테스트 유형을 위한 테스트 컨디션과 테스트 케이스를 도출하기 위해 다양한 테스트 기법을 사용할 수 있다.

2.2.3. 확인 테스팅 및 리그레션 테스팅

일반적으로 컴포넌트나 시스템을 변경하는 이유는 새로운 기능을 추가해 개선하거나 결함을 제거해 수정하기 위함이다. 이때는 테스팅에 확인 테스팅과 리그레션 테스팅을 추가할 필요가 있다.

확인 테스팅은 원래 결함이 성공적으로 수정됐는지 확인한다. 리스크에 따라 다음과 같은 여러 가지 방법으로 수정된 소프트웨어 버전을 테스트할 수 있다:

- 결함으로 인해 이전에 실패했던 모든 테스트 케이스를 실행한다.
- 결함을 수정하기 위해 변경한 사항을 확인하는 새로운 테스트를 추가한다.

그러나 결함을 수정하는 데 시간이나 비용이 부족한 경우, 결함으로 생긴 장애를 재현하기 위한 절차를 거쳐 장애가 발생하지 않는지 확인하는 것만으로 확인 테스팅이 끝날 수도 있다.

리그레션 테스팅은 변경으로 인해 부정적인 영향이 없었는지 확인하는 것이다. 이미 확인 테스팅이 끝난 수정 사항도 여기서 말하는 변경에 포함된다. 이런 부정적인 영향은 변경이 이루어진 컴포넌트 자체, 같은 시스템의 다른 컴포넌트, 또는 연결된 다른 시스템에도 영향을 미칠 수 있다. 리그레션 테스팅은 테스트 대상 자체에만 국한되지 않고, 환경과도 관련이 있을 수 있다. 리그레션 테스팅의 범위를 최적화하기 위해 영향도 분석을 먼저 수행하는 것이 좋다. 영향도 분석은 소프트웨어의 어느 부분이 영향을 받을 수 있는지 보여준다.

리그레션 테스트 스위트는 반복적으로 실행되며, 반복 주기 또는 릴리스 때마다 리그레션 테스트 케이스 수가 늘어나게 되므로, 리그레션 테스트는 자동화하기 매우 적합한 대상이 된다. 이런 테스트의 자동화는 프로젝트 초기에 시작해야 한다. 데브옵스(2.1.4 참조)와 같이 지속적 통합을 사용하는 경우, 자동 리그레션 테스트를 포함하는 것이 좋은 프랙티스이다. 상황에 따라 여러 테스트 레벨의 리그레션 테스트가 포함될 수 있다.

어떤 테스트 레벨이라도 해당 레벨에서 결함을 수정하거나 변경을 적용한 경우, 테스트 대상에 대한 확인 테스팅과 리그레션 테스팅이 필요하다.

2.3. 유지보수 테스팅

유지보수에는 여러 범주가 있다. 문제를 수정하기 위한 유지보수도 있고, 환경 변화에 적응하거나, 성능 또는 유지보수성(자세한 내용은 ISO/IEC 14764 참조)을 개선하기 위한 것도 있기 때문에, 유지보수는 계획된 릴리스/배포 또는 계획되지 않은 릴리스/배포(핫픽스) 모두와 연관돼 있다. 변경 전에 영향도 분석을 수행해 시스템의 다른 영역에 미칠 잠재적 영향을 변경사항 결정에 참고할 수 있다. 운용 환경에서 시스템의 변경사항을 테스트하는 것에는 변경 구현의 성공을 검증하는 것과, 변경되지 않은 시스템 영역(보통은 시스템 대부분)에서 발생할 수 있는 리그레션을 확인하는 작업이 모두 포함된다.

유지보수 테스팅의 범위는 일반적으로 다음에 따라 달라진다:

- 변경의 리스크 수준
- 기존 시스템의 크기
- 변경사항의 크기

유지보수와 유지보수 테스팅의 계기는 다음과 같이 분류할 수 있다:

- 계획된 개선사항(즉, 릴리스 기반), 수정을 위한 변경, 핫픽스(hot-fixes)와 같은 수정사항
- 운영 환경의 업그레이드나 마이그레이션(예: 한 플랫폼에서 다른 플랫폼으로)하는 경우, 변경된 소프트웨어뿐만 아니라 새로운 환경 관련 테스트가 필요할 수 있으며, 다른 애플리케이션의 데이터를 유지보수 중인 시스템으로 마이그레이션할 때, 데이터 변환 테스트가 필요할 수 있다.
- 애플리케이션의 수명이 다하는 등의 단종. 시스템을 단종할 때 데이터 보존 기간이 길면 데이터 보관(archiving) 테스팅이 필요할 수 있다. 보관 기간 중 데이터가 필요한 경우를 대비해, 보관 이후 복원 및 복구 절차 테스팅이 필요할 수 있다.

제 3 장 정적 테스팅 — 80분

용어

이상 사항(anomaly), 동적 테스팅(dynamic testing), 공식 리뷰(formal review), 비공식 리뷰(informal review), 인스펙션(inspection), 리뷰(review), 정적 분석(static analysis), 정적 테스팅(static testing), 기술 리뷰(technical review), 워크쓰루(walkthrough)

3장 학습 목표

3.1 정적 테스팅 기초

FL-3.1.1 (K1) 다양한 정적 테스트 기법으로 검토할 수 있는 산출물 유형을 인식할 수 있다

FL-3.1.2 (K2) 정적 테스팅의 가치를 설명할 수 있다

FL-3.1.3 (K2) 정적 테스팅과 동적 테스팅을 비교하고 대조할 수 있다

3.2 피드백과 리뷰 프로세스

FL-3.2.1 (K1) 이해관계자 피드백을 조기에 자주 받을 때의 이점을 식별할 수 있다

FL-3.2.2 (K2) 리뷰 프로세스에서 수행하는 활동을 요약할 수 있다

FL-3.2.3 (K1) 리뷰를 수행할 때 주요 역할별로 어떤 책임이 부과되는지 상기할 수 있다

FL-3.2.4 (K2) 다양한 리뷰 유형을 비교하고 대조할 수 있다

FL-3.2.5 (K1) 성공적인 리뷰에 기여하는 요소를 상기할 수 있다

3.1. 정적 테스팅의 기초

동적 테스팅과 달리 정적 테스팅은 테스트 대상 소프트웨어를 실행하지 않아도 된다. 코드, 프로세스 명세, 시스템 아키텍처 명세, 기타 작업 산출물을 수동으로(예: 리뷰) 또는 도구를 사용해 평가한다(예: 정적 분석). 테스팅 목표는 품질 개선, 결함 식별, 또한 가독성/완전성/정확성/테스트 용이성/일관성과 같은 특성을 평가하는 것이다. 정적 테스팅은 베리피케이션과 벨리데이션 모두에 적용할 수 있다.

예제 매핑, 사용자 스토리 공동 작성, 백로그 개선 작업 시 테스터/비즈니스 담당자/개발자 모두 같이 협업해 사용자 스토리와 관련 작업 산출물이 정의된 기준을 충족하는지 확인한다. 준비의 정의(5.1.3 참조)가 이런 기준의 예가 될 수 있다. 사용자 스토리가 완전하고 이해하기 쉬우며, 테스트 가능한 인수 조건을 가지고 있는지 확인하기 위해 리뷰 기법을 적용할 수 있다. 적절한 질문을 통해 테스터는 제안된 사용자 스토리를 탐색하고, 문제를 제기하며 개선하는 데 도움을 준다.

정적 분석은 테스트 케이스가 필요 없고, 도구(6장 참조)를 사용하는 경우가 많기 때문에 상대적으로 적은 노력으로 동적 테스팅 전에 문제를 식별할 수 있다. 정적 분석을 지속적 통합(CI) 프레임워크에 통합하는 경우가 많다(2.1.4 참조). 정적 분석은 주로 구체적인 코드 결함을 식별하는 데 사용하지만, 유지보수성과 보안을 평가하는 데도 사용한다. 맞춤법 검사기나 가독성 도구도 정적 분석 도구의 예라고 할 수 있다.

3.1.1. 정적 테스팅으로 검사 가능한 작업 산출물

정적 테스트를 사용해 거의 모든 작업 산출물을 검토할 수 있다. 예를 들어, 요구사항 명세서, 소스 코드, 테스트 계획서, 테스트 케이스, 제품 백로그 항목, 테스트 차터, 프로젝트 문서, 계약서, 모델 등이 그 대상이 될 수 있다.

읽고 이해할 수 있는 모든 작업 산출물은 리뷰 대상이 될 수 있다. 그러나 정적 분석의 경우, 작업 산출물을 검토할 수 있는 기준이 되는 어떤 구조가 필요하다(예: 모델, 공식 문법이 정해진 코드나 텍스트).

사람이 해석하기 어려운 것과 도구로 분석해서는 안 되는 작업 산출물(예: 법적으로 문제가 되는 타사의 실행 코드)은 정적 테스트에 적합하지 않은 작업 산출물이다.

3.1.2. 정적 테스팅의 가치

정적 테스팅은 소프트웨어 개발수명주기(SDLC) 초기 단계에서 결함을 식별하기 때문에 조기 테스팅 원리(1.3절 참조)를 지킬 수 있게 한다. 또한, 동적 테스팅으로 식별할 수 없는 결함(예: 도달 불가능한 코드, 원하는 대로 구현되지 않은 설계 패턴, 비-실행 작업 산출물의 결함)을 찾을 수도 있다.

정적 테스팅은 작업 산출물의 품질을 평가하고 신뢰를 쌓을 방법을 제공한다. 또한, 문서화된 요구사항을 확인함으로써 이해관계자는 요구사항이 실제로 자기가 필요한 것을 묘사하고 있는지 확인할 수 있다. 정적 테스팅은 소프트웨어 개발수명주기(SDLC) 초기에 수행할 수 있으므로 관련된 이해관계자 간 공통

된 이해를 형성할 수 있다. 관련된 이해관계자 간의 의사소통도 개선된다. 때문에 정적 테스팅에 다양한 이해관계자가 참여하는 것이 권장된다.

리뷰를 구축하는 비용은 클 수 있지만, 프로젝트 후반에 결함을 수정하는 시간과 노력이 줄어들어 리뷰를 수행하지 않을 때보다 전체 프로젝트 비용이 훨씬 낮아지는 경우가 많다.

정적 분석은 동적 테스팅보다 효율적으로 코드 결함을 식별할 수 있으며, 대부분의 경우 결과적으로 코드 결함이 줄고, 전반적인 개발 노력도 감소한다.

3.1.3. 정적 테스팅과 동적 테스팅의 차이

정적 테스팅과 동적 테스팅은 서로를 보완한다. 두 활동은 작업 산출물 결함의 식별을 지원하는 등 비슷한 목적을 가지고 있지만(1.1.1 참조), 다음과 같은 차이도 있다:

- 정적 테스팅과 동적 테스팅(장애 분석을 통해) 모두 결함을 식별한다는 것은 같지만, 정적 테스팅이나 동적 테스팅 중 한 가지로만 식별할 수 있는 결함 유형도 있다.
- 정적 테스팅은 결함을 직접 식별하는 반면, 동적 테스팅은 장애를 일으킨 후 연관된 결함을 후속 분석을 통해 찾아낸다.
- 정적 테스팅은 거의 실행되지 않거나 동적 테스팅으로 도달하기 어려운 코드 경로에 있는 결함을 더 쉽게 발견할 수 있다.
- 정적 테스팅은 비-실행(non-executable) 작업 산출물에도 적용할 수 있지만, 동적 테스팅은 실행 가능한 작업 산출물에만 적용할 수 있다.
- 정적 테스팅은 코드 실행에 의존하지 않는 품질 특성(예: 유지보수성)을 측정할 수 있고, 동적 테스팅은 코드 실행에 의존적인 품질 특성(예: 성능 효율성)을 측정할 수 있다.

정적 테스팅을 통해 더 쉽게 적은 비용으로 식별할 수 있는 대표적인 결함은 다음과 같다:

- 요구사항 결함(예: 불일치, 모호성, 모순, 누락, 부정확성, 중복)
- 설계 결함(예: 비효율적인 데이터베이스 구조, 모듈화 불량)
- 특정 유형의 코딩 결함(예: 정의되지 않은 값을 가진 변수, 선언되지 않은 변수, 도달할 수 없거나 중복된 코드, 과도한 코드 복잡성)
- 표준 위반(예: 코딩 표준 명명 규칙을 준수하지 않는 경우)
- 잘못된 인터페이스 명세(예: 매개변수의 개수, 유형, 순서 불일치)
- 특정 유형의 보안 취약성(예: 버퍼 오버플로우)
- 테스트 베이시스 커버리지의 차이 또는 부정확성(예: 인수 조건 하나에 대한 테스트 누락)

3.2. 피드백과 리뷰 프로세스

3.2.1. 이해관계자 피드백을 조기에 자주 받을 때의 이점

피드백을 조기에 자주 받으면 잠재적인 품질 문제를 조기에 파악할 수 있다. 소프트웨어 개발수명주기(SDLC) 동안 이해관계자의 참여가 적으면 개발 중인 제품이 이해관계자의 초기 및 현재 요구를 충족하지 못할 수 있다. 이해관계자가 원하는 것을 전달하지 못하면 큰 비용이 드는 재작업, 납기일 지연, 서로 간의 비난 등이 발생할 수 있으며, 심한 경우 프로젝트가 완전히 실패할 수도 있다.

소프트웨어 개발수명주기(SDLC) 전반에 걸쳐 이해관계자의 피드백을 자주 받으면 요구사항에 대한 오해를 방지하고 요구사항 변경을 조기에 이해하고 구현할 수 있다. 이를 통해 개발팀은 구현 중인 제품에 대한 이해도를 높일 수 있다. 그리고 이해관계자에게 가장 중요하고, 식별한 리스크에 가장 긍정적인 영향을 미치는 기능에 집중할 수 있다.

3.2.2. 리뷰 프로세스 활동

ISO/IEC 20246 표준은 특정 상황에 맞게 리뷰 프로세스를 조정할 수 있는 체계적이면서 유연한 프레임워크를 제공하는 보편적인 리뷰 프로세스를 정의한다. 필요한 리뷰가 공식적일수록 여러 활동에 대해 설명된 작업이 더 많아지게 된다.

작업 산출물이 너무 커서 한 번의 리뷰로 다룰 수 없는 경우도 많다. 전체 작업 산출물의 리뷰를 끝내기 위해 리뷰 프로세스를 여러 번 수행할 수 있다.

리뷰 프로세스에 포함되는 활동은 다음과 같다:

- **계획.** 계획 단계에서 목적, 리뷰 대상 작업 산출물, 평가할 품질 특성, 집중할 영역, 완료 조건, 표준과 같은 추가 정보, 공수, 리뷰 일정 등으로 구성되는 리뷰의 범위를 정의해야 한다.
- **리뷰 착수.** 리뷰 착수의 목표는 관련된 모든 사람과 사항이 리뷰를 시작할 준비가 되었는지 확인하는 것이다. 여기에는 모든 참여자가 리뷰 중인 작업 산출물에 접근할 수 있는지, 자신의 역할과 책임을 이해하고 있는지, 리뷰를 수행하는 데 필요한 것들을 받았는지 확인하는 과정이 포함된다.
- **개별 리뷰.** 모든 검토자(reviewer)는 하나 이상의 리뷰 기법(예: 체크리스트 기반 리뷰, 시나리오 기반 리뷰)을 적용해 리뷰 중인 작업 산출물의 품질을 평가하고, 이상 사항, 권장 사항, 의문 사항을 식별하기 위해 개별 리뷰를 수행한다. ISO/IEC 20246 표준은 여러 리뷰 기법에 대한 보다 심층적인 내용을 제공한다. 검토자는 식별한 모든 이상 사항, 권장 사항, 의문 사항을 기록한다.
- **논의 및 분석.** 리뷰 중에 확인한 이상 사항이 반드시 결함은 아니므로 모든 이상 사항에 대해 분석하고 논의할 필요가 있다. 각 이상 사항의 상태, 담당자, 필요 조치를 판단해야 한다. 이는 일반적으로 리뷰 회의에서 이루어지며, 회의에서 참가자들은 리뷰한 작업 산출물의 품질 수준과 필요한 후속 조치도 결정한다. 조치를 완료하기 위해 후속 리뷰가 필요할 수 있다.

- **수정 및 보고.** 모든 결함에 대한 결함 보고서를 작성해 후속 조치가 추적 가능하도록 한다. 완료 조건을 충족하면 작업 결과물을 승인할 수 있다. 리뷰 결과를 보고한다.

3.2.3. 리뷰에서의 역할과 책임

리뷰에는 다양한 이해관계자가 참여하며, 한 명이 여러 역할을 동시에 맡을 수도 있다. 주요 역할과 책임은 다음과 같다:

- 관리자 - 리뷰할 내용을 결정하고, 리뷰에 필요한 사람과 시간 등 자원을 제공한다.
- 저자 - 리뷰 대상 작업 산출물을 작성하고 수정한다.
- 중재자(퍼실리테이터라고도 함) - 중재, 시간 관리, 모든 사람이 자유롭게 발언할 수 있는 안전한 리뷰 환경 조성 등 리뷰 회의의 효과적인 운영을 담당한다.
- 서기(기록자라고도 함) - 리뷰어로부터 이상 사항을 수집하고, 결정 사항이나 리뷰 회의 중에 발견한 새로운 이상 사항 등 리뷰 정보를 기록한다.
- 리뷰어(검토자) - 리뷰를 수행한다. 리뷰어는 프로젝트에 참여하는 사람 또는 주제 전문가, 기타 이해관계자가 될 수 있다.
- 리뷰 리더 - 리뷰에 참여할 사람을 결정하고, 리뷰 시간과 장소를 협의하는 등 리뷰에 대한 전반적인 책임을 진다.

기타 더 세부적인 역할도 있으며, 관련해서는 ISO/IEC 20246 표준에서 설명하고 있다.

3.2.4. 리뷰 유형

비공식 리뷰부터 공식 리뷰까지 다양한 리뷰 유형이 있다. 리뷰가 어느 정도로 공식적이어야 하는지는 사용 중인 소프트웨어 개발수명주기(SDLC), 개발 프로세스의 성숙도, 리뷰 대상 작업 산출물의 중요도와 복잡성, 법적 또는 규제 요구사항, 감사용 기록의 필요성 등에 따라 달라진다. 처음에는 비공식적으로 리뷰하고, 나중에 보다 공식적으로 리뷰하는 등, 같은 작업 산출물도 여러 형태로 리뷰할 수 있다.

필요한 리뷰 목적을 달성하려면 적절한 리뷰 유형을 선택하는 것이 중요하다(3.2.5 참조). 목적 외에도 프로젝트 요구사항, 가용 자원, 작업 산출물 유형과 리스크, 비즈니스 영역, 사내 문화와 같은 요소도 선택에 반영된다.

많이 사용하는 몇 가지 리뷰 유형은 다음과 같다:

- **비공식 리뷰(Informal review).** 비공식 리뷰는 정의된 프로세스를 따르지 않으며, 공식적인 결과 문서도 요구하지 않는다. 주요 목적은 이상 사항을 식별하는 것이다.
- **워크쓰루(Workthrough).** 저자가 리더가 되는 워크쓰루를 통해 품질 평가 및 작업 산출물에 대한 신뢰 구축, 리뷰어 교육, 합의 도출, 새로운 아이디어 창출, 저자가 개선할 수 있도록 동기 부여 및 지원, 이상 사항 발견 등 여러 가지 목적을 달성할 수 있다. 리뷰어는 워크쓰루 전에

개별 리뷰를 수행할 수도 있지만, 반드시 해야 하는 것은 아니다.

- **기술 리뷰(Technical review).** 기술 리뷰는 기술적인 자격을 갖춘 리뷰어가 수행하고, 중재자가 리더가 된다. 기술 리뷰의 목적은 기술 문제에 대한 합의를 도출하고 결정을 내리는 것뿐만 아니라, 이상 사항 식별, 품질 평가 및 작업 산출물에 대한 신뢰 구축, 새로운 아이디어 창출, 저자가 개선할 수 있도록 동기를 부여하고 지원하는 것이다.
- **인스펙션(Inspection).** 인스펙션은 가장 공식적인 리뷰 유형이므로 보편적 프로세스(3.2.2 참조)를 철저히 따라야 한다. 주요 목적은 이상 사항을 최대한 많이 찾는 것이다. 기타 목적으로 품질 평가, 작업 산출물에 대한 신뢰 구축, 저자가 개선할 수 있도록 동기 부여 및 지원을 들 수 있다. 메트릭(metrics)을 수집해 리뷰 프로세스를 포함한 전체 소프트웨어 개발수명주기(SDLC)를 개선하는 데 사용한다. 인스펙션에서는 저자가 리뷰 리더나 서기가 될 수 없다.

3.2.5. 리뷰의 성공 요소

리뷰의 성공 여부를 결정하는 요소에는 다음과 같은 것들이 있다:

- 명확한 목표와 측정 가능한 완료 조건을 정의한다. 참가자의 평가가 목적이 되어서는 절대 안 된다.
- 주어진 목표를 달성할 수 있으면서 작업 산출물 유형, 리뷰 참여자, 프로젝트 요구사항 및 정황에 맞는 리뷰 유형을 선택한다.
- 리뷰어가 개별 리뷰 또는 리뷰 회의(개최 시)에서 집중력을 잃지 않도록 작은 단위로 리뷰를 진행한다.
- 이해관계자와 저자에게 리뷰 피드백을 제공해 제품 및 활동을 개선할 수 있게 한다(3.2.1 참조).
- 참가자가 리뷰를 준비할 수 있는 충분한 시간을 제공한다.
- 리뷰 프로세스를 관리층이 지원한다.
- 학습 및 프로세스 개선 촉진을 위해 리뷰가 조직 문화의 일부가 되도록 한다.
- 모든 참가자가 자신의 역할을 어떻게 충족해야 하는지 알 수 있도록 적절한 교육을 제공한다.
- 회의에 퍼실리테이션(facilitation)을 적용한다.

제 4 장 테스트 분석과 설계 | 390분

용어

인수 조건(acceptance criteria), 인수 테스트 주도 개발(acceptance test-driven development), 블랙박스 테스트 기법(black-box test technique), 경계값 분석(boundary value analysis), 분기 커버리지(branch coverage), 체크리스트 기반 테스팅(checklist-based testing), 협업 기반 테스트 접근법(collaboration-based test approach), 커버리지(coverage), 커버리지 항목(coverage item), 결정 테이블 테스팅(decision table testing), 동등 분할(equivalence partitioning), 오류 추정(error guessing), 경험 기반 테스트 기법(experience-based test technique), 탐색적 테스팅(exploratory testing), 상태 전이 테스팅(state transition testing), 구문 커버리지(statement coverage), 테스트 기법(test technique), 화이트박스 테스트 기법(white-box test technique)

4장 학습 목표

4.1 테스트 기법 개요

FL-4.1.1 (K2) 블랙박스, 화이트박스, 경험 기반 테스트 기법을 구별할 수 있다

4.2 블랙박스 테스트 기법

FL-4.2.1 (K3) 동등 분할을 사용해 테스트 케이스를 도출할 수 있다
FL-4.2.2 (K3) 경계값 분석을 사용해 테스트 케이스를 도출할 수 있다
FL-4.2.3 (K3) 결정 테이블 테스팅을 사용해 테스트 케이스를 도출할 수 있다
FL-4.2.4 (K3) 상태 전이 테스팅을 사용해 테스트 케이스를 도출할 수 있다

4.3 화이트박스 테스트 기법

FL-4.3.1 (K2) 구문 테스팅을 설명할 수 있다
FL-4.3.2 (K2) 분기 테스팅을 설명할 수 있다
FL-4.3.3 (K2) 화이트박스 테스팅의 가치를 설명할 수 있다

4.4 경험 기반 테스트 기법

FL-4.4.1 (K2) 오류 추정을 설명할 수 있다
FL-4.4.2 (K2) 탐색적 테스팅을 설명할 수 있다
FL-4.4.3 (K2) 체크리스트 기반 테스팅을 설명할 수 있다

4.5. 협업 기반 테스트 접근법

FL-4.5.1 (K2) 개발자 및 업무 대표자와 협업해 사용자 스토리를 작성하는 방법을 설명할 수 있다

FL-4.5.2 (K2) 인수 조건을 작성하는 여러가지 방법을 분류할 수 있다

FL-4.5.3 (K3) 인수 테스트 주도 개발(ATDD)을 사용해 테스트 케이스를 도출할 수 있다

4.1. 테스트 기법 개요

테스트 기법은 테스터의 테스트 분석(무엇을 테스트할지)과 테스트 설계(어떻게 테스트할지) 작업을 지원한다. 테스트 기법은 적은 수이지만 충분한 테스트 케이스를 체계적인 방식으로 개발할 수 있도록 해준다. 또한, 테스트 기법은 테스터가 테스트 분석과 설계에서 테스트 컨디션을 정의하고, 커버리지 항목과 테스트 데이터를 식별하는 데 도움을 준다. 테스트 기법 및 연관된 측정에 관해서는 ISO/IEC/IEEE 29119-4 표준과 Beizer 1990, Craig 2002, Copeland 2004, Koomen 2006, Jorgensen 2014, Ammann 2016, Forgács 2019를 참조할 수 있다.

이 실러버스는 테스트 기법을 블랙박스, 화이트박스, 경험 기반으로 분류하고 있다.

블랙박스 테스트 기법(명세 기반 기법이라고도 함)은 내부 구조를 참조하지 않고, 테스트 대상의 명시된 동작에 대한 분석을 기반으로 한다. 따라서 테스트 케이스는 소프트웨어 구현 방식에 의존하지 않는다. 결국 구현이 바뀌더라도 필요한 동작이 동일하다면, 테스트 케이스는 여전히 유효하게 된다.

화이트박스 테스트 기법(구조 기반 기법이라고도 함)은 테스트 대상의 내부 구조와 처리에 대한 분석을 기반으로 한다. 테스트 케이스는 소프트웨어 설계 방식에 의존하기 때문에 테스트 대상의 설계나 구현이 끝난 후에 만들 수 있다.

경험 기반 테스트 기법은 테스터의 지식과 경험을 테스트 케이스의 설계 및 구현에 효과적으로 활용하게 한다. 이런 기법의 효과성은 테스터의 능력에 따라 크게 달라진다. 경험 기반 테스트 기법은 블랙박스와 화이트박스 테스트 기법으로는 식별하지 못할 수 있는 결함을 찾을 수 있게 해준다. 따라서 경험 기반 테스트 기법은 블랙박스 및 화이트박스 테스트 기법을 보완한다.

4.2. 블랙박스 테스트 기법

많이 사용되는 블랙박스 테스트 기법은 다음과 같다:

- 동등 분할
- 경계값 분석
- 결정 테이블 테스팅
- 상태 전이 테스팅

4.2.1. 동등 분할

동등 분할(EP)은 테스트 대상이 하나의 분할(동등 분할이라고도 함)에 포함된 모든 요소를 동일한 방식으로 처리할 것이라는 가정하에 데이터를 분할 단위로 나눈다. 이 기법의 근거 이론은 동등 분할에 속한 특정 값을 테스트하는 테스트 케이스로 결함을 식별할 수 있다면, 같은 동등 분할의 다른 어떤 값을 테스트하는 테스트 케이스라도 해당 결함을 식별할 수 있어야 한다는 것이다. 따라서 각 분할에 대해

하나의 테스트만 수행하면 충분하다.

동등 분할은 입력, 출력, 형상 항목, 내부 값, 시간 관련 값, 인터페이스 매개변수 등 테스트 대상과 관련된 모든 데이터 요소에 대해 식별할 수 있다. 분할은 연속적이거나, 연속적이지 않을 수도 있으며, 정렬돼 있거나, 유한 또는 무한일 수도 있다. 분할은 서로 겹치지 않아야 하며, 값이 없는 공집합일 수는 없다.

테스트 대상이 단순하다면 동등 분할을 적용하기가 쉬울 수 있지만, 실제로는 테스트 대상이 다양한 값을 어떻게 처리하는지 이해하기 어려울 때가 많다. 따라서 분할 식별은 신중하게 해야 한다.

유효한 값을 포함하는 분할을 유효 분할이라고 한다. 유효하지 않은 값을 포함하는 분할을 비유효 분할이라고 한다. 유효한 값과 유효하지 않은 값의 정의는 팀과 조직마다 다를 수 있다. 예를 들어, 테스트 대상이 처리해야 하는 값 또는 명세에 처리가 정의된 값을 유효 값으로 해석할 수 있다. 비유효 값은 테스트 대상이 무시하거나 거부해야 하는 값, 또는 테스트 대상 명세에 처리방법이 정의되어 있지 않은 값으로 해석할 수 있다.

동등 분할에서 커버리지 항목은 각 분할이 된다. 이 기법으로 100% 커버리지를 달성하려면 테스트 케이스로 각 분할을 최소 한 번씩 다뤄서 식별한 모든 분할(비유효 분할 포함)이 실행되도록 해야 한다. 커버리지는 하나 이상의 테스트 케이스로 실행한 분할 수를, 식별한 총 분할 수로 나눈 값으로 측정하며 백분율로 표시한다.

대부분의 테스트 대상은 여러 분할 집합을 가지고 있기 때문에(예: 입력 변수가 2개 이상인 테스트 대상), 하나의 테스트 케이스는 여러 분할 집합에 속한 분할을 다루게 된다. 분할 집합이 다수인 경우, 가장 간단한 커버리지 기준을 이치 초이스 커버리지(Each Choice coverage)라고 한다(Ammann 2016). 이치 초이스 커버리지는 테스트 케이스가 모든 분할 집합의 각 분할을 최소 한 번은 실행할 것을 요구한다. 이치 초이스 커버리지에서는 분할의 조합을 고려하지 않는다.

4.2.2. 경계값 분석

경계값 분석(BVA)은 동등 분할의 경계 실행을 기반으로 하는 기법이다. 따라서 경계값 분석은 정렬된 분할에만 사용할 수 있다. 분할의 최솟값과 최댓값이 경계값이 된다. 경계값 분석에서 두 값이 같은 분할에 속하는 경우, 둘 사이의 모든 값도 해당 분할에 속해야 한다.

개발자가 분할의 경계에 있는 값을 다룰 때 오류를 범할 가능성이 높기 때문에 경계값 분석은 분할의 경계에 있는 값에 초점을 두게 된다. 경계값 분석으로 많이 찾는 결함은 구현된 경계가 의도한 위치보다 위나 아래에 잘못 배치됐거나 아예 누락된 결함이다.

이 실러버스는 두 가지 유형의 경계값 분석, 즉 두 개 선택(2-value)과 세 개 선택(3-value) 경계값 분석을 다룬다. 이 두 가지 유형은 100% 커버리지 달성을 위해 실행해야 하는 경계별 커버리지 항목의 수에서 차이가 난다.

두 개 선택 경계값 분석(Craig 2002, Myers 2011)은 각 경계값에 대해 두 개의 커버리지 항목을 도출한다.

경계값과 인접 분할에 속한 가장 가까운 값이 커버리지 항목이다. 두 개 선택 경계값 분석에서 100% 커버리지를 달성하려면, 테스트 케이스로 모든 커버리지 항목, 즉 식별한 모든 경계값을 실행해야 한다. 커버리지는 실행한 경계값의 수를 식별한 경계값의 총수로 나눈 값으로 백분율로 표시한다.

세 개 선택 경계값 분석(Koomen 2006, O'Regan 2019)은 각 경계값에 대해 세 개의 커버리지 항목을 도출한다. 경계값과 이웃한 양쪽의 값 모두가 커버리지 항목이다. 따라서 세 개 선택 경계값 분석에서는 경계값이 아닌 커버리지 항목도 있을 수 있다. 세 개 선택 경계값 분석에서 100% 커버리지를 달성하려면, 테스트 케이스로 모든 커버리지 항목, 즉 식별한 경계값과 그 이웃 값을 실행해야 한다. 커버리지는 실행한 경계값과 이웃한 값의 수를, 식별한 경계값과 이웃 값의 총수로 나눈 값으로 측정하며 백분율로 표시한다.

세 개 선택 경계값 분석은 두 개 선택 경계값 분석으로 발견하지 못한 결함을 식별할 수 있으므로 두 개 선택 경계값 분석보다 더 엄격하다고 할 수 있다. 예를 들어, "if (x≤10) ..."이라는 결정 구문이 "if (x=10) ..." 으로 잘못 구현된 경우, 두 개 선택 경계값 분석으로 도출한 테스트 데이터(x=10, x=11)는 결함을 식별할 수 없다. 그러나 세 개 선택 경계값 분석으로 도출한 x=9는 이를 식별할 가능성이 크다.

4.2.3. 결정 테이블 테스팅

결정 테이블은 다중 조건 조합으로 달라지는 결과를 나타내는 시스템 요구사항이 제대로 구현되었는지 테스트하는 데 사용한다. 결정 테이블은 비즈니스 규칙과 같은 복잡한 논리를 기록하는 효과적인 방법이다.

결정 테이블을 만들 때 조건들과 그에 따른 시스템의 동작 결과를 정의한다. 이것이 테이블의 행을 구성한다. 열은 각각 하나의 결정 규칙을 나타내며, 어떤 고유한 조건 조합을 연관 동작과 함께 정의한다. 제한-입력(limited-entry) 결정 테이블은 모든 조건과 동작 결괏값(관련 없거나 실행 불가능한 값 제외; 아래 참조)을 부울값(Boolean value. 참 또는 거짓)으로 표시한다. 확장-입력(extended-entry) 결정 테이블은 조건 및 동작 결괏값의 일부 또는 전부가 복수의 값(예: 숫자 범위, 동등 분할, 불연속 값)을 취할 수 있다.

조건의 표기법은 다음과 같다: "T"(참)는 조건이 충족됨을 의미한다. "F"(거짓)는 조건이 충족되지 않았음을 의미한다. "–"는 해당 조건 값이 결과에 영향이 없음을 의미한다. "N/A"는 해당 규칙에서 조건이 실행 불가능함을 의미한다. 결과 동작은 다음과 같다: "X"는 동작이 발생해야 함을 의미한다. 공백은 동작이 발생하지 않아야 함을 의미한다. 기타 표기법도 사용 가능하다.

전체 결정 테이블에는 모든 조건 조합을 포함할 수 있는 충분한 열이 있다. 실현 불가능한 조건 조합 열을 삭제해 테이블을 더 단순화 할 수 있다. 일부 조건이 결과에 영향을 미치지 않는 열들을 하나로 병합해 테이블을 최소화할 수도 있다. 결정 테이블 최소화를 위한 알고리즘은 이 실러버스의 범위를 벗어난다.

결정 테이블 테스팅에서 커버리지 항목은 실현 가능한 조건 조합을 가진 열이 된다. 이 기법으로 100%

커버리지를 달성하려면, 테스트 케이스가 이런 열을 모두 실행해야 한다. 커버리지는 실행된 열의 수를 실행 가능한 열의 총수로 나눈 값으로 측정하며 백분율로 표시한다.

결정 테이블 테스팅의 강점은 간과했을 수도 있는 조합을 포함한 모든 조건 조합을 식별하는 체계적인 방법을 제공한다는 점이다. 또한, 누락되거나 모순되는 요구사항을 찾는 데 도움이 된다. 조건의 수에 따라 규칙의 수는 기하급수적으로 늘어나기 때문에, 조건이 많으면 모든 결정 규칙을 실행하는 데 오랜 시간이 걸릴 수 있다. 이런 경우, 실행해야 하는 규칙의 수를 줄이기 위해 결정 테이블을 최소화하거나, 리스크 기반 접근법을 사용할 수 있다.

4.2.4. 상태 전이 테스팅

상태 전이 다이어그램은 가능한 상태와 유효한 상태 전이를 표시해 시스템 동작을 모델링한다. 전이는 하나의 이벤트에 의해 발생하며, 별도의 가드(guard) 조건이 있을 수 있다. 전이는 즉각적인 것으로 간주되며, 가끔 소프트웨어의 어떤 동작으로 연결되기도 한다. 전이를 표시하는 형식은 보통 다음과 같다: "이벤트 [가드 조건] / 동작". 가드 조건과 동작이 없거나, 테스터와 관련이 없는 경우 생략될 수 있다.

상태 테이블은 상태 전이 다이어그램을 다르게 표현한 모델이다. 행은 상태를 나타내고, 열은 이벤트를 (가드 조건이 있는 경우 함께) 나타낸다. 테이블의 각 항목(셀, cell)은 전이를 나타내며, 목표 상태는 물론 가드 조건, 정의된 경우에는 결과 동작도 포함한다. 상태 전이 다이어그램과 달리 상태 테이블은 유효하지 않은 전이를 빈 셀로 명확하게 보여준다.

상태 전이 다이어그램이나 상태 테이블을 기반으로 하는 테스트 케이스는 보통 일련의 이벤트 순서와 그 결과로 생기는 상태 변화로 (그리고 필요하다면 동작으로도) 표현된다. 하나의 테스트 케이스는 대부분의 경우 여러 개의 상태 전이를 포함한다.

상태 전이 테스팅에는 여러 가지 커버리지 측정 기준이 있다. 이 실러버스는 그 중에서 다음 세 가지를 설명한다:

모든 상태 커버리지(All state coverage)에서 커버리지 항목은 상태들이다. 모든 상태 커버리지 100%를 달성하려면, 테스트 케이스로 모든 상태를 확인해야 한다. 커버리지는 확인한 상태 수를 상태 총수로 나눈 값으로 측정하고 백분율로 표시한다.

유효 전이 커버리지(Valid transitions coverage, 0-스위치 커버리지라고도 함)에서는 각 유효 전이가 커버리지 항목이 된다. 유효 전이 커버리지 100%를 달성하려면, 테스트 케이스가 모든 유효 전이를 실행해야 한다. 커버리지는 실행된 유효 전이 수를 총 유효 전이 수로 나눈 값으로 측정하며 백분율로 표시한다.

모든 전이 커버리지(All transitions coverage)에서 커버리지 항목은 상태 테이블에 표시된 모든 전이들이다. 모든 전이 커버리지 100%를 달성하려면 테스트 케이스로 모든 유효 전이를 실행하고, 유효하지 않은 비유효 전이의 실행도 시도해야 한다. 하나의 테스트 케이스에서 테스트하는 비유효 전이를 하나로 제한하면 결함 마스킹(fault masking), 즉 하나의 결함으로 인해 다른 결함을 식별하지 못하는 상황을 방지하는 데 도움이 된다. 커버리지는 실행된 테스트 케이스로 수행하거나, 커버하려고 시도한 유효 및 비

유효 전이 수를 총 유효 및 비유효 전이 수로 나눈 값으로 측정하며 백분율로 표시한다.

모든 상태 커버리지는 일반적으로 모든 전이를 실행하지 않고도 달성할 수 있기 때문에, 유효 전이 커버리지보다 약하다. 유효 전이 커버리지는 가장 널리 사용하는 커버리지 조건이다. 유효 전이 커버리지 100%를 달성하면, 모든 상태 커버리지도 달성된다. 모든 전이 커버리지를 100% 달성하면, 모든 상태 커버리지와 유효 전이 커버리지도 모두 보장된다. 미션(mission)과 안전에 치명적인 소프트웨어의 경우, 이것을 충족해야 할 최소 기준으로 삼아야 한다.

4.3. 화이트박스 테스트 기법

이번 절은 코드를 대상으로 한 화이트박스 테스트 기법 중, 간단하면서 널리 사용하는 아래 두 가지를 다룬다:

- 구문 테스팅
- 분기 테스팅

안전이나 미션에 치명적이거나 무결성이 높아야 하는 환경에서 더욱 철저한 코드 커버리지를 달성하기 위해 사용하는 더 엄격한 기법도 있다. 또한, 상위 테스트 레벨(예: API 테스트)에 사용하거나, 코드와 관련 없는 커버리지(예: 신경망 테스트에서 뉴런 커버리지)를 사용하는 화이트박스 테스트 기법도 있다. 이런 기법은 이 실러버스에서 다루지 않는다.

4.3.1. 구문 테스팅과 구문 커버리지

구문 테스팅에서 커버리지 항목은 실행 가능한 구문이 된다. 목적은 코드 구문을 실행하는 테스트 케이스를 설계해서 허용할 수 있는 수준의 커버리지를 달성하는 것이다. 커버리지는 테스트 케이스가 실행한 구문 수를 코드의 실행 가능한 구문 총수로 나누어 계산하며 백분율로 표시한다.

100% 구문 커버리지를 달성하면 코드의 모든 실행 가능한 구문을 적어도 한 번은 실행했다는 것이 보장된다. 이는 결함이 있는 모든 구문도 실행해 결함 존재를 식별할 수 있는 장애가 나타났어야 함을 의미한다고 여길 수 있다. 그러나 테스트 케이스로 구문을 실행했다고 해서 결함이 반드시 식별되는 것은 아니다. 예를 들어, 데이터에 종속적인 결함(예: 분모를 0으로 설정한 경우에만 실패하는 나눗셈)은 식별하지 못할 수 있다. 또한, 100% 구문 커버리지가 모든 결정 논리를 테스트했다고 보장하는 것도 아니다. 예를 들어, 코드의 모든 분기(4.3.2 참조)가 실행되지 않았을 수 있다.

4.3.2. 분기 테스팅과 분기 커버리지

분기는 제어 흐름 그래프에서 두 노드(nodes) 간 제어의 이동으로 테스트 대상에서 소스 코드 구문의 실행 가능 순서를 보여준다. 제어의 이동은 조건 없이(즉, 직선적 코드(straight-line code)) 또는 조건에

따라 이루어질 수 있다(즉, 결정문 결과).

분기 테스팅에서 커버리지 항목은 분기이며, 목적은 코드의 분기를 실행하는 테스트 케이스를 설계해서 허용할 수 있는 수준의 커버리지를 달성하는 것이다. 커버리지는 테스트 케이스가 실행한 분기 수를 분기 총수로 나눈 값으로 측정하며 백분율로 표시한다.

100% 분기 커버리지를 달성하면 코드의 모든 분기(무조건 및 조건 분기)를 테스트 케이스로 실행하게 된다. 조건 분기는 일반적으로 "if...then" 결정문의 참 또는 거짓 결과, 스위치/케이스 구문의 결과, 또는 루프(loop)에서 종료 또는 계속 여부의 결정에 부합한다. 그러나 테스트 케이스로 분기를 실행한다고 해서 반드시 결함을 식별할 수 있는 것은 아니다. 예를 들어, 코드의 특정 경로를 실행해야 하는 결함은 감지하지 못할 수 있다.

분기 커버리지는 구문 커버리지를 포함한다. 즉, 100% 분기 커버리지를 달성하는 테스트 케이스 집합은 100% 구문 커버리지도 달성하지만, 그 반대의 경우는 성립하지 않는다.

4.3.3. 화이트박스 테스팅의 가치

모든 화이트박스 기법이 가진 근본적인 강점은 테스트 중 전체 소프트웨어 구현을 고려하므로 소프트웨어 명세가 모호하거나 뒤떨어지고 불완전한 경우에도 결함을 쉽게 감지할 수 있다는 점이다. 반면에 소프트웨어가 하나 이상의 요구사항을 구현하지 않는 경우, 화이트박스 테스팅으로 그것이 누락됐다는 결함을 식별하지 못할 수 있다는 단점이 있다(Watson 1996).

화이트박스 기법은 정적 테스팅(예: 코드 드라이 런 중)에 사용할 수 있다. 아직 실행할 준비가 되지 않은 코드(Hetzel 1988) 외에도 슈도 코드(pseudocode) 또는 제어 흐름 그래프로 모델링할 수 있는 기타 상위 수준 및 하향식(top-down) 논리를 검토하는 데 적합하다.

블랙박스 테스팅만 수행해서 실제 코드 커버리지 측정치를 얻을 수 없다. 화이트박스 커버리지 측정치는 객관적인 커버리지 측정값을 제공하고 커버리지를 높이기 위해 추가로 필요한 테스트를 만들기 위해 필요한 정보를 제공해서 결국 코드 신뢰도를 높일 수 있게 해준다.

4.4. 경험 기반 테스트 기법

여기에서 다룰 널리 활용하는 경험 기반 테스트 기법은 다음과 같다:

- 오류 추정
- 탐색적 테스팅
- 체크리스트 기반 테스팅

4.4.1. 오류 추정

오류 추정이란 테스터의 지식을 기본으로 오류, 결함, 장애 발생을 예측하는 데 사용하는 기법이다. 테스터의 지식에는 다음이 포함된다:

- 애플리케이션의 과거 동작
- 개발자가 범하기 쉬운 오류 유형과 이런 오류로 인해 발생하는 결함 유형
- 다른 유사 애플리케이션에서 발생한 장애 유형

일반적으로 오류, 결함, 장애는 입력(예: 올바른 입력을 인식하지 못함, 매개변수 오류 또는 누락), 출력(예: 잘못된 형식, 잘못된 결과), 논리(예: 사례 누락, 잘못된 연산자), 계산(예: 잘못된 피연산자, 잘못된 계산), 인터페이스(예: 매개변수 불일치, 호환되지 않는 유형), 데이터(예: 잘못된 초기화, 잘못된 유형)와 관련이 있을 수 있다.

결함 공격은 오류 추정을 구현하는 체계적인 접근법이다. 이 기법은 테스터가 발생 가능한 오류, 결함, 장애 목록을 만들거나 획득해서 오류와 관련된 결함을 식별 및 노출하거나, 장애를 유발하는 테스트를 설계하도록 한다. 이런 목록은 경험, 결함 및 장애 데이터 또는 소프트웨어에 문제가 생기는 원인에 관한 일반적인 지식을 기반으로 작성할 수 있다.

오류 추정과 결함 공격에 대한 자세한 내용은 Whittaker 2002, Whittaker 2003, Andrews 2006을 참조할 수 있다.

4.4.2. 탐색적 테스팅

탐색적 테스팅에서는 테스터가 테스트 대상에 대해 배워가면서 테스트의 설계, 실행, 평가를 동시에 하게 된다. 이 테스팅은 테스트 대상에 대해 더 배우고, 집중하고 있는 테스트는 더 깊이 탐색하고, 테스트되지 않은 영역에 대한 테스트를 만들기 위해 사용된다.

탐색적 테스팅에서 테스트를 체계적으로 수행하기 위해 세션 기반 테스팅을 사용하기도 한다. 세션 기반 접근법을 활용하면 탐색적 테스팅을 시간을 정해 놓고 수행하게 된다. 테스터는 테스트 목적이 정의된 테스트 차터(charter)를 테스팅 지침으로 사용한다. 일반적으로 테스트 세션이 끝나면 보고가 이어진다. 이때 테스터와 테스트 세션 결과에 관심이 있는 이해관계자 간의 토론이 이루어진다. 이 접근법에선 테스트 목적을 상위 수준의 테스트 컨디션으로 취급할 수 있다. 커버리지 항목은 테스트 세션 중에 식별하고 실행한다. 테스터가 수행 절차와 발견 내용을 기록하기 위해 테스트 세션 시트를 사용할 수 있다.

탐색적 테스팅은 명세가 부족하거나 부적합할 경우, 또 테스트에 시간적 압박이 심할 때 유용하다. 탐색적 테스팅은 다른 공식 테스트 기법을 보완하기에도 유용하다. 테스터가 풍부한 경험과 도메인 지식을 가지고 있고 높은 수준의 분석 기술, 호기심, 창의성 등 필요 기술을 갖춘 경우 탐색적 테스팅이 더욱 효과적일 수 있다(1.5.1 참조).

탐색적 테스팅 중 다른 테스트 기법(예: 동등 분할)을 사용할 수 있다. 탐색적 테스팅에 대한 자세한 내용은 Kaner 1999, Whittaker 2009, Hendrickson 2013에서 확인할 수 있다.

4.4.3. 체크리스트 기반 테스팅

체크리스트 기반 테스팅에서 테스터는 체크리스트를 활용해 테스트 컨디션을 확인하는 테스트를 설계, 구현, 실행한다. 체크리스트는 경험, 사용자에게 중요한 것이 무엇인지에 대한 지식, 또는 소프트웨어가 실패하는 이유와 방법에 대한 이해를 바탕으로 작성할 수 있다. 자동으로 점검할 수 있는 항목, 시작/종료 조건으로 더 적합한 항목, 너무 일반적인 항목은 체크리스트에 포함해서는 안 된다. (Brykczynski 1999)

체크리스트 항목을 질문 형식으로 표현하는 경우가 많다. 각 항목은 개별적이면서 직접적으로 확인할 수 있어야 한다. 이런 항목은 요구사항, 그래픽 인터페이스 속성, 품질 특성 또는 기타 유형의 테스트 컨디션일 수 있다. 체크리스트는 기능 및 비기능 테스트를 포함한 다양한 테스트 유형을 지원하기 위해 만들 수 있다(예: 사용성 테스트를 위한 10가지 휴리스틱 평가 기준(Nielsen 1994)).

시간이 지남에 따라 개발자가 같은 오류를 범하지 않게 됨으로 일부 체크리스트 항목은 점차 효과가 떨어질 수 있다. 새로 발견한 심각도가 높은 결함으로 인해 새로운 항목을 추가해야 할 수도 있다. 따라서 체크리스트는 결함 분석을 기반으로 정기적으로 업데이트해야 한다. 하지만 체크리스트가 너무 길어지지 않도록 주의할 필요가 있다. (Gawande 2009)

구체적인 테스트 케이스가 없는 경우에 체크리스트 기반 테스팅은 테스트를 위한 지침과 일정 수준의 일관성을 제공할 수 있다. 체크리스트가 상위 수준으로 작성된 경우 실제 테스트는 조금씩 달라질 수 있으며, 결국 커버리지는 높아지지만 재현 가능성은 떨어질 수 있다.

4.5. 협업 기반 테스트 접근법

위에서 언급한 각 기법(4.2, 4.3, 4.4 참조)은 결함 식별과 관련해 특정 목표를 가진다. 반면 협업 기반 접근법은 협업과 커뮤니케이션을 통한 결함 예방에도 초점을 둔다.

4.5.1. 협업 기반 사용자 스토리 작성

사용자 스토리는 시스템이나 소프트웨어의 사용자 또는 구매자에게 가치를 제공하는 기능을 나타낸다. 사용자 스토리는 다음 세 가지 중요 요소를 가지고 있으며(Jeffries 2000), 이를 합쳐서 '3C'라고 부른다.

- 카드(Card) - 사용자 스토리를 설명하는 매체(예: 인덱스 카드, 전자 게시판 항목)
- 대화(Conversation) - 소프트웨어 사용 방법에 대한 설명(문서 또는 구두로)
- 확인(Confirmation) - 인수 조건(4.5.2 참조)

사용자 스토리의 가장 일반적인 형식은 "[역할]로서 [목표]를 달성해 [역할이 얻게 될 비즈니스 가치]를

얻기를 원한다."이며, 이 후 인수 조건이 뒤따르는 형식이다.

협업 기반 사용자 스토리 작성은 브레인스토밍, 마인드 매핑과 같은 기법을 사용한다. 협업을 통해 팀원들은 비즈니스, 개발, 테스팅의 세 가지 관점을 고려해 만들어서 전달할 것에 대한 공유된 비전을 얻을 수 있다.

좋은 사용자 스토리는 독립적(Independent)이고, 협상 가능(Negotiable)하고, 가치 있고(Valuable), 추정 가능(Estimable)하고, 작고(Small), 테스트 가능(Testable)해야 한다(INVEST). 이해관계자가 사용자 스토리를 테스트하는 방법을 모른다면, 그 사용자 스토리가 명확하지 않거나 사용자에게 중요한 내용이 반영되어 있지 않거나, 이해관계자가 테스트를 수행하는 데 있어 도움이 필요하다는 것을 의미할 수 있다 (Wake 2003).

4.5.2. 인수 조건

사용자 스토리의 인수 조건은 사용자 스토리 구현 결과를 이해관계자가 승인하기 위해 충족되어야 하는 조건이다. 이런 관점에서 인수 조건을 테스트해야 하는 테스트 컨디션으로 볼 수 있다. 인수 조건은 보통 3C 중 대화(Conversation)를 통해 결정된다(4.5.1 참조).

인수 조건은 다음을 위해 사용된다:

- 사용자 스토리 범위 정의
- 이해관계자 간 합의 도출
- 긍정과 부정 시나리오 설명
- 사용자 스토리 인수 테스팅의 베이시스 제공(4.5.3 참조)
- 정확한 계획 및 추정

다양한 사용자 스토리의 인수 조건 작성법이 있으며, 가장 일반적인 두 가지는 다음과 같다:

- 시나리오 기반(예: 행위 주도 개발에서 사용하는 Given/When/Then 형식, 2.1.3 참조)
- 규칙 기반(예: 베리피케이션이 필요한 목록 또는 표로 표현된 입력-출력 매핑)

대부분의 인수 조건은 이 두 가지 형식 중 하나로 문서화할 수 있다. 그러나 팀이 다른 자체 형식을 사용하기로 할 수도 있으며, 이때도 인수 조건이 잘 정의되어 모호하지 않아야 한다.

4.5.3. 인수 테스트 주도 개발(ATTD)

인수 테스트 주도 개발(ATTD, Acceptance Test-driven Development)은 테스트 우선 접근법이다(2.1.3 참조). 테스트 케이스는 사용자 스토리 구현 전에 만들어진다. 고객, 개발자, 테스터 등 서로 다른 관점을 가진 팀원들이 테스트 케이스를 만든다(Adzic 2009). 테스트 케이스는 수동 또는 자동으로 실행할 수 있다.

첫 번째 단계는 명세 워크숍으로 팀원들은 여기서 사용자 스토리와 (아직 정의되지 않은 경우) 인수 조건을

분석하고 토론해서 작성한다. 이 과정에서 사용자 스토리의 불완전성, 모호성, 결함을 해결하게 된다. 다음 단계는 테스트 케이스를 만드는 것이다. 이 작업은 팀 전체가 수행하거나 테스터가 개별적으로 수행할 수 있다. 테스트 케이스는 인수 조건을 기반으로 하며, 소프트웨어가 어떻게 작동하는지에 대한 예제로 볼 수 있다. 이는 팀이 사용자 스토리를 올바르게 구현하는 데 도움을 준다.

여기서 예제와 테스트는 같은 의미를 가지기 때문에 혼용해서 사용하기도 한다. 테스트 설계 시 4.2, 4.3, 4.4에서 설명한 테스트 기법을 적용할 수 있다.

일반적으로 첫 번째 테스트 케이스는 예외나 오류 조건 없이 올바른 동작을 확인하고 모든 것이 예상대로 진행될 경우 실행되는 일련의 활동으로 구성된 긍정/유효 테스트 케이스이다. 유효 테스트 케이스를 끝내고 나면 팀은 비유효/부정 테스트를 수행해야 한다. 마지막으로 팀은 비기능 품질 특성(예: 성능 효율성, 사용성)도 다루어야 한다. 테스트 케이스는 이해관계자가 이해할 수 있는 방식으로 표현되어야 한다. 일반적으로 테스트 케이스는 (있는 경우) 필요한 전제 조건, 입력값, 사후 조건을 포함하며 자연어 문장으로 구성한다.

테스트 케이스는 사용자 스토리의 모든 특성을 다뤄야 하며 스토리를 벗어나면 안 된다. 그러나 인수 조건이 사용자 스토리가 얘기하는 문제의 일부를 구체적으로 설명할 수도 있다. 또한, 두 개 이상의 테스트 케이스가 사용자 스토리의 같은 특성을 설명해서는 안 된다.

테스트 자동화 프레임워크가 지원하는 형식으로 작성하면 개발자는 사용자 스토리에서 설명하는 기능을 구현할 때 필요한 코드를 작성해 테스트 케이스를 자동화할 수 있다. 그러면 인수 테스트가 실행 가능한 요구사항이 된다.

제 5 장 테스트 활동 관리 | 335분

용어

결함 관리(defect management), 결함 보고서(defect report), 시작 조건(entry criteria), 완료 조건(exit criteria), 제품 리스크(product risk), 프로젝트 리스크(project risk), 리스크(risk), 리스크 분석(risk analysis), 리스크 평가(risk assessment), 리스크 제어(risk control), 리스크 식별(risk identification), 리스크 수준(risk level), 리스크 관리(risk management), 리스크 완화(risk mitigation), 리스크 모니터링(risk monitoring), 리스크 기반 테스팅(risk-based testing), 테스트 접근법(test approach), 테스트 완료 보고서(test completion report), 테스트 제어(test control), 테스트 모니터링(test monitoring), 테스트 계획서(test plan), 테스트 계획(test planning), 테스트 진행 보고서(test progress report), 테스트 피라미드(test pyramid), 테스팅 사분면(testing quadrants)

5장 학습 목표

5.1 테스트 계획

FL-5.1.1 (K2) 테스트 계획서의 목적과 내용을 예를 들어 설명할 수 있다

FL-5.1.2 (K1) 테스터가 반복 주기와 릴리스 계획에 가치를 더하는 방법을 인식할 수 있다

FL-5.1.3 (K2) 시작 조건과 완료 조건을 비교 및 대조할 수 있다

FL-5.1.4 (K3) 필요한 테스트 노력의 계산을 위해 추정 기법을 활용할 수 있다

FL-5.1.5 (K3) 테스트 케이스에 우선순위를 적용할 수 있다

FL-5.1.6 (K1) 테스트 피라미드의 개념을 상기할 수 있다

FL-5.1.7 (K2) 테스팅 사분면과 그것이 테스트 레벨 및 유형과 갖는 관계를 요약할 수 있다

5.2 리스크 관리

FL-5.2.1 (K1) 리스크 발생 가능성과 리스크 영향도를 활용해 리스크 수준을 식별할 수 있다

FL-5.2.2 (K2) 프로젝트 리스크와 제품 리스크를 구별할 수 있다

FL-5.2.3 (K2) 제품 리스크 분석이 테스팅의 강도 및 범위에 어떻게 영향을 미치는지 설명할 수 있다

FL-5.2.4 (K2) 분석한 제품 리스크에 대응해 어떤 조치를 취할 수 있는지 설명할 수 있다

5.3 테스트 모니터링, 테스트 제어, 테스트 완료

FL-5.3.1 (K1) 테스팅에 사용하는 메트릭을 상기할 수 있다

FL-5.3.2 (K2) 테스트 보고서의 목적, 내용, 대상을 요약할 수 있다

FL-5.3.3 (K2) 테스팅 상황을 전달하는 방법을 예를 들어 설명할 수 있다

5.4 형상 관리

FL-5.4.1 (K2) 형상 관리가 테스팅을 어떻게 지원하는지 요약할 수 있다

5.5 결함 관리

FL-5.5.1 (K3) 결함 보고서를 작성할 수 있다

5.1. 테스트 계획

5.1.1. 테스트 계획서의 목적과 내용

테스트 계획서는 테스트 프로젝트의 목적, 자원, 프로세스를 설명한다. 테스트 계획서는:

- 테스트 목적 달성을 위한 방법과 일정을 문서화한다.
- 수행한 테스트 활동이 정해진 기준을 충족하는 데 도움을 준다.
- 팀원과 기타 이해관계자의 의사소통 수단으로 사용된다.
- 테스팅이 수립한 테스트 정책 및 전략을 준수함을 보여준다(또는 테스팅이 그것을 준수하지 않는 이유를 설명한다).

테스트 계획 활동은 테스터가 리스크, 일정, 인력, 도구, 비용, 노력 등 향후 문제를 고민하도록 한다. 테스트 계획서를 준비하는 과정은 테스트 프로젝트의 목적을 달성하는 데 필요한 노력을 추론하는 유용한 방법이다.

테스트 계획서는 보통 다음과 같은 내용을 포함한다:

- 테스팅 정황(예: 범위, 테스트 목적, 제약 사항, 테스트 베이시스)
- 테스트 프로젝트의 가정 및 제약 사항
- 이해관계자(예: 역할, 책임, 테스팅 관련성, 채용 및 훈련 요구사항)
- 의사소통(예: 의사소통 방법 및 빈도, 문서 양식)
- 리스크 목록(예: 제품 리스크, 프로젝트 리스크)
- 테스트 접근법(예: 테스트 레벨, 테스트 유형, 테스트 기법, 테스트 산출물, 시작 조건 및 완료 조건, 테스팅의 독립성, 수집할 메트릭, 테스트 데이터 요구사항, 테스트 환경 요구사항, 조직의 테스트 정책 및 테스트 전략과의 편차)
- 예산 및 일정

테스트 계획과 세부 내용에 대한 추가적인 정보는 ISO/IEC/IEEE 29119-3 표준에서 확인할 수 있다.

5.1.2. 반복 주기와 릴리스 계획에 대한 테스터의 기여

일반적으로 반복적 소프트웨어 개발수명주기(SDLC)에서 두 가지의 계획, 즉 릴리스 계획과 반복 주기 계획이 이루어진다.

릴리스 계획은 제품 릴리스를 계획하는 단계로 제품 백로그를 (재)정의하며, 큰 사용자 스토리를 작은 사용자 스토리들로 세분화하는 작업을 포함할 수 있다. 또한, 개별 반복 주기의 테스트 접근법과 테스트 계획을 위한 기반이 된다. 릴리스 계획에 참여하는 테스터는 테스트 가능한 사용자 스토리와 인수 조건이 작성되도록 하고(4.5 참조), 프로젝트 및 품질 리스크 분석에 참여하고(5.2 참조), 사용자 스토리와 관련된 테스트 노력을 추정하고(5.1.4 참조), 테스트 접근법을 결정하고, 릴리스를 위한 테스팅을 계획한다.

반복 주기 계획은 개별 반복 주기를 계획하는 것으로 반복 주기 백로그와 관련이 있다. 반복 주기 계획에 참여하는 테스터는 사용자 스토리의 구체적인 리스크 분석에 참여하고, 사용자 스토리의 테스트 용이성을 판단하고, 사용자 스토리를 업무(특히 테스팅 업무) 단위로 분류하고, 각 테스팅 업무에 필요한 테스트 노력을 추정하고, 테스트 대상의 기능 및 비기능적 측면을 식별하고 구체화한다.

5.1.3. 시작 조건과 완료 조건

시작 조건은 어떤 활동을 수행하기 위한 전제 조건을 정의한다. 시작 조건을 충족하지 못하면 해당 활동을 수행하는 것이 어려워져 들어가는 시간과 비용이 늘어나고 원활하게 진행되지 않을 위험도 높아진다. 완료 조건은 특정 활동의 종료를 선언하기 위해 달성해야 하는 사항을 정의한다. 시작 조건과 완료 조건은 각 테스트 레벨에 대해 정의해야 하며 테스트 목적에 따라 달라진다.

많이 사용하는 시작 조건으로는 자원의 가용성(예: 인력, 도구, 환경, 테스트 데이터, 예산, 시간), 테스트웨어 가용성(예: 테스트 베이시스, 테스트 가능한 요구사항, 사용자 스토리, 테스트 케이스), 테스트 대상의 초기 품질 수준(예: 모든 스모크 테스트가 합격함) 등이 있다.

완료 조건으로 널리 활용하는 것으로는 충분함(thoroughness)에 대한 측정(예: 달성한 커버리지 수준, 해결되지 않은 결함 수, 결함 밀도, 실패한 테스트 케이스 수)과 종료 기준(예: 계획한 모든 테스트 실행, 정적 테스트 수행, 발견한 모든 결함 보고, 모든 리그레션 테스트 자동화 완료) 등이 있다.

시간 및 예산의 소진도 유효한 완료 조건으로 볼 수 있다. 이때 이해관계자들이 추가 테스트 없이 배포하는 리스크를 검토하고 수용했다면, 다른 완료 조건이 충족되지 않더라도 테스트를 종료할 수 있다.

애자일 소프트웨어 개발에서 완료 조건을 완료의 정의(Definition of Done)라고 하는 경우가 많으며, 릴리스 가능 항목에 대한 팀의 목표 메트릭을 정의하게 된다. 개발 및 테스트 활동을 시작하기 위해 사용자 스토리가 충족해야 하는 시작 조건을 준비의 정의(Definition of Ready)라고 한다.

5.1.4 추정 기법

테스트 노력 추정은 테스트 프로젝트의 목적을 달성하는 데 필요한 테스트 관련 작업량을 예측하는 것이다. 추정치는 여러 가정을 기반으로 하며, 추정에는 항상 오류가 있을 수 있다는 점을 이해관계자가 명확히 이해하도록 하는 것이 중요하다. 보통은 규모가 큰 작업보다 작은 작업에 대한 추정치가 정확하다. 따라서 큰 작업에 대해 추정할 때는 우선 여러 개의 작은 작업으로 나눈 다음 추정하게 된다.

이 실러버스는 다음 네 가지 추정 기법을 소개한다.

비율 기반 추정(Estimation based on ratios). 메트릭 기반 기법으로 조직에서 수행한 이전 프로젝트 수치를 수집해 유사 프로젝트를 위한 "표준" 비율을 도출한다. 보통 조직에서 직접 수행한 프로젝트에서 수집한 비율(예: 과거 데이터에서 가져온)이 추정 과정에 사용하기 가장 좋은 자료이다. 이런 표준 비율을 가지고 새로운 프로젝트에 필요한 테스트 노력을 추정할 수 있다. 예를 들어, 이전 프로젝트에서 개발

노력 대 테스트 노력의 비율이 3:2였고, 현재 프로젝트에서 개발 노력을 600MD로 예상한다면, 테스트 노력은 400MD로 추정할 수 있다.

외삽법(Extrapolation). 메트릭 기반 기법으로 현재 프로젝트에서 데이터 수집을 위해 가능한 한 빨리 측정을 수행한다. 관찰 결과가 충분히 쌓이면 이 데이터를 가지고 외삽법(보통 수학적 모델 사용)을 적용해 남은 작업에 필요한 노력의 근사치를 추정할 수 있다. 이 방법은 반복적 소프트웨어 개발수명주기(SDLC)에 매우 적합하다. 예를 들어, 팀이 지난 세 번의 반복 주기에 들인 평균 노력으로 다음 반복 주기의 테스트 노력을 추정할 수 있다.

와이드밴드 델파이(Wideband Delphi). 반복적, 전문가 기반 기법으로 전문가의 경험을 기반으로 추정을 한다. 각 전문가는 독립적으로 노력을 추정한다. 결과를 수집해서 합의된 범위를 벗어난 편차가 있다면 전문가들이 각자의 추정치에 대해 논의한다. 그런 다음 각 전문가에게 논의 결과를 바탕으로 다시 한번 새로운 추정치를 제시하도록 한다. 합의에 도달할 때까지 이 과정을 반복한다. 플래닝 포커(Planning Poker)는 애자일 소프트웨어 개발에서 많이 사용하는 와이드밴드 델파이의 변형이다. 플래닝 포커는 보통 노력의 규모를 나타내는 숫자가 적힌 카드를 사용해 추정한다.

3점 추정(Three-point estimation). 전문가 기반 기법으로 전문가가 세 개의 추정치를 도출한다. 추정치는 가장 낙관적인 추정치(a), 확률적으로 가장 높은 추정치(m), 가장 비관적인 추정치(b)이다. 최종 추정치(E)는 이들의 가중 산술 평균이 된다. 가장 널리 사용하는 계산식은 E=(a+4*m+b)/6이다. 이 기법의 장점은 전문가가 측정 오류를 계산할 수 있다는 것이다. SD(표준 편차, Standard deviation)=(b-a)/6이 된다. 예를 들어, 추정치(MH)가 a=6, m=9, b=18인 경우 E=(6+4*9+18)/6=10이고, SD=(18-6)/6=2이므로 최종 추정치는 10±2(즉, 8과 12 사이의 맨아워)가 된다.

이런 기법과 기타 여러 테스트 추정 기법에 관한 정보는 Kan 2003, Koomen 2006, Westfall 2009를 참조할 수 있다.

5.1.5. 테스트 케이스 우선순위지정

테스트 케이스와 테스트 절차를 도출해서 테스트 스위트로 조립하고 나면 테스트 스위트를 테스트 일정으로 구성할 수 있다. 테스트 일정은 테스트 실행 순서를 정의한다. 테스트 케이스의 우선순위를 정할 때 다양한 요소를 고려할 수 있다. 많이 사용하는 테스트 케이스 우선순위지정법은 다음과 같다:

- 리스크 기반 우선순위지정: 리스크 분석 결과에 따라 테스트 실행 순서를 결정한다(5.2.3 참조). 가장 중요한 리스크를 다루는 테스트 케이스를 먼저 실행한다.
- 커버리지 기반 우선순위지정: 테스트 실행 순서를 커버리지(예: 구문 커버리지)에 따라 결정한다. 가장 높은 커버리지를 달성하는 테스트 케이스를 먼저 실행한다. "추가 커버리지 우선순위지정" 이라는 다른 변형에서는, 가장 높은 커버리지를 달성하는 테스트 케이스가 먼저 실행된다; 이후에 실행되는 각각의 테스트 케이스는 가장 높은 추가 커버리지를 달성하는 것이다.

- 요구사항 기반 우선순위지정: 테스트 실행 순서를 해당 테스트 케이스의 기반이 되는 요구사항의 우선순위에 따라 결정한다. 요구사항의 우선순위는 이해관계자가 정의한다. 가장 중요한 요구사항 관련 테스트 케이스를 먼저 실행한다.

위에서 언급한 3가지 중 하나, 또는 여러 전략을 사용해 우선순위를 지정해서 테스트 케이스 실행 순서를 도출하는 것이 이상적이다. 그러나 테스트 케이스 또는 테스트 대상 기능이 종속성을 가진 경우 그렇게 하기 힘들 수 있다. 우선순위 높은 테스트 케이스가 우선순위 낮은 테스트 케이스에 종속되는 경우, 우선순위가 낮은 테스트 케이스를 먼저 실행해야 할 수도 있다.

테스트 실행 순서에서는 자원의 가용성도 고려해야 한다. 예를 들어, 필요한 테스트 도구, 테스트 환경, 인력이 가용한 시기가 따로 있을 수 있다.

5.1.6. 테스트 피라미드

테스트 피라미드는 테스트에 따라 입도(granularity)가 다를 수 있다는 것을 보여주는 모델이다. 테스트 피라미드 모델은 테스트 자동화 수준에 따라 지원하는 목표가 다를 수 있다는 것을 보여줌으로써 팀의 테스트 자동화와 테스트 노력 할당을 도와준다. 피라미드의 각 층은 하나의 테스트 그룹을 나타낸다. 층이 올라갈수록 테스트 입도가 낮고, 테스트 격리가 어려워지며, 테스트 실행 시간도 길어진다. 아래층에 있는 테스트는 규모가 작고, 독립적이며, 빠르고, 대상이 되는 기능이 작기 때문에 적절한 커버리지를 달성하기 위해 많은 테스트가 필요한 경우가 많다. 높은 층은 복잡하고 상위 수준의 엔드-투-엔드 테스트를 대변한다. 이런 상위 수준 테스트는 보통 하위층의 테스트보다 속도가 느리며, 큰 규모의 기능을 확인하기 때문에 적은 수로 적절한 커버리지를 달성할 수 있다. 층의 수와 각 층의 명칭은 다를 수 있다. 예를 들어, 최초의 테스트 피라미드 모델(Cohn 2009)은 "단위 테스트", "서비스 테스트", "UI 테스트"라는 세 개의 층을 정의했다. 단위(컴포넌트) 테스트, 통합(컴포넌트 통합) 테스트, 엔드-투-엔드 테스트로 정의하는 모델도 많이 알려져 있다. 또한 다른 테스트 레벨 분류(섹션 2.2.1. 참조)를 사용하기도 한다.

5.1.7. 테스팅 사분면

브라이언 마릭(Brian Marick)(Marick 2003, Crispin 2008)이 정의한 테스팅 사분면은 애자일 소프트웨어 개발에서 테스트 레벨을 적합한 테스트 유형, 활동, 테스트 기법, 작업 산출물과 묶고 있다. 이 모델은 이런 사항들을 시각화해서 테스트 관리자가 필요한 모든 테스트 유형과 테스트 레벨을 소프트웨어 개발수명주기(SDLC)에 포함하고, 테스트 레벨에 따라 테스트 유형 별 연관성이 다르다는 점을 이해하는 데 도움을 준다. 또한, 이 모델은 개발자, 테스터, 비즈니스 담당자 등 모든 이해관계자에게 테스트 유형을 구분하고 설명하는 방법을 제공한다.

이 모델에서 테스트는 비즈니스 또는 기술에 대한 테스트일 수 있다. 테스트는 팀을 지원하거나(즉, 개발을 위한 지침으로) 제품을 평가할 수도 있다(즉, 기대치 대비 동작 측정). 이 두 가지의 조합에 따라 네 개의 사분면이 결정된다:

- 1 사분면(기술 측면, 팀 지원). 이 사분면은 컴포넌트 및 컴포넌트 통합 테스트를 포함한다. 이런 테스트는 자동화해야 하며, 지속적인 통합(CI) 프로세스에 포함돼야 한다.
- 2 사분면(비즈니스 측면, 팀 지원). 이 사분면은 기능 테스트, 예제, 사용자 스토리 테스트, 사용자 경험 프로토타입, API 테스팅, 시뮬레이션 등을 포함한다. 이런 테스트는 인수 조건을 확인하며, 수동으로 실행하거나 자동화할 수 있다.
- 3 사분면(비즈니스 측면, 제품 평가). 이 사분면은 탐색적 테스팅, 사용성 테스팅, 사용자 인수 테스팅을 포함한다. 이런 테스트는 사용자를 중심으로 이루어지며, 수동으로 실행하는 경우가 많다.
- 4 사분면(기술 측면, 제품 평가). 이 사분면은 스모크 테스트와 비기능 테스트(사용성 테스트 제외)를 포함한다. 이런 테스트는 자동화되는 경우가 많다.

5.2. 리스크 관리

조직은 목표의 달성 여부와 시기를 불확실하게 만드는 여러 내부 및 외부 요인에 직면한다(ISO 31000). 리스크 관리는 조직이 목표를 달성할 가능성과 제품의 품질을 높이고, 이해관계자의 신뢰와 믿음을 얻을 수 있게 한다.

주요 리스크 관리 활동은 다음과 같다:

- 리스크 분석(리스크 식별과 리스크 평가로 구성, 5.2.3 참조)
- 리스크 제어(리스크 완화와 리스크 모니터링으로 구성, 5.2.4 참조)

리스크 분석과 리스크 제어를 기반으로 테스트 활동을 선택하고 우선순위를 정해 관리하는 테스트 접근법을 리스크 기반 테스팅이라고 한다.

5.2.1 리스크의 정의와 리스크의 속성

리스크는 발생 시 부정적인 영향을 미칠 수 있는 잠재적인 사건, 위험, 위협 또는 상황을 말한다. 리스크는 두 가지 요소로 특징지을 수 있다:

- 리스크 발생 가능성 - 리스크 발생 확률(0 보다 크고 1 보다 작음)
- 리스크 영향(피해) – 발생했을 때 생기게 될 피해

이 두 가지 요소로 리스크 수준을 표현한다. 리스크 수준은 리스크를 측정한 값이 된다. 리스크 수준이 높을수록 그에 대한 조치 또한 중요해진다.

5.2.2. 프로젝트 리스크와 제품 리스크

일반적으로 소프트웨어 테스팅에서 두 가지 유형의 리스크는 프로젝트 리스크나 제품 리스크와 관련이 있다.

프로젝트 리스크는 프로젝트 관리 및 제어와 관련이 있다. 프로젝트 리스크에는 다음이 포함된다:

- 조직 문제(예: 작업 산출물 인도 지연, 부정확한 추정, 예산 축소)
- 인력 문제(예: 기술 부족, 갈등, 의사소통 문제, 직원 부족)
- 기술적 문제(예: 협의되지 않은 개발 범위의 점진적 추가, 도구 지원 부족)
- 공급업체 문제(예: 제 3 자 인도 실패, 지원 기업의 파산)

프로젝트 리스크가 발생하면 프로젝트 일정, 예산, 범위에 영향을 끼쳐 프로젝트의 목표 달성 능력에 영향을 미칠 수 있다.

제품 리스크는(예: ISO 25010 품질 모델이 설명하는) 제품 품질 특성과 관련이 있다. 제품 리스크의 예로 누락 또는 잘못된 기능, 부정확한 계산, 런타임 오류, 열악한 아키텍처, 비효율적인 알고리즘, 부적절한 응답 시간, 열악한 UX(사용자 경험, user experience), 보안 취약점 등이 있다. 제품 리스크가 발현되면 다음과 같은 다양한 부정적인 결과를 초래할 수 있다:

- 사용자 불만족
- 매출, 신뢰, 평판 손실
- 제 3 자 피해
- 높은 유지보수 비용, 고객지원 부서의 과부하
- 형사 처벌
- 극단적인 경우 신체적 손상, 부상 또는 사망

5.2.3. 제품 리스크 분석

테스팅 관점에서 제품 리스크 분석의 목표는 제품 리스크를 인식해 잔존 제품 리스크 수준을 최소화하는 방향으로 테스트 노력을 집중할 수 있도록 하는 것이다. 제품 리스크 분석은 소프트웨어 개발수명주기(SDLC) 초기에 시작하는 것이 이상적이다.

제품 리스크 분석은 리스크 식별과 리스크 평가로 이루어진다. 리스크 식별은 포괄적인 리스크 목록을 생성하는 것이다. 이해관계자는 브레인스토밍, 워크숍, 인터뷰, 원인-결과 다이어그램 등 다양한 기법과 도구를 사용해 리스크를 식별할 수 있다. 리스크 평가는 식별한 리스크를 분류하고, 리스크 발생 가능성/영향/수준을 결정하고, 우선순위를 정해 조치 방법을 제안하는 작업을 포함한다. 같은 범주로 분류된 리스크는 유사한 접근법으로 완화할 수 있는 경우가 많기 때문에 완화 조치를 정하는 데 분류가 도움이 된다.

리스크 평가는 정량적 또는 정성적 접근법을 사용하거나, 이 두 가지를 혼합해 사용할 수 있다. 정량적

접근법에서 리스크 수준은 리스크 발생 가능성과 리스크 영향을 곱한 값으로 계산한다. 정성적 접근법은 리스크 행렬을 사용해 리스크 수준을 판단하기도 한다.

제품 리스크 분석은 테스팅의 강도와 범위에 영향을 미칠 수 있다. 분석 결과는 다음과 같이 활용한다:

- 테스팅 수행 범위 결정
- 테스트 레벨 결정 및 수행할 테스트 유형 제안
- 사용할 테스트 기법과 달성할 커버리지 결정
- 업무별 필요 테스트 노력 추정
- 중요 결함을 가능한 한 빨리 식별하기 위한 테스트 우선순위지정
- 리스크를 줄이기 위해 테스팅 외 다른 활동을 할 수 있는지 판단

5.2.4. 제품 리스크 제어

제품 리스크 제어는 식별 및 평가된 제품 리스크에 대응해 취하는 모든 조치를 말한다. 제품 리스크 제어는 리스크 완화와 리스크 모니터링으로 이루어진다. 리스크 완화는 리스크 평가 때 제안된 조치를 실행해 리스크 수준을 낮추는 활동을 말한다. 리스크 모니터링의 목적은 리스크 완화 조치가 효과적인지 확인하고, 또 리스크 평가 개선을 위해 추가로 필요한 정보를 확보하고, 새롭게 나타난 리스크를 식별하는 것이다.

제품 리스크 제어 측면에서 리스크를 분석하면 거기에 대응하기 위한 다양한 완화 방안을 수립할 수 있다. 예를 들어, 테스팅을 통한 리스크 완화, 리스크 수용, 리스크 전가, 대안 계획 등이 이루어질 수 있다 (Veenendaal 2012). 테스팅으로 제품 리스크 완화를 위해 취할 수 있는 조치는 다음과 같다:

- 주어진 리스크 유형에 적절한 경험과 기술을 갖춘 테스터 선정
- 적절한 수준의 테스팅 독립성 적용
- 리뷰 및 정적 분석 수행
- 적절한 테스트 기법 및 커버리지 수준 적용
- 영향을 받는 품질 특성을 다루는 적절한 테스트 유형 적용
- 리그레션 테스팅을 포함한 동적 테스팅 수행

5.3. 테스트 모니터링, 테스트 제어, 테스트 완료

테스트 모니터링은 테스팅에 대한 정보 수집과 관련이 있다. 이 정보는 테스트 진행 상황을 판단하고, 테스트 완료 조건 또는 완료 조건과 관련된 테스트 업무(예: 제품 리스크, 요구사항, 인수 조건 등의 커버리지 목표 달성)가 충족됐는지 측정하는 데 사용한다.

테스트 제어에서는 테스트 모니터링에서 얻은 정보로 가장 효과적이고 효율적인 테스팅을 위한 제어 지침/

지도/필요 수정 조치 등을 제공한다. 제어 지침의 예는 다음과 같다:

- 식별된 리스크가 발현될 경우 테스트 우선순위 재지정
- 재작업 이후 테스트 항목이 시작 조건 및 완료 조건을 충족하는지 재평가
- 테스트 환경 인도 지연에 대응하기 위한 테스트 일정 조정
- 필요한 지점과 시기에 신규 자원 추가

테스트 완료는, 끝난 테스트 활동에서 데이터를 수집하여 경험, 테스트웨어, 기타 관련 정보를 수집하는 단계다. 테스트 완료 활동은 어떤 프로젝트 마일스톤에 도달했을 때 이루어진다. 테스트 레벨이나 애자일 반복 주기의 끝, 테스트 프로젝트를 완료(또는 취소)한 시점, 소프트웨어를 배포했거나 유지보수 릴리스를 끝냈을 때 등이 여기에 해당한다.

5.3.1. 테스팅에 사용하는 메트릭

테스트 메트릭은 계획한 일정 및 예산 대비 현재 진행 상황, 테스트 대상의 현재 품질, 테스트 목적이나 반복 주기 목표 대비 테스트 활동의 효과를 보여주기 위해 수집한다. 테스트 모니터링은 테스트 제어 및 테스트 완료 활동을 지원하는 다양한 메트릭을 수집한다.

많이 사용하는 테스트 메트릭은 다음과 같다:

- 프로젝트 진행 상황 메트릭(예: 작업 완료율, 자원 사용률, 테스트 노력 투입률)
- 테스트 진행 상황 메트릭(예: 테스트 케이스 구현 진행률, 테스트 환경 준비 진행률, 실행/미실행 및 합격/불합격 테스트 케이스 수, 테스트 실행 시간)
- 제품 품질 메트릭(예: 가용성, 응답 시간, 평균 장애 시간)
- 결함 메트릭(예: 발견/수정한 결함의 수와 우선순위, 결함 밀도, 결함 발견 비율)
- 리스크 메트릭(예: 잔여 리스크 수준)
- 커버리지 메트릭(예: 요구사항 커버리지, 코드 커버리지)
- 비용 메트릭(예: 테스팅 비용, 조직의 품질 비용)

5.3.2. 테스트 보고서의 목적, 내용, 대상

테스트 보고는 테스팅 도중과 이후에 테스트 정보를 요약해 전달하는 활동이다. 테스트 진행 상황 보고서는 테스팅의 지속적인 관리를 지원하며, 계획에서 벗어나거나 상황이 바뀌어 테스트 일정, 자원, 테스트 계획을 수정해야 할 경우 그것을 할 수 있는 충분한 정보를 제공해야 한다. 테스트 완료 보고서는 테스팅의 특정 단계(예: 테스트 레벨, 테스트 주기, 반복 주기)를 요약하고, 후속 테스팅을 위한 정보를 제공할 수 있다.

테스트 모니터링 및 제어 과정 중 테스트팀은 이해관계자에게 정보를 제공하기 위해 테스트 진행 상황 보고서를 작성한다. 일반적으로 테스트 진행 상황 보고서는 정기적으로(예: 매일, 매주, 또는 기타 주기)

작성하며 다음을 포함한다:

- 테스트 기간
- 테스트 진행 상황(예: 예정보다 빠르다 또는 늦어지고 있다), 주목할 만한 편차 포함
- 테스팅 진행 방해 요소와 대응 방법
- 테스트 메트릭(예시는 5.3.1 참조)
- 테스팅 중 식별한 신규 및 변경된 리스크
- 다음 주기에 예정된 테스팅

테스트 완료 보고서는 프로젝트, 테스트 레벨, 테스트 유형이 끝나고, 이상적으로는 완료 조건도 충족된 상황에서 이루어지는 테스트 완료 활동 중 작성한다. 이 보고서는 테스트 진행 상황 보고서와 기타 데이터를 기반으로 작성한다. 일반적인 테스트 완료 보고서는 다음을 포함한다:

- 테스트 요약
- 원래 테스트 계획(즉, 테스트 목적과 완료 조건)에 기반한 테스팅 및 제품 품질 평가
- 테스트 계획과의 편차(예: 계획한 일정, 기간, 공수와의 차이)
- 테스팅 방해 요소와 대응 방법
- 테스트 진행 상황 보고서를 기반으로 한 테스트 메트릭
- 완화되지 않은 리스크, 수정되지 않은 결함
- 테스팅 관련 교훈

보고의 대상에 따라 보고서에서 얻고자 하는 정보가 다르고, 보고의 형식과 빈도도 달라진다. 같은 팀 인원에게 테스트 진행 상황을 보고하는 것은 빈번하고 비공식적으로 이루어지지만, 완료된 프로젝트에 대한 테스트 보고는 정해진 양식을 가지고 한 번만 이뤄진다.

ISO/IEC/IEEE 29119-3 표준은 테스트 진행 상황 보고서(테스트 상황 보고서라고 함)와 테스트 완료 보고서의 양식과 예제를 포함하고 있다.

5.3.3. 테스팅 상황 전달

테스트 상황을 전달하는 가장 좋은 의사소통 방법은 테스트 관리자의 관심, 조직의 테스트 전략, 규제 표준, 또 자체 조직 팀(self-organizing team)일 경우 팀 자체에 따라서도 달라진다(1.5.2 참조). 여기에는 다음과 같은 방법이 있다:

- 팀원 및 기타 이해관계자와의 대화
- 대시보드(예: '지속적 통합'/'지속적 전달' 대시보드, 태스크 보드, 번다운 차트)
- 전자 통신 채널(예: 이메일, 채팅)
- 온라인 문서
- 공식 테스트 보고서(5.3.2 참조)

이런 선택지 중 하나 이상을 사용할 수 있다. 물리적 거리나 시차로 인해 직접적인 대화가 어려운 분산된 팀은 더욱 공식적인 의사소통 방법을 사용하는 것이 적합할 수 있다. 보통 이해관계자마다 관심을 가지는 정보가 다르므로, 그에 따라 의사소통 방식을 조정할 필요가 있다.

5.4. 형상 관리

테스팅에서 형상 관리(CM, Configuration Management)는 테스트 계획서, 테스트 전략서, 테스트 컨디션, 테스트 케이스, 테스트 스크립트, 테스트 결과, 테스트 로그, 테스트 보고서와 같은 작업 산출물을 형상 항목으로 식별, 제어, 추적하는 지침을 제공한다.

복잡한 형상 항목(예: 테스트 환경)의 경우 형상 관리는 구성 항목, 항목 간 관계, 버전 등을 기록한다. 형상 항목이 테스팅용으로 승인되면 베이스라인이 되며, 공식적인 변경 제어 프로세스를 통해서만 수정할 수 있다.

형상 관리는 새 베이스라인을 만들 때 변경된 형상 항목에 대한 기록을 유지한다. 이전 베이스라인으로 되돌리면 이전 테스트 결과를 재현할 수 있게 된다.

테스팅을 적절히 지원하기 위해 형상 관리는 다음을 보장한다:

- 테스트 항목(테스트 대상의 개별 부분)을 포함한 모든 형상 항목에는 고유한 식별자가 부여되고, 버전이 관리되며, 변경사항이 있는지 추적되고, 다른 형상 항목과 가지는 연관성이 식별돼 테스트 프로세스 전체에서 추적성이 유지된다.
- 식별된 모든 문서와 소프트웨어 항목은 테스트 문서에서 명확히 참조된다.

지속적 통합, 지속적 전달, 지속적 배포, 그리고 관련된 테스팅은 일반적으로 자동화된 데브옵스 파이프라인으로 구현하며(2.1.4 참조), 여기에 보통 자동화된 형상 관리가 포함돼 있다.

5.5. 결함 관리

테스트의 주요 목적 중 하나가 결함 식별이므로 잘 확립된 결함 관리 프로세스가 필요하다. 여기서 '결함'이라고 부르고 있지만, 보고된 이상 현상은 실제로 결함일 수도, 아닐 수도 있다(예: 긍정 오류, 변경 요청). 이 문제는 결함 보고서를 처리하는 과정에서 해결된다. 이상 현상은 소프트웨어 개발수명주기(SDLC) 모든 단계에서 보고될 수 있으며, 보고 양식은 소프트웨어 개발수명주기(SDLC)에 따라 다르다. 모든 관련 이해관계자는 이 프로세스를 따라야 한다. 결함 관리 프로세스에는 기본적으로 개별 이상 현상을 발견부터 종료까지 처리하는 작업 흐름(workflow)과 분류규칙이 포함되어야 한다. 이 작업 흐름은 보통, 보고된 이상 현상을 기록하고, 분석 및 분류하고, 수정하거나 유지하기로 하는 등의 적절한 대응책을 결정하고, 끝으로 결함 보고를 종료하는 활동으로 구성된다. 이 프로세스는 관련된 모든 이해관계자가 따라야 한

다. 정적 테스팅(특히, 정적분석)에서 식별한 결함도 비슷한 방식으로 처리하는 것이 좋다.

일반적인 결함 보고서는 다음과 같은 목적을 가진다:

- 결함을 처리 및 해결하는 책임을 진 사람에게 문제 해결을 위한 충분한 정보 제공
- 작업 결과물의 품질을 추적할 수 있는 수단 제공
- 개발 및 테스트 프로세스 개선을 위한 아이디어 제공

일반적으로 동적 테스팅 중에 작성하는 결함 보고서는 다음을 포함한다:

- 고유 식별자
- 보고하는 이상 현상을 요약하는 제목
- 이상 현상이 관찰된 날짜, 보고 주체 조직, 작성자(역할 포함)
- 테스트 대상 및 테스트 환경 식별 정보
- 결함의 정황(예: 실행 중인 테스트 케이스, 수행 중인 테스트 활동, 소프트웨어 개발수명주기(SDLC) 단계, 또 사용 중인 테스트 기법/체크리스트/테스트 데이터와 같은 기타 관련 정보)
- 이상 현상을 발견한 절차, 관련 테스트 로그, 데이터베이스 덤프, 스크린샷, 녹음 파일 등 장애의 재현 및 해결에 필요한 정보
- 기대 결과와 실제 결과
- 결함이 이해관계자의 이익 또는 요구사항에 끼치는 영향의 심각도
- 수정 우선순위
- 결함 상태(예: 신규, 연기, 중복, 수정 대기, 확인 테스트 대기, 재-오픈, 완료, 거부)
- 참조 사항(예: 관련 테스트 케이스)

이런 데이터 중 일부는 결함 관리 도구를 사용하면 자동으로 포함된다(예: 식별자, 날짜, 작성자, 초기 상태 등). 결함 보고서의 양식과 예시는 ISO/IEC/IEEE 29113-3 표준에서 찾을 수 있다. 해당 표준은 결함 보고서를 인시던트 보고서로 지칭하고 있다.

제 6 장 테스트 도구 | 20분

용어

테스트 자동화

6장 학습 목표

6.1 테스팅 지원 도구

FL-6.1.1 (K2) 다양한 유형의 테스트 도구가 어떻게 테스팅을 지원하는지 설명할 수 있다

6.2 테스트 자동화의 효과와 리스크

FL-6.2.1 (K1) 테스트 자동화의 효과와 리스크를 상기할 수 있다

6.1. 테스팅 지원 도구

테스트 도구는 다양한 테스트 활동을 지원하고 촉진한다. 다음은 몇 가지 예이다:

- 관리 도구 - 소프트웨어 개발수명주기(SDLC), 요구사항, 테스트, 결함, 형상 관리를 용이하게 해서 테스트 프로세스 효율성을 높인다.
- 정적 테스팅 도구 - 테스터의 리뷰와 정적 분석 수행을 지원한다.
- 테스트 설계 및 구현 도구 - 테스트 케이스, 테스트 데이터, 테스트 절차 생성을 용이하게 한다.
- 테스트 실행 및 커버리지 도구 – 자동 테스트 실행 및 커버리지 측정을 지원한다.
- 비기능 테스팅 도구 - 수동으로 실행하기 어렵거나 불가능한 비기능 테스트를 테스터가 수행할 수 있게 한다.
- 데브옵스 도구 - 데브옵스 배포 파이프라인, 작업 흐름 추적, 자동 빌드 프로세스, 지속적인 통합 및 배포 등을 지원한다.
- 협업 도구 - 원활한 커뮤니케이션을 지원한다.
- 확장성 및 배포 표준화 지원 도구(예: 가상 머신, 컨테이너화 도구)
- 테스팅에 도움이 되는 기타 도구(예: 테스팅에 활용하면 스프레드시트도 테스트 도구가 된다)

6.2. 테스트 자동화의 효과와 리스크

도구를 도입한다고 성공이 보장되는 것은 아니다. 실질적이고 지속적인 효과를 얻기 위해서는 새로운 도구를 도입할 때마다 노력을 들여야 한다(예: 도구의 도입, 유지보수, 교육 등을 위해). 또한, 분석과 완화가 필요한 리스크도 있다.

테스트 자동화가 가져올 수 있는 효과는 다음과 같다:

- 반복적 수작업(예: 리그레션 테스트 실행, 같은 테스트 데이터의 반복 입력, 예상 결과와 실제 결과의 비교, 코딩 표준 확인)을 줄여 시간 절약
- 일관성 및 재현성 향상으로 사람의 단순 실수 방지(예: 요구사항에서 테스트를 일관되게 도출, 테스트 데이터의 체계적 생성, 테스트를 도구가 같은 주기 및 동일한 순서로 실행)
- 더욱 객관적인 평가(예: 커버리지) 및 사람이 도출하기 너무 복잡한 측정치의 제공
- 테스트 관리 및 테스트 보고에 필요한 테스팅 정보의 접근성 향상(예: 테스트 진행 상황, 결함율, 테스트 실행 기간에 대한 통계, 그래프, 집계 데이터)
- 결함 조기 식별, 빠른 피드백, 출시 시간 단축을 가능하게 하는 테스트 실행 시간 단축
- 테스터가 새로운, 그리고 더 심층적이며 효과적인 테스트를 설계할 시간 확보

테스트 자동화 활용 시 잠재적 리스크는 다음과 같다:

- 도구의 효과(기능 및 사용 편의성 포함)에 대한 비현실적인 기대

- 도구 도입, 테스트 스크립트 유지 관리, 기존 수동 테스트 프로세스 변경에 필요한 시간, 비용, 노력에 대한 부정확한 추정
- 수동 테스팅이 더 적합한 곳에 테스트 도구 사용
- 도구에 지나치게 의존(예: 사람의 비판적 사고의 필요성 무시)
- 폐업, 도구 지원 중단, 다른 공급업체로 도구 매각, 열악한 지원(예: 문의, 업그레이드, 결함 수정에 대한 대응관련) 등의 문제가 생길 수 있는 도구 공급업체에 대한 종속성
- 지원이 중단되거나(즉, 더는 업데이트되지 않거나), 추가 개발을 통해 내부 컴포넌트를 빈번하게 업데이트해야 할 필요가 있을 수 있는 오픈소스 소프트웨어의 사용
- 자동화 도구가 개발 플랫폼과 호환되지 않을 수 있음
- 규제 요건이나 안전 표준을 준수하지 않는 부적합한 도구의 선택

제 7 장 References

Standards

ISO/IEC/IEEE 29119-1 (2022) Software and systems engineering – Software testing – Part 1: General Concepts

ISO/IEC/IEEE 29119-2 (2021) Software and systems engineering – Software testing – Part 2: Test processes

ISO/IEC/IEEE 29119-3 (2021) Software and systems engineering – Software testing – Part 3: Test documentation

ISO/IEC/IEEE 29119-4 (2021) Software and systems engineering – Software testing – Part 4: Test techniques

ISO/IEC 25010, (2011) Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models

ISO/IEC 20246 (2017) Software and systems engineering – Work product reviews

ISO/IEC/IEEE 14764:2022 – Software engineering – Software life cycle processes – Maintenance ISO 31000 (2018) Risk management – Principles and guidelines

Books

Adzic, G. (2009) Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing, Neuri Limited

Ammann, P. and Offutt, J. (2016) Introduction to Software Testing (2e), Cambridge University Press

Andrews, M. and Whittaker, J. (2006) How to Break Web Software: Functional and Security Testing of Web Applications and Web Services, Addison-Wesley Professional

Beck, K. (2003) Test Driven Development: By Example, Addison-Wesley

Beizer, B. (1990) Software Testing Techniques (2e), Van Nostrand Reinhold: Boston MA Boehm, B. (1981) Software Engineering Economics, Prentice Hall, Englewood Cliffs, NJ

Buxton, J.N. and Randell B., eds (1970), Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969, p. 16

Chelimsky, D. et al. (2010) The Rspec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends, The Pragmatic Bookshelf: Raleigh, NC

Cohn, M. (2009) Succeeding with Agile: Software Development Using Scrum, Addison-Wesley Copeland, L. (2004) A Practitioner's Guide to Software Test Design, Artech House: Norwood MA Craig, R. and Jaskiel, S. (2002) Systematic Software Testing, Artech House: Norwood MA

Crispin, L. and Gregory, J. (2008) Agile Testing: A Practical Guide for Testers and Agile Teams, Pearson Education: Boston MA

Forgács, I., and Kovács, A. (2019) Practical Test Design: Selection of traditional and automated test design techniques, BCS, The Chartered Institute for IT

Gawande A. (2009) The Checklist Manifesto: How to Get Things Right, New York, NY: Metropolitan Books

Gärtner, M. (2011), ATDD by Example: A Practical Guide to Acceptance Test-Driven Development, Pearson Education: Boston MA

Gilb, T., Graham, D. (1993) Software Inspection, Addison Wesley

Hendrickson, E. (2013) Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing, The Pragmatic Programmers

Hetzel, B. (1988) The Complete Guide to Software Testing, 2nd ed., John Wiley and Sons

Jeffries, R., Anderson, A., Hendrickson, C. (2000) Extreme Programming Installed, Addison-Wesley Professional

Jorgensen, P. (2014) Software Testing, A Craftsman's Approach (4e), CRC Press: Boca Raton FL Kan, S. (2003) Metrics and Models in Software Quality Engineering, 2nd ed., Addison-Wesley Kaner, C., Falk, J., and Nguyen, H.Q. (1999) Testing Computer Software, 2nd ed., Wiley

Kaner, C., Bach, J., and Pettichord, B. (2011) Lessons Learned in Software Testing: A Context-Driven Approach, 1st ed., Wiley

Kim, G., Humble, J., Debois, P. and Willis, J. (2016) The DevOps Handbook, Portland, OR

Koomen, T., van der Aalst, L., Broekman, B. and Vroon, M. (2006) TMap Next for result-driven testing, UTN Publishers, The Netherlands

Myers, G. (2011) The Art of Software Testing, (3e), John Wiley & Sons: New York NY O'Regan, G. (2019) Concise Guide to Software Testing, Springer Nature Switzerland Pressman, R.S. (2019) Software Engineering. A Practitioner's Approach, 9th ed., McGraw Hill

Roman, A. (2018) Thinking-Driven Testing. The Most Reasonable Approach to Quality Control, Springer Nature Switzerland

Van Veenendaal, E (ed.) (2012) Practical Risk-Based Testing, The PRISMA Approach, UTN Publishers: The Netherlands

Watson, A.H., Wallace, D.R. and McCabe, T.J. (1996) Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, U.S. Dept. of Commerce, Technology Administration, NIST

Westfall, L. (2009) The Certified Software Quality Engineer Handbook, ASQ Quality Press Whittaker, J. (2002) How to Break Software: A Practical Guide to Testing, Pearson

Whittaker, J. (2009) Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design, Addison Wesley

Whittaker, J. and Thompson, H. (2003) How to Break Software Security, Addison Wesley Wiegers, K. (2001) Peer Reviews in Software: A Practical Guide, Addison-Wesley Professional

Articles and Web Pages

Brykczynski, B. (1999) “A survey of software inspection checklists,” ACM SIGSOFT Software Engineering Notes, 24(1), pp. 82-89

Enders, A. (1975) “An Analysis of Errors and Their Causes in System Programs,” IEEE Transactions on Software Engineering 1(2), pp. 140-149

Manna, Z., Waldinger, R. (1978) “The logic of computer programming,” IEEE Transactions on Software Engineering 4(3), pp. 199-229

Marick, B. (2003) Exploration through Example, http://www.exampler.com/old-blog/2003/08/21.1.html#agile-testing-project-1

Nielsen, J. (1994) “Enhancing the explanatory power of usability heuristics,” Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Celebrating Interdependence, ACM Press, pp. 152–158

Salman. I. (2016) “Cognitive biases in software quality and testing,” Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16), ACM, pp. 823-826.

Wake, B. (2003) “INVEST in Good Stories, and SMART Tasks,” https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

Appendix A – Learning Objectives/Cognitive Level of Knowledge

The following learning objectives are defined as applying to this syllabus. Each topic in the syllabus will be examined according to the learning objective for it. The learning objectives begin with an action verb corresponding to its cognitive level of knowledge as listed below.

Level 1: Remember (K1) – the candidate will remember, recognize and recall a term or concept.

Action verbs: identify, recall, remember, recognize.

Examples:

- "Identify typical test objectives ."
- "Recall the concepts of the test pyramid."
- "Recognize how a tester adds value to iteration and release planning"

Level 2: Understand (K2) – the candidate can select the reasons or explanations for statements related to the topic, and can summarize, compare, classify and give examples for the testing concept.

Action verbs: classify, compare, contrast, differentiate, distinguish, exemplify, explain, give examples, interpret, summarize.

Examples:

- "Classify the different options for writing acceptance criteria."
- "Compare the different roles in testing" (look for similarities, differences or both).
- "Distinguish between project risks and product risks" (allows concepts to be differentiated).
- "Exemplify the purpose and content of a test plan."
- "Explain the impact of context on the test process."
- "Summarize the activities of the review process."

Level 3: Apply (K3) – the candidate can carry out a procedure when confronted with a familiar task, or select the correct procedure and apply it to a given context.

Action verbs: apply, implement, prepare, use.

Examples:

- "Apply test case prioritization" (should refer to a procedure, technique, process, algorithm etc.).
- "Prepare a defect report."

- "Use boundary value analysis to derive test cases."

References for the cognitive levels of learning objectives:

Anderson, L. W. and Krathwohl, D. R. (eds) (2001) A Taxonomy for Learning, Teaching

Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, Allyn & Bacon