



소프트웨어 테스팅 Day 2: 명세 기반 테스트와 자동화 기초

2025년 5월 29일

지은경 (KAIST 전산학부)



명세(Specification) 기반 테스트의 이해

정의 및 목적

- 명세 기반 테스트는 소프트웨어의 요구 사항 명세서, 디자인 문서, 사용자 스토리 등을 기반으로 테스트 케이스를 도출
- 소프트웨어가 무엇을 해야 하는지에 집중하며, 내부 구조보다는 기능적 동작에 초점을 맞춤

핵심 특징

- 코드 내부 구조에 대한 지식 없이도 테스트 케이스를 설계할 수 있어, 개발 초기 단계부터 테스트 계획 수립 가능
- 요구사항과 테스트 케이스 간의 추적성을 확보하여 테스트 커버리지를 관리하는데 효과적

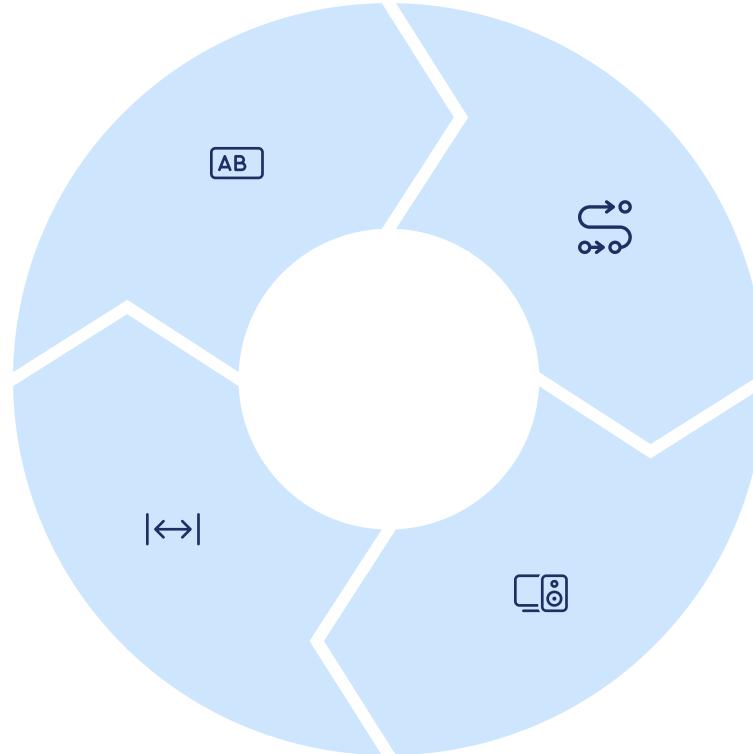
적용 영역

- 기능 테스트, 인수 테스트, 시스템 테스트 단계에서 주로 활용
- 사용자 관점에서 소프트웨어의 완전성을 검증하는 데 유용
- 다양한 입력 조건과 예외 상황을 체계적으로 테스트할 수 있는 기반을 제공

블랙박스 테스트의 핵심 원리

입력 식별
시스템에 입력될 수 있는 모든 가능한 데이터
유형과 범위를 파악

결과 분석
실제 출력과 예상 출력의 차이를 분석하여 결
함을 식별



프로세스 블랙박스화

내부 로직이나 코드 구현에 관계없이 기능 명
세에만 집중

출력 검증

주어진 입력에 대해 예상되는 출력 결과를 명
확히 정의하고 확인

- 블랙박스 테스트는 소프트웨어의 내부 구조나 작동 방식을 모르는 상태에서 수행하는 테스트 방법론
- 사용자 관점에서 소프트웨어를 평가하기 때문에 실제 사용 환경에서 발생할 수 있는 문제를 효과적으로 발견 가능
- 명세서와 실제 구현 사이의 불일치를 찾아내는데 효과적

동등 분할(Equivalence Partitioning) 기법의 핵심



테스트 효율성 극대화

최소한의 테스트로 최대 커버리지 달성



데이터 클래스 구분

유효/무효 입력 영역 식별



대표값 선정

각 분할에서 대표 값으로 테스트

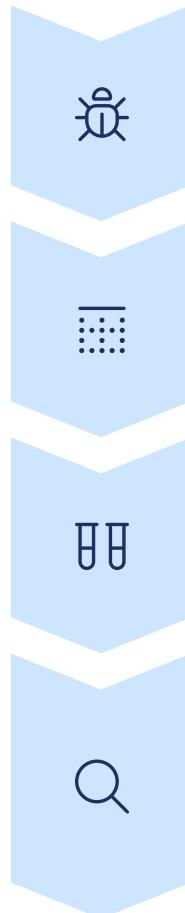
- 동등 분할 기법: 입력 도메인을 동일한 특성을 가진 여러 그룹(동등 클래스)으로 나누어, 각 그룹에서 대표값만 테스트하는 방법
- 기본 원리: 동일한 등가 클래스 내의 모든 값이 프로그램에 의해 동일하게 처리된다는 가정에 기초
- 적용 예시: 1-100 사이의 숫자를 입력받는 프로그램
 - 유효 동등 클래스: 1-100 → 대표값 50으로 테스트
 - 무효 동등 클래스: 0 이하, 101 이상 → 대표값 0, 150으로 테스트
 - 효과: 모든 가능한 입력을 테스트하는 것과 유사한 효과 달성

동등 분할 기법 실전 적용 예시

기능	분할 기준	등가 클래스	테스트 데이터
나이 확인	성인 연령 기준 (19세)	유효 1: 성인 (19세 이상)	25 / 42 / 65
		유효 2: 미성년 (0-18세)	5 / 15 / 18
결제 금액	최소 1,000원, 최대 100만원	무효: 음수 나이	-1 / -10
		유효: 1,000-1,000,000원	1,000 / 50,000 / 1,000,000
		무효 1: 1,000원 미만	0 / 500
		무효 2: 100만원 초과	1,000,001 / 2,000,000

- 테스트 대상 기능의 입력 도메인을 명확히 이해 / 각 기능에 대해 유효한 입력과 무효한 입력을 구분 / 각 등가 클래스에서 대표 값을 선정
- 테스트 케이스의 수를 효과적으로 줄이면서도 주요 시나리오를 모두 커버할 수 있게 해줌
- 특히 입력 범위가 넓은 경우 모든 값을 테스트하는 것은 비효율적이므로, 동등 분할 기법이 유용

경계값 분석의 중요성



결함 발생 빈도

경계값 주변에서 결함이 자주 발생

경계 식별

유효/무효 영역의 경계점 파악

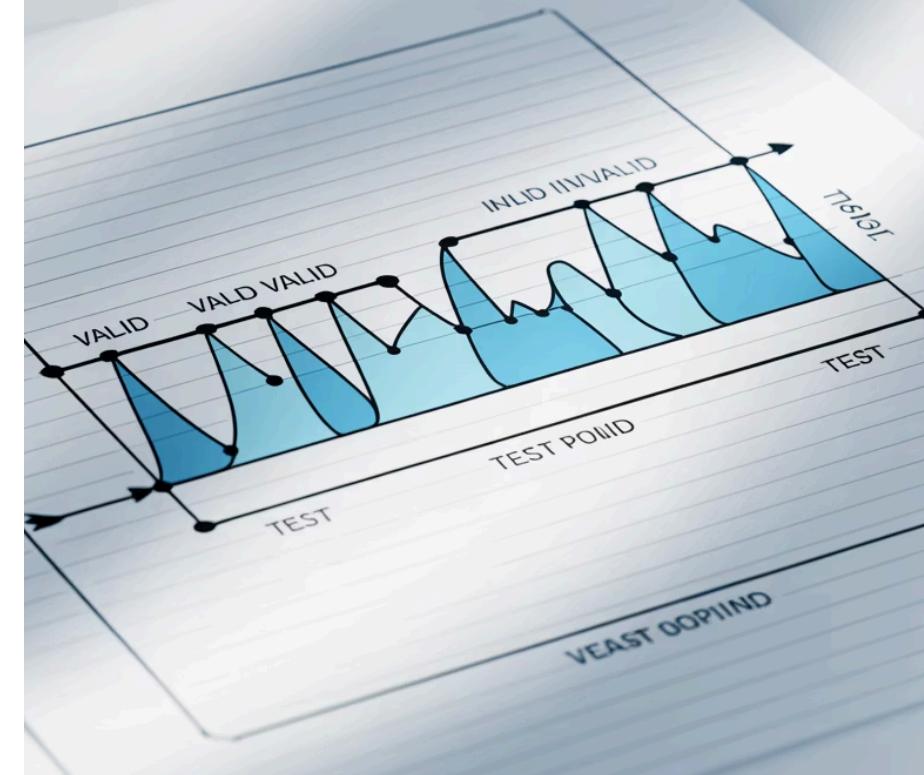
테스트 설계

경계값 및 경계값 근처(바로 안쪽, 바로 바깥쪽) 값으로 테스트

결함 검출

'<', '<=', '>', '>=' 등의 비교 연산자 사용 오류, off-by-one 오류 등 효과적 검출

Boundary Value Analysis



경계값 분석 실전 예제



날짜 입력 필드 (1-31일)

경계값: 0, 1, 2, 30, 31, 32

- 최소 유효값(1)과 최소 유효값 바로 아래(0)
- 최대 유효값(31)과 최대 유효값 바로 위(32)
- 경계 근처 값(2, 30)



비밀번호 길이 제한 (8-16자)

경계값: 7, 8, 9, 15, 16, 17

- 7자 비밀번호 (무효)
- 8자 비밀번호 (유효)
- 16자 비밀번호 (유효)
- 17자 비밀번호 (무효)



송금 한도 (1-100만원)

경계값: 0, 1, 2, 999,999, 1,000,000, 1,000,001

- 0원 송금 시도 (무효)
- 1원 송금 (유효)
- 100만원 송금 (유효)
- 100만 1원 송금 시도 (무효)

- 경계값 분석 적용 시 입력 필드의 제한 조건을 명확히 파악하는 것이 중요
- 웹/모바일 애플리케이션에서 자주 볼 수 있는 입력 필드에 효과적으로 적용 가능
- 데이터 유효성 검사 로직이 복잡한 시스템에서 특히 효과적
- 사용자 입력 처리 과정에서 발생할 수 있는 다양한 예외 상황을 미리 검증

의사결정 테이블 기법 소개

의사결정 테이블이란?

- 다양한 입력 조건의 조합과 그에 따른 예상 결과를 표 형태로 정리
- 복잡한 비즈니스 로직이나 여러 조건이 결합된 상황에서 체계적 테스트에 효과적
- "만약 A이고 B이면서 C가 아니라면 D를 수행한다"와 같은 복잡한 조건부 로직 테스트에 유용
- 모든 조건 조합을 명확히 시각화하여 누락된 테스트 케이스 방지

의사결정 테이블의 구성요소

- 조건 스텝(Condition Stub): 테스트할 조건들의 목록
- 액션 스텝(Action Stub): 조건 조합에 따른 결과 액션
- 조건 항목(Condition Entry): 각 조건의 상태(참/거짓)
- 액션 항목(Action Entry): 액션의 실행 여부(O/X)

의사결정 테이블 작성 절차:

- 모든 가능한 조건 조합 나열
- 각 조합에 대한 예상 결과 정의
- 시스템의 동작을 포괄적으로 검증

의사결정 테이블 구성 실습

온라인 쇼핑몰의 할인 정책을 테스트하기 위한 예시

룰 ID	R1	R2	R3	R4
조건				
회원 여부	Y	Y	N	N
구매액 1만원 이상	Y	N	Y	N
액션				
5% 할인 적용	O	X	X	X
2% 할인 적용	X	O	X	X
회원 가입 안내	X	X	O	O

- 테이블을 통해 모든 가능한 시나리오에 대한 테스트 케이스 도출
- 초기에 모든 조합을 포함한 완전한 테이블을 만든 후, 불필요한 조합을 제거하거나 유사한 조합을 병합하는 과정을 통해 테이블을 최적화 → 테스트 효율성 높임



복잡한 비즈니스 로직에 대한 의사결정 테이블

대출 승인 시스템 예시

- 조건: 신용점수(상/중/하), 연소득(5천만원 이상/미만), 부채비율(30% 이상/미만)
- 액션: 대출 승인, 조건부 승인, 거절
- 총 12개 조합($3 \times 2 \times 2$) 가능, 각 조합별 결과 명확히 정의 필요

의사결정 테이블 최적화

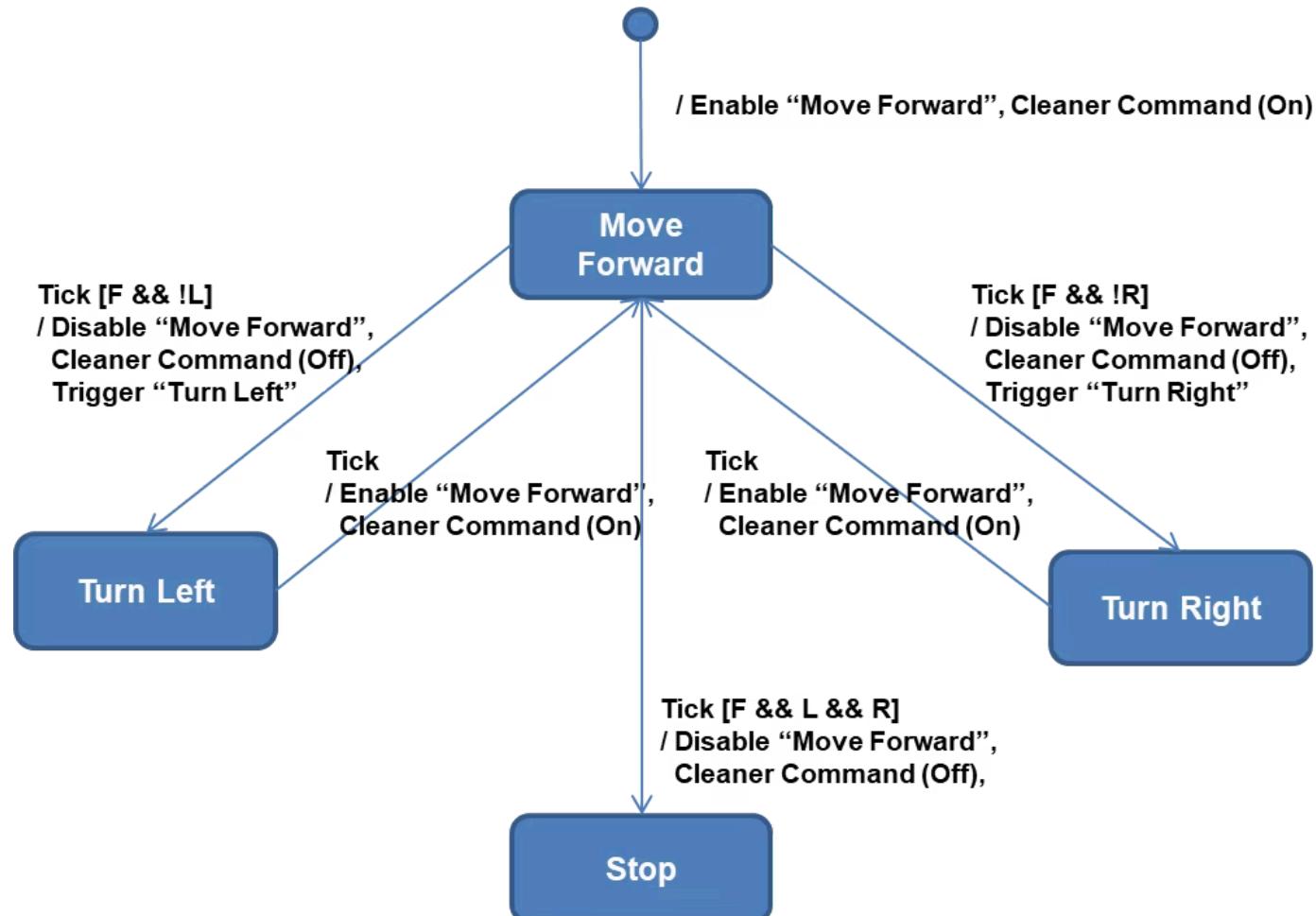
- 조건 증가 시 조합 수 기하급수적 증가
→ 최적화 필요
- 조건 병합: 유사한 결과를 가진 조합 통합
- 불가능한 조합 제거: 현실에서 발생할 수 없는 조합 제외
- 우선순위 기반 선택: 중요도가 높은 조합 우선 테스트

테스트 케이스 도출

- 의사결정 테이블의 각 열(룰) = 하나의 테스트 케이스
- 테스트 케이스 구성: 입력 조건 조합 + 예상 결과
- 예시: "신용점수 상, 연소득 5천만원 이상, 부채비율 30% 미만인 고객 → 대출 승인"

상태 전이 다이어그램

로봇 진공 청소기 제어기 상태 전이 다이어그램 (예)



- 시스템의 다양한 상태 간 전환을 그래픽으로 표현한 모델
- 상태에 따라 동작이 달라지는 시스템 테스트에 효과적
- 로그인 프로세스, 주문 관리, 워크플로우 등에 적합

상태 전이 다이어그램 기반 테스트 생성

상태 전이 다이어그램 기반 테스트 도출 프로세스

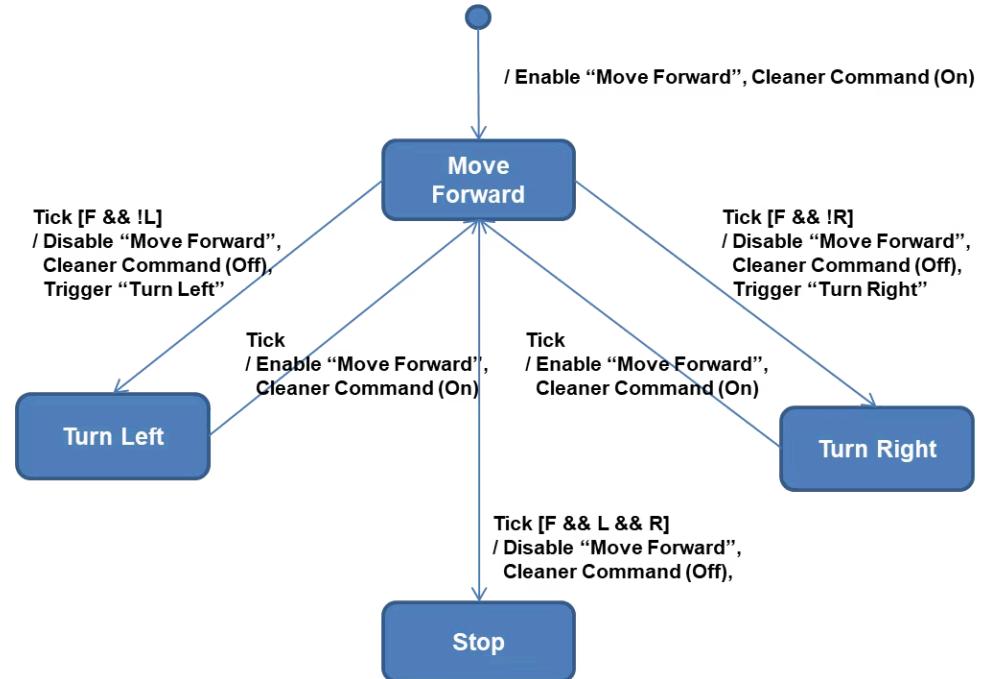


상태 전이 테스트의 목적

- 시스템의 상태 변화가 올바르게 이루어지는지 검증
- 잘못된 상태 전이가 적절히 처리되는지 확인
- 모든 가능한 상태 전이 경로의 정상 작동 여부 테스트
- 상태 기반 시스템의 복잡한 동작을 체계적으로 검증

상태 전이 다이어그램 테스트 커버리지 수준

- 0-스위치 커버리지
 - 모든 상태를 최소 한 번 방문
- 1-스위치 커버리지
 - 모든 전이를 최소 한 번 테스트
- n-스위치 커버리지
 - 특정 전이 시퀀스를 테스트
- 테스트 리소스와 시스템 중요도에 따라 적절한 커버리지 수준 선택



유스케이스 기반 테스트



사용자 중심 접근

- 사용자(액터)의 관점에서 시스템을 테스트
- 사용자 시나리오에 초점 맞춤
- 실사용 환경에서 발생할 수 있는 문제 조기 발견



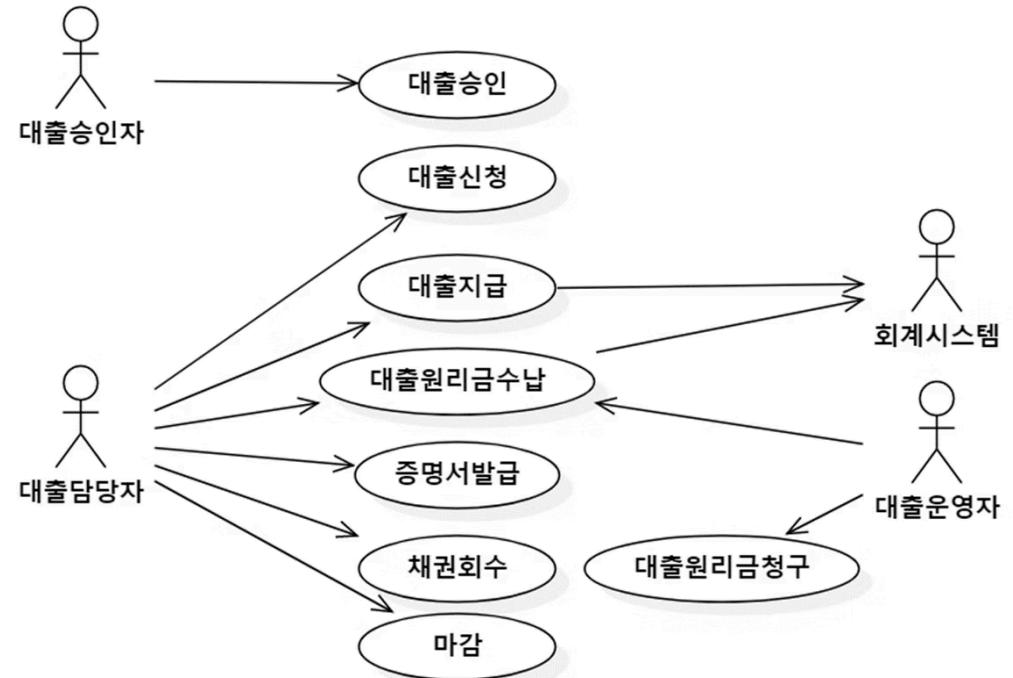
시나리오 기반 검증

- 기본 흐름, 대체 흐름, 예외 흐름을 포함한 다양한 시나리오 테스트
- 시스템의 견고성 확인
- 단순 기능 테스트를 넘어 프로세스 전체 검증



요구사항 추적성

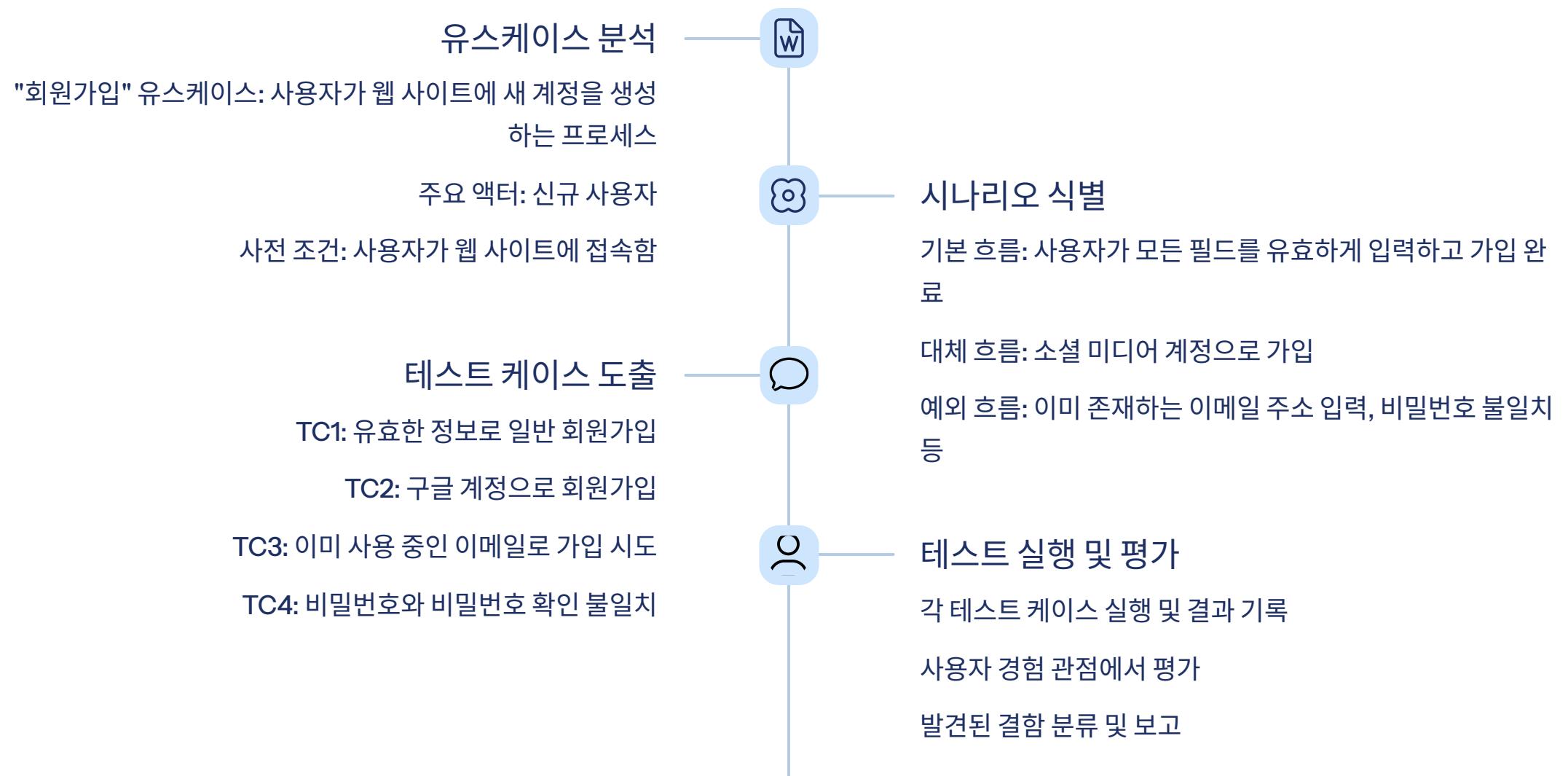
- 유스케이스와 테스트 케이스 간의 명확한 맵핑 제공
- 요구사항 추적성 확보
- 모든 요구사항이 적절히 테스트되었는지 확인 가능



- 유스케이스 명세 요소:
 - 사전 조건
 - 주요 성공 시나리오
 - 대체 경로
 - 예외 상황
- 다양한 사용자 시나리오 테스트에 활용
- 사용자 인터페이스 및 시스템 통합 테스트에 특히 효과적



유스케이스 기반 테스트 사례 연구



테스트 케이스 설계의 기본 원칙

- 효과적인 테스트 케이스 설계는 소프트웨어 품질 보증의 핵심

효과적인 테스트 케이스

결함 발견 가능성 극대화

균형 잡힌 접근

긍정/부정 테스트 균형

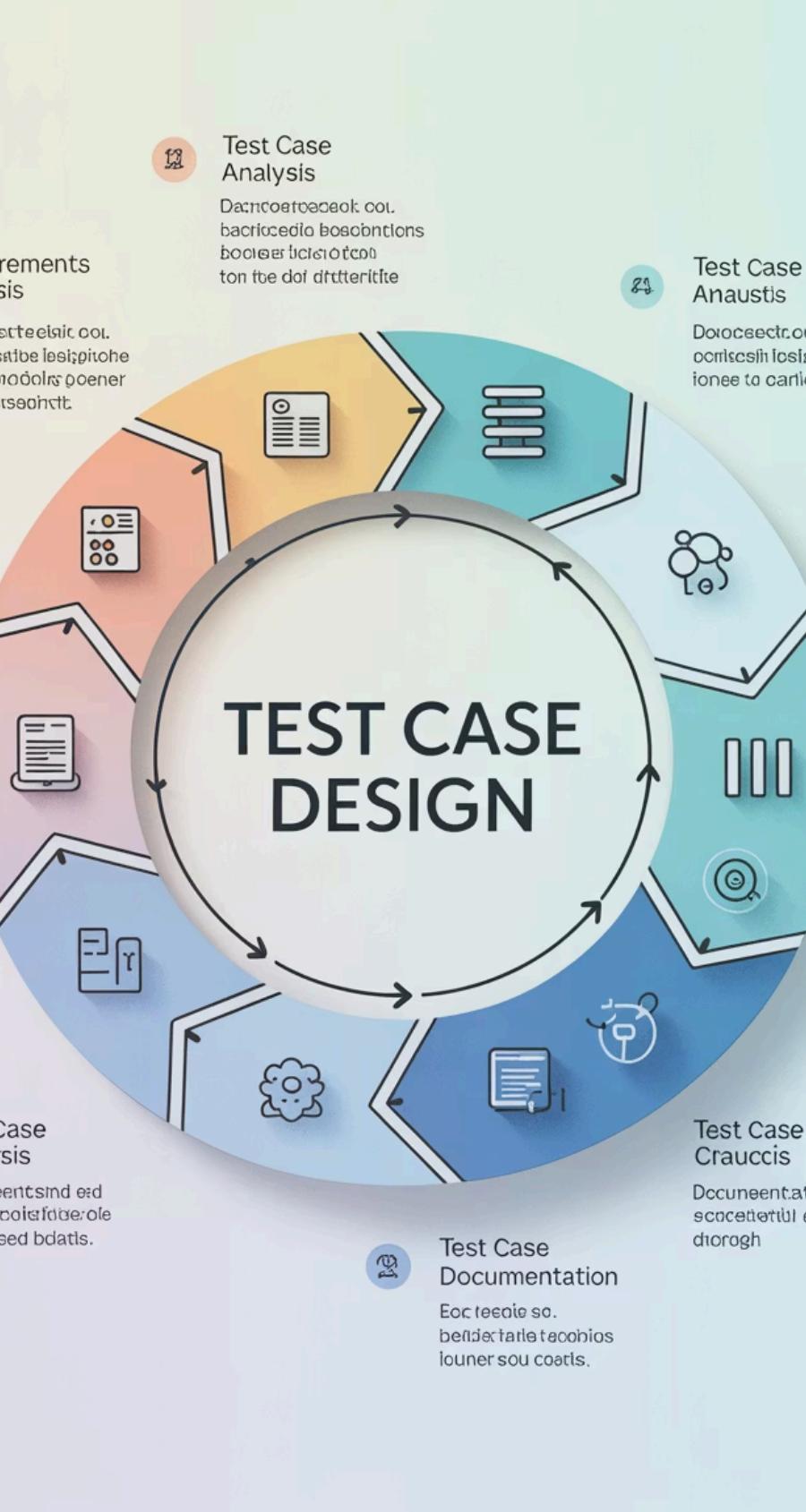
명확한 문서화

구체적이고 재현 가능한 단계

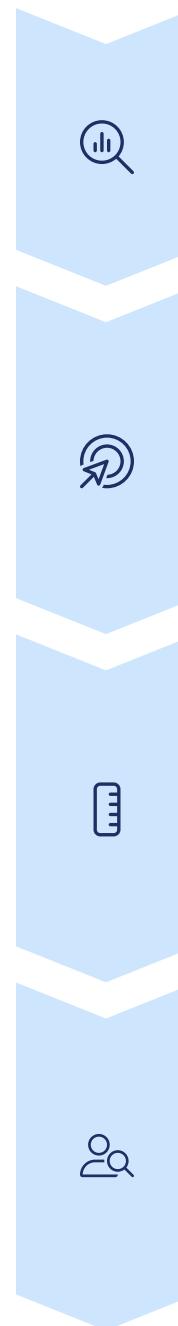
테스트 목표 중심

특정 테스트 목표에 부합





테스트 케이스 설계 절차



테스트 기반 분석

- 요구사항 명세서, 사용자 스토리 등 테스트 기반 문서 분석
- 테스트 대상 식별 및 이해, 테스트 범위 정의

테스트 전략 수립

- 적절한 테스트 기법 선택 (동등 분할, 경계값 분석 등)
- 테스트 우선순위 및 깊이 결정
- 시스템 중요도와 리스크에 따른 접근 방식 조정

테스트 케이스 작성

- 테스트 ID, 설명, 전제조건 정의
- 테스트 단계, 예상 결과, 실제 결과 작성
- 구체적이고 명확한 구조화 진행

검토 및 최적화

- 테스트 케이스 중복 제거 및 누락 부분 보완
- 테스트 케이스 간 의존성 파악
- 실행 순서 최적화

효과적인 테스트 케이스 예시

로그인 기능을 검증하기 위한 테스트 케이스

테스트 ID	TC_LOGIN_001
테스트 제목	유효한 자격 증명으로 로그인
테스트 목적	유효한 사용자 이름과 비밀번호로 로그인이 성공하는지 확인
전제 조건	<ol style="list-style-type: none">애플리케이션이 실행 중임테스트 계정이 시스템에 존재함로그인 페이지가 표시됨
테스트 데이터	사용자 이름: testuser@example.com 비밀번호: Test@123
테스트 단계	<ol style="list-style-type: none">사용자 이름 필드에 'testuser@example.com' 입력비밀번호 필드에 'Test@123' 입력'로그인' 버튼 클릭
예상 결과	<ol style="list-style-type: none">로그인이 성공적으로 처리됨사용자가 대시보드 페이지로 리디렉션됨대시보드에 사용자 이름이 표시됨
실제 결과	(테스트 실행 후 기록)
상태	미실행

실제 프로젝트에서는 테스트 집합을 테스트 관리 도구(예: TestRail, JIRA)에 등록 관리, 테스트 실행 결과와 결함 정보를 연계 추적

테스트 입력/출력 정의하기

테스트 입력 정의

테스트 입력은 테스트 대상 시스템에 제공되는 모든 데이터와 조건을 의미

효과적인 테스트 입력 정의 고려사항:

- 다양성: 유효한 입력과 무효한 입력을 모두 포함
- 경계 조건: 허용 범위의 경계값 포함
- 특수 케이스: null 값, 빈 문자열, 특수 문자 등 고려
- 데이터 의존성: 다른 데이터와의 관계 고려

테스트 데이터 특성:

- 실제 환경을 모방하되, 테스트 목적에 맞게 제어 가능해야 함
- 필요에 따라 테스트 데이터 생성 도구 활용 가능

테스트 출력 정의

테스트 출력은 특정 입력에 대한 시스템의 예상 반응으로, 테스트 결과를 평가하는 기준

명확한 테스트 출력 정의 접근 방법:

- 구체성: 모호하지 않고 측정 가능한 결과 정의
- 완전성: 화면 출력, 데이터 변경, 로그 등 모든 관련 출력 고려
- 검증 방법: 출력을 검증할 수 있는 방법 명시
- 오류 처리: 예외 상황에서의 예상 동작 정의

테스트 출력 정의 기반:

- 요구사항 명세서, 디자인 문서, 사용자 스토리 등을 기반으로 정의
- 필요시 개발자나 비즈니스 분석가와 협의를 통해 명확화

테스트 입력 생성 전략



랜덤 데이터 생성

- 무작위 데이터 생성으로 예상치 못한 결함 발견
- 복잡한 시스템, 경계 조건, 특수 사례에 효과적
- 장점: 다양한 테스트 시나리오 포함 가능
- 단점: 테스트 재현성 낮음



프로덕션 데이터 복제

- 실제 운영 환경 데이터 복사하여 현실적인 테스트 환경 제공
- 데이터 마스킹/익명화 필요 (개인정보 보호)
- 장점: 실제 사용 패턴 반영
- 단점: 특정 테스트 조건 생성 어려움



테스트 데이터 생성 도구

- Faker, DataFactory 등 활용
- 대량의 의미 있는 테스트 데이터 생성 용이
- 테스트 자동화와 결합하여 효율성 향상



테스트 데이터 관리

- 중앙 저장소 활용한 체계적 관리
- 테스트 데이터 버전 관리 구현
- 데이터 세트 분류 및 테스트 케이스 연결
- 재사용성 향상으로 효율적인 테스트 수행

테스트 데이터 생성 도구 예

- 소프트웨어 테스트 데이터 생성을 위한 대표적인 도구들은 목적(예: 구조화된 데이터, 무작위 데이터, 개인정보 마스킹 등)에 따라 다양함
- 대표적 테스트 데이터 생성 도구
 - 간단한 더미 데이터: Mockaroo, Faker
 - DB 테이블용 데이터: Redgate, dbForge
 - 민감정보 기반 생성/마스킹: Tonic.ai, Synthesized
 - 프로그래밍 언어에 따라: Python → Faker, Java → DataFactory, C# → Bogus

테스트 출력 검증 방법

직접 비교 검증

- 예상 결과와 실제 결과를 직접 비교하는 기본적인 검증 방법
- 화면에 표시되는 텍스트나 값의 일치 여부 확인
- 간단한 검증에 적합하나 대량의 데이터나 복잡한 출력 검증에는 비효율적

데이터베이스 검증

- 시스템 동작 후 데이터베이스의 상태 변경 확인
- SQL 쿼리를 사용한 데이터 삽입, 수정, 삭제 검증
- 백엔드 기능이나 데이터 중심 애플리케이션 테스트에 중요

로그 기반 검증

- 시스템 생성 로그 파일 분석을 통한 동작 확인
- UI 출력이 없는 백그라운드 프로세스나 배치 작업 검증에 유용
- 로그 패턴 매칭이나 특정 이벤트 발생 여부 확인 방식으로 진행

자동화된 assertion

- 자동화 테스트 프레임워크의 어설션(assertion) 기능을 활용한 프로그래밍 방식 검증
- `assertEquals()`, `assertTrue()` 등의 메서드로 결과 일치 여부 자동 확인
- 대규모 테스트나 회귀 테스트에 특히 효과적

자동화 테스트의 기본 개념



자동화 테스트란?

- 테스트 스크립트와 도구를 사용하여 사람의 개입 없이 소프트웨어를 테스트하는 방법
- 테스트 실행, 결과 검증, 보고서 생성 등의 과정을 자동화
- 효율성과 일관성 향상에 기여



테스트 자동화 피라미드

- 단위 테스트(하단): 빠르고 안정적, 다수 구현 필요
- 통합 테스트(중간): 컴포넌트 간 상호작용 검증
- UI 테스트(상단): 느리지만 사용자 관점의 종합적 검증 가능



자동화 테스트 수명 주기

- 계획 → 설계 → 개발 → 실행 → 유지보수
- 유지보수는 지속적인 과정으로 진행
- 애플리케이션 변경에 따라 테스트 스크립트도 지속적 업데이트 필요

자동화 테스트의 장점과 단점

장점



효율성 향상

반복적인 테스트를 빠르게 수행하여 테스트 주기를 단축

특히 회귀 테스트에서 큰 시간 절약이 가능



일관성과 정확성

사람의 오류 가능성을 줄이고, 매번 동일한 방식으로 테스트를 실행



테스트 커버리지 확대

더 많은 테스트 케이스를 실행할 수 있어, 소프트웨어의 다양한 측면을 검증 가능



병렬 실행

여러 환경에서 동시에 테스트를 실행하여 시간 절약 가능



CI/CD 통합

지속적 통합 및 배포 파이프라인에 쉽게 통합되어 빠른 피드백을 제공



비용 절감

장기적으로 볼 때 수동 테스트보다 비용 효율적

단점



초기 투자 비용

자동화 테스트 구축에는 상당한 시간과 리소스 필요



유지보수 부담

애플리케이션이 변경될 때마다 테스트 스크립트도 업데이트해야 함



모든 테스트에 적합하지 않음

탐색적 테스트나 사용성 테스트와 같은 일부 테스트는 자동화하기 어려움



잘못된 확신

자동화 테스트가 통과했다고 해서 소프트웨어에 문제가 없다는 것을 보장하지는 않음



도구 의존성

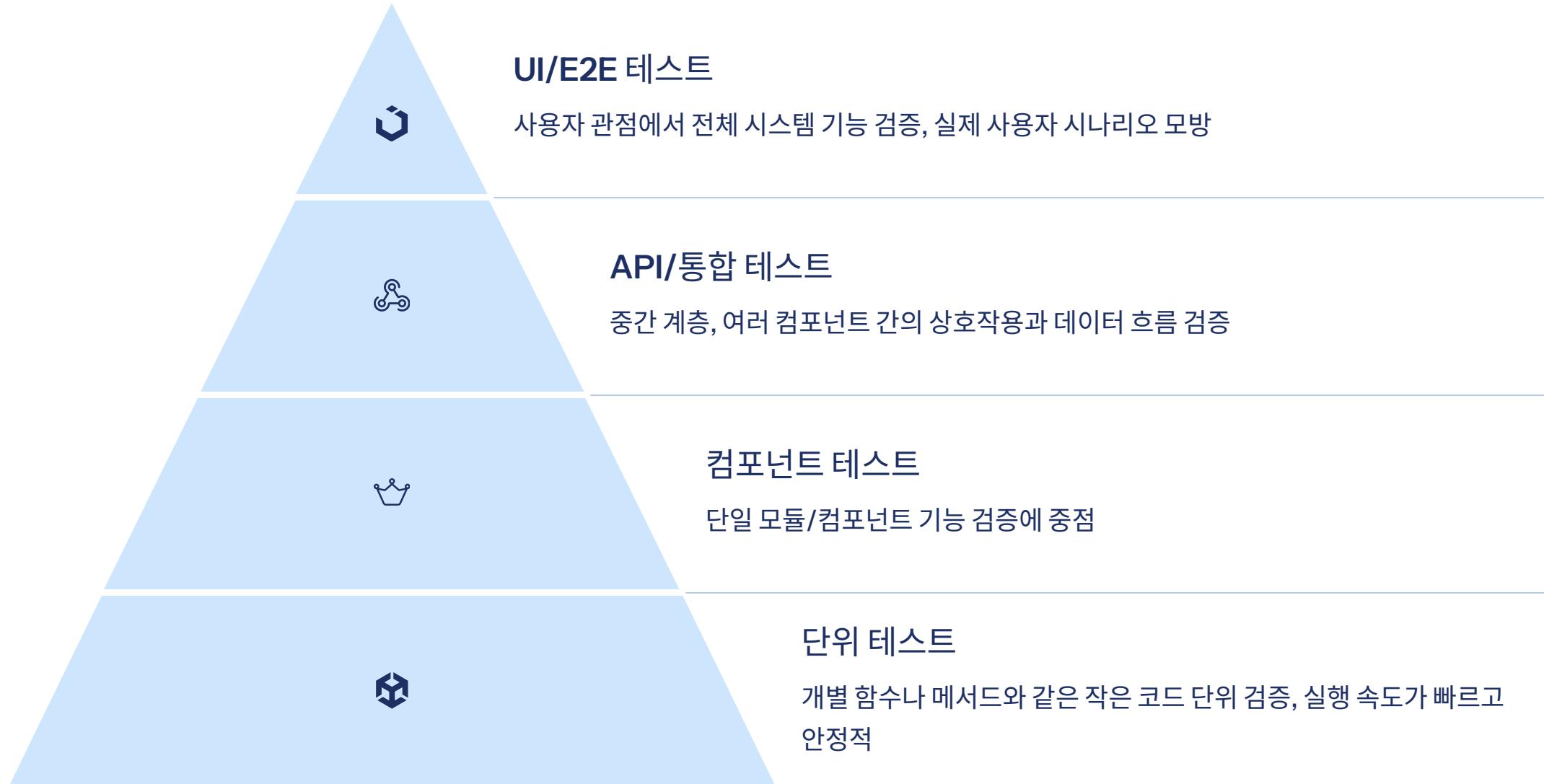
특정 테스트 도구에 의존하게 되어, 도구 자체의 한계나 문제에 영향을 받을 수 있음



학습 곡선

효과적인 자동화 테스트를 위해서는 특정 기술과 도구에 대한 학습이 필요함

자동화 테스트 유형: 범위와 목적에 따라 여러 유형



- 하위 계층 테스트일수록 더 많은 수의 테스트를 구현, 상위 계층으로 갈수록 테스트 수를 줄이는 것이 권장됨

자동화 테스트 도구 선택 기준



프로그래밍 언어 호환성

- 테스트 대상 애플리케이션 언어와의 호환성 고려
- Java: JUnit, TestNG 활용
- JavaScript/TypeScript: Jest, Mocha, Cypress 등 선택



확장성과 유지보수성

- 프로젝트 규모 확장에 따른 대응력 확인
- 테스트 코드의 재사용성, 모듈화, 구조화 지원 기능
- 대규모 프로젝트: 페이지 객체 모델(POM) 지원 도구 선호



커뮤니티 지원 및 문서화

- 활발한 커뮤니티와 풍부한 문서 확인
- GitHub 스타 수, Stack Overflow 활동 점검
- 오픈 소스: 업데이트 빈도와 활성 기여자 수 체크



CI/CD 통합 용이성

- Jenkins, GitHub Actions, CircleCI 등과의 통합성
- 테스트 결과 보고서 자동 생성 기능
- 실패 테스트 알림 및 커버리지 측정 기능 확인

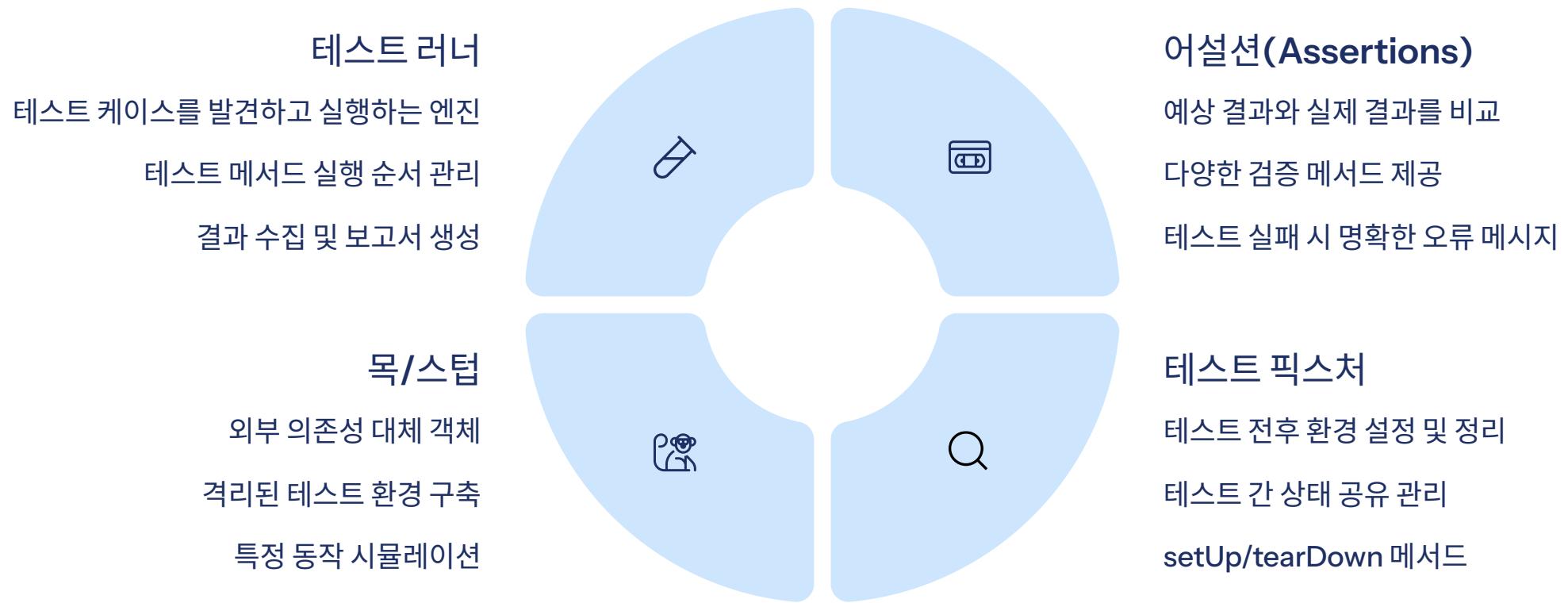
주요 자동화 테스트 도구 비교

도구명	유형	언어	특징	용도
JUnit 5	단위 테스트	Java	확장성, 파라미터화 테스트	Java 코드 단위 테스트
pytest	단위/통합 테스트	Python	간결한 문법, 유연한 픽스처	Python 코드 테스트
Jest	단위/통합 테스트	JavaScript	스냅샷 테스팅, 모킹 내장	React 등 JS 프레임워크
Selenium	UI/E2E 테스트	다양함	브라우저 자동화, 넓은 지원	웹 애플리케이션 E2E 테스트
Cypress	UI/E2E 테스트	JavaScript	실시간 리로드, 디버깅 용이	모던 웹 앱 E2E 테스트
Postman	API 테스트	JavaScript	GUI 인터페이스, 컬렉션	RESTful API 테스트
JMeter	성능 테스트	Java 기반	분산 테스팅, 다양한 프로토콜	부하/성능 테스트

- 프로젝트의 특성과 요구사항에 따라 적절한 도구를 선택하는 것이 중요
 - 예) Java 기반 백엔드 애플리케이션 단위 테스트에는 JUnit, 프론트엔드 React 애플리케이션에는 Jest+Cypress 조합이 효과적 등
- 여러 도구를 조합하여 사용하는 추세 강화되고 있음, 단위 테스트 프레임워크와 E2E 테스트 도구를 함께 활용하는 경우 많음
- 도구 선택 시 팀의 기술 스택과 학습 곡선도 중요한 고려 사항

단위 테스트 프레임워크 개요

단위 테스트 프레임워크는 개별 코드 단위(함수, 메서드, 클래스 등)를 효율적으로 테스트하기 위한 도구



- 현대적인 단위 테스트 프레임워크의 특징:
 - 테스트 작성을 쉽게 만들고, 테스트 실행 속도를 최적화
 - 테스트 결과를 명확하게 보고하는 기능 제공
 - 파라미터화 테스트, 태그 기반 테스트 실행, 테스트 그룹화 등의 고급 기능 지원
 - 대규모 테스트 스위트를 효율적으로 관리 가능

JUnit 5 기본 구조 및 기능

JUnit 5 아키텍처

JUnit 5는 세 가지 주요 모듈로 구성

- **JUnit Platform:** 테스트 프레임워크를 JVM에서 실행하기 위한 기반
- **JUnit Jupiter:** JUnit 5에서 테스트 작성을 위한 새로운 프로그래밍 모델
- **JUnit Vintage:** JUnit 3/4 기반 테스트를 JUnit 5 플랫폼에서 실행하기 위한 지원

주요 애노테이션

- **@Test:** 테스트 메서드 지정
- **@BeforeEach:** 각 테스트 전에 실행
- **@AfterEach:** 각 테스트 후에 실행
- **@BeforeAll:** 모든 테스트 전에 한번 실행 (static)
- **@AfterAll:** 모든 테스트 후에 한번 실행 (static)
- **@DisplayName:** 테스트 이름 지정
- **@Disabled:** 테스트 비활성화

JUnit 5 주요 기능

- **확장 모델:** @ExtendWith를 통한 유연한 확장 가능
- **조건부 테스트:** @EnabledOnOs, @EnabledIf 등
- **파라미터화 테스트:** @ParameterizedTest, @ValueSource 등
- **중첩 테스트:** @Nested를 사용한 계층적 테스트 구조
- **태깅 및 필터링:** @Tag로 테스트 분류 및 선택적 실행
- **반복 테스트:** @RepeatedTest로 테스트 반복 실행

JUnit 5 테스트 코드 예시

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void setUp() {
        calculator = new Calculator();
        System.out.println("테스트 전에 실행됩니다.");
    }

    @Test
    @DisplayName("덧셈 연산 테스트")
    void testAddition() {
        assertEquals(5, calculator.add(2, 3), "2 + 3은 5가 되어야 합니다.");
    }

    @Test
    @DisplayName("0으로 나누기 시 예외 발생 테스트")
    void testDivideByZero() {
        Exception exception = assertThrows(
            ArithmeticException.class,
            () -> calculator.divide(1, 0)
        );
        assertEquals("0으로 나눌 수 없습니다.", exception.getMessage());
    }

    @ParameterizedTest
    @CsvSource({
        "1, 1, 2",
        "5, 3, 8",
        "10, -5, 5"
    })
    @DisplayName("파라미터화된 덧셈 테스트")
    void testAdditionWithParameters(int a, int b, int expected) {
        assertEquals(expected, calculator.add(a, b));
    }

    @Nested
    @DisplayName("곱셈 테스트 그룹")
    class MultiplicationTests {

        @Test
        @DisplayName("양수 곱셈")
        void testPositiveNumbers() {
            assertEquals(6, calculator.multiply(2, 3));
        }

        @Test
        @DisplayName("음수 포함 곱셈")
        void testWithNegativeNumber() {
            assertEquals(-6, calculator.multiply(2, -3));
        }
    }

    @AfterEach
    void tearDown() {
        System.out.println("테스트 후에 실행됩니다.");
    }
}
```

pytest 프레임워크 소개

pytest는 Python 테스트를 위한 프레임워크로, 단순하면서도 강력한 기능을 제공



간결한 문법

- 최소한의 보일러플레이트 코드로 테스트 작성 가능
 - 보일러플레이트 코드: 여러 가지 상황에서 거의 또는 전혀 변경하지 않고 재사용할 수 있는 컴퓨터 언어 텍스트
- Python의 `assert` 문을 그대로 사용하여 직관적인 테스트 코드 작성



강력한 픽스처

- 모듈화된 픽스처 시스템으로 테스트 환경 구성
- 함수, 클래스, 모듈 또는 전체 세션 범위에서 픽스처 정의 가능
- 의존성 주입 방식으로 테스트에 제공



플러그인 생태계

- 다양한 확장 기능과 통합 지원
- 파라미터화 테스트, 마커를 통한 테스트 분류 지원



상세한 보고서

- 실패 시 명확한 정보와 디버깅 지원
- `xfail/skip`을 통한 테스트 제어 등 다양한 기능 제공
- 복잡한 테스트 시나리오를 효과적으로 관리

pytest 테스트 코드 예시

```
# test_calculator.py
import pytest

class Calculator:
    def add(self, a, b):
        return a + b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("0으로 나눌 수 없습니다")
        return a / b

# 픽스처 정의
@pytest.fixture
def calculator():
    """테스트에 사용할 계산기 객체를 제공하는 픽스처"""
    calc = Calculator()
    print("픽스처 설정")
    yield calc # 테스트에 객체 제공
    print("픽스처 정리")

# 기본 테스트
def test_addition(calculator):
    assert calculator.add(2, 3) == 5

# 예외 테스트
def test_divide_by_zero(calculator):
    with pytest.raises(ValueError) as excinfo:
        calculator.divide(1, 0)
    assert "0으로 나눌 수 없습니다" in str(excinfo.value)

# 파라미터화 테스트
@pytest.mark.parametrize("a, b, expected", [
    (1, 1, 2),
    (5, 3, 8),
    (10, -5, 5),
    (0, 0, 0)
])
def test_addition_parametrized(calculator, a, b, expected):
    assert calculator.add(a, b) == expected

# 마커를 사용한 테스트 분류
@pytest.mark.slow
def test_complex_calculation(calculator):
    result = 0
    for i in range(1000000):
        result = calculator.add(result, i)
    assert result == 4999995000

# 건너뛰기 테스트
@pytest.mark.skip(reason="아직 구현되지 않음")
def test_future_feature():
    assert False

# 실패 예상 테스트
@pytest.mark.xfail
def test_known_bug():
    assert 1 == 2 # 현재는 실패하지만 곧 수정될 예정
```

테스트 더블의 이해

- 테스트 더블(Test Double): 실제 객체를 대신하여 테스트에 사용되는 대체 객체
- 주요 목적: 외부 의존성 제어 및 테스트 격리
- 개념 정립: 마틴 파울러와 제라드 메스자로스
- 주요 특징: 단위 테스트의 신뢰성과 속도 향상



더미(Dummy)

전달은 되지만 실제로 사용되지 않는 객체로, 단순히 파라미터 목록을 채우기 위해 사용



스텁(Stub)

미리 정의된 응답을 제공하는 객체로, 테스트 중인 코드가 간접 입력을 얻을 수 있게 함



스파이(Spy)

호출된 내용을 기록하는 스텁으로, 간접 출력을 캡처하여 나중에 검증할 수 있음



목(Mock)

사전에 프로그래밍된 기대치를 가진 객체로, 예상대로 사용되었는지 스스로 검증함



페이크(Fake)

실제 구현과 유사하지만 더 단순한 구현을 가진 객체(예: 인메모리 데이터베이스)

Mockito를 활용한 모킹 예제 (1/2)

```
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

class OrderServiceTest {

    @Test
    void processOrderShouldCalculateCorrectTotal() {
        // 목 객체 생성
        PaymentGateway paymentGatewayMock = mock(PaymentGateway.class);
        InventoryService inventoryServiceMock = mock(InventoryService.class);

        // 목 객체 동작 정의
        when(inventoryServiceMock.isInStock("ITEM001")).thenReturn(true);
        when(paymentGatewayMock.processPayment(anyDouble())).thenReturn(true);

        // 테스트 대상 객체 생성 및 목 주입
        OrderService orderService = new OrderService(paymentGatewayMock, inventoryServiceMock);

        // 테스트 실행
        Order order = new Order("ITEM001", 2, 500.0);
        boolean result = orderService.processOrder(order);

        // 검증
        assertTrue(result);

        // 목 객체 호출 검증
        verify(inventoryServiceMock).isInStock("ITEM001");
        verify(paymentGatewayMock).processPayment(1000.0);
        verify(inventoryServiceMock).decreaseStock("ITEM001", 2);
    }

    .....
}
```

- Java의 인기 있는 모킹 프레임워크인 Mockito를 사용한 테스트 예시
- OrderService 클래스가 외부 서비스인 PaymentGateway와 InventoryService에 의존하는 상황에서, 이러한 외부 의존성을 목 객체로 대체하여 테스트
- Mockito의 mock() 메서드로 목 객체를 생성하고, when().thenReturn() 구문으로 목 객체의 동작을 정의
- 테스트 실행 후에는 verify() 메서드를 사용하여 목 객체가 예상대로 호출되었는지 검증

Mockito를 활용한 모킹 예제 (2/2)

```
class OrderServiceTest {  
    .....  
  
    @Test  
    void processOrderShouldFailWhenItemNotInStock() {  
        // 목 객체 생성  
        PaymentGateway paymentGatewayMock = mock(PaymentGateway.class);  
        InventoryService inventoryServiceMock = mock(InventoryService.class);  
  
        // 재고 없음 상황 시뮬레이션  
        when(inventoryServiceMock.isInStock("ITEM001")).thenReturn(false);  
  
        OrderService orderService = new OrderService(paymentGatewayMock, inventoryServiceMock);  
  
        Order order = new Order("ITEM001", 2, 500.0);  
        boolean result = orderService.processOrder(order);  
  
        // 주문 처리 실패 확인  
        assertFalse(result);  
  
        // 재고 확인만 호출되고 결제는 호출되지 않았는지 검증  
        verify(inventoryServiceMock).isInStock("ITEM001");  
        verify(paymentGatewayMock, never()).processPayment(anyDouble());  
        verify(inventoryServiceMock, never()).decreaseStock(anyString(), anyInt());  
    }  
}
```

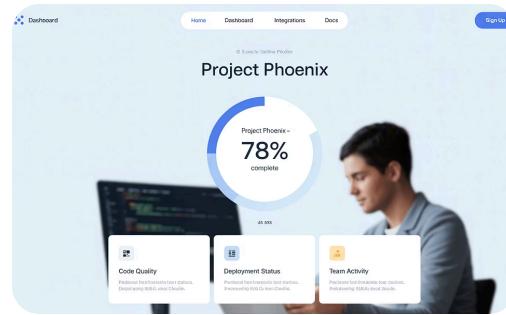
- 두 번째 테스트에서는 `never()` 메서드를 사용하여 특정 메서드가 호출되지 않았음을 확인
- 모킹 기법을 통해 외부 시스템에 의존하지 않고 다양한 시나리오를 테스트할 수 있음

테스트 주도 개발(Test-Driven Development, TDD)



- 테스트를 먼저 작성하고, 그 테스트를 통과하는 코드를 개발하는 방법론
- Red-Green-Refactor 사이클을 반복하는 개발 방식
- 실패하는 테스트(Red) → 테스트 통과 코드(Green) → 코드 품질 개선(Refactor)

TDD의 특징



설계 방법론

단순한 테스트 기법이 아닌 설계 방법론

인터페이스 중심

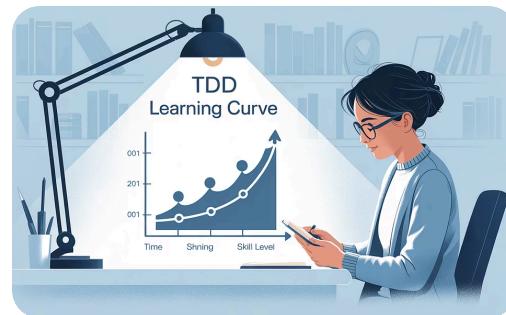
코드의 사용성과 인터페이스에 먼저 집중

지속적 피드백

지속적인 피드백 제공으로 코드 품질 향상

개발자 자신감

개발자의 자신감 향상



유지보수성 향상

코드 유지보수성과 확장성 향상

도입 고려사항

학습 곡선 존재, 프로젝트 특성에 맞게 적용 필요

TDD 실전 예제

1단계: 실패하는 테스트 작성

```
```python def test_calculator_add(): calc = Calculator()  
 assert calc.add(2, 3) == 5 ```
```

- Calculator 클래스와 add 메서드가 아직 미구현 상태
- 테스트는 의도적으로 실패하도록 설계



## 3단계: 테스트 추가

```
```python def test_calculator_add_different_values():  
    calc = Calculator() assert calc.add(1, 1) == 2 ```
```

- 새로운 테스트 케이스 추가
- 기존 구현의 한계를 드러내는 테스트



5단계: 리팩토링

- 코드가 이미 단순하여 추가 리팩토링 불필요
- 복잡한 경우: 코드 구조, 명명법, 중복 제거 등 개선
- 테스트를 통과하면서 코드 품질 향상에 집중



2단계: 최소한의 코드 구현

```
```python class Calculator: def add(self, a, b): return 5  
...```
```

- 테스트 통과를 위한 가장 단순한 구현
- 하드코딩된 값으로 특정 테스트만 통과



## 4단계: 구현 개선

```
```python class Calculator: def add(self, a, b): return a  
+ b ...```
```

- 모든 테스트 케이스 통과하는 실제 구현
- 하드코딩이 아닌 실제 덧셈 로직 구현



자동화 테스트 도입 전략

1 현황 분석 및 목표 설정

- 현재 테스트 프로세스, 결함 발생 패턴, 개발 방법론 분석
- 팀의 기술 역량 평가
- 자동화를 통한 구체적 목표 설정 (회귀 테스트 시간 단축, 배포 주기 단축 등)

2 자동화 대상 선정

- 실행 빈도가 높은 테스트
- 회귀 위험이 높은 영역
- 수동 테스트가 어렵거나 시간 소요가 많은 영역
- 안정적인 기능 (자주 변경되지 않는 영역)
- 비즈니스 중요도가 높은 기능

3 도구 선택 및 프레임워크 구축

- 테스트 유형, 애플리케이션 기술 스택, 팀 역량 고려
- 계층별 적합한 도구 선택
- 단위 테스트: JUnit, pytest 등
- API 테스트: Rest Assured, Postman 등
- UI 테스트: Selenium, Cypress 등
- 재사용 가능한 테스트 프레임워크 구축

4 점진적 구현 및 지속적 개선

- 작은 영역부터 시작하여 점진적 확장
- 초기에는 쉽게 성공할 수 있는 영역에 집중
- 빠른 성과를 통한 팀의 자신감과 지원 확보
- 지속적인 피드백 수집 및 프레임워크 개선
- 테스트의 안정성, 유지보수성, 효율성 향상

CI/CD 파이프라인과 자동화 테스트

CI/CD(Continuous Integration/Continuous Delivery or Deployment) 파이프라인의 개념

- CI/CD(지속적 통합/지속적 배포)는 코드 변경을 빠르고 안정적으로 프로덕션에 전달하는 자동화 프로세스
- 파이프라인 내 자동화 테스트는 각 단계에서 코드 품질과 기능을 검증하는 중추적 역할 수행

파이프라인 단계별 테스트 역할

- **CI 단계:** 코드 변경 통합 시마다 자동 빌드 및 테스트 실행
 - 문제 조기 발견 및 해결 지원
 - 주로 단위 테스트와 일부 통합 테스트 실행
- **CD 단계:** 포괄적 통합 테스트, 시스템 테스트, 성능 테스트 등 실행
 - 배포 준비 상태 검증
 - 자동화 테스트는 파이프라인의 "품질 게이트" 역할 수행
 - 테스트 통과 코드만 다음 단계로 진행 가능



테스트 코드의 유지보수성

테스트 코드 구조화

- 체계적인 구조가 유지보수에 필수적
- 공통 기능은 유ти리티 클래스나 헬퍼 메서드로 추출
- 일관된 이름 규칙 사용 (예: "testLoginWithValidCredentials")
- 테스트 대상과 시나리오를 명확히 표현하는 이름 사용

테스트 격리 및 독립성

- 각 테스트는 독립적으로 실행 가능해야 함
- 테스트 간 의존성은 유지보수를 어렵게 함
- 각 테스트 전후에 상태 초기화 필요
- 테스트 데이터 분리 및 외부 의존성 모킹
- DB 테스트는 트랜잭션 롤백이나 인메모리 데이터베이스 활용

테스트 코드 리팩토링

- 테스트 코드도 정기적인 리팩토링 필요
- 중복된 설정 코드와 복잡한 검증 로직 제거
- Given-When-Then 패턴 적용으로 테스트 구조 명확화
- 픽스처 설정, 테스트 실행, 결과 검증 단계 분리

테스트 가독성 향상 기법

테스트 코드 가독성이 중요한 이유

- 테스트 코드는 프로덕션 코드의 살아있는 문서 역할
- 가독성 높은 테스트는 시스템 동작 방식 이해에 도움
- 새로운 팀원의 온보딩 과정 용이
- 버그 수정 시 문제 진단 속도 향상
- 테스트 코드 가독성은 테스트 신뢰성과 직결
- 복잡하고 이해하기 어려운 테스트는 오류 발생 가능성 높음
- 테스트 실패 시 원인 파악과 해결이 용이

가독성 향상을 위한 구체적 기법

1. **의도가 명확한 이름 사용:** "test1" 대신 "testUserCannotLoginWithInvalidPassword"와 같이 구체적인 이름을 사용
2. **Given-When-Then 패턴 적용:** 테스트 준비, 실행, 검증 단계를 명확히 구분하여 테스트 흐름을 이해하기 쉽게 함
3. **설명적인 어설션 사용:** 단순히 assertEquals(expected, actual) 대신 assertEquals("사용자 로그인 실패해야 함", expected, actual)과 같이 실패 시 메시지를 포함
4. **불필요한 세부 사항 숨기기:** 테스트의 핵심과 관련 없는 설정 코드는 헬퍼 메서드로 추출
5. **테스트 데이터 빌더 패턴 활용:** 복잡한 테스트 객체 생성을 명확하고 유연하게 만듦
6. **주석 대신 코드로 표현:** 주석으로 설명하기보다 코드 자체가 의도를 나타내도록 작성

효과적인 테스트 데이터 관리



테스트 데이터 생성 전략

- 다양한 시나리오를 커버하는 테스트 데이터 필요
- 코드 내 직접 생성, 외부 파일 로드, 테스트 퍽스처 활용 가능
- 경계값, 특수 케이스, 무효한 입력 등 다양한 조건 고려



테스트 데이터 검증

- 안정적인 테스트 실행을 위한 데이터 검증 필수
- 각 테스트는 독립적인 데이터 세트 사용 또는 상태 초기화
- DB 테스트: 트랜잭션 롤백, 전용 스키마, 인메모리 DB 활용



민감한 테스트 데이터 처리

- 실제 사용자 데이터 사용 시 보안 및 개인정보 보호 위험
- 가명화(민감한 정보를 가명이나 별칭으로 대체)와 익명화(개인식별불가) 기법 적용
- 테스트 데이터에 대한 접근 제어 및 암호화 조치 필요



테스트 데이터 버전 관리

- 테스트 데이터도 코드와 마찬가지로 버전 관리 필요
- 외부 파일은 소스 코드 저장소에 함께 저장하여 동기화
- 대용량 데이터는 데이터 자체보다 생성 스크립트 버전 관리가 효율적

테스트 커버리지 측정과 활용



커버리지 목표 설정

- 프로젝트 성격, 중요도, 리스크 수준에 따라 차별화된 목표 설정
- 일반적 기준: 70-80%의 라인 커버리지
- 핵심 모듈: 90-100% 목표
- 중요도 낮은 영역: 더 낮은 목표 설정 가능



커버리지 측정 도구

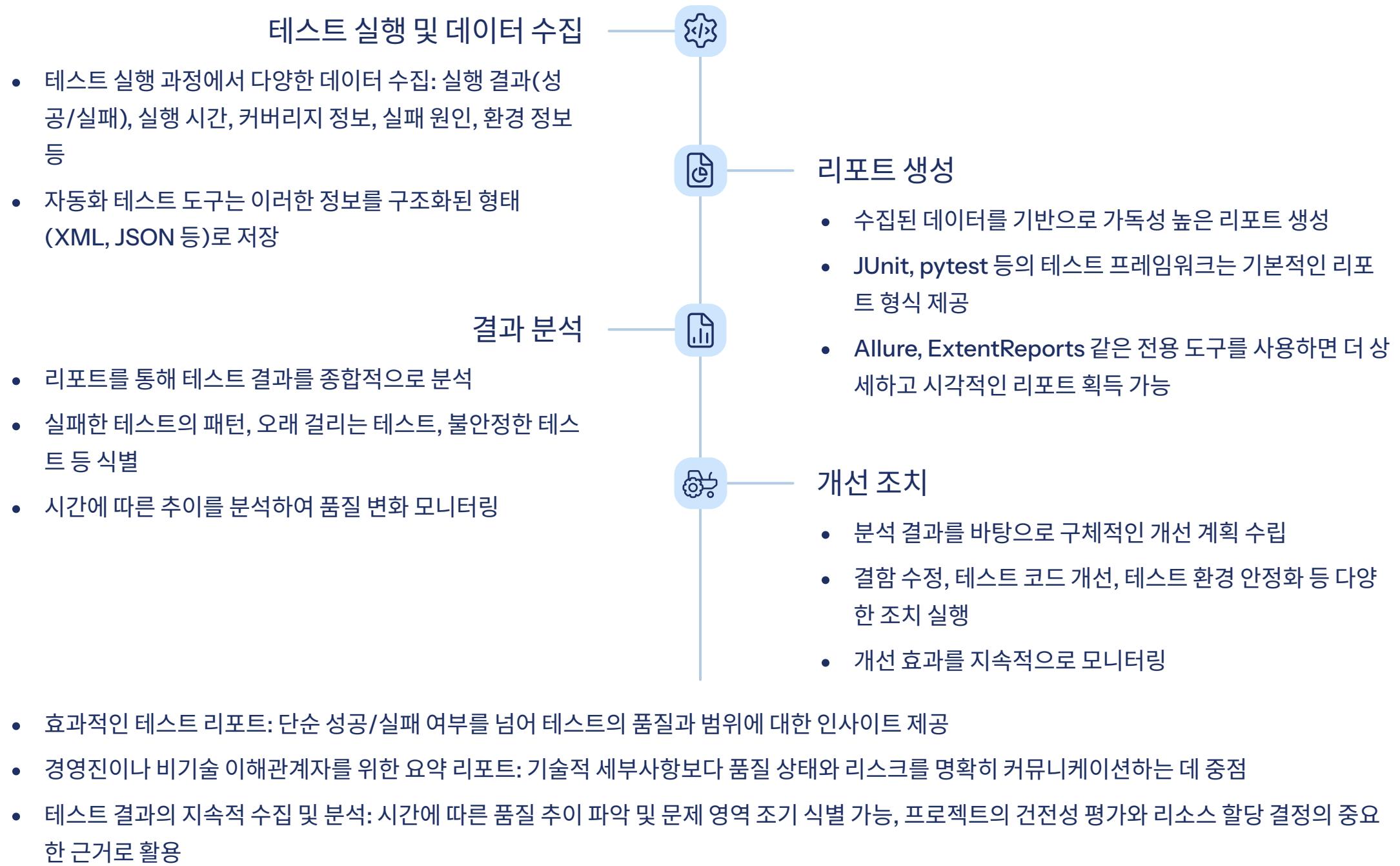
- 언어별 전용 도구: JaCoCo(Java), coverage.py(Python), Istanbul(JavaScript) 등
- 주요 기능: IDE 통합, CI/CD 파이프라인 연동, 상세 보고서 생성



커버리지의 한계 이해

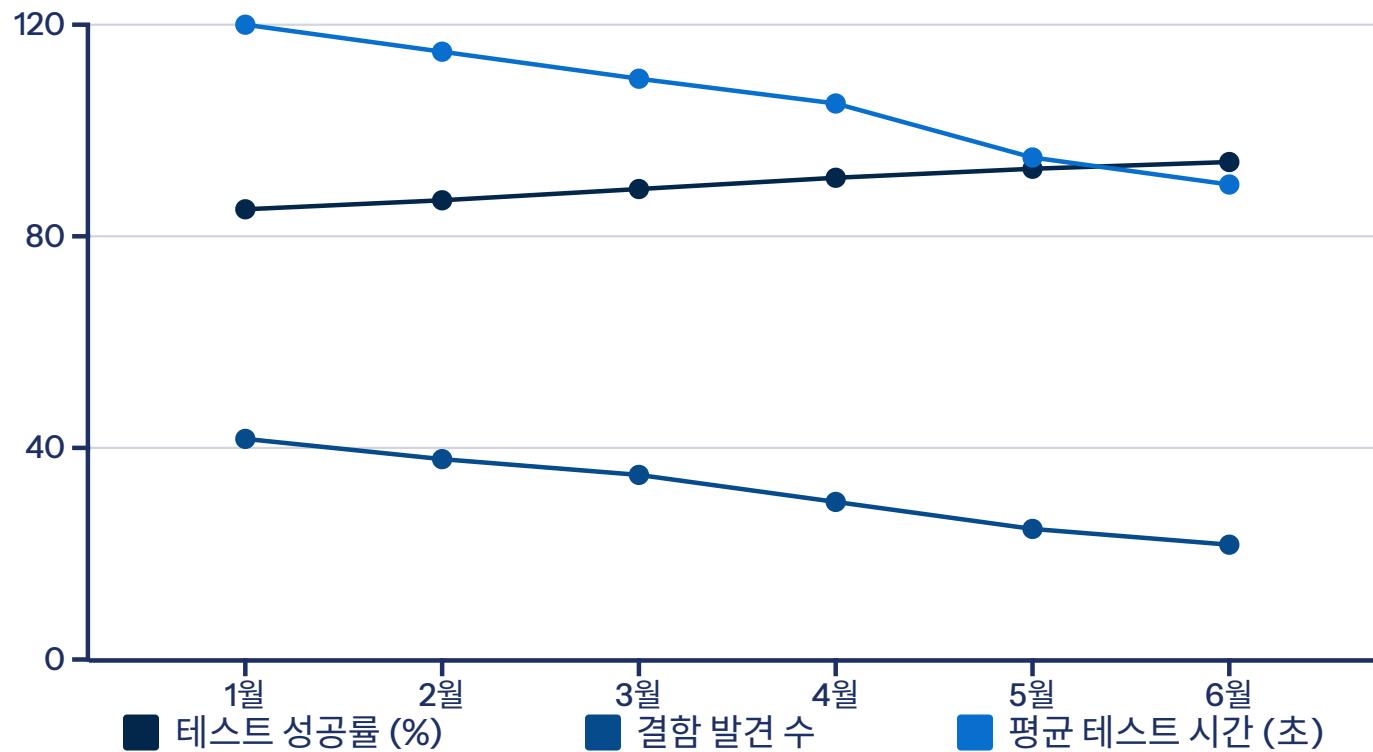
- 높은 커버리지 ≠ 좋은 테스트 품질
- 코드 실행 여부와 모든 시나리오 테스트는 별개의 문제
- 커버리지는 테스트 품질의 필요조건(O), 충분조건(X)
- 다른 품질 지표와 함께 종합적 평가 필요

테스트 리포트 작성 및 분석



데이터 기반 테스트 개선

- 데이터 기반 테스트 개선은 테스트 프로세스와 결과에서 수집한 데이터를 분석하여 테스트 전략과 실행을 최적화하는 접근 방식
- 데이터를 통해 테스트 프로세스의 효율성과 효과성을 측정할 수 있음



핵심 테스트 메트릭

- 수집/분석해야 할 주요 메트릭
 - 테스트 실행 시간, 테스트 성공률, 코드 커버리지, 결함 발견률, 결함 밀도, 결함 재발률, 회귀 결함 비율 등
- 시간에 따른 추이를 분석하여 개선 여부를 판단하는 기준이 됨



패턴 및 추세 분석

- 테스트 데이터에서 패턴을 발견하는 것은 문제의 근본 원인을 찾는데 도움
- 특정 모듈에서 결함이 집중적으로 발생하거나, 특정 유형의 테스트가 자주 실패하는 경우, 이는 해당 영역에 대한 집중적인 개선이 필요함 시사



우선순위 최적화

- 제한된 리소스를 효율적으로 활용하기 위해, 데이터 기반으로 테스트 우선순위를 결정
- 과거에 결함이 많이 발견된 영역, 코드 변경이 잦은 부분, 비즈니스 중요도가 높은 기능 등에 테스트 리소스를 집중하여 ROI를 극대화

테스트 자동화의 ROI 측정

테스트 자동화 투자 비용

- 테스트 자동화의 ROI를 계산하기 위해서는 먼저 투자 비용을 정확히 파악해야 함
- 주요 비용 요소:
 - **초기 개발 비용:** 자동화 프레임워크 설계, 테스트 스크립트 작성, 환경 구축 등에 소요되는 인건비와 시간
 - **도구 및 인프라 비용:** 테스트 도구 라이선스, 테스트 환경 구축 및 유지 비용
 - **교육 및 역량 개발 비용:** 팀원 교육, 컨설팅, 기술 지원 등
 - **유지보수 비용:** 테스트 스크립트 업데이트, 버그 수정, 환경 관리 등에 지속적으로 소요되는 비용
- 테스트 자동화의 ROI는 일반적으로 시간이 지남에 따라 증가함
- 초기에는 높은 투자 비용으로 인해 ROI가 낮거나 마이너스일 수 있음
- 테스트 실행 횟수가 증가함에 따라 투자 수익이 점차 증가함
- ROI 계산 시 충분한 시간 범위(보통 1-2년)를 고려해야 함
- $ROI = (\text{자동화 테스트 효과} - \text{자동화 테스트 비용}) / \text{자동화 테스트 비용} \times 100\%$

테스트 자동화 효과 측정

- 테스트 자동화의 효과는 정량적, 정성적 측면에서 측정 가능:
- **시간 절약:** 수동 테스트 대비 절약된 시간 = (수동 테스트 시간 - 자동화 테스트 실행 및 유지보수 시간) × 테스트 실행 횟수
- **비용 절감:** 절약된 시간을 인건비로 환산한 금액
- **결함 조기 발견:** 개발 초기에 발견된 결함의 수와 수정 비용 절감 효과
- **출시 주기 단축:** 테스트 시간 단축으로 인한 출시 주기 감소 및 이로 인한 비즈니스 가치
- **품질 향상:** 사용자 경험 개선, 고객 만족도 증가, 브랜드 가치 상승 등의 간접적 효과

테스트 자동화 성공 사례 연구

금융 서비스 산업의 테스트 자동화 성공 사례

1. 글로벌 은행 – 테스트 주기 60% 단축 및 결함 누수 75% 감소

- 주요 성과: · 테스트 주기: 12주 → 4.5주 단축 · 결함 누수: 75% 감소 · 테스트 자동화 범위: 84%까지 확대
- 핵심 전략: · 테스트 조직 통합 · Broadcom ARD(Agile Requirements Designer) 기반 비즈니스 프로세스 중심 테스트 설계 · Jenkins 및 Selenium 활용한 병렬 실행 인프라 구축
- 출처: [Accenture 사례 연구\(accenture.com\)](#)

2. Fortune 500 은행 – 비용 63% 절감 및 100% 테스트 커버리지 달성

- 주요 성과: · 테스트 실행 시간: 25인일 → 3인일 단축 · 테스트 커버리지: 100% 달성 · 비용: 63% 절감
- 핵심 전략: · Avo Assure 플랫폼 활용 (1,500개+ 테스트 케이스 자동화) · 병렬 실행 구현 · 이종 시스템 간 엔드투엔드 테스트 구현
- 출처: [Avo Automation 사례 연구\(avoautomation.ai\)](#)

3. 대형 보험사 – 테스트 실행 시간 80% 단축

- 주요 성과: · 테스트 실행 시간: 80% 단축 · 결함: 30% 감소
- 핵심 전략: · Agile 기반 환경 관리 전략 수립 · 테스트 자동화로 엔드투엔드 테스트 가시성 확보
- 출처: [Cigniti 사례 연구\(Cigniti Technologies\)](#)

4. ING 은행 – 테스트 사이클 시간 단축 및 품질 향상

- 주요 성과: · 테스트 사이클 시간 단축 · 애플리케이션 품질 향상
- 핵심 전략: · Testinium 플랫폼 활용 (700개+ 자동화 스크립트 개발) · BDD(Behavior-Driven Development) 접근 방식 도입 · 테스트 프로세스 최적화
- 출처: [Testinium 사례 연구\(Testinium\)](#)

5. 핀테크 기업 – 테스트 시간 72% 단축 및 커버리지 향상

- 주요 성과: · 테스트 시간: 72% 단축 · 회귀 테스트 커버리지 향상
- 핵심 전략: · UiPath 기반 로우코드 자동화 도입 · 회귀 테스트 자동화 · 테스트 정확성 및 효율성 향상
- 출처: [CAI 사례 연구](#)

테스트 자동화의 미래 동향

AI/ML 기술의 혁신적 적용

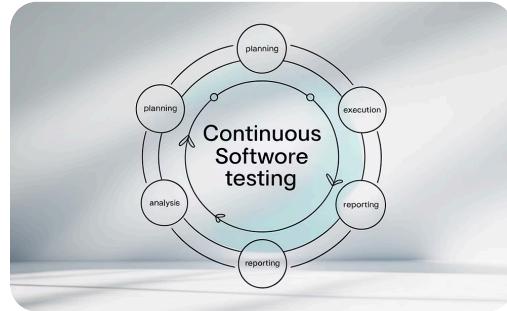
- 패턴 인식을 통한 자동 테스트 케이스 생성
- 코드 변경에 따른 지능적 테스트 우선순위 조정
- 테스트 결과에서 이상 패턴 자동 감지



AI/ML 기반 테스트

지속적 테스트 (Continuous Testing) 의 중요성 증가

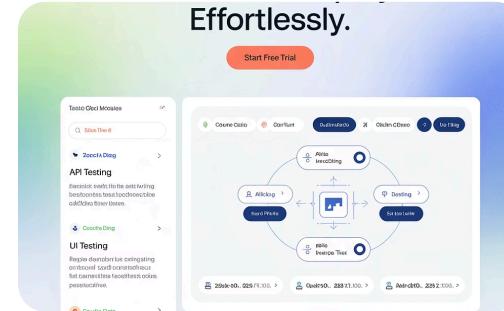
- 개발 주기 전반에 걸친 자동화된 테스트 실행
- 즉각적인 피드백 제공 방식으로 발전



지속적 테스트

새로운 아키텍처에 대응 하는 테스트 기법

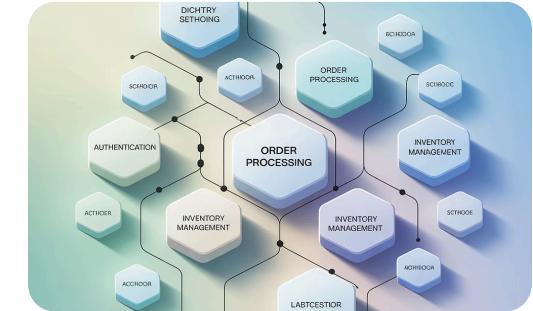
- 마이크로서비스 아키텍처 테스트 방법론
- 컨테이너화 및 클라우드 네이티브 애플리케이션 테스트
- 분산 시스템 테스트, 카오스 엔지니어링, 서비스 가상화 확산



로우코드 테스트 플랫폼

테스트 자동화의 진화 방향

- 단순 회귀 테스트 도구에서 통합 품질 보장 솔루션으로 발전
- 소프트웨어 품질을 전체적으로 관리하는 방향으로 확장



マイクロ서비스 테스트