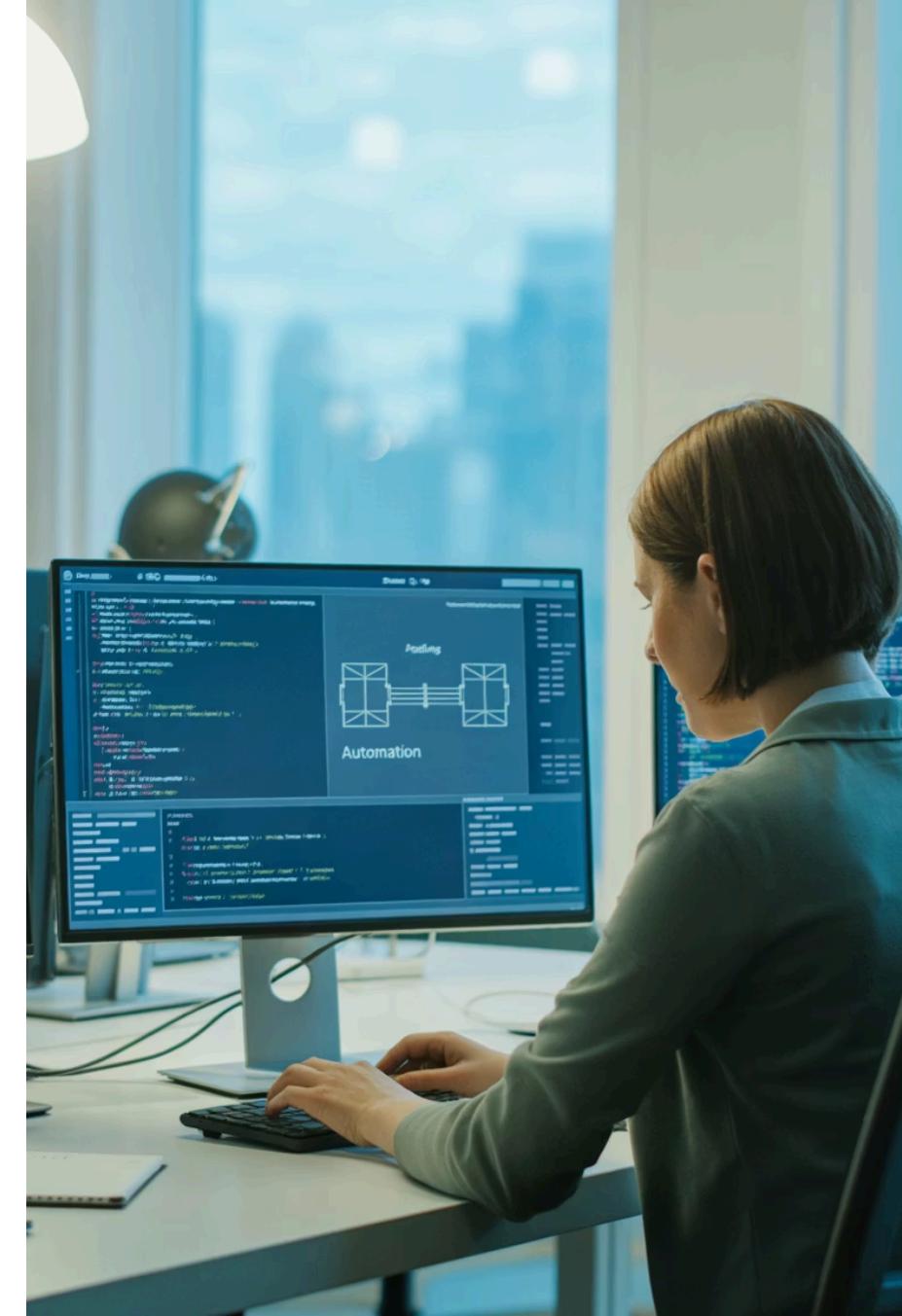


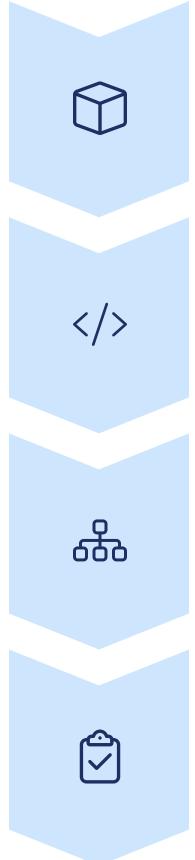
# 소프트웨어 테스팅 Day 3: 고급 자동화 및 테스트 전략

2025년 5월 30일

지은경 (KAIST 전산학부)



# Day 3: 고급 자동화 및 테스트 전략



## 테스트 더블 이해

Dummy, Stub, Spy, Mock, Fake의 개념과 활용 방법을 습득

## Mock 객체 활용 능력 향상

Python의 `unittest.mock`과 Java의 Mockito 등 주요 모킹 프레임워크 사용법 익힘

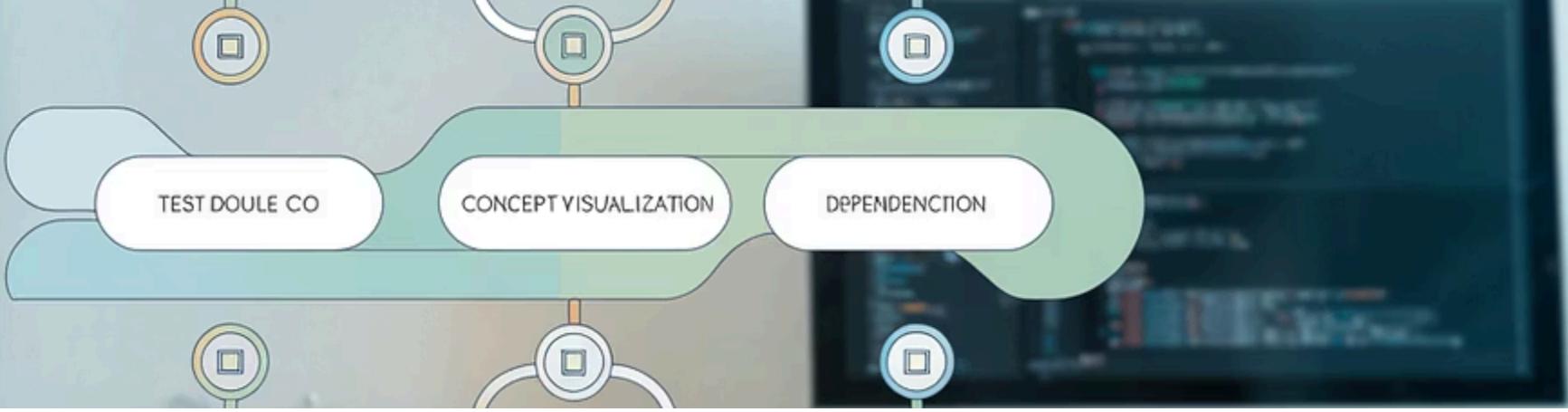
## 테스트 접근법 비교

TDD와 BDD의 차이점과 각 방법론의 장단점을 이해

## 실무 테스트 전략 수립

프로젝트 특성에 맞는 테스트 전략을 설계하고 문서화하는 방법을 습득





# 테스트 더블(Test Double)이란?



## 테스트 더블의 기본 개념

- 테스트 대상 시스템의 의존성을 대체하는 객체
- 테스트를 격리시키고 제어 가능하게 만듦
- 실제 의존성이 테스트하기 어렵거나 비용이 많이 들 때 활용



## 테스트 더블의 필요성

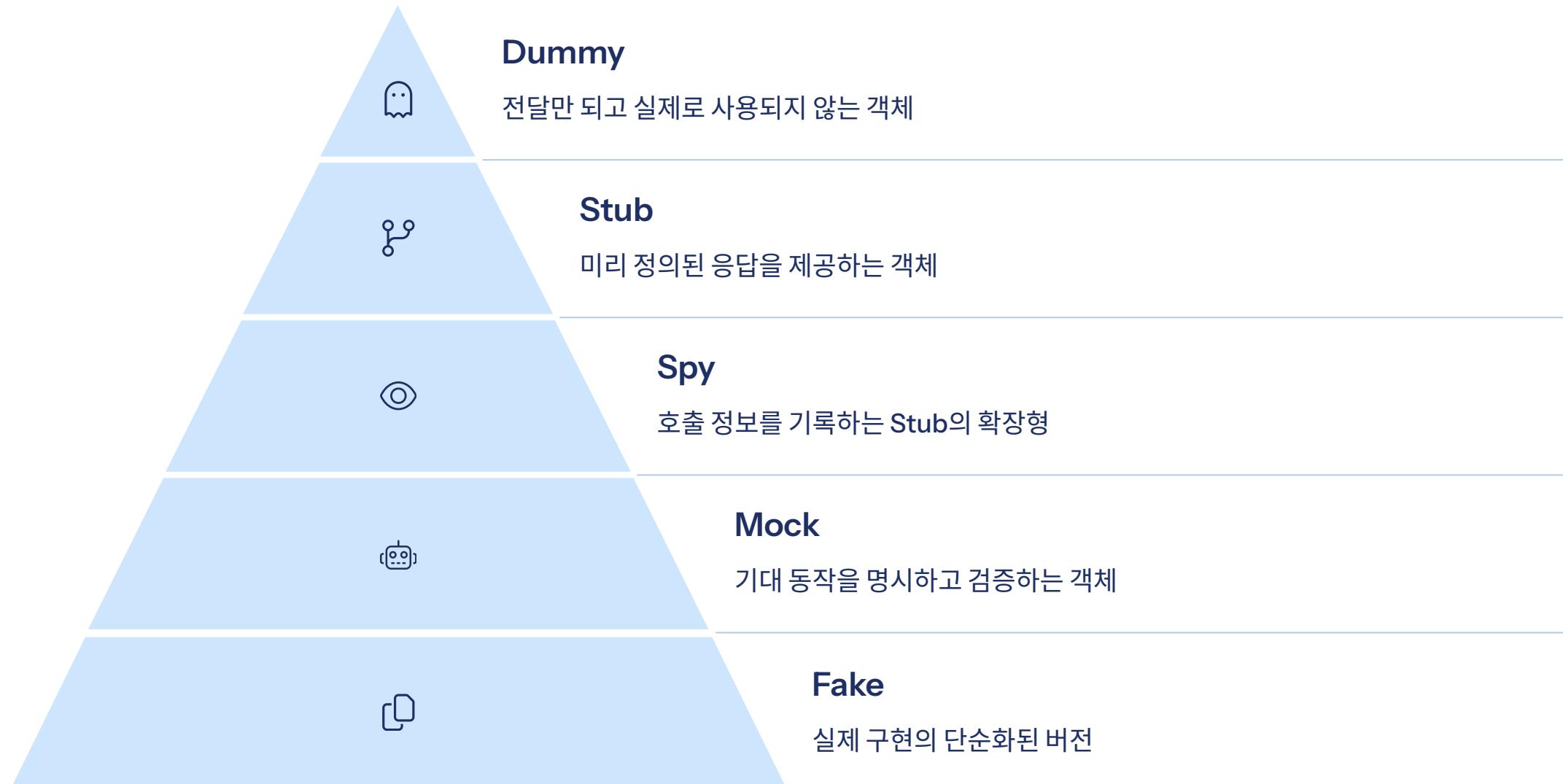
- 외부 서비스 의존성 제어
- 네트워크 통신 대체
- 데이터베이스 연결 시뮬레이션
- 테스트 실행 속도 개선
- 테스트 예측 가능성 확보



## 테스트 더블의 목적

- 테스트 속도 향상
- 결정적(deterministic) 테스트 환경 구성
- 특정 상황이나 예외 상황 시뮬레이션
- 미완성 기능의 사전 테스트 지원

# 테스트 더블의 5가지 유형



- 테스트 더블은 테스트 상황과 목적에 따라 적절히 선택하여 사용해야 함
- 각 유형은 서로 다른 문제를 해결하며, 때로는 여러 유형을 조합하여 사용할 수도 있음

# Dummy 객체 심층 분석

## Dummy 객체의 특징

- 테스트에서 파라미터 목록을 채우기 위해 전달되지만, 실제로는 사용되지 않는 객체
- 메서드가 호출되지 않으며, 단순히 존재함으로써 코드가 컴파일되거나 실행될 수 있게 함
- 주로 메서드 호출 시 필요한 파라미터 중 테스트와 직접적인 관련이 없는 객체를 전달할 때 활용
- 가장 간단한 형태의 테스트 더블

## Dummy 객체 구현 예시

```
# Python Dummy 예시
class DummyLogger:
    def log(self, message):
        pass # 아무 동작도 수행하지 않음

# 테스트 코드
def test_user_registration():
    dummy_logger = DummyLogger()
    user_service = UserService(dummy_logger)
    result = user_service.register("user1", "pass123")
    assert result is True
```

# Stub 객체 활용 기법

## Stub의 정의

미리 정의된 응답을 제공하는 객체로, 호출되는 메서드에 대해 고정된 결과를 반환함

## Stub의 장점

실제 구현체 대신 사용되어 테스트 환경을 제어할 수 있게 함

## Stub의 주요 용도

테스트 대상 객체가 의존하는 컴포넌트의 특정 상태나 응답을 시뮬레이션

## 활용 사례

특정 시나리오나 엣지 케이스 테스트, 네트워크 호출, API 응답, 데이터베이스 조회 등을 대체

## Stub 객체 구현 예시

```
# Python Stub 예시
class StubWeatherService:
    def get_temperature(self, city):
        return 25.0 # 항상 25도를 반환

# 테스트 코드
def test_weather_report():
    stub_service = StubWeatherService()
    weather_app = WeatherApp(stub_service)
    report = weather_app.generate_report("Seoul")
    assert "현재 기온: 25.0°C" in report
```

# Spy 객체의 활용

## Spy 객체의 특징과 목적

- Stub의 확장형으로, 호출에 대한 정보를 기록
- 미리 정의된 응답 제공과 함께 메서드 호출 횟수, 파라미터 등 정보 저장
- 테스트 대상 객체가 의존성을 어떻게 사용하는지 검증할 때 유용

## Spy 객체 구현 방법

- 호출 정보를 기록하기 위한 내부 상태(카운터, 리스트 등) 포함
- 테스트에서 이 정보를 사용해 나중에 검증 가능
- 실제 구현의 일부 기능 수행하면서 호출 정보 추적 가능
- Stub보다 더 다양한 검증이 가능한 장점

## Spy 객체 활용 예시

```
class EmailServiceSpy:  
    def __init__(self):  
        self.sent_emails = []  
        self.send_count = 0  
  
    def send_email(self, to, subject, body):  
        self.sent_emails.append((to, subject, body))  
        self.send_count += 1  
        return True # 항상 성공 반환  
  
# 테스트 코드  
def test_notification():  
    email_spy = EmailServiceSpy()  
    notifier = NotificationService(email_spy)  
    notifier.notify_users(["user1@example.com"], "알림")  
  
    assert email_spy.send_count == 1  
    assert email_spy.sent_emails[0][0] == "user1@example.com"
```

# Mock 객체의 이해

## Mock 객체의 핵심 개념

- Mock은 테스트 전에 기대 동작을 명시적으로 정의하고, 테스트 후에 검증하는 객체
- 단순 응답 제공을 넘어, 특정 메서드가 특정 인자로 호출되었는지, 호출 횟수 등을 검증
- 행위 기반 테스트(behavior verification)에 중점
- 테스트 대상 객체와 의존성 간의 상호작용 검증에 초점
- Spy와 유사하나, 기대치를 미리 설정하고 자동으로 검증하는 점이 차이점

## Mock 객체와 다른 테스트 더블의 차이점

- Stub: 단순히 응답 제공에 중점
- Spy: 호출 정보 기록에 중점
- Mock: 기대 동작 명세와 검증에 중점

## 사용 시 고려사항

- 테스트 더블 중 가장 강력한 기능 제공
- 테스트가 구현 세부사항에 과도하게 결합될 위험 존재
- 신중하게 사용해야 함
- 행위 검증이 중요한 경우에 적합

# Fake 객체 활용 전략

## Fake 객체의 특징

- 실제 구현의 단순화된 버전으로, 실제 동작을 가볍게 구현
- 인터페이스는 동일하지만 프로덕션 환경에서 사용하기에는 부적합한 간소화된 구현 제공
- 실제 구현체를 사용하는 것처럼 동작하면서도 테스트에 적합한 특성(속도, 격리성 등)을 갖춤

## 대표적인 Fake 객체 사례

- 인메모리 데이터베이스 - 실제 DB 대신 메모리에 데이터를 저장/조회하여 테스트 속도 향상
- 파일 시스템의 인메모리 구현 - 디스크 I/O 없이 파일 작업 시뮬레이션
- 경량화된 메시지 큐 - 실제 메시지 브로커 없이 메시지 전달 구현
- 단순화된 인증 서비스 - 보안 로직을 단순화하여 테스트 용이성 확보

```
# 인메모리 사용자 저장소 Fake 구현
class InMemoryUserRepository:
    def __init__(self):
        self.users = {} # 메모리에 사용자 저장

    def save(self, user):
        self.users[user.id] = user

    def find_by_id(self, user_id):
        return self.users.get(user_id)

    def find_all(self):
        return list(self.users.values())
```

# 테스트 더블 선택 가이드라인

## 테스트 목적 파악

- 단순 의존성 채우기: Dummy 사용
- 특정 응답 시뮬레이션: Stub 활용
- 상호작용 검증: Mock 또는 Spy 선택
- 복잡한 의존성의 가벼운 버전 필요: Fake 고려

## 상태 검증 vs 행위 검증

- 상태 검증(state verification) 목적: Stub/Fake 적합
- 행위 검증(behavior verification) 필요: Mock/Spy 적합
- 검증 유형에 따라 적절한 도구 선택 필요

## 복잡성 고려

- 가능한 한 간단한 테스트 더블 사용 권장
- 복잡성 증가 순서: Dummy → Stub → Spy → Mock → Fake
- 필요 이상의 복잡한 더블은 테스트 복잡성 증가 초래
- 테스트 목적에 맞는 최소한의 복잡성 유지

## 테스트 가독성과 유지보수성

- 테스트 더블이 코드 가독성과 유지보수성에 미치는 영향 고려
- 수동 구현한 간단한 더블이 때로는 더 이해하기 쉬움
- 모킹 프레임워크의 복잡성과 학습 비용 감안
- 팀 전체가 이해할 수 있는 방식 선택

# Mock 프레임워크: Python unittest.mock

## unittest.mock 소개

- Python 표준 라이브러리에 포함된 강력한 모킹 프레임워크 (Python 3.3부터 기본 제공)
- 객체의 메서드와 속성을 쉽게 대체하고 호출을 검증하는 기능 제공
- Mock 클래스를 중심으로 다양한 모킹 기능 제공
- patch 데코레이터나 컨텍스트 매니저를 통해 특정 객체나 모듈을 일시적으로 대체 가능
  - patch 데코레이터: 테스트 중 특정 객체나 클래스를 임시로 다른 것으로 대체(mock) 해주는 도구
  - [patch\('경로.이름'\)](#)은 지정된 경로에 있는 함수, 클래스, 객체를 임시로 mock 객체로 바꿈
  - 이 mock 객체는 테스트 함수에 자동으로 인자로 전달됨
  - 테스트 함수 실행이 끝나면 원래 객체로 되돌아감

## 주요 기능 및 사용법

```
from unittest.mock import Mock, patch

# Mock 객체 생성 및 설정
mock_service = Mock()
mock_service.get_data.return_value = [1, 2, 3]

# Mock 객체 사용
result = mock_service.get_data()
assert result == [1, 2, 3]

# 호출 검증
mock_service.get_data.assert_called_once()

# patch 데코레이터 사용
@patch('myapp.service.DataService')
def test_process_data(mock_data_service):
    mock_data_service.return_value.get_data.return_value = [1, 2, 3]
    # 테스트 로직...
```

# Mock 프레임워크: Java Mockito

## Mockito 소개

- Java에서 가장 인기 있는 모킹 프레임워크로, 간결한 API와 강력한 기능 제공
- 직관적인 인터페이스로 테스트 코드를 읽기 쉽게 작성 가능
- 다양한 모킹 시나리오 지원
- 스텁(stubbing)과 검증(verification)을 명확하게 분리하여 테스트 가독성 향상
- 불필요한 기대치 설정을 최소화하는 철학
- 테스트 주도 개발(TDD)과 잘 어울리는 설계

## 기본 사용법

```
import static org.mockito.Mockito.*;  
  
// Mock 객체 생성  
UserRepository mockRepository = mock(UserRepository.class);  
  
// 스텁 설정  
when(mockRepository.findById(1L))  
    .thenReturn(new User(1L, "사용자"));  
  
// 테스트 대상 객체에 Mock 주입  
UserService userService = new UserService(mockRepository);  
  
// 테스트 실행  
User user = userService.getUserById(1L);  
  
// 검증  
verify(mockRepository).findById(1L);  
assertEquals("사용자", user.getName());
```

# 테스트 더블 실전 사용 사례



## 데이터베이스 테스트

- 실제 데이터베이스 대신 인메모리 데이터베이스(Fake) 활용
- Repository 인터페이스의 Mock 구현으로 대체
- 테스트 속도 향상 및 외부 의존성 제거 가능



## 외부 API 테스트

- 외부 API 클라이언트를 Mock으로 대체
- 다양한 응답 시나리오 시뮬레이션 (성공, 실패, 타임아웃)
- 외부 서비스 의존성 없는 안정적 테스트 환경 구축



## 시간 의존적 코드 테스트

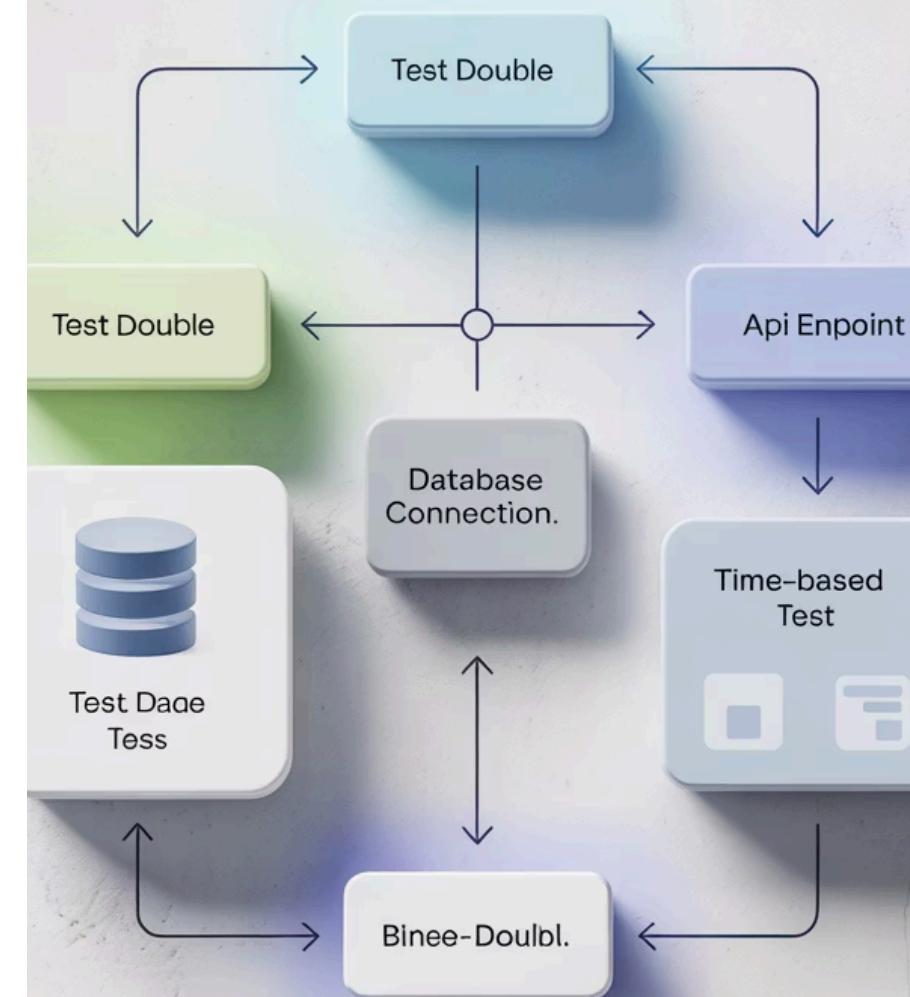
- 현재 시간 제공 서비스를 Mock으로 대체
- 특정 시점의 동작 테스트 가능
- 날짜/시간 기반 로직, 스케줄링, 캐시 만료 테스트에 유용



## 비결정적 동작 테스트

- 난수 생성기, UUID 생성기 등을 Mock으로 대체
- 예측 가능한 테스트 환경 구성
- 재현 가능한 테스트 작성 용이

# Testing Scenarios with Test Doubles



# 테스트 더블 사용 시 주의사항

## 과도한 Mock 사용 지양

- 모든 의존성을 Mock으로 대체 시 실제 시스템 동작 반영 불가능
- 꼭 필요한 경우에만 테스트 더블 사용, 가능하면 실제 객체 활용 권장
- 과도한 Mock 사용 시 테스트 코드 복잡성 증가 및 유지보수 어려움
- 실제 통합 문제 발견 어려움 발생 가능성

## 구현 세부사항에 대한 과도한 결합 X

- Mock은 테스트를 구현 세부사항에 밀접하게 결합시킴
- 코드 리팩토링 시 테스트가 쉽게 깨지는 원인이 될 수 있음
- 공개 API와 최종 결과를 검증하는 방식으로 테스트 작성 권장
- "무엇을" 테스트하는지가 아닌 "어떻게" 구현되는지에 초점 맞추는 경향은 테스트의 가치 떨어뜨릴 수 있음

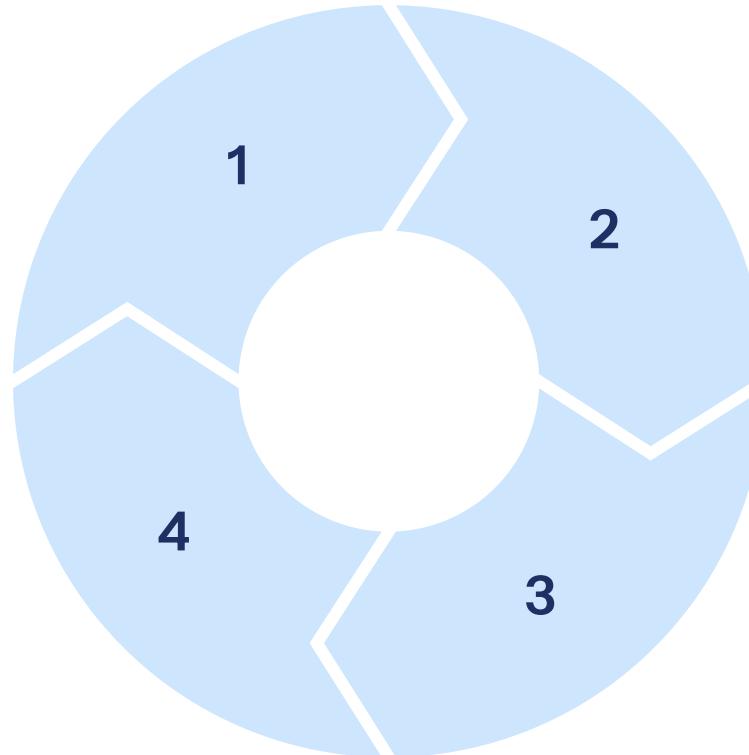
## Mock과 실제 구현의 동기화 문제

- Mock 객체가 실제 구현체 동작을 정확히 반영하지 않으면 테스트는 통과해도 실제 환경에서 문제 발생 가능
- Mock의 동작이 실제 구현체와 일치하도록 주의 필요
- 인터페이스 변경 시 Mock도 함께 업데이트 필요
- 동기화 미흡 시 테스트 신뢰성 저하

# TDD(Test-Driven Development) 개요

## 실패하는 테스트 작성 (Red)

- 기능 구현 전 테스트 먼저 작성
- 실패하는 테스트로 요구사항 명확화
- 개발 목표 구체화



## 테스트를 통과하는 코드 작성 (Green)

- 가장 단순한 방법으로 테스트 통과
- 기능 동작에만 집중
- 최소 구현 원칙 적용

## 코드 리팩토링 (Refactor)

- 테스트 통과 유지하며 코드 개선
- 중복 제거 및 가독성 향상
- 구조 최적화

TDD는 테스트 중심의 개발 방법론으로, 코드 작성 전 테스트를 먼저 작성함으로써 요구사항을 명확히 하고, 설계 개선과 높은 테스트 커버리지를 달성하여 버그 감소와 유지보수성을 향상

# BDD(Behavior-Driven Development) 개요

## BDD의 핵심 개념

- BDD는 TDD에서 발전한 방법론으로, 비즈니스 요구사항과 기술적 구현 사이의 간극을 줄이는 데 중점
- 시스템의 행동(behavior)을 명세하고 테스트하는 방식
- 자연어에 가까운 형태로 테스트를 작성
- 개발자, QA, 비즈니스 이해관계자 간의 협업 강화
- 모든 이해관계자가 이해할 수 있는 공통 언어 사용으로 의사소통 개선
- "Given-When-Then" 형식의 명세가 특징적

## Gherkin 문법 예시

Feature: 사용자 로그인

사용자가 시스템에 인증할 수 있어야 합니다.

Scenario: 유효한 자격 증명으로 로그인

Given 사용자가 로그인 페이지에 있음

When 유효한 이메일과 비밀번호를 입력함

And 로그인 버튼을 클릭함

Then 사용자는 대시보드 페이지로 리디렉션됨

And 환영 메시지가 표시됨

Scenario: 잘못된 비밀번호로 로그인

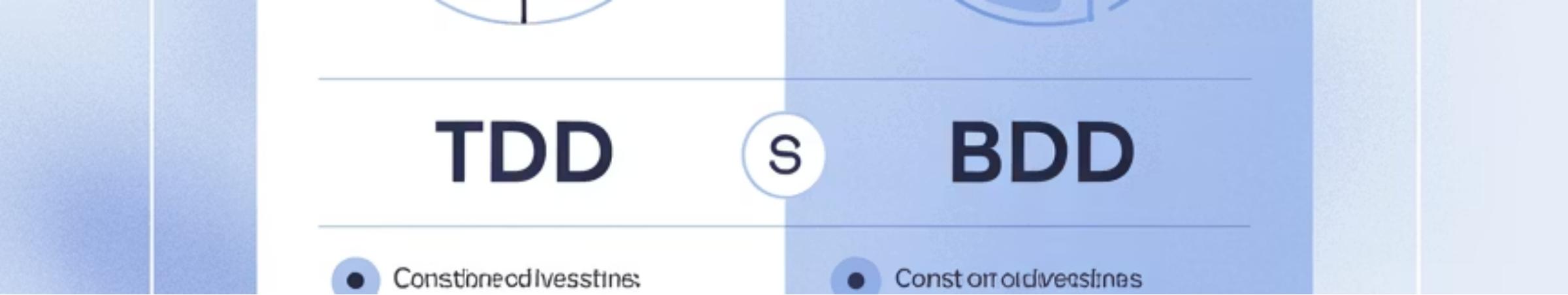
Given 사용자가 로그인 페이지에 있음

When 유효한 이메일과 잘못된 비밀번호를 입력함

And 로그인 버튼을 클릭함

Then 오류 메시지가 표시됨

And 사용자는 로그인 페이지에 남아 있음



**TDD**

**S**

**BDD**

- Constructional test cases

- Constructional specifications

## TDD vs BDD 비교

특성

TDD (테스트 주도 개발)

BDD (행동 주도 개발)

초점	TDD (테스트 주도 개발)	BDD (행동 주도 개발)
작성 주체	주로 개발자	개발자, QA, 비즈니스 분석가 공동
문법	프로그래밍 언어의 테스트 프레임워크	Gherkin 등 자연어에 가까운 DSL
테스트 수준	주로 단위 테스트	단위, 통합, 인수 테스트 모두 가능
커뮤니케이션	개발 팀 내부 중심	이해관계자 간 협업 강화
목표	올바른 코드 구현	올바른 제품 구축
대표 도구	JUnit, PyTest, NUnit	Cucumber, SpecFlow, JBehave

- TDD와 BDD는 상호 배타적이지 않으며, 함께 사용될 수 있음
- BDD는 TDD의 확장으로 볼 수 있으며, 프로젝트의 특성과 팀의 구성에 따라 적절한 방법론을 선택하거나 두 방법론의 장점의 결합 추천

# TDD와 BDD의 실제 적용



## 요구사항 분석 단계

- BDD 접근법으로 사용자 스토리와 시나리오 작성
- 비즈니스 이해관계자와 개발 팀의 공동 참여
- Gherkin 형식의 명세 작성
- 모든 참여자 간 공통된 이해 형성

## 개발 단계

- TDD 방식으로 단위 테스트 작성 및 기능 구현
- Red-Green-Refactor 사이클 적용
- BDD 시나리오를 충족시키는 단위 테스트 작성
- Mock 객체 등 테스트 더블 활용

## 검증 단계

- BDD 시나리오를 자동화된 인수 테스트로 구현
- 전체 시스템의 요구사항 충족 여부 검증
- Cucumber 등 도구로 시나리오를 실행 가능한 테스트로 변환

## 피드백 및 개선

- 테스트 결과를 바탕으로 이해관계자에게 피드백 제공
- 필요시 요구사항과 구현 조정
- 과정 반복을 통한 제품의 점진적 개선

# Python에서의 BDD 도구: Behave

## Behave 소개

- Behave는 Python에서 BDD를 구현하기 위한 인기 있는 프레임워크
- Gherkin 문법으로 작성된 기능 명세를 실행 가능한 테스트로 변환해 줌
- 자연어로 작성된 시나리오와 Python 코드를 연결하여 테스트를 자동화

## 구조:

- feature** 파일: 시나리오를 작성
- steps** 디렉토리: 해당 단계를 구현하는 Python 코드 작성
- 비기술적 이해관계자도 테스트 명세를 이해하고 참여 가능

## Behave 사용 예시

```
# features/login.feature
Feature: 사용자 로그인
  Scenario: 유효한 자격 증명으로 로그인
    Given 사용자가 로그인 페이지에 있음
    When 유효한 이메일 "user@example.com"과 비밀번호 "password"를 입력함
    And 로그인 버튼을 클릭함
    Then 사용자는 대시보드 페이지로 리디렉션됨

# features/steps/login_steps.py
from behave import given, when, then

@given('사용자가 로그인 페이지에 있음')
def step_impl(context):
    context.browser.get('http://example.com/login')

@when('유효한 이메일 "{email}"과 비밀번호 "{password}"를 입력함')
def step_impl(context, email, password):
    context.browser.find_element_by_id('email').send_keys(email)
    context.browser.find_element_by_id('password').send_keys(password)
```

# Java에서의 BDD 도구: Cucumber

## Cucumber 소개

- Cucumber는 Java를 포함한 다양한 언어에서 BDD를 지원하는 가장 인기 있는 도구
- Gherkin 문법으로 작성된 `feature` 파일과 이를 구현하는 `step definition` 코드로 구성
- JUnit과 통합되어 기존 Java 테스트 환경에서 쉽게 사용 가능
- 다국어 지원, 풍부한 보고서 생성, 다양한 IDE 플러그인 등의 기능 제공
- 비즈니스 이해관계자와 개발자 간의 협업 강화
- 자연어로 작성된 명세를 실행 가능한 테스트로 변환

## Cucumber 사용 예시

```
# src/test/resources/features/login.feature
Feature: 사용자 로그인
  Scenario: 유효한 자격 증명으로 로그인
    Given 사용자가 로그인 페이지에 있음
    When 유효한 이메일 "user@example.com"과 비밀번호 "password"를 입력함
    And 로그인 버튼을 클릭함
    Then 사용자는 대시보드 페이지로 리디렉션됨
```

```
# src/test/java/stepdefs/LoginStepDefs.java
public class LoginStepDefs {
  @Given("사용자가 로그인 페이지에 있음")
  public void userOnLoginPage() {
    driver.get("http://example.com/login");
  }

  @When("유효한 이메일 {string}과 비밀번호 {string}를 입력함")
  public void enterCredentials(String email, String password) {
    driver.findElement(By.id("email")).sendKeys(email);
    driver.findElement(By.id("password")).sendKeys(password);
  }
}
```

# 통합 테스트 개요

## 통합 테스트의 정의

- 개별 단위가 함께 작동할 때 올바르게 동작하는지 검증하는 테스트 유형
- 단위 테스트와 달리 모듈 간의 상호작용과 인터페이스를 검증
- 데이터베이스, 파일 시스템, 네트워크 등 실제 환경의 의존성을 포함하여 테스트
- 모듈 간 통합 지점에서 발생할 수 있는 문제를 발견하는데 중점

## 통합 테스트의 범위

- 두 개의 클래스 간 상호작용부터 여러 컴포넌트, 서비스, 외부 시스템과의 통합까지 포함 가능
- 일반적으로 단위 테스트보다 큰 범위를 다루지만, 시스템 테스트보다는 작은 범위를 대상
- 마이크로서비스 아키텍처에서는 서비스 간 통합, 레거시 시스템과의 인터페이스, API 통합 등이 중요 대상

## 통합 테스트의 중요성

- 단위 테스트만으로는 발견할 수 없는 여러 종류의 문제를 찾아낼 수 있음
- 구성 오류, 환경 설정 문제, 인터페이스 불일치, 동시성 이슈 등을 효과적으로 검증 가능
- 현대적인 분산 시스템과 마이크로서비스 환경에서는 통합 테스트의 중요성이 더욱 증가
- 시스템의 안정성과 신뢰성을 보장하는데 필수적인 요소

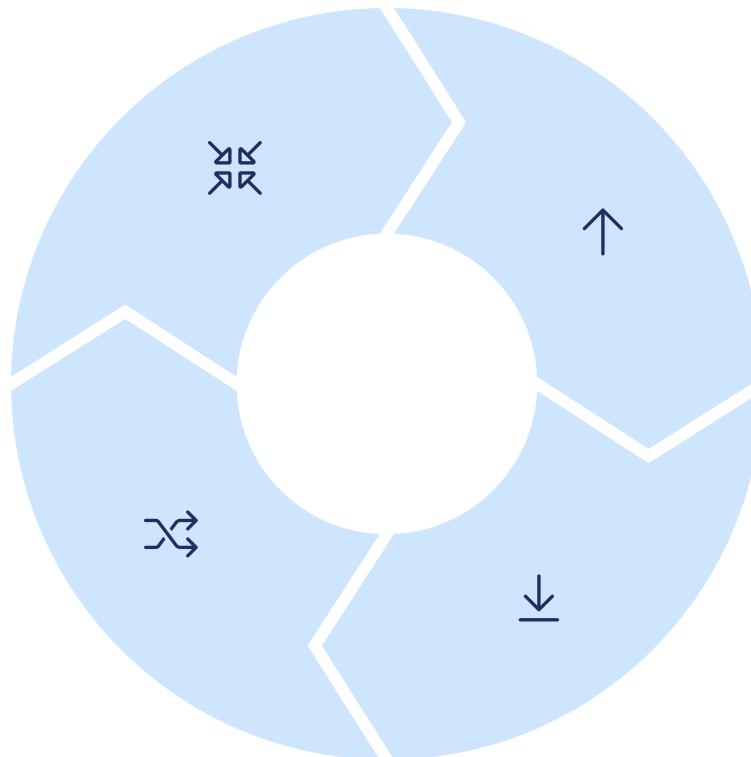
# 통합 테스트 전략

## 빅뱅 통합 (Big Bang Integration)

- 모든 컴포넌트를 한꺼번에 통합하고 테스트하는 방식
- 구현이 간단함
- 오류 발생 시 원인 파악이 어려움
- 통합 과정이 복잡해질 수 있음

## 샌드위치 통합 (Sandwich Integration)

- 상향식과 하향식 접근법을 결합한 방식
- 중간 레이어를 중심으로 양방향으로 통합을 진행
- 복잡한 시스템에 적합함
- 더 효율적인 테스트가 가능



## 상향식 통합 (Bottom-Up Integration)

- 하위 레벨 컴포넌트부터 시작하여 상위 레벨로 점진적으로 통합
- 드라이버(Driver)를 사용하여 아직 개발되지 않은 상위 모듈을 시뮬레이션

## 하향식 통합 (Top-Down Integration)

- 상위 레벨 컴포넌트부터 시작하여 하위 레벨로 점진적으로 통합
- 스텁(Stub)을 사용하여 아직 개발되지 않은 하위 모듈을 시뮬레이션

프로젝트의 특성, 아키텍처, 팀 구성 등을 고려하여 적절한 통합 전략 선택 필요

# 통합 테스트 구현 도구



## Spring Test

- Spring 애플리케이션의 통합 테스트를 위한 도구
- 애플리케이션 컨텍스트 관리, 트랜잭션 지원, 모의 객체 통합 등의 기능 제공
  - @SpringBootTest로 전체 애플리케이션 컨텍스트 로드
  - @WebMvcTest로 웹 레이어 테스트
  - @DataJpaTest로 JPA 컴포넌트 테스트



## TestContainers

- Docker 컨테이너를 활용한 통합 테스트 라이브러리
- 데이터베이스, 메시지 큐 등의 실제 인프라 의존성을 테스트 환경에서 쉽게 구성 가능
  - 다양한 데이터베이스 지원 (MySQL, PostgreSQL 등)
  - 메시지 큐, 웹 서버 등 다양한 서비스 지원
  - Java, Python 등 다양한 언어 지원



## REST Assured

- RESTful API의 통합 테스트를 위한 Java 라이브러리
- 가독성 높은 DSL을 통해 HTTP 요청 작성 및 응답 검증 용이
  - BDD 스타일의 자연스러운 문법
  - JSON, XML 응답 검증 기능
  - 다양한 인증 방식 지원

Integration Testing Solutions

Start free Trial

**Test**

Integrating a variety of test cases, each consisting of a connection to a database and a connection to a service. It's a complex environment.

Learn more

**TestContainers**

Using Docker containers for integration testing. Provides support for various databases and message queues.

Learn more

**Rest Assured**

Testing RESTful APIs using Java. Supports BDD-style assertions and JSON/XML response validation.

Learn more

# 통합 테스트 실행 팁



## 테스트 데이터 관리

- 테스트 실행 전후로 데이터베이스 상태 일관성 유지
- 각 테스트의 독립적 실행을 위한 데이터 설정/정리 자동화
- 인메모리 데이터베이스나 TestContainers로 격리된 환경 구성



## 테스트 실행 시간 최적화

- 단위 테스트보다 긴 실행 시간 고려
- 공유 가능한 설정(애플리케이션 컨텍스트 등) 재사용
- 필요한 컴포넌트만 로드하여 시간 단축
- 병렬 실행을 통한 전체 테스트 시간 감소



## 외부 의존성 관리

- 테스트용 더블 활용
- 실제 외부 서비스 연결 시 발생하는 불안정성 방지
- WireMock, MockServer 등으로 외부 API 시뮬레이션



## CI/CD 파이프라인 통합

- CI 파이프라인에 통합 테스트 포함
- 코드 변경 시마다 자동 실행 구성
- 통합 관련 문제 조기 발견 및 수정
- 테스트 결과 시각화 및 명확한 피드백 제공

# 시스템 테스트 개요



## 시스템 테스트의 정의

- 완전히 통합된 전체 소프트웨어 시스템을 검증하는 테스트 단계
- 명세된 요구사항 충족 여부 확인
- 종단간(end-to-end) 관점에서 시스템 동작 평가



## 시스템 테스트의 목적

- 사용자 관점에서의 시스템 동작 검증
- 모든 구성 요소의 통합적 동작 확인
- 기능적 요구사항 및 비기능적 요구사항(성능, 보안, 사용성 등) 검증



## 시스템 테스트의 관점

- 사용자 관점에서 수행
- 실제 사용 환경과 유사한 환경에서 진행
- 주로 별도의 테스트 팀에 의해 수행
- 블랙박스 테스팅 접근법 주로 활용



## 테스트 레벨에서의 위치

- 단위 테스트와 통합 테스트 이후 수행
- 인수 테스트 이전 단계
- V-모델에서 시스템 설계 단계에 대응하는 검증 활동

# 시스템 테스트 유형

## 기능 테스트(Functional Testing)

- 시스템의 기능적 요구사항 검증
- 각 기능이 명세대로 올바르게 동작하는지 확인
- 정상 흐름, 예외 상황, 경계 조건 포함
- 메뉴 기능, 데이터 처리, 검색, 비즈니스 규칙 검증
- 입출력 검증, 오류 처리 기능 확인
- 사용자 시나리오 기반 테스트 케이스 작성

## 비기능 테스트(Non-functional Testing)

- 시스템의 비기능적 속성 검증
- 품질 특성과 관련된 요구사항 평가
- 사용자 경험과 시스템 품질에 영향을 미치는 요소 검증
- 성능 테스트: 응답 시간, 처리량, 자원 사용률 측정
- 부하 테스트: 높은 작업량에서의 시스템 동작 평가
- 보안 테스트: 취약점 및 보안 요구사항 검증
- 사용성 테스트: 사용자 친화적 인터페이스 확인
- 호환성 테스트: 다양한 환경에서의 작동 확인

## 회귀 테스트(Regression Testing)

- 시스템 변경 후 기존 기능 정상 작동 확인
- 코드 수정, 기능 추가, 버그 수정 등으로 인한 부작용 발견
- 자동화된 테스트 스위트를 통한 효율적 검증
- 변경 영향 분석을 통한 테스트 범위 결정
- CI/CD 파이프라인에 통합하여 지속적 검증

# 시스템 테스트 자동화 도구

## 웹 애플리케이션 테스트 도구

- **개요:** 웹 기반 시스템의 종단간 테스트를 위한 도구들
- **특징:** 실제 브라우저를 제어하여 사용자 상호작용을 시뮬레이션
- **Selenium WebDriver:** 다양한 브라우저에서 웹 애플리케이션 테스트를 자동화 하는 가장 인기 있는 도구
- **Cypress:** 웹 애플리케이션을 위한 종단 간 테스트 프레임워크로, 개발자 친화적 인 디버깅 기능 제공
- **Playwright:** Microsoft에서 개발한 도구로, 다양한 브라우저 지원 및 강력한 자동화 기능 제공

## 모바일 애플리케이션 테스트 도구

- **개요:** 모바일 앱의 시스템 테스트를 위한 도구들
- **특징:** 다양한 기기와 OS 환경에서의 테스트를 지원
- **Appium:** iOS, Android 앱의 자동화 테스트를 위한 오픈소스 프레임워크
- **Espresso(Android):** Android 앱의 UI 테스트를 위한 Google의 테스트 프레임워크
- **XCTest(iOS):** iOS 앱의 UI 테스트를 위한 Apple의 테스트 프레임워크

## API 및 성능 테스트 도구

- **개요:** 백엔드 시스템과 API의 테스트, 그리고 시스템의 성능 평가를 위한 도구들
- **Postman:** API 테스트 및 문서화를 위한 도구
- **JMeter:** 부하 테스트 및 성능 측정을 위한 Apache 프로젝트
- **Gatling:** 고성능 부하 테스트 도구로, 실시간 모니터링 기능 제공
- **Locust:** Python 기반의 분산 부하 테스트 도구

# 시스템 테스트 접근 방법



## 테스트 시나리오 중심 접근법

- 사용자 관점에서의 실제 사용 시나리오 기반 테스트 케이스 설계
- 비즈니스 프로세스 흐름을 따라 시스템의 전체 기능 검증
- 사용자 스토리나 유스케이스를 테스트 시나리오로 변환



## 기능 매트릭스 기반 접근법

- 시스템의 모든 기능을 식별하고 매트릭스 형태로 정리
- 체계적인 테스트 수행 방법론 적용
- 각 기능에 대해 다양한 입력 조건, 경계값, 예외 상황 고려



## 탐색적 테스트 접근법

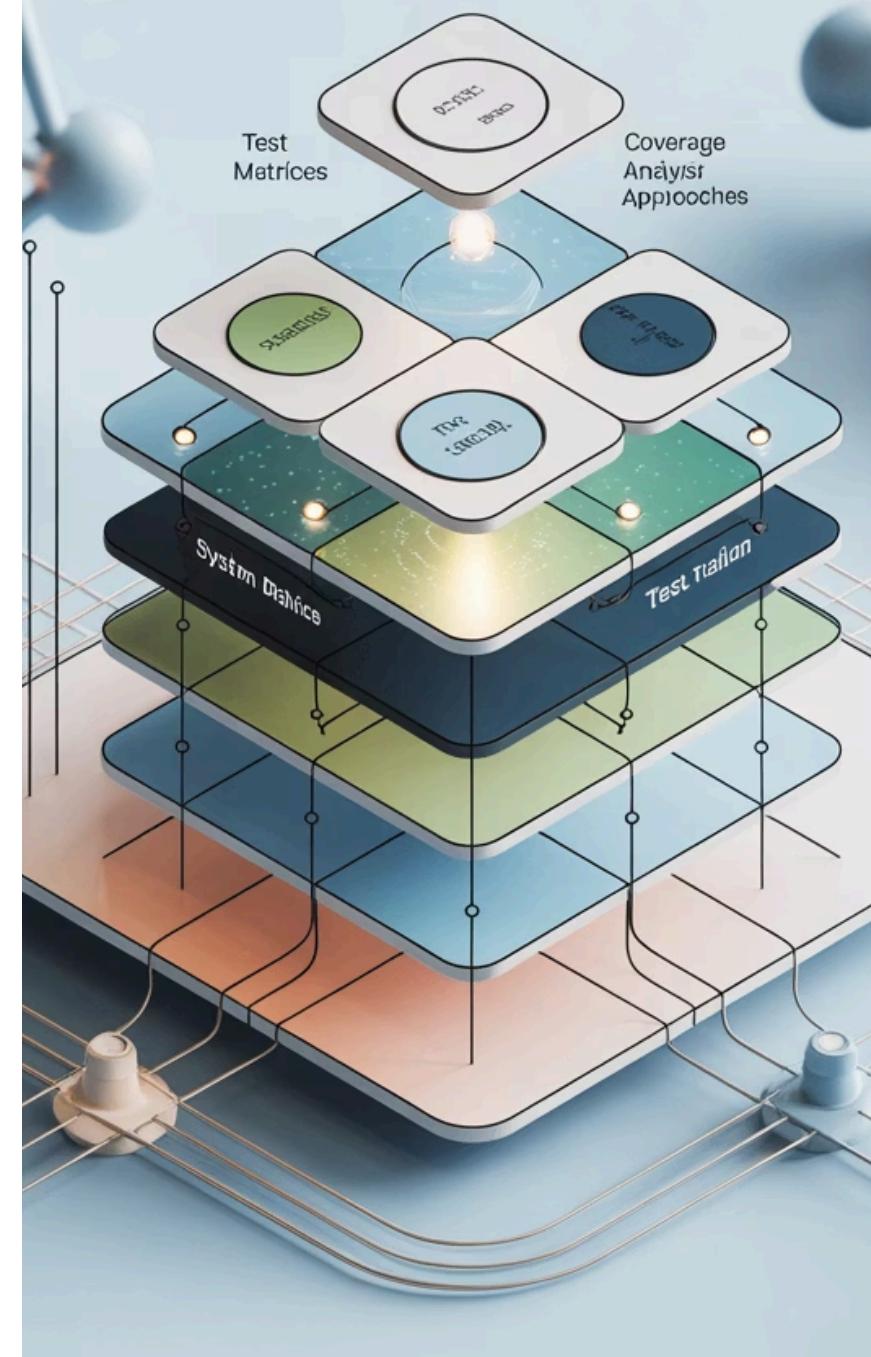
- 미리 정의된 테스트 케이스에 의존하지 않는 방식
- 테스터의 지식과 경험을 바탕으로 자유로운 시스템 탐색
- 예상치 못한 문제 발견에 효과적
- 체계적인 테스트를 보완하는 역할



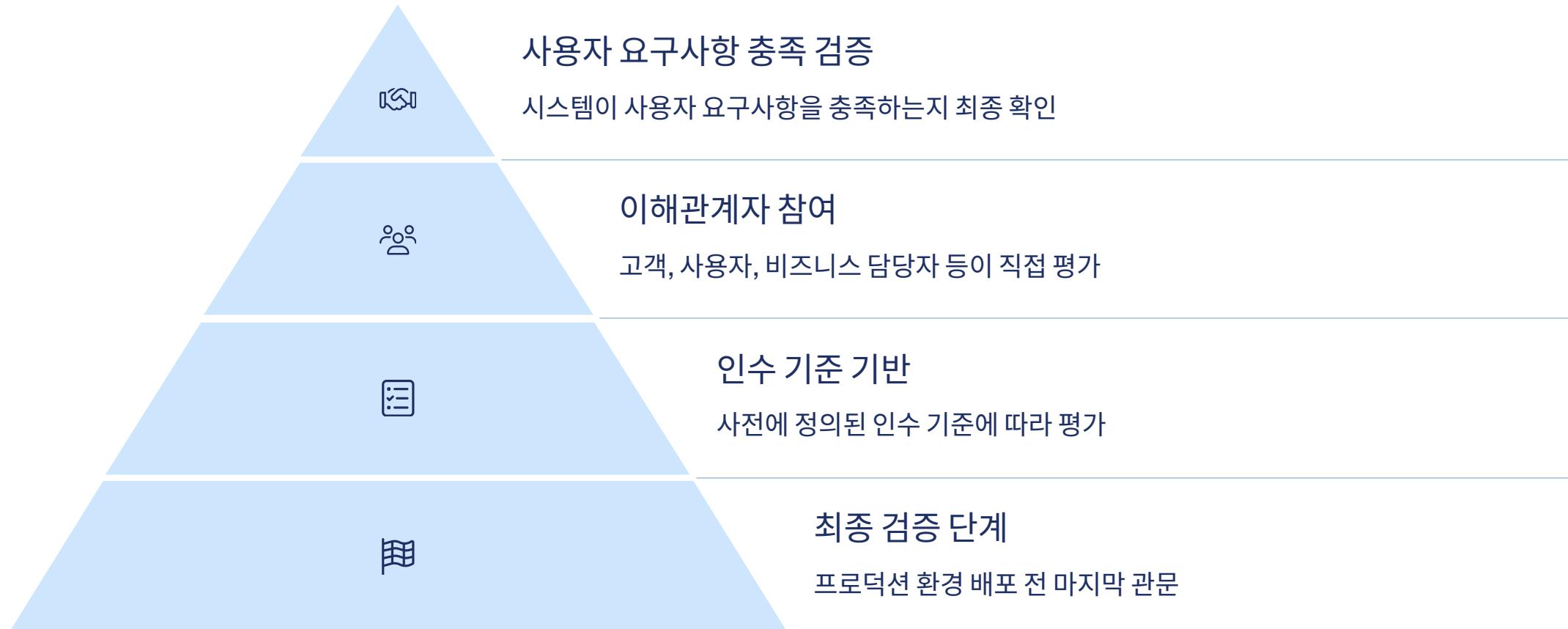
## 리스크 기반 테스트 접근법

- 시스템의 리스크가 높은 영역에 테스트 노력 집중
- 비즈니스 중요도, 실패 가능성, 사용 빈도 등 고려
- 테스트 우선순위 결정 기준 적용
- 제한된 자원으로 효과적인 테스트 수행

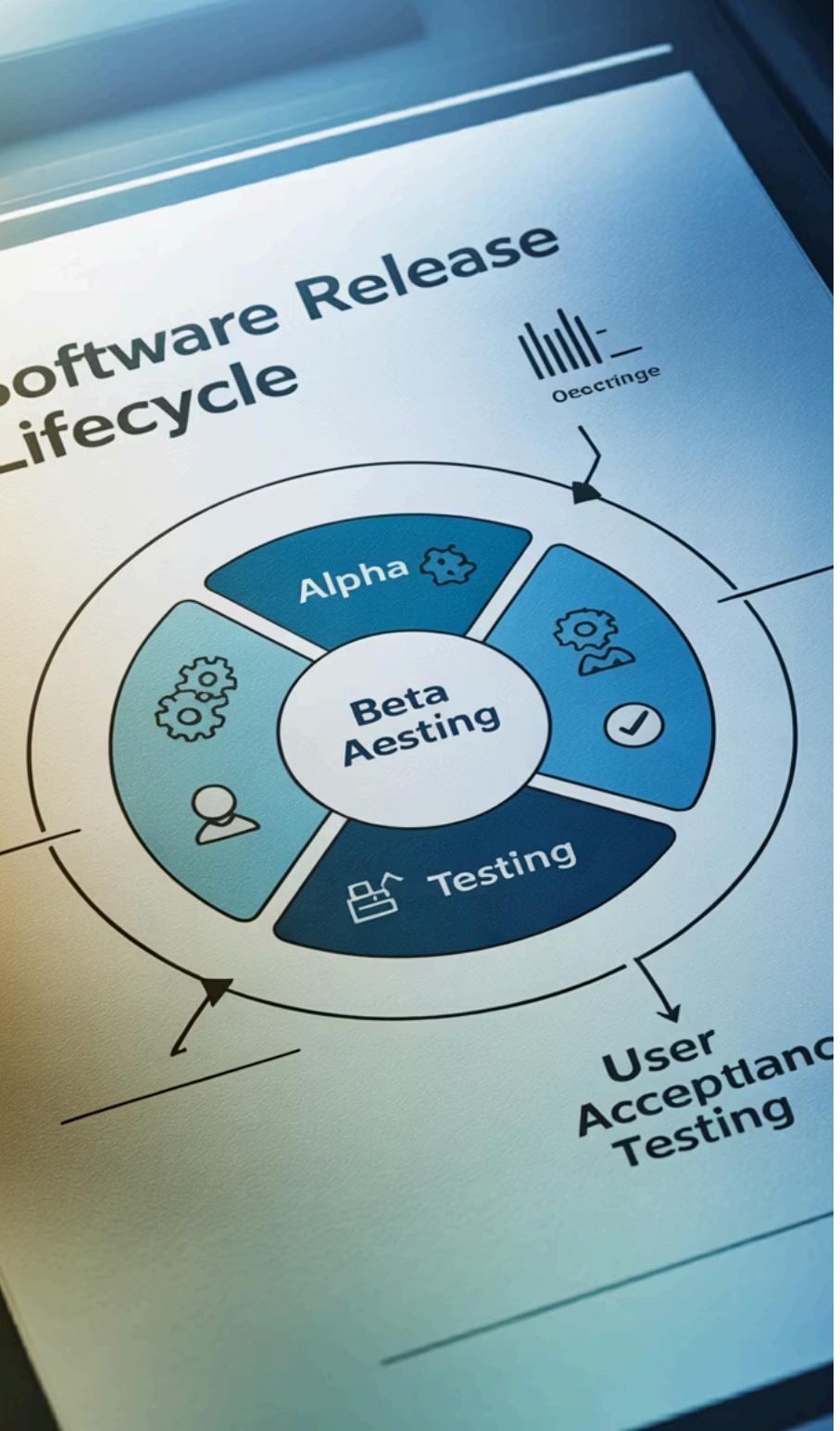
# System Testing Approaches



# 인수 테스트 개요



- 인수 테스트(Acceptance Testing)는 소프트웨어 개발 생명주기의 마지막 테스트 단계
- 개발된 시스템이 사용자의 요구사항과 비즈니스 목표를 충족하는지 검증하는 과정
- 실제 사용자나 고객이 참여하여 진행됨
- 시스템이 의도한 목적을 달성하고 사용자에게 가치를 제공하는지 평가
- 소프트웨어의 공식적인 인수와 최종 지불의 기준이 됨
- 프로덕션 환경으로의 배포 여부를 결정하는 중요한 단계
- 비즈니스 관점에서 시스템의 적합성을 평가하는 데 중점



# 인수 테스트 유형

## 사용자 인수 테스트(User Acceptance Testing, UAT)

- 실제 최종 사용자가 참여하여 비즈니스 요구사항 충족 여부 검증
- 실제 업무 환경과 유사한 조건에서 수행
- 사용자 관점에서 시스템 적합성 평가

## 비즈니스 인수 테스트 (Business Acceptance Testing)

- 비즈니스 분석가, 제품 책임자 등이 참여
- 비즈니스 목표와 프로세스 지원 여부 검증
- ROI, 비즈니스 규칙 준수, 워크플로우 지원 관점 평가

## 알파 테스트(Alpha Testing)

- 개발 조직 내부에서 수행하는 인수 테스트
- 개발 환경이나 유사한 환경에서 진행
- 개발팀이 아닌 내부 테스터나 QA 팀이 주로 수행

## 베타 테스트(Beta Testing)

- 제한된 외부 사용자 그룹에게 시스템 공개
- 실제 환경에서 테스트 진행
- 다양한 사용 환경과 사용자 행동 패턴에서 시스템 동작 검증

# BDD와 인수 테스트

## BDD와 인수 테스트의 연결성

- 행동 주도 개발(BDD)은 인수 테스트와 자연스럽게 연결됨
- BDD의 Given-When-Then 형식 시나리오는 사용자 관점에서 시스템의 기대 동작을 명세
- 인수 기준을 명확하게 정의하는 방법으로 활용
- 비즈니스 이해관계자와 개발팀 간의 공통 언어로 작용
- 요구사항에 대한 이해를 일치시키고 인수 테스트의 기준을 명확히 함
- 자동화된 인수 테스트로 구현되어 지속적인 검증 가능

## BDD 기반 인수 테스트 구현

```
# features/checkout.feature
```

```
Feature: 상품 결제
```

고객으로서 상품을 결제하고 주문을 완료할 수 있어야 합니다.

```
Scenario: 유효한 카드로 결제
```

```
Given 장바구니에 상품이 추가되어 있음
```

```
And 배송 정보가 입력되어 있음
```

```
When 유효한 신용카드 정보를 입력함
```

```
And 결제 버튼을 클릭함
```

```
Then 결제가 성공적으로 처리됨
```

```
And 주문 확인 페이지로 이동함
```

```
And 주문 확인 이메일이 발송됨
```

```
Scenario: 잔액 부족으로 결제 실패
```

```
Given 장바구니에 상품이 추가되어 있음
```

```
And 배송 정보가 입력되어 있음
```

```
When 잔액이 부족한 카드 정보를 입력함
```

```
And 결제 버튼을 클릭함
```

```
Then 결제 실패 메시지가 표시됨
```

```
And 사용자는 결제 페이지에 머무름
```

# 인수 테스트 자동화

## 인수 테스트 자동화의 이점

- 반복적인 테스트 수행 시간 단축
- CI/CD 파이프라인 통합으로 빠른 피드백 제공
- 회귀 테스트의 효율적 수행
- 새로운 변경사항이 기존 기능에 미치는 영향 빠르게 확인

## 자동화 도구 선택

- BDD 기반: Cucumber, SpecFlow, Behave
- UI 자동화: Selenium WebDriver, Appium
- 종합 프레임워크: Robot Framework, Cypress

## 자동화 범위 결정

- 핵심 비즈니스 기능에 자동화 노력 집중
- 자주 사용되는 기능 우선 자동화
- 리스크가 높은 영역 집중 테스트
- UI 변경이 잦은 부분은 하위 레벨에서 테스트

## 지속적 통합 연계

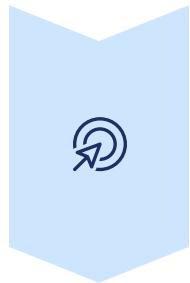
- CI/CD 파이프라인에 자동화 테스트 통합
- 코드 변경 시마다 자동 실행 구성
- 인수 기준 충족 여부 지속적 검증
- 테스트 결과 시각화 및 리포팅으로 투명한 공유

# 테스트 계획과 전략의 차이

구분	테스트 전략(Test Strategy)	테스트 계획(Test Plan)
정의	조직이나 프로젝트에서 테스트를 수행하는 일반적인 접근 방식을 설명하는 상위 수준 문서	특정 프로젝트나 제품의 테스트 활동을 상세히 계획한 문서
범위	여러 프로젝트에 적용되는 조직 수준의 지침	특정 프로젝트나 릴리스에 국한된 계획
작성 시점	조직 수준에서 수립되며, 장기간 사용됨	프로젝트 초기에 작성되고 프로젝트 진행에 따라 업데이트됨
내용	테스트 접근 방법론, 테스트 수준, 테스트 유형, 도구, 환경 등에 대한 일반적인 지침	테스트 일정, 리소스, 테스트 항목, 테스트 케이스, 위험 관리 등 구체적인 계획
목적	테스트 활동의 일관성과 표준화 제공	특정 프로젝트의 테스트 활동 관리 및 조정
책임자	테스트 관리자, QA 리더 등 조직 수준의 책임자	프로젝트 테스트 리더, QA 담당자

테스트 전략은 '무엇을 어떻게 테스트할 것인가'에 대한 전반적인 접근 방식을, 테스트 계획은 '언제, 어디서, 누가, 얼마나' 테스트할 것인지에 대한 구체적인 계획을 다룸

# 효과적인 테스트 전략 수립



## 테스트 목표 정의

- 소프트웨어 품질 목표와 테스트 달성 목표 명확화
- 결함 감소, 사용자 만족도 향상, 성능 개선 등 측정 가능한 목표 설정



## 테스트 레벨 결정

- 단위, 통합, 시스템, 인수 테스트 레벨 선택
- 각 레벨의 범위, 목적, 책임자 명확화
- 테스트 레벨 간 연계 방안 수립



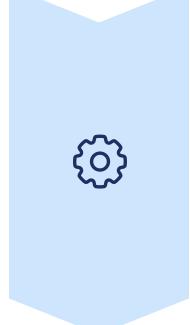
## 테스트 유형 선정

- 기능, 성능, 보안, 사용성 테스트 등 선택
- 비즈니스 요구사항과 리스크 기반 우선순위 결정



## 리스크 기반 접근법 적용

- 프로젝트 리스크 식별 및 평가
- 높은 비즈니스 영향도와 결함 가능성 영역에 리소스 집중



## 테스트 도구 및 환경 계획

- 테스트 자동화, 관리, 결함 추적 도구 선정
- 테스트 환경 구성 방안 수립
- 도구 간 통합과 환경 안정성 확보

# 테스트 전략 템플릿

## 1. 소개

테스트 전략의 목적, 범위, 참조 문서 등을 기술

테스트 전략이 적용되는 프로젝트나 조직에 대한 배경 정보를 제공

- 테스트 전략의 목적
- 적용 범위 및 대상
- 관련 문서 및 참조

## 2. 테스트 접근 방법

전반적인 테스트 접근 방법과 테스트 레벨, 유형 등을 정의  
리스크 기반, 요구사항 기반 등 테스트의 기본 철학을 설명

- 테스트 레벨 (단위, 통합, 시스템, 인수)
- 테스트 유형 (기능, 성능, 보안 등)
- 테스트 우선순위 결정 방법

## 3. 테스트 환경 및 도구

테스트에 필요한 환경 구성과 사용할 도구를 명시

테스트 환경 관리 방안과 도구 선정 기준도 포함

- 테스트 환경 구성 요소
- 테스트 도구 목록 및 용도
- 환경 구성 및 관리 책임

## 4. 리스크 및 완화 방안

테스트 과정에서 발생할 수 있는 리스크를 식별하고, 이에 대한 완화 방안을 수립

일정, 자원, 기술적 제약 등의 리스크를 고려

- 주요 테스트 리스크 목록
- 리스크별 완화 전략
- 리스크 모니터링 방안

# 테스트 계획 작성 가이드

## 1. 테스트 계획 식별

- 테스트 계획의 고유 식별자, 버전 정보, 관련 문서 등 기본 정보 명시
- 테스트 계획의 변경 이력 관리를 위한 정보 포함

## 2. 소개

- 테스트 계획의 목적, 범위, 테스트 대상 시스템에 대한 개요 제공
- 테스트 수행 배경과 전체적인 접근 방법 설명

## 3. 테스트 항목

- 테스트할 소프트웨어 컴포넌트, 기능, 특성 등 명확히 정의
- 테스트 범위에 포함되는 항목과 제외되는 항목 구분하여 기술

## 4. 테스트 기능

- 수행할 테스트의 종류와 각 테스트에서 검증할 기능 설명
- 기능 테스트, 성능 테스트, 보안 테스트 등 테스트 유형별 구체적인 검증 대상 명시

## 5. 일정 및 자원

- 테스트 활동의 일정, 필요한 인적/물적 자원, 환경 요구사항 등 계획
- 마일스톤과 의존성 명확화 및 자원 할당 계획 수립

## 6. 위험 및 대책

- 테스트 과정에서 발생할 수 있는 위험 요소 식별
- 위험에 대한 대응 방안 마련
- 프로젝트 특성에 맞는 구체적인 위험과 완화 전략 포함

# 테스트 활동과 역할 분담



## 테스트 관리자

테스트 전략 수립, 테스트 계획 작성, 리소스 관리, 일정 조정, 이해관계자 커뮤니케이션 등 테스트 프로세스 전반을 관리

테스트 진행 상황을 모니터링하고, 리스크를 식별하여 관리



## 테스트 리더/엔지니어

테스트 케이스 설계, 테스트 환경 구성, 테스트 실행 및 결과 분석, 결함 보고 등 실질적인 테스트 활동을 수행

자동화 테스트 스크립트 개발과 유지보수 담당



## QA 전문가

품질 보증 프로세스 수립, 품질 메트릭 정의 및 측정, 프로세스 개선 활동 등을 담당

테스트 전반의 품질을 모니터링하고, 품질 기준 충족 여부를 평가



## 개발자

단위 테스트 작성 및 실행, 코드 리뷰, 결함 수정 등을 담당

테스트 가능한 코드 설계와 구현, 테스트 환경 구성 지원 등의 역할도 수행



## 비즈니스 분석가

요구사항 명세, 인수 기준 정의, 비즈니스 관점의 테스트 케이스 검토 등을 담당

테스트 결과의 비즈니스 영향을 평가하고, 이해관계자와의 소통을 지원

# 테스트 문서화: 종류와 목적

## 테스트 정책 (Test Policy)

- 조직 수준의 테스팅 철학과 접근 방식을 정의하는 고수준 문서
- 품질에 대한 조직의 약속, 테스트의 일반적인 목표와 원칙, 역할과 책임 등을 명시
- 경영진이 승인하고 장기간 유지되는 문서
- 조직의 모든 테스트 활동의 기초가 됨
- 프로젝트 간 일관성을 제공하고 테스트 문화를 형성

## 테스트 전략 (Test Strategy)

- 특정 프로젝트나 제품군에 대한 테스트 접근 방식을 설명하는 문서
- 테스트 레벨, 테스트 유형, 도구, 환경, 리스크 관리 방안 등을 포함
- 테스트 관리자나 QA 리더가 작성
- 테스트 정책을 프로젝트 상황에 맞게 구체화
- 프로젝트의 전반적인 테스트 방향을 제시하는 지침 역할

## 테스트 계획 (Test Plan)

- 특정 테스트 활동의 범위, 접근 방식, 리소스, 일정 등을 상세히 기술한 문서
- 테스트 항목, 테스트 기능, 테스트 환경, 일정, 담당자 등을 명시
- 테스트 리더가 작성
- 테스트 활동을 계획하고 추적하는 기준이 됨
- IEEE 829 표준은 테스트 계획 문서의 구조와 내용에 대한 지침을 제공

## 테스트 케이스 (Test Case)

- 특정 테스트 목적이나 테스트 조건에 대한 입력 값, 실행 조건, 기대 결과 등을 명세한 문서
- 테스트 케이스 ID, 설명, 전제 조건, 단계, 기대 결과, 상태 등을 포함
- 테스트 엔지니어가 작성
- 실제 테스트 실행의 기준이 됨
- 재사용 가능하고 추적 가능하도록 구조화되어야 함

# 테스트 계획서 (Test Plan) 작성



# 테스트 케이스 (Test Case) 설계

## 테스트 케이스 구성 요소

효과적인 테스트 케이스는 다음과 같은 요소를 포함

- 식별자:** 고유한 ID와 이름
- 설명:** 테스트의 목적과 내용
- 전제 조건:** 테스트 실행 전 필요한 조건
- 테스트 데이터:** 사용할 입력 값
- 테스트 단계:** 순차적인 실행 단계
- 기대 결과:** 예상되는 출력이나 동작
- 실제 결과:** 테스트 실행 후 관찰된 결과
- 상태:** 통과, 실패, 보류 등의 상태
- 비고:** 추가 정보나 특이사항

## 테스트 케이스 예시

테스트 케이스 ID: TC-LOGIN-001

제목: 유효한 사용자 자격 증명으로 로그인

설명: 유효한 사용자 이름과 비밀번호로 로그인 기능 검증  
전제 조건:

- 사용자가 시스템에 등록되어 있음
- 로그인 페이지에 접속 가능함

테스트 데이터:

- 사용자 이름: valid\_user
- 비밀번호: correct\_password

테스트 단계:

- 로그인 페이지로 이동
- 사용자 이름 입력란에 'valid\_user' 입력
- 비밀번호 입력란에 'correct\_password' 입력
- 로그인 버튼 클릭

기대 결과:

- 로그인 성공 메시지 표시
- 사용자가 대시보드 페이지로 리디렉션됨
- 사용자 이름이 화면 상단에 표시됨

# 테스트 보고서 (Test Report) 작성

## 테스트 요약 보고서



- 테스트 활동의 전반적인 상태와 결과를 요약
- 테스트 진행 상황, 주요 결함, 품질 평가, 리스크 상태 포함
- 이해관계자에게 간결한 정보 제공

## 결함 보고서



- 발견된 결함에 대한 상세 정보 기록
- 결함 ID, 설명, 재현 단계, 심각도, 우선순위, 상태 등 포함
- 개발팀이 결함을 이해하고 수정하는 데 필요한 정보 제공



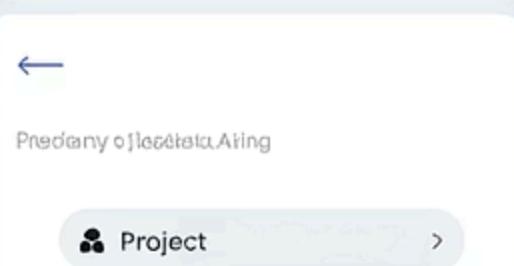
## 테스트 완료 보고서

- 테스트 활동 완료 후 작성하는 종합 보고서
- 계획 대비 실제 테스트 범위, 결과 요약, 미해결 결함, 품질/리스크 평가 포함
- 릴리스 결정을 위한 근거 제공



## 인수 테스트 보고서

- 인수 테스트 결과를 기록한 보고서
- 테스트된 인수 기준, 결과, 미충족 항목, 이해관계자 의견 포함
- 시스템 인수 결정의 근거 제공



## 테스트 메트릭 수집과 분석 예

# 85%

### 테스트 커버리지

- 요구사항, 코드, 기능 등 테스트 대상의 테스트 범위를 나타내는 지표
- 코드 커버리지(구문, 분기, 조건 등) 측정 가능
- 요구사항 커버리지, 기능 커버리지 등 다양한 측면에서 평가

# 42

### 결함 밀도

- 코드 크기 대비 발견된 결함의 수를 나타내는 지표
- KLOC(천 라인 당 결함 수) 단위로 측정
- 코드의 품질과 테스트 효과성 평가에 활용

# 78%

### 테스트 실행율

- 계획된 테스트 케이스 중 실제로 실행된 비율
- 테스트 진행 상황 모니터링에 활용
- 계획 대비 실제 진행 상황 평가 지표

# 4.2

### 결함 제거 효율성

- 테스트 중 발견된 결함 수와 릴리스 후 발견된 결함 수의 비율
- 테스트의 효과성을 평가하는 중요한 지표
- 수치가 높을수록 테스트가 효과적임을 의미

# 테스트 툴체인 구성

## 테스트 관리 도구

- 테스트 케이스 관리, 테스트 실행 계획, 결과 추적 등을 지원하는 도구
- TestRail, Zephyr, qTest 등이 대표적
- 테스트 활동의 계획, 추적, 보고를 효율적으로 수행 가능

## CI/CD 도구

- 지속적 통합과 배포 파이프라인을 구성하여 테스트 자동화를 지원하는 도구
- Jenkins, GitHub Actions, CircleCI, Travis CI 등이 많이 사용됨
- 코드 변경 시 자동으로 테스트를 실행



## 결함 추적 도구

- 발견된 결함을 기록, 할당, 추적하는 도구
- JIRA, Bugzilla, Mantis 등이 많이 사용됨
- 결함의 라이프사이클을 관리와 워크플로우 자동화 지원

## 테스트 자동화 도구

- 테스트 실행을 자동화하는 도구
- 다양한 레벨과 유형의 테스트 지원
- Selenium, Cypress, JUnit, PyTest, Appium, JMeter 등 테스트 유형과 기술 스택에 따라 다양한 도구 존재

## 코드 품질 분석 도구

- 코드의 정적 분석을 통해 잠재적 문제와 품질 이슈를 식별하는 도구
- SonarQube, ESLint, PMD, Checkstyle 등이 대표적
- 코드 품질 메트릭을 수집하고 분석

# 테스트 환경 관리

## 테스트 환경의 유형

- 개발 환경, 테스트 환경, 스테이징 환경, 운영(production) 환경 등 다양한 환경 필요
  - 스테이징(staging) 환경: 운영 환경과 거의 동일한 환경을 만들어 놓고, 운영 환경으로 이전하기 전 여러 비기능적 부분 (보안, 성능, 장애 등) 검증하는 환경
- 각 환경의 목적과 특성 명확히 정의 및 일관성 유지 중요
- 테스트 유형별 특화 환경 구성 가능 (단위 테스트, 통합 테스트, 성능 테스트 등)

## 환경 구성 자동화

- Infrastructure as Code(IaC) 도구 활용하여 환경 구성 자동화
- Terraform, Ansible, Docker, Kubernetes 등으로 일관되고 재현 가능한 환경 구축
- 환경 구성 스크립트 버전 관리 및 CI/CD 파이프라인 통합으로 신속한 프로비저닝

## 테스트 데이터 관리

- 데이터 생성, 마스킹, 리프레시 등 기능 자동화
- 테스트 데이터의 품질과 일관성 보장 필요
- 프로덕션 데이터 사용 시 데이터 익명화와 보안 조치 철저히 필요

## 환경 모니터링

- 테스트 환경 상태와 가용성 지속적 모니터링으로 문제 조기 발견 및 해결
- 리소스 사용량, 응답 시간, 오류율 등 지표 수집 및 분석
- 환경 장애 발생 시 신속 대응 프로세스 마련으로 테스트 중단 최소화

# 리스크(Risk) 기반 테스트 전략

## 리스크 식별

- 프로젝트에 영향을 미칠 수 있는 잠재적 리스크 식별
- 기술적 리스크: 새로운 기술, 복잡한 기능 등
- 비즈니스 리스크: 중요 고객 영향, 재정적 손실 등
- 프로젝트 리스크: 일정 지연, 리소스 제약 등

## 리스크 평가

- 식별된 리스크의 영향도와 발생 가능성 평가
- 리스크 매트릭스 활용하여 고위험, 중위험, 저위험 영역으로 분류
- 객관적인 기준에 따라 우선순위 부여

## 테스트 전략 수립

- 리스크 수준에 따라 테스트 노력 분배
- 고위험 영역: 더 많은 테스트 케이스, 철저한 테스트 기법, 빈번한 테스트 적용
- 저위험 영역: 기본적인 테스트만 수행

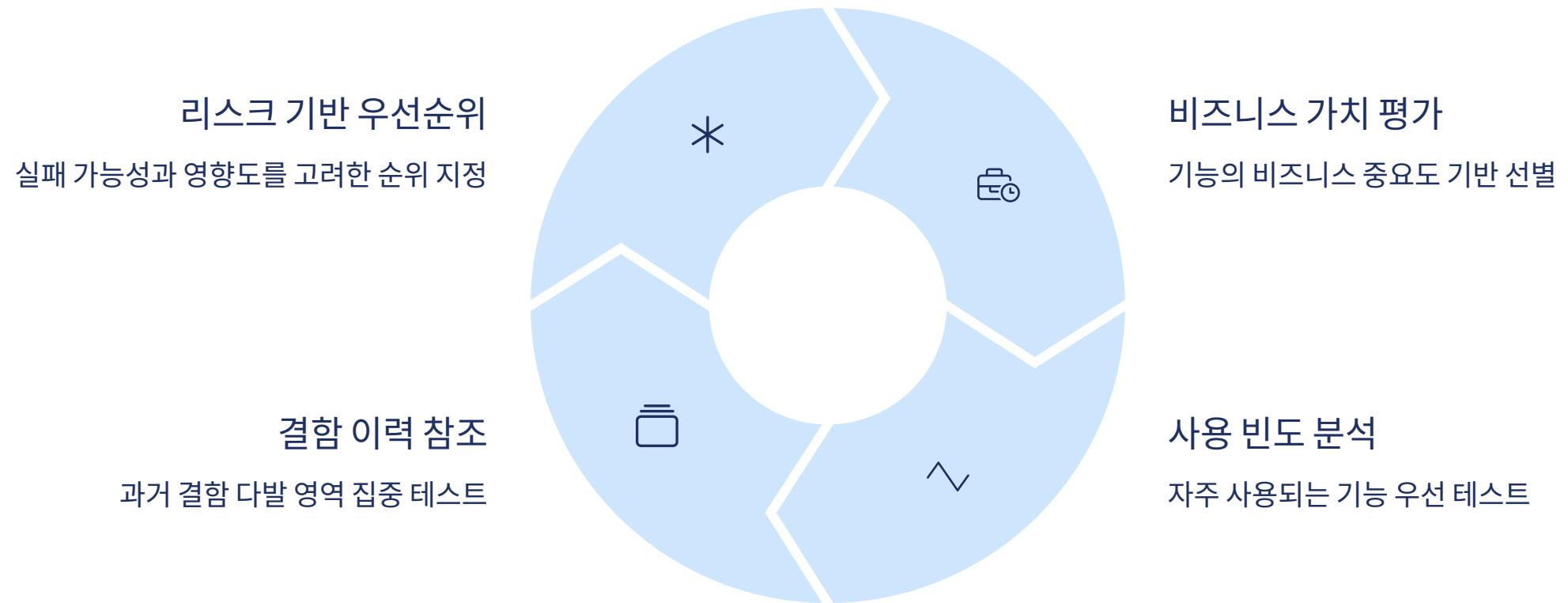
## 테스트 실행 및 모니터링

- 리스크 기반으로 우선순위가 부여된 테스트 실행
- 테스트 결과 지속적 모니터링
- 리스크 상태 지속적 업데이트
- 새로운 리스크 발견 시 테스트 전략 조정

## 리스크 보고 및 의사결정

- 테스트 결과와 리스크 상태를 이해관계자에게 투명하게 보고
- 잔존 리스크와 그 영향을 명확히 커뮤니케이션
- 릴리스 결정을 위한 객관적 근거 제공

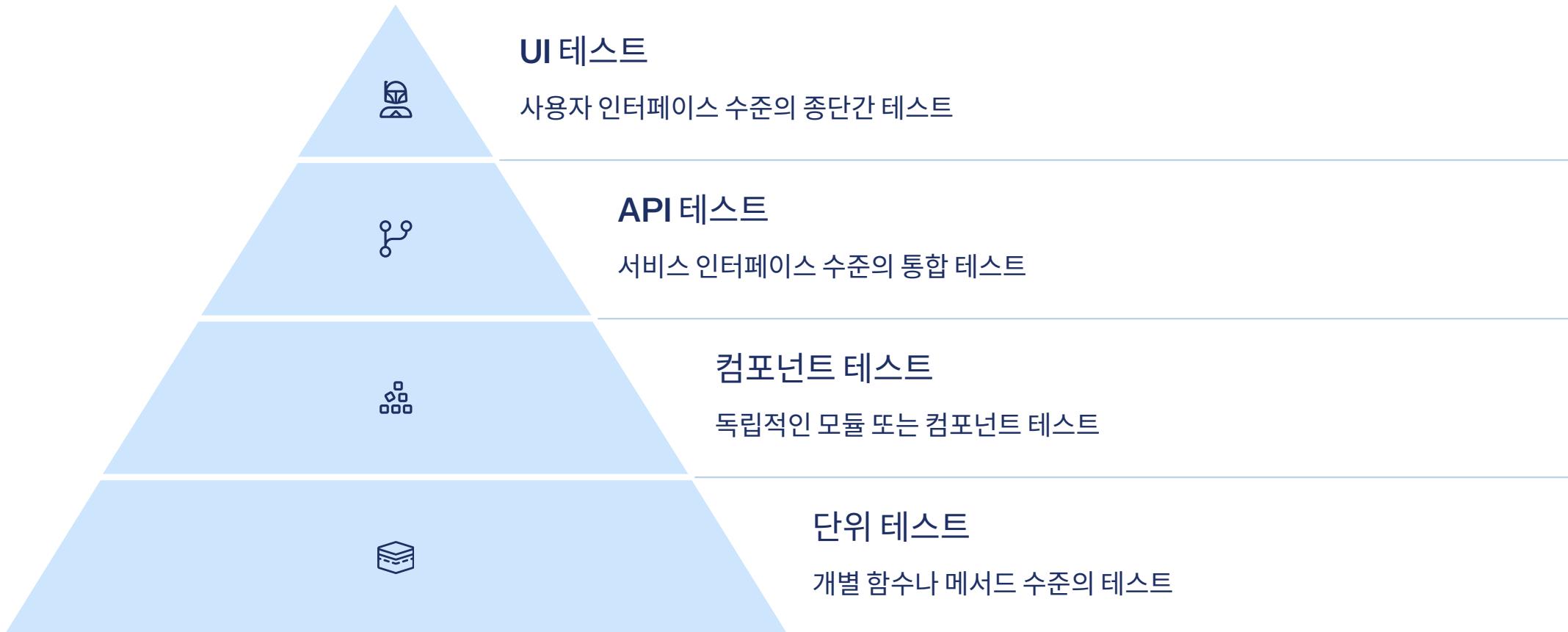
# 테스트 케이스 우선순위 지정 방법



- 목적:** 제한된 시간과 리소스 내에서 효과적인 테스트 수행
- 핵심 원칙:** 중요도와 위험도가 높은 영역을 우선 테스트하여 주요 결함 조기 발견
- 특성:** 정적이 아닌 동적 프로세스로 프로젝트 상황에 따라 조정 필요
- 효과적 방법:** 리스크 기반 테스트 접근법 활용
- 성공 요소:** 개발팀, 비즈니스 이해관계자, 품질 보증 팀 간 긴밀한 협업

우선순위	기준	테스트 범위
P0 (최고)	치명적 오류 가능성, 핵심 비즈니스 기능	모든 릴리스마다 필수 테스트
P1 (높음)	주요 기능, 자주 사용됨, 과거 결합 다수	가능한 모든 릴리스에서 테스트
P2 (중간)	일반 기능, 중간 정도의 사용 빈도	주요 릴리스와 관련 변경 시 테스트
P3 (낮음)	부가 기능, 낮은 사용 빈도, 영향 제한적	시간이 허용될 때 또는 관련 변경 시에만 테스트

# 테스트 자동화 전략



- **테스트 자동화 피라미드**: 다양한 수준의 테스트와 그 비중을 시각화한 모델
- **이상적인 구조**: 피라미드 아래쪽(단위 테스트)이 가장 많은 비중, 위로 올라갈수록(UI 테스트) 수가 감소
- **단위 테스트 특징**: 실행 속도 빠름, 안정적, 유지보수 용이 → 많은 수 작성 권장
- **UI 테스트 특징**: 실행 시간 길, 불안정 가능성 → 중요 시나리오만 선별적 자동화 필요
- **API 테스트 가치**: API 테스트는 UI보다 안정적이면서도 높은 수준의 통합을 검증할 수 있어 좋은 균형점이 됨

# 지속적 통합(CI)과 테스트



## 코드 커밋

- 개발자가 코드를 버전 관리 시스템에 커밋하면 CI 파이프라인 트리거
- 작은 단위의 변경을 자주 커밋하는 것이 권장됨
- 문제를 조기에 발견하고 해결하는 데 도움

## 빌드 및 정적 분석

- 코드 컴파일 및 정적 코드 분석 수행
- 코딩 표준 준수, 잠재적 버그, 보안 취약점 등 검사
- 기본적인 코드 품질 보장

## 단위 테스트

- 개별 코드 단위의 기능을 검증하는 자동화된 테스트 실행
- 빠른 피드백 제공
- 코드 변경으로 인한 회귀 문제 조기 발견

## 통합 테스트

- 컴포넌트 간의 상호작용을 검증하는 테스트 실행
- API 테스트, 데이터베이스 통합 테스트 등 수행

## 보고 및 피드백

- 테스트 결과 수집 및 분석하여 개발자에게 즉시 피드백 제공
- 테스트 커버리지, 코드 품질 메트릭 등 시각화
- 개선 방향 제시

# 지속적 배포(CD)와 테스트

-  CI 단계 완료  
지속적 통합 단계에서 모든 테스트를 통과한 코드가 CD 파이프라인으로 전달됨  
빌드 산출물이 생성되고, 배포 준비가 완료됨
-  보안 및 컴플라이언스 테스트  
보안 취약점 스캔, 컴플라이언스 검증 등의 테스트를 수행  
SAST(정적 애플리케이션 보안 테스트), DAST(동적 애플리케이션 보안 테스트), SCA(소프트웨어 구성 분석) 등의 도구를 활용
-  스테이징 환경 배포  
프로덕션과 유사한 스테이징 환경에 애플리케이션을 배포  
배포 과정 자체를 테스트하고, 환경 구성의 일관성을 검증
-  인수 테스트  
스테이징 환경에서 자동화된 인수 테스트를 실행하여 비즈니스 요구사항 충족 여부를 검증  
UI 테스트, 시나리오 테스트 등이 이 단계에서 수행됨
-  성능 및 부하 테스트  
시스템의 성능, 확장성, 안정성을 검증하는 테스트 수행  
응답 시간, 처리량, 리소스 사용률 등을 측정하고, 성능 병목 현상을 식별
-  프로덕션 배포  
모든 테스트를 통과한 코드를 프로덕션 환경에 배포

# 테스트 효율성 향상 전략

## 테스트 자동화 최적화

테스트 자동화를 효율적으로 구현하여 테스트 실행 시간을 단축하고 유지보수성을 높입니다. 자동화 스크립트의 모듈화, 재사용 가능한 컴포넌트 개발, 데이터 주도 접근법 등을 활용하여 자동화의 ROI를 극대화

- 페이지 객체 모델(POM) 패턴 적용
- 테스트 데이터 외부화
- 병렬 테스트 실행 구현

## 테스트 데이터 관리 개선

테스트에 필요한 데이터를 효율적으로 관리하여 테스트 준비 시간을 단축. 테스트 데이터 생성 자동화, 데이터 셋 버전 관리, 환경 간 데이터 동기화 등의 방법을 활용

- 데이터 생성 도구 활용
- 테스트 데이터 버전 관리
- 필요 시점에 데이터 생성(JIT)

## 리스크 기반 테스트 범위 조정

모든 것을 테스트하려 하기보다 리스크 분석을 통해 중요 영역에 테스트 노력을 집중다. 회귀 테스트 범위를 변경 영향 분석에 기반하여 최적화하고, 불필요한 테스트를 줄임

- 변경 영향 분석 수행
- 리스크 매트릭스 활용
- 테스트 케이스 우선순위 지정

## 테스트 프로세스 간소화

테스트 프로세스에서 불필요한 단계를 제거하고, 병목 현상을 해소하여 전체 사이클 시간을 단축. 애자일 테스팅, 지속적 테스팅 접근법을 도입하여 피드백 루프를 짧게 유지

- 테스트 환경 프로비저닝 자동화
- 테스트 보고 자동화
- 셀프 서비스 테스트 도구 제공

# 테스트 성숙도 모델 (Testing Maturity Model)

## 레벨 1: 초기 (Initial)

- 테스트 프로세스가 혼란스럽고 비정형화됨
- 문서화된 프로세스가 거의 없음
- 테스트가 임시적이고 반응적으로 수행됨
- 개인의 경험과 노력에 크게 의존
- 테스트 결과의 예측 가능성과 반복 가능성이 낮음

## 레벨 2: 관리됨 (Managed)

- 기본적인 테스트 프로세스 수립
- 프로젝트별 테스트 계획 작성
- 테스트 활동이 추적되고 통제됨
- 일부 테스트 기법과 도구 사용
- 조직 전체의 일관성은 아직 부족함

## 레벨 3: 정의됨 (Defined)

- 표준화된 테스트 프로세스가 조직 전체에 적용
- 테스트 정책, 전략, 프로세스가 문서화됨
- 테스트 인력의 역할과 책임이 명확히 정의됨
- 테스트 프로세스의 개선 시작
- 메트릭 수집이 체계화됨

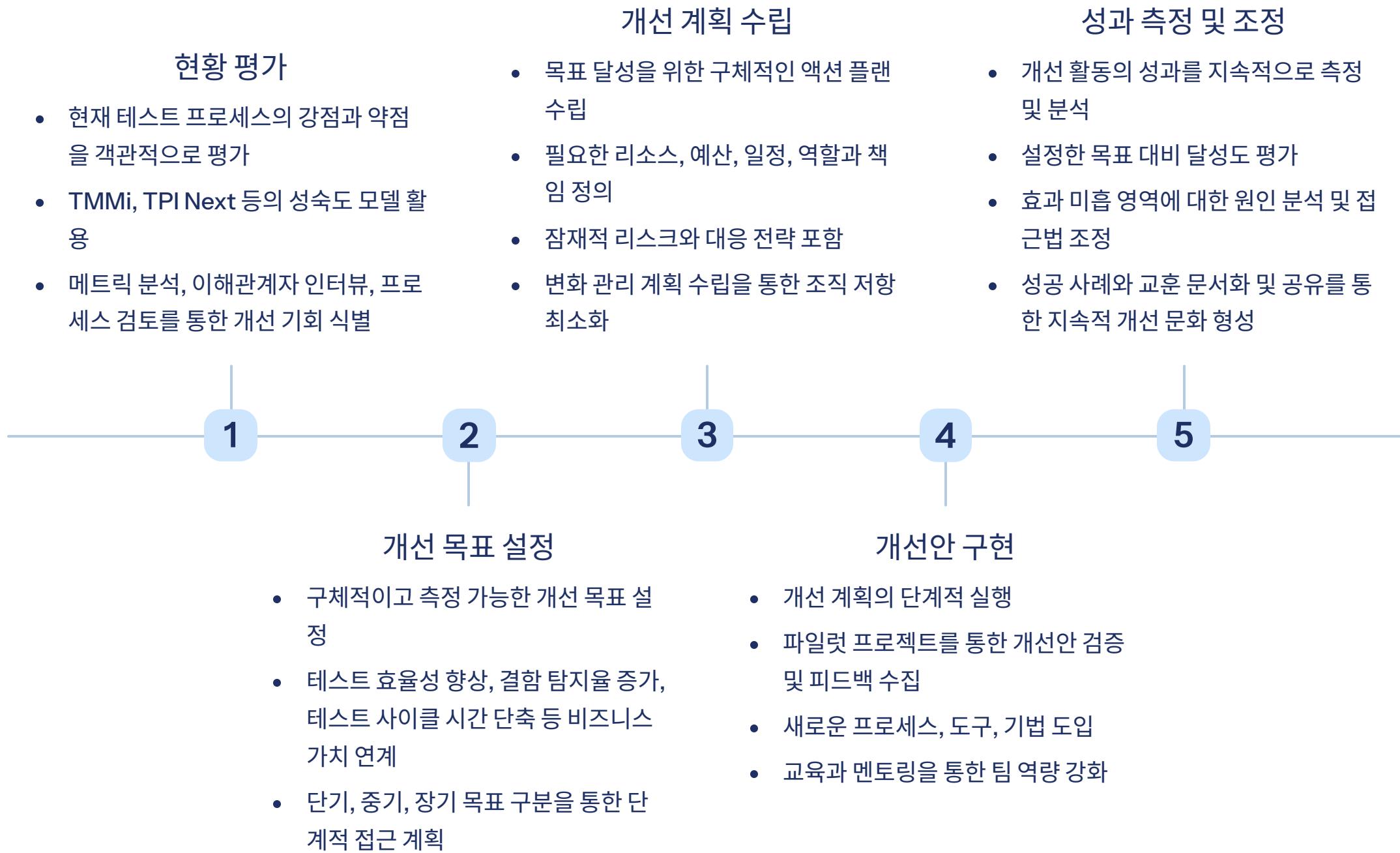
## 레벨 4: 측정됨 (Measured)

- 테스트 프로세스가 정량적으로 측정되고 관리됨
- 테스트 효율성, 효과성, 품질에 대한 메트릭 체계적 수집·분석
- 측정 결과를 바탕으로 테스트 프로세스 예측 가능
- 지속적인 개선이 이루어짐

## 레벨 5: 최적화됨 (Optimized)

- 테스트 프로세스가 지속적으로 개선되고 최적화됨
- 혁신적인 테스트 기법과 도구 도입
- 결함 예방에 중점을 둠
- 테스트 자동화가 광범위하게 적용됨
- 품질이 조직 문화의 핵심 가치가 됨

# 테스트 프로세스 개선



# 변화하는 환경에서의 테스트

## 클라우드 환경에서의 테스트

클라우드 기반 애플리케이션은 확장성, 탄력성, 가용성 등의 특성을 검증하기 위해 다음과 같은 테스트 전략이 중요

- 클라우드 서비스 장애를 시뮬레이션하는 카오스 엔지니어링
- 탄력적 확장 및 축소 기능을 검증하는 확장성 테스트
- 다양한 리전과 가용 영역에서의 성능을 검증하는 지리적 분산 테스트
- 클라우드 비용 최적화를 위한 효율성 테스트

## マイ크로서비스 아키텍처 테스트

マイ크로서비스 아키텍처는 분산 시스템의 복잡성을 증가시키며, 서비스 간 통신, 격리, 복원력 등을 검증하기 위한 전문화된 테스트 전략이 필요

- 서비스 간 계약을 검증하는 계약 테스트 (Contract Testing)
- 서비스 의존성을 관리하는 소비자 주도 계약 테스트(CDC)
- 서비스 복원력을 검증하는 내결함성 테스트
- 분산 트랜잭션과 데이터 일관성 검증

## DevOps 및 지속적 배포 환경

DevOps 문화와 지속적 배포 환경에서는 테스트가 파이프라인의 핵심 부분이 되며, 자동화와 신속한 피드백이 중요

- 모든 레벨의 테스트를 포함하는 지속적 테스트 파이프라인
- 빠른 피드백을 위한 테스트 최적화 및 병렬화
- 프로덕션 유사 환경에서의 테스트 수행
- 모니터링과 관측성을 통한 프로덕션 테스팅

# 테스트 관련 자격증 및 교육



## ISTQB (International Software Testing Qualifications Board)

소프트웨어 테스팅 분야에서 가장 널리 인정받는 자격증 중 하나로, 다양한 레벨의 인증을 제공

- Foundation Level: 테스팅의 기본 개념과 용어
- Advanced Level: Test Manager, Test Analyst, Technical Test Analyst
- Expert Level: Test Management, Test Automation 등

ISTQB 자격증은 국제적으로 인정받으며, 체계적인 테스트 지식을 증명하는데 도움됨



## 기술 및 도구 특화 인증

특정 테스트 도구나 기술에 특화된 자격증은 실무 능력을 증명하는 데 유용

- Selenium Certification: 웹 자동화 테스트 도구
- JMeter Certification: 성능 테스트 도구
- AWS Certified DevOps Engineer: 클라우드 환경의 CI/CD
- Certified Scrum Master: 애자일 환경에서의 테스트

이러한 인증은 실무에서 바로 활용 가능한 기술을 검증하며, 특정 도메인에서의 전문성을 강화



## 추천 교육 자원

지속적인 학습을 위한 다양한 교육 자원

- 온라인 코스: Udemy, Coursera, Pluralsight 등의 플랫폼
- 서적: "테스트 주도 개발", "Agile Testing", "Continuous Delivery" 등
- 커뮤니티: 테스트 블로그, Stack Overflow, 테스트 컨퍼런스
- 실습: 오픈 소스 프로젝트 참여, 개인 프로젝트 수행

테스트 분야는 빠르게 변화하므로, 지속적인 학습과 실무 경험이 중요

# 마무리: 테스트 자동화와 전략 수립의 미래

## 소프트웨어 테스트의 미래 트렌드

- 인공지능과 머신러닝의 활용 확대
- 테스트의 좌측 이동(Shift-Left) 강화
- 지속적 테스팅(Continuous Testing) 체계 정착
- 코드로서의 테스트(Test-as-Code) 접근법 확산
- AI 기반 테스트 케이스 생성 및 우선순위 결정
- 자동화된 결함 예측 및 진단 시스템 발전

## 테스트 전략 수립의 변화

- 고정된 계획에서 적응적이고 유연한 전략으로 전환
- 데이터 기반 의사결정 중요성 증가
- 지속적인 피드백과 측정에 기반한 접근법 강화
- 비즈니스 가치와 사용자 경험에 직접 연계된 테스트 강조
- 지속적 학습과 기술 발전이 테스트 전문가의 핵심 역량으로 부상

