Unit Testing

Software Engineering At Google 책 12장 이야기

https://abseil.io/resources/swe_at_google.2.pdf

단위 테스팅

- small size: 프로세스/스레드/IO/블럭킹X
- small scope: 클래스/메서드
- 구글에선 전체의 80% 정도를 차지
- 버그 예방
- 생산성 향상

생산성 향상

> 버그 예방 외에 가장 중요한 목적

- 빠르고 결정적 → 개발 중 자주 실행하여 즉각적 피드백
- 작성하기 쉬우며, 따라서 커버리지 높이기 좋음 → 자신감
- 잘못된 것을 알려줌
- 문서와 예시로써 동작

maintainable

- 정말 빈번하게 실행됨
- 엔지니어 삶에 큰 부분
- 따라서 유지보수성에 엄청나게 신경 씀
- maintainable = just working

> 테스트가 실패하기 전까지는 신경 쓸 이유 없음. 만약, 실패가 났다면 이는 진짜 버그를 가리키며, 동시에 명확한 원인을 알려줌.

현생

> 메리는 제품에 간단한 기능을 추가하고자 한다. 이는 몇 줄의 코드만으로 빠르게 구현할 수 있다. 하지만 변경을 제출하려고 하니, 자동화 된 테스트가 이것 저것 실패하는 것이 보인다. 그녀는 남은 하루 동안 실패를 하나씩 확인한다. 각 케이스를 살펴 보니, 실제 버그에 의한 실패가 아니다. 대신, 테스트가 코드 내부의 구조에 대해 가정하고 있던 것들이 바뀌었기 때문이다. 테스트에는 이 가정의 변경을 반영해 줘야 할 뿐이다. 그녀는 종종 테스트가 원래 의도했던 것들을 알아차리기 힘들다. 그래서 그녀는 꼼수를 써서 테스트를 고치는데 이는 테스트를 더욱 이해하기 어렵게 만든다. 결국, 금방 끝날 일이 몇 시간 또는 몇 일이 걸리게 되고, 메리의 생산성과 동기부여를 떨어뜨린다.

무엇이 문제였나 ❓

1. brittle: 버그도 아니고 관련도 없는 변경에 테스트들이 깨짐
2. not clear: 실패의 이유를 명확히 찾기가 어려움

깨지기 쉬운 테스트?

- 버그를 가져온 변경이 아닌데도 관련 없는 테스트들이 깨짐
- 자꾸 깨진다면 테스트 유지보수에 점점 더 많은 시간 소모
- 변경이 일어날 때마다 테스트도 함께 수정해줘야 한다? = 자동화 된 테스트 ❌
- 구글과 같은 규모에서는 심각한 문제

언제 깨져야 할까?

> 이상적인 테스트란, 관련 요구사항이 바뀌지 않는 한 바꾸지 않아도 되는 것

- Pure refactorings ❌
- New features ❌
- Bug fixes ❌
- Behavior changes ⭕
  (다른 3가지에 비해 비용 큼)

Preventing Brittle Tests

1. **Test via Public APIs**

```
public void processTransaction(Transaction transaction) {
  if (isValid(transaction)) {
    saveToDatabase(transaction);
  }
}

private boolean isValid(Transaction t) {
  return t.getAmount() < t.getSender().getBalance();
}
```

```java
@Test
public void emptyAccountShouldNotBeValid() {
  var transaction = newTransaction().setSender(EMPTY_ACCOUNT);
  assertThat(processor.isValid(transaction)).isFalse();
}

@Test
public void shouldSaveSerializedData() {
  processor.saveToDatabase(newTransaction()
      .setId(123)
      .setSender("me")
      .setRecipient("you")
      .setAmount(100));
  assertThat(database.get(123)).isEqualTo("me,you,100");
}
```

```java
@Test
public void shouldTransferFunds() {
  processor.setAccountBalance("me", 150);
  processor.setAccountBalance("you", 20);

  processor.processTransaction(newTransaction()
      .setSender("me")
      .setRecipient("you")
      .setAmount(100));

  assertThat(processor.getAccountBalance("me")).isEqualTo(50);
  assertThat(processor.getAccountBalance("you")).isEqualTo(120);
}

@Test
public void shouldNotPerformInvalidTransactions() {
  processor.setAccountBalance("me", 50);
  processor.setAccountBalance("you", 20);

  processor.processTransaction(newTransaction()
      .setSender("me")
      .setRecipient("you")
      .setAmount(100));

  assertThat(processor.getAccountBalance("me")).isEqualTo(50);
  assertThat(processor.getAccountBalance("you")).isEqualTo(20);
}
```

Preventing Brittle Tests

1. **Test via Public APIs**
2. **Test State, Not Interactions**

```
@Test
public void shouldWriteToDatabase() {
  accounts.createUser("foobar");
  verify(database).put("foobar");
}
```

```
@Test
public void shouldCreateUsers() {
  accounts.createUser("foobar");
  assertThat(accounts.getUser("foobar")).isNotNull();
}
```

Writing Clear Tests

1. **Make Your Tests Complete and Concise**

```
@Test
public void shouldPerformAddition() {
  Calculator calculator = new Calculator(
    new RoundingStrategy(),
    "unused",
    ENABLE_COSINE_FEATURE,
    0.01,
    calculusEngine,
    false
  );

  int result = calculator.calculate(newTestCalculation());
  assertThat(result).isEqualTo(5); // Where did this number come from?
}
```

```java
@Test
public void shouldPerformAddition() {
  Calculator calculator = newCalculator();
  int result = calculator.calculate(newCalculation(2, Operation.PLUS, 3));
  assertThat(result).isEqualTo(5);
}
```

Writing Clear Tests

1. Make Your Tests Complete and Concise
2. **Test Behaviors, Not Methods**

```
public void displayTransactionResults(User user, Transaction transaction) {
  ui.showMessage("You bought a " + transaction.getItemName());
  if (user.getBalance() < LOW_BANKACE_THRESHOLD) {
    ui.showMessage("Warning: your balance is low!");
  }
}

@Test
public void testDisplayTransactionResults() {
  transactionProcessor.displayTransactionResults(
    newUserWithBalance(LOW_BALANCE_THRESHOLD.plus(dollars(2))),
    new Transaction("Some Item", dollars(3))
  );

  assertThat(ui.getText()).contains("You bought a Some Item");
  assertThat(ui.getText()).contains("your balance is low");
}
```

```
@Test
public void displayTransactionResults_showsItemName() {
  transactionProcessor.displayTransactionResults(
    new User(),
    new Transaction("Some Item")
  );
  assertThat(ui.getText()).contains("You bought a Some Item");
}

@Test
public void displayTransactionResults_showsLowBalanceWarning() {
  transactionProcessor.displayTransactionResults(
    newUserWithBalance(LOW_BALANCE_THRESHOLD.plus(dollars(2))),
    new Transaction("Some Item", dollars(3))
  );
  assertThat(ui.getText()).contains("your balance is low");
}
```

Test Behaviors, Not Methods 첨언

1. Structure tests to emphasize behaviors (`given when then`)
2. Name tests after the behavior being tested
    - `testUpdateBalance` ❌
    - `displayTransactionResults_showsItemName` ⭕

Writing Clear Tests

1. Make Your Tests Complete and Concise
2. Test Behaviors, Not Methods
3. **Don't Put Logic in Tests**

```
@Test
public void shouldNavigateToAlbumsPage() {
  String baseUrl = "http://photos.google.com/";
  Navigator nav = new Navigator(baseUrl);
  nav.goToAlbumPage();
  assertThat(nav.getCurrentUrl()).isEqualTo(baseUrl + "/albums");
}
```

```
@Test
public void shouldNavigateToPhotosPage() {
  Navigator nav = new Navigator("http://photos.google.com/");
  nav.goToPhotosPage();
  assertThat(nav.getCurrentUrl()))
      .isEqualTo("http://photos.google.com//albums"); // Oops!
}
```

Writing Clear Tests

1. Make Your Tests Complete and Concise
2. Test Behaviors, Not Methods
3. Don't Put Logic in Tests
4. **Write Clear Failure Messages**

```
Test failed: account is closed
```

```
Exepected an account in state CLOSED, but got account: <{name: "my-account", state: "OPEN"}
```

Writing Clear Tests

1. Make Your Tests Complete and Concise
2. Test Behaviors, Not Methods
3. Don't Put Logic in Tests
4. Write Clear Failure Messages
5. **Tests and Code Sharing: DAMP, Not DRY**

*DAMP = Descriptive And Meaningful Phrases
(테스트를 간단하면서도 명료하게 만든다면, DRY 보다는 오히려 약간의 중복을 허용)

```
@Test
public void shouldAllowMultipleUsers() {
  List<User> users = createUsers(false, false);
  Forum forum = createForumAndRegisterUsers(users);
  validateForumAndUsers(forum, users);
}

@Test
public void shouldNotAllowBannedUsers() {
  List<User> users = createUsers(true);
  Forum forum = createForumAndRegisterUsers(users);
  validateForumAndUsers(forum, users);
}
```

```java
@Test
public void shouldAllowMultipleUsers() {
  User user1 = newUser().setState(State.NORMAL).build();
  User user2 = newUser().setState(State.NORMAL).build();

  Forum forum = new Forum();
  forum.register(user1);
  forum.register(user2);

  assertThat(forum.hasRegisteredUser(user1)).isTrue();
  assertThat(forum.hasRegisteredUser(user2)).isTrue();
}

@Test
public void shouldNotRegisterBannedUsers() {
  User user = newUser().setState(State.BANNED).build();

  Forum forum = new Forum();
  try {
    forum.register(user);
  } catch(BannedUserException ignored) {}

  assertThat(forum.hasRegisteredUser(user)).isFalse();
}
```

Writing Clear Tests

1. Make Your Tests Complete and Concise
2. Test Behaviors, Not Methods
3. Don't Put Logic in Tests
4. Write Clear Failure Messages
5. Tests and Code Sharing: DAMP, Not DRY
6. **Shared Values/Setup/Helpers/Validation**

```
private static final Account ACCOUNT_1 = Account...
private static final Account ACCOUNT_2 = Account...
private static final Item ITEM = Item...

@Test
public void canBuyItem_returnsFalseForClosedAccounts() {
  assertThat(store.canBuyItem(ITEM, ACCOUNT_1)).isFalse();
}

@Test
public void canBuyItem_returnsFalseWhenBalanceInsufficient() {
  assertThat(store.canBuyItem(ITEM, ACCOUNT_2)).isFalse();
}
```

```
// 다른 언어
def newContact(
    firstName="Grace", lastName="Hopper", phoneNumber="555-123-4567"):
  return Contact(firstName, lastName, phoneNumber)

// java
private static Contact.Builder newContact() {
  return Contact.newBuilder()
    .setFirstName("Grace")
    .setLastName("Hopper")
    .setPhoneNumber("555-123-4567");
}

// Tests then call methods on the builder to overwrite only the parameters
// that they care about, then call build() to get a real value out of the
// builder.
@Test
public void fullNameShouldCombineFirstAndLastNames() {
  Contact contact = newContact()
      .setFirstName("Ada")
      .setLastName("Lovelace")
      .build();
  assertThat(contact.getFullName()).isEqualTo("Ada Lovelace");
}
```

Shared Setup

```
private NameService nameService;
private UserStore userStore;

@Before
public void setUp() {
  nameService = new NameService();
  nameService.set("user1", "Donald Knuth");
  userStore = new UserStore(nameService);
}

// [... hundreds of lines of tests ...]

@Test
public void shouldReturnNameFromService() {
  // nameService.set("user1", "Margaret Hamilton"); // 숨겨진 의존성을 없애고 싶다면 이 주석을 해제
  UserDetails user = userStore.get("user1");
  assertThat(user.getName()).isEqualTo("Donald Knuth");
}
```

Shared Helpers and Validation

```
private void assertUserHasAccessToAccount(User user, Account account) {
  for (long userId : account.getUsersWithAccess()) {
    if (user.getId() == userId) {
      return;
    }
  }
  fail(user.getName() + " cannot access " + account.getName());
}
```