Clean Code

-TDD: Test Driven Development-

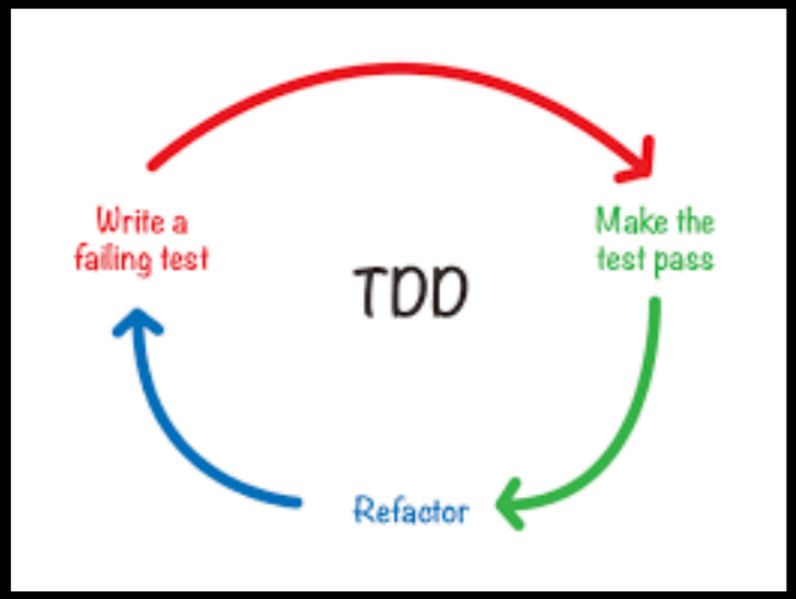
XIO. 220916.

1. 깨끗한 테스트 코드

테스트 코드의 효과

- 단위 테스트 케이스는 실제 코드를 유연하게 만든다.
- 오버 엔지니어링을 방지할 수 있다. (필요한 부분만 구현)
- 설계에 대한 피드백을 빠르게 받을 수 있다.
- 테스트 코드는 실제 코드 변화에 따라 계속해서 변경되어야 하므로 깨끗한 테스트 코드를 작성하여 변경하기 쉽게 한다.
- 테스트 슈트가 없으면 자신이 수정한 코드의 정상 동작을 확인할 수 없다. (결함증가⬆️)

✴️ Q. TDD란?



✅ Test-Driven Development(테스트 주도 개발)

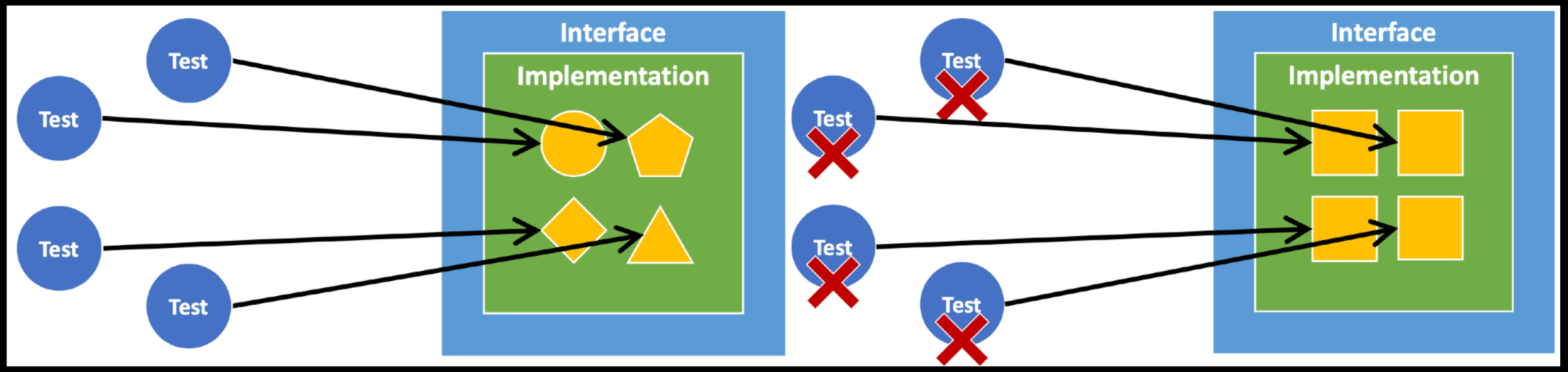
✅ 프로덕션 코드보다 단위 테스트 코드를 먼저 작성하여
코드를 빠르게 검증하고 리팩토링 시 안정성을 확보한다.

✅ 개발 및 테스트 시간 비용을 절감할 수 있다.

[TDD(Test-Driven Development, 테스트 주도 개발) 방법 및 순서]

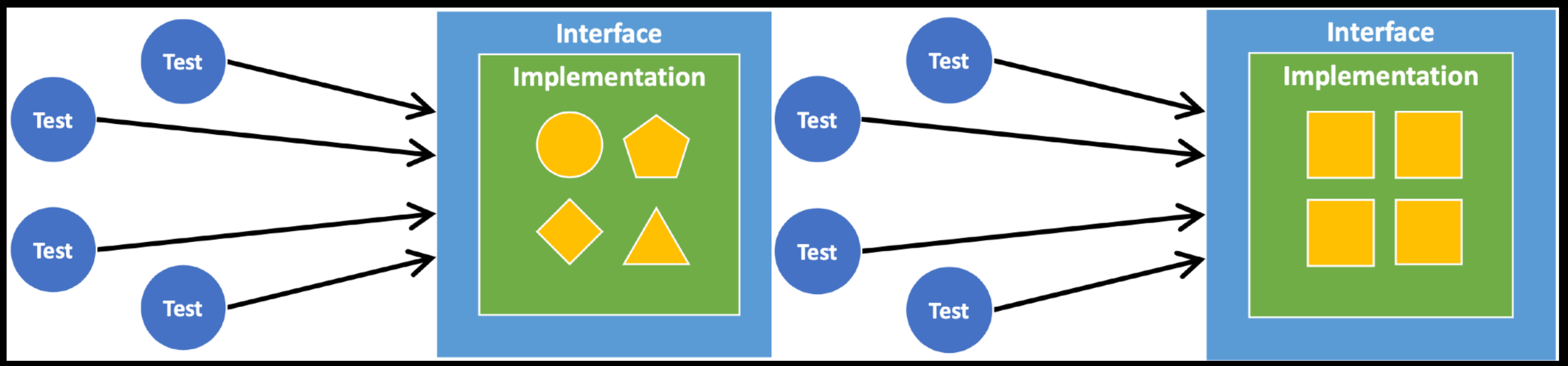
1️⃣ 실패하는 작은 단위 테스트를 작성한다. 처음에는 컴파일조차 되지 않을 수 있다.
2️⃣ 테스트를 통과하기 위해 프로덕션 코드를 빨리 작성한다. (가짜 구현 등)
3️⃣ 그 다음의 테스트 코드를 작성한다. 실패 테스트가 없을 경우에만 성공 테스트를 작성한다.
4️⃣ 새로운 테스트를 통과하기 위해 프로덕션 코드를 추가 또는 수정한다.
5️⃣ 1~4단계를 반복하여 실패/성공의 모든 테스트 케이스를 작성한다.
6️⃣ 개발된 코드들에 대해 모든 중복을 제거하며 리팩토링한다.

구현 테스트



VS

설계 테스트



TEST LEVEL

1. **Unit Testing**
   각 단위 모듈/컴포넌트를 독립적으로 테스트

2. **Integration Testing**
   테스트 된 Unit **모듈을 통합**하는 인터페이스를 테스트
   객체/서비스/시스템 간 여러 작업 단위가 연계된 워크 플로우 테스트

3. **System(Fucntion) Testing**
   통합된 전체 모듈에 오류가 없으면 **기능과 성능이 계획과 일치하는지** 테스트
   주로 API 가장 바깥쪽에 해당하는 코드 검사

4. **Acceptance Testing**
   사용자의 **요구 사항(목적)을 만족하는지** 최종적으로 테스트

🔆 Unit Test

- **가장 작은 단위**의 테스트 (일반적으로 메소드 레벨)
- **변경이 쉬움**
- Sample Code(**코드의 문서화**)의 기반이 됨
  예외상황, 용도, 의존 관계를 한눈에 파악하기 좋음
- 항상 **최신상태**로 **유지**(배포되는 코드와 일치)

" 컴퓨터 프로그래밍에서 소스 코드의 특정 모듈이 의도된 대로 정확히 작동하는지 검증하는 절차 "
" 모든 메소드에 대해 단기간에 문제를 파악할 수 있도록 서로 분리된 Test Case "

+) DSL(Domain Specific Language)

특정 분야에 특화된 언어 -> 특정 영역의 문제 해결에는 그에 맞는 특화된 도구를 사용하자*!*

코드가 일반 자연어를 읽는 것처럼 쉽게 이해되면서 해당 도메인의 전문가가 이해할 수 있는 표현 방식*(ex-Ruby)*

시스템 조작 API대신 API위에 함수, 유틸리티를 구현해서 사용

TDD의 법칙 세가지

1. 실패하는 Unit Case Test를 작성할 때까지 실제 코드를 작성하지 않는다.
2. 컴파일은 실패하지 않으면서 실행이 실패하는 정도로만 단위 테스트를 작성한다.
3. 현재 실패하는 테스트를 통과할 정도로만 실제 코드를 작성한다.

➡️ BUT. 이 규칙을 따르면 매일 수백 수천 개의 테스트 케이스가 나옴

➡️ 사실상 코드 전부를 테스트하는 것

➡️ 실제 코드와 맞먹을 정도로 방대한 테스트 코드는 심각한 관리 문제 유발

2. 가독성

깨끗한 테스트 코드의 가장 중요한 조건: 가독성

✴️ Example

```
public class testGetPageAsXml() throws Exception {
  crawler.addPage(root, PathParser.parse("PageOne"));
1 crawler.addPage(root, PathParser.parse("PageOne.Child"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPAgeResponder();
2 SimpleResponse response = (SimpleResponse) responder.makeResponse(
    new FitNessContext(root), request
  );
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
3 assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}
```

1. PathParse를 통해 문자열을 pagePath 인스턴스로 변환

2. responder 객체 생성, 수집, 변환 / 인수에서 요청 URL 생성

3. 중복, 잡다한 사항이 많아 표현력이 떨어짐

➡️ 테스트 코드와 무관, 의도만 흐림

➡️ 추상화를 통해 잡다하고 세세한 표현을 숨김

```
public class testGetPageAsXml() throws Exception {
  makePages("PageOne", "PageOne.Child", "PageTwo");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
  );
}
```

테스트 자료 생성

테스트 자료 조작

조작한 결과 확인

✴️ BUILD-OPERATE-CHECK PATTERN

Build (Given)
Input 데이터를 생성

Operate (When)
Build 단계에서 생성한 데이터로 실제 코드 실행

Check (Then)
Operate 단계의 결과값을 확인

```
public class BowlingTest {
  @Test
  public void FrameTest() {
    // Build (Given)
    Boll testBoll = Boll.getInstance(10);
    Frame frame = Frame.getInstance(Boll);
    // Operate (When)
       Boolean isStrike = frame.isStrike();
    // Check (Then)
    assertThat(isStrike).isTrue();
  }
}
```

✴️ == GIVEN-WHEN-THEN PATTERN

Given: 테스트 준비 과정, 테스트의 상태 객체, 필요값등을 정의

When: 테스트 진행 필요 조건 명시, 실제 액션을 하는 테스트의 실행 과정

Then: 테스트 검증 과정, 예상되는 변화 설명

```java
@Test
public void When_Get_Discount_Expect_Minus_100() {
  //given
  String name = "americano";
  int defaultPrice = 1000;
  int expectedPrice = 900;
  when(coffeeRepository.findOne(name))
    .thenReturn(Coffee.builder().name(name).isMilk(false).price(defaultPrice));

  //when
  int actualPrice = coffeeService.getDiscountedPrice(name);

  //Then
  assertEquals(expectedPrice, actualPrice);
}
```

➡️커피 가격 할인 메소드 테스트
➡️*1000*원 짜리 아메리카노를 *100*원 할인해서 *900*원을 리턴하는지 검증

3. 이중표준

이중표준

🔆 Ex- 환경제어 시스템 프로토타입

```
public class turnOnLoTempAlarmAtThreashold() throws Exception {
  hw.setTemp(WAY_TOO_COLD);
  controller.tic();
  assertTrue(hw.heaterState());
  assertTrue(hw.blowerState());
  assertFalse(hw.coolerState());
  assertFalse(hw.hitTempAlarm());
  assertTrue(hw.loTempAlarm());
}
```

1

```
public class turnOnLoTempAlarmAtThreashold() throws Exception {
  wayTooCold();
  assertEquals("HBchL", hw.getState());
}
```

2

온도가 급강하(WAY_TOO_COLD)하면 경보, 온풍기, 송풍기가 가동되는지 확인하는 코드

1. 세부적 기능인 tic함수 -> wayTooCold( )안에 숨김
2. heater, blower, cooler, hi-tmp-alarm, lo-temp-alarm을 의미
   대문자는 ON, 소문자는 OFF

➡️ “그릇된 정보(축약어)를 피하라”는 네이밍 컨벤션에 위배되지만 테스트 코드의 수행기능을 재빨리 이해할 수 있기에 허용된 코드

이중표준

- Test API에 적용하는 표준은 실제 코드에 적용하는 표준과 다르다.

- 실제 환경에서는 안되지만 대게 메모리나 CPU 효율과 관련해서 테스트 환경에서 허용되는 방식이 있다.
  ex) StringBuffer 사용
  -> 실시간 임베디드 시스템 어플리케이션 일 때는 자원이 제한적이므로 금지
  -> BUT 테스트 환경은 자원이 제한적일 가능성이 낮으므로 허용

- 이중표준은 코드의 깨끗함과는 무관하다.

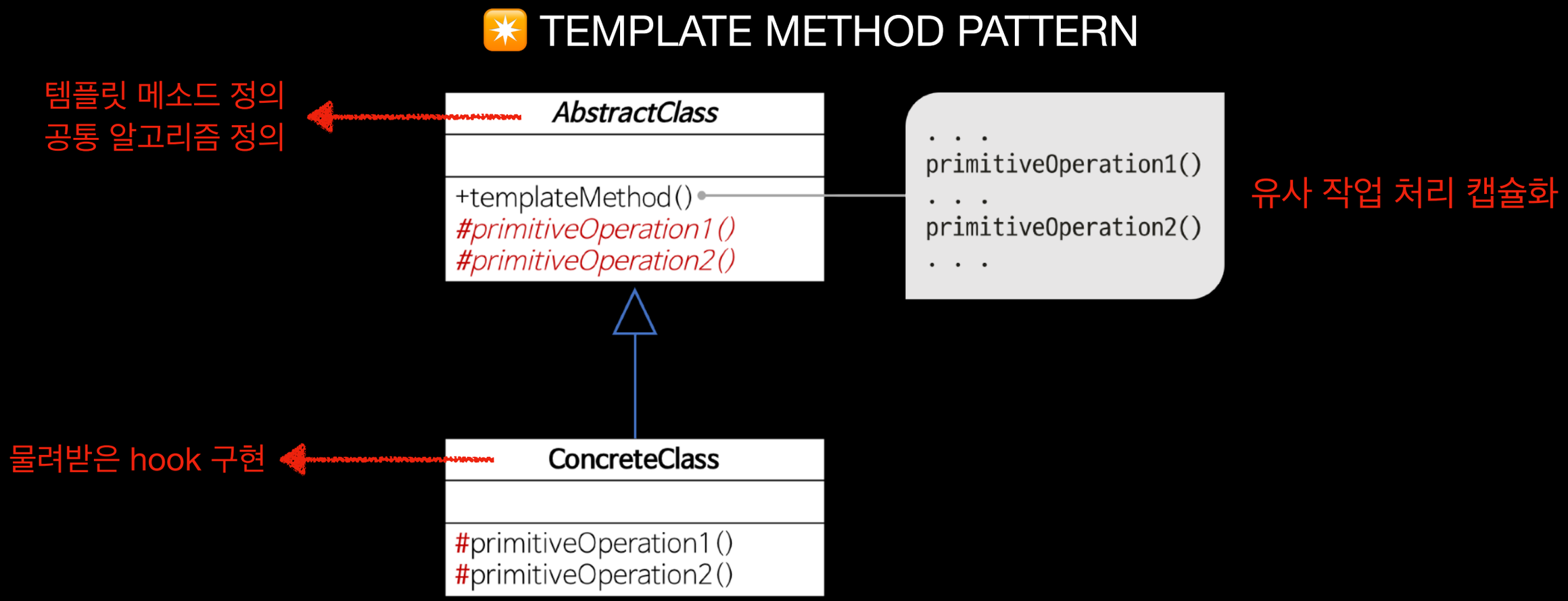
➡️ **빠른 테스트 구현을 위해 세부적인 최적화 과정 생략,**
"B.U.T 깨끗한 테스트 코드의 유지를 위해 적어도 테스트 함수 네이밍에 대한 컨벤션은 맞춰야"

4. 테스트 당 개념 하나

테스트 메소드 하나당 한 개념만 테스트

- 잡다한 개념을 연속으로 테스트하는 긴 함수는 분리해야 한다.
- 독자적인 개념 하나당 독자적인 테스트 한 개로 쪼개야 마땅하다.
- assert문 여러개 사용 가능, BUT ~~한 테스트 함수에서 여러 개념 테스트~~ 지양.
  assert문의 중복 제거 ➡️ Given-When-Then 패턴에 TEMPLATE METHOD 패턴을 사용
  ex) Given, When: 부모 클래스 Then: 자식 클래스

✴️ TEMPLATE METHOD PATTERN



5. F.I.R.S.T 원칙

F.I.R.S.T

Fast
테스트 코드의 **실행시간이 빨라야** 한다.(그래야 자주돌려 문제를 찾아낼 수 있다)

Independent
각 테스트는 **순서에 상관없이 독립적으로 실행 가능** 해야 한다.(실패->도미노)

Repeatable
실제/QA/네트워크X 등 **어떤 환경에서도 테스트가 수행되도록** 해야 한다.(반복가능)

Self-Validating
스스로 true(성공)/false(실패) 검증 → (return 값: boolean)

Timely
적시(테스트하려는 **실제 코드를 구현하기 전**)에

6. Reference

읽어 보기

- TDD https://velog.io/@codemcd/OKKYCON-2018-이규원-당신들의-TDD가-실패하는-이유-u2k4w01f6h
- 테스트 자동화 https://engineering.linecorp.com/ko/blog/server-side-test-automation-journey-1/
- 코드의 문서화 https://ibocon.tistory.com/80

6. Example with Jest

Jest

- Testing Framework made by Facebook

test > unit > products.test.js > ...

```
1   //describe로 그룹화
2   describe('calucation', () => {
3     test('2 더하기 2는 4', () => {
4       expect(2 + 2).toBe(4);
5     });
6
7     test('2 더하기 2는 5가 아님', () => {
8       expect(2 + 2).not.toBe(5);
9     });
10  });
11
```

describe

test

expect 테스트하는값 matcher 기대값