Week 2-2

- Unit Test의 활용
- Unit Test를 활용한 TDD
- 테스트 가능한 코드 작성
- [아하!모먼트]나도 회사를 면접봐야 한다 / 면접에선 울고 나와야 한다
- [아하!모먼트]라이브 이력서 검토 IV

TDD로 구현한 도메인 객체를 활용해 프로그램 완성

구현한 모든 코드를 버린다

다시 TDD로 자동차 경주게임 구현 도전

Unit Test(단위테스트)

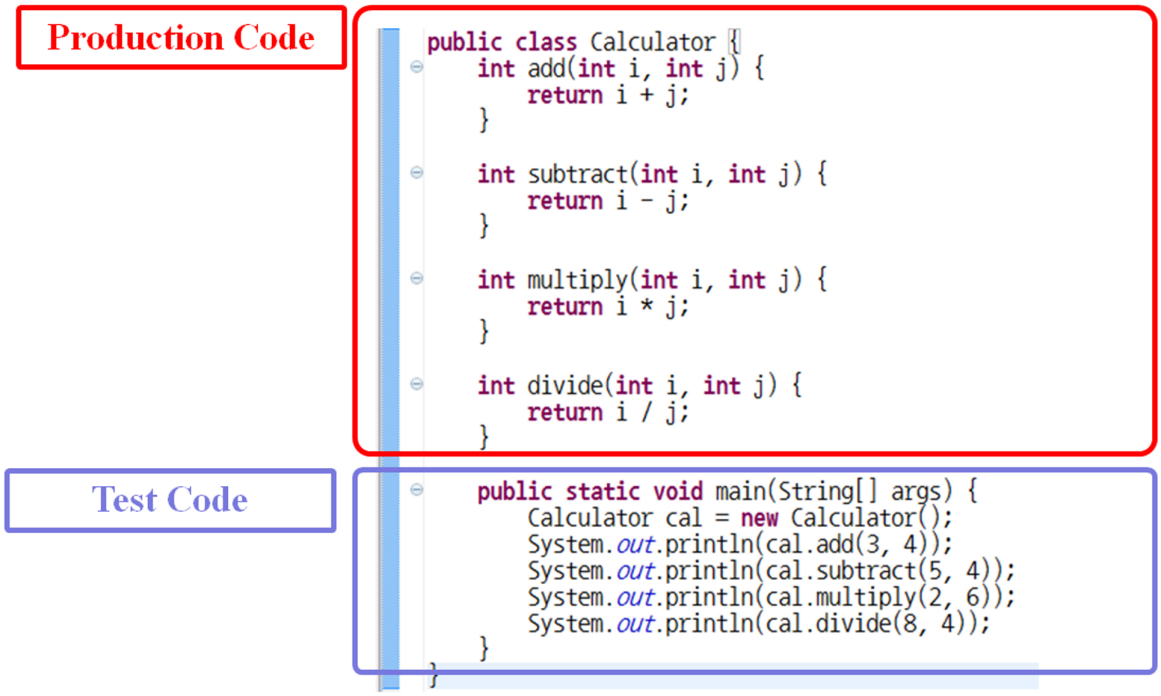
Xcode에서 단위테스트 따라해보기

테스트 맛보기 프로그램 구현

- 사칙연산이 가능한 계산기
  - 덧셈(add)
  - 뺄셈(subtract)
  - 곱셈(multiply)
  - 나눗셈(divide)
- 위 4개의 기능을 구현하고 `XCTest`를 활용해 테스트해봅니다.

- 우리가 앱을 구동하기 위해 작성하는 코드는 대부분 구현 코드(production code)입니다. 구현 코드는 프로그램 구현을 담당하는 부분으로 실제로 사용자에게 전달되어 동작하는 소스 코드를 의미합니다.
- 테스트 코드(test code)는 구현 코드가 정상적으로 동작하는지를 확인하는 코드입니다.
- 단위 테스트는 구현 코드를 구현한 후 작은 단위로 테스트하는 것을 의미합니다.

> 자바 환경에서의 구현 코드와 테스트 코드



XCTest

`XCTest` 는 Xcode 프로젝트에서 활용할 수 있는 테스트 환경을 제공하는 프레임워크입니다.
`XCTest` 를 활용하여 단위 테스트(Unit Test), 사용자 인터페이스 테스트(UI Test), 성능 테스트(Performance Test) 등을 수행할 수 있습니다.
Xcode 프로젝트에서 테스트 코드를 사용하기 위해서는 이 XCTest 프레임워크를 반드시 import 해야합니다.

https://developer.apple.com/documentation/xctest

XCTestCase

```
import XCTest class StrangeCalculatorTests: XCTestCase { ... }
```

SampleTest.swift 파일을 열어보면 테스트를 위한 클래스가 만들어져있는 것을 확인할 수 있을 거예요. 테스트 클래스가 상속하고 있는 `XCTestCase` 는 무엇일까요?

`XCTestCase` 는 추상 클래스인 **XCTest의 하위 클래스**로, 테스트를 작성하기 위해 상속해야 하는 가장 기본적인 클래스입니다. **XCTest는 테스트를 위한 프레임워크의 이름이기도 하고, 테스트에서 가장 기본이 되는 추상 클래스의 이름이기도 합니다.** (프레임워크와 클래스의 이름이 같군요!)
XCTestCase를 상속받은 클래스에서는 test에서 사용되는 다양한 프로퍼티와 메서드를 사용할 수 있습니다.

setUpWithError()

```
override func setUpWithError() throws { // Put setup code here. This method is called before the invocation of each test method in the class. }
```

`setUpWithError()`는 각각의 test case가 실행되기 전마다 호출되어 각 테스트가 모두 같은 상태와 조건에서 실행될 수 있도록 만들어 줄 수 있는 메서드입니다.

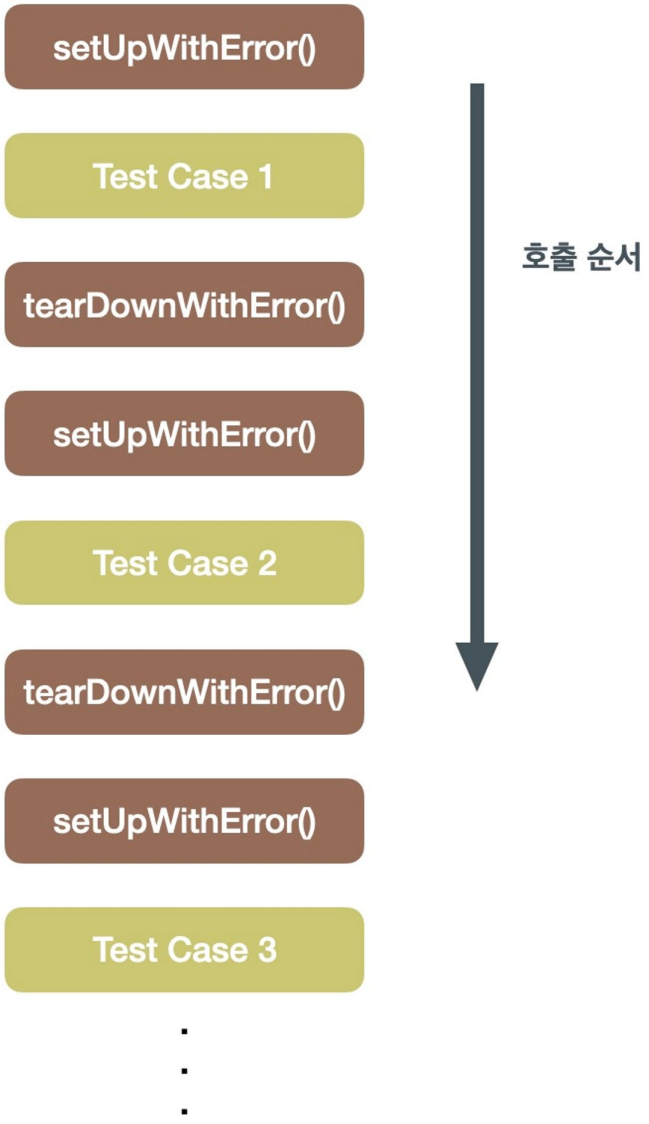
예를 들어 몇몇 테스트가 A라는 값에 대해서 이루어지고 있는데, 먼저 작성된 case에 의해서 값이 변경된다면 다음 테스트는 정상적으로 이루어지지 않을 위험이 있겠죠? 정상적으로 테스트를 하기 위해서는 `setUpWithError`와 같은 메서드를 통해 test case가 이루어질 때마다 테스트의 상태를 reset시켜주어야 합니다.

tearDownWithError()

```
override func tearDownWithError() throws { // Put teardown code here. This method is called after the invocation of each test method in the class. }
```

`tearDownWithError()`는 각각의 test 실행이 끝난 후마다 호출되는 메서드입니다. 보통 `setUpWithError()`에서 설정한 값들을 해제할 때 사용됩니다.

그러니까 여러 개의 테스트 케이스가 실행되는 경우 `setUpWithError()`와 `tearDownWithError()`는 아래의 그림과 같은 순서로 호출됩니다.



> 🤔 setUp()과 tearDown()라는 메서드도 있던데 무슨 차이인가요?
>
> `setUp()`과 `tearDown()` 메서드와 `setUpWithError()`, `tearDownWithError()`의 차이는 에러를 `throw` 할 수 있느냐에 대한 차이입니다. 테스트 메서드를 `throw` 메서드로 만들어주면, 내부에서 따로 에러 핸들링을 해주는 작업없이 테스트를 작성할 수 있다는 이점이 있습니다! 원래는 `setUp()`과 `tearDown()` 메서드가 기본 메서드로 제공이 되었는데 Xcode 11.4 버전 이후로 기본으로 `setUpWithError()`, `tearDownWithError()`를 제공하고 있어요.
> 이에 대한 내용이 좀 더 내용이 궁금하다면 Understanding Setup and Teardown for Test Methods 문서를 참고해보면 좋을 것 같습니다 :)

testExample()

```
func testExample() throws { // This is an example of a functional t
est case. // Use XCTAssert and related functions to verify your tes
ts produce the correct results. }
```

`test`로 시작하는 메서드들은 우리가 작성해야 할 test case가 되는 메서드입니다. 우리가 테스트할 내용을 메서드로 작성해볼 수 있습니다. 메서드 네이밍의 시작은 **무조건** *test*로 시작되어야 합니다.

> 🤔 보통 프로그래밍의 네이밍은 거의 영어로 해주었지만 특이하게 테스트 함수의 네이밍은 한글로 작성하는 경우도 있어요. 그 이유는 테스트 코드가 기능을 명세화, 문서화 하는 역할도 하기 때문이라고 할 수 있습니다. 한국인들끼리 작업하는 경우라면 영어보다 한글로 테스트를 작성할 때 더 쉽게 눈에 들어올 수도 있죠.

단위 테스트에 대해 좀 더 공부해보고 싶다면 아래 링크를 참고해보세요.

Unit Test 작성하기

새로 Calculator.swift 파일을 생성하고, 생성된 Calculator.swift 파일로 이동하여 `Calculator` 타입을 구현합니다.

```
struct Calculator { func add( lhs: Int,  rhs: Int) -> Int { retur
n lhs + rhs } func substract( lhs: Int,  rhs: Int) -> Int { retur
n lhs + rhs } // 이하 생략... }
```

테스트 코드 작성하기

테스트 코드 파일로 이동하여 테스트 코드를 작성해봅니다. 기존에 템플릿으로 작성되어 있던 `testExample()` 메서드 대신 내가 원하는 테스트 메서드를 작성합니다. `add( : )` 메서드를 통해 2 더하기 3이 5가 된다는 검증을 하고 싶어서 `test2더하기3은5`라는 테스트 메서드를 `SampleTest` 클래스에 만들었습니다.

> 💡 테스트 메서드의 이름은 꼭 `test`로 시작해야합니다.

```
func test2더하기3은5() throws { let calculator: Calculator = Calculat
or() XCTAssert(calculator.add(2, 3) == 5, "2 + 3 = 5 실패") XCTAsser
tTrue(calculator.add(2, 3) == 5, "2 + 3 = 5 실패") XCTAssertEqual(ca
lculator.add(2, 3), 5, "2 + 3 = 5 실패") }
```

테스트를 위한 assertion 메서드는 여러 종류가 있습니다. aseertion이 실패하면 테스트가 중지되며 테스트 결과를 실패로 표기합니다.

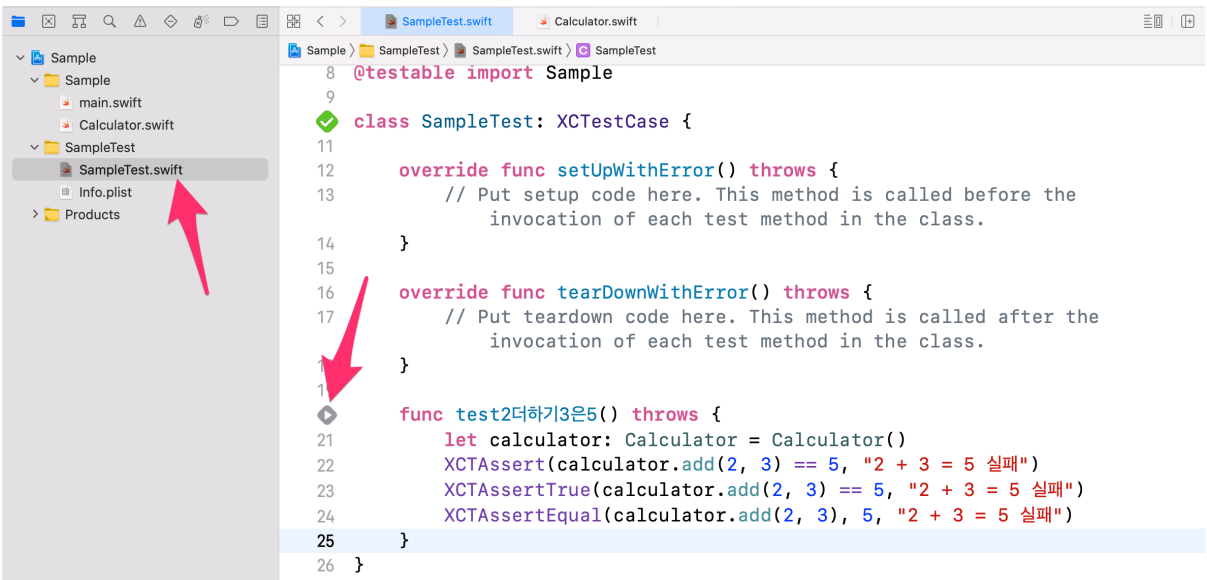
자세한 메서드 종류와 사용방법은 XCTest 문서의 Test Assertions 섹션을 참고해보세요.
https://developer.apple.com/documentation/xctest

대표적인 메서드 이름

- XCTAssert
- XCTAssertTrue
- XCTAssertFalse
- XCTAssertNil
- XCTAssertNotNil
- XCTAssertUnwrap
- XCTAssertEqual
- ... 아주 많아요...

테스트 결과 확인하기

생성한 테스트 메서드의 테스트 결과를 확인하고 싶으면 메서드 왼쪽의 실행 버튼을 클릭하여 테스트를 실행할 수 있습니다.



여러개의 테스트 메서드를 전체적으로 실행하고 싶다면 cmd + u 단축키를 사용하여 전체 테스트를 실행할 수 있습니다.

테스트 실행결과 성공적으로 마치면 녹색 체크로 변경됩니다.

```
    func test2더하기3은5() throws {
        let calculator: Calculator = Calculator()
        XCTAssert(calculator.add(2, 3) == 5, "2 + 3 = 5 실패")
        XCTAssertTrue(calculator.add(2, 3) == 5, "2 + 3 = 5 실패")
        XCTAssertEqual(calculator.add(2, 3), 5, "2 + 3 = 5 실패")
    }
}

```

하나의 테스트 메서드를 추가해 보았습니다.

```
func test큰수에서_작은수를_빼면_항상_양수다() throws { let calculator: Calc
ulator = Calculator() XCTAssertGreaterThan(calculator.substract(5,
3), 0, "5 - 3 > 0 실패") XCTAssertGreaterThan(calculator.substract(-
2, -5), 0, "-2 - (-5) > 0 실패") }
```

어머, 이런 실패했네요. 구현 코드에 문제가 있나봅니다!

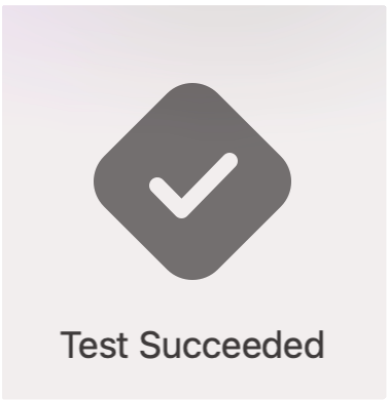
```
func test큰수에서_작은수를_빼면_항상_양수다() throws {
    let calculator: Calculator = Calculator()
    XCTAssertGreaterThan(calculator.substract(5, 3), 0, "5 - 3 > 0 실패")
    XCTAssertGreaterThan(calculator.substract(-2, -5), 0, "-2 - (-5) > 0 실패")  XCTA..
```

테스트 결과를 통해 구현 코드 수정하기

역시 복사 붙여넣기는 코딩의 큰 적이죠? 테스트 코드를 통과하도록 `substract(_:_:)` 메서드를 수정해봅시다.
문제를 찾았나요? 수정하고 다시 테스트 해보니 정상적으로 테스트를 통과했네요. 축하합니다! 🎉

```
func test큰수에서_작은수를_빼면_항상_양수다() throws {
    let calculator: Calculator = Calculator()
    XCTAssertGreaterThan(calculator.substract(5, 3), 0, "5 - 3 > 0 실패")
    XCTAssertGreaterThan(calculator.substract(-2, -5), 0, "-2 - (-5) > 0 실패")
}
```

~~이 테스트 통과 심볼을 만나면 어쩌나 반가운지...~~



여기까지 잘 따라왔다면 아직 테스트하지 않은 계산기 기능(곱셈, 나눗셈)을 모두 테스트 해보세요.

또, 자신이 만들어둔 유틸리티 클래스 등이 있다면 테스트를 통해 예상된 동작이 제대로 이뤄지는 것을 확인해 보는것도 단위테스트 연습에 큰 도움이 됩니다. 내 코드의 품질이 향상되는 것은 말할 필요도 없겠죠? 😉

Code Coverage

구현 코드의 양이 많아지는만큼 테스트 코드도 많아지다보니 제대로 테스트가 이뤄지고 있는지 파악하기 어려울 수 있습니다. 구현 코드가 제대로 테스트되고 있는지 확인하고 싶다면 Code Coverage를 참고할 수 있습니다.

[Edit Scheme...] 메뉴 또는 cmd + shift + ,(쉼표) 단축키를 통해 Scheme 관리 창으로 진입하여 [Test]-[Options] 탭으로 이동합니다. 아래 메뉴 중 Code Coverage 메뉴의 체크박스에 체크한 후 닫아주세요.

> 💡 단위 테스트와 TDD와는 다른 개념입니다. 그것이 다르다는 것을 우선 꼭 명심해주세요.

Test Driven Development(테스트 주도 개발, TDD)

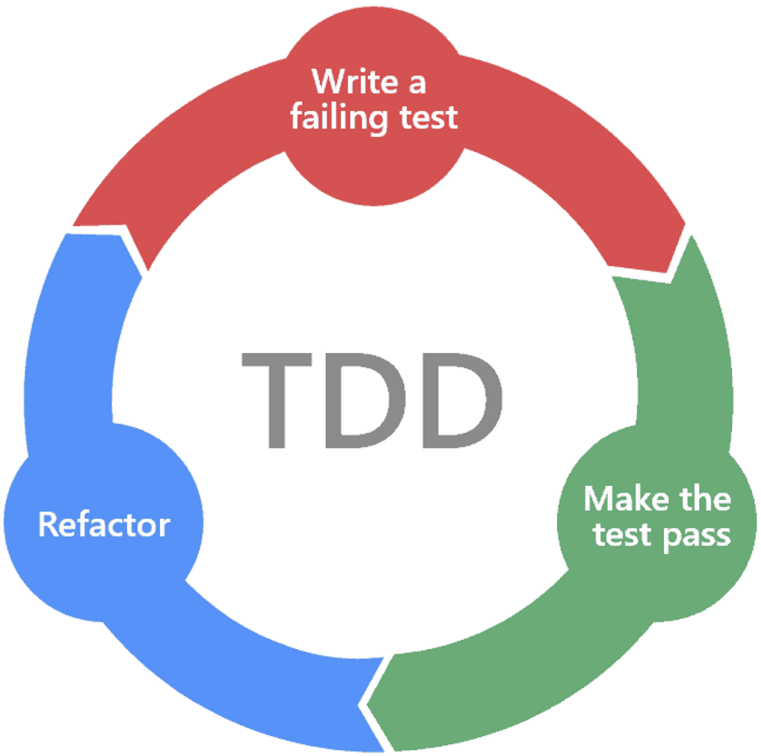
TDD란?

테스트 주도 개발(Test-driven development TDD)은 매우 짧은 개발 사이클을 반복하는 소프트웨어 개발 프로세스 중 하나이다. TDD는 단순한 설계를 장려하고 자신감을 불어 넣어 줄 수 있습니다.

```
TDD = TFD(Test First Development) + 리팩터링
```

프로그래밍 순서

1. 빨강 - 실패하는 작은 테스트를 작성합니다. 처음에는 컴파일조차 되지 않을 수 있습니다.
2. 초록 - 빨리 테스트가 통과하게끔 만듭니다. 이를 위해 어떤 죄악을 저질러도 좋습니다.
3. 리팩터링 - 일단 테스트를 통과하게만 하는 와중에 생겨난 모든 중복을 제거합니다.



> 죄악이란 기존 코드를 복사해서 붙이기 뿐만 아니라 테스트만 간신히 통과 할 수 있게끔 하드 코딩으로 구현하는 것 등을 의미합니다.

원칙

1. 실패하는 단위 테스트를 작성할 때까지 구현 코드(production code)를 작성하지 않습니다.
2. 컴파일은 실패하지 않으면서 실행이 실패하는 정도로만 단위 테스트를 작성합니다.
3. 현재 실패하는 테스트를 통과할 정도로만 실제 코드를 작성합니다.

용기

테스트 주도 개발은 프로그래밍하면서 나타나는 두려움을 관리하는 방법입니다.

- 두려움은 여러분을 망설이게 만듭니다.
- 두려움은 여러분이 커뮤니케이션을 덜 하게 만듭니다.

- 두려움은 여러분이 피드백 받는 것을 피하도록 만듭니다.
- 두려움은 여러분을 까다롭게 만듭니다.

이 중 어떠한 것도 프로그래밍에 도움이 되지 않습니다.

- 불확실한 상태로 있는 대신, 가능하면 재빨리 구체적인 학습을 하는것이 좋습니다.
- 침묵을 지키는 대신, 좀더 분명하게 커뮤니케이션합시다.
- 피드백을 회피하는 대신, 도움이 되고 구체적인 피드백을 찾으세요.
- 불안함에 점점 나빠지는 자신의 성격을 어떻게 고쳐볼 수 있을까요?

TDD로 자동차 경주 구현

기능 요구 사항

- 각 자동차에 이름을 부여할 수 있습니다. 전진하는 자동차를 출력할 때 자동차 이름을 같이 출력합니다.
- 자동차 이름은 쉼표(,)를 기준으로 구분합니다.
- 자동차 경주 게임을 완료한 후 누가 우승했는지를 알려줍니다. 우승자는 한 명 이상일 수 있습니다.

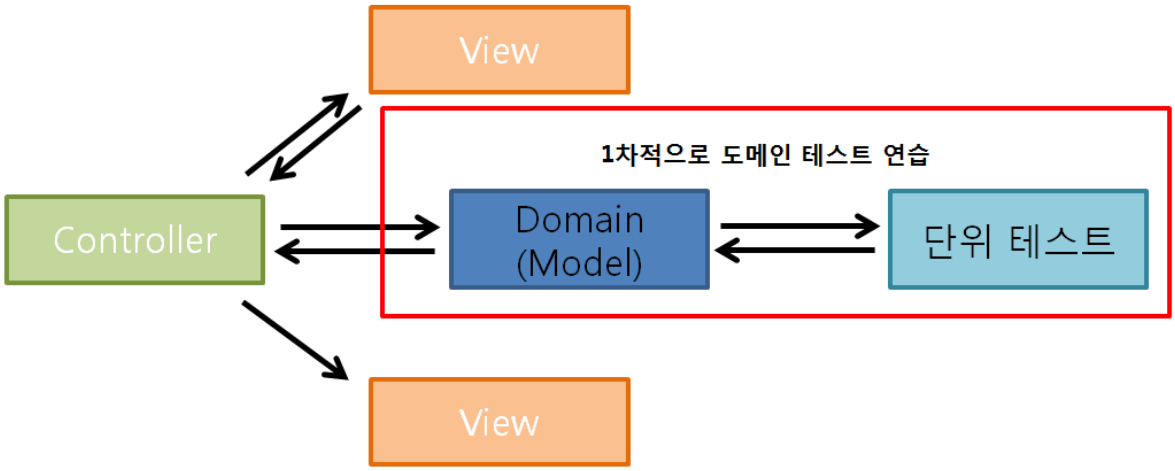
단위 테스트, TDD를 처음 시작할 때의 감정은 어떤가요?

'어디서, 어떻게 시작해야 할지 모르겠다...' 막막하죠?
그렇다면 같이 아래 흐름을 따라가봅시다.

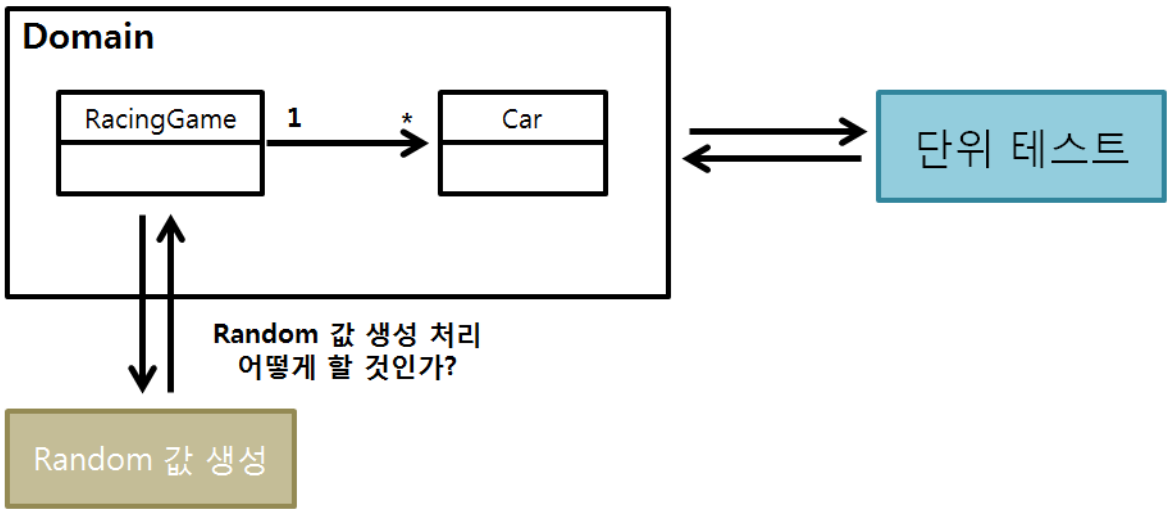
요구 사항 분석 및 설계

- 요구 사항 분석을 통해 대략적인 설계를 해봅니다 - 인스턴스는 어떤 일들을 하고있나요?
- UI, DB 등과 의존 관계를 있지 않는 핵심 도메인 영역을 집중 설계해봅니다

- 일차적으로는 도메인 로직을 테스트하는 것에 집중해보세요



대략적인 도메인 흐름의 설계



구현할 기능 목록 작성하기

- 기능 목록을 작성하는 것도 역량이 필요하다고요?
- 역량도 중요하지만 연습도 많이 필요합니다.

그래도 막막하다면...

- 단위 테스트도 없고, TDD도 아니고, 설계도 하지 않고, 기능 목록을 분리하지도 않고 일단 구현해봅니다.
- 구현하려는 프로그램의 도메인 지식을 습득합니다.
- 구현한 모든 코드를 버립니다.
- 구현할 기능 목록을 작성하거나 간단한 도메인을 설계합니다.
- 기능 목록 중 가장 만만한 녀석부터 TDD로 구현을 시작합니다.

- 복잡도가 높아져 리팩토링하기 힘든 상태가 되면 다시 버립니다.
- 다시 도전합니다.

아무 것도 없는 상태에서 새롭게 구현하는 것보다 레거시 코드가 있는 상태에서 리팩토링하는 것이 몇 배 더 어렵습니다.

TDD로 자동차 경주게임 구현

일단 구현

- 지금까지 자신에게 익숙한 방법으로 일단 구현해보세요.

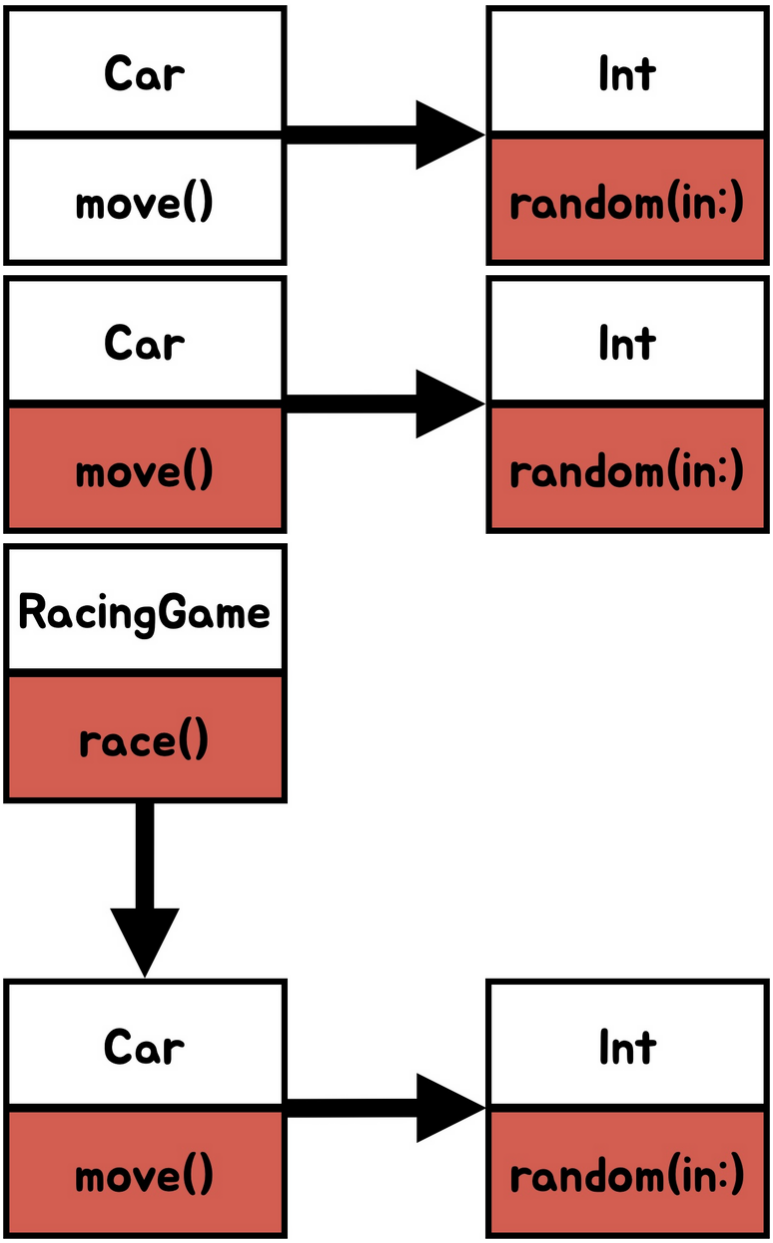
테스트 가능한 부분을 찾아 단위 테스트

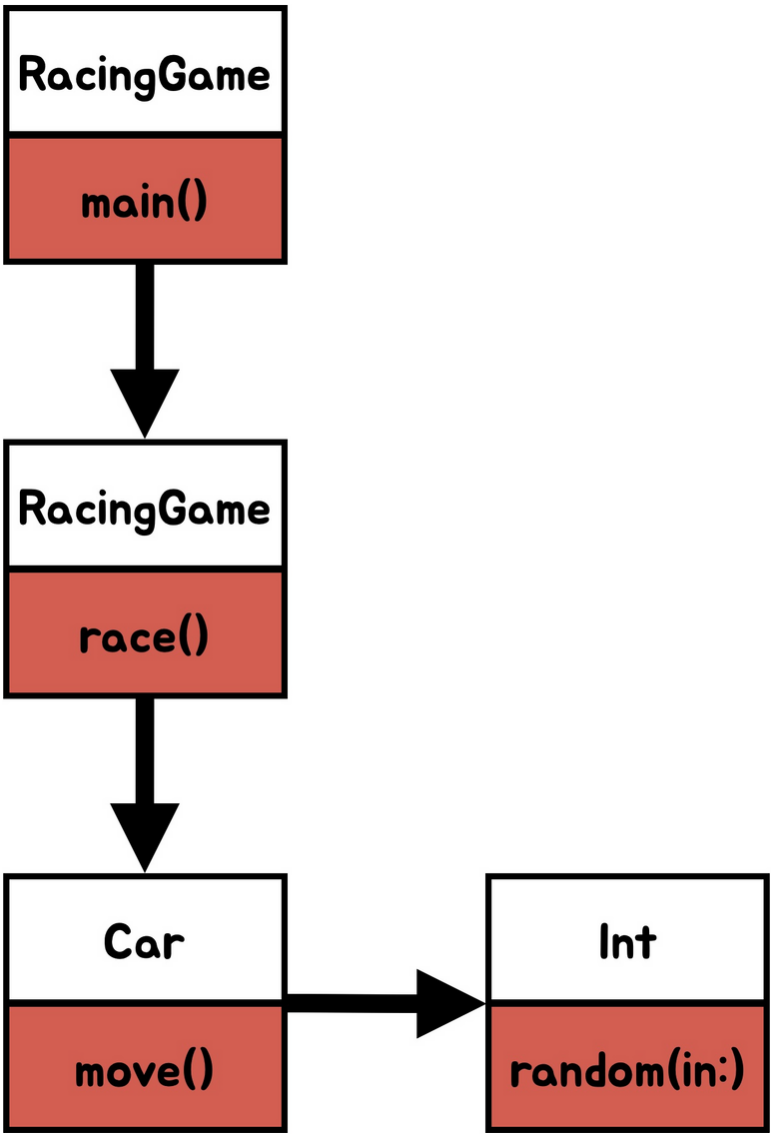
- 입력한 문자열에서 참여자의 이름을 추출하여 테스트합니다.
- 자동차 이동 유무를 테스트합니다.
- 자동차 이동 거리에 따라 생성한 "-"를 테스트합니다.

테스트하기 어려운 부분을 찾아 가능한 구조로 개선

- 타입 관계 그래프에서 다른 타입과 의존관계를 가지는 않는 마지막 노드(Node)를 먼저 찾아보세요.
- 예를 들어 `RacingMain` -> `RacingGame` -> `Car`와 같이 의존 관계를 가진다면 `Car`가 테스트 가능한지 확인해봅니다.

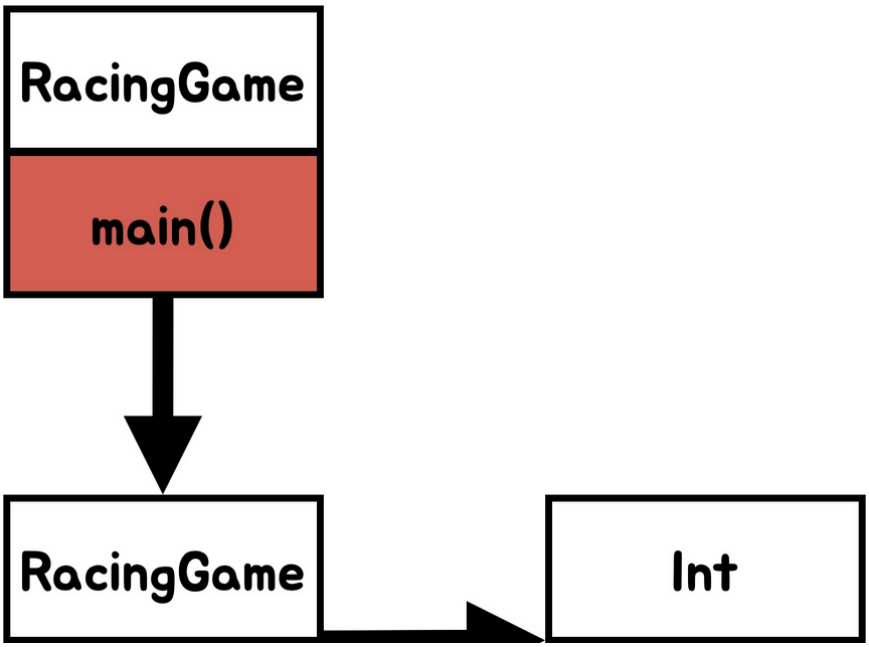
```
class Car { private enum Constants { static let defaultPosition: In
t = 0 static let minimunMovementDistance: Int = 4 static let random
NumberRange: ClosedRange = 0...9 } let owner: String private(set) v
ar position: Int init(owner: String, position: Int = Constants.defa
ultPosition) { self.owner = owner self.position = position } privat
e var randomNumber: Int { Int.random(in: Constants.randomNumberRang
e) } func move() { let movingDistance: Int = randomNumber if moving
Distance >= Constants.minimunMovementDistance { position += movingD
istance } } }
```


Car
move()
Int
random(in:)
Car
move()
Int
random(in:)
RacingGame
race()
Car
move()
Int
random(in:)



해결책

테스트하기 어려운 코드의 의존관계를 관계 그래프의 상위로 이동시킵니다.


RacingGame
main()
RacingGame
Int