Prefer annotations to naming patterns

naming pattens는 레일즈처럼 이름을 기준으로 특정 작업을 하는 것을 의미합니다.

단점으로는 다음과 같습니다.

명명패턴 단점

1. 오타가 나오면 안됩니다.

2. 사용자가 올바른 프로그램 요소에서만 사용하리라 보장할 수 없습니다.

예를 들어 Junit 3의 경우 method에 test이름을 지닌 것만을 실행합니다. 그런데 사용자는 class 내 전체 메서드를 모두 테스트해주길 바라는 마음으로 class 이름 앞에 Test를 붙일 수 있죠. 이때 Junit은 경고를 출력하지도 않고, 테스트하지도 않습니다.

3. 프로그램 요소를 매개변수로 전달할 방법이 없습니다.

기대하는 예외가 발생하는지 확인하고 싶지만 할 도리가 없죠.

애너테이션(Annotation)

애너테이션은 위 문제를 모두 해결할 수 있으며, Junit4에서부터 애너테이션이 본격 적용되었습니다.

이번 글에선 토이 테스트 프레임워크를 만들어서 애너테이션 동작 원리 및 사용법에 대해 말씀드리도록 하겠습니다.

Test라는 이름의 애너테이션을 정의할 것이며, Exception을 던질 때 실패로 처리하도록 합니다.

```
// Marker annotation type declaration
import java.lang.annotation.*;
/**
    * Indicates that the annotated method is a test method.
    * Use only on parameterless static methods.
    */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

@Retention과 @Target

테스트 애너테이션 위에 또 애너테이션이 있습니다.이 애너테이션을 meta-aanotations라고 부릅니다.

@Retention(RetentionPolicy.RUNTIME)

@Test가 런타임에도 유지되어야 한다는 표시입니다.

@Target(ElementType.METHOD)

Test는 반드시 메서드 선언에서만 사용되어야 한다는 표시입니다.

코드 주석

앞 코드 주석에는

```
/**
    * Indicates that the annotated method is a test method.
    * Use only on parameterless static methods.
    */
```

매개변수 없는 정적메서드 용이라고 되어있지만, 이 코드에선 강제하는 수단을 적용하지 않았습니다. 이를 적용하려면 `javax.annotation.processing API`를 참고하시면 됩니다.

강제하는 수단을 적용하지 않았기 때문에 이 코드는 컴파일은 잘 되지만 test framework가 실행될 때 문제가 됩니다.

실제 적용

```
// Program containing marker annotations
public class Sample {
  @Test public static void m1() { } // Test should pass public static void m2()
{ }
  @Test public static void m3() { // Test should fail
    throw new RuntimeException("Boom");
  }
  public static void m4() { }
  @Test public void m5() { } // INVALID USE: nonstatic method public static
void m6() { }
  @Test public static void m7() { // Test should fail
    throw new RuntimeException("Crash");
  }
  public static void m8() { }
}
```

@Test같은 매개변수는 아무 매개 변수없이 단순히 마킹한다는 의미로 marker annotation이라고 부릅니다.

그리고 애너테이션 이름을 잘못 작성했거나, static 이외 메서드에 달았다면 컴파일 에러를 발생시킵니다.

> 아마 javax.annotation.proccessing API를 적용했다 가정하고 예시를 설명하는 것 같습니다.

이 코드에선 단순히 @Test는 표시만 한 것이며, 실제 처리는 @Test 애너테이션에 관심있는 프로그램이 따로 처리할 것 입니다.

Market annotation processor

```
// Program to process marker annotations
import java.lang.reflect .*;

public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " failed: " + exc);
                } catch (Exception exc) {
                    System.out.println("Invalid @Test: " + m);
```

```
                }
            }
        }
        System.out.printf("Passed: %d, Failed: %d%n",
                passed, tests - passed);
    }
}
```

1. test runner tool은 command line으로부터 fully qualified class name을 받습니다.

   (`Class<?> testClass = Class.forName(args[0]);`)

2. 그리고 그 클래스에서 test annotation이 달린 메서드를 찾아주죠.

```
for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
```

3. 테스트 메서드가 예외를 던지면 리플렉션 매커니즘이 예외를 감싸서 다시 던집니다.

   (`InvocationTargetException`)

   만약 그 이외의 예외가 발생했다면 Test Annotation을 잘못사용했다는 뜻입니다. 따라서 이는 두번 째 catch가 Exception 타입을 잡도록 해둡니다.

출력화면

```
public static void Sample.m3() failed: RuntimeException: Boom
Invalid @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Passed: 1, Failed: 3
```

특정 예외를 던져야만 성공하는 예

```
// Annotation type with a parameter
import java.lang.annotation.*;
/**
    * Indicates that the annotated method is a test method that
    * must throw the designated exception to succeed.
    */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
  Class<? extends Throwable> value();
}
```

이 애너테이션의 매개변수 타입은 Class<? extends Throwable>이 됩니다.

```
// Program containing annotations with a parameter
public class Sample2 {
  @ExceptionTest(ArithmeticException.class)
  public static void m1() {  // Test should pass
    int i = 0;
    i = i / i;
  }
  @ExceptionTest(ArithmeticException.class)
  public static void m2() {  // Should fail (wrong exception)
    int[] a = new int[0];
    int i = a[1];
  }
  @ExceptionTest(ArithmeticException.class)
  public static void m3() { }  // Should fail (no exception)
}
```

테스트 도구 수정

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
  tests++;
  try {
    m.invoke(null);
    System.out.printf("Test %s failed: no exception%n", m);
  } catch (InvocationTargetException wrappedEx) {
    Throwable exc = wrappedEx.getCause();
    Class<? extends Throwable> excType =
      m.getAnnotation(ExceptionTest.class).value();
    if (excType.isInstance(exc)) {
```

```
        passed++;
      } else {
        System.out.printf(
          "Test %s failed: expected %s, got %s%n",
          m, excType.getName(), exc);
      }
    } catch (Exception exc) {
      System.out.println("Invalid @Test: " + m);
    }
  }
```

앞 코드와 차이점은 매개변수 값을 추출해서 해당 Exception 타입인지 확인하는 코드가 들어있습니다.

```
Class<? extends Throwable> excType =
  m.getAnnotation(ExceptionTest.class).value();
if (excType.isInstance(exc)) {
  passed++;
```

예외 클래스 파일이 컴파일 타임에는 존재했지만, 런타임에는 없으면 테스트러너가 TypeNotPresentException을 던질수도 있습니다.

특정 예외들을 성공시키는 예

허용 시키는 예외의 나열이 필요할 수 있잖아요? Annotation 파라미터를 배열로 받는 예제를 보여드리겠습니다.

```
// Annotation type with an array parameter
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
  Class<? extends Exception>[] value();
}
```

직관적이게도 value()의 반환 타입을 배열로 선언 해주면 됩니다.

그리고 단일 파라미터로 받았던 애너테이션을 수정할 필요도 없죠.

```
// Code containing an annotation with an array parameter
@ExceptionTest({ IndexOutOfBoundsException.class, NullPointerException.class })
public static void doublyBad() {
       List<String> list = new ArrayList<>();

       // The spec permits this method to throw either
       // IndexOutOfBoundsException or NullPointerException
       list.addAll(5, null);
}
```

표시는 중괄호로 표시한 후 argument들을 작성해주면 됩니다.

테스트러너 수정 코드

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
  tests++;
  try {
    m.invoke(null);
    System.out.printf("Test %s failed: no exception%n", m);
  } catch (Throwable wrappedExc) {
    Throwable exc = wrappedExc.getCause();
    int oldPassed = passed;
    Class<? extends Exception>[] excTypes =
      m.getAnnotation(ExceptionTest.class).value();
    for (Class<? extends Exception> excType : excTypes) {
      if (excType.isInstance(exc)) {
        passed++;
        break;
      }
    }
    if (passed == oldPassed)
      System.out.printf("Test %s failed: %s %n", m, exc);
  }
}
```

Repeatable Annotation

Java 8에선 여러 개의 값을 받는 애너테이션을 `@Repeatable`로 처리할 수 있게 하였습니다.

따라서 해당 애너테이션이 달린 애너테이션은 하나의 프로그램 요소에 여러번 달릴 수 있습니다.

하지만, 주의할 점은 다음과 같죠.

1. `@Repeatble`을 단 애너테잇녀을 반환하는 컨테이너 애너테이션을 하나 더 정의하고 `@Repeatable`에 이 컨테이너 애너테이션의 class객체를 매개변수로 전달해야 한다.
2. 컨테이너 애너테이션은 내부 애너테이션 타입의 배열을 반환하는 value 메서드를 정의해야 한다.
3. 적절한 `@Retentation`과 `@Target`을 명시해야 한다.

이를 지키지 않으면 컴파일 되지 않습니다.

Repeatable annotation type 코드

```
// Repeatable annotation type
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(ExceptionTestContainer.class)
public @interface ExceptionTest {
  Class<? extends Exception> value();
}
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTestContainer {
  ExceptionTest[] value();
}
```

사용자 코드

```
// Code containing a repeated annotation
@ExceptionTest(IndexOutOfBoundsException.class)
@ExceptionTest(NullPointerException.class)
public static void doublyBad() { ... }
```

처리코드

```
// Processing repeatable annotations
if (m.isAnnotationPresent(ExceptionTest.class)
    || m.isAnnotationPresent(ExceptionTestContainer.class)) {
  tests++;
```

```
    try {
      m.invoke(null);
      System.out.printf("Test %s failed: no exception%n", m);
    } catch (Throwable wrappedExc) {
      Throwable exc = wrappedExc.getCause();
      int oldPassed = passed;
      ExceptionTest[] excTests =
        m.getAnnotationsByType(ExceptionTest.class);
      for (ExceptionTest excTest : excTests) {
        if (excTest.value().isInstance(exc)) {
          passed++;
          break;
        }
      }
      if (passed == oldPassed)
        System.out.printf("Test %s failed: %s %n", m, exc);
    }
  }
```

주의사항은 반복가능 애너테이션을 여러 개 달면 하나만 달았을 때와 구분하기 위해 컨테이너 애너테이션 타입이 적용됩니다.

getAnnotationByType은 둘을 구분하지 않아 둘 다 가져오는 반면,

isAnnotationPresent는 이 둘을 구분합니다. 따라서 or 연산자로 ExceptionTest.class인지, ExceptionTestContainer.class인지 따져봐야 합니다.

이처럼 코드 가독성을 높히는데 `@Repeatable`을 사용할 수 있지만, 애너테이션을 선언하는 부분 그리고 처리하는 부분에서는 코드가 복잡해지니 잘 고민하고 사용하시기 바랍니다.

정리

애너테이션으로 처리할 수 있는 일을 명명패턴으로 처리할 필요는 없습니다.

그리고, framework 개발자를 제외하고는 애너테이션을 작성할 일은 거의 없으나, 자바 프로그래머라면 예외없이 자바가 제공하는 애너테이션 타입을 사용해야 하므로 작동 방법을 숙지하시면 좋겠습니다.