Dynamic testing

Tools and Techniques for Software Testing - Barbara Russo

SwSE - Software and Systems Engineering group

unibz

Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Dynamic testing

- **Dynamic testing** concerns testing the **operations (behaviour)** of a program

- Unit tests, integration tests, system tests, acceptance tests and regression tests utilize dynamic testing

Dynamic testing with Unit testing

- Tests with @Test annotation are static tests as they are fully specified at compile-time
- A dynamic test is a test generated during run-time

- In JUnit 5
  - There are new annotations that support it
  - DynamicTest class that provide suitable methods

- Let’s review them!

Dynamic testing

- A dynamic test is generated by a factory annotation: **@TestFactory**
- The annotation identifies a factory method whose goal is to build instances of the DynamicTest class
- @TestFactory methods *must not be private or static* and may optionally declare parameters

DynamicTest class

- DynamicTest is a test case generated at runtime
- Its method *dynamicTest* takes as parameters
  - a *display name* and an *executable*

- Instances of DynamicTest must be **generated by factory methods annotated with @TestFactory**

Example

Factory method

```
@TestFactory
public DynamicTest createTest(){

   return dynamicTest("1st dynamic test", () -> assertTrue(isPalindrome("madam")));

}
```

display name

executable

Key methods of DynamicTest class

public static DynamicTest dynamicTest(String displayName, Executable executable)

- Factory method creates a new DynamicTest instance with the given display name and executable code block

```
dynamicTest("testName", () -> assertTrue(isPalindrome(“madam")))
```

uh! anonymous Lambda expression

Example

Factory method

```
@TestFactory
Collection<DynamicTest> dynamicTestsFromCollection(){

return Arrays.asList(
        dynamicTest("1st dynamic test", () -> assertTrue(isPalindrome("madam"))),
        dynamicTest("2nd dynamic test", () -> assertEquals(4, calculator.multiply(2, 2)))
    );

}
```

Key methods of DynamicTest class

The stream method

```
static <T> Stream<DynamicTest>
stream(Iterator<T> inputGenerator, Function<? super T,String> displayNameGenerator,
ThrowingConsumer<? super T> testExecutor)
```

- Factory method to generate **a stream of dynamic tests** based on the generators and test executor
- inputGenerator generates input values. A DynamicTest is added to the resulting stream for each dynamically generated input value, using the displayNameGenerator and testExecutor.
- `inputGenerator` - an Iterator that serves as a dynamic input generator
- `displayNameGenerator` - a function that generates a display name based on an input value
- `testExecutor` - a consumer that executes a test based on an input value

Review of terms

- What is Functional Interface?
- What is Consumer/Supplier?
- What is Lambda expression?
- What is Function?

Interfaces

Interfaces

Which types of interface do you know?

Interfaces

- **Marker interface** is an **empty** interface

Which types of interface do you know?

Interfaces

- **Marker interface** is an **empty** interface
- **Functional Interface** is an interface with **just one abstract method** declared in it

Which types of interface do you know?

Interfaces

- **Marker interface** is an **empty** interface
- **Functional Interface** is an interface with **just one abstract method** declared in it
- **Regular Interface**

Which types of interface do you know?

Functional Interface

- A functional interface has **only one abstract method** but it can have *multiple default methods*
- **@FunctionalInterface** annotation is used to ensure **at compile time** that an interface cannot have more than one abstract method
  - The use of this annotation is optional

Functional Interface

- Example: Runnable interface has only run() method
- Lambda expression works on functional interfaces to replace anonymous classes

Anonymous class

```java
interface Rectangle {}

class anonymousClassExamples {
    public static void main(String args[]) {
        int a = 5;
        int b = 7;
         //anomymous class
        int ans1 = new Rectangle(){
            public int calculate(int x, int y){return x*y;}
        }.calculate(a,b);

       System.out.println(ans1);
    }
}
```

Lambda expression

- An anonymous function that can be passed around as a variable or as a parameter to a method call

Sintax

```
lambda operator -> body
```

where lambda operator can be:

- **Zero parameter:**

```
() -> System.out.println("Zero parameter lambda”);
```

- **One parameter:**

```
(p) -> System.out.println("One parameter: " + p);
```

- **Multiple parameters :**

```
(p1, p2) -> System.out.println("Multiple parameters: " +
                                p1 + ", " + p2);
```

Example

```
@FunctionalInterface
interface Square {
    int calculate(int x);
}
class Test {
    public static void main(String args[]) {
        int a = 5;
        // lambda expression to define the calculate method
        Square s = (int x)->x*x;
        // parameter passed and return type must be same as
defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }
}
```

Example

```
@FunctionalInterface
interface Square {
    int calculate(int x);
}
class Test {
    public static void main(String args[]) {
        int a = 5;
        // lambda expression to define the calculate method
        Square s = (int x)->x*x;
        // parameter passed and return type must be same as
defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }
}
```

Lambda expression to create an instance of a class and define the method of the Functional Interface

Exercise

A method that prints the area of a rectangle

Exercise

```
@FunctionalInterface
interface Area {
    int calculate(int x, int y);;
}
class Test {
    public static void main(String args[]) {
        int a = 5;
        int b = 7;
        // lambda expression to define the calculate method
        Area s = (int x, int y)->x*y;
        // parameter passed and return type must be same as defined
in the prototype
        int ans = s.calculate(a,b);
        System.out.println(ans);
    }
}
```

Consumer/Supplier

- Functional interface (package function)
- **Consumer**: represents an operation that *accepts a single input argument and returns no result*
- **Supplier:** represents an operation that *accepts no input and returns a result*
- **Function:** represents an operation that *accepts input and returns a result*

Functional interfaces with JUnit5

org.junit.jupiter.api.function

- **Executable:** used to implement any generic block of code that potentially throws a Throwable
- **ThrowingConsumer<T>:** a consumer that potentially throws a Throwable
- **ThrowingSupplier<T>:** a supplier that potentially throws a Throwable

FunctionalInterfaces w. Dynamic Test

```
stream(
Iterator<T> inputGenerator,
Function<? super T,String> displayNameGenerator,
ThrowingConsumer<? super T> testExecutor)
dynamicTest(String displayName, Executable executable)
```

- The implementations of DynamicTest can be provided as lambda expressions or method references for the Functional Interfaces

Function faces w. Dynamic Test

Regular Interface

```
stream(
Iterator<T> inputGenerator,
Function<? super T,String> displayNameGenerator,
ThrowingConsumer<? super T> testExecutor)
dynamicTest(String displayName, Executable executable)
```

- The implementations of DynamicTest can be provided as lambda expressions or method references for the Functional Interfaces

Function fac Dynamic Test

Regular Interface

Functional Interfaces

```
stream(
Iterator<T> inputGenerator,
Function<? super T,String> displayNameGenerator,
ThrowingConsumer<? super T> testExecutor)
dynamicTest(String displayName, Executable executable)
```

- The implementations of DynamicTest can be provided as lambda expressions or method references for the Functional Interfaces

Functional Interfaces of Dynamic Test

```
stream(
Iterator<T> inputGenerator,
Function<? super T,String> displayNameGenerator,
ThrowingConsumer<? super T> testExecutor)
dynamicTest(String displayName, Executable executable)
```

Regular Interface

Functional Interfaces

Functional Interfaces of JUnit5

- The implementations of DynamicTest can be provided as lambda expressions or method references for the Functional Interfaces

Return type of factory methods of DynamicTest

- A *single DynamicNode* or a Stream, Collection, Iterable, Iterator, or array of *DynamicNode instances*
- Instantiable subclasses of DynamicNode are **DynamicContainer** and **DynamicTest**

DynamicContainer and DynamicTest

- DynamicContainer instances are composed of a *display name* and a list of *dynamic child nodes*, enabling the creation of arbitrarily nested hierarchies of dynamic nodes
- DynamicTest instances enable dynamic and even non-deterministic *generation of tests*

Note

- Any Stream returned by a @TestFactory will be properly closed by calling stream.close(), making it safe to use a resource such as Files.lines()

Example 1

```
@TestFactory
List<String> dynamicUnitTest() {
    return Arrays.asList(“Hello”);
}
```

This method returns an **invalid return type**. Since an invalid return type cannot be detected at compile time, a JUnitException is thrown when it is detected at runtime

Generation of Dynamic Tests

```
@TestFactory
    Collection<DynamicTest> dynamicTestsFromCollection() {
        return Arrays.asList(
            dynamicTest("1st dynamic test", () -> assertTrue(isPalindrome("madam"))),
            dynamicTest("2nd dynamic test", () -> assertEquals(4, calculator.multiply(2, 2)))
        );
    }

@TestFactory
    Iterable<DynamicTest> dynamicTestsFromIterable() {
        return Arrays.asList(
            dynamicTest("3rd dynamic test", () -> assertTrue(isPalindrome("madam"))),
            dynamicTest("4th dynamic test", () -> assertEquals(4, calculator.multiply(2, 2)))
        );
    }
```

```java
@TestFactory
 Stream<DynamicTest> generateRandomNumberOfTests() {


     Iterator<Integer> inputGenerator = new Iterator<Integer>() {

         Random random = new Random();
         int current;

         @Override
         public boolean hasNext() {
             current = random.nextInt(100);
             return current % 7 != 0;
         }

         @Override
         public Integer next() {
             return current;
         }
     };

     // Generates display names like: input:5, input:37, input:85, etc.
     Function<Integer, String> displayNameGenerator = (input) -> "input:" + input;

     // Executes tests based on the current input value.
     ThrowingConsumer<Integer> testExecutor = (input) -> assertTrue(input % 7 != 0);

     // Returns a stream of dynamic tests.
     return DynamicTest.stream(inputGenerator, displayNameGenerator, testExecutor);
 }
```

Generates random positive integers between 0 and 100 until a number evenly divisible by 7 is encountered.

Fails if input divides by 7

Functional Interfaces. This one is throwing a Throwable exception

Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Exercise

- Generate a stream of dynamic test for the method placeBid(float AuctionID, double price)

Notes

- @BeforeEach and @AfterEach methods are not executed for dynamic tests

- More examples
- https://github.com/junit-team/junit5/blob/master/documentation/src/test/java/example/DynamicTestsDemo.java

Differences - examples

```
//some imports

@RunWith(JUnitPlatform.class)
public class TranslatorEngineTest {
  private TranslatorEngine translatorEngine;
  @BeforeEach
  public void setUp() {
    translatorEngine = new TranslatorEngine();
  }

  @Test
  public void testTranslateHello() {
    assertEquals("Bonjour",
translatorEngine.translate("Hello"));
  }
  @Test
  public void testTranslateYes() {
    assertEquals("Oui",
translatorEngine.translate("Yes"));
  }
@Test
  public void testTranslateYes() {
    assertEquals(“Non",
translatorEngine.translate(“No"));
  }
}
```

One can also use the Parametrised runner!

BeforeEach does not work with TestFActory

```
//some imports and class declaration

@TestFactory
public Collection<DynamicTest> translateDynamicTests() {
      List<String> inPhrases = new
ArrayList<>(Arrays.asList("Hello", "Yes", “No"));
      List<String> outPhrases = new
ArrayList<>(Arrays.asList("Bonjour", "Oui", "Non"));
      Collection<DynamicTest> dynamicTests = new ArrayList<>();
      TranslatorEngine translatorEngine = new TranslatorEngine();

      for (int i = 0; i < inPhrases.size(); i++) {
            String phr = inPhrases.get(i);
            String outPhr = outPhrases.get(i);
            // create a test execution
            Executable exec = () -> assertEquals(outPhr,
translatorEngine.translate(phr));
            // create a test display name
            String testName = " Test translate " + phr;
            // create dynamic test
            DynamicTest dTest = DynamicTest.dynamicTest(testName,
exec);
            // add the dynamic test to collection
            dynamicTests.add(dTest);
      }
      return dynamicTests;
}
```

Exercise

- Use the parametrized runner

Exercise

```
@RunWith(Parameterized.class)
public static class Example{
    @Parameters(name = "{index}: translation({0})={1}")
    public static Object[][] data(){
        return new Object[][]{{"Hello", "Bonjour"},{"Yes","Oui"},
{"No","Non"}};
    }
    private String input;
    private String output;
    public Example(String input, String output){
        this.input=input;
        this.output=output;
    }
    @org.junit.Test
    public void test(){
        TranslatorEngine translatorEngine = new TranslatorEngine();
        assertEquals(output, translatorEngine.translate(input));
    }
```

TestFactory vs. Test

```
@TestFactory
DynamicTest generateDynamicTest() {
    return DynamicTest.dynamicTest(
         "2 + 2 = 4",
         () -> assertEquals(4, 2 + 2,
         "the world is burning”)
         );
}
```

```
@Test
@DisplayName("2 + 2 = 4")
void testMath() {
       assertEquals(4, 2 + 2, "the
       world is burning");
}
```

- Difference between @ParameterizedTest and @TestFactory

- A parameterized test goes through the normal lifecycle for each invocation
- With a test factory, the entire test can be dynamically generated, not just the parameters
- With a test factory, tests are created at runtime; instead a parameterized test already exists, and different parameters for each invocation are just provided