

11

안전한 프로그래밍

차례

- **소프트웨어 안전성 개요**
- 수동 테스트
- 자동 테스트 케이스 생성

소프트웨어 오류



다양한 소프트웨어 오류 종류

- 안전성 오류 (safety error)
 - 0으로 나누기 (divide-by-zero)
 - 잘못된 배열 접근 (array-out-of-bounds error)
 - 정수 훌러넘침 (integer overflow)
 - 널 접근 (null dereference)
 - 자원 누수 (resource leak) 등...
- 기능성 오류 (functionality error)
 - 바람직한 불변식 위반 (invariant violation)
 - 성능 저하 오류 (performance bug)
 - 원치 않는 무한 루프 (infinite loop) 등 ...

소프트웨어 오류 피해 사례

- 아리안 로켓 5 폭발
 - \$1억 손해, 아리안 프로그램 몇년 후퇴
 - https://youtu.be/PK_yguLapgA?t=50s
- 정수 넘침 에러 (integer overflow)
 - 64비트 실수 (double) 타입 변수를 16비트 정수로 안전하지 못한 방식으로 타입 변환 -> 오버플로우!
 - 로켓의 onboard 컴퓨터에 진행방향을 바꾸라는 신호로 잘못 인식
 - <https://around.com/ariane.html>

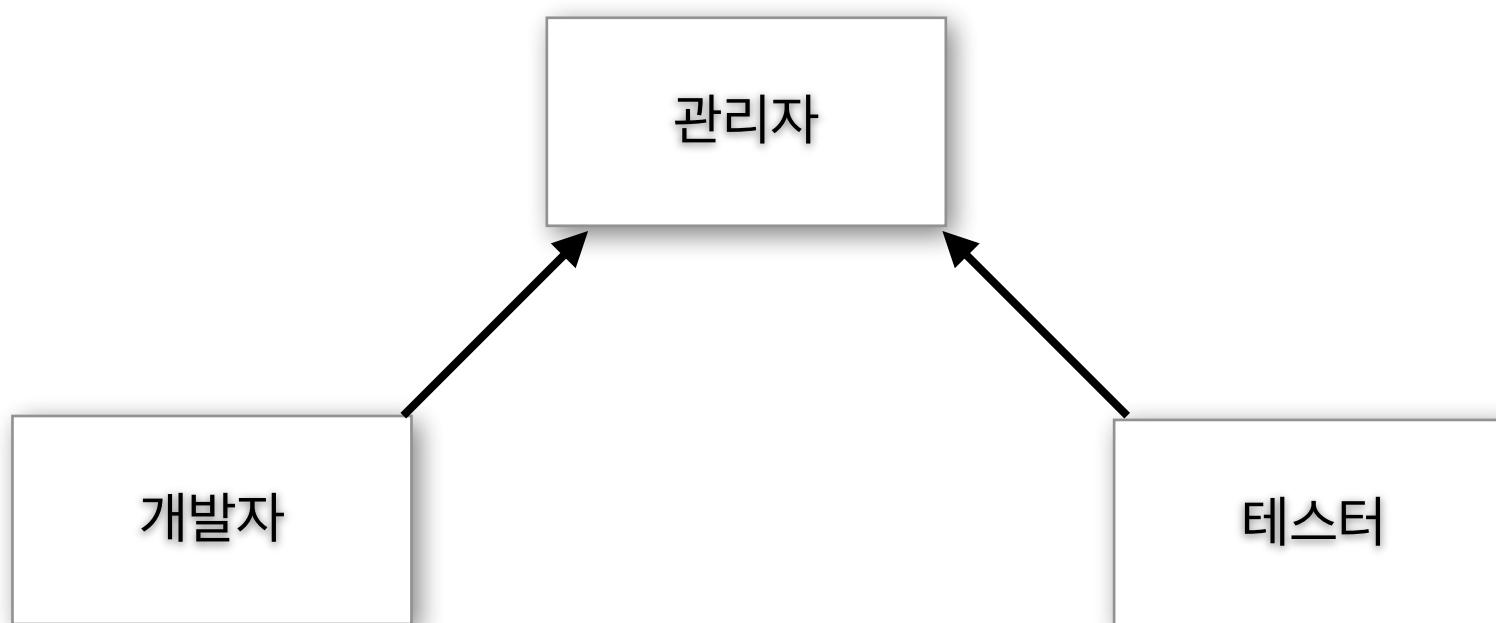
보안 취약점 (Security Vulnerabilities)

- 프로그램에 존재하는 에러를 악용
- 다양한 나쁜 웨어들 (malware)
 - Moonlight Maze (1998)
 - Code Red (2001)
 - Titan Rain (2003)
 - Stuxnet Worm (2010)
- 악성 스마트 폰 앱
- 점점 더 위험하고 많아짐

소프트웨어 개발 보안(Secure Coding)을 위한 국내 노력

- SW 개발 과정에서 지켜야 할 일련의 보안활동
 - 소스코드에 존재할 수 있는 잠재적 보안 취약점을 제거
 - 보안을 고려하여 기능을 설계 및 구현
- SW 개발 시 보안 취약점을 악용한 해킹 등内外부 공격으로부터 시스템을 안전하게 방어할 수 있도록 코딩
- 행정안전부 2012년 5월 시큐어 코딩 의무화 법안: 개발비 40억원 이상 정보화 사업에 시큐어 코딩을 위한 가이드라인을 따르는 것을 의무화

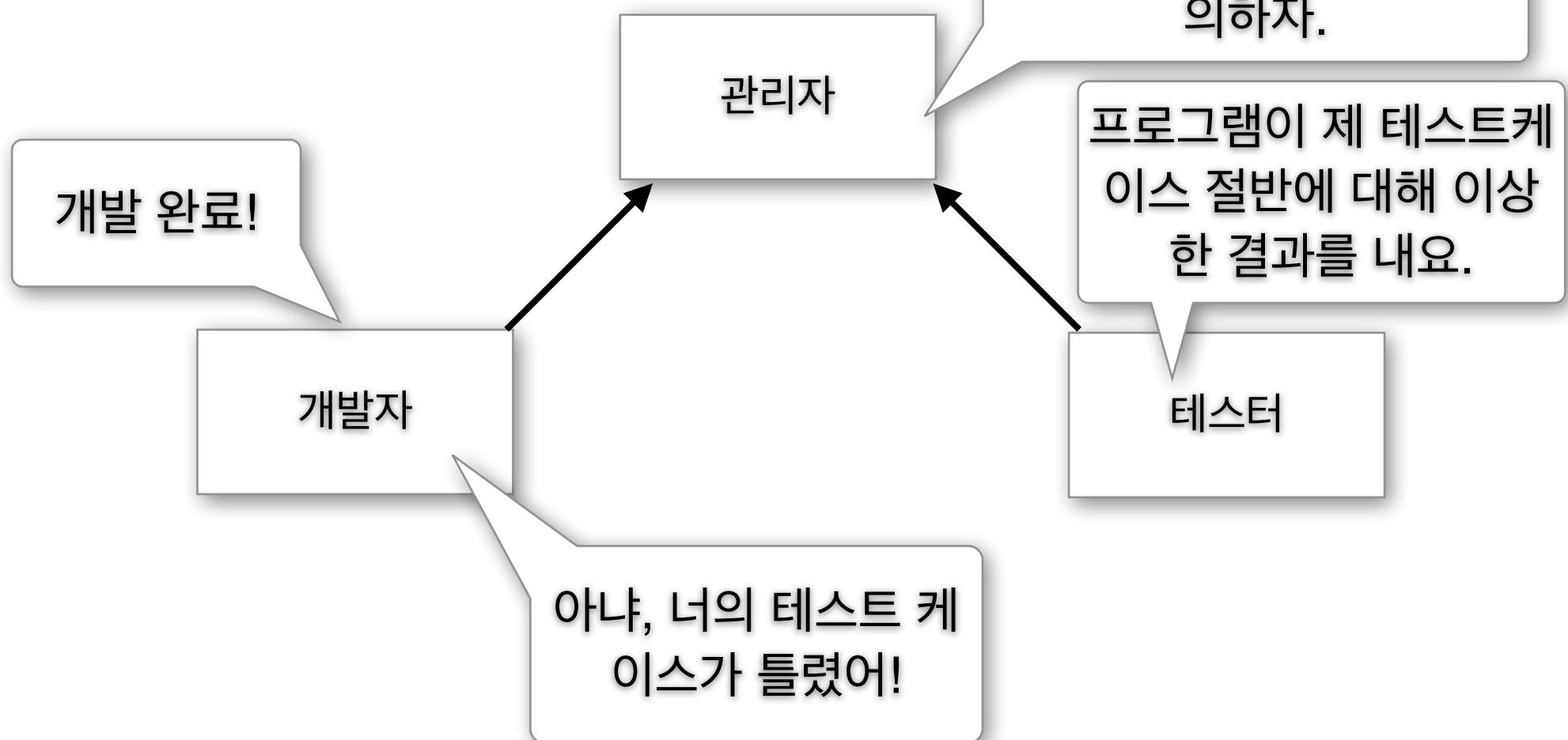
소프트웨어 개발 공정



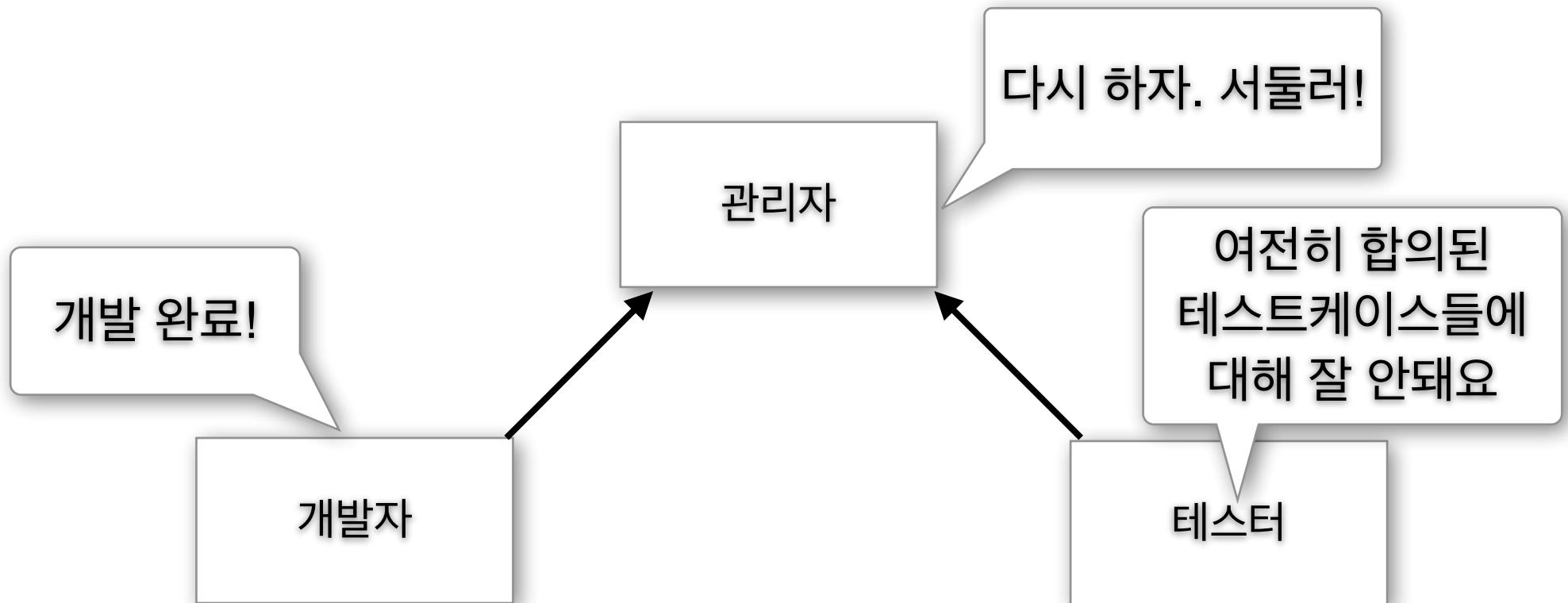
소프트웨어 개발 공정



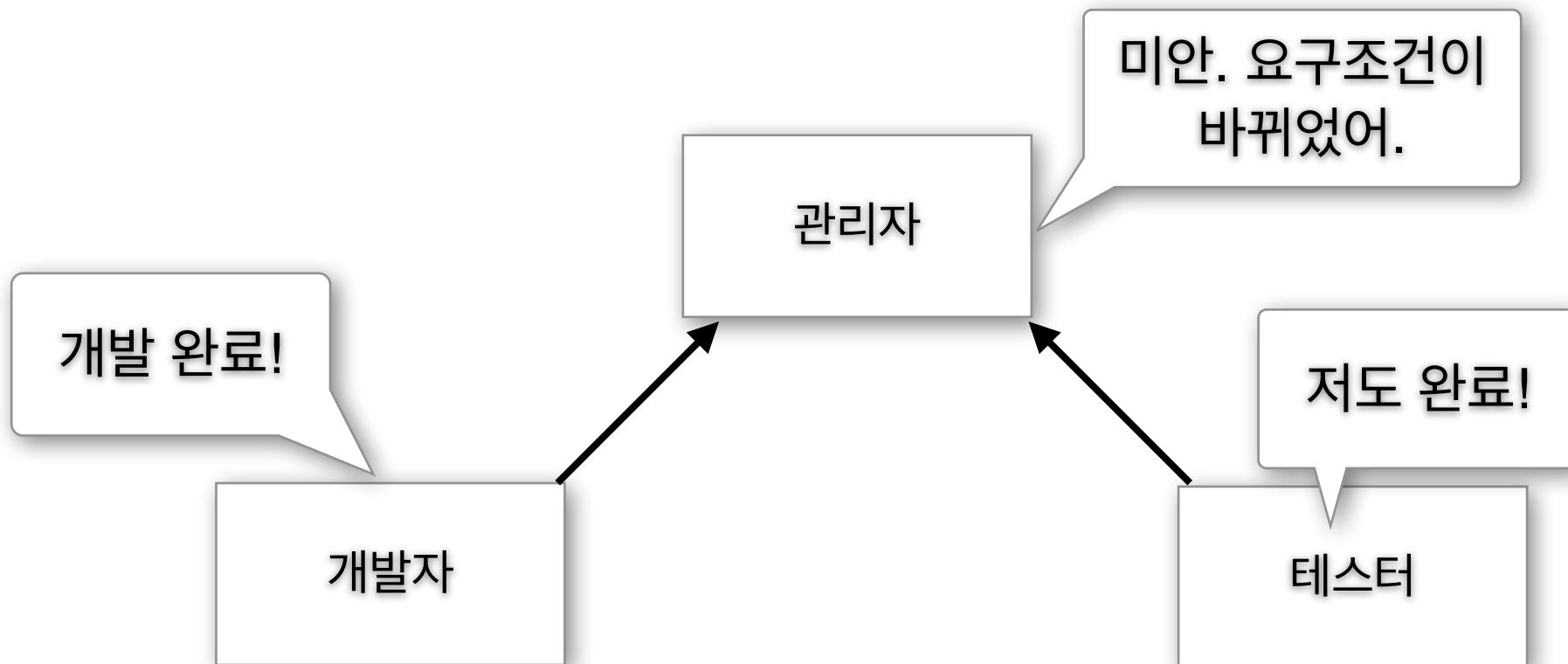
소프트웨어 개발 공정



소프트웨어 개발 공정



소프트웨어 개발 공정



중요한 관찰들

- 요구조건 (spec)은 분명해야 함.
- 개발과 테스팅은 따로 진행
- 사용할 수 있는 자원(시간, 인력 등)은 제한적
- 요구조건은 계속 변화
 - 테스트도 그에 맞게 업데이트 되어야 함.

요구조건 (Specification)

- 프로그램 구현이 요구조건에 맞는지 확인하기 위해 테스트 수행
- 요구조건이 없이는 테스트할 것이 없음.
- 요구조건은 오해의 여지 없이 올바르게 쓰여야.
 - 예: 입/출력 예제, 실행 전/후 만족시켜야 할 조건식

수동 vs. 자동

- 수동 테스팅
 - 프로그램에 대한 이해와 함께 작성될 경우 적은 수의 테스트 케이스로 효율적으로 테스트 수행 가능
 - 프로그램이 바뀌면 테스트 케이스도 함께 바뀌어야.
- 자동 테스팅
 - 오류를 더 빨리 찾을 수 있음
 - 테스트 케이스 수동으로 작성 불필요.
 - 프로그램이 바뀌어도 수동으로 바꿀 필요 없음

수동 단위 테스트 (Unit testing)

- 코드 기본 단위(예: 메소드)에 대해 테스팅을 수행
- 프로그램 전체를 테스팅하는 대신, 단위 별로 테스트를 수행하는 것의 이점:
 - 테스트 케이스를 작성하기 수월
 - 문제를 파악하기 더 수월
- 자바: JUnit 단위 테스트 Framework

Eclipse에서 JUnit 사용하기

7 배열 단원에서 다룬 단순 치환 암호 프로그램을 대상으로.

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer view displays a project named 'translate'. It contains a 'src' folder which has a 'translate' package. Inside the package, there are two source files: 'TranslateString.java' and 'TranslateTable.java'. The 'TranslateString.java' file is currently open in the editor. The code in 'TranslateString.java' is as follows:

```

1 package translate;
2
3 import javax.swing.JOptionPane;
4
5 public class TranslateString {
6
7     public static void main(String[] args) {
8         TranslateTable m = new TranslateTable(1);
9         String original = JOptionPane.showInputDialog("암호화할 문장을 입력하세요.");
10        String encoded = m.encode(original);
11        String decoded = m.decode(encoded);
12        JOptionPane.showMessageDialog(null, "original " + original + "\n encoded:");
    }
}

```

The 'TranslateTable.java' file is also visible in the Project Explorer, but its content is not shown in the screenshot. The code in 'TranslateTable.java' is as follows:

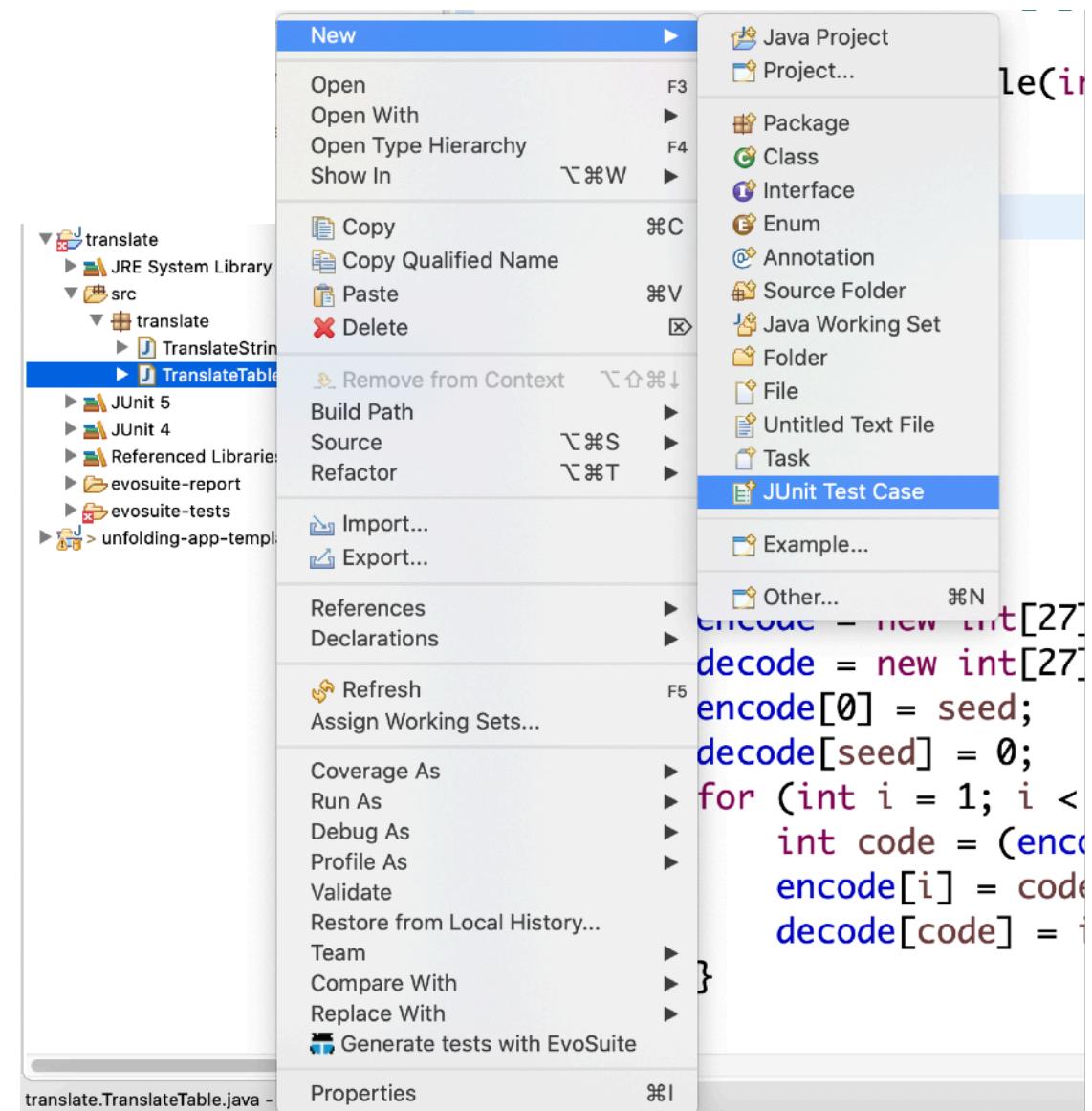
```

1 package translate;
2
3 public class TranslateTable {
4     private int[] encode; // 코드 -> 암호화된 코드 (encode[0]: ' '의 암호화된 코드값)
5     private int[] decode; // 암호화된 코드 -> 코드
6
7     // invariant: encode[0] = 5 <-> decode[5] = 0
8
9     public TranslateTable(int seed) {
10        // if seed = 1
11        // ' ' -> a
12        // a -> e
13        // b -> i
14        // c -> m
15        // d -> q
16        // e -> u
17        // f -> y
18        // g -> b
19        // ...
20        //
21        encode = new int[27];
22        decode = new int[27];
    }
}

```

Eclipse에서 JUnit 사용하기

- 테스트 할 클래스 파일 혹은 패키지 오른쪽 클릭 -> New -> JUnit Test Case 선택 -> Next 버튼
- New JUnit Jupiter test 선택 후 완료



Eclipse에서 JUnit 사용하기

- 다음과 같은 테스트 코드 생성
 - @Test의 의미:
test 메소드가 단위 테스트를 위한 메소드임

```
package translate;

import static
org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TranslateTableTest {

    @Test
    void test() {
        fail("Not yet implemented");
    }

}
```

Eclipse에서 JUnit 사용하기

○ 테스트 코드 작성

- `assertTrue(조건):`
조건이 참이면 테스트 성공, 아니면 실패

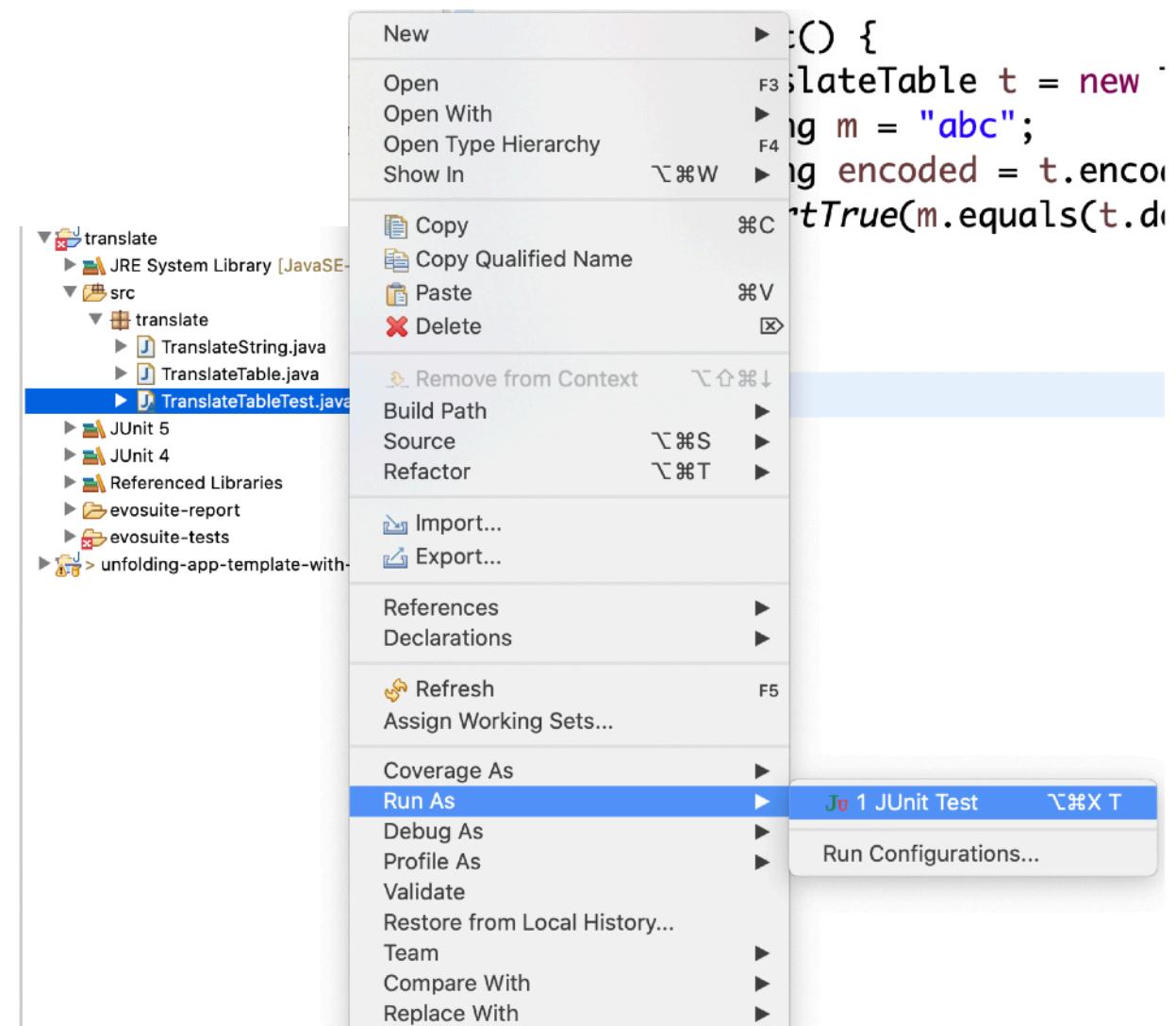
```
package translate;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class TranslateTableTest {
    @Test
    void test() {
        TranslateTable t = new TranslateTable(1);
        String m = "abc";
        String encoded = t.encode(m);
        assertTrue(m.equals(t.decode(encoded)));
    }
}
```

Eclipse에서 JUnit 사용하기

- 테스트 실행: 테스트 클래스 오른쪽 클릭
→ Run As → JUnit Test 선택



Eclipse에서 JUnit 사용하기

- 테스트가 성공시 다음과 같음 (실패 시 에러 출력)

The screenshot shows the Eclipse IDE interface with the JUnit perspective selected. The top menu bar includes 'File', 'Run As', 'JUnit', 'Help', and 'Exit'. The left sidebar has 'Package Explorer' and 'JUnit' tabs, with 'JUnit' currently active. Below the sidebar, status information reads 'Finished after 0.047 seconds', 'Runs: 1/1 Errors: 0 Failures: 0', and a green progress bar. The main workspace displays Java code for a JUnit test class:

```
1 package translate;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 class TranslateTableTest {
6
7     @Test
8     void test() {
9         TranslateTable t = new TranslateTable(1);
10        String m = "abc";
11        String encoded = t.encode(m);
12        assertTrue(m.equals(t.decode(encoded)));
13    }
14
15 }
16
17 }
18
```

The code uses static imports from `org.junit.jupiter.api.Assertions`. The test method `test()` creates a `TranslateTable` object, encodes the string "abc", and then decodes it to check if it equals the original string.

대표적인 단정문

- `assertArrayEquals(a,b)` : 배열 a와 b가 일치함을 확인
- `assertEquals(a,b)` : 객체 a와 b의 값이 같은지 확인
- `assertSame(a,b)` : 객체 a와 b가 같은 객체임을 확인
- `assertTrue(a)` : a가 참인지 확인
- `assertNotNull(a)` : a 객체가 null이 아님을 확인
- 참조: <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

추가 정보 기입 활용

```
import static org.junit.jupiter.api.Assertions.*;
import static java.time.Duration.*;
import org.junit.jupiter.api.Test;
@Test
public void test() { ... }
```

- 테스트 메소드 수행시간 제한 (시간단위: 밀리 초)

```
@Test
public void test() {assertTimeout(ofMillis( 시간 ), ()->{ 할일 })}
```

- 특정 예외가 발생해야 성공

```
@Test
public void test() {assertThrows(예외타입(예: RuntimeException).class,
()->{ 할일 })}
```

좋은 테스트?

- 작성한 테스트 케이스가 좋은것인지 어떻게 판단?
 - 너무 적은 테스트 케이스: 오류를 놓칠 수 있음
 - 너무 많은 테스트 케이스: 테스트 비용 증가, 중복되거나 필요치 않은 테스트 케이스 존재, 프로그램 변화에 따라 테스트 케이스 업데이트하기 어려워짐
- 흔히 실행되는 코드 양 (code coverage) 으로 판단

실행되는 코드 양 (Code Coverage)

- 테스트 케이스들에 의해 프로그램 코드의 얼마나 많은 부분이 실행되는지 측정하는 척도 (%)
- 100%는 달성하기 어려움
 - 모든 부분을 커버하는 테스트 케이스 작성 어려움
 - 일부분은 어느 입력이 주어지든 아예 실행되지 않을 수도 (dead code)
 - 하지만 안전이 중요한 (safety-critical) 소프트웨어에는 간혹 달성이 요구됨

척도의 종류

- 함수 coverage: 테스트 케이스들에 의해 얼마나 많은 함수가 호출되었는가?
- 라인 coverage: 얼마나 많은 코드 줄이 실행되었는가?
- 분기 coverage: 얼마나 많은 조건문 분기가 실행되었는가?
- 이클립스에서 도출 방법
 - 메뉴 “Run” → “Coverage”
 - 하이라이트 효과 끌 때: 메뉴 “Windows” → “Show View” → “Other...” → 텍스트 창 “Coverage” 입력 후 클릭 → 새로 생긴 Coverage View 창에 Remove all sessions 버튼 클릭

척도의 종류

- 테스트 입력: foo(1, 0)
- 라인 coverage: 80%
- 분기 coverage: 50%
- 두 coverage 를 100%로 만들기 위해 필요한 추가 테스트 입력은?
→ foo(1, 1)

```
int foo (int x, int y) {  
    int z;  
    if (x <= y) {  
        z = x;  
    }  
    else {  
        z = y;  
    }  
    return z;  
}
```

자동 테스트 코드 생성

○ 소프트웨어 도구 EvoSuite 에 의해 자동으로 생성된 코드 예

정상 테스트
케이스

```
@Test(timeout = 4000)
public void test1() throws Throwable {
    TranslateTable translateTable0 = new TranslateTable(0);
    String string0 = translateTable0.decode("");
    assertEquals("", string0);
}
```

널 접근 오류
발생 케이스

```
@Test(timeout = 4000)
public void test2() throws Throwable {
    TranslateTable translateTable0 = new TranslateTable(1);
    try {
        translateTable0.encode((String) null);
        fail("Expecting exception: NullPointerException");
    } catch(NullPointerException e) { assertTrue(true); }
```

사용자 정의
예외 발생
케이스

```
@Test(timeout = 4000)
public void test4() throws Throwable {
    TranslateTable translateTable0 = new TranslateTable(0);
    try { translateTable0.decode("8\u0007^Fw-I");
        fail("Expecting exception: RuntimeException");
    } catch(RuntimeException e) { assertTrue(true); }
```

자동 테스트 코드 생성

- 효과적으로 다음 두 종류의 테스트 케이스들을 자동으로 만들어줌을 확인할 수 있음:
 - 오류를 드러내는 테스트 케이스 (Error-revealing test case): 널 접근, 잘못된 배열 접근, 0으로 나누기 등 안전성 오류를 야기시키는 테스트 케이스.
 - 회귀 테스트 케이스 (Regression test case): 현재 버전이 올바르다고 가정하고, 향후 코드 수정시, 현재 버전과 다른 행동을 보이는 경우를 탐지하기 위한 테스트 케이스.

실행 방법

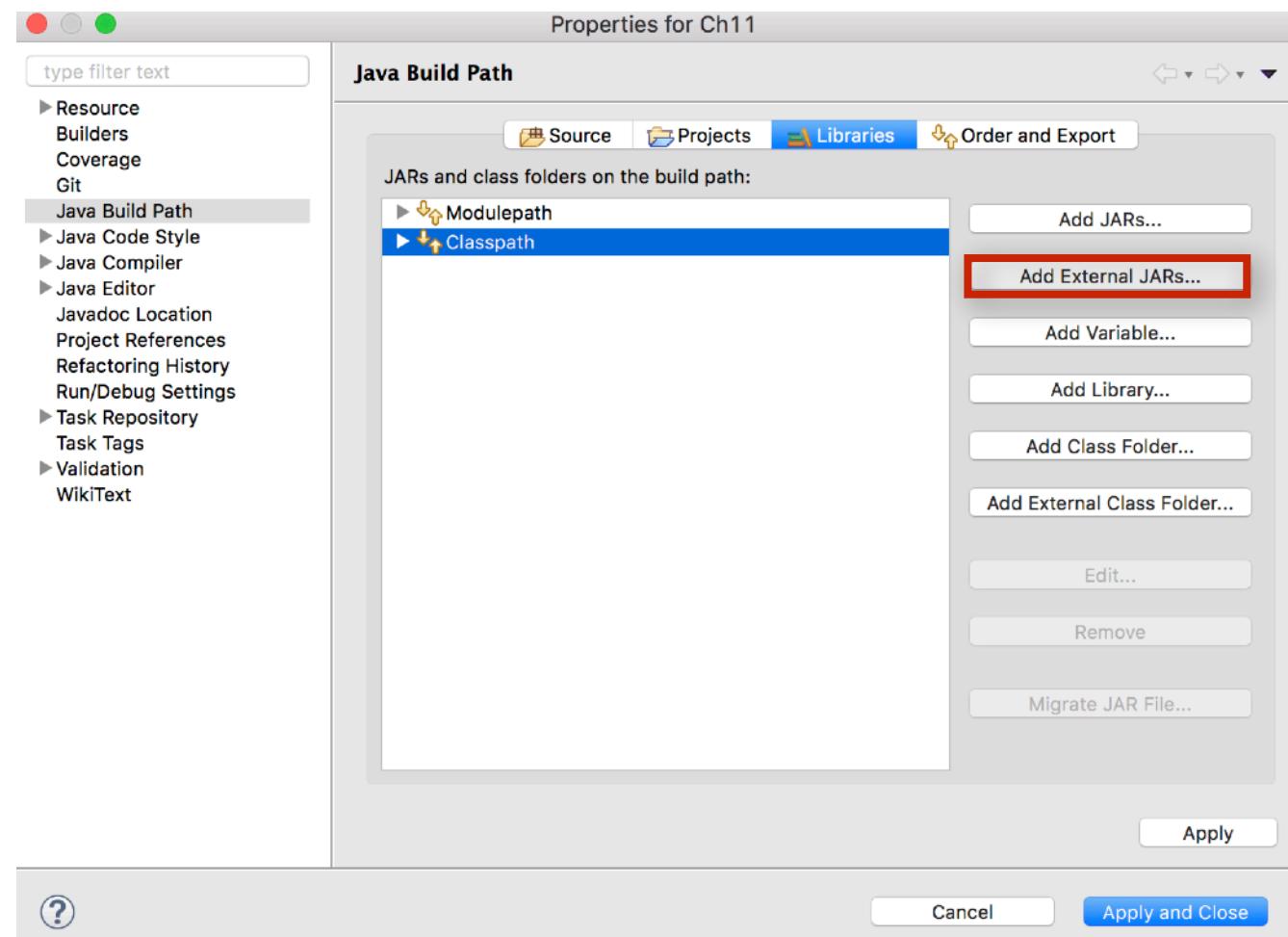
- 사용 도구: EvoSuite (<https://www.evosuite.org>) -
EvoSuite는 Eclipse 플러그인을 제공하지 않으므로, 직접
명령어 인터페이스에서 실행해야 함.
- 먼저 다음 파일들을 한 곳에 다운로드
 - EvoSuite executable: <https://github.com/EvoSuite/evosuite/releases/download/v1.0.6/evosuite-1.0.6.jar>
 - EvoSuite runtime: <https://github.com/EvoSuite/evosuite/releases/download/v1.0.6/evosuite-standalone-runtime-1.0.6.jar>

실행 방법

- 명령어 프롬프트 (혹은 터미널) 실행 후 이클립스 대상 프로젝트 위치로 이동
 - 예: cd C:\eclipse-workspace\translate
- 다음 명령어 수행
 - java -jar [evosuite executable 위치] -generateSuite -Dsearch_budget=60 -Dstopping_condition=MaxTime -projectCP=bin -class=[프로젝트 이름].[클래스 이름] (예: translate.TranslateTable)
 - 경우에 따라 윈도우 환경에서 환경변수 PATH 및 추가 옵션 제공해야 할 필요 (강의 영상 참조)

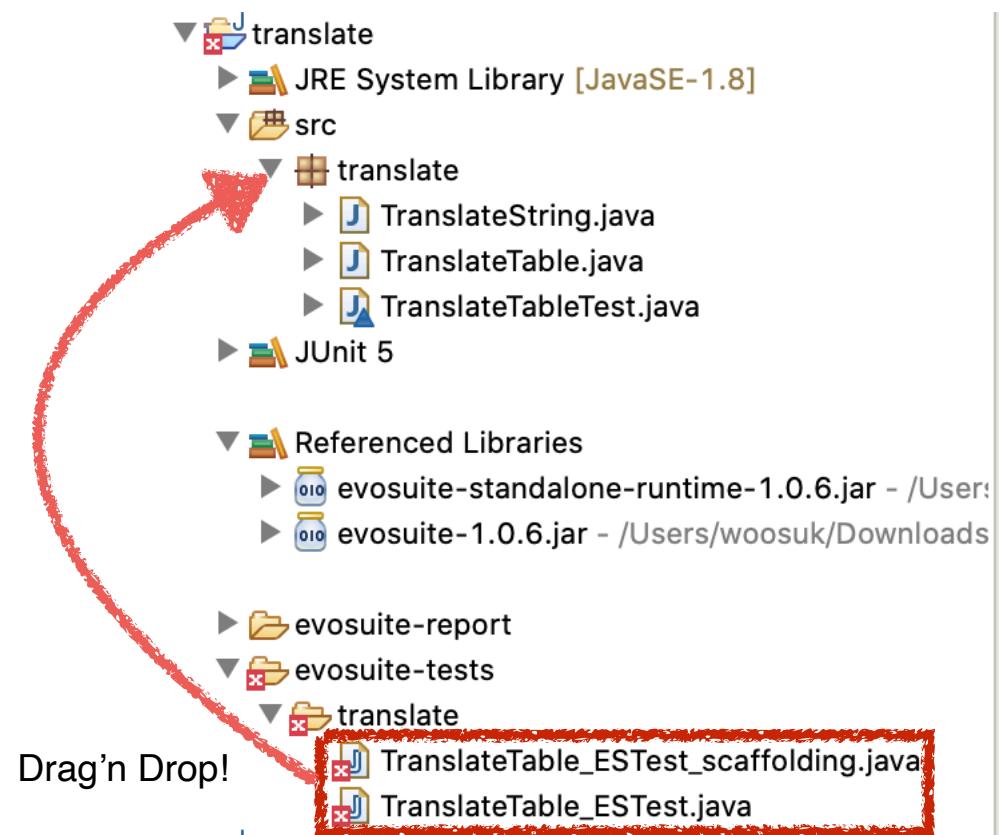
실행 방법

- 다시 이클립스로 돌아가서, Project 이름에 오른쪽 클릭 -> 성질 (Properties) 선택
- Java Build Path 선택
- Libraries 탭을 선택
- Add External JARS... 선택
- 다운로드 받은 evosuite JAR 파일들 선택



실행 방법

- Project 이름 오른쪽 클릭 -> 새로고침 (Refresh) 선택 시 오른쪽과 같이 evosuite-tests 폴더에 새로운 파일들이 생성되어 있음이 보임.
- evosuite-tests 폴더 밑의 java 파일들을 소스코드들과 동일한 위치로 옮김
- 옮겨진 유닛 테스트 코드 (TranslateTable_ESTest.java)를 실행



실행 결과

The screenshot shows the Eclipse IDE interface with the following details:

- Package Explorer View:** Shows a tree structure of test cases under "translate.TranslateTable_ESTest [Runner: JUnit 5]".
 - test0 (0.000 s)
 - test1 (0.000 s)
 - test8 (0.002 s)
 - test6 (0.003 s)
 - test7 (0.002 s)
 - test4 (0.002 s)
 - test5 (0.002 s)
 - test2 (0.013 s)
 - test3 (0.002 s)
- JUnit View:** Shows the results of the last run:
 - Runs: 9/9
 - Errors: 0
 - Failures: 0A green progress bar indicates the run was completed successfully.
- Code Editor View:** Displays the Java source code for "TranslateTable_ESTest".

```
2+ * This file was automatically generated by
5
6 package translate;
7
8+import org.junit.Test;[]
15
16 @RunWith(EvoRunner.class) @EvoRunnerParamet
17 public class TranslateTable_ESTest extends
18
19@Test(timeout = 4000)
20 public void test0() throws Throwable {
21     TranslateTable translateTable0 = new
22     String string0 = translateTable0.enca
23     assertEquals("", string0);
24 }
25
26@Test(timeout = 4000)
27 public void test1() throws Throwable {
```