CS608
Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

CS608

Random Testing

(Essentials of Software Testing, Chapter 12)

Introduction

- Fully automated random testing with no manual input is a goal of the software industry

Introduction

- Fully automated random testing with no manual input is a goal of the software industry
- Reminder: three key issues with automated random testing:
    1. The Test Oracle problem – how to tell if the result is correct.
    2. The Test Data problem – how to generate representative random data.
    3. The Test Completion problem – how to know when to stop generating tests.

Introduction

- Fully automated random testing with no manual input is a goal of the software industry
- Reminder: three key issues with automated random testing:
    1. **The Test Oracle problem – how to tell if the result is correct.**
    2. The Test Data problem – how to generate representative random data.
    3. The Test Completion problem – how to know when to stop generating tests.

Introduction

- Fully automated random testing with no manual input is a goal of the software industry
- Reminder: three key issues with automated random testing:
    1. The Test Oracle problem – how to tell if the result is correct.
    2. **The Test Data problem – how to generate representative random data.**
    3. The Test Completion problem – how to know when to stop generating tests.

Introduction

- Fully automated random testing with no manual input is a goal of the software industry
- Reminder: three key issues with automated random testing:
    1. The Test Oracle problem – how to tell if the result is correct.
    2. The Test Data problem – how to generate representative random data.
    3. **The Test Completion problem – how to know when to stop generating tests.**

Introduction

- Fully automated random testing with no manual input is a goal of the software industry
- Reminder: three key issues with automated random testing:
    1. The Test Oracle problem – how to tell if the result is correct.
    2. The Test Data problem – how to generate representative random data.
    3. The Test Completion problem – how to know when to stop generating tests.
- This is an active research area

Introduction

- Fully automated random testing with no manual input is a goal of the software industry
- Reminder: three key issues with automated random testing:
  1. The Test Oracle problem – how to tell if the result is correct.
  2. The Test Data problem – how to generate representative random data.
  3. The Test Completion problem – how to know when to stop generating tests.
- This is an active research area
- We will consider a simple form of automated random testing using:
  - random data
  - with manually generated black-box or white-box test coverage items

AI in Software Testing

- We’ll look at this next week

Random Data Selection

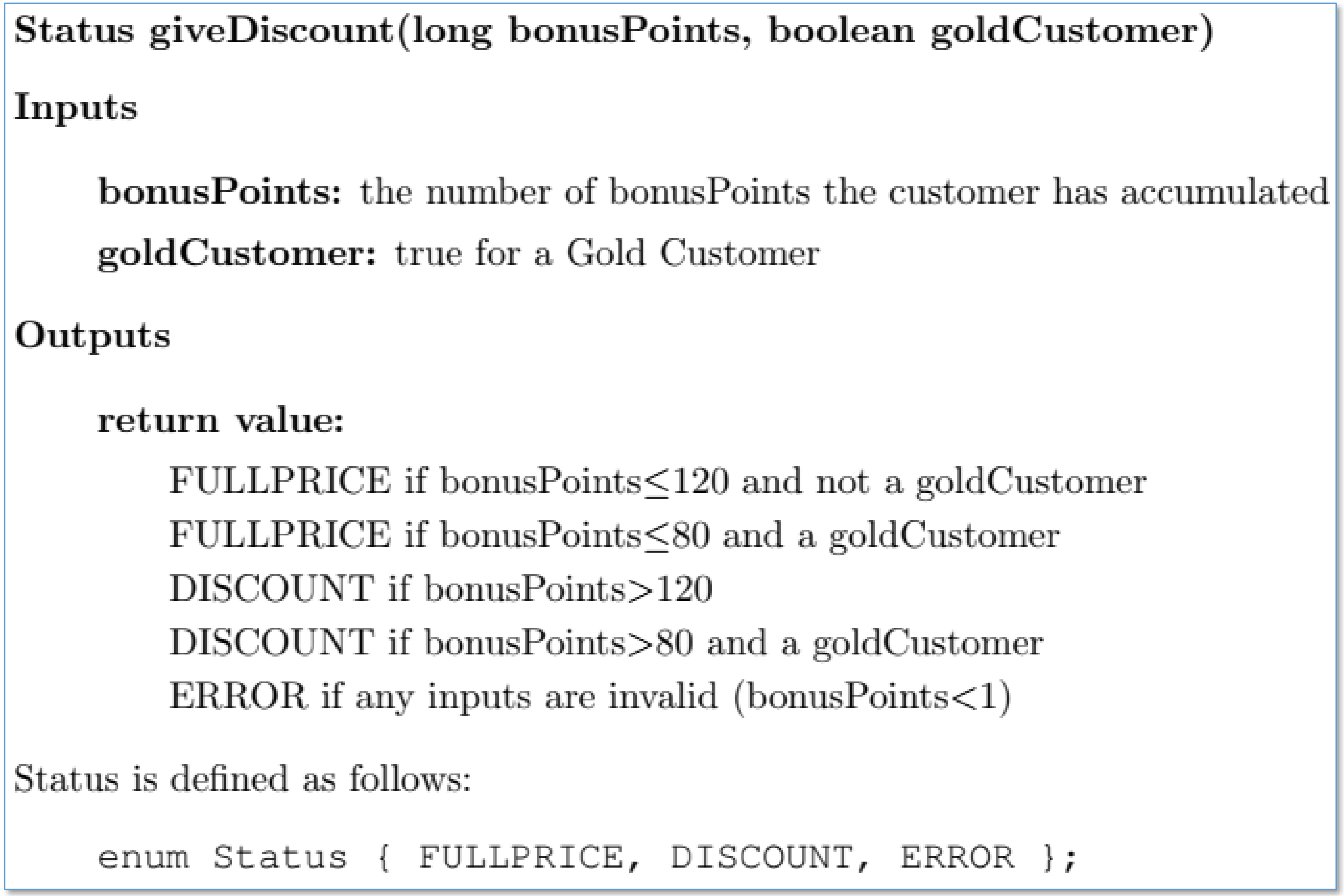
- We will use the same techniques that are covered previously to generate the test coverage items
- But instead of creating test cases with specific test data *values*, more general test data *criteria* are developed
- This allows the selection of random values that meet those criteria at runtime

Addressing Automated Random Test Issues

- Manual development of test coverage items
    - addresses the **test oracle problem**
    - by including the tester in the process.
- Using test data criteria and random number generators
    - addresses the **test data problem**
    - As these allow the generation of ‘good’ test data
- Limiting the test execution time
    - addresses the **test completion problem**
    - by offering a strategy for stopping

Two Examples

- Unit Test
  - Random testing of OnlineSales.giveDiscount()
- Application Test
  - Random testing of Fuel Checker



Status giveDiscount(long bonusPoints, boolean goldCustomer)

Inputs

bonusPoints: the number of bonusPoints the customer has accumulated

goldCustomer: true for a Gold Customer

Outputs

return value:

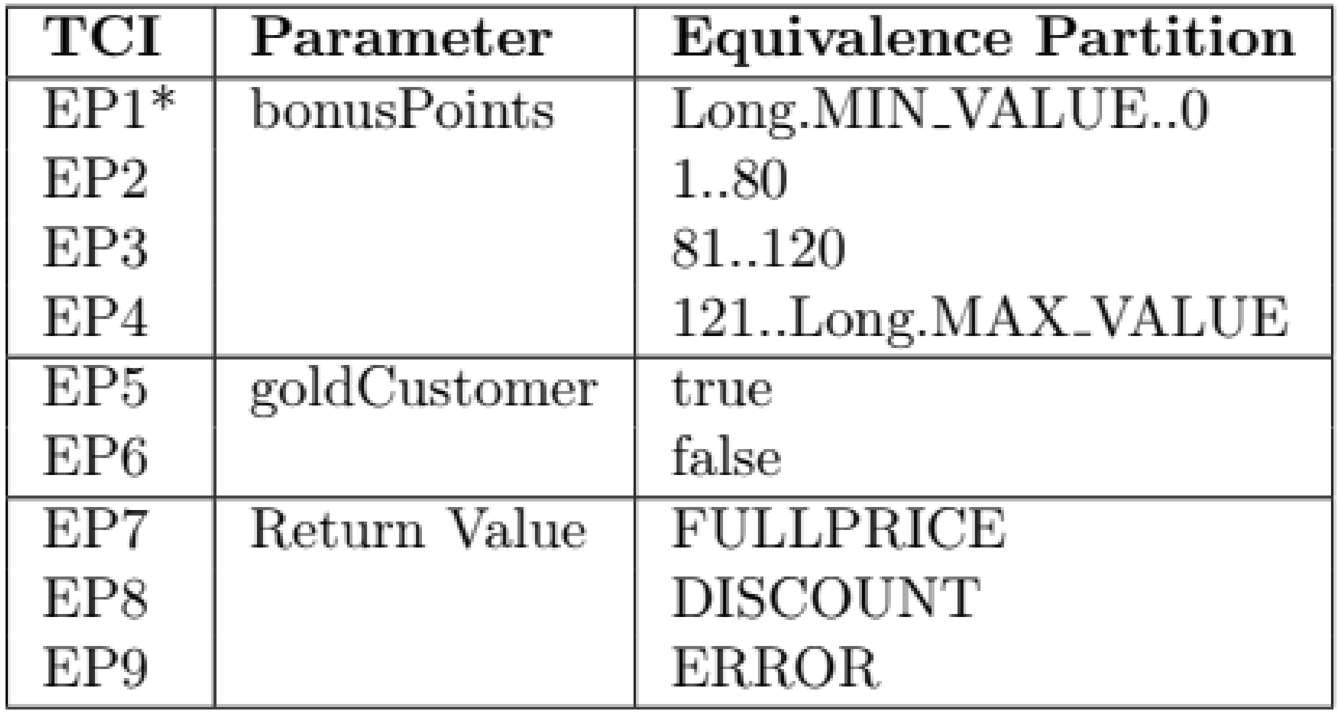
FULLPRICE if bonusPoints$\leq$120 and not a goldCustomer
FULLPRICE if bonusPoints$\leq$80 and a goldCustomer
DISCOUNT if bonusPoints$>$120
DISCOUNT if bonusPoints$>$80 and a goldCustomer
ERROR if any inputs are invalid (bonusPoints$<$1)

Status is defined as follows:

```
enum Status { FULLPRICE, DISCOUNT, ERROR };
```

Test Coverage Items

- Based on the equivalence partition tests developed previously
- All the other black-box and white-box tests could also be used for more comprehensive testing

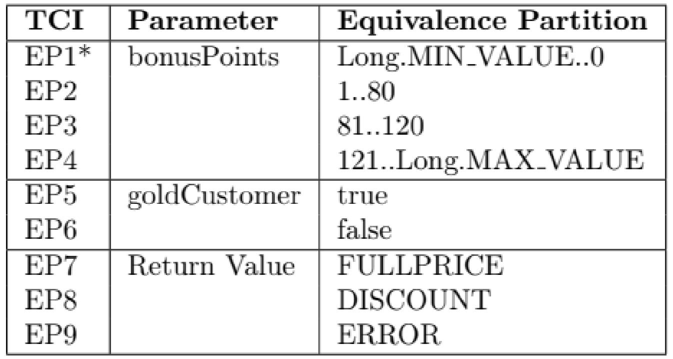


TCI	Parameter	Equivalence Partition
EP1*	bonusPoints	Long.MIN_VALUE..0
EP2		1..80
EP3		81..120
EP4		121..Long.MAX_VALUE
EP5	goldCustomer	true
EP6		false
EP7	Return Value	FULLPRICE
EP8		DISCOUNT
EP9		ERROR

Test Cases

- Instead of selecting actual **data values** to represent each equivalence partition, **criteria** that define that equivalence partition are used instead
- These allow specific data values to be selected at random, based on the criteria, at runtime
- For simple (EP) partitions, the criteria will state the minimum and maximum values in the partition
- In this example, these criteria are based on the partition boundaries for each TCI

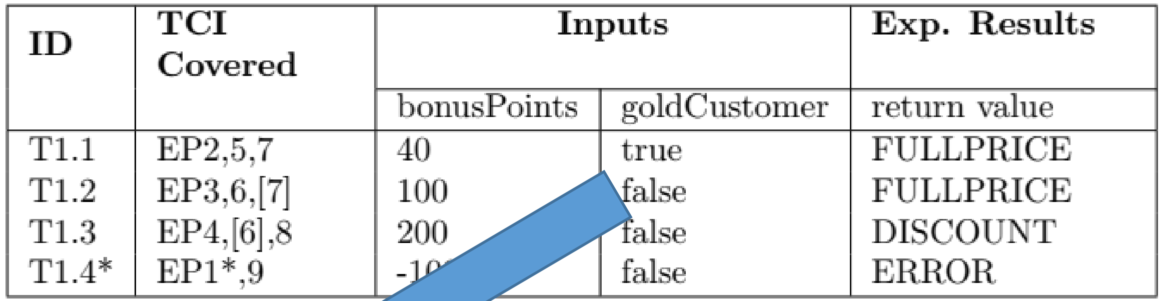
Equivalence Value Criteria



TCI	Parameter	Equivalence Partition
EP1*	bonusPoints	Long.MIN_VALUE..0
EP2		1..80
EP3		81..120
EP4		121..Long.MAX_VALUE
EP5	goldCustomer	true
EP6		false
EP7	Return Value	FULLPRICE
EP8		DISCOUNT
EP9		ERROR

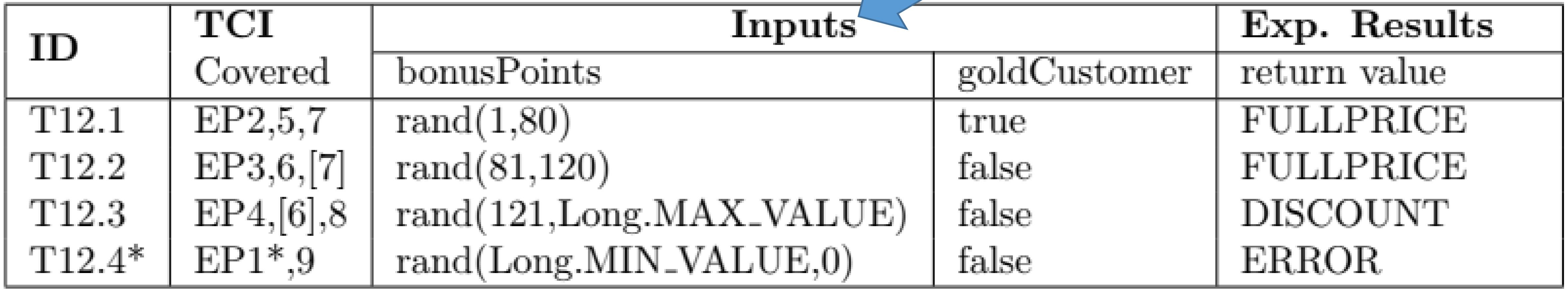
Parameter	EP	Criteria
bonusPoints	Long.MIN_VALUE..0	Long.MIN_VALUE<=bonusPoints && bonusPoints<=0
	1..80	1<=bonusPoints && bonusPoints<=80
	81..120	81<=bonusPoints && bonusPoints<=120
	121..Long.MAX_VALUE	121<=bonusPoints && bonusPoints<=Long.MAX_VALUE
goldCustomer	true	goldCustomer
	false	!goldCustomer
Return Value	FULLPRICE	Return Value==FULLPRICE
	DISCOUNT	Return Value==DISCOUNT
	ERROR	Return Value==ERROR

Random EP Test Cases



ID	TCI Covered	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T1.1	EP2,5,7	40	true	FULLPRICE
T1.2	EP3,6,[7]	100	false	FULLPRICE
T1.3	EP4,[6],8	200	false	DISCOUNT
T1.4*	EP1*,9	-10	false	ERROR

- These criteria are used for each test case instead of values



ID	TCI Covered	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T12.1	EP2,5,7	rand(1,80)	true	FULLPRICE
T12.2	EP3,6,[7]	rand(81,120)	false	FULLPRICE
T12.3	EP4,[6],8	rand(121,Long.MAX_VALUE)	false	DISCOUNT
T12.4*	EP1*,9	rand(Long.MIN_VALUE,0)	false	ERROR

- rand(l,h) indicates a random number between l and h inclusive
- Error test cases: limits number of tests with invalid data
- goldCustomer cannot be random – value required for exp. results

Test Implementation

- The test implementation is similar to that for EP testing
- As the same test method is used for a large number of random data values, a data provider is used
- The data provider returns the criteria required for selecting values, instead of specific values
- The criteria are then used in the test method to create the data values

Data Provider

```
@DataProvider(name="eprandom")
public Object[][] getEpData() {
   return new Object[][] {
      { "T12.1",              1L,              80L,  true, FULLPRICE},
      { "T12.2",             81L,             120L, false, FULLPRICE },
      { "T12.3",            121L, Long.MAX_VALUE, false, DISCOUNT },
      { "T12.4", Long.MIN_VALUE,                0L, false, ERROR }
   };
}
```

Parameterised Random Test Method

```
@Test(dataProvider="eprandom")
public void randomTest(String tid,long minp, long maxp, boolean cf, Status exp) {
   Random rp=new Random();
   long endTime=System.currentTimeMillis()+RUNTIME;
   testId = tid;
   while (System.currentTimeMillis() < endTime) {
      bonusPoints = generateRandomLong( rp, minp, maxp );
      goldCustomer = cf;
      failed = true;
      expected = exp;
      actual = OnlineSales.giveDiscount(bonusPoints,goldCustomer);
      assertEquals( actual, exp );
      failed = false;
   }
}
```

Parameterised Random Test Method

- While loop limits execution time

```
@Test(dataProvider="eprandom")
public void randomTest(String tid,long minp, long maxp, boolean cf
   Random rp=new Random();
   long endTime=System.currentTimeMillis()+RUNTIME;
   testId = tid;
   while (System.currentTimeMillis() < endTime) {
      bonusPoints = generateRandomLong( rp, minp, maxp );
      goldCustomer = cf;
      failed = true;
      expected = exp;
      actual = OnlineSales.giveDiscount(bonusPoints,goldCustomer);
      assertEquals( actual, exp );
      failed = false;
   }
}
```

Parameterised Random Test Method

- While loop limits execution time
- Generate random values for bonusPoints

```
@Test(dataProvider="eprandom")
public void randomTest(String tid,long minp, long maxp, boolean cf
   Random rp=new Random();
   long endTime=System.currentTimeMillis()+RUNTIME;
   testId = tid;
   while (System.currentTimeMillis() < endTime) {
      bonusPoints = generateRandomLong( rp, minp, maxp );
      goldCustomer = cf;
      failed = true;
      expected = exp;
      actual = OnlineSales.giveDiscount(bonusPoints,goldCustomer);
      assertEquals( actual, exp );
      failed = false;
   }
}
```

Parameterised Random Test Method

- While loop limits execution time
- Generate random values for bonusPoints
- Call the method under test

```
@Test(dataProvider="eprandom")
public void randomTest(String tid,long minp, long maxp, boolean cf
   Random rp=new Random();
   long endTime=System.currentTimeMillis()+RUNTIME;
   testId = tid;
   while (System.currentTimeMillis() < endTime) {
      bonusPoints = generateRandomLong( rp, minp, maxp );
      goldCustomer = cf;
      failed = true;
      expected = exp;
      actual = OnlineSales.giveDiscount(bonusPoints,goldCustomer);
      assertEquals( actual, exp );
      failed = false;
   }
}
```

Parameterised Random Test Method

- While loop limits execution time
- Generate random values for bonusPoints
- Call the method under test
- Check result

```
@Test(dataProvider="eprandom")
public void randomTest(String tid,long minp, long maxp, boolean cf
   Random rp=new Random();
   long endTime=System.currentTimeMillis()+RUNTIME;
   testId = tid;
   while (System.currentTimeMillis() < endTime) {
      bonusPoints = generateRandomLong( rp, minp, maxp );
      goldCustomer = cf;
      failed = true;
      expected = exp;
      actual = OnlineSales.giveDiscount(bonusPoints,goldCustomer);
      assertEquals( actual, exp );
      failed = false;
   }
}
```

generateRandomLong(min,max) is tricky

- Random.nextLong()
  - Returns the next pseudorandom, uniformly distributed long value from this random number generator's sequence
  - The general contract of nextLong is that one long value is pseudorandomly generated and returned
  - Because class Random uses a seed with only 48 bits, this algorithm will not return all possible long values
- Also, consider the value of (max-min):
  - If the value of max-min is <= Long.MAX_VALUE this works
  - But if the value of max-min is > Long.MAX_VALUE, this returns a wrapped (negative) long value

generateRandomLong()

```
private static long generateRandomLong( Random r, long min, long max ) {
     long value;
     if ((max-min)>0)
        value = min + Math.round(r.nextDouble() * (max-min));
     else do
        value = r.nextLong();
     while ((value<min)||(value>max));
     return value;
  }
```

Note: there is no standard method to so this, so a customised method is required
This may not be the most efficient or mathematically perfect way…

Or Use

```
// Note: exclusive of max – also (probably) only 48 bits of value
private static long generateRandomLong( Random r, long min, long max ) {
   return ThreadLocalRandom.current().nextLong(min,max);
}
```

Test Failure Reporting

- If a test fails, the standard test report will only show the inputs to the test method, which are the test data criteria rather than the actual data values
- This can make replicating and debugging problems difficult
- To address this, and report the **actual** random data values that caused the failure, the test method saves the input parameter values and the expected results and sets failed to true
- If the test succeeds, failed is set to false
- If the test fails, then failed will remain true and the @AfterMethod method will be called to print the parameters values for a failed test

Saving the Test Data

- Test data is saved:
  bonusPoints
  goldCustomer
  expected

```
@Test(dataProvider="eprandom")
public void randomTest(String tid,long minp, long maxp, boolean cf
   Random rp=new Random();
   long endTime=System.currentTimeMillis()+RUNTIME;
   testId = tid;
   while (System.currentTimeMillis() < endTime) {
      bonusPoints = generateRandomLong( rp, minp, maxp );
      goldCustomer = cf;
      failed = true;
      expected = exp;
      actual = OnlineSales.giveDiscount(bonusPoints,goldCustomer);
      assertEquals( actual, exp );
      failed = false;
   }
}
```

Test Passes

- Test is marked as failed
- If test passes, test is marked as not failed

```
@Test(dataProvider="eprandom")
public void randomTest(String tid,long minp, long maxp, boolean cf
   Random rp=new Random();
   long endTime=System.currentTimeMillis()+RUNTIME;
   testId = tid;
   while (System.currentTimeMillis() < endTime) {
      bonusPoints = generateRandomLong( rp, minp, maxp );
      goldCustomer = cf;
      failed = true;
      expected = exp;
      actual = OnlineSales.giveDiscount(bonusPoints,goldCustomer);
      assertEquals( actual, exp );
      failed = false;
   }
}
```

Test Fails

- If test fails, then assertEquals raises an exception and the method terminates with failed==true

```
@Test(dataProvider="eprandom")
public void randomTest(String tid,long minp, long maxp, boolean cf
   Random rp=new Random();
   long endTime=System.currentTimeMillis()+RUNTIME;
   testId = tid;
   while (System.currentTimeMillis() < endTime) {
      bonusPoints = generateRandomLong( rp, minp, maxp );
      goldCustomer = cf;
      failed = true;
      expected = exp;
      actual = OnlineSales.giveDiscount(bonusPoints,goldCustomer);
      assertEquals( actual, exp );
      failed = false;
   }
}
```

reportFailures()

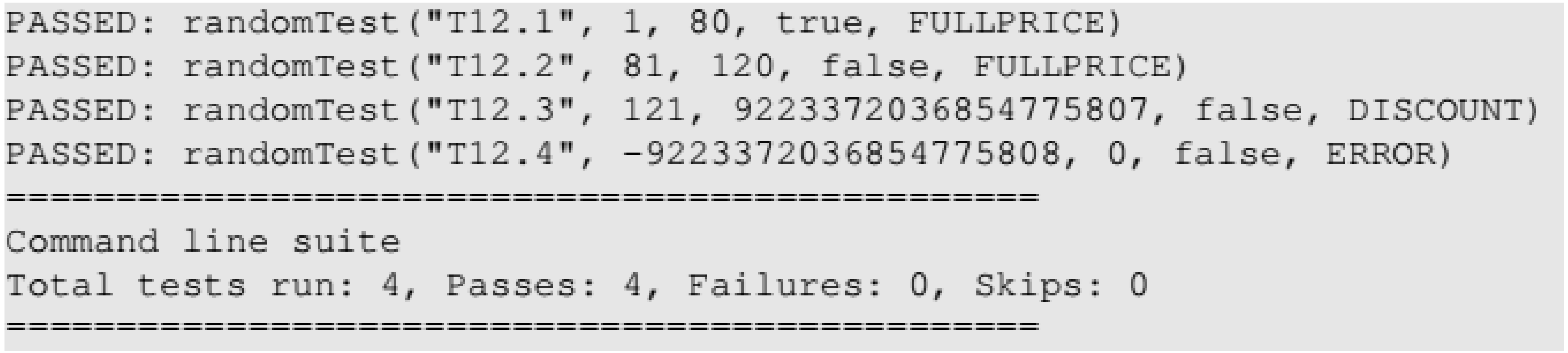
```
@AfterMethod
public void reportFailures() {
   if (failed) {
      System.out.println("Test failure data for test: "+testId);
      System.out.println("   bonusPoints="+bonusPoints);
      System.out.println("   goldCustomer="+goldCustomer);
      System.out.println("   actual result="+actual);
      System.out.println("   expected result="+expected);
   }
}
```

Test Attributes

```
private static long RUNTIME=1000; // Test runtime in milliseconds

// Store the data values in case of test failure
long bonusPoints;
boolean goldCustomer;
boolean failed=false;
Status expected, actual;
String testId;
```

Test Results



```
PASSED: randomTest("T12.1", 1, 80, true, FULLPRICE)
PASSED: randomTest("T12.2", 81, 120, false, FULLPRICE)
PASSED: randomTest("T12.3", 121, 9223372036854775807, false, DISCOUNT)
PASSED: randomTest("T12.4", -9223372036854775808, 0, false, ERROR)
===============================================
Command line suite
Total tests run: 4, Passes: 4, Failures: 0, Skips: 0
===============================================
```

- All the tests have passed
- With the selected value of RUNTIME=1000ms, total test execution time is approximately 4s
- You can add a loop counter…

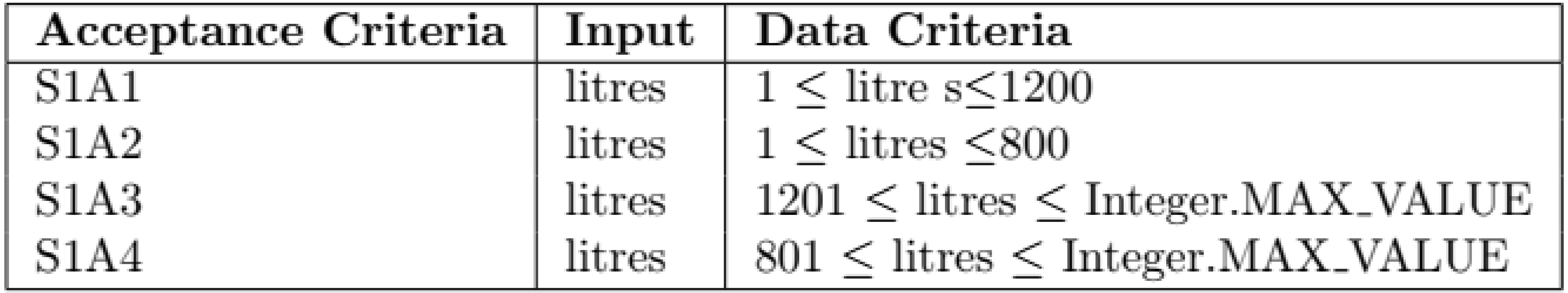
Application Testing Example

- To show the broad applicability of random testing, a random application test example is shown based on the Fuel Checker system
- Reminder:
  - A fuel depot wants a web-based system to enable the dispatcher to determine whether a load of fuel will fit in a tank or not
  - Some fuels can fill the tank; others require extra space for safety reasons
  - The tank capacity is 1200 litres without the extra safety space, and 800 litres with it

Analysis

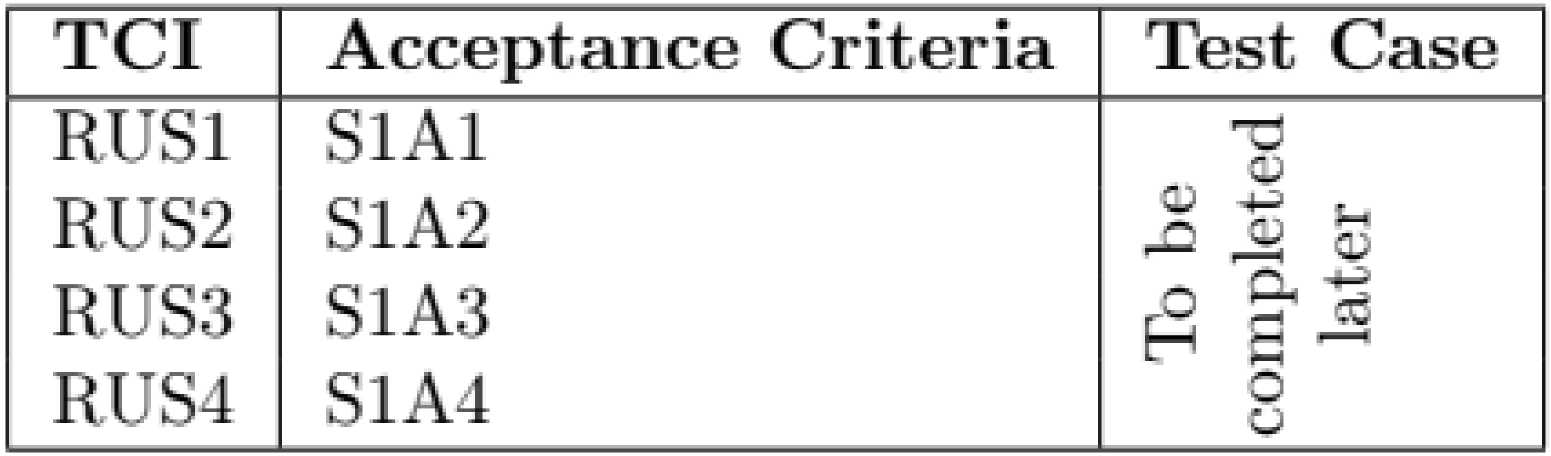
- Determine data criteria from the acceptance criteria
- S1A1 requires a value for litres that fits in a tank without extra safety
  - The range of values is from 1 to 1200
- S1A2 requires avalue for litres that fits in a tank with extra safety
  - The range of values is from 1 to 800.
- S1A3 requires a value for litres that does not fit in a tank without extra safety
  - The range of values is from 1201 to some unspecified maximum value
- S1A4 requires a value for litres that does not fit in a tank with extra safety
  - The range of values is from 801 to some unspecified maximum value

Data Value Criteria



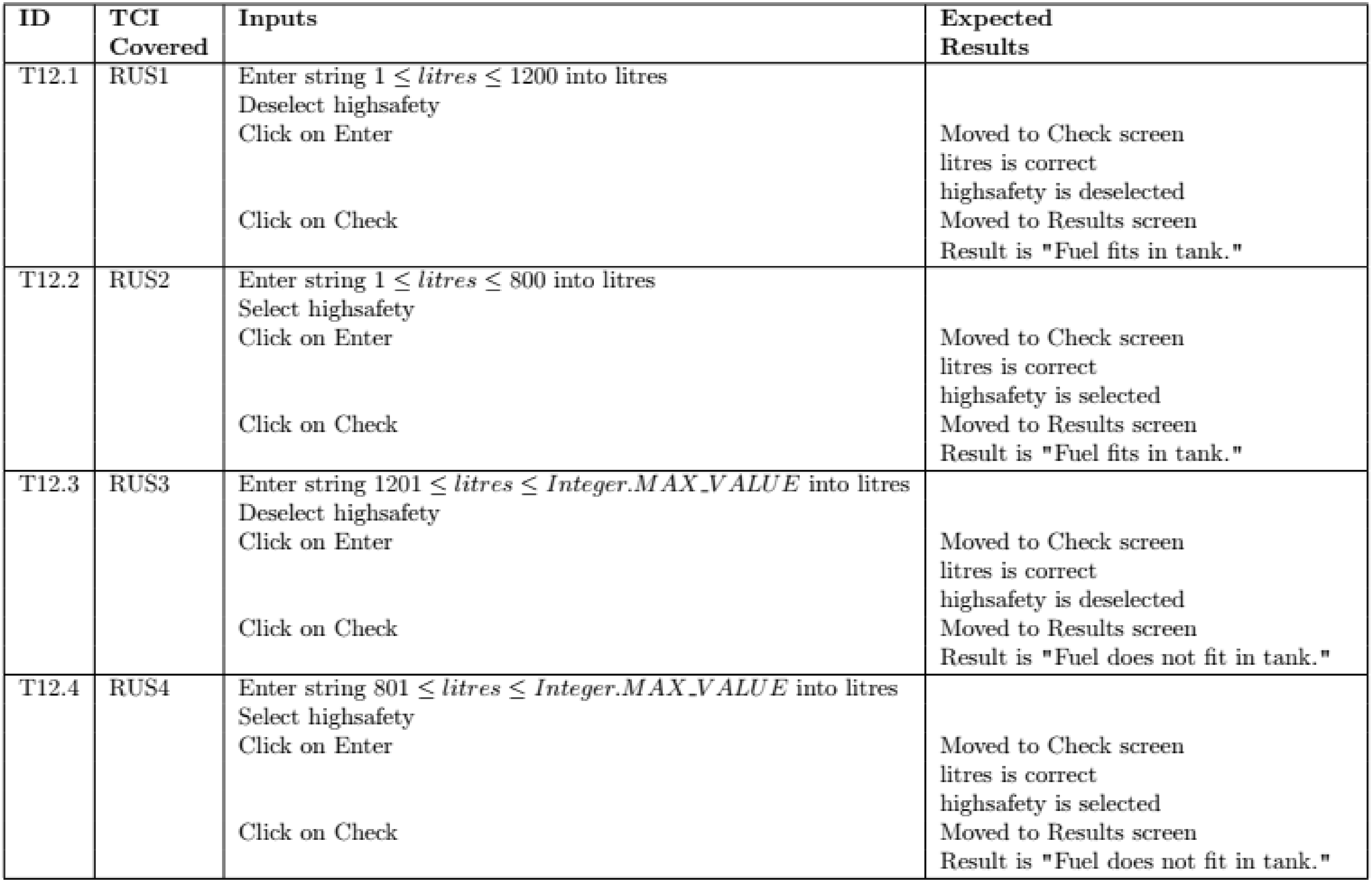
Acceptance Criteria	**Input**	**Data Criteria**
S1A1	litres	$1 \leq$ litre s≤ 1200
S1A2	litres	$1 \leq$ litres ≤ 800
S1A3	litres	$1201 \leq$ litres $\leq$ Integer.MAX_VALUE
S1A4	litres	$801 \leq$ litres $\leq$ Integer.MAX_VALUE

Test Coverage Items



TCI	Acceptance Criteria	Test Case
RUS1	S1A1	To be completed later
RUS2	S1A2	
RUS3	S1A3	
RUS4	S1A4	

Test Cases

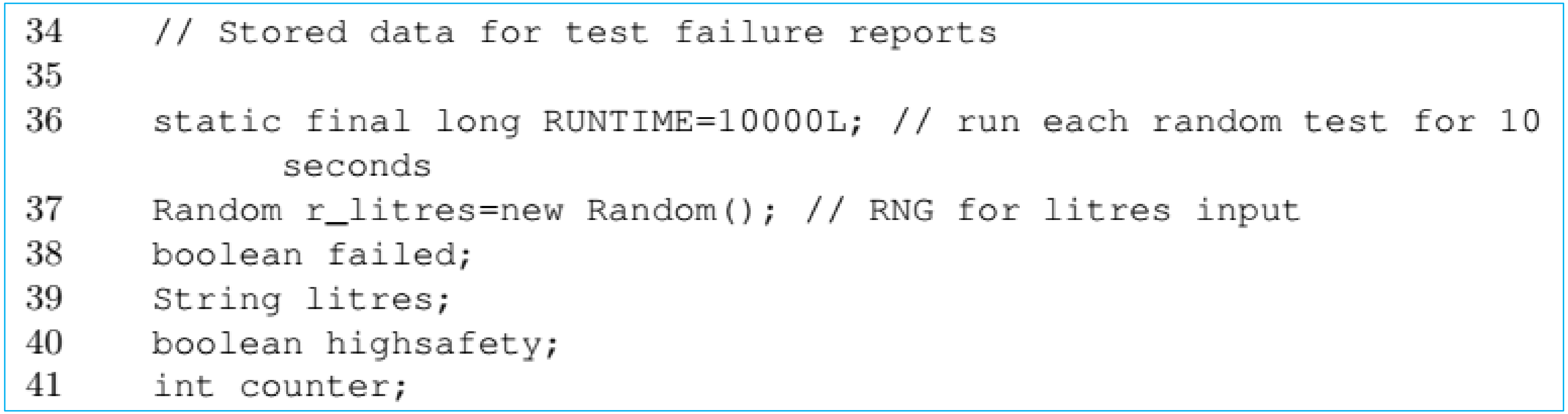


ID	TCI Covered	Inputs	Expected Results
T12.1	RUS1	Enter string $1 \leq litres \leq 1200$ into litres Deselect highsafety Click on Enter Click on Check	 Moved to Check screen litres is correct highsafety is deselected Moved to Results screen Result is "Fuel fits in tank."
T12.2	RUS2	Enter string $1 \leq litres \leq 800$ into litres Select highsafety Click on Enter Click on Check	 Moved to Check screen litres is correct highsafety is selected Moved to Results screen Result is "Fuel fits in tank."
T12.3	RUS3	Enter string $1201 \leq litres \leq Integer.MAX\_VALUE$ into litres Deselect highsafety Click on Enter Click on Check	 Moved to Check screen litres is correct highsafety is deselected Moved to Results screen Result is "Fuel does not fit in tank."
T12.4	RUS4	Enter string $801 \leq litres \leq Integer.MAX\_VALUE$ into litres Select highsafety Click on Enter Click on Check	 Moved to Check screen litres is correct highsafety is selected Moved to Results screen Result is "Fuel does not fit in tank."

Test Implementation

- The test implementation is based on the tests developed for the Fuel Checker application
- Setup and shutdown are essentially the same
- Changes:
    - Extra fields for random testing – including data values for failed tests
    - Data provider with criteria – specific data values are replaced with lower and upper bounds from the criteria developed for randomising the data
    - Test with a loop and random test data generation
    - @AfterMethod – printout failed test data values (and return to main screen)

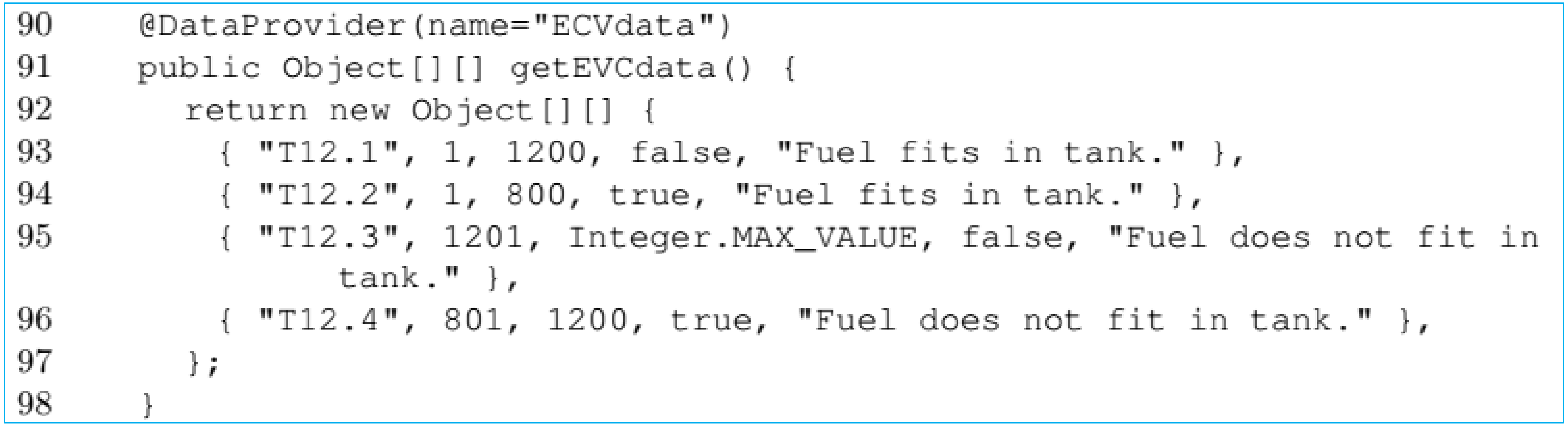
Extra Attributes



```
// Stored data for test failure reports

static final long RUNTIME=10000L; // run each random test for 10
      seconds
Random r_litres=new Random(); // RNG for litres input
boolean failed;
String litres;
boolean highsafety;
int counter;
```

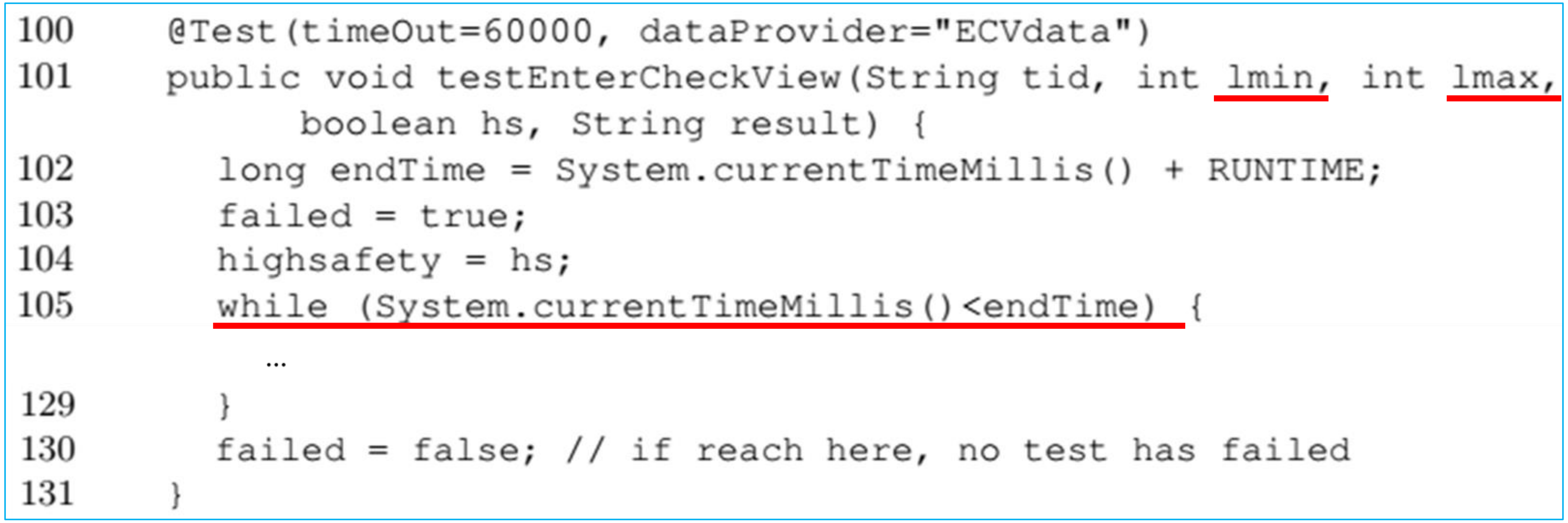
Data Provider



```
90     @DataProvider(name="ECVdata")
91     public Object[][] getEVCdata() {
92        return new Object[][] {
93          { "T12.1", 1, 1200, false, "Fuel fits in tank." },
94          { "T12.2", 1, 800, true, "Fuel fits in tank." },
95          { "T12.3", 1201, Integer.MAX_VALUE, false, "Fuel does not fit in
                   tank." },
96          { "T12.4", 801, 1200, true, "Fuel does not fit in tank." },
97        };
98     }
```

- 93-96: The data provider includes the criteria (for an integer, these are the lower and upper bounds) for litres, instead of specific values

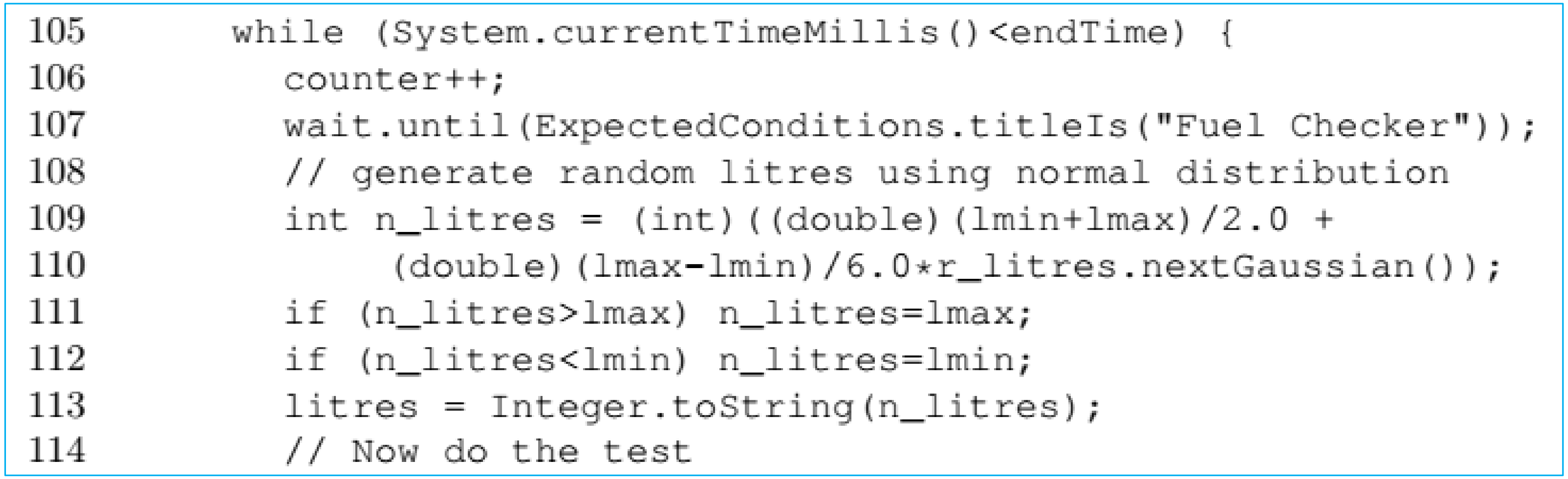
Test Method



```
100      @Test(timeOut=60000, dataProvider="ECVdata")
101      public void testEnterCheckView(String tid, int lmin, int lmax,
              boolean hs, String result) {
102        long endTime = System.currentTimeMillis() + RUNTIME;
103        failed = true;
104        highsafety = hs;
105        while (System.currentTimeMillis()<endTime) {
             ...
129        }
130        failed = false; // if reach here, no test has failed
131      }
```

- Line 101: the parameterised test includes extra input parameters lmin and lmax to support the lower and upper bounds for litres for each test
- Line 105: instead of running a single test, a number of loops with different random values are executed, using a timer to decide when to complete the random test

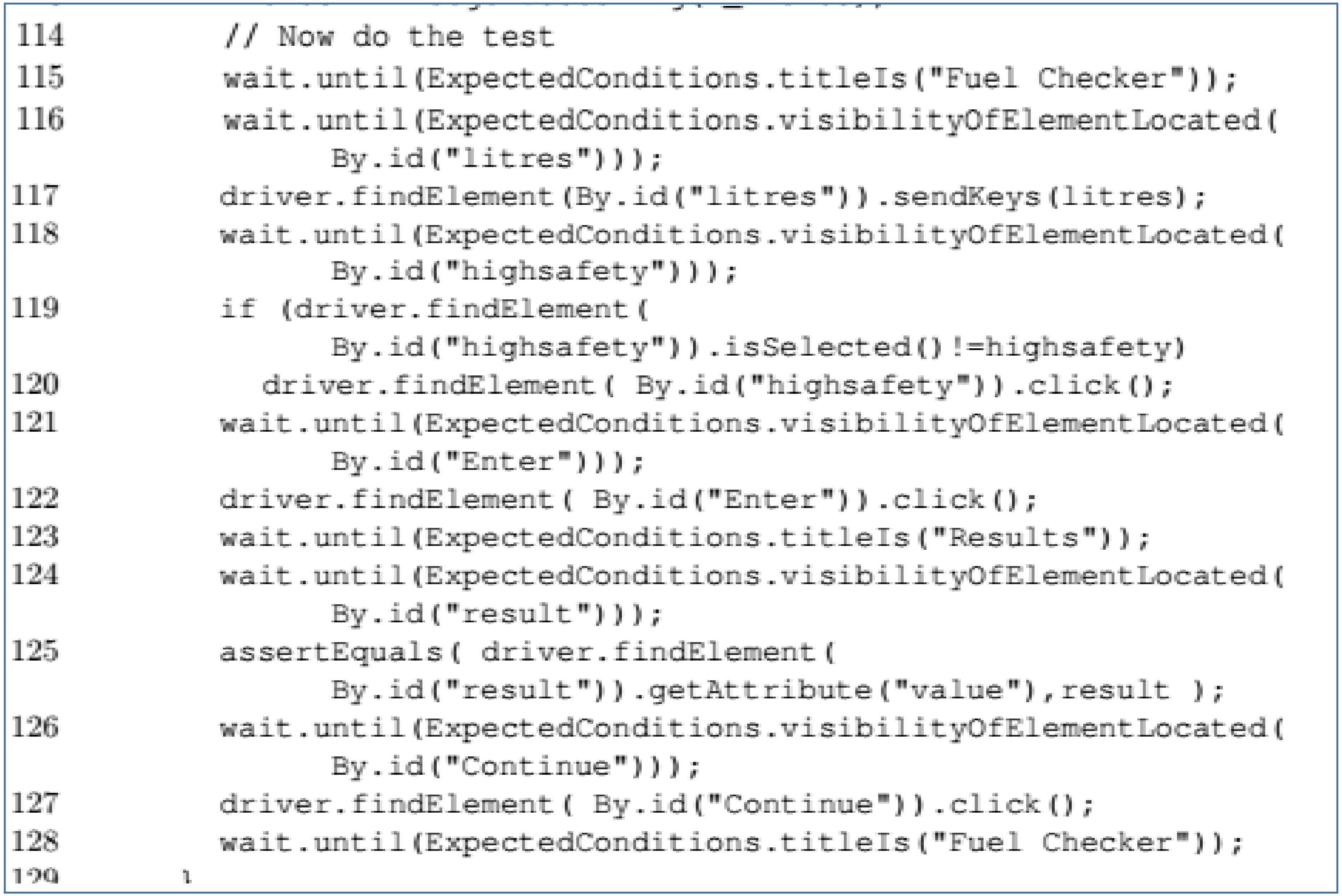
Test Loop/Generate Data



```
105        while (System.currentTimeMillis()<endTime) {
106          counter++;
107          wait.until(ExpectedConditions.titleIs("Fuel Checker"));
108          // generate random litres using normal distribution
109          int n_litres = (int)((double)(lmin+lmax)/2.0 +
110               (double)(lmax-lmin)/6.0*r_litres.nextGaussian());
111          if (n_litres>lmax) n_litres=lmax;
112          if (n_litres<lmin) n_litres=lmin;
113          litres = Integer.toString(n_litres);
114          // Now do the test
```

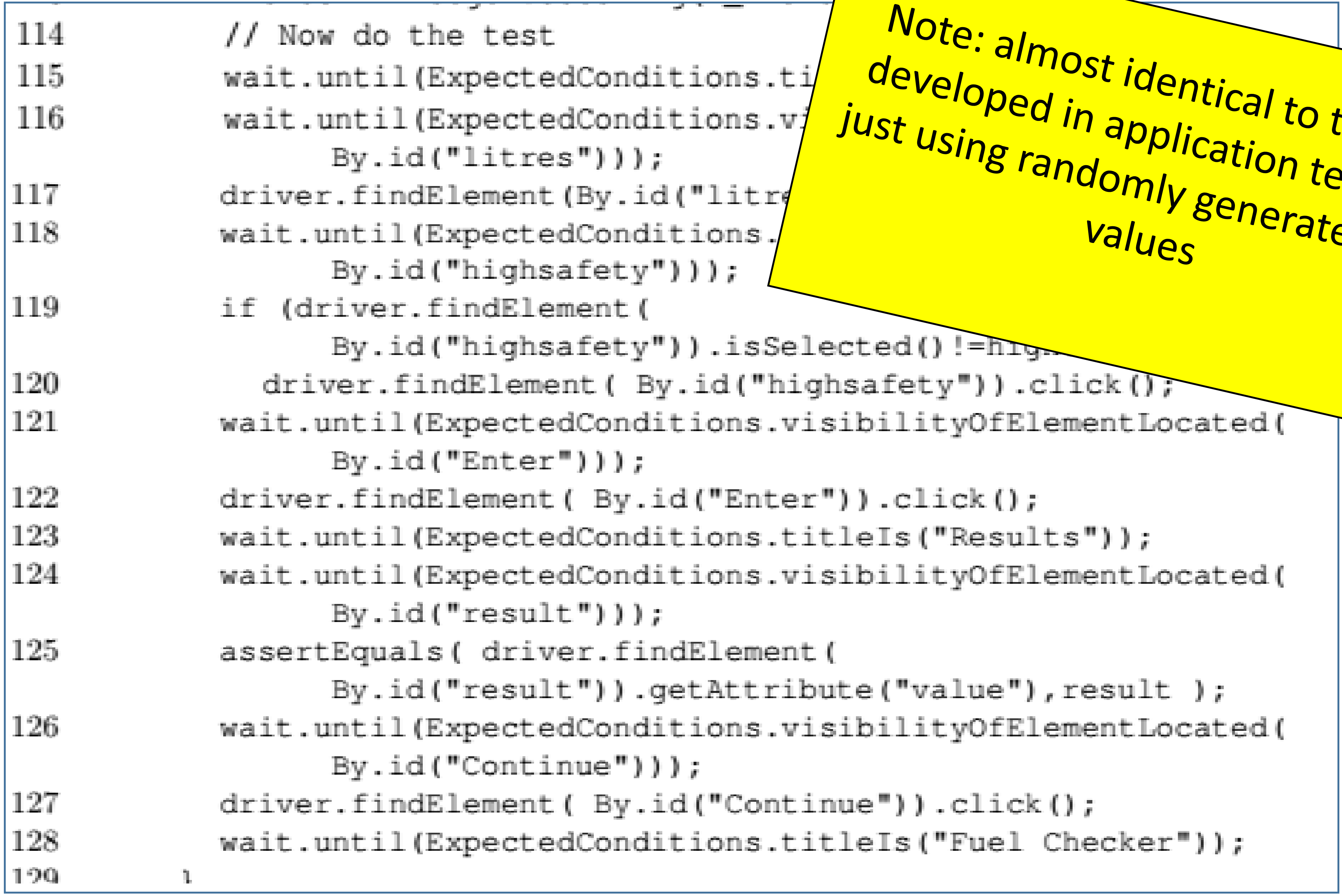
- **108-113**: A random value is selected for litres. For application testing, values based on actual user inputs often used, so testing better matches real-world use – when these statistics are not available, normally distributed values are often used as an approximation
- **110:** Random.nextGaussian() returns a normally distributed value, centered on 0.0, with a Std.Dev. of 3.0
- **109:** For a normal distribution, 99% of the values lie within 3 standard deviations, so this equation produces a random number centered in the middle of the range lmin..lmax, and normally distributed
- **111-112:** 1% of the values will lie outside this range, and these are moved into the range

The Test



```
    // Now do the test
    wait.until(ExpectedConditions.titleIs("Fuel Checker"));
    wait.until(ExpectedConditions.visibilityOfElementLocated(
            By.id("litres")));
    driver.findElement(By.id("litres")).sendKeys(litres);
    wait.until(ExpectedConditions.visibilityOfElementLocated(
            By.id("highsafety")));
    if (driver.findElement(
            By.id("highsafety")).isSelected()!=highsafety)
       driver.findElement( By.id("highsafety")).click();
    wait.until(ExpectedConditions.visibilityOfElementLocated(
            By.id("Enter")));
    driver.findElement( By.id("Enter")).click();
    wait.until(ExpectedConditions.titleIs("Results"));
    wait.until(ExpectedConditions.visibilityOfElementLocated(
            By.id("result")));
    assertEquals( driver.findElement(
            By.id("result")).getAttribute("value"),result );
    wait.until(ExpectedConditions.visibilityOfElementLocated(
            By.id("Continue")));
    driver.findElement( By.id("Continue")).click();
    wait.until(ExpectedConditions.titleIs("Fuel Checker"));
}
```

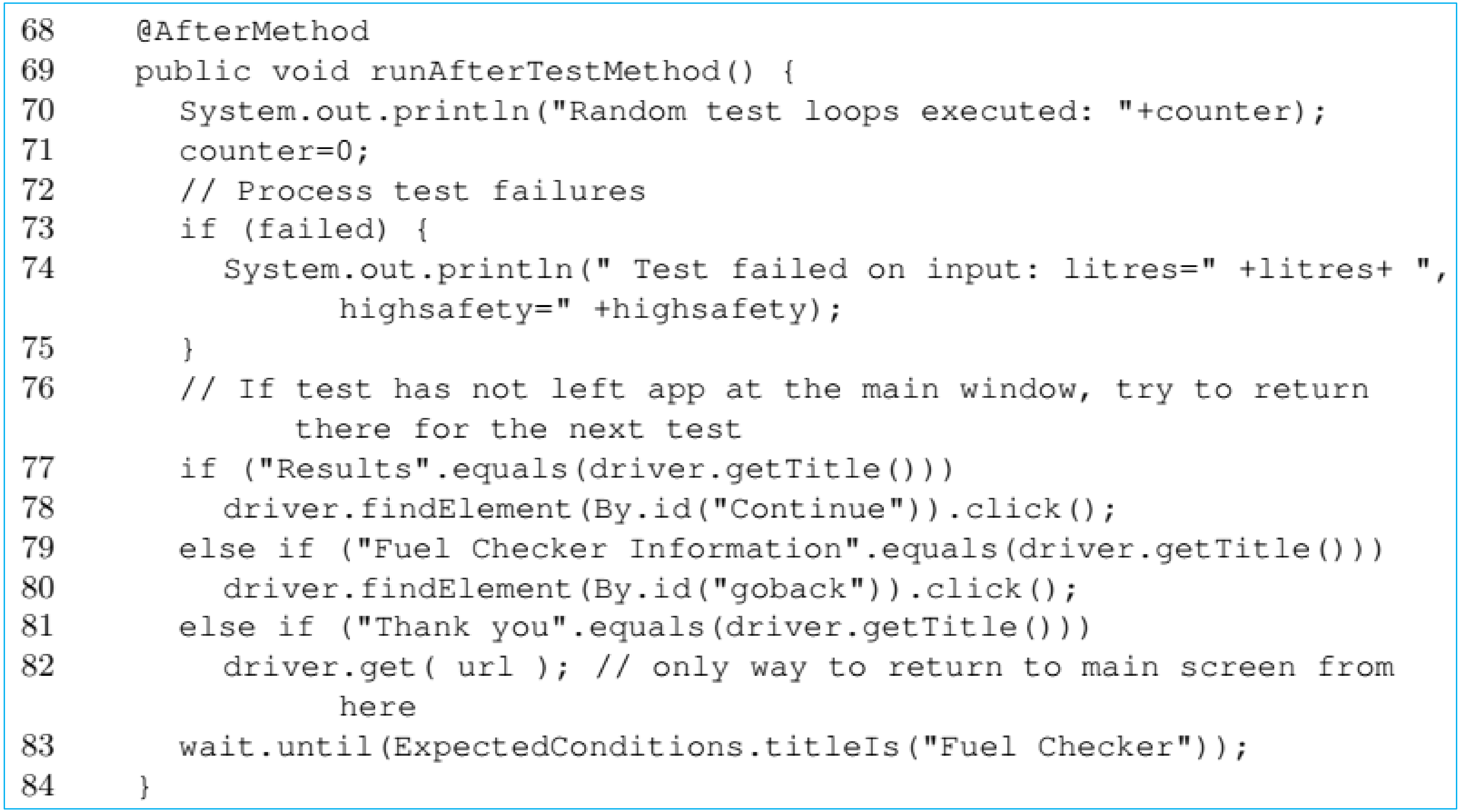
The Test



```
        // Now do the test
        wait.until(ExpectedConditions.ti
        wait.until(ExpectedConditions.vi
                By.id("litres")));
        driver.findElement(By.id("litre
        wait.until(ExpectedConditions.
                By.id("highsafety")));
        if (driver.findElement(
                By.id("highsafety")).isSelected()!=hig
           driver.findElement( By.id("highsafety")).click();
        wait.until(ExpectedConditions.visibilityOfElementLocated(
                By.id("Enter")));
        driver.findElement( By.id("Enter")).click();
        wait.until(ExpectedConditions.titleIs("Results"));
        wait.until(ExpectedConditions.visibilityOfElementLocated(
                By.id("result")));
        assertEquals( driver.findElement(
                By.id("result")).getAttribute("value"),result );
        wait.until(ExpectedConditions.visibilityOfElementLocated(
                By.id("Continue")));
        driver.findElement( By.id("Continue")).click();
        wait.until(ExpectedConditions.titleIs("Fuel Checker"));
    }
```

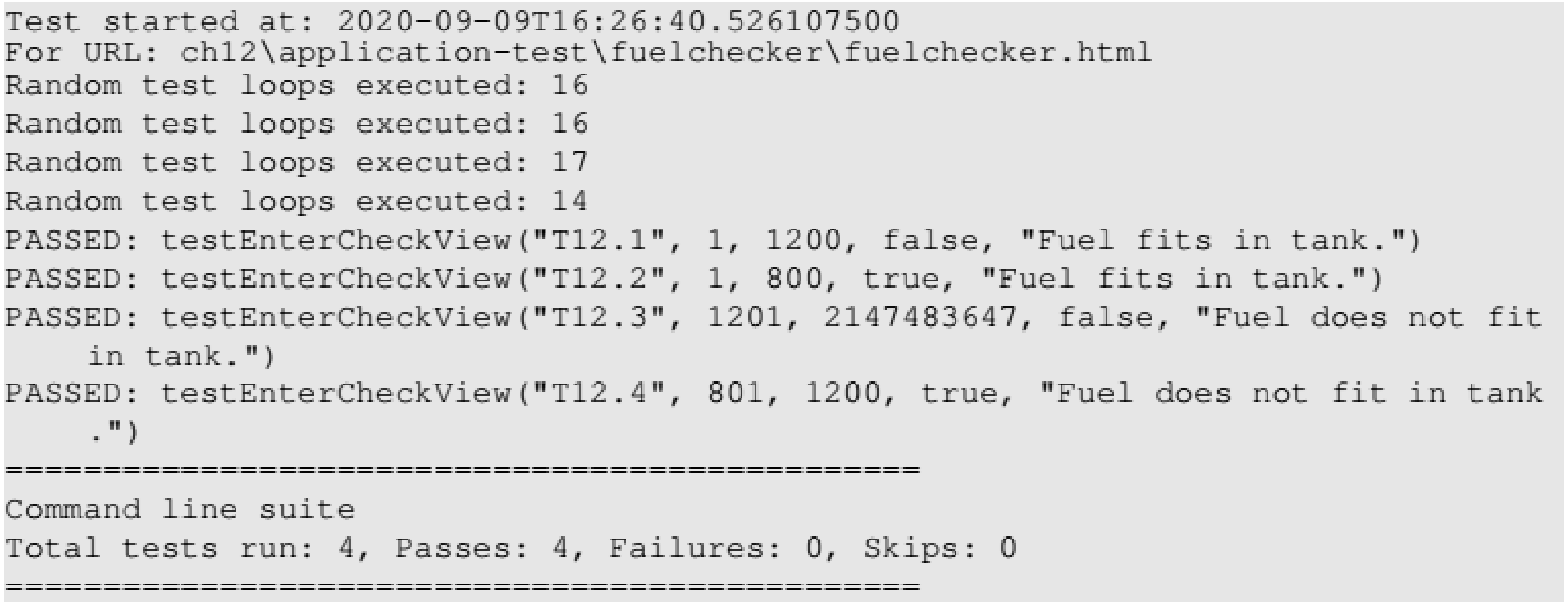
Note: almost identical to the test developed in application testing – just using randomly generated data values

@AfterMethod



```
@AfterMethod
public void runAfterTestMethod() {
  System.out.println("Random test loops executed: "+counter);
  counter=0;
  // Process test failures
  if (failed) {
    System.out.println(" Test failed on input: litres=" +litres+ ", highsafety=" +highsafety);
  }
  // If test has not left app at the main window, try to return there for the next test
  if ("Results".equals(driver.getTitle()))
    driver.findElement(By.id("Continue")).click();
  else if ("Fuel Checker Information".equals(driver.getTitle()))
    driver.findElement(By.id("goback")).click();
  else if ("Thank you".equals(driver.getTitle()))
    driver.get( url ); // only way to return to main screen from here
  wait.until(ExpectedConditions.titleIs("Fuel Checker"));
}
```

Test Results



```
Test started at: 2020-09-09T16:26:40.526107500
For URL: ch12\application-test\fuelchecker\fuelchecker.html
Random test loops executed: 16
Random test loops executed: 16
Random test loops executed: 17
Random test loops executed: 14
PASSED: testEnterCheckView("T12.1", 1, 1200, false, "Fuel fits in tank.")
PASSED: testEnterCheckView("T12.2", 1, 800, true, "Fuel fits in tank.")
PASSED: testEnterCheckView("T12.3", 1201, 2147483647, false, "Fuel does not fit
    in tank.")
PASSED: testEnterCheckView("T12.4", 801, 1200, true, "Fuel does not fit in tank
    .")
===============================================
Command line suite
Total tests run: 4, Passes: 4, Failures: 0, Skips: 0
===============================================
```

All tests passed

RUNTIME=10000, so total test time is approx. 40 seconds

Note: loop counters

Random Testing in More Detail

- Examined one simple type of random testing as an introduction: random input data selection with existing test cases
- Ideally entire testing process random, automated, & provide full coverage of software **specification** (BBT) and **implementation (WBT)**
- Would lead to very comprehensive and low cost testing
- Examine three barriers to full automation:
    1. Test oracle
    2. Test data
    3. Test completion

The Test Oracle Problem

- Once random data has been automatically selected, and the software executed with this data, the problem is how to verify that the output from the software is correct
- For unit tests, this may be the return value
- When testing object-oriented software, this may also include output attributes
- And for application testing, this includes all the output from the software, which may be delivered to the screen, to databases, to files, as inputs to another system, etc.

Four Fundamental Approaches

- Ignore the issue
  - Only requiring test completes without crashing – **stability testing**
  - The test oracle just verifies software keeps running after any test input
  - In principle, this can be applied to all types of testing, but it is usually applied to system testing
- Write software requirements in a higher level, abstract language
  - Use this to generate the expected results: an **executable specification**
  - Higher level language: program 'what' rather than 'how'
  - Alternative: a second programmer implementing the algorithm separately as a test oracle
- Use **constraints** expressed in a logic-based software specification language (e.g. OCL)
  - Instead of *calculating* the expected results, this approach *verifies the actual result* against the specification
  - Requires a formal specification, software tools, and developer skills in formal methods
  - Research example: Java Markup Language (JML), OpenJML Runtime Assertion Checking (RAC)
- Use manual test design techniques in order to develop **criteria**:
  - Criteria instead of data values, as shown in the worked example
  - EP, ST, SC, BC. BVA not likely to be as useful. Random values selected at runtime to match the criteria

The Test Data Problem

- The automated generation of test data : a complex problem
- Stepping stone on the road to full automation is random object generators
- Can be used in conjunction with manually developed test cases (for example equivalence partition), where a goal is selected at random, and then data is generated to match that goal
- This can also be used to address the test oracle problem, if the goal allows the expected results to be easily determined purely random selection of data values is unlikely to cover all the specification or all the implementation
- Research into directed random data generation has focused mainly on white-box coverage, as coverage measurement tools exist (e.g. DART: Directed Automated Random Testing).
- For application testing, data generation is often based on statistics of typical customer input data. This allows the tests to mimic actual software usage. As well as helping to find faults, this type of data generation allows the mean time to failure (MTTF) to also be estimated in advance of software release

The Test Completion Problem

- Ideally, automated random tests would cover all of the specification and all of the implementation, achieving 100% black-box and 100% white-box coverage
- Few (no?) tools exist to measure black-box coverage, and not all white-box coverage metrics can be easily measured (such as all-paths coverage)
- Approaches to solving this problem tend to be based on pragmatism:
  - Number of **loops** is the simplest completion criterion
  - **Time** is a more sophisticated metric - may lead to variations in coverage
  - Use existing **white-box coverage** tools. For example, run random tests until 100% statement and branch coverage have been achieved. This will often result in nonterminating tests, and is usually used in conjunction with time as a completion measure

Other Types of Random Testing

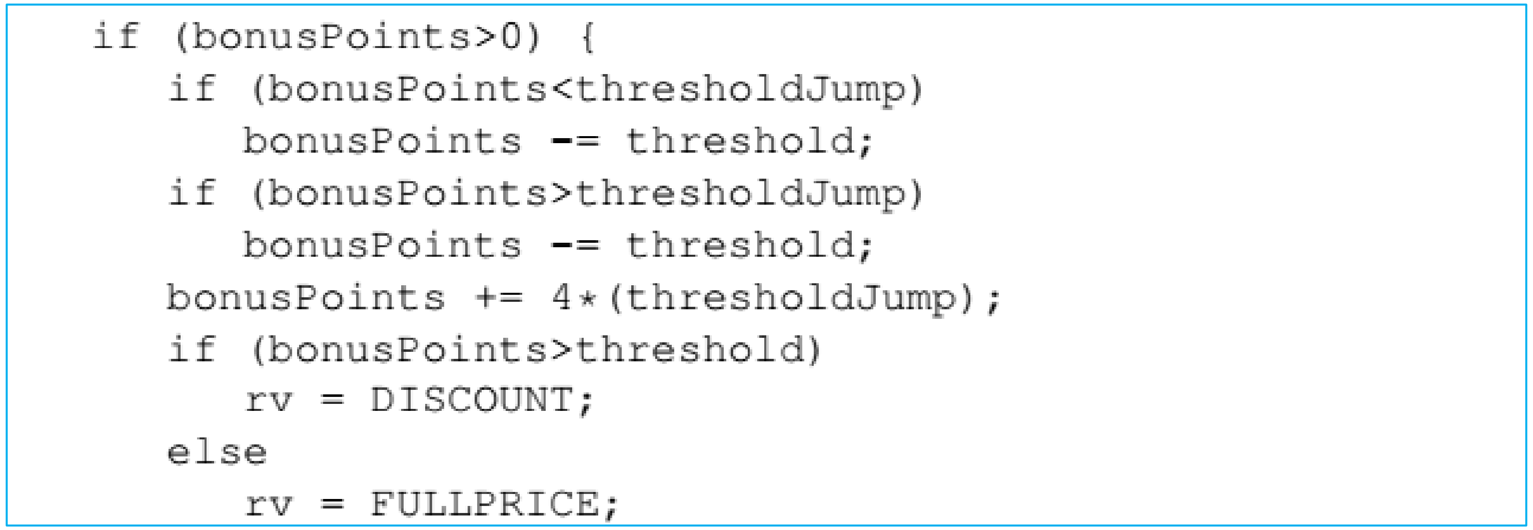
- **Stability Testing:** there are various forms that test a system to ensure that it does not hang or crash, for any input
  - **Undirected Random Testing:** the user interface is scanned for interactive components, one is selected at random, and then an interaction is selected at random (such as a click, text entry, swipe, and so on.) and executed. The test attempts to cause the system to crash. Normally, completion criteria based on time
  - **Directed Random Testing:** white-box coverage criteria, such as statement or branch coverage, are measured dynamically during testing. Once random tests stop producing additional coverage, then analysis of the code takes place, to try and identify input data that will increase the coverage. This may take the form of symbolic execution, or may be based on statistical or machine learning approaches. The goal is typically to maximise the code coverage, maximise the number of crashes, and minimise the length of the input sequences. It can be used for desktop, mobile, and web-based applications, though determining whether a web-based application has crashed can be challenging. The completion criteria are usually coverage and/or time-based, as it may take a long time to achieve the required coverage, even if it is possible
  - **Fuzz Testing**: existing good data is corrupted at random, and provided as input to the system. The goal is to crash the system with invalid input. The completion criteria is usually time

Other Types of Random Testing

- ***Stress Testing**:* the goal is to ensure the system continues to operate correctly when overloaded with inputs. Existing tests may be chosen at random and executed at the highest rate achievable, often running a large number of tests in parallel to maximise the load. This ensures that the system operates correctly, even when overloaded
- ***Decision Trees* and Random Testing**: the goal is to resolve the test oracle and data selection problems, and ensure that the system operates correctly for randomly selected inputs. The test oracle problem is solved by selecting a test *output* at random, and then using specification-based rules to select appropriate test *inputs*. This approach does not work for all forms of specifications, but is well suited to those where it is straightforward to run the specification backwards. A decision tree is used (i) to select the output at random, and then (ii) to implement the rules to create random inputs
- **Automated test case generation from specifications:** there is active research to develop test cases using various approaches such as: analysis, evolutionary algorithms, machine learning. This approach has the potential to solve all three random test problems, providing systems that can automatically generate test with no user intervention

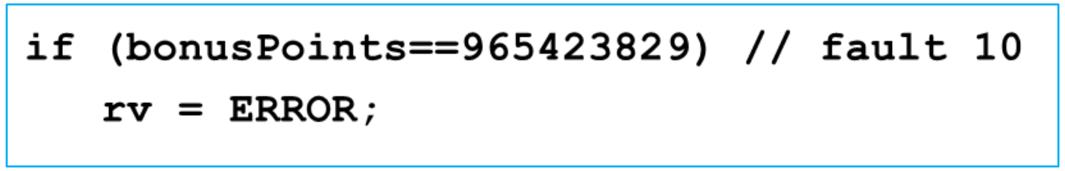
Evaluation

- Random EP Testing with Fault 6

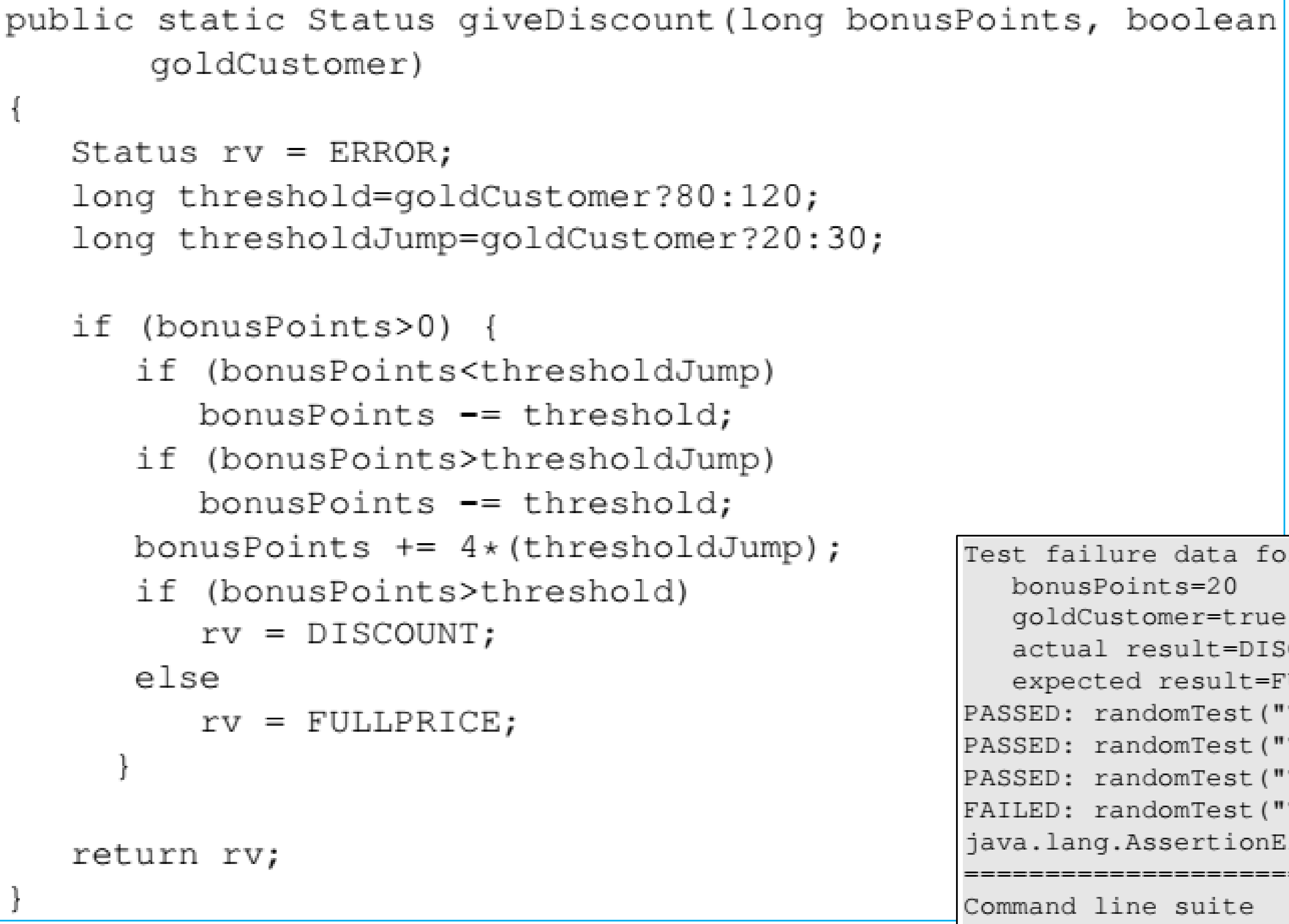


```
    if (bonusPoints>0) {
       if (bonusPoints<thresholdJump)
          bonusPoints -= threshold;
       if (bonusPoints>thresholdJump)
          bonusPoints -= threshold;
       bonusPoints += 4*(thresholdJump);
       if (bonusPoints>threshold)
          rv = DISCOUNT;
       else
          rv = FULLPRICE;
```

- Random EP Testing with Fault 10



```
if (bonusPoints==965423829) // fault 10
   rv = ERROR;
```

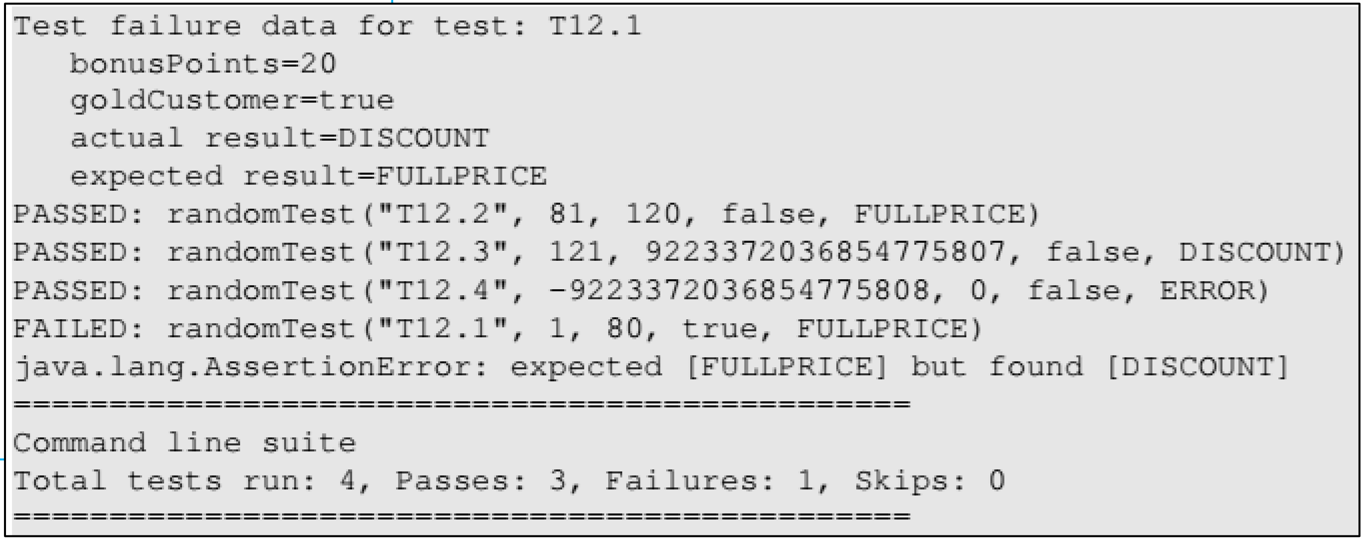


```
public static Status giveDiscount(long bonusPoints, boolean
      goldCustomer)
{
   Status rv = ERROR;
   long threshold=goldCustomer?80:120;
   long thresholdJump=goldCustomer?20:30;

   if (bonusPoints>0) {
      if (bonusPoints<thresholdJump)
         bonusPoints -= threshold;
      if (bonusPoints>thresholdJump)
         bonusPoints -= threshold;
      bonusPoints += 4*(thresholdJump);
      if (bonusPoints>threshold)
         rv = DISCOUNT;
      else
         rv = FULLPRICE;
     }

   return rv;
}
```

Fault 6



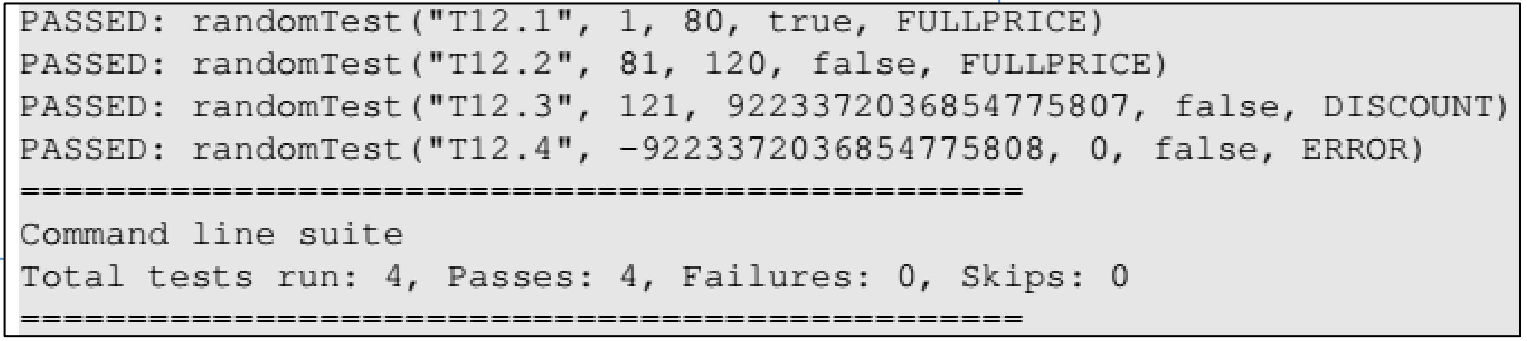
```
Test failure data for test: T12.1
   bonusPoints=20
   goldCustomer=true
   actual result=DISCOUNT
   expected result=FULLPRICE
PASSED: randomTest("T12.2", 81, 120, false, FULLPRICE)
PASSED: randomTest("T12.3", 121, 9223372036854775807, false, DISCOUNT)
PASSED: randomTest("T12.4", -9223372036854775808, 0, false, ERROR)
FAILED: randomTest("T12.1", 1, 80, true, FULLPRICE)
java.lang.AssertionError: expected [FULLPRICE] but found [DISCOUNT]
===============================================
Command line suite
Total tests run: 4, Passes: 3, Failures: 1, Skips: 0
===============================================
```

The fault is detected, and the input data values that caused the test failure are displayed

Fault 10

```java
 public static Status giveDiscount(long bonusPoints, boolean goldCustomer)
{
   Status rv = FULLPRICE;
   long threshold=120;
   if (bonusPoints<=0)
      rv = ERROR;
   else {
      if (goldCustomer)
         threshold = 80;
      if (bonusPoints>threshold)
         rv=DISCOUNT;
   }
   if (bonusPoints==965423829) // fault 10
      rv = ERROR;

   return rv;
}
```



```
PASSED: randomTest("T12.1", 1, 80, true, FULLPRICE)
PASSED: randomTest("T12.2", 81, 120, false, FULLPRICE)
PASSED: randomTest("T12.3", 121, 9223372036854775807, false, DISCOUNT)
PASSED: randomTest("T12.4", -9223372036854775808, 0, false, ERROR)
===============================================
Command line suite
Total tests run: 4, Passes: 4, Failures: 0, Skips: 0
===============================================
```

The fault is not detected

Very low probability: approx. 1 in 2^{63}

General Strengths & Weaknesses

- Random testing is used as an approximation of exhaustive testing
- By covering a wide range of values at random, rather than using a single value in each test, the probability of finding faults in the software is increased
- In this case, as equivalence partition testing has been used as the basis for randomisation, only faults that can be found by equivalence partition testing can be discovered
- It is unlikely that faults related to a single data value will be found, but faults related to a range of values, within an equivalence partition, have a higher chance of being found with random testing
- Using decision tables, statement coverage, or branch coverage test data increases the chance of each of these find faults, by using a wider selection of input values

Strengths & Weaknesses

- **Strengths**
  - Limited, or no, manual intervention is required to run the tests. This holds the potential for a significant improvement in software quality
  - A large number of tests can be run in a limited time
  - Stability testing has proven very successful in finding situations where software hangs or crashes, and has been an important element in improving this aspect of software quality
- **Weaknesses**
  - Most existing tools only perform stability testing, and therefore do not verify the correct operation of the software, which must be tested separately
  - Test oracle, data selection, and test completion problems are still largely unsolved
  - Effective techniques require full software specifications. These are time consuming to develop, and many developers are not experienced in producing these, using formal languages (e.g. OCL)

Key Points

- The three key problems: the test oracle problem, the test data problem, and the test completion problem
- Random data selection can be used with any of the black-box and white-box test techniques, by selecting test data criteria rather than test data values, and then selecting test data values at runtime that match the criteria. In this case the **test oracle** is manual
- **Fully random data** selection can be used with a simple test oracle (the software does not crash) for stability testing.
- There are a number of approaches for determining data **completion**: simple and pragmatic approaches consist of specifying a fixed number of loops, or a fixed amount of time for test execution. More advanced techniques include measuring code coverage at runtime

Notes for the Experienced Tester

- Generating random tests is probably one of the most effective ways for the experienced tester to create large numbers of tests quickly
- The stability of an application can be tested by generating random input actions and checking that the application does not crash
- The tests created using equivalence partition, boundary value analysis, decision tables, statement coverage and branch coverage for unit testing, OO testing, or application testing can be used to expand the breadth of data values used

Notes for the Experienced Tester

- By using input data which is statistically representative of real-world data, collected from execution of the application by real customers, an estimate of the mean time between failures (MTBF) can be derived
- Safety tests can often be developed by adding invariants to represent the safety criteria for an application or a class, writing code to check for these (periodically or after every method call), and simulating random inputs. Instead of checking that the output of each method is correct, this form of testing verifies that the effect of the method is correct with respect to the safety criteria

This Afternoon’s Lab

- Random Unit Testing
- Using Combinations/Decision Trees
- And a static method
- Assessment quiz