Billing2

Starter code: **AmicaJavaTraining/Projects/Billing/Billing1**

Submit as: **CompletedWork/Billing/Billing2**

In this workshop you will develop unit and integration tests for an implementation of the Billing project. In order of priority, create:

1. Unit tests for the **CSVParser**.
2. A unit test for the **ParserFactory**.
3. A unit test for the **Billing** class.
4. An integration test for the **Billing** class.
5. A unit test for the **Reporter** class.
6. An integration test for the **Reporter** class.

Each of these will highlight one or more testing techniques, and those are summarized at the top of each section, below.

Start by making a local copy of the starter code, and create a Maven project in your IDE of choice. The POM is already defined with JUnit, Hamcrest, and Mockito, and the Maven Surefire plugin is configured to recognize JUnit-5 annotations.

Specific behaviors to be covered by tests are listed in detail below. Some refactoring of the existing classes will be necessary to support unit testing, and this is explained below as well.

Review the **com.amica.billing.TestUtility** class before creating your tests. It holds a number of useful constants and helper methods that will save you some time and trouble.

This is an individual workshop, so when you're done you can copy your **src** tree an **pom.xml** to a new **Billing/Billing2** folder on your **CompletedWork** repository, and push it to the remote repository.

CSVParserTest

- *Isolation using streams and collections of strings and objects*

TestUtility defines test data sets, as lists of **Customer** and **Invoice** objects. The starter code for **CSVParser** defines lists of strings that align with those data sets: **GOOD_CUSTOMER_DATA** for **GOOD_CUSTOMERS**, for example. By "bad" customers or invoices, we mean data that includes some unparseable lines, and the corresponding data sets include only the objects that should survive and be reported by the parser.

Give the class a field **parser** and initialize it to a new **CSVParser** instance.

Write test cases that prove that the good and bad data sets can be parsed accurately. Note that **TestUtility** also provides a **GOOD_CUSTOMERS_MAP**, so you don't need to cook that up, just pass it as the second argument to **parseInvoices** when testing that method. Parsing the "bad" data sets should not throw any exceptions; the parser should weed out the unworkable lines and report a stream of objects from the lines that could be parsed.

Then write test cases to prove that the parser can produce streams of strings from the streams of objects. A little canonicalization will be needed here, because the invoice data is intentionally inconsistent in how it provides the fractional part of a whole-dollar invoice amount, so we can test how the parser handles that; but that means that the produced streams won't match up perfectly. Do a simple string replacement to get the actual and expected strings to match up, while still asserting the correctness of the string overall.

ParserFactoryTest

- *Fixing the class under test as new tests expose limitations and bugs*
- *Using a mock object to test correct use of a Supplier<T>*

Write this test class from scratch. You don't need a target object, because **ParserFactory** is a class utility – everything you want to test on it is static. Create test methods that prove that you get the right kind of parser back for a given filename – just use the Hamcrest **instanceOf** matcher and pass the expected class, as in **CSVParser.class** or **FlatParser.class**.

Test for the requirement that the parser factory checks file extensions in a case-insensitive manner. This requirement was not stated before! so the factory will fail the test. Improve the code so that it passes.

The parser factory allows programmatic configuration of its map of file extensions and parsers, and supports a default parser to be used when there is no extension or the extension is not found in the map. To test these features accurately, you'll need a type of parser not already known to the factory. You could create a **MockParser** class for this purpose, but a Mockito mock will be simpler. Give your test class a field **mockParser**, of type **Parser**, and initialize to a **mock** of the **Parser.class**. Give the class a second field **mockParserFactory**, of type **Supplier<Parser>**, and initialize to a lambda expression that returns the **mockParser**.

Now you can write test cases that call **addParser** or **replaceParser**, passing different extensions and supplying **mockParserFactory** as the mapping. Then call **createParser** and pass your expected extension. The **instanceOf** matcher won't cut it here, but you can simply use **equalTo** since there is only one instance of your mock parser.

Be sure to test for error conditions, too. Note that **addParser** rejects attempts to overwrite the mapping for an existing extension, while **replaceParser** will only overwrite, and will reject attempts to create new mappings.

BillingTest

- *Managing input and output files in the Maven folder structure*
- *A custom matcher to simplify testing the content of returned streams*
- *Checking an output file by looking for some expected content, anywhere in the file*
- *Using a mock to verify outbound notifications*

Because it is a key part of the responsibilities of the **Billing** class to open, read, and write files, we will not try to mock the inputs or outputs, nor refactor the class to take **Reader** and **Writer** objects or streams as alternatives to working with the file system. It's within our powers in writing unit tests to use the local file system – staying within the bounds of **src/test/resources** for any required resources – and the resulting tests will not be painfully slow such that they'd be a drag on an automated build.

The starter code for this test class defines a **SOURCE_FOLDER** and provides a **@BeforeEach** method that cleans out a **TEMP_FOLDER** (defined in the utility class) and copies the data files into it. Note that these are not the ones used in the previous Billing exercise; they have the same format, but align with the test data defined in **TestUtility**. This way, even when you test methods that can change the data, every test will get a clean starting file set. The method also creates an **OUTPUT_FOLDER** to hold report files generated during testing.

In **TestUtility.java**, see the **hasNumbers** method. This taps into the **HasKeys** matcher that we saw in use in the **HelpDesk** application: by providing an **Invoice**-specific key extractor, we're able to build a matcher that can check if the numbers of the invoices in a given stream are as expected, in sequence.

Give the class a field **billing**, of type **Billing**, and initialize it in the **@BeforeEach** method to a new object based on two filenames. The customers filename is the **TEMP_FOLDER** (defined in **TestUtility** as "temp") plus a slash, plus the **CUSTOMERS_FILENAME** (also defined in the utility class). The invoices filename is similar, just using **INVOICES_FILENAME**.

Write a test for **getInvoicesOrderedByNumber**: call the method and assert that it **hasNumbers** 1, 2, 3, 4, 5, and 6. **hasNumbers** takes a variable parameter, so you can actually pass those six values directly to it, separated by commas.

You can write a similar test for **getInvoicesOrderedByDate**, with the expected invoices numbers now in the sequence 4, 6, 1, 2, 5, 3.

getInvoicesGroupedByCustomer will be a little more work: call the method and capture the map as a local variable. Then, you can **get** the list of invoices for the first element in **GOOD_CUSTOMERS**: get a stream from that, and pass to **invoiceNumbers** and assert that the list **contains** just the invoice number 1. Do the same for the next two customers, expecting the second customer to have invoices 2, 3, and 4, and the third to have invoice numbers 5 and 6.

Write a test to prove that **getOverdueInvoices** for an **asOf** date of **TestUtility.AS_OF_DATE** returns invoices with numbers 4, 6, and 1.

Now you'll test out the updating methods. Write a test for the **createCustomer** method that passes it a new customer with hard-coded names and term. Rather than trying to compare the complete content of the updated data file, you'll just sift through it and confirm that the new customer is in there somewhere. Open the file using **Files.lines** and **assertThat** at least one line in the stream is the correct CSV representation of your customer.

Take a similar approach to testing **createInvoices** and **payInvoice**.

One missing piece thus far is that we don't prove that the **Billing** object is sending notifications to registered listeners. Back in the **@BeforeEach** method, define a mock **Consumer<Customer>** and a mock **Consumer<Invoice>**, and register them as listeners with the target object. Then add calls to **verify** to assure that notifications are being sent, at the bottom of each of the methods that test an updating method.

Testing **getCustomersAndVolume** or **getCustomersAndVolumeStream** will require some trench work, and we leave this as optional; you may want to move ahead to the next section. To test **getCustomersAndVolume**, capture the returned map, and assert the correct value for each of the customer and volume elements, one by one. For the stream version of the method, the customers should appear in the reverse of the order in which they're defined in **GOOD_CUSTOMERS**, and their volumes should be 1100, 900, and 100, respectively. Because these are **double**s, you'll need to use the **closeTo** matcher, rather than **equalTo**.

BillingIntegrationTest

- *Synthesis of unit- and integration-testing techniques*

You'll build this in large part by copying pieces of **BillingTest**. Create the class **BillingIntegrationTest** and give it a **SOURCE_FOLDER** with a simpler value: just "data", so that you'll pick up the CSV files used in the previous exercise.

Copy in the **invoiceNumbers** method – or, use a static import to tap into the one in **BillingTest**.

Copy the **billing** field, and the **@BeforeEach** method, so you'll have the same temporary and output folders put in place, but with the files from the **data** folder instead of from **src/test/resources/data**. You'll need to get rid of the code that sets up mock listeners.

Copy in your test of the **getInvoicesOrderedByNumber** method, and run it, unchanged. Of course it will fail, because your expectations of the invoice numbers don't align with the "real" data set. Then adjust the expected invoice numbers: this will be a long sequence of numbers from 101 through 124, with only 120 missing. Run again and see test success.

The invoice numbers ordered by date are below; use these for your test of **getInvoicesOrderedByDate**.

```
101, 102, 103, 104, 110, 105, 106, 107, 109, 111, 112, 113, 108, 114, 115, 116,
117, 118, 119, 121, 123, 122, 124
```

A test of **getOverdueInvoices**, as of January 8, 2022, should expect these invoices:

```
102, 105, 106, 107, 113, 116, 118, 122, 124
```

We pounded on the test for **getInvoicesGroupedByCustomer** method pretty hard in **BillingTest**, and with this larger data set we'll go a little lighter: call the method, get the list of invoices for just one customer, and check that. The invoices for Jerry Reed would be just 109 and 122.

Your test of **getCustomersAndVolume** can do more of a spot-check, as well. The highest-volume customer should be Jerry Reed, with a volume of (or **closeTo**) 2640.00, and the lowest-volume customer should be Janis Joplin with a volume of 510.00.

For the tests of methods that make changes to the data, you can whip up whatever customers and invoices you want, and pay any unpaid invoice. Check the results the same way you do in **BillingTest**. No need to build the tests that check error handling on these methods.

ReporterTest

- *More complex mock to isolate the test*
- *Checking output files by their entire content, verbatim*
- *Using an ArgumentCaptor<T> to worm into a private observer method*

There are two important aspects of our strategy for testing the **Reporter** class. One is, for a unit test, we're going to want to avoid dependency on the **Billing** class, as this would bring in a lot of complexity and our tests would rely not only on the logic of that class but on the data files supplied to it. So we're going to create a mock of this object that supplies query results to the **Reporter** that we can control.

We're also going to do more direct comparison of its output to whole files that represent expected report content. See these in **src/test/resources/expected/unit_test**: one for each type of report.

The starter code for **ReporterTest** includes an **EXPECTED_FOLDER** constant that identifies that folder; an **assertCorrectOutput** that compares versions of a given filename in the expected and actual-output folders; and a **@BeforeEach** method that initializes various collections to represent the test data set. Notice that the **CustomerAndVolume** objects require mocking: if we try to use the actual class, we'd need to use a real **Billing** object and prime it with customers and invoices, etc., so pulling that one thread would unravel the whole mocking approach.

Define a **mockBilling** field, of type **Billing**, and a **reporter** of type **Reporter**. Initialize **mockBilling** at the bottom of the **@BeforeEach** method, to a Mockito mock. Stub out each of its query methods to return the appropriate prepared stream or collection. For example **getInvoicesOrderedByNumber** can return **GOOD_INVOICES.stream()**, and **getInvoicesGroupedByCustomer** can return the **invoicesByCustomer** variable. Then initialize **reporter** to a new **Reporter**, passing the **mockBilling** object and the constants **OUTPUT_FOLDER** and **AS_OF_DATE**.

Now you can create tests for the four report methods: each one can call the target method on **reporter**, and pass the expected report filename to **assertCorrectOutput**. For example when testing **reportInvoicesOrderedByNumber** you can pass **Reporter.FILENAME_INVOICES_BY_NUMBER**.

Testing the reporter's "observation" of the source billing object is a bit tricky, for one reason: the **onInvoiceChanged** and **onCustomerChanged** methods are **private**. Otherwise we could simulate change events easily by calling those methods: it's not important who calls them, so a test can essentially act as an event source by calling a listening method.

But we need a way to invoke those private methods, and of course a real **Billing** object has that, by way of its **add***Listener** methods. So this is a job for Mockito's argument-captor feature. In the **@BeforeEach** method, call **ArgumentCaptor.forClass**, passing **Consumer.class**, and use this to initialize a variable **customerCaptor**. (You'll get "unchecked" type warnings on this usage, and it's unavoidable. You can apply a **@SuppressWarnings("unchecked')** if you like.)

Give the test class a field **customerListener**, of type **Consumer<Customer>**. Call **verify**, passing **mockBilling**, and on that verifier call **addCustomerListener**. To this method, pass the results of calling **customerCaptor.capture**. Then set **customerListener** to the results of a call to **customerCaptor.getValue**. Now you have a **Consumer** that is backed by the reporter's **onCustomerChanged** method, and this will be your way to trigger events to the reporter. You can do the same trick for the invoice listener.

Create a test method that calls the customer listener – and you can just pass **null** for the argument, since we know the listener ignores it. Then check the customers-and-volume report, the same way you do in the test for **reportCustomersAndVolume**.

A similar test for the invoice listener would check all four reports, as they should all be re-generated on a change to an invoice.

ReporterIntegrationTest

- *One test's target object is another test's supporting object (a/k/a fixture)*
- *Seeking to text lines of interest so as to verity their content*
- *Checking output files by their entire content, with canonicalization*

In a way, this test is the easiest to set up, because you are assembling the "real" system as it would work in an application. There's no need for test isolation, and so no need for mock objects or other tricks. Just create a **Billing** object and a **Reporter** object and let them run.

The implementation of this test is left as a final challenge to end the workshop exercise. But, a few tips:

- There are prepared, expected reports in **src/test/resources/expected/integration_test**.
- You've got some nice setup logic for a **Billing** instance in your **BillingIntegrationTest**, so it would be nice to re-use that. Not as simple as having one **@BeforeEach** method call the other, but you could break out a lot of the useful pieces to one or more utility methods.
- Most of the reports are predictably ordered, and so asserting that the reporter produces exactly the right output is straightforward. Invoices grouped by customer does not insist on an ordering of the customers, and so you can get some unpredictable output there. A strategy for this is to check for one or more customers individually in the report: seek to the customer name as the content of a text line, and then read the successive, non-blank lines, and that fragment of the file should be as expected, even if you can't guess where it will be in the file. (Stream API lovers, look at **dropWhile** and **takeWhile** as an especially tidy way to do this.)
- To test that the reporter generates reports in response to changes on the **Billing** data set, you'll need to create your own expected-output files that include the modified data – new customer or invoice, paid invoice, and whatever that changes about aggregates such as customer volume.
- In doing so, you'll need to find a way to ignore the date on which you're testing in the file content. For example an invoices-by-number report that includes a new invoice and checks its issue date, verbatim, will pass today and fail tomorrow! A good strategy for this is to use a placeholder such as **@TESTDATE@** instead of the explicit date, and then have logic in your test to replace that with the date of the test.