Capstone Courseware, LLC
1 Washburn Place
Brookline, MA 02446
877-227-2477
capstonecourseware.com

Billing3

Starter code: **AmicaJavaTraining/Projects/Billing/Billing2**

Submit as: **CompletedWork/Billing/Billing3**

In this workshop you'll make the Billing application configurable with respect to the files, folders, and formats it uses. You'll refactor three classes – **Billing**, **ParserFactory**, and **Reporter** – to offer an option to use the Amica configuration manager, and develop additional tests to prove out the behavior of each when configured.

Make a copy of the **Billing2** project, do your work there, and when done you'll push your work to a **Billing3** folder on your CompletedWork repository.

Maven Dependencies

1. To incorporate the configuration manager in the project, first add our parent POM to the project POM. You can remove the project's own <**groupId**>, because this will now inherited from the parent. Copy this from one of the course projects – or, if you copy it from this PDF, double-check, because the hyphens may not be hyphens anymore …

```
<parent>
  <groupId>amica</groupId>
  <artifactId>amica-versions-bom</artifactId>
  <version>0.4.7</version>
</parent>
```

```
  <groupId>amica</groupId>
  <artifactId>Billing</artifactId>
  <version>@STEP@</version>
  <name>Billing</name>
```

2. Now add the dependencies for the configuration manager and for properties-based configuration:

```
<dependency>
  <groupId>com.amica.esa</groupId>
  <artifactId>component-configuration-manager</artifactId>
  <version>0.0.6</version>
</dependency>
<dependency>
  <groupId>com.amica.escm</groupId>
  <artifactId>properties-configuration</artifactId>
  <version>0.0.1</version>
</dependency>
```

Billing

3. Open the **Billing** class and add constants for the configuration name (which will be "Billing") and names of properties we'll read for each of the customers and invoices files ("Billing.customersFile" and "Billing.invoicesFile").
4. Add a constructor that takes a **com.amica.escm.configuration.api.Configuration** object. Have it call the existing constructor, passing the results of calls to **getString** on the configuration, reading the two filename properties from it. This constructor, then, would support creation of a **Billing** object based on a configuration – as loaded from the configuration manager, or cooked up based on a **Properties** map, or any other implementation.
5. Add a third constructor that takes no arguments. Implement it to call the **Configuration-based** constructor, passing the Billing configuration as loaded using the configuration manager.

   Note that, of the three ways we can now instantiate a **Billing** object, only one of them will try to get a configuration-manager instance. So it's safe for existing tests to create the object using the pre-existing constructor, even if there are no configuration files and the configuration manager hasn't been initialized. The no-argument constructor, by contrast, will only work if the configuration-manager infrastructure is in place.
6. Run the existing **BillingTest** and **BillingIntegrationTest** classes, to be sure you haven't broken anything.
7. Created a new **BillingPropertyTest** class and make it extend **BillingTest**. Notice that the **BillingTest** class relies on a few helper functions, to locate the customers and invoices files, and to create the object that will be tested.
8. Override **getCustomersFilename** to return "customers_configured.csv", and do the same for the invoices filename. These two files are provided in the starter code, and they are exact copies of the ones used by **BillingTest**, so if we can create a **Billing** object that uses them, all the base-class test cases should still pass. (**BillingTest** already uses the helper methods to determine what files to copy into the **TEMP_FOLDER**, so they'll be in place when the **Billing** object asks for them.)

9. Override **createTestTarget**. In this override, create a new **Properties** object, and put the two file locations that you want the **Billing** object to use in this map. For example, under the key "Billing.customersFile", concatenate the **TEMP_FOLDER**, a slash character, and the results of a call to **getCustomersFilename**.
10. Now create a new **Billing** object, passing a new **PropertiesConfiguration** object based on your map, and return that object.
11. Run the test, and it should pass. So that proves out your **Configuration**-based constructor.
12. Set up the necessary folders for the configuration manager, under **src/test/resources**. Anticipate an **env.name** value of "Configured" and use any component name you like.
13. In the **EnvironmentResources/Configured** folder, create a **Billing.properties** file, and in that file set the two filenames the same way that you do in the properties-based configuration – but you can just write out the file paths, for example "temp/customersFile".
14. Create another test class, **BillingConfiguredTest**, initially as a copy of **BillingPropertyTest**.
15. Simplify the override of **createTestTarget** to return a new **Billing** object, using the no-argument constructor.
16. Add a **@BeforeAll** method that sets the **env.name** system property to "Configured". Then get the instance of the configuration manager and **initialize** it.
17. Run the test, and it should pass.

ParserFactory

Note that there is a new parser, **QuotedCSVParser**, that can read a slightly different format: a CSV file where the string values are surrounded by double quotes.

18. Open **ParserFactory**, and notice the helper method **createParserWithClassName**. This will translate the full class name of any of the known parsers into an instance of that class. It is implemented in a way that (a) creates compile-time dependencies on all of the parser classes, and (b) doesn't scale well if more parser classes are created. But it will do for now; the answer code replaces this with an implementation that uses the Java Reflection API.
19. Add code to the bottom of the **resetParsers** method. Start by checking if **env.name** is defined. If it is not, **System.getProperty** will return **null**, in which case don't do anything more. This will assure that the class will continue to work as it did before if the configuration manager is not active.
20. If there is an **env.name** setting, then get an instance of the configuration manager and use it to get the "Billing" configuration.
21. Call **getKeys** on the configuration, and loop through the keys, looking for keys that start with "ParserFactory".
22. If you encounter the "ParserFactory" key itself, create a lambda expression that takes no parameters. In the body, pass the value for that key to **createParserWithClassName**. Make that lambda expression (which is a **Supplier<Parser>**) the new default parser (that is, **put** it in the **parsers** map under the key **null**).
23. IF you find a key of the form "ParserFactory.*ext*", derive the part after the dot as the file extension for which you're going to register a new parser. Again, build a lambda expression that returns the results of passing the value for this key to **createParserWithClassName**, and **put** it as the value for this extension.

24. Create a second environment subfolder in your test resources, called “Quoted”.
25. Copy the **Configured/Billing.properties** file into this folder.
26. Add properties to the new properties file as follows:

```
ParserFactory=com.amica.billing.parse.FlatParser
ParserFactory.csv=com.amica.billing.parse.QuotedCSVParser
```

27. Create a class **ParserFactoryConfiguredTest**.
28. Give it a **@BeforeAll** method that sets the **env.name** to “Quoted” and initializes the configuration manager.
29. Write a test method that proves that the default parser is now the **FlatParser**. Run the test and see that this method passes.
30. Write a test method that proves that the **QuotedCSVParser** will be returned for a filename ending in “.csv”. See that this method passes as well.

Reporter

31. Enhance the **Reporter** class so that its **outputFolder** and **asOfDate** are both configurable. You can follow a strategy very much like what you did with the **Billing** class, with constants for the configuration and property names and two new constructors.
32. We'll switch over to integration testing for this one. We'll test a scenario in which the **Billing** object is fed a file in the new quoted-CSV format and the **Reporter** is directed to a different output folder.
33. We can use the existing "Quoted" environment for this: just open that version of **Billing.properties** and add values for the two new properties on **Reporter**. Set the output folder to "alternate" and the as-of date to January 8, 2022 (the same one we've been using in other tests, since so much of the existing test logic depends on that).
34. Create a **ReporterConfiguredIntegrationTest** class, as a subclass of **ReporterIntegrationTest**.
35. Give the class a **@BeforeAll** method that sets **env.name** to "Quoted" and initializes the configuration manager.
36. Override the **getOutputFolder** method to return "alternate" so the base class will use this folder when asserting correct report output.

37. It would be nice to keep doing these short, surgical overrides of base-class helper methods. Alas, the setup here is too involved for us to influence it from the derived test class with just a word here or there. You'll need to override the **setup** method and put everything together yourself.
38. Annotate the override as a **@BeforeEach** method, as well as an **@Override**.
39. Copy the full implementation of the base-class method into your new method. The middle of this code is still useful, but we're going to need to set up files differently, and then we're going to need to create the test fixtures differently.
40. Replace the call to **BillingIntegrationTest.setUpFiles** with the code from that method. This code will create the **temp** folder, as most of our tests do; and then modify the two calls to **Files.copy** so that they copy the **customers_quoted.csv** and **invoices_quoted.csv** files from the **data** folder into the **temp** folder.
41. Since the base class wants references to both the **Billing** and **Reporter** objects, we'll do a little trick. Remove the last two lines, that create the two objects. Instead, set **reporter** to a new **Reporter**, passing no arguments so it uses the configuration manager. Then, call **reporter.getBilling**, and set **billing** to the returned object reference.
42. Run your test class, and it should pass. When it does, we've proved out a scenario where the **Billing** object loads the quoted-CSV files; the **ParserFactory** returns an instance of the new **QuotedCSVParser** that can parse them; and the **Reporter** produces reports to the **alternate** folder as configured. The test data is the same, so all of the base-class test cases drive the same behavior, and find that the same expectations are met.