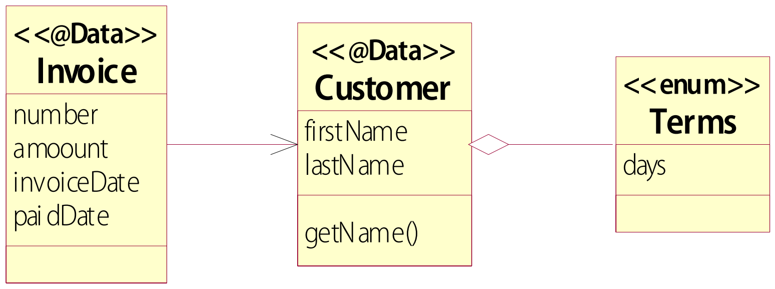
Billing1 – Implementation

Domain Model

The domain model is clearly indicated by the data in the given files:



The **Customer** holds payment terms using an enumerated type **Terms** – and this holds an associated attribute **days** that will be useful in calculating overdue invoices, since each possible value can be reduced to a number of days in which to pay, with **CASH** being zero. **Customer** also offers a **getName** helper, and the full name of the customer will be used as a unique key.

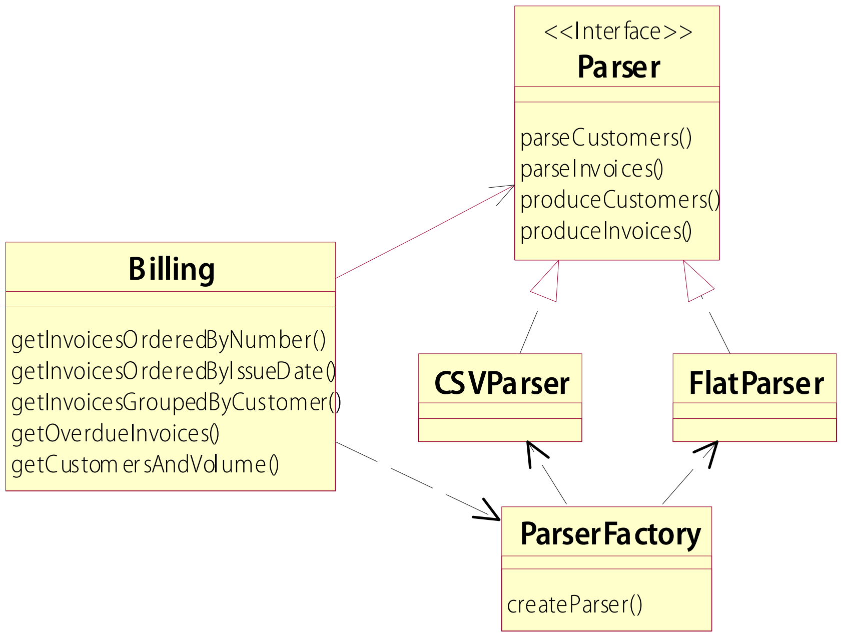
The **Invoice** holds invoice data, including number, amount, and dates. Use **java.time.LocalDate** for the dates, and use an **Optional<LocalDate>** for the **paidDate**.

Invoice relies on a reference to a **Customer**. So it will be necessary to parse customers first, and to compile that data into a **Map**; and then to have reference to all of the customers when parsing the invoices. That way, the first and last name of the customer can be translated to an existing customer object.

Design Patterns

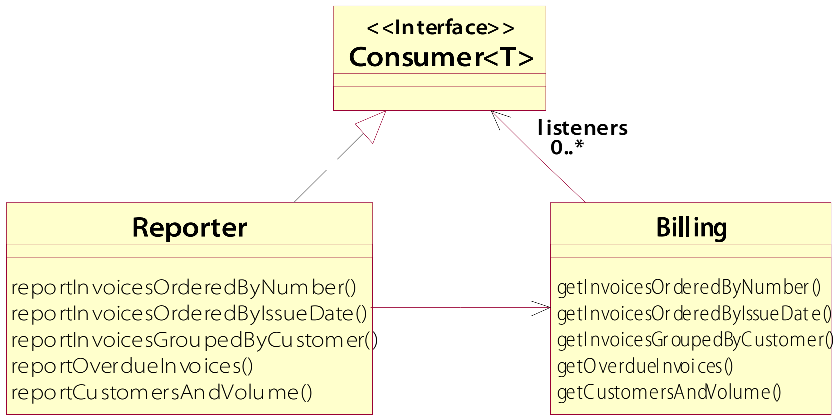
We can identify a few design patterns that apply to the Billing problem:

- **Strategy**: the need to parse information in different formats to a common model can be supported neatly by a strategy interface, which we'll call **Parser**, and multiple implementations including **CSVParser** and **FlatParser**. A main **Billing** class is the context for this strategy, and will call parsing methods to get its data.



- **Factory**: it will be worthwhile to isolate the logic for deciding which parser implementation to create, in a **ParserFactory** class. This keeps the Billing class free from that decision-making and from any dependencies on implementation types.

- **Observer:** a **Reporter** class will offer the four required query methods, and in order to keep up with changes to the data, this class will implement callback or "observer" interfaces defined by the **Billing** object, and the **Billing** object will notify all registered observers by calling methods on those interfaces. The diagram below shows this solution a bit vaguely: there will actually be two uses of **Consumer**, one where **T=Customer** and one where **T=Invoice**, and **Billing** will keep two distinct collections of listeners, one for each type. **Reporter** will not directly implement this interface, either; instead it will define methods that take customers or invoices as parameters, and will register listeners using Java-8 method references.


<<Interface>>
Consumer<T>
listeners
0..*
Reporter
reportInvoicesOrderedByNumber()
reportInvoicesOrderedByIssueDate()
reportInvoicesGroupedByCustomer()
reportOverdueInvoices()
reportCustomersAndVolume()
Billing
getInvoicesOrderedByNumber()
getInvoicesOrderedByIssueDate()
getInvoicesGroupedByCustomer()
getOverdueInvoices()
getCustomersAndVolume()

Development

Here is a rough development plan, including a few tips on how to implement various parts of the design:

1. Build the domain model, in package **com.amica.billing**. Notice that we've specified the Lombok **@Data** annotation as a stereotype for the **Customer** and **Invoice** classes, so take advantage of that. You may also want the **@AllArgsConstructor**.

The **Terms** enum will define a private field **days**, a private constructor that takes a value for this field, and a public getter method for it. Then you can assign the appropriate number of days to each of your enumerated values. Notice that one of the input formats encodes the payment terms for each customer with symbols such as **CREDIT_45**: if you adopt these as your enumerated values, it will be easier to parse that format using **Terms.valueOf**.

2. **com.amica.billling.parse.Parser** is provided, so review the four methods there.

3. Implement each of your parsers, working from starter code provided in **com.amica.billing.parse**. The starter classes already know the specifics of each format and can parse one line of text in that format to a set of local variables of type string, integer, double, or **LocalDate**. See the **//TODO** comments and fill in code that converts the terms string to an enumerated value, creates and returns a customer, creates and returns an invoice, etc. You don't need to worry about the formatting methods just yet.

4. Then make each class implement the **Parser** interface, and implement the two parsing methods. The strategy for each of these is simply to **map** elements of the given string stream to objects of the required type, using the corresponding method that parses a single line. You can just return **null** from the two producing methods for now.

5. Look at the test programs in **com.amica.billing.test**. They are almost identical, but each tests using one of the formats: **TestCountrySingers** reads the CSV files, and **TestMovieStars** reads the flat-format files. In each, the **main** method prepares some copies of the test data in local folders, so you have a clean start each time, and then calls three test methods. Each of those methods has key logic commented out, because it wouldn't have compiled at the start of the exercise. Try commenting out the code in test #1 now, and see that your parser can read short streams of prepared data and return streams of correctly-initialized objects. The test logic is not comprehensive, and you may want to add to it, but it should be a good spot-check before moving ahead.

6. Create the **ParserFactory**. The static **createParser** method will look at a given filename, and either create a **CSVParser** or a **FlatParser**, based on the filename's extension. This is a small bit of logic, but using it from the **Billing** class will establish an important point of extensibility into your system.

7. Create your **Billing** class, in **com.amica.billing**, and give it several fields. It should store the customer and invoice filenames; it should keep reference to a **Parser**; and it should hold the loaded customer and invoice data. Invoices can be held in a **List**, and customers, since we'll need to look them up a lot as referenced by invoices, can be held in a **Map** with a **String** as the key and the customer itself as the value.

8. Define a constructor that takes the two filenames. Store them, and pass either one to **ParserFactory.createParser**. Store the result as your selected parser. Now, open the customers file and get a stream of the lines of the file. Pass that to **parseCustomers**, and compile the stream you get back into a map, and store that as your map of customers. Do the same with **parseInvoices**, except that you'll pass your map of customers as a second parameter to that method, and compile the resulting stream directly to a list.

9. Implement getters for your customers map and your invoices list. Use **Collections.unmodifiableMap** and **Collections.unmodifiableList** so the caller won't be able to use the returned collections to change anything in the data sets directly.

10. At this point you should be able to un-comment the implementation of test #2, and this will prove that your **Billing** object correctly gets a parser for the given filename – which will be a CSV parser for country singers and a flat-format parser for movie stars – and uses the parser to load the data. Again, the test is minimal, and for example you might want to check the invoices list as well as the map of customers.

11. Implement the first of the required queries. Your method should return a **Stream<Invoice>** and the invoices in the stream should be sorted by number.

12. Create the **Reporter** class, and give it fields for a **Billing** object, a folder that will be the target location for reports, and a **LocalDate** called **asOf** that it will use when running reports on overdue invoices. Create a constructor that initializes all three fields.

13. Look in the **expected** folder to get an idea of what your first report should look like: **country_singers_invoices_by_number.txt** or **movie_stars_invoices_by_number.txt**.

14. Create the method that will produce this first report: it takes no parameters and returns nothing. It should open a **PrintWriter** on a file in the given target folder; print out a header for the report; and then call the corresponding query method on the **Billing** object and translate the invoices to lines in the report. Be sure to use try-with-resources to assure that the file is closed when you're done.

15. At this point you should be able to un-comment some of the code in test #3: down as far as the call on the **reporter** object to the first of the reporting methods. Leave the rest commented out for now. If you run this, you should see your report generated into the **reports/country_singers** or **reports/movie_stars** folder, and you can check the content against the corresponding file in the **expected** folder. As you refine the report, focus on the data and correct, readable output, and don't worry too much about how much whitespace or specific column names.

16. We'll get to the other queries and reports, but let's look now at the updating side of the system. It's time now to implement the formatting and producing methods in your parser classes. The formatting methods already know how to format text lines correctly, but they use placeholder values such as "NYI". Replace these with values from the actual customer or invoice passed to each method. Then the producing methods should be straightforward, translating in the opposite direction as compared to the parsing methods.

17. Add helper methods to **saveCustomers** and **saveInvoices** to your **Billing** class. These will in turn derive streams of customers or invoices, pass them to the corresponding producer methods on the parser, and then write the resulting stream of strings to the appropriate file – overwriting the file that was supplied to you originally.

18. Add a **payInvoice** method that takes an invoice number. Implement this to find the invoice by number, set its paid date to the current system date (call **LocalDate.now**). Then call **saveInvoices**.

19. You can un-comment the call to this method in test #3, and run the test. Inspect the invoices file in either the country-singers or movie-stars folder under **data** – not the one right in the **data** folder, as that's used as a backup and isn't the one passed to your **Billing** object! – and see that the specific invoice does indeed have today's date as its paid date.

Notice though that, as of the end of the test, the generated report is out of date, showing that invoice as still unpaid. This reflects the current limitation that the **Reporter** is not aware of changes made through the **Billing** object. You could explicitly re-run the report, of course. But better will be to have the reporter "observe" the billing object.

20. In **Billing**, add a field **invoiceListeners**, of type **List<Consumer<Invoice>>**, and initialize it to an empty **ArrayList**. Create methods **addInvoiceListener** and **removeInvoiceListener** that each take a **Consumer<Invoice>** and either add it to or remove it from the list.

21. In **payInvoice**, after saving the updated invoices, call **accept** on each object in **invoiceListeners**, passing the newly-paid invoice.

22. In **Reporter**, add a method **onInvoiceChanged** that takes an **Invoice** and returns nothing. (This of course matches the method signature of **Consumer.accept**, so we'll be able to use this method all by itself as our "listener".) Implement it to call the existing method that produces the invoices-by-number report

23. At the bottom of the constructor, call **billing.addInvoiceListener**, passing a reference to your **onInvoiceChanged** method.

24. If you test now, you should see that the report is generated and then generated again, overwriting the first report with the updated information about the recently-paid invoice.

At this point, you have a pretty good end-to-end solution, and you've implemented all of the identified design patterns and have essentially one of each thing that the eventual solution will have. Now you can expand the system …

- Implement the remaining queries and reports.
- Add a system of listeners for customer changes, separately from invoice changes.
- Make the reporter register an **onCustomerChange** method as well as **onInvoiceChange**, and make sure any affected report is generated when a given data collection is modified.
- Implement the other two updating methods, to create new customers and invoices. Make sure that these two methods notify registered listeners.

One interesting extension, if you have time, would be to sort the customers-and-volume report in descending order of volume. This is not as simple as some other ordering requirements, because the volume is currently the value in a map, rather than a key; can you devise a better way to arrange the data?