Revised Requirement from Reply - Augmented with Assumptions + Design/ Implementation Notes.

Requirement

A food manufacturing company has a requirement to rapidly calculate the number of days of forecast demand for its finished products their current stock levels of raw materials can support.

The finished products that the company produces are as follows:

- Chocolate Cake
- Iced Buns
- Victoria Sponge
- Malt Loaf

The main raw ingredients for one unit of each of the above are:

Chocolate Cake	• 175g self-raising flour • 3 tbsp cocoa powder • 175g butter • 175g sugar • 1 tsp baking powder • 3 eggs
Iced Buns	• 110 g butter • 110 g sugar • 2 eggs • 1/4 tsp vanilla extract • 140 g plain flour • 1/4 tsp baking powder • 100 g icing sugar
Victoria Sponge	• 225g butter • 225g sugar • 4 eggs • 2 tsp vanilla extract • 225g self raising flour
Malt Loaf	• 450g plain flour • 1 tablespoon yeast • 225g sultanas • 1 teaspoon sugar • 4 tablespoons malt extract • 2 tablespoons black treacle • 25 g butter

Assumption: *To ease addition of new products, they will be read from a file with the following format:*
(first line of file below)

Chocolate Cake
175,g,self-raising flour
15,g,cocoa powder
175,g,butter

175,g,sugar
5,g,baking powder
3,qty,eggs

Iced Buns
110,g,butter
110,g,sugar
2,qty,eggs
1,ml,vanilla extract
140,g,plain flour
1,g,baking powder
100,g,icing sugar

(Products separated by a single blank line. End of file indicates end of last product)

The application must take an input of stock levels for each of the raw materials plus a demand forecast for the coming period for each finished product. The output will be the number of days of demand, rounded to a single decimal place, that each of the raw materials can support.

The application should also flag any exception scenarios (e.g. missing raw material data for a finished product, or an inability to calculate a number of stock days based on the available forecast data).

Assumption: *the program aborts - printing the error message - on finding the first syntax or semantic error it encounters when parsing any input files.*

Assumption: *Input file names are provided on the command line in the following order.*

- *Ingredients stock level file*
- *Product ingredients list file*
- *Product sales forecast file.*

Note also that:

- *the ingredients stock level file "defines" the available ingredients. So the program will produce an error message if the product ingredients file contains an ingredient that is not in the stock level file.*
- *Ditto, the product ingredients file "defines" the available products. An error will occur if the product sales forecast file contains an undefined product.*
- *Products can be omitted from the forecast by simple not including them in the product forecast file.*

Example input formats are provided below.

The application should be able to support the following:

The addition of new finished products, and any new raw materials required for the new products.

(done by introducing the product ingredient list file)

The removal of a finished product from the calculations (to enable the company to see what impact, for example, stopping production of chocolate cakes would have on the stock days)
(all files can contain a comment character (#) at the beginning of a line. To remove a product from the calculation, simply comment it out (or delete it) from the product forecast file).

The ability to easily change the 'mix' of raw materials for each finished product, to see what impact that would have on stock daysThe addition of new finished products, and any new raw materials required for the new products.
(done by introducing the product ingredient list file)

Bonus points – in addition to the above, the company would also like to be able to configure a shelf life for each raw material, and flag when the shelf life will expire before the products have been used up (e.g. if eggs have a shelf life of 5 days, and the current stock level of eggs could support 10 days' worth of finished goods demand, this should be flagged). Extra credit will be given if you are able to either implement this, or describe how you would alter your solution to meet this requirement.
(didn't implement this in full - however it is a trivial extension given the program structure. To do this the ingredients stock level file would be augmented with the addition of the ingredients shelf life - in days - at the end of each line. Once the number of days each ingredient can support has been calculated - see the program - simply compare the two figures and flag if the shelf life is shorter than the number of supported days).

Please read the following instructions carefully:

- Create a Java application to meet the requirements described above. The solution should be packaged appropriately and you should provide instructions on how to run it. Please also provide the source code for the solution.

(done).

- Please include a set of unit tests, separate to the main source code, to prove that your solution works.

(*done - each package contains a file called testXXXX.java - which is a JUnit test file. The test coverage is fairly extensive and takes more lines of code than the application!)*

- You may use any open source third party component that you see fit in your solution

(JUnit 4.8.1)

- Your solution should demonstrate best practice.

(Could be a few minor improvements no doubt, but nothing major).

Do not produce a user interface for this solution; we expect the inputs to be passed in

to the application as flat files, and the output to be given on the command line or in an output file.

For stock levels, the format will be: '[raw material name], [unit of measure],[current stock level]', e.g:

self-raising flour,kg,500
plain flour,kg,200
cocoa powder,kg,10
butter,kg,800
sugar,kg,600
baking powder,kg,15
yeast,kg,10
eggs,qty,10000
vanilla extract,litres,10
icing sugar,kg,100
sultanas,kg,100
malt extract,litres,20
black treacle,litres,5

Assumption: *Unit types must match across all input files - it's an error if they don't. Units like tsp are to be convered into accurate units (grams, ml …). The program will read by kg and gs, and also litres and mls. It will with conversion of these types of units as appropriate.*

For the demand forecast, the first line will contain dates: ',[date1],[date2],…..,[datey]', the following lines will contain expected demand for each finished product for each day: '[finished product name],[date1 demand],[date2 demand],…..,[datey demand]', e.g.:

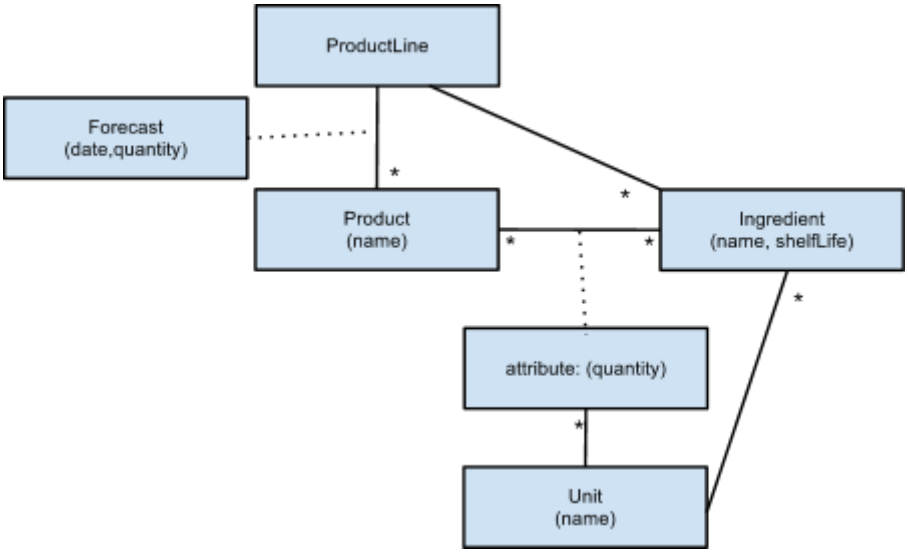
,1/2/12,2/2/12,3/2/12,4/2/12,5/2/12,6/2/12,7/2/12,8/2/12,9/2/12,10/2/12
Chocolate Cake,100,50,200,20,700,60,50,30,100,900
Iced Buns,50,10,100,10,300,200,200,10,200,200
Victoria Sponge,100,100,500,500,90,80,50,80,10,200
Malt Loaf,50,50,50,50,50,50,50,50,50,50

Assumption: *The dates do not need to be continuous. There can be "days off." The results of the program will be in working days (not elapsed days). The dates themselves may be useful in the future - so they will be stored by the program in a usable fashion.*
(Note: the dates should probably have to be in ascending order - but this check was not implemented).

Discussion of Solution

Analysis (domain model)

Initial domain model:



(assume a multiplicity of one if no value shown, "*" indicates many, as per UML)

Major relationships:

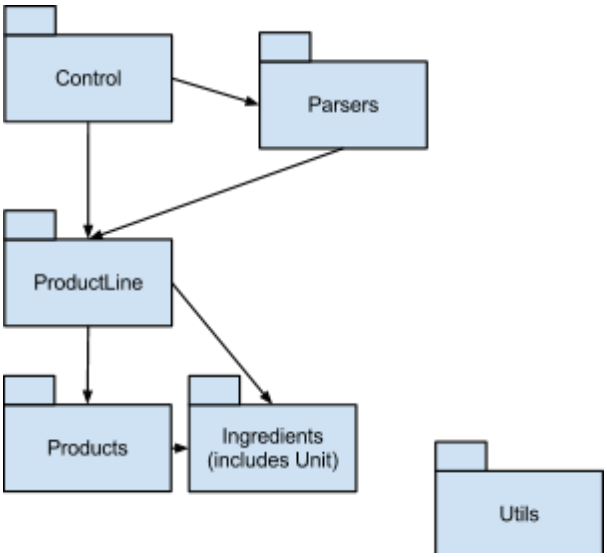
- ProductLine has many Products (chocolate cake, malt loaf ...), Ingredients (sugar, vanilla essence ...) and Units (kgs, liters ...) associated with it.
- Each Product on the ProductLine may have an associated Forecast which says how many items are predicted to be sold on a particular date.
- Product has many Ingredients, each individual Ingredient has a quantity associated with it (e.g. malt loaf has 3g of sugar in it).
- Each Ingredient has a Unit associated with it (e.g. vanilla essence is measured in liters - or litres if you prefer)

Other notes:

- Most entities (classes) have a name, the exceptions being quantity which is really just a number and an attribute of the Product/Ingredients association, and ProductLine - which is really there to bind everything together.
- Ingredients have an associated shelf life (days) - they do in the program, but it is not used or set.

Implementation (package model)

The implementation has the following package structure and dependencies.



- Not all dependencies are shown. Dependencies are transitive (so Parsers know about Products, etc...)
- Utils is misc useful stuff. It is depended on by everything.
- Dependencies <u>are</u> unidirectional when shown as such (by arrowheads).

Design

Apart from the package structure the major design points are:

- ProductLine provides a “service oriented architecture” access point to the domain elements. Control and Parsers use this. In the event of a UI being required, ProductLine would still provide the major point of access into the core of the system. Many of the required methods are already there.
- ProductLine keeps tabs on Products and Ingredients - using HashMaps. These are referenced by the Product or Ingredient name (String).
- Products keep a list of Ingredients that they use. Again, HashMaps and String names.
- Units are part of the Ingredients package. Ingredients keep tabs on their units (e.g. sugar is in grams - and this is stored within the Ingredient. Unit is an enum and also provides some unit conversion aids (e.g. converting litres to ml). This could have been extended to cope with “tbsp”, etc. with relative ease (but wasn’t).
- Parsers raise ParseError exceptions when something is wrong. Control deals with the presentation of this information to the user. None of the lower level packages (Product, Ingredient …) know anything about this (of course).
- To faciliate unit tests, the Parsers take arrays of Strings (instead of files). This also has the nice effect that comment lines (beginning with ‘#’) can be stripped out before the Parser gets to see them.

Running the program and tests.

- unpack the zip into an appropriate location ($DIR)
- install JUnit 4.8.1 or compatible.
- Set CLASSPATH to point to the JUnit jar.

- The top level directory ($DIR) contains all the packages, and three (Unix Shell) scripts:
    - runMainSimpleTestData.sh - runs the main application using some easy to figure out test data.
    - runMainSuppliedData.sh - runs the main application using the test data supplied above (note: the program picked up a few typos - , which were corrected).
    - runTests.sh - runs all the unit tests (in all except the Control package).

Take a look inside these shell scripts to vary the files used or the tests, etc. They are very simple (but are Linux based).

Other information

- test data (files) is held under the testData directory in $DIR. No surprises there.
- there are two sets of test data (file) as per the discussion of scripts (.sh files) above.
- To generate syntax errors, semantic errors (as per the above assumptions) just edit the test data files (all of which have a ".eg" extension.

What could be improved

With more time, I would have implemented the following changes:

- ParseError should be subclassed to make the different types of error more explicit. At present an error String is used to indicate exactly what the error is. This is a weakness when it comes to the JUnit tests (tests for exceptions being thrown).
- To test exceptions I have used "fail" under the line of code where exceptions should be thrown. There are other ways...
- The Parser unit tests could probably do with a little refactoring to tidy them up and remove some duplication.
- testParser.java should probably be broken into three test classes, one for each file type. At present it does all the tests for all Parsers.
- Made it compulsory for the dates in the files to be in ascending order.
- Rationalised the file formats to make them more self consistent. E.g. ingredient,unit,value across all files).
- Allowed "tsp" and "tbsp" to be units (and converted them to grams or millilitres).