SC-3

Updated: 2023-08-08, for sc3 ver. 1.010

This demo is based on the "SC-3" model provided by Rutgers' School of Business team (Ben M. et al), as of July 2023.

The source code of this application is available on GitHub, from https://github.com/eclab/DES-Supply-Chain-demo. The relevant package is edu.rutgers.sc3.

The code in this project represesnts a simple supply chain simulation implemented in Java using the MASON API and the MASON DES library

SC-3 vs. SC-2 and SC-1 (a.k.a. Pharma3)

The source code for the SC-2 app lives in the same Github repository as that for the SC-2 and SC-1 (aka Pharma3) apps developed in 2022-2023. The SC-3 app can be compiled and used in essentially the same way as Pharma3, the main difference being that the main Java class for SC-3 is `edu.rutgers.sc3.Demo` (vs. `edu.rutgers.pharma3.Demo`), and the main driver script is `run-sc3.sh`

Prepare

Software tools

You need a Java compiler, which can be obtained from Oracle or OpenJDK. For convenience, I also use a Git client and Apache Ant (a `make`-like tool for Java).

A visualization tool such as Gnuplot is handy, but is not required.

Check out the code

- Check out the source code for this project from GitHub. Let's say you've put it to ~/mason/work on your machine:

```
mkdir ~/mason
cd ~/mason
mkdir work
cd work
git init
git remote add origin https://github.com/eclab/DES-Supply-Chain-demo.git
git pull origin master
```

We'll call this location your *work directory*.

- Create also `~/mason/lib`, and put into it the MASON jar file, `mason.20.jar`, which you should download from the MASON site above.

Note: as of 2022-06-07, we are using a customized version of mason.20.jar, called mason.new4.jar, recently supplied by Sean, which has extra GUI support. To simplify compilation, I have sim-linked it to the old name:

```
cd ~/mason/lib
ln -s  mason.new4.jar  mason.20.jar
```

If you are outside of Sean's GMU team, ask Vladimir or Sean Luke to provide you with this customized version.

- Obtain the DES library source code from its own GitHub repo (Link: MASON DES library), compile it and package it into `des.jar`, and place the JAR file into the same `~/mason/lib` directory. Sample commands for doing this (assuming you place the DES library source code under `~/mason/git`):

```
#!/bin/csh

set m=~/mason

cd $m
mkdir git
cd git
git init
git remote add origin https://github.com/eclab/mason.git
git pull origin dev

cd $m/git/code/contrib/des
javac -d $m/out -cp $m/lib/mason.20.jar sim/des/*.java
# javadoc  -d $m/javadoc -cp $m/lib/mason.20.jar:$m/out -sourcepath . sim.des
cd $m/out
jar cf des.jar sim/des/*.class
cp des.jar ../lib
```

Warning: the above may take forever, as GitHub will tend to give you megabytes and gigabytes of stuff you don't need. So, alternatively, you can just ask me for precompiled des.jar, and put it into your ~/mason/lib

Compile

I compile with Apache Ant:

```
cd ~/mason/work
ant
```

This command will compile the source code and build the file `~/mason/work/lib/demo.jar`

Apache Ant is the tool that takes its commands from build.xml. If you don't want to use Ant, you can just compile the Java code from `src` using the `javac` command, and making use of the JAR files in the `~/mason/lib` directory.

Javadoc documentation

Most of the documentation for the Java classes of this application is in Javadoc format. It can be built with Apache Ant as well:

```
cd ~/mason/work
ant javadoc
```

The above command will build the documentation in `~/mason/work/web/api`

Run - text output only

You can run a single simulation with a command like this:

```
./run-cs3.sh -until 365 > tmp.log
```

The `-until 365` is a standard MASON option, which will cause the application to run for 365 units of simulation time (simulated days, in this model). Without this option, the application will likely never stop.

Command-line options

There are a few additional command-line options you can use with `run-sc3.sh`:

- `-verbose` --- causes various elements of the supply chain model to print various progress messages
- `-quiet` --- suppresses most of the printing to the standard output
- `-config config/sc2.csv` -- specifies the name of the configuration file (see below)

You can also use any of the options that all MASON applications take, such as the already-mentioned `-until`, or `-seed`. See the MASON manual (found at the same web site) for details.

Run - with charts

This is similar to how it's done in SC-2. See the description of `scripts/sc-2-run-all-disruptions.sh` in Disruptions in SC-2: How to run.

Run - in an optimization program

Suppose you want to run the SC-3 application many times, with different disruption scenarios, collecting some important statistics (your objective function(s)) each time. For a simple (trivial) example of such a program, see the class `edu.rutgers.test.TestSc3`. In this class, the `main()` method runs a loop in which a model (`Demo` object) is created several times, and each time it's run with a different disruption scenario; the objective function is then retrieved from it and printed.

This can be run with the script `scripts/sc-3-optimization-test.sh`. (Despite the name, there is no optimization going on there, of course. But your optimization code can use the Demo class the same way as shown in that trivial example.)

This script can also be run with

```
-repeat N
```

option, in order to perform N runs for each scenario and average the results. The

```
-until N
```

option can be used to specify the length (in simulated days) of each run. The

```
-disrupt file.csv
```

option can be used to run simulation with a specific "custom" disruption scenario, instead of trying all the "standard" scenarios.

For example,

```
scripts/sc-3-optimization-test.sh -repeat 100 -until 5000 -disrupt config/dis-sc3/test-7.csv
```

will run 100 baseline simulations and 100 simulations with the disrupotion scenario from the specified file. Each simulation will run for 5000 simulated days.

Run - GUI

This is not supported in SC-3 at the moment.

Configuration file

(

For more details, see the page SC-3: configuration file parameters)

The simulation tool takes the parameters of each element of the model from the configuration file, which, by default, is `config/cs3.csv`.

The configuration file is in CSV format. All lines beginning with a hash mark (#) are ignored, as well as all empty lines.

The data lines have the following format:

```
elementName,propertyName,val1[,val2,...]
```

The first column contains the name of the element of the network to which the property applies. The second column, the name of the specific property. The third column (and any subsequent) columns contain the value of the property.

When the config file is read in, an instance of the class `edu.rutgers.util.ParaSet` is used to represent all parameters pertaining to a particular `elementName`.

Normally, there is just one value column in each row of the config file. However, some types of properties need several columns to be described. For example, the following line

```
Production,faulty,Uniform,0.1,0.2
```

means the that the parameter `faulty` for the element called `Production` describes a uniform distribution with the range [0.1,0.2]. (This is the distribution from which the model of the post-production QA step draws the number that determines the portion of faulty items in each produced lot. This particular distribution means that 10% to 20% of items in each lot are found faulty.)

Sample output

The output file, for a run without the `-verbose` option, may look like this:

```
%./run-sc3.sh -until 2000

Demo.start
Disruptions=null
SC3 DES/MASON simulation, ver=1.007, config=config/sc3.csv
===== 2000.4874778414937: Finish =================
   --- SUBSTRATES ---
[Production.prepregProd; stored inputs=([
        batch.fiber:1781.  Ordered 5711, received 5711. On order=0; in transit 0.0 ba,
        batch.resin:102. (Discarded expired=382.8919999999998 u = 7.0 ba).  Ordered 3566, received 3370. On order=195; in transit 1.0 ba])
Ever planned: 120132; still to do 2238.0. Ever started: 117894 (112 ba) = (in prod=0+(1) ba){QA: (in QA= 3 ba; discarded=5752 (103 ba); good=1
Queued(+prod): 0+(1); [prepregProd.prodDelay: accepted 112 ba, totaling 117894; released 116094.0 ; utilization=97.178%]
. Released 108523.0]
[Production.substrateLargeProd; stored inputs=([
        batch.prepreg:2091. (Discarded expired=3454.0 u = 3.0 ba).  Ordered 114380, received 108523. On order=5857,
        batch.aluminum:351.  Ordered 18648, received 17820. On order=828; in transit 3.0 ba])
Ever planned: 212; still to do 14.0. Ever started: 198 (99 ba) = (in prod=[0]0+(1) : [1]0+(1) : [2]1+(1) : [3]0+(0) : [4]0+(1) : [5]0+(1) ba){
[substrateLargeProd.pipeline:
        Stage[1] Queued(+prod): 0+(1); [substrateLargeProd.prodDelay.1: accepted 99 ba, totaling 198; released 196.0 ; utilization=75.064%]
        Stage[2] Queued(+prod): 0+(1); [substrateLargeProd.prodDelay.2: accepted 98 ba, totaling 196; released 194.0 ; utilization=72.04%]
        Stage[3] Queued(+prod): 1+(1); [substrateLargeProd.prodDelay.3: accepted 96 ba, totaling 192; released 190.0 ; utilization=73.008%]
        Stage[4] Queued(+prod): 0+(0); [substrateLargeProd.prodDelay.4: accepted 95 ba, totaling 190; released 190.0 ; utilization=45.019%]
        Stage[5] Queued(+prod): 0+(1); [substrateLargeProd.prodDelay.5: accepted 95 ba, totaling 190; released 188.0 ; utilization=44.239%]
        Stage[6] Queued(+prod): 0+(1); [substrateLargeProd.prodDelay.6: accepted 94 ba, totaling 188; released 186.0 ; utilization=44.183%]]
. Released 178.0]
[Production.substrateSmallProd; stored inputs=([
        batch.prepreg:2091. (Discarded expired=3454.0 u = 3.0 ba).  Ordered 114380, received 108523. On order=5857,
        batch.aluminum:351.  Ordered 18648, received 17820. On order=828; in transit 3.0 ba])
Ever planned: 376; still to do 19.0. Ever started: 357 (181 ba) = (in prod=[0]0+(1) : [1]0+(1) : [2]6+(1) : [3]3+(1) : [4]0+(1) ba){QA: (in QA
[substrateSmallProd.pipeline:
        Stage[1] Queued(+prod): 0+(1); [substrateSmallProd.prodDelay.1: accepted 181 ba, totaling 357; released 355.0 ; utilization=77.245%]
        Stage[2] Queued(+prod): 0+(1); [substrateSmallProd.prodDelay.2: accepted 180 ba, totaling 355; released 353.0 ; utilization=77.212%]
        Stage[3] Queued(+prod): 6+(1); [substrateSmallProd.prodDelay.3: accepted 173 ba, totaling 341; released 339.0 ; utilization=81.725%]
        Stage[4] Queued(+prod): 3+(1); [substrateSmallProd.prodDelay.4: accepted 169 ba, totaling 333; released 331.0 ; utilization=81.238%]
        Stage[5] Queued(+prod): 0+(1); [substrateSmallProd.prodDelay.5: accepted 168 ba, totaling 331; released 329.0 ; utilization=79.777%]]
. Released 309.0]
   --- CELL ---
[Production.cellProd; stored inputs=([
        batch.cellRM:34078.  Ordered 544728, received 508806. On order=35922; in transit 1.0 ba])
Ever planned: 582542; still to do 37814.0. Ever started: 544728 (215 ba) = (in prod=[0]0+(1) : [1]2 ba)
[cellProd.pipeline:
        Stage[1] Queued(+prod): 0+(1); [cellProd.prodDelay.1: accepted 215 ba, totaling 544728; released 541952.0 ; utilization=91.447%]
        Stage[2] [cellProd.prodDelay.2: accepted 214 ba, totaling 541952; released 537939.0 ; utilization=150.20%]]
. Released 537939.0]
[Production.cellAssembly; stored inputs=([
        batch.cell:48339. Received 537939.0,
        batch.coverglass:19418.  Ordered 582542, received 509018. On order=73524; in transit 26.0 ba])
No planning (driven by input). Ever started: 489600 (153 ba) = (in prod=0+(1) ba){QA: (in QA= 1 ba; discarded=24274 (151 ba); good=458926}
Queued(+prod): 0+(1); [cellAssembly.prodDelay: accepted 153 ba, totaling 489600; released 486400.0 ; utilization=79.268%]
. Released 458926.0]
[Production.cellPackaging; stored inputs=([
        batch.cell:886. Received 458926.0,
        batch.cellPM:13344.  Ordered 465816, received 449160. On order=16656; in transit 1.0 ba])
No planning (driven by input). Ever started: 458040 (165 ba) = (in prod=0+(0) ba)
Queued(+prod): 0+(0); [cellPackaging.prodDelay: accepted 165 ba, totaling 458040; released 458040.0 ; utilization=12.196%]
. Released 458040.0]
   --- ARRAY ASSEMBLY ---
[Production.arrayLargeAssembly; stored inputs=([
        batch.substrateLarge:7.  Ordered 214, received 178. On order=36,
        batch.cell:1876.  Ordered 559888, received 452488. On order=107400,
        batch.adhesive:55.  Ordered 559, received 507. On order=52; in transit 1.0 ba,
        batch.diode:738.  Ordered 8036, received 4006. On order=4030; in transit 1.0 ba])
Ever planned: 214; still to do 43.0. Ever started: 171 (171 ba) = (in prod=[0]0+(0) : [1]0+(1) : [2]0+(1) : [3]0+(1) : [4]0+(1) : [5]0+(1) : [
[arrayLargeAssembly.pipeline:
        Stage[1] Queued(+prod): 0+(0); [arrayLargeAssembly.prodDelay.1: accepted 171 ba, totaling 171; released 171.0 ; utilization=68.10%]
        Stage[2] Queued(+prod): 0+(1); [arrayLargeAssembly.prodDelay.2: accepted 171 ba, totaling 171; released 170.0 ; utilization=68.192%]
        Stage[3] Queued(+prod): 0+(1); [arrayLargeAssembly.prodDelay.3: accepted 170 ba, totaling 170; released 169.0 ; utilization=67.727%]
        Stage[4] Queued(+prod): 0+(1); [arrayLargeAssembly.prodDelay.4: accepted 169 ba, totaling 169; released 168.0 ; utilization=67.737%]
        Stage[5] Queued(+prod): 0+(1); [arrayLargeAssembly.prodDelay.5: accepted 168 ba, totaling 168; released 167.0 ; utilization=67.017%]
        Stage[6] Queued(+prod): 0+(1); [arrayLargeAssembly.prodDelay.6: accepted 167 ba, totaling 167; released 166.0 ; utilization=66.328%]
        Stage[7] Queued(+prod): 0+(1); [arrayLargeAssembly.prodDelay.7: accepted 166 ba, totaling 166; released 165.0 ; utilization=61.794%]]
. Released 161.0]
```

```
[Production.arraySmallAssembly; stored inputs=([
        batch.substrateSmall:0.  Ordered 368, received 306. On order=62,
        batch.cell:1876.  Ordered 559888, received 452488. On order=107400,
        batch.adhesive:55.  Ordered 559, received 507. On order=52; in transit 1.0 ba,
        batch.diode:738.  Ordered 8036, received 4006. On order=4030; in transit 1.0 ba])
Ever planned: 368; still to do 62.0. Ever started: 306 (306 ba) = (in prod=[0]0+(0) : [1]0+(0) : [2]0+(0) : [3]0+(1) : [4]0+(0) : [5]0+(0) : [
[arraySmallAssembly.pipeline:
        Stage[1] Queued(+prod): 0+(0); [arraySmallAssembly.prodDelay.1: accepted 306 ba, totaling 306; released 306.0 ; utilization=26.863%]
        Stage[2] Queued(+prod): 0+(0); [arraySmallAssembly.prodDelay.2: accepted 306 ba, totaling 306; released 306.0 ; utilization=26.63%]
        Stage[3] Queued(+prod): 0+(0); [arraySmallAssembly.prodDelay.3: accepted 306 ba, totaling 306; released 306.0 ; utilization=26.893%]
        Stage[4] Queued(+prod): 0+(1); [arraySmallAssembly.prodDelay.4: accepted 306 ba, totaling 306; released 305.0 ; utilization=26.592%]
        Stage[5] Queued(+prod): 0+(0); [arraySmallAssembly.prodDelay.5: accepted 305 ba, totaling 305; released 305.0 ; utilization=27.031%]
        Stage[6] Queued(+prod): 0+(0); [arraySmallAssembly.prodDelay.6: accepted 305 ba, totaling 305; released 305.0 ; utilization=26.43%]
        Stage[7] Queued(+prod): 0+(1); [arraySmallAssembly.prodDelay.7: accepted 305 ba, totaling 305; released 304.0 ; utilization=30.608%]]
. Released 302.0]
   --- END CUSTOMER ---
arrayLargeCustomer: Ordered=214.0, received=160.0. Avg waiting time for 17 filled orders 538.1644632068416 days, for 6 unfilled orders 243.154
arraySmallCustomer: Ordered=368.0, received=298.0. Avg waiting time for 56 filled orders 275.5492659824413 days, for 14 unfilled orders 266.97
All customers: Avg waiting time for 73 filled orders 365.8232400283288 days, for 20 unfilled orders 256.3083733638818 days so far, for all 93
================================================
1.747u 0.204s 0:01.22 159.0%    0+0k 0+0io 2687pf+0w
```

(Later, I will add output from more nodes of the model).

Time series files and visualization

When you run the application with the `-charts` *`dir_name`* option, the application writes a number of time series files into the charts directory for the run. Each file pertains to a particular node of the supply chain (e.g. eeCmoProd.csv describes the behavior of the EE CMO Production node); it has one row per day, with a number of columns containing the value of various relevant variables.

The time series can be visualized with a tool such as `gnuplot`. For a sample gnuplot command file, see `gnu/sc3/sc3.gnu`.

~~Explanation of the model~~

~~See design sketch~~

~~Names of resources~~

~~These names appear in reports and configuration files.~~

- ~~EE -- an EE device~~
- ~~DS -- a DS device~~
- ~~RMEE -- raw material for EE~~
- ~~PMEE -- a unit of packaging material for EE~~
- ~~RMDS -- raw material for DS~~
- ~~PMDS -- a unit of packaging material for DS~~

~~Names of supply-chain elements~~

~~These names appear in reports and configuration files, as well as in the names of time-series CSV files.~~

- ~~ServicedPatientPool~~
- ~~WaitingPatientQueue~~
- ~~eeCmoProd -- EE CMO Production unit~~
- ~~eeCmoProd.safety.RMEE -- the EE Raw Material safety stock parameters for eeCmoProd~~
- ~~eeDC -- EE Distribution Center~~
- ~~eeDP -- EE Distibution Pool~~
- ~~eeHEP -- EE Hospital Equipment Pool~~
- ~~eeRMSupplier -- EE Raw Material Supplier Pool~~
- ~~eePMSupplier -- EE Packaging Material Supplier Pool~~
- ~~eePackaging -- EE Packaging facility~~
- ~~eePackaging.safety.PMEE -- the EE Packaging Material safety stock parameters for eePackaging~~
- ~~eeMedTech -- the "headquarters" of the MedTech company (variables related to the EE supply)~~
- ~~the names for the DS branch nodes are similar, with "ds" instead of "ee".~~

~~Resources~~

~~In the previous model (pharma2) all resources were viewed as fungible (a tablet of aspirin being as good as any other tablet), and were modeled by DES `CountableResource` class. In this model (pharma3) we take into account the fact that in reality some products come in identifiable lots; this makes it possible to associate an expiration date with each lot. Additionally, since each lot has an identifiable provenance, one can model use cases which, for example, involve discovering *post factum* that a particular factory had an outbreak of a bacterial contamination, not detected by the QA at the time, and now a certain series of lots need to be discarded.~~

~~Accordingly, we created class `Batch`, based on DES's `Entity`.For any product that comes in identifiable lots (e.g., "bulk drug"), a *pro forma* (prototype) Batch instance is created with~~

```
   Batch(CountableResource typicalUnderlying, double _shelfLife);
```

~~or~~

```
   Batch.mkPrototype(CountableResource typicalUnderlying, ParaSet para);
```

~~After that, Batch instances representing actual lots of that product can be created based on the *pro forma* Batch with~~

~~`Batch.mkNewLot(double size, double now);`~~

~~Each Batch object contains a `lotNo` variable, which can be used to look up any relevant lot details using the class `Lot`.~~

~~In our application pharma3, the raw material, the excipient, the API, the bulk drug, and the packaged drug are all modeled with the Batch class, each one with an appropriate CountableResource as the underlying resource class. The packaging material is modeled with CountableResource, because the empty glass bottles or blister packs are viewed as fully fungible, and don't have expiration dates of their own.~~

~~Different handling of Batch and CountableResource resources~~

~~In many of our classes modeling various supply chain steps, the `Batch` resources are handled differently from the "fungible" (`CountableResource`) ones.~~

~~In paricular, the QA process on the Batch results in the discaring (or "reworking") of an entire batch with a certain probability; in contrast, when a certain amount (a batch, lower-case) of a fungible resource goes through QA, we pick a "failed percentage number" from a certain distribution, and discard a certain percentage of the batch. This is meant to represent different modes of testing of different products: e.g., if you taste the contents of a barrel of beer or milk, you dump the entire barrel if you think the beer or milk is bad, while for something like glass bottles you can test each bottle individually with an optical system, and discard just the cracked bottles.~~

~~Obviously, in real life, testing modes may not precisely map to our Batch/fungible resources; so, if necessary, the choice of the testing mode may eventually be "divorced" from the resource Batch/fungibility property.~~

~~Units~~

~~In pharma3, all order sizes and product amounts in the model are in single units, so we indeed have values running into billions.~~

~~All time values are in days.~~

~~Main application class (Demo.java)~~

~~The main class. Starts and schedules the HospitalPool and PharamCompany instances.~~

~~Splitters~~

~~A `Splitter` extends the DES `If` class. It serves to take input from one supplier, and send it to multiple receivers. A splitter is configured by specifying (with Splitter.addReceiver(...)) what percentage of the input goes to which output.~~

~~When a batch of resource comes to a splitter, the splitter decides where to send it (attempting to approximate the target ratios as close as possible), and immediately sends to the batch to the chosen receiver.~~

~~The Pharmacy/Hospital Pool (HospitalPool.java)~~

~~Configuration parameters:~~

```
HospitalPool,intervalBetweenOrders,30
HospitalPool,order,Triangular,1209021340,1209030000,1209038660
```

~~The HospitalPool object, as a subclass of `Queue` represents the amount of product that has been supplied to it by the industry (specifically, the Distributor object).~~

~~At this point, this amount can be only added to (by push actions from Distributor); it is never reduced, as we do not model consumption (which itself could be driven by epidemic outbreaks etc, for example).~~

~~Every now and then, at a fixed interval (intervalBetweenOrders=30) the HospitalPool sends an order of a size drawn from a specified distribution to PharmaCompany.~~

~~The Pharma Company (PharmaCompany.java)~~

~~This is the main class for the modeling of the focal company and the CMO company.~~

~~Config:~~

```
PharmaCompany,expiration,3650
```

~~The PharamaCompany class is Sink, which represents the functionality of receiving orders from the outside world.~~

~~The `expiration` parameter specifies the shelf life, in days, of the product it produces, which is the packaged drug.~~

~~Internally, it contains the following elements:~~

- ~~MaterialSupplier rawMatSupplier, pacMatFacility, excipientFacility; -- each of these represents an external supplier of an input material, and a quality testing (QA) step.~~
- ~~Production apiProduction, drugProduction, packaging; -- each of these elements represents an in-house production stage, plus the subsequent QA step.~~
- ~~Production cmoApiProduction, cmoDrugProduction, cmoPackaging; -- each of these elements represents a production stage, plus the subsequent QA step.~~
- ~~Splitter rawMatSplitter, apiSplitter, drugSplitter, cmoApiSplitter; -- each of this represents the splitting of output after some step, as per the PowerPoint diagram.~~
- ~~Distributor distro; -- represents the Distribiution element.~~

~~The PharmaCompany class schedules the Production and Distribution objects, but not any other objects.~~

~~PharmaCompany does not actually have a `step()` method. Instead, its actions are triggered by the accept() method, which is activated when an order from HospitalPool is received. At this point, the PharmaCompany object carries out the following actions:~~

- ~~Places an appropriately sized order for the raw material with the MaterialSupplier objects of each kind. At present, the raw material order size is directly based on the size of the drug order. (We have 1:1 ratio of units, i.e. one unit of raw material is needed to make one unit of drug, absent any losses).~~

- Sends an instant message to Distributor (the Distribution center) appropriately adjusting the total amount of finished product that we have been asked to ship out to HospitalPool.

MaterialSupplier.java (rawMatSupplier, pacMatFacility, excipientFacility)

We have 3 instances of this java class in the PharamCompany object. They represent, respectively, the Raw Material Supplier Pool, the Excipients Facility, and the Packaging Material Facility, each with its own QA testing step.

Config (for one of them):

```
RawMaterialSupplier,batch,5e5
RawMaterialSupplier,prodDelay,Uniform,0.0031,0.0217
RawMaterialSupplier,transDelay,Uniform,0.0031,0.0217
RawMaterialSupplier,qaDelay,Uniform,0.003,0.02
RawMaterialSupplier,faulty,0.05
RawMaterialSupplier,expiration,3650
```

A MaterialSupply includes, internally, a production, transportation, and QA stages, each implemented as a Delay of some kind. Each of them simulates a facility that is capable of processing only one batch of resource at a time. The throughput is controlled by setting the capacity of each delay to 1, and creating a Queue in front of each delay (e.g. `Queue needProd` before `prodDelay`.

For each of the delay objects a parameter exists in the config file representing the distribution from which the delay time is drawn.

The PharmaCompany object, soon after creating each MaterialSupplier element, links its last stage (the QaDelay) to a Receiver, which is one of the input buffers of one of the Production elements (discussed below).

At the QA stage, different process is used for fungibles (CountableResource, such as the packing material) and for identifiable lots (Batch instances). For the former, a random number is drawn from the faulty-portion distribution, and is used as the fraction of the product lot (which usually corresponds to the monthly order) that needs to be discarded as faulty. The rest of the batch is released to the Receiver (a Production step) linked to this supplier element. For the latter, the entire lot is accepted or discarded.

Production.java (API Production + API Testing; Drug Production + Drug Testing; Packaging + Packaging Testing)

The PharmaCompany object has 6 instance of the Production class: 3 for the focal company, and 3 for the CMO. Each one represents one production stage plus the subsequent inspection stage.

Config (for one of the Production objects):

```
ApiProduction,inBatch,5e5
ApiProduction,batch,5e5
ApiProduction,batchesPerDay,69
#-- faulty + rework
ApiProduction,faulty,0.01
ApiProduction,rework,0.01
ApiProduction,prodDelay,Uniform,0.1,0.2
ApiProduction,qaDelay,Uniform,0.1,0.2
ApiProduction,expiration,2555
```

A Production element itself is a Steppable (so that it can be scheduled), and internally contains an array of Queue objects, representing the input buffers for the input ingredients:

```
    InputStore[] inputStore;
```

and two delay objects, representing the production step and the QA step:

```
    ProdDelay prodDelay;
    QaDelay qaDelay;
```

During the initialization, the input of the input buffers is linked to the output of the QA stages of the appropriate suppliers (sometimes, through a Splitter), so that they can push the materials to these buffers as the pass the QA.

Conceptually, each production element carries out its production in batches of a fixed size (e.g. `ApiProduction,batch,5e5`). For `Batch` inputs, and for outputs (all of which are of a `Batch` type), a batch corresponds to a `Batch` instance; for "fungible" inputs (the only such input is packing material), a batch is simple a certain amount of CountableResource.

Although the PowerPoint document calls for what we can call a "throttled delay" implementation (same as what's done in MaterialSupplier, i.e. a production line that works on 1 batch of product at a time), our Production class at this point does not implement this approach. Instead, a simpler technique is used to limit capacity: that of starting no more than a certain number of batches per day. (E.g. `ApiProduction,batchesPerDay,69`).

To manufacture a batch of product, a certain amount of each involved input is needed. E.g., for drug production we have `DrugProduction,inBatch,5e5,5e5`, meaning that to make a batch (500,000 units) of the drug, one needs 500,000 units of the API and 500,000 units of the excipient. (The application knows which number stands for which input because of the parameters to the constructor of the relevant production object). Accordingly, at each time step (each simulated day), the Production object checks if the input buffers contains a sufficient amounts of all necessary ingredients to start 1 or several batches, and if such ingredients are available, starts the max possible number of batches (limited by `batchesPerDay`, of course). Programmatically, this is done by putting the appropriate amount of the resource into the ProdDelay.

The output of the ProdDelay is linked to the input of the QaDelay. The QaDelay operates on the "entire batch" principle, i.e. each produced batch is tested and, with certain probability, is discarded, sent back to the production stage for reword (i.e. re-queued in the ProdDelay), or released to the next stage.

What the next stage for each Production element is, is configured during the construction of the production objects by the PharamCompanyt object. They are linked as per our PowerPoint slide, with the last one (Packaging) linked to the Distributor unit.

Distributor.java (Distribution)

Config.

```
Distributor,batch,1000
Distributor,interval,30
Distributor,shipOutDelay,Uniform,1,2
```

The operation of the Distributor object is slightly different from what's written on the PowerPoint slide, simply because I couldn't figure out how that was supposed to work.

The Distributor object is Queue, into which the last stage of the production process (i.e. the Packaging Test) pushes the finished product. The Distributor then decides when ship some of that product out, based on the following rules.

Our Distributor object works as follows. It continuously keeps track of how much product needs to be eventually sent to the HospitalPool (as instructed by the PharmaCompany object upon receipt of orders), and how much it has already sent. It can do sending only on specified days (say, once a month), and only in batches of fixed size. (This can simulate e.g. a company that uses a regularly sailing container ship or container train to send products from Tel Aviv to Elizabeth, NJ, or from Wuhan to Disuburg), and which prefers to send full container loads whenever possible. Accordingly, every now and then (with the interval set by Distributor,interval,30), it checks the "need to ship eventually" amount vs. "ever shipped", to determine the "still to be shipped" amount, i.e. to see if the company still has not fulfillied the shipment plan. If the "still to be shipped" amount is non-zero (i.e the plan has not been fully fulfilled yet), the Distributor checks if the amount of product on hand is enough to fill 1 or more full batches, and if so, ships the max possible number of full batches. Additionally, if the "still to be shipped" includes an "odd lot" (a partial batch), and the Distributor has enough product to supply that odd lot, it does so as well. (In other words, if the company has been ordered 3.5 container-loads of aspirin, then we have to send a half-full container at some point; but we'll only do it after the 3 full containers have been sent).

The "sending" is accomplished programmatically by putting resource into a Delay object (with the parameters Distributor,shipOutDelay,Uniform,1,2); the output of the Delay is connected to the input of the HospitalPool (which is a Queue), so that once the product comes out of the Delay, it increments the amount of product in the HospitalPool.

Handling product expiration

While the PDF file describing the Rutgers model mentions expiration dates of some commonities, it provides no further details on how product expiration is to be handled. This section provides a brief explanation on how this is done in our application.

All products that can expire are handled as `Batch` objects (encapsulating an underlying `CountableResource`). Each Batch is linked (via a unique lot number) to a `Lot` object which stores the manufacturing date and expiration date.

Creation of batches

Each prototype Batch object (on which all Batch objects representing a given commodity type are modeled) contains 2 fields describing the expiration-related properties of this commodity type:

```
  boolean inheritsExpiration;
  double shelfLife;
```

If `inheritsExpiration` is false (which is the usual situation), then the expiration date of each newly created lot will be computed as `manufacturingDate + shelfLife`. The expiration dates of the materials used in the production of the lot don't matter; for example, if you bake a cake using milk or eggs with the expiration date of tomorrow, but this type of cake is supposed to have the shelf life of 10 days, then the expiration date of the newly baked cake will be set to 10 days from the date of baking. In the Pharma3 model this applies, fore example, for the manufacturing of API and the manufacturing of bulk drug. If `inheritsExpiration` is true, then the expiration date of each newly created lot will be set to the earliest of the expiration date of the lots of ingredients that are used to produce the new lot. For example, if a factory makes bagged trail mix using nuts, raisins, rice crackers, and plastic bags (for packaging), then the expiration date of each lot of trail mix will be the earliest of the expiration date of the lot of nuts, the lot of raisins, and the lot of crackers (assuming that the packaging material does not have an expiration date of its own). In the Pharma3 model, the product like this is packaged drug, since each lot of packaged drug simply inherits the expiration date of the bulk drug that was packaged.

Discarding expired batches

In the Distribution center, when lots are taken from storage to be sent to the Hospital/Pharmacy Pool, the expirarion date of each Batch is checked. The policy is not to send out products with less than 1 year (365 days) left until its expiration dates; so every batch that's within 1 year from the expiration date is removed from the Queue and discarded.

In the Production class (modeling various production units), a similar process takes place in each input buffer queue (Production.InputStore) object that represents storage of an input ingredient that has an expiration date. However, here only the lots expired as of the current date are discarded; there is no requirement that the lot had any extra time of shelf life left in it.

Disruptions

The disruptions mechanism in the SC-2 app is build upon that of Pharma3 (aka SC-1), with some enhancments.

See:

- Disruptions in Pharma3 (aka SC-1)
- Disruptions in SC-2