Winter 2018 OOP345 Project Milestone 5

Introduction

Everyone should have completed a CSV reader (Milestone 1) and used it to read the website data files. The csv reader role was to create a vector<vector<string>> 2-D string table that faithfully represented the information in the data files. It was not to interprete anything with respect to the data such as eliminating blank lines.

Milestone 2, based on milestone 1, used the csv reader to read task data. Once the csv data was read, the vector<vector<string>> 2-D string table was parsed, printed, and visualized by generating a graphviz graph.

Milsetone 3 read and processed item and order data files. The approach was based on what was done for milestone 2.

Milestone 4 had two phases.

The first phase read all three data file types in the same program. This required splitting (code factoring) the item.cpp, order.cpp, and task.cpp files into i.h/i.cpp, o.h/o.cpp, and t.h/t.cpp files. It was trivial to then write a iot.cpp file which read all three data types.

The second phase was integrity checking. Specifically, it verified:

1. All passed/failed tasks referenced in the task data exit.
2. All installer/remover tasks referenced in the item data exist in the task data.
3. All items ordered by a customer exist in the item data.
4. There are no task or items with duplicate names. <-- optional

We need to build a factory that can assemble each product as ordered.

Milestone 5.

Milestone 5 is a minimal viable factory simulation.

There is great introduction to simulation at http://www.cs.fsu.edu/~baker/realtime/restricted/notes/simulation.html. *In case the link goes down, here it is:*

Simulations, in General

A simulation is an imitation of a real system, that faithfully reproduces enough of the characteristics of the real system that one can pretend it is the real system for certain purposes. Probably the most familiar examples of simulations are video games, like flight simulators. A person who is flying a flight simulator is not really flying an airplane, but if the simulator is good the experience is close enough to flying a real airplane that skills developed on the simulator transfer to flying a real airplane. The obvious advantage of using a simulation is a safer and cheaper way to train a pilot than using a real airplane.

In general, a simulation program attempts to mimic in abstract form, using computations, some real-world system. This is sometimes also called computer modelling.

Simulations are used in computer science to test out ideas for software systems before full scale implementation. For example, before implementing an operating system with virtual memory one might write a simulation to evaluate the comparative performance of different page replacement policies. The objective of the simulation would be to determine which of the given policies results in better system performance. Doing such a simulation would generally be less costly than producing different versions of the actual operating system. It may also be easier in the simulation than in a working operating sytems to extract the performance measures of interest, or to set up test scenarios that explore the particular questions of interest.

Discrete Event Simulations

A discrete-event simulation starts with an abstract model of the real system to be simulated, expressed in terms of transitions of the system state that occur at discrete points in time. Each state such transition corresponds to an event. Events are classified as instances of a finite number of types, each of which has its own associated action on the system state. Each event instance is associated with a particular time of occurrence. Times are represented by integer counts of some fundamental time unit. The granularity of the time unit is chosen to fit the type of system being simulated and the level of detail one hopes to extract about the system's behaviour.

The algorithmic structure of a discrete-event simulation is as follows:

```
loop
   update clock to time of next event;
   execute the action(s) associated with this event,
   to update the system state;
end loop;
```

Tick-by-tick Event Architecture

As we get into more detail in the design of the simulation, we have to make a choice. One

approach is to simulate every tick of the clock explicitly, as follows:

```
loop
   now := now + 1;
   execute the action(s) associated with this clock tick;
end loop;
```

In an object-oriented simulation, we might have several different software objects, that correspond to different conceptual components of the real system being simulated. For example, in a coarse-grained simulation of an computer system there might be a system clock object, a simulated CPU object, and several simulated disk objects. The simulator loop might look like the following:

```
loop
   clock.increment;
   cpu.update;
   for each disk in the set of disks
      disk.update;
   end for;
end loop;
```

This tick-by-tick architecture makes a lot of sense if we are doing a detailed simulating a digital computer, where we expect some kind of change in state on every tick of the hardware clock. On the other hand, it does not work very well if we are simulating CPU or disk scheduling in an operating system, where there may be very many clock ticks between significant events.

Event Queue Architecture

To simulate systems where the timings of events are sparse and irregular, we can introduce a queue of events, ordered by time of occurrence. The simulator loop then looks like the following:

```
loop
   remove next event from queue,
   make it the current event;
   now = time of current event;
   execute the action(s) associated with the current event;
end loop;
```

Graph Source, Sink, and Singleton Nodes

A graph **source node** is a node which does not have any incoming edges. If there is only one source node, any other node can be reached by starting at the source. This may or may not be true if there are multiple sources.

A graph **sink node** is a node which does not have any outgoing edges. The graph terminates in sink

nodes. Once something flows to a sink node, there is no way out.

A **singleton node** is a node which does not have any edges entering or leaving the node. It is an isolated node. There is no way to enter the node from an edge and there is no way to leave the node on an edge. There is a singleton node in the fish task data.

We can classify our task nodes as source, sink, or singleton nodes.

If the task does not reference a pass task, it is a sink. (It cannot have a fail task unless if also has a pass task.) You can create a **bool Task::isSink() { return taskPass.empty();** } member function.

Source nodes can be identified by counting the number of edges connecting to that task. If the count is zero, the task is a source.

If a task is both a source and a sink (no incoming, no outgoing edges) the task is a singleton.

So how do we count incoming nodes?

Something like this:

Add an incoming count data variable to class Task. Initialize all counts to zero.

```
For each task {
   if this task has pass or fail tasks {
      (the pass/fail tasks have an incoming edge from this task)
      increment the incoming count for the pass/fail tasks
   }
}
```

If the count for a task is zero, the task is a source. A singleton node is both a source and a sink node.

Milestone 5 Architecture

For milestone 5, we are going to build a minimal viable tick-by-tick factory simulation.

A job assembled in a factory where the job moves from machine to machine, or work station to workstation, is an example of network job flow.

A job is created from an customer order. An order consists of a list of items to be installed.

A job advances through the network.

The task data is a graph of the job flow network. The nodes are the set of tasks, machines, robots, people, or work stations which perform a task as part of the assembly of the product. The edges are node pairs which define how a job flows from one task to another. Each edge is directed. It has a source and a destination. The job flows along an edge from the source node to the designation node. Each node (source) possibly performs some work on the job and forwards it along the graph to the next

(destination) node.

The job flows through the factory from node to node, starting at the graph source node.

Recall a source node is a node that is never the destination node for an edge. All edges for a source node exit the node.

A job exits the system when it reaches a sink node. A sink node does not have eny edges leaving the node. It cannot leave. It is terminal. Game over.

Let's simulate the factory using the *Tick-by-tick Event Architecture*

```
loop
  now = now + 1;

  execute the action(s) associated with this clock tick;
end loop;
```

Coding like most things in life is an exercise in organization. The secret is to understand what you are trying to do. If you do not know what you are trying to do, you cannot do it.

Look at the flow chart. Simple, isn't it?

We recommend and use the Agile development method:

Write simple code starting at the top of the flow chart, moving toward the bottom. Keep it simple (KISS). Take baby steps. Do not over-think what you are attempting to do. Functions should only be added or enhanced when required by the step you are working on. Do not over-think what you are doing by writing code required for future steps. Yes all sorts of things need to be written for the C++ classes. They will be written as required. This saves a lot of time by keeping everything simple as possible at each step. Simple code is easy to test. Simple code is easy to write.

Test each function as it is written. Print out the function result. If it in a loop, call 'exit(int);' to stop the loop. Verify it is producing what you expect before you move on. (Remove the exit call.)

Complexity is the enemy of organization. Complex things rarely work. They are expensive to develop. They are hard to test. They have bugs. They take a long time to code, and test.

Agile development works. It is commonly used in industry.

The latest Agile development rage is pair programming. This is where two programmers sit down at the same computer and write the code together. Each programmer looks over the shoulder of the other, catching logic errors, and improving the quality of code, simplifying, pondering better ideas for the programming task at hand. Yes at first it appears to double the cost of writing code. Not quite true. It reduces the total cost by producing code with far fewer errors. The cost saving results from not have to fix as many bugs.

Factory Pseudo Code

1. Create a factory consisting of a network of machines, one machine per task.
A machine has an input job queue which is a queue of jobs waiting to be processed by the machine.
The item data file entries state whether a machine is an installer or remover.

2. Find the 'source' node for the machine network.

3. Each order is a job. A job has a data element that flags whether an item has been installed.
Load all the orders into the input queue for the 'source' node.

4. While there are incomplete jobs:

5. Loop: over all machines

6. If the input queue for the machine is not empty

7. Remove the job from the input queue

8. Is this a sink node machine?

9. Mark job completed, ship it
goto 19

10. If the machine is a installer and the job needs this item

11. Install item

12. If this machine is a remover and the job has the item installed

13. Remove item
goto 19

14. Inspector

15. If the machine has only a 'passed'

16. Move the job to the passed machine input queue

17. The machine has both a 'passed' and 'failed'

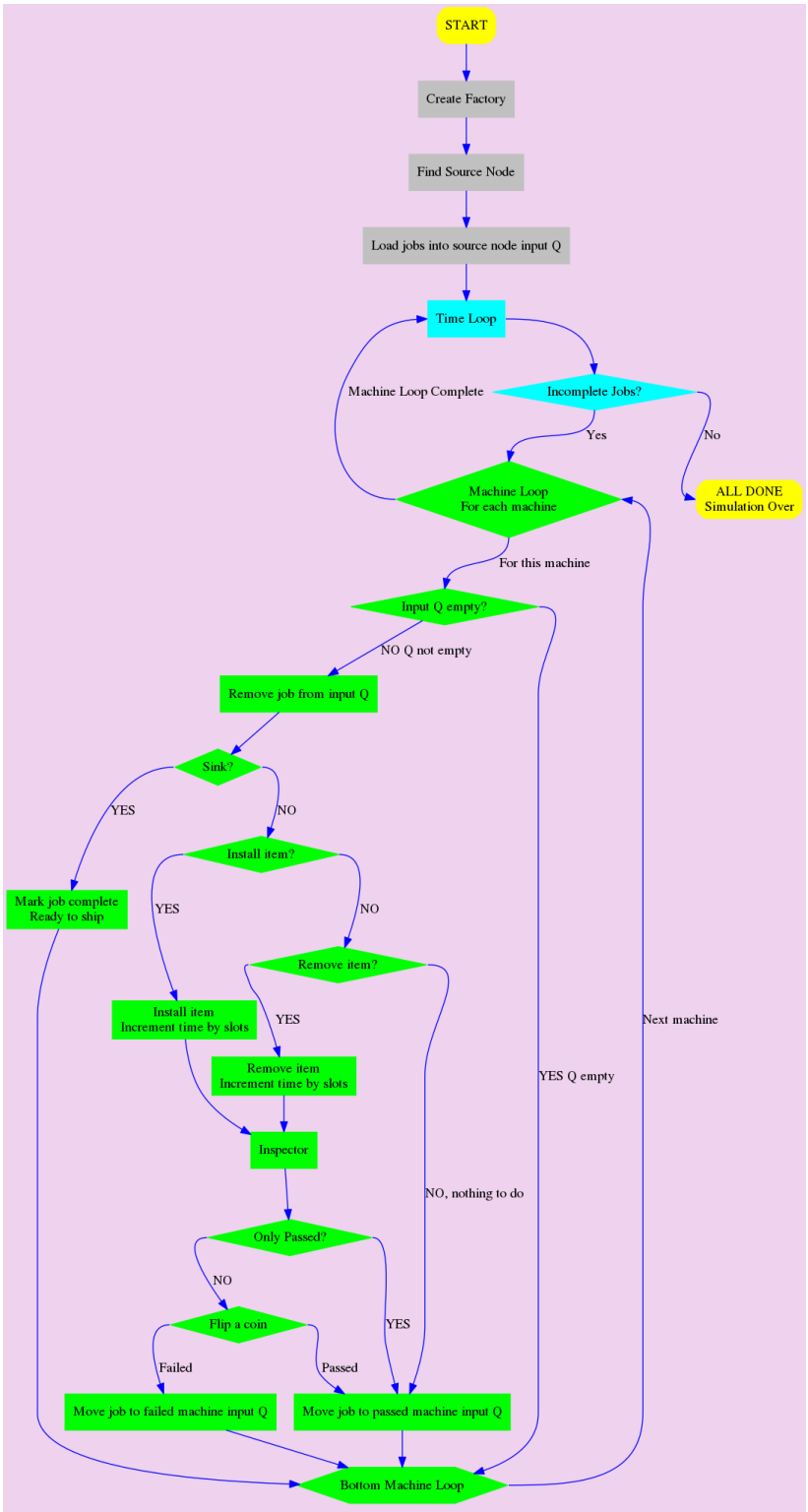
Flip a coin: calculate a random number to 'inspect' the job.
If the inspection failed

18. Move the job to the failed machine input queue

If the inspection passed

goto 16
19. End machine loop

Factory Flow Chart



MILESTONE 5
MINIMAL VIABLE SKELETON FACTORY
(assume slots is time units required for a task to complete)
FILE m5.gv.dot.png

How To Succeed in Writing Milestone 5

Coding like most things in life is an exercise in organization. The secret is to understand what you are trying to do. If you do not know what you are trying to do, you cannot do it.

Look at the flow chart. Simple, isn't it?

We recommend and use the Agile development method:

Write simple code starting at the top of the flow chart, moving toward the bottom. Keep it simple (KISS). Take baby steps. Do not over-think what you are attempting to do. Functions should only be added or enhanced when required by the step you are working on. Do not over-think what you are doing by writing code required for future steps. Yes all sorts of things need to be written for the C++ classes. They will be written as required. This saves a lot of time by keeping everything simple as possible at each step. Simple code is easy to write.

Test each function as it is written. Print out the function result. Verify it is producing what you expect before you move on.

Complexity is the enemy of organization. Complex things rarely work. They are expensive to develop. They are hard to test. They have bugs. They take a long time to code, and test.

Agile development works. It is commonly used in industry.

The latest Agile development rage is pair programming. This is where two programmers sit down at the same computer and write the code together. Each programmer looks over the shoulder of the other, catching logic errors, and improving the quality of code, simplifying, pondering better ideas for the programming task at hand. Yes at first it appears to double the cost of writing code. Not quite true. It reduces the total cost of producing code with far fewer errors. The cost saving results from not have to fix as many bugs.

Let's Do It!

Look at the flow chart.

Step 1: Load the factory.

Copy iot.cpp to factory.cpp.

Inside factory.cpp, Create a class Factory and a class Machine.

Machine is derived (inherits) from Task. At this step, that's it. Do not overthink the problem. Let Machine be empty.

Factory has a list of Machines, one machine per Task. Create a vector of Machines.

Print out the list of machines.

Step 2: Find the source node.

See the above decription on how to find a source node in a graph.

For class Machine, add a counter for the number of in-coming edges.

For each Machine in the Factory, increment the counter for Machine Pass and Machine Fail.

Add a isSource member function to Machine

bool Machine::isSource() { return counter == 0; }

Check that there is only one source node. Ignore singleton nodes.

For singletons, you need to know if the node is both a sink and a source. Add a isSink member function to machine.

Bool Machine::isSink() { return taskPass.empty(); }

Bool Machine::isSingleton() { return taskPass.isSource() && taskFail.isSink(); }

Step 3. Load Jobs into the source node input Q.

Create a class Job that derived(inherits) from class Order. Leave the class empty.

Add a Job queue member variable into class Machine. (Each machine has a inut job queue.) Do not overthink the problem. Let Job be empty. Yes, more things will be added to job. Add them later.

Load all the jobs into the job queue for the source machine. Use std::move to add jobs to the input queue.

Print out the job queue size for each machine. The source machine should have all the jobs. The other machine input queues should be empty.

If you want to print the job queue, there is a problem. STD::QUEUE does not support iterators. We can only push and pop things off the queue. We cannot dump (print) a queue by walking it with an iterator. We can print the size of a std::queue. Keep things simple and print the queue sizes. If you need to walk the queue, change std::queue to std::list and use iterators to walk the list.

Step 4. Time Loop

Create a variable time. Initialize it to zero. Create a simple while loop.

Step 5. Incomplete Jobs?

We need to capture the state of items in a order, which items are installed and which items need to be installed. Create a vector<bool> ‘installed’ member variable in class Job. The size of ‘installed’ is the number of items in the order. Initialize the vector to all ‘false’ (item not installed).

To decide if there are incomplete jobs, examine the list of machines. If any machine has a non-empty input queue, that machine has an incomplete job, enter the machine loop.

If all jobs are complete, everything has been assembled. You are done. Print an ‘All Done’ mesage and exit.

Step 6 Machine loop.

For each machine

Step 7 Input Q Empty?

This machine has nothing to do if the input job queue is empty.

If it not empty:

Step 8 Remove job from the input queue.

Use a std::move operation to move the job from th head of the input queue to a local variable ‘job’.

Step 9 Sink?

If this node is a sink and a job was routed here, we are finished with this job.

Print a message stating the job is finished.

Step 10 Install Item?

> Step 10 prep.
>
> We need to identify whether this machine is an installer or a remover
>
> Add some code to the machine constructor that classifies the machine as an installer or a remover. We can do this by looking at the item data.
>
> Calling getTaskName() returns our task name.
>
> Look through the item data installer task and remover task fields. If one matches our task (machine) name set a flag (in the class Machine) noting whether this machine is an installer or a remover.

If this machine is an installer, examine the bool ‘installed’ vector looking for a item that is not installed. For each item in the job that is not installed, look up the installer name from the item data. If the installer name matches our task name, we need to install the item.

Step 11 Install item

Set the bool ‘installed’ flag to true for this item.

Increment time by one.

Proceed to Step 14, Inspector.

Step 12 Remove Item?

If this machine is a remover, examine the bool ‘installed’ vector looking for a item that is installed that uses our task as a remover. Note you need to examine the ‘installed’ vector backwards, from the end. We need to remove the last item that matches our remover name.

If there is nothing to remove, move the job to the input queue of the ‘passed’ machine.

Step 13 Remove item

Set the bool ‘installed’ flag to false for this item.

Increment time by one.

Proceed to Step 14, Inspector.

Step 14 Inspector

If the taskFail field is empty, route the job to the input queue of the ‘passed’ machine.

If there is a taskFail, flip a coin. Calculate a random number.

If the number is odd, assume the job passed. Route it to the input queue of the ‘passed’ machine.

Otherwise, the job failed. Route it to the input queue of the ‘failed’ machine.

That’s all for a minimal viable factory simulation.

Look at the flow chart. We are done. Nothing else to do for a minimal viable factory simulation.

What’s next?

We have a working minimal factory. We can enhance it to do more.

Out factory as-built does not install mulitple instances of an item. There are two easy methods to add multiple item support.

1. Change pass. If a machine installs an item and the test is successful meaning the item normally would be routed to ‘pass’, check to see if the job requests more items of the same type. If there are, route the job to the machine’s input queue. If not, route the job to ‘pass’.
2. Change sink node logic. Check to see if all items for the job are installed. If not, route the job

to the source node for the factory.

Task has a un-documented numeric field ‘slots’. Why not interprete ‘slots’ as the number of time units or time ‘slots’ for a task to complete its function? Instead of incrementing the time counter by ‘1’, increment it by ‘slots’.

The item file includes a sequence number. Add a sequence number field for each item to a job. When an item is installed, copy the item sequence number to the job sequence number. Increment the item sequence number.

Modify the print functions to print things in a table where the columns line up. This will require two passes. The first pass measures the maximum column width for each column. The second pass prints the table with column headers respecting the maximum column wdth for each column.

Appendix 1 - Simple Data Files

SimpleItem.dat

```
I5  | Install CPU | Remove CPU | 300 | Intel I5 Central Processing Unit

I7  | Install CPU | Remove CPU | 400 | Intel I7 Central Processing Unit

A12 | Install CPU | Remove CPU | 500 | AMD A12 APU – Accelerated Processing Unit (CPU+GPU in same chip)

DDR 266 | Install Memory | Remove Memory | 125 | Samsung DDR 266 Memory Stick

DDR 400 | Install Memory | Remove Memory | 940 | Samsung DDR 400 Memory Stick

Geforce 750M | Install GPU | Remove GPU | 395 | Nvidia Geforce 750M GPU

Nano | Install GPU | Remove GPU | 30 | AMD Nano GPU

Power Supply 200 Watt | Install Power Supply | Remove Power Supply | 1100

Power Supply 300 Watt | Install Power Supply | Remove Power Supply | 9100
```

SimpleOrder.dat

```
Biance      | Dell 123 | DDR 266 | I7 | DDR 266 | Nano | Power Supply 300 Watt

Salt'N Pepa | HP 345   | A12 | DDR 400 | Geforce 750M | DDR 400 | Power Supply 300 Watt

Brianna     | Acer 567 | I5 | Power Supply 200 Watt | DDR 266 | Nano | DDR 266
```

SimpleTask.dat

```
Factory Start | 1 | Install Power Supply

Install Power Supply|4|Install Motherboard|Remove Power Supply

Remove Power Supply|2|Install Power Supply

Install Motherboard|3|Install CPU|Remove Motherboard

Remove Motherboard|1|Install Motherboard

Install CPU|5|Install Memory|Remove CPU

Remove CPU|1|Install CPU

Install Memory|4|Install SSD|Remove Memory

Remove Memory|1|Install Memory

Install SSD|4|Install GPU|Remove SSD

Remove SSD|1|Install SSD

Install GPU|3|Test| Remove GPU

Remove GPU|3|Install GPU

Test|4|Approve|Repair

Approve

Repair
```