Richard Hayes Crowley

07/19/2021

CSC\_242\_Lab\_011

**SOURCE CODE:**

**Topologies.py**

*from* simple\_term\_menu *import* TerminalMenu

*from* classes *import* Product, Asset

def navigateBatch(*batch*: Asset):

prev\_asset\_stack = []

def recursiveNavigation(*asset*: Asset, *prevAsset*: Asset or None = None):

*if* *prevAsset*:

prev\_asset\_stack.append(*prevAsset*)

print(

f"\n ~\*~\*~\*~\* Viewing {*asset*.getTag()} ({*asset*.getUnitOfMeasure()}) Children ~\*~\*~\*~ \n")

print(*asset*.getAssetTable()

*if* *asset*.assetChildren *else* "No children to view")

*while* True:

options = list(filter(lambda *item*: *item* *is* *not* None, ["View Children",

"View Child Asset" *if* *asset*.assetChildren *else* None, "View Parent Asset" *if* *prevAsset* *else* None, "View Event Log", "Update Event Log", "Exit"]))

entry = TerminalMenu(options,

*title*=f"\nWhat would you like to do with {*asset*.getTag()}?").show()

*if* options[entry] == "View Children":

print(

f"\n ~\*~\*~\*~\* Viewing {*asset*.getTag()} ({*asset*.getUnitOfMeasure()}) Children ~\*~\*~\*~ \n")

print(*asset*.getAssetTable())

*pass*

*if* options[entry] == "Update Event Log":

print(

f"\nUpdating event log for {*asset*.getTag()}. \nNOTE: Updates are recursive and will update the event log for all of this assets' children, as well.")

event = input(

str("Please enter an event (e.g., Lost, Damaged, Arrived at Location, Scan): "))

*asset*.updateEventLog(event)

print("Update success!")

*pass*

*if* options[entry] == "View Event Log":

print(

f"\n ~\*~\*~\*~\* Viewing {*asset*.getTag()} ({*asset*.getUnitOfMeasure()}) Event Log ~\*~\*~\*~ \n")

print(*asset*.getEventLogTable())

*pass*

*if* options[entry] == "View Child Asset":

asset\_li = *asset*.getListIterator()

asset\_li.first()

items = [asset\_li.next() *for* i *in* range(len(*asset*))]

entry = TerminalMenu([i.getTag() *for* i *in* items],

*title*="\nWhich child asset would you like to view?").show()

assetToView = items[entry]

recursiveNavigation(assetToView, *asset*)

*break*

*if* options[entry] == "View Parent Asset":

recursiveNavigation(prev\_asset\_stack.pop(), prev\_asset\_stack.pop() *if* len(

prev\_asset\_stack) > 0 *else* None)

*break*

*if* options[entry] == "Exit":

*break*

*return* recursiveNavigation(*batch*)

def createBatch():

batch\_name = str(input("\nEnter a name for your batch: "))

product = Product(str(input("\nWhat product is in the batch?: ")))

palletCount = int(

input("How many pallets in the batch? Integer only: "))

caseCount = int(

input("How many cases per pallet? Integer only: "))

itemCount = int(

input("How many items per case? Integer only: "))

*# recursive list comprehension builds a linked list of linked lists... this is our asset hierarchy*

batch = Asset(product, "batch", [Asset(product, "pallet", [Asset(product, "case", [Asset(product, "item", None, *tag*=f"{product}-{pallet+1}-{case+1}-{item+1}")

*for* item *in* range(itemCount)], *tag*=f"{product}-{pallet+1}-{case+1}") *for* case *in* range(caseCount)], *tag*=f"{product}-{pallet + 1}") *for* pallet *in* range(palletCount)], *tag*=batch\_name)

*return* batch

def main():

'''

Topologies.py is a proof of concept application that allows warehouse managers to create digital representations of assets and collections of assets, called "batches"

Nomenclature: An "asset" is a unit of measure of a "product". "Batches" are collections of "assets", called "units of measure"

Assets come in three hierarchical units of measure: Pallets, Cases, and Items

The hierarchical units of measure have parent/child relationships. Each asset has assetChildren, except for items.

Each asset has an "event log" that consists of several date/time and an event such Arrive at Location, Damaged, Lost

Updates to the event log are recursive and traverse hierarchies from the top down, so that e.g. an update to a Pallet will update all cases and items within that pallet, but will not update cases and items within the other pallet, etc.

Users of this proof-of-concept can do the following:

- Create batches, navigate batch hierarchies, and delete batches

- Update event log for batch or asset in a batch

- View event log for batch or asset in a batch

'''

print("\n~\*~\*~\* Welcome to Topologies, a warehouse management tool. ~\*~\*~\*~\*\n")

batches = []

*while* True:

entry = TerminalMenu(["Create a batch", "View batch", "Delete a batch",

"Exit"], *title*="\nWhat would you like to do?").show()

*if* entry == 0:

batch = createBatch()

batches.append(batch)

print(f"\n{batch.getTag()} batch created! Top level of batch is: \n")

print(batch.getAssetTable())

*elif* entry == 1:

*if* len(batches) <= 0:

print("No batches created yet!")

*else*:

batch\_to\_view = TerminalMenu(

[i.getTag() *for* i *in* batches], *title*="\nWhich batch would you like to view?").show()

print(

f"\nViewing {batches[batch\_to\_view].getTag()}, created at {batches[batch\_to\_view].getTimeCreated()}")

navigateBatch(batches[batch\_to\_view])

*elif* entry == 2:

*if* len(batches) <= 0:

print("No batches to delete!")

*else*:

batch\_to\_delete = TerminalMenu(

[i.getTag() *for* i *in* batches], *title*="\nWhich batch would you like to remove?").show()

print(f"\nDeleting {batches[batch\_to\_delete].getTag()}")

batches.pop(batch\_to\_delete)

*else*:

print("\nGoodbye!")

exit()

*if* \_\_name\_\_ == "\_\_main\_\_":

main()

**classes.py**

*from* LinkedList.LinkedList *import* LinkedList

*from* LinkedList.LinkedListIterator *import* LinkedListIterator

*from* datetime *import* datetime

*from* tabulate *import* tabulate

class Product(object):

def \_\_init\_\_(*self*, *name*: str) -> None:

super().\_\_init\_\_()

*self*.name = *name*

def \_\_str\_\_(*self*):

*return* str(*self*.name)

def getName(*self*):

*return* *self*.name

class Event(object):

def \_\_init\_\_(*self*, *assetTag*: str, *type*: str) -> None:

super().\_\_init\_\_()

*self*.type = *type*

*self*.assetTag = *assetTag*

now = datetime.now()

current\_time = now.strftime("%H:%M:%S")

*self*.time\_of\_log = current\_time

def \_\_str\_\_(*self*):

*return* f"{*self*.type} : {*self*.time\_of\_log}"

def getTimeOfLog(*self*):

*return* *self*.time\_of\_log

def getEvent(*self*):

*return* *self*.type

def getAssetTag(*self*):

*return* *self*.assetTag

class Asset(LinkedList):

def \_\_init\_\_(*self*, *product*: Product, *unitOfMeasure*: str, *assetChildren*, *tag*: str) -> None:

super().\_\_init\_\_(*assetChildren*)

*self*.tag = *tag*

*self*.assetChildren = *assetChildren*

*self*.eventLog = [Event(*tag*, "origin")]

*self*.product = *product*

*self*.unitOfMeasure = *unitOfMeasure*

*self*.list\_iterator = LinkedListIterator(*self*)

*self*.list\_iterator.first()

now = datetime.now()

current\_time = now.strftime("%H:%M:%S")

*self*.time\_created = current\_time

def \_\_str\_\_(*self*):

*return* f"{*self*.unitOfMeasure} {*self*.product} {*self*.assetChildren}"

def \_\_updateEventLog(*self*, *event*: Event):

*self*.eventLog.append(*event*)

def getListIterator(*self*):

*return* *self*.list\_iterator

def getTag(*self*):

*if* *self*.tag:

*return* str(*self*.tag)

def getUnitOfMeasure(*self*):

*return* *self*.unitOfMeasure

def getProduct(*self*):

*return* *self*.product

def getEventLog(*self*):

*return* *self*.eventLog

def getTimeCreated(*self*):

*return* *self*.time\_created

def updateEventLog(*self*, *event*: str):

newEvent = Event(*self*.getTag(), *event*)

*self*.\_\_updateEventLog(newEvent)

*# reset cursor*

*self*.list\_iterator.first()

*while* *self*.list\_iterator.hasNext():

theNode = *self*.list\_iterator.next()

theNode.updateEventLog(*event*)

*# reset cursor*

*self*.list\_iterator.first()

def getAssetTable(*self*):

data = []

*for* i *in* *self*:

data.append([i.tag, i.unitOfMeasure, ", ".join(

[child.tag *for* child *in* i.assetChildren] *if* i.assetChildren *else* []), i.time\_created, i.eventLog[len(i.eventLog)-1]])

*return* tabulate(data, *headers*=[

"Asset tag", "Unit of Measure", "Asset Children", "Time Created", "Last Event"], *tablefmt*="github", *numalign*="left")

def getEventLogTable(*self*):

eventLog = *self*.getEventLog()

data = [(e.getAssetTag(), e.getEvent(), e.getTimeOfLog())

*for* e *in* eventLog]

data.sort(*reverse*=True)

*return* tabulate(data, *headers*=["Asset", "Event", "Time of Log"], *tablefmt*="github", *numalign*="left")

**LinkedList.py**

*from* LinkedList.node *import* TwoWayNode

*from* LinkedList.AbstractList *import* AbstractList

class LinkedList(AbstractList):

def \_\_init\_\_(*self*, *sourceCollection*=None) -> None:

'''uses a circular structure with a "dummy" sentinel node

the empty TwoWayNode instance self.head is the sentinel node...

the cursor will set its head NOT to the head node, which is the dummy sentinel, but to the next node, which is the first node containing data (if it exists)

when the cursor cycles around to the dummy head sentinel node, the iterators loop terminates.

'''

*self*.head = TwoWayNode()

*self*.head.previous = *self*.head.next = *self*.head

super().\_\_init\_\_(*sourceCollection*=*sourceCollection*)

*# accessors*

def \_\_iter\_\_(*self*):

cursor = *self*.head.next

*while* cursor != *self*.head:

*yield* cursor.data

cursor = cursor.next

*# helpers*

def getNode(*self*, *i*):

'''Helper method that returns a pointer to the node at position i'''

*# constant time access to head node*

*if* *i* == len(*self*):

*return* *self*.head

*# or last data node*

*if* *i* == len(*self*) - 1:

*return* *self*.head.previous

probe = *self*.head.next

*while* *i* > 0:

probe = probe.next

*i* -= 1

*return* probe

def \_\_getitem\_\_(*self*, *i*):

*if* *i* < 0 or *i* >= len(*self*):

*raise* IndexError("List index out of range")

*return* *self*.getNode(*i*).data

*# mutators*

def \_\_setitem\_\_(*self*, *i*, *item*):

'''replaces element at i with item'''

*if* *i* < 0 or *i* >= len(*self*):

*raise* IndexError("List index out of range!")

*self*.getNode(*i*).data = *item*

def insert(*self*, *i*, *item*):

*if* *i* < 0:

*i* = 0

*elif* *i* > len(*self*):

*i* = len(*self*)

theNode = *self*.getNode(*i*)

newNode = TwoWayNode(*item*, theNode.previous, theNode)

theNode.previous.next = newNode

theNode.previous = newNode

*self*.size += 1

*self*.incModCount()

def pop(*self*, *i*=None):

*# return last data node if index not specified*

*if* *i* == None:

*i* = len(*self*) - 1

*if* *i* < 0 or *i* > len(*self*) - 1:

*raise* IndexError("List index out of range")

popped = *self*.getNode(*i*)

popped.previous.next = popped.next

popped.next.previous = popped.previous

*self*.size -= 1

*self*.incModCount()

*return* popped.data

def replace(*self*, *i*, *item*):

'''do not increment mod count with replace method'''

*if* *i* < 0:

*i* = 0

*elif* *i* > len(*self*):

*i* = len(*self*)

theNode = *self*.getNode(*i*)

newNode = TwoWayNode(*item*, theNode.previous, theNode.next)

*# break link of the node being replaced*

theNode.previous.next = newNode

theNode.next.previous = newNode

*return* theNode.data

**LinkedListIterator.py**

class LinkedListIterator(object):

def \_\_init\_\_(*self*, *backingStore*) -> None:

super().\_\_init\_\_()

*self*.backingStore = *backingStore*

*self*.modCount = *backingStore*.getModCount()

*self*.first()

def first(*self*):

'''resets cursor to the beginning of the backing store'''

*self*.cursor = 0

*self*.lastItemPos = -1

def getPosition(*self*):

*return* *self*.lastItemPos

*# Navigational methods*

def hasNext(*self*):

'''returns true if the iterator has a next item or False otherwise'''

*return* *self*.cursor < len(*self*.backingStore)

def next(*self*):

''' preconditions: hasNext is true and the list has not been modified EXCEPT by this iterator's mutators

Returns the current item and advances the cursor to the next item

'''

*if* not *self*.hasNext():

*raise* ValueError("No next item in list iterator")

*if* *self*.modCount != *self*.backingStore.getModCount():

*raise* AttributeError("Illegal modficiation of backing store")

*self*.lastItemPos = *self*.cursor

*self*.cursor += 1

*return* *self*.backingStore.getNode(*self*.lastItemPos).data

def last(*self*):

'''Moves cursor to end of backing store'''

*self*.cursor = len(*self*.backingStore)

*self*.lastItemPos = -1

def hasPrevious(*self*):

*return* *self*.cursor > 0

def previous(*self*):

'''preconditions: hasPrevious returns true and the list has not been modified EXCEPT by this iterator's mutators

Returns the current item and moves the cursor to the previous item

'''

*if* not *self*.hasPrevious():

*raise* ValueError("No previous item in list iterator")

*if* *self*.modCount != *self*.backingStore.getModCount():

*raise* AttributeError("Illegal modficiation of backing store")

*self*.cursor -= 1

*self*.lastItemPos = *self*.cursor

*return* *self*.backingStore.getNode(*self*.lastItemPos).data

*# Mutators*

def replace(*self*, *item*):

'''preconditions: the current position is defined, and the list has not been modified EXCEPT by this iterator's methods'''

*if* *self*.lastItemPos == -1:

*raise* AttributeError("The current position is undefined")

*if* *self*.modCount != *self*.backingStore.getModCount():

*raise* AttributeError("List has been modified illegally")

nodeReplaced = *self*.backingStore.replace(*self*.lastItemPos, *item*)

*self*.lastItemPos = -1

*return* nodeReplaced

def insert(*self*, *item*):

'''Preconditions: the list has not been modified except by this iterator's methods'''

*if* *self*.modCount != *self*.backingStore.getModCount():

*raise* AttributeError("List has been modified illegally")

*if* *self*.lastItemPos == -1:

*# cursor not defined, so add item to end of list*

*self*.backingStore.add(*item*)

*else*:

*self*.backingStore.insert(*self*.lastItemPos, *item*)

*self*.lastItemPos = -1

*self*.modCount += 1

def remove(*self*):

'''preconditions: the current position is defined, and the list has not been modified EXCEPT by this iterator's methods'''

*if* *self*.lastItemPos == -1:

*raise* AttributeError("The current position is undefined")

*if* *self*.modCount != *self*.backingStore.getModCount():

*raise* AttributeError("List has been modified illegally")

item = *self*.backingStore.pop(*self*.lastItemPos)

*# if the item was obtained via next, move cursor back*

*if* *self*.lastItemPos < *self*.cursor:

*self*.cursor -= 1

*self*.modCount += 1

*self*.lastItemPos = -1

*return* item

**OUTPUT:**

**PLEASE SEE ATTACHED VIDEO ☺**