Project 6

Project title: Evaluating cross-lingual projection of semantic role labels and inter-annotator agreement in Python

Team members: Adam Pollins

Final state of system

As of Project 4, it was mentioned that the iterator and composite patterns had been implemented. However, key to getting this to work ideally was a fully recursive composite iterator. With few resources on how to do this in Python, trying to get this to work consumed much time and was ultimately not fruitful. What the system can do now is produce a bare version of one intended use case. (The f-score of around 8% is probably accurate, as the group I originally got this data from had yet to produce good data, and this use case would be for future use.)

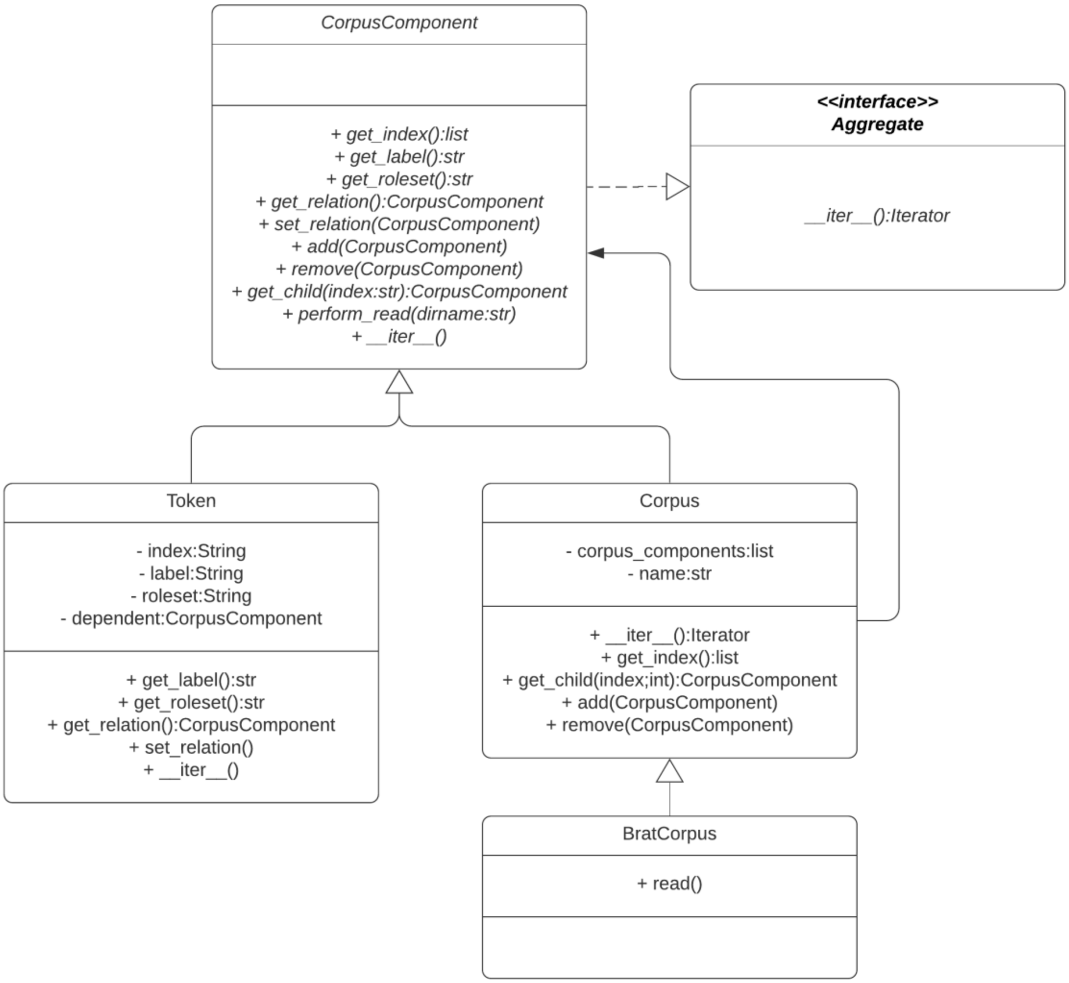
The calculation of the balanced f-score requires (a) the number of times two tags coincide between two sets (b) the number of times one tag occurs in the first set but not the second (c) the number of times one tag occurs in the second set but not the first. The formula is then (2 * a) / ((2 * a) + b + c). It occurred to me too late that the best way to calculate a, b, and c is through set operations which are ill-suited to a tree. Therefore, everything had to be read out into a set, and this part was not done through OO, but using a single function.

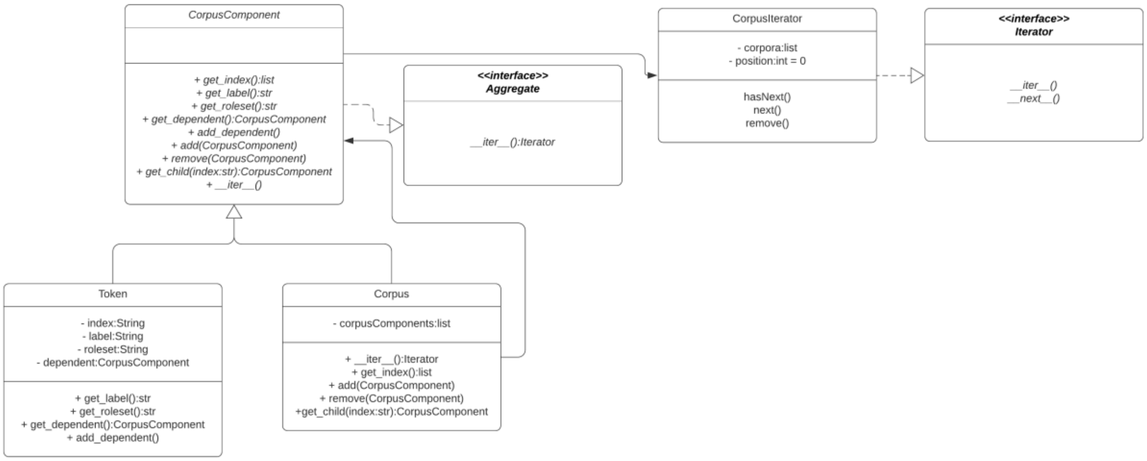
Printing the data was not an intended use case. However, it does read in the file and print it out in a different format which is actually far more human-readable than the original .ann files. Therefore, this is submitted in lieu of one of the originally desired further use cases.

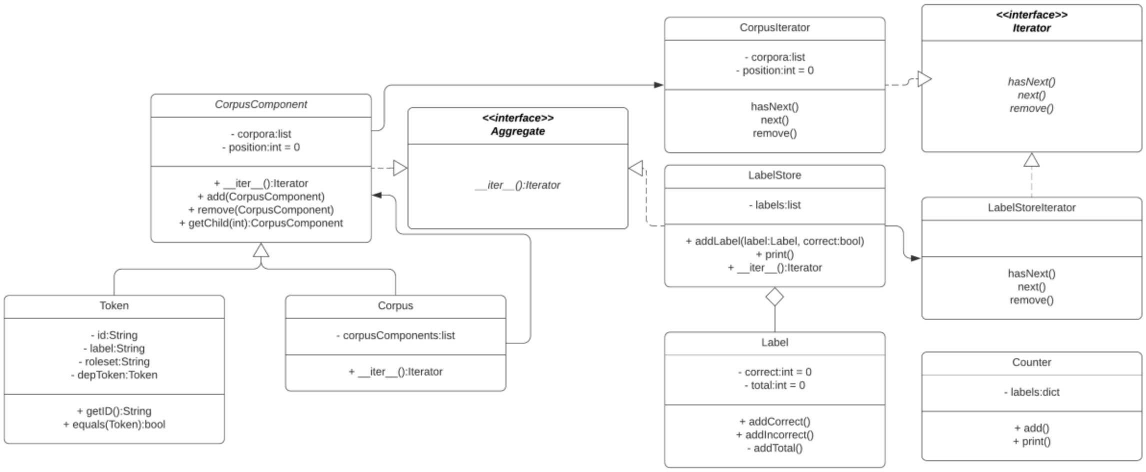
The “constituency-to-dependency” version use case is not implemented, but you can see the beginnings of the Strategy pattern which would have been used to encapsulate the algorithm to read in a completely differently formatted file into the same composite tree structure.

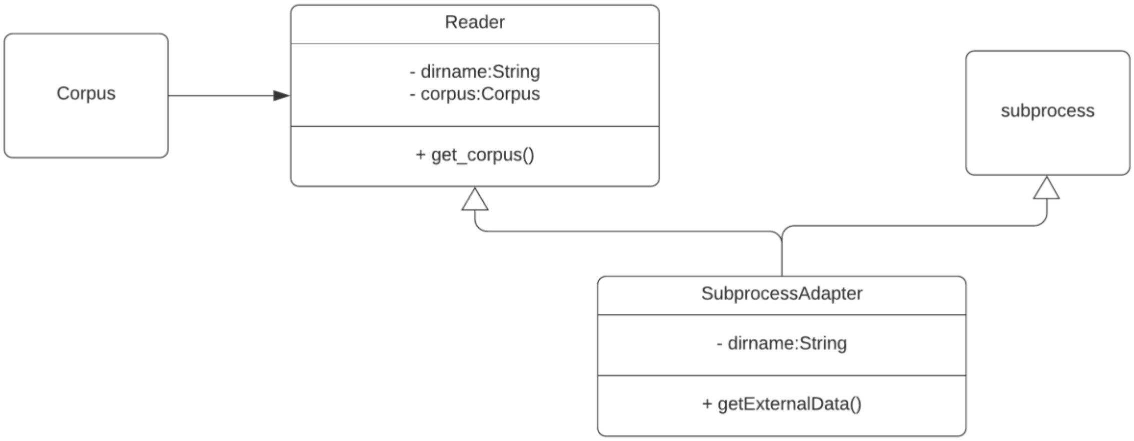
Final class diagram

Diagrams are presented in reverse chronological order


CorpusComponent
+ get_index():list
+ get_label():str
+ get_roleset():str
+ get_relation():CorpusComponent
+ set_relation(CorpusComponent)
+ add(CorpusComponent)
+ remove(CorpusComponent)
+ get_child(index:str):CorpusComponent
+ perform_read(dirname:str)
+ __iter__()
<<interface>>
Aggregate
__iter__():Iterator
Token
- index:String
- label:String
- roleset:String
- dependent:CorpusComponent
+ get_label():str
+ get_roleset():str
+ get_relation():CorpusComponent
+ set_relation()
+ __iter__()
Corpus
- corpus_components:list
- name:str
+ __iter__():Iterator
+ get_index():list
+ get_child(index;int):CorpusComponent
+ add(CorpusComponent)
+ remove(CorpusComponent)
BratCorpus
+ read()


CorpusComponent
+ get_index():list
+ get_label():str
+ get_roleset():str
+ get_dependent():CorpusComponent
+ add_dependent()
+ add(CorpusComponent)
+ remove(CorpusComponent)
+ get_child(index:str):CorpusComponent
+ __iter__()
<<interface>>
Aggregate
__iter__():Iterator
CorpusIterator
- corpora:list
- position:int = 0
hasNext()
next()
remove()
<<interface>>
Iterator
__iter__()
__next__()
Token
- index:String
- label:String
- roleset:String
- dependent:CorpusComponent
+ get_label():str
+ get_roleset():str
+ get_dependent():CorpusComponent
+ add_dependent()
Corpus
- corpusComponents:list
+ __iter__():Iterator
+ get_index():list
+ add(CorpusComponent)
+ remove(CorpusComponent)
+get_child(index:str):CorpusComponent





This will run the subprocess to read all the external data from the files using the third-party tools.

Comparison

The code ultimately evolved to exclude the Adapter pattern and include the beginnings of the Strategy pattern. The Strategy pattern was meant to encapsulate the read() function for different file formats. Other than this, I changed a few method names and corrected a few mistakes with parameters.

It is somewhat embarrassing that such heavy effort on the backend delivered so little functionality to demonstrate, as the cor. However, the actual composite and iterator patterns are promising in representing data contained in a directory structure, and this code could be reused in the future and appears more maintainable than functional solutions. Though the read function is a bit of a code blob, it is actually much more maintainable than previous solutions to this problem where the code is in a main method and employs the built-in data structures.

An iterator interface was not needed and this is not normally how the iterator protocol is implemented in Python, so it was discarded.

Third-party code vs. original code

A few lines of the code in the read() method, for reading in the files, was adapted from work I did before this course, but this is not posted publicly. The method is largely rewritten, because it uses the composite data structure instead of the Python built-in data types I had previously relied on. Composite Iterator was adapted from Head-First Design Patterns. The original second use case was slated to use more third-party software that was not incorporated.

Statement on the OOAD process

1) It is fairly difficult to know exactly what patterns to incorporate before development with little experience with these patterns. For instance, it was originally decided to use the adapter pattern. However, if the algorithms that called for the adapter pattern had been fully implemented, this probably wouldn't be necessary and could just be included in those strategy pattern methods.
2) Patterns you are familiar with may emerge late in development. In this case, it was not apparent that the strategy pattern was suitable to encapsulate different blocks of "reading-in" code until the actual obstacle arose during implementation.
3) Stamping out functional practices entirely is extremely difficult to do from the beginning and probably requires several iterations. The to_set() method in the main module was needed fairly late in development, and adding this as another object after already designing everything else probably would have required at least a partial redesign.