Concurrent-UMLs

Overview:

By leveraging java's concurrency and using ReentrantLock to avoid thread deadlocking, the software is able to parse through Java files and maintain the class information which then will be translated into PlantUML syntax("a highly versatile tool that facilitates the rapid and straightforward creation of a wide array of diagrams"). Each individual file will be entered into a BlockingQueue for task coordination, and an ExecutorService to enable parallel file parsing via thread pools and a CountDownLatch will be used so that each thread reaches completion. This will then send you to a link to view the class diagram!

Analysis / Design:

UMLModel.Java:

- This is the class that was designed to be used as a shared resource to hold all the class information such as the class\interface\abstract names, and attributes\field and method information. The attribute and method information consists of their visibility, name, return type, and if they are either final and static. Thought it was too much information so later new UMLModel instances will be used to avoid shared state corruption.
- This class uses a ReentrantLock to prevent deadlocking so that it can ensure that only one thread can execute which prevents race conditions.

DependenciesModel:

- Creates the arrows to show the relationships between classes depending on dependencies, implementations, inheritance, and other relationships.
    - Not fully thought out

FileParser.java (implements Runnable):

- Is in charge of parsing the java files for the information
    - Uses regex patterns to find the class name, attribute/field line, and method signature.
- Depending on the brace count, it will identify if the method body has started or ended.

ParserManager.java

- PlantUmlGenerator stores all UML strings in allCurrents.
- BlockingQueue ensures thread-safe task distribution and manages file parsing tasks.
- Uses ExecutorService to create a thread pool for parallel parsing.
- CountDownLatch ensures graceful shutdown after all threads finish.
- The file that is created in plantuml syntax can be added into Draw.io to create the image there as well, but my program doesn't account for this implementation.

PlantUmlGenerator:

- Generates the syntax that PlantUML uses to create the diagrams.

PlantUmlEncoder:

- After getting the information in the syntax that PlanUML wants, it is
    - Encoded in UTF-8
    - Compressed using Deflate algorithm
    - Reencoded in ASCII using a transformation close to base64
- This is then added to the website path to receive that ParserManager uses.

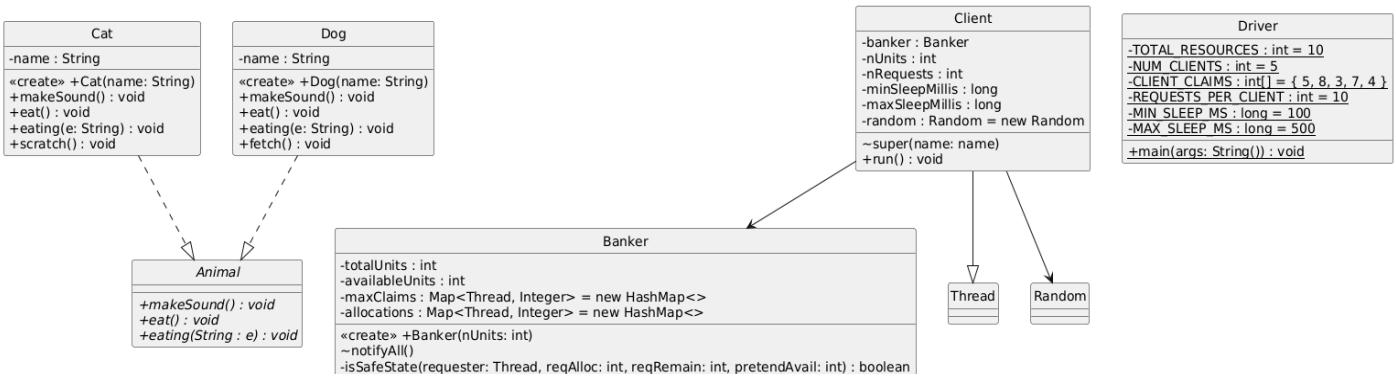
Test Plan:

I used multiple files to test for different edge cases

- Whitespace
- Braces (when does a method start, info in a method body)
- Regular variable vs attribute/field

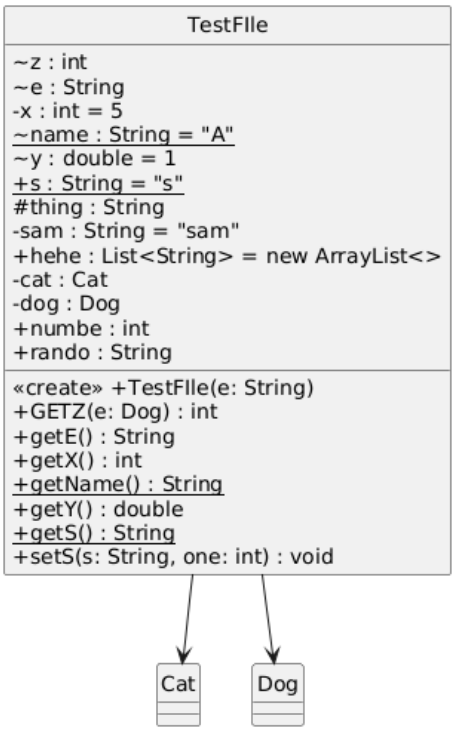
TestingFile.java shows my thought process of how to go about this. After figuring this out, I needed to see how it was different from an interface. I wanted to create a test for this, but I wanted to just print out the current progress of what I had from time to time and save it in current_output.txt.

Project Results:

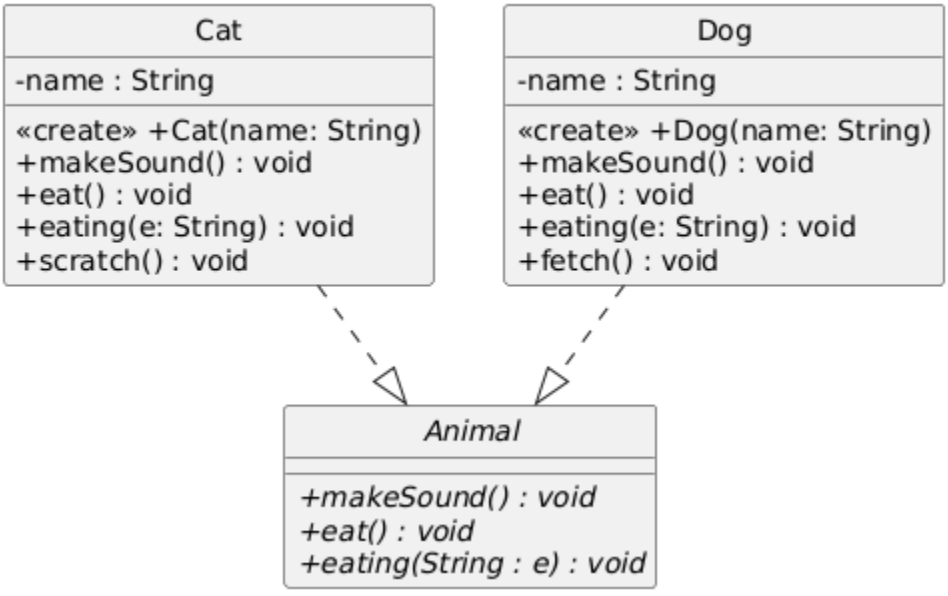


link to img

^link to image that my program creates



link to img



link to img

Lessons Learned:

Overall, it is still very buggy since there are some things I need to still take account for that I don't think of until I reach to that obstacle, but I enjoyed this project since I started this out before coming to this class, and attempted it, but was always very lazy to do so which may have been from frustration of applying certain design patterns before. I still need to take into account multilined items and other formats. I find it ironic that I don't make a class diagram for code that was used to create this, but it would look too messed up.