Auto UML Generators for Python

Vihan Patel

Introduction

This paper is going to discuss about an approach taken towards the development of an auto UML generator in python. While there are many languages that one could use to achieve an auto UML generator for python, I have decided to use C++ as the primary language for this project. Like every programming language, C++ has certain benefits and drawbacks when being used to create an auto UML generator. The paper will illustrate techniques used to determine class relationships as well as other important information present in a UML diagram.

C++ As a Primary Tool

When it comes to an auto UML generator, one needs to think about the necessary components when creating one. One requirement for an auto UML generator is that it needs to show correct results visually for an arbitrary number of files. Secondly, efficiency is important when a user is utilizing an auto UML generator. Surely, it would not be enjoyable for one to wait for several minutes while an auto UML generator is processing through twenty or thirty files. The main motivation behind an Auto UML generator is to create a UML diagram more accurately and faster than an average person.

C++ is a great tool for an auto UML generator as the primary goal of the language is efficiency. Efficiency plays a big role when a creating a program that contains several thousand
lines of code. In addition, C++ is very useful when reading a file. Given that C++ is very good at interfacing with other languages, reading a python file can be done with less effort as opposed to a language such as Java.

Achieving efficiency in an Auto UML generator may normally come with with certain hardships. When it comes to aiming for a very good performance, features such as bounds checking for vectors or other data structures should not be present in software. Since these features are absent when trying to make an efficient software, many bugs can be introduced and lead to undefined behavior. Undefined behavior in my Auto UML generator included randomly occurring segmentation faults. My Auto UML generator would produce segmentation faults at times and it appeared to work perfectly well at other times. I have conducted certain trials when detecting any hidden bugs in my programs. For one trial, I have ran my Auto UML generator at least 10 times and they have appeared to work perfectly fine. However, my program terminated abnormally with a segmentation fault when I ran it the 11th time.

Since achieving efficiency can lead to many problems regarding undefined behavior, it can truly sacrifice productivity as one would have to take several hours to fix them. Although debuggers such as lldb and gdb can detect segmentation faults, it may fail to detect undefined behaviors exhibited by a program.

When comparing C++ with other programming languages for the design of an auto UML

generator, C++ gives a rewarding end result although much effort and time is needed to attain it. Writing a UML generator in python may be highly productive compared to C++, but it may be slow while executing it.

Graphics for UML generation

It is important to note that the C++ compiler does not provide default support for graphics libraries. However, it is not highly difficult to set a graphics library that would be used for visual output of UML diagrams. This is assuming that one has homebrew or something similar installed on their computer. C++ is flexible in that it will allow many graphics libraries such as OpenGL, SDL2 and SDL to be used. It is not necessary to download them from an online source. In fact, downloading them from online will not ensure that they are properly set up for usage. Installing a graphics library through an operating system such as Linux will ensure that an external graphics library can be linked with one's computer architecture (i.e. x86_64) in the future. For my auto UML generator, I have used the SDL2 graphics library in addition to SDL2_ttf (SDL2's truetype font library) and SDL2_image (SDL2's image library).

Figure 1 - SDL2_image, SDL2_ttf, and SDL2 installation on Linux

```
Vihans-MacBook-Pro:~ vihanpatel$ brew install sdl2_image
Vihans-MacBook-Pro:~ vihanpatel$ brew install sdl2_ttf
Vihans-MacBook-Pro:~ vihanpatel$ brew install sdl2
```

Figure 2 - Inclusion of SDL2 files in my Auto UML generator program

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_ttf.h>
#include <SDL2/SDL_image.h>
```

Figure 3 - Linking SDL2 libraries during compilation of my Auto UML generator

```
g++ visual.cpp -lSDL2 -lSDL2_image -lSDL2_ttf
```

As shown in the figures above, using graphics in C++ may take additional steps as compared to Java and python, but setup of graphic libraries in C++ do not prove to be tedious. Unlike Java, you must link the libraries in your programs. It can be easy to get a linker error when forgetting to link graphic libraries in C++ programs.

Figure 4 - A sample Linker error

```
Undefined symbols for architecture x86_64:
  "_IMG_SavePNG", referenced from:
      Graphics::~Graphics() in visual-80a9dd.o
  "_SDL_CreateRGBSurface", referenced from:
      Graphics::~Graphics() in visual-80a9dd.o
  "_SDL_CreateRenderer", referenced from:
      Graphics::Graphics(int, int, char const*, int) in visual-80a9dd.o
  "_SDL_CreateTextureFromSurface", referenced from:
      Graphics::drawText(char const*, double, double, bool) in visual-80a9dd.o
  "_SDL_CreateWindow", referenced from:
      Graphics::Graphics(int, int, char const*, int) in visual-80a9dd.o
```

Phase 1 - Python file analysis

File Parsing

The most important step that should be taken before being able to analyze class relationships and important information needed for a UML diagram is simple parsing of a python files. In C++, this can be achieved by using a vector to store all lines in a python file. Simple parsing simply involves reading every line of a file and removing all comments (i.e. multi-line comments, single line comments) as they can make it difficult for one to make an accurate analysis. Comments in python use triple quotes for multi line comments and a hash symbol for single line comments. In order to remove a single line comment, I have gone through each line of a python file that is not a whitespace and removed every character that occurred after the first instance of a hash symbol if a single line comment was present. Then, I have created a method that could remove all multi line comments. The first approach towards removing a multi line comment in python would be to store information regarding which lines have a triple quote string. Each multi line comment has a specific starting and ending location in a file. Therefore, I have created a vector to store all the start and end locations for each multi line comment. Using the vector, I retrieved all start and end locations of each multi line comment and removed every line between them.

Algorithm for removing multiline comments
if comment is docstring
  -> target character is “
else if comment is a regular multiline comment
  -> target character is ‘

vector<int> locations;
vector<string> allLines = all lines in a python file

for all lines in a python file
  if line.length() > 1 and line has target character
    count the # of times in which a target character occurs 3 times in a row
    if count >= 2

make the line blank

for all lines in a python file
  if line.length() > 1 and line has target character
    if target character occurs three times in a row
      store the location of the line in the file

for [x = 0; x < locations.size(); x += 2]
  remove all lines between locations[x] and locations[x+1] inclusive
  → locations[x] represents a start of some multiline comment
  → locations[x+1] represents an end of some multiline comment

Gathering UML information

After parsing through a file and filtering out comments, it is important to determine important pieces of information that can be derived for each class in a file. These important pieces include instance variables, class variables, method declarations and class names.

In order for these pieces of information to be derived in a satisfactory manner from my auto UML generator, I had to devise basic methods to trim whitespaces from a word and determine the first place in which a non-whitespace character occurs in line. These methods would be very useful as python is a programming language in which indentation matters. In addition, they serve as helper methods for all methods used in my Parser class.

Figure 5 - Parser class methods

```
class Parser {

  public:

   vector< vector<string> > relations(vector<string> fileNames);
   vector< vector<string> > UMLdata(vector<string> fileNames);

  private:

   void noMLComments(string type, vector<string> &allLines);
   string trim(string word);
   int firstCharPlace(string& word);
   vector<string> parse(string& fileName);
   vector<string> pass(vector<string> &list);
   vector< vector<string> > UMLinfo(string& fileName);
   vector< vector<string> > inheritance(vector<string> list);
   vector< vector<string> > composition(vector<string> list);
   vector< vector<string> > magnitude(vector<string> list);
   vector< vector<string> > getInfo(vector<string> list, vector<string> files, vector<vector<string> >& classes, bool onlyInit, bool useOtherFiles);
   void modify(vector<string>& assignments, vector<vector<string> > pieces);
   string grammarCheck(string& line);
   vector< vector<string> > aggregation(vector<string> list, vector<string> files);
   vector< vector<string> > association(vector<string> list, vector<string> files);
   vector< string > topClasses(vector<string> files);

};
```

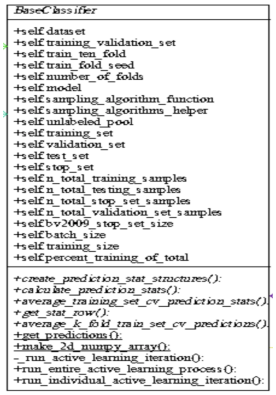
Note: As seen by the figure above, seventeen methods are created to do an analysis of python files. Fifteen of the methods use firstCharPlace and trim methods extensively.

Once I have implemented the trim and firstCharPlace methods, I was able to use them to devise a satisfactory procedure for deciding all important UML information for each class. I simply had to take advantage of the fact that each class, method and instance variable declarations for each class have a different level of indentation. This can be done if there is a vector or another data structure that stores all information about a python file and free from any comments.

The general principle for solving this problem was to start simple and make my approach increasingly detailed. I have started simple by going through comment free version of a python file. Every time I found a line that had a class declaration, I would check that the term "class " was found and that the first thing that was found in the whitespace-free version of the term was "class" . If those conditions satisfied, I would continue going through the comment free version of the python file until I saw a very first method declaration. I would use the firstCharPlace method to attain the position of the first non-whitespace character in the method declaration. This would symbolize the indentation level of the method declaration within a class and it would be needed when trying to capture class variables of a certain class and determining whether a method belonged to a class. I would go through assignment statements above the method declaration and see whether they had the same indentation level. If they did, the assignment statements would count as a class variables and be considered relevant pieces of information. If the method declaration was "def __init__" , I would perform a traversal starting below the method declaration and capture instance variables. In python, instance variables have a different indentation level compared to "def __init__" . Therefore, I had to make sure that every assignment statement that started with a "self." had a higher indentation level than "def __init__" . This traversal would continue until there was a line that had a lower indentation level than all the assignment statements that had instance variables. After capturing all necessary class and instance variables, I continued traversing for method declarations and saw if it started with "def " and had the same indentation level as the very first method declaration I had come across underneath a class declaration. I also looked for lines that had keywords such as "pass" , "__metaclass__ = ABCMeta" and "@" . This continued until I reached the end of the python file or found another class declaration. If I came across another class declaration, I would repeat the process again as I did with a first class declaration in the python file.

After gathering important information from a comment free version of a python file, I had to manipulate them. I would use decorators and the "pass" , "__metaclass__" , and "ABCMeta" keywords I retrieved to determine whether a class or method should be abstract and/or static. Then, I manipulated each class and method name accordingly.

Figure 6 - UML information of BaseClassifier class



Description: A diagram below shows a sample abstract class with static and abstract methods from classifiers.py in the src codebase. It intends to show the quality of output from methods that were responsible for gathering and displaying important UML information from python files.

Phase 2 - Analyzing Class Relationships

Aggregation

The solution to the problem of detecting aggregation in a python class can be subject to debate. It is one problem that does not have a straightforward answer. In python, one can detect aggregation by checking to see if an instance variable of one class invokes methods or instance variables of another class.

Observing methods used by instance variables of a class may be useful when making inferences about an aggregation relationship. When it comes to an instance variable invoking a method of a different class, it is necessary that an instance variable is an object of another class. When looking at an assignment statement consisting of an instance variable being equated to an expression containing a method invocation of another class, it is possible to make meaningful inferences regardless of whether a method is static or non-static. If a method is non-static, a variable calling it would have to be an object of another class. If a method were static, a classname would have to appear directly before a method invocation. The presence of variables or classnames right before a method invocation can be useful in determining which class an instance variable is an object of. Many approaches exist in determining aggregation relationships based on method calls, but the general idea is that analyzing method calls that are related to instance variables generally lead to an accurate analysis of aggregation relationships.

Unlike method calls / invocations, making inferences about instance variable invocations by an instance variable of a class can lead to many errors when detecting an aggregation relationship. This is due to the fact that many classes can share same instance variable names. In addition, there is almost little to no context clues that can be derived from other lines of code that tell which class that an instance variable being invoked belongs to.

Figure 7 - Sample code from BaseClassifier class in classifiers.py

```
# If batch size is 0, raise an error
if self.batch_size == 0:
    raise ValueError('Batch size is equal to 0. Increase batch_percent.')

# If we are doing train_ten_fold, and batch size is less than 10, raise an error
if self.train_ten_fold and self.batch_size < verify.number_of_folds:
    raise ValueError(f'Batch size of {self.batch_size} is too small to use '
                     f'{verify.number_of_folds}-fold cross validation training. Increase batch_percent.')

# Store the training size
self.training_size = len(self.training_set.ids)
```

Description - As seen in the figure, an expression with a variable, self.training_size, is equated to the len(self.training_set.ids). In the term, self.training_set.ids, ids is an instance variable of a class called Pool and invoked by an instance variable, self.training_set, of class BaseClassifier. As shown, little to no lines of code above the expression hold clues about what class owns the "ids" instance variable. This is a main limitation of making inferences based on instance variable invocations.

When I created an approach towards analyzing aggregation relationships between two classes, I was vigilant about the limitations behind using instance variable invocations as an inference tool. Therefore, I analyzed aggregation between two classes using method invocations only.

The first step towards preparing for a thorough analysis on aggregation relationships is to eliminate any source of ambiguities. Of course, one would need a data structure to store a comment free version of a python file first. In this case, I simply utilized a vector from the C++ library. When I traversed through the vector, I would check every line to see whether it contained any strings surrounded by double quotes or single quotes. At times, strings can often lead to a misleading analysis as is possible for them to consist of keywords that might make one think that an instance variable is an object of another class. Therefore, it is important to trim every string in order to avoid future errors in the analysis of aggregation relationships.

Another step taken to ensure that my analysis of aggregation relationships was going to be accurate was grouping all consecutive lines that together formed an expression with balanced parentheses and/or brackets.

In addition to removing strings and ensuring that all expressions were balanced, looking at import statements in a file is also important. Since the main goal of my auto UML generator is to have it work for an arbitrary number of python files, I would have to see that the files being imported inside of a python file are the same as the file(s) that are being input by a user. Imported files would be useful when trying to gather as much as data as possible on aggregation relationships. Firstly, I had to generate important UML information (classes and their methods) of each imported file. Secondly, I had to gather the UML information of the original python file in which I am analyzing as well.

When determining whether an instance variable is an object of a different class, it is important to count the number of parameters contained for each method of every class.

This measure would prove helpful when seeing a method invocation in which a method name might be the same for two or more classes.

Afterwards, we analyze each class contained in a python file. Since aggregation is concerned with instance variables, we only should be looking at lines that have a "self." in them and store their location relative to a comment free version of a python file. In addition to storing locations of such lines, I also had to store locations of each method declaration. This way, I could capture all assignment statements between a line containing the "self." keyword and the nearest method declaration. The variables in assignment statements that had method invocations or class names would have their values substituted inside a line containing the "self." keyword if it was appropriate.

Figure 8 - Example (from TextClassifier class in classifiers.py)

```
stat_row = super().get_stat_row()

stat_row.extend(self.test_set_metrics)
```

Description - Based on my approach, the last statement, stat_row.extend(self.test_set_metrics), would become, super().get_stat_row().extend(self.test_set_metrics).

After doing some manipulation of every line that contained "self." keyword, I had to perform a deep context analysis to see which method invocations or class names appeared. By performing a detailed analysis of these, I would be able to derive desirable results regarding aggregation relationships between classes.

Association

The approach towards association is similar to that of aggregation, but the analysis of association between two classes does not need to be highly detailed as that of aggregation. Since association is simply a loose relationship between two classes, I only had to look for lines that had important keywords regarding class names and method names within a certain class. In order to accurately determine whether a classname was being used, I had to see that a static method was being invoked in a class or that a constructor of another class was present. For method calls, I had to look for the number of parameters being used in them to accurately identify what class was being used by a certain class.

Composition

Composition is a strong form of aggregation that is defined by "part-of" relationship instead of a "has-a" relationship in aggregation. If an instance variable of Class A were to be an object of Class B, an object of Class B cannot exist when an object of class A is destroyed. Therefore, it is important to check whether an instance variable is equal to an object of another class.

Figure 9 - Example of composition relationship

```
self.output_helper = OutputStructureImageOutputHelper(**kwargs)
```

Description - This line was taken from CreateOutputStructureImage class. It shows that OutputStructureImageOutputHelper is a "part-of" CreateOutputStructureImage.

Inheritance

Finding inheritance relationships in python has a very straightforward approach. Each time one sees a class declaration, they would have to look within the parentheses of a class declaration to look for a list of classes being used. It would be important to note several exceptions can exist.

Figure 10 - Finding out inheritance relationships in python from class declarations

```
class TextClassifier(BaseClassifier, metaclass=helper.ConcreteClassAndAutoDocMeta):
```

Note: A class, TextClassifier, inherits from BaseClassifier. However, an item that has keywords such as "metaclass=" does not have anything to do with inheritance. When devising a procedure for determining inheritance, it is important to realize which items within parentheses of a class declaration do and do not count as parent classes.

Final Phase - Graphical Displays

Basic Necessities

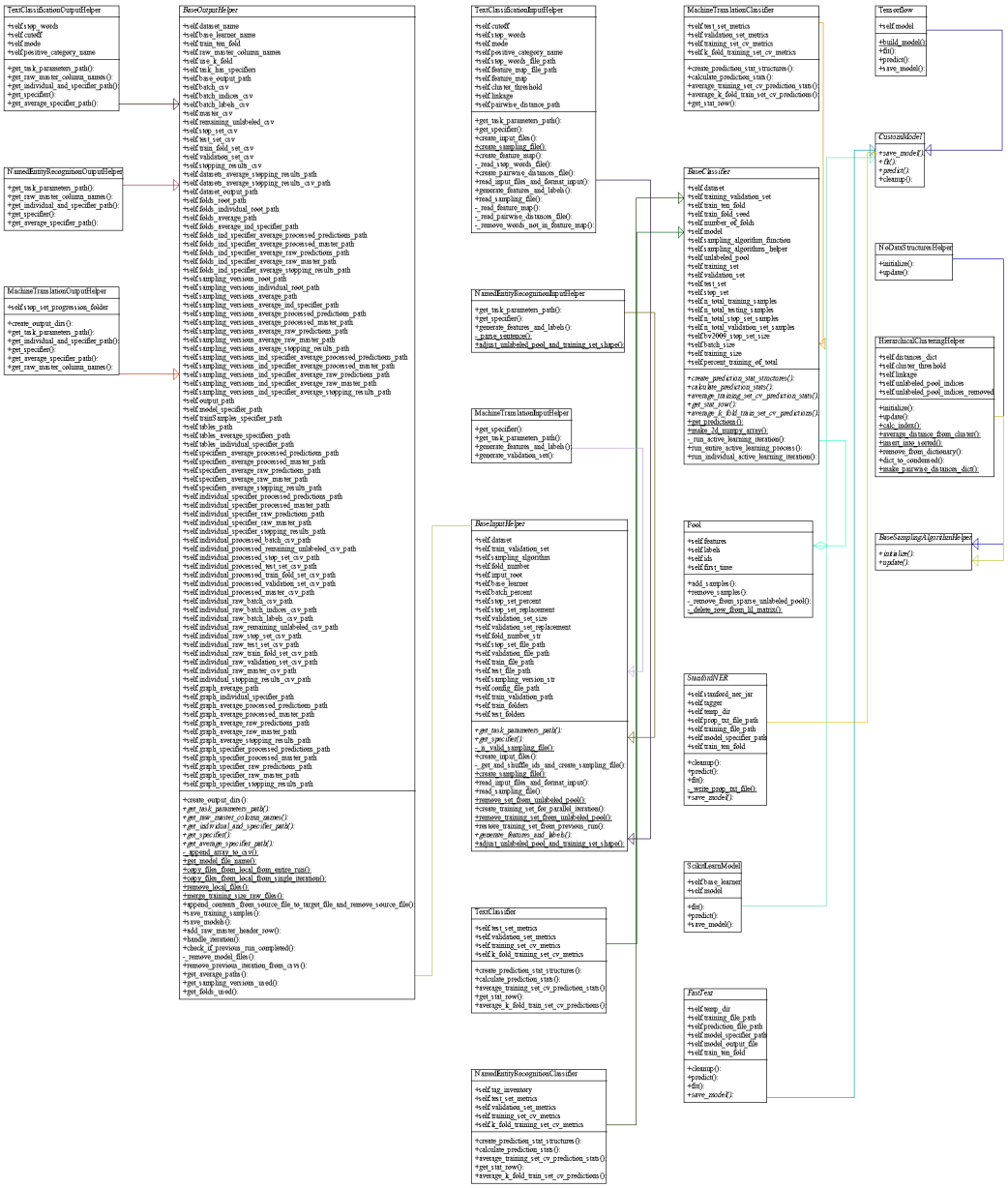
It is important to keep in mind that different types of relationships in a UML diagram are represented by different types of lines. This means that I had to devise procedures to draw dashed and solid lines that either have a triangle, diamond or no arrow. I had to use a triangle to draw lines that represented inheritance while I would use filled and unfilled diamonds to illustrate composition and aggregation respectively. Besides lines, UML diagrams also consist of boxes that store information about each class. Boxes are used to store class names, instance / class variable names and method names.

Minor Challenges

As opposed to analyzing class relationships from a python file, drawing boxes and lines for a UML Diagram may not be very time consuming. However, drawing lines can be somewhat challenging. When drawing lines, one has to ensure that lines are not intersecting each other in a way that makes it difficult for a viewer of a UML diagram to interpret. In addition, one also has to check that lines do not intersect boxes representing a class's information (instance/class variables, method and class names).

Finalized UML Diagram

Figure 11 - A complete UML Diagram of the src codebase



Description - This is a UML Diagram of my research team's codebase.Although my work has been very impressive, room for improvements remain although I have little to no known errors in my implementation of an auto UML generator. As always, I will have room to add new features and remove any redundancies within my implementation. Redundancies within my implementation involve bounds checking or unused functions / code.