Text-to-Art Interpreter: Software Design Document

By Lawrence Dickey, Wallis Muraca, Adam Carlson

Revision History:

Date	Version	Description	Authors
11/05/17	1.0	Initial version of document	Lawrence Dickey, Wallis Muraca, Adam Carlson
11/17/17	2.0	Rewritten document according to feedback	Lawrence Dickey, Wallis Muraca, Adam Carlson

Version 2.0

Table of Contents:

List of Figures:

1. Overview

1.1. Scope

This document contains a design description for the Python Text-To-Art Interpreter. The software will use a parser to scan words within text documents in order to map the individual characters to artistic outputs, more specifically drawings and audio. We will create the drawings using the Turtle graphics module. A second version of the software, time-permitting, will include an audio representation of the writings making use of the Multimedia Services Platform in Python. All members of the development group will have complete access to make changes to the design as is needed.

1.2. Purpose

This document provides the design details of the Python Text-To-Art Interpreter application designed for the Software Engineering I (CS 397) course at Colby College. The guidelines described in this document should clearly explain the steps needed to construct the software.

1.3. Intended audience

The document should be written such that the students of CS 397, the professor, those that maintain the software, and the Python Text-To-Art Interpreter creators can clearly understand the design of the application.

1.4. Conformance

This SDD conforms to clauses 4 and 5 of the standard IEEE layout:

- We describe the required content and organization of the SDD.
- We define several viewpoints for use in producing SDD's.

2. Definitions

For the purposes of this application, we define the following terms:

- *Pen Stroke*: The drawing of a single line using the turtle object
- *GUI*: Graphical User Interface
- *OOD*: Object-oriented design
- *The Standard Words Per Page*: We define the standard words per page of a text document to be 300.
- *Turtle*: A python module providing turtle graphics primitives used for drawing simple lines.
- *Parser*: A parser is a module which can sift through large amounts of text, or more generally data, and return useful information for an application.

3. Conceptual model for software design descriptions

3.1 Software design in context

We plan on creating an object-oriented design for our application, in which every class is responsible for one, and only one, task. This design will help keep our code both modular and maintainable. In addition, an object-oriented approach allows future developers to extend the application without breaking its initial functionality. For example, someone might want to add an additional artistic interpretation besides drawing or audio but would not want to completely rework the fundamental design of the code.

3.2 Software design descriptions within the life cycle

3.2.1 Influences on SDD Preparation: We created this SDD with the requirements of our stakeholders in mind. These stakeholders include the developers (Adam Carlson, Lawrence Dickey, and Wallis Muraca) along with Professor Codabux and the various application testers. Please refer to our SRS document for more details on the specific requirements pertaining to this project..

3.2.2 Influences on Software Lifecycle Products: This application will rely heavily on the control of data between our text processing software and our final GUI. The GUI will allow users to upload a text document and hand off the data to our backend processing. Consequently, the backend software will scan the text and pick out a sample of characters that is representative of the whole document but still manageable to fully process. Using our mapping between characters and drawing motions or musical notes, such as ‘a’ => draw a 50px black line, we will be able to create a final artistic output for the frontend of the application to display. No external server will be necessary for this project because all of our code can run on the user’s computer upon downloading the software.

3.2.3 Design Verification and Design Role in Validation: We plan on unit testing our design throughout the development process. This testing will ensure that each class and all of its methods function correctly and processes information in a reasonable amount of time in order to ensure a quality user experience. In addition, it is very important that we separate the GUI logic from the data processing logic in order to ensure the reusability, maintainability, and extendability of our software. Finally, we plan to test our final code on different computers to validate that it works for a diverse range of computer processors and operating systems.

4. Design description information content

4.1. Introduction

In this segment of the document, information surrounding design description will be explained in terms of its description later within the document. Here the document will explain the SDD identification, design stakeholders and subsequent concerns, design views, design viewpoints, design elements, design overlays, design rationale, and design languages.

4.2. SDD identification

This is the first version of the SDD created for the Python Text-To-Art Interpreter. The issue date for this version is 10.6.2017. The document is subject to change as the software design needs to be changed. The supervisor for this project is Zadia Codabux. This document will utilize UML standards for explaining the overall design and its varying viewpoints.

4.3. Design stakeholders and their concerns

The stakeholders in this design are the 3 students designing the software, Adam Carlson, Wallis Muraca, and Lawrence Dickey.
Lawrence has concerns that the software interface is too simplified to represent a complex set of words, particularly so because of the need to have an engine to draw every single line/shape that will be created within the drawing.

4.4. Design views

This project will be implemented following design principles taught in CS 397 taught by Zadia Codabux at Colby College. Specifically, the project will adhere to elegant software design principles explained by Professor Dale Skrien as explained in his guest lecture for CS 397 on October 31st, 2017. Professor Skrien's lecture explained the principles of SOLID design, an acronym representing principles which Professor Skrien believes to aid in elegant design. To put it briefly, these principles emphasize high cohesion and low coupling in object-oriented design. Restricting limitations to the design can be found within the SRS document.

4.5 Design viewpoints

The expected behavior from the user actor within the system is shown by the context viewpoint. In addition, our composition viewpoint showcases a potential class structure of our design. Please see section 5 of this SDD for a more extensive list of viewpoints.

4.6 Design elements

4.6.1 Entities:

a. tKinter (4.6.1.1)
b. Turtle (4.6.1.2)

c. Multimedia Services Platform (4.6.1.3)
d. File I/O (4.6.1.4)

4.6.2 Attributes:

e. tKinter (4.6.1.1): This is a library in Python. We will be using it in order to create the main GUI for our users. This library was created by the developers of Python.
f. Turtle (4.6.1.2): This is a library in Python. We will use it in order to draw images upon our GUI to represent texts as art. This library was created by the developers of Python.
g. Multimedia Services Platform (4.6.1.3): This is an entire package in Python. We will use it in order to generate sound files as an alternate artistic representation of the text. This package was created by the developers of Python.
h. File I/O (4.6.1.4): This is a built-in module in Python. We will make use of it in order to read in files and extract representative characters from the text. This module was created by the developers of Python.

4.6.3 Relationships: Our software should separate the dependencies between our GUI and word processor in order to increase code reusability. However, an overarching controller will have to have references to both the GUI and processor in order to facilitate the transfer of information.

4.6.4 Constraints: Our main constraint is the processing power of the user's computer because it could negatively affect our program's runtime.

4.7 Design overlays

We have no design overlays to report at this time for our design views.

4.8 Design rationale

Each design decision was made in order to ensure maintainability and extendability of the software. In general, the OOD will allow programmers to clearly understand what each module or class is expected to do and how they each interact with other classes. This separation of logic allows for new features to be added to the project without the necessity to refactor the code.

4.9 Design languages

We are using the Unified Modeling Language (UML) in order to create our design diagrams shown in section 5.

5. Design viewpoints

5.1 Introduction

In this section we highlight the significant viewpoints associated with our software applications.

Many of these viewpoints utilize diagrams, following the UML standard, in order to clarify their designs.

5.2 Context viewpoint

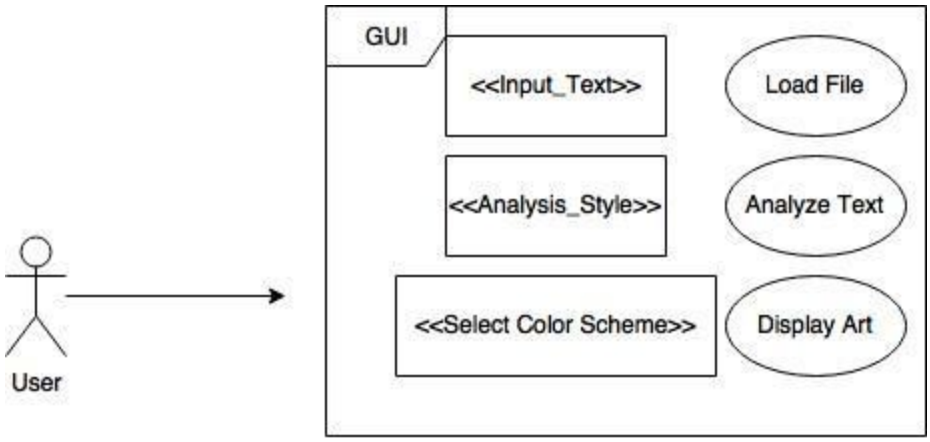


Figure 1: Context viewpoint

Load File: After filling out the input text, the user can load a file to be analyzed and displayed with the name written in input text.
Analyze Text: By pushing the analyze text button, and choosing a style of analysis, the user can analyze the text data taken in from load file.
Display Art: Display the final version of art in the GUI.

5.3 Composition viewpoint

5.3.1 Design elements

Design Entities: The program contains classes for the GUI, the text parser, the character interpreter, a module to control the turtle drawing output, and a data storage class.

Design Relationships: As a means of avoiding coupling, each module of the program operates on objects rather than having to have the whole system composed in a single class. While this may hinder efforts to mitigate representation exposure, this problem is less important in comparison to coupling avoidance. All methods of each module are public, making communication between parts easy without reliance on their existence.

5.3.2 Function attributes

GUI: Allows the user to interact with the software, displays artistic creations, allows the user to specify file names, and to clear previous drawings.

Character Interpreter: Finds artistic meaning relating to each character/word. Then develops an artistic way of representing this character. This includes a direction, color, width,

and length of a pen-stroke. These attributes are then compiled into a single list which can be accessed using the Interpret accessor method, which returns the single composite list.

Data storage: stores data so that it can be accessed later on in the program. This module aims to reduce the coupling by having data separated from any of the entities to prevent representation exposure.

Parser: iterates through text files parsing out each character. Stores each character in a list. The class has methods to open file and parse it. The parse method returns the list of characters created through the parsing process.

Turtle module: draws meaningful artistic output using the list of interpreted values created by the interpreter. Contains attributes field, the list of attributes given to each line the module draws.

5.4 Logical viewpoint

The classes to be implemented for the Python Text-To-Art Interpreter will be explained in this segment of the document.

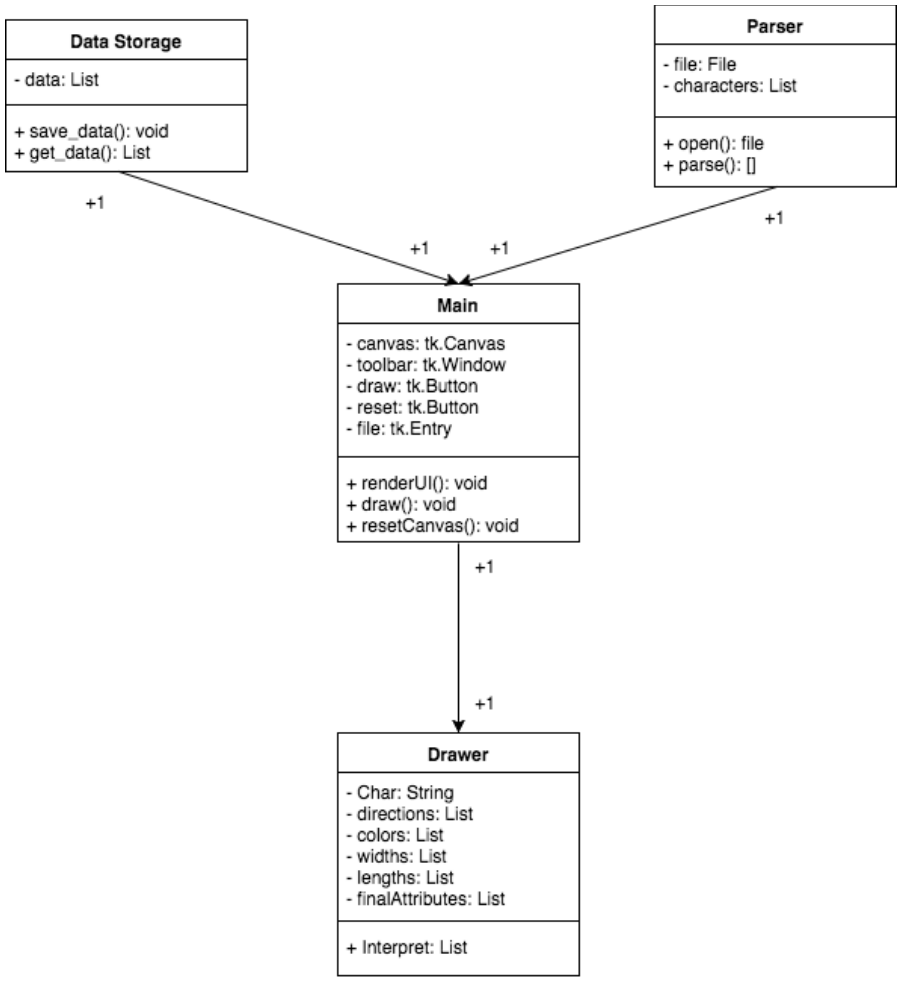


Figure 2: Class diagram

5.5 Dependency viewpoint

For this project, we will rely on a pipe-and-filter pattern. The data is transformed from one state to another: initial raw input, usable parsed data, and finally communications to the Turtle and GUI. Aside from initial user settings to begin the process, the user should not have to interact with the main rendering process.

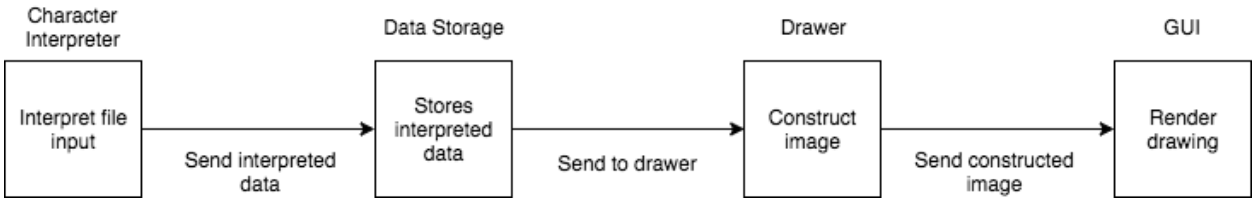


Figure 3: Diagram of system's pipe-and-filter architecture

5.6 Information viewpoint

Input Data	Interpretation Data
- raw_data: List - parsed_data: List	- styles: {}

Figure 4: Information viewpoint

5.7 Patterns use viewpoint

Some concepts that we wish to incorporate throughout our application are a quality UX experience, quick performance, reliability in program correctness. This can be seen throughout the variety of viewpoints in this section.

5.8 Interface viewpoint

An interface viewpoint is not applicable to our software because it is not interacting with any greater software or systems.

5.9 Structure viewpoint

A structure viewpoint is not applicable because our software does not use reuse elements.

5.10 Interaction viewpoint

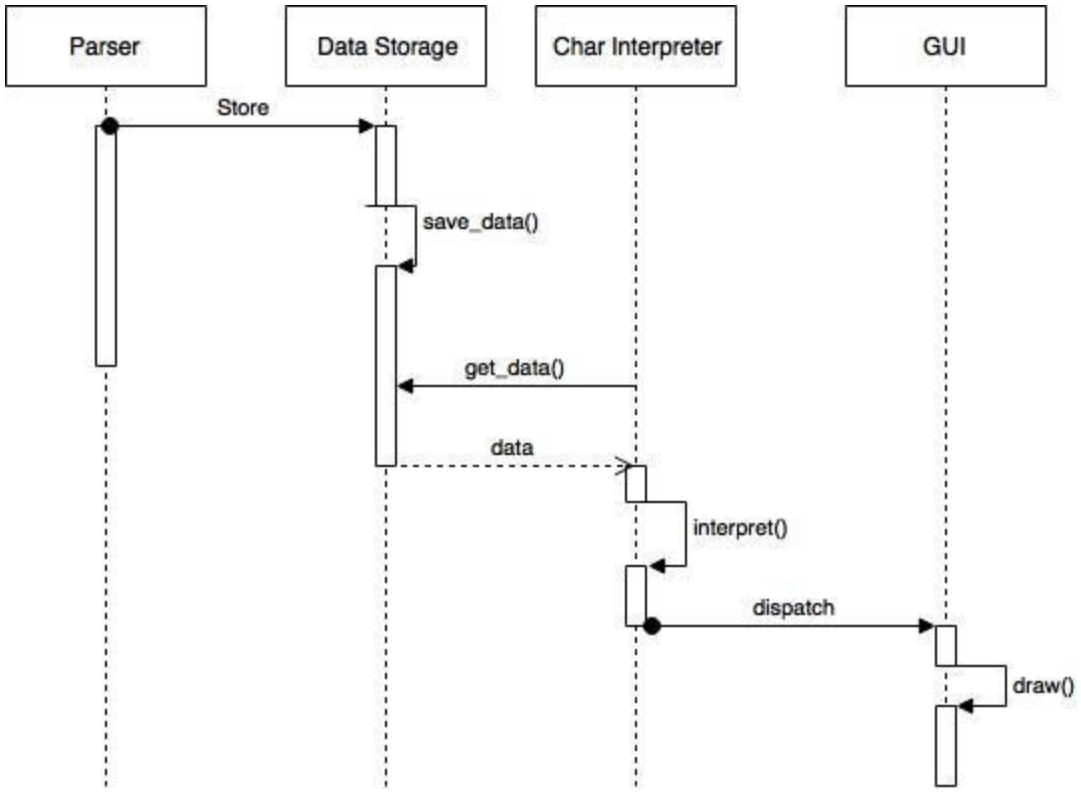


Figure 5: Sequence diagram for Use Case 1

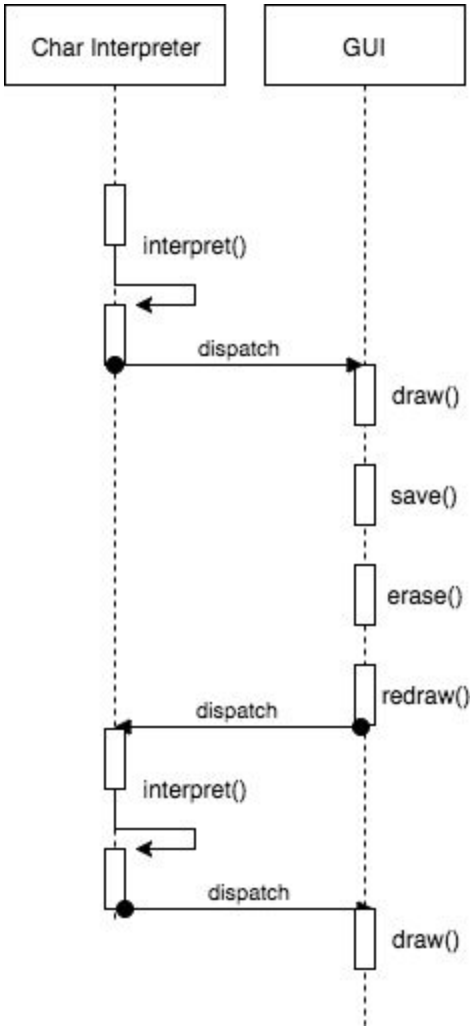


Figure 6: Sequence diagram for Use Case 3

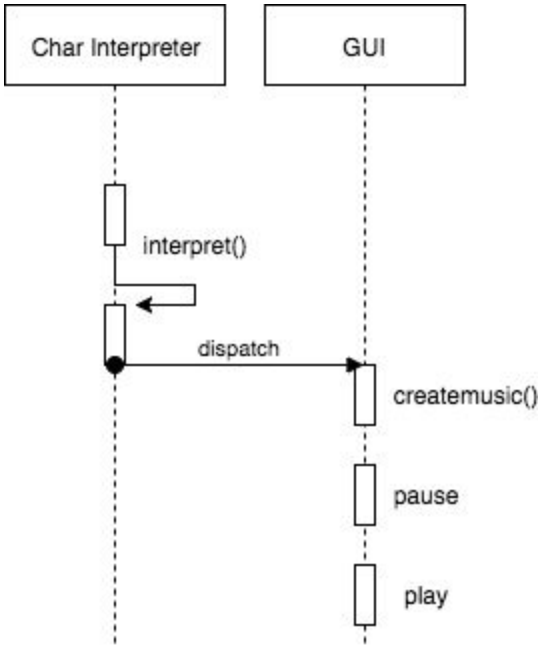


Figure 7: Sequence diagram for Use Case 4 (Optional Requirement)

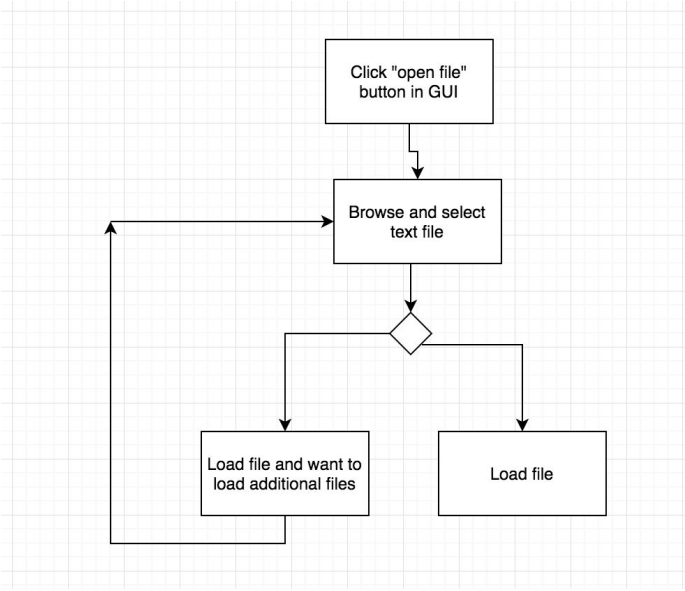


Figure 8: Activity Diagram for Use Case 2

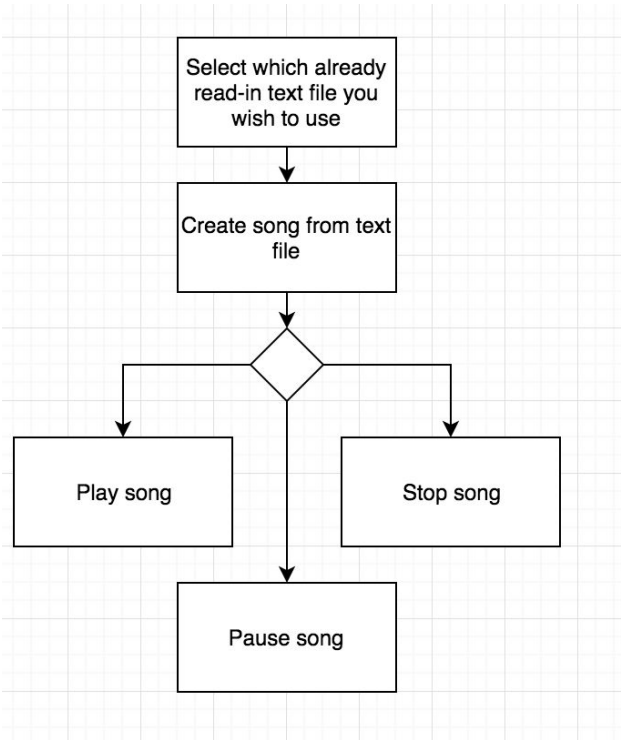


Figure 9: Activity Diagram for Use Case 4 (Optional Requirement)

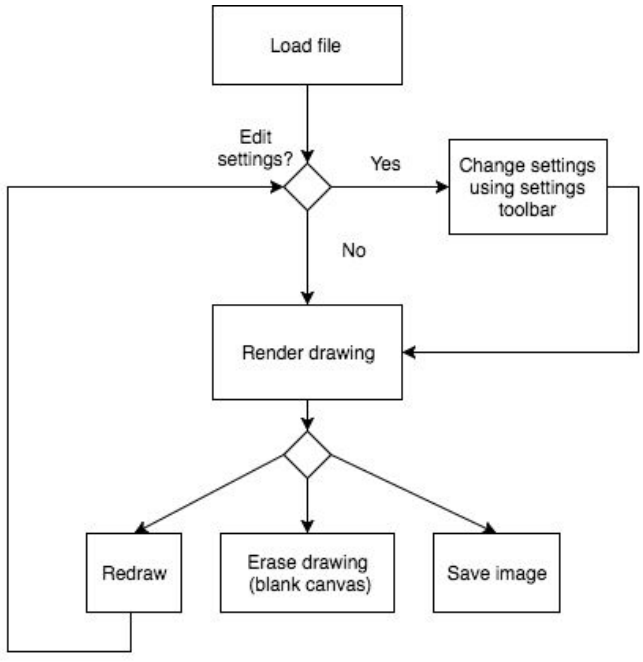


Figure 10: Activity Diagram for Use Case 3

5.11 State dynamics viewpoint

None of our components require a more in-depth look at composition because our high-level designs breakdown the tasks thoroughly enough for a SDD. Further details will arise as the design of the classes begins.

5.12 Algorithm viewpoint

We will use Python lists in order to store data from the read-in text file. Standard looping over the lists will apply to gather information but no further sorting or searching will be necessary.

5.13 Resource viewpoint

This viewpoint does not apply to our project since we are not heavily resource-dependent.

References:

Example SDD: http://user.ceng.metu.edu.tr/~e1746221/docs/SDDv1.1.pdf

Template: IEEE Standard for Information Technology—Systems Design—Software Design Descriptions

Table of Contents:

1. Overview

1.1. Scope

This document contains a design description for the Python Text-To-Art Interpreter. The software will use a parser to scan words within text documents in order to map the individual characters to artistic outputs, more specifically drawings and audio. We will create the drawings using the Turtle graphics module. A second version of the software, time-permitting, will include an audio representation of the writings making use of the Multimedia Services Platform in Python. All members of the development group will have complete access to make changes to the design as is needed.

1.2. Purpose

This document provides the design details of the Python Text-To-Art Interpreter application designed for the Software Engineering I (CS 397) course at Colby College. The guidelines described in this document should clearly explain the steps needed to construct the software.

1.3. Audience

The document should be written such that the students of CS 397, the professor, those that maintain the software, and the Python Text-To-Art Interpreter creators can clearly understand the design of the application.

1.4. Conformance

This SDD conforms to clauses 4 and 5 of the standard IEEE layout:

- We describe the required content and organization of the SDD.
- We define several viewpoints for use in producing SDD's.

2. Definitions

For the purposes of this application, we define the following terms:

- *Pen Stroke*: The drawing of a single line using the turtle object
- *GUI*: Graphical User Interface
- *OOD*: Object-oriented design
- *The Standard Words Per Page*: We define the standard words per page of a text document to be 300.
- *Turtle*: A python module providing turtle graphics primitives used for drawing simple lines.
- *Parser*: A parser is a module which can sift through large amounts of text, or more generally data, and return useful information for an application.

3. Conceptual model for software design descriptions

3.1 Software design in context

We plan on creating an object-oriented design for our application, in which every class is responsible for one, and only one, task. This design will help keep our code both modular and maintainable. In addition, an object-oriented approach allows future developers to extend the application without breaking its initial functionality. For example, someone might want to add an additional artistic interpretation besides drawing or audio but would not want to completely rework the fundamental design of the code.

3.2 Software design descriptions within the life cycle

1. **Influences on SDD Preparation:** We created this SDD with the requirements of our stakeholders in mind. These stakeholders include the developers (Adam Carlson, Lawrence Dickey, and Wallis Muraca) along with Professor Codabux and the various application testers. Please refer to our SRS document for more details on the specific requirements pertaining to this project..
2. **Influences on Software Lifecycle Products:** This application will rely heavily on the control of data between our text processing software and our final GUI. The GUI will allow users to upload a text document and hand off the data to our backend processing. Consequently, the backend software will scan the text and pick out a sample of characters that is representative of the whole document but still manageable to fully process. Using our mapping between characters and drawing motions or musical notes, such as ‘a’ => draw a 50px black line, we will be able to create a final artistic output for the frontend of the application to display. No external server will be necessary for this project because all of our code can run on the user’s computer upon downloading the software.
3. **Design Verification and Design Role in Validation:** We plan on unit testing our design throughout the development process. This testing will ensure that each class and all of its methods function correctly and processes information in a reasonable amount of time in

order to ensure a quality user experience. In addition, it is very important that we separate the GUI logic from the data processing logic in order to ensure the reusability, maintainability, and extendability of our software. Finally, we plan to test our final code on different computers to validate that it works for a diverse range of computer processors and operating systems.

4. Design description information content

4.1. Introduction

In this segment of the document, information surrounding design description will be explained in terms of its description later within the document. Here the document will explain the SDD identification, design stakeholders and subsequent concerns, design views, design viewpoints, design elements, design overlays, design rationale, and design languages.

4.2. SDD identification

This is the first version of the SDD created for the Python Text-To-Art Interpreter. The issue date for this version is 10.6.2017. The document is subject to change as the software design needs to be changed. The supervisor for this project is Zadia Codabux. This document will utilize UML standards for explaining the overall design and its varying viewpoints.

4.3. Design stakeholders and their concerns

The stakeholders in this design are the 3 students designing the software, Adam Carlson, Wallis Muraca, and Lawrence Dickey.

Lawrence has concerns that the software interface is too simplified to represent a complex set of words, particularly so because of the need to have an engine to draw every single line/shape that will be created within the drawing.

4.4. Design views

This project will be implemented following design principles taught in CS 397 taught by Zadia Codabux at Colby College. Specifically, the project will adhere to elegant software design principles explained by Professor Dale Skrien as explained in his guest lecture for CS 397 on October 31st, 2017. Professor Skrien's lecture explained the principles of SOLID design, an acronym representing principles which Professor Skrien believes to aid in elegant design. To put it briefly, these principles emphasize high cohesion and low coupling in object-oriented design. Restricting limitations to the design can be found within the SRS document.

4.5 Design viewpoints

The expected behavior from the user actor within the system is shown by the context viewpoint. In addition, our composition viewpoint showcases a potential class structure of our design. Please see section 5 of this SDD for a more extensive list of viewpoints.

4.6 Design elements

Entities:

i. tKinter (4.6.1.1)
j. Turtle (4.6.1.2)
k. Multimedia Services Platform (4.6.1.3)
l. File I/O (4.6.1.4)

Attributes:

m. tKinter (4.6.1.1): This is a library in Python. We will be using it in order to create the main GUI for our users. This library was created by the developers of Python.
n. Turtle (4.6.1.2): This is a library in Python. We will use it in order to draw images upon our GUI to represent texts as art. This library was created by the developers of Python.
o. Multimedia Services Platform (4.6.1.3): This is an entire package in Python. We will use it in order to generate sound files as an alternate artistic representation of the text. This package was created by the developers of Python.
p. File I/O (4.6.1.4): This is a built-in module in Python. We will make use of it in order to read in files and extract representative characters from the text. This module was created by the developers of Python.

Relationships: Our software should separate the dependencies between our GUI and word processor in order to increase code reusability. However, an overarching controller will have to have references to both the GUI and processor in order to facilitate the transfer of information.
Constraints: Our main constraint is the processing power of the user's computer because it could negatively affect our program's runtime.

4.7 Design overlays

We have no design overlays to report at this time for our design views.

4.8 Design rationale

Each design decision was made in order to ensure maintainability and extendability of the software. In general, the OOD will allow programmers to clearly understand what each module or class is expected to do and how they each interact with other classes. This separation of logic allows for new features to be added to the project without the necessity to refactor the code.

4.9 Design languages

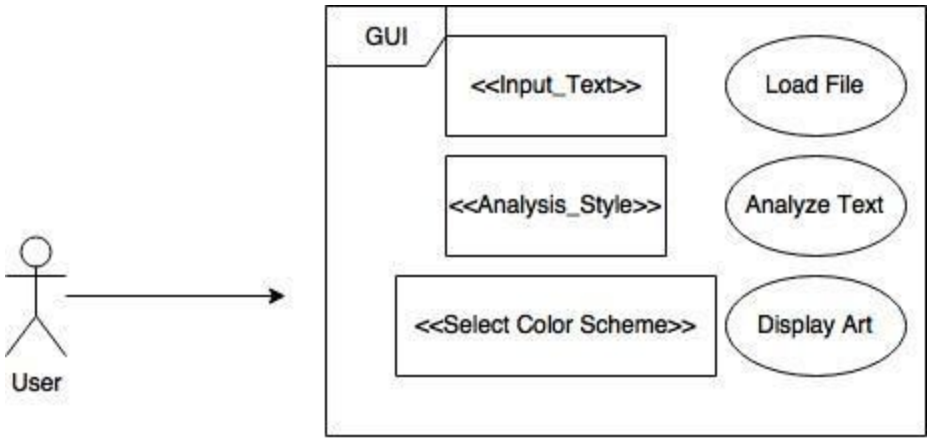
We are using the Unified Modeling Language (UML) in order to create our design diagrams shown in section 5.

5. Design viewpoints

5.1 Introduction

In this section we highlight the significant viewpoints associated with our software applications. Many of these viewpoints utilize diagrams, following the UML standard, in order to clarify their designs.

5.2 Context viewpoint



Load File: After filling out the input text, the user can load a file to be analyzed and displayed with the name written in input text.

Analyze Text: By pushing the analyze text button, and choosing a style of analysis, the user can analyze the text data taken in from load file.

Display Art: Display the final version of art in the GUI.

5.3 Composition viewpoint

5.3.1 Design elements

Design Entities: The program contains classes for the GUI, the text parser, the character interpreter, a module to control the turtle drawing output, and a data storage class.

Design Relationships: As a means of avoiding coupling, each module of the program operates on objects rather than having to have the whole system composed in a single class. While this may hinder efforts to mitigate representation exposure, this problem is less important in comparison to coupling avoidance. All methods of each module are public, making communication between parts easy without reliance on their existence.

5.3.2 Function attributes

GUI: Allows the user to interact with the software, displays artistic creations, allows the user to specify file names, and to clear previous drawings.

Character Interpreter: Finds artistic meaning relating to each character/word. Then develops an artistic way of representing this character. This includes a direction, color, width, and length of a pen-stroke. These attributes are then compiled into a single list which can be accessed using the Interpret accessor method, which returns the single composite list.

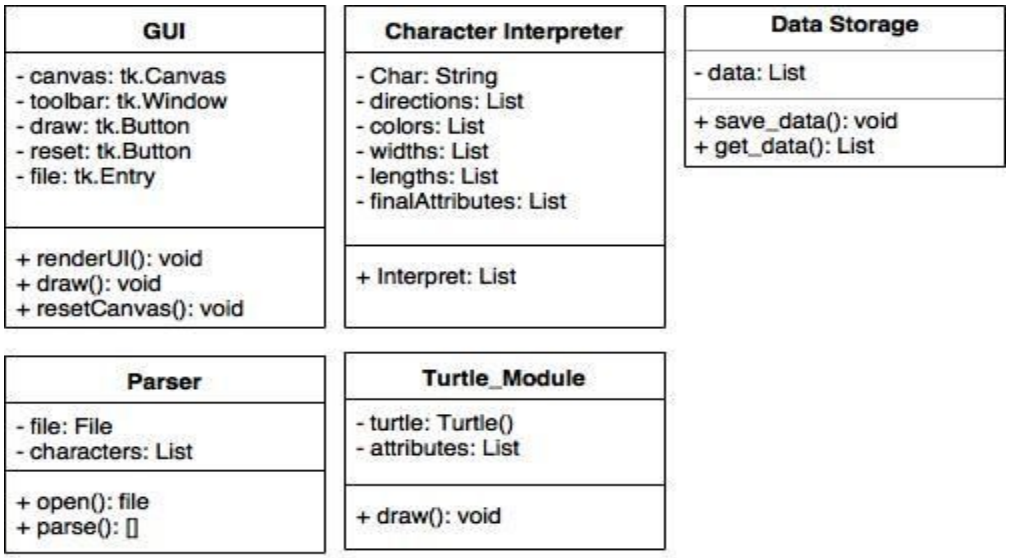
Data storage: stores data so that it can be accessed later on in the program. This module aims to reduce the coupling by having data separated from any of the entities to prevent representation exposure.

Parser: iterates through text files parsing out each character. Stores each character in a list. The class has methods to open file and parse it. The parse method returns the list of characters created through the parsing process.

Turtle module: draws meaningful artistic output using the list of interpreted values created by the interpreter. Contains attributes field, the list of attributes given to each line the module draws.

5.4 Logical viewpoint

The classes to be implemented for the Python Text-To-Art Interpreter will be explained in this segment of the document.



5.5 Dependency viewpoint

We will rely on a pipe-and-filter pattern. The data is transformed from one state to another: initial raw input, usable parsed data, and finally communications to the Turtle and GUI. Because pipe-and-filter is not very friendly for user interaction, we will likely change this to a model-view-controller pattern later.

5.6 Information viewpoint

Input Data	Interpretation Data
- raw_data: List - parsed_data: List	- styles: {}

5.7 Patterns use viewpoint

Some concepts that we wish to incorporate throughout our application are a quality UX experience, quick performance, reliability in program correctness. This can be seen throughout the variety of viewpoints in this section.

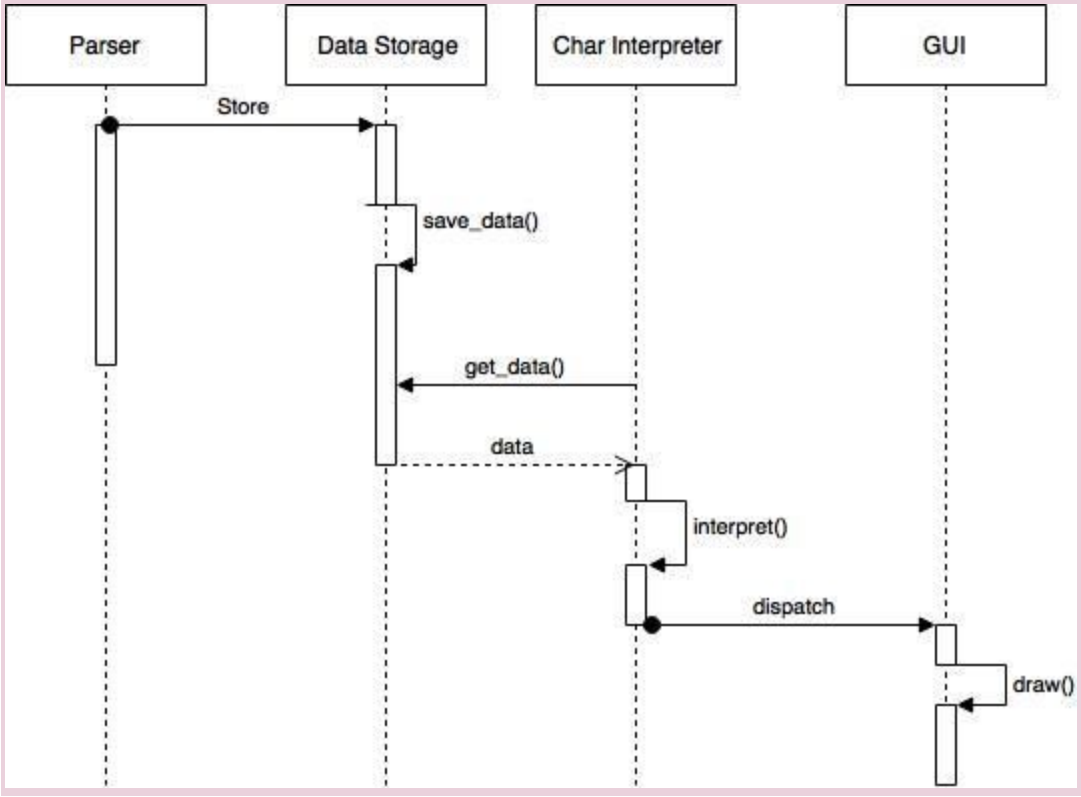
5.8 Interface viewpoint

An interface viewpoint is not applicable to our software because it is not interacting with any greater software or systems.

5.9 Structure viewpoint

A structure viewpoint is not applicable because our software does not use reuse elements.

5.10 Interaction viewpoint



5.11 State dynamics viewpoint

None of our components require a more in-depth look at composition because our high-level designs breakdown the tasks thoroughly enough for a SDD. Further details will arise as the design of the classes begins.

5.12 Algorithm viewpoint

We will use Python lists in order to store data from the read-in text file. Standard looping over the lists will apply to gather information but no further sorting or searching will be necessary.

5.13 Resource viewpoint

This viewpoint does not apply to our project since we are not heavily resource-dependent.

References:

Example SDD: http://user.ceng.metu.edu.tr/~e1746221/docs/SDDv1.1.pdf

Template: IEEE Standard for Information Technology—Systems Design—Software Design Descriptions