Octo-Docs

Software Design

Dr. James Palmer

Garrison Smith
Peter Huettl
Kristopher Moore
Brian Saganey

2/9/17

version 1

Table of Contents

1. Introduction

Octo-Docs is a team that was formed with a goal to improve how software development groups create, edit, and interact with comments in their projects. The members of team Octo-Docs are Garrison Smith, Peter Huettl, Kristopher Moore, and Brian Saganey and we are working on creating a new documentation management system called CrossDoc. The project is sponsored by Dr. James Palmer, who first proposed that commenting is in need of an improvement.

Software development teams currently face the problem that their documentation is highly dependent on their project's codebase. This dependence results in documentation that is hard to view or edit externally, particularly by non-tech savvy individuals. The globalization of software development groups means that not all developers working on a particular codebase speak the same language, and when this is coupled with the fact that software development is one of the biggest modern industries, it becomes apparent why this is an issue. This is why many modern teams have groups dedicated to the localization of the work environment. Employees such as this are often not familiar with current commenting practices and have a hard time combing through the codebase to find the language to change. The longer this comment management time takes, the more time and money a team is using not developing their product.

Octo-Docs and Dr. Palmer aim to fix this with CrossDoc, a commenting system that connects external comment stores to a codebase. By simply referencing external comments, this new comment system will provide a great deal of flexibility and improvability to the standard commenting system. Improvements such as distinct comment categories, an adapting comment history, and user-specific comment modules will not only solve the initial commenting problem but also provide an adaptable canvas to solve future documentation issues. Dr. Palmer, as the Associate Director of Undergraduate Programs at NAU, is particularly interested in this product for its implications for teams and even individuals working on their own projects. As such, this technology works well in an educational environment, and the categorization of comments can be used to help beginners and experienced developers alike.

In this document, we will be providing technical insight into the architecture of our CrossDoc implementation. We will outline exactly what components and modules should be in place to create a tool such as CrossDoc and elaborate on how each of these elements will interact. These components and connections will serve as a blueprint to describing how CrossDoc specifically tackles the collection of functional, non-functional, and user-level requirements necessary to create the product.

The primary requirements for CrossDoc, as outlined in our Requirements Specification Document, are as follows: implement core functional data storage capabilities, create an intuitive and easily adoptable system, expand the functional capability of comments and comment storage, support the scalability of a team environment, and incorporate powerful tools for experienced users. These functional requirements directly target the core aspects of the problem that CrossDoc was created to solve. CrossDoc will also prioritize maintainability, performance, readability, security, and usability throughout all developmental efforts. These non-functional requirements create guidelines and standards for all aspects of the product. The project also accommodates the environmental requirements of needing to support specific text editors (Sublime, Atom, Emacs, and Vim), and needing to closely integrate with Git. This integration will increase the usability and collaborative nature of the product.

The modules present in the project architecture were specifically created to create a product that is capable of meeting the requirements outlined. Through modularity, adaptability, and security, the components and connections presenting in this document will provide a blueprint for a product that can address all requirements.

2. Implementation Overview

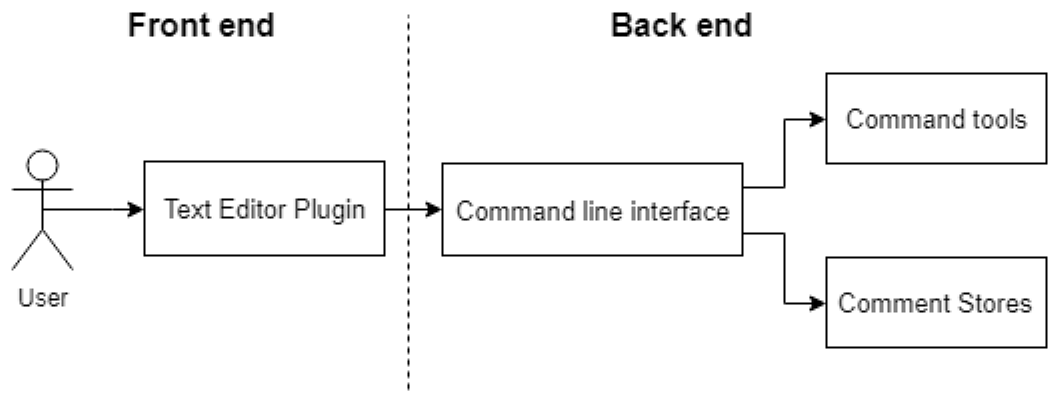
We will further develop the comment system developed by Dr. Palmer and Nakai McAddis. The current system is structured like a ball up system where the comments are stored externally and are fetched when needed. The issue is when different teams scattered across the world and from different languages and cultures embed their comments in the program, the program gets messy. This new developed external storage of comments will make it easier to store comments, help structure comments made by non-developers, store comments created by different language, and identify unique development issues.

We will use the PIP package management system and use the Python, JavaScript, Elisp, and Vimscript to expand on the CrossDoc program used in the current comment system. The plugin for Sublime will be encoded with Python; Atom's plugin will be Javascript; Emacs's plugin will be Elisp, and Vim's plugin will be VimScript. The Python library will be used to fetch and update remote data to create and use local and remote comment storage system. Git hooks and Commit hashes will be used to integrate with Git closely.

The command line program portion of CrossDoc will be written exclusively in Python, and delivered using PIP. This command line program will be used as a "back-end" service to provide consistent operations across the text editor plugins listed above. Each of the text editor plugin languages provides a method to interact with the shell and to execute shell scripts. This is the method we will use to interface with our "back-end" program from the "front-end" text editor plugins.

3. Architectural Overview

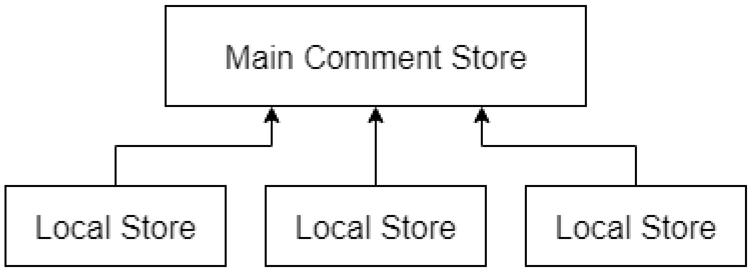
The general idea for the architecture of this project is to have a user access CrossDoc through a plug-in within their text editor. Then the user will be able to use the plug-in to access the command line interface. This is where they will have access to all the command tools and the comment stores within their project. The user accessing CrossDoc is out way of describing the front-end of our project and the command line interface where the user can access the command tools and the comment stores will be the back-end of this project. *Figure 1* diagrams the broad picture of what the architecture will look like at the highest level, however, within each box is a subset of tools and interfaces that the user can use while using our project.



(Figure 1, interaction architecture diagram)

The first interaction the user will have is to download CrossDoc through a text editor plug-in. The goal is to make sure that whatever plug-in the user is using it should be a quick and easy process for them to download CrossDoc within the text editor. All of this is more of how the front-end of the project will work and once the user has access to CrossDoc, they will be able to use all the functions present in the command line program. Once CrossDoc is downloaded on the text editor, the user will then have access to the back-end of CrossDoc. The purpose of the back-end is to have centralized command line interface that branches off onto the tools and functions that the user will be using.

Once the user has access to the command line interface, they will initialize a project and begin using the CrossDoc repository to create, read, update, and delete comments within their project. This repository will be a general repository where all their creations will go to. In addition, they will have a localized file that they can also update that will be shared with the general repository. *Figure 2* demonstrates this connectivity.



(Figure 2, repository connection to stores)

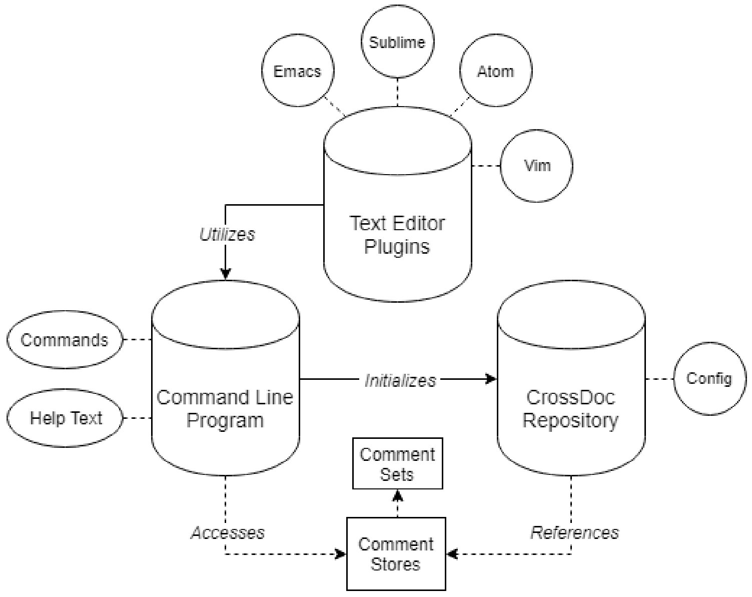
Each comment store will also have a set with it too that can categorize where to find what comment for the project. This will allow users to have an organized method so that they are not struggling to find a comment within their project. The command line tools are essentials for the users to have so that they can access these comment stores and with their command line interface and manipulate them into anyone they might need to for the project. However, the command line tools are more of a modular/interface description which will be discussed later. The main focus of the architecture of CrossDoc is to make sure that our project can stay up to date with the text editors that come out as well as making sure that the idea behind CrossDoc remains relevant.

When a user uses the plug-in for CrossDoc on their text editor the goal is to make sure that it spans across all available text editors and can easily be updated as time goes on. This will allow CrossDoc to be used in future projects as well as become adaptable to the changes that happen through programming in computer science. Furthermore, making sure that the testing of CrossDoc and its functionalities are easily testable and maintainable for future goals within CrossDoc. In general, this is the overview of what the architecture for CrossDoc is and how it can continue working as the future comes along.

4. Module and Interface Descriptions

In this section, we will provide an in-depth analysis of the CrossDoc architecture. For each module in the design, we will provide a brief description of its purpose, create a diagram that outlines how this module fits into the larger whole, and describe exactly how and when this module is interfaced through detailed interface descriptions. These technical descriptions will serve as the blueprint to CrossDoc and establish how such a system should be formed.

The specific high-level architectural modules we will focus on are the *CrossDoc repository*, the *command line program*, and the *text editor plugins*. These modules each have sub-modules and aspects that need to be implemented to properly implement the system. We will discuss these sub-modules in detail in the breakdown of each module. Each of these modules is connected either functionally or conceptually through the operations that link them. A *CrossDoc repository* is initialized using the *command line program*, but the modules are each conceptually unique. In fact, all CrossDoc operations can or will use a mixture of each of these modules. A user may install and only interact with CrossDoc through a *text-editor plugin*, whose functionality is implemented modularly in the *command line program*.



(Figure 3, architecture module connectivity)

The interconnectivity of the CrossDoc architecture is demonstrated in *Figure 3*. Each high-level module is surrounded by components that make up the module's implementation. The connection between these main modules is unidirectional in the direction of lower level functional abstractions. This architecture synchronizes well with the use case of CrossDoc as a program. Users will often interact with the *text-editor plugins* which serve as the user interface to the *command line program*, which in turn, manipulates the state of the *CrossDoc repository*. In short, the CrossDoc system can be used and manipulated at each of these high-level modules, but the more abstracted the interaction becomes, the more architectural modules will be utilized. The user could theoretically manipulate all comment storage state by simply editing the text and JSON files that make up the CrossDoc repository, but this architecture creates the most accessible and usable developmental flow.

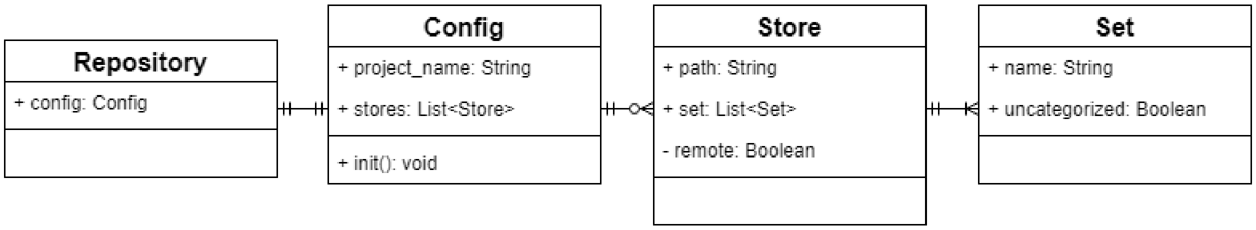
4.1. CrossDoc Repository

The repository module is the conceptual element of the architecture that represents a project-specific instance of the CrossDoc storage system. Similar to Git, a CrossDoc repository can be initialized within a particular project. Within this module of the architecture, there are submodules that comprise the functionality. Submodules such as remote and local comment storage, comment sets, and a CrossDoc configuration file that stores persistent information for a particular repository (see *Figure 4*).

This architectural module can be seen as the lowest level of interaction with the system. A user could manually create comment stores, sets, and tweak their configuration file to support the CrossDoc system, but the repository will primarily be interfaced with the *text-editor plugins*. In terms of the larger architectural system, the CrossDoc repository will be initialized by the *CrossDoc command line program*, and the repository directly holds the references and connections to both the local and remote file stores.

4.1.1. UML Diagram

Figure 4 demonstrates the connectivity of the elements in the CrossDoc repository module. Although the Repository object only includes a reference to the Config object, the distinction is important as the repository includes the concept of a system instance. Although a single user will only interact with a single repository, this object should be versioned in a project, and as such, is distributed to all developers who have pulled a CrossDoc project's source.



(Figure 4, CrossDoc repository component connectivity

4.1.2. Interface Description

This section will contain a detailed analysis of the component interfaces present in the CrossDoc repository module. Because the Repository object is not intended to be interfaced, we will first discuss the Config object.

Config Component

The Config is a file stored within a particular development project that has been initialized as a CrossDoc project. The Config holds the following persistent data:

- **project_name** - The internal name for the initialized CrossDoc repository
- **stores** - A list of registered comment storage paths
  - This includes a flag denoting whether or not the source is *local* or *remote*

This information *can* be manually edited through the user's preferred text-editor. The primary method of interfacing with this object will be through either the *command line program* or the *text-editor plugins*. This object is crucial to the CrossDoc architecture and is what makes such a system feasible. Without it, or an equivalent persistent storage system, users would need to re-enter this information with every relevant operation. This component also enables the possibility of a distributed system for team environments.

Teams can, and likely will, have a single collection of comment stores in which the documentation for all projects can be centralized and easily maintained. Implementing a Config object as we have enables this because the configuration file can be versioned alongside the source of a teams program. This will ensure that every individual working on a project will reference the centralized repository of documentation, with the possibility of local extensions.

Store Component

The next object in the CrossDoc repository is the Store object. This object represents the "database" from which comments can be created to, and pulled from. Although this object is functionally a database, in practice, there are several reasons why it should not explicitly be one. Firstly, creating and utilizing a single consistent database format across local and remote instances is a difficult task and obfuscates the storage method for non-tech individuals working in a team. Secondly, creating a usable and interactive interface for these comments in their destination would be an unnecessary challenge to developing the system. Lastly, a database is not as flexible across individual user operating systems.

To tackle all of these issues, our Stores are implemented as simple directories. The sets within the comment store are simply text-files within the directory. This format is flexible, intuitive for non-tech team members, and operating system agnostic. The one extra variable required to facilitate this is a flag denoting whether the store is remote or local. This tells CrossDoc how to read and create comments to the store. Besides this, the store only intrinsically stores its name (denoted by the directory name) and the references to its sets (the files present in the directory).

Set Component

The last component in the CrossDoc repository module is the Set. A set is a named collection of comments that all correlate to the theme of the set. For example, within a single comment store, there might be a set for “New Developer Notes,” “Design Insight,” or “TODO Comments.” These can then be toggled through by the user of the application.

The main variable in a set is the theme of the set. We have chosen to denote the theme of the set by its file name. The set file is stored in the comment store directory, with a single flag denoting whether or not the set is for uncategorized comments. Not all CrossDoc comments need to be stored in a comment set, and those that are not, are categorized as having “No Set.” The implementation of this object is very important as it largely defines the functionality of CrossDoc, and it is a primary feature.

4.2. Command Line Program

In our plugin, we want to make sure that the user is able to use all the commands that they need to use in order to fulfill a requirement within their project. Most of these commands are used within the command line interface where the user brings up a command prompt and will be able to use all the basic functionalities of CRUD within the command prompt.

4.2.1. Interface Description

Since CrossDoc is a command line tool and it allows users to manipulate comments for their software, we want to make sure that the general idea behind CRUD is established. In regards to CRUD, the general idea is to make sure a user can create, read, update, and delete any information that is run through the command prompt. These basic commands will allow for future testing to be much easier as well as ensuring that the commands can be changed and expanded as the project progresses in its lifetime.

Generating Anchors *(generate-anchor)*

The first command that we had to make for our program was a *generate-anchor* function. This function is used to distinguish between comments that were created as well as making sure that the comments are separated by an anchor symbol of our choosing. This symbol will be shown as a unique character, followed by a unique identification number. The plan is to use the git to generate a commit identification number followed by some pound signs to hide the user who made the comment.

When a user calls the function *generate-anchor* it will generate the anchor for the comment inside the local file where the user can then make comments under that anchor or manipulate that anchor for their project. The user will be able to call *generate-anchor* using shorthand notations such as ga, g, or just *generate-anchor*. This function was not in the general aspect of CRUD but it did need to be included into the command lines so that the user could create their own anchors for the project.

Creating Comments *(create-comment)*

The next command starts off the idea of CRUD by implementing a *create-comment* command. This command will allow a user to create a comment within their comment storage file and also make a comment in the repository comment storage file as well. The idea behind *create-comment* is to make sure that a user is able to generally create comments for their project. The main function is to call *create-comment* in the command prompt and this will then create a comment and attach the comment to a new anchor. When that comment is then created, the user will be able to see the new anchor it generated within the repository file and then the user will be able to use that anchor to manipulate the comment in later command line tools or just later on in their project. When the user calls *create-comment* the user can also use shorthand for this function as well so they can call the function by entering cc, c or *create-comment*.

The *create-comment* function also has some optional parameters that can be given as well. The first one is the store, which is allowing the user to create a comment within a comment storage place. The user can pick a certain local file on their computer or they can choose to create the comment in the repository file. The next parameter that *create-comment* takes in is the set parameter. When *create-comment* gets called it can add a set to the parameters as well. A set is a place in which the user can but the comment in a set location within the comment store so that it is can stay organized within the store for other users. So when *create-comment* is called in the end it can either be called with no parameters, called with the comment store location, called with the set location within the store, or both with the comment store and with the set.

Reading Comments *(fetch-comment)*

As part of CRUD the project should also be able to read comments within the comment stores, which leads us into the read part of CRUD. When the user calls *fetch-comment* it will essentially take a comment that is stored within a comment store and by using the parameters of an anchor. The general scope is that a user will give the command prompt an anchor after *fetch-comment* and the program will search through the comment stores until it finds that anchor and then it will output that anchor with the comment to the user.

The user will also be allowed to give an optional parameter to the command prompt which is what comment store the search should be looking at. When the user gives the anchor with the comment store then it can be easier for the search to find the comment. In regards to CRUD the read command is fairly straightforward because all that *fetch-comment* does is take in an anchor and output the value at that anchor. If there is no such anchor that exists then the output to the user will let the user know that the anchor is incorrect.

Updating Comment *(update-comment)*

The third aspect of CRUD is an update functionality, which allows a user to update information through their comment stores. The *update-comment* function allows the user to find a comment by stating what anchor the comment is at and then allowing the user to update that comment with a new comment. For example, the user will call update-comment which takes in two required parameters, which are the anchor that the comment is stored at as well as a block of text so that the comment gets updated with this new comment.

The *update-comment* functionality works similarly to the *fetch-comment* function in that it runs through all the comment stores to find the comment that the user is looking for and once it finds it, then it will update the current comment with the given text and return back the newly updated comment. An optional parameter that *update-comment* has, similar to *fetch-comment*, is that it can take in a comment store location so that it is easier to find the comment for the program and the user can get to the comment and update it faster. The general idea for this function is to make sure that a user can not only create comments but update them as well because many times while working on a project the comments can start to change based on the changes of code and we do not want to have comments that are not being used efficiently for the project.

Deleting Comments *(delete-comment)*

For the last main plan of implementation for CRUD, we want to make sure that we can delete a comment that gets added to a comment store. So we decided that we wanted to create a function that deletes a comment. While we were implementing this functions we the idea was that we could implement the same ideas from *fetch-comment* and *update-comment* by using the same algorithm in finding the comment and updating it.

For *delete-comment*, a user will take in an anchor from where a comment is located and that user will be able to delete the comment at that location in the comment store. The other parameters that the user can use are that they can give the command prompt the store location for where the comment is at so that a user can delete the exact comment either from the main repository of the store or the local file.

The *delete-comment* function will also have shorthand like the rest of the functions, so when the user calls *delete-comment* they can give the command prompt either d, dc, or *delete-comment*. This shorthand notation allows for experienced programmers to have an easy way to access the functions of CrossDoc in a quick and easy manner. The functionality of *delete-comment* works really well because of the *fetch-comment*, essentially when a user calls *delete-comment*, it will run through the same *fetch-comment* code and then once it receives the comment, it will then delete the comment as well as the anchor that goes with that comment.

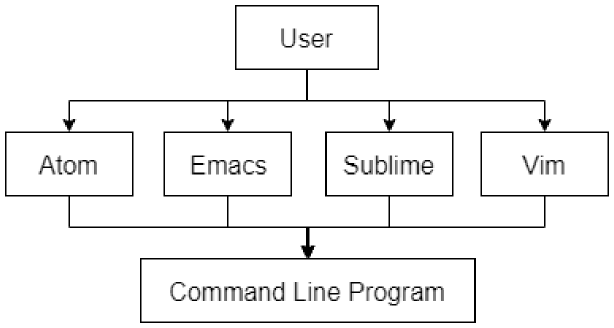
In conclusion, all these functions allow us to set the tone for the basic needs of users when using our project and it will allow us to expand and create more functions that can utilize these functions to the fullest potential.

4.3. Text Editor Plugins

For CrossDoc to be easily integrated into a programmer's professional work, it must be compatible with the major programming environments. The first version of CrossDoc will ship with unity between the Command-Line Parser which serves as the heart of CrossDoc functionality, and four major text editors: Atom, Emacs, Sublime, and Vim. The text editor plugins act as a bridge between the User and CL-Parser. They provide editor tools and functions for the user to document with CrossDoc easily within their environment. Although these editors may server the same sub-system role, due to their implementation differences and APIs each Text Editor plugin is its own system that must be developed, integrated, and tested independently.

4.3.1. UML Diagram

Figure 5 demonstrates the objects present in this architectural module. Users interact with a selection of text editors that we have created plugins to support. This text editor plugin tool directly communicates with the command line tool to fetch and utilize the CrossDoc functionalities. An in-depth analysis of the text editor objects is below.



(Figure 5, user-plugin interaction flow)

4.3.2. Interface Description

This section will contain a detailed analysis of the component interfaces present in the Text Editor Plugin module. Specifically, we will focus on the group of supported text editors and how they will integrate into the CrossDoc system. An understanding of these editor APIs is critically in implementing such a system.

Atom Text Editor

The Atom editing environment is touted as a hackable cross-platform editing tool. Within users can customize Atom to a higher degree than most editors and being built upon the Electron programming language can take advantage of CSS, HTML, JavaScript, and Node.js integrations. Moreover, the editor is open-source allowing for easier modification if circumstances arise. Atom also provides a developer API equipped with especially useful functionality for the CrossDoc project within their text editor functions. The CrossDoc program will be leveraging this section of API along with custom tools to provide a fully integrated plugin.

Emacs Text Editor

The GNU Emacs text editor is created as an Emacs Lisp interpreter that has, through extensions, been modified to support text editing. As such the editor is high extendable, customizable, and has self-documentation functionality. Working within Emacs will require all CrossDoc related functionality to be written as extensions of the editor in the Lisp programming language.

Sublime Text Editor

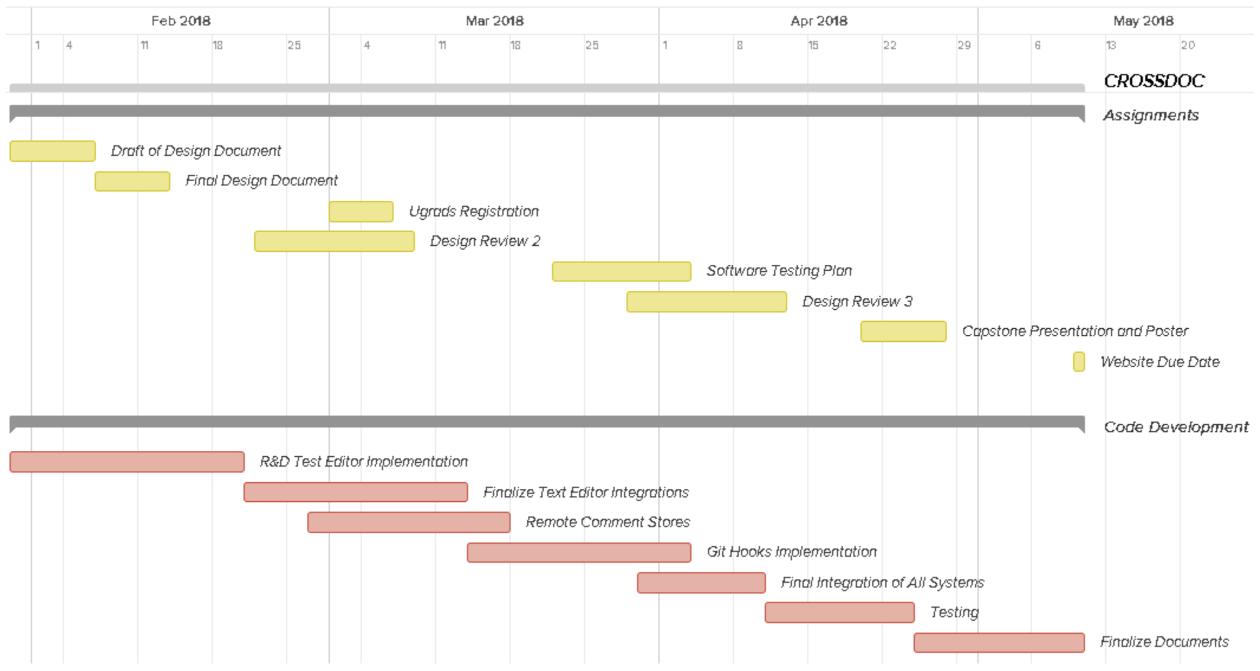
The Sublime text editor is a text editor built upon the Python programming language, with the goal of providing an extendable editing environment through individual packages and themes. Through the Sublime Package Control plugin, users can quickly install modifications to the editor. Additionally, Sublime has its own modules for the Python API, which like in Atom CrossDoc will leverage with custom Python modules to integrate with the CL-Parser.

Vim Text Editor

Vim is a configurable text editor written in the C, and Vim Script languages. Vim Script is the main scripting language built into Vim and the point of support for plugin modules. Although some plugins can utilize languages like Perl and Python if the supportive binaries are present within the running Vim.

5. Implementation Plan

CrossDoc was designed with full-modularity in mind, such that any and all of the pieces we put together would be fully functional as individual units, and serve a greater purpose when paired with other modules. Since CrossDoc is a conglomerate of multiple working systems (Command-Line Parser, Text-Editor Plugins, Remote Comment Stores, and a Git-Hooks Interpreter), our development with reflect that of multiple system creations and a final integration stage. In essence, we will be creating these systems in phases, following an iterative cycle within their development, then complete a final stage where all of the created tools are integrated and put through a testing phase for that integration (see *Figure 6*).



(Figure 6, CrossDoc Gantt chart)

Continuing our work from the last semester of Capstone, where we completed the Command-Line Parser, we begin with Text Editor Implementations. This phase will take a large deal of development time as our four baseline text editors that CrossDoc must be compatible with (Emacs, Vim, Atom, and Sublime), are developed and tested within this phase. We broke this phase further into an R&D phase and an Implementation phase to better familiarize ourselves with the tools and APIs necessary for each editor. Afterwards we continue along the Systems Implementation with the Remote Comment Stores, here we will focus development time on the creation of our sub-system to handle a remote data repository for the CrossDoc comment storing system, along with the repository we will be implementing a web-based storage access (such as a wiki) solution for ease of modifying information in the comment stores.

Following will be the development of our Git-Hooks subsystem that integrates directly with the CL Parser, to ensure the most necessary of CrossDoc functionality, pre and post-commit scripts to separate the commenting system from the codebase. Finally, we will end development with a phase dedicated to the final integrations of these major systems into our final CrossDoc package, within this phase we will complete software and packages necessary for a simple install of CrossDoc and documentation to support.

In summary, we supported the nature of our CrossDoc design, by creating an implementation plan that follows alongside its modularity and allows iterative cycles within major subsystems of the platform.

6. Conclusion

CrossDoc was designed to solve the problem of code dependent documentation, and by that problem's very nature, there can be difficulty in the review and editing processes. The CrossDoc solution that Octo-Docs and Dr. Palmer provides is a commenting system integrated with industry standard text editors to implement an external comment storage system with easy to use tools and remote storage access. Additionally, CrossDoc will administer improvements to the documentation process by allowing a distinct comment categorization system, comment modules, and an adaptive comment history. This document serves as a technical detail into the design and implementation of the architecture of CrossDoc, and the subsystems within. Software Development teams that integrate their projects with CrossDoc will be administered the tools necessary for programming in a modern environment.