Octo-Docs

Requirements Specification

Dr. James Palmer

Garrison Smith
Peter Huettl
Kristopher Moore

x_____

x_____

11/20/17

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, and Kristopher Moore 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, as a whole, 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.

2. Problem Statement

Knowing that the target audience for our product is software developer both in a team environment, and working individually, we can more thoroughly explain the problem we aim to fix. In traditional commenting and documentation systems, comments are placed sporadically throughout a file and added to the file at sporadic times (depending on the team or individual's preference). This variance in comment location and creation time will often result in incomplete or inadequate documentation. In a team environment, inadequate documentation can often lead to frustration among other developers or even bugs in the codebase due to confusing blocks of code.

This confusion can be costly to development time and budget, and also effects translators whose job is to improve the accessibility of the codebase. In large companies with large development teams, these costs will quickly impact the company's profit margins on a product. The following diagram demonstrates exactly how this disconnect affects the development process:
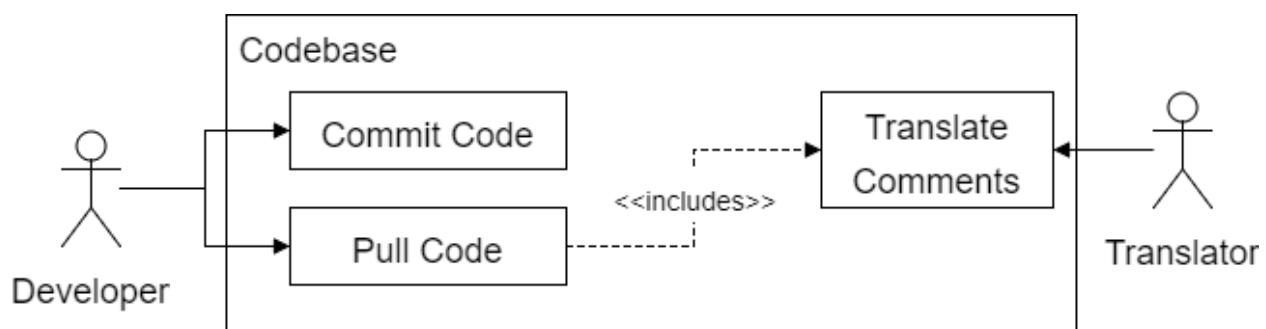

This diagram outlines the primary problems with the current multicultural software development process. Firstly, in order for a developer to properly pull the work done by other developers, the translator needs to have already translated the comments buried within these commits. This is a direct blocker of efficient work and slows down the iteration process. Secondly, the translator is having to do their work within the codebase. This means that the translator is having to dig through code to find the various forms of comments that exist from language to language. Lastly, the diagram also demonstrates how code and comments are currently tightly intertwined. These are the primary problems that CrossDoc aims to solve.

3. Solution Vision

CrossDoc is a command line tool that integrates with select text editors to provide a modularized and seamless commenting system. Contrary to traditional commenting, CrossDoc provides the means to store comments and code documentation externally from the code. By utilizing custom anchors, CrossDoc can naturally insert the comments into the file for developers, while still providing an easily accessible external location to edit these same comments. This solution extends the core functionality of comments by:

- Allowing comments and code to be edited on distinct timelines
- Providing easy access to view and edit documentation externally
- Facilitating compartmentalization of distinct comment groups

These improvements directly solve the problems that large, multicultural software development teams are facing. By providing an external location that is updated only when the comments are updated, comments can be dynamically updated without a dependence on the codebase. Rather than needing to pull the latest update to the codebase to view the most recent comments, the developer will have the changes updated live on their computer. The diagram below outlines the improved workflow provided with CrossDoc:
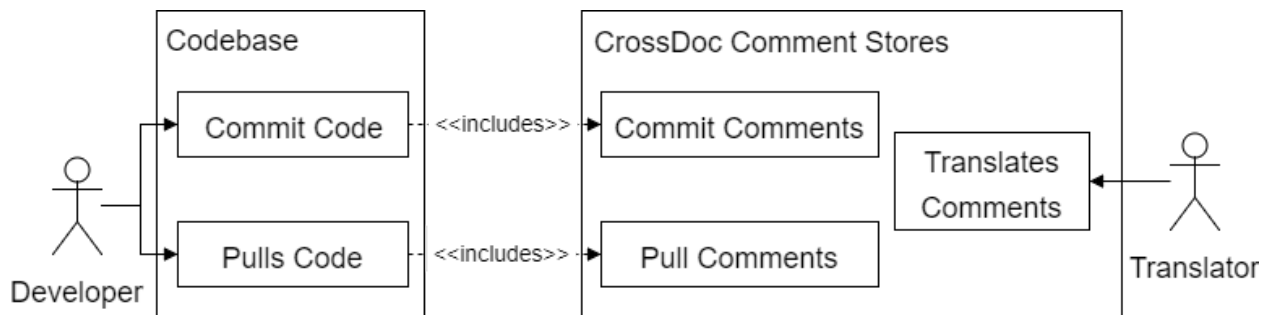
CrossDoc provides a clear separation between the *codebase* and *documentation store*. This facilitates modularization of both files and roles, allowing the developers to work solely within the codebase, while other team members may still view, edit, and use the comments they create.

4. Project Requirements

In order to deliver a product that matches this solution vision, we have developed a collection of functional, non-functional, and environmental requirements. These requirements will act as a guide to ensure that we are providing a product that actually solves the problems presented and demonstrates value to the user. Some of these requirements are higher level abstractions of ideas we intend to implement, while others are low level, actionable items. All of these requirements fall into five key requirement domains.

1. *Implementing core functional data storage capabilities*
2. *Creating an intuitive and easily adoptable system*
3. *Expanding the functional capability of comments and comment storage*
4. *Supporting the scalability of a team environment*
5. *Incorporate powerful tools for experienced users*

Domain 1 encompasses the core functionality of CrossDoc and basic file storage and access. The **2nd domain** is concerned with the process of onboarding users. One large risk to CrossDoc (more thoroughly discussed in the *Risks* section) is a lack of user adoption of the system. **Domain 3** is focused on providing value to CrossDoc's end users by expanding the functionality of comments in a meaningful way. Alongside modularizing comments, we also hope to expand the descriptive capability of any particular comment. The **4th domain** is focused on supporting the use of CrossDoc in a team environment. This domain is particularly important as it addresses specifically the problem presented in this document. Lastly, **domain 5** is targeted at experienced users of the product. Providing support for power-users is key to improving user's workflow.

Each of these domains contain functional, non-functional, and environmental requirements specifically related to their appropriate domain. By nesting these requirements in high and low-level requirements within a domain, we can more accurately demonstrate their relation and the overall purpose of particular requirements. We will address these high and low-level requirements with respect to the requirement type (functional, non-functional, and environmental) while properly addressing their respective key requirement domain.

Functional Requirements

Domain 1: *Implementing core functional data storage capabilities*

The core guiding principle of this domain is ensuring **CRUD** operations [6]. As such, the four primary requirements for this domain are:

    1.a.   *The ability to **create** comments*

    1.b.   *The ability to **read** comments*

    1.c.   *The ability to **update** comments*

    1.d.   *The ability to **delete** comments*

Each of these functional requirements is key to CrossDoc's basic operations and serve as the baseline functional assumption made in other requirements. By following CRUD principles, we are able to ensure persistent storage of comments and provide additional basic features that expand on traditional commenting.

Some of the basic features/requirements of CrossDoc that rely on these CRUD operations are functionalities such as comment searching. In order to accurately search the collection of comments in one or more comment stores, CrossDoc needs the capability to **read** from these data storage locations. **Comment stores** in CrossDoc are comment storage databases that hold the comments and information related to them. These operational principles need to apply regardless of where the comments are stored; whether that be in a local file or a wiki. As such, one low-level requirement within this domain is to provide a consistent API for CRUD operations, regardless of storage location. We will be able to accomplish this by supporting a finite number of comment storage locations (plain-text files, and wiki-based stores).

Because our first domain is focused only on these four core basic operations, there are no levels of requirement nesting, instead, we will build on these requirements in later domains.

Domain 2: *Creating an intuitive and easily adoptable system*

To ensure user adoption of our product, we have a domain of requirements targeted specifically at ensuring that our system is easily adoptable and intuitive. Initial user adoption is one of the largest challenges to any new software product, particularly one that is developer facing. We have identified a *difficult initial installation process* and *lack of documentation* as two of the main contributors to this lack of user adoption. As such, our two high-level functional requirements for this domain directly target these two concerns, while their respective low-level requirements provide distinct, actionable development goals:

    2.a.   *Providing an easy installation processes*

        i.   *Intuitive installation process through package managers*

    2.b.   *Ensuring the product is debuggable and well documented*

        i.   *Develop command specific help text*

        ii.   *Provide identical command detection*

        iii.   *Graceful error handling and response*

Providing an easy installation for new users has been a priority since the beginning of developing this project. As such, in our Technical Feasibility paper [4], we researched the viability of package management systems and which most accurately fit our project. Our current plan is to provide CrossDoc through PIP and Homebrew. (Although requirement **2.a.i** is the only *functional* requirement nested in requirement **2.a**, it is not the only requirement)

When creating a software development tool targeted at developers, documentation is imperative. Specifically, we plan to solve the documentation issue by providing command-level help text (**2.b.i**). This means that, although CrossDoc will have documentation and help-text associated with the whole of the program, a user will also be able to find documentation for any command they try to use. This will provide the vital usability information to users in small, easily understood chunks.

In order to receive command-level help text, the user first needs to know which command they are looking to run. To accommodate this, CrossDoc will provide helpful response text on mistyped commands that directs the user to the commands they were intending to type (**2.b.ii**).

```
λ python src/cdoc.py fetch-comm
cdoc: 'fetch-comm' is not a command.

Similar commands:
  fetch-comment
```

(Example command detection message)

Lastly, it is important that CrossDoc gracefully handles any errors it encounters and responds with an appropriate message so our new users can accurately debug the issue (**2.b.iii**). This error handling will permeate our program and every command to avoid generic error messages.

Domain 3: *Expanding the functional capability of comments and comment storage*

Alongside providing a seamless transition into using CrossDoc is distinguishing our product from traditional commenting. We will accomplish this meaningfully expanding the functionality present in traditional commenting systems. This will ensure that CrossDoc is easy to adopt *and* worth adopting.

3.a. *Implement comment scoping*

3.b. *Integrate dynamically toggleable comment sets*

3.c. *Provide external commenting capabilities*

  i. *Create and insert comments from the command line*

  ii. *Modify comments directly from the comment stores*

Each high-level requirement within this domain targets a specific feature that CrossDoc aims to implement to expand the descriptive capabilities of documentation. Comments have always had relational information provided by the location of a comment, but previously this meta-data was limited. CrossDoc will provide comment scoping that describes both the beginning and end scope of a particular comment, rather than only the beginning (**3.a**).

Additionally, comment categories will be a primary feature of CrossDoc, as they will allow users to group comments that serve a similar purpose into distinct views. Requirement **3.b** is focused specifically on the ability to toggle between these views within supported text editors through convenient hot-keys. A future requirement domain pertains to the modular aspect of the comment set implementation. **Comment sets** are collections of related comments spanning any number of comment stores. Both requirements **3.a** and **3.b** are focused on expanding descriptive properties of comments.


```
1   public class TestClass {
2     private int testField;
3
4     public TestClass(int testField) {
5       this.testField = testField;
6     }
7   // <&> fa09129:nakai:0
    Overview
    Returns the secret value of the instance
    which is useful for secret calculations
8     public int getSecretValue() {
9       return testField / 2;
10    }
11  }
```

```
1   public class TestClass {
2     private int testField;
3
4     public TestClass(int testField) {
5       this.testField = testField;
6     }
7   // <&> fa09129:nakai:0
    TODO
    Change this method to return a double
8     public int getSecretValue() {
9       return testField / 2;
10    }
11  }
12
```

(Example comment sets)

Requirement **3.c** focuses on expanding *how* and *when* the user can edit and view comments as opposed to *what* the comments say. This will be accomplished by featuring the ability to create, read, update, and delete [6] comments from the command line (**3.c.i**) as well as the comment stores directly (**3.c.ii**). This separation of disciplines is one of the key aspects of CrossDoc, and one that directly addresses the problem that CrossDoc aims to solve. With this separation, comes the added functionality and adaptability that these requirements address.

Domain 4: *Supporting the scalability of a team environment*

CrossDoc is a product that is targeted at both individual developers, and software development teams. To provide value to these target markets, CrossDoc needs to support the large-scale operations required in a team environment in a way that is flexible enough to not sacrifice the small-scale uses of the product. As such, this domain contains requirements focused on the **scalability** and **portability** of CrossDoc. These two guiding principles will ensure that the product can reach that ideal balance of large-scale support while not ignoring the small-scale uses. The two main high-level requirements within this domain directly target the two principles respectively:

    4.a.   *Flexible and modular comment storage*

       i.   *Support for multiple distinct comment stores in a project*

      ii.   *Flexible classification of comment types*

    iii.   *Comment store hierarchy*

    4.b.   *Easily share documentation data between users*

       i.   *Support for wiki comment storage*

      ii.   *Provide project-specific customizable configuration files*

Because this domain addresses a problem that CrossDoc aims to solve, there are many requirements nested within. Firstly, requirement **4.a** addresses the **scalability** of CrossDoc. Specifically, we plan on implementing support for multiple comment stores within a single project (**4.a.i**). By allowing the user to add any number of distinct comment stores, we provide the user with the tools to modularize and compartmentalize their documentation base. This also provides support for user-specific comment stores as a single employee in a team may want to include specifically their own comment store alongside the project-specific comment stores.

The next requirement addressing the scalability of CrossDoc is the implementation of comment classifications, otherwise known as comment sets (**4.a.ii**). By modularizing comments one level deeper than comment stores, teams can still filter the combined comment stores by meaningful classifications. Requirement **3.b** addressed the purpose and use-case for comment sets, but this requirement is particularly interested in the actual implementation and support of these sets.

Lastly, in regards to scalability, CrossDoc will support a comment store hierarchy (**4.a.iii**). This means that comment stores may inherit from other comment stores, reducing redundant data and improving polymorphism of the documentation.

The second high-level requirement within this domain is requirement **4.b** which focuses on the portability of CrossDoc. We will accomplish this through the support of a wiki-based comment store (**4.b.i**). By supporting wikis, we expand the accessibility of comment data to any individual or department on the network. This requirement is an example of how the product aims to improve inter-departmental documentation access because wikis include and easy-to-view and easy-to-edit interface.

The last functional requirement included in this domain is requirement **4.b.ii**, the inclusion of a highly customized configuration file. This configuration file will be used to reference the multiple comment stores associated with a specific project, as the configuration file is initialized on a project-by-project basis. By providing configuration files for each project, we ensure that users of the product can retain the portability of their pre-existent comment stores, while also allowing the product to be tailored to each project.

Domain 5: *Incorporate powerful tools for experienced users*

Because CrossDoc is a software development tool, it is important that the product can be optimized to improve workflow efficiency. This domain focuses specifically on this aspect by outlining requirements that improve the user's interaction with the product. The following functional requirements demonstrate exactly this:

5.a. *Implement command shorthand alternatives*
5.b. *Provide text-editor specific command hotkeys*

Firstly, requirement **5.a** is focused on simplifying the use of CrossDoc for experienced users. For example, rather than needing to type *"cdoc fetch-comment,"* we provide an alias for the command that simplifies to *"cdoc fc."* Although this only removes the need to type a few characters, these small time improvements add up to a greatly improved workflow. A prototype example of this can be seen in the image below.

```
λ python src/cdoc.py fetch-comment
usage: fetch-comment <commentId>
```

```
λ python src/cdoc.py fc
usage: fc <commentId>
```

(Example command shorthand)

Secondly, requirement **5.b** pertains to the text editor integration of CrossDoc. Most development occurs in text-editors or IDEs and in order to facilitate this, CrossDoc will feature keyboard hot-keys that quickly execute the most important CrossDoc commands. These hot-keys will perform actions such as switching comment sets, insert a CrossDoc comment and delete a CrossDoc comment. By optimizing how quickly developers can perform these actions, we provide meaningful reasons for users to adopt our system and continue developing with it.

Non-Functional Requirements

In contrast to the functional requirements, the non-functional requirements are used to structure what CrossDoc will be. To ensure the system is capable of providing all the domain requirements along with the original design goals, it must fulfill the non-functional requirements of Maintainability, Performance, Reliability, Security, and Usability.

Maintainability

One characteristic that the CrossDoc system must have, is high maintainability or modifiability. To adhere to the Domain 2 requirement of a highly adoptable system, one aspect of adoptability is the rate of which potential fixes can be applied to the system. To provide updates to the system in a quick and efficient manner the system must first be designed in a manner that easily allows for modifications. CrossDoc must be built on a foundation of modular functions that are loosely linked, providing a structure of independent actions working within a larger system. Additionally, CrossDoc systems will be implemented with detailed source documentation to assure swift comprehension of original coding practices and provide a modified solution when necessary.

Performance

For the CrossDoc system, the overall performance of it is highly important for numerous reasons, with a primary concern on providing swift commands and actions to end-users. A fast and efficient system will support our Domain 2 requirement: Creating an intuitive and easily adoptable system. The CrossDoc system must meet ideal scores in Benchmark tests in response time and computer resource utilization. In operation the system will be responsible for reading/writing data and transmission of the data to comment stores, the response time for the reading and writing actions must be above 1.0 MB/s. Performance requirements for web-based data transmission to and from comment stores must be at least 5.0 Mb/s. Although the performance of web data transmission may be bottle-necked from an end-user connection, the transmission must not be slowed from CrossDoc implemented systems. To ensure the system meets or exceeds the performance requirements, five Benchmarking tests will be conducted the average result of these tests provide the basis for success or failure.

Reliability

The Domain 2 requirement of an easily adoptable system requires that CrossDoc be a highly desirable system to end-users to accommodate adaptability. The reliability of CrossDoc is a critical part of its desirability, with reliability defined as the systems ability to provide its functionality without failure over a specific period of time. To measure the failure rate of the CrossDoc system, we will utilize an MTBF (mean time between failure) statistic to monitor failures of the system. To provide a stable and consistent system, CrossDoc must maintain a minimum of two failures per six-month cycle period. With a failure of the system being defined as "a complete shut-down of the application/plugin, such as a crash" or "a component or function of the CrossDoc system which does not perform its required task when executed".

Security

As CrossDoc will feature the ability to both edit comment stores through the application or its respective plug-ins, and through an online system such as a wiki, security of user's data is of high priority. The CrossDoc system will resolve this issue by providing layers of Authentication and Authorization. In the Authentication layer, the system will check for valid user credentials before allowing access to the respective comment stores. The Authorization layer of the system will only allow users with proper access levels to utilize respective tools. The access levels will consist of: Administrator, Editor, Viewer. These security checks allow the system to be maintained in a safe manner, while accessible in an online environment, protecting the end-users from data manipulation of outside sources. Additionally, these requirements provide a means for users to incorporate access for a multitude of users in a protected environment supporting our Domain 4 requirement: Supporting the scalability of a team environment. For testing of the system's security measures, three Penetration Tests of the system through the application and online channels will be performed to evaluate Vulnerabilities in the authentication process.

Usability

In order to create a truly intuitive system, it first must be a system that is inherently designed for usability. The system should be efficient, effective, easy to use, and ultimately have a small learning curve. For CrossDoc to comply with this requirement it must provide users with support documentation such as readmes, operation documentation, and help functionality for all of its application facets and plug-ins. Additionally, CrossDoc must be designed in a user-friendly format, such that the operations associated are intuitive to an individual unfamiliar with the system. Measurement of the system's usability will be reflected in a number of statistics:

1. Average time it takes a user to complete a given task, in direct comparison to time taken by a developer
2. Average amount of errors made by a user, before a task is completed
3. Overall attitude of the user, and emotion during operation of the given task

Using these metrics we will perform a number of test runs of CrossDoc with volunteer users, to record information about their experience, to determine the overall usability of the system.

Environmental Requirements

Alongside our functional and nonfunctional requirements, there are also several environmental requirements. These are requirements that were provided by the original problem statement and the several outside software integrations. Due to the sparse nature of these requirements, they span multiple distinct domains. This section will address these requirements in order of their respective domains, and explain how these requirements conform to the larger picture of said domain.

Domain 2: *Intuitive and easily adoptable systems*

The intuitive and easily adoptable systems have many functional and nonfunctional requirements within it because it needs to be able to make sure that users can acquire the document in a functional and easy package managing system and be able to integrate on a universal scale for developers. The environmental requirement that comes with integrating for universal systems is making sure that the product runs on popular text editors.

For example, the popular text editors that are used in development are **Sublime, VIM, emacs, and Atom**. In addition, if the product runs on these text editors then it will be able to adapt to other text editors that people may use or even some IDEs that are used in development. This requirement of making sure that the product runs on these text editors is considered an environmental requirement because of how the product should work in different environments of text editors. This then relates back to the idea that we want to make sure that our product is easily adoptable for most systems and allows many users to then use this product on a familiar system that they might be used to.

Domain 4: *Scalable system that can be used in team environments*

This domain level requirement expands into two different key requirements which are flexible and modular comment storage, and easy to share documentation between users. For the environmental purposes, the constraint that we have with our product is focused more on the easy to share documentation between users. When development teams are working on large company projects together they normally have some central hub in which they keep all their code base and documentation that has happened with the project. One of the most popular version control systems for teams is **Git**. For example, the website GitHub is a central site making sure that the codebase is managed well and that the chaos that happens when projects are being developed can be managed in a central area [5].

In addition, with this integration, our project wants to make sure that we also integrate into a Git environment and this causes us to have an environmental requirement from our client. This Git integration will allow people to still manage the codebase systems like before however, they will also be integrating our functional requirements that include comment stores and configuration files. The Git integration allows a user to also use our comments from the comment stores but when being committed to Git they will not be integrated into the codebase. This will help solve the general problem of separating software from the documentation. The GitHub website is a resource that allows for users to have a team environment setting and this setting is one of our domain level requirements within our project.

Domain 5: *Powerful tools for experienced users and developers*

Our product will be easily accessible and easily used for the end users however we will also have room for the more experienced developers and users so they can use high-level programming for our product. In regards to having an expansion for experienced users and developers, our requirements in this domain level expand into three separate requirements. The shorthand alternatives are considered under a functional requirement and the performance of the command line interface is under the nonfunctional requirements. However, we will be focusing on the environmental requirement which is making sure that our product has an integration with **Javadoc** and **Doxygen**. Both of these tools are ways that documentation can be managed on a broad scale so that the documentation within a code base is easily accessible and easy to handle.

Javadoc is a specific documentation tool allowing programmers to manage their documentation using Java [3]. Doxygen is a web application tool that allows users to create documentation from a source code [2]. Furthermore, our product needs to be able to integrate with these document-based systems because our products end goal is to separate the software from the documentation. So in this project, we want to make sure that the environment of Javadoc and Doxygen are met because both of these systems are used to manage the documentation and the products end goal is to manage the documentation as well.

5. Potential Risks

When developing products for a company, the developers that are creating the product must always consider the major risks that can be taken when developing and releasing the product. The product that we are designing is a new way to implement documentation within a programming language and this, however, can come with some risks for the product. For example, there could be a company that is already creating a product that is similar to CrossDoc, or a user could just prefer the traditional commenting method over using a system like CrossDoc, and finally, the users could have a misunderstanding of what our product does [1].

As our product gets developed, one of the main risks that we may face it that another company out there could be also developing a project based commenting system. In addition, if this risk comes up we want to make sure that we can have a suitable product that can compete with another companies product. We want to make sure that we stick to all of our original ideas and make sure that we stay unique compared to another company. Furthermore, making sure that our product is set up so that it can last in the long run is also a key solution to our product. We also want to make sure that our product can adapt to the technological expansion in the field of technology [1]. This product should be able to adapt easily and this can make sure that the product can stay neck and neck with our competitors. This shows that one of the risks that we may come up against is having a competitor who is creating a program that is similar to ours. However, there are many risks that can come up in this field.

Another risk that we may come up against is that users may just continue to use the traditional documentation of group project programming. Our goal in this risk is to make sure that we can make the product an easy-to-use environment and make sure that the product has a versatile management system as well. Keeping this in mind we want to make sure that we stick with our technical feasibilities because they help make the product an easy-to-use product. For example, the package management system that we have chosen which is called PIP (Python Package Index) [7], this packaging system is a universal and easy to access package system for our product and using this allows users to use our product on multiple platforms.

Another feasibility that we have in regards to this risk is that we are using Python for the implementation of the product and this will allow for a user code flow when it comes to using this product as well. This shows that another risk that we may come across is that users will continue to use the traditional documentation for group projects in the workplace.

Finally, another risk that came up within our project is that our documentation and help tools will not be clear enough for users to understand what the product is used for [1]. This risk could affect the end users and make them not want to use the product because they have no idea on what the product actually does. One of the ways that we can manage this risk is to make sure that the users have an easy-to-use product and make sure that our help tools are easy to access and that the product has an easy to download packaging system. This risk will ensure that we will have an easy to navigate command line interface and making sure that the command line interface will have a help tool that will allow the users to access what the product is used for.

Another implementation that we could have in the future is a web document that will explain what CrossDoc is and how it is used in a team environment. Furthermore, we are aware of the many risks that can come up when developing the product but we are also prepared to create and develop solutions that will manage these risks as we develop. In conclusion, the main risks that we are thinking of are companies releasing a similar product, no user adoption, and user not understanding what our product is used for. Our solutions have been to create an easier package system and making sure that we can adapt to the competition, and make sure that our command line interface is well documented with easy-to-use help commands for end users.

6. Project Plan

During the development of a project, there must always be a plan that the group of the project should have as well as a preparation for future plans to come. In this situation, we are making sure that for the project CrossDoc that all the requirements, design reviews, and standards for the project are met. Furthermore, as a group, we must keep in mind that there are many key features that need to be accomplished before the project itself is complete. In addition, each feature requires a few days to complete but some of the features can be done coinciding with each other. For example, the main goal is to make sure that we can get a document complete that outlines our requirements for out document, however, we also need to complete a design review presentation that will also outline our requirements for the document as well as explain the design process for our product. This shows that the project plan can pace itself depending on how certain objectives can get done.
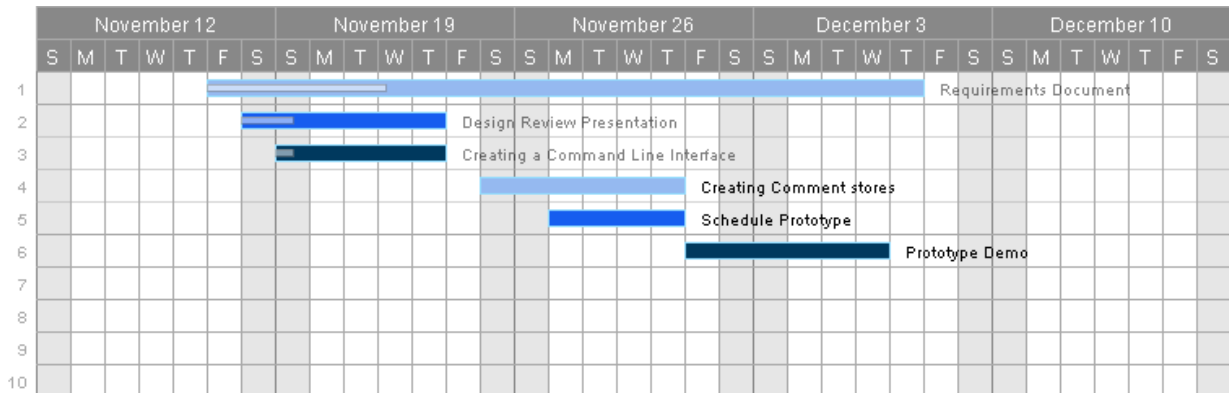
(CrossDoc Fall semester schedule)

The above Gantt chart displays the main goals for the semester, the most substantial of which is the completed requirements document. This also requires us to have a presentation that we want to complete so that we can present the project and what it is required to do as well to our client. Finally, the last goal is to make sure that we can have a working prototype demo ready to show our client so they are aware that we are creating the project and that we can have some information to come together with by the next semester.

The objectives that we need to complete before jumping into the prototype demo is to make sure that we have a command line interface that can be used to help create the command lines for the users as well as making sure that we have the comment stores ready so that when a user uses the command lines that they can initialize into a comment store and start storing their comments within their project. These two objectives will then work together to then create the prototype demo that we need to show to our client by the end of the semester.

7. Conclusion

In conclusion, 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. In the end, as a team, we wanted to make sure that we were able to iron out all the requirements for this project. In addition, we want to better understand how this project is going to function in regards to solving our solution and understanding the problem. The problem right now is that there is no separation between software and documentation for large company projects. The solution is that Crossdoc will bridge the gap to help maintain, organize, and create a commenting based system that allows users to have a separation between software and documentation. Furthermore, we wanted to make sure that as we develop this product we want to highlight the key risks that could happen as well as create a general overview of how we are going to accomplish our end goals. Finally, we wanted to make sure that we can understand the key requirements for this project and lay them out so when we start programming we know exactly what needs to be completed for this project.

8. Sources

[1] - Potential Risks

(http://groups.engin.umd.umich.edu/CIS/course.des/cis375/projects/risktable/risks.htm)

[2] - Doxygen (http://www.stack.nl/~dimitri/doxygen/)

[3] - Javadoc (https://en.wikipedia.org/wiki/Javadoc)

[4] - Technological Feasibility paper

(https://www.cefns.nau.edu/capstone/projects/CS/2018/OctoDocs/assets/documents/Technological%20Feasibility%20Analysis.pdf)

[5] - GitHub (https://github.com/)

[6] - CRUD operations (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete)

[7] - Python Package Index (https://pypi.python.org/pypi)