CrossDoc

Garrison Smith, Peter Huettl, Kristopher Moore, Brian Saganey

School of Informatics, Computing, and Cyber Systems; Dr. James Palmer

Problem Statement

In software development, **comments** are human-readable annotations that describe the code. Currently, these comments can only exist within the code they are describing. This causes several issues such as:

- Poor comment **discovery**
- **Disorganized** documentation
- Not friendly to **non-developers**
- Lack of comment **type-grouping**

```
# The Logger class is responsible for providing output to the console
#
# API documentation
# standard(message)
#   logs `message` to stdout
#
# usage(command=None)
#   logs the usage message for the command that calls this method
#   alternatively, logs the usage message for the given `command`
#
# program(message)
#   logs `message` to stdout prefixed with the program name
#
# fatal(message)
#   logs `message` to stdout prefixed with "fatal" and kills program
#
# Things to do
# * Modify where fatal logs its message (stdout -> stderr)
# * Add a warning logging method that prefixes messages with "warning"
class Logger:

  def standard(message):
    """Logs a message to the user (non-ending)"""

    print(message)
    return
```

Solution & Requirements

Solution: CrossDoc is a comment management system that **decouples** software comments from the code it describes. CrossDoc accomplishes this by connecting external comment stores to a project's codebase.

This separation of concerns enables distinct comment categories, external comment management functionalities, and advanced comment tooling.

These capabilities will improve the documentation systems for both individual developers and software developers working in a team.

Requirements: The primary requirements of CrossDoc are:

- **Simple** setup process
- **External** comment storage
- **Intuitive** comment editing
- **Functional** text editor plugins
  - Atom
  - Emacs
  - Sublime
  - Vim

```
# <&> 20807c [No Set]
# The Logger class is responsible for providing output to the console
class Logger:

  def standard(message):
    """Logs a message to the user (non-ending)"""

    print(message)
    return
```

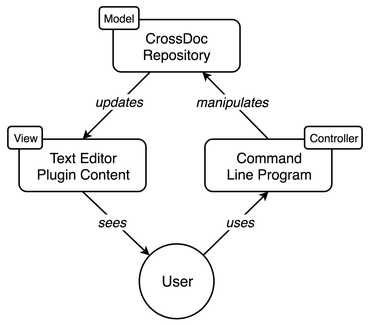
```
# <&> 20807c [TODO]
# * Modify where fatal logs its message (stdout -> stderr)
# * Add a warning logging method that prefixes messages with "warning"
class Logger:

  def standard(message):
    """Logs a message to the user (non-ending)"""

    print(message)
    return
```

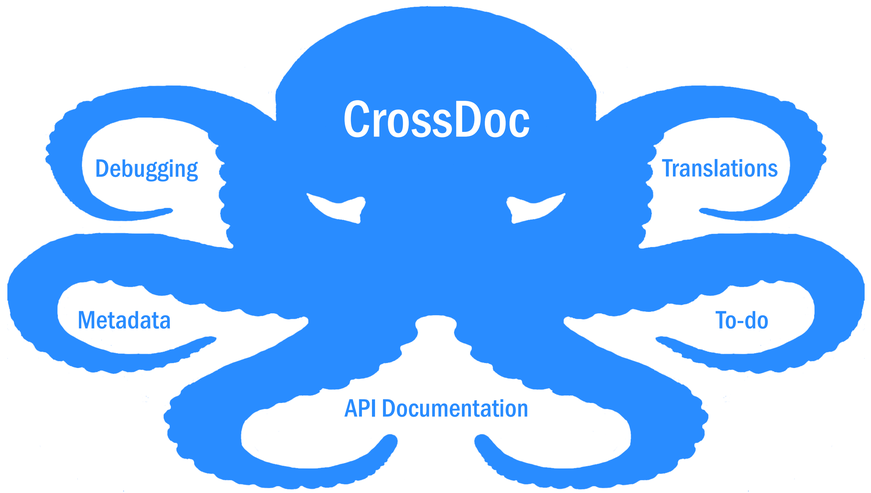
```
# <&> 20807c [Documentation]
# standard(message)
#   logs `message` to stdout
#
# usage(command=None)
#   logs the usage message for the command that calls this method
#   alternatively, logs the usage message for the given `command`
#
# program(message)
#   logs `message` to stdout prefixed with the program name
#
# fatal(message)
#   logs `message` to stderr prefixed with fatal and kills the program
class Logger:

  def standard(message):
    """Logs a message to the user (non-ending)"""

    print(message)
    return
```

Architecture

CrossDoc was designed with a MVC architecture. This architectural pattern is comprised of 3 main components:

- **Model:** The **CrossDoc Repository** stores the system's meta-data.
- **View:** The **Text Editor Plugins** are the visual representation of the system information.
- **Controller:** The **Command Line Program** handles user interaction and manipulates the model.





Technologies

The CrossDoc back-end tool is written in **Python**. The program is distributed and installed through the **pip** package management system.

We utilize the **unittest** and **urllib** Python libraries for automated testing and network requests respectively.

Text Editor Plugins
Atom – JavaScript
Emacs – Elisp
Sublime – Python
Vim – VimScript

Team Collaboration
Communication – Slack
Documents – Google Drive
Task Management – Trello
Version Control – GitHub

Prototype

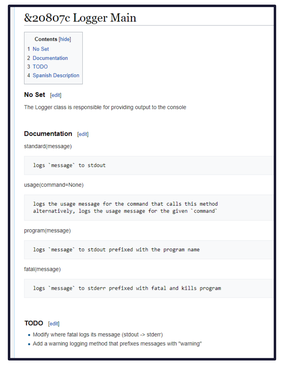
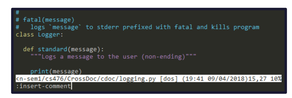


Figure 1: **Wiki** interface to edit comments outside the code.

```
# <&> 20807c [TODO]
# * Modify where fatal logs its message (stdout -> stderr)
# * Add a warning logging method that prefixes messages with "warning"
class Logger:

  def standard(message):
    """Logs a message to the user (non-ending)"""
```

Figure 2: Integrated text editor plugins (**Vim** & **Sublime** pictured)



```
# fatal(message)
#   logs `message` to stderr prefixed with fatal and kills program
class Logger:

  def standard(message):
    """Logs a message to the user (non-ending)"""

    print(message)
:n-semi/c1470/CrossDoc/cdoc/logging.py [dos] (19:41 09/04/2018)15,27 10%
:insert-comment
```

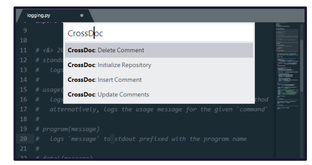
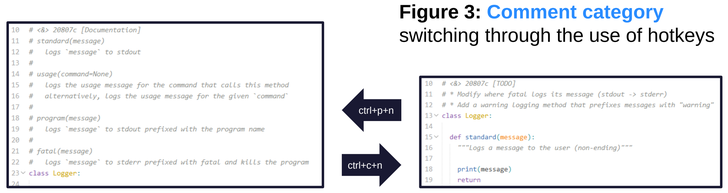


Figure 3: **Comment category** switching through the use of hotkeys




```
# <&> 20807c [Documentation]
# standard(message)
#   logs `message` to stdout
#
# usage(command=None)
#   logs the usage message for the command that calls this method
#   alternatively, logs the usage message for the given `command`
#
# program(message)
#   logs `message` to stdout prefixed with the program name
#
# fatal(message)
#   logs `message` to stderr prefixed with fatal and kills the program
class Logger:
```

```
# <&> 20807c [TODO]
# * Modify where fatal logs its message (stdout -> stderr)
# * Add a warning logging method that prefixes messages with "warning"
class Logger:

  def standard(message):
    """Logs a message to the user (non-ending)"""

    print(message)
    return
```

Conclusion

The tight coupling of comments and code in software projects creates inefficiencies in the development process. These inefficiencies cost development effort and delay work on team projects. CrossDoc aims to fix this by providing the following:

- **Searchable** comment storage
- **External** storage sets
- **Intuitive** wiki interface
- **Distinct** comment categories

We have integrated these features into text editor plugins, and in this way, CrossDoc has addressed the primary requirements outlined.

CrossDoc utilizes an MVC style architecture with its 3 main modules, the **CrossDoc Repository**, the **Text Editor Plugins**, and the **Command Line Program**.