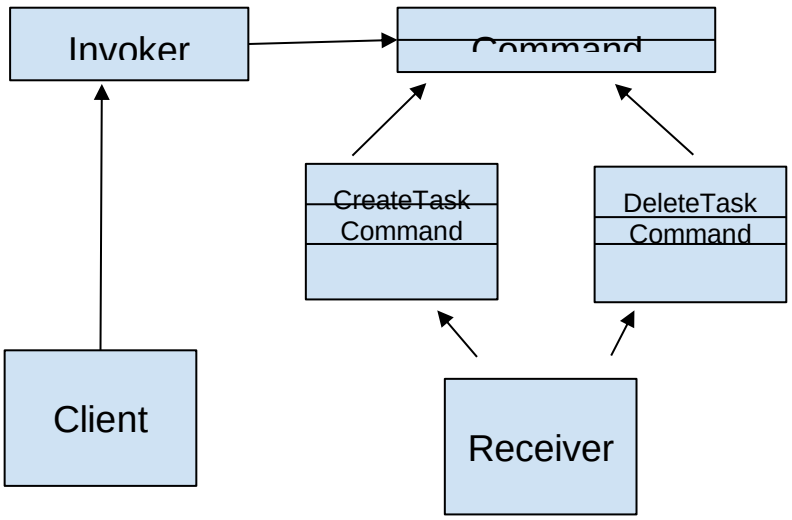
High-level Design

- The architectural pattern we would use to structure our system is Pipe and Filter. The Pipe and Filter architectural pattern is a great fit for organizing a task manager system like our CheckMate. Imagine it as a series of processing stations, or filters, each handling a specific task-related operation like creating, updating, or prioritizing tasks. These filters are like building blocks that we can reuse across the system, making it easy to manage and scale as our needs evolve. They also keep things neat and tidy by separating different tasks and responsibilities, making it simpler to develop and maintain the system over time. Plus, this approach lets us speed things up by processing tasks simultaneously, helps with testing and fixing issues, and allows us to connect with other tools or services seamlessly.

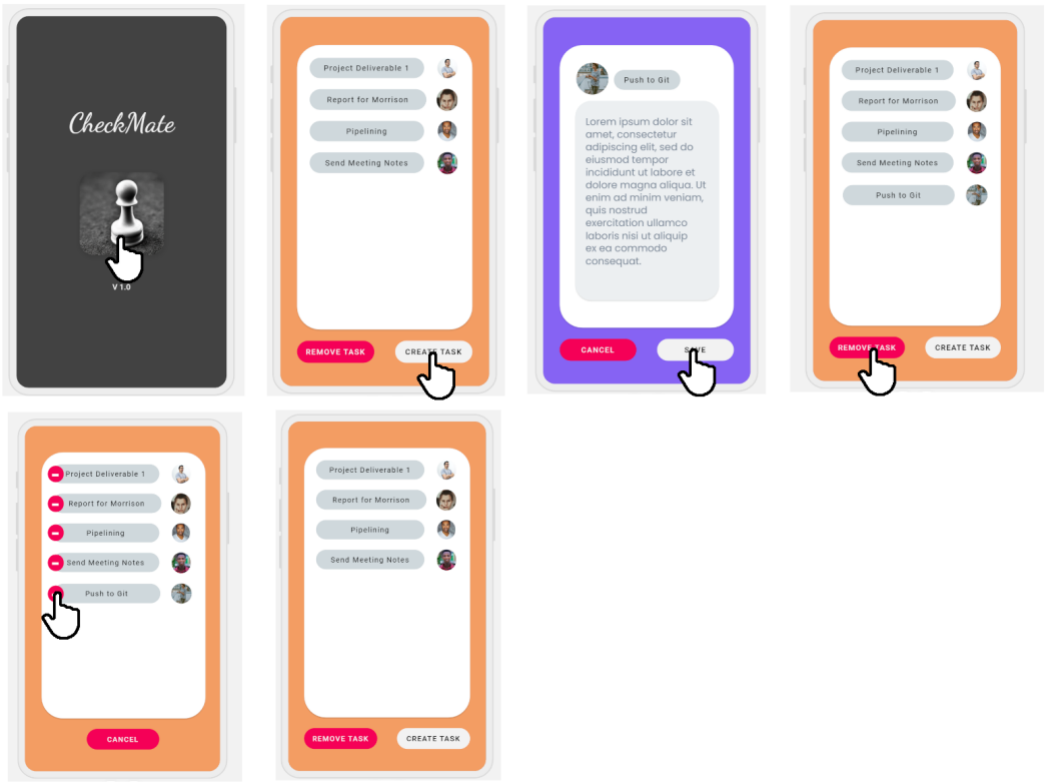
Low-level Design

- The design pattern family that might be helpful for implementing our project would be Behavioral: Command. This pattern can be used to encapsulate user actions as commands. Each action, like creating tasks, assigning tasks, setting due dates, or commenting on tasks, can be represented as a command. It offers flexibility and allows for undo/redo functionality, which is valuable in a task manager where users often need to perform various operations on tasks.
- Pseudocode:

```
class CreateTaskCommand:
    """
    A command to create a task
    """
    def __init__(self) -> None:

    def execute(self, filename: str) -> None:

    def undo(self) -> None:

class DeleteTaskCommand:
    """
    A command to delete a task
    """
    def __init__(self) -> None:

    def execute(self, filename: str) -> None:

    def undo(self) -> None:

class EditTaskCommand:
    """
    A command to edit task
    """
    def __init__(self) -> None:

    def execute(self, filename: str) -> None:

    def undo(self) -> None:

def main():
    """
    >>> item1 = MenuItem(DeleteFileCommand())

    >>> item2 = MenuItem(HideFileCommand())

    # create a task
    >>> test_file_name = 'test-file'

    # delete a task
    >>> item1.on_do_press(test_file_name)
    deleting test-file

    # edit a task
    >>> item1.on_undo_press()
    restoring test-file

    """
```



Design Sketch



Description

The user opens the app. The user then attempts to create a task. The user fills out a description, and presses on save. The user notices that they made a mistake, so they need to remove the task. The user presses on the remove task button. The user presses on the minus sign next to the task they want to remove. The task is successfully removed.

Rationale

The above wireframe makes use of the pipe-and-filter high-level design process in order to inform the user interface. Since multiple processes, such as adding or removing tasks, could be happening simultaneously, it is imperative that the user interface is as simplistic as possible. Small icons depicting the users that last updated task information appear next to the task itself. Additionally, the number of buttons is minimized to simplify interactions to minimize the amount of commands our app has to process, simplifying both user interaction and development costs needed when bug fixing, which could be an issue with the code-and-fix approach. Simplification allows for more user-friendliness in our app. It invites people to come back and keep using it as they do not have to stress about learning all the features of the app and can just do their own tasks peacefully and effectively.

Process Deliverable

The Code and Fix method has been a valuable approach for our project. In this essay, we will discuss the progress we have made while adopting the Code and Fix Software Engineering method. We will discuss the advantages and disadvantages of this method.

One of the primary advantages of the Code and Fix method is its flexibility. This approach allows us to adapt quickly to changing requirements and evolving user needs. We have been able to make rapid adjustments and updates to our app, responding promptly to user feedback and market trends. This flexibility has enabled us to explore new features, experiment with different designs, and refine the app to ensure we deliver the best user experience possible as we progress through the developmental stages. This is important as we need to apply Code and Fix's best qualities throughout the development process in order to know exactly what we have to fix. It allows for a more thorough understanding of the problem and how we can learn to tackle it better and more efficiently.

Code and Fix encourages an iterative development process. Instead of spending extensive time on elaborate planning and documentation, we have embraced the philosophy of "build, test, learn." We come up with a feature or component, test it within our team, and learn from our feedback and usage patterns. This iterative approach has allowed us to refine our app design incrementally, resulting in a product that better aligns with user expectations as we continue to work on it. The emphasis in this process is the learn component. We need to continuously learn from the mistakes that we make and fix anything that does not abide by what our stakeholders want. Since these stakeholders are the most important part of learning which requirements are necessary, they are the people who we constantly go back to for feedback.

While Code and Fix allows for quick development, it is not without its challenges. The absence of comprehensive planning and design documentation can lead to hasty decisions or suboptimal code. However, each mistake has become an opportunity for learning and improvement. We have instituted regular reviews and retrospectives to identify areas for enhancement, enabling us to continually refine our development process as we move forward. Our reviews allow us to circle back and know what works and what does not so we can not go through the same mistakes in the future. These reviews save us time and place us in a better light with our stakeholders. These challenges show us that although Code and Fix may seem menial with us constantly going back and forth through the developmental process, our final product will be an incredible update and innovation than our starting product.

In conclusion, our experience with the ongoing use of the Code and Fix method for CheckMate has been promising. It has allowed us to embrace the ever-changing nature of software development, pivot when necessary, and focus on delivering a product that aligns with the needs of developers. The flexibility, iterative approach, user-centric focus, and the learning opportunities provided by this method have proven to be valuable assets in our ongoing project.

While Code and Fix may not be the answer for all software development scenarios, it continues to serve us well in the dynamic environment of our task management app. By combining it with careful planning, thoughtful architecture, and a commitment to addressing user needs, we anticipate that our ongoing adoption of the Code and Fix method will remain a viable and productive approach as we work toward the completion of CheckMate.