Sprint 3 Deliverables

Table of Contents

Defining criteria and the corresponding Metrics

Completeness of the solution direction (i.e. to what extent are all key functionalities covered in the design) [Functional completeness; functional correctness]

- All critical game functionalities are covered, and the submission clearly explains how they are addressed.
- All classes have relevant attributes, methods, relationships, and cardinalities, covering all required functionality.

Metric: The number of critical functionalities covered out of all the expected ones, rated on a scale of (poor) to 5 (excellent), code coverage percentage is converted to the rating scale out of 5.

The rationale behind the chosen solution direction [Functional appropriateness]

- Design rationale covers all required areas: classes, relationships, inheritance, cardinality, and design pattern.
- Solid and compelling justifications for the architecture and design choices are provided.
- Used Design Patterns are identified, their choice is explained and used appropriately, or the lack of application of Design Patterns is well justified.
- Sound object-oriented design principles are followed; "God classes" are avoided.

Metric: Clarity and justification of the design decisions and trade-offs, rated on a scale of 1 (poor) to 5 (excellent), then converted to a percentage.

Understandability of the solution direction [Appropriateness recognizability]

- A clear and neat Class Diagram for the Fiery Dragons game.
- Correct usage of notations in the class diagram.

Metric: Ease of comprehension of the design and implementation, rated on a scale of 1 (difficult to understand) to 5 (very easy to understand), then converted to a percentage.

Extensibility of the solution direction (in anticipation of extensions to the game for Sprint 4) [Modifiability]

- Use of principles like Open-Closed principles.
- Use of interfaces or abstract classes for loose coupling and separation of concerns.
- Separation of UI and business logic.

Metric: Degree of modularity and adherence to design principles (like the Open-Closed Principle), rated on a scale of 1 (low extensibility) to 5 (highly extensible), then converted to a percentage.

Quality of the written source code (e.g., coding standards, reliance on case analysis and down-casts) [Maintainability]

- The branch includes a 'README' or similar documentation that describes its organisation and details the locations of specific files.
- Adhering to coding standards like naming conventions
- Ensuring most repeated entities are inherited and polymorphed to the lowest level (abstract class/ interface)
- Reliance on very few external libraries
- Modularity of the code (Functions are appropriately sized, code is organised into packages, directories or namespaces)

Metric: Adherence to coding standards, appropriate use of abstractions, and avoidance of code smells rated on a scale of 1 (poor maintainability) to 5 (excellent maintainability), then converted to a percentage.

Aesthetics of the user interface [User engagement]

- The game and all its elements are visible
- The visual appeal of the game (layout, colour schemes)
- The game has well-defined affordability (Know how to interact with the game)
- Appropriate feedback for all game interactions.

Metric: Visual appeal, layout, colour schemes, and clarity, rated on a scale of 1 (unappealing and hard to use) to 5 (highly appealing and easy to use).

Sprint 2 Solution Reviews

Ahmad Sprint 2 Review

Functional completeness and functional correctness - 3/5

The implementation covered most of the essential functionalities required for the game. However, certain critical features, such as the volcano cards and cave cards, were missing. This indicates that while the core elements were present, some key functionalities were not addressed. This level of coverage earned Ahmad a score of 3 out of 5, reflecting a mostly complete but not fully comprehensive implementation.

Functional appropriateness - 3/5

The design rationale was detailed but did not address all necessary areas. He provided solid justifications for his architectural and design choices. However, despite the explanations, the actual presence of design patterns in the class diagram or code was lacking. This inconsistency led to a score of 3 out of 5. The evaluation highlighted the importance of aligning design intentions with actual implementation.

Appropriateness recognizability - 5/5

The class diagram was neat and clear. It was easy to read and understand, effectively communicating the relationships and interactions between different components. The mostly correct use of notations further enhanced the comprehensibility of the design, crucial for both current development and future modifications. This high level of clarity and ease of comprehension earned a perfect score of 5 out of 5.

Modifiability - 2/5

The solution showed limited use of inheritance, hindering the ability to extend the system easily. The design did not adhere to the Open-Closed Principle, a fundamental guideline for creating maintainable and extendable systems. Additionally, there was a lack of separation between the user interface (UI) and business logic, complicating future enhancements and maintenance efforts. This led to a score of 2 out of 5, indicating significant room for improvement in making the design more flexible and modifiable.

Maintainability - 3/5

The maintainability of the code was affected by several factors. The absence of a README file meant no clear documentation to guide new developers or reviewers through the project's structure and components. While coding standards were generally followed, the scarcity of comments reduced the code's readability and ease of understanding. Despite low reliance on external libraries, which is a positive aspect, the functions were not very modular, making the codebase harder to manage and evolve. These issues contributed to a score of 3 out of 5.

User Engagement - 4/5

The user interface design was aesthetically pleasing and intuitively easy to use. Most elements of the game were clearly visible, and the overall layout was user-friendly. However, some UI elements were missing, potentially impacting the user experience and the ability to interact with all aspects of the game. This aspect of the project received a score of 4 out of 5.

Fahad Sprint 2 Review

Functional completeness and functional correctness - 5/5

The design successfully covered all key functionalities expected for the game, ensuring that all critical features were implemented and operational. This comprehensive coverage earned a perfect score of 5 out of 5, reflecting a robust and complete implementation.

Functional appropriateness - 4/5

The design rationale was detailed and thoughtful, encompassing all necessary aspects such as class responsibilities, relationships, inheritance, and design patterns. However, some classes did not have their responsibilities appropriately divided, leading to the presence of a god class. This issue was not justified within the design rationale. Despite this, the design patterns were identified, choices were well justified, and the overall rationale was sound, resulting in a score of 4 out of 5.

Appropriateness recognizability - 4/5

The class diagram was clear and easy to read, effectively communicating the structure and relationships within the system. However, the inclusion of packages, which should not be part of a UML class diagram, docks some points. The notations used were correct, contributing to a mostly comprehensible design, which earned a score of 4 out of 5.

Modifiability - 5/5

The design was highly modifiable, characterised by loose coupling and adherence to principles like the Open-Closed Principle. This made it easy to extend the game functionality without requiring modifications to existing code. The use of interfaces or abstract classes promoted separation of concerns, and there was a good separation of UI and business logic, all of which contributed to a perfect score of 5 out of 5.

Maintainability - 5/5

The maintainability of the code was excellent, adhering to coding standards and making good use of inheritance. There was a low reliance on external libraries, and the code was very modular, organised into packages that facilitated easy management and evolution. These practices resulted in a score of 5 out of 5.

User Engagement - 5/5

The user interface was highly engaging, with all game elements clearly visible and the overall layout being aesthetically appealing. The UI provided appropriate feedback for all game interactions, and the player's turn was displayed at the bottom of the page, ensuring that players were always aware of their status. This thorough and user-friendly approach earned a score of 5 out of 5.

Hanif Sprint 2 Review

Functional completeness and functional correctness - 5/5

The implementation covered all critical game functionalities, ensuring that every required attribute, method, relationship, and cardinality was addressed. This comprehensive coverage resulted in a perfect score of 5 out of 5.

Functional appropriateness - 4/5

The design rationale was comprehensive, addressing all necessary elements such as classes, relationships, inheritance, and design patterns. However, the presence of god classes indicated a concentration of responsibilities that was not ideal. Despite this, the design patterns were well identified and justified, leading to a score of 4 out of 5.

Appropriateness recognizability - 4/5

The UML diagram provided by Hanif was somewhat difficult to read due to the lack of colour coding or shading, which could have improved legibility. Additionally, the inclusion of packages, which should not be part of a UML class diagram, was a drawback. However, the rest of the notations were correct, resulting in a score of 4 out of 5 for understandability.

Modifiability - 4/5

The design showed good use of interfaces and abstract classes for loose coupling, along with dependency inversion principles. There was a clear separation of UI and business logic, which facilitated easier modifications and extensions. This approach earned a score of 4 out of 5 for modifiability.

Maintainability - 4/5

The code adhered to coding standards, but some docstrings were missing, which affected readability and ease of understanding. Despite this, the overall adherence to standards and good use of inheritance contributed to a score of 4 out of 5.

User Engagement - 5/5

The user interface created was aesthetically pleasing, easy to understand, and use. All game elements were clearly visible, contributing to a high level of user engagement. This thorough and appealing UI earned a perfect score of 5 out of 5.

Umair Sprint 2 Review

Functional completeness and functional correctness - 5/5

The implementation was thorough, covering all expected functionalities and ensuring that class relationships, methods, and cardinalities addressed all required aspects. This comprehensive approach earned a perfect score of 5 out of 5.

Functional appropriateness - 4/5

Umair appropriately identified and used all design patterns, although the justifications for some of these choices were not entirely satisfactory. This led to a slightly reduced score of 4 out of 5, reflecting the need for more robust reasoning behind design decisions.

Appropriateness recognizability - 3/5

The class diagram provided was clear and easy to read, but included packages that should not be part of a UML diagram. Additionally, the appropriate notations were not used consistently in all places. These issues resulted in a score of 3 out of 5 for recognizability.

Modifiability - 3/5

The absence of necessary abstractions in some areas led to a design that did not fully adhere to the Open-Closed Principle. The lack of a strategy pattern in appropriate locations necessitated modifications to the base class, and there was not a very good separation of UI and business logic. These factors resulted in a score of 3 out of 5 for modifiability.

Maintainability - 3/5

While the code adhered to coding standards and made good use of inheritance, it was missing some docstrings. There was low reliance on external libraries, but the absence of a README and lack of package usage affected the maintainability score, which was 3 out of 5.

User Engagement - 3/5

The user interface was moderately appealing, with all game elements visible. However, the overall visual appeal and ease of use could have been improved. This resulted in a score of 3 out of 5 for user engagement.

Sprint 3: Consolidation of Ideas

For sprint 3, ideas from across the team members' designs were implemented into the new design. Key aspects that were carried over from each member’s design was their implementation of certain design patterns.

The singleton pattern from Umair’s design was incorporated in a fashion through the use of the EventManager class, with a single instance handling all events, while providing a global access point to this manager. Umair’s design incorporated such managers in a variety of ways, through PlayerManager, TileManager, and ChitManager.

The observer pattern from Fahad’s sprint 2 implementation was incorporated as well. A subscription mechanism was incorporated to notify various objects about relevant events. At runtime, game elements subscribe to relevant events and are notified of the events occurring throughout the game, and they can then process the trigger as defined in their class.

The command pattern was also incorporated, and inspiration was taken from Hanif’s implementation of the pattern from sprint 2. It was used to handle actions by decoupling the object that invokes the operation from the object that performs it. It promotes reusability as actions can be reused by different senders invoking the same command. Classes such as DragonCardCommand are part of this implementation.

The idea of using .json files to save or edit game configurations instead of hard coding values was taken from Hanif’s implementation. Aspects such as board size (number of tiles), player count, and more can be entered into these .json files.

Sprint 3: New Ideas

We have introduced a number of new ideas and elements to improve our prototype for the Fiery Dragons game implementation.

Introduction of iterable classes to reduce dependency on array lists
Using iterable classes instead of arrays or array lists can make the code more flexible and easier to maintain, especially when dealing with collections of objects. Iterables provide a standard way to iterate over the elements, making the code more generic and reusable and allows for better abstraction and encapsulation since the internal representation of the collection can be changed without affecting the code that iterates over it.

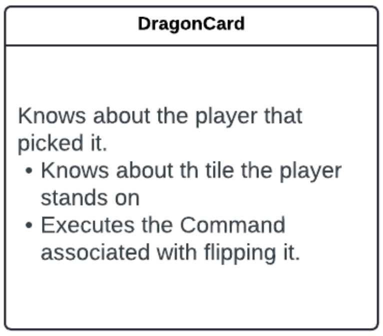
Changed tile implementation to be more graph-like
A graph-like implementation better models the relationship between tiles on the gameboard leading to a more intuitive representation where each tile is a node that knows about its adjacent tiles. It facilitates extensibility and new types of tiles can be introduced while using the same tile node representation. This also promotes separation of concerns as movement logic can be encapsulated within the tile objects, decoupling it from the game and making movement the responsibility of the tiles.

Use of player factory to create players iteratively: Introducing a factory pattern for creating player objects can help in separating the object creation logic from the rest of the code leading to a more modular and maintainable code. It also promotes extensibility as new players such as AI Players can be introduced without requiring any modification to existing code that uses the players and promotes code reuse as the player factory can be shared across different parts of the application.

Introduced new idea for track of tiles containing players in an ArrayList and maintaining a pointer to the current player's tile node using the same ArrayList.
This approach combines the idea of tracking only the tiles with players and maintaining a pointer to the current player's tile node within the same data structure. This makes the iteration much more efficient and **simplified player turn management** as now it's a very straightforward process to just update the pointer to the next tile node in the ArrayList effectively moving the turn to the next player without having to iterate over the entire board. Also makes it very easy to **provide updates to game state** as when a player moves you only need to update the current player's tile node references in the array list making the process very efficient and thus improving the overall game performance and responsiveness.

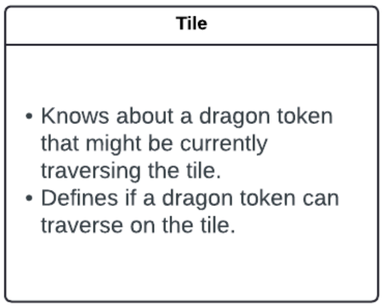
CRC

Initial CRC Draft



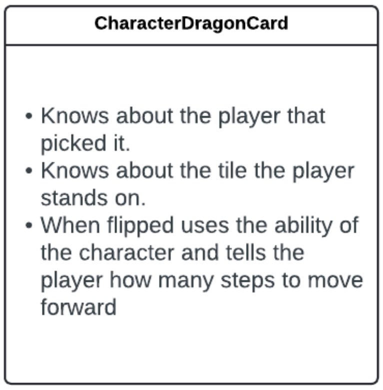
DragonCard

Knows about the player that picked it.
- Knows about th tile the player stands on
- Executes the Command associated with flipping it.



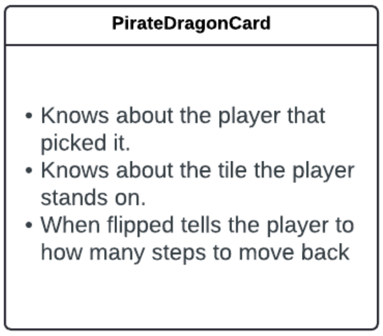
Tile

- Knows about a dragon token that might be currently traversing the tile.
- Defines if a dragon token can traverse on the tile.



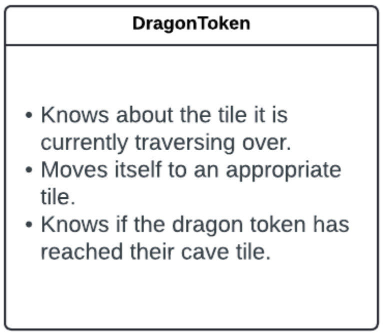
CharacterDragonCard

- Knows about the player that picked it.
- Knows about the tile the player stands on.
- When flipped uses the ability of the character and tells the player how many steps to move forward



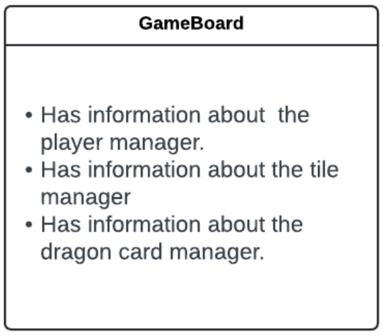
PirateDragonCard

- Knows about the player that picked it.
- Knows about the tile the player stands on.
- When flipped tells the player to how many steps to move back



DragonToken

- Knows about the tile it is currently traversing over.
- Moves itself to an appropriate tile.
- Knows if the dragon token has reached their cave tile.

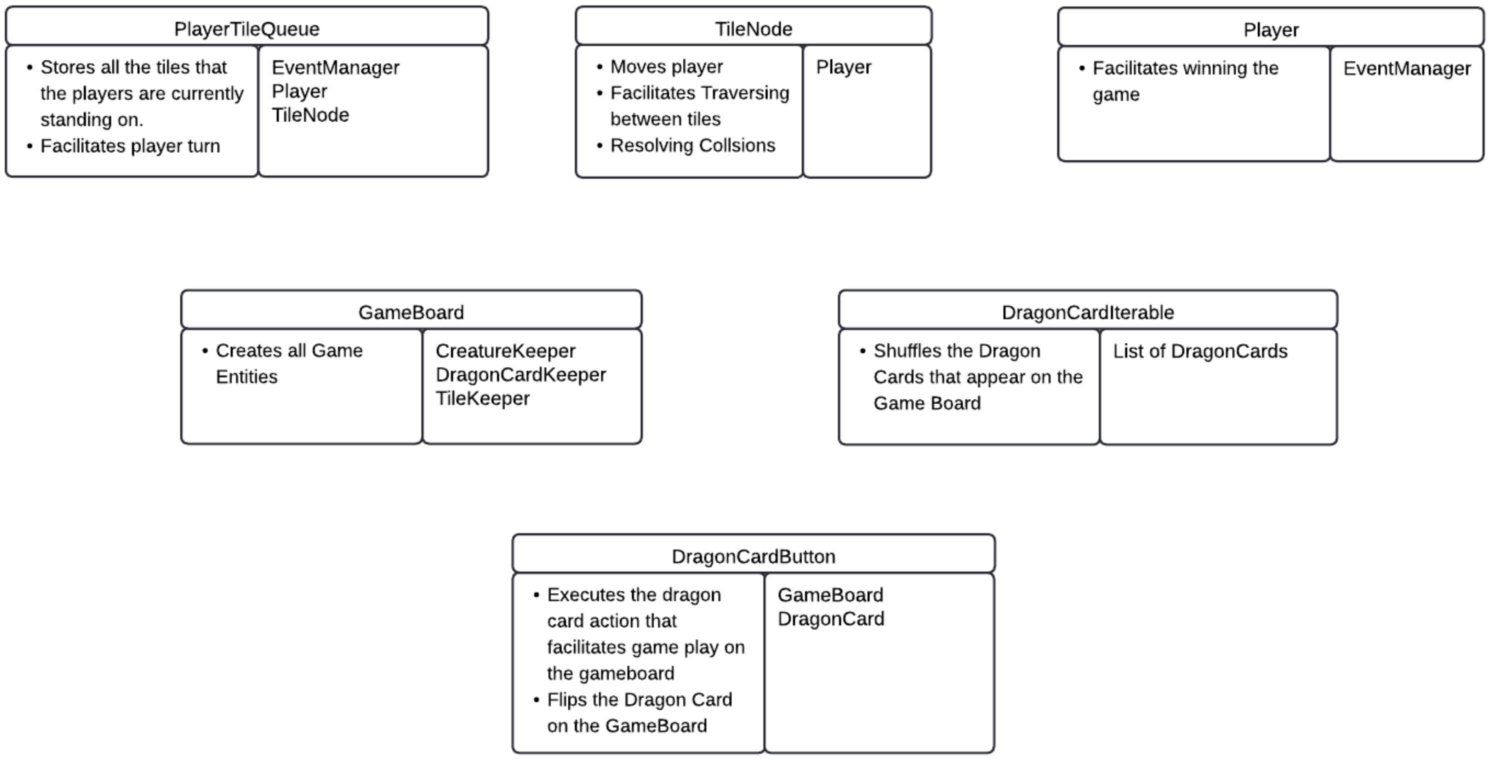


GameBoard

- Has information about the player manager.
- Has information about the tile manager
- Has information about the dragon card manager.

The first Draft of the CRC introduced GameBoard as a god class and falsely tried dividing responsibilities by creating manager classes. These manager classes would not help remove responsibilities but only dependencies. The only appropriate way of reducing responsibilities is by delegating those responsibilities to entities in the game like the Tile or the DragonCard. These changes were reflected on the new CRC after a massive overhaul of the final design.

Final CRC Diagram



PlayerTileQueue	
• Stores all the tiles that the players are currently standing on. • Facilitates player turn	EventManager Player TileNode

TileNode	
• Moves player • Facilitates Traversing between tiles • Resolving Collsions	Player

Player	
• Facilitates winning the game	EventManager

GameBoard	
• Creates all Game Entities	CreatureKeeper DragonCardKeeper TileKeeper

DragonCardIterable	
• Shuffles the Dragon Cards that appear on the Game Board	List of DragonCards

DragonCardButton	
• Executes the dragon card action that facilitates game play on the gameboard • Flips the Dragon Card on the GameBoard	GameBoard DragonCard

GameBoard

Acts as the main access to the backend of the game. Initialises the CreatureKeeper, DragonCardKeeper and the TileKeeper that in turn initialise all the different aspects of the game, helping create loosely coupled entities.
All major responsibilities of items under this entity are encapsulated within it.

DragonCardIterable

An interface to access all the dragon cards on the game board. Since it has access to the dragon card list, it facilitates shuffling the dragon cards.

DragonCardButton

The front-end wrapper class for the dragon card that is clickable. When pressed, it executes the main logic of the game by executing the dragon card command associated with the dragon card that was clicked.

TileNode

The graph node class that stores different tile types like the VolcanoTile and the CaveTIle. Only knows about the tiles adjacent to it. Facilitates player movement using recursion to access certain tiles that are at steps distance away from it.

Player

The entity that represents the interface for the user to interact with the game. The player only knows about how many total moves it has traversed. Uses the total moves to facilitate the feature of winning the game.

PlayerTileQueue

The player tile queue is a queue interface that helps us access and modify a list of tiles. These tiles always store the ones that the players are currently standing on. This helps us facilitate responsibilities like player turn and getting the player that won.

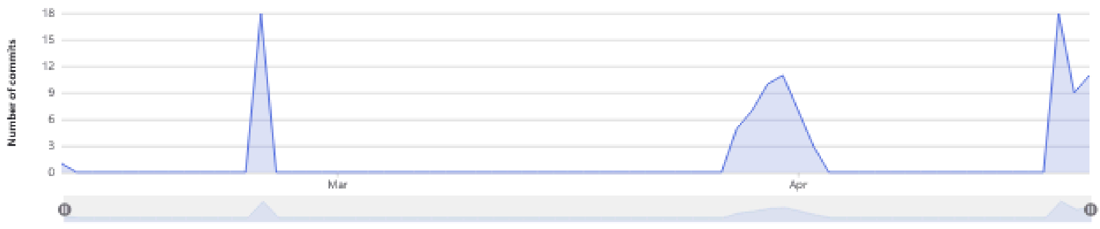
UML

https://lucid.app/lucidchart/154ee335-390f-4b02-8af0-826f0b462fab/edit?viewport_loc=-8500%2C-565%2C8959%2C4240%2C0_0&invitationId=inv_f3e4cee1-f9fb-4caf-a60c-02a96a0197c8

Contributor Analytics

Commits to master

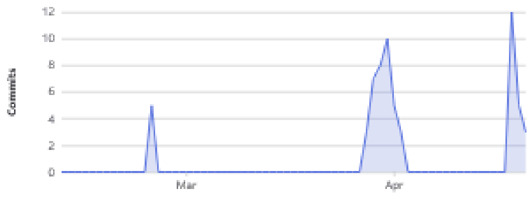
Excluding merge commits. Limited to 6,000 commits.



— **Commits** Avg: 1.47 · Max: 18

Fahad Assadi

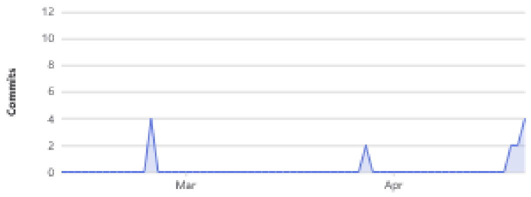
61 commits (fass0001@student.monash.edu)



— **Commits** Avg: 897m · Max: 12

umoh0005

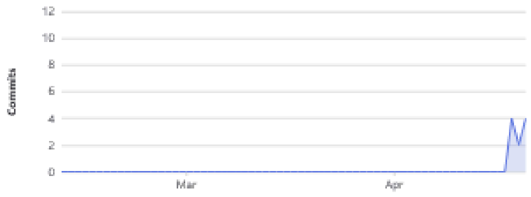
14 commits (umoh0005@student.monash.edu)



— **Commits** Avg: 206m · Max: 4

Hanif

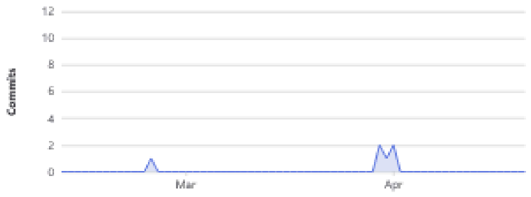
10 commits (hanifzafari@gmail.com)



— **Commits** Avg: 147m · Max: 4

fahad

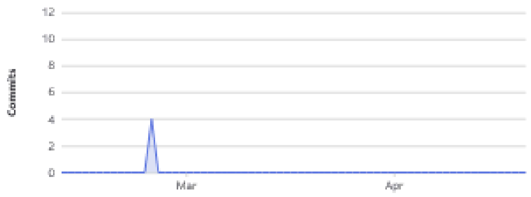
6 commits (91127049+fahadassadi@users.noreply.github.com)



— **Commits** Avg: 88.2m · Max: 2

aikr0002

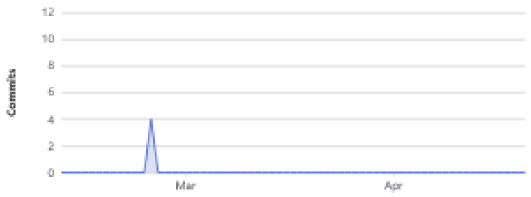
4 commits (aikr0002@student.monash.edu)



— **Commits** Avg: 58.8m · Max: 4

hanif zafari

4 commits (hmoh0035@student.monash.edu)



— **Commits** Avg: 58.8m · Max: 4