Francisco Zhou
Davit Barsamyan
Victor Zheng
Umair Hussain
Jazli Muhammad Khairi Leong
Clark Zhu
Claire Ramsumair

CSC207 Scrabble Project Documentation

Changes Since Milestone 4:

- Bug fixes for incorrect word played, displaying score, checking word validity, reset move bug
- Reduced dependencies
- Javadocs for all files
- New test cases for use cases and entities
- Created HandManager
- Remove print statements
- Removed unused code
- TileChecker tests
- Fixed inconsistent package naming
- Created EndGameManager
- MoveInfo became an entity
- Unhappy test flows for tests
- Template for pull requests
- Started application with Main instead of ScrabbleApp
- Fixed warnings
- Removed dependency in ScoringSystem
- Created HandCheck interface
- Refactored location of resources out of src
- Refactored package and class names to follow CA layers
- Moved CollinsWords text file to an outer layer
- Feature added involving loading a game that didn't exist
- Change method names for BoardManager
- Package names to follow snake case convention

- Interfaces are now inside use case packages
- Added test coverage report to resources in GitHub and ReadMe
- Generated UML Class Diagram

Workflow Changes:

- Implemented develop branch into workflow
    - Branched all new features from the develop branch
    - Once features accumulate we thoroughly test develop and make a pull request to main
    - Allowed us to tackle bugs early on and prevent them from ruining the final product
- Implemented project view into our workflow
    - Created an issues workflow view to keep track of issues and there progress
    - Allowed team to more effectively keep track of what needs to be done and what is already completed

PlayerManager: Use Case

Function:

PlayerManager takes care of mutations to the Player class. These include getting a Player's hand and updating the score attribute of a Player.

Design Pattern:

Follows mediator design pattern. The role of player manager is to allow other classes to interact with the Player entity without directly accessing said entity. This creates a centralized communication medium between different objects in the system. PlayerManager has its own functions to allow other objects to mutate the contents of Player. This design pattern reduces reliability and connections to the entity and serves as a mediator, hence the name.

SOLID:

Firstly the S, single responsibility principle. In previous implementations of our project PlayerManager had more than one task which included manipulating player hands and returning the winner of the game. However, after reviewing PlayerManager's other responsibilities have been stripped away into separate classes leaving it to be just a mediator for Player. Next O, this class follows the open-closed principle as each method within the class also has one purpose; they're all able to be extended without modifying the code. For example, updateScoreForCurrentPlayer, if there was a subclass for PlayerManager it would be able to extend this method without changing the implementation of the method in the parent class. Additionally, even if there was a subclass to the Player entity the function would still be able to be extended since it would still have its score attribute. Since PlayerManager does not have any subclasses Liskov's Substitution Principle does not apply. I: Interface Segregation as a group we have created many interfaces for specific purposes so a client does not have to implement an unnecessary interface. For this case specifically I have created the updateScore interface which contains a method allowing a player's score to be updated. D: Dependency Inversion Principle, PlayerManger used to be very dependent on other classes in the same layer of Clean Architecture. However, after refactoring we have removed any dependencies from the same layer.

Clean Architecture: PlayerManager is part of the use case layer of clean architecture. The methods are called by the ScrabbleGameController class which is in the controller layer of clean architecture and only interacts with the Player entity through the PlayerManager class.

HandManager: Use Case

Function:

Hand manager takes care of the draw tile use case. This use case is relevant after a player has played their move and needs to draw tiles to fill their hand back up. Thus, the purpose of this class is to mutate the player's hand and then mutate the LetterBag object being used for the game. This will simulate drawing tiles out of a finite bag.

Design Pattern:

The design pattern implemented for this was also the mediator. Since, this is another class that acts as a connection between the entity's attributes and the controller. This class makes it easy to mutate both the Player's hand attribute and the litterbug without accessing either directly from the Player and LetterBag class.

SOLID:

For SOLID, this class abides by the SRP as its only job is to act as a way to mutate a Player's hand and the LetterBag. Since, each method only has a single purpose, if there were subclasses of player, letter bag, or even HandManager the methods would still be able to be used. This is due to the implementation for the methods within HandManager not being very Hand Manager does not have any subclasses so Liskov's Substitution Principle does not apply. There is no dependency on other use cases within the class.

Clean Architecture:

HandManager is part of the use case layer within clean architecture. The methods are called by the ScrabbleGameController class which is in the controller layer of clean architecture and only interacts with the Player and LetterBag entity through the HandManager class.

EndGameManager: Use Case

Function:
EndGameManager takes care of the end game use case. This use case is important when the game of scrabble comes to an end. Specifically, it is used to determine the winner at the end of a game.

Design Pattern:
EndGameManager follows the Visitor design pattern. Instead of putting the functionality of EndGame within the Player Function we decided to split up the behaviour into a seperate class called EndGameManager. This class has the sole responsibility of determining which players won at the end of a Scrabble game. This design pattern allows it to just take an instance of Game and be able to determine the winner of the scrabble game.

SOLID:
The EndGameManager follows the Single Responsibility Principle as its only purpose is to determine the winner of a scrabble game. Objects of this class are open to extension, but closed for modification. Meaning the class can be extended without having to modify the class itself. EndGameManager currently has no subclasses. Next, interface segregation, our group decided to create an interface for each use case. This allowed our program to have many split up interfaces as use cases and keep them separate. EndGameManager does not depend on concretions since it calls methods within Player for its functionality. So even if, player changes its methods the functionality of EndGame should not change.

Clean Architecture:
Within the clean architecture layers EndGameManager is within the use case layer. The methods are called by the ScrabbleGameController class which is in the controller layer of clean architecture.

ScoringSystem: Use Case

Function:
ScoringSystem takes care of the scoring use case. This class is able to calculate the score of a word or multiple words. This is done at the end of a valid move by a player. Additionally it is also able to calculate the score of unplaced tiles given access to the player's hand. This is done at the end of a game of Scrabble.

Design Pattern:
ScoringSystem follows the visitor design pattern. This allows it to take the functionality of scoring tiles out of the Cell class and put it in a separate class. This way any change to Cell will not affect the scoring system method and they do not rely on each other. So if the Cell class had subclasses created the ScoringSystem would still be able to score the words given the setters and getter methods were still available.

SOLID:
The scoring system follows the single responsibility principle as its only purpose is to score cells. ScoringSystem has two interfaces one for scoring cells on the board and one for scoring cells off the board. Objects of this class are open to extension, but closed for modification. Since ScoringSystem does not do many mutations there is no problem when extending from this class. If there were subclasses created they can easily use the methods within ScoringSystem without any problems. ScoringSystem has no subclasses currently so the substitution principle does not apply. If any other class wants to implement these interfaces there is enough segregation to give the user the choice of which one they'd like to implement without being forced to implement a useless interface. Finally, the class used to depend on the PlayerManager and BoardManager class. After refactoring the class does not depend on these other classes in the use case layer.

Clean Architecture:
Within the clean architecture layers scoring system falls into the use case layer. The methods are called by the ScrabbleGameController class which is in the controller layer of clean architecture.

Player: Entity

Function:
The player entity holds the information regarding a Player. This class is instantiated at the beginning for each player of the game. The entity is responsible for keeping track of the player's name, score, and tiles in their hand.

Design Pattern:
The Player entity does not follow any design patterns per se due to it being relatively simple and not needing to be extended. The Factory design pattern was considered, but the main driving factor about this design pattern is the versatility it offers. Meaning you can instantiate a new object differently depending on where the method lies. However, since Player does not have subclasses nor is there any defining characteristic that would make a new subclass. We decided that there would not be much purpose in having a factory design pattern here.

SOLID:
The player class fulfills the single responsibility principle as the entire function of the class is to keep track of attributes regarding each player. The methods within Player are mainly getter and setter functions and if these methods needed to be extended it could be done without issue. Since these subclasses would have the same attributes along with their own, all these methods could still be used without modification. There are no subclasses for the Player class so the substitution principle does not apply. The Player entity does not have any interfaces so interface segregation does not apply. The class follows the dependency inversion principle since Player does not rely on any other classes for its methods. Every method within Player simply changes or retrieves an attribute of the Player class.

Clean Architecture:
Within the clean architecture layers the Player class falls into the entity layer.

GameSaverSystem: Gateway

Function:
GameSaverSystem has one function that is responsible for saving the game state of the game to a file named data.ser located in the data package. It implements the Serializable class and takes and saves the entity Game which in turn holds things like GameBoard, Players, Turn and LetterBag. I needed to save the game board, players scores and turn where we created that Game entity to save all of these into one. This file would then be loaded by another use case to load the same game state. The controller calls this at the end of every turn.

Design Pattern: Singleton, Originator
The role of the GameSaverSystem is to allow the data from the entities of Cell, GameBoard, Player and LetterBag to be saved by saving the entity Game which holds all of these. Therefore the class only has one instance and one responsibility. It controls access to the shared resource of a file and is the one class that has access to editing and creating this file. It also ensures that there is only one "data.ser" file instance that is saved. So new game states will be overwritten each time the saveGame method is called. This is also the Originator design pattern because it creates snapshots of its state and saves it to a file. It is the origin of where the snapshot is created. The ScrabbleGameController uses this to save the game state to this file every time it is used where this is the global access point to this use case. It uses the same static file path to the file each time it saves the game state.

SOLID:
Game
S: Single responsibility principle, the GameSaveSystem is only responsible for one responsibility. It only needs to save a game to a file using the Game entity. O: Open-Closed principle, for the GameSaveSystem each method within the class has one purpose and GameSaveSystem allows itself to be extended without modifying existing code. L: Liskov's Substitution Principle, as GameSaverSystem does not have any classes it applies Liskov's Substitution principle vacuously. I: Interface Segregation, There were several interfaces created for each use case and therefore GameSaverSystem implemented a GameSave interface with a

saveGame method to ensure that only GameSaverSystem was allowed to save games to a file. This is called by the ScrabbleGameController class which is the Controller layer of our project.
D: Dependency Inversion Principle, GameSaverSystem isn't dependent on any other class and is only called by the ScrabbleGameController when it needs to be saved.

Clean Architecture:
GameSaverSystem is part of the gateway and use case layer of clean architecture. The method is called by the controller, ScrabbleGameController of the project.

GameLoaderSystem: Gateway

Function:
GameLoaderSystem uses the “data.ser” file located in the data package and loads it as a Game object. As the Game entity implements serializable it can be easily loaded back into an object through this interface. Through the Game entity it will then save the saved game board, players’s hands, names and scores and as well as the game turn. This function is called when the user presses the “Load Game” in the startup screen in the GUI.

Design Pattern: Memento
The role of the GameLoaderSystem is to load data from the data.ser file and then load the Game object to in turn load the GameBoard, Player, LetterBag entities to a newly created game state. These would be the state snapshots of the game state where the GameLoaderSystem is the editor class. Storing these snapshots in a memento or our data file “data.ser”. Thus the GameLoaderSystem uses the design pattern of memento. ScrabbleGameController uses this loaded object/game state for the user to use.

SOLID:
S: Single responsibility principle, the GameLoadSystem is only responsible to load a game since it is only responsible for loading a game from the data.ser file.
O: Open-Closed principle, for the GameLoadSystem only one method exists within the class which also only has one purpose and therefore allows itself to be extended. It doesn’t need to be modified with existing code.
L: Liskov’s Substitution Principle, as GameLoadSystem is only responsible for loading a game it also does not have any subclasses and passes Liskov’s Substitution Principle vacuously.
I: Interface Segregation, GameLoadSystem only has one interface which is GameLoad since we needed to separate interfaces by use cases where the GameLoad interface has one method which is load game. This ensures only one class/use case is allowed to load a game.
D: Dependency Inversion Principle, GameLoadSystem isn’t dependent on any other classes since it only loads a game from a file which ScrabbleGameController calls.

Clean Architecture:

GameLoadSystem is part of the gateway and use case layer of clean architecture. The method is called by the ScrabbleGameController class which is the Controller layer of our project.

GameCreator: Use Case

Function:
GameCreator is responsible for creating a new Game instance or using a given previously loaded game from the file located in the data package. This was loaded into GameCreator as a Game object. The ScrabbleGameController calls this method and its use case to create a game which will load in an array of strings to add to the game. This will fully initialize the newly created game. This use case is called when the user presses the "Create Game" button in the startup screen of the GUI.

Design Pattern: Factory Method
The role of the GameCreator is to create an instance of Game whether by creating a completely new instance or loading back an old instance that was already made from the same data.ser file. So the GameCreator uses the Factory Method design pattern which uses the createGame method which is the special factory method which passes in an array of Strings to give the game. The product that is given is a fully instantiated Game with all of its players correctly added and ready to be used.

SOLID:
S: Single responsibility principle, the GameCreator is only responsible for implementing or performing one use case and that is to create a game and therefore the methods within this class also adhere to this because it is only responsible for creating a game. This could also create a game from a previously loaded game object.
O: Open-Closed principle, the GameCreator method has a method that is only responsible for creating a game and therefore only has one purpose and therefore allows itself to be extended without needing to modify existing code.
L: Liskov's Substitution Principle, as GameCreator is only responsible for loading a game it also does not have any subclasses and passes Liskov's Substitution Principle vacuously.
I: Interface Segregation, the GameCreator implements both the CreateGame interface so that no other class can also create a game. This interface has one method createNewGame which ensures that only one class/use case implementer i.e. GameCreator is allowed to create a new game.

D: Dependency Inversion Principle, GameCreator is responsible for creating a new game which can either use a loaded game object from loaded game.

Clean Architecture:
GameCreator is part of the Use Case layer of clean architecture. The methods are called by ScrabbleGameController which is the controller of the project to enact the use case of “create a game”.

Cell: Entity

Function:

The Cell entity holds the information regarding a cell. This is responsible for being an empty tile on the board plus a real letter tile. This holds properties such as score, value/letter, and multiplier. For an empty tile the value/letter is a string "-" to indicate empty. It is instantiated when creating the game board and for the player's hand.

Design Pattern:

The cell entity does not follow any design patterns per se due to it being relatively simple and not needing to be extended. The Factory design pattern was considered, but the main driving factor about this design pattern is the versatility it offers. Meaning you can instantiate a new object differently depending on where the method lies. However, since the cell does not have subclasses nor is there any defining characteristic that would make a new subclass. We decided that there would not be much purpose in having a factory design pattern here.

SOLID:

The cell class fulfills the single responsibility principle as the entire function of the class is to keep track of attributes regarding each tile. The methods within Cell are only getter and setter functions and if these methods needed to be extended it could be done without issue. Since these subclasses would have the same attributes along with their own, all these methods could still be used without modification. There are no subclasses for the Cell class so the substitution principle does not apply. The Cell entity does not have any interfaces so interface segregation does not apply. The class follows the dependency inversion principle since Cell does not rely on any other classes for its methods. Every method within Cell simply changes or retrieves an attribute of the Cell class.

Clean Architecture:

Within the clean architecture layers the Cell class falls into the entity layer. Which is used by use cases within BoardManager and HandManager.

GameBoard: Entity

Function:
The GameBoard entity holds the information regarding the game board being used. It is responsible for holding all the tiles on the board of the game. This array is made up of empty Cell entities for blank tiles and for letter tiles uses the “value” instance variable to insert its letter. An empty blank tile uses string “-” to be empty. It is instantiated when a new Game instance is created where a blank game board is created. This game board is saved within the Game entity which also holds the rest of the game information such as players and letterbag.

Design Pattern:
The GameBoard entity does not follow any design patterns per se due to it being relatively simple and not needing to be extended. The Factory design pattern was considered, but the main driving factor about this design pattern is the versatility it offers. Meaning you can instantiate a new object differently depending on where the method lies. However, since the game board does not have subclasses nor is there any defining characteristic that would make a new subclass. We decided that there would not be much purpose in having a factory design pattern here.

SOLID:
The GameBoard class fulfills the single responsibility principle as the entire function of the class is to keep track of attributes regarding each player. The methods within GameBoard are only getter and setter functions and if these methods needed to be extended it could be done without issue. Since these subclasses would have the same attributes along with their own, all these methods could still be used without modification. There are no subclasses for the GameBoard class so the substitution principle does not apply. The GameBoard entity does not have any interfaces so interface segregation does not apply. The class follows the dependency inversion principle since GameBoard does not rely on any other classes for its methods. Every method within GameBoard simply changes or retrieves an attribute of the GameBoard class. Though one method exists named isEmpty to check each tile to see if they are blank tiles.

Clean Architecture:

Within the clean architecture layers the GameBoard class falls into the entity layer. Which is used by use cases within BoardManager.

LetterBag: Entity

Function:
The LetterBag entity holds all the information regarding the bag of tiles being used. It is responsible for holding all the other tiles that currently exist in the game that are not in a player's hand or on the game board itself. This entity would be edited when the player draws a tile out of the bag or when the player shuffles their whole hand into the bag to draw out of it. This LetterBag entity is held within the Game entity so that this entity can save all of the entities at once. In the case of a load game use case, that same letter bag would be used to ensure that no duplicate letters exist in the game.

Design Pattern:
The LetterBag entity does not follow any design patterns per se due to it being relatively simple and not needing to be extended. The Factory design pattern was considered, but the main driving factor about this design pattern is the versatility it offers. Meaning you can instantiate a new object differently depending on where the method lies. However, since LetterBag does not have subclasses nor is there any defining characteristic that would make a new subclass. We decided that there would not be much purpose in having a factory design pattern here.

SOLID:
The LetterBag class fulfills the single responsibility principle as the entire function of the class is to keep track of attributes regarding each player. The methods within LetterBag are only getter and adder functions and if these methods needed to be extended it could be done without issue. Since these subclasses would have the same attributes along with their own, all these methods could still be used without modification. There are no subclasses for the LetterBag class so the substitution principle does not apply. The LetterBag entity does not have any interfaces so interface segregation does not apply. The class follows the dependency inversion principle since LetterBag does not rely on any other classes for its methods. Every method within LetterBag simply changes or retrieves an attribute of the LetterBag class. The HandManager implements methods/use cases that will edit the LetterBag when tiles are drawn in and out of it.

Clean Architecture:

Within the clean architecture layers the LetterBag class falls into the entity layer. Which is used by use cases/methods within HandManager.

MoveInfo: Entity

Function:

The MoveInfo class is responsible for storing moves made during a turn. This is specifically useful for the BoardManager as it allows it to keep track of the users move. MoveInfo contains the coordinates of a letter wanting to be placed. With this info it can be passed to other classes or be used to look up the gameboard with the stored coordinates.

Design Pattern:

The MoveInfo entity does not follow any design patterns per se due to it being relatively simple and not needing to be extended. The Factory design pattern was considered, but the main driving factor about this design pattern is the versatility it offers. Meaning you can instantiate a new object differently depending on where the method lies. However, since MoveInfo does not have subclasses nor is there any defining characteristic that would make a new subclass. We decided that there would not be much purpose in having a factory design pattern here.

SOLID:

The MoveInfo class fulfills the single responsibility principle as the entire function of the class is to keep track of attributes regarding each player. The methods within MoveInfo are only getter and adder functions and if these methods needed to be extended it could be done without issue. Since these subclasses would have the same attributes along with their own, all these methods could still be used without modification. There are no subclasses for the MoveInfo class so the substitution principle does not apply. The MoveInfo entity does not have any interfaces so interface segregation does not apply. The class follows the dependency inversion principle since MoveInfo does not rely on any other classes for its methods. Every method within MoveInfo simply changes or retrieves an attribute of the MoveInfo class.

Clean Architecture:

Within the clean architecture layers the MoveInfo class falls into the entity layer. Which is used by use cases/methods within BoardManager.

TileChecker: Use Case

Function:

TileCheck only has one function which is to check for the validity of a set of inputs from Cells. The methods within the class will check for adjacency, alignment, and boundaries before generating a list of coordinates for ScrabbleDictionary to check the correctness of the words. It will then validate the move for BoardManager to update the board.

Design Pattern:

The design pattern of TileChecker is to allow the data from entities of cell and gameboard to be used by another use case BoardManager. The sole role of this class is to determine whether a combination of Cells inputs are valid and communicate it with the BoardManager. The validity of letters are checked through adjacency of the tiles and generates a list of coordinates. This allows the BoardManager to be untouched before the inputs are validated for updating.

SOLID:

This class adheres to the SRP as its only responsibility is to check the validity of the coordinates from the Cell entity (new word). All the methods within the class only have one purpose, so any subclasses would not affect the functionality of the methods. Since there aren't any subclasses, it adheres to the LSP by default. There is no dependency on other usecases for this class, but other classes, such as Boardmanager and ScrabbleDictionary, will use this class. This functionality determines the new word that has been placed down and any other adjacent words to the new word and returns a nested list of coordinates of letters that needs to be checked for points. The class which the function is in only has one responsibility, which is checking the validity of the new word, and interacts with GameBoard and ScrabbleDictionary, and therefore adheres to the SOLID principles.

Clean Architecture:

Tilechecker is part of the Use Case layer of clean architecture. The methods are called by BoardManager which is a use case that manages the state of the board.

BoardManager: Use Case

Function:
BoardManager takes care of mutations to the GameBoard entity class. These include updating the String letter values of the cells on the board. It does this by implementing smaller use cases such as PlaceTile, PlaceWord and ResetMove which perform the specific tasks of changing the values of the cells on the board based on the different user inputs received from the controller class.

Design Pattern:
BoardManager follows the mediator design pattern since it directs communication between other objects. The role of board manager is to allow the controller and other classes to interact with the GameBoard entity without directly accessing the entity and violating clean architecture. BoardManager has its own getter and setter functions that allow other objects to mutate the GameBoard entity.

SOLID:
Starting with the "S", the single responsibility of BoardManager is to update and make changes to the values of the GameBoard entity. Although the BoardManager also implements other smaller use case methods such as PlaceTile, PlaceWord and ResetMoves, these also only update the GameBoard values so they still follow the single responsibility principle. The BoardManager also follows the open-closed principle since it's open for extension but not modification. Since our code doesn't use superclasses and subclasses, this extension isn't used and also the Liskov's Substitution principle doesn't apply to BoardManager either. The interface segregation principle states that the class shouldn't depend on interfaces that it doesn't need, and all interfaces implemented by BoardManager are useful for managing the GameBoard entity. The BoardManager class implements PlaceTile, PlaceWord and ResetMove which are all essential use cases when the user interacts with the board so BoardManager follows the interface segregation principle. Finally, for the "D" in SOLID, BoardManager does not depend on other classes, rather it implements three abstract interfaces.

Clean Architecture:

BoardManager is part of the use case layer of clean architecture. The methods are called by the ScrabbleGameController class which is in the controller layer of clean architecture and only interacts with the GameBoard entity through the BoardManager class.

Gamepage: View

Function: The GamePage UI displays the current scrabble game state of the game. It has buttons, labels, and text fields that display information from the current data from the game board. It also has an update view method that updates the UI following every turn.

Design Pattern: No design patterns were used for

S: SRP is followed since there is only a single responsibility of the class: to display the state of the game after each move.

O: OCP doesn't really apply because they are not software entities but rather display data.

L: LSP doesn't apply since it doesn't use subtypes so there is no substitution involved.

I: ISP is followed since the page is different than other pages, allowing easier modification and extension of the view

D: DIP is followed since it allows someone to change the view of the game without changing the rest of the game and the other pages.

Clean Architecture: The GamePage displays data but doesn't change data, allowing business rules to be followed. It is located on the UI/View layer.

EndGamePage, NewGamePage, RulesPage, StartUpPage: View

Function: These classes' focus is on displaying data from buttons and starting new pages. They call on components to create buttons, text fields, and labels for the user to view.

Design Pattern: These classes do not have any design patterns because of the simplicity of them.

SOLID: These classes follow SOLID principles since they only depend on the components and otherwise will run. They have single responsibilities of creating and displaying pages, and will not cause other pages to break if they break.

Clean Architecture: These classes are located on the UI layer. They follow CA since they allow the details on the outer layers to not affect the business layers inside. They display data but don't change data.

Button, DialogueBox, Panel, Label, and TextField: View

Function: These 5 classes are responsible for creating their respective buttons, dialogue boxes, panels, labels, and text fields from the JavaSwing library. They allow for customization of buttons, labels, and text.

Design Pattern: these components don't follow any design patterns since they are made to simply build components. We didn't find any problems with the implementation so we didn't invest time into building a design pattern for this.

S: Single Responsibility Principle: Each class is only responsible for creating each component and maintaining the contents of that component. This follows SRP because every class only has a single responsibility. In addition, it means each class should only have one reason to change.

O: Open-Closed principle, each component only has one method within each class, allowing it to to be extended with new plugin behaviours.

L: Liskov's Substitution Principle: this does not apply since these components are not subtypes of anything.

I: Interface Segregation: the components follow ISP because it splits up irrelevant elements of the View into specific classes that are easier to extend and modify design. It also allows for customization within each component without having the client implementing components every time.

D: Dependency Inversion Principle: this follows DIP because we can change individual components without having to worry about anything else. These are low level models that don't depend on details.

Clean Architecture:

Within the clean architecture layers the components classes fall into the UI layer. Which is called by pages that create instances of the components after each action click on the same layer. It allows for changes in outer layers to not affect the business rules of the project.

ScrabbleDictionary: Use Case

Function:
ScrabbleDictionary constructs a searchable hashmap of valid scrabble words, and determines whether or not a list of given coordinates correspond to words existing in this dictionary.

Design Pattern:
The singleton design pattern was considered while creating the ScrabbleDictionary class, as the hashmap only needs to be created once, but ultimately making the dictionary static resulted in conflicts with other classes.

SOLID:
The ScrabbleDictionary function slightly violates the single responsibility principle, as in addition to constructing and searching the dictionary, it also needs to parse words from a given list of coordinates. This is something that could be fixed by moving the wordparser method from the dictionary class to the TileChecker class and changing the inputs for the indictionary function to accept only a list of words. In doing so, ScrabbleDictionary would not need to interact with the GameBoard, and would not be responsible for parsing words. Otherwise, it adheres to SOLID principles. The ScrabbleDictionary can be extended without having to modify much code, as it primarily interacts with itself. ScrabbleDictionary has no subclasses and thus passes Liskov's Substitution Principle, It also implements no classes, and thus does not depend on interfaces it does not use. Since ScrabbleDictionary does not rely on other classes for its methods, it also follows the Dependency Inversion principle.

Clean Architecture:
ScrabbleDictionary is part of the Use Case layer of Clean Architecture

ScrabbleGameController: Controller

Function:
The ScrabbleGameController activates the correct use cases when specific UI elements are interacted with.

Design Patterns:
The controller itself acts as a mediator because all communication between the UI and the rest of the program is directed through this class. It is also possible to think about this as the controller taking advantage of the facades that each use case provides.

SOLID:

This class adheres to the single responsibility principle because the only reason for the class to change is modify the behaviour of UI elements.

This class adheres to the open-close principle because new behaviours can be added by modifying the use cases without modifying the controller class itself.

This class adheres to the Liskov substitution principle because it does not have any subclasses, so it obeys by default.

This class adheres to the interface segregation principle because it does not implement an interface because it communicates with the view directly.

This class adheres to dependency inversion because it uses only functions from the use case interfaces provided rather than the functions from the use cases themselves.

Clean Architecture:
This class is part of the Controller layer of clean architecture. It does not interact with the entities other than through the use cases.

TurnManager: Use Case

Function: Increments the turn counter when a turn is over

Design Patterns: There was really no need for a design pattern as the goal was quite straightforward

SOLID:
This class adheres to the single responsibility principle because the only reason for the class to change is to modify the behaviour of the turn counter.

This class adheres to the open-close principle because new behaviours involving the turn counter can be created by implementing the IncrementTurnCounterUsecase interface.

This class adheres to the Liskov substitution principle because it does not have any subclasses, so it obeys by default.

This class adheres to the interface segregation principle because the interface it implements only has the function that is required, which is IncrementTurn.

This class adheres to dependency inversion because it implements an interface that the controller uses.