Entities & Responsibilities

GameWorld: Manages game initialisation & admin tasks (e.g game config, game setup, main game loop, player turn.)

Tile: Represents the tiles which the dragons will interact with (stand on)

PlayableEntity: Represents the playable entity a player interacts with

GameBoard: Represents the game board. It runs the interactions with the game board by the players (e.g performing movement, flipping chit card)

ChitCard: Represents the chit cards and their effects

EventBus: Handles registration of listeners, and notification of appropriate listeners on event fire

WinEventXxxx....: Publisher and listeners for win event

MoveActionXxxx....: Publisher and listeners for a move action (for characters) that is fired

DrawableByAsset: Indicates that the object is drawable by pygame using assets

DrawAssetInstruction: A data class for organising data required for drawing an asset

ModularClickableSprite: Allows classes to be represented as a sprite that is clickable on a screen.

TileFactory, DefaultTileFactory, TileId: Tile abstract factory interface, concrete tile abstract factory that serves the regular form of tiles. TileId uniquely identify tiles to create

Patterns Used

Observer: WinEventPublisher, WinEventListener
- Why?: Don't have to check all starting tiles to see if win occured. Allows for wins from other sources

Singleton: EventBus
- Why?: Should be one central event bus managing all events

Abstract Factory: TileFactory
- Why?: Prevent hard dependencies in DefaultGameBoard and other gameboards. DIP

Todo

Cardinalities

Starting tiles must be winning tiles for DefaultGameBoard (need to make the typing more strong, extra class inheriting from Tile [WinnableTile])

Notes

Upcasts are safe
Circular dependencies = too many responsibilities
https://softwareengineering.stackexchange.com/questions/306483/how-to-solve-circular-dependency
Java supports circular dependencies
https://www.reddit.com/r/ProgrammingLanguages/comments/yvkysh/languages_which_support_circular_dependency/

<<enum>>
Animal
.salamander
.baby_dragon
.spider
.bat

AnimalChitCard
- animal: Animal
+ __init__((Int, (Int, Int), (Int, Int))
+ perform_effect(PlayableCharacter) : Void
+ get_draw_clickable_assets_instructions() : [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click() : Void

PirateChitCard
+ __init__((Int, (Int, Int), (Int, Int))
+ perform_effect(PlayableCharacter) : Void
+ get_draw_clickable_assets_instructions() : [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click() : Void

DefaultGameBoard.__init__: First parameter is the sequence of tiles as their identifiers. Second parameter is [(starting tile, tile # to connect to in first parameter, playable character to start in it)]. The second parameter (starting tiles) will be used to make insertions to main tile sequence property.
DefaultGameBoard.starting_tiles: [starting tiles]
DefaultGameBoard.move_character_by_steps: Moves a dragon from its tile by x number of steps
DefaultGameBoard.flip_chit_card: Flip a chit card on this game board. Chit card object will be obtained by click listener in pygame.
DefaultGameBoard.add_chit_card: Add a chit card to the game board

DefaultGameBoard
- main_tile_sequence: [Tile]
- starting_tiles: [Tile]
- chit_cards: [ChitCard] = []
+ __init__([TileId], [(Tileid, Int, PlayableCharacter)], TileFactory)
+ move_character_by_steps(PlayableCharacter, Int) : Void
+ get_character_floor_tile(PlayableCharacter) : Tile
+ flip_chit_card(PlayableCharacter, ChitCard) : Void
+ add_chit_card(ChitCard)
+ get_draw_assets_instructions() : [DrawAssetInstruction]

<<abstract>>
ChitCard
- flipped: Bool
- symbol_count: Int
- coordinates: (Int, Int)
- size: (Int, Int)
+ __init__((Int, (Int, Int), (Int, Int))
+ perform_effect(PlayableCharacter) : Void
+ get_draw_clickable_assets_instructions() : [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click() : Void
+ set_flipped(Bool) : Void

ChitCard.__init__: Create a chit card with the # number of symbols
ChitCard.perform_effect: Perform the chit card's effect on the dragon. Run by flipChitCard
ChitCard.set_flipped: Set flipped

<<abstract>>
ModularClickableSprite
- rect: Dictionary<pygame.Rect, ModularClickableSprite>
+ __init__()
+ get_draw_clickable_assets_instructions() : [(DrawAssetInstruction, ModularClickableSprite)]
+ on_click() : Void
+ update_click_hitboxes() : Void

ModularClickableSprite.__init__: Run update click hitboxes
ModularClickableSprite.rect: Dictionary mapping click hit boxes to the object associated with it
ModularClickableSprite.on_click: Do something to the object on click
ModularClickableSprite.update_click_hitboxes: Update click hitboxes based on draw instructions (get_draw_clickable_assets_instructions)

DrawAssetInstruction
- asset_path: String
- x: Int
- y: Int
- size: (Int, Int)
- rotate: Float
+ __init__(String, Int, Int, (Int, Int), Float)
+ get_asset_path(): String
+ get_x_coord(): Int
+ get_y_coord(): Int
+ get_width(): Int
+ get_height(): Int
+ get_rotation(): Float

<<enum>>
TileId
.normal
.cave

<<interface>>
TileFactory
+ create_normal_tile((Int,Int), (Int,Int), Animal, Character = None) : NormalTile
+ create_cave_tile((Int,Int), (Int,Int), Character = None) : CaveTile
+ createFromTileId(TileId): Tile

TileFactory.create_normal_tile: Create a NormalTile based on input (coordinates, size, animal, character (optional))
TileFactory.create_cave_tile: Create a CaveTile based on input (coordinates, size, character (optional))
TileFactory.create_from_tile_id: Create a Tile based on tile id

<<interface>>
DrawableByAsset
+ get_draw_assets_instructions() : [DrawAssetInstruction]

DrawableByAsset.get_draw_assets_instructions: Gets the drawing instructions for drawing an object onto the pygame screen using assets

<<abstract>>
GameBoard
+ move_character_by_steps(PlayableCharacter, Int) : Void
+ get_character_floor_tile(PlayableCharacter) : Tile
+ flip_chit_card(PlayableCharacter, ChitCard) : Void
+ add_chit_card(ChitCard): Void
+ get_draw_assets_instructions() : [DrawAssetInstruction]
+ on_move_action_fired(PlayableCharacter, Int) : Void

<<abstract>>
PlayableCharacter
+ __init__()
+ take_turn() : Void

Dragon
+ take_turn() : Void

Dragon.take_turn: Allows a dragon to take its turn & perform actions (e.g chit card flipping) by interacting with screen

GameWorld
- playable_characters: [PlayableCharacter]
- chit_card: [ChitCard]
- game_board: GameBoard
+ __init__([PlayableCharacter], [ChitCard], GameBoard)
+ run(params): Void
+ on_player_win(PlayableCharacter) : Void

DefaultTileFactory
+ createNormalTile((Int,Int), (Int,Int), Animal, Character = None) : NormalTile
+ createCaveTile((Int,Int), (Int,Int), Character = None) : CaveTile
+ createFromTileId(TileId): Tile

GameWorld.__init__: Configure and initialise playable characters, game board & its tiles & its chit cards.
GameWorld.run: Contains the main game loop. Handles policy of player turns, and drawing logic (based on drawable)

<<interface>>
MoveActionHandler
+ on_move_action_fired(PlayableCharacter, Int): Void

<<abstract>>
Tile
- on_tile: PlayableCharacter
- animal: Animal
- coordinates: (Int, Int)
- size: (Int, Int)
+ __init__((Int, Int), (Int, Int), Animal PlayableCharacter = None)
+ character_on_tile(): PlayableCharacter
+ get_animal_on_tile(): Animal
+ remove_character() : Void
+ place_character(PlayableCharacter) : Void
+ get_draw_assets_instructions() : [DrawAssetInstruction]

NormalTile
+ __init__((Int, Int), (Int, Int), Animal PlayableCharacter = None)
+ place_character(PlayableCharacter) : Void
+ get_draw_assets_instructions() : [DrawAssetInstruction]

CaveTile
+ __init__((Int, Int), (Int, Int), Animal PlayableCharacter = None)
+ place_character(PlayableCharacter) : Void
+ get_draw_assets_instructions() : [DrawAssetInstruction]

CaveTile.place_character: Place dragon onto tile and perform check whether it was return back to own cave, trigger win

Tile.__init__: Initialise tile with playable character (dragon) and animal
Tile.place_character: Place dragon onto tile and perform related functionalities
Tile remove_character: Remove the dragon that's on the tile (if any)
Tile.character_on_tile: Returns the dragon on the tile (if any)

MoveActionPublisher.notifySubscribers: Notify subscribers about a move action that needs to be handled. Character and steps to take passed in

MoveActionPublisher
- subscribers: [MoveActionHandler]
+ __init__()
+ subscribe(MoveActionHandler) : Void
+ unsubscribe(MoveActionHandler) : Void
+ notify_subscribers(PlayableCharacter, Int) : Void

EventBus
- win_event_publisher: WinEventPublisher
- move_action_publisher: MoveActionPublisher
- shared: EventBus
- __init__()
+ publish_win_event(PlayableCharacter) : Void
+ publish_move_action(PlayableCharacter, int): Void
+ add_listener(EventType) : Void
+ shared() : EventBus

EventBus.shared: Get the eventbus singleton
EventBus.publishWinEvent: Publish win event onto bus
EventBus.addListener: Add a listener for a certain event type

<<interface>>
WinEventListener
+ on_player_win(PlayableCharacter)

WinEventPublisher
- subscribers: [WinEventListener]
+ __init__()
+ subscribe(WinEventListener): Void
+ unsubscribe(WinEventListener) : Void
+ notify_subscribers(PlayableCharacter) : Void

WinEventListener.onPlayerWin: On a player win, do something

WinEventPublisher.notifySubscribers: Notify subscribers about the player who won

<<enum>>
EventType
.win
.move_action_fired