Design Example: A chess application

We'll be designing a chess application's model.

Models

Board A board maintaining a list of the pieces and their locations.
Piece A piece of the board. It knows what board it is in, and what its coordinates are.
Move Identifies a "move". It should contain information about "color", i.e. whose move it is, as well as what the move is, in the form of a piece object to move, and where to move it to.
Game The model representing an overall game. It must have a board associated with it, and keep track of a series of moves and whose turn it is.

A main decision is where the logic goes for knowing if a move is valid, if the king is in check, etc. We will put that logic mostly in the Game class.

Board

Needs to know about

- `Piece` class, as it is responsible for creating and managing the pieces.

Instance variables These are meant to be private, but we probably won't enforce it. Users should not access these directly

- `locations`: Stores the locations of pieces. We will treat it as an array of arrays, where `locations[i][j]` would return the piece at the `i,j` location, if there is one, or null if there isn't a piece there.

  Indexing on locations starts at 1 (so the arrays we will be using will have an empty entry at 0).

- `detached`: An array that stores all the pieces that have been "detached" i.e. captured, and are technically no longer part of the board.

Class Methods

- `new`: Creates a new board instance. Calls initialize.

Instance Methods

- initialize: Creates a set of pieces and initializes their position.
- set(piece, i, j): Places the piece at the i,j location. If the piece had a location prior to that, it "removes" it from that location.
- get(i, j): Returns the "contents" of the i,j location, namely a Piece or null.
- isClear(i, j): Simply get(i,j)===null.
- pieceIterator: Returns an iterator that traverses the pieces that are *on the board* one at a time.

Piece

Needs to know about Nothing else.

Instance Variables

- board: A piece maintains a link back to the board that created it. (Needed?)
- i, j: The current location of the piece. Valid numbers in the range 1 to 8, with 0 representing the idea that the piece is not placed yet.
- type: What type the piece is. Available types: "king", "queen", "rook", "bishop", "knight", "pawn".
- color: Whether it's a white or black piece.

Class Methods

- new(board, type, color): Creates a new piece of a given type and associated with a given board. The piece initially has no placement (i,j are 0).

Instance Methods

- moveTo(i, j): Sets a piece's location.
- detach: Detaches the piece (sets location to 0,0).
- isDetached: Whether the piece is "detached" from the board.
- canMoveTo(i, j): Determines if the piece can move from its current location to the destination location (i, j).

  This method only determines if this is a valid move for this kind of piece, and not whether there are other considerations that might prevent the move. For instance it will check that there are no obstacles along the way, but it will not check whether it is that color's turn to move, whether the king is in check and so on.

  This method is the reason pieces need to have access to their board, so they can ask it about the squares they would need to "go through".

This is also essentially the only method that depends on what type the piece is. We can either have a separate subclass for each piece, or we can dynamically set this method upon creation of the piece, based on what its type is. We will opt for this second approach.

- validDestinations(): Returns an array of all destinations of valid moves for that piece, in the form of objects `{ i: i, j: j }`.

Move

Moves will essentially implement the command pattern.

Needs to know about Nothing.

Instance Variables

- piece: Which piece is meant to be moved.
- i, j: Location of destination for the piece's move.
- capturedPiece: Some moves include a piece that will be "captured" by the move. It is stored here.

Class Methods

- new(piece, i, j, capturedPiece): Creates a new "move" for the piece to move to the location (i,j).
- castleShort, castleLong: Creates the special moves for castling.

Instance Methods

- isValid(): True if this is a "valid" move from the point of view of the piece under consideration.
- execute: "Perform" the move. Special consideration will need to be given to moves like "castling", which involve the movement of two pieces.
- unexecute: Reverses the move. The move needs to have enough information to reverse itself (provided it was the "last" move to be executed, or if we've already unexecuted all the moves that followed it).

Game

Needs to know about

- Board: To create the initial board.
- MovesList: To create the moves list.

Instance Variables

- whiteTurn: True if it is white's turn.
- moves: A MovesList instance.
- board: The board for the game.

Class Methods

- new: Creates a new game, initializing a new board and an empty set of moves, and setting it to be white's turn.

Instance Methods

- isValid(move): Determines if the suggested move would be a valid move for the current player.
- addMove(move): Adds the corresponding move to the moves list. Does not yet perform it. Should only happen after a isValid. Will remove any moves that are the "current" or following it.
- redo: Performs the "current" move (which may be a recently added move).
- undo: Undoes the lastly performed move.
- canRedo, canUndo: Whether those are possible steps.
- isInCheck: Determines if the player up next is currently in check or not.

MovesList

Maintains a list of "moves" with undo-redo functionality built in.

Needs to know about Nothing. If desired, we could use a separate DoubleLinkedList implementation, and adjust its interface. Or we could bake the idea of a double-linked list in it.

Instance Variables

- moves: Structure holding the moves. Probably a double-linked list.
- current: Points to the move that would be the next to be executed.

Class Methods

- new.

Instance Methods

- addMove(move): Adds a move at current, removing any moves that were following it.
- redo: Performs the "current" move (which may be a recently added move).
- undo: Undoes the lastly performed move.
- canRedo, canUndo: Whether those are possible steps.
- nextMove: The move that would be performed next (essentially current).
- prevMove: The last performed move.
- prevMovesIterator: An iterator that traverses all played moves starting from the most recent.
- gameIterator: An iterator that traverses the played moves starting from the first one and ending before current.
- unDoneMovesIterator: Traverses the moves that can be redone, starting with current.

Unresolved Issues

- How to determine if a castle move is allowed (e.g. if king or rook have moved at any point prior to now, move's not allowed). Should some information be kept somewhere, or should it be computed when needed?
- There is a special en passant move. It needs to be treated specially. How and where would that happen?
- Would our design allow us to change the "rules", board arrangement, etc? What modifications would I need to do to allow for that?
- We would like to generate moves from the standard chess notation, as well as produce that notation from moves (to say print a game). How and where would that go?