GL

Functional & Technical Requirements Document

Data Structures (CS-221) | WVUP | Fall 2016

glFreecell

groovyLlama DevTeam

Casey Murphy Brandon Wharton Ryan Whytsell Gordon Finnie

Table of Contents

1. PROJECT INFORMATION
   - 1.1. Purpose……3
   - 1.2. Scope……3
   - 1.3. Time-Table/Objectives……3
   - 1.4. Communication……4
   - 1.5. Collaboration……4
2. GAME DESIGN
   - 2.1. Game Rules……4
   - 2.2. Game Specifications……5
3. FUNCTIONAL REQUIREMENTS
   - 3.1. Object Types……6
   - 3.2. Object Interaction……6
4. TECHNICAL REQUIREMENTS
   - 4.1. Project Structure……7
   - 4.2. Data Structures……8
5. SOFTWARE
   - 5.1. Interfaces……9
     - 5.1.1. CLI……9
     - 5.1.2. GUI……10
   - 5.2. Integration……10
   - 5.3. License……10
6. DIAGRAMS
   - 6.1. Relational Diagram……11
   - 6.2. UML Class Diagrams……12

1. PROJECT INFORMATION

1.1 Purpose

For the final project in Data Structures, we are to work as a team to plan, develop, and fully realize a GUI-based Freecell game with all functionality one would normally see in such a game. Freecell will be written with Java 1.8+ and will utilize coding techniques covered throughout the course of this semester, including the implementation of customized data structures unique to Freecell and solving Freecell games.

1.2 Scope

Development progression will be reviewed weekly by our peers and professors in a classroom environment. The project will culminate in a presentation of our finished product to the class, school personnel, and members of the Parkersburg area development community.

1.3 Time-Table/Objectives

Week 1 –

- Plan the overall structure as a rough draft.
- Determine a list of objects that will be needed.
- Build initial diagrams.

Week 2 –

- Discuss refinements to the design and update diagrams accordingly.
- Build basic structures that will be needed (i.e. Card, Deck, Cell).
- Design interfaces (CLI, GUI).

Week 3 –

- Define mechanics (Engine, Turn, Board) and game rules.
- Build basic versions of game classes.
- Build a lightweight and functioning model of the design.

Week 4 –

- Make any necessary tweaks to the design.
- Complete game mechanics.
- Develop solver structures and algorithms.

Week 5 –

- Test and debug mechanics that are in place.
- Refine and tweak codebase for improvements in design and efficiency.
- Game fully playable in CLI and GUI.

Week 6 –

- Optimize solver algorithms for speed.
- Full game with complete interface integration.
- Update all documentation (JavaDocs) and diagrams.

1.4 Communication

We will meet in person as our schedules permit. Primarily, we will use our designated Slack channel as our means of communication throughout the project. For conference calls, we will use our designated Discord channel.

For task assignment and issue handling we will use the GitHub software repository framework. It will allow us to maintain real-time updates and tracking at each stage of development. If an issue arises, any team member has the ability to add, update, and/or close the issue. Utilizing this tool will greatly aid in development flow.

1.5 Collaboration

Our project is hosted on GitHub and has six branches. Master will house stable versions and will be updated when the team deems fit. Develop will house working versions that still have issues to be resolved. The other four branches are for each team member, respectively. Each team member will work from their branch. We will collaborate by pulling from one another as we see fit and to ensure that we are working from the same codebase. Develop is the branch designated to that end, but if multiple team members are working on a specific task they may choose to keep the work in their own branch until a stable point is reached to merge. Both Master and Develop will be free from execution errors to ensure a clean working/testing environment.

2. GAME DESIGN

2.1 Game Rules (adapted from: http://www.casteel.org/Pages/FreeCellrules.html)

As the game begins, glFreecell deals all the cards from the deck face up into the playing piles, starting from left to right, top to bottom.

The object of the game is to build all the cards face up on the home cells (top right). Each home cell builds upward, in sequence, starting with the Ace. Only aces may be moved to an empty home cell, and only the next higher card of the same suit can be added to the home cell.

Only one card at a time may be moved in free cell, either the top card of a playing pile (bottom columns) or the single card which was placed in a free cell. As a short cut, glFreecell allows you to move a legal sequence of cards from one column to another, provided there are enough empty cells to have made this move one card at a time. The move will be scored as if you had moved one card at a time.

Only the next higher card of the same suit can be added to each home cell. Cards may be moved to the home cells from the playing piles or from a free cell. Cards cannot be removed from a home cell.

Any card may be moved to an empty free cell or an empty playing pile.

A card may be added to a non-empty playing pile only if it is the next lower value than the current top card and of alternating color. For example, only the 9 of hearts or diamonds (red suits) may play on the 10 of spades (a black suit).

2.2 Game Specifications

- On the deal, glFreecell will automatically stack appropriate cards into the home cells (i.e. Aces on the top of a playing pile will stack into a home cell). For any amount auto-stacked, the turn increment for that action will be only one.
- If the next card in a home cell sequence is opened up, it will automatically be placed in the appropriate home cell and the turn will be incremented once per card stacked.
- Double-clicking a card will place it in its home cell if it is valid. If it cannot go into a home cell, the action will place the card into the first available free cell.
- Card sequences can be pulled from a playing pile if there are available slots to make the sequenced move (i.e. moving a group 9, 8, 7 to a pile with an alt-color 10 on top).
- The 'New Deal' button deals a new hand of glFreecell and resets engine statistics. Can be done at any time.
- The 'Undo' button sets the board to the state it was in one move ago. If there are no moves to undo, the button is grayed out (GUI) or the option is unavailable (CLI).
- The 'Redo' button sets the board to the state it was in one move ahead. If there are no moves to redo, the button is grayed out (GUI) or the option is unavailable (CLI).
- The 'Hint' button will highlight all available moves from that specific turn. For instance, if a free cell is open, then all top cards in the playing piles will be highlighted.
- The 'Solve' button will initiate program generated movement on the board towards the calculated solution.
- All resources (images, sounds, artwork, notes) can be found in the project folder under /resources/.

3. FUNCTIONAL REQUIREMENTS

3.1 Object Types

- Card – Playing card
- Deck – Standard and Easy Win
- Board – Playing board
- Free Cell – Top left region
- Home Cell – Top right region
- Playing Pile – Bottom region
- Key – Region/Element designation
- KeyMap – Mapping of a card placement within the course of a turn
- Game Engine – Drives the game, contains main loop
- Game Turn – One valid card movement/placement
- Solver Engine – Drives the solver
- Solver Pattern – Pattern and priority definitions
- Solver Search – Logic used for pattern matching
- CLI Client – Command line interface
- GUI Client – Graphical user interface
- System Utilities – Game independent utilities (OS compatibility, generic structures)

3.2 Object Interaction

For a detailed layout of object relationships please refer to section 6.
This is a basic depiction of the main objects used in each break down.

main →	GUI, CLI
GUI →	Engine
CLI →	Engine, GUI
Engine →	Board, KeyMap, Turn
Turn →	Board, KeyMap, Solver
Solver →	Solver Engine, Search, Pattern, Solution
Board →	Deck, Free Cell array, Home Cell array, Playing Pile array
Deck →	Card array
Free Cell →	Card, Key
Home Cell →	Card, Key
Playing Pile →	Card array, Key
KeyMap →	Source Key, Destination Key, Board

4. TECHNICAL REQUIREMENTS

4.1 Project Structure

glFreecell includes six top-level packages and two sub packages (in client):

- board
  - Board – game board; central to all other components in this package
  - CellInterface – interface for cell objects
  - FreeCell – top leftmost of board
  - HomeCell – top rightmost of board
  - PlayingPile – columned piles
  - Key – key enumeration; assigned to each element on the board
  - KeyMap – maps source and destination for a move
- client
  - cli
    - CLI – command line interface
    - Debugger – toggles all debug switches
    - Tester – runs all class unit tests and holds customized tests
  - gui
    - FreeGUI – graphical user interface
  - Driver – entry point into program
- engine
  - Engine – drives the game; houses main game loop
  - Turn – represents one turn as the successful movement of a single card
- playingCards
  - CardInterface – interface for card objects
  - StdCard – standard playing card
  - DeckInterface – interface for deck objects
  - StdDeck – standard deck of 52 playing cards (no jokers)
  - DeckException – for error handling of a deck object
- solver
  - Pattern – solver algorithms
  - Search – search pattern
  - Engine – drives solver through turns with no human interaction
  - Solution – result of the solver, attached to the turn
- utils
  - SysUtils – access to system utilities such as path building to game resources (images, sound effects, etc)

4.2 Data Structures

- Playing cards are generated based on several in-class arrays of card values and card suits.
- A deck is generated by passing looped modulus expressions into the playing card constructor.
- The game board contains a deck and mapped regions designating each free cell, each home cell, and each playing pile with a generated key. The free cell, home cell, and playing pile regions are fixed-size arrays of their respective type.
- A free cell can contain one card or no card.
- A home cell can contain one card or no card. As each subsequent card in the sequence is added to the home cell it becomes the top card and the underlying cards are no longer accessible. They can be retrieved within the turn history. As cards cannot be removed from a home cell, nothing is lost in utilizing this method.
- A playing pile contains an ArrayList of the cards currently in that pile. As the cards need to be accessed by index when the board is painted/printed, the ArrayList provides a functional mechanism for working within the given constraints. This structure is tentative and may be changed for resource optimization.
- A key is an enumeration that holds the key value (alphabetic mapping of a – p), the associated region value (1 - freecell, 2 - homecell, or 3 - playing pile), and the element value of that region (regions are arrays of the associated type). When the board is initialized, keys are generated and assigned to each component on the board.
- A keymap hold two keys, designated the source key and the destination key. It also holds a copy of the current board. It can be used to validate a move, to access the card being moved (source), or the state of the destination key. The keymap will be used in the solver as well to facilitate lightweight manipulation of the board when searching for a solution.
- The engine holds the current board, turn history, and move number (controlled by the game loop) as primary members. The source and destination mapping (keymap) for a played turn is validated and then passed into the turn to be processed. The turn history is a stack of turns corresponding to each move played along with its associated board object.
- The turn holds the game board that is passed in from the main loop with its move number and keymap. The turn can only take a valid keymap, so the keymap must be checked before the turn is created. Turn will also hold a solution object that will be generated by the solver when the turn is created.
- Once the solver is in place, the turn will run a check upon instantiation to determine if the layout is winnable. The solution and winnable status will be associated with the turn so that if the move is undone and redone, turn does not have to recalculate a solution.
- Tentatively, when a solver is requested by the turn, it will initialize by creating a keymap array of possible moves. Each element will then become the base node of a structure (to be determined) where defined, prioritized patterns will be used to search for a solution path.

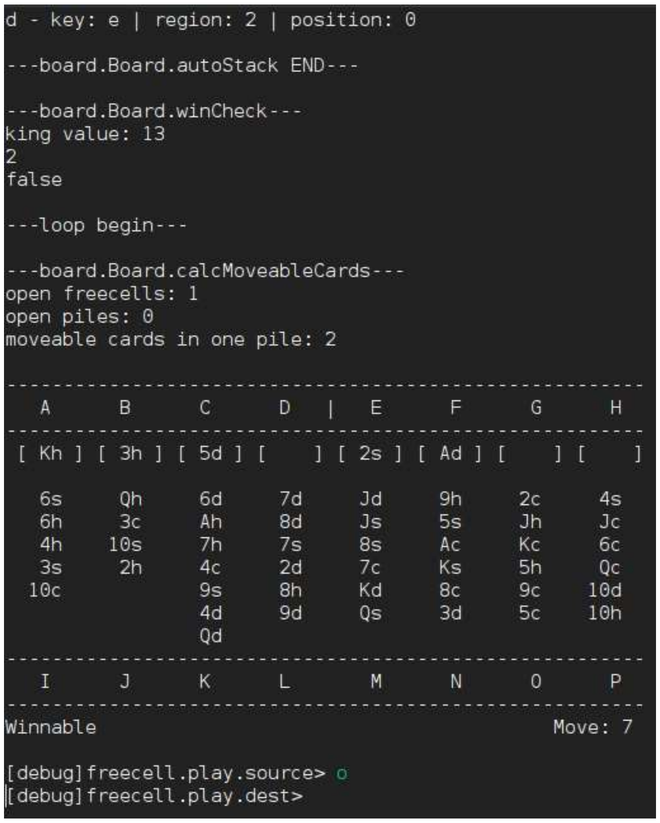
5. SOFTWARE

5.1 Interfaces

glFreecell has two types of interface. The GUI will be the production version default for execution. The CLI is accessed through the '-t' or '—test' command-line option. For the purposes of our final presentation, we will demonstrate both interfaces.

5.1.1 CLI

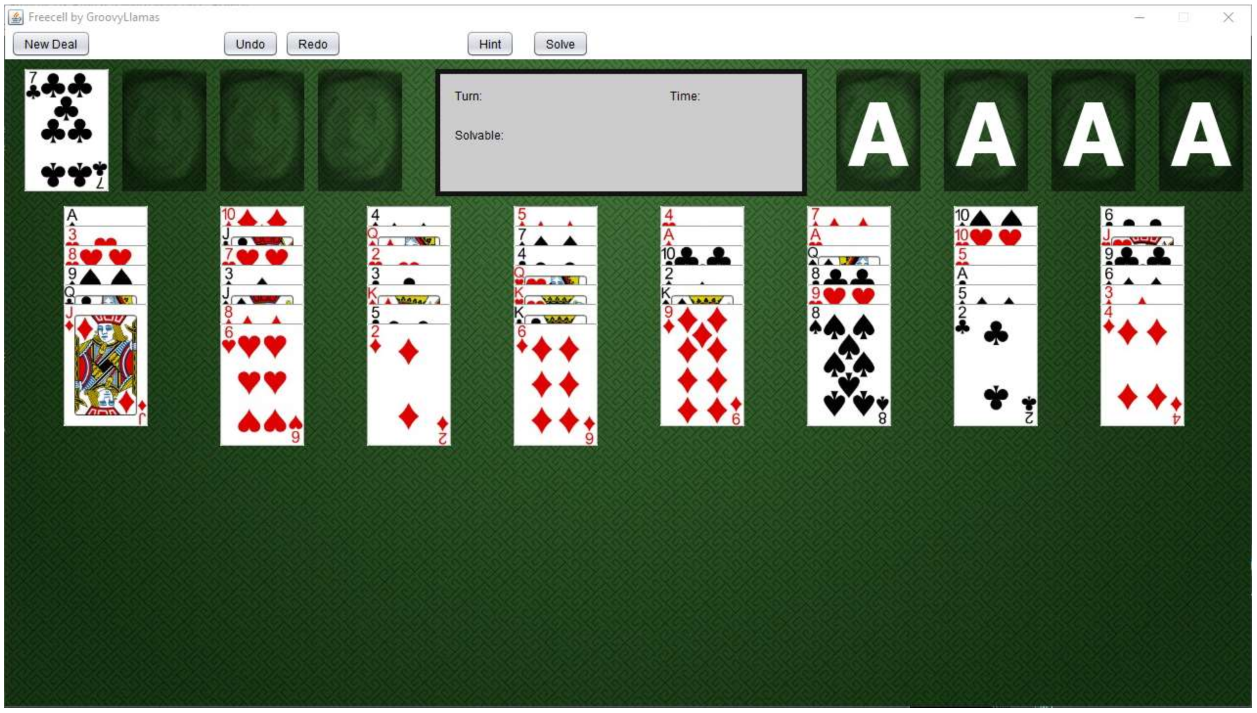
A playable, command-line version of the game will be accessible through an in-game command prompt. This mode offers an option to start a game in GUI or CLI. An integrated testing and debug framework is built into the program so that execution steps and debug statements may be observed and logged while playing/testing. Individualized tests accompany most classes to verify accuracy and stability of that particular unit. These tests can be accessed with the 'test' command while in CLI.



```
d - key: e | region: 2 | position: 0

---board.Board.autoStack END---

---board.Board.winCheck---
king value: 13
2
false

---loop begin---

---board.Board.calcMoveableCards---
open freecells: 1
open piles: 0
moveable cards in one pile: 2

--------------------------------------------------------
   A      B      C      D   |   E      F      G      H
--------------------------------------------------------
 [ Kh ] [ 3h ] [ 5d ] [    ] [ 2s ] [ Ad ] [    ] [    ]

   6s     Qh     6d     7d     Jd     9h     2c     4s
   6h     3c     Ah     8d     Js     5s     Jh     Jc
   4h    10s     7h     7s     8s     Ac     Kc     6c
   3s     2h     4c     2d     7c     Ks     5h     Qc
  10c            9s     8h     Kd     8c     9c    10d
                 4d     9d     Qs     3d     5c    10h
                 Qd
--------------------------------------------------------
   I      J      K      L       M      N      O      P
--------------------------------------------------------
Winnable                                        Move: 7

[debug]freecell.play.source> o
[debug]freecell.play.dest>
```

5.1.2 GUI

A playable Freecell game will be dealt and await user input on program execution. If started from the CLI in debug mode, the debug statements will print to track program execution while in the GUI.



5.2 Integration

The GUI integrates with the game engine and touches no other components. Within the main game loop, the board is rendered after each turn is played. The transitions will be animated and flow with user action so that the re-render is a seamless transition.

GUI event handling happens within the client.GUI package and is transferred to engine.Engine.

5.3 License

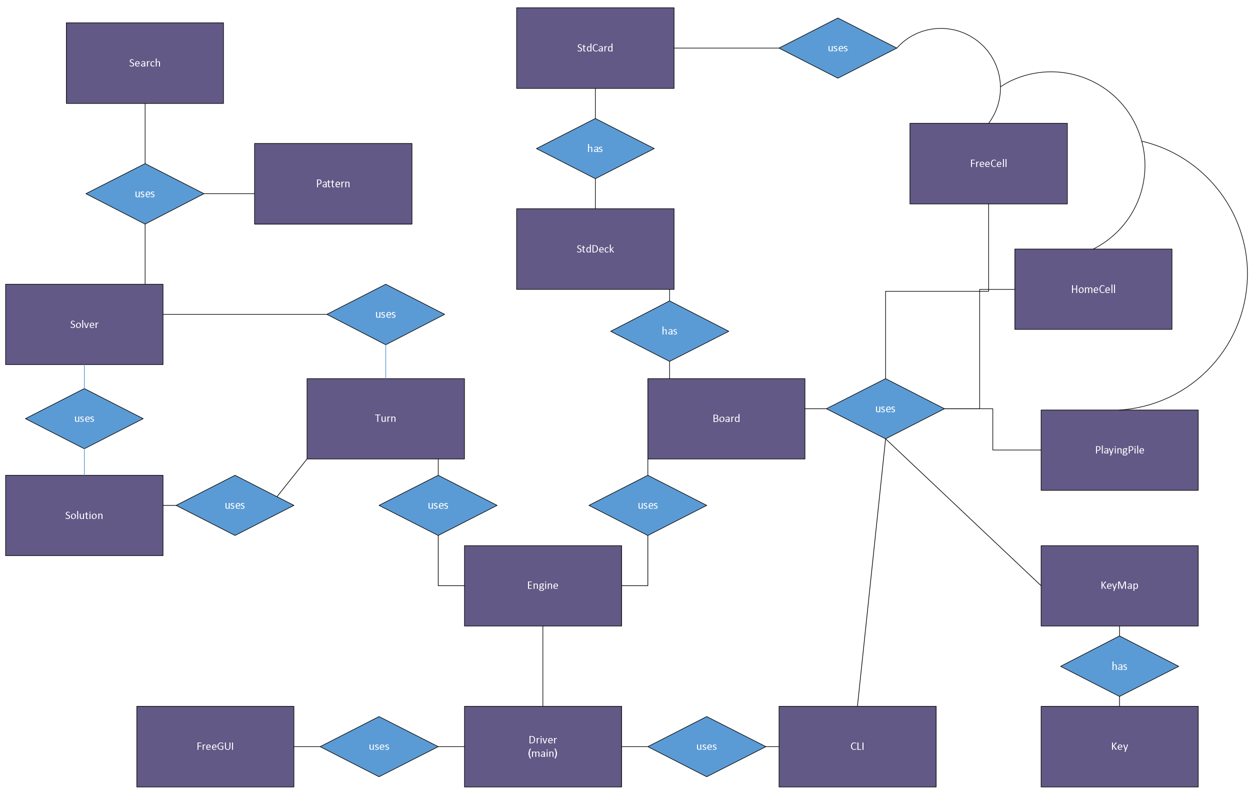
GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

https://www.gnu.org/licenses/gpl-3.0.txt

6. DIAGRAMS

6.1 Relational Diagram



6.2 UML Class Diagrams

Legend

+ public
- private
# protected

package.client

Driver

Methods

main(String[] arg) : void

package.client.cli

CLI

Attributes

- scan : Scanner obj
- dbgStr : String
- prompt : String
- debug : boolean

Methods

+ prompt( ) : void
+ inGame( ) : String
- printHelp( ) : void
+ cliInstructions( ) : void
- credz( ) : void
- formFeed( ) : void
- toggleDebug( ) : void

Tester

Methods

+ enter( ) : void
- testAll( ) : void

Debugger

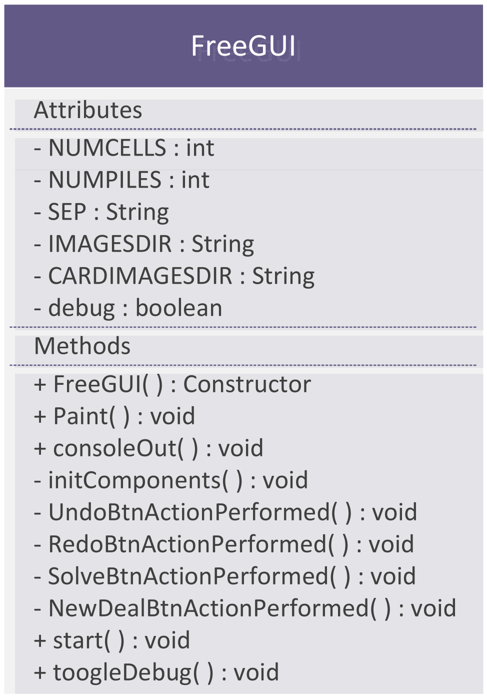
Attributes

- debug : boolean

Methods

+ masterToggleDebug( ) : void

package.client.gui



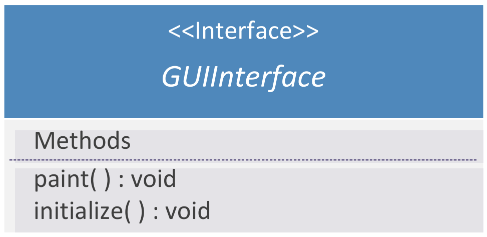
FreeGUI

Attributes

- NUMCELLS : int
- NUMPILES : int
- SEP : String
- IMAGESDIR : String
- CARDIMAGESDIR : String
- debug : boolean

Methods

+ FreeGUI( ) : Constructor
+ Paint( ) : void
+ consoleOut( ) : void
- initComponents( ) : void
- UndoBtnActionPerformed( ) : void
- RedoBtnActionPerformed( ) : void
- SolveBtnActionPerformed( ) : void
- NewDealBtnActionPerformed( ) : void
+ start( ) : void
+ toogleDebug( ) : void



<<Interface>>

GUIInterface

Methods

paint( ) : void
initialize( ) : void

package.engine

Engine

Attributes

- isGui : boolean
- gameOver : boolean
- curBoard : Board
- history : Stack<Board>
- moveNum : int
- src : String
- dest : String
- autoStack : boolean
- gui : FreeGUI

Methods

+ start( ) : void
-memberName
- checkUiMode( ) : FreeGUI
- gameLoop( ) : void
- getSourceCLI( ) : String
- getDestCLI( ) : String
+ getMapping( ) : void
+ doubleClick( ) : void
+ dragDrop( ) : void
+ autoStack( ) : void
+ newDeal( ) : void
+ undo( ) : void
+ redo( ) : void
+ hint( ) : void
+ solve( ) : void
+ snapshot( ) : void
+ printSnapshot( ) : void
- checkGameOver( ) : boolean
+ toggleAutoStack( ) : void

Turn

Attributes

- winnable : boolean
- moveNum : int
- board : Board
- solution : Solution
- keymap : KeyMap

Methods

+ Turn( ) : Constructor
- isWinnable( ) : boolean
+ getWinnable( ) : boolean
+ getMoveNum( ) : int
+ toString( ) : String

package.board

Board

Attributes

- CELLS : int
- PILES : int
- winnable : boolean
- moveNum : int
- d : StdDeck
- freeAry : FreeCell[ ]
- homeAry : HomeCell[ ]
- pileAry : PlayingPile[ ]

Methods

```
+ Board( ) : Constructor
+ getFreecells( ) : FreeCell[ ]
+ getHomecells( ) : HomeCell[ ]
+ getCardAt( ) : StdCard
+ getPiles( ) : PlayingPile[ ]
+ getPile( ) : PlayingPile
- init( ) : void
- dealRow( ) : void
+ makeMove( ) : boolean
+ place( ) : void
+ remove( ) : void
+ updateBoardStats( ) : void
- intoFreecell( ) : boolean
- intoHomecell( ) : boolean
- intoPlayingPile( ) : boolean
+ toString( ) : String <<override>>
- buildCellsCLI( ) : String
- buildRowCLI( ) : String
+ winCheck( ) : boolean
+ autoStack( ) : Queue<KeyMap>
- genMap( ) : void
- maxPileSize( ) : int
```

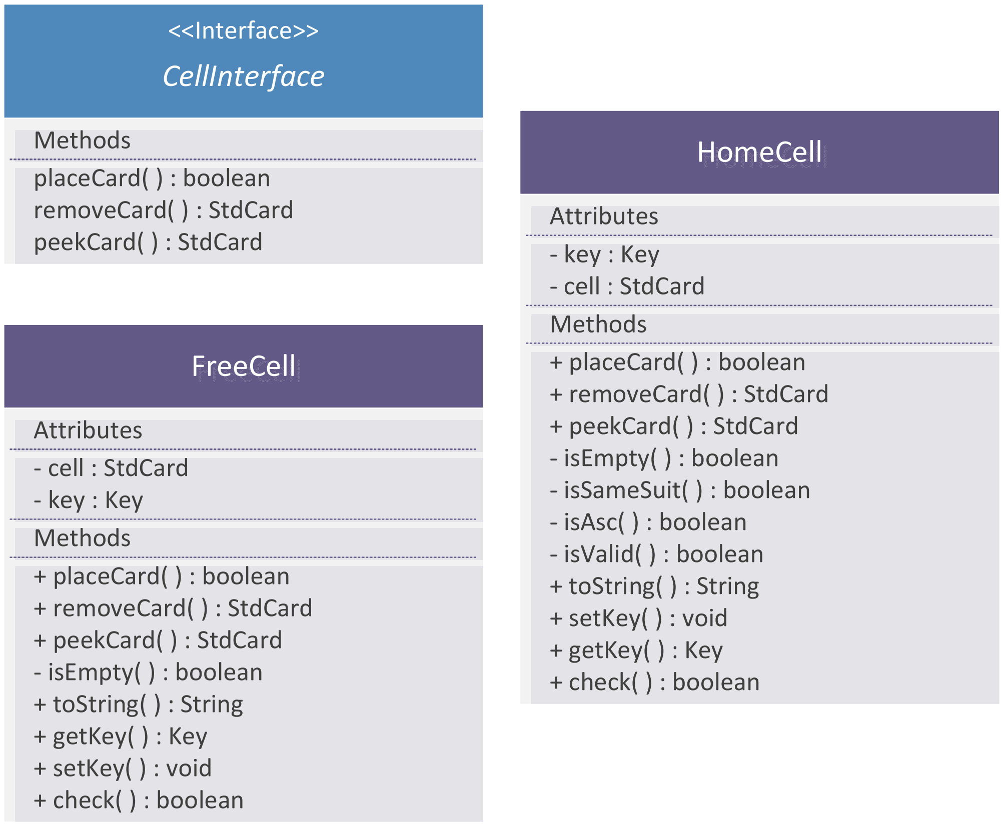
PlayingPile

Attributes

```
- debug  : boolean
- pile : ArrayList<StdCard>
- key : Key
```

Methods

```
+ PlayingPile( ) : Constructor
- placeCardOnDeal( ) : void
- placeCard( ) : boolean
- removeCard( ) : StdCard
+ getCardAt( ) : StdCard
+ getKey( ) : Key
+ setKey( ) : void
+ peekLastCard( ) : StdCard
+ check( ) : boolean
+ size( ) : int
- isEmpty( ) : boolean
- isDsc( ) : boolean
- isAltColor( ) : boolean
- isValid( ) : boolean
+ toString( ) : String
```



<<Interface>>
CellInterface

Methods
placeCard( ) : boolean
removeCard( ) : StdCard
peekCard( ) : StdCard

FreeCell

Attributes
- cell : StdCard
- key : Key

Methods
+ placeCard( ) : boolean
+ removeCard( ) : StdCard
+ peekCard( ) : StdCard
- isEmpty( ) : boolean
+ toString( ) : String
+ getKey( ) : Key
+ setKey( ) : void
+ check( ) : boolean

HomeCell

Attributes
- key : Key
- cell : StdCard

Methods
+ placeCard( ) : boolean
+ removeCard( ) : StdCard
+ peekCard( ) : StdCard
- isEmpty( ) : boolean
- isSameSuit( ) : boolean
- isAsc( ) : boolean
- isValid( ) : boolean
+ toString( ) : String
+ setKey( ) : void
+ getKey( ) : Key
+ check( ) : boolean

<<Enumeration>>

Key

A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

Attributes

- KEY : String
- REGION : int
- POSITION : int

Methods

+ Key( ) : Constructor
+ getKey( ) : String
+ getRegion( ) : int
+ isFreeCell( ) : boolean
+ isHomeCell( ) : boolean
+ isPlayingPile( ) : boolean
+ equals( ) : boolean

KeyMap

Attributes

+ src : Key
+ dest : Key
+ srcCard : StdCard
+ board : Board

Methods

+ KeyMap( ) : Constructor
  + KeyMap( ) : Overload Constructor
- genKeys( ) : void
- genCard( ) : void
+ getSourceCard( ) : StdCard
+ getSrcKey( ) : Key
+ getDestKey( ) : Key
+ isValid( ) : boolean

package.PlayingCards

<<Interface>>
CardInterface

Methods
+ getRank( ) : int
+ getSuit( ) : int

<<Interface>>
DeckInterface

Methods
+ shuffle( ) : void
+ getCard( ) : StdCard
+ print( ) : void

StdCard

Attributes
- rankAry : String[ ]
- suitAry : String[ ]
- defSymAry : Character[ ]
- uniSymAry : Character[ ]
- rank : int
- suit : int
- debug : boolean
- unicode : boolean

Methods
+ StdCard( ) : Constructor
    + StdCard( ) : Overload Constructor
- setRank( ) : void
- setSuit( ) : void
+ toggleUnicode( ) : void
+ getRank( ) : int
+ getValue( ) : int
+ getSuit( ) : int
+ getRankString( ) : String
+ getSuitString( ) : String
+ getDefSym( ) : String
+ getUniSym( ) : String
+ getName( ) : String
+ toString( ) : String
+ isUnicode( ) : boolean
+ isBlack( ) : boolean
+ toggleDebug( ) : void

DeckException

Methods
+ DeckException( ) : Constructor

StdDeck

Attributes
- SUITS : int
- RANKS : int
- SIZE : int
- ERR1 : String
- deck : StdCard[ ]
- deckCount : int
- debug : boolean

Methods
- StdDeck( ) : Constructor
+ size( ) : int
+ getDeckCount( ) : int
+ getCard( ) : StdCard
+ isEmpty( ) : boolean
- init( ) : void
+ shuffle( ) : void
+ print( ) : void
+ toggleUnicode( ) : void
+ unitTest( ) : void
+ toggleDebug( ) : void

package.solver

Search

Pattern

Solver

Attributes

- possibleMoves : KeyMap[ ]
- board : Board
- movePossible : boolean
- solution : Solution

Methods

+ Solver( ) : Constructor
+ moveIsPossible( ) : boolean
+ getSolution( ) : Solution
- setMovePossible( ) : void
- init( ) : void
+ findMoves( ) : void

package.utils

SysUtils

Attributes

- SEP : String
- PATH : String

Methods

+ isWindows( ) : boolean
+ getPath( ) : String
+ getSeparator( ) : String
+ exitDoor( ) : void